

2-3-4 Trees (BST)

CSE 3318 – Algorithms and Data Structures
Alexandra Stefan
Includes materials from Vassilis Athitsos
University of Texas at Arlington

Search Trees

- "search tree" as a term does **NOT** refer to a specific implementation.
- The term refers to a **family of implementations**, that may have **different properties**.
- We will discuss:
 - Binary search trees (BST).
 - 2-3-4 trees (a special type of a B-tree).
 - Mention briefly: red-black trees, AVL trees, splay trees, B-trees and other variations.
- Main operations in search trees: *search, insert and delete*
- Insertions and deletions can differ among trees, and have important implications on overall performance.
- The main goal is to have insertions and deletions that:
 - Are *efficient* (at most logarithmic time).
 - *Leave the tree balanced*, to support efficient search (at most logarithmic time).

2-3-4 Trees

Nodes:

- 2-node : 1 item, 2 children
- 3-node: 2 items, 3 children
- 4-node: 3 items, 4 children

(no missing children in any of these nodes)

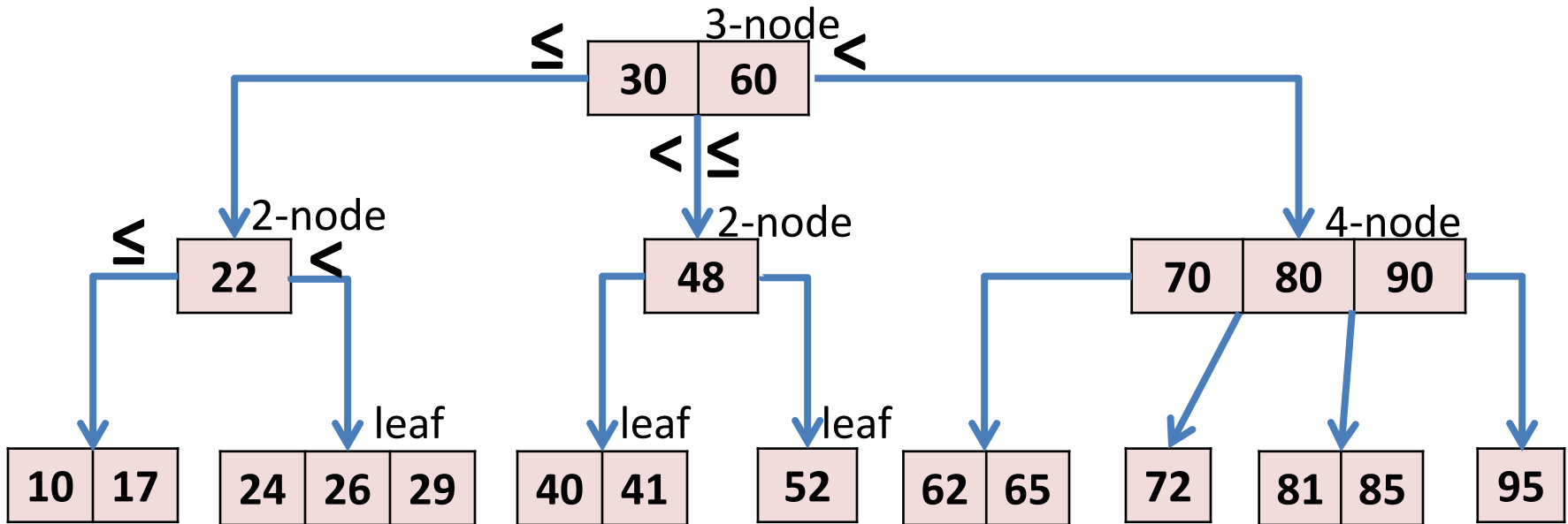
- Items in a node are *in order of keys*
- Given item with key k :
 - * Keys in left subtree: $< k$ ($\leq k$, if duplicates)
 - * Keys in right subtree: $> k$

All leaves must be at the same level. (It grows and shrinks from the root.)

A node that is not a leaf, will have ALL the children around an item. E.g. cannot simply remove node 72 (No 'gaps')

The tree grows in height from the root.

Web animation: [Data Structures Visualization](#), Indexing->B-Trees, select Max degree = 4 and "preemptive split/merge"



Difference between items and nodes. How many nodes? Items? Types of nodes?

2-3-4 Trees

- All leaves are at the same level.
- Any internal node has at least 2 children (has all the children around its keys)

It is similar to a perfect tree.
There are no 'gaps' in the tree.

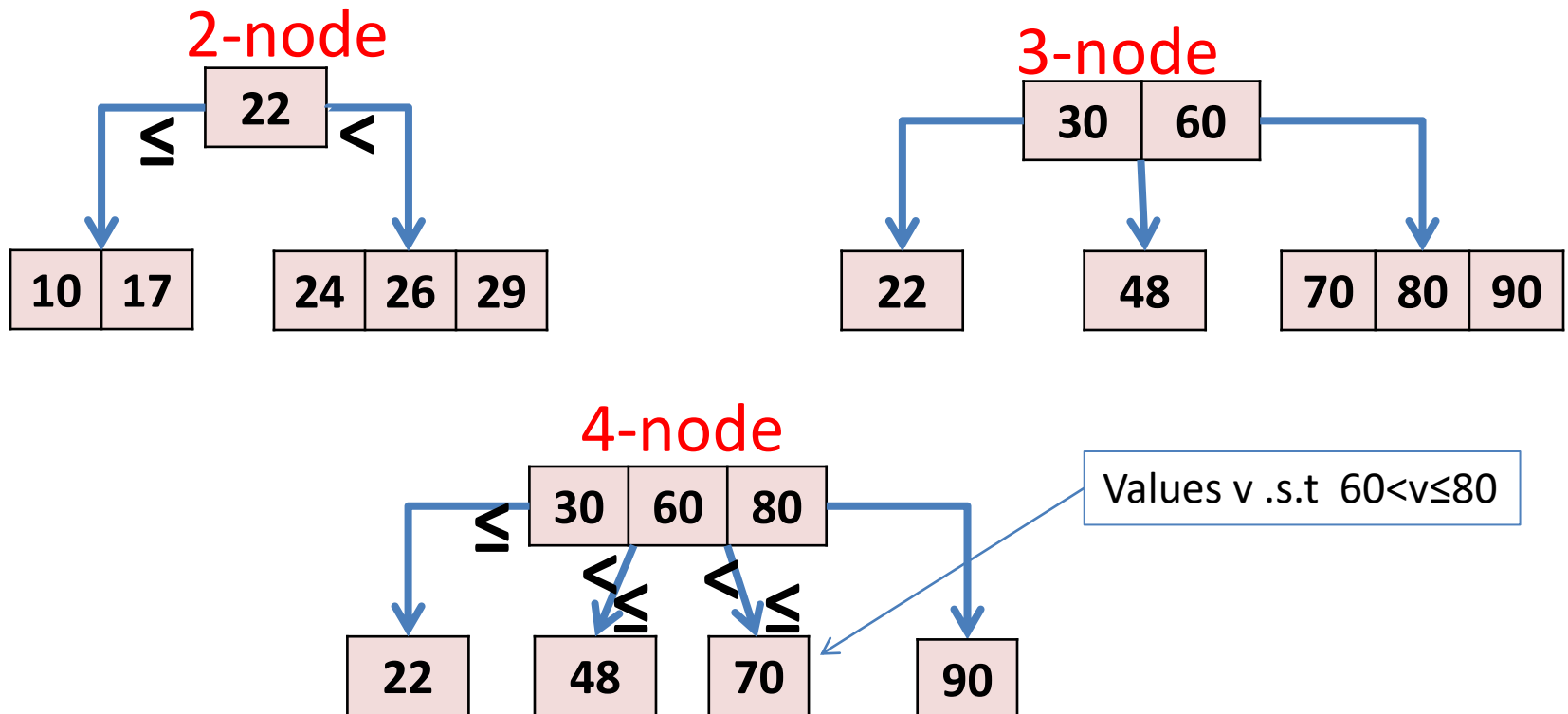
– See picture on next page

- The tree is guaranteed to stay balanced regardless of the order of insertions.
- Has three types of nodes:

- 2-nodes, which contain:
 - An item with key K .
 - A left subtree with keys $\leq K$.
 - A right subtree with keys $> K$.
- 3-nodes, which contain:
 - Two items with keys K_1 and K_2 , $K_1 \leq K_2$.
 - A left subtree with keys $\leq K_1$.
 - A middle subtree with $K_1 < \text{keys} \leq K_2$.
 - A right subtree with keys $> K_2$.
- 4-nodes, which contain:
 - Three items with keys K_1, K_2, K_3 , $K_1 \leq K_2 \leq K_3$.
 - A left subtree with keys $\leq K_1$.
 - A middle-left subtree with $K_1 < \text{keys} \leq K_2$.
 - A middle-right subtree with $K_2 < \text{keys} \leq K_3$.
 - A right subtree with keys $> K_3$.

Types of Nodes

```
struct TreeNode234{  
    int keys[3];  
    struct TreeNode234* children[4];  
    int numChildren; //2, 3 or 4  
    //to know if a leaf or not, use children[0]==NULL or another field  
};
```



Search in 2-3-4 Trees

- For simplicity, we assume that all keys are unique.
- Search in 2-3-4 trees is a generalization of search in binary search trees.
 - select one of the subtrees by comparing the search key with the 1, 2, or 3 keys that are present at the node.
- Search time is logarithmic to the number of items.
 - The time is at most linear to the height of the tree.
 - $\log_4(N/3) \leq \text{height} \leq \log_2 N$. Remember that $\log_4(N/3) = \Theta(\log_2 N)$
 - This inequality is derived based on the extreme cases where all the nodes are of type 2-node (so only one item in each node \Rightarrow N nodes and a binary tree) and all the nodes are of type 4-node (so the tree will have $N/3$ nodes and every node has 4 children \Rightarrow $\text{height} = \log_4(\text{num_of_tree_nodes}) = \log_4(N/3)$)
- Next:
 - how to implement insertions and deletions so that the tree keeps its property: all leaves are at the same level.

Insertion in 2-3-4 Trees

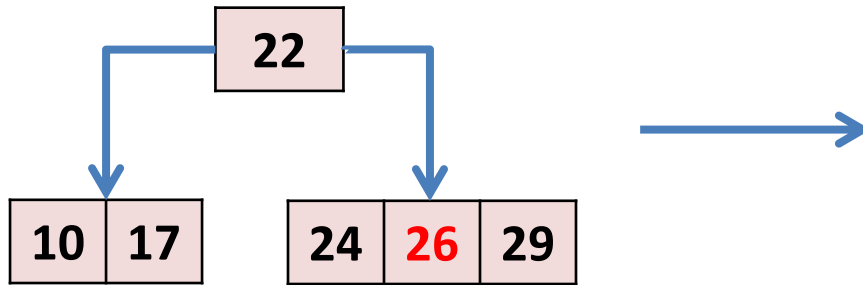
- We follow the same path as if we are searching for the item.
- We cannot just insert the item at the end of that path:
 - Case 1: If the leaf is a 2-node or 3-node, there is room to insert the new item with its key - OK
 - Case 2: **If the leaf is a 4-node, there is NO room for the new item.** In order to insert it here , we would have to create a new leaf that would be on a different level than all the other leaves – PROBLEM => ‘break this node’ =>
 - **Fix all 4-nodes on the way as you search down in the tree.**
- **The tree will grow from the root.**

Insertion in 2-3-4 Trees

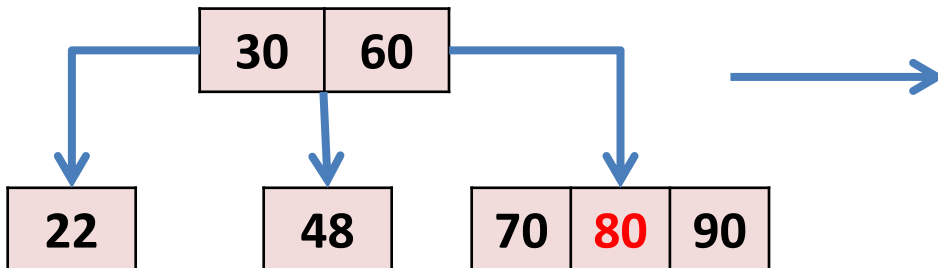
- Given our key K: we follow the same path as in search.
- **“Preemptive Split”**: on the way we **“fix all the 4 nodes we meet”**:
 - Web animation: [Data Structures Visualization](#), Indexing->B-Trees, select “preemptive split/merge”
 - If the parent is a 2-node, transform the pair into a 3-node connected to two 2-nodes.
 - If the parent is a 3-node, we transform the pair into a 4-node connected to two 2-nodes.
 - **If there is no parent (the root itself is a 4-node), split it into three 2-nodes (root and children). - This is how the tree height grows.**
- These transformations:
 - Are local (they only affect the nodes in question).
 - Do not affect the overall balance or height of the tree (except for splitting a 4-node root).
- This way, when we get to the bottom of the tree, we know that the node we arrived at is not a 4-node, and thus it has room to insert the new item.

Transformation Examples

- If we find a 2-node being parent to a 4-node, we transform the pair into a 3-node connected to two 2-nodes, by pushing up the **middle** key of the 4-node.

