CS302 - Data Structures
using C++

Topic: Red-Black Trees

Kostas Alexis

CS302 - Data Structures
using C++

Topic: 2-3-4 Trees

Kostas Alexis

2-3-4 Trees

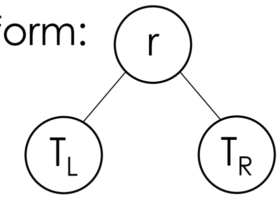
- If a 2-3 tree offers benefits, are trees whose nodes can have more than three children even better?

2-3-4 Trees

- If a 2-3 tree offers benefits, are trees whose nodes can have more than three children even better?
  - To some extent, yes.

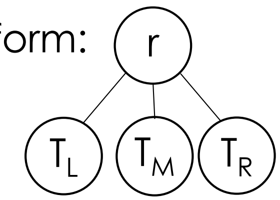
2-3-4 Trees - Definition

- **T** is a 2-3-4 tree of height h if one of the following is true:
  - **T** is empty, in which case h is 0.
  - **T** is of the form:



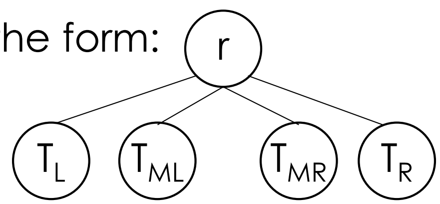
where r is a node that contains one data item and T_L and T_R are both 2-3-4 trees of height h-1. In this case: r must be greater than each item in T_L and smaller than each item in T_R.

  - **T** is of the form:



where r is a node that contains two data items and T_L, T_M and T_R are 2-3-4 trees, each of height h-1. In this case: the smaller item in r must be greater than each item in T_L and smaller than each item in T_M. The larger item in r must be greater than each item in T_M and smaller than each item in T_R.

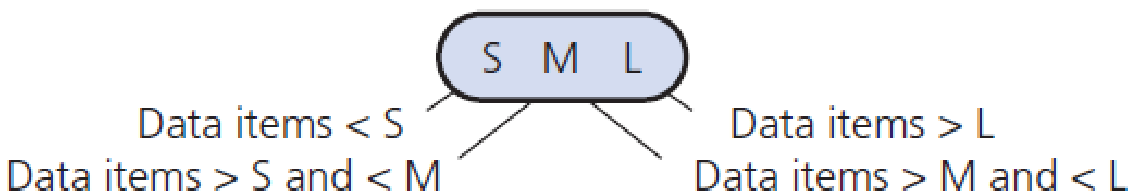
  - **T** is of the form:



where r is a node that contains three data items and T_L, T_{ML}, T_{MR} and T_R are 2-3-4 trees of height h-1. In this case: smallest item in r must be greater than each item in T_L and smaller than each item in T_{ML}. The middle item in r must be greater than each in T_{ML} and smaller than each item in T_{MR}. The largest item in r must be greater than each item in T_{MR} and smaller than T_R.

2-3-4 Trees - Definition

- **Rules for placing data items in the nodes of a 2-3-4 tree**
- The previous definition of a 2-3-4 tree implies the following rules for data placement:
  - A 2-node, which has two children, must contain a single data item that satisfies the relationships as in a 2-3 tree.
  - A 3-node, which has three children, must contain two data items that satisfy the relationships as in a 2-3 tree.
  - A 4-node, which has four children, must contain three data items S, M, and L that satisfy the following relationships: S is greater than the left child’s item(s) and less than the middle-left child’s item(s); M is greater than the middle-left child’s item(s) and less than the middle-right child’s item(s); L is greater than the middle-right child’s item(s) and less than the right child’s item(s).



  - A leaf may contain either one, two, or three data items.

2-3-4 Trees

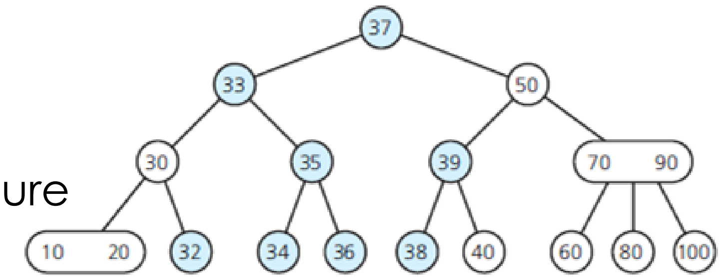
- If a 2-3 tree offers benefits, are trees whose nodes can have more than three children even better?
  - **More efficient addition and removal operations than a 2-3 tree**
  - **Has greater storage requirements due to the additional data members in its 4-nodes**

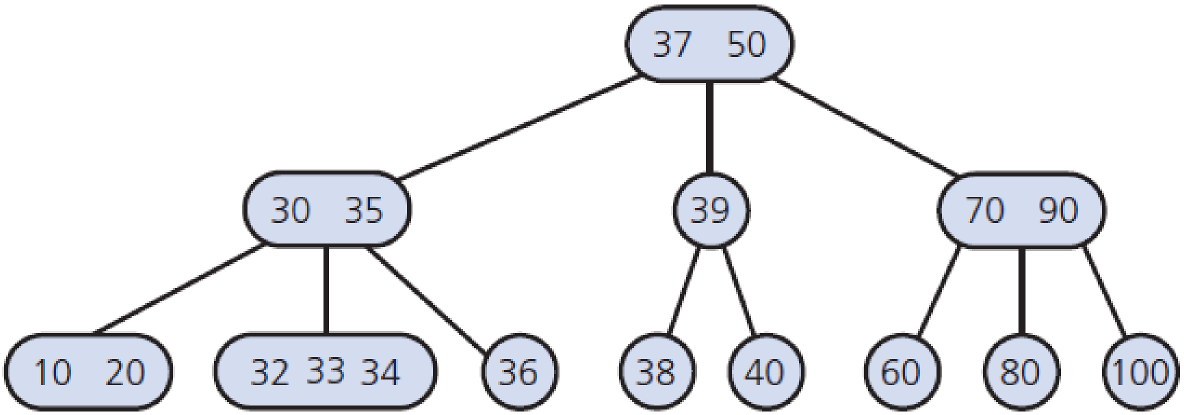
2-3-4 Trees

- If a 2-3 tree offers benefits, are trees whose nodes can have more than three children even better?
  - **More efficient addition and removal operations than a 2-3 tree**
  - **Has greater storage requirements due to the additional data members in its 4-nodes**
  - **However, a 2-3-4 tree can be transformed into a special binary tree that reduces the storage requirements**

2-3-4 Trees

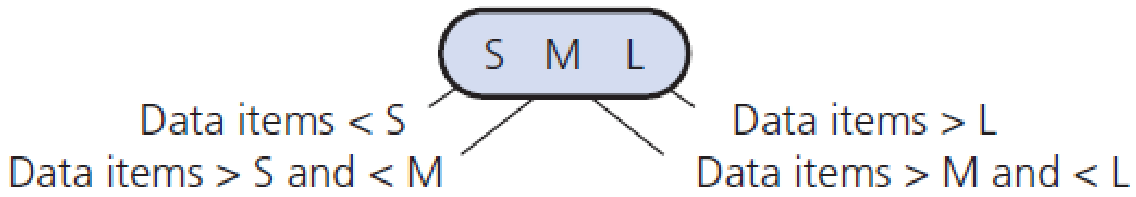
A 2-3-4 tree with the same data items as the 2-3 tree in Figure





2-3-4 Trees

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

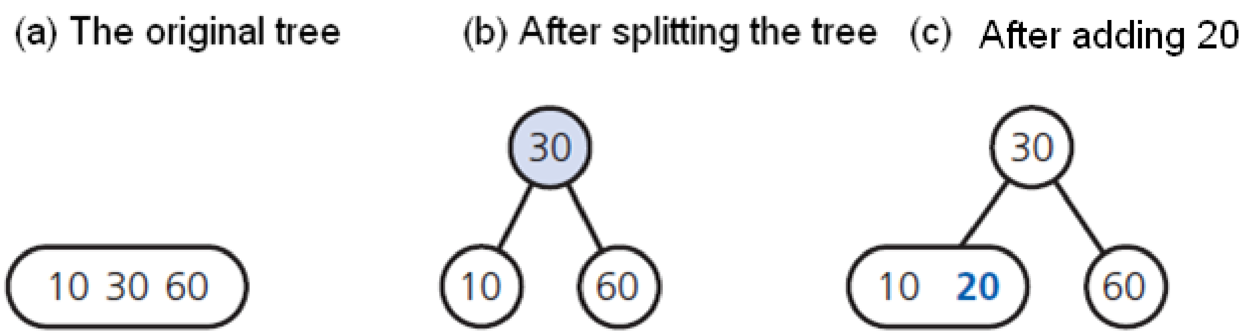


2-3-4 Trees

- Searching and traversing
  - Simple extensions of corresponding algorithms for a 2-3 tree
- Adding data
  - Like addition algorithm for 2-3 tree
  - Splits node by moving one data item up to parent node [**bubble up**]

