

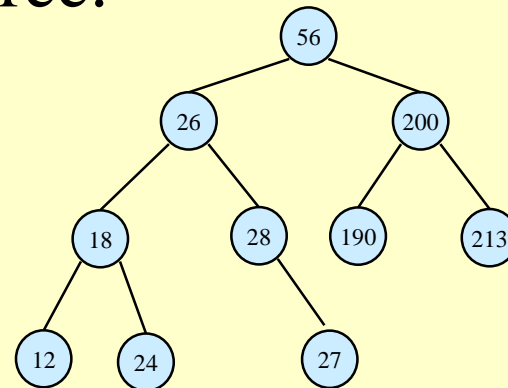
# Binary Search Trees

- What is a binary search tree?
- Tree searching
- Inorder traversal of a binary search tree
- Find Min & Max
- Predecessor and successor
- BST insertion and deletion

# Binary Trees

- ◆ Recursive definition
  1. An empty tree is a binary tree
  2. A node with two child subtrees is a binary tree
  3. Let  $A$  and  $B$  be two binary trees. A tree with root  $r$ , and  $A$  and  $B$  as its left and right subtrees, respectively, is a binary tree.

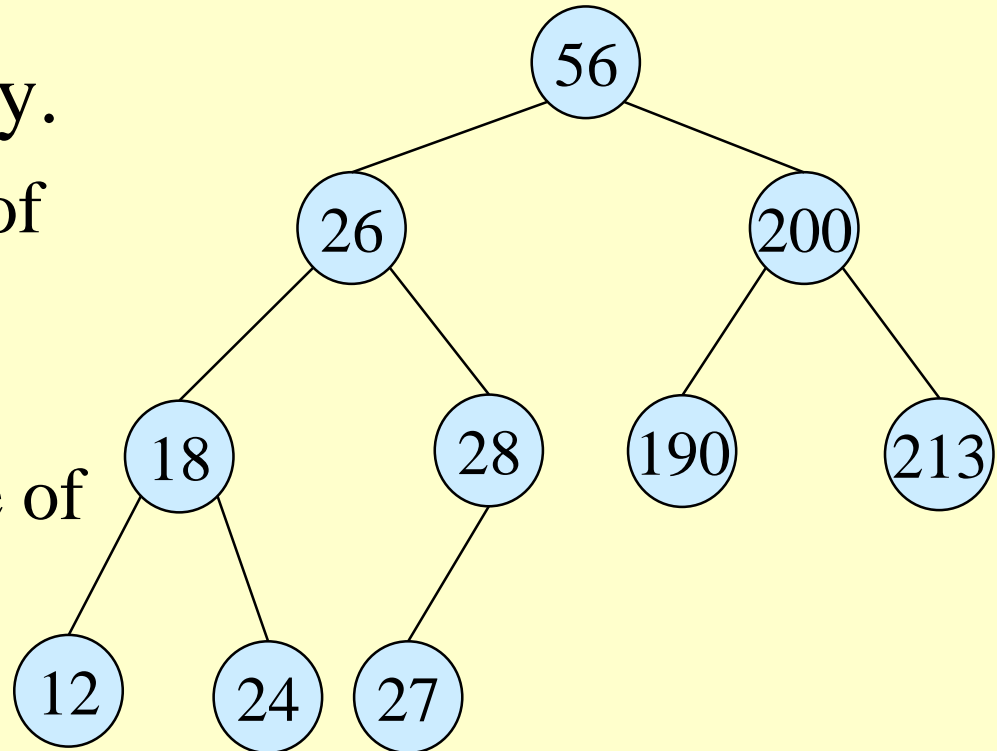
Is this a binary tree?



# Binary Search Tree

- ◆ Stored keys must satisfy the *binary search tree* property.

- »  $\forall y$  in left subtree of  $x$ , then  $key[y] < key[x]$ .
- »  $\forall y$  in right subtree of  $x$ , then  $key[y] \geq key[x]$ .

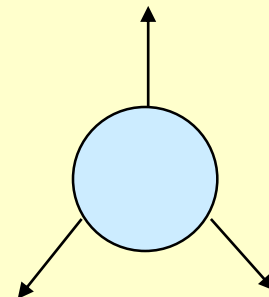


# Binary Search Trees

- ◆ A **BST** is a data structures that can support **dynamic set operations**.
  - » Search, Inorder traversal, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- ◆ Can be used to build
  - » **Dictionaries**.
  - » **Priority Queues**.
- ◆ Basic operations take time proportional to the height of the tree –  $O(h)$ .

