
# Search Trees

Paolo Costa

*paolo.costa@polimi.it*

# Applications: Databases

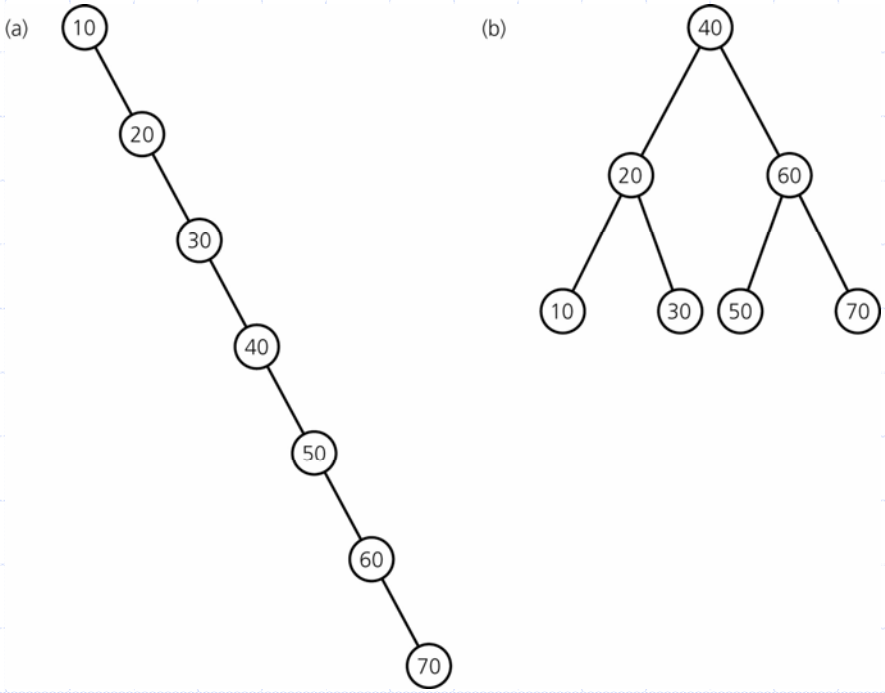
- It is not uncommon for a database to contain millions of records requiring many gigabytes of storage.
  - TELSTRA, an Australian telecommunications company, maintains a customer billing database with 51 billion rows (yes, billion) and 4.2 terabytes of data.
  - For example, searching an non-indexed and unsorted database containing *n* key values will have a worst case running time of *O(n)*;
  - Data indexed with a b-tree, the same operation will run in *O(log n)*.

    A search for a single key on a set of one million keys (1,000,000),
    - a linear search will require at most 1,000,000 comparisons.
    - If the same data is indexed with a b-tree of minimum degree 10,

      114 comparisons will be required in the worst case.

# Why Balance is Important

- Searches in unbalanced tree can be O(n)



# Balanced Search Trees

- The efficiency of the binary search tree implementation is related to the tree's height
  - Height of a binary search tree of n items
    - Maximum: n
    - Minimum: $\lceil \log_2(n + 1) \rceil$
- Height of a binary search tree is sensitive to the order of insertions and deletions
- Variations of the binary search tree
  - Can retain their balance despite insertions and deletions

# Tree Balancing

**FIGURE 11.3**
Unbalanced Tree Before Rotation

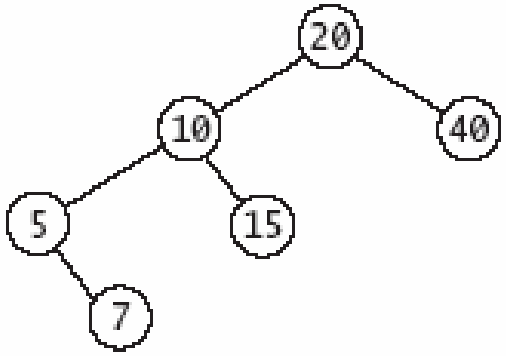



**FIGURE 11.4**
Right Rotation

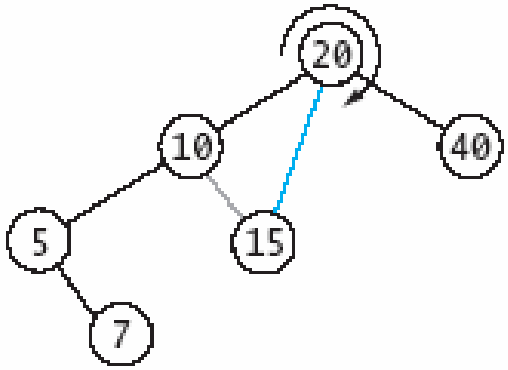
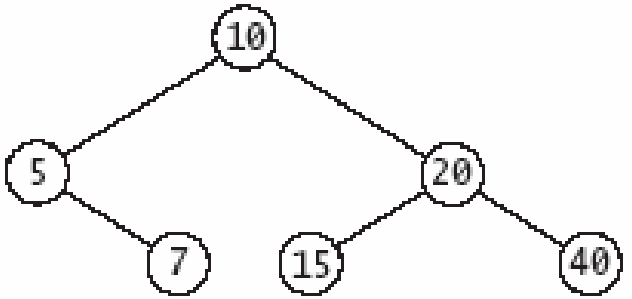



**FIGURE 11.5**
More Balanced Tree After Rotation



# 2-3 Tree (1)

A 2-3 Tree has the following properties:

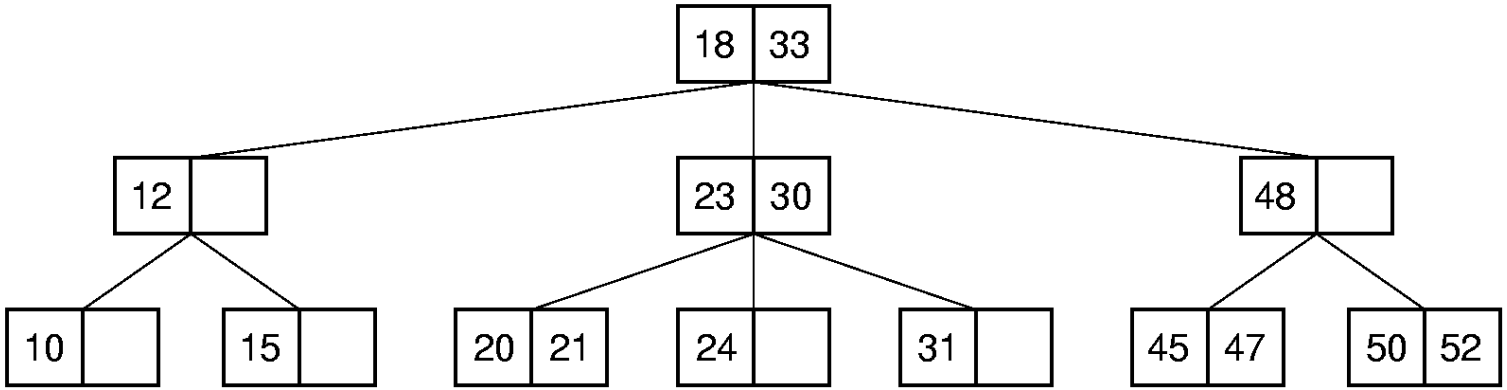
1. A node contains one or two keys
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
3. All leaves are at the same level in the tree, so the tree is always height balanced.

- Searching a 2-3 tree Searching a 2-3 tree is as efficient as searching the shortest binary search tree
  - Searching a 2-3 tree is $O(\log_2 n)$

# 2-3 Tree

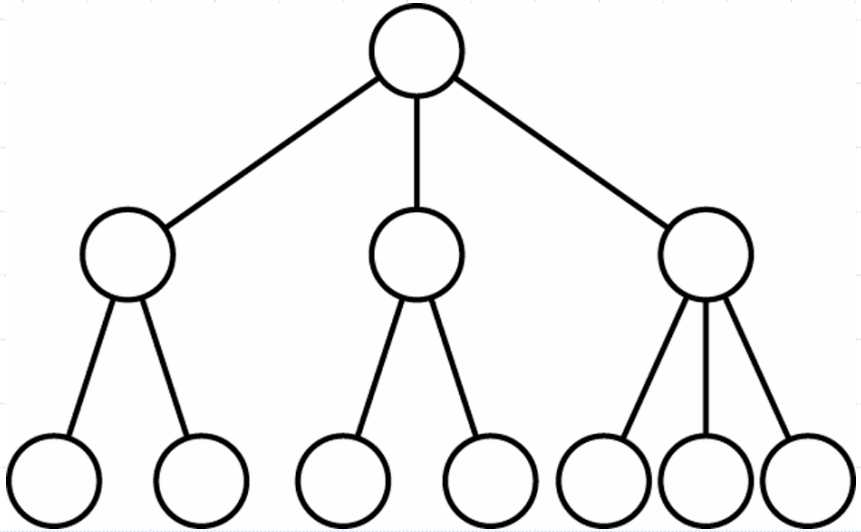
- A 2-3 Tree has the following properties:
  1. A node contains one or two keys
  2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys)
  3. All leaves are at the same level in the tree, so the tree is always height balanced
- Searching a 2-3 tree is O(log2n)

# 2-3 Tree




- The advantage of the 2-3 Tree over the BST is that it can be updated at low cost

# Features

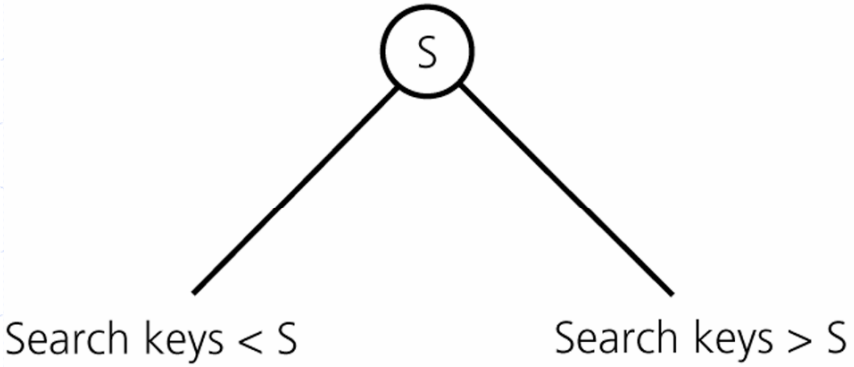


- each internal node has either 2 or 3 children
- all leaves are at the same level

# 2-3 Trees with Ordered Nodes

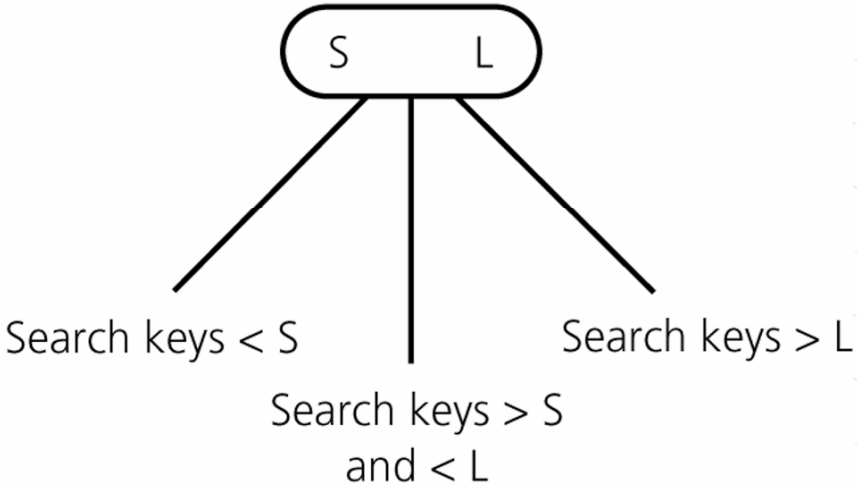
**2-node**

(a)



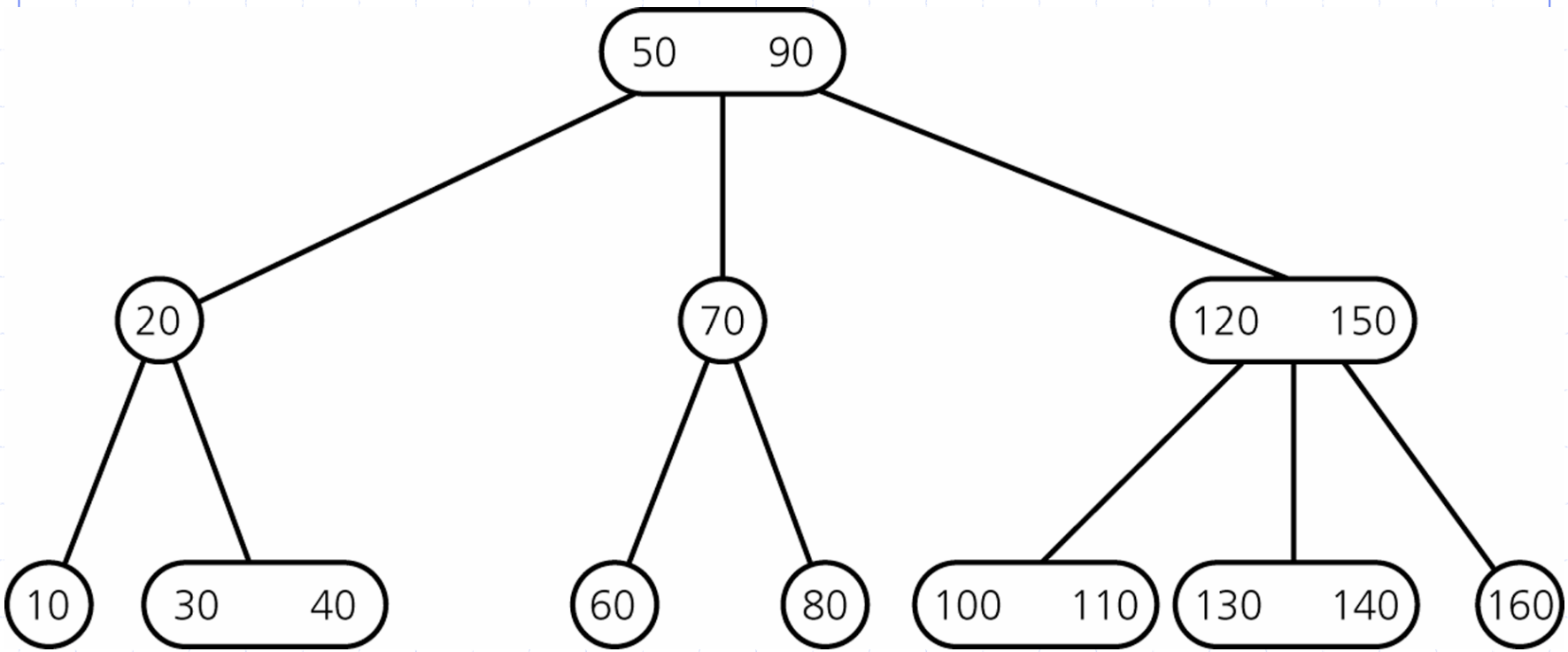

**3-node**

(b)