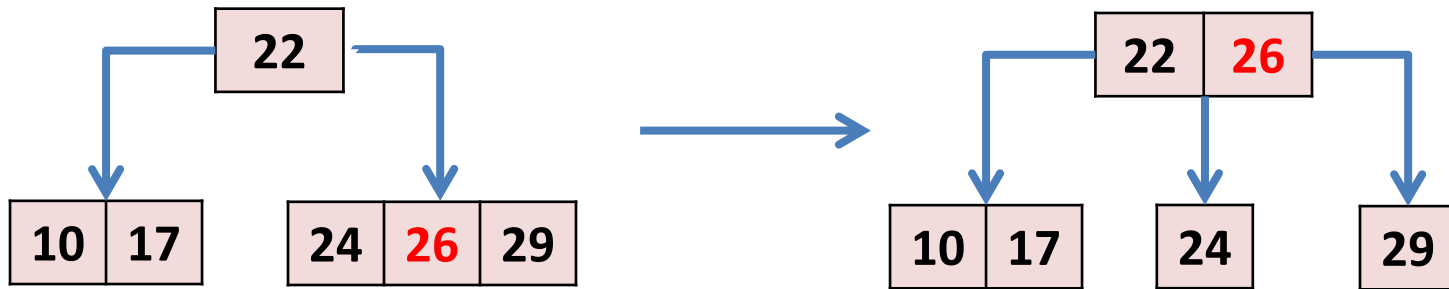


- If we find a 3-node being parent to a 4-node, we transform the pair into a 4-node connected to two 2-nodes, by pushing up the **middle** key of the 4-node.

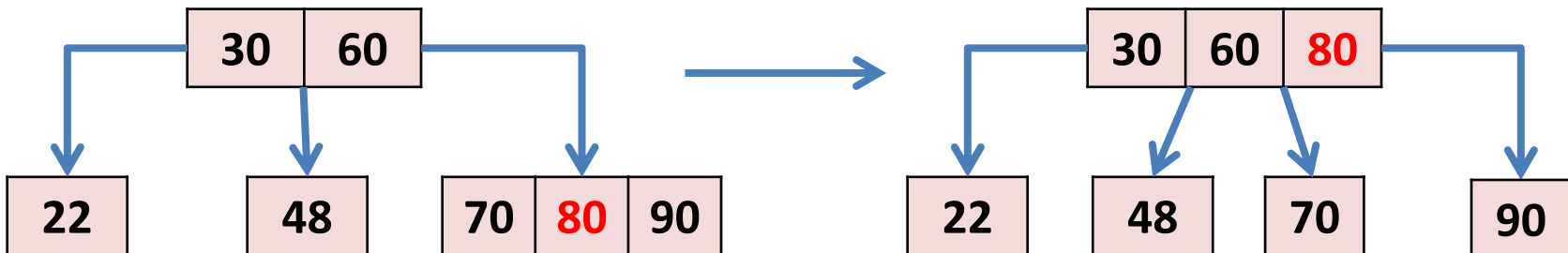


Transformation Examples

- If we find a 2-node being parent to a 4-node, we transform the pair into a 3-node connected to two 2-nodes, by pushing up the **middle** key of the 4-node.



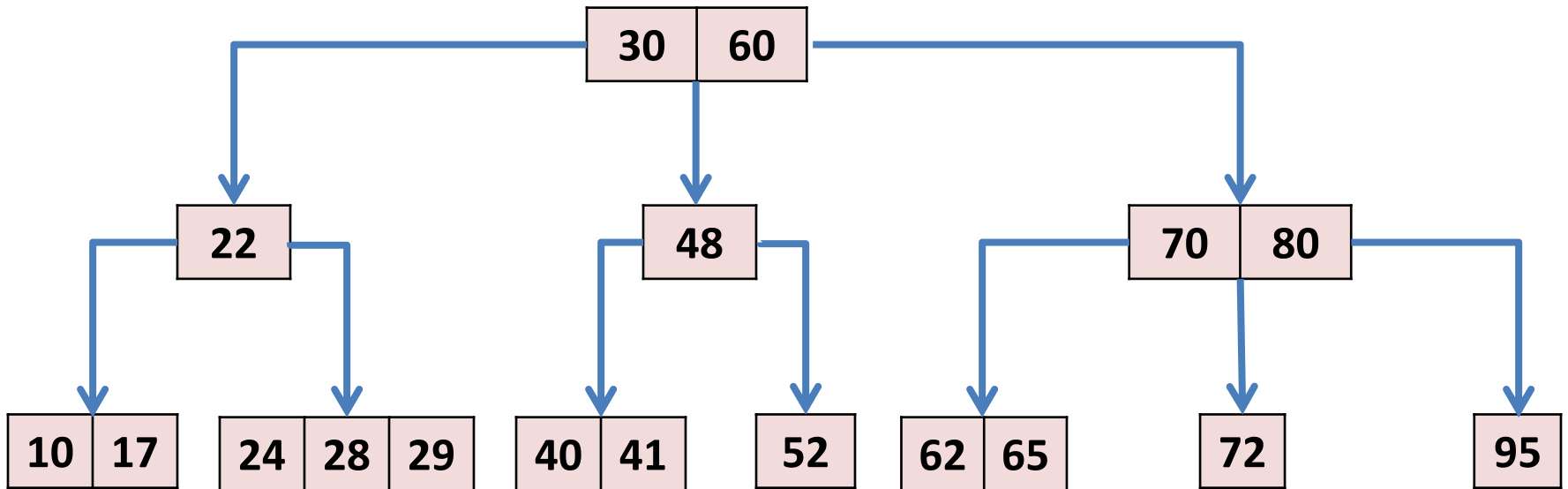
- If we find a 3-node being parent to a 4-node, we transform the pair into a 4-node connected to two 2-nodes, by pushing up the **middle** key of the 4-node.



Insertion Examples

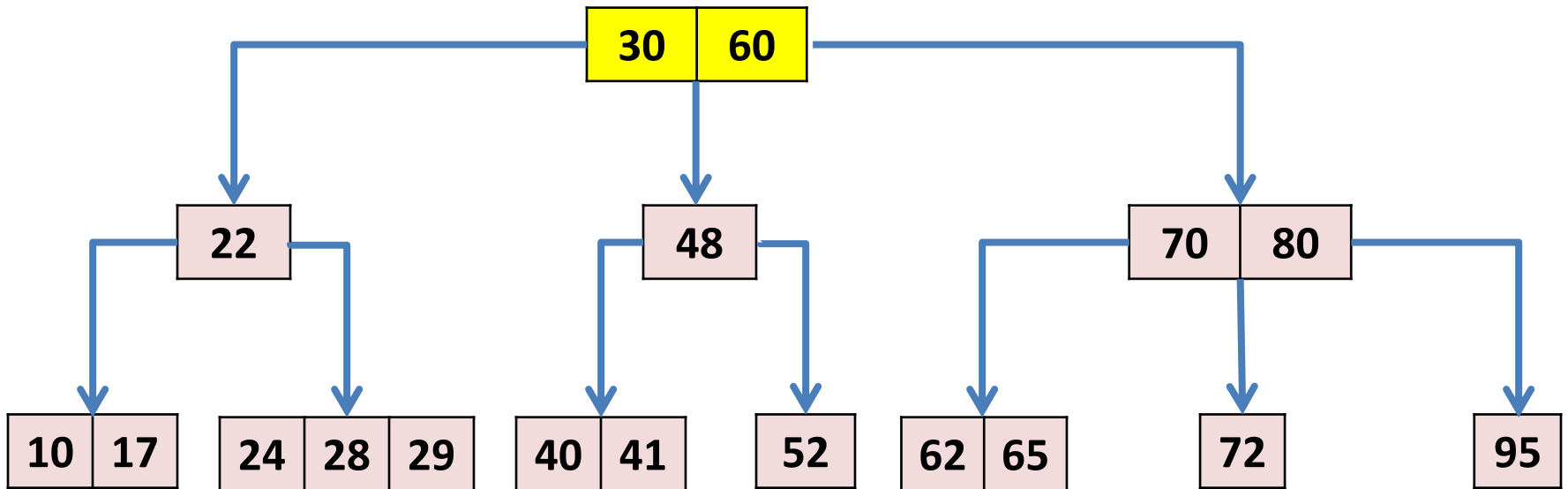
Insert 25

- Inserting an item with key 25:



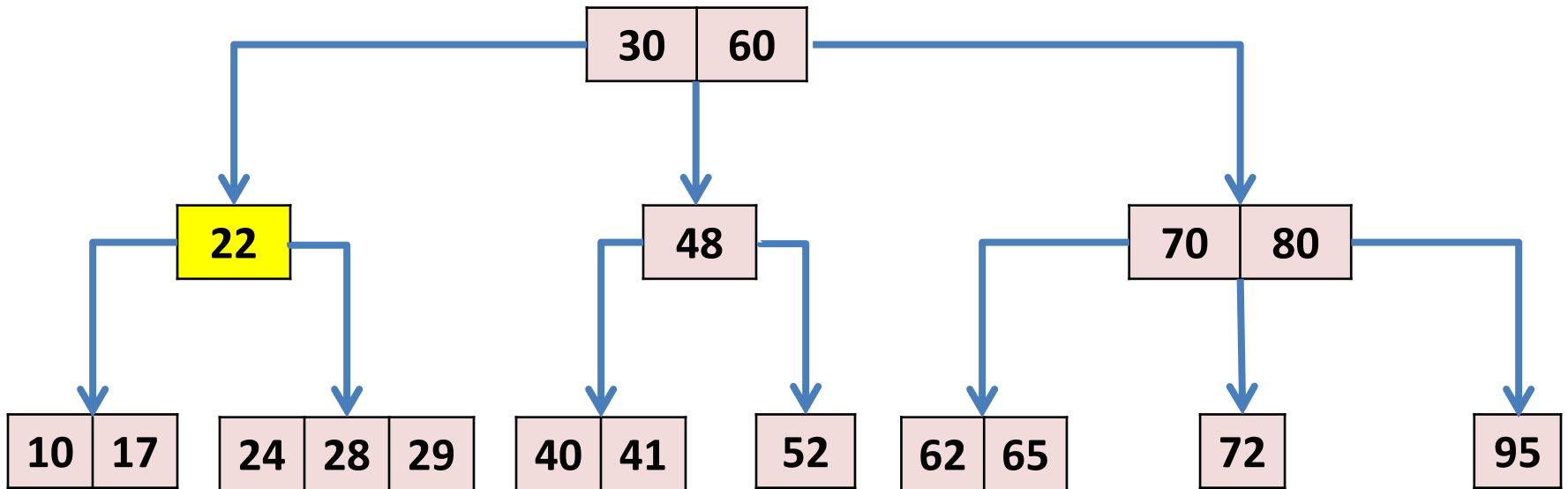
Insert 25

- Inserting an item with key 25:



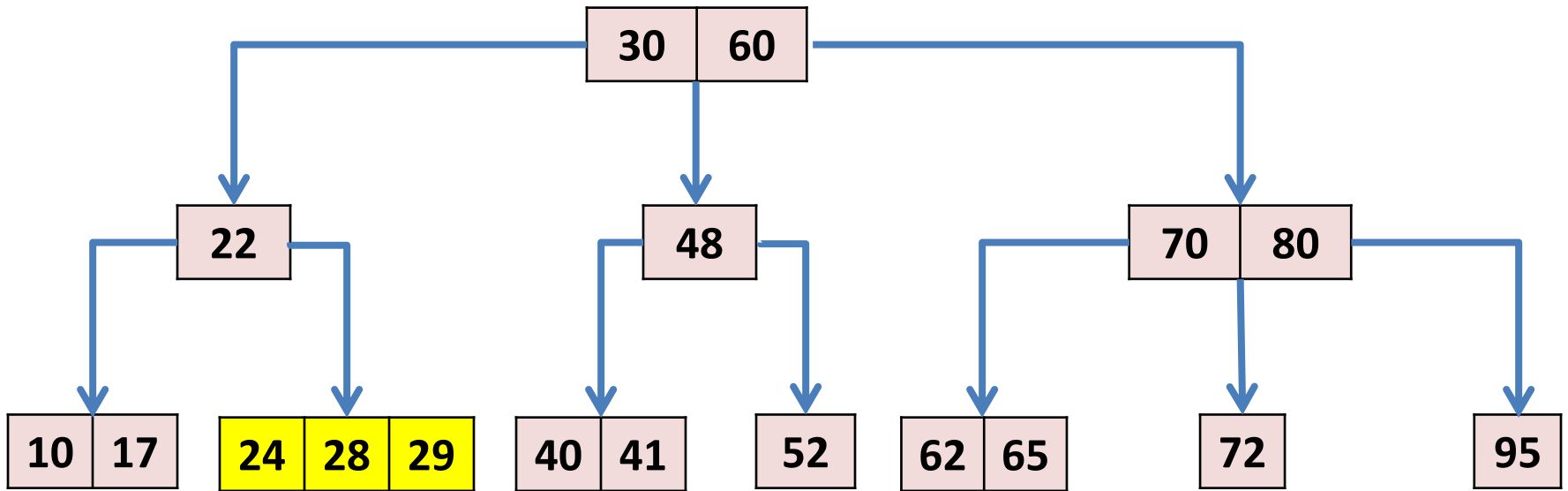
Insert 25

- Inserting an item with key 25:



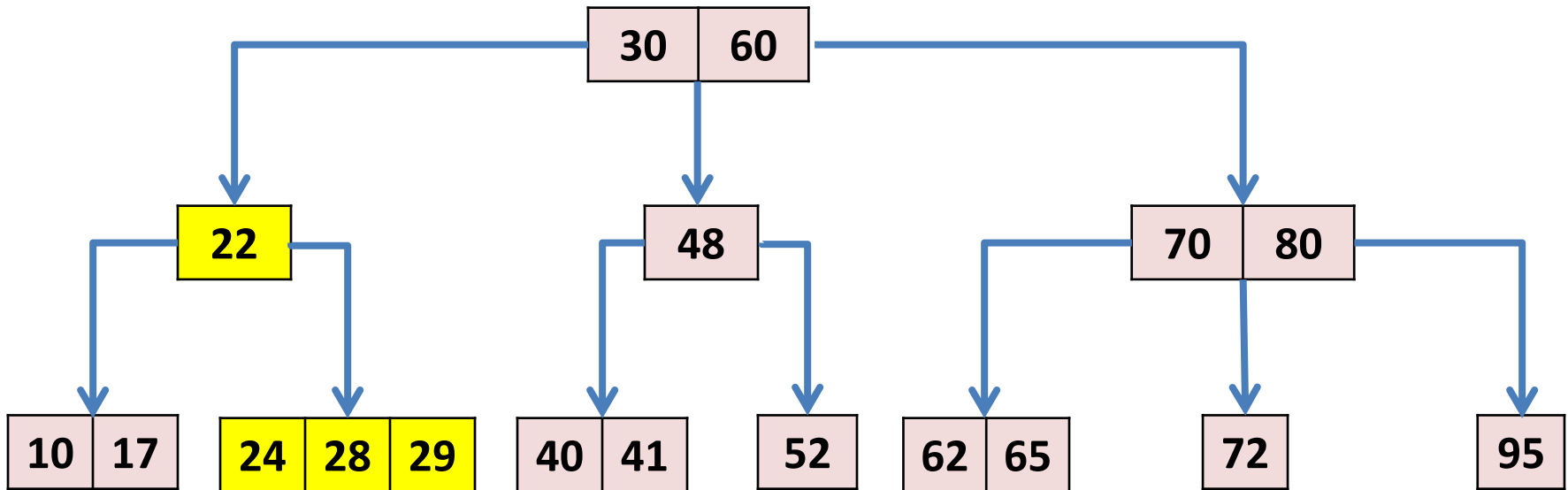
Insert 25

- Inserting an item with key 25:



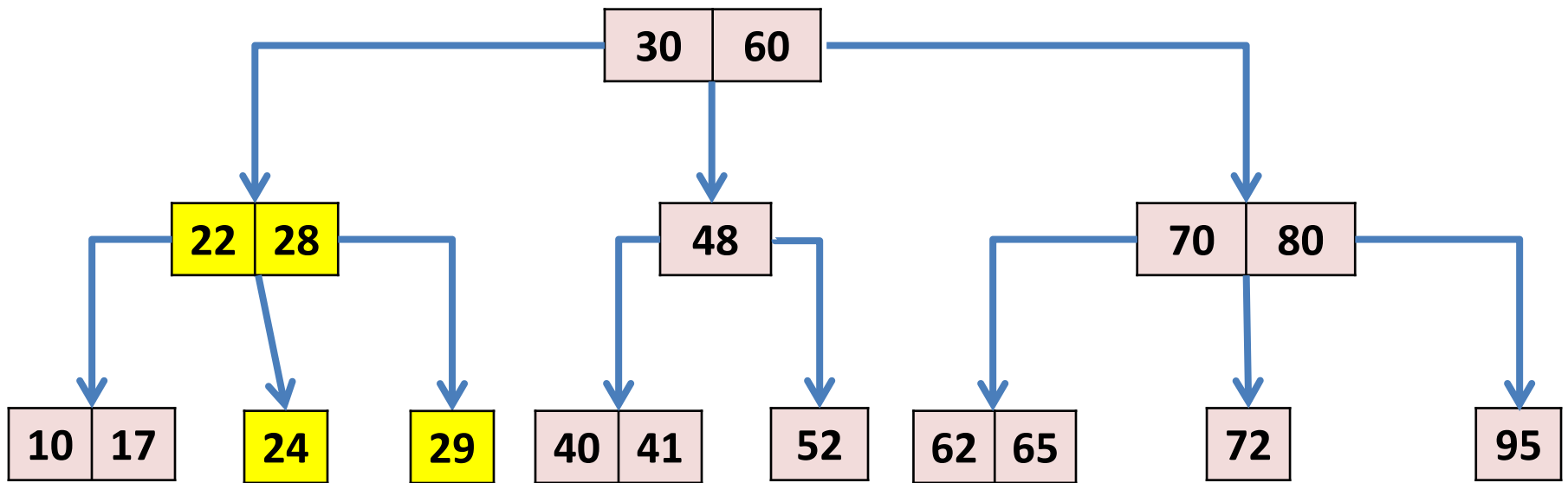
Insert 25

- We found a 4-node, we must split it and send an item up to the parent (2-node) which will become a 3-node.



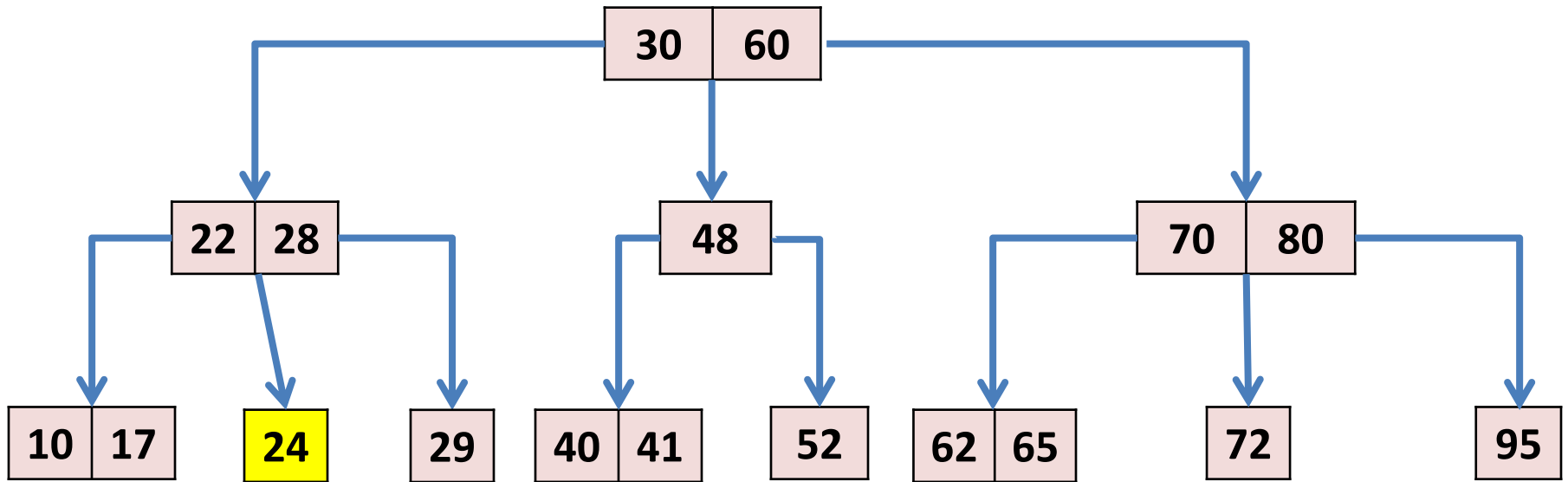
Insert 25

- Continue search for 25 from the updated (22,28) node.



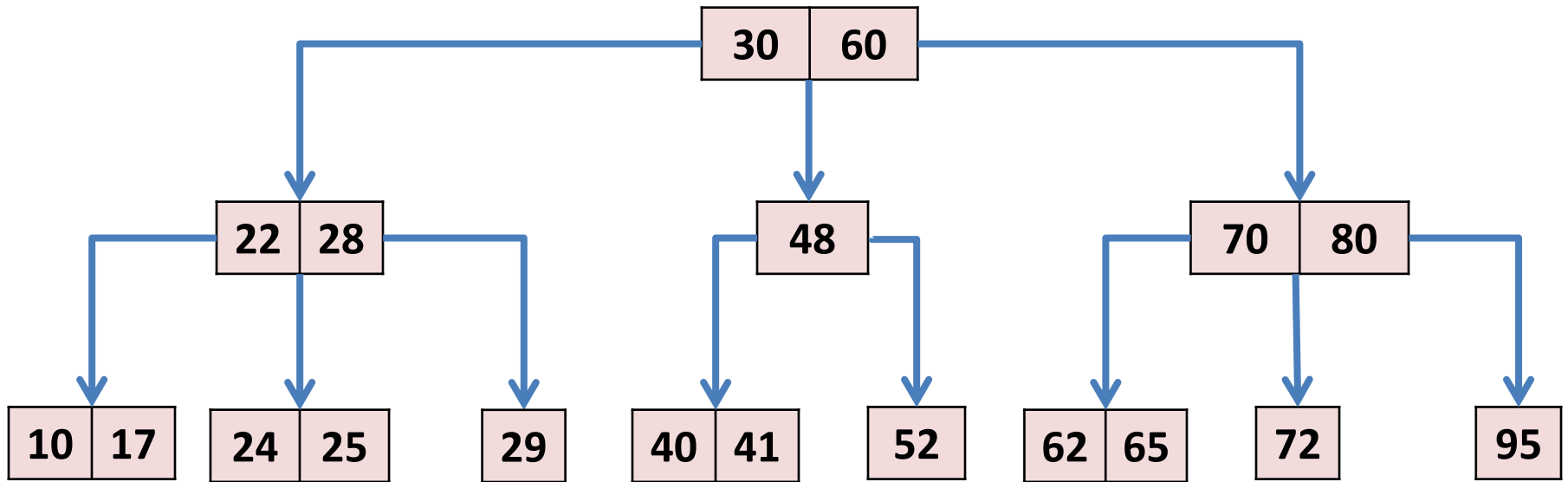
Insert 25

- Reached a leaf with less than 3 items. Add the item.

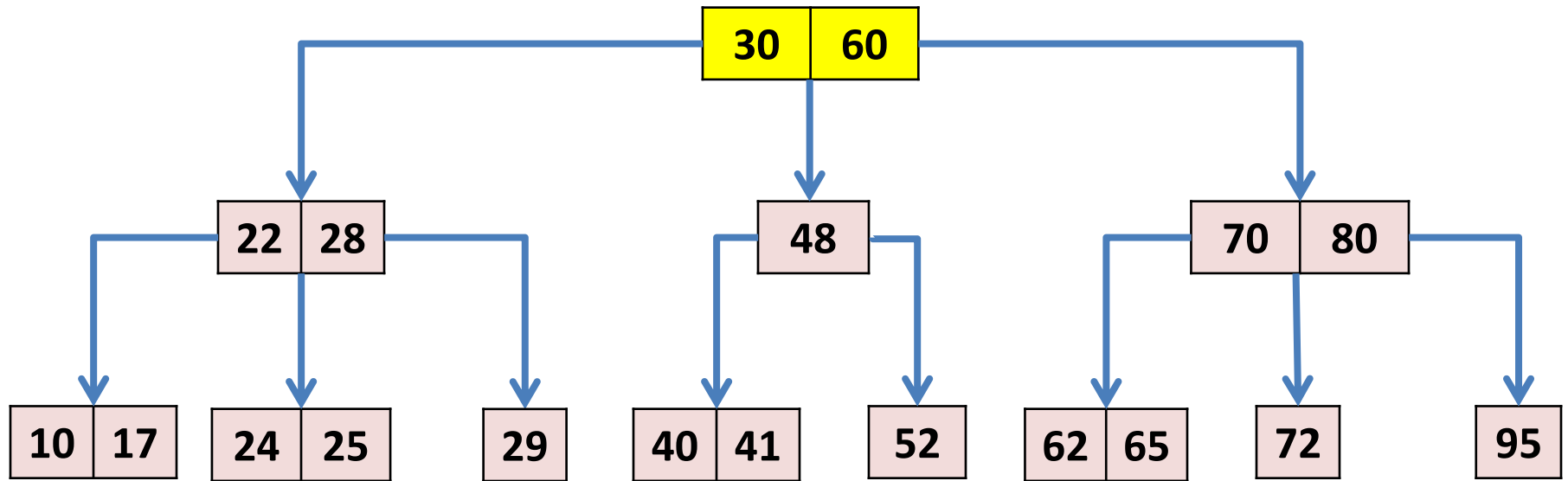


Insert 27

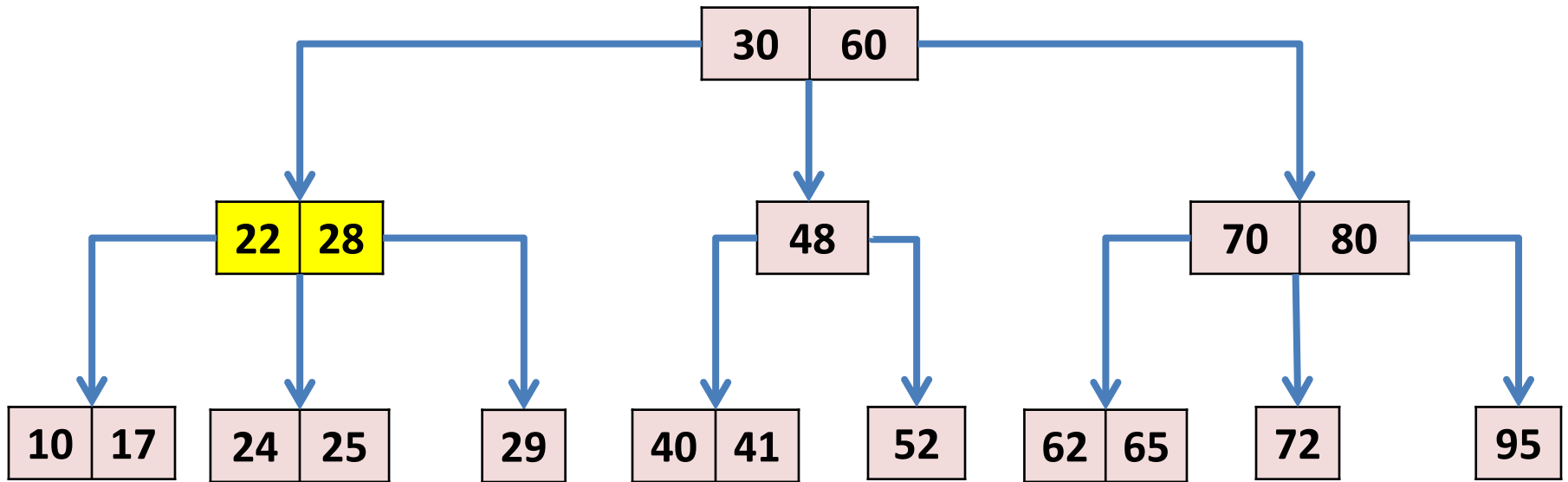
- Next: insert an item with **key = 27**.



