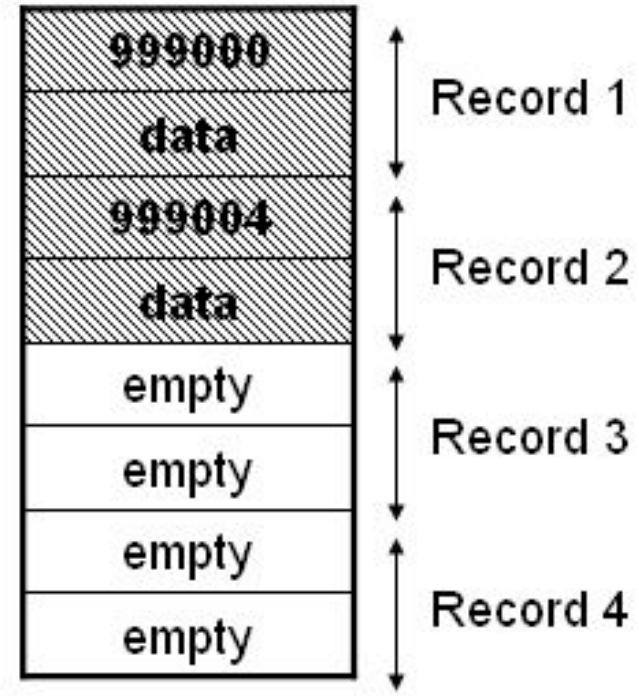
B-Trees

- Disk Storage
- What is a multiway tree?
- What is a B-tree?
- Why B-trees?
- Insertion in a B-tree
- Deletion in a B-tree

Disk Storage

- Data is stored on disk (i.e., secondary memory) in blocks.
- A block is the smallest amount of data that can be accessed on a disk.
- Each block has a fixed number of bytes – typically 512, 1024, 2048, 4096 or 8192 bytes
- Each block may hold many data records.

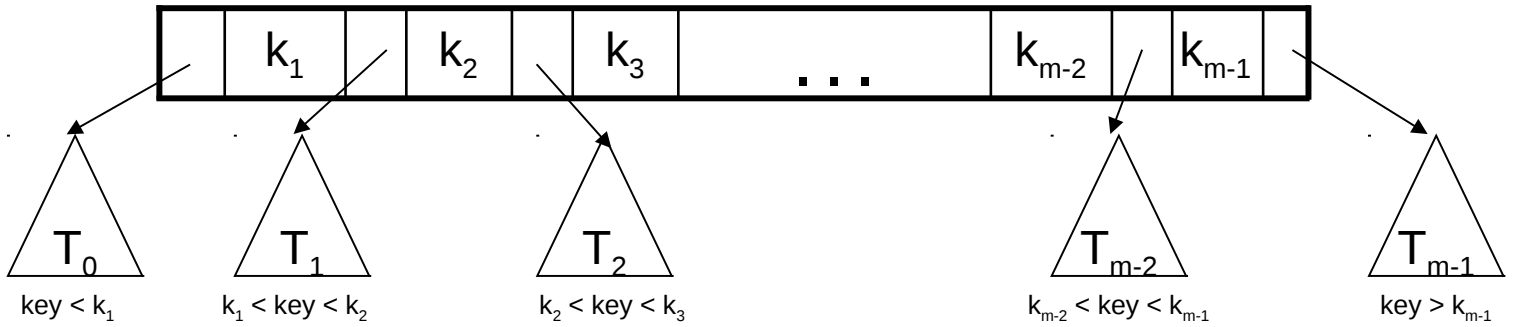


Motivation for studying Multi-way and B-trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.

What is a Multi-way tree?

- A multi-way (or m-way) search tree of **order m** is a tree in which
  - Each node has at-most **m** subtrees, where the subtrees **may be empty**.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.

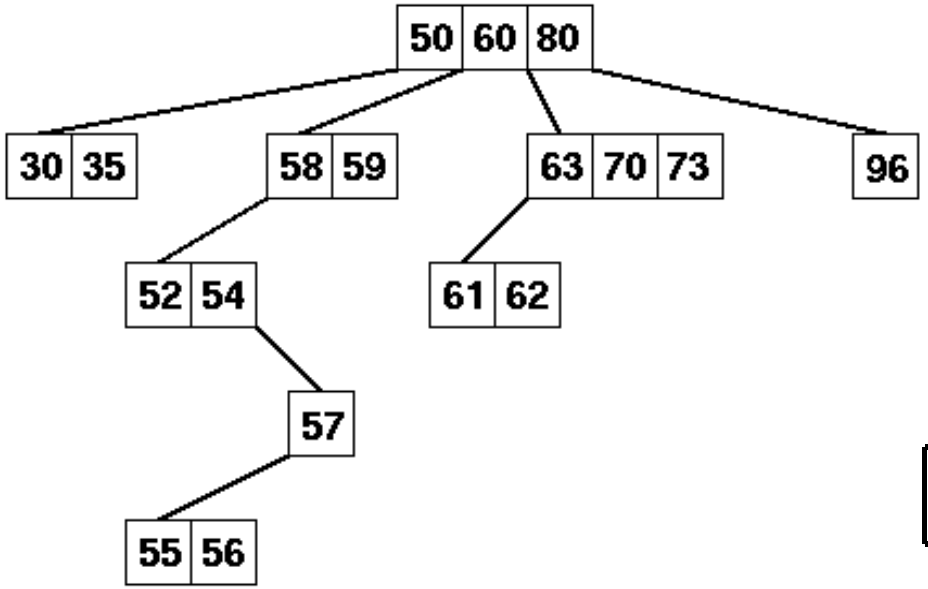


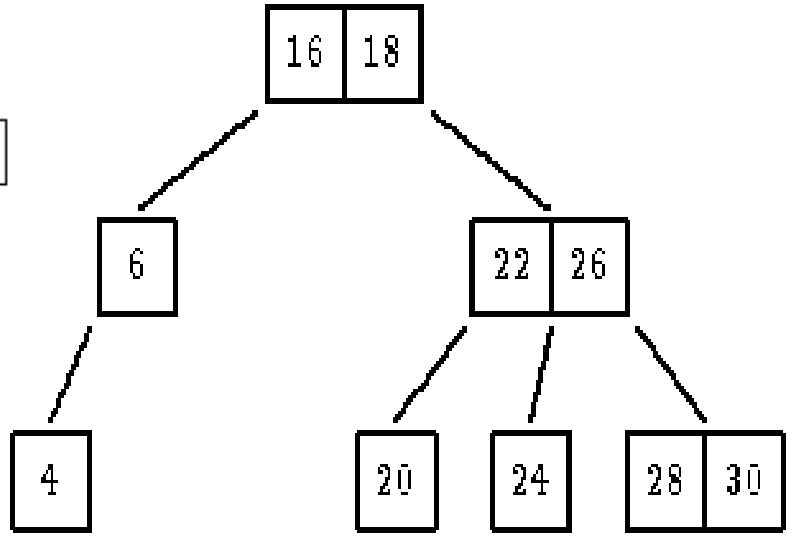
- The keys and subtrees of a non-leaf node are ordered as:

  $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:

  - All keys in subtree T0 are less than k1.
  - All keys in subtree T_i , 1 <= i <= m - 2, are greater than k_i but less than k_{i+1}.
  - All keys in subtree T_{m-1} are greater than k_{m-1}

Examples of Multi-way Trees





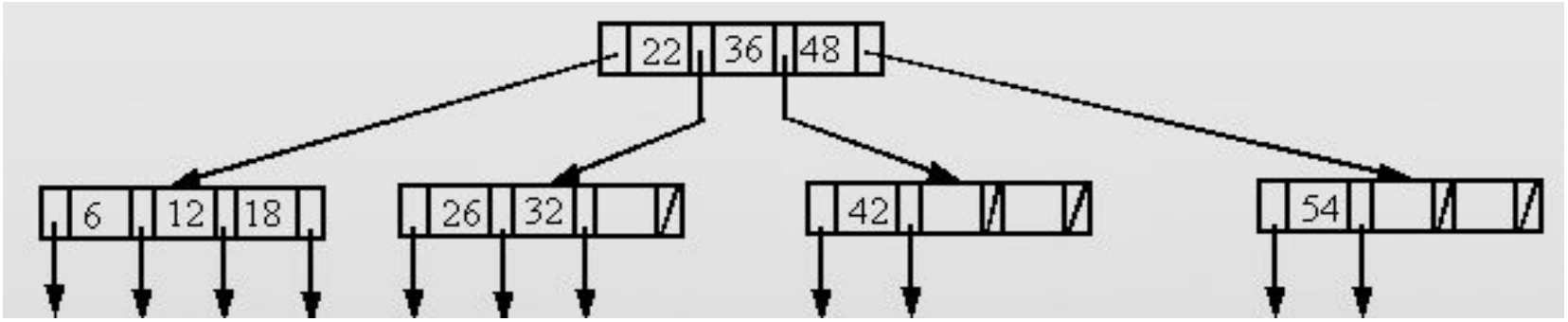
- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with **n** keys may contain less than **n + 1** non-empty subtrees.

What is a B-Tree?

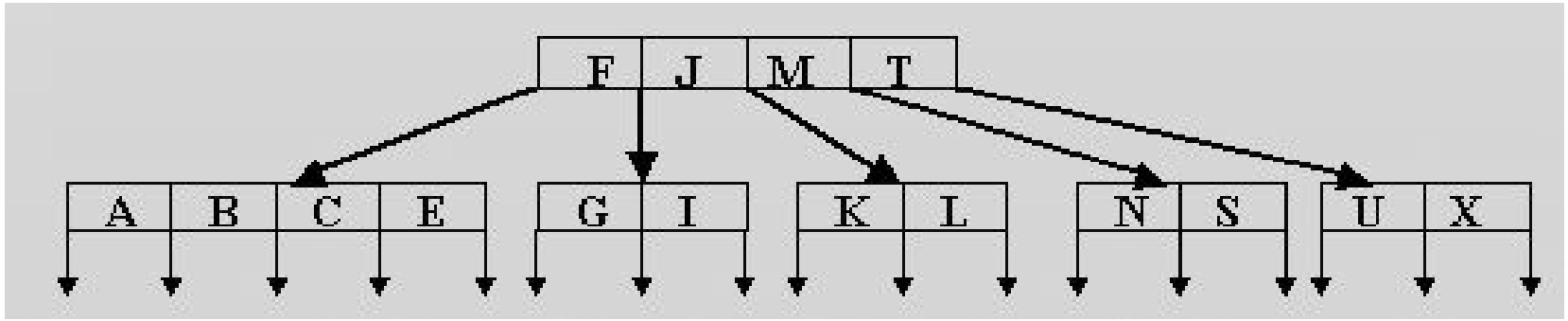
- A B-tree of order m (or branching factor m) is either an empty tree or a multiway search tree with the following properties:
  - Each non-root node contains **at least (m-1)/2 keys**
  - Each node in the tree has a **maximum of m-1 keys and m subtrees**

B-Tree Examples

Example: A B-tree of order 4



Example: A B-tree of order 5



More on Why B-Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:
  1. With a large branching factor m, the height of a B-tree is low resulting in fewer disk accesses.
     Note: As m increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.
  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.
  3. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and ***quickly*** finds all records with that property.

