fit@hcmus

# DATA STRUCTURES & ALGORITHMS

## Lecture 6: TREES – Part 3

## B-trees, 2-3 trees, 2-3-4 trees

Lecturer: Dr. Nguyen Hai Minh

# OUTLINE

- Balanced Search Tree
  - B-tree
  - 2-3 tree
  - 2-3-4 tree
- Comparing trees

# B-TREES

- Definition
- Example
- Traversal, Search, Insert, Delete
- Advantage of B-trees

# Motivation of B-trees

- So far, we have assumed that we can store an entire data structure in ***main memory***
- What if we have so much data that it won't fit?
  - Storing it on disk requires different approach to ***efficiency***
  - Assuming that a disk spins at 7200 RPM, one revolution occurs in **8.33ms**
  - Crudely speaking, one disk access takes about the same time as **100,000 instructions**

# Motivation of B-trees

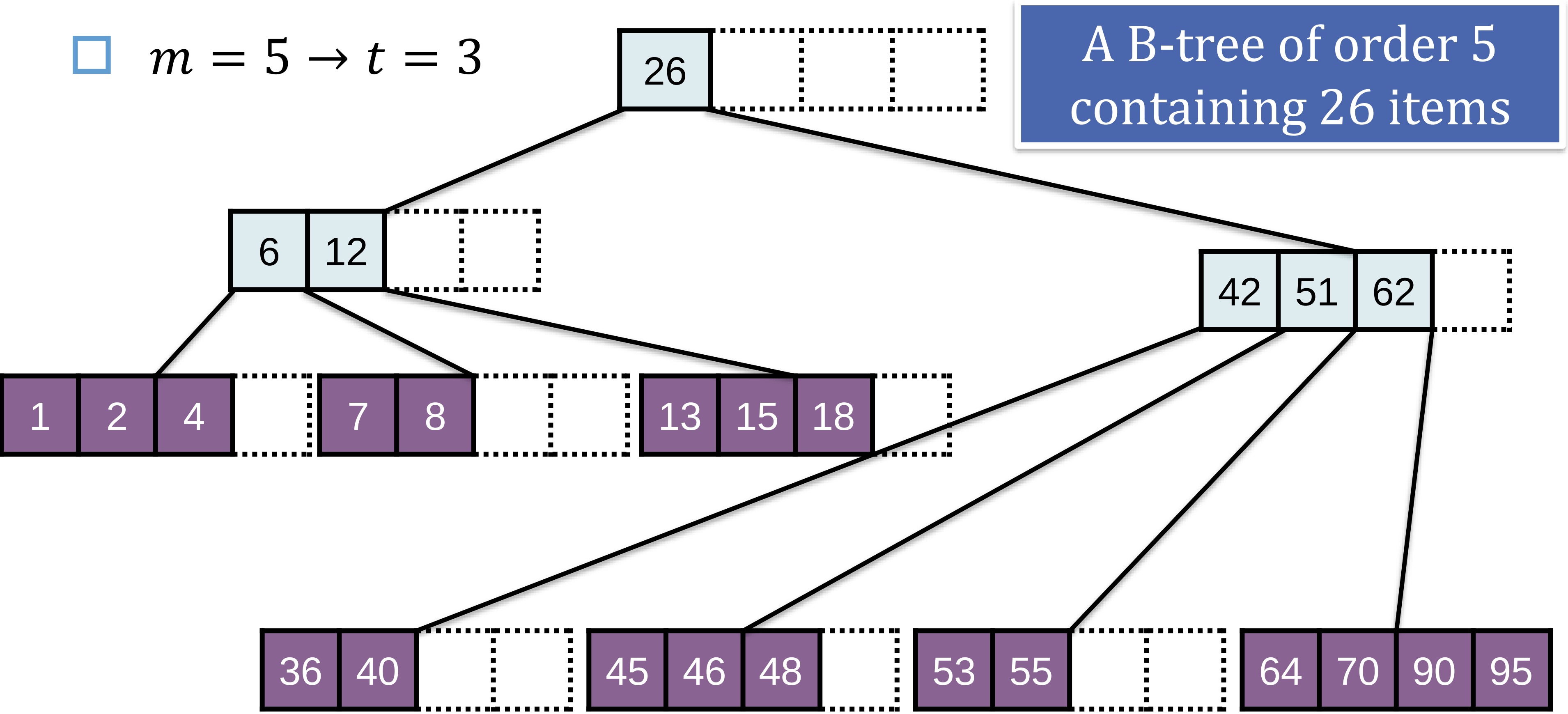
- Assume that we use an AVL tree to store about 20 million records
  - We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000 \approx 24$, so this takes about 0.2 seconds
- We know we can't improve on the $\log_2 n$ lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - → As branching increases, depth decreases
  - → B-tree: 1970 by Rudolf Bayer & Edward M. McCreight

# Definition of a B-tree

- A B-tree of order $m$ is an $m$-way tree (i.e., a tree where each node may have up to $m$ children) in which:
  1. The number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. All leaves are on the same level
  3. All non-leaf nodes except the root have at least $t = \left\lceil \frac{m}{2} \right\rceil$ children
  4. The root is either a leaf node, or it has from $2$ to $m$ children
  5. Every nodes except the root has $\left\lceil \frac{m}{2} \right\rceil - 1$ to $m - 1$ keys
- The number $m$ is usually odd

# An Example B-tree

- $m = 5 \rightarrow t = 3$

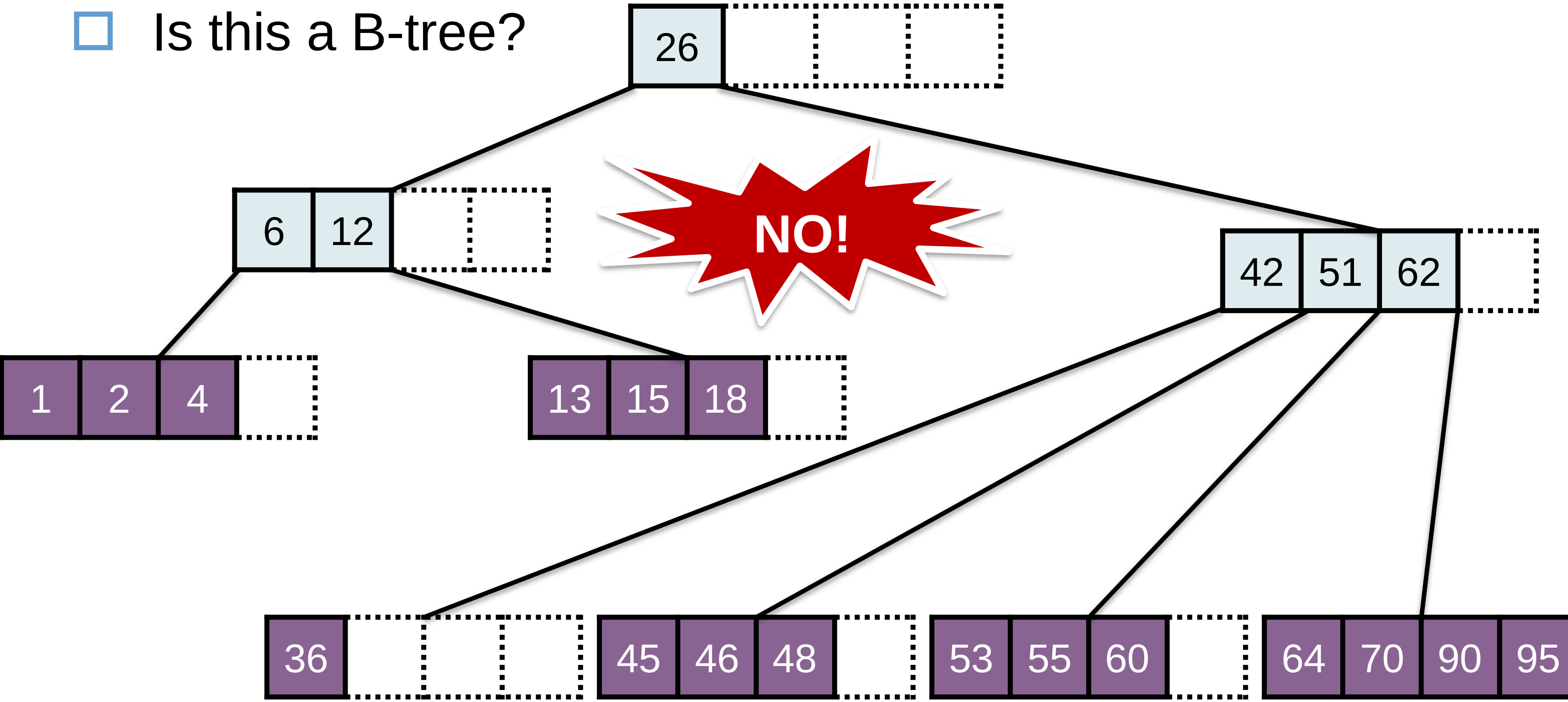



→ Each internal nodes (except root) has at least 3 children

→ Each node has 2-4 keys

# An Example B-tree

- Is this a B-tree?



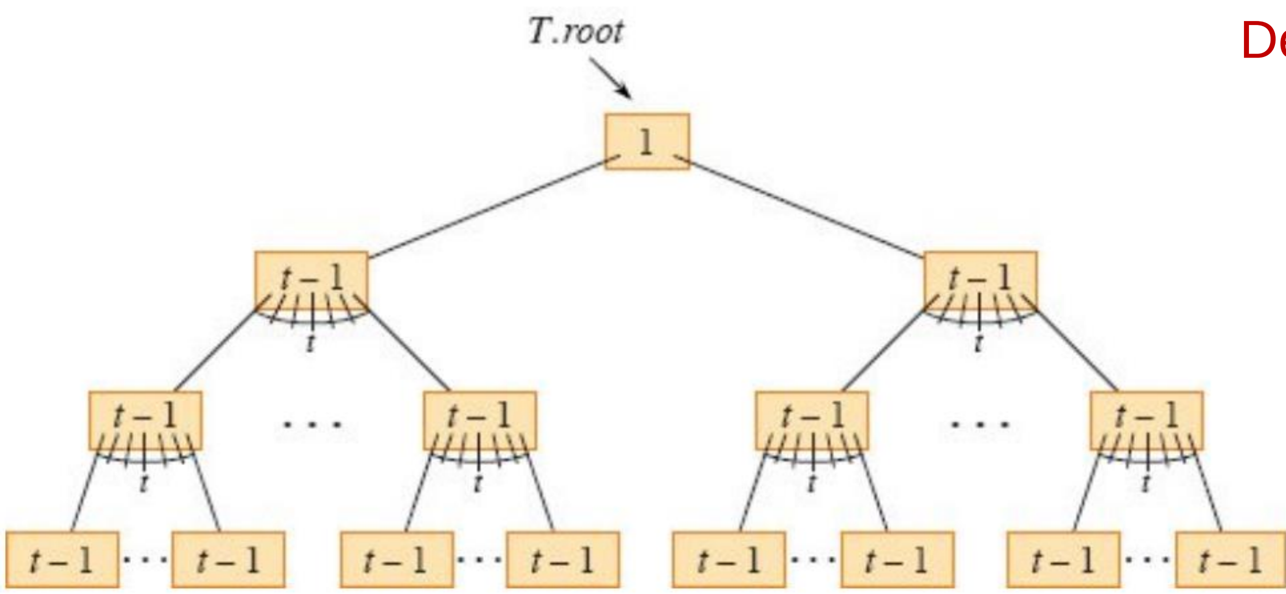

- Why?

# B-tree Example

- Draw a B-tree of order 3 and height = 3 containing as many keys/data items as possible.
  - How many nodes are there?
  - How many keys does your tree have?

# Height of B-tree

- Let $h$ be the height, $n$ is the total keys on a B-tree
- $t$: minimum degree of $m$-way B-tree
- Minimum keys on a B-tree:




| Depth | #nodes | #keys |
|---|---|---|
| $0$ | $1$ | $1$ |
| $1$ | $2$ | $2(t-1)$ |
| $2$ | $2t$ | $2t(t-1)$ |
| $3$ | $2t^2$ | $2t^2(t-1)$ |

# B-tree's height

- Minimum keys on a B-tree:

$$n \geq 1 + 2(t-1) + 2t(t-1) + \cdots + 2t^{h-1}(t-1)$$

$$n \geq 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1} = 1 + 2(t-1)\left(\frac{t^{h-1}}{t-1}\right)$$

$$n \geq 2t^h - 1$$

- **Theorem:** If $n \geq 1$, then for any n-keys B-tree of height $h$ and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}$$

→ $h = O(\log_t n)$

# B-tree's maximum nodes

- The maximum number of keys 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 keys is:

$$n \leq (1 + m + m^2 + m^3 + \ldots + mh)(m - 1) =$$

$$\left[\frac{m^{h+1} - 1}{m - 1}\right](m - 1) = m^{h+1} - 1$$

$$\rightarrow h \geq \log_m(n + 1) - 1$$

# Traversal and Search in B-tree

- Traversal in B-tree:
  - Similar to *In-Order Traversal* of Binary Tree.
- Search in B-tree:
  - Similar to searching in a Binary Search Tree, except that instead of making a binary, or “two-way,” branching decision at each node, we make a *multiway* branching decision according to the number of the node’s children.

# B-tree-SEARCH

```
B-tree-SEARCH(x,k)
```

$$\boxed{O(th) = O(t\log_t n)}$$

//x: pointer to the root node of a subtree, k: key to be searched

```
i = 0
while i < x.n and k > x.key_i
    i = i + 1                              O(t)
if i < x.n and k == x.key_i
    return (x,i)
elseif x.leaf
    return NULL
else DISK-READ(x.c_i)
    return B-tree-SEARCH(x.c_i,k)          O(h)
```

# Inserting into a B-tree $O(t \log_t n)$



- Attempt to insert the new key into a leaf
- If *leaf* becomes too big,
  - Split the leaf into two
  - Promoting the middle key to the leaf's parent
- If *the parent* becomes too big
  - Split the parent into two
  - Promoting the middle key

→ This strategy might have to be repeated all the way to the top.

