Transform-and-Conquer

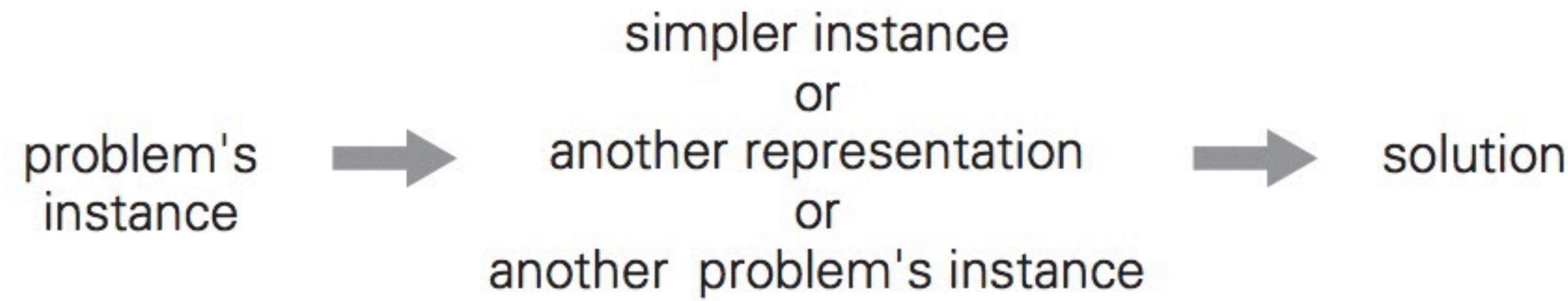
Natasha Dejdumrong

CPE 212 Algorithms Design

Topics

- Presorting
- Heap and Heapsort
- Balanced search tree

General Concept



Presorting

- Element uniqueness
- Computing a mode

Element Uniqueness

Algorithm UniqueElement

Input: A sequence of numbers $a_1, a_2, \cdots, a_n$

Output: Return "true" if all elements are distinct and "false" otherwise.

for $i = 1$ to $n - 1$ **do**

  for $j = i + 1$ to n **do**

    if $a_i = a_j$ **then**

      Return *false*

Return *true*

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).
\end{aligned}
$$

- Brute-force uniqueness algorithm is $\Theta(n^2)$

- Running time is the sum of two components
  - Sorting time is $\Theta(n \log n)$
  - Comparison is $\Theta(n)$

3 13 1 6 5 8 13 9 8

1 3 5 6 8 8 9 13 15

Algorithm PresortUniqueElement

```
Input: A sequence of numbers a_1, a_2, ..., a_n
Output: Return "true" if all elements are distinct and "false" otherwise.

Sort the array A
for i = 1 to n - 1 do
    if a_i = a_{i+1} then
        Return false
Return true
```

Computing a Mode

- Brute-force approach
  - Store the values already encountered along with their frequencies in an auxiliary list
  - On each iteration, scan the auxiliary list for the match and increase the frequency or add the new entry

6 3 2 9 1 3 6 6 2 3 1 3 2 6 9 3 2 1 3

Iteration	# counts for each value				
	1	2	3	6	9
1				1	
2			1	1	
3		1	1	1	
4		1	1	1	1
5	1	1	1	1	1
6	1	1	2	1	1
7	1	1	2	2	1
8	1	1	2	3	1
9	2	1	2	3	1
...	..	..	..	..	..
19	3	4	6	4	2

Worst-case C(n) when no equal elements

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Presort mode

1 1 1 2 2 2 2 3 3 3 3 3 3 6 6 6 6 9 9

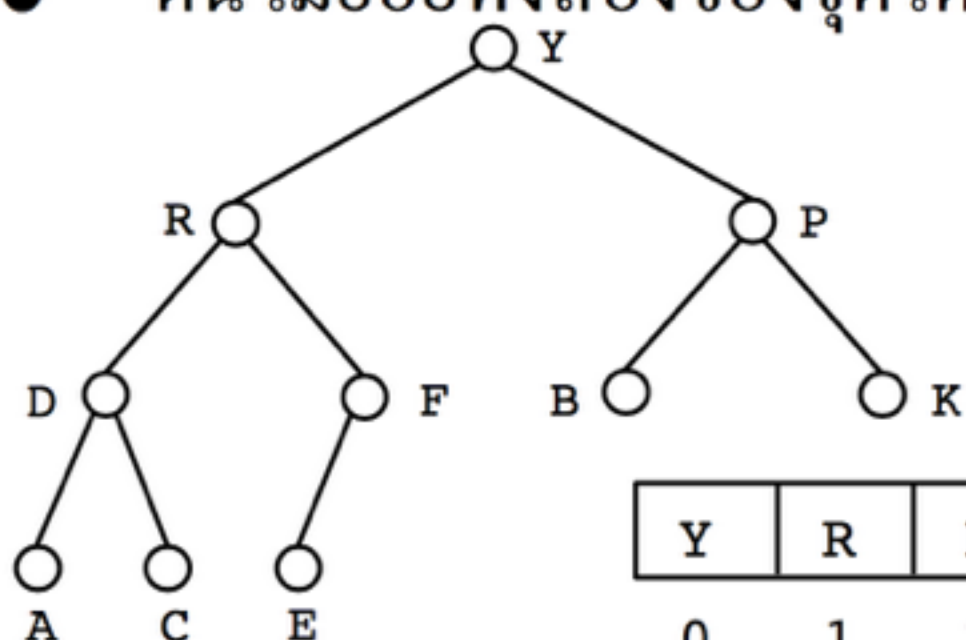
ALGORITHM $PresortMode(A[0..n-1])$

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \le n-1$ **do**
    $runlength \leftarrow 1; \quad runvalue \leftarrow A[i]$
    while $i + runlength \le n-1$ **and** $A[i + runlength] = runvalue$
        $runlength \leftarrow runlength + 1$
    if $runlength > modefrequency$
        $modefrequency \leftarrow runlength; \quad modevalue \leftarrow runvalue$
    $i \leftarrow i + runlength$
return $modevalue$

Heap

Heap คือต้นไม้แบบทวิภาคที่มีคีย์ของข้อมูลกำกับจุด โดยที่

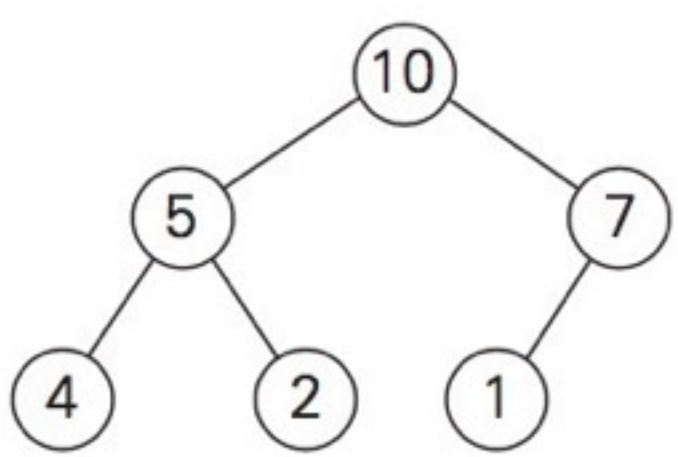
- ทุกๆใบในต้นไม้จะอยู่ในระดับเดียวกัน หรือระดับที่ติดกัน
- จุดในทุกๆระดับจะเต็ม ยกเว้นระดับล่างสุด
- ใบในระดับล่างสุด จะอยู่ชิดทางซ้าย
- คีย์ของข้อมูลที่จุดระดับพ่อ/แม่ จะมีค่ามากกว่าคีย์ของลูกหลาน
- ต้นไม้ย่อยทั้งสองของจุดใดๆ ก็คือ heap เช่นกัน

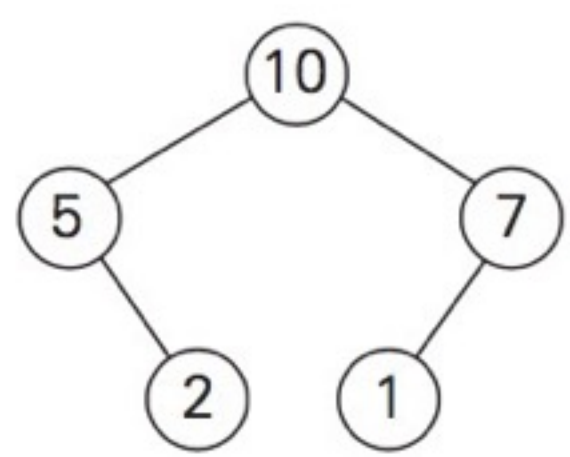


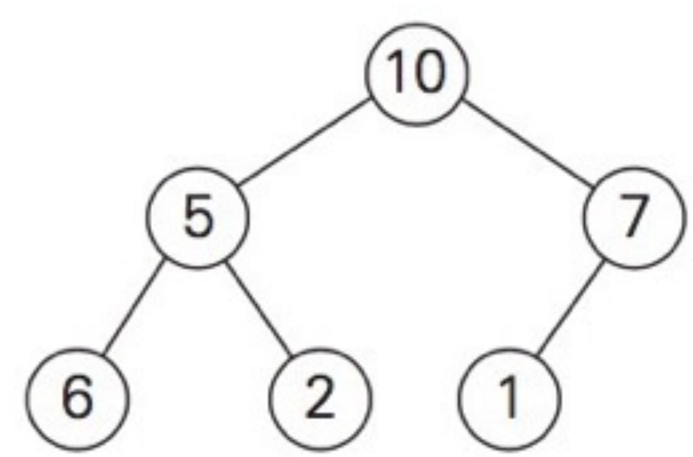
Y	R	P	D	F	B	K	A	C	E
0	1	2	3	4	5	6	7	8	9

Heap

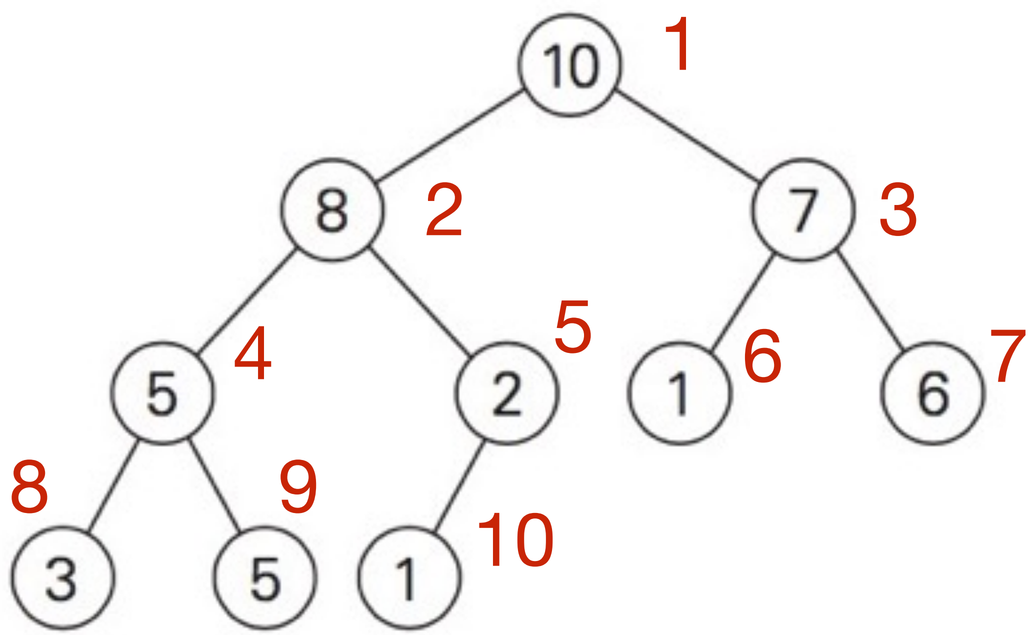
- A heap is a binary tree with the following properties:
  - Shape property is **essentially complete**, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing
  - **Parental dominance**: The key value at each node is at least those of its children. (Root is the largest)

