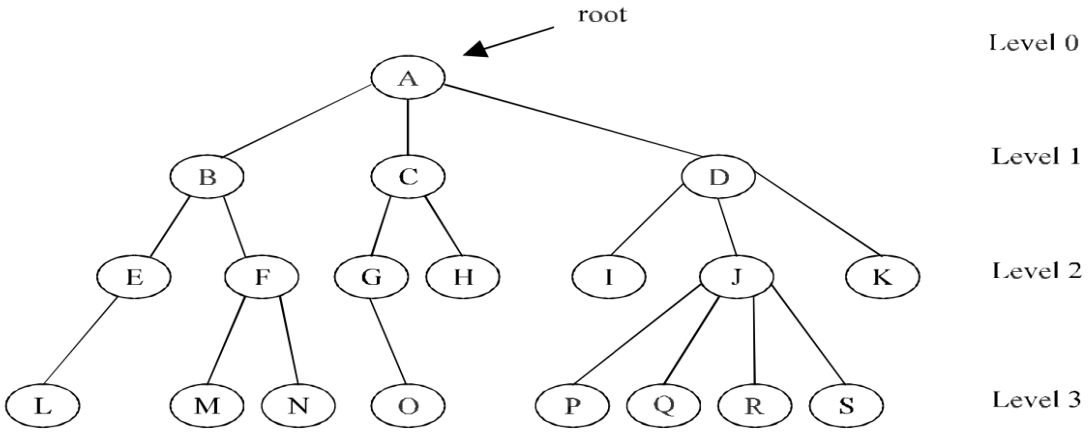
Unit 6: Trees

- Linked list usually provides greater flexibility than array, but they are linear structures and it is difficult to use them to organize a hierarchical representation of objects.
- To overcome these limitations, we create a new data type called a tree that consists of nodes and arcs.
- Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand-children as so on.



- A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

i. There is a special node called the root of the tree.
ii. Removing nodes (or data item) are partitioned into number of mutually exclusive (*i.e.,* disjoined) subsets each of which is itself a tree, are called sub tree.

Basic Terminologies:

1. **Root:** A tree contains a unique first node which has no parents. It is shown at the top of the tree structure.

2. **Node:** Each data item in a tree is called a *node*. It specifies the data information and links (branches) to other data items.

3. **Degree of a Node:** It is the number of subtrees of a node in a given tree. Example: Degree of Node A in above tree is 3. Degree of Node B and C is 2.

4. **Degree of a Tree:** It is the maximum degree of node in a given tree. The degree of above tree is 4(which is the degree of J).

5. **Terminal/ Leaf Node**: A node with degree Zero is called terminal or leaf node. It is the node which does not have any child.

6. **Non-Terminal Node:** Any node whose degree is not zero is called non-terminal node. They are intermediate nodes in traversing a given tree from its root node to the terminal node.

7. **Levels:** The tree is structured in different levels. The entire tree is leveled in such a way that the root is always at level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level n+1.

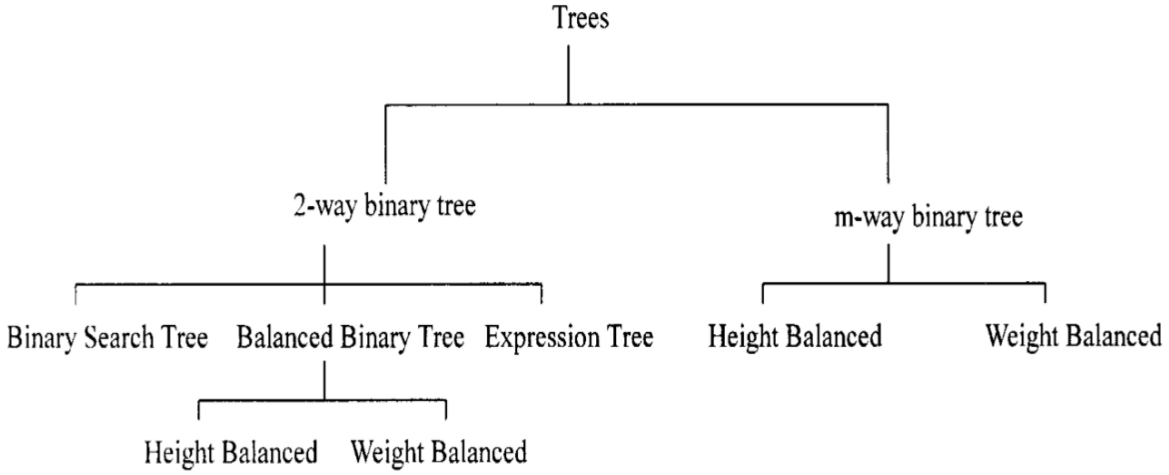
8. **Depth of a Tree:** It is the maximum level of any node in a given tree. The depth of above tree is 3. The term height is also used to denote the depth.

9. **Siblings:** If two or more nodes have same parent, then these nodes are called siblings to each other. Children of same parent are siblings.

10. **Ancestor:** A node is called ancestor of another node if either it is the parent of that node or it is the parent of some other ancestor of that node.

11. **Descendants:** A node is called Descendants of another node if it is the child of that node or the child of some other descendants of that node.

Trees can be divided in different classes as below:



Binary Tree

- A tree is called binary tree if each node has either zero child or one child or two children.
- A binary tree is a tree in which no node can have more that two children. Typically these children are described as "left child" and "right child" of the parent node.
- A binary tree T is defined as a finite set of elements (called nodes) such that:
  1. T is empty (i.e., if T has no nodes called the null tree or empty tree).
  2. T contains a special node R, called root node of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is not empty then its root is called the left successor of R, similarly if T2 is non empty then its root is called the right successor of R.

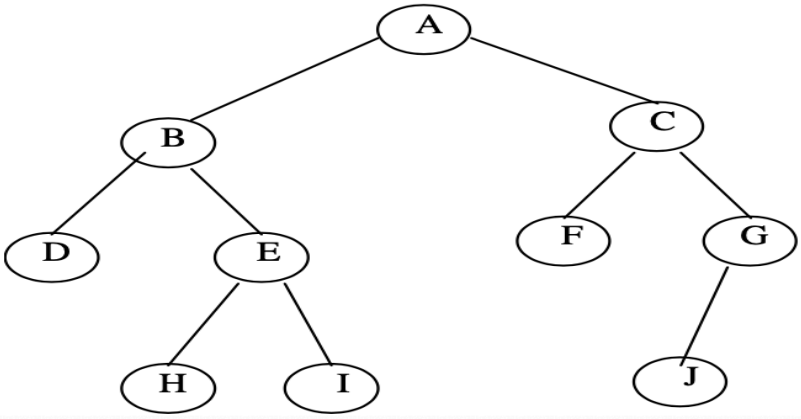


Fig. 1 Binary Tree

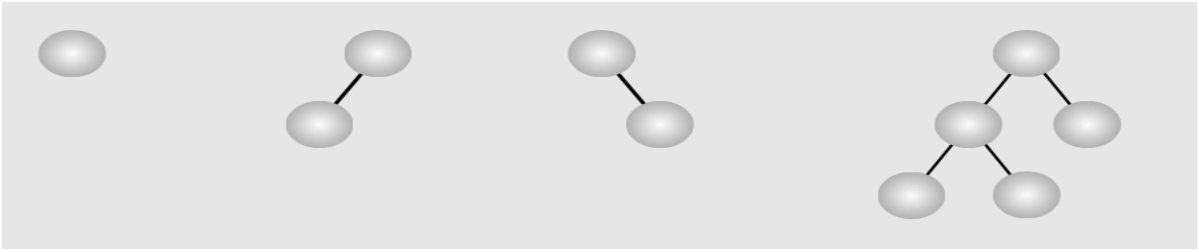
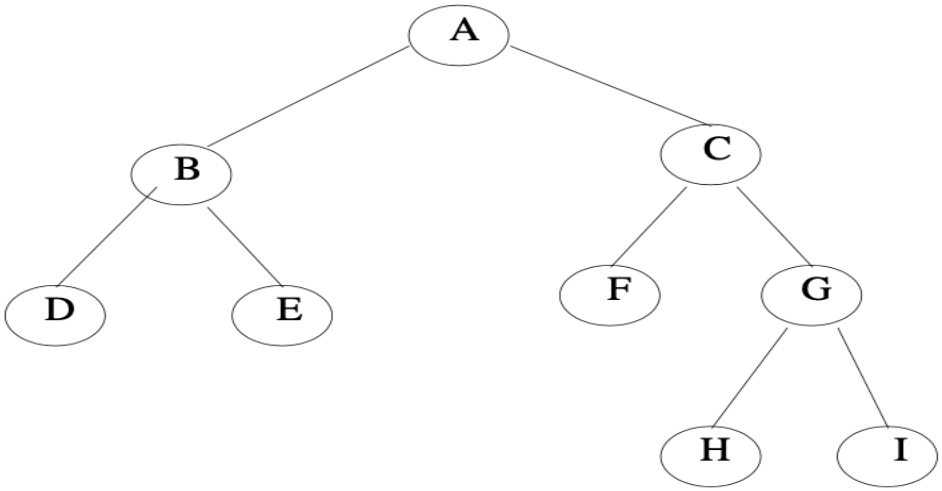


Fig. 2 Binary Trees

Strictly Binary Tree

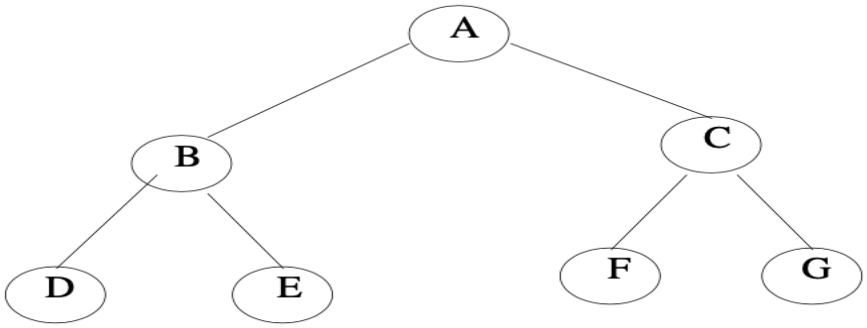
- It is binary tree with every node has exactly two children or no children.
- The tree is said to be *strictly binary tree*, if every non-leaf made in a binary tree has non-empty left and right sub trees.
- They are also called 2-*tree* or *extended binary tree.*
- A strictly binary tree with n leaves always contains $2n-1$ node.

- The below tree is strictly binary tree:



Complete Binary Tree

- It is a special type of strictly binary tree where all the leaves of the tree reside at the same level.
- A complete binary tree at depth '*d*' is the strictly binary tree, where all the leaves are at level *d*. The below tree is a complete binary tree of depth 2.



- A binary tree with n nodes, $n > 0$, has exactly $n - 1$ edge.
- A binary tree of depth d, $d>0$, has at least d and at most 2^d -1 nodes in it.
- If a binary tree contains n nodes at level l, then it contains at most $2n$ nodes at level $l + 1$.
- A complete binary tree of depth d is the binary tree of depth d contains exactly $2l$ nodes at each level l between 0 and d.

Difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.
2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
3. The sub tree of each element in a binary tree are ordered, left and right sub trees. The sub trees in a tree are unordered.

Left and Right Skewed binary tree:

- If a binary tree has only left sub trees, then it is called left skewed binary tree.
- If a binary tree has only right sub trees, then it is called right skewed binary tree.

(a) Left skewed binary tree *(b) Right skewed binary tree*

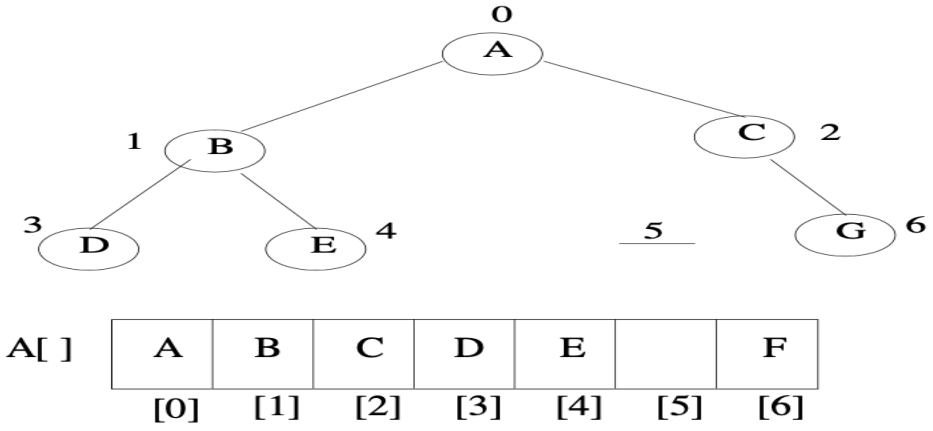
Implementing Binary Trees:

Binary trees can be implemented in at least two ways: as arrays and as linked list.

Array Implementation of Binary Tree:

- The sequential representation of binary tree uses array for storing the data for each node.
- This is very simple. If any parent node is stored at index i then its left child will be stored at 2*i+1 and right child will be stored at 2*i+2.
- The root of the tree is stored at first index of the array (index 0).

The array representation of above BST is as shown below:



- It is clear that the most of the locations in array are empty.

- This causes wastage of memory space. That is, the array representation of binary tree is quite inefficient.
- In general, for a binary tree with height H, the size of the array will be approximately be 2^H.

Linked List Representation of Binary Tree:

- The linked list representation is the most popular, efficient and most frequently used representation of binary tree.
- In the linked list representation, every node is represented by data, a reference to left child and a reference to right child.
- The node of binary tree can be created as follows:

```
class Node
{
        int data;
        Node left;
        Node right;
        Node ()
        {
                left = null;
                right = null;
        }
        Node (int n)
        {
                data = n;
                left = null;
                right = null;
        }
}
```

Basic Operations on Binary Tree:

1. Create an empty Binary tree
2. Traversing a Binary tree
3. Inserting a New Node
4. Delete a Node
5. Searching for a Node
6. Return the information stored in a Node
7. Copying the mirror image of a Node
8. Determine the total number of Node
9. Find the height of the tree
10. Finding the Father(parent)/ Left child/ Right child/ Sibling of a Node

Binary Search Tree

The application of binary tree is searching and sorting. By enforcing certain rules on the values of the elements stored in a binary tree, it could be used to search and sort.

Binary search tree is a tree in which value of each node in the tree is greater than the value of node in its left (if exists) and it is less than the value in its right child (if it exists).

As name suggests, binary search tree (BST) is used for searching purpose. For every node N in the BST, the following property will be true.

- Every node has a unique value.
- The data at left node will be smaller than data at node N
- The data at right node will be larger than data at node N

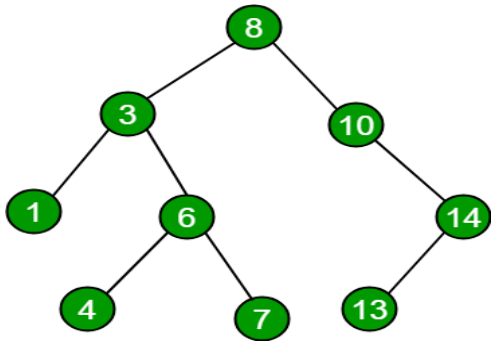


Figure: Binary Search Tree

Searching a Binary Search Tree

An algorithm for locating an element in binary search tree is quite straightforward. For every node, compare the key to be located with the value stored in the node root.

- If the key is equal to value, then stop.
- If the key is less than the value, go to the left subtree and try again.
- If key is greater than that value, try the right subtree.

Method for Searching a Binary Search Tree:

```
public BST search (BST root, int el)
{
        while (root != null)
        {
                if (el == root.key)
                        return root;
                else if (el < root.key)
                        root = root.left;
                else
                        root = root.right;
        }
        return null;
}
```

Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once.

