King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 8

Balanced Trees, AVL Trees and Heaps

Information and Computer Science ICS

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 6 Sections 7, 9 and 10.

Objectives

Discuss the following topics:

- Balancing a Tree
- Rotations
- DSW Algorithms
- AVL Trees
- Heaps
- Polish Notation and Expression Trees

Balancing a Tree

- A binary tree is **height-balanced** or **balanced** if the difference in height of both subtrees of any node in the tree is either zero or one
- A tree is considered **perfectly balanced** if it is balanced and all leaves are to be found on one level or two levels

Balancing a Tree

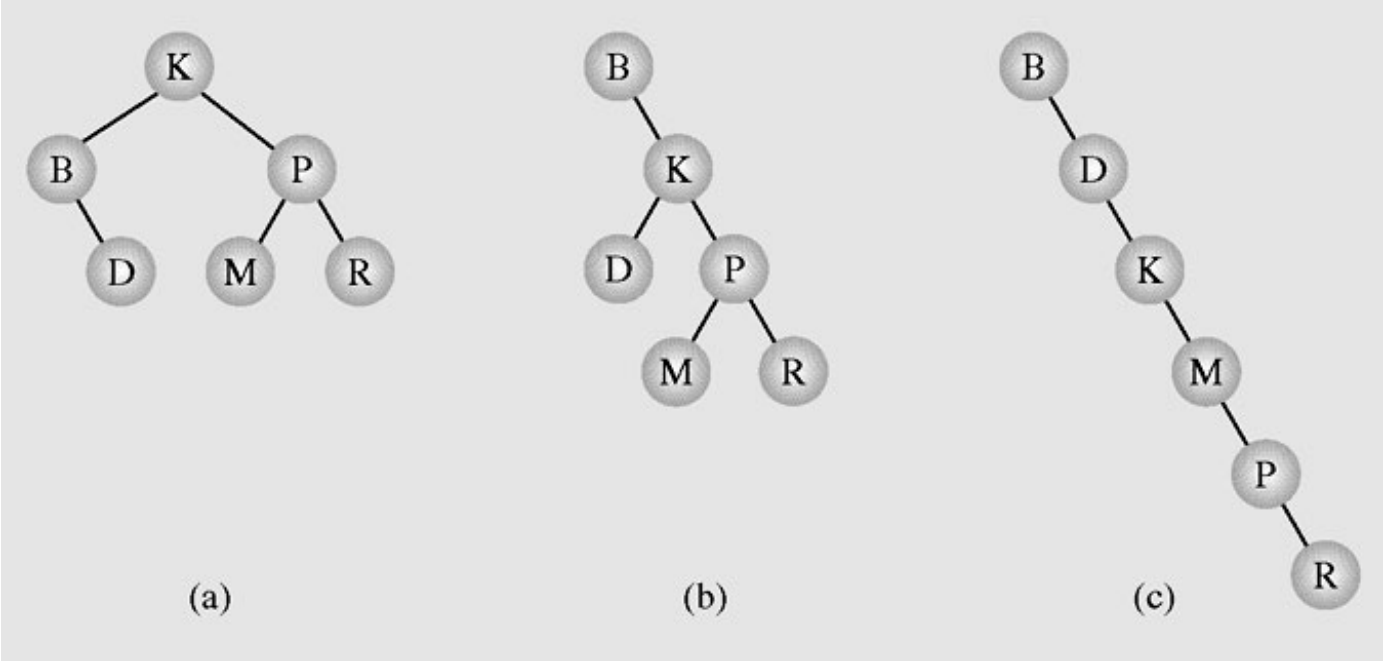
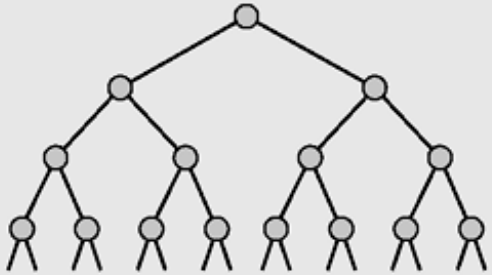


Figure 6-34 Different binary search trees with the same information

Balancing a Tree



Height	*Nodes at One Level*	*Nodes at All Levels*
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

Figure 6-35 Maximum number of nodes in binary trees of different heights

Balancing a Tree Using an Array

```
void balance (T data[], int first, int last) {
  if (first <= last) {
    int middle = (first + last)/2;
    insert(data[middle]); // into the BST
    balance(data,first,middle-1);
    balance(data,middle+1,last);
  }
}
```

Balancing a Tree (continued)

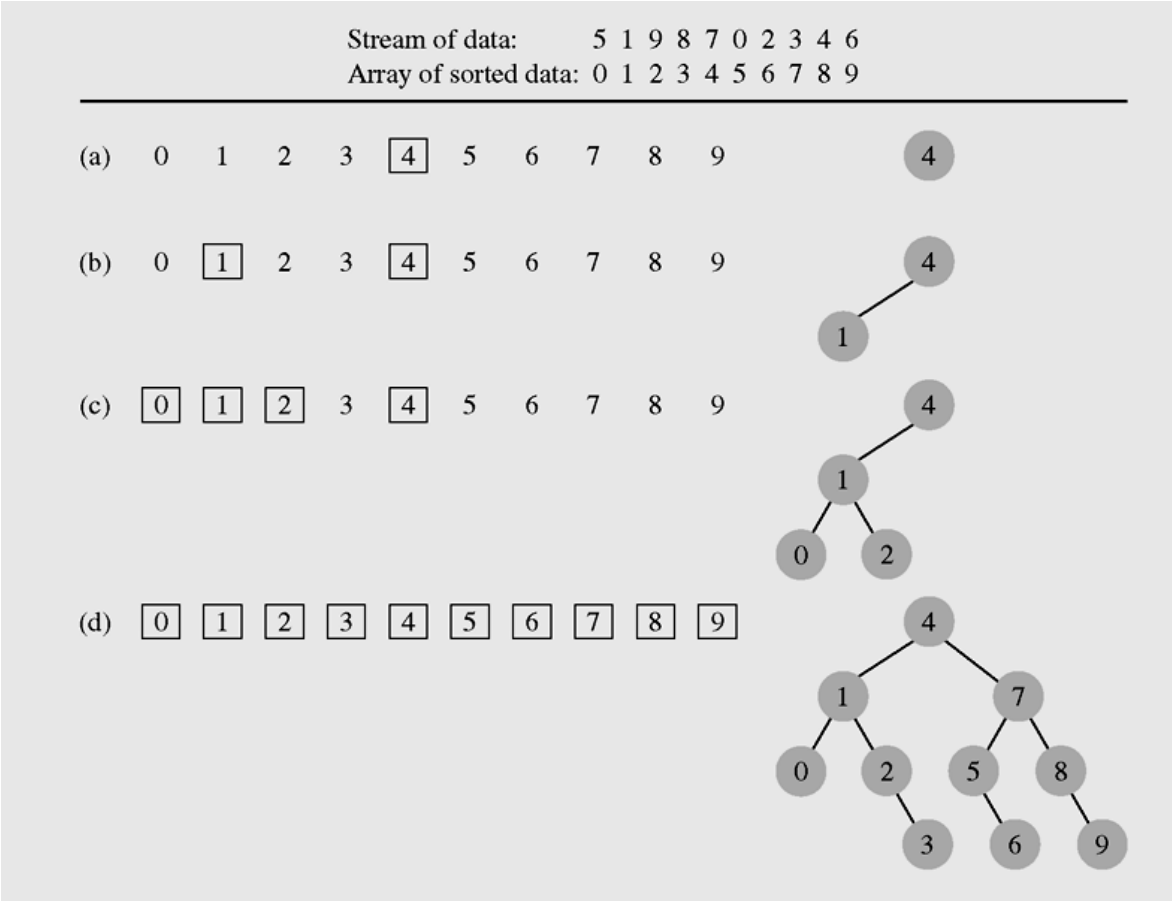
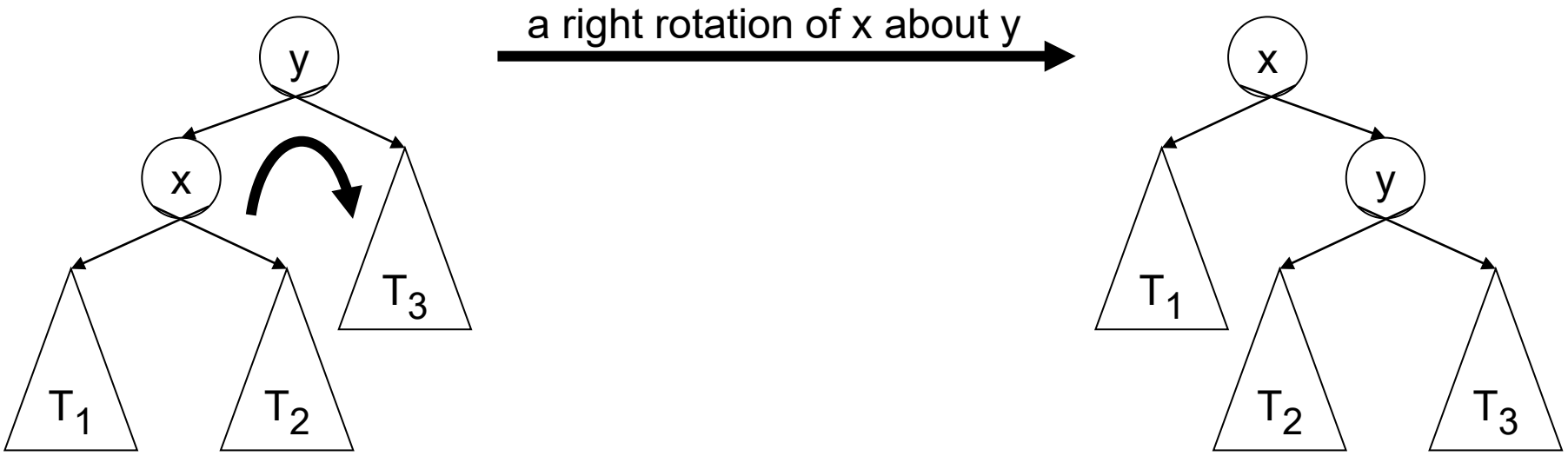


Figure 6-36 Creating a binary search tree from an ordered array

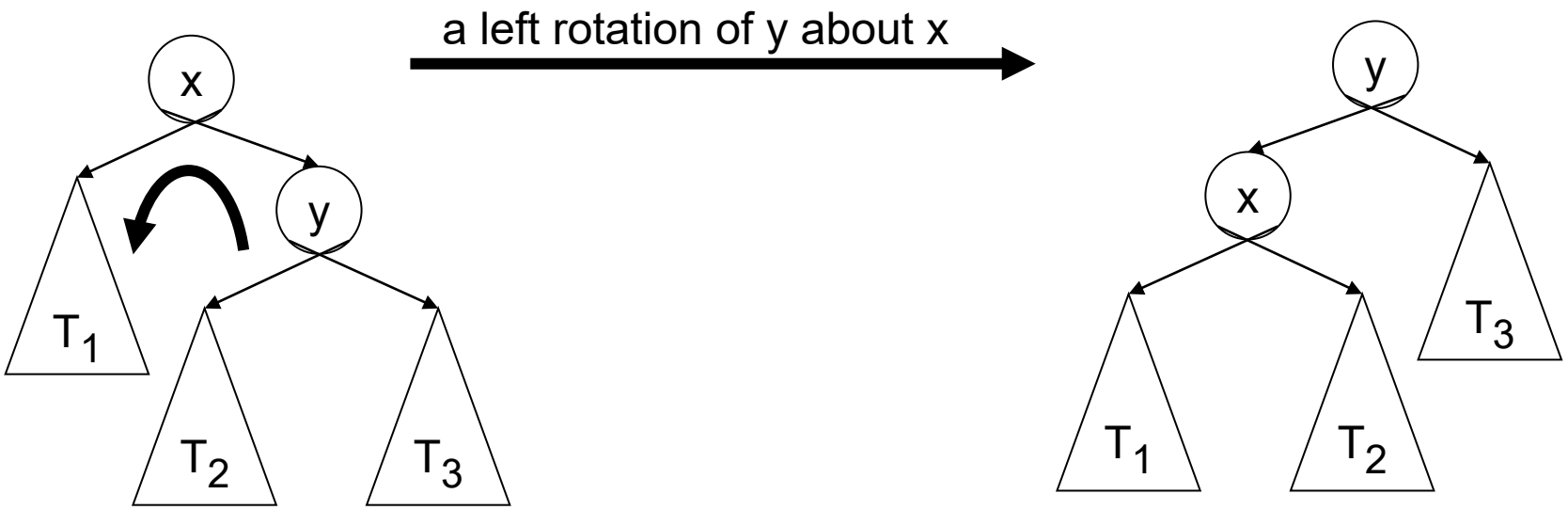
Rotations

- A rotation is a process of switching children and parents among two or three adjacent nodes.
- Single right rotation:
  - The left child x of a node y becomes y's parent.
  - y becomes the right child of x.
  - The right child T_2 of x, if any, becomes the left child of y.



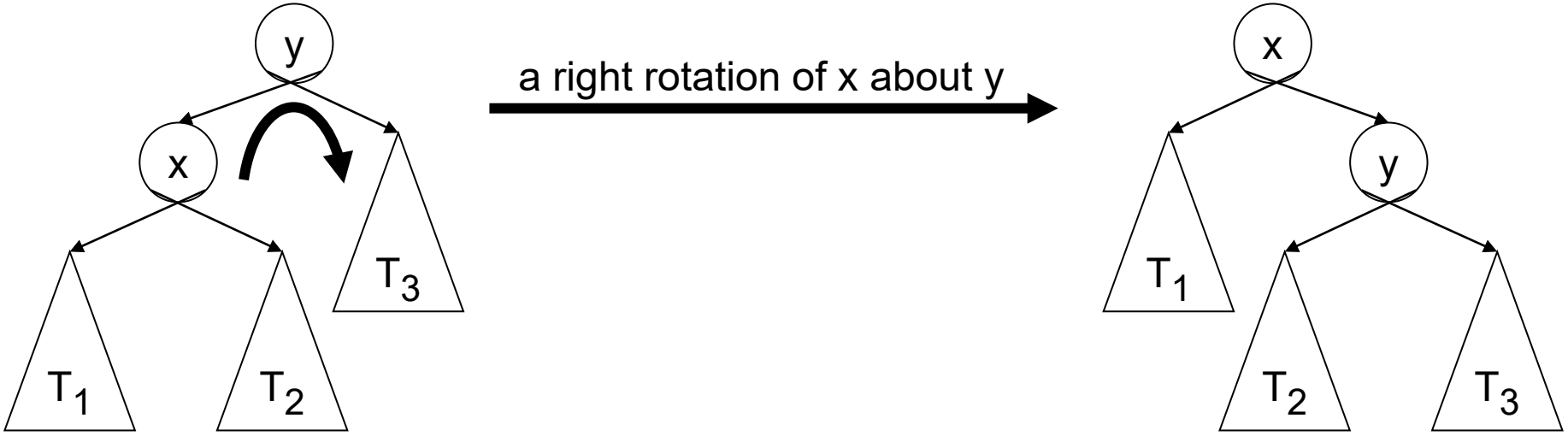
Rotations

- Single left rotation:
  - The right child y of a node x becomes x's parent.
  - x becomes the left child of y.
  - The left child T_2 of y, if any, becomes the right child of x.



BST ordering property after a rotation

- A rotation does not affect the ordering property of a BST (Binary Search Tree).



BST ordering property requirement:

$T_1 < x < y$

$x < T_2 < y$

$x < y < T_3$

Similar

BST ordering property requirement:

$T_1 < x < y$

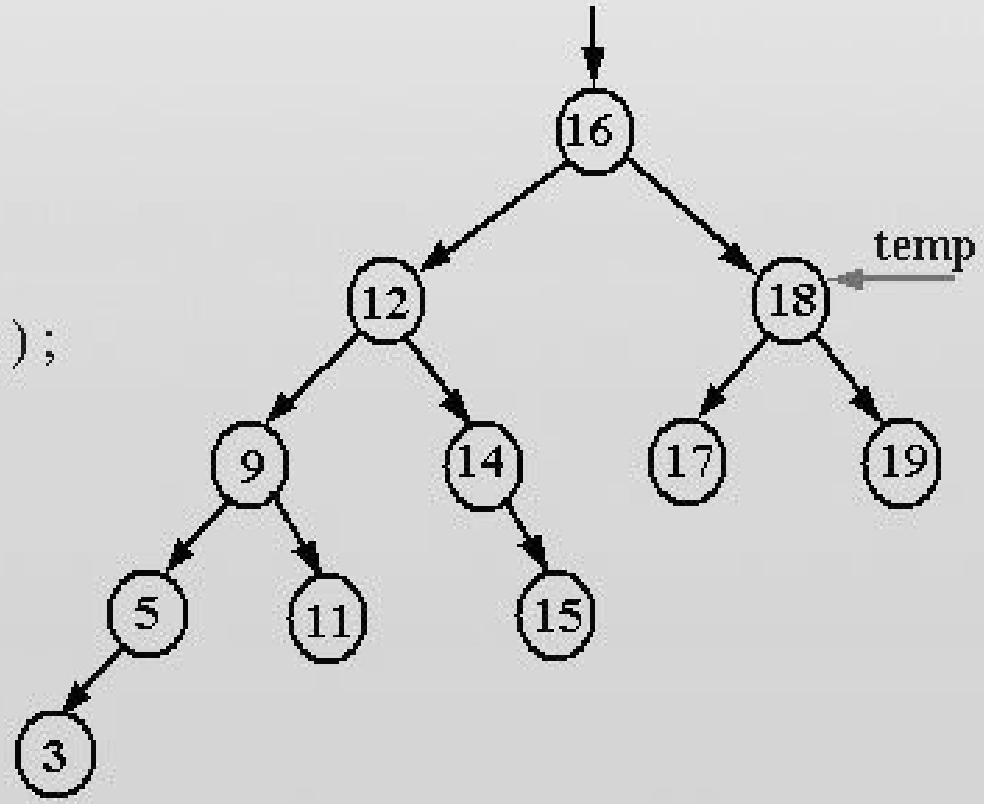
$x < T_2 < y$

$x < y < T_3$

- Similarly for a left rotation.

