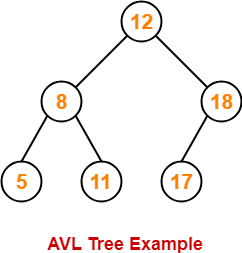
**[AVL Tree | AVL Tree Example | AVL Tree Rotation](https://www.gatevidyalay.com/avl-tree-avl-tree-example-avl-tree-rotation/)**

**AVL Tree-**

* Invented by[Georgy Adelson-Velsky](https://www.google.com/search?rlz=1C1CHBD_enGH889GH889&q=Georgy+Adelson-Velsky&stick=H4sIAAAAAAAAAONgVuLUz9U3MCkrLkhbxCrqnppflF6p4JiSmlOcn6cbBqSyKwHPv4p0JQAAAA&sa=X&ved=2ahUKEwjDlY-mmq_tAhUL1BoKHdDTA4wQmxMoATAnegQIHRAD) and [Evgenii Landis](https://www.google.com/search?rlz=1C1CHBD_enGH889GH889&q=Evgenii+Landis&stick=H4sIAAAAAAAAAONgVuLUz9U3MCkrLk5exMrnWpaempeZqeCTmJeSWQwAjOUiCx4AAAA&sa=X&ved=2ahUKEwjDlY-mmq_tAhUL1BoKHdDTA4wQmxMoAjAnegQIHRAE)
* AVL trees are special kind of binary search trees.
* In AVL trees, height of left subtree and right subtree of every node differs by at most one.
* AVL trees are also called as **self-balancing binary search trees**.

**Example-**

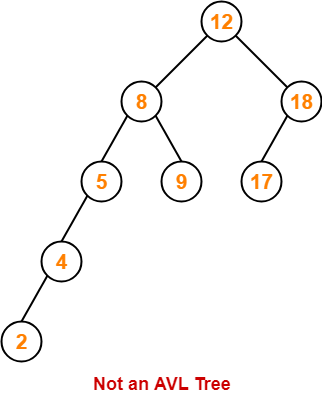
The following tree is an example of an AVL tree-



This tree is an AVL tree because-

* It is a binary search tree.
* The difference between the height of left subtree and right subtree of every node is at most one.

Following tree is not an example of AVL Tree-



This tree is not an AVL tree because-

* The difference between height of left subtree and right subtree of root node = 4 – 2 = 2.
* This difference is greater than one.

**Balance Factor-**

In AVL tree,

* Balance factor is defined for every node.
* Balance factor of a node = Height of its left subtree – Height of its right subtree

|  |
| --- |
| In AVL tree,  Balance factor of every node is either 0 or 1 or -1. |

**AVL Tree Operations-**

Like [**Binary Search T**](https://www.gatevidyalay.com/binary-search-tree-insertion-bst-deletion/)**ree**[**Operations**](https://www.gatevidyalay.com/binary-search-tree-insertion-bst-deletion/), commonly perform

operations on AVL tree are-

1. Search Operation
2. Insertion Operation
3. Deletion Operation

After performing any operation on AVL tree, the balance factor of each node is checked.

There are following two cases possible-

**Case-01:**

* After the operation, the balance factor of each node is either 0 or 1 or -1.
* In this case, the AVL tree is considered to be balanced.
* The operation is concluded.

**Case-02:**

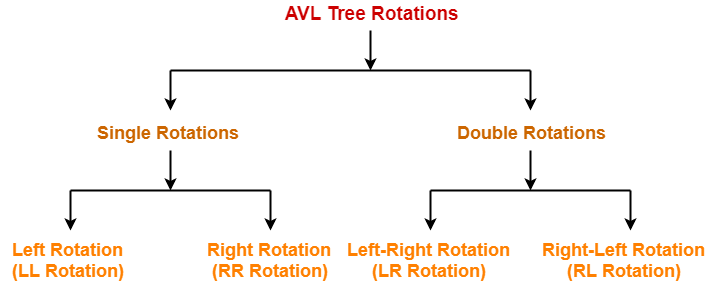
* After the operation, the balance factor of at least one node is not 0 or 1 or -1.
* In this case, the AVL tree is considered to be imbalanced.
* Rotations are then performed to balance the tree.

**AVL Tree Rotations-**

|  |
| --- |
| Rotation is the process of moving the nodes to make tree balanced. |

**Kinds of Rotations-**

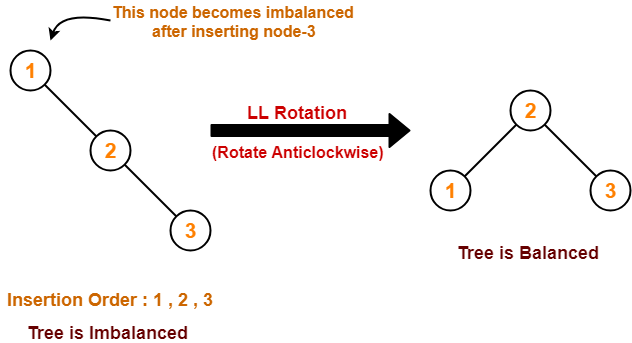
There are 4 kinds of rotations possible in AVL Trees-



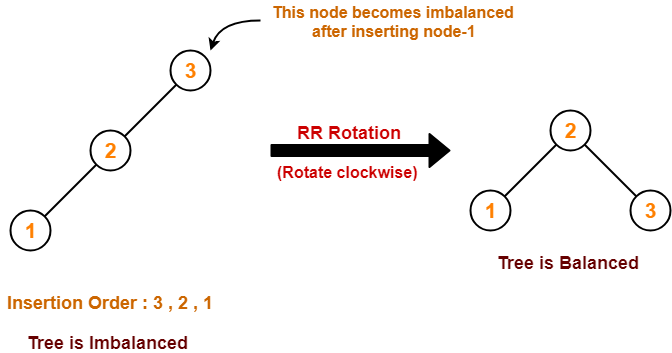
1. Left Rotation (LL Rotation)
2. Right Rotation (RR Rotation)
3. Left-Right Rotation (LR Rotation)
4. Right-Left Rotation (RL Rotation)

**Cases Of Imbalance And Their Balancing Using Rotation Operations-**

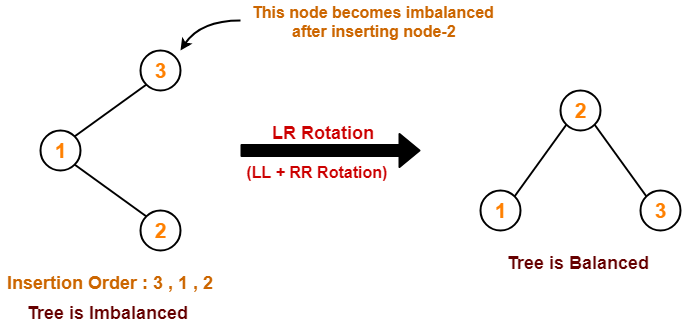
**Case-01:**



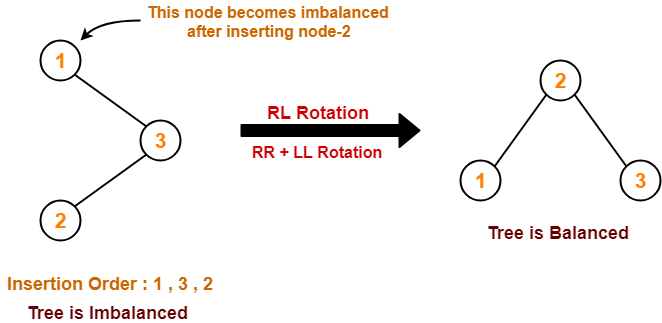
**Case-02:**



**Case-03:**



**Case-04:**



[**AVL Tree Properties | Problems on AVL Tree**](https://www.gatevidyalay.com/avl-tree-properties-avl-trees/)

**AVL Tree-**

We have discussed-

* AVL trees are self-balancing binary search trees.
* In AVL trees, balancing factor of each node is either 0 or 1 or -1.

**AVL Tree Properties-**

Important properties of AVL tree are-

**Property-01:**

|  |
| --- |
| **Maximum possible number of nodes in AVL tree of height H**  **= 2H+1 – 1** |

**Example-**

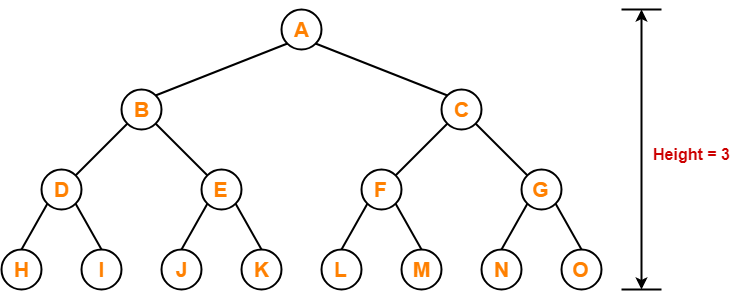
Maximum possible number of nodes in AVL tree of height-3

= 23+1 – 1

= 16 – 1

= 15

Thus, in AVL tree of height-3, maximum number of nodes that can be inserted = 15.