The definition of traversal specifies only one condition-visiting each node only one time but it does not specify the order in which the nodes are visited. Two types of tree traversal are:

1. Breadth First Traversal (BFT)
2. Depth First Traversal (DFT)

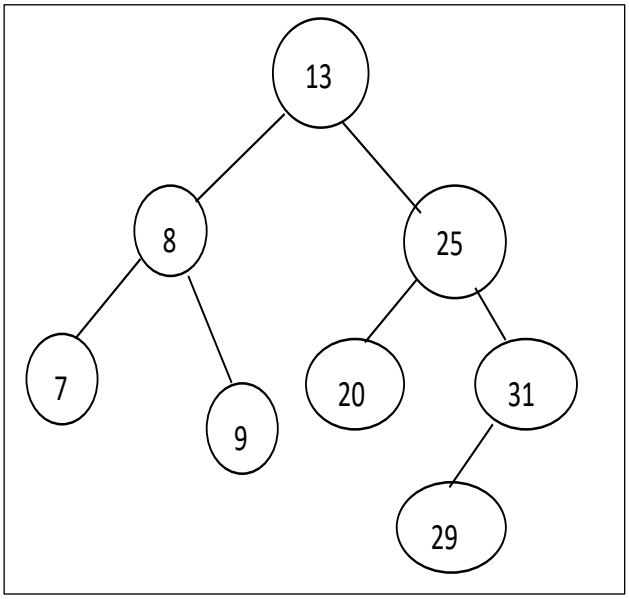
Breadth First Traversal

Breadth first search traversal is visiting each node from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left). There are thus four possibilities, and one such possibility, a top - down, left to right, breadth first traversal of the tree.

Implementation of this kind of traversal is straightforward when a queue is used.

Consider a top-down, left to right, breadth first traversal. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited. The restriction is that all nodes on level n must be visited before visiting any node on level n+1 is accomplished. For example, consider the following tree:

```
public void breadthFirst()
{
        Node p = root;
        Queue queue = new Queue;
        if (p != null)
        {
                queue.enqueue(p);
                while (!queue.isempty())
                {
                        p = (Node) queue.dequeue();
                        visit(p);
                        if (p.left != null)
                                queue.enqueue(p.left);
                        if (p.right != null)
                                 queue.enqueue(p.right);
                }
        }
}
```



BFT Sequence: 13, 8, 25, 7, 9, 20, 31, 29

Depth First Traversal

Depth first traversal proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right). We repeat this process until all nodes are visited. There are some variations of the depth first search traversal.

There are three tasks of interest in these types of traversals:

V - Visiting a node

L - Traversing the left subtree.

R - Traversing the right subtree.

The three unique traversals are as:

1. Preorder three traversal: VLR
2. Inorder tree traversal: LVR
3. Postorder tree traversal: LRV

Preorder Traversal

To traverse a nonempty binary tree in **preorder**, we perform the following three operations:

- Visit the root
- Traverse the left tree in preorder
- Traverse the right sub tree in preorder

Inorder Traversal

To traverse a nonempty binary tree in **inorder**, we perform the following three operations:

- Traverse the left sub tree in inorder
- Visit the root
- Traverse the right sub tree inorder

Postorder Traversal

To traverse a non-empty tree in **Postorder**, we perform the following

- Traverse the left sub tree in postoder
- Traverse the right sub tree in postorder
- Visit the root

Methods for Depth First Traversal:

1. Preorder (VLR):

```
void preorder(Node p)
{
        if (p != null)
        {
                visit(p);
                preorder(p.left);
                preorder(p.right);
        }
}
```

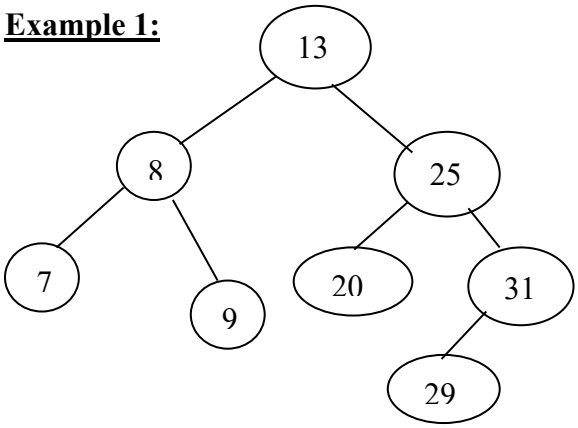
2. Inorder (LVR):

```
void inorder(Node p)
{
        if (p != null)
        {
                inorder(p.left);
                visit(p);
                inorder(p.right);
        }
}
```

3. Postorder (LRV):

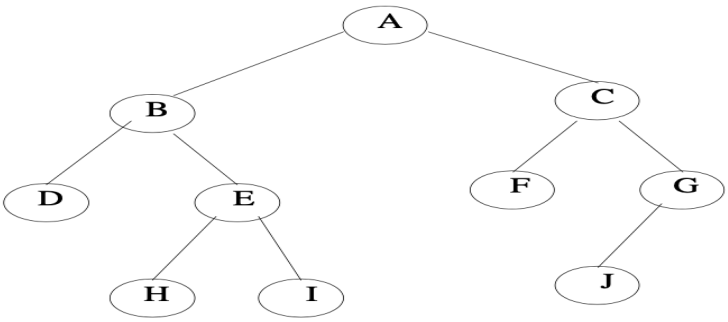
```
void postorder(Node p)
{
        if (p != null)
        {
                postorder(p.left);
                postorder(p.right);
                visit(p);
        }
}
```

Example 1:



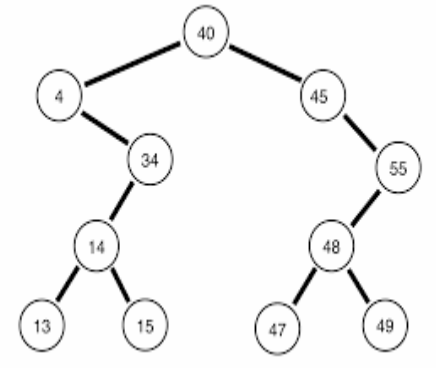
The preorder traversal will print: 13 8 7 9 25 20 31 29
The inorder traversal will print: 7 8 9 13 20 25 29 31
The postorder traversal will print: 7 9 8 20 29 31 25 13

Example 2:



- The Pre Order Traversal of above binary tree is: **A, B, D, E, H, I, C, F, G, J**.
- The In Order Traversal of above binary tree is: **D, B, H, E, I, A, F, C, J, G**.
- The Post Order Traversal of above binary tree is: **D, H, I, E, B, F, J, G, C, A.**

Traverse the below tree in pre order, in order and post order traversal:
Example 2:



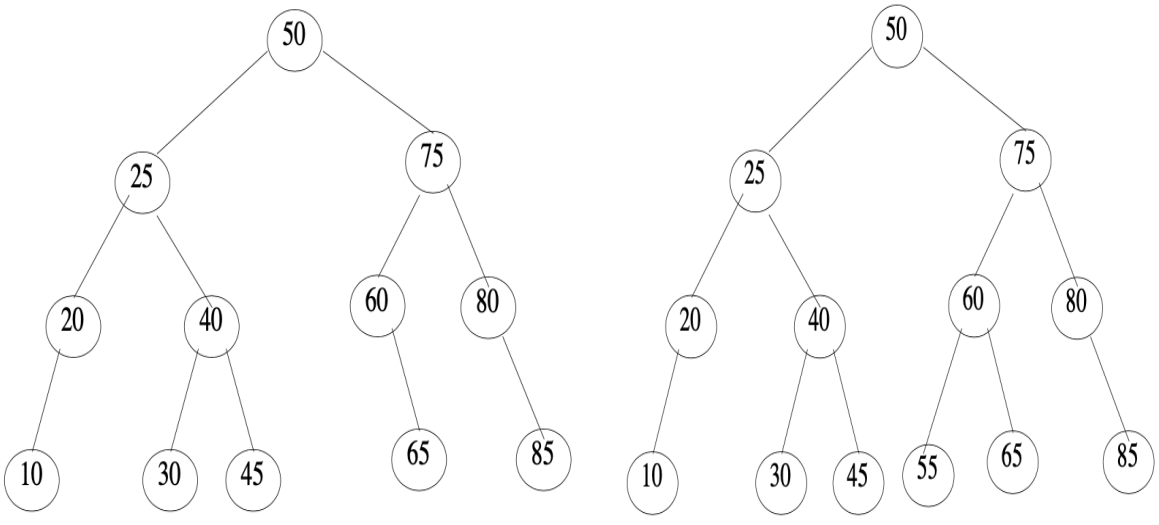
Insertion

To insert a new node with key el into a BST, a tree node with a dead end has to be reached, and new node has to be attached to it. Such a tree node is found using the same technique as used in tree searching. Such a tree node is found using the same technique as used in tree searching.
The key el is compared to the key of the node currently being examined during tree scan. If el is less than that key, the left child (if any) of p is tried; otherwise, the right child is tested. If the child of p to be tested is empty, the scanning is discontinued and the new node becomes this child.

Function to insert a new node into a BST:

```
void insert (Node root, int el)
{
        if (root == null)
        {
                root.left = root.right = null;
                root.info = el;
        }
        else
        {
                if (el < root.info)
                        root.left =  insert(root.left, el);
                else
                        root.right = insert(root.right, el);
        }
}
```

Example: Insert DATA = 55 into below BST



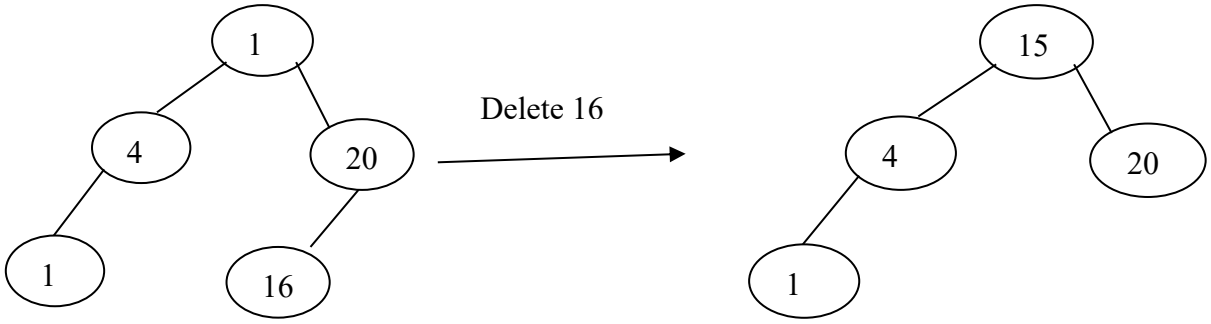
(a) Before Inserting 55 *(b) After Inserting 55*

Deletion

Deleting a node is another operation to maintain a binary Search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree.

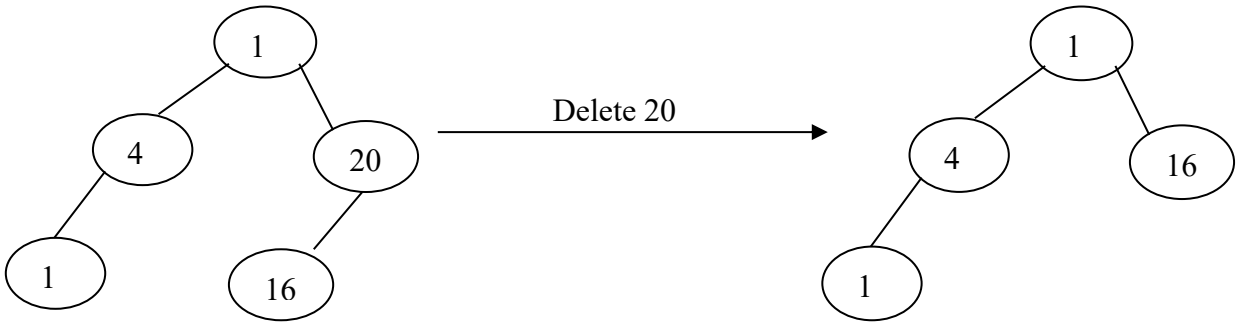
1. **Deleting a leaf Node:**

   The node is a leaf; it has no children: This is the easiest case to deal with. Simply delete the node and place its parent node by the NULL pointer.



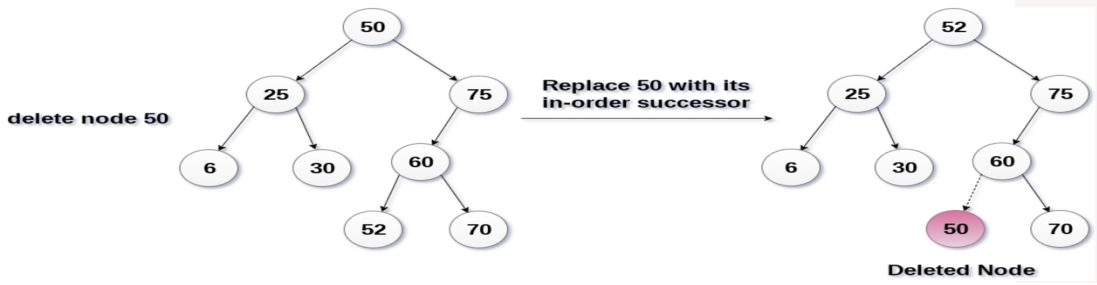
2. **Deleting a node with exactly one child:**
   The node has one child: This case is not complicated. The parent's child reference to the node is reset to refer to the node's child. In this way, the node's children are lifted up be one level.

   If N has one child, check whether it is a right or left child. If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node. If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

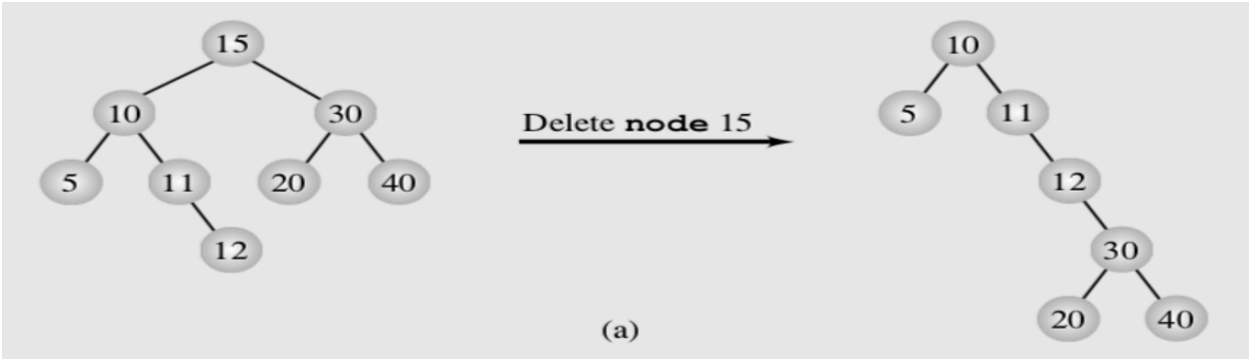


3. **The node has two children**:
   In this case, no one step option can be performed because the parent's right or left reference cannot refer to both the node's children at the same time.
   There are following two cases to delete a node with two children:

   (i) Deletion by Copying:
   Suppose p is the node to be deleted. Then either inorder successor of p must take its place where the inorder successor cannot have left subtree or inorder predecessor of p must take its place where the inorder predecessor cannot have right subtree.



   (ii) Deletion by Merging:
   Suppose p is the node to be deleted. Then, merge left subtree of p into right subtree of p (in this case merging start from the smallest node of right subtree). We can also merge right subtree into left subtree (in this case merging start from the largest node of left subtree).



