Advanced Data Structures

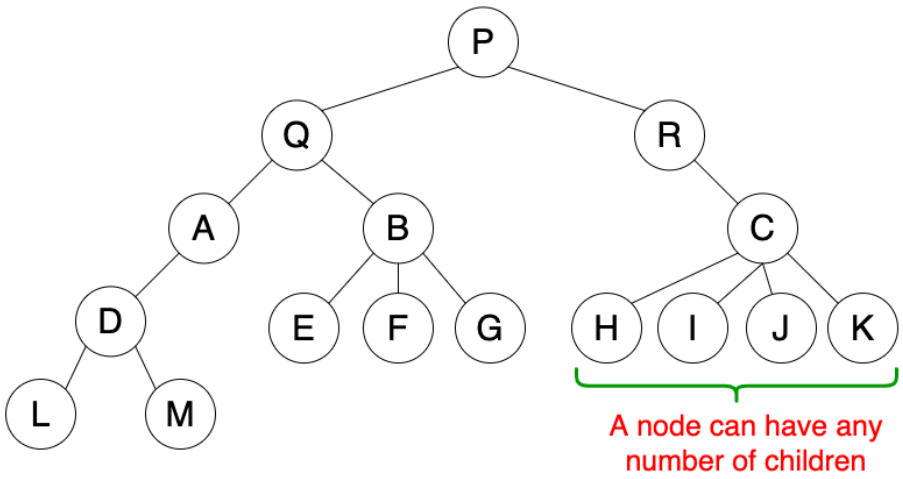
Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**? : It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data.

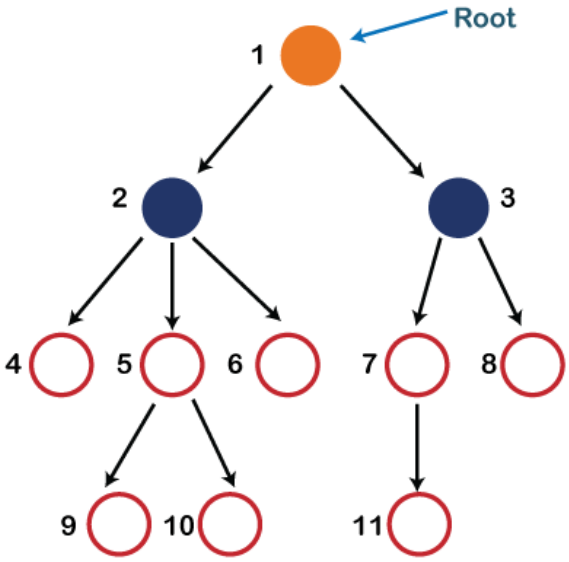


Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.

- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.



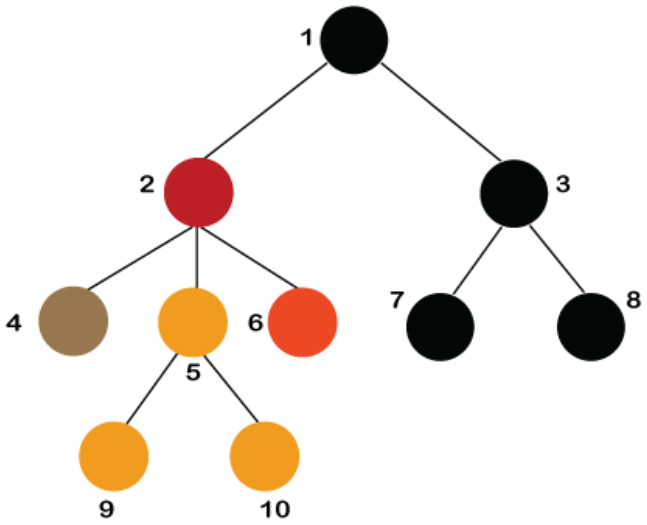
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node: -** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node: -** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Applications of trees

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

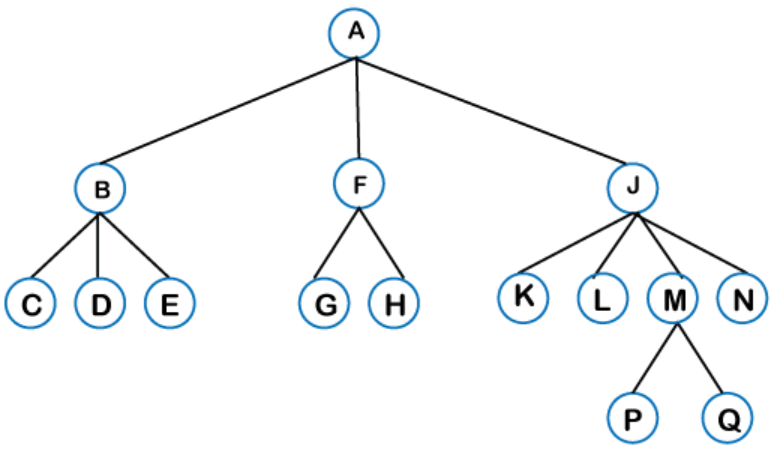
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree
- Red-Black Tree
- Splay Tree
- Treap
- B-Tree

General Tree

The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.

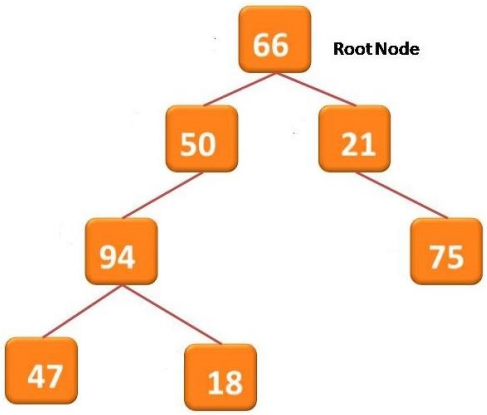


There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.

Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

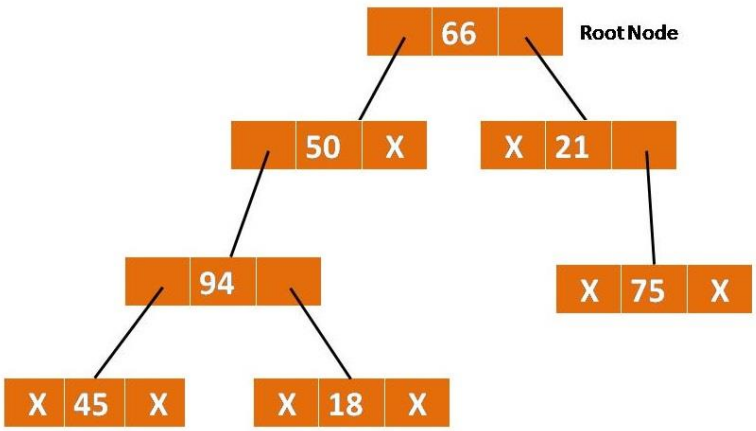
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.



The logical / memory representation of the above tree is:

Linked Representaion:



In the above tree, node 66 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 50 contains only one node (left); therefore, it has one pointer (left) and right ponter is having NULL value. Similary for other nodes.

Array Representaion:

A small and almost complete binary tree can be easily stored in a linear array. Small tree is preferably stored in linear array because searching process in a linear array is expensive. Complete means that if most of the nodes have two child nodes.

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

To store binary tree in a linear array, you need to consider the positional indexes of the nodes. This indexing must be considered starting with '0' from the root node going from left to right as you go down from one level to other.

Assigning of indexes is done in this way:

Remember root node will start at index '0'.

Index of parent= int [index of child node/2]

Index of Left Child = (2 * Index of parent) + 1

Index of Right Child = (2 * Index of parent) + 2

Index	0	1	2	3	4	5	6	7	8	9
Node	66	50	21	94	-	-	75	47	18	

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + ...2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

$h = n-1$

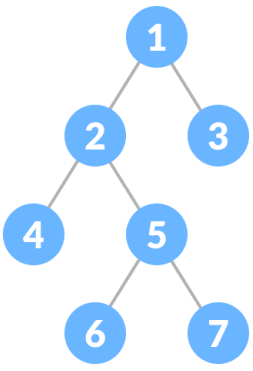
Types of Binary Tree

- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

❖ Full/ Proper/ Strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.

The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

$n = 2*h - 1$

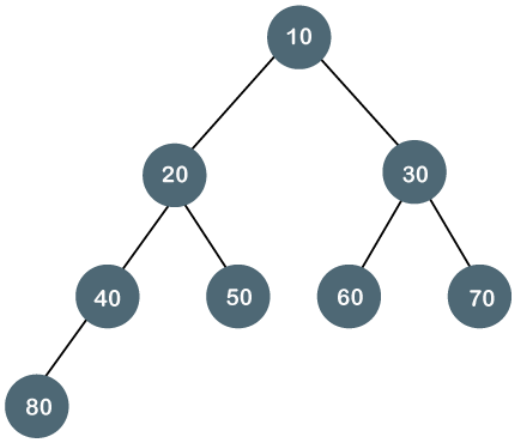
$n+1 = 2*h$

$h = n+1/2$

❖ Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



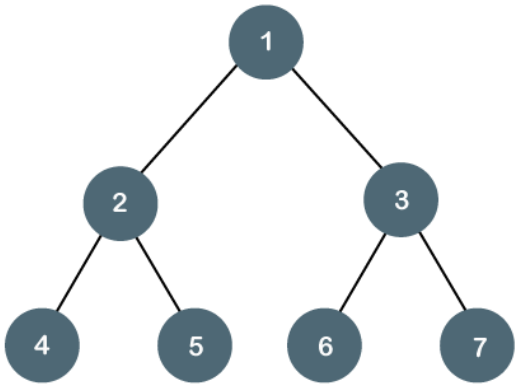
Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is 2^{h+1} - 1.

- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

❖ Perfect Binary Tree

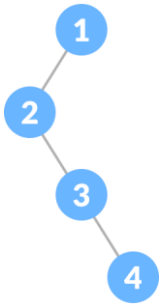
A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

❖ Degenerate Binary Tree

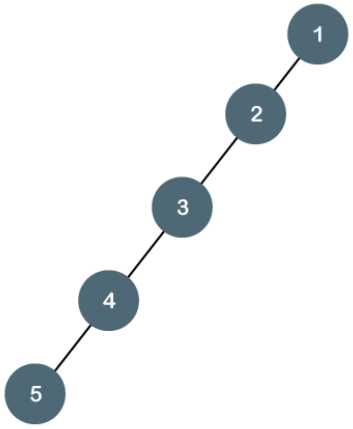
The degenerate binary tree is a tree in which all the internal nodes have only one children.



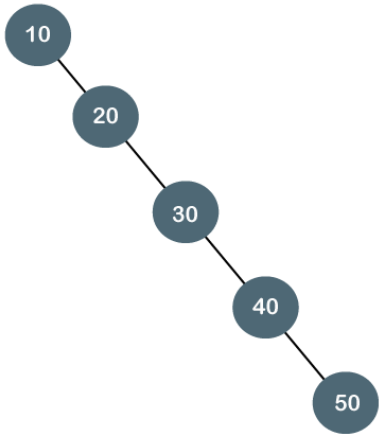
Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.

Thus, there are two types of skewed binary tree**: left-skewed binary tree** and **right-skewed binary tree.**



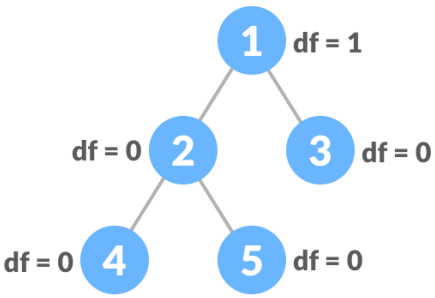
Left Skewed



Right Skewed

❖ **Balanced Binary Tree**

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Tree Traversals

The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- ❖ Preorder traversal
- ❖ Inorder traversal
- ❖ Postorder traversal

❖ Preorder traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

The applications of preorder traversal include -

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

❖ Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

The applications of postorder traversal include –

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

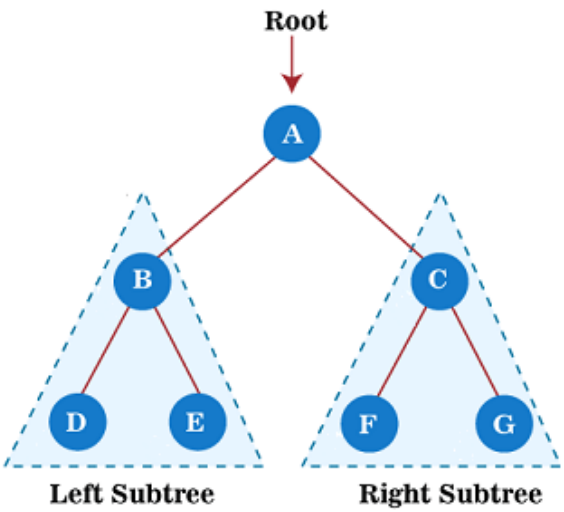
❖ Inorder traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

Example-1:

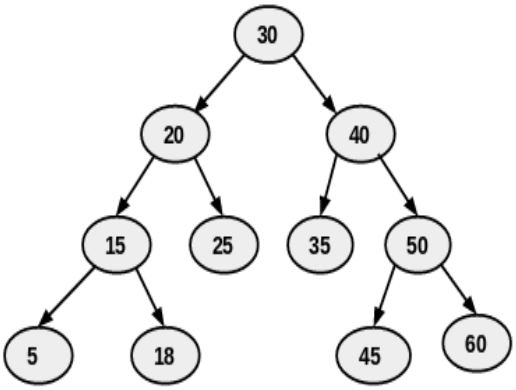


Preorder: A→ B→ D→ E→ C→ F→ G

Postorder: D→ E→ B→ F→ G→ C→ A

Inorder: D→ B→ E→ A→ F→ C→ G

Example-2



Preorder: 30→ 20→ 15→ 5→ 18→ 25→ 40→ 35→ 50→ 45→ 60

Postorder: 5→ 18→ 15→ 25→ 20→ 35→ 45→ 60→ 50→ 40→ 30

Inorder: 5→ 15→ 18→ 20→ 25→ 30→ 35→ 40→ 45→ 50→ 60

Implementation:

```
// Binary Tree in C++

#include <stdlib.h>
#include <iostream>
using namespace std;

struct node {
  int data;
  struct node *left;
  struct node *right;
};

// New node creation
struct node *newNode(int data) {
  struct node *node = (struct node *)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return (node);
}
```

```
// Traverse Preorder
void traversePreOrder(struct node *temp) {
  if (temp != NULL) {
    cout << " " << temp->data;
    traversePreOrder(temp->left);
    traversePreOrder(temp->right);
  }
}

// Traverse Inorder
void traverseInOrder(struct node *temp) {
  if (temp != NULL) {
    traverseInOrder(temp->left);
    cout << " " << temp->data;
    traverseInOrder(temp->right);
  }
}

// Traverse Postorder
void traversePostOrder(struct node *temp) {
  if (temp != NULL) {
    traversePostOrder(temp->left);
    traversePostOrder(temp->right);
    cout << " " << temp->data;
  }
}

int main() {
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);

  cout << "preorder traversal: ";
  traversePreOrder(root);
  cout << "\nInorder traversal: ";
  traverseInOrder(root);
  cout << "\nPostorder traversal: ";
  traversePostOrder(root);
}
```

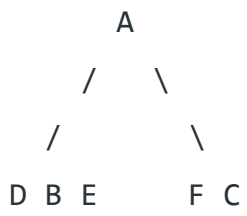
Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

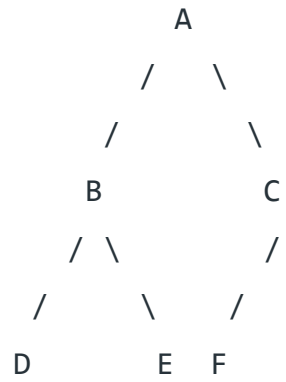
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, the leftmost element is the root of the tree. So we know 'A' is the root for given sequences. By searching 'A' in the Inorder sequence, we can find out all elements on the left side of 'A' is in the left subtree and elements on right in the right subtree. So we know the below structure now.



We recursively follow the above steps and get the following tree.



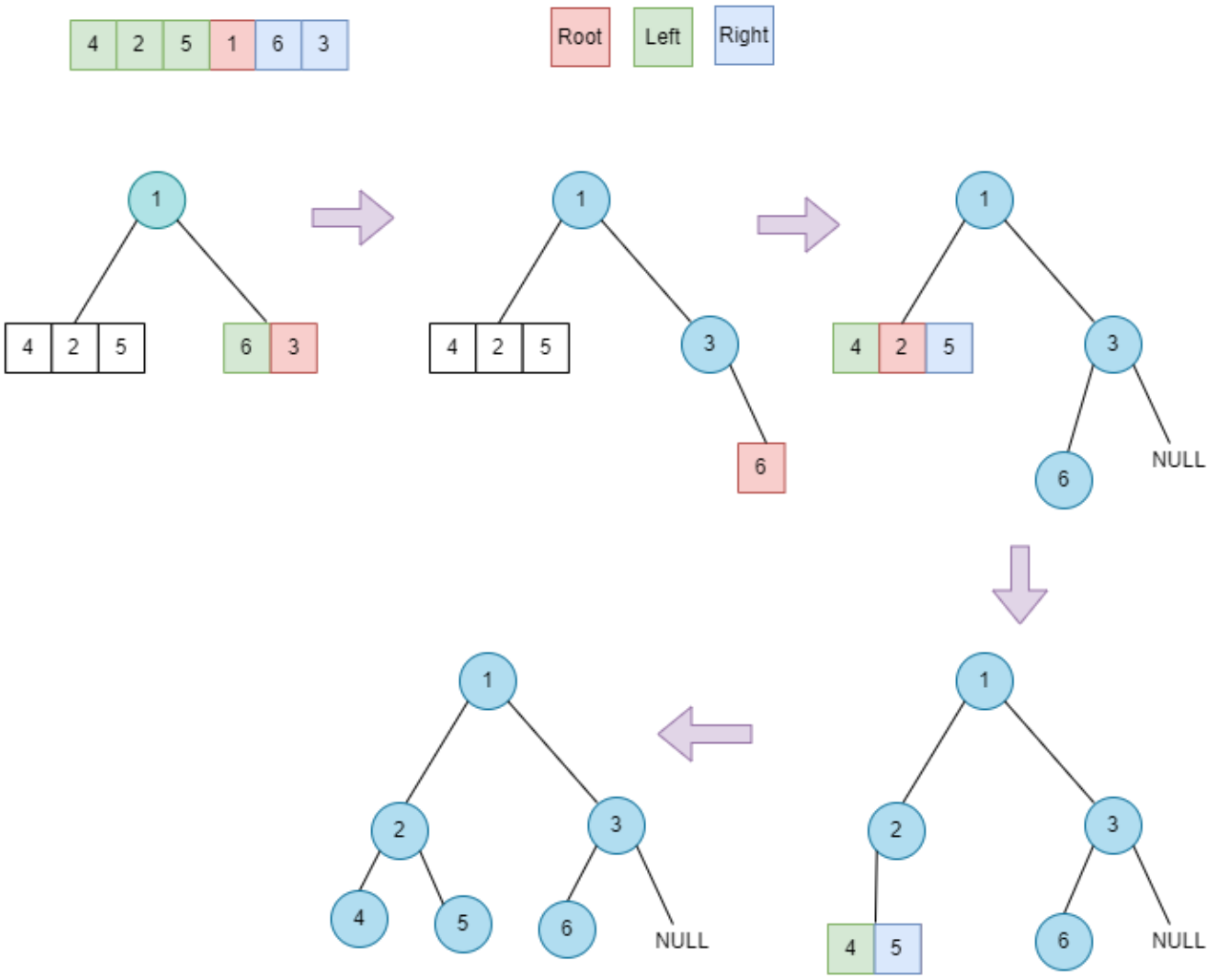
Algorithm: buildTree()

1. Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick the next element in the next recursive call.
2. Create a new tree node tNode with the data as the picked element.
3. Find the picked element's index in Inorder. Let the index be inIndex.
4. Call buildTree for elements before inIndex and make the built tree as a left subtree of tNode.
5. Call buildTree for elements after inIndex and make the built tree as a right subtree of tNode.
6. return tNode.

Construct a Binary Tree from a given Postorder and Inorder Traversal

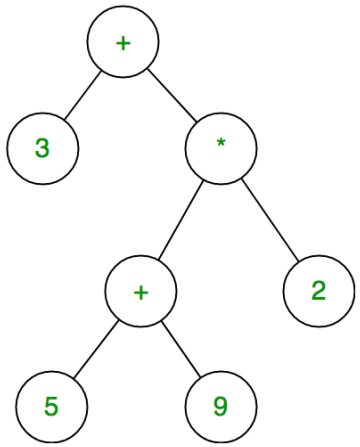
Post-order Sequence : 4 5 2 6 3 1

In-order Sequence : 4 2 5 1 6 3



Expression tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for **3 + ((5+9)*2)** would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)

Evaluating the expression represented by an expression tree:

```
Let t be the expression tree
If  t is not null then
      If t.value is operand then
                Return  t.value
      A = solve(t.left)
      B = solve(t.right)

      // calculate applies operator 't.value'
      // on A and B, and returns value
      Return calculate(A, B, t.value)
```

Construction of Expression Tree:

For constructing an expression tree we use a Stack. We loop through input expression and do the following for every character.

- If a character is an operand push that into the stack
- If a character is an operator pop two values from the Stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

// C++ Program to construct Expression Tree

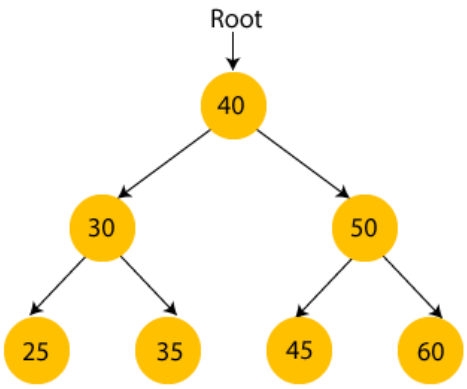
```
#include iostream
using namespace std;
// Tree Node Definition
class TreeNode
{
public:
    char val;
    TreeNode *left, *right;
    TrreeNode()
    {
        this->left = NULL;
        this->right = NULL;
    }
    // Constructor Method
    TreeNode(char val)
    {
        this->val = val;
        this->left = NULL;
        this->right = NULL;
    }
};
// Stack to hold the latest node
class Stack
{
public:
    TreeNode *treeNode;
    Stack *next;
    // Constructor Method
    Stack(TreeNode *treeNode)
    {
        this->treeNode = treeNode;
        next = NULL;
    }
};
class ExpressionTree
{
private:
    Stack *top;
public:
    // Constructor Method
    ExpressionTree()
```

```
{
    top = NULL;
}
// function to push a node in stack
void push(TreeNode *ptr)
{
    if (top == NULL)
        top = new Stack(ptr);
    else
    {
        Stack *nptr = new Stack(ptr);
        nptr->next = top;
        top = nptr;
    }
}
TreeNode *pop()
{
    TreeNode *ptr = top->treeNode;
    top = top->next;
    return ptr;
}
TreeNode *peek()
{
    return top->treeNode;
}
// function to insert character
void insert(char val)
{
    // If the encountered character is Number make a node an push it on stack
    if (isOperand(val))
    {
        TreeNode *nptr = new TreeNode(val);
        push(nptr);
    }
    // else if it is operator then make a node and left and
    else if (isOperator(val))
    {
        TreeNode *nptr = new TreeNode(val);
        nptr->left = pop();
        nptr->right = pop();
        push(nptr);
    }
}
// function to check if operand
bool isOperand(char ch)
{
    return ch >= '0' && ch <= '9' || ch>='A' && ch<='Z' || ch>='a' && ch<='z';
}
// function to check if operator
bool isOperator(char ch)
{
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}
// function to construct expression Tree
void construct(string eqn)
{
```

```
        for (int i = eqn.length() - 1; i >= 0; i--)
            insert(eqn[i]);
    }
    void inOrder(TreeNode *ptr)
    {
        if (ptr != NULL)
        {
            inOrder(ptr->left);
            cout<<ptr->val;
            inOrder(ptr->right);
        }
    }
};
int main()
{
    string exp;
    ExpressionTree et;
    cout<<"Enter expression in Prefix form: ";
    cin>>exp;
    et.construct(exp);
    cout<<"In-order Traversal of Expression Tree : ";
    et.inOrder(et.peek());
    return 0;
};
```

Binary Search tree

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.



Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

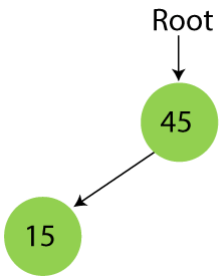
Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Step 1 - Insert 45.

