

# ***Advanced Set Representation Methods***

AVL trees. 2-3(-4) Trees.  
Union-Find Set ADT

## Advanced Set Representation. AVL Trees

---

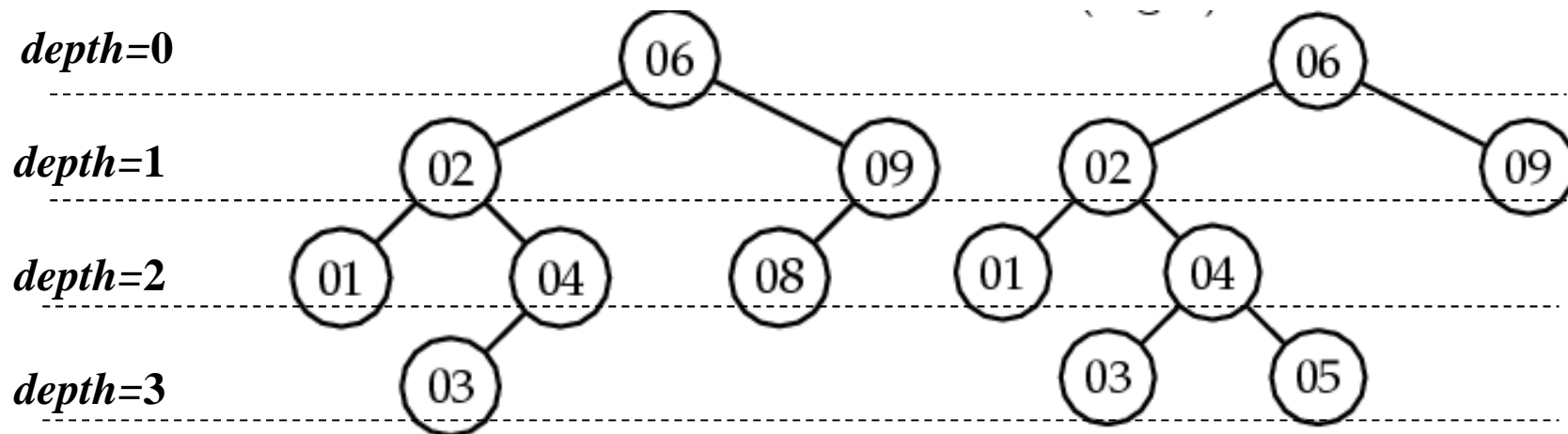
- Problem with BSTs: worst case operation may take  $O(n)$  time. One solution:
- **AVL tree**: binary search tree with a *balance condition*:

For every node in an AVL tree  $T$ , the height of the left ( $T_L$ ) and right ( $T_R$ ) subtrees can differ by at most 1:  $|h_L - h_R| \leq 1$

- Balance condition must be easy to maintain, and it ensures that the tree depth is  $O(\log n)$
- Requiring that the left and right subtrees have the same height does not suffice (tree may not be shallow)

## Two BSTs

- Which BST is AVL?



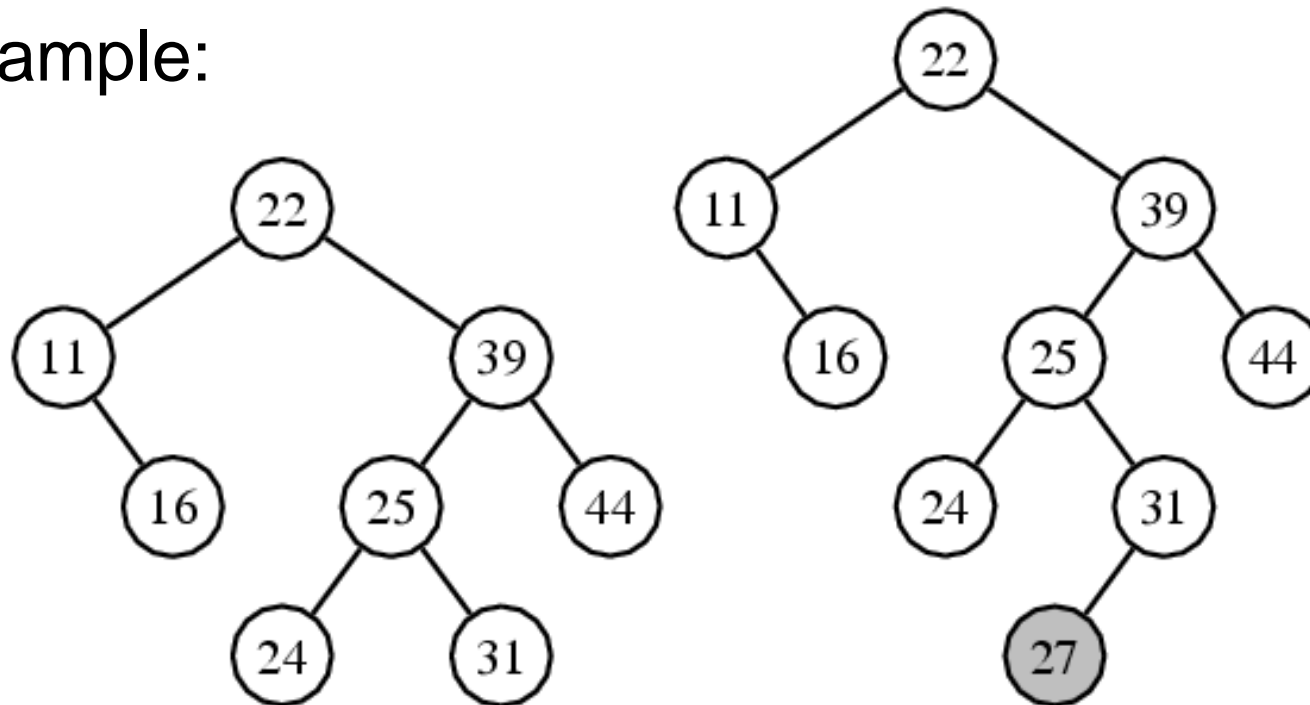
For every node in an AVL tree  $T$ , the height of the left ( $T_L$ ) and right ( $T_R$ ) subtrees can differ by at most 1:  $|h_L - h_R| \leq 1$

# AVL Tree Height

- Claim: height of an AVL tree storing  $n$  keys is  $O(\log n)$
- Proof
  - Let  $n(h)$  denote the number of nodes of an AVL tree of height  $h$ .
  - It is easy to see that  $n(1) = 1$  and  $n(2) = 2$
  - For  $n > 2$ , an AVL tree of height  $h$  contains:
    - the root node,
    - one AVL subtree of height  $h-1$  and
    - another of height  $h-2$ .
  - That is,  $n(h) = 1 + n(h-1) + n(h-2)$
  - Knowing  $n(h-1) > n(h-2)$ , we get  $n(h) > 2n(h-2)$ . By induction  $n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(h-6), \dots, n(h) > 2^i n(h-2i)$
  - Solving the base case we get:  $n(h) > 2^{h/2-1}$
  - Taking logarithms:  $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is  $O(\log n)$

## Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by attaching an external node.
- Example:



- After insertion the tree may need rebalancing

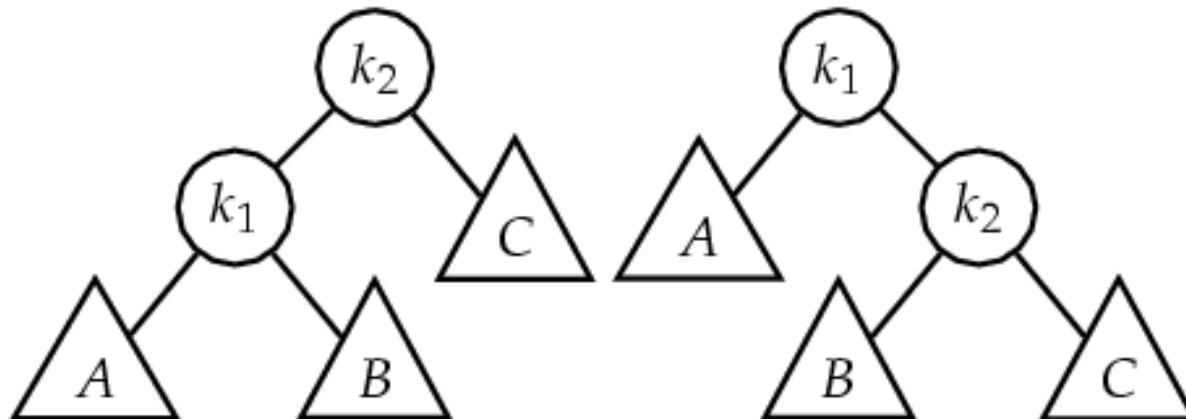
# AVL tree rebalancing

---

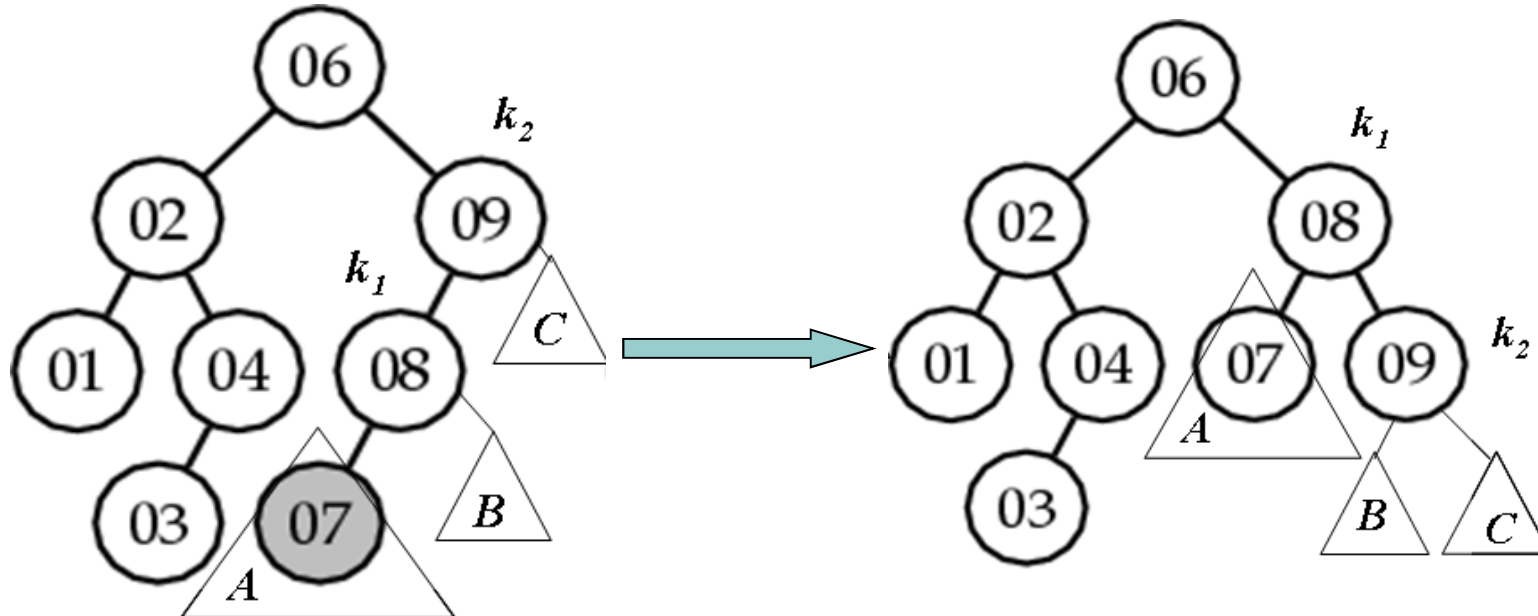
- Two kinds of rotation:
  - single rotation (left or right)
  - double rotation
    - **left-right:** *left rotation* around the *left child* of a node followed by a *right rotation* around the *node itself*
    - **right-left:** *right rotation* around the *left child* of a node followed by a *left rotation* around the *node itself*
- Rotation features:
  - Nodes not in the subtree of the node rotated are unaffected
  - A rotation takes constant time
  - Before and after the rotation tree is still BST
  - Code for left rotation is symmetric to code for a right rotation

