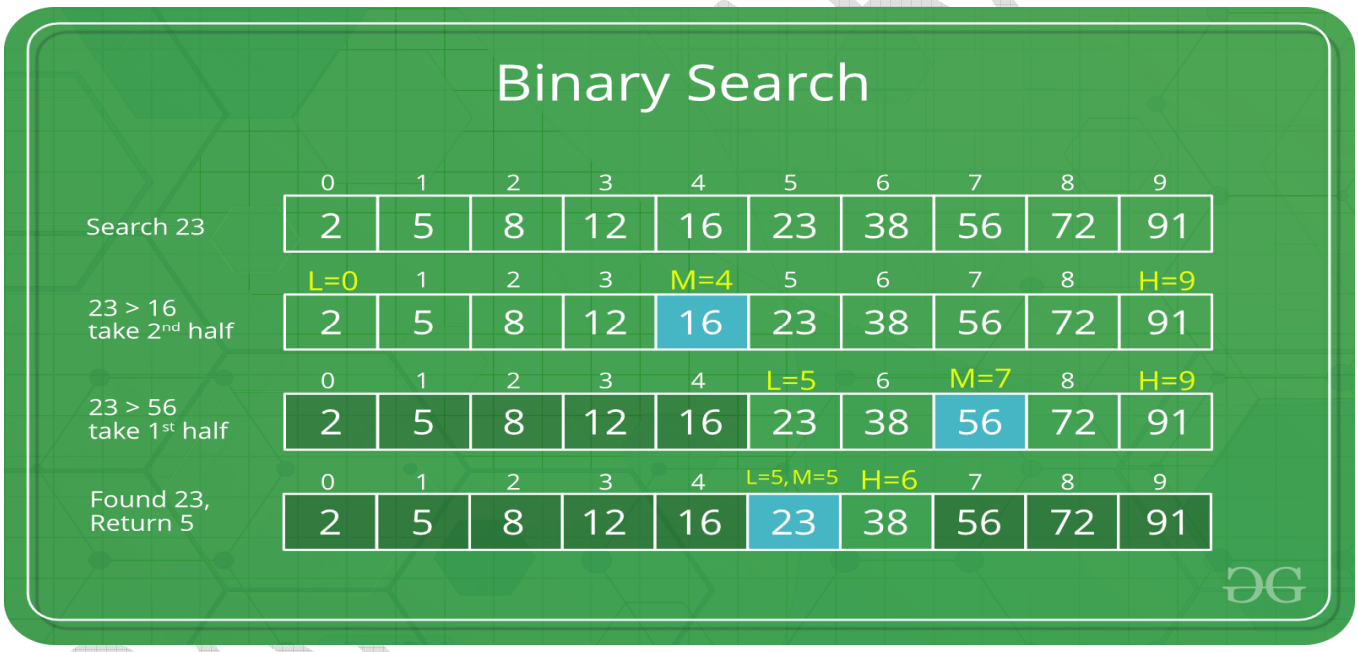
Binary search Algorithm

Class	Search algorithm
Data structure	Array
Worst case performance	*O*(log *n*)
Best case performance	*O*(1)
Average case performance	*O*(log *n*)
Worst case space complexity	*O*(1)

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
A simple approach is to do **linear search.** The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

```
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):
```

```
        # Check base case
        if r >= l:

                mid = l + (r - l)/2

                # If element is present at the middle itself
                if arr[mid] == x:
                        return mid

                # If element is smaller than mid, then it
                # can only be present in left subarray
                elif arr[mid] > x:
                        return binarySearch(arr, l, mid-1, x)

                # Else the element can only be present
                # in right subarray
                else:
                        return binarySearch(arr, mid + 1, r, x)

        else:
                # Element is not present in the array
                return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
        print "Element is present at index % d" % result
else:
        print "Element is not present in array"
```

// C program to implement recursive Binary Search

```
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
        if (r >= l) {
                int mid = l + (r - l) / 2;

                // If the element is present at the middle
                // itself
                if (arr[mid] == x)
                        return mid;

                // If element is smaller than mid, then
                // it can only be present in left subarray
                if (arr[mid] > x)
```

```
                return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array") : printf("Element is present at index %d", result);
    return 0;
}
```

Iterative implementation of Binary Search

```
# Python code to implement iterative Binary
# Search.

# It returns location of x in given array arr
# if present, else returns -1
def binarySearch(arr, l, r, x):

    while l <= r:

        mid = l + (r - l)/2;

        # Check if x is present at mid
        if arr[mid] == x:
            return mid

        # If x is greater, ignore left half
        elif arr[mid] < x:
            l = mid + 1

        # If x is smaller, ignore right half
        else:
            r = mid - 1

    # If we reach here, then the element
    # was not present
    return -1


# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
```

```
result = binarySearch(arr, 0, len(arr)-1, x)
if result != -1:
        print "Element is present at index % d" % result
else:
        print "Element is not present in array"
```

```
// C program to implement iterative Binary Search
#include <stdio.h>
// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
        while (l <= r) {
                int m = l + (r - l) / 2;

                // Check if x is present at mid
                if (arr[m] == x)
                        return m;
        // not present
        return -1;
}

int main(void)
{
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = sizeof(arr) / sizeof(arr[0]);
        int x = 10;
        int result = binarySearch(arr, 0, n - 1, x);
        (result == -1) ? printf("Element is not present in array") : printf("Element is present at index %d", result);
        return 0;
}
```

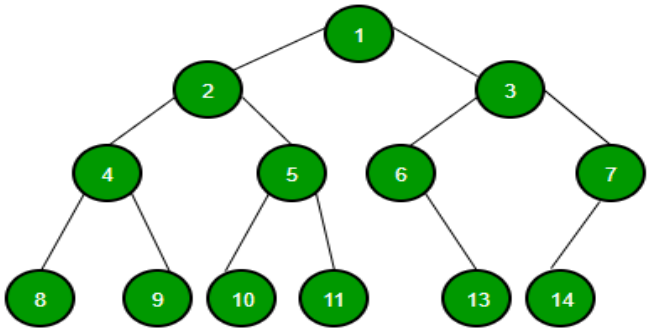
The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of

Master Method and solution of the recurrence is .

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.

2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms

6. Form of a multi-stage decision-making (see business chess).

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

This can be proved by induction.

For root, l = 1, number of nodes = $2^{1-1} = 1$

Assume that maximum number of nodes on level l is 2^{l-1}

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$.

Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h – 1$.

In some books, height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} – 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is ? $Log_2(N+1)$?

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes ? $Log_2(N+1)$? – 1

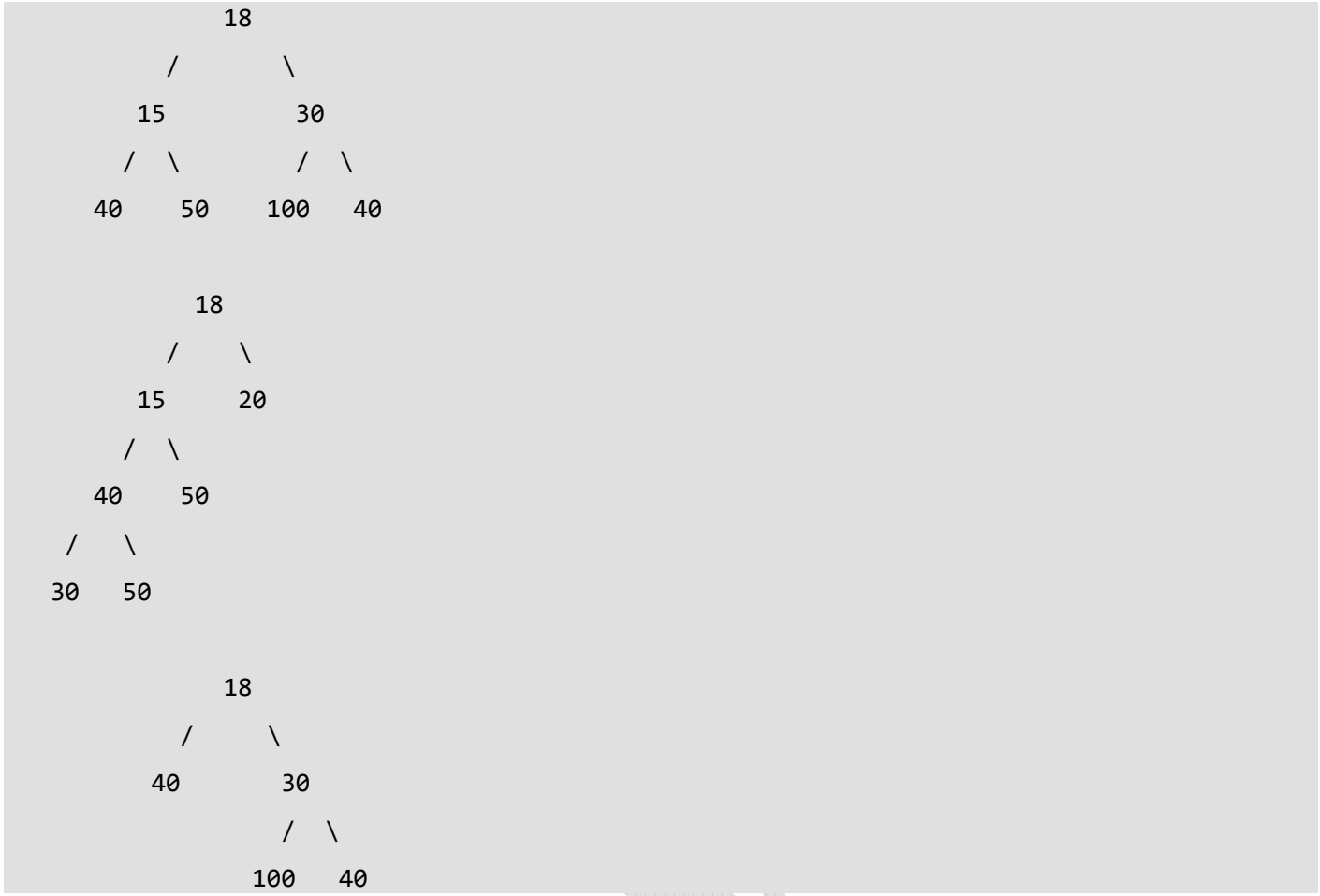
4) A Binary Tree with L leaves has at least ? Log_2L ? + 1 levels

A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

```
L = T + 1

Where L = Number of leaf nodes

      T = Number of internal nodes with two children
```

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



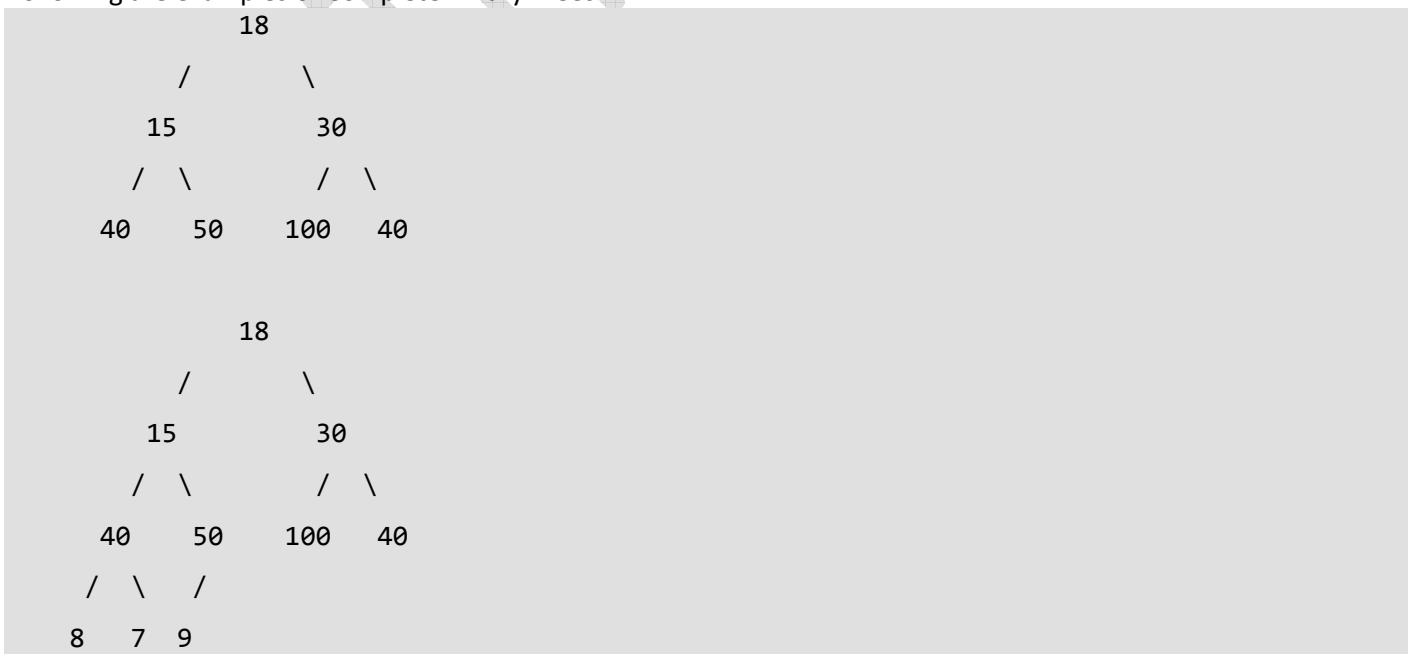
n a Full Binary, number of leaf nodes is number of internal nodes plus 1

L = I + 1

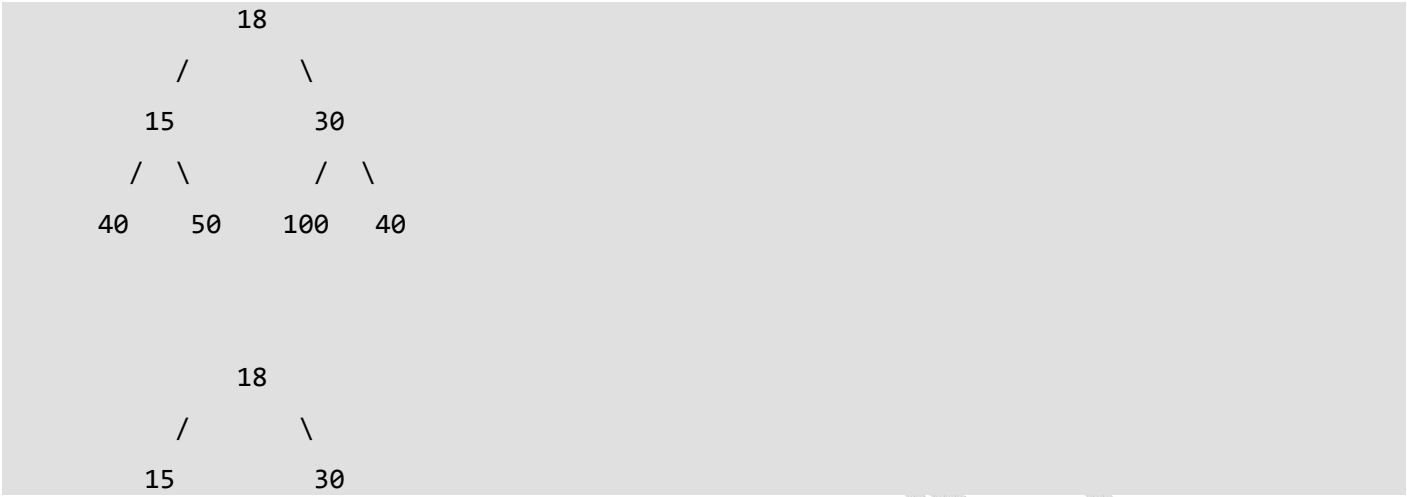
Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees



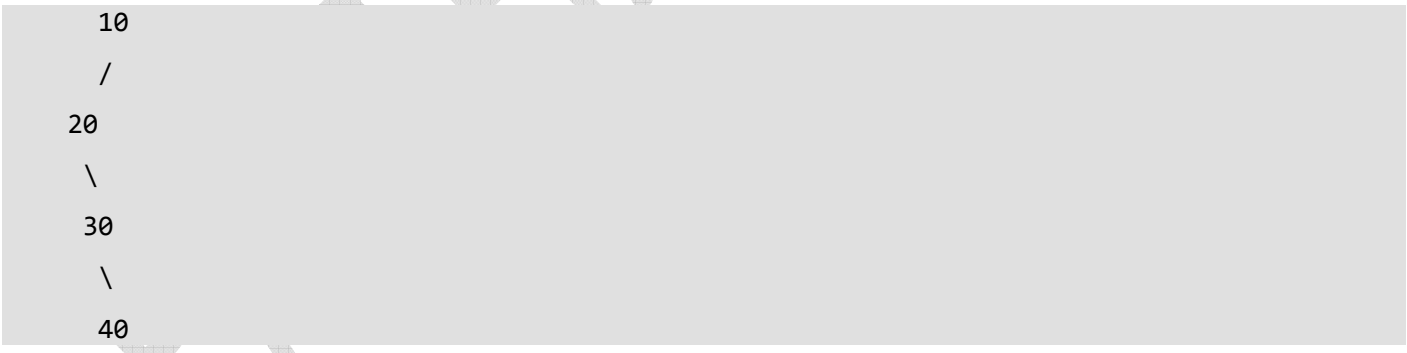
Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.
Following are examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.
Example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

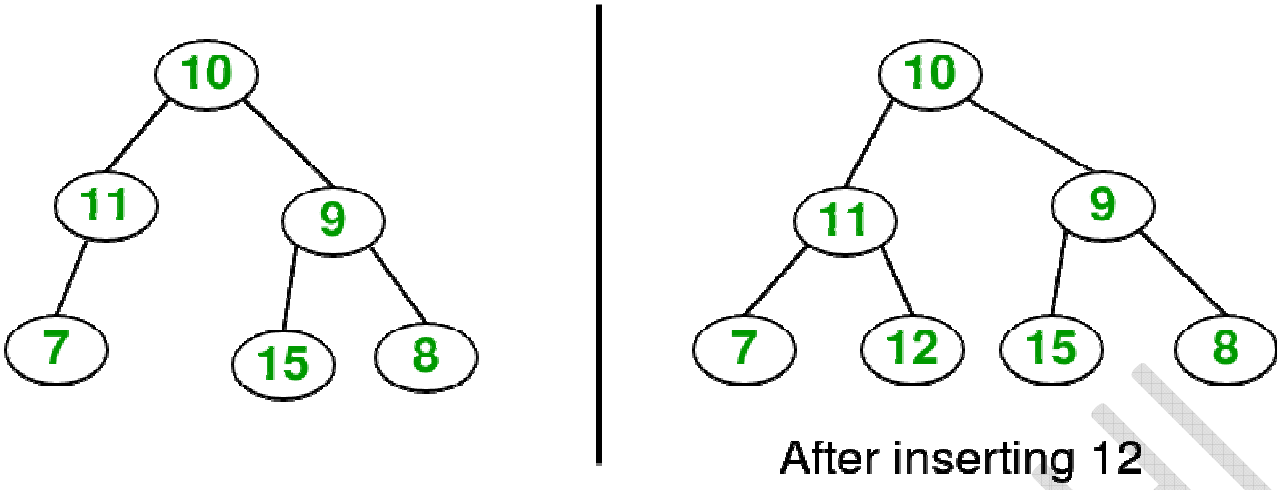
Balanced Binary Tree
A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, AVL tree maintains O(Log n) height by making sure that the difference between heights of left and right subtrees is atmost 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.



Insertion in a Binary Tree in level order

Given a binary tree and a key, insert the key into the binary tree at first position available in level order.



The idea is to do iterative level order traversal of the given tree using queue. If we find a node whose left child is empty, we make new key as left child of the node. Else if we find a node whose right child is empty, we make new key as right child. We keep traversing the tree until we find a node whose either left or right is empty.

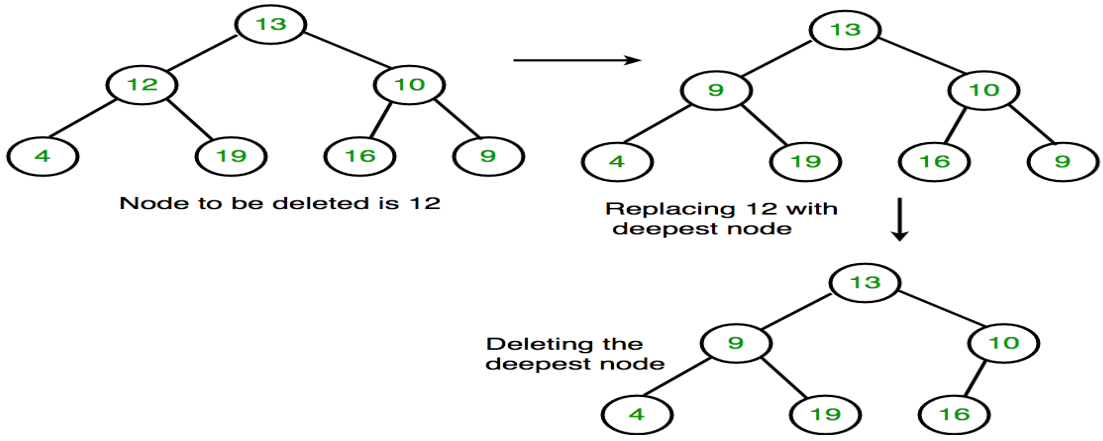
Deletion in a Binary Tree

Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by bottom most and rightmost node). This different from BST deletion. Here we do not have any order among elements, so we replace with last element.
Examples :

```
Delete 10 in below tree
       10
     /    \
    20     30
Output :
       30
     /
    20


Delete 20 in below tree
       10
     /    \
    20     30
             \
              40
Output :
       10
     /   \
    40    30
```

Algorithm
1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.

2. Replace the deepest rightmost node's data with node to be deleted.
3. Then delete the deepest rightmost node.

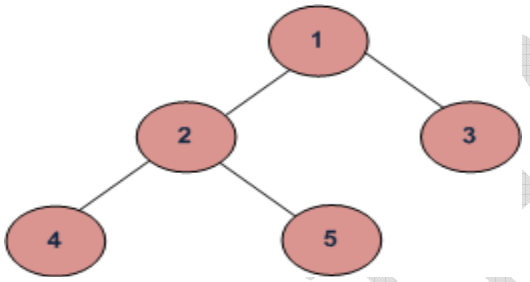


BFS vs DFS for Binary Tree

What are BFS and DFS for Binary Tree?

A Tree is typically traversed in two ways:

- Breadth First Traversal (Or Level Order Traversal)
- Depth First Traversals
  - Inorder Traversal (Left-Root-Right)
  - Preorder Traversal (Root-Left-Right)
  - Postorder Traversal (Left-Right-Root)




```
BFS and DFSs of above Tree


Breadth First Traversal : 1 2 3 4 5
Depth First Traversals:
      Preorder Traversal : 1 2 4 5 3
      Inorder Traversal  :  4 2 5 1 3
      Postorder Traversal : 4 5 2 3 1
```

Breadth First Traversal

Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
```

```
for d = 1 to height(tree)
   printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Inorder Traversal (Practice):

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal (Practice):

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Uses of Postorder
Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Is there any difference in terms of Time Complexity?
All four traversals require O(n) time as they visit every node exactly once.
Is there any difference in terms of Extra Space?
There is difference in terms of extra space required.

