Contents

Contents

Chapter 1

A program to check if a binary tree is BST or not

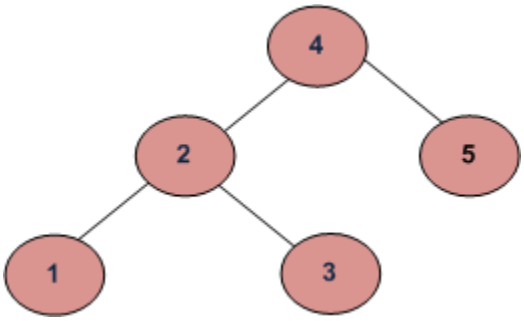
A program to check if a binary tree is BST or not - GeeksforGeeks

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

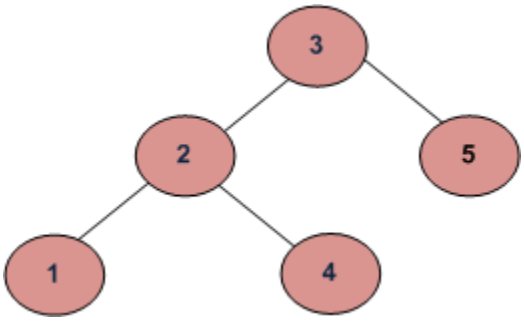


METHOD 1 (Simple but Wrong)
Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
 int isBST(struct node* node)
{
  if (node == NULL)
    return 1;
```

```
  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;

  /* false if right is < than node */
  if (node->right != NULL && node->right->data < node->data)
    return 0;

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return 0;

  /* passing all that, it's a BST */
  return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)
For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
 /* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);
```

```
  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
  return(true);
}
```

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)
Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
```

Implementation:
C

```
 #include <stdio.h>
#include <stdlib.h>
#include <limits.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;
};

int isBSTUtil(struct node* node, int min, int max);
```

```
/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
  /* an empty tree is BST */
  if (node==NULL)
     return 1;

  /* false if this node violates the min/max constraint */
  if (node->data < min || node->data > max)
     return 0;

  /* otherwise check the subtrees recursively,
   tightening the min or max constraint */
  return
    isBSTUtil(node->left, min, node->data-1) &&  // Allow only distinct values
    isBSTUtil(node->right, node->data+1, max);  // Allow only distinct values
}

/* 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);
}

/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(4);
  root->left        = newNode(2);
  root->right       = newNode(5);
  root->left->left  = newNode(1);
  root->left->right = newNode(3);
```

```
  if(isBST(root))
    printf("Is BST");
  else
    printf("Not a BST");

  getchar();
  return 0;
}
```

Java

```
 //Java implementation to check if given Binary tree
//is a BST or not

/* Class containing left and right child of current
 node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    /* can give min and max value according to your code or
    can write a function to find min and max value of tree. */

    /* returns true if given search tree is binary
     search tree (efficient version) */
    boolean isBST()  {
        return isBSTUtil(root, Integer.MIN_VALUE,
                               Integer.MAX_VALUE);
    }

    /* Returns true if the given tree is a BST and its
      values are >= min and <= max. */
    boolean isBSTUtil(Node node, int min, int max)
    {
```

```
        /* an empty tree is BST */
        if (node == null)
            return true;

        /* false if this node violates the min/max constraints */
        if (node.data < min || node.data > max)
            return false;

        /* otherwise check the subtrees recursively
        tightening the min/max constraints */
        // Allow only distinct values
        return (isBSTUtil(node.left, min, node.data-1) &&
                isBSTUtil(node.right, node.data+1, max));
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(5);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);

        if (tree.isBST())
            System.out.println("IS BST");
        else
            System.out.println("Not a BST");
    }
}
```

Python

```
 # Python program to check if a binary tree is bst or not

INT_MAX = 4294967296
INT_MIN = -4294967296

# A binary tree node
class Node:

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

```
# Returns true if the given tree is a binary search tree
# (efficient version)
def isBST(node):
    return (isBSTUtil(node, INT_MIN, INT_MAX))

# Retusn true if the given tree is a BST and its values
# >= min and <= max
def isBSTUtil(node, mini, maxi):

    # An empty tree is BST
    if node is None:
        return True

    # False if this node violates min/max constraint
    if node.data < mini or node.data > maxi:
        return False

    # Otherwise check the subtrees recursively
    # tightening the min or max constraint
    return (isBSTUtil(node.left, mini, node.data -1) and
          isBSTUtil(node.right, node.data+1, maxi))

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

if (isBST(root)):
    print "Is BST"
else:
    print "Not a BST"

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

Time Complexity: O(n)
Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

Simplified Method 3
We can simplify method 2 using NULL pointers instead of INT_MIN and INT_MAX values.

```
 // C++ program to check if a given tree is BST.
#include <bits/stdc++.h>
using namespace std;
```

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

// Returns true if given tree is BST.
bool isBST(Node* root, Node* l=NULL, Node* r=NULL)
{
    // Base condition
    if (root == NULL)
        return true;

    // if left node exist then check it has
    // correct data or not i.e. left node's data
    // should be less than root's data
    if (l != NULL and root->data < l->data)
        return false;

    // if right node exist then check it has
    // correct data or not i.e. right node's data
    // should be greater than root's data
    if (r != NULL and root->data > r->data)
        return false;

    // check recursively for every node.
    return isBST(root->left, l, root) and
           isBST(root->right, root, r);
}

/* 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 = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = newNode(3);
    root->left        = newNode(2);
    root->right       = newNode(5);
```

```
    root->left->left  = newNode(1);
    root->left->right = newNode(4);

    if (isBST(root,NULL,NULL))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

Output :

```
Not a BST
```

Thanks to **Abhinesh Garhwal** for suggesting above solution.

METHOD 4(Using In-Order Traversal)
Thanks to *LJW489* for suggesting this method.
1) Do In-Order Traversal of the given tree and store the result in a temp array.
3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to *ygos* for this space optimization.

C

```
 bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
          return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
          return false;

        prev = root;

        return isBST(root->right);
    }
```

```
    return true;
}
```

Java

```
 // Java implementation to check if given Binary tree
// is a BST or not

/* Class containing left and right child of current
 node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    // To keep tract of previous node in Inorder Traversal
    Node prev;

    boolean isBST()  {
        prev = null;
        return isBST(root);
    }

    /* Returns true if given search tree is binary
       search tree (efficient version) */
    boolean isBST(Node node)
    {
        // traverse the tree in inorder fashion and
        // keep a track of previous node
        if (node != null)
        {
            if (!isBST(node.left))
                return false;

            // allows only distinct values node
```

```
            if (prev != null && node.data <= prev.data )
                return false;
            prev = node;
            return isBST(node.right);
        }
        return true;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(5);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);

        if (tree.isBST())
            System.out.println("IS BST");
        else
            System.out.println("Not a BST");
    }
}
```

Python3

```
 # Python implementation to check if
# given Binary tree is a BST or not

# A binary tree node containing data
# field, left and right pointers
class Node:
    # constructor to create new node
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None

# global variable prev - to keep track
# of previous node during Inorder
# traversal
prev = None

# function to check if given binary
# tree is BST
def isbst(root):
```

```
    # prev is a global variable
    global prev
    prev = None
    return isbst_rec(root)


# Helper function to test if binary
# tree is BST
# Traverse the tree in inorder fashion
# and keep track of previous node
# return true if tree is Binary
# search tree otherwise false
def isbst_rec(root):

    # prev is a global variable
    global prev

    # if tree is empty return true
    if root is None:
        return True

    if isbst_rec(root.left) is False:
        return False

    # if previous node'data is found
    # greater than the current node's
    # data return fals
    if prev is not None and prev.data > root.data:
        return False

    # store the current node in prev
    prev = root
    return isbst_rec(root.right)


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

if isbst(root):
    print("is BST")
else:
    print("not a BST")

# This code is contributed by
```

```
# Shweta Singh(shweta44)
```

