CHAPTER 2

Efficient Search Trees

Objectives ...

- To study AVL Trees with its Operations
- To learn Red Black Trees
- To understand B and B+ Tree with its Operations

2.0 INTRODUCTION

- The efficiency of many operations on trees is related to the height of the tree - for example searching, inserting, and deleting.
- The efficiency of various operations on a Binary Search Tree (BST) decreases with increase in the differences between the heights of right sub tree and left sub tree of the root node.
- Hence, the differences between the heights of left sub tree and right sub tree should be kept to the minimum.
- Various operations performed on binary tree can lead to an unbalanced tree, in which either the height of left sub tree is much more than the height of right sub tree or vice versa.
- Such type of tree must be balanced using some techniques to achieve better efficiency level.
- The need to have balanced tree led to the emergence of another type of binary search tree known as height-balanced tree (also known as AVL tree) named after their inventor G. M. Adelson-Velsky and E. M. Landis.
- The AVL tree is a special kind of binary search tree, which satisfies the following two conditions:
  1. The heights of the left and right sub trees of a node differ by one.
  2. The left and right sub trees of a node (if exist) are also AVL trees.

2.1 TERMINOLOGY

- In 1962, Adelson-Velsky and Landis introduced a binary tree structure that is balanced with respect to the height of subtree. That is why it is called a height balanced tree.
- The basic objective of the height balanced tree is to perform the searching, insertion and deletion operations efficiently.
- A balanced binary tree is a binary tree in which the heights of two sub trees (left and right) of every node never differ by more than 1.

- A binary search tree is said to height balanced binary tree if all its nodes have a balance factor of 1, 0 or – 1 i.e.,

$$|h_L - h_R| \leq 1$$

Where, h_L and h_R are heights of left and right subtrees respectively.

1. Balance Factor:

[Oct. 17, April 19]

- The term Balancing Factor (BF) is used to determine whether the given binary search tree is balanced or not.
- The BF of a code is calculated as the height of its left sub tree minus the height of right sub tree i.e.,

Balance factor = $h_L - h_R$

- The balance factor of a node is a binary tree can have value +1, –1 or 0 depending on whether the height of its left sub tree is greater than or equal to the height of its right subtree.
- The balance factor of each node is indicated in the Fig. 2.1.
  - If balance factor of any node is +1, it means that the left sub-tree is one level higher than the right sub-tree.
  - If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
  - If balance factor of any node is –1, it means that the left sub-tree is one level lower than the right sub-tree.

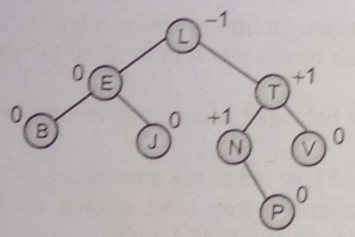


Fig. 2.1: Balance Factor in Binary Tree

2. AVL Tree:

- AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one.
- The technique of balancing the height of binary trees was developed by in the year 1962 by G. M. Adelson-Velsky and E. M. Landis in the year 1962 and hence given the short form as AVL tree Balanced Binary Tree.
- AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference (balance factor) is not more than 1.

Fig. 2.2 shows a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

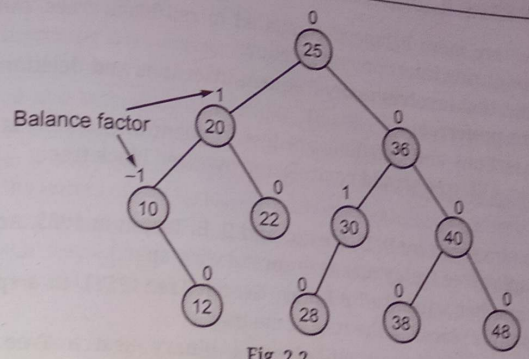


Fig. 2.2

3. Red Black Tree:

- A red black tree is a variant of Binary Search Tree (BST) in which an additional attribute, 'color' is used for balancing the tree. The value of this attribute can be either red or black.
- The red black trees are self-balancing binary search tree. In this type of tree, the leaf nodes are the NULL/NIL child nodes storing no data.
- In addition to the conditions satisfied by binary search tree, the following conditions/rules must also be satisfied for being a red black tree:
  - (i) Each and every node is either red or black.
  - (ii) The roof node and leaf nodes are always black in color.
  - (iii) If any node is red, then both its child nodes are black.
  - (iv) Each and every path from a given node to the leaf node contains same number of black nodes. The number of black nodes on such a path is known as black-height of a node.
- Fig. 2.3 shows a red back tree.

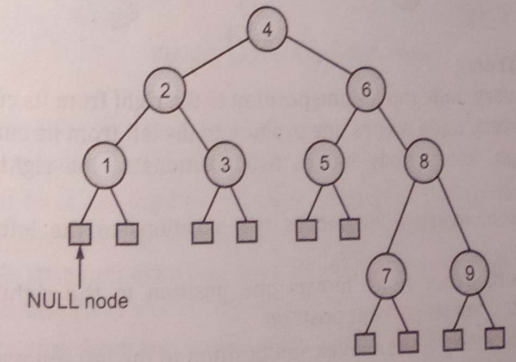


Fig. 2.3

- The AVL trees are more balanced compared to red black trees, but they may cause more rotations during insertion and deletion.
- So if the application involves many frequent insertions and deletions, then Red Black trees should be preferred.
- And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over red black tree.

4. Splay Tree:

- Splay tree was invented by D. D. Sleator and R. E. Tarjan in 1985. According them the tree is called splay tree (splay means to spread wide apart).
- Splay tree is another variant of a Binary Search Tree (BST). In a splay tree, recently accessed element is placed at the root of the tree.
- Splay Tree is a self-adjusted/self-balancing Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.
- All normal operations on a binary search tree are combined with one basic operation, called splaying.
- Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.
- Fig. 2.4 shows an example of splay tree.

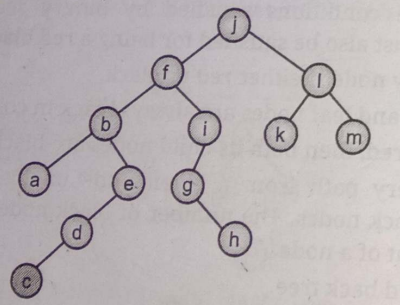


Fig. 2.4

Rotations in Splay Tree:

- In zig rotation, every node moves one position to the right from its current position.
- In zag rotation, every node moves one position to the left from its current position.
- In zig-zig rotation, every node moves two positions to the right from its current position.
- In zag-zag rotation, every node moves two positions to the left from its current position.
- In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position.
- In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

5. Lexical Search Tree (Trie):

- Instead of searching a tree using the entire value of a key, we can consider the key to be a sequence of characters, such as word or non-numeric identifier.
- When placed in a tree, each node has a place for each of the possible values that the characters in the lexical tree can assume.
- For example, if a key can contain the complete alphabet, each node has 26 entries one for each of the letters of the alphabet and known as a lexical 26-ary tree.
- Each and every entry in the lexical search tree contains a pointer to the nest level. In addition, each node of 26-ary tree contains 26 pointers, the first representing the letter A, the second the letter B, and so forth until the last pointer which represents the letter Z.
- Because each letter in the first level must point to a complete set of values, the second level contains 26*26 entries, one node of 26 entries for each of the 26 letters in the first level.
- At the third level 26*26*26 entries and finally we store the actual key at the leaf.
- If a key has three letters, these are at least three levels in the tree. If a key has ten letters, these are ten levels in the tree. Because of a lexical tree can contain many different keys, the largest word determines the height of the tree.
- Fig. 2.5 shows a lexical tree.

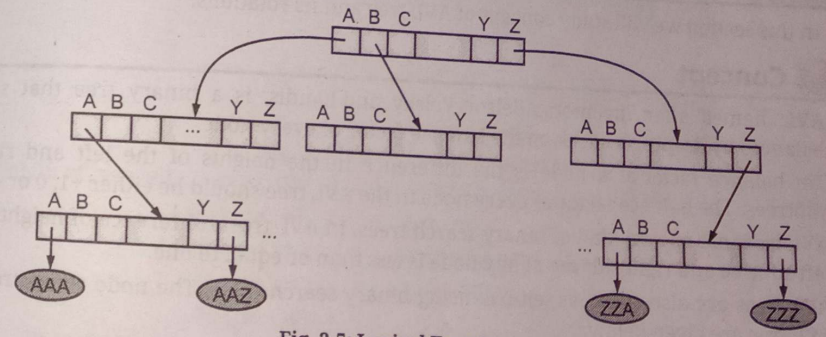


Fig. 2.5: Lexical Tree Structure

Trie:

- A trie is a lexical m-ary tree in which the pointers pointing to non-existing characters are replaced by null pointers.
- All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings.
- Trie is a data structure which is used to store the collection of strings and makes searching of a pattern in words more easy.
- The term trie came from the word retrieval. Trie is an efficient information storage and retrieval data structure.

- Trie data structure makes retrieval of a string from the collection of strings more easily.
- Trie is also called as Prefix Tree and sometimes Digital Tree. Trie is a tree like data structure used to store collection of strings.
- Fig. 2.6 shows an example of trie. Consider a list of strings Cat, Bat, Ball, Rat, Cap, Be.

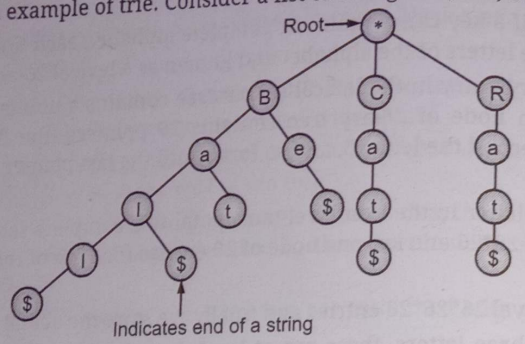


Fig. 2.6

2.2 AVL TREE

- In this section we will study concept of AVL tree and its rotations.

2.2.1 Concept

- AVL, named after inventors Adelson-Velsky and Landis, is a binary tree that self-balances by keeping a check on the balance factor of every node.
- The balance factor of a node is the difference in the heights of the left and right subtrees. The balance factor of every node in the AVL tree should be either +1, 0 or -1.
- AVL trees are special kind of binary search trees. In AVL trees, difference of heights of left subtree and right subtree of any node is less than or equal to one.
- AVL trees are also called as self-balancing binary search trees. The node structure of AVL tree are given below:

```
struct AVLNode
{
   int data;
   struct AVLNode *left, *right;
   int balfactor;
  };
```

- AVL tree can be defined as, let T be a non-empty binary tree with T_L and T_R as its left and right subtrees. The tree is height balanced if:
  - T_L and T_R are height balanced.
  - $h_L - h_R \leq 1$, where $h_L - h_R$ are the heights of T_L and T_R.