# AVL Single Rotations

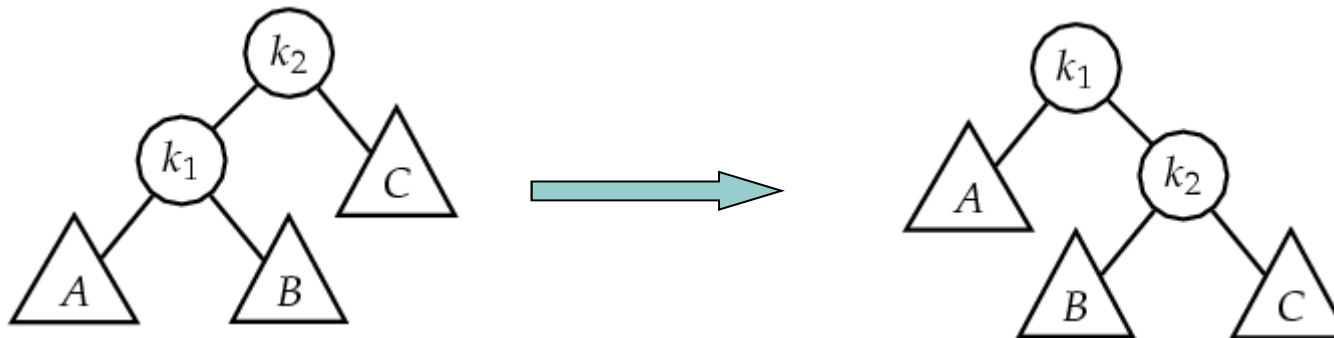
- Single Rotation
  - $k_1 < k_2$
  - all elements in subtree  $A$  are smaller than  $k_1$
  - all elements in subtree  $C$  are larger than  $k_2$
  - all elements in subtree  $B$  are in between  $k_1$  and  $k_2$



## Single Rotation Right Example

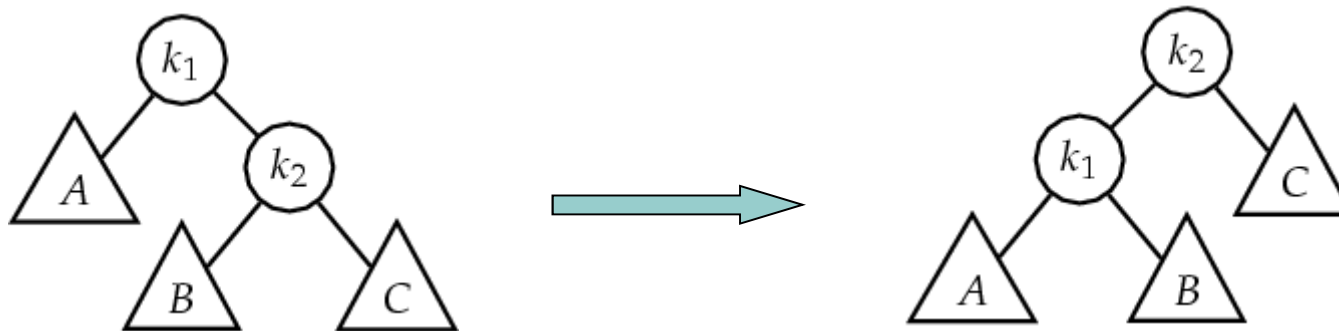
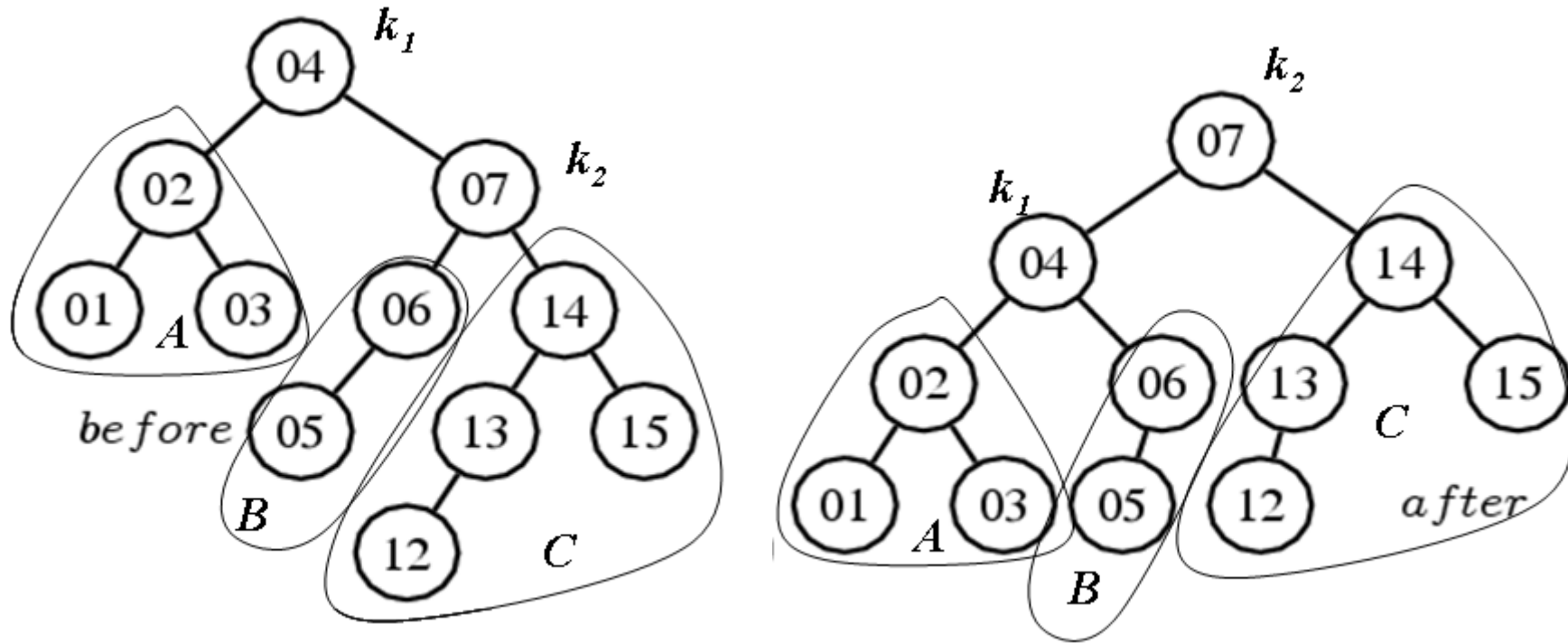


- Note that subtrees B and C are empty





# Single Rotation Left Example



## **AVL Implementation Detail**

---

```
typedef struct
{
    ElementT element;
    AVLPtr left;
    AVLPtr right;
    int height;
} AVLNode;
typedef AVLNode *AVLPtr;
```

# AVL Single Rotations

```
void snglRotRight( AVLPtr *k2 )
```

```
{
```

```
  AVLPtr k1 ;
```

```
  k1 = (*k2)->left ;
```

```
  (*k2)->left = k1->right ;
```

```
  k1->right = *k2 ;
```

```
  (*k2)->height = max( height(
    (*k2)->left ),
    height( (*k2)->right)
```

```
  ) + 1 ;
```

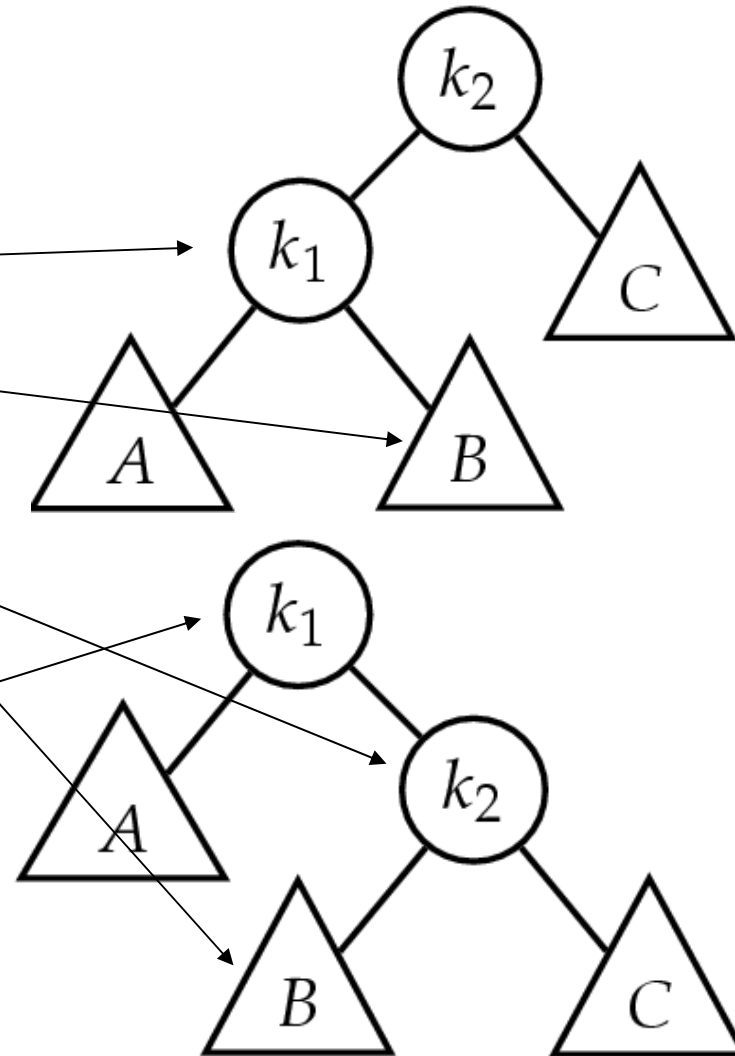
```
  k1->height = max( height( k1->
    left ),
```

```
    (*k2)->height ) + 1 ;
```