We can not insert more number of nodes in this AVL tree.

**Property-02:**

|  |
| --- |
| **Minimum number of nodes in AVL Tree of height H is given by a recursive relation-**  **N(H) = N(H-1) + N(H-2) + 1** |

Base conditions for this recursive relation are-

* N(0) = 1
* N(1) = 2

**Example-**

**N(H) = N(H-1) + N(H-2) + 1**

**N(4) = N(4-1) + N(4-2) + 1**

N(4)=N(3)+N(2)+1

N(4)=7+4+1=12

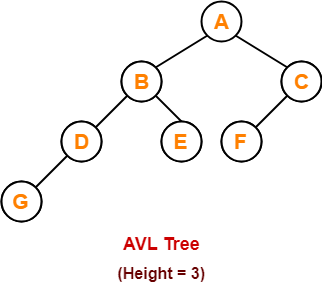
N(3)=7

N(2)=N(2-1)+N(2-2)+1

N(2)=N(1)+N(0)+1

N(2)=2+1+1=4

Minimum possible number of nodes in AVL tree of height-3 = 7



(For explanation, Refer problem-01)

**Property-03:**

|  |
| --- |
| **Minimum possible height of AVL Tree using N nodes**  **= ⌊log2N⌋** |

**Example-**

Minimum possible height of AVL Tree using 8 nodes

= ⌊log28⌋

= ⌊log223⌋

= ⌊3log22⌋

= ⌊3⌋

= 3

**Property-04:**

|  |
| --- |
| **Maximum height of AVL Tree using N nodes is calculated using recursive relation-**  **N(H) = N(H-1) + N(H-2) + 1** |

Base conditions for this recursive relation are-

* N(0) = 1
* N(1) = 2

**NOTE-**

* If there are n nodes in AVL Tree, its maximum height can not exceed 1.44log2n.
* In other words, Worst case height of AVL Tree with n nodes = 1.44log2n.

**PRACTICE PROBLEMS BASED ON AVL TREE PROPERTIES-**

**Problem-01:**

Find the minimum number of nodes required to construct AVL Tree of height = 3.

**Solution-**

 We know, minimum number of nodes in AVL tree of height H is given by a recursive relation-

|  |
| --- |
| **N(H) = N(H-1) + N(H-2) + 1** |

 where N(0) = 1 and N(1) = 2

**Step-01:**

 Substituting H = 3 in the recursive relation, we get-

N(3) = N(3-1) + N(3-2) + 1

N(3) = N(2) + N(1) + 1

N(3) = N(2) + 2 + 1                (Using base condition)

N(3) = N(2) + 3                      …………(1)

 To solve this recursive relation, we need the value of N(2).

**Step-02:**

 Substituting H = 2 in the recursive relation, we get-

N(2) = N(2-1) + N(2-2) + 1

N(2) = N(1) + N(0) + 1

N(2) = 2 + 1 + 1                 (Using base conditions)

∴ N(2) = 4                          …………(2)

**Step-03:**

 Using (2) in (1), we get-

N(3) = 4 + 3

∴ N(3) = 7

Thus, minimum number of nodes required to construct AVL tree of height-3 = 7.

**Problem-02:**

 Find the minimum number of nodes required to construct AVL Tree of height = 4.

**Solution-**

 We know, minimum number of nodes in AVL tree of height H is given by a recursive relation-

|  |
| --- |
| **N(H) = N(H-1) + N(H-2) + 1** |

where N(0) = 1 and N(1) = 2

**Step-01:**

Substituting H = 4 in the recursive relation, we get-

N(4) = N(4-1) + N(4-2) + 1

N(4) = N(3) + N(2) + 1               …………(1)

To solve this recursive relation, we need the value of N(2) and N(3).

**Step-02:**

 Substituting H = 2 in the recursive relation, we get-

N(2) = N(2-1) + N(2-2) + 1

N(2) = N(1) + N(0) + 1

N(2) = 2 + 1 + 1                 (Using base conditions)

∴ N(2) = 4                          …………(2)

**Step-03:**

 Substituting H = 3 in the recursive relation, we get-

N(3) = N(3-1) + N(3-2) + 1

N(3) = N(2) + N(1) + 1

N(3) = 4 + 2 + 1                 (Using (2) and base condition)

∴ N(3) = 7                          …………(3)

**Step-04:**

 Using (2) and (3) in (1), we get-

N(4) = 7 + 4 + 1

∴ N(4) = 12

Thus, minimum number of nodes required to construct AVL tree of height-4 = 12.

**Problem-03:**

What is the maximum height of any AVL tree with 10 nodes?

**Solution-**

For calculating the maximum height of AVL tree with n nodes, we use a recursive relation-

|  |
| --- |
| **N(H) = N(H-1) + N(H-2) + 1** |

**Step-01:**

Substituting H = 2 in the recursive relation, we get-

N(2) = N(2-1) + N(2-2) + 1

N(2) = N(1) + N(0) + 1

N(2) = 2 + 1 + 1                (Using base conditions)

∴ N(2) = 4                          …………(1)

So, minimum number of nodes required to construct AVL tree of height-2 = 4.

**Step-02:**

Substituting H = 3 in the recursive relation, we get-

N(3) = N(3-1) + N(3-2) + 1

N(3) = N(2) + N(1) + 1

N(3) = 4 + 2 + 1                (Using (1) and base condition)

∴ N(3) = 7                          …………(2)

So, minimum number of nodes required to construct AVL tree of height-3 = 7.

**Step-03:**

Substituting H = 4 in the recursive relation, we get-

N(4) = N(4-1) + N(4-2) + 1

N(4) = N(3) + N(2) + 1

N(4) = 7 + 4 + 1                (Using (1) and (2))

∴ N(4) = 12

So, minimum number of nodes required to construct AVL tree of height-4 = 12.

But given number of nodes = 10 which is less than 12.

Thus, maximum height of AVL tree that can be obtained using 10 nodes = 3.

**Problem-04:**

What is the maximum height of any AVL tree with 77 nodes?

**Solution-**

For calculating the maximum height of AVL tree with n nodes, we use a recursive relation-

|  |
| --- |
| **N(H) = N(H-1) + N(H-2) + 1** |

**Step-01:**

Substituting H = 2 in the recursive relation, we get-

N(2) = N(2-1) + N(2-2) + 1

N(2) = N(1) + N(0) + 1

N(2) = 2 + 1 + 1                (Using base conditions)

∴ N(2) = 4                          …………(1)

So, minimum number of nodes required to construct AVL tree of height-2 = 4.

**Step-02:**

Substituting H = 3 in the recursive relation, we get-

N(3) = N(3-1) + N(3-2) + 1

N(3) = N(2) + N(1) + 1

N(3) = 4 + 2 + 1                (Using (1) and base condition)

∴ N(3) = 7                          …………(2)

So, minimum number of nodes required to construct AVL tree of height-3 = 7.

**Step-03:**

Substituting H = 4 in the recursive relation, we get-

N(4) = N(4-1) + N(4-2) + 1

N(4) = N(3) + N(2) + 1

N(4) = 7 + 4 + 1                (Using (1) and (2))

∴ N(4) = 12                        …………(3)

So, minimum number of nodes required to construct AVL tree of height-4 = 12.

**Step-04:**

Substituting H = 5 in the recursive relation, we get-

N(5) = N(5-1) + N(5-2) + 1

N(5) = N(4) + N(3) + 1

N(5) = 12 + 7 + 1                (Using (2) and (3))

∴ N(5) = 20                          …………(4)

So, minimum number of nodes required to construct AVL tree of height-5 = 20.

**Step-05:**

Substituting H = 6 in the recursive relation, we get-

N(6) = N(6-1) + N(6-2) + 1

N(6) = N(5) + N(4) + 1

N(6) = 20 + 12 + 1                (Using (3) and (4))

∴ N(6) = 33                            …………(5)

So, minimum number of nodes required to construct AVL tree of height-6 = 33.

**Step-06:**

Substituting H = 7 in the recursive relation, we get-

N(7) = N(7-1) + N(7-2) + 1

N(7) = N(6) + N(5) + 1

N(7) = 33 + 20 + 1                (Using (4) and (5))

∴ N(7) = 54                            …………(6)

So, minimum number of nodes required to construct AVL tree of height-7 = 54.

