# UNIT 2 AVL-TREE and B-TREE

## Structure



# 2.0 INTRODUCTION

In an earlier section on Binary Trees, we talked about perfectly balanced binary trees and 2-trees. The effectiveness of the searching process in a binary search tree depends on how data are organized to make up a specific tree. For example consider the two shapes given in figure 1(a) and 1(b).

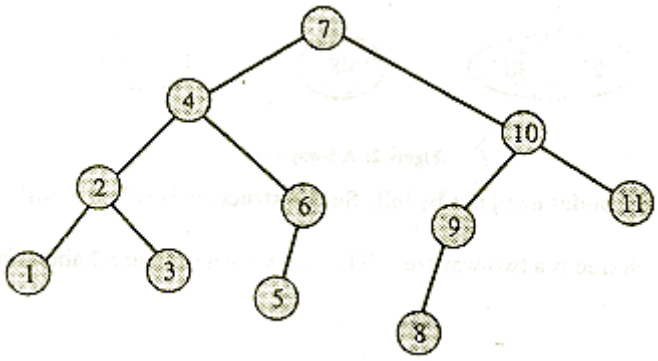



**Figure 1 (a): A nearly full binary tree**

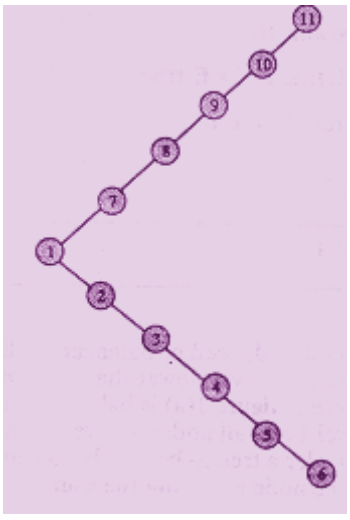



**Figure 1(b): A degenerate binary tree**

The efficiency of searches will be rather different in these two cases, although the same elements are organized in the two structures. The tree in figure 1(a) is rather short and compact while the tree in figure 1(b) is a long and thin tree. We may say that the tree in figure 1(a) is somewhat more balanced than that in figure 1(b).

In a binary search tree, each node holds a single value and has at most two branches. Those in the left branch have values less than the node value, while those in the right branch have values greater than the node value. This can be generalized by allowing more values at each node. For example, if we keep two values in each node, that means at most three branches, the descendants are split into three groups (maximum). The leftmost descendant will have the values less than the first value in the root, the middle descendant will have values between the two values in the root node and the rightmost descendant will have values grater than the second value in the root node. Let us understand it more clearly by the following figure (Figure 2).

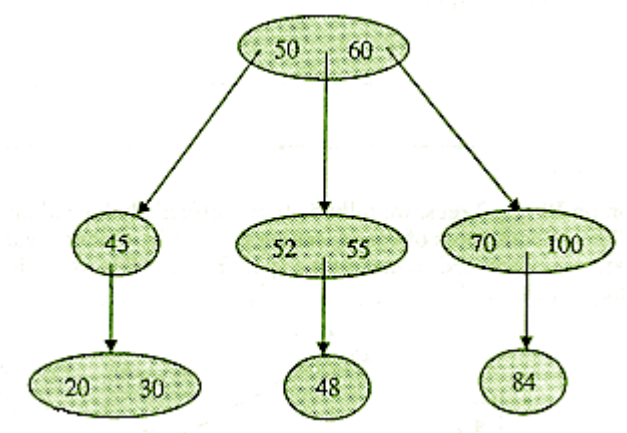



**Figure 2: A 3-way Tree**

Note that all the nodes need not be full. Such a structure is called a multiway tree.

A binary search tree is a two-way tree. The tree shown in figure 2 above is a 3-way tree.

# 2.1 OBJECTIVES

At the end of this unit, you shall be able, to

- define a multiway-tree, an AVL tree and a B-tree
- differentiate among different tree structures
- create AVL and B-Trees

# 2.2 HEIGHT BALANCED TREE

A binary tree of height h is completely balanced or balanced if all leaves occur at nodes of level h or h-1 and if all nodes at levels lower than h-1 have two. children. According to this definition, the tree in figure 1(a) is balanced, because all leaves occur at levels 3 considering at level 1 and all node's at levels 1 and 2 have two children. Intuitively we might consider a tree to be well balanced if, for each node, the longest paths from the left of the node are about the same length as the longest paths on the right.

More precisely, a tree is height balanced if, for each node in the tree, the height of the left subtree differs from the height, of the right subtree by no more than 1. The tree in figure 2(a) height balanced, but it is not completely balanced. On the other hand, the tree in figure 2(b) is completely balanced tree.