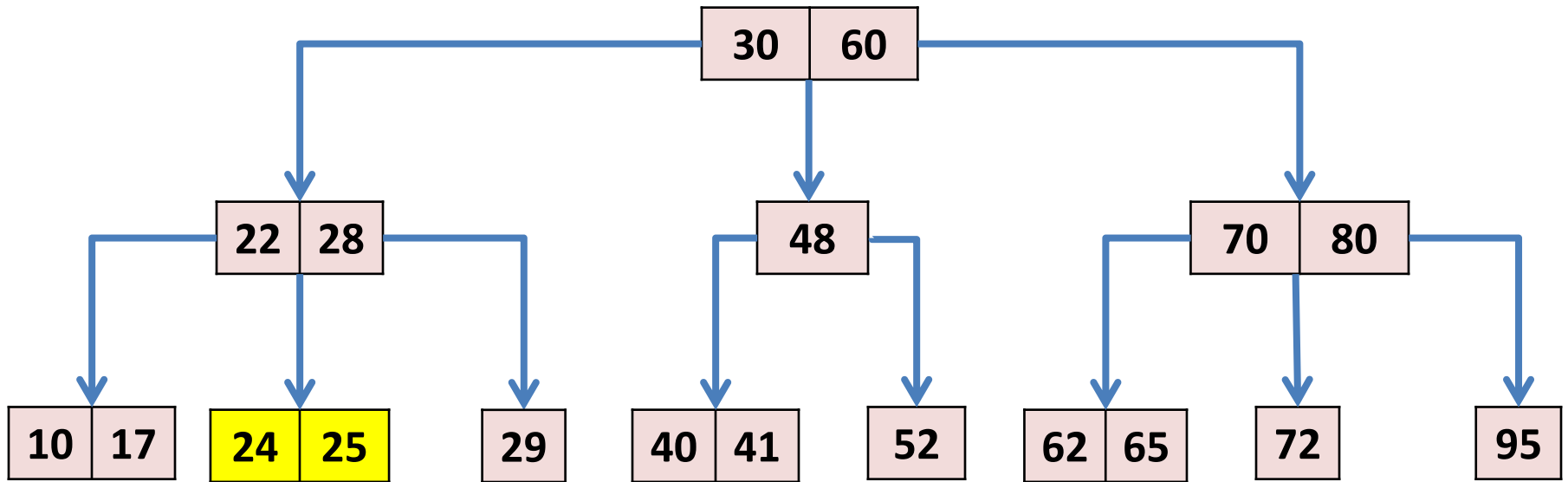
Insert 27



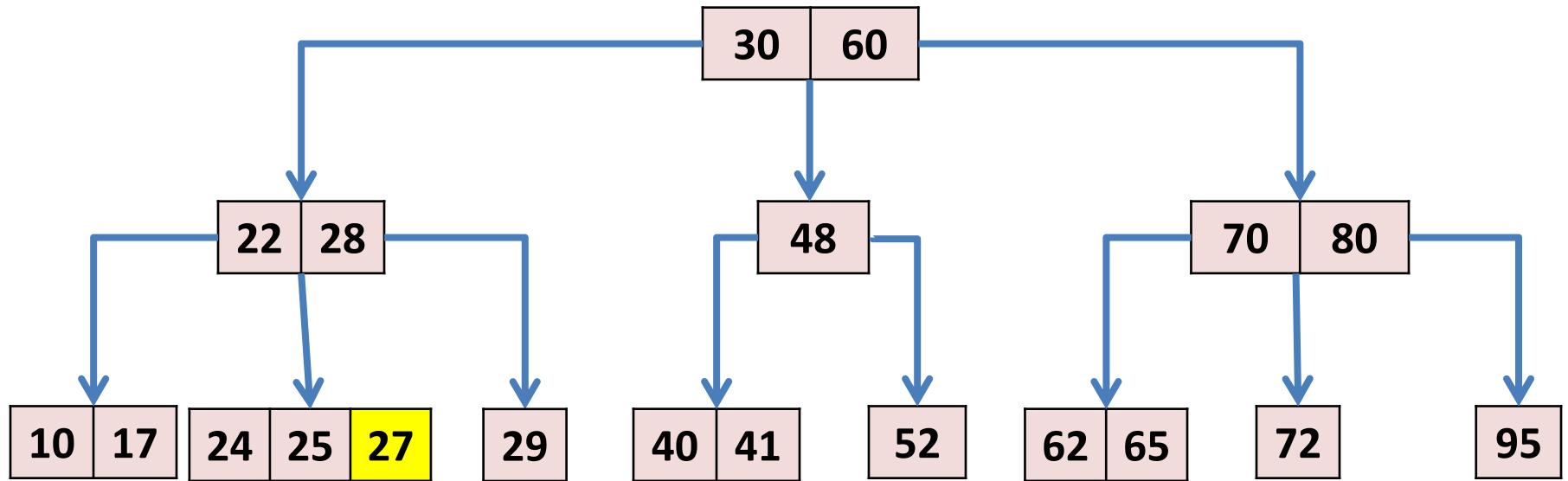
Insert 27



Insert 27

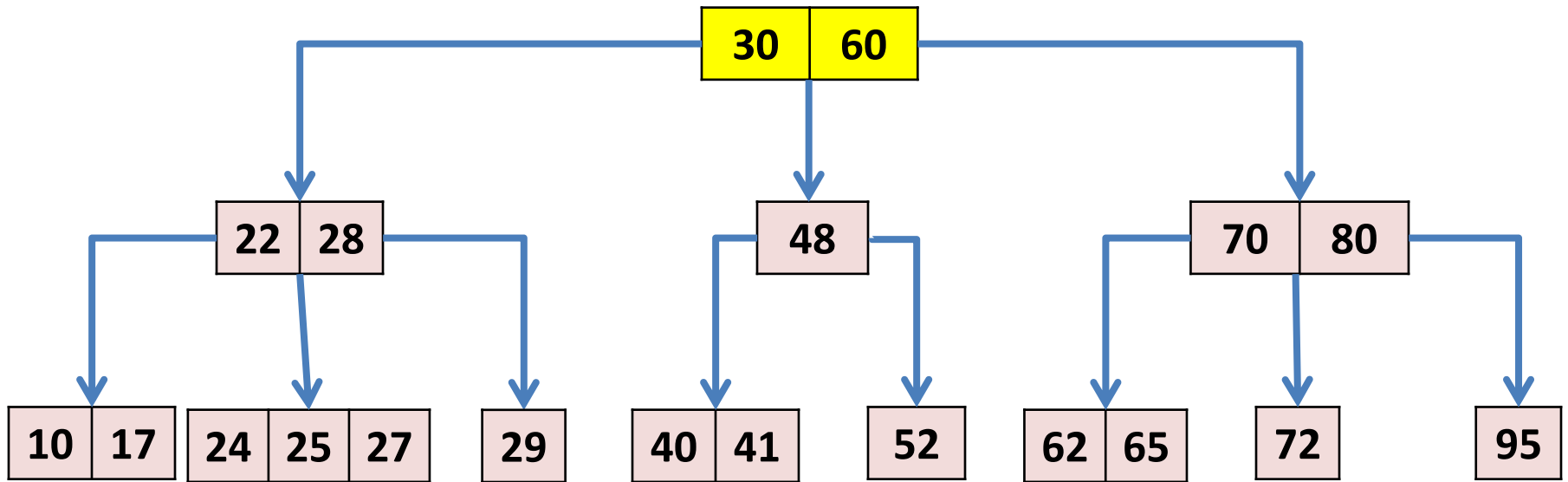


Insert 27

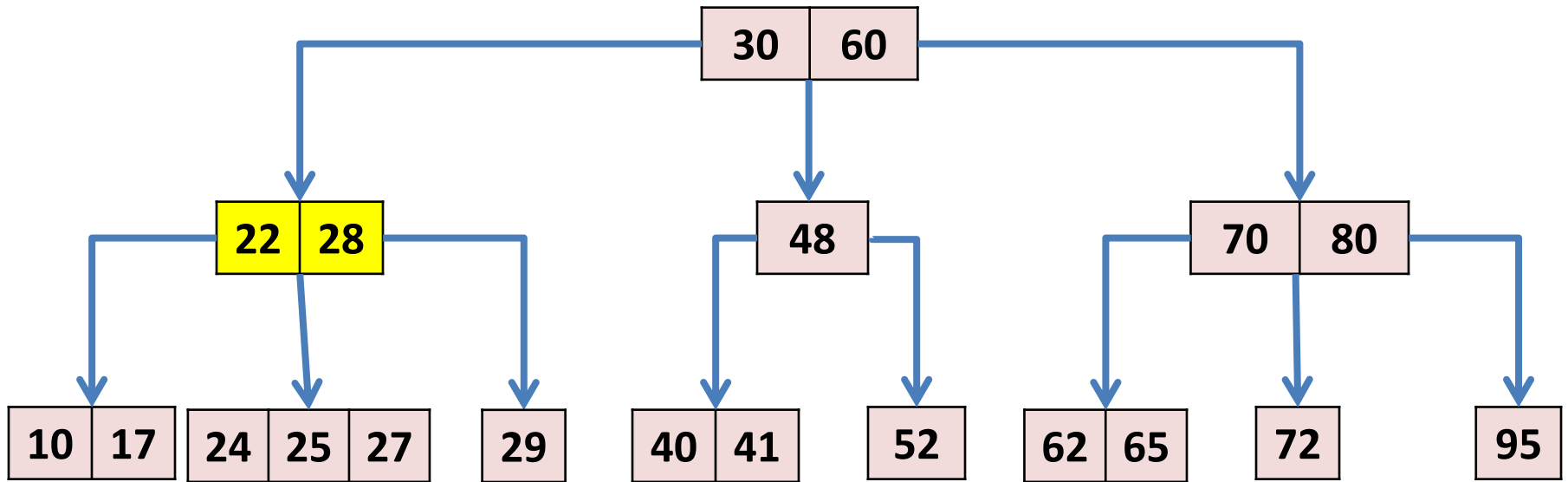


Insert 26

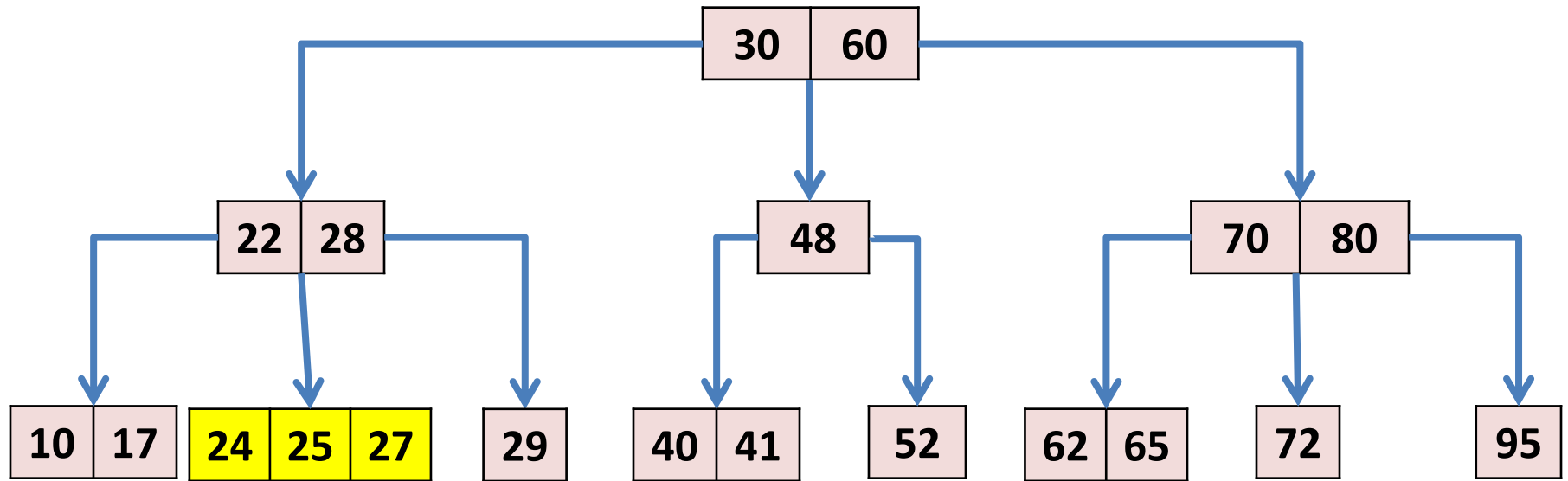
- Next: insert an item with key = 26.