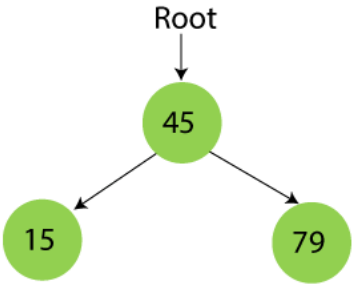
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



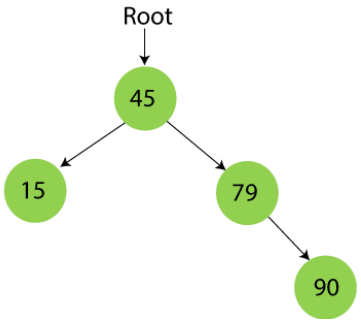
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



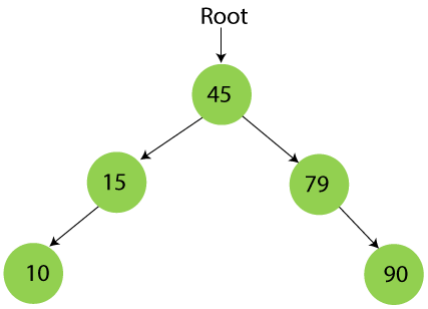
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



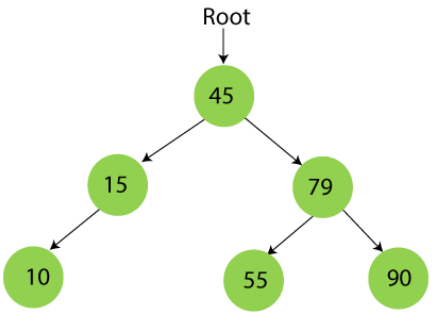
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



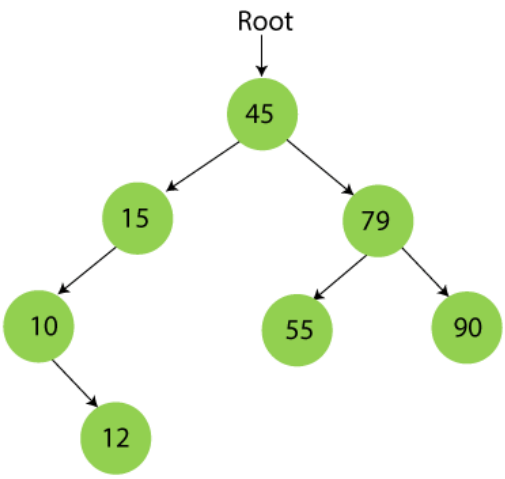
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



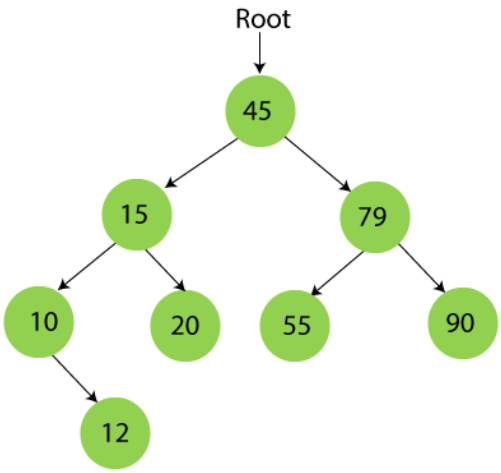
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



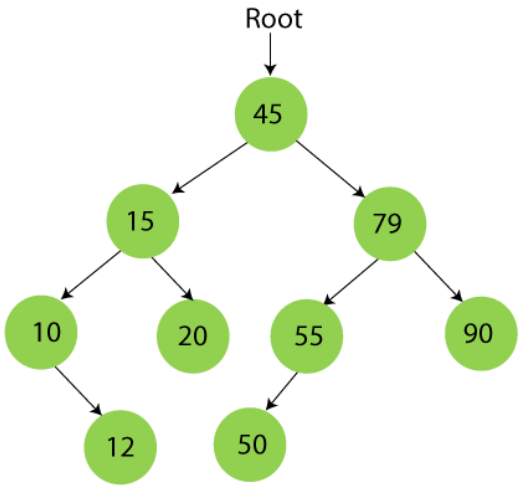
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed.

Algorithm to search an element in Binary search tree

```
Search (root, item)
Step 1 -    if (item = root → data) or (root = NULL)
                return root
            else if (item < root → data)
                return Search(root → left, item)
            else
                return Search(root → right, item)
            END if
Step 2 - END
```

Deletion in Binary Search tree

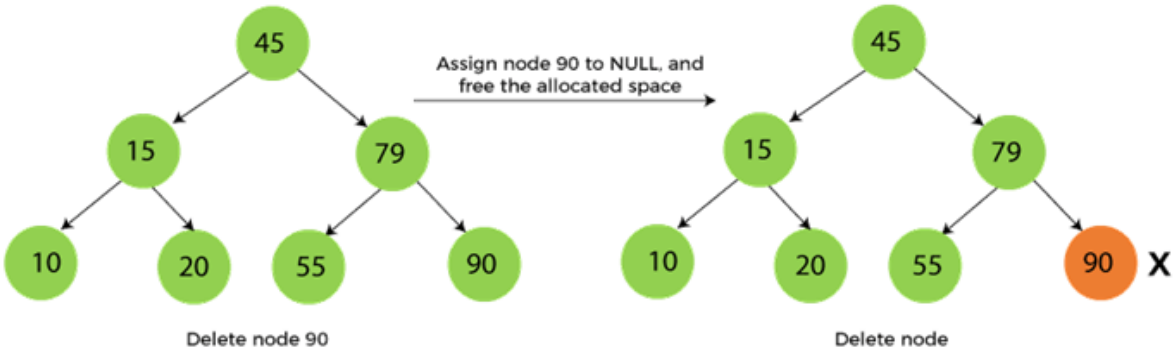
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

When the node to be deleted is the leaf node:

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

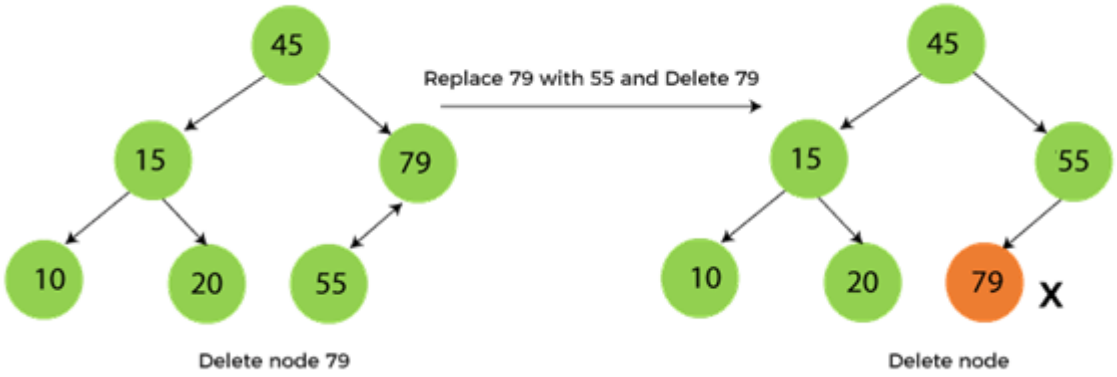


When the node to be deleted has only one child:

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



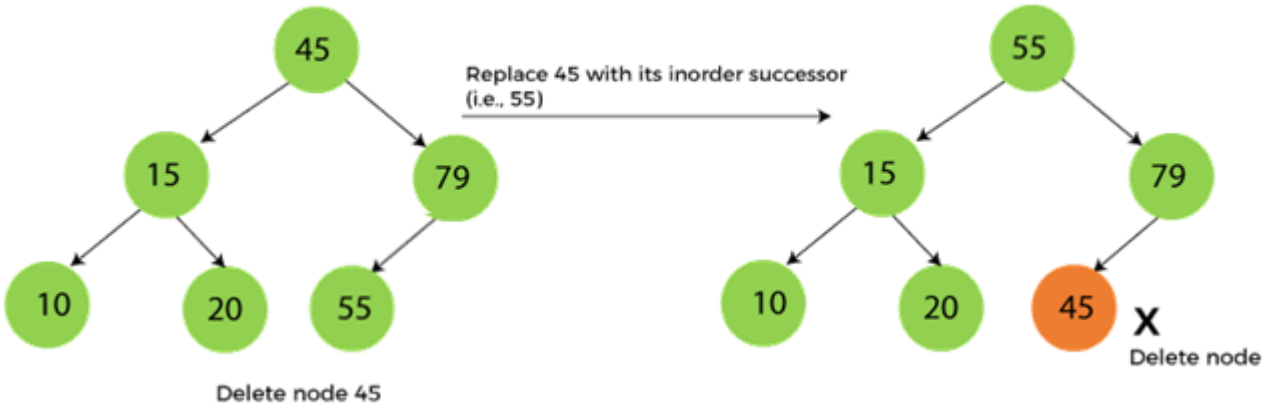
When the node to be deleted has two children:

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



The complexity of the Binary Search tree

Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.

