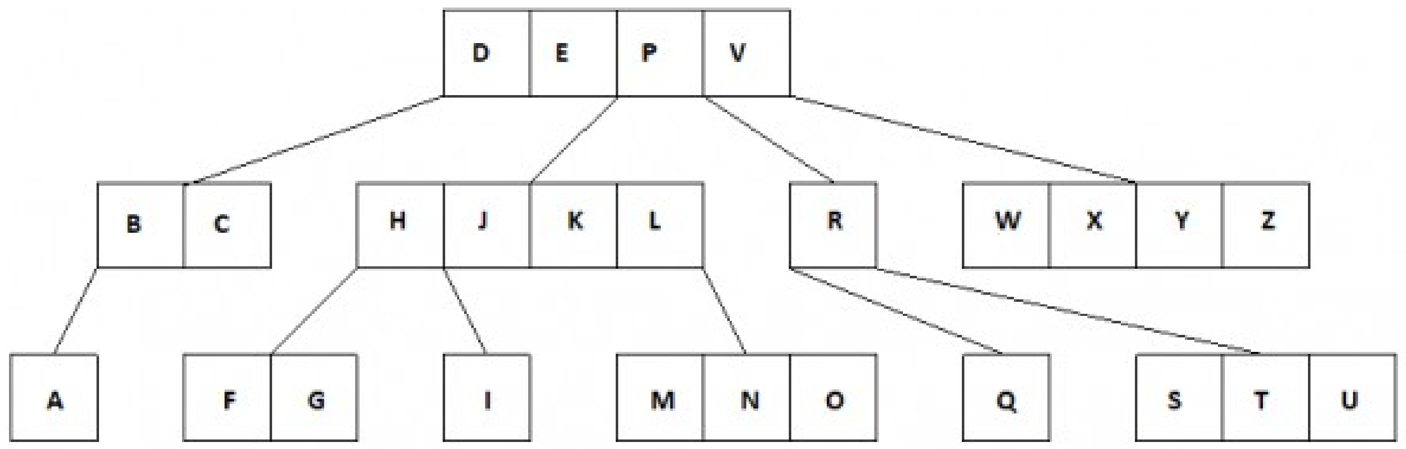
Multi-way Tree
&
B – Tree

Prof. Siddharth Shah

Multi-way Tree

- A multiway tree is defined as a tree that can have more than two children.
- If a multiway tree can have maximum **M** children, then this tree is called as multiway tree of order **M** (or an **M-way** tree).
- The nodes in an m-way tree will be made up of **M-1** key fields and pointers to children.
- **e.g. multiway tree of order 5**



Multi-way Search Tree

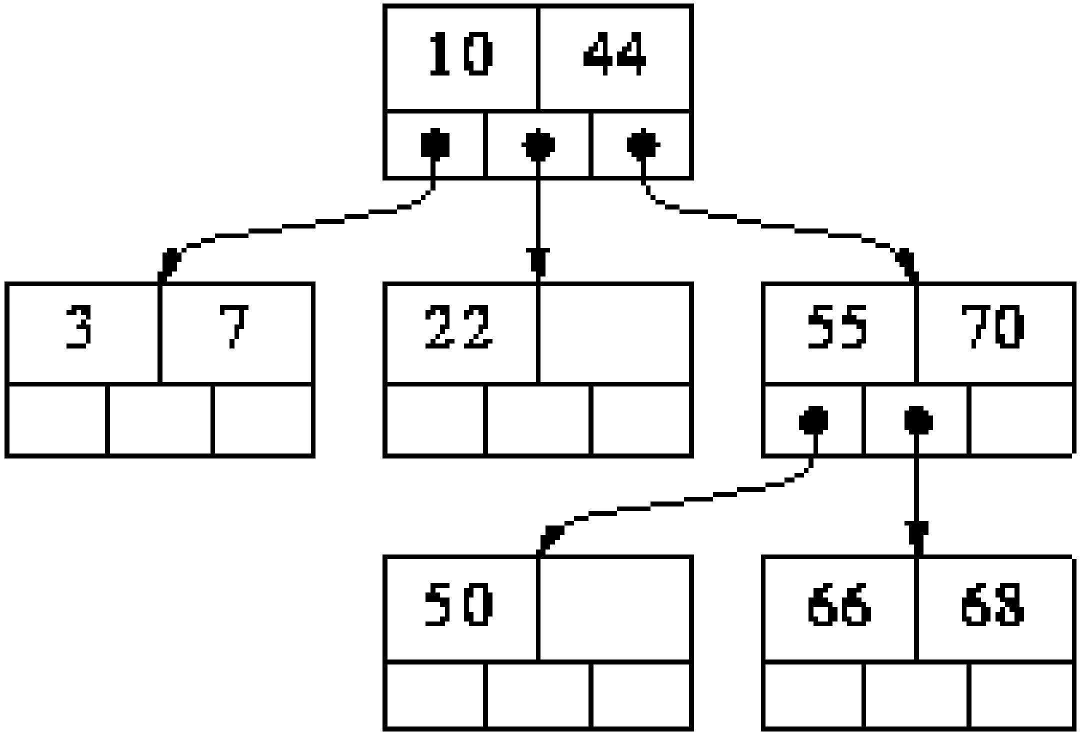
- To make the processing of M-way trees easier some type of constraint or order will be imposed on the keys within each node, resulting in a multiway search tree of order M (or an M-way search tree).
- By definition an M-way search tree is a M-way tree in which following condition should be satisfied −
  - Each node is associated with at max M children and M-1 key fields.
  - The keys in each node are arranged in ascending order.
  - The keys in the first j children are less than the j-th key.
  - The keys in the last M-j children are higher than the j-th key.