Adding Data to 2-3-4 Trees

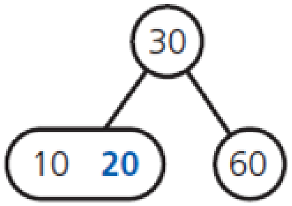
Adding 20 to a one-node 2-3-4 tree

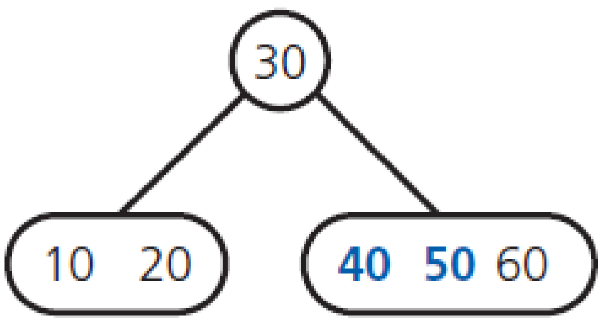


Adding Data to 2-3-4 Trees

After adding 50 and 40 to the tree in Figure

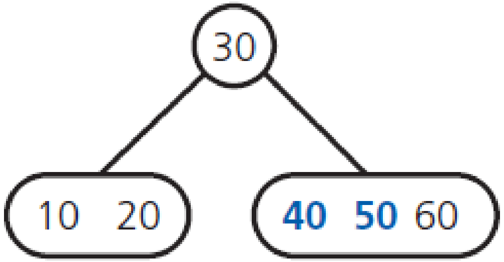
(c) After adding 20

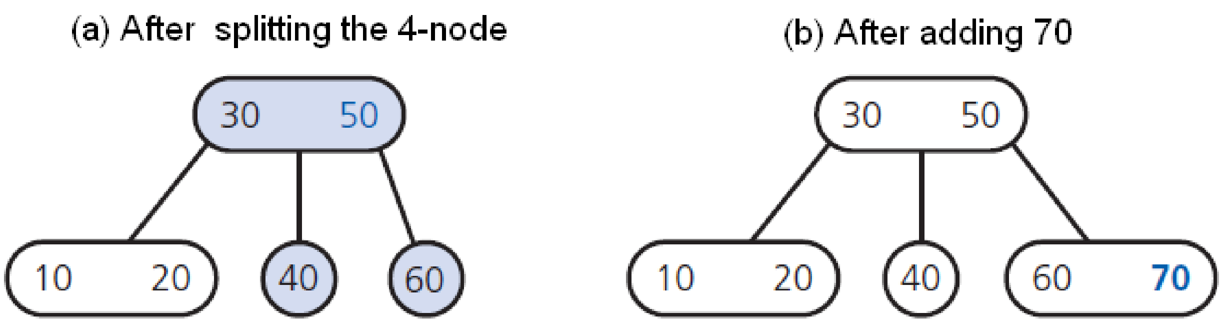




Adding Data to 2-3-4 Trees

The steps for adding 70 to the tree in Figure

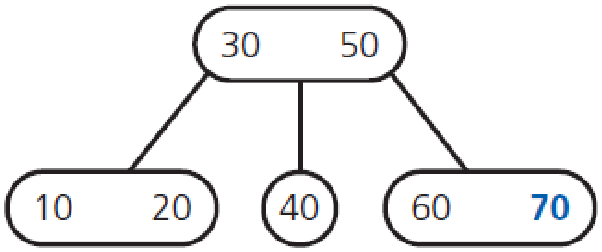


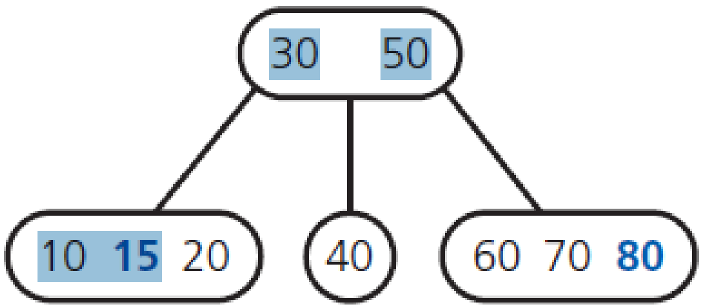


Adding Data to 2-3-4 Trees

After adding 80 and 15 to the tree in Figure

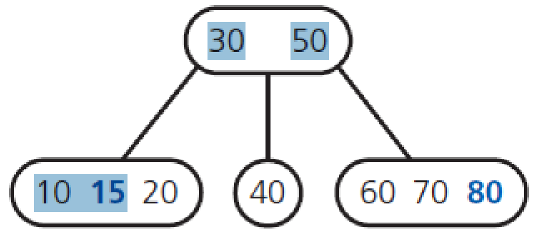
(b) After adding 70



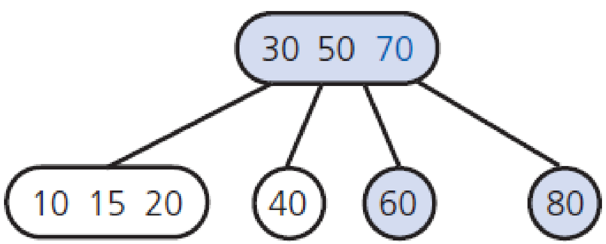


Adding Data to 2-3-4 Trees

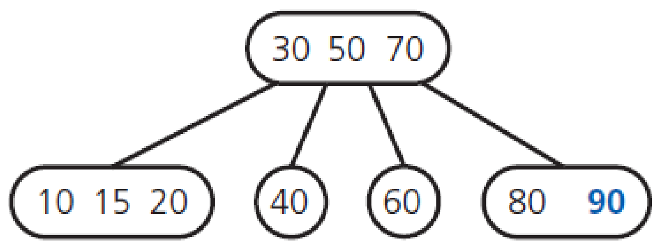
The steps for adding 90 to the tree in Figure



(a) After splitting the root's right child



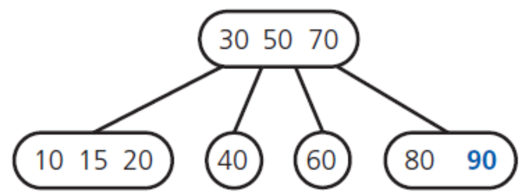
(b) After adding 90 to the root's right child



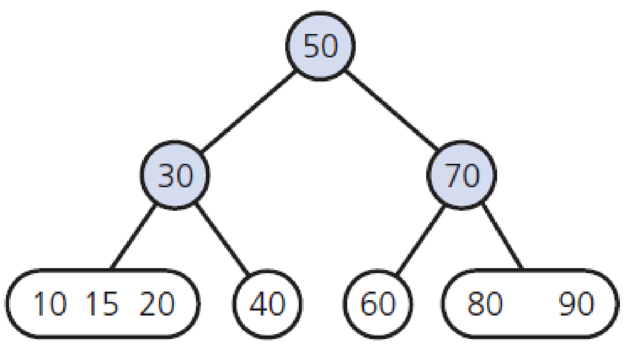
Adding Data to 2-3-4 Trees

The steps for adding 100 to the tree in Figure

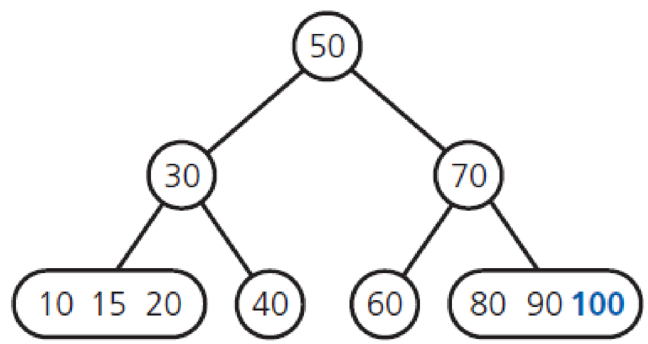
(b) After adding 90 to the root's right child



(a) After splitting the 4-node



(b) After adding 100 to the rightmost leaf



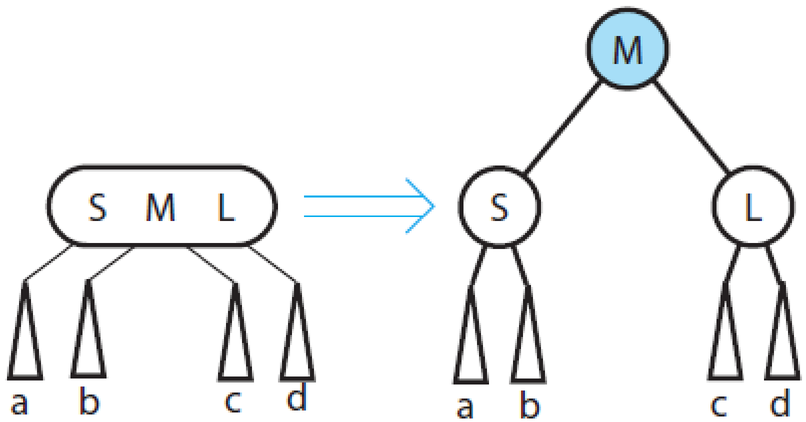
Adding Data to 2-3-4 Trees

Splitting a 4-node root when adding data to a 2-3-4 tree

- The practice is to split each 4-node as soon as it is encountered during the search from the root to the leaf that will accommodate the additional data item.
- As a result, each 4-node either will:
  - Be the root,
  - Have a 2-node parent, or
  - Have a 3-node parent