Space Complexity

The space complexity of all operations of Binary search tree is O(n).

```
// Binary Search Tree operations in C++

#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
    inorder(root->right);
```

```
  }
}

// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }
```

```
    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

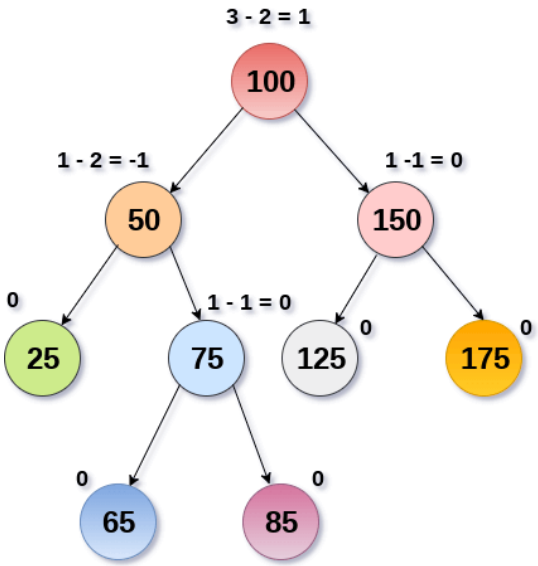
$$\text{Balance Factor (k)} = \text{height (left(k))} - \text{height (right(k))}$$

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Opera-tion	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

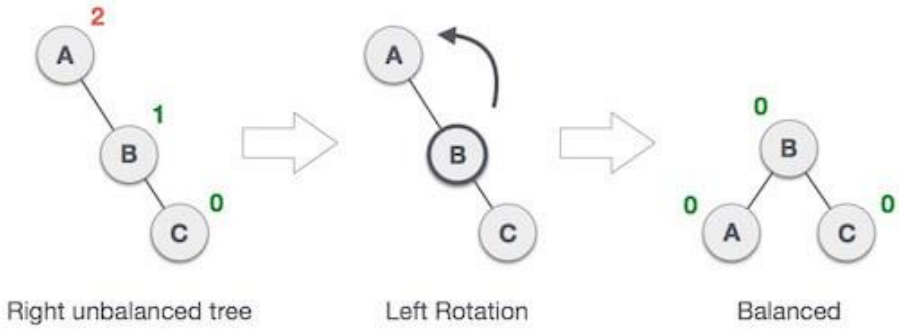
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

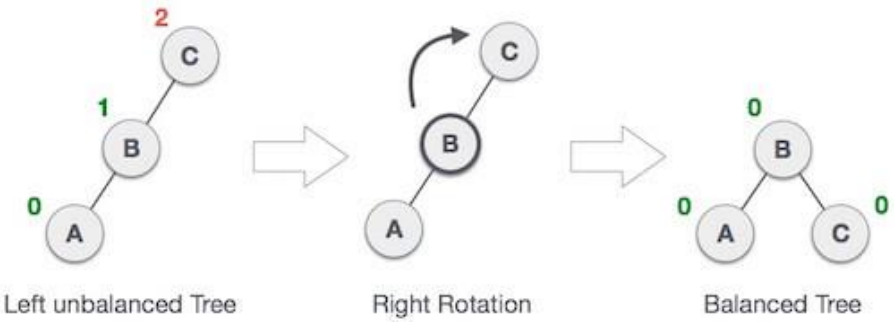
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

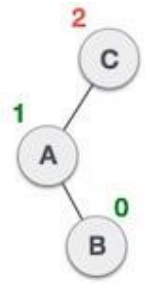
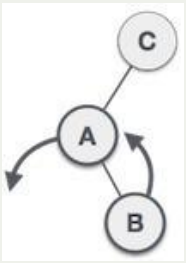
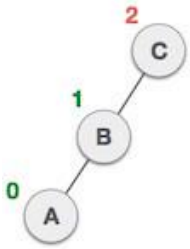
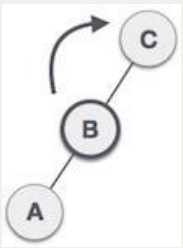
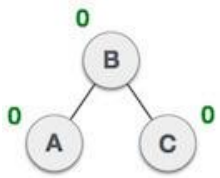
When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

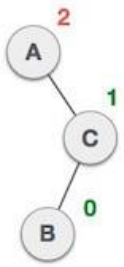
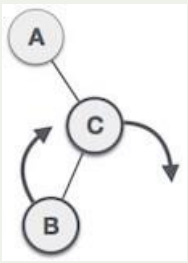
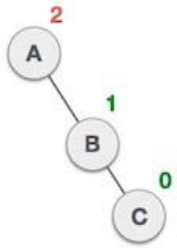
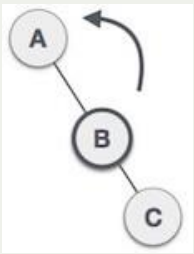
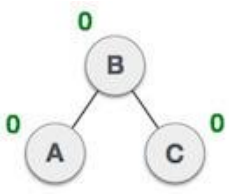
3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

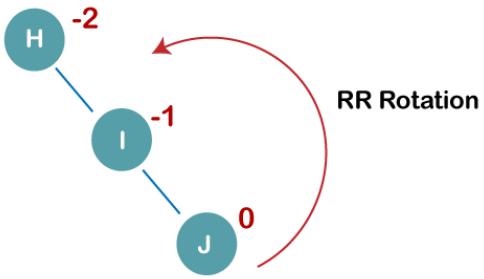
As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Construction of AVL tree having the following elements

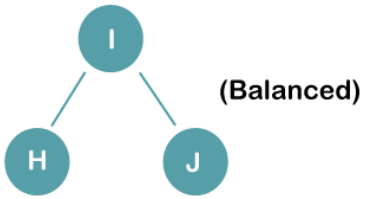
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

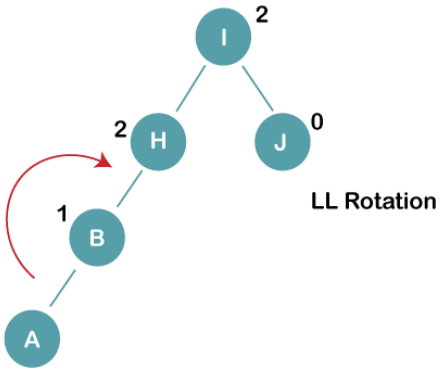


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

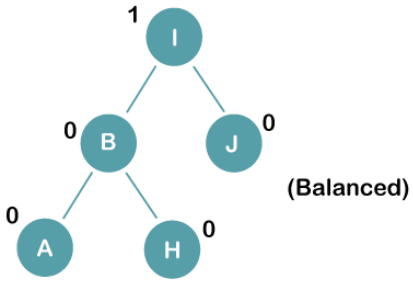


2. Insert B, A

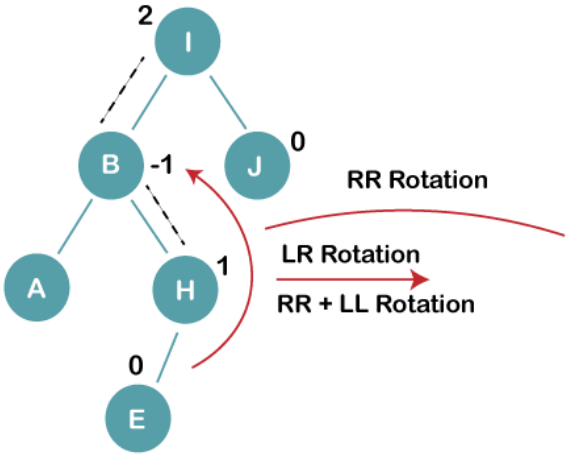


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



3. Insert E

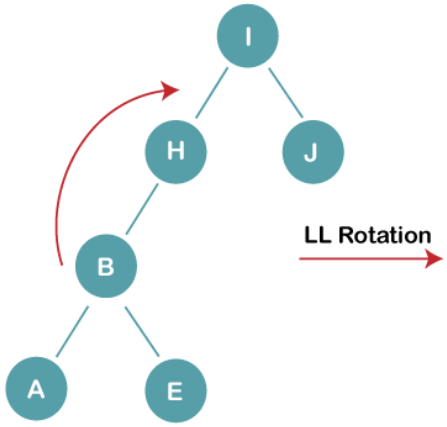


On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I.

LR = RR + LL rotation

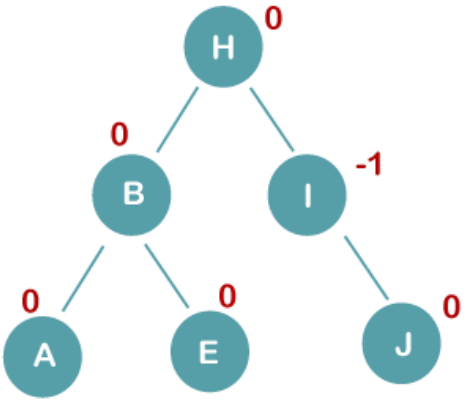
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:



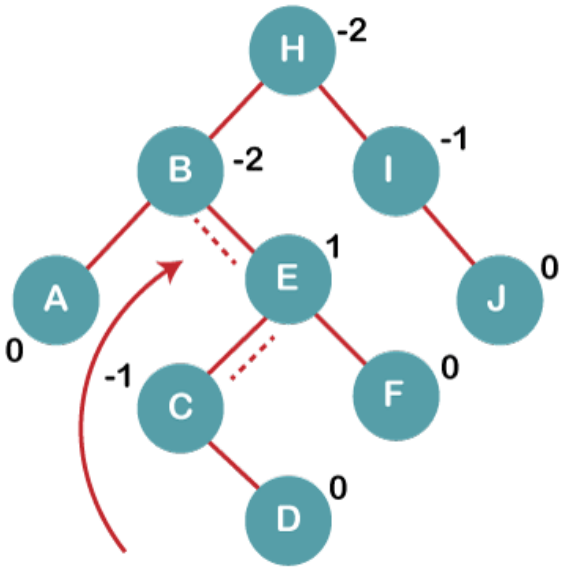
3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



(Balanced)

4. Insert C, F, D

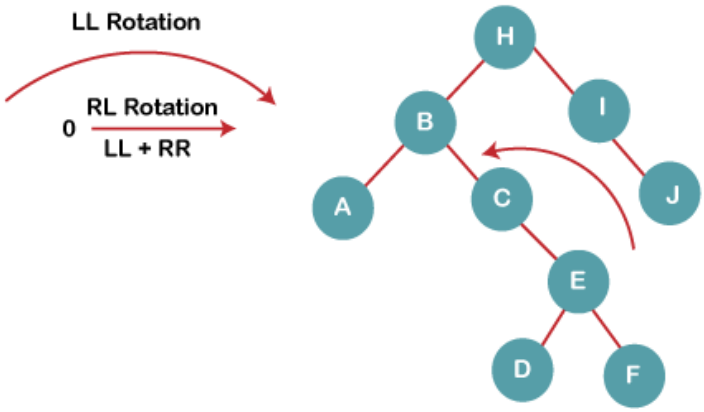


On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I.

RL = LL + RR rotation.

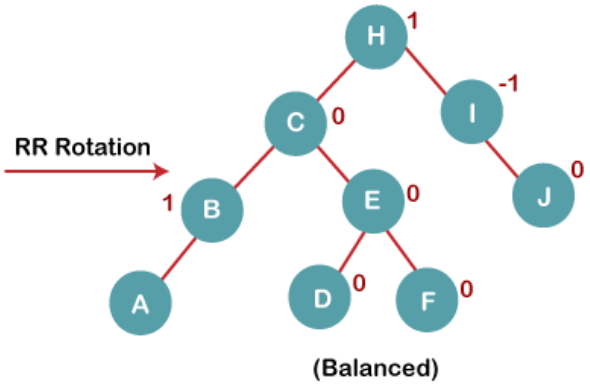
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

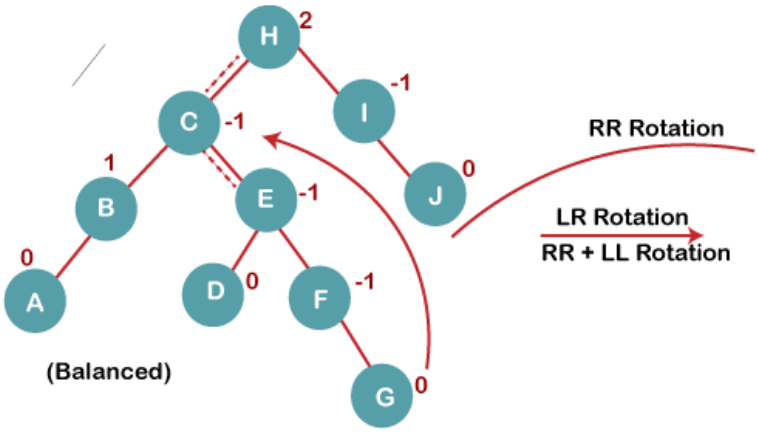


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



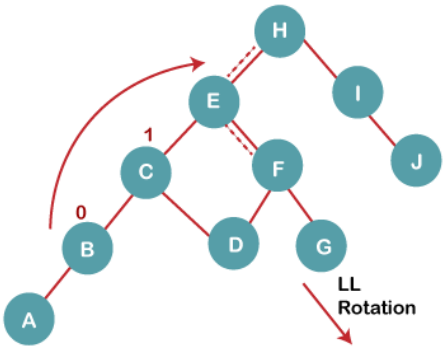
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

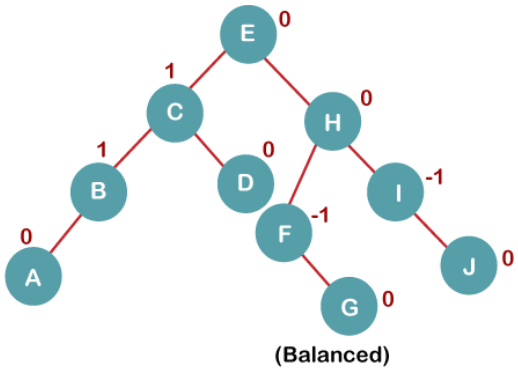
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

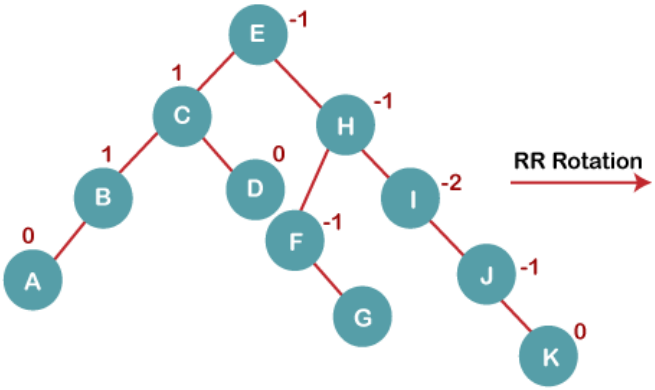


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

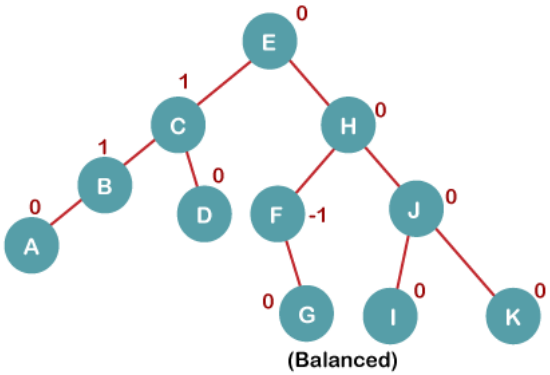


6. Insert K



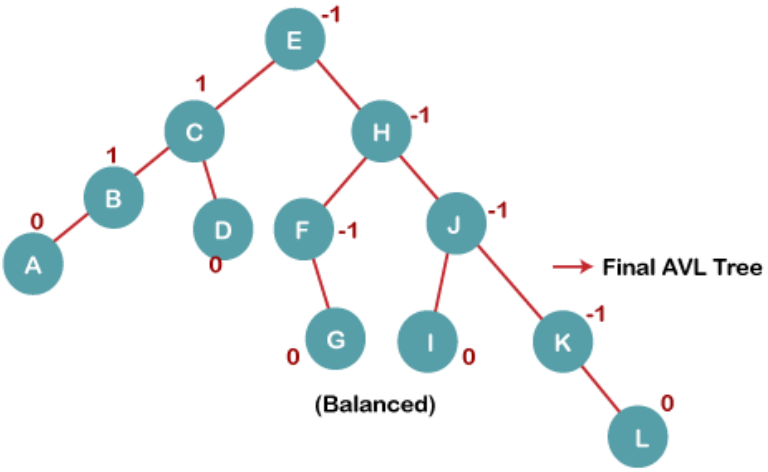
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Red-Black Tree

The Red-Black tree is a binary search tree where each node contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc.

Properties of Red-Black tree

- It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
- This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
- In the Red-Black tree, the root node is always black in color.
- In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
- If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
- Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Insertion in Red Black tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied, then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

1. **Recolor**
2. **Rotation**
3. **Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Let's understand with an example:

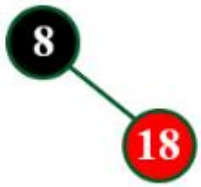
8, 18, 5, 15, 17, 25, 40, 80

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

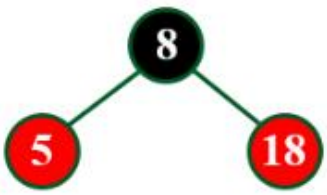
insert (18)

Tree is not Empty. So insert newNode with red color.



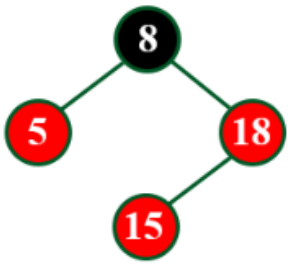
insert (5)

Tree is not Empty. So insert newNode with red color.



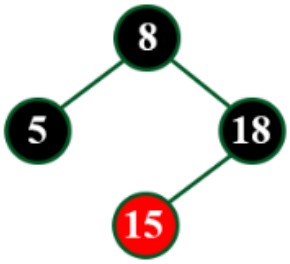
insert (15)

Tree is not Empty. So insert newNode with red color.



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

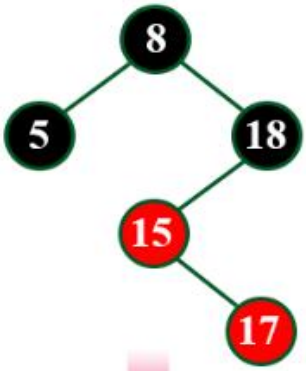
After RECOLOR



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

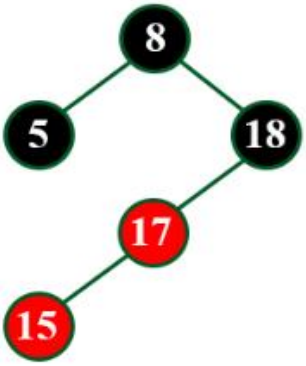
insert (17)

Tree is not Empty. So insert newNode with red color.

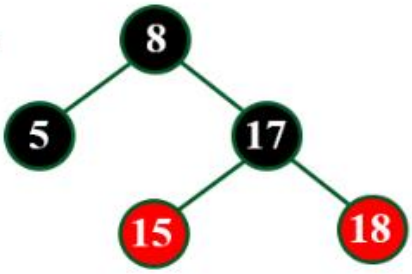


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

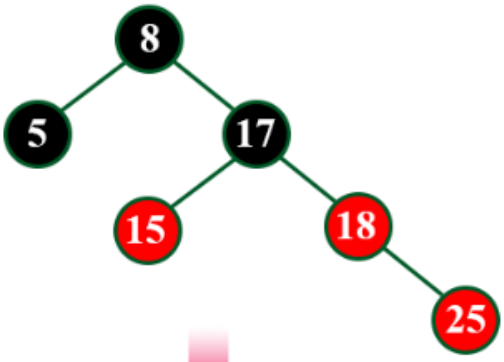


After Right Rotation & Recolor



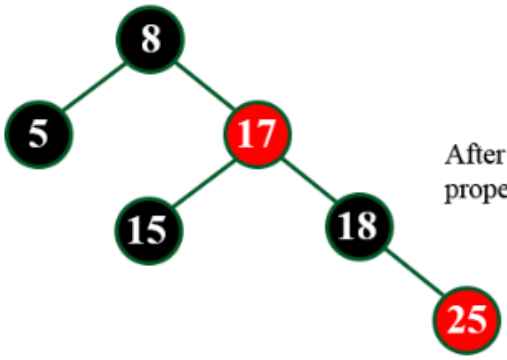
insert (25)

Tree is not Empty. So insert newNode with red color.



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

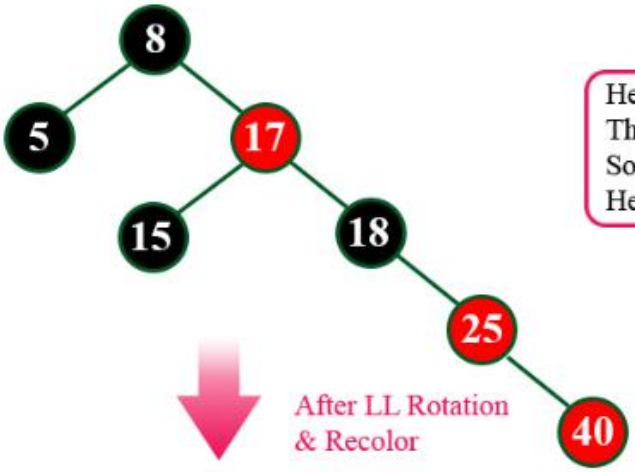
After Recolor



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

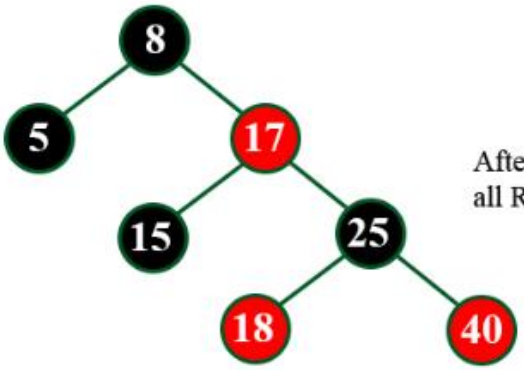
insert (40)

Tree is not Empty. So insert newNode with red color.



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

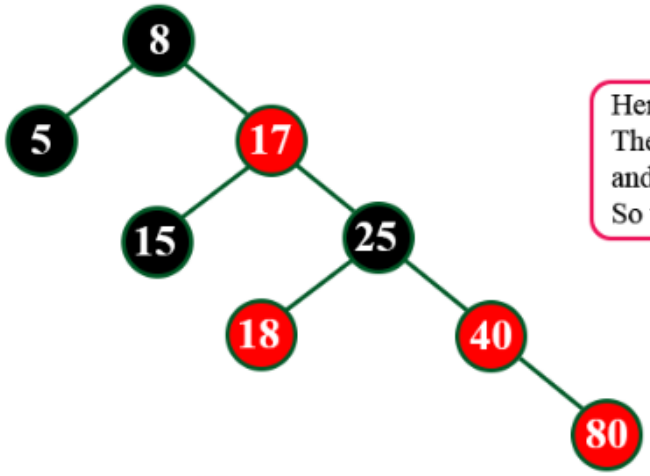
After LL Rotation
& Recolor



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

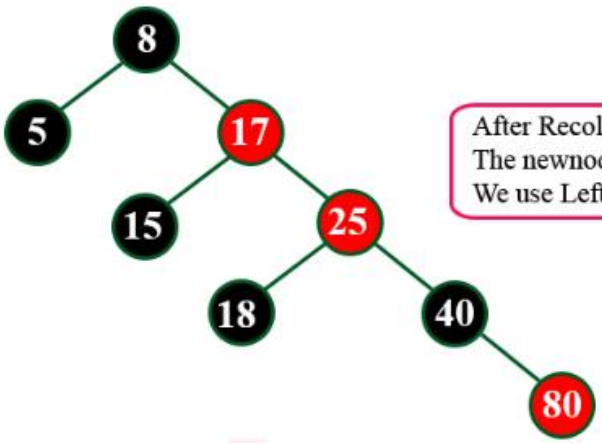
insert (80)

Tree is not Empty. So insert newNode with red color.



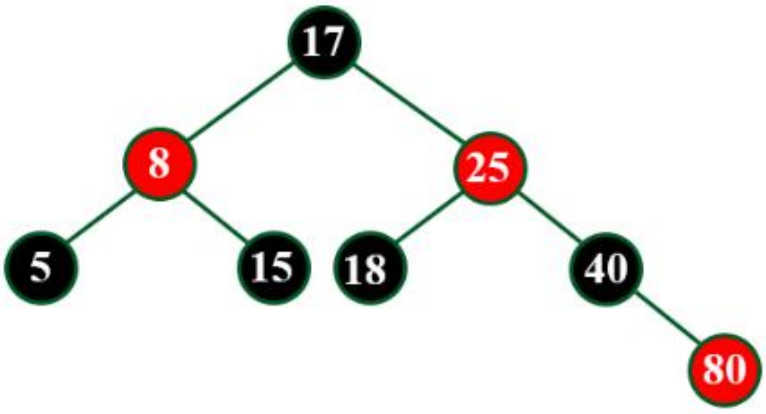
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



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

After Left Rotation
& Recolor



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

Deletion in Red Back tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Now, the question arises that ***why do we require a Red-Black tree*** if AVL is also a height-balanced tree. The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between the AVL tree

and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees O(log2n) time for all operations like insertion, deletion, and searching.

Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.

Is every AVL tree can be a Red-Black tree?

Yes, every AVL tree can be a Red-Black tree if we color each node either by Red or Black color. But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

Splay Tree

Splay Tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree.

A **Splay Tree** is defined as a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

A **Splay Tree** is a self-balancing tree, but **AVL** and **Red-Black** trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is **splaying**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

By splaying elements, we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

Rotations

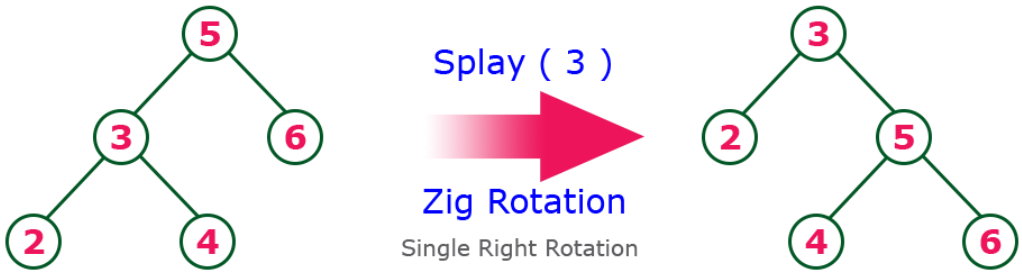
There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...

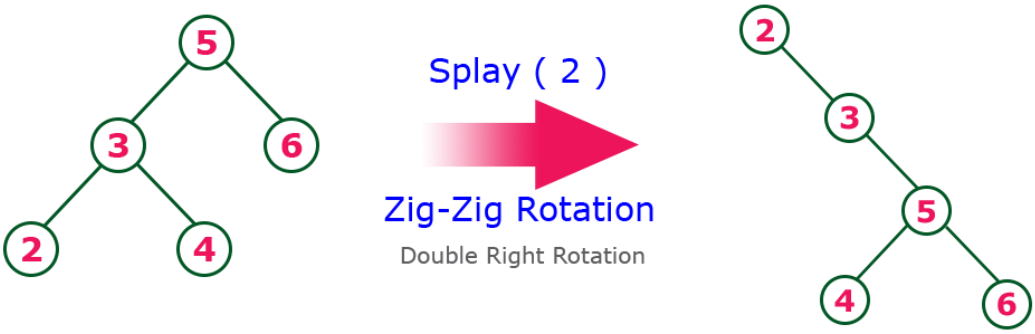


Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

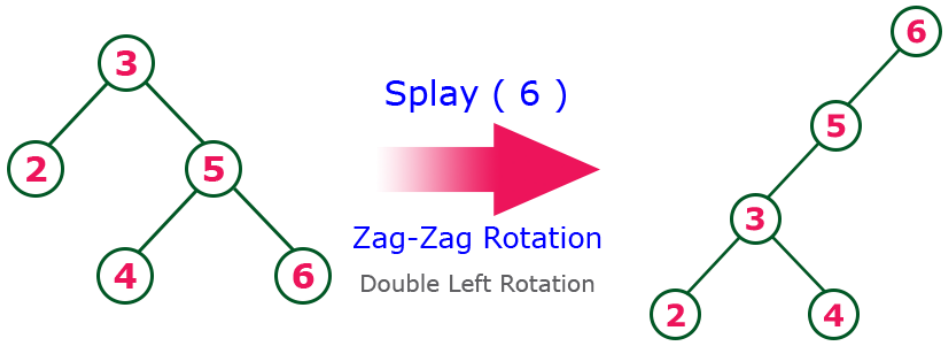
Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



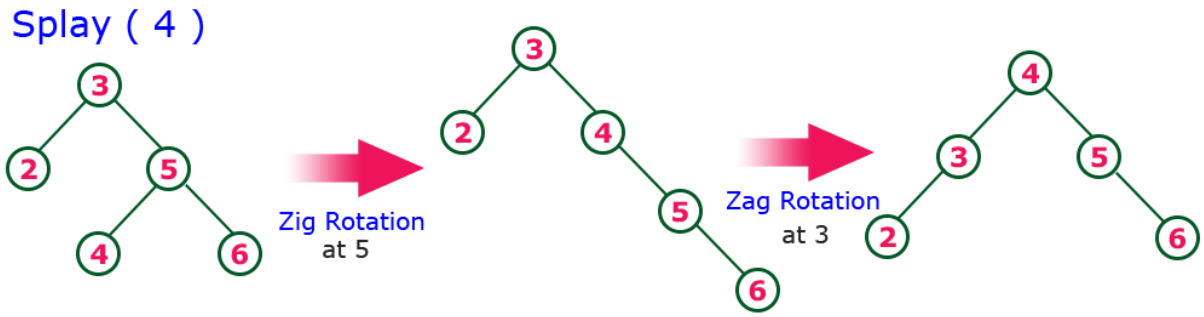
Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



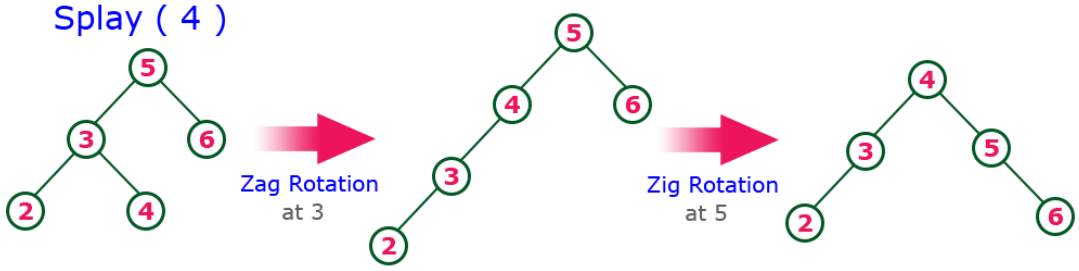
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.

Step 4 - After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

B Tree

B-Tree can be defined as a self-balanced search tree in which every node contains multiple keys ans has more than two children.

