UNIT-V : MULTIWAY SEARCH TREES

M-Way Search Trees, Definition and Properties- Searching an M-Way Search Tree, B-Trees, Definition and Properties- Number of Elements in a B-tree- Insertion into B-Tree- Deletion from a B-Tree- B+-Tree Definition- Searching a B+-Tree- Insertion into B+-tree- Deletion from a B+-Tree.

A **multiway tree** is a tree that can have more than two children. A **multiway tree of order m** (or an **m-way tree**) is one in which a tree can have m children.

m-WAY Search Trees

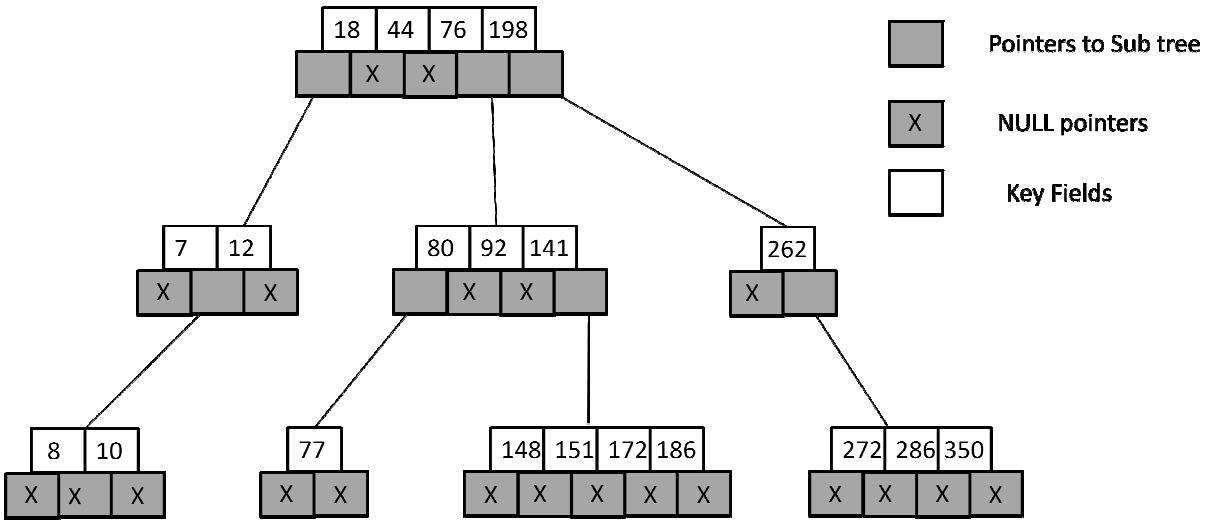
The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order **m**, each node contains a maximum of **m – 1** elements and m children.

The goal of m-Way search tree of height h calls for O(h) no. of accesses for an insert/delete/retrieval operation. Hence, it ensures that the height **h** is close to **log_m(n + 1)**.

The number of elements in an m-Way search tree of height **h** ranges from a minimum of **h** to a maximum of m^h-1.

An m-Way search tree of n elements ranges from a minimum height of **log_m(n+1)** to a maximum of **n**

An example of a 5-Way search tree is shown in the figure below. Observe how each node has at most 5 child nodes & therefore has at most 4 keys contained in it.



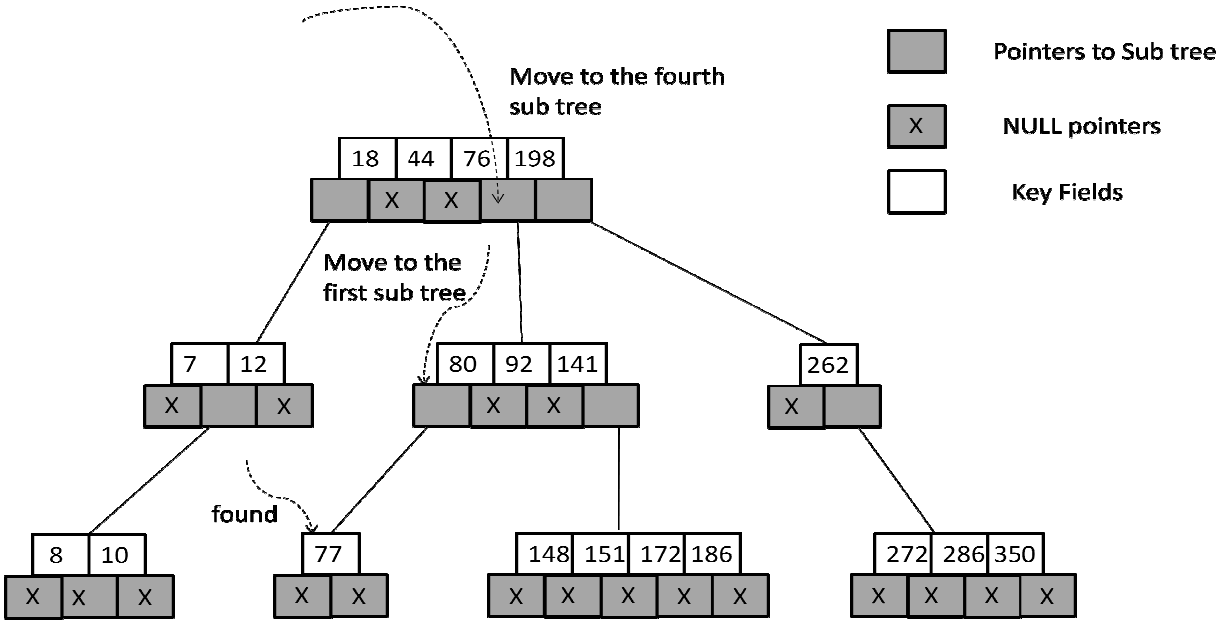
The structure of a node of an m-Way tree is given below:

```
struct node
{
   int count;
   int value[MAX + 1];
   struct node* child[MAX + 1];
};
```

- Here, **count** represents the number of children that a particular node has
- The values of a node stored in the array **value**
- The addresses of child nodes are stored in the **child** array
- The **MAX** macro signifies the maximum number of values that a particular node can contain

Searching in an m-Way search tree:

- Searching for a key in an m-Way search tree is similar to that of binary search tree
- To search for 77 in the 5-Way search tree, shown in the figure, we begin at the root & as 77> 76> 44> 18, move to the fourth sub-tree
- In the root node of the fourth sub-tree, 77< 80 & therefore we move to the first sub-tree of the node. Since 77 is available in the only node of this sub-tree, we claim 77 was successfully searched




```
// ------------------------------------------------------------------Searches value in the node
struct node* search(int val,  struct node* root,   int* pos)
{
    //------------------------------------------------------ if root is Null then return
   if (root == NULL)
      return NULL;
   else
   {

      //------------------------------------------------------------------ if node is found
      if (searchnode(val, root, pos))
         return root;
                              //---------------------- if not then search in child nodes
      else
         return search(val, root->child[*pos],  pos);
   }
}
```

```
//------------------------------------------------------------------------------ Searches the node
int searchnode(int val,  struct node* n,  int* pos)
{
   //---------------------------------------- if val is less than node->value[1]
   if (val < n->value[1])
   {
      *pos = 0;
      return 0;
   }
            //------------------------------------------------------------------ if the val is greater
   else
    {
      *pos = n->count;

                  //--------------------------------------- check in the child array for correct position
      while ( (val < n->value[*pos]) && (*pos > 1))
         (*pos)--;

      if (val == n->value[*pos])
         return 1;
      else
         return 0;
   }
}
```

search():

- The function **search()** receives three parameters
- The first parameter is the value to be searched, second is the address of the node from where the search is to be performed and third is the address of a variable that is used to store the position of the value once found
- Initially a condition is checked whether the address of the node being searched is NULL
- If it is, then simply a NULL value is returned
- Otherwise, a function **searchnode()** is called which actually searches the given value
- If the search is successful the address of the node in which the value is found is returned
- If the search is unsuccessful then a recursive call is made to the **search()** function for the child of the current node

searchnode():

- The function **searchnode()** receives three parameters
- The first parameter is the value that is to be searched
- The second parameter is the address of the node in which the search is to be performed and third is a pointer **pos** that holds the address of a variable in which the position of the value that once found is stored
- This function returns a value 0 if the search is unsuccessful and 1 if it is successful
- In this function initially it is checked whether the value that is to be searched is less than the very first value of the node
- If it is then it indicates that the value is not present in the current node. Hence, a value 0 is assigned in the variable that is pointed to by **pos** and 0 is returned, as the search is unsuccessful

2-3 Trees

2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element **leftVal**, it has two subtrees (children) namely **left** and **middle**. Whereas if a node contains two data elements **leftVal** and **rightVal**, it has three subtrees namely **left**, **middle** and **right**.

The main advantage with 2-3 trees is that it is balanced in nature as opposed to a binary search tree whose height in the worst case can be O(n). Due to this, the worst case time-complexity of operations such as search, insertion and deletion is O(log(n)) as the height of a 2-3 tree is O(log(n)).

Search: To search a key **K** in given 2-3 tree **T**, we follow the following procedure:

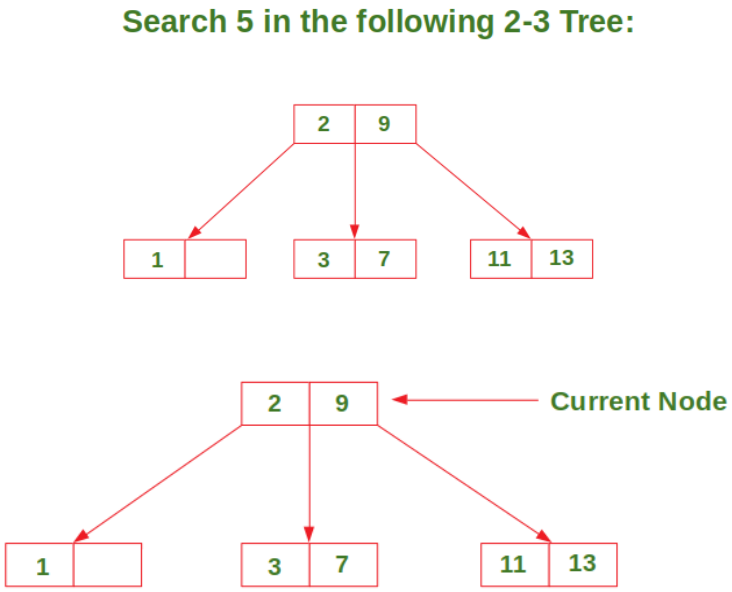
Base cases:

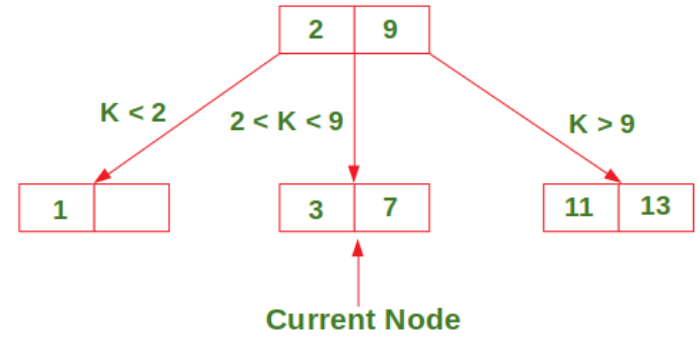
1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

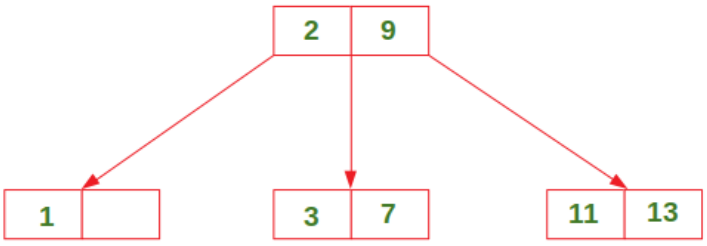
Recursive Calls:

1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.

Consider the following example:



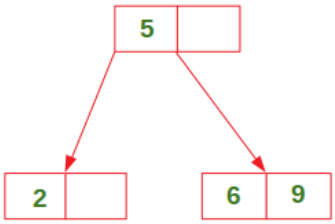




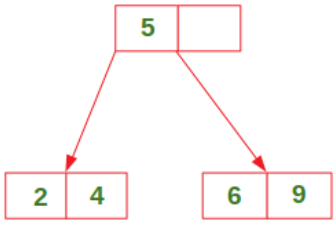
5 Not Found. Return False

Insertion: There are 3 possible cases in insertion which have been discussed below:
Case 1: Insert in a node with only one data element

Insert 4 in the following 2-3 Tree:



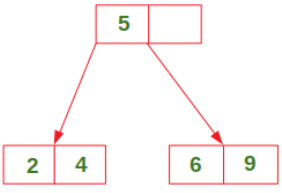
Initial



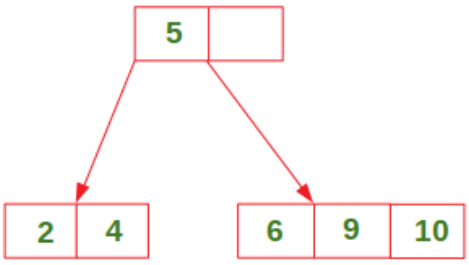
After Insertion

Case 2: Insert in a node with two data elements whose parent contains only one data element.

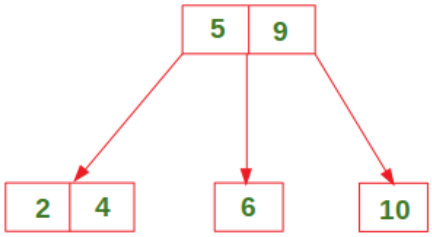
Insert 10 in the following 2-3 Tree:



Initial



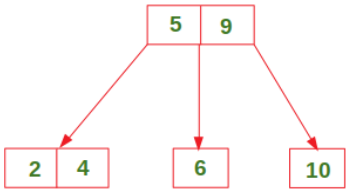
Temporary Node with 3 data elements



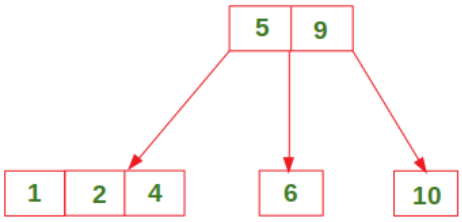
Move the middle element to
parent and split the current Node

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

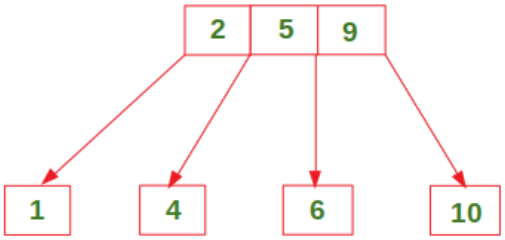
Insert 1 in the following 2-3 Tree:



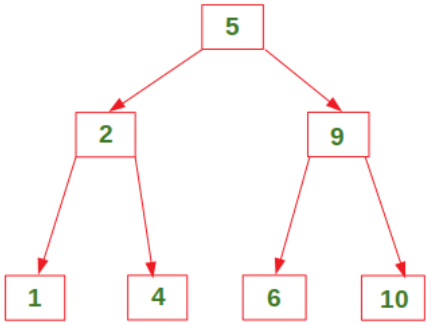
Initial



Temporary Node with 3 data elements



Move the middle element to the parent and split the current Node



Move the middle element to the parent and split the current Node

2-3 Tree

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

Here are the properties of a 2-3 tree:

1. each node has either one value or two values
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

Insertion Algorithm

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

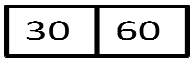
1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

Example 1) Insert the following sequence of numbers 60, 30, 90, 10, 50, 80, 100, 20, 40, 70, 39, 38, 37, 36, 35, 34, 33, 32 in a 2-3 Tree

Insert (60)

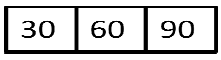
Inserting a node with value 60 to an empty 2-3 tree creates root node as well as leaf node. Each node can contain a maximum of 2 values and a maximum of 3 children.

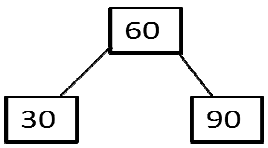
Insert (30)



Inserting value 30. As there is a space in the leaf node we can store one more value and 30 is placed in the same node. The values are stored in ascending order of their values.

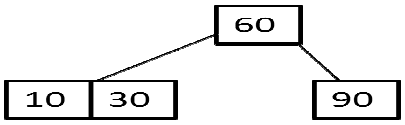
Insert (90)





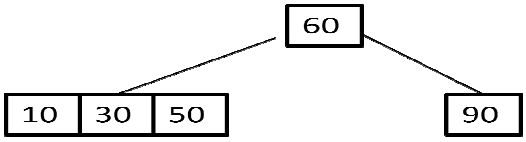
Inserting 90. While inserting 90 in the node already had 2 values. 90 is placed in temporarily and the node is splitted into two nodes and the middle value is promoted to parent level. Thus 60 becomes root node. 30 becomes left node of 60 and 90 becomes right node of 60.

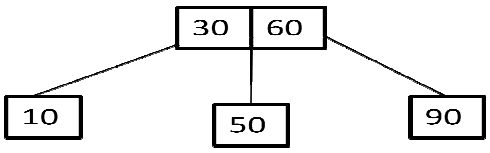
Insert (10)



To insert value 10 we start searching with root node to find the leaf node to insert value 10. As 10 is less than 60 we go to left sub tree. The left leaf node contains only one value and it can accommodate another value. So, 10 is stored there in ascending order.

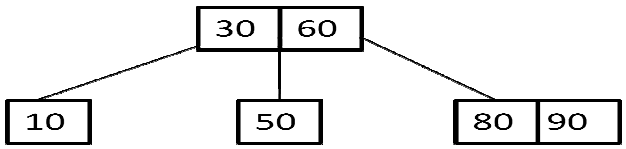
Insert (50)





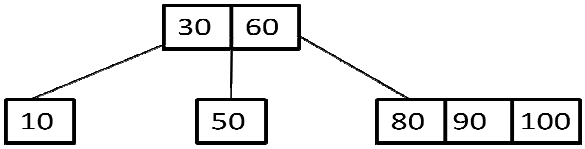
To insert value 50 we start searching with root node to find the leaf node to insert value 50. As 50<60 search ends at node containing two value 10 and 30. 50 is supposed to be placed at this node but there is no space. The node is splitted into two and the middle value is promoted to its parent. 30 is moved and stored along with 60. 10 is in left sub tree and 50 is in its right subtree.

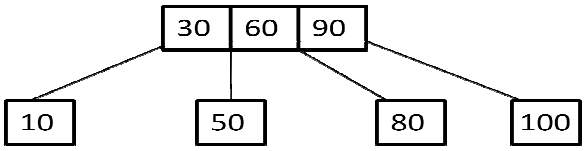
Insert (80)

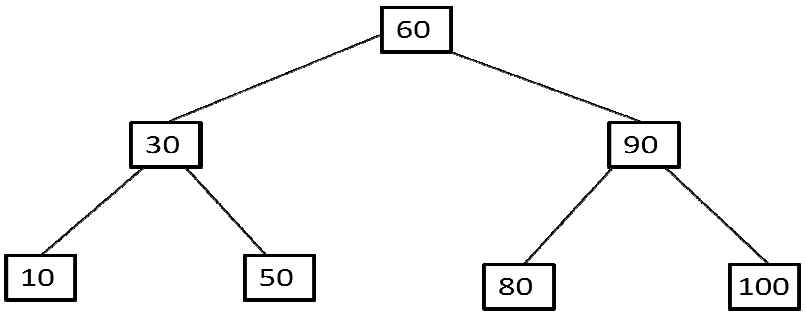


To insert value 80 we start searching with root node to find the leaf node to insert value 80. As 80 > 60 we go to right sub tree. The right leaf node contains only one value and it can accommodate another value. So, 80 is stored there in ascending order.

Insert (100)



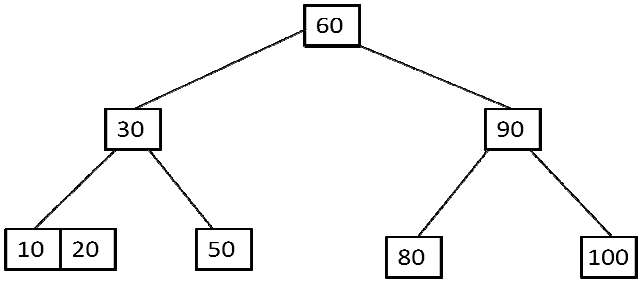




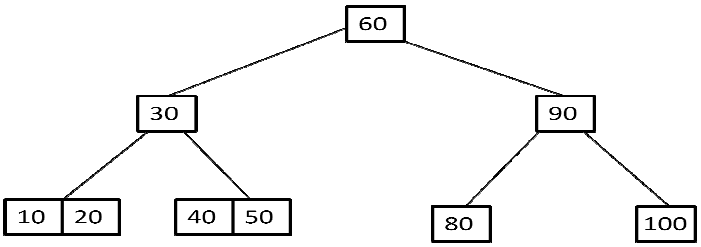
To insert value 100 we start searching with root node to find the leaf node to insert value 100. As 100>60 we go to right sub tree and the search ends at node containing two values (80,90). 100 is supposed to be placed at this node but there is no space. The node is splitted into two and the middle value is promoted to its parent. 90 is promoted to parent node.

Now the parent node is overflowing as it is already contains two values. Again it is also splitted and the middle value is promoted as its parent node.

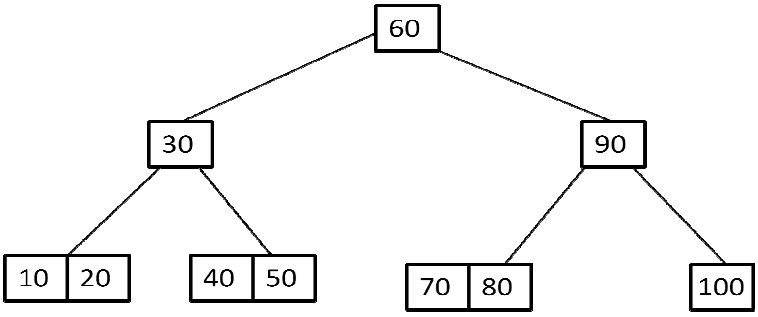
Insert(20)



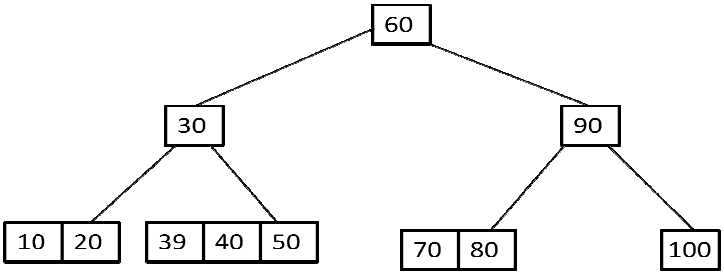
Insert(40)

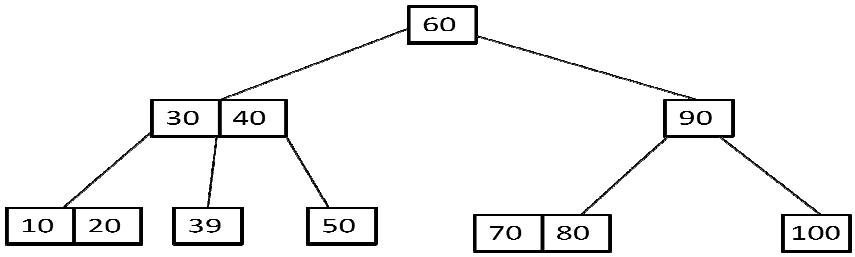


Insert(70)

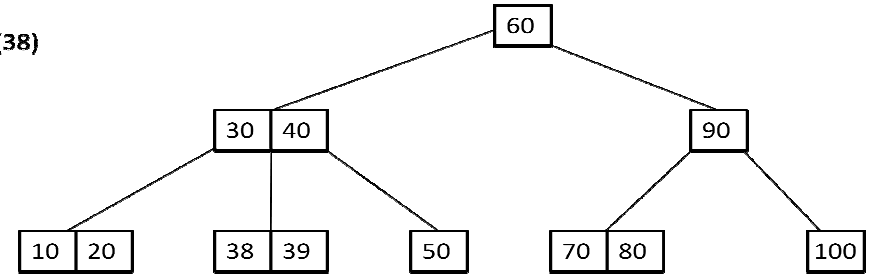


Insert(39)

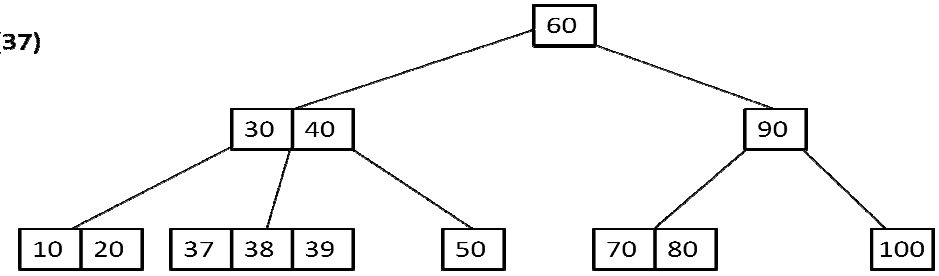


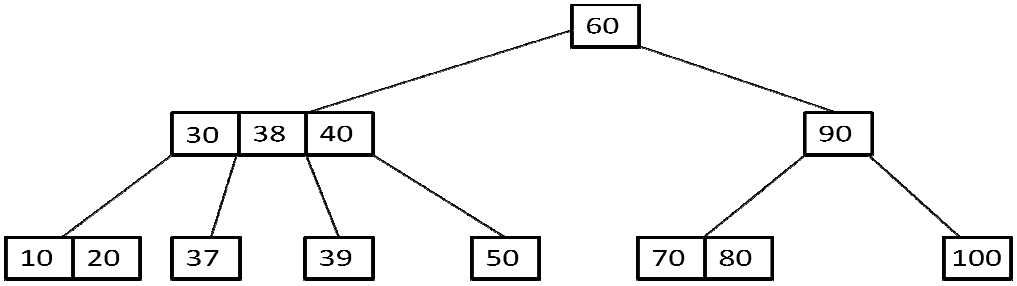


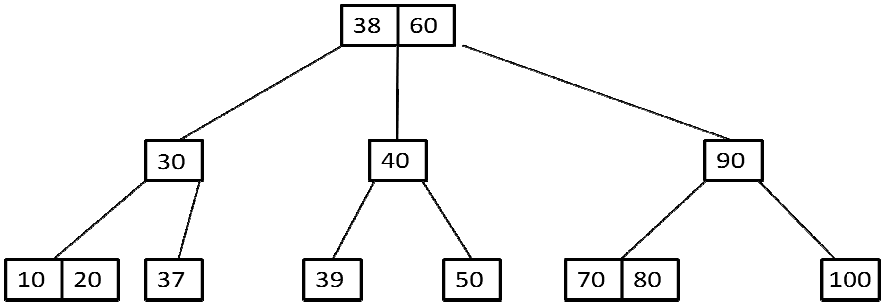
Insert(38)



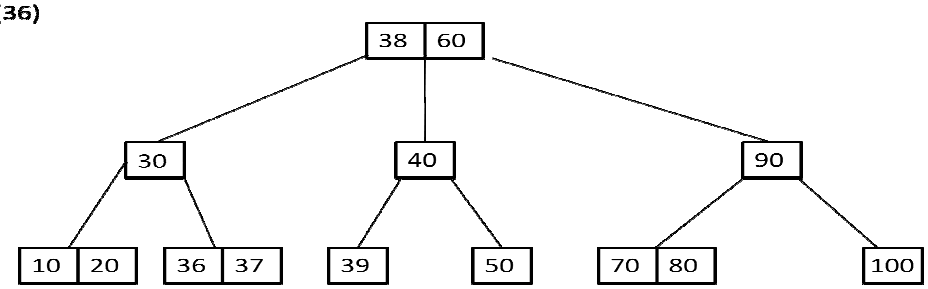
Insert(37)



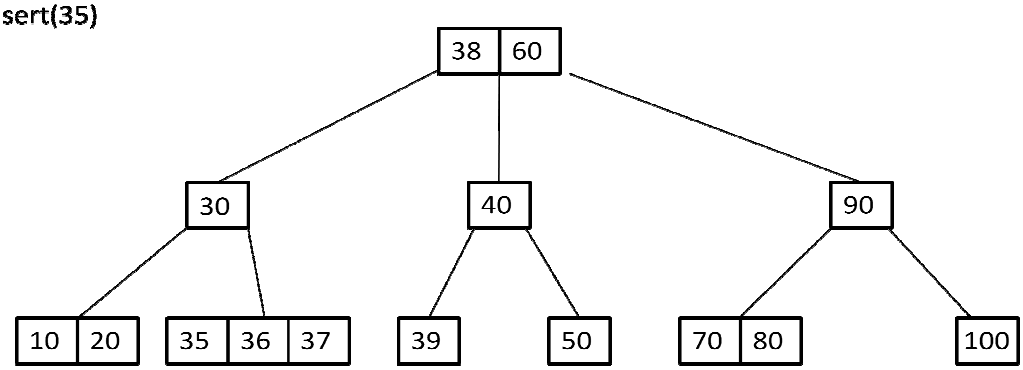


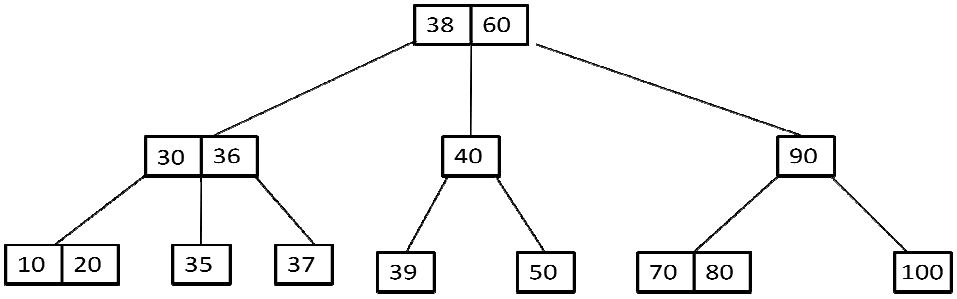


Insert(36)