→ If necessary, the root is split in two and the middle key is promoted to a new root, *making the tree one level higher*

# Inserting into a B-tree – Example

- 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 5

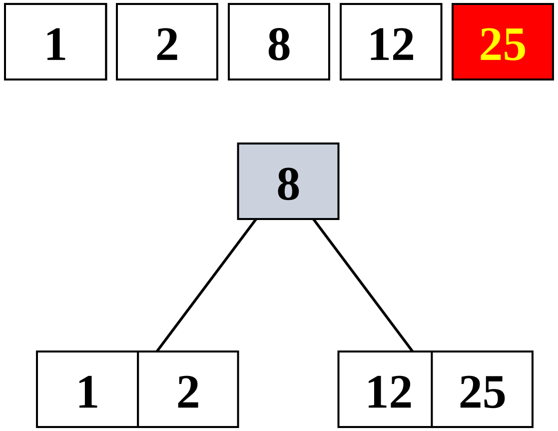
# Inserting into a B-tree – Example

- **1 12 8 2**

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

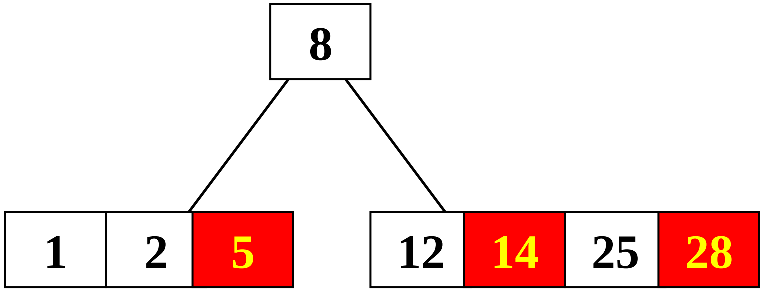
# Inserting into a B-tree – Example

- 1 12 8 2 **25**



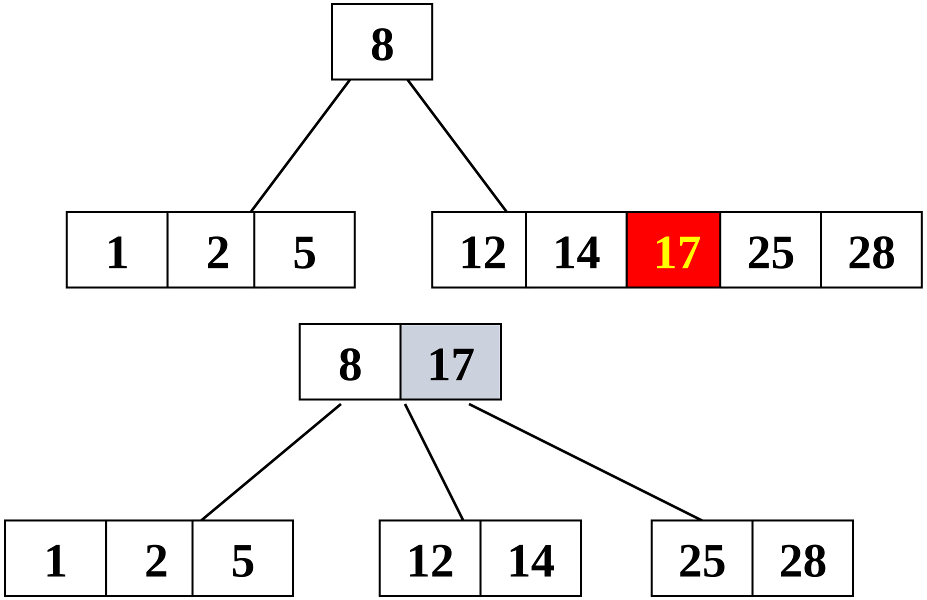
# Inserting into a B-tree – Example

- 1 12 8 2 25 **5 14 28**



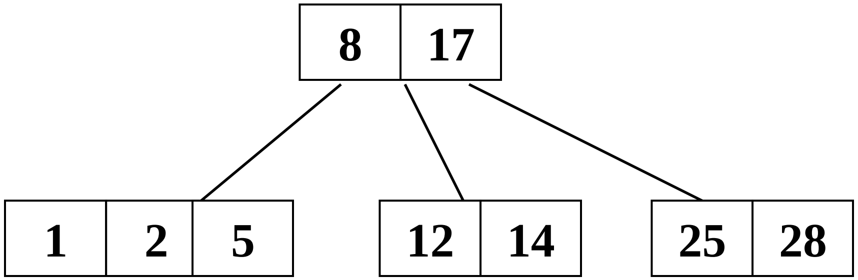
# Inserting into a B-tree – Example

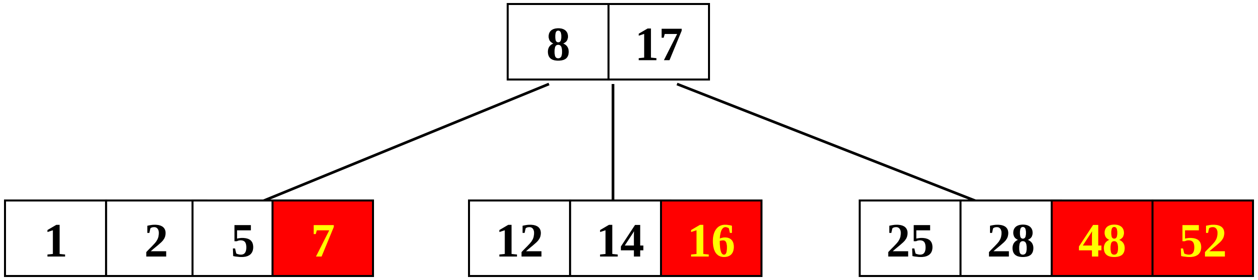
- 1 12 8 2 25 5 14 28 **17**



# Inserting into a B-tree – Example

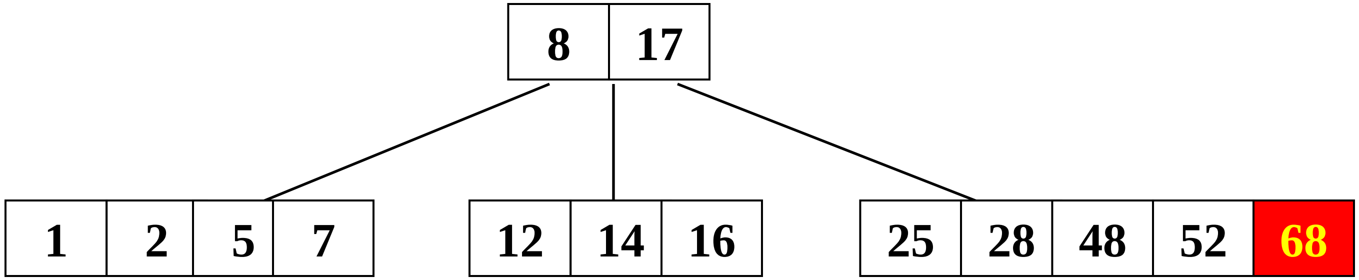
- 1 12 8 2 25 5 14 28 17 **7 52 16 48**

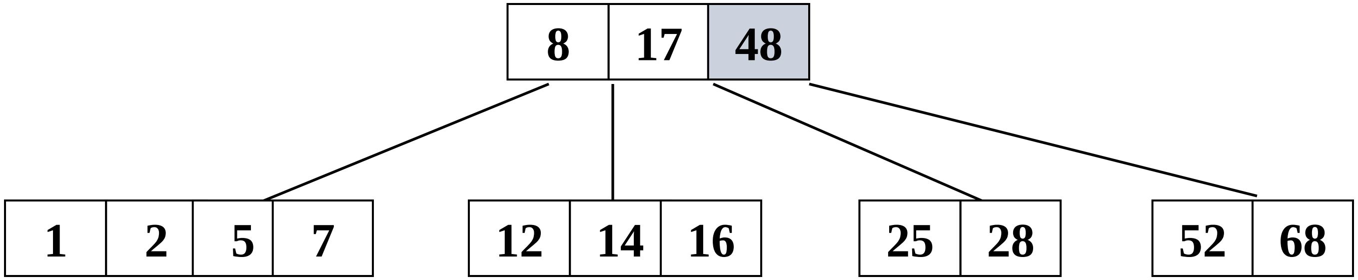


# Inserting into a B-tree – Example

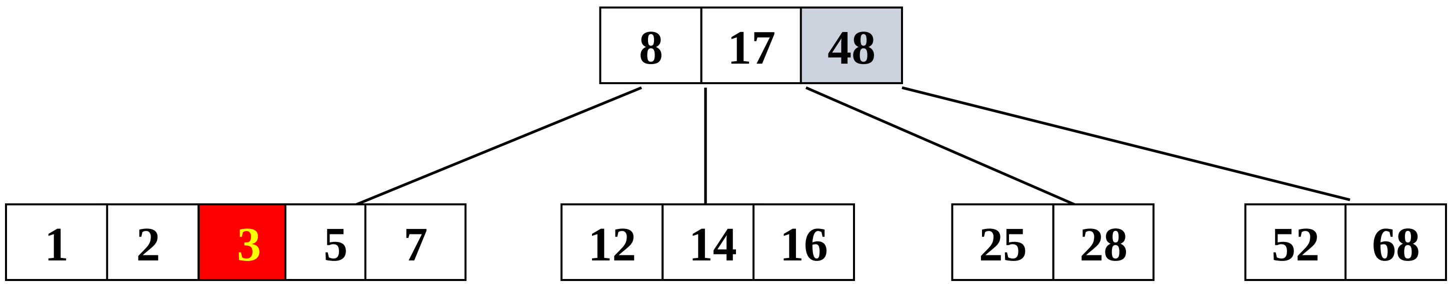
- 1 12 8 2 25 5 14 28 17 7 52 16 48 **68**

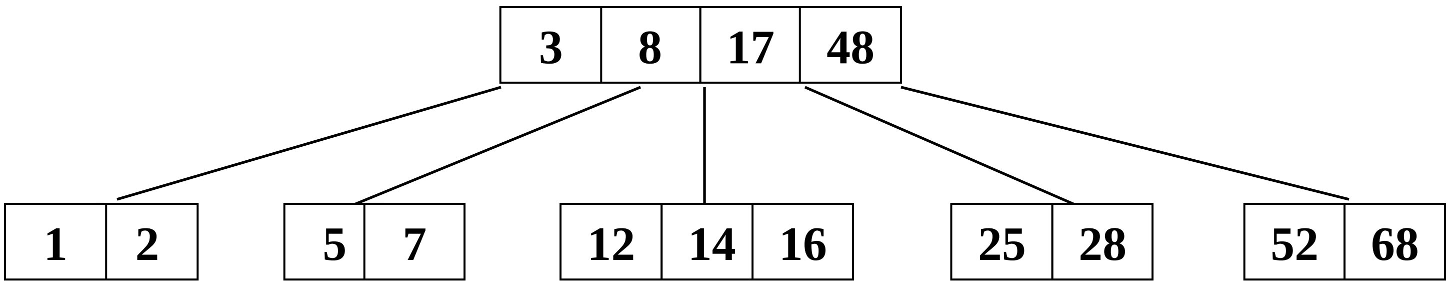


# Inserting into a B-tree – Example

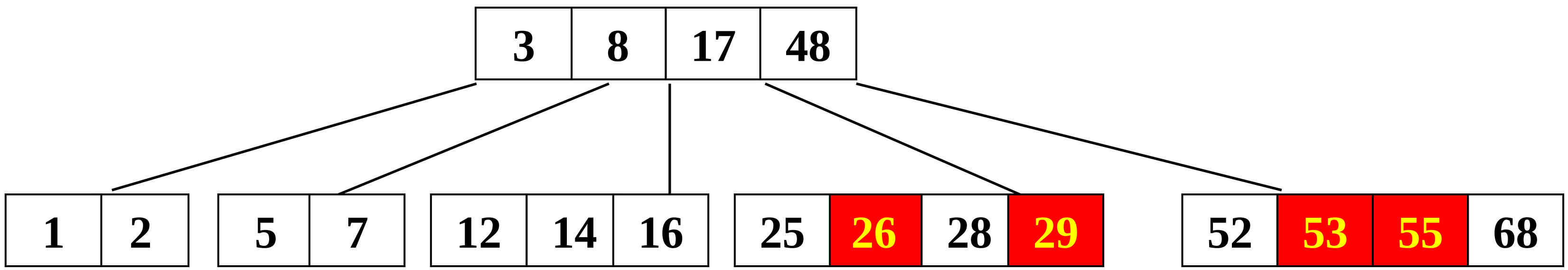
- 1 12 8 2 25 5 14 28 17 7 52 16 48 68 **3**