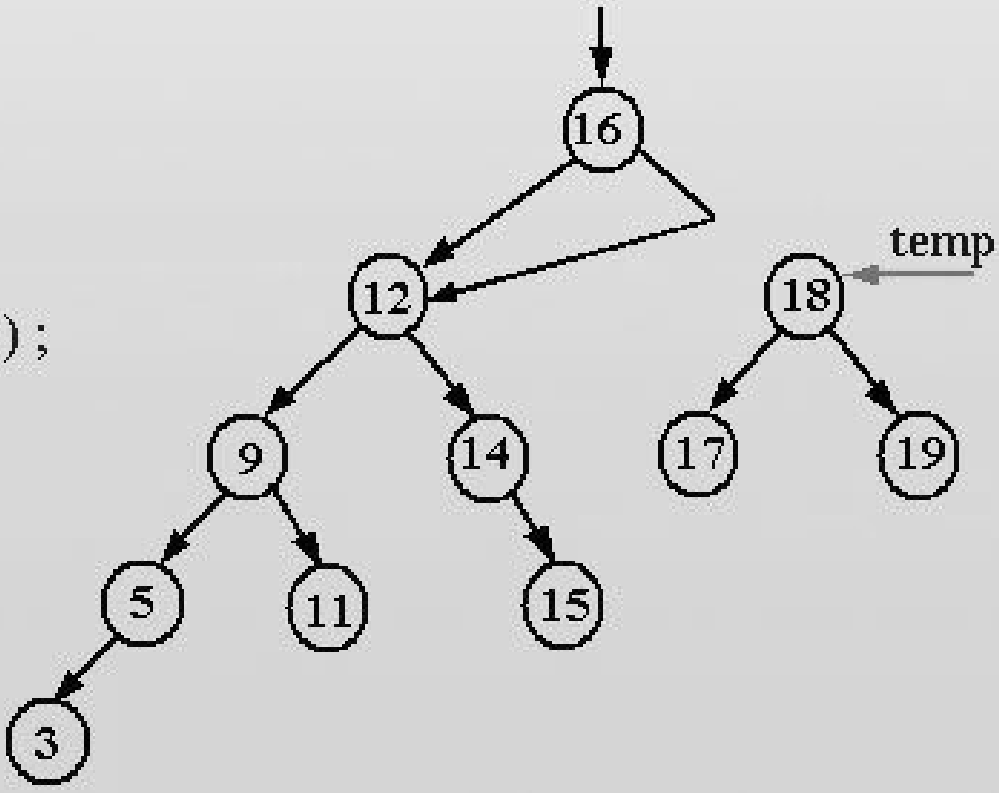
Single Right Rotation Implementation (example)

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



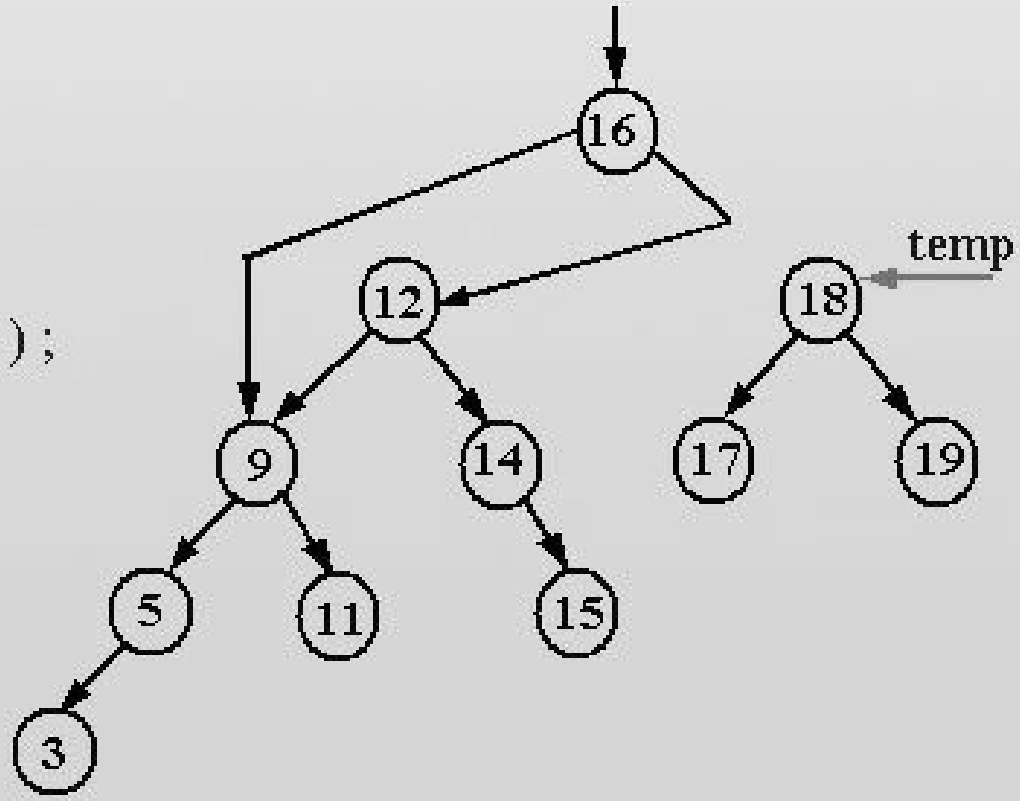
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



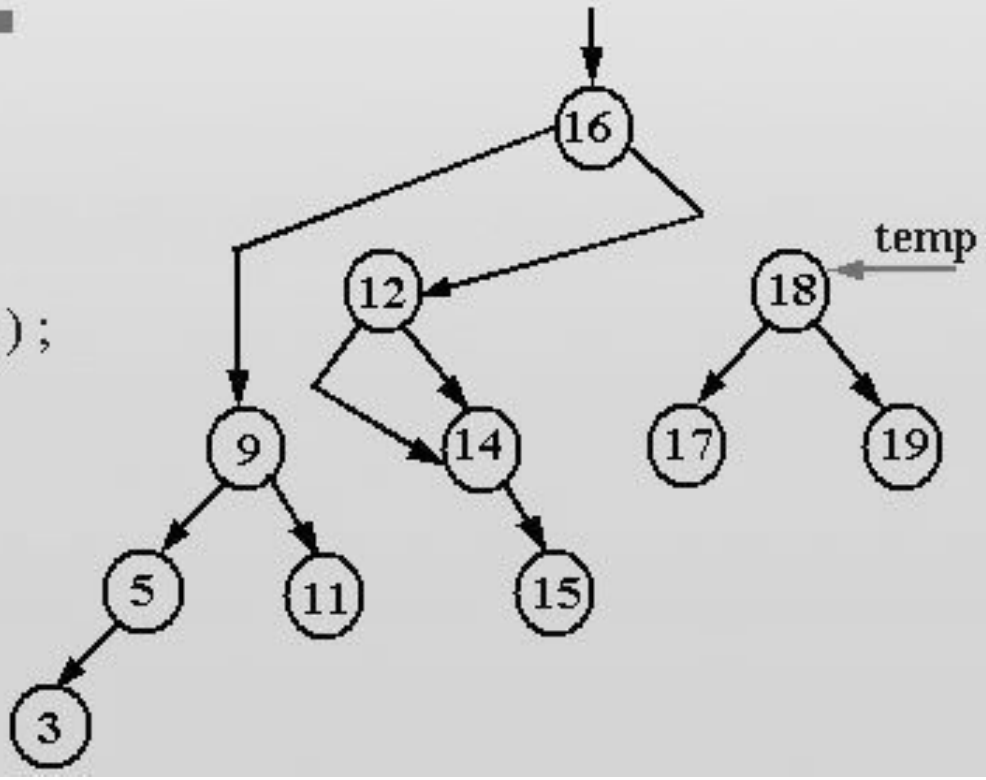
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



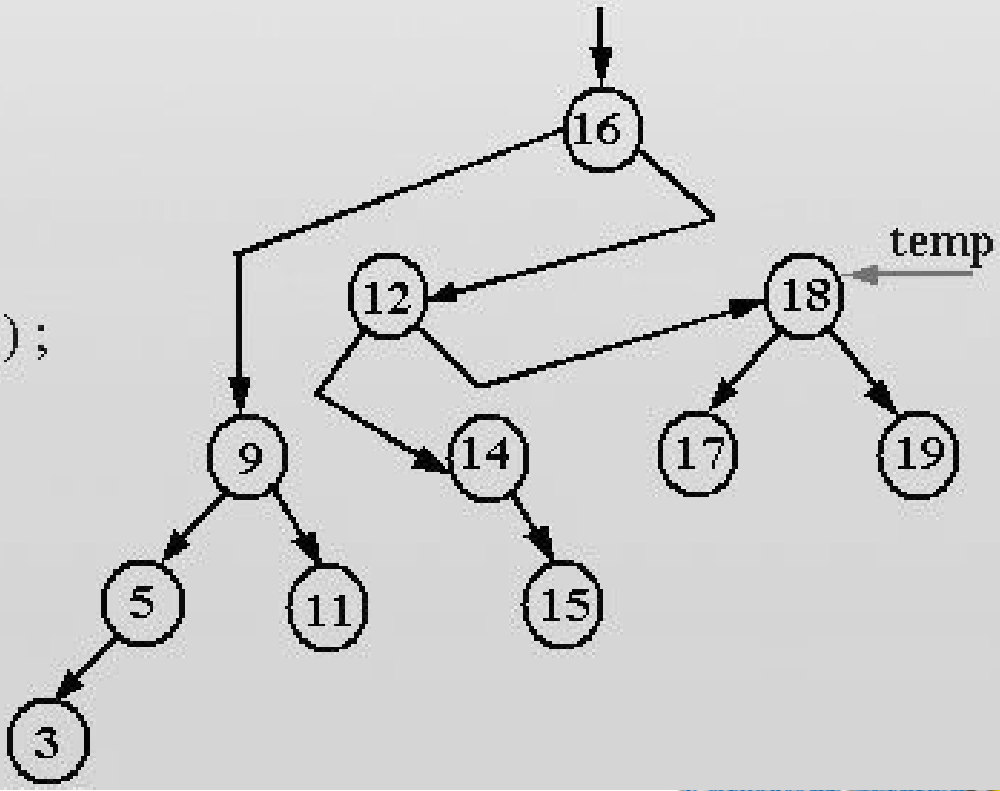
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



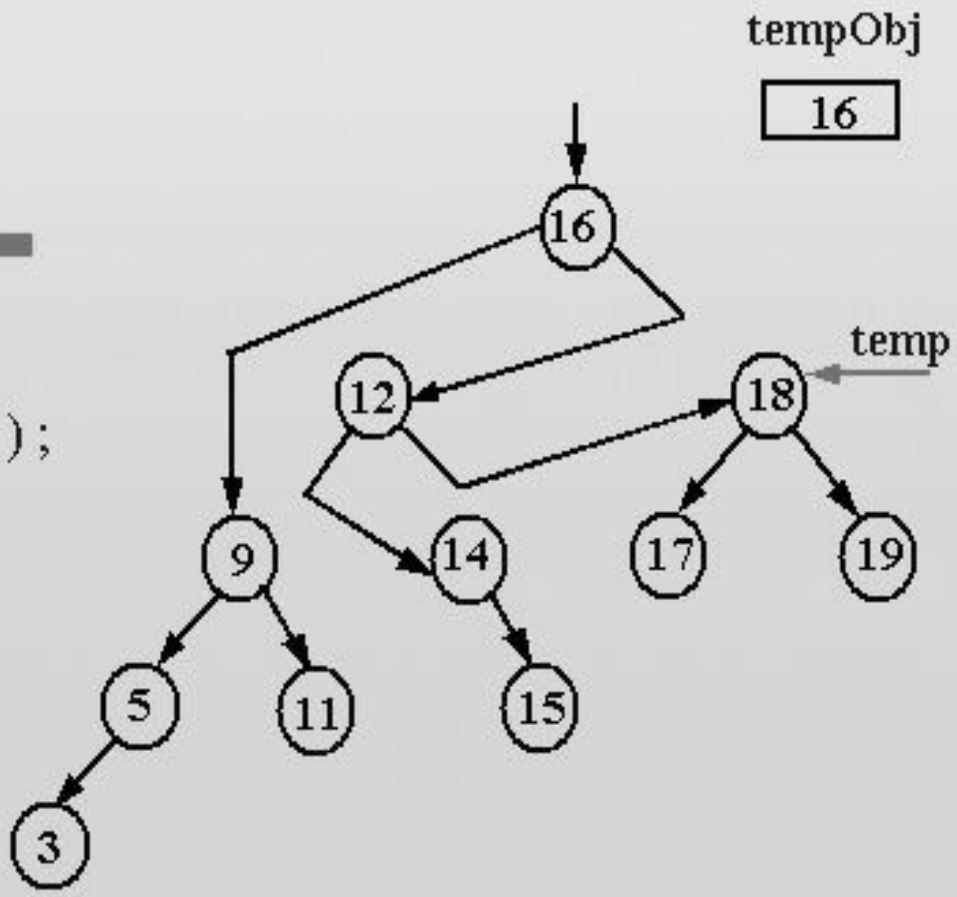
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



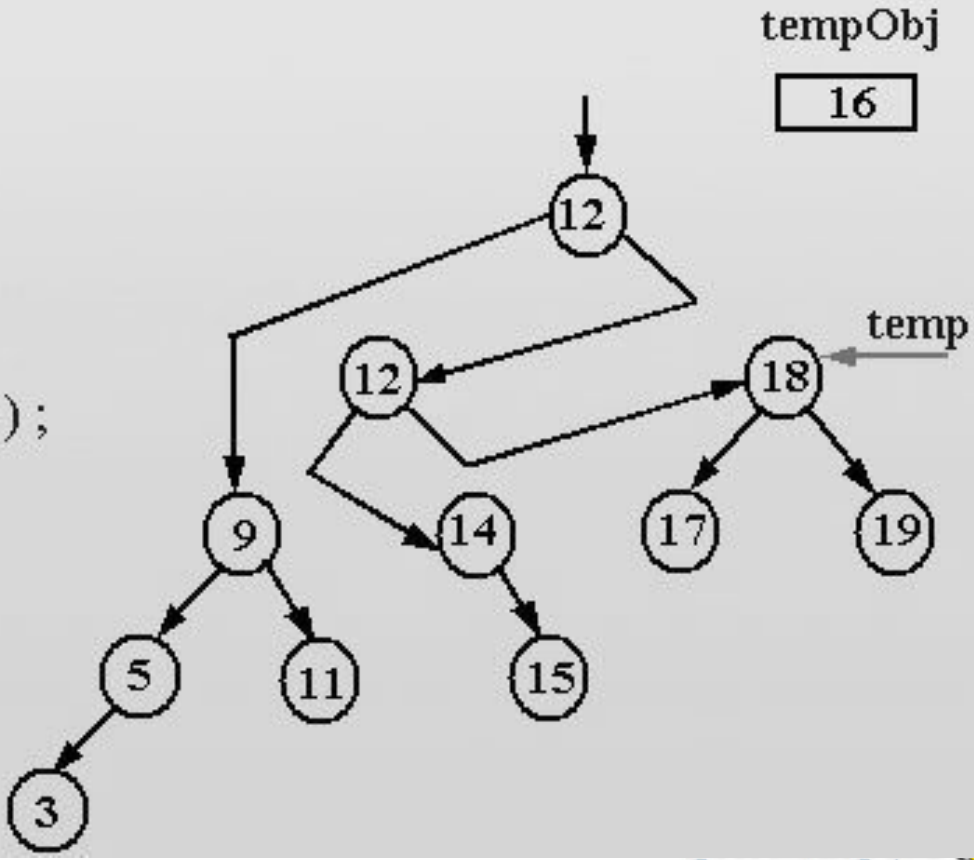
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



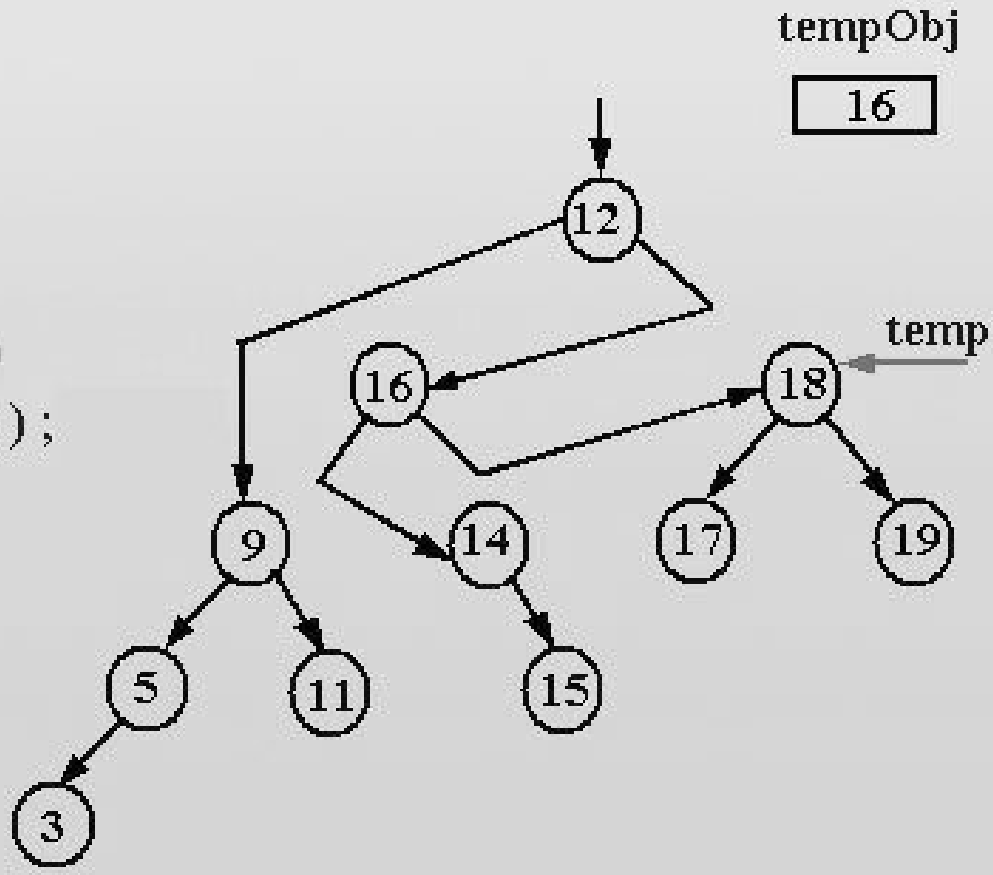
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