**Step-07:**

Substituting H = 8 in the recursive relation, we get-

N(8) = N(8-1) + N(8-2) + 1

N(8) = N(7) + N(6) + 1

N(8) = 54 + 33 + 1                (Using (5) and (6))

∴ N(8) = 88                            …………(6)

So, minimum number of nodes required to construct AVL tree of height-8 = 88.

But given number of nodes = 77 which is less than 88.

Thus, maximum height of AVL tree that can be obtained using 77 nodes = 7.

[**AVL Tree Insertion | Insertion in AVL Tree**](https://www.gatevidyalay.com/insertion-in-avl-tree-avl-trees/)

**AVL Tree-**

We have discussed-

* AVL trees are self-balancing binary search trees.
* In AVL trees, balancing factor of each node is either 0 or 1 or -1.

**Insertion in AVL Tree-**

|  |
| --- |
| Insertion Operation is performed to insert an element in the AVL Tree. |

To insert an element in the AVL tree, follow the following steps-

* Insert the element in the AVL tree in the same way the insertion is performed in BST.
* After insertion, check the balance factor of each node of the resulting tree.

Now, following two cases are possible-

**Case-01:**

* After the insertion, the balance factor of each node is either 0 or 1 or -1.
* In this case, the tree is considered to be balanced.
* Conclude the operation.
* Insert the next element if any.

**Case-02:**

* After the insertion, the balance factor of at least one node is not 0 or 1 or -1.
* In this case, the tree is considered to be imbalanced.
* Perform the suitable rotation to balance the tree.
* After the tree is balanced, insert the next element if any.

**Rules To Remember-**

**Rule-01:**

After inserting an element in the existing AVL tree,

* Balance factor of only those nodes will be affected that lies on the path from the newly inserted node to the root node.

**Rule-02:**

To check whether the AVL tree is still balanced or not after the insertion,

* There is no need to check the balance factor of every node.
* Check the balance factor of only those nodes that lies on the path from the newly inserted node to the root node.

**Rule-03:**

After inserting an element in the AVL tree,

* If tree becomes imbalanced, then there exists one particular node in the tree by balancing which the entire tree becomes balanced automatically.
* To re balance the tree, balance that particular node.

To find that particular node,

* Traverse the path from the newly inserted node to the root node.
* Check the balance factor of each node that is encountered while traversing the path.
* The first encountered imbalanced node will be the node that needs to be balanced.

To balance that node,

* Count three nodes in the direction of leaf node.
* Then, use the concept of AVL tree rotations to re balance the tree.

 NOVEMBER 08, 2023 IT

**PRACTICE PROBLEM BASED ON AVL TREE INSERTION-**

**Problem-**

Construct AVL Tree for the following sequence of numbers-

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

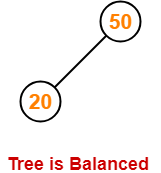
**Solution-**

**Step-01: Insert 50**



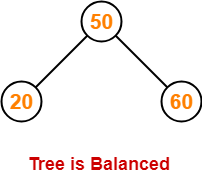
**Step-02: Insert 20**

* As 20 < 50, so insert 20 in 50’s left sub tree.



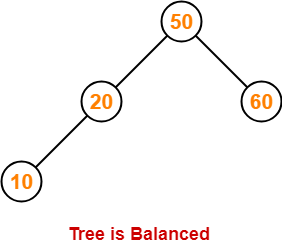
**Step-03: Insert 60**

* As 60 > 50, so insert 60 in 50’s right sub tree.



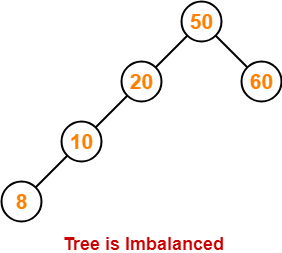
**Step-04: Insert 10**

* As 10 < 50, so insert 10 in 50’s left sub tree.
* As 10 < 20, so insert 10 in 20’s left sub tree.



**Step-05: Insert 8**

* As 8 < 50, so insert 8 in 50’s left sub tree.
* As 8 < 20, so insert 8 in 20’s left sub tree.
* As 8 < 10, so insert 8 in 10’s left sub tree.

 height of LST=3

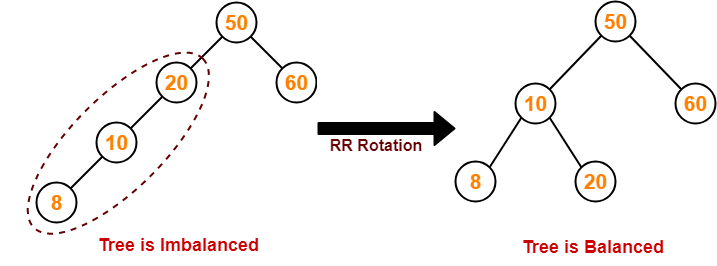
Height of RST=1

 3-1=2

To balance the tree,

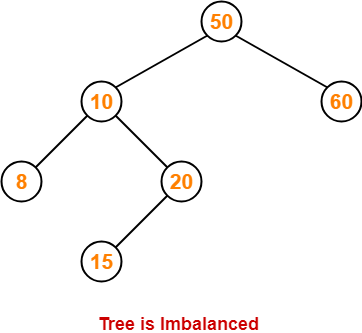
* Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
* The first imbalanced node is node 20.
* Now, count three nodes from node 20 in the direction of leaf node.
* Then, use AVL tree rotation to balance the tree.

Following this, we have-



**Step-06: Insert 15**

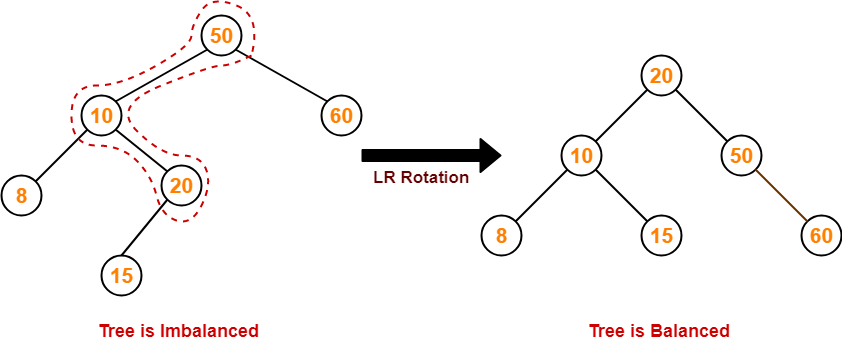
* As 15 < 50, so insert 15 in 50’s left sub tree.
* As 15 > 10, so insert 15 in 10’s right sub tree.
* As 15 < 20, so insert 15 in 20’s left sub tree.



To balance the tree,

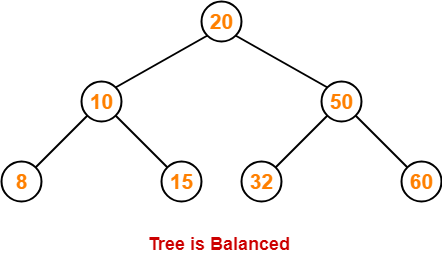
* Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
* The first imbalanced node is node 50.
* Now, count three nodes from node 50 in the direction of leaf node.
* Then, use AVL tree rotation to balance the tree.

Following this, we have-



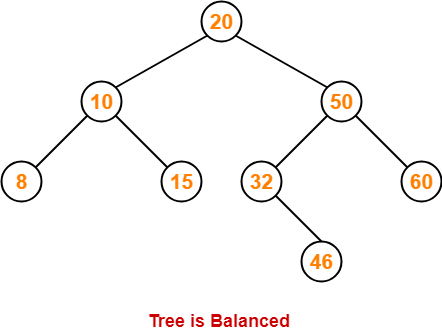
**Step-07: Insert 32**

* As 32 > 20, so insert 32 in 20’s right sub tree.
* As 32 < 50, so insert 32 in 50’s left sub tree.



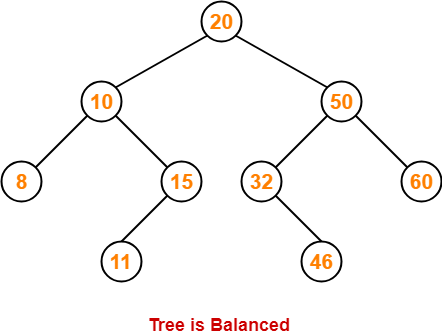
**Step-08: Insert 46**

* As 46 > 20, so insert 46 in 20’s right sub tree.
* As 46 < 50, so insert 46 in 50’s left sub tree.
* As 46 > 32, so insert 46 in 32’s right sub tree.