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties...

Property #1 - All **leaf nodes** must be **at same level**.

Property #2 - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.

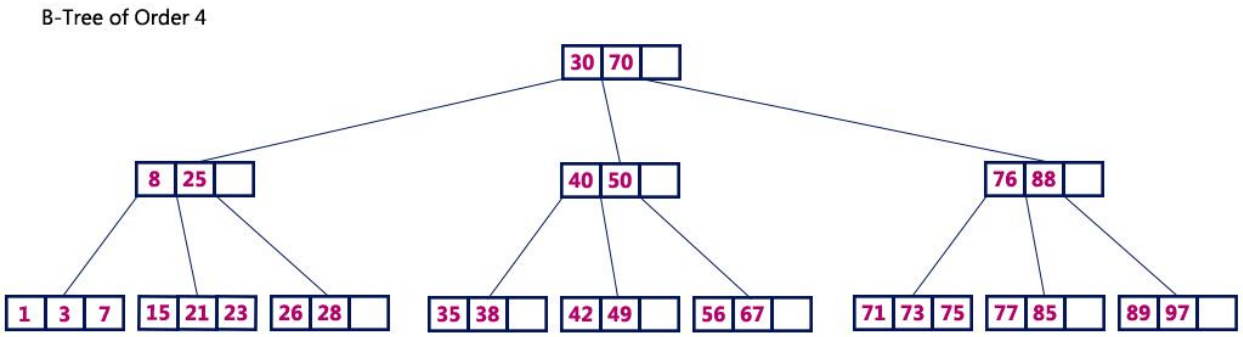
Property #4 - If the root node is a non leaf node, then it must have **atleast 2** children.

Property #5 - A non leaf node with **n-1** keys must have **n** number of children.

Property #6 - All the **key values in a node** must be in **Ascending Order**.

For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.

Example



Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

insert(1)

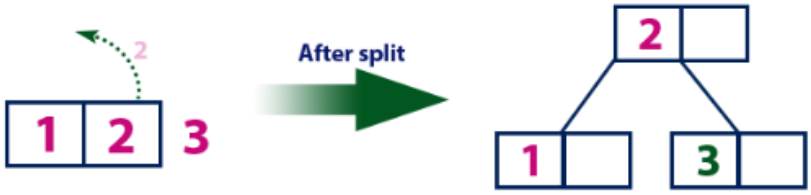
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

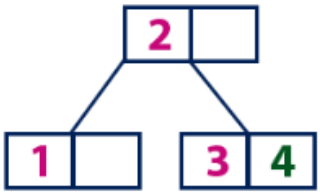
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

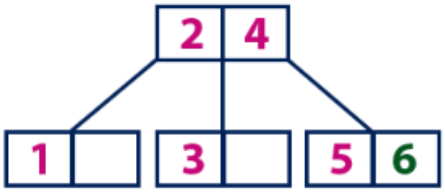


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

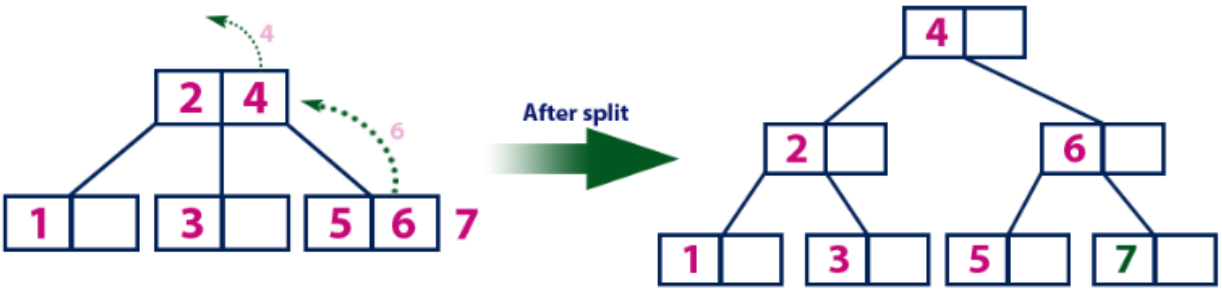
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



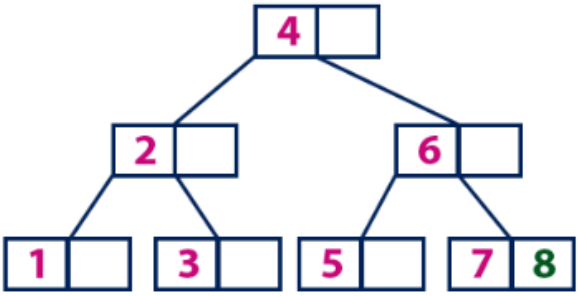
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



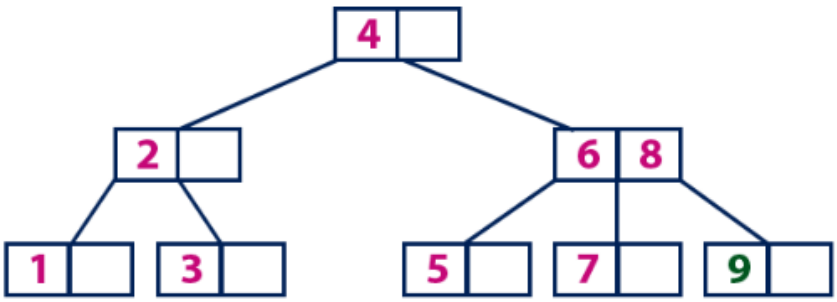
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



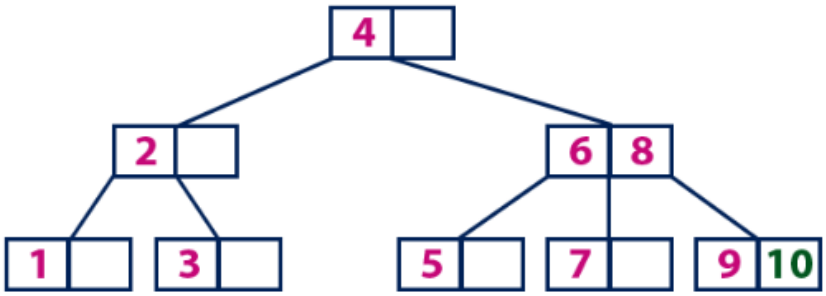
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

B+ Tree

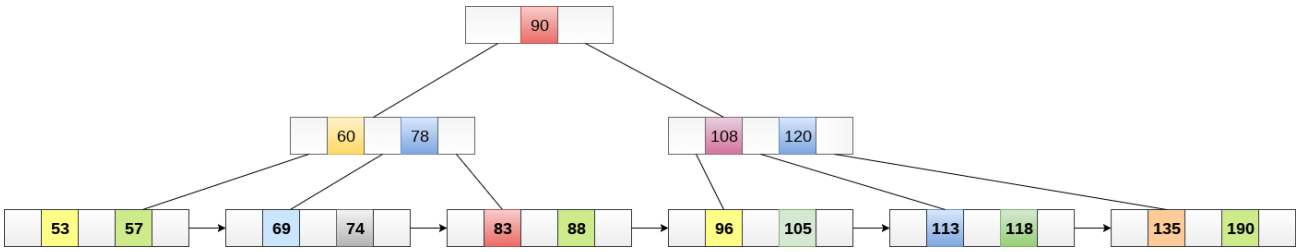
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Priority Queue

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

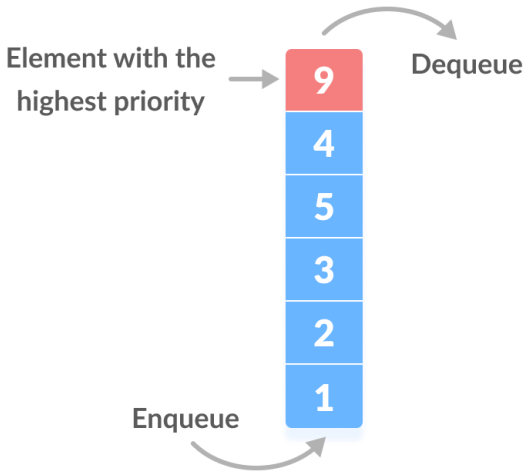
However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Hence, we will be using the heap data structure to implement the priority queue in this tutorial. A max-heap is implemented in the following operations. If you want to learn more about it, please visit max-heap and min-heap.

A comparative analysis of different implementations of priority queue is given below.

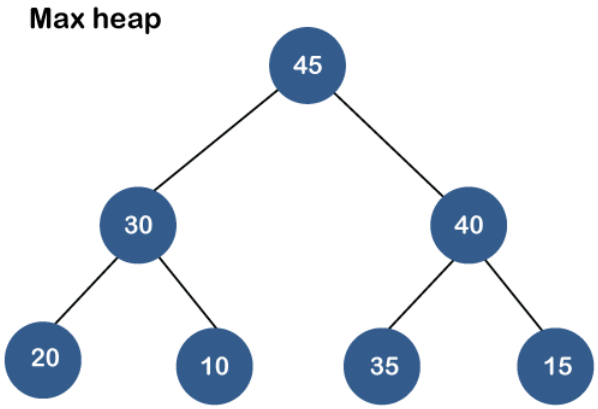
Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)

Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

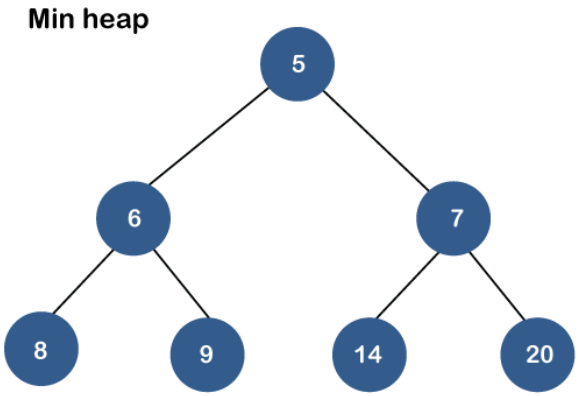
What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

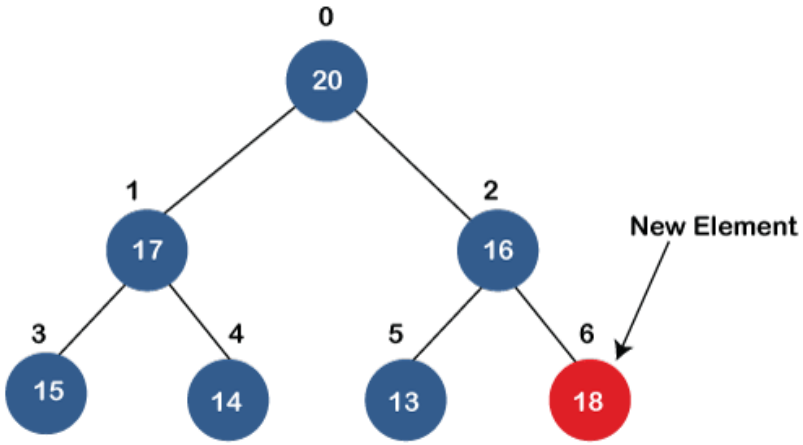
Priority Queue Operations

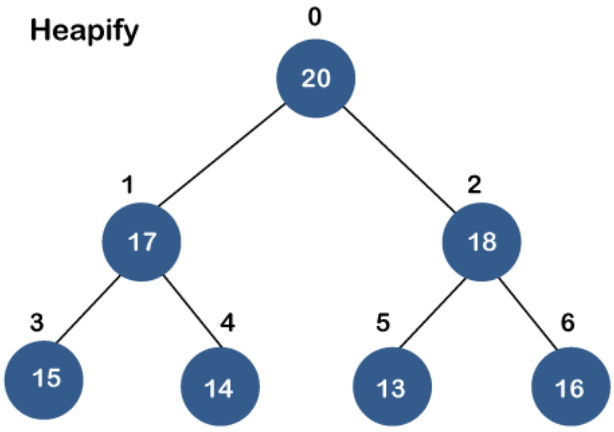
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

Inserting the element in a priority queue (max heap)

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





Removing the minimum element from the priority queue

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Binomial Queue / Binary Heap

The main application of Binary Heap is as implement a priority queue. Binomial Heap is an extension of Binary Heap that provides faster union or merge operation with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

What is a Binomial Tree?

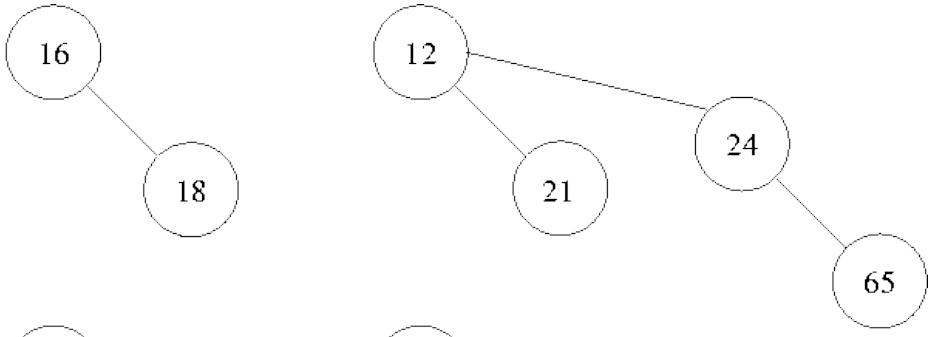
A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other.

A Binomial Tree of order k the has following properties.

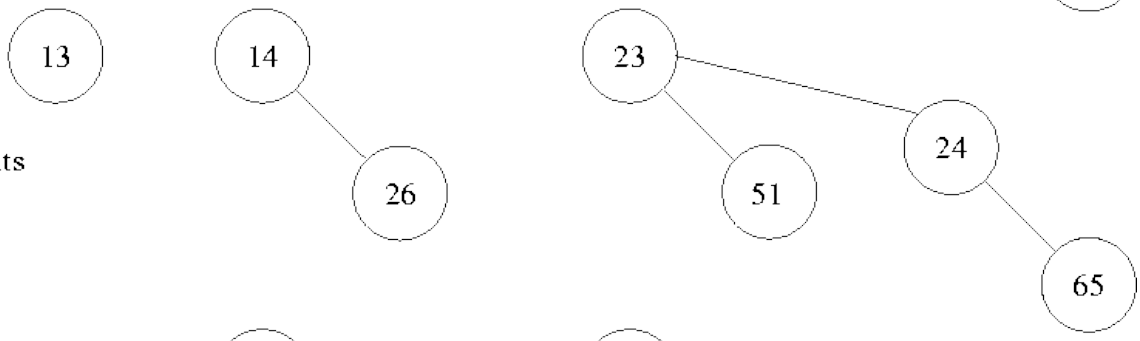
- It has exactly 2^k nodes.
- It has depth as k.
- There are exactly kaiCi nodes at depth i for i = 0, 1, . . . , k.
- The root has degree k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

Merge

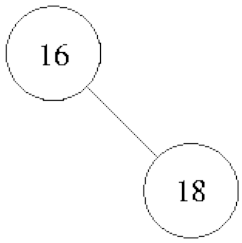
H_1 : 6 elements



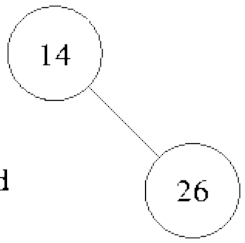
H_2 : 7 elements



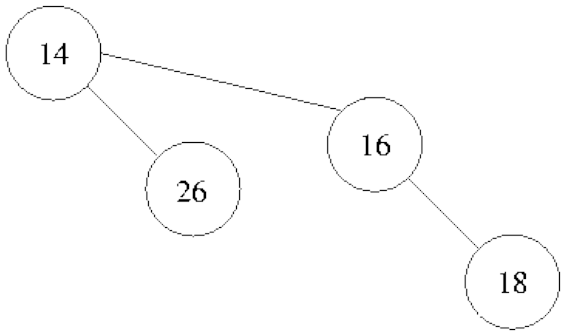
Merge of



and



yields

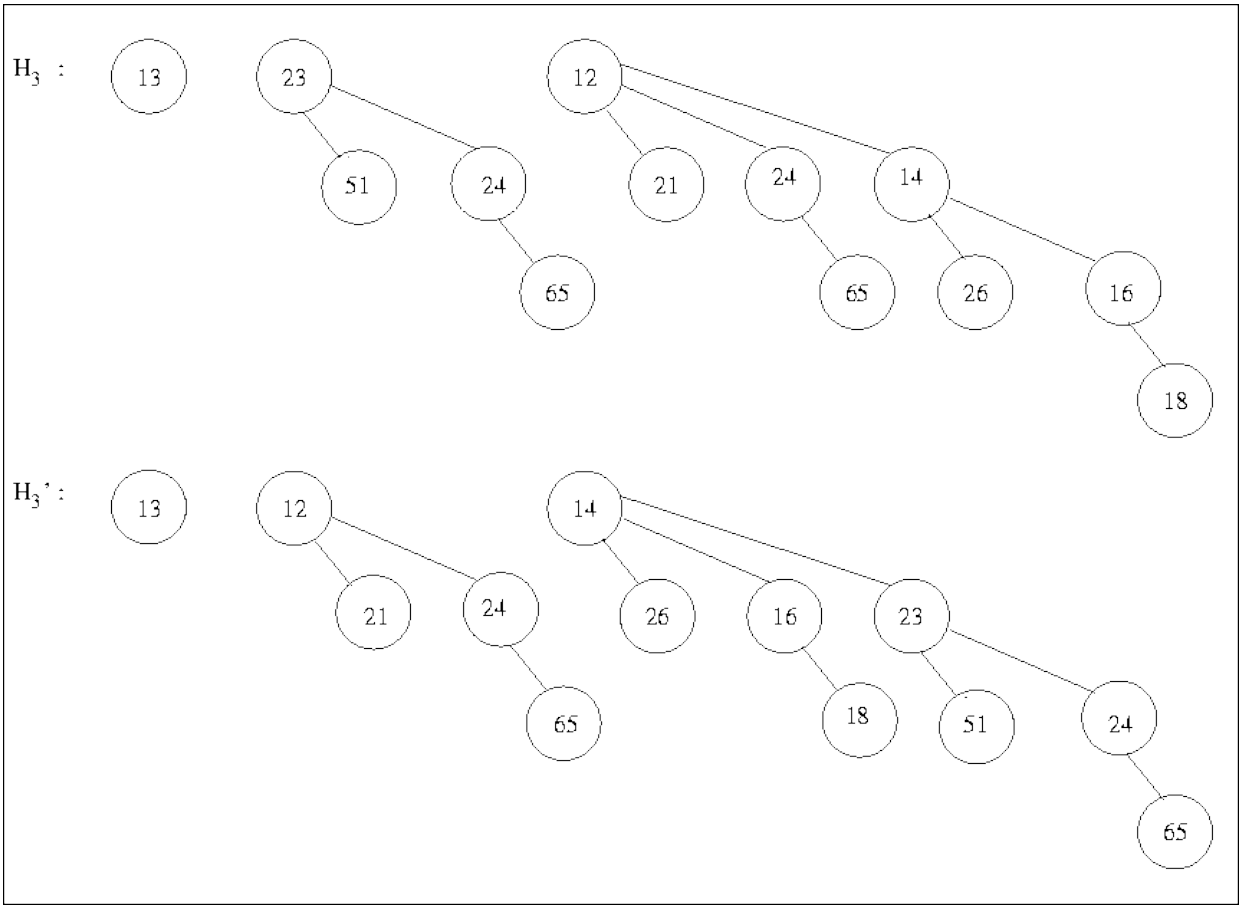


We are now left with

1 tree of height 0

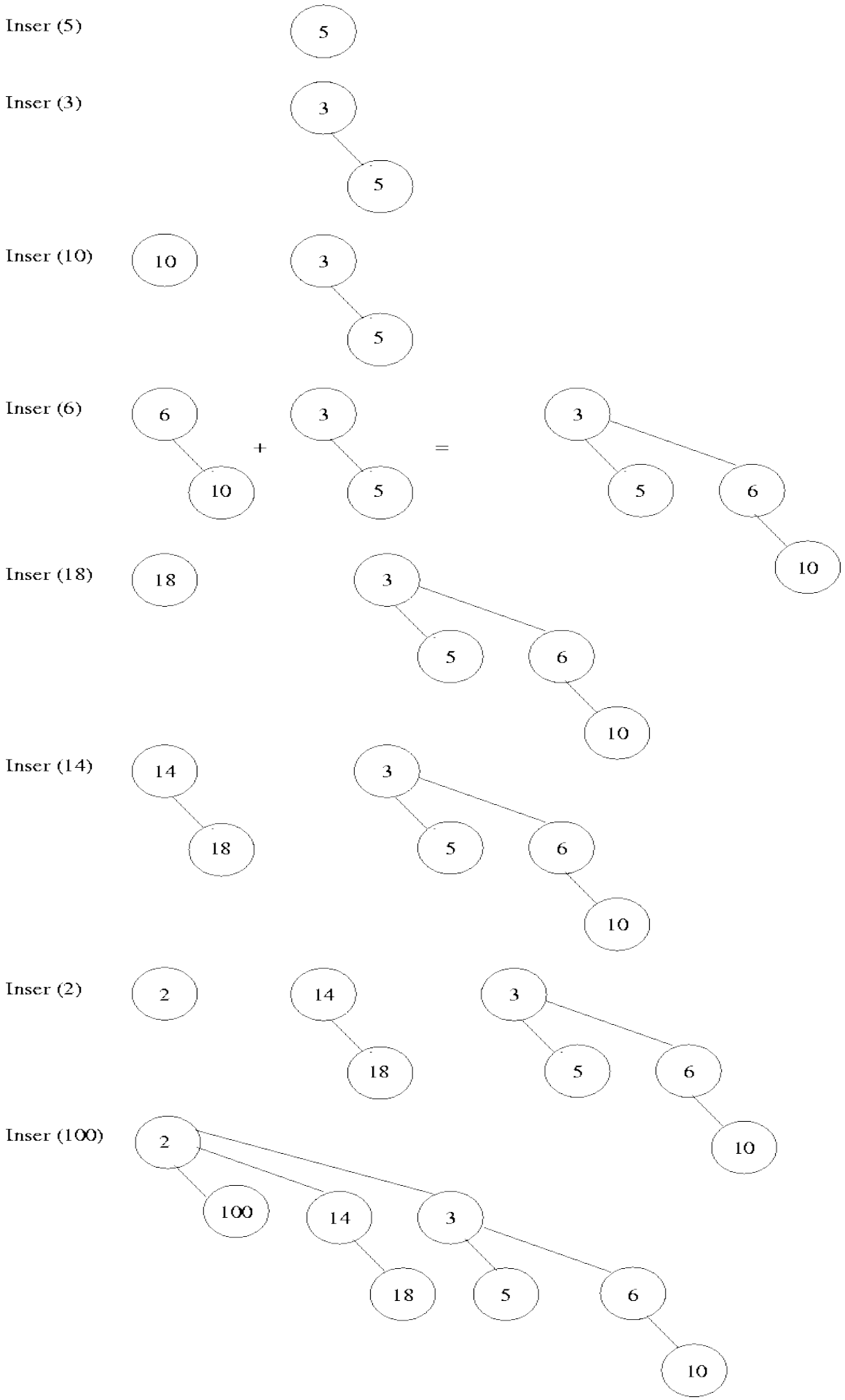
3 trees of height 2

Figure 6.8: Merge of H1 and H2



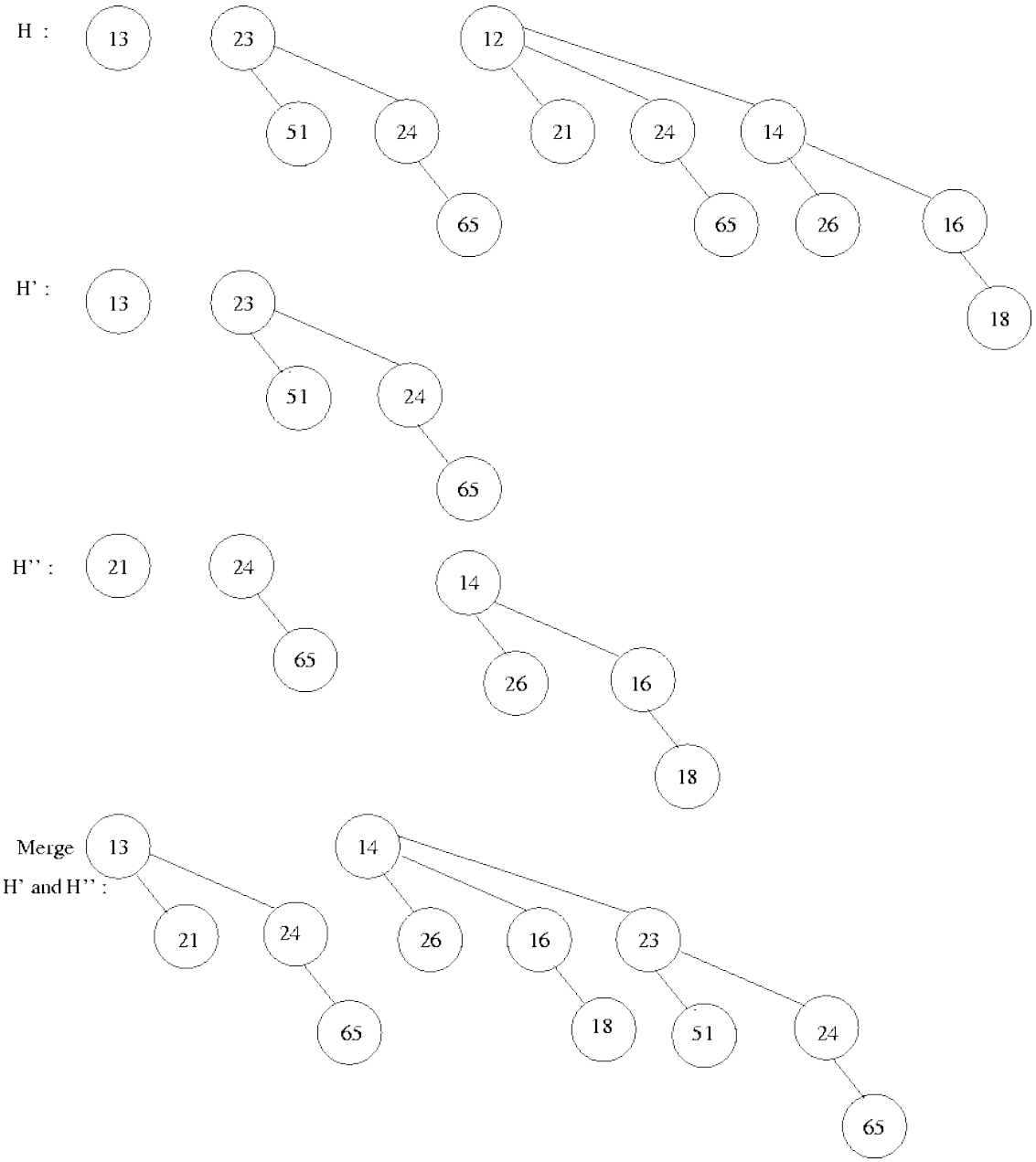
Insertion

- This is a special case of merging since we merely create a one-node tree and perform a merge.
- Worstcase complexity therefore will be O(log n).
- More precisely; if the priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is Bi, the running time is proportional to i + 1.
- Eg: Insertion into H3 (which has 13 elements) terminates in two steps.
- Since each tree of a particular degree in a binomial queue is present with probability $\frac{1}{2}$, if we define the random variable X as representing the number of steps in an insert operation, then


Inser (5)
5
Inser (3)
3
5
Inser (10)
10
3
5
Inser (6)
6
10
+
3
5
=
3
5
6
10
Inser (18)
18
3
5
6
10
Inser (14)
14
18
3
5
6
10
Inser (2)
2
14
18
3
5
6
10
Inser (100)
2
100
14
18
3
5
6
10

Deletemin

- First find the binomial tree with the smallest root. Let this be Bk. Let H be the original priority queue.
- Remove Bk from the forest in H forming another binomial queue H'.
- Now remove the root of Bk creating binomial trees B0, B1, ..., Bk - 1, which collectively form a binomial queue H''.
- Now, merge H' and H''.



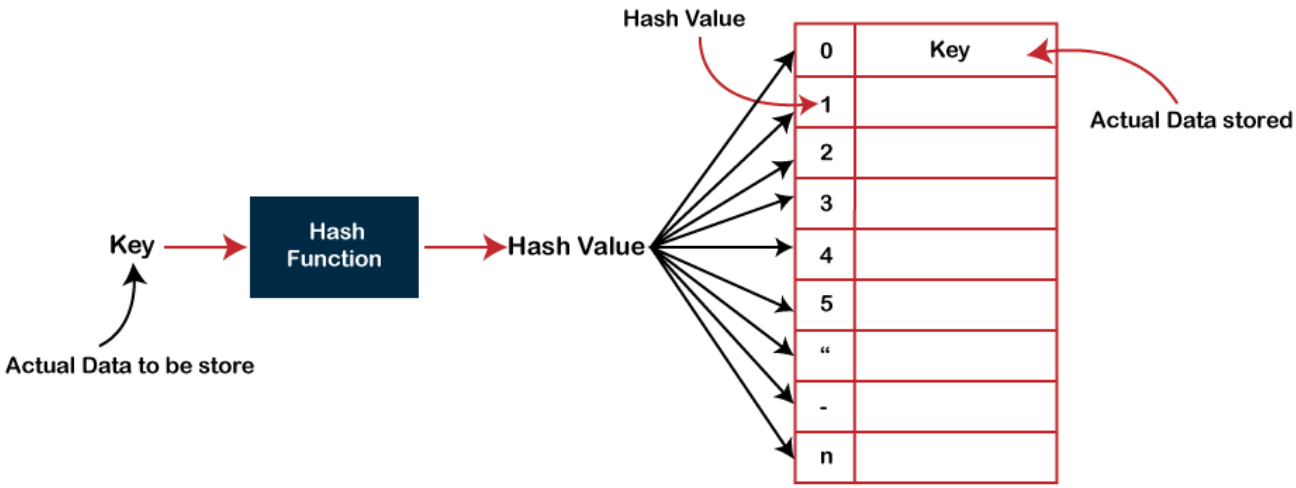
Hashing

Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored.



There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

In the division method, the hash function can be defined as:

$h(k_i) = k_i \% m;$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

h(6) = 6%10 = 6

The index is 6 at which the value is stored.

Collision

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

h(26) = 26%10 = 6

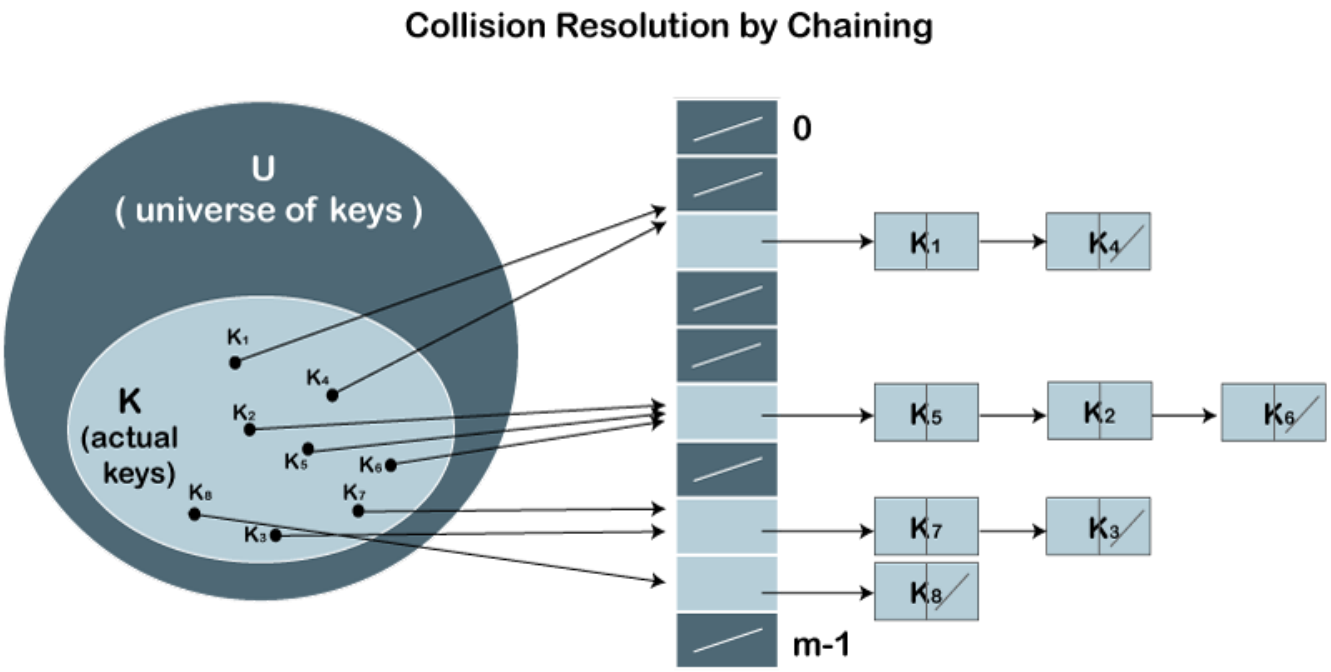
Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.



Perfect Hashing

Perfect hashing is a technique for storing records in a hash table in a way that guarentees no collisions. Perfect hashing sort of turns the concept of hashing on its head, in that it requires that the full set of keys to be stored be available in advance, and a hash function is then generated for that key set. Besides guarenteeing no collisions, perfect hashing techniques can store n records in a table with only n slots.

Some application of perfect hashing includes:

- data storage on a CD ROM
- set of reserved words in a programming language

Cuckoo Hashing

Cuckoo Hashing is a technique for implementing a hash table. As opposed to most other hash tables, it achieves constant time worst-case complexity for lookups.

Collisions are handled by evicting existing keys and moving them from one array to the other. This resembles the way a cuckoo chick pushes out an egg from the nest to make room for itself, hence the name Cuckoo Hashing.

There are three basic operations that must be supported by a hash table (or a dictionary):

- **Lookup(key):** return true if key is there on the table, else false
- **Insert(key):** add the item 'key' to the table if not already present
- **Delete(key):** removes 'key' from the table

To close the gap of expected time and worst case expected time, two ideas are used:

- **Multiple-choice hashing:** Give each element multiple choices for positions where it can reside in the hash table
- **Relocation hashing:** Allow elements in the hash table to move after being placed

Cuckoo hashing applies the idea of multiple-choice and relocation together and guarantees O(1) worst case lookup time!

- **Multiple-choice:** We give a key **two choices** the h1(key) and h2(key) for residing.
- **Relocation**: It may happen that h1(key) and h2(key) are preoccupied. This is resolved by imitating the Cuckoo bird: *it pushes the other eggs or young out of the nest when it hatches*. Analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location. This leaves us with the problem of re-placing the older key.
  - If the alternate position of older key is vacant, there is no problem.