Insert 26

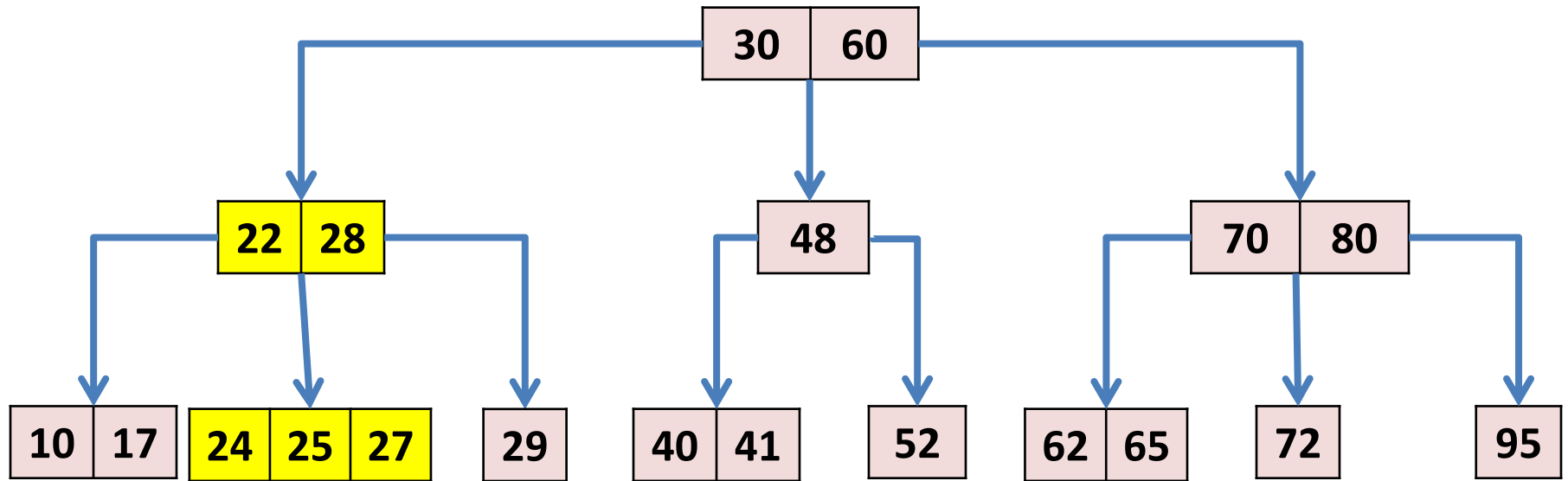


Insert 26



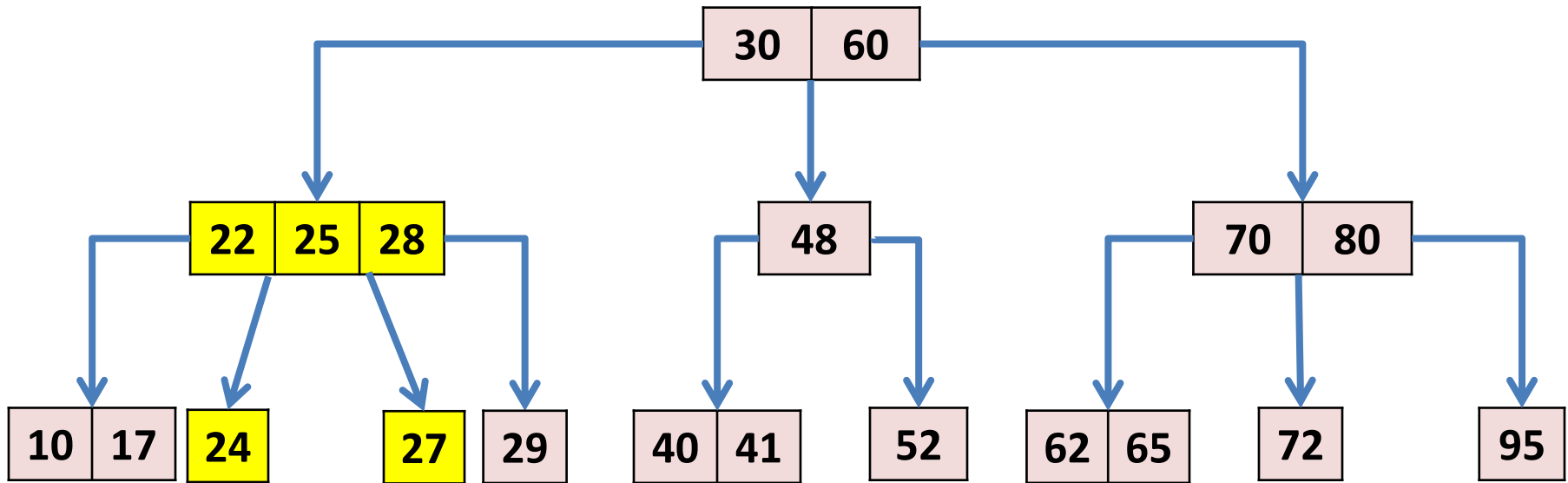
Insert 26

- Found a 3-node being parent to a 4-node, we must transform the pair into a 4-node connected to two 2-nodes.



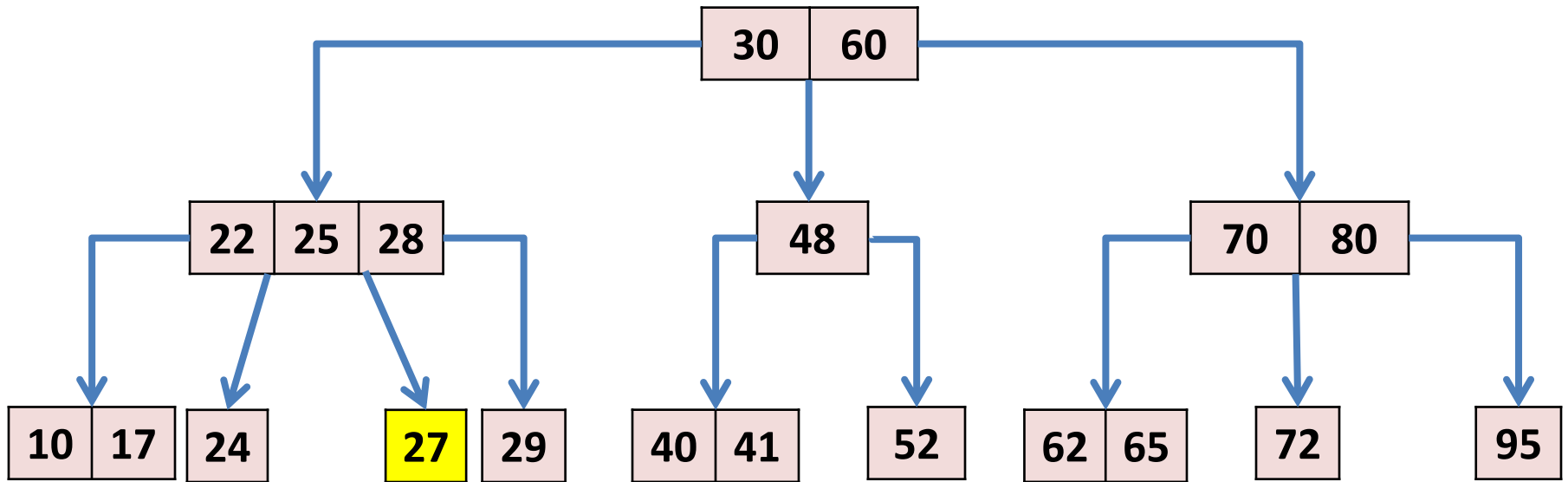
Insert 26

- Found a 3-node being parent to a 4-node, we must transform the pair into a 4-node connected to two 2-nodes.



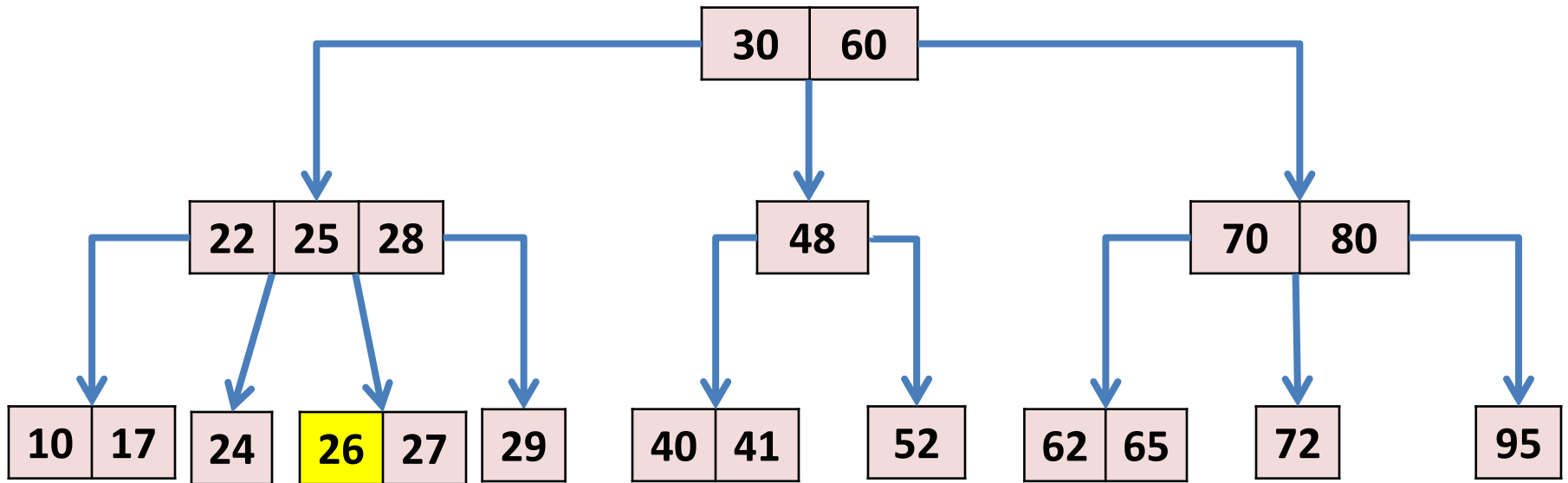
Insert 26

- Reached the bottom. Make insertion of item with key 26.



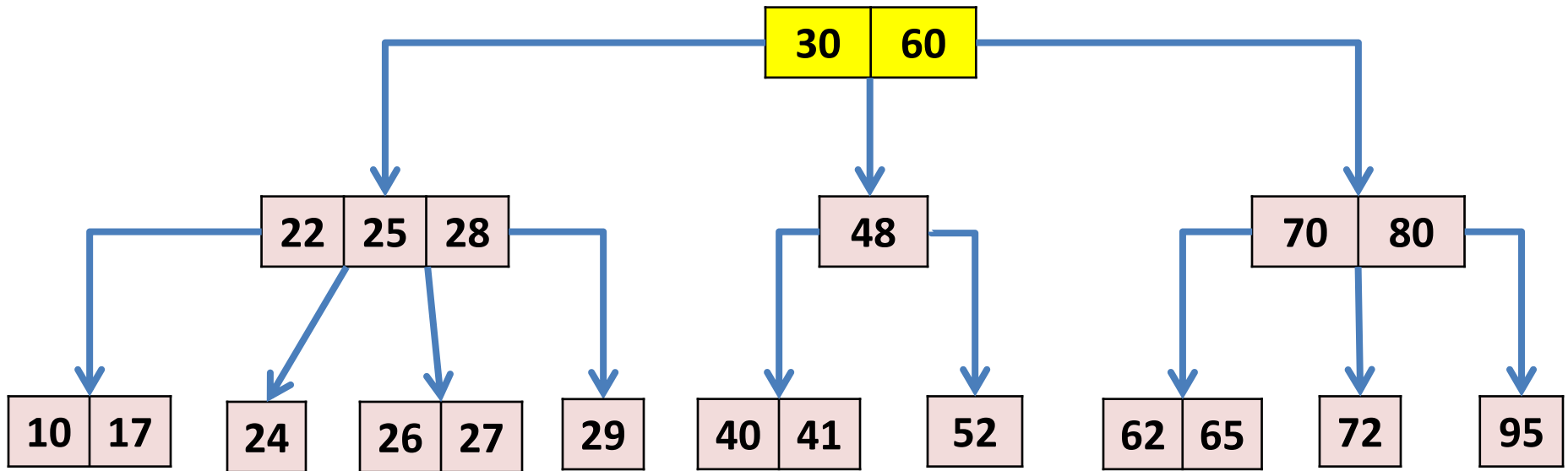
Insert 26

- Reached the bottom. Make insertion of item with key 26.



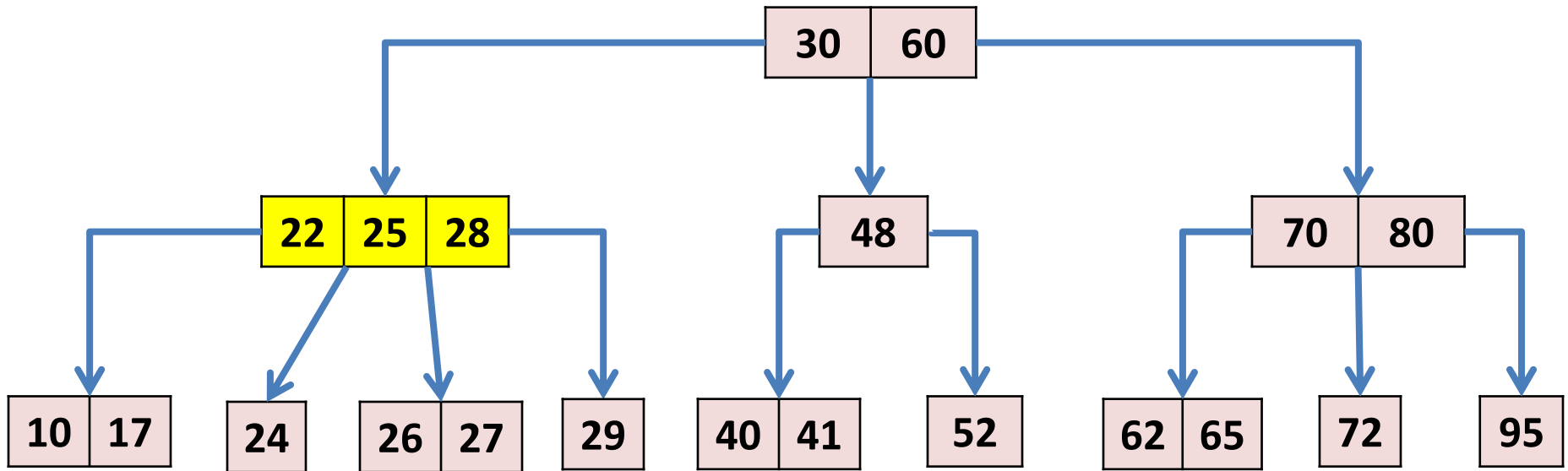
Insert 13

- Insert an item with key = 13.