Insertion in B-Trees

- **OVERFLOW CONDITION:**

  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

- **Insertion algorithm:**

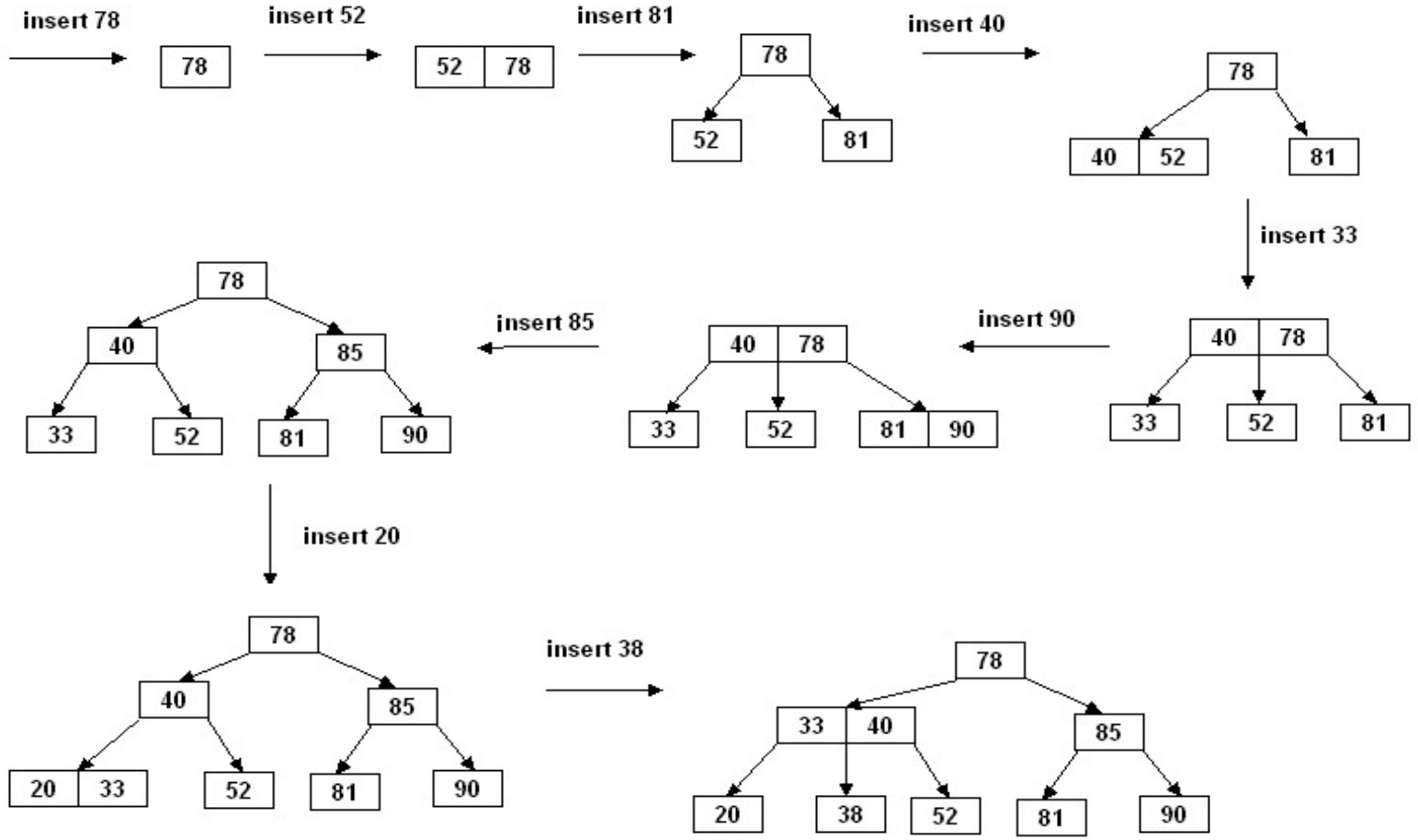
  If a node overflows, **split it into two**, **propagate the "middle" key to the parent of the node**. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- **Note: Insertion of a key always starts at a leaf node.**

Insertion in B-Trees

- **Insertion in a B-tree of odd order**

- Example: Insert the keys **78, 52, 81, 40, 33, 90, 85, 20,** and **38** in this order in an initially empty B-tree of **order 3**



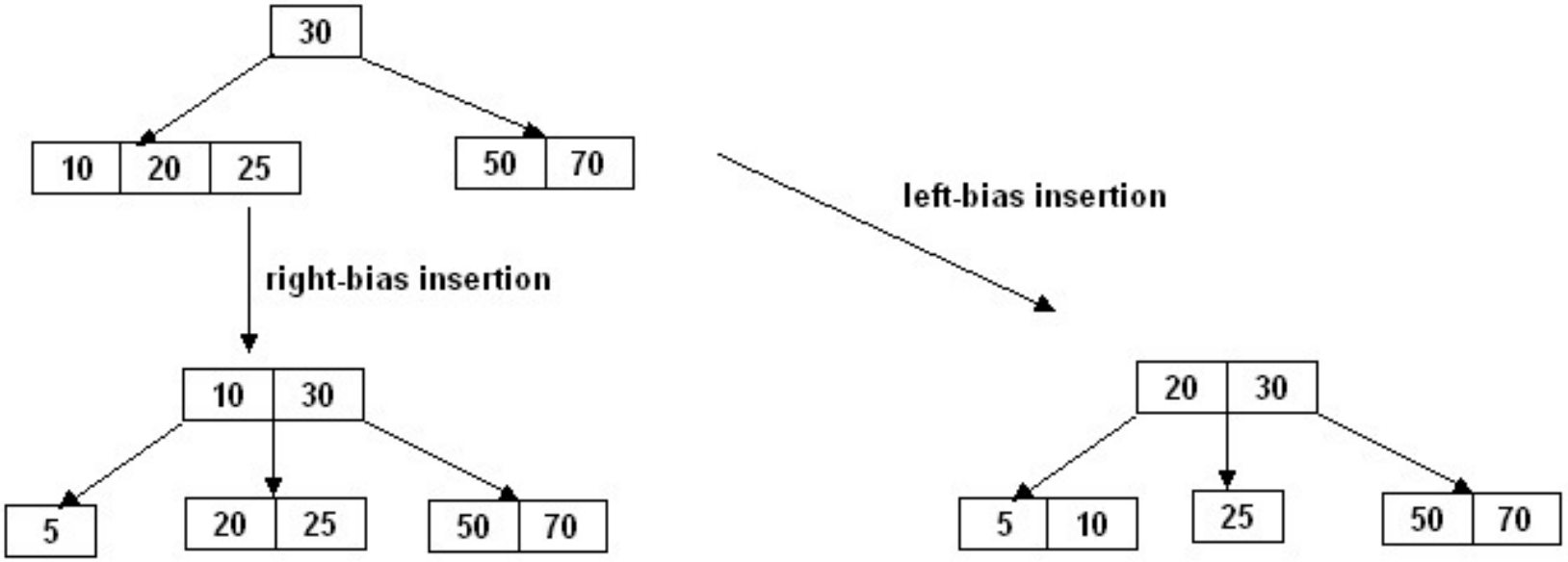
Insertion in B-Trees

Insertion in a B-tree of <u>even</u> order

At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

- **Example**: Insert the key **5** in the following B-tree of order **4**:



Exercise in Inserting a B-Tree

Insert the following keys to a 5-way B-tree:

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

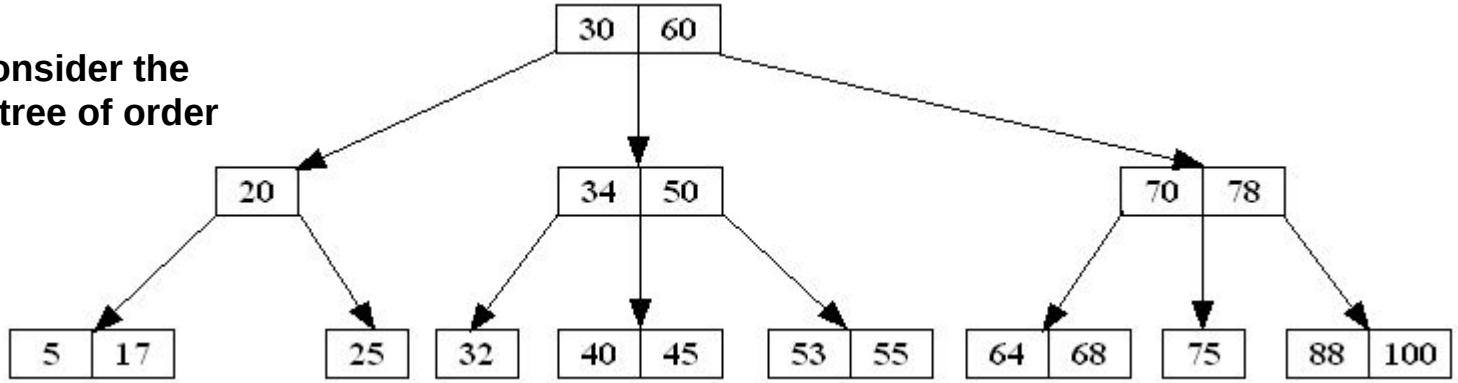
Deletion in B-Tree

- **Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).**

- The successor of a key **k** is the smallest key greater than **k**.
- The predecessor of a key **k** is the largest key smaller than **k**.

- IN A B-TREE THE SUCCESSOR AND PREDECESSOR, IF ANY, OF ANY KEY IS IN A LEAF NODE

Example: Consider the following B-tree of order 3:



key	predecessor	successor
20	17	25
30	25	32
34	32	40
50	45	53
60	55	64

B^+-Trees

- In B^+ Trees, all keys are maintained in leaves, and keys are replicated in nonleaf nodes to define paths for locating individual records.

- The leaves are linked together to provide a sequential path for traversing the keys in the tree

Removal from a B-tree

During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:

1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.

2 - If the key 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 the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:

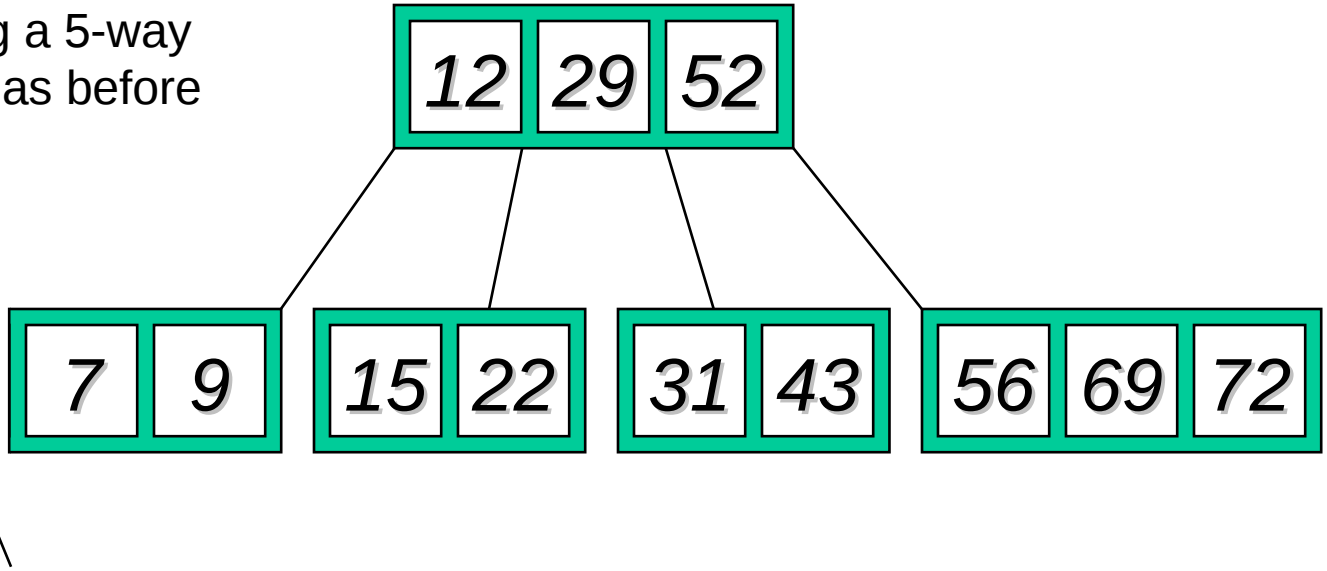
3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf

4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

Type #1: Simple leaf deletion

Type 1: If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.