The use of static variable can also be avoided by using reference to prev node as a parameter.

```
 // C++ program to check if a given tree is BST.
#include <bits/stdc++.h>
using namespace std;

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

    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};


bool isBSTUtil(struct Node* root, Node *&prev)
{
    // traverse the tree in inorder fashion and
    // keep track of prev node
    if (root)
    {
        if (!isBSTUtil(root->left, prev))
          return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
          return false;

        prev = root;

        return isBSTUtil(root->right, prev);
    }

    return true;
}

bool isBST(Node *root)
{
   Node *prev = NULL;
   return isBSTUtil(root, prev);
```

```
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(3);
    root->left     = new Node(2);
    root->right     = new Node(5);
    root->left->left = new Node(1);
    root->left->right = new Node(4);

    if (isBST(root))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

Sources:
http://en.wikipedia.org/wiki/Binary_search_tree
http://cslibrary.stanford.edu/110/BinaryTrees.html

Improved By : shweta44, ChandrahasAbburi

Source

https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/

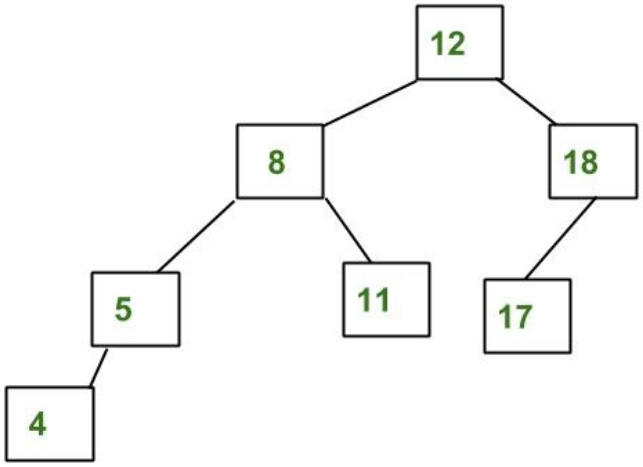
Chapter 2

AVL Tree Set 1 (Insertion)

AVL Tree Set 1 (Insertion) - GeeksforGeeks

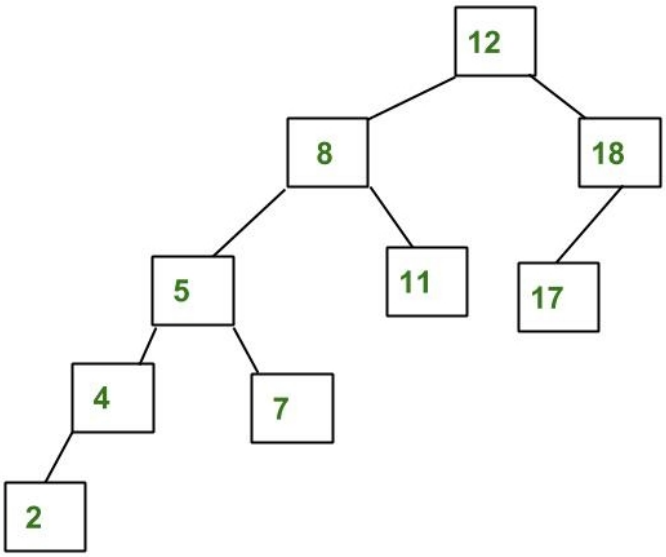
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

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 thisvideo lecture for proof).

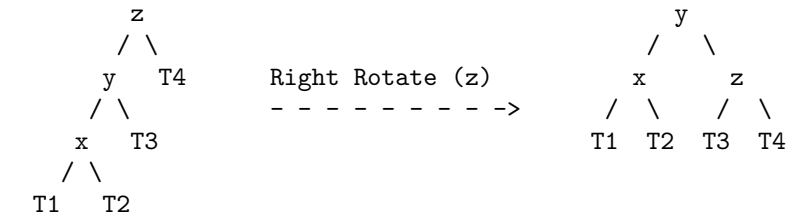
Insertion
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).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

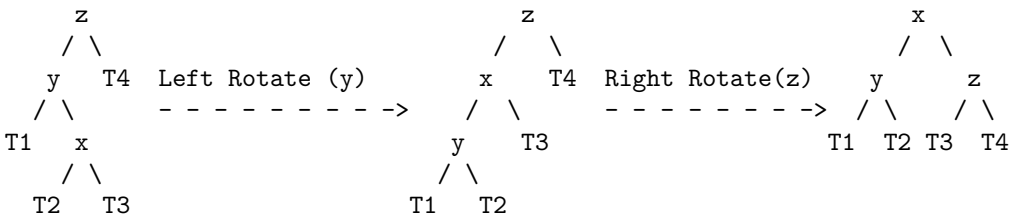
Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on 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 thisvideo lecture for proof)

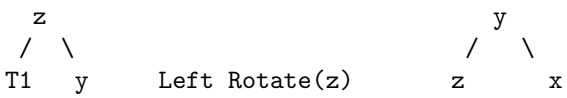
a) Left Left Case



b) Left Right Case



c) Right Right Case

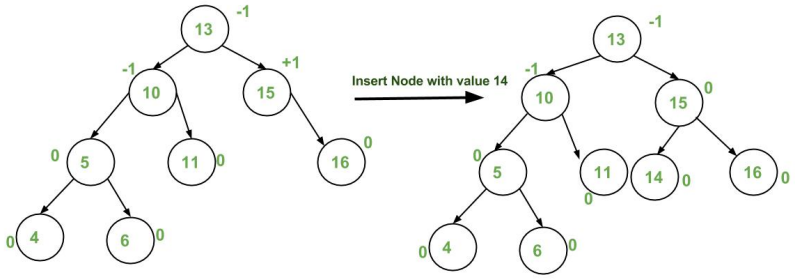


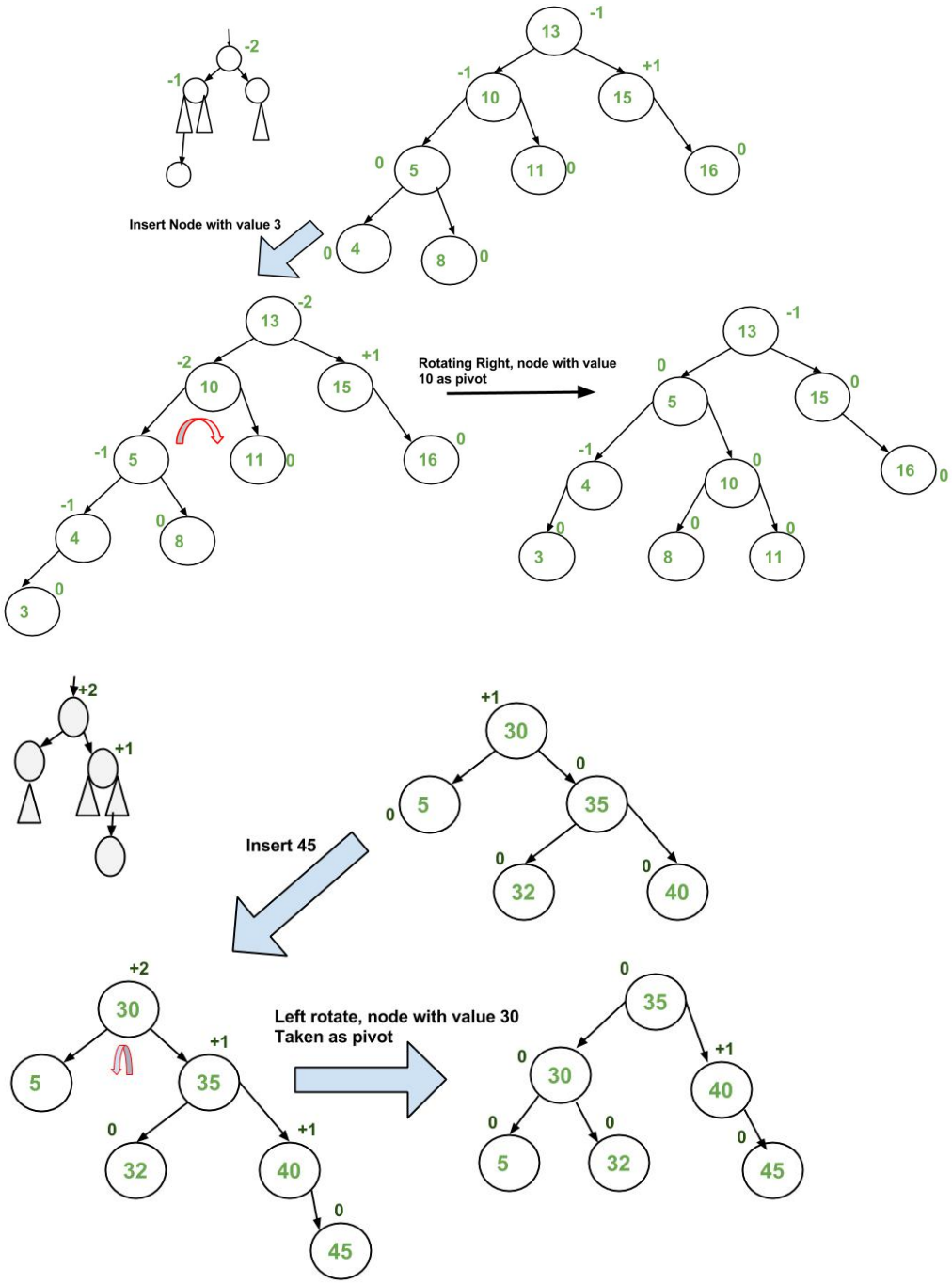
```
    /  \   - - - - - - - - ->    / \    / \
   T2   x                      T1  T2 T3  T4
       / \
     T3  T4
```

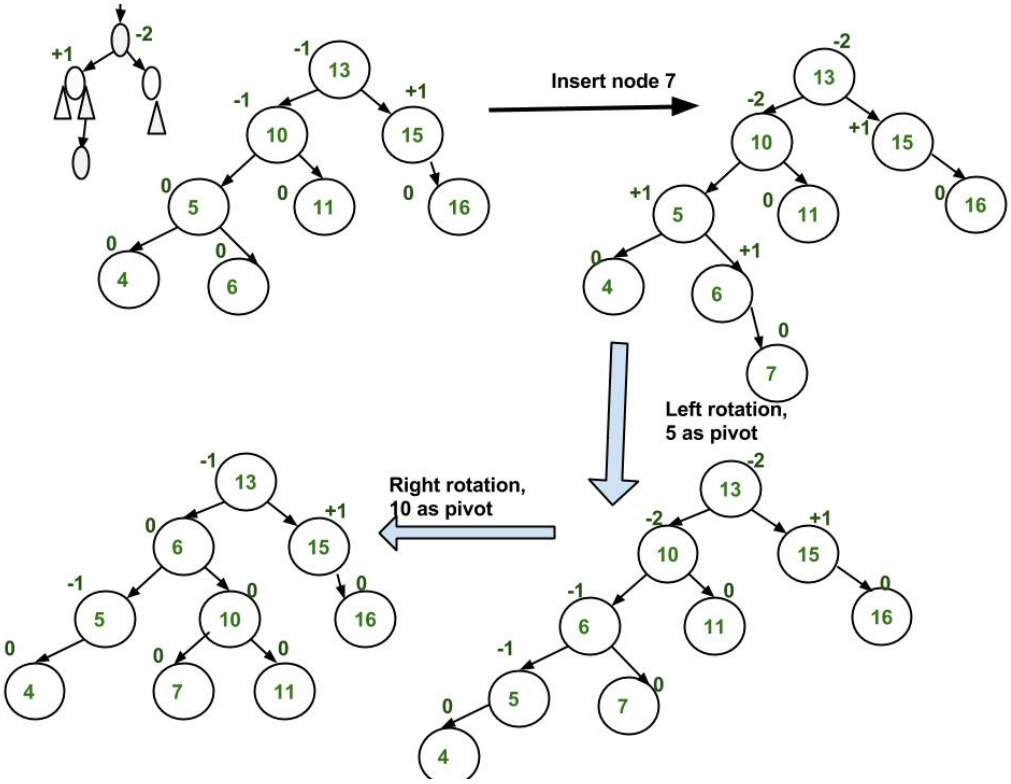
d) Right Left Case

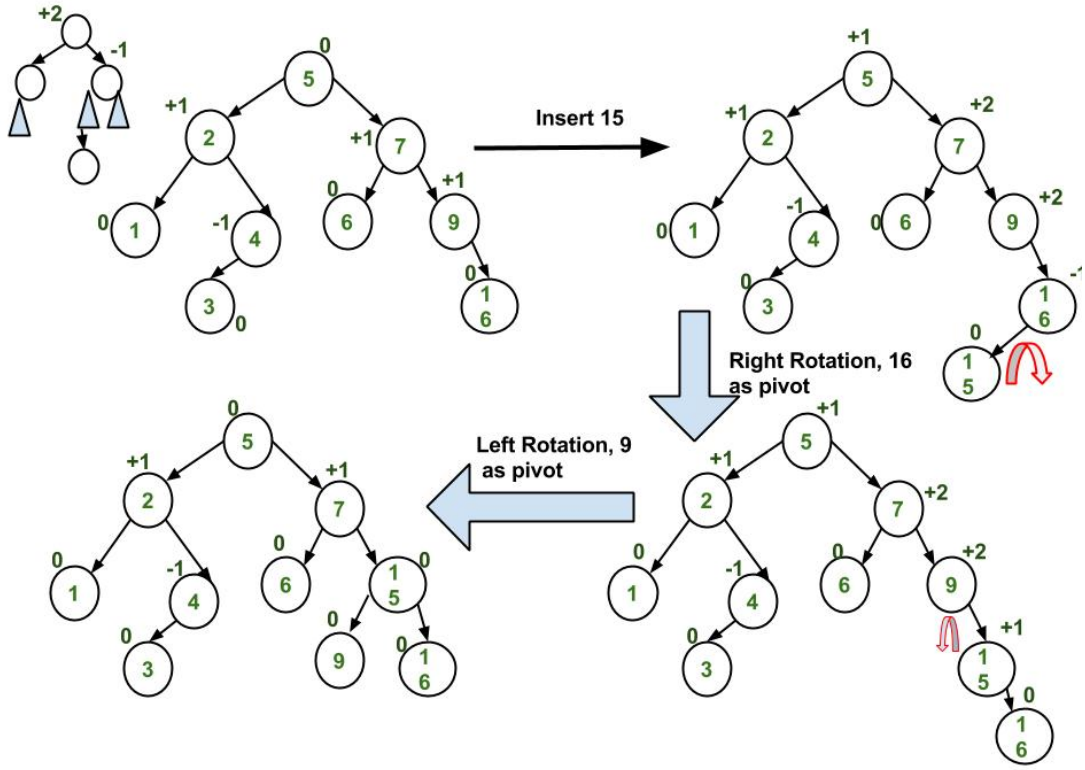
```
   z                              z                              x
  / \                            / \                            /  \
T1   y   Right Rotate (y)     T1   x      Left Rotate(z)     z      y
    / \  - - - - - - - - - ->     /  \   - - - - - - - - ->  / \    / \
   x   T4                        T2   y                    T1  T2  T3  T4
  / \                                /  \
T2   T3                            T3   T4
```

Insertion Examples:




Insert Node with value 3
Rotating Right, node with value 10 as pivot
Insert 45
Left rotate, node with value 30
Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot



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

```
 // 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;
}
```

Java

```
 // Java program for insertion in AVL Tree
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}
```

```
class AVLTree {

    Node root;

    // A utility function to get the height of the tree
    int height(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;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        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.
    Node leftRotate(Node x) {
        Node y = x.right;
        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(Node N) {
    if (N == null)
        return 0;

    return height(N.left) - height(N.right);
}

Node insert(Node node, int key) {

    /* 1.  Perform the normal BST insertion */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Duplicate 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;
    }

    // A utility function to print preorder traversal
    // of the tree.
    // The function also prints height of every node
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public static void main(String[] args) {
        AVLTree tree = new AVLTree();

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

        /* The constructed AVL Tree would be
             30
            /  \
          20   40
         /  \     \
        10  25    50
        */
        System.out.println("Preorder traversal" +
                           " of constructed tree is : ");
        tree.preOrder(tree.root);
    }
}
// This code has been contributed by Mayank Jaiswal
```

Python3

```
 # 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()

# This code is contributed by Ajitesh Pathak
```

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).

Comparison with Red Black Tree
The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. 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 the more frequent operation, then AVL tree should be preferred overRed Black Tree.

Following is the post for delete.
AVL Tree Set 2 (Deletion)

Following are some posts that have used self-balancing search trees.

Median in a stream of integers (running integers)
Maximum of all subarrays of size k
Count smaller elements on right side

References:
IITD Video Lecture on AVL Tree Introduction
IITD Video Lecture on AVL Tree Insertion and Deletion

Source

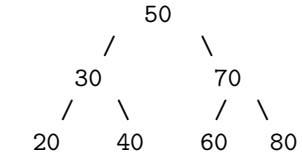
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Chapter 3

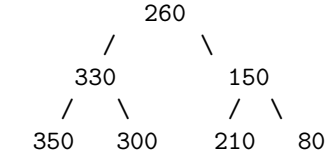
Add all greater values to every node in a given BST

Add all greater values to every node in a given BST - GeeksforGeeks

Given a **B**inary **S**earch **T**ree (BST), modify it so that all greater values in the given BST are added to every node. For example, consider the following BST.




