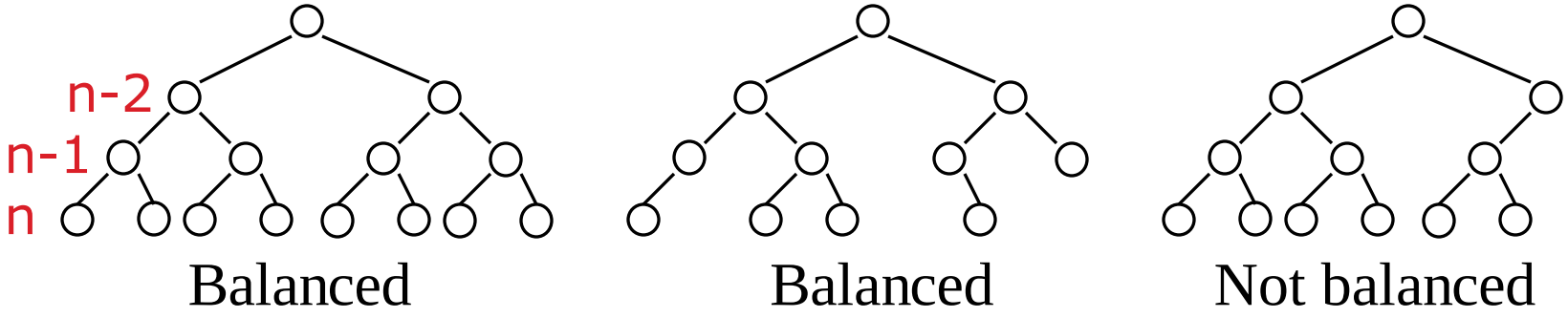
Unit V: Multiway trees

Heap data structure

- Binary tree
- Balanced
- Left-justified or Complete
- Heap property:
  - (Max): no node has a value greater than the value in its parent
  - (Min): no node has a value lesser than the value in its parent

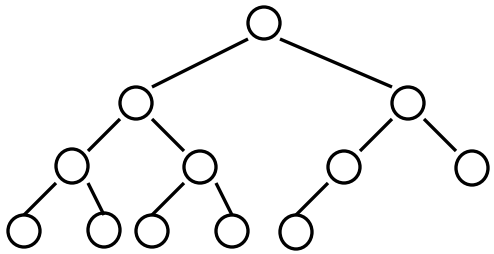
Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children

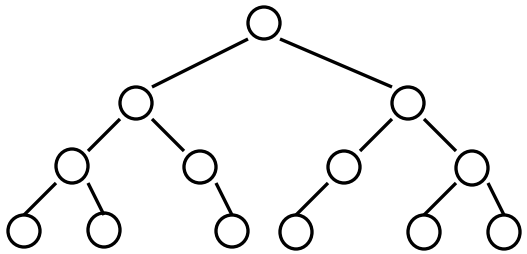


Left-justified binary trees

- A balanced binary tree of depth n is left-justified if:
  - it has 2^n nodes at depth n (the tree is "full"), or
  - it has 2^k nodes at depth k, for all $k < n$, *and* all the leaves at depth n are as far left as possible



Left-justified



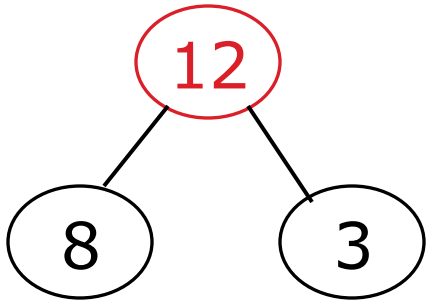
Not left-justified

Building up to heap sort

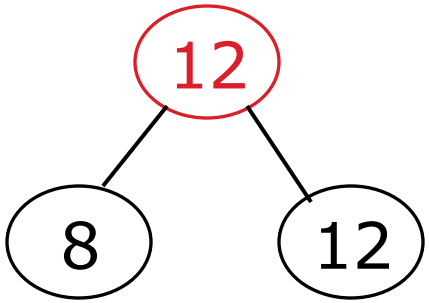
- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

The heap property

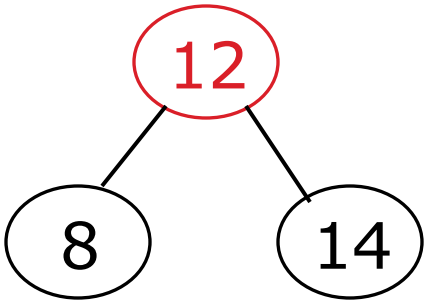
- A node has the heap property(Max Heap) if the value in the node is as large as or larger than the values in its children



Red node has heap property



Red node has heap property

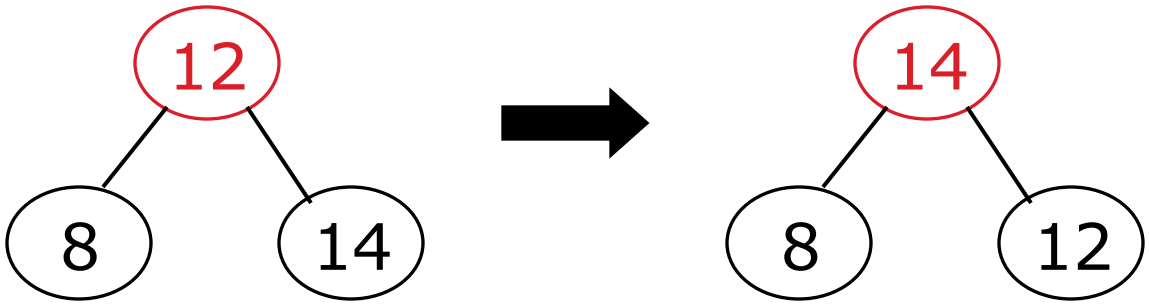


Red node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

Sift Up/Heapify/Heap Formation

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



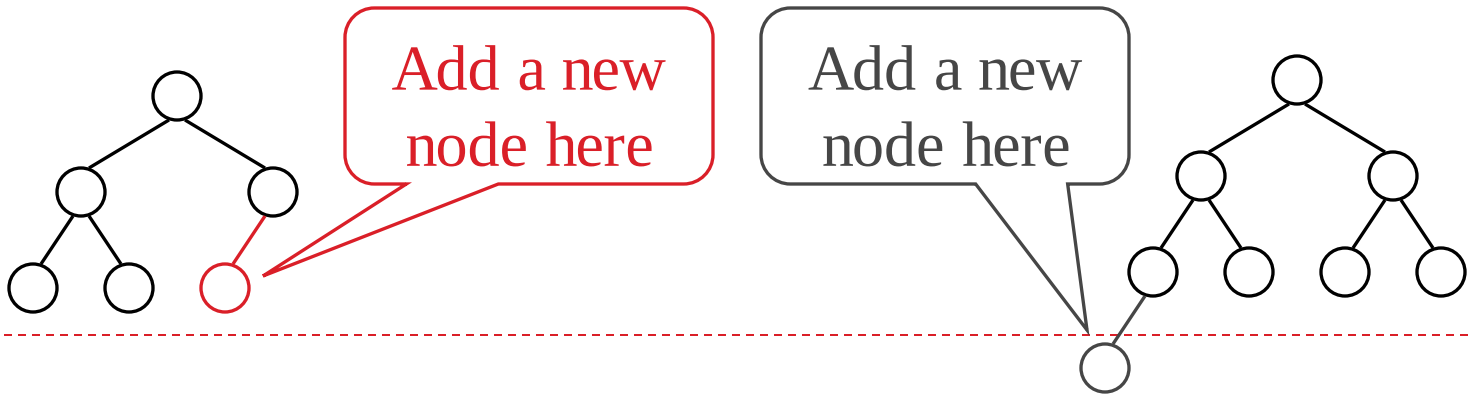
Red node does not have heap property

Red node has heap property

  - This is sometimes called sifting up

Constructing a heap I

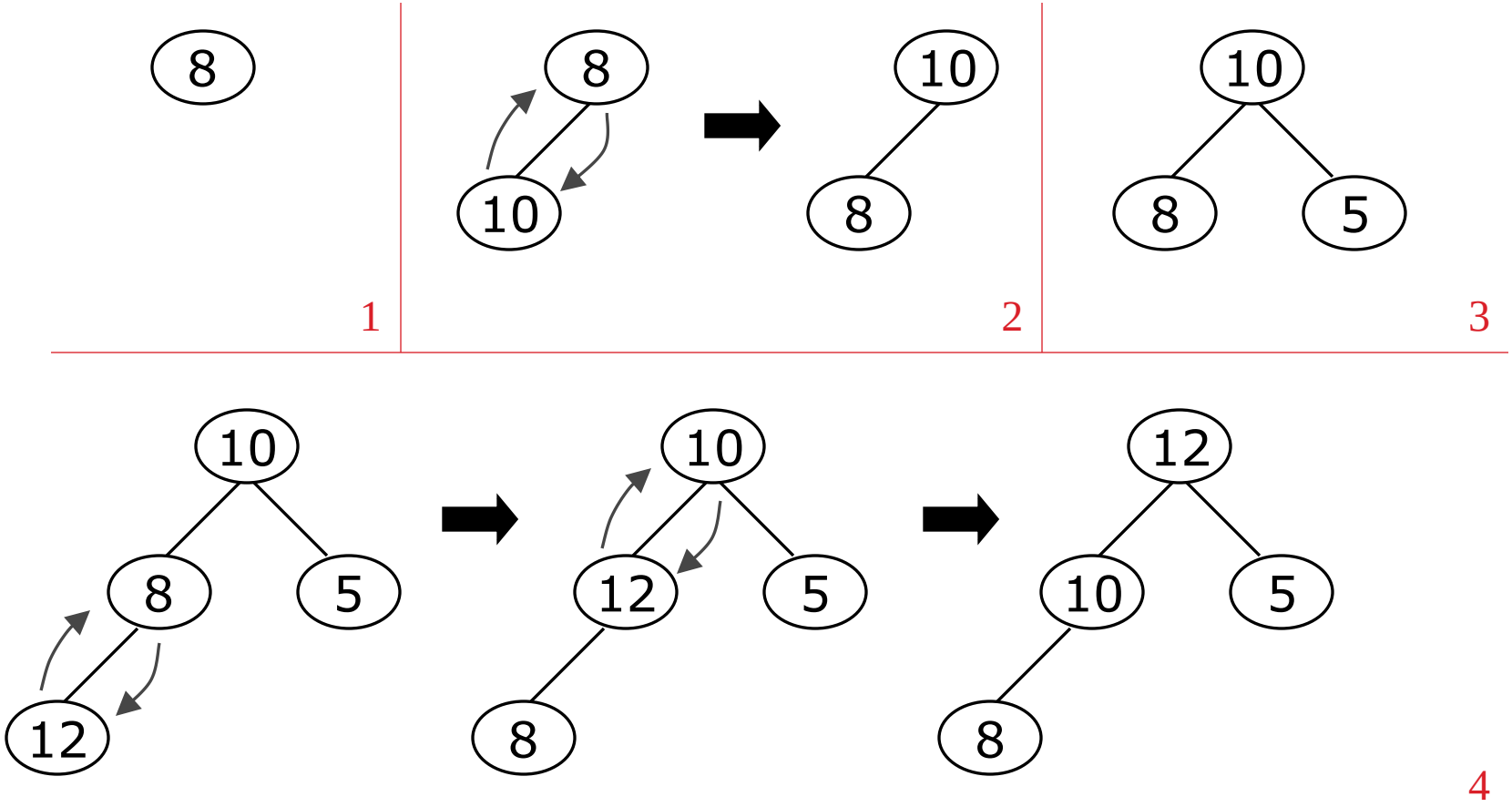
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