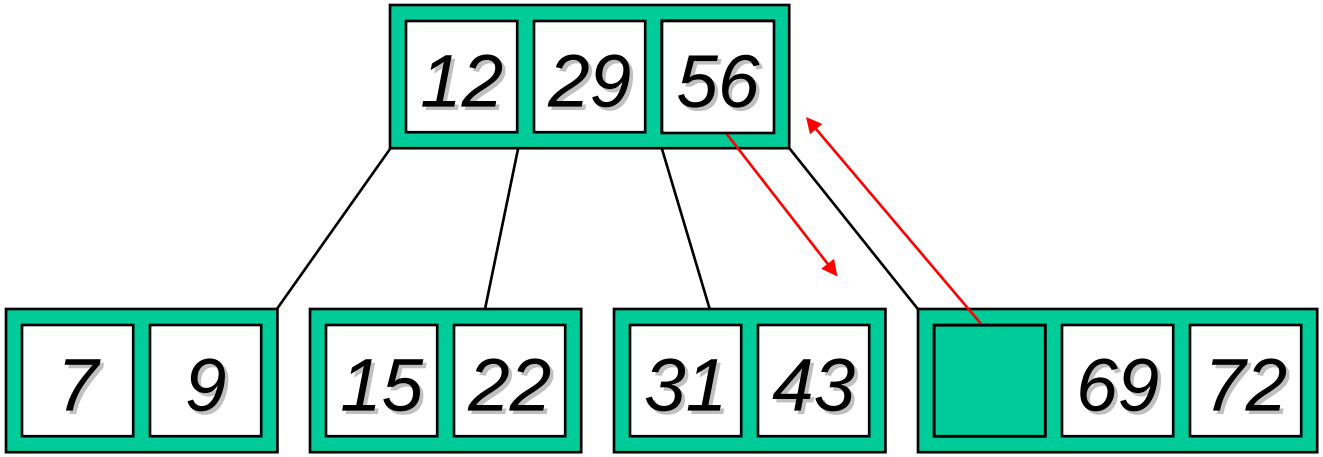
Assuming a 5-way ...B-Tree, as before



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

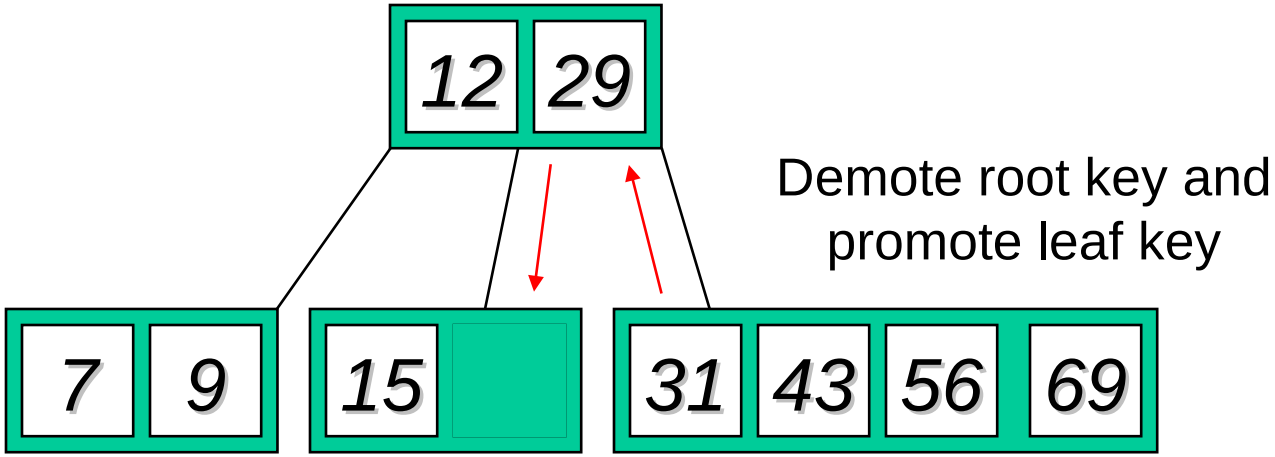
Type #2: Simple non-leaf deletion

Type 2: If the key 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 the key and promote the predecessor or successor key to the non-leaf deleted key's position.

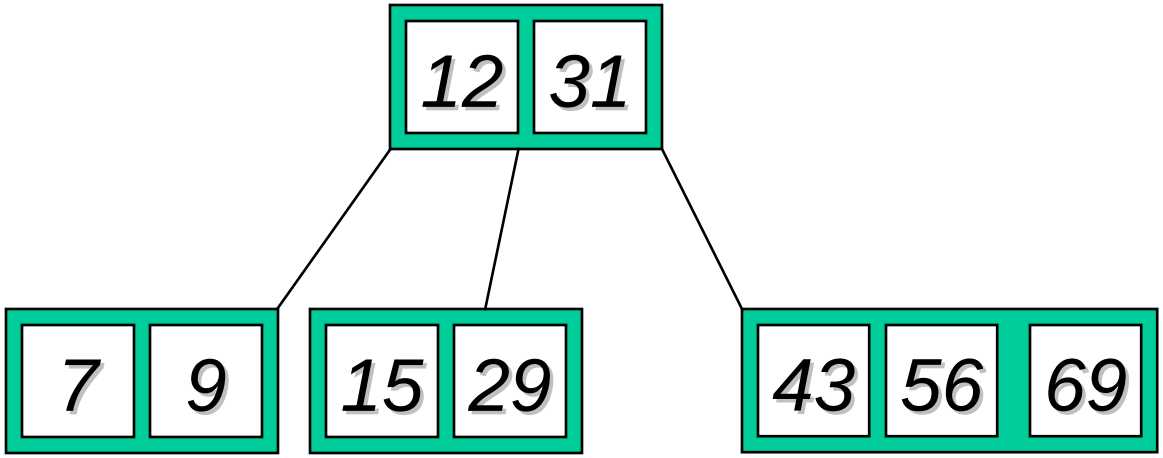


Type #3: Enough siblings

Type 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 our lacking leaf

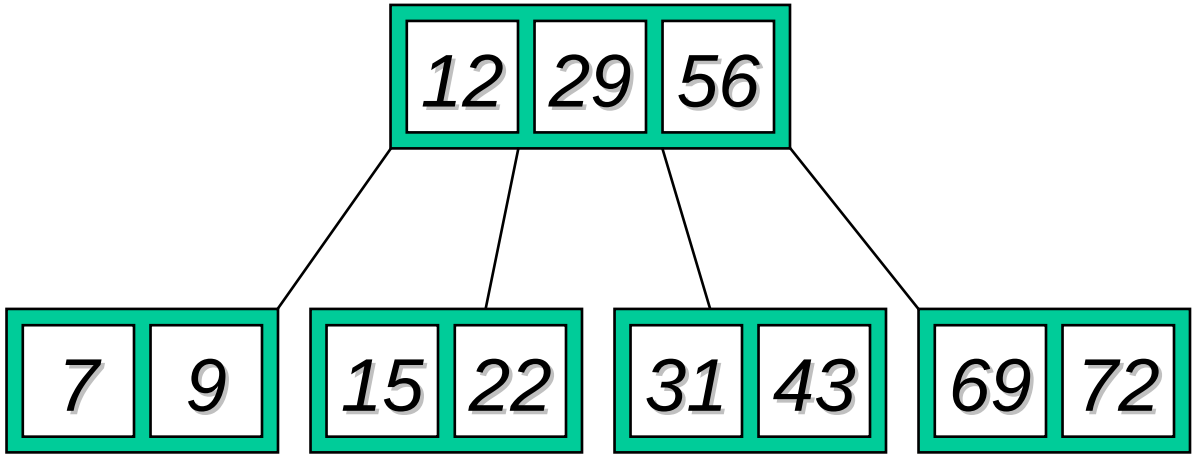


Type #3: Enough siblings

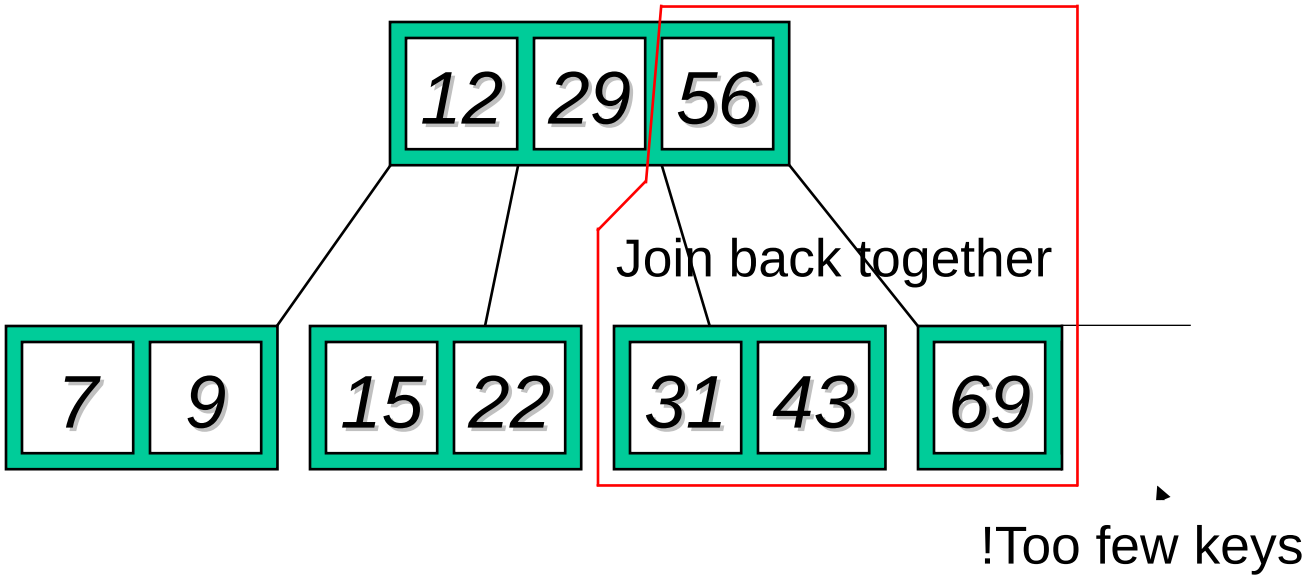


Type #4: Too few keys in node and its siblings

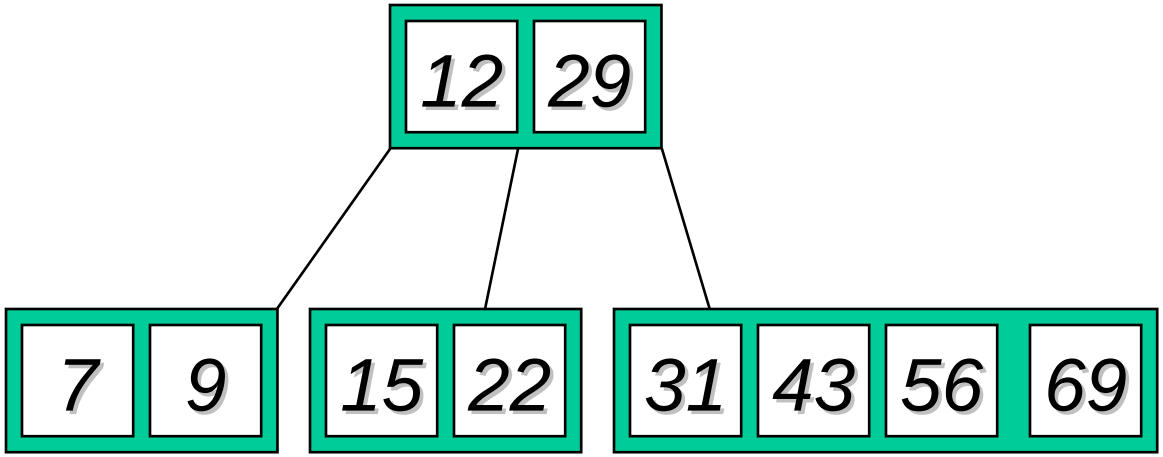
Type 4: If neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required



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



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



Exercise in Removal from a B-Tree

Given 5-way B-tree created by these data (last exercise):

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

Comparing Trees

Binary trees

Can become *unbalanced* and *lose* their good time complexity (big O)

AVL trees are strict binary trees that *overcome the balance problem*

Heap tree remain balanced but only *prioritise* (not order) the keys

Multi-way trees

B-Trees can be *m*-way, they can have any (odd) number of children

One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

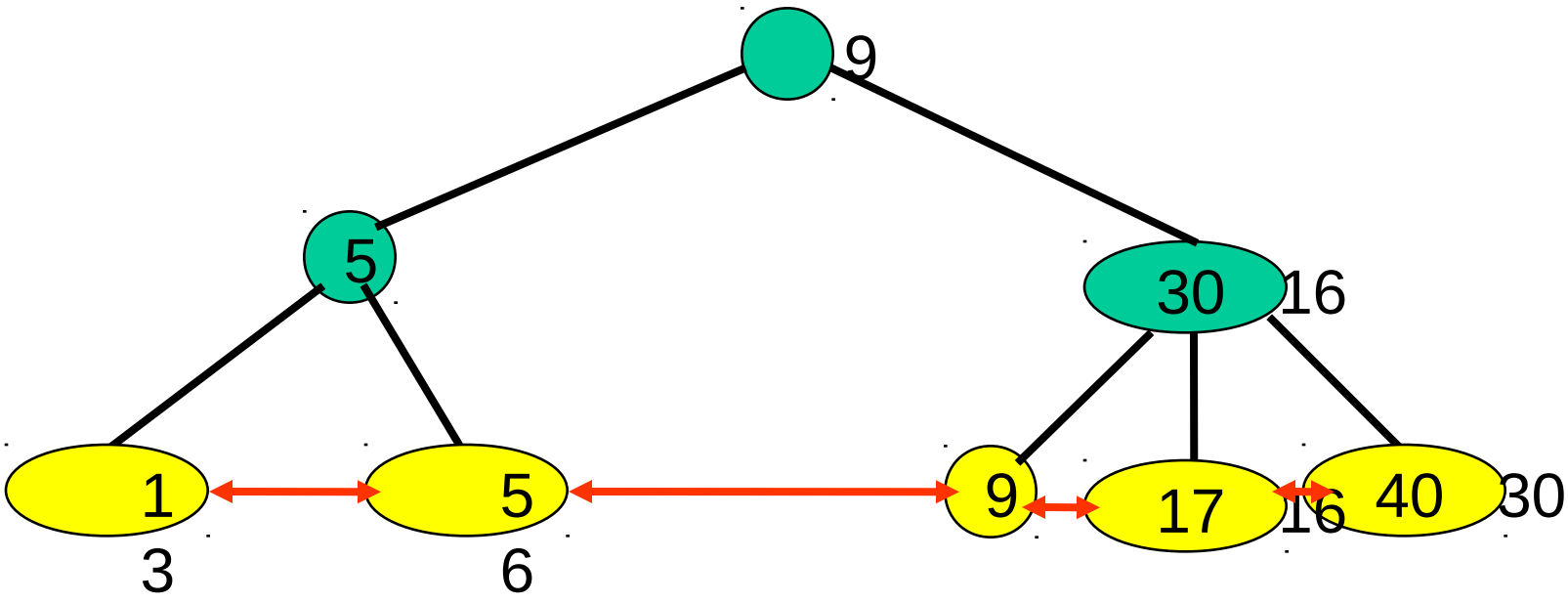
B^+-Trees

- Same structure as B-trees.
- Dictionary pairs are in leaves only. Leaves form a doubly-linked list.
- Remaining nodes have following structure:

$$j\ a_0\ k_1\ a_1\ k_2\ a_2\ \ldots\ k_j\ a_j$$

- j = number of keys in node.
- a_i is a pointer to a subtree.
- k_i <= smallest key in subtree a_i and largest in a_{i-1}.