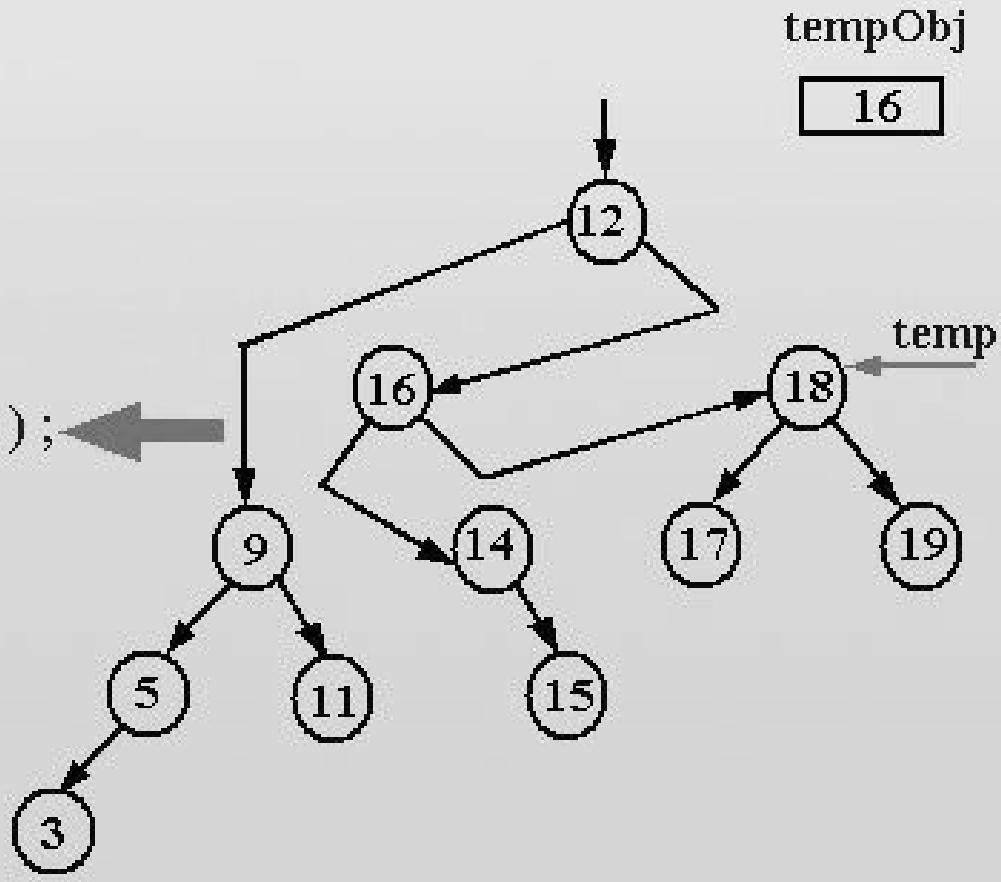
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



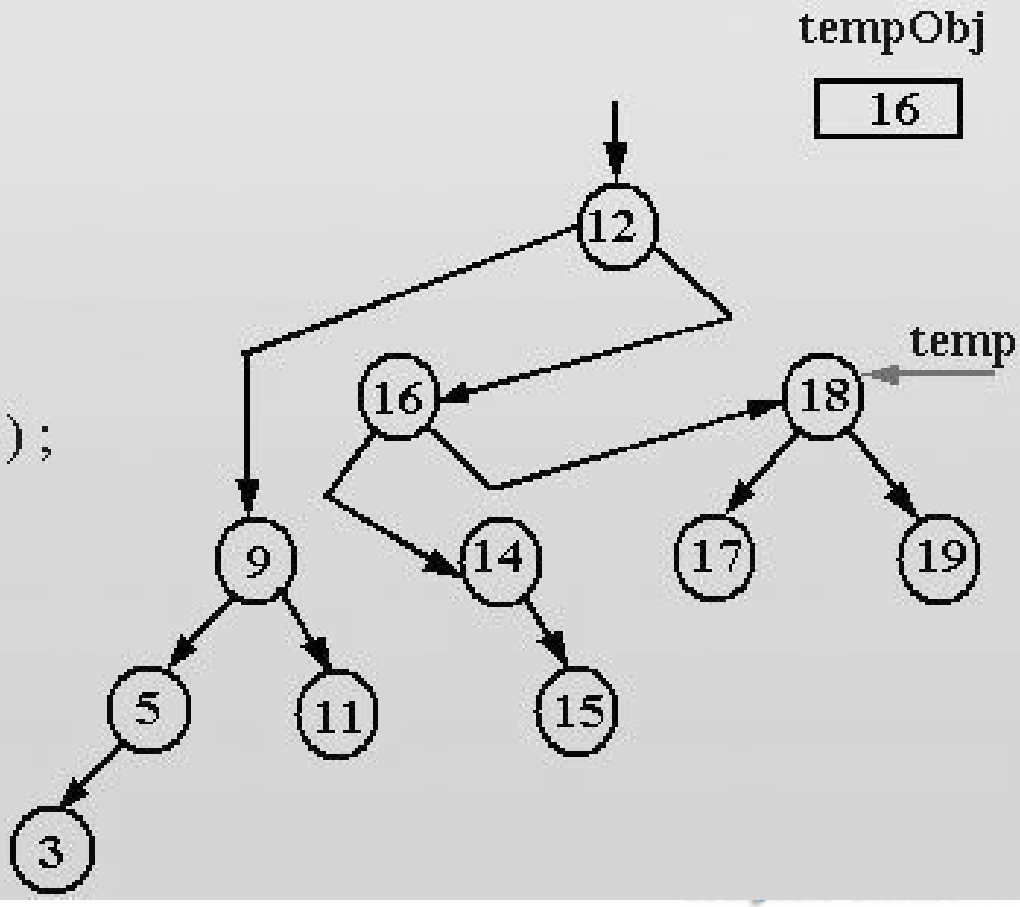
Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tempObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



Single Right Rotation Implementation (example) contd

```
1  protected void   rightRotate( ) {
2      if( isEmpty( )) throw new InvalidOperationException( ) ;
3      BinaryTree temp = right ;
4      right = left ;
5      left = right.left ;
6      right.left = right.right ;
7      right.right = temp ;
8      Object tmpObj = key ;
9      key = right.key ;
10     right.key = tmpObj ;
11     getRightAVL( ).adjustHeight( ) ;
12     adjustHeight( ) ;
13 }
```



DSW Algorithm

```
Algorithm DSW
  createBackbone(root, n);
  createPerfectTree(n);
End Algorithm
```

Algorithm createBackbone

```
createBackbone(root, n)
  tmp = root;
  while (tmp != null)
    if tmp has a left child
       rotate this child about tmp; // hence the left child
                                    // becomes parent of tmp;
       set tmp to the child which just became parent;
    else set tmp to its right child;
```

Algorithm createBackbone

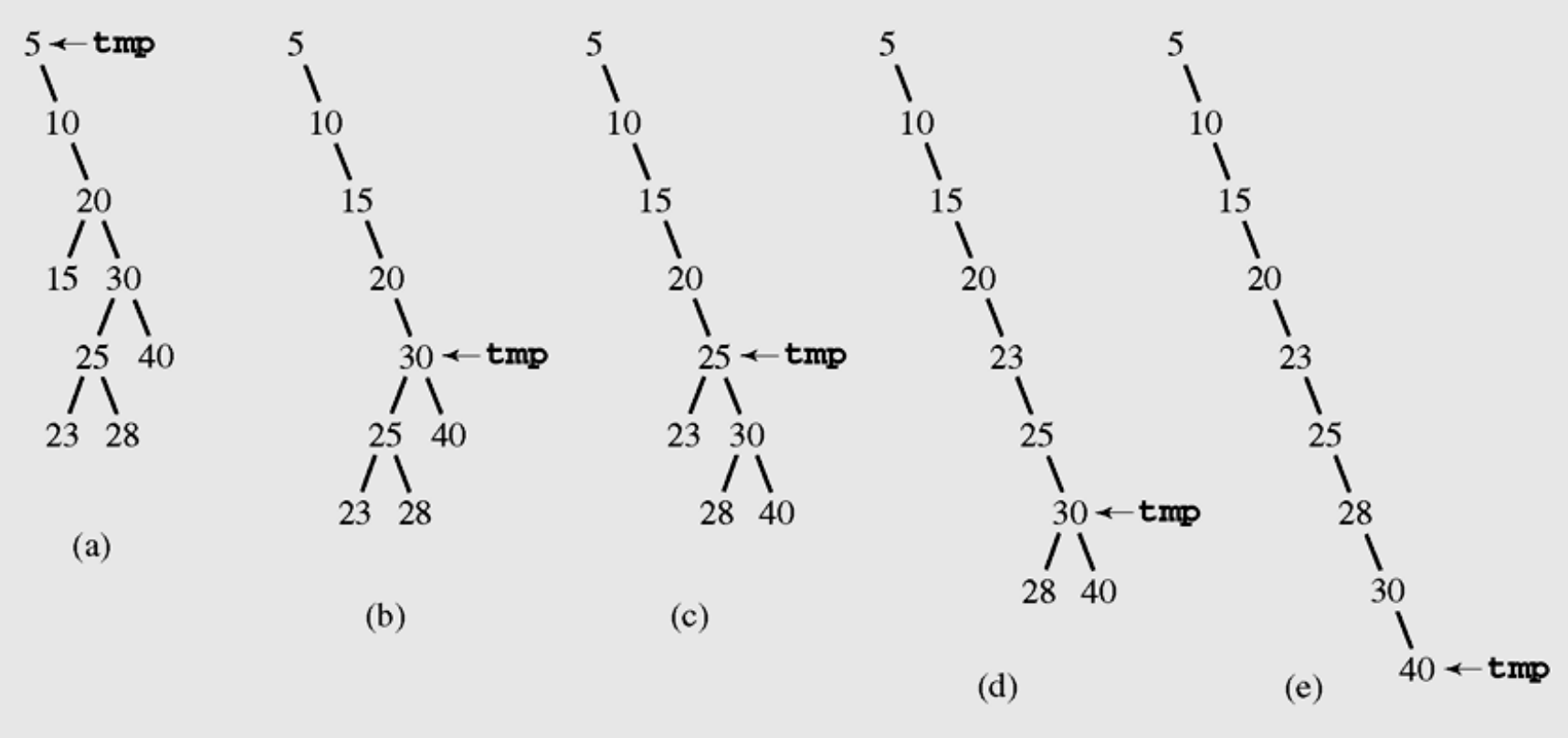


Figure 6-38 Transforming a binary search tree into a backbone

Algorithm createPerfectTree

```
createPerfectTree(n)
   m = 2^⌊lg(n+1)⌋ − 1;
   make n-m rotations starting from the top of backbone;
   while (m > 1)
      m = m/2;
      make m rotations starting from the top of backbone;
```

Algorithm createPerfectTree

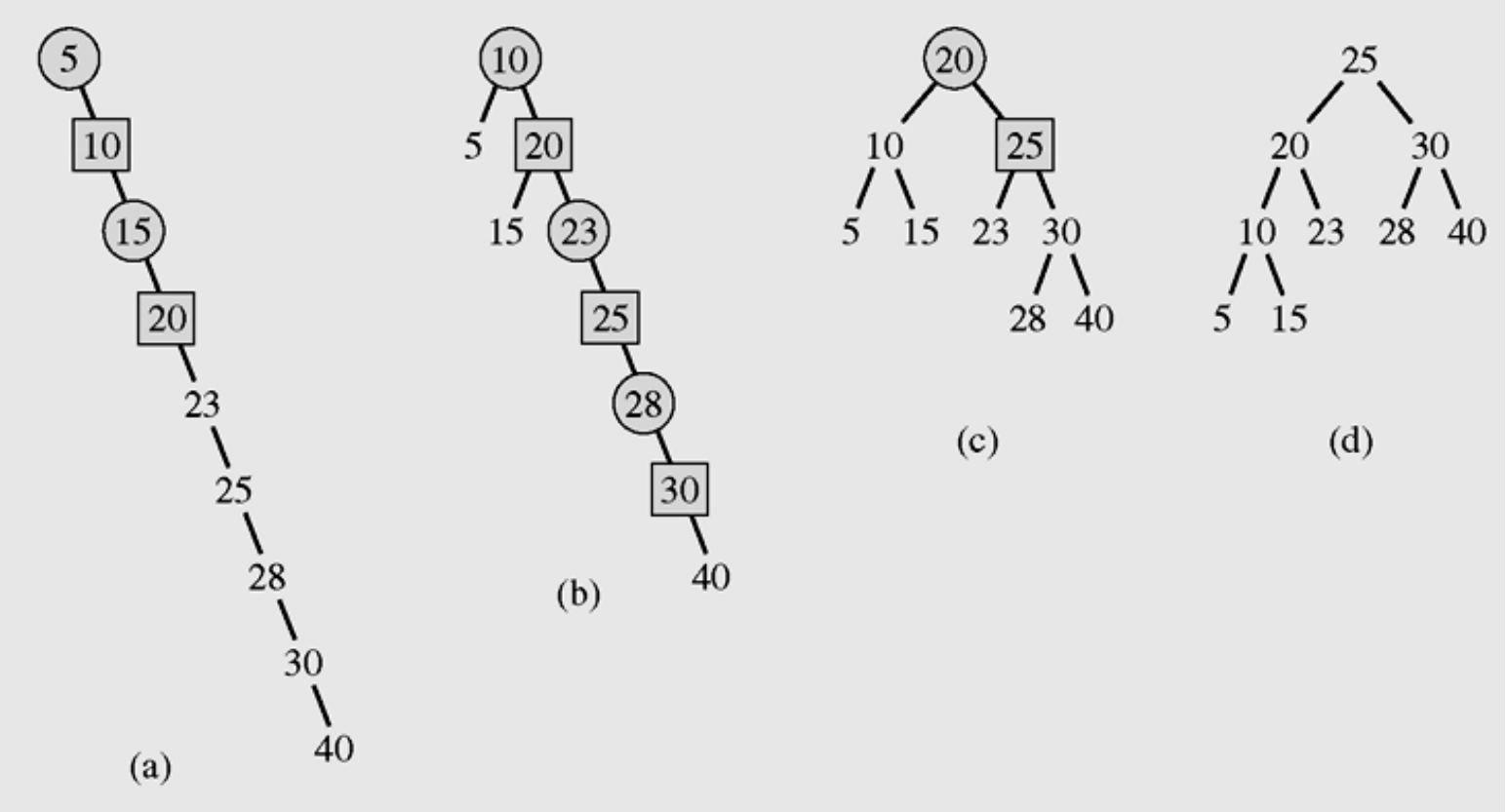


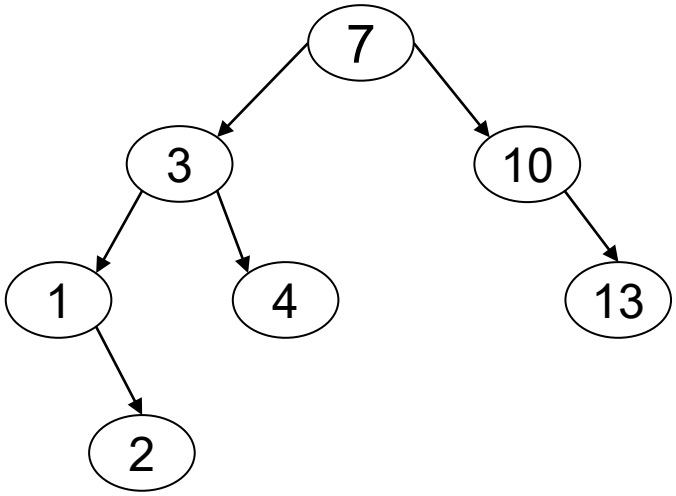
Figure 6-39 Transforming a backbone into a perfectly balanced tree

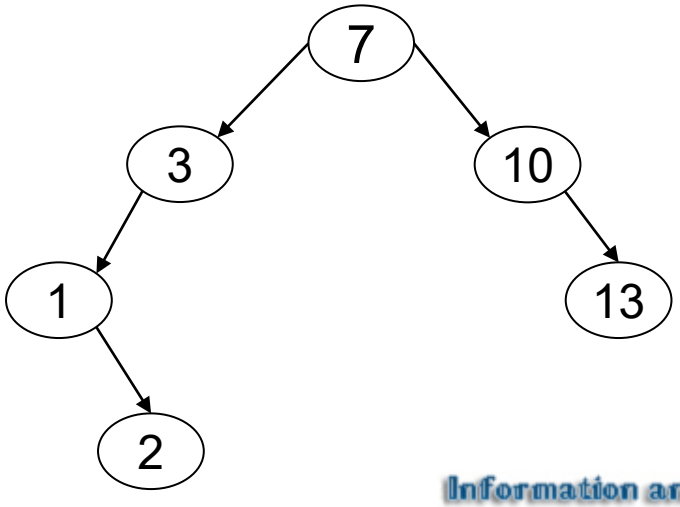
AVL Trees

- An AVL tree is a binary search tree with a height balance property:
  - For each node v, the heights of the subtrees of v differ by at most 1.

- A subtree of an AVL tree is also an AVL tree.
- For each node of an AVL tree:
  Balance factor = height(right subtree) - height(left subtree)

- An AVL node can have a balance factor of -1, 0, or 1.
- Determine whether the trees below are AVL trees or not.



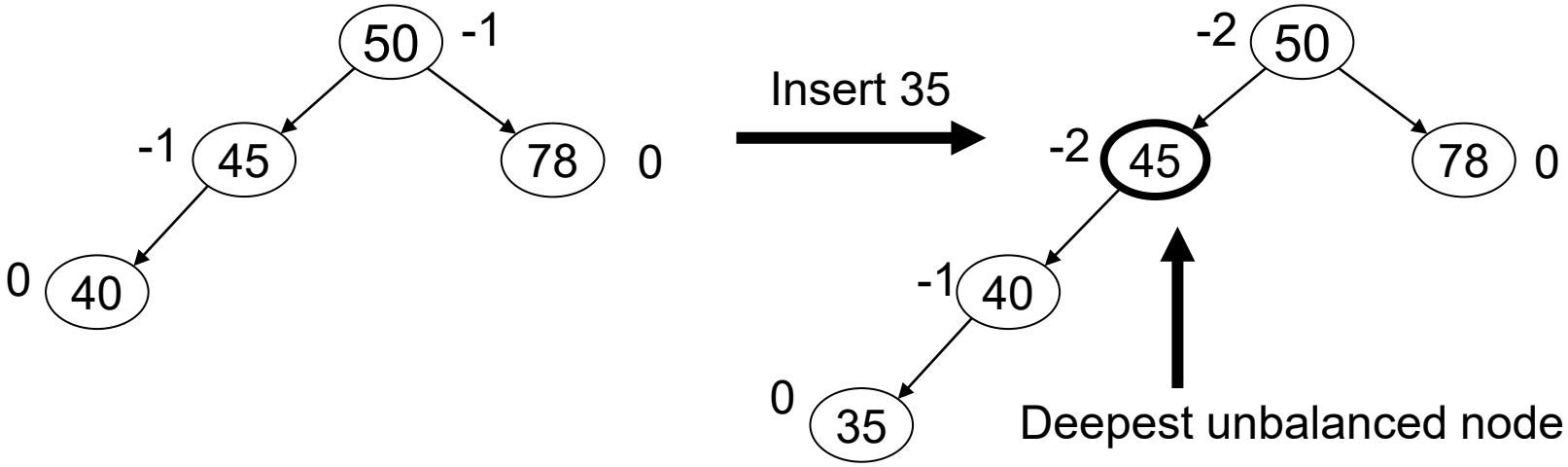


Why AVL Trees?