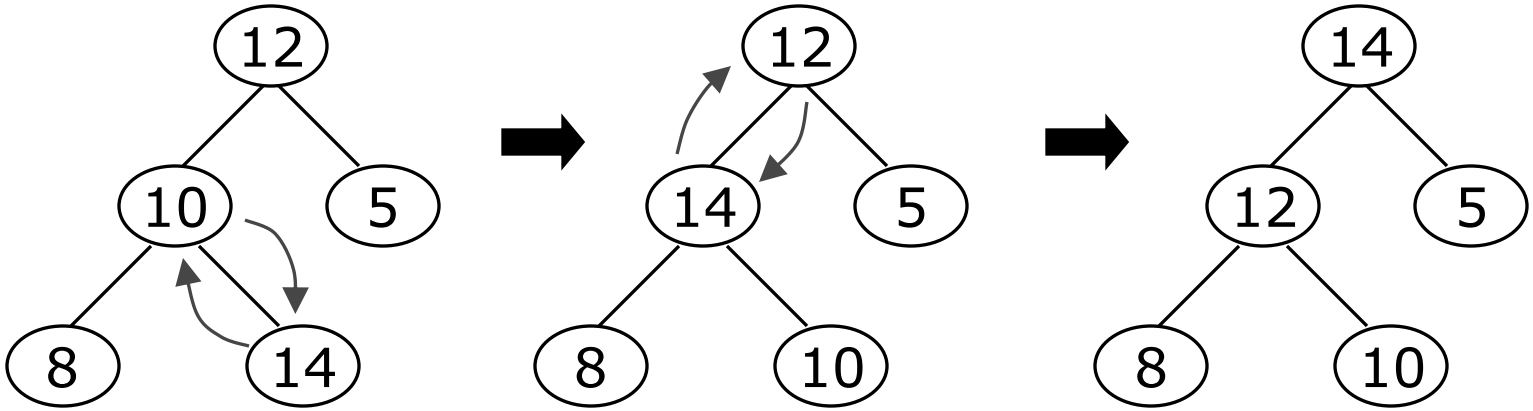
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III



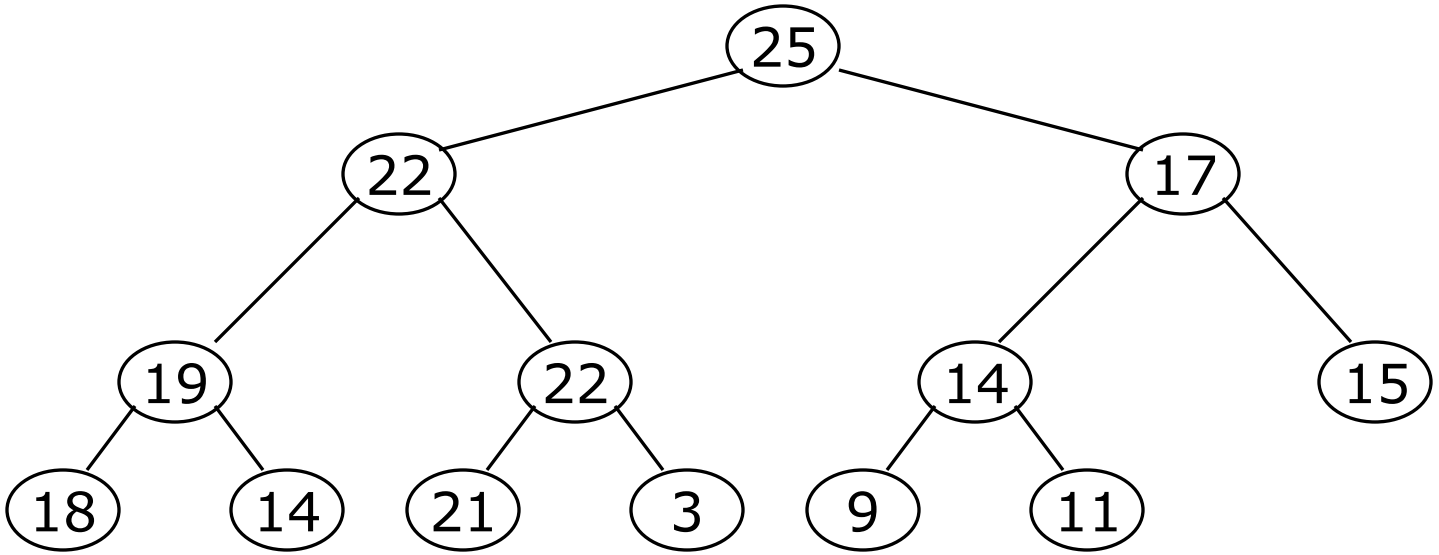
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

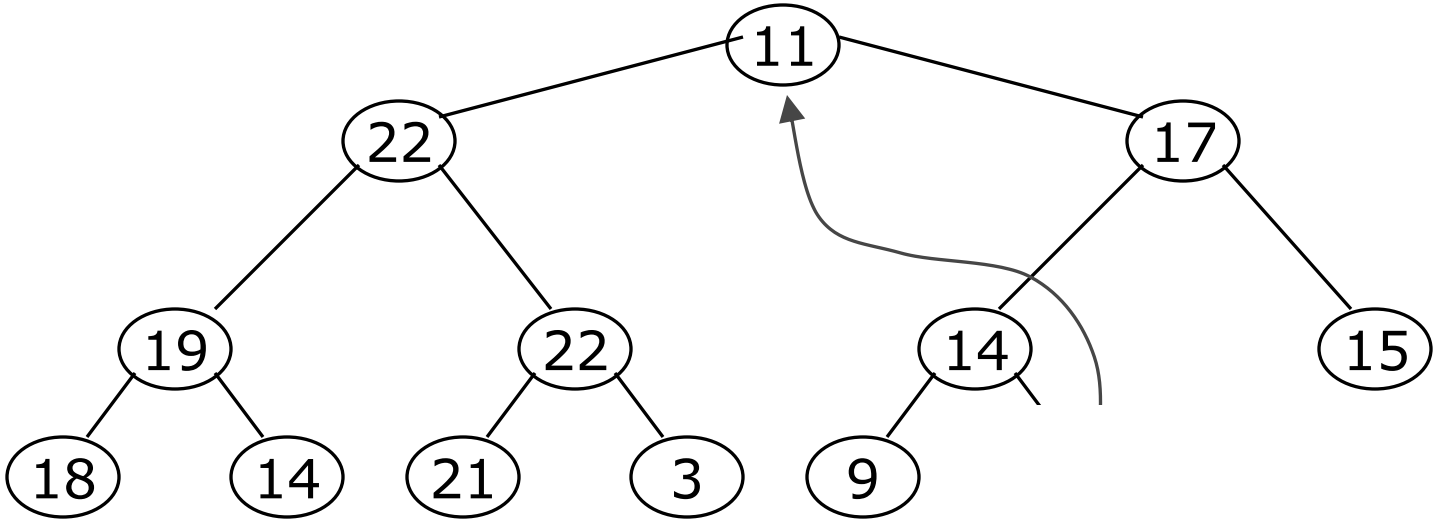
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root (animated)

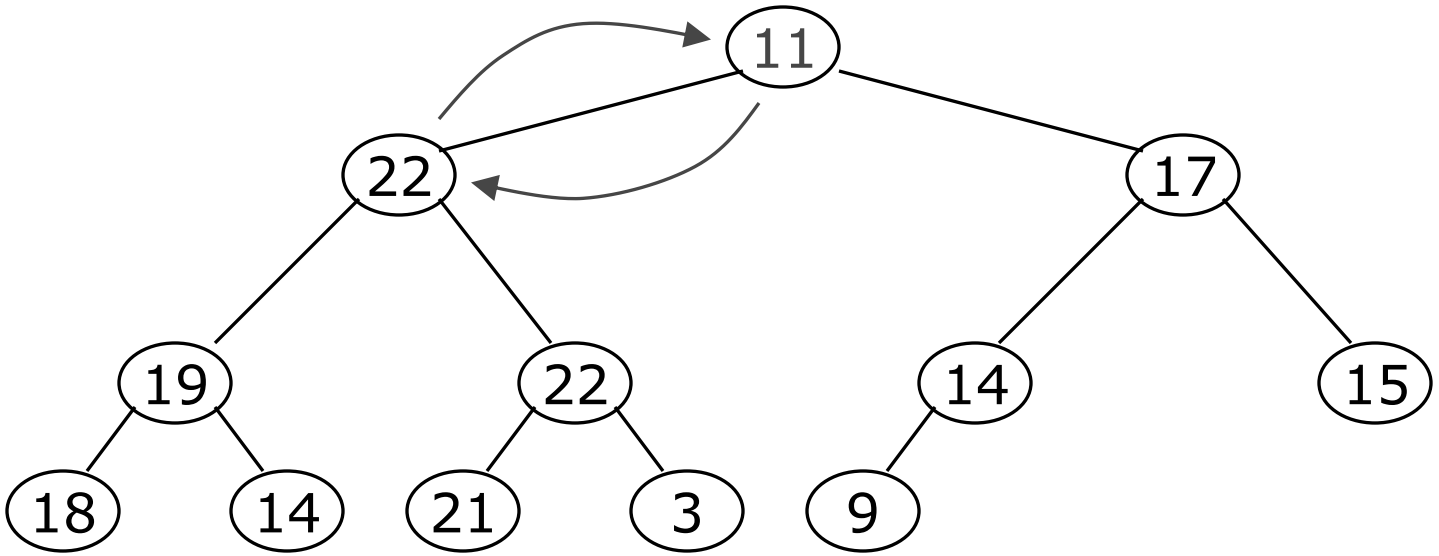
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

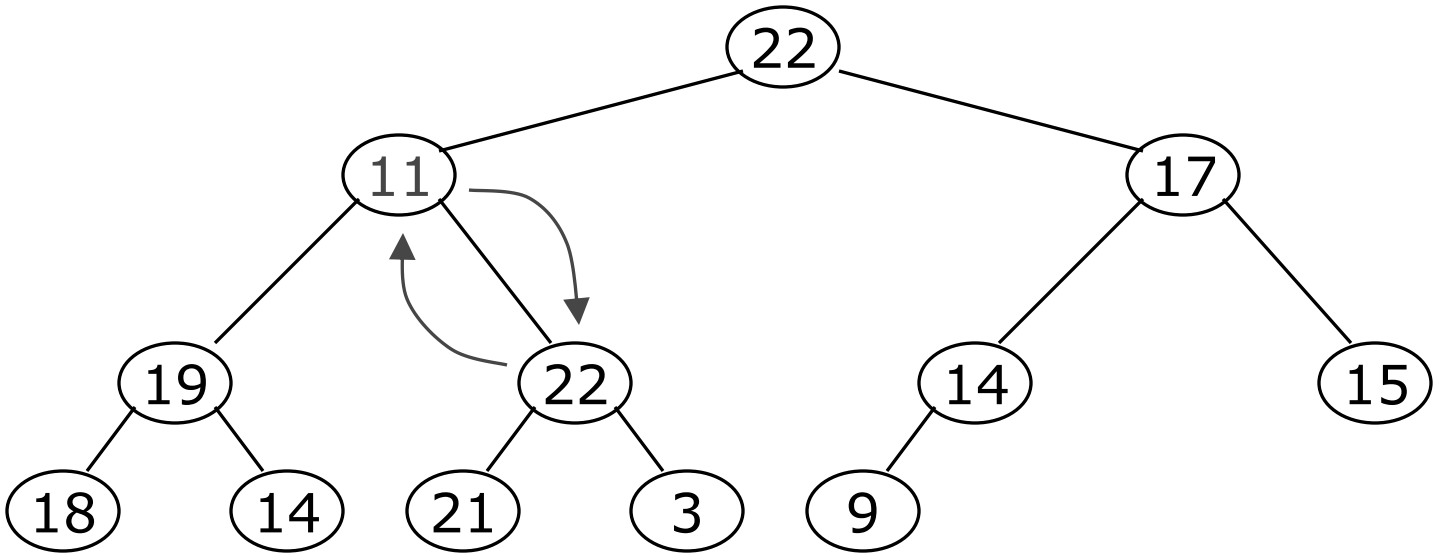
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can siftDown() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