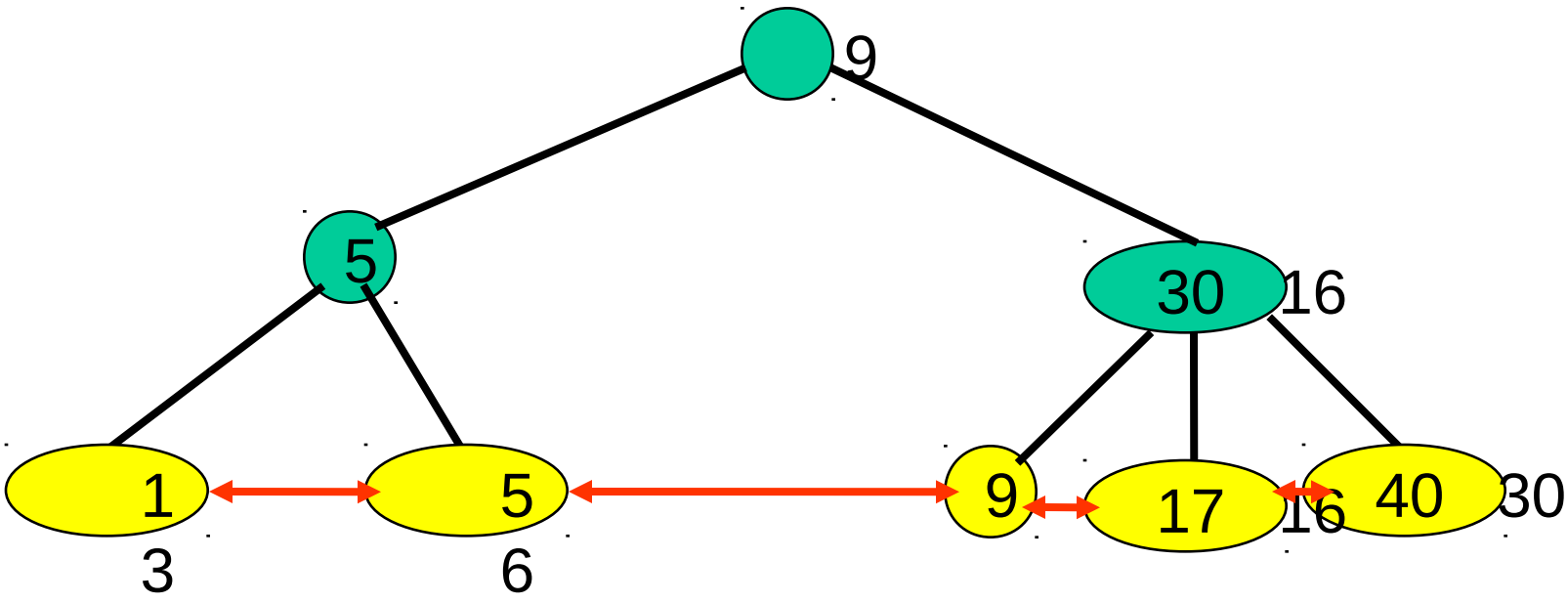
Example B+-tree



➔ index node

➔ leaf/data node

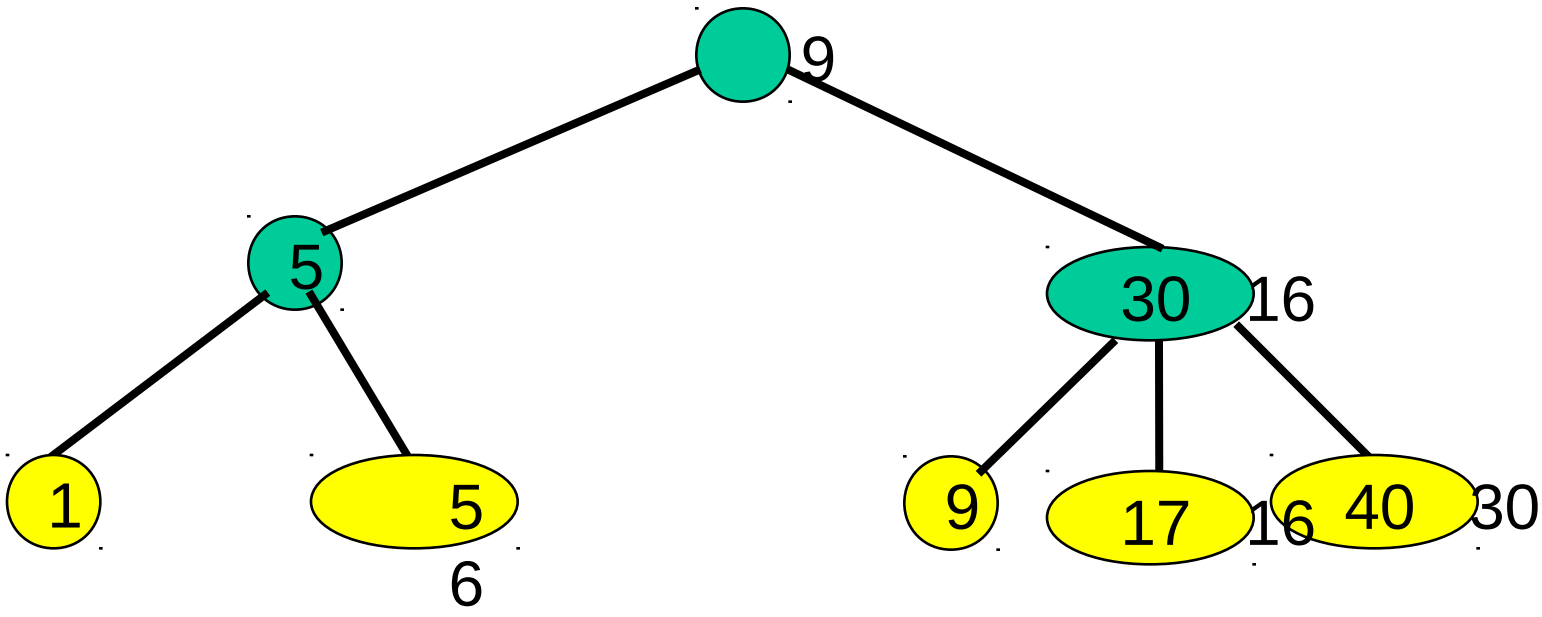
B+-tree—Search



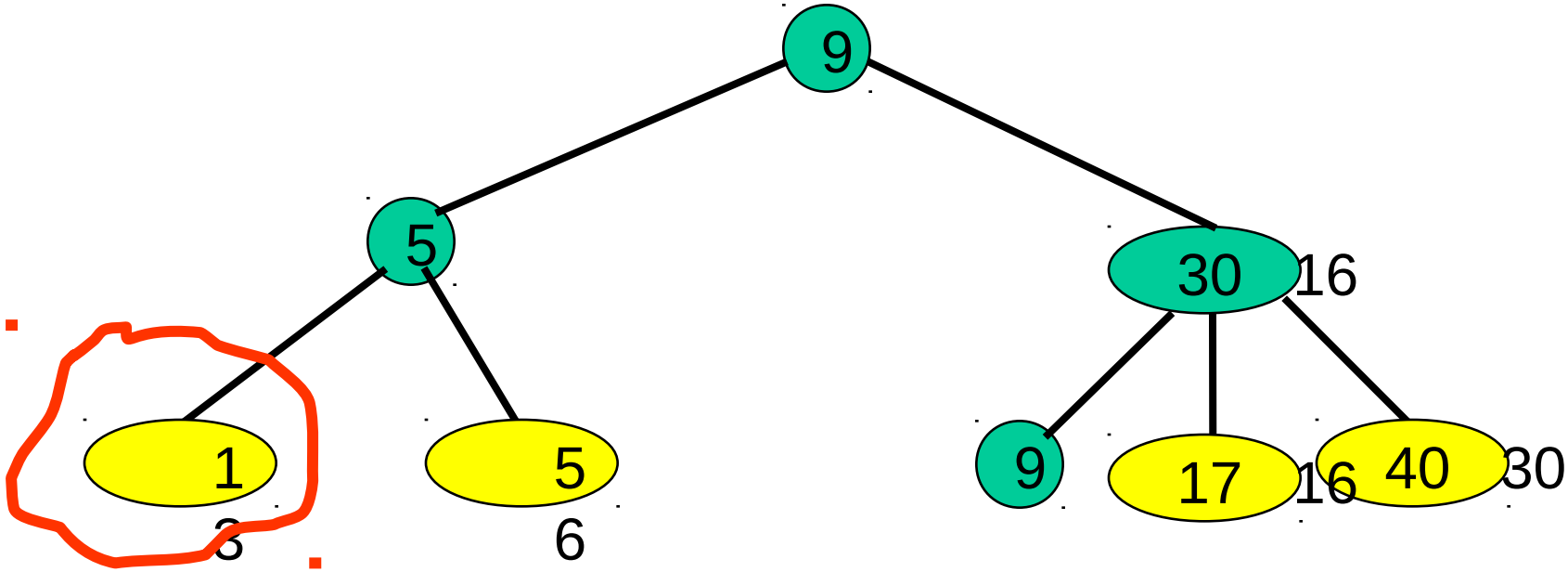
key = 5

key <= 20 <=6

B+-tree—Insert



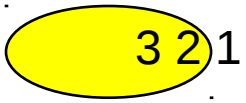
Insert



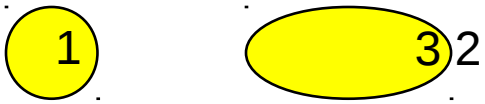
- Insert a pair with key = 2.
- New pair goes into a 3-node.

Insert Into A 3-node

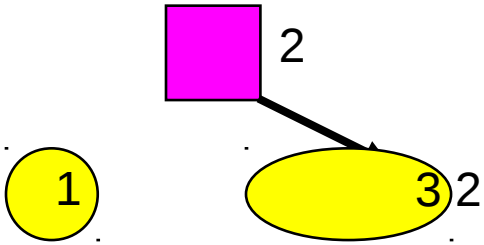
- Insert new pair so that the keys are in ascending order.