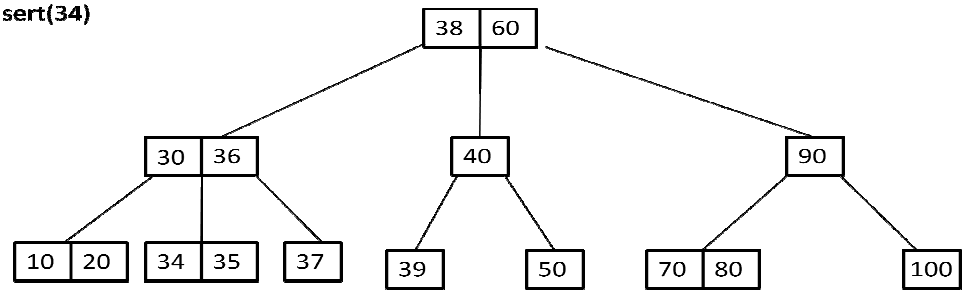


Insert(35)

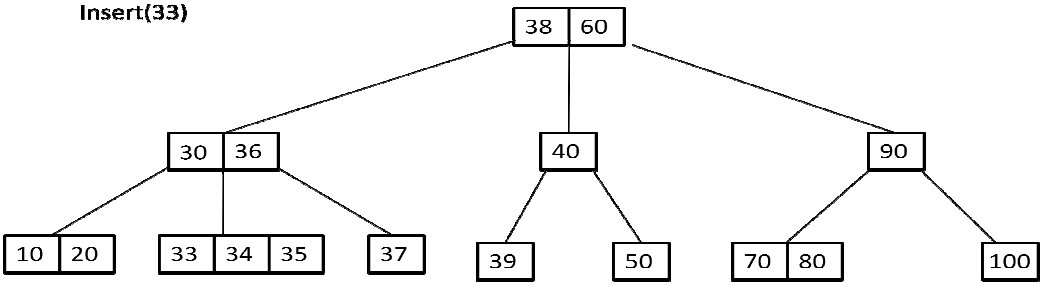




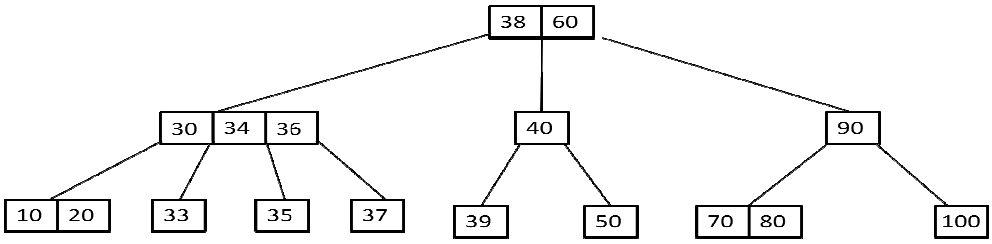
Insert(34)



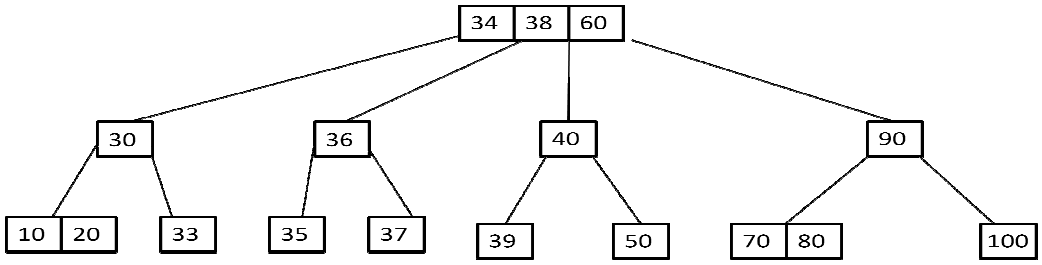
Insert(33)



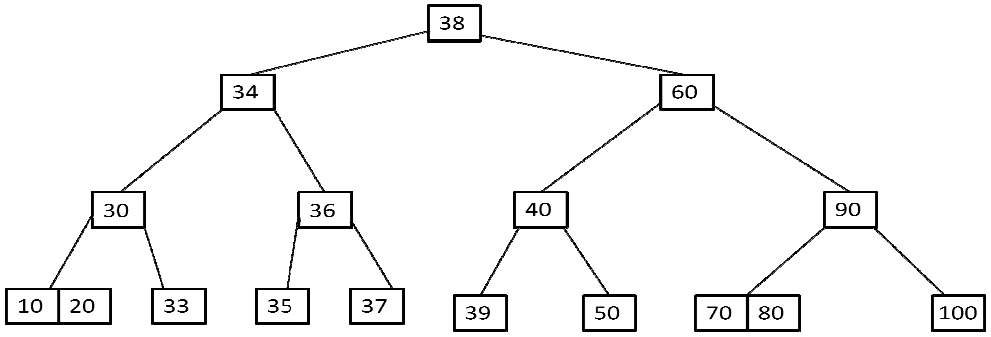
To insert value 33 we start with root. We search for the leaf node where we can insert 33. We find leaf node where it contains two nodes 34 and 35. Placing 33 in this node splits the node into two and the middle value is promoted to its parent.



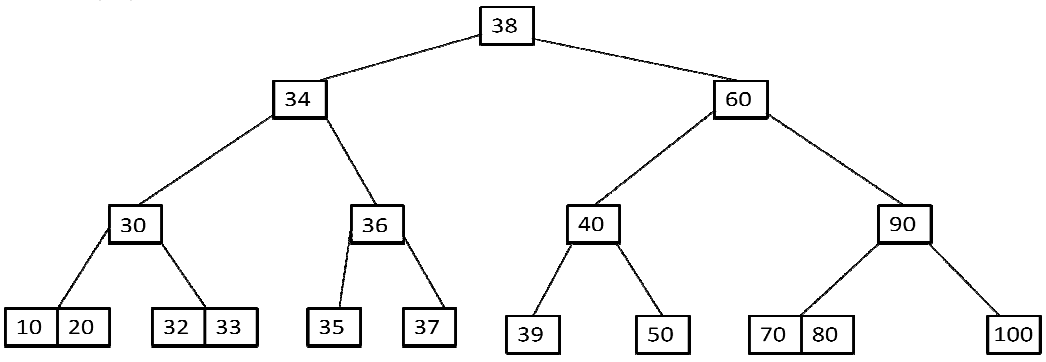
Parent node already contains two nodes and adding this value the node is splitted into two nodes and the middle value is promoted to its parent node.



Again Parent node already contains two nodes and adding this value the node is splitted into two nodes and the middle value is promoted to its parent node.



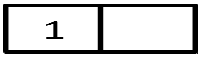
Insert(32)



To insert value 32 search starts with root node and finds leaf node to insert value 32.

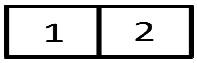
Example 2) Construct a B-Tree of order 3 by inserting numbers from 1 to 10 (2-3 Tree)

Insert 1



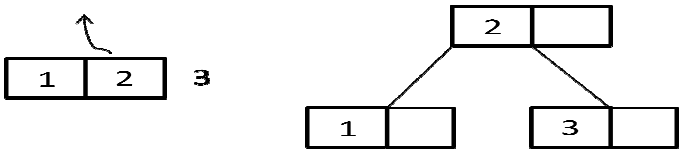
Inserting a node with value 1 to an empty 2-3 tree creates root node as well as leaf node. Each node can contain a maximum of 2 values and a maximum of 3 children.

Insert 2



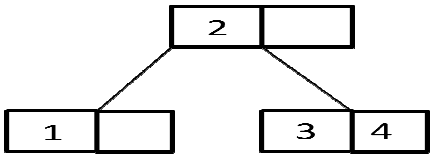
Inserting value 2. We start searching with root node to find the leaf node to insert value 2. As there is only one value we can store one more value and 2 is placed in the same node. The values are stored in ascending order of their values.

Insert 3



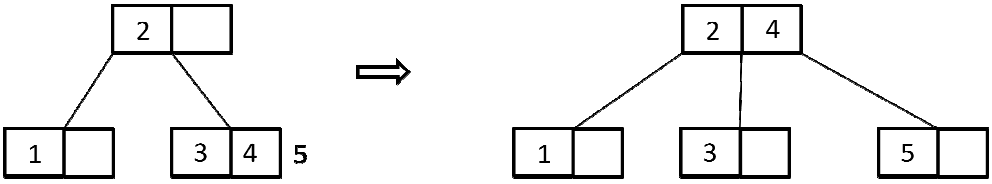
Inserting value 3. We start searching with root node to find the leaf node to insert value 3. As the node already has two values 3 is overflowing. The node is splitted into two and the middle value is promoted to parent.

Insert 4



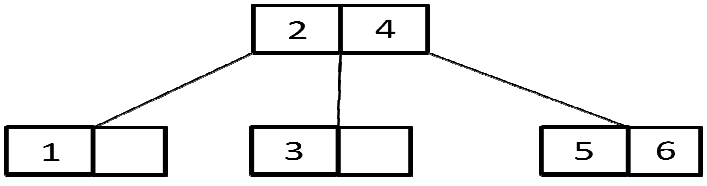
To insert value 4, We start searching with root node to find the leaf node to insert value 4. As 4>2 it selects right sub tree As the node has only one value 4 is placed in the node in ascending order.

Insert 5



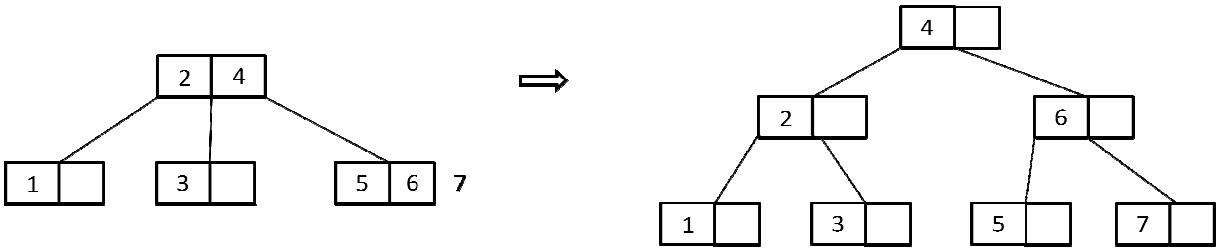
To insert value 5, We start searching from root node to find the leaf node to insert value 5. As 5>2 it selects right sub tree As the node already has two values 5 is overflowing. The node is splitted into two and the middle value is promoted to parent.

Insert 6



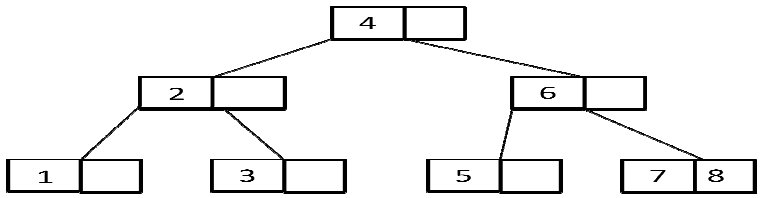
To insert value 6, We start searching with root node to find the leaf node to insert value 6. As 6>4 it selects right sub tree As the node has only one value 6 is placed in the node in ascending order.

Insert 7



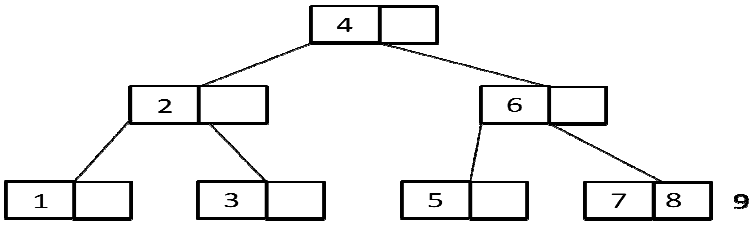
To insert value 7, We start searching from root node to find the leaf node to insert value 7. As 7>4 it selects right sub tree As the node already has two values 7 is overflowing. The node is splitted into two and the middle value is promoted to parent.

Insert 8

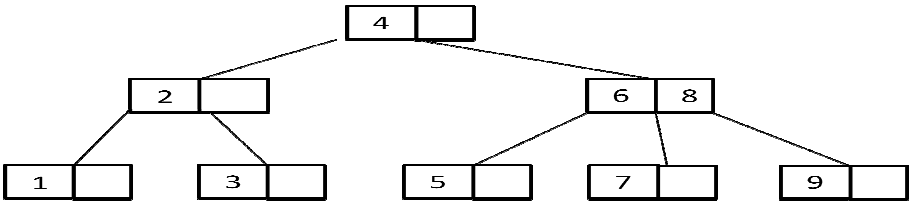


To insert value 8, We start searching with root node to find the leaf node to insert value 8. As 8>4 and 8>6 it selects right sub tree to the node 6. As the node has only one value 8 is placed in the node in ascending order.

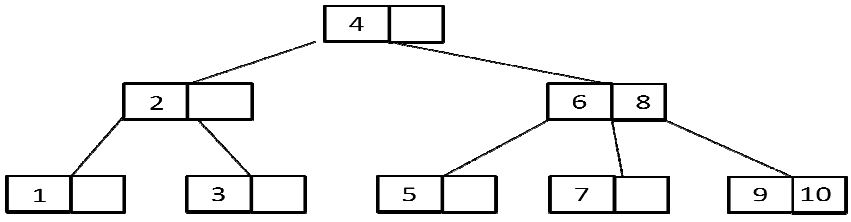
Insert 9



To insert value 9, We start searching from root node to find the leaf node to insert value 9. As 9>4 and 9>6 it selects right sub tree As the node already has two values 9 is overflowing. The node is splitted into two and the middle value is promoted to parent.



Insert 10



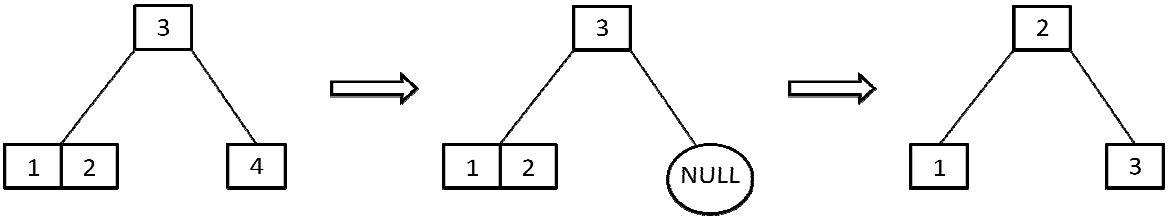
To insert value 10, We start searching with root node to find the leaf node to insert value 10. As 10>4 and 10>8 it selects right sub tree to the node 6,8. As the node has only one value 10 is placed in the node in ascending order.

Deleting a node from B-Tree (2-3 Tree)

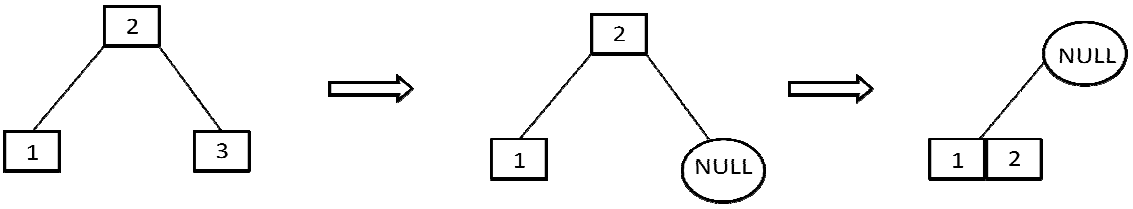
Deletion Algorithm :: Deleting an Item I from a 2-3 Tree

1. Locate node n, which contains item I
2. If node n is not a leaf node , swap I with Inorder successor.
3. If leaf node contains another item, just delete I, else try to redistribute nodes from siblings-if not possible merge nodes.

Redistribution after deleting 4

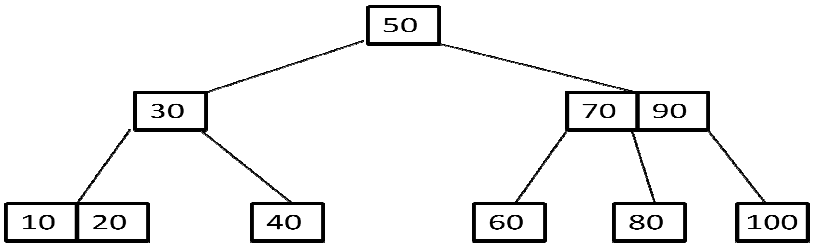


Merging after deleting 3

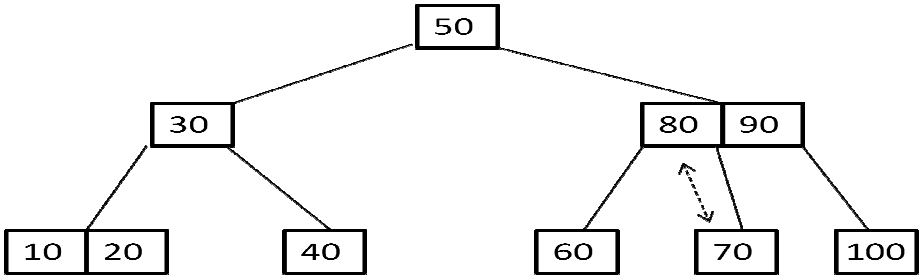


Example 1) Delete node with value 70 from the following 2-3 tree.

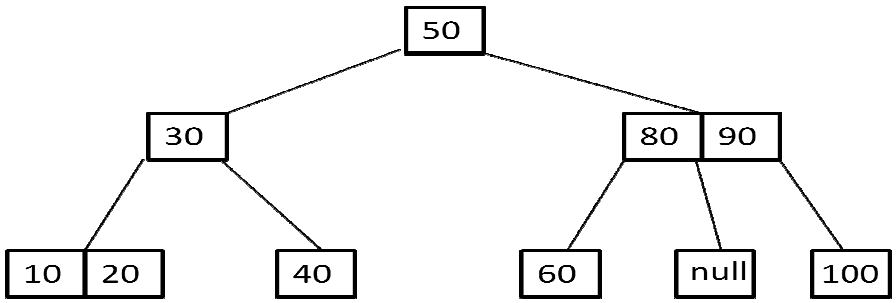
Delete 70



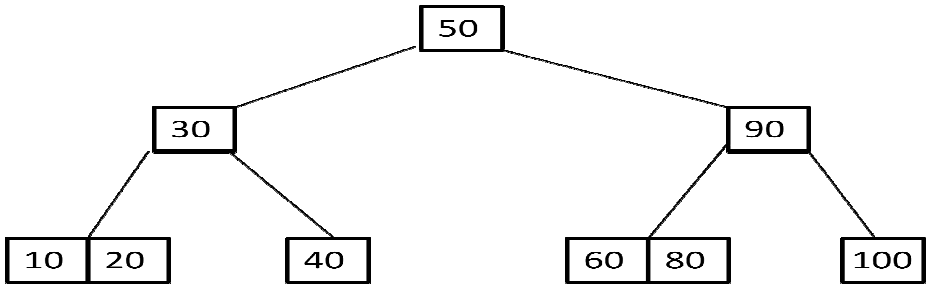
Identify node 70. Check for its inorder successor. 80 is its inorder successor. Swap node to be deleted with its inorder successor.



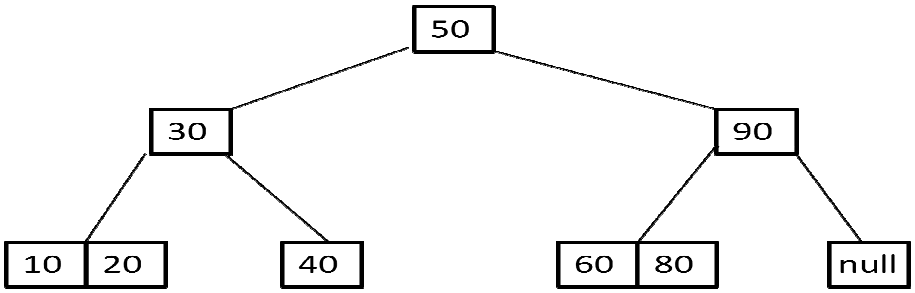
Delete the leaf node with value 70. Now we need to merge nodes after deleting leaf node.



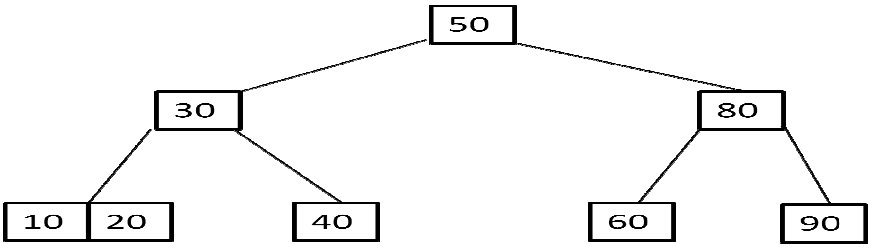
As there is no representation for the link between 80 and 90. Node 80 will be moving down and will be merged with node 60.



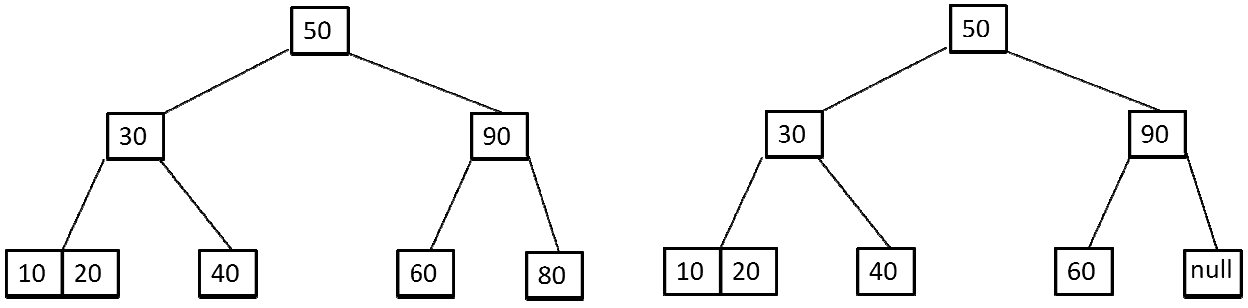
Delete 100 : identify the node 100. It is a leaf node. Delete node.



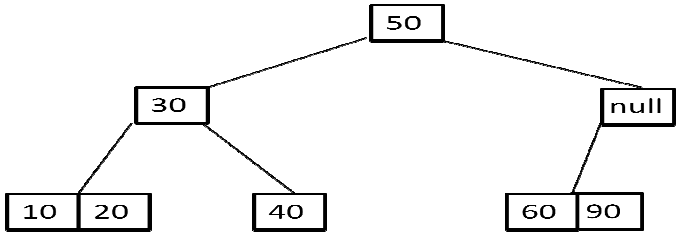
Readjusting nodes to balance the tree. Node 80 will be moving upwards and node 90 will be moved downwards.



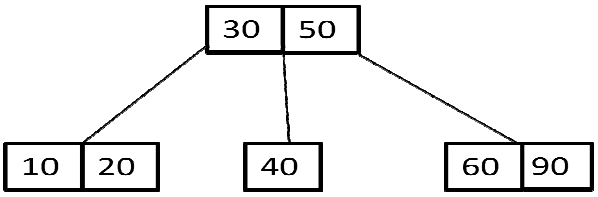
Delete node 80 : identify node 80. It is not a leaf node. Find inorder successor of node 80. Node 90 is inorder successor of 80. Node 90 is a leaf node.. swap node 80 and its inorder successor.



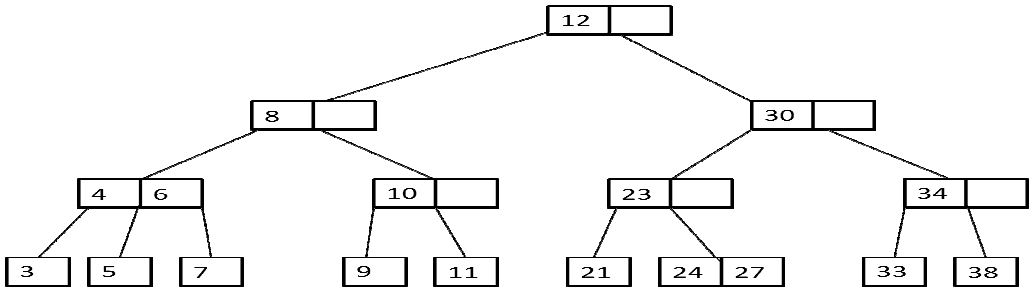
As there is no representation to the right sub tree of node 90. Node 90 will move down and will be merged with node 60 to keep the tree balanced.



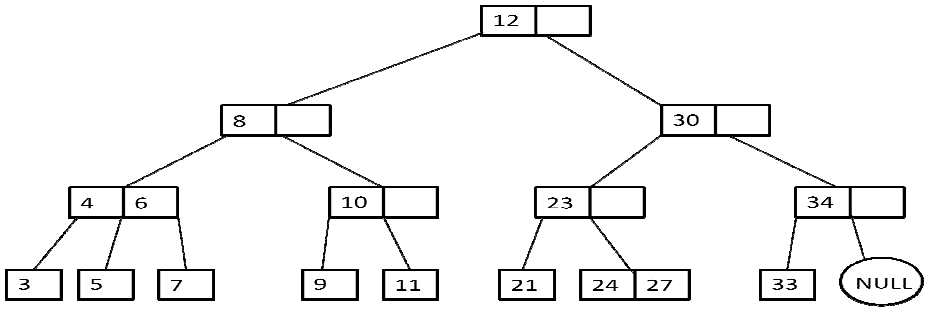
Now right of 50 is null but it is not leaf node. 50 will move down and will be merged with 30



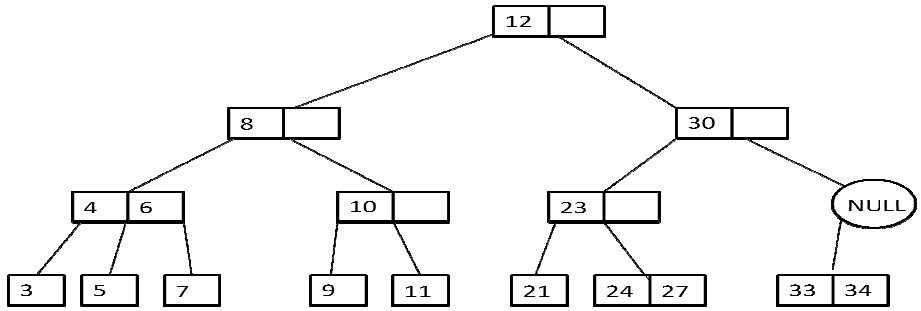
Example 2) Delete 38, 5, 8 from the following 2-3 trees and show the resulting 2-3 tree after every deletion operation.



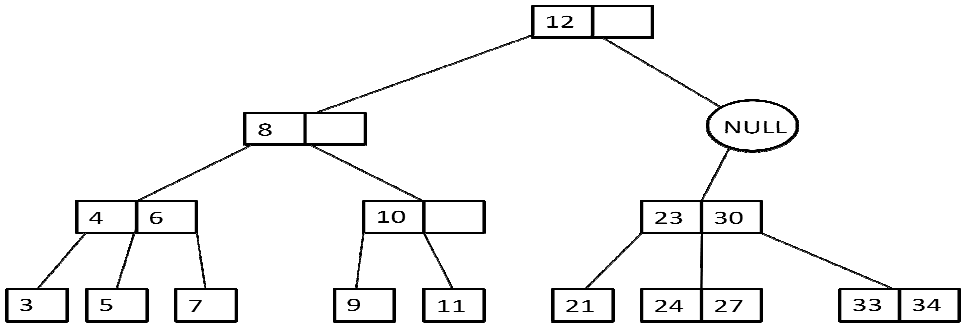
Delete 38 : Locate node with value 38. It is leaf node and it is the only value in that node. Delete the node. As 38 is deleted the right of 34 has no reference so, redistribute siblings or merge nodes.



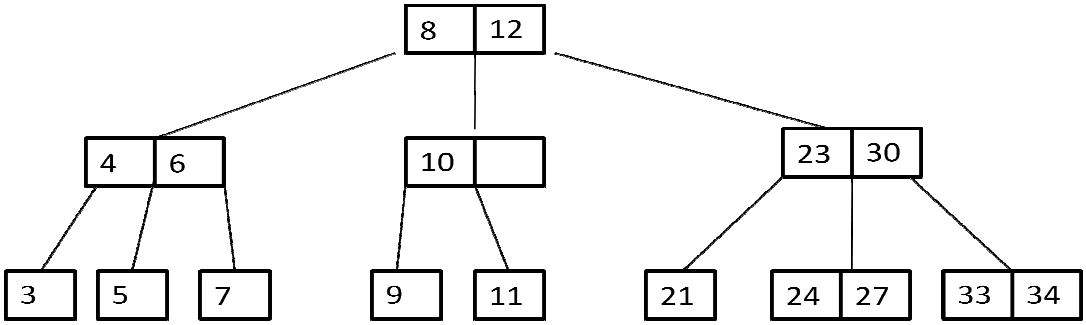
34 will go down and merges with 33.



