Other Binary Trees

AVL Trees

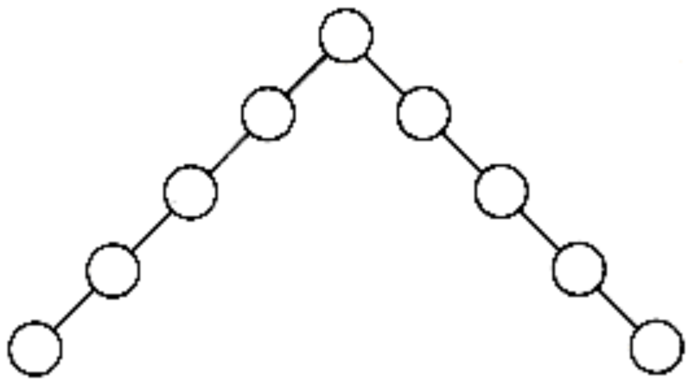
- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

Definition:

An AVL tree is a binary search tree such that
for any node in the tree, the height of the left and
right subtrees can differ by at most 1.

Balance Factor

- With each node of the AVL tree is associated a ***balance factor*** that is left high, equal or right high according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.



A bad binary tree. Requiring balance at the root is not enough.

Figure 19.21

Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)

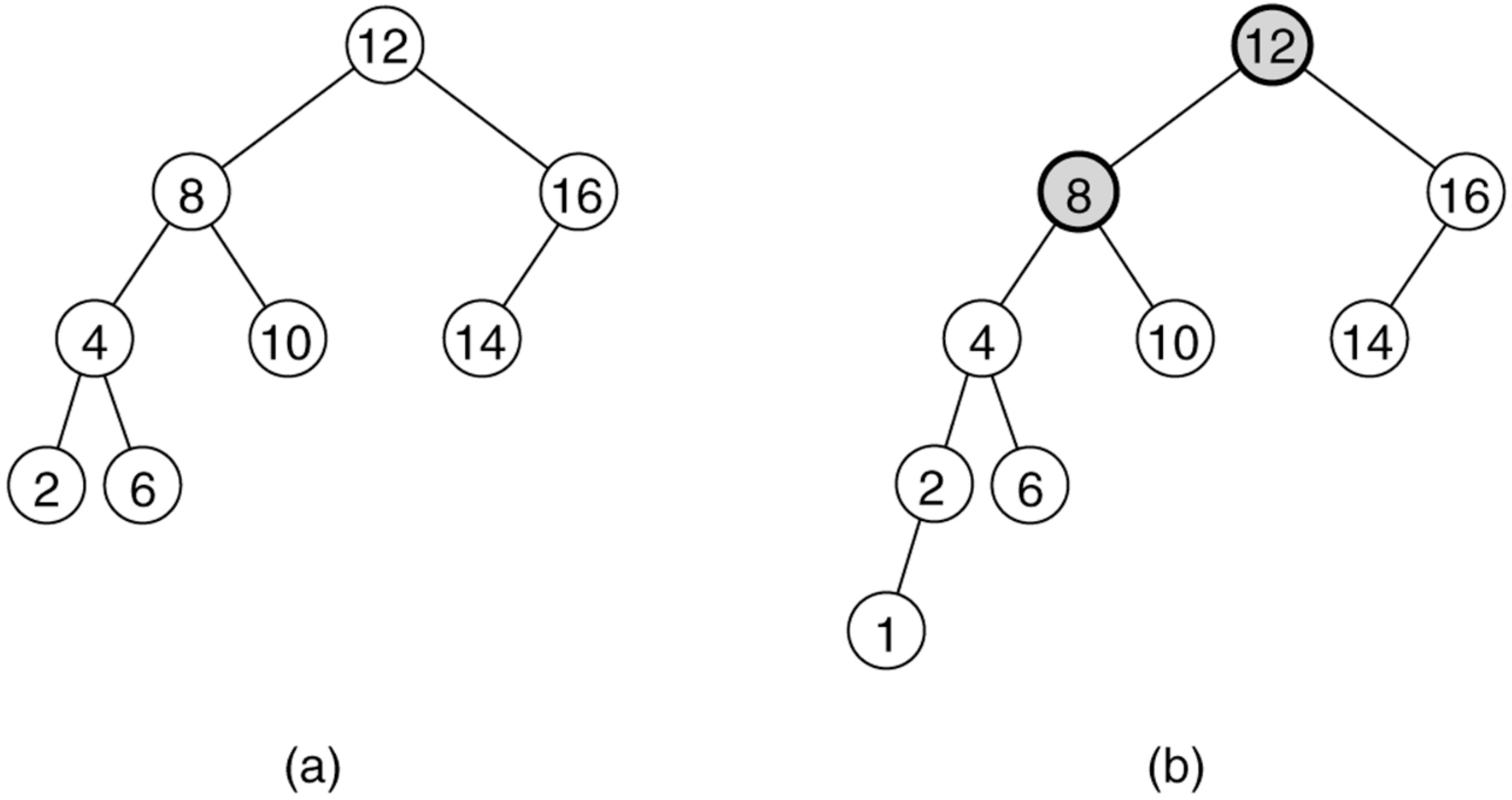
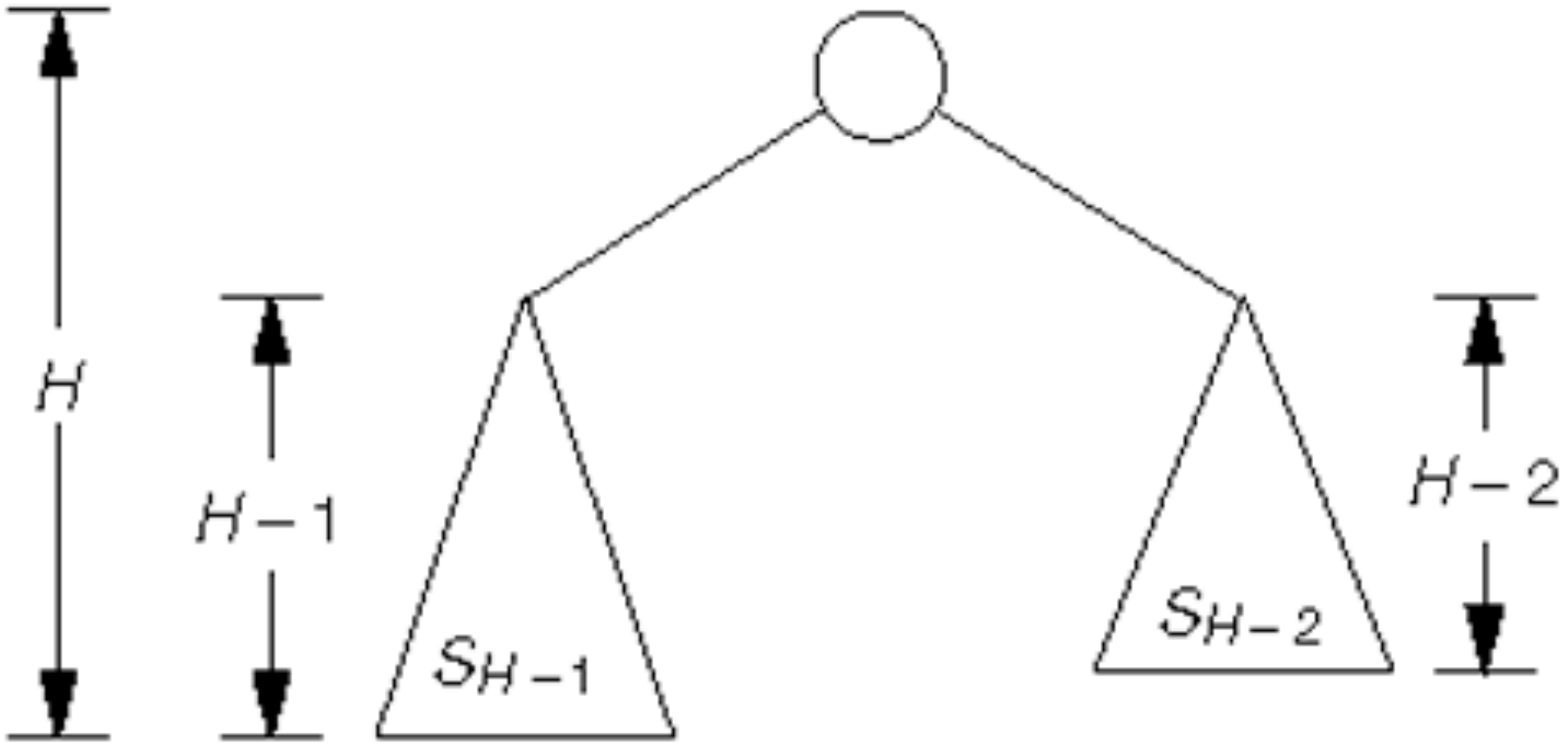


Figure 19.22

Minimum tree of height *H*



Properties

- The depth of a typical node in an AVL tree is very close to the optimal $log_2 N$.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

Rebalancing

- Suppose the node to be rebalanced is X. There are 4 cases that we might have to fix (two are the mirror images of the other two):
  1. An insertion in the left subtree of the left child of X,
  2. An insertion in the right subtree of the left child of X,
  3. An insertion in the left subtree of the right child of X, or
  4. An insertion in the right subtree of the right child of X.
- Balance is restored by tree *rotations*.