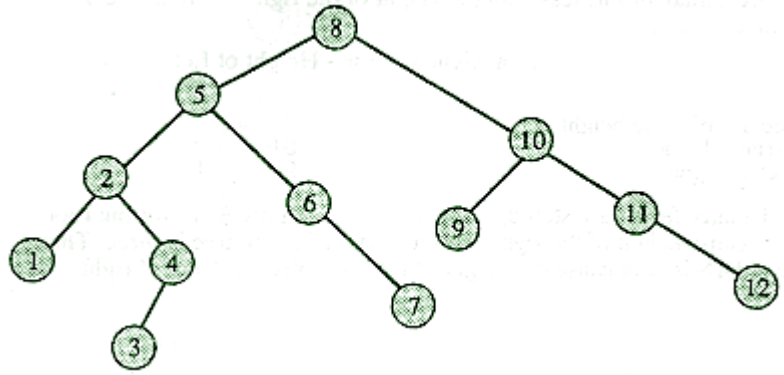



**Figure 2(a)**

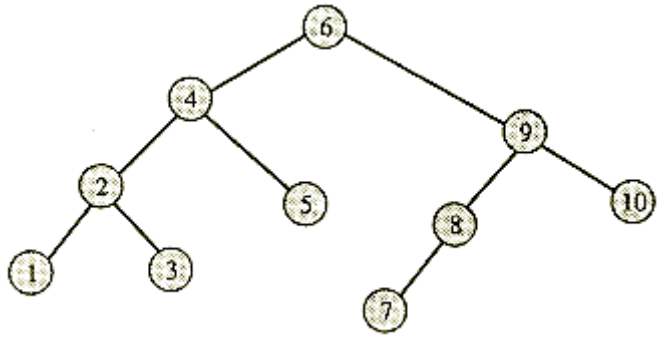



**Figure 2(b)**

An almost height balanced tree is called an AYL tree after the Russian mathematician G. M. Adelson - Velskii and E. M. Lendis, who first defined and studied this form of a tree. AVL Tree may or may not be perfectly balanced.

Let us determine how many nodes might be there in a balanced tree of height h.

The, root will be the only node at level 1;

Each subsequent levels will be as full as possible i.e. 2 nodes at level 2, 4 nodes at level 3 and so on, i.e. in general there will be $2^{1-1}$ nodes at level 1. Therefore the number of nodes from level 1 through level h-1 will be

$$1 + 2 + 2^2 + 2^3 + ... + 2^{h-2} = 2^{h-1} - 1$$

The number of nodes at level h may range from a single node to a maximum of $2^{h-1}$ nodes. Therefore, the total number of nodes n of the tree may range for $(2^{h-1}-1+1)$ to $(2^{h-1}-1+2^{h-1})$

or $2^{h-1}$ to $2^h$ -1.

# 2.3 BUILDING HEIGHT BALANCED TREE

Each node of an AVL tree has the Property that the height of the left subtree is either one more, equal, or one less than the height of the right subtree. We may define a balance factor (BF) as

BF = (Height of Right-subtree - Height of

Left-subtree)

Further

| | |
|---|---|
| If two subtree are of same height | BF = 0 |
| if Right subtree is higher | BF = +1 |
| if Left subtree is higher | BF = -1 |

For example balance factor are stated near the nodes in Figure 3. BF of the root node is zero because height of the right subtree and the left subtree is three. The BF at the node DIN is -17because the height of its left subtree is 2 and of right subtree is 1 etc.

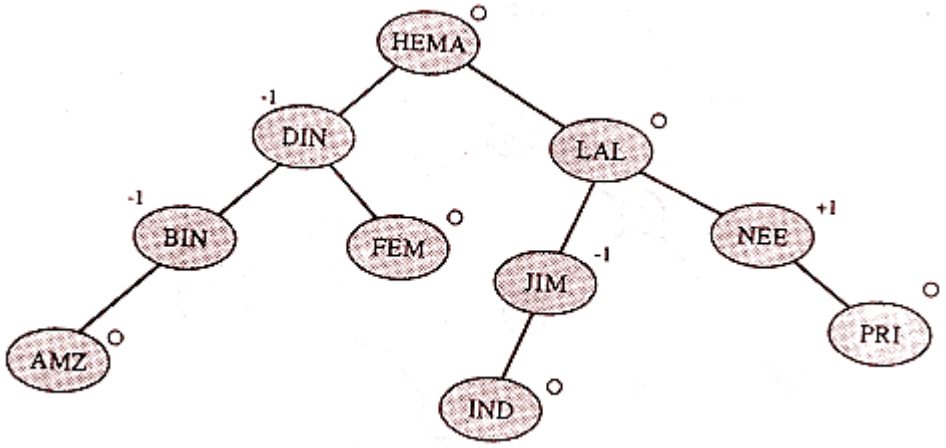



**Figure 3: Tree having a balance factor at each node**

Let the values given were in the order

BIN, FEM, IND, NEE, LAL, PRI, JIM, AMI, HEM, DIN and we needed to make the height balanced tree. It would work out as follows:

We begin at the root of the tree since the tree is initially empty we have


**Figure 4 (a)**

We have FEM to be added. It would be on the right of the already existing tree. Therefore we have

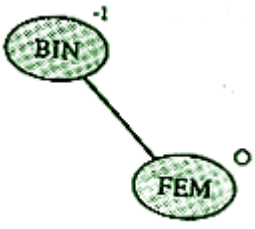



**Figure 4 (b)**

The resulting tree is still height balanced. Now we need to add IND ie. on the, further right of FEM.

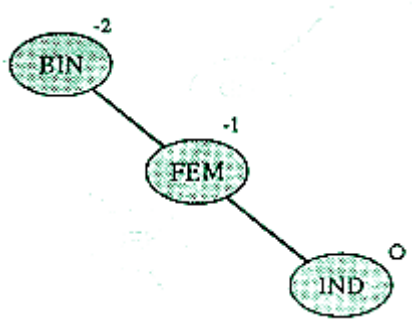



**Figure 4 (c)**

Since BF of one of the nodes is other than 0, + 1, or -1, we need to rebalance the tree. In such a case, when the new node goes to the longer side, we need to rotate the structure counter clockwise i.e. a rotation is carried out in counter clockwise direction around the closest parent of the inserted node with BF = + 2.