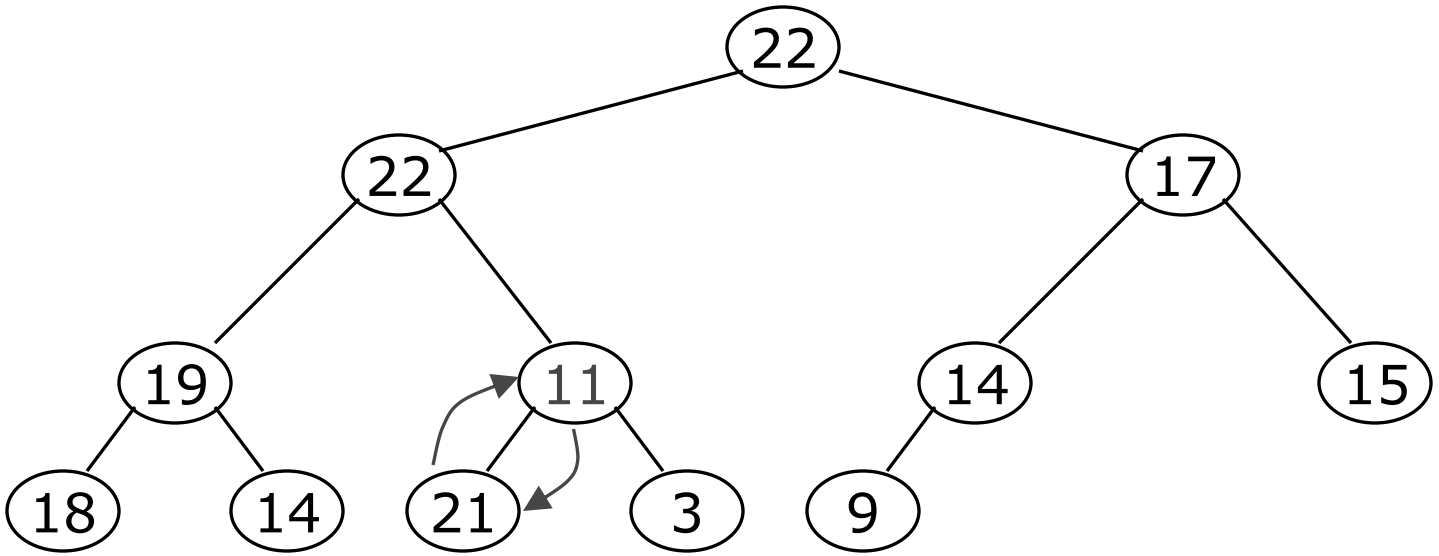
- Now the left child of the root (still the number `11`) lacks the heap property



- We can siftDown() this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method III

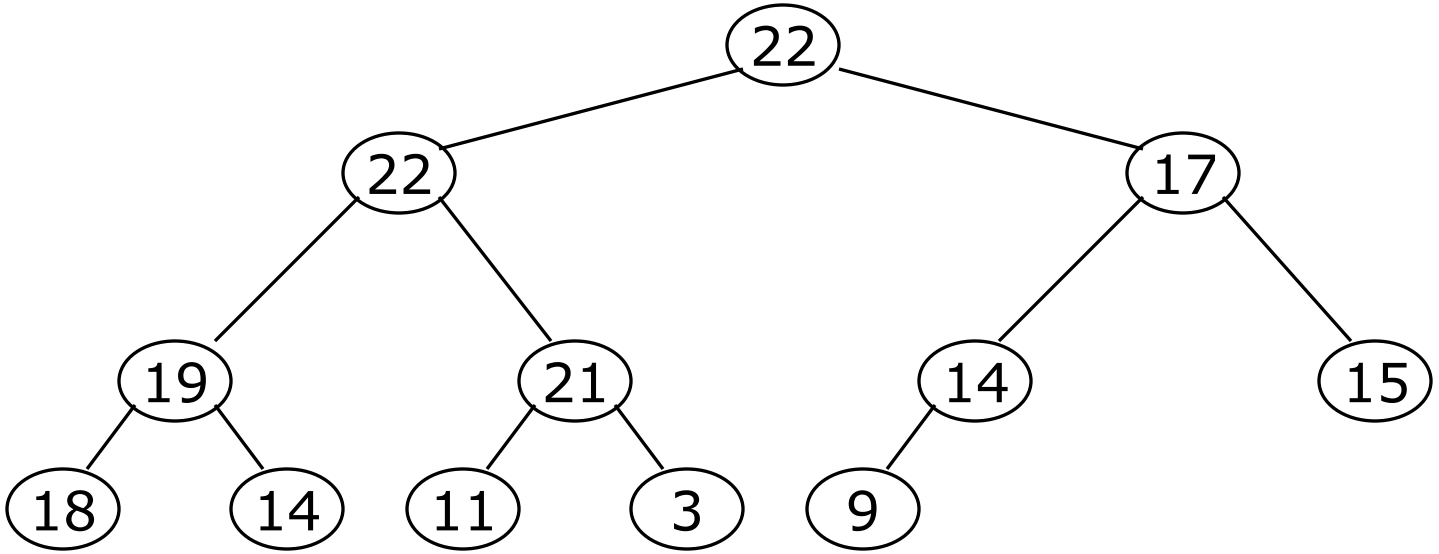
- Now the right child of the left child of the root (still the number `11`) lacks the heap property:



- We can siftDown() this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

- Our tree is once again a heap, because every node in it has the heap property



- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *left justified*, it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Key properties

- Determining location of root and “last node” take constant time

- Remove n elements, re-heap each time

Analysis

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Multiway Search Trees

- B-Trees
- Search
- Insertion
- Deletion

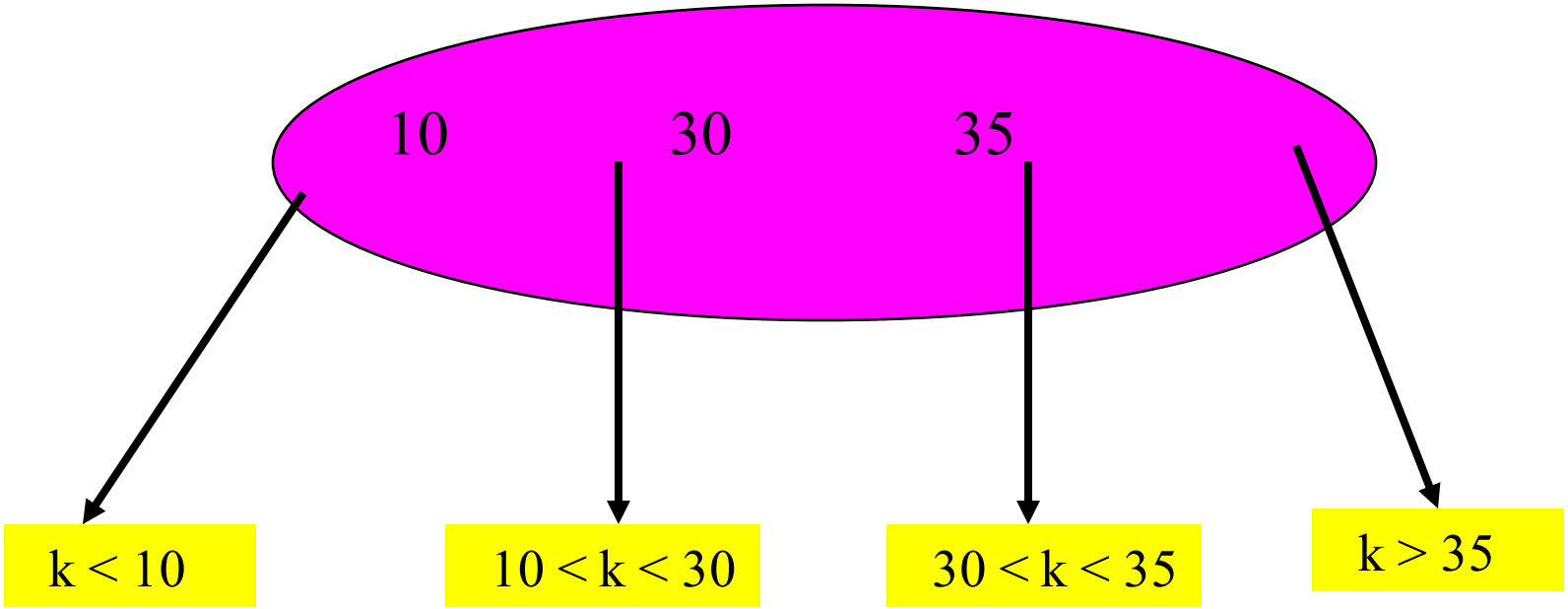
AVL Trees

- $n = 2^{30} = 10^9$ (approx).
- $30 <= height <= 43$.
- When the AVL tree resides on a disk, up to 43 disk access are made for a search.
- This takes up to (approx) 4 seconds.
- Not acceptable.

m-way Search Trees

- Each node has up to $m - 1$ pairs and m children.
- $m = 2 \Rightarrow$ binary search tree.

4-Way Search Tree



Maximum no Of Pairs

- Happens when all internal nodes are m-nodes.
- Full degree m tree.
- No of nodes = $1 + m + m^2 + m^3 + \ldots + m^{h-1} = (m^h - 1)/(m - 1)$.
- Each node has $m - 1$ pairs.
- So, no of pairs = $m^h - 1$.