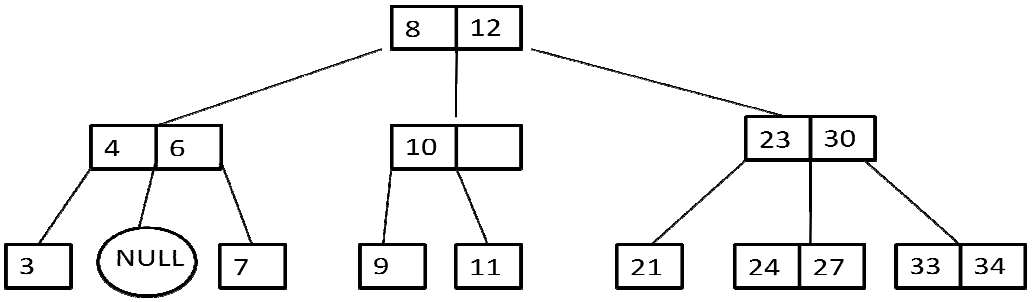
Node 30 will go down and will be merged with 23.



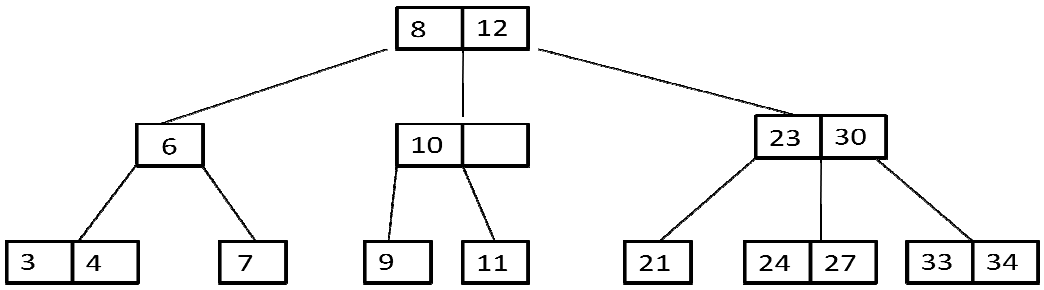
Node 12 will go down and will be merged with 8



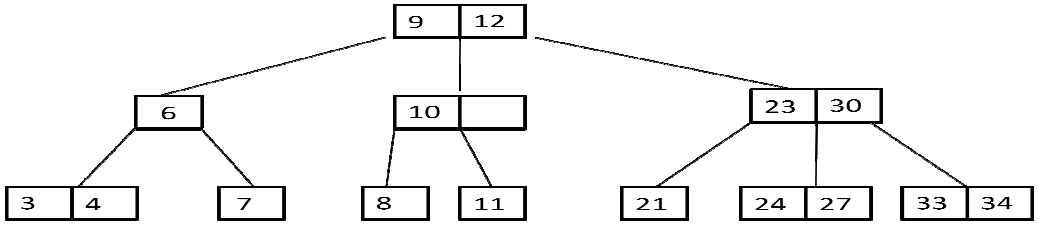
Delete 5 ; identify node with value 5. It is a leaf node. There are no other items in that node. Delete causes the link points to NULL



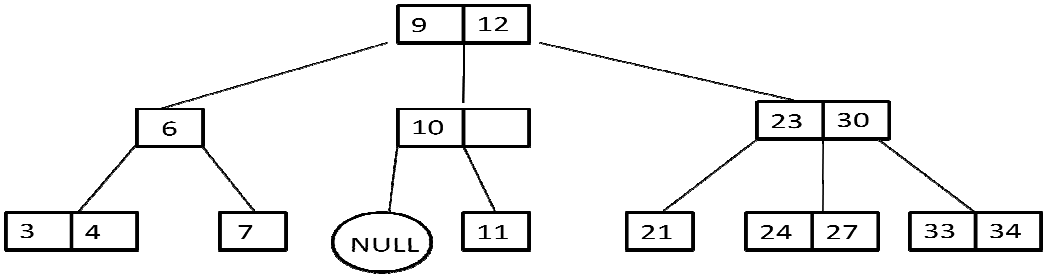
Middle of 4,6 has null link so 4 will go down



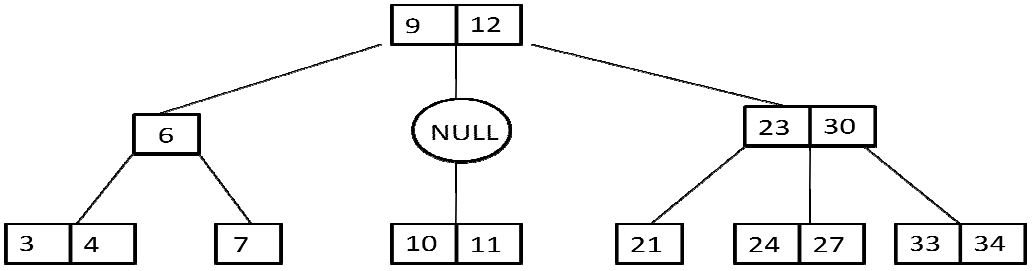
Delete 8 ; identify node with value 8. It is root node not a leaf node. Swap with its inorder successor. 9 is inorder successor of 8.



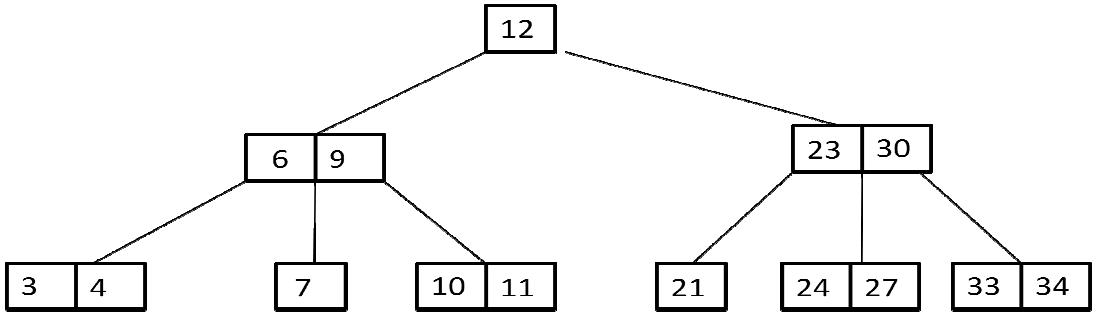
Deleting node 8 causes a NULL node pointing to left of 10.



Value 10 will godown and merges with 11

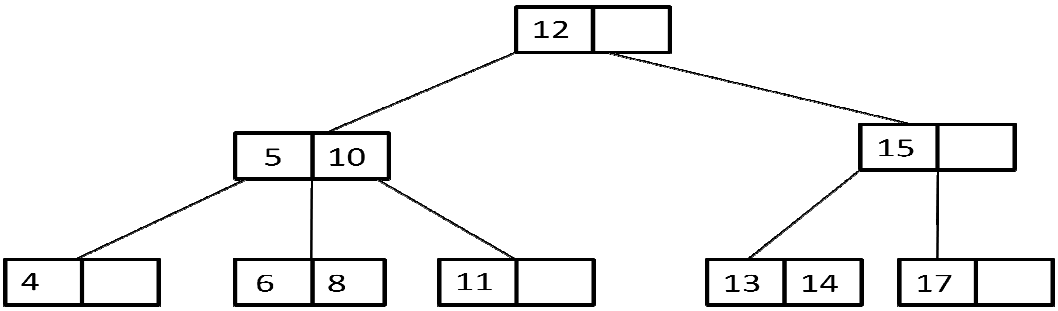


Value 9 will go down and merges with 6

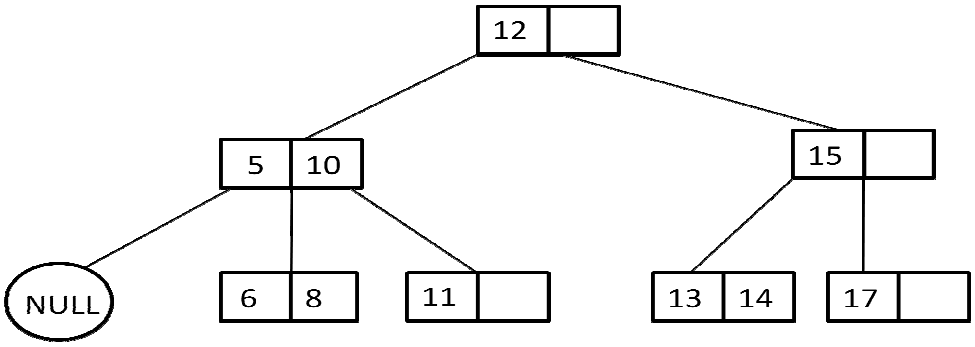


Resultant 2-3 tree after deletion

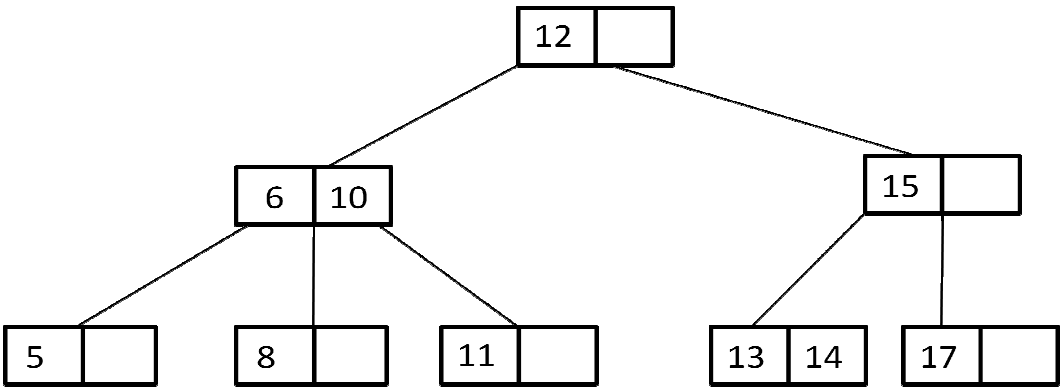
Example 3) Delete 4 from the following 2-3 tree.



Locate node with value 4. It is a leaf node. There is only one value in the node. Deleting causes left of 5 points to a NULL node.



Redistributing siblings. 6 will be promoted to parent and 5 will go down



The resultant 2-3 tree after deletion.

B-Trees

An extension of a multiway search tree of order m is a **B-tree of order m**. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order m is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.
2. Each nonroot and each nonleaf node have at most m nonempty children and at least m/2 nonempty children.
3. The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

Searching a B-tree

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

Insertion into a B-tree

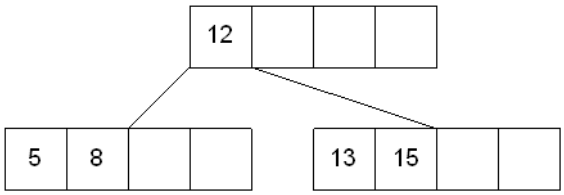
The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

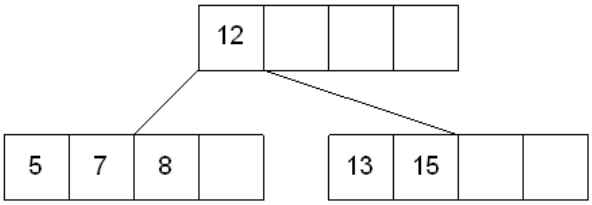
1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:

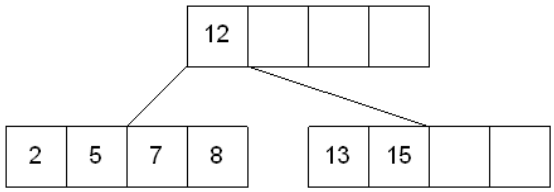


Case 2: The leaf in which a key is to be placed is full

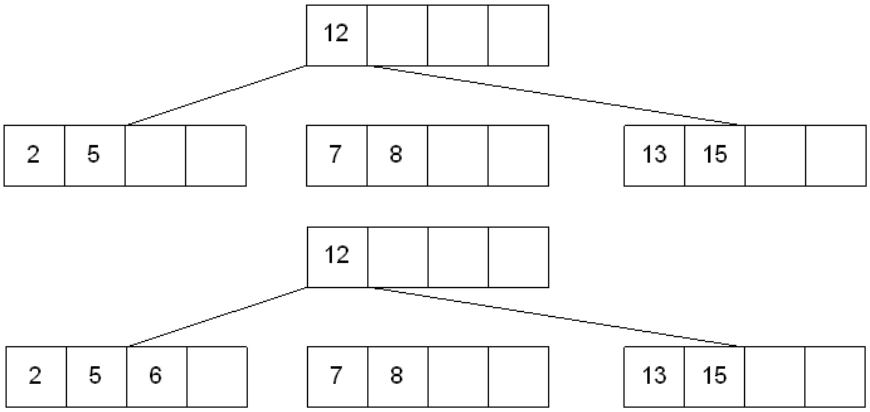
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

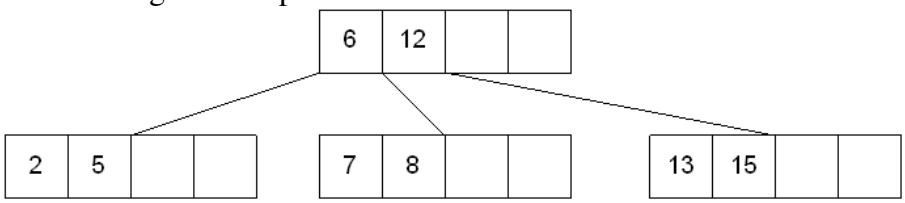
Insert 6 into the following B-tree:



results in a split of the first leaf node:



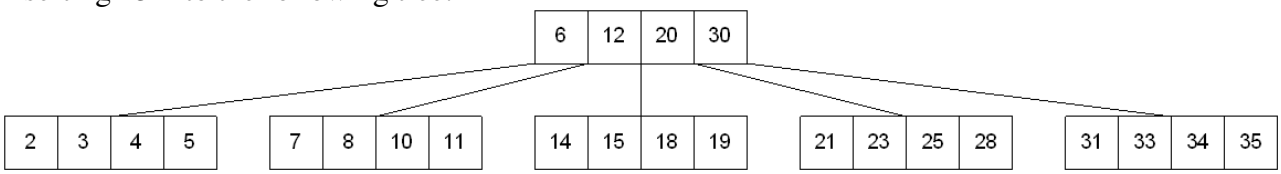
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



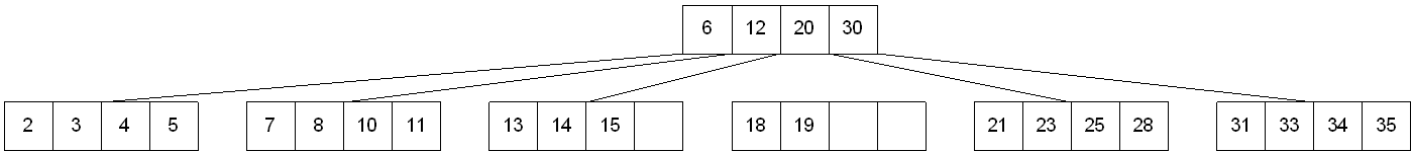
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

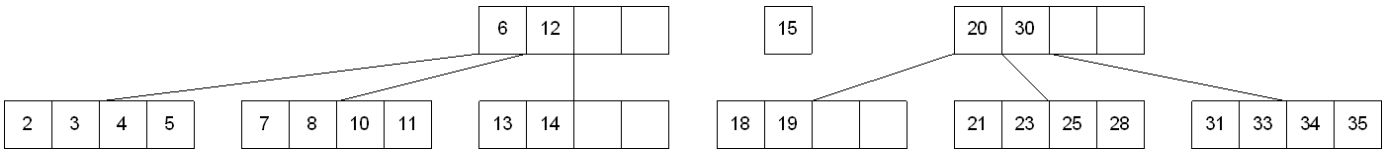
Inserting 13 into the following tree:



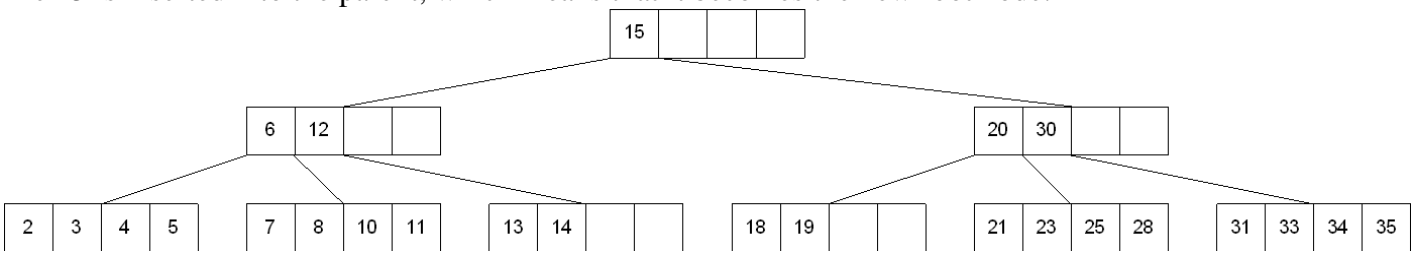
Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



Deleting from a B-tree

As usual, this is the hardest of the processes to apply. The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

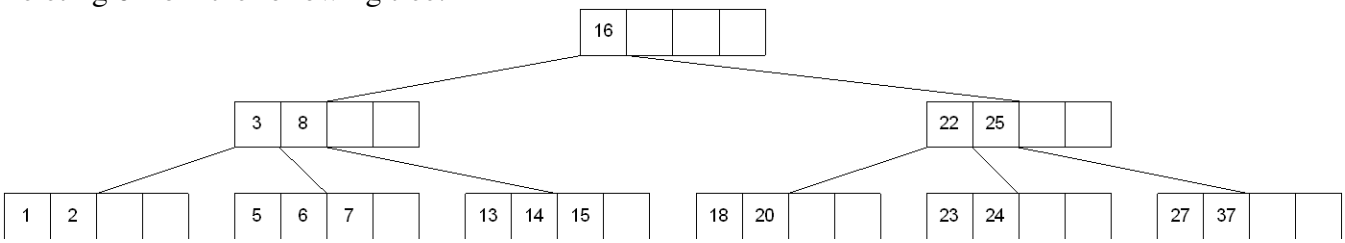
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

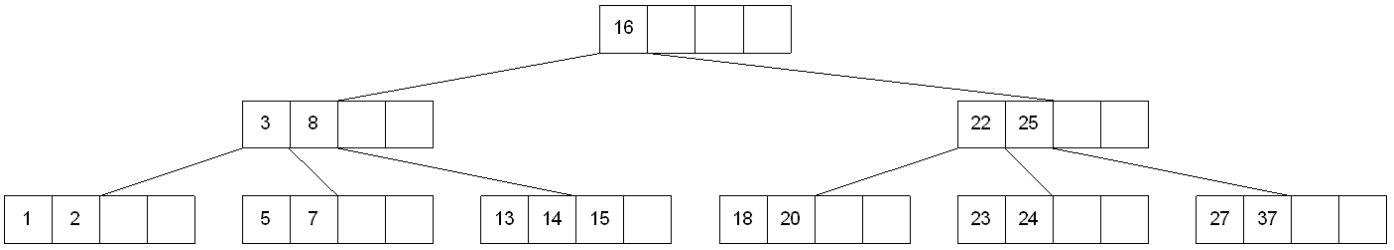
Case 1: Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

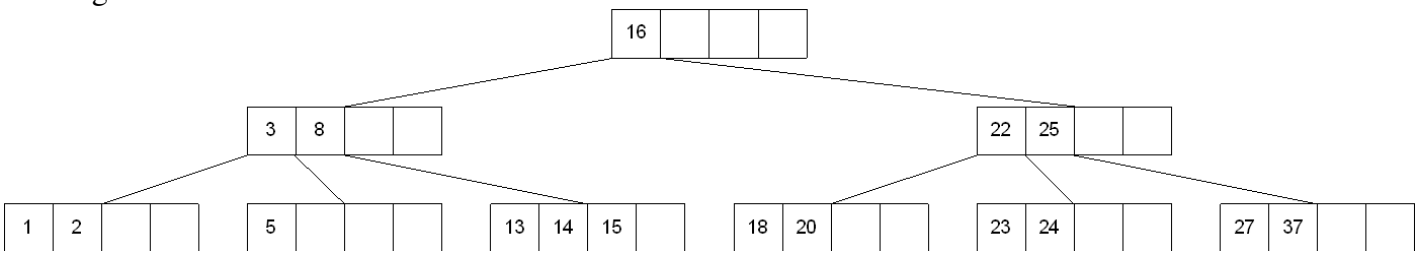


results in:

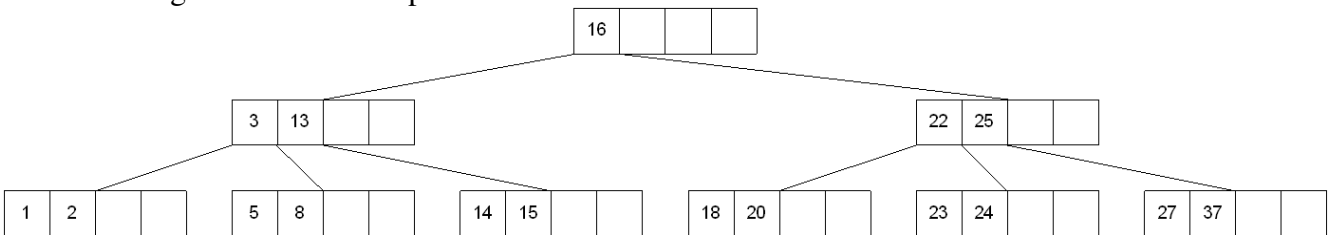


1b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

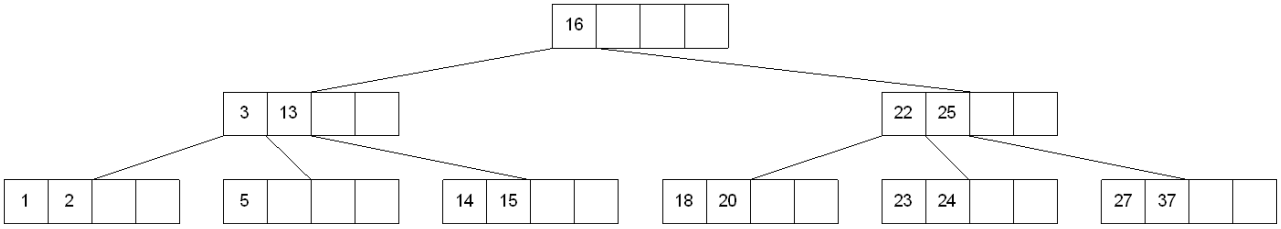
Deleting 7 from the tree above results in:



1b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

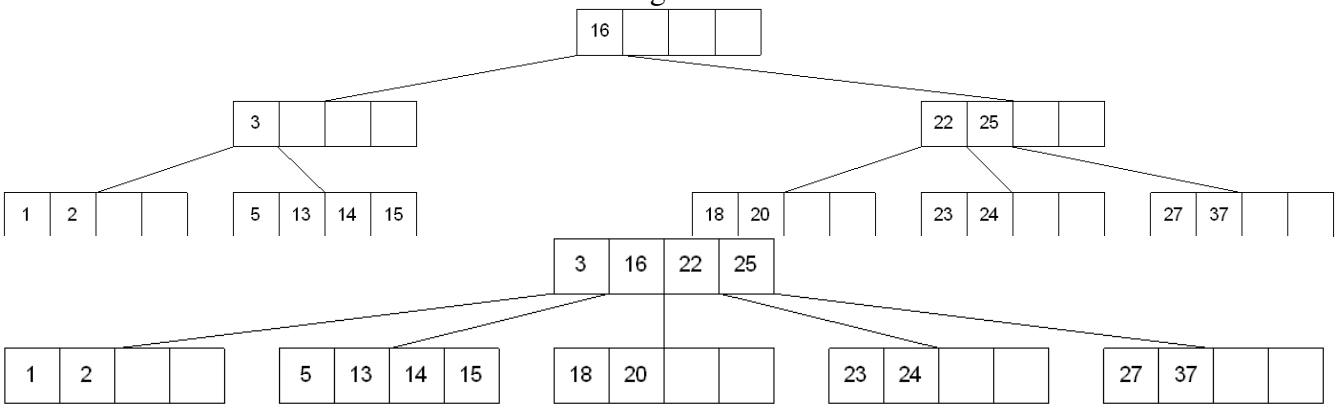


Now delete 8 from the tree:



1b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step 1b-2 until the minimum requirement is met or the root of the tree is reached.

Special Case for 1b-2: When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will become the new root for the B-tree. Both the sibling and the old root will be discarded.

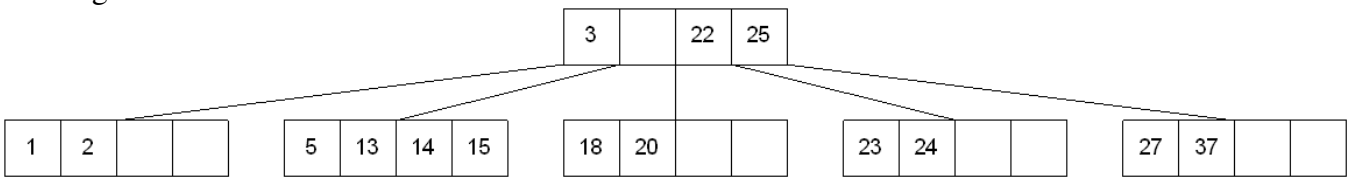


Case 2: Deletion from a non-leaf

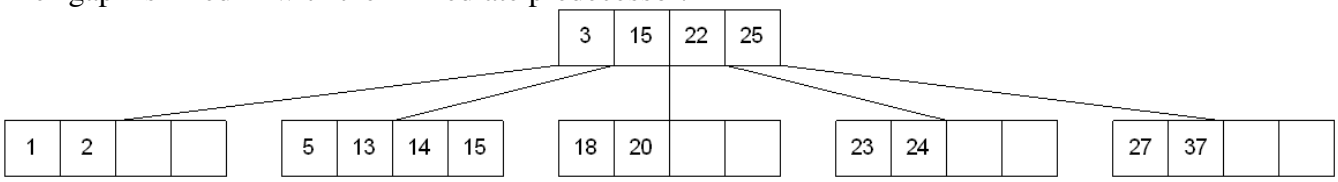
This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

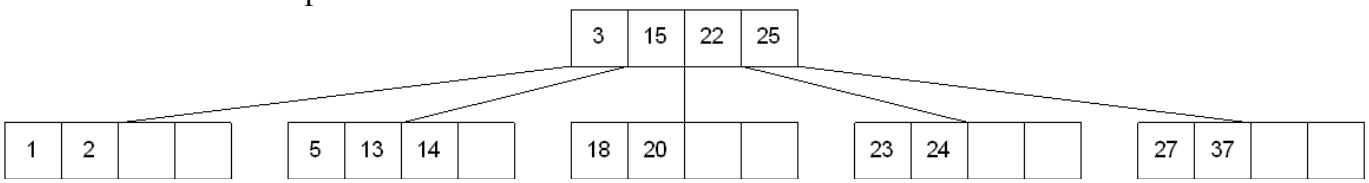
Deleting 16 from the tree above results in:



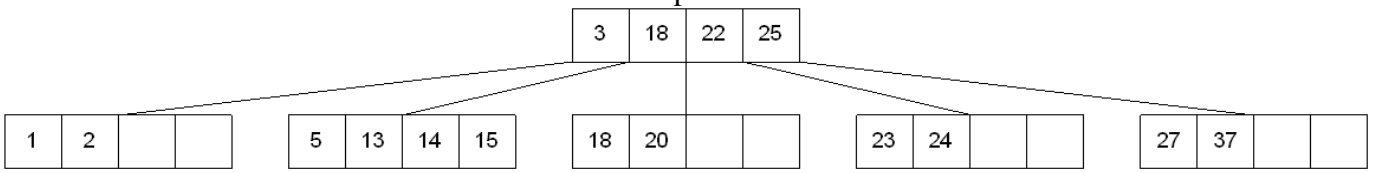
The "gap" is filled in with the immediate predecessor:



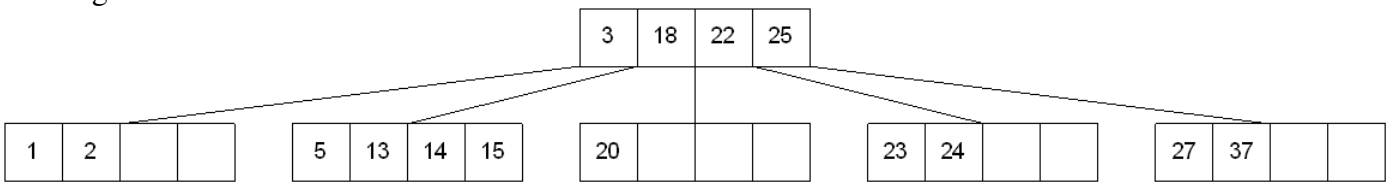
and then the immediate predecessor is deleted:



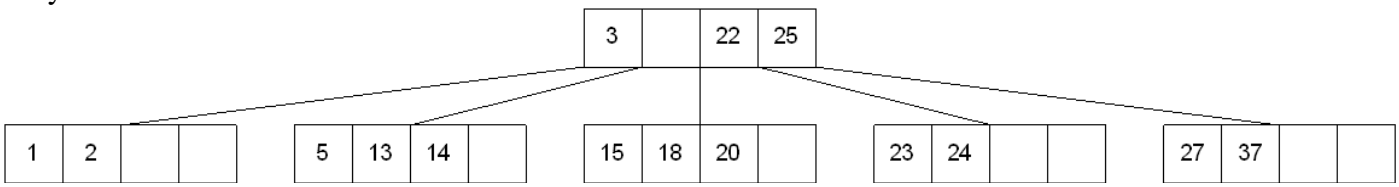
If the immediate successor had been chosen as the replacement:



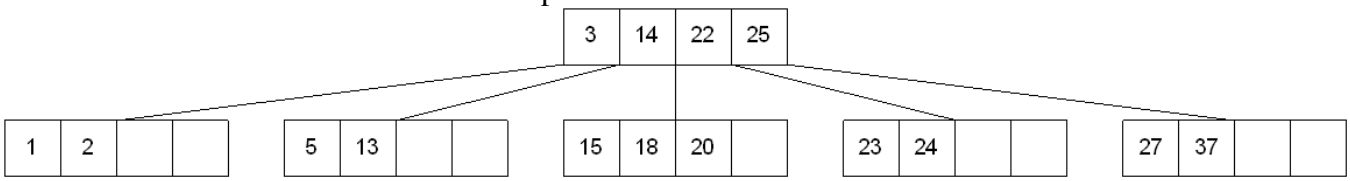
Deleting the successor results in:



The vales in the left sibling are combined with the separator key (18) and the remaining values.
They are divided between the 2 nodes:

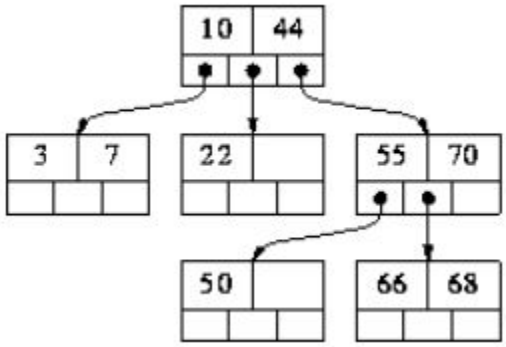


and then the middle value is moved to the parent:



What is a multiway search tree? Explain with an example.

