UNIT-V

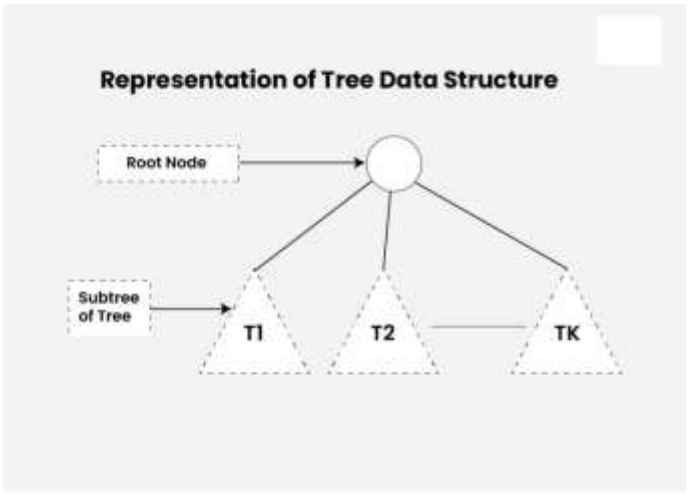
Trees: Introduction to Trees, binary tree and its properties, binary tree representation. Binary Search Tree – Insertion, Deletion & Traversal.
Height Balanced Search Tree: AVL Tree and operations.

Hashing: Brief introduction to hashing and hash functions, Collision resolution techniques: chaining and open addressing, Hash tables: basic implementation and operations, Applications of hashing in unique identifier generation, caching, etc.

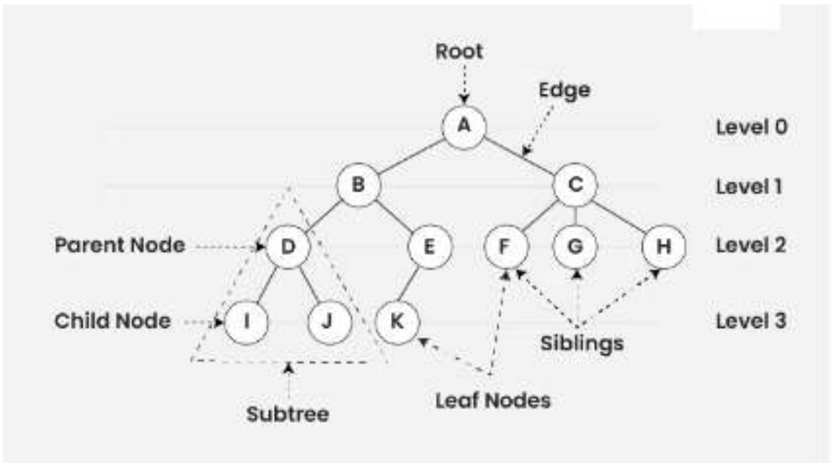
UNIT-V-I

1.Introduction to Trees

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected**.**



Tree representation with terminologies



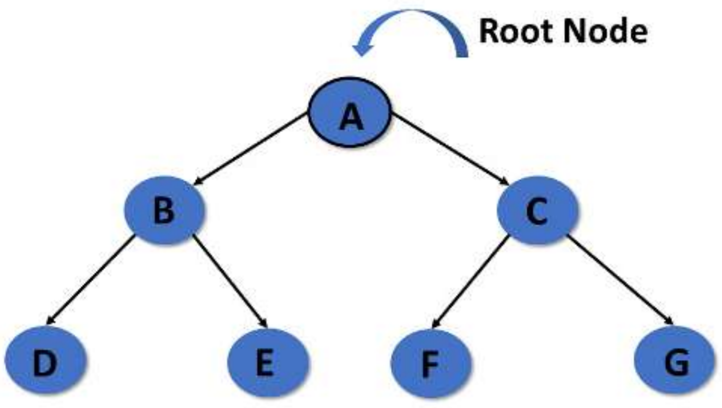
Tree Node

A node is a structure that contains a key or value and pointers in its child node in the tree data structure.

Tree Terminologies

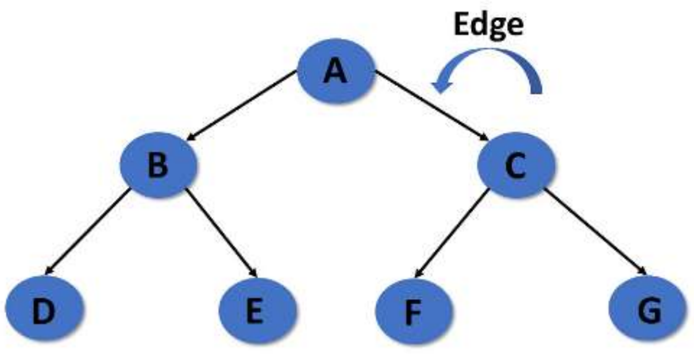
Root node

- In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures.
- The topmost node of a tree or the node which does not have any parent node is called the root node.
- In the tree data structure, there must be only one root node.



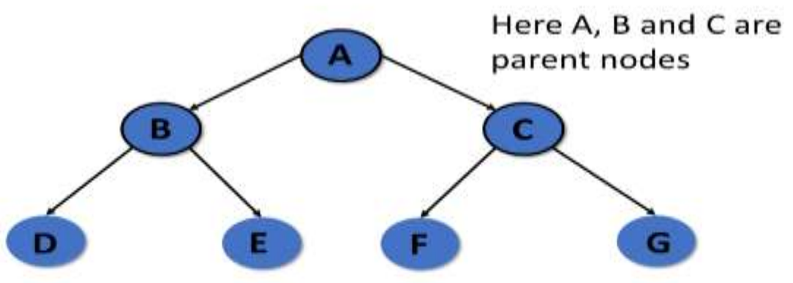
Edge

- In a tree in data structures, the connecting link of any two nodes is called the edge of the tree data structure.
- In the tree data structure, N number of nodes connecting with N -1 number of edges.



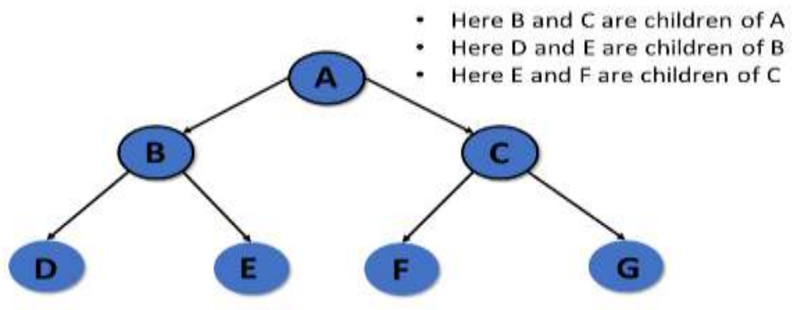
Parent node

In the tree in data structures, the node that is the predecessor of any node is known as a parent node, or a node with a branch from itself to any other successive node is called the parent node



Child node

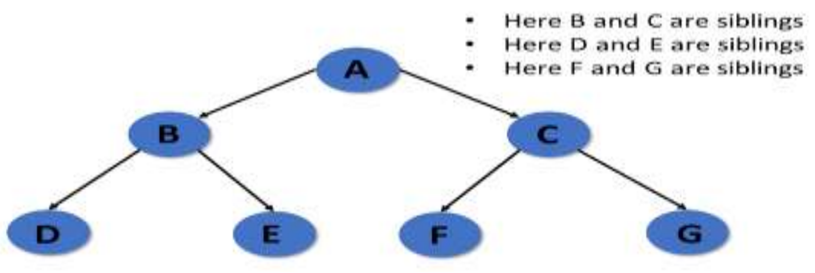
- The node, a descendant of any node, is known as child nodes in data structures.
- The node which is the immediate successor of a node is called the child node of that node
- In a tree, any number of parent nodes can have any number of child nodes.
- In a tree, every node except the root node is a child node.


• Here B and C are children of A
• Here D and E are children of B
• Here E and F are children of C
A
B
C
D
E
F
G

Siblings

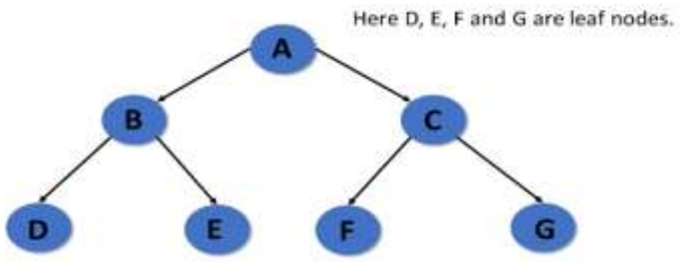
In trees in the data structure, nodes that belong to the same parent are called siblings.

Children of the same parent node are called siblings.


• Here B and C are siblings
• Here D and E are siblings
• Here F and G are siblings
A
B
C
D
E
F
G

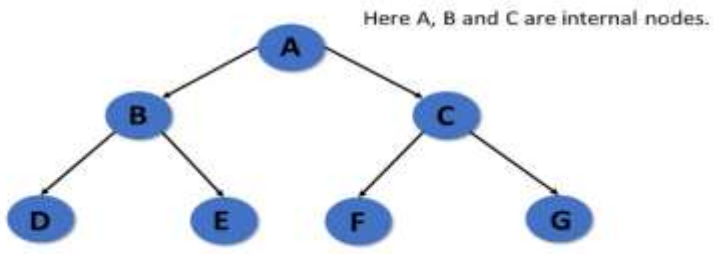
Leaf node

- Trees in the data structure, the node with no child, is known as a leaf node.
- In trees, leaf nodes are also called external nodes or terminal nodes.



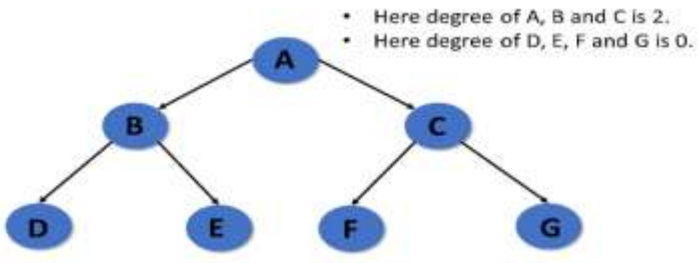
Internal nodes

- Trees in the data structure have at least one child node known as internal nodes.
- In trees, nodes other than leaf nodes are internal nodes.
- Sometimes root nodes are also called internal nodes if the tree has more than one node.



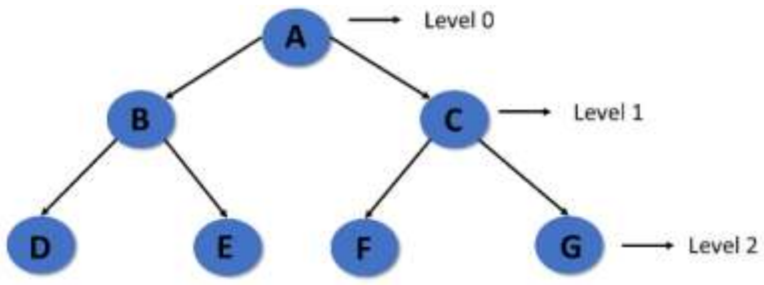
Degree

- In the tree data structure, the total number of children of a node is called the degree of the node.
- The highest degree of the node among all the nodes in a tree is called the Degree of Tree.


Here degree of A, B and C is 2.
Here degree of D, E, F and G is 0.
A
B
C
D
E
F
G

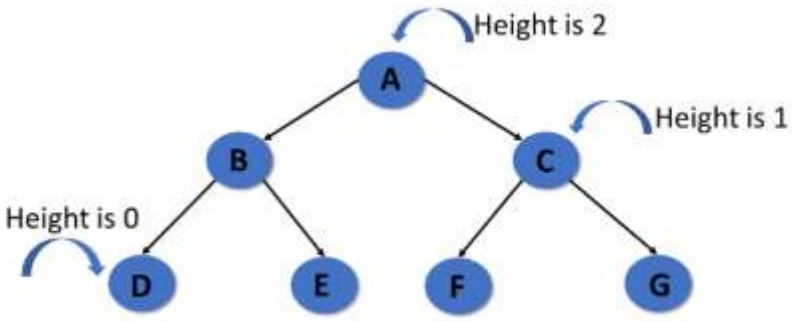
Level

In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



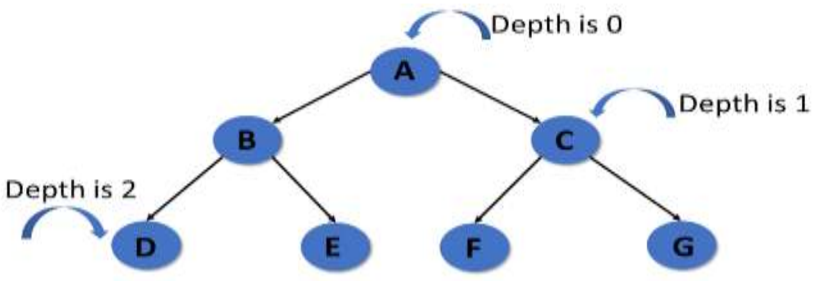
Height

- In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node.
- In the tree, the height of the root node is called "Height of Tree".
- The tree height of all leaf nodes is 0.