1. Extra Space required for Level Order Traversal is O(w) where w is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.
2. Extra Space required for Depth First Traversals is O(h) where h is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Maximum Width of a Binary Tree at depth (or height) h can be 2^h where h starts from 0. So the maximum number of nodes can be at the last level. And worst case occurs when Binary Tree is a perfect Binary Tree with numbers of nodes like 1, 3, 7, 15, …etc. In worst case, value of 2^h is **Ceil(n/2)**.
Height for a Balanced Binary Tree is O(Log n). Worst case occurs for skewed tree and worst case height becomes O(n).
So in worst case extra space required is O(n) for both. But worst cases occur for different types of trees.

```
# Python program to insert element in binary tree
class newNode():
    def __init__(self, data):
        self.key = data
        self.left = None
        self.right = None

# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)



# A function to do postorder tree traversal
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),


# A function to do preorder tree traversal
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val),

        # Then recur on left child
```

```
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)


"""function to insert element in binary tree """
def insert(temp,key):

    q = []
    q.append(temp)

    # Do level order traversal until we find
    # an empty place.
    while (len(q)):
        temp = q[0]
        q.pop(0)

        if (not temp.left):
            temp.left = newNode(key)
            break
        else:
            q.append(temp.left)

        if (not temp.right):
            temp.right = newNode(key)
            break
        else:
            q.append(temp.right)

# function to delete the given deepest node (d_node) in binary tree
def deleteDeepest(root,d_node):
    q = []
    q.append(root)
    while(len(q)):
        temp = q.pop(0)
        if temp is d_node:
            temp = None
            return
        if temp.right:
            if temp.right is d_node:
                temp.right = None
                return
            else:
                q.append(temp.right)
        if temp.left:
            if temp.left is d_node:
                temp.left = None
                return
            else:
                q.append(temp.left)

# function to delete element in binary tree
def deletion(root, key):
    if root == None :
        return None
    if root.left == None and root.right == None:
        if root.key == key :
            return None
        else :
            return root
    key_node = None
    q = []
```

```
    q.append(root)
    while(len(q)):
        temp = q.pop(0)
        if temp.data == key:
            key_node = temp
        if temp.left:
            q.append(temp.left)
        if temp.right:
            q.append(temp.right)
    if key_node :
        x = temp.data
        deleteDeepest(root,temp)
        key_node.data = x
    return root


# Driver code
if __name__ == '__main__':
      root = newNode(10)
      root.left = newNode(11)
      root.left.left = newNode(7)
      root.right = newNode(9)
      root.right.left = newNode(15)
      root.right.right = newNode(8)

      print "Preorder traversal of binary tree is"
      printPreorder(root)

      print "\nInorder traversal of binary tree is"
      printInorder(root)

      print "\nPostorder traversal of binary tree is"
      printPostorder(root)


      key = 12
      insert(root, key)

      print "Preorder traversal of binary tree is"
      printPreorder(root)

      print "\nInorder traversal of binary tree is"
      printInorder(root)

      print "\nPostorder traversal of binary tree is"
      printPostorder(root)

      key = 11
      root = deletion(root, key)

      print "Preorder traversal of binary tree is"
      printPreorder(root)

      print "\nInorder traversal of binary tree is"
      printInorder(root)

      print "\nPostorder traversal of binary tree is"
      printPostorder(root)


# Recursive Python program for level order traversal of Binary Tree
```

```
# A node structure
class Node:

    # A utility function to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None


# Function to  print level order traversal of tree
def printLevelOrder(root):
    h = height(root)
    for i in range(1, h+1):
        printGivenLevel(root, i)


# Print nodes at a given level
def printGivenLevel(root , level):
    if root is None:
        return
    if level == 1:
        print "%d" %(root.data),
    elif level > 1 :
        printGivenLevel(root.left , level-1)
        printGivenLevel(root.right , level-1)


""" Compute the height of a tree--the number of nodes
    along the longest path from the root node down to
    the farthest leaf node
"""
def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level order traversal of binary tree is -"
printLevelOrder(root)

#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -

1 2 3 4 5
```

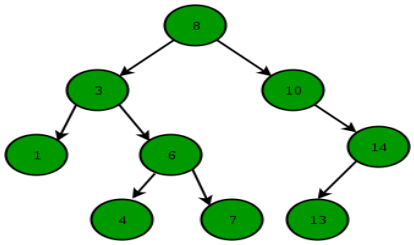
Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

Binary search tree

Binary search tree		
Type	Tree	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

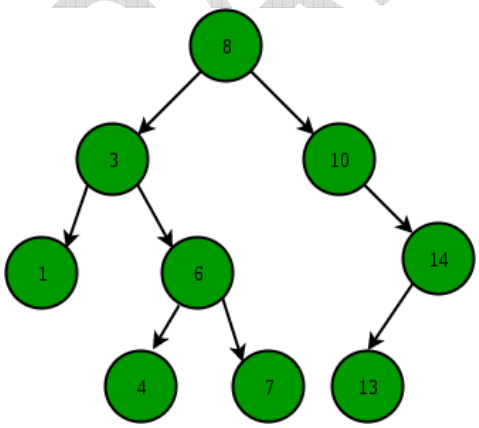
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



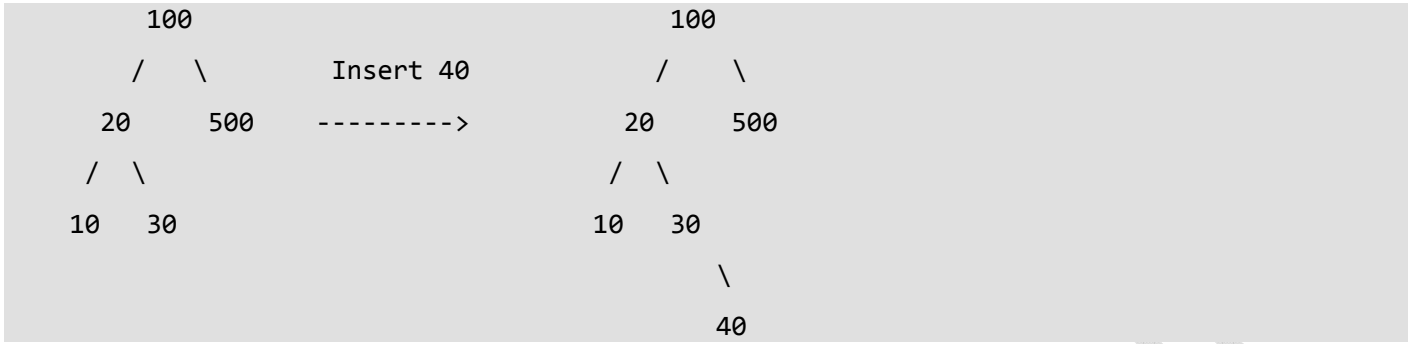
The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.
To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Illustration to search 6 in below tree:
1. Start from root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If element to search is found anywhere, return true, else return false.



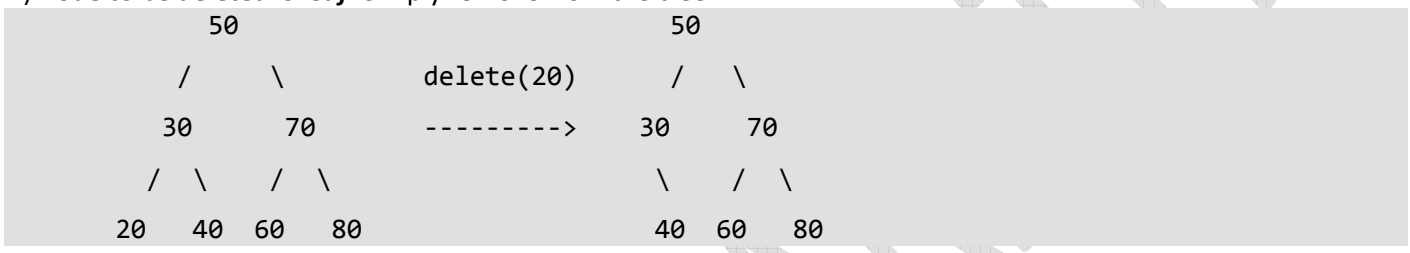
Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

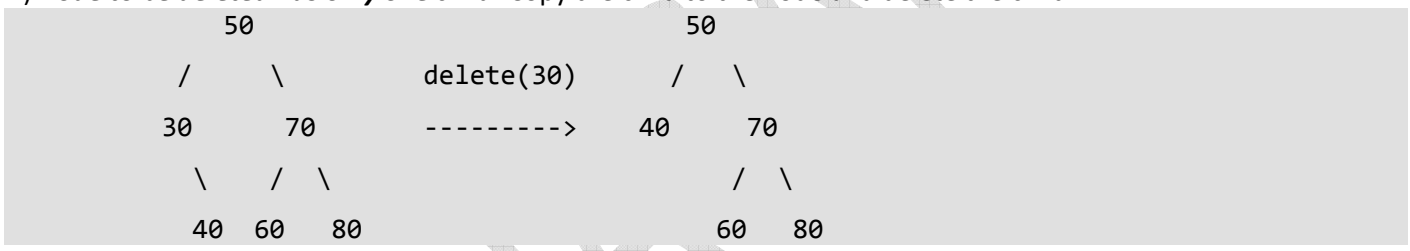


Binary Search Tree When we delete a node, three possibilities arise.

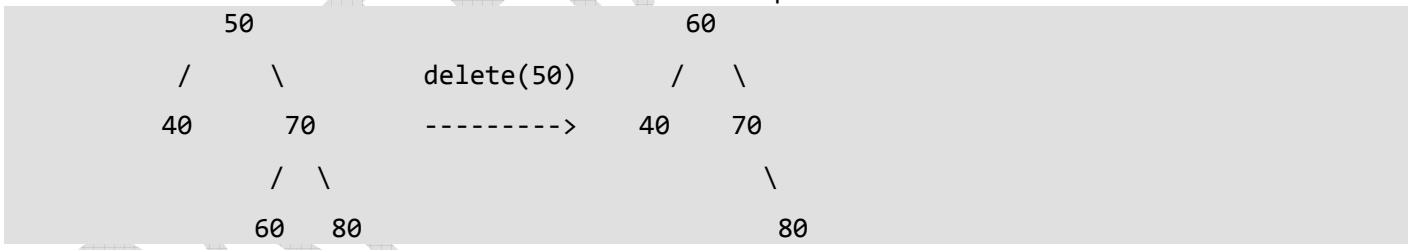
1) ***Node to be deleted is leaf****:* Simply remove from the tree.



2) ***Node to be deleted has only one child****:* Copy the child to the node and delete the child.



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

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

// A utility function to create a new BST 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;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
	if (root != NULL)
	{
		inorder(root->left);
		printf("%d ", root->key);
		inorder(root->right);
	}
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
	/* If the tree is empty, return a new node */
	if (node == NULL) return newNode(key);

	/* Otherwise, recur down the tree */
	if (key < node->key)
		node->left = insert(node->left, key);
	else
		node->right = insert(node->right, key);

	/* return the (unchanged) node pointer */
	return node;
}

/* Given a non-empty binary search tree, return the node with minimum
key value found in that tree. Note that the entire tree does not
need to be searched. */
struct node * minValueNode(struct node* node)
{
	struct node* current = node;

	/* loop down to find the leftmost leaf */
	while (current && current->left != NULL)
		current = current->left;

	return current;
}

/* Given a binary search tree and a key, this function deletes the key
and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
	// base case
	if (root == NULL) return root;
```

```
	// If the key to be deleted is smaller than the root's key,
	// then it lies in left subtree
	if (key < root->key)
		root->left = deleteNode(root->left, key);

	// If the key to be deleted is greater than the root's key,
	// then it lies in right subtree
	else if (key > root->key)
		root->right = deleteNode(root->right, key);

	// if key is same as root's key, then This is the node
	// to be deleted
	else
	{
		// node 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;
		}

		// node with two children: Get the inorder successor (smallest
		// in the right subtree)
		struct node* temp = minValueNode(root->right);

		// Copy the inorder successor's content to this node
		root->key = temp->key;

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

// Driver Program to test above functions
int main()
{
	/* Let us create following BST
		  50
		 /  \
		30   70
		/ \  / \
	       20 40 60 80 */
	struct node *root = NULL;
	root = insert(root, 50);
	root = insert(root, 30);
	root = insert(root, 20);
	root = insert(root, 40);
```

```
        root = insert(root, 70);
        root = insert(root, 60);
        root = insert(root, 80);

        printf("Inorder traversal of the given tree \n");
        inorder(root);

        printf("\nDelete 20\n");
        root = deleteNode(root, 20);
        printf("Inorder traversal of the modified tree \n");
        inorder(root);

        printf("\nDelete 30\n");
        root = deleteNode(root, 30);
        printf("Inorder traversal of the modified tree \n");
        inorder(root);

        printf("\nDelete 50\n");
        root = deleteNode(root, 50);
        printf("Inorder traversal of the modified tree \n");
        inorder(root);

        return 0;
}

# Python program to demonstrate delete operation
# in binary search tree

# A Binary Tree Node
class Node:

        # Constructor to create a new node
        def __init__(self, key):
                self.key = key
                self.left = None
                self.right = None


# A utility function to do inorder traversal of BST
def inorder(root):
        if root is not None:
                inorder(root.left)
                print root.key,
                inorder(root.right)


# A utility function to insert a new node with given key in BST
def insert( node, key):

        # If the tree is empty, return a new node
        if node is None:
                return Node(key)

        # Otherwise recur down the tree
        if key < node.key:
```

```
                node.left = insert(node.left, key)
        else:
                node.right = insert(node.right, key)

        # return the (unchanged) node pointer
        return node

# Given a non-empty binary search tree, return the node
# with minum key value found in that tree. Note that the
# entire tree does not need to be searched
def minValueNode( node):
        current = node

        # loop down to find the leftmost leaf
        while(current.left is not None):
                current = current.left

        return current

# Given a binary search tree and a key, this function
# delete the key and returns the new root
def deleteNode(root, key):

        # Base Case
        if root is None:
                return root

        # If the key to be deleted is smaller than the root's
        # key then it lies in left subtree
        if key < root.key:
                root.left = deleteNode(root.left, key)

        # If the kye to be delete is greater than the root's key
        # then it lies in right subtree
        elif(key > root.key):
                root.right = deleteNode(root.right, key)

        # If key is same as root's key, then this is the node
        # to be deleted
        else:

                # Node with only one child or no child
                if root.left is None :
                        temp = root.right
                        root = None
                        return temp

                elif root.right is None :
                        temp = root.left
                        root = None
                        return temp

                # Node with two children: Get the inorder successor
                # (smallest in the right subtree)
                temp = minValueNode(root.right)
```

```
                # Copy the inorder successor's content to this node
                root.key = temp.key

                # Delete the inorder successor
                root.right = deleteNode(root.right , temp.key)


        return root

# Driver program to test above functions
""" Let us create following BST
                        50
                       /  \
                     30   70
                     / \   / \
                   20 40 60 80 """

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print "Inorder traversal of the given tree"
inorder(root)

print "\nDelete 20"
root = deleteNode(root, 20)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 30"
root = deleteNode(root, 30)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 50"
root = deleteNode(root, 50)
print "Inorder traversal of the modified tree"
inorder(root)
```

Advantages of BST over Hash Table

Hash Table supports following operations in Θ(1) time.

1) Search
2) Insert
3) Delete

The time complexity of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn).

So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables, what are advantages. Following are some important points in favor of BSTs.

1. We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
2. Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
3. BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.
4. With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

Construct BST from given preorder traversal

Given preorder traversal of a binary search tree, construct the BST.
For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.

```
     10
   /   \
  5     40
 /  \      \
1    7      50
```

Method 1 (O(n^2) time complexity)

The first element of preorder traversal is always root. We first construct the root. Then we find the index of first element which is greater than root. Let the index be 'i'. The values between root and 'i' will be part of left subtree, and the values between 'i+1' and 'n-1' will be part of right subtree. Divide given pre[] at index "i" and recur for left and right sub-trees.
For example in {10, 5, 1, 7, 40, 50}, 10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.

```
              10
            /    \
           /      \
  {5, 1, 7}       {40, 50}
```

We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

```
# A O(n^2) Python3 program for construction of BST from preorder traversal

# A binary tree node
class Node():

        # A constructor to create a new node
        def __init__(self, data):
                self.data = data
                self.left = None
                self.right = None


# constructTreeUtil.preIndex is a static variable of
# function constructTreeUtil

# Function to get the value of static variable
# constructTreeUtil.preIndex
def getPreIndex():
        return constructTreeUtil.preIndex
```

```
# Function to increment the value of static variable
# constructTreeUtil.preIndex
def incrementPreIndex():
        constructTreeUtil.preIndex += 1

# A recurseive function to construct Full from pre[].
# preIndex is used to keep track of index in pre[[].
def constructTreeUtil(pre, low, high, size):

        # Base Case
        if( getPreIndex() >= size or low > high):
                return None

        # The first node in preorder traversal is root. So take
        # the node at preIndex from pre[] and make it root,
        # and increment preIndex
        root = Node(pre[getPreIndex()])
        incrementPreIndex()

        # If the current subarray has onlye one element,
        # no need to recur
        if low == high :
                return root

        # Search for the first element greater than root
        for i in range(low, high+1):
                if (pre[i] > root.data):
                        break

        # Use the index of element found in preorder to divide
        # preorder array in two parts. Left subtree and right
        # subtree
        root.left = constructTreeUtil(pre, getPreIndex(), i-1 , size)

        root.right = constructTreeUtil(pre, i, high, size)

        return root

# The main function to construct BST from given preorder
# traversal. This function mailny uses constructTreeUtil()
def constructTree(pre):
        size = len(pre)
        constructTreeUtil.preIndex = 0
        return constructTreeUtil(pre, 0, size-1, size)


def printInorder(root):
        if root is None:
                return
        printInorder(root.left)
        print root.data,
        printInorder(root.right)


# Driver program to test above function
```

```
pre = [10, 5, 1, 7, 40, 50]

root = constructTree(pre)
print "Inorder traversal of the constructed tree:"
printInorder(root)
```

Method 2 (O(n) time complexity)
The idea used here is inspired from method 3 of this post. The trick is to set a range {min .. max} for every node. Initialize the range as {INT_MIN .. INT_MAX}. The first node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT_MIN …root->data}. If a values is in the range {INT_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data..max .. INT_MAX}.

```
/* A O(n) program for construction
of BST from preorder traversal */
#include <bits/stdc++.h>
using namespace std;


/* A binary tree node has data, pointer to left child
and a pointer to right child */
class node
{
        public:
        int data;
        node *left;
        node *right;
};

// A utility function to create a node
node* newNode (int data)
{
        node* temp = new node();

        temp->data = data;
        temp->left = temp->right = NULL;

        return temp;
}

// A recursive function to construct
// BST from pre[]. preIndex is used
// to keep track of index in pre[].
node* constructTreeUtil( int pre[], int* preIndex, int key,
                                                        int min, int max, int size )
{
        // Base case
        if( *preIndex >= size )
                return NULL;

        node* root = NULL;

        // If current element of pre[] is in range, then
        // only it is part of current subtree
        if( key > min && key < max )
```

```
        {
                // Allocate memory for root of this
                // subtree and increment *preIndex
                root = newNode ( key );
                *preIndex = *preIndex + 1;

                if (*preIndex < size)
                {
                        // Contruct the subtree under root
                        // All nodes which are in range
                        // {min .. key} will go in left
                        // subtree, and first such node
                        // will be root of left subtree.
                        root->left = constructTreeUtil( pre, preIndex, pre[*preIndex],
                                                                        min, key, size );

                        // All nodes which are in range
                        // {key..max} will go in right
                        // subtree, and first such node
                        // will be root of right subtree.
                        root->right = constructTreeUtil( pre, preIndex, pre[*preIndex],
                                                                        key, max, size );
                }
        }

        return root;
}

// The main function to construct BST
// from given preorder traversal.
// This function mainly uses constructTreeUtil()
node *constructTree (int pre[], int size)
{
        int preIndex = 0;
        return constructTreeUtil ( pre, &preIndex, pre[0], INT_MIN,
                                                        INT_MAX, size );
}

// A utility function to print inorder
// traversal of a Binary Tree
void printInorder (node* node)
{
        if (node == NULL)
                return;
        printInorder(node->left);
        cout << node->data << " ";
        printInorder(node->right);
}

// Driver code
int main ()
{
        int pre[] = {10, 5, 1, 7, 40, 50};
        int size = sizeof( pre ) / sizeof( pre[0] );
```

```
        node *root = constructTree(pre, size);

        cout << "Inorder traversal of the constructed tree: \n";
        printInorder(root);

        return 0;
}

# A O(n) program for construction of BST from preorder traversal

INT_MIN = float("-infinity")
INT_MAX = float("infinity")

# A Binary tree node
class Node:

        # Constructor to created a new node
        def __init__(self, data):
                self.data = data
                self.left = None
                self.right = None

# Methods to get and set the value of static variable
# constructTreeUtil.preIndex for function construcTreeUtil()
def getPreIndex():
        return constructTreeUtil.preIndex

def incrementPreIndex():
        constructTreeUtil.preIndex += 1

# A recursive function to construct BST from pre[].
# preIndex is used to keep track of index in pre[]
def constructTreeUtil(pre, key, mini, maxi, size):

        # Base Case
        if(getPreIndex() >= size):
                return None

        root = None

        # If current element of pre[] is in range, then
        # only it is part of current subtree
        if(key > mini and key < maxi):

                # Allocate memory for root of this subtree
                # and increment constructTreeUtil.preIndex
                root = Node(key)
                incrementPreIndex()

                if(getPreIndex() < size):

                        # Construct the subtree under root
                        # All nodes which are in range {min.. key} will
                        # go in left subtree, and first such node will
                        # be root of left subtree
```

```
                root.left = constructTreeUtil(pre,
                                        pre[getPreIndex()], mini, key, size)

                # All nodes which are in range{key..max} will
                # go to right subtree, and first such node will
                # be root of right subtree
                root.right = constructTreeUtil(pre,
                                pre[getPreIndex()], key, maxi, size)

        return root

# This is the main function to construct BST from given
# preorder traversal. This function mainly uses
# constructTreeUtil()
def constructTree(pre):
        constructTreeUtil.preIndex = 0
        size = len(pre)
        return constructTreeUtil(pre, pre[0], INT_MIN, INT_MAX, size)


# A utility function to print inorder traversal of Binary Tree
def printInorder(node):

        if node is None:
                return
        printInorder(node.left)
        print node.data,
        printInorder(node.right)


# Driver program to test above function
pre = [10, 5, 1, 7, 40, 50]
root = constructTree(pre)

print "Inorder traversal of the constructed tree: "
printInorder(root)
```

Merge Two Balanced Binary Search Trees

You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take O(m+n) time.

In the following solutions, it is assumed that sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See this).

Method 1 (Insert elements of first tree to second)

Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See this) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)

1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.

2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.

3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.

4) Construct a balanced tree from the merged array using the technique discussed in this post. This step takes O(m+n) time.
Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.
Following is implementation of this method.

