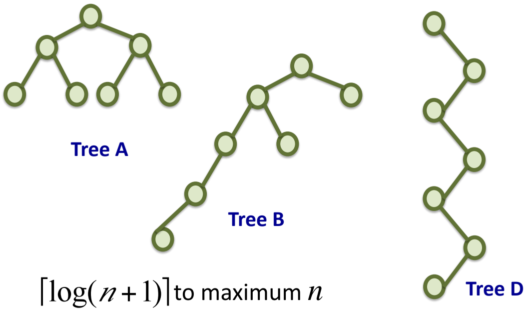
Part 6.
Trees (2)

CS 200 Algorithms and Data Structures

Outline

- **2-3 Trees**
- 2-3-4 Trees
- Red-Black Trees
- AVL Trees

Balanced Search Trees



$\lceil \log(n+1) \rceil$ to maximum n

Balanced Search Trees

- A search of a binary search tree can be as inefficient as a sequential search of a linked list.
  - Balanced search trees address this problem
- Insert and delete items *without* deteriorating the tree's <u>balance</u> while maintaining a minimum-height search tree.
- A type of binary search tree where costs are ***guaranteed*** *to be logarithmic*

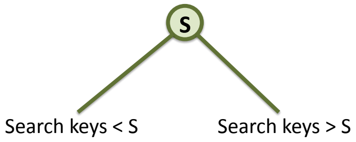
2-3 search trees

- A tree in which each internal node (nonleaf) has either **two or three children**, and all leaves are at the same level.
- A 2-3 search tree is not a binary tree.

Rules for placing items

- A 2-node (with two children) must contain a single data item whose search key is greater than the left child's search key(s) and less than the right child's search key(s)
- A 3-node (with three children) must contain two data items whose search keys S and L satisfy the following relationships.
  - S is greater than the left child's search key(s) and less than the middle child's search key(s).
  - L is greater than the middle child's search key(s) and less than the right child's search key(s).
- A leaf may contain either one or two data items.

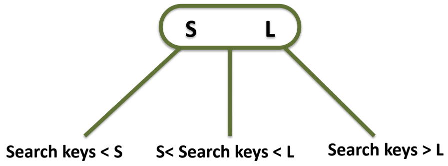
Placing items in a 2-node



Search keys < S

Search keys > S

Placing items in a 3-nodes



Search keys < S

S< Search keys < L

Search keys > L

Traversing a 2-3 tree

```
inorder (in ttTree:TwoThreeTree)
  if (ttTree’s root node r is a leaf){
      visit the data item(s)
  }


  else if (r has two data items){
      inorder(left subtree of ttTree’s root)
      Visit the first data item
      inorder(middle subtree of ttTree’s root)
      Visit the second data item
      Inorder(right subtree of ttTree’s root)
  }
  else {// r has one data item
      inorder(left subtree of ttTree’s root)
      visit the data item
      inorder(right subtree of ttTree’s root)
  }
```

Searching a 2-3 tree (1/2)

```
retrieveItem(in ttTree:TwoThreeTree,
             in searchKey:KeyType):TreeItemType
   if(searchKey is in ttTree’s root node r){
       treeItem = the data portion of r
   } else if (r is a leaf){
       treeItem = null;
   }

   // else search has the appropriate subtree
   else if (r has two data items){
       if (searchKey < smaller search key of r){
          treeItem = retrieveItem(r’s left subtree, searchKey)
       }else if (searchKey < larger search key of r){
          treeItem = retrieveItem(r’s middle subtree, searchKey)
       }else {
          treeITem = retrieveItem(r’s right subtree, searchKey)
       }
   }
}
```

Searching a 2-3 tree (2/2)

```
  // r has only one data item
  else {
       if (searchKey < r’s searckKey){
         treeItem = retrieveItem(r’s left subtree,
                                 searchKey)
      }else {
         treeITem = retrieveItem(r’s right subtree,
                                 searchKey)
      }
  }
```

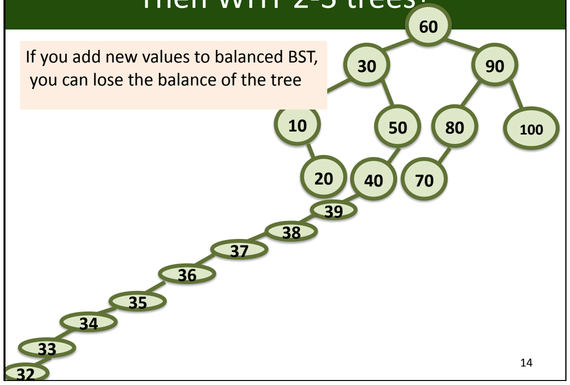
Efficiency

- A binary search tree with n nodes cannot be shorter than $\lceil \log(n+1) \rceil$
- A 2-3 tree with n nodes cannot be taller than $\lceil \log(n+1) \rceil$
- A node in a 2-3 tree has at most two items.

Is searching a 2-3 tree is more efficient than a BST?

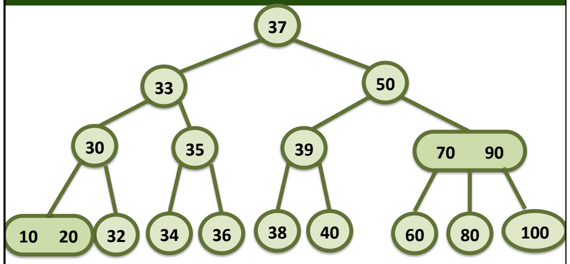
- After all, the nodes of a 2-3 tree can have three children
  - **Shorter than the shortest possible binary search tree!**
- More comparisons for each of the node. (twice the number of comparisons)
  - **Approximately equal to the number of comparisons** in BST that is **as balanced as possible.**

Then WHY 2-3 trees?



If you add new values to balanced BST, you can lose the balance of the tree

Then WHY 2-3 trees?

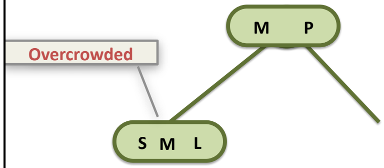


The 2-3 tree algorithm will keep the balance of the tree

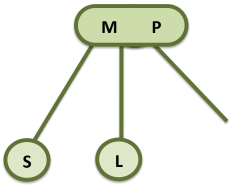
The insertion algorithm

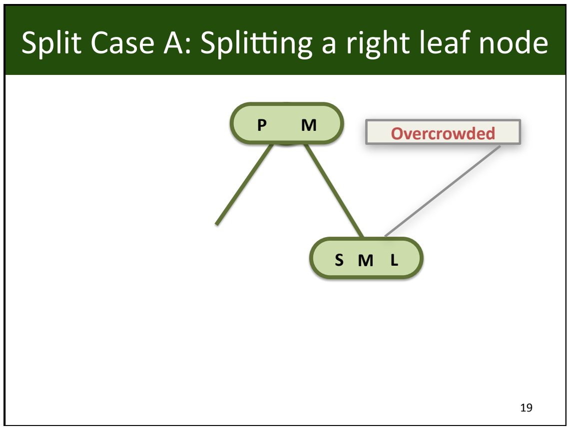
- **Locate the leaf at which the search for *I* would terminate.**
- Insert the new item into the leaf ***I***
  - **Case 1. If the leaf *I* contains two items: you are done**
  - **Case 2. If the leaf *I* contains three items: must split into n1 and n2.**
    - **Split Case A: Split a *leaf node***
    - **Split Case B: Split an *internal node***
    - **Split Case C: Split a root node**

Split Case A: Splitting a left leaf node



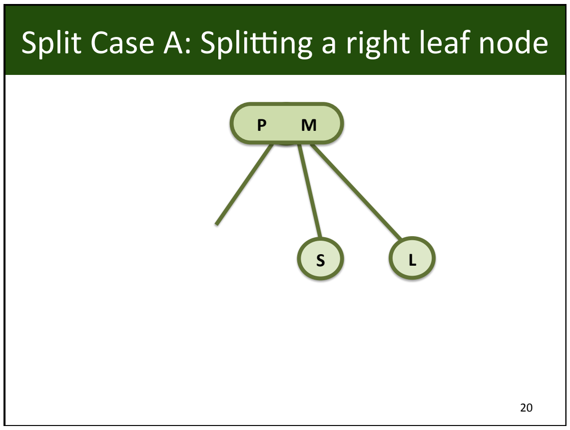
Split Case A: Splitting a left leaf node





