Balanced Trees.

A balanced life is a prefect life.

Balanced Search Trees

- The efficiency of the binary search tree implementation of the ADT table is related to the tree's height
  - Height of a binary search tree of n items
    - Maximum: n
    - Minimum: $\lceil \log_2(n + 1) \rceil$
- Height of a binary search tree is sensitive to the order of insertions and deletions
- Variations of the binary search tree
  - Can retain their balance despite insertions and deletions

2-3 Trees

- A 2-3 tree
  - Has 2-nodes and 3-nodes
    - A 2-node
      - A node with one data item and two children
    - A 3-node
      - A node with two data items and three children
  - Is not a binary tree
  - Is never taller than a minimum-height binary tree
    - A 2-3 tree with n nodes never has height greater than $\lceil \log_2(n + 1) \rceil$

2-3 Trees

- Rules for placing data items in the nodes of a 2-3 tree
  - A 2-node must contain a single data item whose search key is
    - Greater than the left child's search key(s)
    - Less than the right child's search(s)
  - A 3-node must contain two data items whose search keys S and L satisfy the following
    - S is
      - Greater than the left child's search key(s)
      - Less than the middle child's search key(s)
    - L is
      - Greater than the middle child's search key(s)
      - Less than the right child's search key(s)
  - A leaf may contain either one or two data items

2-3 Trees

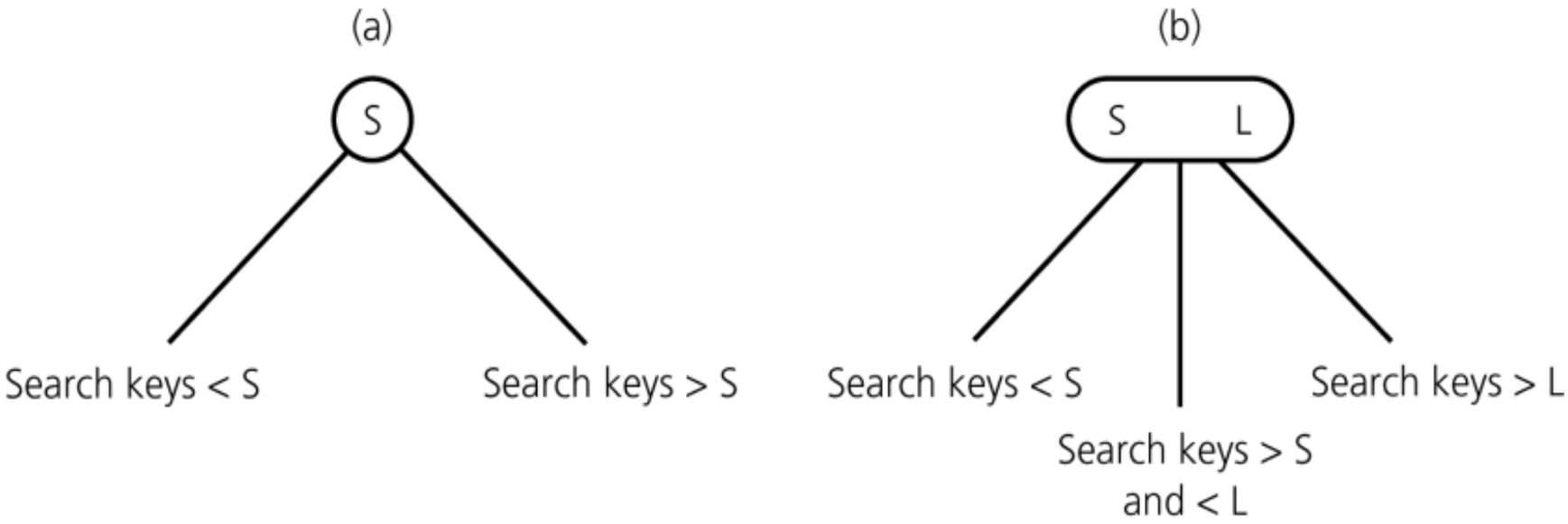
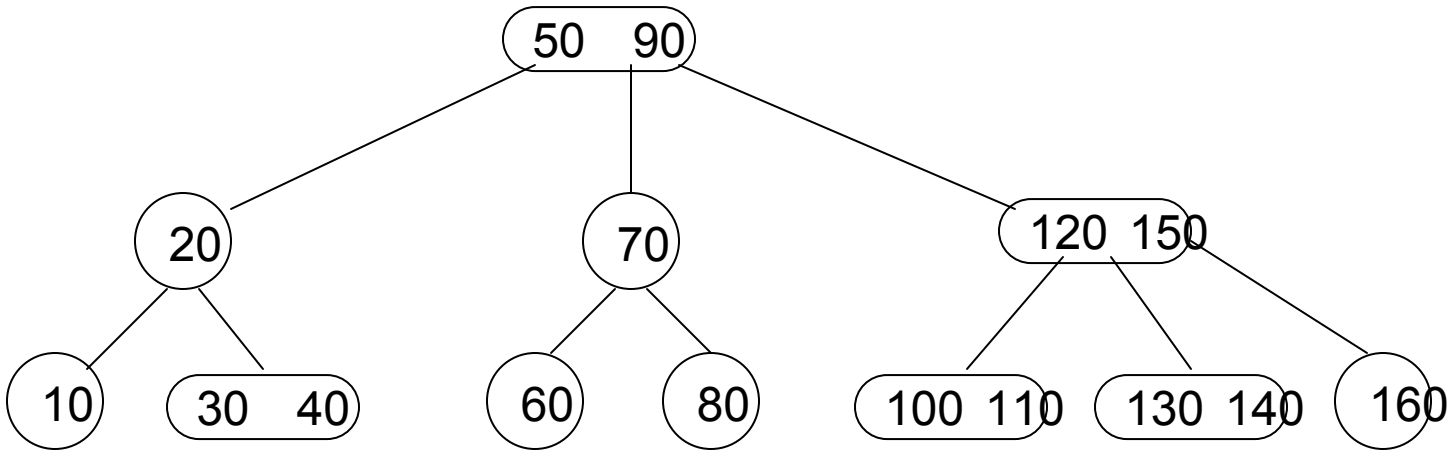


Figure 13-3

Nodes in a 2-3 tree a) a 2-node; b) a 3-node

A 2-3 tree

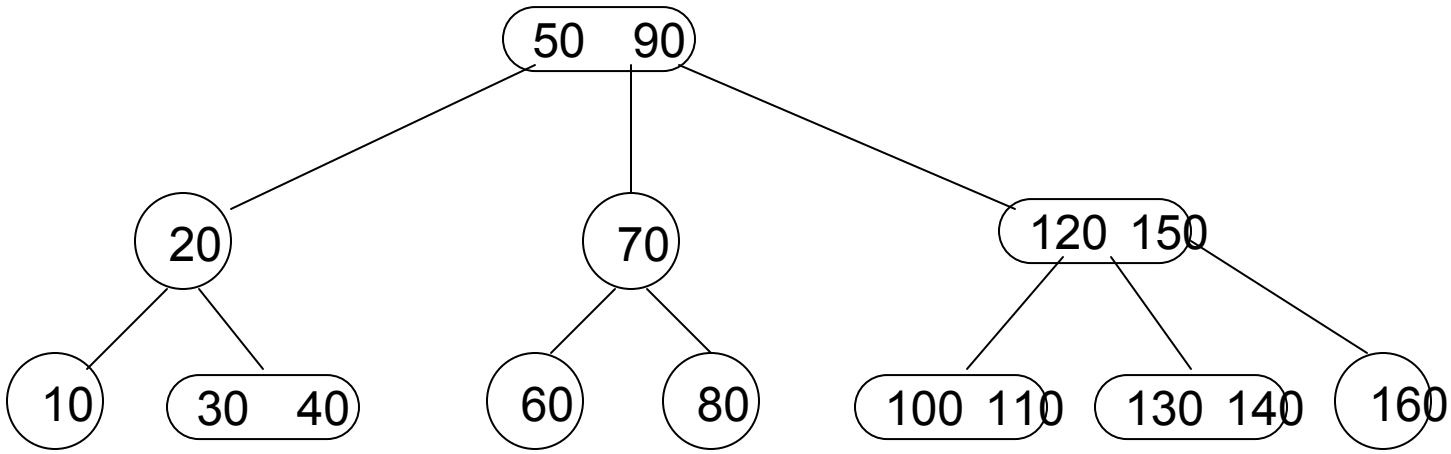


2-3 inorder traversal.

```
inorder(TwoTreeNode n){
   if (n == null)
        return;
   if (n is a 3-node){
        inorder (n.leftChild);
        visit (n.firstData);
        inorder (n.middleChild);
        visit (n.secondData);
        inorder (n.rightChild);
   }
   if (n is a 2-node){
        inorder (n.leftChild);
        visit (n.data);
        inorder (n.rightChild);

   }
}
```

A 2-3 tree



Inorder Traversal:

10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160

Searching a 2-3 tree

```
TreeNode Search( TreeNode n , item x){
    if (n == null)
          return null;
    if (n is a 2-node){
          if (n.data == x)
                    return n;
          else if ( n.data < x)
                    return n.rightChild;
          else
                    return n.leftChild;
    }
    else { //n is a 3-node
          if (n.firstData == x or n.secondData == x)
                    return n;
          if (x < n.firstData)
                    return n.leftChild;
          if (x < n.secondData)
                    return n.middleChild;
          return n.rightChild;
    }
}
```

2-3 Trees

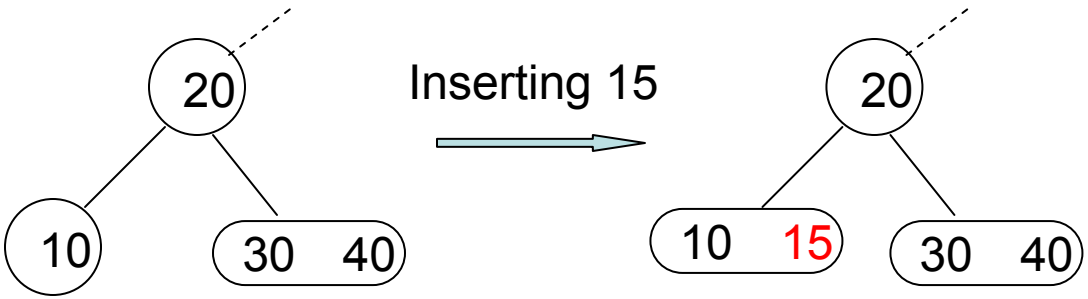
- Searching a 2-3 tree
  - Searching a 2-3 tree is as efficient as searching the shortest binary search tree
    - Searching a 2-3 tree is $O(\log_2 n)$
  - Since the height of a 2-3 tree is smaller than the height of a balanced tree the number of compared node is less than that of binary search tree.
  - However, we need to make two comparisons at the 3-nodes in a 2-3 tee.
  - Number of comparisons approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible

2-3 Trees

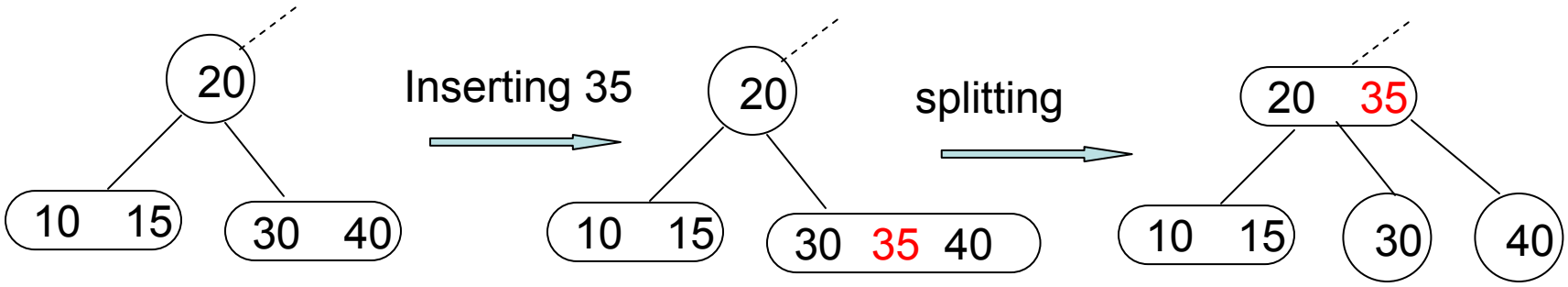
- Advantage of a 2-3 tree over a balanced binary search tree
  - Maintaining the balance of a binary search tree is difficult
  - Maintaining the balance of a 2-3 tree is relatively easy

2-3 Trees: Inserting Into a 2-3 Tree

- First we need to locate the position of the new item in the tree.
  - This is done by a search on the tree
  - The location for inserting a new item is always a leaf in 2-3 tree
- Insertion into a 2-node leaf is simple



- Insertion into a 3-node leaf splits the leaf



2-3 Trees: The Insertion Algorithm

- To insert an item I into a 2-3 tree
  - Locate the leaf at which the search for I would terminate
  - Insert the new item I into the leaf
  - If the leaf now contains only two items, you are done
  - If the leaf now contains three items, split the leaf into two nodes, n_1 and n_2

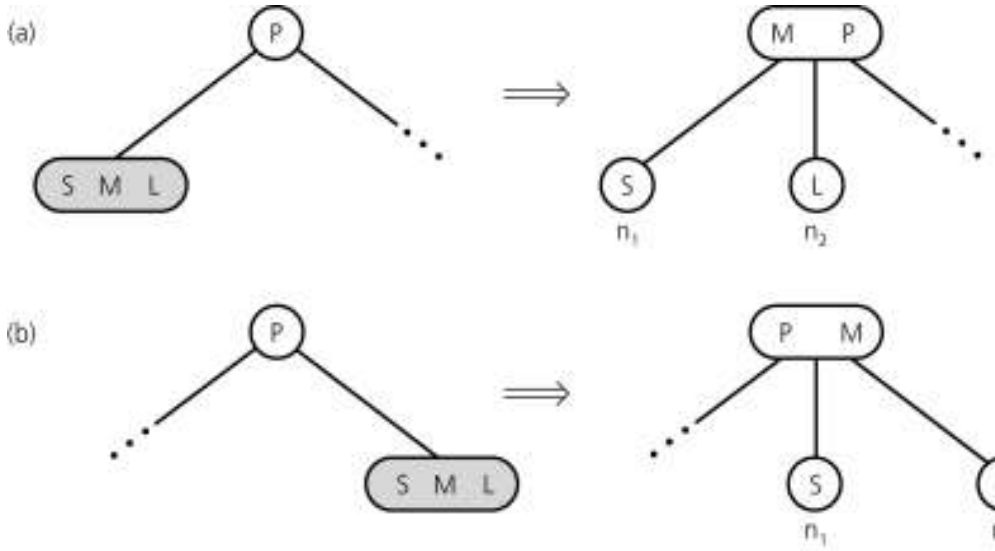
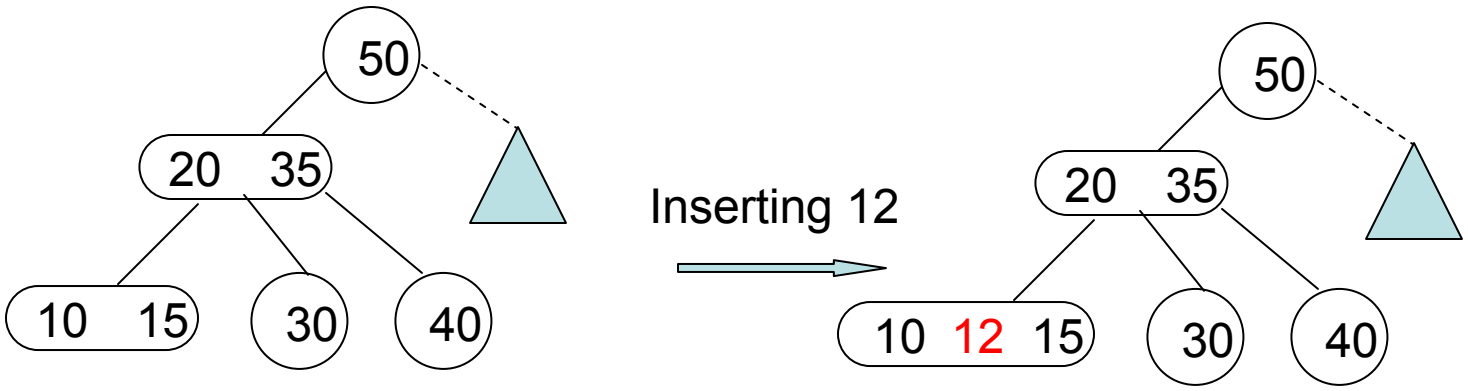


Figure 13-12

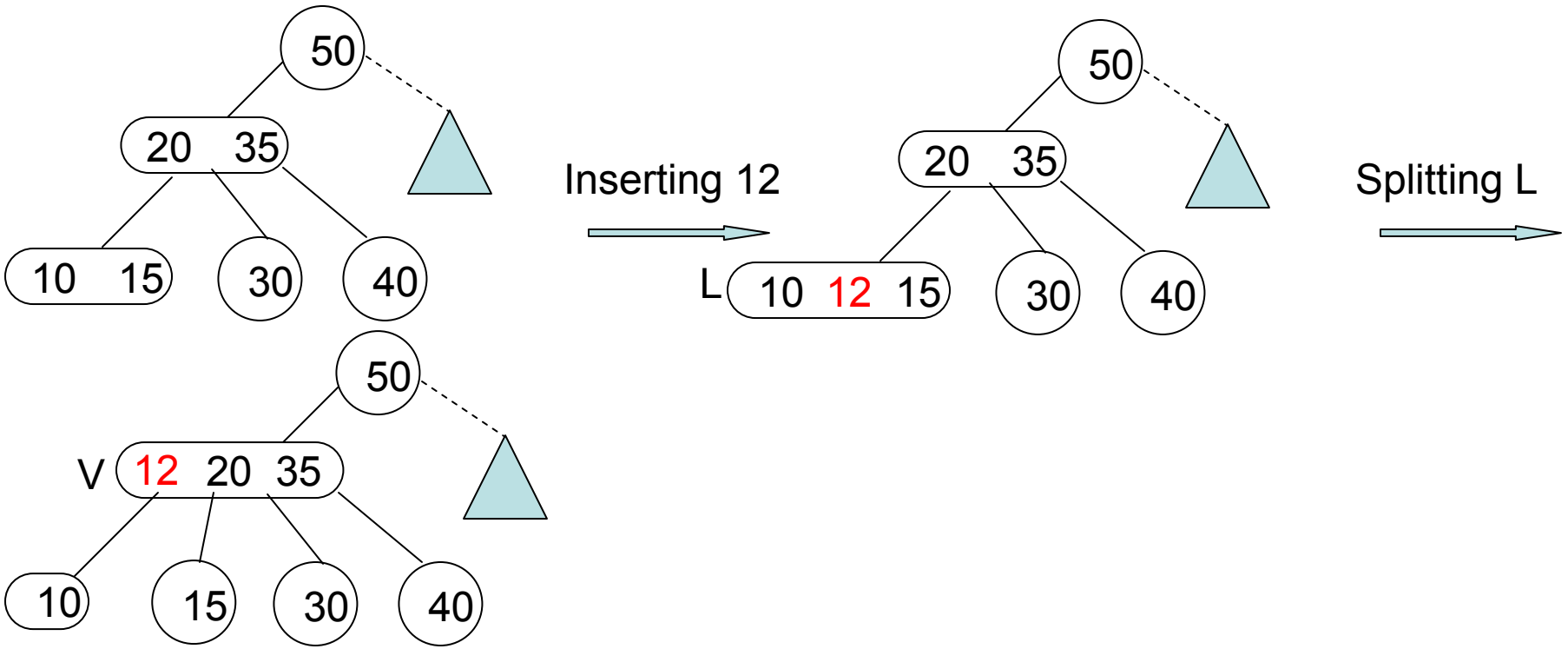
Splitting a leaf in a 2-3 tree

- When a leaf has more than 3 values we need to move the middle value to the leaf's parent and split the leaf.
- If the leaf's parent is a 2-node it simply becomes a 3-node.
- If it is a 3-node it will contain 3 values after the insertion so we need to split it as well.
- Splitting an internal 3-node is very similar to splitting a 3-node leaf.