```
The above tree should be modified to following
```



A **simple method** for solving this is to find sum of all greater values for every node. This method would take O(n^2) time.
We can do it **using a single traversal**. The idea is to use following BST property. If we do reverse Inorder traversal of BST, we get all nodes in decreasing order. We do reverse Inorder traversal and keep track of the sum of all nodes visited so far, we add this sum to every node.

C

```
// C program to add all greater values in every node of BST
```

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    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->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to add all greater values in every node
void modifyBSTUtil(struct Node *root, int *sum)
{
    // Base Case
    if (root == NULL)  return;

    // Recur for right subtree
    modifyBSTUtil(root->right, sum);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    *sum = *sum + root->data;
    root->data = *sum;

    // Recur for left subtree
    modifyBSTUtil(root->left, sum);
}

// A wrapper over modifyBSTUtil()
void modifyBST(struct Node *root)
{
    int sum = 0;
    modifyBSTUtil(root, &sum);
}

// A utility function to do inorder traversal of BST
void inorder(struct Node *root)
{
    if (root != NULL)
    {
```

```
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

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

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

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

// 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);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    modifyBST(root);

    // print inoder tarversal of the modified BST
    inorder(root);

    return 0;
}
```

Java

```
 // Java code to add all greater values to
// every node in a given BST

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // Inorder traversal of the tree
    void inorder()
    {
        inorderUtil(this.root);
    }

    // Utility function for inorder traversal of
    // the tree
    void inorderUtil(Node node)
    {
        if (node == null)
            return;

        inorderUtil(node.left);
        System.out.print(node.data + " ");
        inorderUtil(node.right);
    }

    // adding new node
```

```
public void insert(int data)
{
    this.root = this.insertRec(this.root, data);
}

/* A utility function to insert a new node with
given data in BST */
Node insertRec(Node node, int data)
{
    /* If the tree is empty, return a new node */
    if (node == null) {
        this.root = new Node(data);
        return this.root;
    }

    /* Otherwise, recur down the tree */
    if (data <= node.data) {
        node.left = this.insertRec(node.left, data);
    } else {
        node.right = this.insertRec(node.right, data);
    }
    return node;
}

// This class initialises the value of sum to 0
public class Sum {
    int sum = 0;
}

// Recursive function to add all greater values in
// every node
void modifyBSTUtil(Node node, Sum S)
{
    // Base Case
    if (node == null)
        return;

    // Recur for right subtree
    this.modifyBSTUtil(node.right, S);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    S.sum = S.sum + node.data;
    node.data = S.sum;

    // Recur for left subtree
    this.modifyBSTUtil(node.left, S);
}
```

```
    // A wrapper over modifyBSTUtil()
    void modifyBST(Node node)
    {
        Sum S = new Sum();
        this.modifyBSTUtil(node, S);
    }

    // Driver Function
    public static void main(String[] args)
    {
        BinarySearchTree tree = new BinarySearchTree();

          /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */

        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        tree.modifyBST(tree.root);

        // print inoder tarversal of the modified BST
        tree.inorder();
    }
}

// This code is contributed by Kamal Rawal
```

Output

```
350 330 300 260 210 150 80
```

Time Complexity: O(n) where n is number of nodes in the given BST.

As a side note, we can also use reverse Inorder traversal to find kth largest element in a BST.

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/add-greater-values-every-node-given-bst/

Chapter 4

Advantages of BST over Hash Table

Advantages of BST over Hash Table - GeeksforGeeks

Hash Table supports following operations in $\Theta(1)$ 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, $\Theta(1)$ is average time and some particular operations may be costly, especially when table resizing happens.

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/

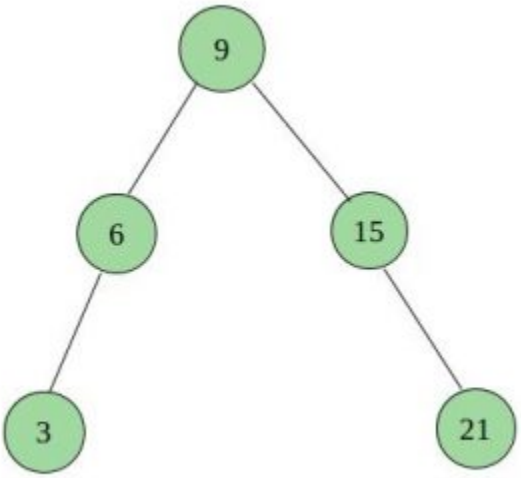
Chapter 5

BST to a Tree with sum of all smaller keys

BST to a Tree with sum of all smaller keys - GeeksforGeeks

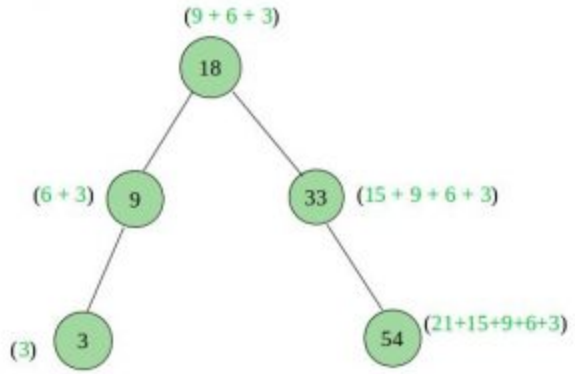
Given a Binary Search Tree(BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all smaller keys in BST.

Given a BST with N Nodes we have to convert into Binary Tree



Given above BST with **N=5** Nodes. The values at Node being **9, 6, 15, 3, 21**

Binary Tree after convertion



Binary Tree after convertion, the values at Node being **18, 9, 33, 3, 54**

Solution: We will perform a regular Inorder traversal in which we keep track of sum of Nodes visited. Let this sum be *sum*. The Node which is being visited, add that key of Node to *sum* i.e. *sum = sum + Node->key*. Change the key of current Node to *sum* i.e. *Node->key = sum*.
When a BST is being traversed in inorder, for every key currently being visited, all keys that are already visited are all smaller keys.

C++

```
 // Program to change a BST to Binary Tree such
// that key of a Node becomes original key plus
// sum of all smaller keys in BST
#include <stdio.h>
#include <stdlib.h>

/* A BST Node has key, left child and
   right child */
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

/* 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 = new Node;
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return (node);
```

```
}

// A recursive function that traverses the
// given BST in inorder and for every key,
// adds all smaller keys to it
void addSmallerUtil(struct Node* root, int* sum)
{
    // Base Case
    if (root == NULL)
        return;

    // Recur for left subtree first so that
    // sum of all smaller Nodes is stored
    addSmallerUtil(root->left, sum);

    // Update the value at sum
    *sum = *sum + root->key;

    // Update key of this Node
    root->key = *sum;

    // Recur for right subtree so that
    // the updated sum is added
    // to greater Nodes
    addSmallerUtil(root->right, sum);
}

// A wrapper over addSmallerUtil(). It
// initializes sum and calls addSmallerUtil()
// to recursively update and use value of
void addSmaller(struct Node* root)
{
    int sum = 0;
    addSmallerUtil(root, &sum);
}

// A utility function to print inorder
// traversal of Binary Tree
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->key);
    printInorder(node->right);
}

// Driver program to test above function
```

```
int main()
{
    /* Create following BST
            9
            / \
        6     15 */
    Node* root = newNode(9);
    root->left = newNode(6);
    root->right = newNode(15);

    printf(" Original BST\n");
    printInorder(root);

    addSmaller(root);

    printf("\n BST To Binary Tree\n");
    printInorder(root);

    return 0;
}
```

Java

```
 // Java program to convert BST to binary tree
// such that sum of all smaller keys is added
// to every key

class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class Sum {

    int addvalue = 0;
}

class BSTtoBinaryTree {

    static Node root;
    Sum add = new Sum();
```

```
// A recursive function that traverses
// the given BST in inorder and for every
// key, adds all smaller keys to it
void addSmallerUtil(Node node, Sum sum)
{

    // Base Case
    if (node == null) {
        return;
    }

    // Recur for left subtree first so that
    //  sum of all smaller Nodes is stored at sum
    addSmallerUtil(node.left, sum);

    // Update the value at sum
    sum.addvalue = sum.addvalue + node.data;

    // Update key of this Node
    node.data = sum.addvalue;

    // Recur for right subtree so that the
    // updated sum is added to greater Nodes
    addSmallerUtil(node.right, sum);
}

// A wrapper over addSmallerUtil().  It
// initializes addvalue and calls
// addSmallerUtil() to recursively update
// and use value of addvalue
Node addSmaller(Node node)
{
    addSmallerUtil(node, add);
    return node;
}

// A utility function to print inorder
// traversal of Binary Tree
void printInorder(Node node)
{
    if (node == null) {
        return;
    }
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}
```

```
    // Driver program to test the above functions
    public static void main(String[] args)
    {
        BSTtoBinaryTree tree = new BSTtoBinaryTree();
        tree.root = new Node(9);
        tree.root.left = new Node(6);
        tree.root.right = new Node(15);

        System.out.println("Original BST");
        tree.printInorder(root);
        Node Node = tree.addSmaller(root);
        System.out.println("");
        System.out.println("BST To Binary Tree");
        tree.printInorder(Node);
    }
}
```

Source

https://www.geeksforgeeks.org/bst-tree-sum-smaller-keys/

Chapter 6

Binary Search Tree insert with Parent Pointer

Binary Search Tree insert with Parent Pointer - GeeksforGeeks

We have discussed simple BST insert. How to insert in a tree where parent pointer needs to be maintained. Parent pointers are helpful to quickly find ancestors of a node, LCA of two nodes, successor of a node, etc.

In recursive calls of simple insertion, we return pointer of root of subtree created in a subtree. So the idea is to store this pointer for left and right subtrees. We set parent pointers of this returned pointers after the recursive calls. This makes sure that all parent pointers are set during insertion. Parent of root is set to NULL. We handle this by assigning parent as NULL by default to all newly allocated nodes.

```
 // C++ program to demonstrate insert operation
// in binary search tree with parent pointer
#include<stdio.h>
#include<stdlib.h>

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

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

```
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct Node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("Node : %d, ", root->key);
        if (root->parent == NULL)
          printf("Parent : NULL \n");
        else
          printf("Parent : %d \n", root->parent->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 *lchild = insert(node->left, key);
        node->left  = lchild;

        // Set parent of root of left subtree
        lchild->parent = node;
    }
    else if (key > node->key)
    {
        Node *rchild = insert(node->right, key);
        node->right  = rchild;

        // Set parent of root of right subtree
        rchild->parent = node;
    }

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

// 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);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // print iNoder traversal of the BST
    inorder(root);

    return 0;
}
```

Output :

```
Node : 20, Parent : 30
Node : 30, Parent : 50
Node : 40, Parent : 30
Node : 50, Parent : NULL
Node : 60, Parent : 70
Node : 70, Parent : 50
Node : 80, Parent : 70
```

Exercise:
How to maintain parent pointer during deletion.

Source

https://www.geeksforgeeks.org/binary-search-tree-insert-parent-pointer/

Chapter 7

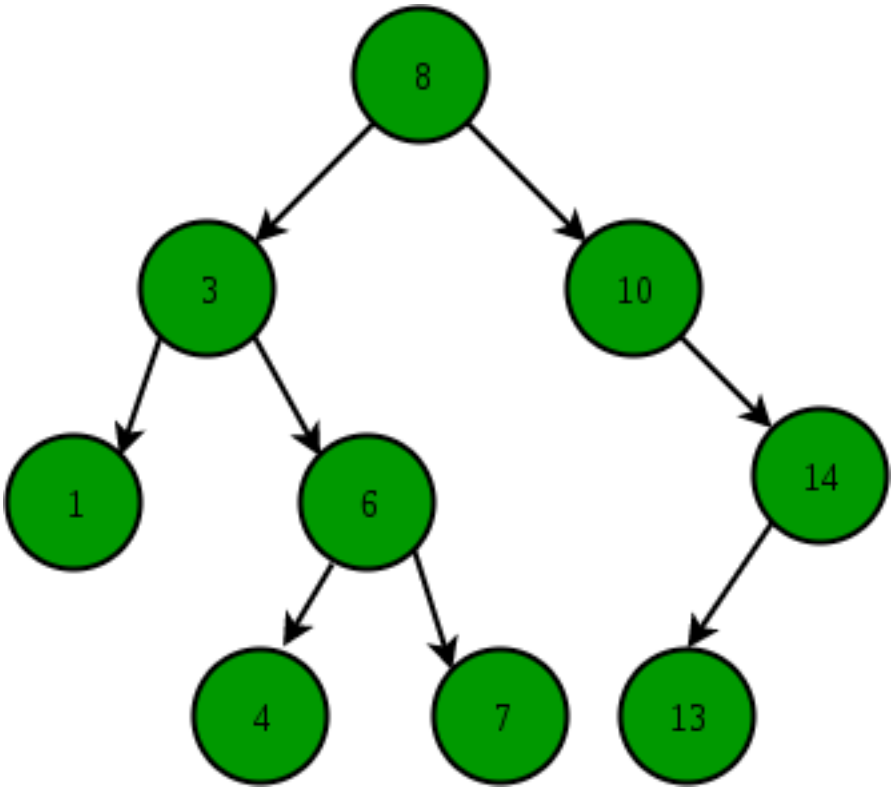
Binary Search Tree Set 1 (Search and Insertion)

Binary Search Tree Set 1 (Search and Insertion) - GeeksforGeeks

The following is definition of Binary Search Tree(BST) according to Wikipedia

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.
  There must be no duplicate nodes.



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.

Searching a 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.

C/C++