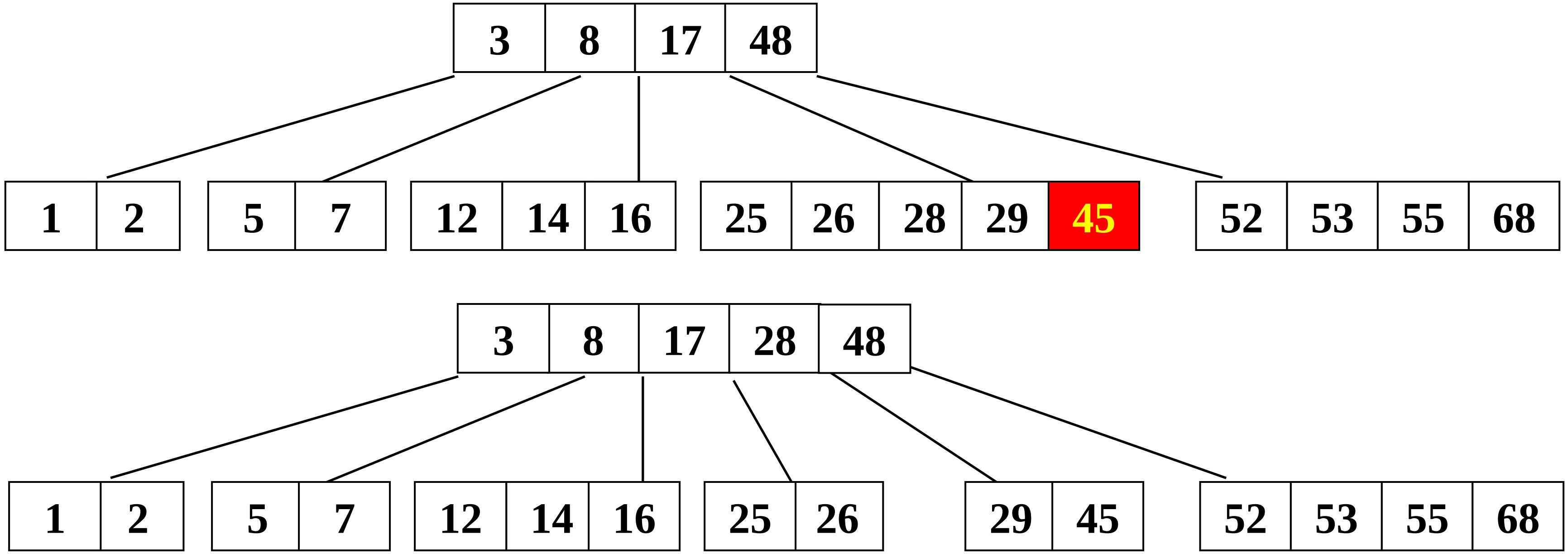
# Inserting into a B-tree – Example

- 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 **26 29 53 55**



# Inserting into a B-tree – Example

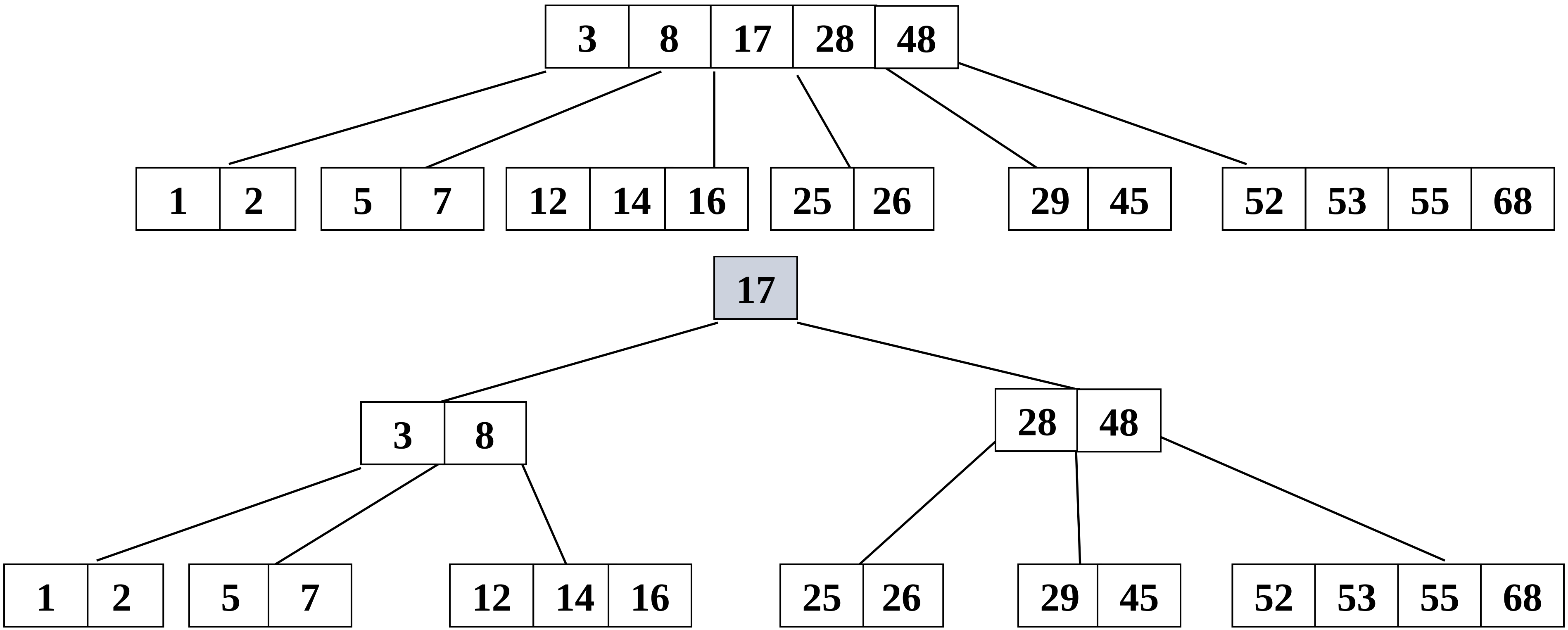
- 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45




Insert **45**

# Inserting into a B-tree – Example

- 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 **45**



# Removal from a B-tree

- Remove a key *k*

1. If *k* is in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove *k*.
2. If *k* is NOT in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete *k* and promote the predecessor or successor of *k* to *k*'s position.

# Removal from a B-tree (2)

- (1) & (2) may lead to a leaf node *L* has less than min. number of keys

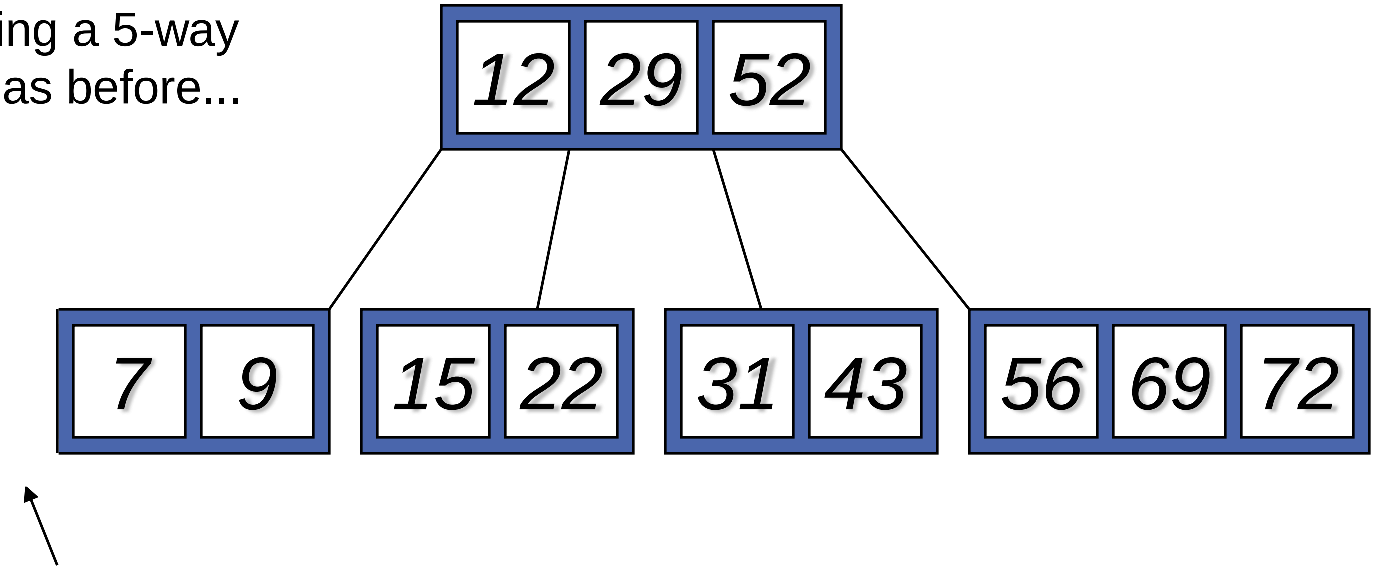
→ Look at the siblings immediately adjacent to the leaf

  - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into *L*
  - 4: otherwise, combine *L* and one of its neighbours with their shared parent, repeat the process up to the root, if required

$$\boxed{O(t \log_t n)}$$

# Type #1: Simple leaf deletion

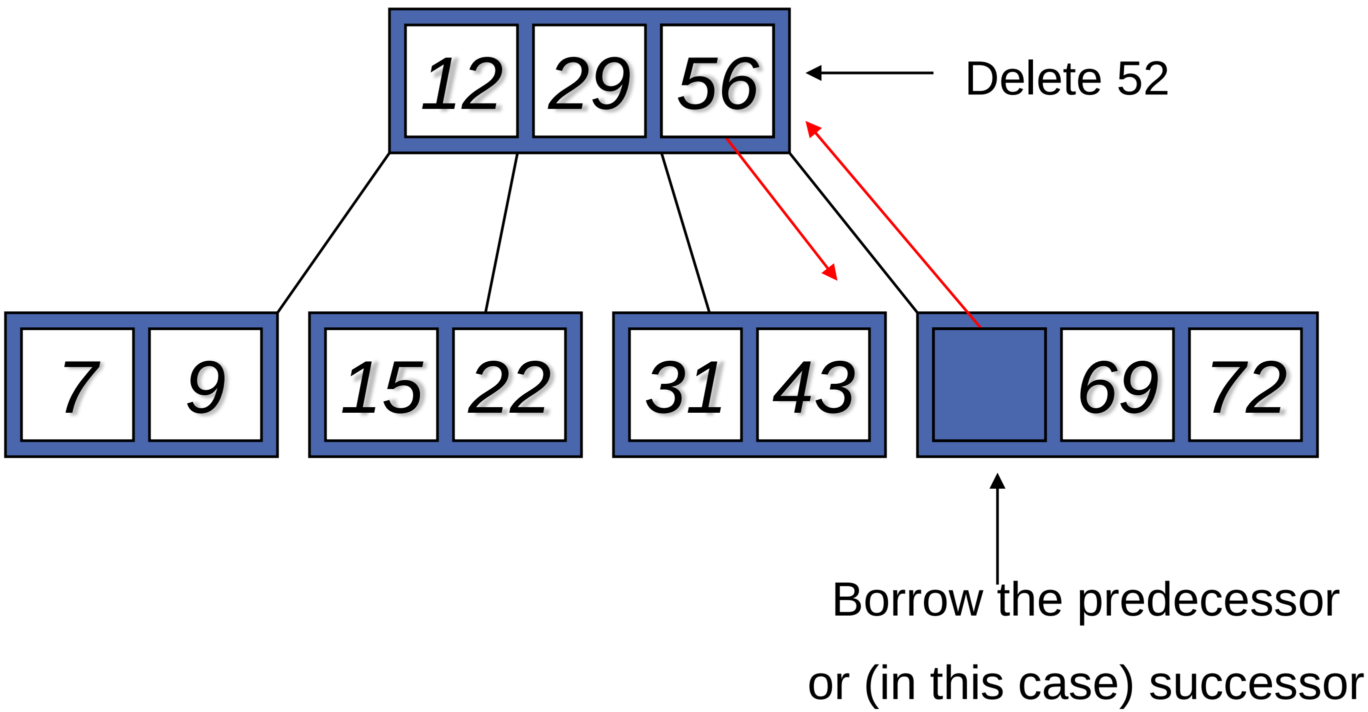
Assuming a 5-way B-tree, as before...




Delete 2: Since there are enough keys in the node, just delete it

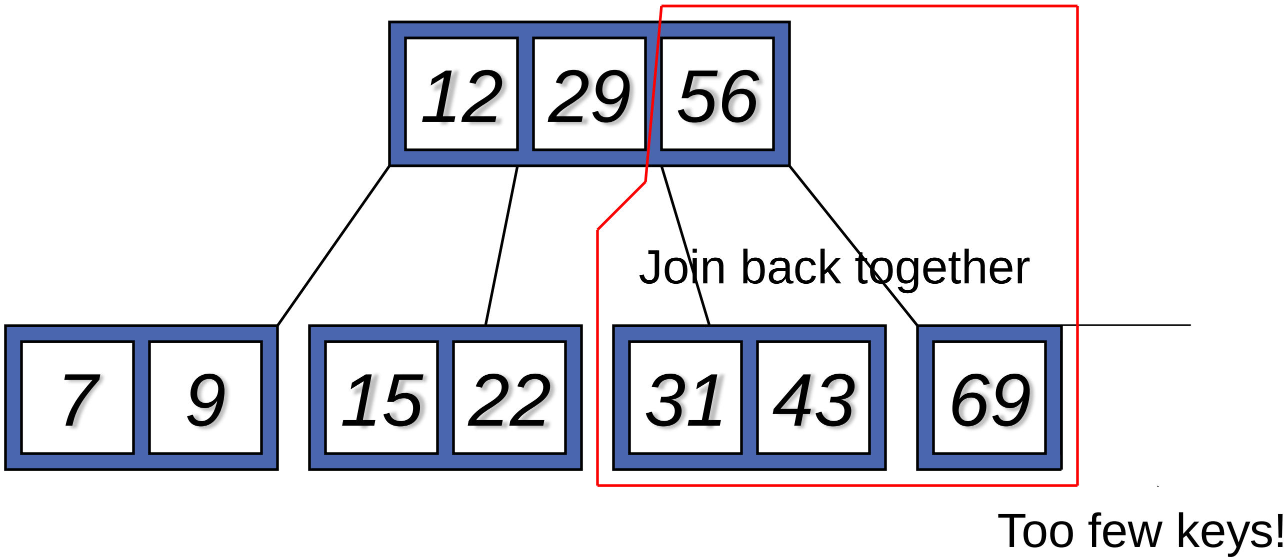
*Note when printed: this slide is animated*

# Type #2: Simple non-leaf deletion



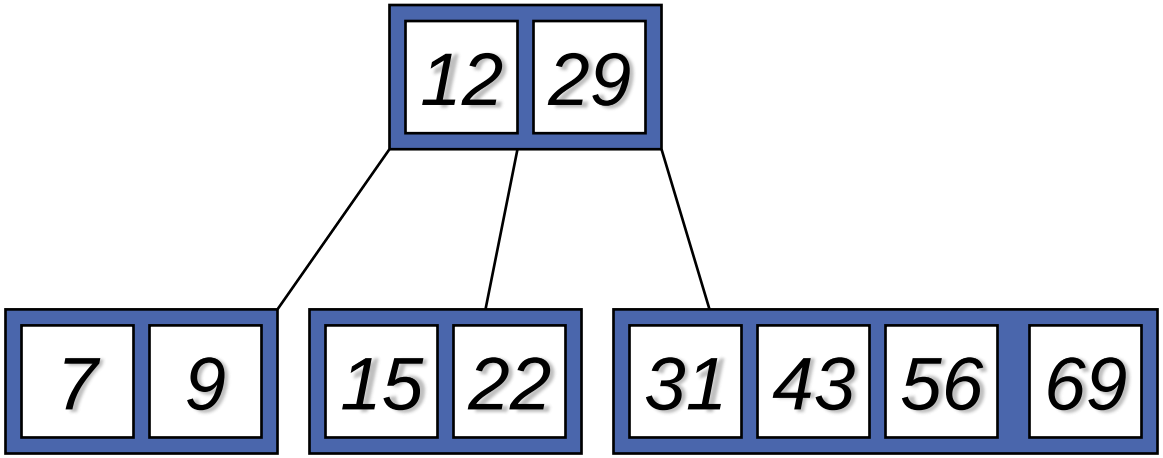

*Note when printed: this slide is animated*

# Type #4: Too few keys in node and its siblings



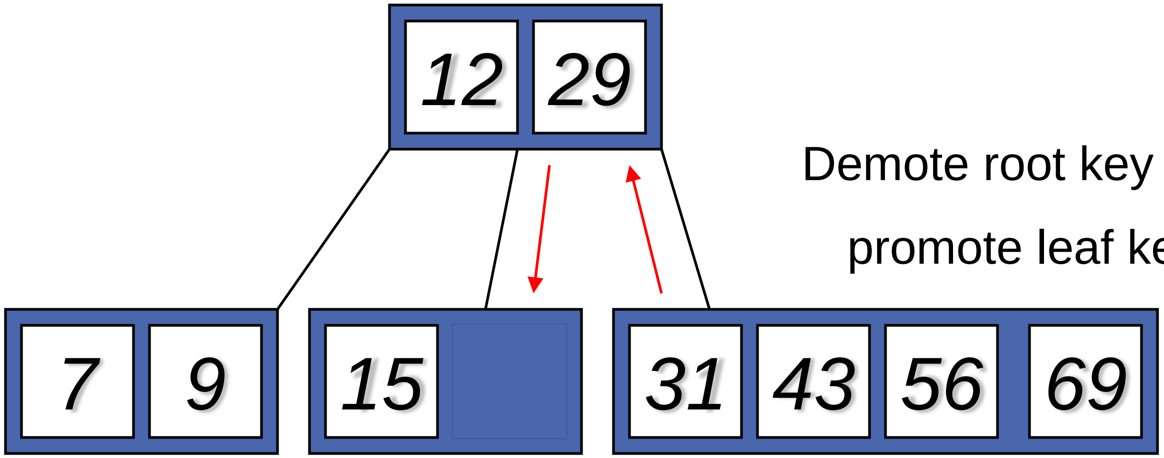

*Note when printed: this slide is animated*

# Type #4: Too few keys in node and its siblings




*Note when printed: this slide is animated*

# Type #3: Enough siblings

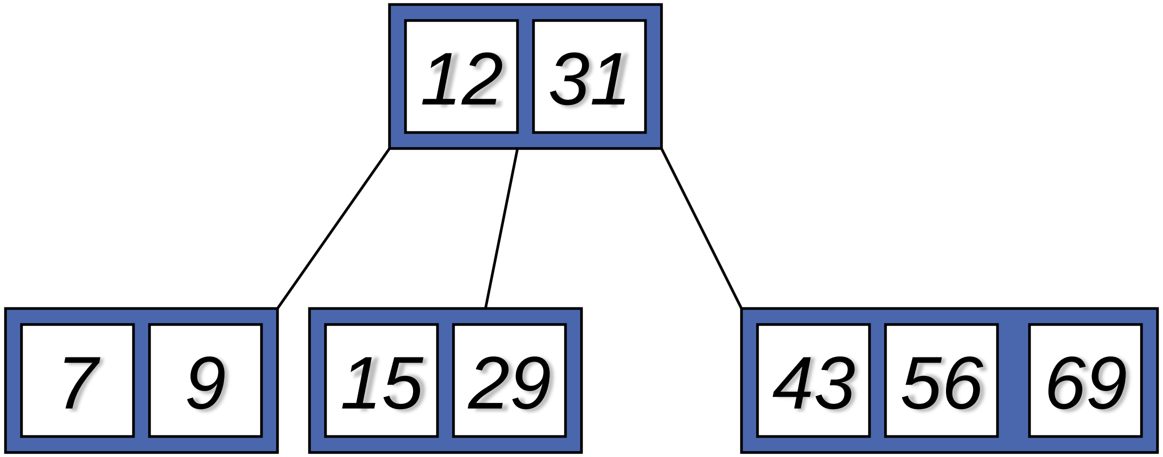



Demote root key and

promote leaf key

*Note when printed: this slide is animated*

# Type #3: Enough siblings




*Note when printed: this slide is animated*

# Exercise in B-tree

- Given 5-way B-tree created by these data:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

