B Tree & B+ Tree

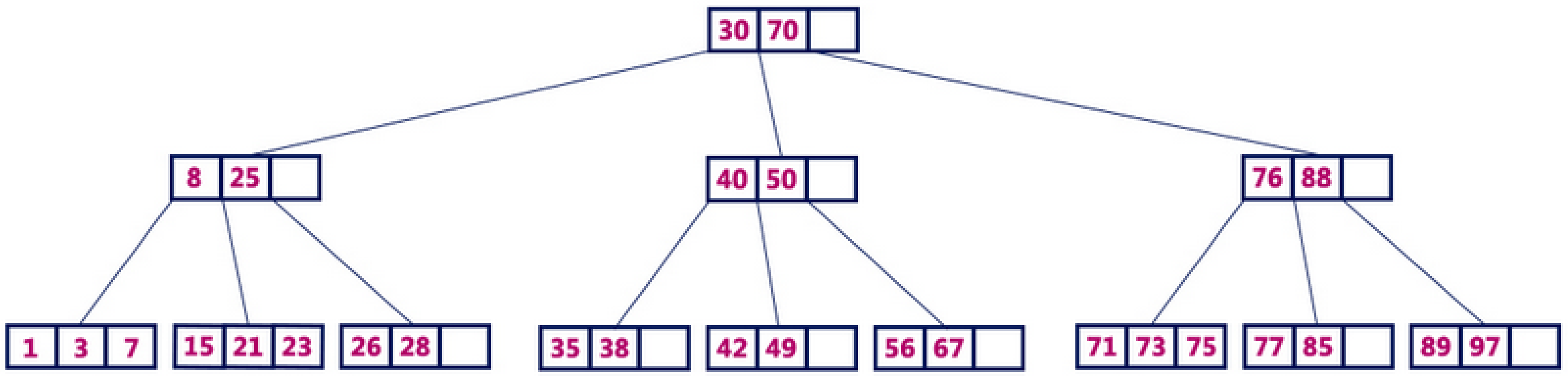
B - Tree

- B-Tree is m way search tree (m is odd number)
- It is a self-balanced search tree in which every node contains multiple keys and has more than two children.
- Large degree B-trees used to represent very large dictionaries that reside on disk.
- Smaller degree B-trees used for internal-memory dictionaries to overcome cache-miss penalties

Properties of B-Tree

- **#1** - All **leaf nodes** must be **at same level**.
- **#2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **#3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **#4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **#5** - A non leaf node with **n-1** keys must have **n** number of children.
- **#6** - All the **key values in a node** must be in **Ascending Order**.

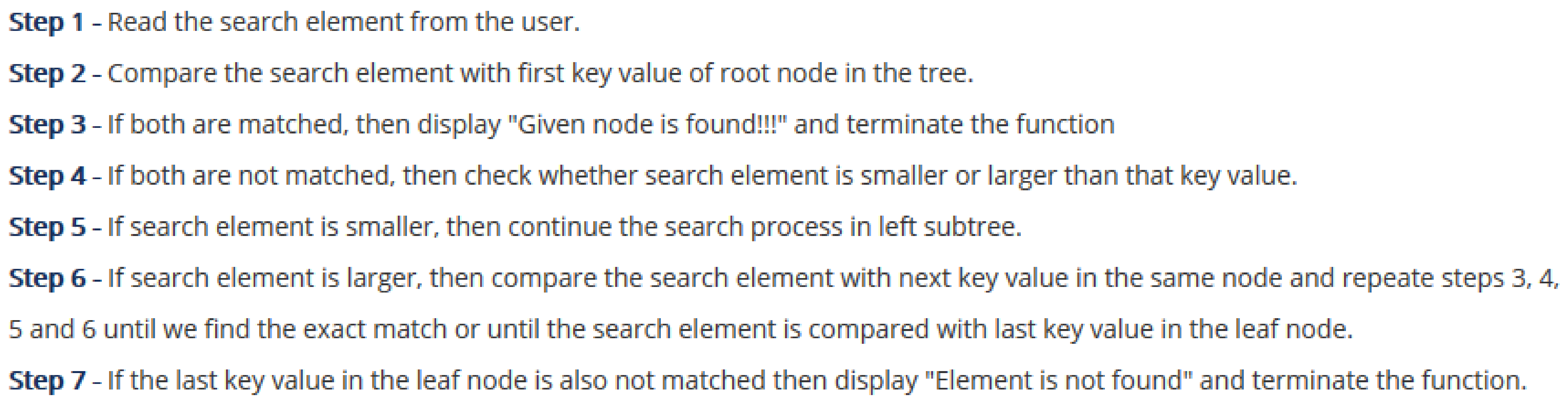
B-Tree of Order 4



Operations on a B-Tree

- **Search**
- **Insertion**
- **Deletion**

Search Operation in B-Tree



Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

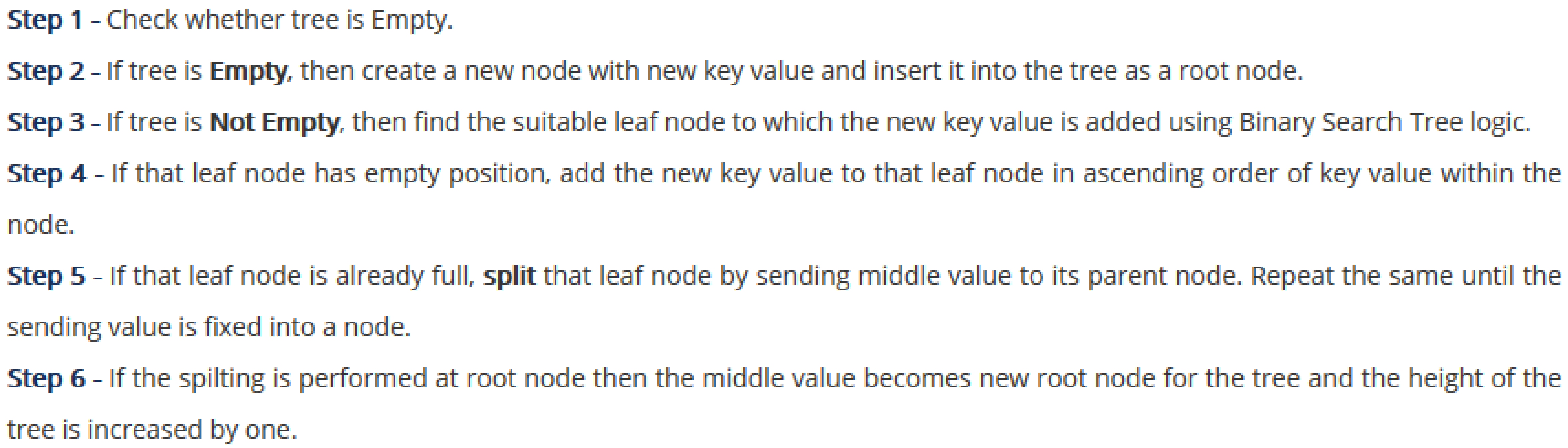
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree (Same as construction)



Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

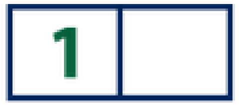
Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