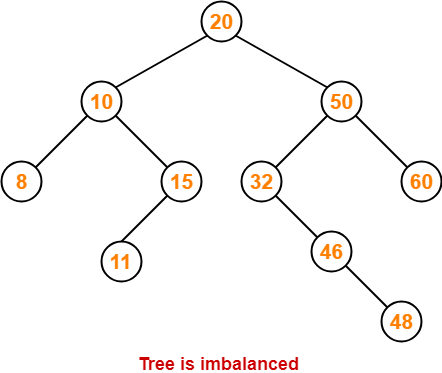
**Step-09: Insert 11**

* As 11 < 20, so insert 11 in 20’s left sub tree.
* As 11 > 10, so insert 11 in 10’s right sub tree.
* As 11 < 15, so insert 11 in 15’s left sub tree.



**Step-10: Insert 48**

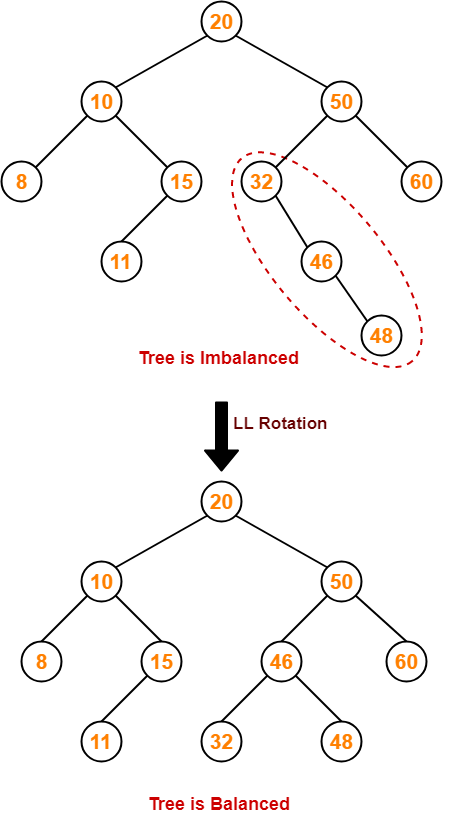
* As 48 > 20, so insert 48 in 20’s right sub tree.
* As 48 < 50, so insert 48 in 50’s left sub tree.
* As 48 > 32, so insert 48 in 32’s right sub tree.
* As 48 > 46, so insert 48 in 46’s right sub tree.



To balance the tree,

* Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
* The first imbalanced node is node 32.
* Now, count three nodes from node 32 in the direction of leaf node.
* Then, use AVL tree rotation to balance the tree.

Following this, we have-



This is the final balanced AVL tree after inserting all the given elements.

[**Heap Data Structure | Binary Heap | Examples**](https://www.gatevidyalay.com/heap-data-structure-binary-heap-examples/)

**Heap Data Structure-**

In data structures,

* Heap is a specialized data structure.
* It has special characteristics.
* A heap may be implemented using a n-ary tree.

**Binary Heap-**

A binary heap is a [**Binary Tree**](https://www.gatevidyalay.com/binary-tree-types-of-trees-in-data-structure/) with the following two properties-



* Ordering Property
* Structural Property

**1. Ordering Property-**

By this property,

* Elements in the heap tree are arranged in specific order.
* This gives rise to two types of heaps- min heap and max heap.

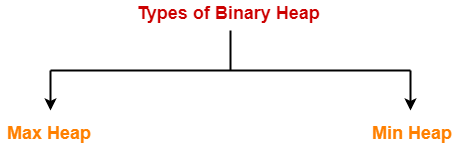
**2. Structural Property-**

By this property,

* Binary heap is an almost complete binary tree.
* It has all its levels completely filled except possibly the last level.
* The last level is strictly filled from left to right.

**Types of Binary Heap-**

Depending on the arrangement of elements, a binary heap may be of following two types-



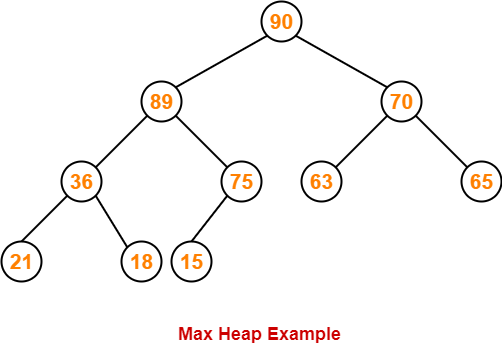
1. Max Heap
2. Min Heap

**1. Max Heap-**

* Max Heap conforms to the above properties of heap.
* **In max heap, every node contains greater or equal value element than its child nodes.**
* Thus, root node contains the largest value element.

**Example-**

Consider the following example of max heap-



This is max heap because-

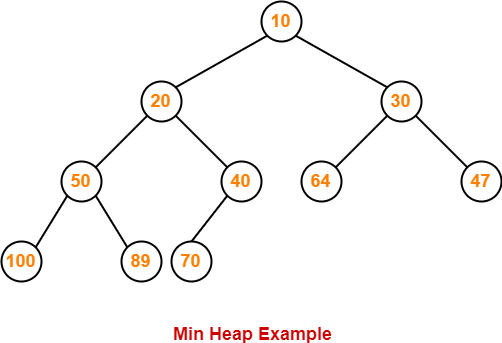
* Every node contains greater or equal value element than its child nodes.
* It is an almost complete binary tree with its last level strictly filled from left to right.

**2. Min Heap-**

* Min Heap conforms to the above properties of heap.
* In min heap, every node contains lesser value element than its child nodes.
* Thus, root node contains the smallest value element.

**Example-**

Consider the following example of min heap-

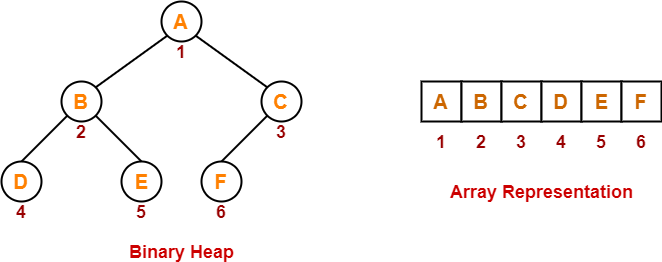


This is min heap because-

* Every node contains lesser value element than its child nodes.
* It is an almost complete binary tree with its last level strictly filled from left to right.

**Array Representation of Binary Heap-**

A binary heap is typically represented as an array.



  0 1 2 3 4 5

For a node present at index ‘i’ of the array Arr-

**If indexing starts with 0,**

**I=2**

* Its parent node will be present at array location = Arr [ i/2 ]=Arr[2/2]=Arr[1]=B

Its left child node will be present at array location = Arr [ 2i+1 ]=Arr[2\*2+1]=Arr[5]=F

* Its right child node will be present at array location = Arr [ 2i+2 ] = Arr[2\*1+2]=Arr[4]=E

**If indexing starts with 1,**

* Its parent node will be present at array location = Arr [ ⌊i/2⌋ ]
* Its left child node will be present at array location = Arr [ 2i ]
* Its right child node will be present at array location = Arr [ 2i+1 ]

**Important Notes-**

**Note-01:**

* Level order traversal technique may be used to achieve the array representation of a heap tree.
* Array representation of a heap never contains any empty indices in between.
* However, array representation of a binary tree may contain some empty indices in between.

**Note-02:**

Given an array representation of a binary heap,

* If all the elements are in descending order, then heap is definitely a max heap.
* If all the elements are not in descending order, then it may or may not be a max heap.
* If all the elements are in ascending order, then heap is definitely a min heap.
* If all the elements are not in ascending order, then it may or may not be a min heap.

**Note-03:**

* In max heap, every node contains greater or equal value element than all its descendants.
* In min heap, every node contains smaller value element that all its descendants.

**PRACTICE PROBLEMS BASED ON HEAP DATA STRUCTURE-**

**Problems-**

Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

1. 25, 14, 16, 13, 10, 8, 12
2. 25, 12, 16, 13, 10, 8, 14
3. 25, 14, 12, 13, 10, 8, 16
4. 25, 14, 13, 16, 10, 8, 12

**Solutions-**

**Part-01: 25, 14, 16, 13, 10, 8, 12-**

  25

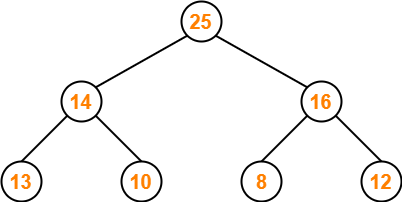
14

13 16

10

8 12

The given array representation may be converted into the following structure-



25

14

13 16

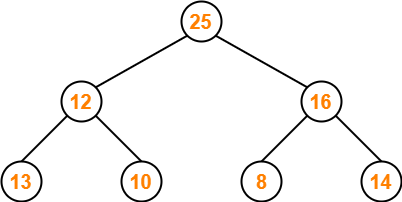
Clearly,

* It is a complete binary tree.
* Every node contains a greater value element than its child nodes.