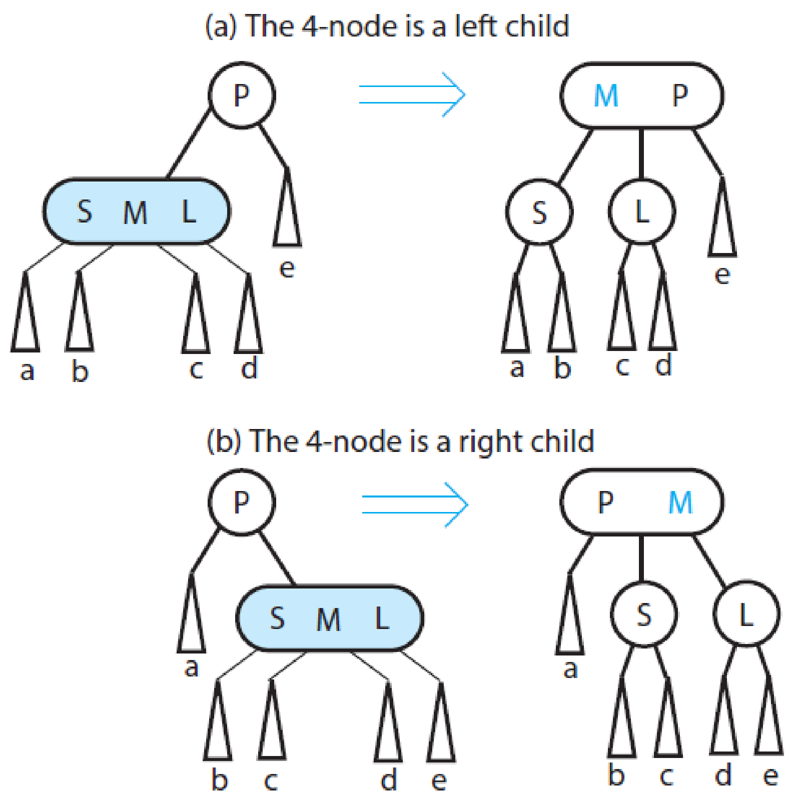
Adding Data to 2-3-4 Trees

Splitting a 4-node root when adding data to a 2-3-4 tree



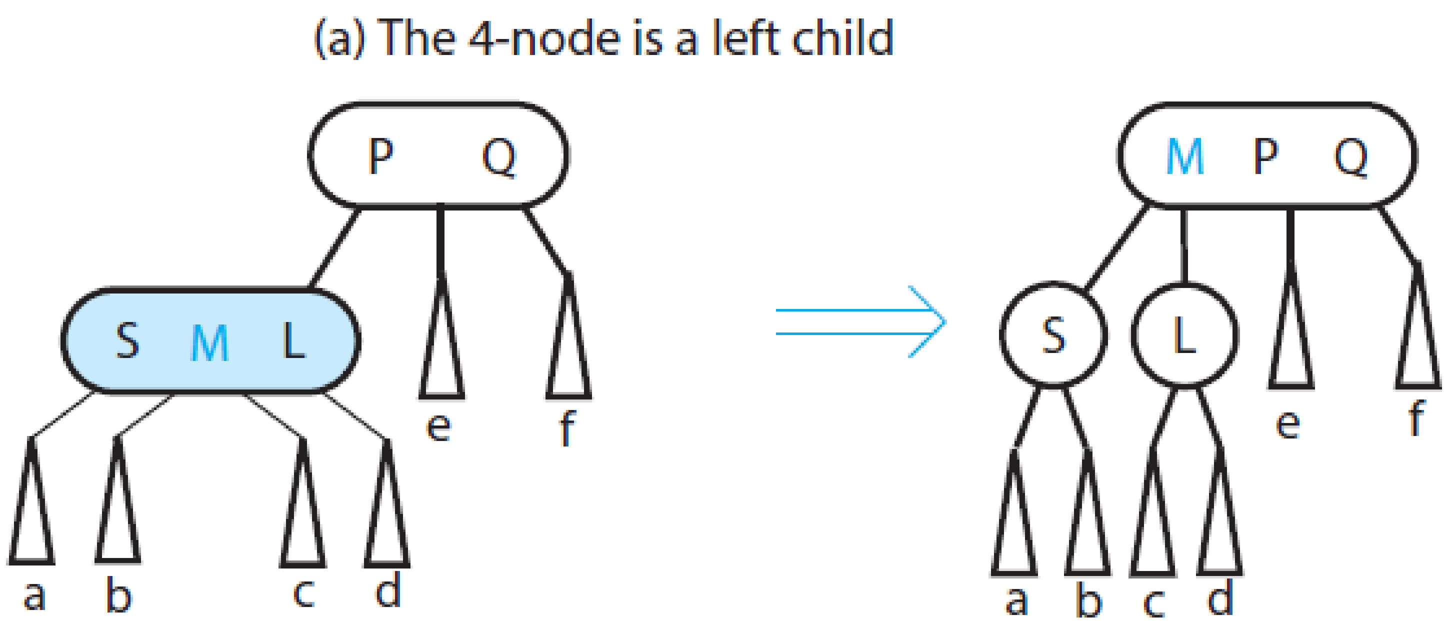
Adding Data to 2-3-4 Trees

Splitting a 4-node whose parent is a 2-node when adding data to a 2-3-4 tree



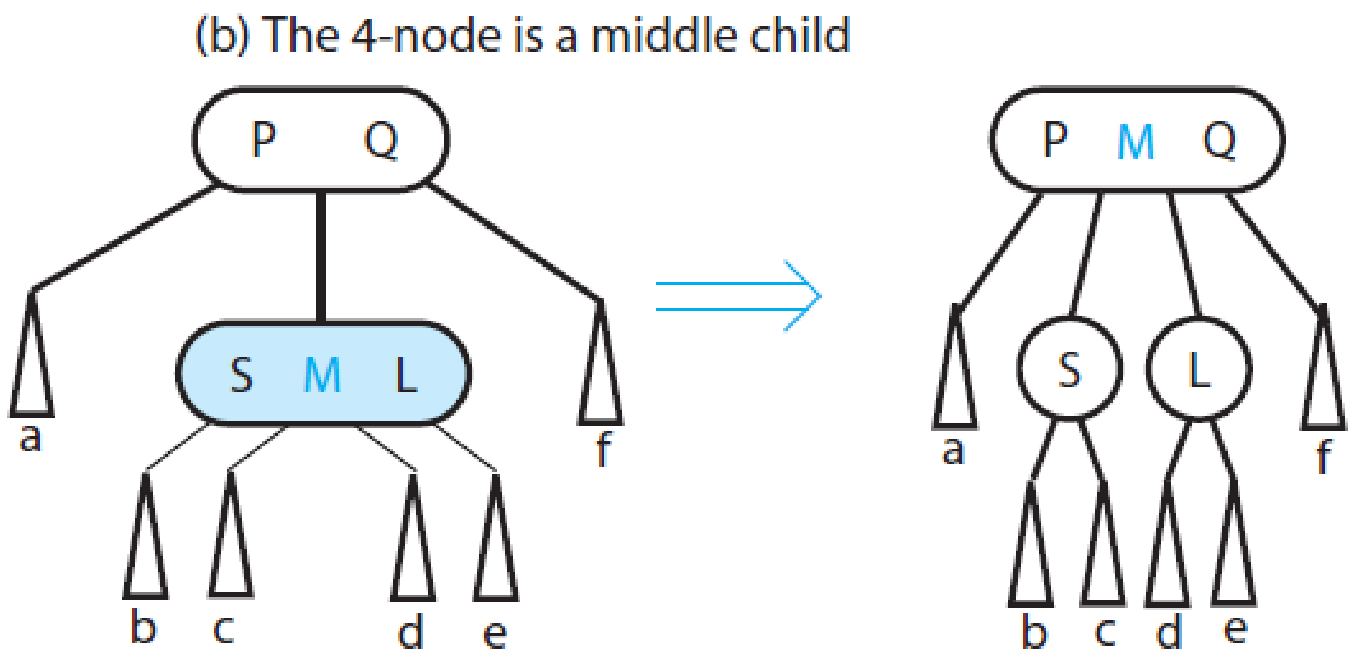
Adding Data to 2-3-4 Trees

Splitting a 4-node whose parent is a 3-node when adding data to a 2-3-4 tree



Adding Data to 2-3-4 Trees

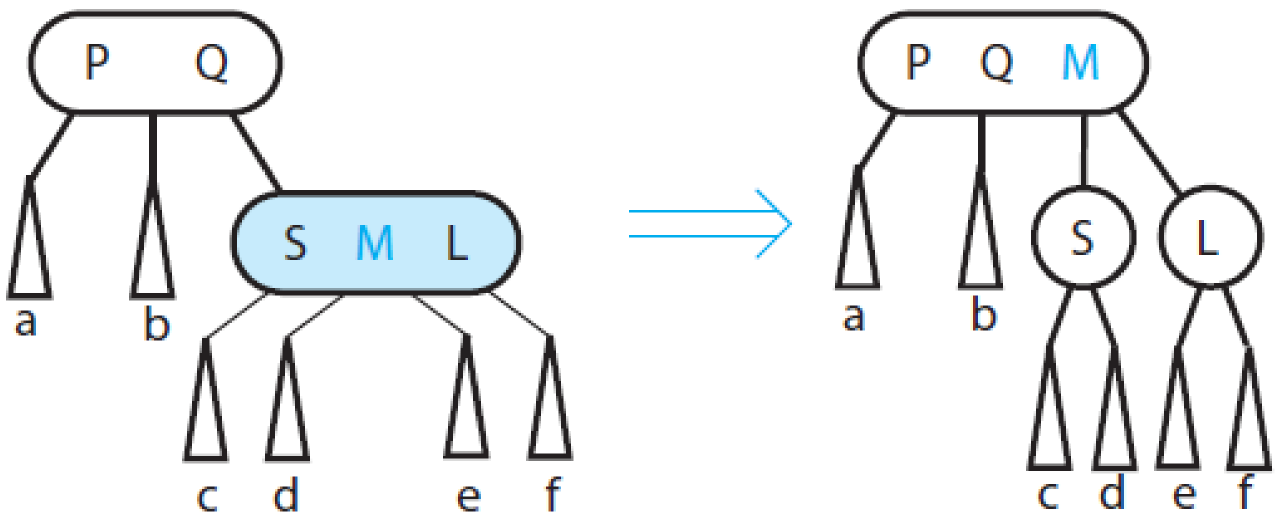
[Continued]



Adding Data to 2-3-4 Trees

[Continued]

(c) The 4-node is a right child



Removing Data from a 2-3-4 Tree

- Has same beginning as removal algorithm for a 2-3 tree
- Transform each 2-node into a 3-node or a 4-node
- Insertion and removal algorithms for 2-3-4 tree require fewer steps than for 2-3 tree

CS302 - Data Structures
using C++

Topic: Red-Black Trees

Kostas Alexis

Red-Black Trees

- A 2-3-4 tree is efficient with respect to addition and removal operations but requires more storage than binary search tree

Red-Black Trees

- A 2-3-4 tree is efficient wrt to addition and removal operations but requires more storage than binary search tree
- **Red-black** tree has the advantages of a 2-3-4 tree but requires less storage

Red-Black Trees

- A 2-3-4 tree is efficient wrt to addition and removal operations but requires more storage than binary search tree
- **Red-black** tree has the advantages of a 2-3-4 tree but requires less storage
- In a **red-black** tree,

Red-Black Trees

- A 2-3-4 tree is efficient wrt to addition and removal operations but requires more storage than binary search tree
- **Red-black** tree has the advantages of a 2-3-4 tree but requires less storage
- In a **red-black** tree,
  - **Red** pointers link 2-nodes that now contain values that were in a 3-node or a 4-node

Red-Black Trees

- A 2-3-4 tree is efficient wrt to addition and removal operations but requires more storage than binary search tree
- **Red-black** tree has the advantages of a 2-3-4 tree but requires less storage
- In a **red-black** tree,
  - **Red** pointers link 2-nodes that now contain values that were in a 3-node or a 4-node
  - A **red** pointer references a **red** node

Red-Black Trees

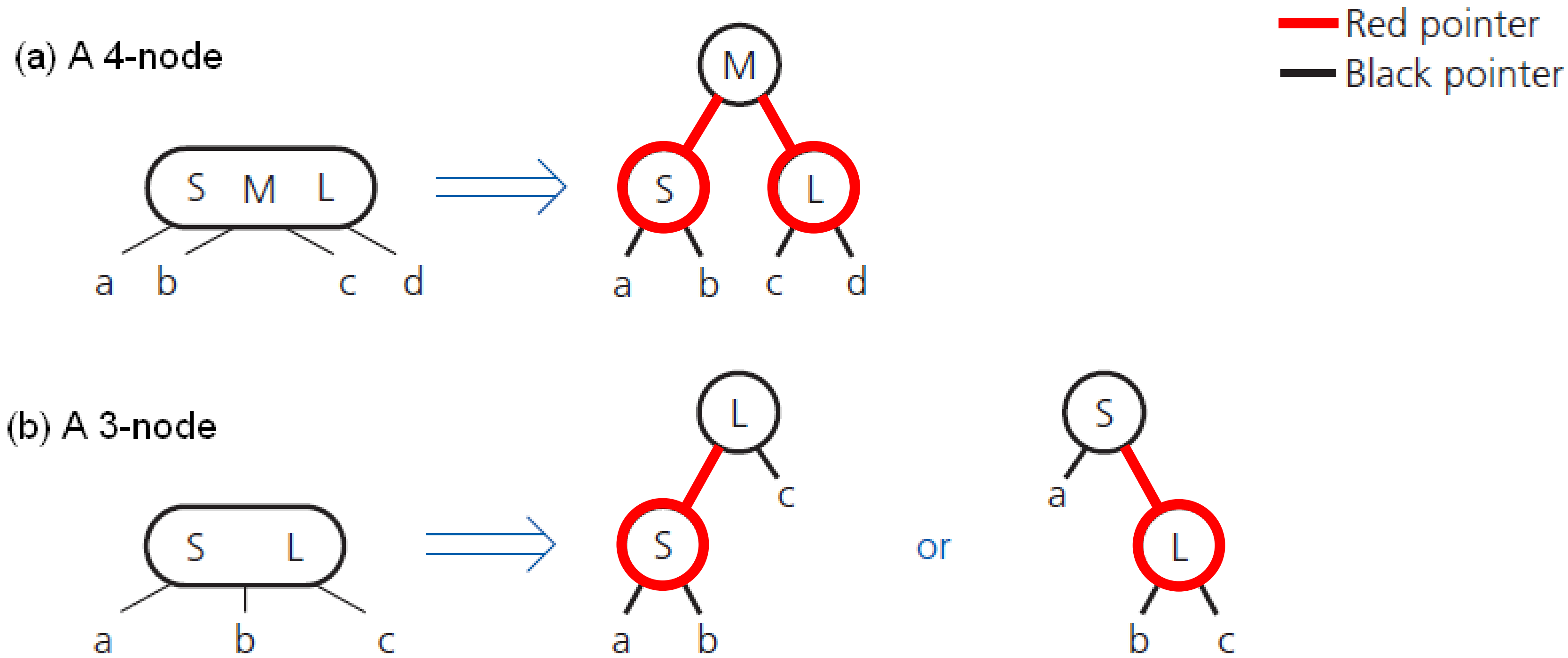
- A 2-3-4 tree is efficient wrt to addition and removal operations but requires more storage than binary search tree
- **Red-black** tree has the advantages of a 2-3-4 tree but requires less storage
- In a **red-black** tree,
  - **Red** pointers link 2-nodes that now contain values that were in a 3-node or a 4-node
  - A **red** pointer references a **red** node
  - A **black** pointer references a **black** node

Red-Black Trees

- Represent 2-3-4 tree as a BST
- Use "internal" red edges for 3- and 4-nodes

Red-Black Trees

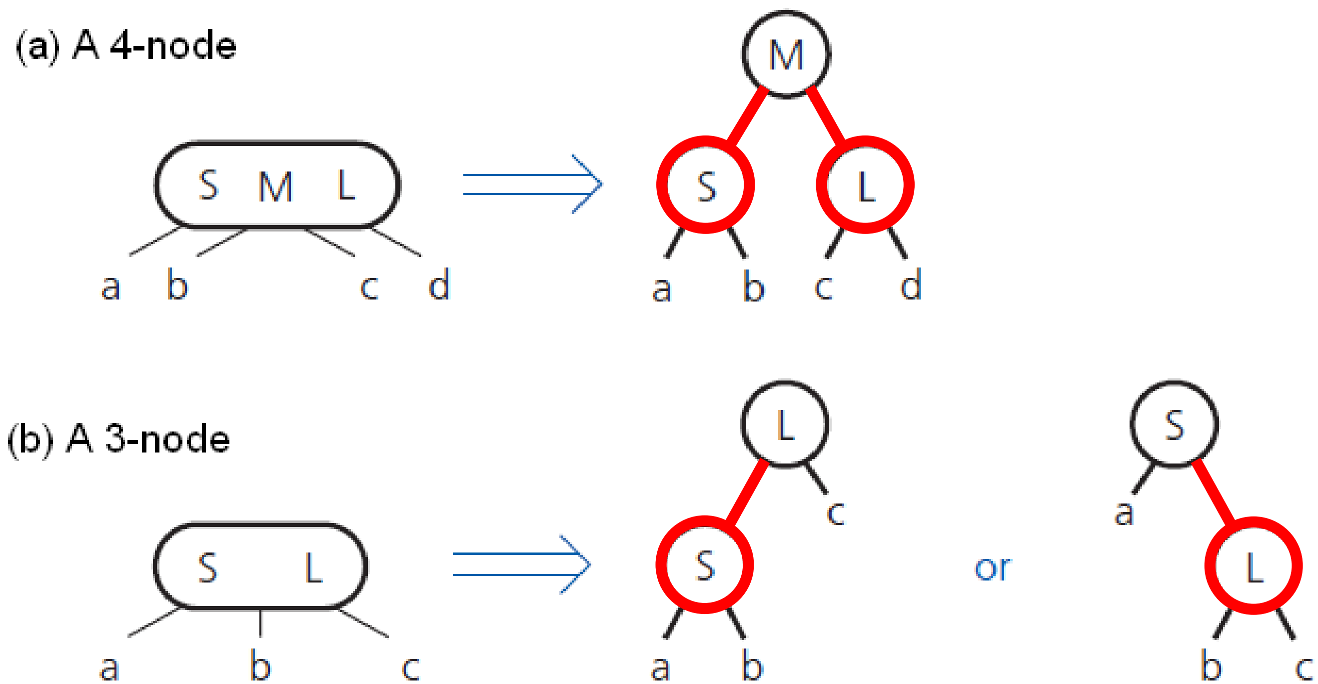
Red-black representations of a 4-node and a 3-node