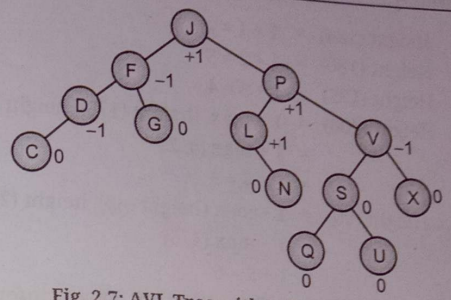


Fig. 2.7: AVL Tree with Balance Factors

- Balance factor of a node is the difference between the heights of the left and right subtrees of that node. Consider the following binary search tree in Fig. 2.8.

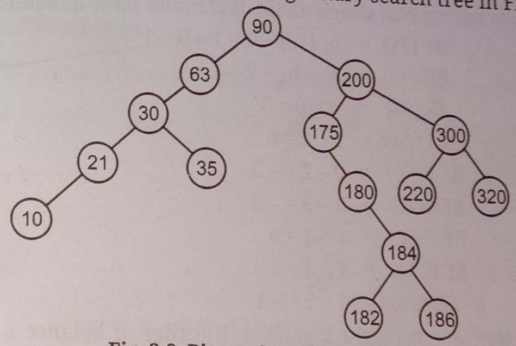


Fig. 2.8: Binary Search Tree (BST)

Height of tree with root 90 (90) = 1 + max (height (63), height (200))
Height of (63) = 1 + height (30)
Height (30) = 1 + max (height (21), height (35))
Height (21) = 1 + height (10)
Height (10) = 1

Therefore,

Height (21) = 1 + 1 = 2
Height (30) = 1 + max (2, 1) = 1 + 2 = 3
Height (63) = 1 + 3 = 4
Height (200) = 1 + max (height (175), height (300))
Height (175) = 1 + height (180)
Height (180) = 1 + height (184)
Height (184) = 1 + max (height (182), height (186))
Height (182) = 1
Height (186) = 1

Height (184) = 1 + 1 = 2
Height (180) = 1 + 2 = 3
Height (175) = 1 + 3 = 4
Height (200) = 1 + max. (height (175), height (300))
= 1 + max (4, 2)
= 1 + 4 = 5
Height (90) = 1 + max (height (63), height (200))
= 1 + max (4, 5)
= 1 + 5 = 6

- Thus, this tree has height 6. But from this we do not get any information about balance of height. The tree is said to be balanced if the difference in the right subtree and left subtree is not more than 1.
- Consider the above example in which all the leaf nodes have a balance factor of 0.

BF (21) = h_L (10) – 0 = 1 – 0 = 1
BF (30) = $h_L - h_R$ = 2 – 1 = 1
BF (63) = 3 – 0 = 3
BF (184) = 1 – 1 = 0
BF (180) = 0 – 2 = – 2
BF (175) = 0 – 3 = – 3
BF (300) = 1 – 1 = 0
BF (200) = 4 – 2 = 2
BF (90) = 4 – 5 = – 1

- Hence, the above tree is not height balanced. In order to balance a tree, we have to perform rotations on the tree.

2.2.2 Rotations

- Rotation is the process of moving nodes either to left or to right to make the tree balanced. To balance itself, an AVL tree may perform the following four kinds of rotations:
  1. Left Rotation (LL Rotation)
  2. Right Rotation (RR Rotation)
  3. Left-Right Rotation (LR Rotation)
  4. Right-Left Rotation (RL Rotation)
- The first two rotations are single rotations and the next two rotations are double rotations.

Single Left Rotation (LL Rotation):

- In LL Rotation, every node moves one position to left from the current position.
- To understand LL Rotation, let us consider the following insertion operation in AVL tree.

insert 1, 2 and 3

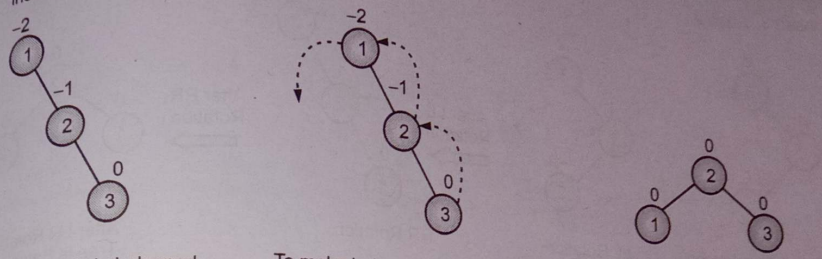


Fig. 2.9

Single Right Rotation (RR Rotation):

- In RR Rotation, every node moves one position to right from the current position.
- To understand RR Rotation, let us consider the following insertion operation in AVL tree.

insert 3, 2 and 1

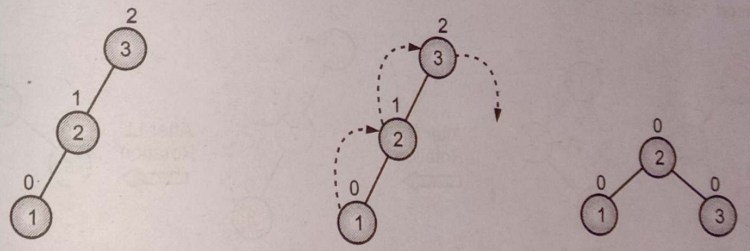


Fig. 2.10

Left Right Rotation (LR Rotation):

- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AVL tree.

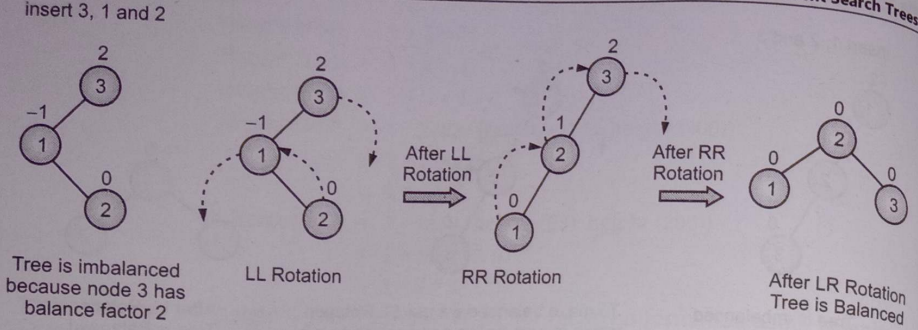


Fig. 2.11

Right Left Rotation (RL Rotation):

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- To understand RL Rotation, let us consider the following insertion operation in AVL tree.

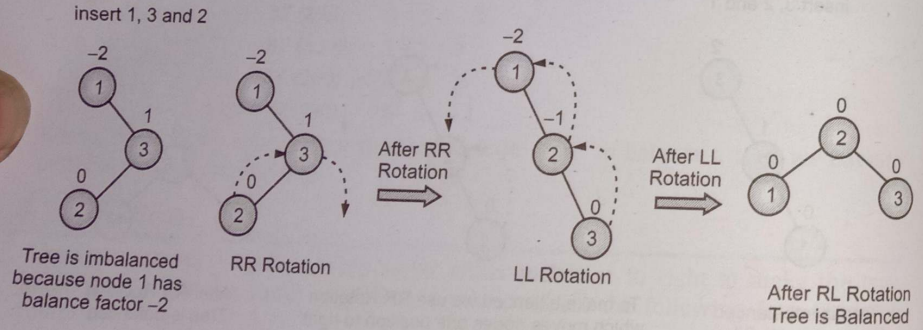
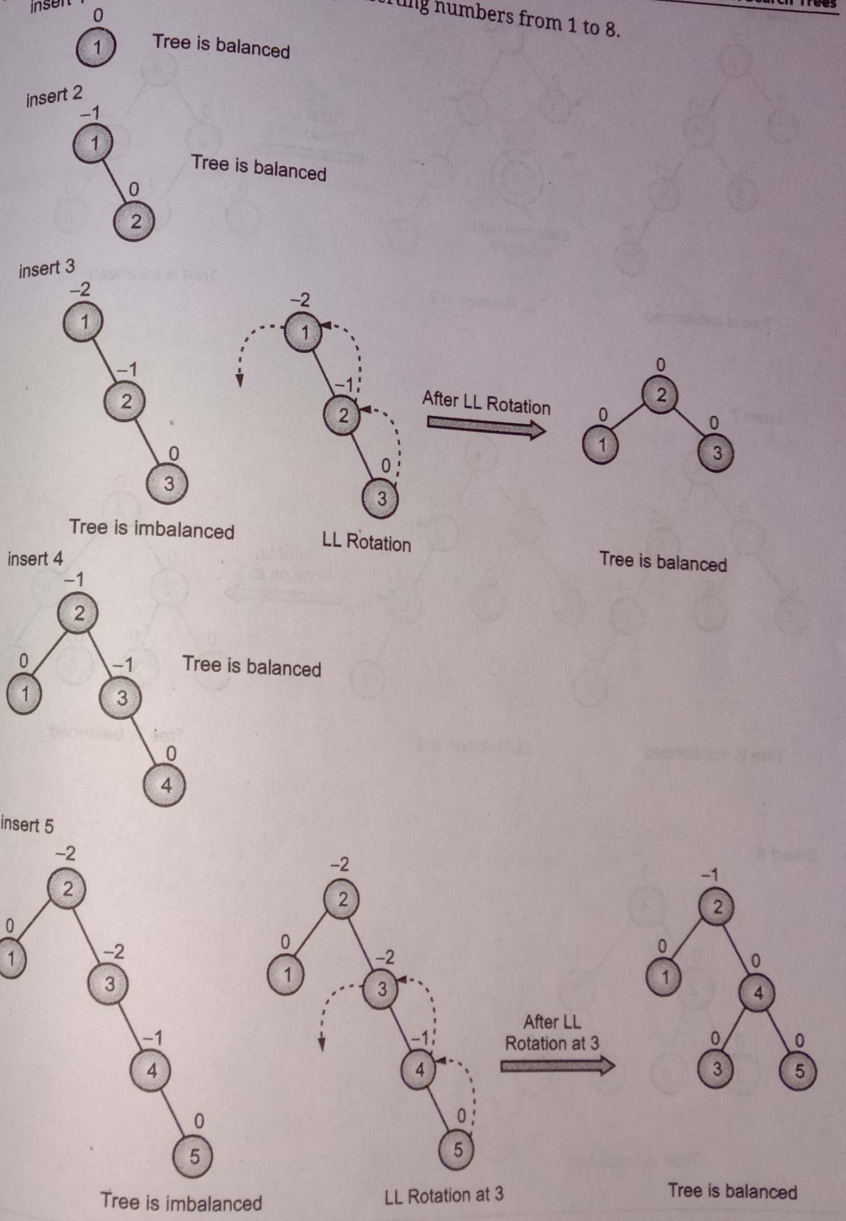