- Insertion or deletion in an ordinary Binary Search Tree can cause large imbalances.
- In the worst case searching an imbalanced Binary Search Tree is $O(n)$
- An AVL tree is rebalanced after each insertion or deletion.
  - The height-balance property ensures that the height of an AVL tree with n nodes is $O(\log n)$.
  - Searching, insertion, and deletion are all $O(\log n)$.

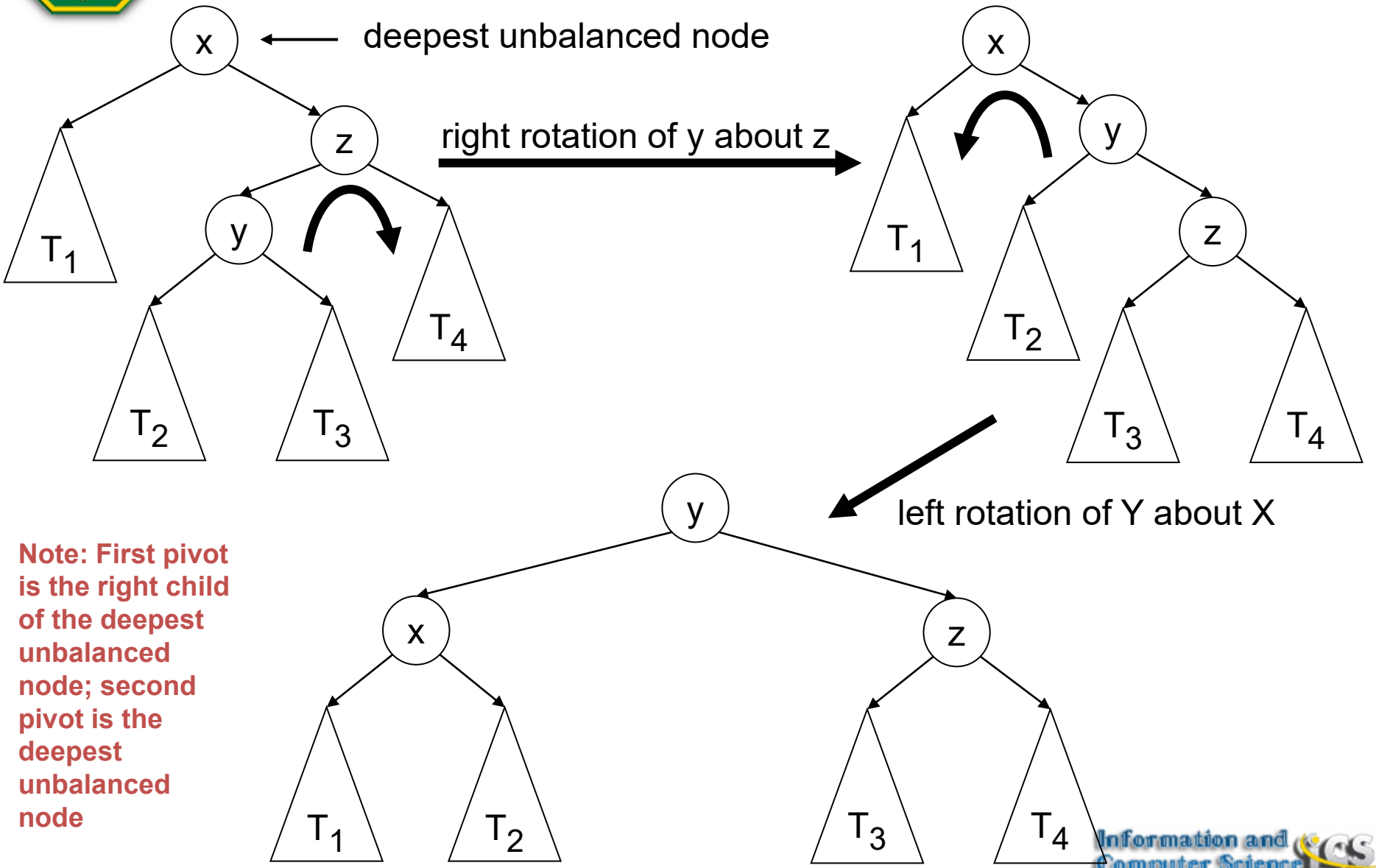
Balancing an AVL Tree

- **An insertion or deletion may cause an imbalance in an AVL tree.**
- The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.



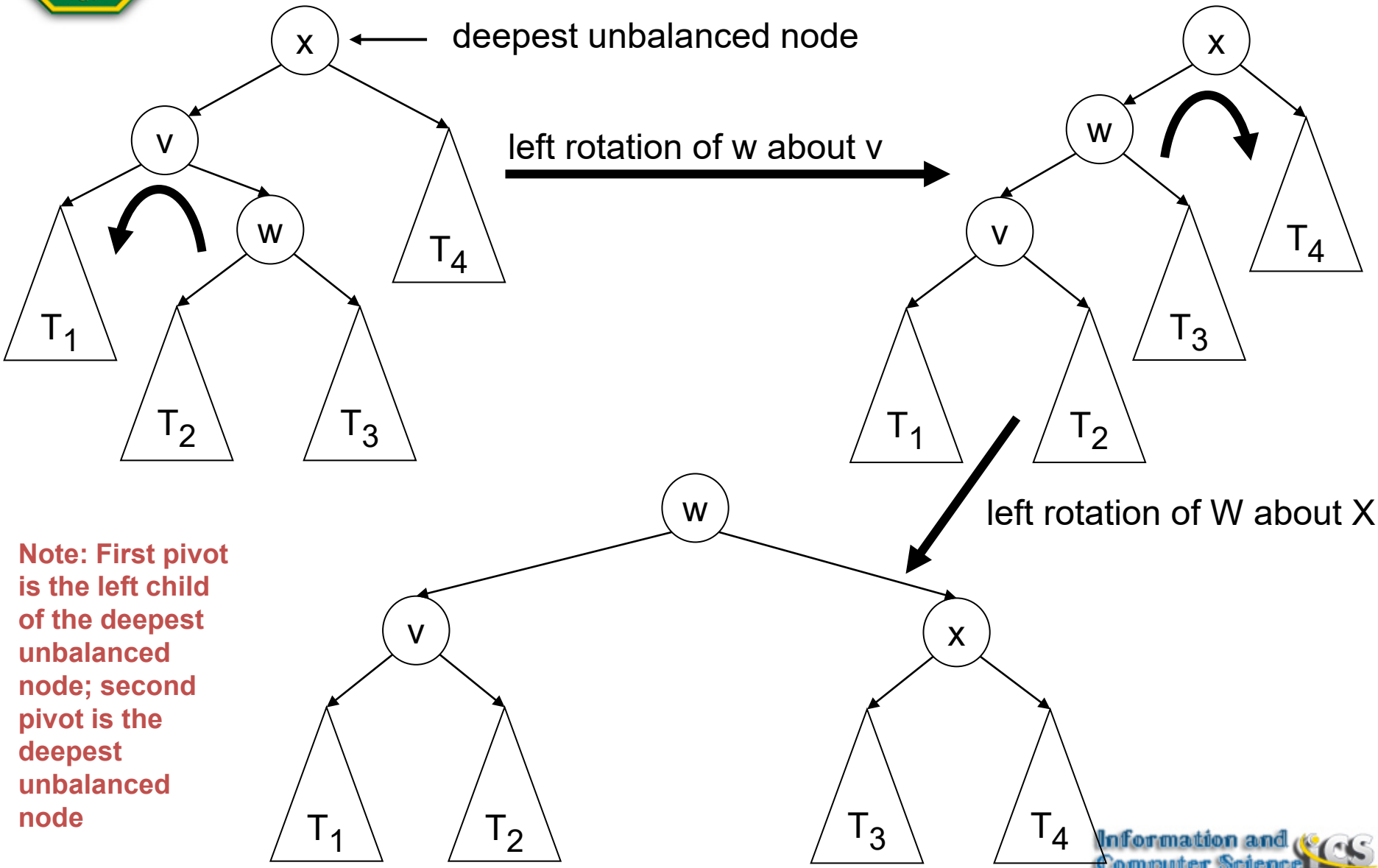
- Balance is restored using a single rotation or a double rotation
  - Single right and left rotations are same as before.
  - A double right-left rotation is a right rotation followed by a left rotation.
  - A double left-right rotation is a left rotation followed by a right rotation.

Double Right-Left Rotation



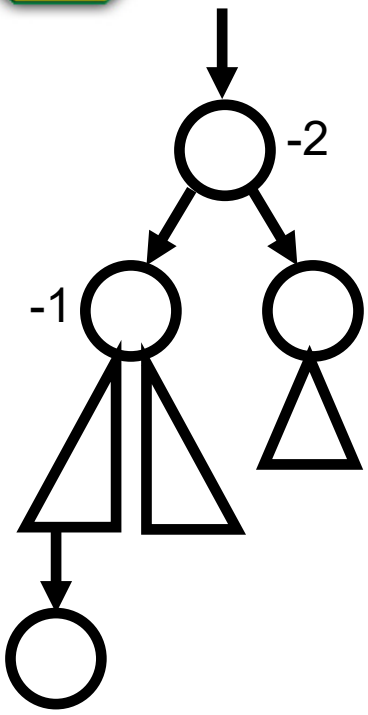
Note: First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node

Double Left-Right Rotation

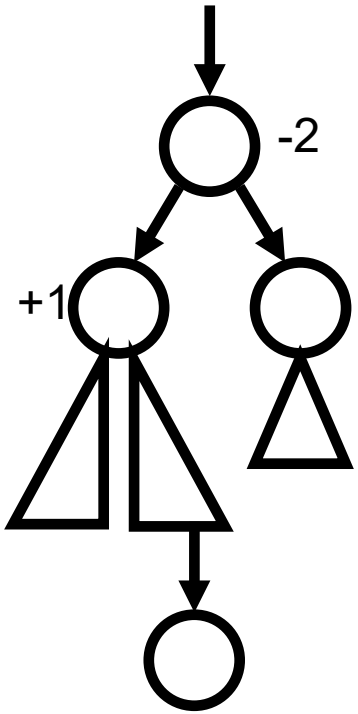


Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

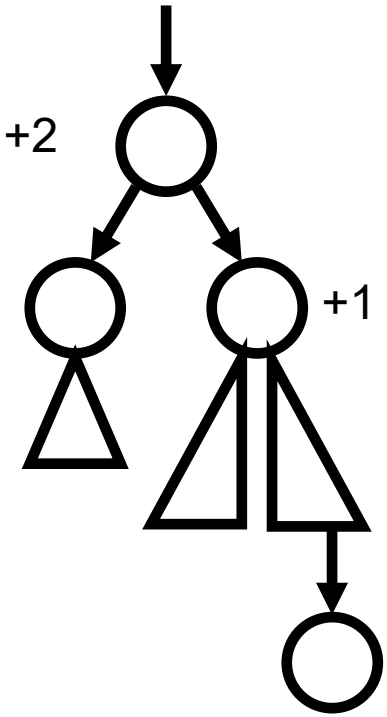
When to do Which Rotation



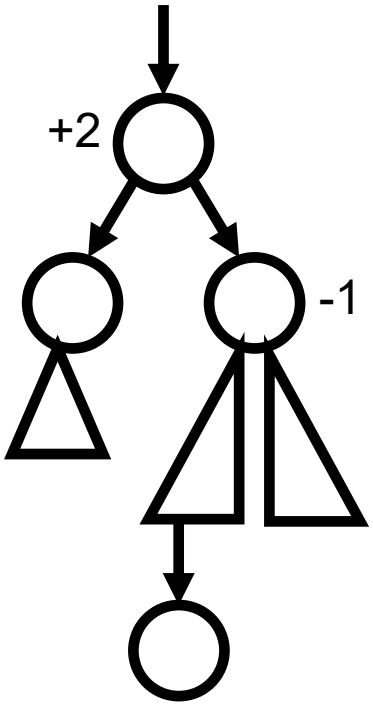
Single right rotation



Double left-right rotation



Single left rotation



Double right-left rotation

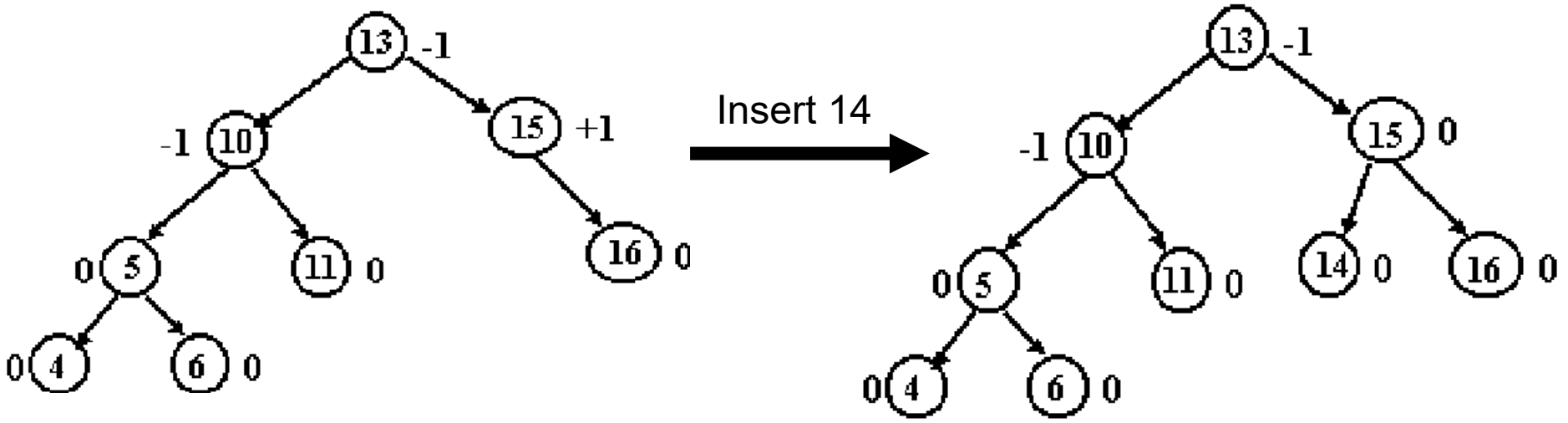
Insertion

- Insert using a BST insertion algorithm.
- Rebalance the tree if an imbalance occurs.
- An imbalance occurs if a node's balance factor changes from -1 to -2 or from+1 to +2.
- Rebalancing is done at the deepest or lowest unbalanced ancestor of the inserted node.

- There are three insertion cases:
  1. Insertion that does not cause an imbalance.
  2. Same side (left-left or right-right) insertion that causes an imbalance.
     - Requires a single rotation to rebalance.
  3. Opposite side (left-right or right-left) insertion that causes an imbalance.
     - Requires a double rotation to rebalance.

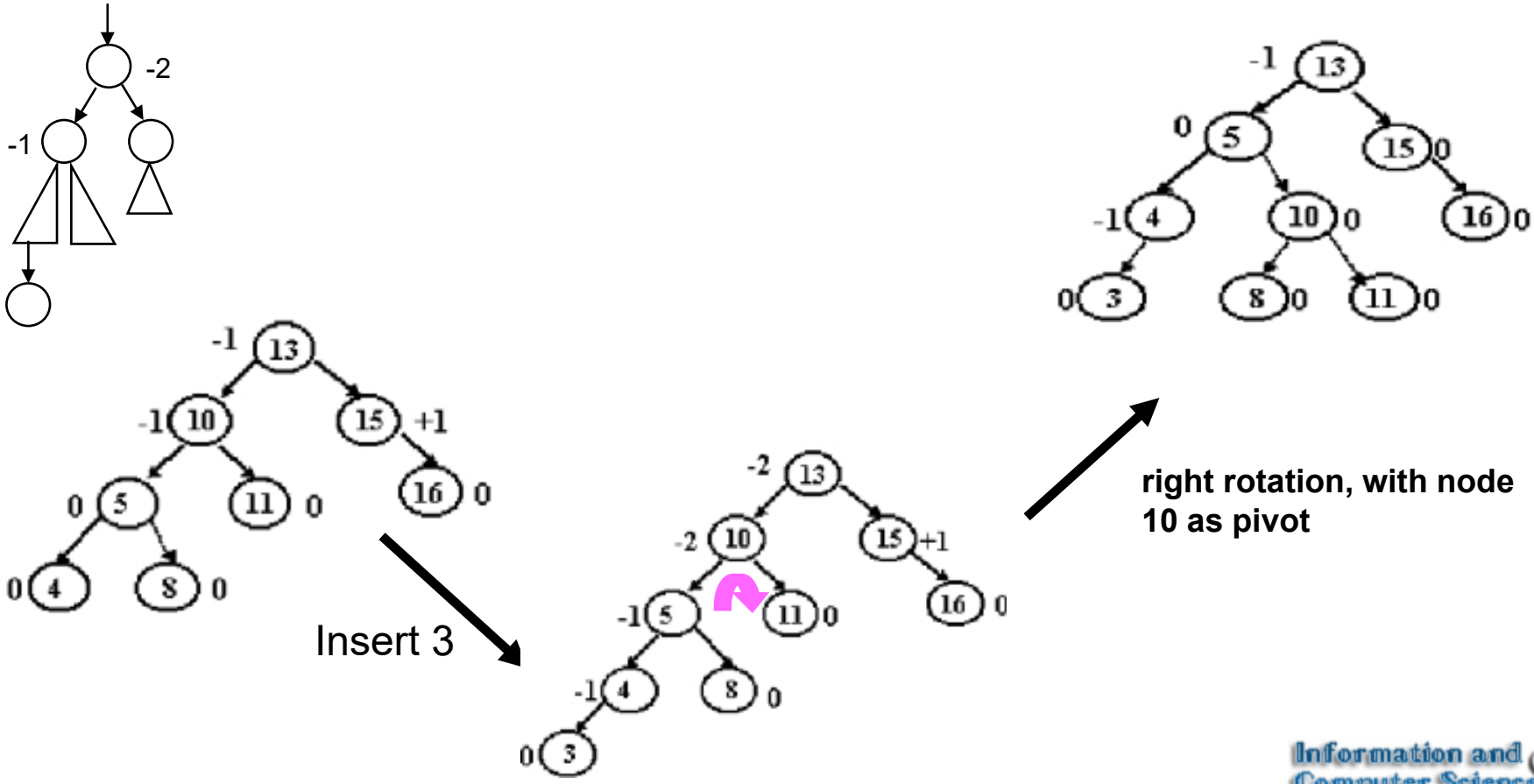
Insertion: case 1

- Example: An insertion that does not cause an imbalance.