```
 // C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
```

```
}
```

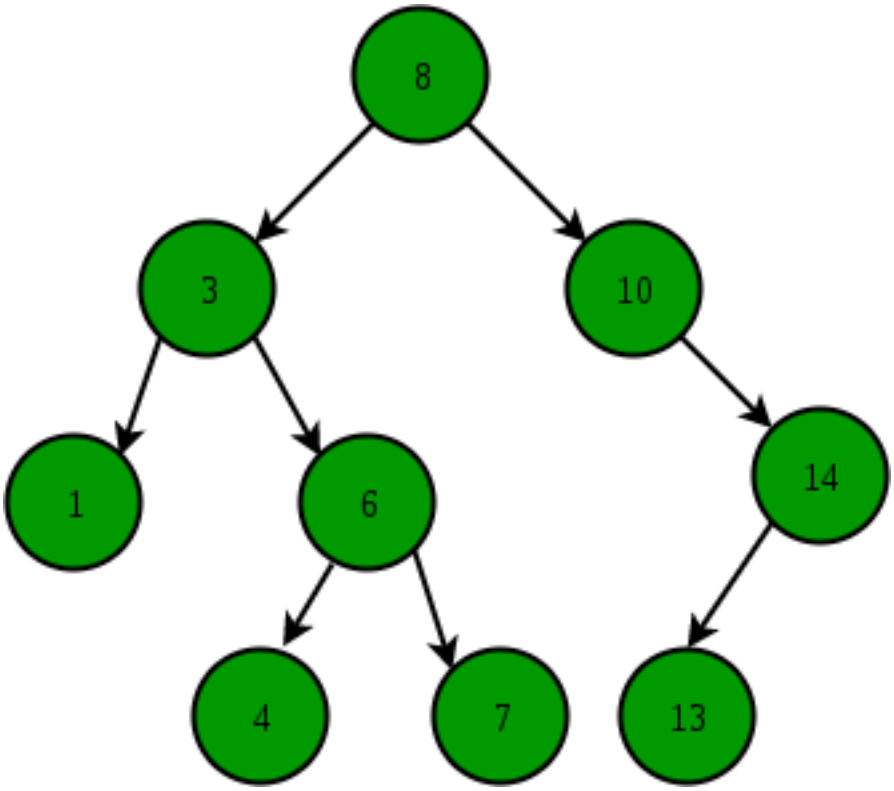
Python

```
 # A utility function to search a given key in BST
def search(root,key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    return search(root.left,key)

# This code is contributed by Bhavya Jain
```

Java

```
 // A utility function to search a given key in BST
public Node search(Node root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root==null || root.key==key)
        return root;

    // val is greater than root's key
    if (root.key > key)
        return search(root.left, key);

    // val is less than root's key
    return search(root.right, key);
}
```

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.

```
        100                                  100
       /   \        Insert 40               /    \
     20     500    --------->             20     500
    /  \                                 /  \
   10   30                              10   30
                                              \
                                              40
```

C/C++

```
 // C program to demonstrate insert 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 \n", 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 if (key > node->key)
        node->right = insert(node->right, key);

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

// 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);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // print inoder traversal of the BST
    inorder(root);

    return 0;
}
```

Python

```
 # Python program to demonstrate insert operation in binary search tree

# A utility class that represents an individual node in a BST
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key

# A utility function to insert a new node with the given key
def insert(root,node):
    if root is None:
        root = node
    else:
        if root.val < node.val:
            if root.right is None:
                root.right = node
            else:
                insert(root.right, node)
        else:
            if root.left is None:
                root.left = node
            else:
                insert(root.left, node)

# A utility function to do inorder tree traversal
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
```

```
        inorder(root.right)


# Driver program to test the above functions
# Let us create the following BST
#      50
#    /      \
#   30     70
#   / \    / \
#  20 40  60 80
r = Node(50)
insert(r,Node(30))
insert(r,Node(20))
insert(r,Node(40))
insert(r,Node(70))
insert(r,Node(60))
insert(r,Node(80))

# Print inoder traversal of the BST
inorder(r)

# This code is contributed by Bhavya Jain
```

Java

```
 // Java program to demonstrate insert operation in binary search tree
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // This method mainly calls insertRec()
```

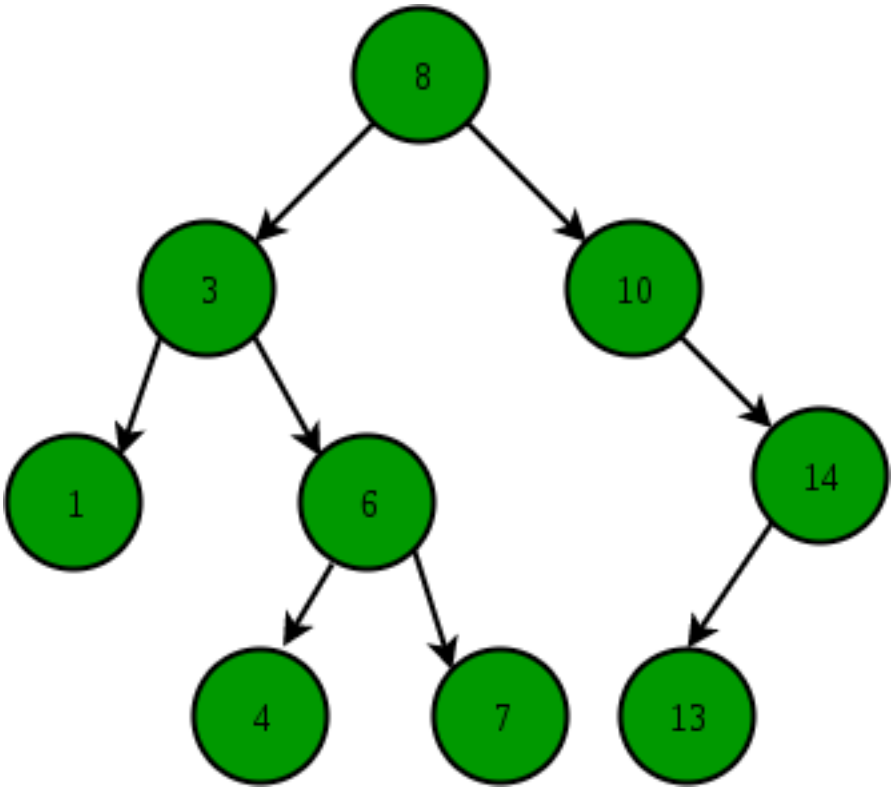
```
void insert(int key) {
   root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key) {

    /* If the tree is empty, return a new node */
    if (root == null) {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

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

// This method mainly calls InorderRec()
void inorder()  {
   inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.println(root.key);
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
          50
       /     \
      30      70
     /  \    /  \
   20   40  60   80 */
    tree.insert(50);
```

```
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        // print inorder traversal of the BST
        tree.inorder();
    }
}
// This code is contributed by Ankur Narain Verma
```

Output:

```
20
30
40
50
60
70
80
```

Illustration to insert 2 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. After reaching end,just insert that node at left(if less than current) else right.



Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Some Interesting Facts:

- Inorder traversal of BST always produces sorted output.
- We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.
- Number of unique BSTs with n distinct keys is Catalan Number

Related Links:

- Binary Search Tree Delete Operation
- Quiz on Binary Search Tree
- Coding practice on BST
- All Articles on BST

Source

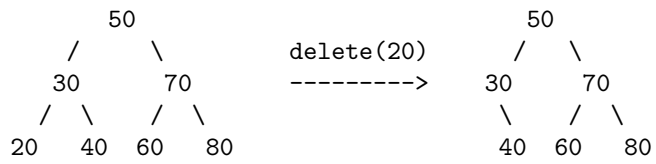
https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Chapter 8

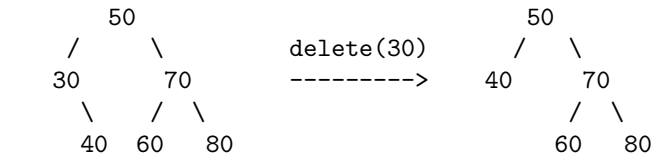
Binary Search Tree Set 2 (Delete)

Binary Search Tree Set 2 (Delete) - GeeksforGeeks

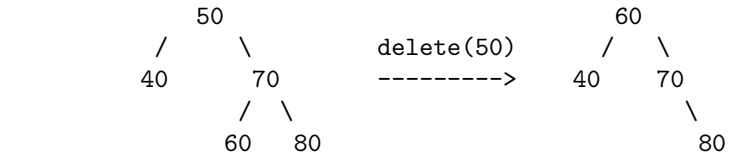
We have discussed BST search and insert operations. In this post, delete operation is discussed. When we delete a node, three possibilities arise.
1) ***Node to be deleted is leaf:*** Simply remove from the tree.



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/C++

```
 // 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->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;
}
```

Java

```
 // Java program to demonstrate delete operation in binary search tree
class BinarySearchTree
{
    /* Class containing left and right child of current node and key value*/
    class Node
    {
        int key;
        Node left, right;

        public Node(int item)
        {
            key = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // This method mainly calls deleteRec()
    void deleteKey(int key)
    {
        root = deleteRec(root, key);
    }
```

```
/* A recursive function to insert a new key in BST */
Node deleteRec(Node root, int key)
{
    /* Base Case: If the tree is empty */
    if (root == null)  return root;

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(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)
            return root.right;
        else if (root.right == null)
            return root.left;

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        root.key = minValue(root.right);

        // Delete the inorder successor
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

int minValue(Node root)
{
    int minv = root.key;
    while (root.left != null)
    {
        minv = root.left.key;
        root = root.left;
    }
    return minv;
}

// This method mainly calls insertRec()
void insert(int key)
{
    root = insertRec(root, key);
```

```
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key)
{

    /* If the tree is empty, return a new node */
    if (root == null)
    {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

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

// This method mainly calls InorderRec()
void inorder()
{
    inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root)
{
    if (root != null)
    {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
          50
       /     \
```

```
          30      70
         /  \    /  \
        20   40  60   80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        System.out.println("Inorder traversal of the given tree");
        tree.inorder();

        System.out.println("\nDelete 20");
        tree.deleteKey(20);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 30");
        tree.deleteKey(30);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 50");
        tree.deleteKey(50);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();
    }
}
```

Python