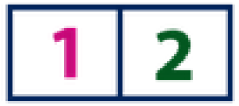
insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



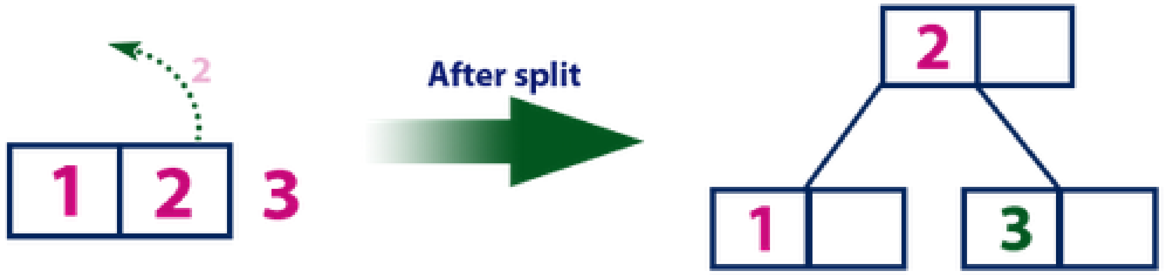
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



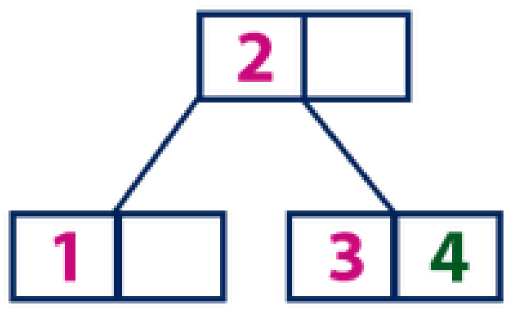
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



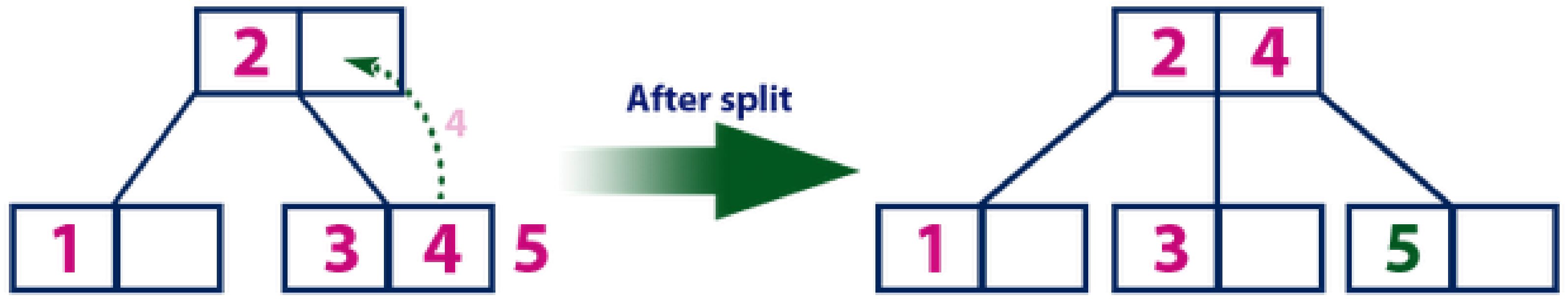
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



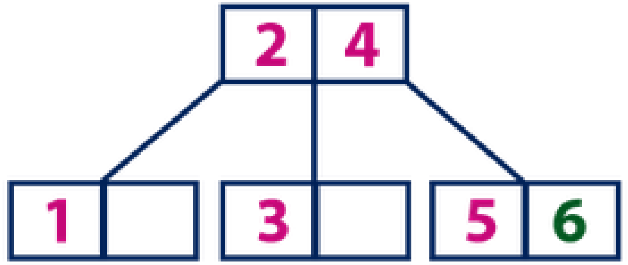
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



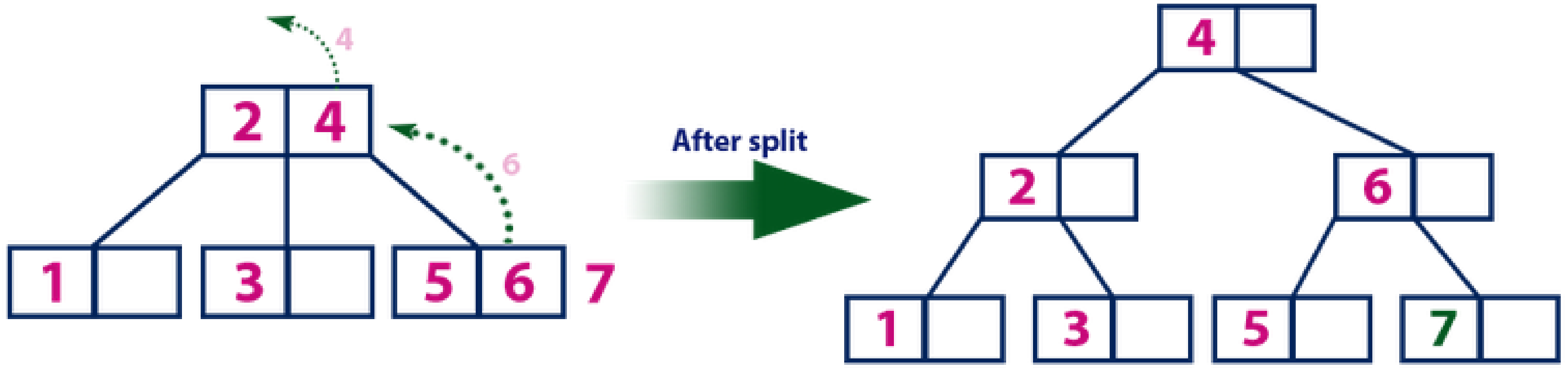
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



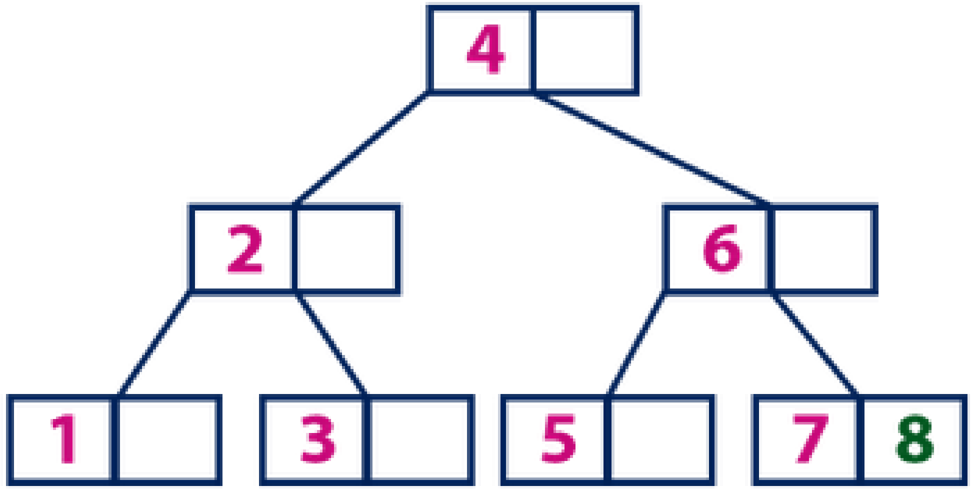
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