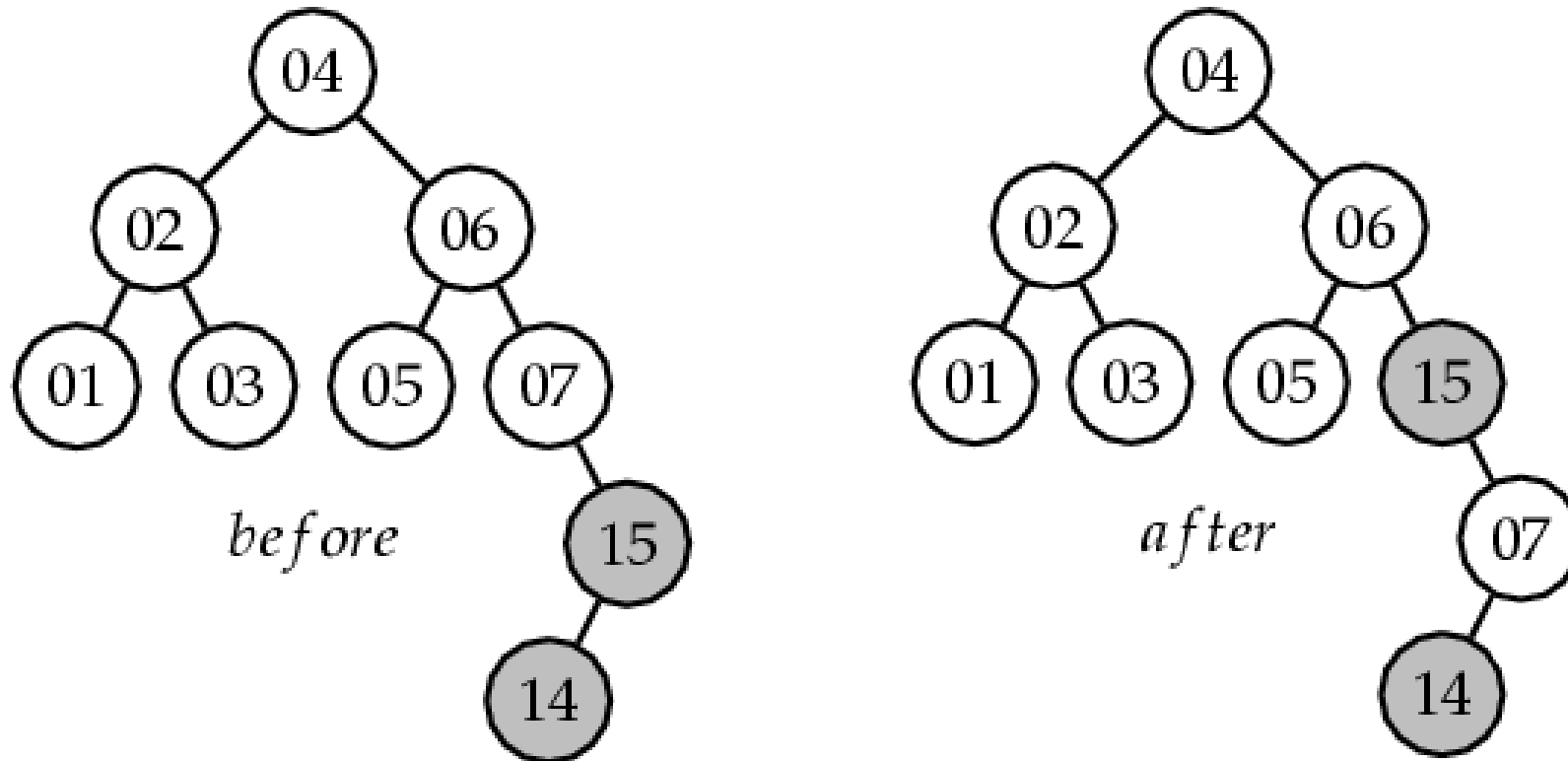
```
  *k2 = k1 ; /* assign new root */
```

```
}
```

```
/* snglRotLeft is symmetric */
```



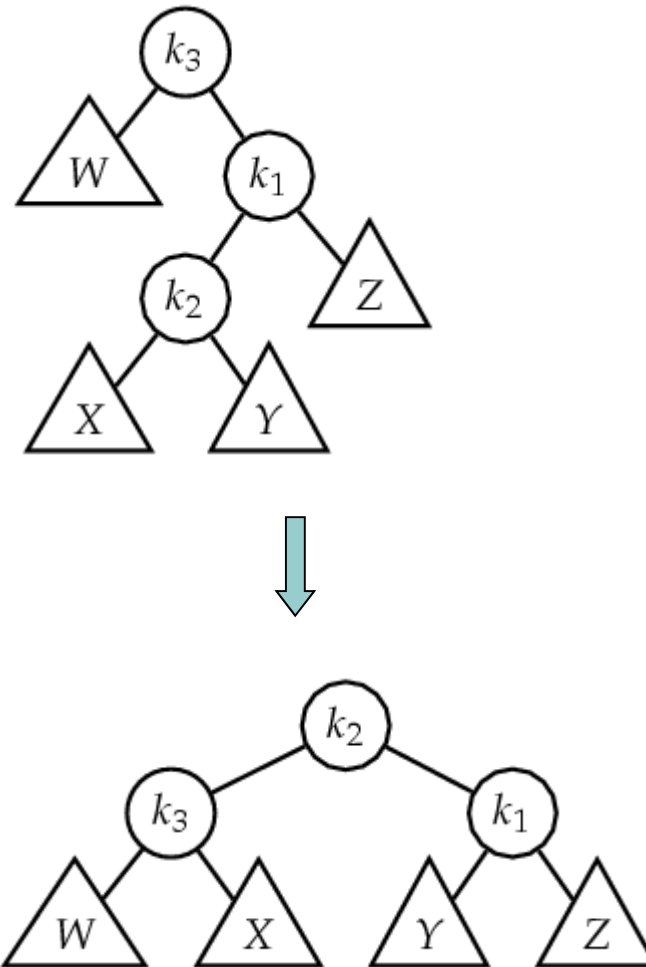
## Single Rotation not enough



# AVL Double Rotation

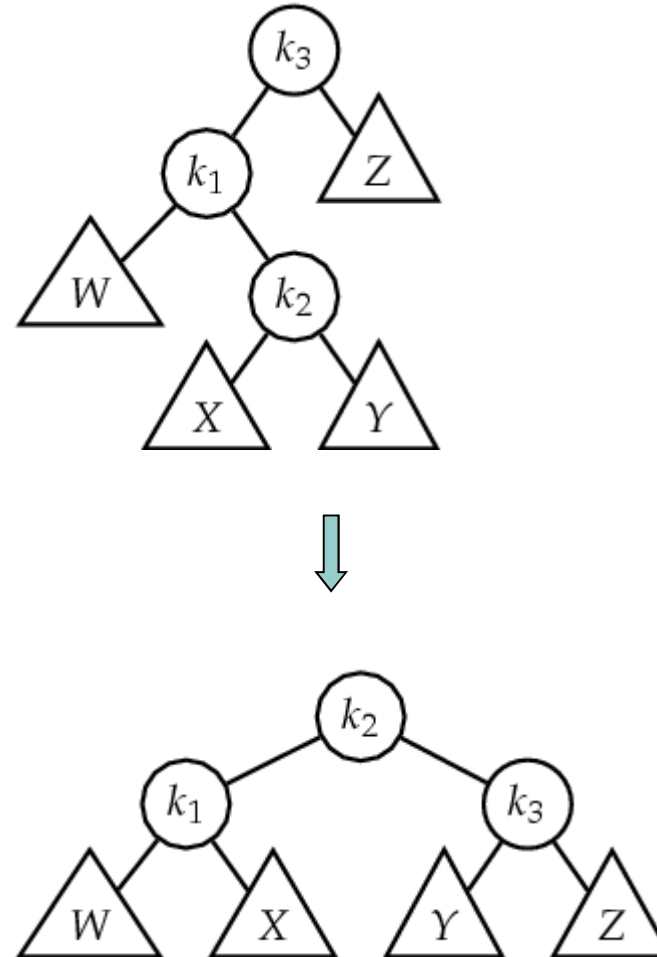
- Right-left
  - $k_2 < k_1, k_3 < k_1, k_3 < k_2$
  - right rotation around the left child of a node followed by
  - a left rotation around the node itself
  - Rotate to make  $k_2$  topmost node

```
void dblRotLeft( AVLPtr k3 )  
{  
    /* rotate between k1 and k2 */  
    snglRotRight ( (*k3)->left );  
    /* rotate between k3 and k2 */  
    snglRotLeft ( k3 );  
}
```