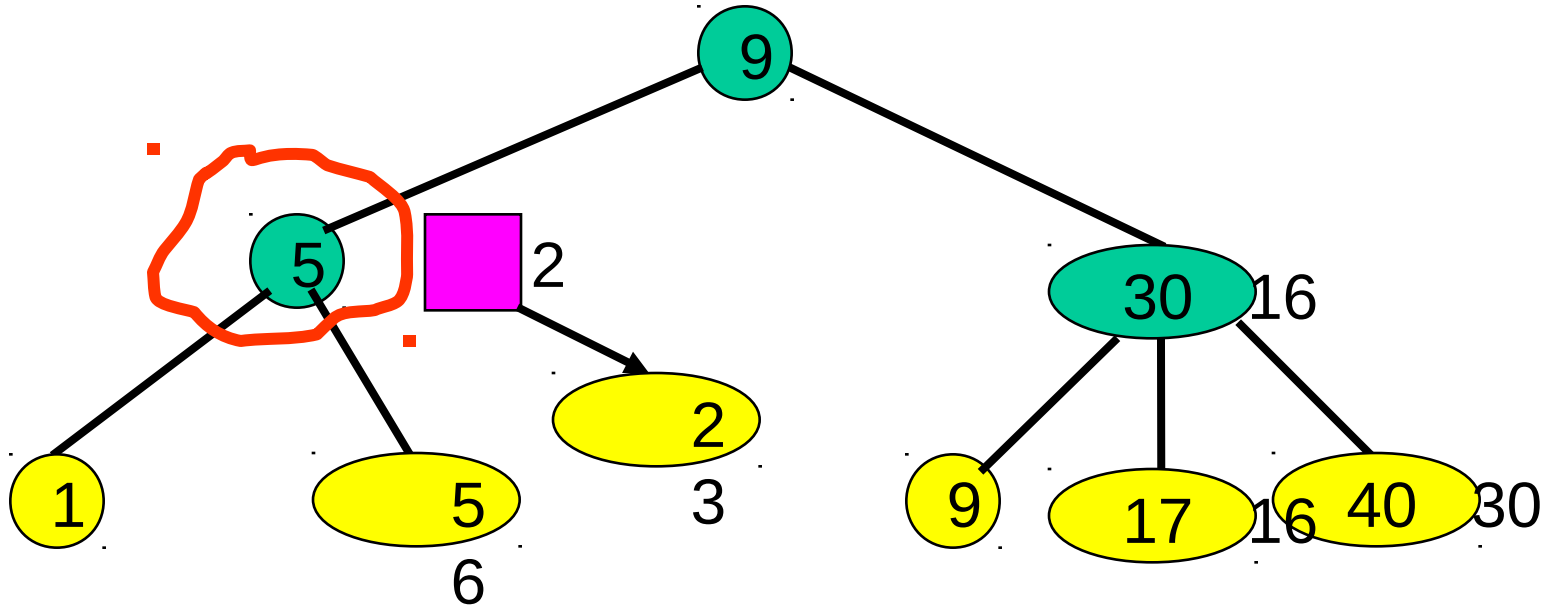
- .Split into two nodes



- Insert smallest key in new node and .pointer to this new node into parent

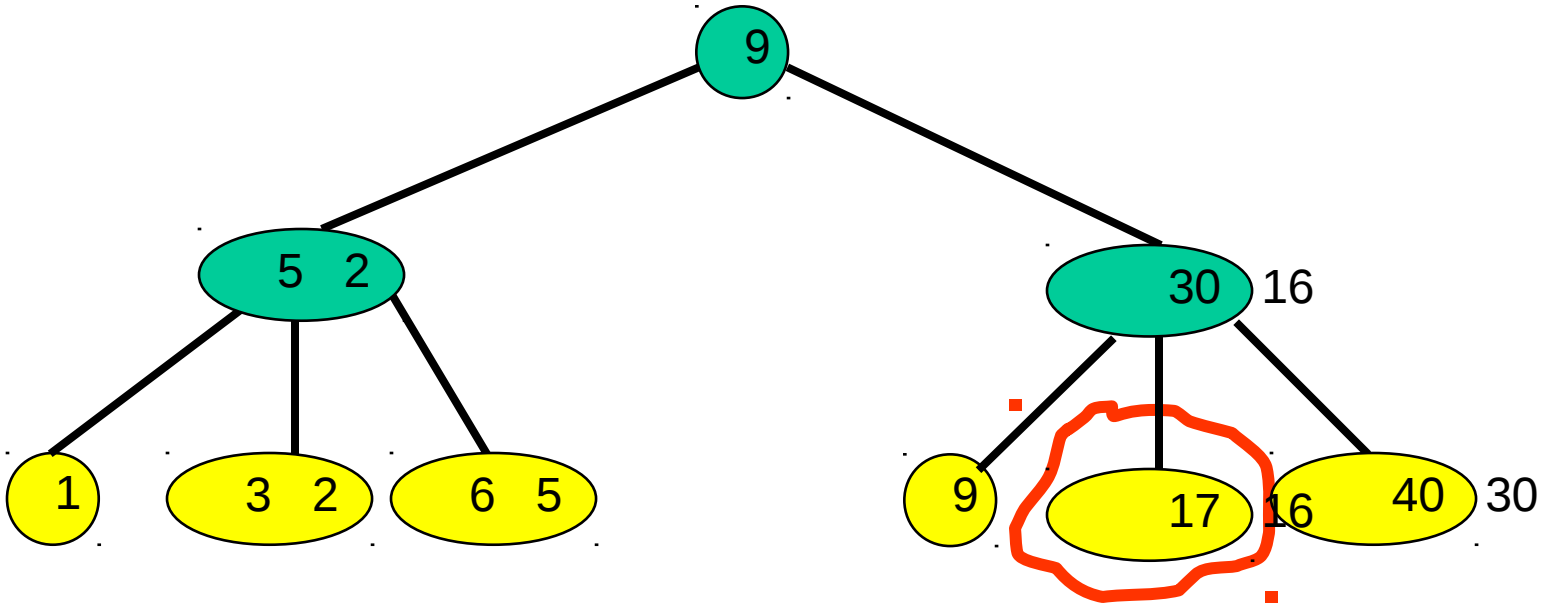


Insert



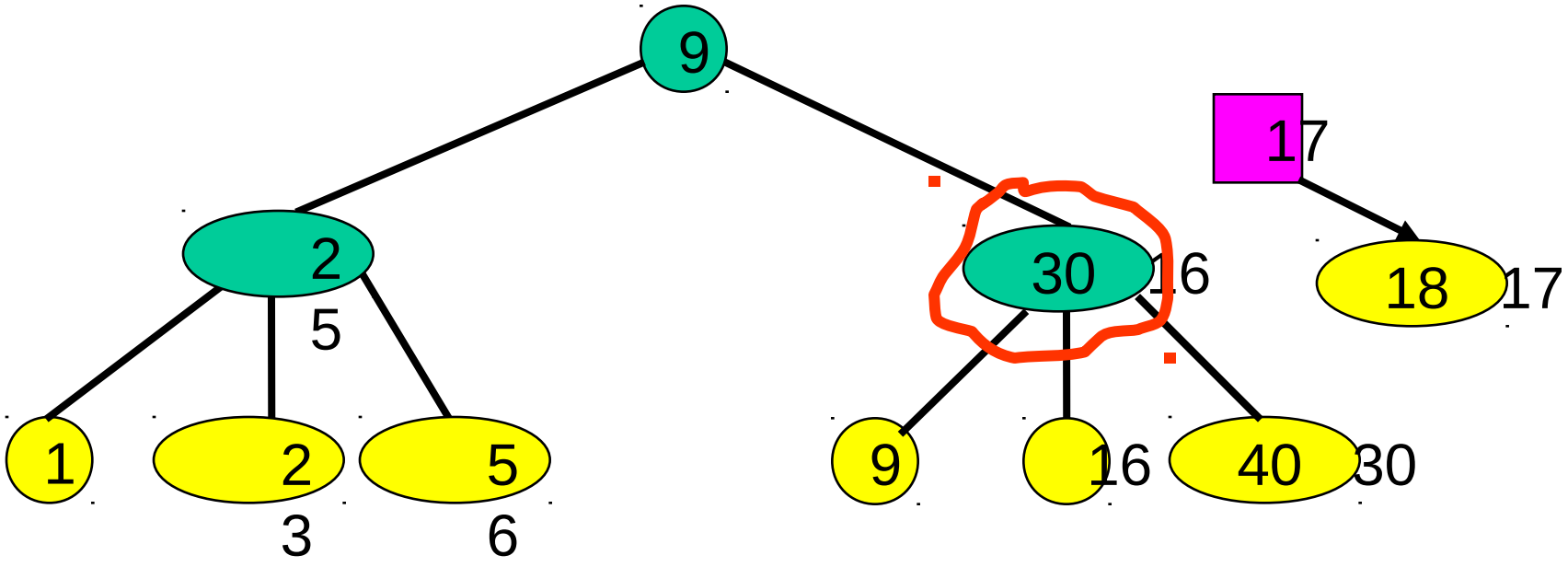
.Insert an index entry 2 plus a pointer into parent •

Insert



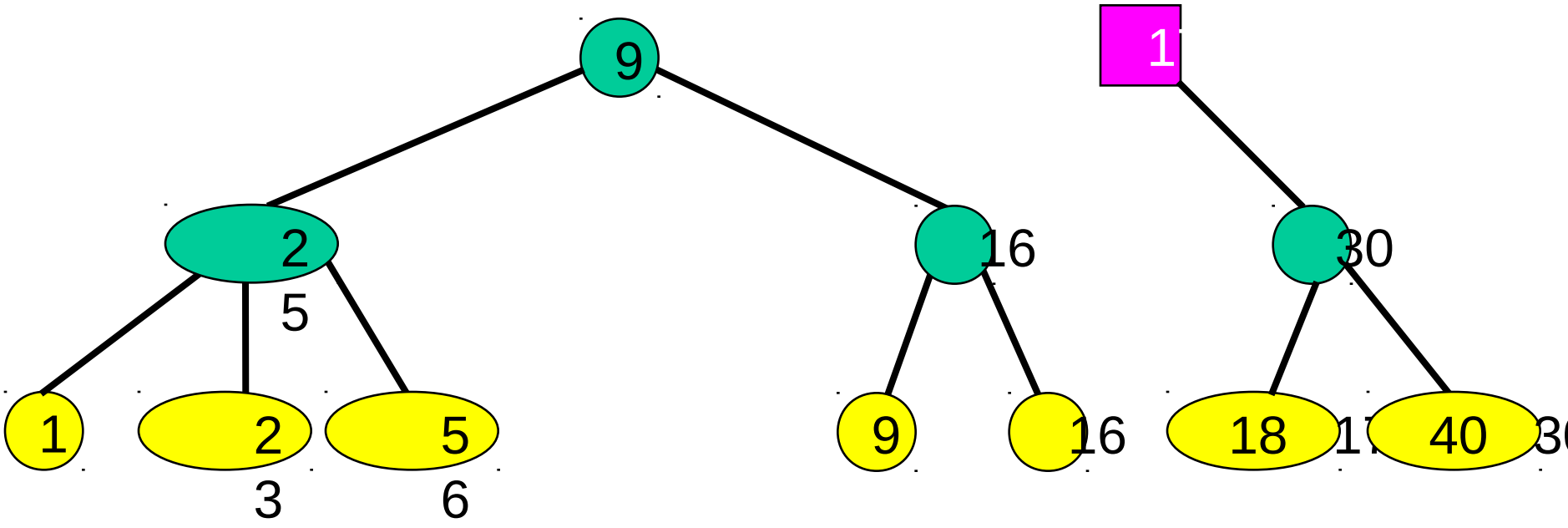
.Now, insert a pair with key = 18 •

Insert



- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

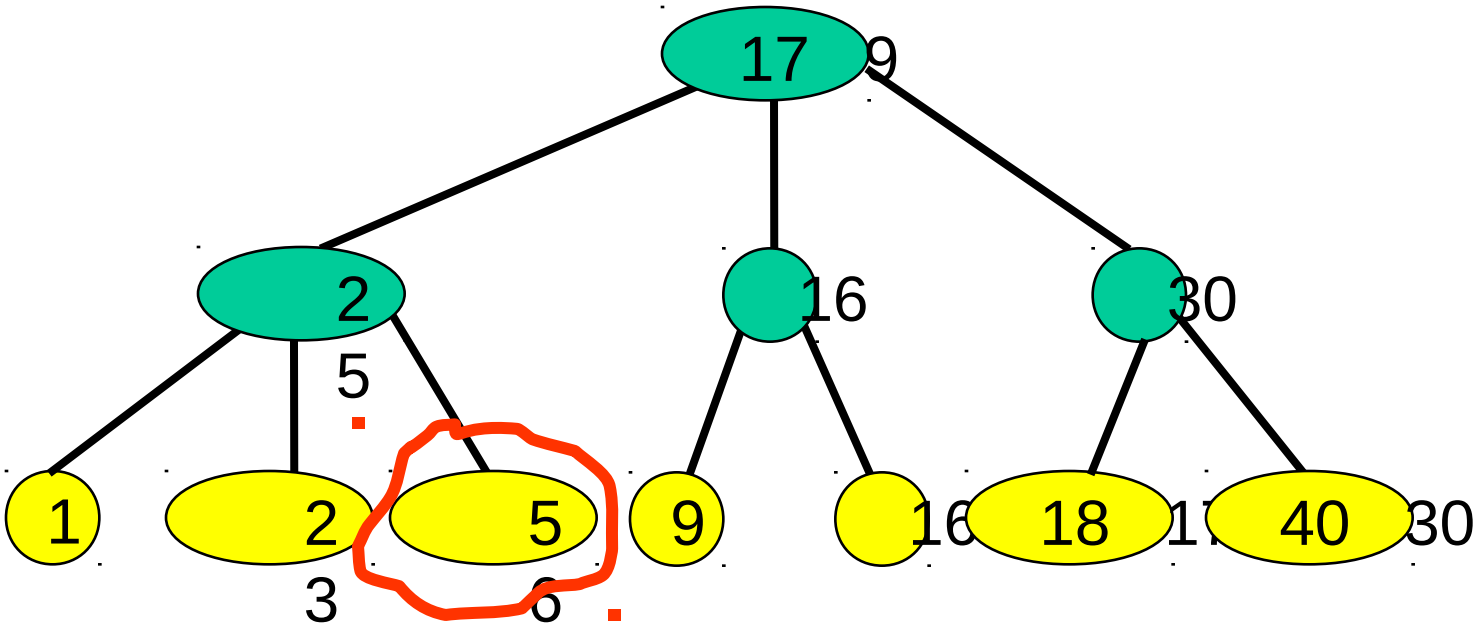
Insert



- Now, insert a pair with key = 18.