Red-Black Trees

Red-black representations of a 4-node and a 3-node

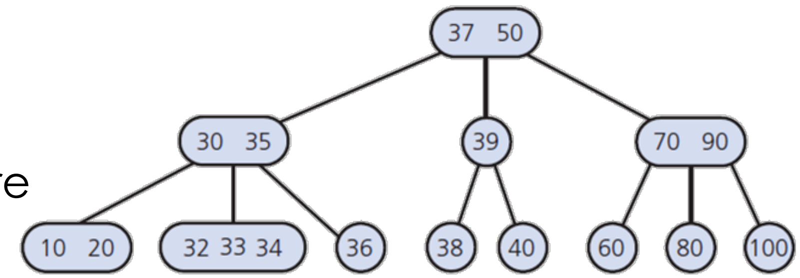
Red pointer
Black pointer

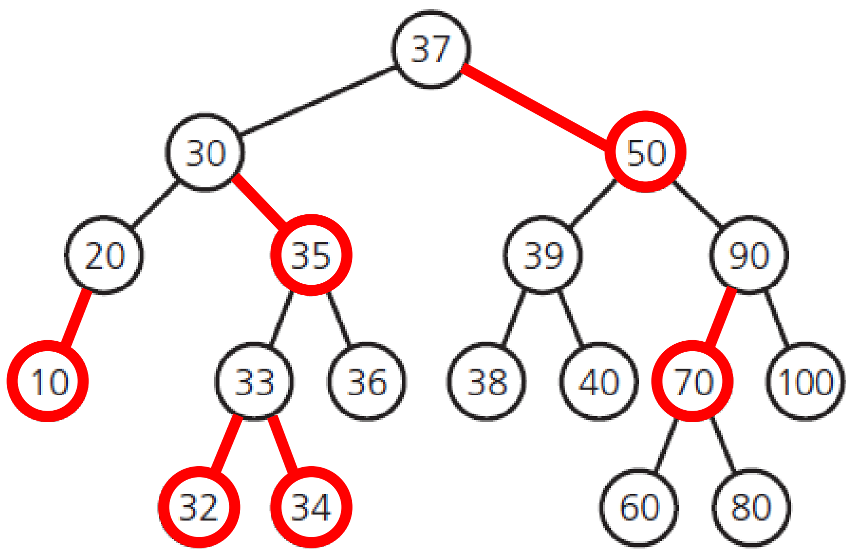


Representation of a 3-node is non-unique

Red-Black Trees

A red-black tree that represents the 2-3-4 tree in Figure



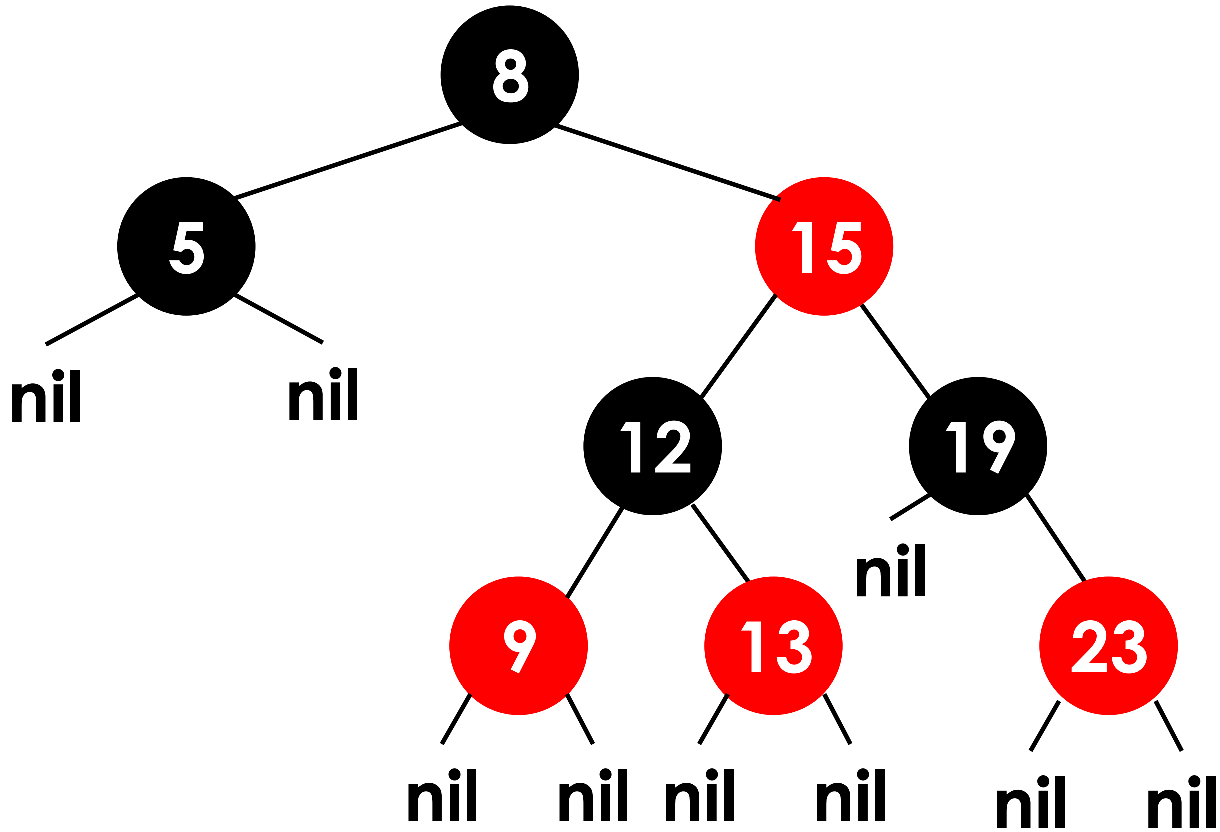


Red-Black Trees

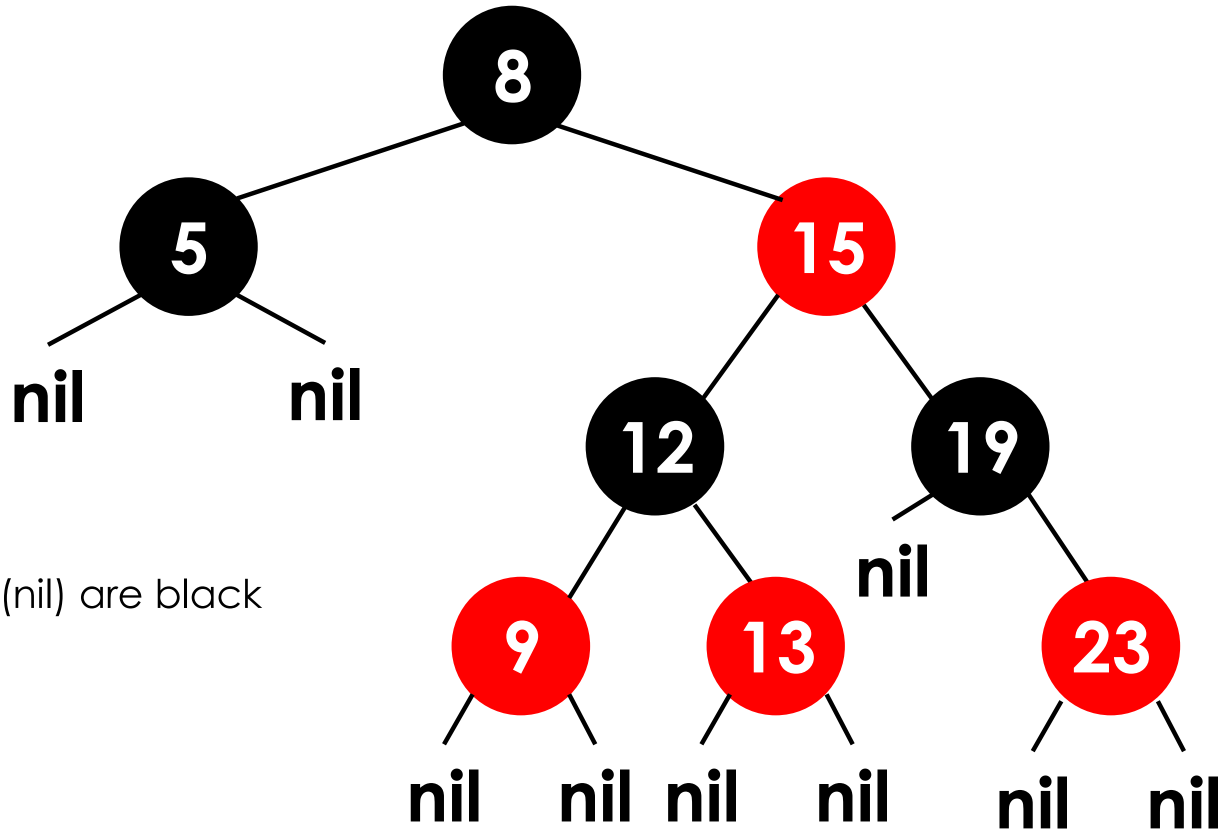
Properties of a Red-Black Tree:

- The root is **black**
- Every **red** node has a **black** parent
- Any children of a **red** node are **black**; that is, a **red** node cannot have **red** children
- Every path from the root to a leaf contains the same number of **black** nodes