- Add these further keys: 2, 6,12
- Delete these keys: 4, 5, 7, 3, 14

# 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) → smallest height is guarantee

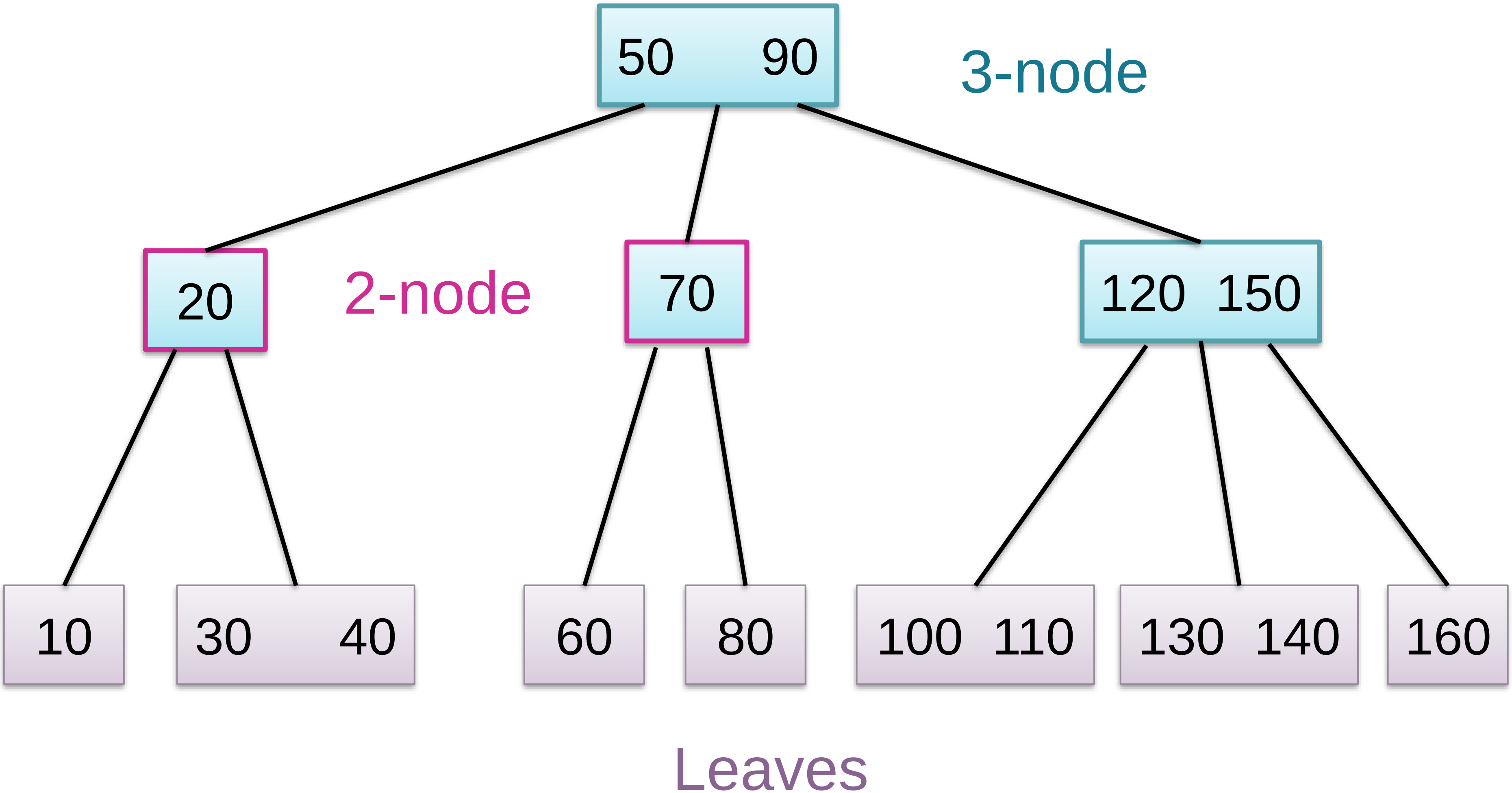
# 2-3 TREES

- Definition
- Traversal & Searching
- Insert & Delete
- Why 2-3 tree?

# Definition of 2-3 tree

- For B-tree, if we take $m = 3$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
- Definition: A **2-3 tree** is a B-tree in which:
  - Each internal node has either **2** or **3** children.
    - A node with 2 children is called a 2-node.
    - A node with 3 children is called a 3-node.
  - All leaves are at the same level (leaf contains 1 or 2 keys)

# 2-3 tree – Example



# Height of a 2-3 tree

- A 2-3 tree is **not** a binary tree.
  - If a 2-3 tree contains only 2 nodes, it is likely a perfect binary tree → $n = 2^{h+1} - 1$
  - If, on the other hand, some of the internal nodes of a 2-3 tree do have 3 children, the tree will contain more nodes than a perfect binary tree of the same height → $n > 2^{h+1} - 1$
- Therefore, height of a 2-3 tree is:

$$h \leq \lceil \log_2(n+1) - 1 \rceil$$

$$\boxed{O(\log_2 n)}$$

# Traversing a 2-3 tree

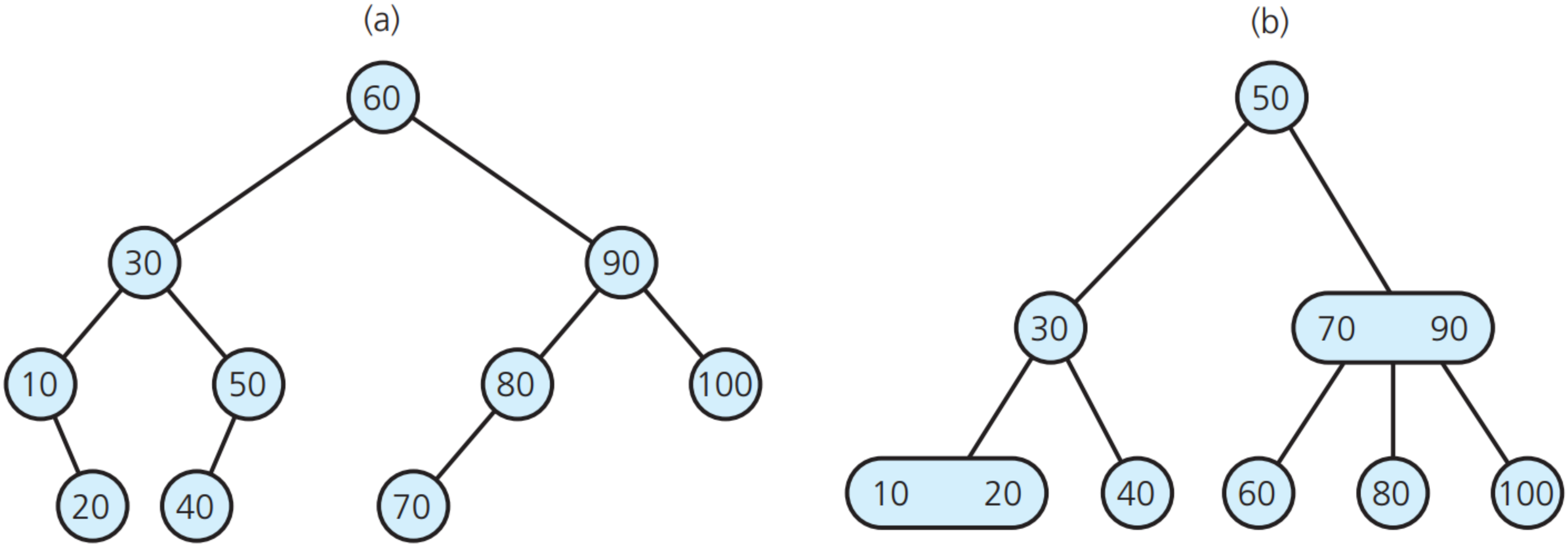
- You can traverse a 2-3 tree in sorted order by performing the analogue of an in-order traversal on a binary tree

# Searching a 2-3 tree

- The ordering of keys in a 2-3 tree is analogous to the ordering for a BST
  - → Searching on a 2-3 tree is efficient & quite similar to a BST
- The same efficiency:
  - A BST with $n$ nodes cannot be shorter than $\lceil \log_2(n+1) - 1 \rceil$
  - A 2-3 tree with $n$ nodes cannot be taller than $\lceil \log_2(n+1) - 1 \rceil$
  - A node in a 2-3 tree has at most 2 keys.
- *Then, why should we use 2-3 tree?*

# Inserting to a 2-3 tree

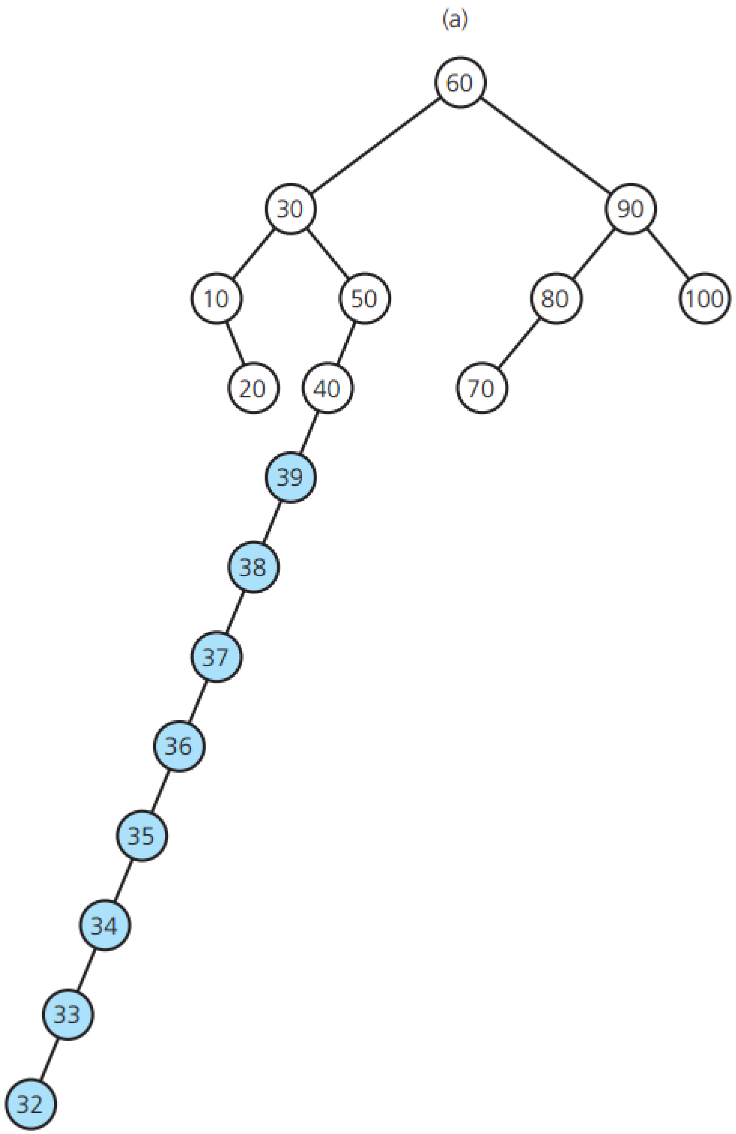
- A balanced BST and a 2-3 tree with the same keys




- Now, insert 39, 38, …, 32 to these trees

# Inserting to a 2-3 tree

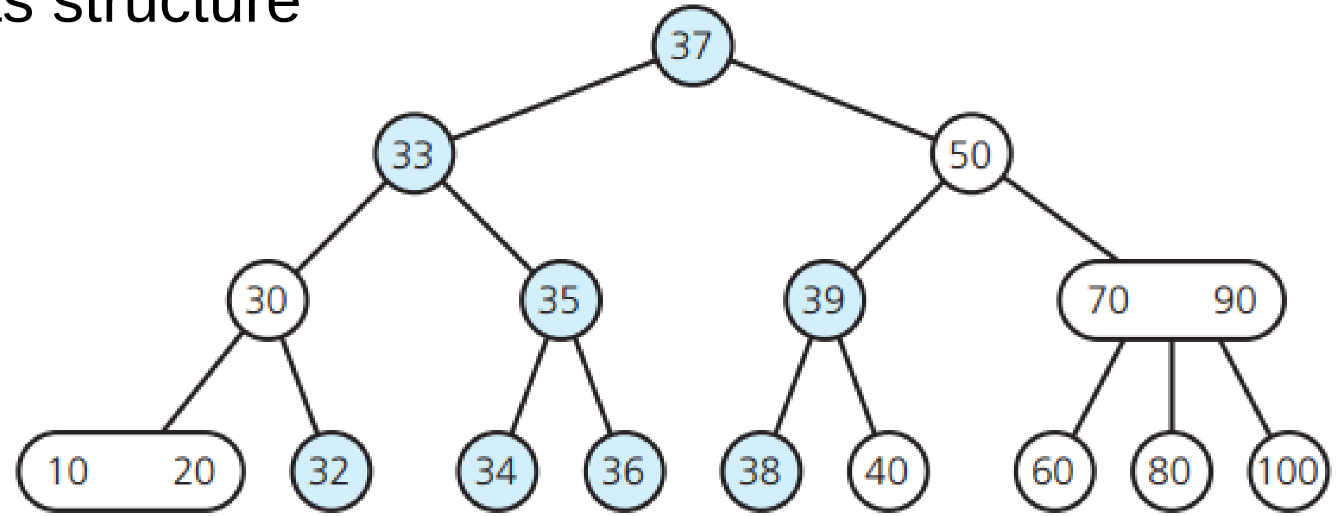
- Now, insert 39, 38, …, 32 to these trees
- Binary Search Tree
  - Quickly loose its balance