```
 # 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
```

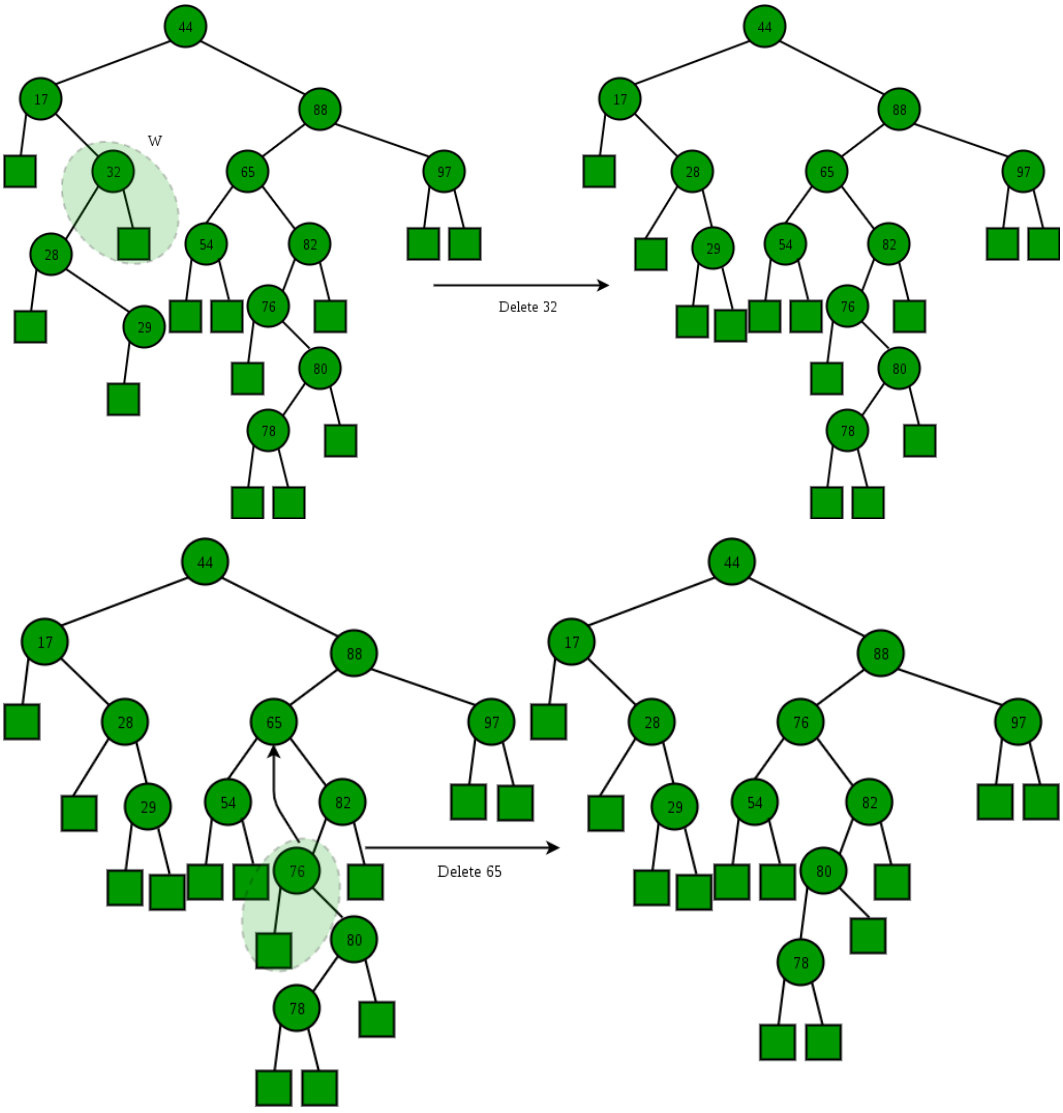
```
    # then it lies in right subtree
    elif(key > root.key):
        root.right = deleteNode(root.right, key)

    # 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)

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

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

Illustration:



Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

Related Links:

- Binary Search Tree Introduction, Search and Insert/a>
- Quiz on Binary Search Tree
- Coding practice on BST
- All Articles on BST

Improved By : Manoj Kumar 20

Source

https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

Chapter 9

Binary Tree to Binary Search Tree Conversion

Binary Tree to Binary Search Tree Conversion - GeeksforGeeks

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Examples.

```
Example 1
Input:
          10
         /  \
        2    7
       / \
      8   4
Output:
          8
         /  \
        4    10
       / \
      2   7


Example 2
Input:
          10
         /  \
        30   15
       /      \
      20       5
```

```
Output:
          15
         /  \
       10    20
       /      \
      5        30
```

Solution
Following is a 3 step solution for converting Binary tree to Binary Search Tree.
1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.
2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in O(nLogn) time using Heap Sort or Merge Sort.
3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

C

```
 /* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

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

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++;  // increase index for next entry
```

```
    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
     return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}

// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
   This functon basically does Inorder traversal of Binary Tree and
   one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
      return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if(root == NULL)
       return;

    /* Count the number of nodes in Binary Tree so that
       we know the size of temporary array to be created */
```

```
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in arr[]
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid meory leak
    delete [] arr;
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

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

    /* then print the data of node */
    printf("%d ", node->data);

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

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;
```

```
    /* Constructing tree given in the above figure
          10
         /  \
        30   15
       /      \
      20       5   */
    root = newNode(10);
    root->left = newNode(30);
    root->right = newNode(15);
    root->left->left = newNode(20);
    root->right->right = newNode(5);

    // convert Binary Tree to BST
    binaryTreeToBST (root);

    printf("Following is Inorder Traversal of the converted BST: \n");
    printInorder (root);

    return 0;
}
```

Python

```
 # Program to convert binary tree to BST

# A binary tree node
class Node:

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

# Helper function to store the inroder traversal of a tree
def storeInorder(root, inorder):

    # Base Case
    if root is None:
        return

    # First store the left subtree
    storeInorder(root.left, inorder)

    # Copy the root's data
    inorder.append(root.data)
```

```
    # Finally store the right subtree
    storeInorder(root.right, inorder)

# A helper funtion to count nodes in a binary tree
def countNodes(root):
    if root is None:
        return 0

    return countNodes(root.left) + countNodes(root.right) + 1

# Helper function that copies contents of sorted array
# to Binary tree
def arrayToBST(arr, root):

    # Base Case
    if root is None:
        return

    # First update the left subtree
    arrayToBST(arr, root.left)

    # now update root's data delete the value from array
    root.data = arr[0]
    arr.pop(0)

    # Finally update the right subtree
    arrayToBST(arr, root.right)

# This function converts a given binary tree to BST
def binaryTreeToBST(root):

    # Base Case: Tree is empty
    if root is None:
        return

    # Count the number of nodes in Binary Tree so that
    # we know the size of temporary array to be created
    n = countNodes(root)

    # Create the temp array and store the inorder traveral
    # of tree
    arr = []
    storeInorder(root, arr)

    # Sort the array
    arr.sort()

    # copy array elements back to binary tree
```

```
    arrayToBST(arr, root)

# Print the inorder traversal of the tree
def printInorder(root):
    if root is None:
        return
    printInorder(root.left)
    print root.data,
    printInorder(root.right)

# Driver program to test above function
root = Node(10)
root.left = Node(30)
root.right = Node(15)
root.left.left = Node(20)
root.right.right= Node(5)

# Convert binary tree to BST
binaryTreeToBST(root)

print "Following is the inorder traversal of the converted BST"
printInorder(root)

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

Output:

```
Following is Inorder Traversal of the converted BST:
5 10 15 20 30
```

We will be covering another method for this problem which converts the tree using O(height of tree) extra space.

Source

https://www.geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion/

Chapter 10

Binary Tree to Binary Search Tree Conversion using STL set

Binary Tree to Binary Search Tree Conversion using STL set - GeeksforGeeks

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

This solution will use Sets of C++ STL instead of array based solution.

Examples:

```
Example 1
Input:
          10
         /  \
        2    7
       / \
      8   4
Output:
          8
         /  \
        4    10
       / \
      2   7


Example 2
Input:
          10
         /  \
        30   15
       /      \
```

```
      20       5
Output:
          15
         /  \
        10    20
        /      \
       5        30
```

Solution

1. Copy the items of binary tree in a **set** while doing inorder traversal. This takes O(n log n) time. Note that set in C++ STL is implemented using a Self Balancing Binary Search Tree like Red Black Tree, AVL Tree, etc
2. There is no need to sort the set as **sets** in C++ are implemented using Self-balancing binary search trees due to which each operation such as insertion, searching, deletion etc takes O(log n) time.
3. Now simply copy the items of **set** one by one from beginning to the tree while doing inorder traversal of tree. Care should be taken as when copying each item of **set** from its beginning, we first copy it to the tree while doing inorder traversal, then remove it from the set as well.

Now the above solution is simpler and easier to implement than the array based conversion of Binary tree to Binary search tree explained here- Conversion of Binary Tree to Binary Search tree (Set-1), where we had to separately make a function to sort the items of the array after copying the items from tree to it.

C++ program to convert a binary tree to binary search tree using set.

```
 /* CPP program to convert a Binary tree to BST
   using sets as containers. */
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// function to store the nodes in set while
// doing inorder traversal.
void storeinorderInSet(Node* root, set<int>& s)
{
    if (!root)
        return;

    // visit the left subtree first
```

```
    storeinorderInSet(root->left, s);

    // insertion takes order of O(logn) for sets
    s.insert(root->data);

    // visit the right subtree
    storeinorderInSet(root->right, s);

} // Time complexity  = O(nlogn)

// function to copy items of set one by one
// to the tree while doing inorder traversal
void setToBST(set<int>& s, Node* root)
{
    // base condition
    if (!root)
        return;

    // first move to the left subtree and
    // update items
    setToBST(s, root->left);

    // iterator initially pointing to the
    // beginning of set
    auto it = s.begin();

    // copying the item at beginning of
    // set(sorted) to the tree.
    root->data = *it;

    // now erasing the beginning item from set.
    s.erase(it);

    // now move to right subtree and update items
    setToBST(s, root->right);

} // T(n) = O(nlogn) time

// Converts Binary tree to BST.
void binaryTreeToBST(Node* root)
{
    set<int> s;

    // populating the set with the tree's
    // inorder traversal data
    storeinorderInSet(root, s);

    // now sets are by default sorted as
```

```
    // they are implemented using self-
    // balancing BST

    // copying items from set to the tree
    // while inorder traversal which makes a BST
    setToBST(s, root);

} // Time complexity  =  O(nlogn),
  // Auxiliary Space = O(n) for set.

// helper function to create a node
Node* newNode(int data)
{
    // dynamically allocating memory
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to do inorder traversal
void inorder(Node* root)
{
    if (!root)
        return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main()
{
    Node* root = newNode(5);
    root->left = newNode(7);
    root->right = newNode(9);
    root->right->left = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->right->right = newNode(11);

    /* Constructing tree given in the above figure
           5
         /   \
        7     9
       /\    / \
      1  6   10 11   */

    // converting the above Binary tree to BST
```

```
    binaryTreeToBST(root);
    cout << "Inorder traversal of BST is: " << endl;
    inorder(root);
    return 0;
}
```

Output:

```
Inorder traversal of BST is:
1 5 6 7 9 10 11
```

Time Complexity : O(n Log n)
Auxiliary Space : (n)

Source

https://www.geeksforgeeks.org/binary-tree-binary-search-tree-conversion-using-stl-set/

Chapter 11

C Program for Red Black Tree Insertion

C Program for Red Black Tree Insertion - GeeksforGeeks

Following article is extension of article discussed here.

InAVL 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

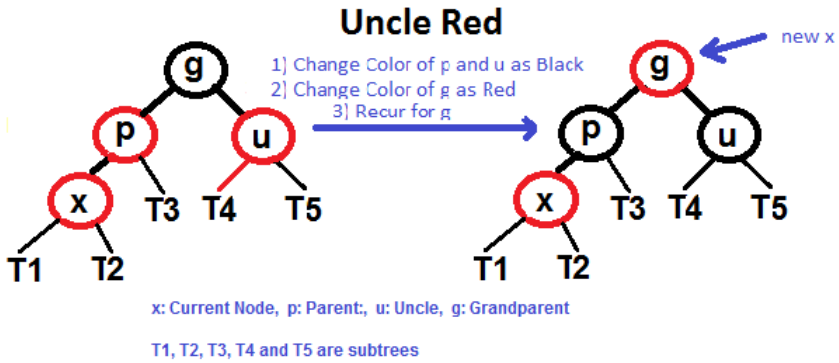
We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

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).

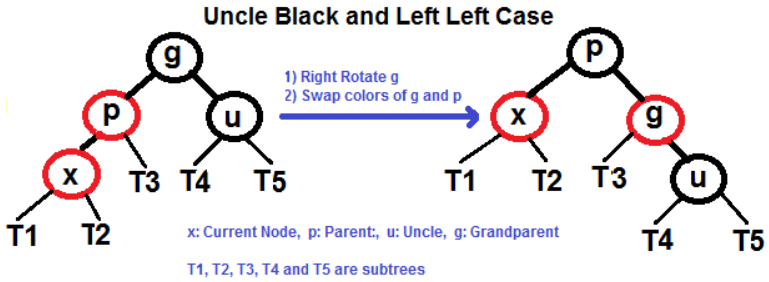
3) Do following if color of x's parent is not BLACK or x is not root.
....**a) If x's uncle is RED** (Grand parent must have been black from property 4)
.....**(i)** Change color of parent and uncle as BLACK.
.....**(ii)** color of grand parent as RED.
.....**(iii)** Change x = x's grandparent, repeat steps 2 and 3 for new x.



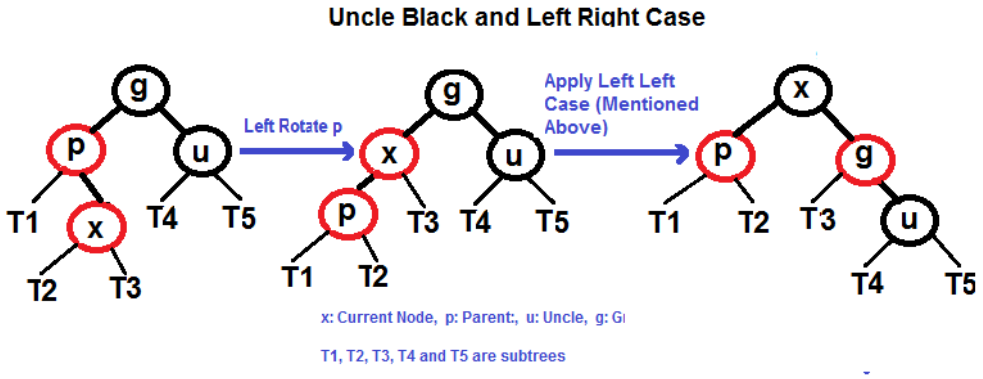
....**b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar toAVL Tree)
.....**i)** Left Left Case (p is left child of g and x is left child of p)
.....**ii)** Left Right Case (p is left child of g and x is right child of p)
.....**iii)** Right Right Case (Mirror of case a)
.....**iv)** Right Left Case (Mirror of case c)

Following are operations to be performed in four subcases when uncle is BLACK.

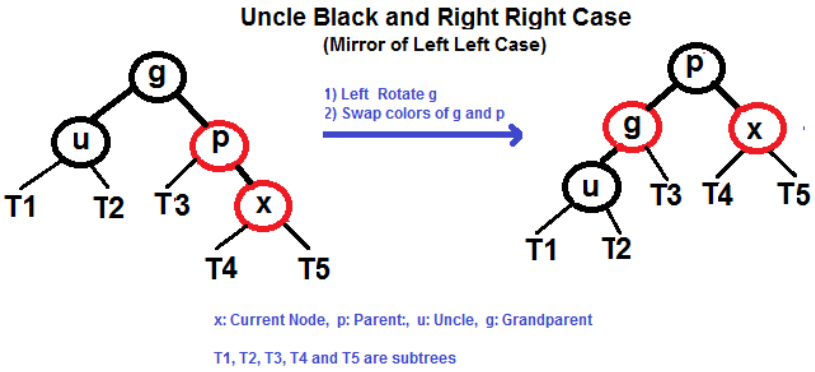
Left Left Case (See g, p and x)



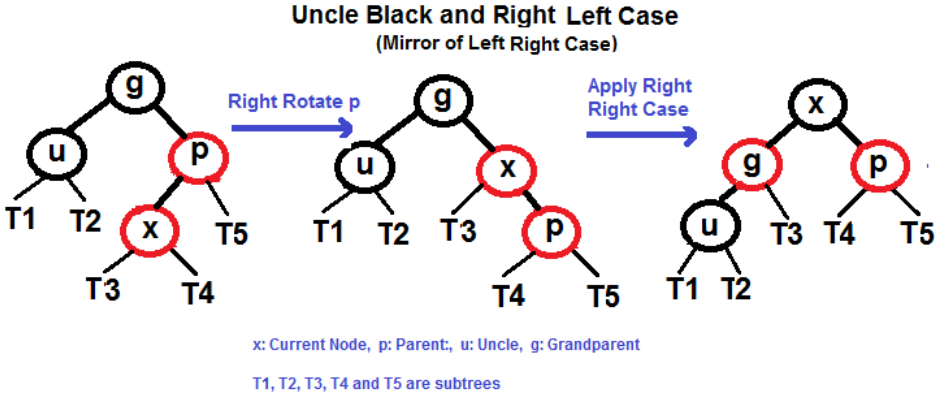
Left Right Case (See g, p and x)



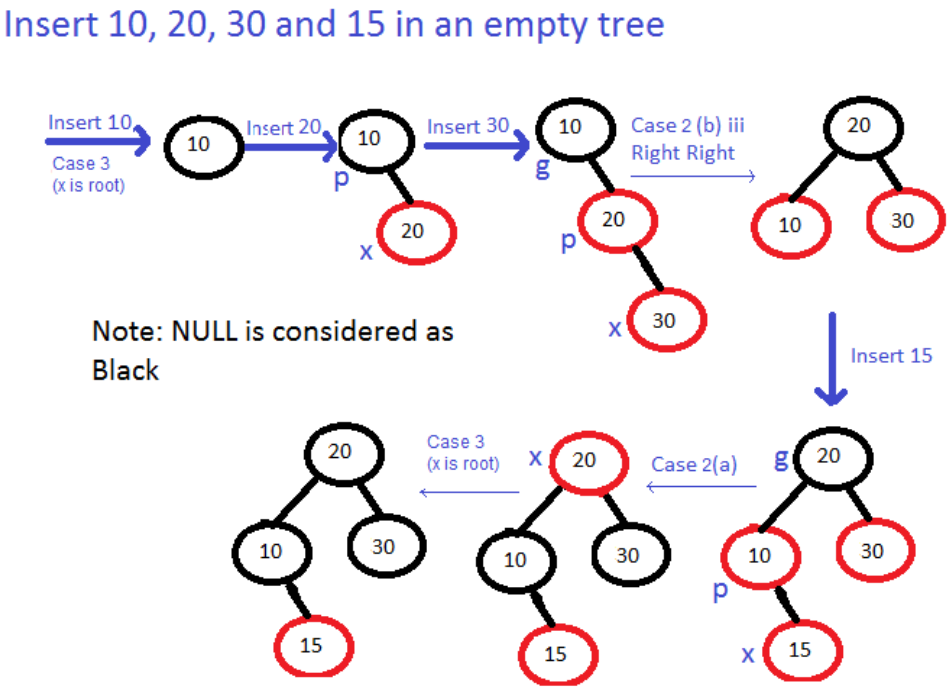
Right Right Case (See g, p and x)



Right Left Case (See g, p and x)



Examples of Insertion



Below is C++ Code.

```
 /** C++ implementation for Red-Black Tree Insertion
   This code is adopted from the code provided by
   Dinesh Khandelwal in comments **/
#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;
    }
};
```

```
// 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;
}
```

Output:

```
Inoder Traversal of Created Tree
1  2  3  4  5  6  7

Level Order Traversal of Created Tree
6  4  7  2  5  1  3
```

This article is contributed by **Mohsin Mohammaad**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/

Chapter 12

Check for Identical BSTs without building the trees

Check for Identical BSTs without building the trees - GeeksforGeeks

Given two arrays which represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.

Examples
For example, the input arrays are {2, 4, 3, 1} and {2, 1, 4, 3} will construct the same tree

```
 Let the input arrays be a[] and b[]

Example 1:
a[] = {2, 4, 1, 3} will construct following tree.
   2
 /  \
1    4
    /
   3
b[] = {2, 4, 3, 1} will also also construct the same tree.
   2
 /  \
1    4
    /
   3
So the output is "True"

Example 2:
a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13}
b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13}
```

```
They both construct the same following BST, so output is "True"
            8
          /    \
        3       10
      /  \        \
    1     6       14
        /   \     /
       4     7   13
```

Solution:
According to BST property, elements of the left subtree must be smaller and elements of right subtree must be greater than root.
Two arrays represent the same BST if, for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.
The idea is to check of if next smaller and greater elements are same in both arrays. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements. Following is an interesting recursive implementation of the idea.

```
 // A C program to check for Identical BSTs without building the trees
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>

/* The main function that checks if two arrays a[] and b[] of size n construct
  same BST. The two values 'min' and 'max' decide whether the call is made for
  left subtree or right subtree of a parent element. The indexes i1 and i2 are
  the indexes in (a[] and b[]) after which we search the left or right child.
  Initially, the call is made for INT_MIN and INT_MAX as 'min' and 'max'
  respectively, because root has no parent.
  i1 and i2 are just after the indexes of the parent element in a[] and b[]. */
bool isSameBSTUtil(int a[], int b[], int n, int i1, int i2, int min, int max)
{
   int j, k;

   /* Search for a value satisfying the constraints of min and max in a[] and
      b[]. If the parent element is a leaf node then there must be some
      elements in a[] and b[] satisfying constraint. */
   for (j=i1; j<n; j++)
       if (a[j]>min && a[j]<max)
           break;
   for (k=i2; k<n; k++)
       if (b[k]>min && b[k]<max)
           break;

   /* If the parent element is leaf in both arrays */
   if (j==n && k==n)
```

```
        return true;

    /* Return false if any of the following is true
       a) If the parent element is leaf in one array, but non-leaf in other.
       b) The elements satisfying constraints are not same. We either search
          for left child or right child of the parent element (decinded by min
          and max values). The child found must be same in both arrays */
    if (((j==n)^(k==n)) || a[j]!=b[k])
        return false;

    /* Make the current child as parent and recursively check for left and right
       subtrees of it. Note that we can also pass a[k] in place of a[j] as they
       are both are same */
    return isSameBSTUtil(a, b, n, j+1, k+1, a[j], max) &&  // Right Subtree
           isSameBSTUtil(a, b, n, j+1, k+1, min, a[j]);    // Left Subtree
}

// A wrapper over isSameBSTUtil()
bool isSameBST(int a[], int b[], int n)
{
    return isSameBSTUtil(a, b, n, 0, 0, INT_MIN, INT_MAX);
}

// Driver program to test above functions
int main()
{
    int a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13};
    int b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13};
    int n=sizeof(a)/sizeof(a[0]);
    printf("%s\n", isSameBST(a, b, n)?
              "BSTs are same":"BSTs not same");
    return 0;
}
```

Output:

```
BSTs are same
```

This article is compiled by**Amit Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : codeShaurya

Source

https://www.geeksforgeeks.org/check-for-identical-bsts-without-building-the-trees/

Chapter 13

Check given array of size n can represent BST of n levels or not

Check given array of size n can represent BST of n levels or not - GeeksforGeeks

Given an array of size n, the task is to find whether array can represent a BST with n levels.

Since levels are n, we construct a tree in the following manner.
Assuming a number X,

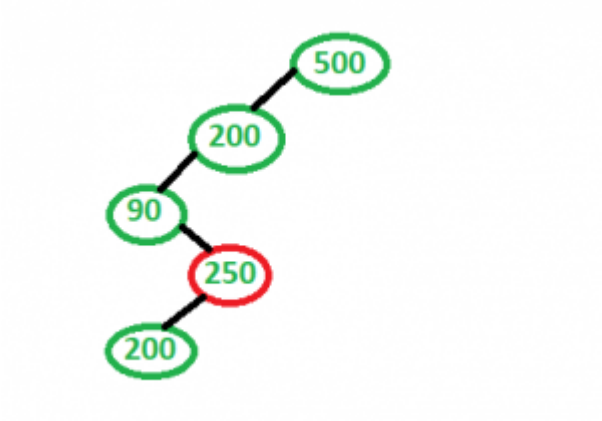
- Number higher than X is on the right side
- Number lower than X is on the left side.

Note: during the insertion, we never go beyond a number already visited.

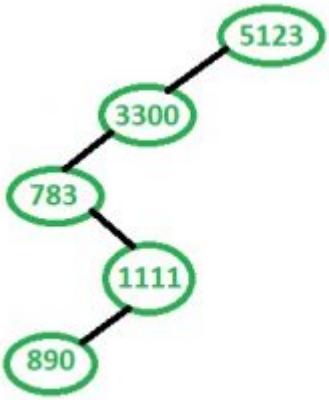
Examples:

```
Input : 500, 200, 90, 250, 100
Output : No

Input : 5123, 3300, 783, 1111, 890
Output : Yes
```

Explanation :



For the sequence **500, 200, 90, 250, 100** formed tree(in above image) can't represent BST.



The sequence **5123, 3300, 783, 1111, 890** forms a binary search tree hence its a correct sequence.

Method 1 : By constructing BST
We first insert all array values level by level in a Tree. To insert, we check if current value is less than previous value or greater. After constructing the tree, we check if the constructed tree is Binary Search Tree or not.