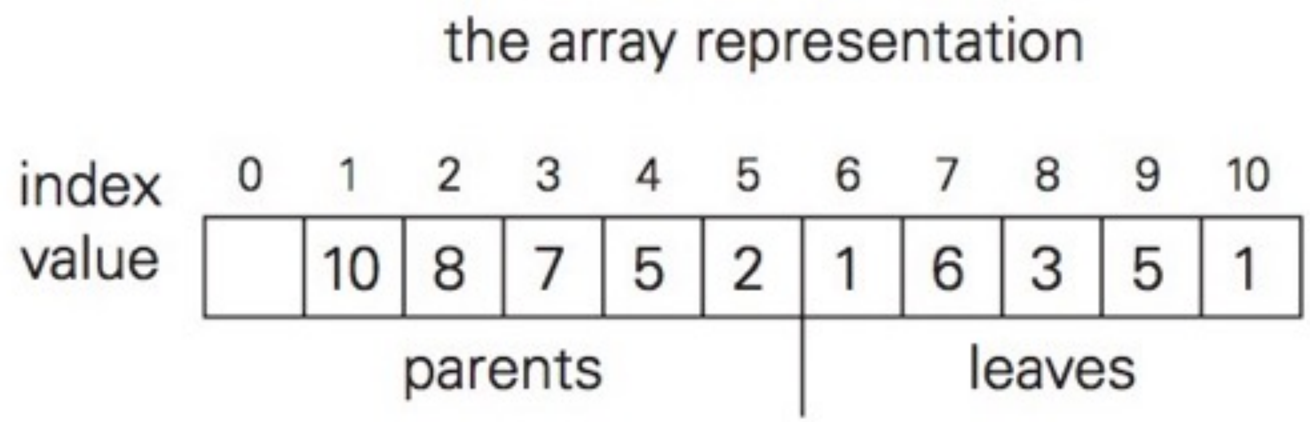






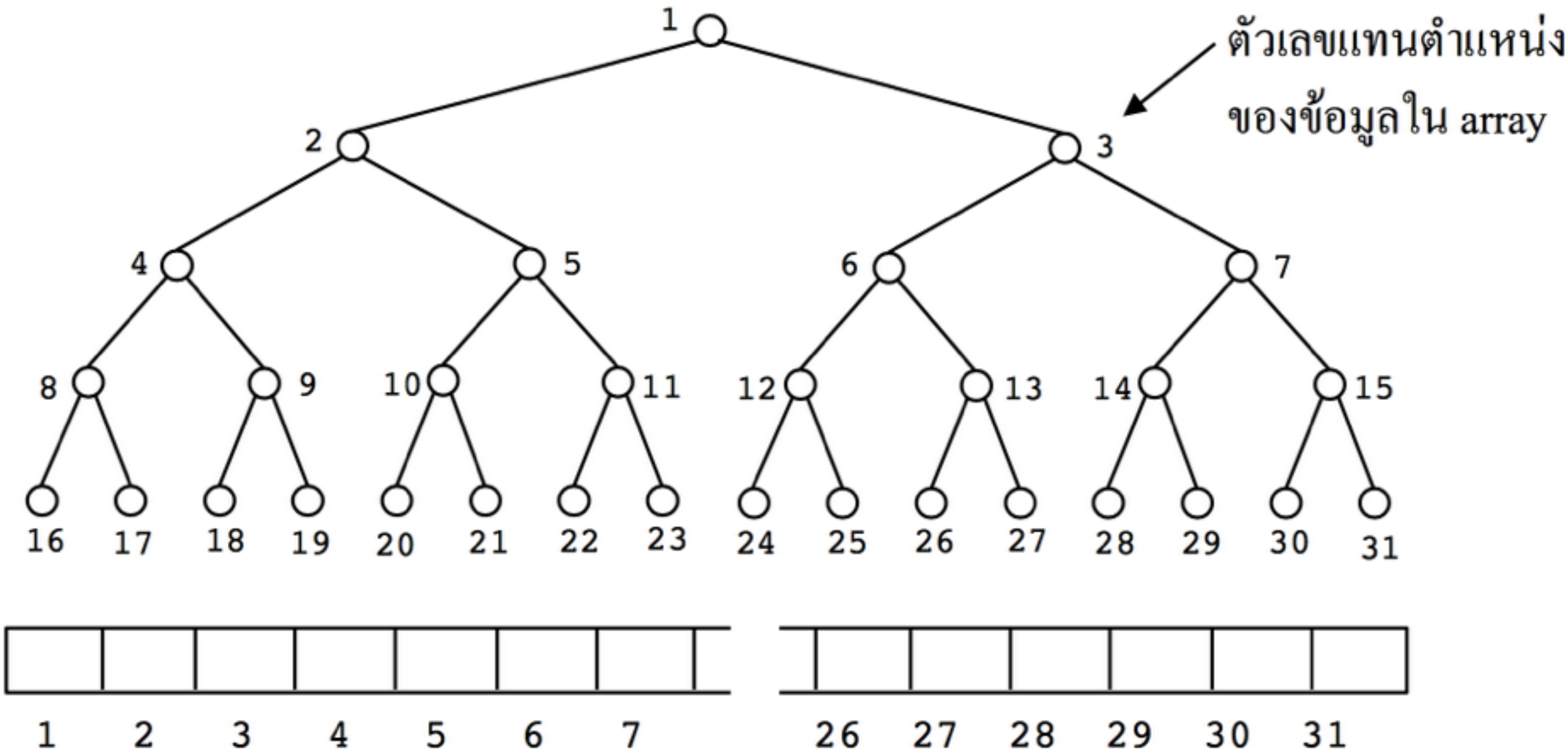
Array Representation of Heap





- Array size n+1 elements with position 0 unused.
- For node j
  - Left child index is 2j
  - Right child index is 2j+1
  - Parent index is floor(j/2)
- Non-leaf (parental) nodes at locations 1 to floor(n/2)

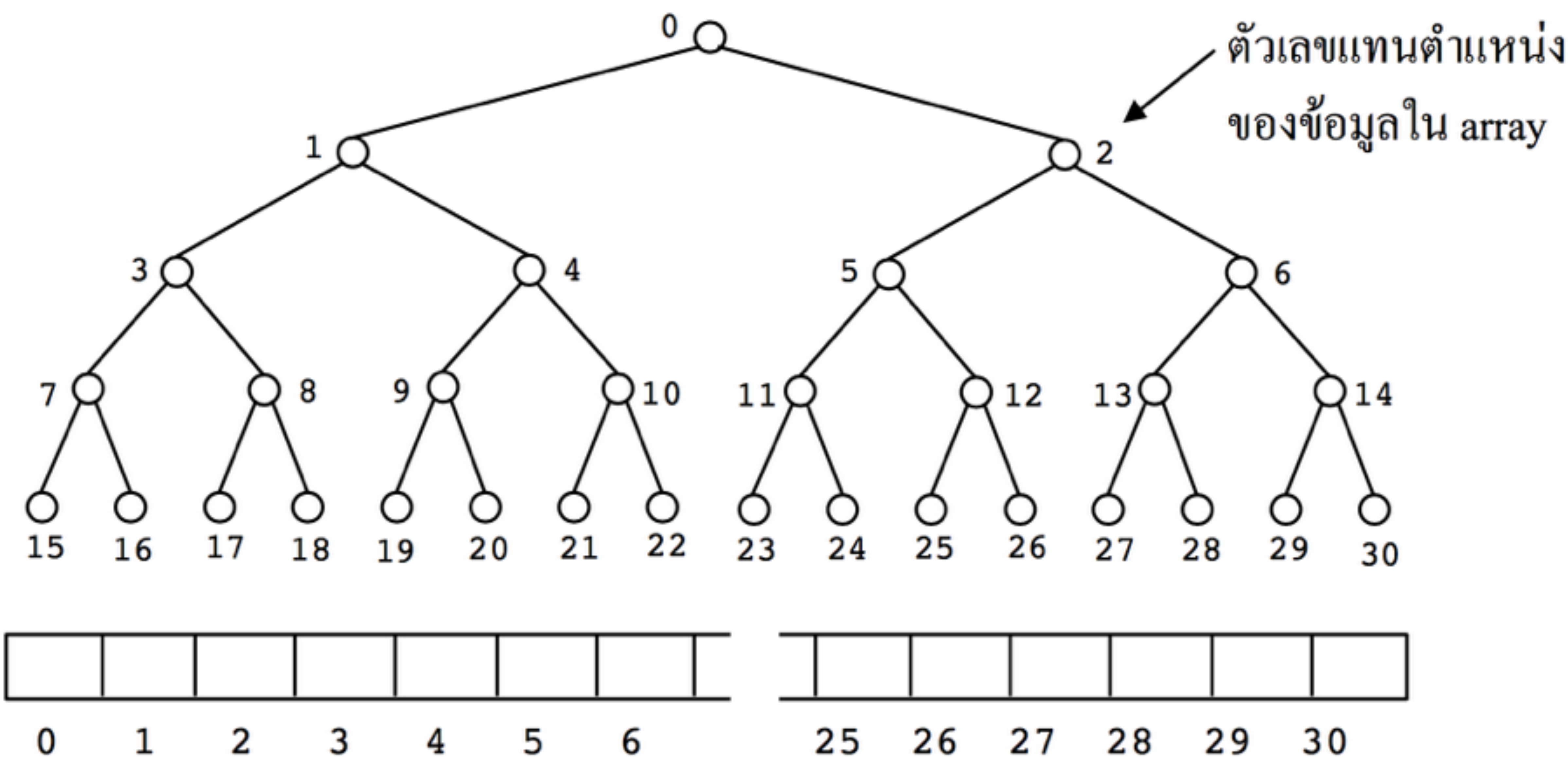
Array Implementation



ลูกทางซ้ายของจุดที่ตำแหน่ง k จะอยู่ที่ตำแหน่งที่ $2k$

" ขวา " " " " $2k+1$

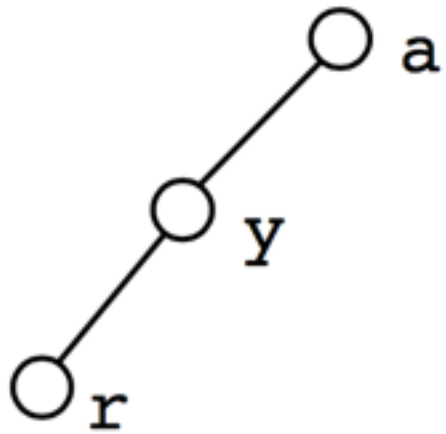
Array Implementation



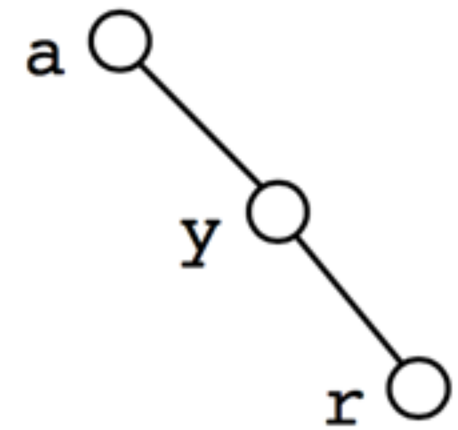
ลูกทางซ้ายของจุดที่ตำแหน่ง k จะอยู่ที่ตำแหน่งที่ $2k+1$