- A **multiway tree** is a tree that can have more than two children
- A **multiway tree of order m (or an m-way tree**) is one in which a tree can have m children.
- An m-way search tree is a m-way tree in which:
  a) Each node has m children and m-1 key fields
  b) The keys in each node are in ascending order.
  c) The keys in the first i children are smaller than the ithith key
  d) The keys in the last m-i children are larger than the ithith key



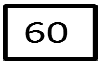
- In a binary search tree, m=2. So it has one value and two sub trees.
- The figure above is a m-way search tree of order 3.
- M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update.
- However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance.
- In m-way search tree, each sub-tree is also a m-way search tree and follows the same rules.
- An extension of a multiway search tree of order m is a **B-tree of order m**.
- This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

2-3-4 Tree

Example 1) Insert 60, 30, 10, 20, 50, 40, 70, 80, 15, 90 and 100 in the 2-3-4 tree.

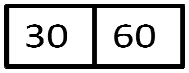
2-3 tree means every node may contain 2 values and 3 children. 2-3-4 tree means node may contain 2 values and 3 children or node may contain 3 values and 4 children.

Insert 60



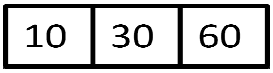
Inserting node with value 60 into an empty 2-3-4 tree. The node becomes root node.

Insert 30



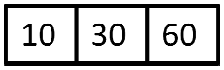
Inserting a node with value 30 in the node where there is a scope to add another value. In a 2-3-4 tree there is a possibility of having maximum 3 values in every node.

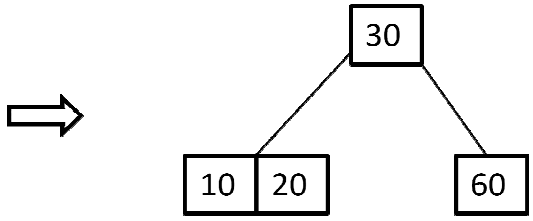
Insert 10



Inserting a node with value 10 in the node where there is a scope to add another value. In a 2-3-4 tree there is a possibility of having maximum 3 values in every node.

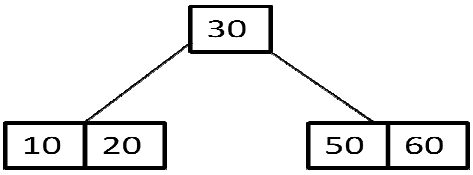
Insert 20





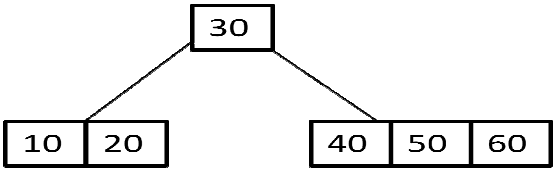
Inserting a node with value 20. We try to search for the leaf node in which we can insert value 20. The leaf node is already filled. So the node is splitted into two promoting the middle value to its parent. 30 is the mid value . it becomes parent and 20 is place in the left of 30.

Insert 50

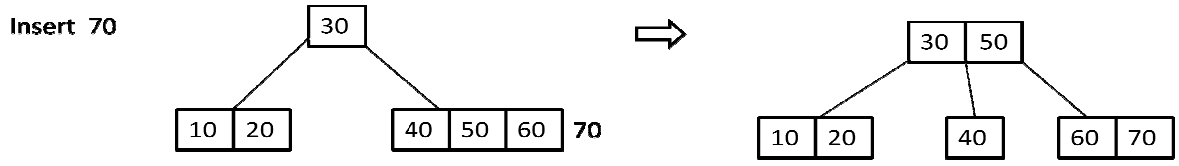


Inserting a node with value 50. We start our search with root node to find a leaf node in which we can insert the value 50. As 50>30 we choose right sub tree. Right leaf node contains only one value. So 50 is inserted to left of 60.

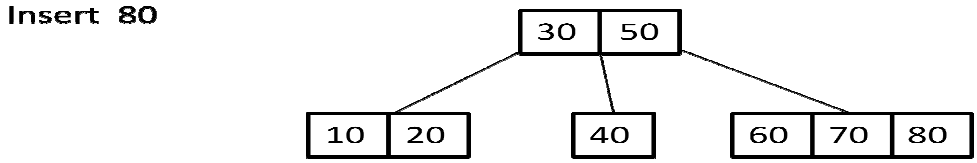
Insert 40



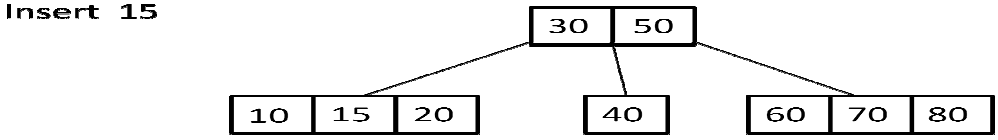
Inserting a node with value 40. We start our search with root node to find a leaf node in which we can insert the value 40. As 40>30 we choose right sub tree. Right leaf node contains two values and there is a space for one more node we can insert 40.



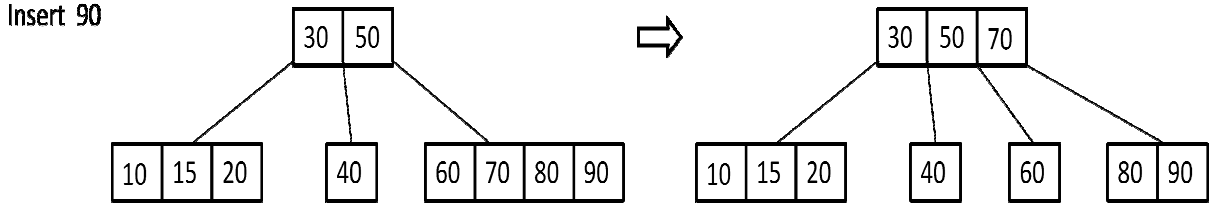
Inserting a node with value 70. We start our search with root node to find a leaf node in which we can insert the value 70. As 70>30 we choose right sub tree. Right leaf node contains three values and there is no space for one more value. So the node is splitted into two and the middle value is promoted to its parent and the value 70 is inserted in the node 60.



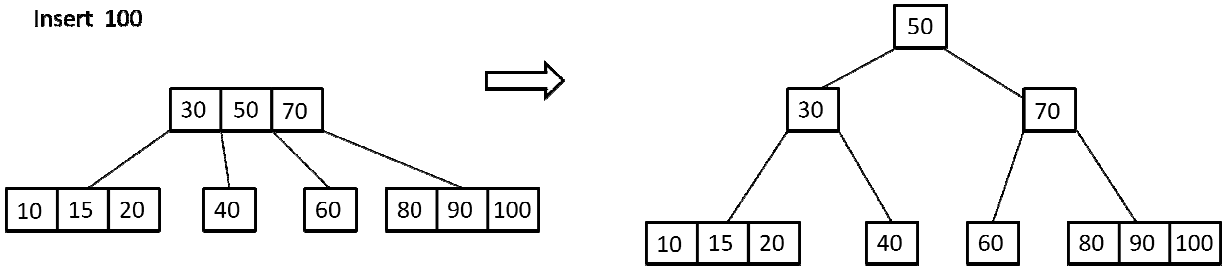
Inserting a node with value 80. We start our search with root node to find a leaf node in which we can insert the value 80. As 80>50 we choose right sub tree. Right leaf node contains two values and there is a space for one more node so, we can insert 80.



Inserting a node with value 15. We start our search with root node to find a leaf node in which we can insert the value 15. As 15<30 we choose left sub tree. left leaf node contains two values and there is a space for one more node so, we can insert 15.



Inserting a node with value 90. We start our search with root node to find a leaf node in which we can insert the value 90. As 90>50 we choose right sub tree. Right leaf node contains three values and there is no space for one more value. So the node is splitted into two and the middle value 70 is promoted to its parent and the value 90 is inserted in the node 80.

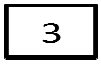


Inserting a node with value 100. We start our search with root node to find a leaf node in which we can insert the value 100. As 100>70 we choose right sub tree. Right leaf node contains two values and there is a space for one more value. So the value is inserted. And the parent node is splitted into two and the middle value is promoted to its parent/root.

Example 2) Insert 3, 1, 5, 4, 2, 9, 10. 8, 7, 6 in the empty 2-3-4 tree.

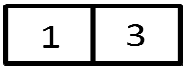
Assume that the 2-3-4 tree is empty

Insert 3



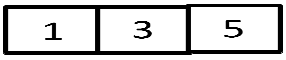
As we are trying to insert a node with value 3 in an empty 2-3-4 tree. The node with value 3 becomes its root node.

Insert 1



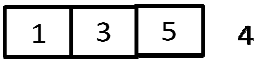
To insert 1 we search for appropriate node. There is only one node. Which can store 2 more values. So 1 added.

Insert 5

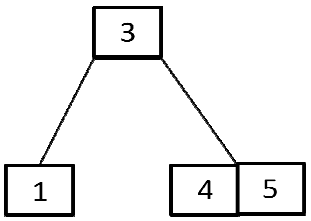


To insert 5 we search for appropriate node. There is only one node. Which can store 1 more value. So 5 added

Insert 4

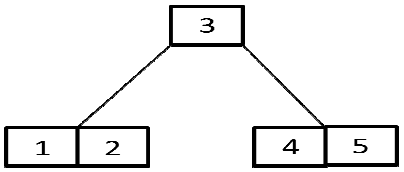


4



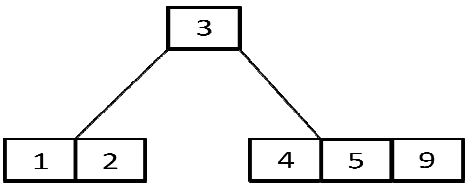
In a 2-3-4 tree node may contain a maximum of 3 values. When trying to insert another node, the node will be splitted into to and the middle will be promoted to its parent. New value will be added to the appropriate node.

Insert 2

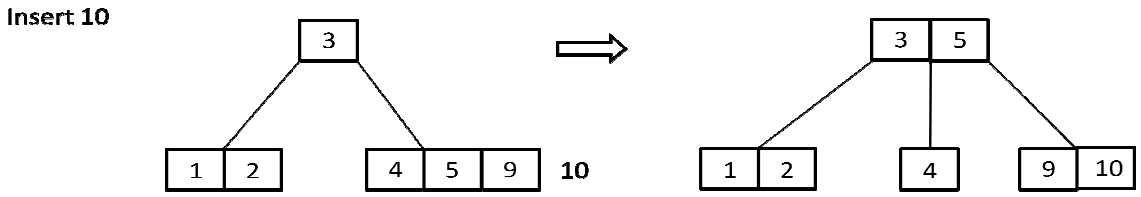


To insert value 2 search starts with root. 2 is less than 3. So take left child. Left child is leaf. There is some room to store 2. So 2 added to left leaf.

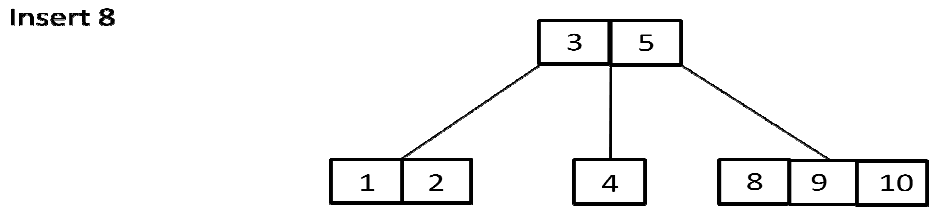
Insert 9



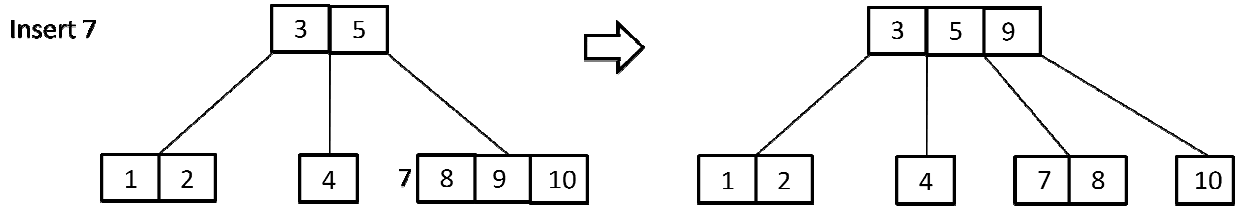
To insert value 9 search starts with root. 9 >3. So take right child. right child is leaf. There is some room to store 9. So 9 added to righht leaf.



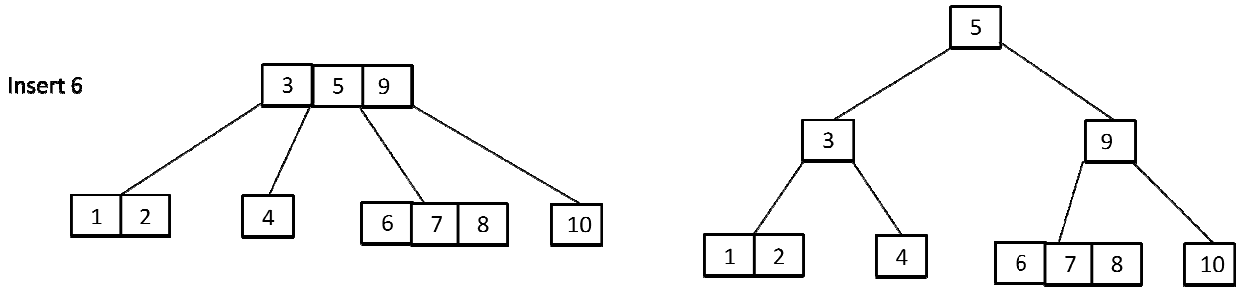
To insert value 10 search starts with root. 10 >3. So take right child. Right child is leaf. There is no room to store 10. So the node is splitted into 2 and the middle value is promoted to parent. As there is some free space in parent node 5 is added to parent/root. Node 10 is added to the leaf node adjacent to 9.



To insert 8 search starts with root. 8 > 5 so it takes right most sub tree. As the right most leaf has free space to add one more value 8 is added to it.



To insert value 7 search starts with root node. 7 > 5. So take right child. Right child is leaf. There is no room to store 7. So the node is splitted into 2 and the middle value is promoted to parent. As there is some free space in parent node and 9 is added to parent/root. Node 7 is added to the leaf node adjacent to 8.



To insert value 6 search starts with root node. 6 > 5. So take right child of 5. Right child of 5 is leaf. There is some free space to store 6. Node added.

B+ TREE

What is a B+ Tree?

A **B+ Tree** is primarily utilized for implementing dynamic indexing on multiple levels. Compared to B- Tree, the B+ Tree stores the data pointers only at the leaf nodes of the Tree, which makes search process more accurate and faster.

Rules for B+ Tree

Here are essential rules for B+ Tree.

- Leaves are used to store data records.
- Index vales stored in the internal nodes of the Tree.
- If a target key value is less than the internal node, then the point just to its left side is followed.
- If a target key value is greater than or equal to the internal node, then the point just to its right side is followed.
- The root has a minimum of two children.

Why use B+ Tree

Here, are reasons for using B+ Tree:

- Keys are primarily utilized to aid the search by directing to the proper Leaf.
- B+ Tree uses a "fill factor" to manage the increase and decrease in a tree.
- In B+ trees, numerous keys can easily be placed on the page of memory because they do not have the data associated with the interior nodes. Therefore, it will quickly access tree data that is on the leaf node.
- A comprehensive full scan of all the elements is a tree that needs just one linear pass because all the leaf nodes of a B+ tree are linked with each other.

B+ Tree vs. B Tree

Here, are the main differences between B+ Tree vs. B Tree

B + Tree	B Tree
Search keys can be repeated.	Search keys cannot be redundant.
Data is only saved on the leaf nodes.	Both leaf nodes and internal nodes can store data
Data stored on the leaf node makes the search more accurate and faster.	Searching is slow due to data stored on Leaf and internal nodes.
Deletion is not difficult as an element is only removed from a leaf node.	Deletion of elements is a complicated and time-consuming process.
Linked leaf nodes make the search efficient and quick.	You cannot link leaf nodes.

Search Operation

In B+ Tree, a search is one of the easiest procedures to execute and get fast and accurate results from it.

The following search algorithm is applicable:

- To find the required record, you need to execute the binary search on the available records in the Tree.
- In case of an exact match with the search key, the corresponding record is returned to the user.
- In case the exact key is not located by the search in the parent, current, or leaf node, then a "not found message" is displayed to the user.
- The search process can be re-run for better and more accurate results.

Search Operation Algorithm

1. Call the binary search method on the records in the B+ Tree.
2. If the search parameters match the exact key
   The accurate result is returned and displayed to the user
   Else, if the node being searched is the current and the exact key is not found by the algorithm
   Display the statement "Recordset cannot be found."

Output:

The matched record set against the exact key is displayed to the user; otherwise, a failed attempt is shown to the user.

Insert Operation

The following algorithm is applicable for the insert operation:

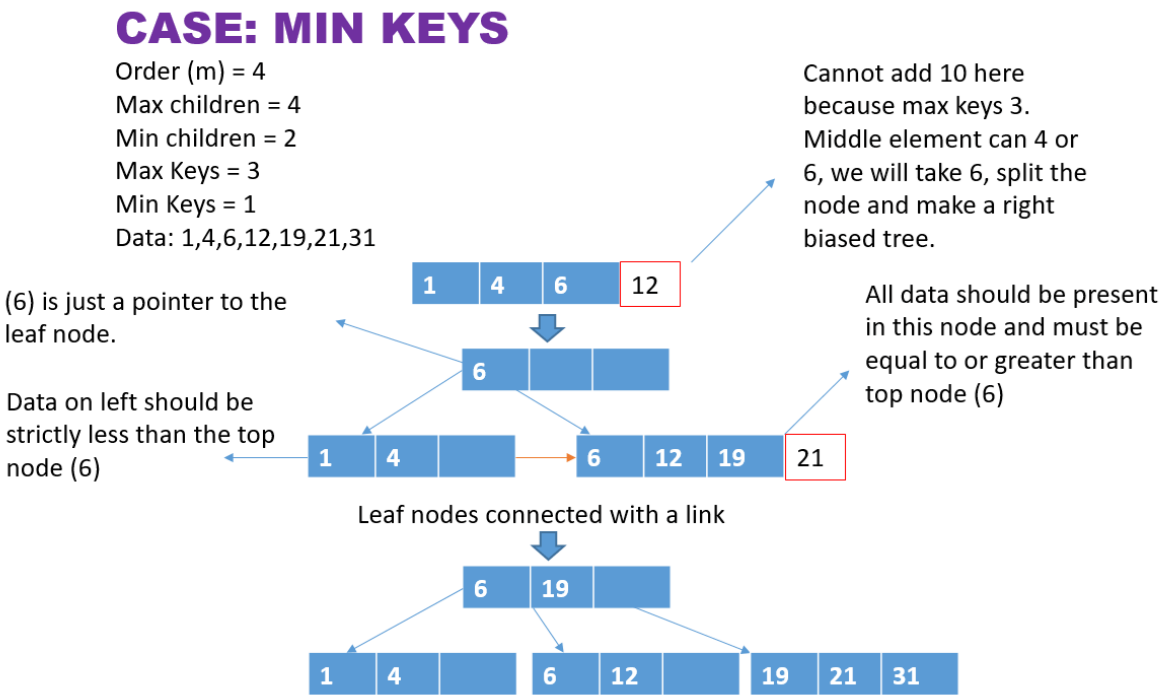
- 50 percent of the elements in the nodes are moved to a new leaf for storage.
- The parent of the new Leaf is linked accurately with the minimum key value and a new location in the Tree.
- Split the parent node into more locations in case it gets fully utilized.
- Now, for better results, the center key is associated with the top-level node of that Leaf.
- Until the top-level node is not found, keep on iterating the process explained in the above steps.

Insert Operation Algorithm

1. Even inserting at-least 1 entry into the leaf container does not make it full then add the record
2. Else, divide the node into more locations to fit more records.
   a. Assign a new leaf and transfer 50 percent of the node elements to a new placement in the tree
   b. The minimum key of the binary tree leaf and its new key address are associated with the top-level node.
   c. Divide the top-level node if it gets full of keys and addresses.
      i. Similarly, insert a key in the center of the top-level node in the hierarchy of the Tree.
   d. Continue to execute the above steps until a top-level node is found that does not need to be divided anymore.

3) Build a new top-level root node of 1 Key and 2 indicators.

Output:
The algorithm will determine the element and successfully insert it in the required leaf node.



The above B+ Tree sample example is explained in the steps below:

- Firstly, we have 3 nodes, and the first 3 elements, which are 1, 4, and 6, are added on appropriate locations in the nodes.
- The next value in the series of data is 12 that needs to be made part of the Tree.
- To achieve this, divide the node, add 6 as a pointer element.
- Now, a right-hierarchy of a tree is created, and remaining data values are adjusted accordingly by keeping in mind the applicable rules of equal to or greater than values against the key-value nodes on the right.

Delete Operation

The complexity of the delete procedure in the B+ Tree surpasses that of the insert and search functionality.

The following algorithm is applicable while deleting an element from the B+ Tree:

- Firstly, we need to locate a leaf entry in the Tree that is holding the key and pointer. , delete the leaf entry from the Tree if the Leaf fulfills the exact conditions of record deletion.
- In case the leaf node only meets the satisfactory factor of being half full, then the operation is completed; otherwise, the Leaf node has minimum entries and cannot be deleted.
- The other linked nodes on the right and left can vacate any entries then move them to the Leaf. If these criteria is not fulfilled, then they should combine the leaf node and its linked node in the tree hierarchy.
- Upon merging of leaf node with its neighbors on the right or left, entries of values in the leaf node or linked neighbor pointing to the top-level node are deleted.

Deletion Algorithm

STEP 1 Find leaf L containing (key, pointer) entry to delete

STEP 2 Remove entry from L

STEP 2a If L meets the "half full" criteria, then we're done.

STEP 2b Otherwise, L has too few data entries.

STEP 3 If L's right sibling can spare an entry, then move smallest entry in right sibling to L

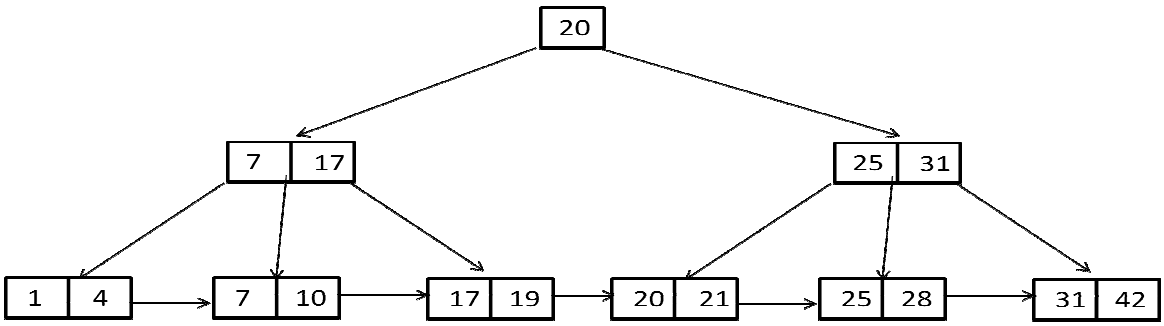
STEP 3a Else, if L's left sibling can spare an entry then move largest entry in left sibling to L

STEP 3b Else, merge L and a sibling

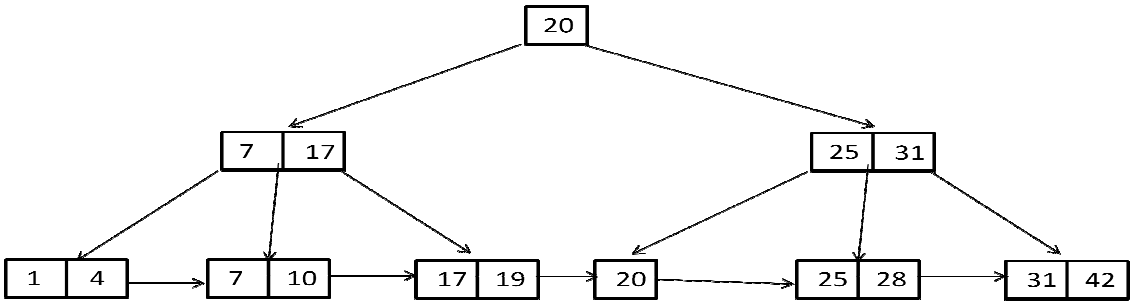
STEP 4 If merging, then recursively deletes the entry (pointing to L or sibling) from the parent.

STEP 5 Merge could propagate to root, decreasing height

Delete 21from the following B+ Tree

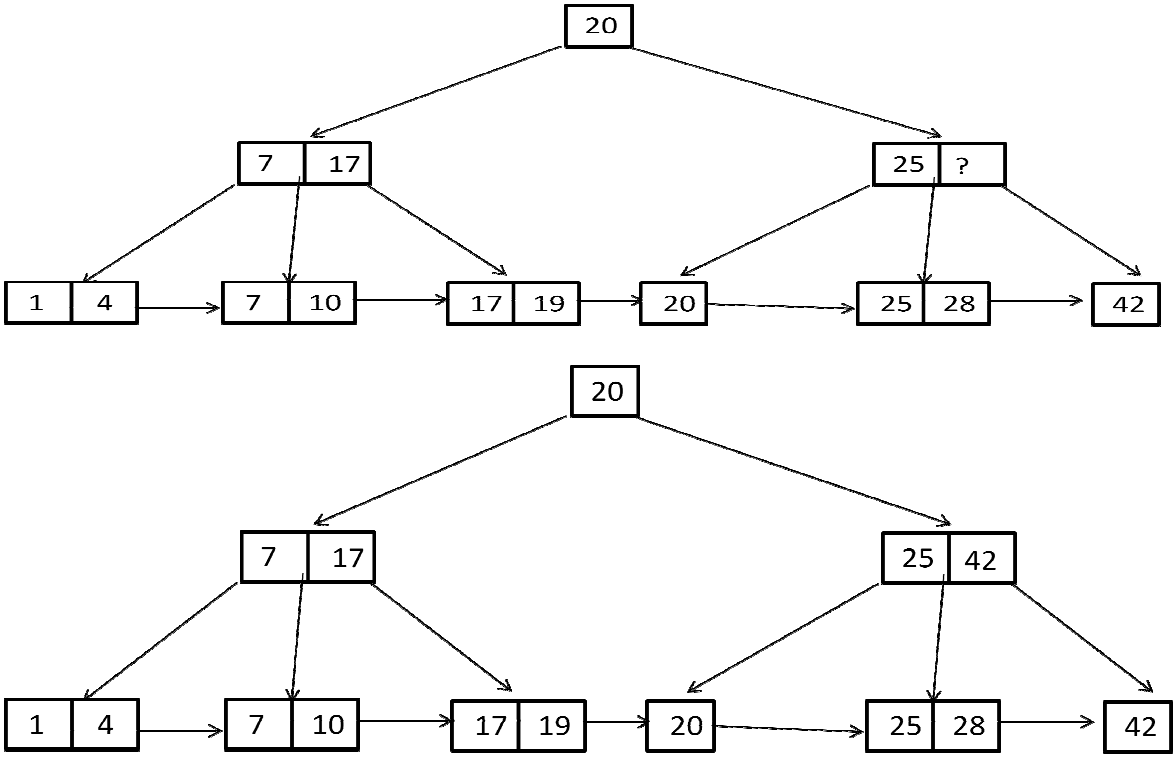


Locate 21 in the leaf nodes. check any internal node has 21. No.
Every leaf node can store 2 values. After removing one value leaf node should contain at least 1 value. Removing 21 leaves the leaf with one value..



Delete 31 from the tree

Locate 31 in the leaf node. 31 is there in internal node also. Delete both.
Deleting 31 from the internal node leave 42 in the leaf node without reference. So 42 is copied in its parent node.