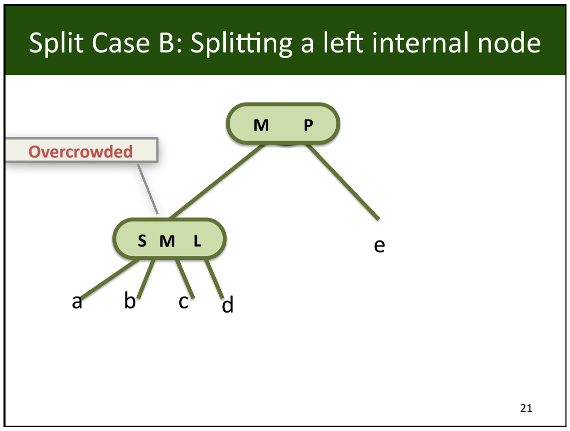
Split Case A: Splitting a right leaf node

19



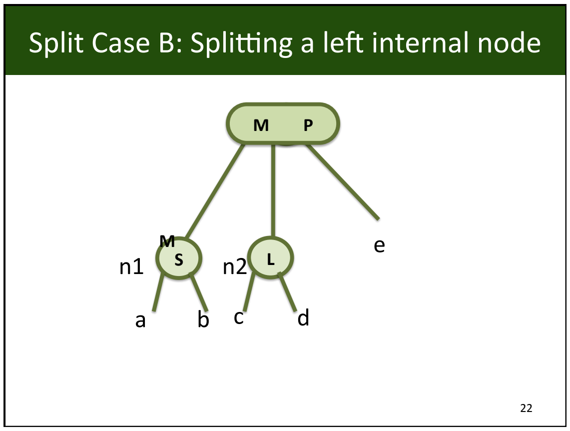
Split Case A: Splitting a right leaf node

20



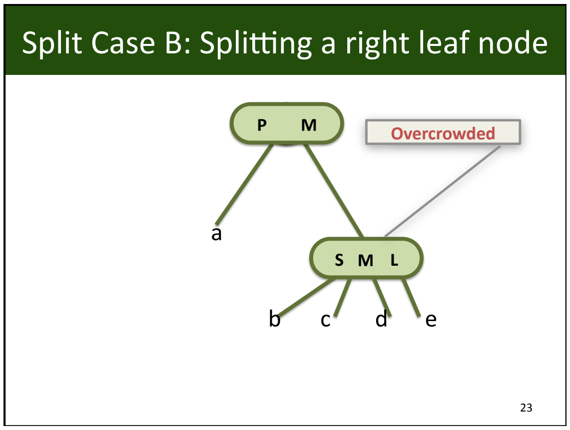
Split Case B: Splitting a left internal node

21



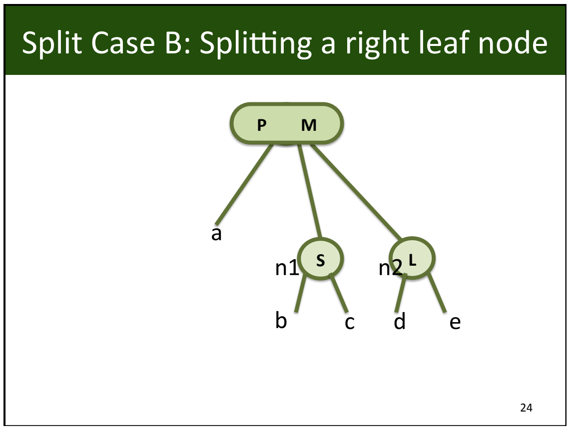
Split Case B: Splitting a left internal node

22



Split Case B: Splitting a right leaf node

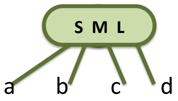
23



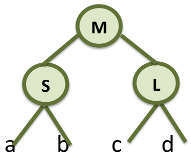
Split Case B: Splitting a right leaf node

24

Split Case C: Splitting the root of a 2-3 tree



Split Case C: Splitting the root of a 2-3 tree



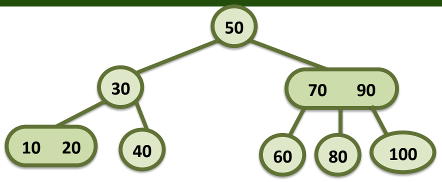
Growing the heights

- If every node on the path from the root of the tree to the leaf (into which the new item is inserted) contains two items.
- The recursive process of splitting a node and moving an item up to the node's parent will reach the root r
- Split r into r1 and r2
- Create a new node r with a middle item
- The new node becomes a new root of the tree

Example: Inserting into a 2-3 tree

- You can insert items into the tree while maintaining its shape.
- Insert 39, 38, 37, 36, 35, 34, 33, 32

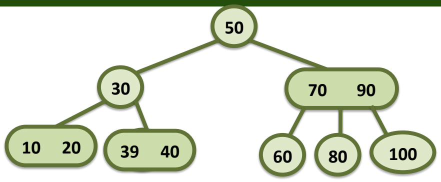
Insert 39



Case 1. node I contains two items

Finished

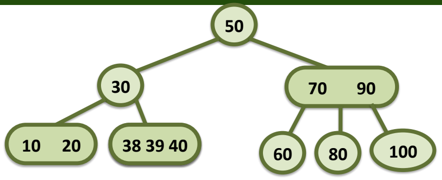
Insert 39



Case 1. node I contains two items

Finished

Insert 38

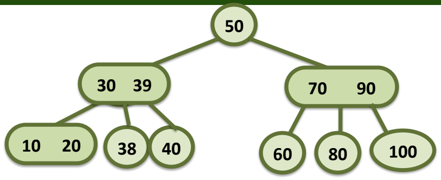


Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Finished

Insert 38

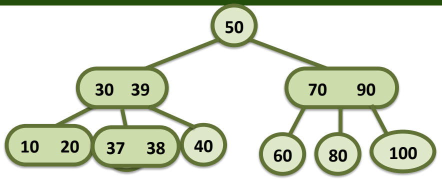


Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Finished

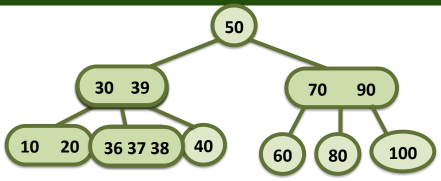
Insert 37



Case 1. 37 is added to a leaf node and the leaf node has two items

Finished

Insert 36 (1/2)



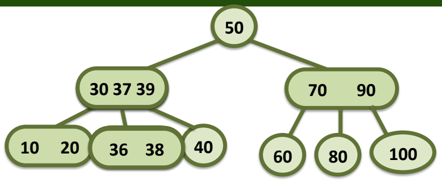
Case 2. Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2. Internal node became overcrowded

→ Split case B: Internal node: Move up the middle one

Insert 36 (1/2)



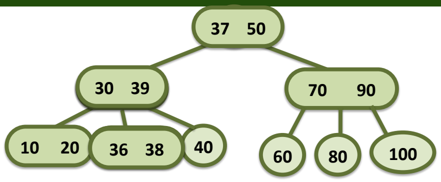
Case 2. Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2. Internal node became overcrowded

→ Split case B: Internal node: Move up the middle one

Insert 36 (1/2)

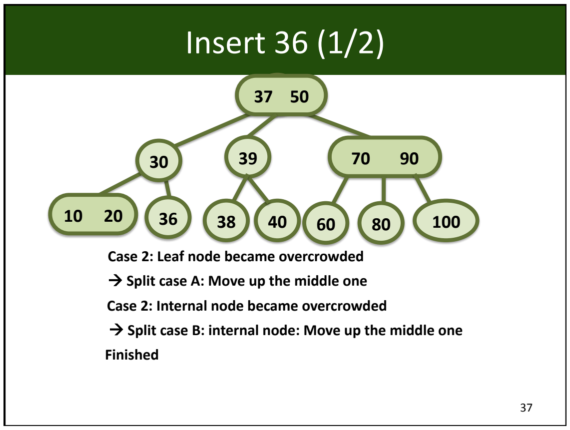


Case 2. Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2. Internal node became overcrowded

→ Split case B: Internal node: Move up the middle one



Insert 36 (1/2)

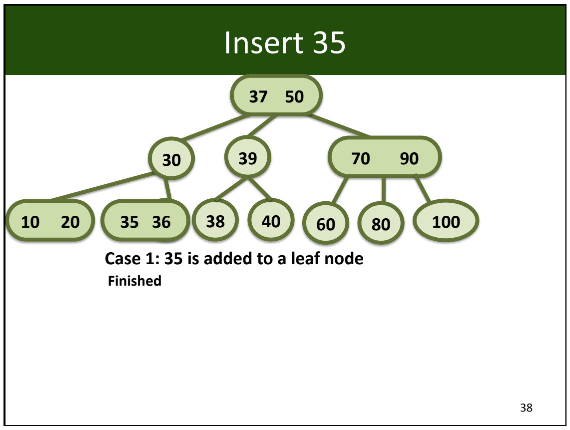
Case 2: Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split case B: internal node: Move up the middle one