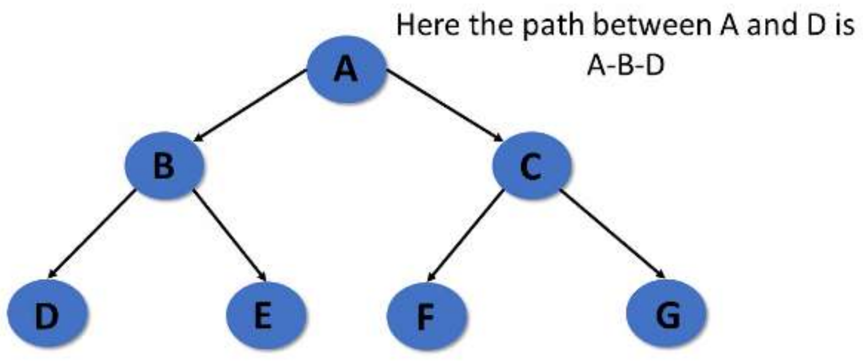
Depth

- In a tree, many edges from the root node to the particular node are called the depth of the tree.
- In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree".
- In the tree data structures, the depth of the root node is 0.



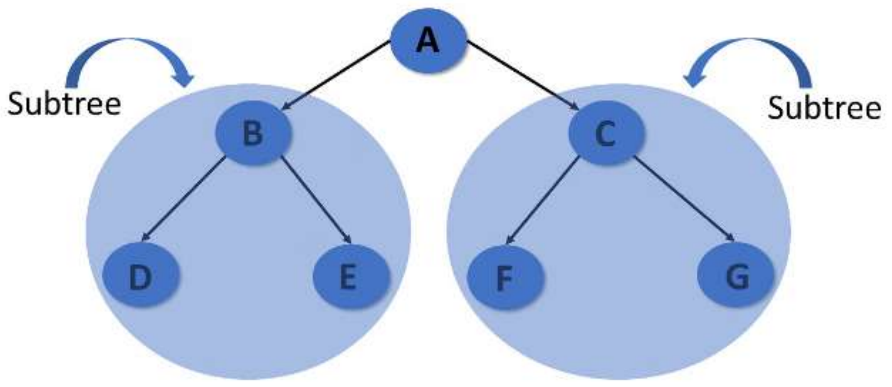
Path

- In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes.
- The length of a path is the total number of nodes in a path.



Sub tree

In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.



2.Binary tree

2.1.Introduction

Binary tree is a tree data structure (non-linear) in which each node can have at most two children which are referred to as the left child and the right child.

The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.

Binary tree: In a binary tree, each node can have a maximum of two children linked to it.

Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.

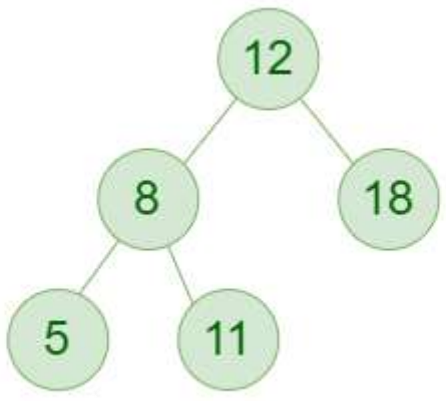
Types of Binary Tree

- Full binary trees
- Complete binary trees
- Perfect binary tree

Full binary trees

A full binary tree is a binary tree with either zero or two child nodes for each node.

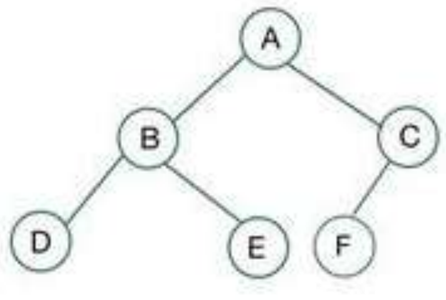
A full binary tree, on the other hand, does not have any nodes that have only one child node



Full binary tree

Complete binary trees

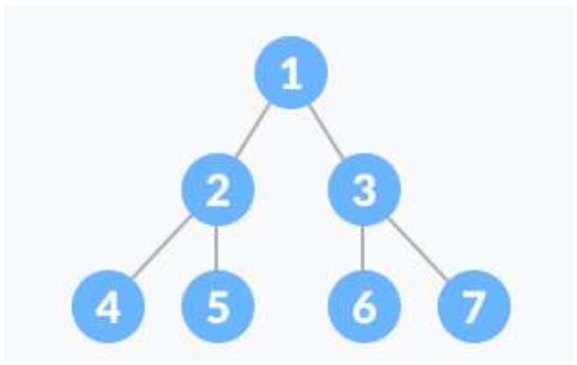
A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.



Complete Binary Tree

Perfect binary tree

A **perfect binary tree** is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.

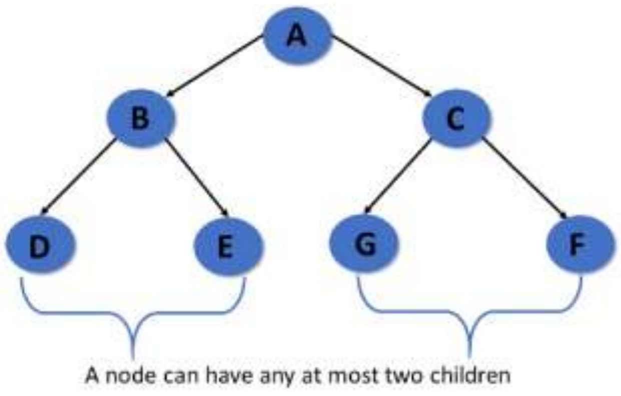


Perfect binary tree

2.2.Properties of binary tree

A binary tree has the following properties:

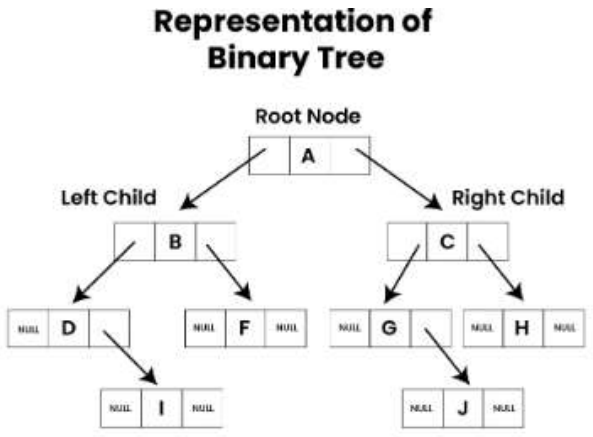
- Follows all properties of the tree data structure.
- Binary trees can have at most two child nodes.
- These two children are called the left child and the right child.
- A complete binary tree is said to be a proper binary tree where all leaves have the same depth.
- In a complete binary tree number of nodes at depth **d** is **2d**.
- In a complete binary tree with **n** nodes height of the tree is **log(n+1)**.
- All the levels **except the last level** are completely full.



2.3.Representation of Binary Tree

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child



3.Binary Search Tree

3.1. Definition

To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree.

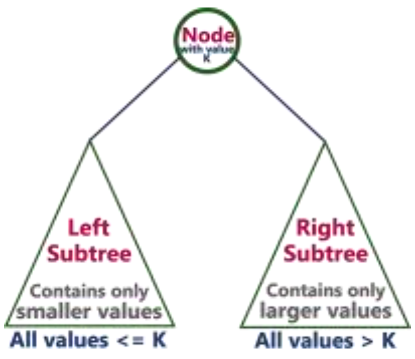
Binary search tree mainly focuses on the search operation in a binary tree.

Every binary search tree is a binary tree but every binary tree need not to be binary search tree**.**

Binary search tree can be defined as follows...

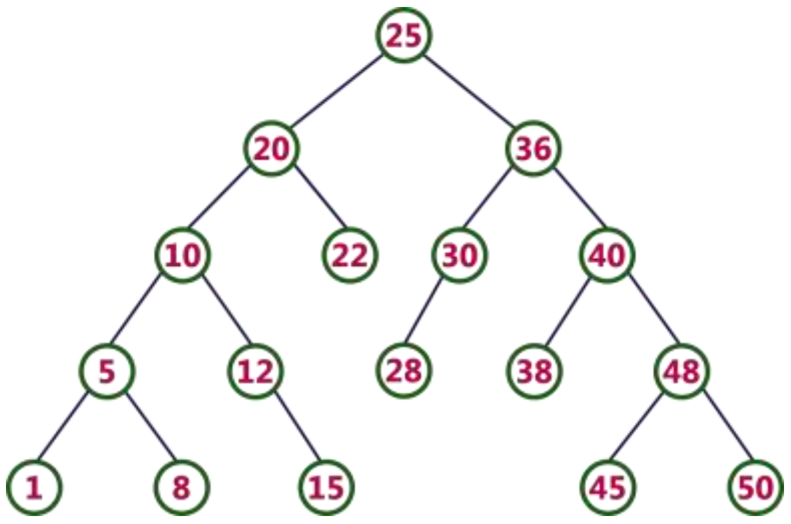
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

<u>3.2.OPERATONS ON A BINARY SEARCH TREE</u>

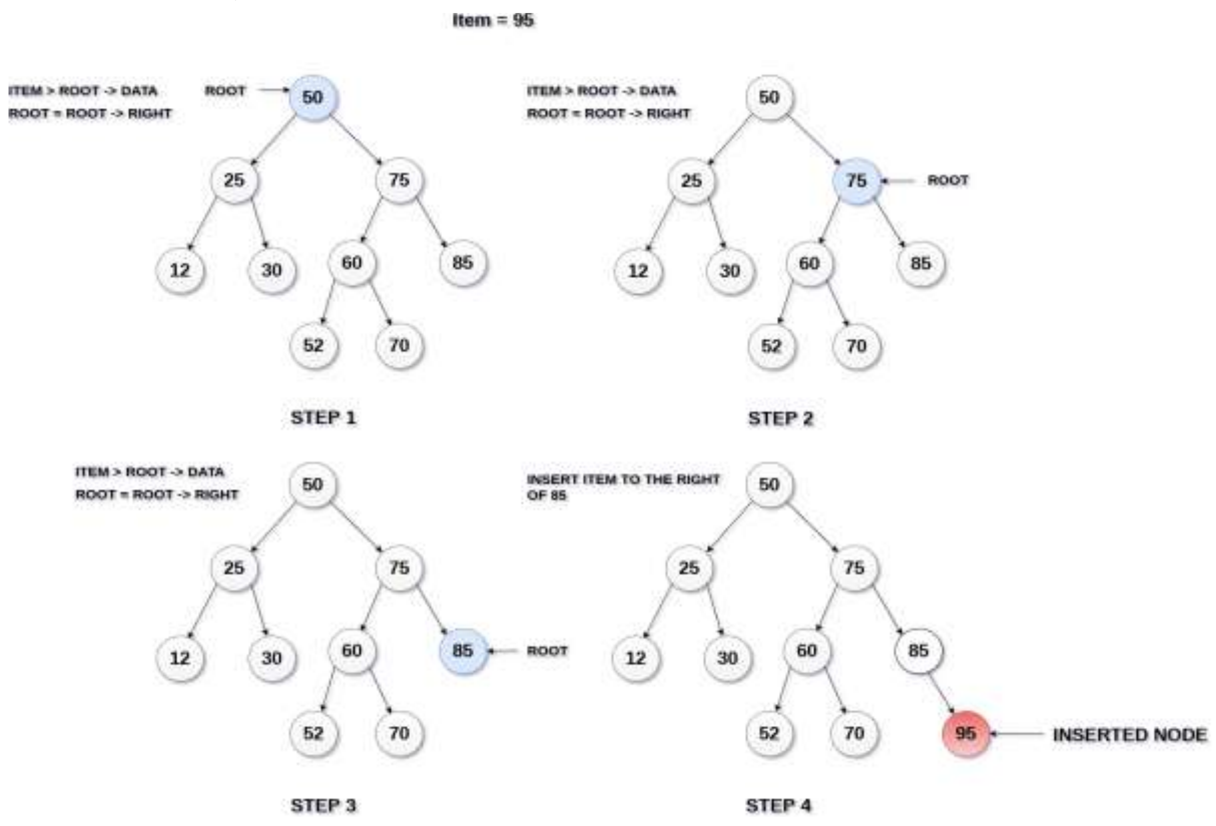
The following operations are performed on a binary search tree...