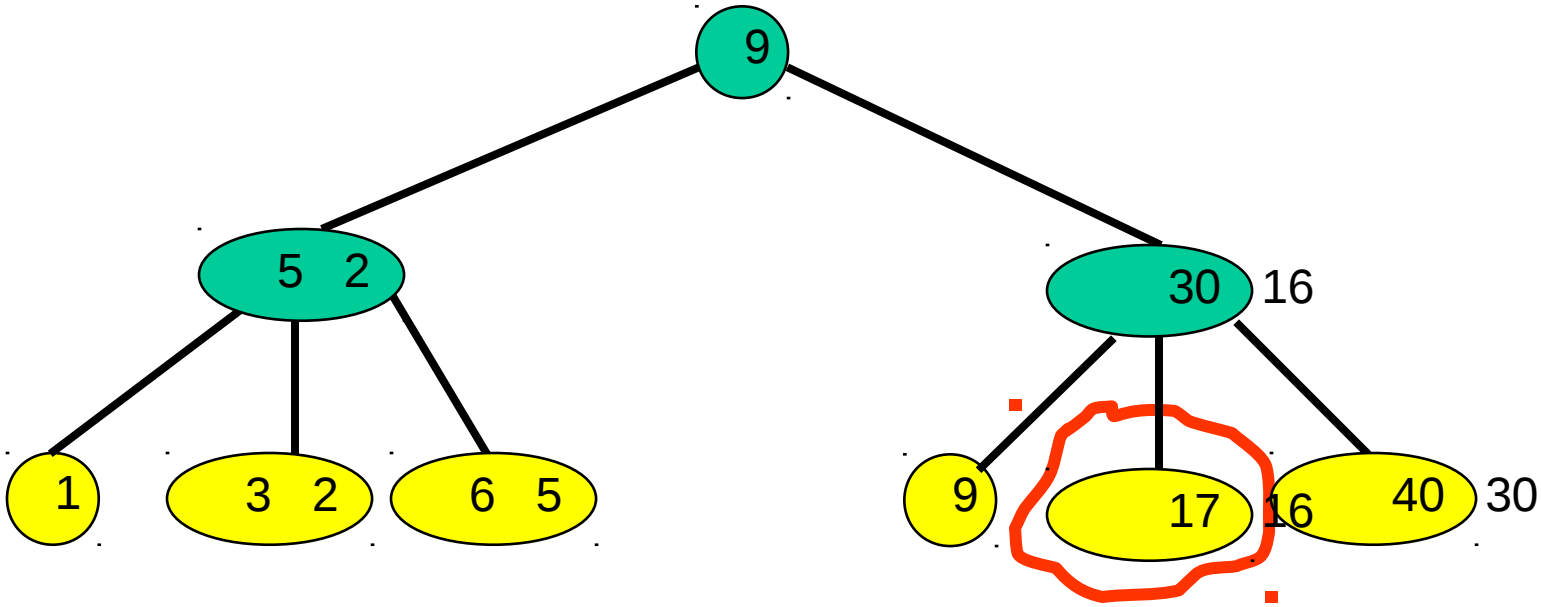
- Insert an index entry 17 plus a pointer into parent.

Insert



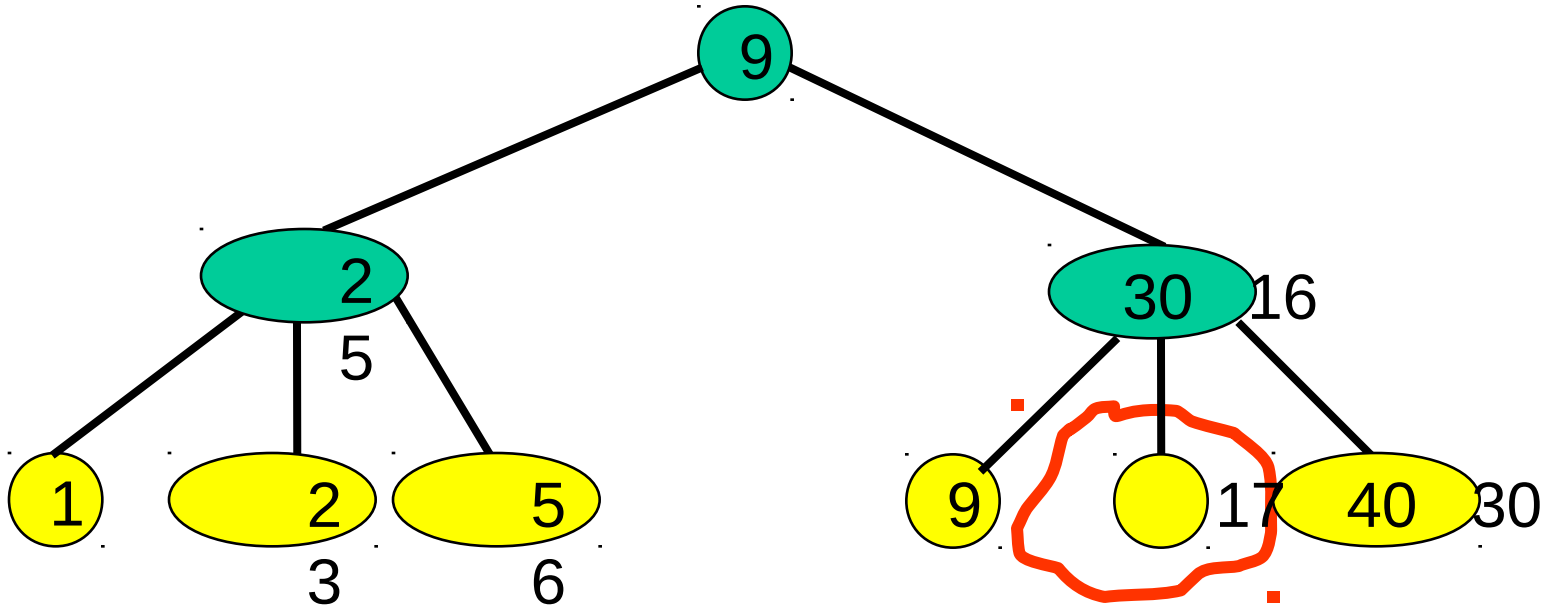
.Now, insert a pair with key = 7 •

Delete



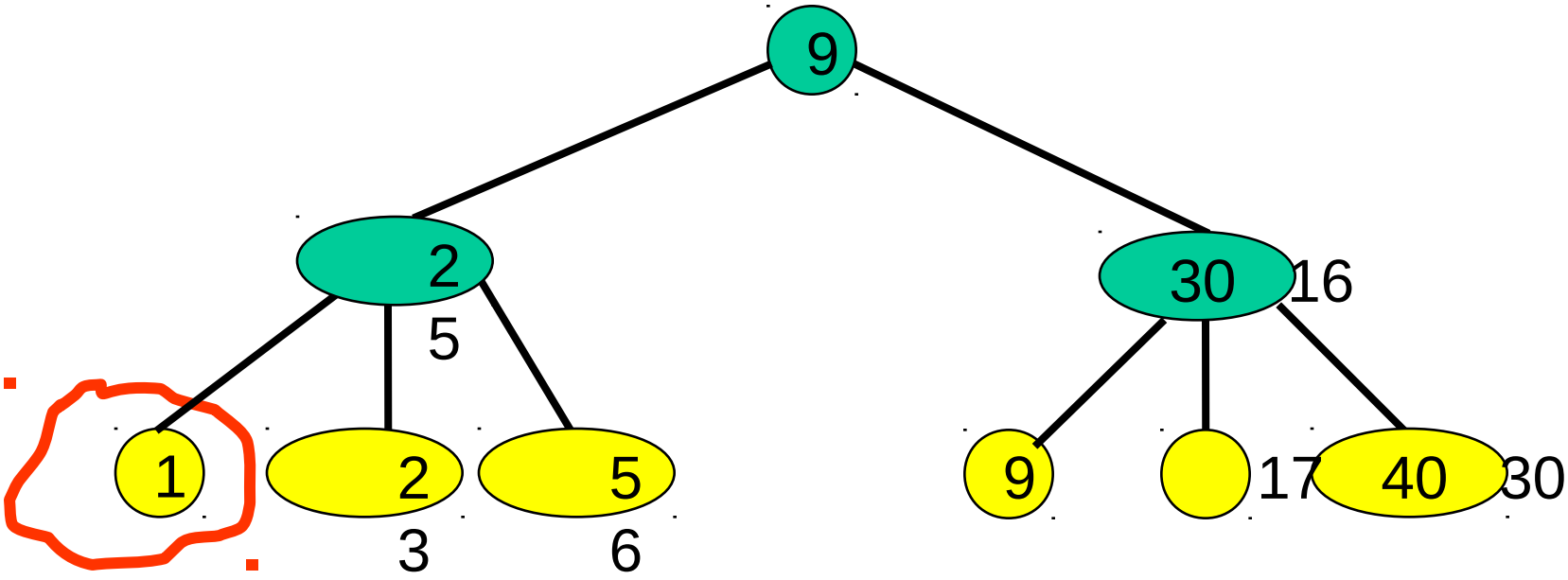
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

Delete



- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

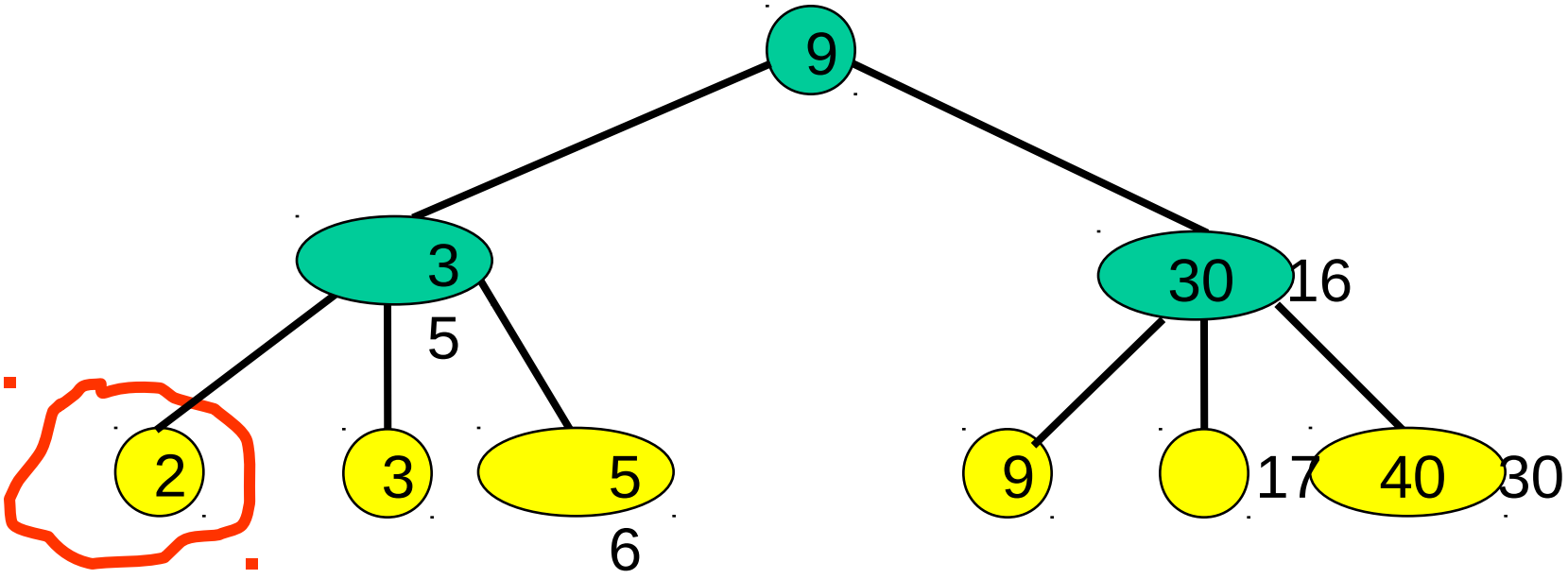
Delete



- Delete pair with key = 1.

- Get >= 1 from sibling and update parent key.