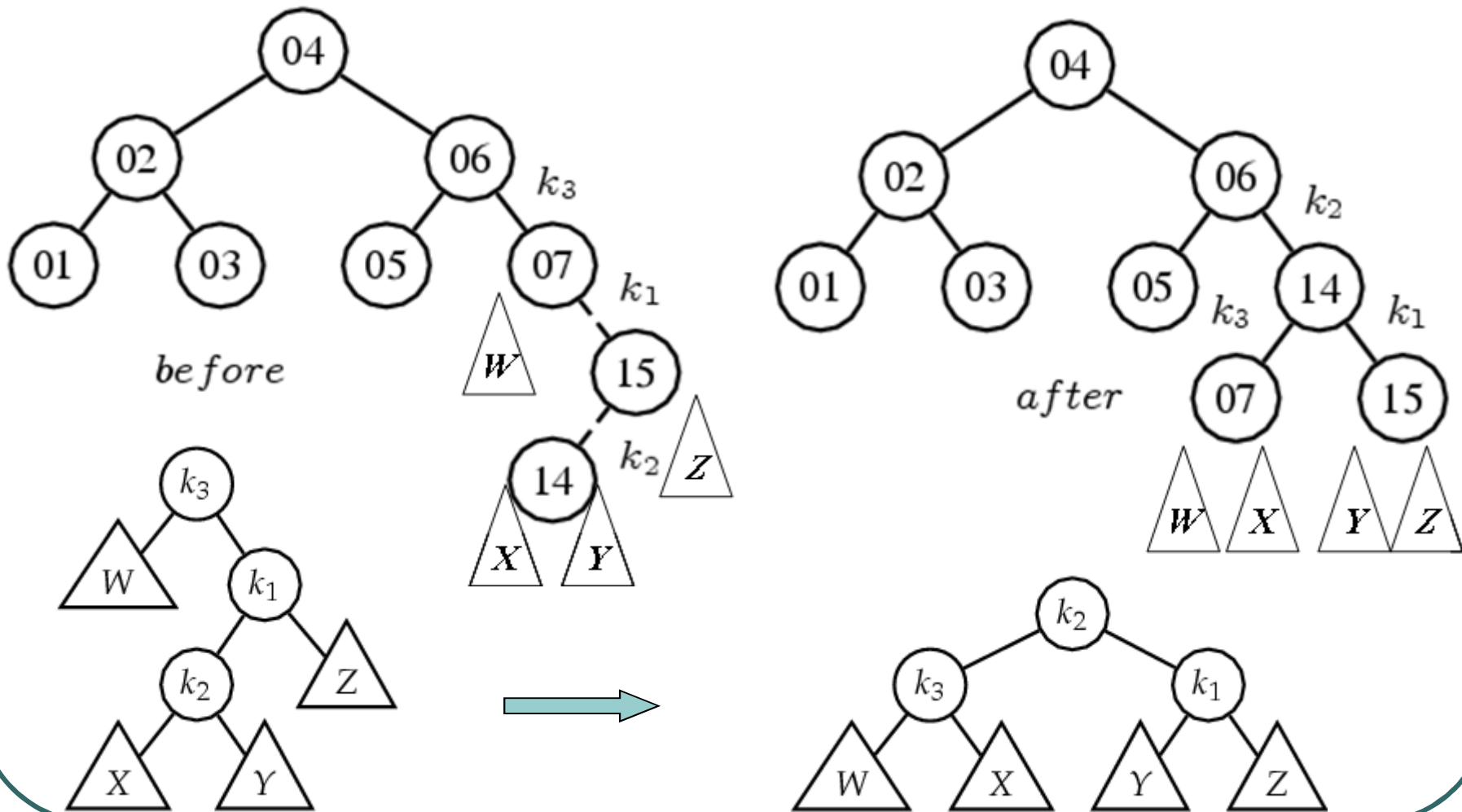
# AVL Double Rotation

- Left-right:
  - $k_1 < k_2, k_1 < k_3, k_2 < k_3$
  - left rotation around the left child of a node followed by a
  - right rotation around the node itself
  - Rotate to make  $k_2$  topmost node



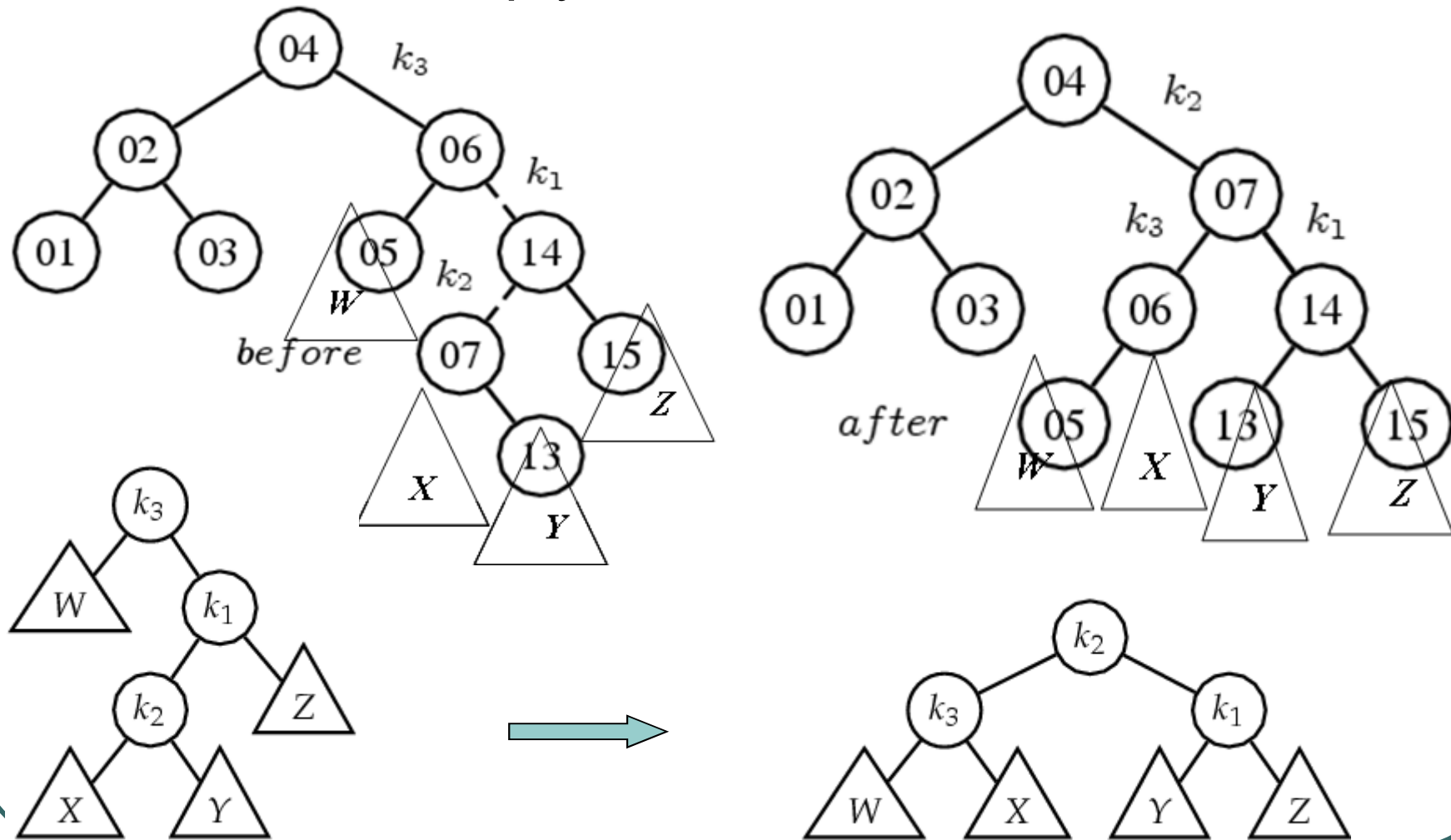
## AVL Right-left Double Rotation Example 1

- Subtrees  $W$ ,  $X$ ,  $Y$ ,  $Z$  are empty



## AVL Right-left Double Rotation Example 2

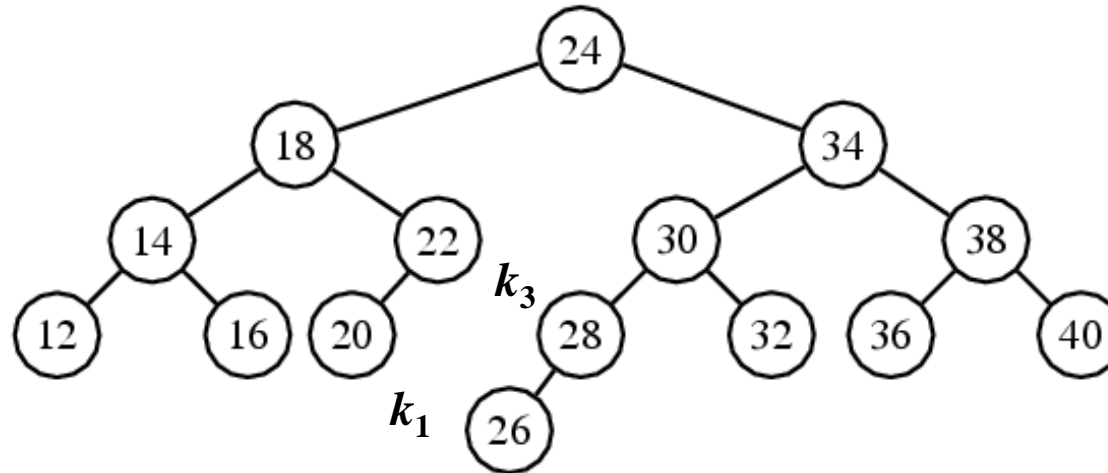
- Subtree  $X$  is empty





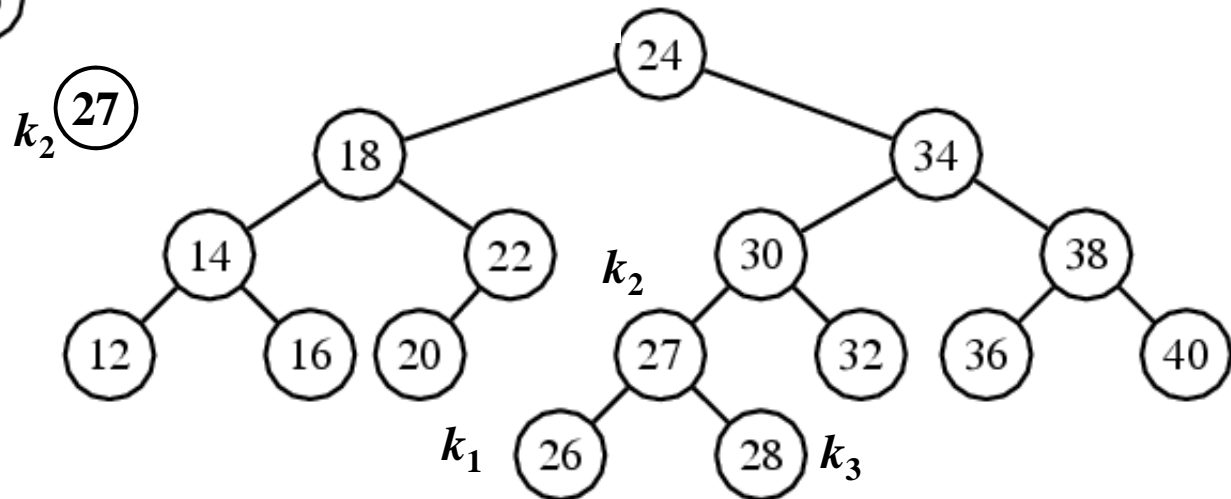
## AVL Double Rotation example

- A symmetric case of double rotation



Tree before  
insertion of 27

Tree after  
insertion of 27  
and re-balancing



# AVL Trees. When to use rotations

---

- Balance factor:  $height(T_L) - height(T_R)$
- Use
  - **Single rotation left:** when a node is inserted in the right subtree of the right child (B) of the nearest ancestor (A) with balance factor -2
  - **Single rotation right:** when a node is inserted in the left subtree of the left child (B) of the nearest ancestor (A) with balance factor +2.
  - **Right-left double rotation:** when a node is inserted in the left subtree of the right child (B) of the nearest ancestor (A) with balance factor -2.
  - **Left-right double rotation:** when a node is inserted in the right subtree of the left child (B) of the nearest ancestor (A) with balance factor +2.

## AVL Trees. Deletions

---

- A node is deleted using the standard inorder successor (predecessor) logic for binary search trees
- Imbalance is fixed using rotations
- Identify the parent of the actual node that was deleted, then:
  - If the left child was deleted, the balance factor at the parent decreased by 1.
  - If the right child was deleted, the balance factor at the parent increased by 1.