  - Otherwise, the older key displaces another key. This continues until the procedure finds a vacant position, or enters a cycle. In the case of a cycle, new hash functions are chosen and the whole data structure is 'rehashed'. Multiple rehashes might be necessary before Cuckoo succeeds.

Let's start by inserting **20** at its possible position in the first table determined by h1(20):

table[1]	-	-	-	-	-	-	-	-	-	20	-
table[2]	-	-	-	-	-	-	-	-	-	-	-

Next: 50

table[1]	-	-	-	-	-	-	50	-	-	20	-
table[2]	-	-	-	-	-	-	-	-	-	-	-

Next: 53. h1(53) = 9. But 20 is already there at 9. We place 53 in table 1 & 20 in table 2 at h2(20)

.

table[1]	-	-	-	-	-	-	50	-	-	53	-
table[2]	-	20	-	-	-	-	-	-	-	-	-

Next: 75. h1(75) = 9. But 53 is already there at 9. We place 75 in table 1 & 53 in table 2 at h2(53)

table[1]	-	-	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	-	-

Next: 100. h1(100) = 1.

table[1]	-	100	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	-	-

Next: 67. h1(67) = 1. But 100 is already there at 1. We place 67 in table 1 & 100 in table 2

table[1]	-	67	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	100	-

Next: 105. h1(105) = 6. But 50 is already there at 6. We place 105 in table 1 & 50 in table 2 at h2(50) = 4. Now 53 has been displaced. h1(53) = 9. 75 displaced: h2(75) = 6.

table[1]	-	67	-	-	-	-	105	-	-	53	-
table[2]	-	20	-	-	50	-	75	-	-	100	-

Next: 3. h1(3) = 3.

table[1]	-	67	-	3	-	-	105	-	-	53	-
table[2]	-	20	-	-	50	-	75	-	-	100	-

Next: 36. h1(36) = 3. h2(3) = 0.

table[1]	-	67	-	36	-	-	105	-	-	53	-
table[2]	3	20	-	-	50	-	75	-	-	100	-

Next: 39. h1(39) = 6. h2(105) = 9. h1(100) = 1. h2(67) = 6. h1(75) = 9. h2(53) = 4. h1(50) = 6. h2(39) = 3. Here, the new key 39 is displaced later in the recursive calls to place 105, which it displaced.

table[1]	-	100	-	36	-	-	50	-	-	75	-
table[2]	3	20	-	39	53	-	67	-	-	105	-

Hopscotch Hashing

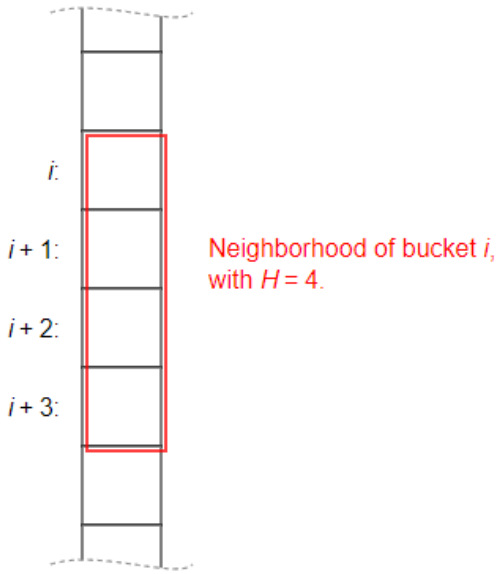
A Hopscotch hash table is based on open addressing i.e. it has an array of buckets and stores at most one key-value pair in each bucket.

Upon collisions, Hopscotch hashing aims to keep key-value pairs close to the original bucket (in it's neighborhood). This keeps the chains short and achieves good memory locality.

Neighborhoods

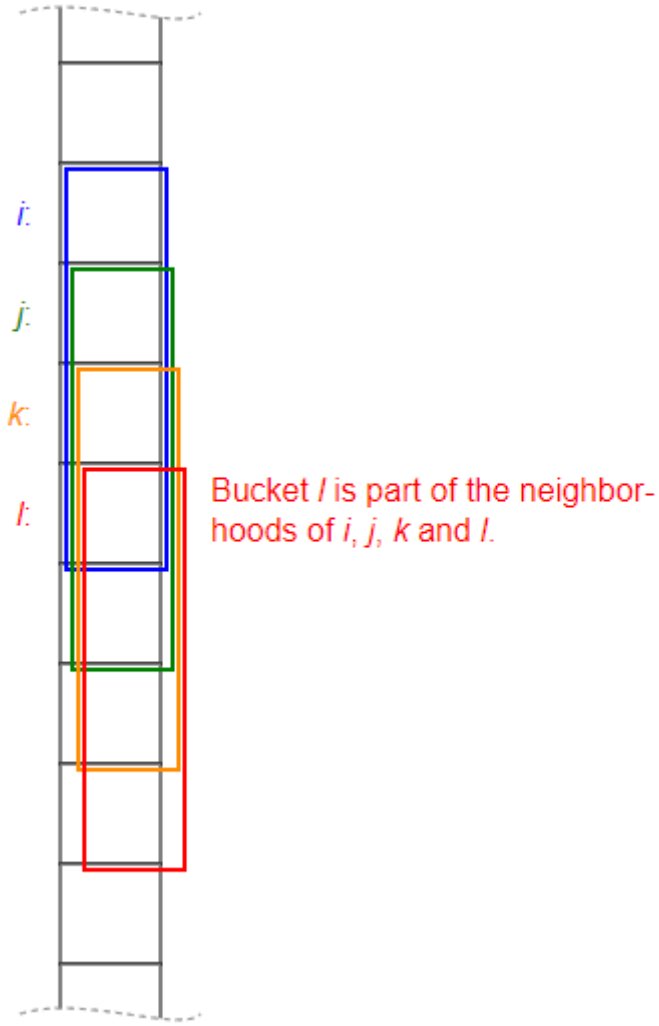
The neighborhood of bucket i are the H consecutive buckets starting in i. H is a configurable constant.

A key-value pair will always be stored within the neighborhood of the bucket it originally hashes to. (During lookup, only the buckets in the relevant neighborhood are considered.)



Neighborhoods Overlap

The neighborhoods of consecutive buckets overlap. Each bucket is part of H neighborhoods.



Optimal value for H**:** The hash table can't handle more than H collisions in one bucket, so H shouldn't be too small. Also, if H is too large, lookups will be less efficient. In practice, a good choice for H is one that optimizes the usage of the cache lines. Good performance can be achieved if an entire neighborhood can be fetched in one memory roundtrip. The implementation in the original paper uses H = 32.

Insertion

Suppose a key, k, is to be inserted, and that it hashes to bucket i. Bucket i is referred to as the home bucket of k.

Case: Home bucket is empty

If the home bucket is empty, the key is placed there and we're done.

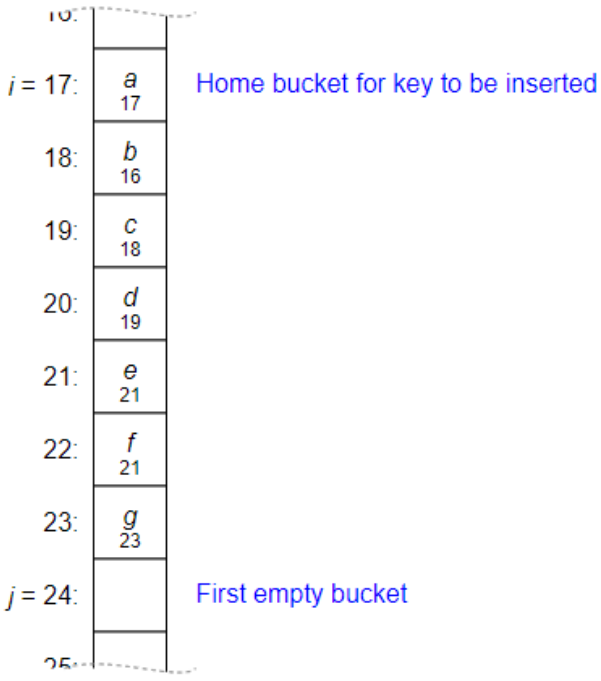
Case: Home bucket is occupied

If bucket i is not empty, then we search for an empty bucket linearly, starting from i (linear probing). Suppose we find an empty bucket at index j.

If j is within in the neighborhood of i we put k there and return.

If j is outside of i's neighborhood, we try to move an element between i and j downwards so that the empty space moves upwards, closer to i. We repeat this process until the empty space has been moved into the neighborhood of i, and then use that space for k.

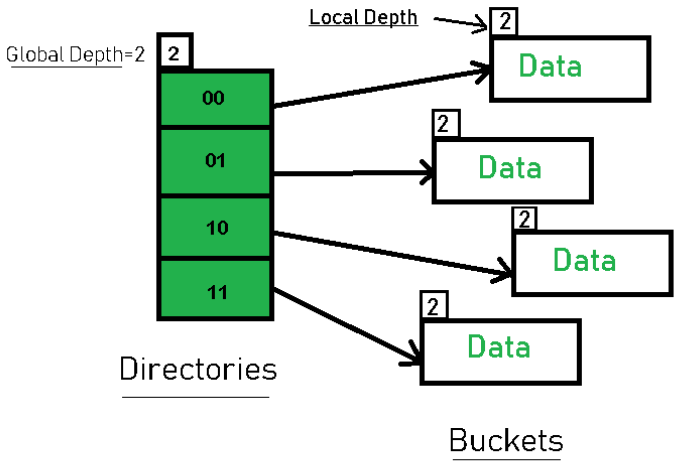
For the sake of this example, let's assume that i = 17 and j = 24 and that we have neighborhoods of size 4. The home bucket of each key will be noted below the key.



Extendible Hashing

Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Main features of Extendible Hashing: The main features in this hashing technique are:



Extendible Hashing

Directories: These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.

Buckets: They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.

Global Depth: It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.

Local Depth: It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

Bucket Splitting: When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

Directory Expansion: Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Example based on Extendible Hashing:

Now, let us consider a prominent example of hashing the following elements:

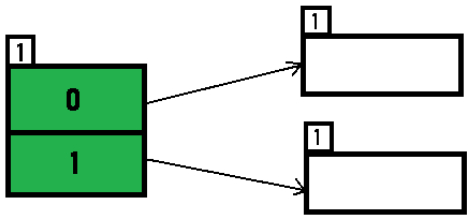
16, 4, 6, 22, 24, 10, 31, 7, 9, 20, 26.

Bucket Size: 3 (Assume)

Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

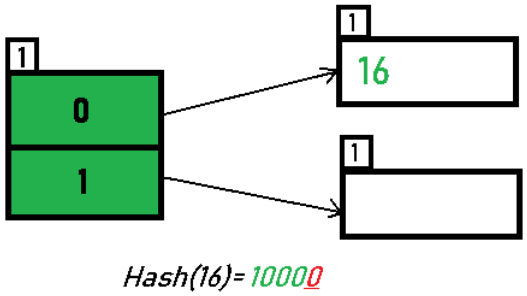
16- 10000
4- 00100
6- 00110
22- 10110
24- 11000
10- 01010
31- 11111
7- 00111
9- 01001
20- 10100
26- 11010

- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



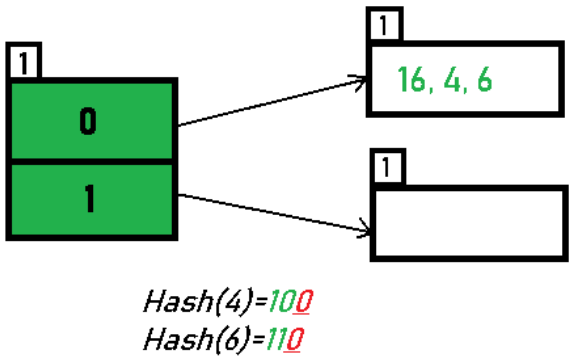
Inserting 16:

The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.

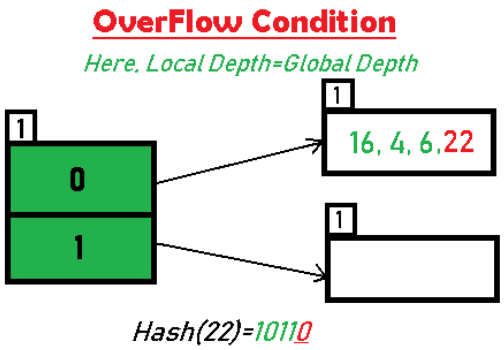


Inserting 4 and 6:

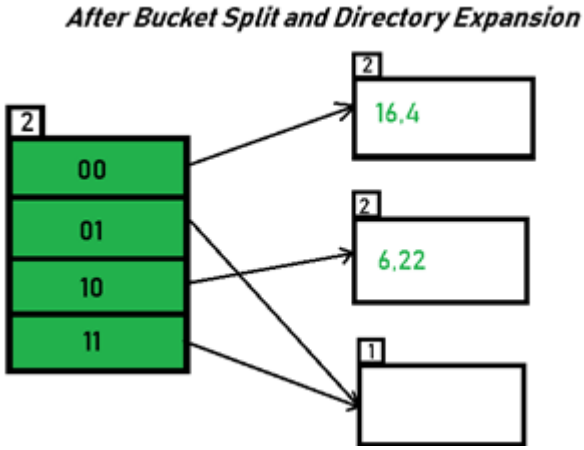
Both 4(10**0**) and 6(11**0**) have 0 in their LSB. Hence, they are hashed as follows:



Inserting 22: The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

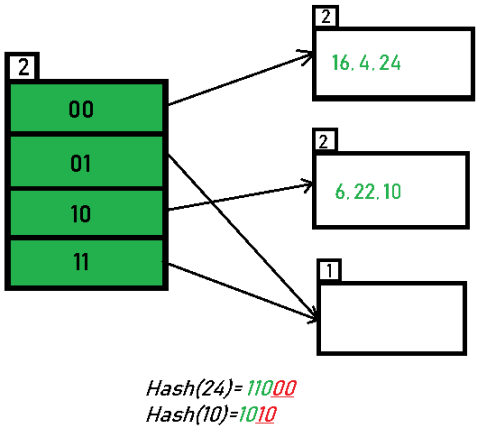


As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

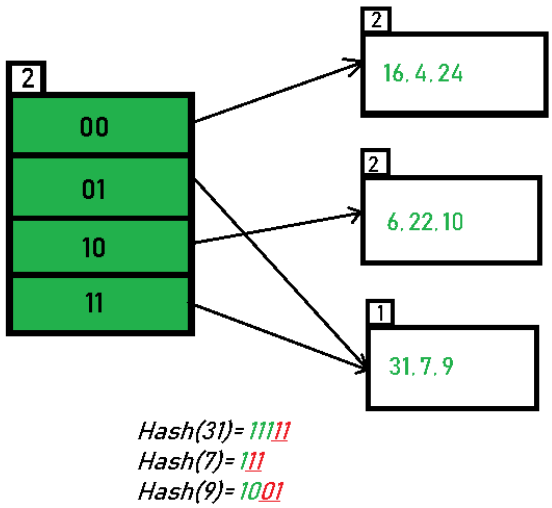


*Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.

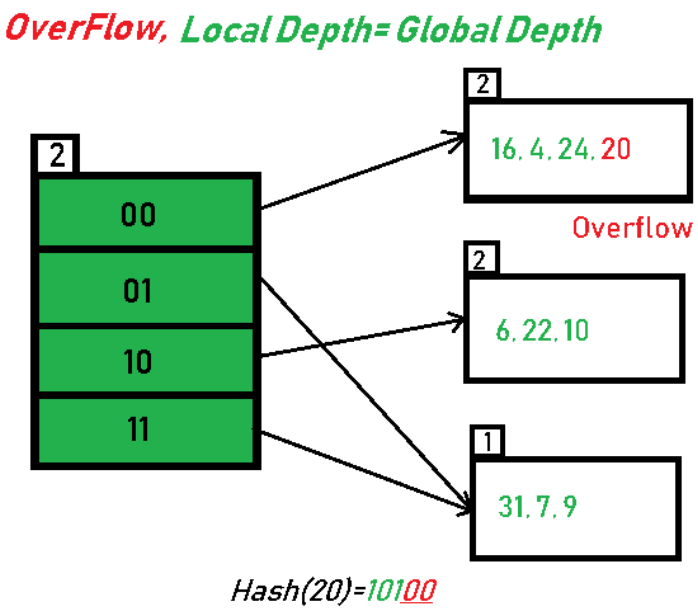
Inserting 24 and 10: 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



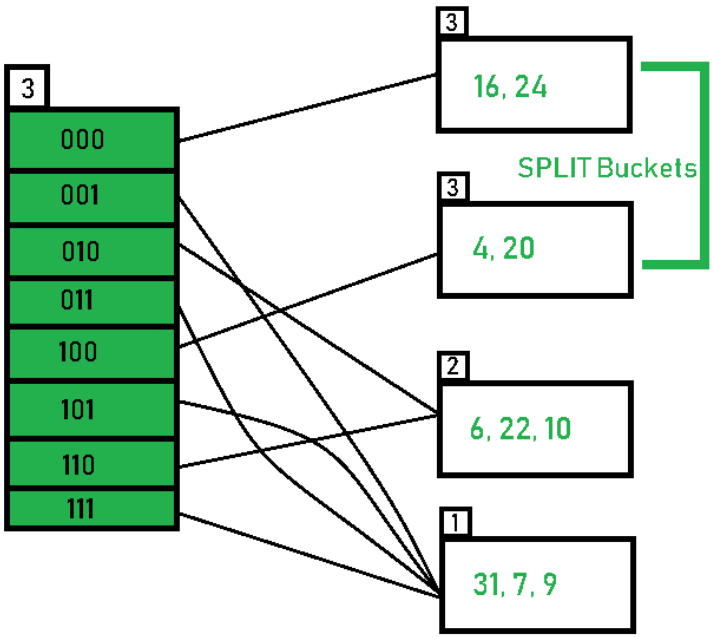
Inserting 31,7,9: All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



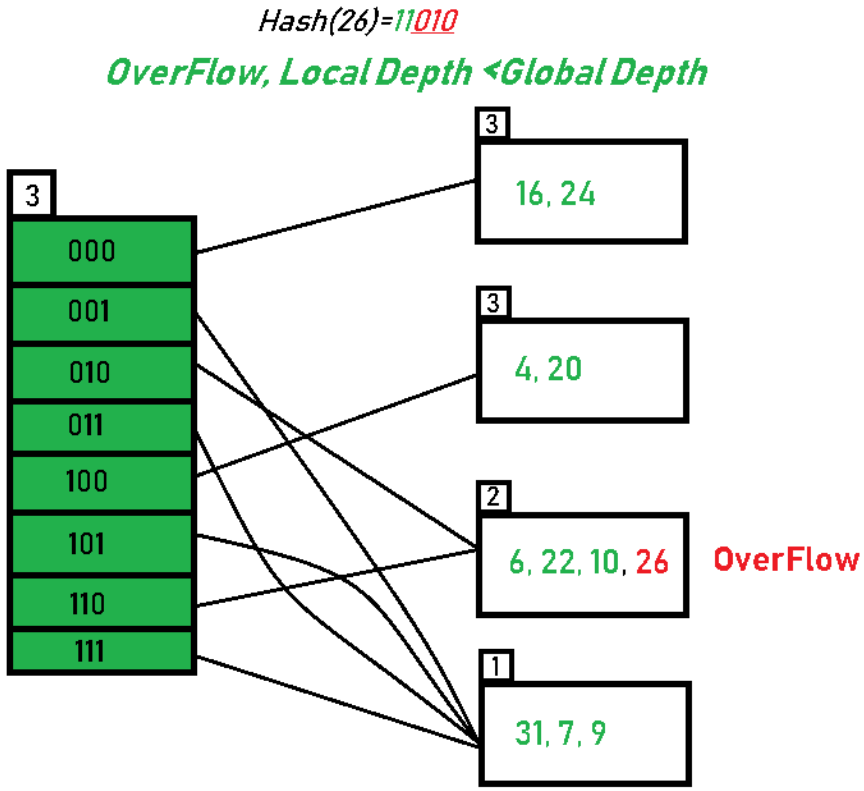
Inserting 20: Insertion of data element 20 (101**00**) will again cause the overflow problem.



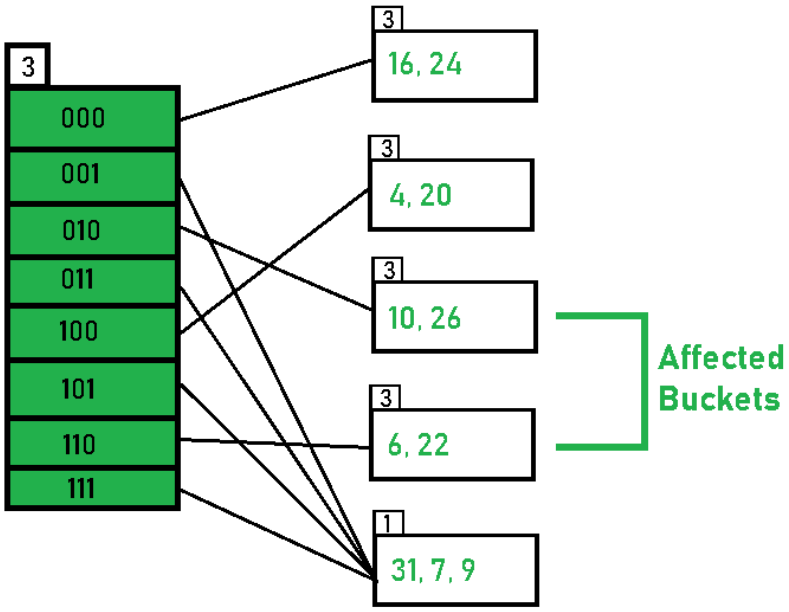
20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



Inserting 26: Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



Hashing of 11 Numbers is Thus Completed.

Advantages:

- Data retrieval is less expensive (in terms of computing).
- No problem of Data-loss since the storage capacity increases dynamically.
- With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations of Extendible Hashing:

- The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
- Size of every bucket is fixed.
- Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
- This method is complicated to code.

Shell Sort

Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

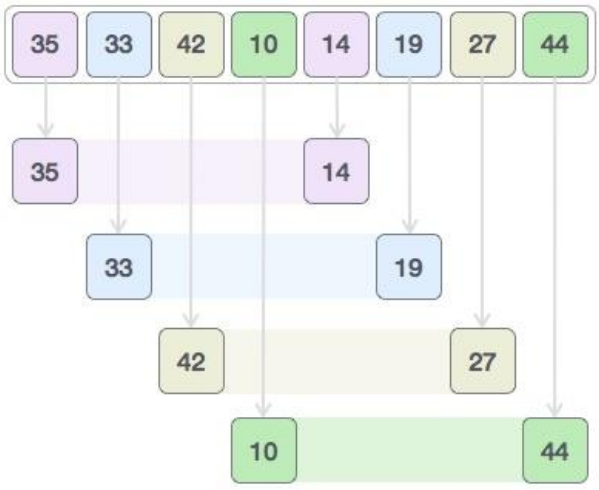
Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

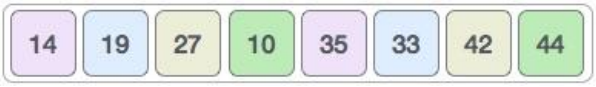
This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as **interval.**

Example:

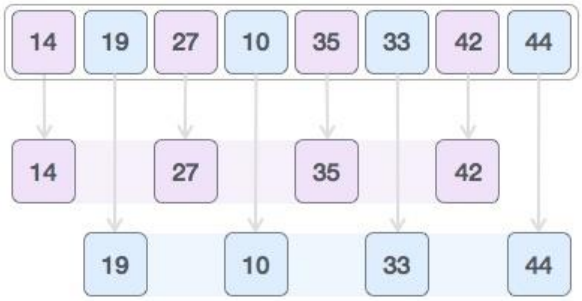
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



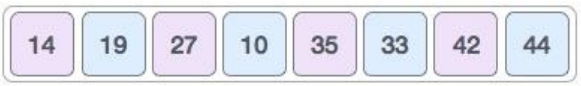
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction –

We see that it required only four swaps to sort the rest of the array.