Multi-way Search Tree

- In fact, it is not necessary for every node to contain exactly (M-1) values and have exactly M subtrees.
- In an M-way subtree a node can have anywhere from 1 to (M-1) values, and the number of (non-empty) subtrees can range from 0 (for a leaf) to 1+(the number of values).
- M is thus a fixed upper limit on how much data can be stored in a node.
- The values in a node are stored in ascending order, $V_1 < V_2 < \ldots V_k$ ($k <= M-1$) and the subtrees are placed between adjacent values, with one additional subtree at each end.
- We can thus associate with each value a ‘left’ and ‘right’ subtree, with the right subtree of V_i being the same as the left subtree of V_{i+1}.
- All the values in V_1's left subtree are less than V_1 ; all the values in V_k's right subtree are greater than V_k; and all the values in the subtree between V_i and V_{i+1} are greater than V_i and less than V_{i+1}.

Example: Multi-way Search Tree

3- way search tree



Search Operation

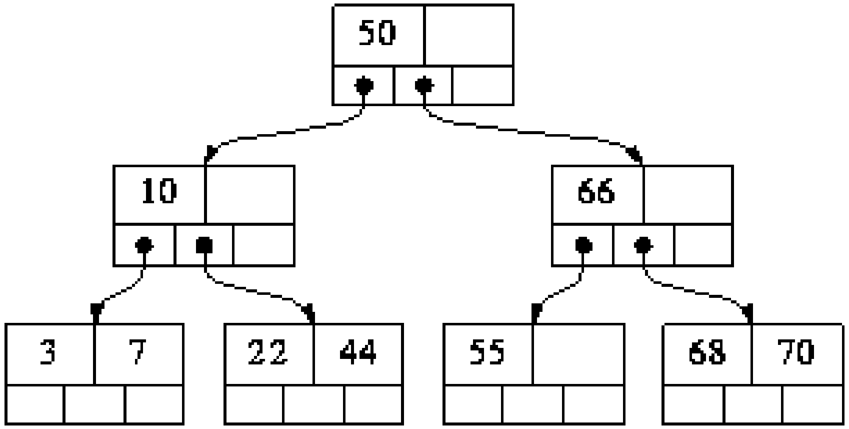
- The algorithm for searching for a value in an M-way search tree is the obvious generalization of the algorithm for searching in a binary search tree.
- If we are searching for value X, and currently at node consisting of values $V_1 \ldots V_k$, there are four possible cases that can arise:
    1. If $X < V_1$, recursively search for X in V_1's left subtree.
    2. If $X > V_k$, recursively search for X in V_k's right subtree.
    3. If $X=V_i$, for some i, then we are done (X has been found).
    4. The only remaining possibility is that, for some i, $V_i < X < V_{i+1}$. In this case recursively search for X in the subtree that is in between V_i and V_{i+1}.

Insertion & Deletion Operations

- The other algorithms for binary search trees - insertion and deletion - generalize in a similar way.
- As with binary search trees, inserting values in ascending order will result in a degenerate M-way search tree; i.e. a tree whose height is O(N) instead of O(logN). This is a problem because all the important operations are O(height), and it is our aim to make them O(logN).
- One solution to this problem is to force the tree to be height-balanced.
- Such height-balanced M-way search tree is called **B-Tree**

B – Tree

- B-Tree is a self-balanced M-way search tree.
- Every B-Tree has order, which decides the maximum number of keys and children of a node.
- A B-Tree with order M has following properties:
  - All the leaf nodes must be at same level.
  - All non leaf nodes except root (i.e. all internal nodes) must have at least **ceiling (M/2)** children.
  - If the root node is a non leaf node, then it must have at least 2 children.
  - A tree is constructed in Bottom-Up manner
- **e.g. B tree of order 3**



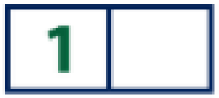
Insertion in B – Tree

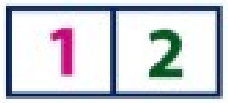
In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to leaf node only. The insertion operation is performed as follows.

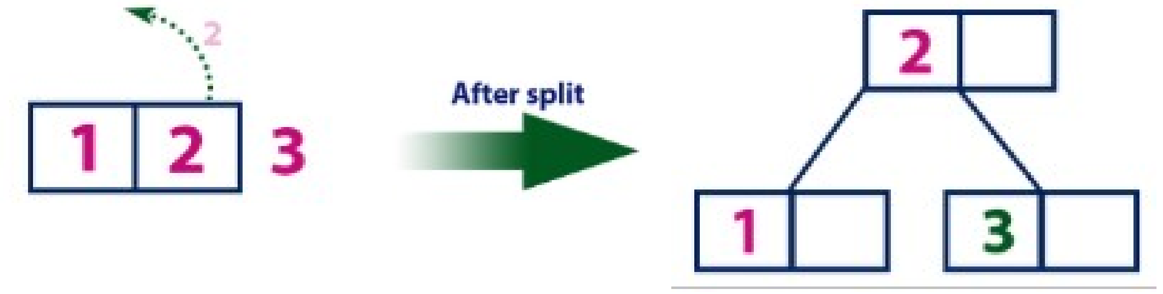
1. Check whether tree is Empty.
2. If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
3. If tree is Not Empty, then find a leaf node to which the new key value can be added using Binary Search Tree logic.
4. If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
5. If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.
6. If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