In this case we get

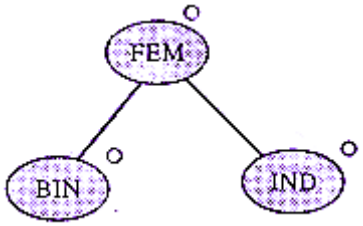



**Figure 4 (d)**

We now have a balanced tree.

On adding NEE, we get

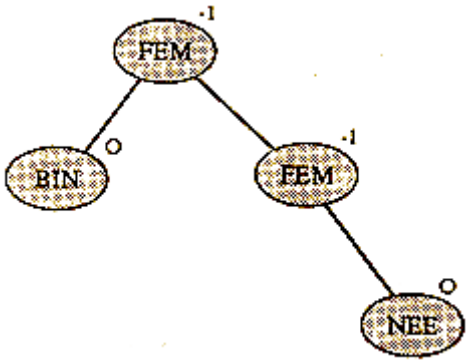



**Figure 4 (e)**

Since all the nodes have B. F. < + 2 we continue with the next node.

Now we need to add LAL

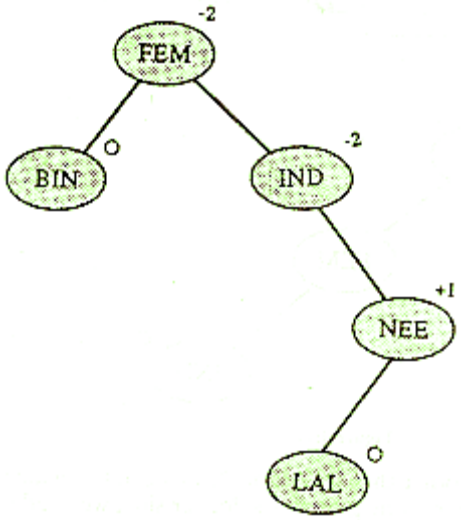



**Figure 4 (f)**

To regain balance we need to rotate the tree counter clockwise at IND and we get

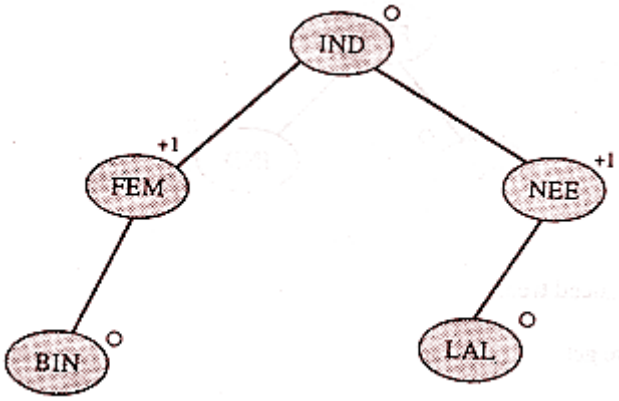



**Figure 4 (g)**

On adding PRI we get

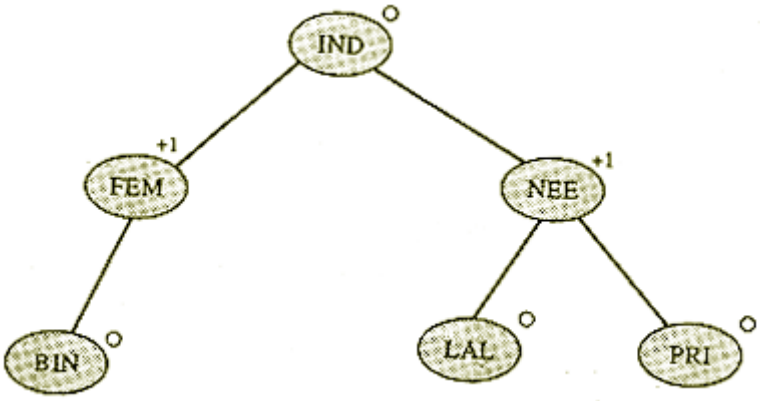



**Figure 4 (h)**

On adding JIM, we get

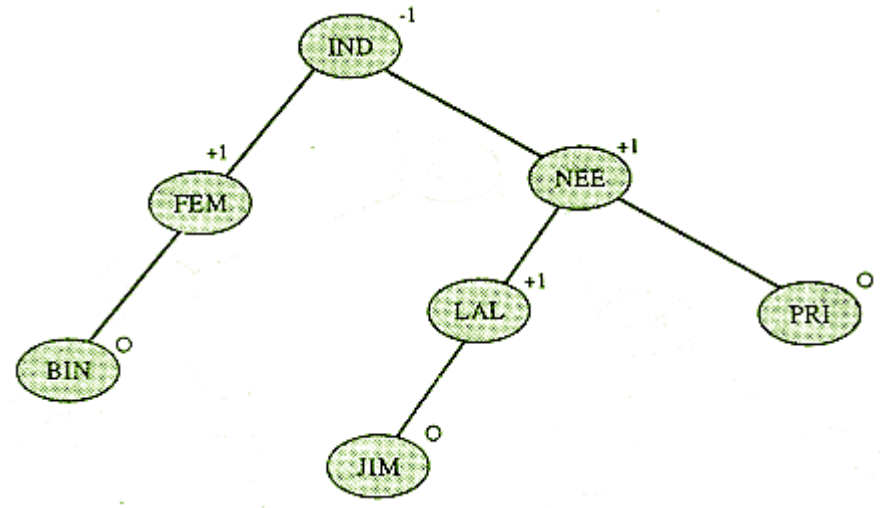



**Figure 4 (i)**

The tree is still balanced. Now we add AMI

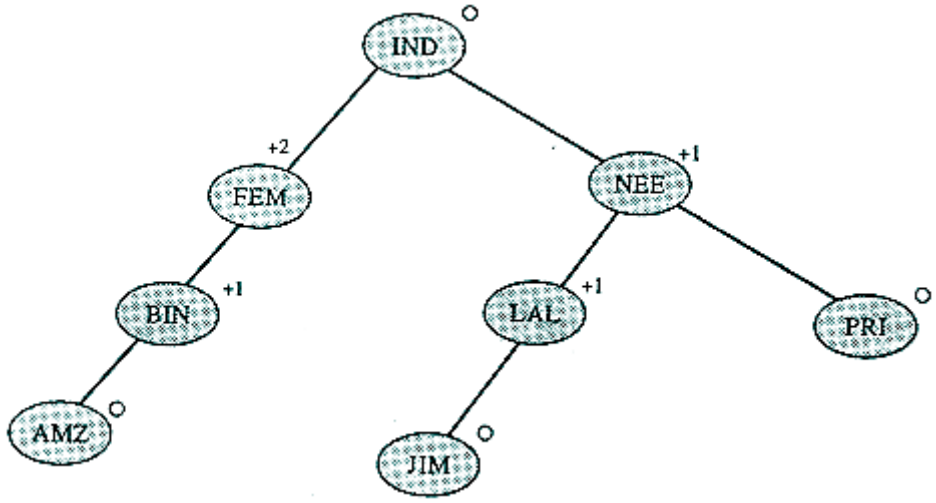