1. Insertion
2. Deletion
3. Traversal

<u>1.Insert Operation in BST</u>

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, thenrecursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of theroot.



Algorithm for Insert Operation

Insert (TREE, ITEM)

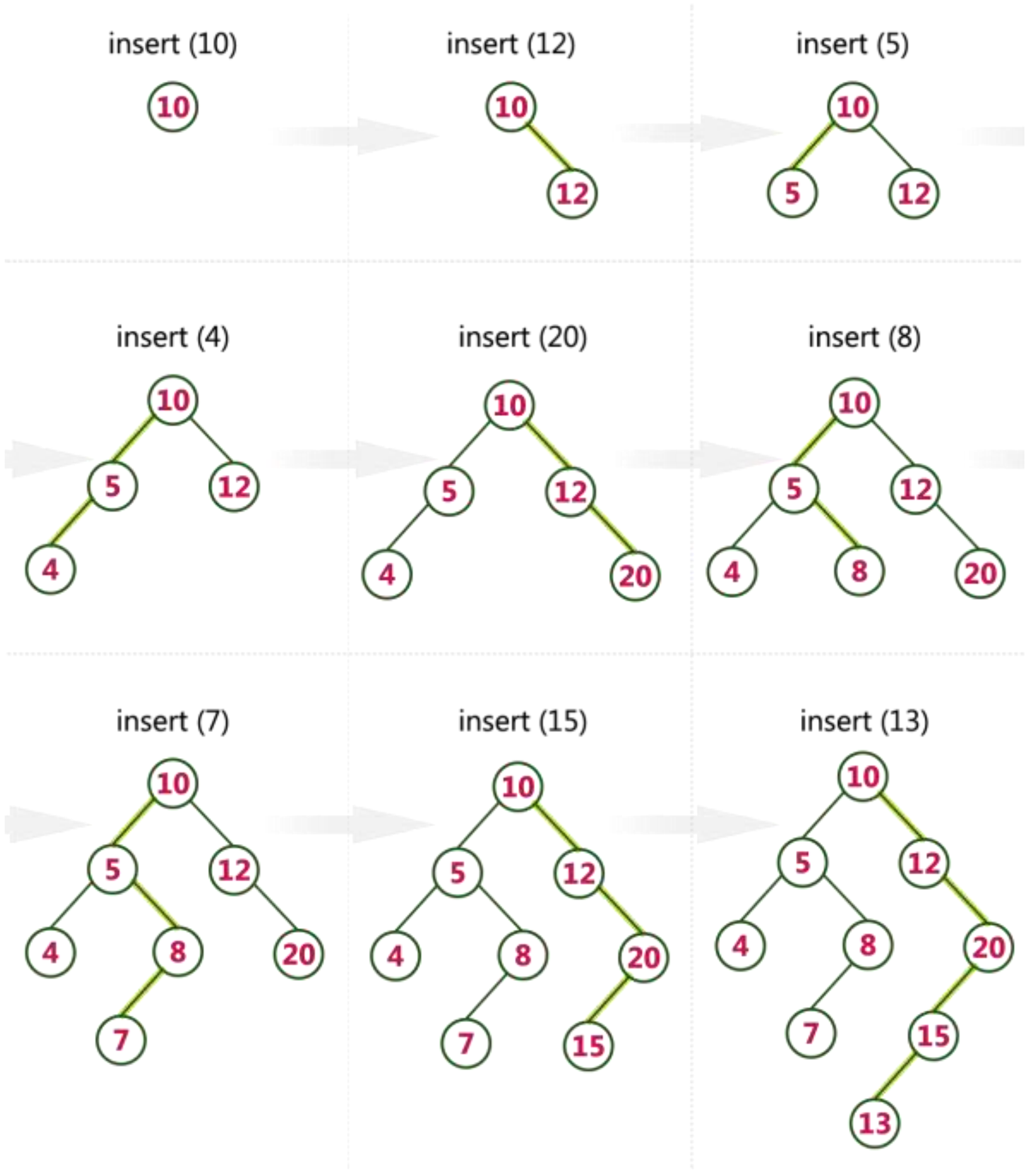
```
o   Step 1: IF TREE = NULL
                  Allocate memory for TREE
                  SET TREE -> DATA = ITEM
                  SET TREE -> LEFT = TREE -> RIGHT = NULL
            ELSE
                  IF ITEM < TREE -> DATA
                        Insert(TREE -> LEFT, ITEM)
                  ELSE
                        Insert(TREE -> RIGHT, ITEM)
                  [END OF IF][END OF IF]
o   Step 2: END
```

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...

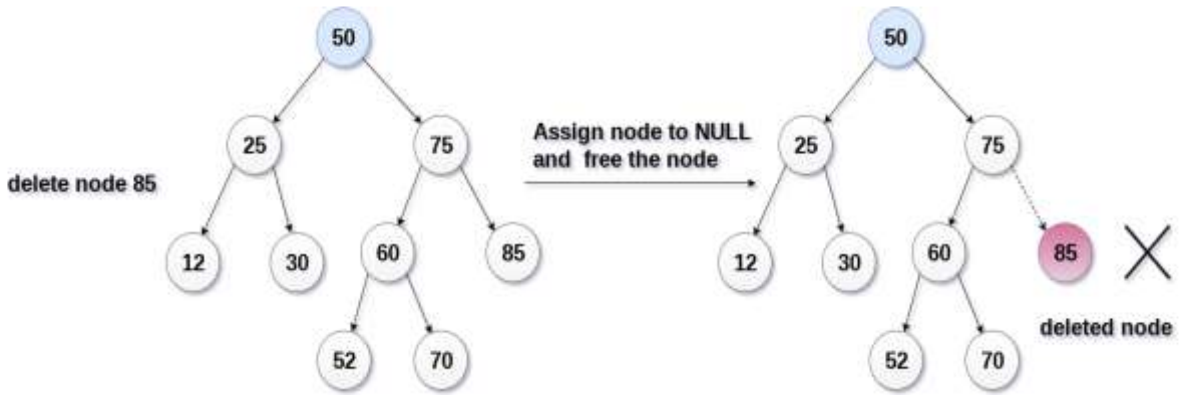


2.Delete Operation in BST

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.
There are **three situations of deleting a node** from binary search tree.

a) The node to be deleted is a leaf node

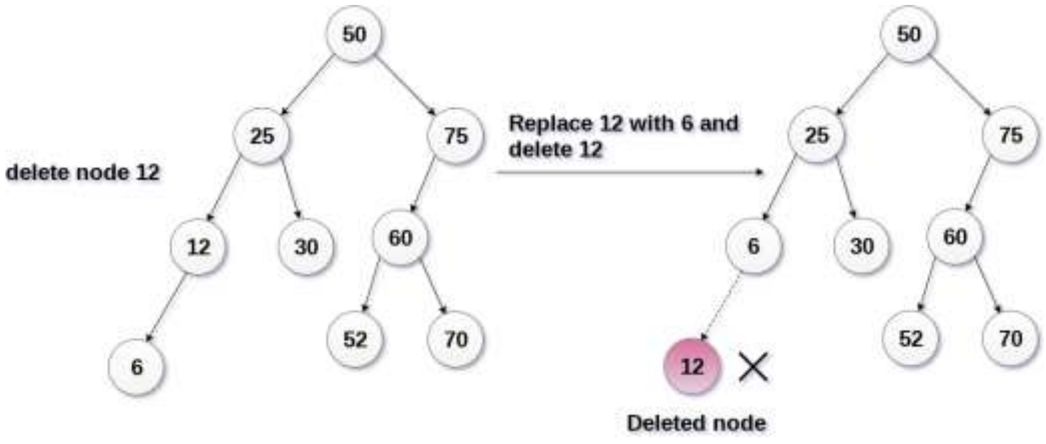
It is the simplest case, in this case, replace the leaf node with the NULL and simple free theallocated space.
In the following image, we are deleting the node 85, since the node is a leaf node, thereforethe node will be replaced with NULL and allocated space will be freed.



b) The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply bedeleted.

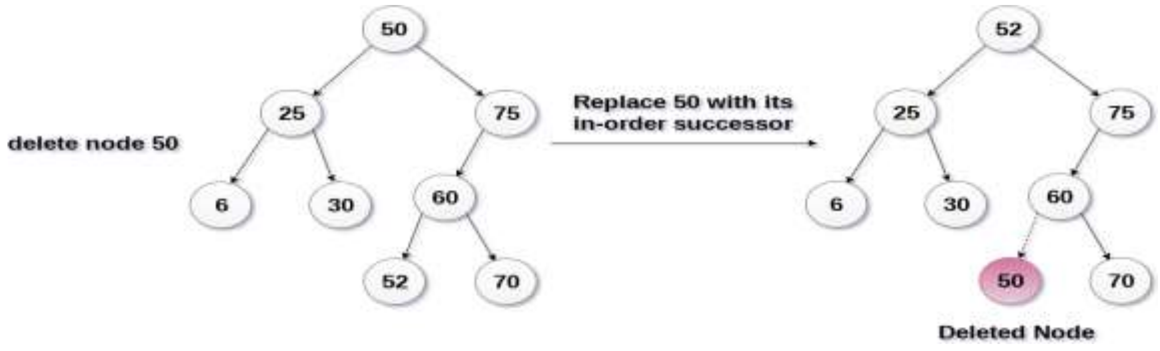


a)The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value(to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.
In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, whichwill simply be deleted.



Algorithm Delete (TREE, ITEM)

```
Step1: IFTREE=NULL
                Write "item not found in the tree"
        ELSE IF
             ITEM < TREE -> DATA
            Delete(TREE->LEFT,ITEM)
        ELSE IF ITEM>TREE->DATA Delete(TREE->RIGHT,ITEM)
        ELSE IF TREE->LEFT AND TREE->RIGHT
                SET TEMP = findLargestNode(TREE -> LEFT)
                SET TREE -> DATA = TEMP -> DATA Delete(TREE -> LEFT, TEMP -
                > DATA)
        ELSE
                SET TEMP = TREE
        IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
        SET TREE = NULL
        ELSE IF TREE -> LEFT != NULL
        SET TREE = TREE -> LEFT
        ELSE
                SET TREE = TREE -> RIGHT
        [END OF IF] FREE TEMP [END OF IF]
Step 2: END
```

3.Traversal operation of BST

Traversal of the tree in data structures is a process of visiting each node and prints their value. There are three ways to traverse tree data structure.

- Pre-order Traversal
- In-Order Traversal
- Post-order Traversal

Pre-Order Traversal

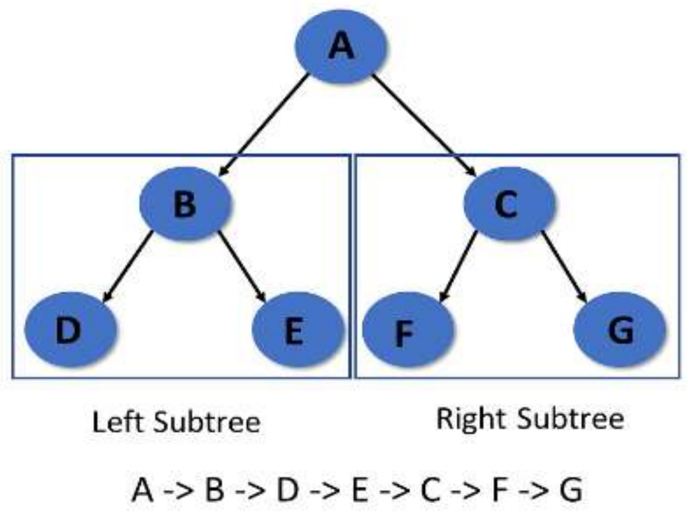
In pre-order traversal, it visits the root node first, then the left subtree, and lastly right subtree.

Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree



In-Order Traversal

In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

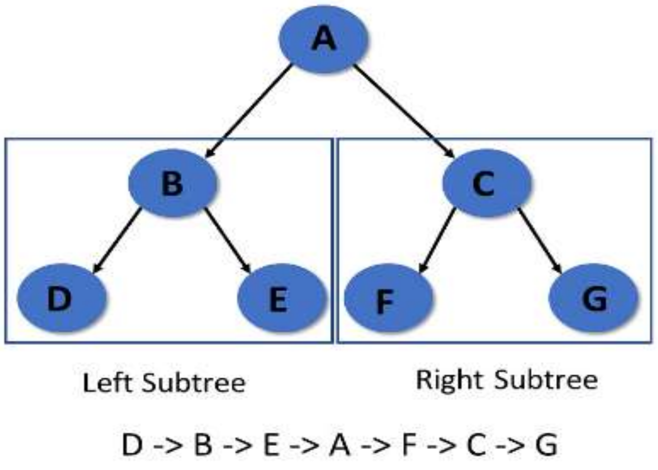
Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree

Example:



<u>Post-Order Traversal</u>

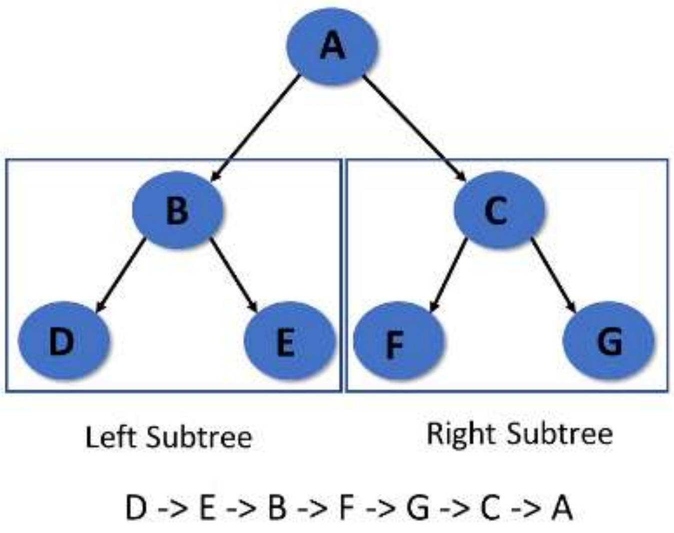
It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



What is Height Balanced Search Tree

A height-balanced binary tree is defined as a binary tree in which the height of the left and the right subtree of any node differ by not more than 1. AVL tree, red-black tree are examples of height-balanced trees.

Conditions for Height-Balanced Binary Tree:

Following are the conditions for a height-balanced binary tree:

- The difference between the heights of the left and the right subtree for any node is not more than one.
- The left subtree is balanced.
- The right subtree is balanced.

4.AVL TREES

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.

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 -1, 0 or +1.

In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1.

In an AVL tree, every node maintains an extra information known as **balance factor**.

The AVL tree was introduced in the year 1962 by G.M.Adelson-Velsky andE.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node.

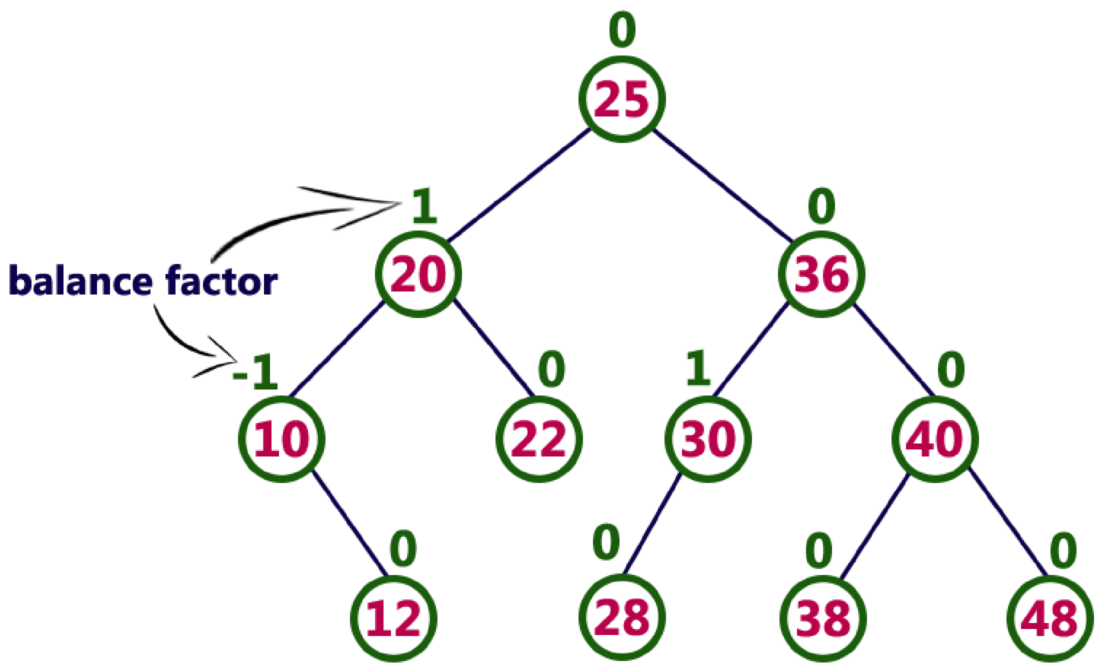
The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**.

In the following explanation, we calculate as follows...

Balance factor = height Of LeftSubtree – height Of RightSubtree

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

Example of AVL Tree



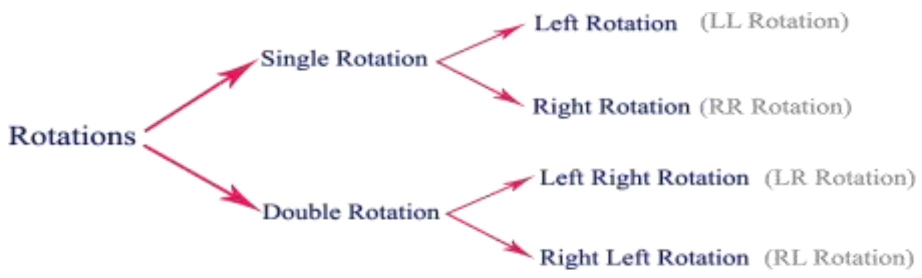
The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

4.1.AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.
Rotation operations are used to make the tree balanced.

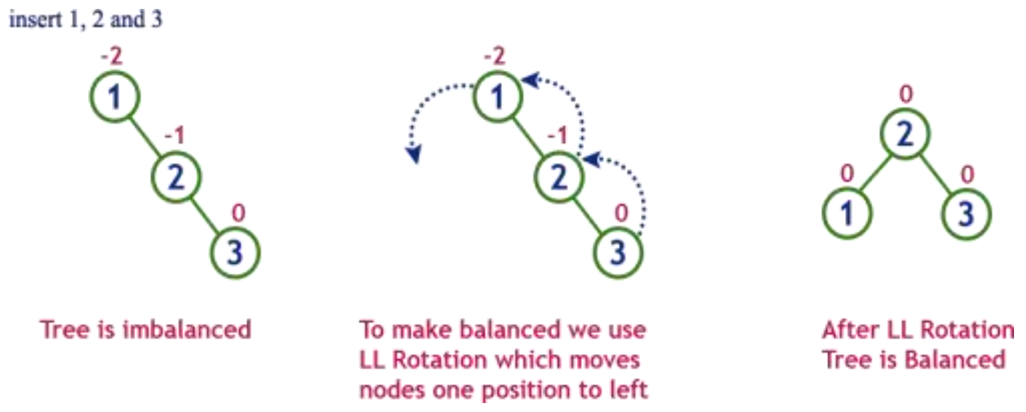
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

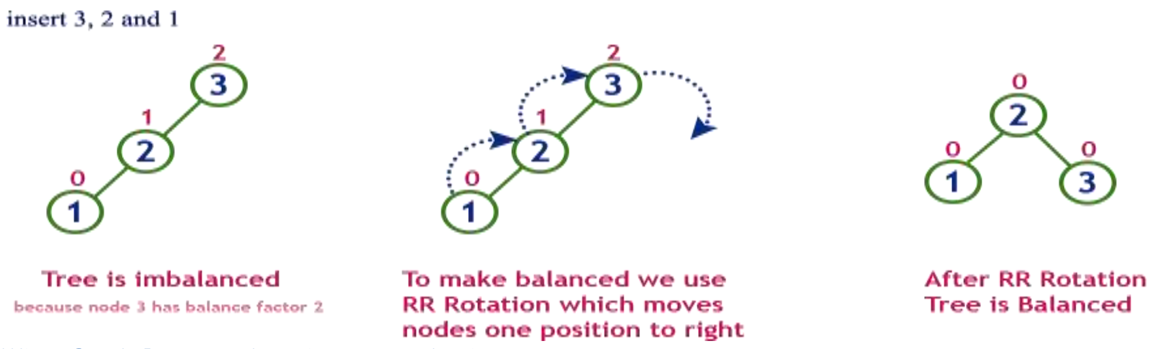
i) Single Left Rotation (LL Rotation)

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
-2
1
-1
2
0
3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

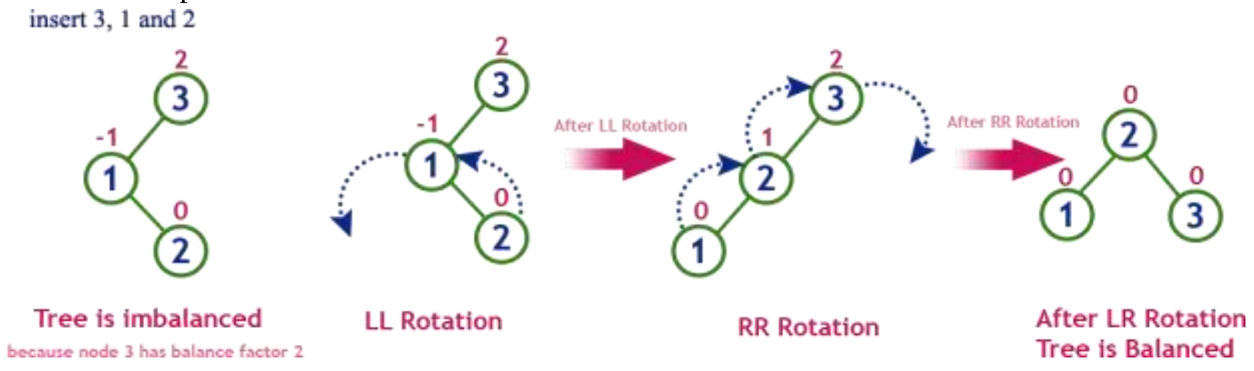
ii) Single Right Rotation (RR Rotation)

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
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

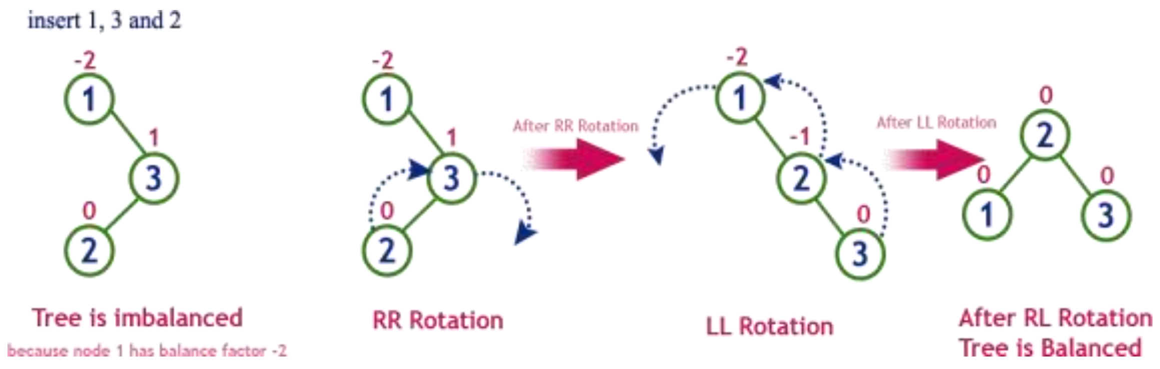
iii) Left Right Rotation (LR Rotation)

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...



iv)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 operationin AVL Tree...



4.2.Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

1.Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then continue the search process in right subtree.