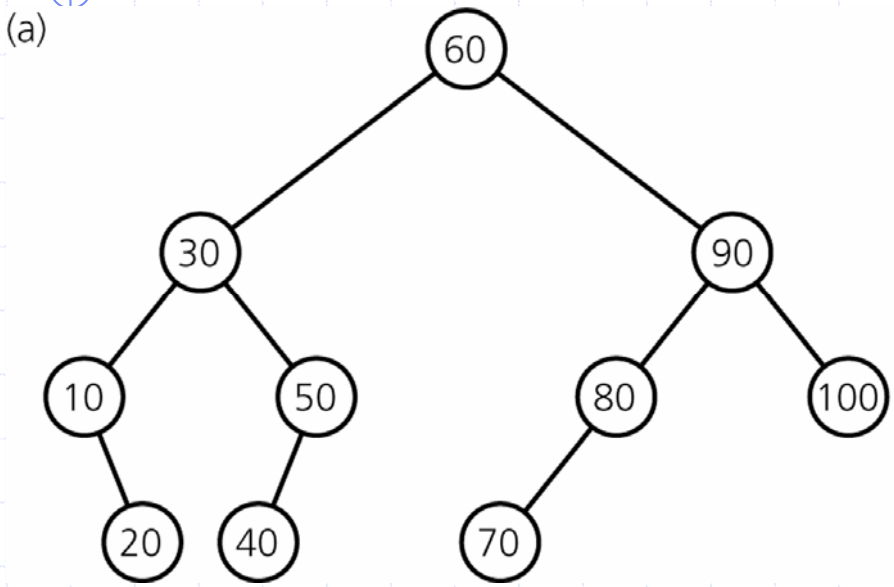


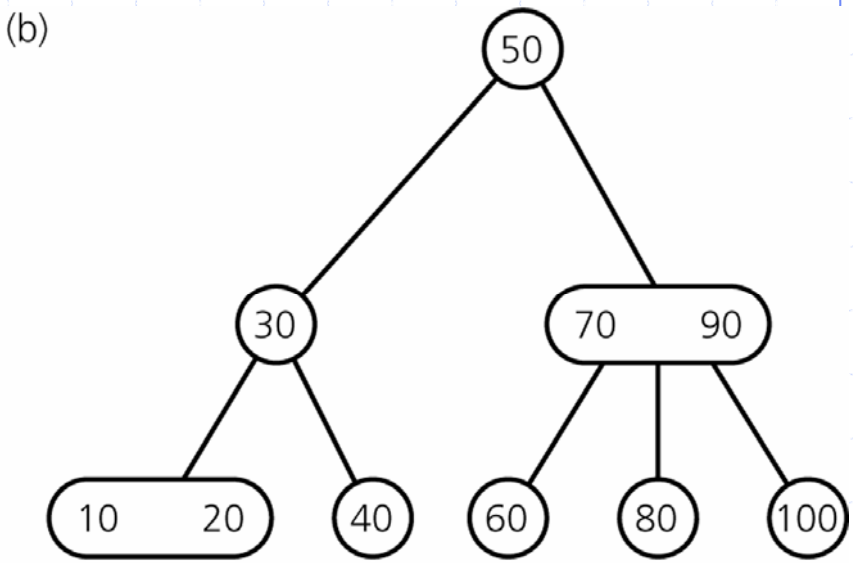

- leaf node can be either a 2-node or a 3-node

# Example of 2-3 Tree



# What did we gain?



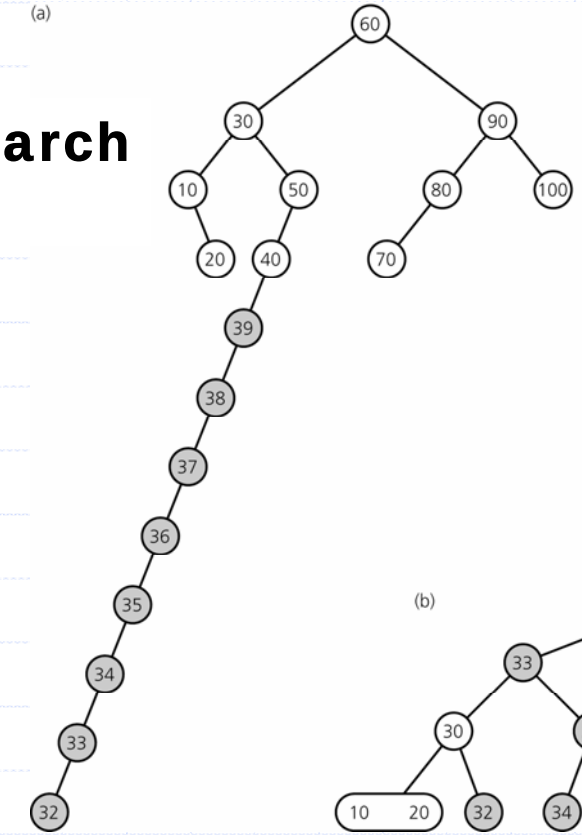




**What is the time efficiency of searching for an item?**

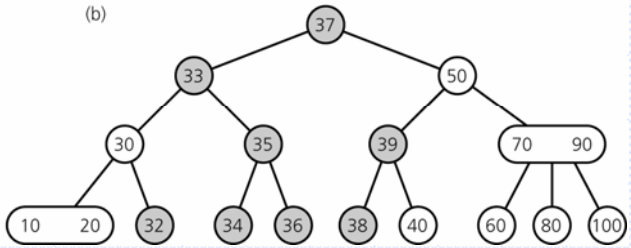
# Gain: Ease of Keeping the Tree Balanced