Red-Black Trees

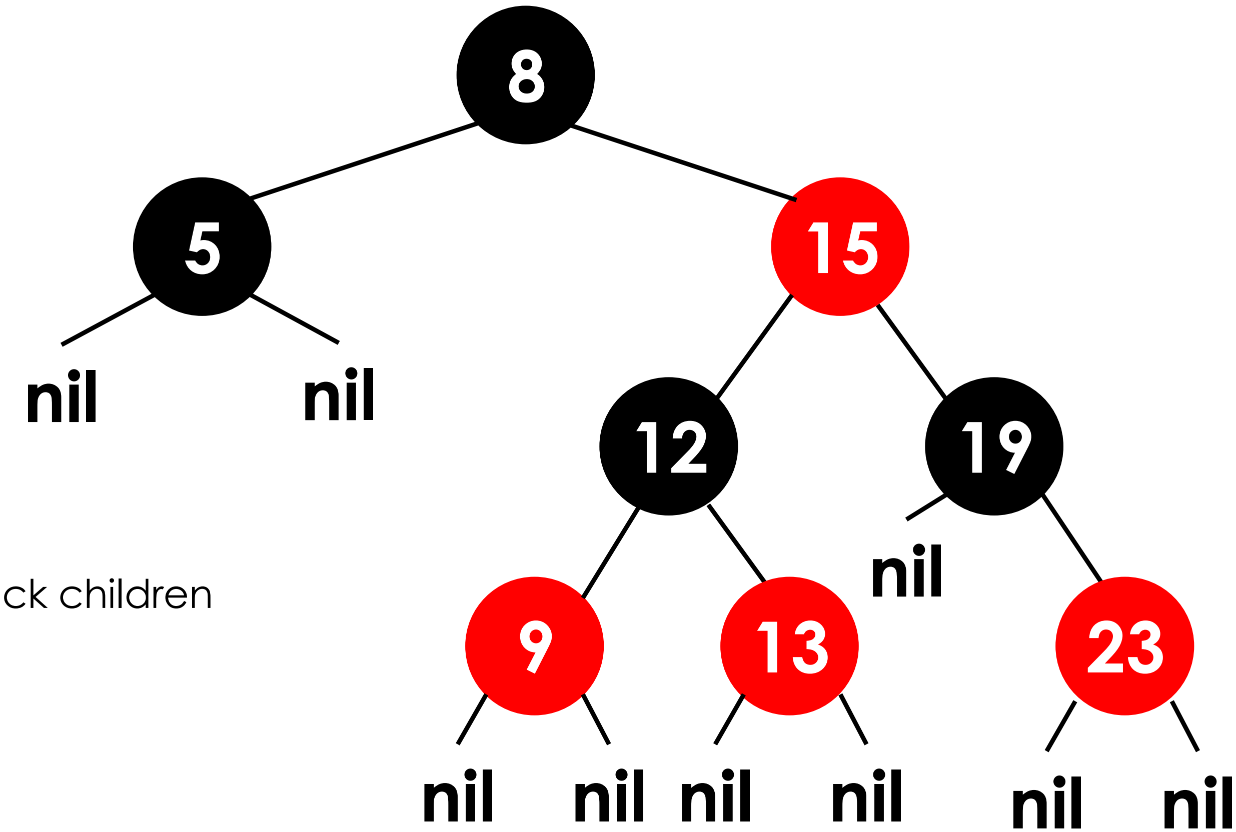


Red-Black Trees



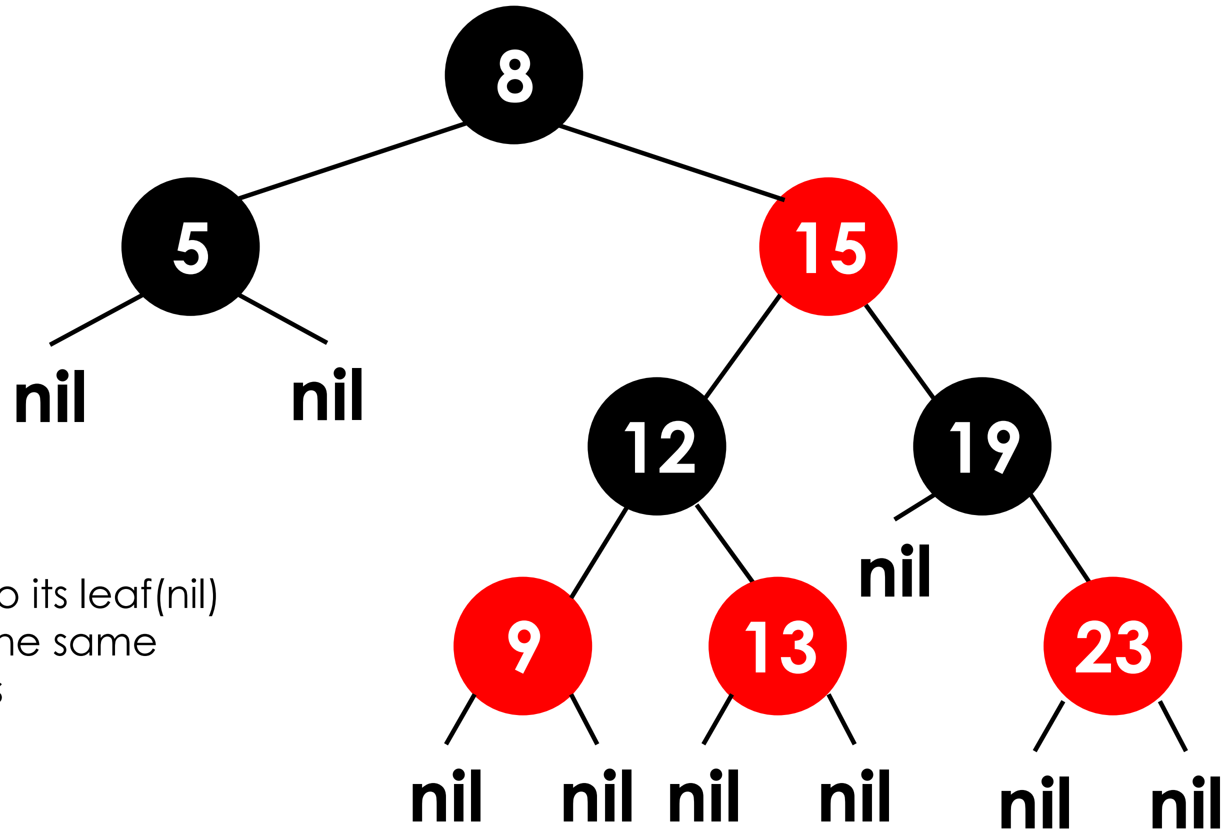
The root and all leaves (nil) are black

Red-Black Trees



All red nodes have black children

Red-Black Trees



All paths from a node to its leaf(nil) descendants contain the same number of black nodes

Red-Black Trees

We derive the class of Red-Black nodes from the class BinaryNode

```
enum Color {RED, BLACK};

template<class ItemType>
class RedBlackNode : public BinaryNode<ItemType>
{
private:
    Color leftColor;
    Color rightColor;
public:
    // Get and set methods for leftColor and rightColor
    // …

} // end RedBlackNode
```

Red-Black Trees

Further properties

- Nodes require at a minimum one storage bit to keep track of color
- The longest path (root to furthest leaf) is no more than twice the length of the shortest path (root to nearest leaf)
  - Shortest path: all **black** nodes
  - Longest path: alternating between **red** and **black** nodes

Searching and Traversing a Red-Black Tree

- A red-black tree is a binary search tree
- Thus, **search** and **traversal**
  - Use algorithms for binary search tree
  - Simply ignore color of pointers
  - **Code may not change at all!**

Adding to and Removing from a Red-Black Tree

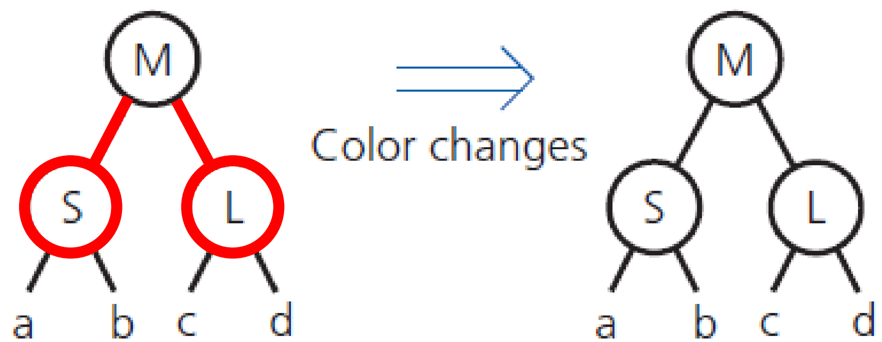
- **Red-black tree represents a 2-3-4 tree**
  - Simply adjust 2-3-4 addition algorithms
  - Accommodate red-black representation
- **Splitting equivalent of a 4-node requires simple color changes**
  - Pointer changes called **rotations** result in a shorter tree

Adding to and Removing from a Red-Black Tree

- When adding a new node, the Red-Black Tree properties must be maintained.

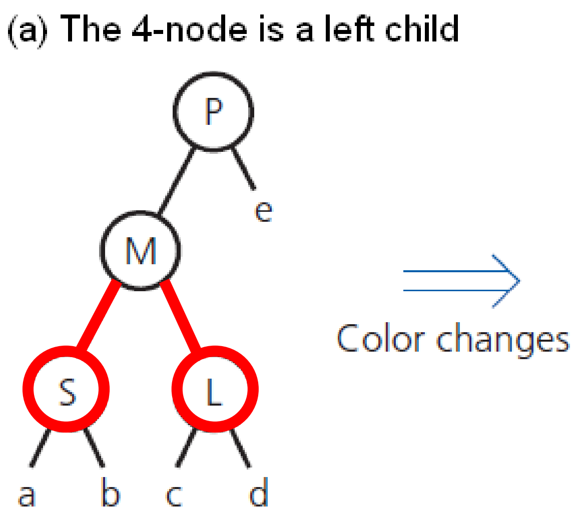
Adding to and Removing from a Red-Black Tree

Case 1: Splitting a red-black representation of a 4-node root



Adding to and Removing from a Red-Black Tree

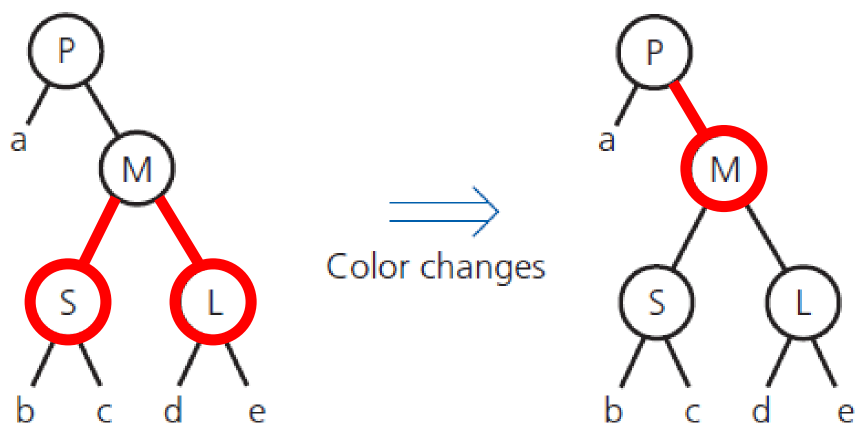
Case 2: Splitting a red-black representation of a 4-node whose parent is a 2-node



Adding to and Removing from a Red-Black Tree

[Continued]