```
// C program to Merge Two Balanced Binary Search Trees
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
        int data;
        struct node* left;
        struct node* right;
};

// A utility unction to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder traversal of a tree in inorder array
void storeInorder(struct node* node, int inorder[], int *index_ptr);

/* A function that constructs Balanced Binary Search Tree from a sorted array
See https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/ */
struct node* sortedArrayToBST(int arr[], int start, int end);

/* This function merges two balanced BSTs with roots as root1 and root2.
m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m, int n)
{
        // Store inorder traversal of first tree in an array arr1[]
        int *arr1 = new int[m];
        int i = 0;
        storeInorder(root1, arr1, &i);

        // Store inorder traversal of second tree in another array arr2[]
        int *arr2 = new int[n];
        int j = 0;
        storeInorder(root2, arr2, &j);

        // Merge the two sorted array into one
        int *mergedArr = merge(arr1, arr2, m, n);

        // Construct a tree from the merged array and return root of the tree
        return sortedArrayToBST (mergedArr, 0, m+n-1);
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
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);
}

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
        if (node == NULL)
                return;

        /* first recur on left child */
        printInorder(node->left);

        printf("%d ", node->data);

        /* now recur on right child */
        printInorder(node->right);
}

// A utility unction to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
        // mergedArr[] is going to contain result
        int *mergedArr = new int[m + n];
        int i = 0, j = 0, k = 0;

        // Traverse through both arrays
        while (i < m && j < n)
        {
                // Pick the smaler element and put it in mergedArr
                if (arr1[i] < arr2[j])
                {
                        mergedArr[k] = arr1[i];
                        i++;
                }
                else
                {
                        mergedArr[k] = arr2[j];
                        j++;
                }
                k++;
        }

        // If there are more elements in first array
        while (i < m)
        {
                mergedArr[k] = arr1[i];
                i++; k++;
        }

        // If there are more elements in second array
```

```
        while (j < n)
        {
                mergedArr[k] = arr2[j];
                j++; k++;
        }

        return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
        if (node == NULL)
                return;

        /* first recur on left child */
        storeInorder(node->left, inorder, index_ptr);

        inorder[*index_ptr] = node->data;
        (*index_ptr)++; // increase index for next entry

        /* now recur on right child */
        storeInorder(node->right, inorder, index_ptr);
}

/* A function that constructs Balanced Binary Search Tree from a sorted array
See https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/ */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
        /* Base Case */
        if (start > end)
        return NULL;

        /* Get the middle element and make it root */
        int mid = (start + end)/2;
        struct node *root = newNode(arr[mid]);

        /* Recursively construct the left subtree and make it
        left child of root */
        root->left = sortedArrayToBST(arr, start, mid-1);

        /* Recursively construct the right subtree and make it
        right child of root */
        root->right = sortedArrayToBST(arr, mid+1, end);

        return root;
}

/* Driver program to test above functions*/
int main()
{
        /* Create following tree as first balanced BST
                100
                 / \
               50 300
```

```
                / \
               20  70
        */
        struct node *root1 = newNode(100);
        root1->left                = newNode(50);
        root1->right       = newNode(300);
        root1->left->left = newNode(20);
        root1->left->right = newNode(70);

        /* Create following tree as second balanced BST
                 80
                 / \
               40 120
        */
        struct node *root2 = newNode(80);
        root2->left                = newNode(40);
        root2->right       = newNode(120);

        struct node *mergedTree = mergeTrees(root1, root2, 5, 3);

        printf ("Following is Inorder traversal of the merged tree \n");
        printInorder(mergedTree);

        getchar();
        return 0;
}
```

Inorder predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question: There is BST given with root node with key part as integer only. The structure of each node is as follows:

```
struct Node
{
        int key;
        struct Node *left, *right ;
};
```

Following is the algorithm to reach the desired result. Its a recursive method:

```
Input: root node, key
output: predecessor node, successor node


1. If root is NULL
      then return
2. if key is found then
    a. If its left subtree is not null
        Then predecessor will be the right most
        child of left subtree or left child itself.
    b. If its right subtree is not null
        The successor will be the left most child
        of right subtree or right child itself.
    return
3. If key is smaller then root node
```

```
        set the successor as root
        search recursively into left subtree
    else
        set the predecessor as root
        search recursively into right subtree
```

```
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
        int key;
        struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
        // Base case
        if (root == NULL) return ;

        // If key is present at root
        if (root->key == key)
        {
                // the maximum value in left subtree is predecessor
                if (root->left != NULL)
                {
                        Node* tmp = root->left;
                        while (tmp->right)
                                tmp = tmp->right;
                        pre = tmp ;
                }

                // the minimum value in right subtree is successor
                if (root->right != NULL)
                {
                        Node* tmp = root->right ;
                        while (tmp->left)
                                tmp = tmp->left ;
                        suc = tmp ;
                }
                return ;
        }

        // If key is smaller than root's key, go to left subtree
        if (root->key > key)
        {
                suc = root ;
                findPreSuc(root->left, pre, suc, key) ;
        }
        else // go to right subtree
```

```
        {
                pre = root ;
                findPreSuc(root->right, pre, suc, key) ;
        }
}

// A utility function to create a new BST node
Node *newNode(int item)
{
        Node *temp = new Node;
        temp->key = item;
        temp->left = temp->right = NULL;
        return temp;
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
        if (node == NULL) return newNode(key);
        if (key < node->key)
                node->left = insert(node->left, key);
        else
                node->right = insert(node->right, key);
        return node;
}

// Driver program to test above function
int main()
{
        int key = 65; //Key to be searched in BST

/* Let us create following BST
                        50
                /         \
                30        70
                / \ / \
        20 40 60 80 */
        Node *root = NULL;
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);


        Node* pre = NULL, *suc = NULL;

        findPreSuc(root, pre, suc, key);
        if (pre != NULL)
        cout << "Predecessor is " << pre->key << endl;
        else
        cout << "No Predecessor";
```

```
        if (suc != NULL)
        cout << "Successor is " << suc->key;
        else
        cout << "No Successor";
        return 0;
}

# Python program to find predecessor and successor in a BST

# A BST node
class Node:

        # Constructor to create a new node
        def __init__(self, key):
                self.key = key
                self.left = None
                self.right = None

# This function finds predecessor and successor of key in BST
# It sets pre and suc as predecessor and successor respectively
def findPreSuc(root, key):

        # Base Case
        if root is None:
                return
        # If key is present at root
        if root.key == key:
                # the maximum value in left subtree is predecessor
                if root.left is not None:
                        tmp = root.left
                        while(tmp.right):
                                tmp = tmp.right
                        findPreSuc.pre = tmp
                # the minimum value in right subtree is successor
                if root.right is not None:
                        tmp = root.right
                        while(temp.left):
                                tmp = tmp.left
                        findPreSuc.suc = tmp
                return
        # If key is smaller than root's key, go to left subtree
        if root.key > key :
                findPreSuc.suc = root
                findPreSuc(root.left, key)

        else: # go to right subtree
                findPreSuc.pre = root
                findPreSuc(root.right, key)

# A utility function to insert a new node in with given key in BST
def insert(node , key):
        if node is None:
                return Node(key)
        if key < node.key:
                node.left = insert(node.left, key)
```

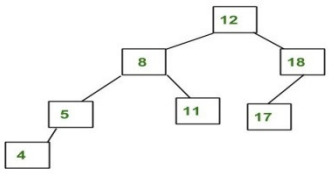
```
        else:
                node.right = insert(node.right, key)
        return node
# Driver program to test above function
key = 65 #Key to be searched in BST

""" Let us create following BST
                50
                /       \
                30      70
                / \ / \
        20 40 60 80
"""
root = None
root = insert(root, 50)
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

# Static variables of the function findPreSuc
findPreSuc.pre = None
findPreSuc.suc = None

findPreSuc(root, key)

if findPreSuc.pre is not None:
        print "Predecessor is", findPreSuc.pre.key

else:
        print "No Predecessor"

if findPreSuc.suc is not None:
        print "Successor is", findPreSuc.suc.key
else:
        print "No Successor"
```

AVL tree

AVL tree		
Type	Tree	
Time complexity in big O notation		
	Average	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)

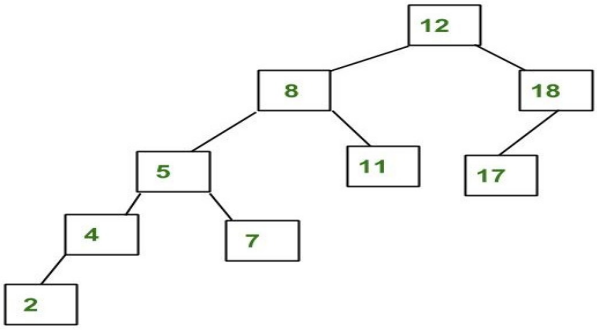
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree (See this video lecture for proof).

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1) Left Rotation
2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree

rooted with y (on the left side) or x (on

the right side)

     y                               x

    / \     Right Rotation          /  \

   x   T3   - - - - - - - >        T1   y

  / \       < - - - - - - -           / \

 T1  T2     Left Rotation            T2  T3

Keys in both of the above trees follow the

following order

 keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.
```

Steps to follow for insertion

Let the newly inserted node be w

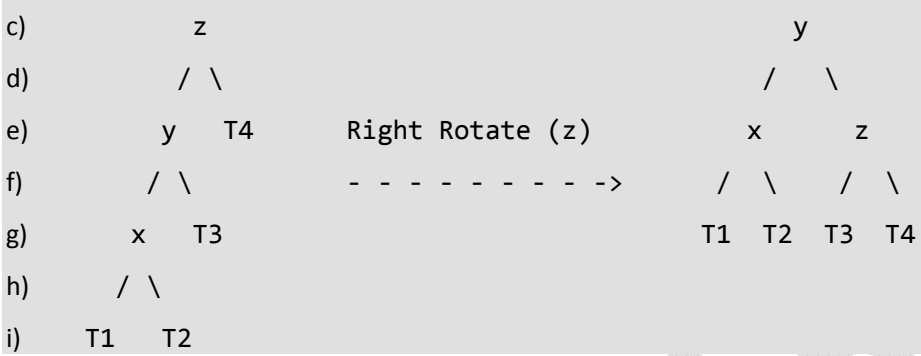
1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of

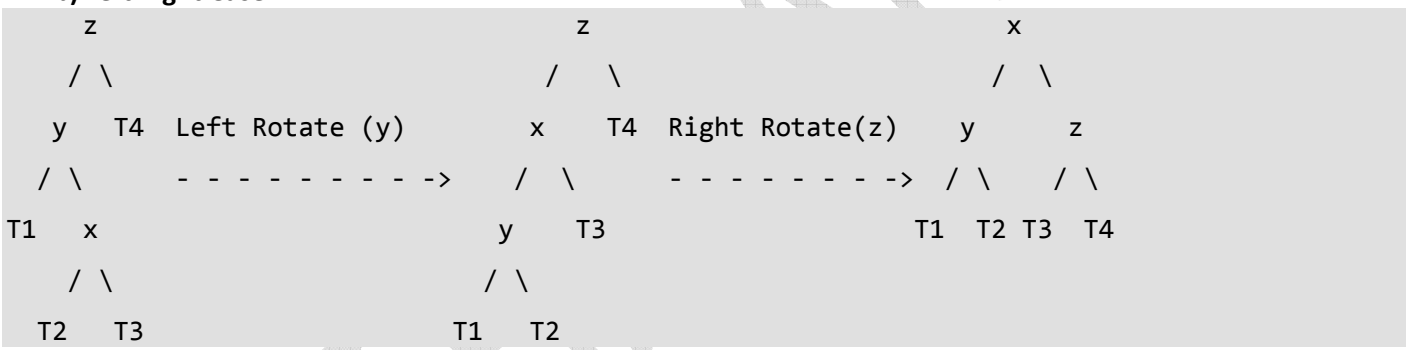
z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See this video lecture for proof)

a) Left Left Case

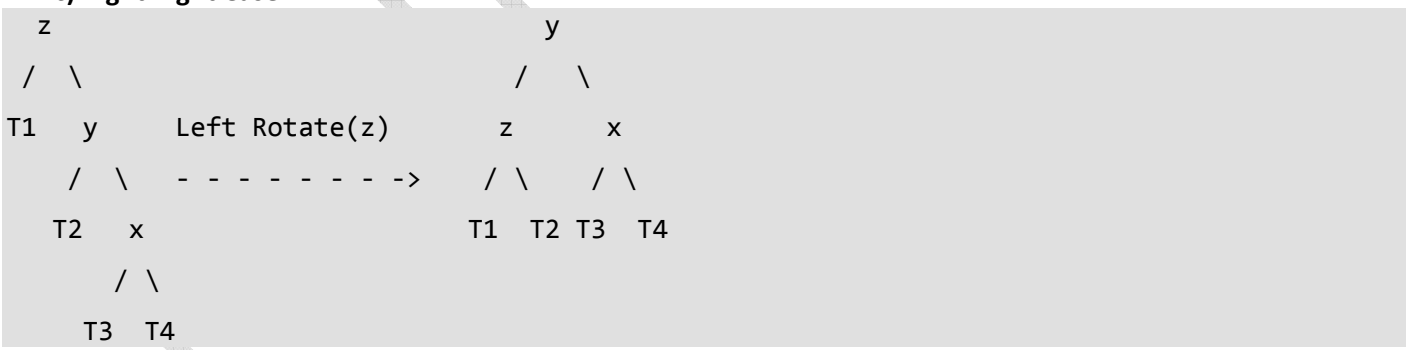
b) T1, T2, T3 and T4 are subtrees.