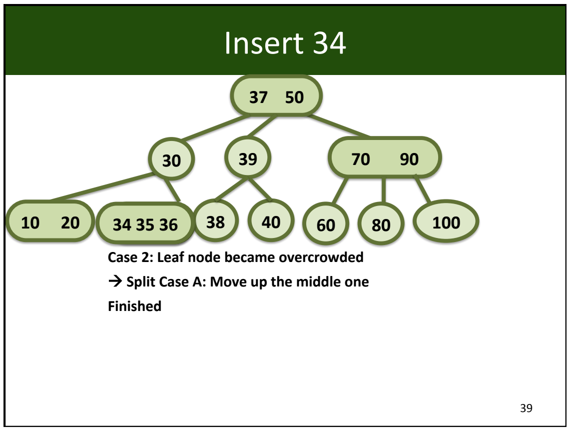
Finished



Insert 35

Case 1: 35 is added to a leaf node

Finished

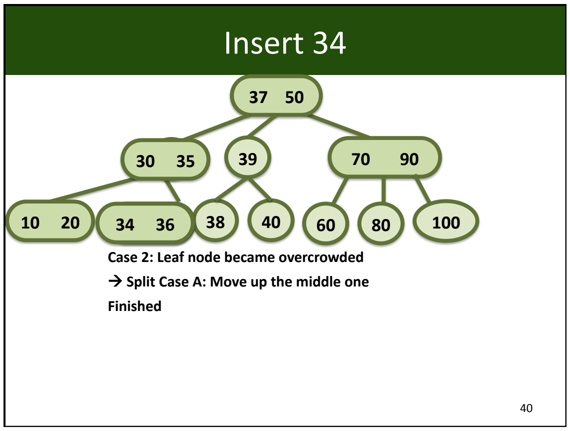


Insert 34

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Finished

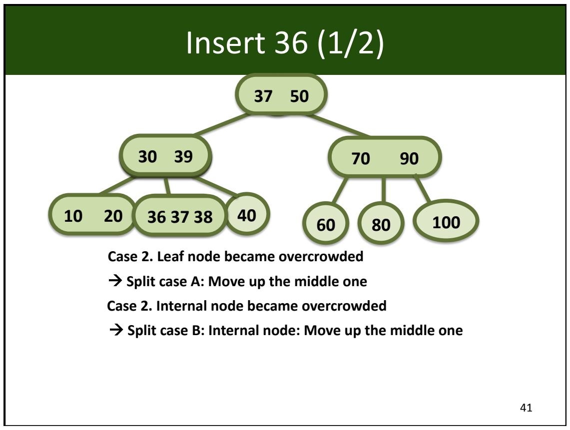


Insert 34

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Finished



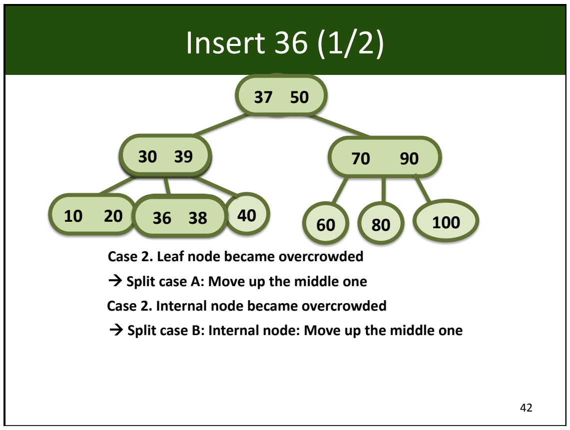
Insert 36 (1/2)

Case 2. Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2. Internal node became overcrowded

→ Split case B: Internal node: Move up the middle one



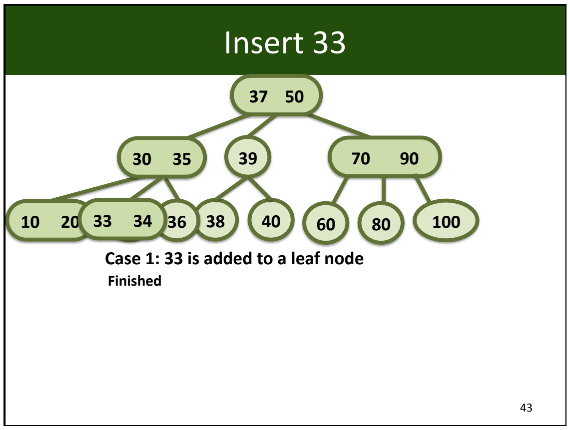
Insert 36 (1/2)

Case 2. Leaf node became overcrowded

→ Split case A: Move up the middle one

Case 2. Internal node became overcrowded

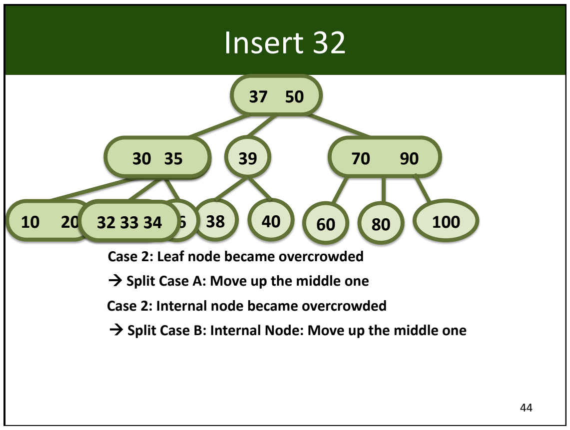
→ Split case B: Internal node: Move up the middle one



Insert 33

Case 1: 33 is added to a leaf node

Finished



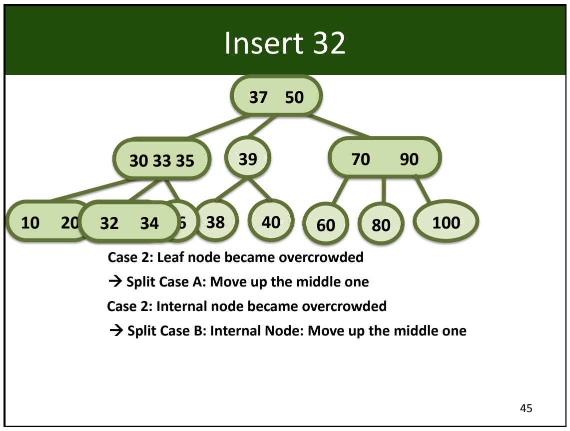
Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one



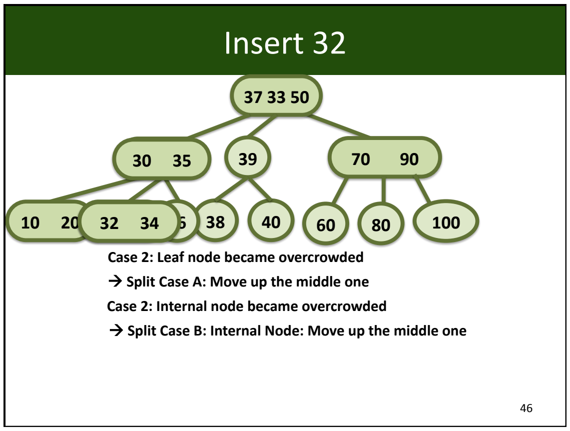
Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one



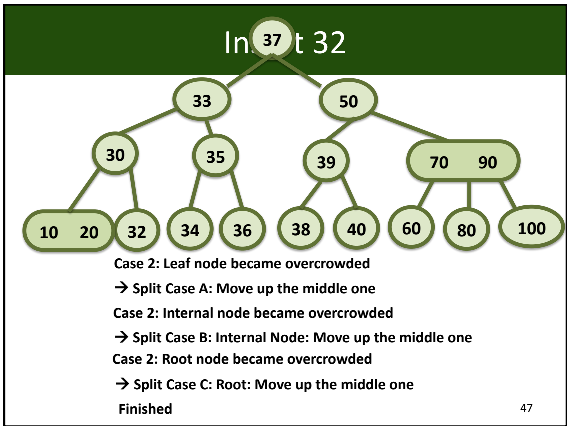
Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one



In t 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Case 2: Root node became overcrowded

→ Split Case C: Root: Move up the middle one

Finished

insertItem(2/2)