Fig. 2.12

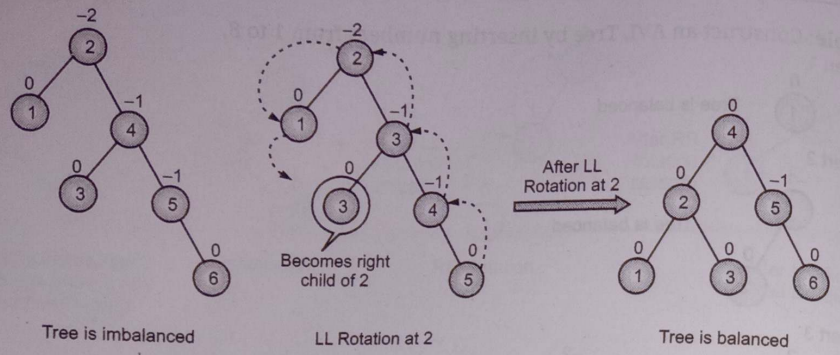
Operations on AVL Tree:

- *The operations are performed on AVL tree are search, insert and delete.*
- *The search operation in the AVL tree is similar to the search operation in a binary search tree. In AVL tree, a new node is always inserted as a leaf node.*
- *The deletion operation in AVL tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor (BF) condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree balanced.*

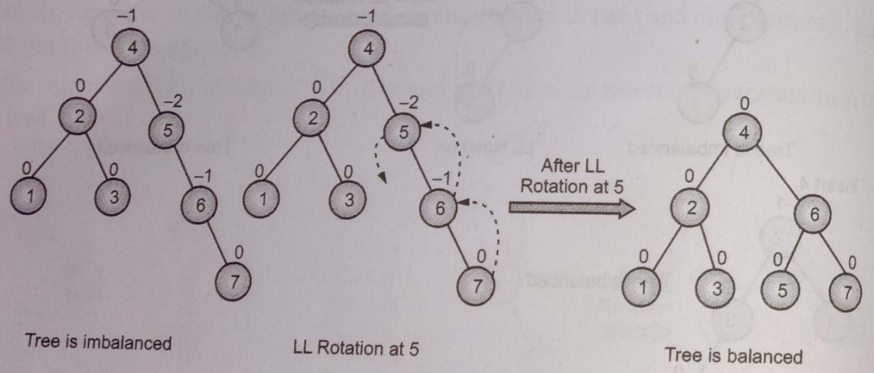
Example: Construct an AVL Tree by inserting numbers from 1 to 8.



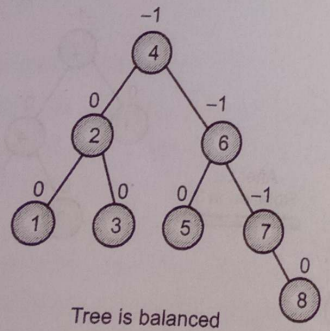
insert 6



insert 7



insert 8



Example: Delete the node 30 from the AVL tree shown in the Fig. 2.13.

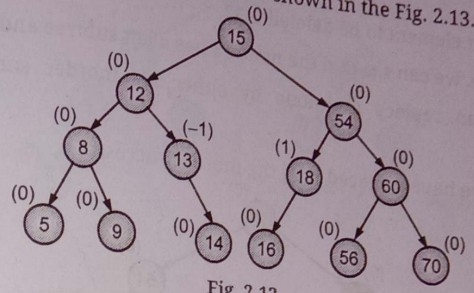


Fig. 2.13

Step 1:

- The node to be deleted from the tree is 8.
- If we observe it is the parent node of the node 5 and 9.
- Since the node 8 has two children it can be replaced by either of its child nodes.

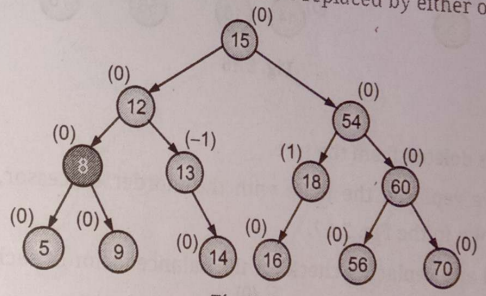


Fig. 2.14

Step 2:

- The node 8 is deleted from the tree.
- As the node is deleted we replace it with either of its children nodes.
- Here we **replaced** the node with the **inorder successor**, i.e. 9.
- Again we check the balance factor for each node.

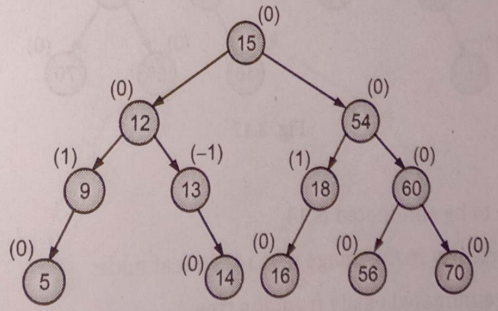


Fig. 2.15

Step 3:

- Now The next element to be deleted is 12.
- If we observe, we can see that the node 12 has a left subtree and a right subtree.
- We again can replace the node by either it's inorder successor or inorder predecessor.
- In this case we have replaced it by the inorder successor.

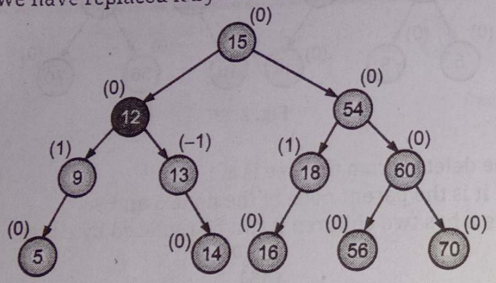


Fig. 2.16

Step 4:

- The node 12 is deleted from the tree.
- Since we have replaced the node with the inorder successor, the tree structure looks like shown in the Fig. 2.17.
- After removal and replacing check for the balance factor of each node of the tree.

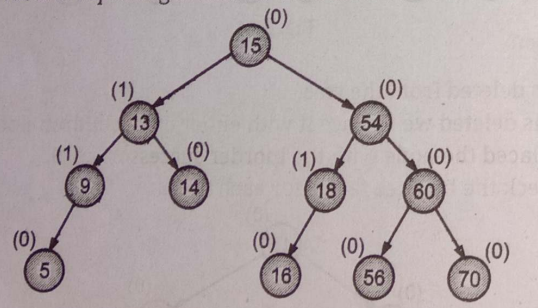


Fig. 2.17

Step 5:

- The next node to be eliminated is 14.
- It can be seen clearly in the image that 14 is a leaf node.
- Thus it can be eliminated easily from the tree.

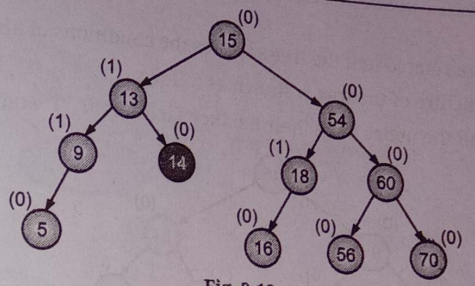


Fig. 2.18

Step 6:

- As the node 14 is deleted, we check the balance factor of all the nodes.
- We can see the balance factor of the node 13 is 2.
- This violates the terms of the AVL tree thus we need to balance it using the rotation mechanism.

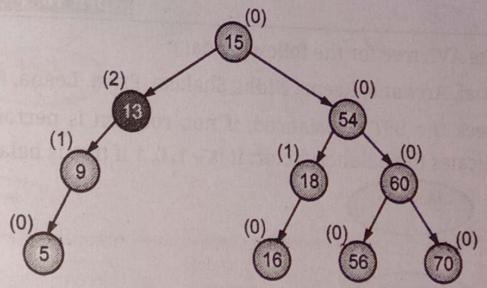


Fig. 2.19

Step 7:

- In order to balance the tree, we identify the rotation mechanism to be applied.
- Here, we need to use LL Rotation.
- The nodes involved in the rotation is shown in Fig. 2.20.

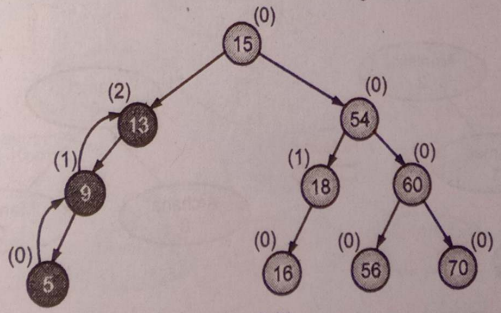


Fig. 2.20

Step 8:

- The nodes are rotated and the tree satisfies the conditions of an AVL tree.
- The final structure of the tree is shown as follows.
- We can see all the nodes have their balance factor as '0', '1' and '-1'.

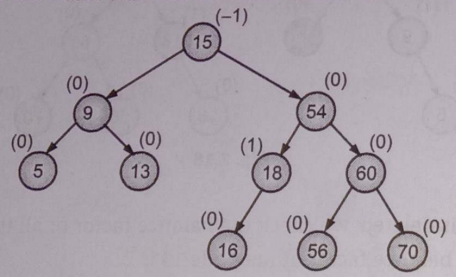


Fig. 2.21

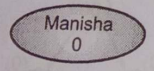
Examples: [April 16, 17, 18, 19, Oct. 17, 18]

Example 1: Create AVL tree for the following data:

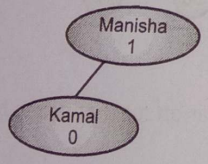
Manisha, Kamal, Archana, Reena, Nidhi, Shalaka, Priya, Leena, Meena

Solution: We check the BST is balanced, if not, rotation is performed. The number within each node indicates the balance factor. It is – 1, 0, 1 if tree is balanced.