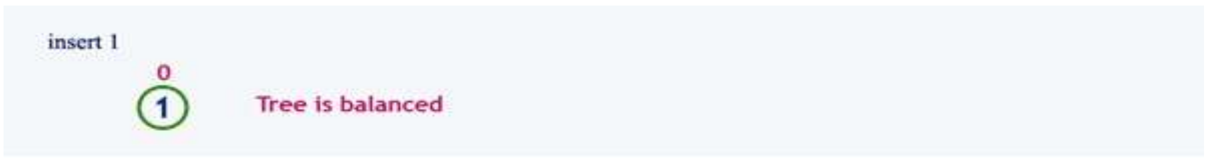
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

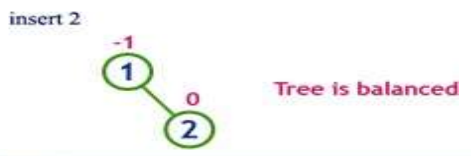
2.Insertion Operation in AVL Tree

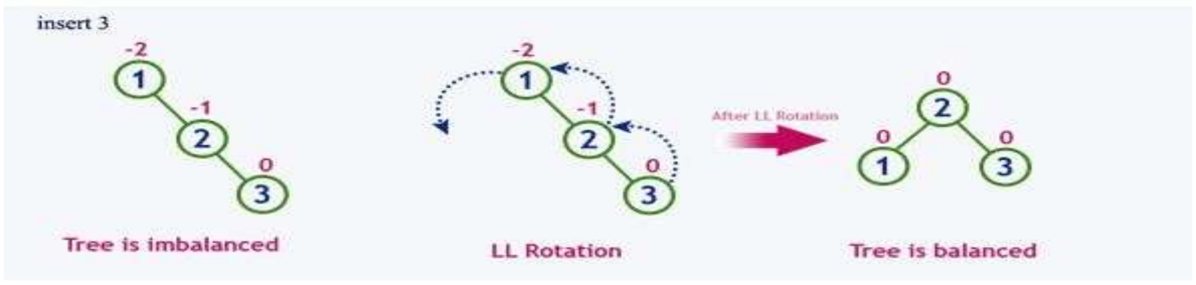
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

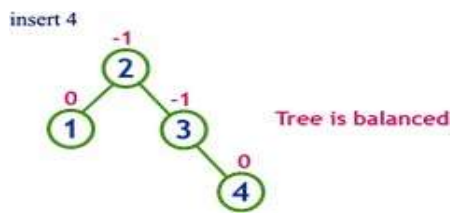
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation

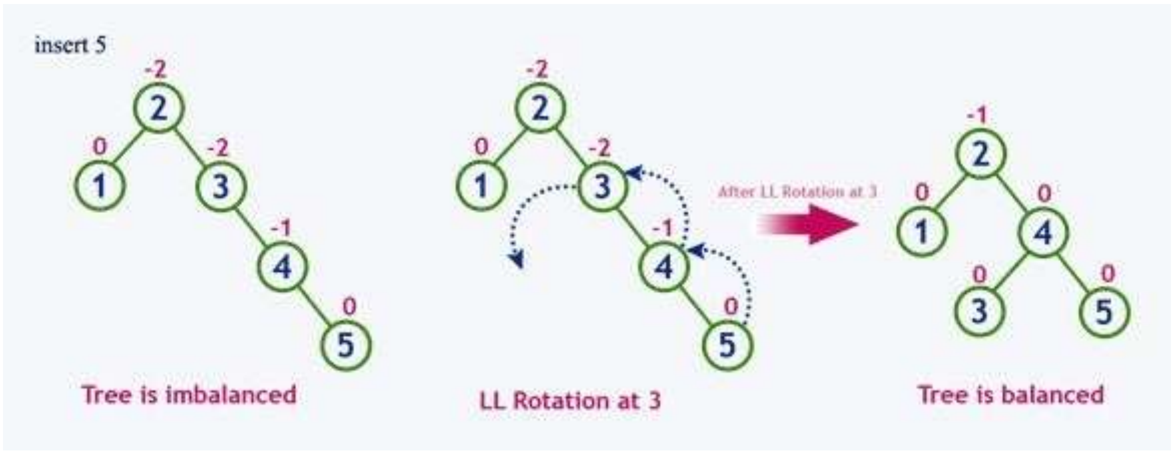
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

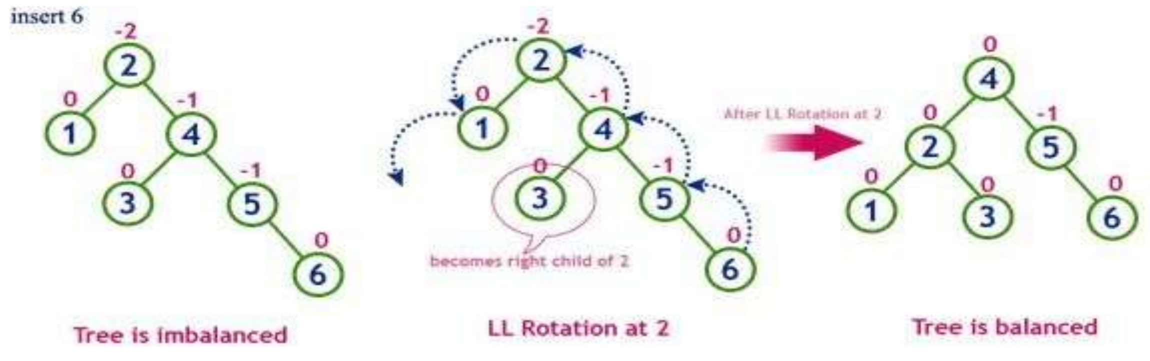


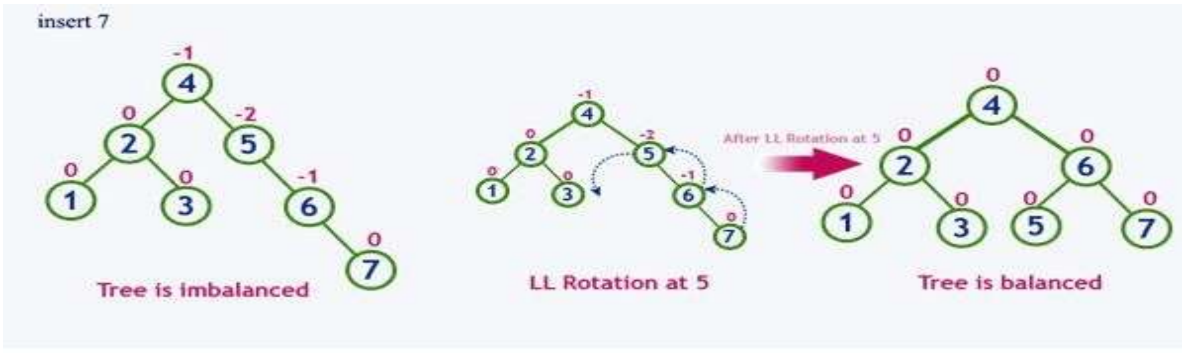


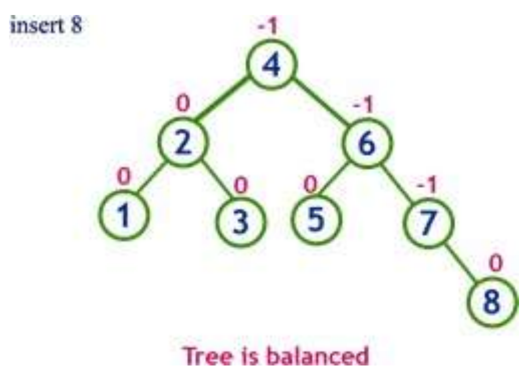





insert 5
After LL Rotation at 3
Tree is imbalanced
LL Rotation at 3
Tree is balanced


insert 6
becomes right child of 2
After LL Rotation at 2
Tree is imbalanced
LL Rotation at 2
Tree is balanced


insert 7
After LL Rotation at 5
Tree is imbalanced
LL Rotation at 5
Tree is balanced


insert 8
Tree is balanced

3.Deletion Operation in AVL Tree

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 condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

The two types of rotations are **L rotation** and **R rotation.** Here, we will discuss R rotations.L rotations are the mirror images of them.

If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub- tree of the critical node, the R rotation will be applied.

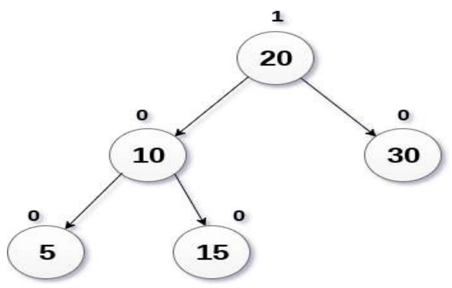
Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

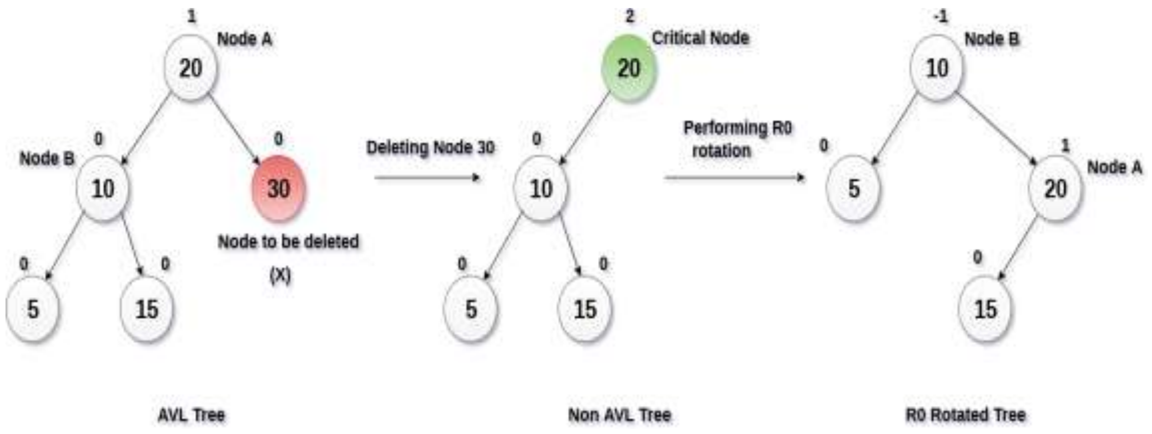
a) R0 rotation (Node B has balance factor 0)

If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

Example:

Delete the node 30 from the AVL tree shown in the following image.



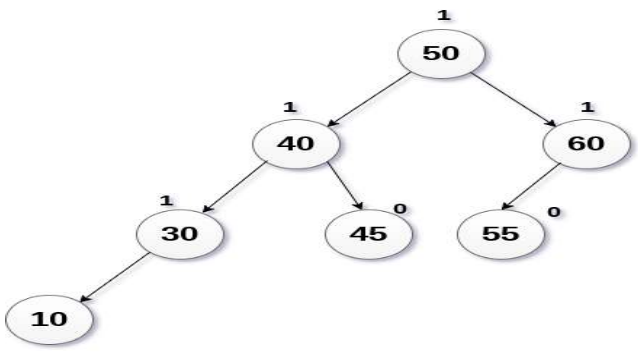


b)R1 Rotation (Node B has balance factor 1)

R1 Rotation is to be performed if the balance factor of Node B is 1.

Example

Delete Node 55 from the AVL tree shown in the following image.

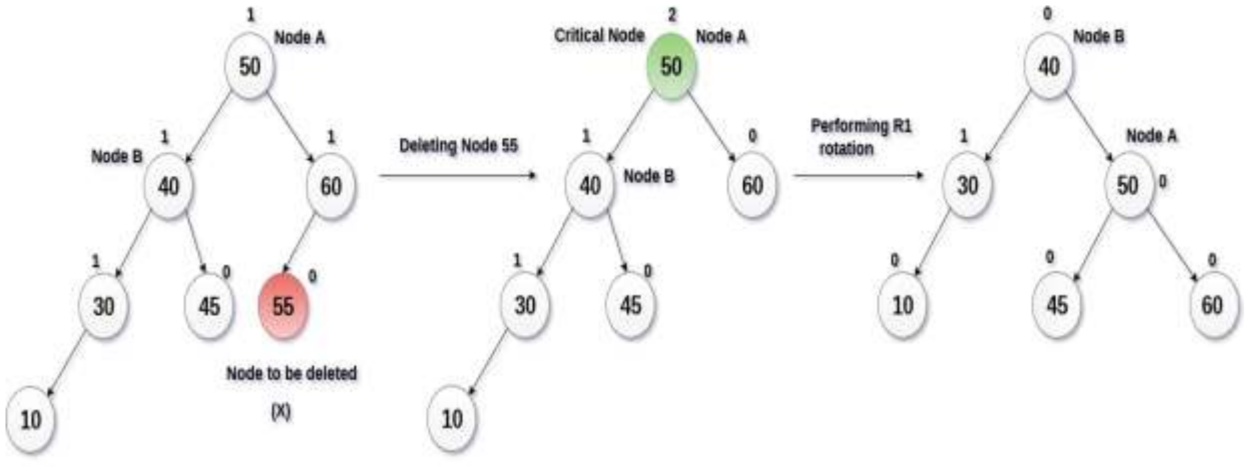


AVL Tree

Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A(i.e. 45).