Algorithm to delete a node from BST:

1. Start

2. If a node to be deleted is a leaf node at left/right side

a) simply delete and set null pointer to its parent's left/right pointer.

3. If a node to be deleted has only one child

a) connect its child pointer with its parent pointer and delete it from the tree.

4. If a node to be deleted has two children

a) replace the node being deleted either by a right most node of its left subtree (replace with inorder predecessor node) or leftmost node of its right subtree (replace with inorder successor node)

5. End

Balancing a Tree

There are following two arguments that support the use of tree:

1. The first one is tree is used to represent the hierarchical structure of a certain domain and
2. search process is much faster is using trees than using linked lists.

The second argument, however, does not always hold. It all depends on what the tree looks like. If tree is not balanced tree, it is more or less similar to linked list.

A binary tree is height balanced tree or simply balanced tree if the difference in height of the left and the right subtrees of any node in the tree is either zero or one.

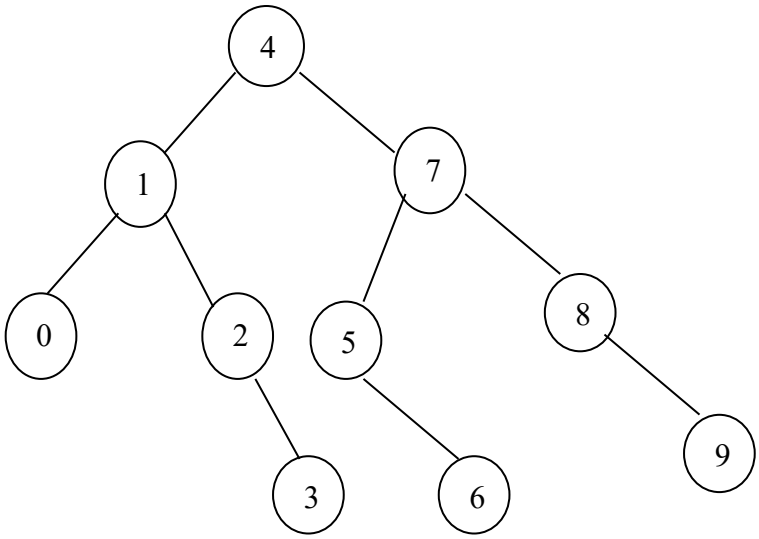
A tree is considered *perfectly balanced* if it is balanced and all leaves are to be found on one level or two levels.

Height of Tree(**h**)	Minimum number of Nodes (2^{h-1})	Maximum number of Nodes (2^h**-1**)
1	$2^0 = 1$	$2^1 - 1 = 1$
2	$2^1 = 2$	$2^2 - 1 = 3$
3	$2^2 = 4$	$2^3 - 1 = 7$
4	$2^3 = 8$	$2^4 - 1 = 15$

Balancing a binary tree (binary search technique)

```
void balance (int data [], int first, int last)
{
        if(first<=last)
        {
                int middle = (first+last)/2;
                insert(data[middle]);
                balance (data, first, middle-1);
                balance (data, middle+1, last);
        }
}
```

Let us consider the following example:
Stream of data [] = {5, 1, 9, 8, 7, 0, 2, 3, 4, 6}
Array of sorted data [] = {0, **1,** 2, 3, **4,** 5, 6, 7, 8, 9}



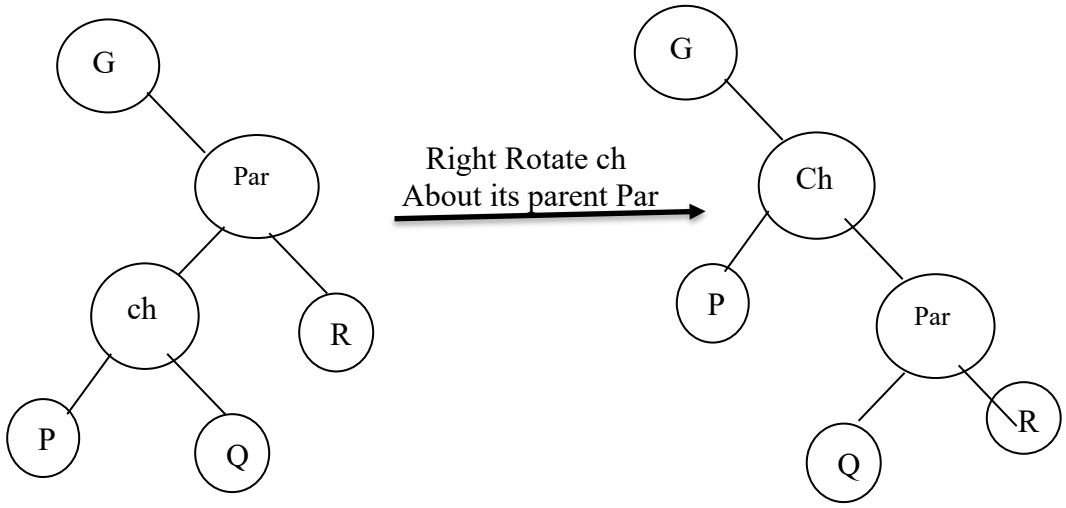
The DSW Algorithm

The earlier method is no efficient in that it requires an array and needs array to be sorted. To avoid sorting, it required deconstructing the tree after placing elements in the array using the inorder traversal, and then reconstructing the tree, which is inefficient except for relatively small tree. There are, however, algorithms that require little additional storage for intermediate variables and use no sorting procedures. The very elegant DSW algorithm was devised by Colin day and later improved by block Quentin F, Stout.

- The building block for tree transformation in this algorithm is the rotation. There are two of rotations, left and right, which are symmetrical to each other.

The right rotation of the node ch about its parent par is performed according to the following algorithm:

rotateRight (Gr, Par, Ch)
if Par is not the root of the tree
grandparent Gr of child Ch becomes Ch's parent;
right subtree of Ch becomes left subtree of Ch's parent Par;
node Ch acquires Par as its right child;



- At first, the DSW algorithm transforms an arbitrary binary search tree into a linked list like tree called a backbone or vine.

Algorithm for creating backbone

CreateBackbone (root, n)

tmp = root;

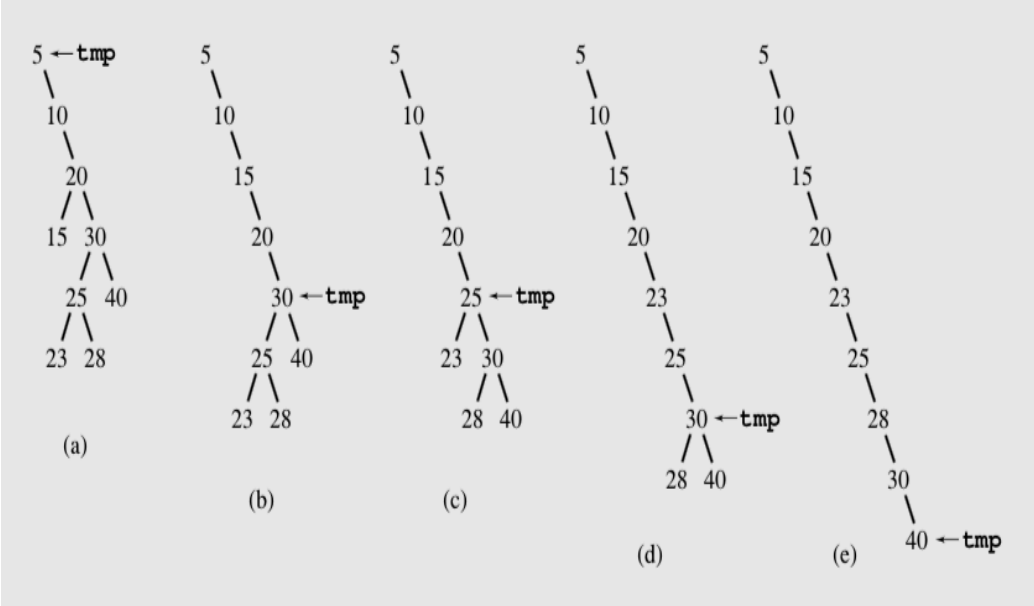
while (tmp != null)

if tmp has a left child

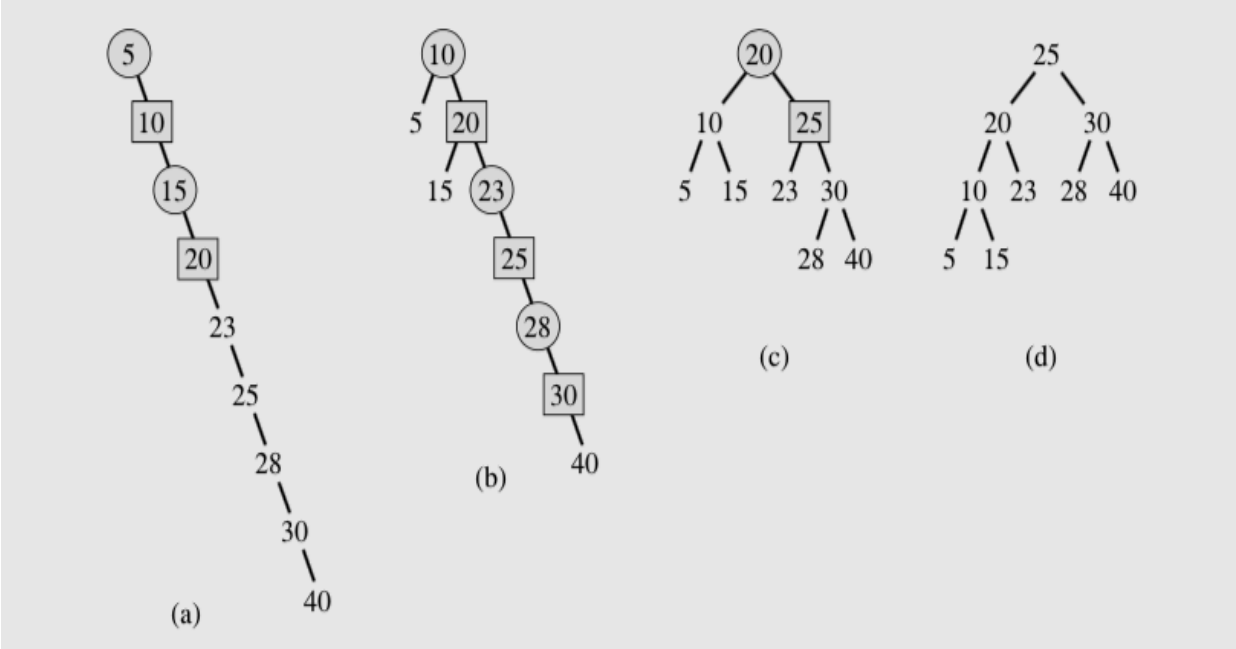
rotate this about tmp;

set tmp to the child that just became parent;

else set tmp to its right child;



- Then this elongated tree (backbone) is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent as below:



AVL Trees (Admissible tree):

- The AVL tree was proposed by **Adelson, Velsky and Landis**.
- An AVL tree (originally called an admissible tree) is one in which the height of the left and right subtrees of every node differs by at most one. The definition of AVL tree is same as definition of the balanced tree.
- However, the concept of the AVL tree always implicitly includes the techniques for balancing the tree.

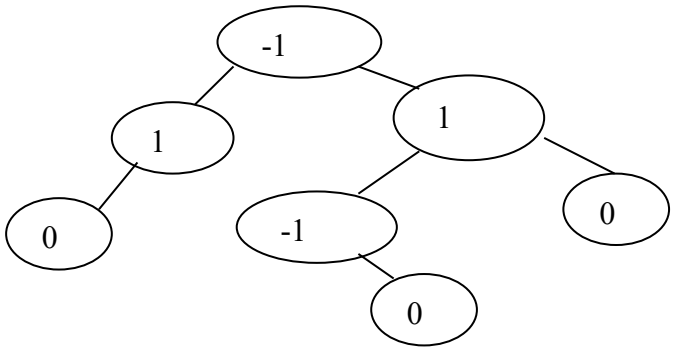
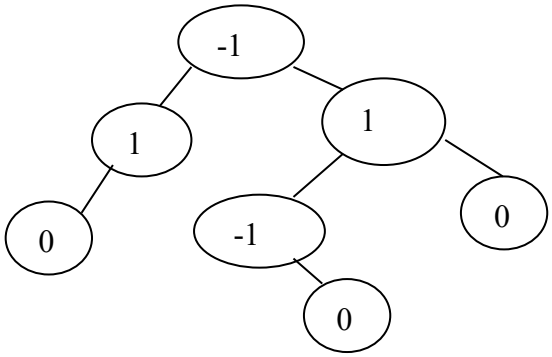


Figure: AVL Tree

- Numbers in the nodes indicate the balance factors that are difference between the height of the left subtrees minus the height of the right subtrees.
- For an AVL tree, all balance factors should be +1, 0 or -1.
- If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced.

The minimum number of nodes in an AVL tree is determined by the recurrence equation:
$\mathbf{AVL_h = AVL_{h-1}+AVL_{h-2}+1}$
Where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.
$AVL_2 = AVL_1 + AVL_0 + 1 = 1+0+1 = 2$
$AVL_3 = AVL_2 + AVL_1 + 1 = 2+1+1 = 4$
$AVL_4 = AVL_3 + AVL_2 + 1 = 4+2+1 = 7$

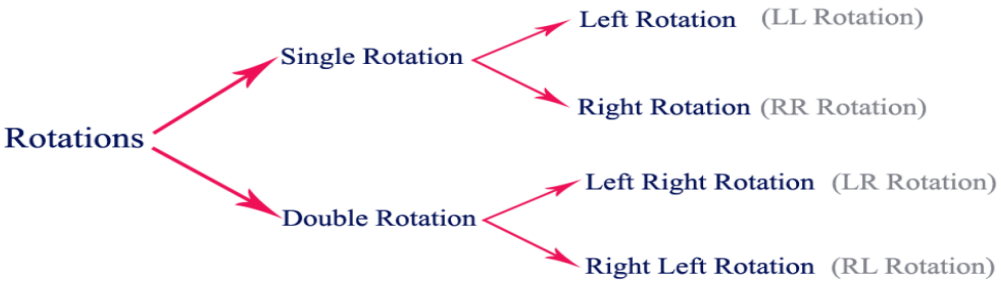
An AVL tree of height 4 is as below:



- The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated.
- If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done.

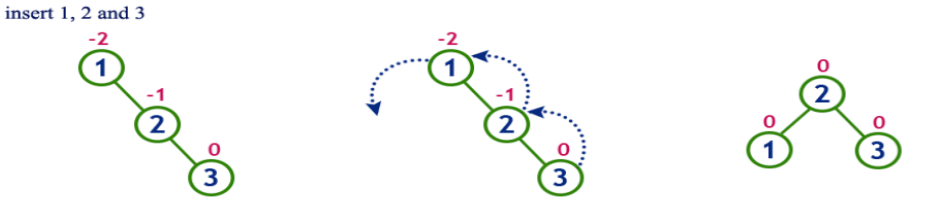
To make Non-AVL tree to AVL tree:

- Perform following rotations to make a non AVL tree to an AVL tree:



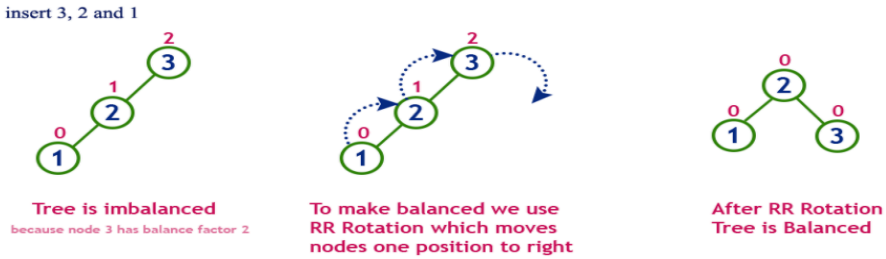
Single Left Rotation (LL Rotation)

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...