both trees after inserting items 39, 38, ... 32

**Binary Search Tree**

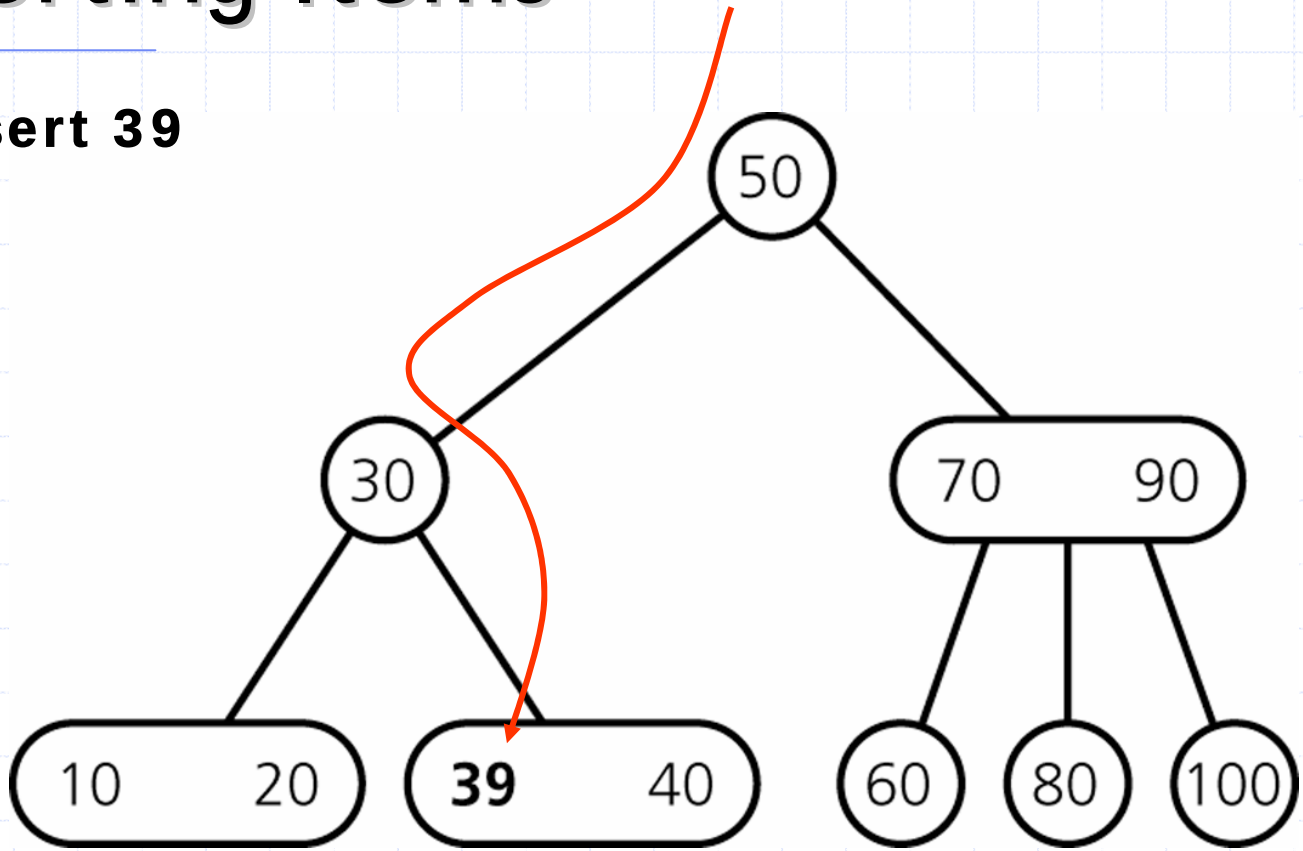



**2-3 Tree**



# Inserting Items

**Insert 39**

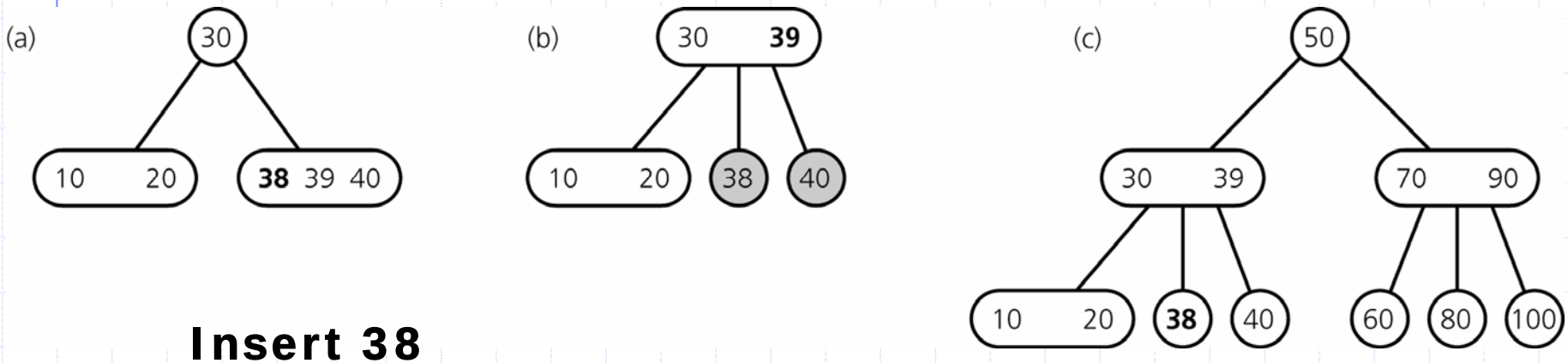


# Inserting Items

insert in leaf

divide leaf
and move middle
value up to parent

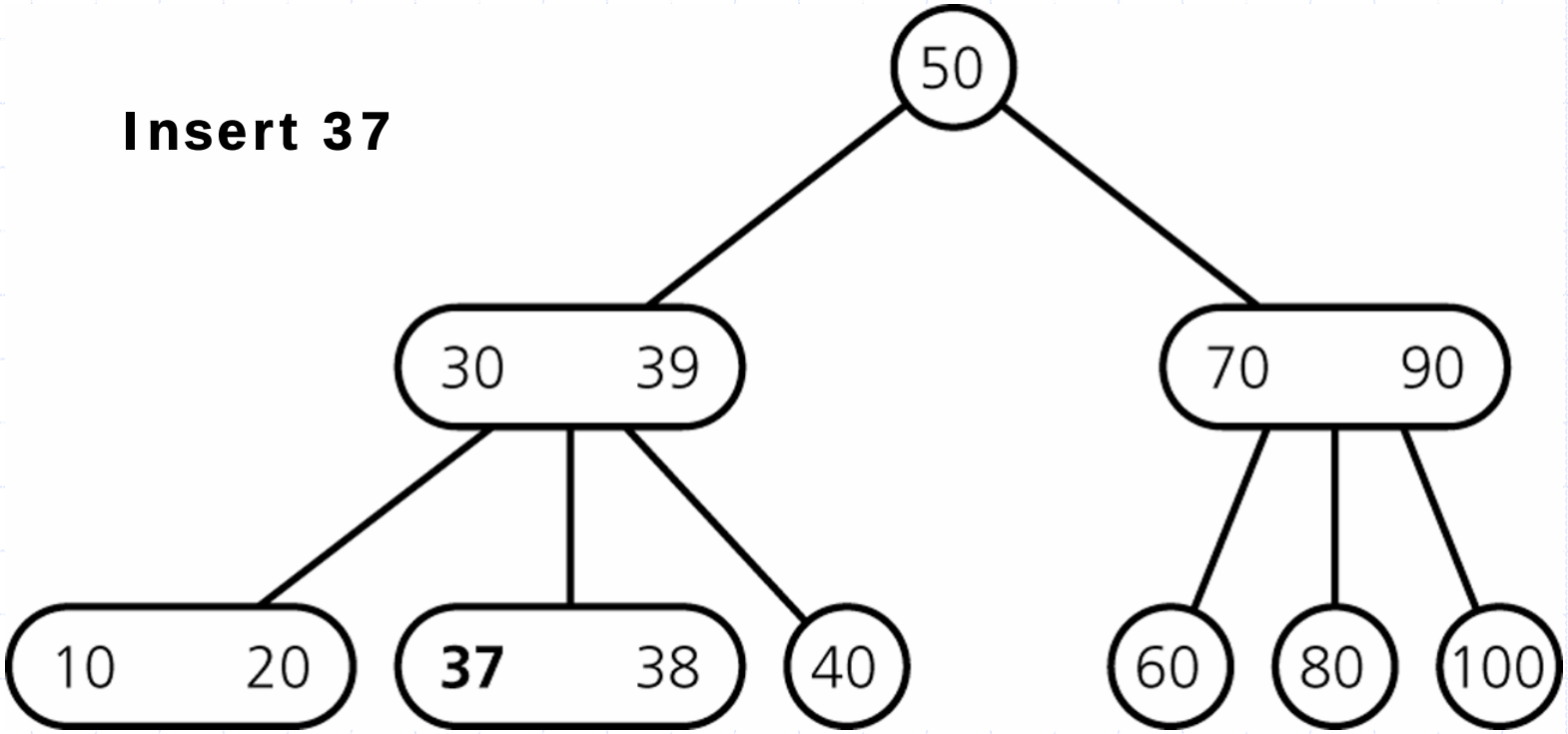
result




**Insert 38**

# Inserting Items

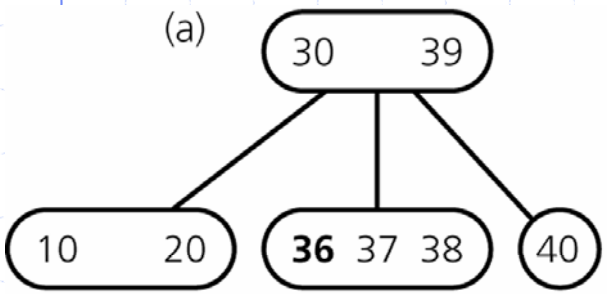
**Insert 37**



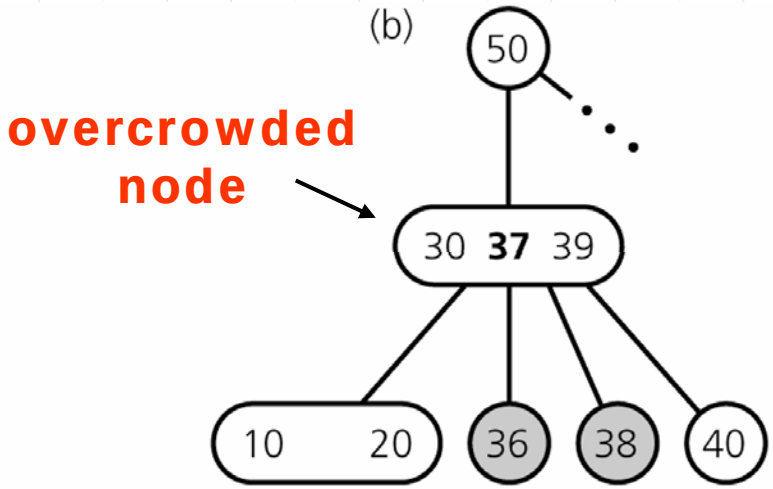
# Inserting Items

insert in leaf

divide leaf
and move middle
value up to parent




**Insert 36**

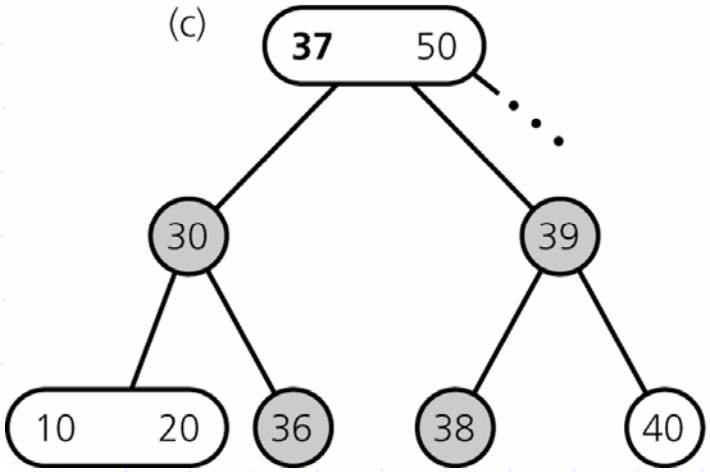


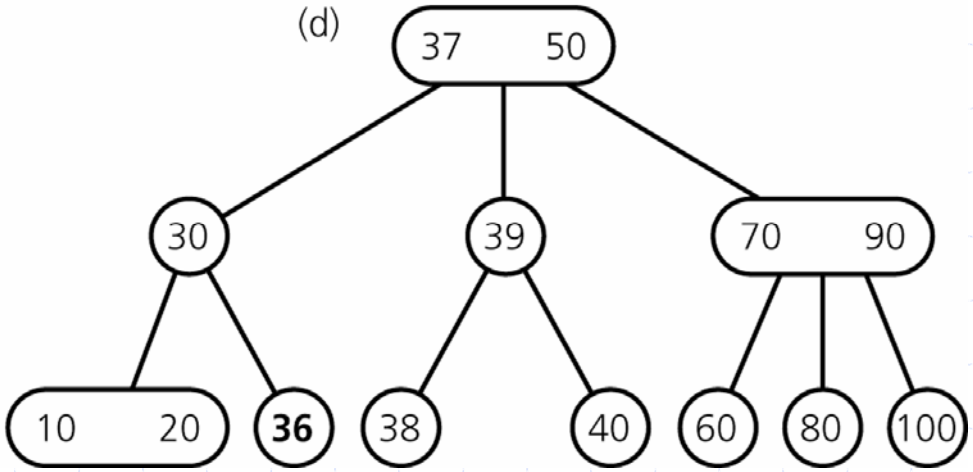
# Inserting Items

**... still inserting 36**

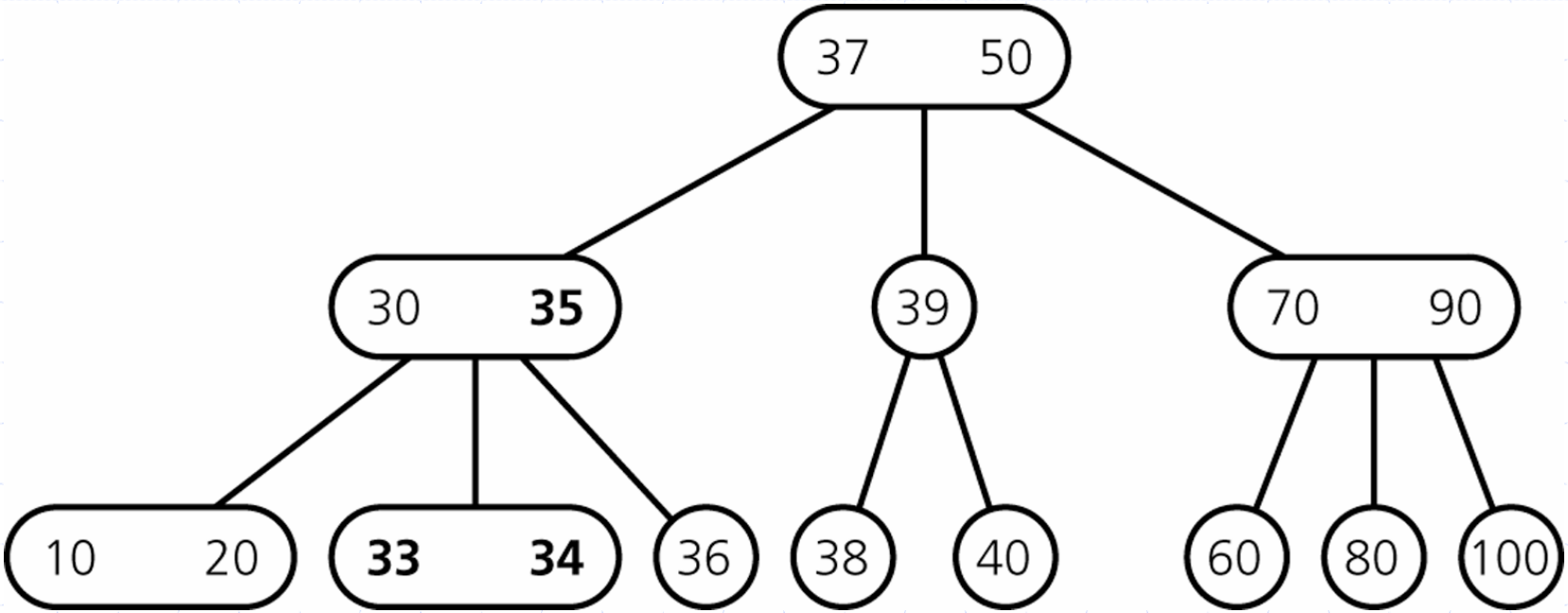
divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