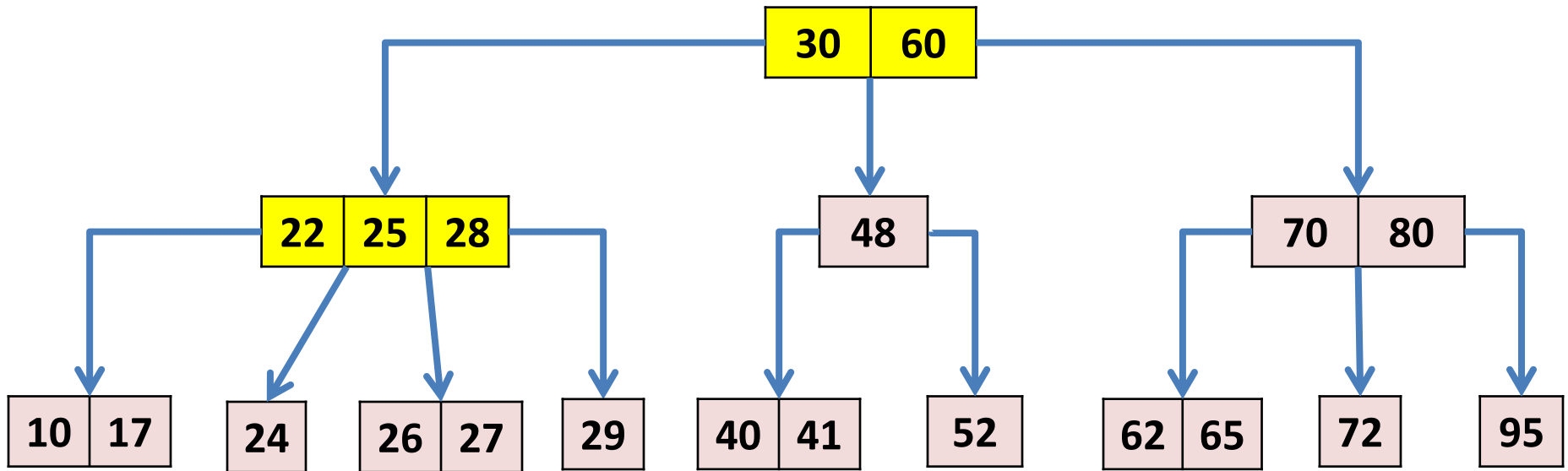
Insert 13 (uses preemptive split)

Our convention: **preemptive split** : Split this node b.c. it is full and I found it on my search down! (Note that we split it even though there is room for 13 in the leaf)



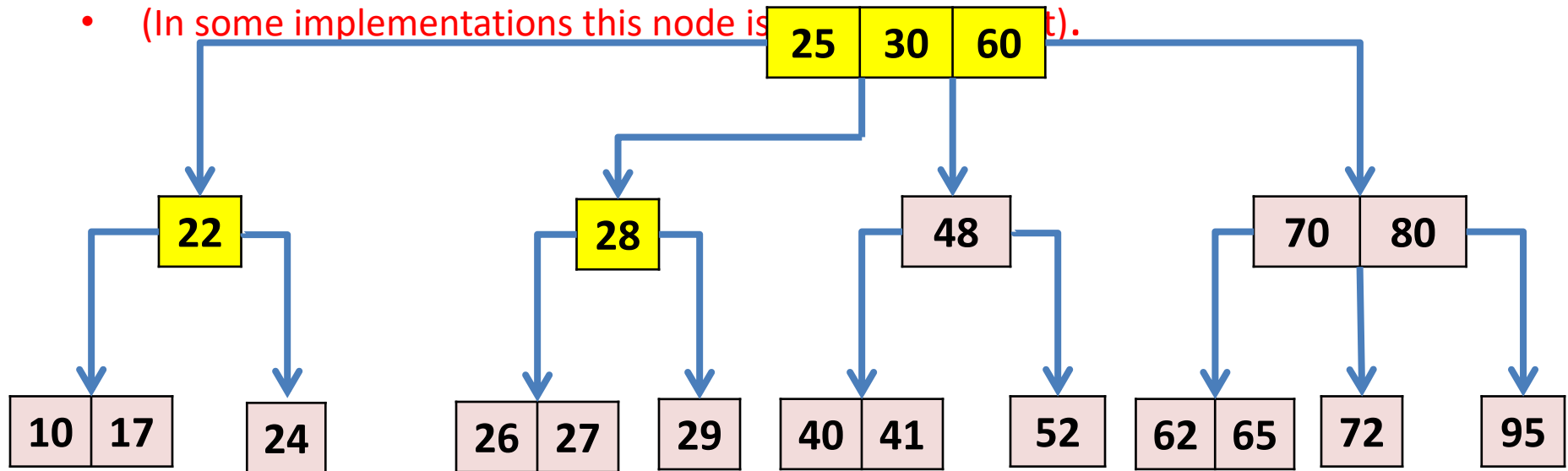
Insert 13 (uses preemptive split)

- Found a 3-node being parent to a 4-node, we must transform the pair into a 4-node connected to two 2-nodes.



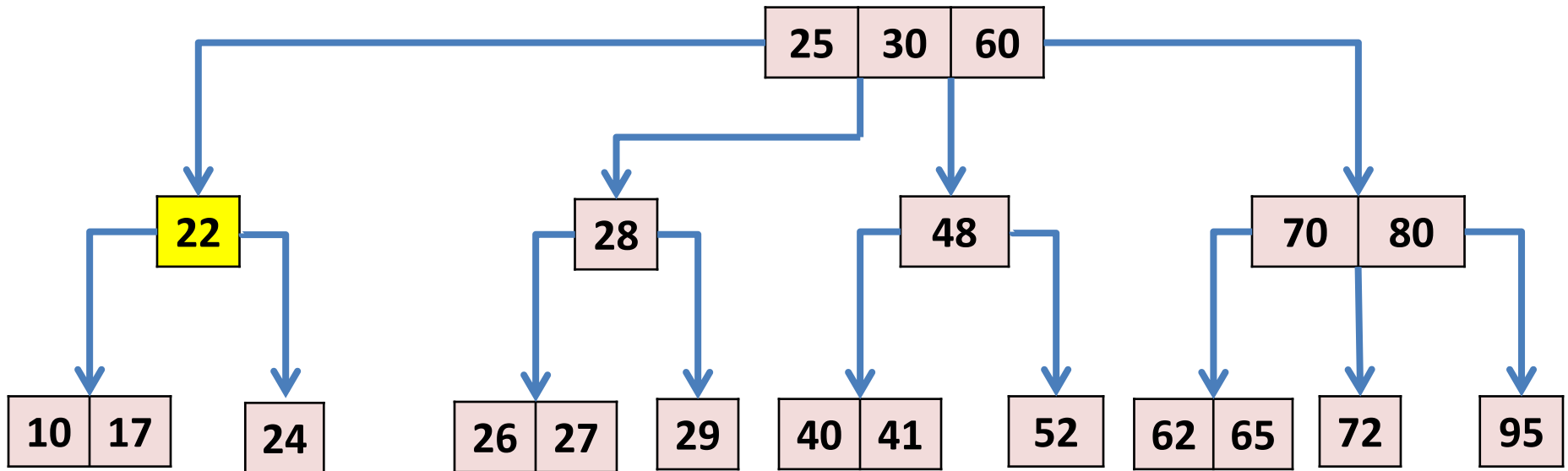
Insert 13

- The root became a 4-node, but we will NOT split it. (*Split what was full, not what just became full.*)
- If a node JUST became 4-node due to us pushing an item in it from splitting one of its children, we do NOT split this node.
- (In some implementations this node is split.)



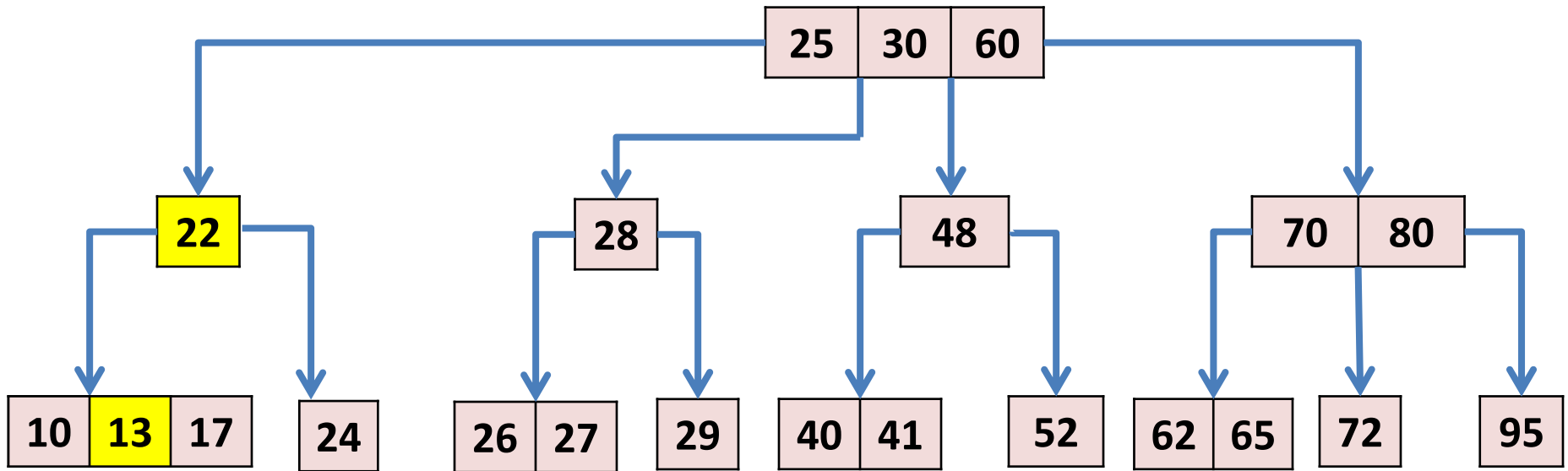
Insert 13

- Continue the search.



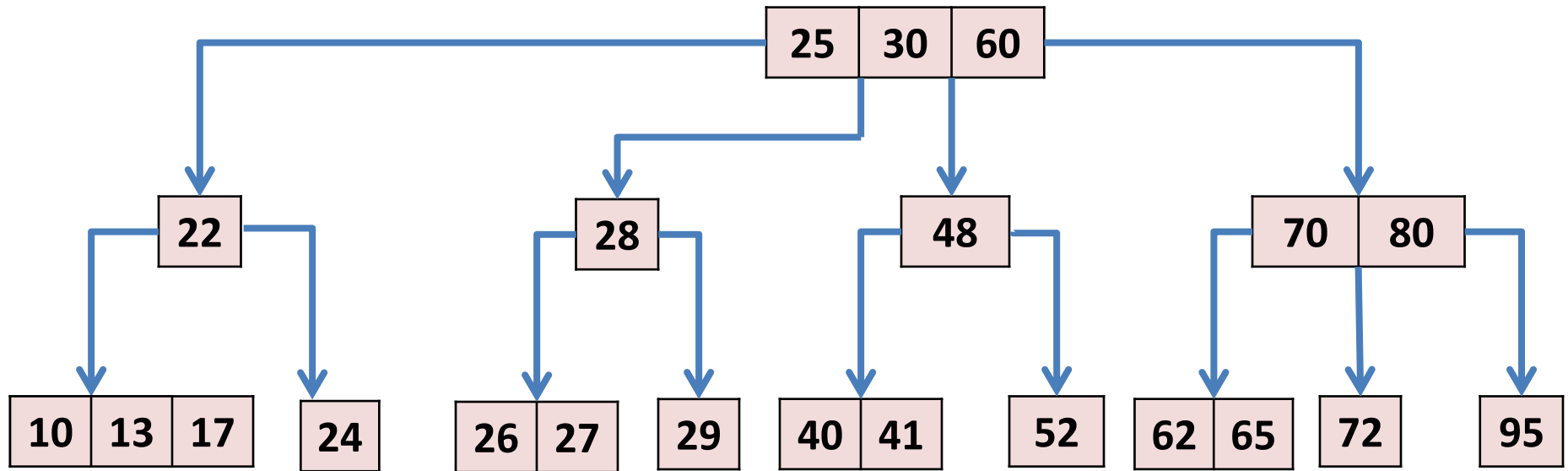
Insert 13

- Insert in leaf node.