Capacity Of m-Way Search Tree

	m = 2	m = 200
h = 3	7	$8 * 10^6 - 1$
h = 5	31	$3.2 * 10^{11} - 1$
h = 7	127	$1.28 * 10^{16} - 1$

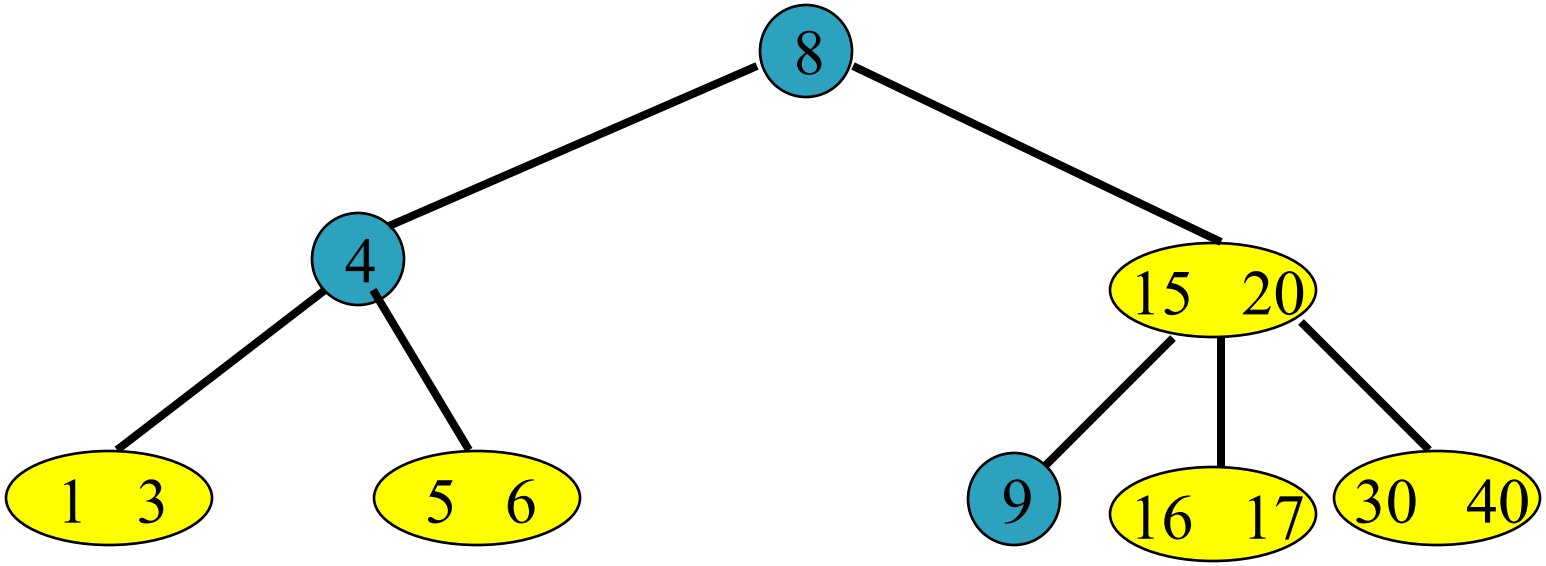
Definition Of B-Tree

- Definition assumes external nodes (extended m-way search tree).
- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

B Trees

- 2-3 tree is B-tree of order 3.
- 2-3-4 tree is B-tree of order 4.
- B-tree of order 5 is 3-4-5 tree (root may be 2-node though).
- B-tree of order 2 is full binary tree.

Insert



Insertion into a full leaf triggers bottom-up node splitting pass.

Split An Overfull Node

$m\ a_0\ p_1\ a_1\ p_2\ a_2\ \ldots\ p_m\ a_m$

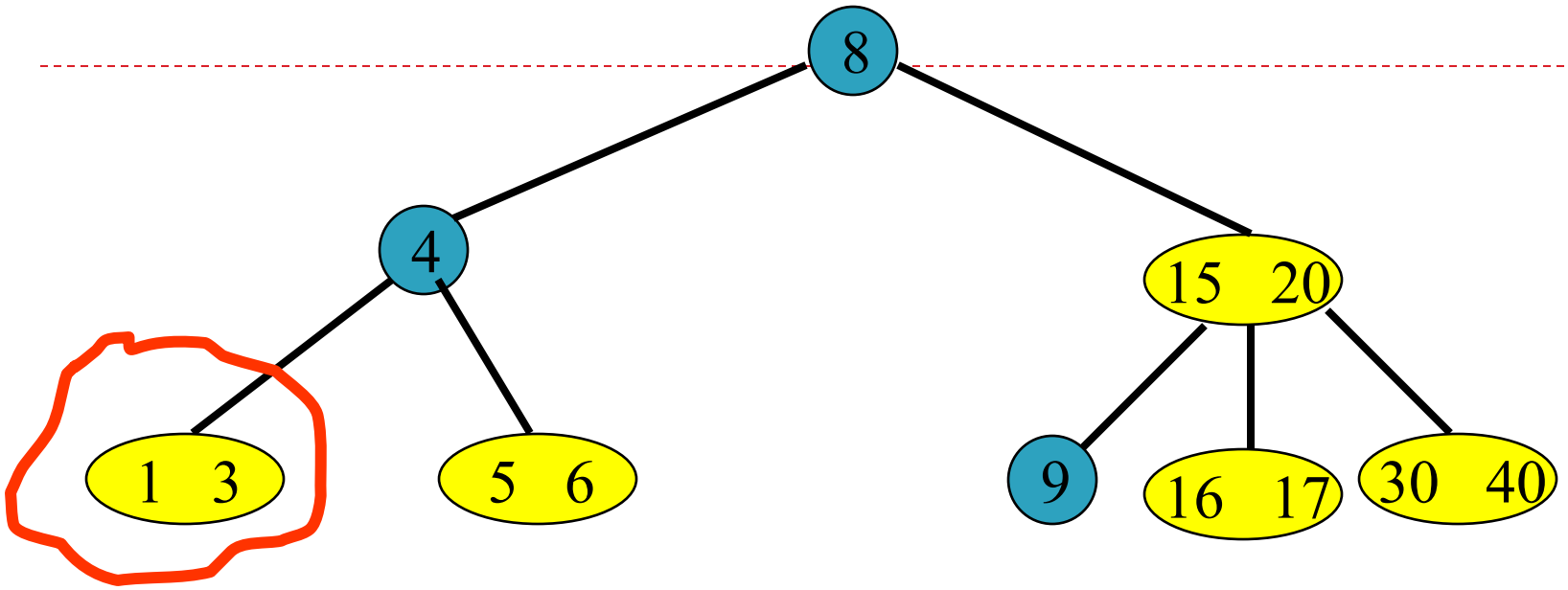
- a_i is a pointer to a subtree.
- p_i is a dictionary pair.

$ceil(m/2)\text{-}1\ \ a_0\ p_1\ a_1\ p_2\ a_2\ \ldots\ p_{ceil(m/2)-1}\ a_{ceil(m/2)-1}$

$m\text{-}ceil(m/2)\ \ a_{ceil(m/2)}\ p_{ceil(m/2)+1}\ a_{ceil(m/2)+1}\ \ldots\ p_m\ a_m$

- $p_{ceil(m/2)}$ plus pointer to new node is inserted in parent.

Insert

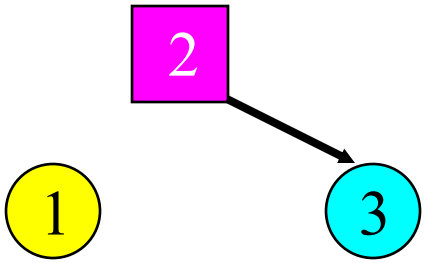


- Insert a pair with key = 2.
- New pair goes into a 3-node.

Insert Into A Leaf 3-node

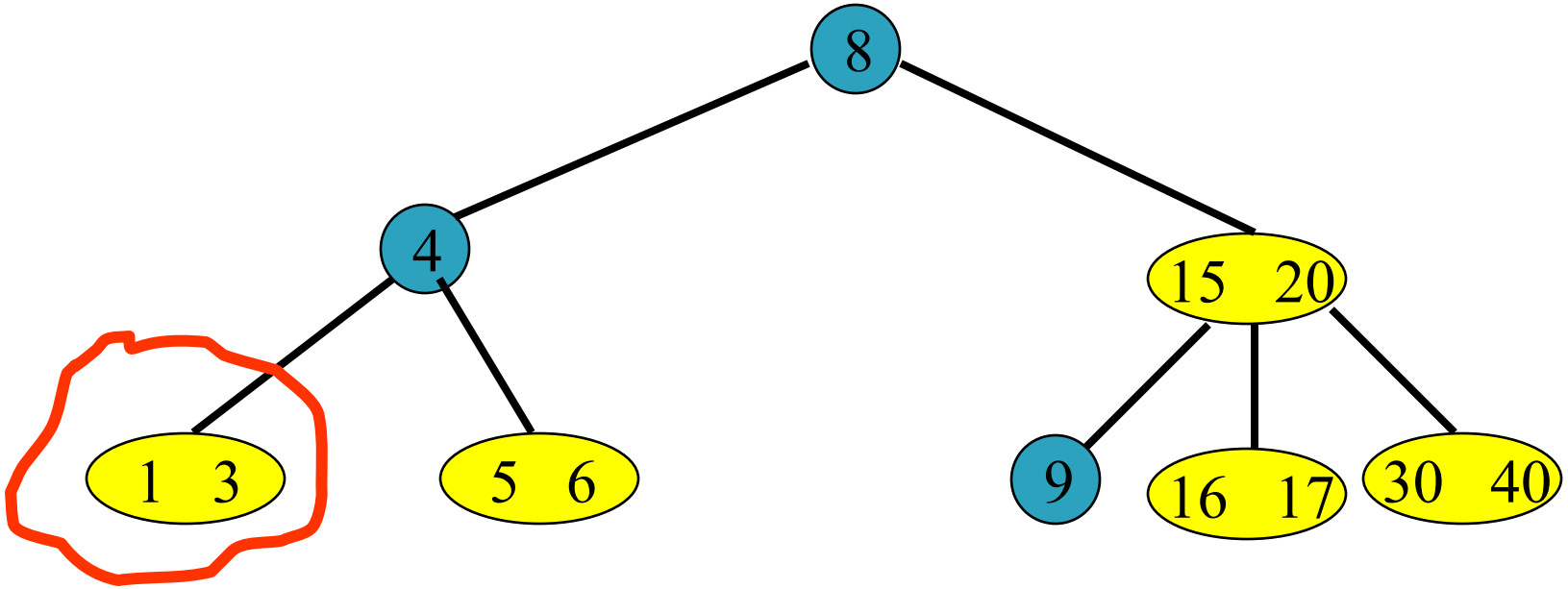
- Insert new pair so that the 3 keys are in ascending order.

- Split overflowed node around middle key.