" ขวา " " " " $2k+2$

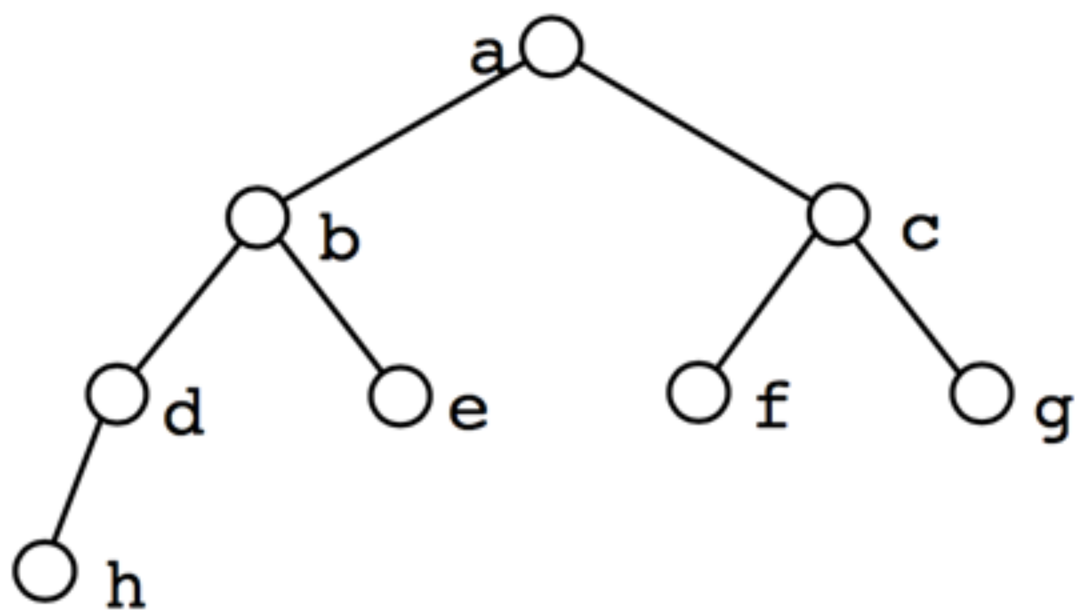
Array Implementation



a	y	-	r
0	1	2	3



a	-	y	-	-	-	r
0	1	2	3	4	5	6



a	b	c	d	e	f	g	h
0	1	2	3	4	5	6	7

Heap Construction

- Initialize an essentially complete binary tree from the keys given
- **Heapify** the tree as following:
  - Start with the last (rightmost) parental node, exchange key K with its children if the parental dominance does not hold.
  - Check the parental dominance for the new position of K and repeat until the key K goes to right location.
  - Continue with other parental nodes up to the root.

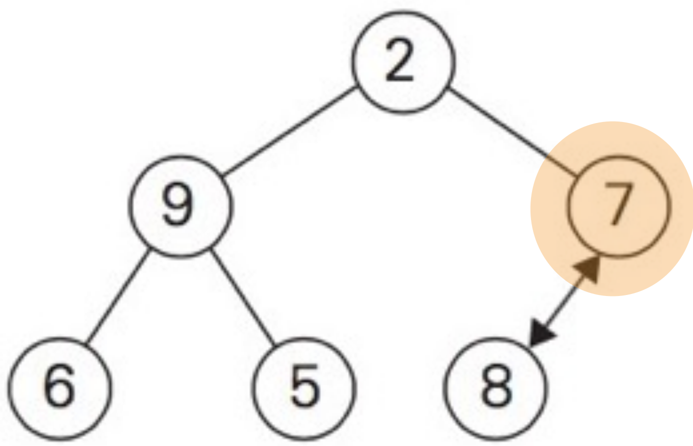
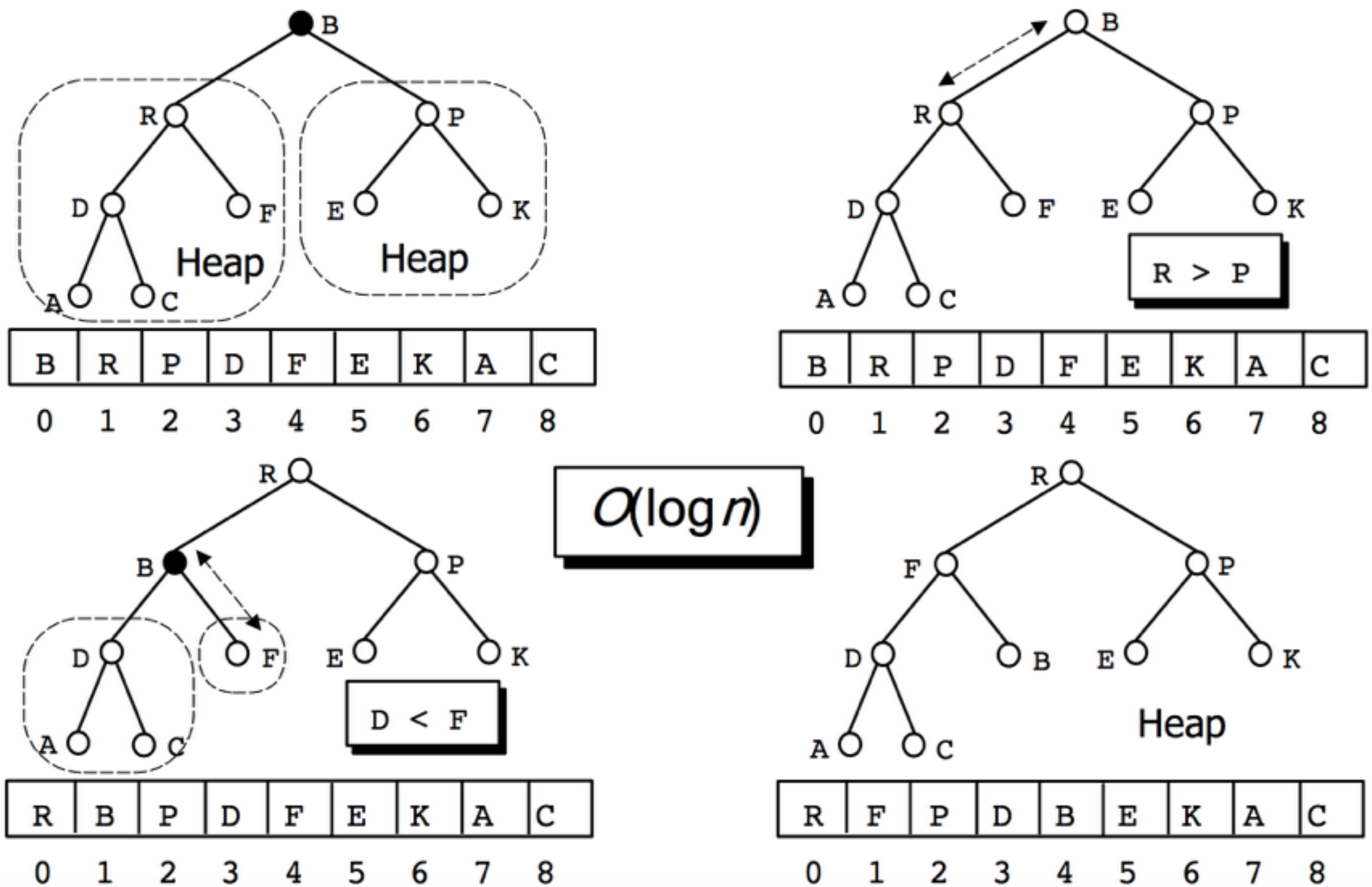


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

Heapify


B
R
P
D
F
E
K
A
C
Heap
Heap
B R P D F E K A C
0 1 2 3 4 5 6 7 8
R > P
B R P D F E K A C
0 1 2 3 4 5 6 7 8
O(log n)
D < F
R B P D F E K A C
0 1 2 3 4 5 6 7 8
Heap
R F P D B E K A C
0 1 2 3 4 5 6 7 8