Balancing Operations: Rotations

- Case 1 and case 4 are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation*.
- Case 2 and case 3 are symmetric and requires the same operation for balance.
  - Cases 2,3 are handled by *double rotation*.

Figure 19.23

Single rotation to fix case 1: Rotate right

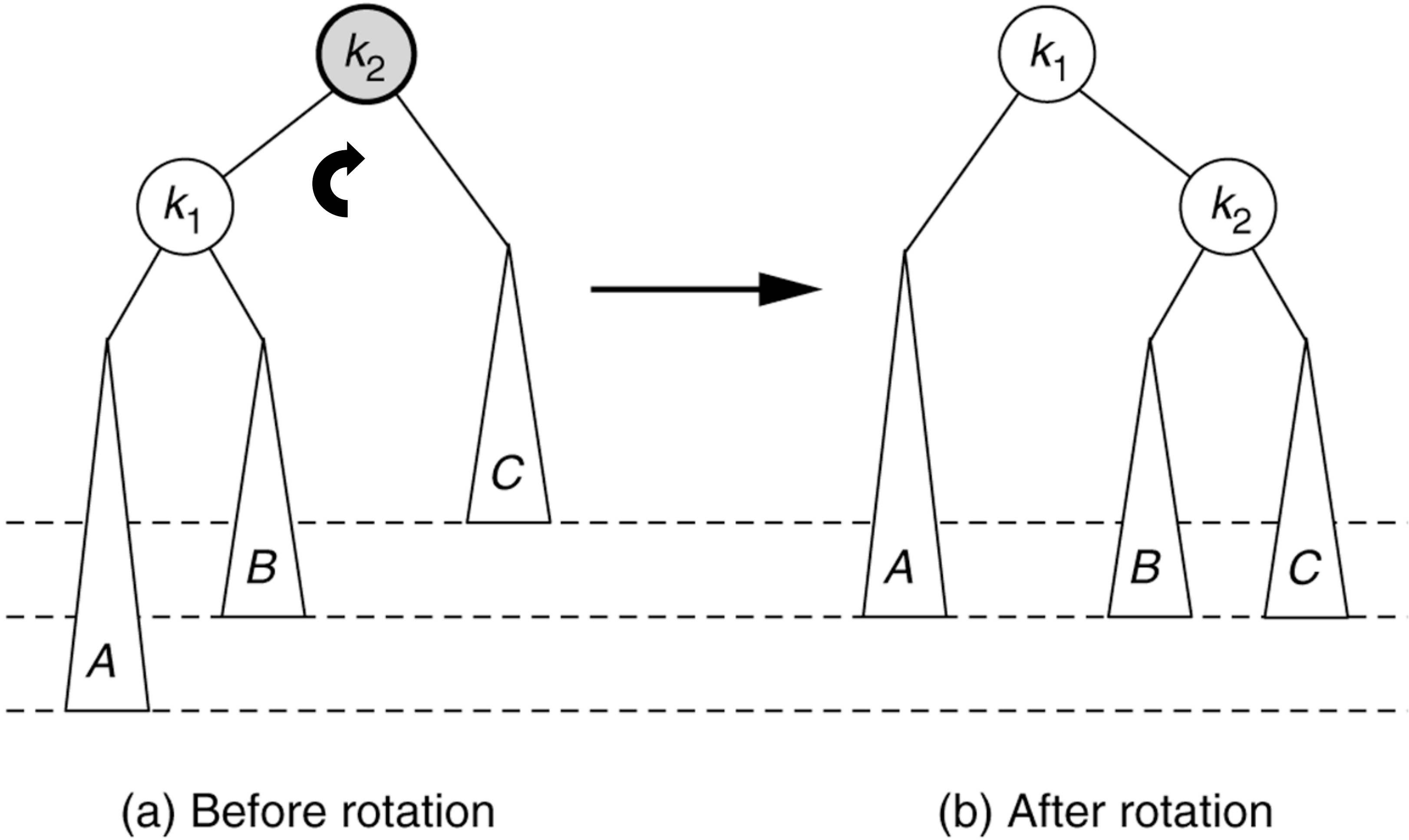
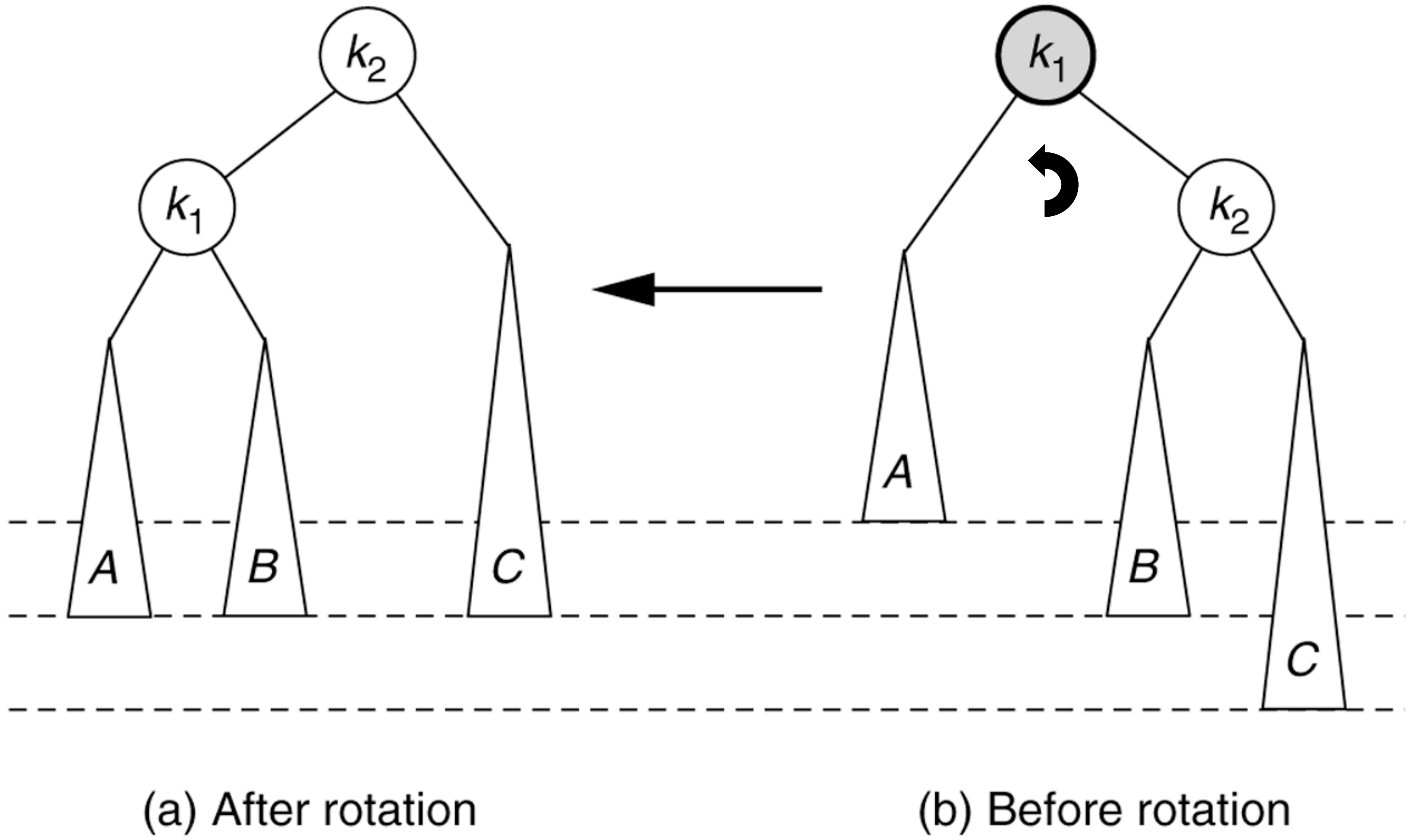
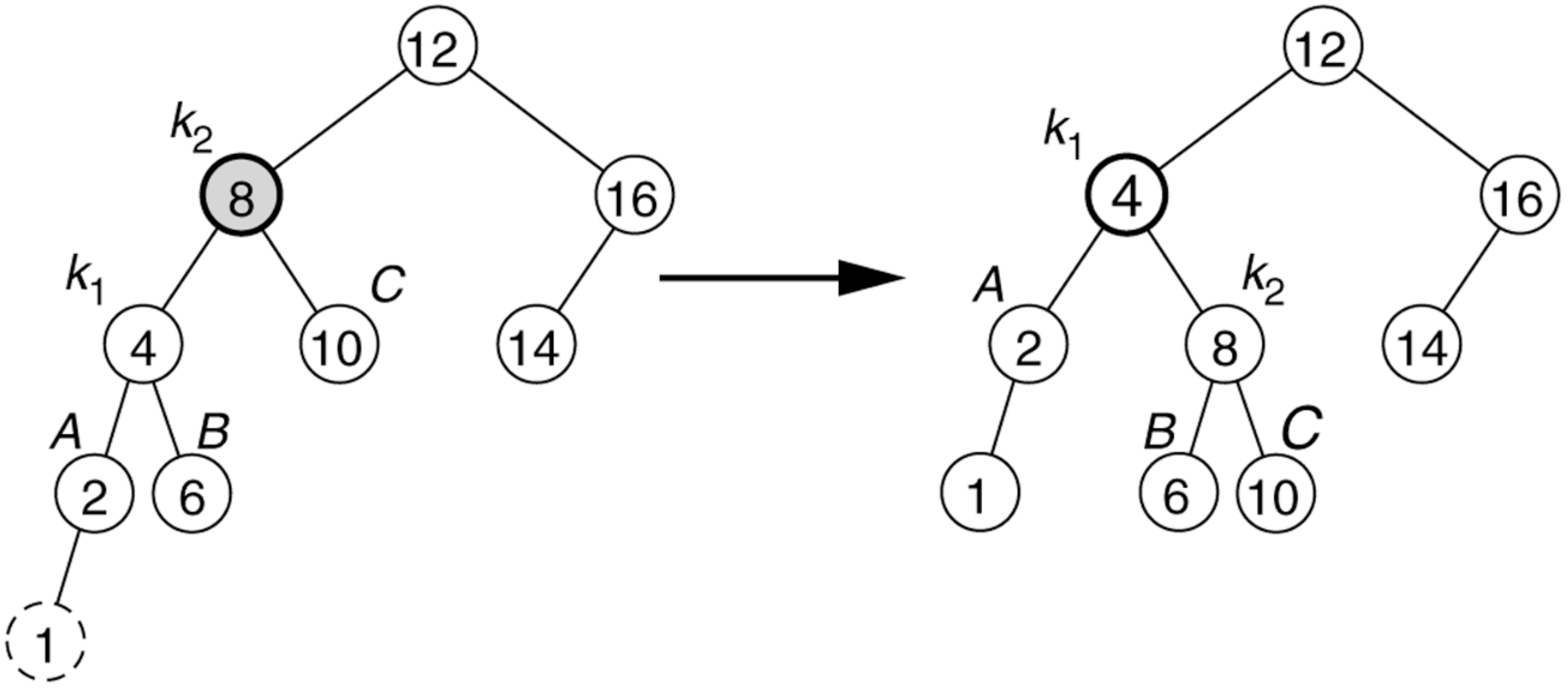