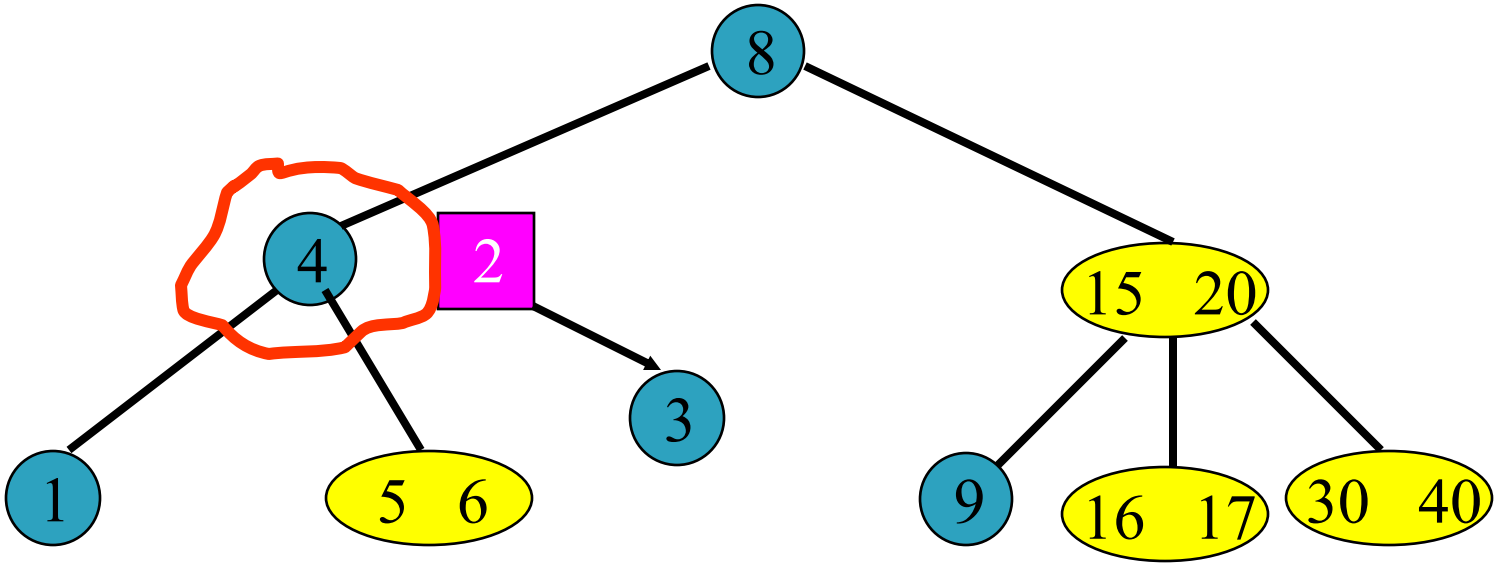
- Insert middle key and pointer to new node into parent.

Insert



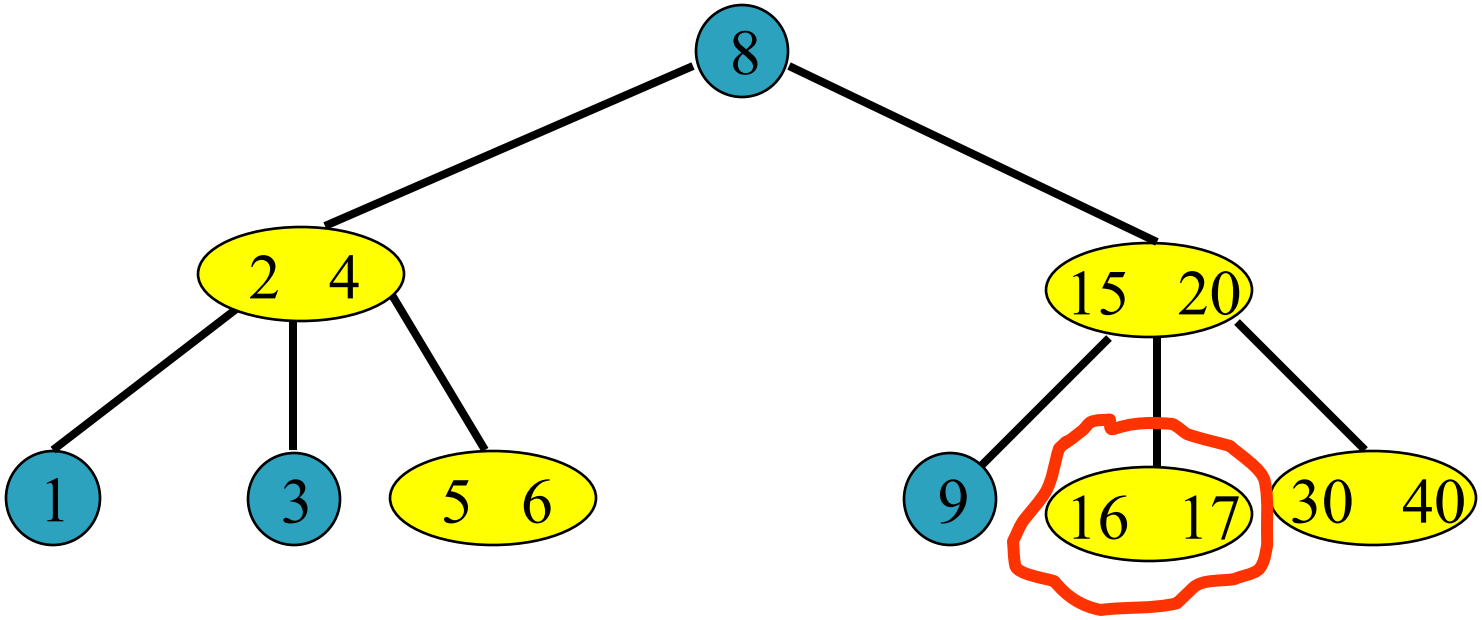
- Insert a pair with key = 2.

Insert



- Insert a pair with key = 2 plus a pointer into parent.

Insert

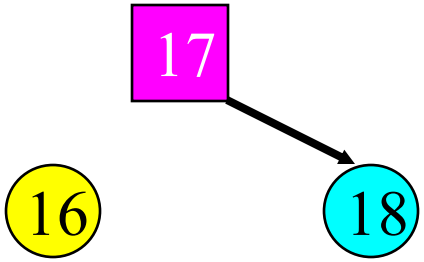


- Now, insert a pair with key = 18.

Insert Into A Leaf 3-node

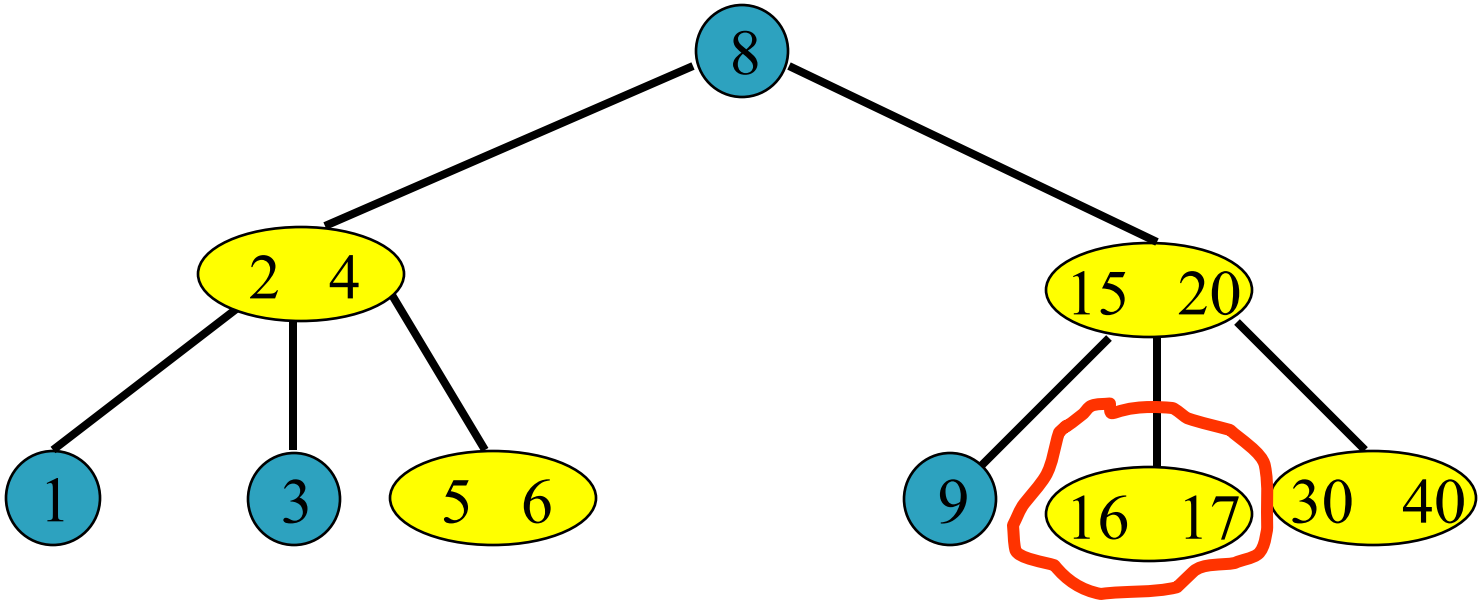
- Insert new pair so that the 3 keys are in ascending order.

  - Split the overflowed node.



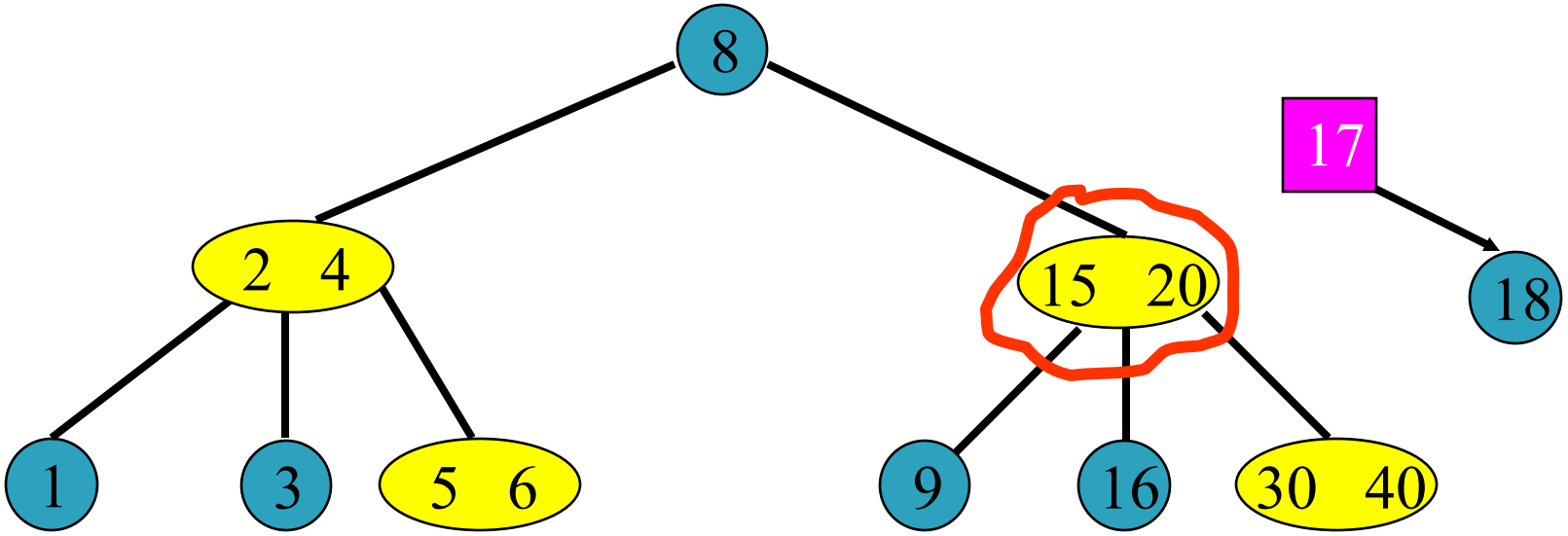
- Insert middle key and pointer to new node into parent.