Summary:

- B+ Tree is a self-balancing data structure for executing accurate and faster searching, inserting and deleting procedures on data
- We can easily retrieve complete data or partial data because going through the linked tree structure makes it efficient.
- The B+ tree structure grows and shrinks with an increase/decrease in the number of stored records.
- Storage of data on the leaf nodes and subsequent branching of internal nodes evidently shortens the tree height, which reduces the disk input and output operations, ultimately consuming much less space on the storage devices.

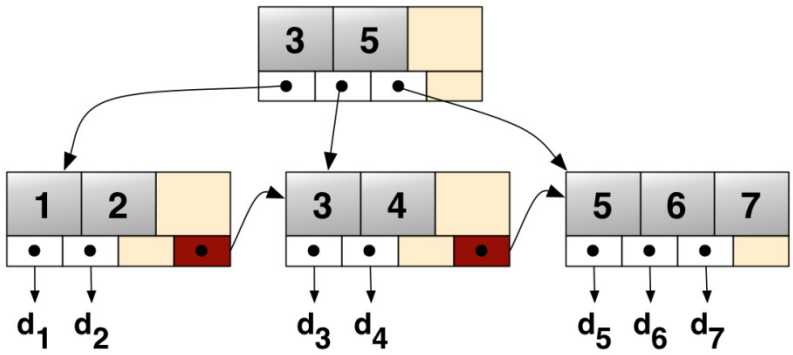


Figure) A Simple B+ tree example linking the keys 1-7 to data values d1-d7. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is b=4.

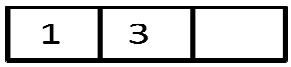
Example 1) Create a B + Tree with the values 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

B+ Tree stores all the values in its leaf nodes. Assume that we are creating a B+ tree of order (4). Every node is capable of storing 3 values.

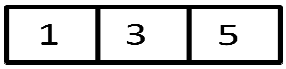
Insert 1

1 inserted in the leaf node.

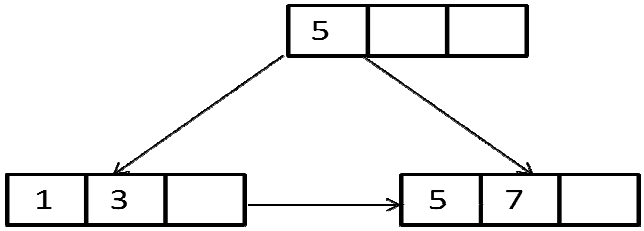
Insert 3



Insert 5

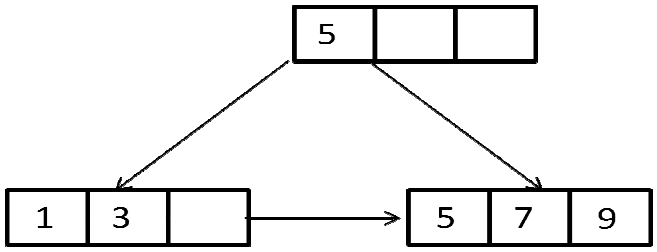


Insert 7



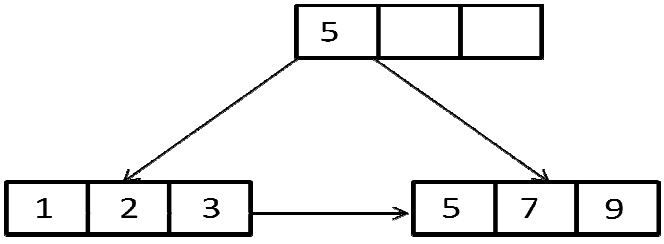
To insert 7 there is no room in the leaf node. Arrange the elements in ascending order. Split the node in to two equal parts. The 3rd value will be considered as index and is promoted to its parent. Parent node is treated as index node. And all the values we can see in the leaf nodes.

Insert 9

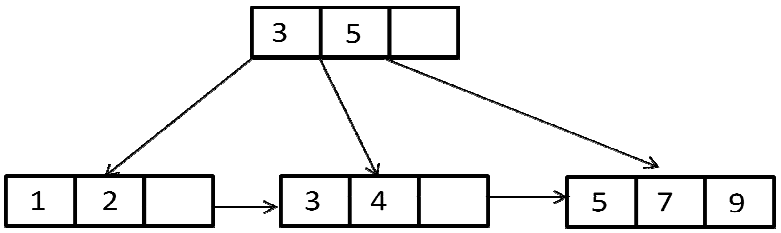


Locate the leaf node to insert 9. If the leaf node has some room to accommodate store the value.

Insert 2

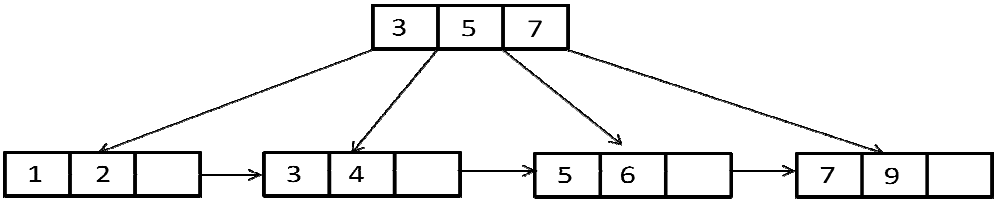


Insert 4

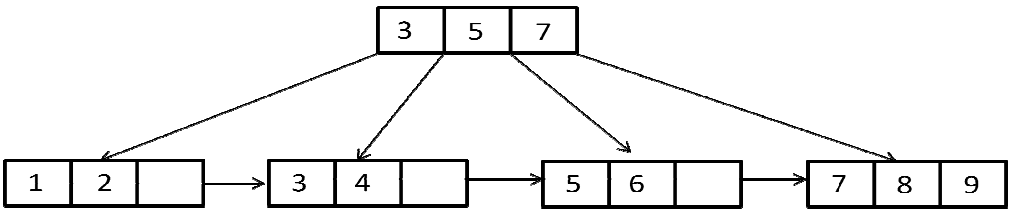


To insert 4 in the leaf node which is already filled with 3 values it is splitted into two nodes and the 3rd value is promoted to its parent.

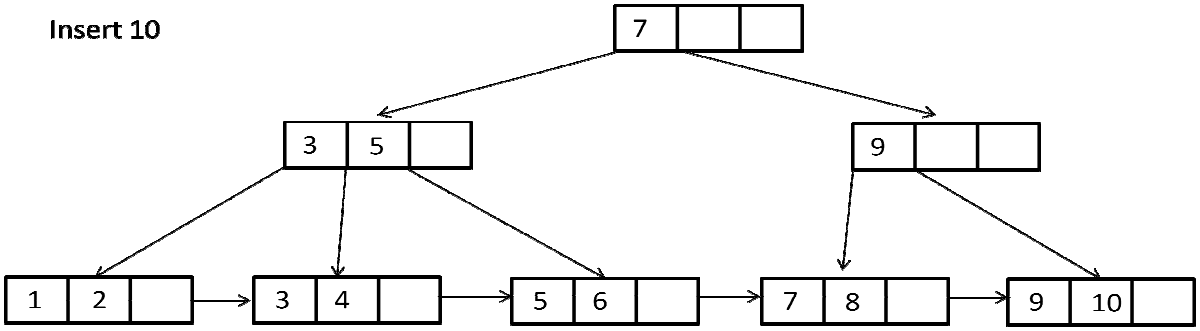
Insert 6



Insert 8



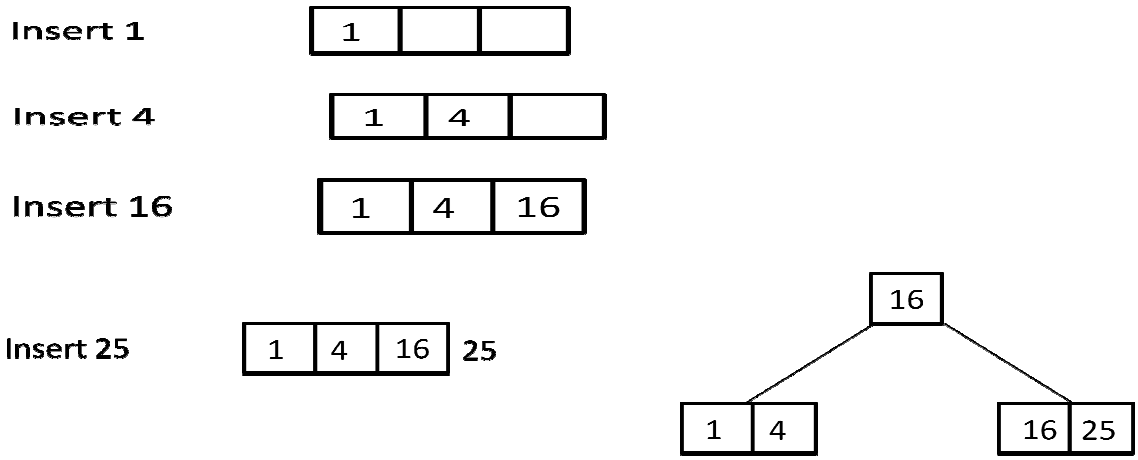
Insert 10



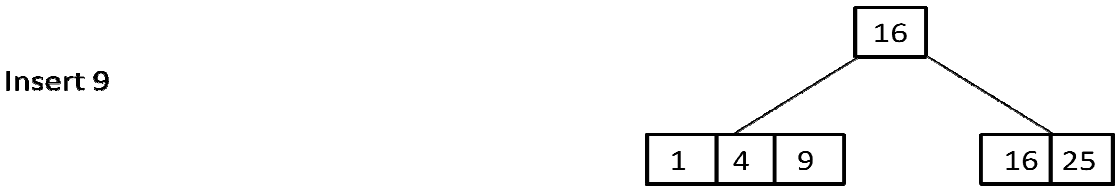
Locate the leaf node to insert 10. To insert 10 in the leaf node which is already filled with 3 values it is splitted into two nodes and the 3rd value is promoted to its parent. Parent is also filled with 3 values. It also splitted into two nodes. 3,5 as one node and 7,9 as another node. But we donat have 3 refereces to link with node 7,9. So 7 is promoted to parent.

We can observe that all the values we have inserted are available in leaf nodes. More over we can see that the vales are arranged in the ascending order. The internal nodes are used as index nodes used for searching of vales.

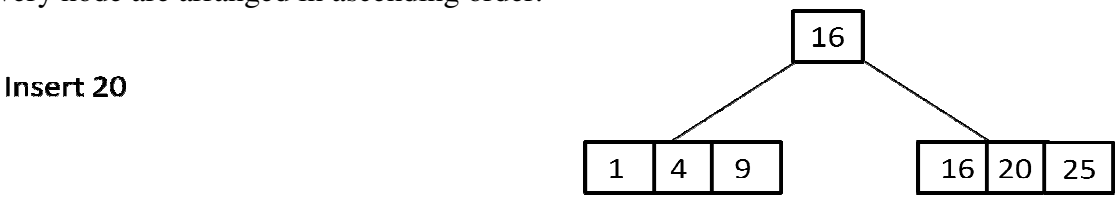
Example 2) Insert 1, 4, 16, 25, 9, 20, 13, 15, 10, 11 into an empty B+ tree of order 4



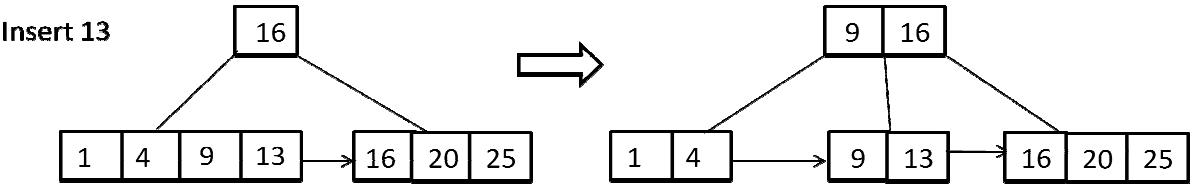
Inserting 25 in the leaf node causes the node to be splitted in to two halfs and the first value of the second node will be promoted to parent as index node.



To insert value 9 search starts with root node. Locate the leaf to insert 9. As node has room to store one more value 9 is placed there. Every leaf node can store 3 values. The values of every node are arranged in ascending order.

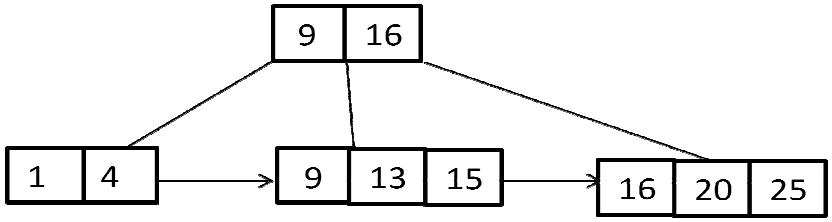


To insert value 20 search starts with root node. Locate the leaf to insert 20. As node has room to store one more value 20 is placed there. Every leaf node can store 3 values. The values of every node are arranged in ascending order.

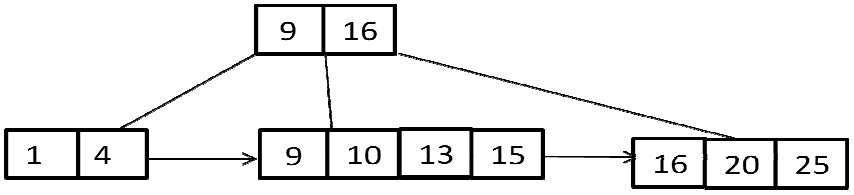


To insert value 13 search starts with root node. Locate the leaf to insert 13. Search ends with 1,4,9. The node is full. So the node is divided into two nodes 1,4 and 9,13. The first value of second node will be pushed as index.

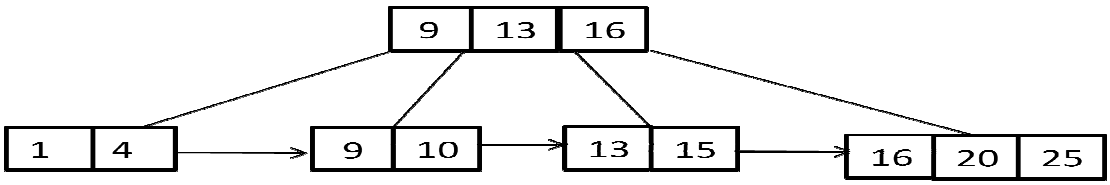
Insert 15



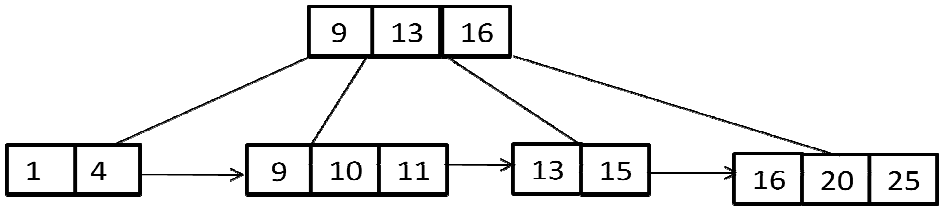
Insert 10



To insert value 10 search starts with root node. Locate the leaf to insert 10. Search ends with 9, 13, 15. The node is full. So the node is divided into two nodes 9, 10 and 13, 15. The first value of second node will be pushed as index.



Insert 11



Deletion Algorithm (B+ Tree)

STEP 1 Find leaf L containing (key, pointer) entry to delete

STEP 2 Remove entry from L

STEP 2a If L meets the "half full" criteria, then we're done.

STEP 2b Otherwise, L has too few data entries.

STEP 3 If L's right sibling can spare an entry, then move smallest entry in right sibling to L

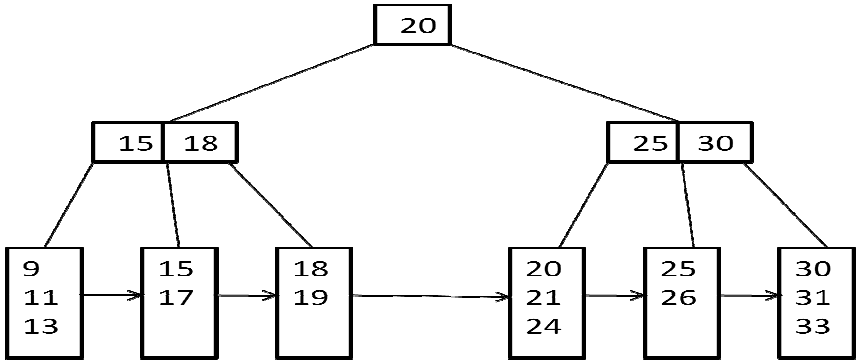
STEP 3a Else, if L's left sibling can spare an entry then move largest entry in left sibling to L

STEP 3b Else, merge L and a sibling

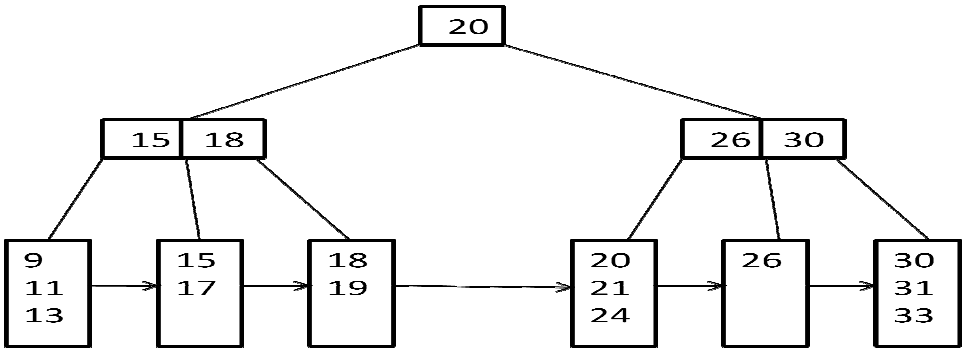
STEP 4 If merging, then recursively deletes the entry (pointing to L or sibling) from the parent.

STEP 5 Merge could propagate to root, decreasing height

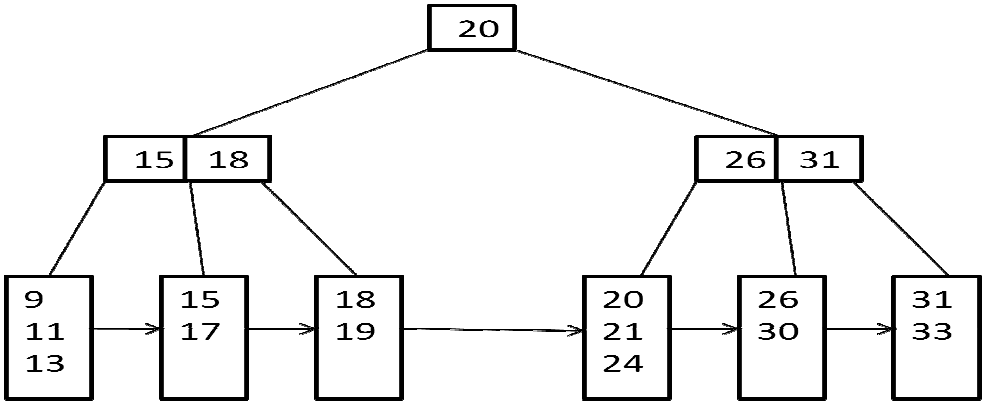
Example 1) Delete 25 from B+ tree of order m=3 and l=3.



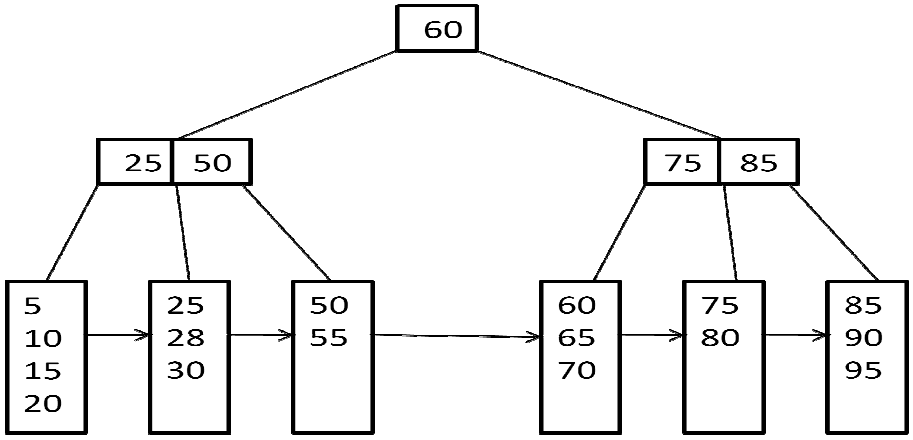
Locate 25 in the leaf node. 25 is in leaf node as well as 25 is there in index in the internal node. Deleting 25 from leaf node as well as 25 from the internal node also. Then the minimum value from the same node will replace 25 of index. So 25 is replaced by 26.



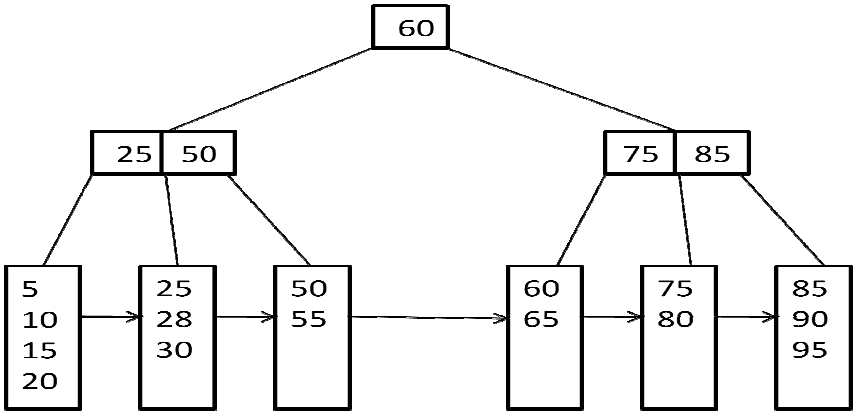
The leaf containing only one value i.e. 26 underflow. Because we need to have at least 2 values in the leaf node. So borrow min key from the right sibling. Value 30 will move to this node. now 31 becomes first element in the next leaf node and replace 30 in the parent by a copy of new min in the right sibling.



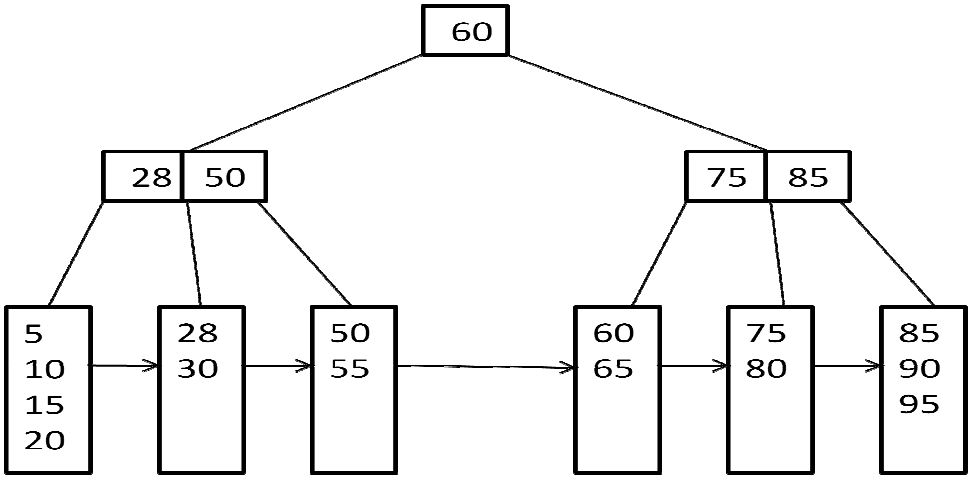
Example 2) Delete 70 from B+ tree of order m=5 and l=4



Delete 70 : Locate the leaf node having value 70. After deleting a value leaf node should contain a min of 2 values. After deleting 70 leaf node contains 2 values. so we can delete value 70 from the node without any adjustments.



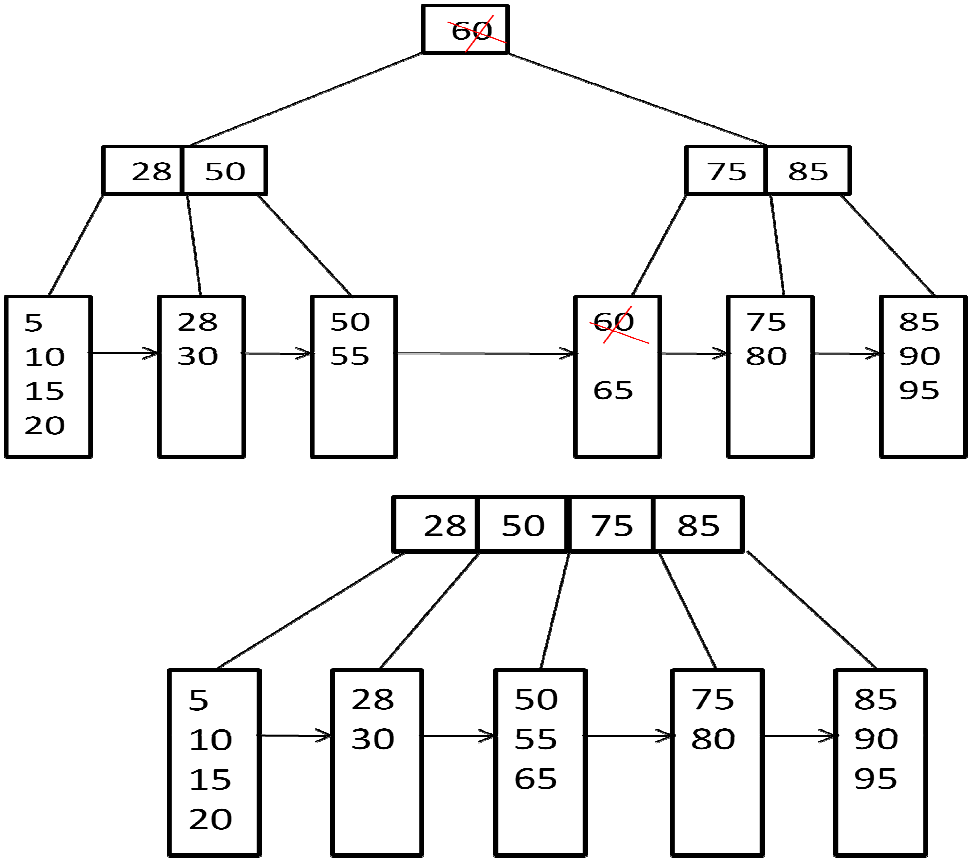
Delete 25: locate the leaf node which contains value 25. We can delete 25 from the leaf node as after deleting 25 this leaf node will have 2 values. the parent having value 25 as index. So we need to delete 25 from parent node also. The minimum value from the node will replace 25 in the parent node after deleting of 25. 28 will be placed as index.



Delete 60 : locate value in leaf node. One internal node also has value 60. We need to delete both the values.

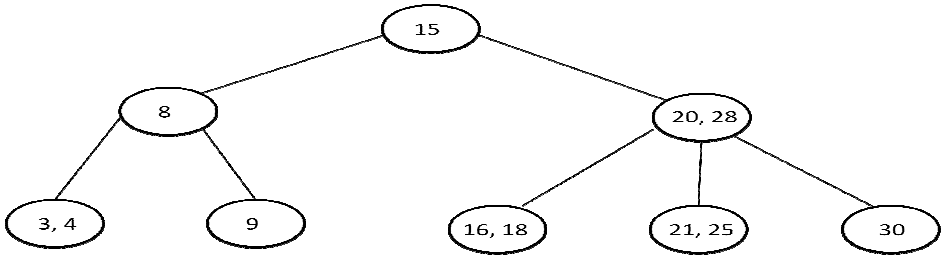
Deleting 60 from the leaf node causes the leaf underflow. After deleting it will contain only one value. We need to get value from the adjacent leaf. Both the adjacent leaves has only 2 values. So leaves need to be merged.

Deleting 60 from root merges its children. Right leaf of 50 and left leaf of 75 will be merged.



QUESTIONS FROM OLD QUESTION PAPERS
(MULTIWAY SEARCH TREES)

1) What information we can store at the nodes of 2-3 tree to quickly find the key value of the ith smallest item? Explain the use of this information to find the 9th item in the 2-3 tree below.



If we consider inorder traversal of the above tree we get sorted order. 3, 4, 8, 9, 15, 16, 18, 20, 21,25, 28, 30. The 9th item or the 9th smallest element in the inorder traversal is 21. The steps to search for node containing value 21.

2) Differentiate Binary Trees and multiway trees

Definition of B-tree

A B-tree is the balanced M-way tree and also known as the balanced sort tree. It is similar to binary search tree where the nodes are organized on the basis of inorder traversal. The space complexity of B-tree is O(n). Insertion and deletion time complexity is O(log n).

There are certain conditions that must be true for a B-tree:

- The height of the tree must lie as minimum as possible.
- Above the leaves of the tree, there should not be any empty subtrees.
- The leaves of the tree must come at the same level.
- All nodes should have least number of children except leave nodes.

Properties of B-tree of order M

- Each node can have maximum M number of children and minimum M/2 number of children or any number from 2 to the maximum.
- Each node has one key less than children with maximum M-1 keys.
- The arrangement of the keys is in some specific order within the nodes. All keys in the subtree present in the left of the key are predecessors of the key, and that present in the right of the key are called successors.
- At the time of insertion of a full node, the tree splits into two parts, and the key with median value is inserted at parent node.
- Merging operation takes place when the nodes are deleted.

Definition of Binary tree

A Binary tree is a tree structure which can have at most two pointers for its child nodes. It means that the highest degree a node can have is 2 and there could be zero or one-degree node too.