Split Case B: split an internal node

```
 if (n is not a leaf){
    n1 becomes the parent of n’s two leftmost
children
    n2 becomes the parent of n’s two right most
children
}

Move the item in n that has the middle search-key
value up to p

if (p now has three items){
   split(p)
 }
```

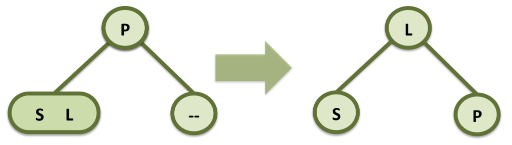
Recursive Method

Deletion Algorithm

- **Locate the node n**
- Case 1. Is the node a leaf node?
- Case 2. Is the node an internal node?
  - Find inorder successor and swap it
    - Deletion will be in the leaf now.
- Fix case A.
  - If an item will be left in the node: done
- Fix case B.
    - A. If sibling has two items: redistributing values
    - B. If no sibling has two items: merging a leaf

Deletion begins at a leaf

A. Redistributing Values: If sibling has two items



insertItem(1/2)

```
insertItem(in ttTree, in newItem
  Let sKey be the search key of newItem.
  Locate the leaf leafNode in which sKey belongs
  Add newItem to leafNode
  if (leafNode now has three items){
       split(leafNode)
  }

split (inout n:TreeNode)
  if (n is the root){
       Create a new node p
  }
  else{
       Let p be the parent of n
  }
  Replace node n with two nodes, n1 and n2, so that p is
  their parent
   Give n1 the item in n with the smallest search-key value
   Give n2 the item in n with the largest search-key value
```

First step: Locate the leafnode l

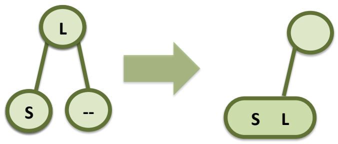
Case 2:Leafnode l has 3 items

Case 1:Leafnode l has 2 items

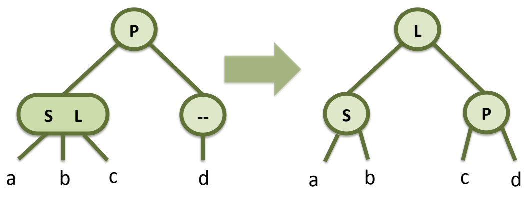
Split Case C: split a root node

Split Case A,B: split a leaf/internal node

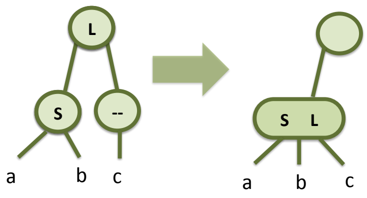
B. Merging a Leaf : If no sibling has two items

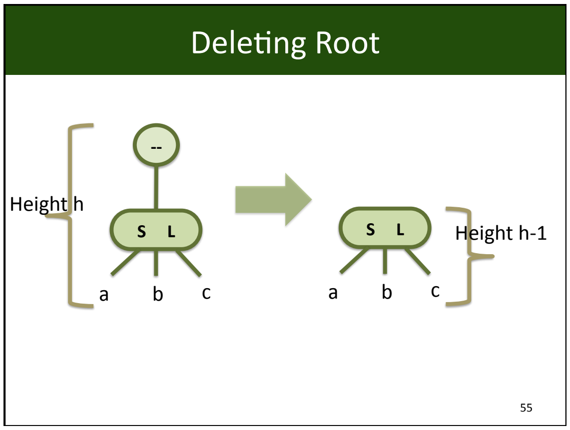


Redistributing values and children

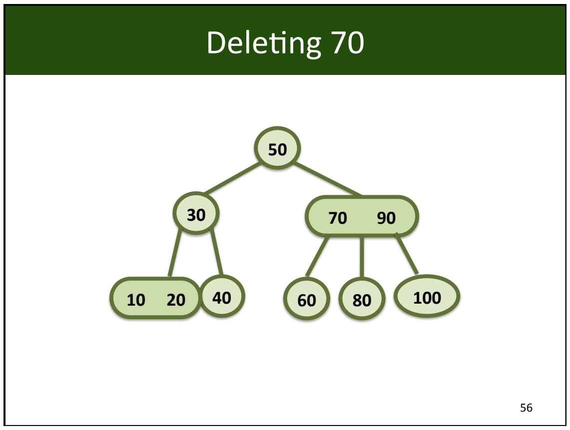


Merging internal nodes

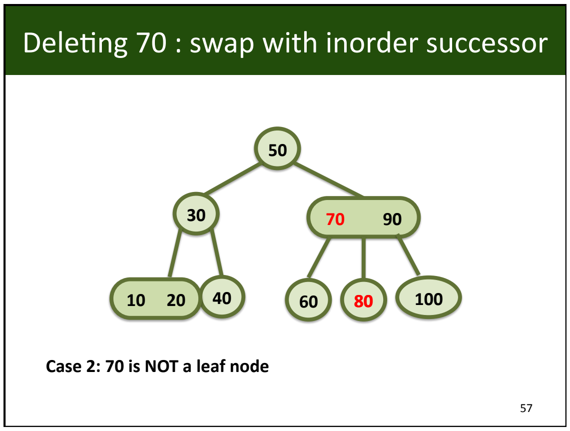




Deleting Root

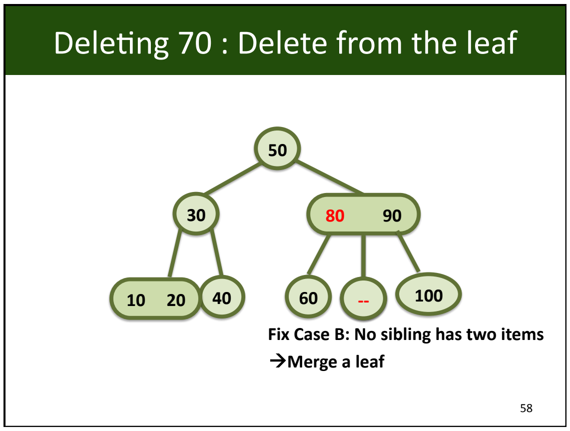


Deleting 70



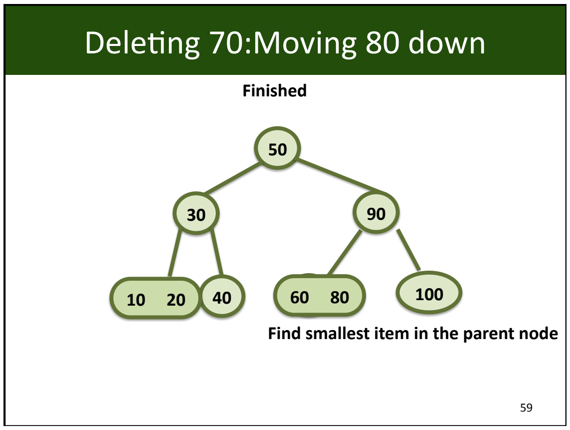
Deleting 70 : swap with inorder successor

Case 2: 70 is NOT a leaf node



Deleting 70 : Delete from the leaf

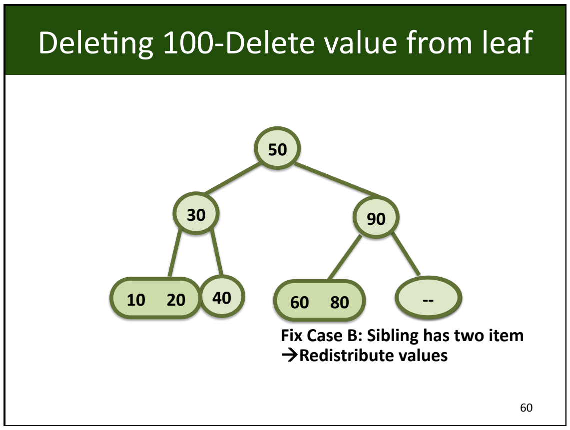
Fix Case B: No sibling has two items
→Merge a leaf



Deleting 70:Moving 80 down

Finished

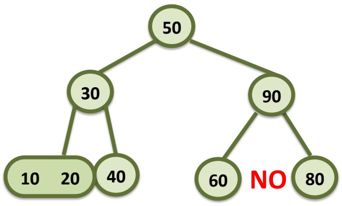
Find smallest item in the parent node



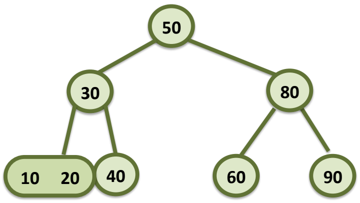
Deleting 100-Delete value from leaf

Fix Case B: Sibling has two item
→Redistribute values

Deleting 100-Does it work?