Figure 19.26

Symmetric single rotation to fix case 4 : Rotate left



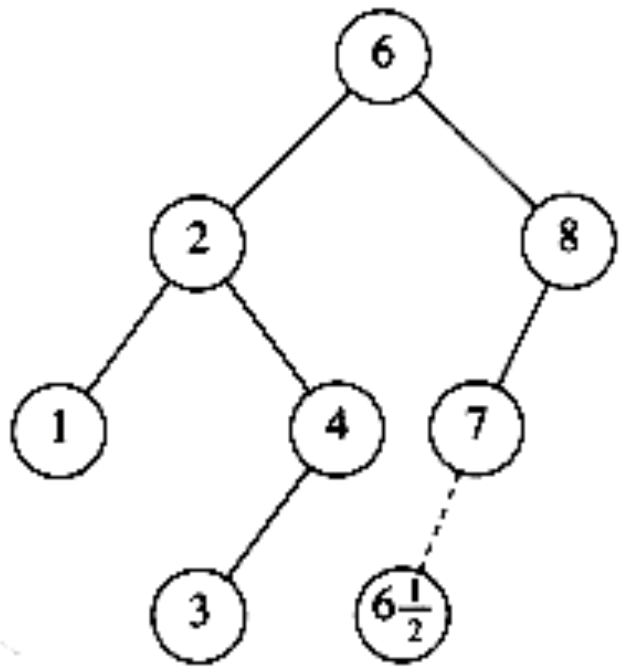
Example 1: Single rotation fixes an AVL tree after insertion of 1.

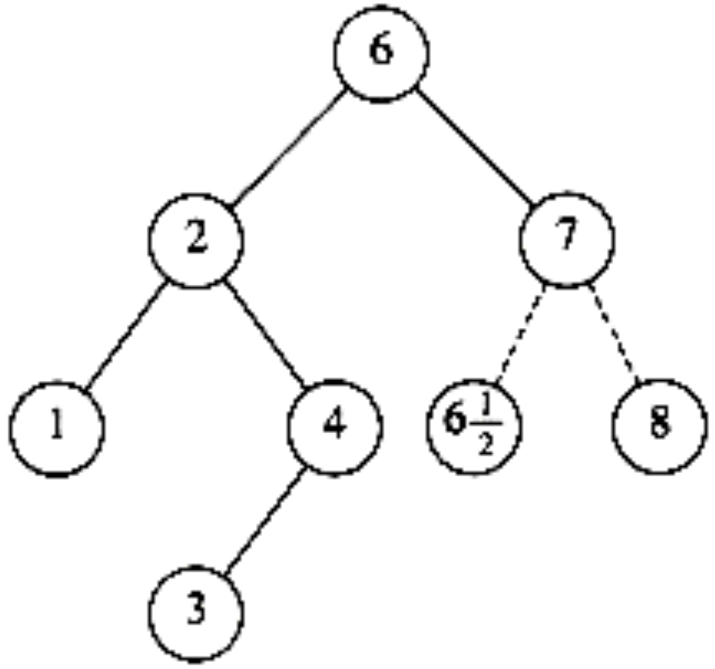


(a) Before rotation

(b) After rotation

Example 2: Single rotation fixes an AVL tree after insertion of 6.5





Analysis

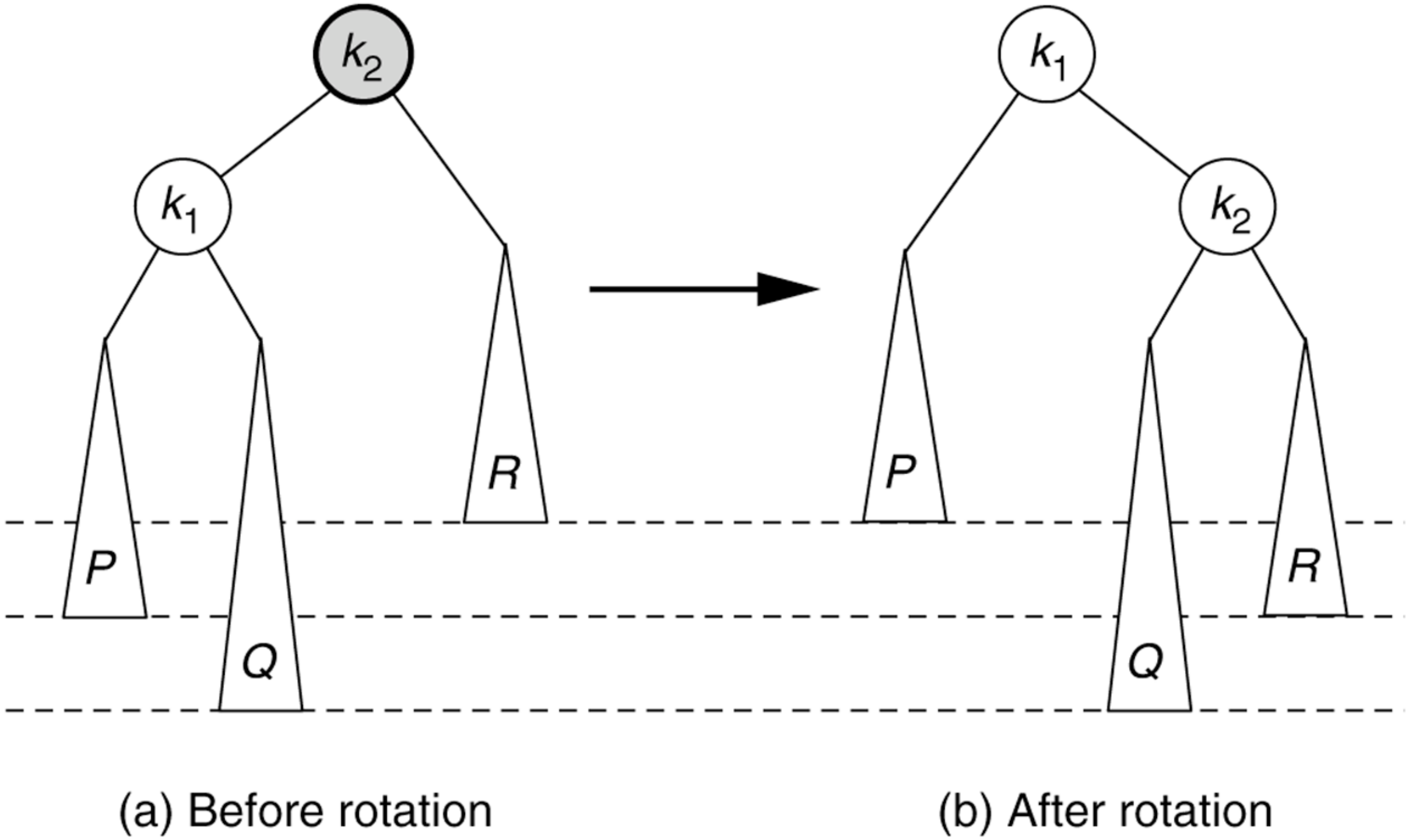
- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is $O(\log_2 N)$

Double Rotation

- Single rotation does not fix the inside cases (2 and 3).
- These cases require a *double* rotation, involving three nodes and four subtrees.