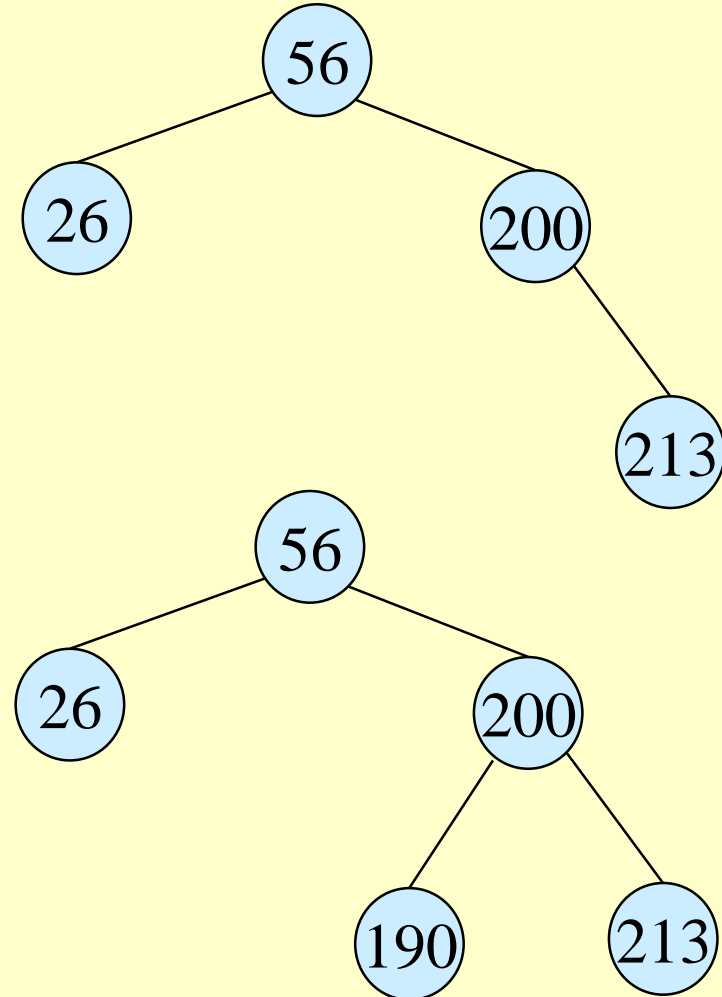
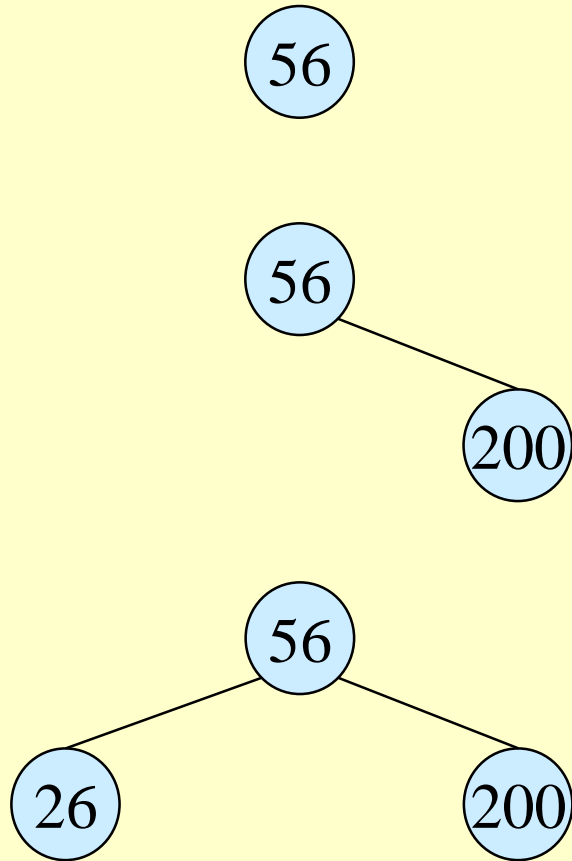
# BST – Representation

- ◆ Represented by a linked data structure of nodes.
- ◆ *root(T)* points to the root of tree *T*.
- ◆ Each node contains fields:
  - » *key*
  - » *left* – pointer to left child: root of left subtree.
  - » *right* – pointer to right child : root of right subtree.
  - » *p* – pointer to parent.  $p[\text{root}[T]] = \text{NIL}$  (optional).



# Binary Search Tree Construction

56, 200, 26, 213, 190, 28, 27, 18, 12, 24



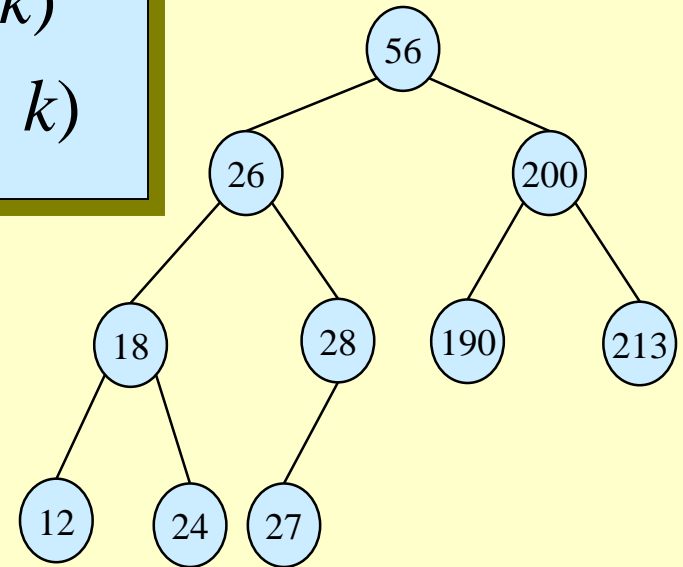
# Tree Search

## Tree-Search( $x, k$ )

1. **if**  $x = \text{NIL}$  *or*  $k = \text{key}[x]$
2.     **then** return  $x$
3. **if**  $k < \text{key}[x]$
4.     **then** return  $\text{Tree-Search}(\text{left}[x], k)$
5.     **else** return  $\text{Tree-Search}(\text{right}[x], k)$

**Running time:**  $O(h)$

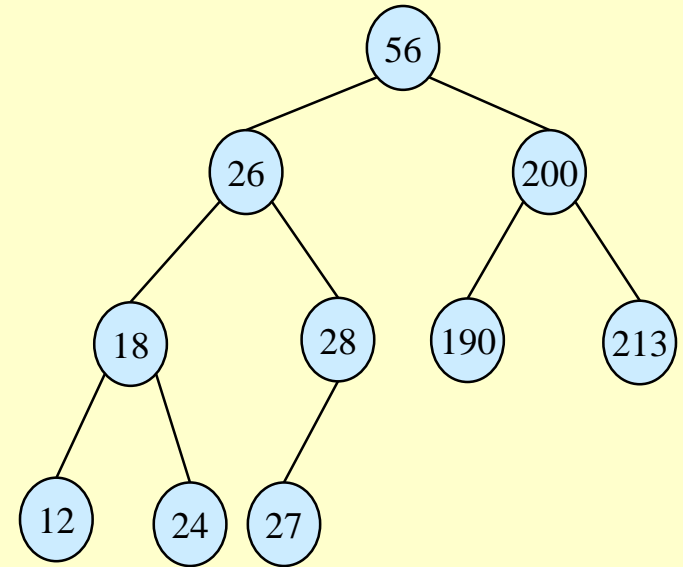
**Aside:** tail-recursion



# Iterative Tree Search

## Iterative-Tree-Search( $x, k$ )

1. **while**  $x \neq NIL$  **and**  $k \neq key[x]$
2.     **do if**  $k < key[x]$
3.         **then**  $x \leftarrow left[x]$
4.         **else**  $x \leftarrow right[x]$
5. **return**  $x$



The iterative tree search is more efficient on most computers.  
The recursive tree search is more straightforward.