Insert



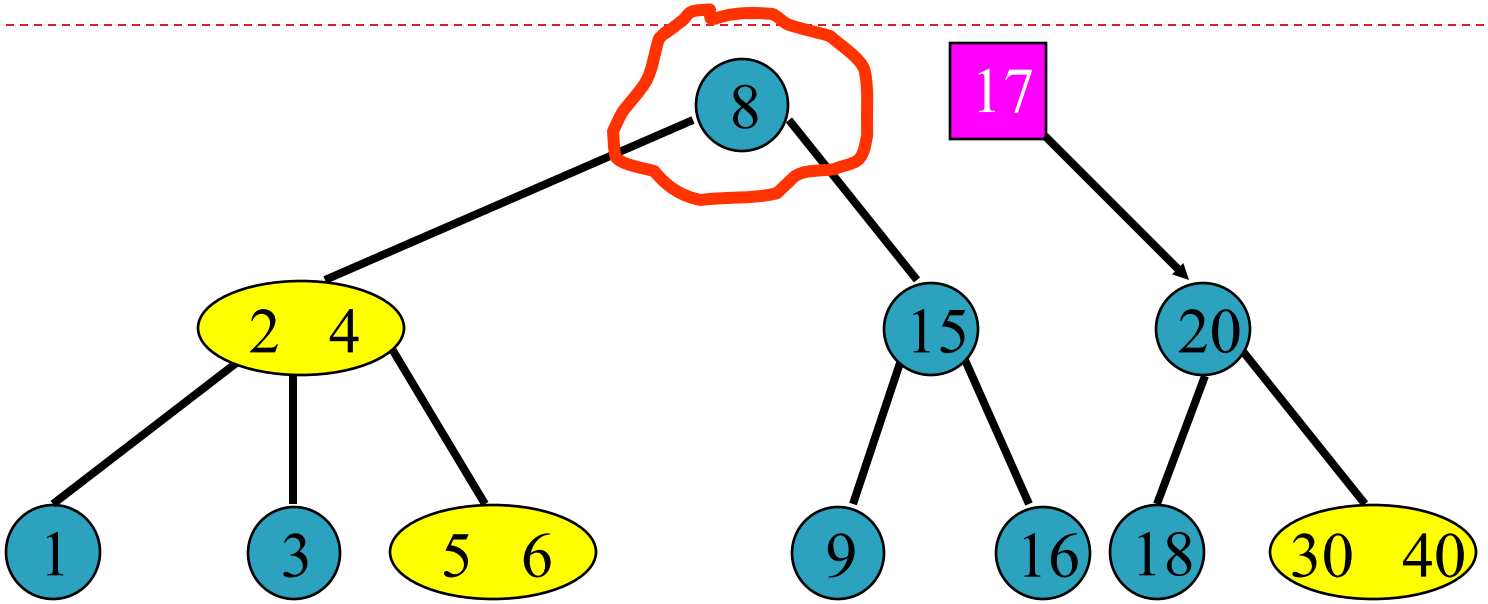
- Insert a pair with key = 18.

Insert



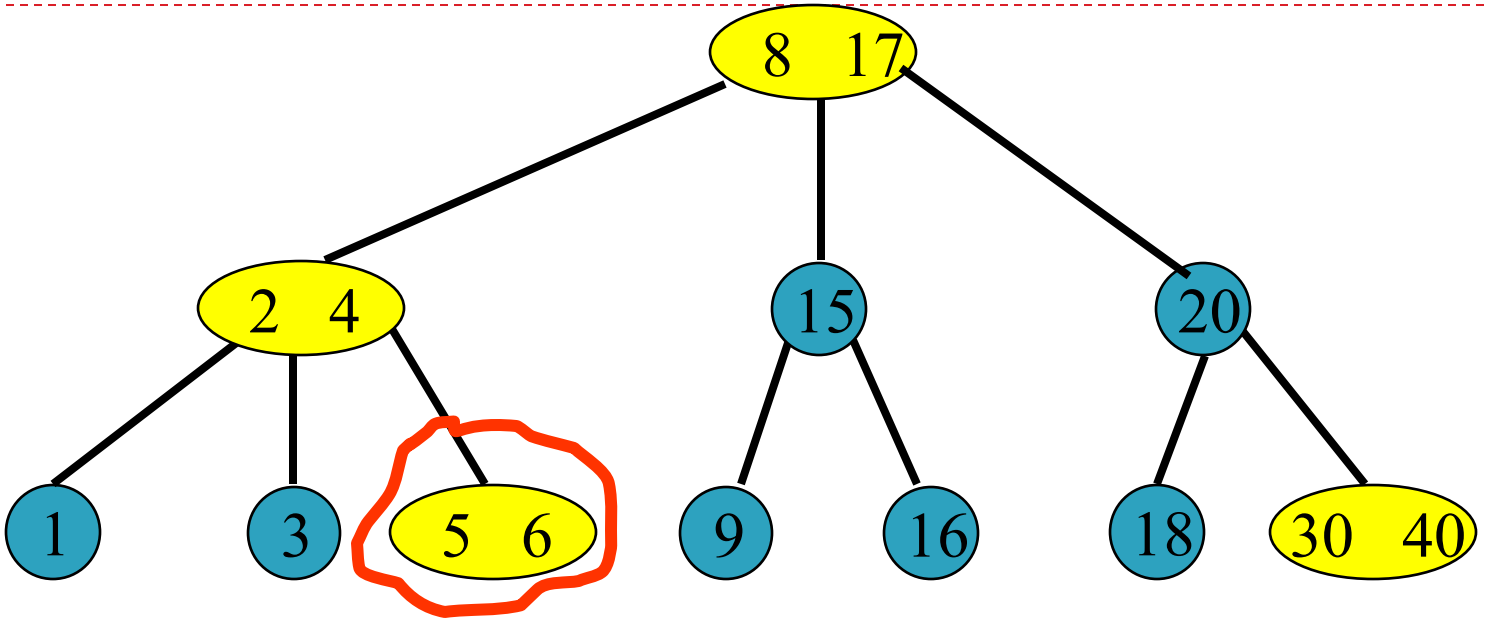
- Insert a pair with key = 17 plus a pointer into parent.

Insert



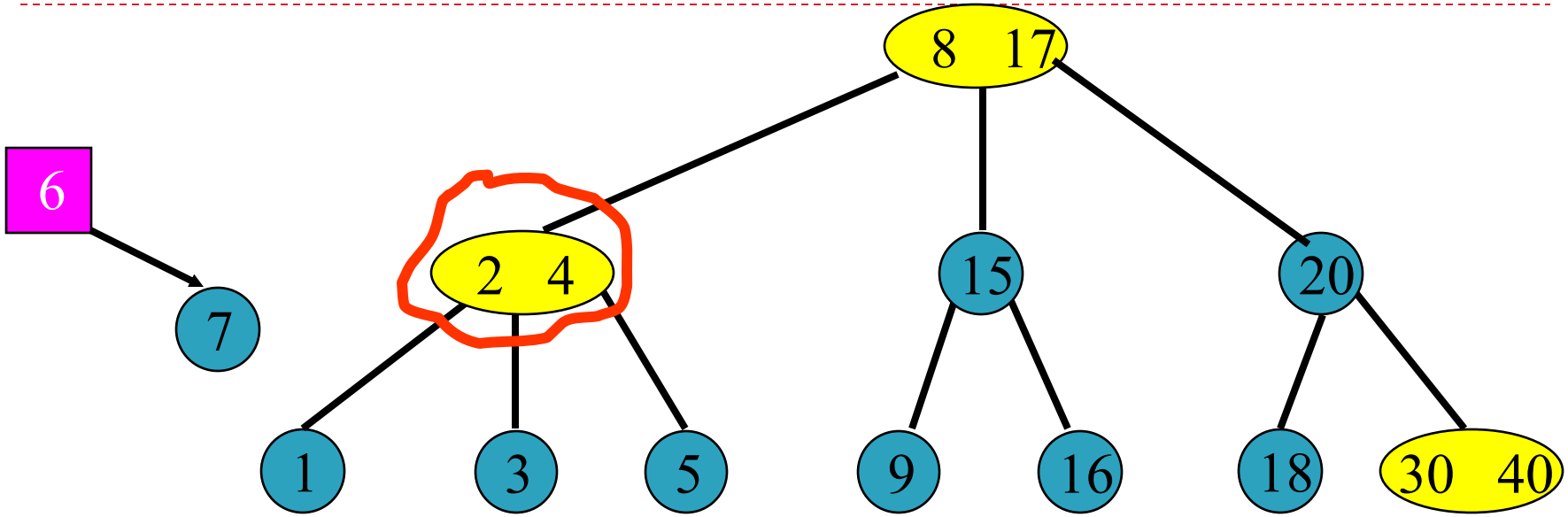
- Insert a pair with key = 17 plus a pointer into parent.

Insert



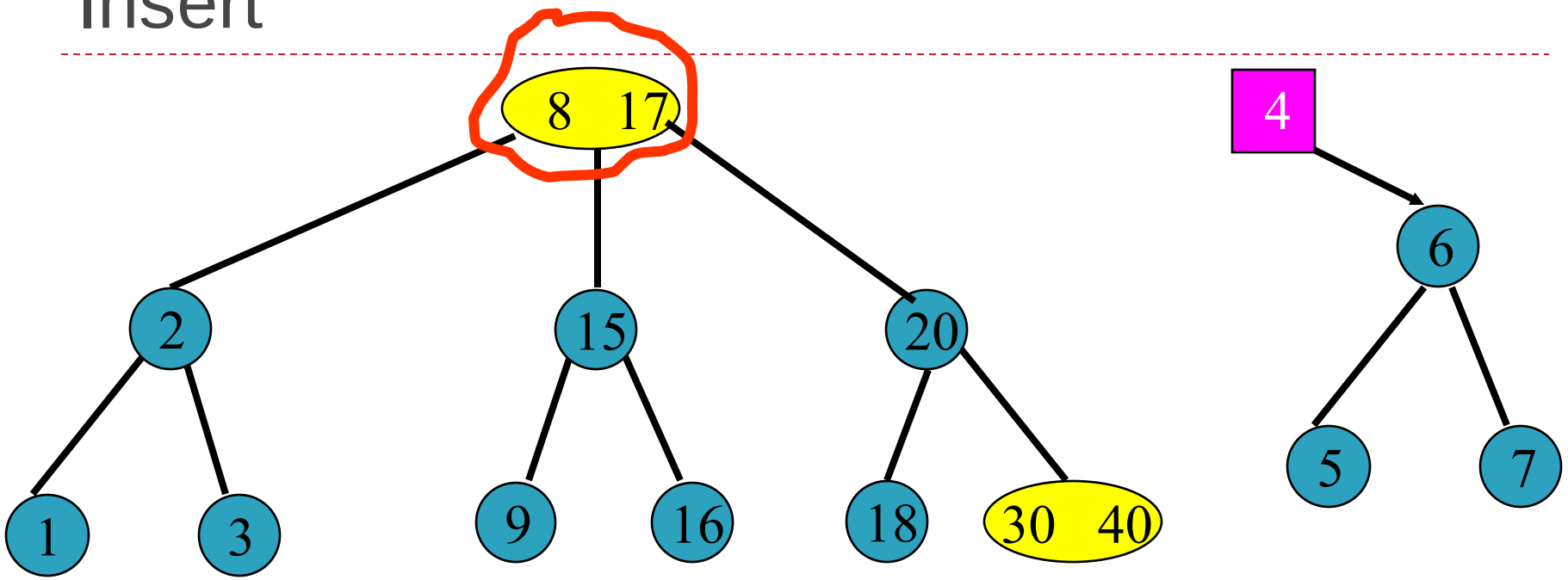
- Now, insert a pair with key = 7.