## AVL Trees. Deletions. Rebalancing (I)

---

- Let A be the node where balance must be restored.
- If the deleted node was in A's right subtree, then let B be the root of A's left subtree. Then:
  - **B has balance factor 0 or +1 after deletion** -- then perform a single right rotation
  - **B has balance factor -1 after deletion** -- then perform a left-right rotation
- If the deleted node was in A's left subtree, then let B be the root of A's right subtree. Then:
  - **B has balance factor 0 or -1 after deletion** -- then perform a single left rotation
  - **B has balance factor +1 after deletion** -- then perform a right-left rotation

## **AVL Trees. Deletions. Rebalancing (II)**

---

- Unlike insertion, *one rotation may not be enough to restore balance to the tree*. If this is the case, then locate the next node where the balance factor is "bad" (call this A):
- If A's balance factor is positive, then let B be A's left child
  - If B's left subtree height is larger than B's right subtree height, then perform a **single right rotation**.
  - If B's left subtree height is smaller than B's right subtree height, then perform a **left-right rotation**.
  - If B's left subtree height is equal to B's right subtree height, then perform **either rotation**.

## AVL Trees. Deletions. Rebalancing (III)

- If A's balance factor is negative, then let B be A's right child
  - If B's right subtree height is larger than B's left subtree height, then perform a **single left rotation**.
  - If B's right subtree height is smaller than B's left subtree height, then perform a **right-left rotation**.
  - If B's right subtree height is equal to B's left subtree height, then perform **either rotation**.

## AVL trees demo

---

- Demos from:

<http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm>

<http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>

- Replacement for AVL trees: Red-Black trees (not discussed here)

## Running times for AVL tree operations

---

- a single restructure is  $O(1)$ 
  - using a linked-structure binary tree
- find is  $O(\log n)$ 
  - height of tree is  $O(\log n)$ , no restructures needed
- insert is  $O(\log n)$ 
  - initial find is  $O(\log n)$
  - restructuring up the tree, maintaining heights is  $O(\log n)$
- remove is  $O(\log n)$ 
  - initial find is  $O(\log n)$
  - restructuring up the tree, maintaining heights is  $O(\log n)$



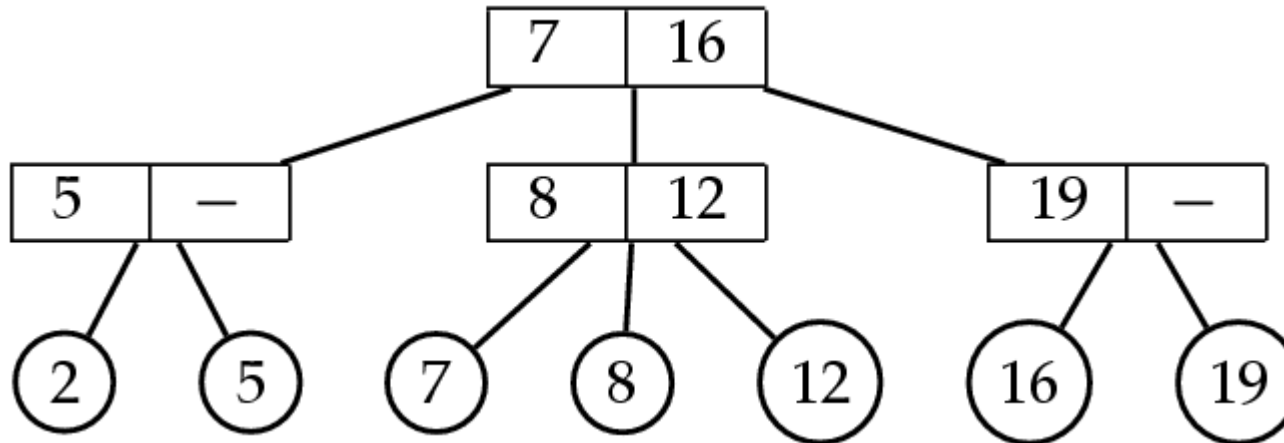
## 2-3 Trees

---

- 2-3 tree properties:
  - Each interior node has two or three children.
  - Each path from the root to a leaf has the same length.
  - A tree with zero or one node(s) is a special case of a 2-3 tree.
- Representing sets with 2-3 trees:
  - Elements are placed at the leaves
  - If element  $a$  is to the left of element  $b$ , then  $a < b$  must hold.
  - Ordering of elements based on one field of a record: a **key**.
  - At each **interior** node: key of the smallest descendant of the second child and, if there is a third child, key of the smallest descendant of third child.

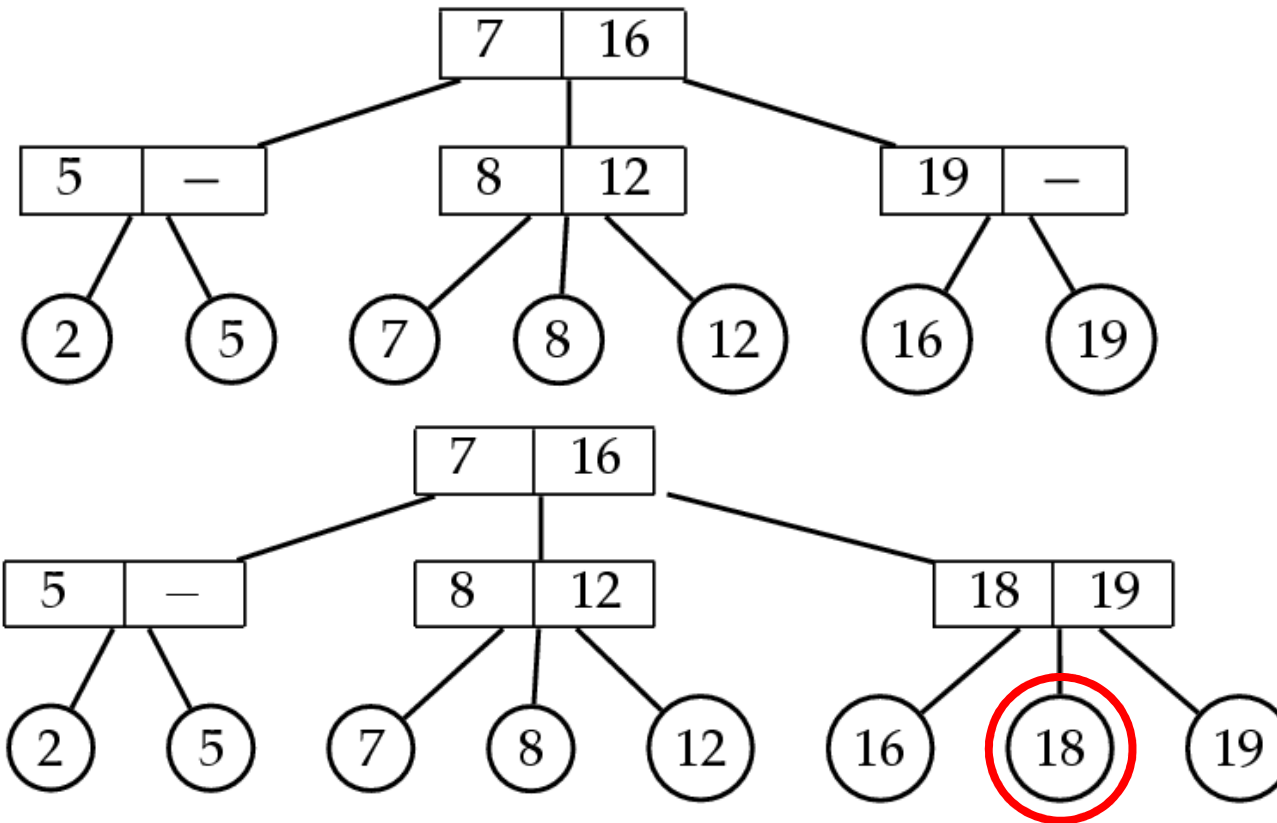
## 2-3 trees

- A 2-3 tree of  $k$  levels has between  $2^{k-1}$  and  $3^{k-1}$  leaves, i.e.
  - representing a set of  $n$  elements requires at least  $1 + \log_3 n$  levels and no more than  $1 + \log_2 n$  levels. Thus, path lengths in the tree are  $O(\log n)$ .



## 2-3 trees. Insertion – 2 children

- If parent node has only two children:
  - insert in order, adjust keys

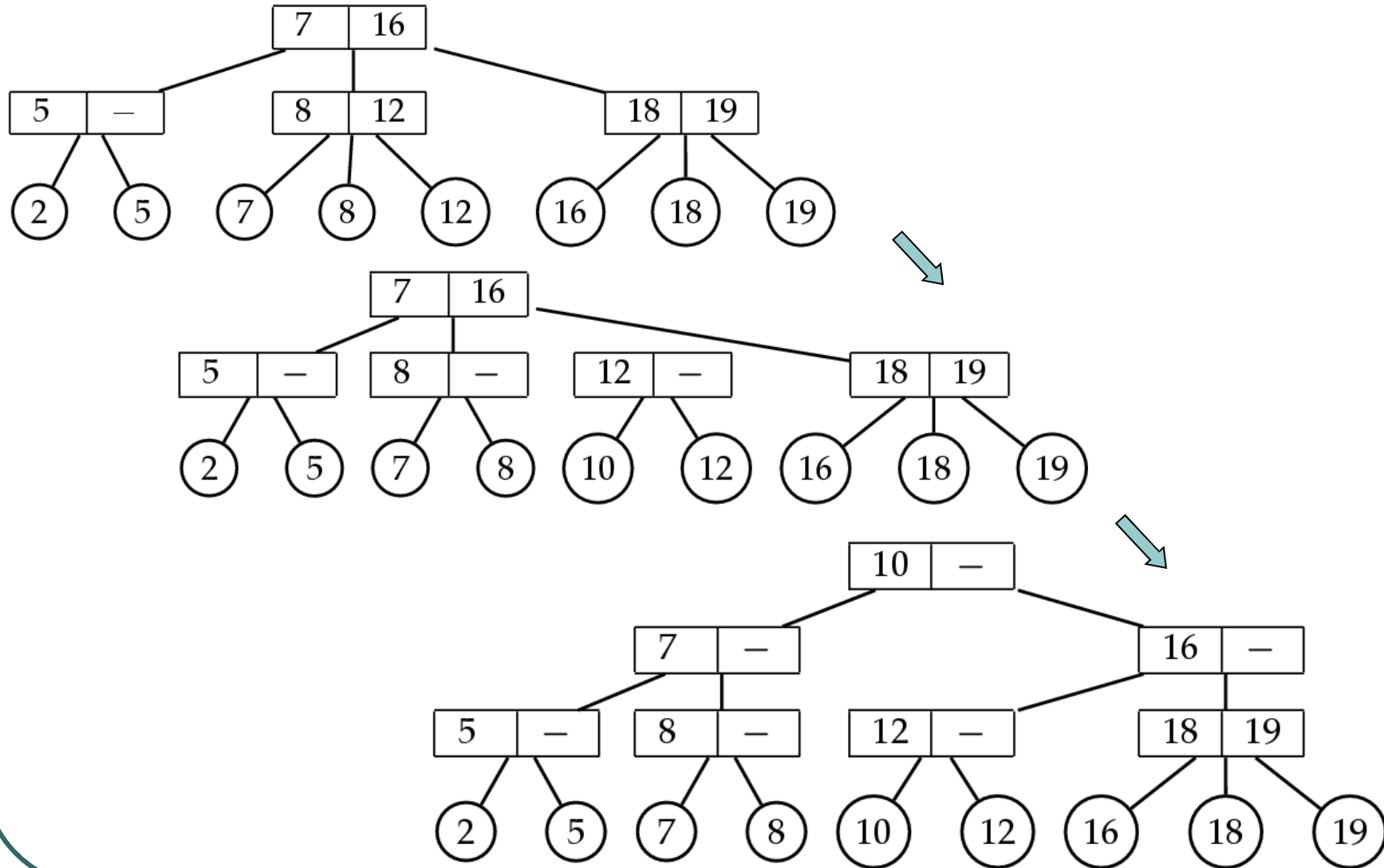


## 2-3 trees. Insertion – 3 children

---

- If parent node *node* already has three children:
  - We cannot have four nodes, and thus we have to split parent in two nodes *node* and *node'*.
  - The two smallest elements among the four children of *node* stay with *node*,
  - The two larger will become the children of *node'*
  - Continue process up the tree
  - Special case when splitting the root