Thus, the given array represents a max heap.

**Part-02: 25, 12, 16, 13, 10, 8, 14-**

The given array representation may be converted into the following structure-



**25, 12, 16, 13, 10, 8, 14-**

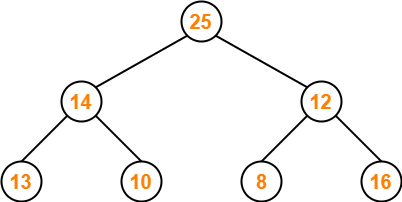
Clearly,

* It is a complete binary tree.
* Every node does not contain a greater value element than its child nodes. (Node 12)
* So, it is not a max heap.
* Every node does not contain a smaller value element than its child nodes.
* So, it is not a min heap.

Thus, the given array does not represents a heap.

**Part-03: 25, 14, 12, 13, 10, 8, 16-**

The given array representation may be converted into the following structure-



**25, 14, 12, 13, 10, 8, 16**

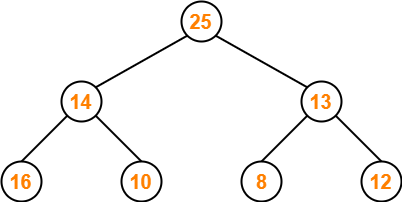
Clearly,

* It is a complete binary tree.
* Every node does not contain a greater value element than its child nodes. (Node 12)
* So, it is not a max heap.
* Every node does not contain a smaller value element than its child nodes.
* So, it is not a min heap.

Thus, the given array does not represents a heap.

**Part-04: 25, 14, 13, 16, 10, 8, 12-**

The given array representation may be converted into the following structure-



**25, 14, 13, 16, 10, 8, 12**

Clearly,

* It is a complete binary tree.
* Every node does not contain a greater value element than its child nodes. (Node 14)
* So, it is not a max heap.
* Every node does not contain a smaller value element than its child nodes.
* So, it is not a min heap.

Thus, the given array does not represents a heap.

**Heap Operations | Max Heap Operations | Examples**

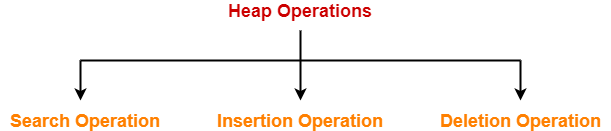
**Heap Data Structure-**

We have discussed-

* Heap is a specialized data structure with special properties.
* A binary heap is a binary tree that has ordering and structural properties.
* A heap may be a max heap or a min heap.

**Heap Operations-**

The most basic and commonly performed operations on a heap are-



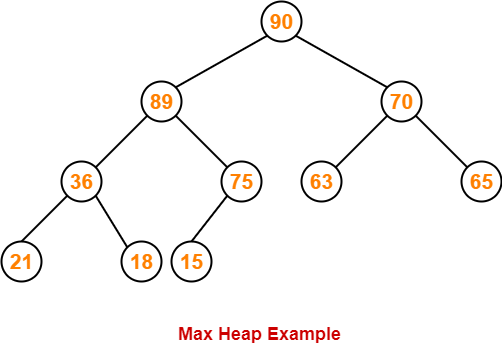
1. Search Operation
2. Insertion Operation
3. Deletion Operation

**Max Heap-**

* **In max heap, every node contains greater or equal value element than its child nodes.**
* Thus, root node contains the largest value element.

**Example-**

The following heap is an example of a max heap-



**Max Heap Operations-**

We will discuss the construction of a max heap and how following operations are performed on a max heap-

* Finding Maximum Operation
* Insertion Operation
* Deletion Operation

**Max Heap Construction-**

Given an array of elements, the steps involved in constructing a max heap are-

**Step-01:**

Convert the given array of elements into an almost complete binary tree.

**Step-02:**

Ensure that the tree is a max heap.

* Check that every non-leaf node contains a greater or equal value element than its child nodes.
* If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
* Start checking from a non-leaf node with the highest index (bottom to top and right to left).

**Finding Maximum Operation-**

* In max heap, the root node always contains the maximum value element.
* So, we directly display the root node value as maximum value in max heap.

**Insertion Operation-**

|  |
| --- |
| Insertion Operation is performed to insert an element in the heap tree. |

The steps involved in inserting an element are-

**Step-01:**

Insert the new element as a next leaf node from left to right.

**Step-02:**

Ensure that the tree remains a max heap.

* Check that every non-leaf node contains a greater or equal value element than its child nodes.
* If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
* Start checking from a non-leaf node with the highest index (bottom to top and right to left).

**Deletion Operation-**

|  |
| --- |
| Deletion Operation is performed to delete a particular element from the heap tree. |

When it comes to deleting a node from the heap tree, following two cases are possible-

**Case-01: Deletion Of Last Node-**

* This case is pretty simple.
* Just remove / disconnect the last leaf node from the heap tree.

**Case-02: Deletion Of Some Other Node-**

* This case is little bit difficult.
* Deleting a node other than the last node disturbs the heap properties.

The steps involved in deleting such a node are-

**Step-01:**

* Delete the desired element from the heap tree.
* Pluck the last node and put in place of the deleted node.

**Step-02:**

Ensure that the tree remains a max heap.

* Check that every non-leaf node contains a greater or equal value element than its child nodes.
* If there exists any node that does not satisfies the ordering property of max heap, swap the elements.
* Start checking from a non-leaf node with the highest index (bottom to top and right to left).

**PRACTICE PROBLEMS BASED ON MAX HEAP OPERATIONS-**

**Problem-01:**

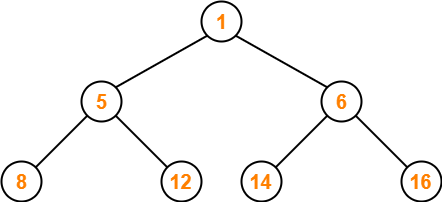
Construct a max heap for the given array of elements-

1, 5, 6, 8, 12, 14, 16

**Solution-**

**Step-01:**

We convert the given array of elements into an almost complete binary tree-

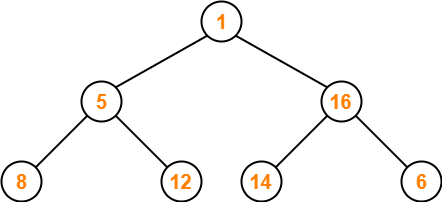


1, 5, 6, 8, 12, 14, 16

**Step-02:**

* We ensure that the tree is a max heap.
* Node 6 contains greater element in its right child node.
* So, we swap node 6 and node 16.

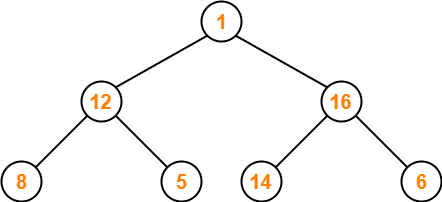
The resulting tree is-



**Step-03:**

* Node 5 contains greater element in its right child node.
* So, we swap node 5 and node 12.

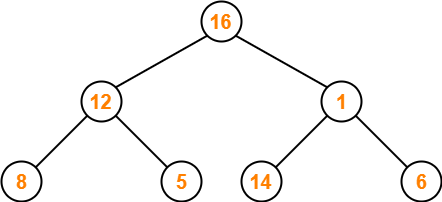
The resulting tree is-



**Step-04:**

* Node 1 contains greater element in its right child node.
* So, we swap node 1 and node 16.

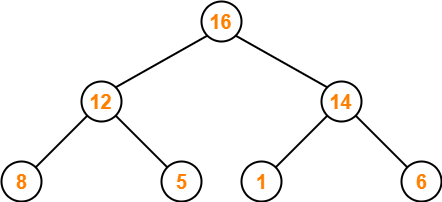
The resulting tree is-



**Step-05:**

* Node 1 contains greater element in its left child node.
* So, we swap node 1 and node 14.

The resulting tree is-



This is the required max heap for the given array of elements.