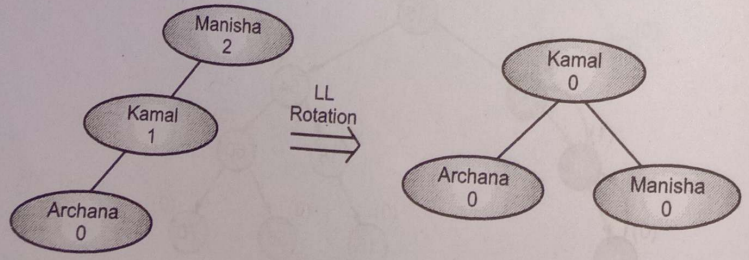
(i) Manisha



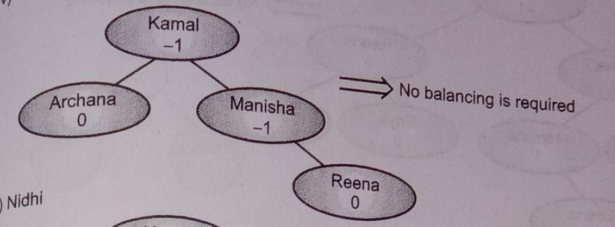
(ii) Kamal



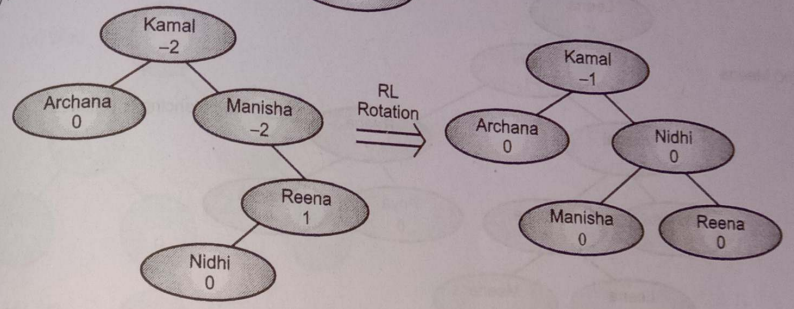
(iii) Archana



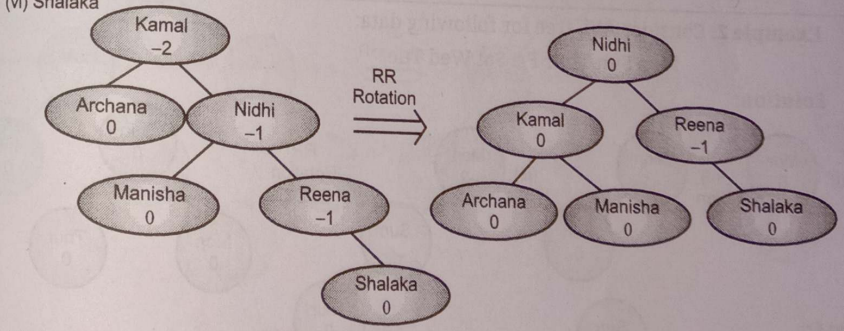
(iv) Reena



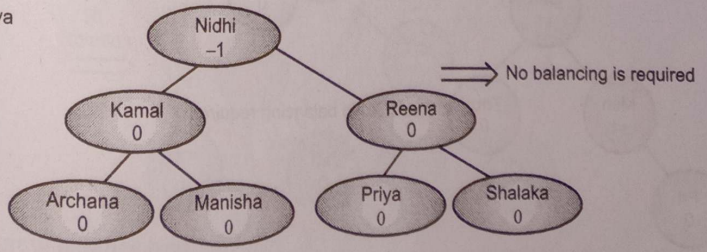
(v) Nidhi



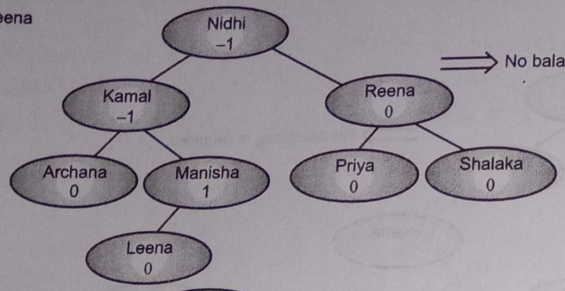
(vi) Shalaka



(vii) Priya

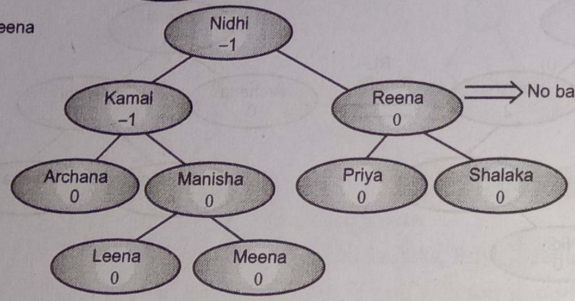


(viii) Leena



⟹ No balancing is required

(ix) Meena



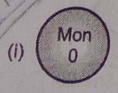
⟹ No balancing is required

Example 2: Consider AVL tree for following data:

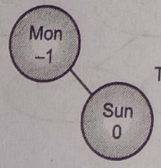
Mon Sun Thur Fri Sat Wed Tue

Solution:

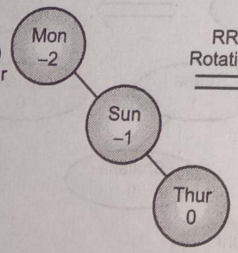
(i)



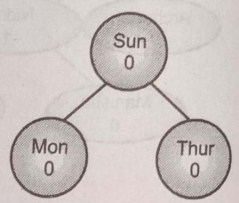
(ii) Sun



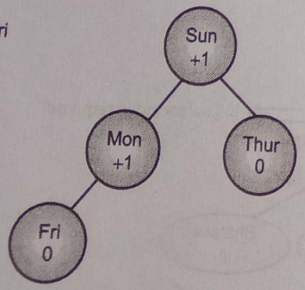
(iii) Thur



RR Rotation ⟹

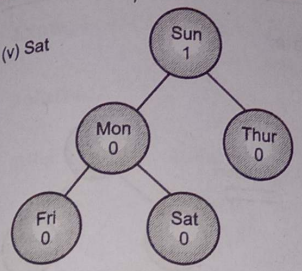


(iv) Fri

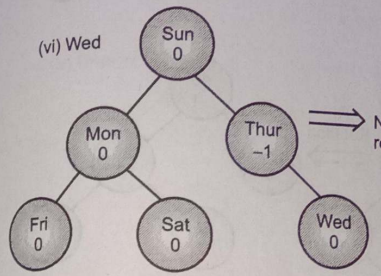


⟹ No balancing required

(v) Sat

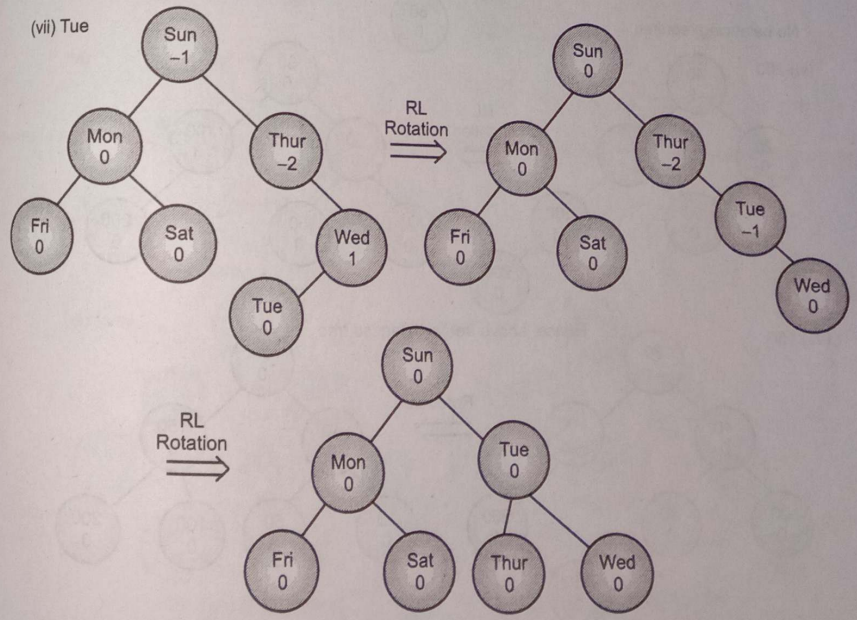


(vi) Wed



⟹ No balancing required

(vii) Tue

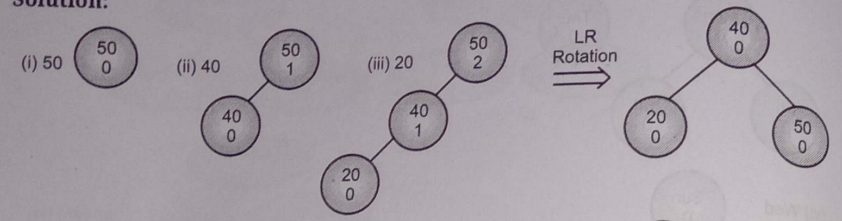


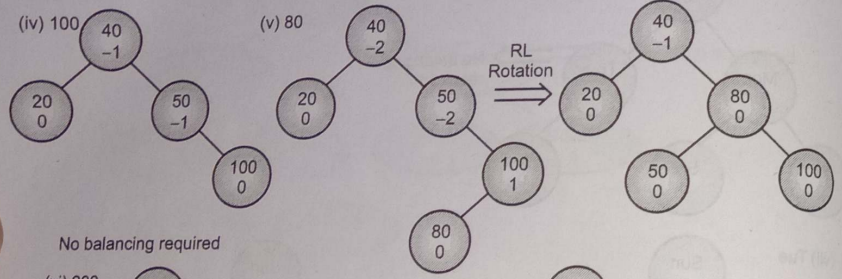
This is now balance tree.

Example 3: Construct an AVL tree for the following data:

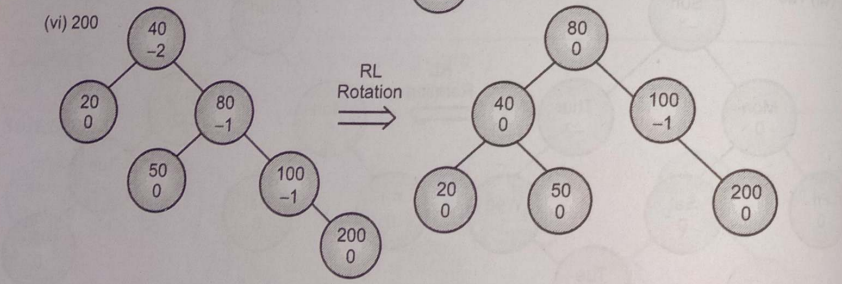
50, 40, 20, 100, 80, 200, 150

Solution:

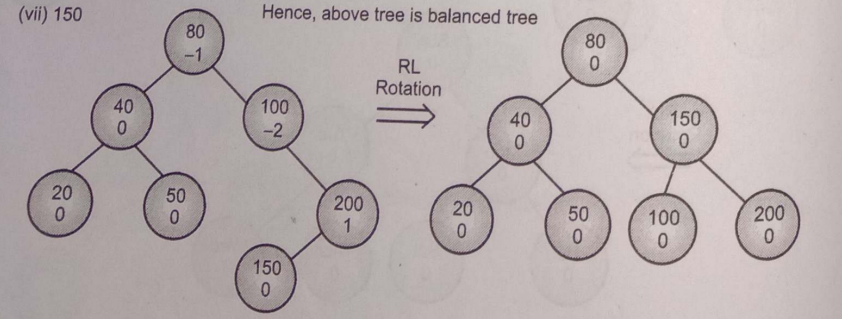




No balancing required



Hence, above tree is balanced tree

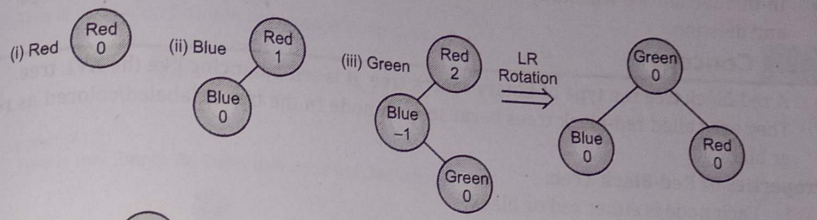


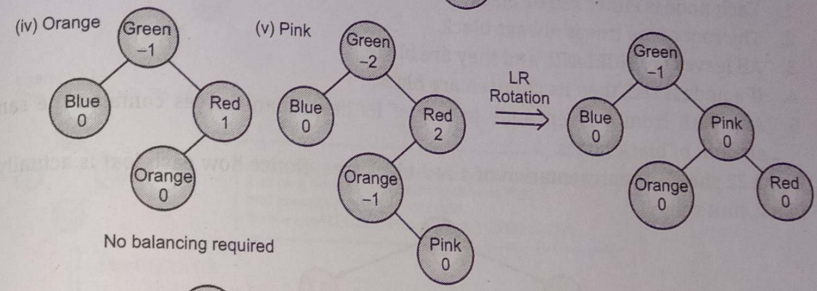
Hence, above tree is balanced tree.