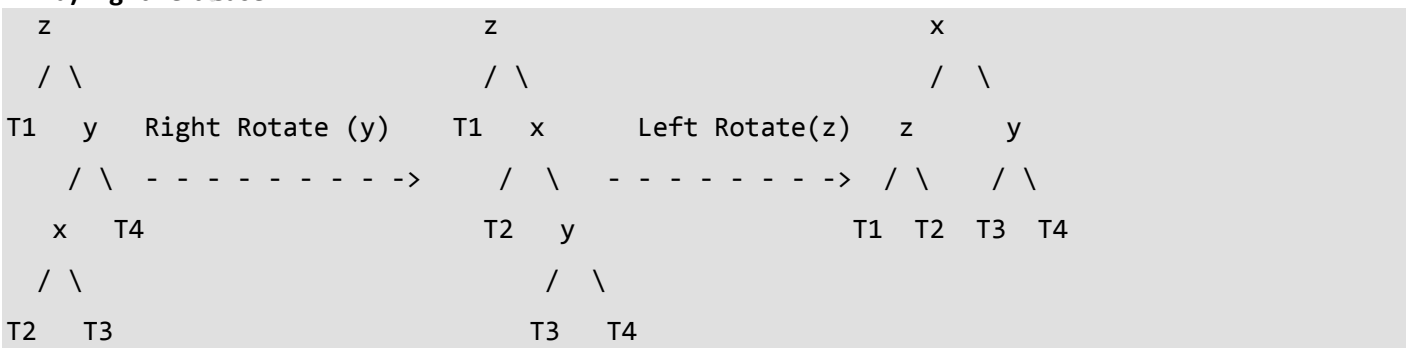
b) Left Right Case



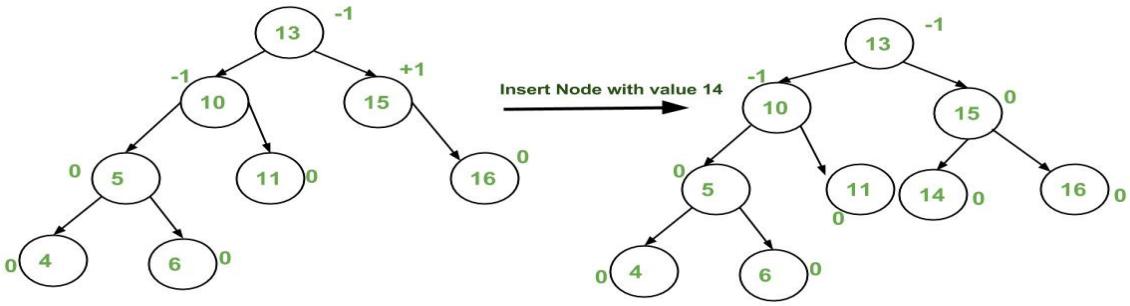
c) Right Right Case

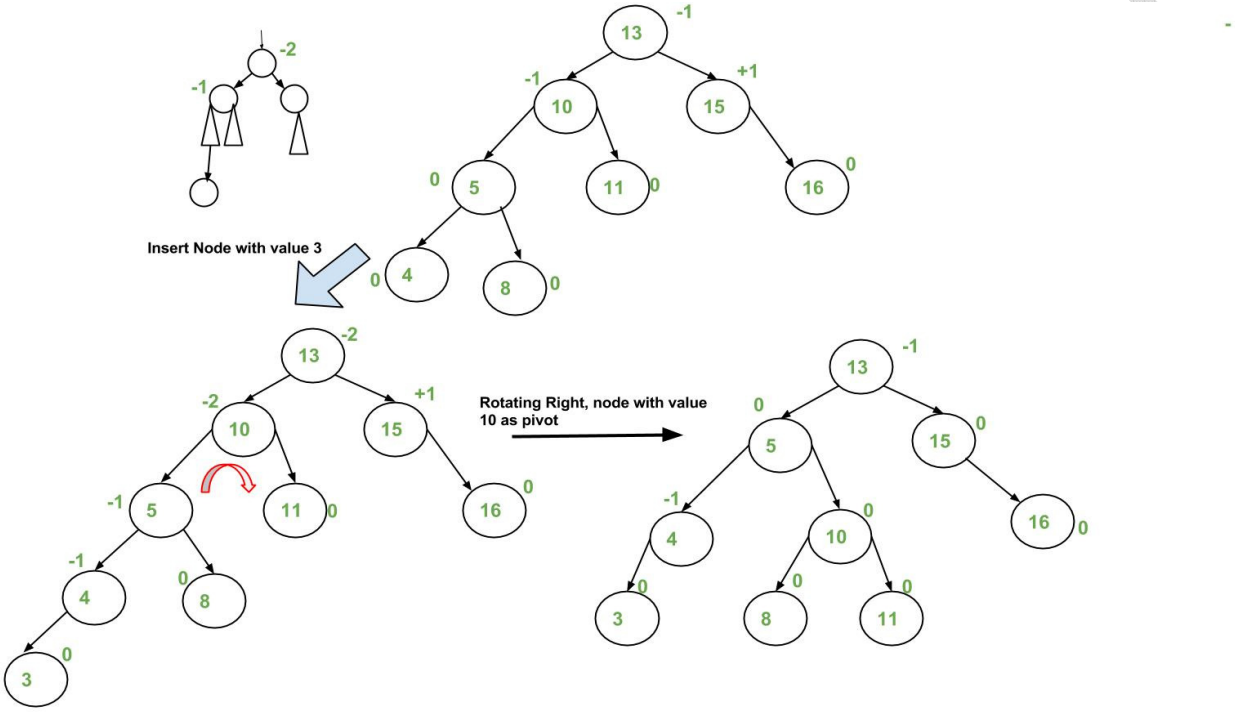


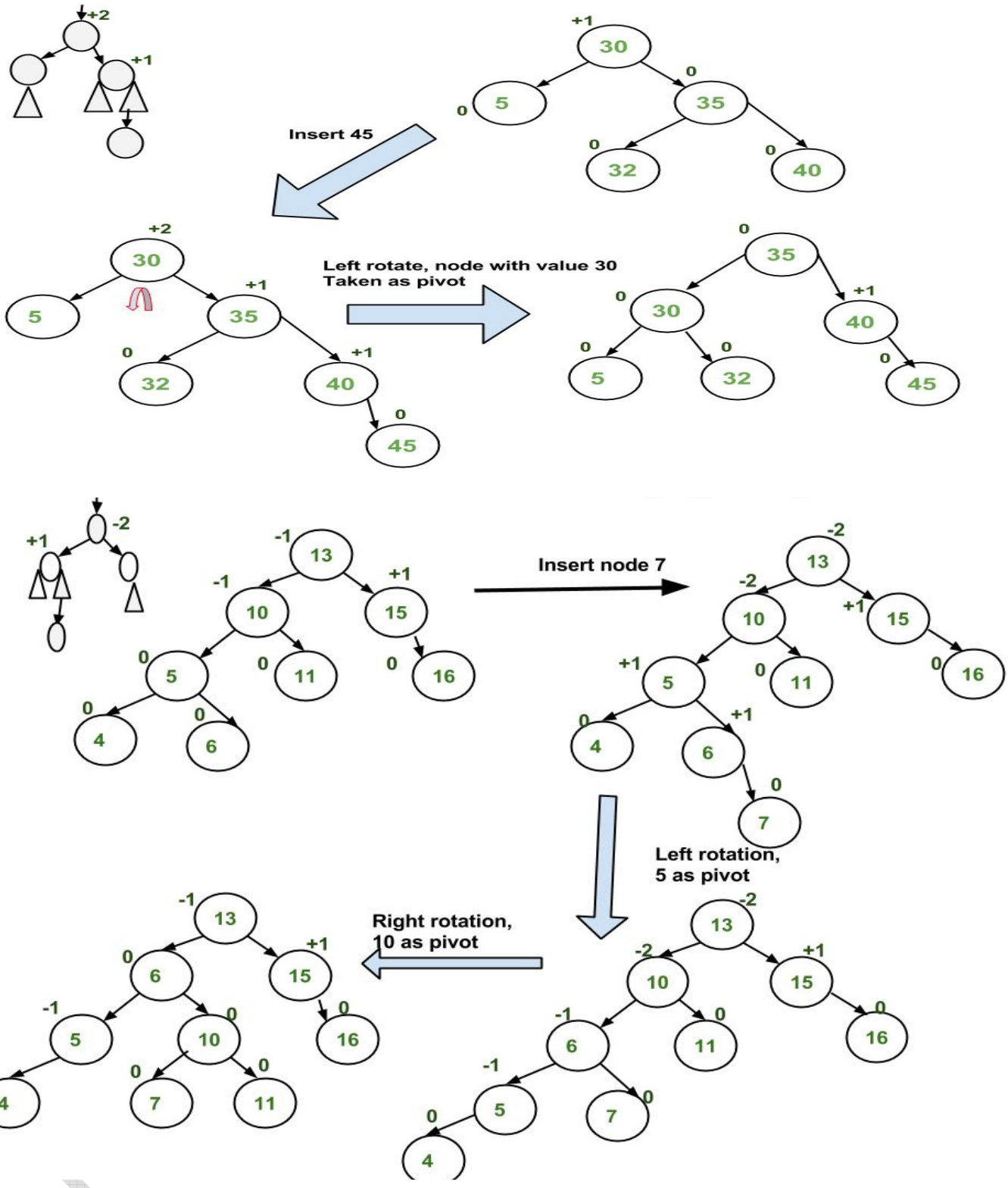
d) Right Left Case

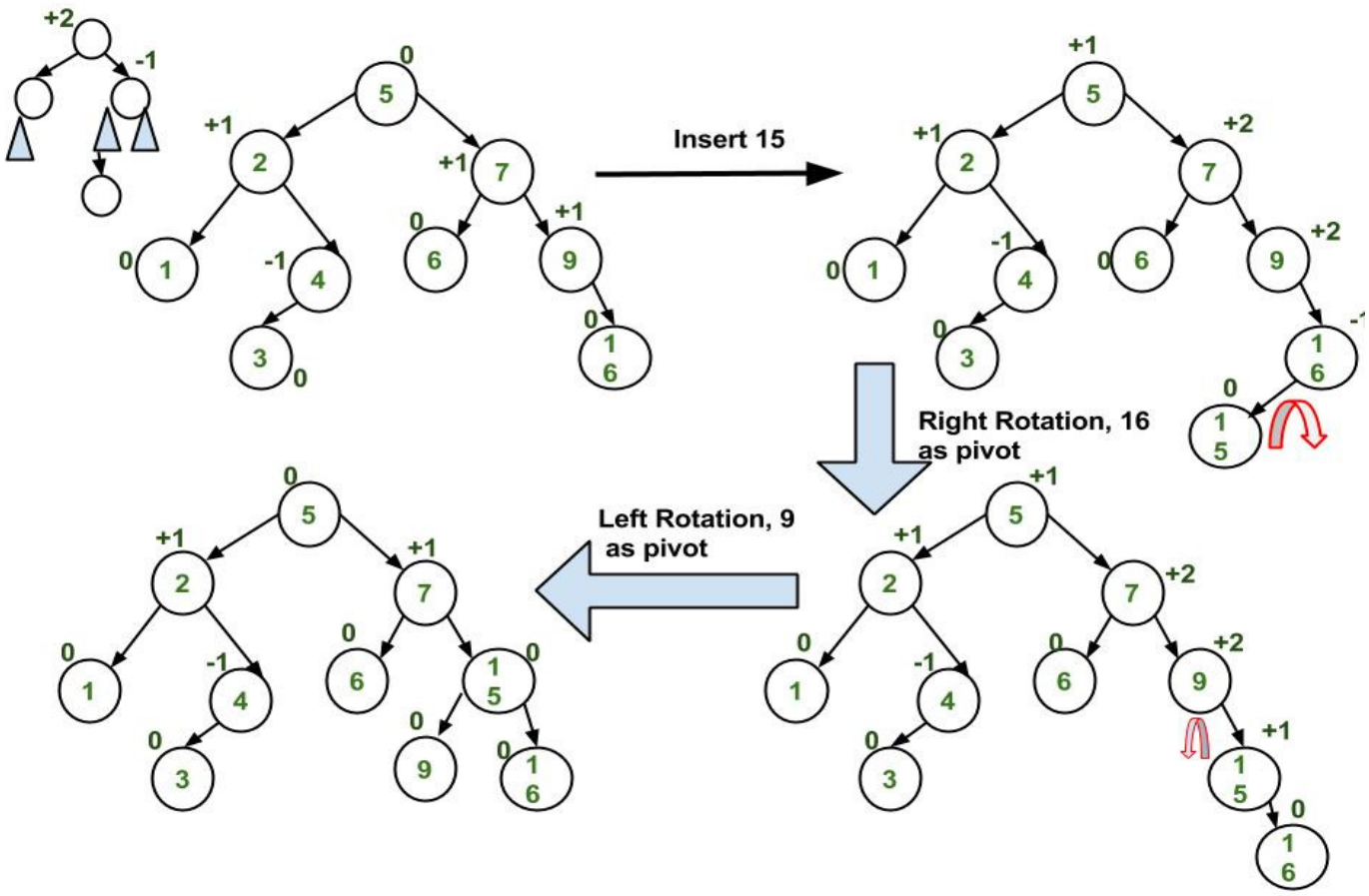


Insertion Examples:






+2
+1
Insert 45
+1
30
0
5
35
0
32
0
40
+2
30
5
+1
35
0
32
+1
40
0
45
Left rotate, node with value 30
Taken as pivot
0
35
0
30
+1
40
0
5
0
32
0
45
+1
-2
-1
13
-1
10
+1
15
0
5
0
11
0
16
0
4
0
6
Insert node 7
-2
13
-2
10
+1
15
+1
5
0
11
0
16
0
4
+1
6
0
7
Left rotation,
5 as pivot
Right rotation,
10 as pivot
-1
13
0
6
+1
15
-1
5
0
10
0
16
0
4
0
7
0
11
-2
13
-2
10
+1
15
-1
6
0
11
0
16
-1
5
0
7
0
4



implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

```
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
        int key;
        struct Node *left;
        struct Node *right;
        int height;
};
```

```
// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get the height of the tree
int height(struct Node *N)
{
	if (N == NULL)
		return 0;
	return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
	return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
	NULL left and right pointers. */
struct Node* newNode(int key)
{
	struct Node* node = (struct Node*)
						malloc(sizeof(struct Node));
	node->key = key;
	node->left = NULL;
	node->right = NULL;
	node->height = 1; // new node is initially added at leaf
	return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
	struct Node *x = y->left;
	struct Node *T2 = x->right;

	// Perform rotation
	x->right = y;
	y->left = T2;

	// Update heights
	y->height = max(height(y->left), height(y->right))+1;
	x->height = max(height(x->left), height(x->right))+1;

	// Return new root
	return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
	struct Node *y = x->right;
```

```
	struct Node *T2 = y->left;

	// Perform rotation
	y->left = x;
	x->right = T2;

	// Update heights
	x->height = max(height(x->left), height(x->right))+1;
	y->height = max(height(y->left), height(y->right))+1;

	// Return new root
	return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
	if (N == NULL)
		return 0;
	return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
	/* 1. Perform the normal BST insertion */
	if (node == NULL)
		return(newNode(key));

	if (key < node->key)
		node->left = insert(node->left, key);
	else if (key > node->key)
		node->right = insert(node->right, key);
	else // Equal keys are not allowed in BST
		return node;

	/* 2. Update height of this ancestor node */
	node->height = 1 + max(height(node->left),
					height(node->right));

	/* 3. Get the balance factor of this ancestor
		node to check whether this node became
		unbalanced */
	int balance = getBalance(node);

	// If this node becomes unbalanced, then
	// there are 4 cases

	// Left Left Case
	if (balance > 1 && key < node->left->key)
		return rightRotate(node);

	// Right Right Case
	if (balance < -1 && key > node->right->key)
```

```
                return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node->left->key)
        {
                node->left = leftRotate(node->left);
                return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node->right->key)
        {
                node->right = rightRotate(node->right);
                return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
        if(root != NULL)
        {
                printf("%d ", root->key);
                preOrder(root->left);
                preOrder(root->right);
        }
}

/* Drier program to test above function*/
int main()
{
struct Node *root = NULL;

/* Constructing tree given in the above figure */
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);

/* The constructed AVL Tree would be
                30
                / \
              20 40
             / \   \
           10 25   50
*/

printf("Preorder traversal of the constructed AVL tree is \n");
```

```
preOrder(root);

return 0;
}

# Python code to insert a node in AVL tree

# Generic tree node class
class TreeNode(object):
	def __init__(self, val):
		self.val = val
		self.left = None
		self.right = None
		self.height = 1

# AVL tree class which supports the
# Insert operation
class AVL_Tree(object):

	# Recursive function to insert key in
	# subtree rooted with node and returns
	# new root of subtree.
	def insert(self, root, key):

		# Step 1 - Perform normal BST
		if not root:
			return TreeNode(key)
		elif key < root.val:
			root.left = self.insert(root.left, key)
		else:
			root.right = self.insert(root.right, key)

		# Step 2 - Update the height of the
		# ancestor node
		root.height = 1 + max(self.getHeight(root.left),
					self.getHeight(root.right))

		# Step 3 - Get the balance factor
		balance = self.getBalance(root)

		# Step 4 - If the node is unbalanced,
		# then try out the 4 cases
		# Case 1 - Left Left
		if balance > 1 and key < root.left.val:
			return self.rightRotate(root)

		# Case 2 - Right Right
		if balance < -1 and key > root.right.val:
			return self.leftRotate(root)

		# Case 3 - Left Right
		if balance > 1 and key > root.left.val:
			root.left = self.leftRotate(root.left)
			return self.rightRotate(root)
```

```
        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)

        return root

def leftRotate(self, z):

        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                                        self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                                        self.getHeight(y.right))

        # Return the new root
        return y

def rightRotate(self, z):

        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                                        self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                                        self.getHeight(y.right))

        # Return the new root
        return y

def getHeight(self, root):
        if not root:
                return 0

        return root.height

def getBalance(self, root):
        if not root:
                return 0

        return self.getHeight(root.left) - self.getHeight(root.right)
```

```
    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.val), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)


# Driver program to test above function
myTree = AVL_Tree()
root = None

root = myTree.insert(root, 10)
root = myTree.insert(root, 20)
root = myTree.insert(root, 30)
root = myTree.insert(root, 40)
root = myTree.insert(root, 50)
root = myTree.insert(root, 25)

"""The constructed AVL Tree would be
            30
            / \
          20 40
         / \   \
        10  25  50"""

# Preorder Traversal
print("Preorder traversal of the constructed AVL tree is")
myTree.preOrder(root)
print()
```

Output:

```
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50
```

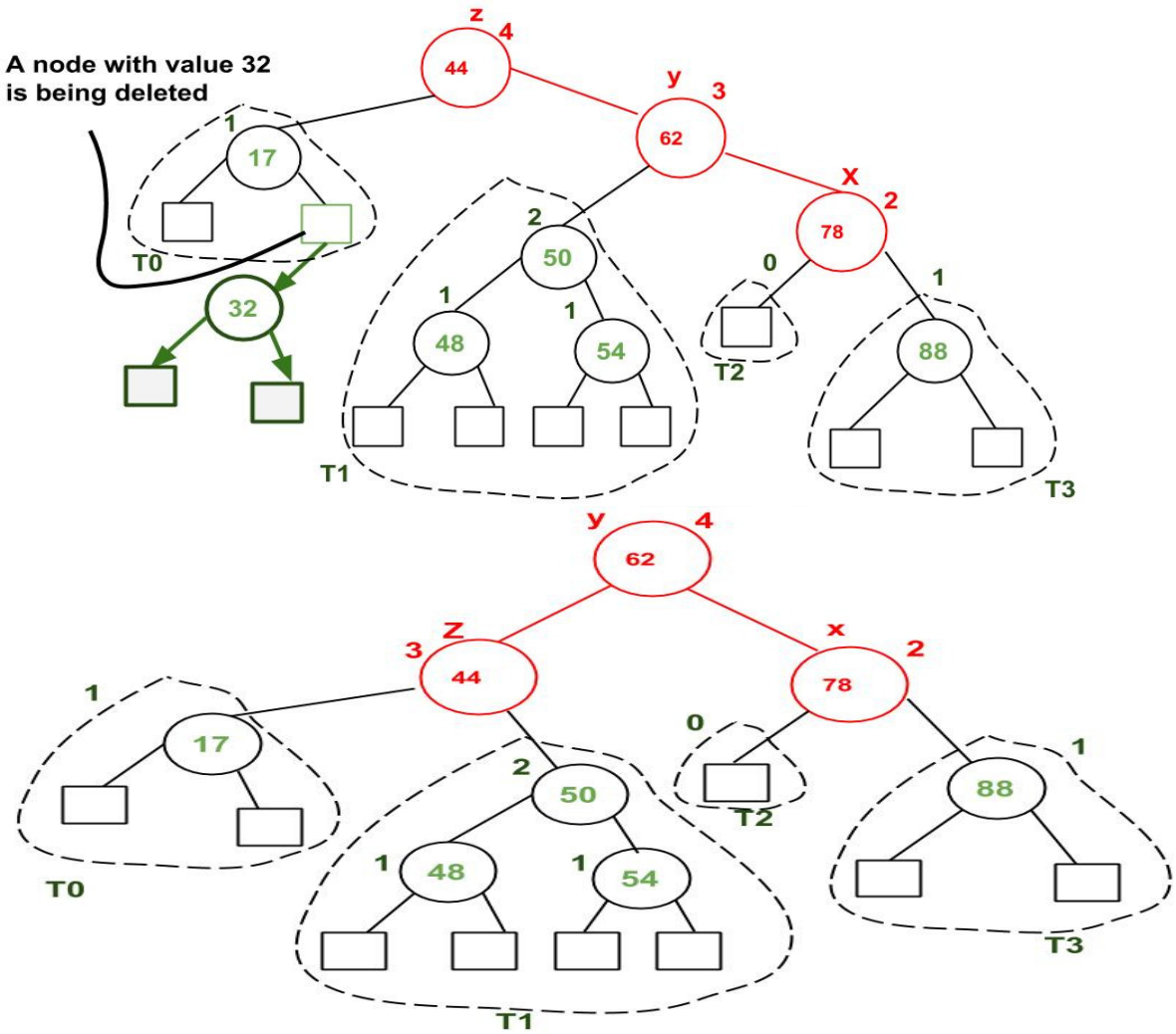
Time Complexity: The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
1) Left Rotation
2) Right Rotation

Let w be the node to be deleted
1) Perform standard BST delete for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different

from insertion here.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

Example of deletion from an AVL Tree:



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanaced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

C implementation
Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

1) Perform the normal BST deletion.
2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

```
// C program to delete a node from AVL Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
	int key;
	struct Node *left;
	struct Node *right;
	int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct Node *N)
{
	if (N == NULL)
		return 0;
	return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
	return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
	NULL left and right pointers. */
struct Node* newNode(int key)
{
	struct Node* node = (struct Node*) malloc(sizeof(struct Node));
	node->key = key;
	node->left = NULL;
	node->right = NULL;
	node->height = 1; // new node is initially added at leaf
	return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
```

```
{
        struct Node *x = y->left;
        struct Node *T2 = x->right;

        // Perform rotation
        x->right = y;
        y->left = T2;

        // Update heights
        y->height = max(height(y->left), height(y->right))+1;
        x->height = max(height(x->left), height(x->right))+1;

        // Return new root
        return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
        struct Node *y = x->right;
        struct Node *T2 = y->left;

        // Perform rotation
        y->left = x;
        x->right = T2;

        // Update heights
        x->height = max(height(x->left), height(x->right))+1;
        y->height = max(height(y->left), height(y->right))+1;

        // Return new root
        return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
        if (N == NULL)
                return 0;
        return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key)
{
        /* 1. Perform the normal BST rotation */
        if (node == NULL)
                return(newNode(key));

        if (key < node->key)
                node->left = insert(node->left, key);
        else if (key > node->key)
                node->right = insert(node->right, key);
        else // Equal keys not allowed
                return node;
```

```
        /* 2. Update height of this ancestor node */
        node->height = 1 + max(height(node->left), height(node->right));

        /* 3. Get the balance factor of this ancestor
              node to check whether this node became
              unbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then there are 4 cases

        // Left Left Case
        if (balance > 1 && key < node->left->key)
              return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node->right->key)
              return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node->left->key)
        {
              node->left = leftRotate(node->left);
              return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node->right->key)
        {
              node->right = rightRotate(node->right);
              return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
}

/* Given a non-empty binary search tree, return the
node with minimum key value found in that tree.
Note that the entire tree does not need to be
searched. */
struct Node * minValueNode(struct Node* node)
{
        struct Node* current = node;

        /* loop down to find the leftmost leaf */
        while (current->left != NULL)
              current = current->left;

        return current;
}

// Recursive function to delete a node with given key
// from subtree with given root. It returns root of
// the modified subtree.
```

```
struct Node* deleteNode(struct Node* root, int key)
{
	// STEP 1: PERFORM STANDARD BST DELETE

	if (root == NULL)
		return root;

	// If the key to be deleted is smaller than the
	// root's key, then it lies in left subtree
	if ( key < root->key )
		root->left = deleteNode(root->left, key);

	// If the key to be deleted is greater than the
	// root's key, then it lies in right subtree
	else if( key > root->key )
		root->right = deleteNode(root->right, key);

	// if key is same as root's key, then This is
	// the node to be deleted
	else
	{
		// node with only one child or no child
		if( (root->left == NULL) || (root->right == NULL) )
		{
			struct Node *temp = root->left ? root->left : root->right;

			// No child case
			if (temp == NULL)
			{
				temp = root;
				root = NULL;
			}
			else // One child case
			*root = *temp; // Copy the contents of
							// the non-empty child
			free(temp);
		}
		else
		{
			// node with two children: Get the inorder
			// successor (smallest in the right subtree)
			struct Node* temp = minValueNode(root->right);

			// Copy the inorder successor's data to this node
			root->key = temp->key;

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

	// If the tree had only one node then return
	if (root == NULL)
	return root;
```

```
        // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
        root->height = 1 + max(height(root->left), height(root->right));

        // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
        // check whether this node became unbalanced)
        int balance = getBalance(root);

        // If this node becomes unbalanced, then there are 4 cases

        // Left Left Case
        if (balance > 1 && getBalance(root->left) >= 0)
                return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root->left) < 0)
        {
                root->left = leftRotate(root->left);
                return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root->right) <= 0)
                return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root->right) > 0)
        {
                root->right = rightRotate(root->right);
                return leftRotate(root);
        }

        return root;
}

// A utility function to print preorder traversal of
// the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
        if(root != NULL)
        {
                printf("%d ", root->key);
                preOrder(root->left);
                preOrder(root->right);
        }
}

/* Driver program to test above function*/
int main()
{
struct Node *root = NULL;

/* Constructing tree given in the above figure */
        root = insert(root, 9);
        root = insert(root, 5);
```

```
	root = insert(root, 10);
	root = insert(root, 0);
	root = insert(root, 6);
	root = insert(root, 11);
	root = insert(root, -1);
	root = insert(root, 1);
	root = insert(root, 2);

	/* The constructed AVL Tree would be
		  9
		 / \
		1 10
	       / \   \
	      0   5  11
	     /    / \
	    -1    2  6
	*/

	printf("Preorder traversal of the constructed AVL "
		"tree is \n");
	preOrder(root);

	root = deleteNode(root, 10);

	/* The AVL Tree after deletion of 10
		  1
		 / \
		0  9
	       /   / \
	      -1   5  11
		  / \
		 2  6
	*/
	printf("\nPreorder traversal after deletion of 10 \n");
	preOrder(root);

	return 0;
}

# Python code to delete a node in AVL tree
# Generic tree node class
class TreeNode(object):
	def __init__(self, val):
		self.val = val
		self.left = None
		self.right = None
		self.height = 1

# AVL tree class which supports insertion,
# deletion operations
class AVL_Tree(object):

	def insert(self, root, key):

		# Step 1 - Perform normal BST
```

```
        if not root:
                return TreeNode(key)
        elif key < root.val:
                root.left = self.insert(root.left, key)
        else:
                root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
                return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
                return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)

        return root

# Recursive function to delete a node with
# given key from subtree with given root.
# It returns root of the modified subtree.
def delete(self, root, key):

        # Step 1 - Perform standard BST delete
        if not root:
                return root

        elif key < root.val:
                root.left = self.delete(root.left, key)

        elif key > root.val:
                root.right = self.delete(root.right, key)

        else:
                if root.left is None:
                        temp = root.right
                        root = None
```

```
                return temp

            elif root.right is None:
                temp = root.left
                root = None
                return temp

            temp = self.getMinValueNode(root.right)
            root.val = temp.val
            root.right = self.delete(root.right, temp.val)

        # If the tree has only one node,
        # simply return it
        if root is None:
            return root

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and self.getBalance(root.left) >= 0:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and self.getBalance(root.right) <= 0:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and self.getBalance(root.left) < 0:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and self.getBalance(root.right) > 0:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    def leftRotate(self, z):

        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
```

```
                z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
                y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))

                # Return the new root
                return y

        def rightRotate(self, z):
                y = z.left
                T3 = y.right

                # Perform rotation
                y.right = z
                z.left = T3

                # Update heights
                z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
                y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))

                # Return the new root
                return y

        def getHeight(self, root):
                if not root:
                        return 0
                return root.height
        def getBalance(self, root):
                if not root:
                        return 0
                return self.getHeight(root.left) - self.getHeight(root.right)

        def getMinValueNode(self, root):
                if root is None or root.left is None:
                        return root
                return self.getMinValueNode(root.left)

        def preOrder(self, root):

                if not root:
                        return

                print("{0} ".format(root.val), end="")
                self.preOrder(root.left)
                self.preOrder(root.right)
myTree = AVL_Tree()
root = None
nums = [9, 5, 10, 0, 6, 11, -1, 1, 2]

for num in nums:
        root = myTree.insert(root, num)

# Preorder Traversal
print("Preorder Traversal after insertion -")
myTree.preOrder(root)
print()
```

```
# Delete
key = 10
root = myTree.delete(root, key)

# Preorder Traversal
print("Preorder Traversal after deletion -")
myTree.preOrder(root)
print()
```