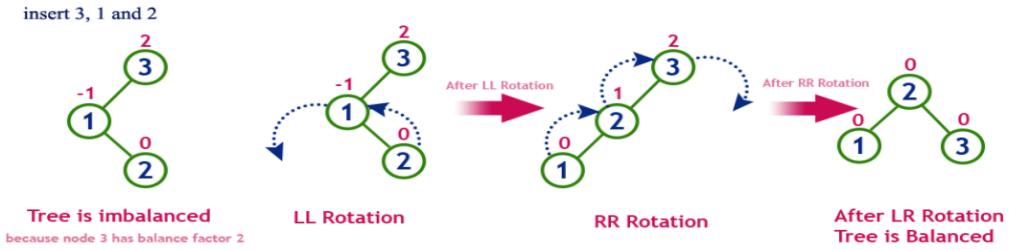
Single Right Rotation (RR Rotation)

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...



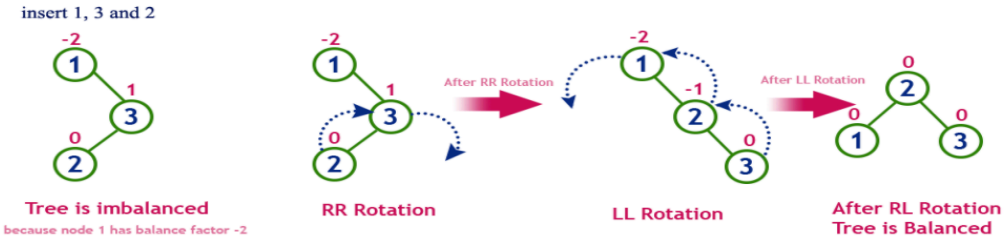
Left Right Rotation (LR Rotation)

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...



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...



Algorithm for constructing an AVL Tree:

1. Set first element of a given array of elements as first node of AVL tree.
2. Set next(second) node either left side or right side of given AVL tree.
    a) if node (1) > node (2)
        Set node (2) in left side of node (1)
    b) else
        Set node (2) in right side of node (1)
3. Continue the process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either -1 or 0 or 1.
4. If balance factor of a particular node is not in given range (-1 to 1)
    a) if straight path take place
        perform single rotation
    b) else
        perform double rotation
5. Exit

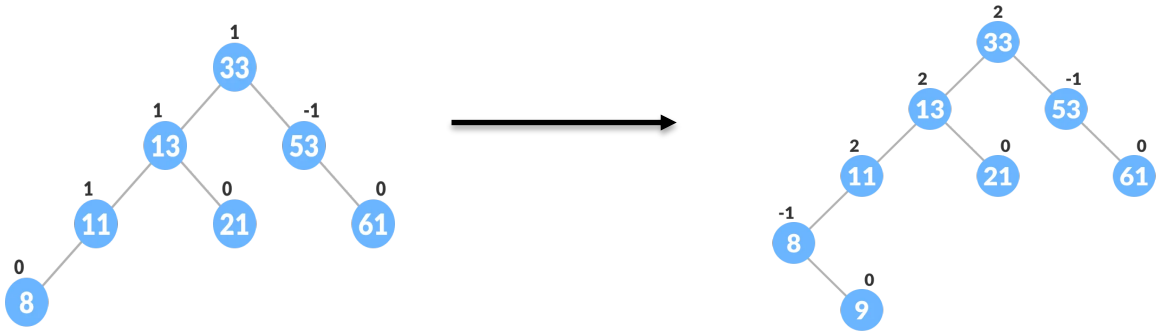
Algorithm to insert an element in an AVL tree:

1. Insert the node in the same way as in an ordinary binary tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

Function to insert an element in an AVL tree:

```
Node insert (Node node, int value)
{
        if (node == null)
                return new Node(value);
        if (value < node.value)
                node.left = insert(node.left, value);
        else if (value > node.value)
                node.right = insert(node.right, value);
        else
                 return node;
        node.height = 1 + Math.max(height(node.left), height(node.right));
        int balance = height(node.left) - height(node.right)
        if (balance > 1 && value < node.left.value)
                return rightRotate(node);       // Left Left Case
        if (balance < -1 && value > node.right.value)
                return leftRotate(node);        // Right Right Case
        if (balance > 1 && value > node.left.value)
        {
                node.left = leftRotate(node.left);              // Left Right Case
                return rightRotate(node);
        }
        if (balance < -1 && value < node.right.value)
        {
                node.right = rightRotate(node.right);           // Right Left Case
                return leftRotate(node);
        }
        return node;
  }
```

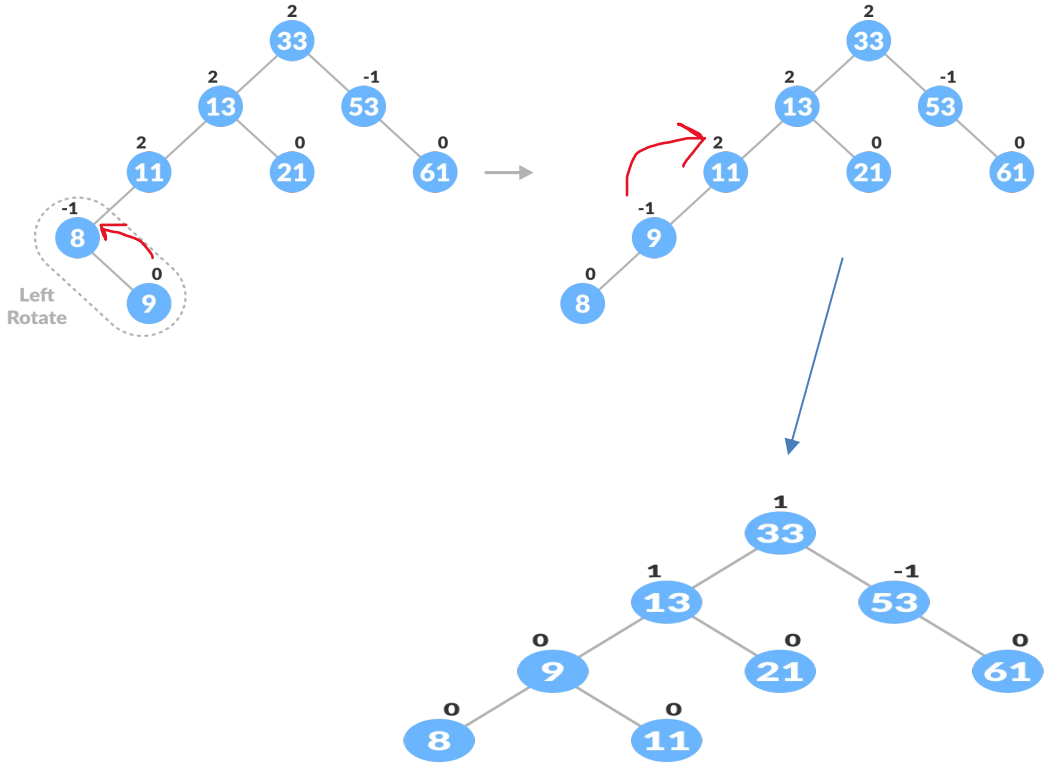
Example: ***Insert 9***



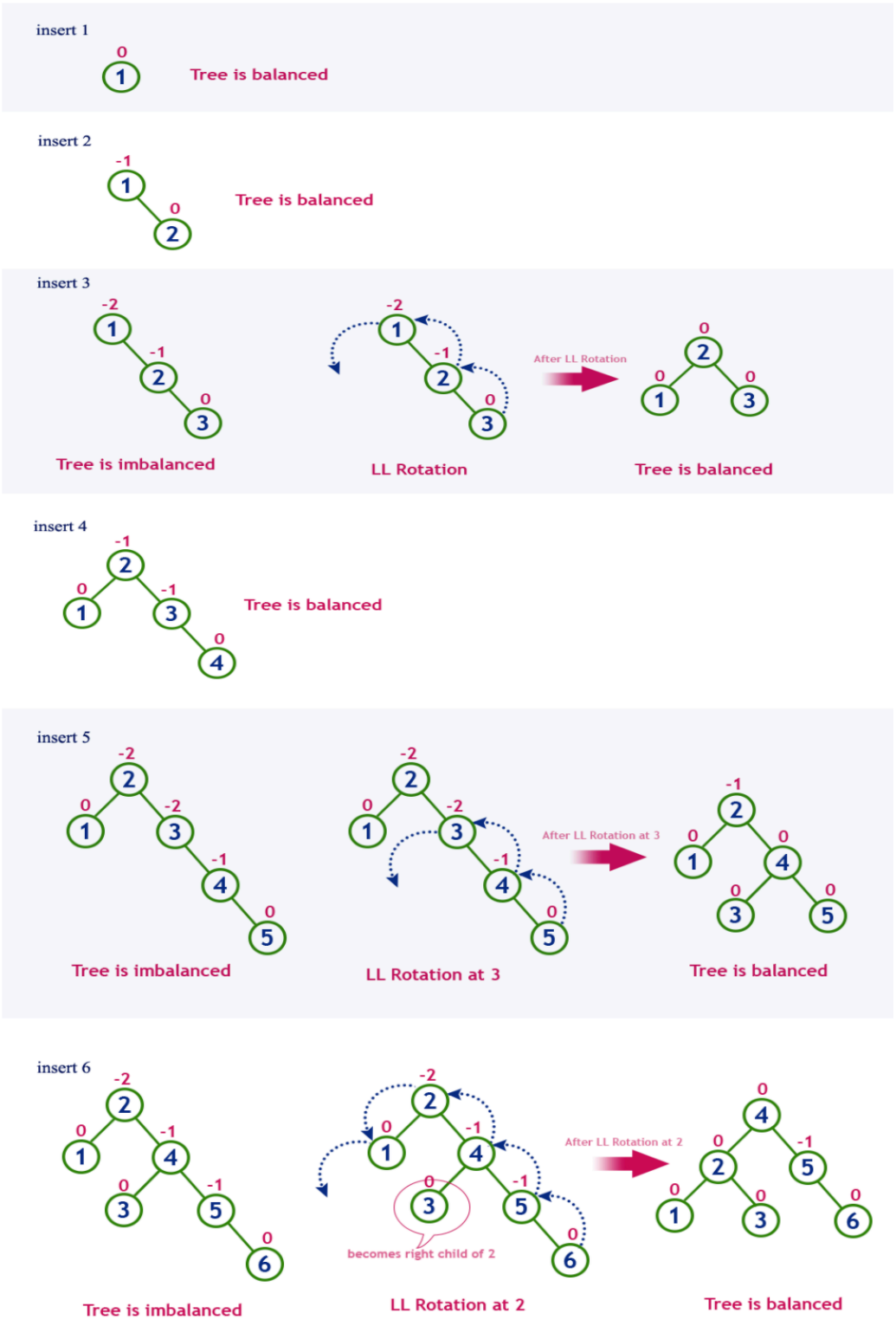
Updating Balance Factor:

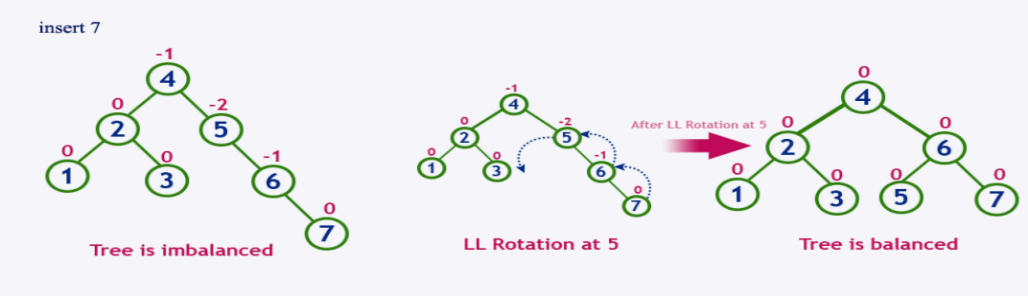
Since ***balance_factor > 1 && value > node.left.value***

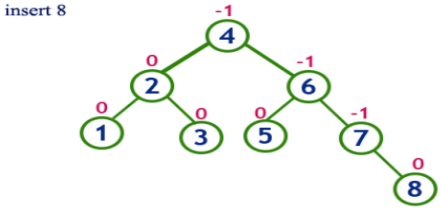
Perform left-right rotation



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







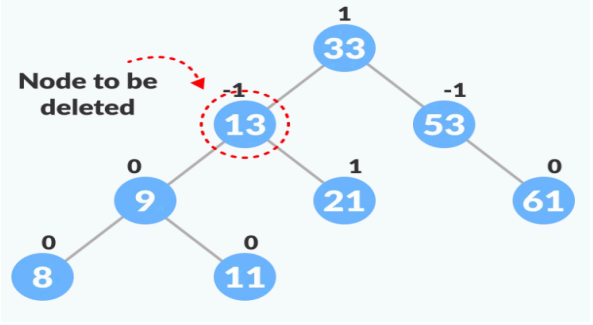
Deleting an element from an AVL tree:

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

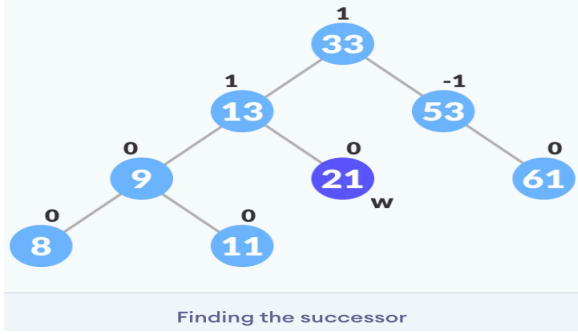
Algorithm:

1. Locate a node to be deleted *nodeToBeDeleted.*
2. There are three cases for deleting a node:
    a) if *nodeToBeDeleted* is the leaf node
        remove *nodeToBeDeleted*
    b) if *nodeToBeDeleted* has one child
        substitute the contents of *nodeToBeDeleted* with that of the child
        remove the child
    c) if *nodeToBeDeleted* has two children
        find the inorder successor of *nodeToBeDeleted*
        substitute the contents of *nodeToBeDeleted* with that of inorder successor
        remove the inorder successor
3. Update the balance factor of the nodes.
4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1 or 0 or 1.
    a) if balance factor of currentNode > 1
        i) if balance factor of leftChild >= 0
            do right rotation
        ii) else
            do left-right rotation
    b) if balance factor of currentNode < -1
        i) if balance factor of rightChild <=0
            do left rotation
        ii) else
            do right-left rotation
5. Exit

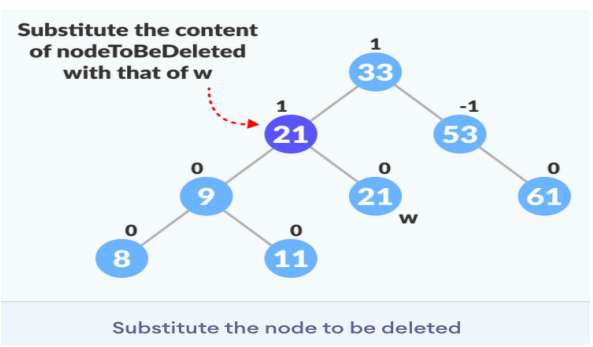
Example:



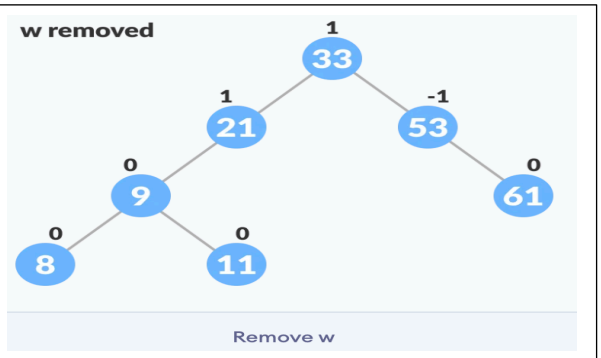
(a)



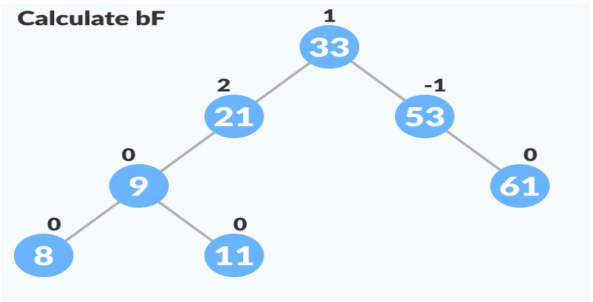
(b)



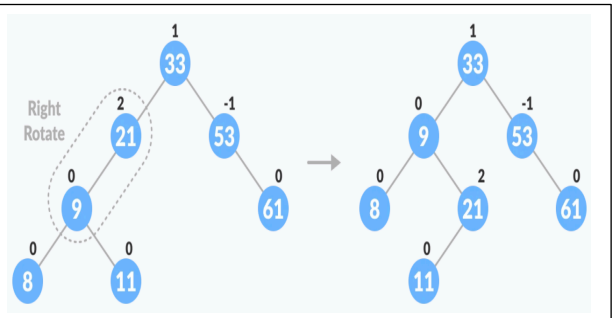
(c)



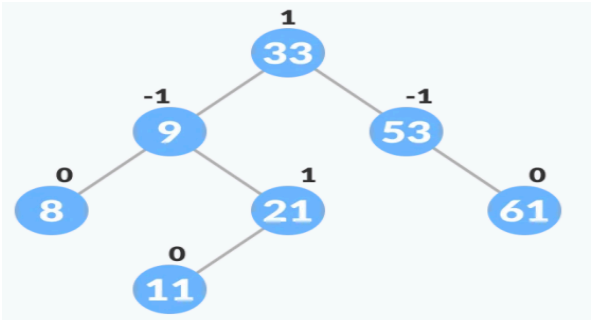
(d)



(e)



(f)



(g) AVL Tree Final

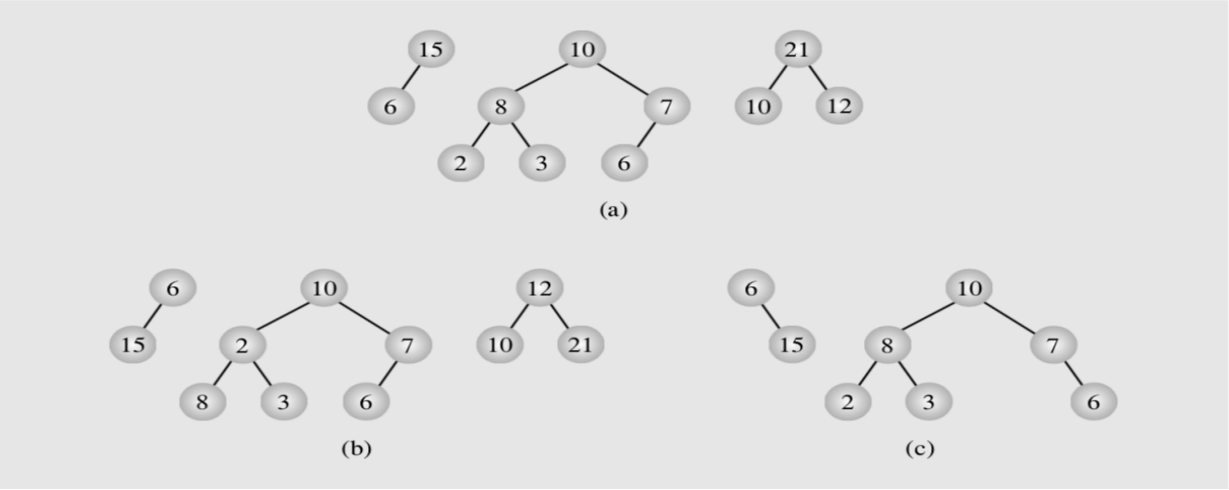
Heap Tree

Heap tree (max heap) is a particular kind of binary tree which has the following two properties.

1. The value of each node is greater than or equal to the values of stored in each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

If "greater" in the first properties is replaced with "less" than the definition specifies a min heap. This means that the root of a max heap contains the largest element, whereas the root of a min heap contains the smallest. A tree has the heap property if each non-leaf has the first property. Due to the second condition, the number of leaves in the tree is O(logn).

FIGURE 6.51 Examples of (a) heaps and (b–c) nonheaps.

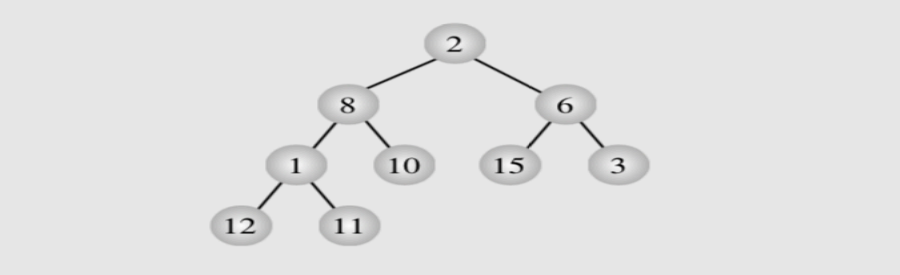


Interestingly, heaps can be implemented by arrays. The elements are placed at sequential locations representing the nodes from top to bottom and in each level from left to right. The second property reflects the fact that the array is packed, with no gaps. Now, a heap can be defined as an array heap of length n in which:

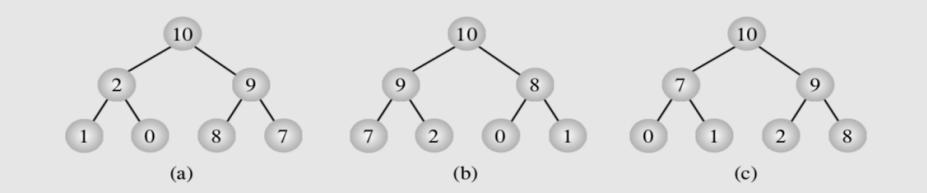
Heap[i] ≥ heap[2.i+1], for 0≤i<(n-1)/2

And heap[i] ≥ heap[2.i+1], for 0≤i<(n-2)/2.

The array [2 8 6 1 10 15 3 12 11] seen as a tree.



Elements in a heap are not perfectly ordered. We know only that the largest element is in the root node and that, for each node, all its descendants are less than or equal to that node. But the relation between siblings (brothers) nodes or to continue the kinship terminology, between uncle and nephew nodes is not determined. The order of the elements obeys a linear line of descent, disregarding lateral lines.



Heaps as Priority Queues

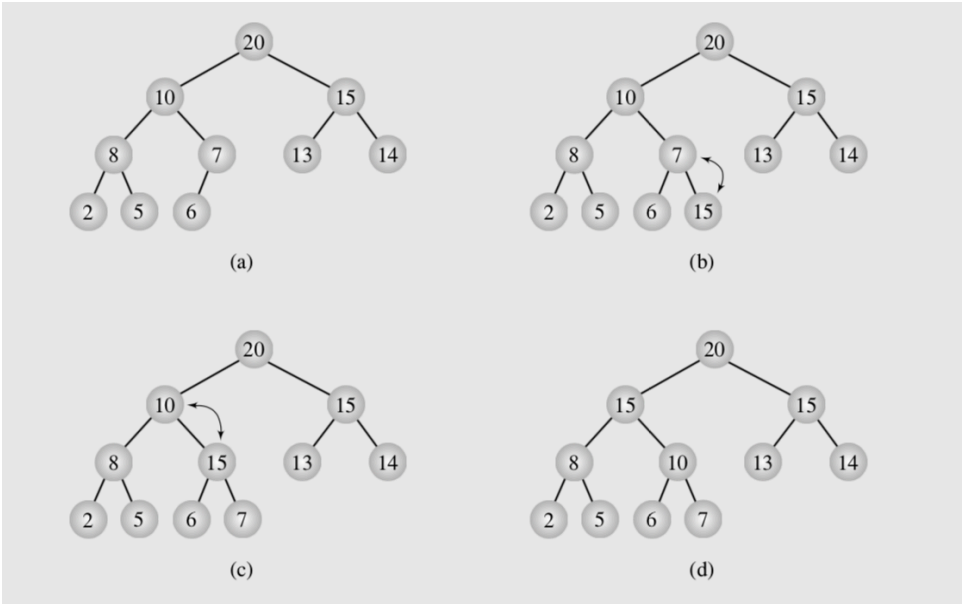
A heap is an excellent way to implement a priority queue. Priority queue can be implemented using linked list for which the complexity was expressed in O(n). For large n , this may be too inefficient. On the other hand, a heap is a perfectly balanced tree, hence, reaching a leaf requires O(logn) searches. This efficiency is promising. Therefore, heaps can be used to implement priority queue. To this end, however, two procedures have to be implemented to enqueue and dequeue elements on a priority queue.

To enqueue an element, the element is added at the end of the heap as the last leaf. Restoring the heap priority in the case of enqueuing is achieved by moving from the leaf towards the root.

The algorithm for enqueuing is as follows:

heapEnqueue (el)
put el at the end of heap;
while el is not in the root and el > parent (el)
swap el with its parents;

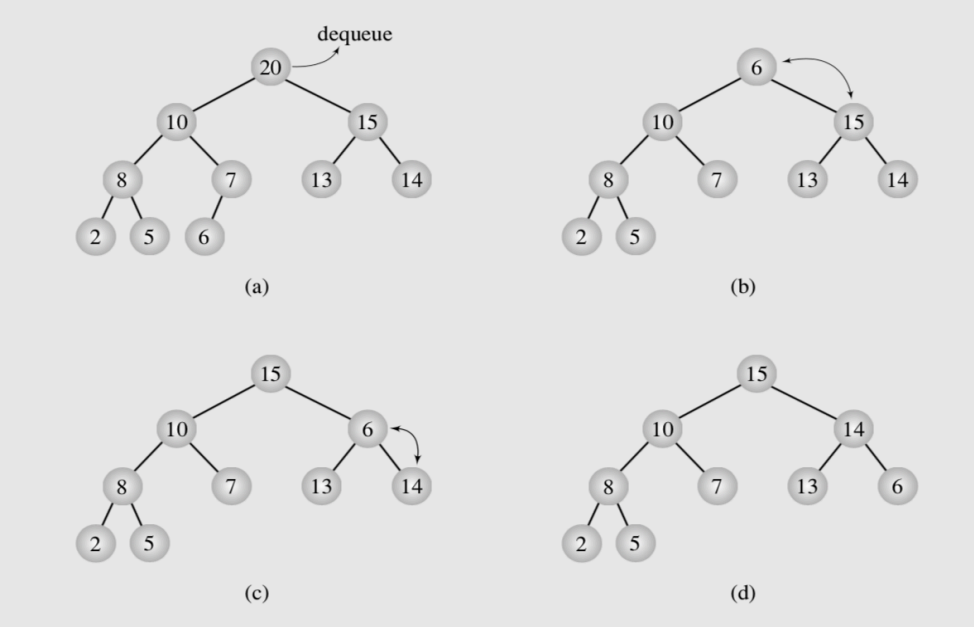
Enqueuing an element 15 to the heap



Dequeuing an element from the heap consists of removing the root element from the heap, because by the heap property it is the element with the greatest priority. Then the last leaf is put in its place and the heap property almost certainly has to be restored, this time by moving from the root down the tree.

The algorithm for Dequeuing:

heapDequeue()
    extract the element from the root;
    put the element from the last leaf in its place;
    remove the last leaf;
    p = the root;
    while p is not a leaf and p < any of its children
        swap p with the larger child;



Organizing Arrays as Heaps

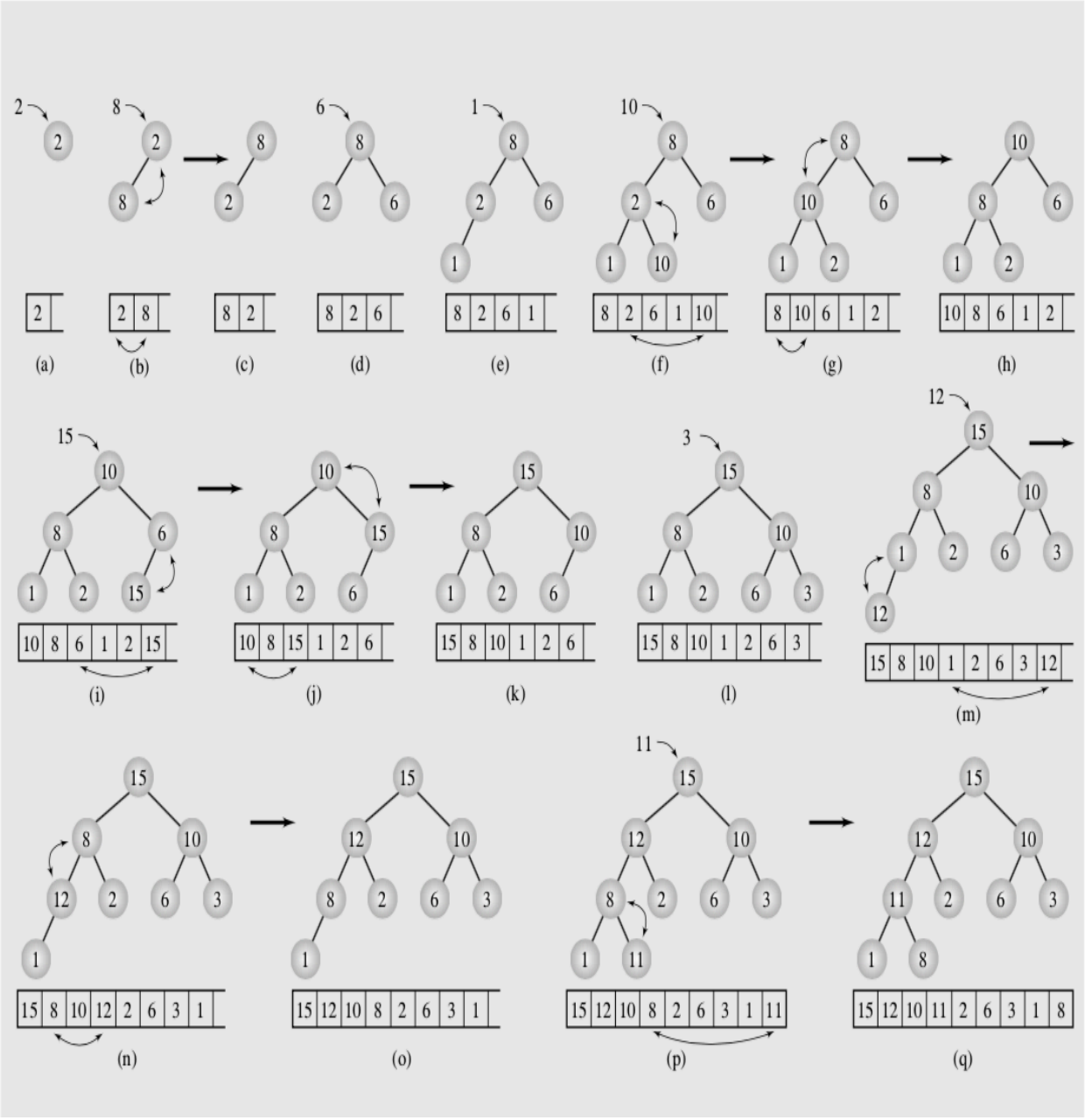
Heaps can be implemented as arrays, and in that sense, each heap is an array, but all arrays are not heaps. In some situations, however, most notably in heap sort, we need to convert an array into a heap. There are several ways to do this which are as follows:

1. **<u>Top-Down Method:</u> (the simplest way)**

- Starts with an empty heap and sequentially include elements into a growing heap.
- It extends the heap by enqueuing new element in the heap.
- It was proposed by John Williams

Example:

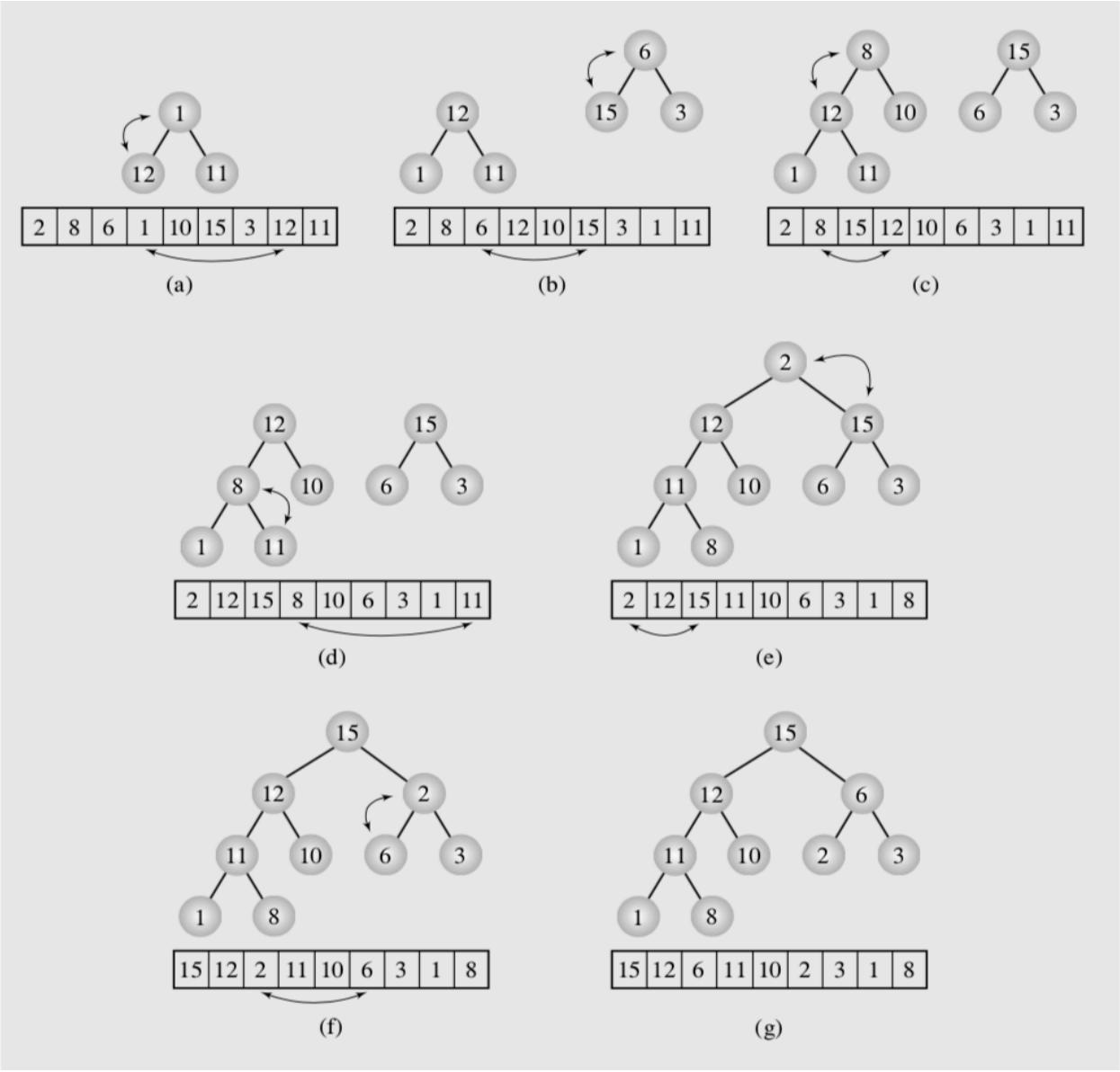
Organizing an array [2,8,6,1,10,15,3,12,11] as a Heap



2. **<u>Bottom-up Method:</u> (Floyd Algorithm)**

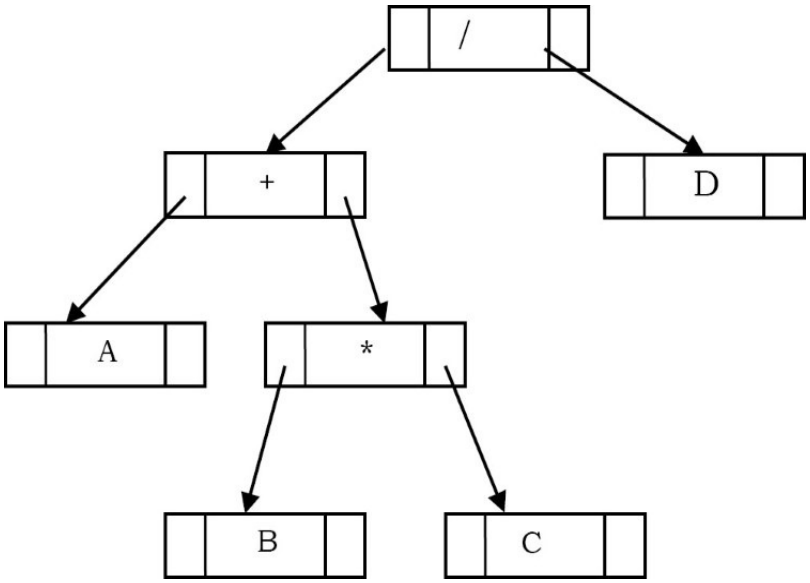
- In this approach, small heaps are formed and repetitively merged into larger heaps.
- Developed by Robert Floyd.

FloydAlgorithm(data [])
for i = index of the last nonleaf down to 0
restore the heap property for the tree whose root is data[i] by calling moveDown(data,i,n-1);



Expression Tree:

- A tree representing an expression is called an *expression tree*.
- In expression trees, leaf nodes are operands and non-leaf nodes are operators.
- That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands.
- An expression tree consists of binary expression. But for a unary operator, one subtree will be empty.
- The figure below shows a simple expression tree for (A + B * C) / D:

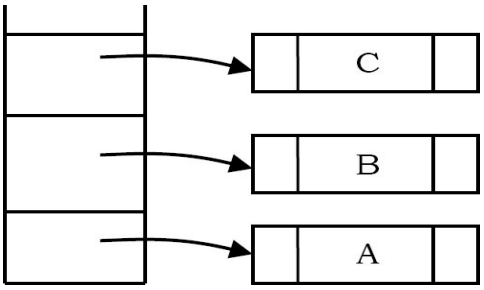


Building Expression Tree from Postfix Expression

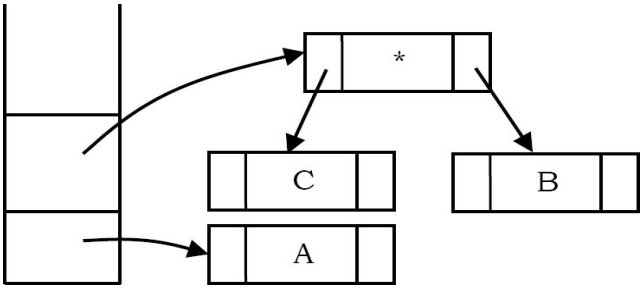
- Assume that one symbol is read at a time.
- If the symbol is an operand, we create a tree node and push a pointer to it onto a stack.
- If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively.
- A pointer to this new tree is then pushed onto the stack.

Example: Input = A B C * + D /

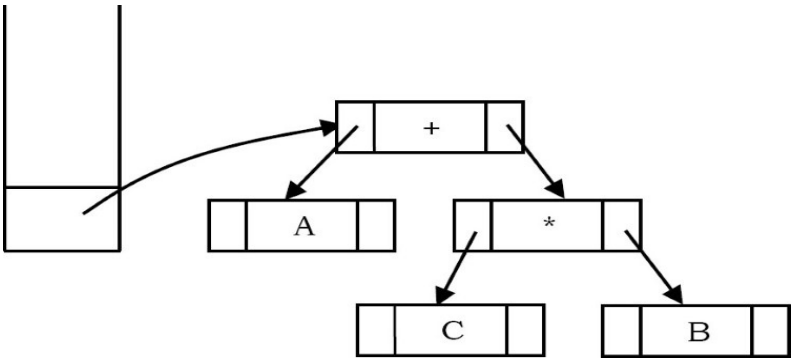
The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below:



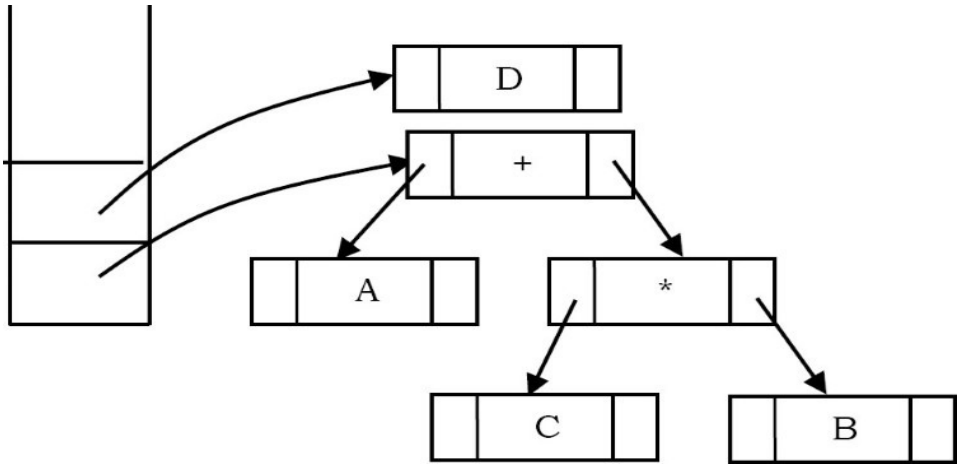
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



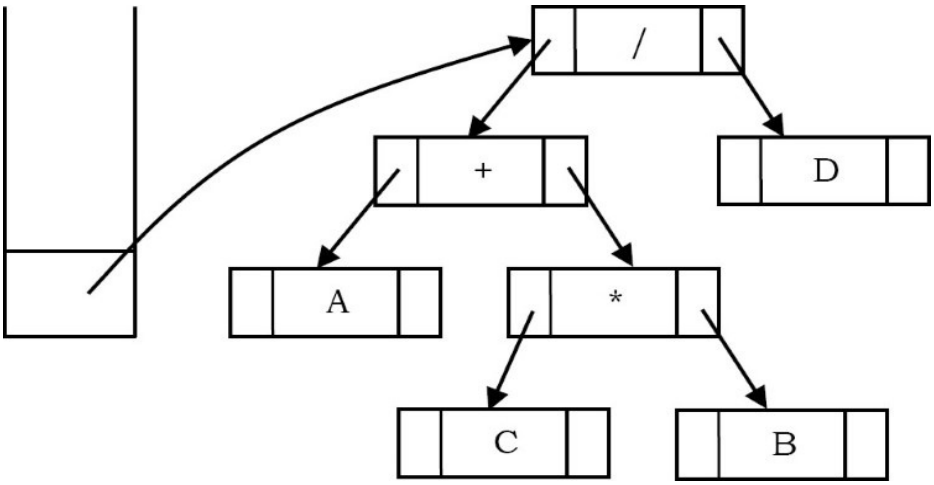
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



Next, an operand ‘D’ is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Finally, the last symbol (‘/’) is read, two trees are merged and a pointer to the final tree is left on the stack.



The Huffman Algorithm:

- Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.
- Developed by David Huffman in 1951.
- This algorithm is applicable to many forms of data transmission.
- It is now standard method used for data compression.

Algorithm:

- Initially two nodes with smallest weights are considered and their sum forms their parent node.
- When a new element is considered, it can be added to the tree.
- Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.
- The final single tree is an extended binary tree.

Example:

Characters	Frequencies
A	10
E	15
I	12
O	3
U	4
S	13
T	1

Since There are 7 characters, so 3 bit is sufficient. Thus, take initially 3 bits for each character which is called fixed length character.

Now sort the above characters according to their frequencies in non-decreasing order as below:

Characters	Frequencies	Code
t	1	000
o	3	001
u	4	010
a	10	011
i	12	100
s	13	101
e	15	110

Total number of bits required before using Huffman algorithm (NB)

= 1*3 + 3*3 + 4*3 + 10*3 + 12*3 + 13*3 + 15*3 = 174 bits

Now using Huffman Algorithm:

Step 1:

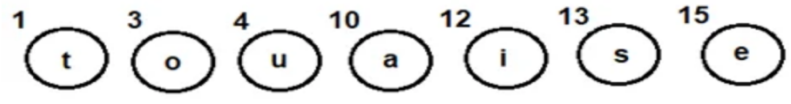


Figure 1: Leaf nodes for each character

Step 2: Extract two nodes (t and o) with minimum frequency and their sum forms a new internal node with t as its left child and 0 as its right child.

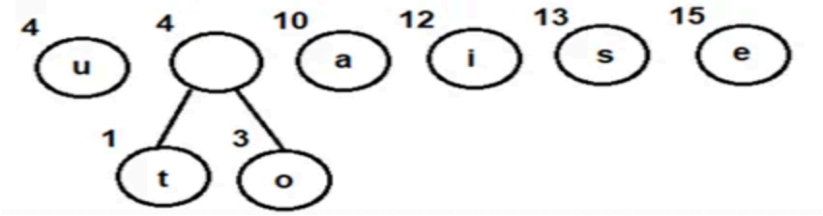


Figure 2: Combining nodes t and o

Step 3: Extract and combine node u with an internal node having frequency 4 and add the new internal node

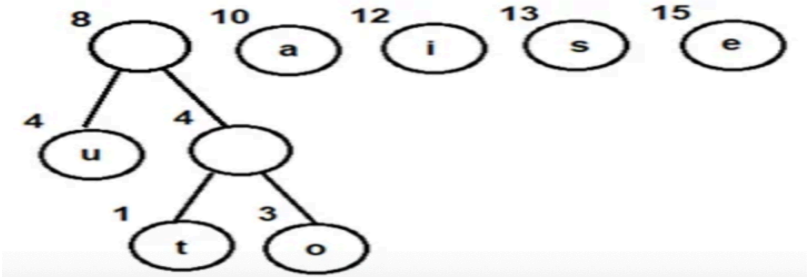


Figure 3: Combining node u with an internal node having frequency 4

Step 4: Extract and combine node a with an internal node having frequency 8 and add the new internal node.