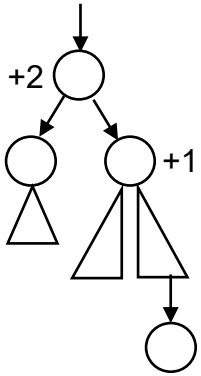
Insertion: case 2

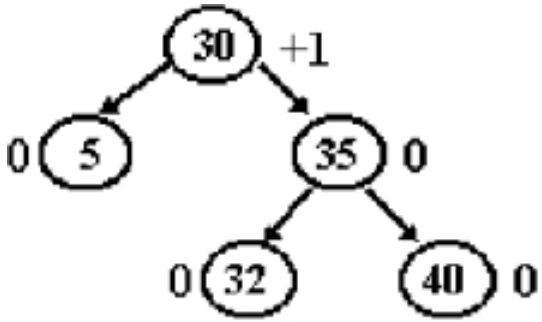
- Case 2a: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the left-subtree of its left child.
- Requires single right rotation to rebalance.



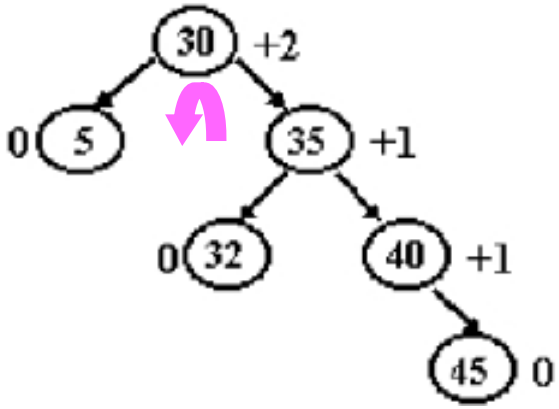
Insertion: case 2 (contd)

- Case 2b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the right-subtree of its right child.
- Requires single left rotation to rebalance.

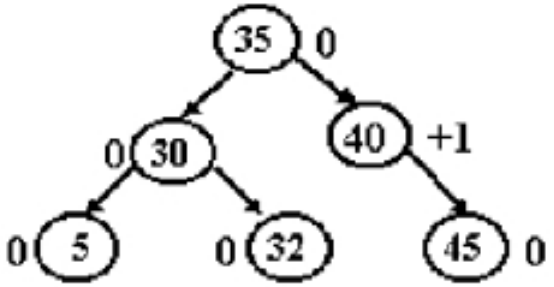




Insert 45

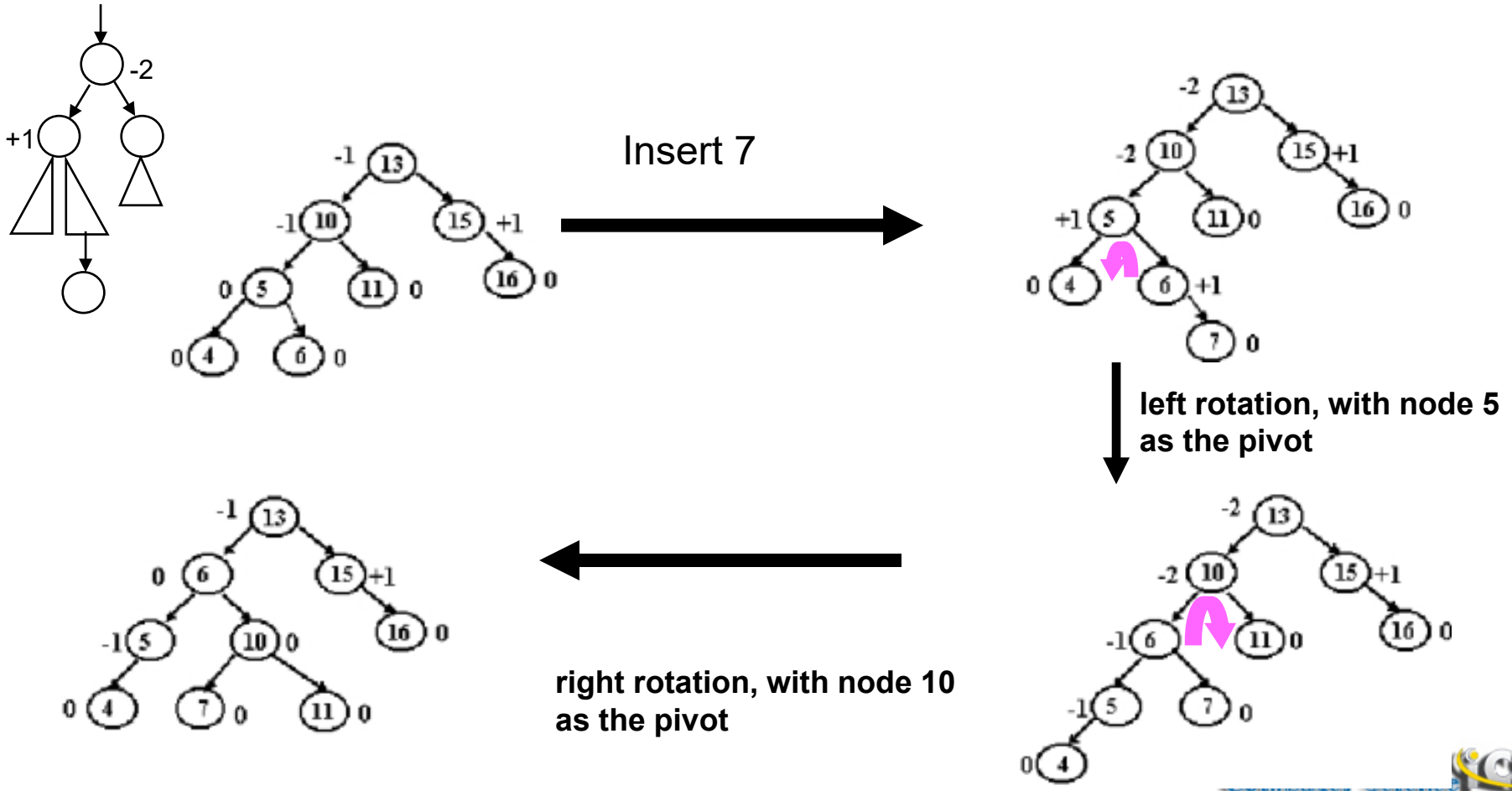


left rotation, with node 30 as the pivot



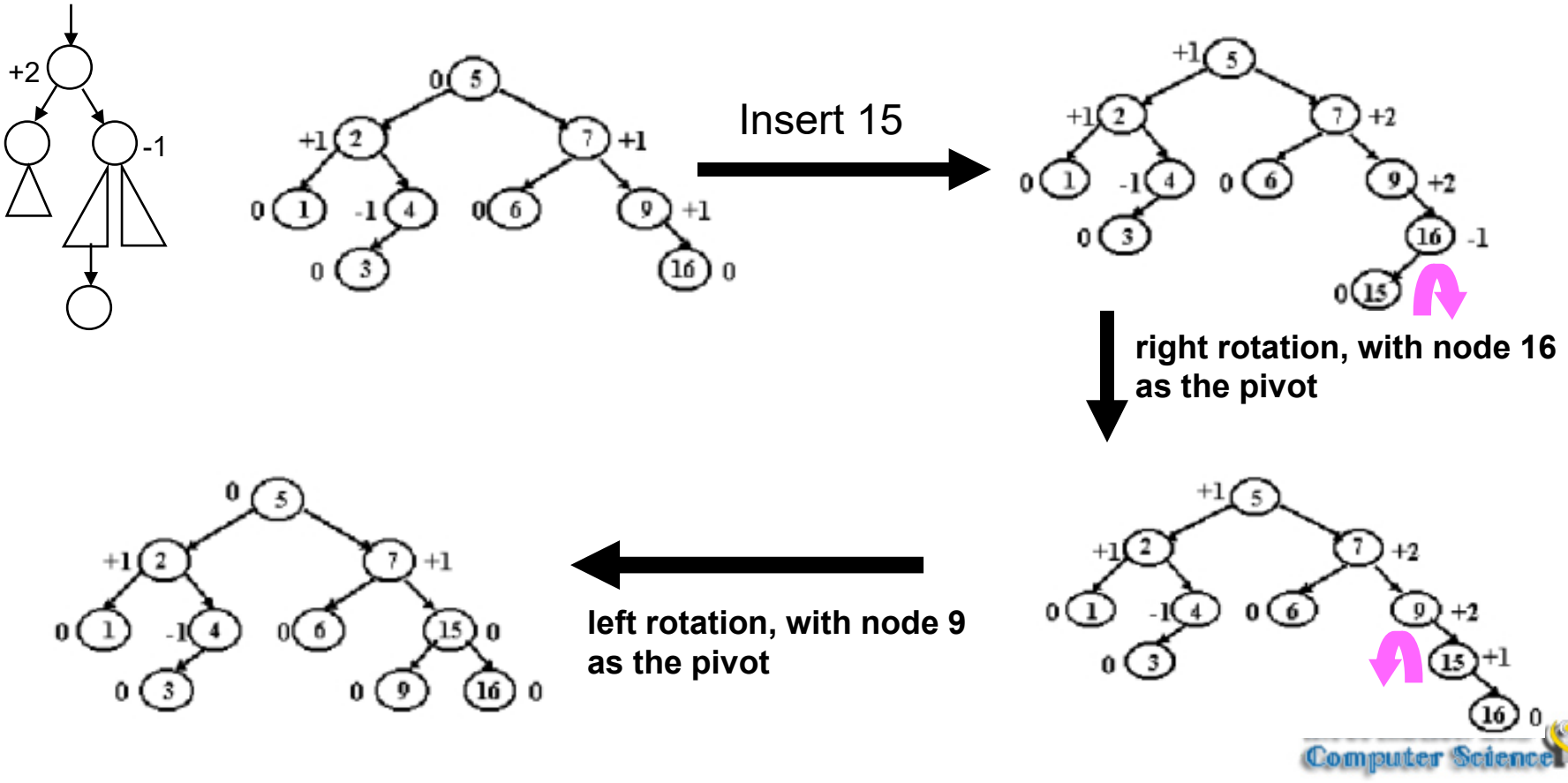
Insertion: case 3

- Case 3a: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the right-subtree of its left child.
- Requires a double left-right rotation to rebalance.



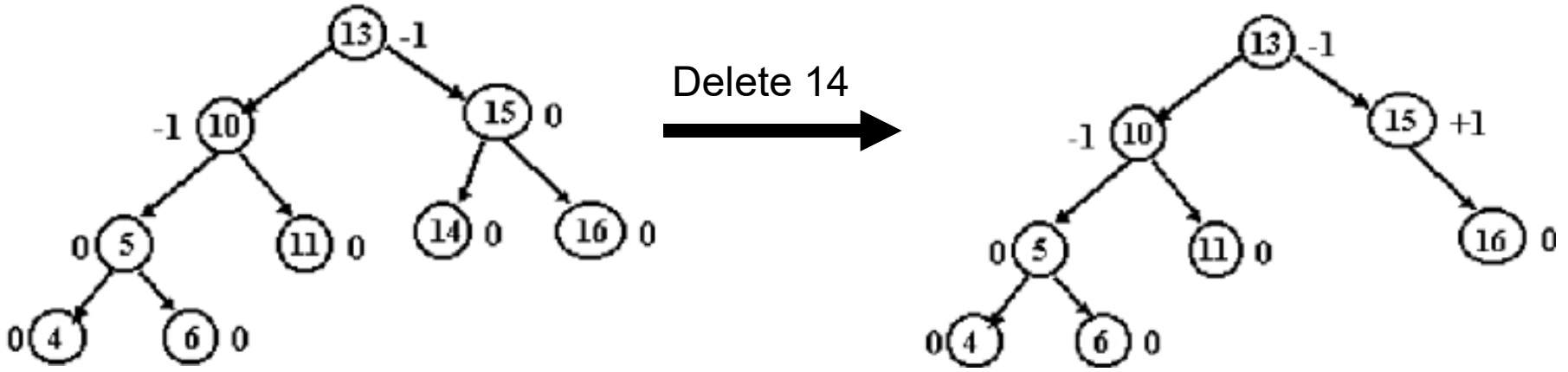
Insertion: case 3 (contd)

- Case 3b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the left-subtree of its right child.
- Requires a double right-left rotation to rebalance.

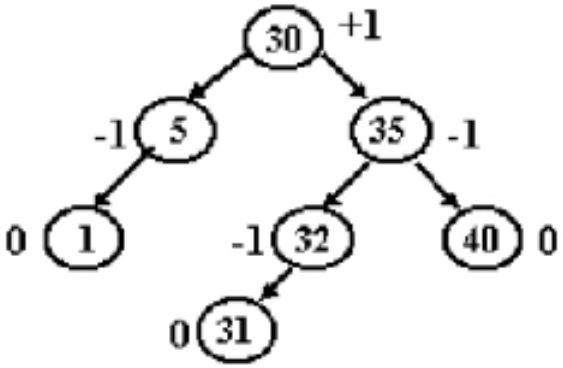


Deletion

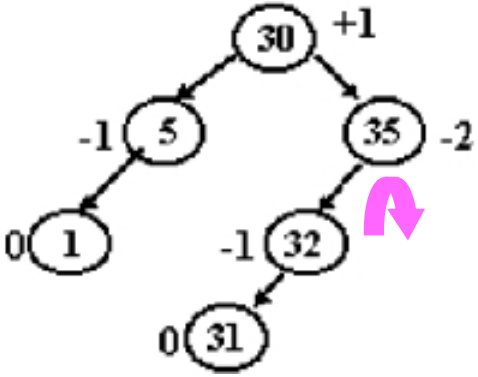
- Delete by a BST deletion by copying algorithm.
- Rebalance the tree if an imbalance occurs.
- There are three deletion cases:
  1. Deletion that does not cause an imbalance.
  2. Deletion that requires a single rotation to rebalance.
  3. Deletion that requires two or more rotations to rebalance.
- Deletion case 1 example:



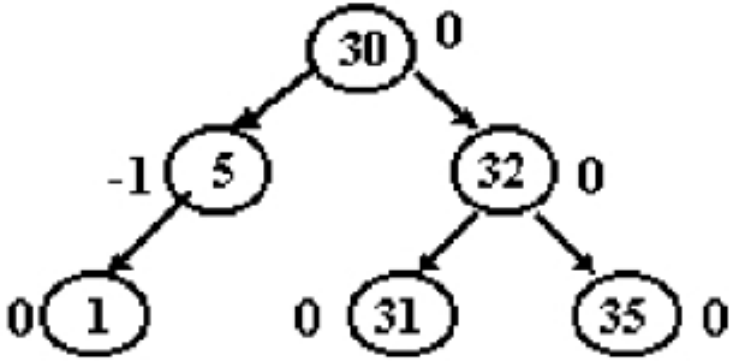
Deletion: case 2 examples



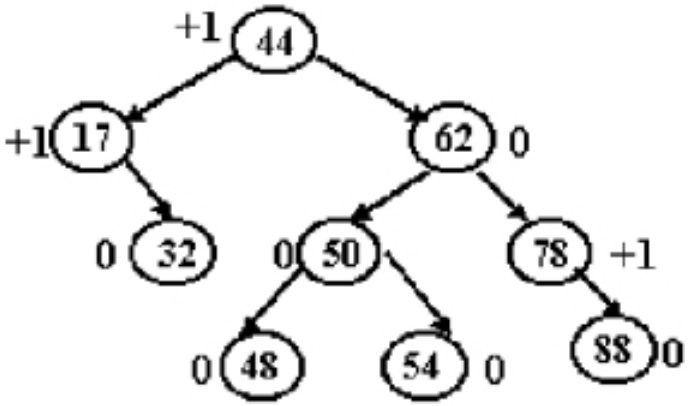
Delete 40



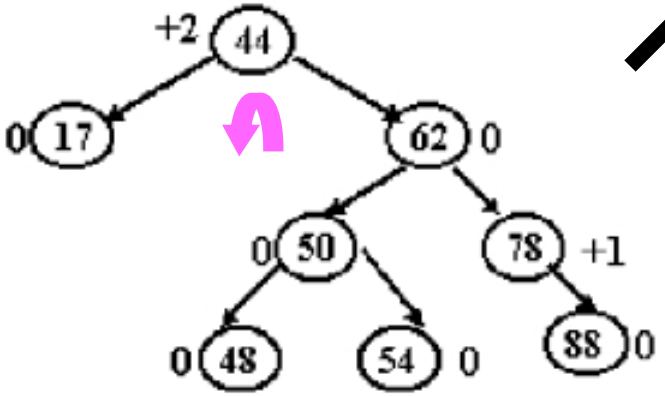
right rotation, with node 35 as the pivot



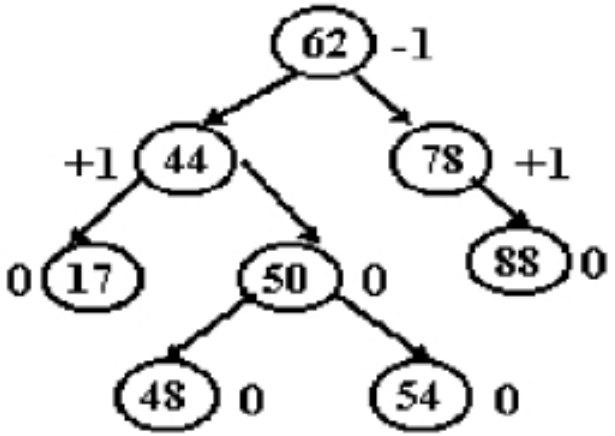
Deletion: case 2 examples (contd)