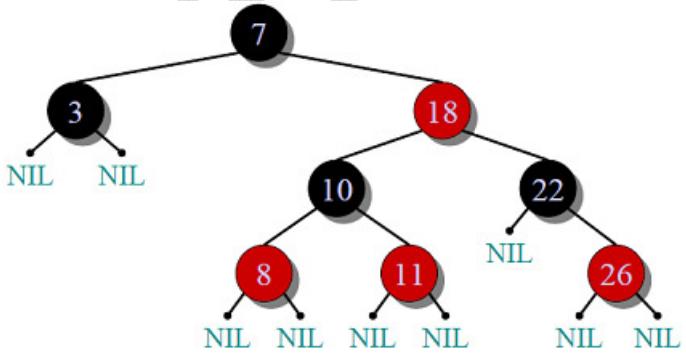
```
Preorder traversal of the constructed AVL tree is

9 1 0 -1 5 2 6 10 11

Preorder traversal after deletion of 10

1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n).

Red–black tree

Red-black tree		
Type	Tree	
Time complexity in big O notation		
	Average	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 is a self-balancing Binary Search Tree (BST) where every node follows following rules.
1) Every node has a color either red or black.
2) Root of tree is always black.
3) All leaves (NIL) are black. (All leaves are same color as the root.)
4) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
5) Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.



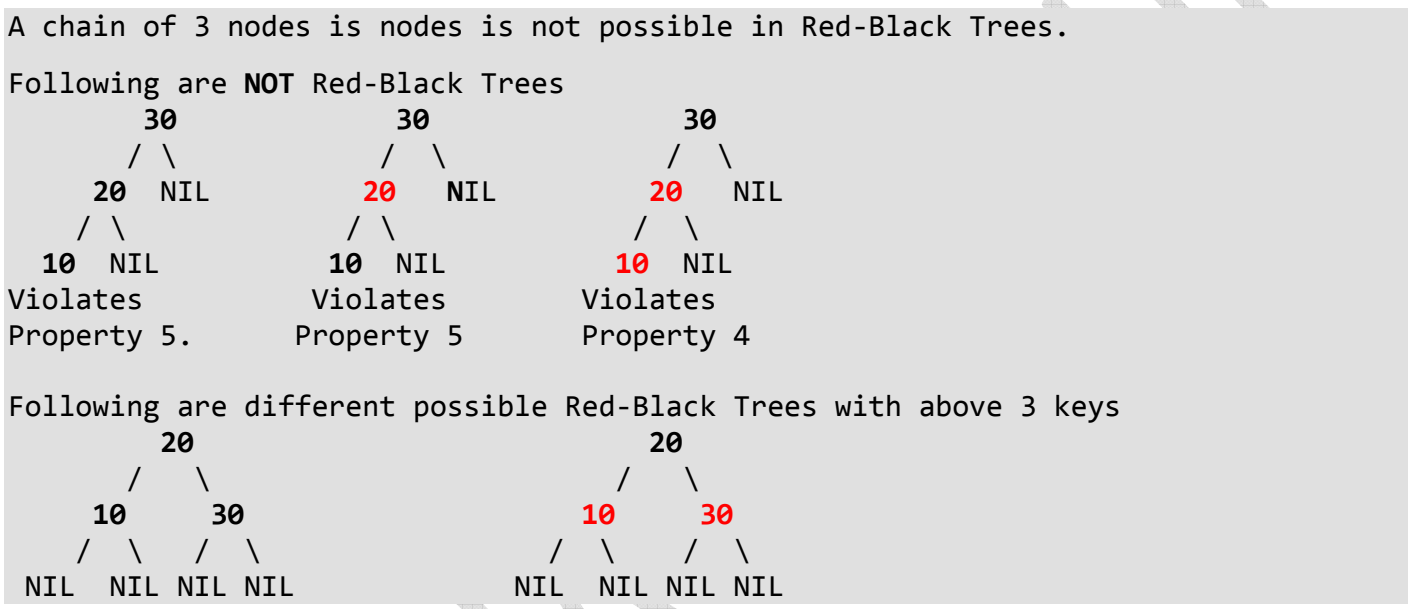
Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree

remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

Comparison with AVL Tree
The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

How does a Red-Black Tree ensure balance?
A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

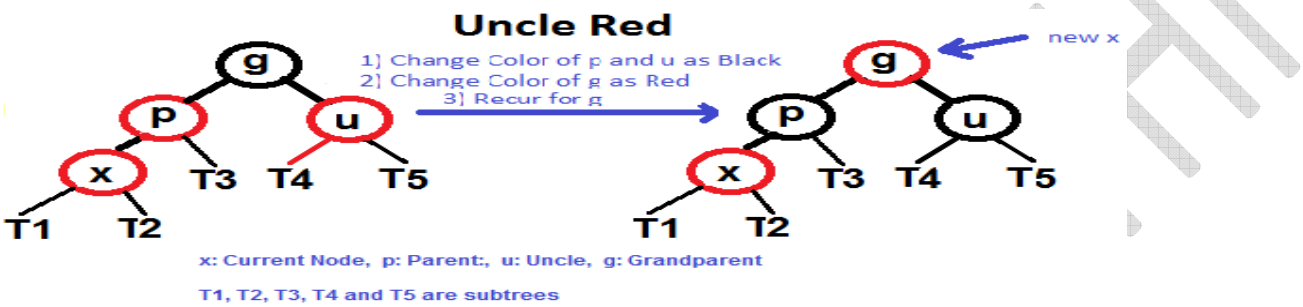


From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.
Black Height of a Red-Black Tree :
Black height is number of black nodes on a path from root to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, ***a Red-Black Tree of height h has black-height >= h/2****.*

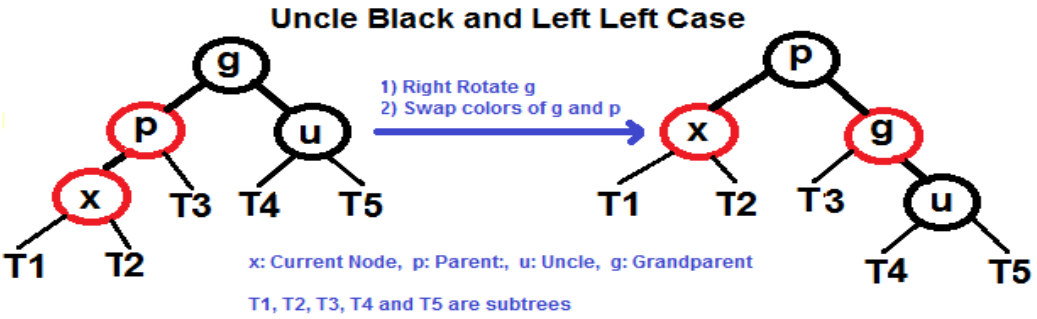
Every Red Black Tree with n nodes has height <= $2Log_2(n+1)$
This can be proved using following facts:
1) For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then $n >= 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k <= Log_2(n+1)$
2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $Log_2(n+1)$ black nodes.
3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is the total number of nodes.
From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height $<= 2Log_2(n+1)$
In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.

In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.
1) Recoloring
2) Rotation

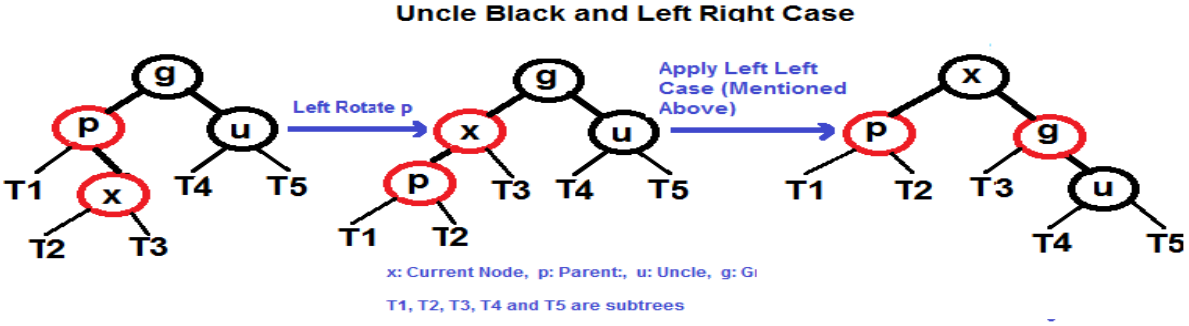
We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.
Color of a NULL node is considered as BLACK.
Let x be the newly inserted node.
1) Perform standard BST insertion and make the color of newly inserted nodes as RED.
2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
3) Do following if color of x's parent is not BLACK **and** x is not root.
….**a) If x's uncle is RED** (Grand parent must have been black from property 4)
……..**(i)** Change color of parent and uncle as BLACK.
……..**(ii)** color of grand parent as RED.
……..**(iii)** Change x = x's grandparent, repeat steps 2 and 3 for new x.



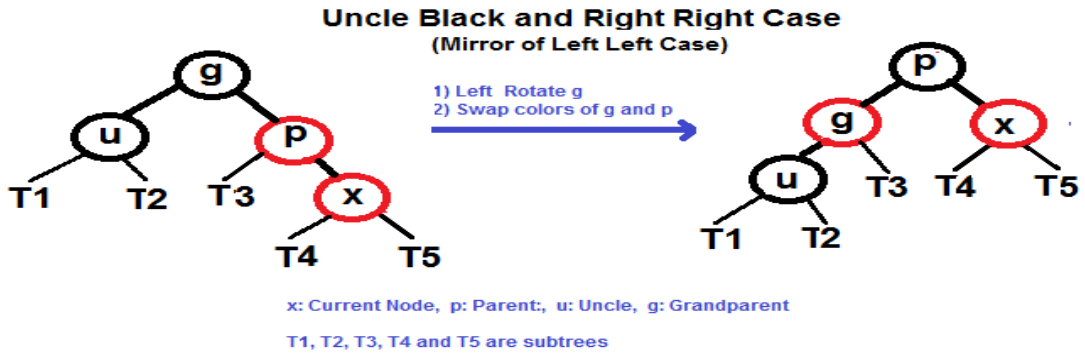
….**b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)
……..**i)** Left Left Case (p is left child of g and x is left child of p)
……..**ii)** Left Right Case (p is left child of g and x is right child of p)
……..**iii)** Right Right Case (Mirror of case i)
……..**iv)** Right Left Case (Mirror of case ii)
Following are operations to be performed in four subcases when uncle is BLACK.
All four cases when Uncle is BLACK
Left Left Case (See g, p and x)



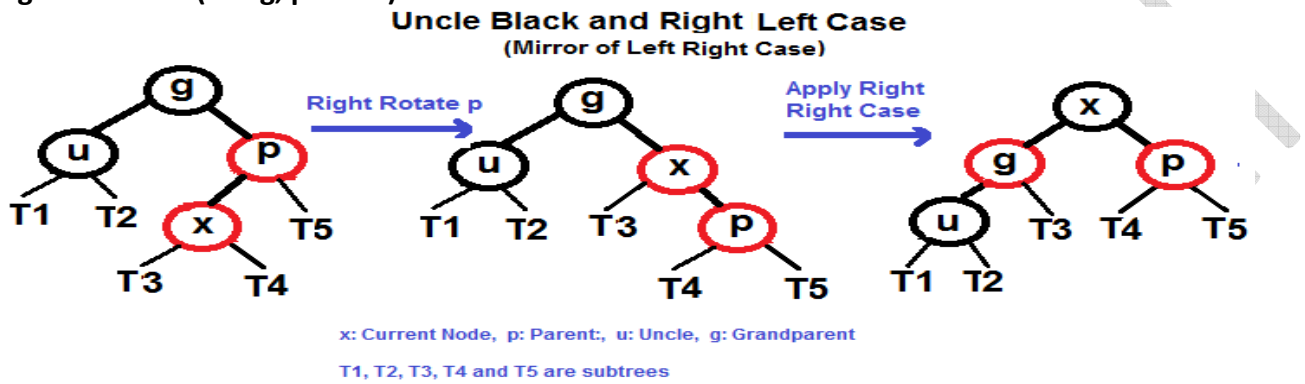
Left Right Case (See g, p and x)



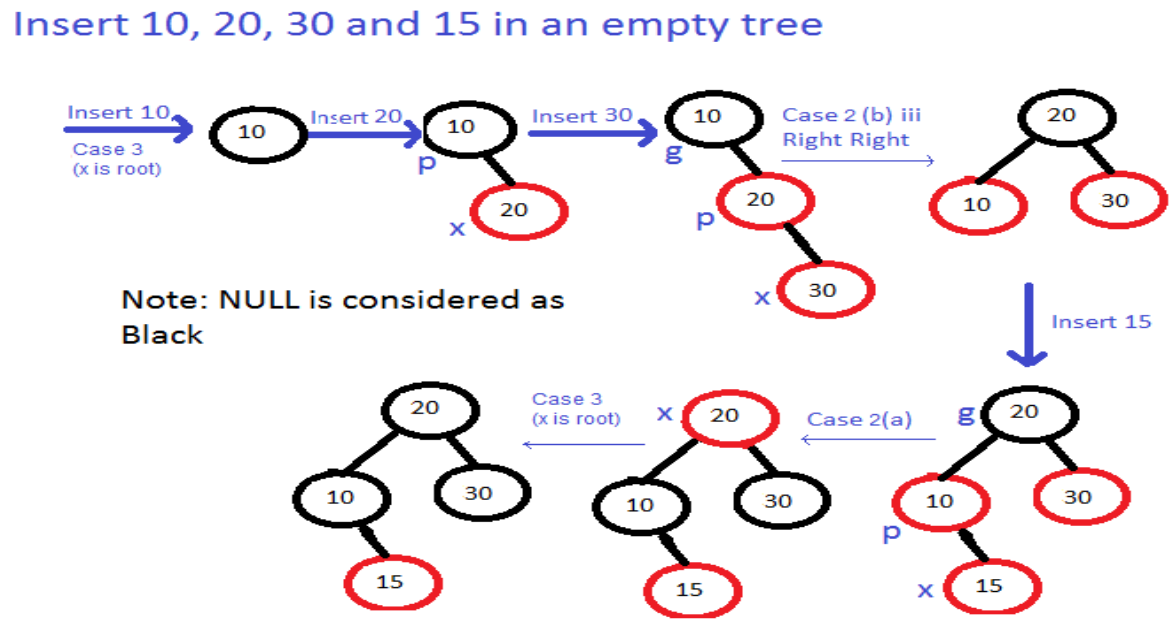
Right Right Case (See g, p and x)



Right Left Case (See g, p and x)



Examples of Insertion




```
/** C++ implementation for Red-Black Tree Insertion **/
#include <bits/stdc++.h>
using namespace std;

enum Color {RED, BLACK};

struct Node
{
	int data;
	bool color;
	Node *left, *right, *parent;

	// Constructor
```

```
        Node(int data)
        {
        this->data = data;
        left = right = parent = NULL;
        this->color = RED;
        }
};

// Class to represent Red-Black Tree
class RBTree
{
private:
        Node *root;
protected:
        void rotateLeft(Node *&, Node *&);
        void rotateRight(Node *&, Node *&);
        void fixViolation(Node *&, Node *&);
public:
        // Constructor
        RBTree() { root = NULL; }
        void insert(const int &n);
        void inorder();
        void levelOrder();
};

// A recursive function to do level order traversal
void inorderHelper(Node *root)
{
        if (root == NULL)
                return;

        inorderHelper(root->left);
        cout << root->data << " ";
        inorderHelper(root->right);
}

/* A utility function to insert a new node with given key
in BST */
Node* BSTInsert(Node* root, Node *pt)
{
        /* If the tree is empty, return a new node */
        if (root == NULL)
        return pt;

        /* Otherwise, recur down the tree */
        if (pt->data < root->data)
        {
                root->left = BSTInsert(root->left, pt);
                root->left->parent = root;
        }
```

```
	else if (pt->data > root->data)
	{
		root->right = BSTInsert(root->right, pt);
		root->right->parent = root;
	}

	/* return the (unchanged) node pointer */
	return root;
}

// Utility function to do level order traversal
void levelOrderHelper(Node *root)
{
	if (root == NULL)
		return;

	std::queue<Node *> q;
	q.push(root);

	while (!q.empty())
	{
		Node *temp = q.front();
		cout << temp->data << " ";
		q.pop();

		if (temp->left != NULL)
			q.push(temp->left);

		if (temp->right != NULL)
			q.push(temp->right);
	}
}

void RBTree::rotateLeft(Node *&root, Node *&pt)
{
	Node *pt_right = pt->right;

	pt->right = pt_right->left;

	if (pt->right != NULL)
		pt->right->parent = pt;

	pt_right->parent = pt->parent;

	if (pt->parent == NULL)
		root = pt_right;

	else if (pt == pt->parent->left)
		pt->parent->left = pt_right;
```

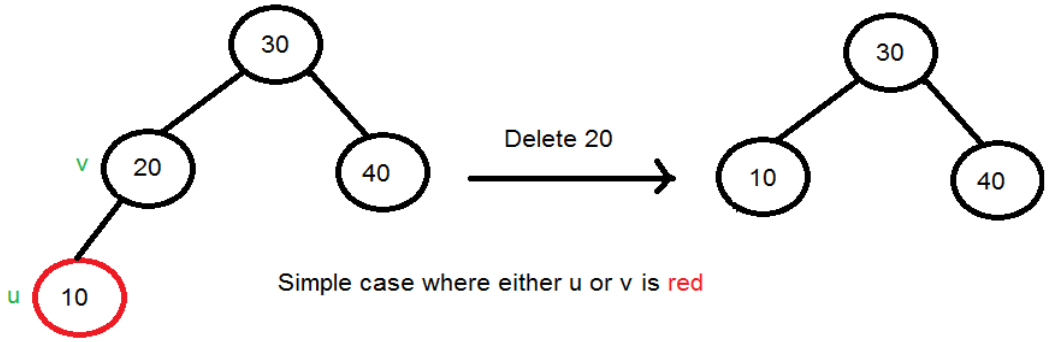
```
        else
                pt->parent->right = pt_right;

        pt_right->left = pt;
        pt->parent = pt_right;
}

void RBTree::rotateRight(Node *&root, Node *&pt)
{
        Node *pt_left = pt->left;

        pt->left = pt_left->right;

        if (pt->left != NULL)
                pt->left->parent = pt;

        pt_left->parent = pt->parent;

        if (pt->parent == NULL)
                root = pt_left;

        else if (pt == pt->parent->left)
                pt->parent->left = pt_left;

        else
                pt->parent->right = pt_left;

        pt_left->right = pt;
        pt->parent = pt_left;
}

// This function fixes violations caused by BST insertion
void RBTree::fixViolation(Node *&root, Node *&pt)
{
        Node *parent_pt = NULL;
        Node *grand_parent_pt = NULL;

        while ((pt != root) && (pt->color != BLACK) &&
                (pt->parent->color == RED))
        {

                parent_pt = pt->parent;
                grand_parent_pt = pt->parent->parent;

                /* Case : A
                        Parent of pt is left child of Grand-parent of pt */
                if (parent_pt == grand_parent_pt->left)
                {

                        Node *uncle_pt = grand_parent_pt->right;
```

```
            /* Case : 1
            The uncle of pt is also red
            Only Recoloring required */
            if (uncle_pt != NULL && uncle_pt->color == RED)
            {
                    grand_parent_pt->color = RED;
                    parent_pt->color = BLACK;
                    uncle_pt->color = BLACK;
                    pt = grand_parent_pt;
            }

            else
            {
                    /* Case : 2
                    pt is right child of its parent
                    Left-rotation required */
                    if (pt == parent_pt->right)
                    {
                            rotateLeft(root, parent_pt);
                            pt = parent_pt;
                            parent_pt = pt->parent;
                    }

                    /* Case : 3
                    pt is left child of its parent
                    Right-rotation required */
                    rotateRight(root, grand_parent_pt);
                    swap(parent_pt->color, grand_parent_pt->color);
                    pt = parent_pt;
            }
    }

    /* Case : B
    Parent of pt is right child of Grand-parent of pt */
    else
    {
            Node *uncle_pt = grand_parent_pt->left;

            /* Case : 1
                    The uncle of pt is also red
                    Only Recoloring required */
            if ((uncle_pt != NULL) && (uncle_pt->color == RED))
            {
                    grand_parent_pt->color = RED;
                    parent_pt->color = BLACK;
                    uncle_pt->color = BLACK;
                    pt = grand_parent_pt;
            }
            else
```

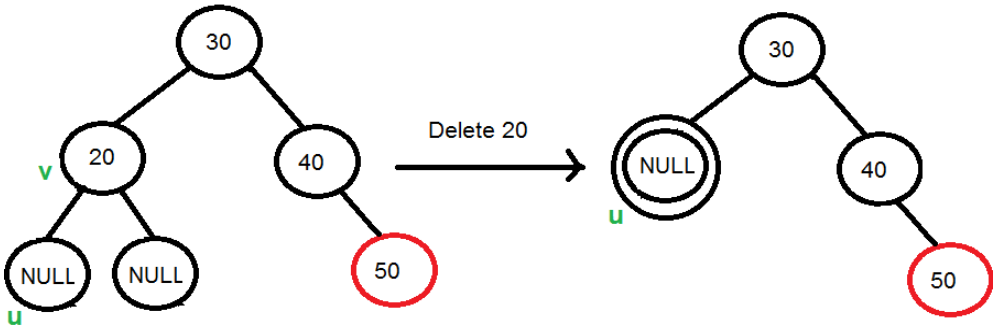
```
                {
                        /* Case : 2
                        pt is left child of its parent
                        Right-rotation required */
                        if (pt == parent_pt->left)
                        {
                                rotateRight(root, parent_pt);
                                pt = parent_pt;
                                parent_pt = pt->parent;
                        }

                        /* Case : 3
                        pt is right child of its parent
                        Left-rotation required */
                        rotateLeft(root, grand_parent_pt);
                        swap(parent_pt->color, grand_parent_pt->color);
                        pt = parent_pt;
                }
        }
    }

    root->color = BLACK;
}

// Function to insert a new node with given data
void RBTree::insert(const int &data)
{
    Node *pt = new Node(data);

    // Do a normal BST insert
    root = BSTInsert(root, pt);

    // fix Red Black Tree violations
    fixViolation(root, pt);
}

// Function to do inorder and level order traversals
void RBTree::inorder()      { inorderHelper(root);}
void RBTree::levelOrder() { levelOrderHelper(root); }

// Driver Code
int main()
{
    RBTree tree;

    tree.insert(7);
    tree.insert(6);
    tree.insert(5);
    tree.insert(4);
    tree.insert(3);
```