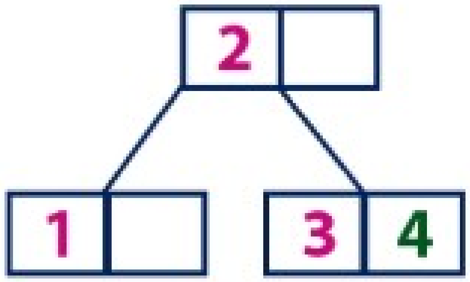
Example

Construct a B-Tree of order 3 (**2-3 Tree**) by inserting numbers 1 to 10



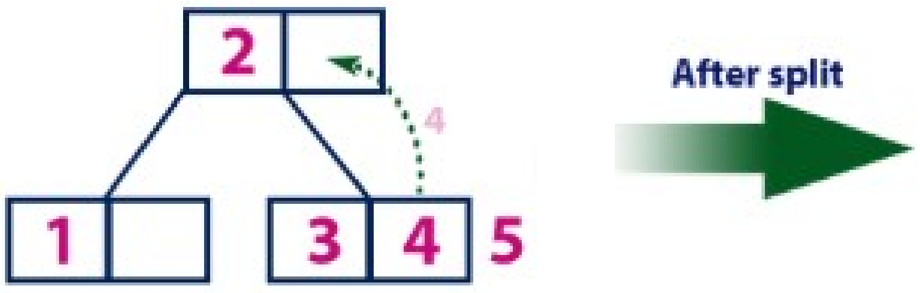


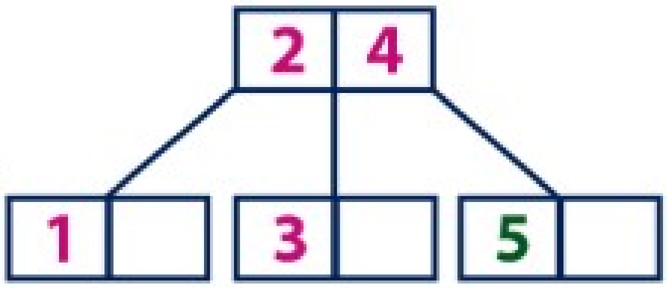


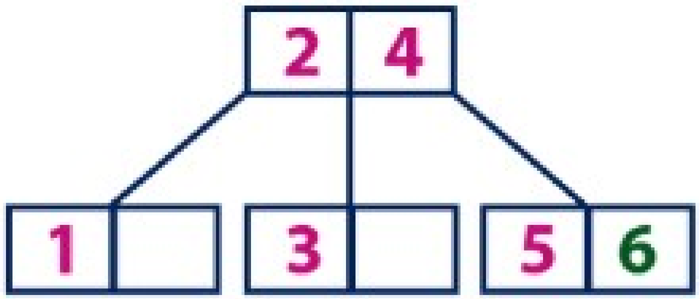


Example

Construct a B-Tree of order 3 (**2-3 Tree**) by inserting numbers 1 to 10

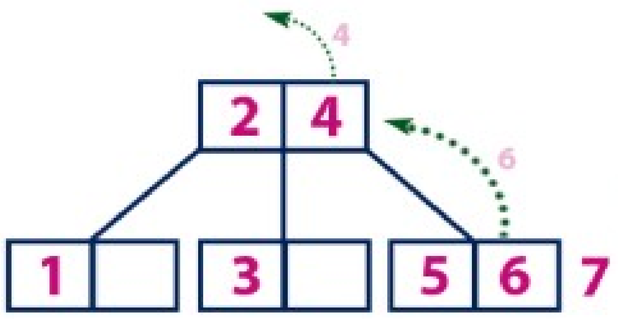


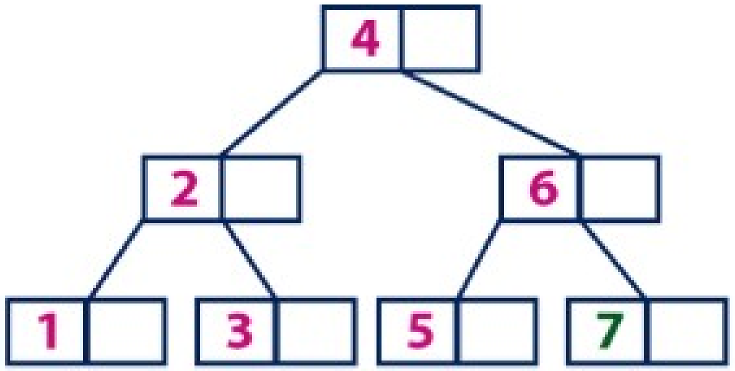


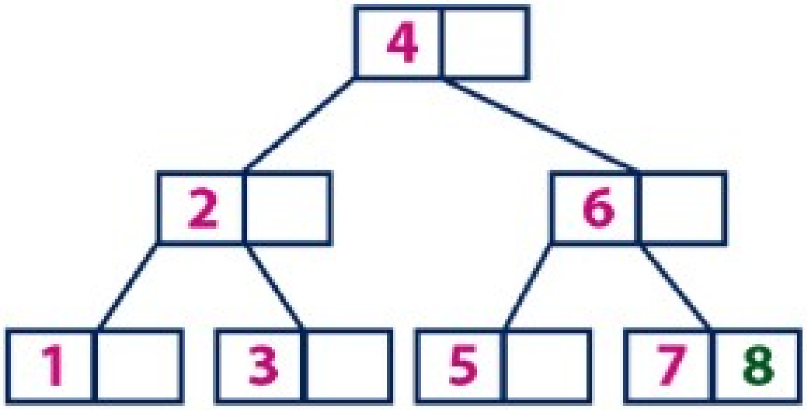


Example

Construct a B-Tree of order 3 (**2-3 Tree**) by inserting numbers 1 to 10

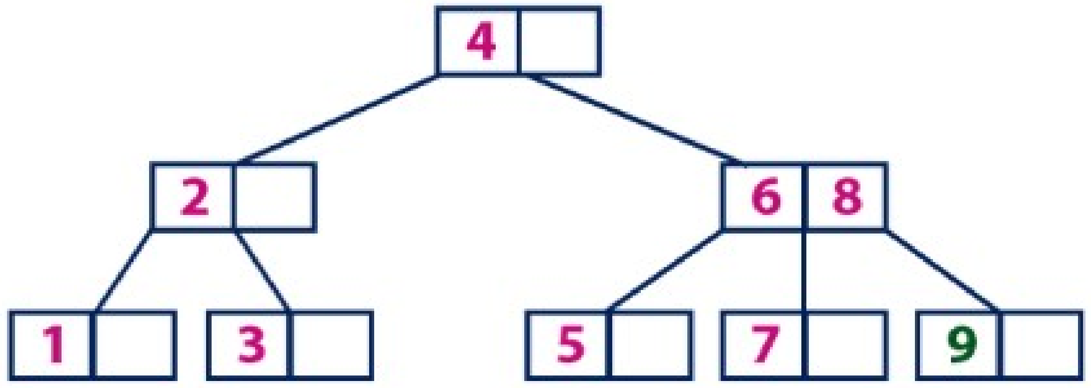


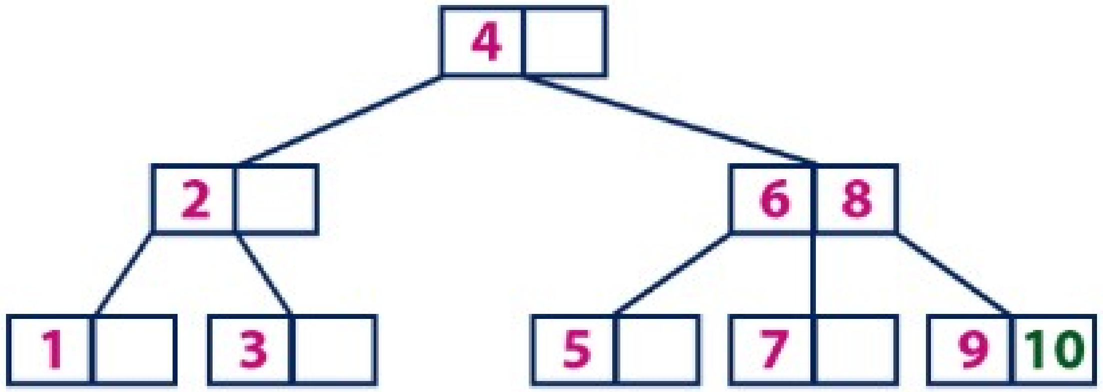




Example

Construct a B-Tree of order 3 (**2-3 Tree**) by inserting numbers 1 to 10

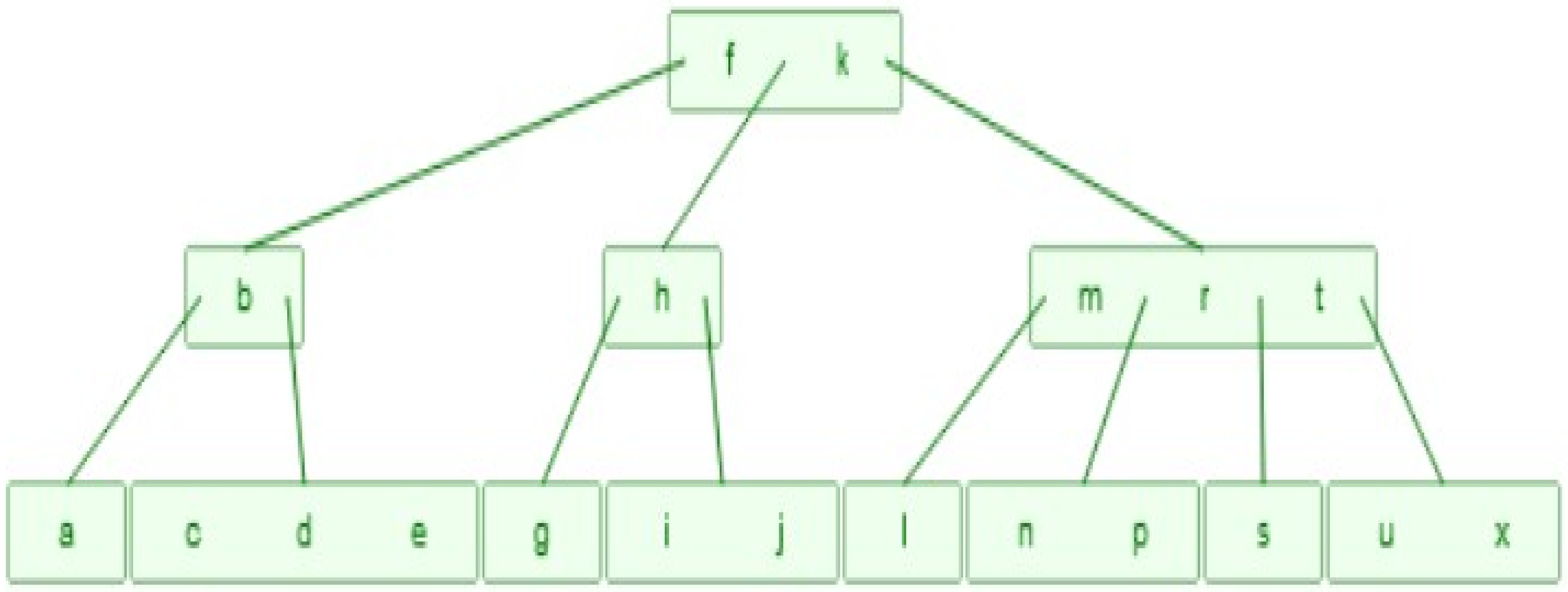




Assignment – 1

Construct a B-Tree of order 4 (2-3-4 Tree) by inserting values:

a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p.



Deletion from B – Tree

- Recall deletion algorithm for binary search trees: if the value to be deleted is in an internal node, it would be replaced by its in-order successor’s (or predecessor’s) value and then delete the node originally contained successor (or predecessor) value.

- A similar strategy is used to delete a value from a B-tree. If the value to be deleted does not occur in a leaf, it will be replaced by its successor (the smallest value in its right subtree) or predecessor (the largest value in its left subtree) and then successor or predecessor value from the node that originally contained it, will be deleted.

- As in a B-tree, the successor or predecessor is guaranteed to be in leaf. Therefore wherever the value to be deleted initially resides, the following deletion algorithm always begins at a leaf.

Deletion from B – Tree

- To delete value X from a B-tree, starting at a leaf node, there are 2 steps:
   1. Remove X from the current node. Being a leaf node there are no subtrees to worry about.
   2. Removing X might cause the node containing it to have too few values.
- It is required that the root to have at least 1 value in it and all other nodes to have at least **(M-1)/2** values in them.
- If the node has too few values, it has underflowed.
- If underflow does not occur, then the deletion process is finished.
- If it does occur, it must be fixed.

Deletion from B – Tree

Fixing the underflowed node

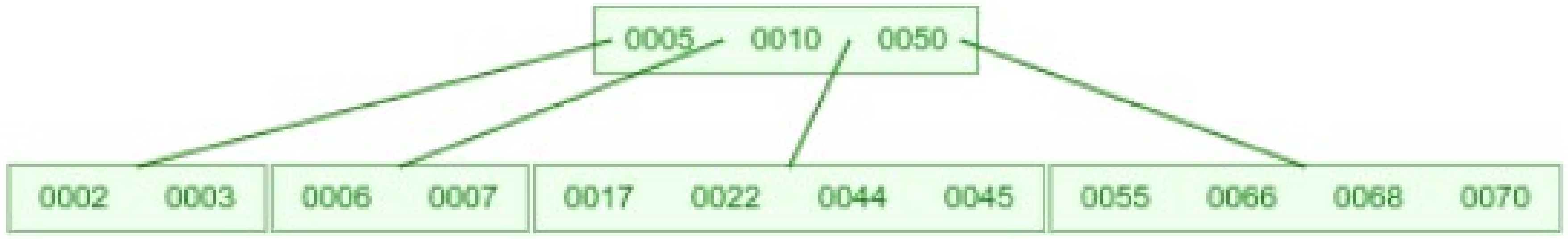
- Case 1: Fixing the underflow at non root node
- Case 2: Fixing the underflow at root node