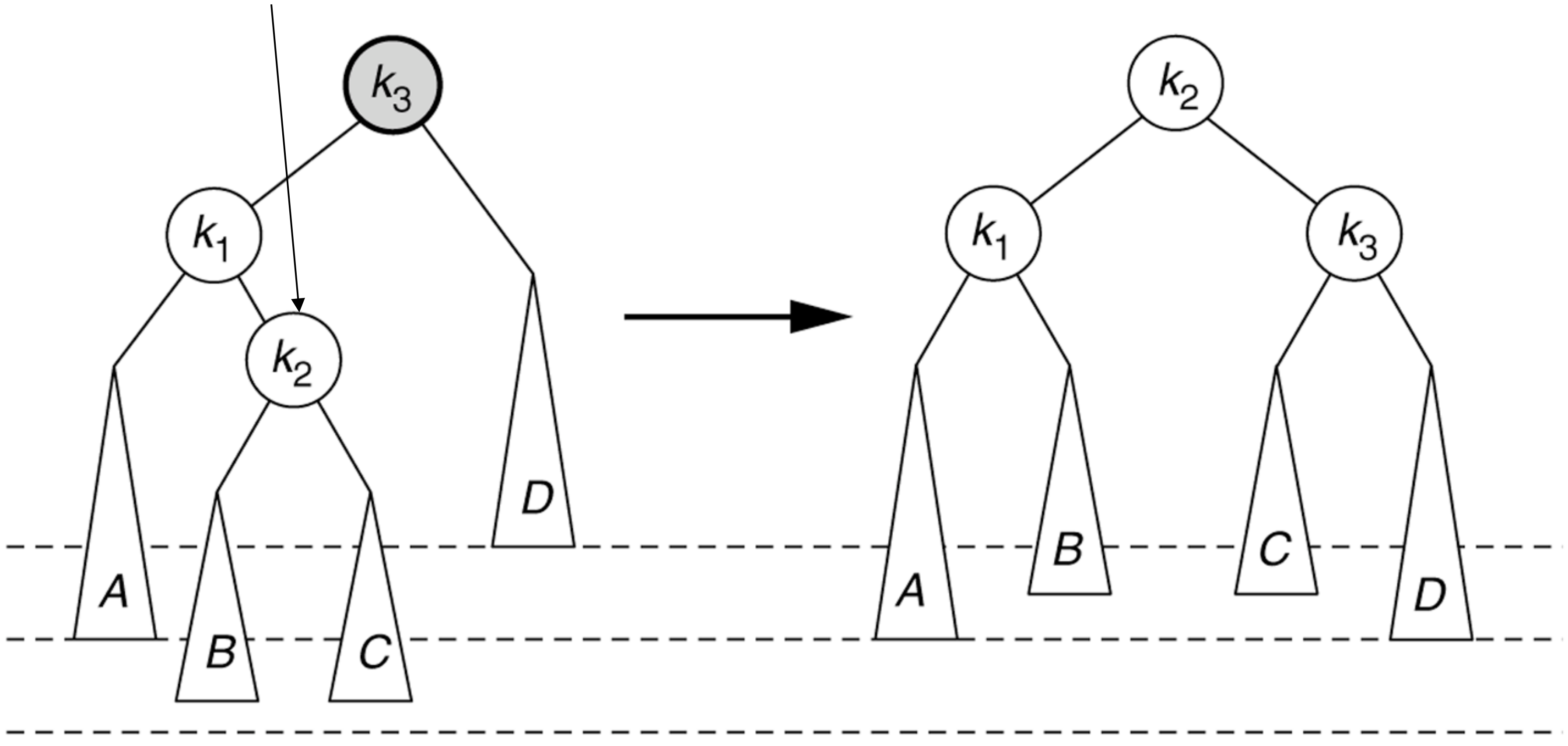
Figure 19.28

Single rotation **does not** fix case 2.



Left–right double rotation to fix case 2

Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)