```c
// Shell Sort in C programming

#include <stdio.h>

// Shell sort
void shellSort(int array[], int n) {
  // Rearrange elements at each n/2, n/4, n/8, ... intervals
  for (int interval = n / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < n; i += 1) {
      int temp = array[i];
      int j;
      for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
        array[j] = array[j - interval];
      }
      array[j] = temp;
    }
  }
}

// Print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// Driver code
int main() {
  int data[] = {9, 8, 3, 7, 5, 6, 4, 1};
  int size = sizeof(data) / sizeof(data[0]);
  shellSort(data, size);
  printf("Sorted array: \n");
  printArray(data, size);
}
```

Heap Sort

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

What is a heap?

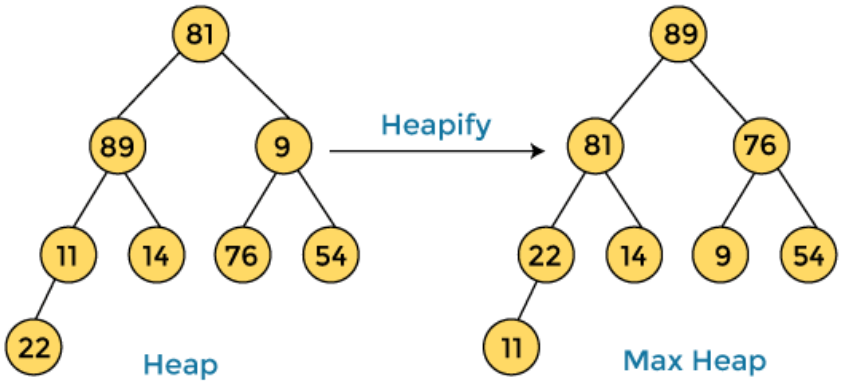
A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

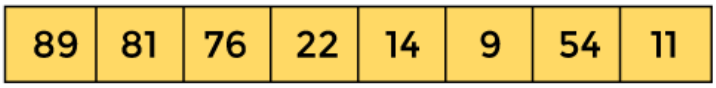
Heapsort is the in-place sorting algorithm.

To understand it more clearly, let's take an unsorted array and try to sort it using heap sort.

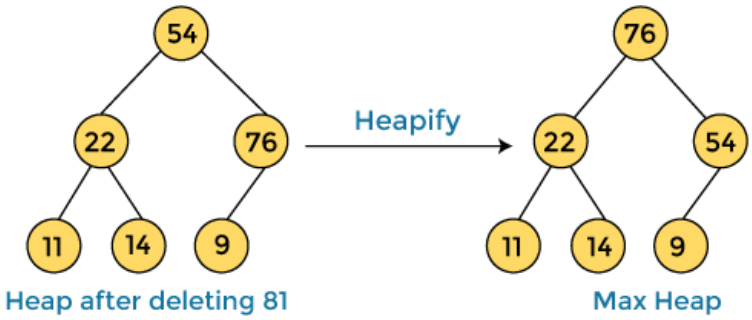
First, we have to construct a heap from the given array and convert it into max heap.



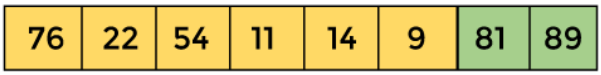
After converting the given heap into max heap, the array elements are -



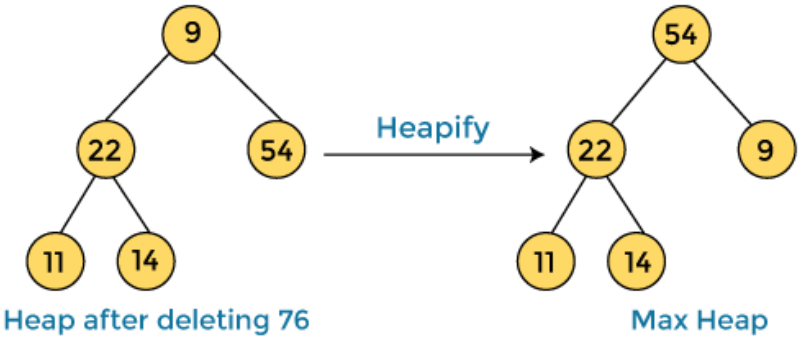
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



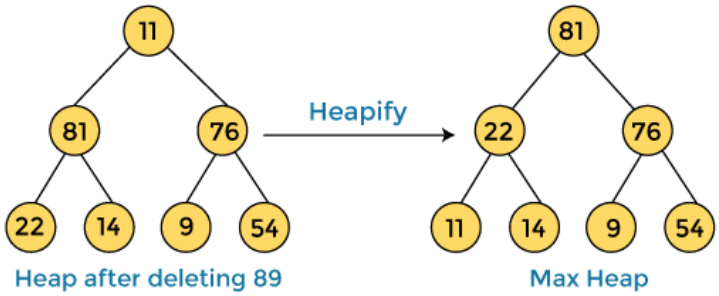
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -



In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



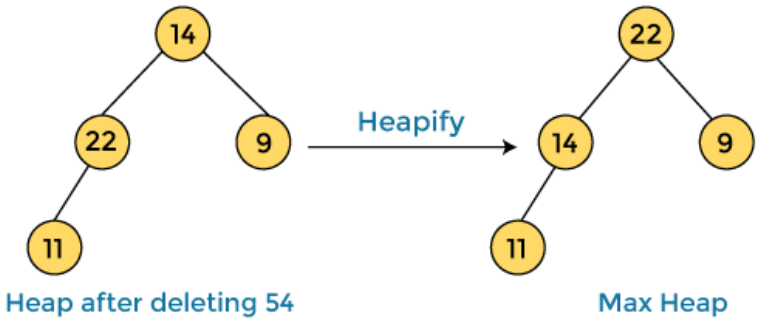
After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -



After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.

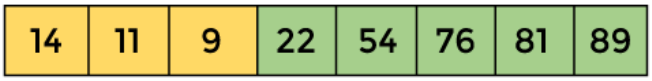
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



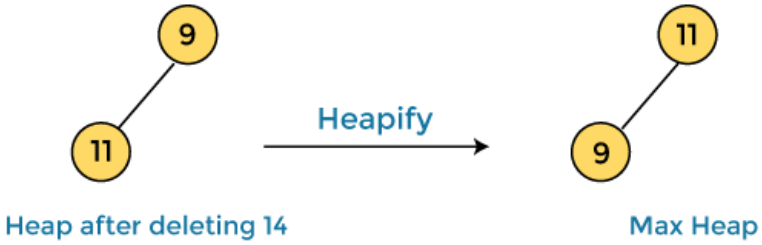
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

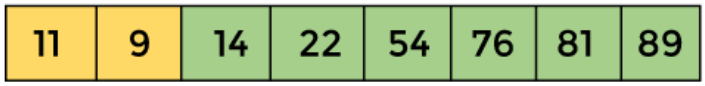
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

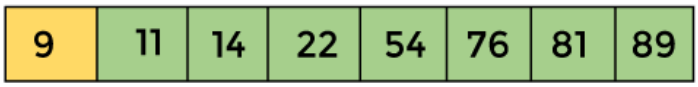


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



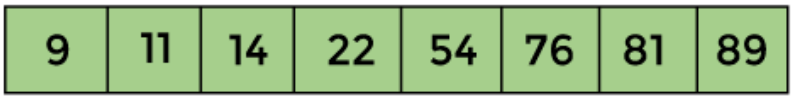
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Time Complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

Algorithm:

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
   swap arr[1] with arr[i]
      heap_size[arr] = heap_size[arr] ? 1
      MaxHeapify(arr,1)
End
```

i. **BuildMaxHeap(arr)**

```
BuildMaxHeap(arr)

   heap_size(arr) = length(arr)
   for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

ii. MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

Quick Sort

Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

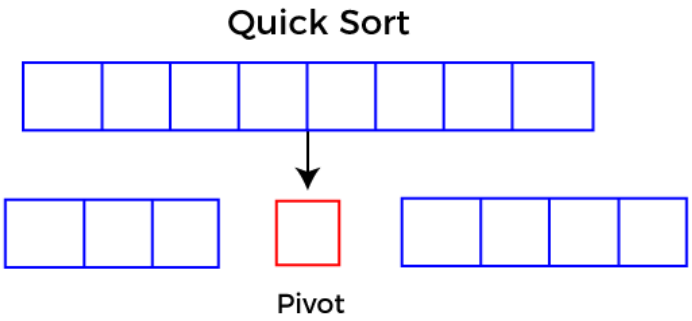
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



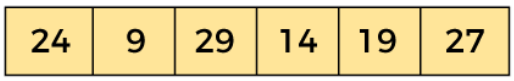
Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

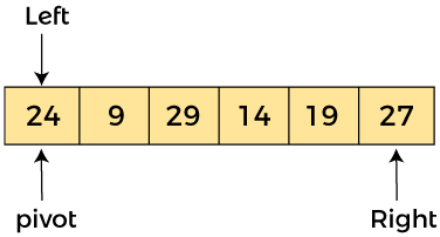
To understand the working of quick sort, let's take an unsorted array.

Let the elements of array are -

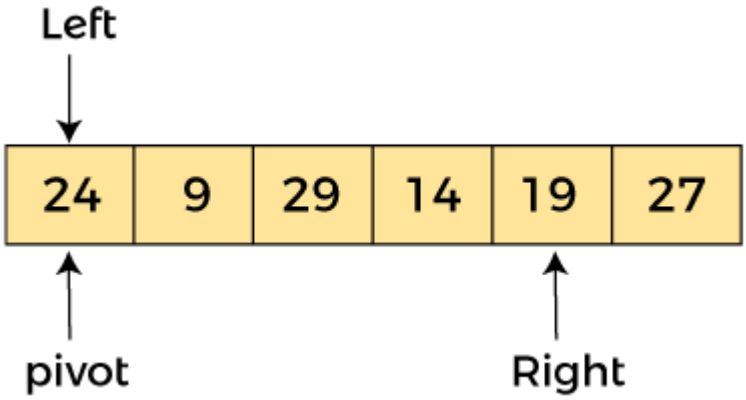


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

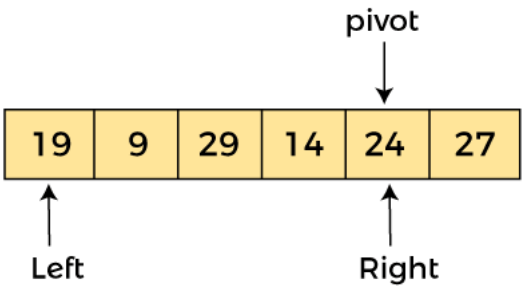


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



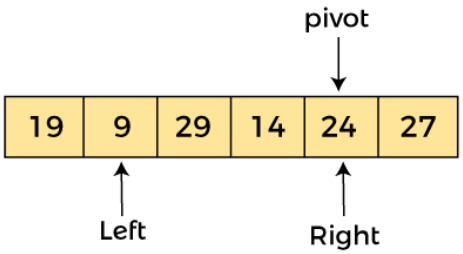
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

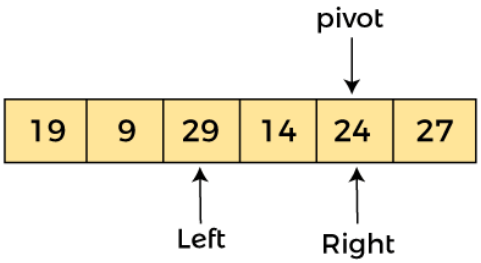


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

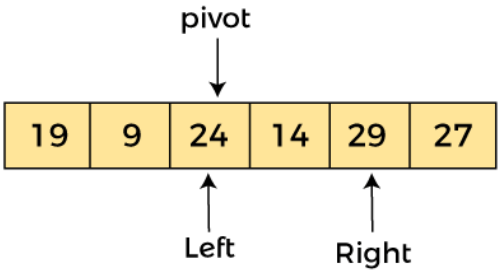
As a[pivot] > a[left], so algorithm moves one position to right as -



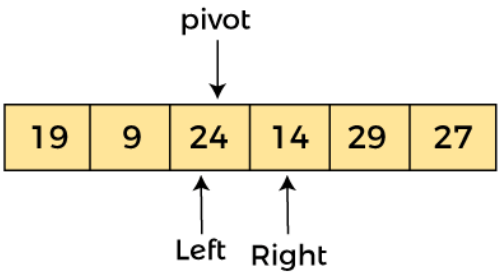
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



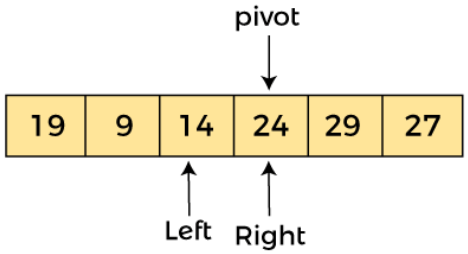
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



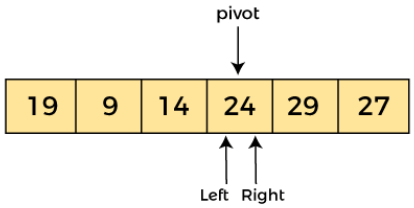
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



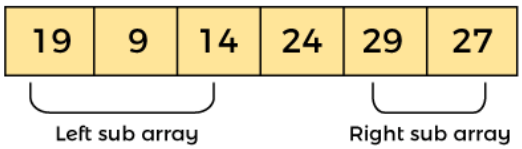
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



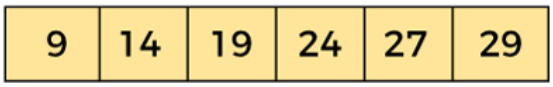
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Time Complexity

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.

- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Algorithm:

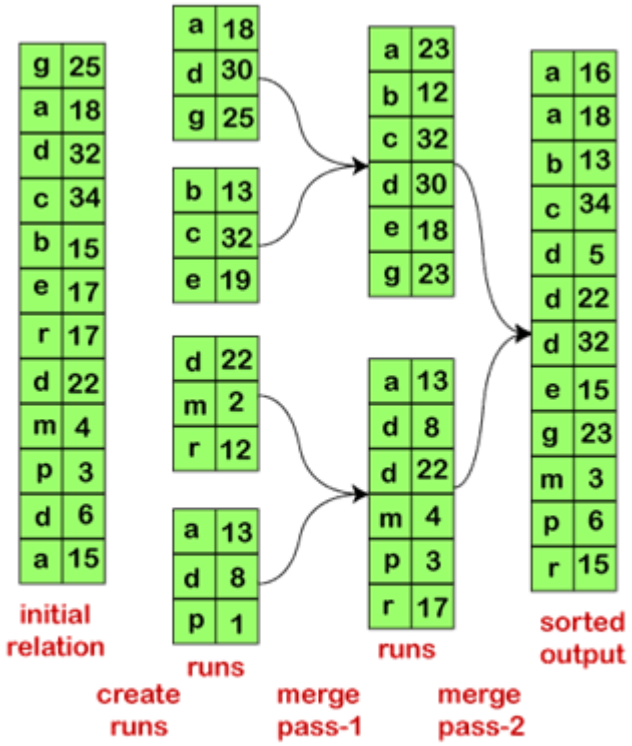
```
QUICKSORT (array A, start, end)
{
    if (start < end)
    {
        p = partition(A, start, end)
        QUICKSORT (A, start, p - 1)
        QUICKSORT (A, p + 1, end)
    }
}