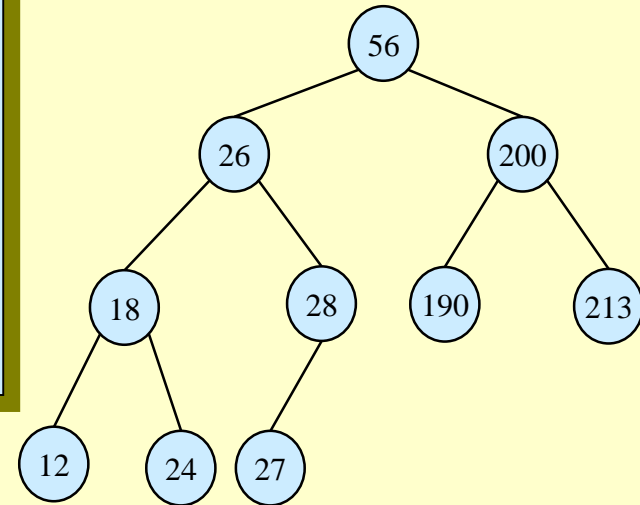


# Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

## Inorder-Tree-Walk ( $x$ )

1. **if**  $x \neq \text{NIL}$
2. **then** Inorder-Tree-Walk( $\text{left}[x]$ )
3.     print  $\text{key}[x]$
4.     Inorder-Tree-Walk( $\text{right}[x]$ )



- ◆ How long does the walk take?
- ◆ Can you prove its correctness?

# Correctness of Inorder-Walk

- ◆ Must prove that it prints all elements, in order, and that it terminates.
- ◆ By **induction on size of tree**. **Size=0**: Easy.
- ◆ **Size >1**:
  - » Prints left subtree in order by induction.
  - » Prints root, which comes after all elements in left subtree (still in order).
  - » Prints right subtree in order (all elements come after root, so still in order).

# Querying a Binary Search Tree

- ◆ All dynamic-set search operations can be supported in  $O(h)$  time.
- ◆  $h = \Theta(\lg n)$  for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- ◆  $h = \Theta(n)$  for an unbalanced tree that resembles a linear chain of  $n$  nodes in the worst case.

# Exercise: Sorting Using BSTs

Sort ( $A$ )

for  $i \leftarrow 1$  to  $n$

do tree-insert( $A[i]$ )

inorder-tree-walk( $root$ )

- » What are the worst case and best case running times?
- » In practice, how would this compare to other sorting algorithms?

# Finding Min & Max

- ◆ The binary-search-tree property guarantees that:
  - » The **minimum** is located at the **left-most** node.
  - » The **maximum** is located at the **right-most** node.

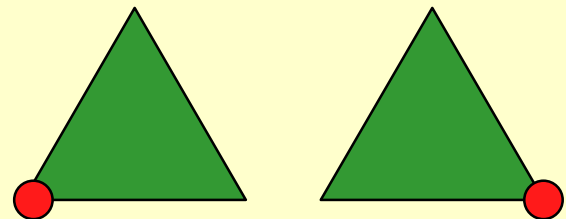
## Tree-Minimum( $x$ )

1. **while**  $left[x] \neq NIL$
2.     **do**  $x \leftarrow left[x]$
3. **return**  $x$

## Tree-Maximum( $x$ )

1. **while**  $right[x] \neq NIL$
2.     **do**  $x \leftarrow right[x]$
3. **return**  $x$

Q: How long do they take?

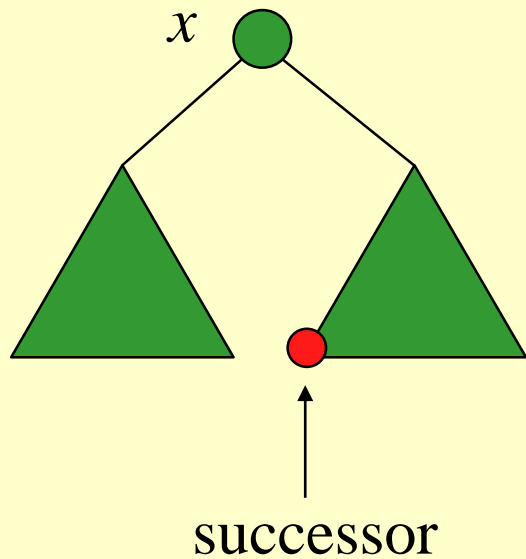


# Predecessor and Successor

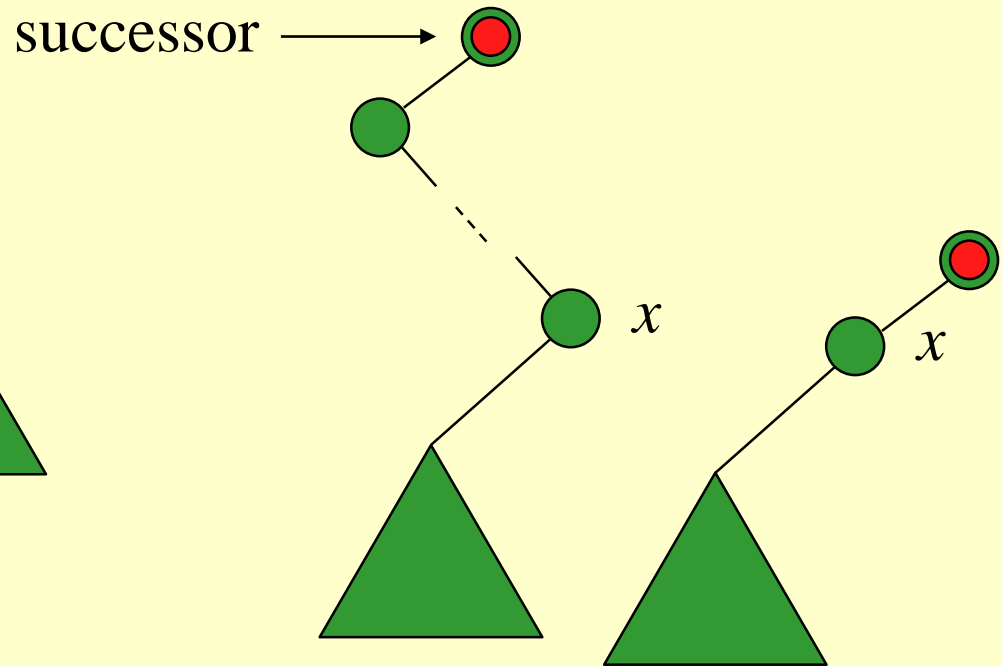
- ◆ Predecessor of node  $x$  is the node  $y$  such that  $key[y]$  is the greatest key smaller than  $key[x]$ .
- ◆ Successor of node  $x$  is the node  $y$  such that  $key[y]$  is the smallest key greater than  $key[x]$ .
- ◆ The successor of the largest key is NIL.
- ◆ Search consists of two cases.
  - » If node  $x$  has a non-empty right subtree, then  $x$ 's successor is the minimum in the right subtree of  $x$ .
  - » If node  $x$  has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - $x$ 's successor  $y$  is the node that is the predecessor of  $x$  ( $x$  is the maximum in  $y$ 's left subtree).
    - In other words,  $x$ 's successor  $y$ , is the lowest ancestor of  $x$  whose left child is also an ancestor of  $x$  or is  $x$  itself.