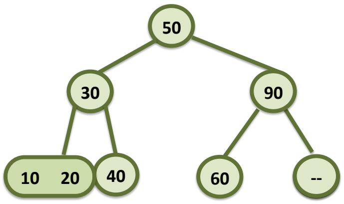


Deleting 80: swap with inorder successor



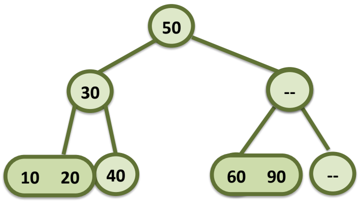
Case 2: 80 is NOT a leaf node
→ Swap with inorder successor

Deleting 80: Delete value from leaf



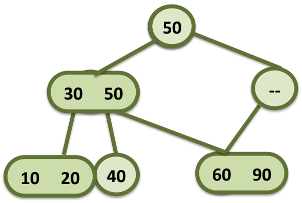
Fix Case 2: No sibling has two items
→Merge a leaf

Deleting 80: Merge by moving 90 down



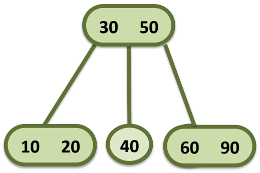
Recursively called
Fix Case 2: No sibling has two items
→Merge an internal node (with children)

Deleting 80: Merge 50 down



After Deleting 80

Finished



High level algorithm (1/2)

```
deleteItem(in ttTree:TwoThreeTree in searchKey)
   Attempt to locate item theItem whose search key equals searchKey
   if (theItem is present){
        if (theItem is not in a leaf){
              Swap item the Item with its inorder     successor, which
                will be in a leaf theLeaf
        }
        Delete item theItem from leaf theLeaf
        if (theLeaf now has no items){
            fix(theLeaf)
        }
        return true
   }
   else{
        return false
   }
```

Find the node

Case 2: Is the node an Internal node?

Recursive call

High level algorithm (1/2)

```
fix(in n:TreeNode)
   if (n is the root){
    Remove the root
   }else{
        Let p be the parent of n
        if (some sibling of n has two items){
             Distribute items appropriately among n,
                the sibling, and p
            if (n is internal){
                Move the appropriate child from sibling to n
            }
        }
        else {
            Choose an adjacent sibling s of n
            Bring the appropriate item down from p into s
            if (n is internal) {
                Move n’s child to s
            }
            remove node n
            if (p is now empty){
                fix(p)
            }
        }
   }
```

Fix Case A

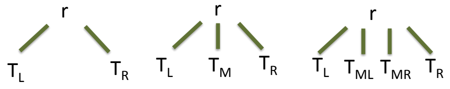
Fix Case B

Outline

- 2-3 Trees
- **2-3-4 Trees**
- Red-Black Trees
- AVL Trees

2-3-4 Trees

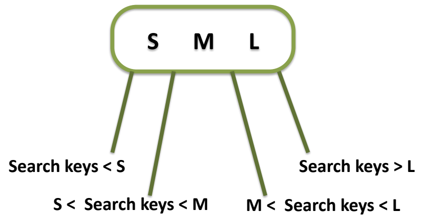
- 2-nodes, 3-nodes, and 4-nodes
  - 4-nodes: nodes that have four children
- T is a 2-3-4 tree of height h if
  - T is empty
  - T is of the form



Rules for Placing Data Items in the Nodes of a 2-3-4 Tree

- A 2-node must contain a single data item whose search key satisfies the relationship in a 2-3 Tree
- A 3-node must contain two data items whose search keys satisfy the relationship in a 2-3 Tree
- A 4-node must contain three data items, whose search keys **S,M, and L** satisfy the following relationship:
  1. **left child’s search key(s) < S < middle-left child’s search key(s)**
  2. **middle-left child’s search key(s) < M < middle-right child’s search key(s)**
  3. **middle-right child’s search key(s) < L < right child’s search key(s)**

A 4-node in a 2-3-4 tree



Searching and traversing a 2-3-4 tree

- Simple extension of the corresponding algorithms for a 2-3 tree
- Adding comparisons for the 4-node

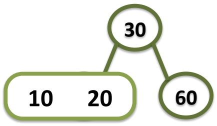
Inserting into a 2-3-4 tree

- Algorithm is similar to the insertion into a 2-3 tree.
  - 2-3 tree: Split a node by moving one of its items up to its parent node.
  - 2-3-4 tree: As soon as the search process encounters 4-nodes, it splits the 4-node.

Insert 20

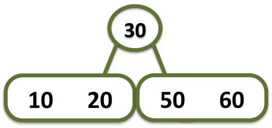
While determining the insertion point, you encounter the 4-node
Split by moving the middle value 30 up
Keep searching
Add 20

Insert 20

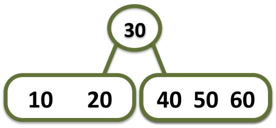


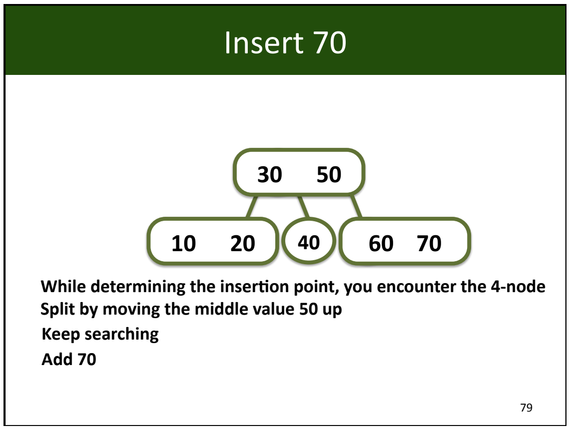
While determining the insertion point, you encounter the 4-node
Split by moving the middle value 30 up
Keep searching
Add 20