result



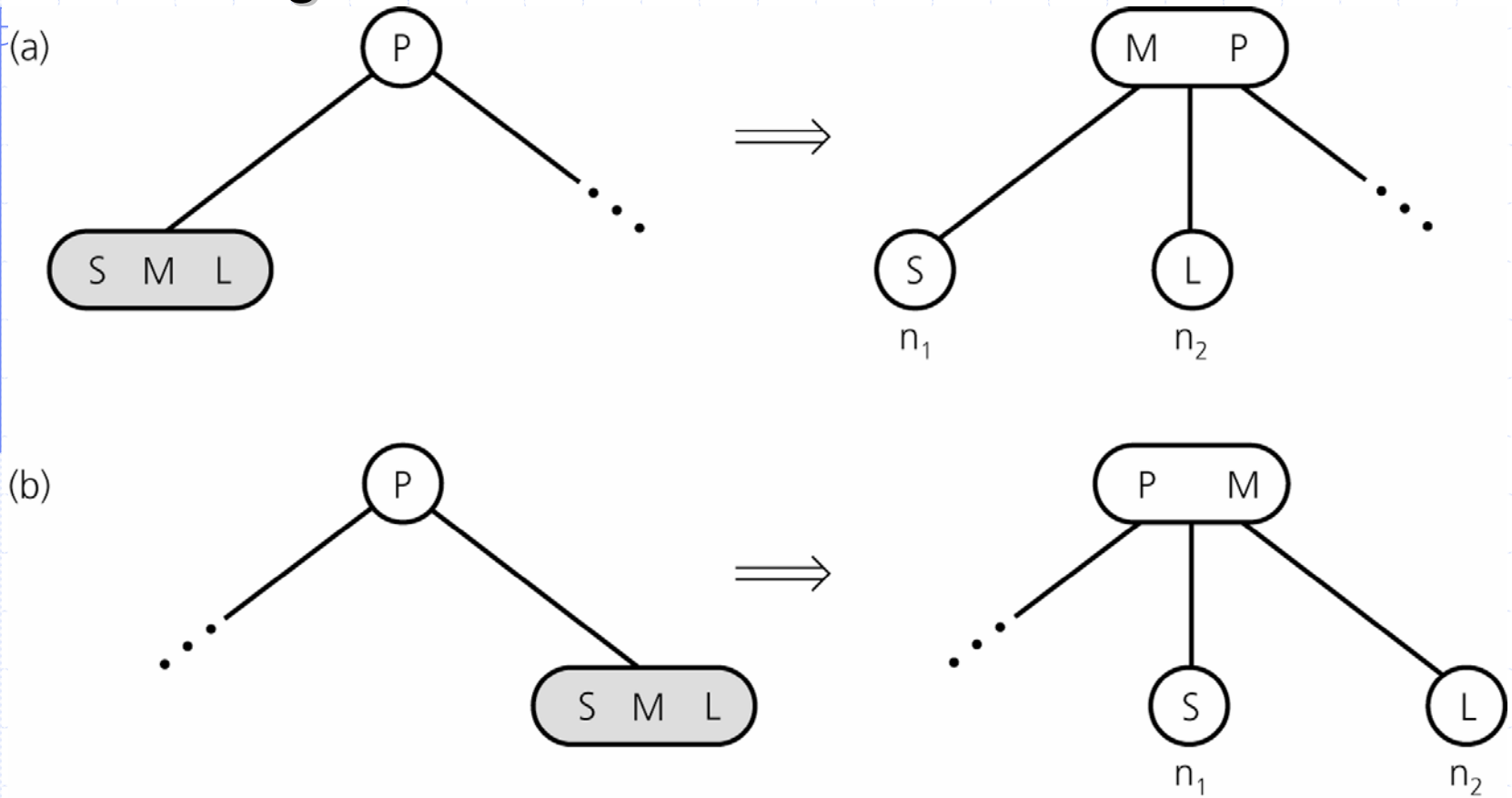



# Inserting Items

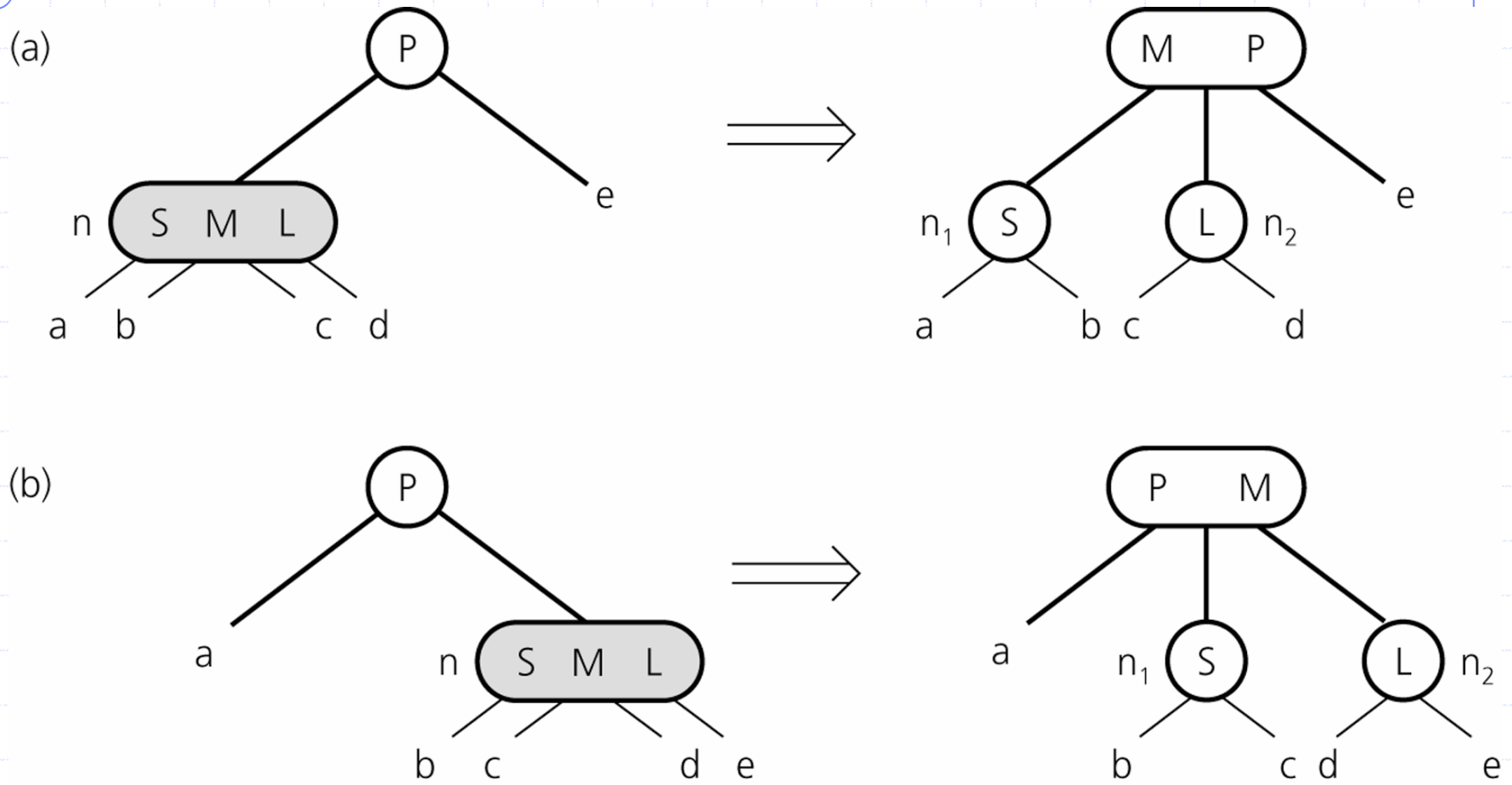



**After Insertion of 35, 34, 33**

# Inserting so far

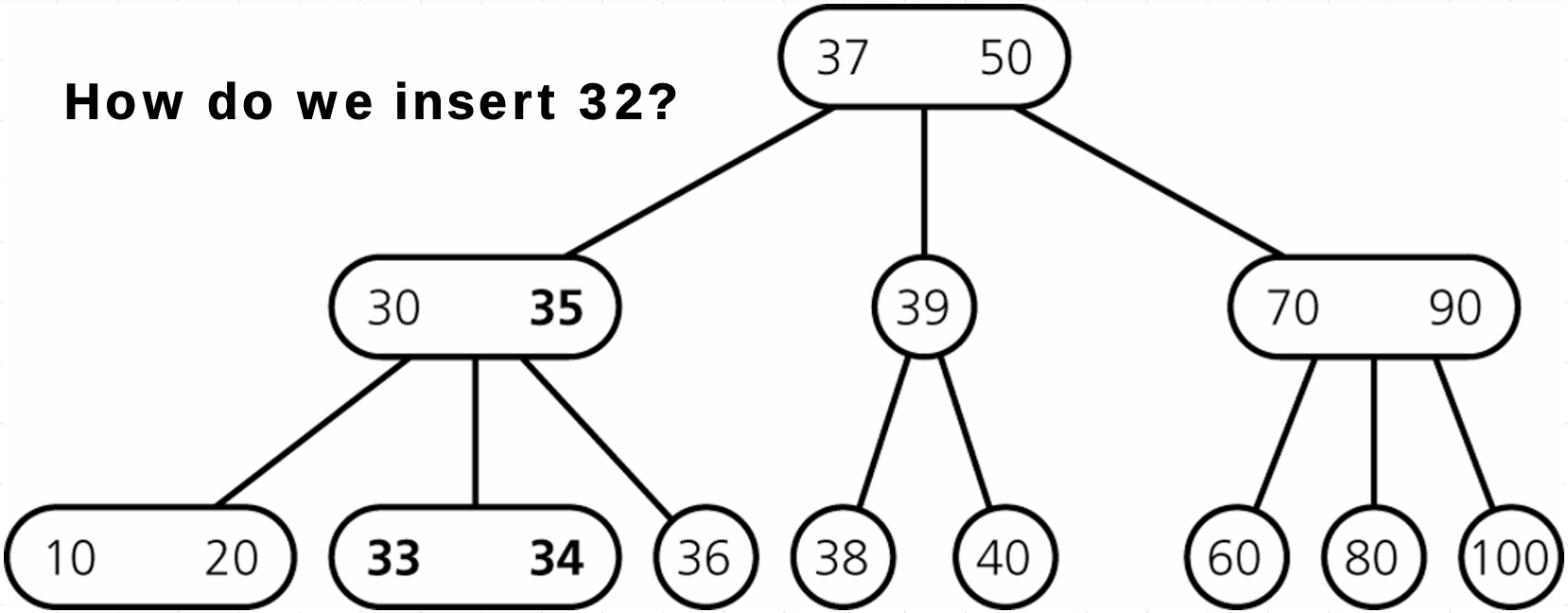


# Inserting so far



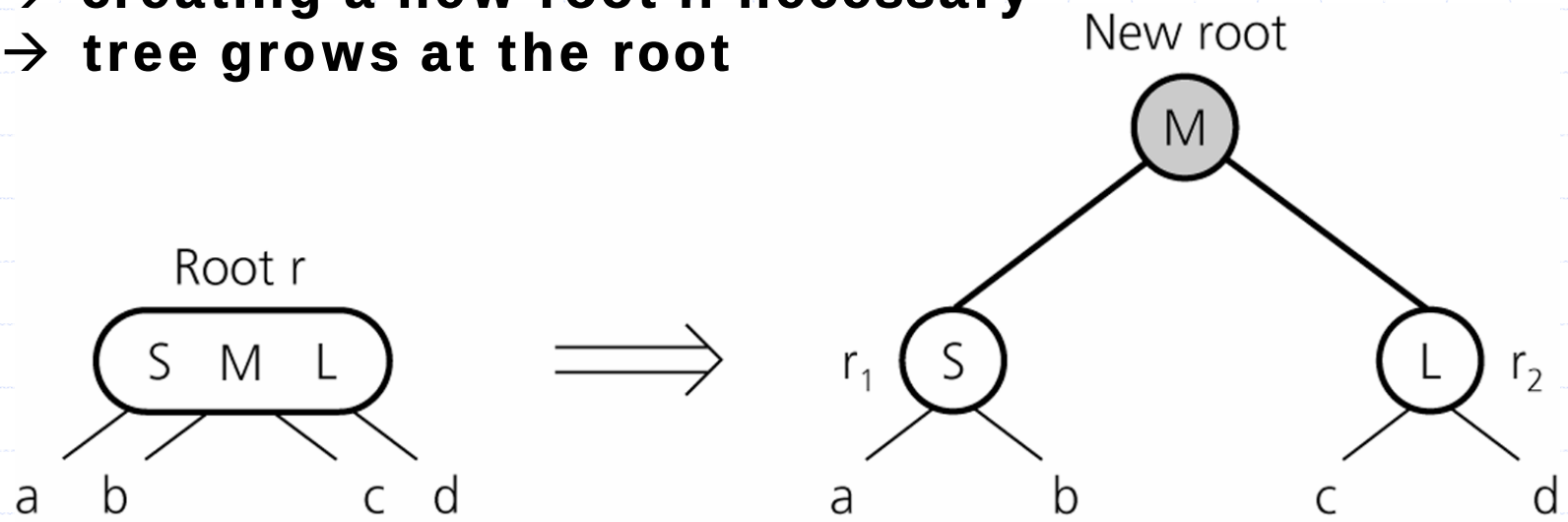
# Inserting Items

**How do we insert 32?**

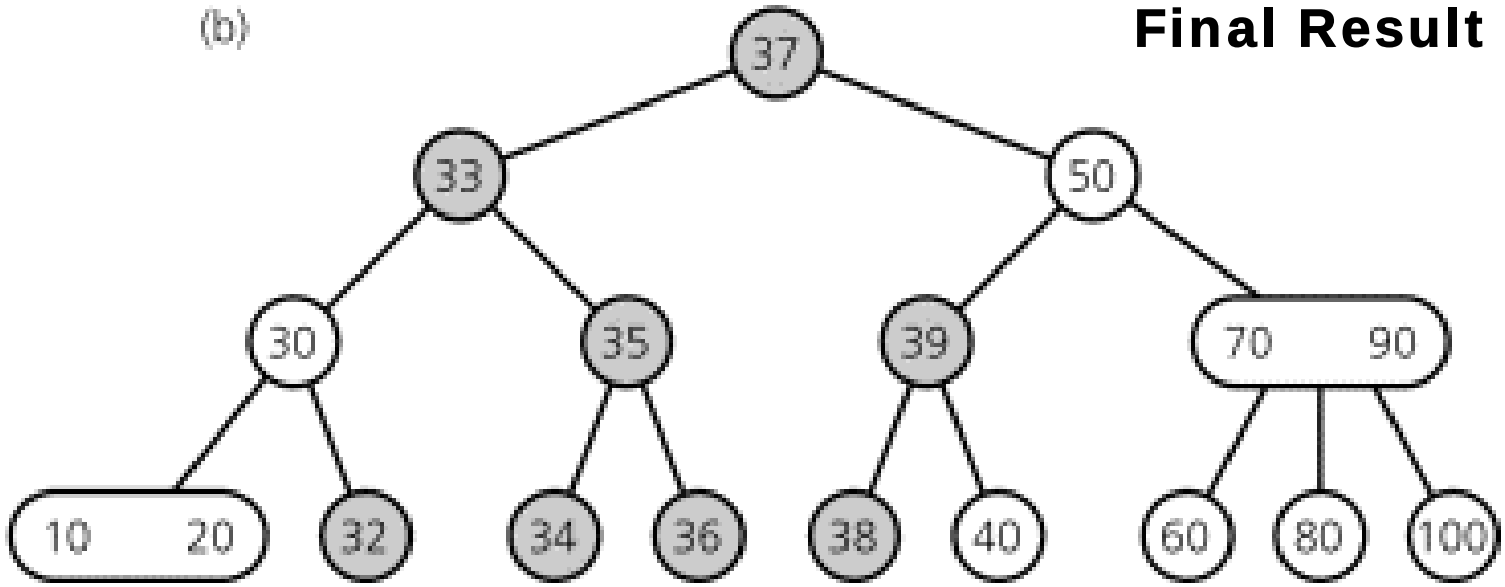


# Inserting Items

→ **creating a new root if necessary**
→ **tree grows at the root**



# Inserting Items



# Operations of 2-3 Trees

**all operations have time complexity of log n**

# B-Trees (1)

The B-Tree is an extension of the 2-3 Tree.

The B-Tree is now **the** standard file organization for applications requiring insertion, deletion, and key range searches.

# B-Trees (2)

1. B-Trees are always balanced.
2. B-Trees keep similar-valued records together on a disk page, which takes advantage of locality of reference.
3. B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

# B-Tree Definition

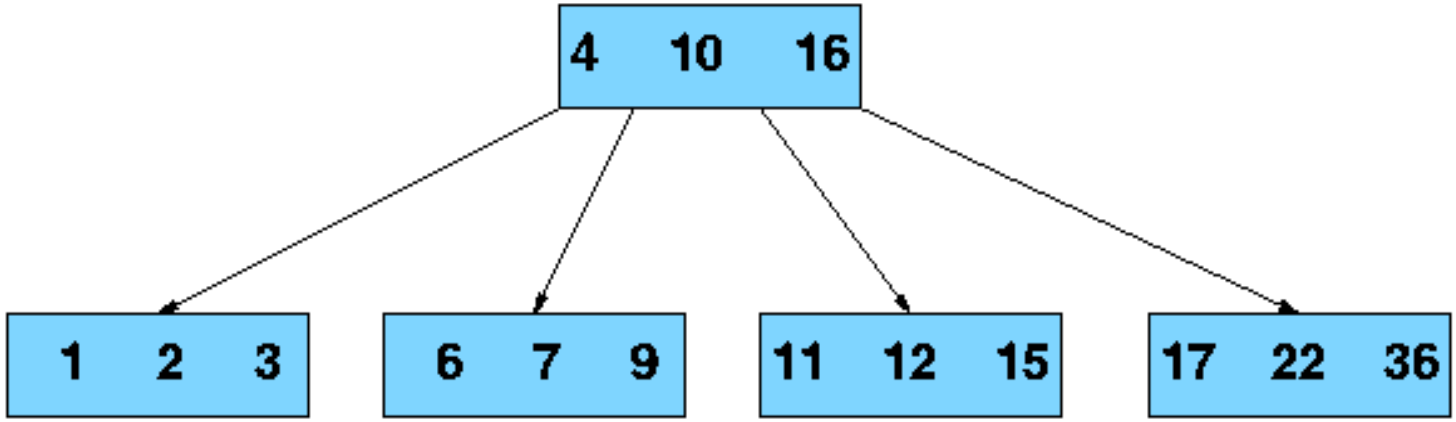
A B-Tree of order *m* has these properties:

- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and *m* children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

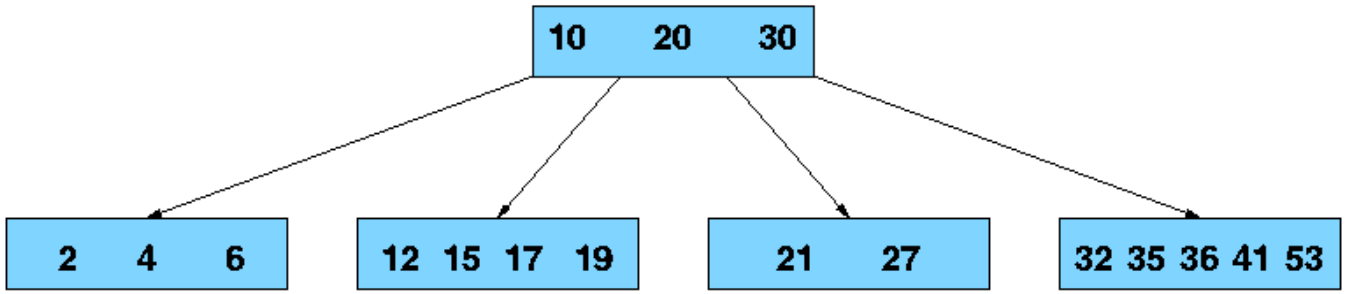
A B-Tree node is usually selected to match the size of a disk block.

- A B-Tree node could have hundreds of children.

# An example B-Tree




B-Tree: Minimization Factor t = 3, Minimum Degree = 2, Maximum Degree = 5




Search(21)

# B-Tree Insertion Algorithm

- It is a generalization of 2-3 tree insertion
  1. Find the leaf node that should contain the key space permitting
  2. If there is room in this node, then insert the key
  3. If there is not, then split the node into two and promote the middle key to the parent
  4. If the parent becomes full, then it is split in turn and the middle key promoted

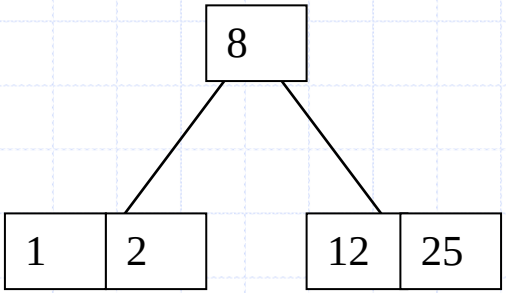
# Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order m=5. So every node (except root) contains at least 2 keys, and at most 4 keys.
- The first four items go into the root:

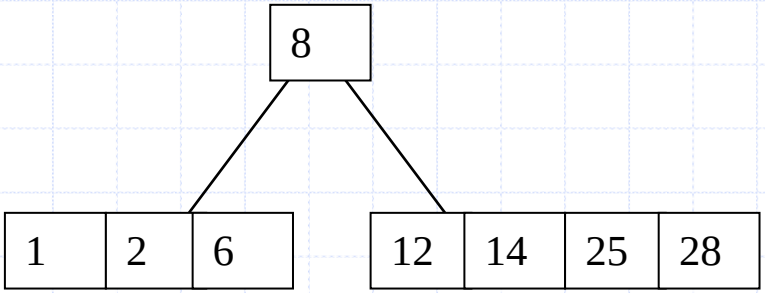
| 1 | 2 | 8 | 12 |
|---|---|---|---|

- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

# Constructing a B-tree (contd.)

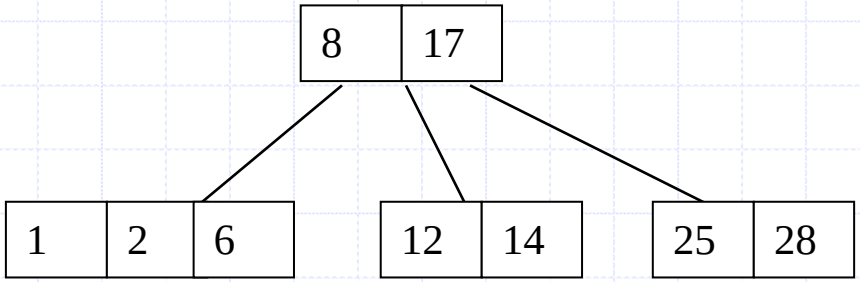



6, 14, 28 get added to the leaf nodes:

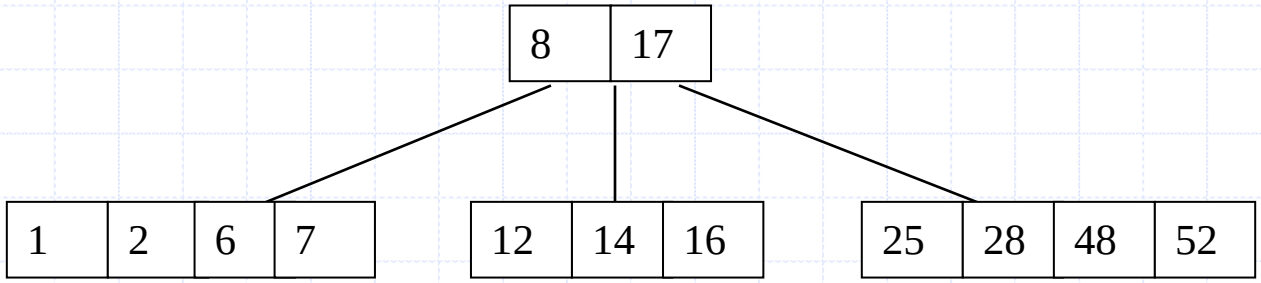


# Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

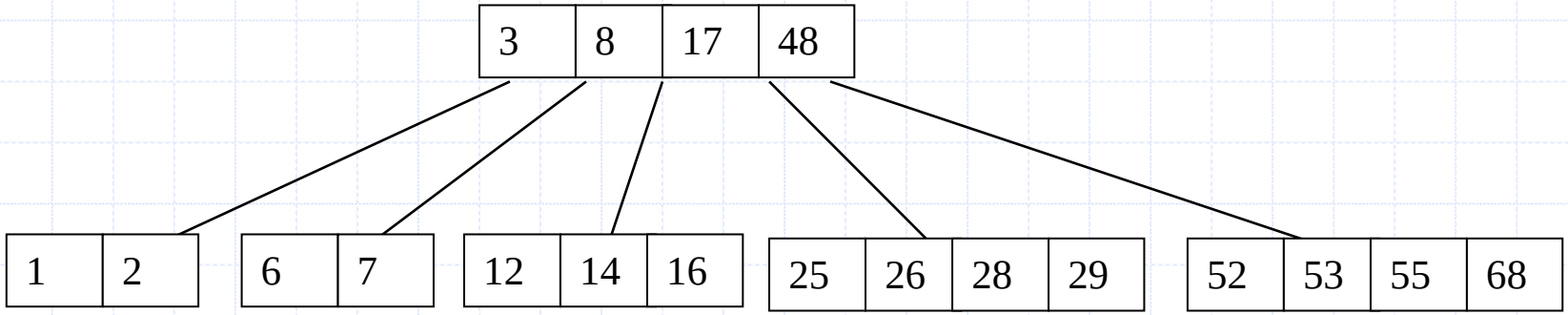



7, 52, 16, 48 get added to the leaf nodes

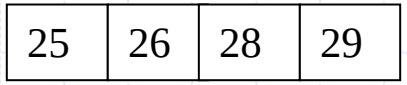


# Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

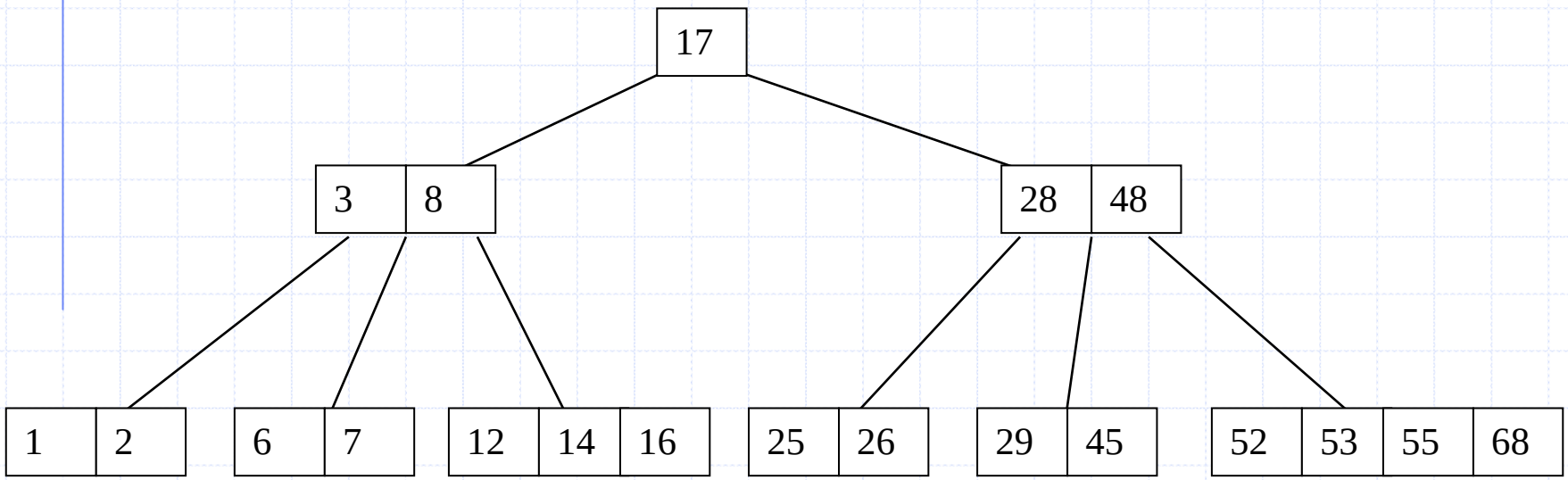



Adding 45 causes a split of




and promoting 28 to the root then causes the root to split

# Constructing a B-tree (contd.)



# Analysis of B-Trees

- The maximum number of items in a B-tree of order $m$ and height $h$:

  root $m - 1$

  level 1 $m(m - 1)$

  level 2 $m^2(m - 1)$

  . . .