# Successor

Case 1:  $x$  has a non-empty right subtree.



Case 2:  $x$  has an empty right subtree.



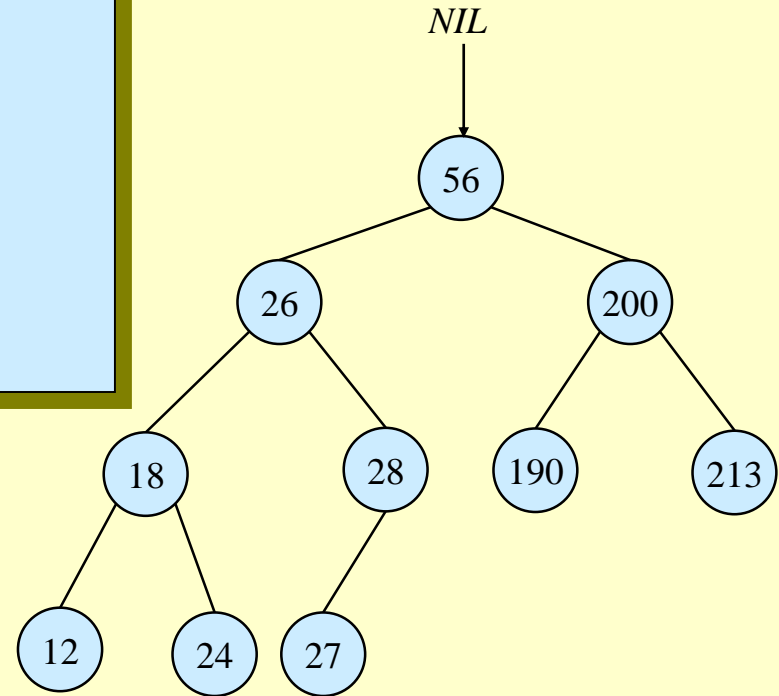
# Pseudo-code for Successor

## Tree-Successor( $x$ )

1. **if**  $right[x] \neq NIL$
2.     **then** return Tree-Minimum( $right[x]$ )
3.  $y \leftarrow p[x]$
4. **while**  $y \neq NIL$  **and**  $x = right[y]$
5. **do**  $x \leftarrow y$
6.      $y \leftarrow p[y]$
7. **return**  $y$

Code for *predecessor* is symmetric.

Running time:  $O(h)$

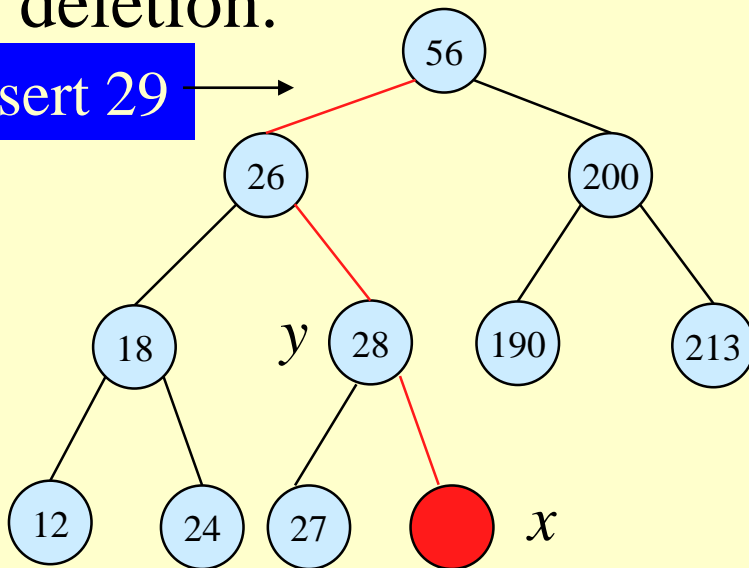




# BST Insertion – Pseudocode

- ◆ Change the dynamic set represented by a BST.
- ◆ Ensure the binary-search-tree property holds after change.
- ◆ Insertion is easier than deletion.

insert 29



## Tree-Insert( $T, z$ )

1.  $y \leftarrow \text{NIL}$
2.  $x \leftarrow \text{root}[T]$
3. **while**  $x \neq \text{NIL}$
4.     **do**  $y \leftarrow x$
5.         **if**  $\text{key}[z] < \text{key}[x]$
6.             **then**  $x \leftarrow \text{left}[x]$
7.             **else**  $x \leftarrow \text{right}[x]$
8.      $p[z] \leftarrow y$
9.     **if**  $y = \text{NIL}$
10.         **then**  $\text{root}[T] \leftarrow z$
11.         **else if**  $\text{key}[z] < \text{key}[y]$
12.             **then**  $\text{left}[y] \leftarrow z$
13.             **else**  $\text{right}[y] \leftarrow z$

# Analysis of Insertion

- ◆ **Initialization:**  $O(1)$
  - ◆ **While loop in lines 3-7** searches for place to insert  $z$ , maintaining parent  $y$ .  
This takes  $O(h)$  time.
  - ◆ **Lines 8-13** insert the value:  $O(1)$
- ⇒ **TOTAL:**  $O(h)$  time to insert a node.