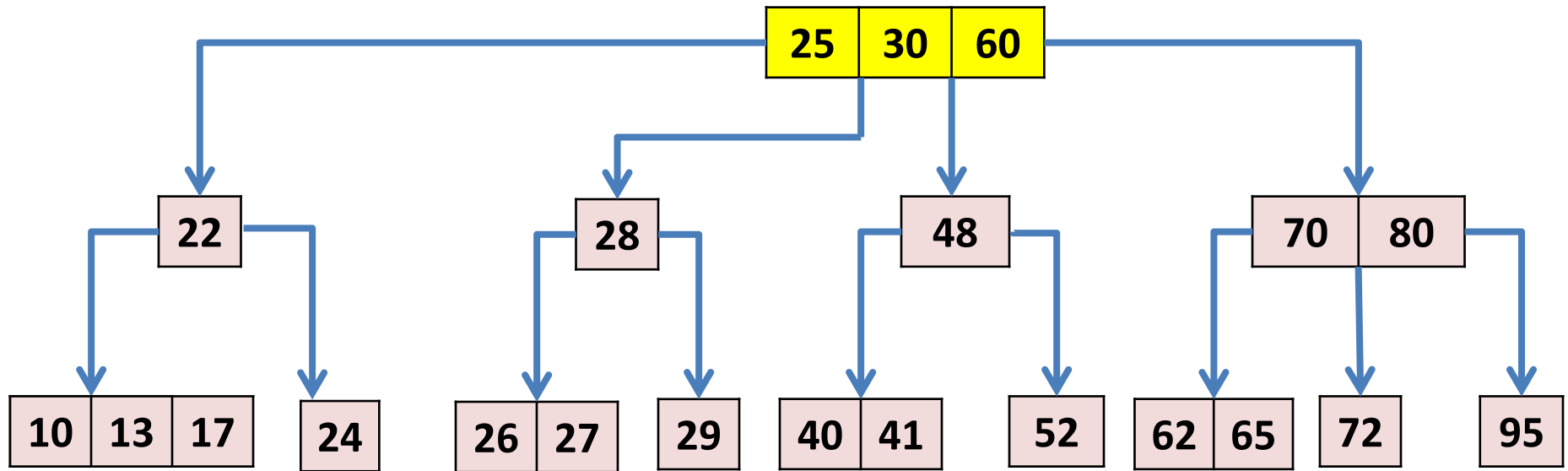
Insert 90

- Insert 90.



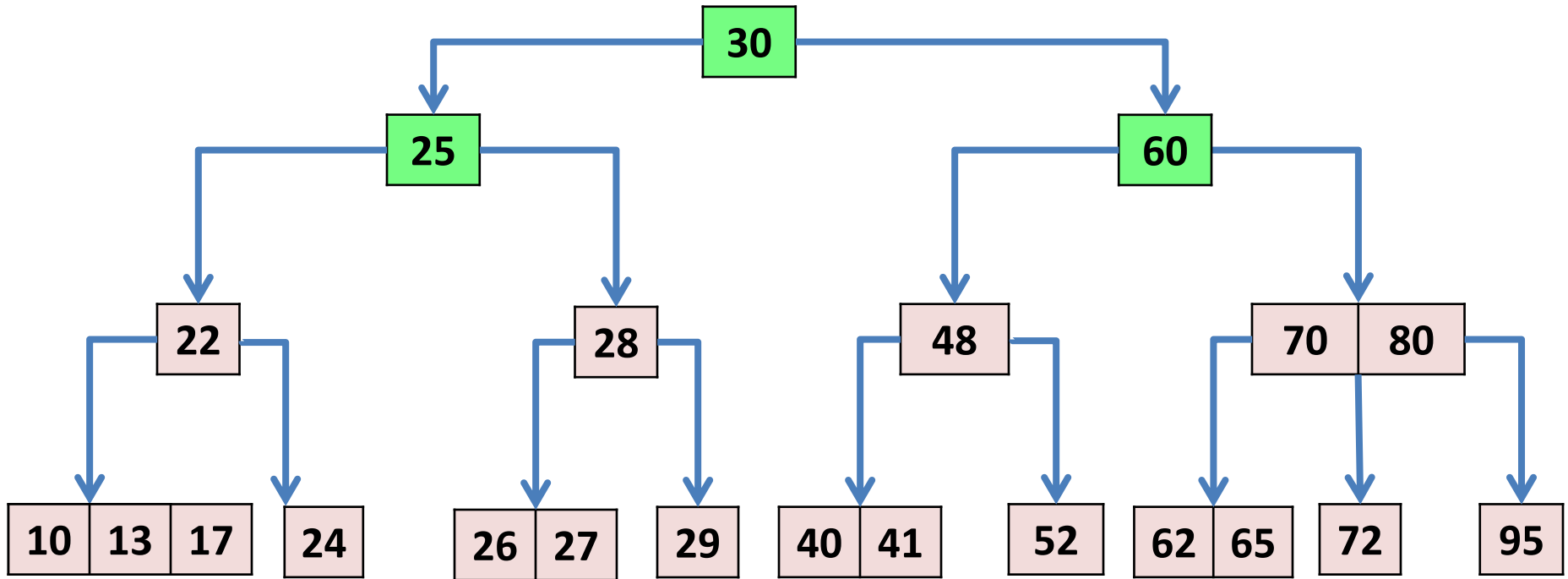
Insert 90 (part 1) (preemptive split)

- Insert 90. The root is a 4-node. **Preemptive split**: split it.



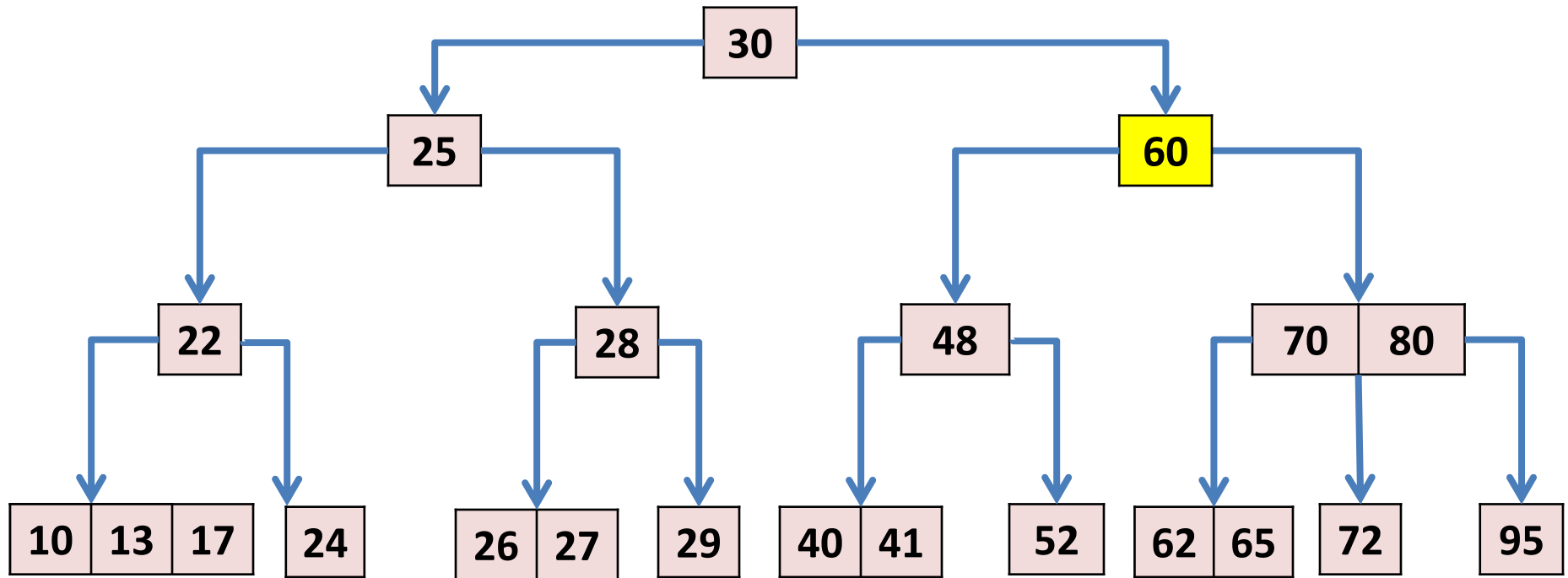
Insert 90 (part 2)

- The root is now split! (preemptive split)
- **THIS IS HOW THE TREE HEIGHT GROWS!**



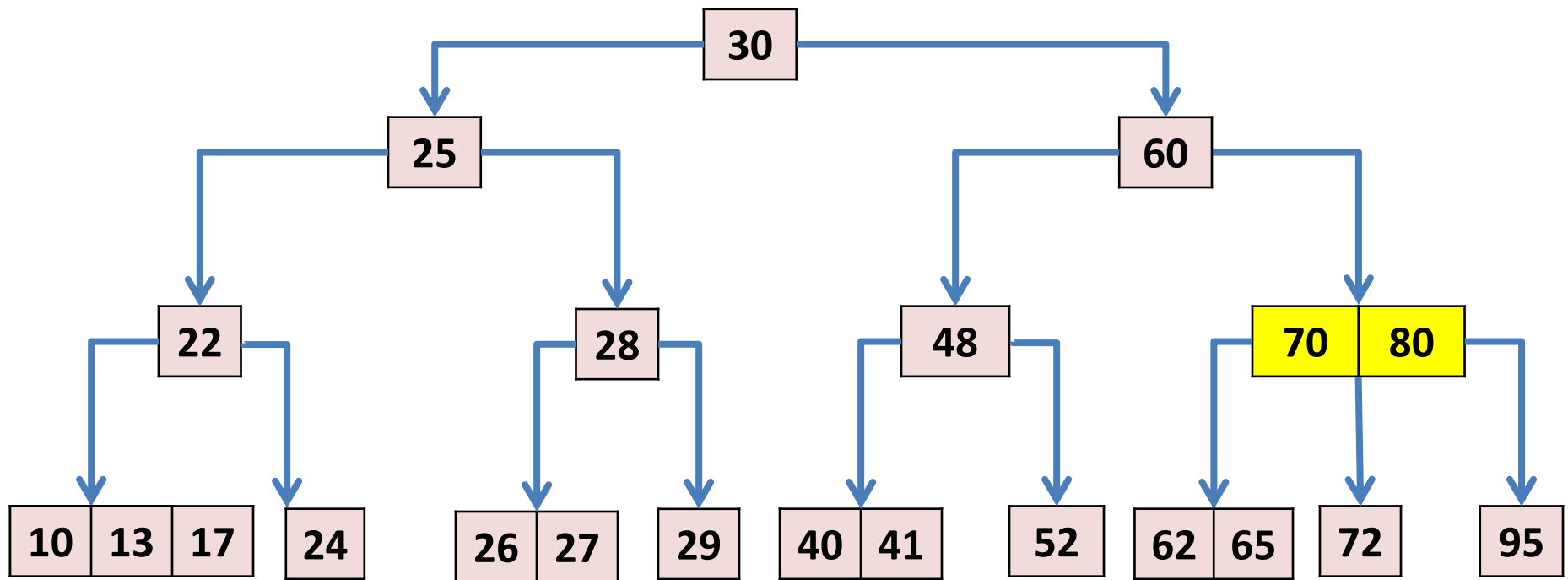
Insert 90 (part 3)

- Continue to search for 90.



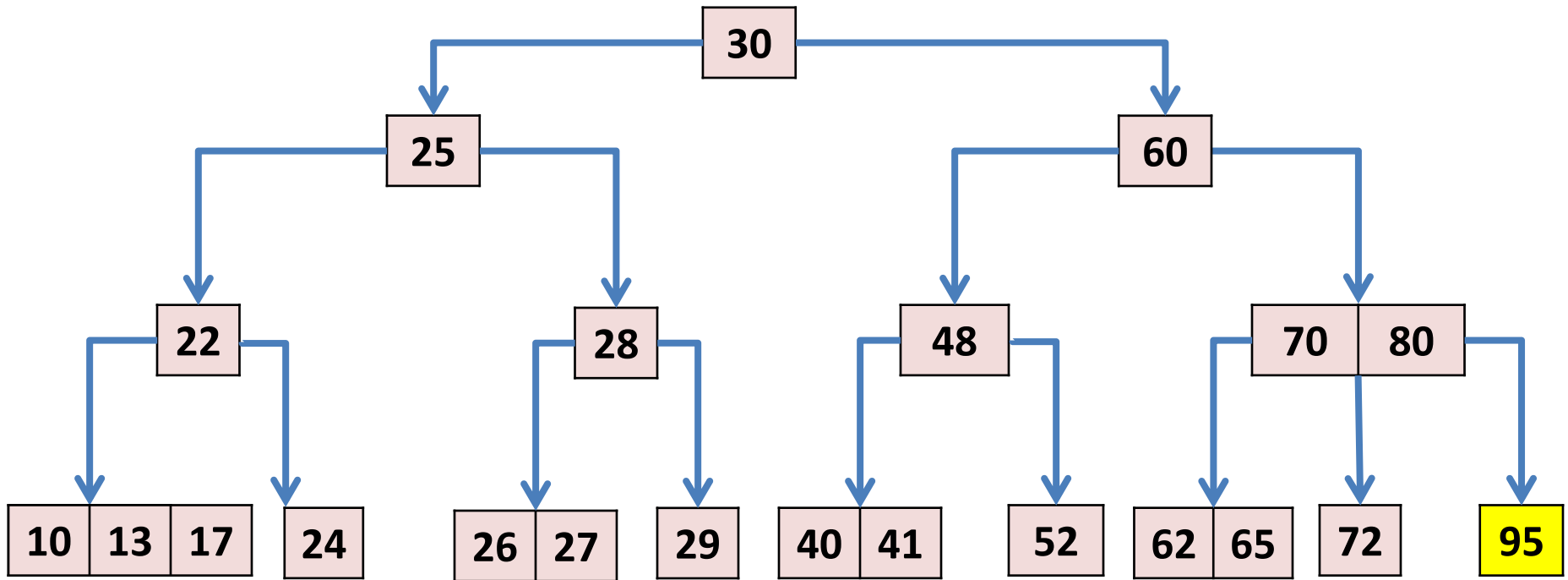
Insert 90 (part 4)

- Continue to search for 90.