```
        tree.insert(2);
        tree.insert(1);

        cout << "Inoder Traversal of Created Tree\n";
        tree.inorder();

        cout << "\n\nLevel Order Traversal of Created Tree\n";
        tree.levelOrder();

        return 0;
}
```

Insertion Vs Deletion:
Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.
The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.
Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.
Deletion Steps
Following are detailed steps for deletion.
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).
2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.
3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

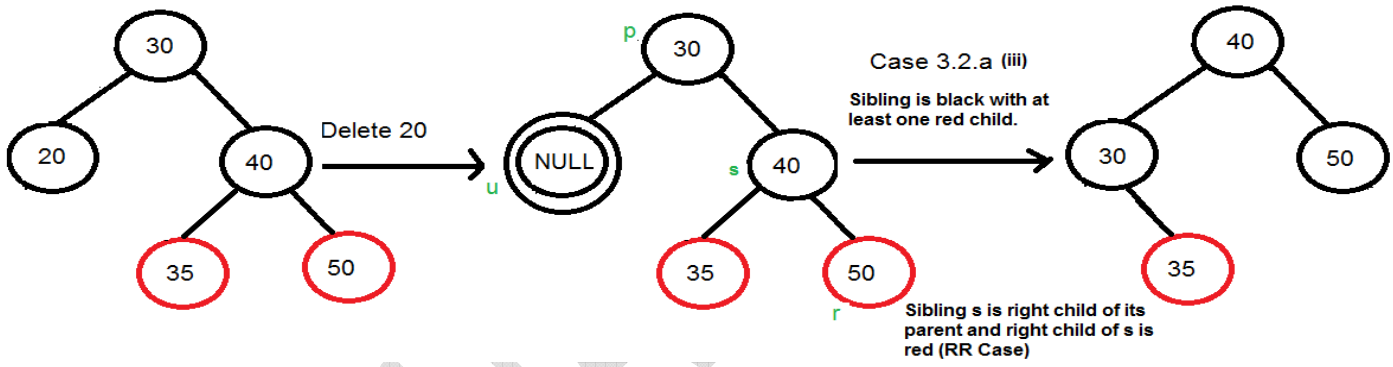
3.2) Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

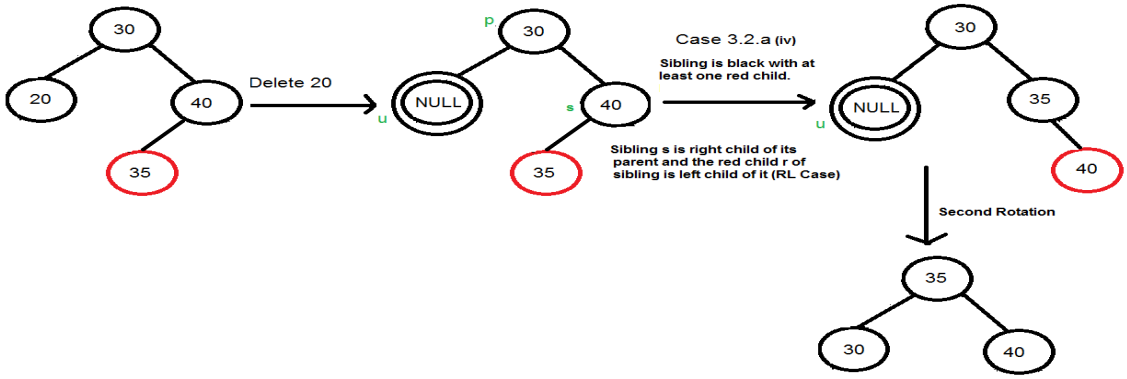
……**(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

……**(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

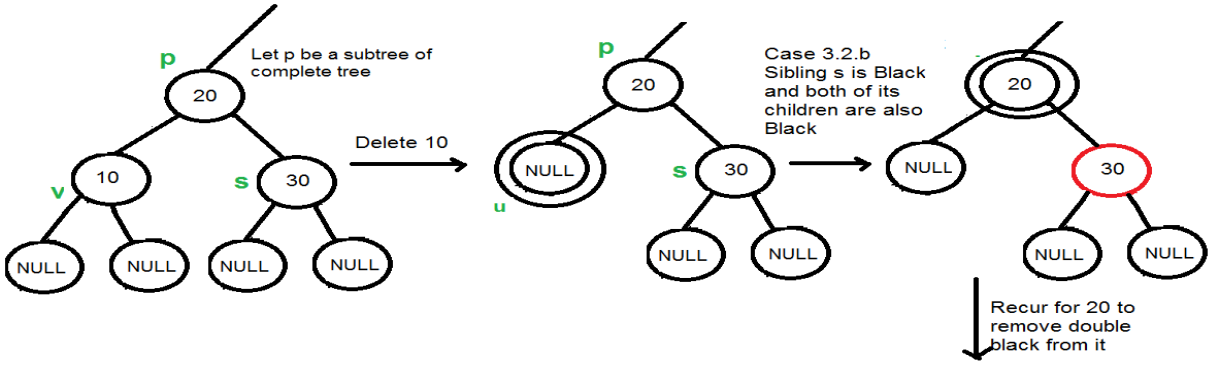
……**(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



……**(iv)** Right Left Case (s is right child of its parent and r is left child of s)



…..**(b): If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.

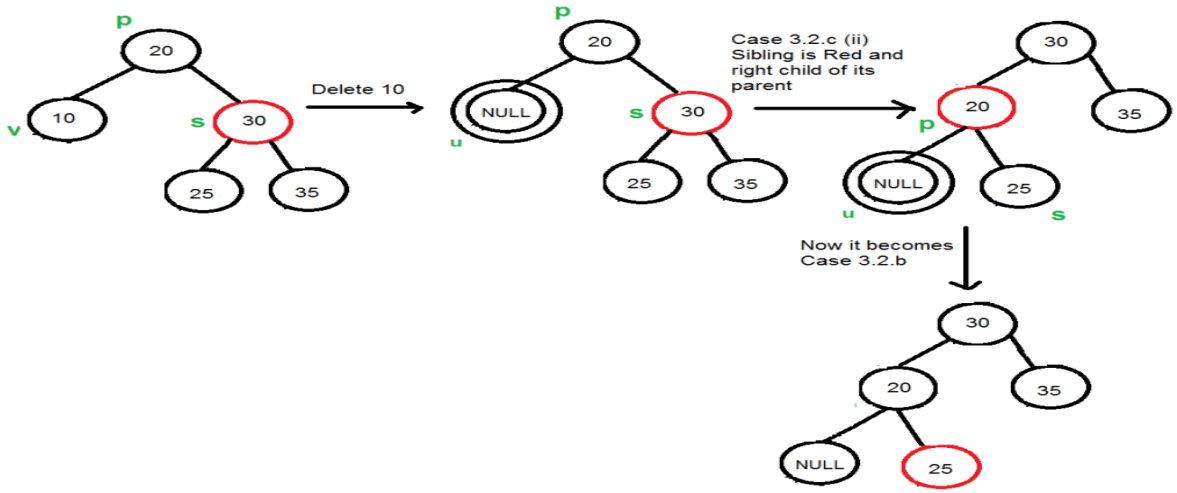


In this case, if parent was red, then we didn't need to recur for prent, we can simply make it black (red + double black = single black)

…..**(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

……**(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

……**(iii)** Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

below is the C++ implementation of above approach:

```
#include <iostream>
#include <queue>
using namespace std;

enum COLOR { RED, BLACK };

class Node {
public:
int val;
COLOR color;
Node *left, *right, *parent;

Node(int val) : val(val) {
        parent = left = right = NULL;

        // Node is created during insertion
        // Node is red at insertion
        color = RED;
}
```

```
// returns pointer to uncle
Node *uncle() {
	// If no parent or grandparent, then no uncle
	if (parent == NULL or parent->parent == NULL)
	return NULL;

	if (parent->isOnLeft())
	// uncle on right
	return parent->parent->right;
	else
	// uncle on left
	return parent->parent->left;
}

// check if node is left child of parent
bool isOnLeft() { return this == parent->left; }

// returns pointer to sibling
Node *sibling() {
	// sibling null if no parent
	if (parent == NULL)
	return NULL;

	if (isOnLeft())
	return parent->right;

	return parent->left;
}

// moves node down and moves given node in its place
void moveDown(Node *nParent) {
	if (parent != NULL) {
	if (isOnLeft()) {
		parent->left = nParent;
	} else {
		parent->right = nParent;
	}
	}
	nParent->parent = parent;
	parent = nParent;
}

bool hasRedChild() {
	return (left != NULL and left->color == RED) or
		(right != NULL and right->color == RED);
}
};

class RBTree {
Node *root;

// left rotates the given node
void leftRotate(Node *x) {
	// new parent will be node's right child
```

```
        Node *nParent = x->right;

        // update root if current node is root
        if (x == root)
        root = nParent;

        x->moveDown(nParent);

        // connect x with new parent's left element
        x->right = nParent->left;
        // connect new parent's left element with node
        // if it is not null
        if (nParent->left != NULL)
        nParent->left->parent = x;

        // connect new parent with x
        nParent->left = x;
}

void rightRotate(Node *x) {
        // new parent will be node's left child
        Node *nParent = x->left;

        // update root if current node is root
        if (x == root)
        root = nParent;

        x->moveDown(nParent);

        // connect x with new parent's right element
        x->left = nParent->right;
        // connect new parent's right element with node
        // if it is not null
        if (nParent->right != NULL)
        nParent->right->parent = x;

        // connect new parent with x
        nParent->right = x;
}

void swapColors(Node *x1, Node *x2) {
        COLOR temp;
        temp = x1->color;
        x1->color = x2->color;
        x2->color = temp;
}

void swapValues(Node *u, Node *v) {
        int temp;
        temp = u->val;
        u->val = v->val;
        v->val = temp;
}

// fix red red at given node
```

```
void fixRedRed(Node *x) {
	// if x is root color it black and return
	if (x == root) {
	x->color = BLACK;
	return;
	}

	// initialize parent, grandparent, uncle
	Node *parent = x->parent, *grandparent = parent->parent,
		*uncle = x->uncle();

	if (parent->color != BLACK) {
	if (uncle != NULL && uncle->color == RED) {
		// uncle red, perform recoloring and recurse
		parent->color = BLACK;
		uncle->color = BLACK;
		grandparent->color = RED;
		fixRedRed(grandparent);
	} else {
		// Else perform LR, LL, RL, RR
		if (parent->isOnLeft()) {
		if (x->isOnLeft()) {
			// for left right
			swapColors(parent, grandparent);
		} else {
			leftRotate(parent);
			swapColors(x, grandparent);
		}
		// for left left and left right
		rightRotate(grandparent);
		} else {
		if (x->isOnLeft()) {
			// for right left
			rightRotate(parent);
			swapColors(x, grandparent);
		} else {
			swapColors(parent, grandparent);
		}

		// for right right and right left
		leftRotate(grandparent);
		}
	}
	}
}

// find node that do not have a left child
// in the subtree of the given node
Node *successor(Node *x) {
	Node *temp = x;

	while (temp->left != NULL)
	temp = temp->left;

	return temp;
```

```
}

// find node that replaces a deleted node in BST
Node *BSTreplace(Node *x) {
        // when node have 2 children
        if (x->left != NULL and x->right != NULL)
        return successor(x->right);

        // when leaf
        if (x->left == NULL and x->right == NULL)
        return NULL;

        // when single child
        if (x->left != NULL)
        return x->left;
        else
        return x->right;
}

// deletes the given node
void deleteNode(Node *v) {
        Node *u = BSTreplace(v);

        // True when u and v are both black
        bool uvBlack = ((u == NULL or u->color == BLACK) and (v->color == BLACK));
        Node *parent = v->parent;

        if (u == NULL) {
        // u is NULL therefore v is leaf
        if (v == root) {
                // v is root, making root null
                root = NULL;
        } else {
                if (uvBlack) {
                // u and v both black
                // v is leaf, fix double black at v
                fixDoubleBlack(v);
                } else {
                // u or v is red
                if (v->sibling() != NULL)
                        // sibling is not null, make it red"
                        v->sibling()->color = RED;
                }

                // delete v from the tree
                if (v->isOnLeft()) {
                parent->left = NULL;
                } else {
                parent->right = NULL;
                }
        }
        delete v;
        return;
        }
```

```
	if (v->left == NULL or v->right == NULL) {
	// v has 1 child
	if (v == root) {
		// v is root, assign the value of u to v, and delete u
		v->val = u->val;
		v->left = v->right = NULL;
		delete u;
	} else {
		// Detach v from tree and move u up
		if (v->isOnLeft()) {
		parent->left = u;
		} else {
		parent->right = u;
		}
		delete v;
		u->parent = parent;
		if (uvBlack) {
		// u and v both black, fix double black at u
		fixDoubleBlack(u);
		} else {
		// u or v red, color u black
		u->color = BLACK;
		}
	}
	return;
	}

	// v has 2 children, swap values with successor and recurse
	swapValues(u, v);
	deleteNode(u);
}

void fixDoubleBlack(Node *x) {
	if (x == root)
	// Reached root
	return;

	Node *sibling = x->sibling(), *parent = x->parent;
	if (sibling == NULL) {
	// No sibiling, double black pushed up
	fixDoubleBlack(parent);
	} else {
	if (sibling->color == RED) {
		// Sibling red
		parent->color = RED;
		sibling->color = BLACK;
		if (sibling->isOnLeft()) {
		// left case
		rightRotate(parent);
		} else {
		// right case
		leftRotate(parent);
		}
		fixDoubleBlack(x);
	} else {
```

```
                // Sibling black
                if (sibling->hasRedChild()) {
                // at least 1 red children
                if (sibling->left != NULL and sibling->left->color == RED) {
                        if (sibling->isOnLeft()) {
                        // left left
                        sibling->left->color = sibling->color;
                        sibling->color = parent->color;
                        rightRotate(parent);
                        } else {
                        // right left
                        sibling->left->color = parent->color;
                        rightRotate(sibling);
                        leftRotate(parent);
                        }
                } else {
                        if (sibling->isOnLeft()) {
                        // left right
                        sibling->right->color = parent->color;
                        leftRotate(sibling);
                        rightRotate(parent);
                        } else {
                        // right right
                        sibling->right->color = sibling->color;
                        sibling->color = parent->color;
                        leftRotate(parent);
                        }
                }
                parent->color = BLACK;
                } else {
                // 2 black children
                sibling->color = RED;
                if (parent->color == BLACK)
                        fixDoubleBlack(parent);
                else
                        parent->color = BLACK;
                }
        }
        }
}

// prints level order for given node
void levelOrder(Node *x) {
        if (x == NULL)
        // return if node is null
        return;

        // queue for level order
        queue<Node *> q;
        Node *curr;

        // push x
        q.push(x);

        while (!q.empty()) {
```

```
		// while q is not empty
		// dequeue
		curr = q.front();
		q.pop();

		// print node value
		cout << curr->val << " ";

		// push children to queue
		if (curr->left != NULL)
			q.push(curr->left);
		if (curr->right != NULL)
			q.push(curr->right);
		}
}

// prints inorder recursively
void inorder(Node *x) {
		if (x == NULL)
		return;
		inorder(x->left);
		cout << x->val << " ";
		inorder(x->right);
}

public:
// constructor
// initialize root
RBTree() { root = NULL; }

Node *getRoot() { return root; }

// searches for given value
// if found returns the node (used for delete)
// else returns the last node while traversing (used in insert)
Node *search(int n) {
		Node *temp = root;
		while (temp != NULL) {
		if (n < temp->val) {
			if (temp->left == NULL)
			break;
			else
			temp = temp->left;
		} else if (n == temp->val) {
			break;
		} else {
			if (temp->right == NULL)
			break;
			else
			temp = temp->right;
		}
		}

		return temp;
}
```

```
// inserts the given value to tree
void insert(int n) {
	Node *newNode = new Node(n);
	if (root == NULL) {
	// when root is null
	// simply insert value at root
	newNode->color = BLACK;
	root = newNode;
	} else {
	Node *temp = search(n);

	if (temp->val == n) {
		// return if value already exists
		return;
	}

	// if value is not found, search returns the node
	// where the value is to be inserted

	// connect new node to correct node
	newNode->parent = temp;

	if (n < temp->val)
		temp->left = newNode;
	else
		temp->right = newNode;

	// fix red red voilaton if exists
	fixRedRed(newNode);
	}
}

// utility function that deletes the node with given value
void deleteByVal(int n) {
	if (root == NULL)
	// Tree is empty
	return;

	Node *v = search(n), *u;

	if (v->val != n) {
	cout << "No node found to delete with value:" << n << endl;
	return;
	}

	deleteNode(v);
}

// prints inorder of the tree
void printInOrder() {
	cout << "Inorder: " << endl;
	if (root == NULL)
	cout << "Tree is empty" << endl;
	else
	inorder(root);
```

```
        cout << endl;
}

// prints level order of the tree
void printLevelOrder() {
        cout << "Level order: " << endl;
        if (root == NULL)
        cout << "Tree is empty" << endl;
        else
        levelOrder(root);
        cout << endl;
}
};

int main() {
RBTree tree;

tree.insert(7);
tree.insert(3);
tree.insert(18);
tree.insert(10);
tree.insert(22);
tree.insert(8);
tree.insert(11);
tree.insert(26);
tree.insert(2);
tree.insert(6);
tree.insert(13);

tree.printInOrder();
tree.printLevelOrder();

cout<<endl<<"Deleting 18, 11, 3, 10, 22"<<endl;

tree.deleteByVal(18);
tree.deleteByVal(11);
tree.deleteByVal(3);
tree.deleteByVal(10);
tree.deleteByVal(22);

tree.printInOrder();
tree.printLevelOrder();
return 0;
}
```

```
Inorder:
2 3 6 7 8 10 11 13 18 22 26
Level order:
10 7 18 3 8 11 22 2 6 13 26