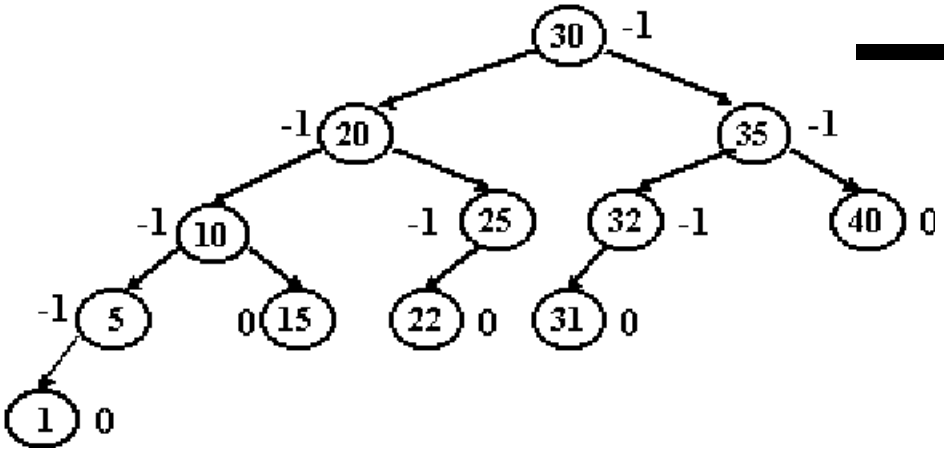
Delete 32



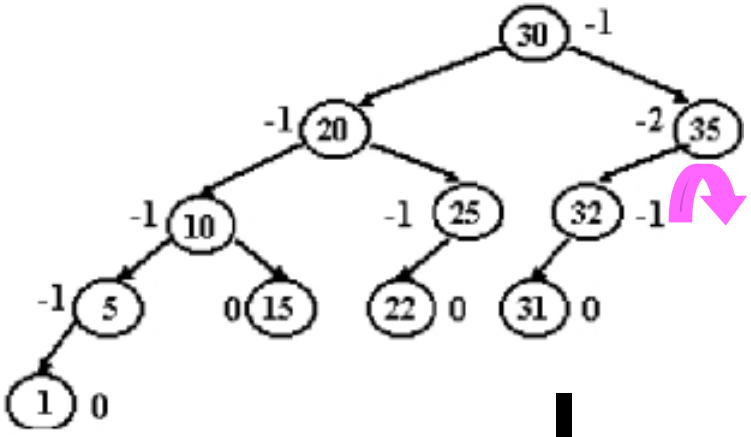
left rotation, with node 44 as the pivot



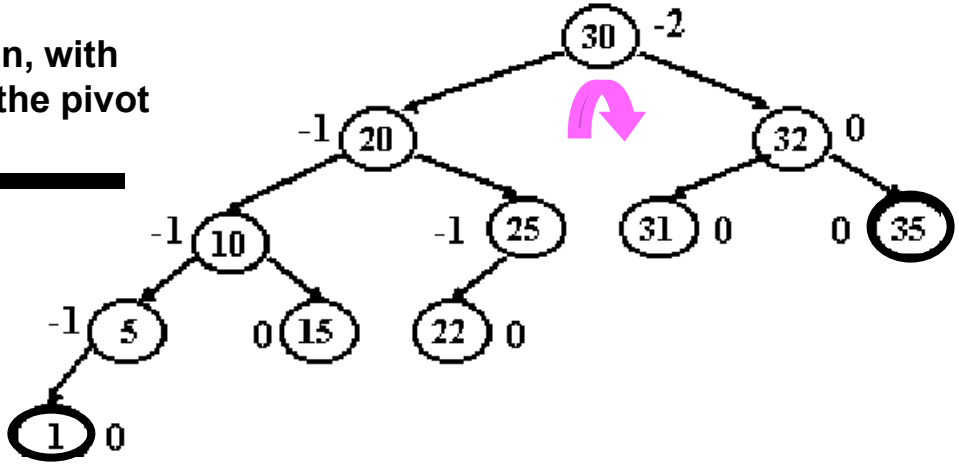
Deletion: case 3 examples



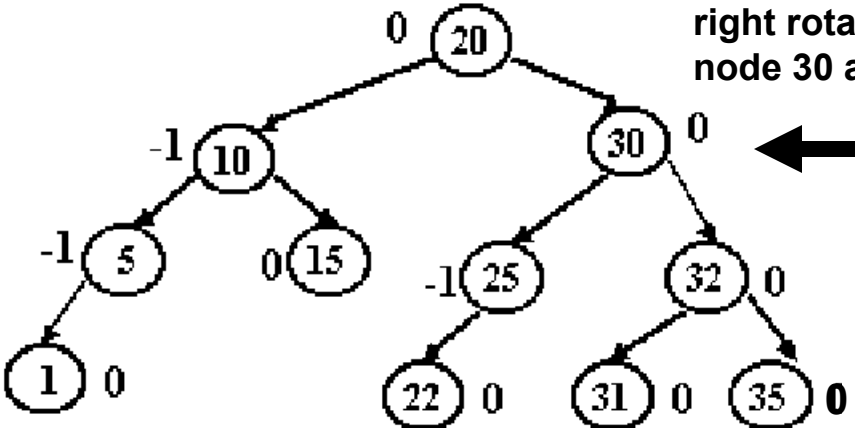
Delete 40



right rotation, with node 35 as the pivot



right rotation, with node 30 as the pivot

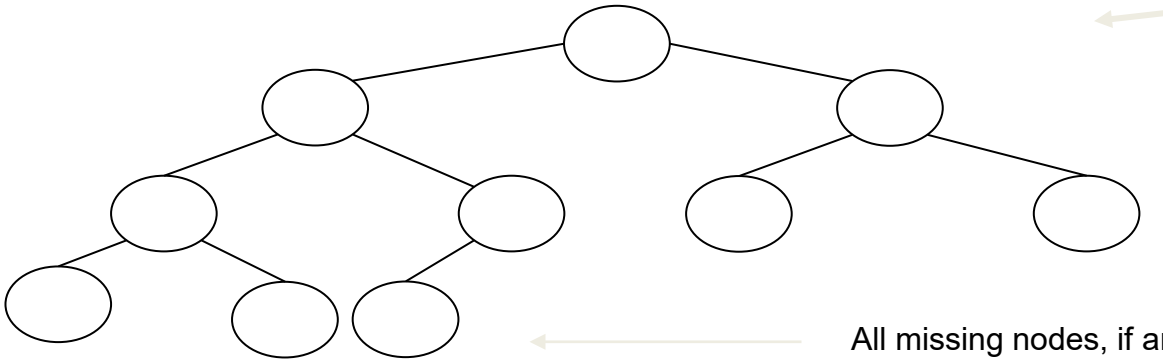


Binary Heaps

- What is a Binary Heap?
- Array representation of a Binary Heap
- MinHeap implementation
- Operations on Binary Heaps:
  - enqueue
  - dequeue
  - deleting an arbitrary key
  - changing the priority of a key
- Building a binary heap
  - top down approach
  - bottom up approach
- Heap as a priority queue

What is a Binary Heap

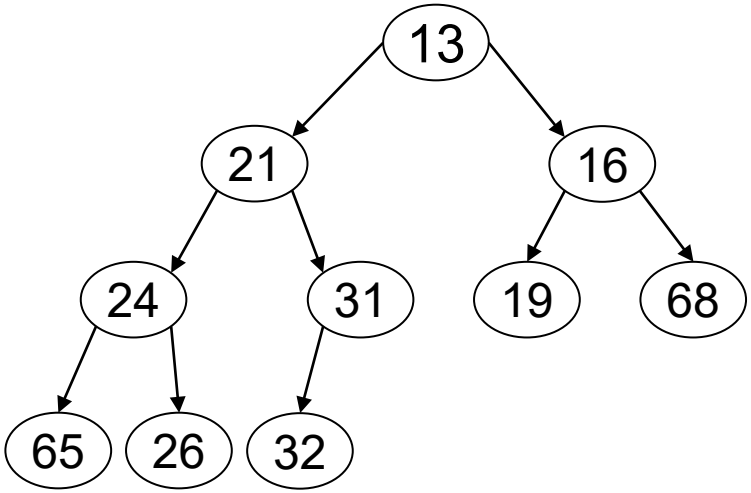
- A particular kind of binary tree, called a **heap**, has two properties:
  - The value of each node is greater than or equal to the values stored in each of its children
  - The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions
- These two properties define a **max heap**
- If "greater" in the first property is replaced with "less," then the definition specifies a **min heap**

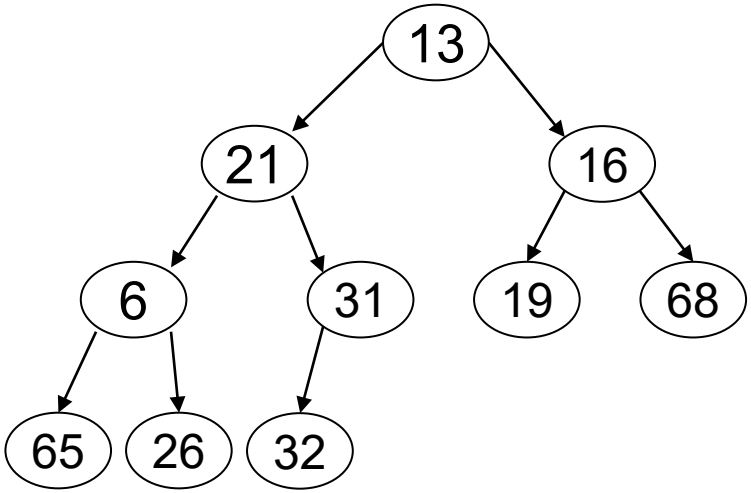


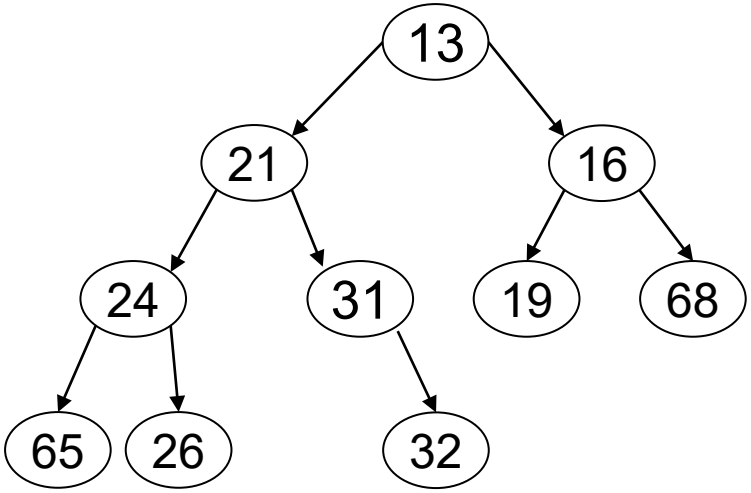
All levels except the bottom one must be fully populated with nodes

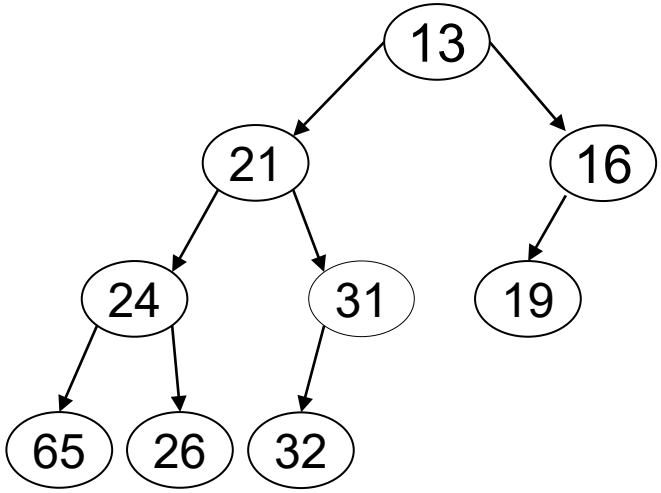
All missing nodes, if any, must be on the right side of the lowest level