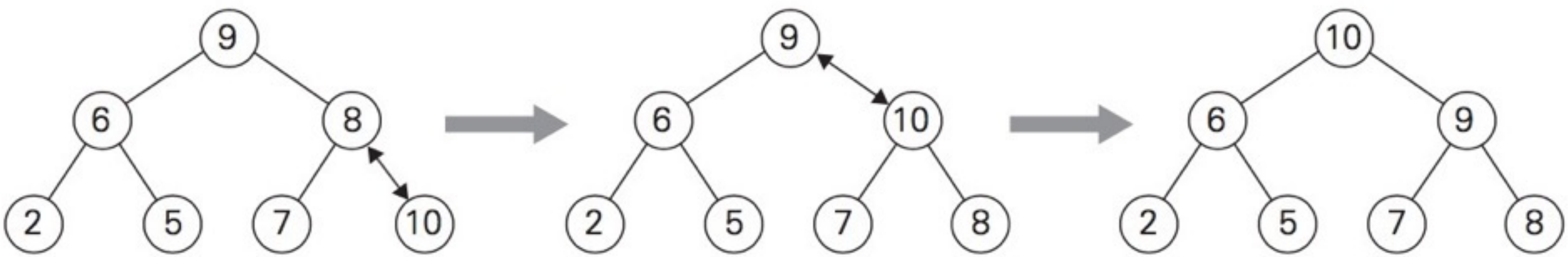
ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]                          Get current parental node
    heap ← false                               Parental dominance false
    while not heap and 2 ∗ k ≤ n do            Still have children
        j ← 2 ∗ k                              Get left child
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1      j is the larger child index
        if v ≥ H[j]
            heap ← true                        Parent already larger than child
        else H[k] ← H[j];   k ← j              Swap child with parent
    H[k] ← v                                   Put v to the correct position
```

Heapify

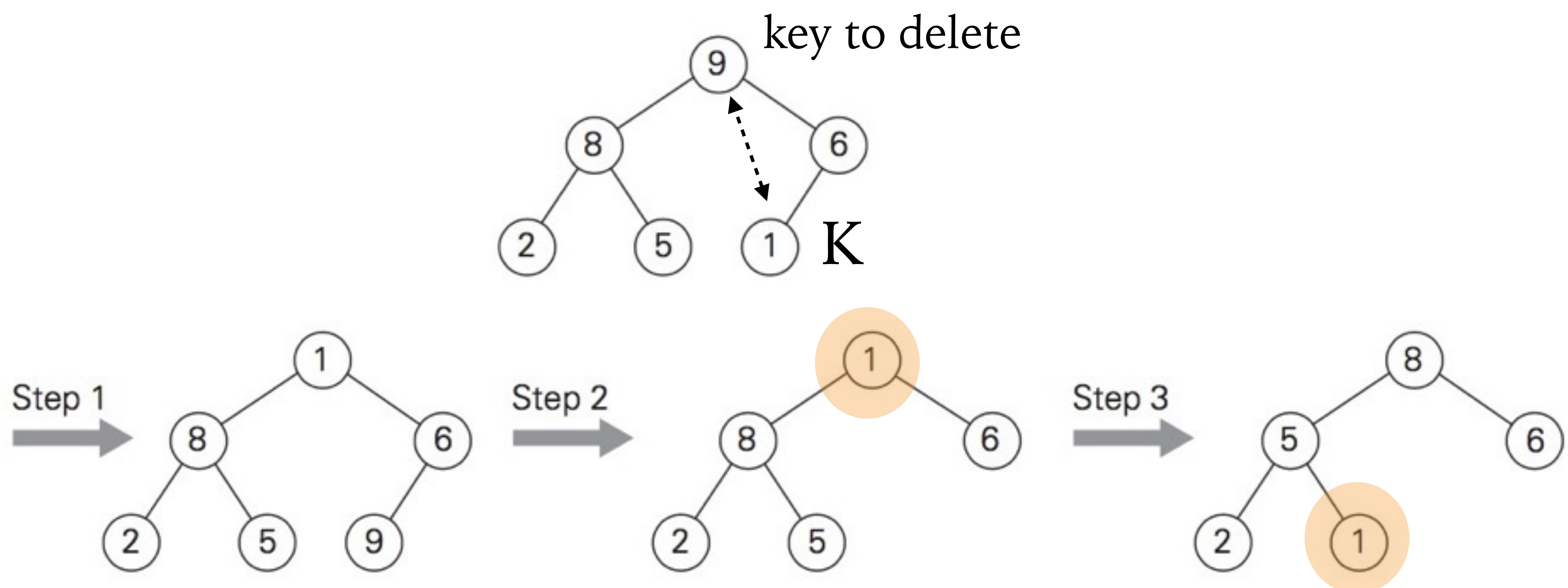
```
void  Heapify( ListType *pHeap, int i )
{
  int    largest;