**Figure 4 (j)**

Now we need to rotate the tree at FEN (i.e. the closest parent to AMI with BF= +2) in clockwise direction. On rotating it once we get

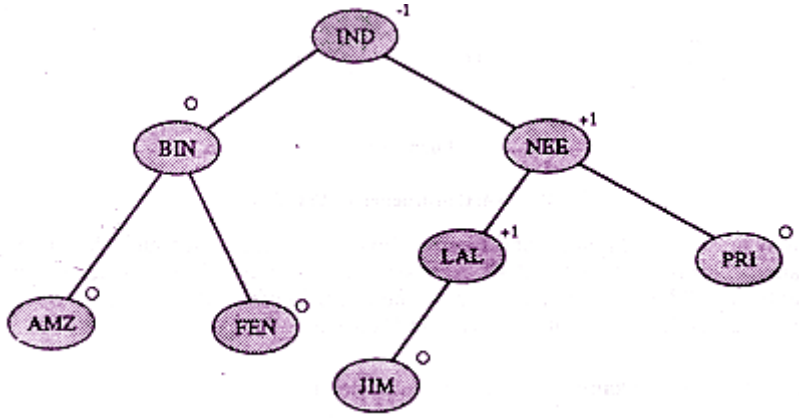



**Figure 4 (k)**

Now HEM is to be added. On doing so the structures we will get

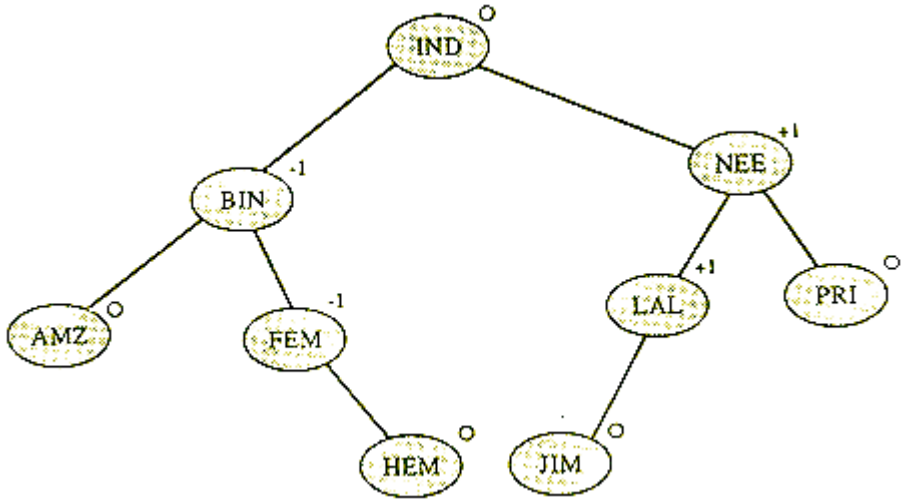



**Figure 4 (l)**

Tree is still balance. Now we need to add DIN, we get

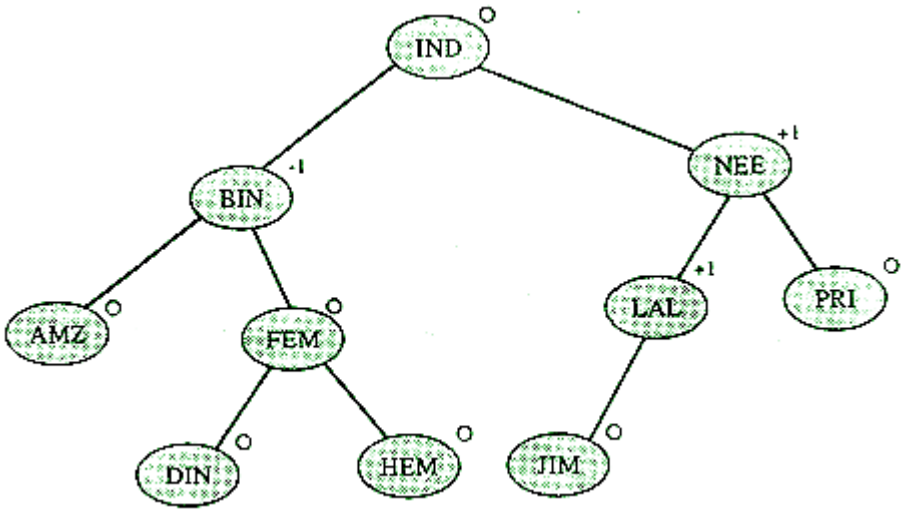



**Figure 4: Construction of AVL Tree**

You may notice that Figure 3 and Figure 4 (m) are different although the elements are same. This is because the AVL tree structure depends on the order in which elements are added. Can you determine the order in which the elements were added for a resultant structure as given in Figure 3?