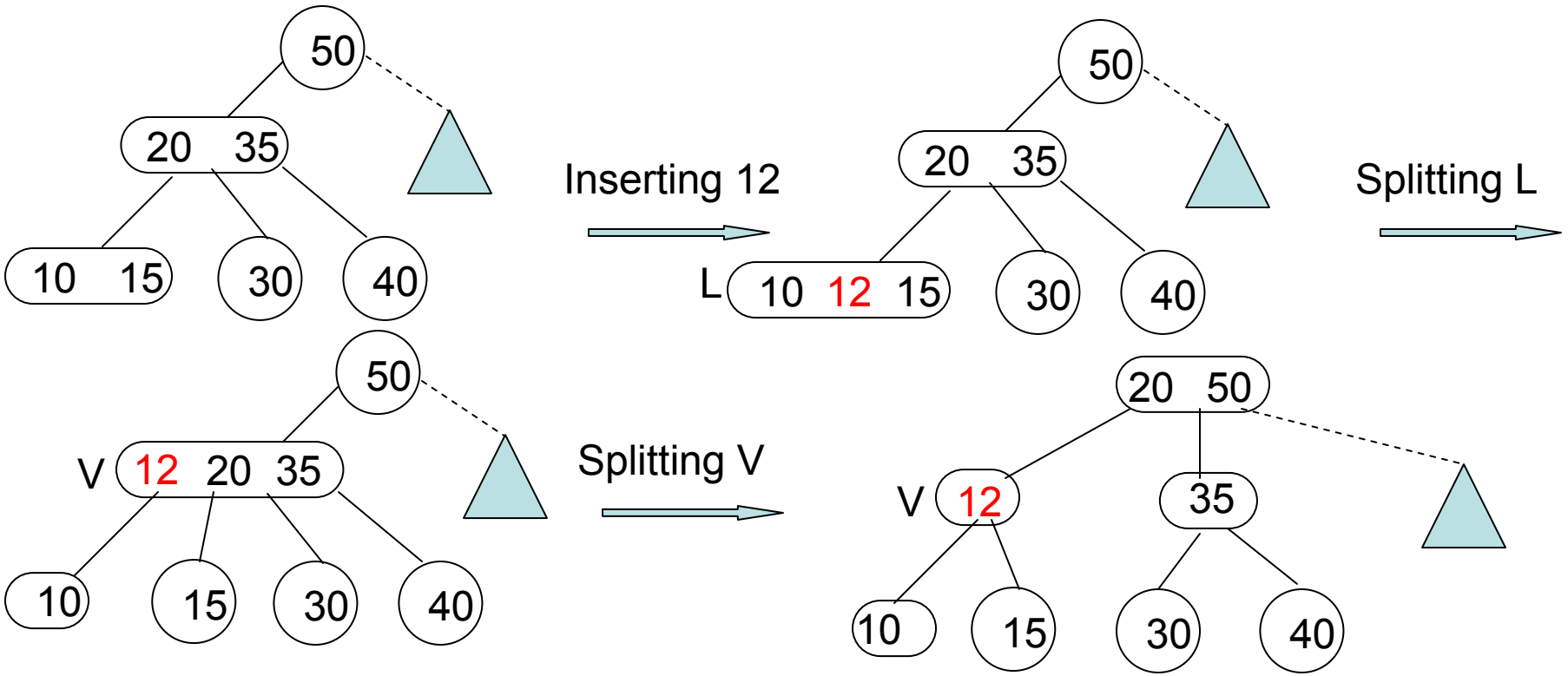
- Insertion into a 3-node leaf splits the leaf



- Insertion into a 3-node leaf splits the leaf



- Insertion into a 3-node leaf splits the leaf



2-3 Trees: The Insertion Algorithm

- When an internal node contains three items
  - Move the middle value to the node's parrent
  - Split the node into two nodes
  - Accommodate the node's children

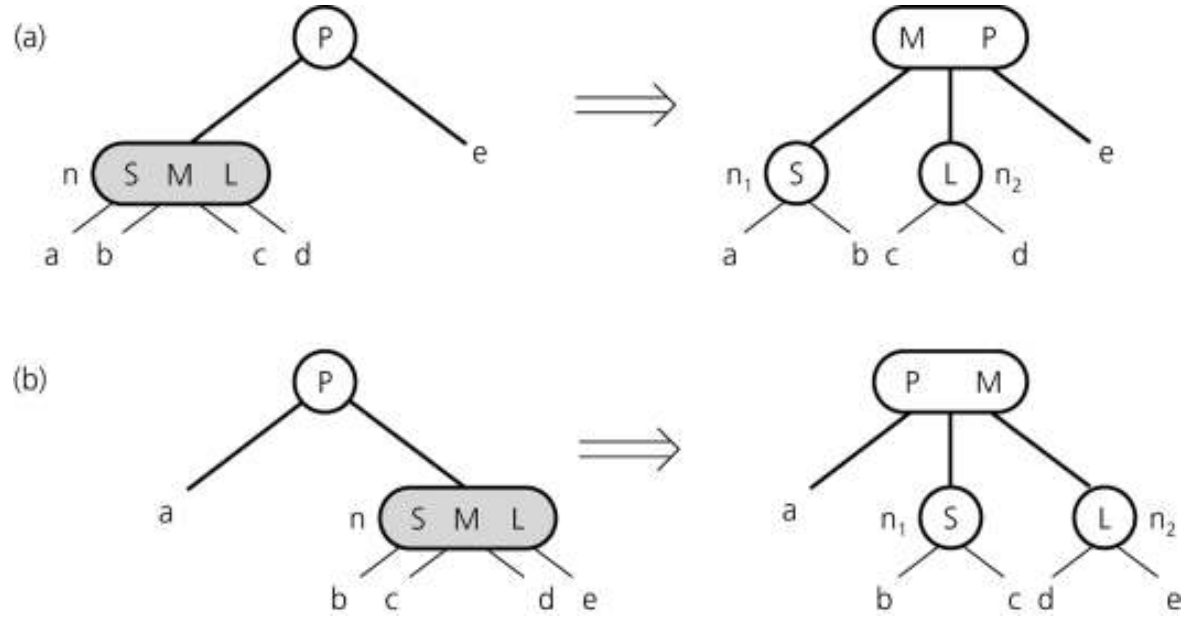


Figure 13-13

Splitting an internal node in a 2-3 tree

2-3 Trees: The Insertion Algorithm

- When the root contains three items
  - Split the root into two nodes
  - Create a new root node

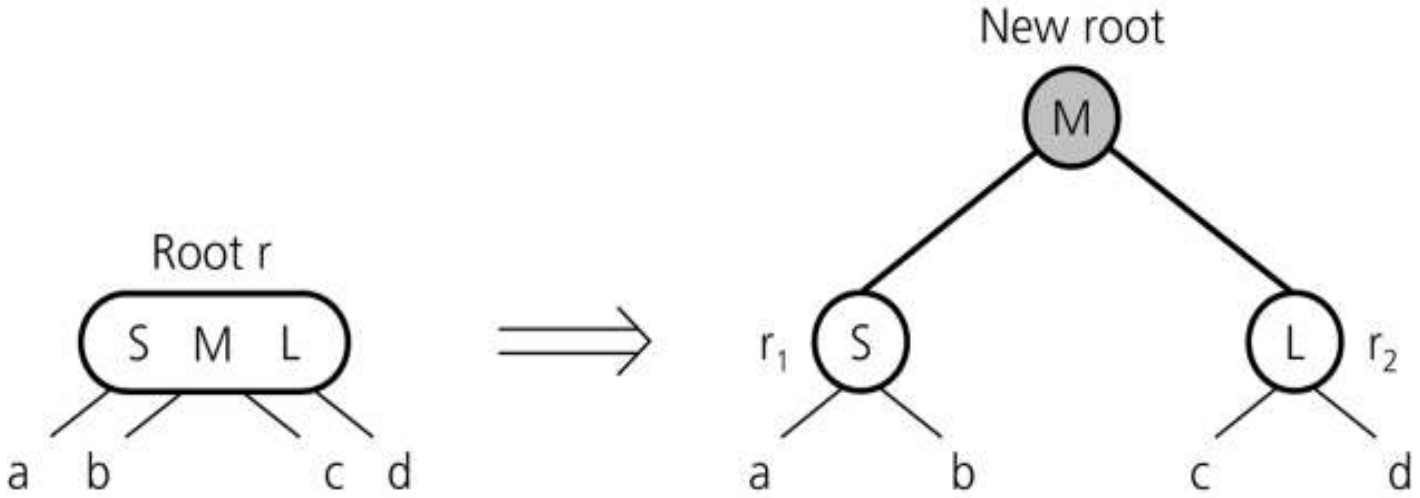


Figure 13-14

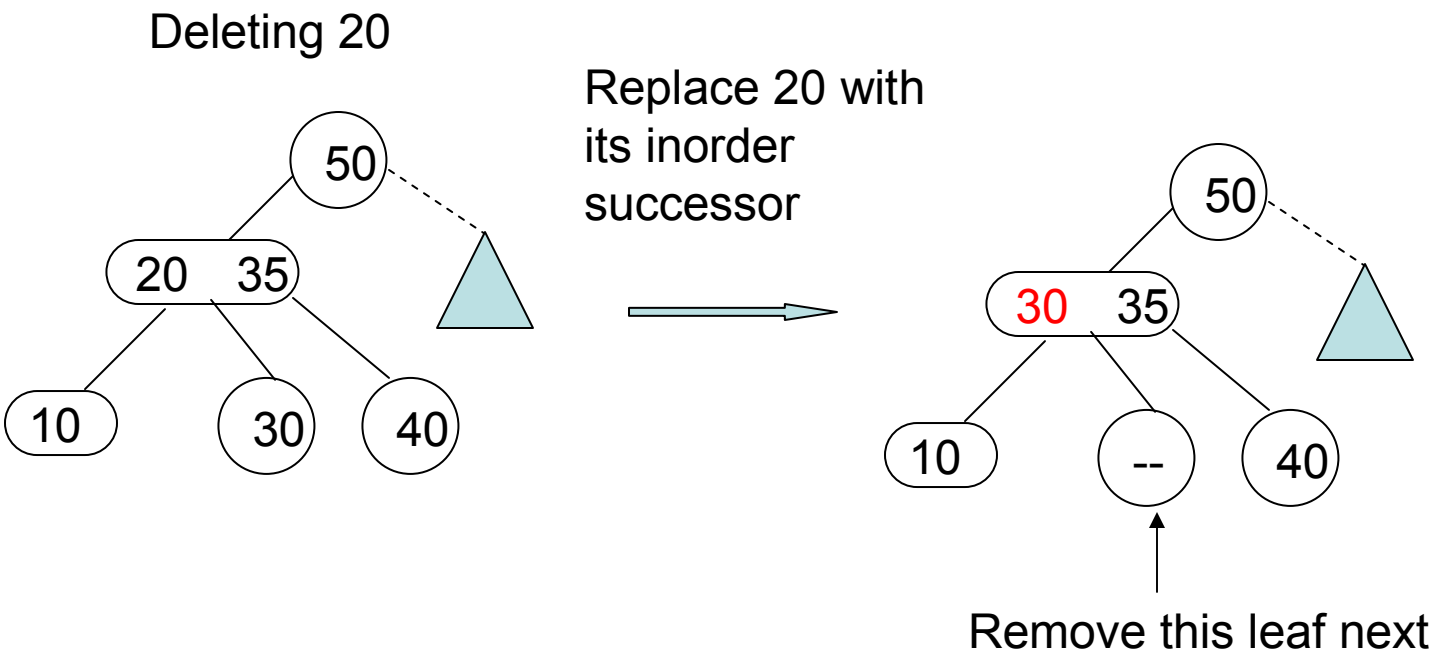
Splitting the root of a 2-3 tree

2-3 Trees: Deleting a node

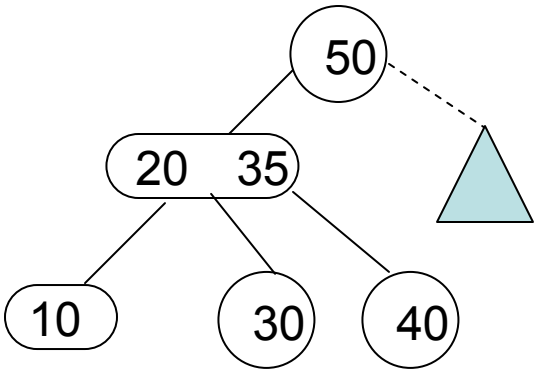
- Deletion from a 2-3 tree
  - Does not affect the balance of the tree
- Deletion from a balanced binary search tree
  - May cause the tree to lose its balance

2-3 Trees: Deleting a node

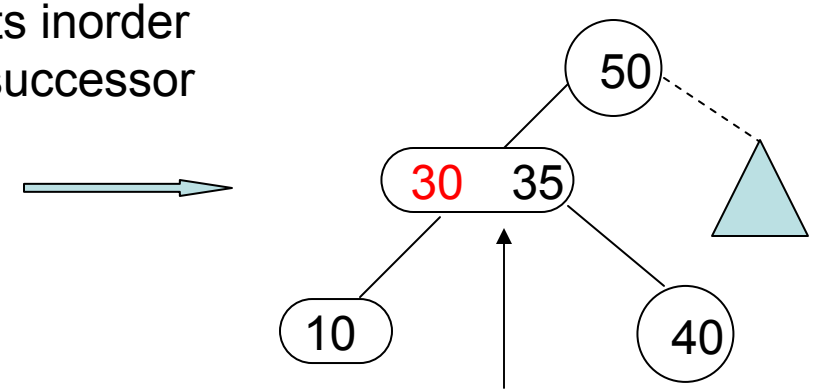
- The delete strategy is the inverse of the insert strategy.
- We merge the nodes when the become empty.
- We always want to begin the deletion process from a leaf (it’s just easier this way).
- Hence, for deleting an internal node first we exchange its value with a leaf and delete the leaf.