Insert 50



Insert 40





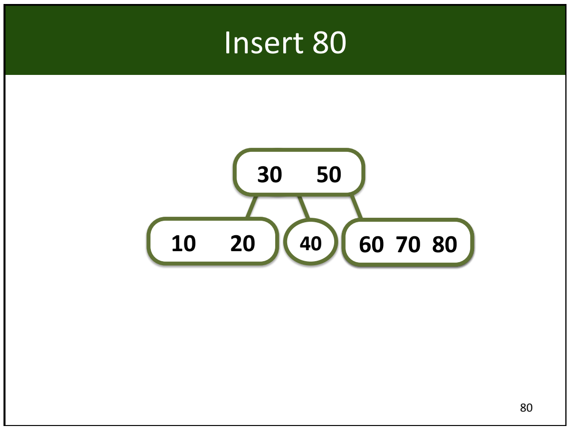
Insert 70

While determining the insertion point, you encounter the 4-node

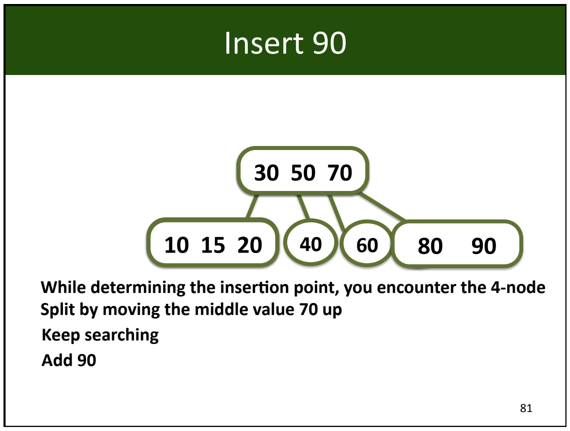
Split by moving the middle value 50 up

Keep searching

Add 70



Insert 80



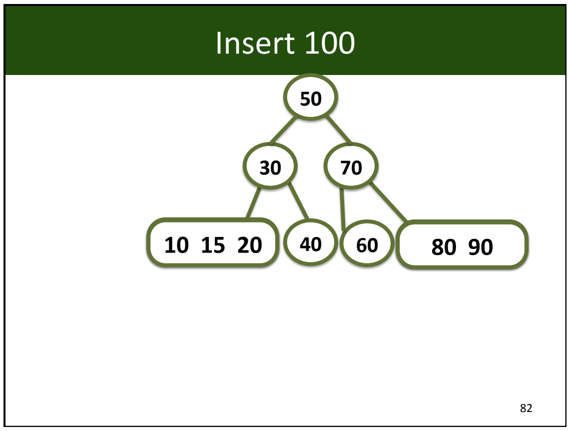
Insert 90

While determining the insertion point, you encounter the 4-node

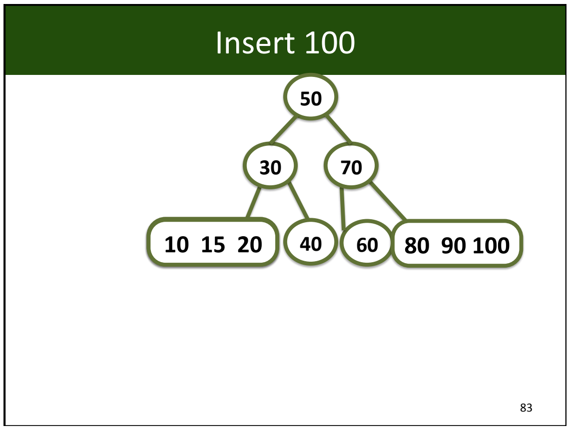
Split by moving the middle value 70 up

Keep searching

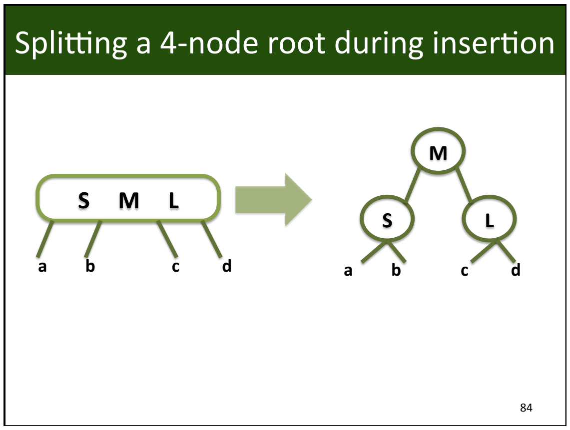
Add 90



Insert 100

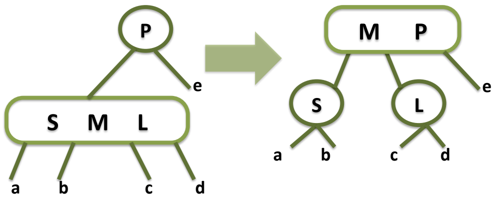


Insert 100

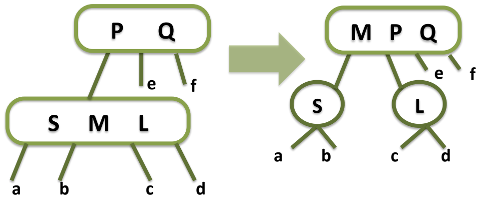


Splitting a 4-node root during insertion

Splitting a 4-node whose parent is a 2-node during insertion



Splitting a 4-node whose parent is a 3-node during insertion



Deleting from 2-3-4 tree

- Locate the node
- Swap with inorder successor
  - Deletion should be always in the leaf node
- If the leaf is a 3-node or 4-node, remove item
- If you ensure that the item you delete does not occur in a 2-node, you can delete the item in one pass through the tree from root to leaf

2-3 and 2-3-4 trees

- 2-3 and 2-3-4 trees are easy-to-maintain in balance
  - The reduction in height is offset by the increased number of comparisons that the search algorithm may require at each node.
  - The 2-3-4 tree needs **only one pass** through the tree for its insertion and deletion.

Trees with **MANY** children nodes?

- Tree with many child nodes (e.g. 100 children) requires more comparisons at each node to determine which subtree to search.
  - It is appropriate for external storage.
    - Moving from node to node is far more expensive than comparing the data values.

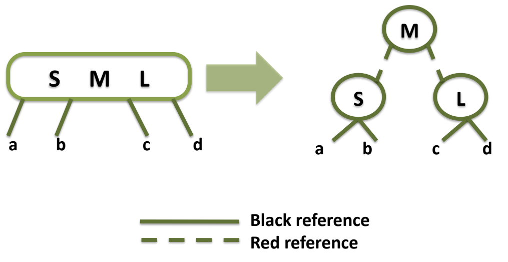
Outline

- 2-3 Trees
- 2-3-4 Trees
- **Red-Black Trees**
- AVL Trees

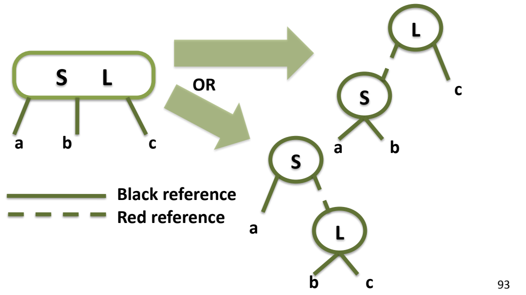
Red-Black Trees

- Represent a 2-3-4 tree and retain the advantages of a 2-3-4 tree without the storage overhead.
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree.
  - Use red and black child references to distinguish between original 2-nodes, and 2-nodes that were generated from 3-nodes and 4-nodes.
  - **2-nodes from original 2-3-4 tree : black**
  - **2-nodes those result from splitting 3 and 4-nodes : red**

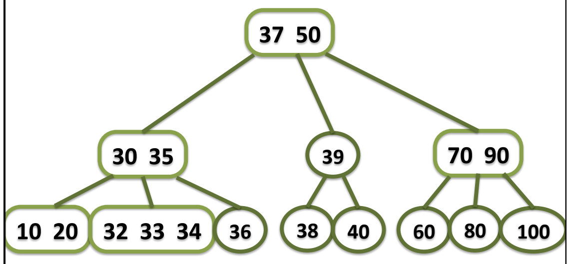
Red-black representation of a 4-node



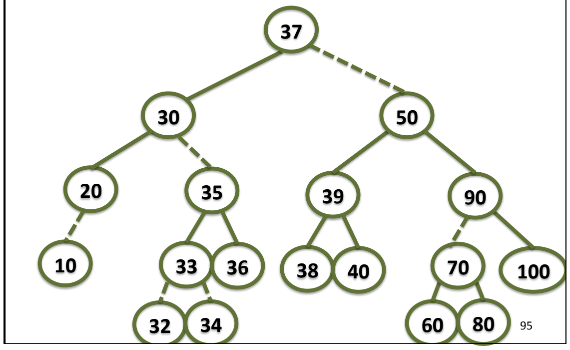
Red-black representation of a 3-node



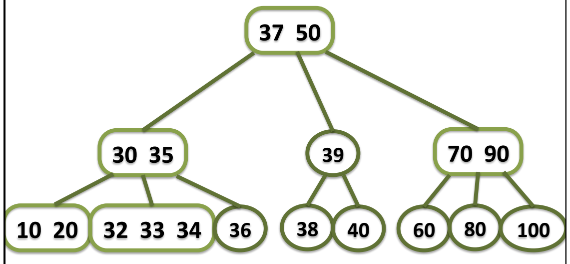
2-3-4 tree to a Red-black tree



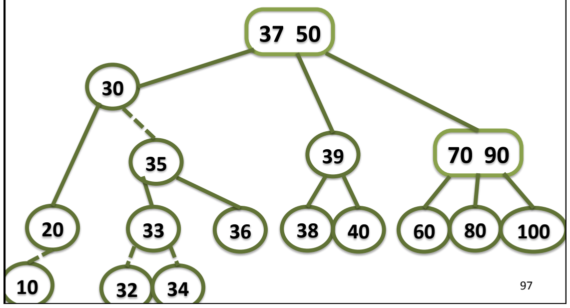
2-3-4 tree to a Red-black tree



2-3-4 tree to a Red-black tree



2-3-4 tree to a Red-black tree



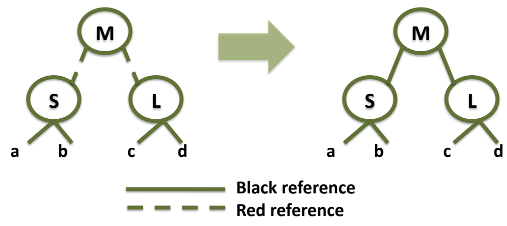
Searching and traversing a red-black tree

- A red-black tree is a binary search tree
  - Search and traverse it with binary search tree algorithms
  - Ignore the color of the references