Let us take another example. Consider the following list of elements

3,5,11,8,4,1,12,7,2,6,10

The tree structures generated till we add 11 are all balanced structure as given in Figure 5 (a) to (c)

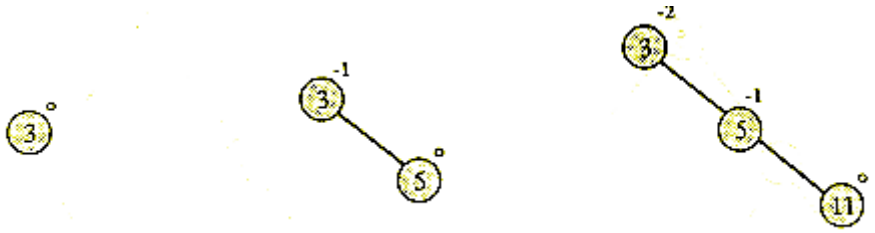



**Figure 5 (a) to (c)**

Here we need rebalancing. Rotation around 3 would give


**Figure 5 (d)**

Further we add 8,4,1,12,7,2 as depicted in Figure 5 (a) and (k).

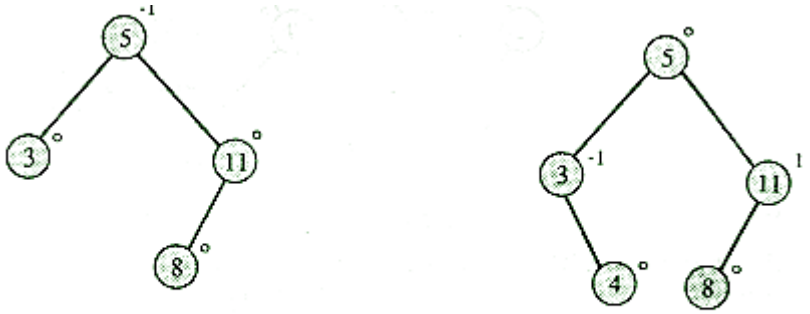



**Figure 5 (e) to (f)**

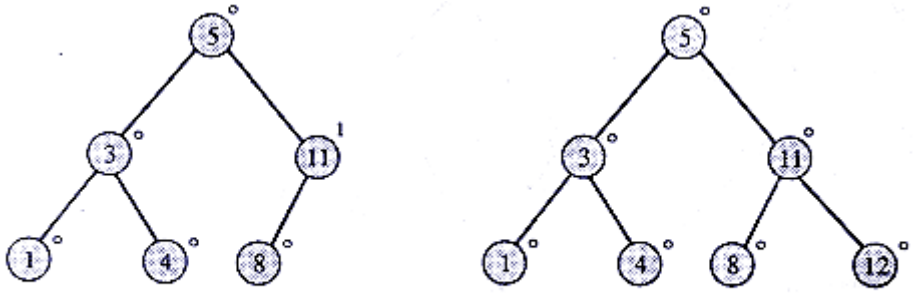



**Figure 5 (g) to (h)**

After adding 6, the tree becomes unbalanced. We need to rebalance by rotating the structure at the node 8. It is shown in Figure 5 (I)