**Problem-02:**

Consider the following max heap-

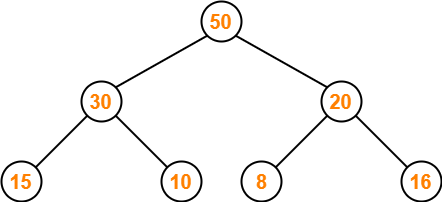
50, 30, 20, 15, 10, 8, 16

Insert a new node with value 60.

**Solution-**

**Step-01:**

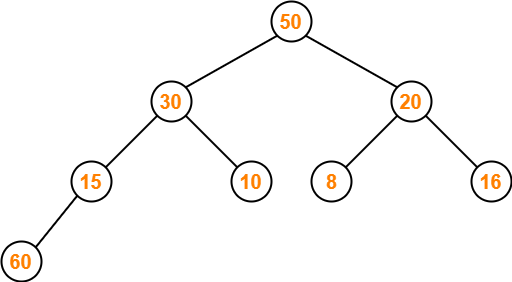
We convert the given array of elements into a heap tree-



**Step-02:**

We insert the new element 60 as a next leaf node from left to right.

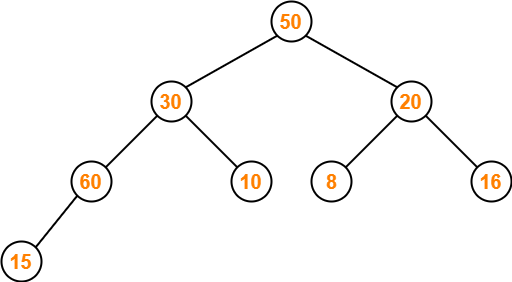
The resulting tree is-



**Step-03:**

* We ensure that the tree is a max heap.
* Node 15 contains greater element in its left child node.
* So, we swap node 15 and node 60.

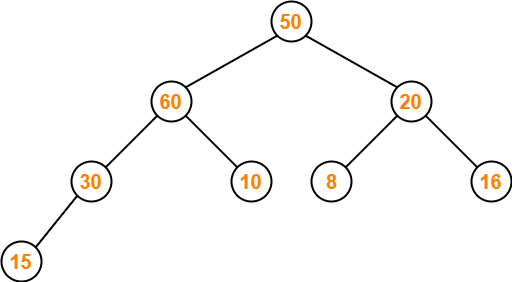
The resulting tree is-



**Step-04:**

* Node 30 contains greater element in its left child node.
* So, we swap node 30 and node 60.

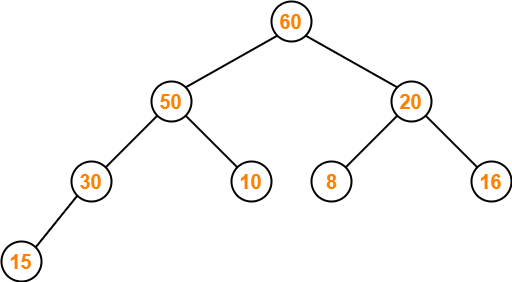
The resulting tree is-



**Step-05:**

* Node 50 contains greater element in its left child node.
* So, we swap node 50 and node 60.

The resulting tree is-



This is the required max heap after inserting the node with value 60.

**Problem-03:**

Consider the following max heap-

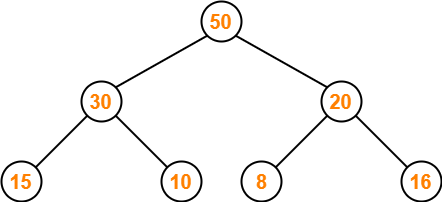
50, 30, 20, 15, 10, 8, 16

Delete a node with value 50.

**Solution-**

**Step-01:**

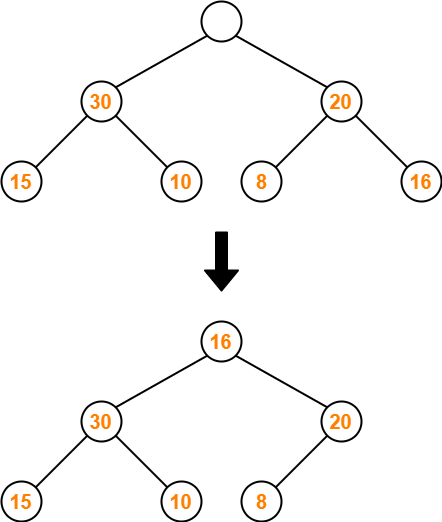
We convert the given array of elements into a heap tree-



**Step-02:**

* We delete the element 50 which is present at root node.
* We pluck the last node 16 and put in place of the deleted node.

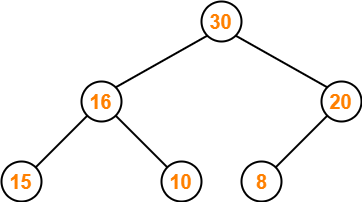
The resulting tree is-



**Step-03:**

* We ensure that the tree is a max heap.
* Node 16 contains greater element in its left child node.
* So, we swap node 16 and node 30.

The resulting tree is-



This is the required max heap after deleting the node with value 50.

[**Hashing in Data Structure | Hash Functions**](https://www.gatevidyalay.com/hashing/)

**Searching Techniques-**

In data structures,

* There are several searching techniques like linear search, binary search, search trees etc.
* In these techniques, time taken to search any particular element depends on the total number of elements.

**Example-**

* [**Linear Search**](https://www.gatevidyalay.com/linear-search-searching-algorithms/) takes O(n) time to perform the search in unsorted arrays consisting of n elements.
* [**Binary Search**](https://www.gatevidyalay.com/binary-search-binary-search-algorithm/) takes O(logn) time to perform the search in sorted arrays consisting of n elements.
* It takes O(logn) time to perform the search in [**Binary Search Tree**](https://www.gatevidyalay.com/binary-search-trees-data-structures/)consisting of n elements.

**Drawback-**

The main drawback of these techniques is-

* As the number of elements increases, time taken to perform the search also increases.
* This becomes problematic when total number of elements become too large.

**Hashing in Data Structure-**

In data structures,

* Hashing is a well-known technique to search any particular element among several elements.
* It minimizes the number of comparisons while performing the search.

**Advantage-**

Unlike other searching techniques,

* Hashing is extremely efficient.
* The time taken by it to perform the search does not depend upon the total number of elements.
* It completes the search with constant time complexity O(1).

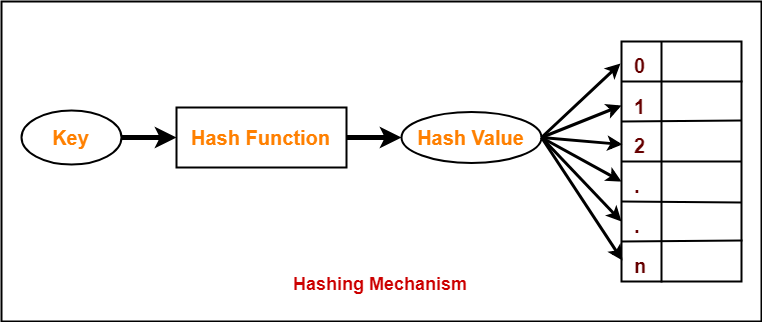
**Hashing Mechanism-**

In hashing,

* An array data structure called as **Hash table** is used to store the data items.
* Based on the hash key value, data items are inserted into the hash table.

**Hash Key Value-**

* Hash key value is a special value that serves as an index for a data item.
* It indicates where the data item should be stored in the hash table.
* Hash key value is generated using a hash function.



**Hash Function-**

|  |
| --- |
| Hash function is a function that maps any big number or string to a small integer value. |

* Hash function takes the data item as an input and returns a small integer value as an output.
* The small integer value is called as a hash value.
* Hash value of the data item is then used as an index for storing it into the hash table.

H(x)=H(y) it does not mean x=y

**Types of Hash Functions-**

There are various types of hash functions available such as-

1. Mid Square Hash Function
2. Division Hash Function
3. Folding Hash Function etc

It depends on the user which hash function he wants to use.

**Properties of Hash Function-**

The properties of a good hash function are-

* It is efficiently computable.
* It minimizes the number of collisions.
* It distributes the keys uniformly over the table.

[**Separate Chaining | Collision Resolution Techniques**](https://www.gatevidyalay.com/collision-resolution-techniques-separate-chaining/)

**Hashing in Data Structure-**

We have discussed-

* Hashing is a well-known searching technique.
* It minimizes the number of comparisons while performing the search.
* It completes the search with constant time complexity O(1).