```
 // C++ program to Check given array
// can represent BST or not
#include <bits/stdc++.h>
using namespace std;

// structure for Binary Node
struct Node {
    int key;
    struct Node *right, *left;
};
```

```
Node* newNode(int num)
{
    Node* temp = new Node;
    temp->key = num;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// To create a Tree with n levels. We always
// insert new node to left if it is less than
// previous value.
Node* createNLevelTree(int arr[], int n)
{
    Node* root = newNode(arr[0]);
    Node* temp = root;
    for (int i = 1; i < n; i++) {
        if (temp->key > arr[i]) {
            temp->left = newNode(arr[i]);
            temp = temp->left;
        }
        else {
            temp->right = newNode(arr[i]);
            temp = temp->right;
        }
    }
    return root;
}

// Please refer below post for details of this
// function.
// https:// www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/
bool isBST(Node* root, int min, int max)
{
    if (root == NULL)
        return true;

    if (root->key < min || root->key > max)
        return false;

    // Allow only distinct values
    return (isBST(root->left, min,
                  (root->key) - 1)
            && isBST(root->right,
                     (root->key) + 1, max));
}
```

```
// Returns tree if given array of size n can
// represent a BST of n levels.
bool canRepresentNLevelBST(int arr[], int n)
{
    Node* root = createNLevelTree(arr, n);
    return isBST(root, INT_MIN, INT_MAX);
}

// Driver code
int main()
{
    int arr[] = { 512, 330, 78, 11, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);

    if (canRepresentNLevelBST(arr, n))
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Output:

```
Yes
```

Method 2 (Array Based)
1. Take two variables max = INT_MAX to mark the maximum limit for left subtree and min = INT_MIN to mark the minimum limit for right subtree.
2. Loop from arr[1] to arr[n-1]
3. for each element check
a. If (arr[i] > arr[i-1] && arr[i] > min && arr[i] < max), update min = arr[i-1]
b. Else if (arr[i] min && arr[i] < max), update max = arr[i]
c. If none of the above two conditions hold, then element will not be inserted in a new level, so break.

```
 // C++ program to Check given array
// can represent BST or not
#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    int arr[] = { 5123, 3300, 783, 1111, 890 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int max = INT_MAX;
    int min = INT_MIN;
    bool flag = true;

    for (int i = 1; i < n; i++) {

        // This element can be inserted to the right
        // of the previous element, only if it is greater
        // than the previous element and in the range.
        if (arr[i] > arr[i - 1] && arr[i] > min && arr[i] < max) {
            // max remains same, update min
            min = arr[i - 1];
        }
        // This element can be inserted to the left
        // of the previous element, only if it is lesser
        // than the previous element and in the range.
        else if (arr[i] < arr[i - 1] && arr[i] > min && arr[i] < max) {
            // min remains same, update max
            max = arr[i - 1];
        }
        else {
            flag = false;
            break;
        }
    }

    if (flag) {
        cout << "Yes";
    }
    else {
        // if the loop completed successfully without encountering else condition
        cout << "No";
    }

    return 0;
}
```

Output:

```
Yes
```

Improved By : Sakshi Parashar

Source

https://www.geeksforgeeks.org/check-given-array-of-size-n-can-represent-bst-of-n-levels-or-not/

Chapter 14

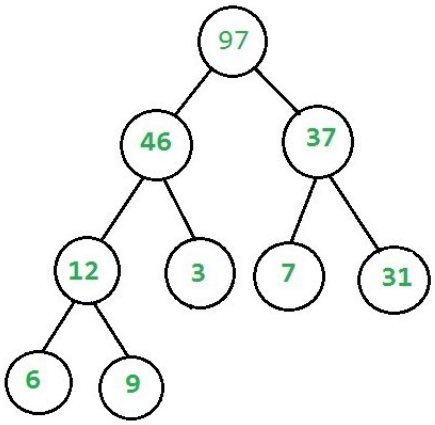
Check if a given Binary Tree is Heap

Check if a given Binary Tree is Heap - GeeksforGeeks

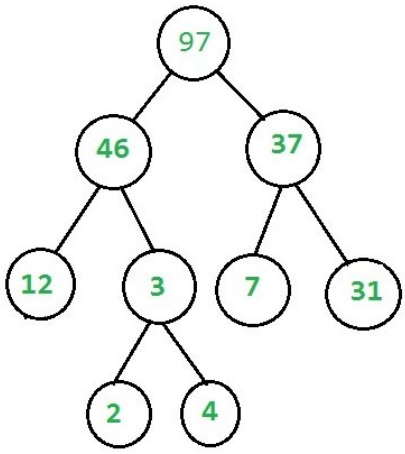
Given a binary tree we need to check it has heap property or not, Binary tree need to fulfill following two conditions for being a heap –

1. It should be a complete tree (i.e. all levels except last should be full).
2. Every node's value should be greater than or equal to its child node (considering max-heap).

For example this tree contains heap property –



While this doesn't –



We check each of the above condition separately, for checking completeness isComplete and for checking heap isHeapUtil function are written.
Detail about isComplete function can be found here.

isHeapUtil function is written considering following things –

1. Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).
2. If Node has No child then it's a leaf node and return true (Base case)
3. If Node has one child (it must be left child because it is a complete tree) then we need to compare this node with its single child only.
4. If Node has both child then check heap property at Node at recur for both subtrees. Complete code.

Implementation

C/C++

```
 /* C program to checks if a binary tree is max heap ot not */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/*  Tree node structure */
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
};
```

```
/* Helper function that allocates a new node */
struct Node *newNode(int k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function counts the number of nodes in a binary tree */
unsigned int countNodes(struct Node* root)
{
    if (root == NULL)
        return (0);
    return (1 + countNodes(root->left) + countNodes(root->right));
}

/* This function checks if the binary tree is complete or not */
bool isCompleteUtil (struct Node* root, unsigned int index,
                     unsigned int number_nodes)
{
    // An empty tree is complete
    if (root == NULL)
        return (true);

    // If index assigned to current node is more than
    // number of nodes in tree, then tree is not complete
    if (index >= number_nodes)
        return (false);

    // Recur for left and right subtrees
    return (isCompleteUtil(root->left, 2*index + 1, number_nodes) &&
            isCompleteUtil(root->right, 2*index + 2, number_nodes));
}

// This Function checks the heap property in the tree.
bool isHeapUtil(struct Node* root)
{
    //  Base case : single node satisfies property
    if (root->left == NULL && root->right == NULL)
        return (true);

    //  node will be in second last level
    if (root->right == NULL)
    {
        //  check heap property at Node
        //  No recursive call , because no need to check last level
```

```
        return (root->key >= root->left->key);
    }
    else
    {
        //  Check heap property at Node and
        //  Recursive check heap property at left and right subtree
        if (root->key >= root->left->key &&
            root->key >= root->right->key)
            return ((isHeapUtil(root->left)) &&
                    (isHeapUtil(root->right)));
        else
            return (false);
    }
}

//  Function to check binary tree is a Heap or Not.
bool isHeap(struct Node* root)
{
    // These two are used in isCompleteUtil()
    unsigned int node_count = countNodes(root);
    unsigned int index = 0;

    if (isCompleteUtil(root, index, node_count) && isHeapUtil(root))
        return true;
    return false;
}

// Driver program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(9);
    root->right = newNode(8);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    root->left->left->left = newNode(3);
    root->left->left->right = newNode(2);
    root->left->right->left = newNode(1);

    if (isHeap(root))
        printf("Given binary tree is a Heap\n");
    else
        printf("Given binary tree is not a Heap\n");

    return 0;
```

```
}
```

Java

```
 /* Java program to checks if a binary tree is max heap ot not */

// A Binary Tree node
class Node
{
    int key;
    Node left, right;

    Node(int k)
    {
        key = k;
        left = right = null;
    }
}

class Is_BinaryTree_MaxHeap
{
    /* This function counts the number of nodes in a binary tree */
    int countNodes(Node root)
    {
        if(root==null)
            return 0;
        return(1 + countNodes(root.left) + countNodes(root.right));
    }

    /* This function checks if the binary tree is complete or not */
    boolean isCompleteUtil(Node root, int index, int number_nodes)
    {
        // An empty tree is complete
        if(root == null)
            return true;

        // If index assigned to current node is more than
        // number of nodes in tree, then tree is not complete
        if(index >= number_nodes)
            return false;

        // Recur for left and right subtrees
        return isCompleteUtil(root.left, 2*index+1, number_nodes) &&
               isCompleteUtil(root.right, 2*index+2, number_nodes);

    }

    // This Function checks the heap property in the tree.
```

```
boolean isHeapUtil(Node root)
{
    //  Base case : single node satisfies property
    if(root.left == null && root.right==null)
        return true;

    //  node will be in second last level
    if(root.right == null)
    {
        //  check heap property at Node
        //  No recursive call , because no need to check last level
        return root.key >= root.left.key;
    }
    else
    {
        //  Check heap property at Node and
        //  Recursive check heap property at left and right subtree
        if(root.key >= root.left.key && root.key >= root.right.key)
            return isHeapUtil(root.left) && isHeapUtil(root.right);
        else
            return false;
    }
}

//  Function to check binary tree is a Heap or Not.
boolean isHeap(Node root)
{
    if(root == null)
        return true;

    // These two are used in isCompleteUtil()
    int node_count = countNodes(root);

    if(isCompleteUtil(root, 0 , node_count)==true && isHeapUtil(root)==true)
        return true;
    return false;
}

// driver function to test the above functions
public static void main(String args[])
{
    Is_BinaryTree_MaxHeap bt = new Is_BinaryTree_MaxHeap();

    Node root = new Node(10);
    root.left = new Node(9);
    root.right = new Node(8);
    root.left.left = new Node(7);
    root.left.right = new Node(6);
```

```
        root.right.left = new Node(5);
        root.right.right = new Node(4);
        root.left.left.left = new Node(3);
        root.left.left.right = new Node(2);
        root.left.right.left = new Node(1);

        if(bt.isHeap(root) == true)
            System.out.println("Given binary tree is a Heap");
        else
            System.out.println("Given binary tree is not a Heap");
    }
}

// This code has been contributed by Amit Khandelwal
```

Python

```
 # To check if a binary tree
# is a MAX Heap or not
class GFG:
    def __init__(self, value):
        self.key = value
        self.left = None
        self.right = None

    def count_nodes(self, root):
        if root is None:
            return 0
        else:
            return (1 + self.count_nodes(root.left) +
                        self.count_nodes(root.right))

    def heap_propert_util(self, root):

        if (root.left is None and
            root.right is None):
            return True

        if root.right is None:
            return root.key >= root.left.key
        else:
            if (root.key >= root.left.key and
                root.key >= root.right.key):
                return (self.heap_propert_util(root.left) and
                        self.heap_propert_util(root.right))
            else:
                return False
```

```
    def complete_tree_util(self, root,
                           index, node_count):
        if root is None:
            return True
        if index >= node_count:
            return False
        return (self.complete_tree_util(root.left, 2 *
                                         index + 1, node_count) and
                self.complete_tree_util(root.right, 2 *
                                         index + 2, node_count))

    def check_if_heap(self):
        node_count = self.count_nodes(self)
        if (self.complete_tree_util(self, 0, node_count) and
            self.heap_propert_util(self)):
            return True
        else:
            return False

# Driver Code
root = GFG(5)
root.left = GFG(2)
root.right = GFG(3)
root.left.left = GFG(1)

if root.check_if_heap():
    print("Given binary tree is a heap")
else:
    print("Given binary tree is not a Heap")

# This code has been
# contributed by Yash Agrawal
```

Output:

```
Given binary tree is a Heap
```

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : scorncer17

Source

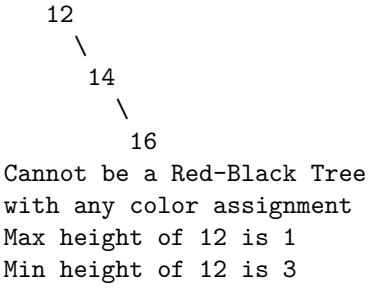
https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/

Chapter 15

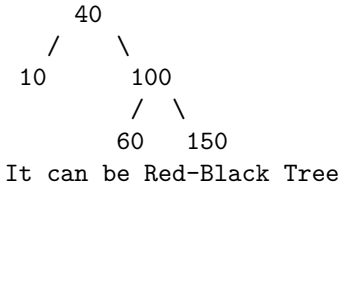
Check if a given Binary Tree is height balanced like a Red-Black Tree

Check if a given Binary Tree is height balanced like a Red-Black Tree - GeeksforGeeks

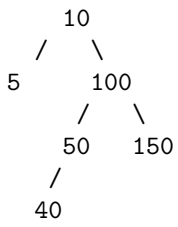
In a Red-Black Tree, the maximum height of a node is at most twice the minimum height (The four Red-Black tree properties make sure this is always followed). Given a Binary Search Tree, we need to check for following property.
For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.



Cannot be a Red-Black Tree
with any color assignment
Max height of 12 is 1
Min height of 12 is 3



It can be Red-Black Tree



```
 It can also be Red-Black Tree
```

Expected time complexity is O(n). The tree should be traversed at-most once in the solution.