MinHeap and non-MinHeap examples

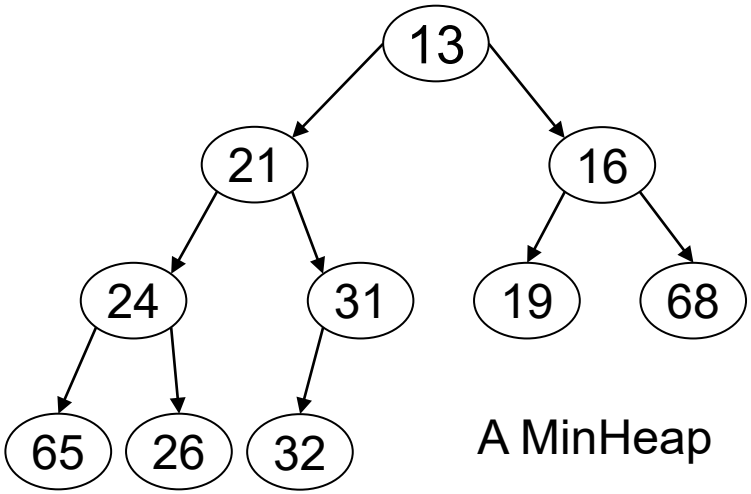






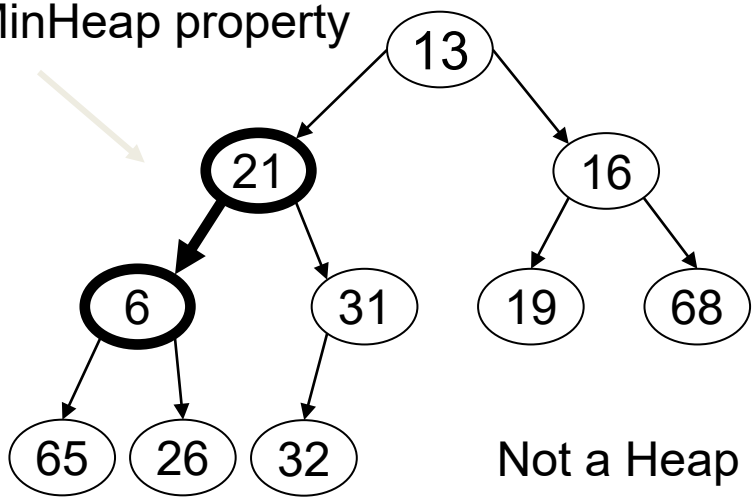


MinHeap and non-MinHeap examples

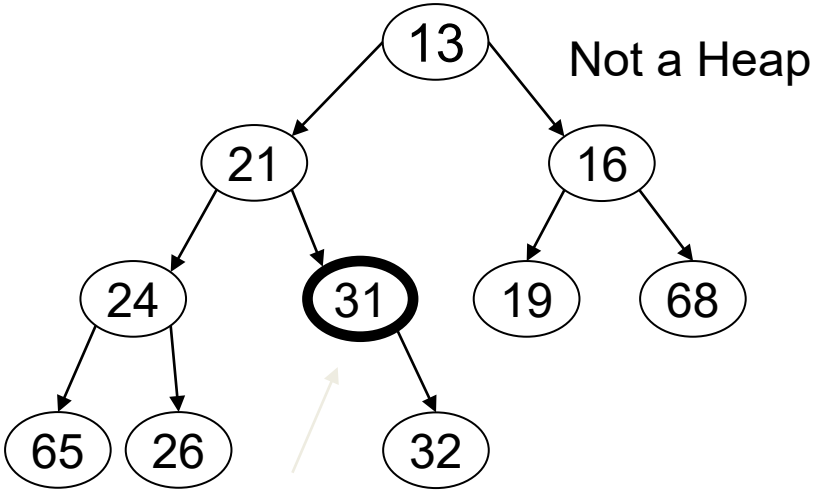


A MinHeap

Violates MinHeap property
21>6

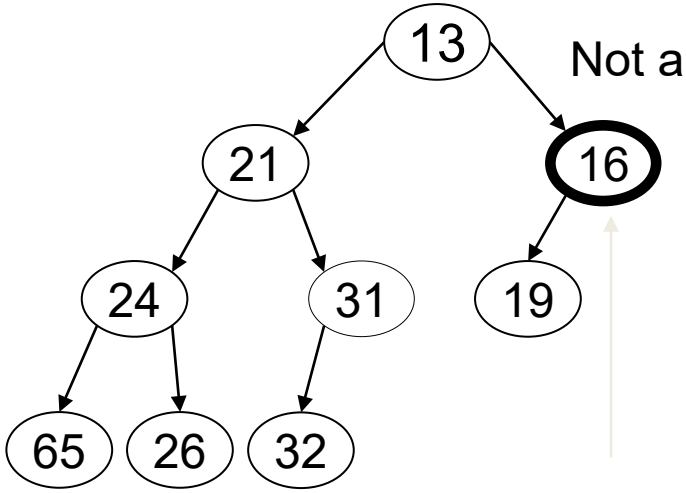


Not a Heap



Not a Heap

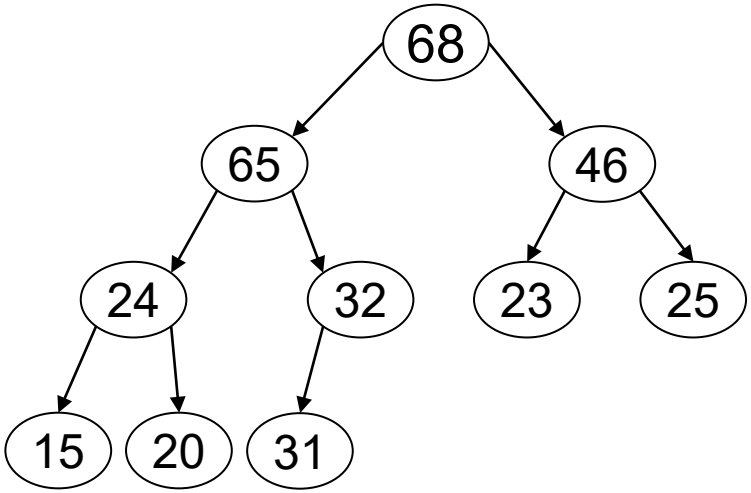
Violates heap structural property

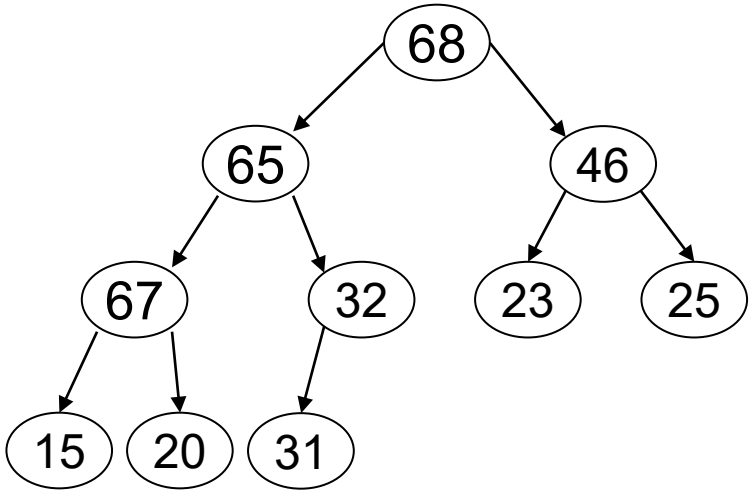


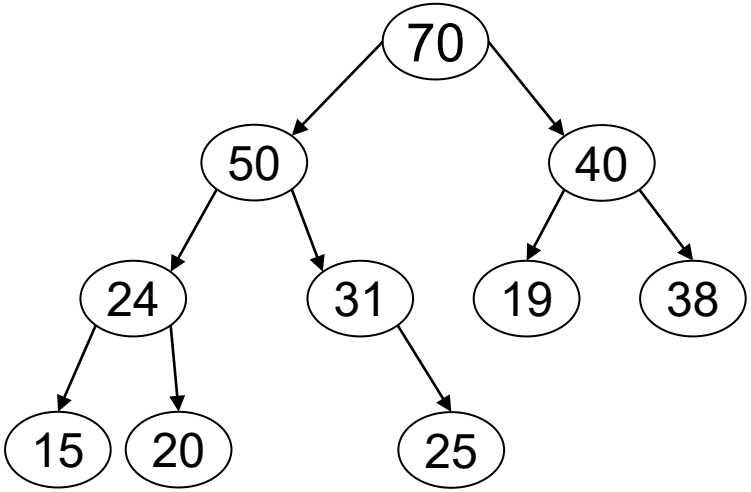
Not a Heap

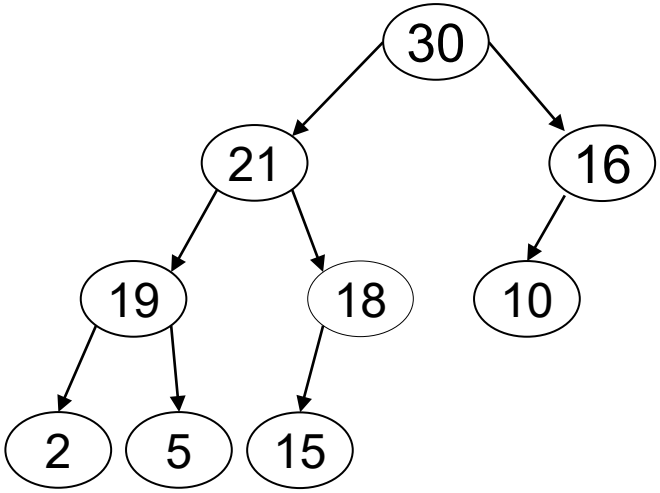
Violates heap structural property

MaxHeap and non-MaxHeap examples

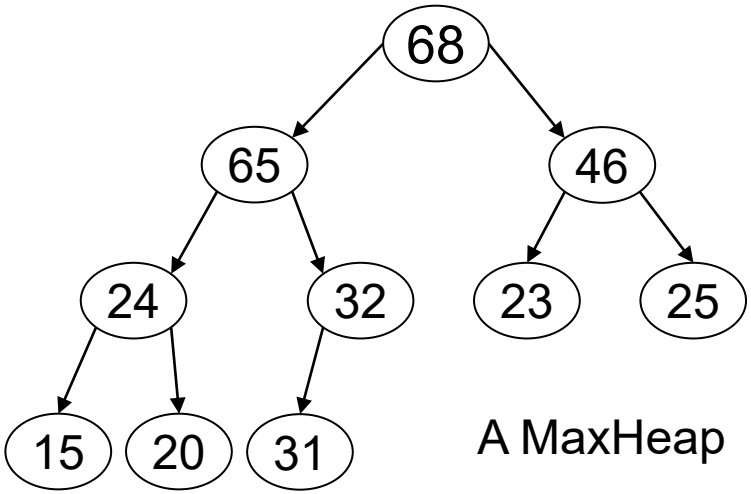






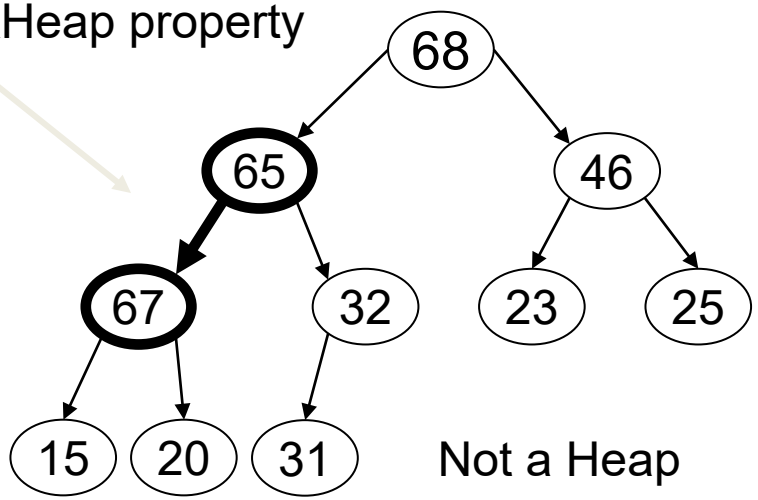


MaxHeap and non-MaxHeap examples

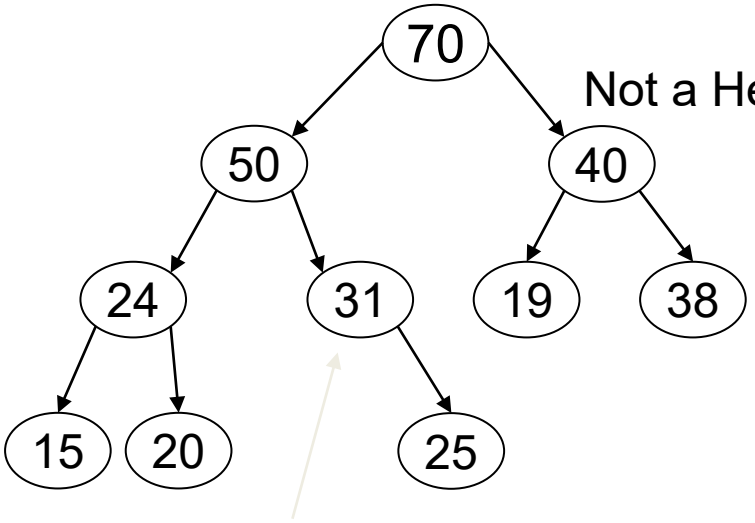


A MaxHeap

Violates MaxHeap property
65 < 67



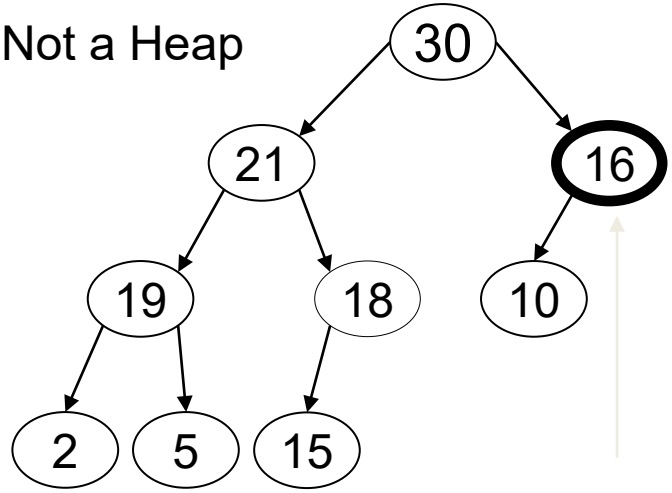
Not a Heap



Not a Heap

Violates heap structural property

Not a Heap

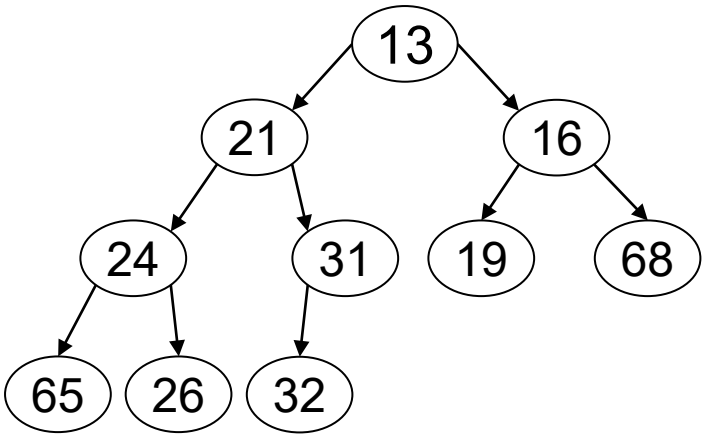


Violates heap structural property

Array Representation of a Binary Heap

- A heap is a dynamic data structure that is represented and manipulated more efficiently using an array.
- Since a heap is a complete binary tree, its node values can be stored in an array, without any gaps, in a breadth-first order, where:

Value(node $_{i+1}$) ⟶ array[i], for i ≥ 0



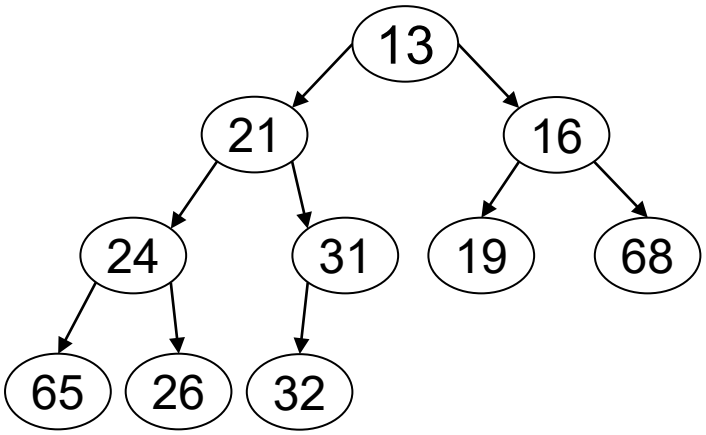
13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9

- The root is array[0]
- The parent of array[i] is array[(i – 1)/2], where i > 0
- The left child, if any, of array[i] is array[2i+1].
- The right child, if any, of array[i] is array[2i+2].

Array Representation of a Binary Heap (contd.)

- We shall use an implementation in which the heap elements are stored in an array starting at index 1.

Value(node $_i$) ⟶ array[i] , for $i \geq 1$



	13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9	10

- The root is array[1].
- The parent of array[i] is array[i/2], where i > 1
- The left child, if any, of array[i] is array[2i].
- The right child, if any, of array[i] is array[2i+1].

Percolate Up

- In a MinHeap, if the key at a node, other than the root, becomes less than its parent, the heap property can be restored by swapping the current node and its parent, repeating this process for the parent if necessary, until
  - the key at the node is greater than or equal to that of the parent.
  - we reach the root.

```
Procedure percolateUp
Input:  H[1..n], i where 1 ≤ i ≤ n.
Output: H, where no node is less than its parent on the path from
node i to the root.
 done = false;
 while (!done && (i != 1)) {
   if H[i].key < H[i/2].key
      swap(H[i],H[i/2]);
   else
      done = true;
   i := i/2;
 }
```

What is the complexity of percolateUp?

Percolate Down

- In a MinHeap, if the value at a node becomes greater than the key of any of its children, the heap property can be restored by swapping the current node and the child with minimum key value, repeating this process if necessary until
  - the key at the node is less than or equal to the keys of both children.
  - we reach a leaf.

```
Procedure percolateDown
Input:  H[1..n], i where 1 ≤ i ≤ n.
Output: H[i] is percolated down, if needed, so that it's not
        greater than its children.
 done = false;
 while ( (2*i <= n) && !done) {
   i = 2*i;
   if ((i+1 ≤ n) and (H[i+1].key < H[i].key))   i = i+1;
   if (H[i/2].key > H[i].key)
      swap(H[i],H[i/2]);
   else
      done := true;
 }
```

What is the complexity of percolateDown?

MinHeap enqueue

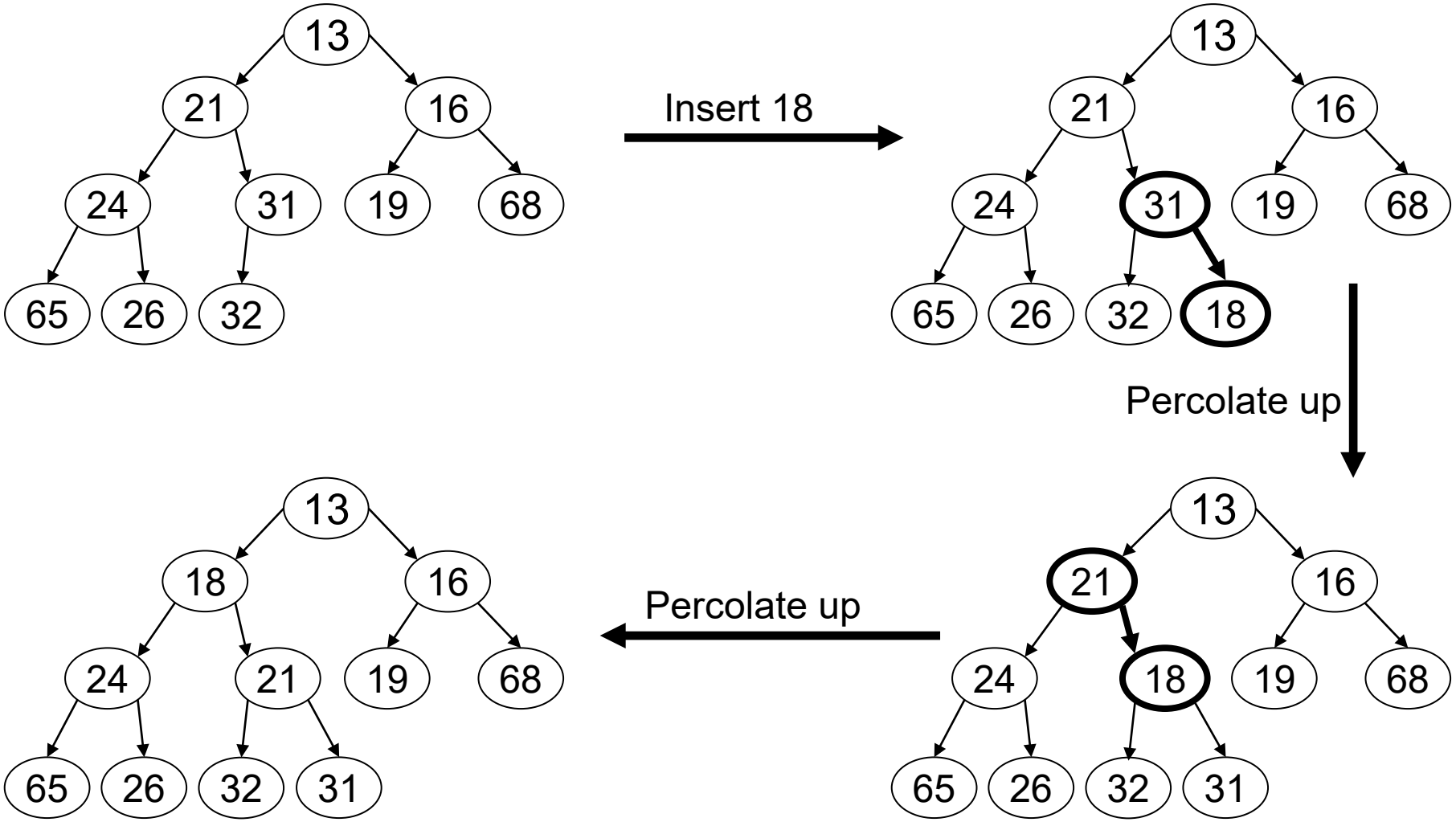
- The pseudo code algorithm for enqueing a key in a MinHeap is:

```
Algorithm enqueue
Input: A heap H[1..n] & a heap element x.
Output: A new heap H[1..n+1] with x
        being one of its elements.
1. if (Heap is full) throw an exception;
2. n = n + 1;
3. H[n] = x;
4. percolateUp(H, n);
```

What is the complexity of enqueue method?

- Thus, the steps for enqueue are:
  1. Enqueue the key at the end of the heap.
  2. As long as the heap order property is violated, percolate up.

MinHeap Insertion Example



Deleting an Arbitrary Key

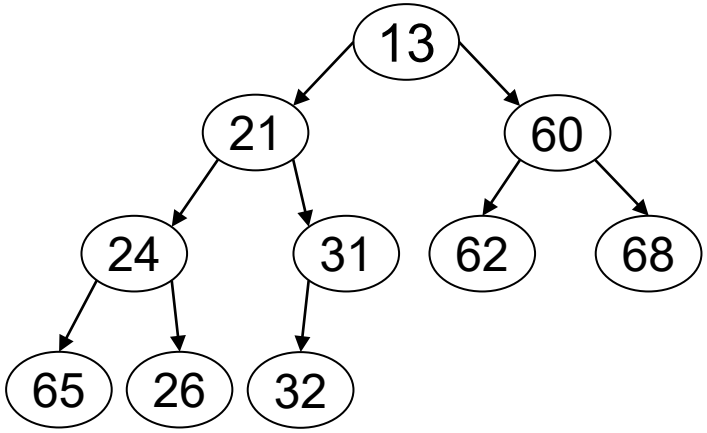
```
Algorithm Delete
Input: A nonempty heap H[1..n] and i where
       1 ≤ i ≤ n.
Output: H[1..n-1] after H[i] is removed.

1. if (Heap is empty) throw an exception
2. x = H[i]; y = H[n];
3. n := n - 1;
4. if i == n+1 then return; // deleting last node
5. H[i] = y;
6. if y.key <= x.key then
7.   percolateUp(H, i);
8. else percolateDown(H, i);
```

What is the complexity of Delete method?

- What about dequeueMin()?