  left  = LeftHeap( i );
  right = RightHeap( i );
  largest = i;
  if ( left  <= pHeap->count-1 ) &&
       GT( pHeap->entry[left].key, pHeap->entry[largest].key ) )
    largest = left;
  if ( right <= pHeap->count-1 ) &&
       GT( pHeap->entry[right].key, pHeap->entry[largest].key ) )
    largest = right;
  if ( largest != i ) {
    Swap( pHeap, i, largest );
    Heapify( pHeap, largest );
  }
}
```

Inserting Key into Heap



- Insert the new key to the last leaf.
- Repeatedly swap the new key with its parent until the parental dominance is satisfied.
- Also be used as *top-down heap construction*

Deleting Key from Heap



- Swap the key to delete with the key K in last leaf.
- Delete the last leaf.
- Heapify the tree by sifting K down the tree

Heapsort

- Stage 1 (heap construction): Construct a heap for a given array with HeapBottomUP().
- Stage 2 (maximum deletion): Apply the root-deletion operation $n-1$ times to the remaining heap.
    - The largest value (root) will be deleted first.
    - Array elements are eliminated in decreasing order.
- Since an element being deleted is placed last, the resulting array will be exactly the original array sorted in an increasing order.

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

Searching

- n comparisons in the worst case for linear search.

3 13 1 6 5 8 12 9 10

- If array is sorted first, the binary search can be applied.

3 13 1 6 5 8 12 9 10 $\Rightarrow$ 1 3 5 6 8 9 10 12 13

Search key = 12 1 3 5 6 **8** 9 10 12 13

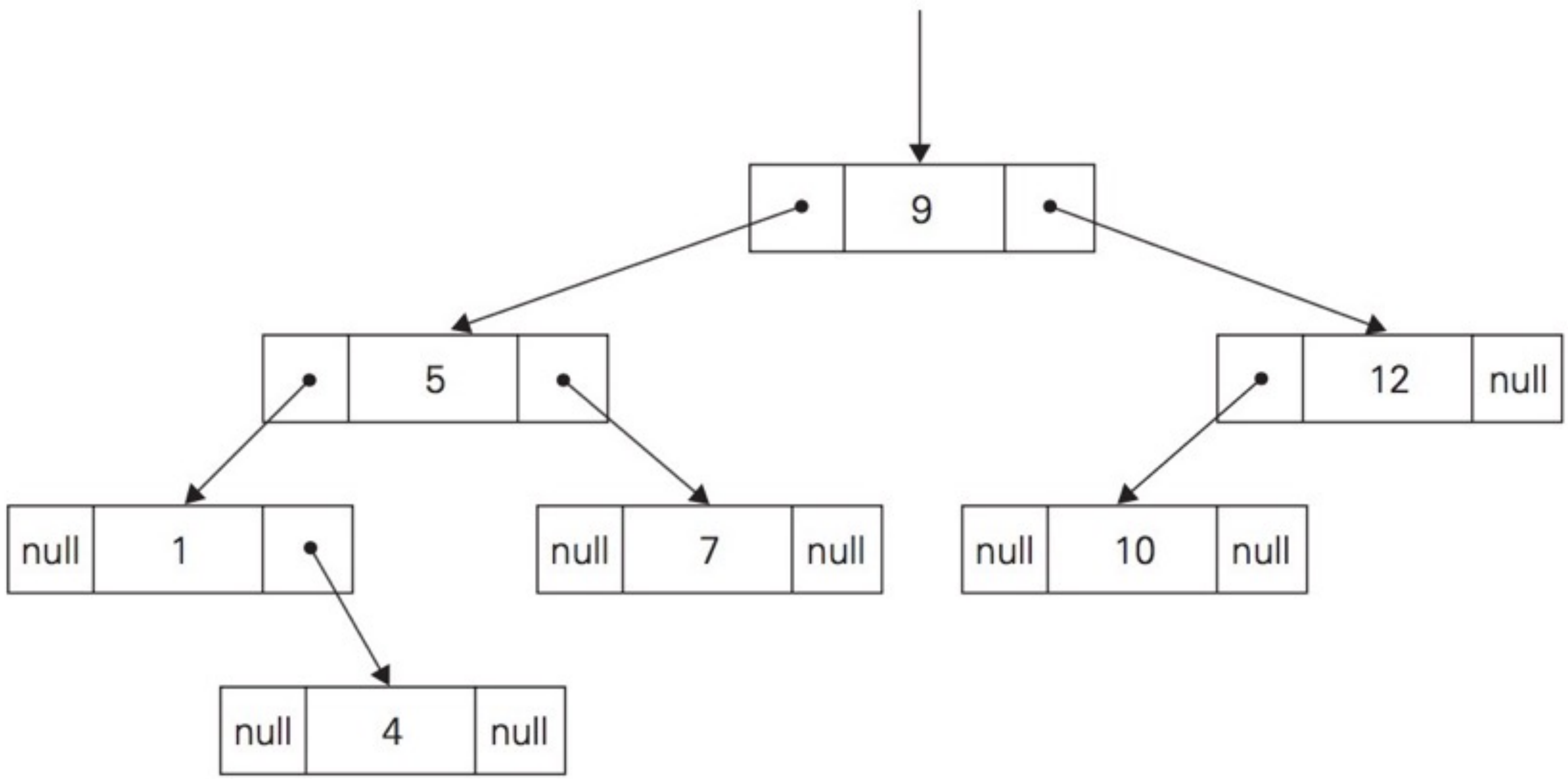
(Array-based implementation) 9 **10** 12 13

12

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

- Inferior to linear search but justifiable if we are to search the same list more than once. (Why?)

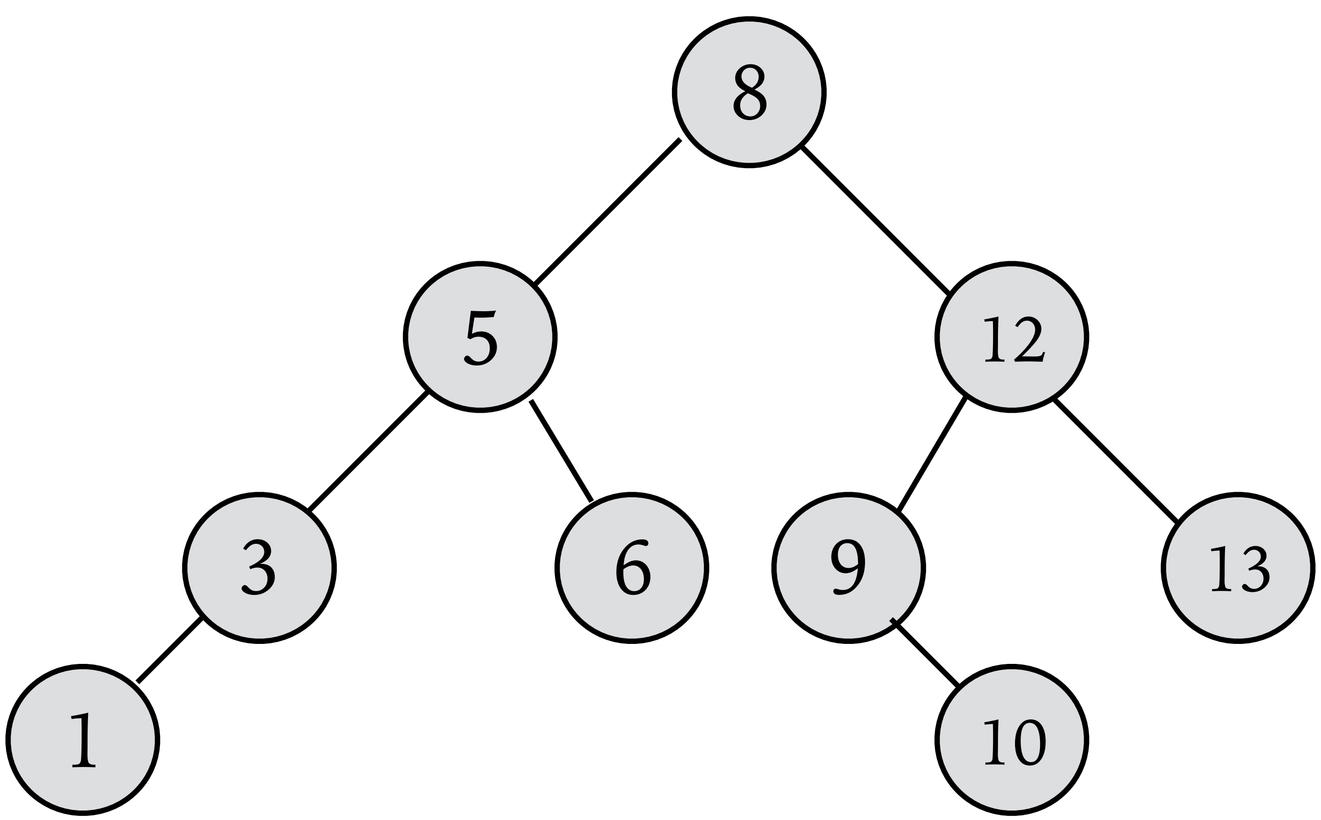
- Two implementations
  - Array-based
  - Tree-based, e.g., binary search tree, 2-3 tree
- What are the trade-offs?

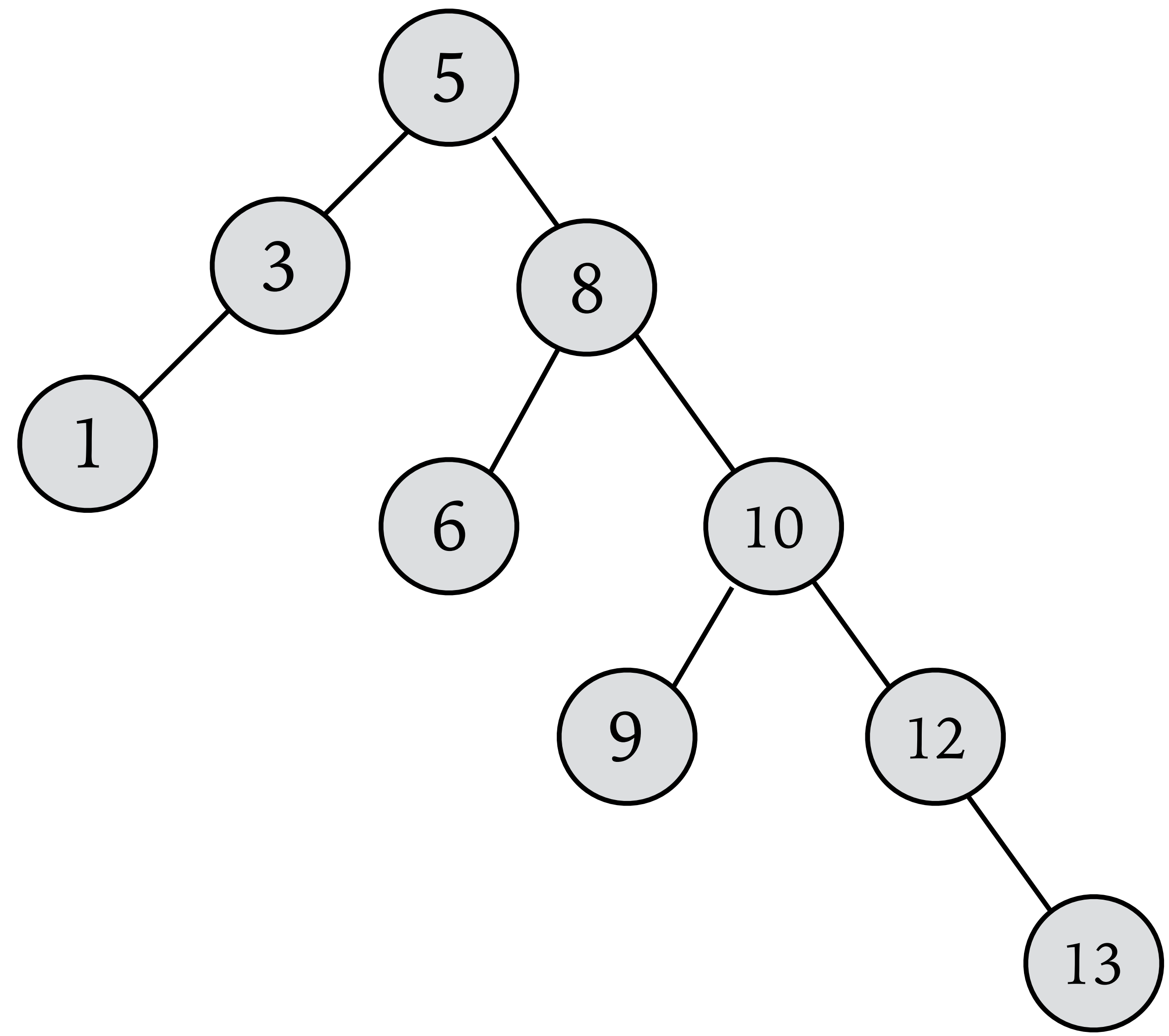


- Many forms when using tree-based implementation

1 3 5 6 8 9 10 12 13

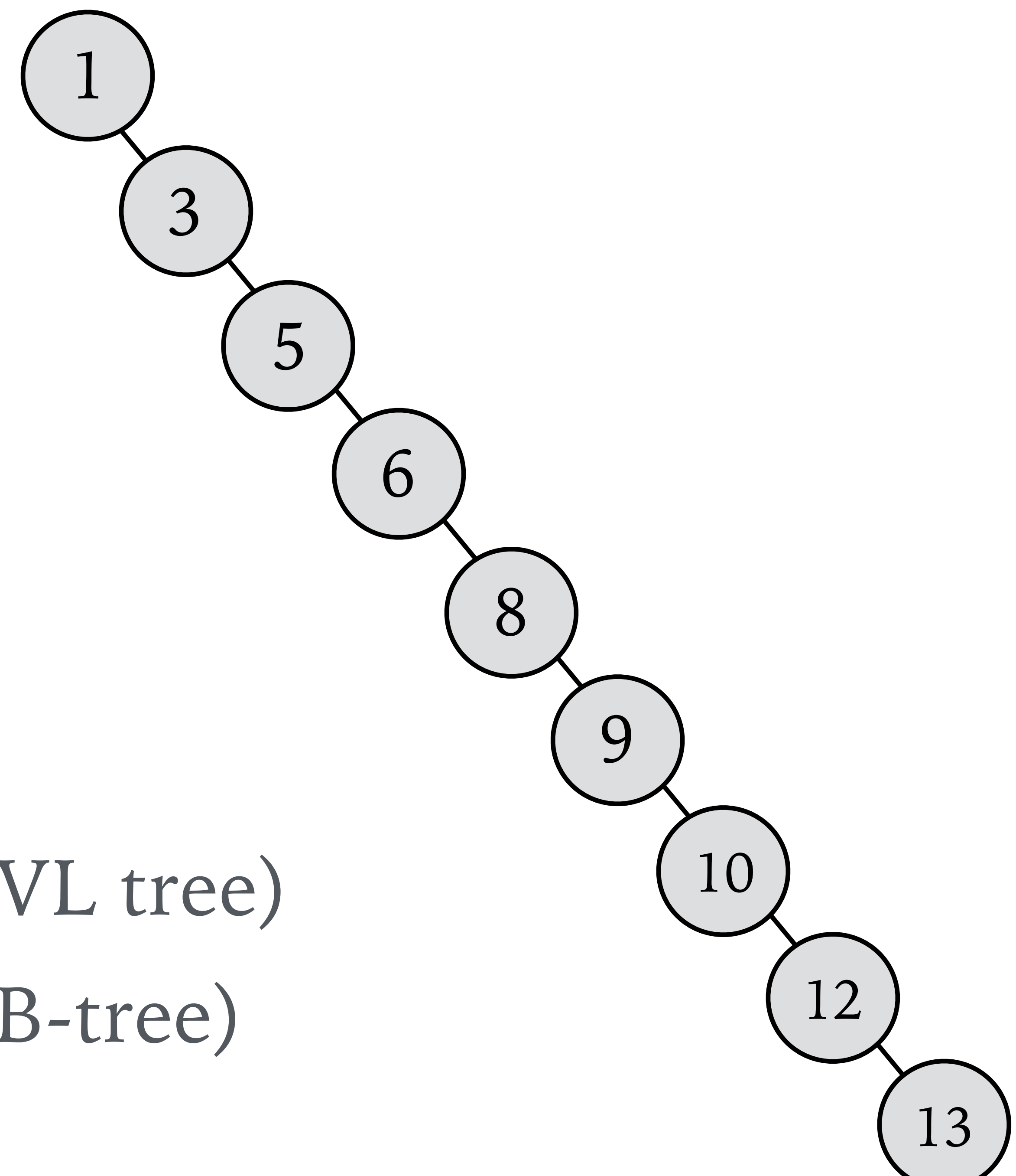
How many comparisons to search for key 13?





Balanced Search Tree

- Worst case running time is $\Theta(n)$ for degenerate tree (height n-1)



- Need to *balance* the tree
  - Instance simplication (AVL tree)
  - Representation change (B-tree)

G. M. Adelson-Velsky and E. M. Landis

AVL Tree

- **Balance factor** of all nodes is either 0, or +1 or -1.

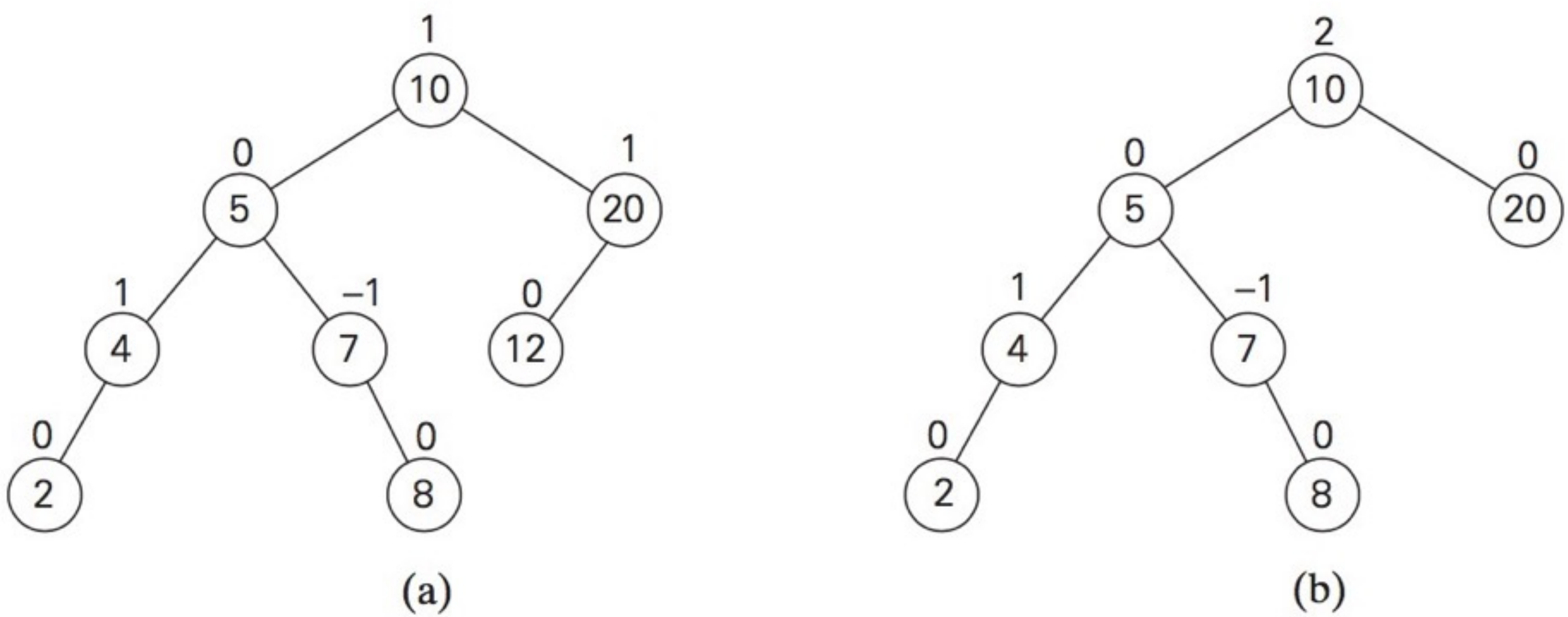