  level h $m^h(m - 1)$

- So, the total number of items is

  $$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

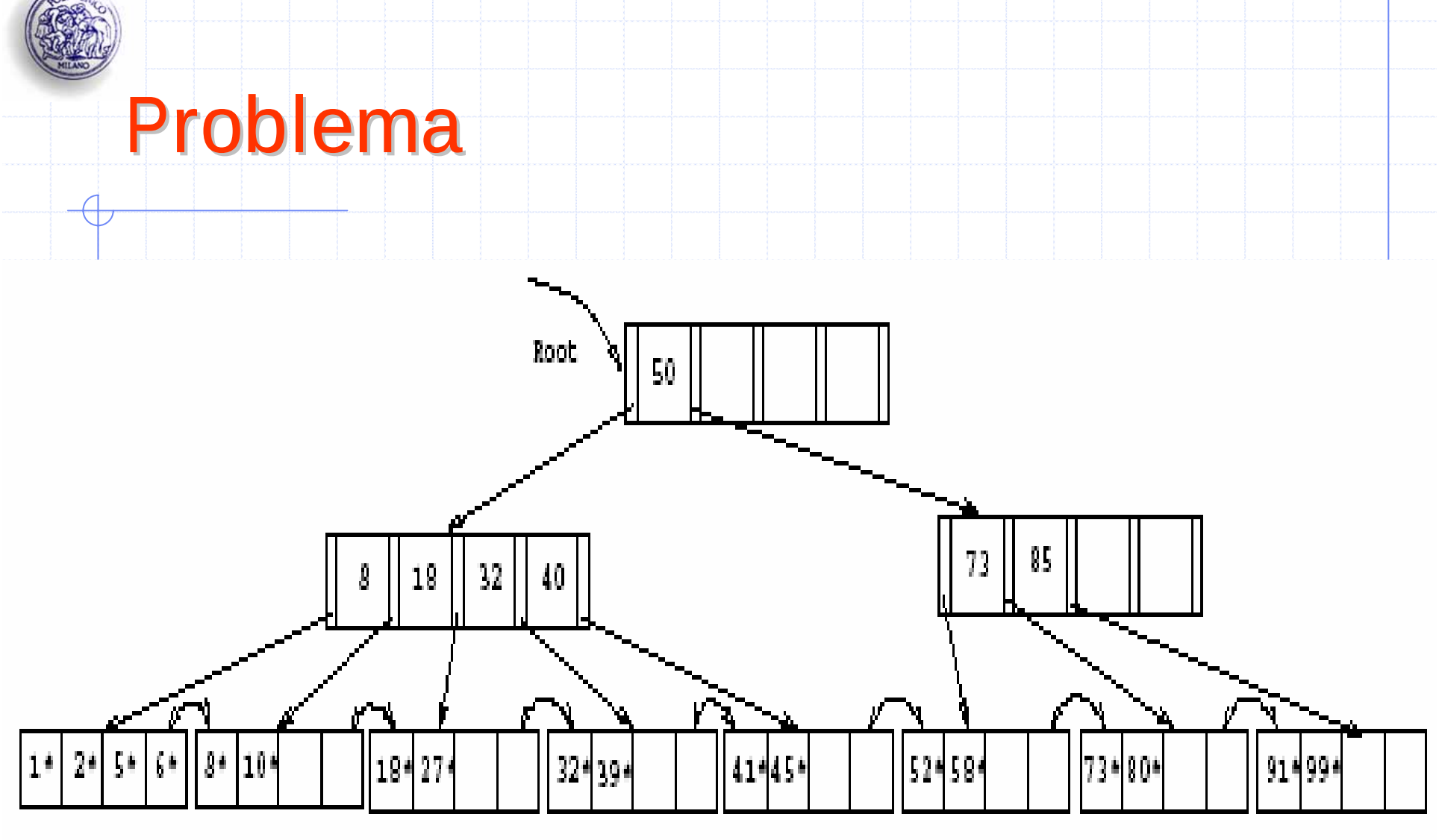
  $$[(m^{h+1} - 1) / (m - 1)] (m - 1) = \mathbf{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

# Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree


# Alberi B+



# Problema

# Domanda 1

- Che tipo di albero e`?

  *B+*

- Che caratteristiche ha questo tipo di albero?

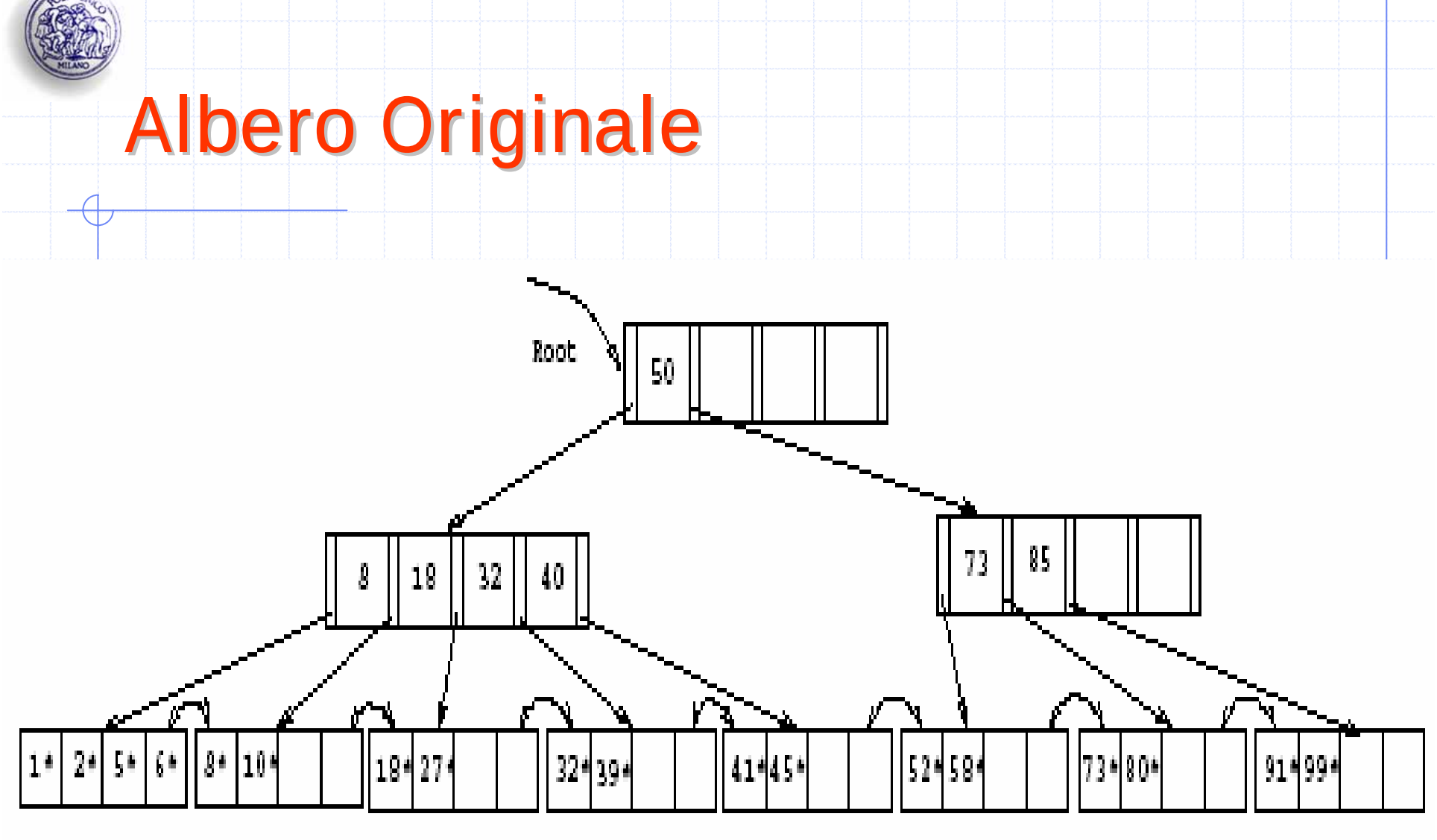
  … *proprieta` dei B+* …

- Di che ordine è?

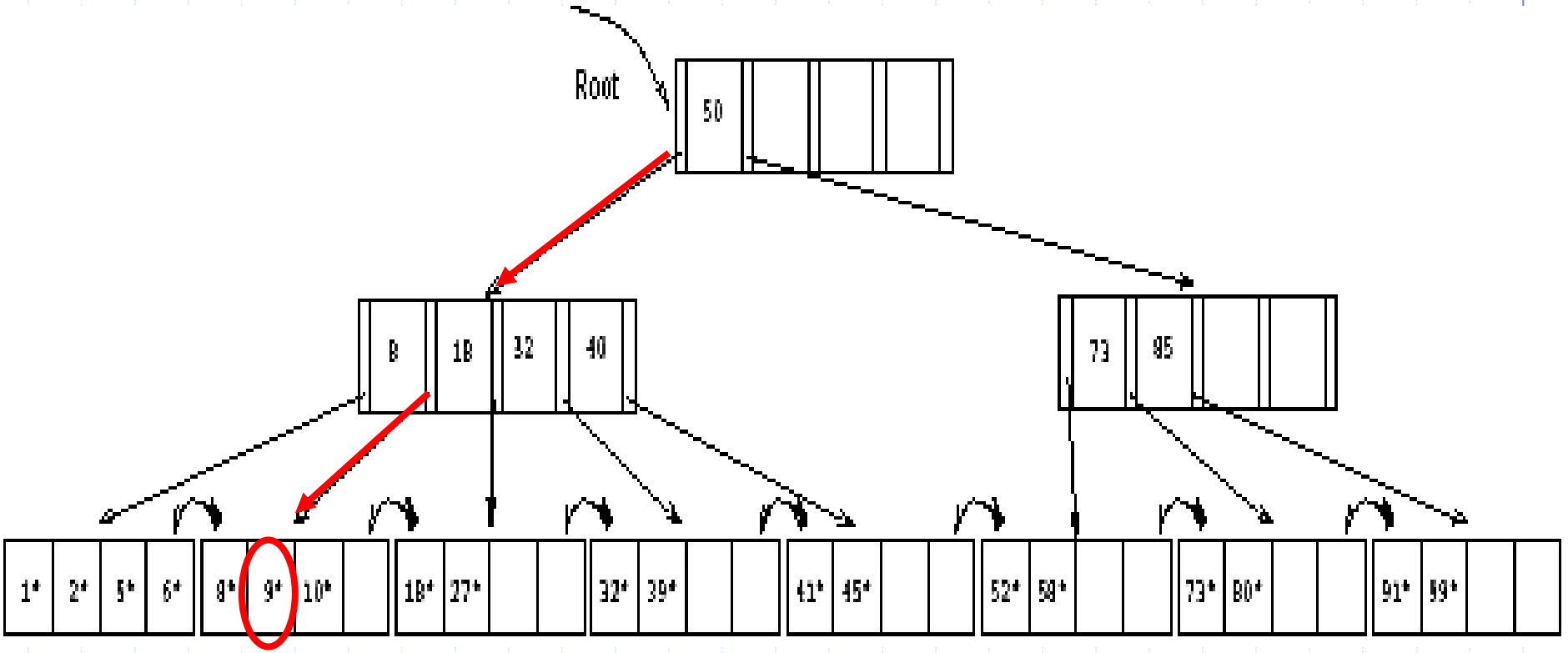
  5

# Domanda 2

- Mostrare cosa accade dopo aver inserito un elemento con chiave 9

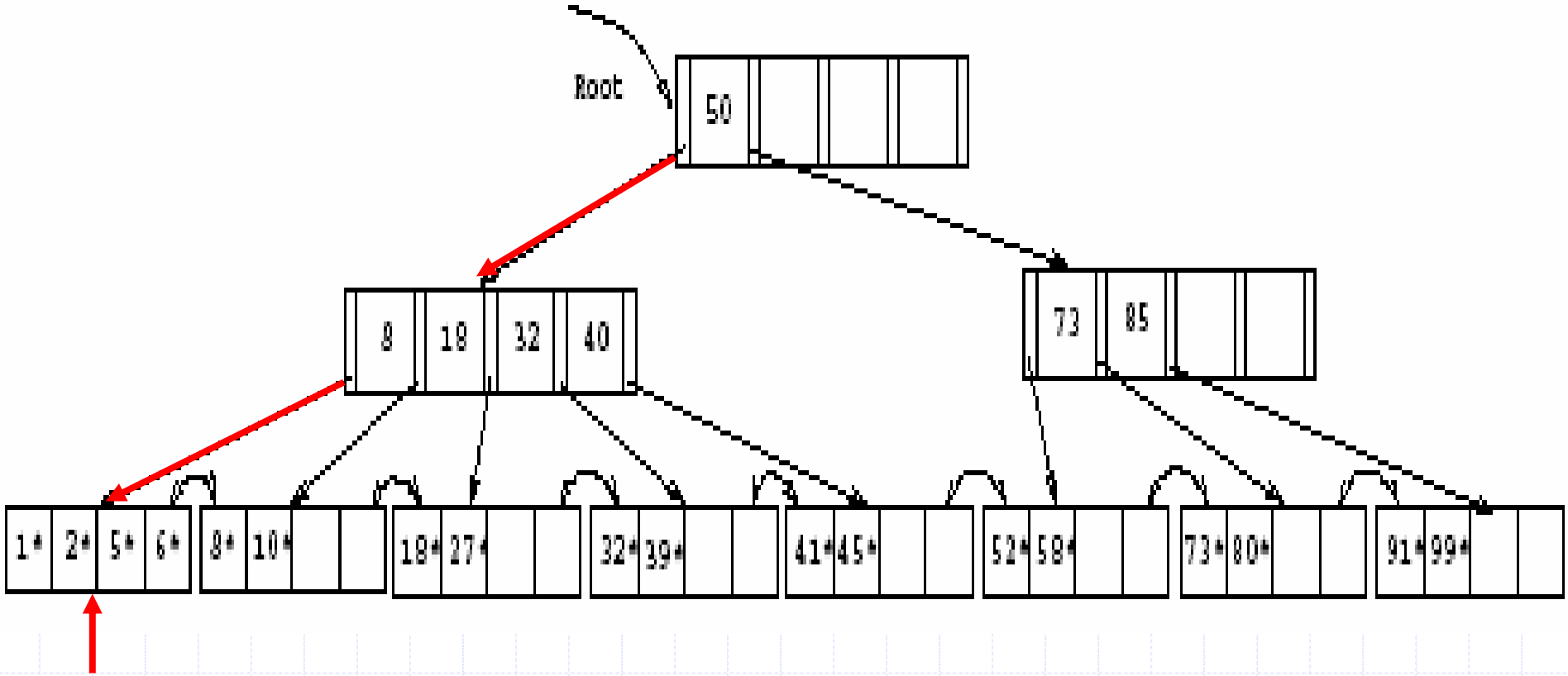


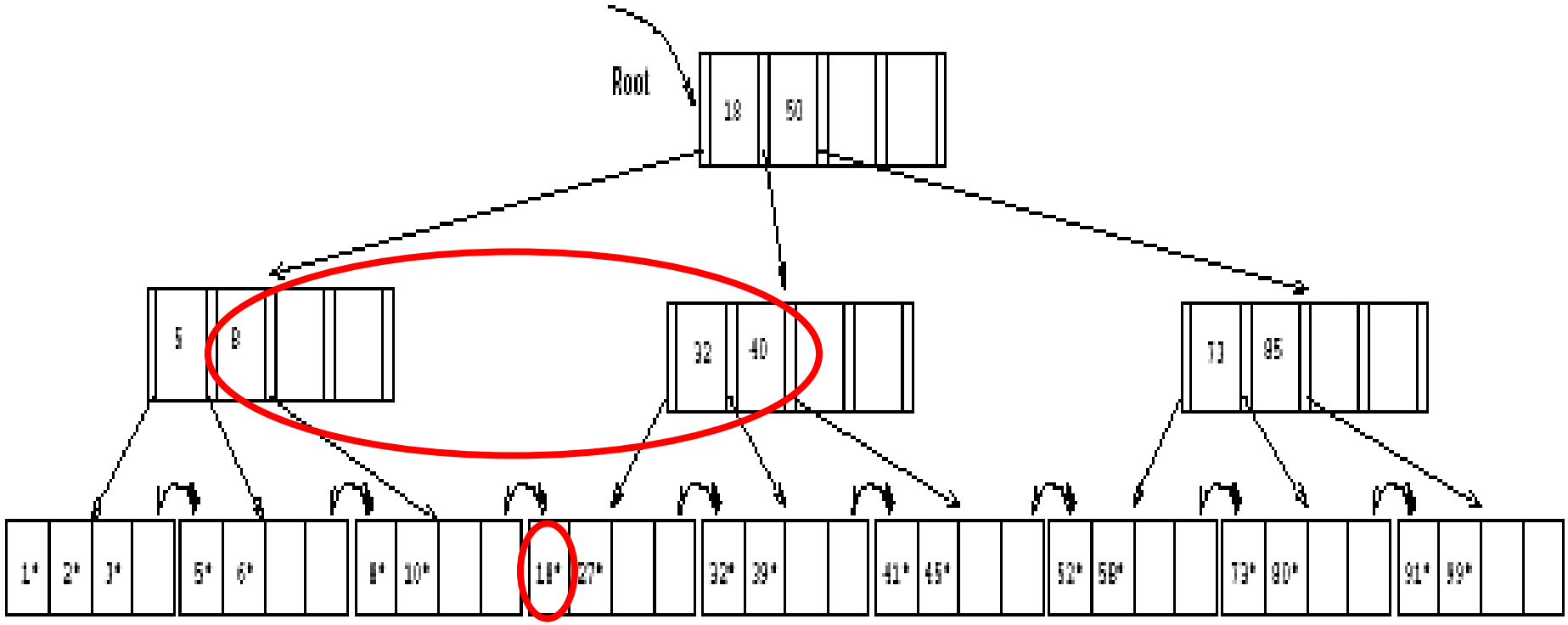
# Albero Originale


Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
9*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