Deleting 18, 11, 3, 10, 22
```

```
Inorder:
2 6 7 8 13 26
Level order:
13 7 26 6 8 2
```

Red Black Tree vs AVL Tree

1. AVL trees provide **faster lookups** than Red Black Trees because they are more strictly balanced.
2. Red Black Trees provide **faster insertion and removal** operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
3. AVL trees store **balance factors or heights** with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
4. Red Black Trees are used in most of the language libraries like **map**, **multimap**, **multiset** in C++ whereas AVL trees are used in **databases** where faster retrievals are required.

2-3 Trees

2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element **leftVal**, it has two subtrees (children) namely **left** and **middle**. Whereas if a node contains two data elements **leftVal** and **rightVal**, it has three subtrees namely **left**, **middle** and **right**.

The main advantage with 2-3 trees is that it is balanced in nature as opposed to a binary search tree whose height in the worst case can be O(n). Due to this, the worst case time-complexity of operations such as search, insertion and deletion is O(log(n)) as the height of a 2-3 tree is O(log(n)) .

Search: To search a key **K** in given 2-3 tree **T**, we follow the following procedure:

Base cases:

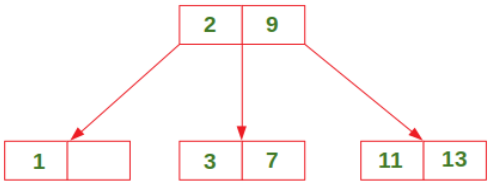
1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

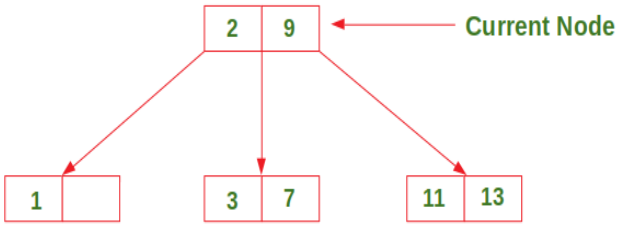
Recursive Calls:

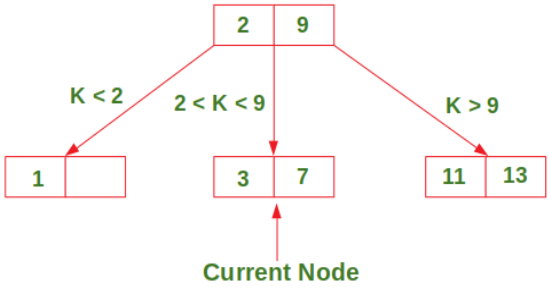
1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.

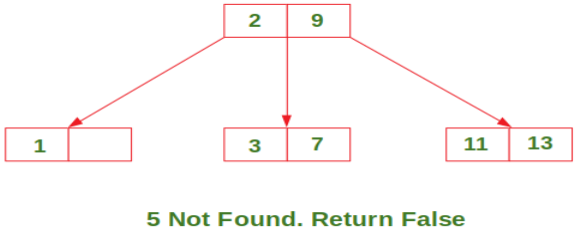
Consider the following example:

Search 5 in the following 2-3 Tree:





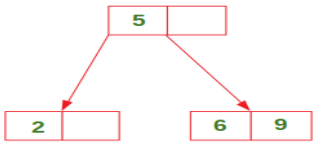




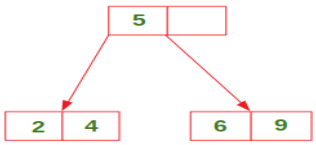
5 Not Found. Return False

Insertion: There are 3 possible cases in insertion which have been discussed below:
Case 1: Insert in a node with only one data element

Insert 4 in the following 2-3 Tree:



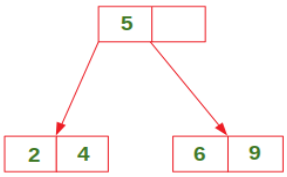
Initial



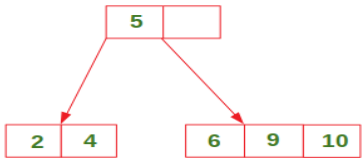
After Insertion

Case 2: Insert in a node with two data elements whose parent contains only one data element.

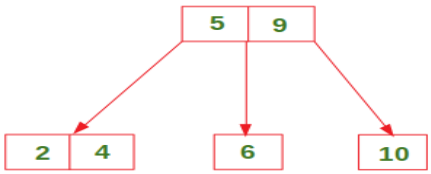
Insert 10 in the following 2-3 Tree:



Initial



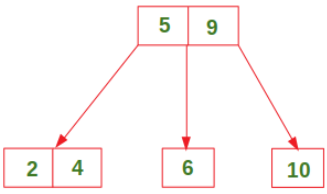
Temporary Node with 3 data elements



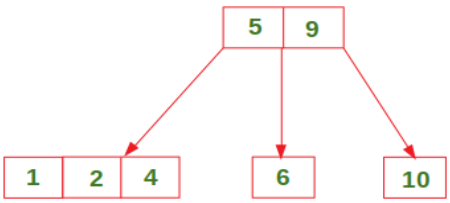
Move the middle element to
parent and split the current Node

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

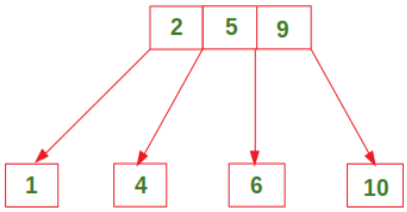
Insert 1 in the following 2-3 Tree:



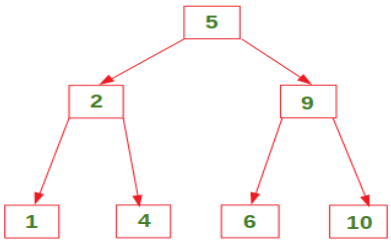
Initial



Temporary Node with 3 data elements



Move the middle element to the parent and split the current Node

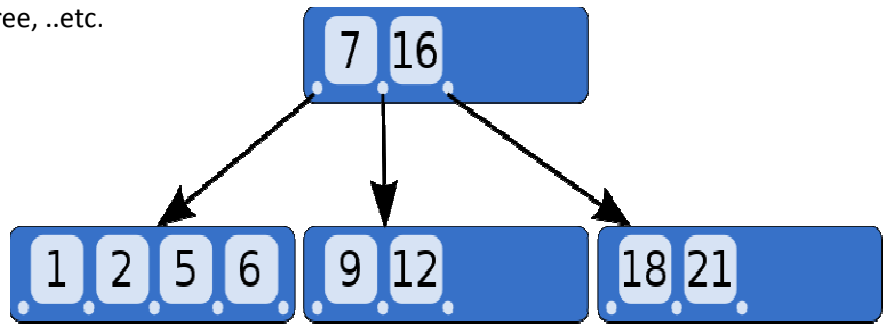


Move the middle element to the parent and split the current Node

B-Tree

B- tree		
Type	Tree	
Time complexity in big O notation		
	Average	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)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.



In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but maywaste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes aretypically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**),each internal node may have only 2 or 3 child nodes.

Variants

The term **B-tree** may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+-tree and the B*-tree.

• In the B+-tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access.(Comer 1979, p. 129)

• The B*-tree balances more neighboring internal nodes to keep the internal nodes more densely packed.(Comer 1979, p. 129) This variant requires non-root nodes to be at least 2/3 full instead of 1/2. (Knuth 1998, p. 488) To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three.

• Counted B-trees store, with each pointer within the tree, the number of nodes in the subtree below that pointer.[1]

This allows rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.

The B-tree uses all those ideas

The B-tree uses all of the above ideas:

• It keeps records in sorted order for sequential traversing

• It uses a hierarchical index to minimize the number of disk reads

• It uses partially full blocks to speed insertions and deletions

• The index is elegantly adjusted with a recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least ½ full. A B-tree can handle an arbitrary number of insertions and deletions.

Definition

According to Knuth's definition, a B-tree of order *m* (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every node (except root) has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with *k* children contains $k-1$ keys.
5. All leaves appear in the same level, and carry information.

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 separation values or elements: $a1$ and $a2$. All values in the leftmost subtree will be less than $a1$, all values in the middle subtree will be between $a1$ and $a2$, and all values in the rightmost subtree will be greater than $a2$.

Internal nodes

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and a **minimum** of *L* children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between $L-1$ and $U-1$). *U* must be either $2L$ or $2L-1$; therefore each internal node is at least half full. The relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there's room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

The root node

The root node's number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree, with no children at all.

Leaf nodes

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

A B-tree of depth $n+1$ can hold about U times as many items as a B-tree of depth n, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value. Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

Insertion

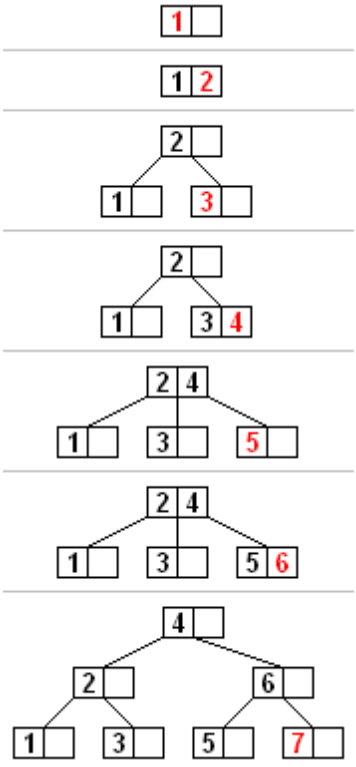
A B Tree insertion example with eachiteration. The nodes of this B tree have at most 3 children (Knuth order 3).

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.

2. Otherwise the node is full, evenly split it into two nodes so:
    1. A single median is chosen from among the leaf's elements and the new element.
    2. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.

    3. The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).



If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is $U-1$. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number $U-1$ of elements into two legal nodes. If this number is odd, then $U=2L$ and one of the new nodes contains $(U-2)/2 = L-1$ elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If $U-1$ is even, then $U=2L-1$, so there are $2L-2$ elements in the node. Half of this number is $L-1$, which is the minimum number of elements allowed per node.

Deletion

There are two popular strategies for deletion from a B-Tree.

1. Locate and delete the item, then restructure the tree to regain its invariants, **OR**
2. Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. The element in an internal node is a separator for its child nodes
2. Deleting an element may put its node under the minimum number of elements and children

The procedures for these cases are in order below.

Deletion from a leaf node

1. Search for the value to delete
2. If the value's in a leaf node, simply delete it from the node
3. If underflow happens, check siblings, and either transfer a key or fuse the siblings together
4. If deletion happened from right child, retrieve the max value of left child if it has no underflow
5. In vice-versa situation, retrieve the min element from right

Deletion from an internal node

1. If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator
2. This has deleted an element from a leaf node, and so is now equivalent to the previous case

Rebalancing after deletion

1. If the right sibling has more than the minimum number of elements
    1. Add the separator to the end of the deficient node
    2. Replace the separator in the parent with the first element of the right sibling
    3. Append the first child of the right sibling as the last child of the deficient node
2. Otherwise, if the left sibling has more than the minimum number of elements
    1. Add the separator to the start of the deficient node
    2. Replace the separator in the parent with the last element of the left sibling
    3. Insert the last child of the left sibling as the first child of the deficient node
3. If both immediate siblings have only the minimum number of elements
    1. Create a new node with all the elements from the deficient node, all the elements from one of its siblings, and the separator in the parent between the two combined sibling nodes
    2. Remove the separator from the parent, and replace the two children it separated with the combined node
    3. If that brings the number of elements in the parent under the minimum, repeat these steps with that deficient node, unless it is the root, since the root is permitted to be deficient

    The only other case to account for is when the root has no elements and one child. In this case it is sufficient to replace it with its only child.

We sketch how deletion works with various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

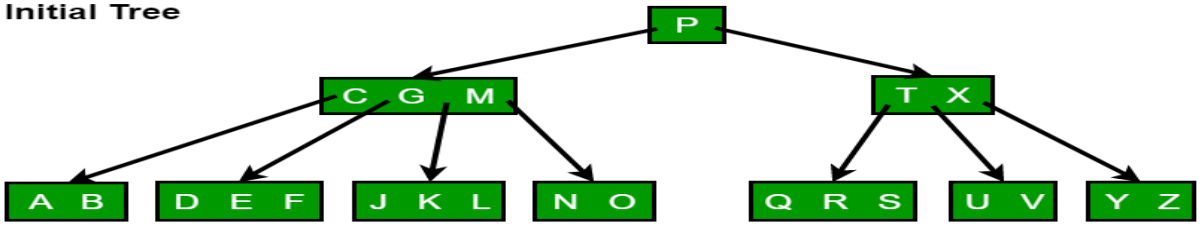
a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

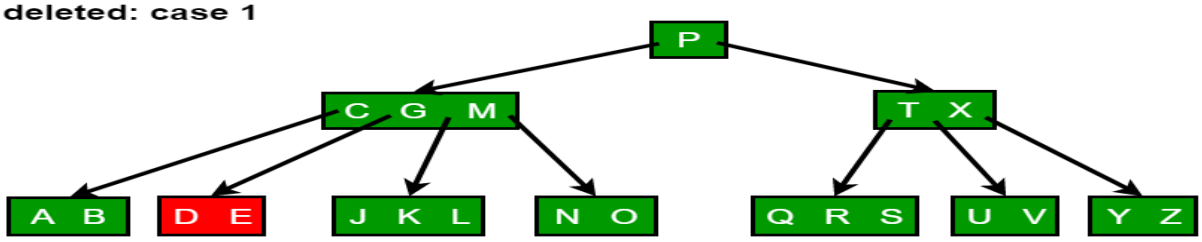
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may

have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).
The following figures explain the deletion process.

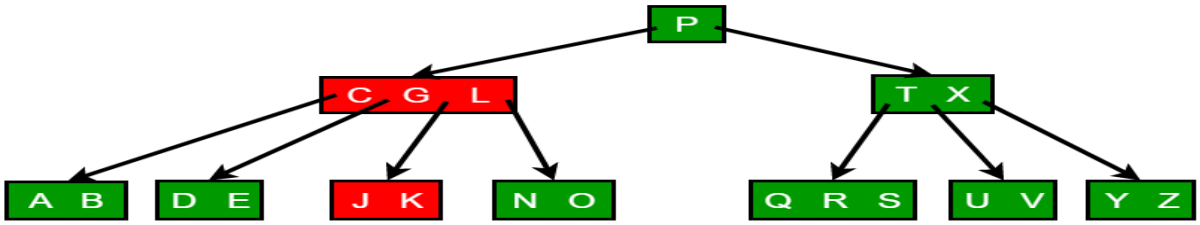
(a) Initial Tree



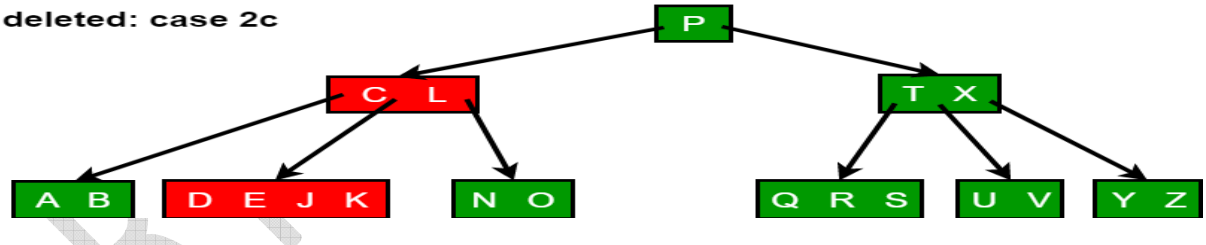
(b) F deleted: case 1



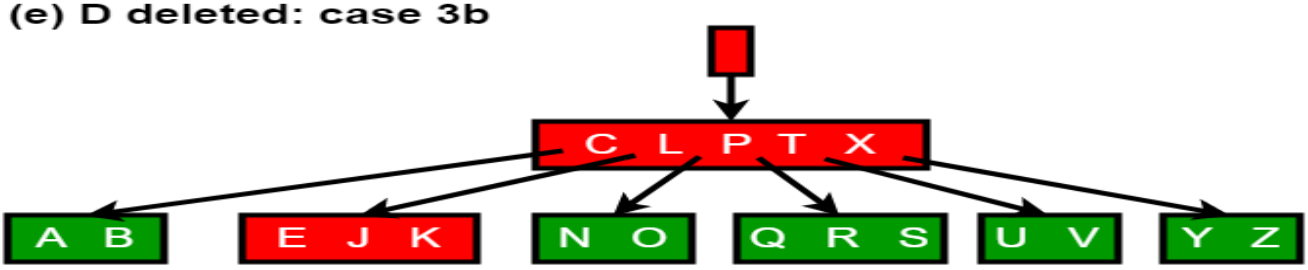
(c) M deleted: case 2a



(d) G deleted: case 2c

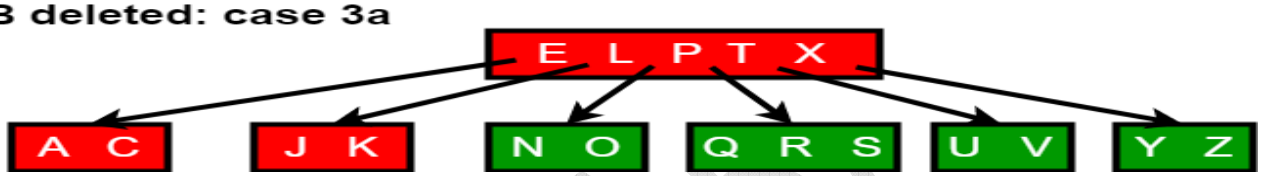


(e) D deleted: case 3b



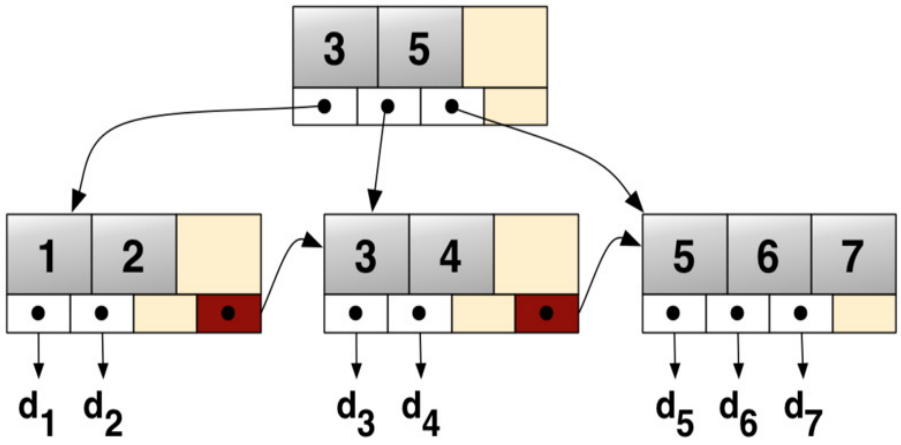
(e') tree shrinks in height

(f) B deleted: case 3a



B+ tree

In computer science, a **B+ tree** is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.



Node Type	Children Type	Min Children	Max Children	Example b =7	Example b =100
Root Node (when it is the only node in the tree)	Records	1	b	1 - 7	1 - 100
Root Node	Internal Nodes or Leaf Nodes	2	b	2 - 7	2 - 100
Internal Node	Internal Nodes or Leaf Nodes	Celling(b/2)	b	4 - 7	50 - 100
Leaf Node	Records	Floor(b/2)	b - 1	3 - 6	50 - 99

```
Function: search (k)
    return tree_search (k, root);
Function: tree_search (k, node)
    if node is a leaf then
        return node;
    switch k do
        case k < k_0
            return tree_search(k, p_0);
        case k_i ≤ k < k_{i+1}
            return tree_search(k, p_i);
        case k_d ≤ k
            return tree_search(k, p_d);
```

Insertion

Perform a search to determine what bucket the new record should go into.

- If the bucket is not full (at most b - 1 entries after the insertion), add the record.
- Otherwise, split the bucket.
    - Allocate new leaf and move half the bucket's elements to the new bucket.
    - Insert the new leaf's smallest key and address into the parent.
    - If the parent is full, split it too.
        - Add the middle key to the parent node.
    - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers.

B-trees grow at the root and not at the leaves.

Note that, for a non-leaf node split, we can simply push up the middle key (17). Contrast this with a leaf node split.

Deletion

- Start at root, find leaf L where entry belongs.
- Remove the entry.
    - If L is at least half-full, done!
    - If L has fewer entries than it should,
        - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
        - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

Characteristics

For a *b*-order B+ tree with *h* levels of index:

- The maximum number of records stored is $n_{max} = b^h - b^{h-1}$
- The minimum number of records stored is $n_{min} = 2 \left\lceil \frac{b}{2} \right\rceil^{h-1}$
- The minimum number of keys is $n_{kmin} = 2 \left\lceil \frac{b}{2} \right\rceil^{h} - 1$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations
- Finding a record requires $O(\log_b n)$ operations
- Removing a (previously located) record requires $O(\log_b n)$ operations

• Performing a range query with *k* elements occurring within the range requires $O(\log_b n + k)$ operations

• Performing a pagination query with page size *s* and page number *p* requires $O(p * s)$ operations

Advantage –
A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and presence of P_{next} pointers imply that B+ tree are very quick and efficient in accessing records from disks.

Segment Tree

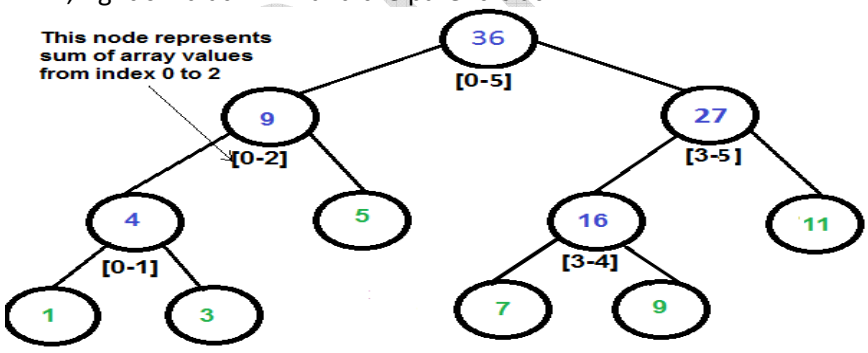
Let us consider the following problem to understand Segment Trees.
We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of elements from index l to r where 0 <= l <= r <= n-1
2 Change value of a specified element of the array to a new value x. We need to do arr[i] = x where 0 <= i <= n-1.

A simple solution is to run a loop from l to r and calculate the sum of elements in the given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and the second operation takes O(1) time.

Another solution is to create another array and store sum from start to i at the ith index in this array. The sum of a given range can now be calculated in O(1) time, but update operation takes O(n) time now. This works well if the number of query operations is large and very few updates.

What if the number of query and updates are equal? Can we perform both the operations in O(log n) time once given the array? We can use a Segment Tree to do both operations in O(Logn) time.

Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.
An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

How does above segment tree look in memory?
Like Heap, the segment tree is also represented as an array. The difference here is, it is not a complete binary tree. It is rather a full binary tree (every node has 0 or 2 children) and all levels are filled except possibly the last level. Unlike Heap, the last level may have gaps between nodes. Below are the values in the segment tree array for the above diagram.
Below is memory representation of segment tree for input array {1, 3, 5, 7, 9, 11}
st[] = {36, 9, 27, 4, 5, 16, 11, 1, 3, DUMMY, DUMMY, 7, 9, DUMMY, DUMMY}

The dummy values are never accessed and have no use. This is some wastage of space due to simple array representation. We may optimize this wastage using some clever implementations, but code for sum and update becomes more complex.

Construction of Segment Tree from given array

We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the sum in the corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always a full binary tree with n leaves, there will be n-1 internal nodes. So the total number of nodes will be 2*n – 1. Note that this does not include dummy nodes.

What is the total size of the array representing segment tree?

If n is a power of 2, then there are no dummy nodes. So the size of the segment tree is 2n-1 (n leaf nodes and n-1) internal nodes. If n is not a power of 2, then the size of the tree will be 2*x – 1 where x is the smallest power of 2 greater than n. For example, when n = 10, then size of array representing segment tree is 2*16-1 = 31.

An alternate explanation for size is based on heignt. Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. The following is the algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
   if the range of the node is within l and r
        return value in the node
   else if the range of the node is completely outside l and r
        return 0
   else
    return getSum(node's left child, l, r) +
           getSum(node's right child, l, r)
}
```

Update a value

Like tree construction and query operations, the update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from the root of the segment tree and add *diff* to all nodes which have given index in their range. If a node doesn't have a given index in its range, we don't make any changes to that node.

Implementation:

Following is the implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e -s)/2; }

/* A recursive function to get the sum of values in given range
```

```
        of the array. The following are parameters for this function.

        st --> Pointer to segment tree
        si --> Index of current node in the segment tree. Initially
                        0 is passed as root is always at index 0
        ss & se --> Starting and ending indexes of the segment represented
                                by current node, i.e., st[si]
        qs & qe --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
        // If segment of this node is a part of given range, then return
        // the sum of the segment
        if (qs <= ss && qe >= se)
                return st[si];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
                return 0;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
                getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
        st, si, ss and se are same as getSumUtil()
        i --> index of the element to be updated. This index is
                        in the input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
        // Base Case: If the input index lies outside the range of
        // this segment
        if (i < ss || i > se)
                return;

        // If the input index is in range of this node, then update
        // the value of the node and its children
        st[si] = st[si] + diff;
        if (se != ss)
        {
                int mid = getMid(ss, se);
                updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
                updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
        }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
        // Check for erroneous input index
        if (i < 0 || i > n-1)
```

```
        {
                printf("Invalid Input");
                return;
        }

        // Get the difference between new value and old value
        int diff = new_val - arr[i];

        // Update the value in array
        arr[i] = new_val;

        // Update the values of nodes in segment tree
        updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start)
// to qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
        // Check for erroneous input values
        if (qs < 0 || qe > n-1 || qs > qe)
        {
                printf("Invalid Input");
                return -1;
        }

        return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
        // If there is one element in array, store it in current node of
        // segment tree and return
        if (ss == se)
        {
                st[si] = arr[ss];
                return arr[ss];
        }

        // If there are more than one elements, then recur for left and
        // right subtrees and store the sum of values in this node
        int mid = getMid(ss, se);
        st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
                        constructSTUtil(arr, mid+1, se, st, si*2+2);
        return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
        // Allocate memory for the segment tree
```

```
        //Height of segment tree
        int x = (int)(ceil(log2(n)));

        //Maximum size of segment tree
        int max_size = 2*(int)pow(2, x) - 1;

        // Allocate memory
        int *st = new int[max_size];

        // Fill the allocated memory st
        constructSTUtil(arr, 0, n-1, st, 0);

        // Return the constructed segment tree
        return st;
}

// Driver program to test above functions
int main()
{
        int arr[] = {1, 3, 5, 7, 9, 11};
        int n = sizeof(arr)/sizeof(arr[0]);

        // Build segment tree from given array
        int *st = constructST(arr, n);

        // Print sum of values in array from index 1 to 3
        printf("Sum of values in given range = %dn",
                        getSum(st, n, 1, 3));

        // Update: set arr[1] = 10 and update corresponding
        // segment tree nodes
        updateValue(arr, st, n, 1, 10);

        // Find sum after the value is updated
        printf("Updated sum of values in given range = %dn",
                        getSum(st, n, 1, 3));
        return 0;
}
```

Output:

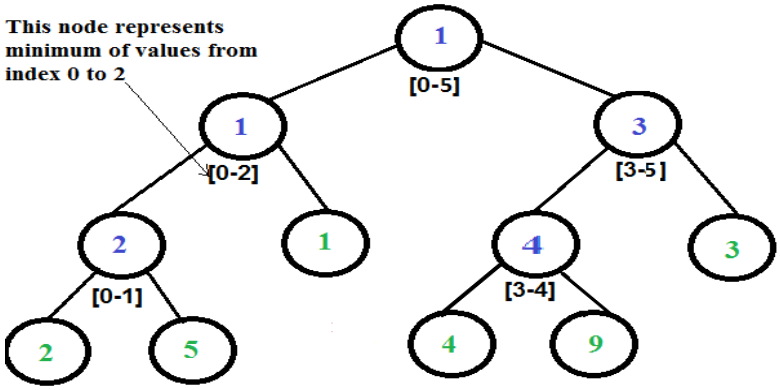
```
Sum of values in given range = 15
Updated sum of values in given range = 22
```

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.
Time complexity to query is O(Logn). To query a sum, we process at most four nodes at every level and number of levels is O(Logn).
The time complexity of update is also O(Logn). To update a leaf value, we process one node at every level and number of levels is O(Logn).

We have introduced segment tree with a simple example in the previous post. In this post, Range Minimum Query problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array arr[0 . . . n-1]. We should be able to efficiently find the minimum value from index *qs* (query start) to *qe* (query end) where *0 <= qs <= qe <= n-1*.

A **simple solution** is to run a loop from *qs* to *qe* and find minimum element in given range. This solution takes O(n) time in worst case.
Another solution is to create a 2D array where an entry [i, j] stores the minimum value in range arr[i..j]. Minimum of a given range can now be calculated in O(1) time, but preprocessing takes O(n^2) time. Also, this approach needs O(n^2) extra space which may become huge for large input arrays.
Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is O(n) and time to for range minimum query is O(Logn). The extra space required is O(n) to store the segment tree.
Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.
An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {2, 5, 1, 4, 9, 3 }

Construction of Segment Tree from given array
We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.
All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2*n – 1.
Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.
Query for minimum value of given range
Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e -s)/2; }

/* A recursive function to get the minimum value in a given range
        of array indexes. The following are parameters for this function.
```

```
        st --> Pointer to segment tree
        index --> Index of current node in the segment tree. Initially
                          0 is passed as root is always at index 0
        ss & se --> Starting and ending indexes of the segment represented
                                  by current node, i.e., st[index]
        qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
        // If segment of this node is a part of given range, then return
        // the min of the segment
        if (qs <= ss && qe >= se)
                return st[index];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
                return INT_MAX;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}
// Return minimum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
        // Check for erroneous input values
        if (qs < 0 || qe > n-1 || qs > qe)
        {
                printf("Invalid Input");
                return -1;
        }
        return RMQUtil(st, 0, n-1, qs, qe, 0);
}
// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
        // If there is one element in array, store it in current node of
        // segment tree and return
        if (ss == se)
        {
                st[si] = arr[ss];
                return arr[ss];
        }
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = getMid(ss, se);
        st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                                          constructSTUtil(arr, mid+1, se, st, si*2+2));
        return st[si];
}

/* Function to construct segment tree from given array. This function
```

```
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int)(ceil(log2(n)));

        // Maximum size of segment tree
        int max_size = 2*(int)pow(2, x) - 1;
        int *st = new int[max_size];

        // Fill the allocated memory st
        constructSTUtil(arr, 0, n-1, st, 0);

        // Return the constructed segment tree
        return st;
}

// Driver program to test above functions
int main()
{
        int arr[] = {1, 3, 2, 7, 9, 11};
        int n = sizeof(arr)/sizeof(arr[0]);

        // Build segment tree from given array
        int *st = constructST(arr, n);

        int qs = 1; // Starting index of query range
        int qe = 5; // Ending index of query range

        // Print minimum value in arr[qs..qe]
        printf("Minimum of values in range [%d, %d] is = %d\n",
                                        qs, qe, RMQ(st, n, qs, qe));

        return 0;
}
```

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.
Time complexity to query is O(Logn). To query a range minimum, we process at most two nodes at every level and number of levels is O(Logn).