Example 4: Construct AVL tree for following data:

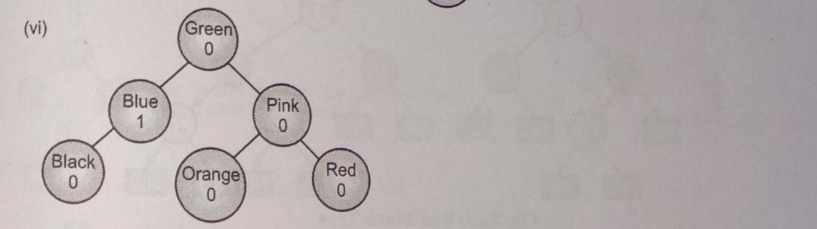
Red, blue, green orange, pink, black, grey, white, violet

Solution:

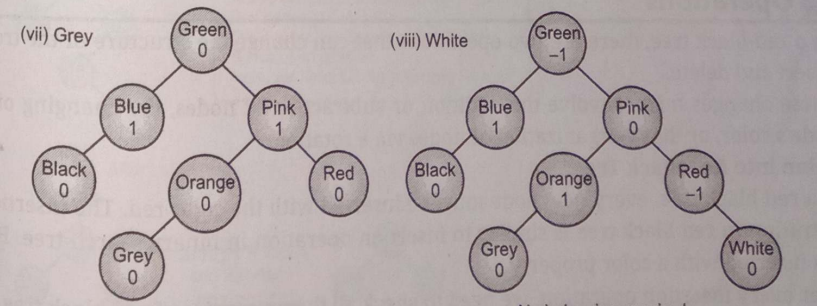




No balancing required



No balancing required



No balancing required

No balancing required

Hence, the tree is balance tree.

2.3 RED BLACK TREE

- In this section we will study concept of red black tree and its operations like insertion and deletion.

2.3.1 Concept

- A red-black tree is a type of binary search tree. It is self-balancing like the AVL tree.
- They are called red-black trees because each node in the tree is labeled/colored as red or black.

Properties of Red-Black Tree:

1. Each node is either red or black.
2. The root of the tree is always black.
3. All leaves are NULL/NIL and they are black.
4. If a node is red, then its children are black.
5. Any path from a given node to any of its descendant leaves contains the same amount of black nodes.

- Fig. 2.22 shows a representation of a red-black tree. Notice how each leaf is actually a black, null value.

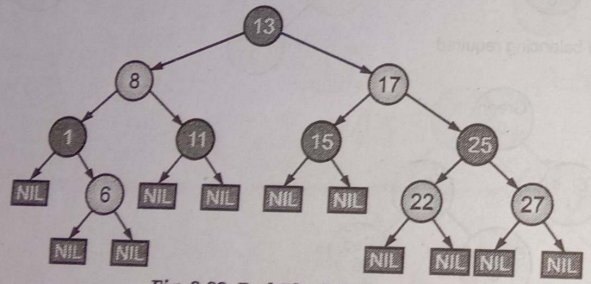


Fig. 2.22: Red-Black Tree

2.3.2 Operations

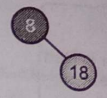
- In a red-black tree, there are two operations that can change the structure of the tree, insert and delete.
- These changes might involve the addition or subtraction of nodes, the changing of a node's color, or the re-organization of nodes via a rotation.

Insertion into Red Black Tree:

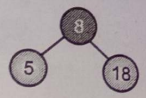
- In a red black tree, every new node must be inserted with the color red. The insertion operation in red black tree is similar to insertion operation in binary search tree. But it is inserted with a color property.
- After every insertion operation, we need to check all the properties of red-black tree. If all the properties are satisfied, then we go to next operation otherwise we perform the operations like recolor, rotation, rotation followed by recolor to make it red black tree.

insert (8)
Tree is Empty. So insert new node as Root node with black color.

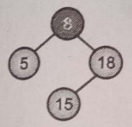
insert (18)
Tree is not Empty. So insert new node with red color.



insert (5)
Tree is mat Empty. So insert new node with red color.

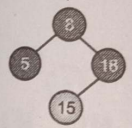


insert (15)
Tree is not Empty. So insert new node with red color.



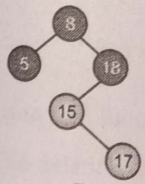
Here there are two consecutive Red nodes (18 and 15).
The new node's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR



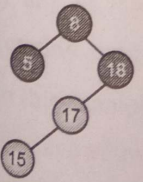
After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (17)
Tree is not Empty. So insert new node with red color.

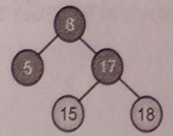


Here there are two consecutive
Red nodes (15 and 17).
The new node's parent sibling is NULL.
So we need rotation.
Here. we need LR Rotation and Recolor

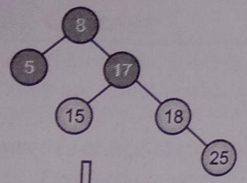
After Left Rotation



After Right Rotation and Recolor

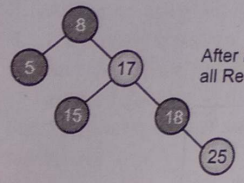


insert (25)
Tree is not Empty. So insert new node with red color.



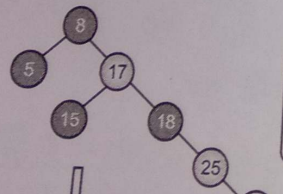
Here there are two consecutive
Red nodes (18 and 25),
The new node's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



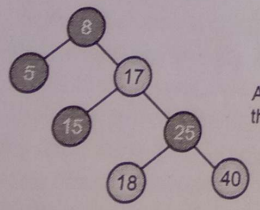
After Recolor operation, the tree is satisfying
all Red Black Tree properties.

insert (40)
Tree is not Empty. So insert new node with red color.



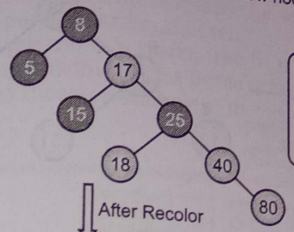
Here there are two consecutive
Red nodes (25 and 40).
The new node's parent sibling is NULL
So we need a Rotation and Recolor.
Here, we use LL Rotation and Recheck

After LL Rotation
and Recolor



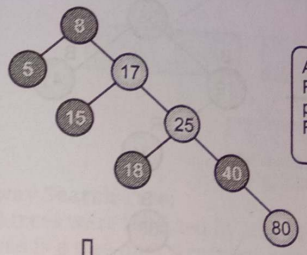
After LL Rotation and Recolor operation,
the tree is satisfying all Red Black Tree properties.

Insert (80)
Tree is not Empty. So insert new node with red color.



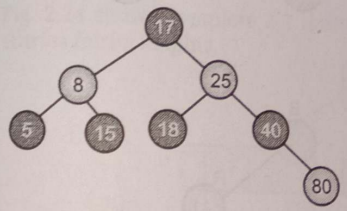
Here there are two consecutive
Red nodes (40 and 80). The new node's
parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor again there are two consecutive
Red nodes (17 and 25). The new node's
parent sibling color is Black. So we need
Rotation. We use Left Rotation and Recolor.

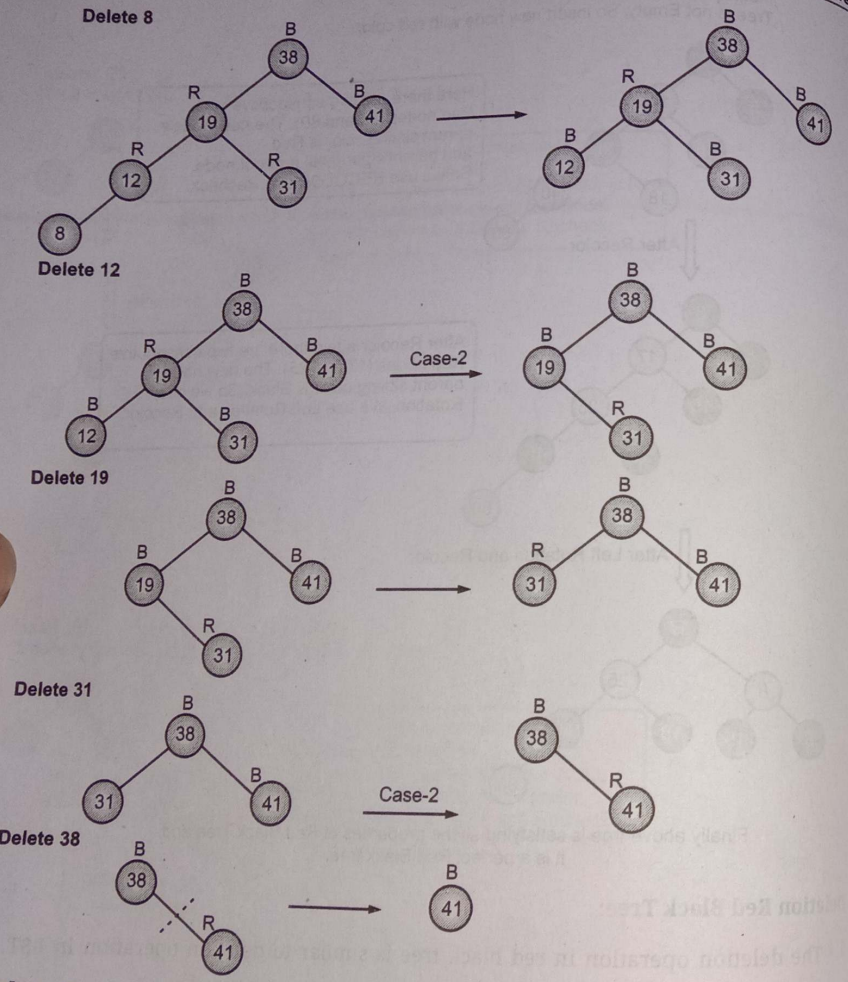
After Left Rotation and Recolor



Finally above tree is satisfying all the properties of Red Black Tree and
it is a perfect Red Black tree.