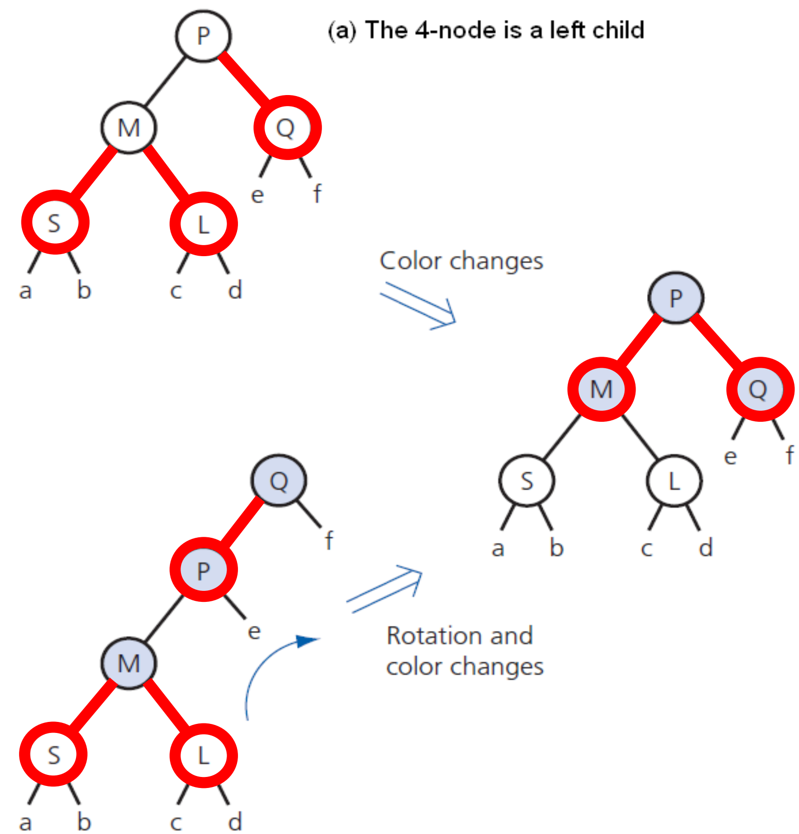
(b) The 4-node is a right child



Adding to and Removing from a Red-Black Tree

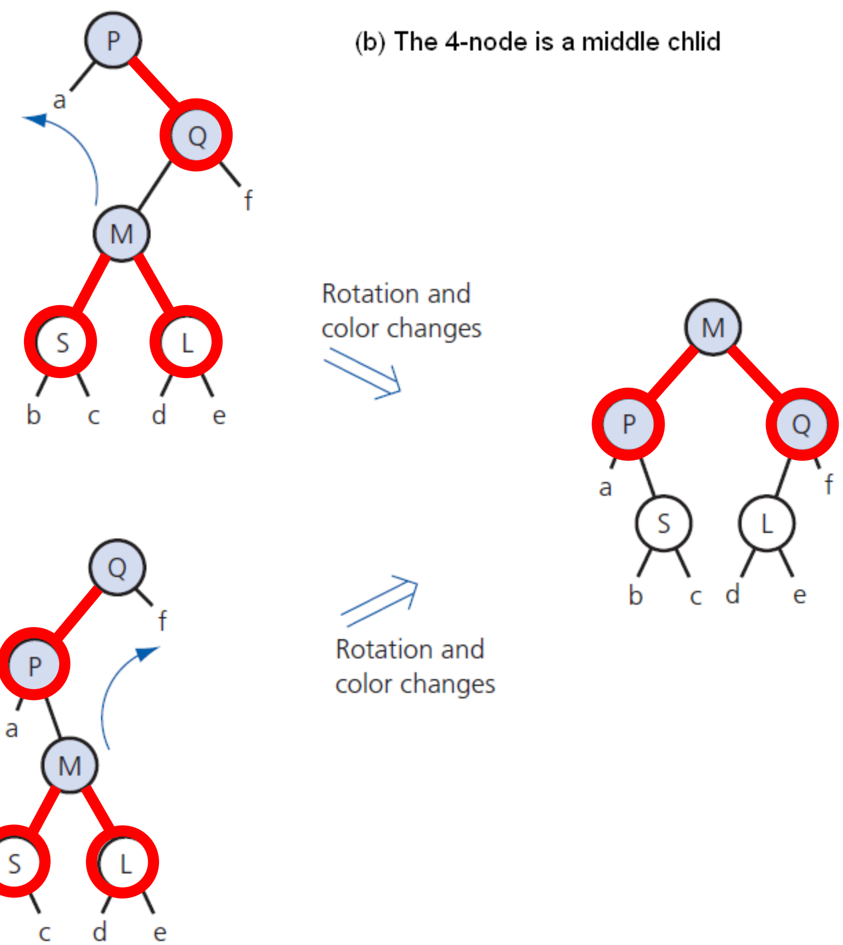
Case 3: Splitting a red-black representation of a 4-node whose parent is a 3-node

(a) The 4-node is a left child



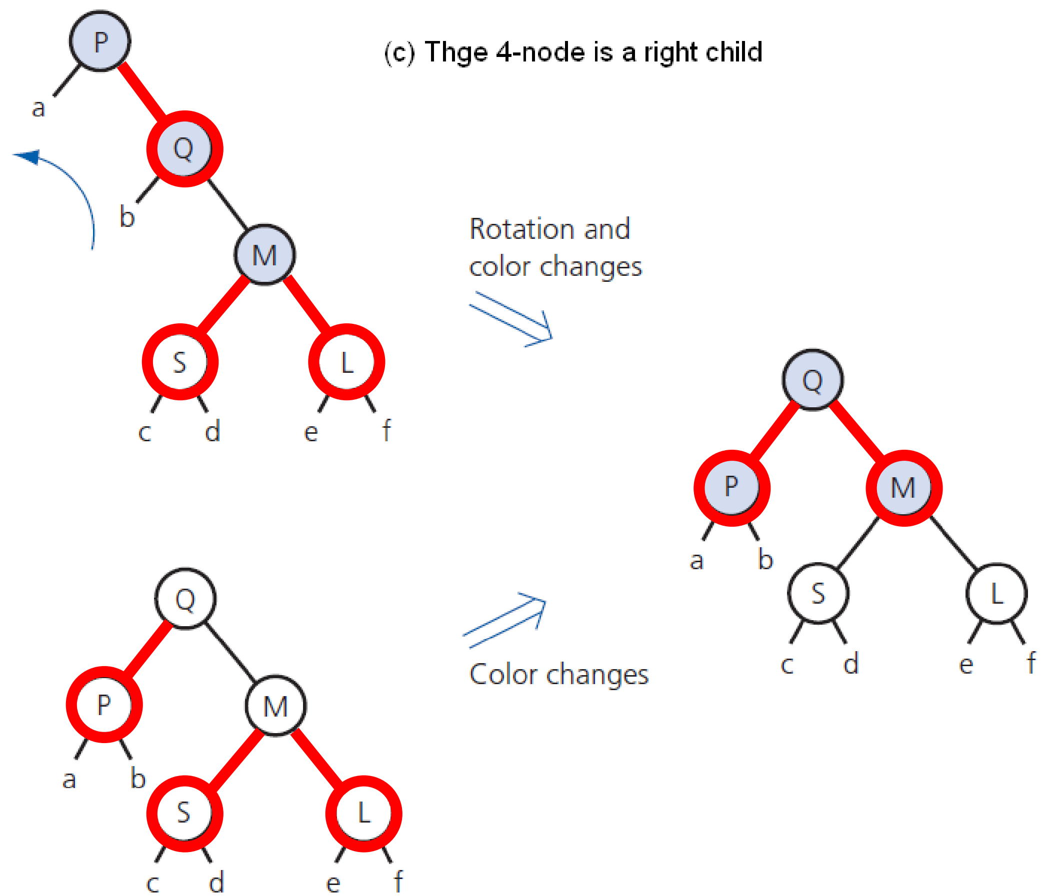
Adding to and Removing from a Red-Black Tree

[Continued]



Adding to and Removing from a Red-Black Tree

[Continued]



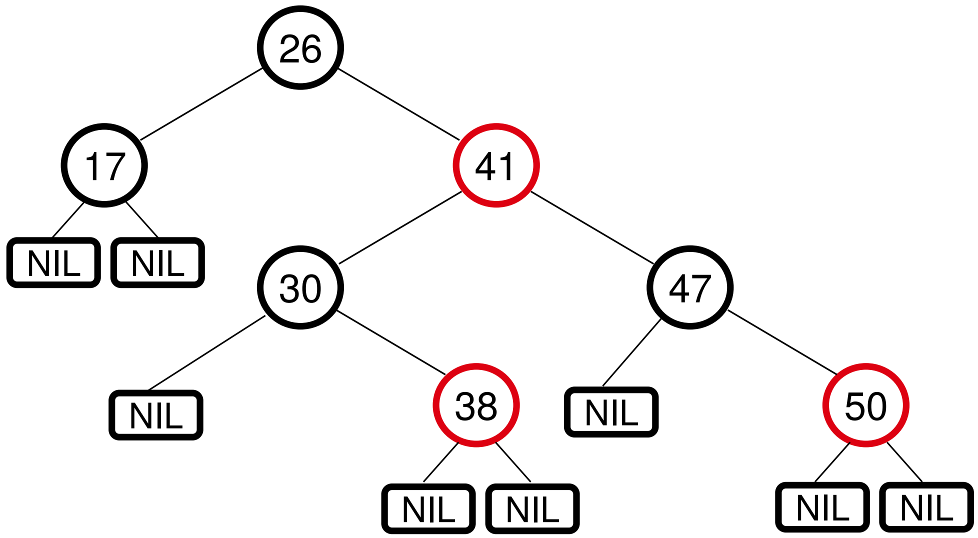
Red-Black Trees

Properties re-written

- Every node is either **red** or **black**
- The root is **black**
- Every leaf (NIL) is **black**
- If a node is **red**, then both its children are **black**
  - No two consecutive **red** nodes on a simple path from the root to a leaf
- For each node, all paths from that node to a leaf contain the same number of **black** nodes

Red-Black Trees

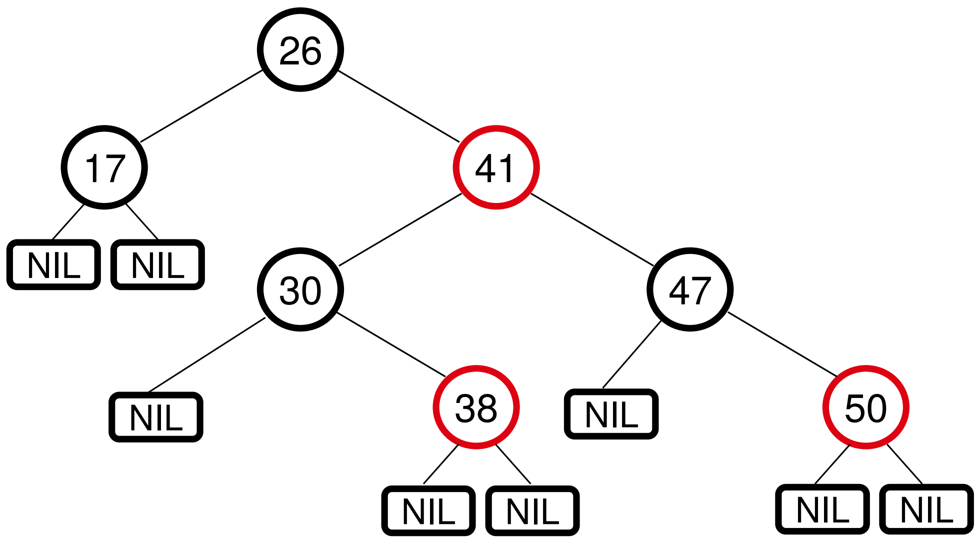
An example



- For convenience, we add NIL nodes and refer to them as the leaves of the tree (Color[NIL]=**Black**)

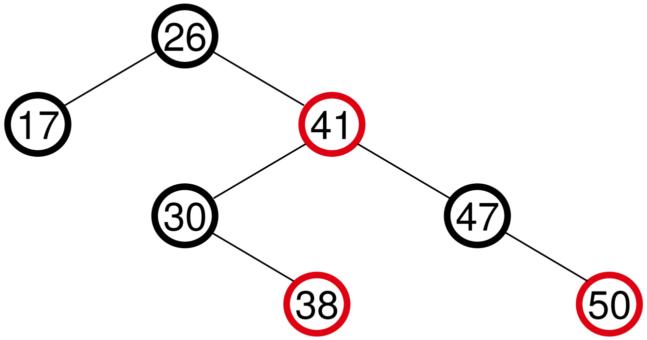
Red-Black Trees

- **Definitions**



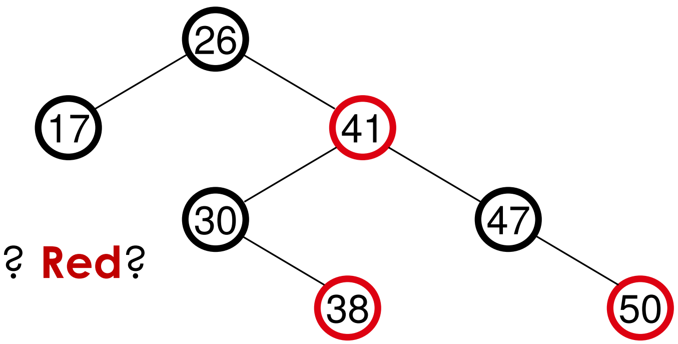
- **Height of a node =** the number of edges in the longest path to a leaf
- **Black-height bh(x) of a node x =** the number of black nodes (including NIL on the path from x to a leaf, not counting x)