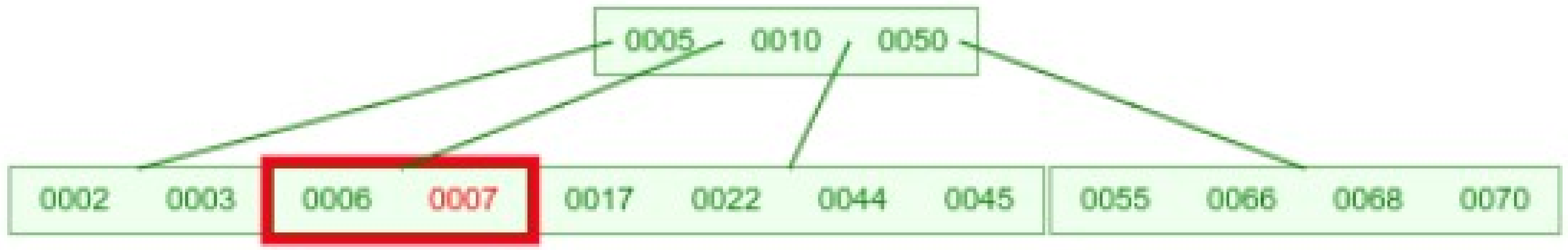
Fixing underflow at **non root** node

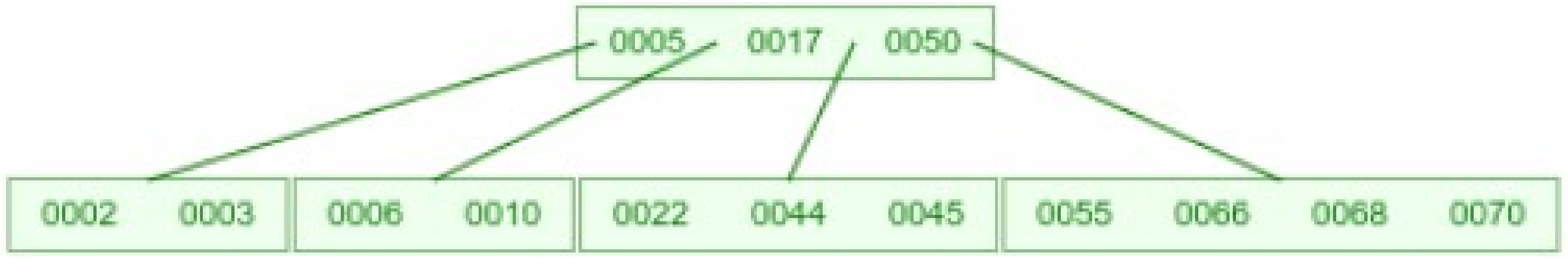
- **Case 1:** Borrow a key from its immediate neighboring sibling node in the order of left to right.
  - First, visit the immediate left sibling.If the left sibling node has more than a minimum number of keys, then borrow a key from this node.
  - Else if, immediate right sibling node has more than a minimum number of keys, then borrow a key from right sibling.
- **Case 2:** If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.

Example 1

Delete key **7** from the below given B-Tree of order **5**

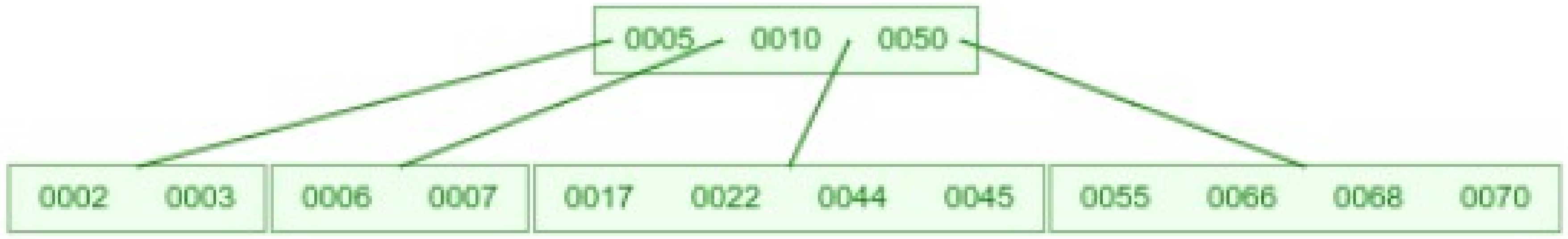


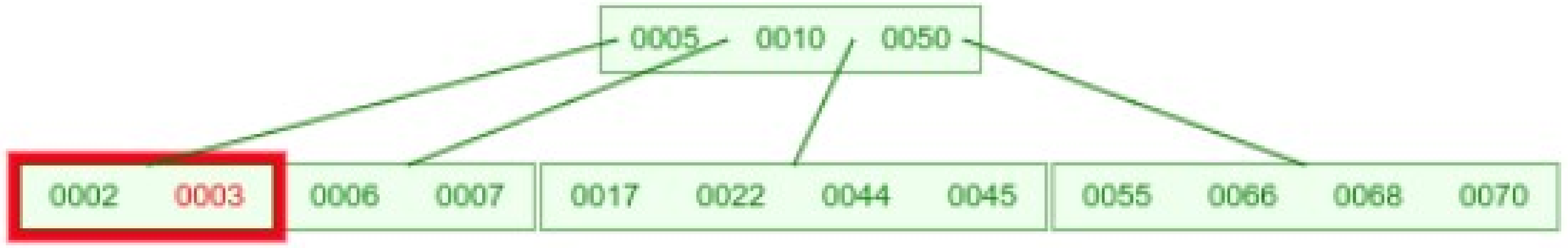


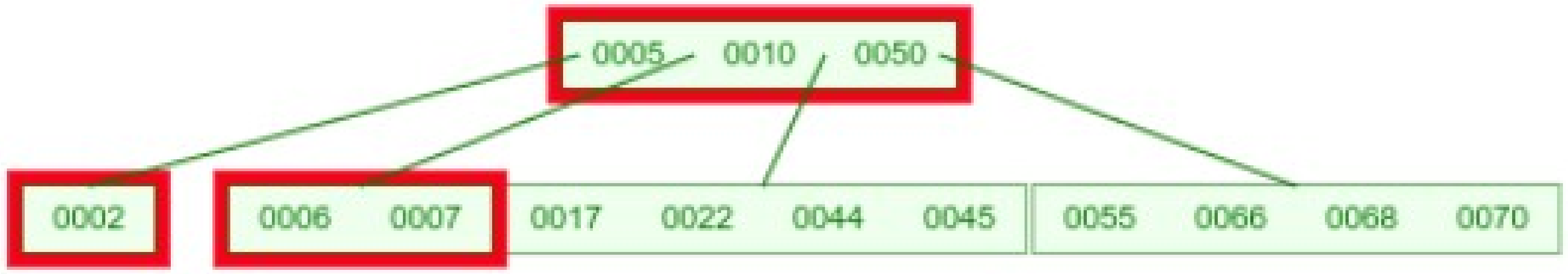


Example 2

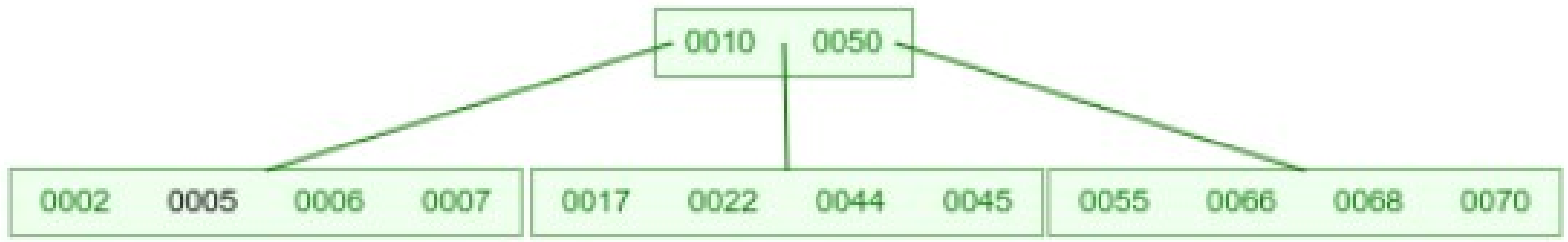
Delete key **3** from the below given B-Tree of order **5**