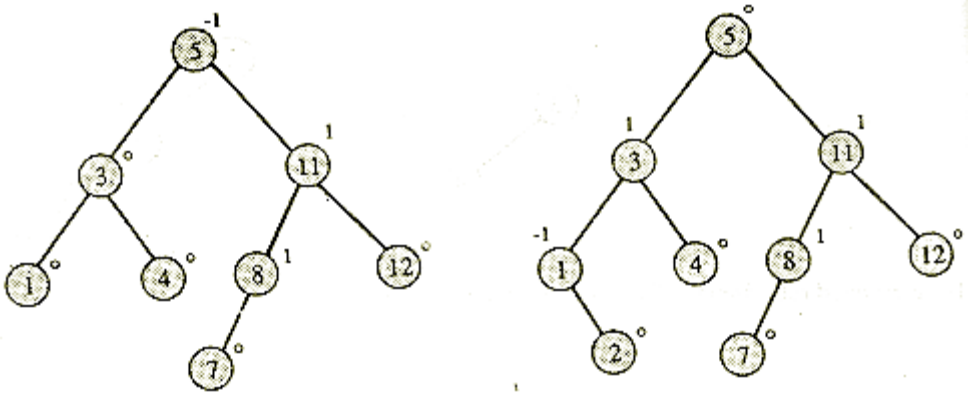


**Figure 5 (i) to (j)**

Again on adding 10 we need to balance

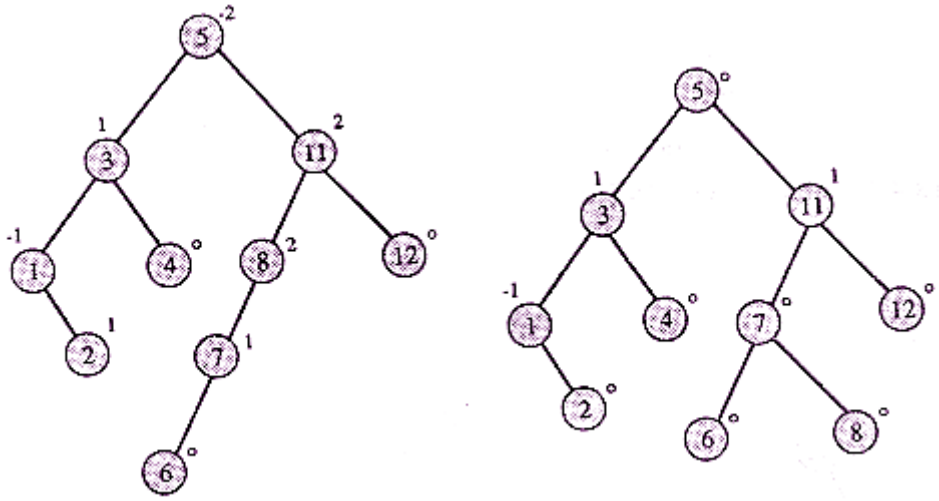


**Figure 5 (k) to (l)**

Since the closest parent with BF = 2 is at node 11 we first rotate at it in clockwise direction. However, we would still not get a balanced structure as the right of node 7 shall have move nodes than the left of it. Therefore, we shall need another rotation, in anti-clockwise direction at node 7, and finally we get a structure as shown in Figure 5 (n).

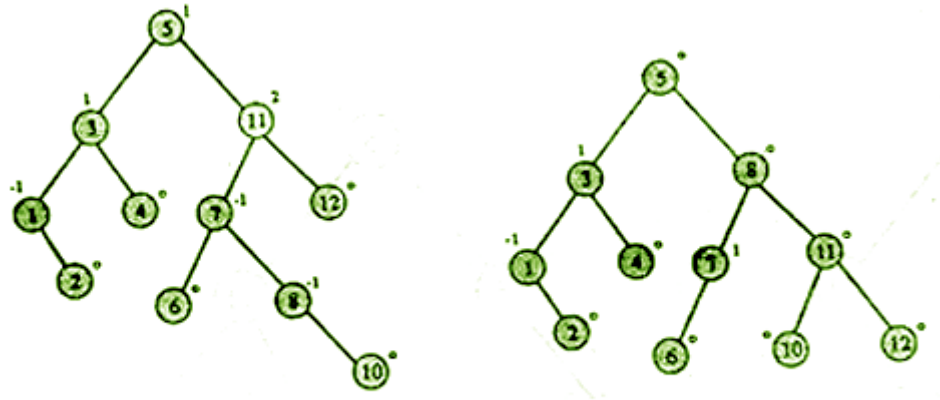


**Figure 5 (l) to (m)**

Which is the height balance tree.

# 2.4 B-TREE

We have already defined m-way tree in the Introduction. A B-tree is a balanced M-way tree. A node of the tree may contain many records or keys and pointers to children.

A B-tree is also known as the balanced sort tree. It finds its use in external sorting. It is not a binary tree.

To reduce disk accesses, several conditions of the tree must be true;

- the height of the tree must be kept to a minimum,
- there must be no empty subtrees above the leaves of the tree;
- the leaves of the tree must all be on the same level; and
- all nodes except the leaves must have at least some minimum number of children.

B-Tree of order M has following properties:

1. Each node has a maximum of M children and a minimum of M/2 children or any no. from 2 to the maximum.

2. Each node has one fewer keys than children with a maximum of M-1 keys.

3. Keys are arranged in a defined order within the node. All keys in the subtree to the left of a key are predecessors of the key and that on the right are successors of the key.

4. When a new key is to be inserted into a full node, the node is split into two nodes, and the key with the median value is inserted in the parent node. In case the parent node is the root, a new root is created.

5. All leaves are on the same level i.e. there is no empty subtree above the level of the leaves.

The order imposes a bound on the business of the tree.

While root and terminal nodes are special cases, normal nodes have between M/2 and M children. For example, a normal node of tree of order 11 has at least 6 more than 11 children.

## 2.4.1 B-Tree Insertion

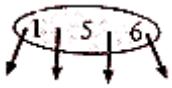
First the search for the place where the new record must be put is done. If the node can accommodate the new record insertion is simple. The record is added to the node with an appropriate pointer so that number of pointes remain one more that the number of records. If the node overflows because there is an upper bound on the size of a node, splitting is required. The node is spilt into three parts. The middle record is passed upward and inserted into the parent, leaving two children behind where there was one before. Splitting may propagate up the tree because the parent into which a record to be split in its child node, may overflow. Therefore it may also split. If the root is required to be split, a new root is created with just two children, and the tree grows taller by one level.

The method is well explained by the following examples:

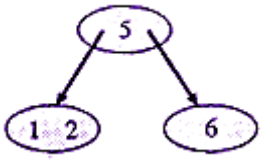
Example: Consider building a B-tree of degree 4 that is a balanced four-way tree where each node can hold three data values and have four branches. Suppose it needs to contain the following values.

1 5 6 2 8 11 13 18 20 7 9

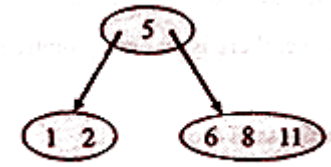
The first value 1 is placed in a new node which can accommodate the next two values also i.e.



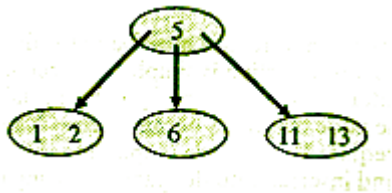

when the fourth value 2 is to be added, the node is split at a median value 5 into two leaf nodes with a parent at 5.