Deleting 20
50
20 35
10
30
40
Replace 20 with its inorder successor
50
30 35
10
--
40
Remove this leaf next

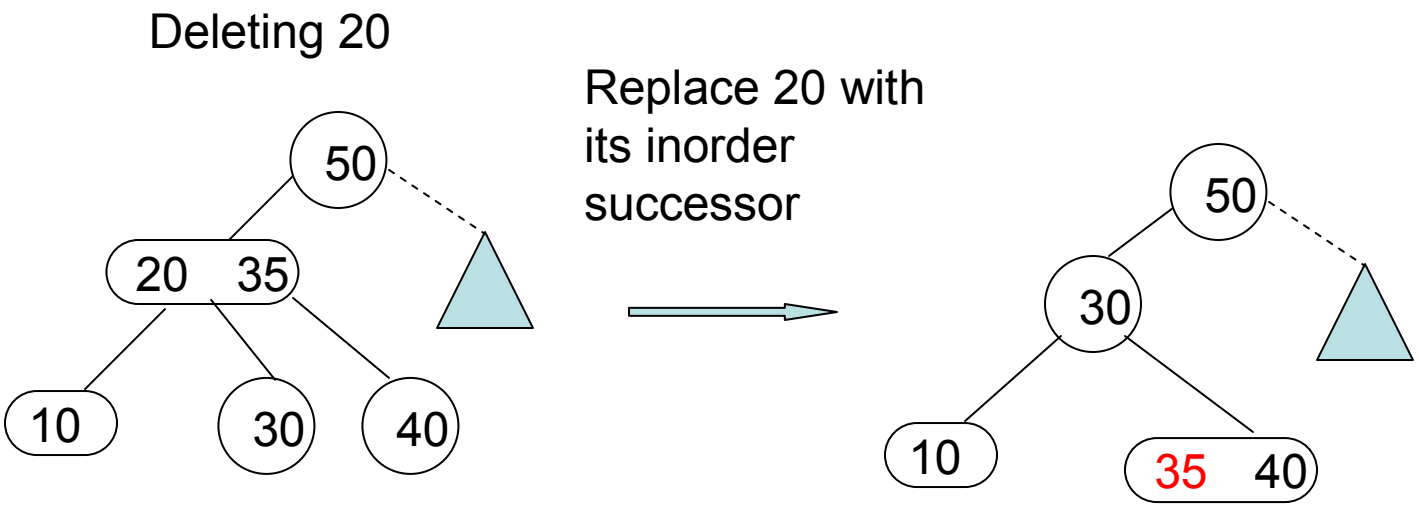
Deleting 20



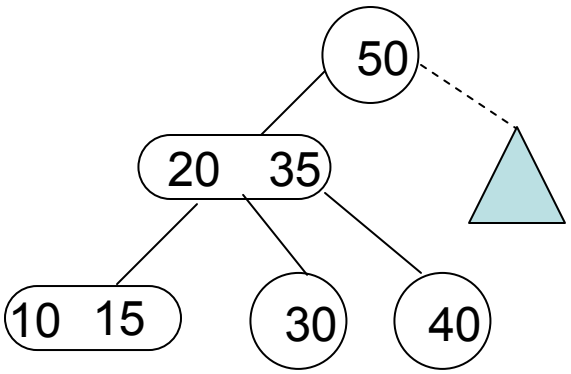
Replace 20 with its inorder successor



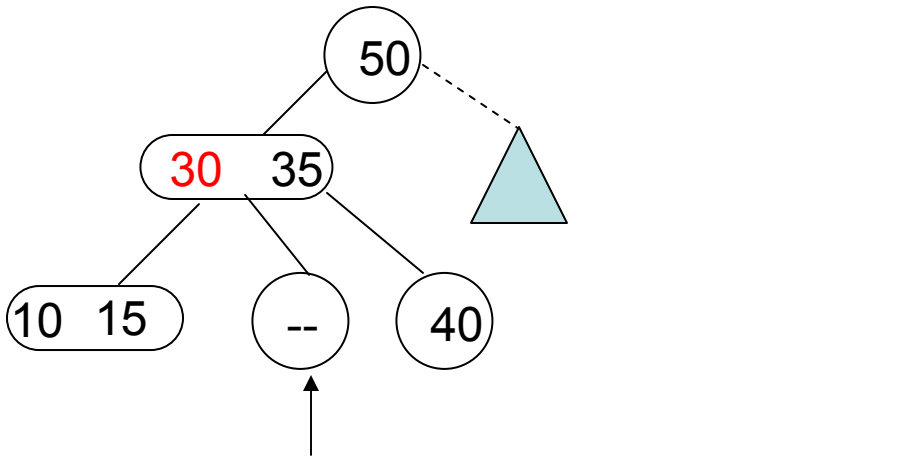
This must become a 2-node, move one of its values down.


Deleting 20
50
20 35
10
30
40
Replace 20 with
its inorder
successor
50
30
10
35 40

Deleting 20

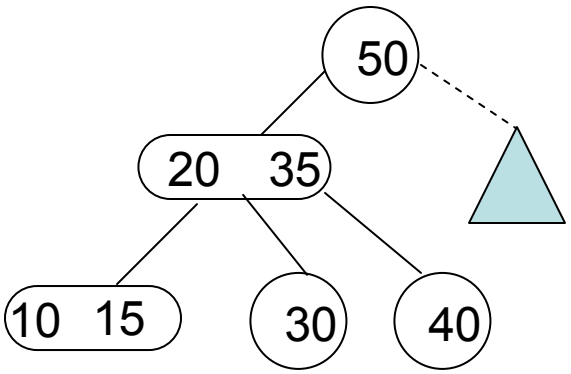


Replace 20 with
its inorder
successor

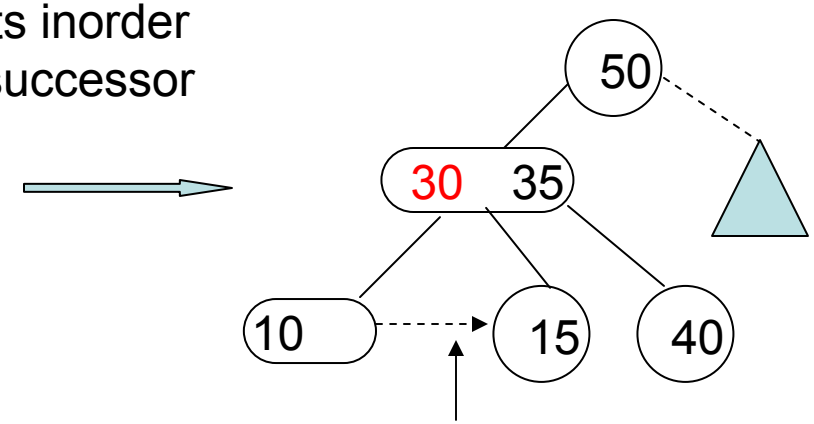


Fill this leaf by borrowing a value from
The left sibling (this is called redistributing)

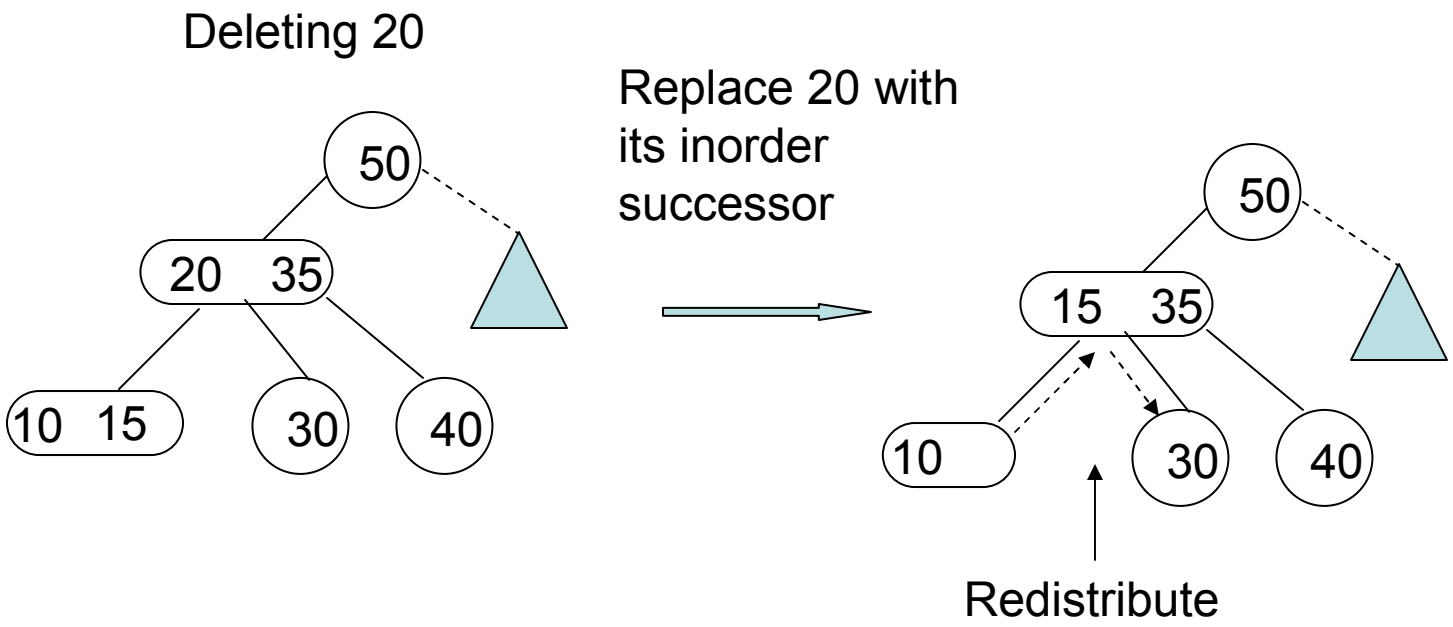
Deleting 20



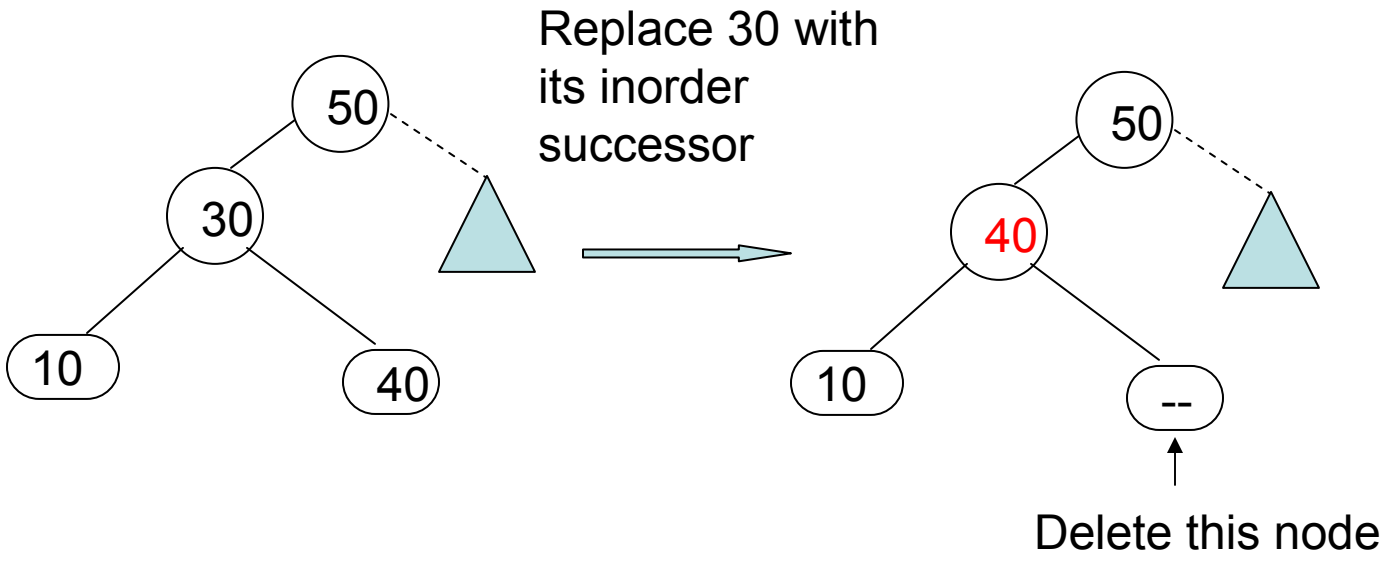
Replace 20 with its inorder successor



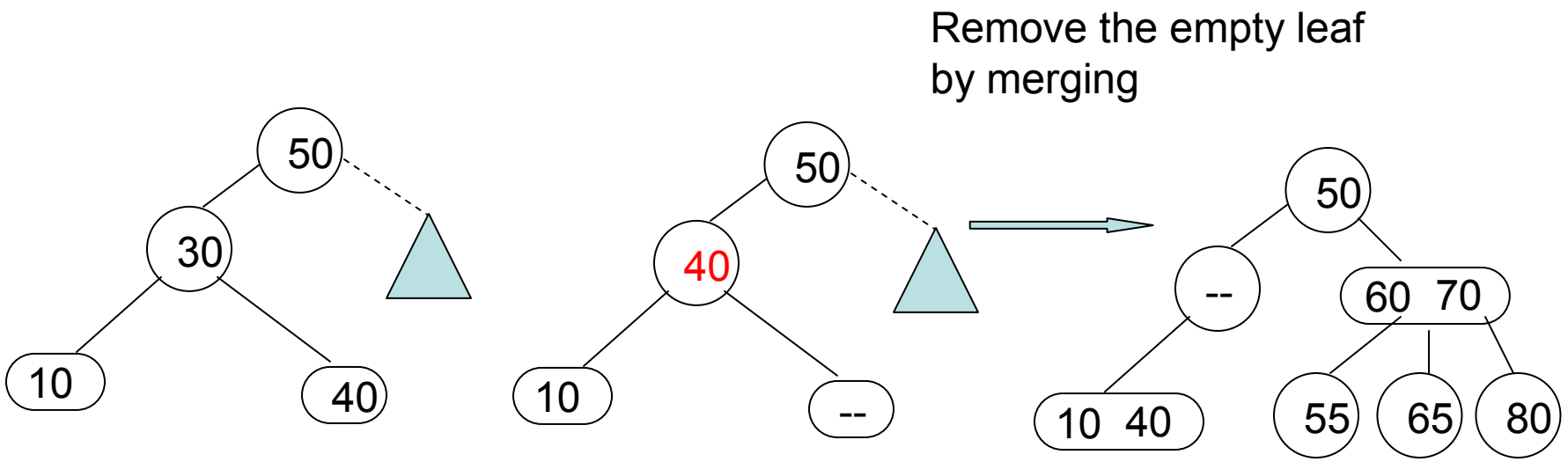
This does not work.


Deleting 20
50
20 35
10 15
30
40
Replace 20 with
its inorder
successor
50
15 35
10
30
40
Redistribute

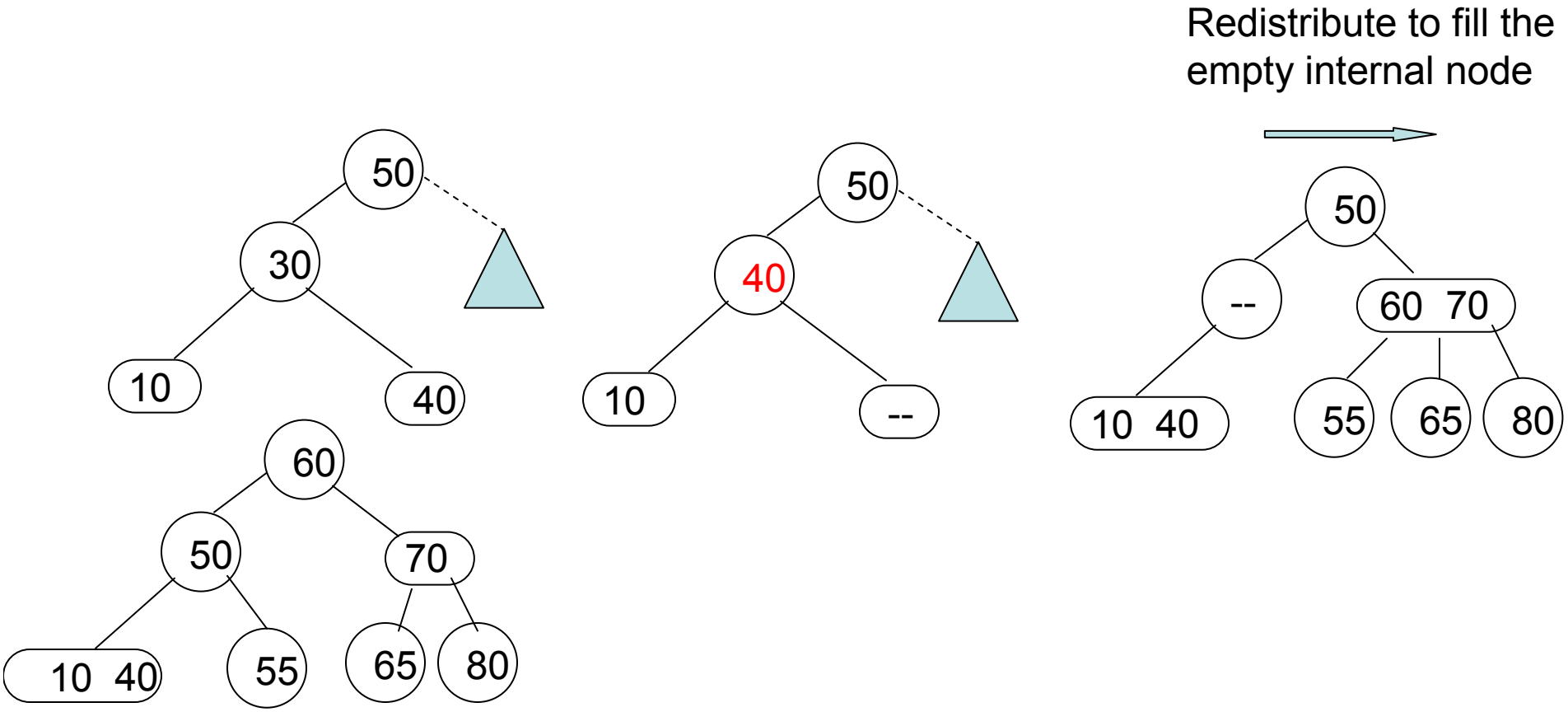
- Sometimes we may need to merge an internal node.



- Sometimes we may need to merge an internal node.



- Sometimes we may need to merge an internal node.