## 2-3 trees. Insertion

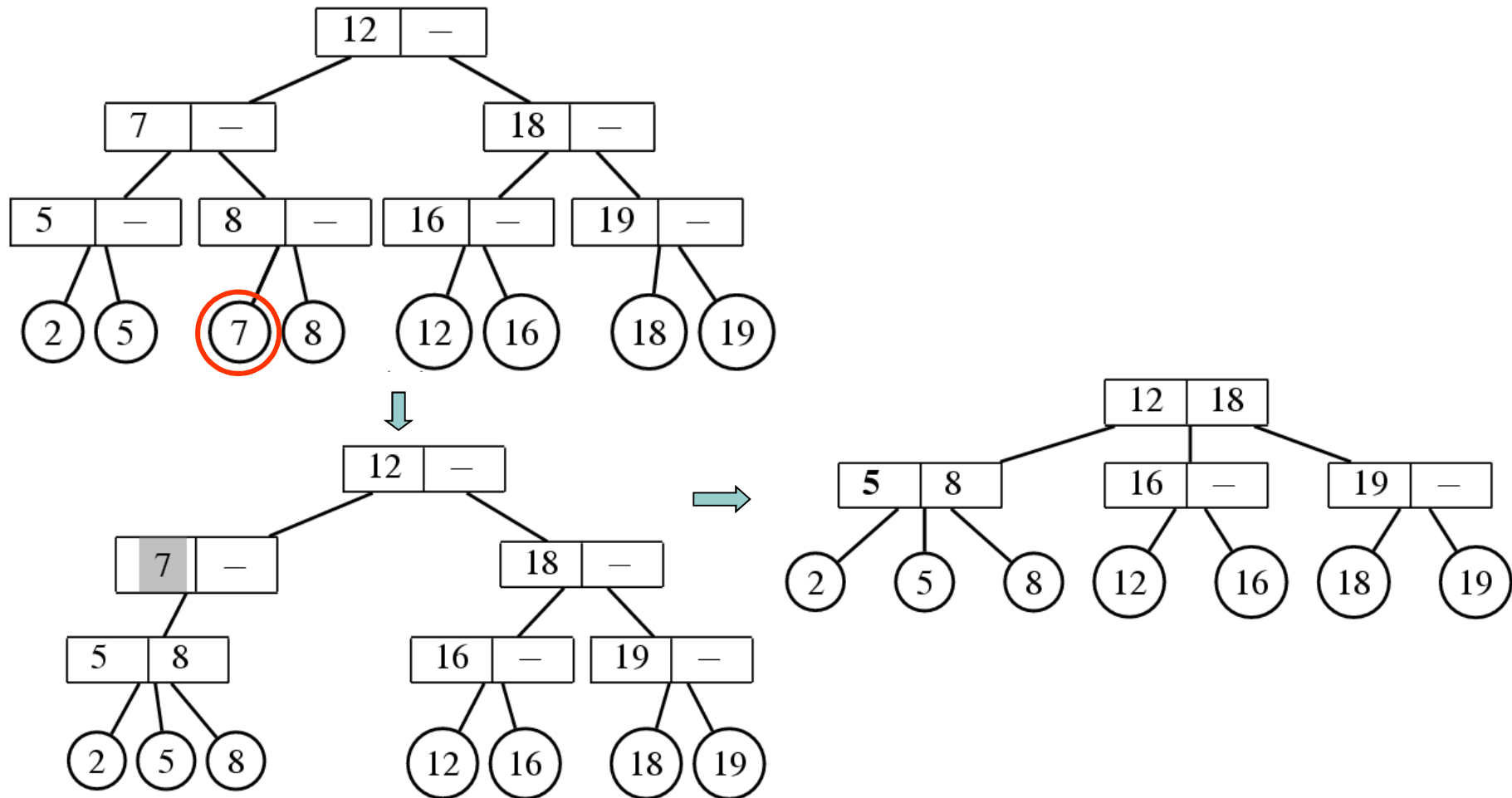


## 2-3 trees. Deletion

---

- Leaf deletion could leave parent with only 1 child
  - Parent=root: delete *node* and let its lone child be the new root
  - Otherwise, let  $p$  be the parent of *node*
    - if  $p$  has another child, and that child of  $p$  has three children, then we can transfer the proper one to *node*; done;
    - if the children of  $p$ , have only two children, transfer the lone child of *node* to an adjacent sibling of *node*, and delete *node*;
    - $p$  has only one child, repeat all the above, recursively, with  $p$  in place of *node*.

## 2-3 trees. Deletion



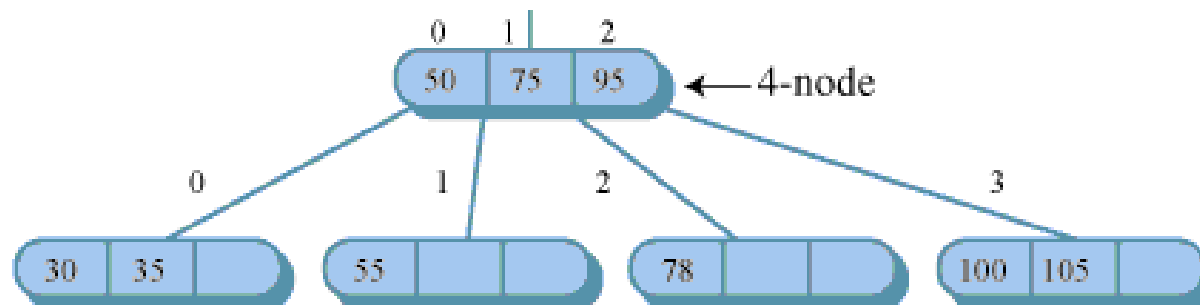
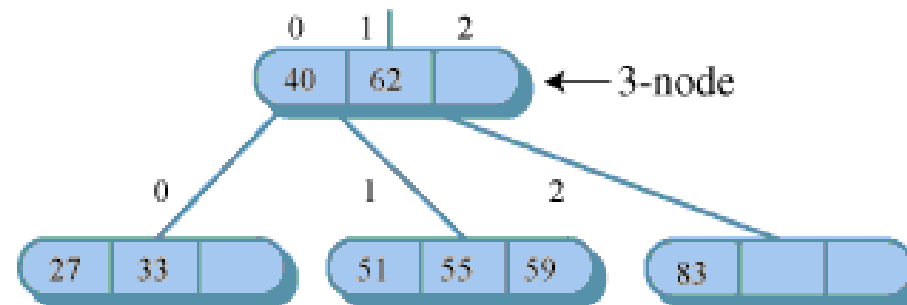
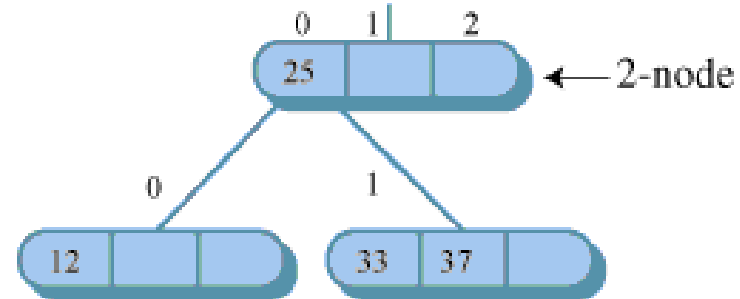
## 2-3-4 trees.

---

- 2-3-4 tree refer to how many links to child nodes can potentially be contained in a given node. For non-leaf nodes, three arrangements :
  - A node with one data item always has two children
  - A node with two data items always has three children
  - A node with three data items always has four children
  - In short, a non-leaf node must always have one more child than it has data items.
- Symbolically, if the number of child links is  $L$  and the number of data items is  $D$ , then  $L = D + 1$
- Empty nodes are not allowed.