(a) Before rotation

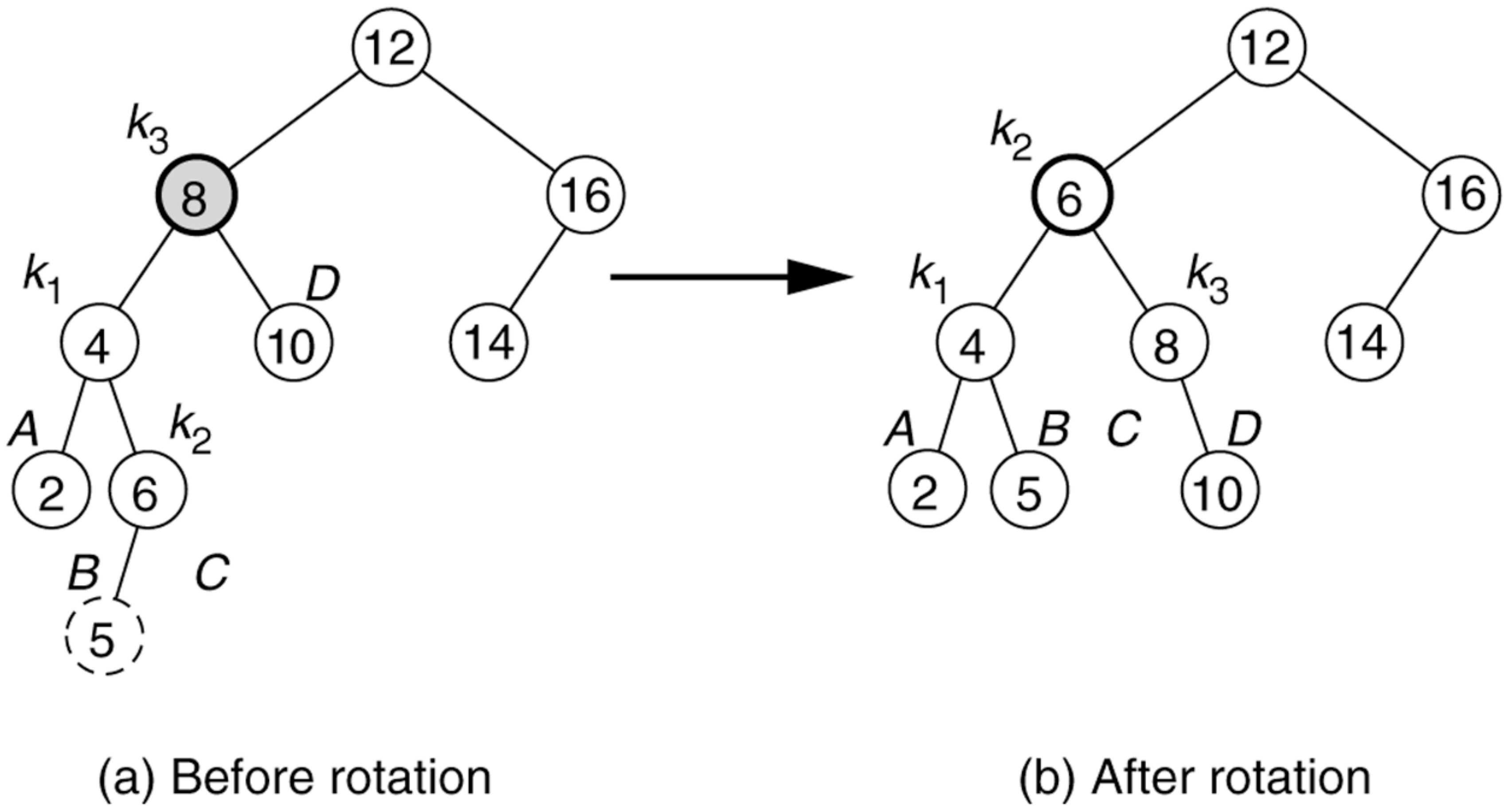
(b) After rotation

Left-Right Double Rotation

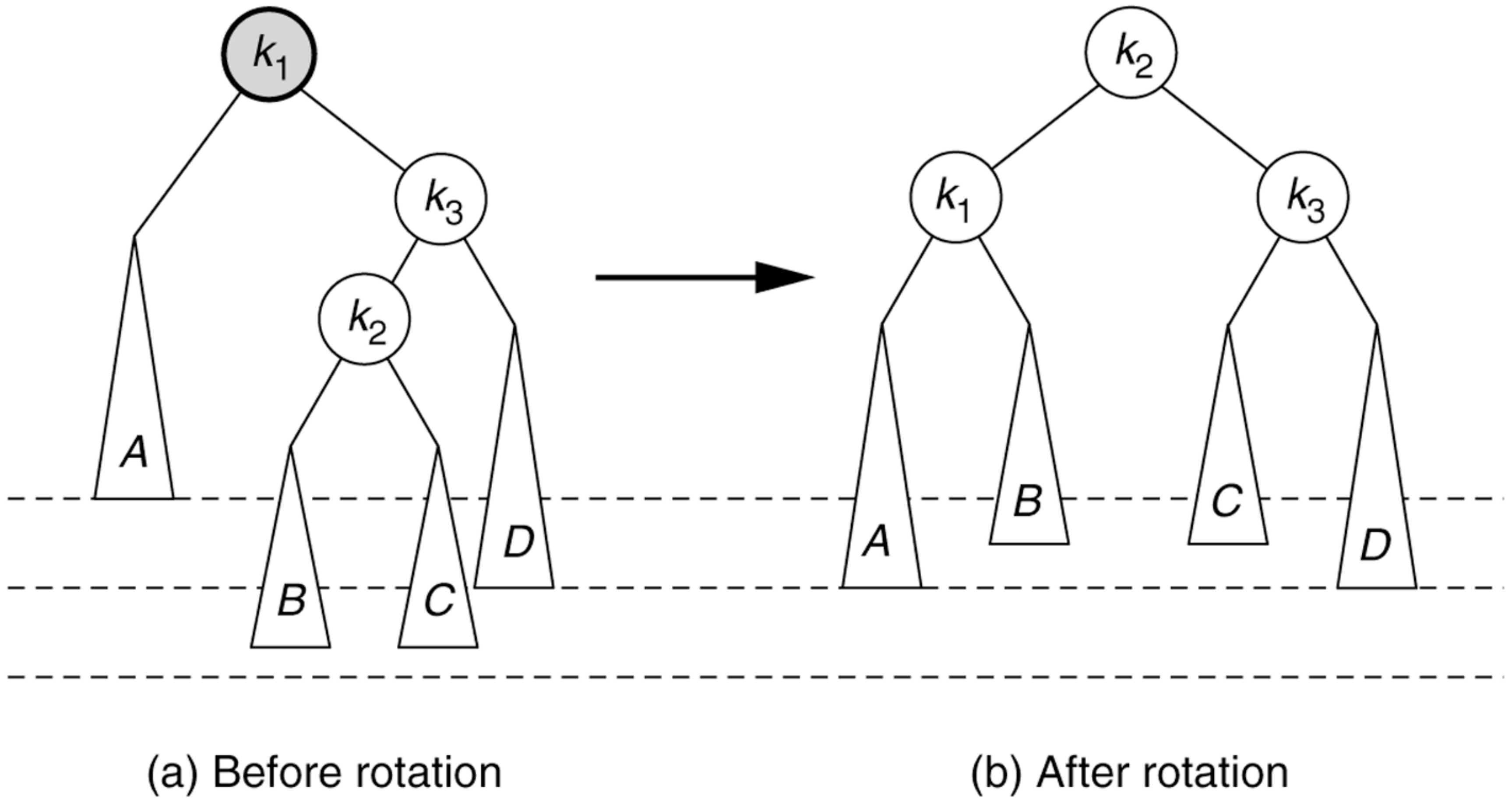
- A left-right double rotation is equivalent to a sequence of two single rotations:
  - 1st rotation on the original tree: a *left* rotation between X's left-child and grandchild
  - 2nd rotation on the new tree: a *right* rotation between X and its new left child.

Figure 19.30

Double rotation fixes AVL tree after the insertion of 5.



Right–Left double rotation to fix case 3.



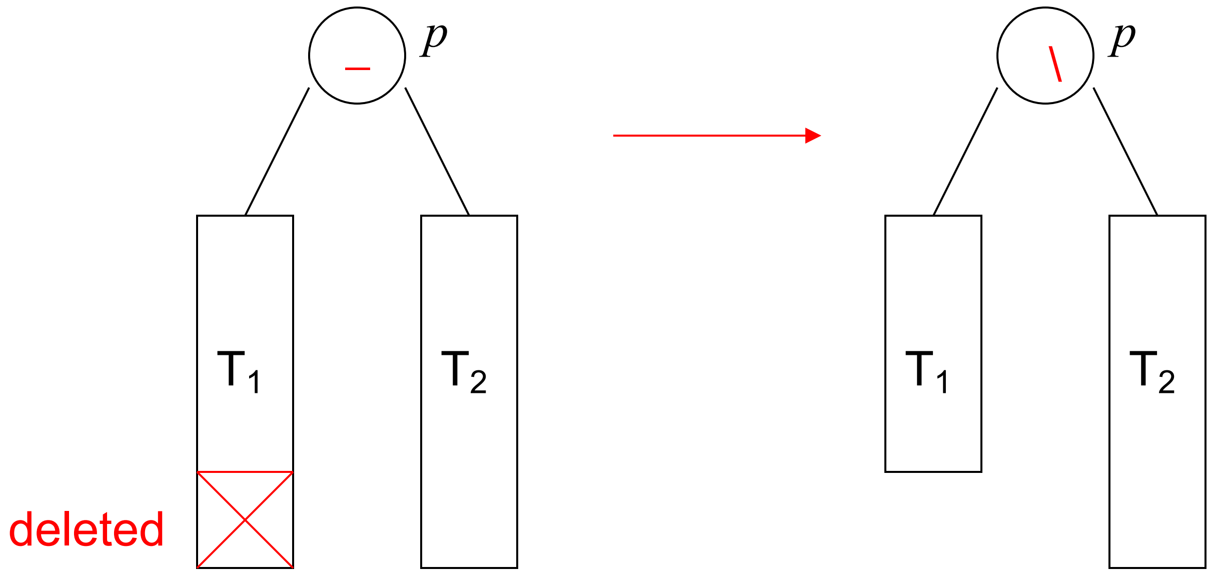
AVL Tree -- Deletion

- Deletion is more complicated.
- We may need more than one rebalance on the path from deleted node to root.
- Deletion is $O(\log_2 N)$

Case 1

Case 1: The current node *p* has balance factor equal.

– Change the balance factor of *p*.

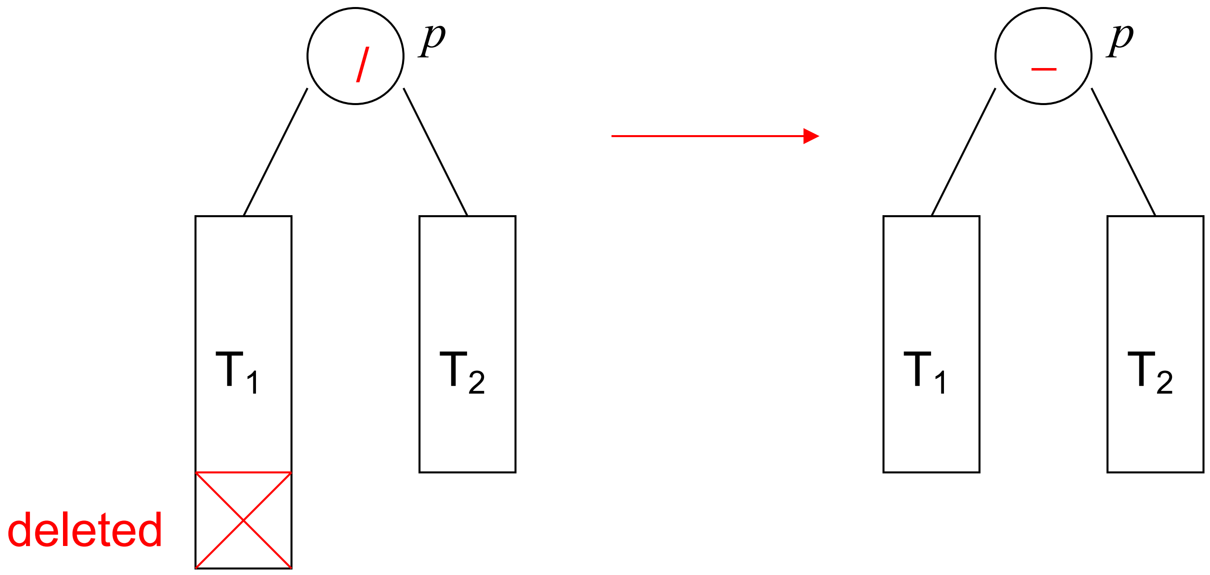


- No rotations
- Height unchanged

Case 2

Case 2: The balance factor of *p* is not equal and the taller subtree was shortened.