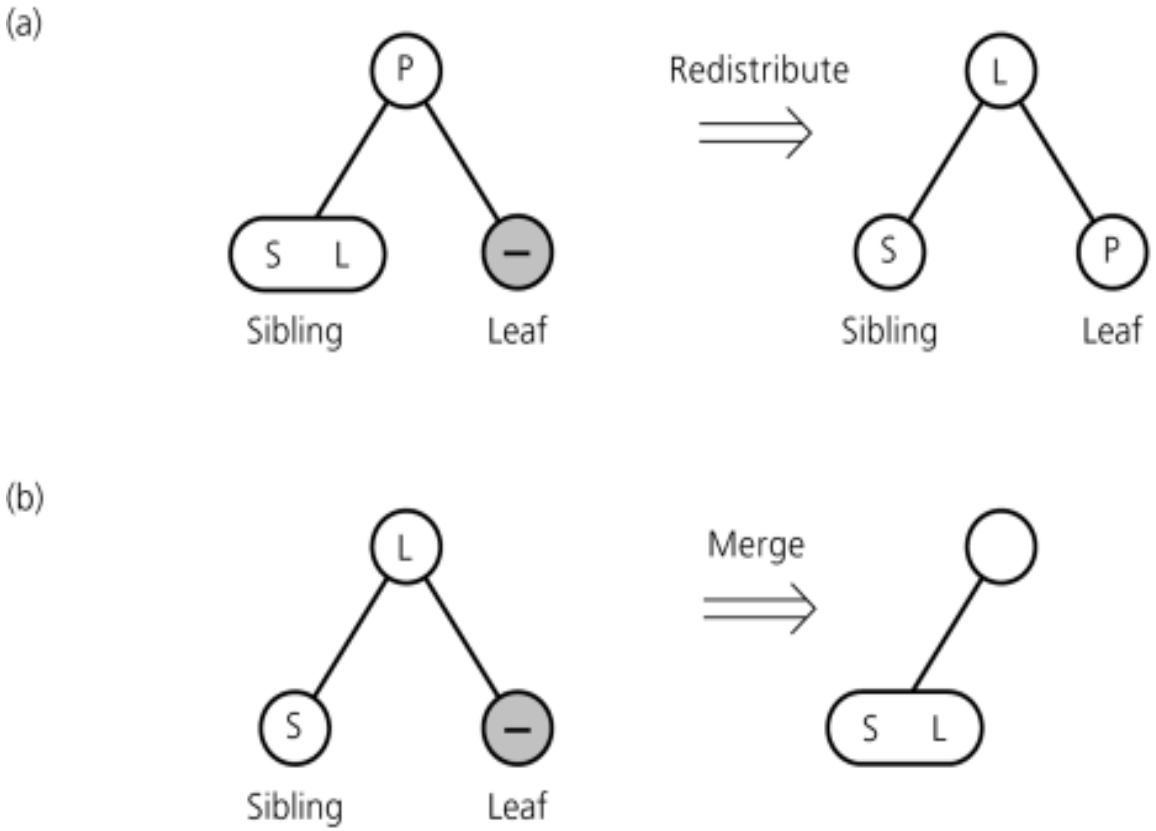


2-3 Trees: Deletion

Figure 13-19a and 13-19b

a) Redistributing values;

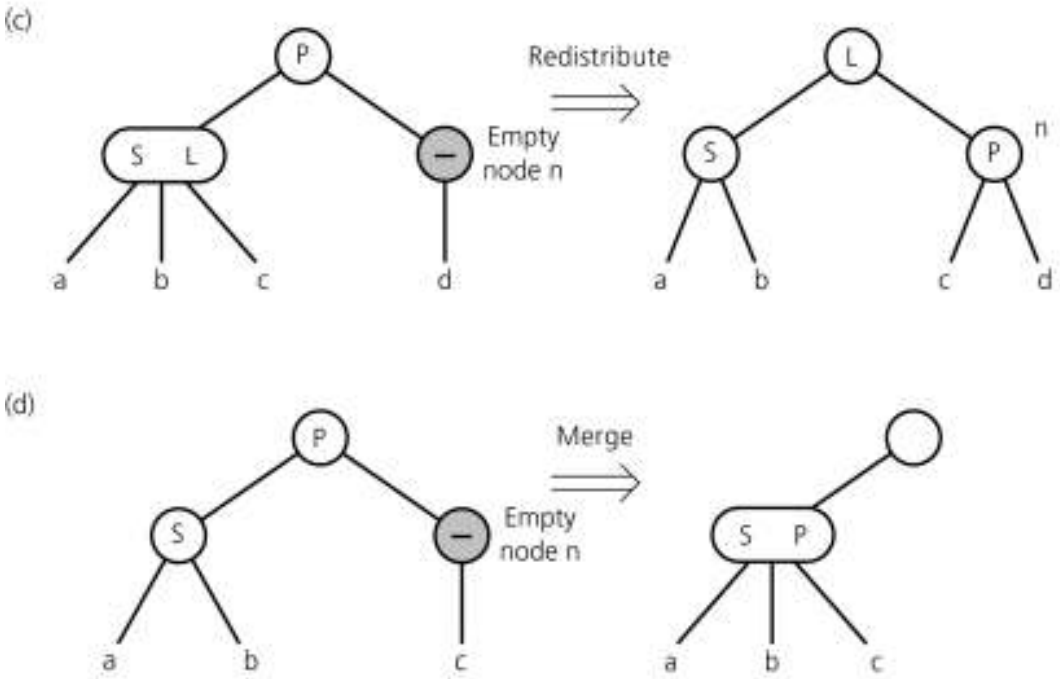
b) merging a leaf



2-3 Trees: The Deletion Algorithm

Figure 13-19c and 13-19d

c) redistributing values and children; d) merging internal nodes



2-3 Trees: The Deletion Algorithm

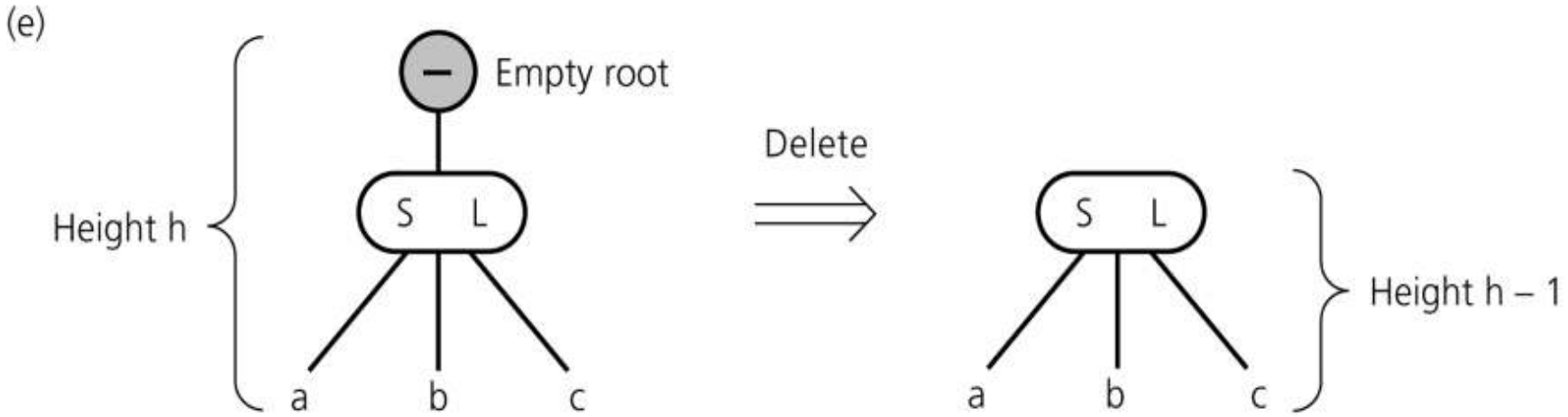


Figure 13-19e

e) deleting the root

2-3 Trees: The Deletion Algorithm

- When analyzing the efficiency of the `insertItem` and `deleteItem` algorithms, it is sufficient to consider only the time required to locate the item
- A 2-3 implementation of a table is $O(\log_2 n)$ for all table operations
- A 2-3 tree is a compromise
  - Searching a 2-3 tree is not quite as efficient as searching a binary search tree of minimum height
  - A 2-3 tree is relatively simple to maintain

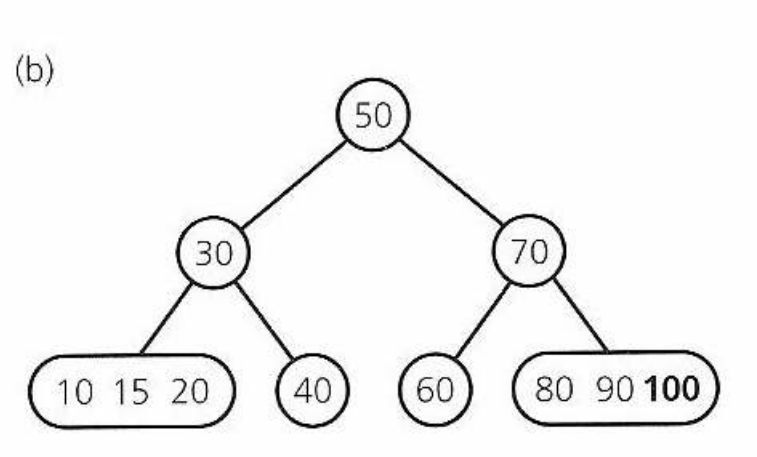
2-3-4 Trees

- 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 keys satisfy the relationships pictured in Figure 13-3a
  - A 3-node must contain two data items whose search keys satisfy the relationships pictured in Figure 13-3b
  - A 4-node must contain three data items whose search keys S, M, and L satisfy the relationship pictured in Figure 13-21
  - A leaf may contain either one, two, or three data items

Figure 13-21

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

- A 2-3-4 tree.



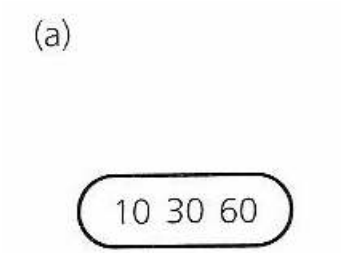
2-3-4 Trees: Searching and Traversing a 2-3-4 Tree

- Search and traversal algorithms for a 2-3-4 tree are simple extensions of the corresponding algorithms for a 2-3 tree

2-3-4 Trees: Inserting into a 2-3-4 Tree

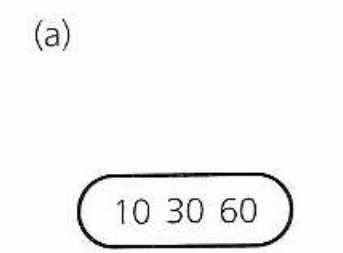
- The insertion algorithm for a 2-3-4 tree
  - Splits a node by moving one of its items up to its parent node
  - Splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf
    - Result: when a 4-node is split and an item is moved up to the node's parent, the parent cannot possibly be a 4-node and can accommodate another item

Result of inserting 10, 60, 30 in an empty 2-3-4 tree

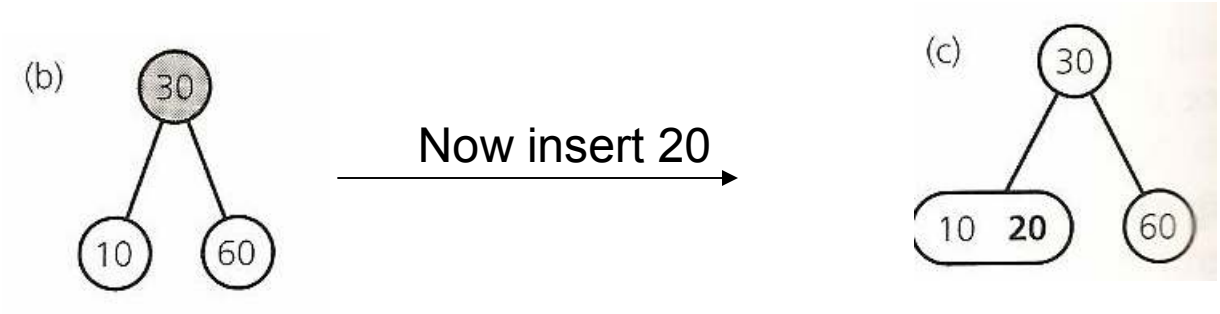


Now inserting 20. before that the 4-node <10, 30, 60> must be split

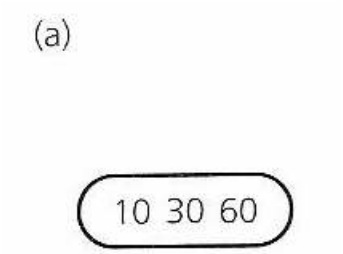
Result of inserting 10, 60, 30 in an empty 2-3-4 tree



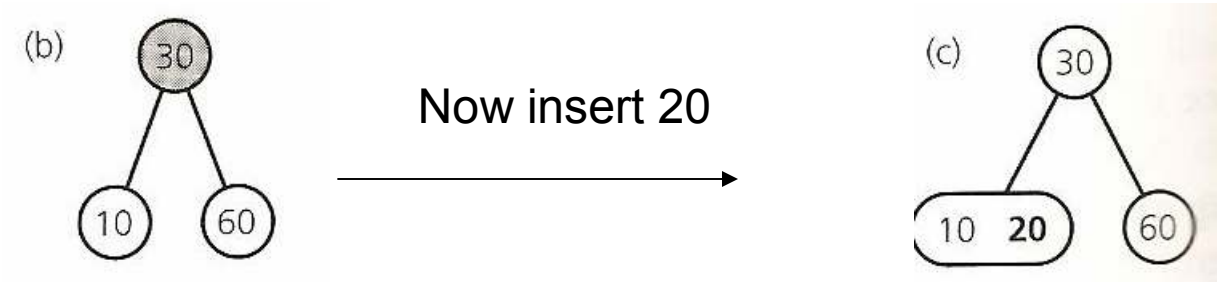
Inserting 20. before that the 4-node <10, 30, 60> must be split



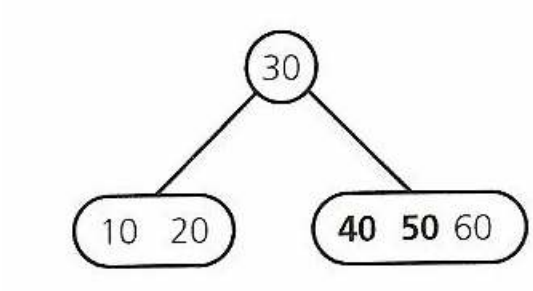
Result of inserting 10, 60, 30 in an empty 2-3-4 tree



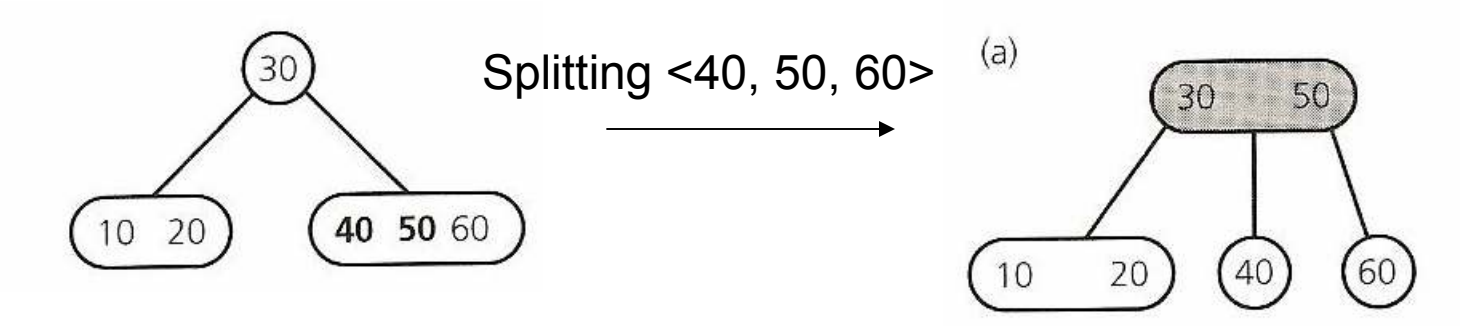
Inserting 20. before that the 4-node <10, 30, 60> must be split



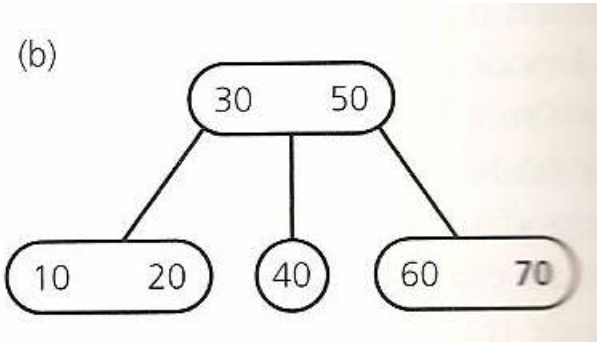
Inserting 50 and 40.



Inserting 70. Before that node <40, 50, 60> must split.



Now 70 is inserted.