**Collision in Hashing-**

In hashing,

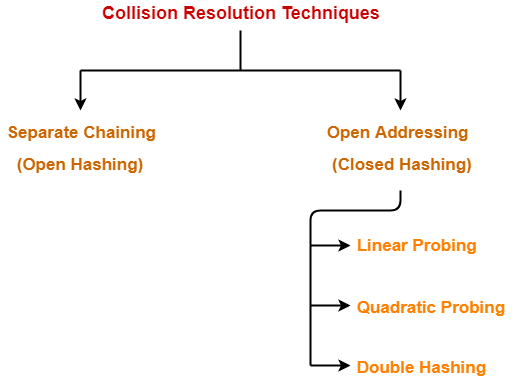
* Hash function is used to compute the hash value for a key.
* Hash value is then used as an index to store the key in the hash table.
* Hash function may return the same hash value for two or more keys.

|  |
| --- |
| When the hash value of a key maps to an already occupied bucket of the hash table,  it is called as a **Collision**. |

**Collision Resolution Techniques-**

|  |
| --- |
| Collision Resolution Techniques are the techniques used for resolving or handling the collision. |

Collision resolution techniques are classified as-



1. Separate Chaining
2. Open Addressing

**Separate Chaining-**

To handle the collision,

* This technique creates a linked list to the slot for which collision occurs.
* The new key is then inserted in the linked list.
* These linked lists to the slots appear like chains.
* That is why, this technique is called as **separate chaining**.

**Time Complexity-**

**For Searching-**

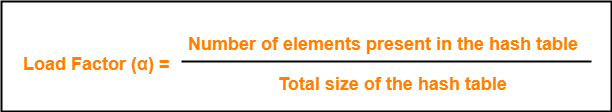
* In worst case, all the keys might map to the same bucket of the hash table.
* In such a case, all the keys will be present in a single linked list.
* Sequential search will have to be performed on the linked list to perform the search.
* So, time taken for searching in worst case is O(n).

**For Deletion-**

* In worst case, the key might have to be searched first and then deleted.
* In worst case, time taken for searching is O(n).
* So, time taken for deletion in worst case is O(n).

**Load Factor (α)-**

Load factor (α) is defined as-



If Load factor (α) = constant, then time complexity of Insert, Search, Delete = Θ(1)

**PRACTICE PROBLEM BASED ON SEPARATE CHAINING-**

**Problem-**

-3 mod 5= -3 +5 mod 5= 2 mod 5=2

11/ 13 mod 7= (11+n\*7)/13 mod 7

For n=1 ,11+n\*7= 11+1\*7=18

For n=2 ,11+2\*7= 11+2\*7=11+14=25

For n=3 ,11+3\*7= 11+3\*7=11+21=32

For n=4 ,11+4\*7= 11+4\*7=11 + 28=39 /13= 3 mod 7=3

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

 50 mod 7=1, 700 mod 7=3, 76 mod 7=6, 85 mod 7= 1, 92 mod 7=1, 73 mod 7=3, 101 mod 7=3

Use separate chaining technique for collision resolution.

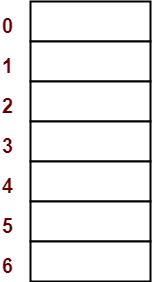
**Solution-**

The given sequence of keys will be inserted in the hash table as-

 1,0,6,1,1,3,3

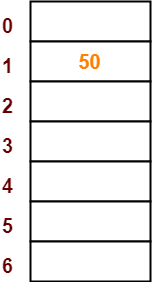
**Step-01:**

* Draw an empty hash table.
* For the given hash function, the possible range of hash values is [0, 6].
* So, draw an empty hash table consisting of 7 buckets as-



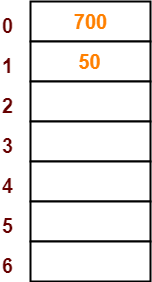
**Step-02:**

* Insert the given keys in the hash table one by one.
* The first key to be inserted in the hash table = 50.
* Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
* So, key 50 will be inserted in bucket-1 of the hash table as-

 50, 700, 76, 85, 92, 73 and 101

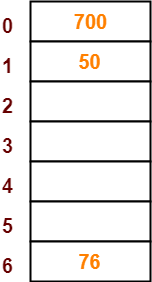
**Step-03:**

* The next key to be inserted in the hash table = 700.
* Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
* So, key 700 will be inserted in bucket-0 of the hash table as-



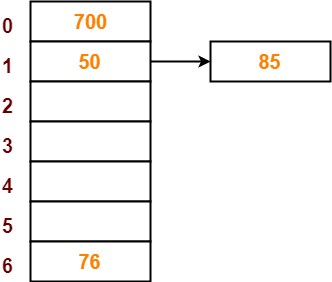
**Step-04:**

* The next key to be inserted in the hash table = 76.
* Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
* So, key 76 will be inserted in bucket-6 of the hash table as-



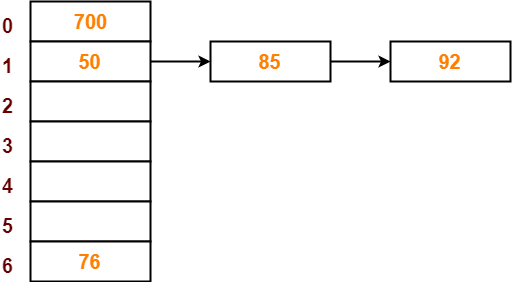
**Step-05:**

* The next key to be inserted in the hash table = 85.
* Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
* Since bucket-1 is already occupied, so collision occurs.
* Separate chaining handles the collision by creating a linked list to bucket-1.
* So, key 85 will be inserted in bucket-1 of the hash table as-



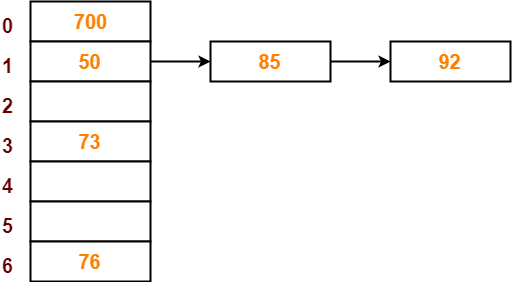
**Step-06:**

* The next key to be inserted in the hash table = 92.
* Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
* Since bucket-1 is already occupied, so collision occurs.
* Separate chaining handles the collision by creating a linked list to bucket-1.
* So, key 92 will be inserted in bucket-1 of the hash table as-



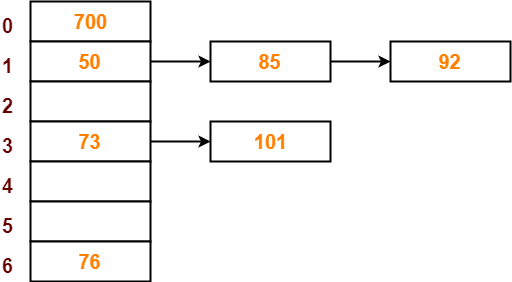
**Step-07:**

* The next key to be inserted in the hash table = 73.
* Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
* So, key 73 will be inserted in bucket-3 of the hash table as-



**Step-08:**

* The next key to be inserted in the hash table = 101.
* Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
* Since bucket-3 is already occupied, so collision occurs.
* Separate chaining handles the collision by creating a linked list to bucket-3.
* So, key 101 will be inserted in bucket-3 of the hash table as-

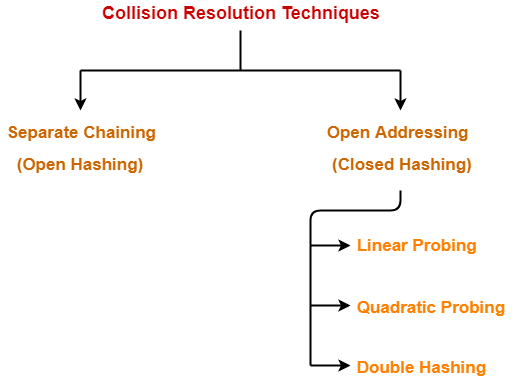


# [Open Addressing | Linear Probing | Collision](https://www.gatevidyalay.com/open-addressing-collision-resolution-techniques/)

## ****Collision Resolution Techniques-****

We have discussed-

* [**Hashing**](https://www.gatevidyalay.com/hashing/) is a well-known searching technique.
* Collision occurs when hash value of the new key maps to an occupied bucket of the hash table.
* Collision resolution techniques are classified as-



## ****Open Addressing-****

In open addressing,

* Unlike separate chaining, all the keys are stored inside the hash table.
* No key is stored outside the hash table.

Techniques used for open addressing are-

* Linear Probing
* Quadratic Probing
* Double Hashing

## ****Operations in Open Addressing-****

Let us discuss how operations are performed in open addressing-

### ****Insert Operation-****

* Hash function is used to compute the hash value for a key to be inserted.
* Hash value is then used as an index to store the key in the hash table.