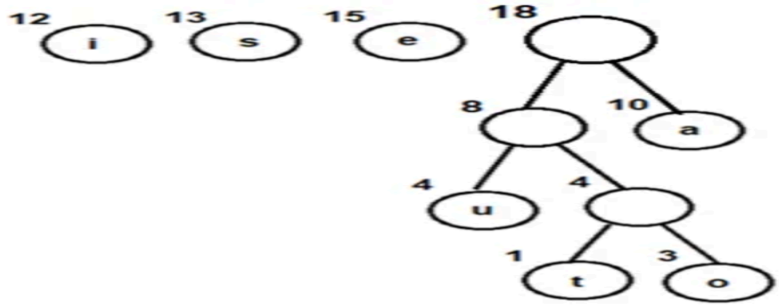


Figure 4: Combine node a with an internal node having frequency 8

Step 5: Extract and combine nodes i and s and forms a new internal node

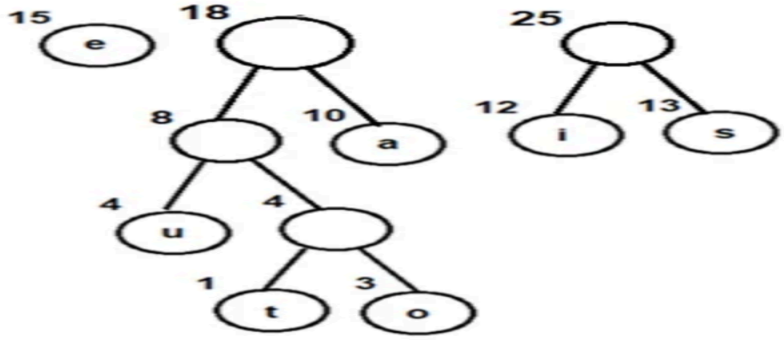


Figure 5: Combining node i and s

Step 6: Extract and combine node e with an internal node having frequency 18 and add the new internal node.

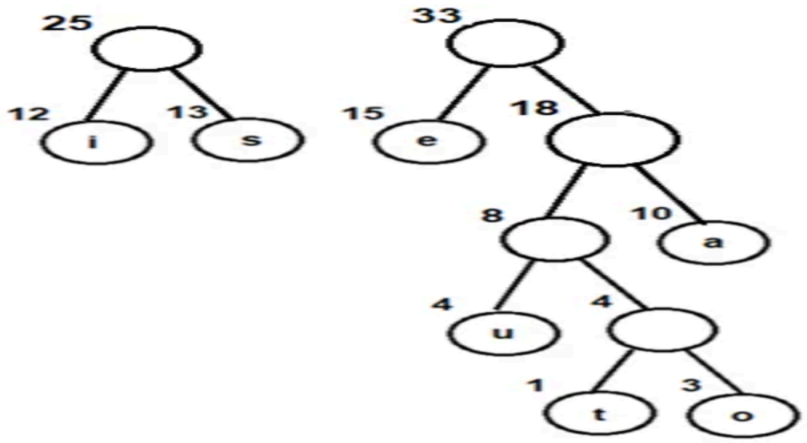


Figure 6: Combining node e with an internal node having frequency 18

Step 7: Finally, Extract and Combine internal nodes having 25 and 33 as the frequency and add the new internal node.

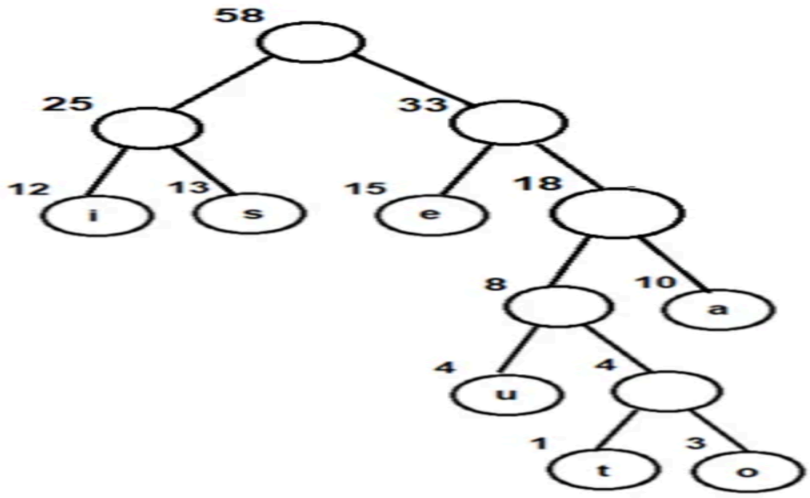
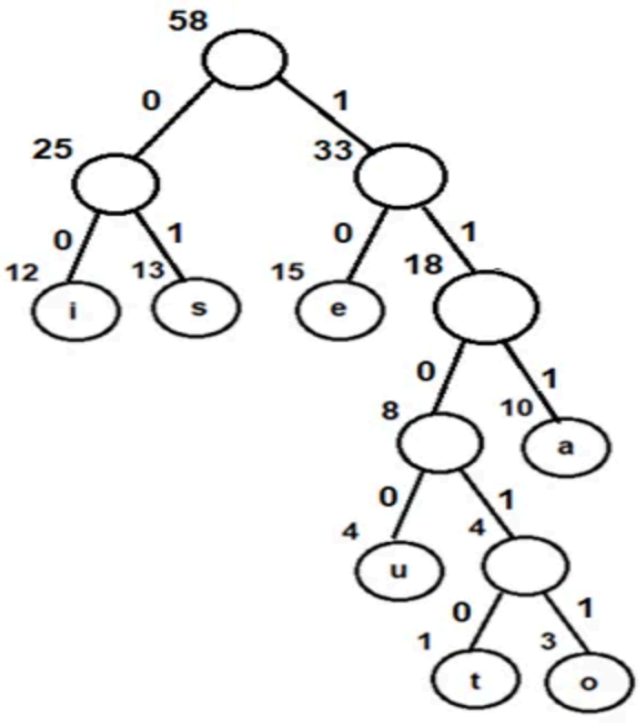


Figure 7: Final Huffman Tree

Step 8: Assigning Binary codes to Huffman tree. Left branch is 0 and right branch is 1.



Characters	Binary Codes
i	00
s	01
e	10
a	111
u	1100
t	11010

Now from the variable length code we get following code sequence:

Characters	Frequencies	Code
t	1	11010
o	3	11011
u	4	1100
a	10	111
i	12	00
s	13	01
e	15	10

Total number of bits required after using Huffman algorithm (NB)
= 1*5 + 3*5 + 4*4 + 10*3 + 12*2 + 13*2 + 15*2 = 146 bits

Reduced bits = 174 – 146 = 28 bits
% Reduced bits = 28/174*100% = 16.01%

Hence Huffman algorithm is used to compress data.

Multiway Trees:

A multiway tree is a tree that can have more than two children. This tree is called a multiway tree of order m or an m-way tree.

A multiway (m-way) search tree of order m or an m way search tree, is a multiway tree in which

1. Each node has m children and m-1 keys.
2. The keys in each node are in ascending order.
3. The keys in the first i children are smaller than the i^{th} key.
4. The keys in the last m-i children are larger than the i^{th} key.

The m-way search trees are used for the purpose of fast information retrieval and update.

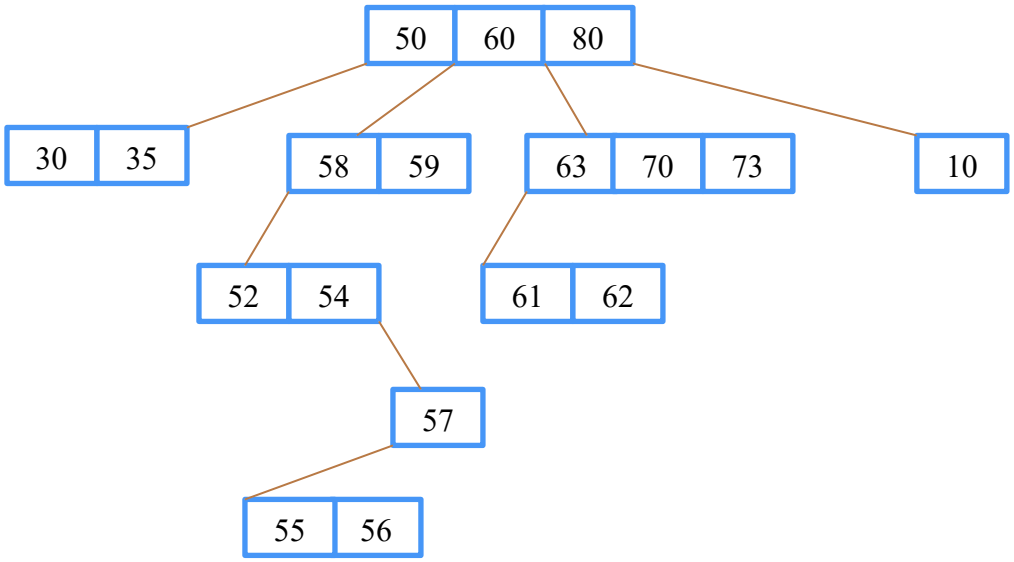


Figure: A 4 - way Tree

To access a large amount of data at one time than to jump from one position on the disk to another position to transfer small portions of data, the new trees such as B-tree, B*-tree and B+ -tree were introduced

B-Trees:

To understand the use of B-Trees, we must think of the huge amount of data that can not fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time.

The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations such as searching, insertion, deletion, etc. requires O(h) disk access where h is the height of tree.

B-Tree is a fat tree that means the height of B-tree is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Hence h is low for B-Tree. So, the disk access is reduced in B-Tree compared to balanced binary trees like AVL tree.

A **B -Tree** of order m is a multiway search tree with the following properties:

1. The root has at least two subtrees unless it is a leaf.
2. Each non-root and non- leaf node hold k-1 keys and k references to subtrees where $\Gamma\ m/2\ \urcorner \leq k \leq m$.
3. Each leaf node holds k-1 keys where $\lceil\ m/2 \urcorner \leq k \leq m$
4. All the leaves are on the same level.

According to these conditions, a B -Tree is always at least half full, has few levels and is perfectly balanced.

A node of a B -Tree is usually implemented as class containing an array of m-1 cells for keys, and m cell array of references to other nodes, and possibly other information facilitating tree maintenance, such as the number of keys in a node and a leaf/ non lean flag as in

Inserting a key into a B- Tree

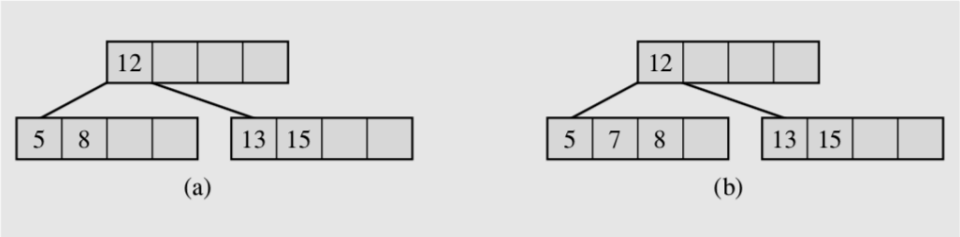
Both insertion and deletion operations appear to be somewhat challenging if we remember that all have to be at the same level. Implementing insertion becomes easier when the strategy of building a tree is changed. Insertion into B-Tree is different from insertion into BST.

B-Tree is built from the bottom up so that the root is an entity always in flux, and only at the end of all insertions, we can know the contents of the root. In this process, given an incoming key, we go directly to a leaf and place it there, if there is room. When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent. If the key is full, the process is repeated until the root is reached and a new root is created.

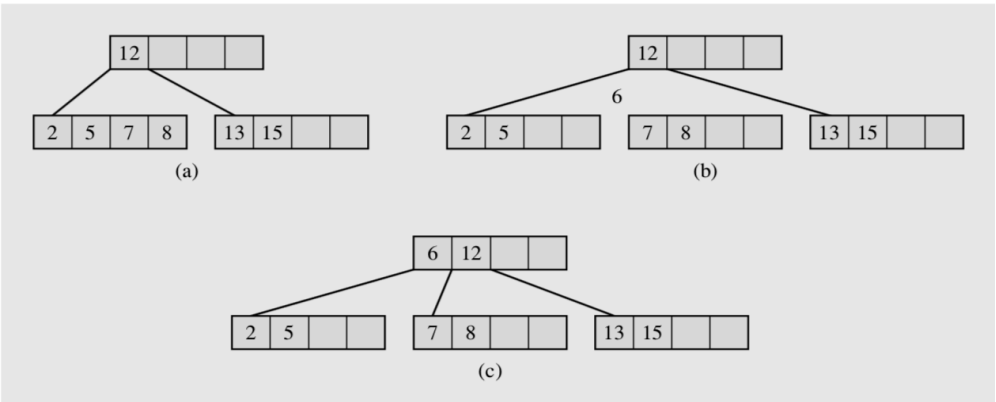
To approach the problem more systematically, there are three common situations encountered when inserting a key into a B-Tree.

1. A key is placed in a leaf that still has some room.

A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells.

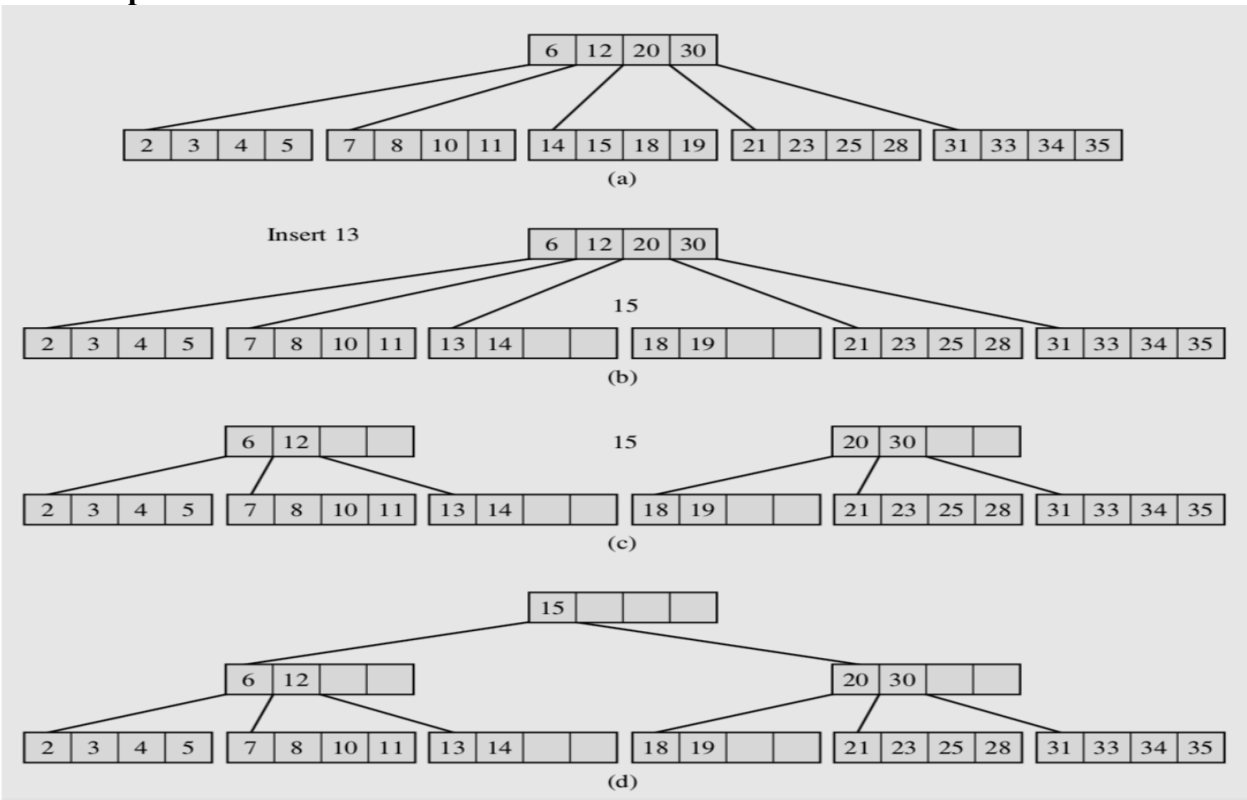


2. The leaf in which a key should be placed is full then the leaf is split, creating s new leaf, and half of the keys are moved from the full leaf to the new leaf. But the new leaf has to be incorporated into the B-Tree. The middle key is moved to the parent, and a reference to the new leaf is placed in the parent as well. The same procedure can be repeated for each internal node of the B Tree so that each such split adds one more node to the B-Tree. Moreover, such a split guarantee that each leaf never has less than $\lceil m/2 \rceil - 1$ keys.