K Dimensional Tree

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.
Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

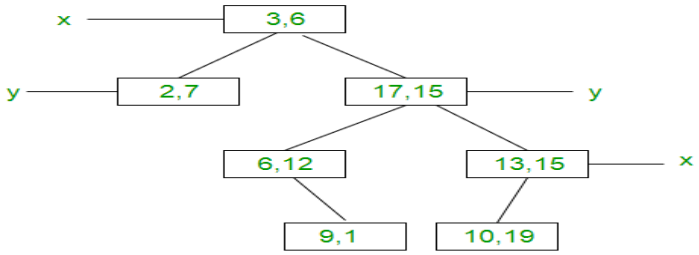
For the sake of simplicity, let us understand a 2-D Tree with an example.
The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization:
Let us number the planes as 0, 1, 2, …(K – 1). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:
A = D mod K

How to determine if a point will lie in the left subtree or in right subtree?
If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

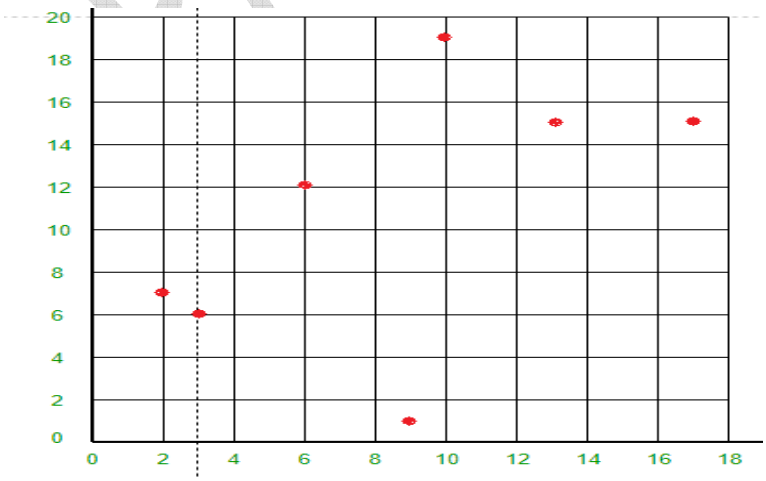
Creation of a 2-D Tree:
Consider following points in a 2-D plane:
(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the rightsubtree or in the right subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 < 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1):Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7):Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).

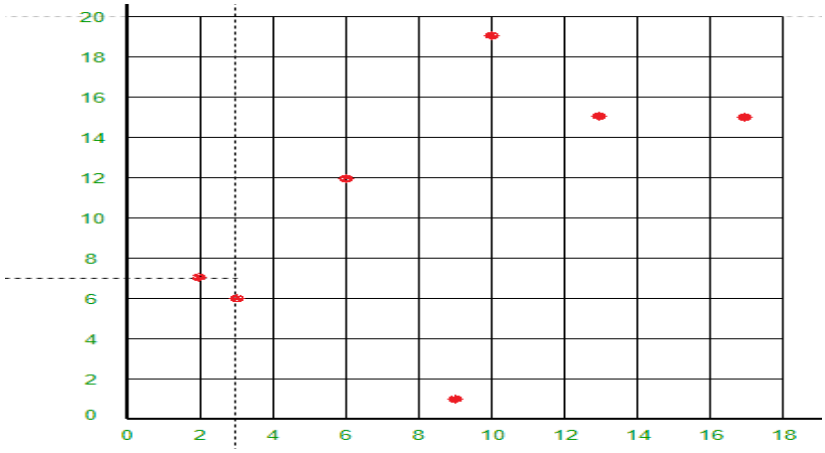


How is space partitioned?
All 7 points will be plotted in the X-Y plane as follows:

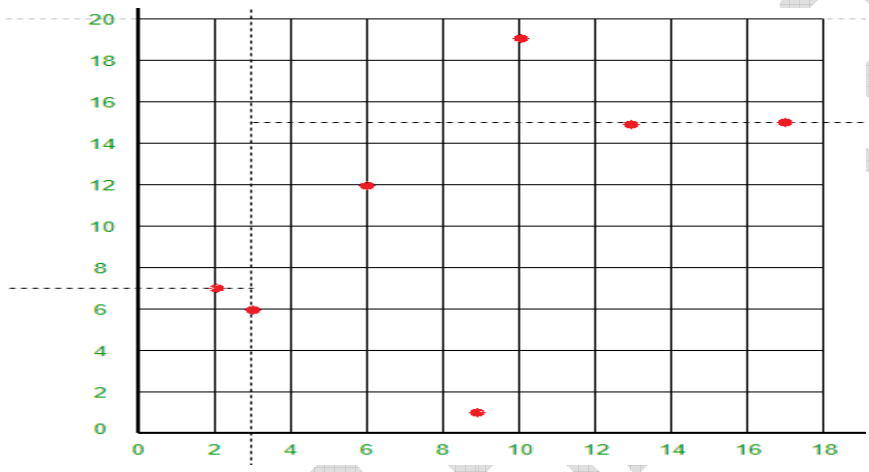
1. Point (3, 6) will divide the space into two parts: Draw line X = 3.



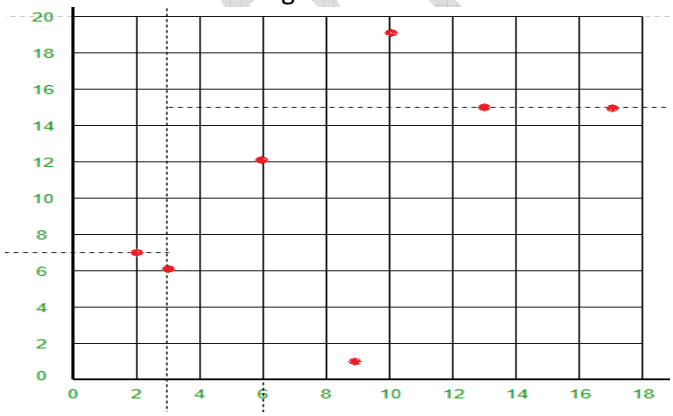
Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally.
Draw line Y = 7 to the left of line X = 3.



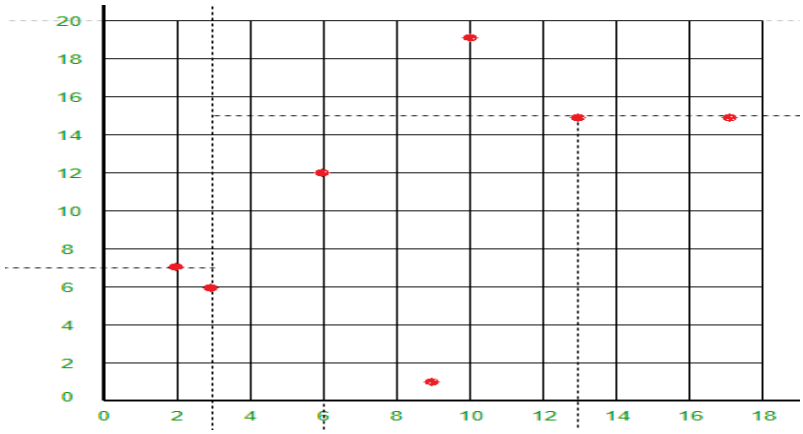
Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally.
Draw line Y = 15 to the right of line X = 3.



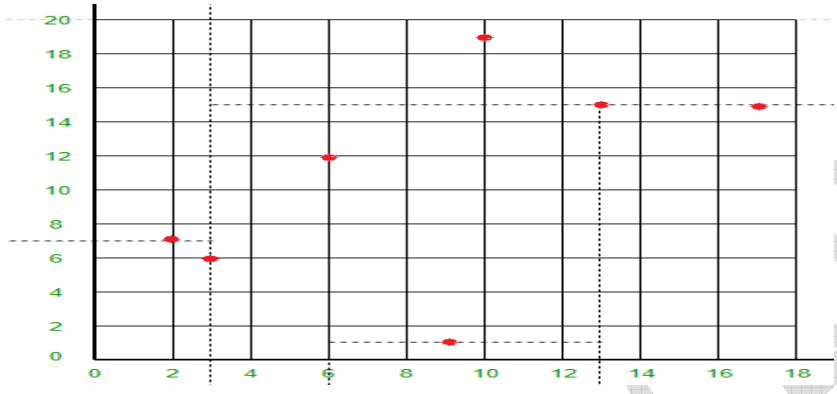
Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts.
Draw line X = 6 to the right of line X = 3 and below line Y = 15.



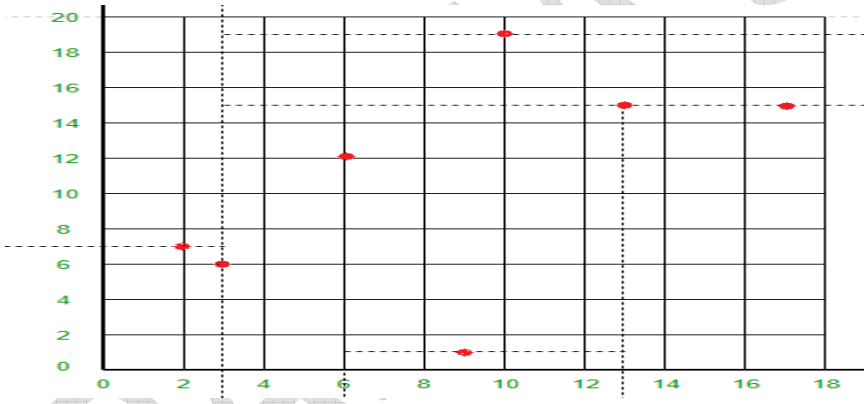
Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts.
Draw line X = 13 to the right of line X = 6 and below line Y = 15.



Point (9, 1) will divide the space between lines X = 3, X = 6 and Y = 15 into two parts.
Draw line Y = 1 between lines X = 3 and X = 6.



Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts.
Draw line Y = 19 to the right of line X = 3 and above line Y = 15.



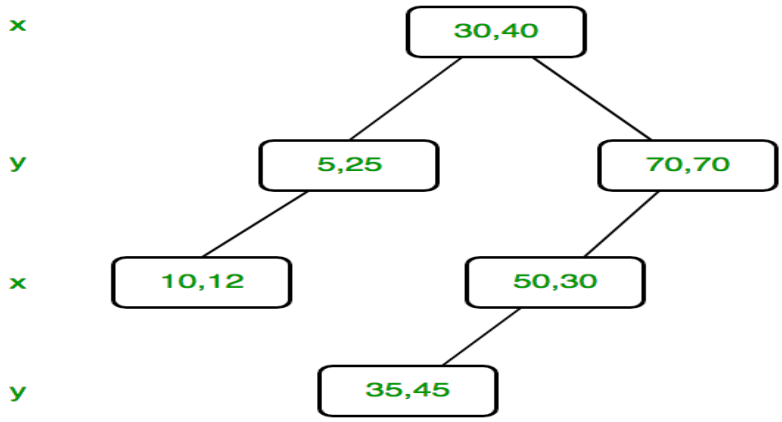
Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
	int point[k]; // To store k dimensional point
	Node *left, *right;
};
```

```
// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
	struct Node* temp = new Node;

	for (int i=0; i<k; i++)
	temp->point[i] = arr[i];

	temp->left = temp->right = NULL;
	return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
	// Tree is empty?
	if (root == NULL)
	return newNode(point);

	// Calculate current dimension (cd) of comparison
	unsigned cd = depth % k;

	// Compare the new point with root on current dimension 'cd'
	// and decide the left or right subtree
	if (point[cd] < (root->point[cd]))
		root->left = insertRec(root->left, point, depth + 1);
	else
		root->right = insertRec(root->right, point, depth + 1);

	return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
	return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
	// Compare individual pointinate values
	for (int i = 0; i < k; ++i)
		if (point1[i] != point2[i])
			return false;

	return true;
}

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
```

```
bool searchRec(Node* root, int point[], unsigned depth)
{
	// Base cases
	if (root == NULL)
		return false;
	if (arePointsSame(root->point, point))
		return true;

	// Current dimension is computed using current depth and total
	// dimensions (k)
	unsigned cd = depth % k;

	// Compare point with root with respect to cd (Current dimension)
	if (point[cd] < root->point[cd])
		return searchRec(root->left, point, depth + 1);

	return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
	// Pass current depth as 0
	return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
	struct Node *root = NULL;
	int points[][k] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
					{9, 1}, {2, 7}, {10, 19}};

	int n = sizeof(points)/sizeof(points[0]);

	for (int i=0; i<n; i++)
	root = insert(root, points[i]);

	int point1[] = {10, 19};
	(search(root, point1))? cout << "Found\n": cout << "Not Found\n";

	int point2[] = {12, 19};
	(search(root, point2))? cout << "Found\n": cout << "Not Found\n";

	return 0;
}
```

The operation is to find minimum in the given dimension. This is especially needed in delete operation.
For example, consider below KD Tree, if given dimension is x, then output should be 5 and if given dimensions is y, then output should be 12.



In KD tree, points are divided dimension by dimension. For example, root divides keys by dimension 0, level next to root divides by dimension 1, next level by dimension 2 if k is more then 2 (else by dimension 0), and so on.

To find minimum we traverse nodes starting from root. ***If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child***. This is same as Binary Search Tree Minimum.
Above is simple, what to do when current level's dimension is different. ***When dimension of current level is different, minimum may be either in left subtree or right subtree or current node may also be minimum***. So we take minimum of three and return. This is different from Binary Search tree.

The operation is to delete a given point from K D Tree.
Like Binary Search Tree Delete, we recursively traverse down and search for the point to be deleted. Below are steps are followed for every node visited.

1) If current node contains the point to be deleted

a. If node to be deleted is a leaf node, simply delete it (Same as BST Delete)
b. If node to be deleted has right child as not NULL (Different from BST)
    1) Find minimum of current node's dimension in right subtree.
    2) Replace the node with above found minimum and recursively delete minimum in right subtree.
c. Else If node to be deleted has left child as not NULL (Different from BST)
    1) Find minimum of current node's dimension in left subtree.
    2)Replace the node with above found minimum and recursively delete minimum in left subtree.
    3)Make new left subtree as right child of current node.

2) If current doesn't contain the point to be deleted

a. If node to be deleted is smaller than current node on current dimension, recur for left subtree.
b. Else recur for right subtree.

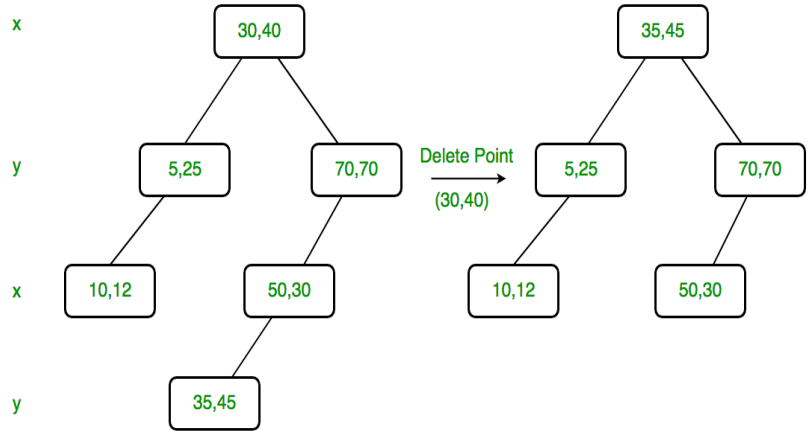
Why 1.b and 1.c are different from BST?

In BST delete, if a node's left child is empty and right is not empty, we replace the node with right child. In K D Tree, doing this would violate the KD tree property as dimension of right child of node is different from node's dimension. For example, if node divides point by x axis values. then its children divide by y axis, so we can't simply replace node with right child. Same is true for the case when right child is not empty and left child is empty.
Why 1.c doesn't find max in left subtree and recur for max like 1.b?
Doing this violates the property that all equal values are in right subtree. For example, if we delete (!0, 10) in below subtree and replace if with

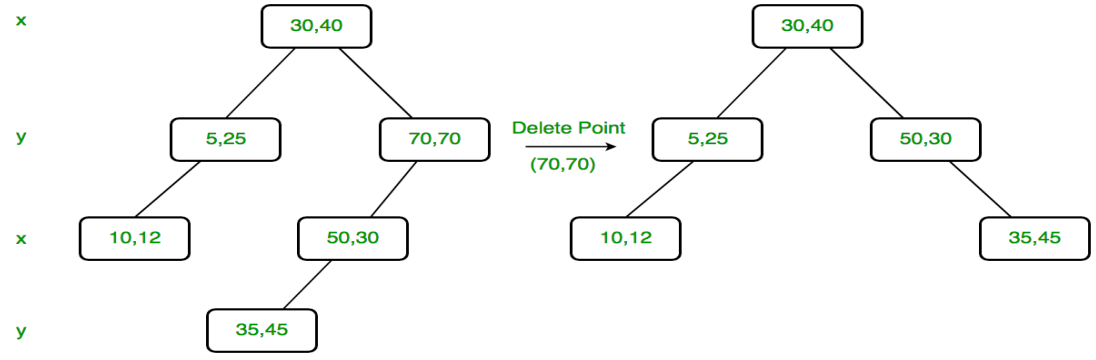
```
Wrong Way (Equal key in left subtree after deletion)
            (5, 6)                                (4, 10)
             /              Delete(5, 6)           /
        (4, 10)            ------------>      (4, 20)
             \
```

```
          (4, 20)


Right way (Equal key in right subtree after deletion)
            (5, 6)                          (4, 10)
            /              Delete(5, 6)           \
        (4, 10)            ------------>        (4, 20)
             \
            (4, 20)
```

Delete (30, 40): Since right child is not NULL and dimension of node is x, we find the node with minimum x value in right child. The node is (35, 45), we replace (30, 40) with (35, 45) and delete (35, 45).



Delete (70, 70): Dimension of node is y. Since right child is NULL, we find the node with minimum y value in left child. The node is (50, 30), we replace (70, 70) with (50, 30) and recursively delete (50, 30) in left subtree. Finally we make the modified left subtree as right subtree of (50, 30).




```
// A C++ program to demonstrate delete in K D tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
        int point[k]; // To store k dimensional point
        Node *left, *right;
```

```
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
	struct Node* temp = new Node;

	for (int i=0; i<k; i++)
		temp->point[i] = arr[i];

	temp->left = temp->right = NULL;
	return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
	// Tree is empty?
	if (root == NULL)
		return newNode(point);

	// Calculate current dimension (cd) of comparison
	unsigned cd = depth % k;

	// Compare the new point with root on current dimension 'cd'
	// and decide the left or right subtree
	if (point[cd] < (root->point[cd]))
		root->left = insertRec(root->left, point, depth + 1);
	else
		root->right = insertRec(root->right, point, depth + 1);

	return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
	return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
Node *minNode(Node *x, Node *y, Node *z, int d)
{
	Node *res = x;
	if (y != NULL && y->point[d] < res->point[d])
	res = y;
	if (z != NULL && z->point[d] < res->point[d])
	res = z;
	return res;
}

// Recursively finds minimum of d'th dimension in KD tree
```

```
// The parameter depth is used to determine current axis.
Node *findMinRec(Node* root, int d, unsigned depth)
{
	// Base cases
	if (root == NULL)
		return NULL;

	// Current dimension is computed using current depth and total
	// dimensions (k)
	unsigned cd = depth % k;

	// Compare point with root with respect to cd (Current dimension)
	if (cd == d)
	{
		if (root->left == NULL)
			return root;
		return findMinRec(root->left, d, depth+1);
	}

	// If current dimension is different then minimum can be anywhere
	// in this subtree
	return minNode(root,
			findMinRec(root->left, d, depth+1),
			findMinRec(root->right, d, depth+1), d);
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
Node *findMin(Node* root, int d)
{
	// Pass current level or depth as 0
	return findMinRec(root, d, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
	// Compare individual pointinate values
	for (int i = 0; i < k; ++i)
		if (point1[i] != point2[i])
			return false;

	return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
for (int i=0; i<k; i++)
	p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
```

```
Node *deleteNodeRec(Node *root, int point[], int depth)
{
	// Given point is not present
	if (root == NULL)
		return NULL;

	// Find dimension of current node
	int cd = depth % k;

	// If the point to be deleted is present at root
	if (arePointsSame(root->point, point))
	{
		// 2.b) If right child is not NULL
		if (root->right != NULL)
		{
			// Find minimum of root's dimension in right subtree
			Node *min = findMin(root->right, cd);

			// Copy the minimum to root
			copyPoint(root->point, min->point);

			// Recursively delete the minimum
			root->right = deleteNodeRec(root->right, min->point, depth+1);
		}
		else if (root->left != NULL) // same as above
		{
			Node *min = findMin(root->left, cd);
			copyPoint(root->point, min->point);
			root->right = deleteNodeRec(root->left, min->point, depth+1);
		}
		else // If node to be deleted is leaf node
		{
			delete root;
			return NULL;
		}
		return root;
	}

	// 2) If current node doesn't contain point, search downward
	if (point[cd] < root->point[cd])
		root->left = deleteNodeRec(root->left, point, depth+1);
	else
		root->right = deleteNodeRec(root->right, point, depth+1);
	return root;
}

// Function to delete a given point from K D Tree with 'root'
Node* deleteNode(Node *root, int point[])
{
// Pass depth as 0
return deleteNodeRec(root, point, 0);
}

// Driver program to test above functions
int main()
```

```
{
        struct Node *root = NULL;
        int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                                        {10, 12}, {50, 30}, {35, 45}};

        int n = sizeof(points)/sizeof(points[0]);

        for (int i=0; i<n; i++)
                root = insert(root, points[i]);

        // Delet (30, 40);
        root = deleteNode(root, points[0]);

        cout << "Root after deletion of (30, 40)\n";
        cout << root->point[0] << ", " << root->point[1] << endl;

        return 0;
}
```

```
Root after deletion of (30, 40)

35, 45
```