In case of collision,

* Probing is performed until an empty bucket is found.
* Once an empty bucket is found, the key is inserted.
* Probing is performed in accordance with the technique used for open addressing.

### ****Search Operation-****

To search any particular key,

* Its hash value is obtained using the hash function used.
* Using the hash value, that bucket of the hash table is checked.
* If the required key is found, the key is searched.
* Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
* The empty bucket indicates that the key is not present in the hash table.

### ****Delete Operation-****

* The key is first searched and then deleted.
* After deleting the key, that particular bucket is marked as “deleted”.

## ****NOTE-****

* During insertion, the buckets marked as “deleted” are treated like any other empty bucket.
* During searching, the search is not terminated on encountering the bucket marked as “deleted”.
* The search terminates only after the required key or an empty bucket is found.

## ****Open Addressing Techniques-****

Techniques used for open addressing are-

## ****1. Linear Probing-****

In linear probing,

* When collision occurs, we linearly probe for the next bucket.
* We keep probing until an empty bucket is found.

### ****Advantage-****

* It is easy to compute.

### ****Disadvantage-****

* The main problem with linear probing is clustering.
* Many consecutive elements form groups.
* Then, it takes time to search an element or to find an empty bucket.

### ****Time Complexity-****

|  |
| --- |
| Worst time to search an element in linear probing is O (table size). |

This is because-

* Even if there is only one element present and all other elements are deleted.
* Then, “deleted” markers present in the hash table makes search the entire table.

## ****2. Quadratic Probing-****

In quadratic probing,

* When collision occurs, we probe for i2‘th bucket in ith iteration.
* We keep probing until an empty bucket is found.

## ****3. Double Hashing-****

In double hashing,

* We use another hash function hash2(x) and look for i \* hash2(x) bucket in ith iteration.
* It requires more computation time as two hash functions need to be computed.

## ****Comparison of Open Addressing Techniques-****

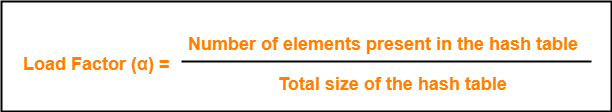
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Linear Probing** | **Quadratic Probing** | **Double Hashing** |
| **Primary Clustering** | Yes | No | No |
| **Secondary Clustering** | | Yes | Yes | No |
| **Number of Probe Sequence****(m = size of table)** | m | m | m2 |
| **Cache performance** | Best | Lies between the two | Poor |

### ****Conclusions-****

* Linear Probing has the best cache performance but suffers from clustering.
* Quadratic probing lies between the two in terms of cache performance and clustering.
* Double caching has poor cache performance but no clustering.

## ****Load Factor (α)-****

Load factor (α) is defined as-



In open addressing, the value of load factor always lie between 0 and 1.

This is because-

* In open addressing, all the keys are stored inside the hash table.
* So, size of the table is always greater or at least equal to the number of keys stored in the table.

## ****PRACTICE PROBLEM BASED ON OPEN ADDRESSING-****

## ****Problem-****

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

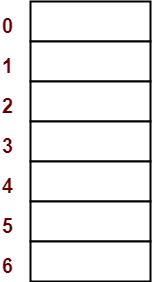
Use linear probing technique for collision resolution.

## ****Solution-****

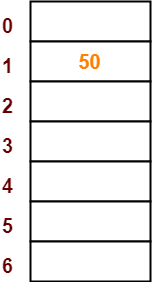
The given sequence of keys will be inserted in the hash table as-

 Draw an empty hash table.

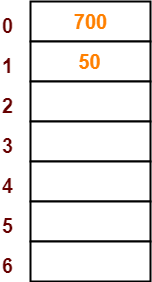
* For the given hash function, the possible range of hash values is [0, 6].
* So, draw an empty hash table consisting of 7 buckets as-



* Insert the given keys in the hash table one by one.
* The first key to be inserted in the hash table = 50.
* Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
* So, key 50 will be inserted in bucket-1 of the hash table as-

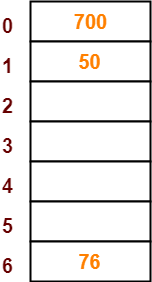


* The next key to be inserted in the hash table = 700.
* Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
* So, key 700 will be inserted in bucket-0 of the hash table as-



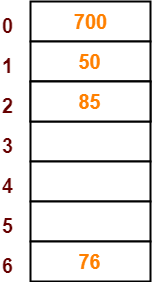
### ****Step-04:****

* The next key to be inserted in the hash table = 76.
* Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
* So, key 76 will be inserted in bucket-6 of the hash table as-



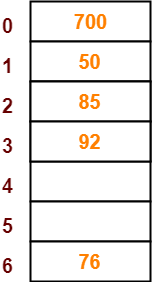
### ****Step-05:****

* The next key to be inserted in the hash table = 85.
* Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
* Since bucket-1 is already occupied, so collision occurs.
* To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
* The first empty bucket is bucket-2.
* So, key 85 will be inserted in bucket-2 of the hash table as-



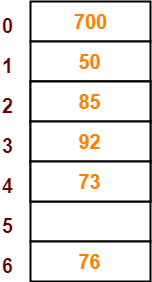
### ****Step-06:****

* The next key to be inserted in the hash table = 92.
* Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
* Since bucket-1 is already occupied, so collision occurs.
* To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
* The first empty bucket is bucket-3.
* So, key 92 will be inserted in bucket-3 of the hash table as-



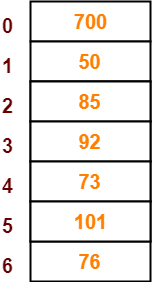
### ****Step-07:****

* The next key to be inserted in the hash table = 73.
* Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
* Since bucket-3 is already occupied, so collision occurs.
* To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
* The first empty bucket is bucket-4.
* So, key 73 will be inserted in bucket-4 of the hash table as-



### ****Step-08:****

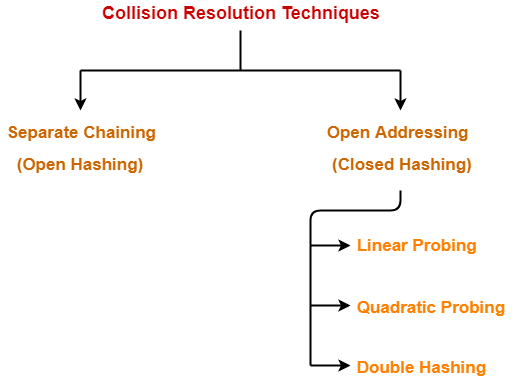
* The next key to be inserted in the hash table = 101.
* Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
* Since bucket-3 is already occupied, so collision occurs.
* To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
* The first empty bucket is bucket-5.
* So, key 101 will be inserted in bucket-5 of the hash table as-



[**Separate Chaining Vs Open Addressing**](https://www.gatevidyalay.com/separate-chaining-open-addressing-comparison/)

**Collision Resolution Techniques-**

In [**Hashing**](https://www.gatevidyalay.com/hashing/), collision resolution techniques are classified as-



1. [**Separate Chaining**](https://www.gatevidyalay.com/collision-resolution-techniques-separate-chaining/)
2. [**Open Addressing**](https://www.gatevidyalay.com/open-addressing-collision-resolution-techniques/)

**Separate Chaining Vs Open Addressing-**

|  |  |
| --- | --- |
| **Separate Chaining** | **Open Addressing** |
| Keys are stored inside the hash table as well as outside the hash table. | All the keys are stored only inside the hash table.  No key is present outside the hash table. |
| The number of keys to be stored in the hash table can even exceed the size of the hash table. | The number of keys to be stored in the hash table can never exceed the size of the hash table. |
| Deletion is easier. | Deletion is difficult. |
| Extra space is required for the pointers to store the keys outside the hash table. | No extra space is required. |
| Cache performance is poor.  This is because of linked lists which store the keys outside the hash table. | Cache performance is better.  This is because here no linked lists are used. |
| Some buckets of the hash table are never used which leads to wastage of space. | Buckets may be used even if no key maps to those particular buckets. |

**Which is the Preferred Technique?**

The performance of both the techniques depend on the kind of operations that are required to be performed on the keys stored in the hash table-

**Separate Chaining-**

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

* Insertion Operation
* Deletion Operation
* Searching Operation

**NOTE-**

* Deletion is easier in separate chaining.
* This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

**Open Addressing-**

Open addressing is advantageous when it is required to perform only the following operations on the keys stored in the hash table-

* Insertion Operation
* Searching Operation

**NOTE-**

* Deletion is difficult in open addressing.
* This is because deleting a key from the hash table requires some extra efforts.
* After deleting a key, certain keys have to be rearranged.