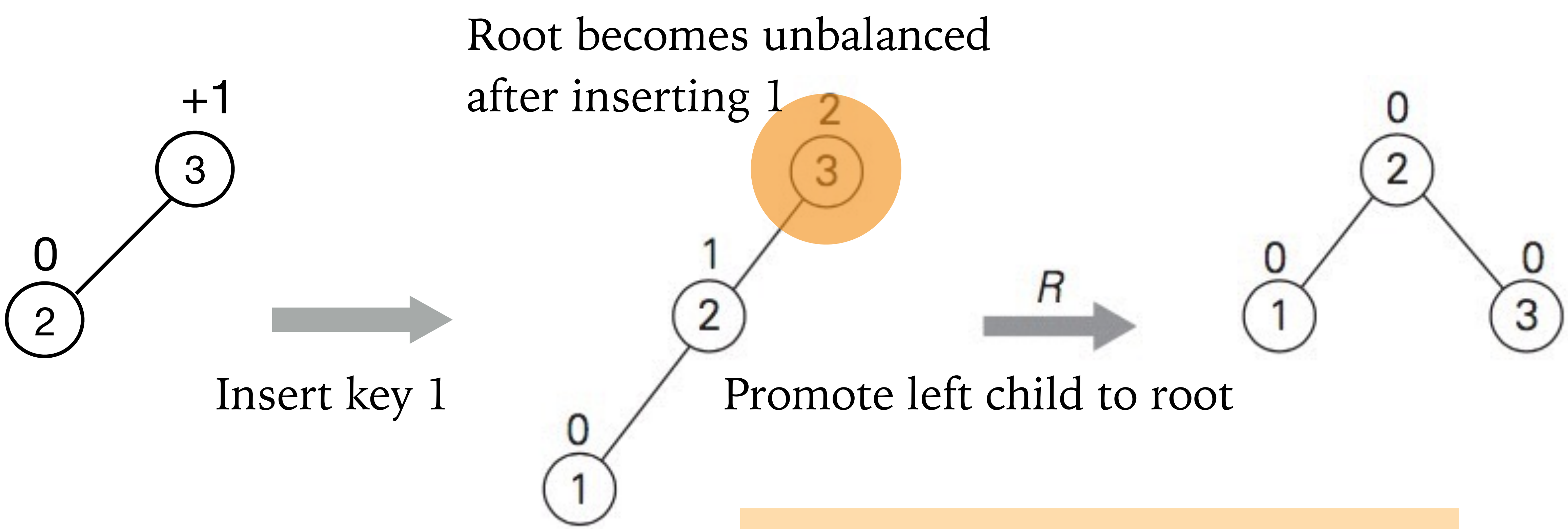


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

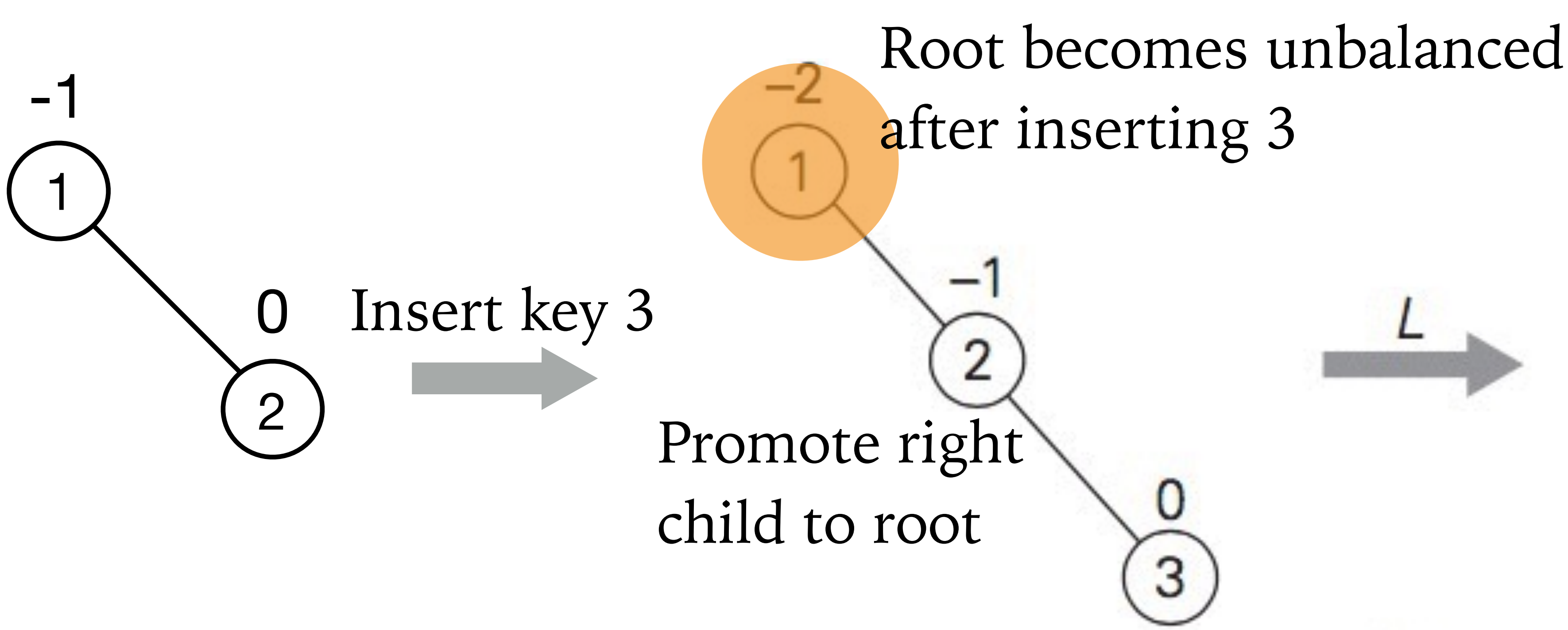
- Perform rotation at the **unbalanced node closest the newly inserted node**.
- Four types of rotations depending on where the new key has been inserted
  - Single R-rotation
  - Single L-rotation
  - Double LR-rotation
  - Double RL-rotation

New key inserted into the ***left subtree of the left child*** of a tree whose root had the balance of +1 before the insertion.



Single R-rotation
(Promote left child to root)

New key inserted into the ***right subtree of the right child*** of a tree whose root had the balance of -1 before the insertion.



Single L-rotation
(Promote right child to root)

R-Rotation

New key inserted into the ***left subtree of the left child*** of a tree whose root had the balance of +1 before the insertion.

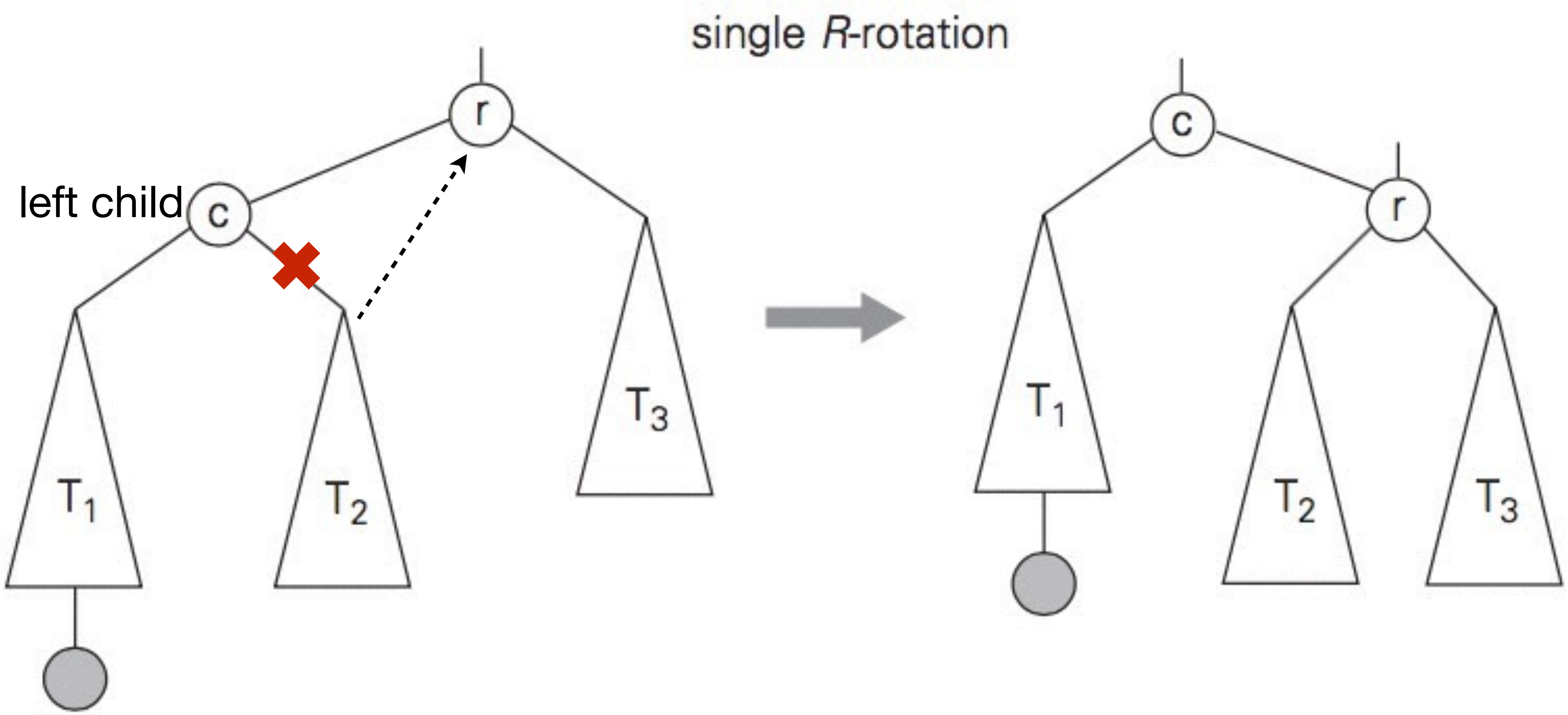
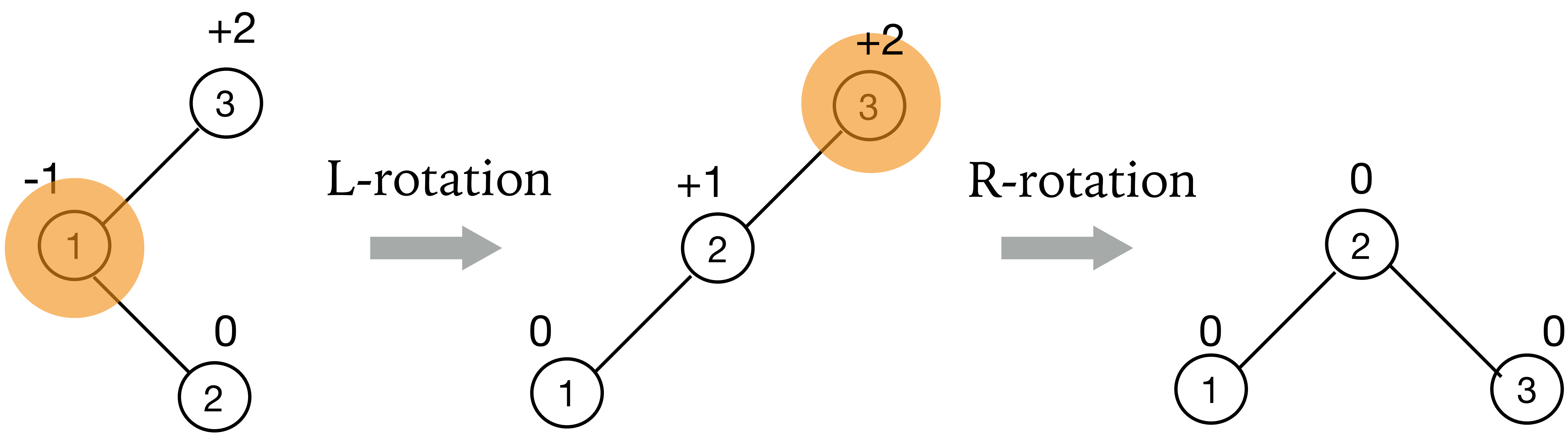


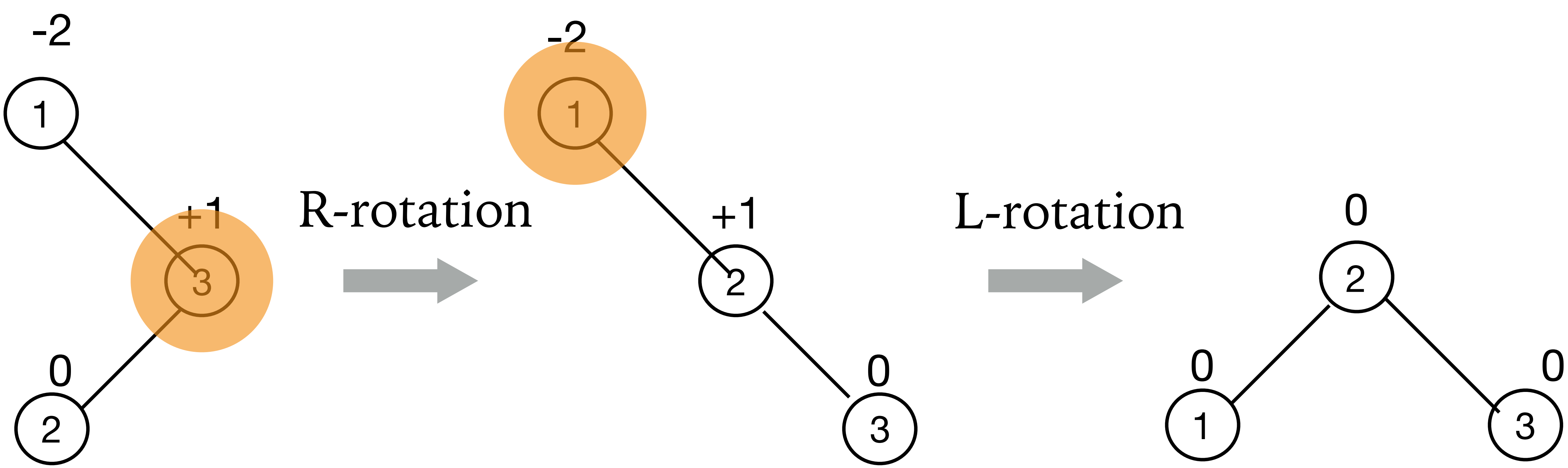
FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

New key inserted into the ***right subtree of the left child*** of a tree whose root had the balance of +1 before the insertion.



Double LR-rotation