Insert



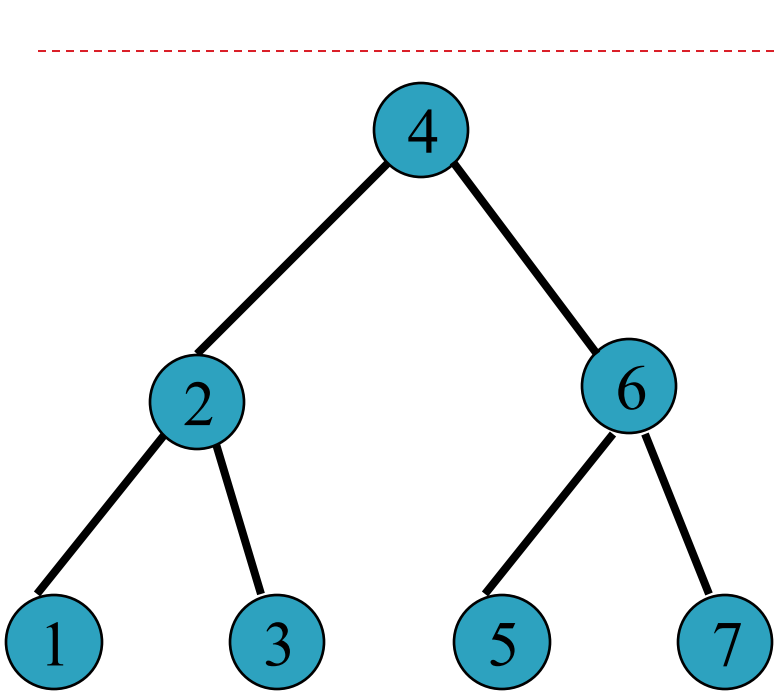
- Insert a pair with key = 6 plus a pointer into parent.

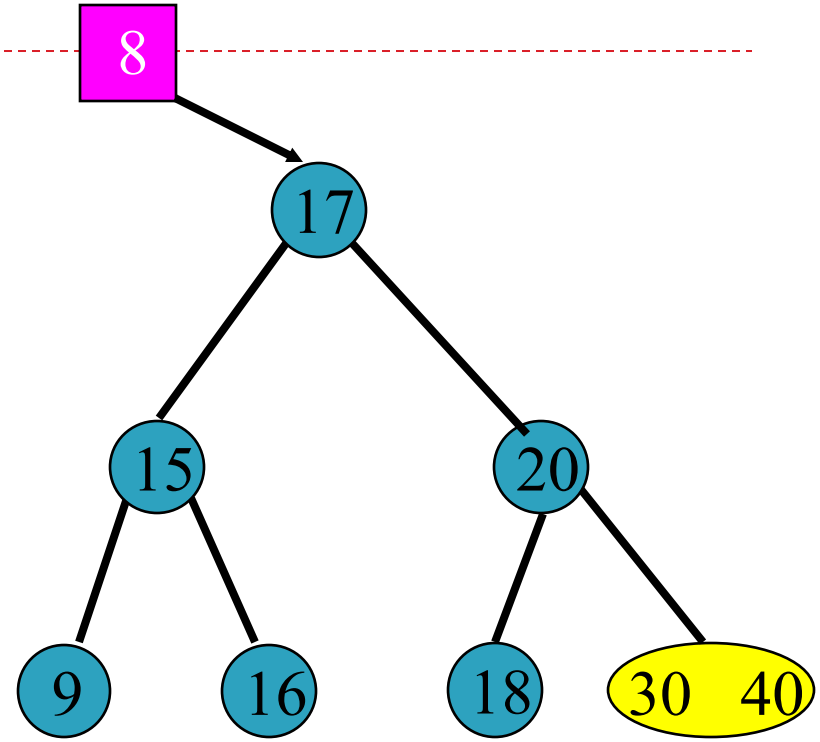
Insert



- Insert a pair with key = 4 plus a pointer into parent.

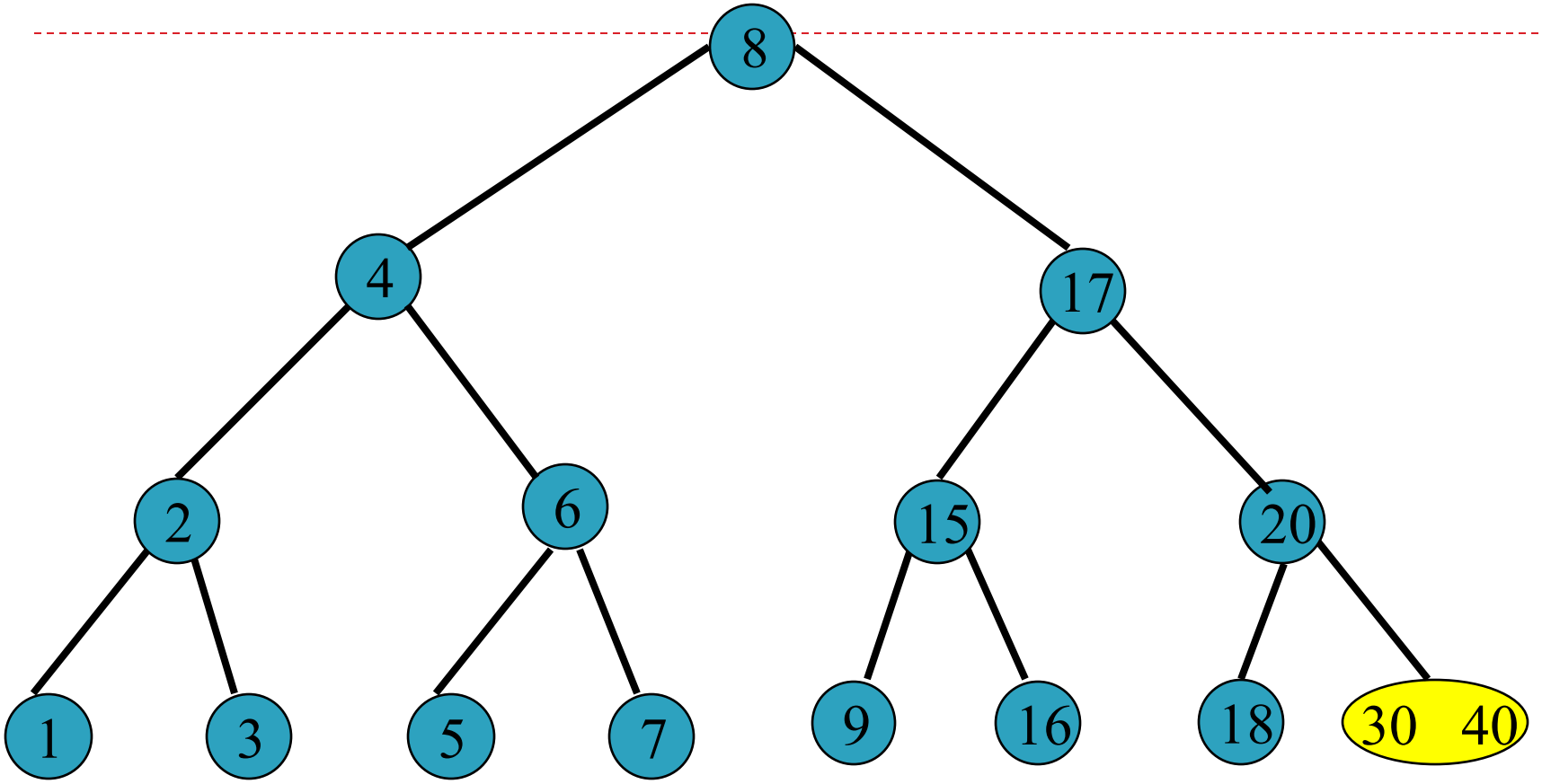
Insert





- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

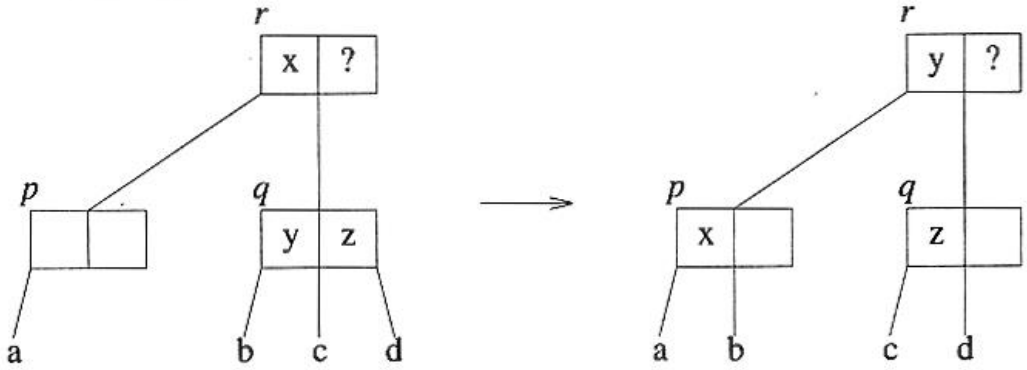
Insert



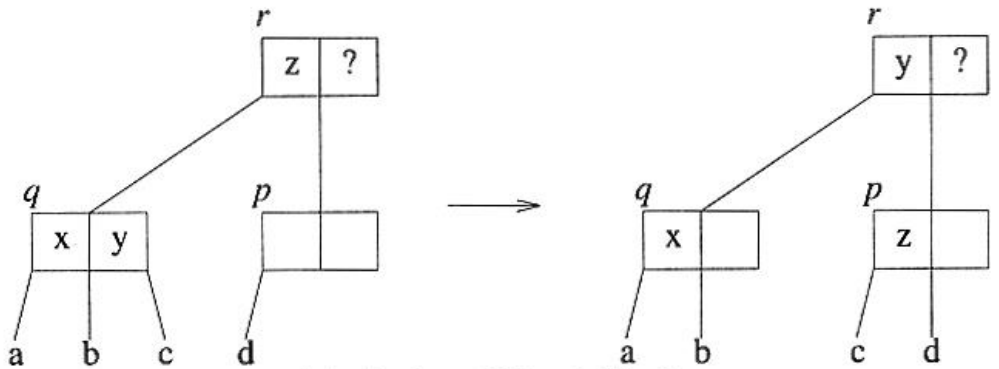
- Height increases by 1.

Delete A Pair

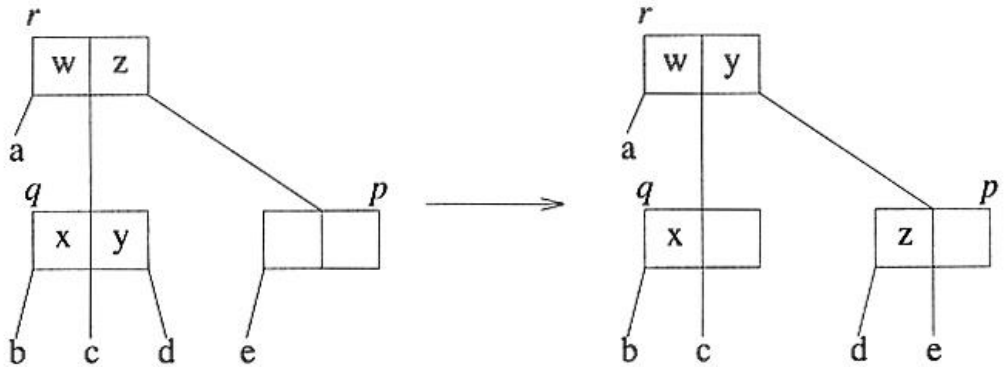
- Deletion from interior node is transformed into a deletion from a leaf node.
- Deficient leaf triggers bottom-up borrowing and node combining pass.
- Deficient node is combined with an adjacent sibling who has exactly ceil(m/2) − 1 pairs.
- After combining, the node has [ceil(m/2) – 2] (original pairs) + [ceil(m/2) – 1] (sibling pairs) + 1 (from parent) <= m –1 pairs.