– Change the balance factor of *p* to equal

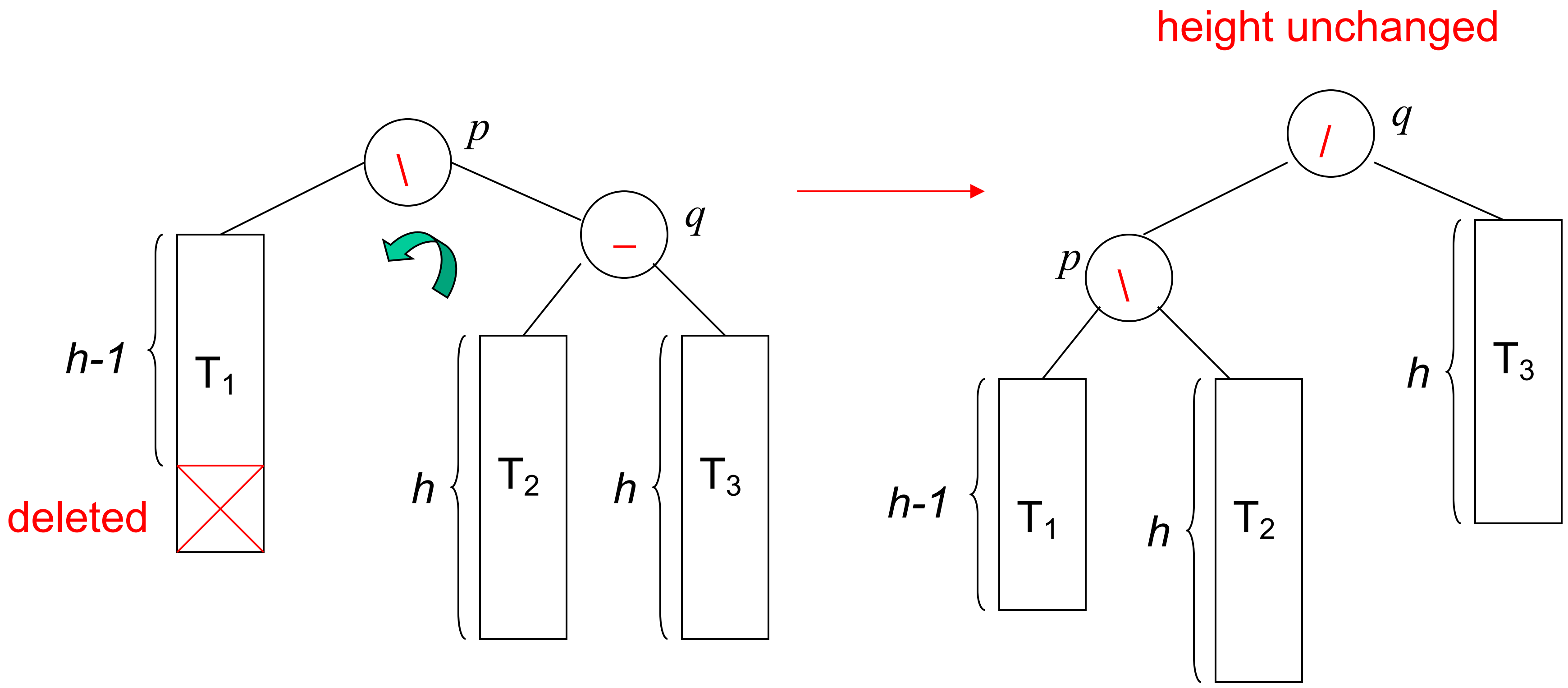


- No rotations
- Height reduced

Case 3a

Case 3*a*: The balance factor of *q* is equal.

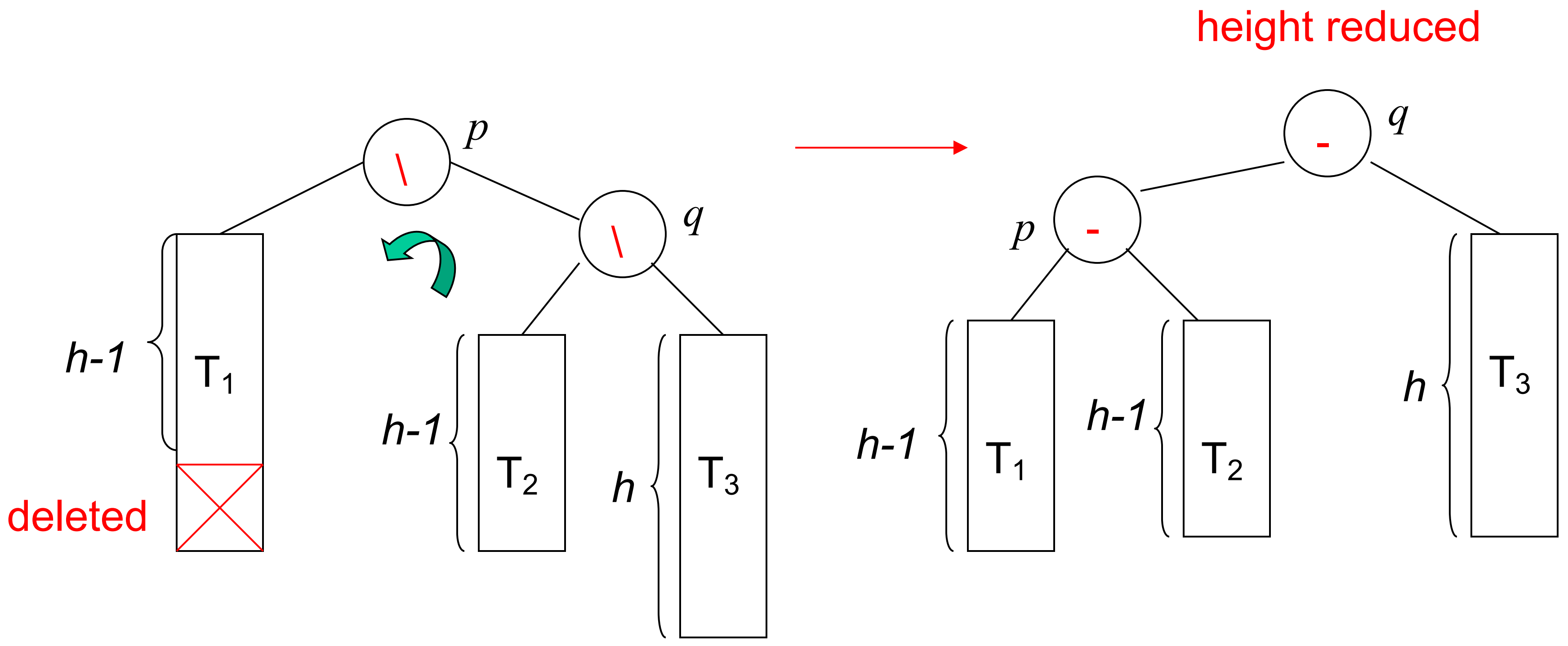
– Apply a single rotation



Case 3b

Case 3*b*: The balance factor of *q* is the same as that of p.

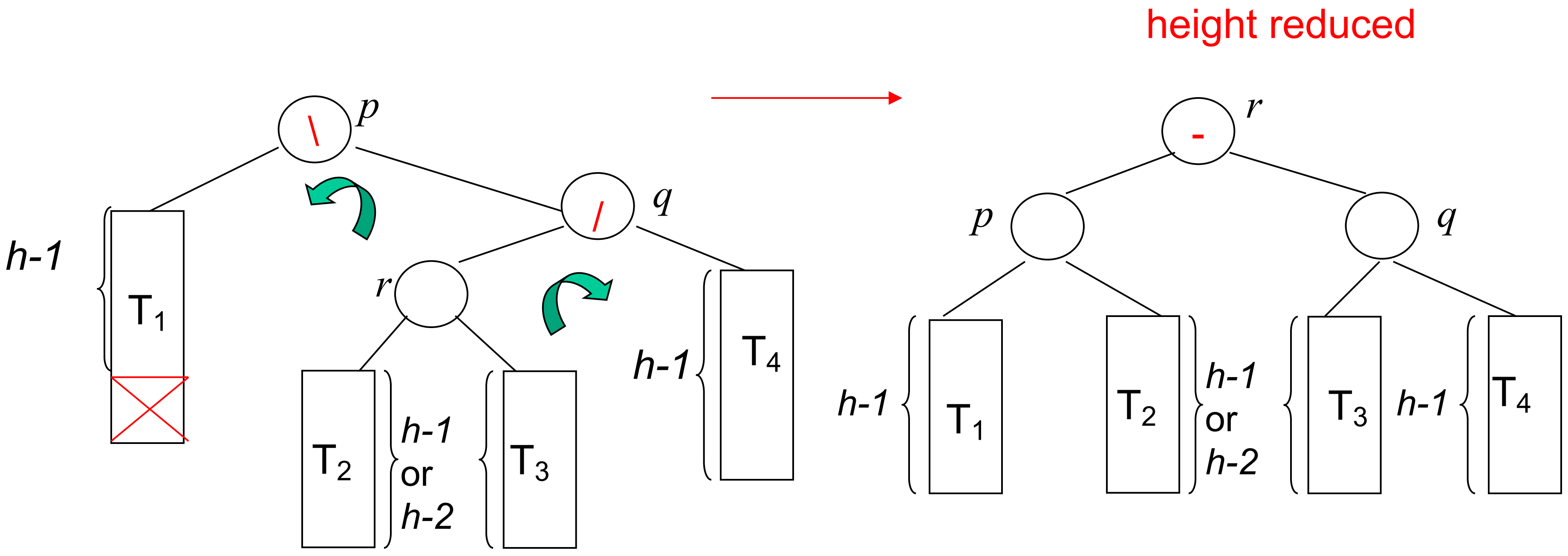
- Apply a single rotation
- Set the balance factors of *p* and *q* to equal



Case 3c

Case 3*c*: The balance factors of *p* and *q* are opposite.