Red-Black Trees



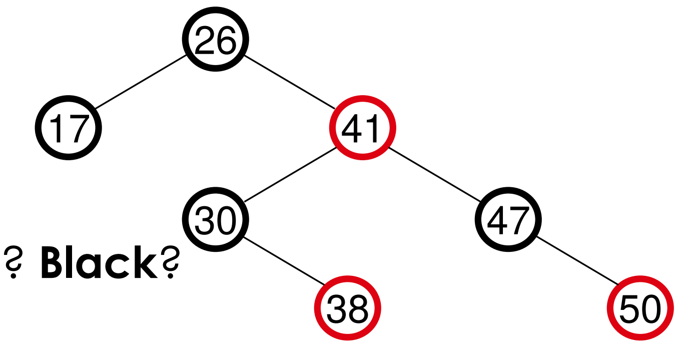
- **Addition (Insertion)**
  - What color to make the new node?
    - **Red**?
      - Let's insert 35
        - Property 4 is violated: *if a node is red, then both children are black*
    - **Black**?
      - Let's insert 14
        - Property 5 is violated: *all paths from a node to its leaves contain the same number of black nodes*

Red-Black Trees



- **Deletion of item**
  - What color was the node that was removed? **Red**?
    - Every node is either **red** or **black** (**OK**)
    - The root is **black** (**OK**)
    - Every leaf (NIL) is **black** (**OK**)
    - If a node is **red**, then both its children are **black** (**OK**)
    - For each node, all paths from the node to descendant leaves contain the same number of **black** nodes (**OK**)

Red-Black Trees



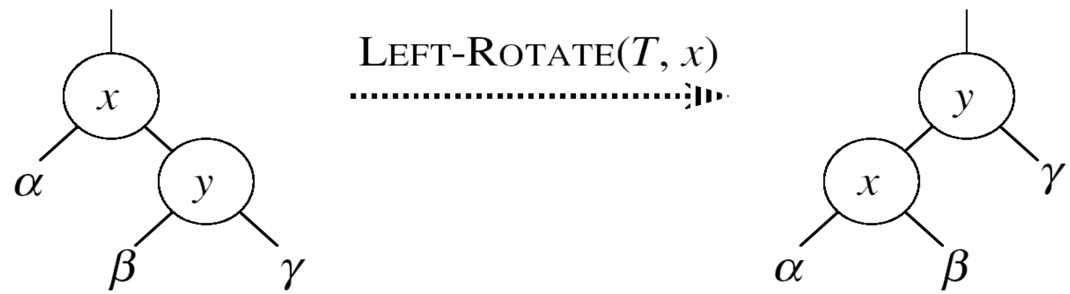
- **Deletion of item**
  - What color was the node that was removed? **Black**?
    - Every node is either **red** or **black** (**OK**)
    - The root is **black** (**NOT OK!** If removing the root and the child that replaces it is red)
    - Every leaf (NIL) is **black** (**OK**)
    - If a node is **red**, then both its children are **black** (**Not OK!** Could create two red nodes in a row)
    - For each node, all paths from the node to descendant leaves contain the same number of **black** nodes (**Not OK!** Could change the black heights of some nodes)

Red-Black Trees

- **Rotations**
  - Operations for re-structuring the tree after insert and delete operations
    - Together with some node re-coloring they help restore the red-black tree property
    - Change some of the pointer structure
    - Preserve the binary search-tree property
  - Two types of rotations
    - Left & right rotations

Red-Black Trees

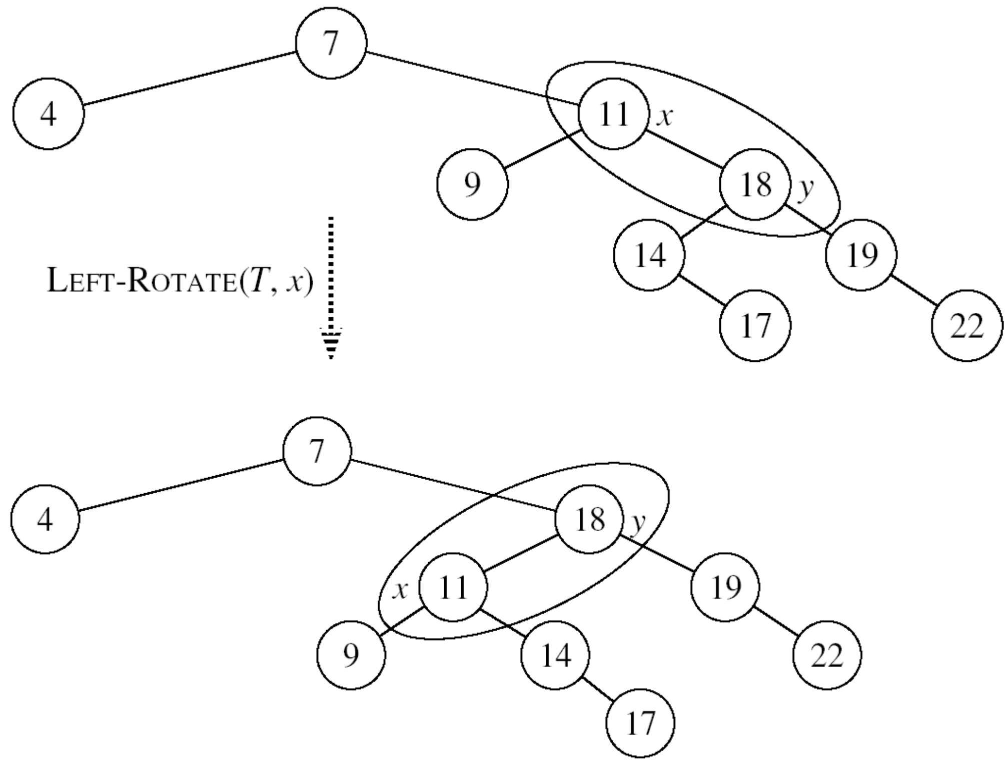
- **Left Rotations**
  - Assumptions for a left rotation on a node x
    - The right child y of x is not NIL



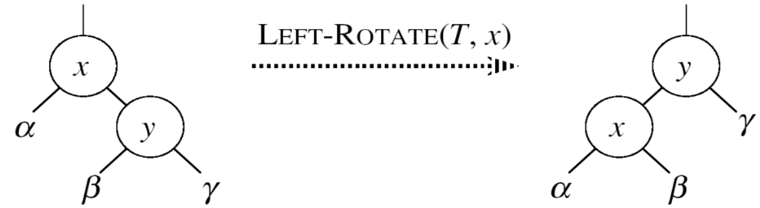
  - **Idea**
    - Pivots around the link from **x** to **y**
    - Makes **y** the new root of the subtree
    - **x** becomes **y**'s left child
    - **y**'s left child becomes **x**'s right child

Red-Black Trees

- **Left Rotations: Example**



Red-Black Trees

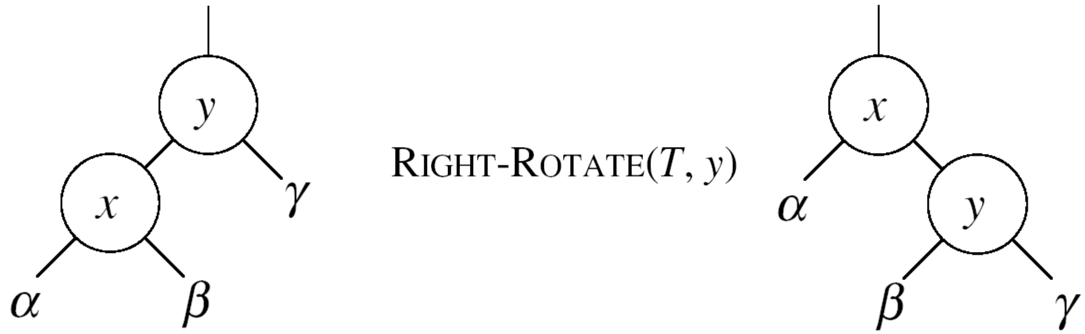


- **Left-Rotate(T,x)**

```
y ← right[x]                          // Set y
right[x] ← left[y]     // y's left subtree becomes x's right subtree
if left[y] ≠ NIL
    then p[left[y]] ← x // Set the parent relation from left[y] to x
p[y] ← p[x]               // The parent of x becomes the parent of y
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
                then left[p[x]] ← y
                else right[p[x]] ← y
left[y] ← x           // Put x on y's left
p[x] ← y                       // y becomes x's parent
```

Red-Black Trees

- **Right Rotations**
  - Assumptions for a right rotation on a node x
    - The right child x of y is not NIL

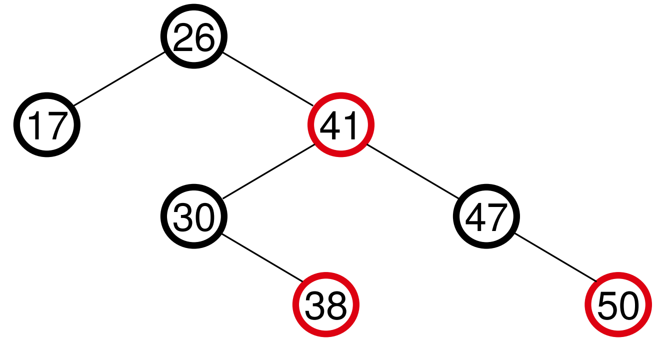


  - Idea
    - Pivots around the link from **y** to **x**
    - Makes **x** the new root of the subtree
    - **y** becomes **x**'s right child
    - **x**'s right child becomes **y**'s left child

Red-Black Trees

- **Add (Insert) Item**
  - **Goal**
    - Insert a new node **z** into a red-black tree
  - **Idea**
    - Insert node **z** into the tree as for an ordinary binary search tree
    - Color the node **red**
    - Restore the red-black tree properties

Red-Black Trees



- **RB-Insert(T,z)**

```
y ← NIL
x ← root[T]
```

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

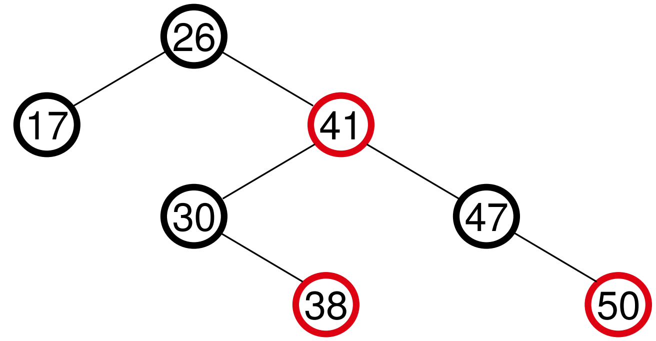
```
while x ≠ NIL
        do y ← x
            if key[z] < key[x]
              then x ← left[x]
            else x ← right[x]
```