# Inserting to a 2-3 tree

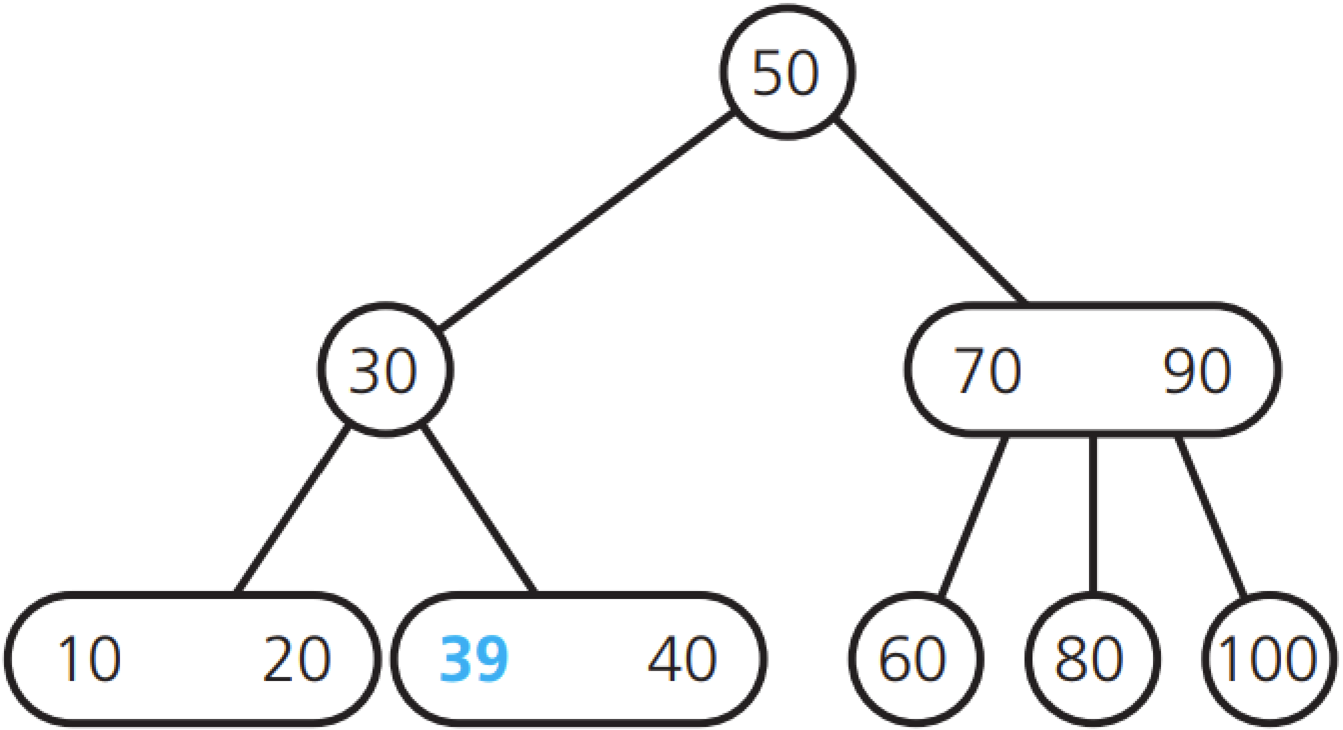
- Now, insert 39, 38, …, 32 to these trees
- 2-3 tree:
  - Retains its structure

(b)