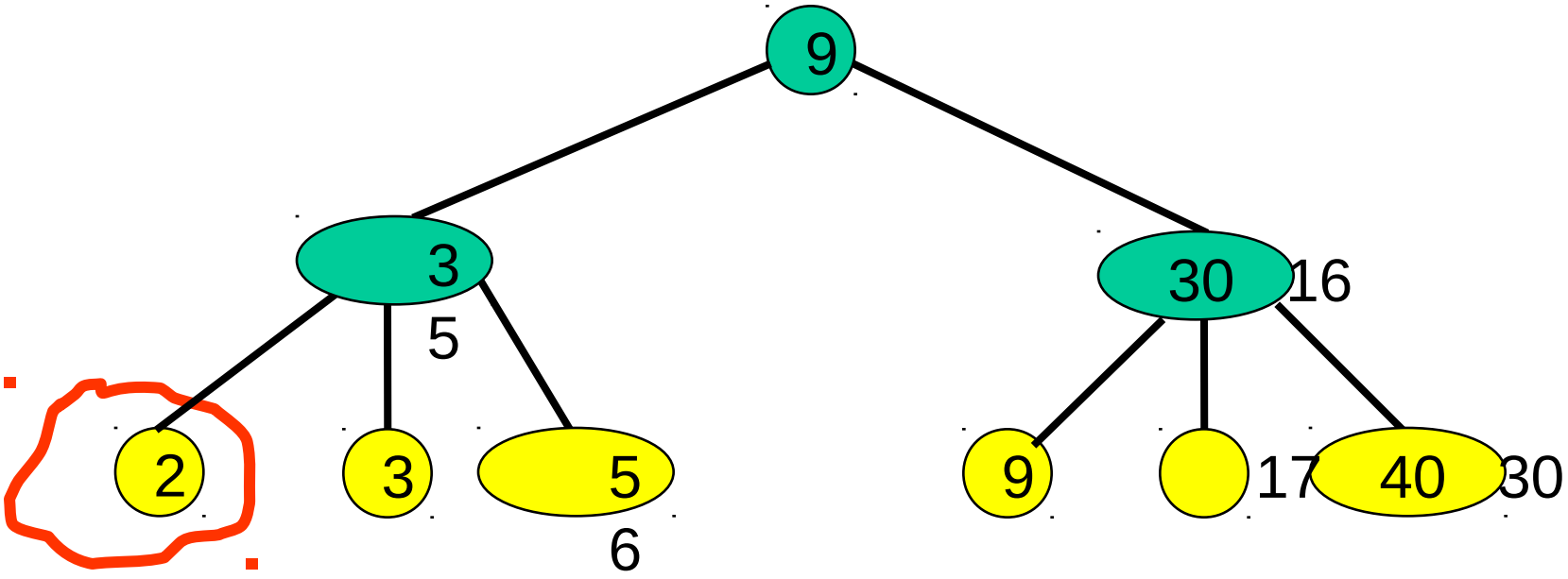
Delete



- Delete pair with key = 1.

- Get >= 1 from sibling and update parent key.

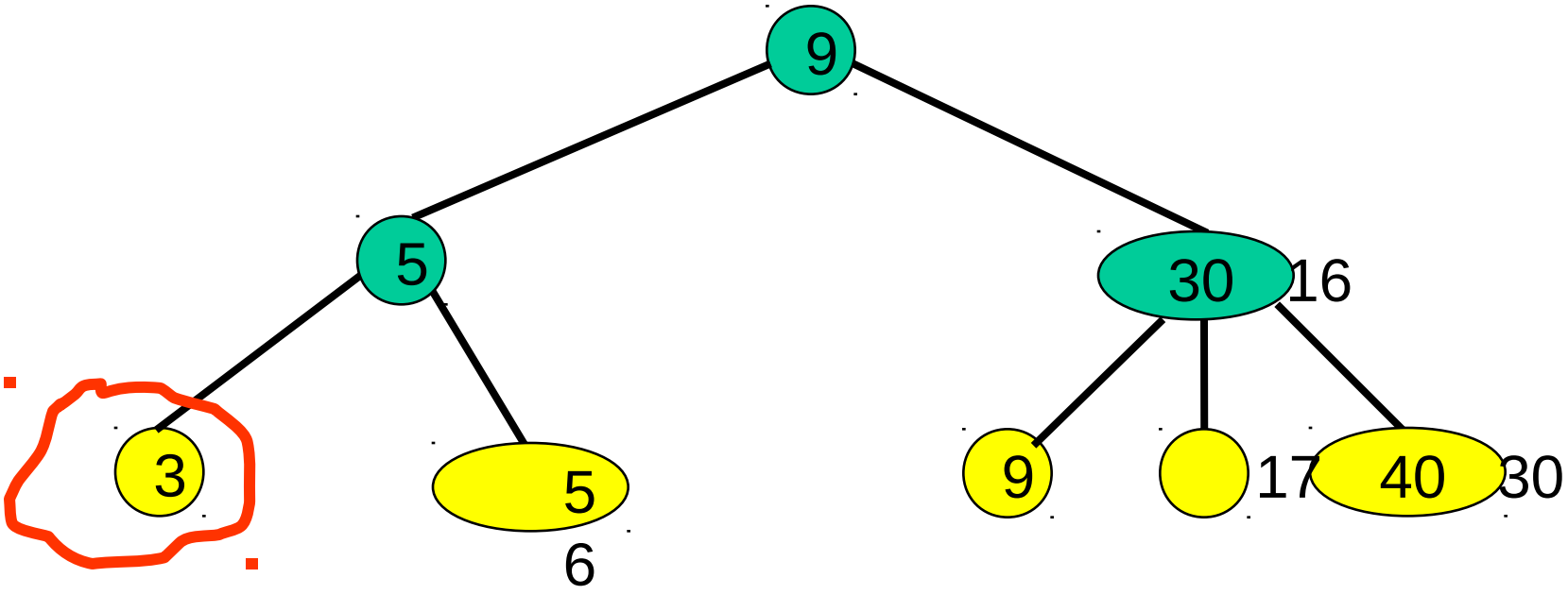
Delete



- Delete pair with key = 2.

- Merge with sibling, delete in-between key in parent.