Deletion Red Black Tree:

- The deletion operation in red black tree is similar to deletion operation in BST. but after every deletion operation, we need to check with the red-black tree properties.
- If any of the properties are violated then make suitable operations like recolor, rotation and rotation followed by recolor to make it red-black tree.
- In this example, we show the red black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.



Delete 41

No/Empty Tree.

2.4 MULTIWAY SEARCH TREE

- *A multiway tree is a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way search tree).*

- A multiway tree can have more than one value/child per node. They are written as m-way trees where the m means the order of the tree. A multiway tree can have m-1 values per node and m children.
- An m-way tree is a search tree in which each node can have from 0 to m subtrees, where m is defined as the B-tree order.
- Fig. 2.23 shows an example of m-way tree of order 4.

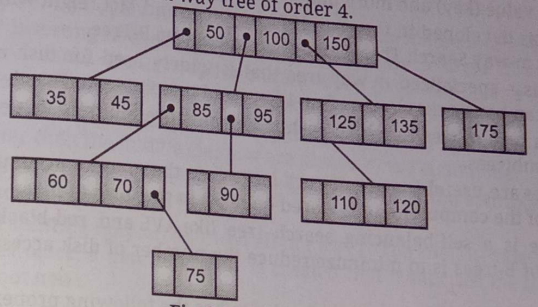


Fig. 2.23: Four-way Tree

2-3 Multi-way Search Tree:

- The 2–3 trees were invented by John Hopcroft in 1970. A 2-3 tree is a B-tree of order 3.
- A 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements.
- Fig. 2.24 shows complete 2-3 multi-way search tree. It has the maximum number of entries for its heights.

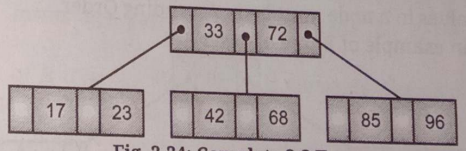


Fig. 2.24: Complete 2-3 Tree

2-3-4 Multi-way Search Tree:

- A 2-3-4 tree is a B-tree of order 4. It is also called as called a 2–4 tree.
- In 2-3-4 tree every node with children (internal node) has either two, three, or four child nodes.
- Fig. 2.25 shows 2-3-4 tree.

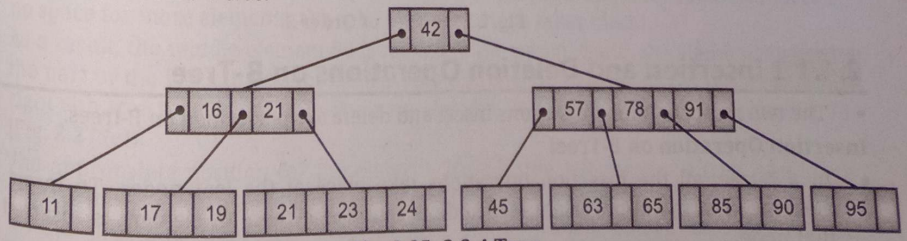


Fig. 2.25: 2-3-4 Tree

2.4.1 B-Tree

- In search trees like binary search tree, AVL tree, red-black tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called B-tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.
- A B-tree is a specialized m-way tree that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.
- B-tree is a type of tree in which each node can store multiple values and can point to multiple subtrees.
- The B-trees are useful in case of very large data that cannot accommodated the main memory of the computer and is stored on disks in the form of files.
- The B-tree is a self-balancing search tree like AVL and red-black trees. The main objective of B-trees is to minimize/reduce the number of disk accesses for accessing a record.
- Every B-tree has an order. B-tree of order m has the following properties:
  1. All leaf nodes must be at same level.
  2. All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
  3. All non-leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
  4. If the root node is a non-leaf node, then it must have at least two children.
  5. A non-leaf node with n-1 keys must have n number of children.
  6. All the key values in a node must be in Ascending Order.
- Fig. 2.26 shows an example of B-tree of order 5.

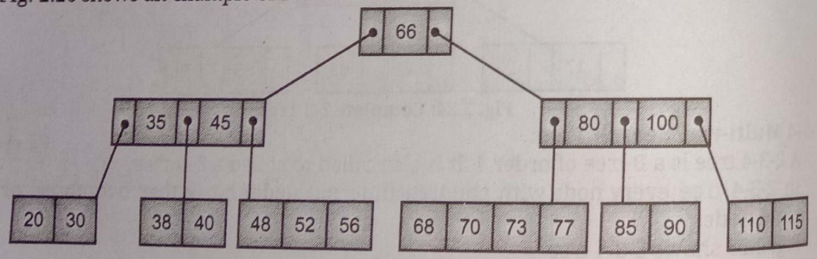


Fig. 2.26: B-Tree of Order 5

2.4.1.1 Insertion and Deletion Operations on B-Tree

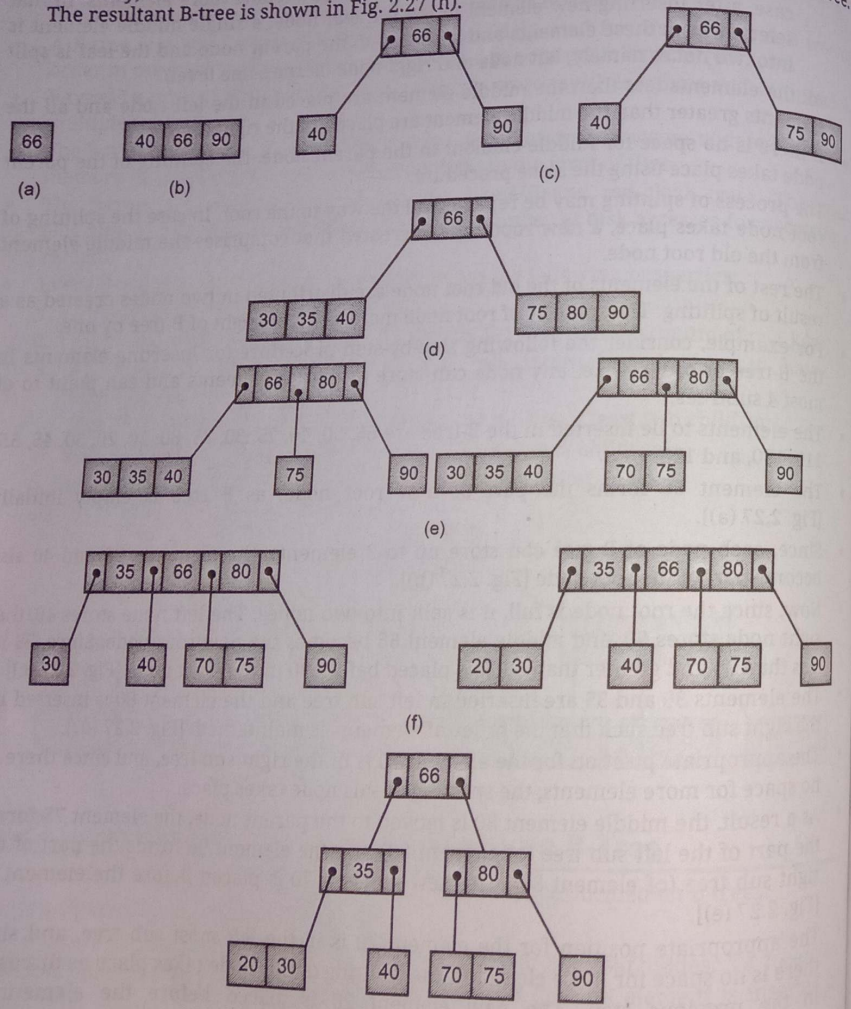
- The two most common operations insert and delete are performed on B-trees.

Insertion Operation on B-Tree:

- In a B-tree, all the insertion operations take place at the leaf nodes. To insert an element, first the appropriate leaf node is found and then an element is inserted in that node.

- Now, while inserting the element in the searched leaf node following one of the two cases may arise:
  1. There may be a space in the leaf node to accommodate more elements. In that case, the element is simply added to the node in such a way that the order of elements is maintained.
  2. There may not be a space in the leaf node to accommodate more elements. In that case, after inserting new element in the full leaf node, a single middle element is selected from these elements and is shifted to the parent node and the leaf is split into two nodes namely, left node and right node (at the same level).
- All the elements less than the middle element are placed in the left node and all the elements greater than the middle element are placed in the right node.
- If there is no space for middle element in the parent node, the splitting of the parent node takes place using the same procedure.
- The process of splitting may be repeated all the way to the root. In case the splitting of root node takes place, a new root node is created that comprises the middle element from the old root node.
- The rest of the elements of the old root node are distributed in two nodes created as a result of splitting. The splitting of root node increases the height of B-tree by one.
- For example, consider the following step-by-step procedure for inserting elements in the B-tree of order 4, i.e. any node can store at most 3 elements and can point to at most 4 subtrees.
- The elements to be inserted in the B-tree are 66, 90, 40, 75, 30, 35, 80, 70, 20, 50, 45, 55, 110, 100, and 120.
- The element 66 forms the part of new root node, as B tree is empty initially [Fig. 2.27 (a)].
- Since, each node of B tree can store up to 3 elements, the elements 90 and 40 also become part of the root node [Fig. 2.27 (b)].
- Now, since the root node is full, it is split into two nodes. The left node stores 40, the right node stores 90, and middle element 66 becomes the new root node. Since, 75 is less than 90 and greater than 66, it is placed before 90 in the right node [Fig. 2.27 (c)].
- The elements 30 and 35 are inserted in left sub tree and the element 80 is inserted in the right sub tree such that the order of elements is maintained [Fig. 2.27 (d)].
- The appropriate position for the element 70 is in the right sub tree, and since there is no space for more elements, the splitting of this node takes place.
- As a result, the middle element 80 is moved to the parent node, the element 75 forms the part of the left sub tree (of element 80) and the element 90 forms the part of the right sub tree (of element 80). The new element 70 is placed before the element 75 [Fig. 2.27 (e)].
- The appropriate position for the element 20 is in the left most sub tree, and since there is no space for more elements, the splitting of this node takes place as discussed in the previous step. The new element 20 is placed before the element 30 [Fig. 2.27 (f)].

- This tree can be used for future insertions, but a situation may arise when any of the sub trees splits and it will be required to adjust the middle element from that sub tree to the root node where there is no space for more elements.
- Hence, keeping in mind the future requirements, as soon as root node becomes full, splitting of root node must take place [Fig. 2.27 (g)]. This splitting of root node increases the height of tree by one.
- Similarly, other elements 50, 45, 55, 110, 100, and 120 can be inserted in this B-tree. The resultant B-tree is shown in Fig. 2.27 (h).