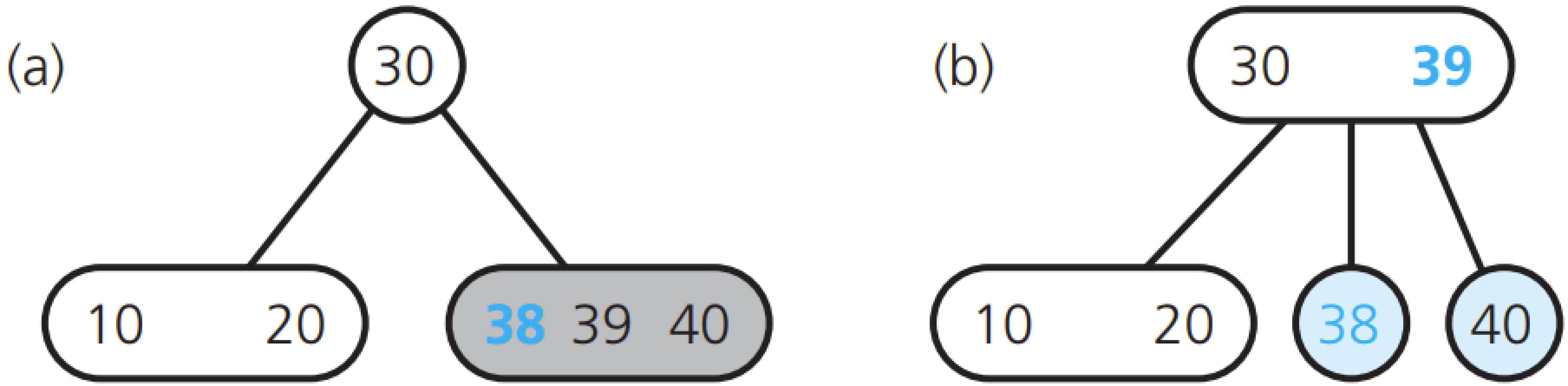
# Inserting to a 2-3 tree

- Insert 39



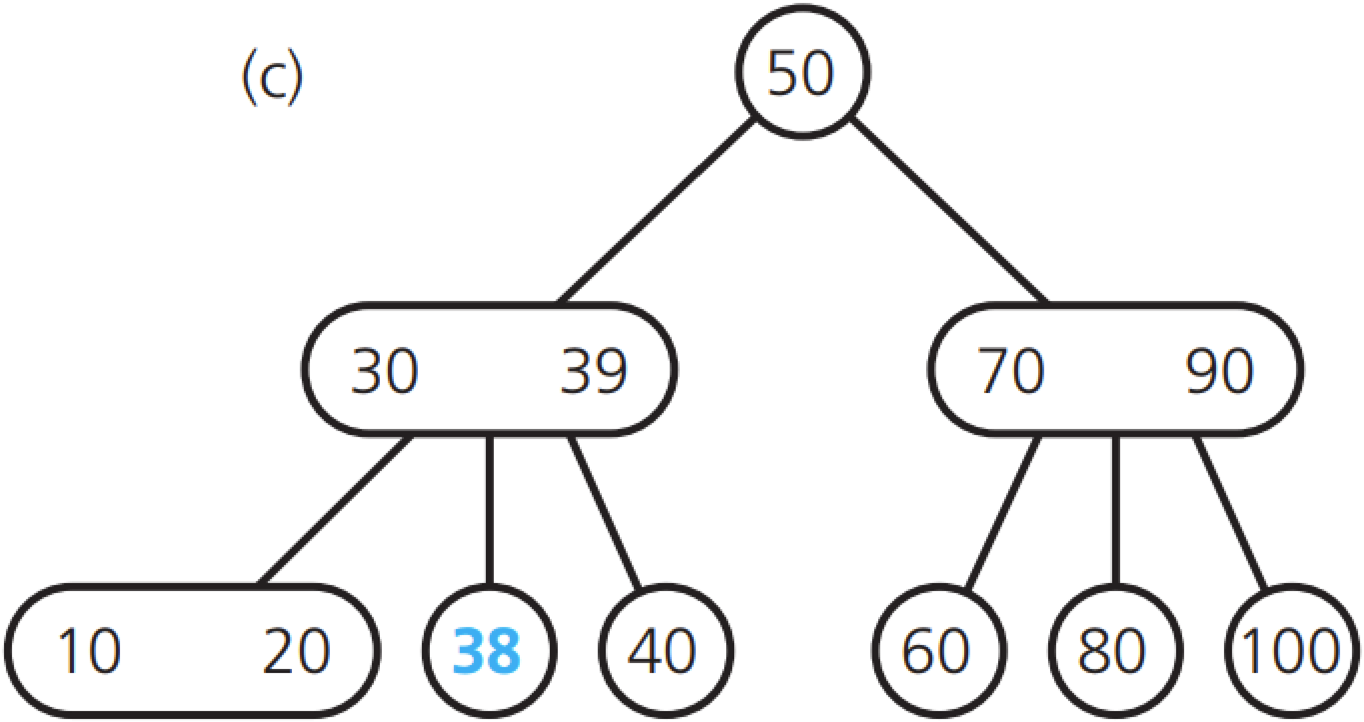
# Inserting to a 2-3 tree

- Insert 38



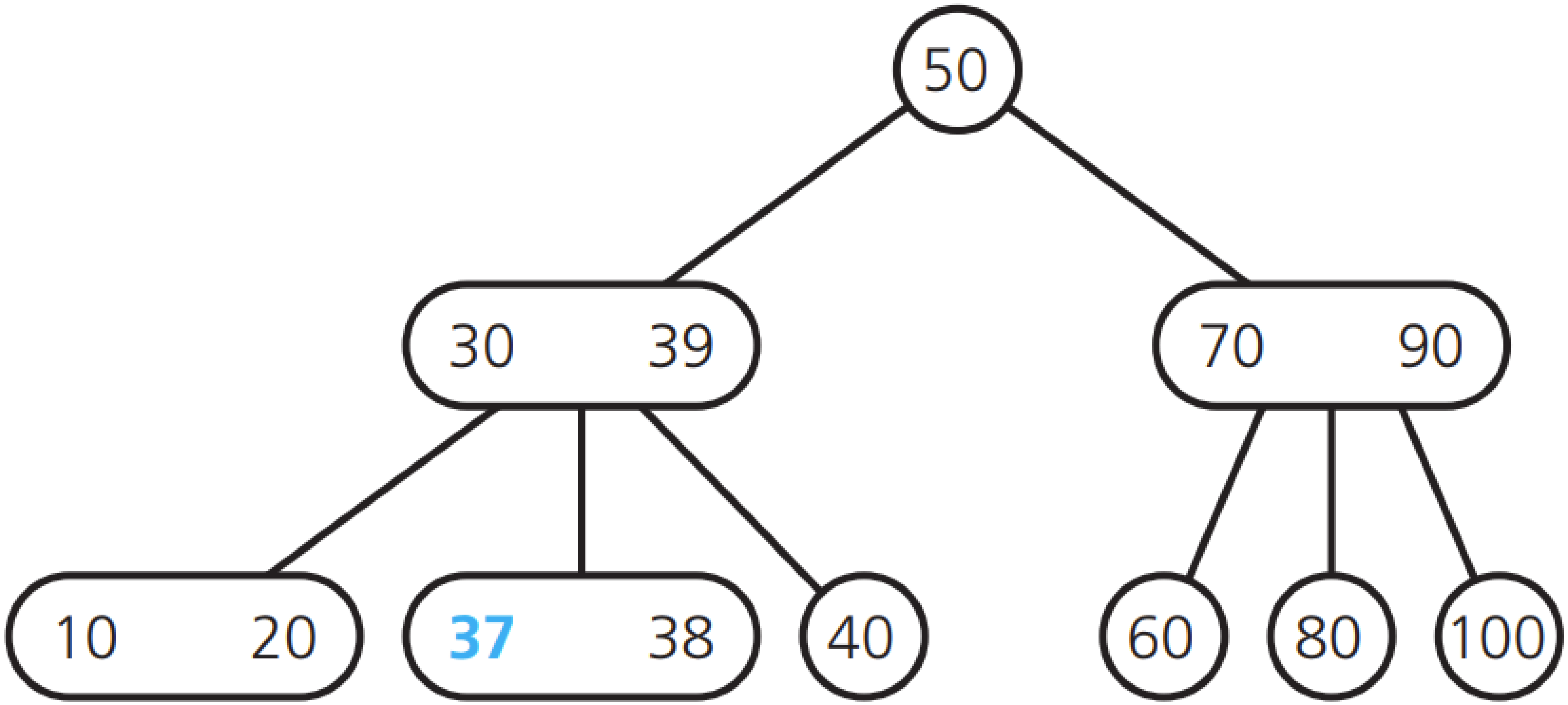
# Inserting to a 2-3 tree

- Insert 38



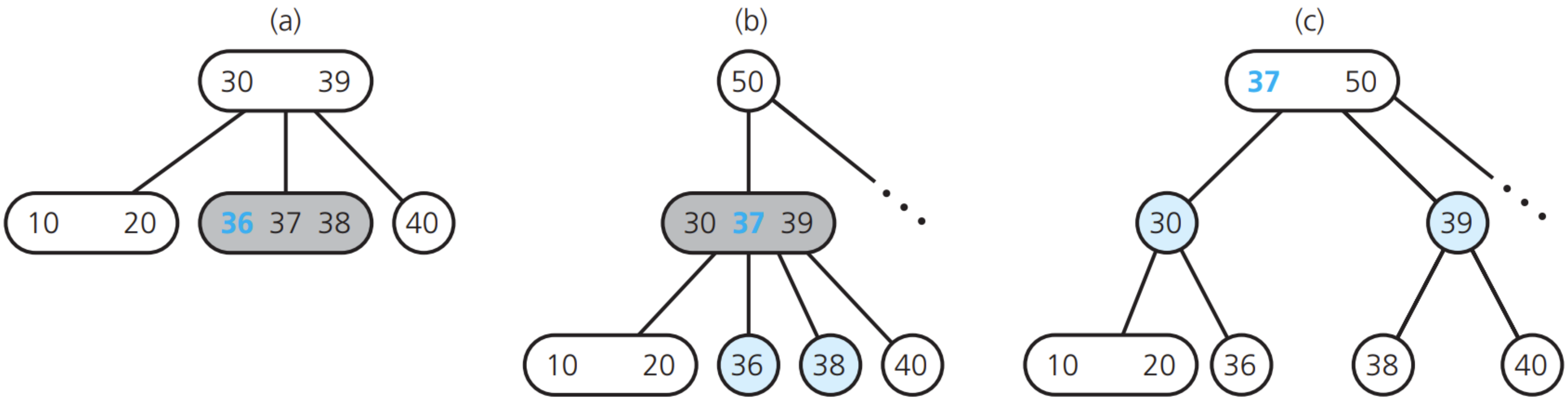
# Inserting to a 2-3 tree

- Insert 37



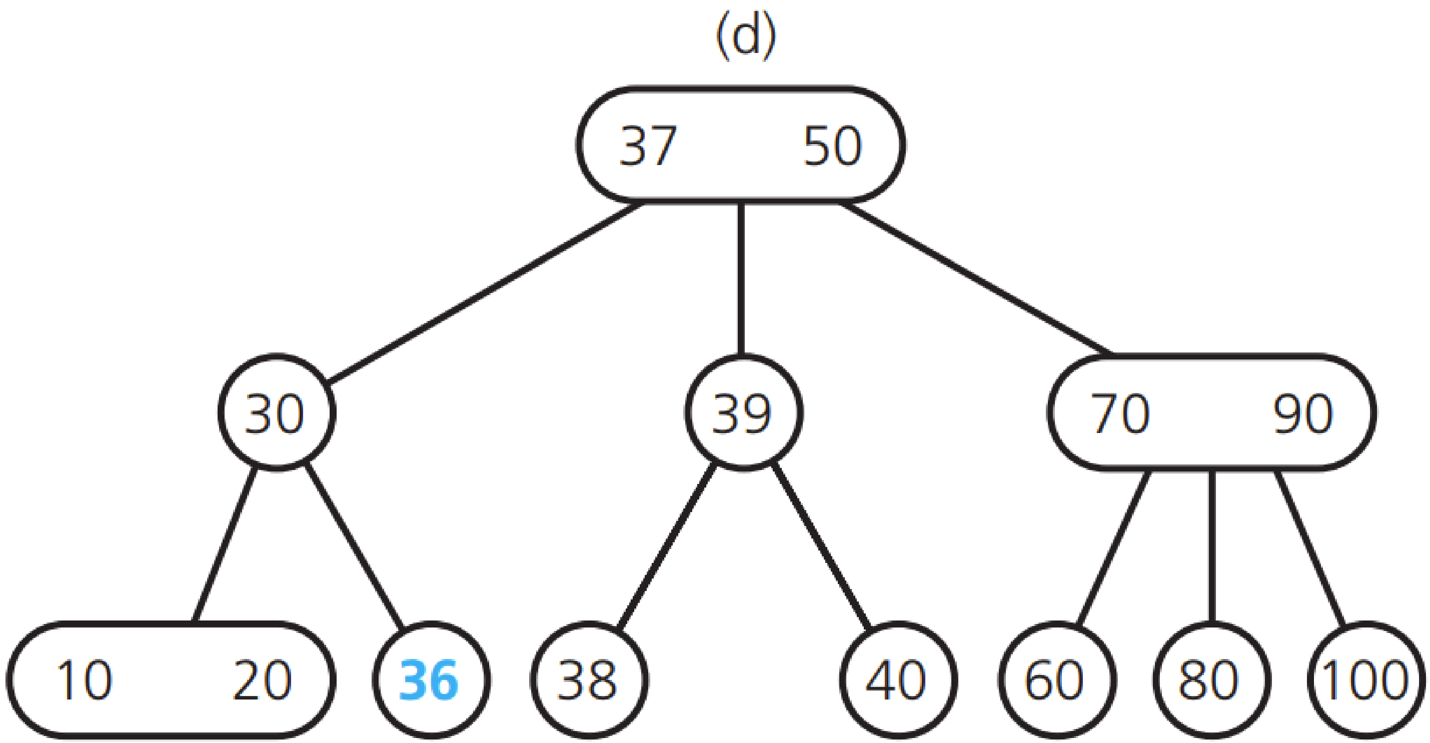
# Inserting to a 2-3 tree

- Insert 36



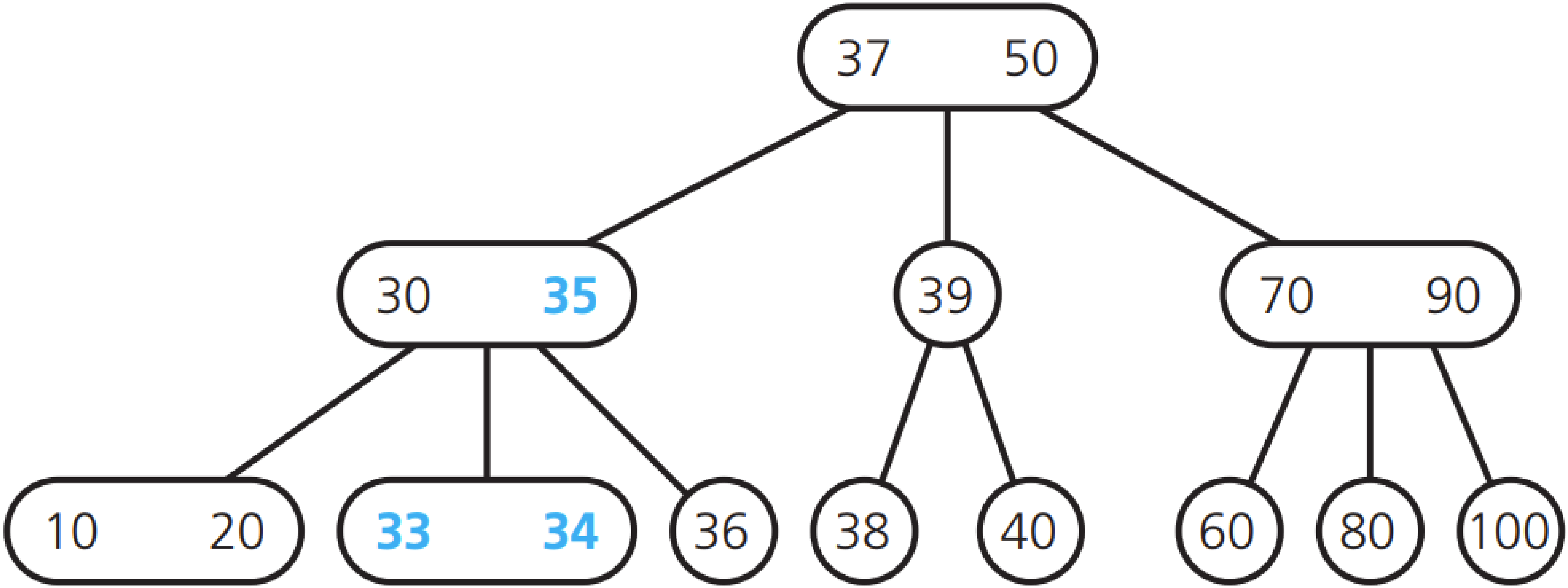
# Inserting to a 2-3 tree

- Insert 36



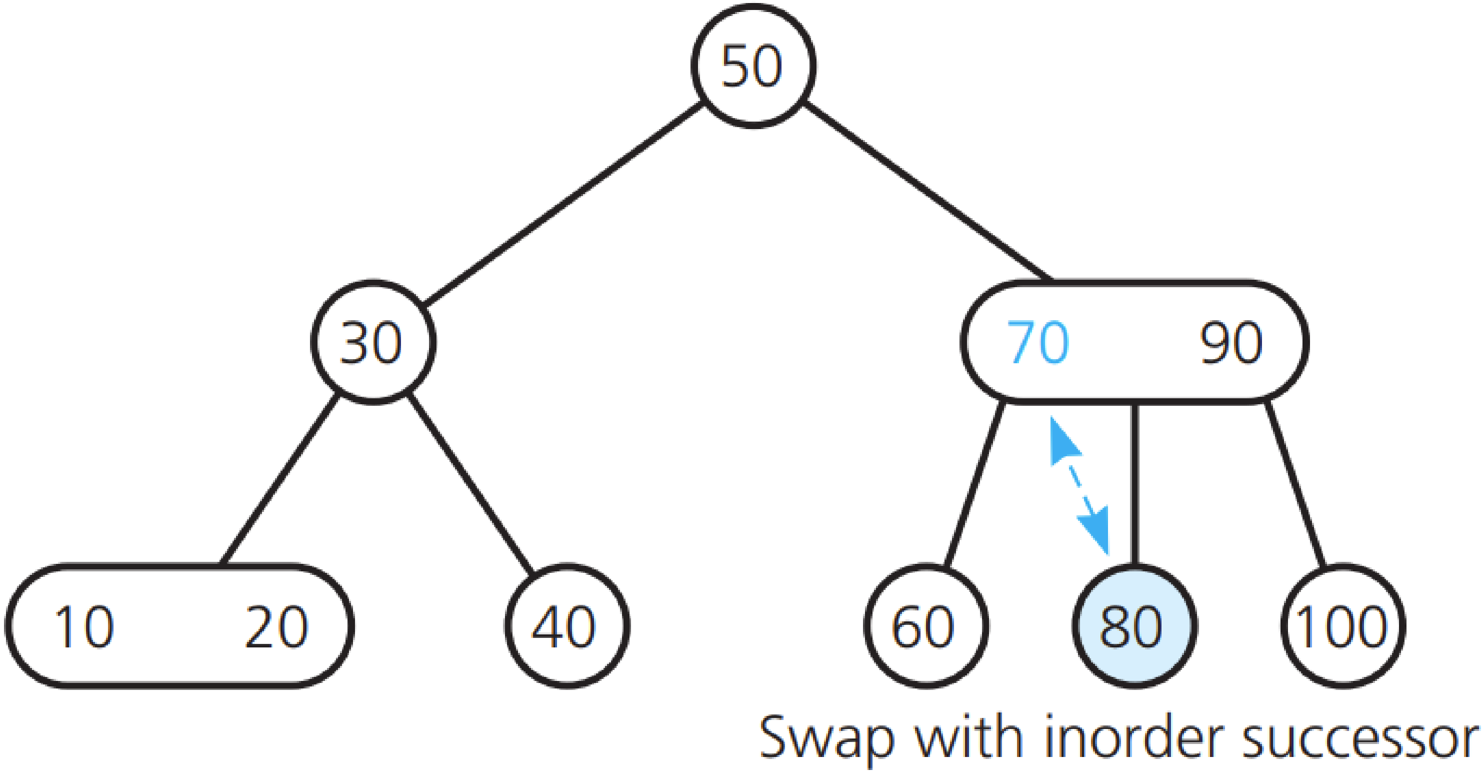
# Inserting to a 2-3 tree

- Insert 35, 34, 33



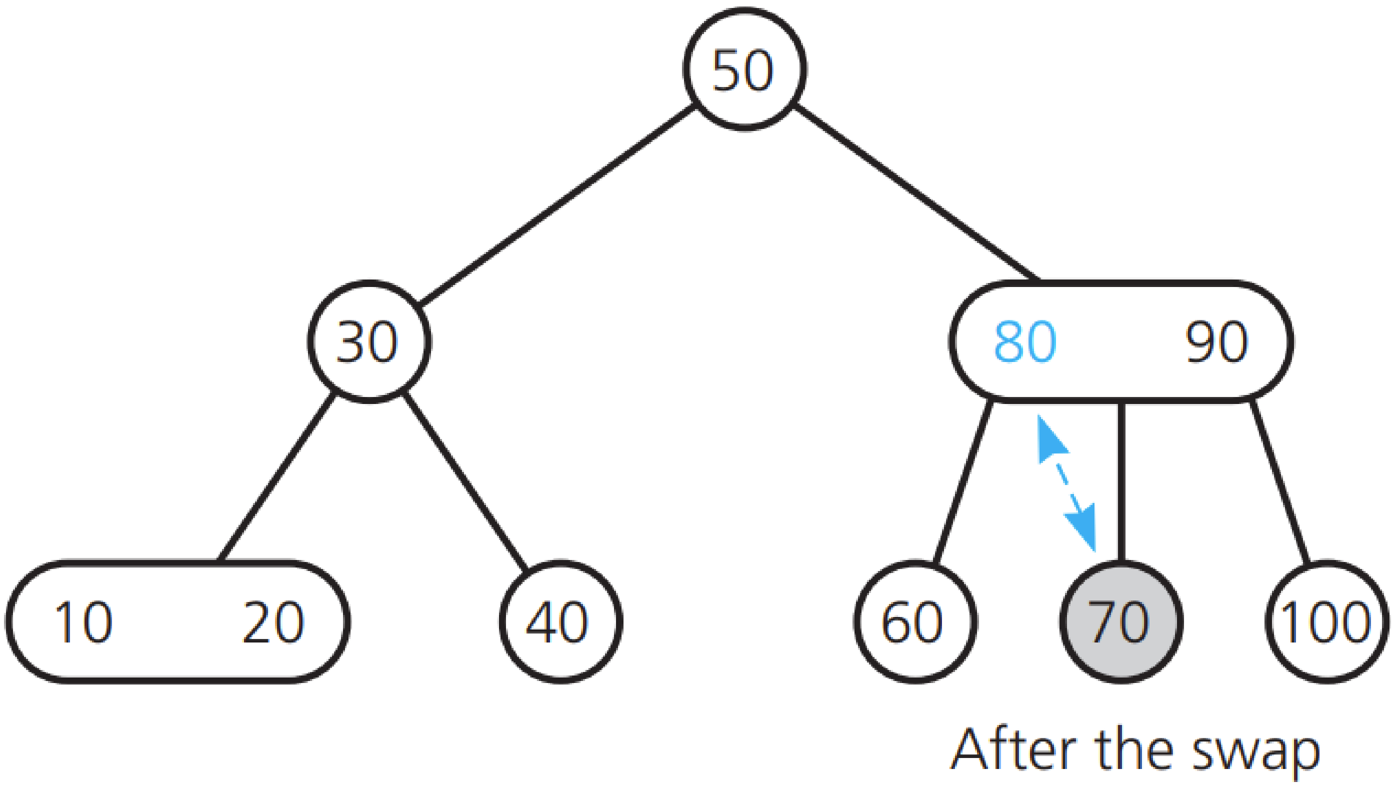
# Deleting from a 2-3 tree

- Delete 70:



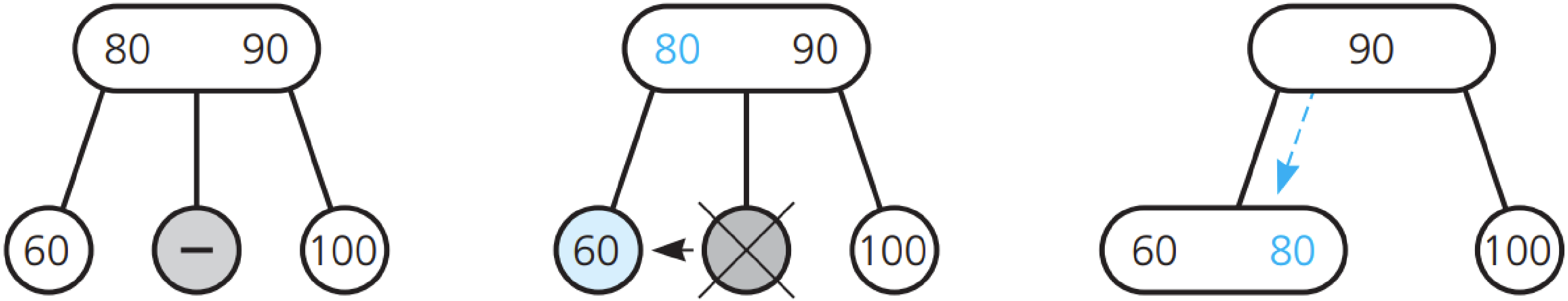
# Deleting from a 2-3 tree

- Delete 70:



# Deleting from a 2-3 tree

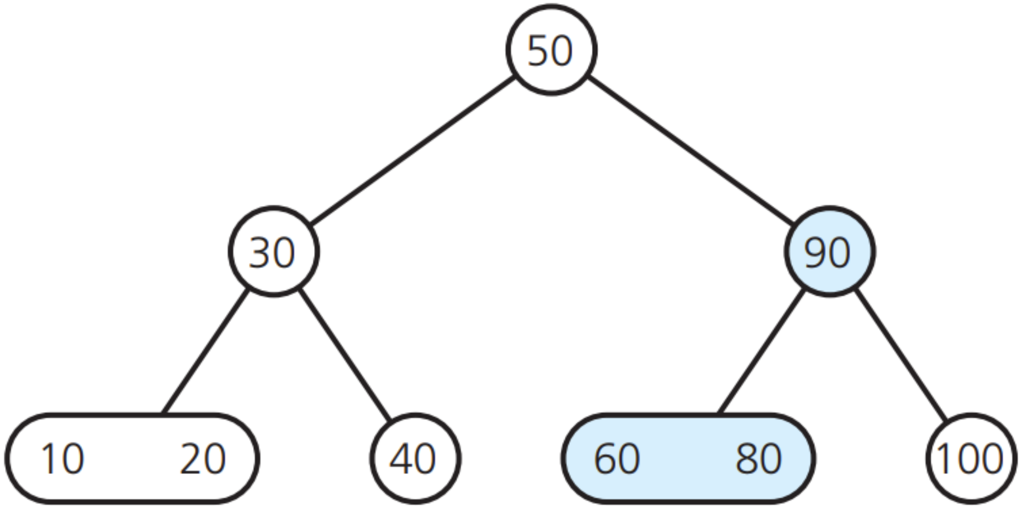
- Delete 70:




Delete value from leaf

Merge nodes by deleting empty leaf and moving 80 down

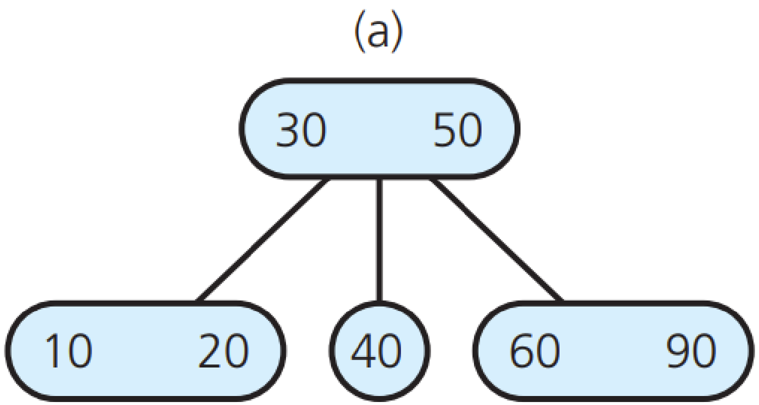
Result tree



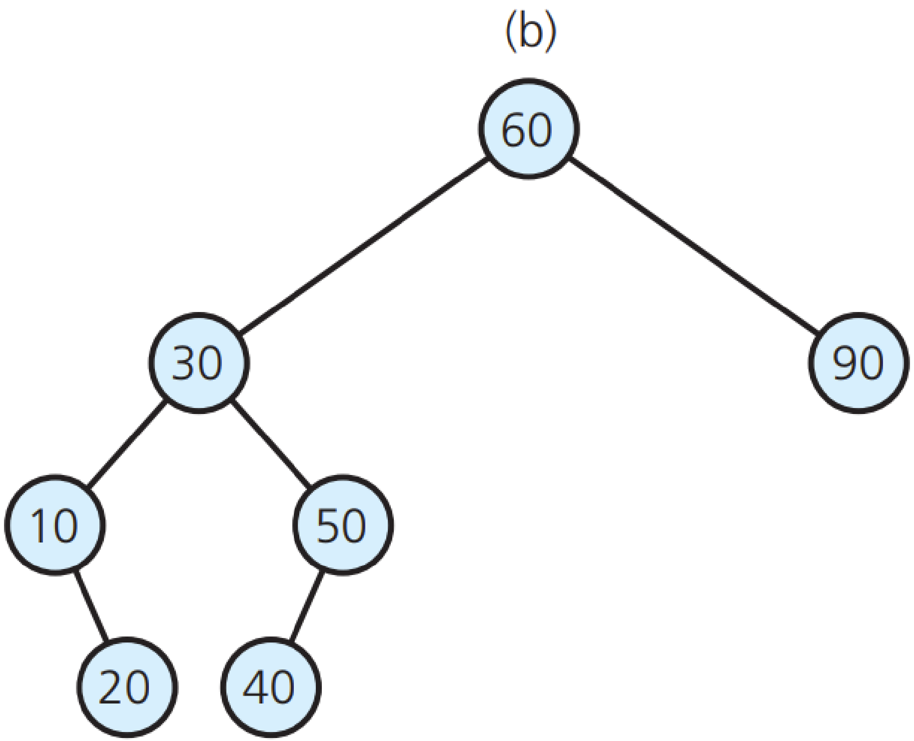
# Deleting from a 2-3 tree

- Delete 70, 100, 80:

B-tree




Binary Search Tree



# Again, Why 2-3 tree?

- Searching: sometimes you have to make 2 comparisons to get pass a 3-node.
  - However, it is still $O(\log n)$ running time
- Insertion/Deletion needs extra work: split nodes, merge nodes.
  - However, this extra work is not a real concern
  - It is easier to keep the tree balanced than a normal BST.

# 2-3-4 TREES

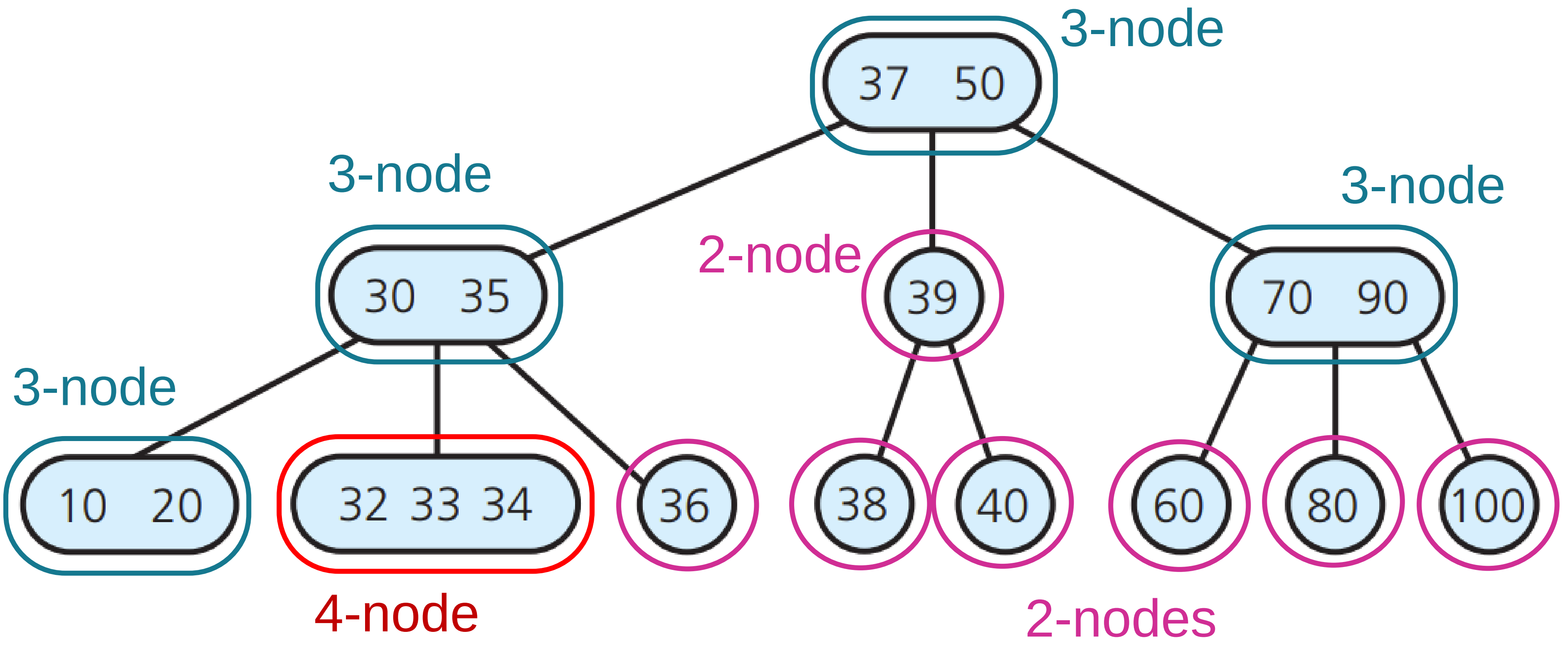
# Definition of 2-3-4 tree

- If a 2-3 tree is good, are trees with more than 3 children better?
- Definition: A 2-3-4 tree is a B-tree in which:
  - Each internal node has either 2, 3 or 4 children.
    - 2-node: node contains 1 key
    - 3-node: node contains 2 keys
    - 4-node: node contains 3 keys

    *Different with 2-3 tree*
  - All leaves are at the same level, each contains 1 to 3 keys

# Example of 2-3-4 tree

- 2-3-4 tree of the same data as the previous 2-3 tree



# Searching and Traversing a 2-3-4 tree

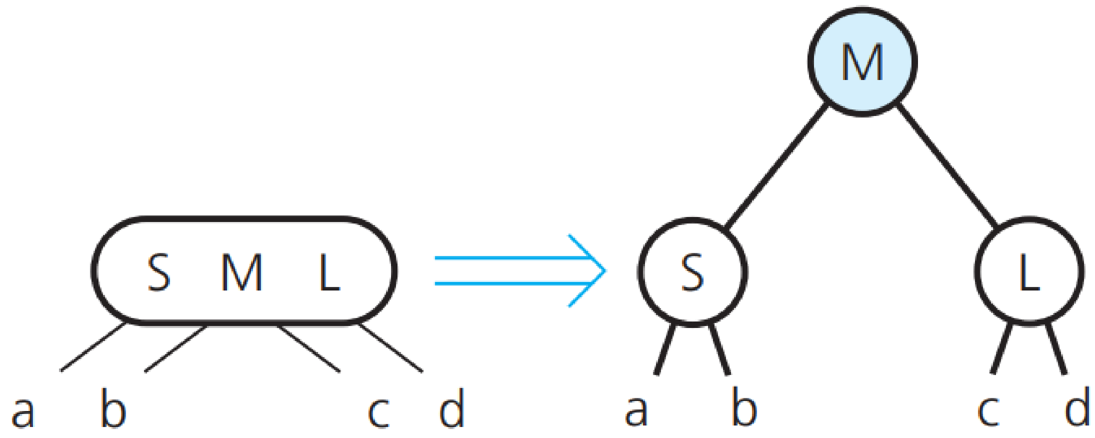
- Traversing & Searching are the same as in B-tree (or 2-3 tree)
- Example:
  - Search for 31 on the Example 2-3-4 tree:
    - Search the left subtree since 31 < 37
    - Search the middle subtree of the node <30 35> since 31> 30 and 31 < 35
    - Terminate the search at the left child pointer of <32 33 34> since 31 < 32
    - Result: 31 is not in the tree.

# Inserting to 2-3-4 tree

- For inserting a key to a 2-3 tree, the search algorithm traces a path from the root to a leaf and then backs up from the leaf as it splits nodes.
- For a 2-3-4 tree: splits 4-node as soon as it encounters them on the way down from the root to the leaf. → *Avoid return path*
  - Each 4-node either:
    1. Be the root, or
    2. Have a 2-node parent, or
    3. Have a 3-node parent

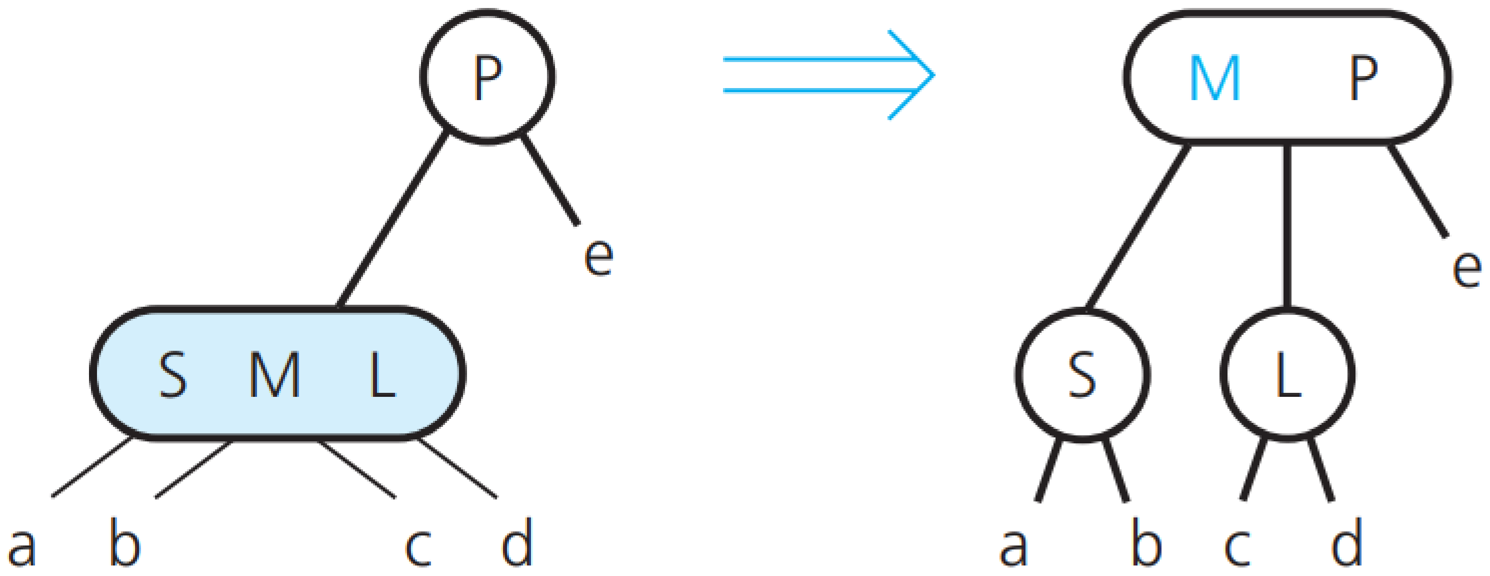
# Splitting a 4-node in 2-3-4 tree

1. The 4-node is the root



# Splitting a 4-node in 2-3-4 tree

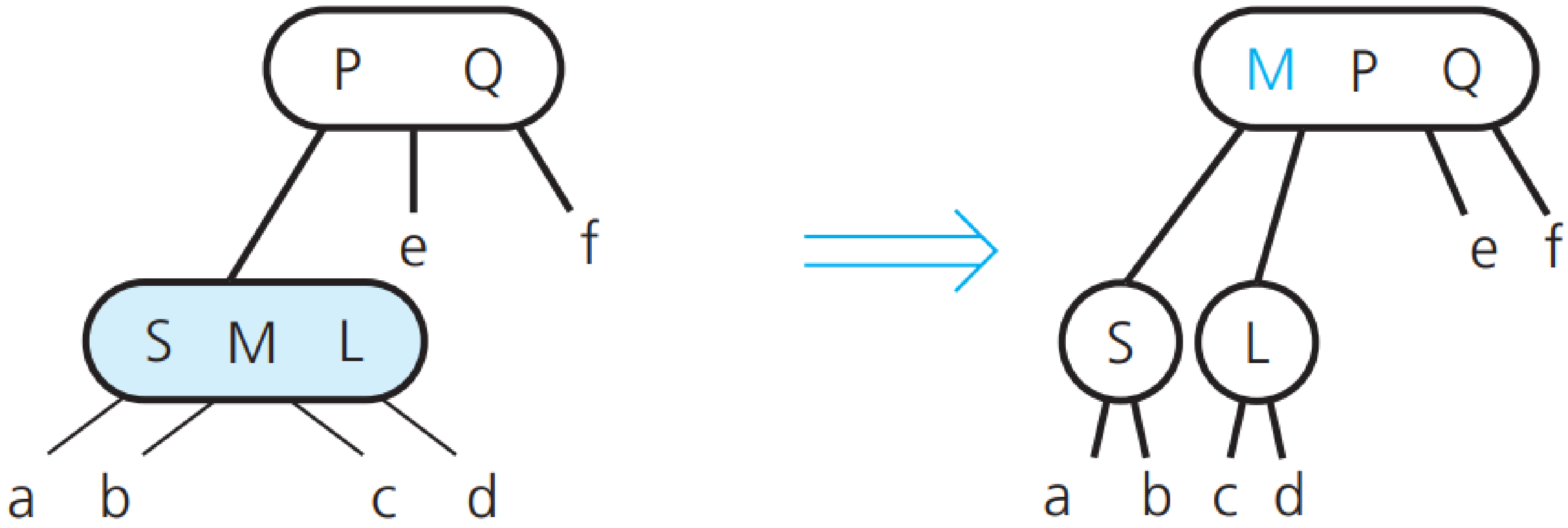
2. The 4-node has a 2-node parent:
   - 4-node is left child:




   - 4 node is right child?