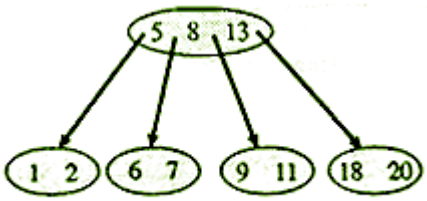

The following item 8 is to be added in a leaf node. A search for its appropriate place puts it in the node containing 6. Next, 11 is also put in the same. So we have




Now 13 is to be added. But the right leaf node, where 1-3 finds appropriate plane, is full. Therefore it is split at median value 8 and this it moves up to the parent. Also it splits up to make two nodes,




The remaining items may also be added following the above procedure. The final result is




Note that the tree built up in this manner is balanced, having all of its leaf nodes at one level. Also the tree appears to grow at its root, rather than at its leaves as was the case in binary tree.

A B-tree of order 3 is popularly known as 2-3 tree and that of order 4 as 2-3-4 tree.

# 2.4.2 B-Tree Deletion

As in the insertion method, the record to be deleted is first searched for.

If the record is in a terminal node, deletion is simple. The record along with an appropriate pointer is deleted.

If the record is not in terminal node, it is replaced by a copy of its successor, that is, a record with a next, higher value. The successor of any record not at the lowest level will always be in a terminal node. Thus in all cases deletion involves removing a record from a terminal node.

If on deleting the record, the new node size is not below the minimum, the deletion is over. If the new node size is lower than the minimum, an underflow occurs.

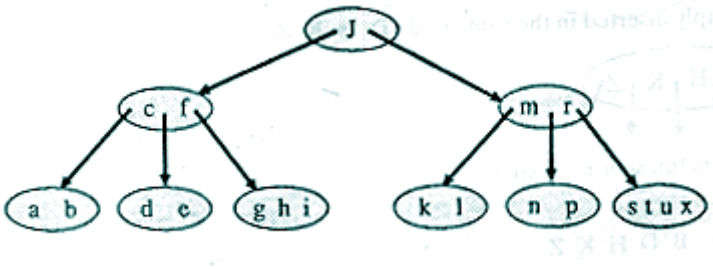
Redistribution is carried out if either of adjacent siblings contains more than the minimum number of records. For redistribution, the contents of the node which has less than minimum records, the contents of its adjacent sibling which has more the minimum records, and the separating record from parent are collected. The central record from this collection is written back to parent. The left and right halves are written back to the two siblings.

In case the node with less than minimum number of records has no adjacent sibling that is more than minimally full. Concatentation is used. In this case the node is merged with its adjacent sibling and the separating record from its parent.

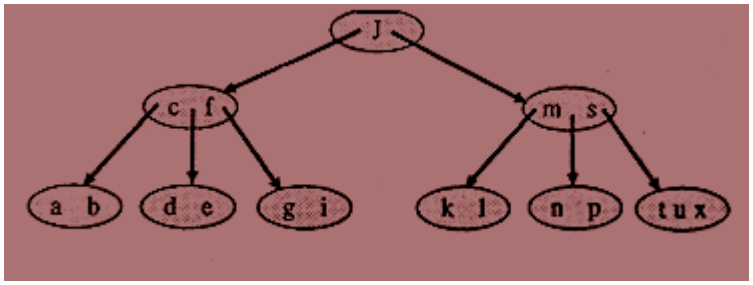
It may be solved by redistribution or concatenation.

We will illustrate by deleting keys from the tree given below:

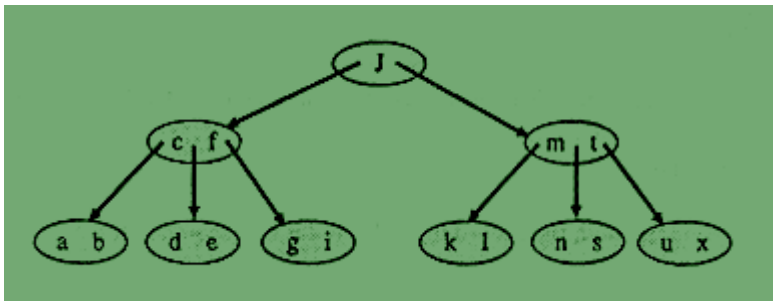
1. Delete 'h'. This is a simple deletion



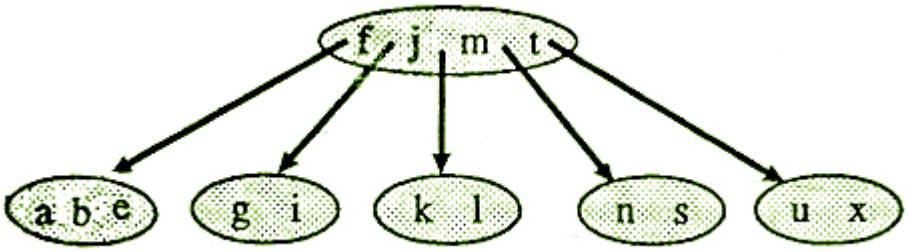

2. Delete, 'r'. 'r' is not at a leaf node . Therefore its successor Is, is moved up 'r' moved down and deleted.




3. Delete 'P'. The node contains less than minimum numbers of keys required The slibing can spare a key. So ,t, moves up and 's' moves down.




4. Deletion 'd'. Again node is less than minimal required. This leaves the parent with only one key. Its slibing cannot contribute. therefore f,j, m and t are combined to form the new root. Therefore the size of tree shrinks by one level.