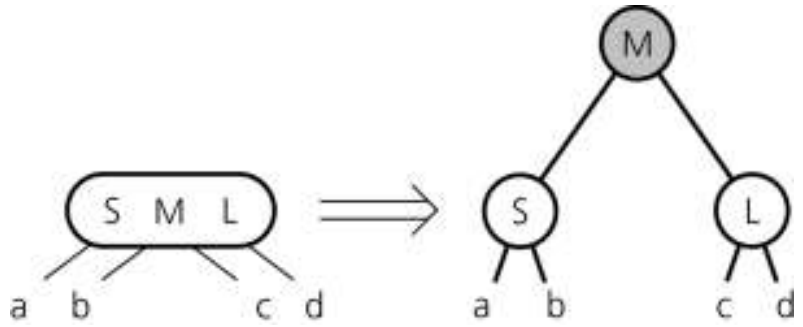


2-3-4 Trees: Splitting 4-nodes During Insertion

- A 4-node is split as soon as it is encountered during a search from the root to a leaf
- The 4-node that is split will
  - Be the root, or
  - Have a 2-node parent, or
  - Have a 3-node parent

Figure 13-28

Splitting a 4-node root during insertion



2-3-4 Trees: Splitting 4-nodes During Insertion

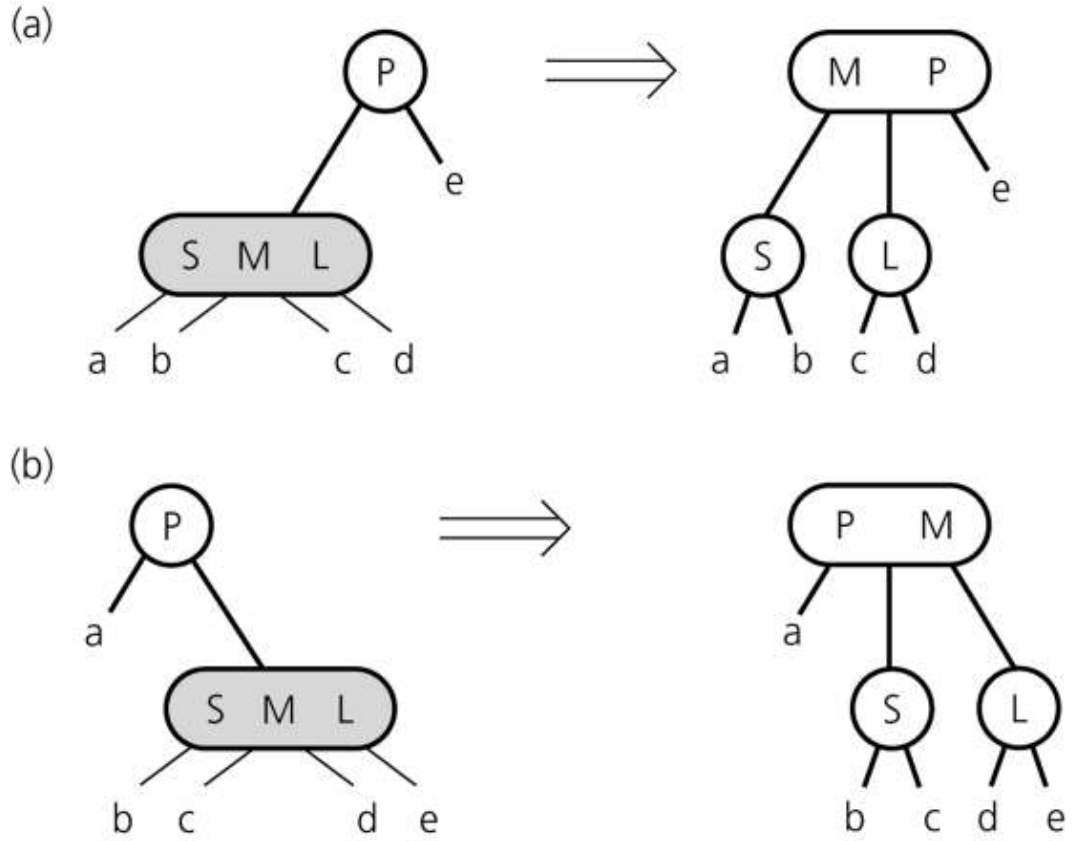


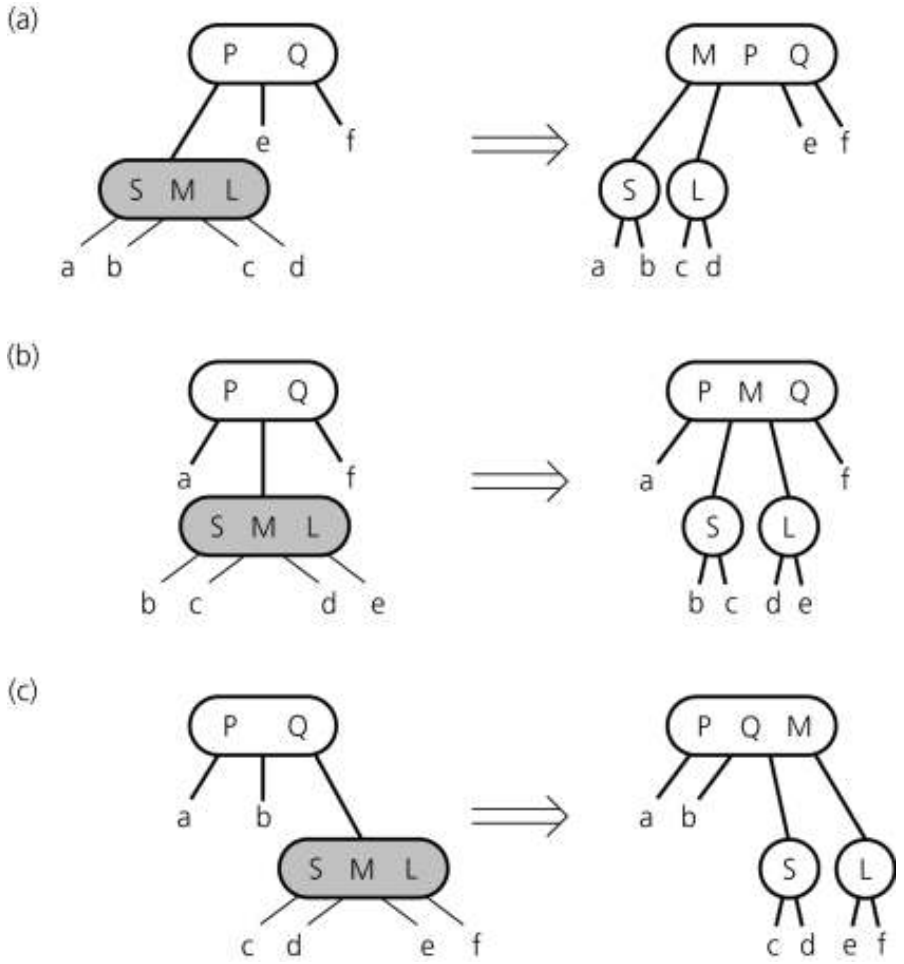
Figure 13-29

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

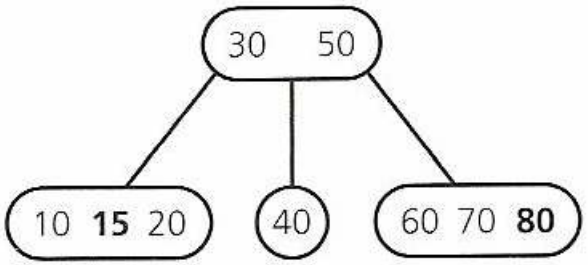
2-3-4 Trees: Splitting 4-nodes During Insertion

Figure 13-30

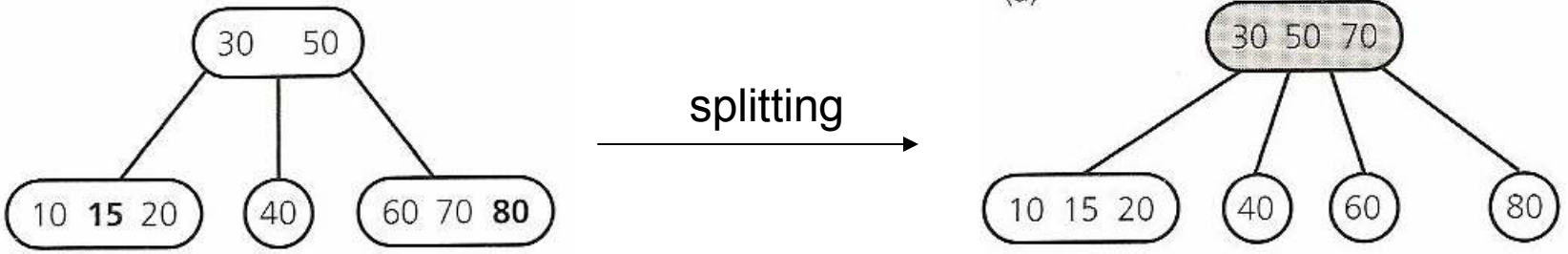
Splitting a 4-node whose parent is a 3-node during insertion



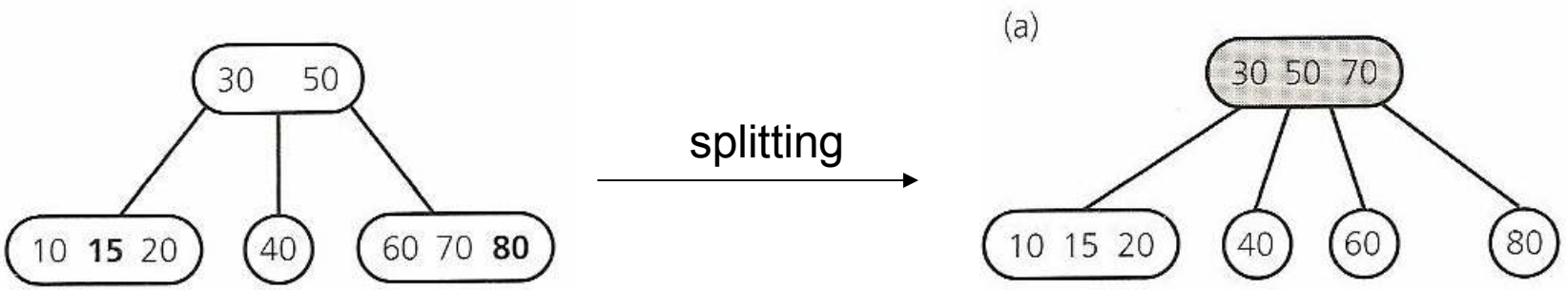
Inserting 90. First the node <60, 70, 80> must split.



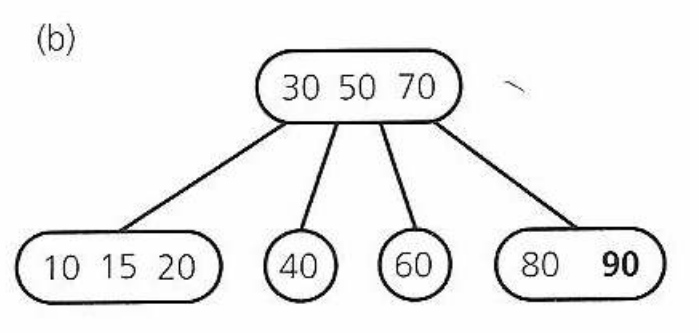
Inserting 90. First the node <60, 70, 80> must split.



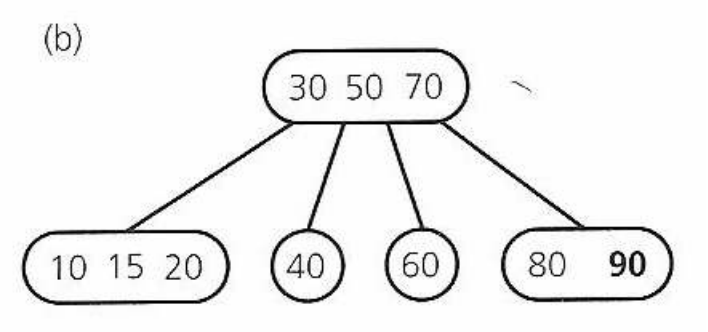
Inserting 90. First the node <60, 70, 80> must split.



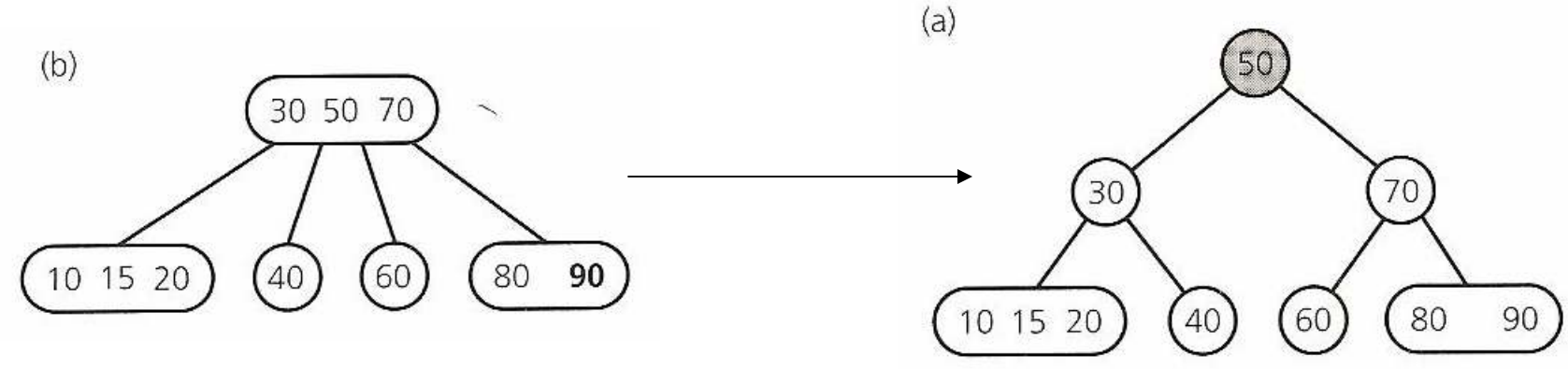
Inserting 90 in node <80>



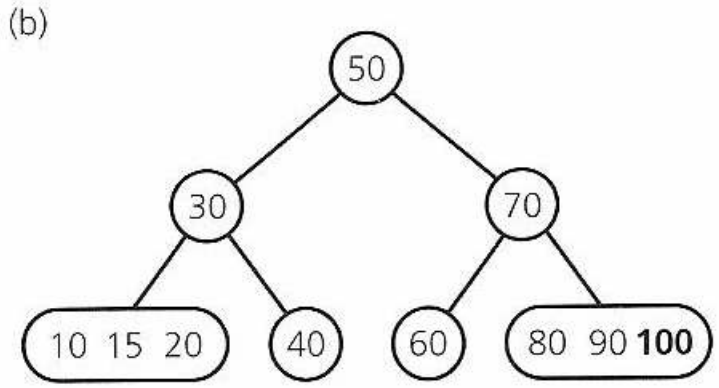
Inserting 100. First the root must split (because it's the first 4-node encountered in the path for searching 100 in the tree).



Inserting 100. First the root must split (because it's the first 4-node encountered in the path for searching 100 in the tree).



Inserting 100 in node <80, 90>

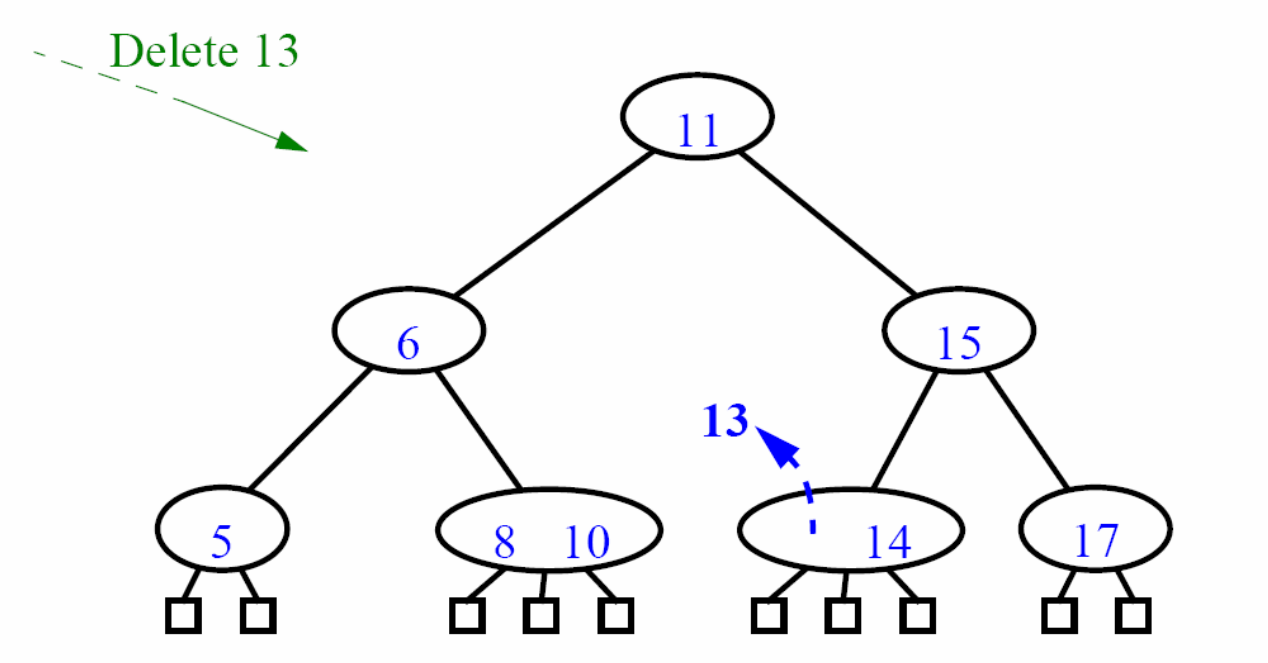


2-3-4 Trees: Deleting from a 2-3-4 Tree

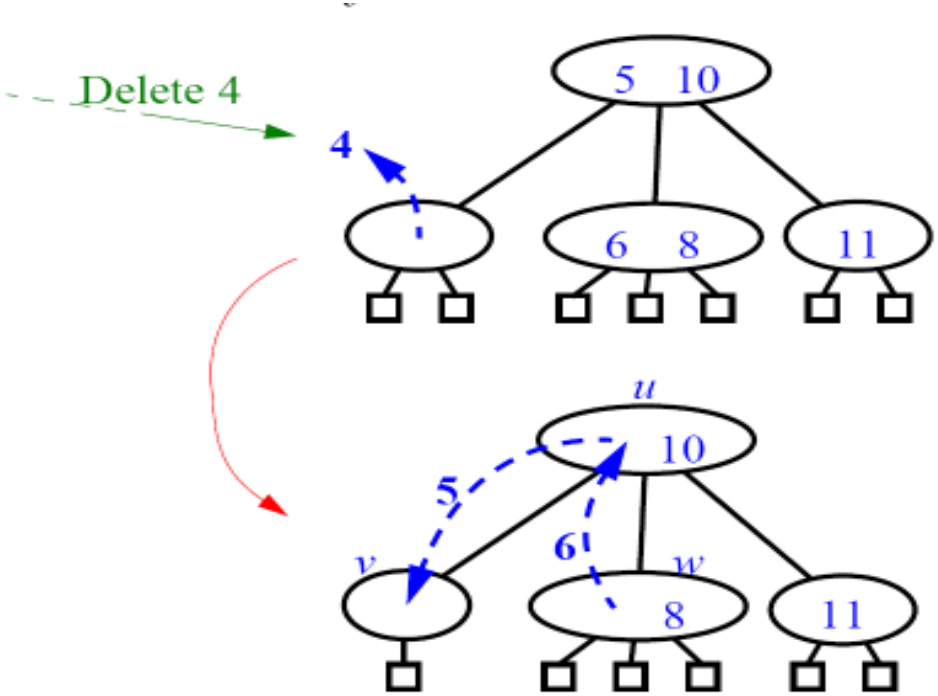
- The deletion algorithm for a 2-3-4 tree is the same as deletion from a 2-3 tree.
  - Locate the node n that contains the item `theItem`
  - Find `theItem`'s inorder successor and swap it with `theItem` (deletion will always be at a leaf)
  - Delete the leaf.
  - To ensure that `theItem` does not occur in a 2-node
    - Transform each 2-node the you encountered during the search for `theItem` into a 3-node or a 4-node