# Splitting a 4-node in 2-3-4 tree

3. The 4-node has a 3-node parent
   - 4-node is left child:

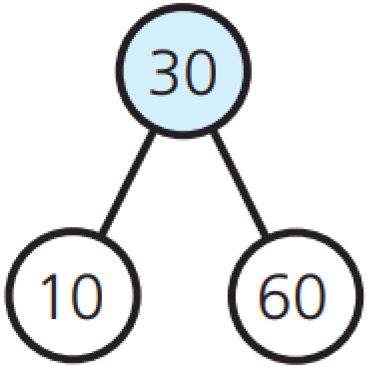



   - 4-node is middle child?
   - 4-node is right child?

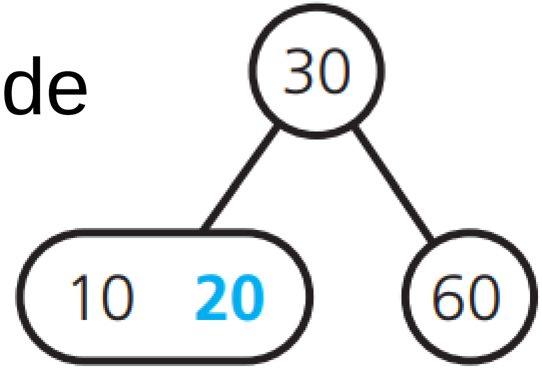
# Inserting to 2-3-4 tree

- Insert **20** into a one-node 2-3-4 tree


- This is a 4-node → split it

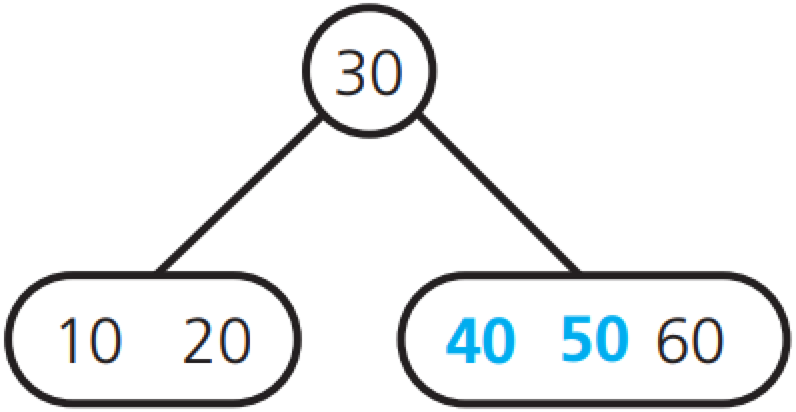



- Then, insert to the leaf node

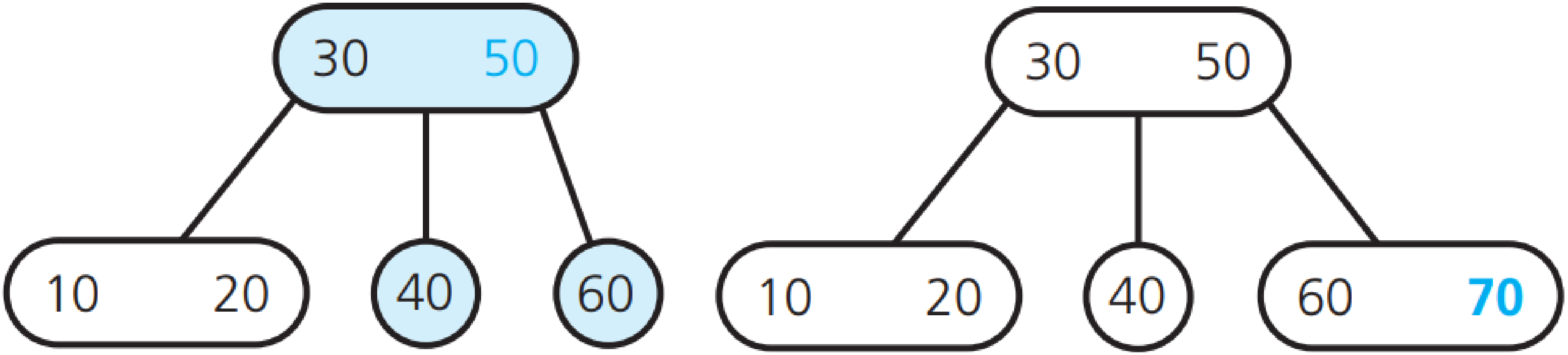


# Inserting to 2-3-4 tree

- Insert **50, 40** do not require split nodes

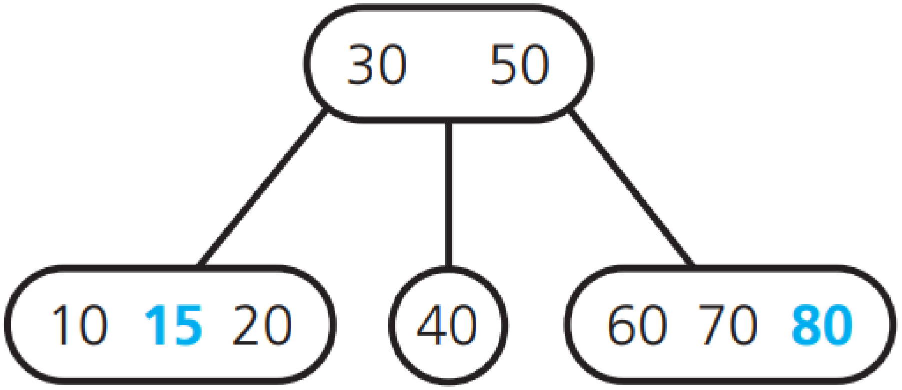



- Insert **70** → split the <40 50 60> node, then insert

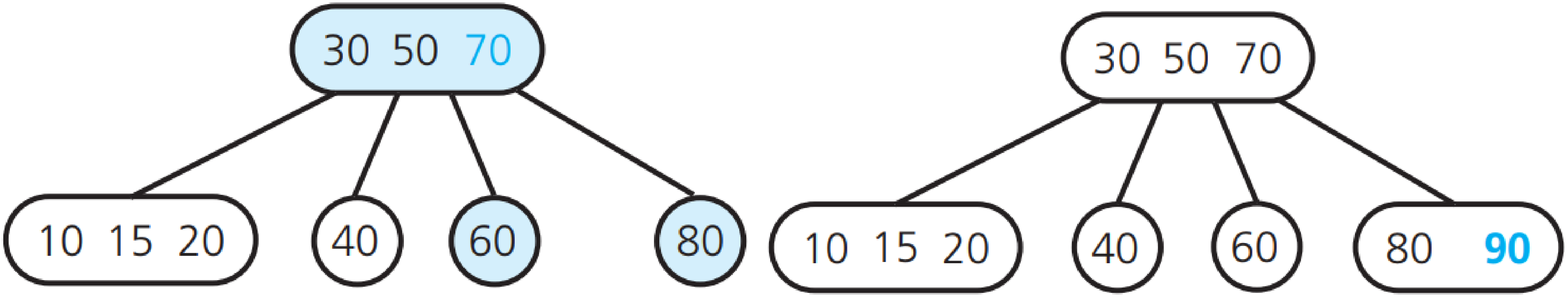


# Inserting to 2-3-4 tree

- Insert **80, 15** do not require split nodes

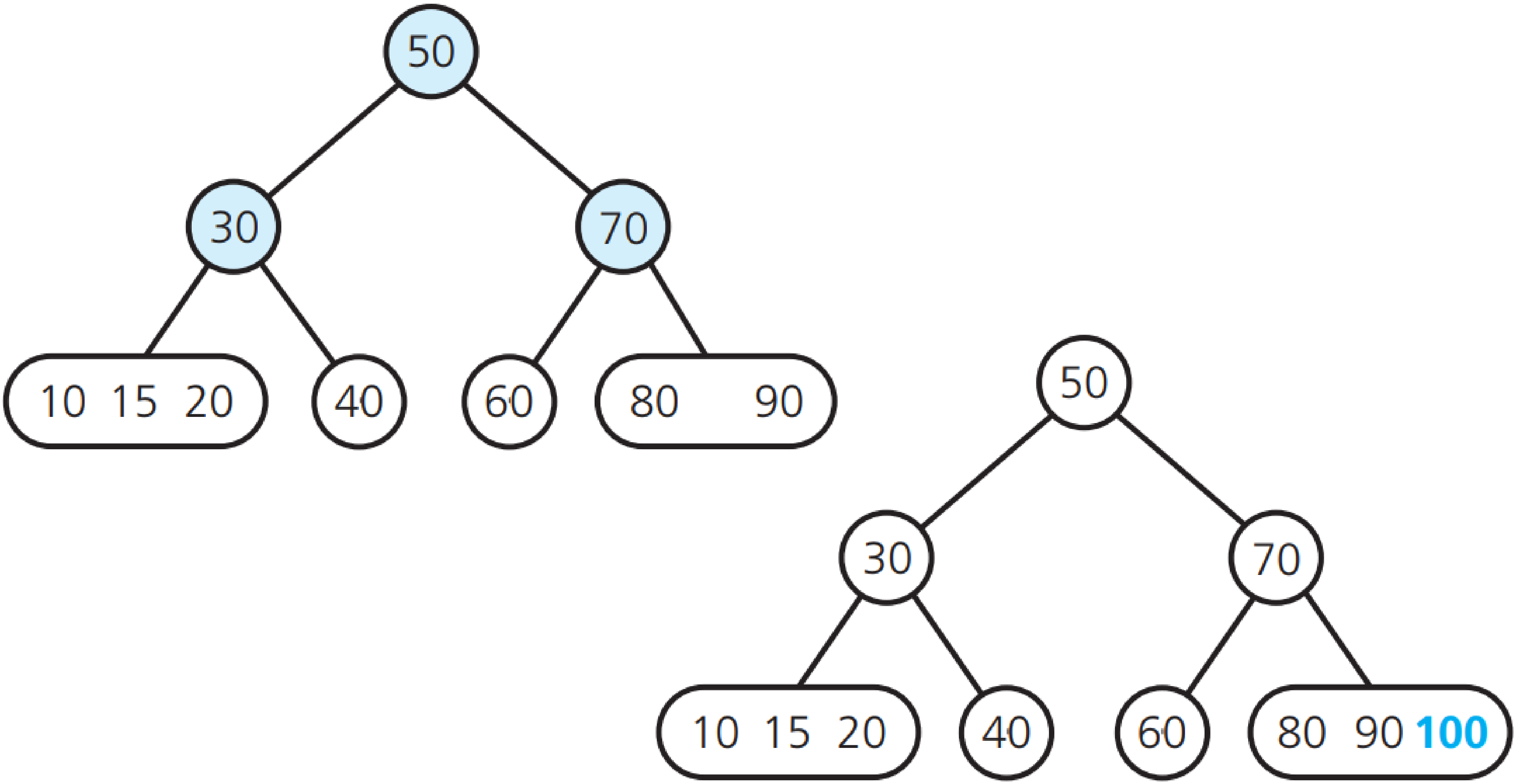



- Insert **90** → split the <60 70 80> node, then insert



# Inserting to 2-3-4 tree

- Insert **100** → split the <30 50 70> node, then insert



# Deleting from a 2-3-4 tree

- Similar to deleting a key from a 2-3 tree:
  - Locate the node *i* that contains the key *k* to remove
  - Find *k*'s successor (at leaves) and swap with *k*
  - Ensure that *k* doesn't occur in a 2-node so that we can perform one pass removal (unlike removal from a 2-3 tree)

→ *Transform each 2-node that we encounter during the search for k into a 3-node or 4-node*

→ Detail of deleting from a 2-3-4 are left as exercises

# Conclusion

- Pros:
  - A 2-3-4 is balanced
  - Its insertion and removal operations use only one pass from root to leaf.
- Cons:
  - Requires more storage than a binary search tree that contains the same data

→ Red-black tree can represent a 2-3-4 tree that retains the advantages of a 2-3-4 tree without the storage overhead.

# Comparing Trees

- Binary search tree can become *unbalanced* and *lose* their good time complexity
- A 2-3 tree and a 2-3-4 tree are variants of a BST that keeps the tree balanced easily.
- A Red-black tree is a binary tree representation of a 2-3-4 tree that requires less storage than a 2-3-4 tree. Insertions and removals in RB tree are more efficient than 2-3-4 tree.
- AVL tree is strict binary trees that *overcome the balance problem*
- B-tree is balanced search tree designed to work well on disk drivers or other direct-access secondary storage devices

# What’s next?

- After today:
  - Reading
    - B-tree: Textbook 1 chap 18 (page 655~)
    - 2-3 tree: Textbook 2 sec 19.2 (page 569~)
    - 2-3-4 tree: Textbook 2 sec 19.3 (page 585~)
    - 2-3-4 tree to Red-black tree: Textbook 2 sec 19.4 (page 592~)
  - Do homework 6.2
- Next class (August $2^{nd}$):
  - Individual Assignment 4
  - Lecture 7: Graphs

# Q&A