Example



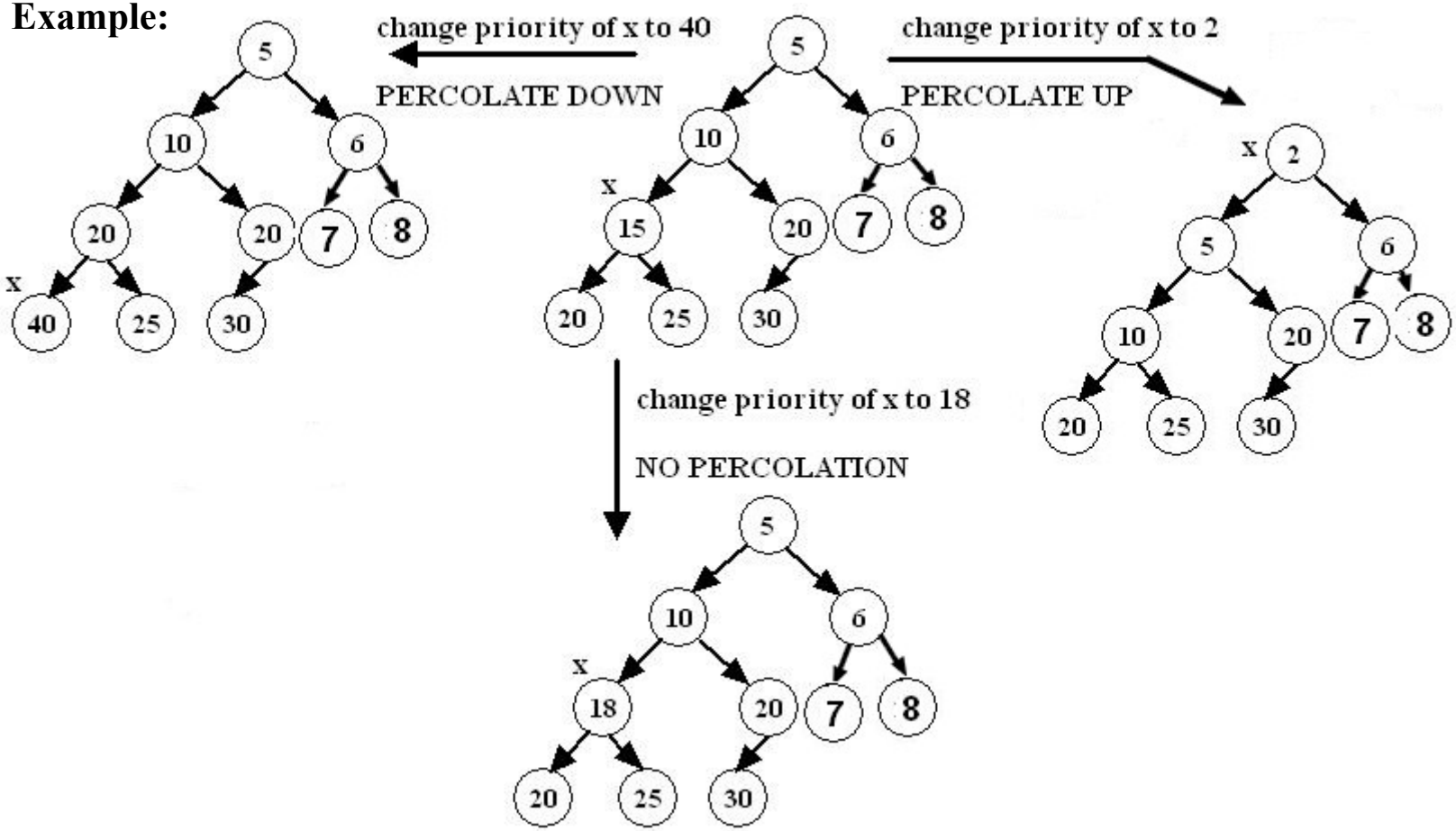
- Delete 68
- Delete 13

Changing the priority of a key

There are three possibilities when the priority of a key **x** is changed:

1. The heap property is not violated.
2. The heap property is violated and **x** has to be percolated up to restore the heap property.
3. The heap property is violated and **x** has to be percolated down to restore the heap property.

Example:



Building a heap (top down)

- A heap is built top-down by inserting one key at a time in an initially empty heap.
- After each key insertion, if the heap property is violated, it is restored by percolating the inserted key upward.

The algorithm is:

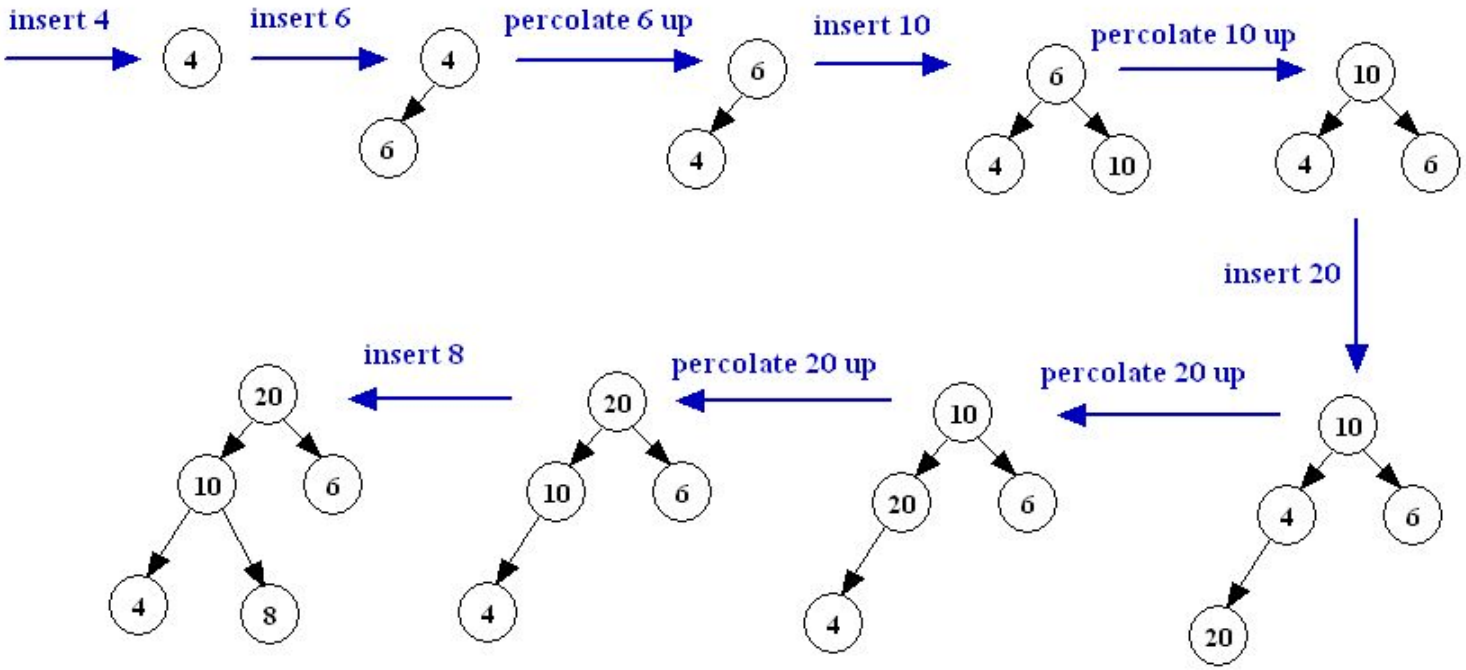
```
for(int i=1; i <= heapSize; i++){
   read  key;
   binaryHeap.enqueue(key);
}
```

What is the complexity of BuildHeap top-down?

Example: Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap

Building a heap (top down)

Example: Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap



Converting an array into a Binary heap (Building a heap bottom-up)

- Robert Floyd developed an algorithm to convert an array into a binary heap as follows:

1. Start at the level containing the last non-leaf node (i.e., array[n/2], where n is the array size).
2. Make the subtree rooted at the last non-leaf node into a heap by invoking percolateDown.
3. Move in the current level from right to left, making each subtree, rooted at each encountered node, into a heap by invoking percolateDown.
4. If the levels are not finished, move to a lower level then go to step 3.

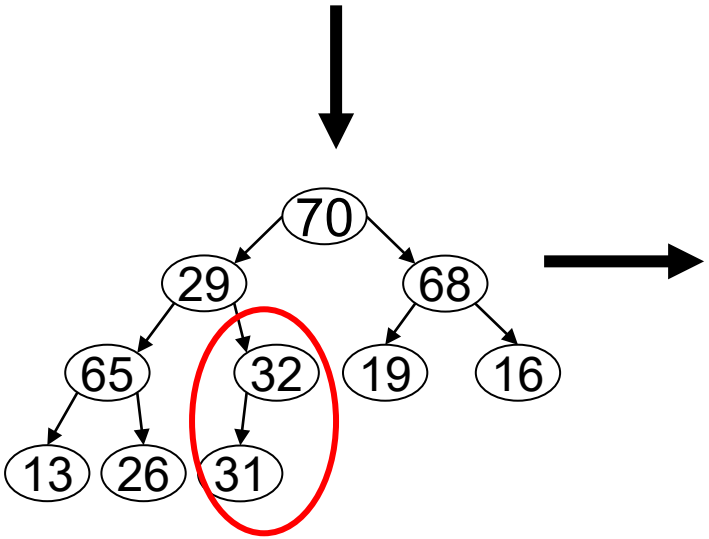
- The above algorithm can be refined to the following method of the BinaryHeap class:

```
private void buildHeapBottomUp()
{
 for(int i = count / 2; i >= 1; i--)
            percolateDown(i);
}
```

- BuildHeapBottomUp runs in O(n) time.

Converting an array into a MinHeap (Example)

70	29	68	65	32	19	16	13	26	31



Heap Applications: Priority Queue

- A heap can be used as the underlying implementation of a priority queue.
- A priority queue is a data structure in which the items to be inserted have associated priorities.
- Items are withdrawn from a priority queue in order of their priorities, starting with the highest priority item first.
- Priority queues are often used in resource management, simulations, and in the implementation of some algorithms (e.g., some graph algorithms, some backtracking algorithms).
- Several data structures can be used to implement priority queues. Below is a comparison of some:

Data structure	Enqueue	Find Max	Dequeue Max
Unsorted List	O(1)	O(n)	O(n)
Sorted List	O(n)	O(1)	O(1)
AVL Tree	O(log n)	O(log n)	O(log n)
MaxHeap	O(log n)	O(1)	O(log n)

Polish Notation and Expression Trees

- **Polish notation** is a special notation for propositional logic that eliminates all parentheses from formulas
- The compiler rejects everything that is not essential to retrieve the proper meaning of formulas rejecting it as "syntactic sugar"

Polish Notation and Expression Trees

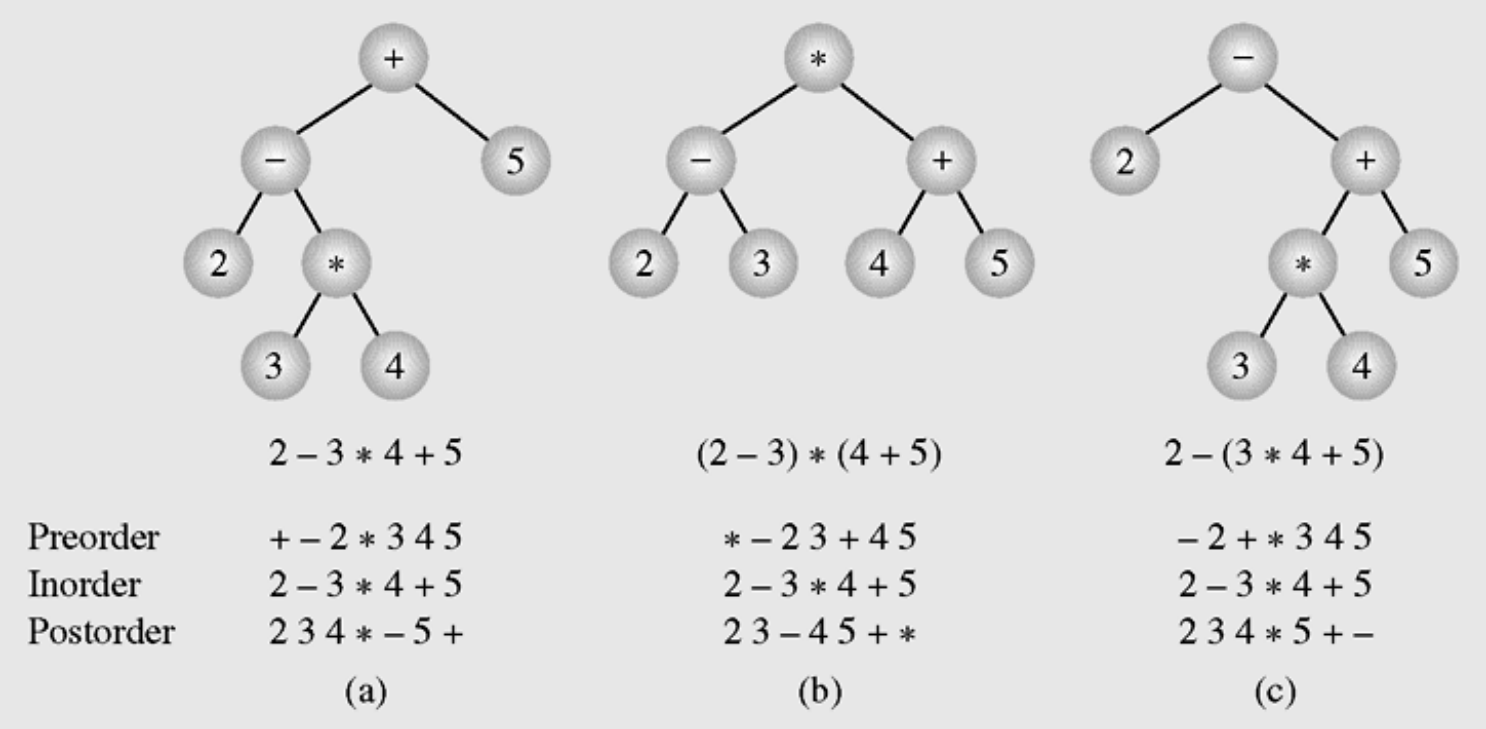


Figure 6-59 Examples of three expression trees and results of their traversals

Operations on Expression Trees

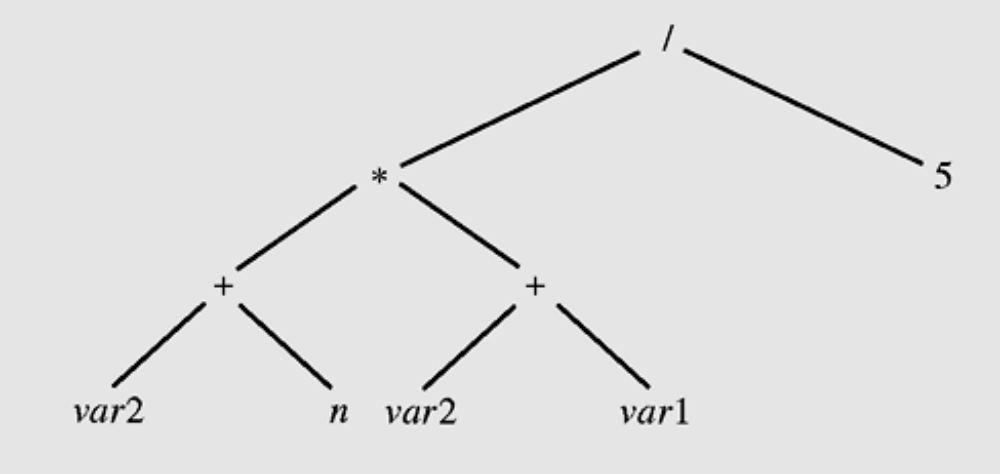


Figure 6-60 An expression tree

Operations on Expression Trees

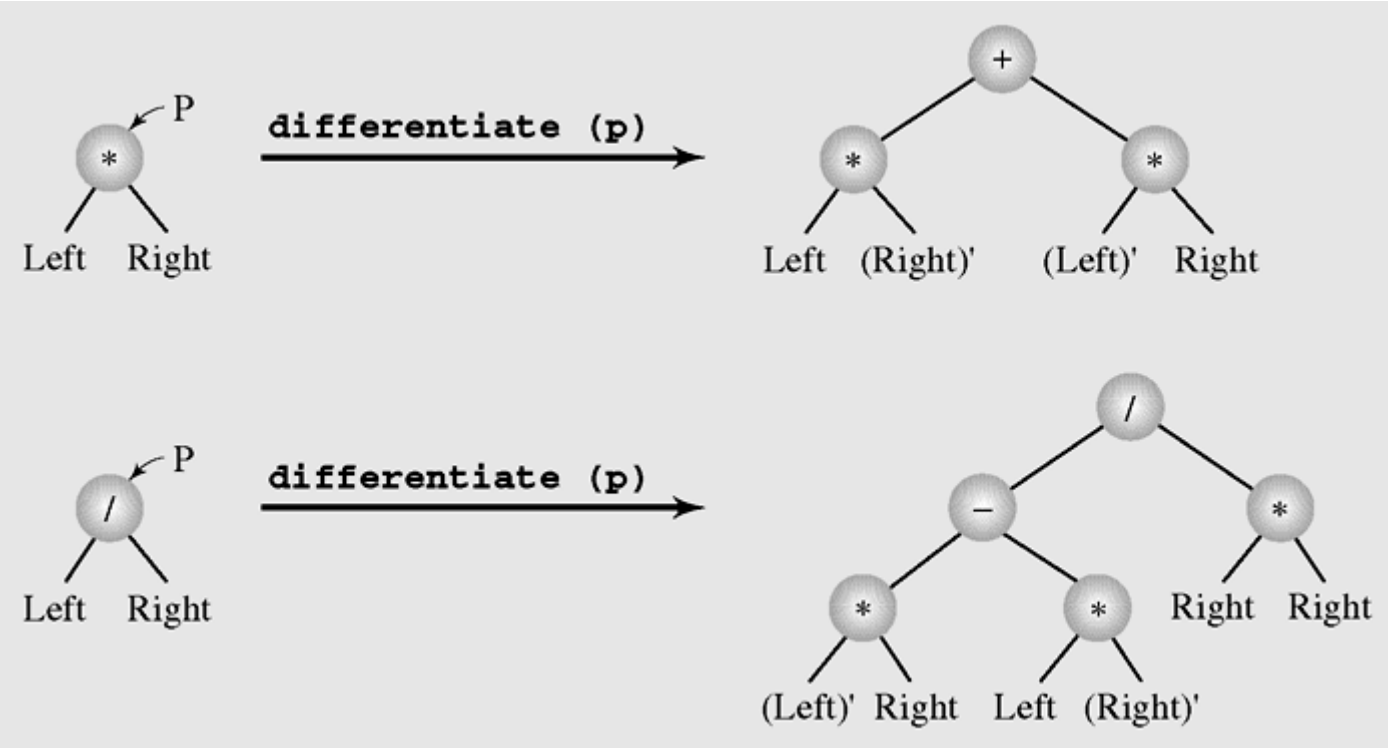


Figure 6-61 Tree transformations for differentiation of multiplication and division

Summary

- A tree is a data type that consists of nodes and arcs.
- The root is a node that has no parent; it can have only child nodes.
- Each node has to be reachable from the root through a unique sequence of arcs, called a path.
- An orderly tree is where all elements are stored according to some predetermined criterion of ordering.

Summary (continued)

- A binary tree is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child.
- A decision tree is a binary tree in which all nodes have either zero or two nonempty children.
- Tree traversal is the process of visiting each node in the tree exactly one time.

Summary (continued)

- An AVL tree is one in which the height of the left and right subtrees of every node differ by at most one.
- Polish notation is a special notation for propositional logic that eliminates all parentheses from formulas.