# Domanda 3

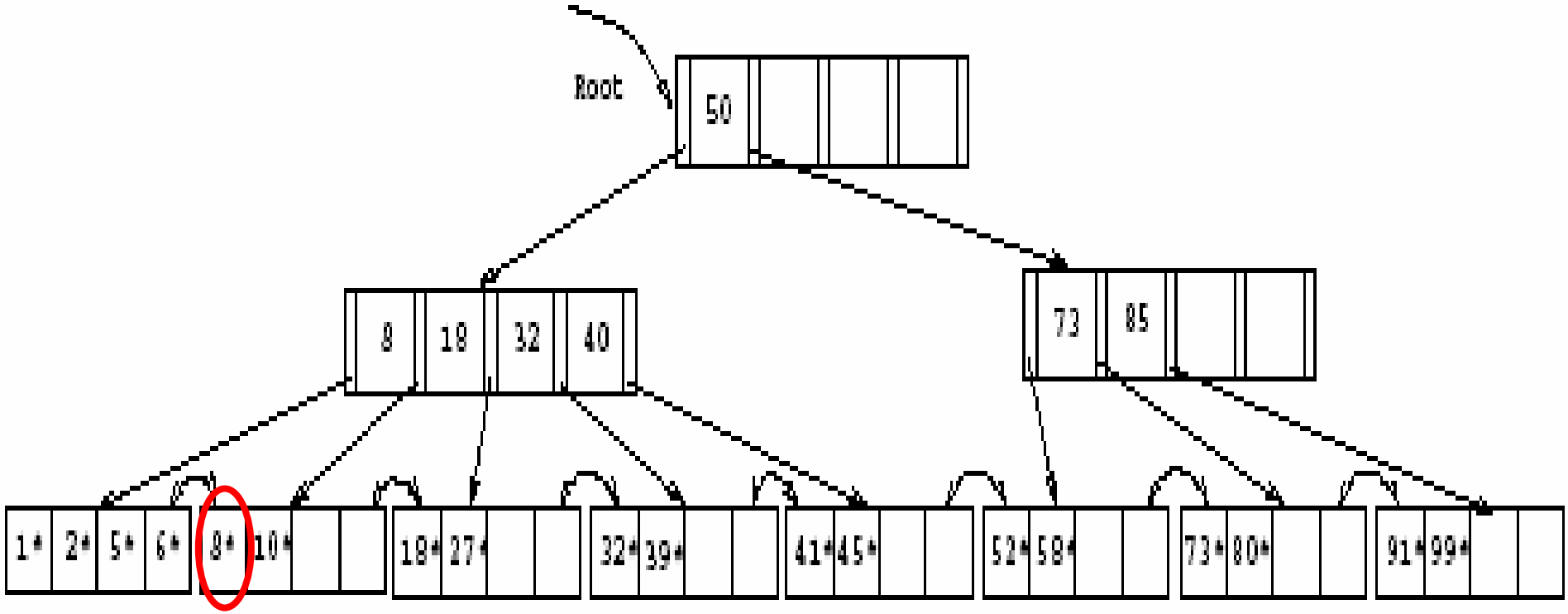
- Mostrare cosa accade dopo aver inserito un elemento con chiave 3 nell'albero originale

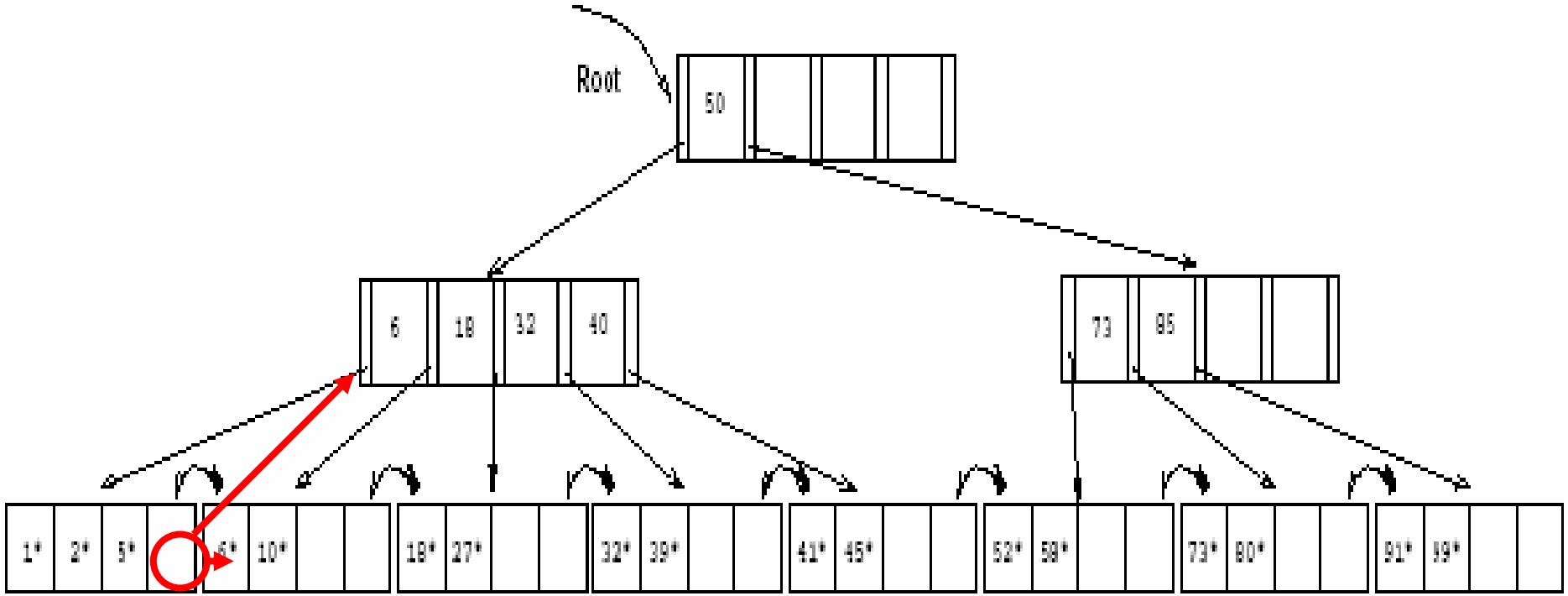

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*


Root
18
50
5
13
32
40
73
85
1*
2*
3*
5*
6*
8*
10*
14*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

# Domanda 4

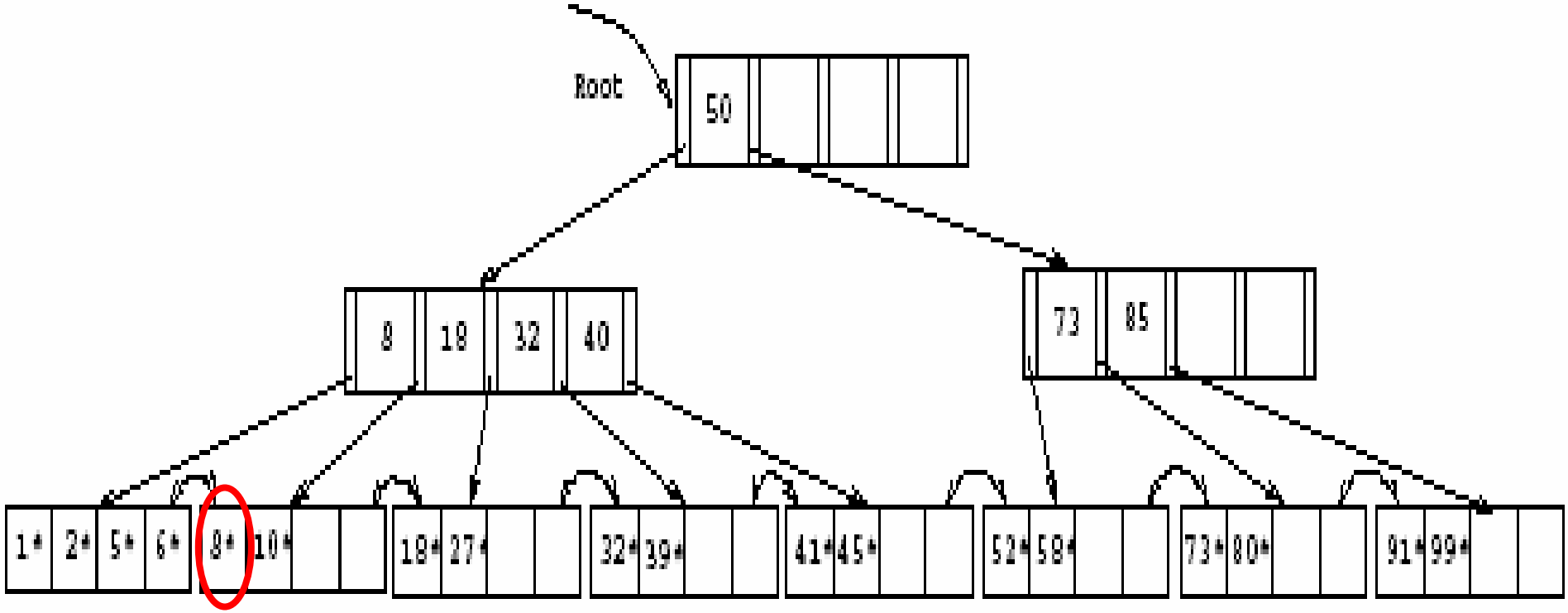
- Mostrare cosa accade dopo aver cancellato l'elemento con chiave 8 dall'albero originale ridistribuendo, in caso di necessità, verso sinistra

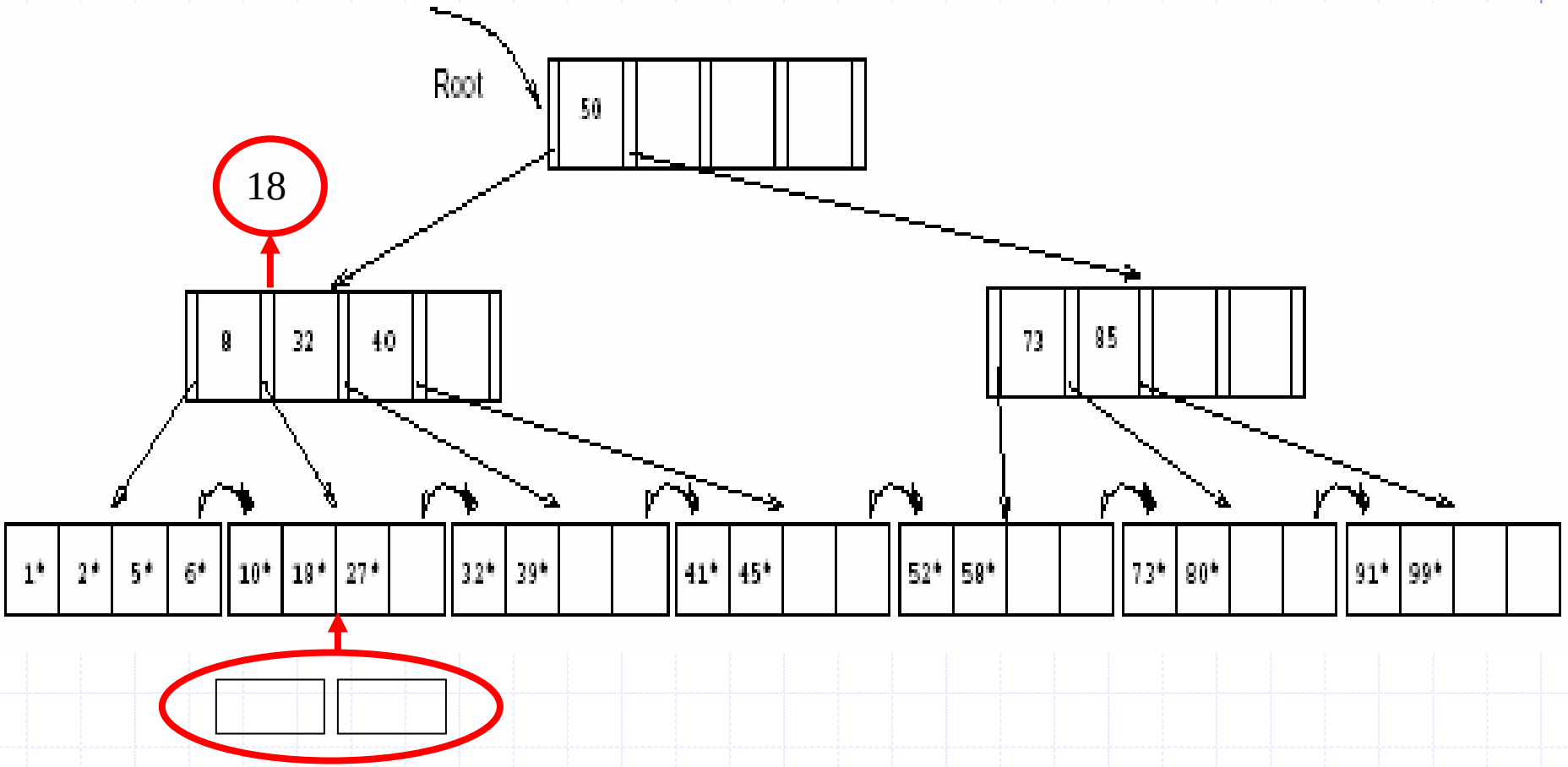

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*


Root
50
6
18
32
40
73
85
1*
2*
5*
6*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

# Domanda 5

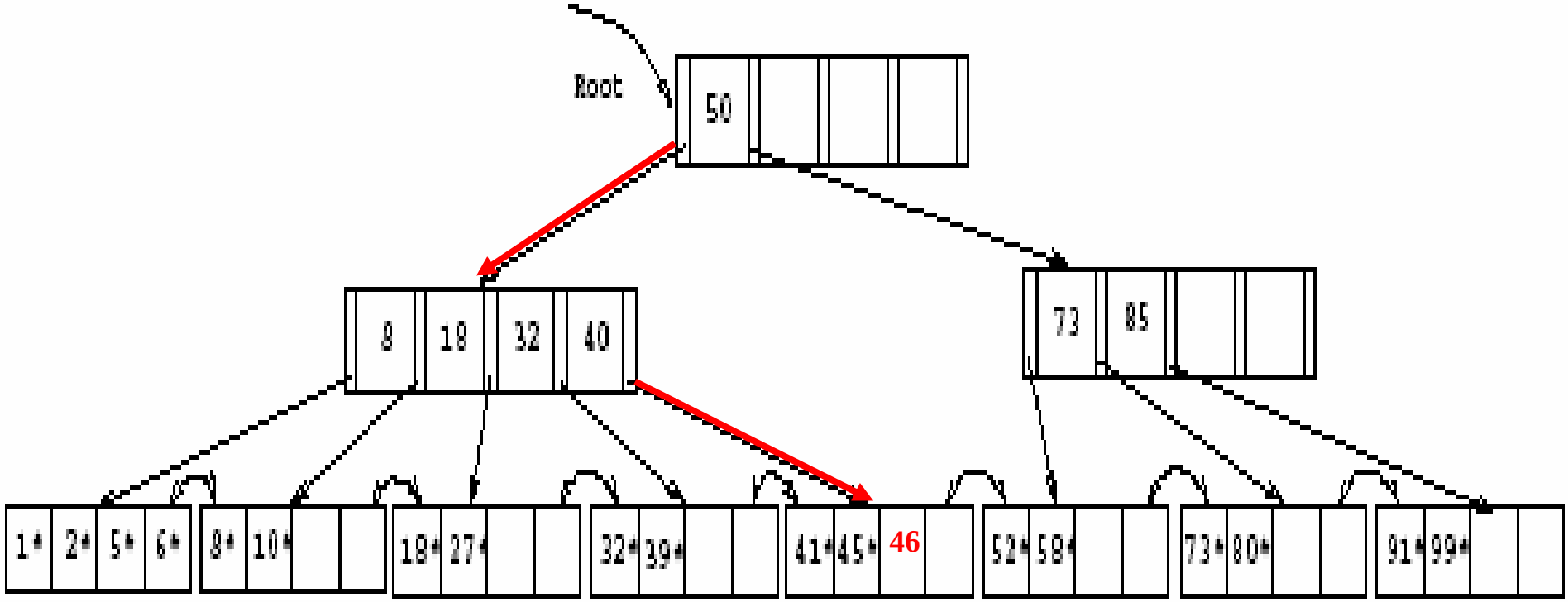
- Mostrare cosa accade dopo aver cancellato l'elemento con chiave 8 dall'albero originale ridistribuendo, in caso di necessità, verso destra