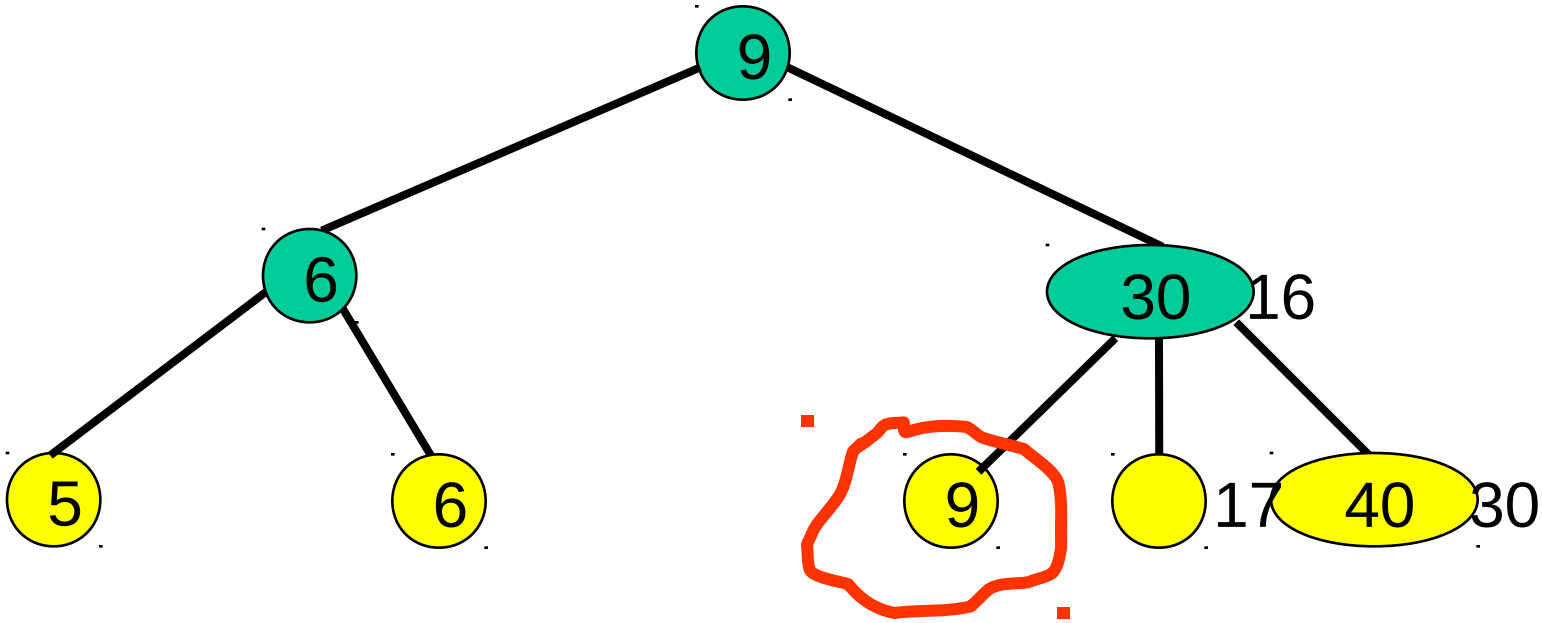
Delete



.Delete pair with key = 3 •

.Get >= 1 from sibling and update parent key•

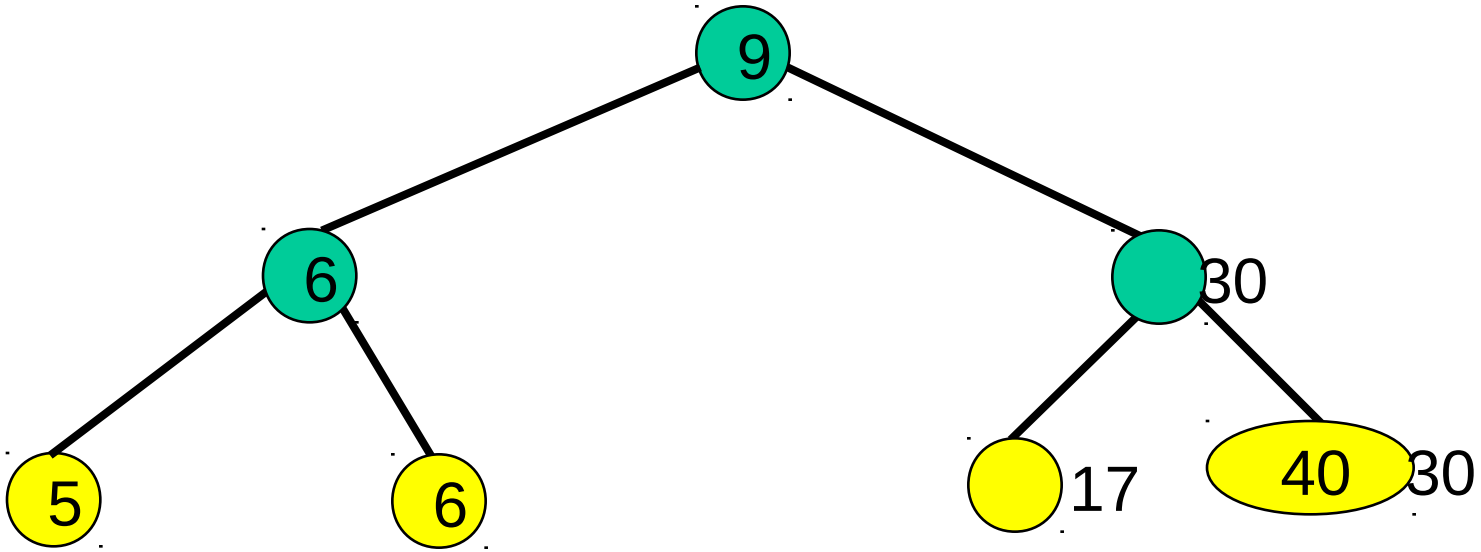
Delete



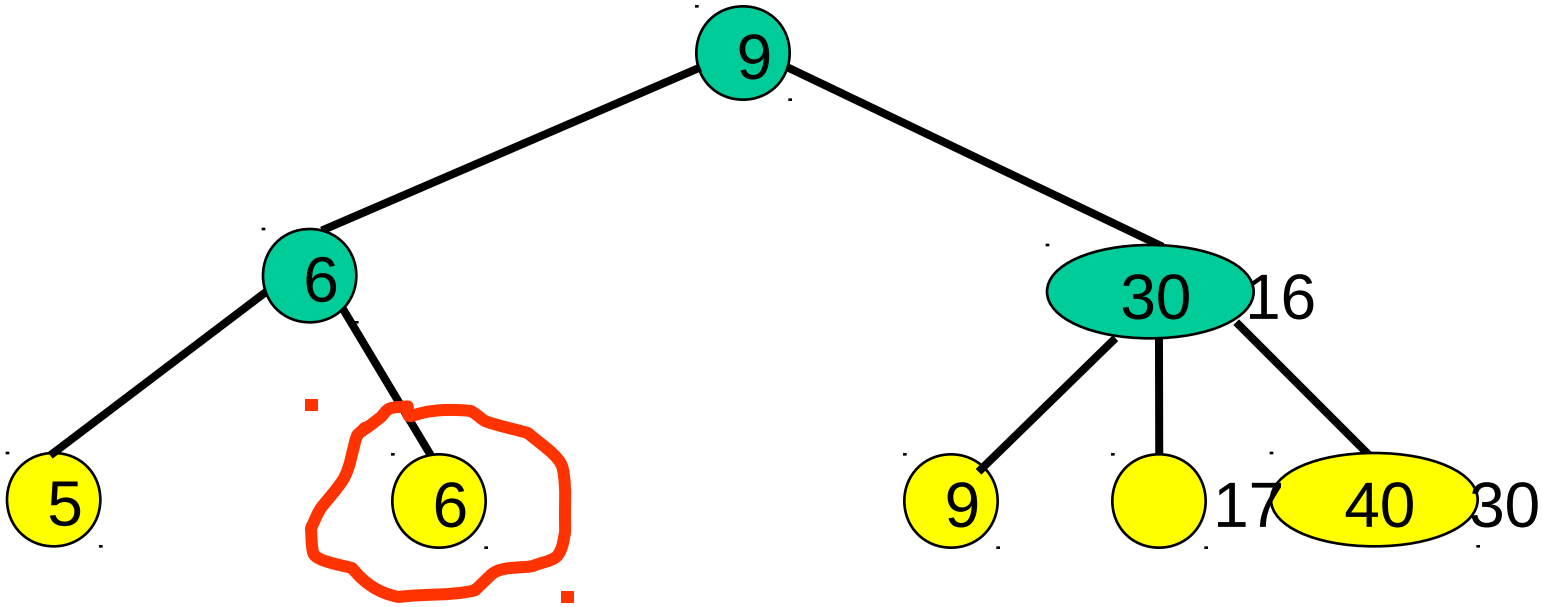
- Delete pair with key = 9.

- Merge with sibling, delete in-between key in parent.

Delete



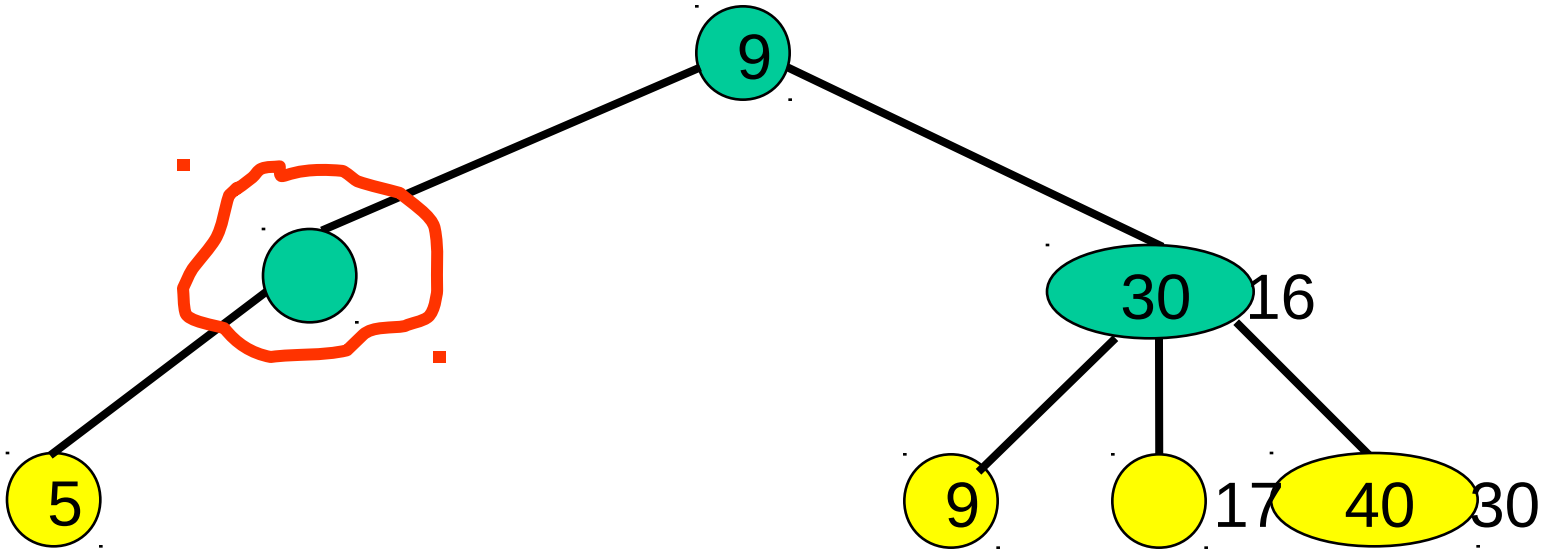
Delete



- Delete pair with key = 6.

- Merge with sibling, delete in-between key in parent.

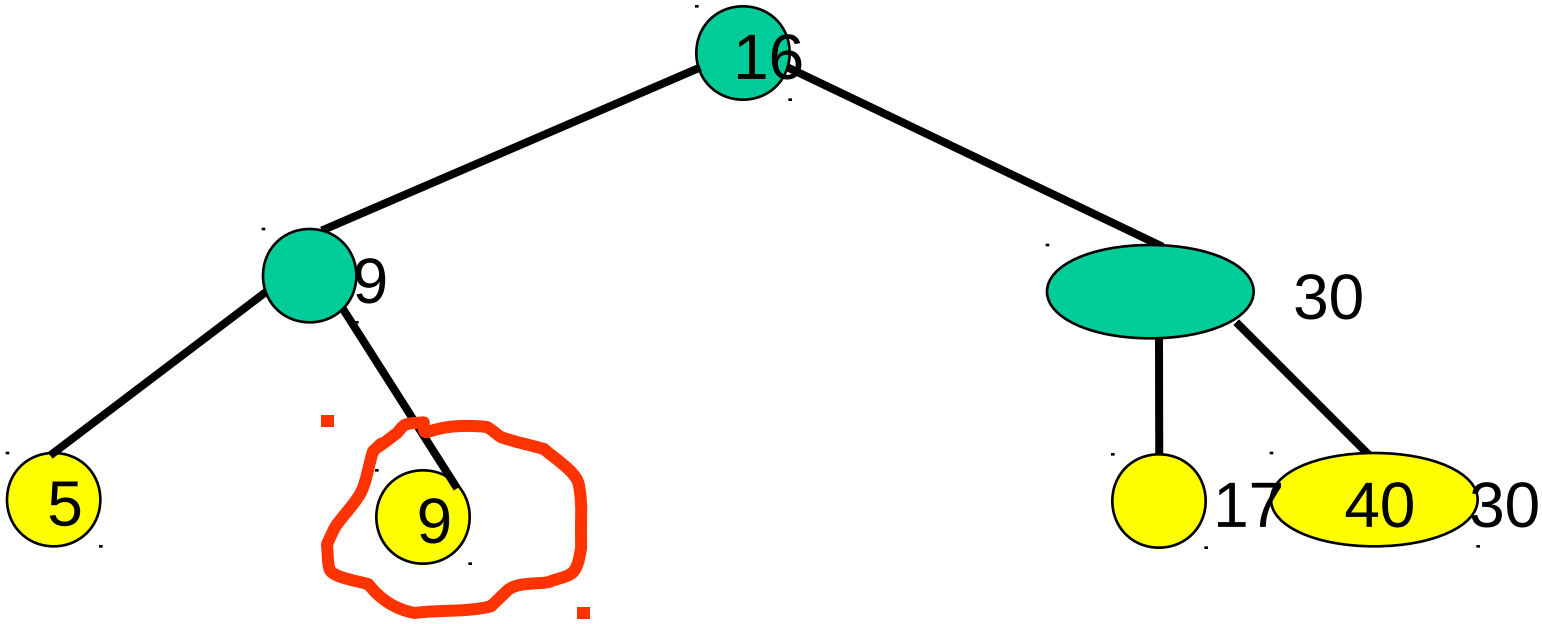
Delete



- Index node becomes deficient.

- Get >= 1 from sibling, move last one to parent,
get parent key.

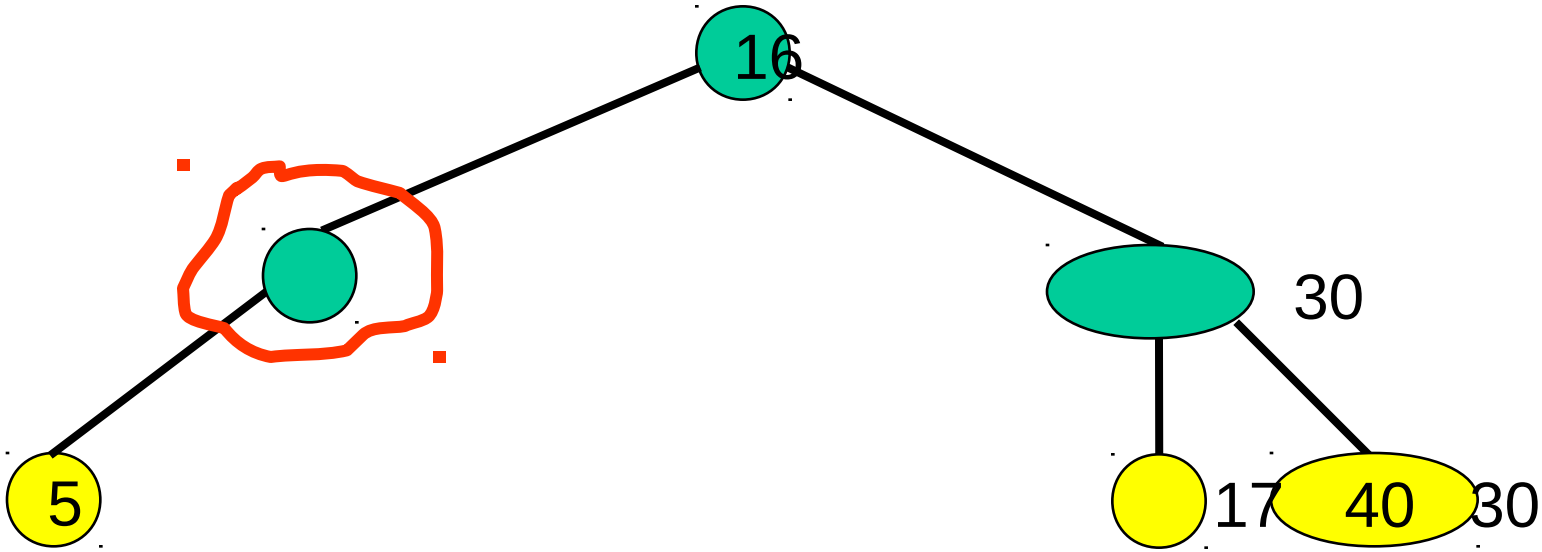
Delete



.Delete 9 •

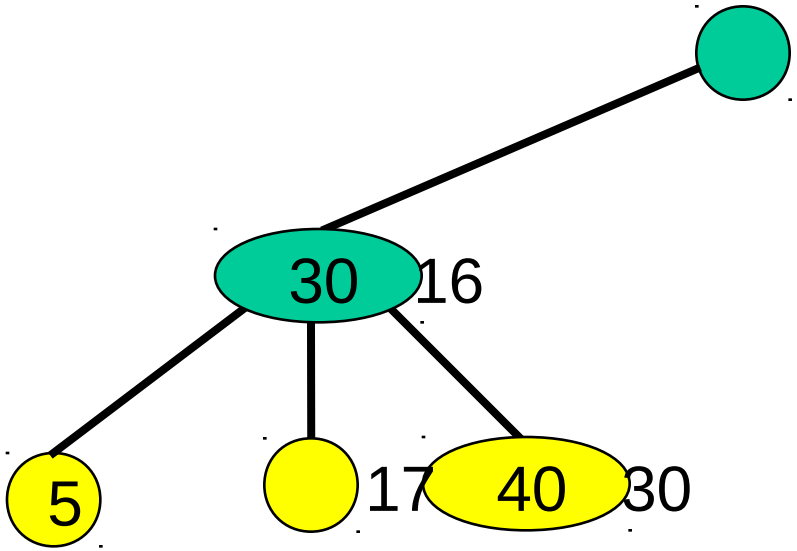
Merge with sibling, delete in-between key in • .parent

Delete



- Index node becomes deficient.
- Merge with sibling and in-between key in parent.

Delete



- Index node becomes deficient.

- It's the root; discard.