How do we delete a leaf

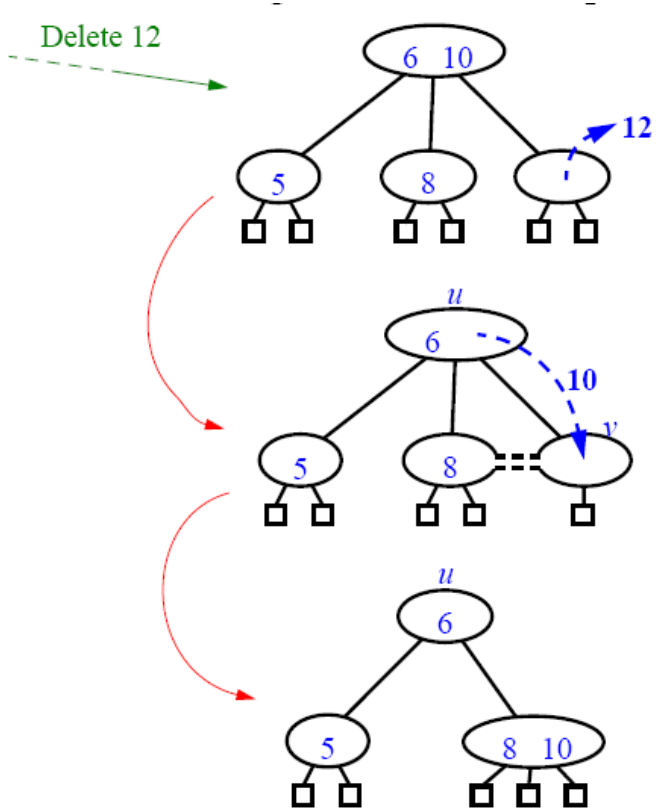
– If that leaf is a 3-node or a 4-node, remove `theItem` and we are done.



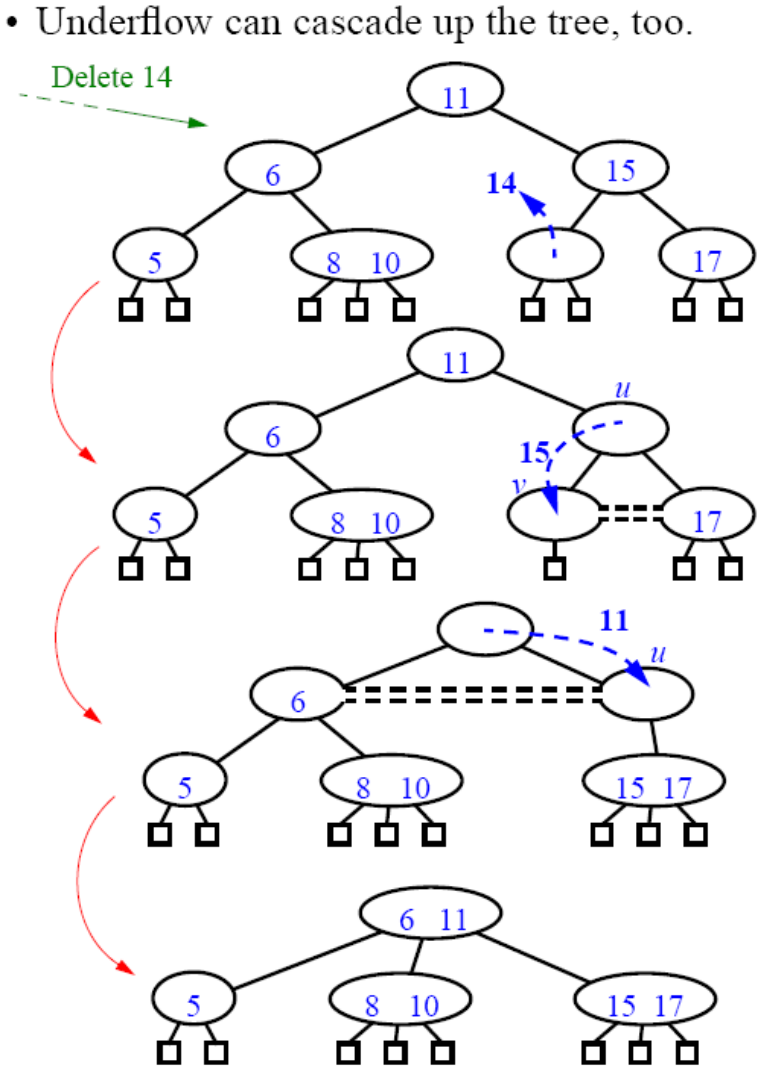
- What if the leaf is a 2-node
  - This is called underflow
  - We need to consider several cases.
  - **Case 1**: the leaf's sibling is not a 2-node
    - Transfer an item from the parent into the leaf and replace the pulled item with an item from the sibling.



- **Case 2**: the leaf's sibling is a 2-node but it's parent is not a 2-node.
  - We *fuse* the leaf and sibling.



- **Case 3**: the leaf's sibling and parent are both 2-node.



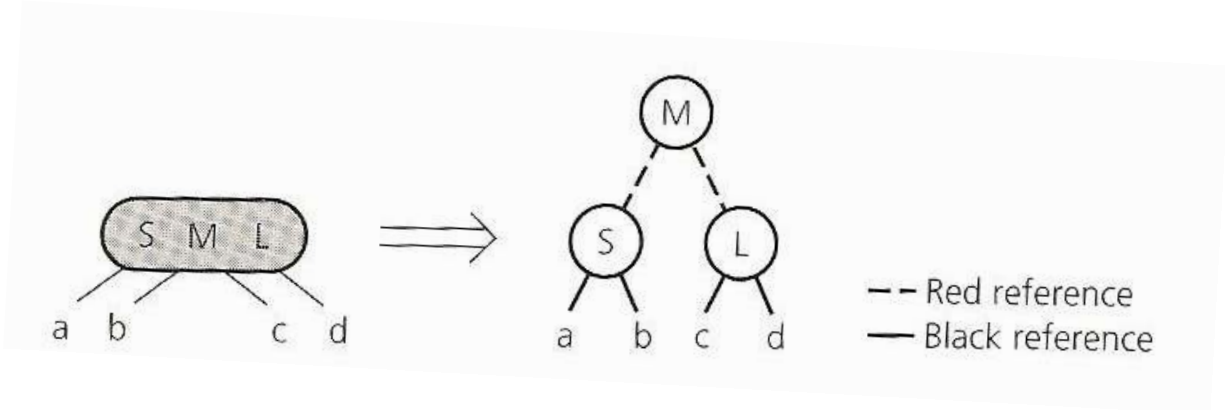
2-3-4 Trees: Concluding Remarks

- Advantage of 2-3 and 2-3-4 trees
  - Easy-to-maintain balance
- Insertion and deletion algorithms for a 2-3-4 tree require fewer steps that those for a 2-3 tree
- Allowing nodes with more than four children is counterproductive

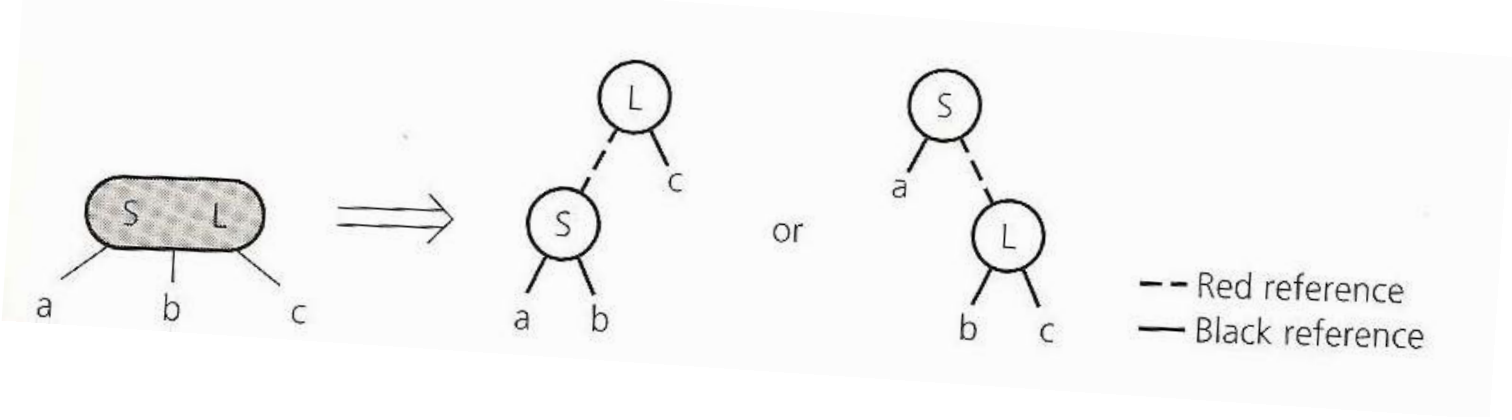
- Red-Black trees (Optional).

- A 2-3-4 tree requires more space than a binary search tree that contains the same data.
- It's because the nodes of a 2-3-4 must accommodate 3 data values.
- We can use a special BST called the red-black tree that has the advantages of a 2-3-4 tree without the memory over head.
- The idea is to represent the 3-nodes and 4-nodes in a 2-3-4 tree as an equivalent BST node.

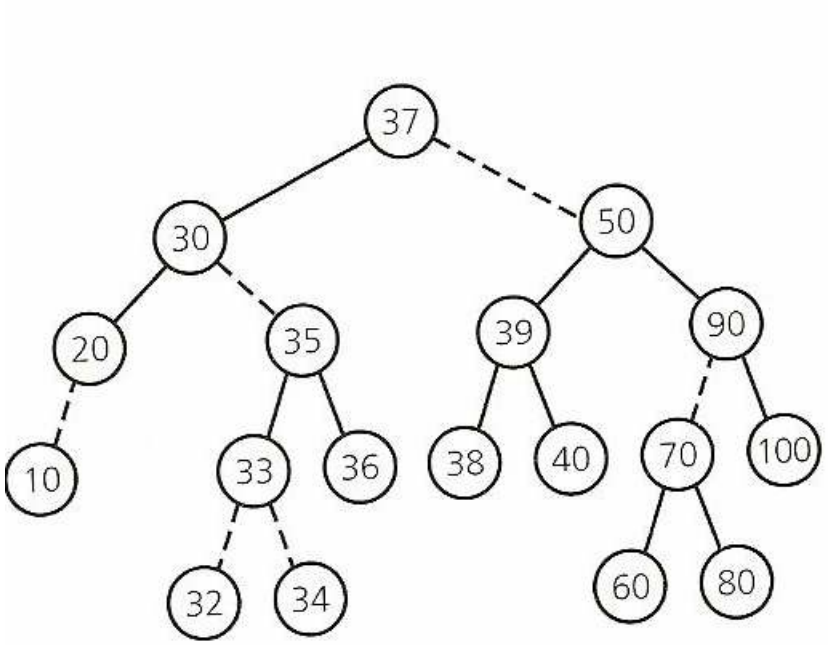
- Red-black representation of a 4-node



- Red-black representation of a 3-node



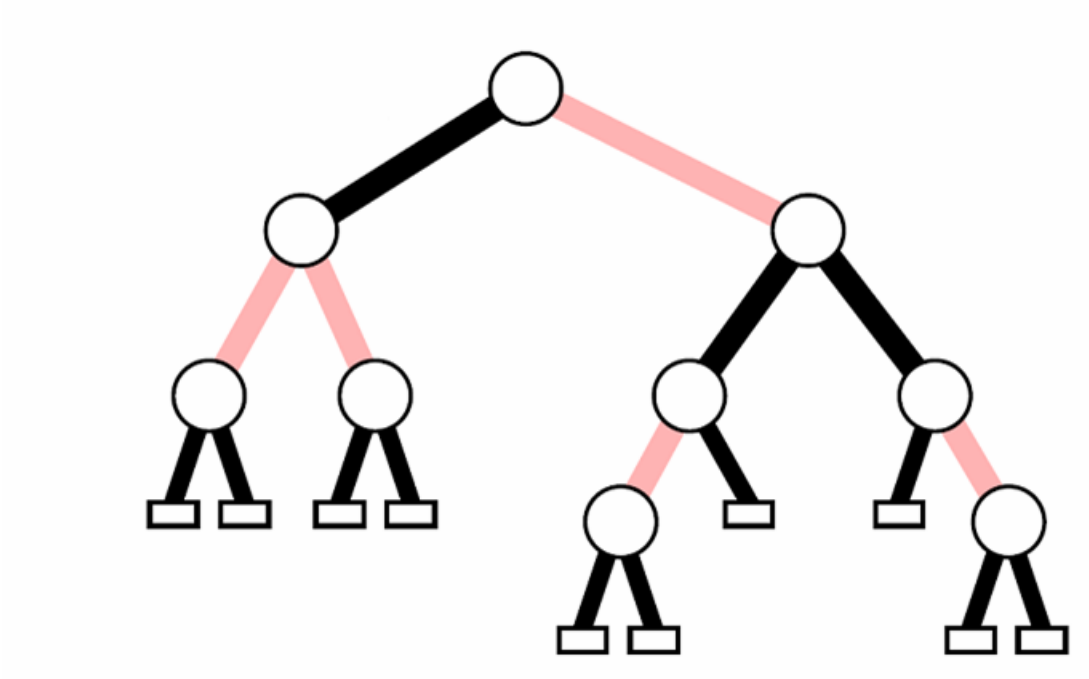
Red-black equivalent of a 2-3-4 tree



Red-black tree properties.

- Let
  - N: number of internal nodes.
  - H: height of the tree.
  - B: black height.
- Property 1: $2^B \leq N + 1 \leq 4^B$
- Property 2: $\frac{1}{2}\log(N + 1) \leq B \leq \log(N + 1)$
- Property 3: $\log(N + 1) \leq H \leq 2\log(N + 1)$
- This implies that searches take O(log N)

- In addition **false** nodes are added so that every (real) node has two children
  - These are **pretend** nodes, they don't have to have space allocated to them
  - The incoming edges to these nodes are colored black
  - We do not count them when measuring a height of nodes



Pretend nodes are squared nodes at the bottom.

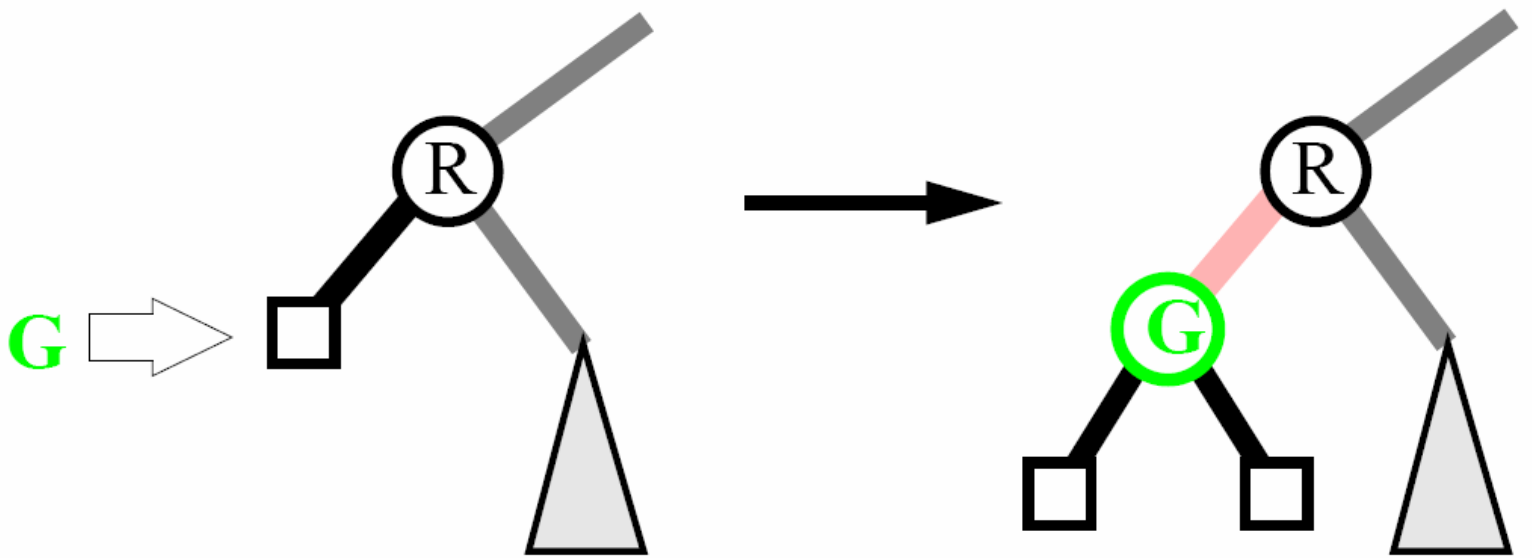
Important properties

- No two consecutive red edges exist in a red-black tree.
- The number of black edges in all the paths from root to a leaf is the same.