## Tree-Insert( $T, z$ )

```
1.   $y \leftarrow \text{NIL}$ 
2.   $x \leftarrow \text{root}[T]$ 
3.  while  $x \neq \text{NIL}$ 
4.    do  $y \leftarrow x$ 
5.      if  $\text{key}[z] < \text{key}[x]$ 
6.        then  $x \leftarrow \text{left}[x]$ 
7.        else  $x \leftarrow \text{right}[x]$ 
8.   $p[z] \leftarrow y$ 
9.  if  $y = \text{NIL}$ 
10.    then  $\text{root}[t] \leftarrow z$ 
11.    else if  $\text{key}[z] < \text{key}[y]$ 
12.      then  $\text{left}[y] \leftarrow z$ 
13.      else  $\text{right}[y] \leftarrow z$ 
```

# Tree-Delete ( $T, z$ )

if  $z$  has no children

then remove  $z$

if  $z$  has one child

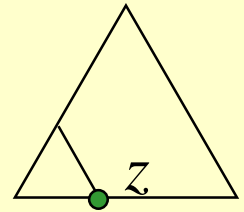
then make  $p[z]$  point to child

if  $z$  has two children (subtrees)

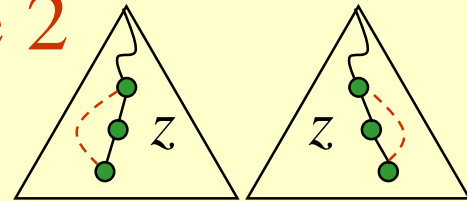
then swap  $z$  with its successor

perform case 1 or case 2 to delete it

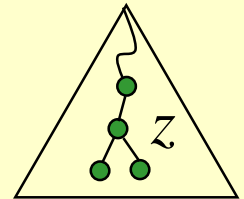
◆ case 1



◆ case 2



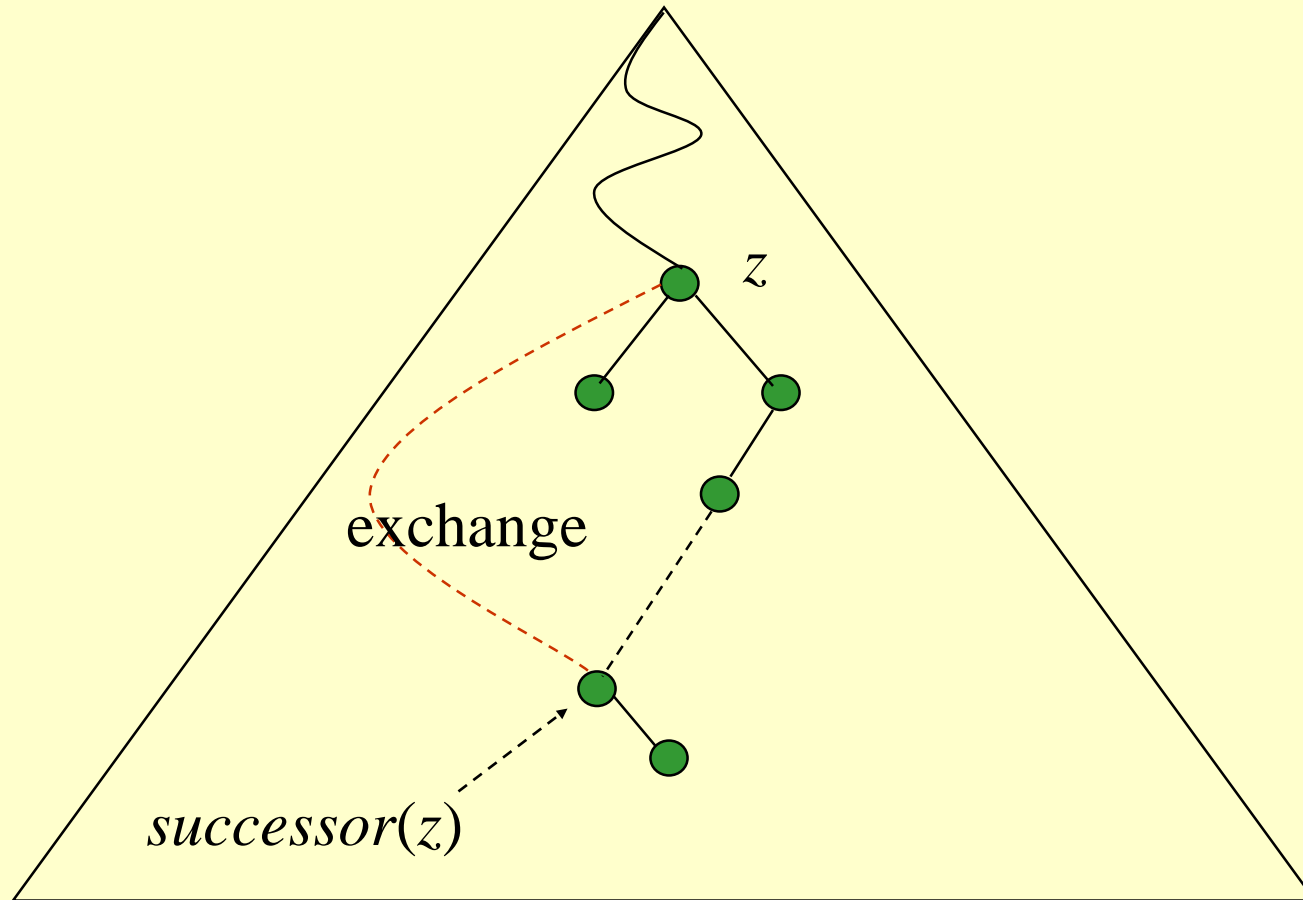
◆ case 3



⇒ TOTAL:  $O(h)$  time to delete a node

# Tree-Delete ( $T, z$ )

Illustration for case 3:



# Deletion – Pseudocode

## Tree-Delete( $T, z$ )

/\* Determine which node to splice out: either  $z$  or  $z$ 's successor. \*/

1. **if**  $left[z] = \text{NIL}$  **or**  $right[z] = \text{NIL}$
2.     **then**  $y \leftarrow z$                     /\*Case 1 or Case 2\*/
3.     **else**  $y \leftarrow \text{Tree-Successor}[z]$    /\*Case 3\*/

/\* Set  $x$  to a non-NIL child of  $y$ , or to NIL if  $y$  has no children. \*/

4. **if**  $left[y] \neq \text{NIL}$                     /\* $y$  has one child or no child.\*/
5.     **then**  $x \leftarrow left[y]$             /\* $x$  can be a child of  $y$  or NIL.\*/
6.     **else**  $x \leftarrow right[y]$

/\*  $y$  is removed from the tree by manipulating pointers of  $p[y]$  and  $x$  \*/

7. **if**  $x \neq \text{NIL}$
8.     **then**  $p[x] \leftarrow p[y]$

/\* Continued on next slide \*/

$y$  is the node be deleted, which has at most one child.