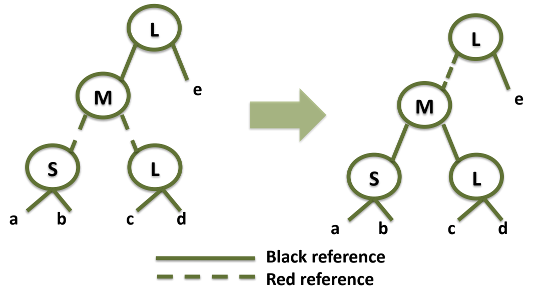
Inserting into a red-black tree

- Split each 4-node that you encounter
  - **Case 1: 4-node that is a root**
  - **Case 2: 4-node whose parent is a 2-node**
  - **Case 3: 4-node whose parent is a 3-node**
  - **There is no 4-node whose parent is a 4-node. WHY?**

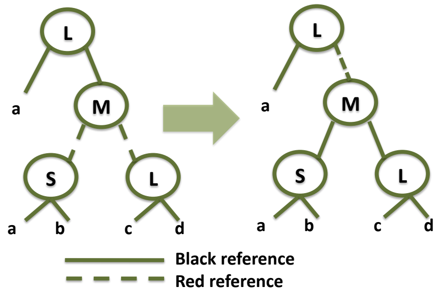
Inserting Case 1: 4-node that is a root

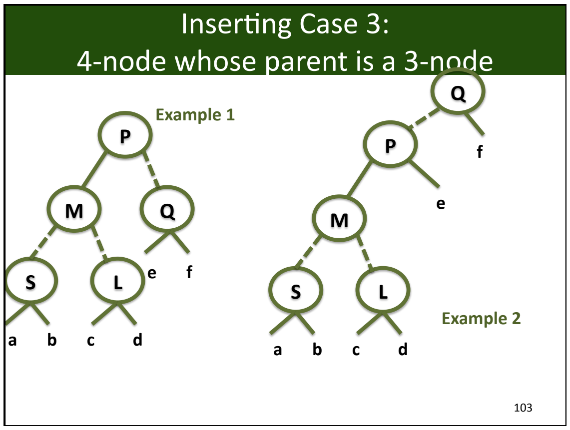


Inserting Case 2: 4-node whose parent is a 2-node



Inserting Case 2: 4-node whose parent is a 2-node



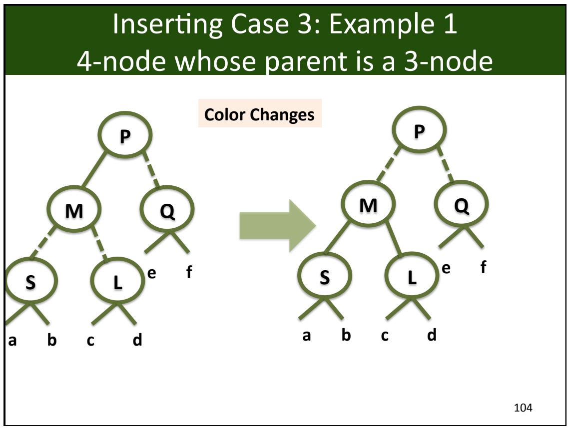


Inserting Case 3:
4-node whose parent is a 3-node

Example 1

Example 2

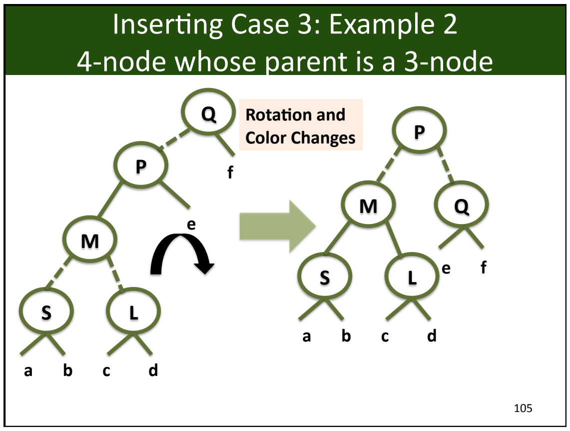
103



Inserting Case 3: Example 1
4-node whose parent is a 3-node

Color Changes

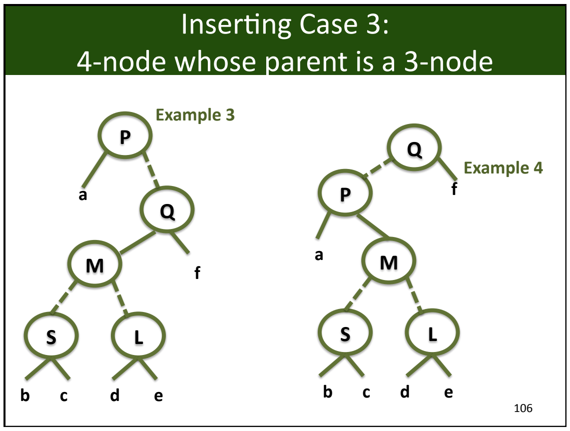
104



Inserting Case 3: Example 2
4-node whose parent is a 3-node

Rotation and
Color Changes

105

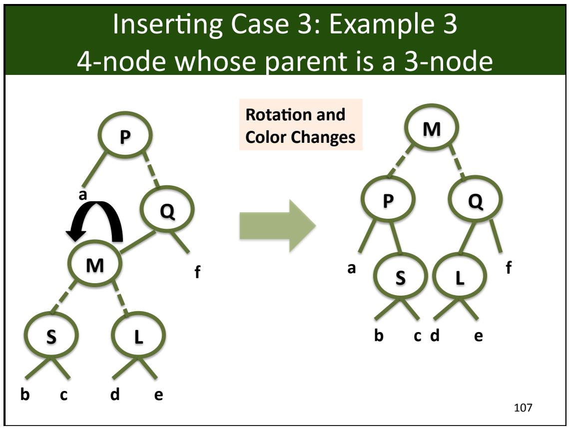


Inserting Case 3:
4-node whose parent is a 3-node

Example 3

Example 4

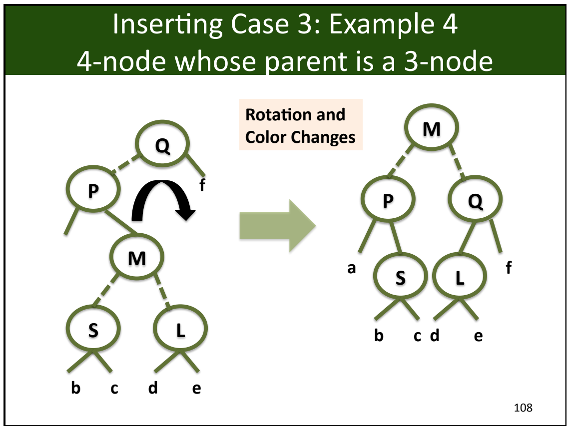
106



Inserting Case 3: Example 3
4-node whose parent is a 3-node

Rotation and
Color Changes

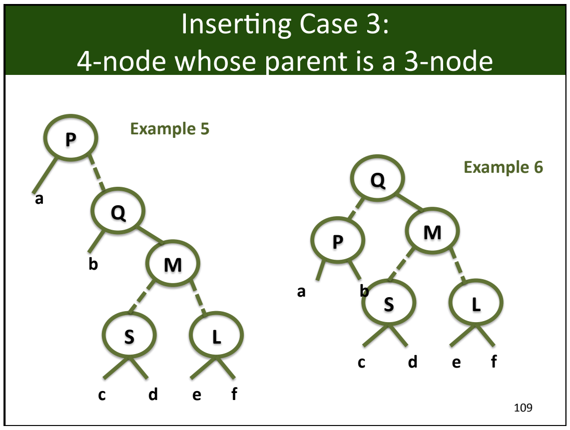
107



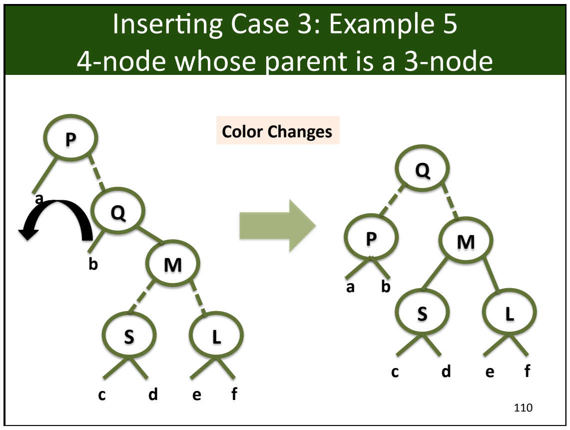
Inserting Case 3: Example 4
4-node whose parent is a 3-node

Rotation and
Color Changes

108



Inserting Case 3:
4-node whose parent is a 3-node



Inserting Case 3: Example 5
4-node whose parent is a 3-node

Deleting from a red-black tree

- This is similar to the 2-3-4 deletion algorithm
  - Frequently requires only color changes
  - More efficient than the corresponding operations on a 2-3-4 tree