(L-rotation followed by R-rotation)

New key inserted into the *left subtree of the right child* of a tree whose root had the balance of -1 before the insertion.



Double RL-rotation
(R-rotation followed by L-rotation

Double LR-rotation

New key inserted into the *right subtree of the left child* of a tree whose root had the balance of +1 before the insertion.

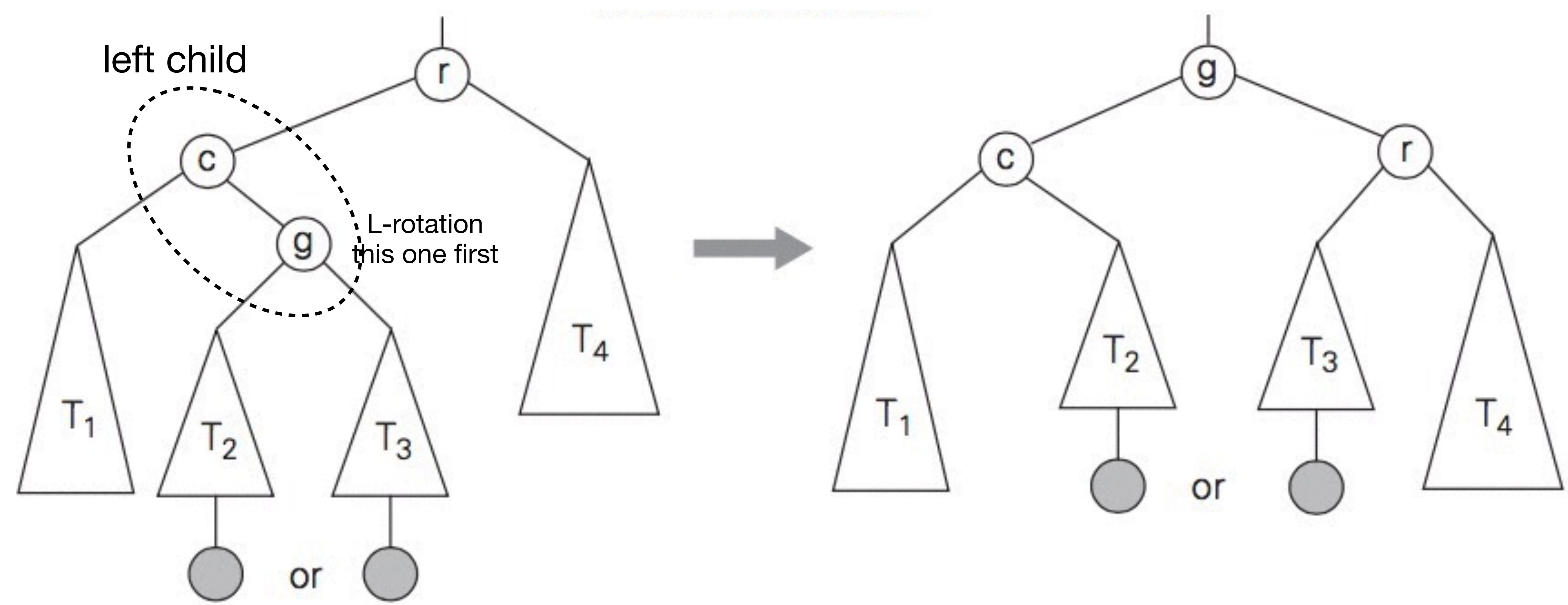
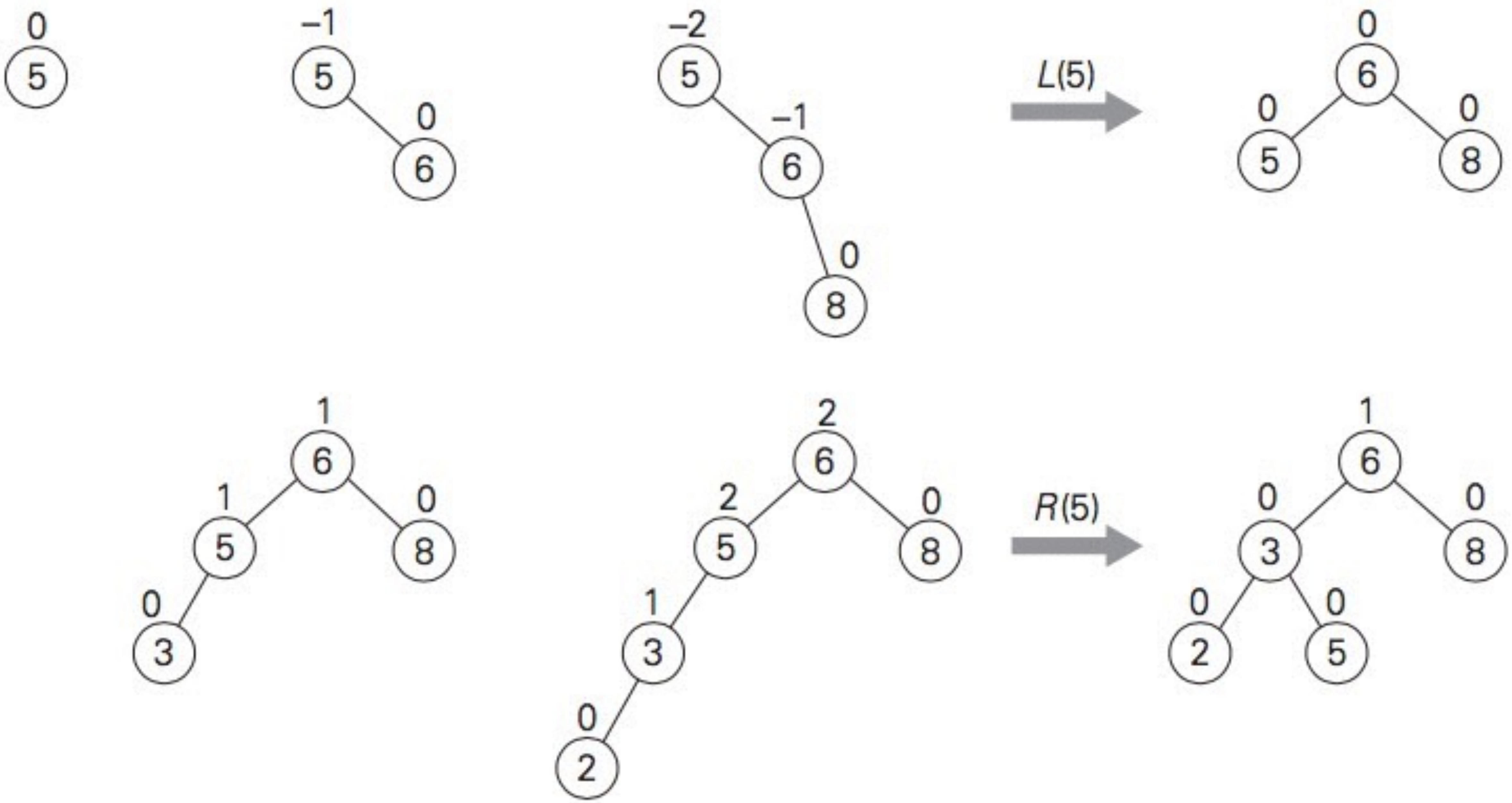
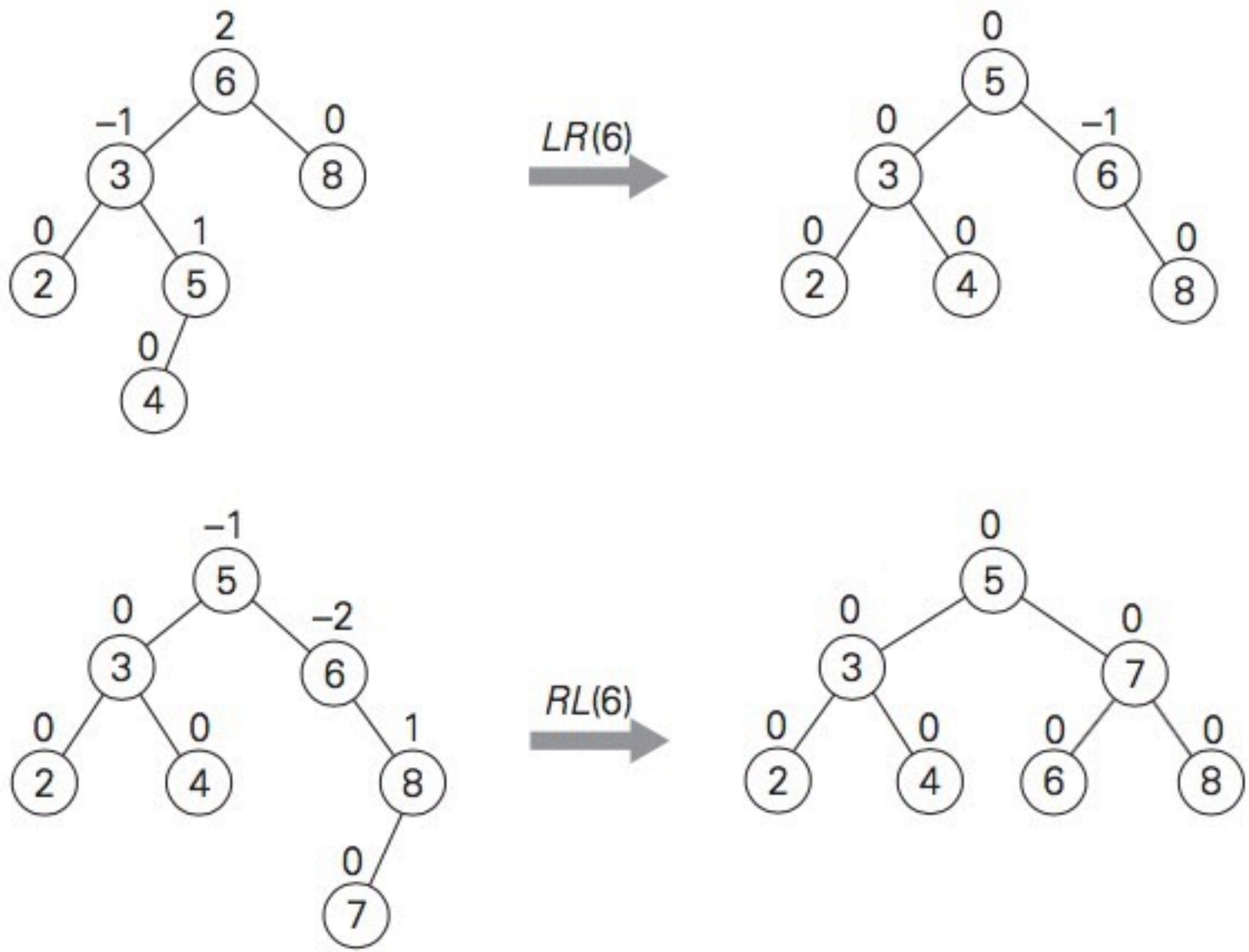


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

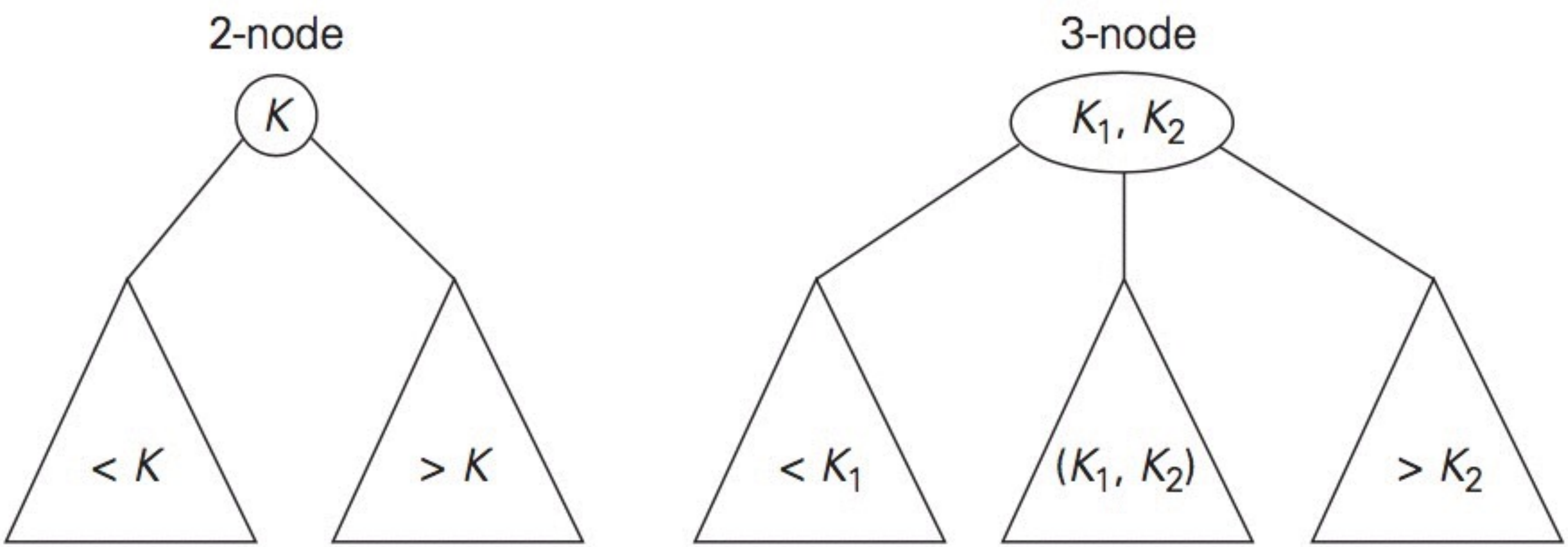
Sequence inserted: 5, 6, 8, 3, 2, 4, 7



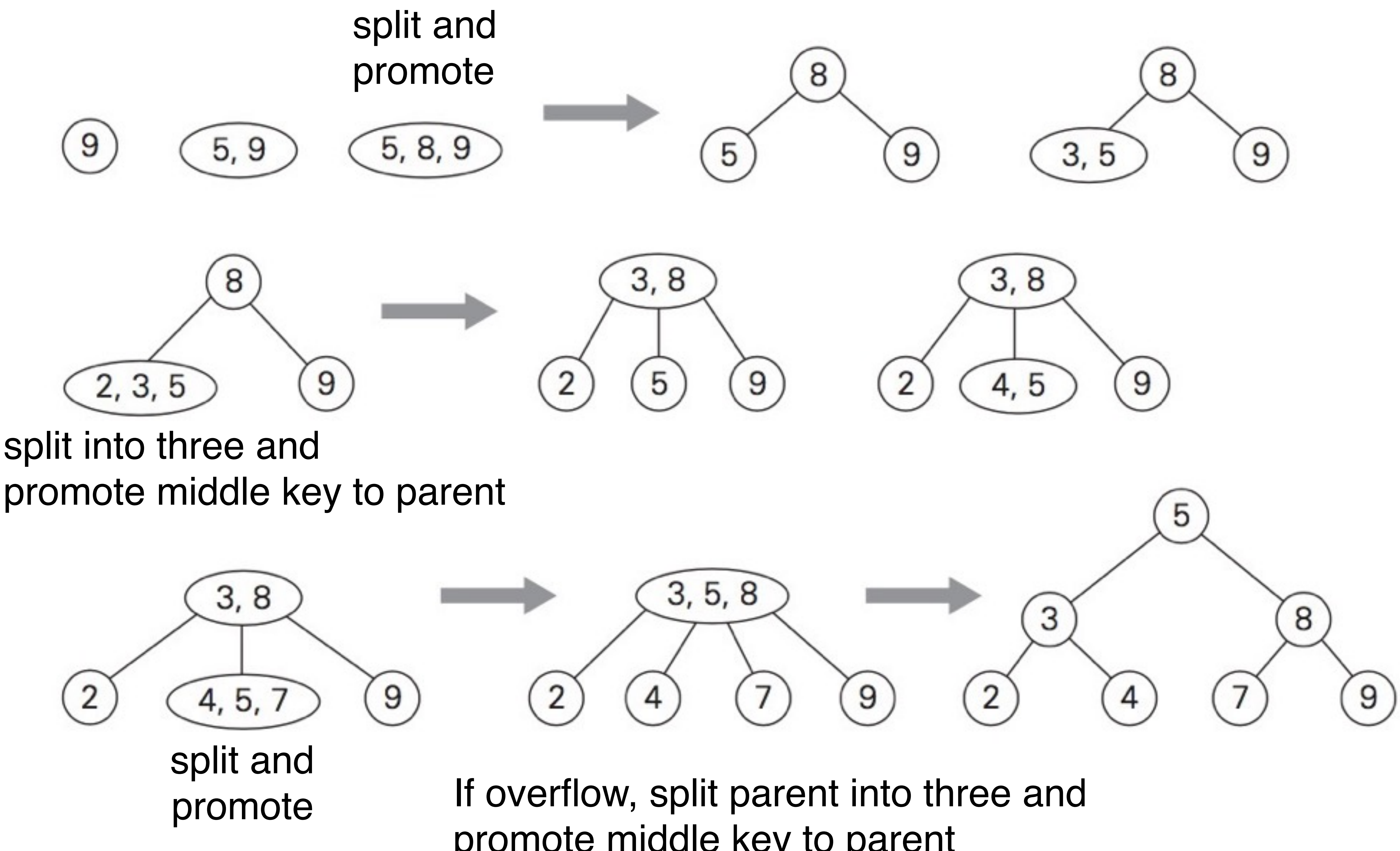

LR(6)
RL(6)

2-3 Tree

- A 2-3 tree is a search tree that
  - may have 2-nodes and 3-nodes
  - height-balanced (all leaves are on the same level)



Sequence Inserted: 9, 5, 8, 3, 2, 4, 7



Summary

- Transform a problem so that it is easier to solve.
- Instance simplification by presorting
  - Take $\Theta(n \log n)$ to sort but $\Theta(\log n)$ to search
  - More efficient for searching same list many times.
- Balance search tree with AVL to maintain good performance
- Representation change by using a heap
  - Allow for implementing priority list as well as sorting.