$x$  is the unique child of  $y$ .

# Deletion – Pseudocode

## Tree-Delete( $T, z$ ) (Contd. from previous slide)

```
9.   if  $p[y] = \text{NIL}$                                /*if y is the root*/
10.  then  $root[T] \leftarrow x$ 
11.  else if  $y = left[p[y]]$                           /*y is a left child.*/
12.      then  $left[p[y]] \leftarrow x$ 
13.      else  $right[p[y]] \leftarrow x$ 
/* If z's successor was spliced out, copy its data into z */
14.  if  $y \neq z$                                      /*y is z's successor.*/
15.      then  $key[z] \leftarrow key[y]$ 
16.          copy y's satellite data into z.
17.  return y
```

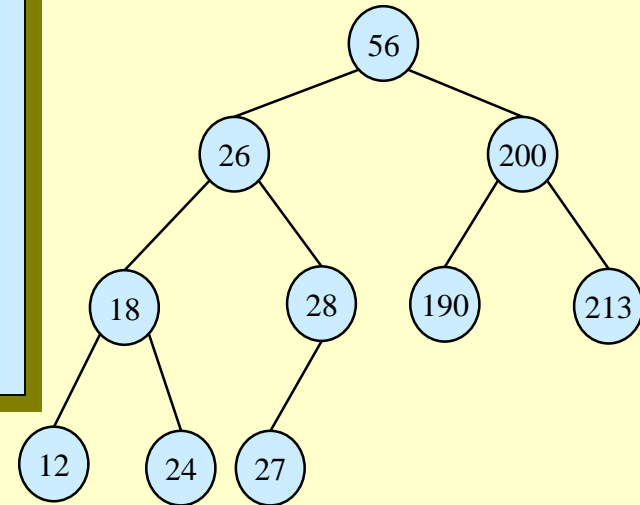
# Correctness of Tree-Delete

- ◆ How do we know case 2 should go to case 1 or case 2 instead of back to case 3?
  - » Because when  $x$  has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 1 or 2 child).
- ◆ Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.

## More on tree traversal

### preOrder-Tree-Walk ( $x$ )

1. **if**  $x \neq \text{NIL}$
2. **then** print  $key[x]$
3.     preOrder-Tree-Walk( $left[x]$ )
4.     preOrder-Tree-Walk( $right[x]$ )



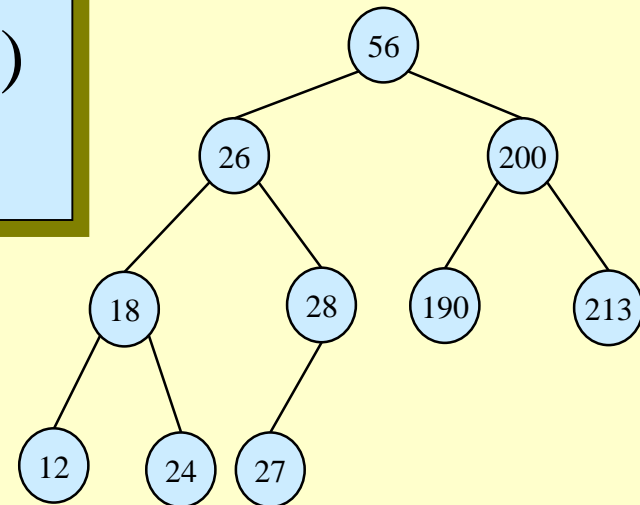
Also called top-down searching, depth-first searching



## More on tree traversal

### postOrder-Tree-Walk ( $x$ )

1. **if**  $x \neq \text{NIL}$
2. **then** postOrder-Tree-Walk( $\text{left}[x]$ )
3.     postOrder-Tree-Walk( $\text{right}[x]$ )
4.     print  $\text{key}[x]$

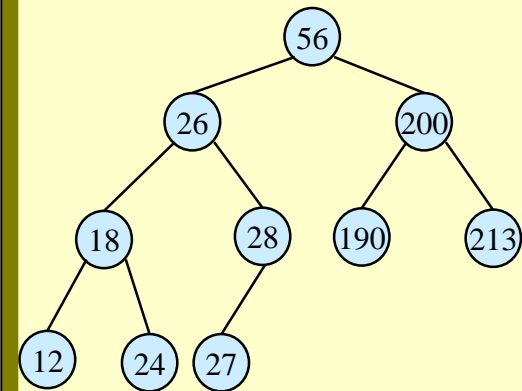


Also called bottom-up searching

# More on tree traversal

## Breadth-first ( $x$ )

1. **enqueue**( $Q, x$ )
2. **while**  $Q \neq \text{empty}$  **do**
3.      $v := \text{dequeue}(Q)$
4.     print  $\text{key}[x]$
5.     Let  $v_1, \dots, v_k$  be the children of  $x$
6.     **for** ( $i = 1$  to  $k$ ) **enqueue**( $Q, v_i$ )

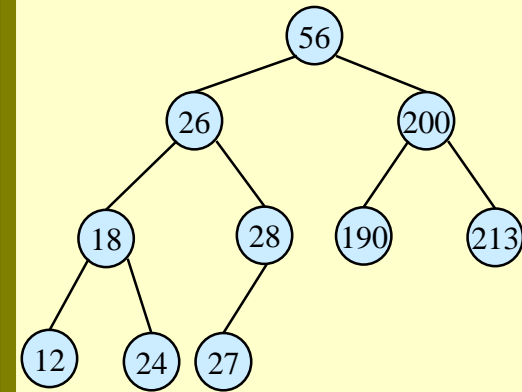


$Q$  is a queue.