PARTITION (array A, start, end)
{
    pivot ? A[end]
    i ? start-1
    for j ? start to end -1 {
            do if (A[j] < pivot) {
            then i ? i + 1
            swap A[i] with A[j]
            }
    }
    swap A[i+1] with A[end]
    return i+1
}
```

External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted does not fit into the main memory of a computing device (usually RAM) and instead, must reside in the slower external memory (usually a hard drive).

External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in the main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.



External sorting using sort-merge

Dictionaries

Dictionary is one of the important Data Structures that is usually used to store data in the key-value format. Each element presents in a dictionary data structure compulsorily have a key and some value is associated with that particular key. In other words, we can also say that Dictionary data structure is used to store the data in key-value pairs. Other names for the Dictionary data structure are associative array, map, symbol table but broadly it is referred to as Dictionary.

A dictionary or associative array is a general-purpose data structure that is used for the storage of a group of objects.

The various operations that are performed on a Dictionary or associative array are:

Add or Insert: In the Add or Insert operation, a new pair of keys and values is added in the Dictionary or associative array object.

Replace or reassign: In the Replace or reassign operation, the already existing value that is associated with a key is changed or modified. In other words, a new value is mapped to an already existing key.

Delete or remove: In the Delete or remove operation, the already present element is unmapped from the Dictionary or associative array object.

Find or Lookup: In the Find or Lookup operation, the value associated with a key is searched by passing the key as a search argument.

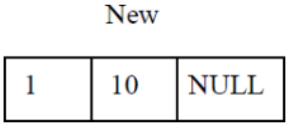
Linear-List Representation:

Linear List Representation The dictionary can be represented as a linear list. The linear list is a collection of pair and value. There are two methods of representing linear list.

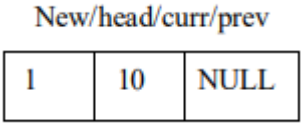
1. **Sorted Array**- An array data structure is used to implement the dictionary.

2. **Sorted Chain**- A linked list data structure is used to implement the dictionary

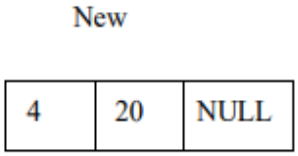
Insertion of new node in the dictionary: Consider that initially dictionary is empty then head = NULL We will create a new node with some key and value contained in it.



Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node.

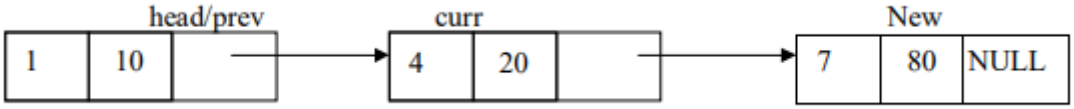


Insert a record, key=4 and value=20,

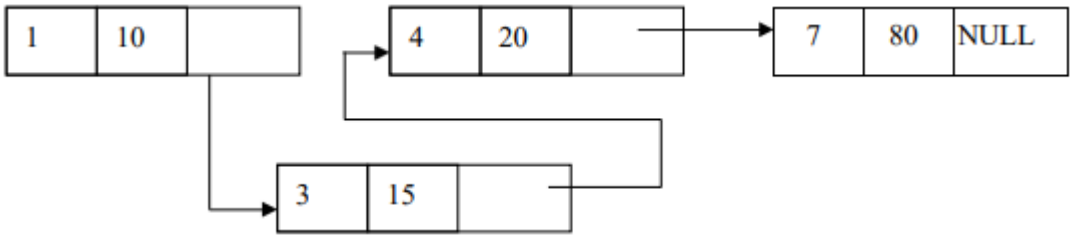


Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node

Add a new node then

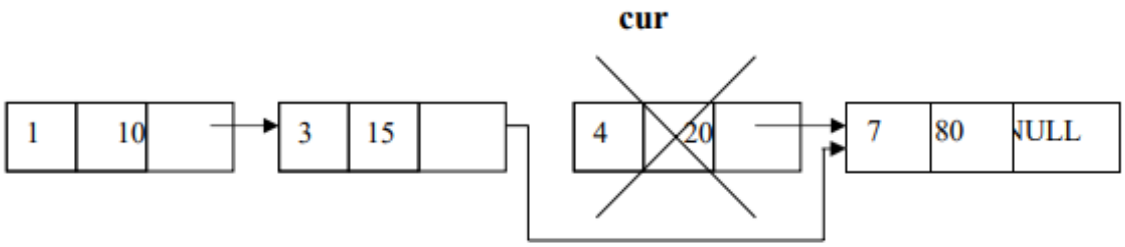


If we insert, then we have to search for it proper position by comparing key value. (curr->key < New->key) is false. Hence else part will get executed

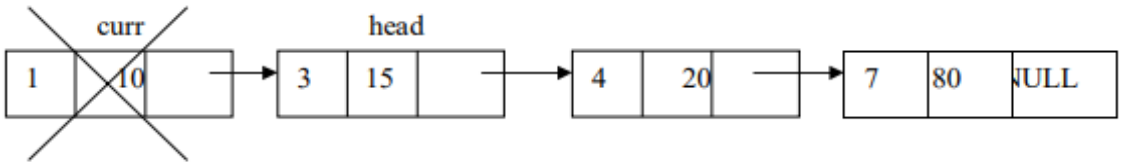


The delete operation:

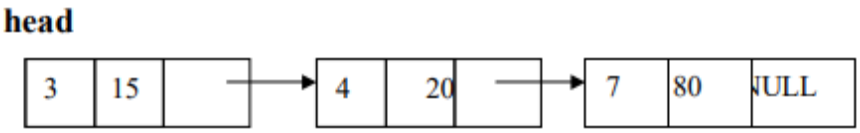
Case 1: Initially assign 'head' node as 'curr' node. Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then



Case 2: If the node to be deleted is head node i.e. if(curr==head) Then, simply make 'head' node as next node and delete 'curr'



Hence the list becomes



Skip-List Representation:

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

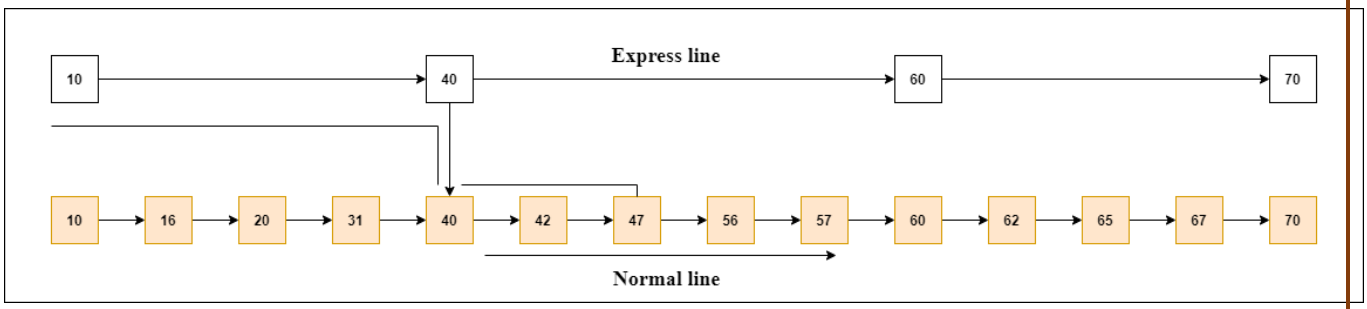
The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



Basic Operations

Insertion operation: It is used to add a new node to a particular location in a specific situation.

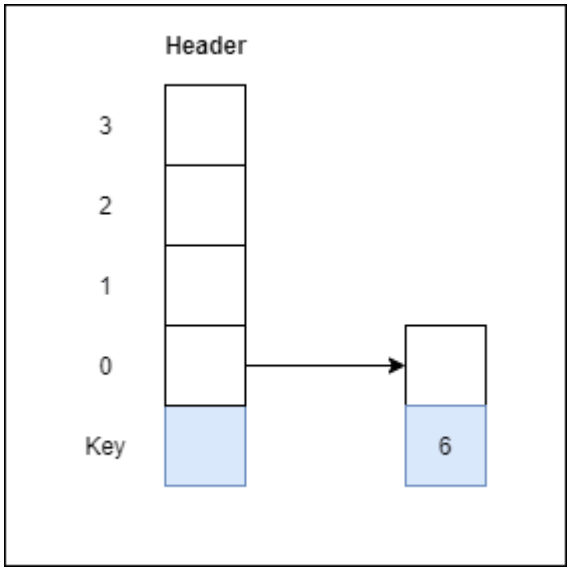
Deletion operation: It is used to delete a node in a specific situation.

Search Operation: The search operation is used to search a particular node in a skip list.

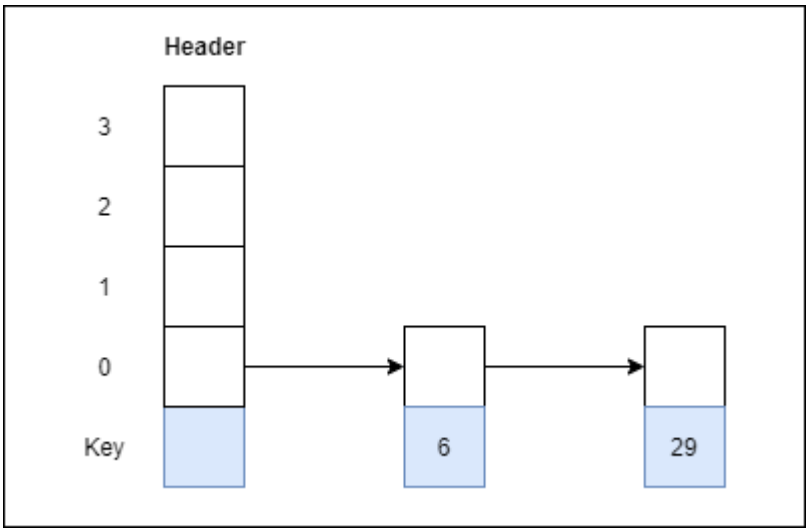
Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

1. 6 with level 1.
2. 29 with level 1.
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

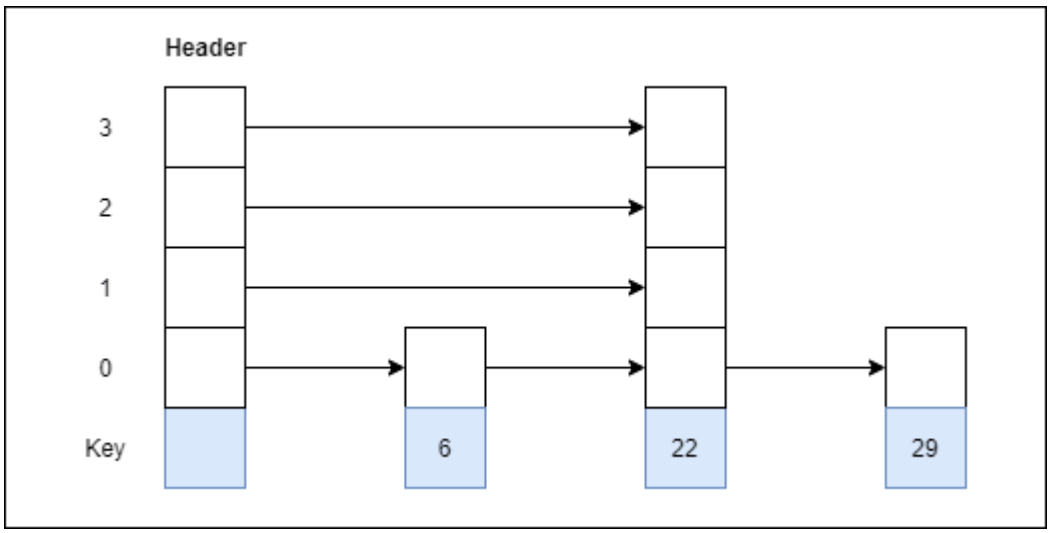
Step 1: Insert 6 with level 1



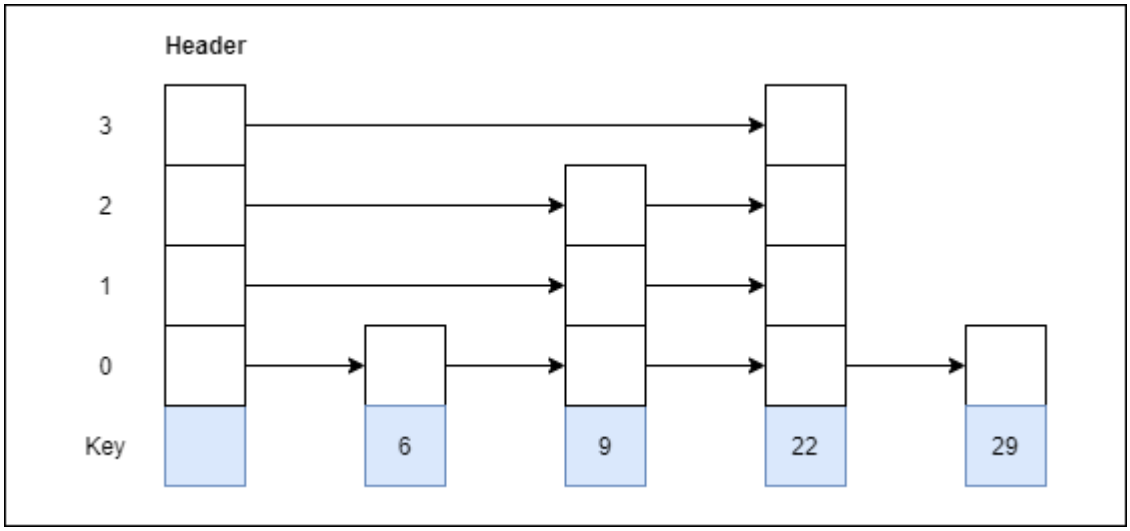
Step 2: Insert 29 with level 1



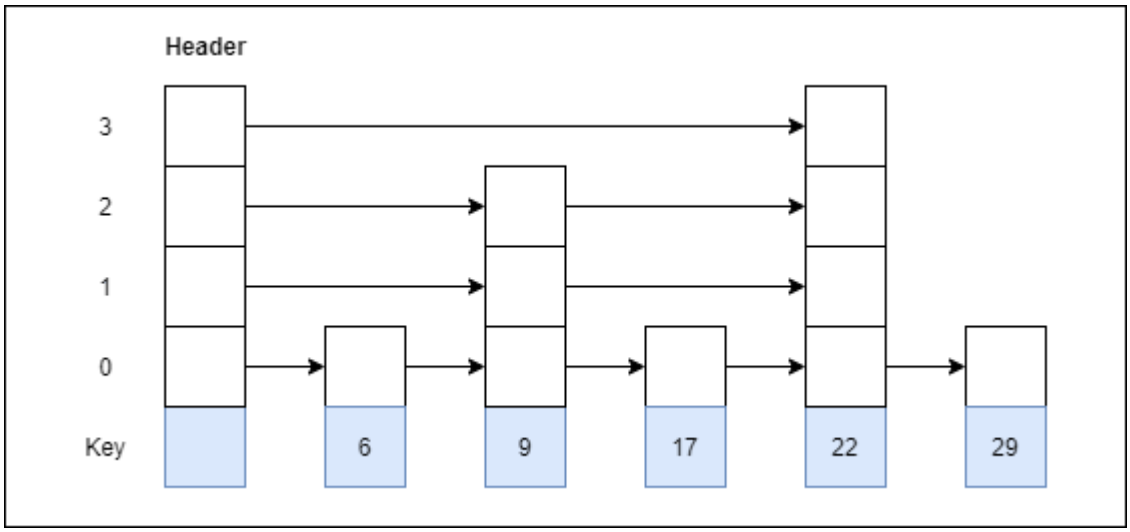
Step 3: Insert 22 with level 4



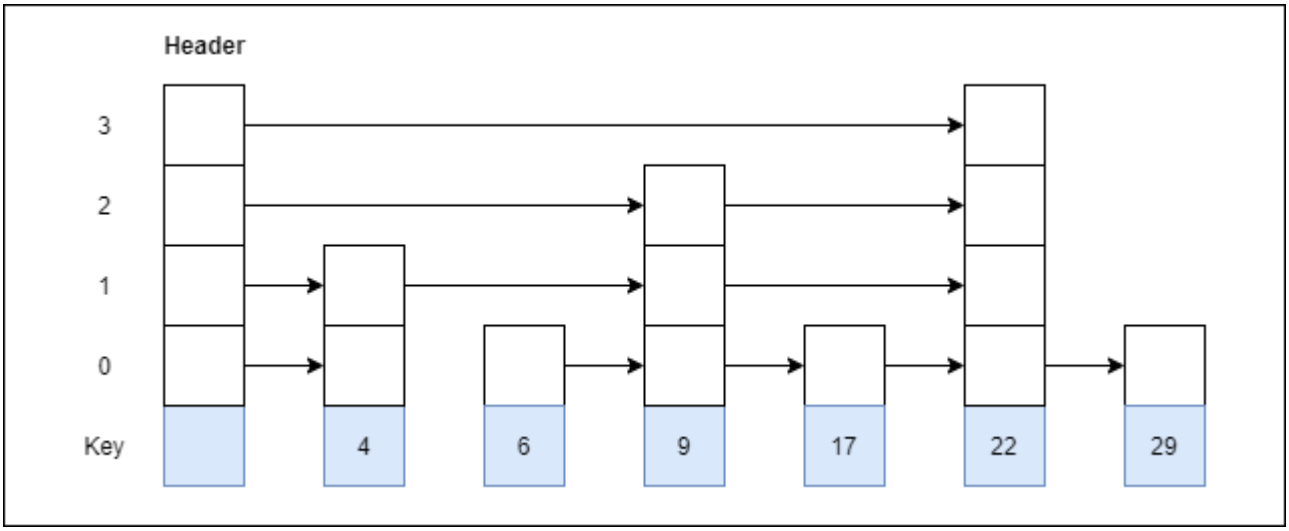
Step 4: Insert 9 with level 3



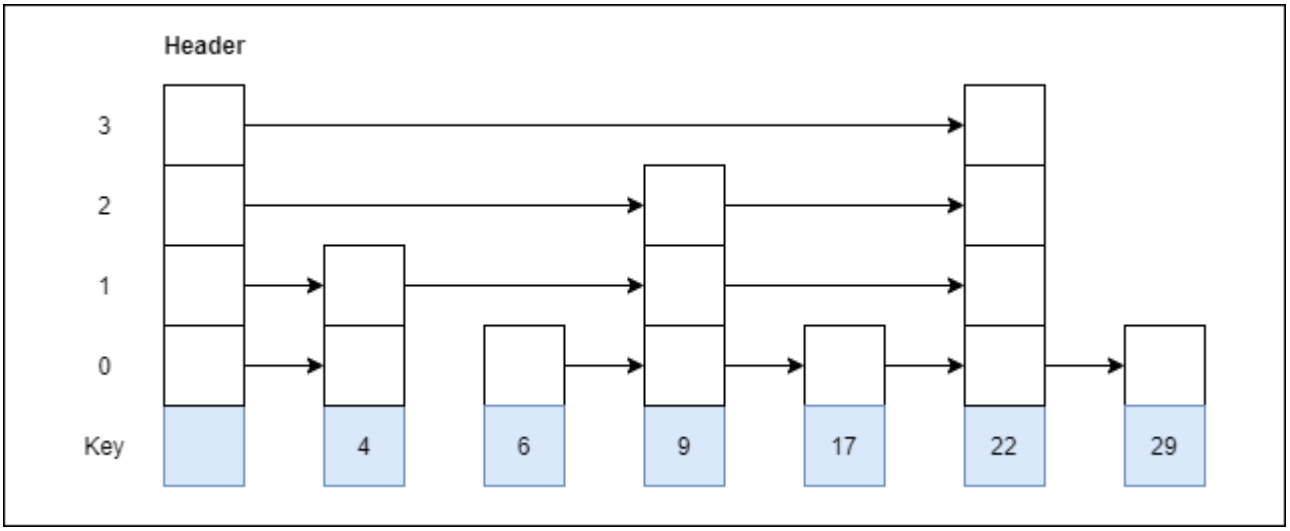
Step 5: Insert 17 with level 1



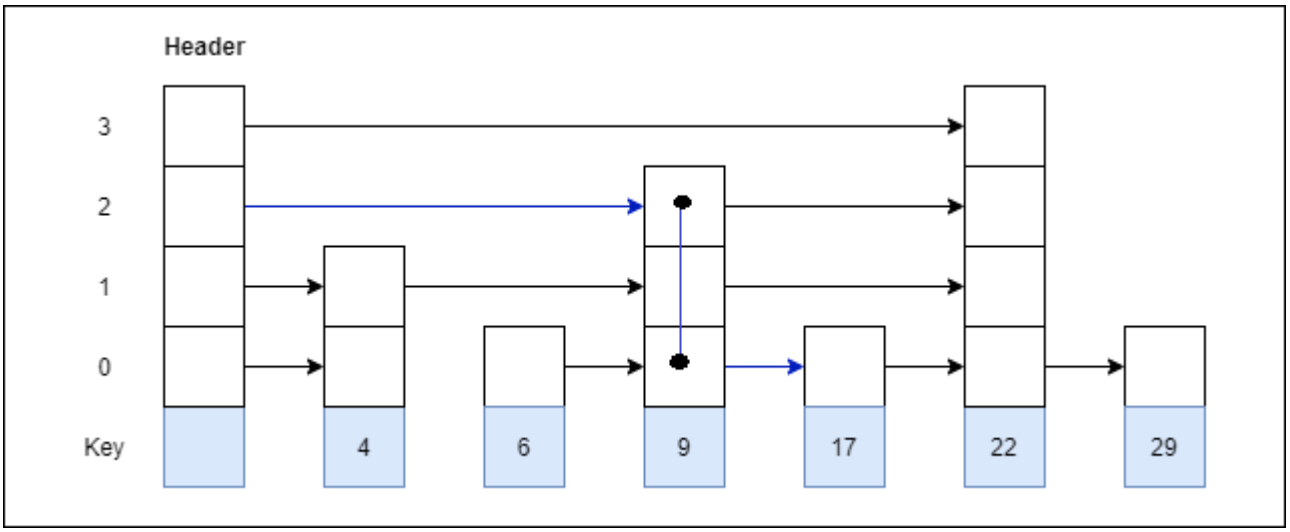
Step 6: Insert 4 with level 2



Example 2: Consider this example where we want to search for key 17.



Ans:



Advantages of the Skip list

- If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
- The skip list is simple to implement as compared to the hash table and the binary search tree.
- It is very simple to find a node in the list because it stores the nodes in sorted form.
- The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
- The skip list is a robust and reliable list.

Disadvantages of the Skip list

- It requires more memory than the balanced tree.
- Reverse searching is not allowed.
- The skip list searches the node much slower than the linked list.

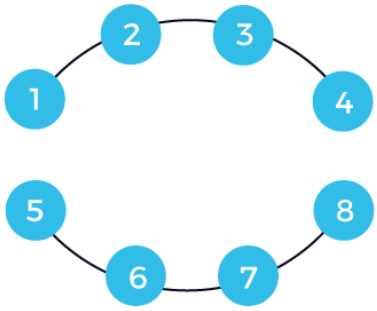
Disjoint Sets

It is a data structure that contains a collection of disjoint or non-overlapping sets. The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets. In this case, we can add new sets, we can merge the sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.

The disjoint set can be defined as the subsets where there is no common element between the two sets. Let's understand the disjoint sets through an example.

s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

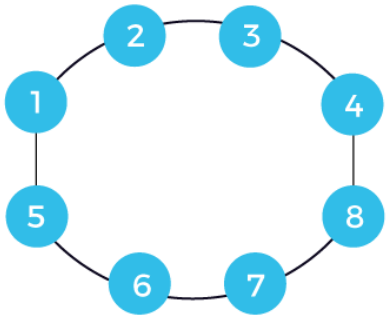


We have two subsets named s1 and s2. The s1 subset contains the elements 1, 2, 3, 4, while s2 contains the elements 5, 6, 7, 8. Since there is no common element between these two sets, we will not get anything if we consider the intersection between these two sets. This is also known as a disjoint set where no elements are common. Now the question arises how we can perform the operations on them. We can perform only two operations, i.e., find and union.

In the case of find operation, we have to check that the element is present in which set. There are two sets named s1 and s2 shown below:

Suppose we want to perform the union operation on these two sets. First, we have to check whether the elements on which we are performing the union operation belong to different or same sets. If they belong to the different sets, then we can perform the union operation; otherwise, not. For example, we want to perform the union operation between 4 and 8. Since 4 and 8 belong to different sets, so we apply the union operation. Once the union operation is performed, the edge will be added between the 4 and 8 shown as below:

When the union operation is applied, the set would be represented as:



s1Us2 = {1, 2, 3, 4, 5, 6, 7, 8}

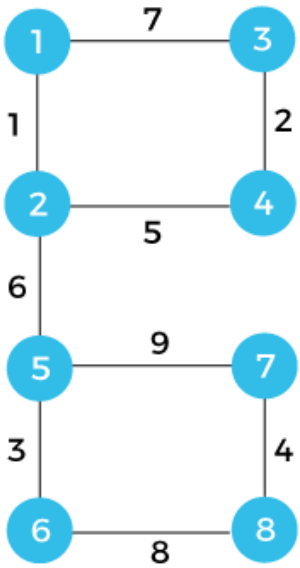
Suppose we add one more edge between 1 and 5. Now the final set can be represented as:

s3 = {1, 2, 3, 4, 5, 6, 7, 8}

If we consider any element from the above set, then all the elements belong to the same set; it means that the cycle exists in a graph.

How can we detect a cycle in a graph?

We will understand this concept through an example. Consider the below example to detect a cycle with the help of using disjoint sets.



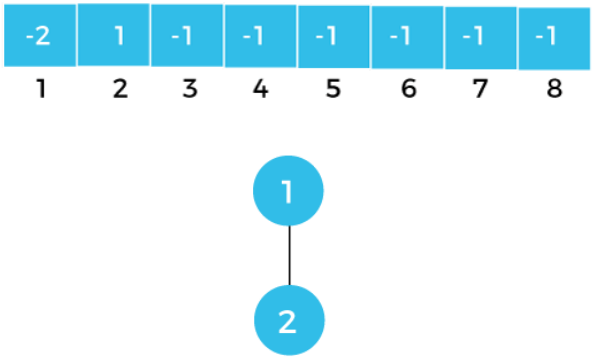
How can we detect a cycle with the help of an array?

Consider the below graph:

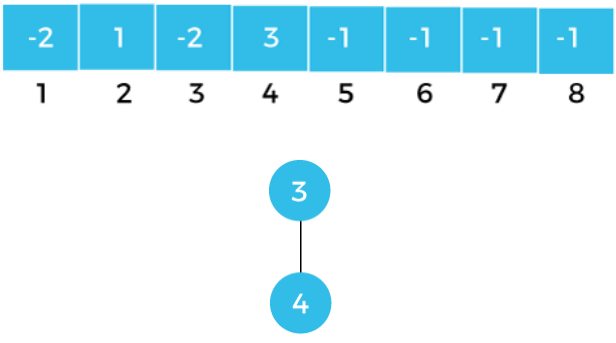
The above graph contains the 8 vertices. So, we represent all these 8 vertices in a single array. Here, indices represent the 8 vertices. Each index contains a -1 value. The -1 value means the vertex is the parent of itself.

Now we will see that how we can represent the sets in an array.

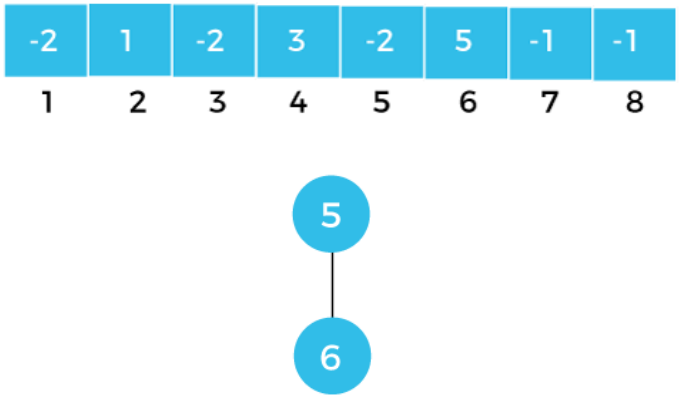
First, we consider the edge (1, 2). When we find 1 in an array, we observe that 1 is the parent of itself. Similarly, vertex 2 is the parent of itself, so we make vertex 2 as the child of vertex 1. We add 1 at the index 2 as 2 is the child of 1. We add -2 at the index 1 where '-' sign that the vertex 1 is the parent of itself and 2 represents the number of vertices in a set.



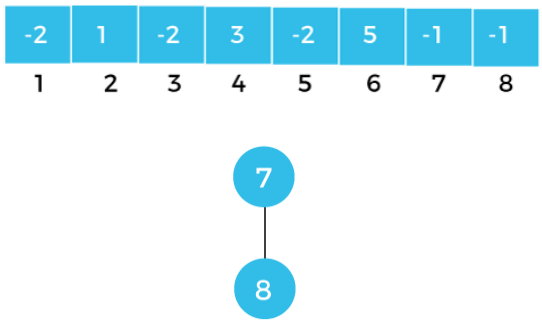
The next edge is (3, 4). When we find 3 and 4 in array; we observe that both the vertices are parent of itself. We make vertex 4 as the child of the vertex 3 so we add 3 at the index 4 in an array. We add -2 at the index 3 shown as below:



The next edge is (5, 6). When we find 5 and 6 in an array; we observe that both the vertices are parent of itself. We make 6 as the child of the vertex 5 so we add 5 at the index 6 in an array. We add -2 at the index 5 shown as below:

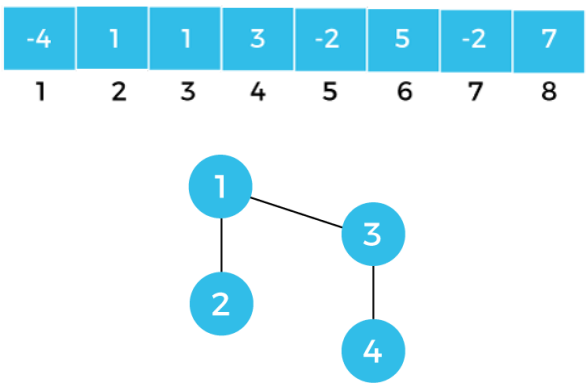


The next edge is (7, 8). Since both the vertices are parent of itself, so we make vertex 8 as the child of the vertex 7. We add 7 at the index 8 and -2 at the index 7 in an array shown as below:



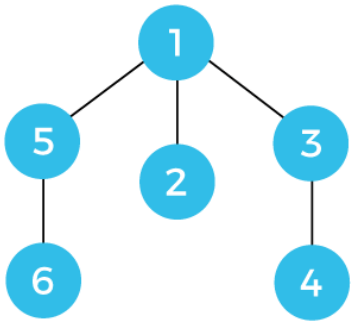
The next edge is (2, 4). The parent of the vertex 2 is 1 and the parent of the vertex is 3. Since both the vertices have different parent, so we make the vertex 3 as the child of vertex 1. We add 1 at the index 3. We add -4 at the index 1 as it contains 4 vertices.

Graphically, it can be represented as



The next edge is (2, 5). When we find vertex 2 in an array, we observe that 1 is the parent of the vertex 2 and the vertex 1 is the parent of itself. When we find 5 in an array, we find -2 value which means vertex 5 is the parent of itself. Now we have to decide whether the vertex 1 or vertex 5 would become a parent. Since the weight of vertex 1, i.e., -4 is greater than the vertex of 5, i.e., -2, so when we apply the union operation then the vertex 5 would become a child of the vertex 1 shown as below:

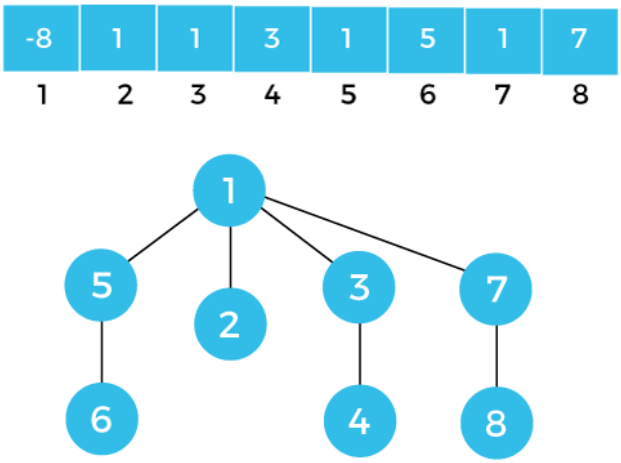
In an array, 1 would be added at the index 5 as the vertex 1 is now becomes a parent of vertex 5. We add -6 at the index 1 as two more nodes are added to the node 1.



The next edge is (1,3). When we find vertex 1 in an array, we observe that 1 is the parent of itself. When we find 3 in an array, we observe that 1 is the parent of vertex 3. Therefore, the parent of both the vertices are same; so, we can say that there is a formation of cycle if we include the edge (1,3).

The next edge is (6,8). When we find vertex 6 in an array, we observe that vertex 5 is the parent of vertex 6 and vertex 1 is the parent of vertex 5. When we find 8 in an array, we observe that vertex 7 is the parent of the vertex 8 and 7 is the parent of itself. Since the weight of vertex 1, i.e., -6 is greater than the vertex 7, i.e., -2, so we make the vertex 7 as the child of the vertex and can be represented graphically as shown as below:

We add 1 at the index 7 because 7 becomes a child of the vertex 1. We add -8 at the index 1 as the weight of the graph now becomes 8.



The last edge to be included is (5, 7). When we find vertex 5 in an array, we observe that vertex 1 is the parent of the vertex 5. Similarly, when we find vertex 7 in an array, we observe that vertex 1 is the parent of vertex 7. Therefore, we can say that the parent of both the vertices is same, i.e., 1. It means that the inclusion (5,7) edge would form a cycle.