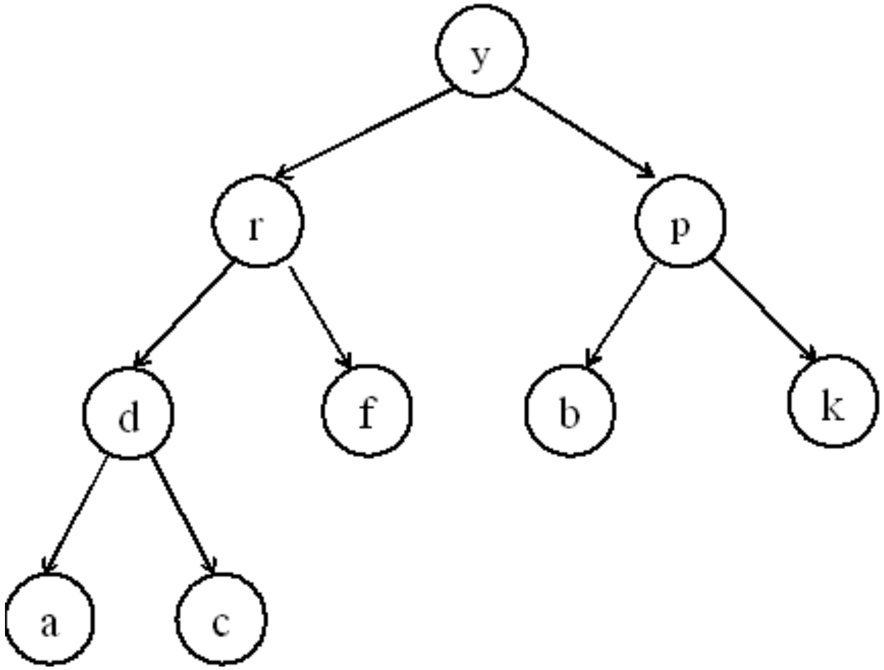
- Apply a double rotation
- set the balance factors of the new root to equal



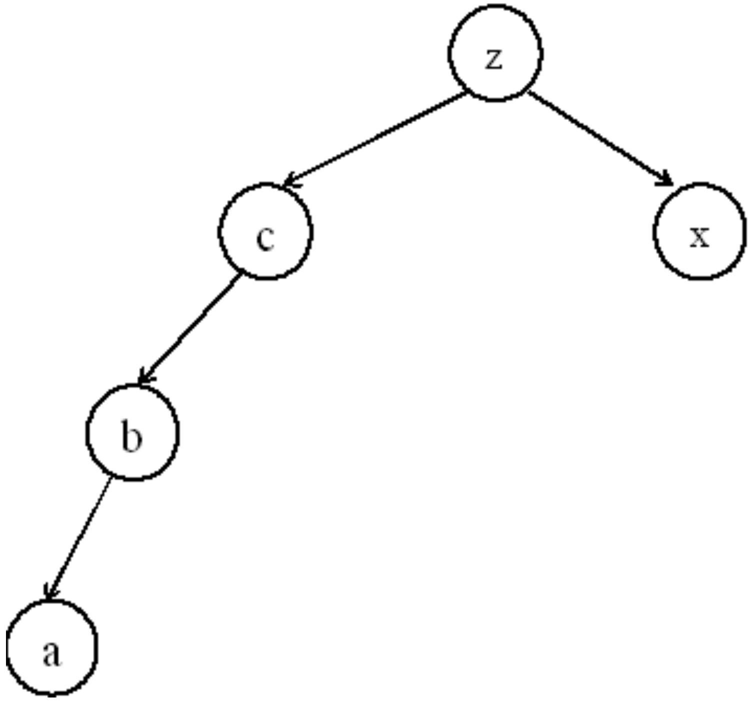
Heaps

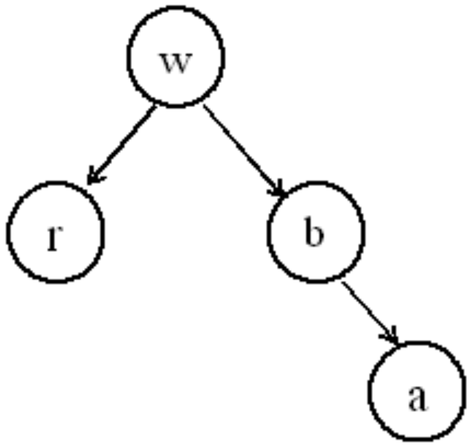
A heap is a binary tree with a key in each node such that

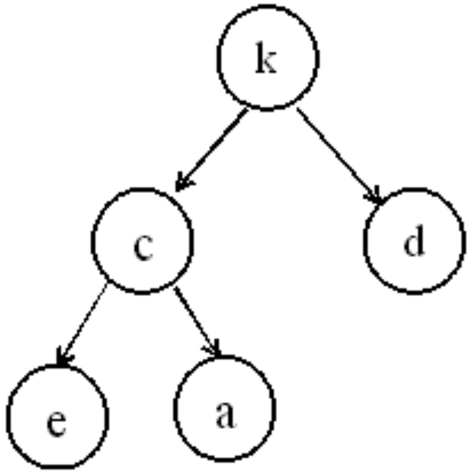
1. All the leaves of the tree are on adjacent levels.
2. All leaves on the lowest level occur to the left and all levels, except possibly the lowest, are filled.
3. The key in the root is at least as large as the keys in its children (if any), and the left and right subtrees (if they exist) are again heaps.



Examples (not heaps)

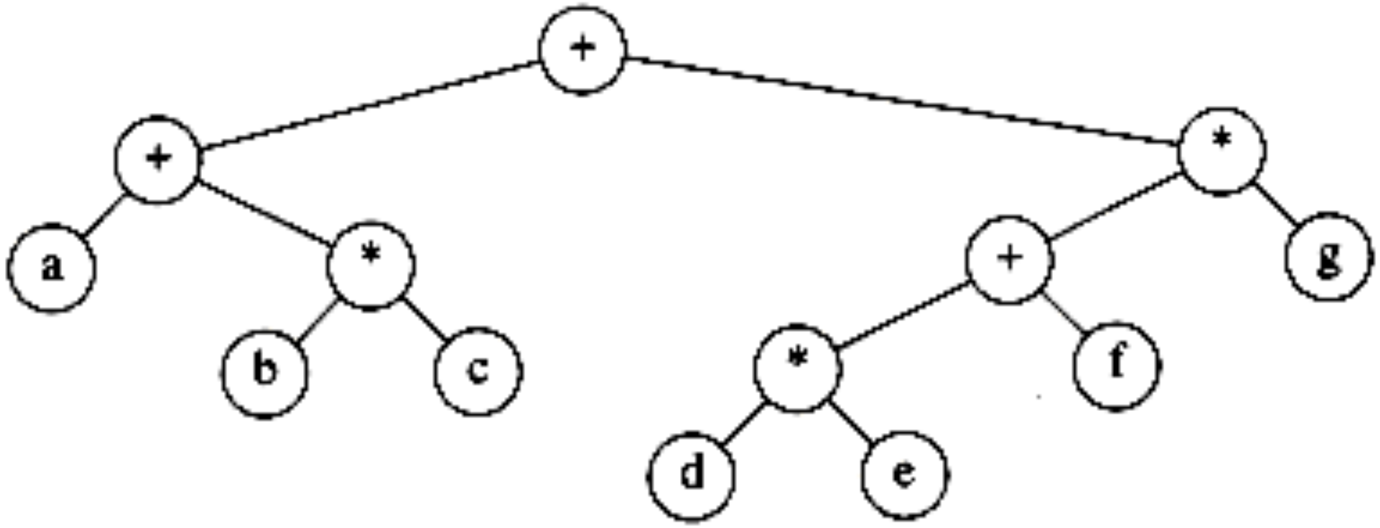




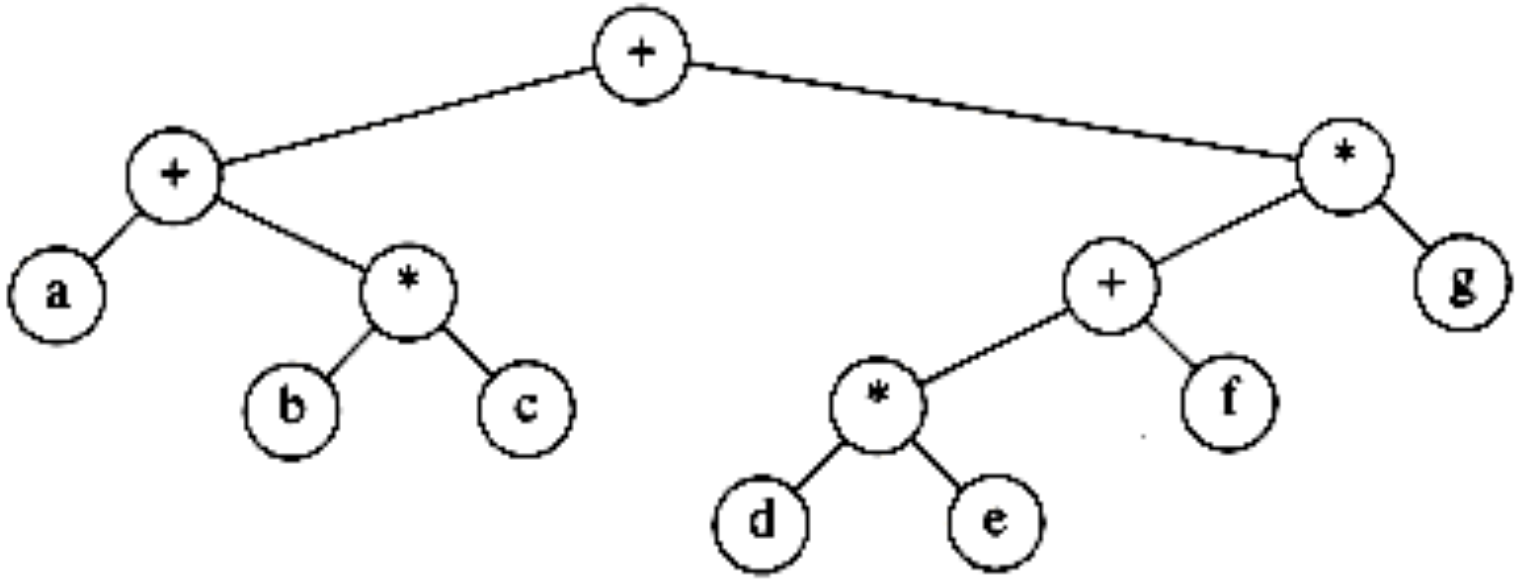


Expression Trees

- The leaves of an expression tree are operands, such as constants or variable names, and
- The other nodes contain operators.
- This tree is binary, because all of the operations are binary, and it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator.