# More on tree traversal

## Depth-first( $x$ )

1. **push**( $S, x$ )
2. **while**  $S \neq$  empty **do**
3.      $v :=$  **pop**( $S$ )
4.     print  $key[x]$
5.     Let  $v_1, \dots, v_k$  be the children of  $x$
6.     **for** ( $i = k$  to 1) **push**( $S, v_i$ )



$S$  is a stack.

It is also called the preorder search and top-down search.

# Binary Search Trees

- ◆ View today as data structures that can support **dynamic set operations**.
  - » Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- ◆ Can be used to build
  - » **Dictionaries**.
  - » **Priority Queues**.
- ◆ Basic operations take time proportional to the height of the tree –  $O(h)$ .

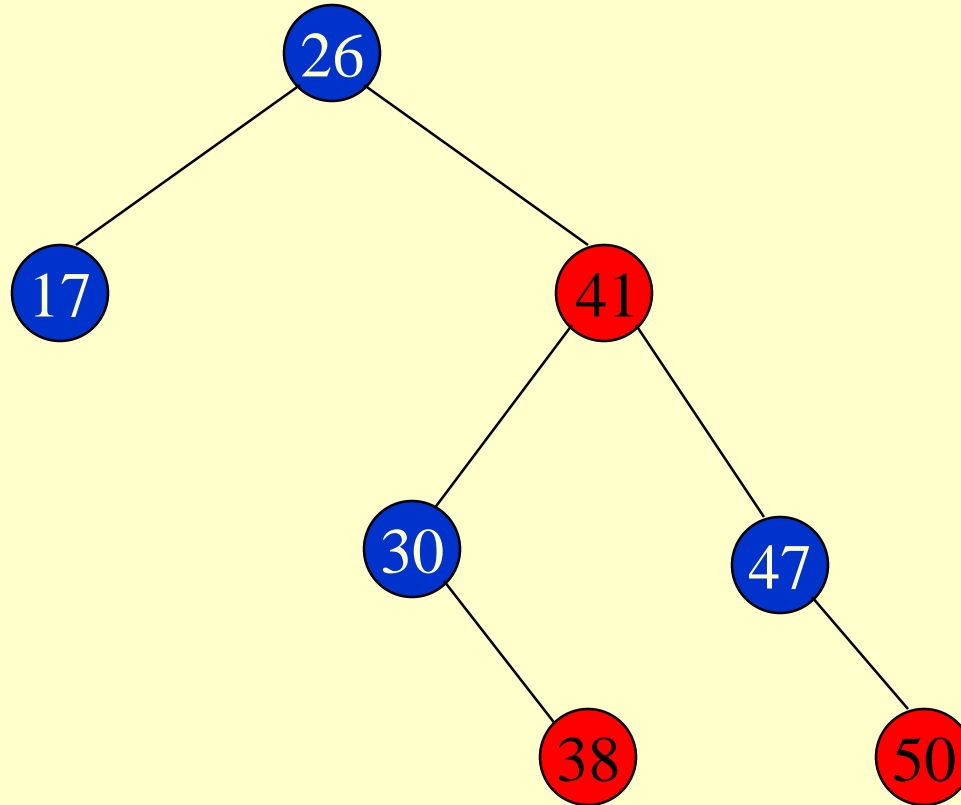
# Red-black trees: Overview

- ◆ Red-black trees are a variation of binary search trees to ensure that the tree is *balanced*.
  - » Height is  $O(\lg n)$ , where  $n$  is the number of nodes.
- ◆ Operations take  $O(\lg n)$  time in the *worst case*.