Till now, we have learnt the weighted union where we perform the union operation according to the weights of the vertices. The higher weighted vertex becomes a parent and the lower weighted vertex becomes a child. The disadvantage of using this approach is that some nodes take more time to reach its parent. For example, in the above graph, if we want to find the parent of vertex 6, vertex 5 is the parent of vertex 6 so we move to the vertex 5 and vertex 1 is the parent of the vertex 5. To overcome such problem, we use the concept 'collapsing find'.

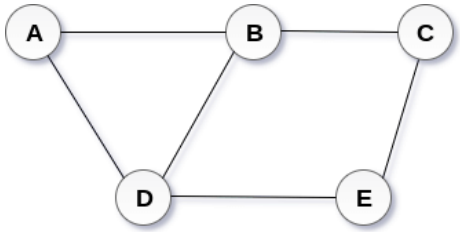
Graphs

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

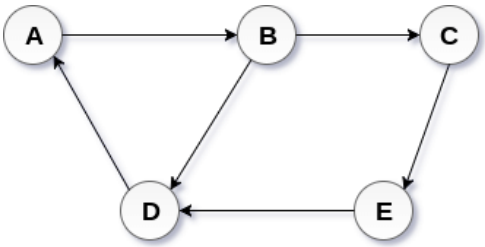
Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph



Directed Graph

Representation of Graphs in Data Structures

Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges. The vertices in a network represent entities.

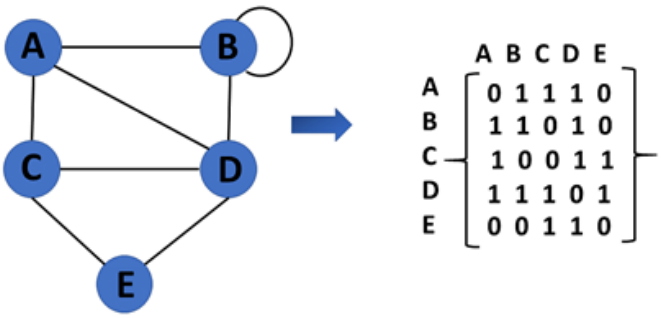
The most frequent graph representations are the two that follow:

- **Adjacency matrix**
- **Adjacency list**

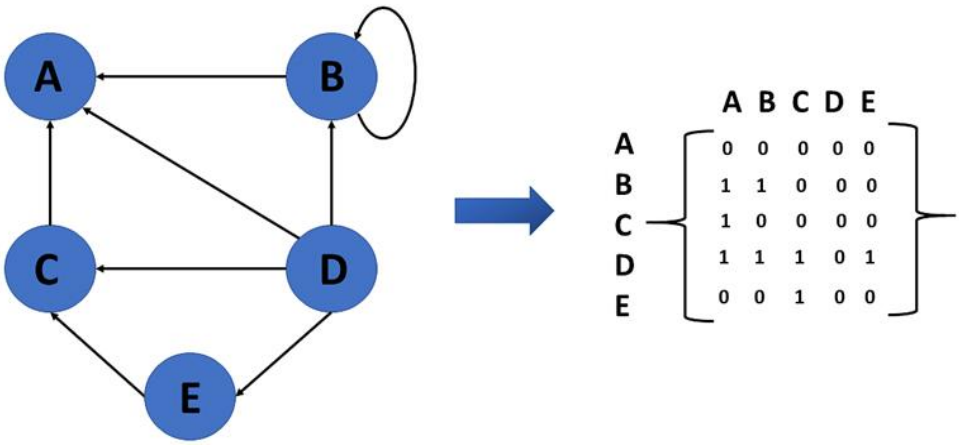
Adjacency Matrix

- A sequential representation is an adjacency matrix.
- It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph?
- You create an MXM matrix G for this representation. If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.
- If there is a weighted graph, you can record the edge's weight instead of 1s and 0s.

Undirected Graph Representation

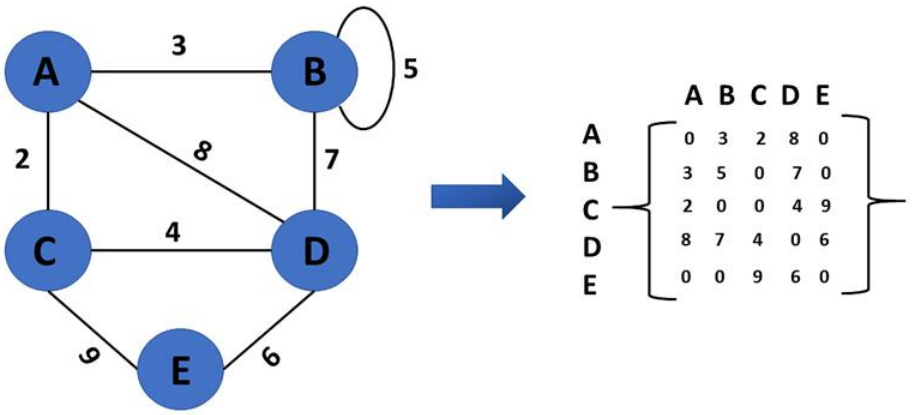


Directed Graph Representation



Weighted Undirected Graph Representation

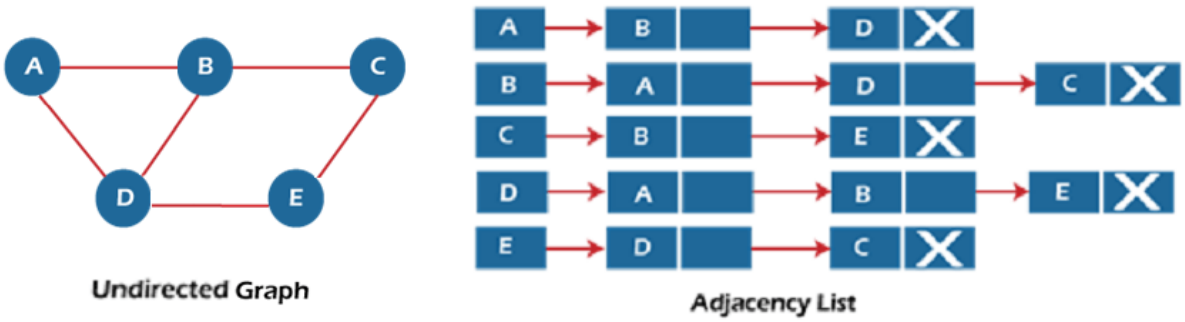
Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.

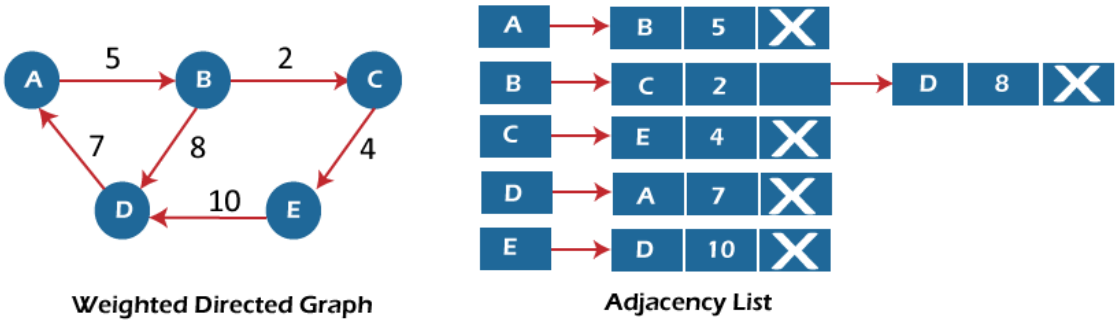


Adjacency List Representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.





Graph Traversal Algorithm

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

There are two techniques to implement a graph traversal algorithm:

- **Breadth-first search**
- **Depth-first search**

Breadth-First Search or BFS

BFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

It begins at the root of the graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally you require a queue.

Algorithm of breadth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in your graph, say v1, from which you want to traverse the graph.

Step 3: Examine any two data structures for traversing the graph.

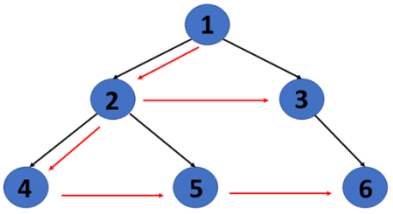
Visited array (size of the graph)

Queue data structure

Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.



BFS	1	2	3	4	5	6

Depth-First Search or DFS

DFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

The depth-first search (DFS) algorithm traverses or explores data structures such as trees and graphs. The DFS algorithm begins at the root node and examines each branch as far as feasible before back-tracking.

To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally a stack, is required.

Algorithm of depth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in our graph, say v1, from which you want to begin traversing the graph.

Step 3: Examine any two data structures for traversing the graph.

Visited array (size of the graph)

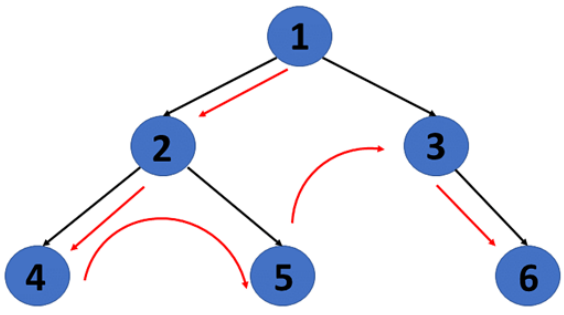
Stack data structure

Step 4: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.

Step 5: Now, using the FIFO principle, pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.

Step 6: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.

Step 7: Repeat step 6 until the stack data structure isn't empty.



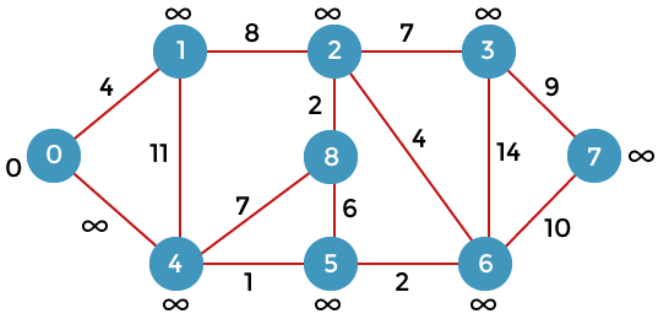
DFS	1	2	4	5	3	6

Dijkstra's Algorithm

Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

To understand the Dijkstra's Algorithm lets take a graph and find the shortest path from source to all nodes.

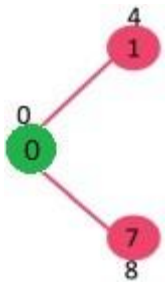
Consider below graph and **src = 0**



Step 1:

- The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

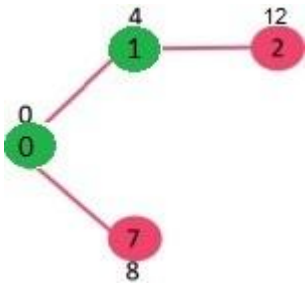
The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



Step 2:

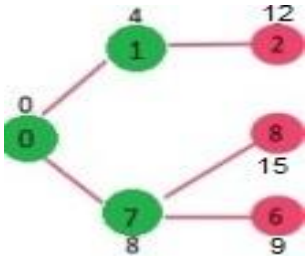
- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet.
- So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.

The distance value of vertex 2 becomes 12.



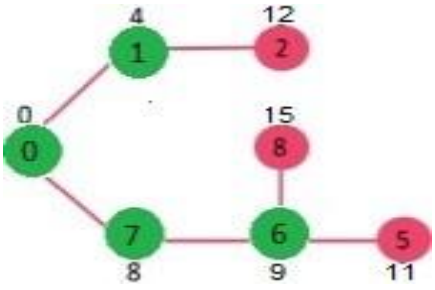
Step 3:

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}.
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

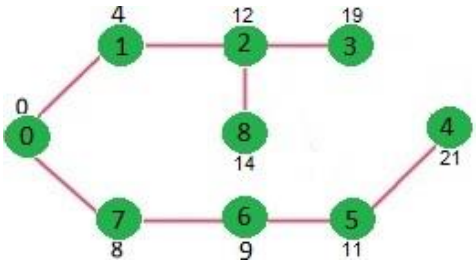


Step 4:

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}.
- Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



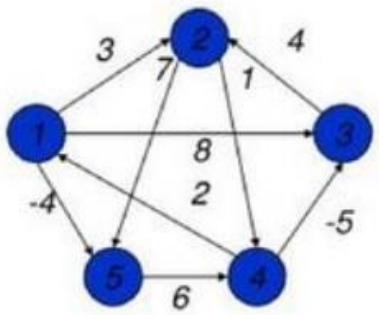
We repeat the above steps until sptSet includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



All-Pairs Shortest Path

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

It aims to figure out the shortest path from each vertex v to every other u. Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u's row and v's column should be the weight of the shortest path from u to v.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

Algorithm

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

Prim's Algorithm

Prim's algorithm is used to find the Minimum Spanning Tree for a given graph. But, what is a Minimum Spanning Tree, or MST for short? A minimum spanning tree T(V', E') is a subset of graph G(V, E) with the same number of vertices as of graph G (V' = V) and edges equal to the number of vertices of graph G minus one (E' = |V| - 1). Prim's approach identifies the subset of edges that includes every vertex in the graph, and allows the sum of the edge weights to be minimized.

Prim's algorithm starts with a single node and works its way through several adjacent nodes, exploring all of the connected edges along the way. Edges with the minimum weights that do not cause cycles in the graph get selected for t inclusion in the MST structure. Hence, we can say that Prim's algorithm takes a locally optimum decision in order to find the globally optimal solution. That is why it is also known as a Greedy Algorithm.

How to Create MST Using Prim's Algorithm

Step 1: Determine the arbitrary starting vertex.

Step 2: Keep repeating steps 3 and 4 until the fringe vertices (vertices not included in MST)remain.

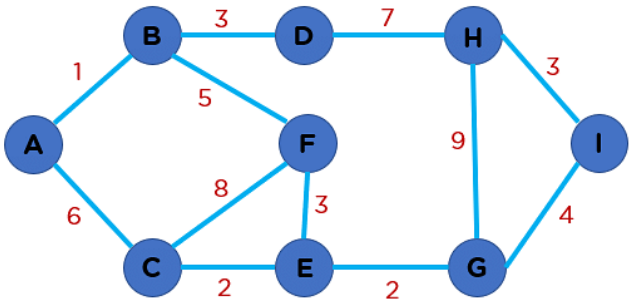
Step 3: Select an edge connecting the tree vertex and fringe vertex having the minimum weight.

Step 4: Add the chosen edge to MST if it doesn't form any closed cycle.

Step 5: Exit

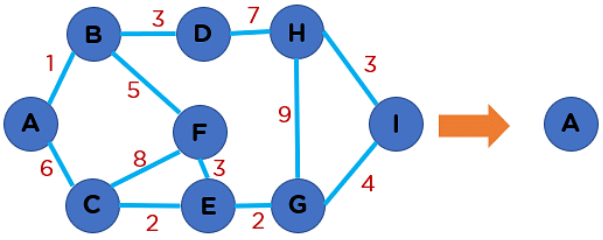
Using the steps mentioned above, we are supposed to generate a minimum spanning tree structure. Let's have a look at an example to understand this process better.

Graph G(V, E) given below contains 9 vertices and 12 edges. We are supposed to create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 9 and edges will be 8 (9-1).



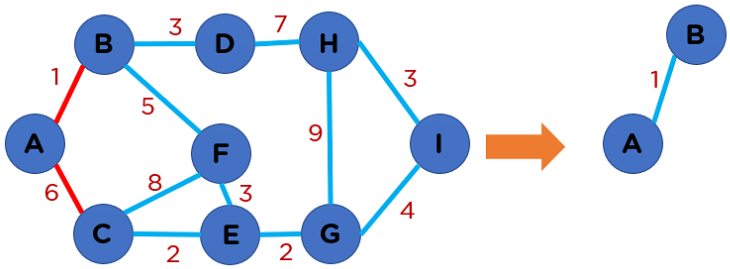
Graph G(V, E)

Primarily, to begin with the creation of MST, you will choose an arbitrary starting vertex. Let's say node A is your starting vertex. This means it will be included first in your tree structure.



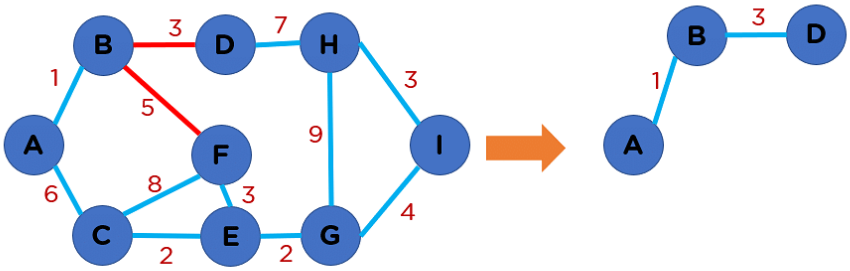
Choose an arbitrary starting vertex.

After the inclusion of node A, you will look into the connected edges going outward from node A and you will pick the one with a minimum edge weight to include it in your T(V', E') structure.



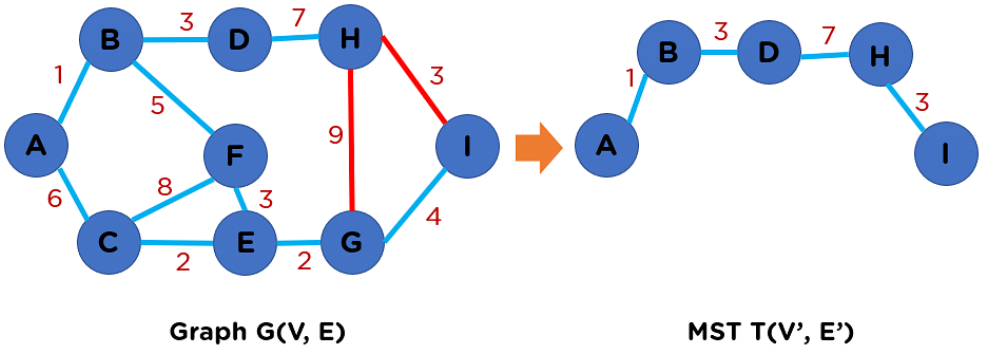
Select the edge with minimal value to include it in MST.

Now, you have reached node B. From node B, there are two possible edges out of which edge BD has the least edge weight value. So, you will include it in your MST.

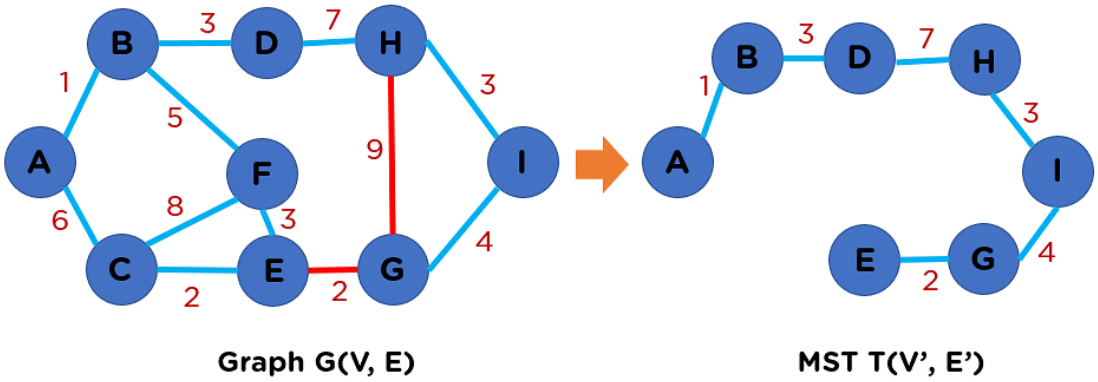


Graph G(V, E) **MST T(V', E')**

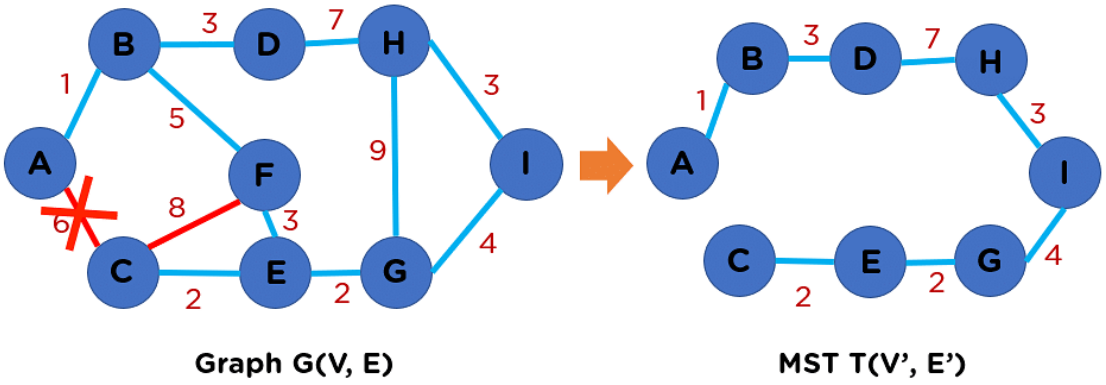
From node D, you only have one edge. So, you will include it in your MST. Further, you have node H, for which you have two incident edges. Out of those two edges, edge HI has the least cost, so you will include it in MST structure.



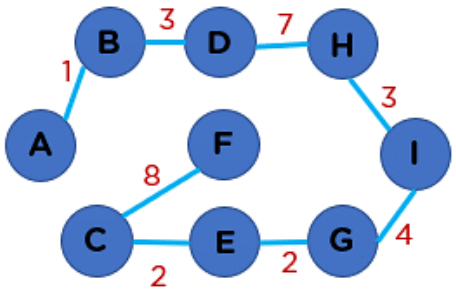
Similarly, the inclusion of nodes G and E will happen in MST.



After that, nodes E and C will get included. Now, from node C, you have two incident edges. Edge CA has the tiniest edge weight. But its inclusion will create a cycle in a tree structure, which you cannot allow. Thus, we will discard edge CA as shown in the image below.



And we will include edge CF in this minimum spanning tree structure.



MST T(V', E')

The summation of all the edge weights in MST T(V', E') is equal to 30, which is the least possible edge weight for any possible spanning tree structure for this particular graph.

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

Creating Minimum Spanning Tree Using Kruskal Algorithm

Step 1: Sort all edges in increasing order of their edge weights.

Step 2: Pick the smallest edge.

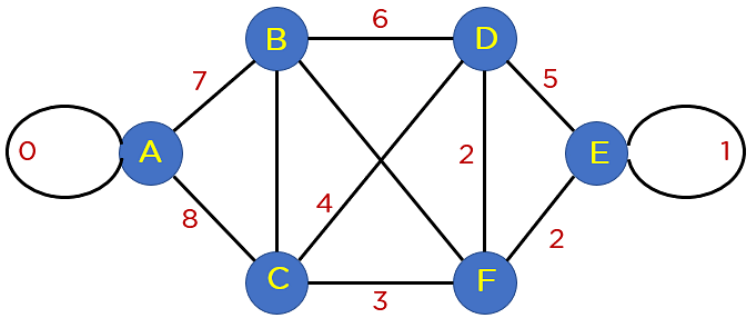
Step 3: Check if the new edge creates a cycle or loop in a spanning tree.

Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.

Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

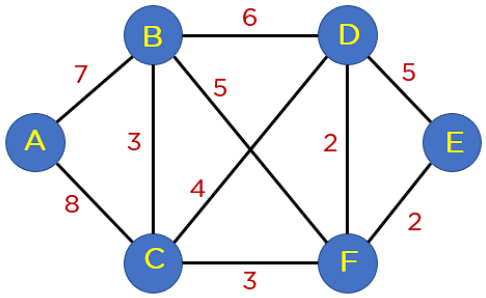
Using the steps mentioned above, you will generate a minimum spanning tree structure. So, now have a look at an example to understand this process better.

The graph G(V, E) given below contains 6 vertices and 12 edges. And you will create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 6 and edges will be 5 (6-1).



Graph G(V, E)

If you observe this graph, you'll find two looping edges connecting the same node to itself again. And you know that the tree structure can never include a loop or parallel edge. Hence, primarily you will need to remove these edges from the graph structure.

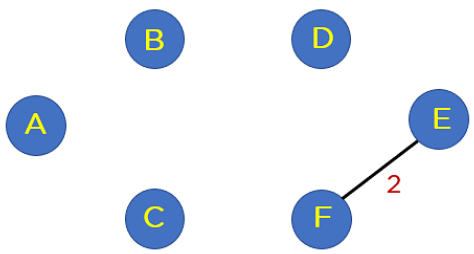


Removing parallel edges or loops from the graph.

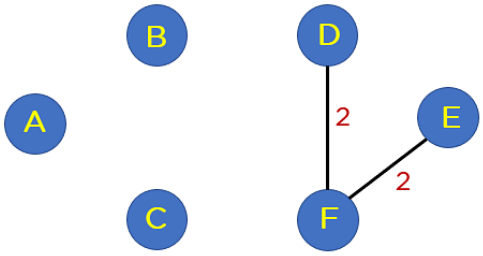
The next step that you will proceed with is arranging all edges in a sorted list by their edge weights.

The Edges of the Graph	Edge Weight	
Source Vertex	Destination Vertex	
E	F	2
F	D	2
B	C	3
C	F	3
C	D	4
B	F	5
B	D	6
A	B	7
A	C	8

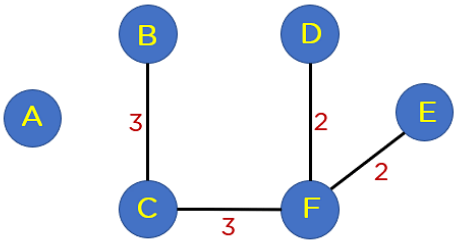
After this step, you will include edges in the MST such that the included edge would not form a cycle in your tree structure. The first edge that you will pick is edge EF, as it has a minimum edge weight that is 2.



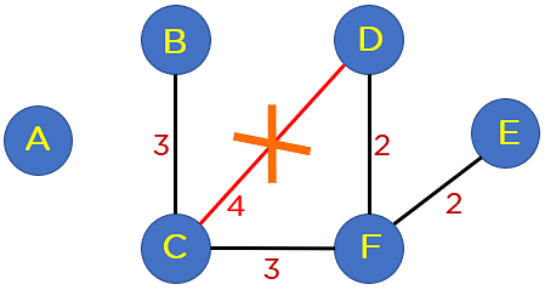
Add edge FD to the spanning tree.



Add edge BC and edge CF to the spanning tree as it does not generate any loop.

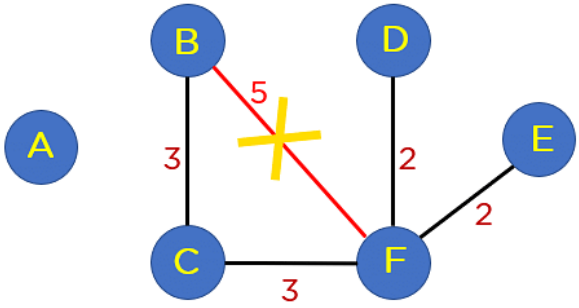


Next up is edge CD. This edge generates the loop in Your tree structure. Thus, you will discard this edge.



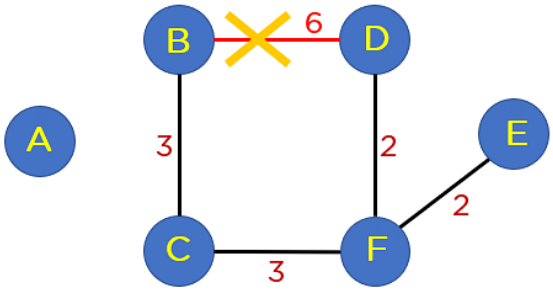
Edge CD should be discarded, as it creates loop.

Following edge CD, you have edge BF. This edge also creates the loop; hence you will discard it.



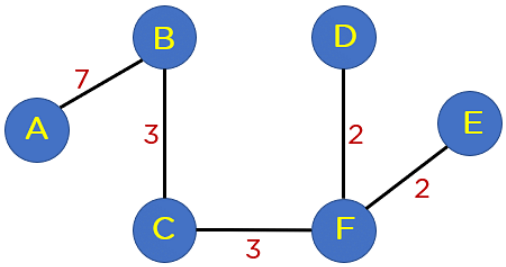
Edge BF should be discarded.

Next up is edge BD. This edge also formulates a loop, so you will discard it as well.



Edge BD should be discarded.

Next on your sorted list is edge AB. This edge does not generate any cycle, so you need not include it in the MST structure. By including this node, it will include 5 edges in the MST, so you don't have to traverse any further in the sorted list. The final structure of your MST is represented in the image below:



Minimum Spanning Tree.

The summation of all the edge weights in MST T(V', E') is equal to 17, which is the least possible edge weight for any possible spanning tree structure for this particular graph.