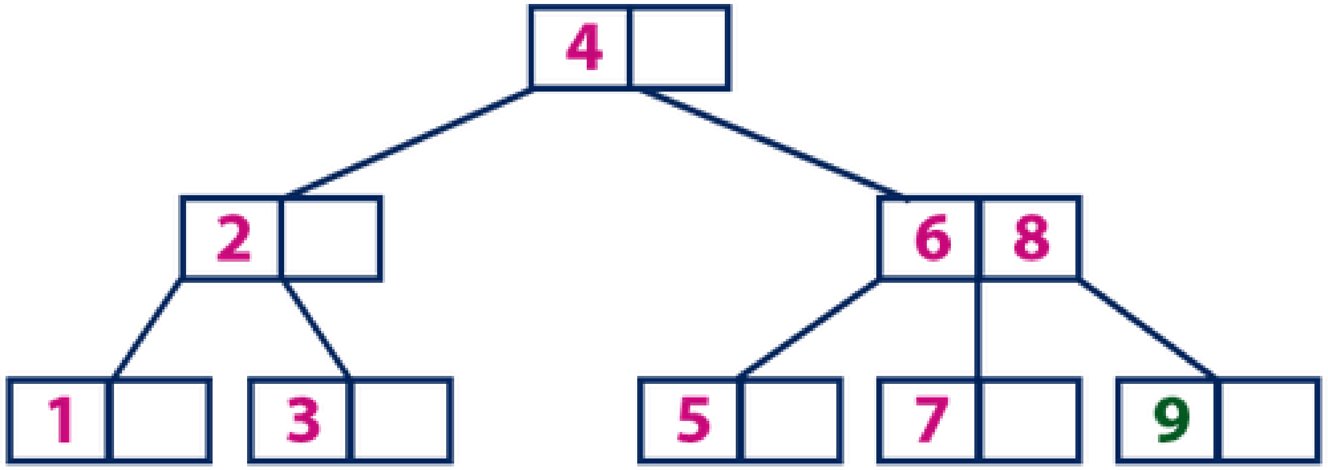
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



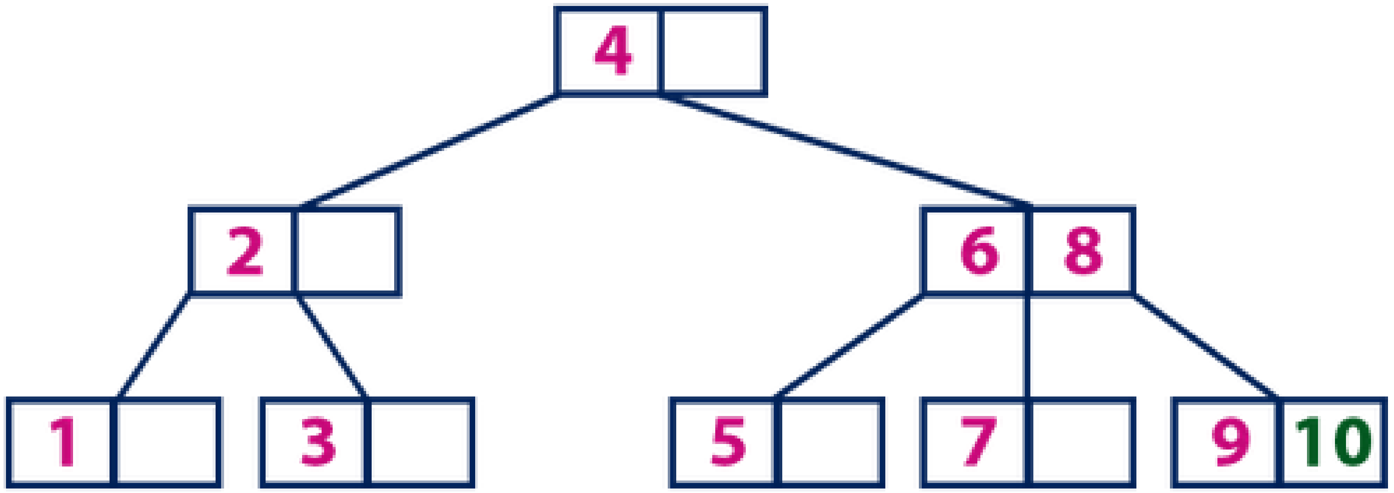
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.

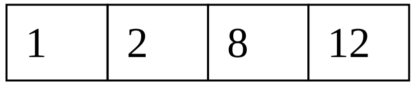


Constructing a B-tree : Example 2

- Suppose we start with an empty B-tree and keys arrive in the following order:

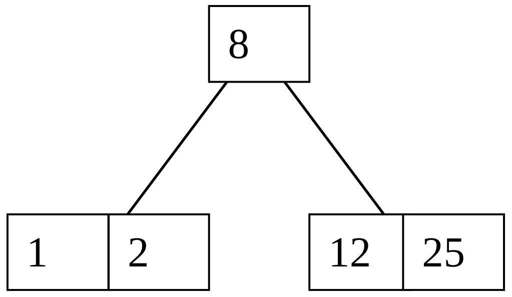
  1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

- We want to construct a B-tree of order 5
- The first four items go into the root:

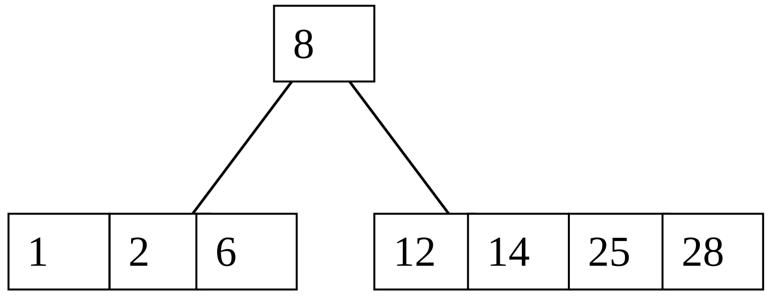


- To put the fifth item in the root would violate rule (maximum m-1 nodes)
- Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

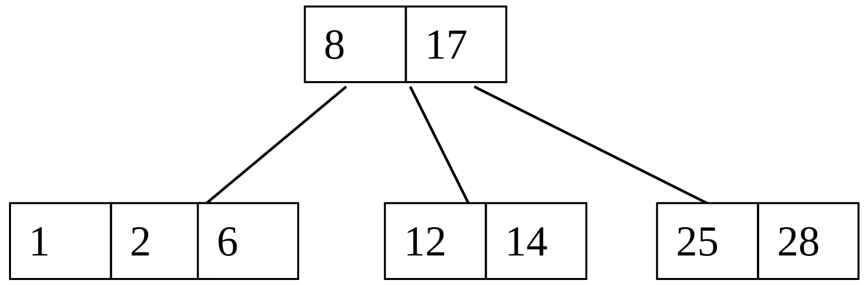


6, 14, 28 get added to the leaf nodes:

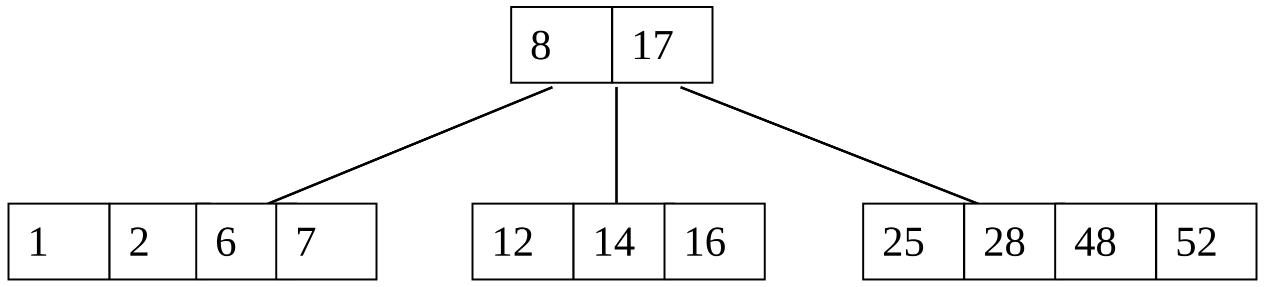


Constructing a B-tree (contd.)

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

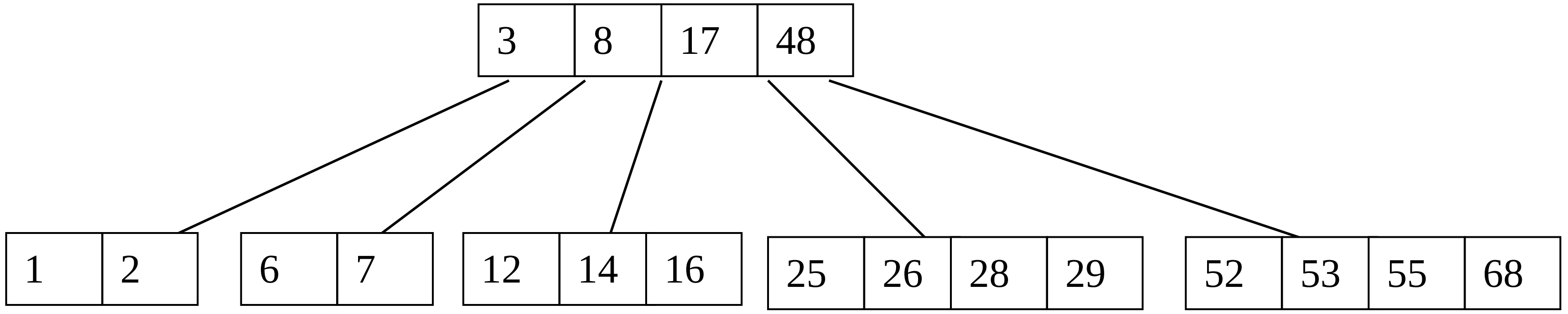


7, 52, 16, 48 get added to the leaf nodes

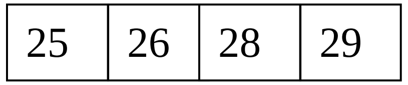


Constructing a B-tree (contd.)

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

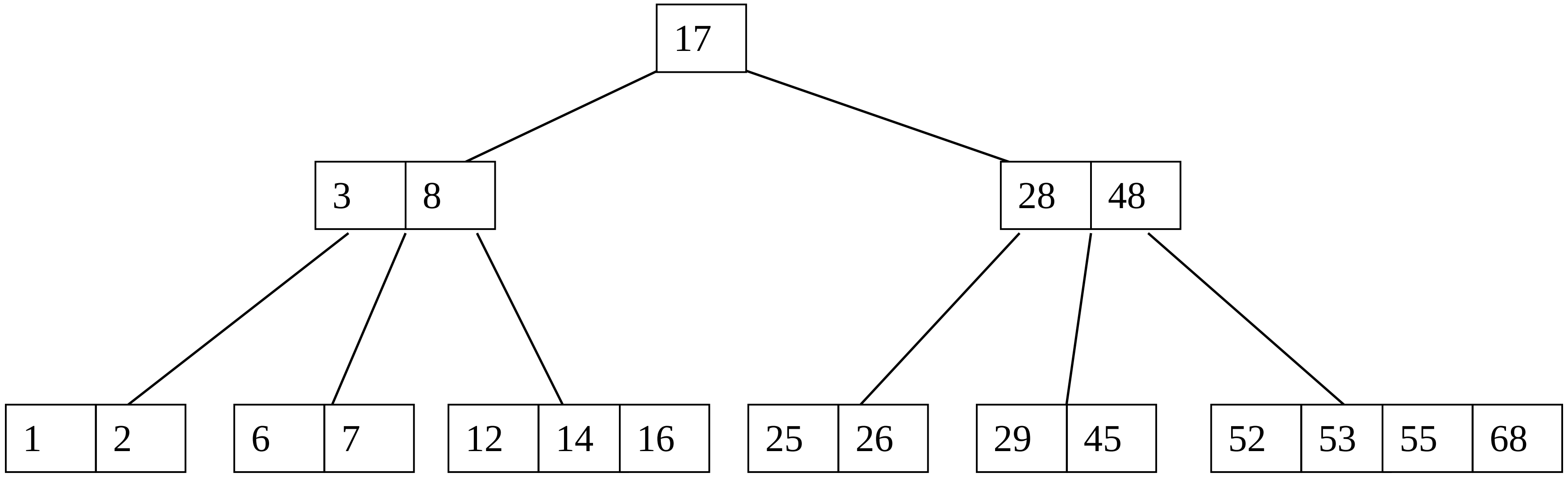


Adding 45 causes a split of



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

Constructing a B-tree (contd.)



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:

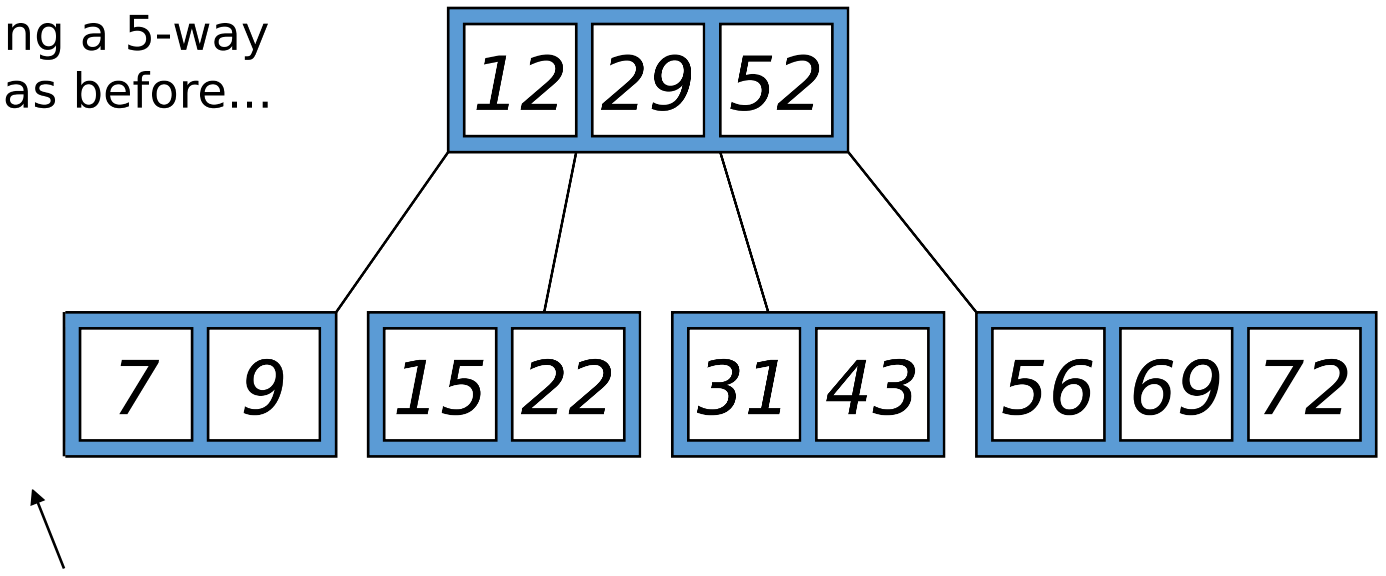
- 1 - If the key is already in a leaf node, and removing it doesn’t cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this 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