Outline

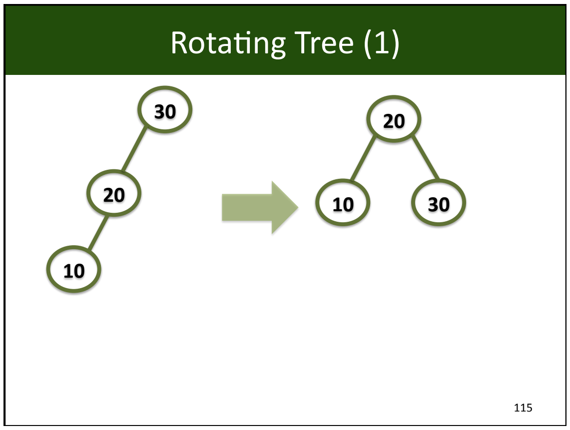
- 2-3 Trees
- 2-3-4 Trees
- Red-Black Trees
- **AVL Trees**

AVL Tree

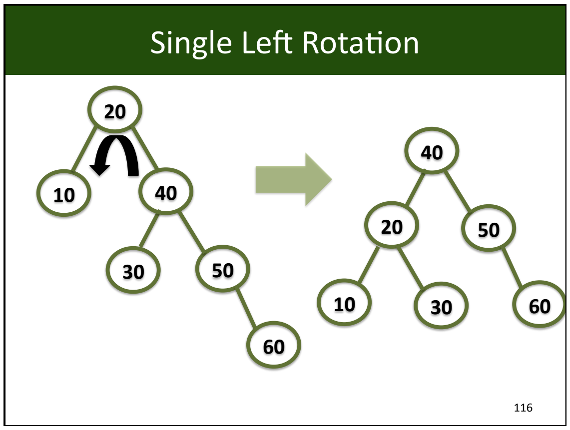
- Named after its inventors Adelson-Velskii and Landis
- A balanced binary search tree
- Almost as efficient as a minimum-height binary search tree

AVL Tree

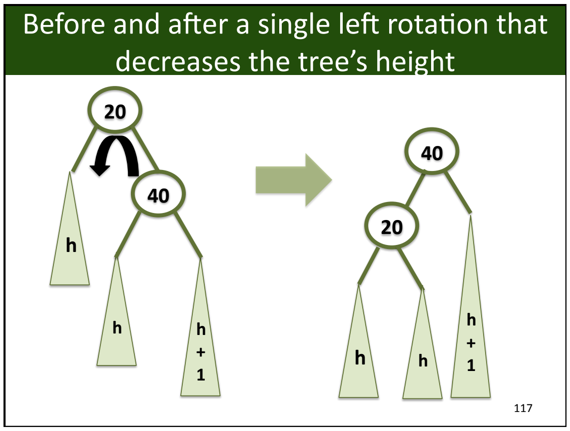
- Maintains a binary search tree with a height close to the minimum
  1. Insert/Delete nodes following the algorithm of the BST.
  2. Monitor the shape.
     - Determine whether any node in the tree has left and right subtrees whose heights differ by more than 1.
  3. If it is not a balanced binary search tree, rotate the tree to rebalance the tree



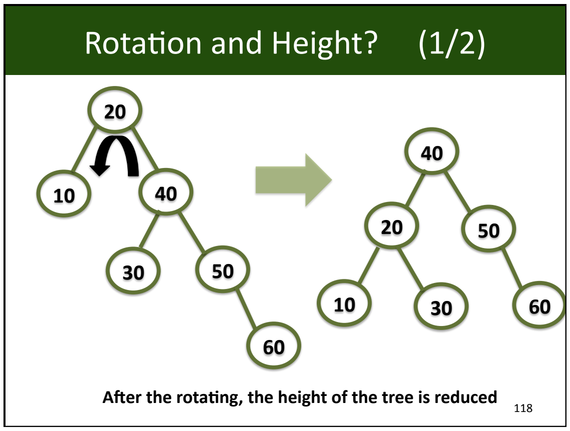
Rotating Tree (1)



Single Left Rotation

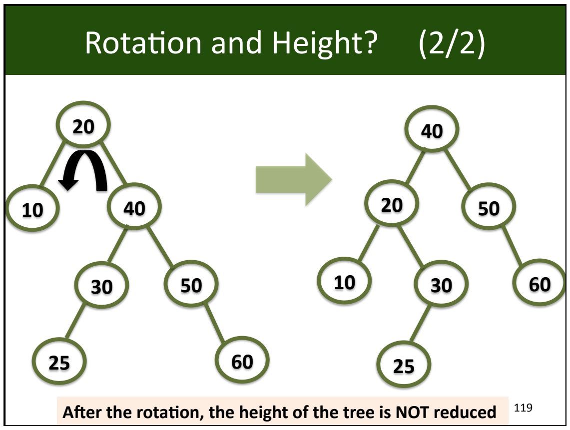


Before and after a single left rotation that decreases the tree's height



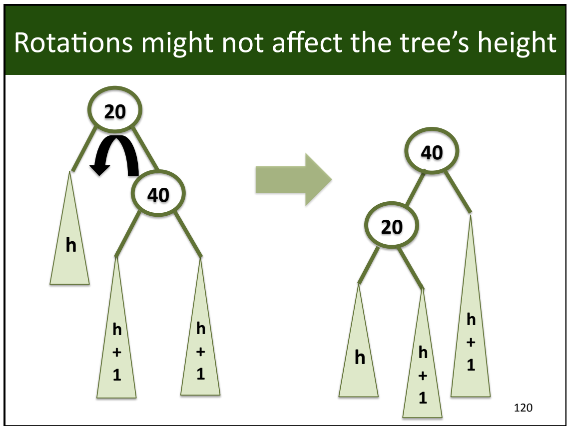
Rotation and Height? (1/2)

After the rotating, the height of the tree is reduced

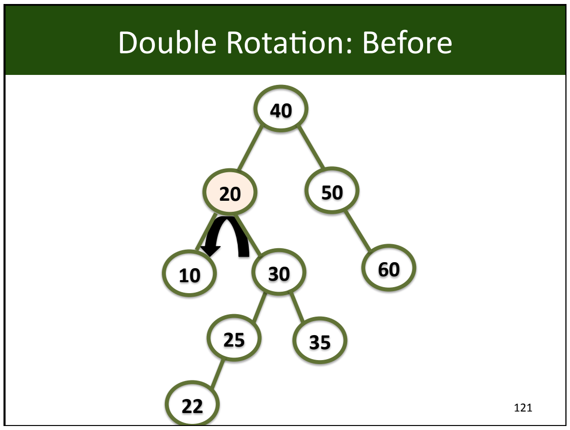


Rotation and Height? (2/2)

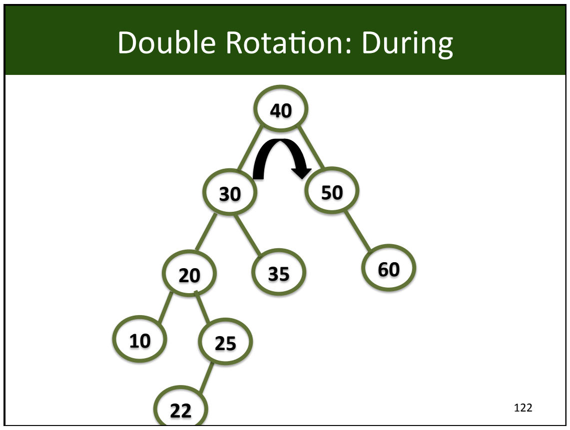
After the rotation, the height of the tree is NOT reduced



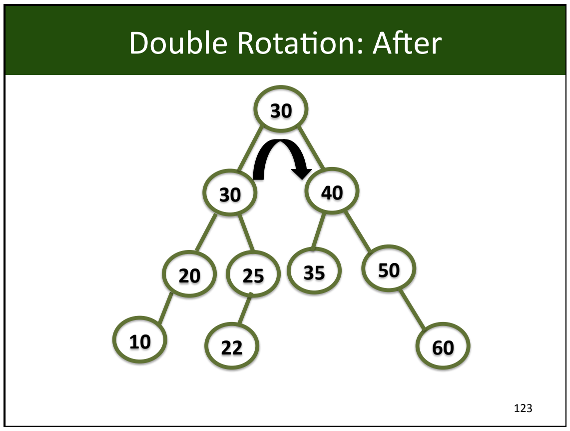
Rotations might not affect the tree's height



Double Rotation: Before



Double Rotation: During



Double Rotation: After