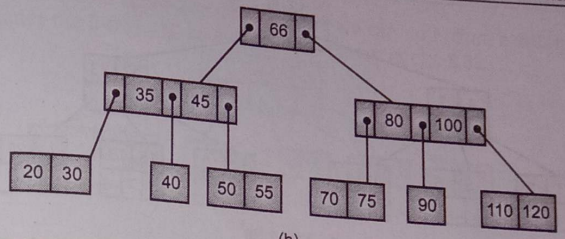


Fig. 2.27: Insertion Operation in B-Tree Deletion Operation on B-Tree:

- Deletion of an element from a B-tree involves following two steps:
  1. Searching the desired element and
  2. Deleting the element.
- Whenever, an element is deleted from *a* B-tree, it must be ensured that no property of B-tree is violated after the deletion.
- The element to be deleted may belong to either leaf node or internal node.
- Consider a B-tree in Fig. 2.28.

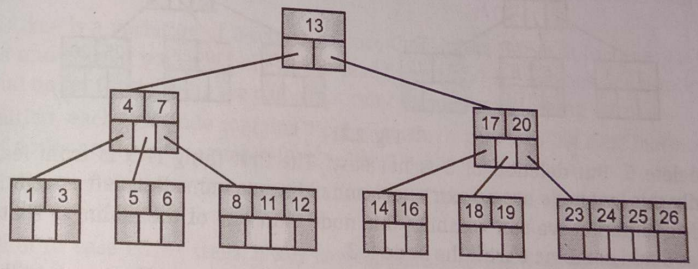


Fig. 2.28

- If we want to delete 8 then it is very simple (See Fig. 2.29).

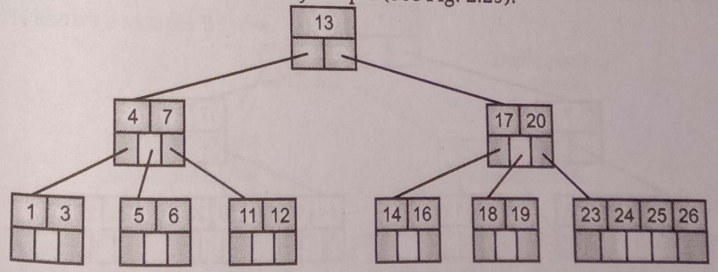


Fig. 2.29

- Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23. Hence, 23 will be moved upto replace 20.

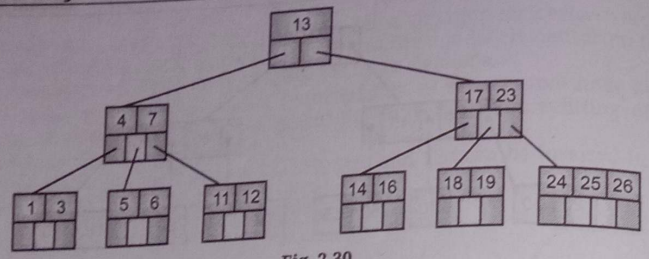


Fig. 2.30

- Next we will delete 18. Deletion of 18 from the corresponding node causes the node with only one key, which is not desired in B-tree of order 5.
- The slibling node to immediate right has an extra key. In such a case we can borrow a key from parent and move spare key of sibling to up (See Fig. 2.31).

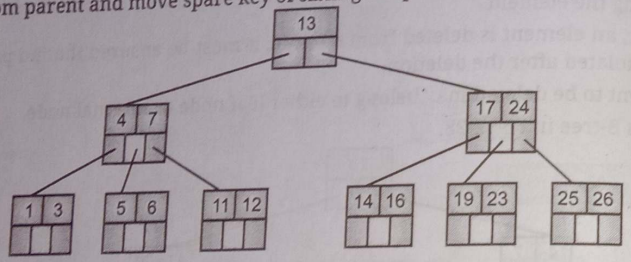


Fig. 2.31

- Now delete 5. But deletion of 5 is not easy. The first thing is 5 is from leaf node. Secondly this leaf node has no extra keys nor siblings to immediate left or right.
- In such a situation we can combine this node with one of the siblings. That means removes 5 and combine 6 with the node 1, 3.
- To make the tree balanced we have to move parent's key down. Hence, we will move 4 down as 4 is between 1, 3 and 6. The tree will be looks like as shown in Fig. 2.32.

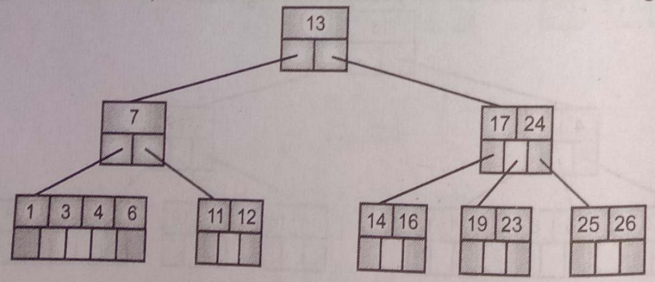


Fig. 2.32

- But again internal node of 7 contains only one key which not allowed in B-tree. We then will try to borrow a key from sibling.

- But sibling 17, 24 has no spare key. Hence, what we can do is that, combine 7 with 13 and 17, 24. Hence the B-tree will be looks like as shown in Fig. 2.33.

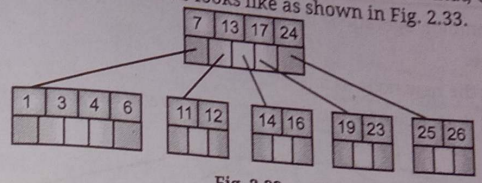


Fig. 2.33

2.4.2 B+ Tree

- B+ (B Plus) Tree is a balanced, multi-way binary tree. The B+ Trees are extended version of B-Trees.
- A B+ tree is an m-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.
- A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.
- The B+ tree is a variation of B-tree in a sense that unlike B-tree, it includes all the key values and record pointers only in the leaf nodes, and key values are duplicated in internal nodes for defining the paths that can be used for searching purposes.
- In addition, each leaf node contains a pointer, which points to the next leaf node, that is, leaf nodes are linked sequentially (See Fig. 2.34).
- Hence, B+ tree supports fast sequential access of records in addition to the random access feature of B-tree.
- Note that in case txt B+ trees, if key corresponding to the desired record is found in any internal node, the traversal continues until its respective leaf node is reached to access the appropriate record pointer.
- Fig. 2.34 shows a sample B+ tree.

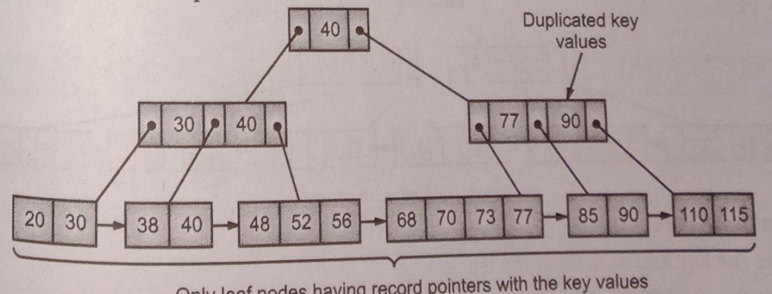


Fig. 2.34

4.2.2.1 Insertion and Deletion in B+ Tree

- The two most common operations insert and delete performed on B+ trees.

Insertion in B+ Tree:

- In B+ tree insert the new node as a leaf node.
- **Example:** Insert the value 195 into the B+ tree of order 5 shown in the Fig. 2.35.

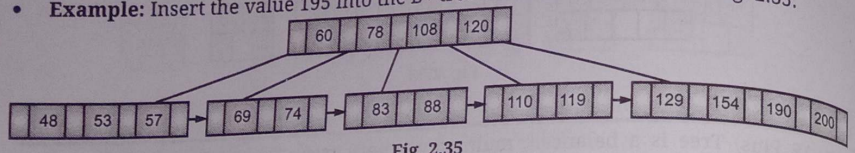


Fig. 2.35

- The value 195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position (See Fig. 2.36).

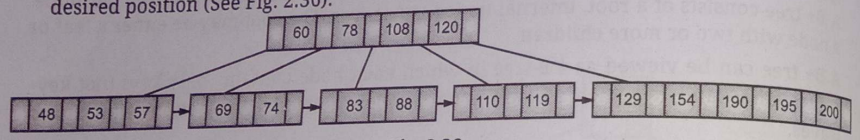


Fig. 2.36

- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.

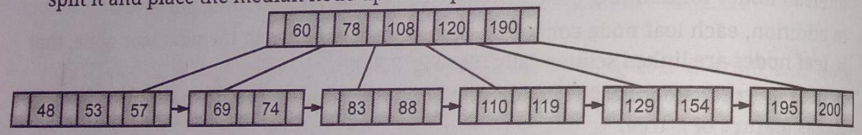


Fig. 2.37

- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown in Fig. 2.38.

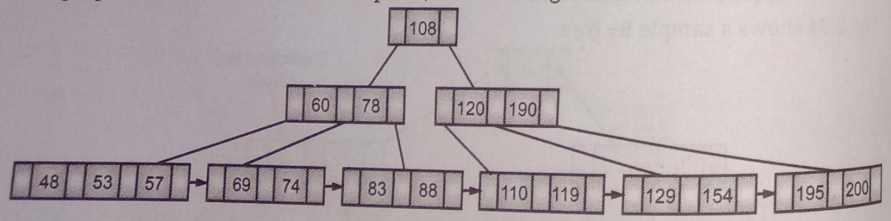


Fig. 2.38

Deletion in B+ Tree:

- *Delete the key and data from the leaves.*
- *Delete the key 200 from the B+ tree shown in the Fig. 2.39.*

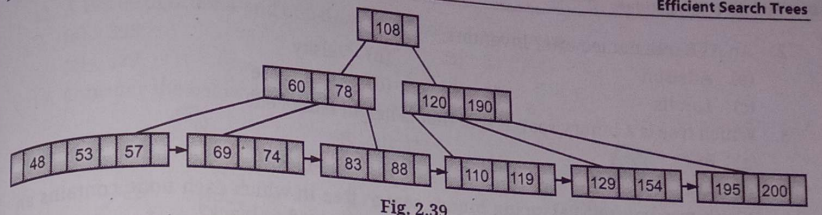


Fig. 2.39

- The value 200 is present in the right sub-tree of 190, after 195, delete it.

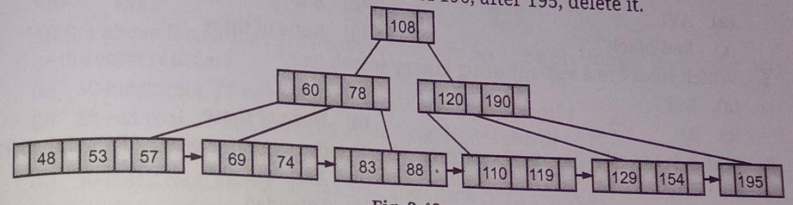


Fig. 2.40

- Merge the two nodes by using 195, 190, 154 and 129.