Example



Expression tree for (a + (b * c)) + (((d * e) + f) * g)

Prefix expression: + + a * b c * + * d e f g

Postfix expression: a b c * + d e * f + g * +

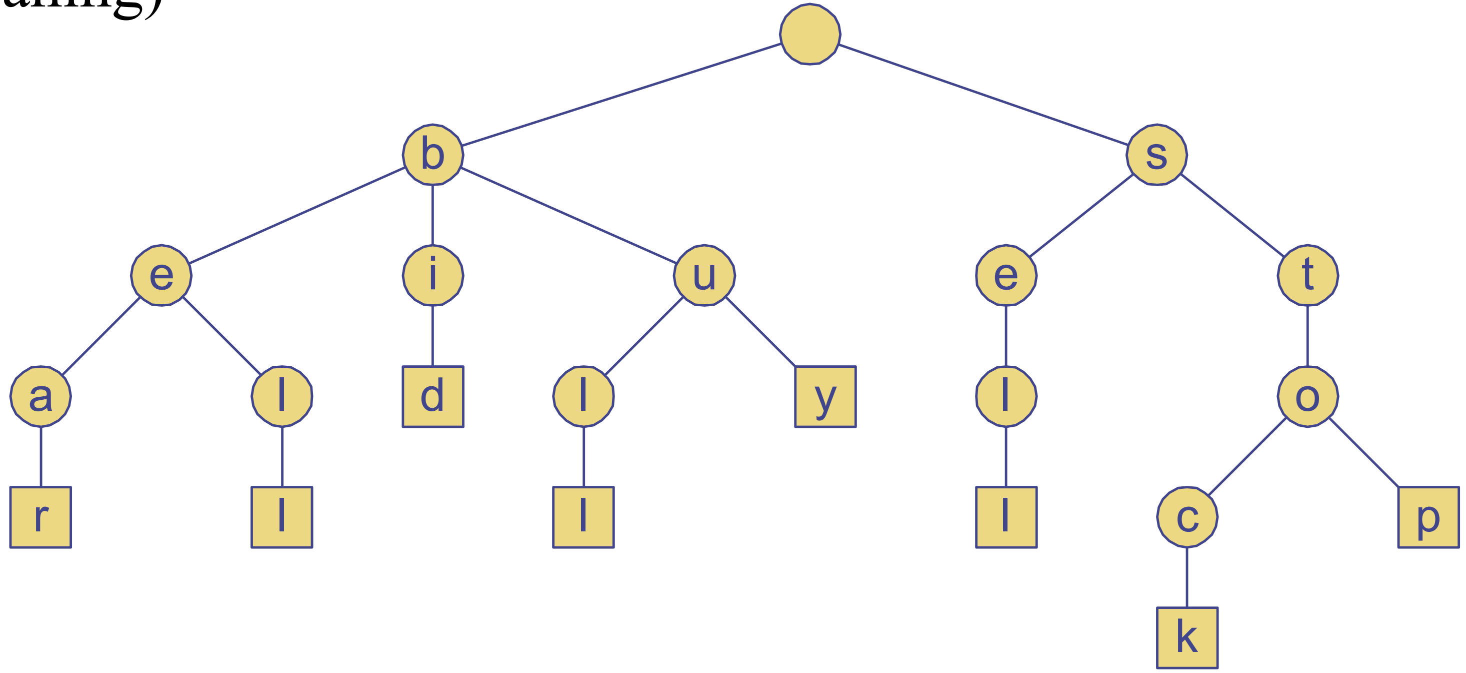
Infix expression: a + b * c + d * e + f * g

Trie

- The term trie comes from re**trie**val.
- A trie structure is used to store pieces of data that have a key and a value.

Key: used to identify data (i.e., word)

Value: holds additional data associated with the key (i.e., meaning)



Trie (cont.)

- Trie is used for
  - storing words in a dictionary,
  - fast dictionary look-up,
  - less memory space to store words in a dictionary.
- Each node stores a character and pointers to other nodes.
- Each node should have one pointer for each letter in the alphabet.
- Leaf nodes store the values.

Coding Trees

- Used to compress data.
- For each character in the data, a binary code is computed, and the character is replaced with its corresponding code.
- Frequently occurring characters have short codes.

Example: BABACABA → 8 chars = 64 bits

Character	Code
A	00
B	01
C	10

8 * 2 = 16 bits

Character	Frequency	Code
A	4	0
B	3	10
C	1	11

4*1 + 3*2 + 1*2 = 12 bits

Huffman Coding Tree

- Sort characters w.r.t. frequencies in descending order.
- Take the two characters having the lowest frequency and connect them via an internal node.
- Compute the frequency of the internal node by adding the frequencies of the connected characters.
- Remove the two characters from the list.
- Insert the internal node into the list.
- Sort the list w.r.t. frequency values in descending order and repeat the above steps until only one element left in the list.
- The remaining element forms the root of the tree.
- Start from the root node, assign 0 to each left branch, and 1 to each right branch.
- To generate the codes for the characters, start from the root node, take the label given to each branch until the leaf node.

Example

For the characters and their frequency values given in the table generate binary codes using Huffman coding tree.

Character	**Frequency**
A	20
B	14
C	10
D	8
E	7

Space requirement = 59 bytes = 59 * 8 = 472 bits

Example (cont.)

Character	Frequency
A	20
B	14
C	10
D	8
E	7

D + E : 15

Character	Frequency
A	20
D + E	15
B	14
C	10

B + C : 24

Example (cont.)

Character	**Frequency**
B + C	24
A	20
D + E	15

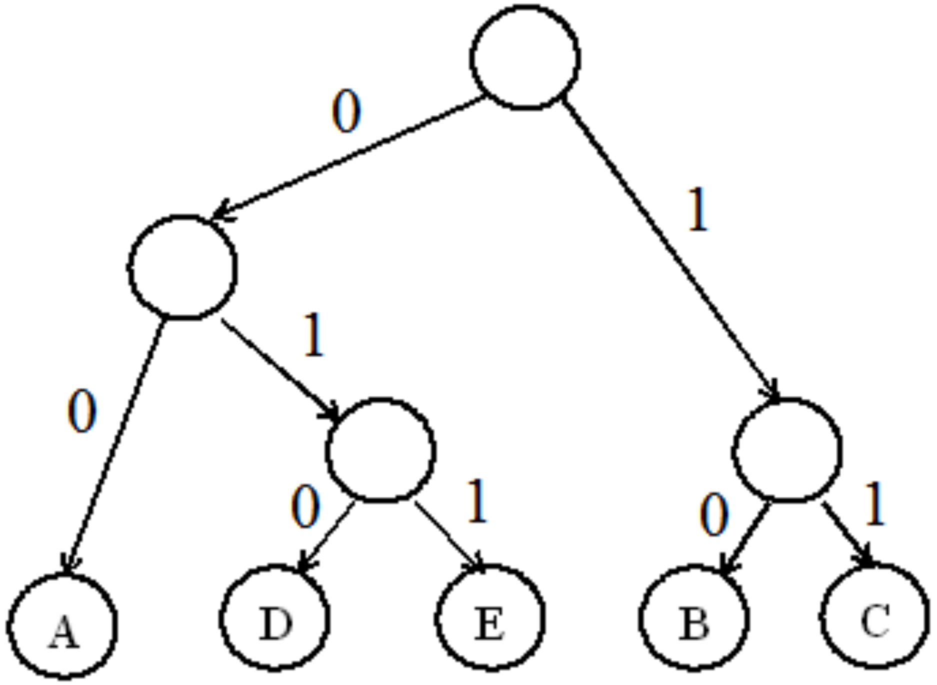
A + (D + E) : 35

Character	**Frequency**
A +(D + E)	35
B + C	24

(A + (D + E)) + (B+C) : 59

Example (cont.)

(A + (D + E)) + (B+C) : 59



Character	Code
A	00
B	10
C	11
D	010
E	011

Space required = 20*2 + 14 *2 + 10*2 + 8*3 + 7*3=133 bits

Shannon-Fano Coding

1. Sort the characters w.r.t their probabilities in descending order.

Probability = Frequency / Total frequency

2. Divide characters into 2 sets S1 and S2 such that total probability for S1 and S2 are equal or near to each other.
3. Assign 1 to S1, 0 to S2.
4. Continue to divide each set into 2 sets until each set has 1 element.

Example

- Compute codes by using Shannon-Fano coding for the characters with the given probability values.

S = { a: 0.10, b: 0.05, c: 0.20, d: 0.15, e: 0.15, f: 0.25, g: 0.10}

f: 0.25	1 1
c: 0.20	1 0
d: 0.15	0 1 1
e: 0.15	0 1 0
a: 0.10	0 0 1
g: 0.10	0 0 0 1
b: 0.05	0 0 0 0