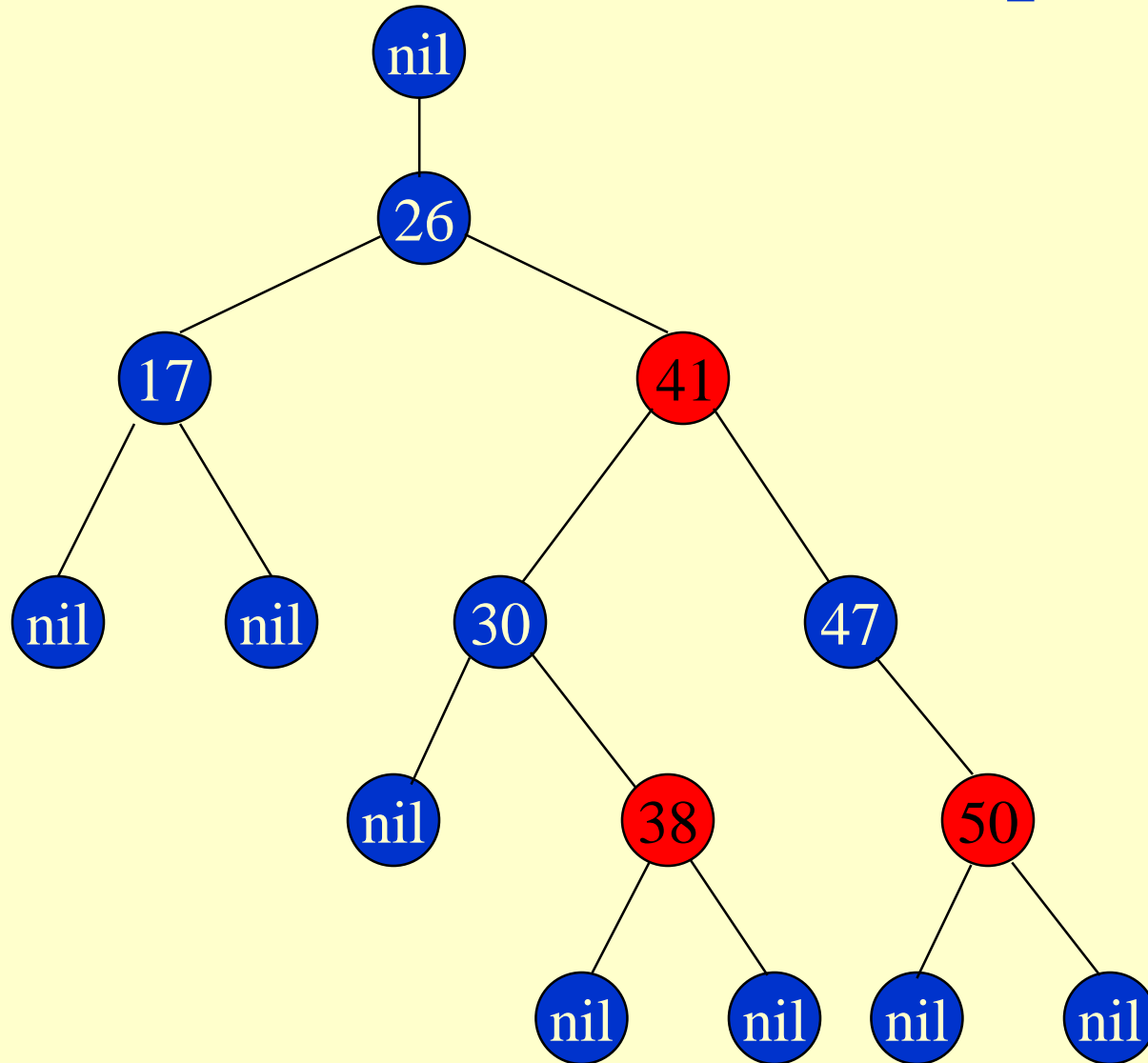
# Red-black Tree

- ◆ Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- ◆ All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- ◆ If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value *nil*.
- ◆ Sentinel - *nil*[*T* ], representing all the *nil* nodes.

# Red-black Tree – Example

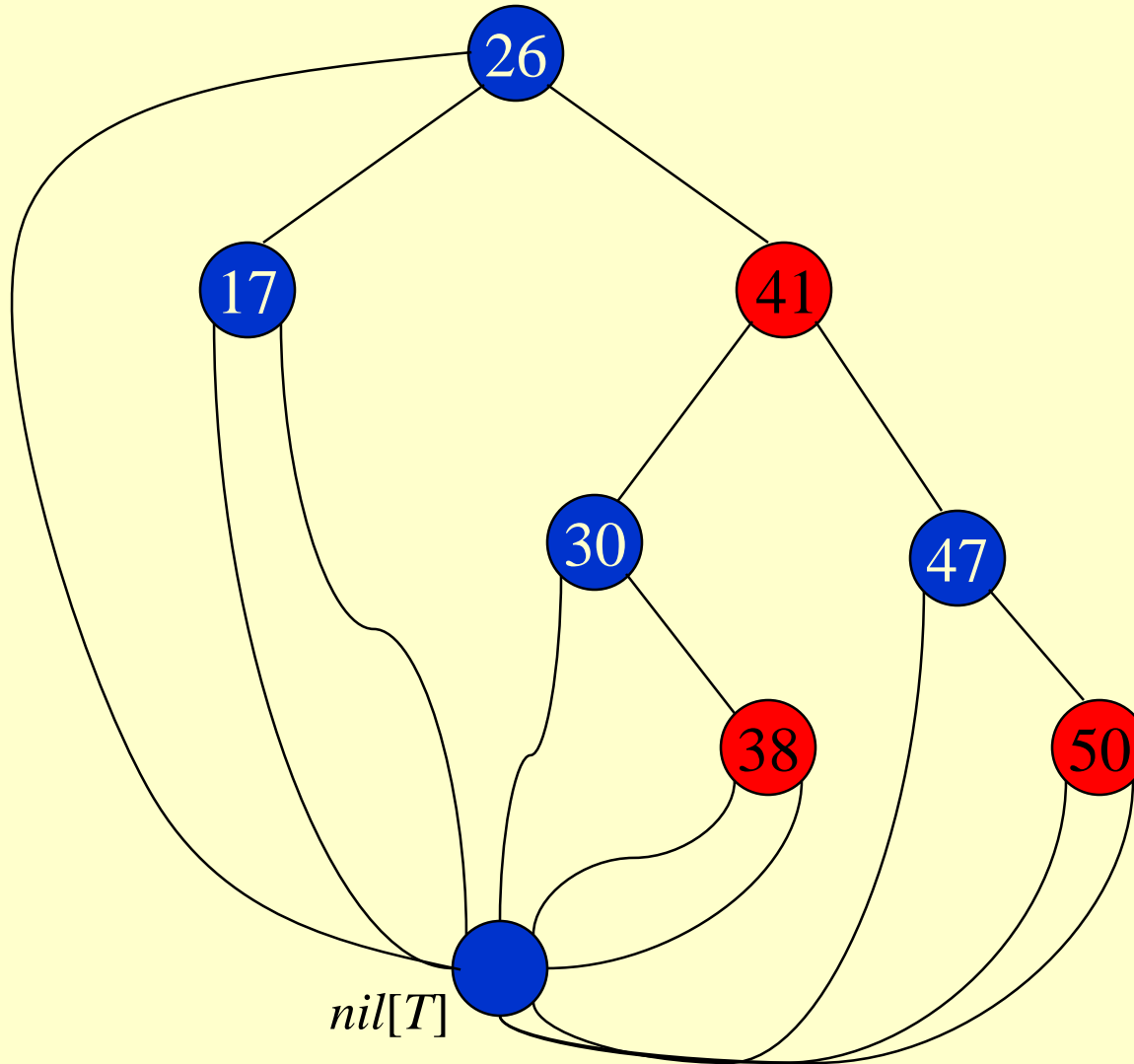


# Red-black Tree – Example





# Red-black Tree – Example



# Red-black Properties

1. Every node is either **red** or **black**.
2. The **root** is **black**.
3. Every **leaf** (*nil*) is **black**.
4. If a node is **red**, then both its children are **black**.
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes.

# Height of a Red-black Tree

- ◆ **Height of a node:**

- » Number of edges in a longest path to a leaf.

- ◆ **Black-height of a node  $x$ ,  $bh(x)$ :**

- »  $bh(x)$  is the number of black nodes (including  $nil[T]$ ) on the path from  $x$  to leaf, not counting  $x$ .

- ◆ **Black-height of a red-black tree is the black-height of its root.**

- » By Property 5, **black height is well defined.**

# Height of a Red-black Tree

◆ Example:

◆ **Height of a node:**

» Number of edges in a longest path to a leaf.

◆ **Black-height of a node**

$bh(x)$  is the number of black nodes on path from  $x$  to leaf, not counting  $x$ .