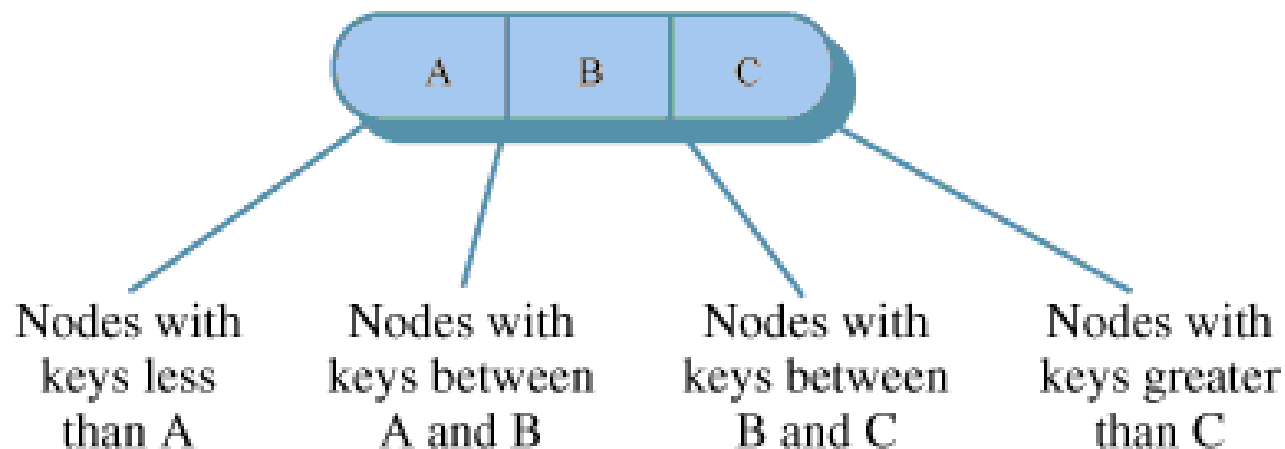


## 2-3-4 trees. Node kinds.



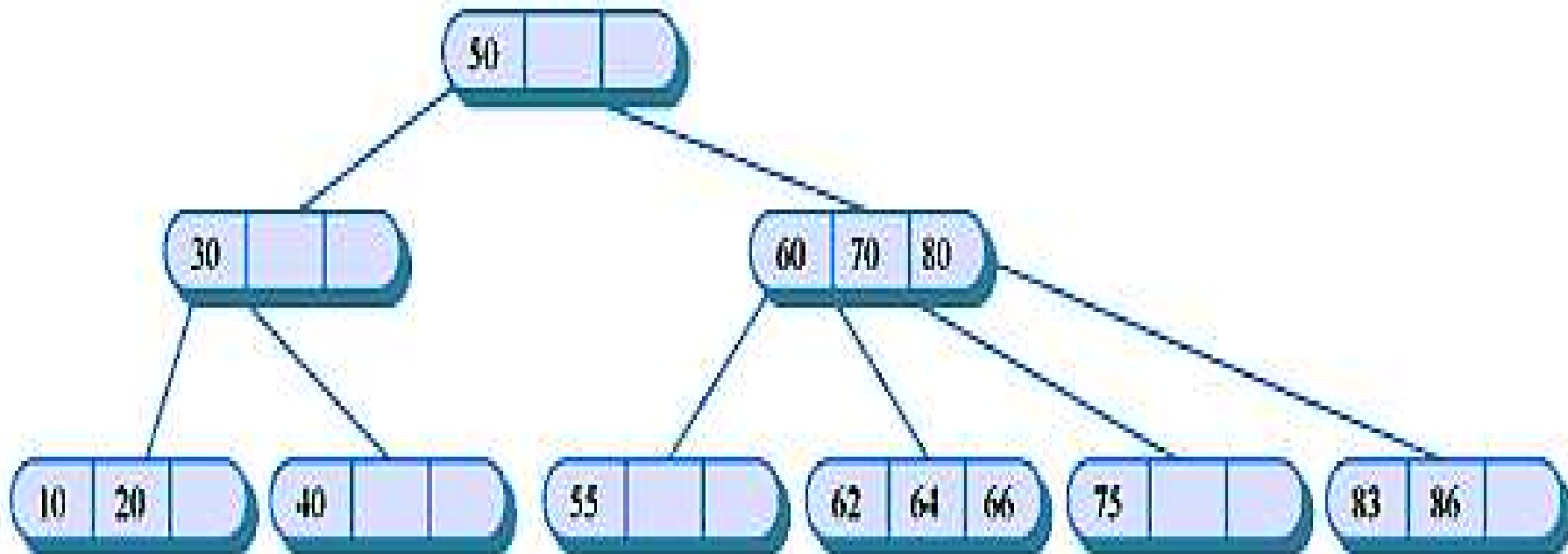
## 2-3-4 trees. Properties and example.

- All children in the subtree rooted at child 0 have key values less than key 0.
- All children in the subtree rooted at child 1 have key values greater than key 0 but less than key 1.
- All children in the subtree rooted at child 2 have key values greater than key 1 but less than key 2.
- All children in the subtree rooted at child 3 have key values greater than key 2.



## 2-3-4 trees. Example.

---



## 2-3-4 Trees. Insertion

---

- **Insertion procedure:**
  - similar to insertion in 2-3 trees
  - items are inserted at the leafs
  - since a 4-node cannot take another item, 4-nodes are split up during insertion process
- **Strategy**
  - on the way from the root down to the leaf: split up all 4-nodes "on the way"
  - → insertion can be done in one pass (remember: in 2-3 trees, a reverse pass might be necessary)

## 2-3-4 Trees. Deletion

---

- **Deletion procedure:**
  - similar to deletion in 2-3 trees
  - items are deleted at the leafs
    - swap item of internal node with inorder successor
  - note: a 2-node leaf creates a problem
  
- **Strategy** (different strategies possible)
  - on the way from the root down to the leaf:  
turn 2-nodes (except root) into 3-nodes
  - → deletion can be done in one pass  
(remember: in 2-3 trees, a reverse pass might be necessary)

## Demo: 2-3-4 trees

---

- Demo from:  
<http://www.cs.unm.edu/~rlpm/499/tfft.html>
- Local:  
[Demos\2-3-4trees\TTFT.jar](#)

## Disjoint Sets with the UNION and FIND Operations

---

- Applicable to problems where:
  - start with a collection of objects, each in a set by itself;
  - combine sets in some order, and from time to time
  - ask which set a particular object is in
- Equivalence classes:
  - If set  $S$  has an equivalence relation (reflexive, symmetrical, transitive) defined on it, then the set  $S$  can be partitioned into disjoint subsets  $S_1, S_2, \dots, S_k$  with  $\bigcup_k S_k = S$
- Equivalence problem:
  - given a set  $S$  and a sequence of statements of the form  $a \equiv b$
  - process the statements in order in such a way that at any time we are able to determine in which equivalence class a given element belongs

## Union-find set ADT

- Example statements (see previous slide)

$$S = \{1, 2, \dots, 7\}$$

$$1 \equiv 2, 5 \equiv 6, 3 \equiv 4, 1 \equiv 4$$

$$1 \equiv 2 \quad \{1, 2\} \{3\} \{4\} \{5\} \{6\} \{7\}$$

$$5 \equiv 6 \quad \{1, 2\} \{3\} \{4\} \{5, 6\} \{7\}$$

$$3 \equiv 4 \quad \{1, 2\} \{3, 4\} \{5, 6\} \{7\}$$

$$1 \equiv 4 \quad \{1, 2, 3, 4\} \{5, 6\} \{7\}$$

- Operations:
  - union(A, B) takes the union of the components A and B and calls the result either A or B, arbitrarily.
  - find(x), a function that returns the name for the component of which x is a member.
  - initial(A, x) creates a component named A that contains only the element x.

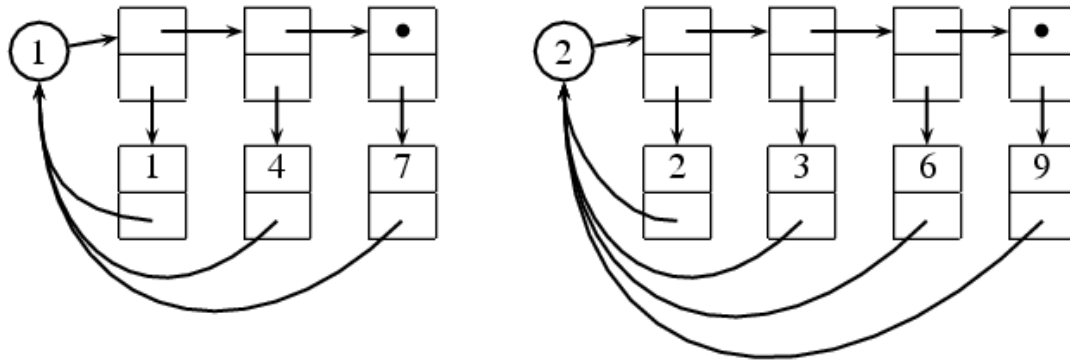


## Union-find set ADT. Implementations

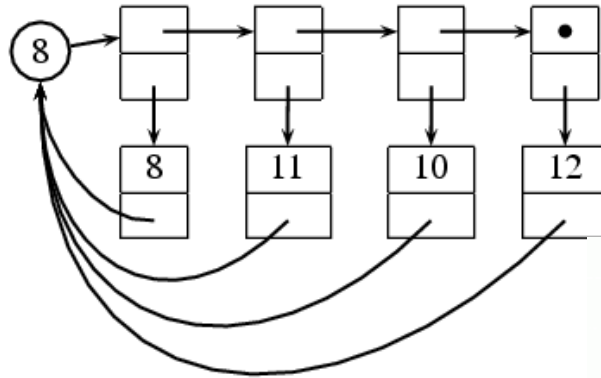
---

- List-based
  - Each set is stored in a sequence represented with a linked-list
  - Each node should store an object containing the element and a reference to the set name
- Tree based
  - Each element is stored in a node, which contains a pointer to a set name
  - A node  $v$  whose set pointer points back to  $v$  is also a set name
  - Each set is a tree, rooted at a node with a self-referencing set pointer

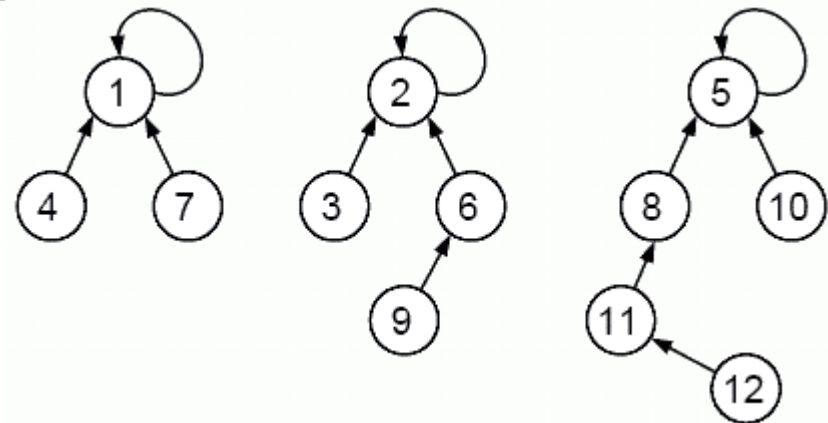
# Examples



List based



Tree based



## Union-find set. List based implementation

```

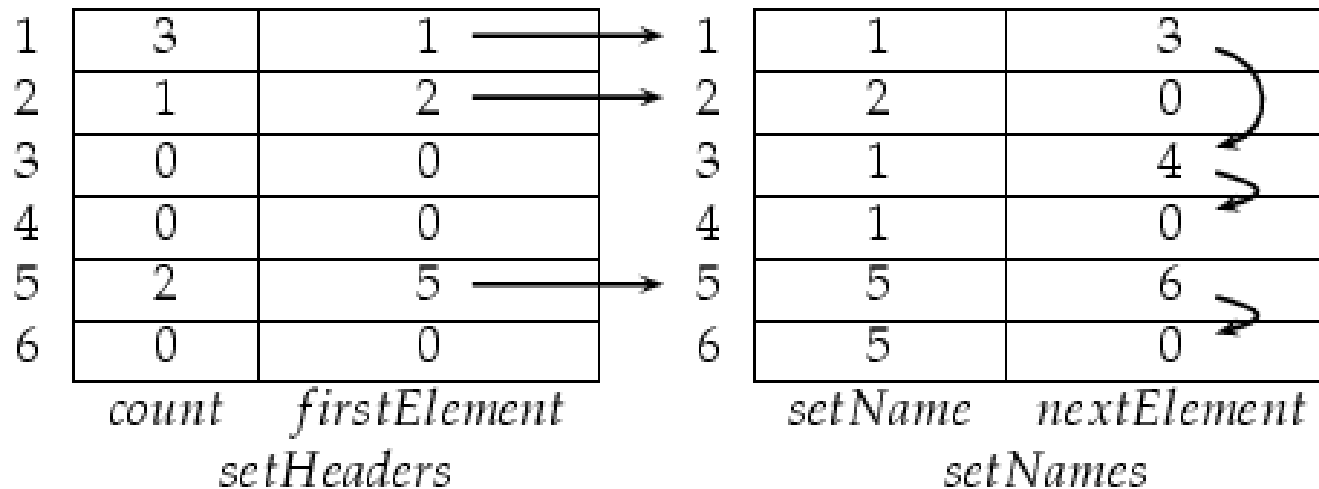
const int n = /* appropriate value */
typedef int NameT;
typedef int ElementT;
typedef struct ufset
{
    struct /* headers for set lists */
    {
        int count;
        int firstElement ;
    } setHeaders[ n ];