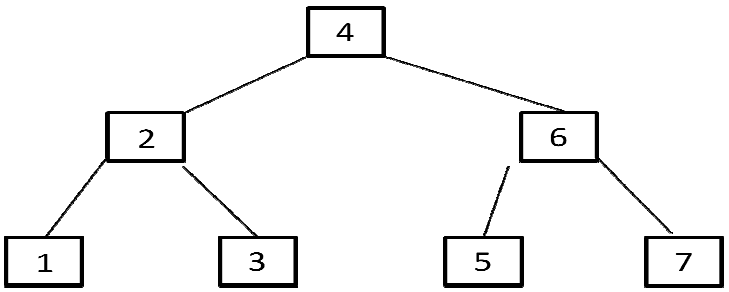
There are certain variants of a binary tree such as strictly binary tree, complete binary tree, extended binary tree, etc.

- The strictly binary tree is a tree where each non-terminal node must have left subtree and right subtree.
- A tree is called a Complete Binary tree when it satisfies the condition of having 2^i nodes at each level where i is the level.
- Threaded binary is a binary tree which consists of either 0 no of nodes or 2 number of nodes.

3) Show that all B trees of order 2 are full binary trees.

B Tree is a multiway tree. All the leves are at the same level. All internal nodes of multiway tree of order 2 contains one value and exactly two children. So the resultant tree will definitely be a full binary tree.

Actually it is not possible to construct a B-Tree of order 2. We can get that kind of tree in 2-3 tree.



4) Explain how range search is performed in a B+ tree. Give example.

Search Operation

In B+ Tree, search is one of the easiest procedures to execute and get fast and accurate results from it.

The following search algorithm is applicable:

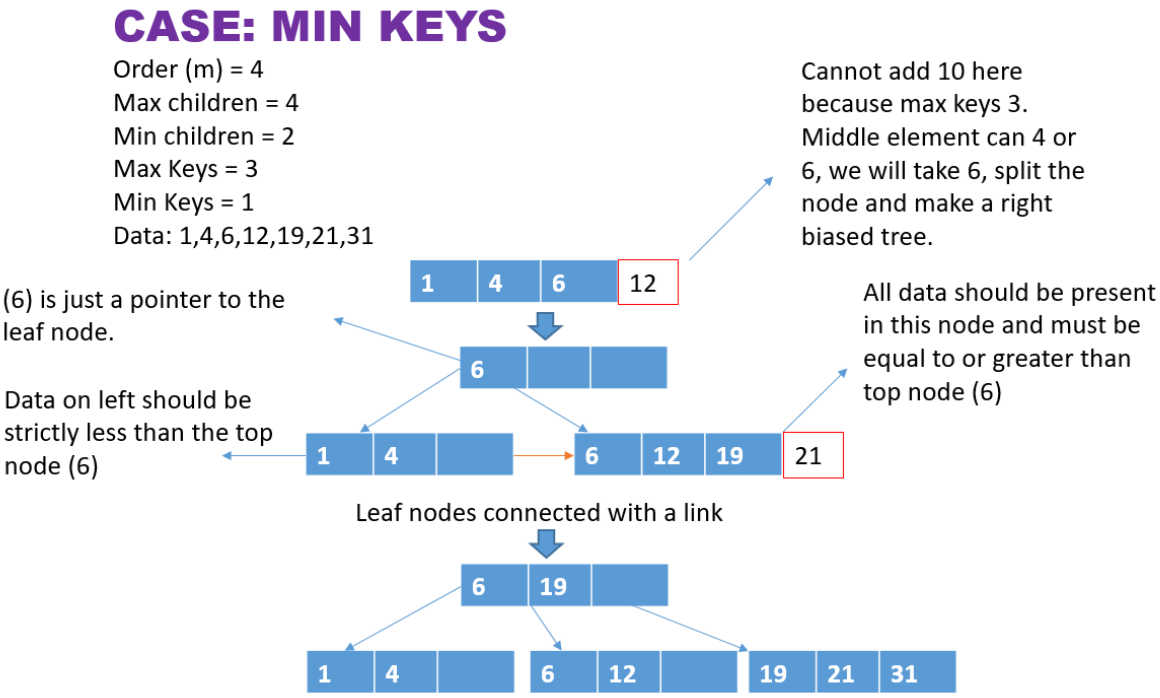
- To find the required record, you need to execute the binary search on the available records in the Tree.
- In case of an exact match with the search key, the corresponding record is returned to the user.
- In case the exact key is not located by the search in the parent, current, or leaf node, then a "not found message" is displayed to the user.
- The search process can be re-run for better and more accurate results.

Searching more than one value in a range is called range search. In B+ Tree all the data is placed in leaf nodes only. All the data items are stored in sorted order. If we know the starting and ending points we can have all the data within the rage easily

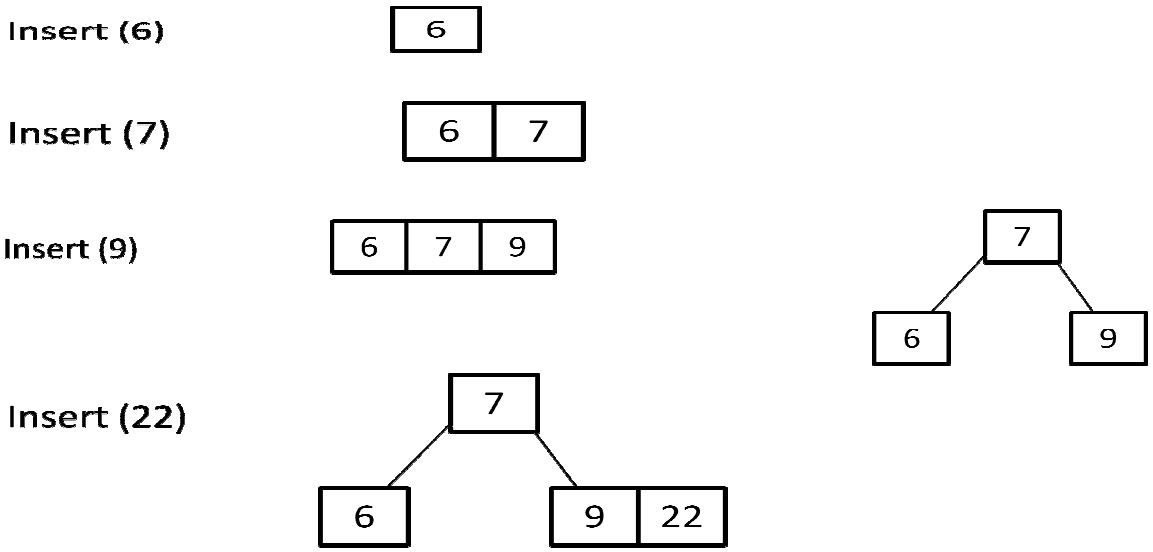
5) Explain the insertion operation in B+ tree with suitable example.

The following algorithm is applicable for the insert operation:

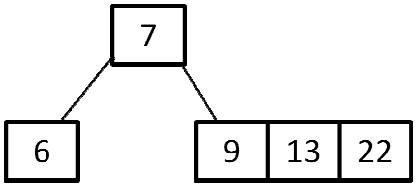
a) 50 percent of the elements in the nodes are moved to a new leaf for storage.
b) The parent of the new Leaf is linked accurately with the minimum key value and a new location in the Tree.
c) Split the parent node into more locations in case it gets fully utilized.
d) Now, for better results, the center key is associated with the top-level node of that Leaf.
e) Until the top-level node is not found, keep on iterating the process explained in the above steps.

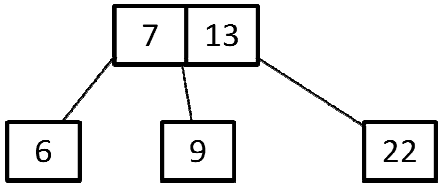


6) Construct 2-3 tree by using the following sequence of numbers. 6, 7, 9, 22, 13, 31, 35, 28, 24, 5, 34, 8, 25, 10, 11, 12, 14, and 39.

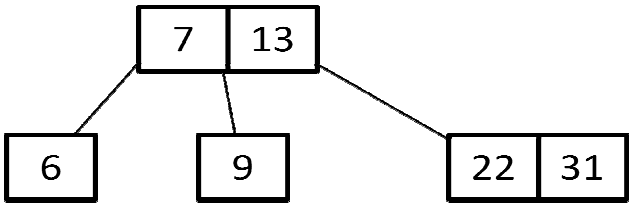


Insert (13)

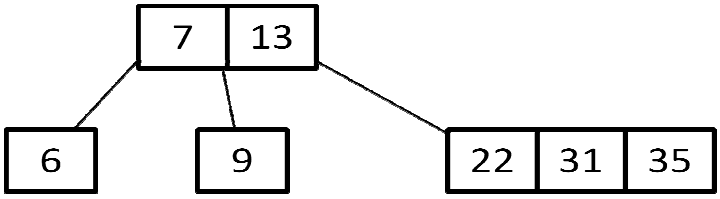


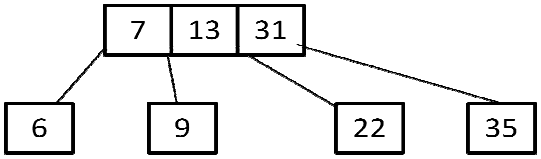


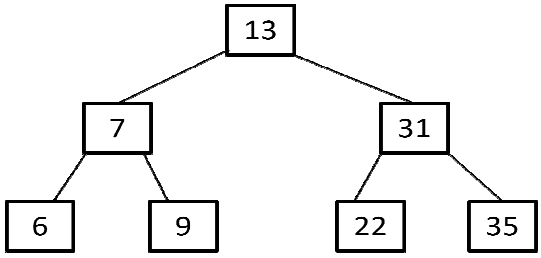
Insert (31)



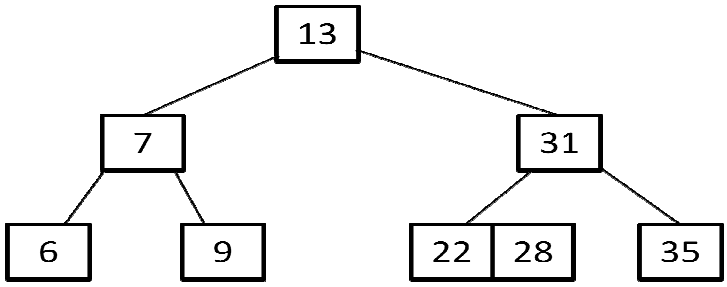
Insert (35)



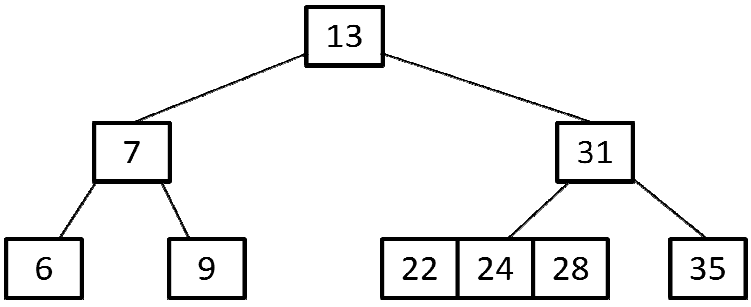


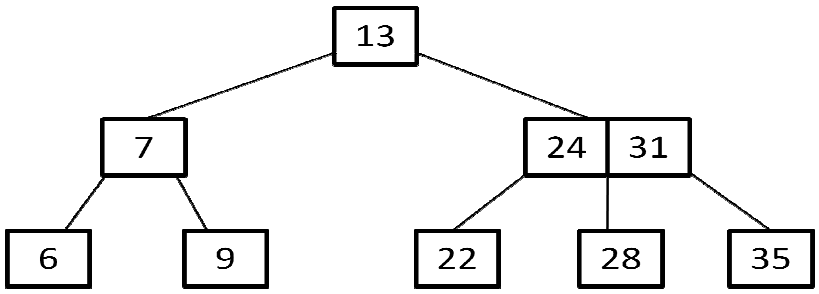


Insert (28)

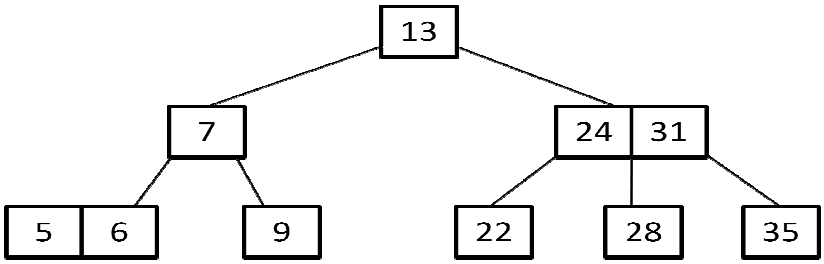


Insert (24)

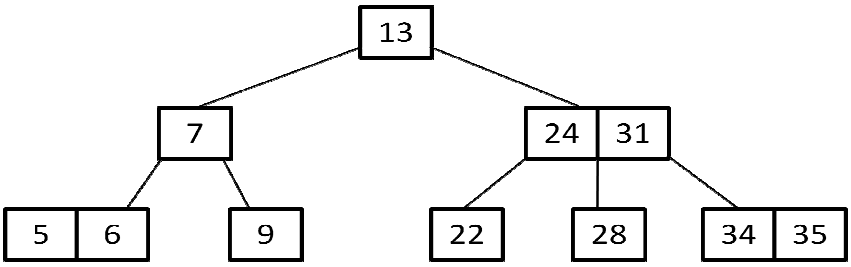




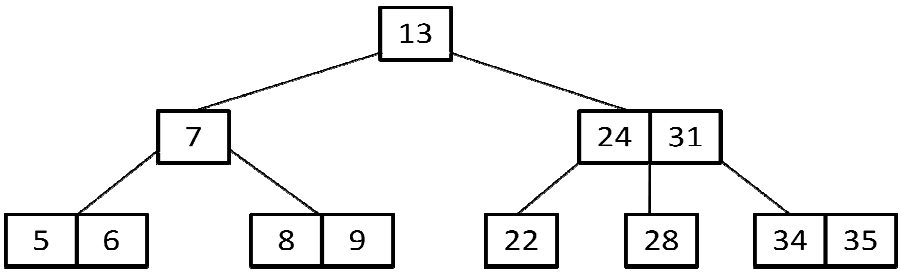
Insert (5)



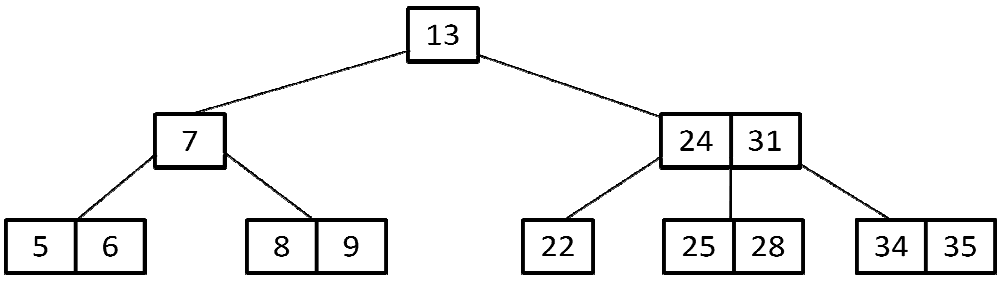
Insert (34)



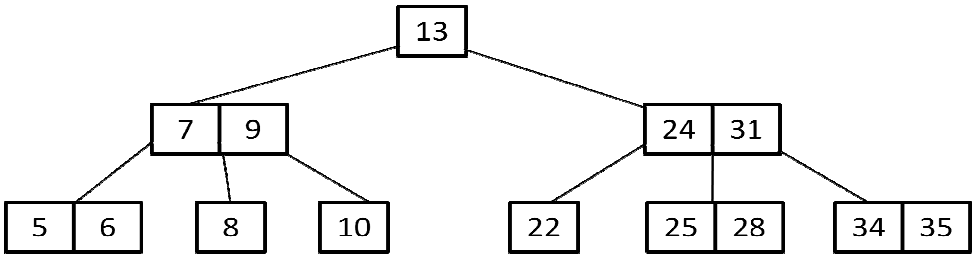
Insert (8)



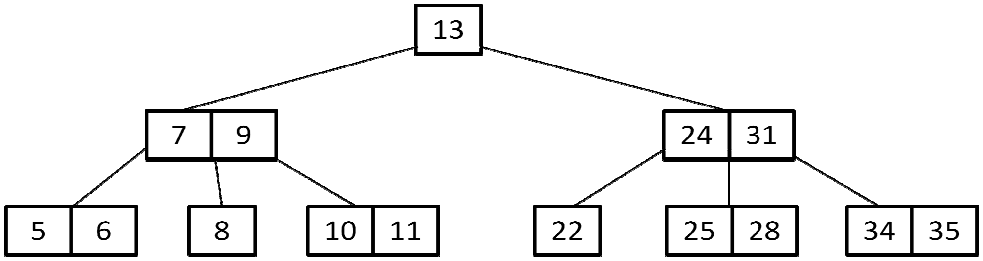
Insert (25)



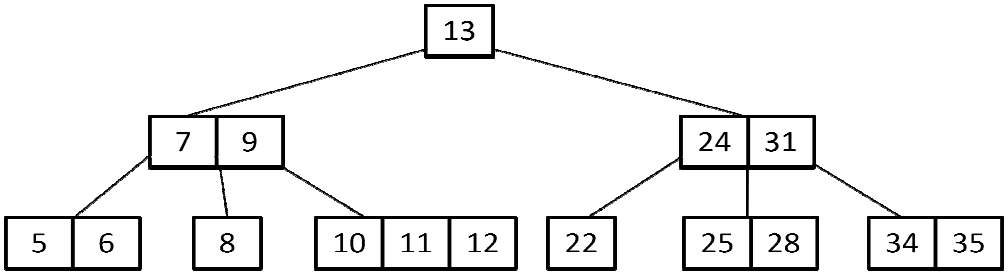
Insert (10)

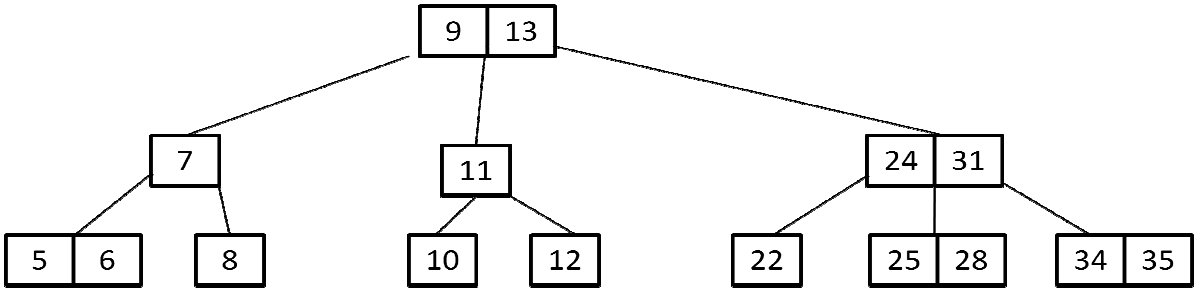


Insert (11)

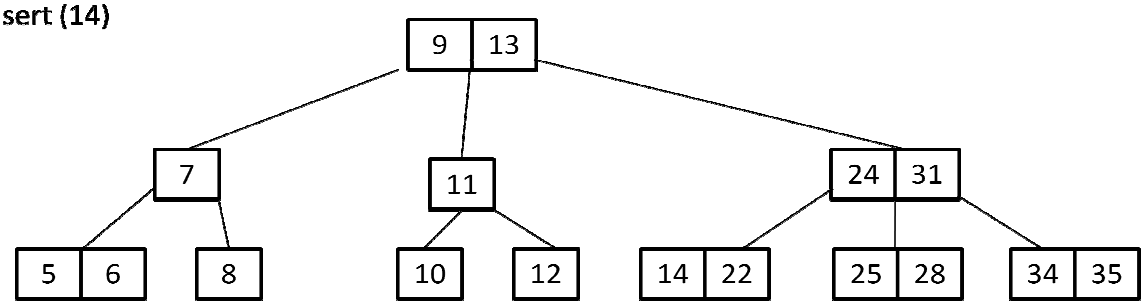


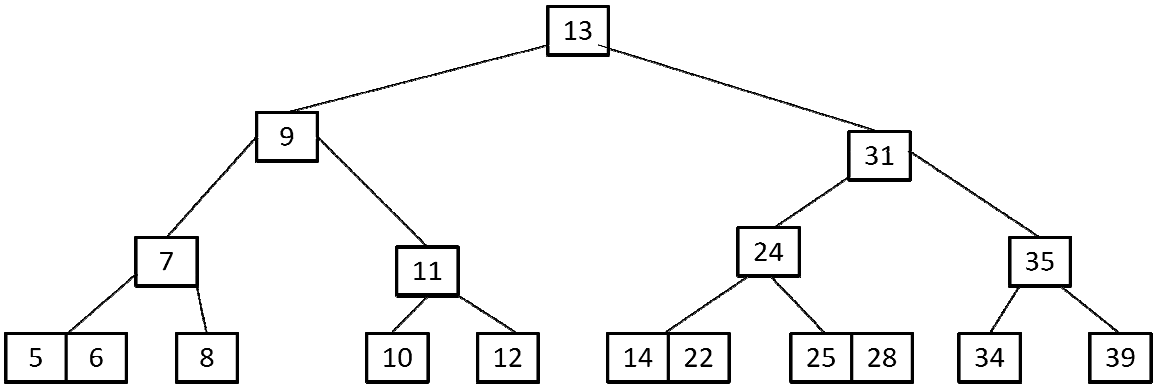
Insert (12)





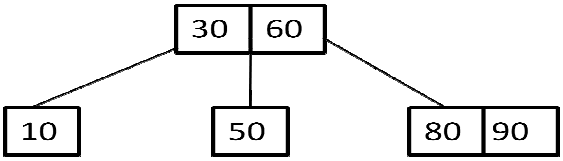
Insert (14)

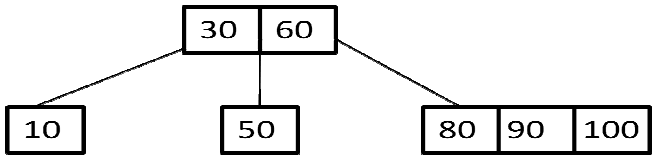




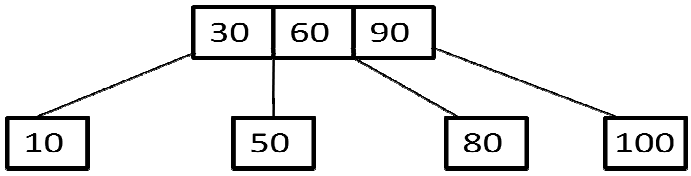
7) Explain how to split an internal node for right and left branch in 2-3 trees.

For example to insert value 100 into the following 2-3 tree . where 2-3 tree means 3 way tree. Each node may have a maximum of 2 values and a maximum of 3 children. we try to locate the leaf node to insert value 100.

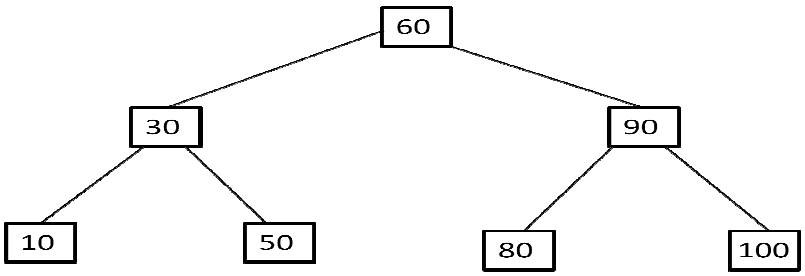




To insert value 100 we start searching with root node to find the leaf node to insert value 100. As 100>60 we go to right sub tree and the search ends at node containing two values (80,90). 100 is supposed to be placed at this node but there is no space. The node is splitted into two and the middle value is promoted to its parent. 90 is promoted to parent node.



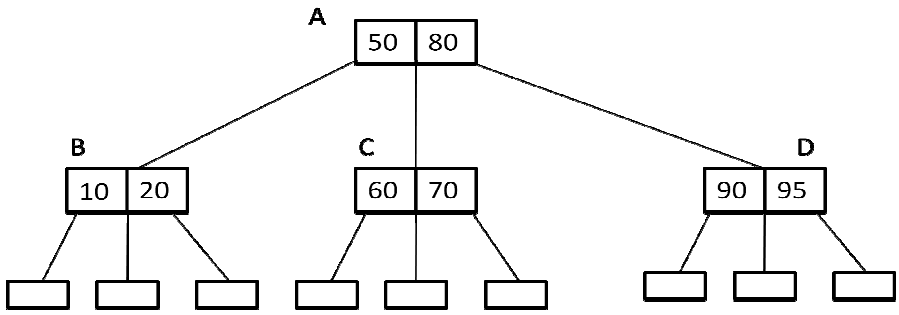
Now 90 is going to store in the parent node where parent node already contains 2 nodes and there is no room for 90.Elements will be arranged in the ascending order in temporary buffer and parent will be splitted into two nodes and the middle will be promoted to its parent.



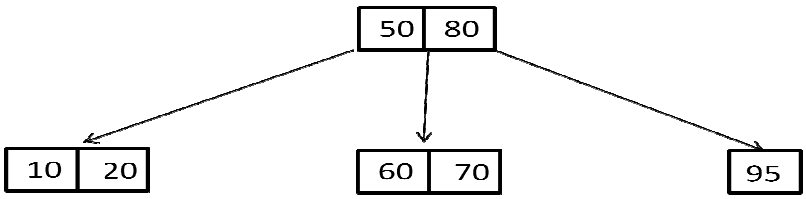
8) Explain how to split a leaf node for right and left branch in 2-3 trees.

same as q. no 7.

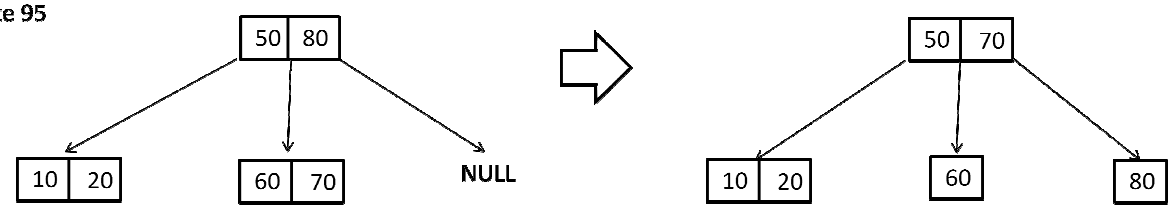
9) Use the deletion algorithm to delete the elements with keys 90, 95, 80, 70, 60, and 50 in this order from the given following 2-3 trees. Show the resulting 2-3 tree following each deletion.



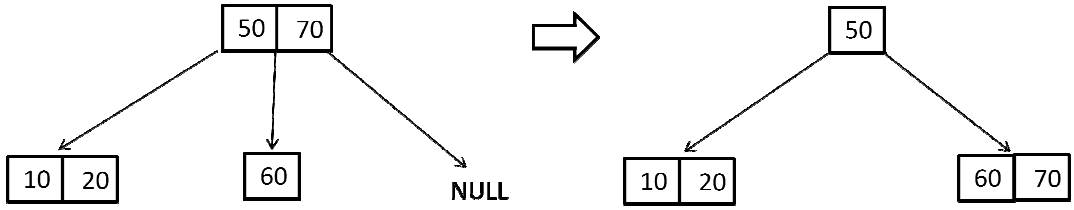
Delete 90



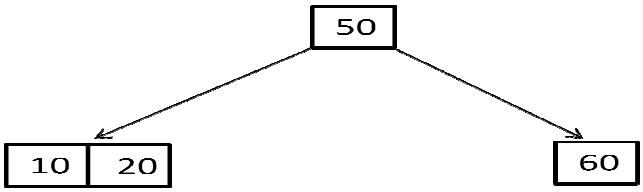
Delete 95



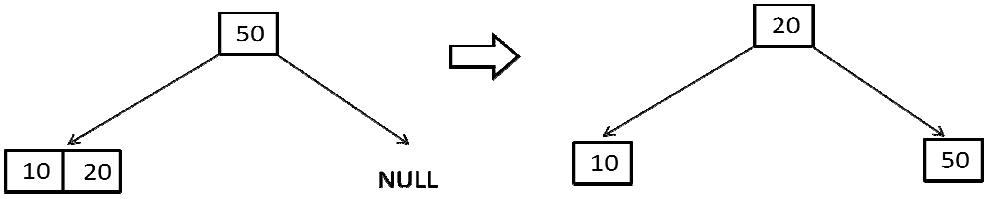
Delete 80



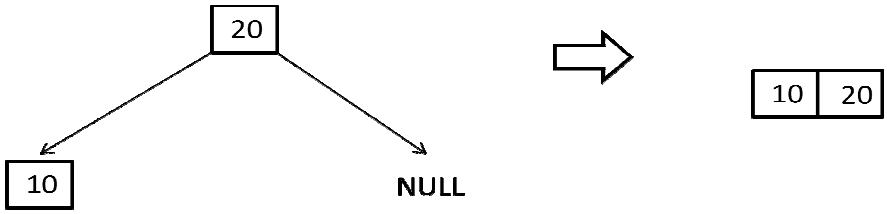
Delete 70



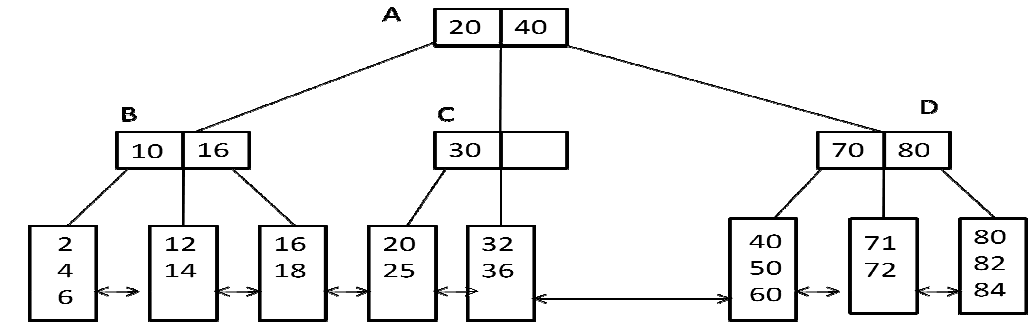
Delete 60

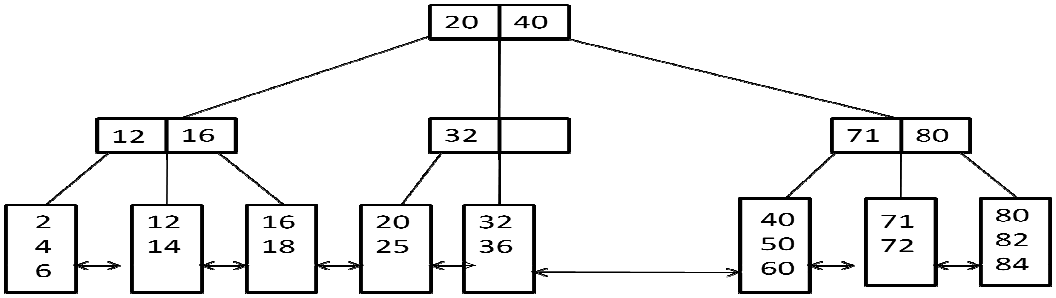


Delete 50

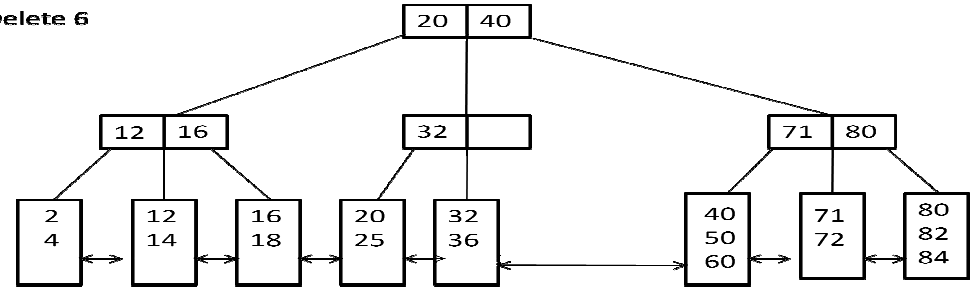


10) Show resulting B+ tree after deleting each of the following keys 6, 71, 14, 18, 16 and 2.