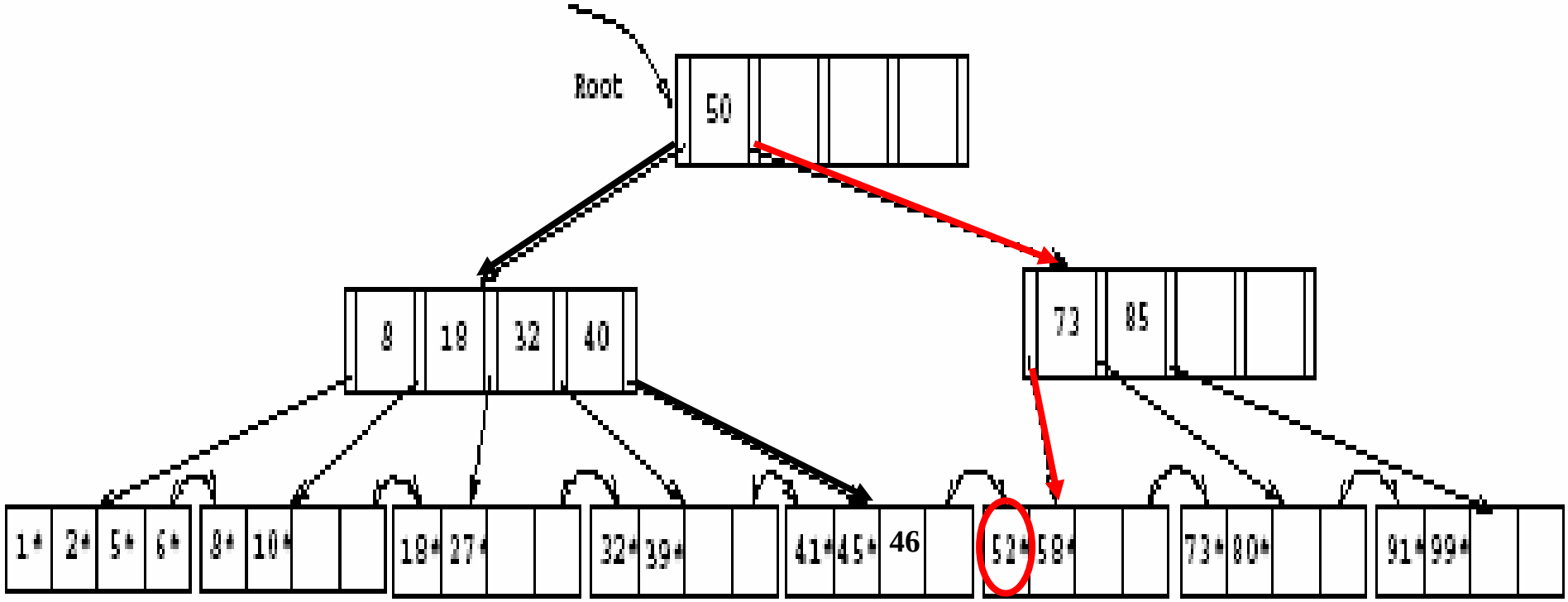

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

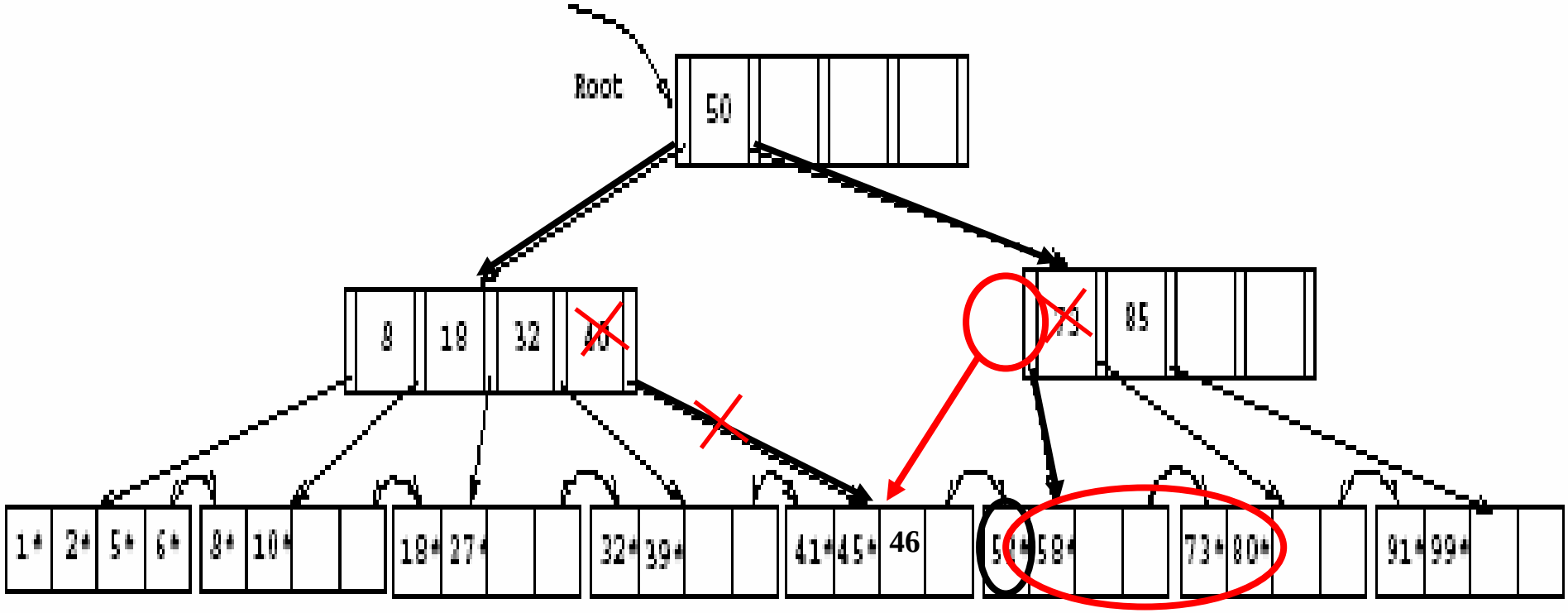

Root
50
18
8
32
40
73
85
1*
2*
5*
6*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

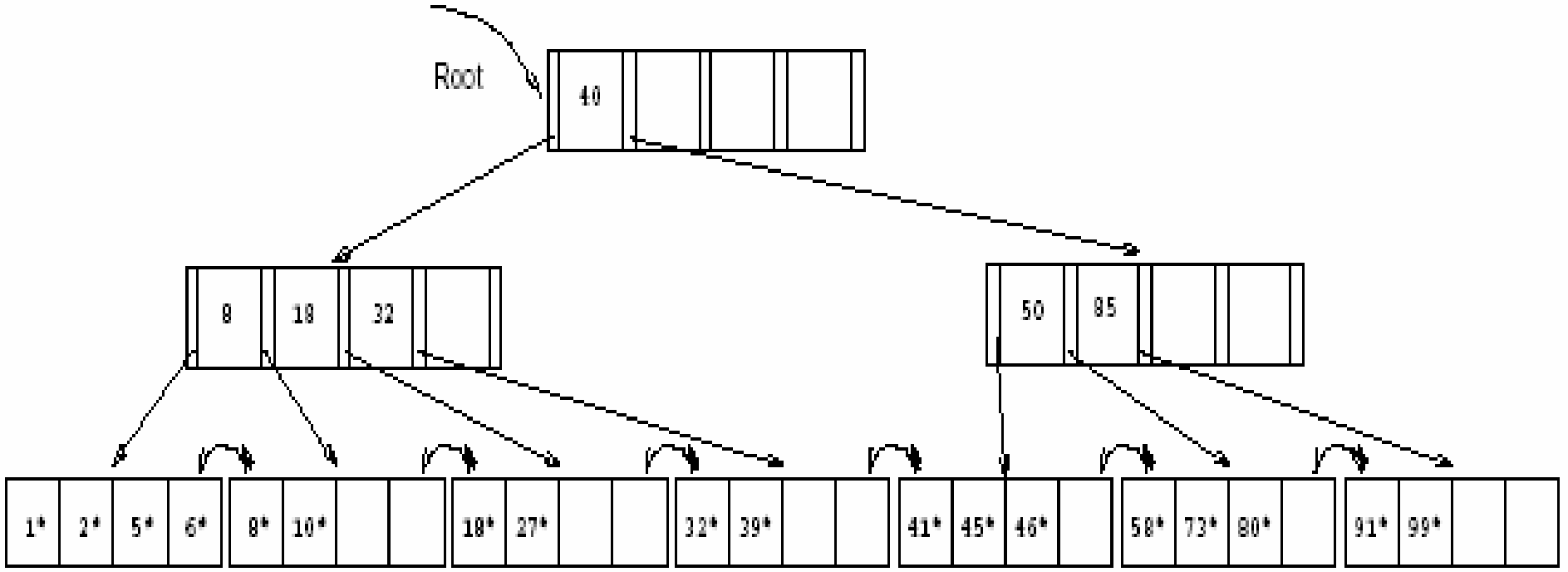
# Domanda 6

- Mostrare cosa accade nell'albero originale dopo aver inserito un elemento con chiave 46 e poi cancellato l'elemento con chiave 52


Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
46
52*
58*
73*
80*
91*
99*

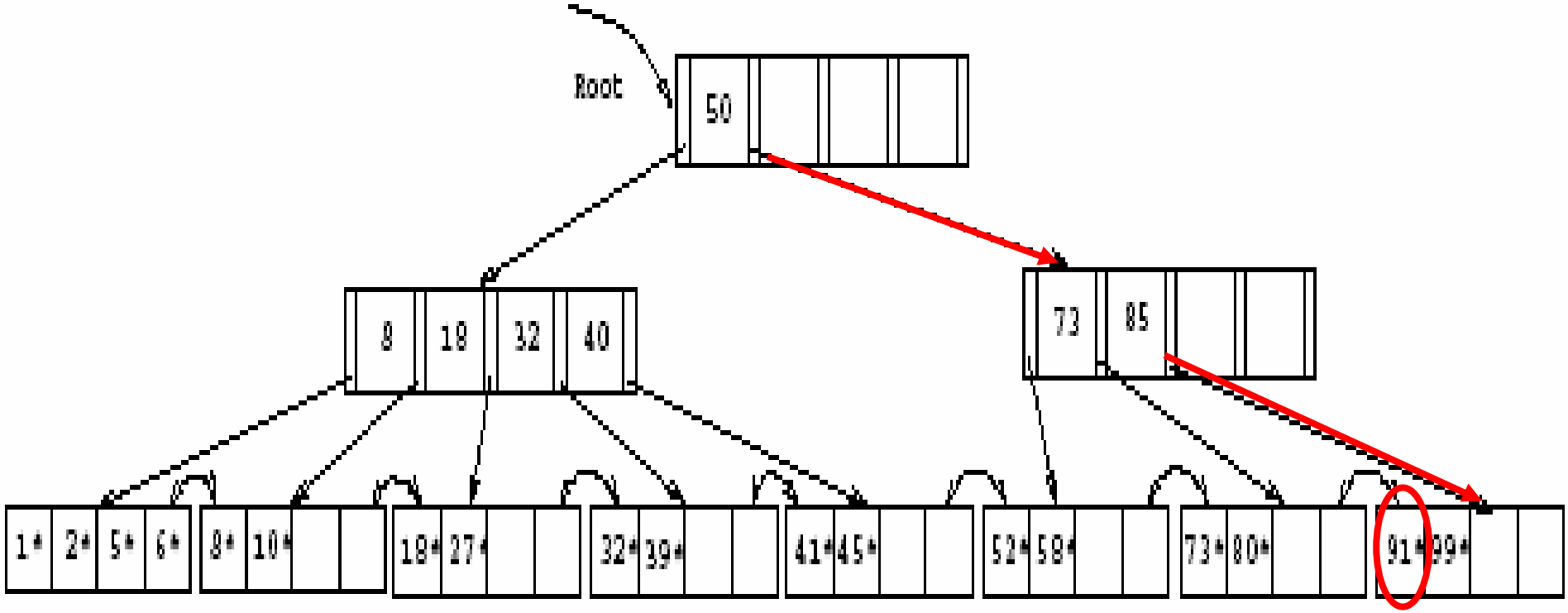

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
46
52*
58*
73*
80*
91*
99*

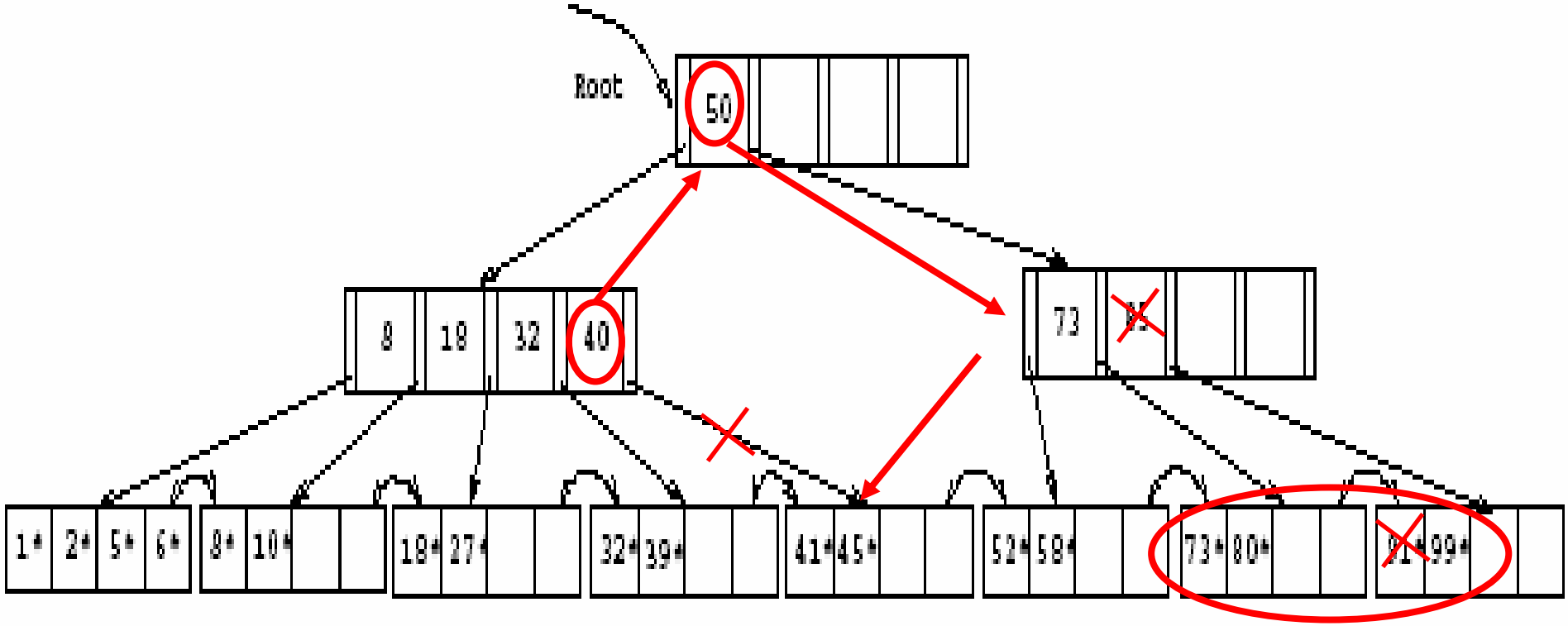

Root
50
8
18
32
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
46
58*
73*
80*
91*
99*

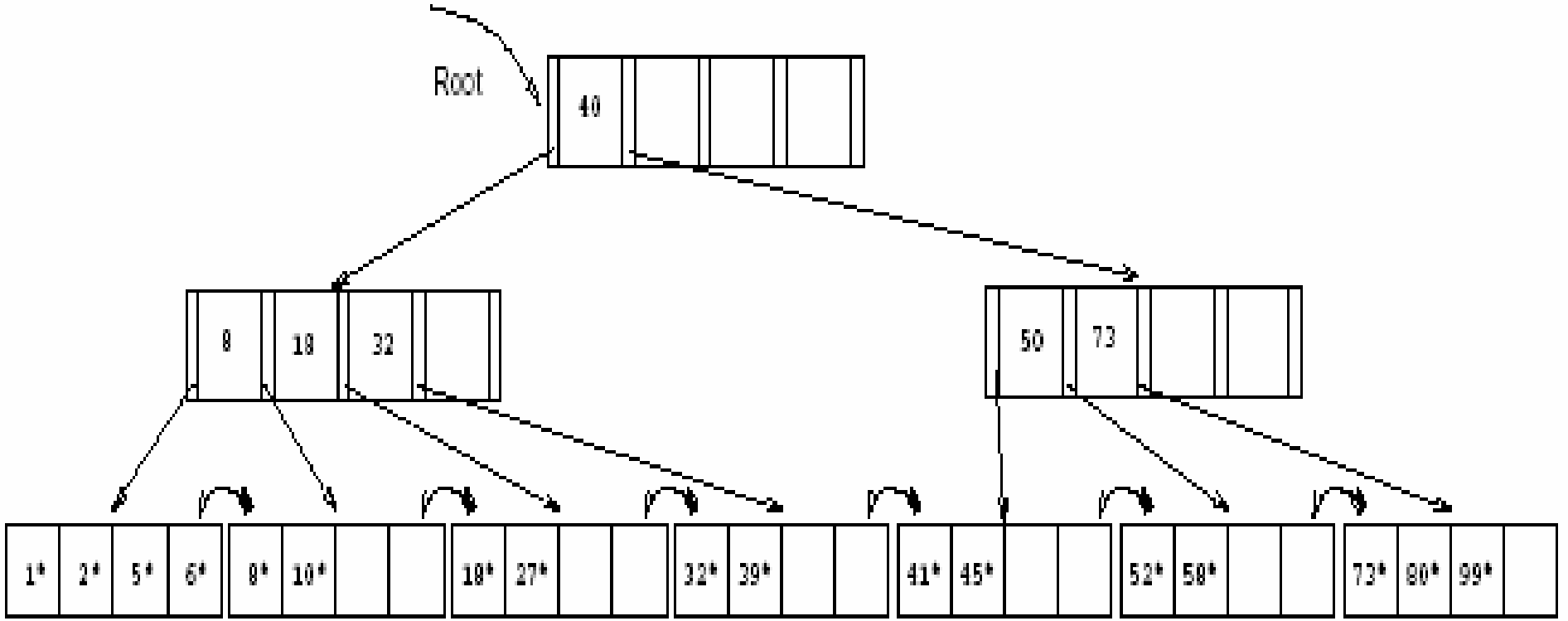

Root
40
8
18
32
50
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
46*
58*
73*
80*
91*
99*