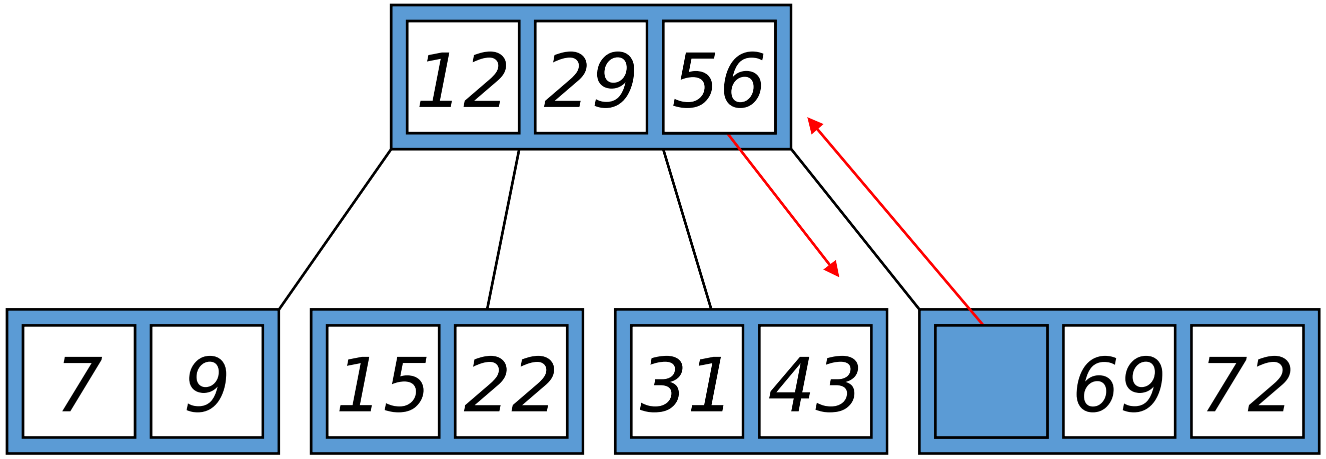
Type #1: Simple leaf deletion

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

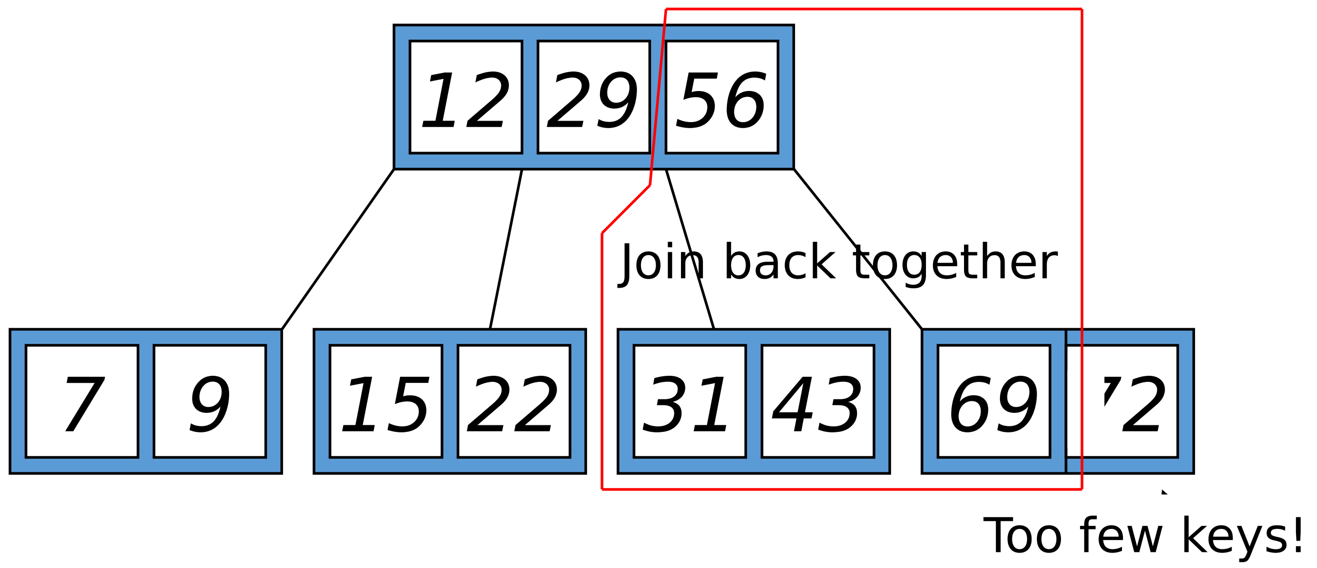


Delete 2: Since there are enough keys in the node, just delete it

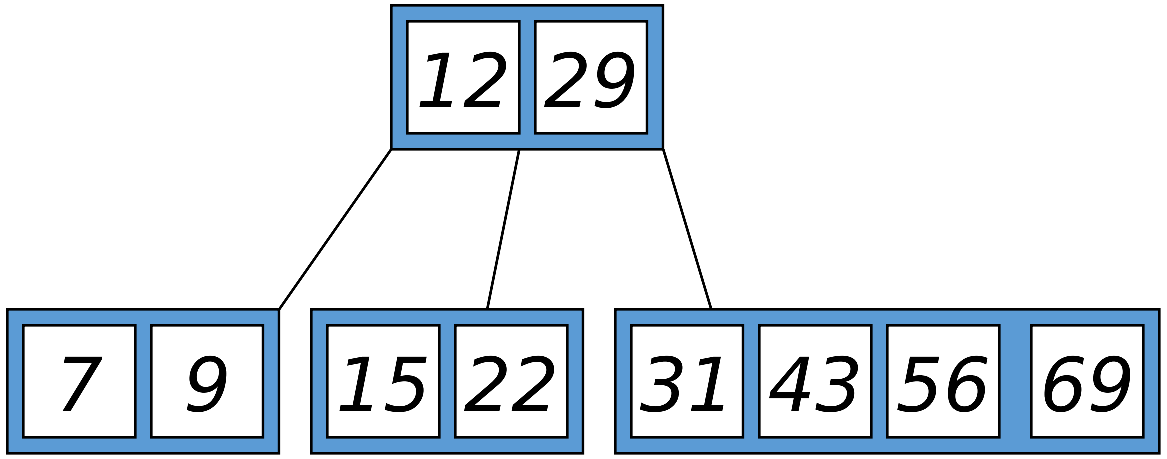
Type #2: Simple non-leaf deletion



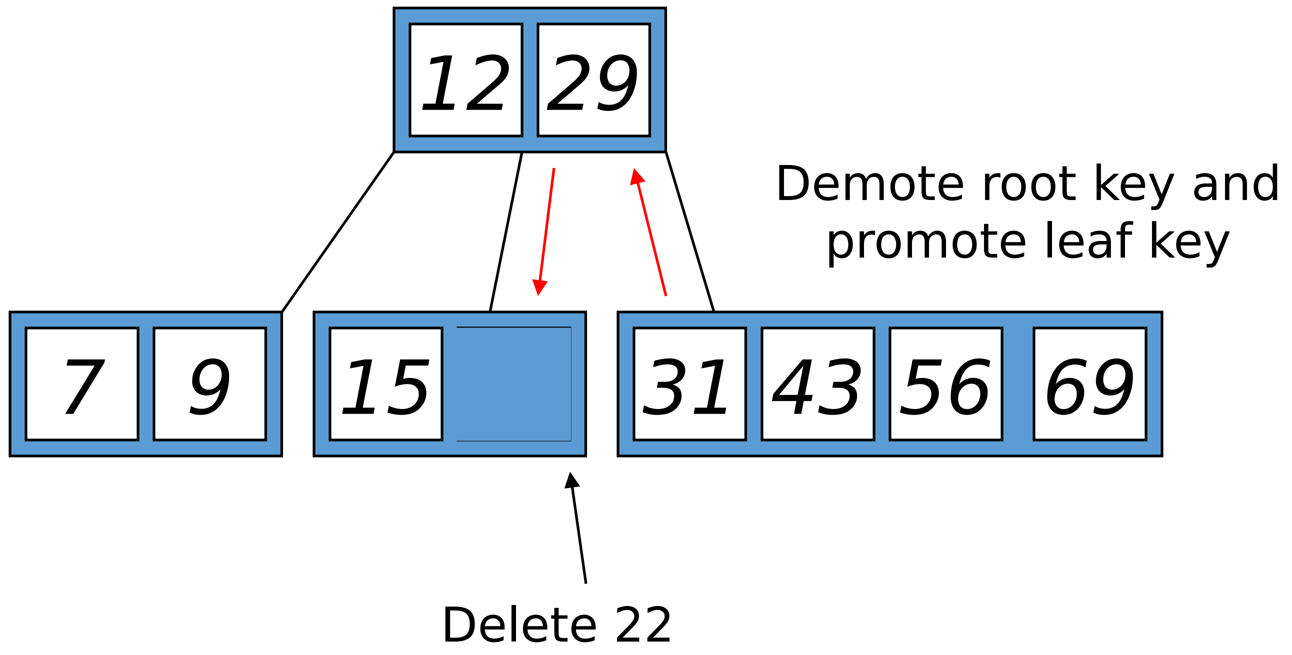
Type #4: Too few keys in node and its siblings



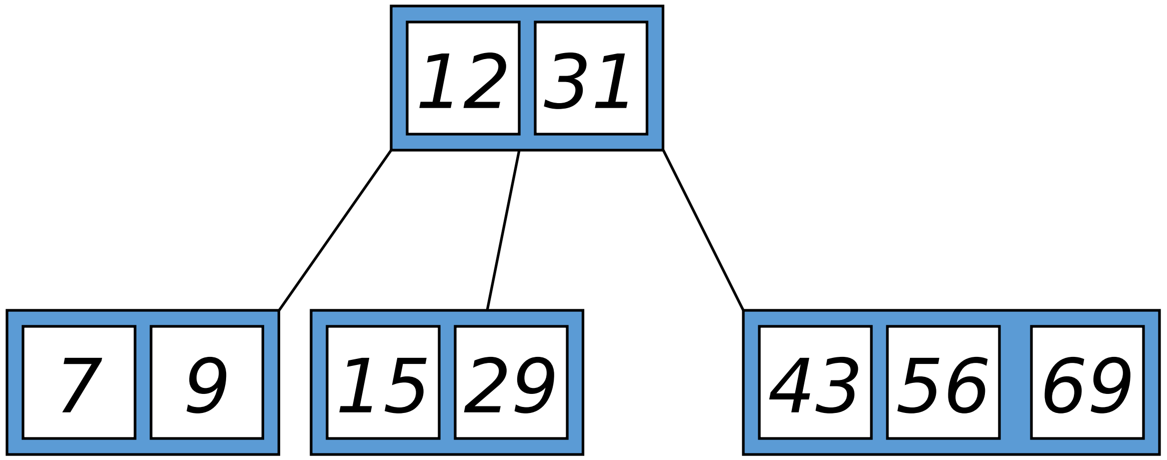
Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Type #3: Enough siblings



Advantages of B tree

- Ordered sequential access over the key value on O(n) time.
- O(log n) insert time, while maintaining the ordering of the items.
- O(log n) delete time of items within the B-Tree.
- If sequential access is handled through the B-Tree then O(log n) delete time is provided for the underlying table as well.

Comparison

Basis for comparison	B-tree	Binary tree
Essential constraint	A node can have at max M number of child nodes(where M is the order of the tree).	A node can have at max 2 number of subtrees.
Used	It is used when data is stored on disk.	It is used when records and data are stored in RAM.
Height of the tree	$\log_M N$ (where m is the order of the M-way tree)	$\log_2 N$
Application	Code indexing data structure in many DBMS.	Code optimization, Huffman coding, etc

What is a B+ Tree?

- A variation of B trees in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth

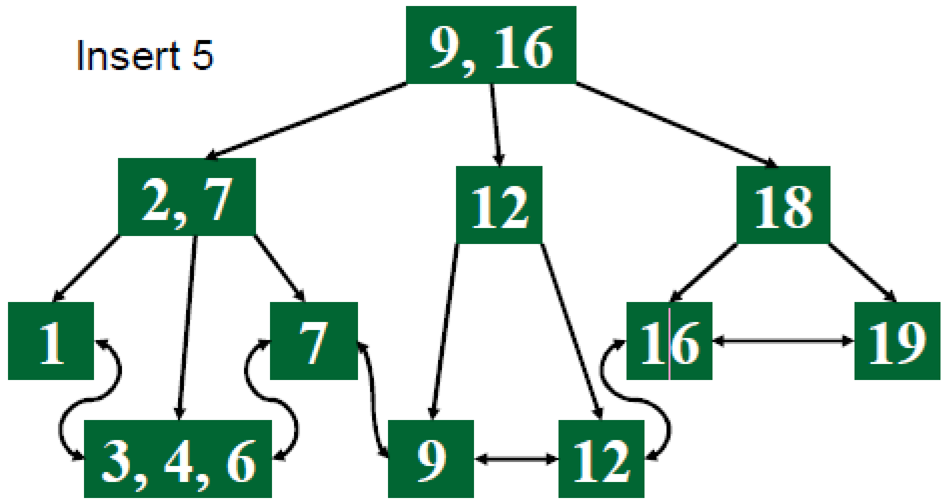
A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between [M/2] and [M] children, where n is fixed for a particular tree.