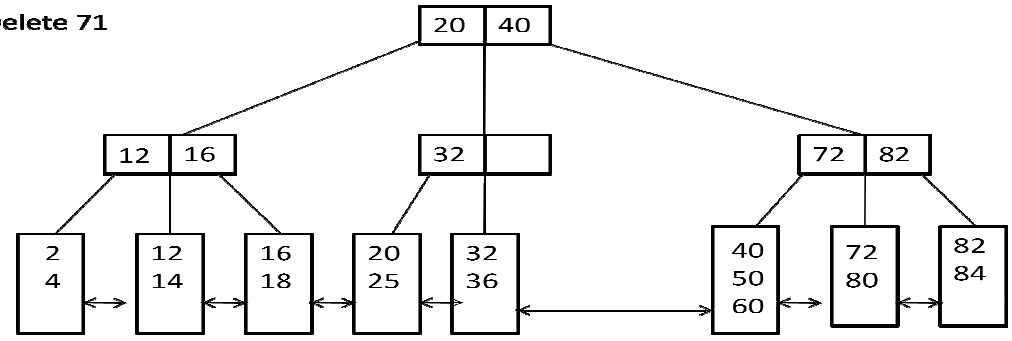




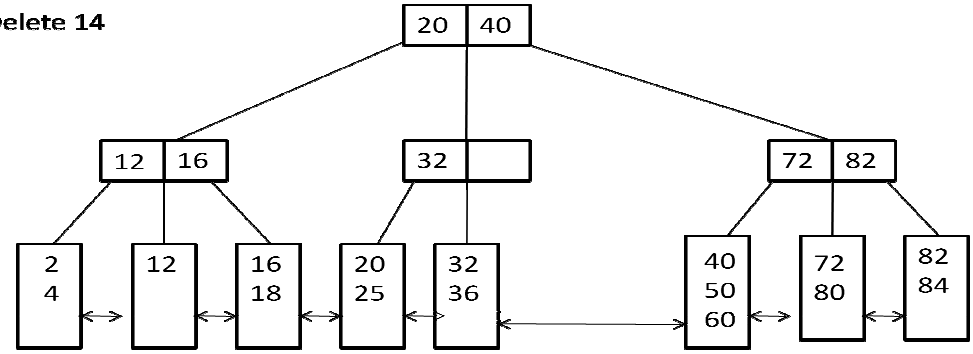
Delete 6

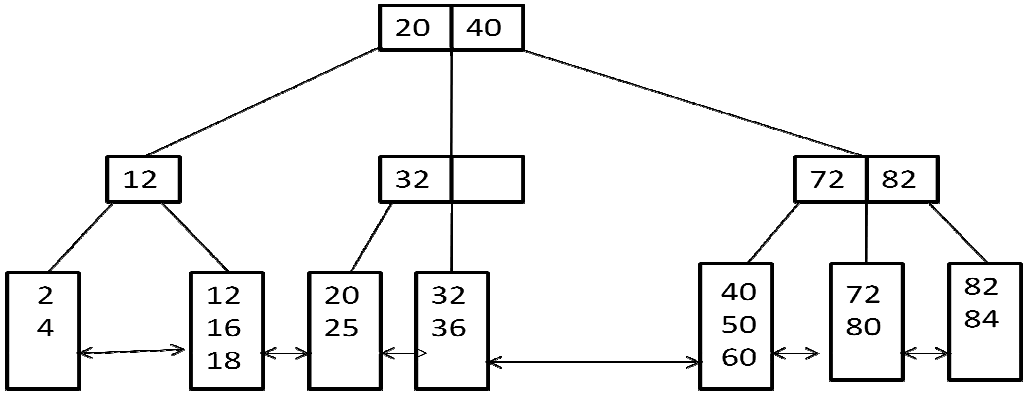


Delete 71

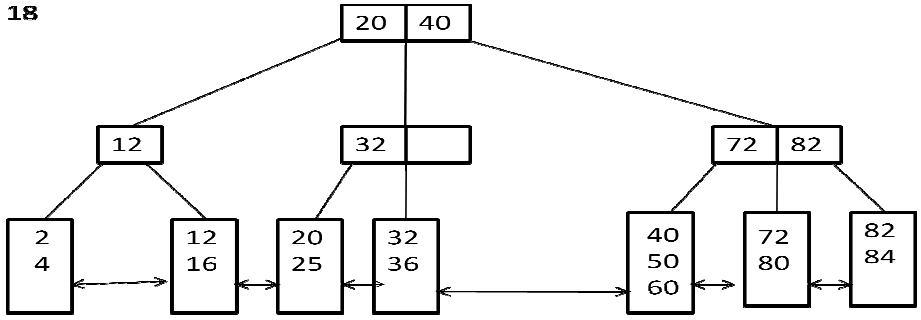


Delete 14

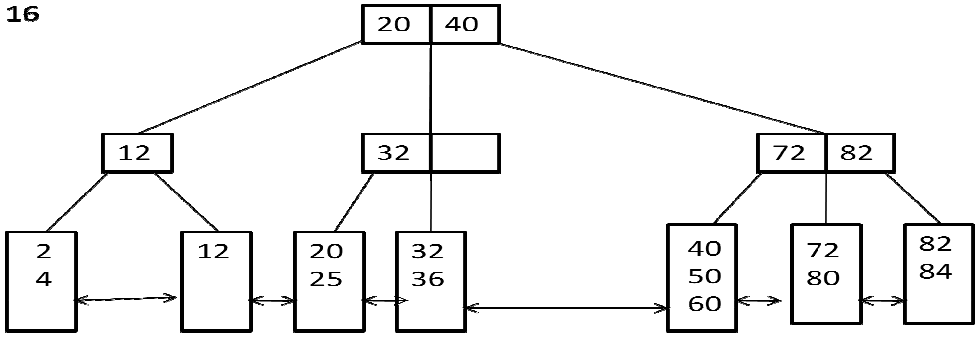


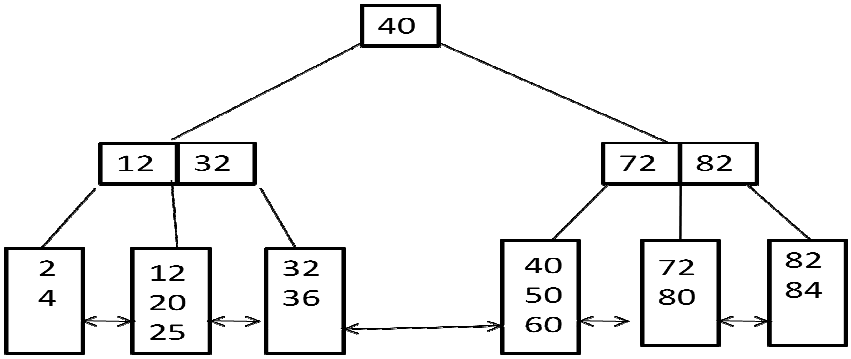


Delete 18

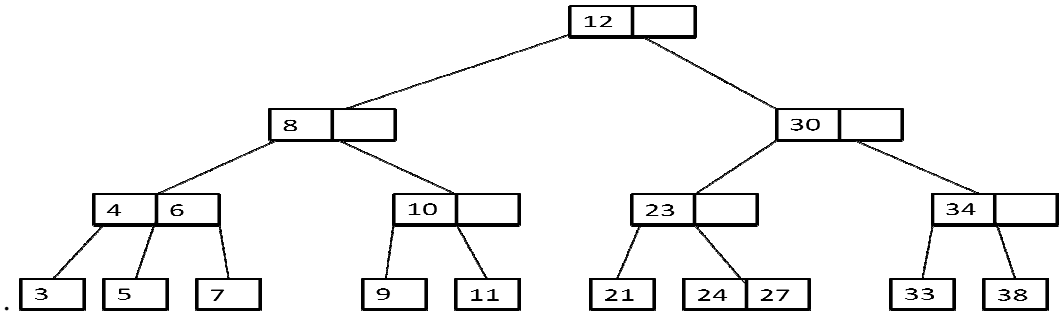


Delete 16





11) Deletion 38, 5, 8 from the following 2-3 trees and show the resulting 2-3 tree after every deletion operation.



12) Explain the insertion procedure with suitable example in B+ trees.

13) What is a B-Tree

A B-tree of order m is a multiway search tree in which:

- The root has at least two subtrees unless it is the only node in the tree.
- Each nonroot and each nonleaf node have at most m nonempty children and at least m/2 nonempty children.
- The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
- All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

14) Give the definition and properties of m-way search tree.

The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order **m**, each node contains a maximum of **m – 1** elements and m children.

All leaves are at the same level

15) Construct the 2-3 tree with the following sequence of numbers 5, 21, 8, 63, 69, 32, 7, 19 and 25.

16) Define an m-way search tree.

The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order **m**, each node contains a maximum of **m – 1** elements and m children.

All leaves are at the same level

17) Give an analysis of the B-Tree insertion process.

a) If the tree is empty, create a node and put value into the node

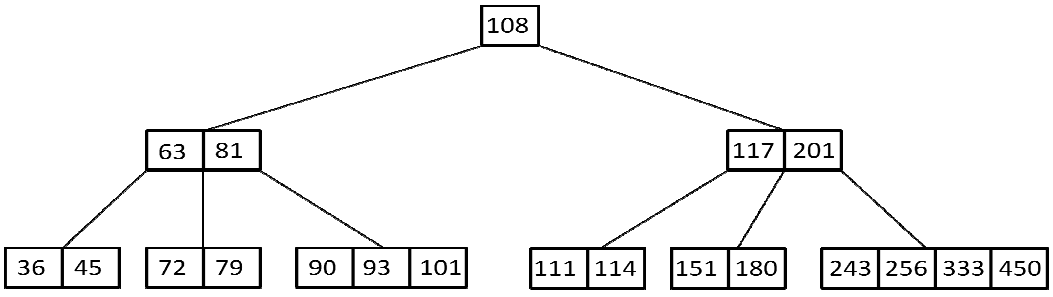
b) Otherwise find the leaf node where the value belongs.
c) If the leaf node has only one value, put the new value into the node
d) If the leaf node has more than two values, split the node and promote the median of the three values to parent.
e) If the parent then has three values, continue to split and promote, forming a new root node if necessary

Insertion Algorithm

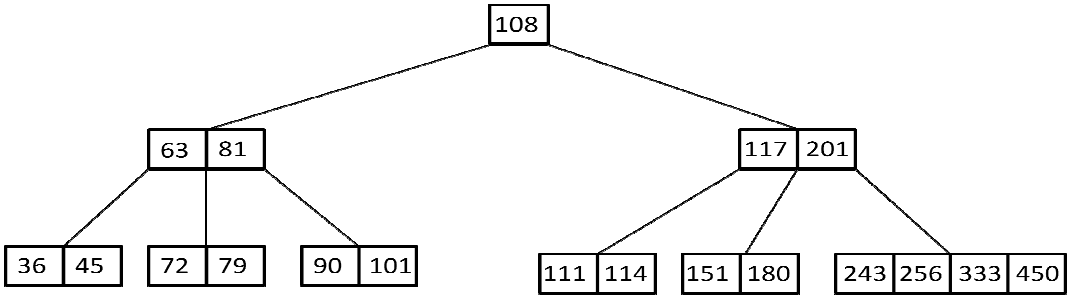
The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

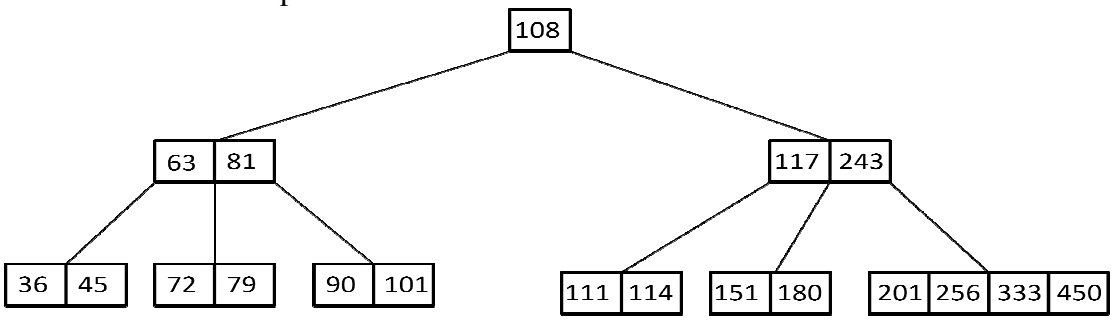
18) One after the other delete the keys 93, 201,180, and 72 from the below B-Tree of order 5.



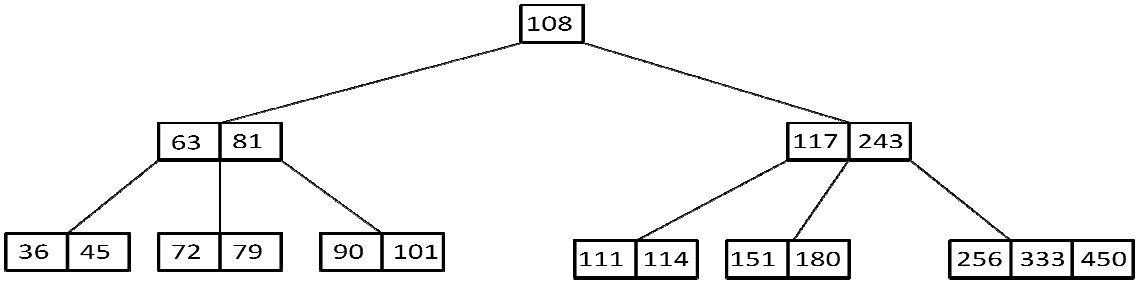
To delete value 93 first Locate node with 93. It is a leaf node. After deleting value from a leaf node the leaf node should contain half of the elements. Even after deleting 93 this node contains 2 values. So we can delete 93.



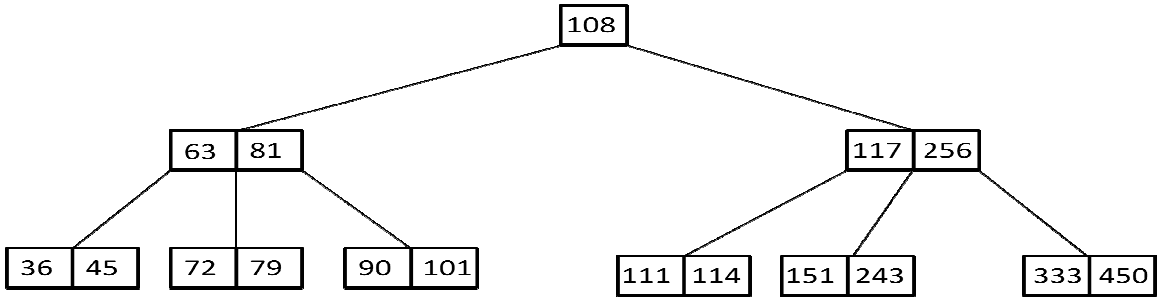
To delete value 201 first Locate node with 201. It is not a leaf node. Find inorder successor of 201. Swap the values .



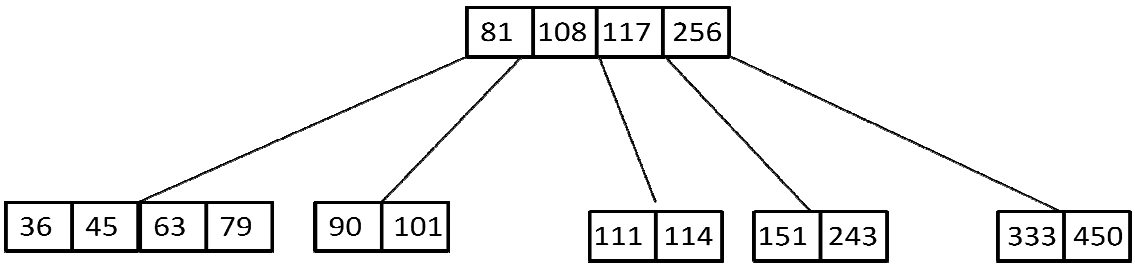
There are some more values in the node along with 201. Delete value 201. After deleting the node should contain half of the values. It contains 3 values. So we can delete 201 from the leaf node.



To delete value 180 first Locate node with value 180. It is a leaf node. Delete the value. And after deleting the value the node contains only one value 151. In the tree of order 5 leaf node should contain atleast 2 values. So readjust siblings.



To delete value 72 first Locate node with value 72. It is a leaf node. Delete the value. And after deleting the value the node contains only one value 79. In the tree of order 5 leaf node should contain atleast 2 values. So readjust siblings.



19) Define a B-Tree

B-tree is a multiway tree.

An extension of a multiway search tree of order m is a **B-tree of order m**. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order m is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.
2. Each nonroot and each nonleaf node have at most m nonempty children and at least m/2 nonempty children.
3. The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

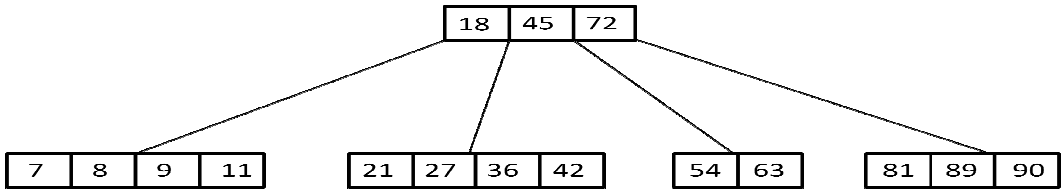
20) Discuss the advantage of using m-way search trees over binary trees.

In m-way tree all leaves are at the same level. Whereas all the leaves of binary tree need not be at same level.

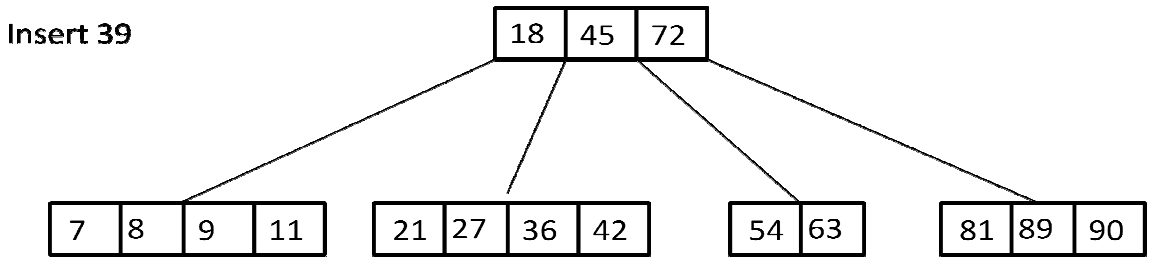
Every Node of Binary tree contains only one value and a maximum of 2 sub trees

Node of a m-way tree contains a maximum of m-1 values and a maximum of m-sub trees.

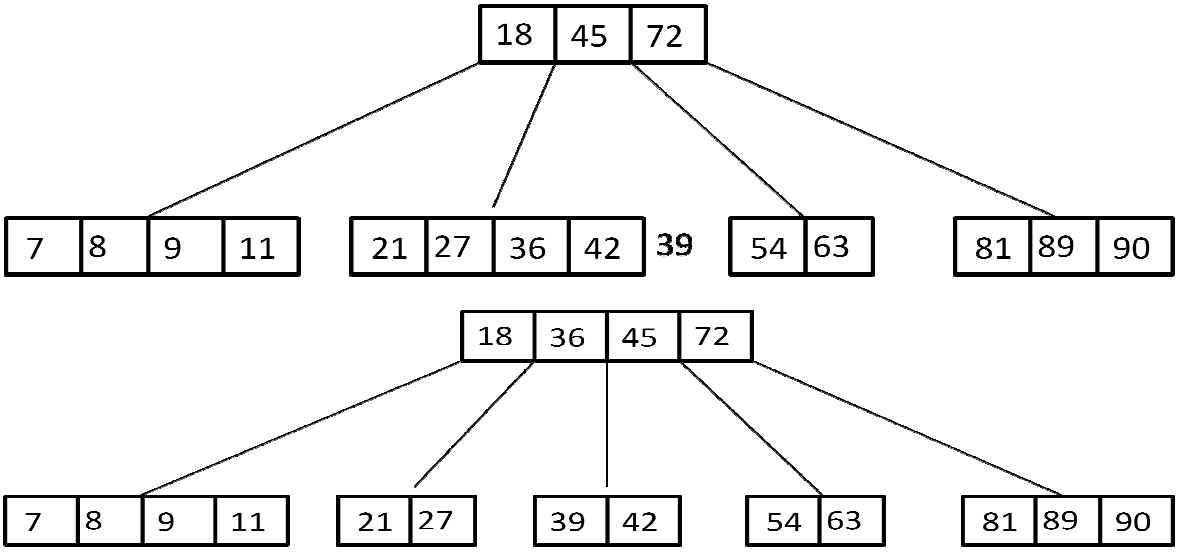
21) Identify the type of tree given below. Then insert 39 and 4 into the tree given below and perform necessary restructuring to balance the tree.

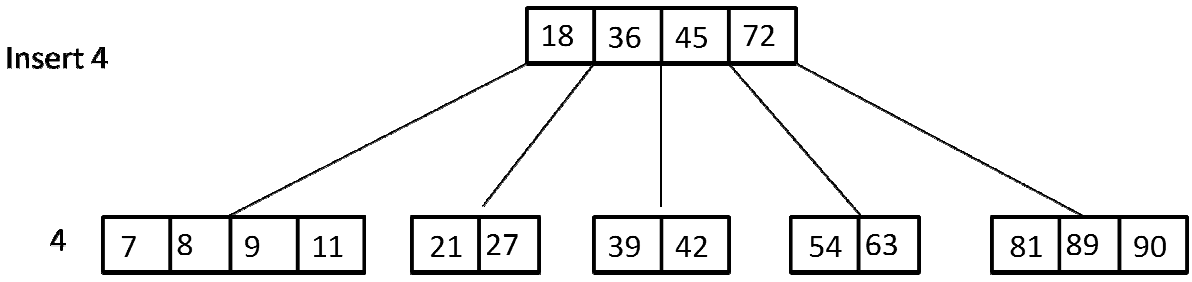


The following tree is a multi way tree of order 5. In this tree every node may have a maximum of 4 nodes and 5 sub trees.

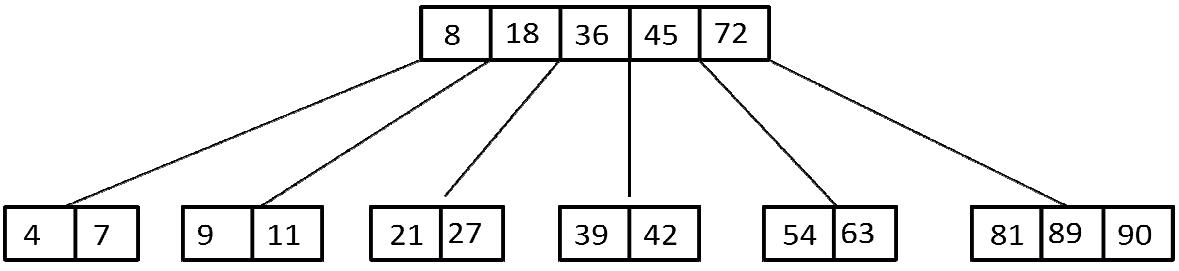


Identify the leaf node to insert value 39. As the leaf node is already having 4 values it is overflowing. So the median is promoted to its parent by splitting the node into two nodes

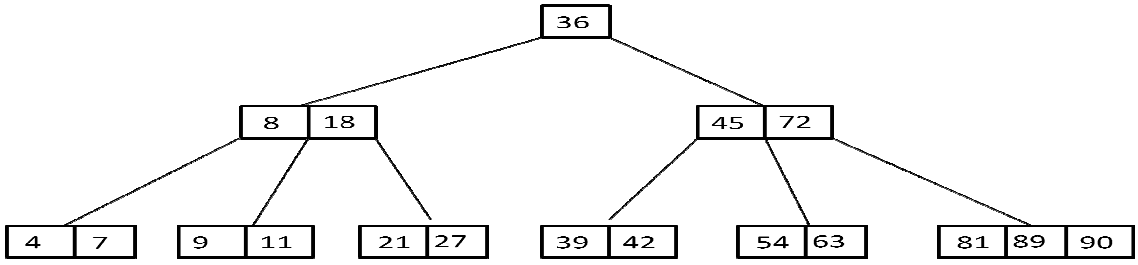




Identify the leaf node to insert value 4. As the leaf node is already having 4 values it is overflowing. So the median is promoted to its parent by splitting the node into two nodes.



While inserting value 8 in the parent node , the parent already having 4 values and the median is promoted to its parent by splitting the node int two.



22) What is the maximum number of disk accesses required for B-Trees deletion if its height is h and only one node can be retrieved at a time.

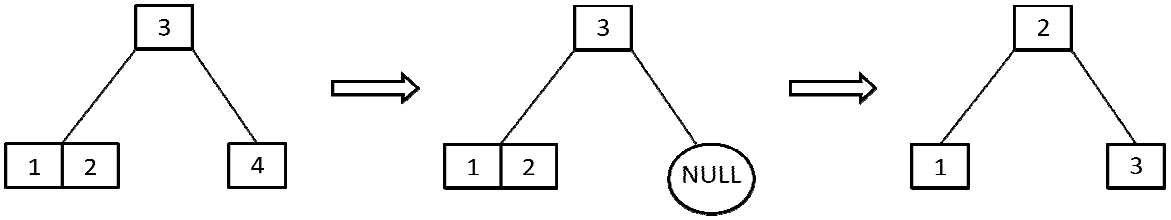
B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node.

23) Provide high level description of algorithm for B-Tree deletion.

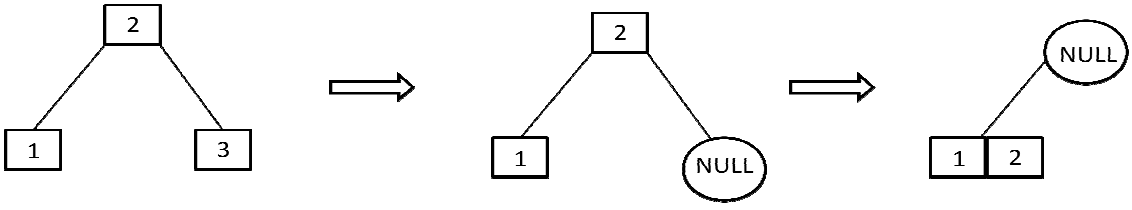
Deletion Algorithm :: Deleting an Item I from a B- Tree

1. Locate node n, which contains item I
2. If node n is not a leaf node , swap I with Inorder successor.
3. If leaf node contains another item, just delete I, else try to redistribute nodes from siblings-if not possible merge nodes.