The process involved in the solution is shown in the following image.



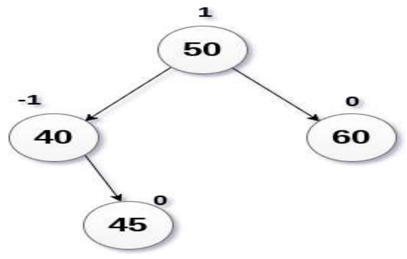
AVL Tree Non AVL Tree R1 Rotated Tree

c)R-1 Rotation (Node B has balance factor -1)

R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation.

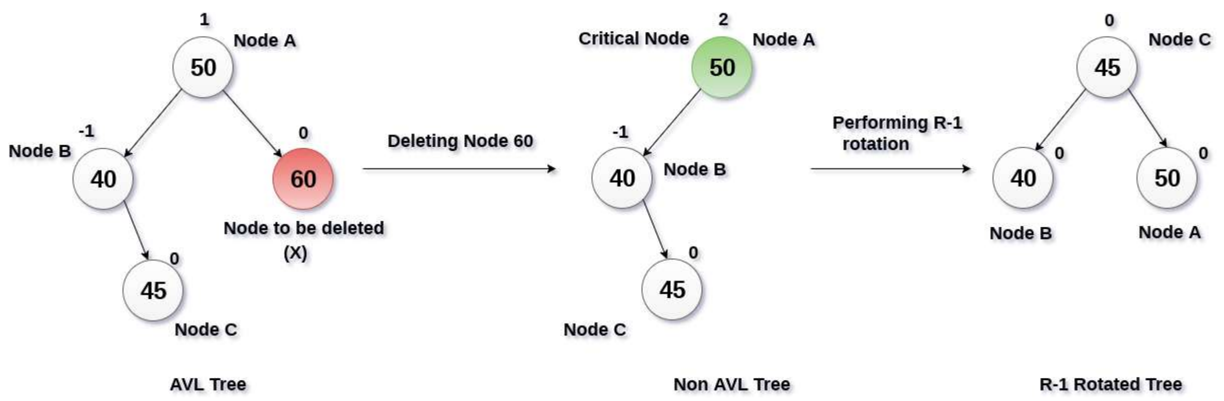
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



<u>Applications of A VL Trees</u>

AVL trees are applied in th e following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearl y sorted
- AVL trees are mostly used for **in-memory sorts of sets and dictionaries**.
- AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from the database applications
- AVL tree is a balanced binary search tree which employees rotation to maintain balance.
- It has application in story-line games as well.

UNIT-V-II

1.Brief introduction to hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values.

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

Hashing is the process of transforming any given key or a string of characters into another value.

This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

The most popular use for hashing is the implementation of hash tables.

A hash table stores key and value pairs in a list that is accessible through its index.

Because key and value pairs are unlimited, the hash function will map the keys to the table size. A hash value then becomes the index for a specific element.

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

2.Brief introduction to hash functions

It is a function, which distributes the keys evenly among the cells in the Hash Table. Using the same hash function

we can retrieve data from the hash table. Hash function is used to implement hash table. The integer value returned by the hash function is called hash key. If the input keys are integer, the commonly used hash function is

H (key) = key % Tablesize

A Hash Function is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table.

A hash function is any function that can be used to map data of arbitrary size to fixed-size values, though there are some hash functions that support variable length output. The values returned by a hash function are called hash values, hash codes, hash digests, digests, or simply hashes.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a “function” that maps keys to values.

The key for a given object can be calculated using a function called a hash function.

Characteristics of a Good Hash Function

There are four main characteristics of a good hash function:

1) The hash value is fully determined by the data being hashed.
2) The hash function uses all the input data.
3) The hash function "uniformly" distributes the data across the entire set of possible hash values.
4) The hash function generates very different hash values for similar strings. Let's examine why each of these is important:

Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

Rule 2: If the hash function doesn't use all the inp5u5t data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4: In real world applications, many data sets contain very similar data elements.

Types of Hash Functions

1. Division Method

2. Mid Square Method

3. Multiplicative Hash Function

4. Digit Folding

1. **Division Method**:

It depends on remainder of division. Divisor is Table Size.

Formula is

(H (key) = key % table size)

Example:

key= 12345
table size= 95
h(12345) = 12345 % 95
= 90

key = 1276
table size = 11
h(1276) = 1276 % 11
= 0

2.**Mid Square Method:**

We first square the item, and then extract some portion of the resulting digits.

For example, if the item were 44, we would first compute 442=1,936. Extract the middle two digit 93 from the answer. Store the key 44 in the index 93

3.**Multiplicative Hash Function**:

Key is multiplied by some constant value.

Hash function is given by, H(key)=Floor (P * (key * A))

P = Integer constant [e.g. P=50]

A = Constant real number [A=0.61803398987],suggested by Donald Knuth to use this constant

E.g. Key 107 H(107)=Floor(50*(107*0.61803398987)) =Floor(3306.481845)

H(107)=3306 Consider table size is 5000

4.Digit Folding Method:

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash key value.

What is collision?

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied slot in the hash table is called collision,

Or

If two more keys hashes to the same index, the corresponding records cannot be stored in the same location. This condition is known as collision.

Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

3.Collision resolution techniques:

There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

3.1.Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

In collision by chaining the hash table itself is restructured where a separate list of all elements that hash to the same value is maintained.

In other words chaining sets up lists of items with the same index. The array elements are pointers to the first nodes of the lists.

When a new item hashes to a slot in the table it is inserted to the front of the list.

In this method a linked list is built for each bucket, and a linear search is conducted within each list to find an element that hashes to the index of the array that points to that particular list (Figure 18.3). This method is simple and widely used.

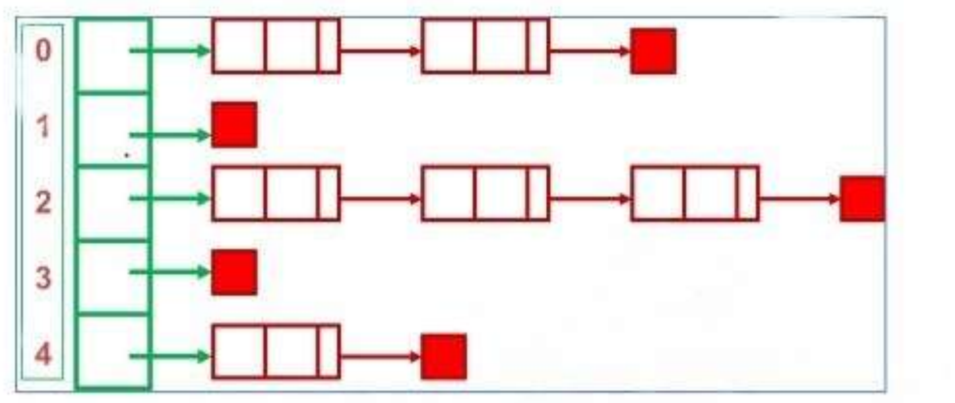
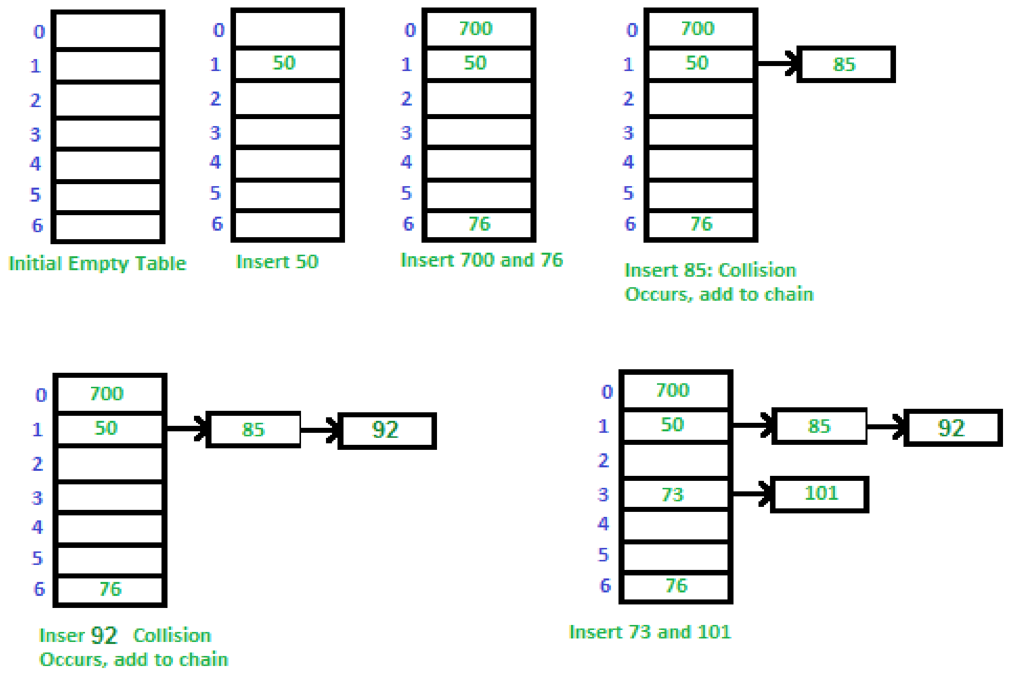


Figure 18.3 Collision Resolution with Chaining

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, add to chain
Inser 92 Collision Occurs, add to chain
Insert 73 and 101
700
50
76
85
92
73
101

3.2Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing.

Different ways of Open Addressing:

3.2.1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing

Let **hash(x)** be the slot index computed using a hash function and **S** be the table size
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

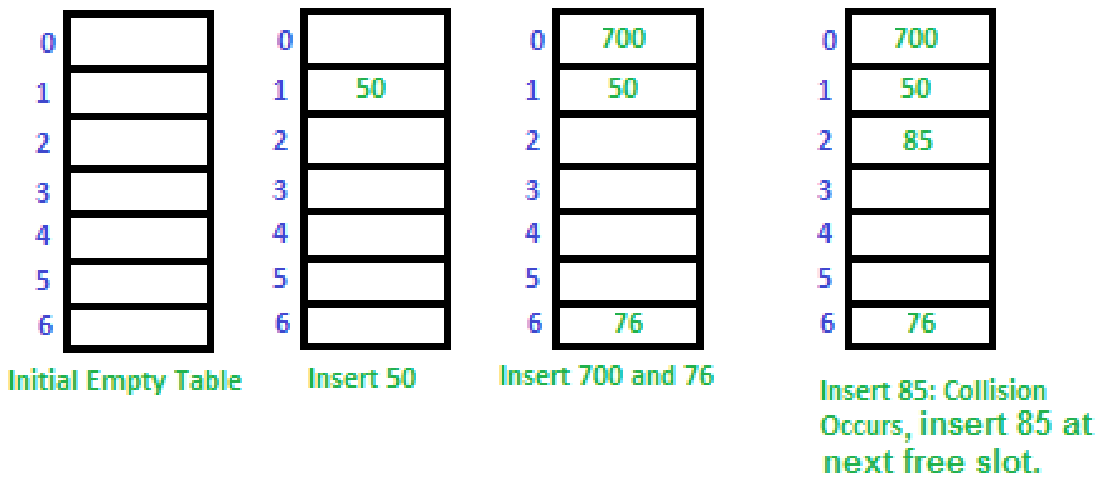
Example:1

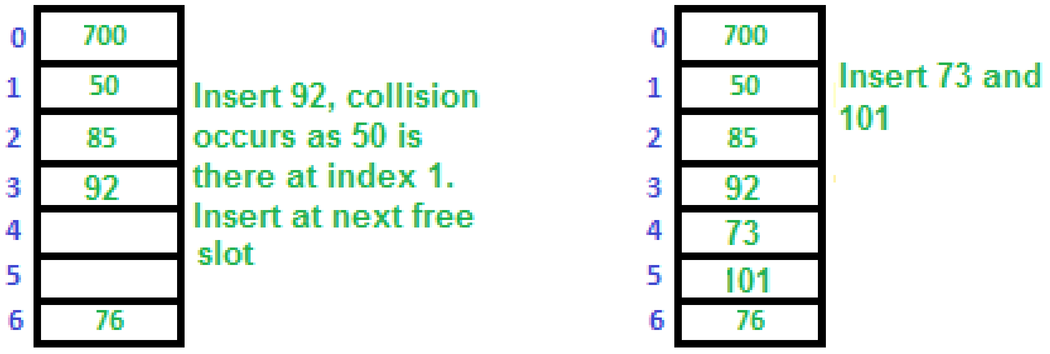
Sr.No.	**Key**	**Hash**	**Array Inex**	**fter Linear Probing, Array Index**
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Example:2

Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table,although it is fixed internally with a pre-defined formula.