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

```
p[z] ← y
```

- Sets the parent of z to be y

Red-Black Trees



- **RB-Insert(T,z)**

```
if y = NIL
    then root[T] ← z
    else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
left[z] ← NIL
right[z] ← NIL
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

The tree was empty: set the new node to be the root

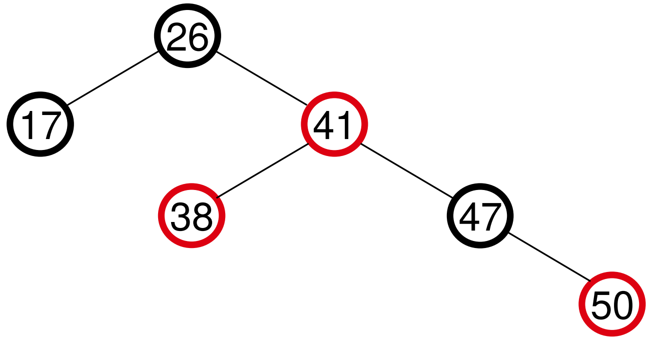
Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

Set the fields of the newly added node

Fix any inconsistencies that could have been introduced by adding this new red node

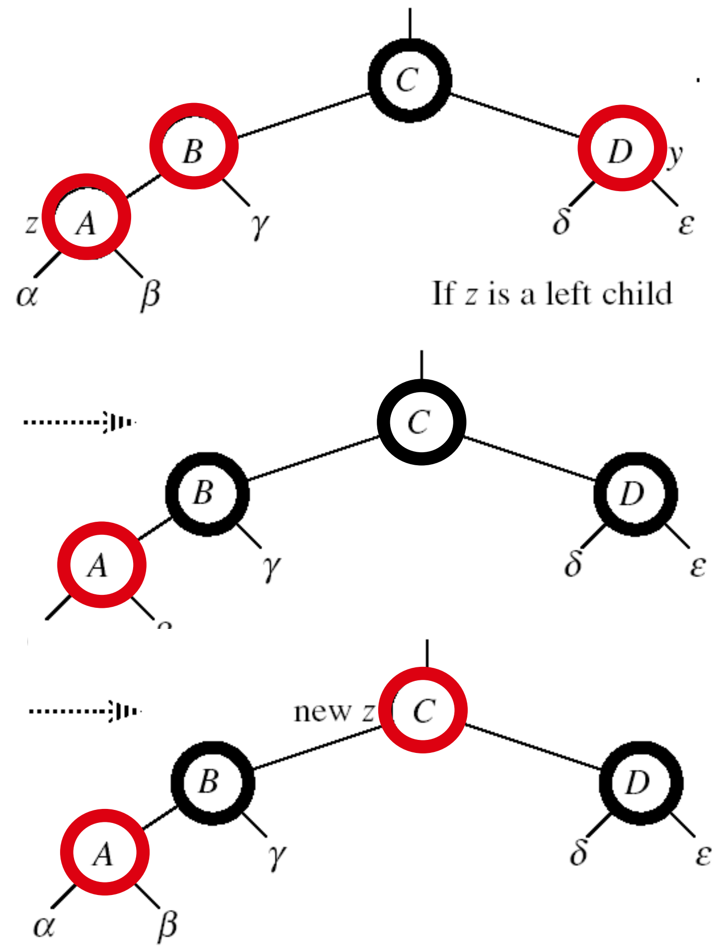
Red-Black Trees

- **Red-Black Tree Properties affected by Insert**
  1. Every node is either red or black (**OK**)
  2. The root is black (**Not OK!** – If z is the root)
  3. Every leaf (NIL) is black (**OK**)
  4. If a node is red then both its children are black (**Not OK!** – if p(z) is red, z and p(z) are both red)
  5. For each node, all paths from node to descendant leaves contain the same number of black nodes (**OK**)



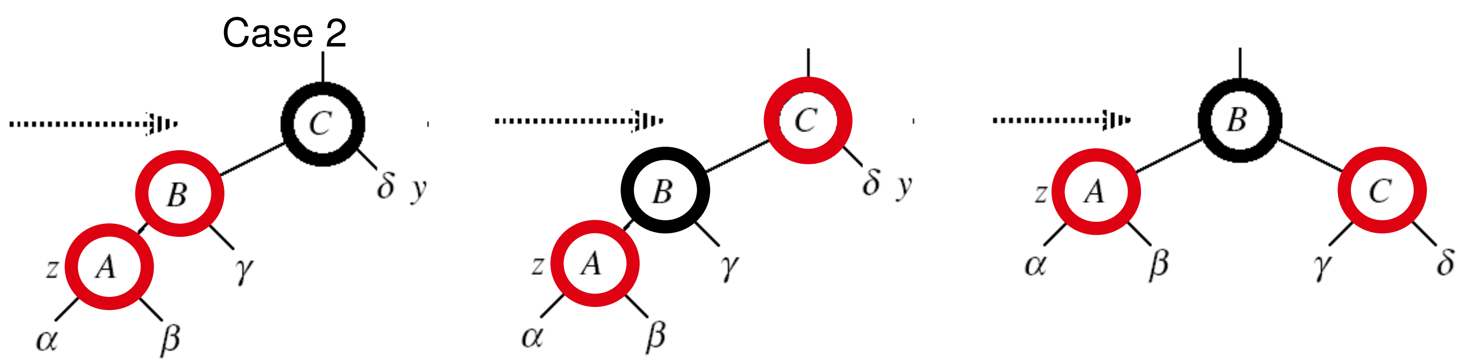
Red-Black Trees

- **RB-Insert-Fixup**
  - **Case 1**
    - z's "uncle" (y) is **red**
    - z either left or right child
  - **Idea**
    - p[p[z]] (z's grandparent) must be **black**
    - color p[z] ← **black**
    - color y ← **black**
    - color p[p[z]] ← **red**
    - z = p[p[z]]
      - Push the **"red"** violation up the tree



Red-Black Trees

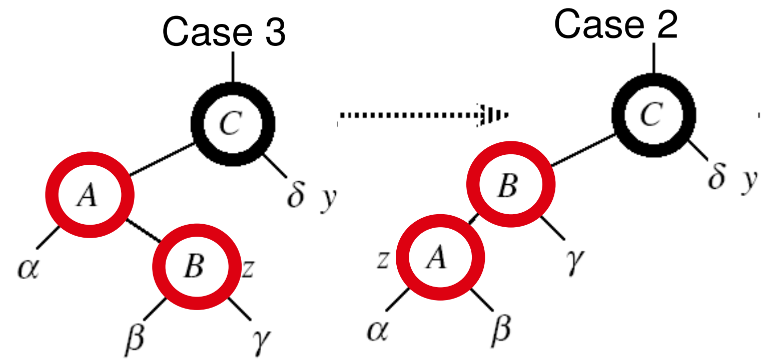
- **RB-Insert-Fixup**
  - **Case 2**
    - z's "uncle" (y) is **black**
    - Z is a left child
- **Idea**
  - color p[z] ← **black**
  - color p[p[z]] ← **red**
  - RIGHT-ROTATE(T, p[p[z]])
  - No longer have 2 **reds** in a row
  - p[z] is now **black**



Red-Black Trees

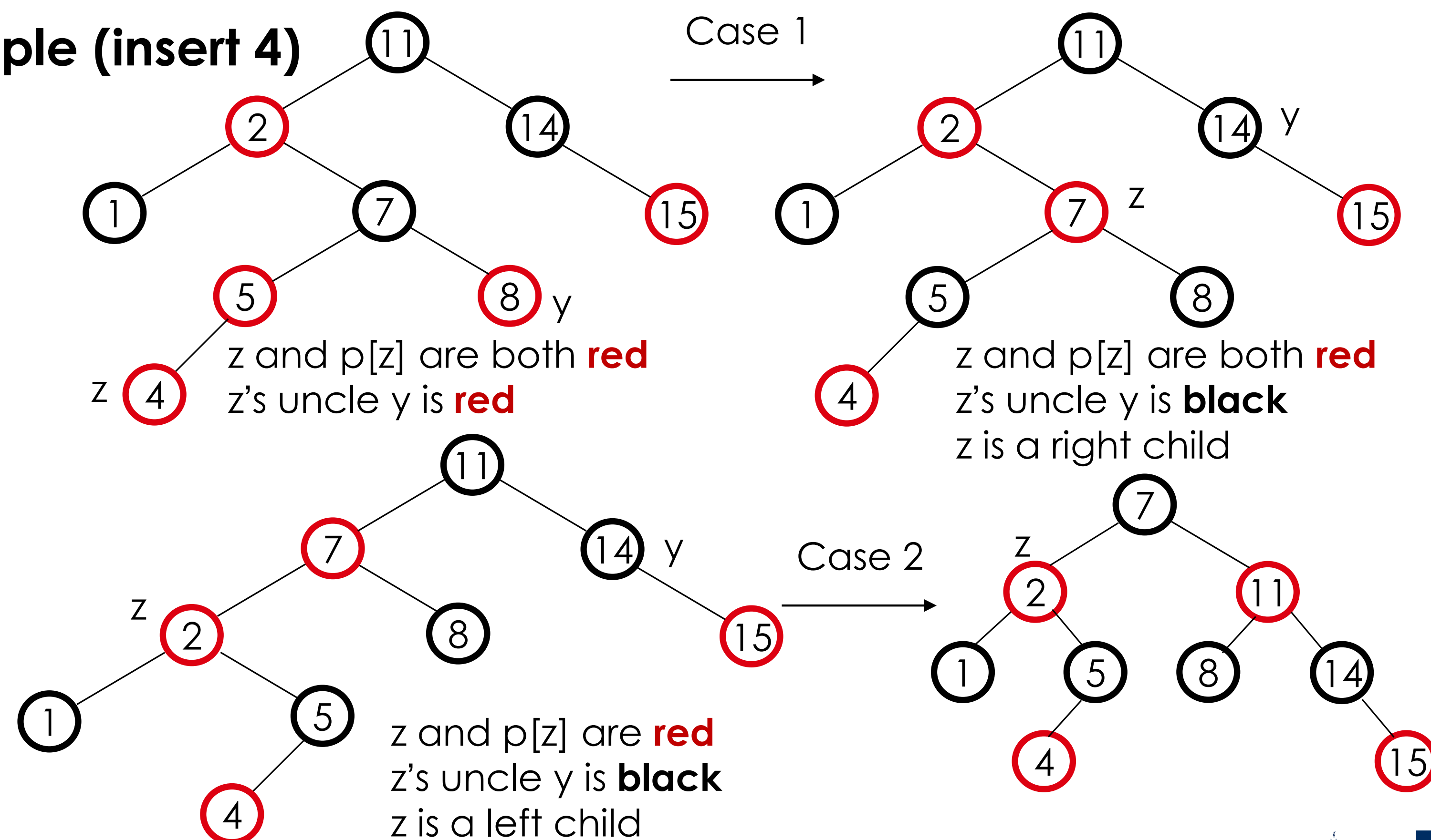
- **RB-Insert-Fixup**
  - **Case 3**
    - z's "uncle" (y) is **black**
    - z is a right child
  - **Idea**
    - z ←p[z]
    - LEFT-ROTATE(T, z)

    ⇒ now z is a left child, and both z and p[z] are **red** ⇒ **case 2**

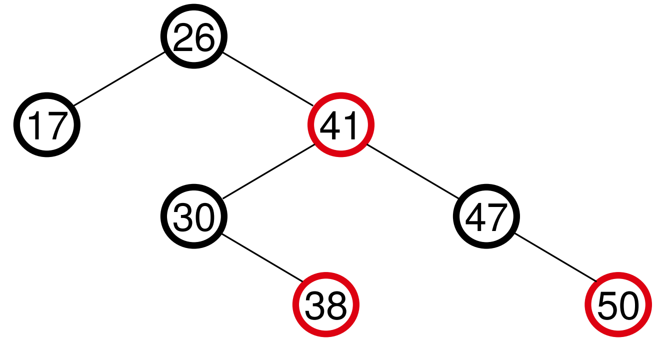


Red-Black Trees

- **Insertion Example (insert 4)**



Red-Black Trees



- **RB-Insert-Fixup(T,z)**

```
1.  while color[p[z]] = RED                    ← The while loop repeats only when
                                                 case1 is executed: O(logN) times
2.      if p[z] = left[p[p[z]]]                } Set the value of x's "uncle"
3.          then y ← right[p[p[z]]]            }
4.              if color[y] = RED
5.                  then Case1
6.              else if z = right[p[z]]
7.                      then Case3
8.                          Case2
9.      else (same as then clause with "right" and "left" exchanged for lines 3-4)
10. color[root[T]] ← BLACK                     ← We just inserted the root, or
                                                 The red violation reached the root
```

Red-Black Trees

Time Complexity

- Search: $O(logn)$
- Insert: $O(logn)$
- Remove: $O(logn)$

Red-Black Trees

Time Complexity

- Search: $O(logn)$
- Insert: $O(logn)$
- Remove: $O(logn)$

Storage Complexity

- $O(n)$