Redistribution after deleting 4

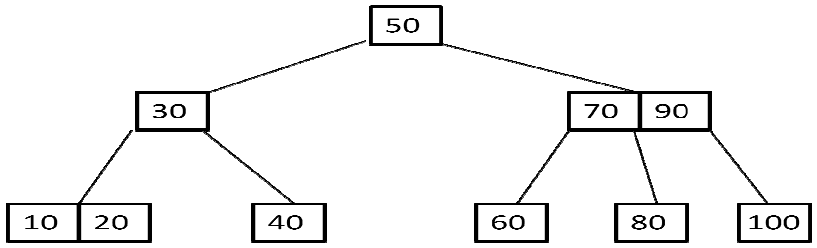


Merging after deleting 3

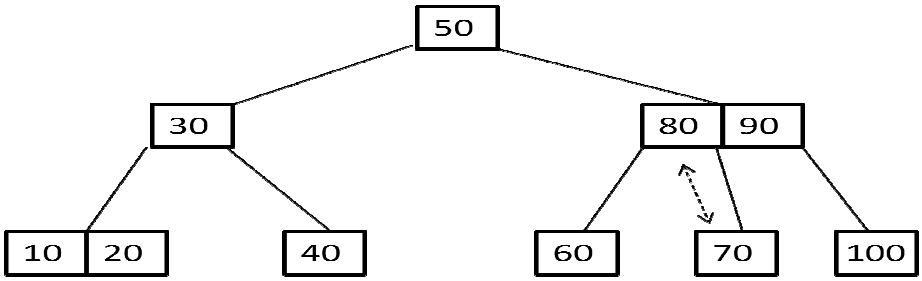


For Example Delete node with value 70 from the following 2-3 tree.

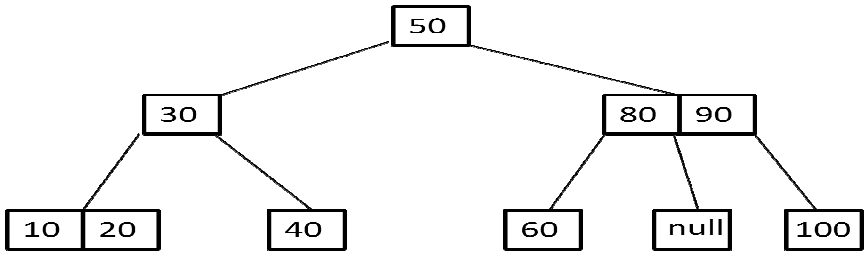
Delete 70



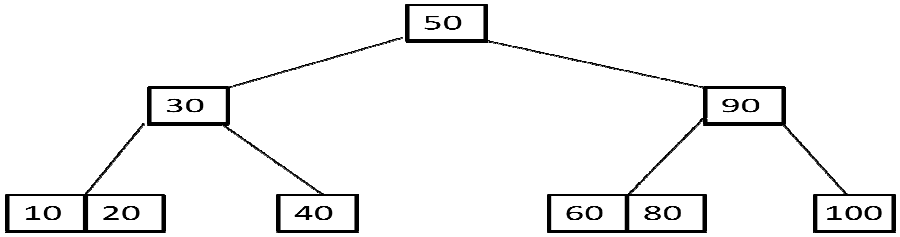
Identify node 70. Check for its inorder successor. 80 is its inorder successor. Swap node to be deleted with its inorder successor.



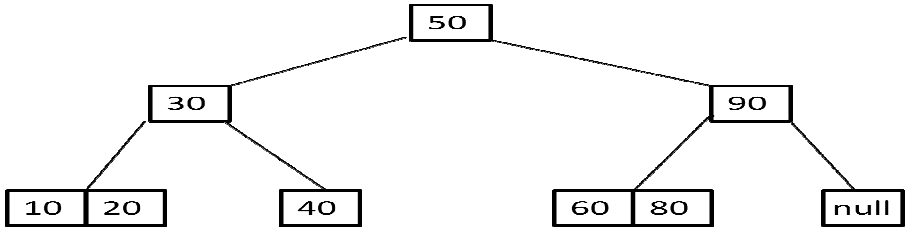
Delete the leaf node with value 70. Now we need to merge nodes after deleting leaf node.



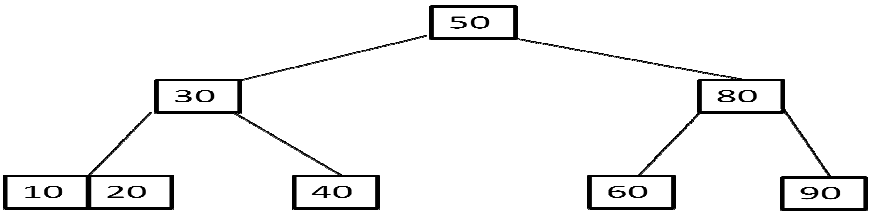
As there is no representation for the link between 80 and 90. Node 80 will be moving down and will be merged with node 60.



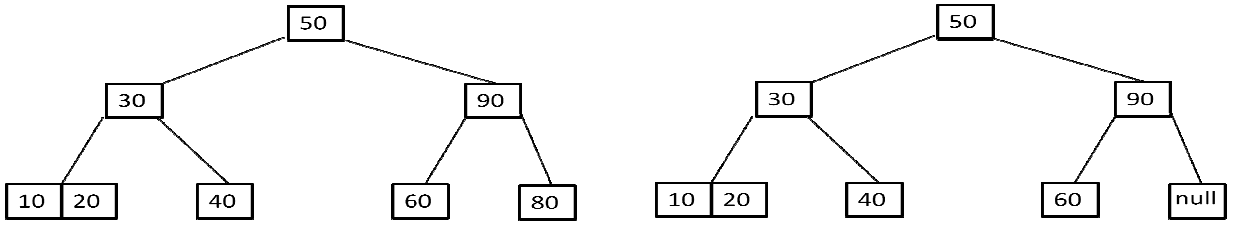
Delete 100 : identify the node 100. It is a leaf node. Delete node.



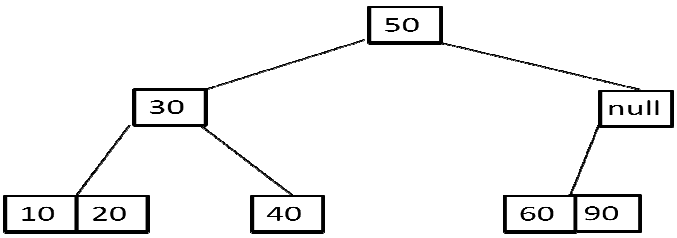
Readjusting nodes to balance the tree. Node 80 will be moving upwards and node 90 will be moved downwards.



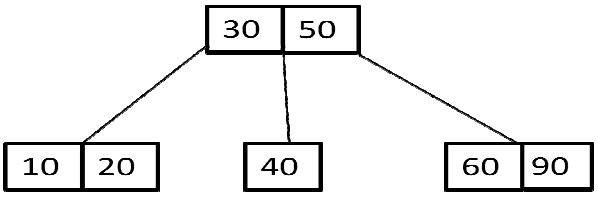
Delete node 80 : identify node 80. It is not a leaf node. Find inorder successor of node 80. Node 90 is inorder successor of 80. Node 90 is a leaf node.. swap node 80 and its inorder successor.



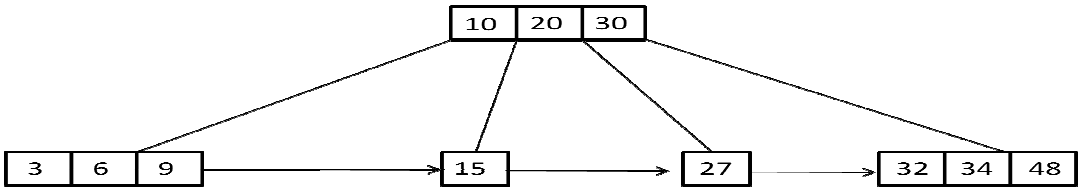
As there is no representation to the right sub tree of node 90. Node 90 will move down and will be merged with node 60 to keep the tree balanced.



Now right of 50 is null but it is not leaf node. 50 will move down and will be merged with 30

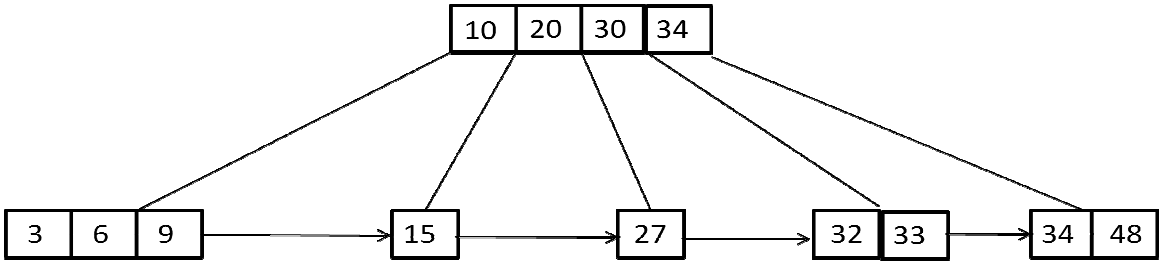


24) Identify the type of tree given below. Then insert 33 and 44 into the tree given below and perform necessary restructuring to balance the tree.



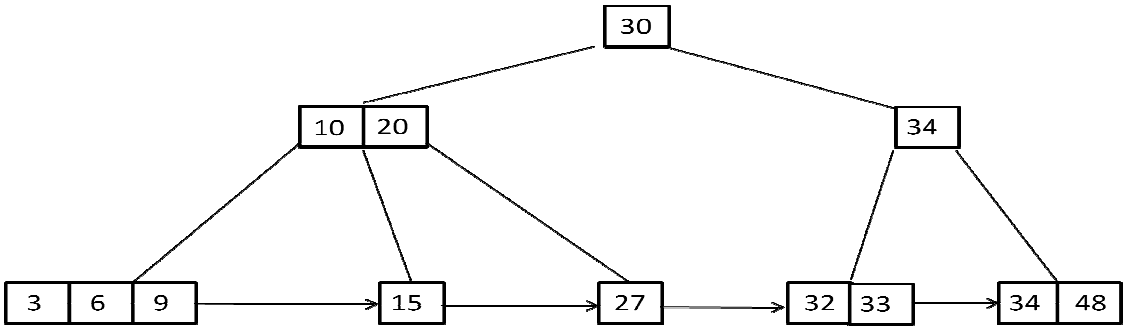
This is a B+ tree of order 4. Every node may contain a maximum of 3 values and a maximum of 4 branches.

Insert 33

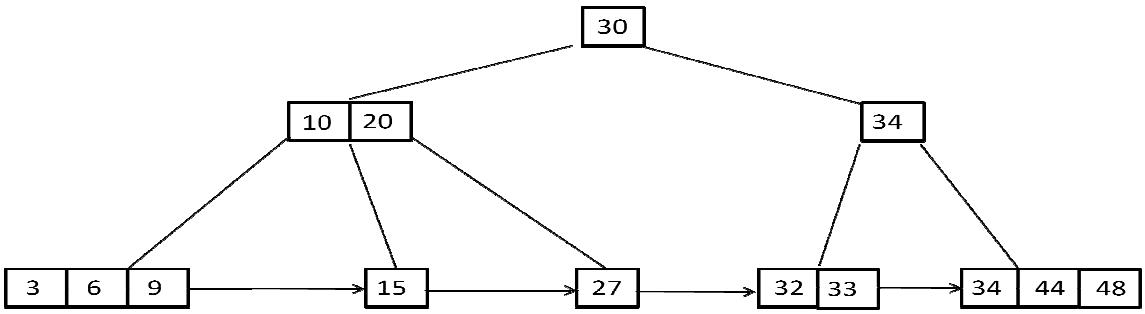


Every node may contain a maximum of 3 values. Locate the leaf node to insert value 33. This node already has 3 values. Arrange values in ascending order. Split the node into two halfs. First value of second node will be promoted as index to the parent node.

But the parent already has 3 nodes. Inserting 34 causes overflow. So the node is splitted into two. And readjusts. The node having 10 and 20 values has 3 links but the node having 30 34 has 2 links only. One link is missing. So, internal nodes will be readjusted.



Insert 44



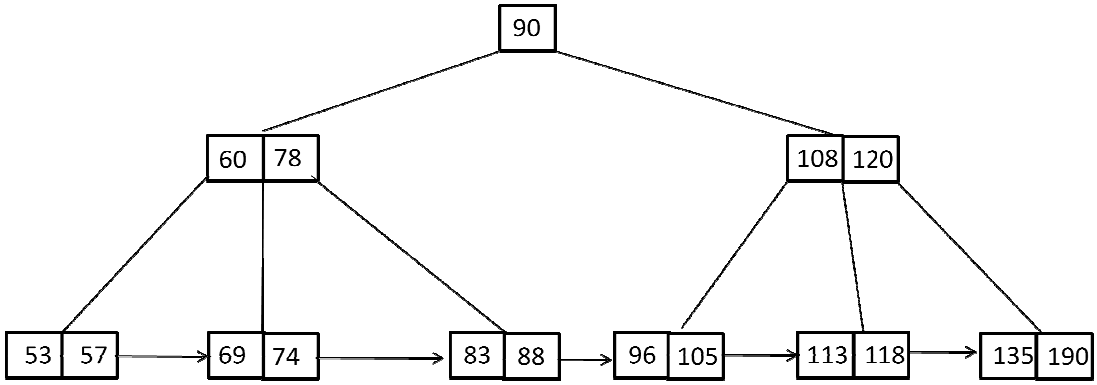
Locate the leaf node to insert value 44. The leaf node contains only 2 nodes and there is a room to store 44. So it was inserted there.

25) Define a B+ Tree.

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.



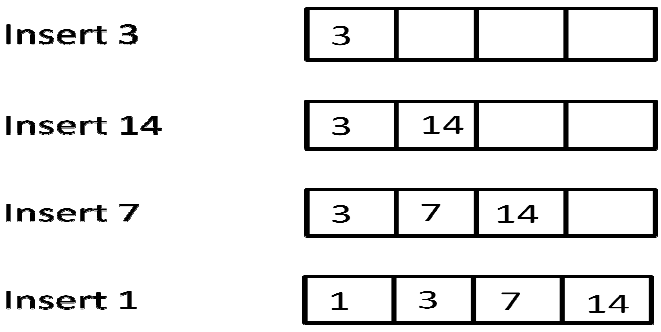
Advantages of B+ Tree

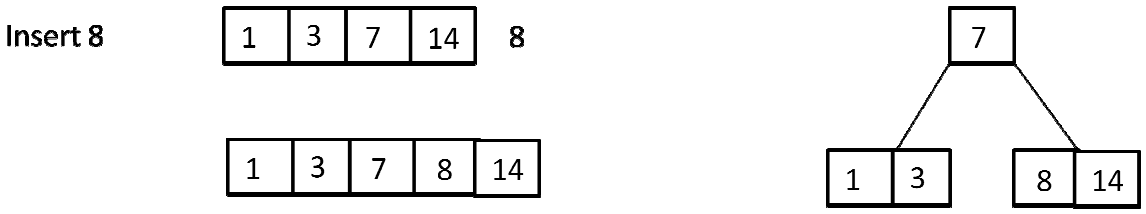
1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

26) Compare and contrast B-Trees and B+ Trees.

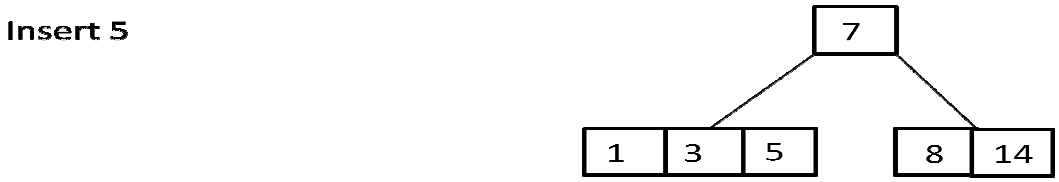
SNo	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as in internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

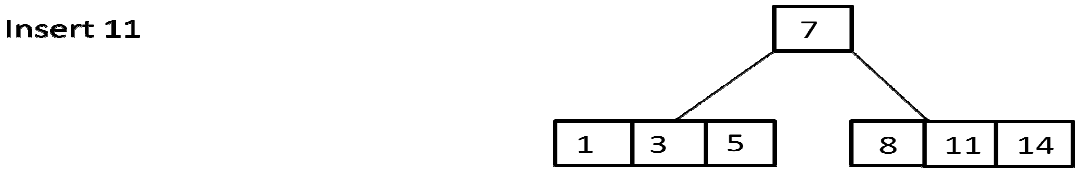
27) Create a B Tree of order **5** by inserting the following elements one after the other: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25 and 19.



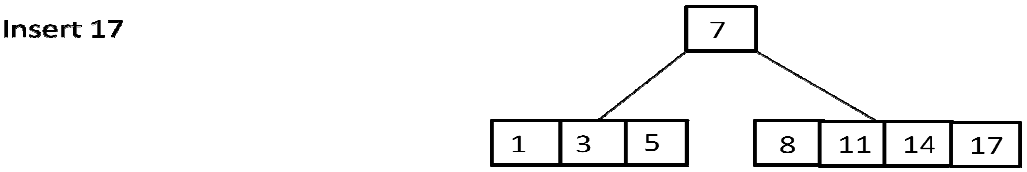


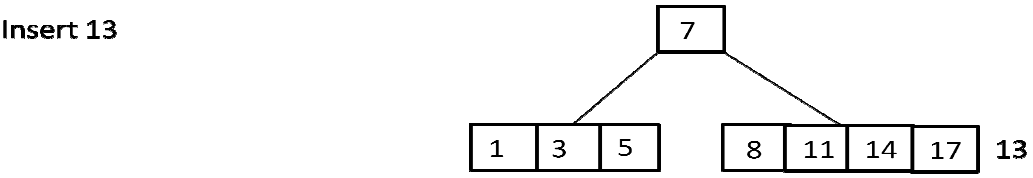
To insert 8 there is no room so the middle value is promoted to its parent by splitting the node into two nodes.



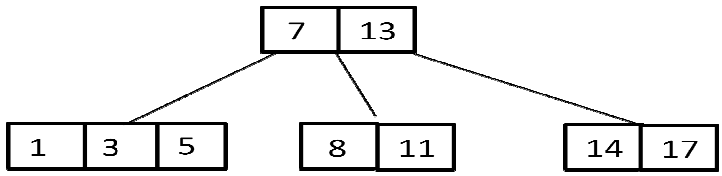


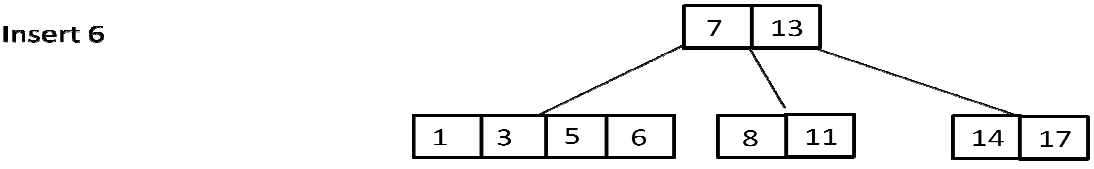
Locate the leaf node to inset 11. In the leaf node as there is room to store 11 it is inserted in ascending order.



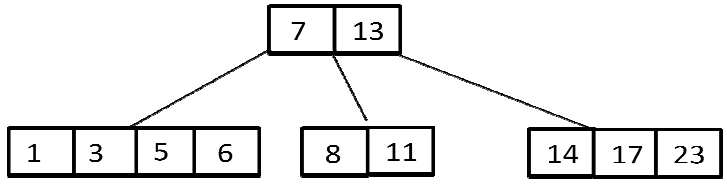


Locate the leaf node to inset 13. In the leaf node insert 13 there is no room so the middle value is promoted to its parent by splitting the node into two nodes.

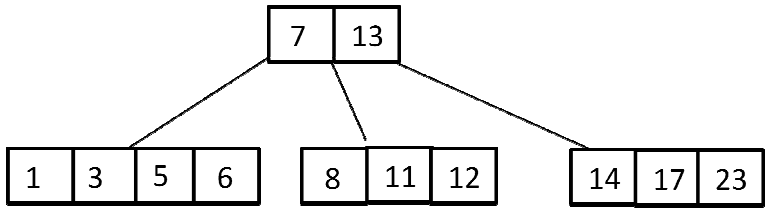




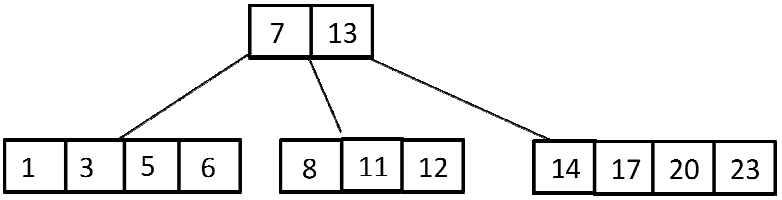
Insert 23



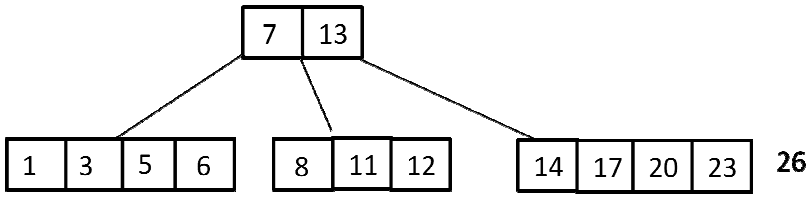
Insert 12



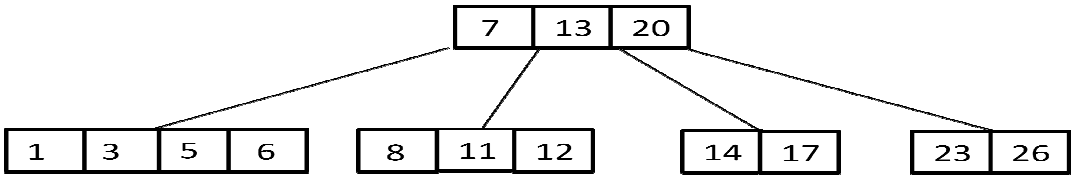
Insert 20



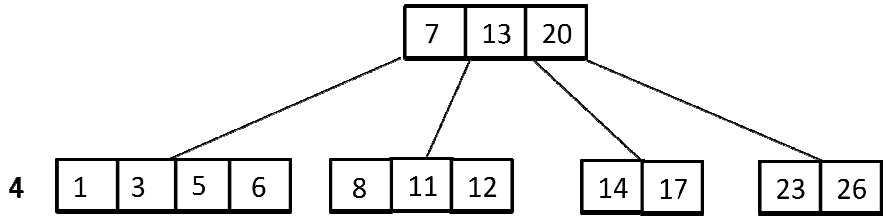
Insert 26



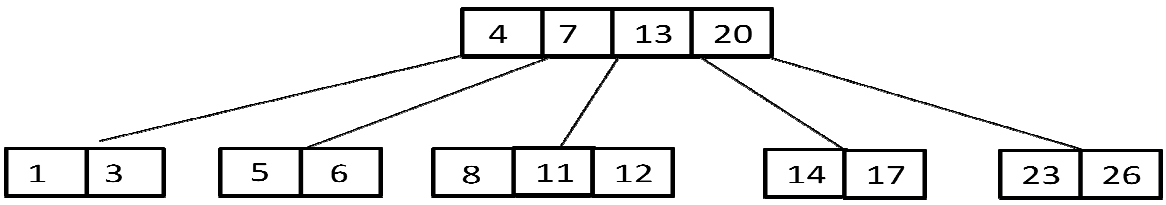
Locate the leaf node to inset 26. In the leaf node insert 26 there is no room so the middle value is promoted to its parent by splitting the node into two nodes.



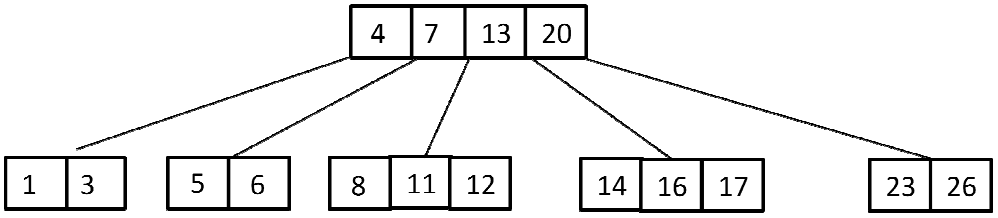
Insert 4



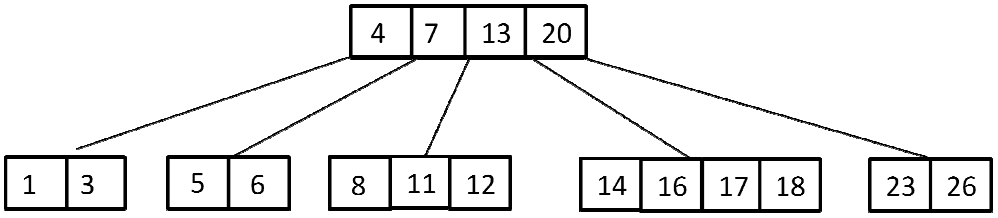
Locate the leaf node to inset 4. In the leaf node insert 4 there is no room so the middle value is promoted to its parent by splitting the node into two nodes.



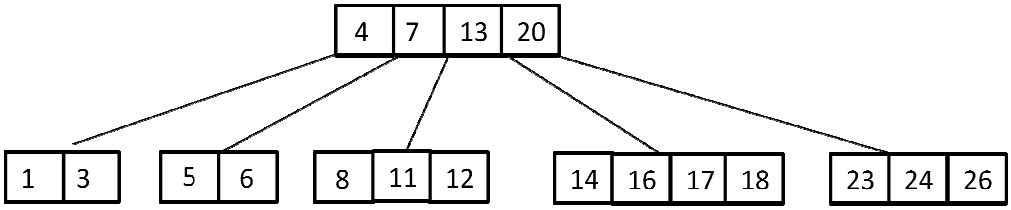
Insert 16



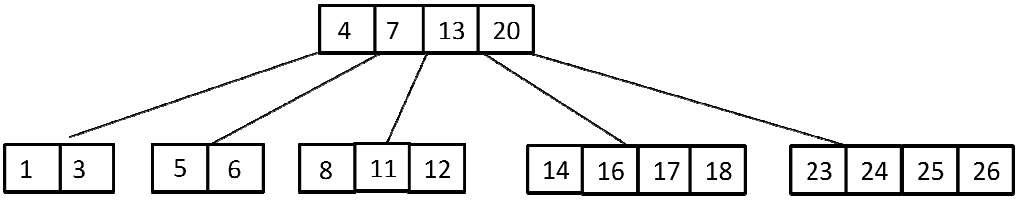
Insert 18



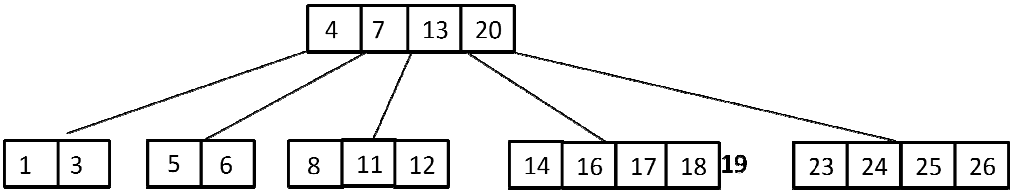
Insert 24



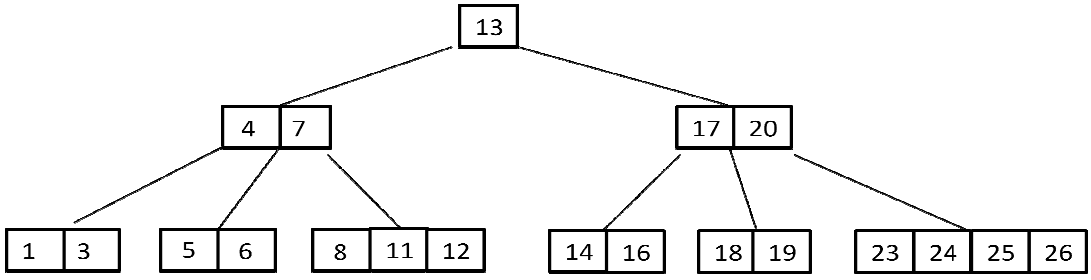
Insert 25



Insert 19



Locate the leaf node to inset 19. In the leaf node insert 19 there is no room so the middle value is promoted to its parent by splitting the node into two nodes.



28) In what way B+ trees are different from B Trees?

Here are some of the differences between B and B+ tree that I've come across:

1. In a **B tree**, search keys and data are stored in internal or leaf nodes. But in **B+-tree**, data is stored only in leaf nodes.
2. Searching any data in a **B+ tree** is very easy because all data are found in leaf nodes. In a **B tree**, data can be found in leaf nodes/internal nodes.
3. In a **B tree**, data may be found in leaf nodes or internal nodes. Deletion of internal nodes is very complicated. In a **B+ tree**, data is only found in leaf nodes. Deletion of leaf nodes is very easy as it can be directly deleted.
4. Insertion in **B tree** is more complicated than **B+ tree**.
5. **B+ trees** store redundant search key but **B tree** has no redundant value.
6. In a **B+ tree**, leaf nodes data are ordered as a sequential linked list but in **B tree** the leaf node cannot be stored using a linked list. Many database systems' implementations prefer the structural simplicity of a B+ tree.

The **basic difference** lies between how they make use of the **internal storage**.

29) Insert the below elements one by one by one into an initially empty B tree of order 3: i.e. 3 way search tree. 45, 35, 95, 96, 80, 70, 60, 50, 92, 75

30) What are different nodes in B+ trees? How they are corrected? Give examples.

The root of a B+ Tree represents the whole range of values in the tree, where every internal node is a subinterval.

In **B+-tree**, data is stored only in leaf nodes.

In a **B+ tree**, leaf nodes data are ordered as a sequential linked list
Searching any data in a **B+ tree** is very easy because all data are found in leaf nodes.

31) Search insert and delete algorithms of B+ Tree