0 1 2 3 4 5 6
700 50 85 92 76
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
700 50 85 92 73 101 76
Insert 73 and 101

3.2.2Quadratic Probing

Quadratic probing eliminates the problem of "Primary Clustering" that occurs in Linear probing techniques.

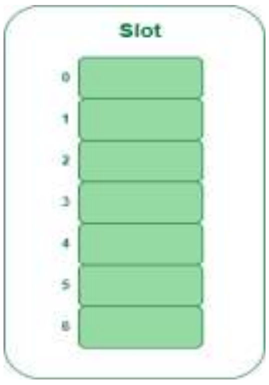
The working of quadratic probing involves taking the initial hash value and probing in the hash table by adding successive values of an arbitrary quadratic polynomial.

let hash(x) be the slot index computed using hash function.

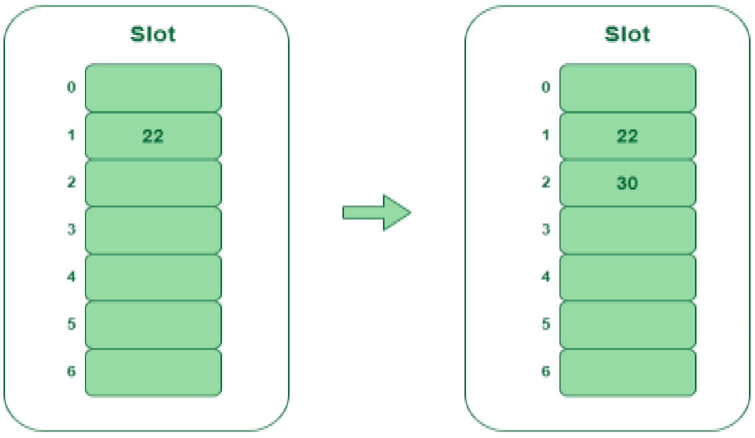
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50.

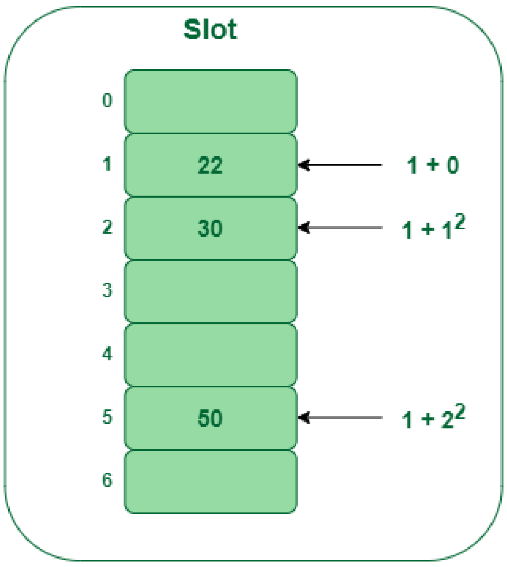
- **Step 1:** Create a table of size 7.



- **Step 2 – Insert 22 and 30**
    - Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
    - Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



- **Step 3:** Inserting 50
    - Hash(50) = 50 % 7 = 1
    - In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
    - Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
    - Now, cell 5 is not occupied so we will place 50 in slot 5.



3.2.3 Double Hashing

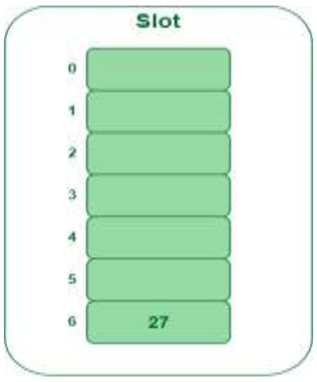
Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i**th rotation.

let hash(x) be the slot index computed using hash function.

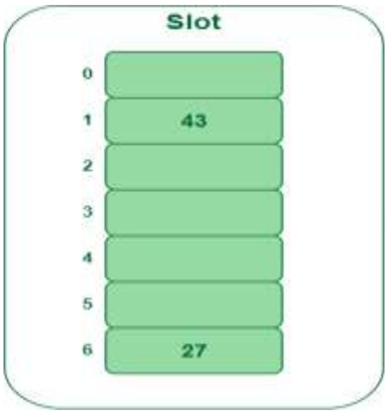
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
  - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



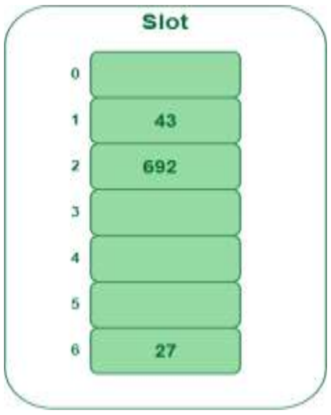
- **Step 2:** Insert 43
  - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



- **Step 3:** Insert 692
  - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
  - So we need to resolve this collision using double hashing.

    h_{new} = [h1(692) + i * (h2(692)] % 7
    = [6 + 1 * (1 + 692 % 5)] % 7
    = 9% 7
    = 2

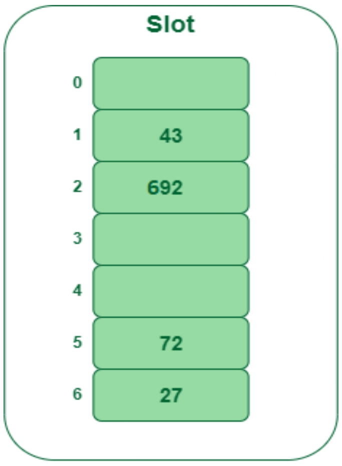
Now, as 2 is an empty slot, so we can insert 692 into 2nd slot.



- **Step 4:** Insert 72
  - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
  - So we need to resolve this collision using double hashing.

    h_{new} = [h1(72) + i * (h2(72)] % 7
    = [2 + 1 * (1 + 72 % 5)] % 7
    = 5 % 7
    = 5,

    Now, as 5 is an empty slot,
    so we can insert 72 into 5th slot.



S.No.	**Separate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hashfunction or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency andnumber of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash tablein chaining are never used).	In Open addressing, a slot can be used even if aninput doesn’t map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

4.Hash tables: basic implementation and operations

4.1Hash table: basic implementation

Hash table is a data structure in which keys are mapped to array positions by a hash function.

- A hash table is a data structure that is used to store and retrieve data (keys) very quickly.
- It is an array of some fixed size, containing the keys.
- Hash table run from 0 to Tablesize – 1.
- Each key is mapped into some number in the range 0 to Tablesize – 1.
- This mapping is called Hash function.
- Insertion of the data in the hash table is based on the key value obtained from the hash function.
- Using same hash key value, the data can be retrieved from the hash table by few or more Hash key comparison.
- The load factor of a hash table is calculated using the formula:

(Number of data elements in the hash table) / (Size of the hash table)

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.

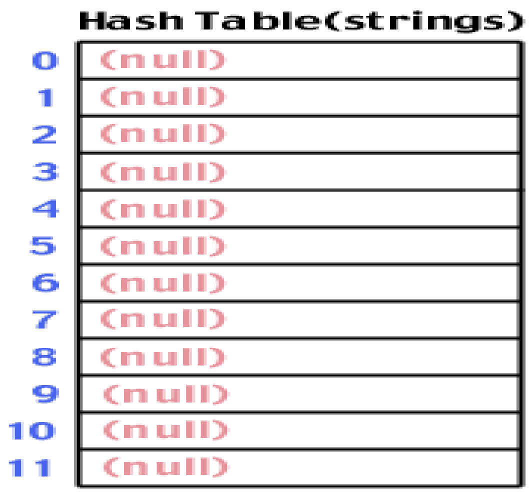
By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function.

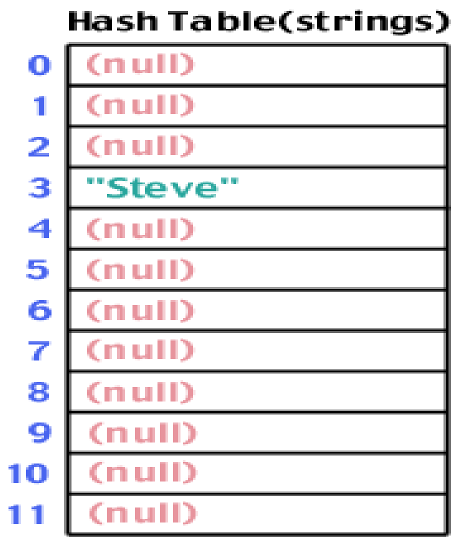
The hash function is a mapping from the input space to the integer space that defines the indices of the array.

In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

Let's take a simple example. First, we start with a hash table array of strings (we'll use strings as the databeing stored and searched in this example). Let's say the hash table size is 12:



We run "Steve" through the hash function, and find that hash("Steve",12) yields 3:



4.2.Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table. DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
Struct  DataItem

{

int data;

int key;

};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item. int hashCode(int key)

```
{

return key % SIZE;

}
```

<u>Search Operation</u>

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

<u>Insert Operation</u>

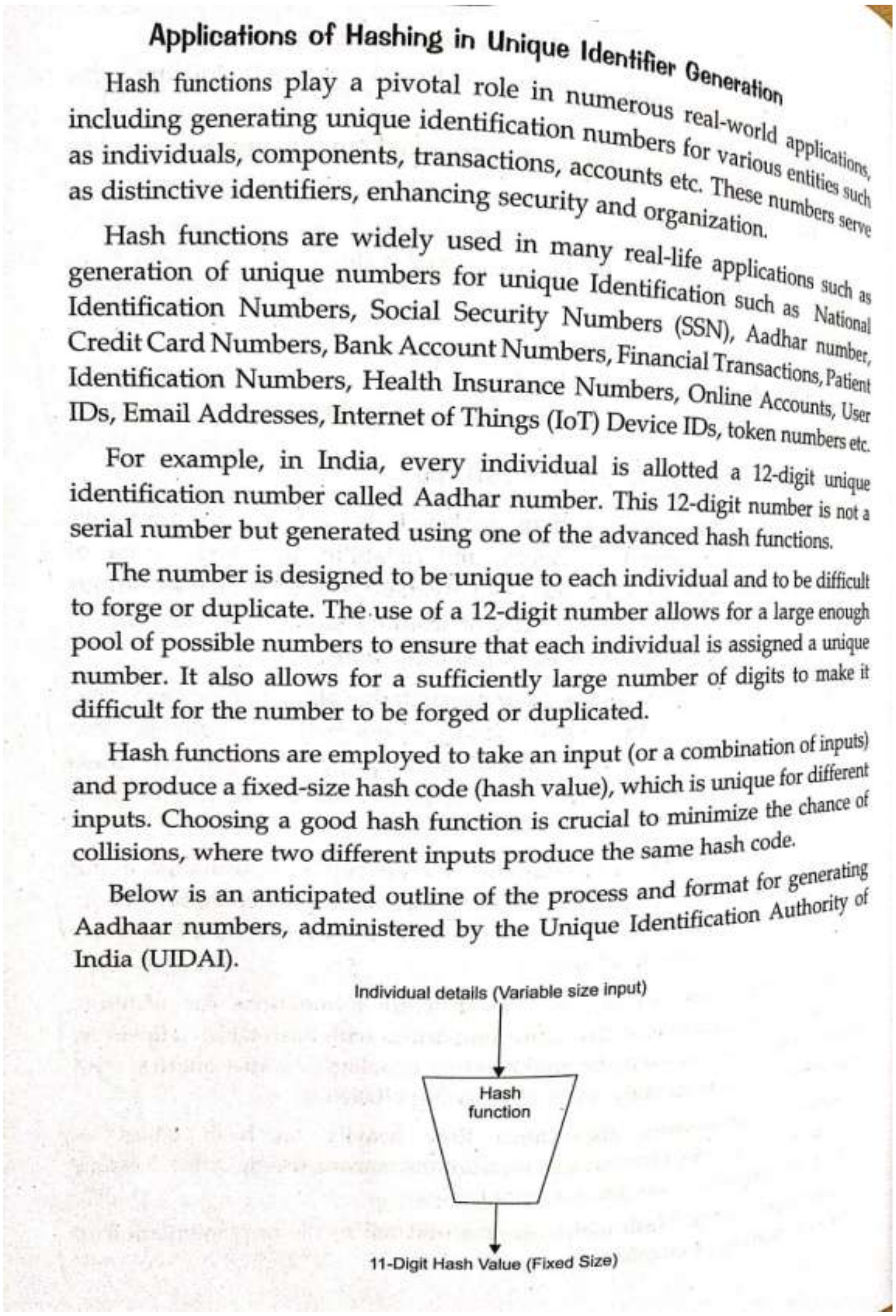
Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

<u>Delete Operation</u>

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

5. Applications of hashing

5.1.Applications of hashing in unique identifier generation



Applications of Hashing in Unique Identifier Generation

Hash functions play a pivotal role in numerous real-world applications, including generating unique identification numbers for various entities such as individuals, components, transactions, accounts etc. These numbers serve as distinctive identifiers, enhancing security and organization.