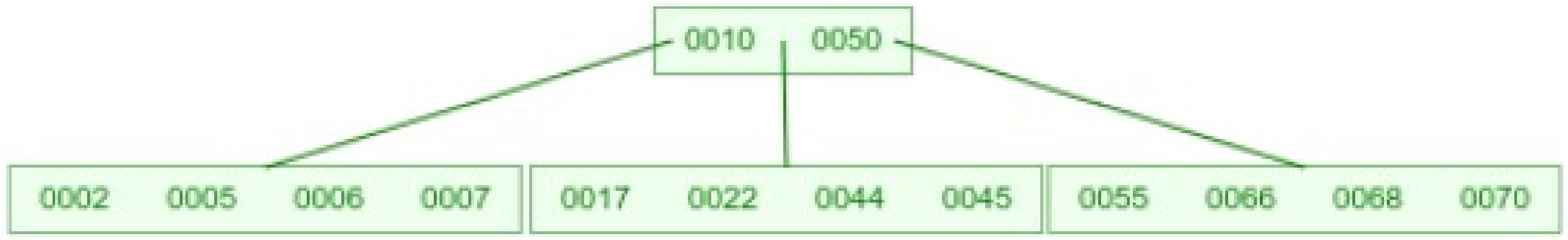






Example 2


0010 0050
0002 0005 0006 0007
0017 0022 0044 0045
0055 0066 0068 0070

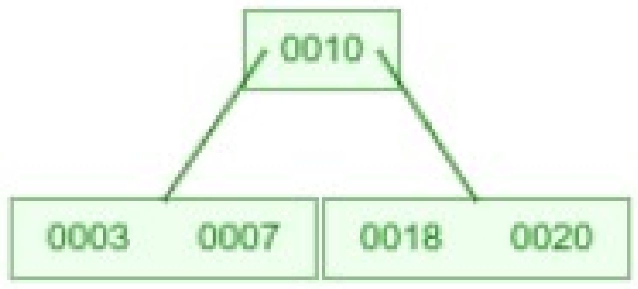

0010 0050
0002 0005 0006 0007
0017 0022 0044 0045
0055 0066 0068 0070

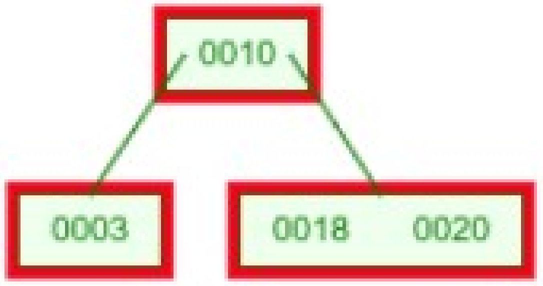
Fixing underflow at **root** node

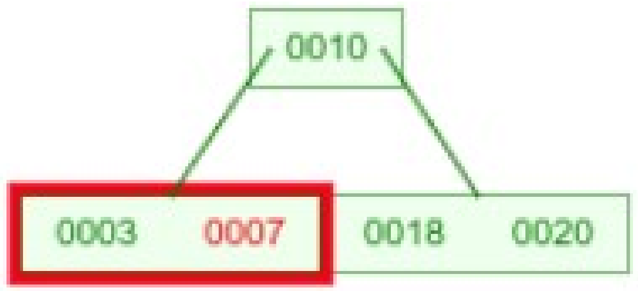
- For the root to underflow, it must have originally contained just one value, which now has been removed.
- If the root was also a leaf, then there is no problem: in this case the tree has become completely empty.
- If the root is not a leaf, it must originally have had two subtrees (because it originally contained one value). How could it possibly underflow?
    - The deletion process always starts at a leaf and therefore the only way the root could have its value removed is through the Case 2 processing described previously.
    - The root's two children have been combined, along with the root's only value to form a single node. But if the root's two children are now a single node, then that node can be used as the new root, and the current root (which has underflowed) can simply be deleted.