# 2.5 B-Tree of order 5

Let us build a B-tree of order 5 for following data:

1. 

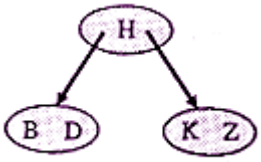

2. H K Z are simply inserted in the same node D H K Z


3. Add B: node is full so it must split

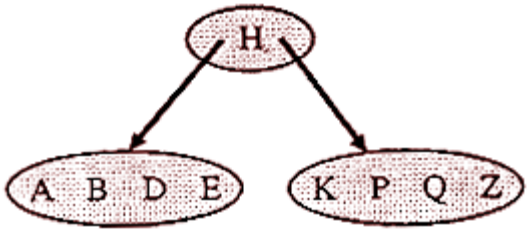
   H is median for

   B D H K Z

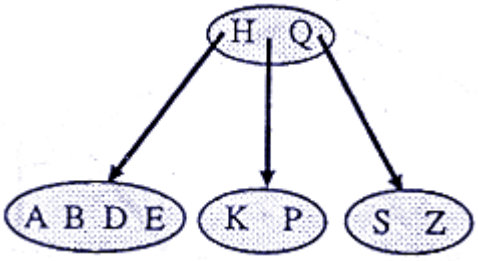
- H is made as the parent. Since the splitting is at root node we must make one more node.



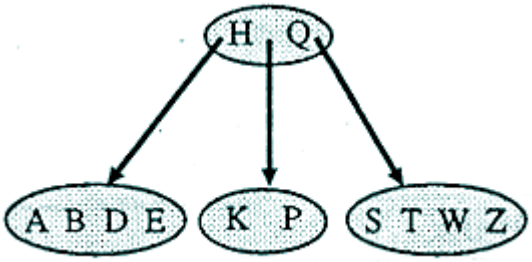

4. P, Q,E,A are simply inserted in



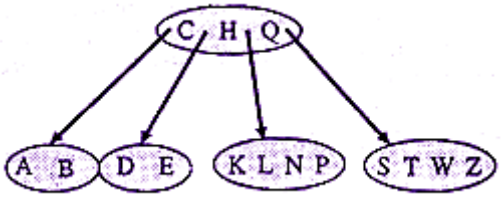

S to be added at K P Q Z
Q becomes the median



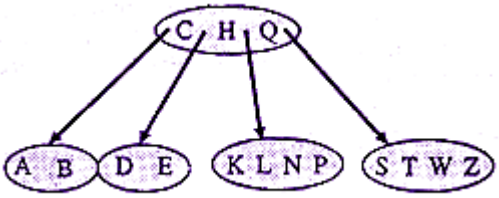

W and T are simply inserted after Q



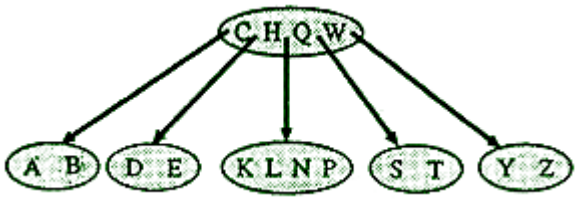

After Adding C we get



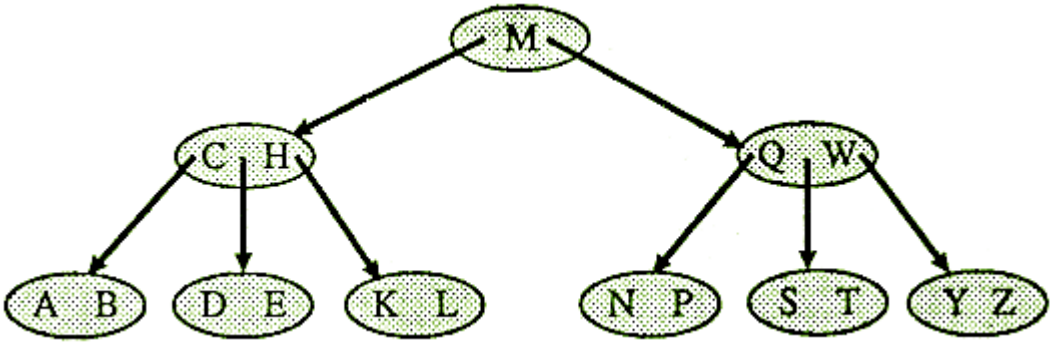

L,N are simply inserted




Y to be added in S T W Z Since W is median




M to be put in K L N P & is median. Then, it will be promoted to CHQW but that is also full therefore this root will be split and new root will be created




# 2.6 SUMMARY

In this Unit, we discussed the AVL trees and B-Trees. AVL trees are restricted growth Binary trees. Normal binary trees suffer with the problem of balancing. AVL trees offer one kind of solution. AVL trees are not completely balanced trees. In completely balanced trees the number of nodes in the two subtrees of a node differ by at most 1. In AVL trees the heights of two subtrees of a node may differ by at most 1.

Multiway - Trees are generalization of binary trees. A node in an m-way tree can contain R records and R + 1 pointers (or children). Consequently the branching factor increases or tree becomes shorter as compared to the binary tree for the same number of records. B-trees are balanced multiway trees. It restricts the number of records in a node between m/2 and m-1, i.e. it requires a node for a tree of order m to be at least half full.

Further the structure of B-Tree self balancing operations, as insertions and deletions are performed.

Two variations of B-Tree also exist as B* Tree and B + tree. The student is advised to read and some text on these structure as well.