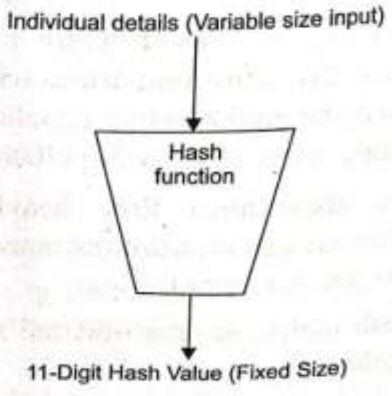
Hash functions are widely used in many real-life applications such as generation of unique numbers for unique Identification such as National Identification Numbers, Social Security Numbers (SSN), Aadhar number, Credit Card Numbers, Bank Account Numbers, Financial Transactions, Patient Identification Numbers, Health Insurance Numbers, Online Accounts, User IDs, Email Addresses, Internet of Things (IoT) Device IDs, token numbers etc.

For example, in India, every individual is allotted a 12-digit unique identification number called Aadhar number. This 12-digit number is not a serial number but generated using one of the advanced hash functions.

The number is designed to be unique to each individual and to be difficult to forge or duplicate. The use of a 12-digit number allows for a large enough pool of possible numbers to ensure that each individual is assigned a unique number. It also allows for a sufficiently large number of digits to make it difficult for the number to be forged or duplicated.

Hash functions are employed to take an input (or a combination of inputs) and produce a fixed-size hash code (hash value), which is unique for different inputs. Choosing a good hash function is crucial to minimize the chance of collisions, where two different inputs produce the same hash code.

Below is an anticipated outline of the process and format for generating Aadhaar numbers, administered by the Unique Identification Authority of India (UIDAI).


Individual details (Variable size input)
Hash function
11-Digit Hash Value (Fixed Size)

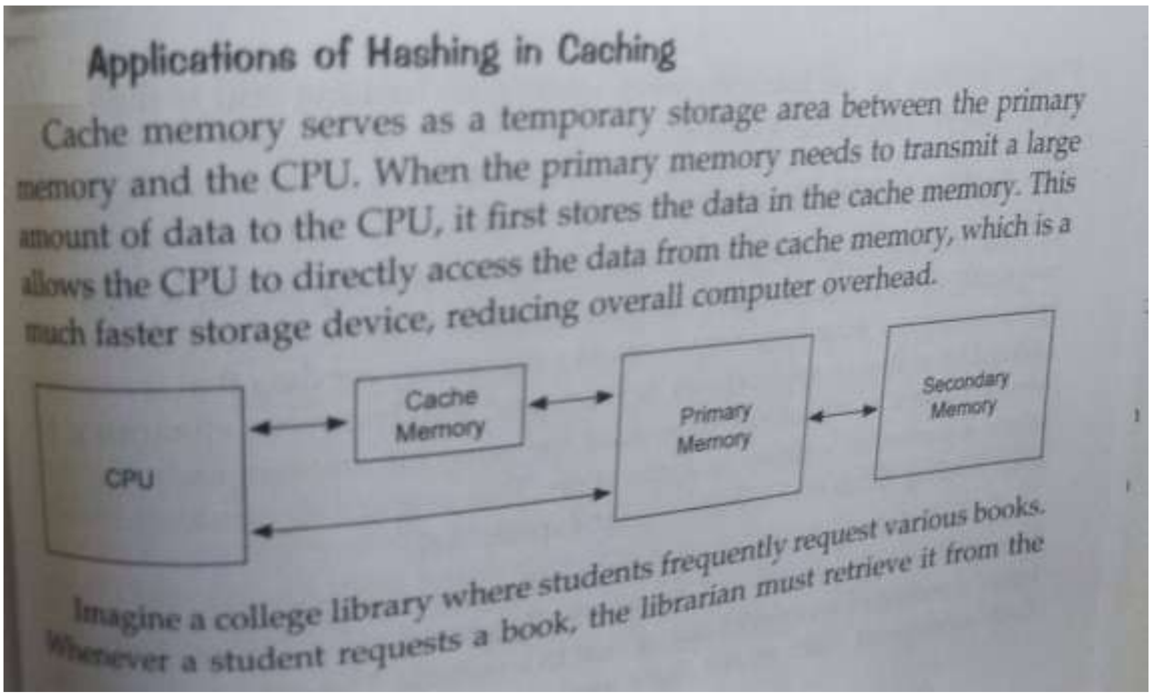
Aadhaar Number Generation Process:

The Aadhaar number generation process is a secure and complex procedure that involves multiple steps to ensure the uniqueness and validity of each Aadhaar number. Here's a simplified overview of the process:

1. **Enrolment:** An individual visits an authorized Aadhaar Enrolment Centre (AEC) and submits demographic and biometric information, along with proof of identity and address documents.
2. **Data Capture:** The AEC captures the individual's demographic data, including name, date of birth, address, and gender. They also capture biometric data, including fingerprints and iris scans.
3. **Data Upload:** The captured data is securely uploaded to the Central Identities Data Repository (CIDR) maintained by the Unique Identification Authority of India (UIDAI).
4. **De-duplication:** The UIDAI performs a de-duplication process to check if the individual's biometric data matches any existing Aadhaar numbers.
5. **Aadhaar Generation:** If no match is found, a unique 12-digit Aadhaar number is generated for the individual. The number is assigned based on a random number generator and a Verhoeff checksum to ensure its validity.
6. **Aadhaar Letter:** An Aadhaar Letter containing the Aadhaar number and demographic details is generated and sent to the individual's registered address.
7. **Aadhaar Card:** Alternatively, the individual can visit the AEC or download the e-Aadhaar app to generate their Aadhaar card, which is a physical or digital representation of their Aadhaar number and demographic details.

The entire process is designed to ensure that each Aadhaar number is unique, valid, and linked to the correct individual. The UIDAI also maintains strict data security measures to protect Aadhaar data.

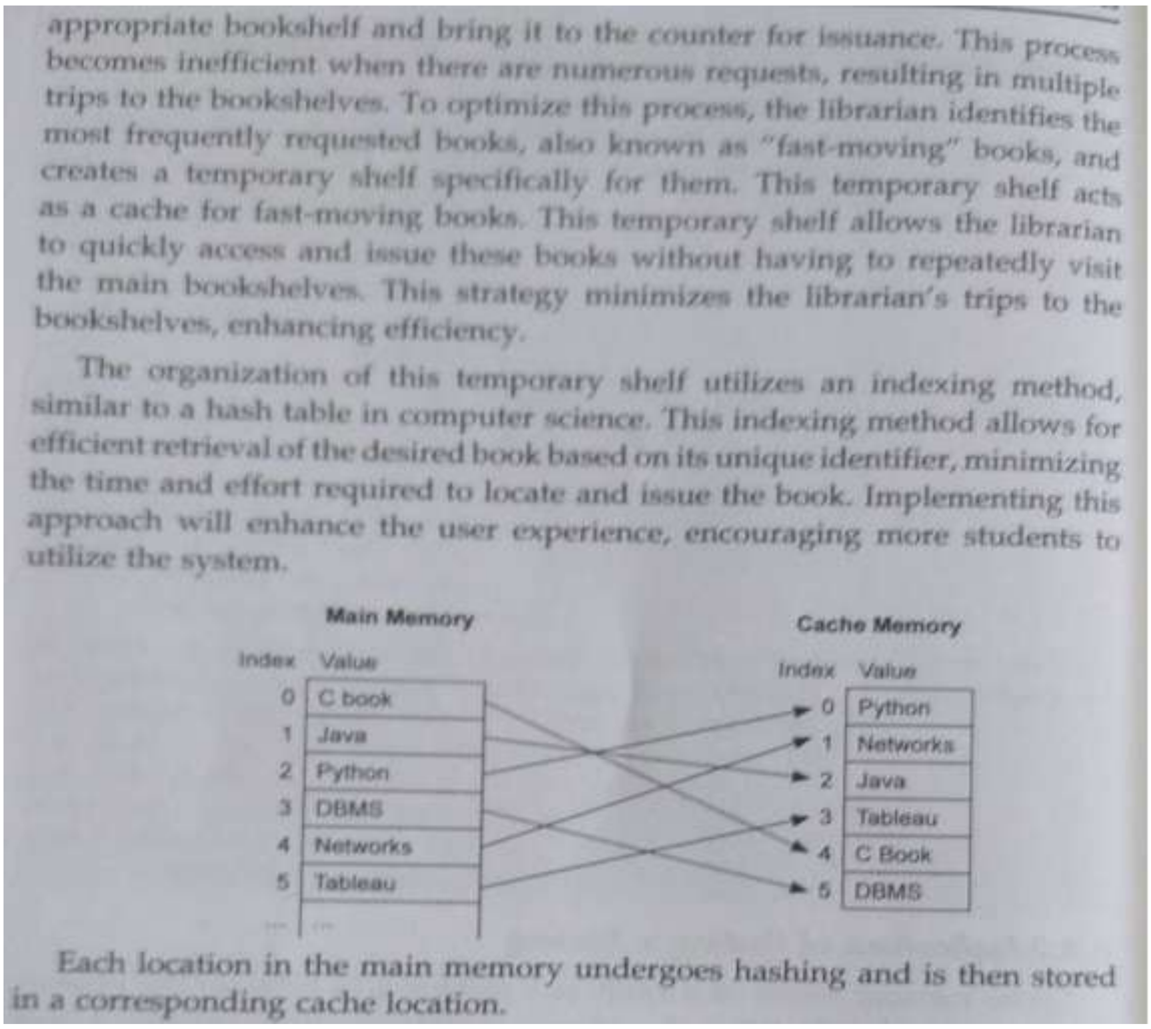
5.2.Applications of hashing in caching



Applications of Hashing in Caching

Cache memory serves as a temporary storage area between the primary memory and the CPU. When the primary memory needs to transmit a large amount of data to the CPU, it first stores the data in the cache memory. This allows the CPU to directly access the data from the cache memory, which is a much faster storage device, reducing overall computer overhead.

Imagine a college library where students frequently request various books. Whenever a student requests a book, the librarian must retrieve it from the



appropriate bookshelf and bring it to the counter for issuance. This process becomes inefficient when there are numerous requests, resulting in multiple trips to the bookshelves. To optimize this process, the librarian identifies the most frequently requested books, also known as "fast-moving" books, and creates a temporary shelf specifically for them. This temporary shelf acts as a cache for fast-moving books. This temporary shelf allows the librarian to quickly access and issue these books without having to repeatedly visit the main bookshelves. This strategy minimizes the librarian's trips to the bookshelves, enhancing efficiency.

The organization of this temporary shelf utilizes an indexing method, similar to a hash table in computer science. This indexing method allows for efficient retrieval of the desired book based on its unique identifier, minimizing the time and effort required to locate and issue the book. Implementing this approach will enhance the user experience, encouraging more students to utilize the system.

Main Memory

Index	Value
0	C book
1	Java
2	Python
3	DBMS
4	Networks
5	Tableau
...	...

Cache Memory

Index	Value
0	Python
1	Networks
2	Java
3	Tableau
4	C Book
5	DBMS

Each location in the main memory undergoes hashing and is then stored in a corresponding cache location.