(a) p is the left child of r



(b) p is the middle child of r



(c) p is the right child of r

Figure 11.7: The three cases for rotation in a 2-3 tree

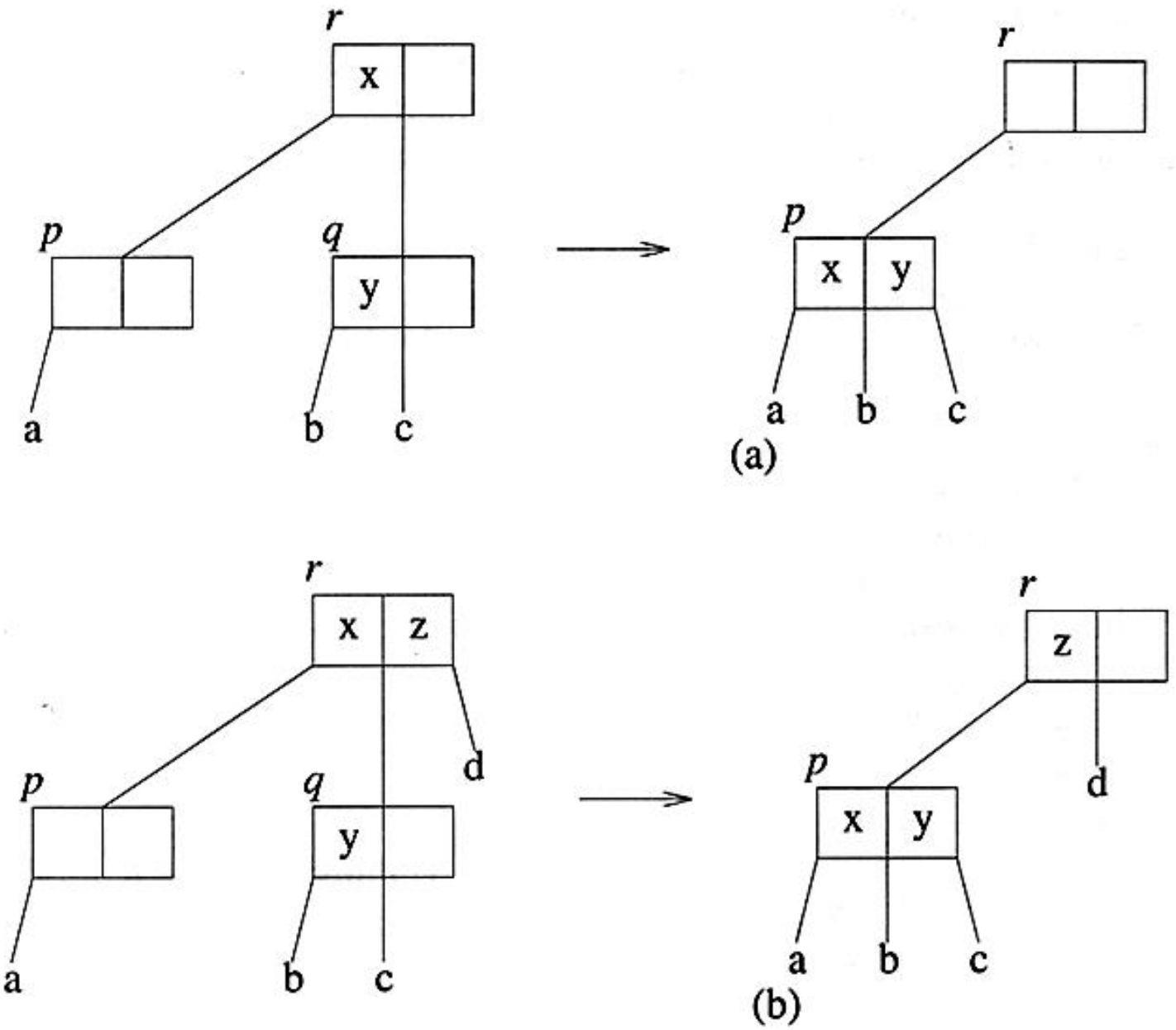
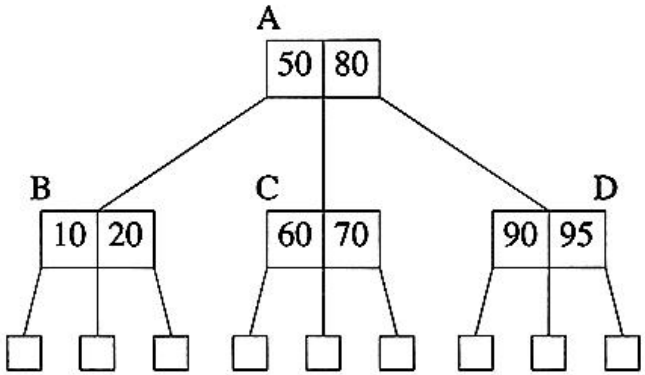
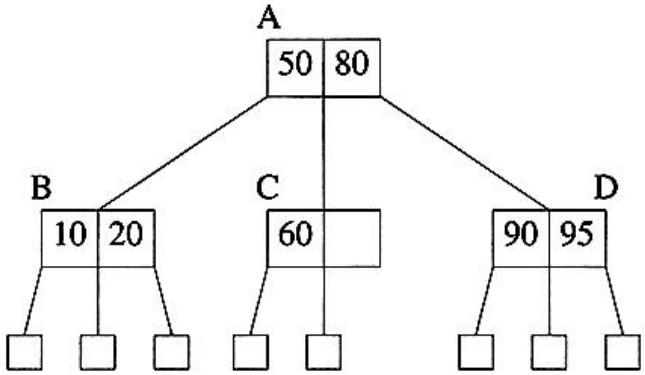


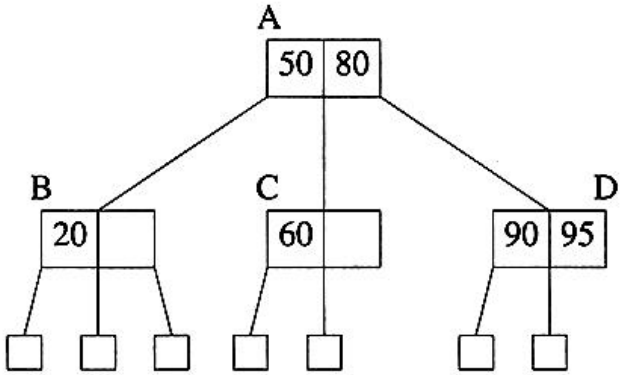
Figure 11.8: Combining in a 2-3 tree when p is the left child of r



(a) Initial 2-3 tree

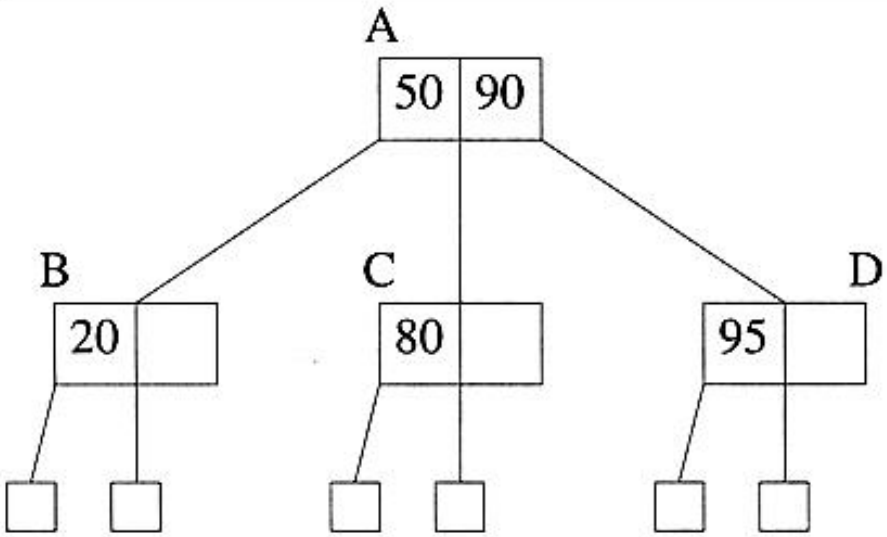


(b) 70 deleted

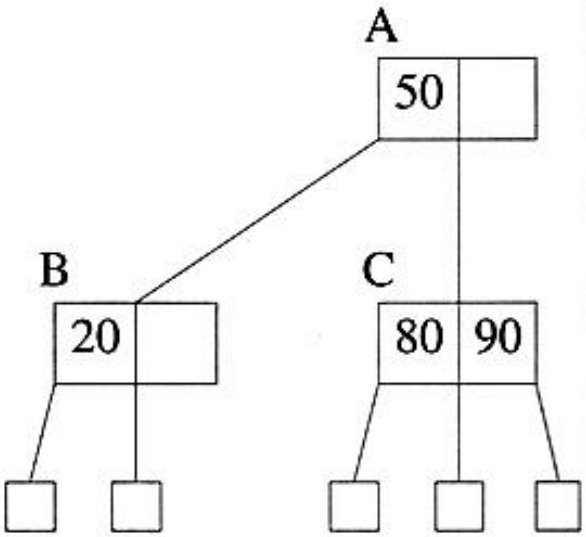


(c) 10 deleted

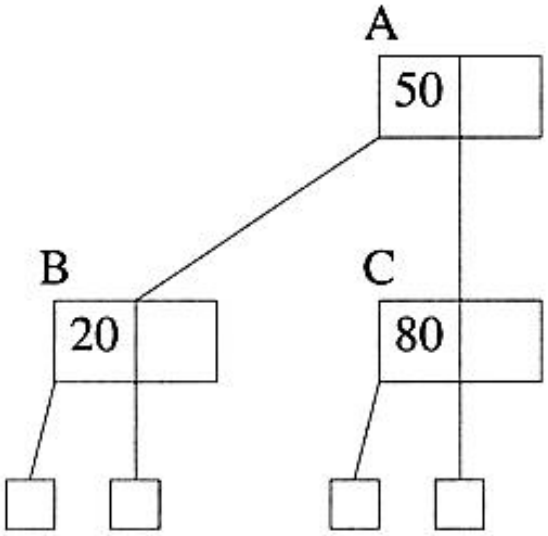
Figure 11.9: Deletion from a 2-3 tree (continued on next page)



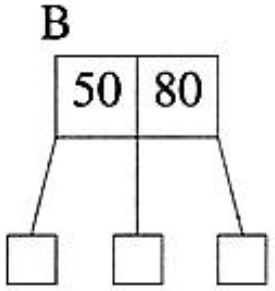
(d) 60 deleted



(e) 95 deleted



(f) 90 deleted



(g) 20 deleted

Figure 11.9: Deletion from a 2-3 tree