Insertion into Red-Black Trees

1. Perform a standard search to find the leaf where the key should be added
2. Replace the leaf with an internal node with the new key
3. Color the incoming edge of the new node red
4. Add two new leaves, and color their incoming edges black
5. If the parent had an incoming red edge, wenow have two consecutive red edges! We must reorganize tree to remove that violation. What must be done depends on the sibling of the parent.

- Inserting new node G.

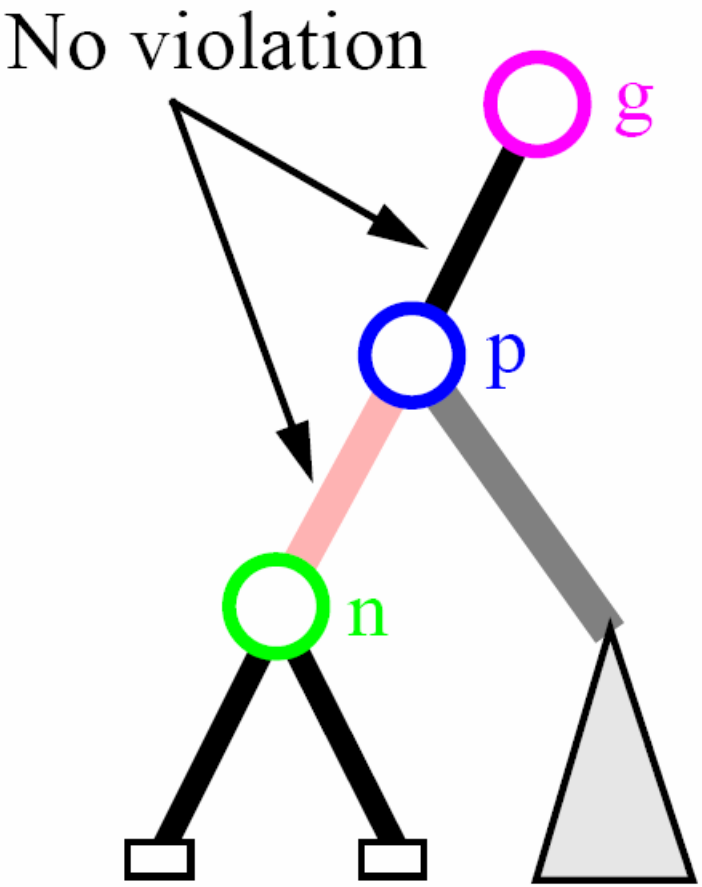


Insertion into a red-black tree

Let:

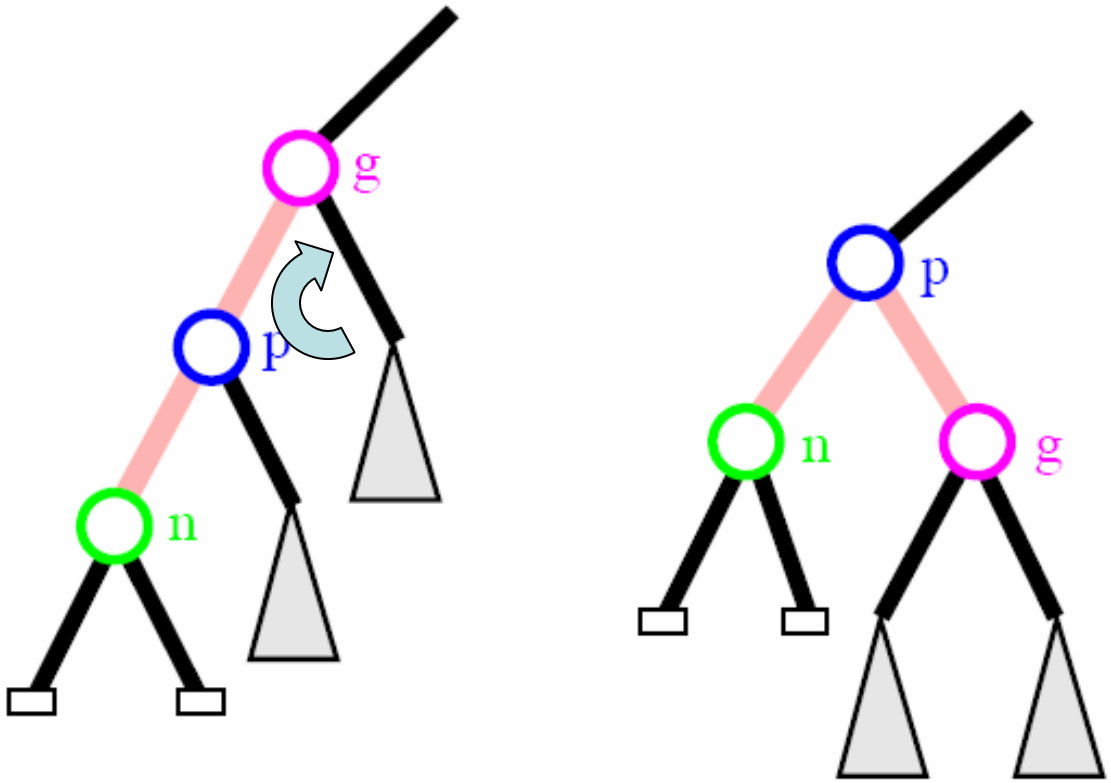
- n be the new node
- p be its parent
- g be its grandparent

Case 1: Incoming edge of p is black



STOP!

Case 2: Incoming edge of p is red, and its sibling is black

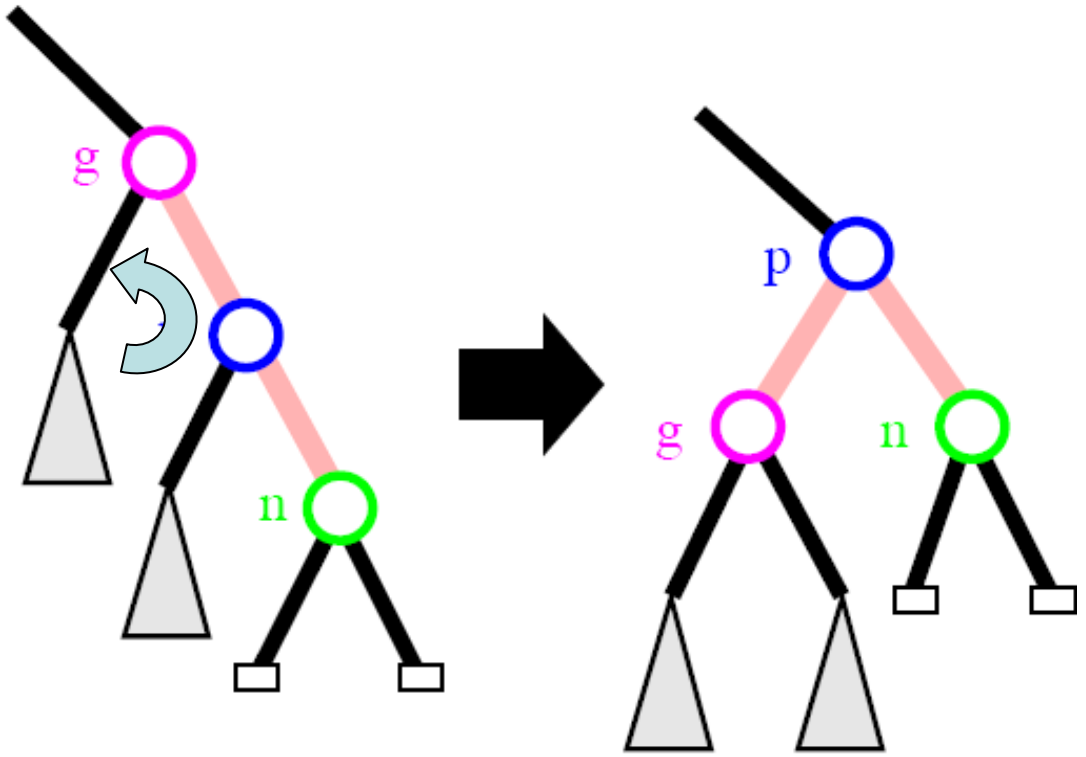


We call this a "*right rotation*"

- No further work on tree is necessary

Case 2. Continued.

Similar to a right rotation, we can do a "**left rotation**"...

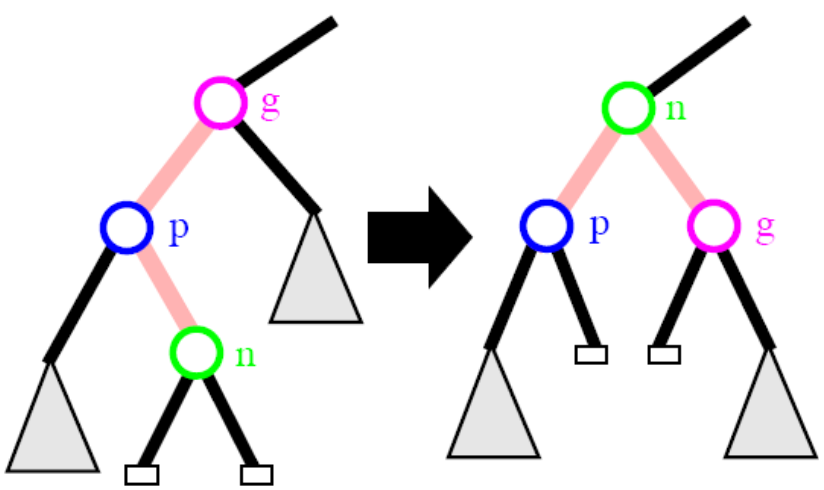


Case 2. Continued.

Double Rotations

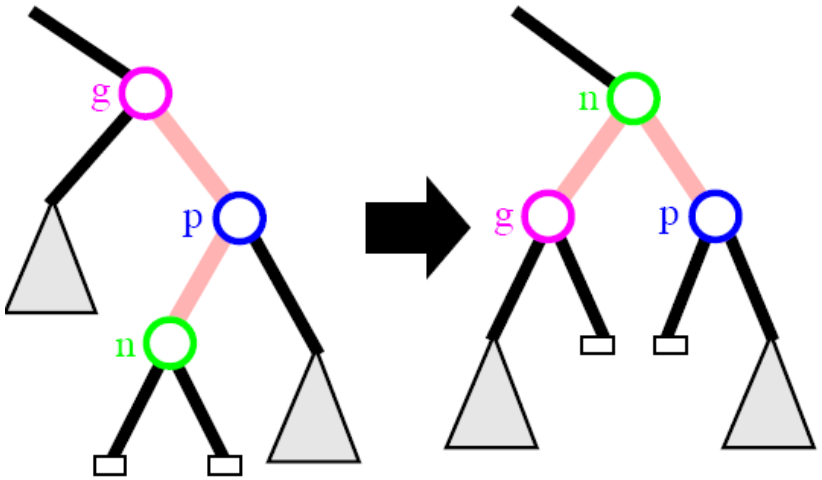
What if the new node is between its parent and grandparent in the inorder sequence?

We must perform a "double rotation"
(which is no more difficult than a "single" one)

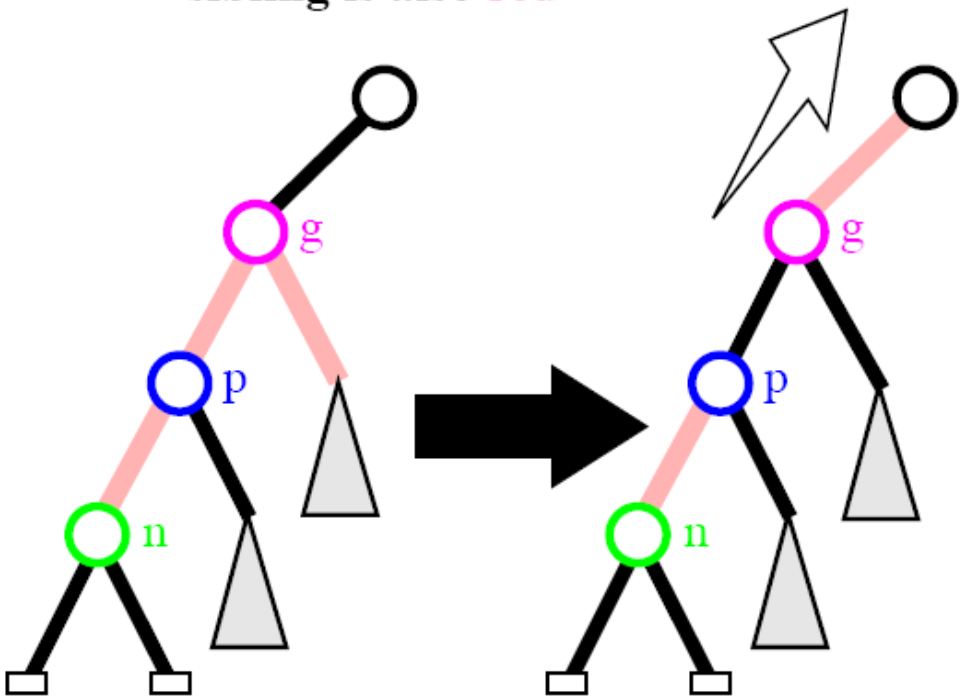


This would be called a
"left-right double rotation"

And this would be called a
"right-left double rotation"



Case 3: Incoming edge of p is red and its sibling is also red

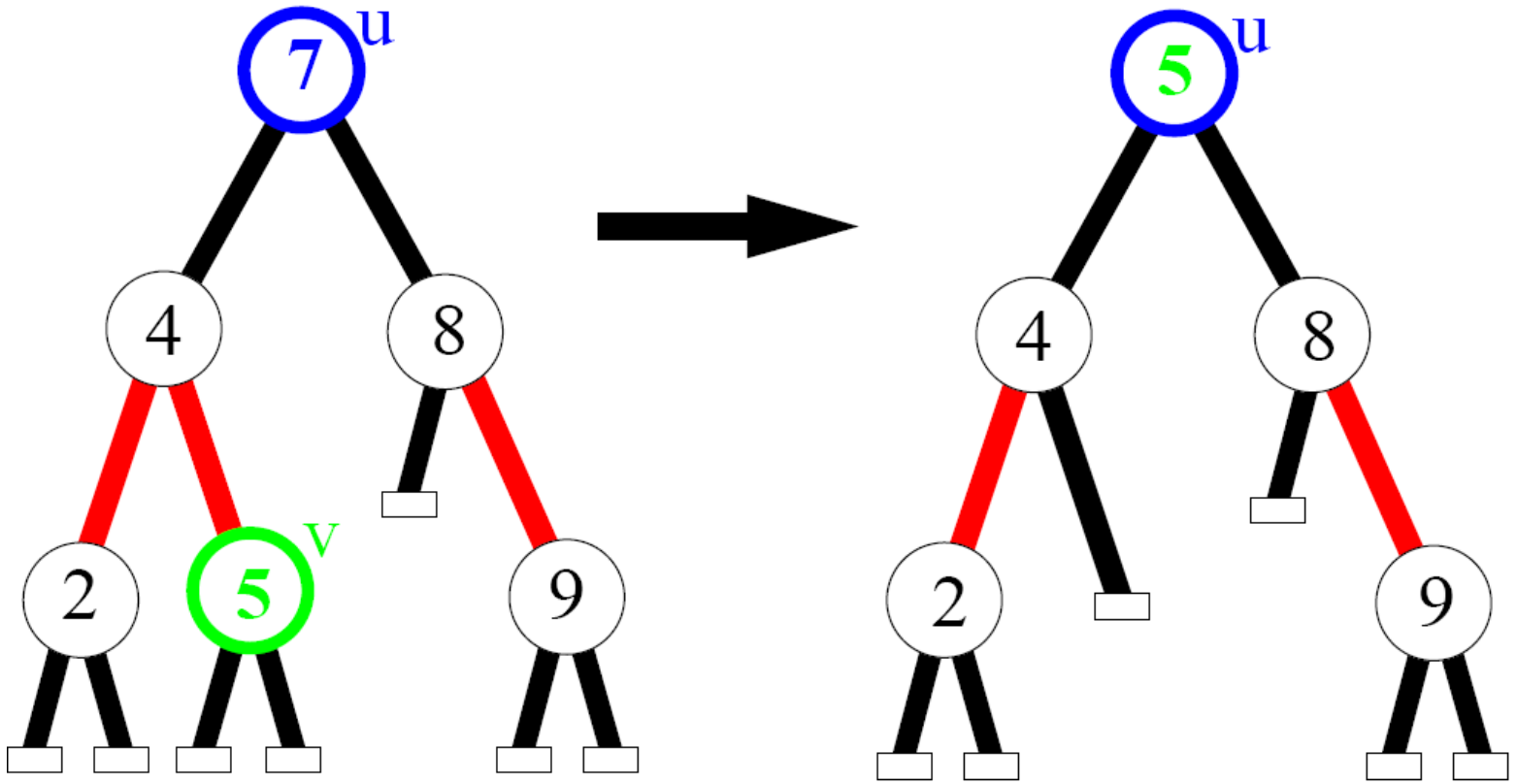


- We call this a "***promotion***"
- Note how the black depth remains unchanged for all of the descendants of **g**
- This process will continue upward beyond **g** if necessary: rename **g** as **n** and repeat.