Advantages of B+ tree usage for databases

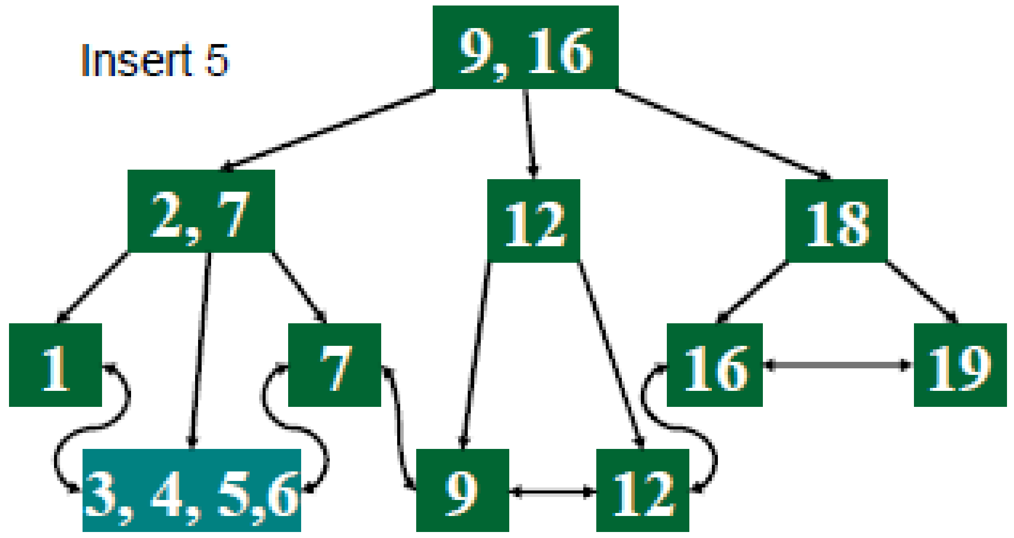
- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm
- In addition, a B+ tree minimizes waste by making sure the interior nodes are at least half full. A B+ tree can handle an arbitrary number of insertions and deletions.

B+ Tree insertion

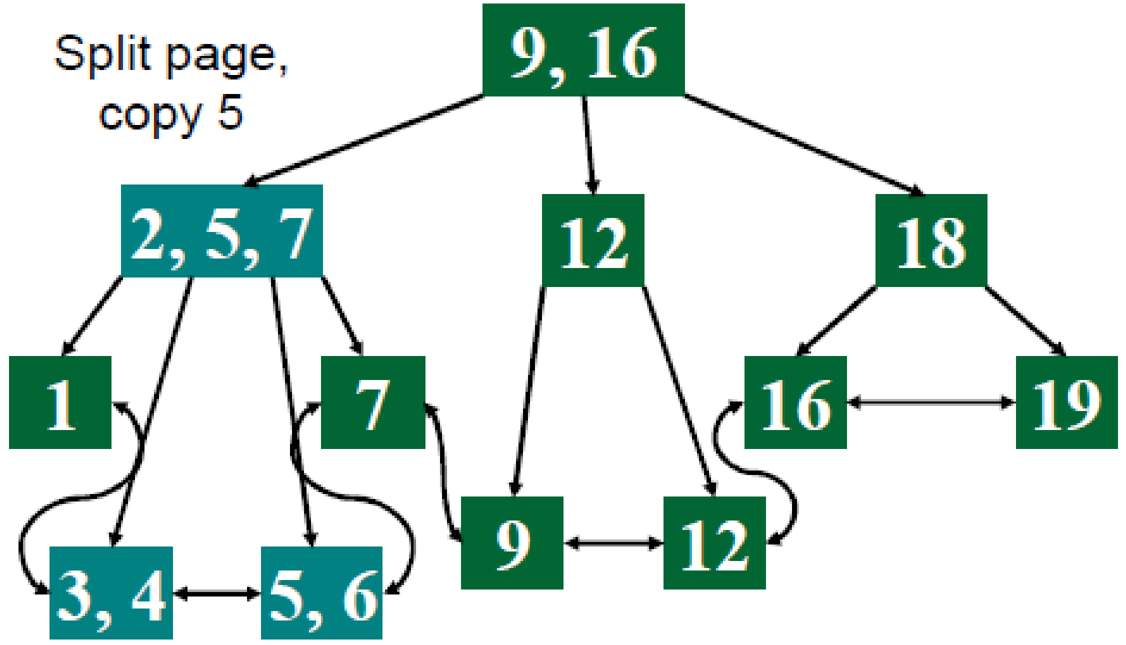
- Insert at bottom level
- If leaf page overflows, split page and copy middle element to next index page
- If index page overflows, split page and move middle element to next index page



Insert 5



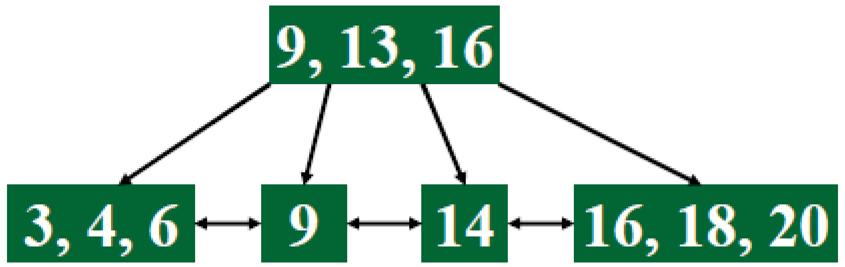
Split page, copy 5



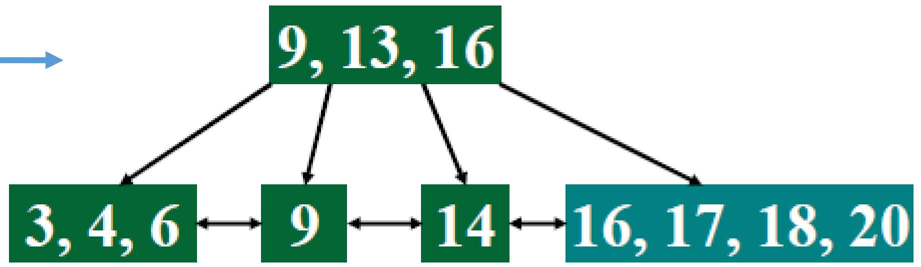
Insertion – Example 2

Consider the following tree.