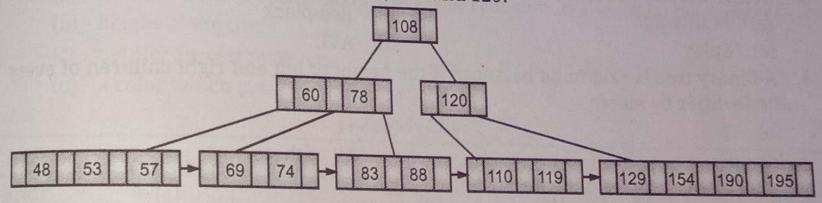


Fig. 2.41

- Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.
- Now, the height of B+ tree will be decreased by 1.

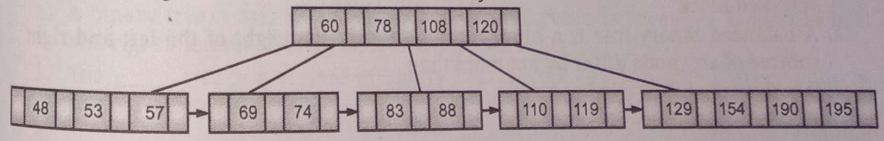


Fig. 2.42

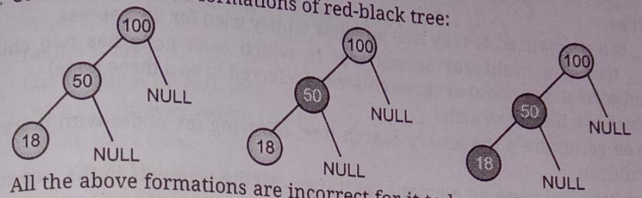
PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which factor on many tree operations related to the height of the tree?
   (a) Efficiency (b) Degree
   (c) Sibling (d) Path

2. An AVL tree named after inventors,
   (a) Adelson (b) Velsky
   (c) Landis (d) All of these
3. Which tree is a binary search tree that is height balanced?
   (a) BST (b) B+
   (c) AVL (d) Red-black
4. Which tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black?
   (a) AVL (b) BST
   (c) Red-black (d) None of these
5. Which trees are a self-adjusting binary search tree?
   (a) BST (b) Splay
   (c) B+ (d) None of these
6. Which is is a tree-like data structure whose nodes store the letters of an alphabet?
   (a) Trie (b) AVL
   (c) B+ (d) Extended
7. Which search tree is one with nodes that have two or more children?
   (a) Multiway (b) Red-black
   (c) Splay (d) AVL
8. A binary tree is said to be balanced if the height of left and right children of every node differ by either,
   (a) -1 (b) +1
   (c) 0 (d) All of these
9. A 2-3 tree is a B-tree of order,
   (a) 2 (b) 1
   (c) 3 (d) None of these
10. In which tree a new element must be added only at the leaf node.
    (a) B-Tree (b) AVL
    (c) Red-black (d) Splay
11. A balanced binary tree is a binary tree in which the height of the left and right subtree of any node differ by not more than _____.
    (a) 0 (b) 1
    (c) 3 (d) 2
12. In an AVL tree which factor is the difference between the height of the left subtree and that of the right subtree of that node.
    (a) Root (b) Node
    (c) Degree (d) Balance
13. Which of the following is the most widely used external memory data structure?
    (a) AVL tree (b) B-tree
    (c) Lexical tree (d) Red-black tree

14. A B-tree of order 4 and of height 3 will have a maximum of _____ keys.
    (a) 255 (b) 63
    (c) 127 (d) 188
15. Consider the below formations of red-black tree:



All the above formations are incorrect for it to be a red black tree. then what may be the correct order?
    (a) 50-black root, 18-red left subtree, 100-red right subtree
    (b) 50-red root, 18-red left subtree, 100-red right subtree
    (c) 50-black root, 18-black left subtree, 100-red right subtree
    (d) 50-black root, 18-red left subtree, 100-black right subtree
16. What is the special property of red-black trees and what root should always be?
    (a) a color which is either red or black and root should always be black color
    (b) height of the tree
    (c) pointer to next node
    (d) a color which is either green or black

Answers

1. (a)	2. (d)	3. (c)	4. (c)	5. (b)	6. (a)	7. (a)
8. (d)	9. (c)	10.(a)	11. (b)	12. (d)	13. (b)	14. (a)
15. (a)	16. (a)					

Q. II Fill in the Blanks:

1. Balancing or self-balancing (height balanced) tree is a _____ search tree.
2. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either _____.
3. The _____ tree is a variant of Binary Search Tree (BST) in which every node is colored either red or black.
4. _____ is a process in which a node is transferred to the root by performing suitable rotations.
5. _____ is a tree-based data structure, which is used for efficient retrieval of a key in a large data-set of strings.
6. _____ tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
7. In the _____ m-ary tree, the key is represented as a sequence of characters.

8. Trie is an efficient information _____ data structure.
9. A splay tree also known as _____ tree is a type of binary search tree which reorganizes the nodes of tree to move the most recently accessed node to the root of the tree.
10. A _____ is a specialized M-way tree which is widely used for disk access.
11. A _____ tree is a multi-way search tree in which each node has two children (referred to as a two node) or three children (referred to as a three node).
12. 2–3–4 trees are B-trees of order _____.
13. The B-tree generalizes the binary search tree, allowing for nodes with more than _____ children.
14. In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the _____ of the tree.
15. A red-black tree is a _____ search tree in which each node is colored either red or black.

Answers

1. binary	2. −1, 0 or +1	3. red black	4. Splaying	5. Trie	6. AVL
7. lexical	8. retrieval	9. self-adjusting	10. B-Tree	11. 2-3	12. 4
13. two	14. root	15. binary			

Q. III State True or False:

1. An AVL tree is a self-balancing binary search tree.
2. The self-balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.
3. An AVL tree is a binary search tree where the sub-trees of every node differ in height by at most 1.
4. B Tree is a self-balancing data structure based on a specific set of rules for searching, inserting, and deleting the data in a faster and memory efficient way.
5. Red-black Tree is a self-balancing Binary Search Tree (BST).
6. In a AVL tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called Splaying (the process of bringing it to the root position by performing suitable rotation operations).
7. A trie, also called digital tree or prefix tree used to store collection of strings.
8. To have an unbalanced tree, we at least need a tree of height 1.
9. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.
10. The B-tree is a generalization of a binary search tree in that a node can have more than two children.
11. B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

Answers

1. (T)	2. (T)	3. (T)	4. (T)	5. (T)	6. (F)	7. (T)
8. (F)	9. (T)	10. (T)	11. (T)			

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is height balance tree?
2. List operations on AVL tree.
3. What is Trie?
4. Define B-Tree.
5. List rotations on AVL tree.
6. Define multi-way search trees.
7. Define splay tree.
8. What is lexical search tree?
9. Define balance factor.

(B) Long Answer Questions:

1. Describe need for height balanced trees.
2. With the help of example describe concept of AVL tree.
3. Explain concept of red-black tree diagrammatically.
4. Write short note on: Trie.
5. Describe lexical search tree with example.
6. How to insert and delete operations carried by red-black tree? Explain with example.
7. Describe AVL tree rotations with example,
8. With explain B-Tree insert and delete operation.
9. Compare B-Tree and B+ Tree (any four points).
10. Write short note on: Splay tree.
11. With the help of example describe multi-way search tree.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Construct AVL tree for the following data: **[5 M]**
   Mon., Wed., Tue., Sat., Sun., Thur.

Ans. Refer to Section 2.2, Examples.

April 2017

1. Construct AVL tree for the following data: **[5 M]**
   Pen, Eraser, Book, Scale, Sketch pen, Crayon, Color pencil.

Ans. Refer to Section 2.2, Examples.

October 2017

1. Define balance factor. **[1 M]**

Ans. Refer to Section 2.1, Point (1).

2. Construct AVL tree for the following data: [5 M]
NFD, ZIM, IND, AUS, NEL, ENG, SRL, PAK.

Ans. Refer to Section 2.2, Examples.

April 2018

1. Construct AVL tree for the following data: [5 M]
SUN, FRI, MON, WED, TUE, THUR, SAT.

Ans. Refer to Section 2.2, Examples.

October 2018

1. Construct AVL tree for the following data: [5 M]
55, 40, 25, 100, 80, 200, 150.

Ans. Refer to Section 2.2, Examples.

April 2019

1. Define balance factor. [1 M]

Ans. Refer to Section 2.1, Point (1).

2. Construct the AVL tree for the following data: [5 M]
Chaitra, Magh, Vaishakh, Kartik, Falgun, Aashadh.

Ans. Refer to Section 2.2, Examples.

Syllabus ...

1. Tree **(10 Hrs.)**

1.1 Concept and Terminologies

1.2 Types of Binary Trees - Binary Tree, Skewed Tree, Strictly Binary Tree, Full Binary Tree, Complete Binary Tree, Expression Tree, Binary Search Tree, Heap

1.3 Representation - Static and Dynamic

1.4 Implementation and Operations on Binary Search Tree - Create, Insert, Delete, Search, Tree Traversals - Preorder, Inorder, Postorder (Recursive Implementation), Level-Order Traversal using Queue, Counting Leaf, Non-Leaf and Total Nodes, Copy, Mirror

1.5 Applications of Trees

1.5.1 Heap Sort, Implementation

1.5.2 Introduction to Greedy Strategy, Huffman Encoding (Implementation using Priority Queue)

2. Efficient Search Trees **(8 Hrs.)**

2.1 Terminology: Balanced Trees - AVL Trees, Red Black Tree, Splay Tree, Lexical Search Tree -Trie

2.2 AVL Tree - Concept and Rotations

2.3 Red Black Trees - Concept, Insertion and Deletion

2.4 Multi-Way Search Tree - B and B+ Tree - Insertion, Deletion

3. Graph **(12 Hrs.)**

3.1 Concept and Terminologies

3.2 Graph Representation - Adjacency Matrix, Adjacency List, Inverse Adjacency List, Adjacency Multi-list

3.3 Graph Traversals - Breadth First Search and Depth First Search (With Implementation)

3.4 Applications of Graph

3.4.1 Topological Sorting

3.4.2 Use of Greedy Strategy in Minimal Spanning Trees (Prim's and Kruskal's Algorithm)

3.4.3 Single Source Shortest Path - Dijkstra's Algorithm

3.4.4 Dynamic Programming Strategy, All Pairs Shortest Path - Floyd Warshall Algorithm

3.4.5 Use of Graphs in Social Networks

4. Hash Table **(6 Hrs.)**

4.1 Concept of Hashing

4.2 Terminologies - Hash Table, Hash Function, Bucket, Hash Address, Collision, Synonym, Overflow etc.

4.3 Properties of Good Hash Function

4.4 Hash Functions: Division Function, Mid Square, Folding Methods

4.5 Collision Resolution Techniques

4.5.1 Open Addressing - Linear Probing, Quadratic Probing, Rehashing

4.5.2 Chaining - Coalesced, Separate Chaining

■■■