```

```

    struct /* table giving set containing
           each member */
    {
        NameT setName ;
        int nextElement ;
    } setName[ n ] ;
} UFSetT;

```



## Union-find sets. Tree representation

---

- Maintain  $S$  as a collection of trees (a forest), one per partition
- Initially, there are  $n$  trees, each containing a single element
- $\text{find}(i)$  returns the root of the tree containing  $i$
- $\text{union}(i, j)$  merges the trees containing  $i$  and  $j$
- Typical traversals not needed, so no need for pointers to children, but we need pointers to parent
- Parent pointers can be stored in an array:  $\text{parent}[i]$  ( $=-1$  for root)

# Union-find sets. Tree implementation

---

INIT()

```
1  for  $i \leftarrow 1$  to  $n$ 
2      do  $parent[i] \leftarrow 0$ 
```

FIND( $i$ )

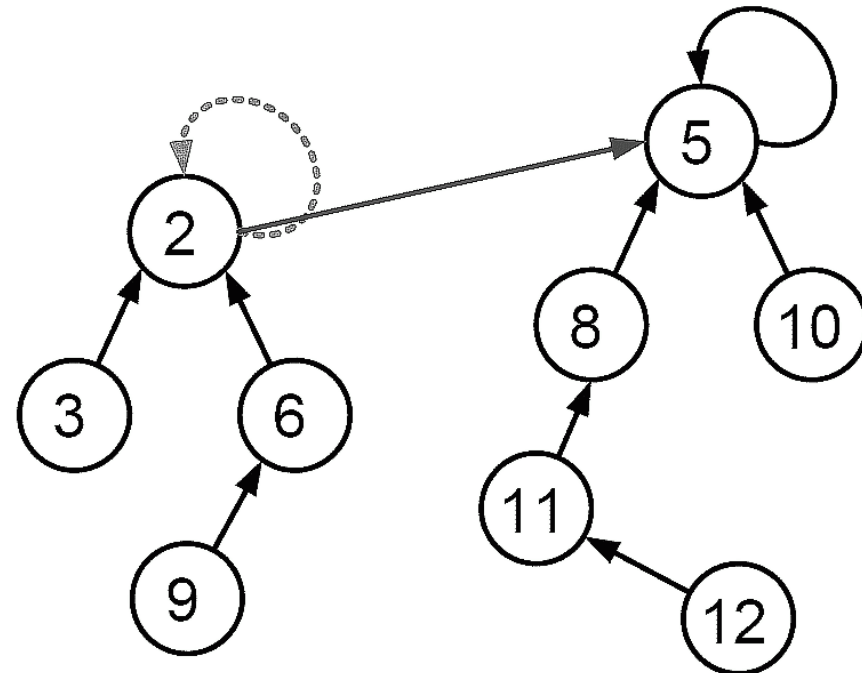
```
1   $j \leftarrow i$ 
2  while  $parent[j] > 0$ 
3      do  $j \leftarrow parent[j]$ 
4  return  $j$ 
```

UNION( $i, j$ )

```
1   $root_1 \leftarrow FIND(i)$ 
2   $root_2 \leftarrow FIND(j)$ 
3  if  $root_1 \neq root_2$ 
4      then  $parent[root_2] \leftarrow root_1$ 
```

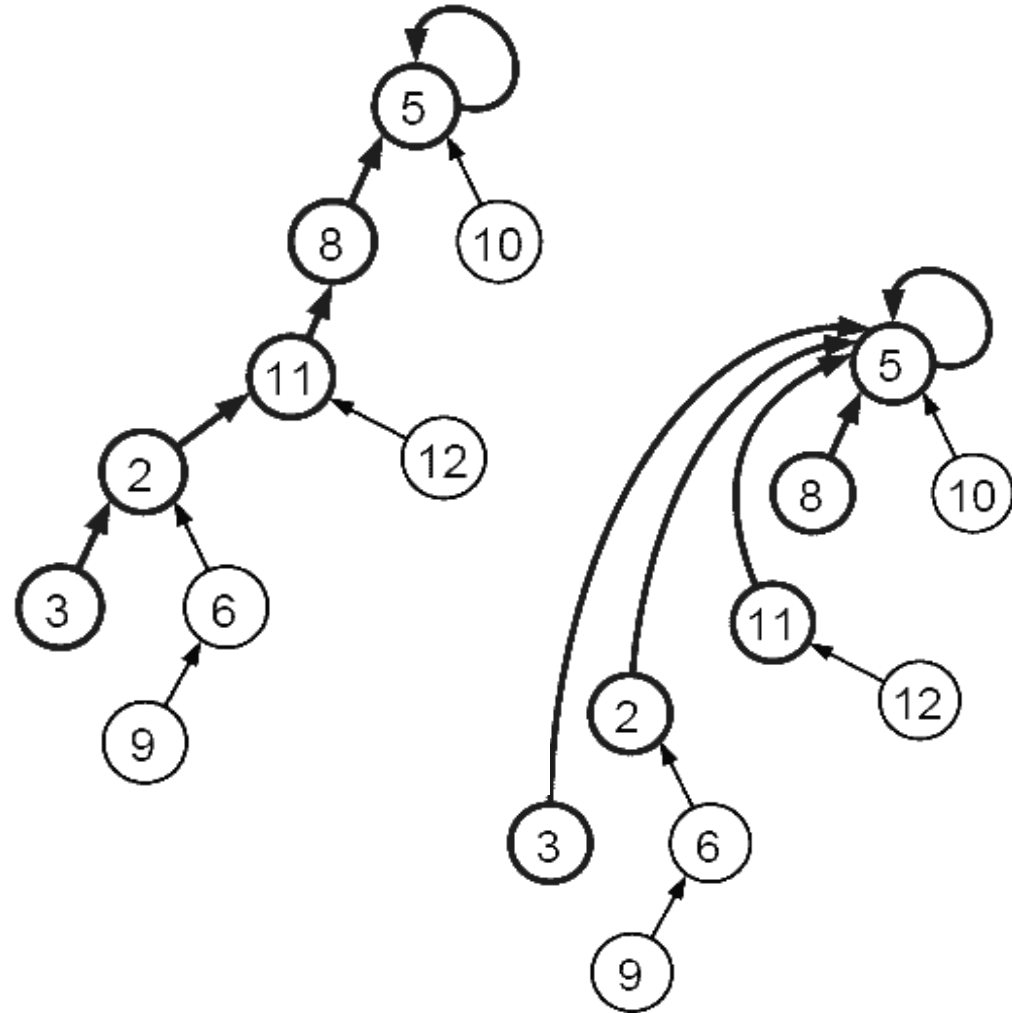
## Union-find sets. Speeding up

- Union by size (rank):
  - When performing a union, make the root of smaller tree point to the root of the larger
- Implies  $O(n \log n)$  time for performing  $n$  unionfind operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most  $O(\log n)$  pointers for any find.



## Union-find sets. Speeding up

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root
- Implies  $O(n \log^* n)$  time for performing  $n$  union- find



## Using union by rank and path compression

MAKESET( $x$ )

```
1  $p[x] \leftarrow x$   
2  $r[x] \leftarrow 0$ 
```

FINDSET( $x$ )

```
1 if  $x \neq p[x]$   
2   then  $p[x] \leftarrow \text{FINDSET}(p[x])$   
3 return  $p[x]$ 
```

LINK( $x, y$ )

```
1 if  $r[x] > r[y]$   
2   then  $p[y] \leftarrow x$   
3   else  $p[x] \leftarrow y$   
4       if  $r[x] = r[y]$   
5         then  $r[y] \leftarrow r[y] + 1$ 
```

UNION( $x, y$ )

```
1 LINK(FINDSET( $x$ ), FINDSET( $y$ ))
```

- Time bound of  $O(m\alpha(n))$  where  $\alpha(n)$  is a very slowly growing function



## **Union-find ADT demo**

---

- Demo from:

<http://www.cs.unm.edu/~rlpm/499/uf.html>

- Local:

[Demos\UnionFind\UnionFind.jar](#)

## A very quickly growing function and its inverse

- For integers  $k \geq 0$  and  $j \geq 1$ , define  $A_k(j)$ :

$$A_k(j) = \begin{cases} j+1, & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j), & \text{if } k \geq 1 \end{cases}$$

- where  $A_{k-1}^0(j) = j$ ,  $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$  for  $i \geq 1$ .
- $k$  is called the level of the function and
- $i$  in the above is called iterations.
- $A_k(j)$  strictly increase with both  $j$  and  $k$ .
- Let us see how quick the increase is

## A very quickly growing function and its inverse

---

- Let us see  $A_k(1)$ : for  $k=0,1,2,3,4$ .
  - $A_0(1)=1+1=2$
  - $A_1(1)=2\times 1+1=3$
  - $A_2(1)=2^{1+1}\times(1+1)-1=7$
  - $A_3(1)=A_2^{(1+1)}(1)=A_2^{(2)}(1)=A_2(A_2(1))=A_2(7)=2^{7+1}(7+1)-1=2^8\times 8-1=2047$
  - $A_4(1)=A_3^2(1)=A_3(A_3(1))=A_3(2047)=A_2^{(2048)}(2047)$
  - $\gg A_2(2047)=2^{2048}\times 2048-1 > 2^{2048}=(2^4)^{512}=(16)^{512}$
  - $\gg \mathbf{10^{80}}$ . (estimated number of atoms in universe)

## **Inverse of $A_k(n):\alpha(n)$**

---

- $\alpha(n) = \min\{k: A_k(1) \geq n\}$
- $\alpha(n) = 0$  for  $0 \leq n \leq 2$
- $1$  for  $n = 3$
- $2$  for  $4 \leq n \leq 7$
- $3$  for  $8 \leq n \leq 2047$
- $4$  for  $2048 \leq n \leq A_4(1)$ .
- Extremely slow increasing function.
- $\alpha(n) \leq 4$  for all practical purpose.

## Reading

---

- AHU, chapter 5, sections 5.4, 5.5
- Preiss, chapter: Search Trees – section AVL Search Trees
- CLR, chapter 22, sections 1-4
- CLRS chapter 21
- Notes