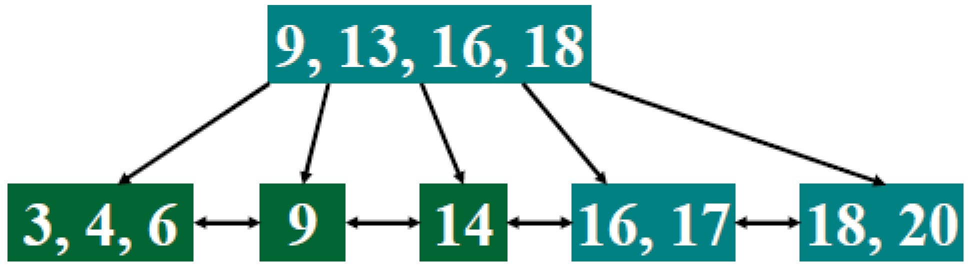
Insert 17



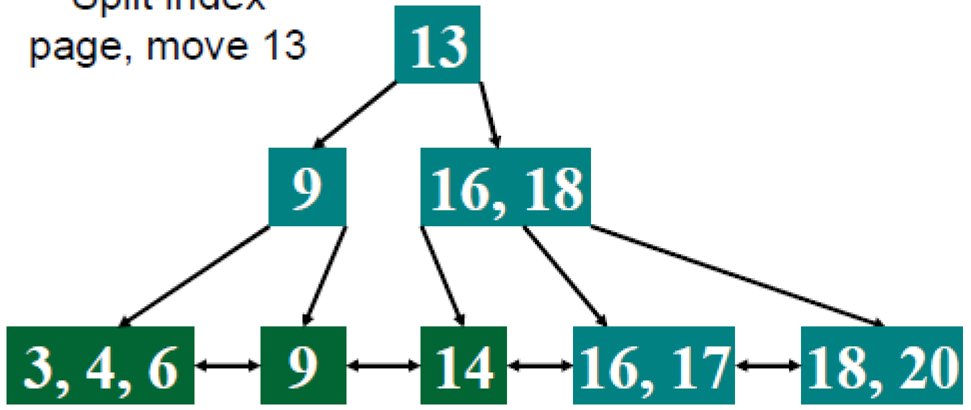
Insert 17



Split leaf page, copy 18

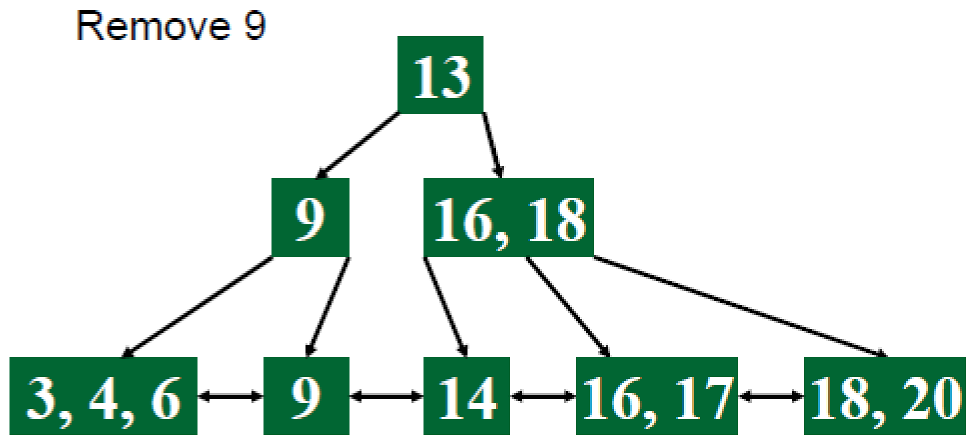


Split index page, move 13

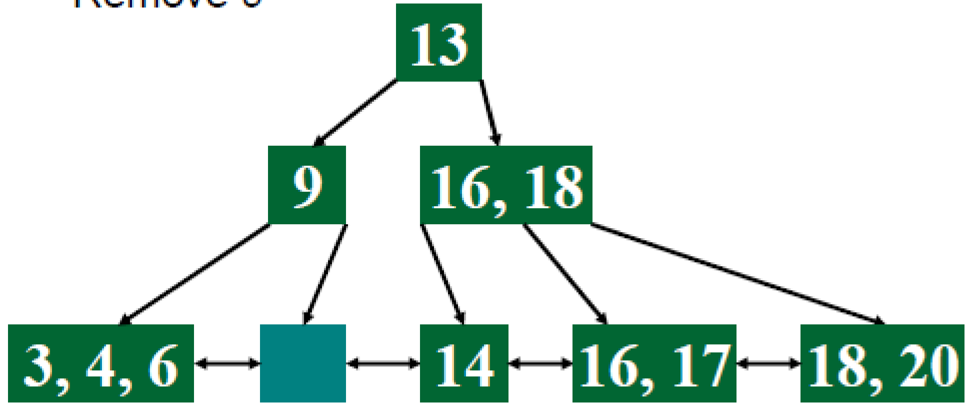


B+ Tree Deletion

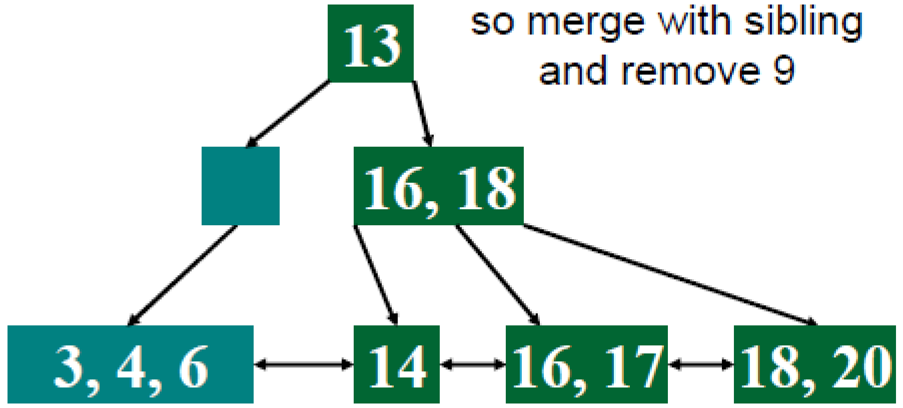
- Delete key and data from leaf page
- If leaf page underflows, merge with sibling and delete key in between them
- If index page underflows, merge with sibling and move down key in between them



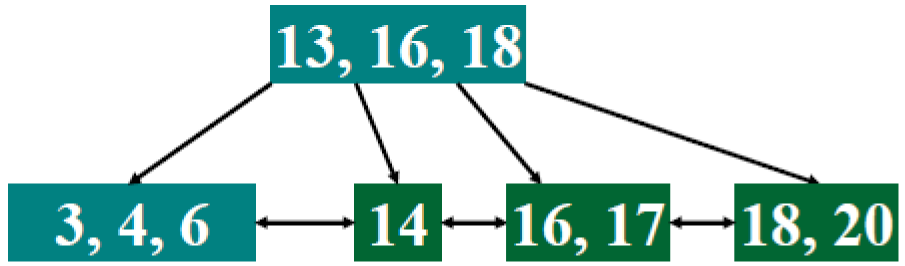
Remove 9



Leaf page underflow, so merge with sibling and remove 9



Index page underflow, so merge with sibling and demote 13



13, 16, 18
3, 4, 6
14
16, 17
18, 20