3. A special case arises if the root of the B-Tree is full. In this case, a new root and a new sibling of the existing root have to be created. This split results in two new nodes in the B-Tree.

Example: Insert 13

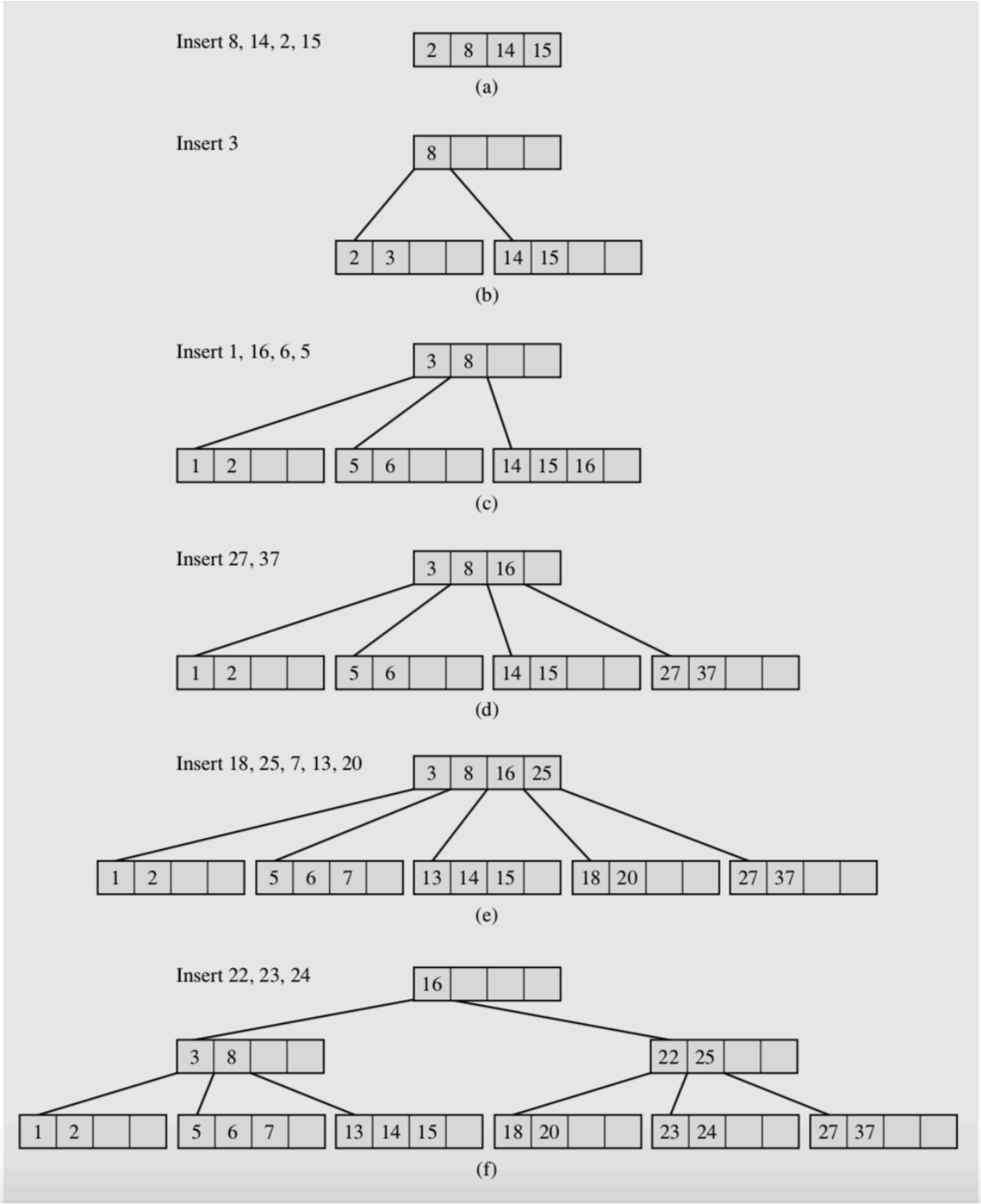


An algorithm for inserting keys in B-Trees as follows:

1. Start
2. Read key element to be inserted i.e., K
2. find a leaf node to insert K
4. while(true)
- *a. find a proper position in array keys for K;*
- *b. if node is not full*
  - *i. insert K and increment keyTally;*
  - *ii. return;*
- *c. else*
  - *i. split node into node1 and node2;*
    distribute keys and references evenly between node1 and node2 and
    initialize properly their keyTally's
  - *ii. Set, K = middle key*
  - *iii. if node was the root*
    create a new root as parent of node1 and node2;
    put K and references to node1 and node2 in the root, and set its keyTally to 1;
    return;
  - *iv. else*
    node = its parent; // and now process the node's parent;

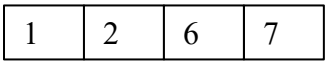
5. Stop

Create a B-Tree of order 5 from the following set of data: 8 14 2 15 3 1 16 6 5 27 37 18 25 7 13 20 22 23 24

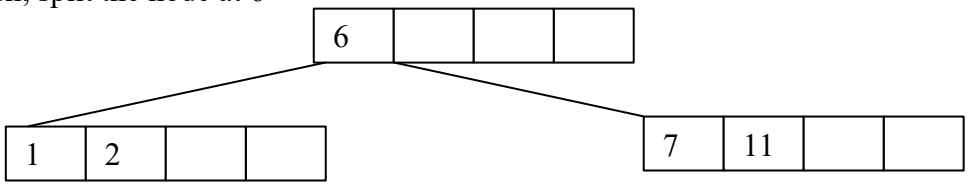


Create a B-Tree of order 5 from the following set of data: 1, 7, 6, 2, 11, 4, 8, 13,10, 5, 19 , 9, 18, 2

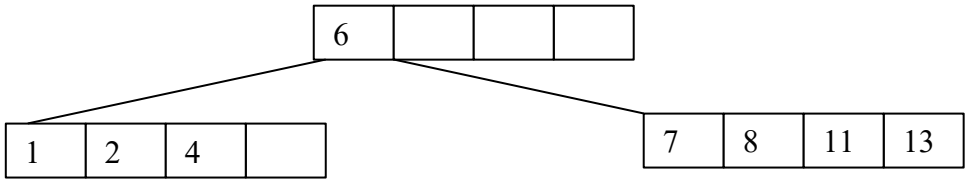
Insert: 1, 7, 6, 2



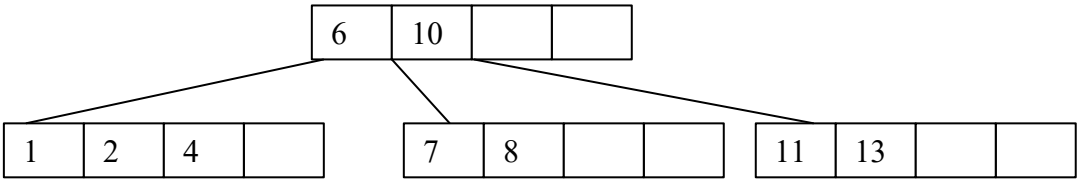
Insert 11: Node is full, split the node at 6



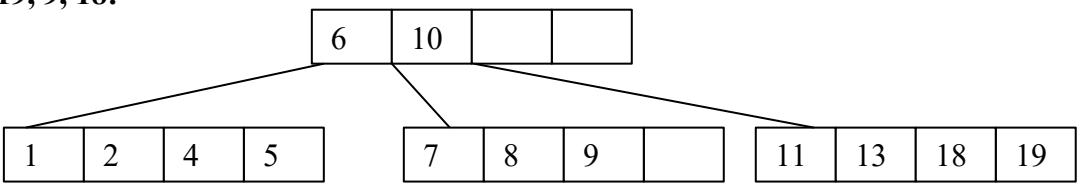
Insert 4, 8, 13



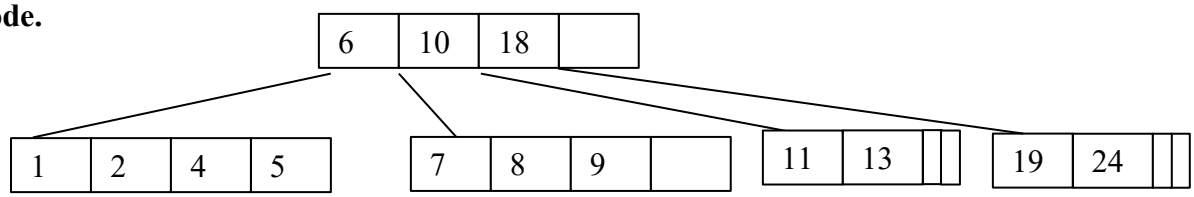
Insert 10: Node is full, split the node at median 10. It is will be joined with 6 in root node.



Insert 5, 19, 9, 18:



Insert 24: The node is full, split the node at the median 18, and it will be joined with root node.



Deleting a node from a B- Tree

Deletion is to a great extent a reversal of insertion, although it has more special cases. Care has to be taken to avoid allowing any node to be less than half full after a deletion. This means that nodes sometimes have to be merged.

In deletion, there are two main cases: deleting a key from a leaf and deleting a key from a nonleaf node.

In the later cases we use a procedure similar to deleteByCopying () used for binary search trees.

1. Deleting a key from a leaf
    - 1.1 If, after deleting a key, the leaf is at least half full and only keys greater than K are moved to the left to fill the hole, this is the inverse of insertion's case 1.
    - 1.2 If, deleting K, the number of keys in the leaf is less than ⌈m/2⌉ -1 causing an underflow:
        - 1.2.1 If there is a left or right sibling with the number of keys exceeding the minimal ⌈m/2⌉ -1, then all keys from this leaf and this sibling are redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.
        - 1.2.2 If the leaf underflows and the number of keys in its siblings is ⌈m/2⌉ -1, then the leaf and a sibling are merged; the keys from the leaf, from its sibling, and the separating key from the parent are all put in the leaf, and the sibling node is discarded. The keys in the parent are moved if a hole appears. This can initiate a chain of operations if the parent underflows. The parent is now treated as though it were a leaf and either step 1.2.2 is repeated until step 1.2.1 can be executed or the root of the tree has been reached. This is the inverse of insertion's case 2.
            - 1.2.2.1 A particular case results in merging a leaf or non-leaf with its sibling when its parent is the root with only one key. In this, case the keys from the node and its sibling, along with the only key of the root, are put in the node, which becomes a new root, and both the sibling and the old root nodes are discarded. This is the only case when two nodes disappear at one time. Also, the height of the tree is decreased by one. This is the inverse of insertion's case 3.

2. Deleting a key from a non-leaf. This may lead to a problem with tree reorganization. Therefore, deletion from a non-leaf is reduced to deleting a key from a leaf. The key to be replaced by its immediate predecessor (the successor could also be used), which can only be found in a leaf. This successor key is deleted from the leaf, which brings us to the preceding case 1

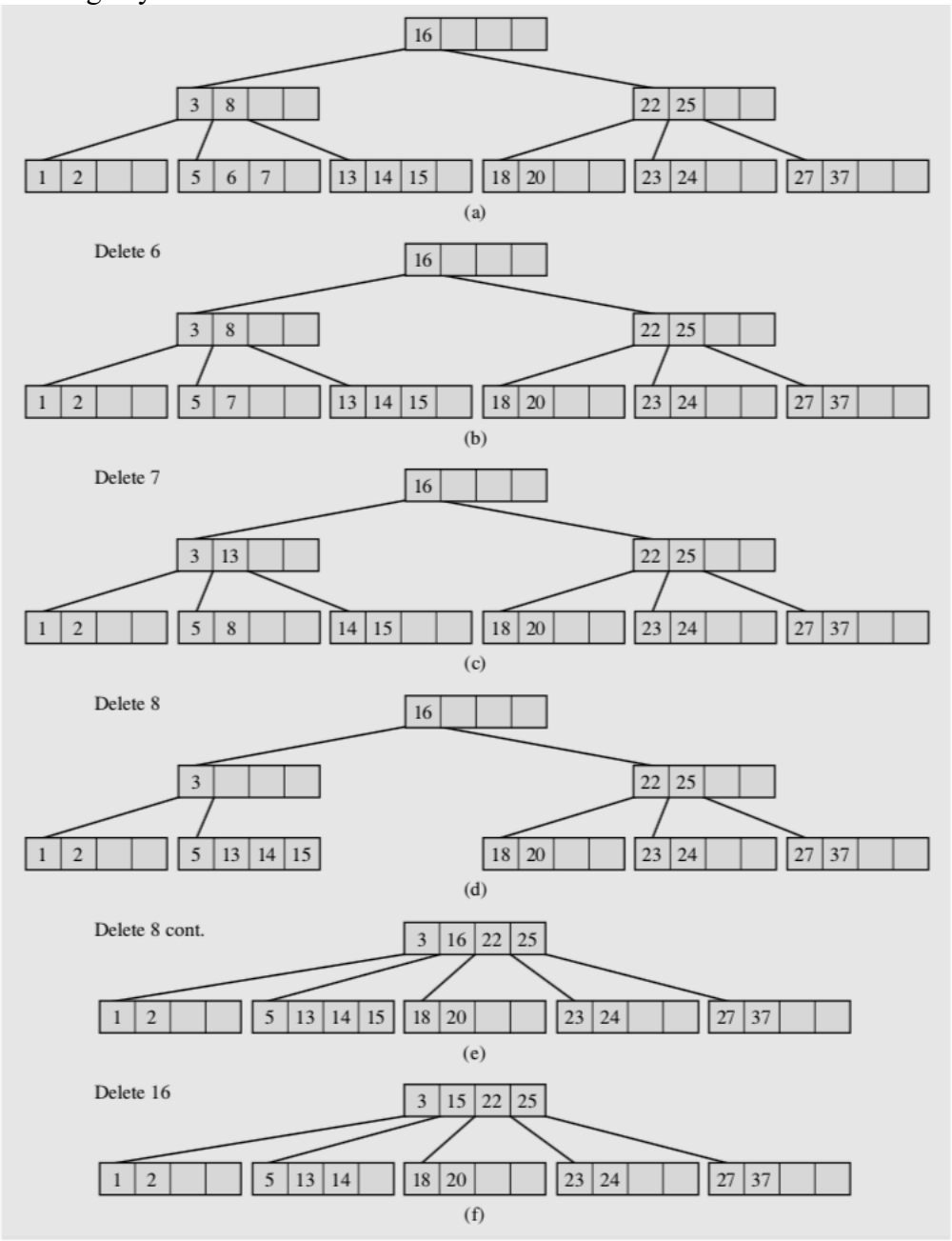
The deletion algorithm is as follows:

1. Start
2. Let K be the key element to be deleted
3. Node = BTreeSearch(k,root);
4. if (node != null)
    a. if node is not a leaf
        i. Find a leaf with the closest predecessor S of K;
        ii. Copy S over K in node;
        iii. node = the leaf containing S;
        iv. delete S from node;
    b. else
        i. delete K from node;
        ii. while(true)

- if node does not overflow
        a. return;
- else if there is a sibling of node with enough keys redistribute the keys between node and its sibling
        b. return;
- else if node's parent is the root
        c. if the parent has only one key
            merge node, its sibling and the parent to form a new root;
- else
        merge node and its sibling;
            return;

5. else merge node and its sibling
    Node = its parent;
6. Stop

Deleting keys from a B-Tree



Create a B-Tree of order 5 with following data: 10,70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90,180, 240, 30, 120, 140, 200, 210, 160. Then delete following elements from that tree: 80, 180, 160, 40