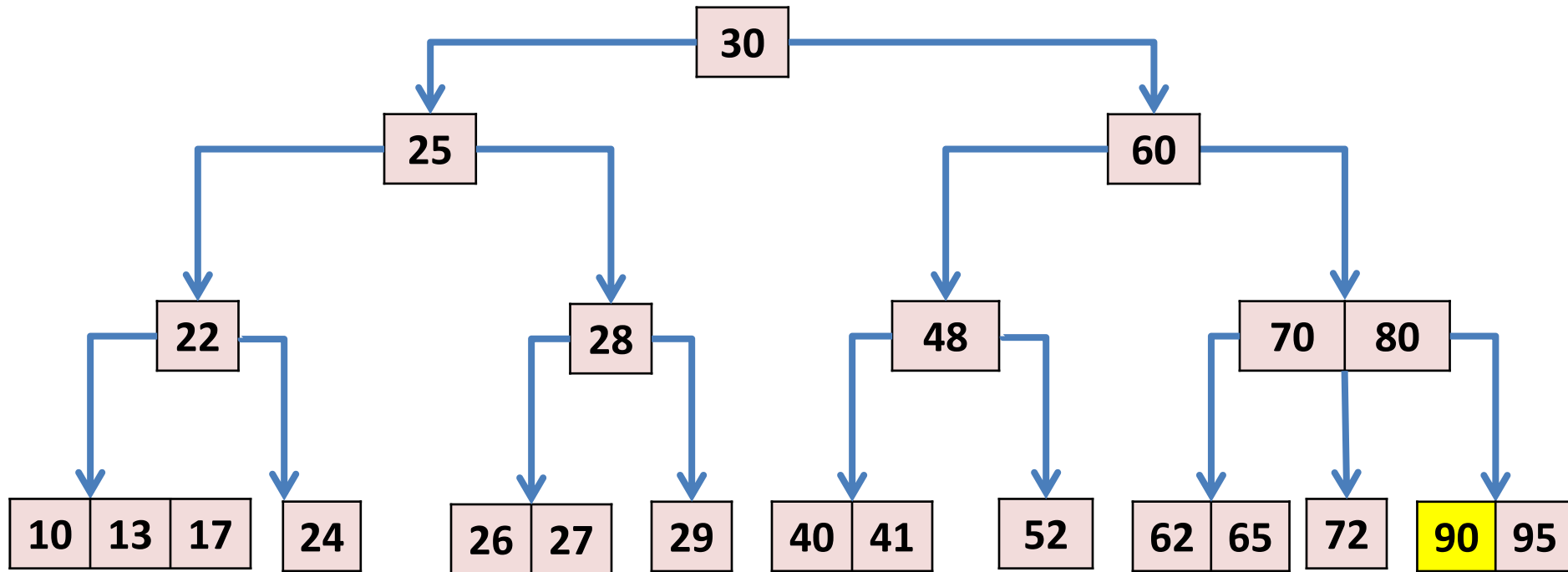
Insert 90 (part 5)

- Leaf, has space, insert 90.



Insert 90 (part 6)

- Leaf, has space, insert 90.



REMEMBER

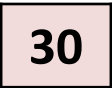
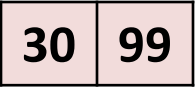
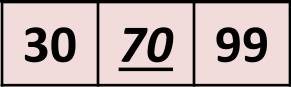
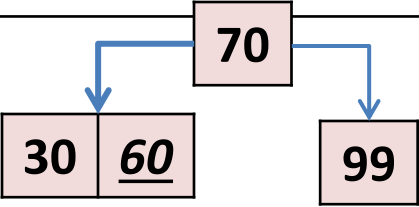
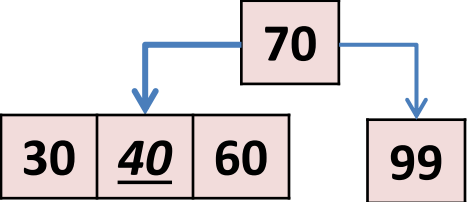
our convention: **preemptive split**

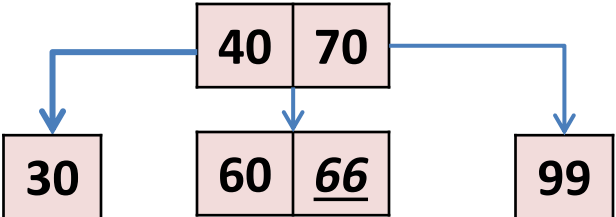
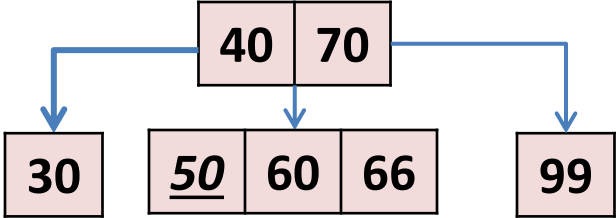
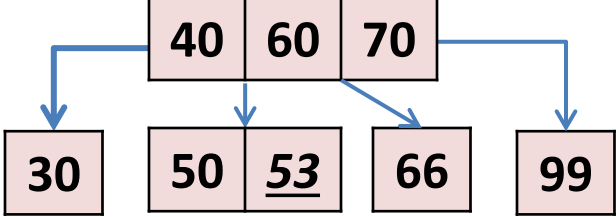
- If on your path to insert, you see a 4 node, you split it!
 - I will refer to this convention/choice as “preemptive split”
- You do that even if there is room in the leaf and you can insert without splitting this node.

Example: Build a Tree

- In an empty tree, insert items given in order:
30, 99, 70, 60, 40, 66, 50, 53, 45, 42
- On the [Data Structures Visualization](#) website, if you select *Max degree = 4* and “*preemptive split*”, it will follow the same process.

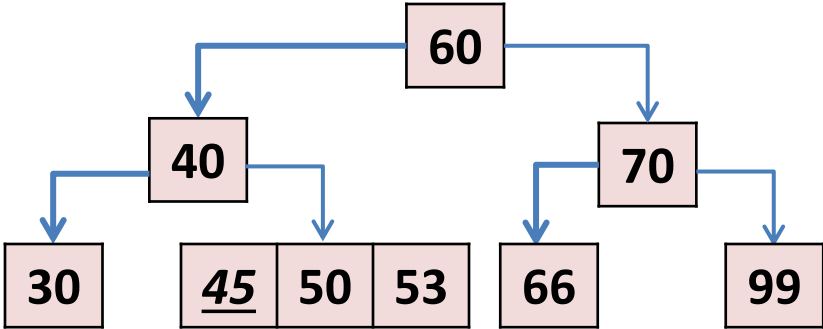
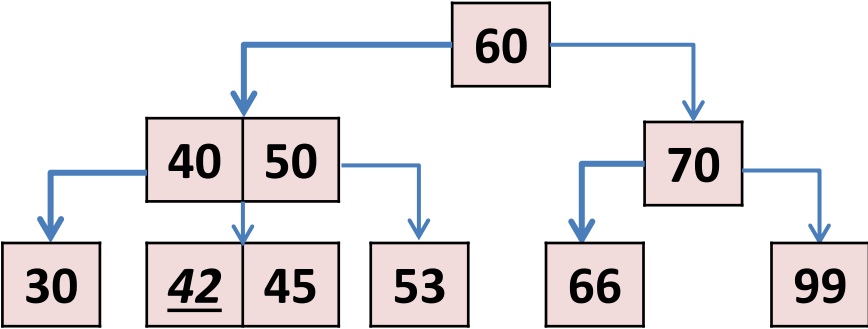
Building a Tree

Node to insert	Tree
30	
99	
70	
60	
40	

Node to insert	Tree
66	
50	
53	<p>Preemptive split of node 50\60\66. Root becomes full but does not split now.</p> 

Continues on next page ...

Building a Tree

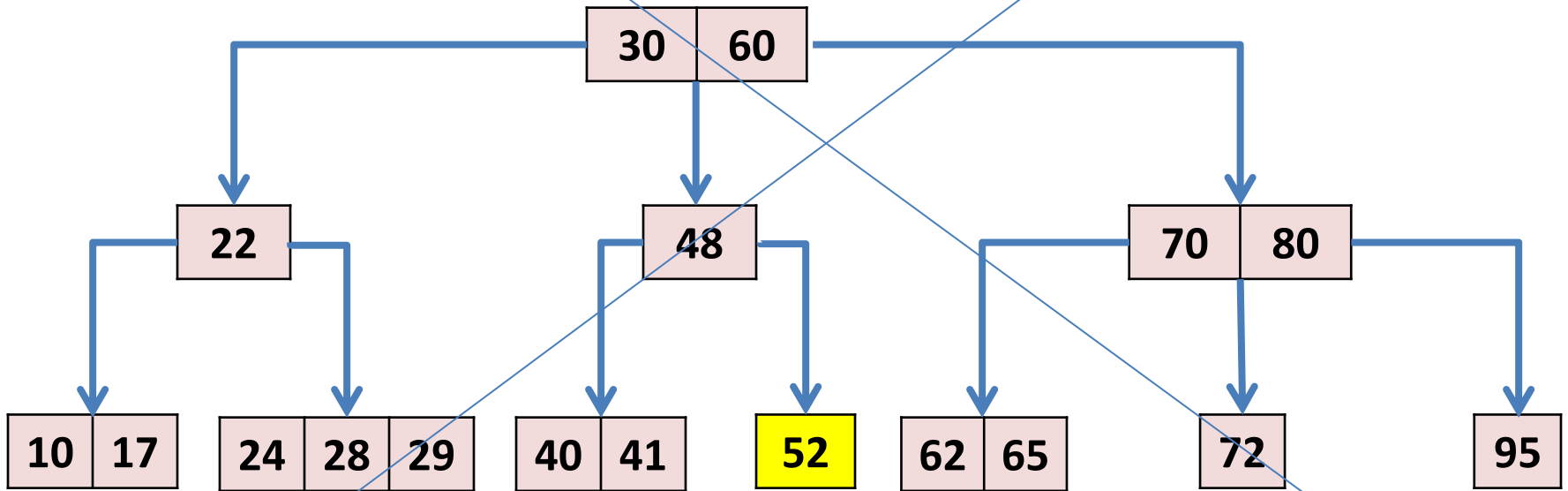
Node to insert	Tree
<p>45</p> <p>Preemptive split at root (split root even though node 50 53 had room)</p>	 <pre> graph TD 60[60] --> 40[40] 60 --> 70[70] 40 --> 30[30] 40 --> 45_50_53["45 50 53"] 70 --> 66[66] 70 --> 99[99] </pre>
<p>42</p>	 <pre> graph TD 60[60] --> 40_50["40 50"] 60 --> 70[70] 40_50 --> 30[30] 40_50 --> 42_45["42 45"] 40_50 --> 53[53] 70 --> 66[66] 70 --> 99[99] </pre>

Deletion in 2-3-4 Trees

- More complicated.
 - Sedwick book does not cover it.
- Idea: in order to delete item x (with key k) search for k . When find it:
 - If in a leaf remove it,
 - Else replace it with the successor of x , y . (y is the item with the first key larger than k .) Note: y will be in a leaf.
 - remove y and put it in place of x .
- When removing y we have problems as with insert, but now the nodes may not have enough keys (need 2 or 3 keys) => fix nodes that have only one key on the path from root to y .

Delete 52

- Delete item with key 52:



- How about deleting item with key 95:

Deletion in 2-3-4 Trees

- Case 1: leaf has 2 or more items: remove y
- Case 2: node on the path has 2 or more items: fine
- Case 3: node on the path has only 1 item
 - A) Try to get a key from the sibling – must rotate with the parent key to preserve the order property
 - B) If no sibling has 2 or more keys, get a key from the parent and merge with your sibling neighboring that key.
 - C) The parent is the root and has only one key (and therefore exactly 2 children): merge the root and the 2 siblings together.

Self Balancing Binary Trees

- Red-Black trees
- AVL trees
- Splay trees
-

Red-Black

- Red-Black trees (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)
 - Red & black nodes
 - Every node is either red or black
 - Root is black – CLRS requirement, but not in other works
 - a red node will have both his children black,
 - all NIL nodes are black (NIL = unexisting child node)
 - For every node, any path from it to a leaf will have the same number of black nodes (i.e. same ‘black height’) => actual path lengths differ by at most a factor of two (cannot have 2 consecutive red nodes)
 - 2-3-4 trees can be mapped to red-black trees: 2-node = 1 black node, 3-node = 1 black & 1 red (left or right) 4 node = 1 black & 2 red children (see [wikipedia](#))

AVL

- AVL trees (<https://www.youtube.com/watch?v=FNeL18KsWPc&t=2586s>)
 - Height of the two children of any node differs by 1 at most. Go to <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html> and insert numbers in order: 41, 20, 65, 11, 29, 50, 26 (to build the tree from the MIT example) and then insert 23 and then 55.
 - In CLRS and MIT video leaves have height 0.
 - In the visualization leaves have height 1 (all heights are off by 1)
 - Also start with empty tree and insert: 30, 50, **70**, 90, **85**, **60**

Splay trees

- Splay trees
 - Self adjusting: the items used more recently (inserted or read/visited) move towards the top.
 - The tree is not balanced, but performs well because it brings the frequent items to the top.
 - Splay insertion: the new item is inserted at the root by a ***rotation that replaces a node with his grandchild***.
 - Search in the tree for the new node. It takes you to a leaf location. Insert new node there and continue with repeated rotations to bring it to the root. This process will reduce the length of that original path to half (because with each rotation, the grandchild node moves two levels up => the path on which these rotations are done, is cut in half).
 - See visualization: <https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>
 - For more on splay trees see Sedgewick, pages 540-546.

Java

- [TreeMap](#) – Red-Black tree
- [TreeSet](#) – implementation based on TreeMap