Red-black tree deletion

- As with the binary search tree we will try to remove a node with at most one children.
- A node with at most one children is a node with at least one **pretended** or **external** child (i.e the null pointers that are treated as fake leaves).
- To remove an internal node we replace its value with the value of its successor and remove the successor node.
- The successor node always has at least one external child.

Example: to delete key 7, we move key 5 to node u, and delete node v

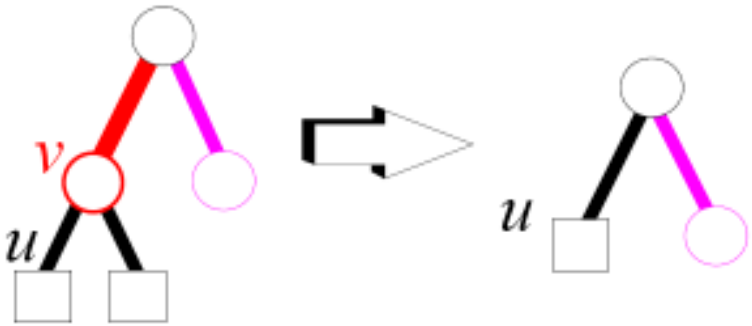


Square nodes are called external or pretended nodes.

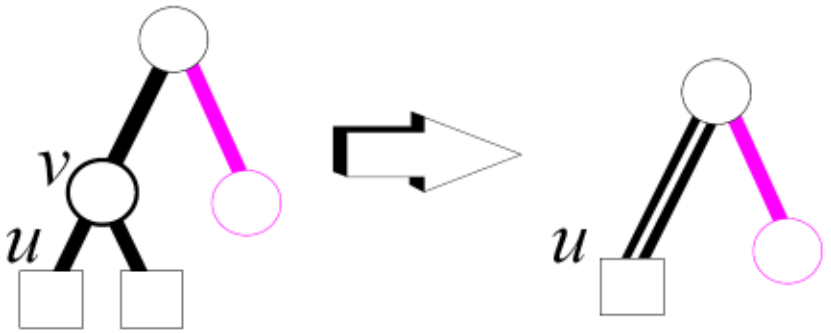
Deletion algorithm

- Assume we want to delete node v.
- V has at leas one external child.

1. Remove v by setting its parent points to u.
2. If v was red color u black and we are done.
3. If v was black color u double black.
4. Next, remove the double black edges.



We are done in this case



We need to reconstruct the tree.

Eliminating the double black nodes.

- The intuitive idea is to perform a "***color compensation***"
- Find a red edge nearby, and change the pair (***red*** , ***double black***) into (*black , black*)

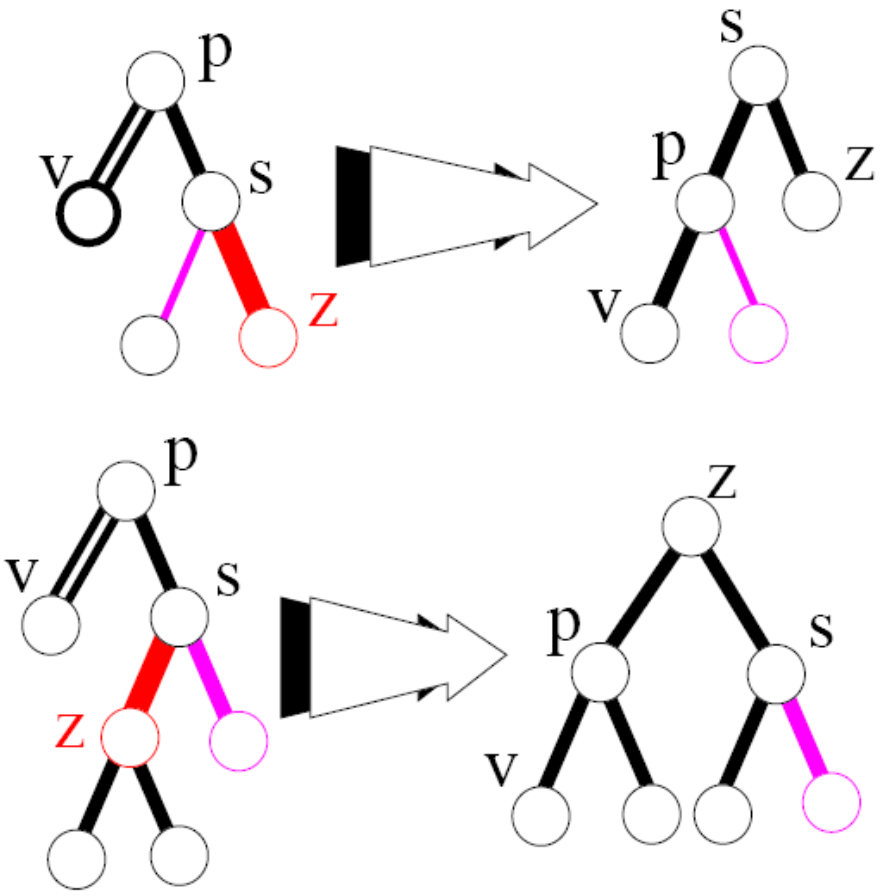
- As for insertion, we have two cases:
  - ***restructuring***, and
  - ***recoloring*** (***demotion,*** inverse of promotion)

- Restructuring resolves the problem locally, while recoloring may propagate it two levels up

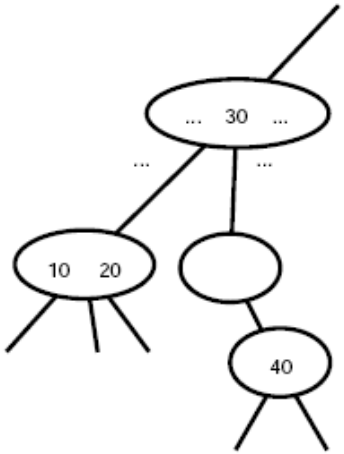
- Slightly more complicated than insertion, since two restructurings may occur (instead of just one)

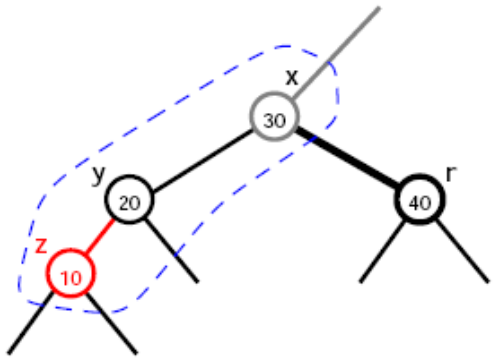
Case 1: black sibling with a red child

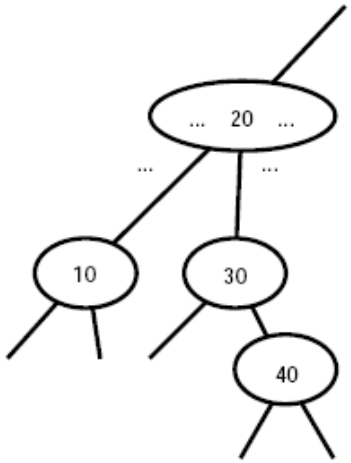
- If sibling is **black** and one of its children is **red**, perform a ***restructuring***

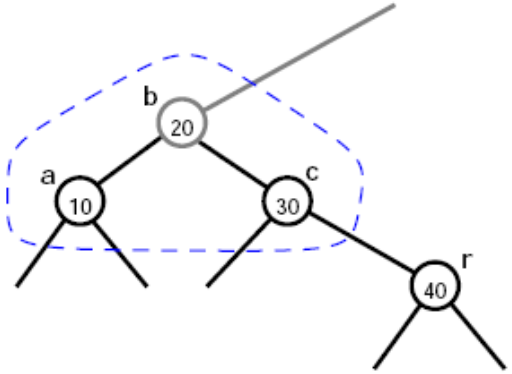


(2,4) Tree Interpretation



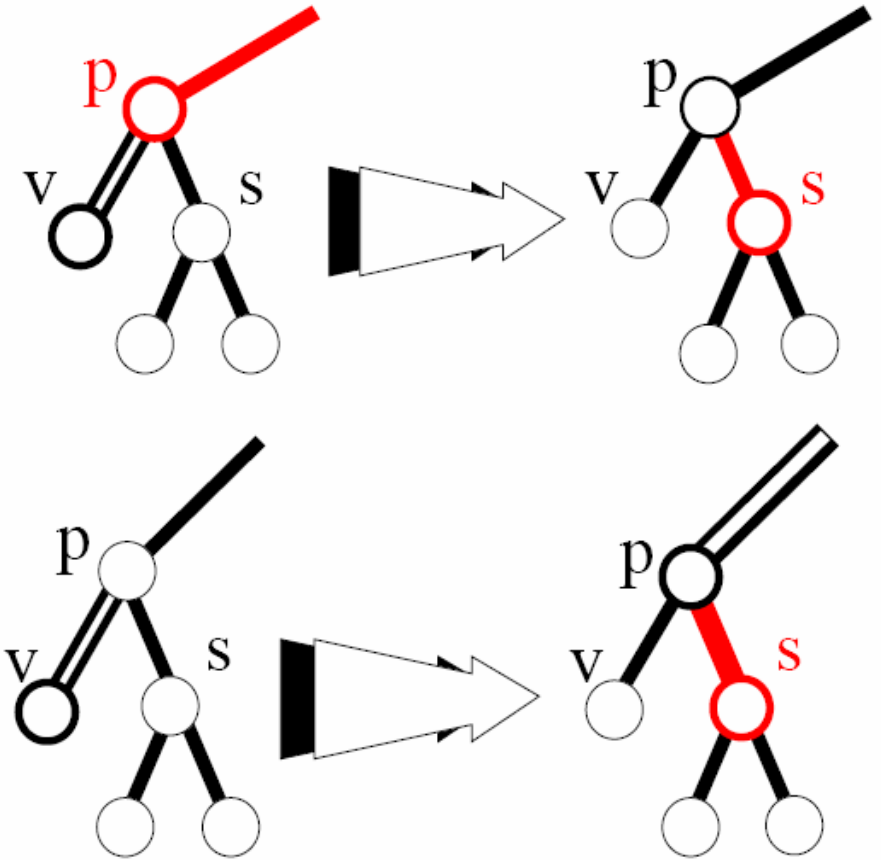




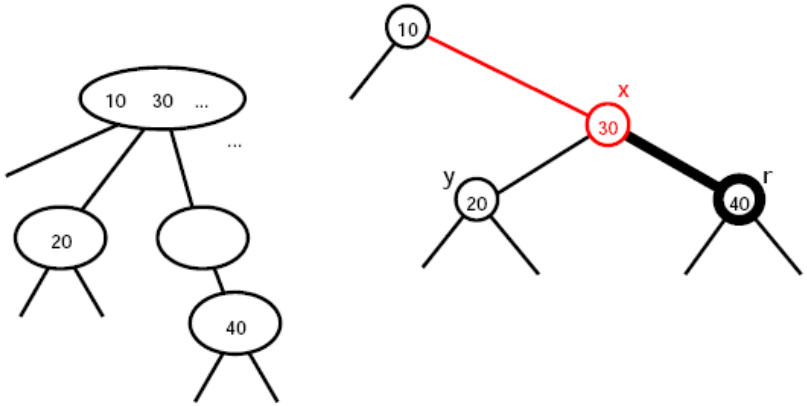


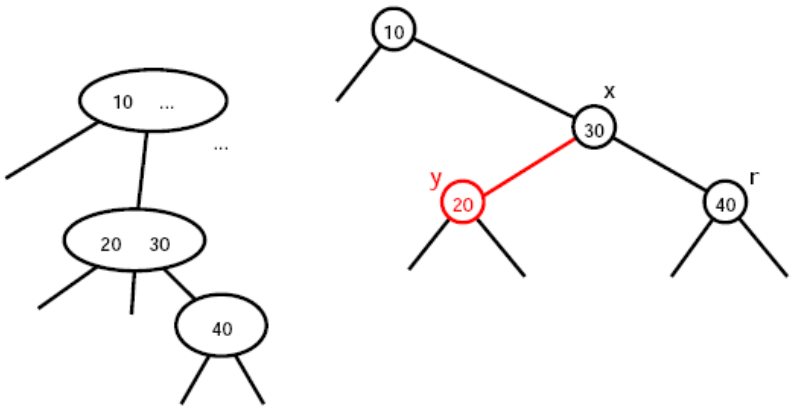
Case 2: black sibling with black childern

- If sibling and its children are **black**, perform a ***recoloring***
- If parent becomes **double black**, ***continue*** upward



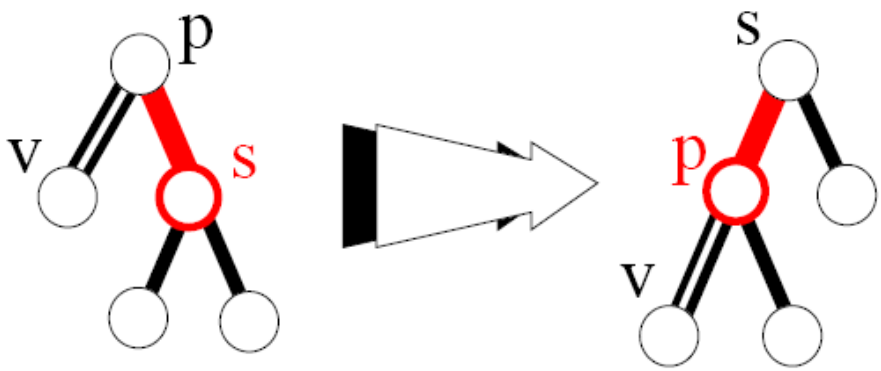
(2,4) Tree Interpretation





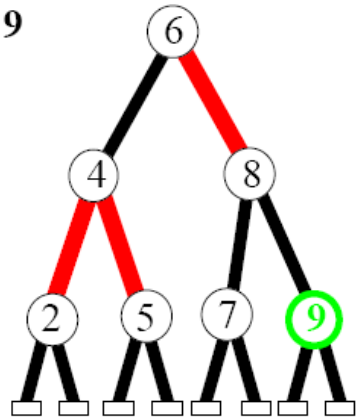
Case 3: red sibling

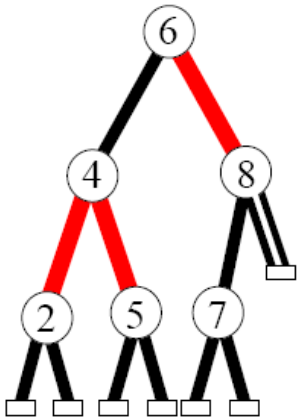
- If sibling is red, perform an ***adjustment***
- Now the sibling is **black** and one the of previous cases applies
- If the next case is recoloring, there is no propagation upward (parent is now red)



How About an Example?

Remove 9





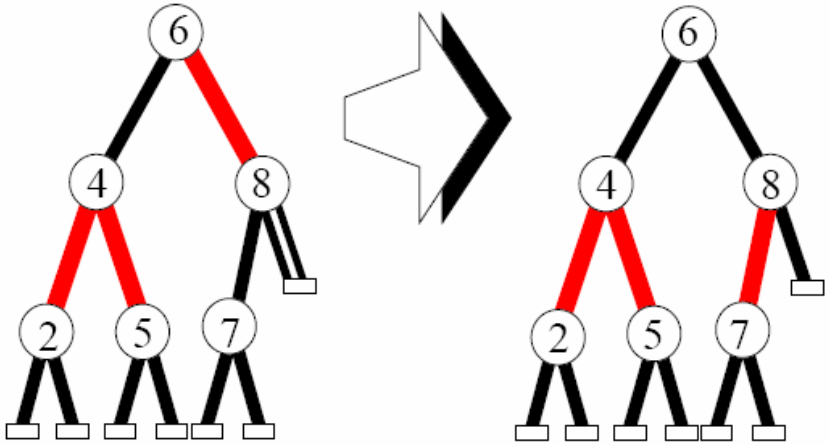
Example

What do we know?

- Sibling is black with black children

What do we do?

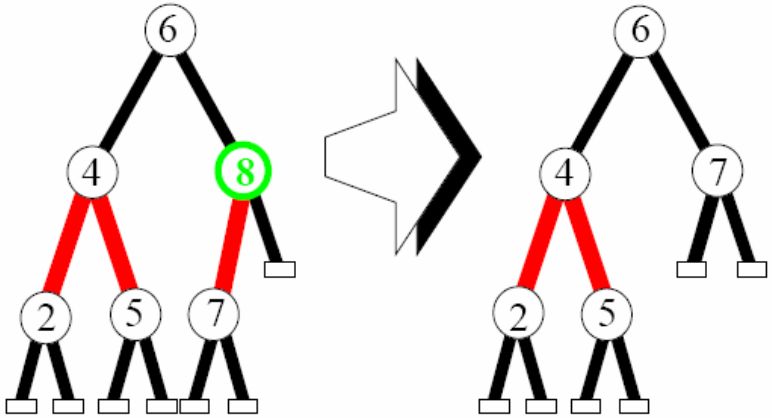
- Recoloring



Example

Delete 8

- no double black



Example

Delete 7

- Restructuring