# Domanda 7

- Mostrare cosa accade dopo aver cancellato l'elemento di chiave 91 dall'albero originale

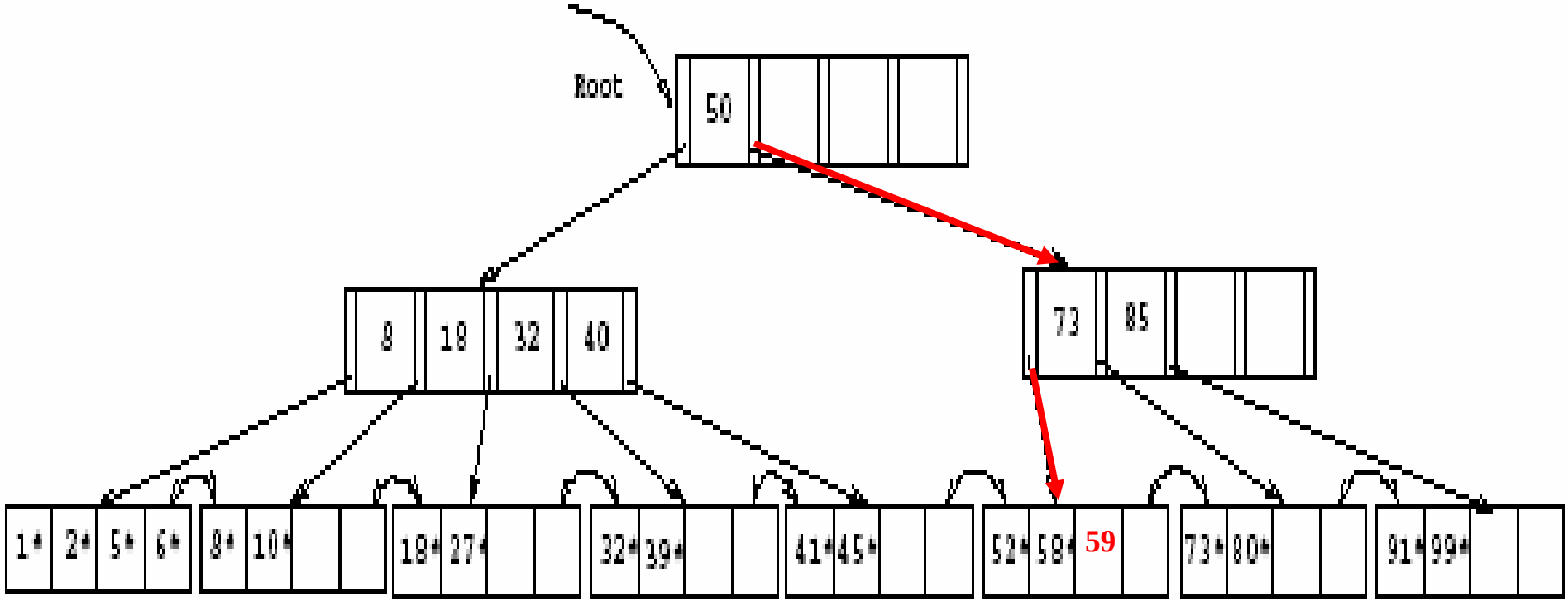

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

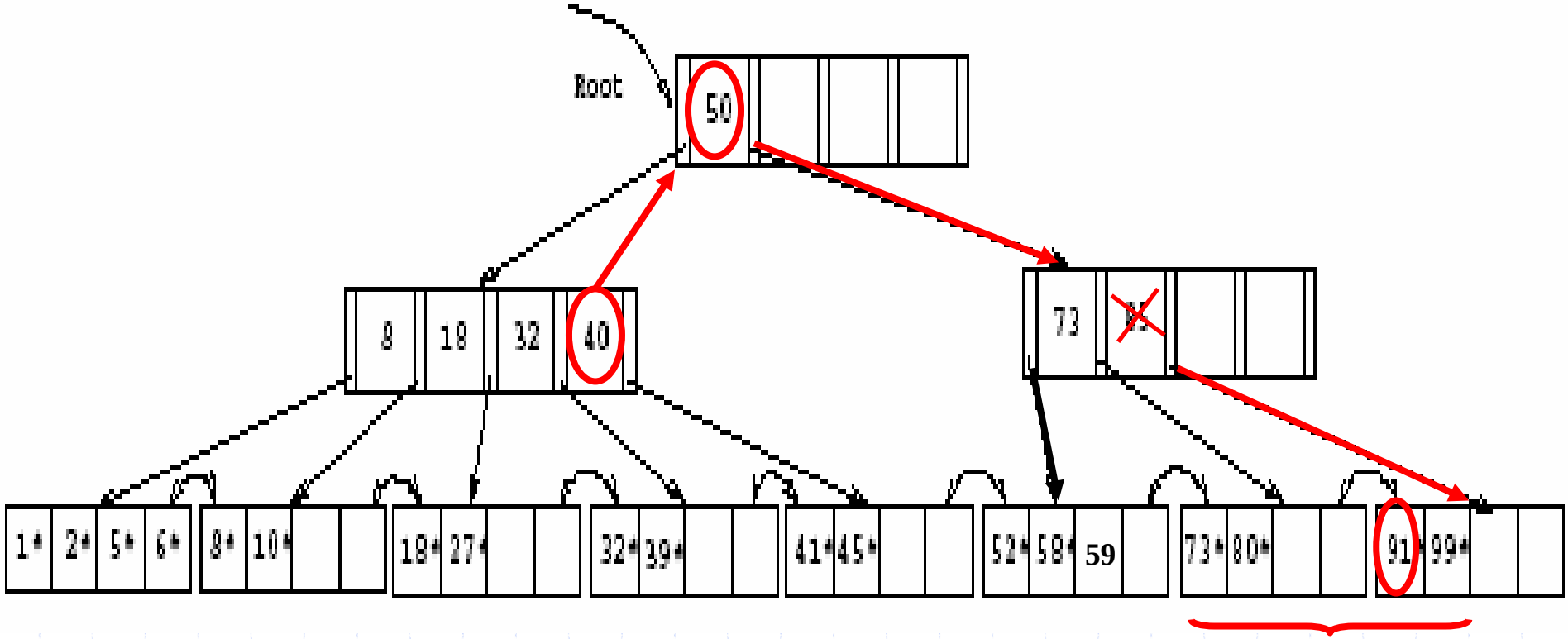

Root
50
8
18
32
40
73
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
91*
99*

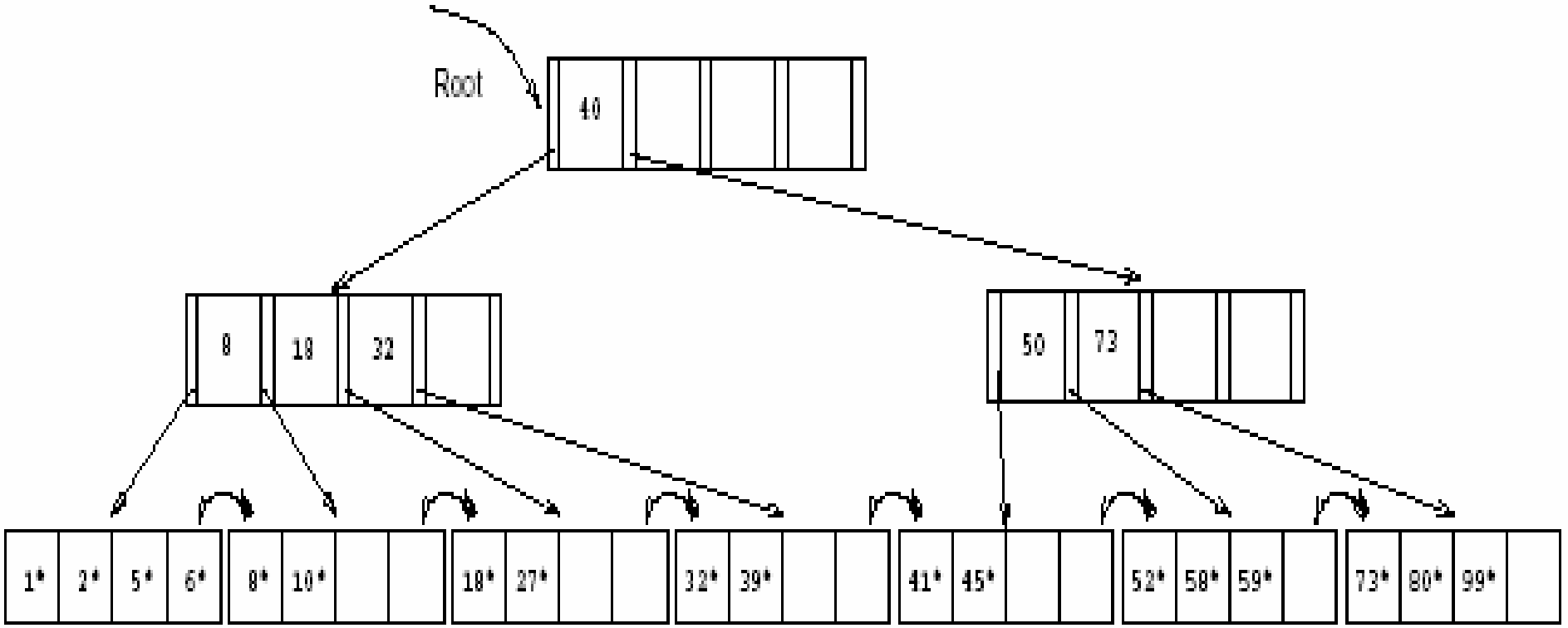

Root
40
8
18
32
50
73
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
73*
80*
99*

# Domanda 8

- Mostrare cosa accade nell'albero originale dopo aver inserito un elemento con chiave 59 e poi cancellato l'elemento con chiave 91

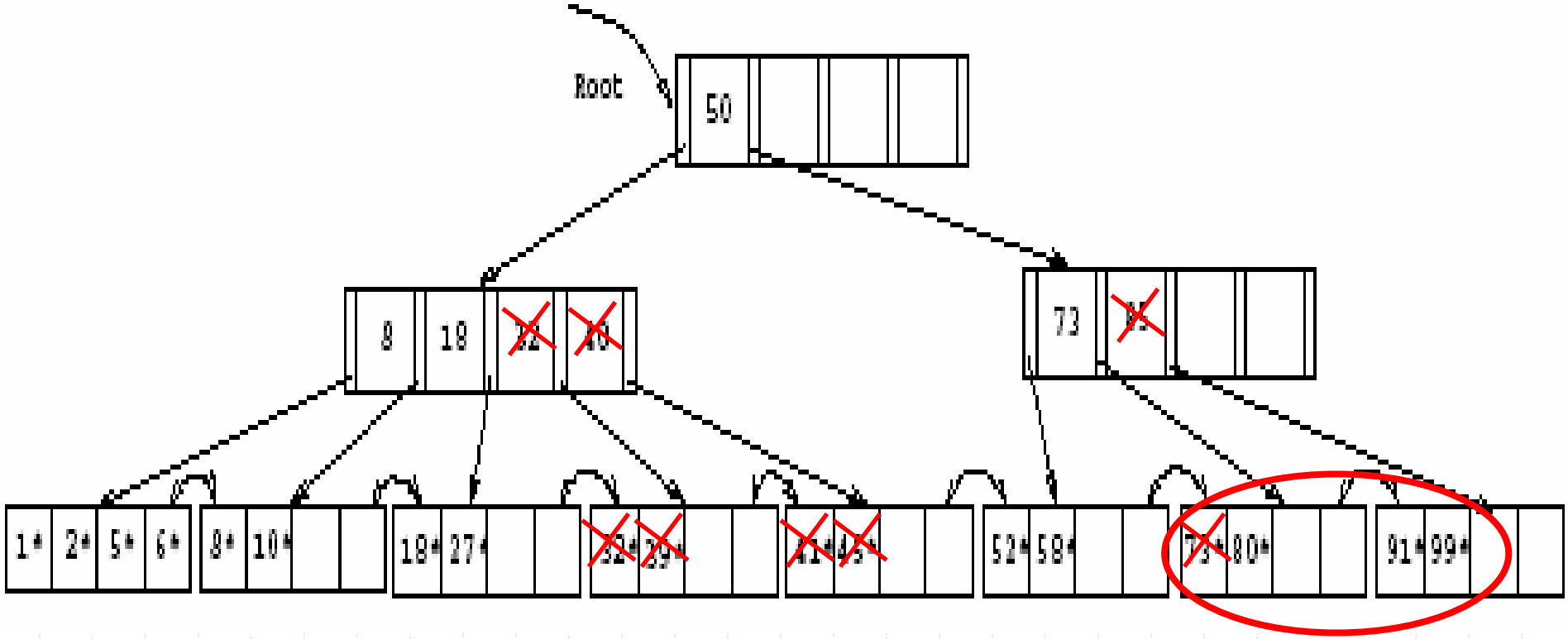

Root
50
8
18
32
40
73
85
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
59
73*
80*
91*
99*

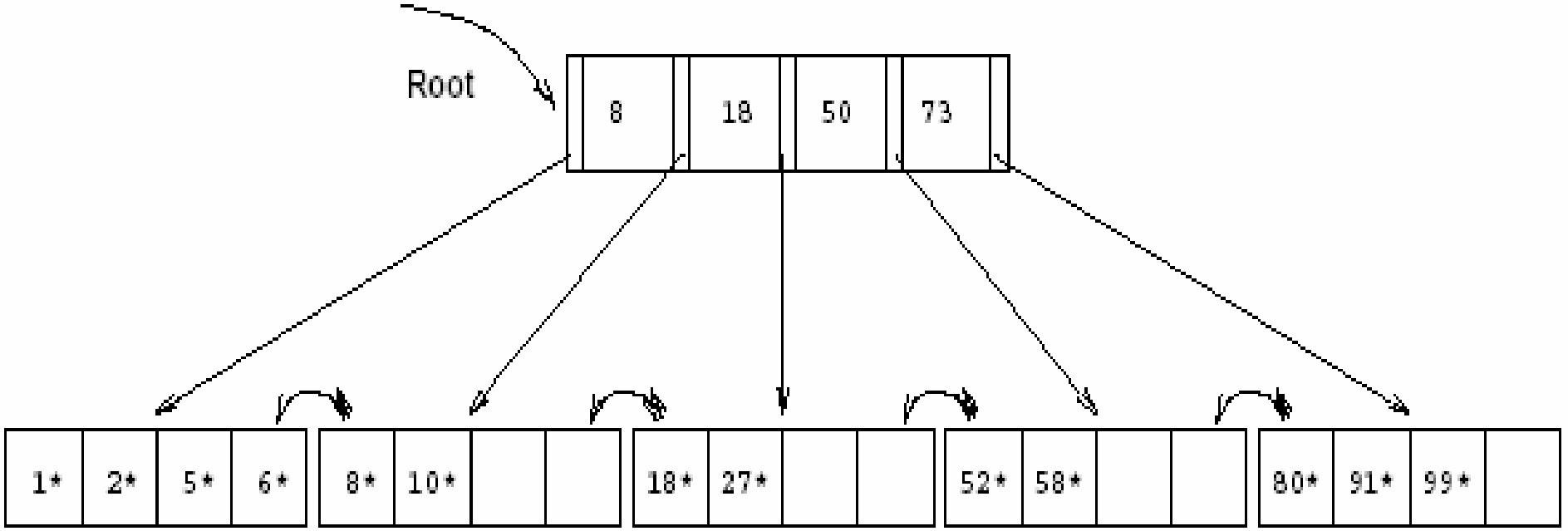

Root
50
8
18
32
40
73
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
59
73*
80*
91*
99*


Root
40
8
18
32
50
73
1*
2*
5*
6*
8*
10*
18*
27*
32*
39*
41*
45*
52*
58*
59*
73*
80*
99*

# Domanda 9

- Mostrare cosa accade dopo aver cancellato dall'albero originale gli elementi 32, 39, 41, 45 e 73


Root
50
8
18
73
1*
2*
5*
6*
8*
10*
18*
27*
52*
58*
73*
80*
91*
99*


Root
8
18
50
73
1*
2*
5*
6*
8*
10*
18*
27*
52*
58*
80*
91*
99*