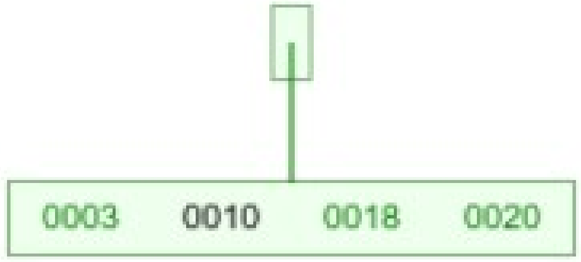
Example 3

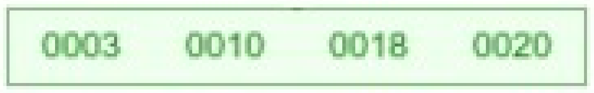
- Delete key 7











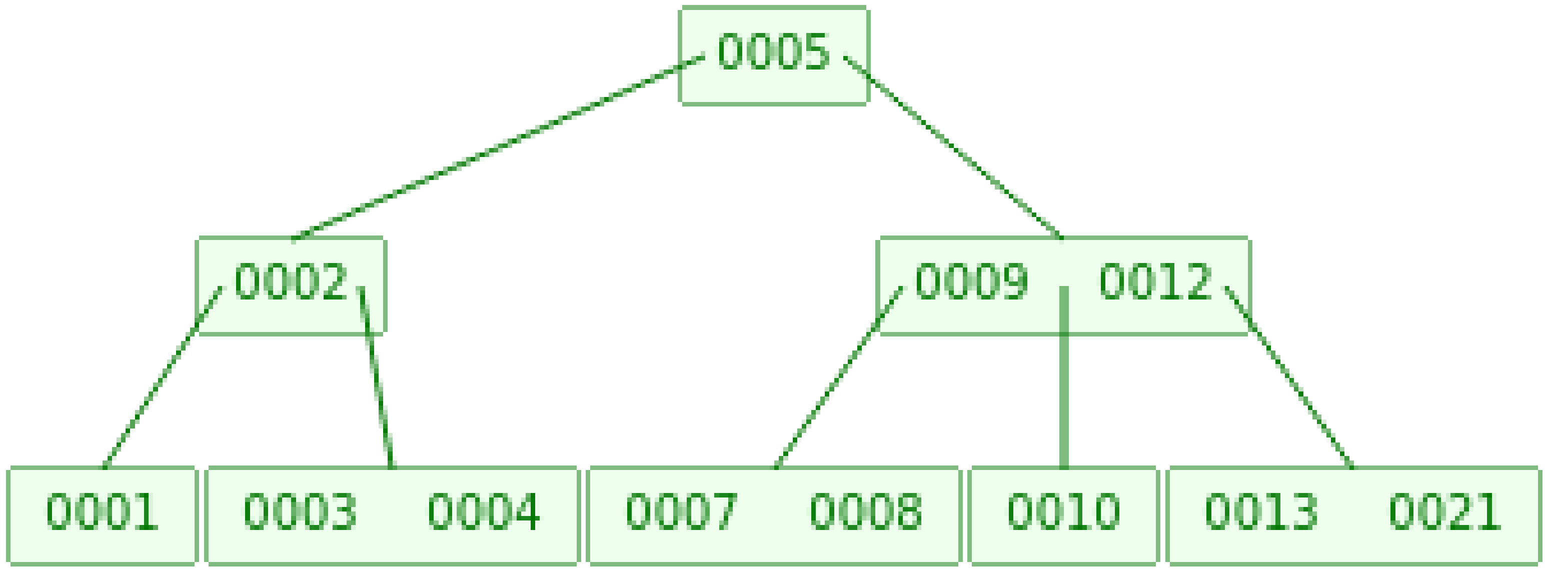
Assignment – 2

Construct B-Tree of order **4** by inserting following keys and then delete the mentioned keys in the given order.

- Insert: **5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8**
- Delete: **2, 21, 10, 3, 4**

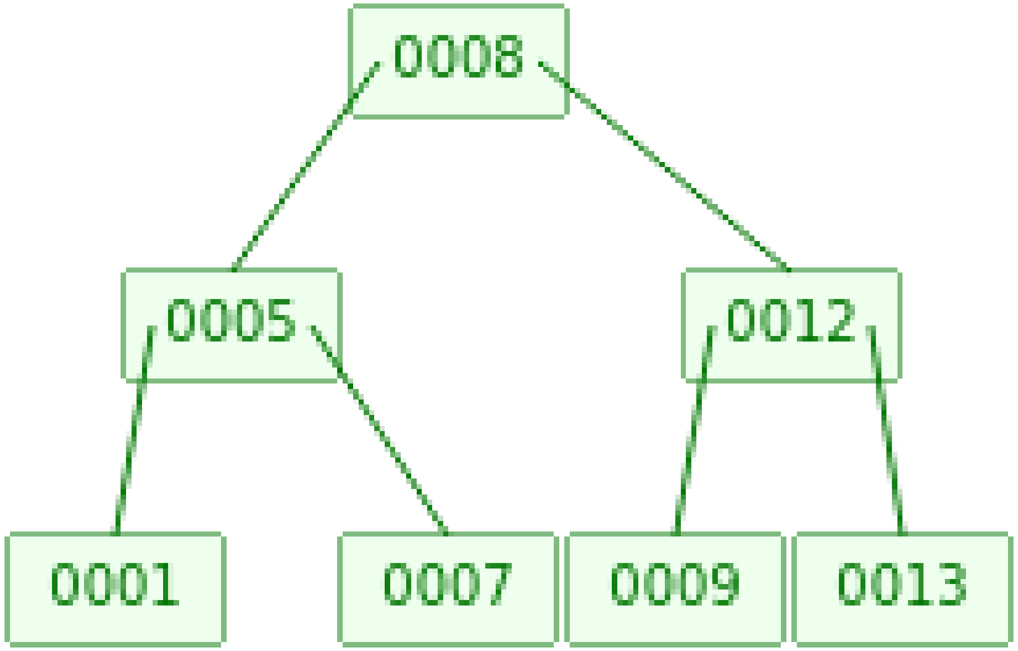
Solution to Assignment – 2

After Insertion of keys : **5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8**



Solution to Assignment – 2

After Deletion of keys : **2, 21, 10, 3, 4**


0008
0005
0012
0001
0007
0009
0013