We strongly recommend to minimize the browser and try this yourself first.
For every node, we need to get the maximum and minimum heights and compare them. The idea is to traverse the tree and for every node check if it's balanced. We need to write a recursive function that returns three things, a boolean value to indicate the tree is balanced or not, minimum height and maximum height. To return multiple values, we can either use a structure or pass variables by reference. We have passed maxh and minh by reference so that the values can be used in parent calls.

```
 /* Program to check if a given Binary Tree is balanced like a Red-Black Tree */
#include <iostream>
using namespace std;

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

/* utility that allocates a new Node with the given key  */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// Returns returns tree if the Binary tree is balanced like a Red-Black
// tree. This function also sets value in maxh and minh (passed by
// reference). maxh and minh are set as maximum and minimum heights of root.
bool isBalancedUtil(Node *root, int &maxh, int &minh)
{
    // Base case
    if (root == NULL)
    {
        maxh = minh = 0;
        return true;
    }

    int lmxh, lmnh; // To store max and min heights of left subtree
    int rmxh, rmnh; // To store max and min heights of right subtree

    // Check if left subtree is balanced, also set lmxh and lmnh
    if (isBalancedUtil(root->left, lmxh, lmnh) == false)
```

```
        return false;

    // Check if right subtree is balanced, also set rmxh and rmnh
    if (isBalancedUtil(root->right, rmxh, rmnh) == false)
        return false;

    // Set the max and min heights of this node for the parent call
    maxh = max(lmxh, rmxh) + 1;
    minh = min(lmnh, rmnh) + 1;

    // See if this node is balanced
    if (maxh <= 2*minh)
        return true;

    return false;
}

// A wrapper over isBalancedUtil()
bool isBalanced(Node *root)
{
    int maxh, minh;
    return isBalancedUtil(root, maxh, minh);
}

/* Driver program to test above functions*/
int main()
{
    Node * root = newNode(10);
    root->left = newNode(5);
    root->right = newNode(100);
    root->right->left = newNode(50);
    root->right->right = newNode(150);
    root->right->left->left = newNode(40);
    isBalanced(root)? cout << "Balanced" : cout << "Not Balanced";

    return 0;
}
```

Output:

```
Balanced
```

Time Complexity: Time Complexity of above code is O(n) as the code does a simple tree traversal.

Source

https://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/

Chapter 16

Check if a given array can represent Preorder Traversal of Binary Search Tree

Check if a given array can represent Preorder Traversal of Binary Search Tree - GeeksforGeeks

Given an array of numbers, return true if given array can represent preorder traversal of a Binary Search Tree, else return false. Expected time complexity is O(n).

Examples:

```
Input:  pre[] = {2, 4, 3}
Output: true
Given array can represent preorder traversal
of below tree
    2

      4
     /
    3

Input:  pre[] = {2, 4, 1}
Output: false
Given array cannot represent preorder traversal
of a Binary Search Tree.

Input:  pre[] = {40, 30, 35, 80, 100}
Output: true
Given array can represent preorder traversal
of below tree
```

```
     40
   /
 30    80

  35     100
```

```
Input:  pre[] = {40, 30, 35, 20, 80, 100}
Output: false
Given array cannot represent preorder traversal
of a Binary Search Tree.
```

A **Simple Solution** is to do following for every node pre[i] starting from first one.

```
1) Find the first greater value on right side of current node.
   Let the index of this node be j. Return true if following
   conditions hold. Else return false
    (i)  All values after the above found greater value are
         greater than current node.
    (ii) Recursive calls for the subarrays pre[i+1..j-1] and
         pre[j+1..n-1] also return true.
```

Time Complexity of the above solution is $O(n^2)$

An **Efficient Solution** can solve this problem in O(n) time. The idea is to use a stack. This problem is similar to Next (or closest) Greater Element problem. Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

```
1) Create an empty stack.
2) Initialize root as INT_MIN.
3) Do following for every element pre[i]
     a) If pre[i] is smaller than current root, return false.
     b) Keep removing elements from stack while pre[i] is greater
        then stack top. Make the last removed item as new root (to
        be compared next).
        At this point, pre[i] is greater than the removed root
        (That is why if we see a smaller element in step a), we
        return false)
     c) push pre[i] to stack (All elements in stack are in decreasing
        order)
```

Below is implementation of above idea.

C++

```
 // C++ program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
#include<bits/stdc++.h>
using namespace std;

bool canRepresentBST(int pre[], int n)
{
    // Create an empty stack
    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

    // Traverse given array
    for (int i=0; i<n; i++)
    {
        // If we find a node who is on right side
        // and smaller than root, return false
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,
        // Keep removing items smaller than pre[i]
        // and make the last removed item as new
        // root.
        while (!s.empty() && s.top()<pre[i])
        {
            root = s.top();
            s.pop();
        }

        // At this point either stack is empty or
        // pre[i] is smaller than root, push pre[i]
        s.push(pre[i]);
    }
    return true;
}

// Driver program
int main()
{
    int pre1[] = {40, 30, 35, 80, 100};
    int n = sizeof(pre1)/sizeof(pre1[0]);
    canRepresentBST(pre1, n)? cout << "truen":
                              cout << "falsen";
```

```
    int pre2[] = {40, 30, 35, 20, 80, 100};
    n = sizeof(pre2)/sizeof(pre2[0]);
    canRepresentBST(pre2, n)? cout << "truen":
                              cout << "falsen";

    return 0;
}
```

Java

```
 // Java program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
import java.util.Stack;

class BinarySearchTree {

    boolean canRepresentBST(int pre[], int n) {
        // Create an empty stack
        Stack<Integer> s = new Stack<Integer>();

        // Initialize current root as minimum possible
        // value
        int root = Integer.MIN_VALUE;

        // Traverse given array
        for (int i = 0; i < n; i++) {
            // If we find a node who is on right side
            // and smaller than root, return false
            if (pre[i] < root) {
                return false;
            }

            // If pre[i] is in right subtree of stack top,
            // Keep removing items smaller than pre[i]
            // and make the last removed item as new
            // root.
            while (!s.empty() && s.peek() < pre[i]) {
                root = s.peek();
                s.pop();
            }

            // At this point either stack is empty or
            // pre[i] is smaller than root, push pre[i]
            s.push(pre[i]);
        }
        return true;
    }
```

```
    public static void main(String args[]) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] pre1 = new int[]{40, 30, 35, 80, 100};
        int n = pre1.length;
        if (bst.canRepresentBST(pre1, n) == true) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
        int[] pre2 = new int[]{40, 30, 35, 20, 80, 100};
        int n1 = pre2.length;
        if (bst.canRepresentBST(pre2, n) == true) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
}

//This code is contributed by Mayank Jaiswal
```

Python

```
 # Python program for an efficient solution to check if
# a given array can represent Preorder traversal of
# a Binary Search Tree

INT_MIN = -2**32

def canRepresentBST(pre):

    # Create an empty stack
    s = []

    # Initialize current root as minimum possible value
    root = INT_MIN

    # Traverse given array
    for value in pre:
        #NOTE:value is equal to pre[i] according to the
        #given algo

        # If we find a node who is on the right side
        # and smaller than root, return False
        if value < root :
            return False
```

```
        # If value(pre[i]) is in right subtree of stack top,
        # Keep removing items smaller than value
        # and make the last removed items as new root
        while(len(s) > 0 and s[-1] < value) :
            root = s.pop()

        # At this point either stack is empty or value
        # is smaller than root, push value
        s.append(value)

    return True

# Driver Program
pre1 = [40 , 30 , 35 , 80 , 100]
print "true" if canRepresentBST(pre1) == True else "false"
pre2 = [40 , 30 , 35 , 20 ,  80 , 100]
print "true" if canRepresentBST(pre2) == True else "false"

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

Output:

```
true
false
```

This article is contributed by **Romil Punetha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree/

Chapter 17

Check if an array represents Inorder of Binary Search tree or not

Check if an array represents Inorder of Binary Search tree or not - GeeksforGeeks

Given an array of **N** element. The task is to check if it is Inorder traversal of any Binary Search Tree or not. Print "Yes" if it is Inorder traversal of any Binary Search Tree else print "No".

Examples:

```
Input : arr[] = { 19, 23, 25, 30, 45 }
Output : Yes

Input : arr[] = { 19, 23, 30, 25, 45 }
Output : No
```

The idea is to use the fact that the inorder traversal of Binary Search Tree is sorted. So, just check if given array is sorted or not.

```
 // C++ program to check if a given array is sorted
// or not.
#include<bits/stdc++.h>
using namespace std;

// Function that returns true if array is Inorder
// traversal of any Binary Search Tree or not.
bool isInorder(int arr[], int n)
{
```

```
    // Array has one or no element
    if (n == 0 || n == 1)
        return true;

    for (int i = 1; i < n; i++)

        // Unsorted pair found
        if (arr[i-1] > arr[i])
            return false;

    // No unsorted pair found
    return true;
}

// Driver code
int main()
{
    int arr[] = { 19, 23, 25, 30, 45 };
    int n = sizeof(arr)/sizeof(arr[0]);

    if (isInorder(arr, n))
        cout << "Yesn";
    else
        cout << "Non";

  return 0;
}
```

Output:

```
Yes
```

Source

https://www.geeksforgeeks.org/check-array-represents-inorder-binary-search-tree-not/

Chapter 18

Check if each internal node of a BST has exactly one child

Check if each internal node of a BST has exactly one child - GeeksforGeeks

Given Preorder traversal of a BST, check if each non-leaf node has only one child. Assume that the BST contains unique entries.

Examples

```
Input: pre[] = {20, 10, 11, 13, 12}
Output: Yes
The give array represents following BST. In the following BST, every internal
node has exactly 1 child. Therefor, the output is true.
        20
       /
      10
       \
        11
          \
           13
           /
          12
```

In Preorder traversal, descendants (or Preorder successors) of every node appear after the node. In the above example, 20 is the first node in preorder and all descendants of 20 appear after it. All descendants of 20 are smaller than it. For 10, all descendants are greater than it. In general, we can say, if all internal nodes have only one child in a BST, then all the descendants of every node are either smaller or larger than the node. The reason is simple, since the tree is BST and every node has only one child, all descendants of a node will either be on left side or right side, means all descendants will either be smaller or greater.

Approach 1 (Naive)
This approach simply follows the above idea that all values on right side are either smaller or larger. Use two loops, the outer loop picks an element one by one, starting from the leftmost element. The inner loop checks if all elements on the right side of the picked element are either smaller or greater. The time complexity of this method will be O(n^2).

Approach 2
Since all the descendants of a node must either be larger or smaller than the node. We can do following for every node in a loop.
1. Find the next preorder successor (or descendant) of the node.
2. Find the last preorder successor (last element in pre[]) of the node.
3. If both successors are less than the current node, or both successors are greater than the current node, then continue. Else, return false.

C

```
 #include <stdio.h>

bool hasOnlyOneChild(int pre[], int size)
{
    int nextDiff, lastDiff;

    for (int i=0; i<size-1; i++)
    {
        nextDiff = pre[i] - pre[i+1];
        lastDiff = pre[i] - pre[size-1];
        if (nextDiff*lastDiff < 0)
            return false;;
    }
    return true;
}

// driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre, size) == true )
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Java

```
 // Check if each internal node of BST has only one child
```

```
class BinaryTree {

    boolean hasOnlyOneChild(int pre[], int size) {
        int nextDiff, lastDiff;

        for (int i = 0; i < size - 1; i++) {
            nextDiff = pre[i] - pre[i + 1];
            lastDiff = pre[i] - pre[size - 1];
            if (nextDiff * lastDiff < 0) {
                return false;
            };
        }
        return true;
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{8, 3, 5, 7, 6};
        int size = pre.length;
        if (tree.hasOnlyOneChild(pre, size) == true) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python3

```
 # Check if each internal
# node of BST has only one child

def hasOnlyOneChild (pre, size):
    nextDiff=0; lastDiff=0

    for i in range(size-1):
        nextDiff = pre[i] - pre[i+1]
        lastDiff = pre[i] - pre[size-1]
        if nextDiff*lastDiff < 0:
            return False
    return True

# driver program to
# test above function
if __name__ == "__main__":
```

```
    pre = [8, 3, 5, 7, 6]
    size= len(pre)

    if (hasOnlyOneChild(pre,size) == True):
        print("Yes")
    else:
        print("No")

# This code is contributed by
# Harshit Saini
```

Output:

```
 Yes
```

Approach 3
1. Scan the last two nodes of preorder & mark them as min & max.
2. Scan every node down the preorder array. Each node must be either smaller than the min node or larger than the max node. Update min & max accordingly.

C

```
 #include <stdio.h>

int hasOnlyOneChild(int pre[], int size)
{
    // Initialize min and max using last two elements
    int min, max;
    if (pre[size-1] > pre[size-2])
    {
        max = pre[size-1];
        min = pre[size-2]):
    else
    {
        max = pre[size-2];
        min = pre[size-1];
    }


    // Every element must be either smaller than min or
    // greater than max
    for (int i=size-3; i>=0; i--)
    {
        if (pre[i] < min)
            min = pre[i];
        else if (pre[i] > max)
            max = pre[i];
        else
```

```
            return false;
    }
    return true;
}

// Driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre,size))
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Java

```
 // Check if each internal node of BST has only one child

class BinaryTree {

    boolean hasOnlyOneChild(int pre[], int size) {
        // Initialize min and max using last two elements
        int min, max;
        if (pre[size - 1] > pre[size - 2]) {
            max = pre[size - 1];
            min = pre[size - 2];
        } else {
            max = pre[size - 2];
            min = pre[size - 1];
        }

        // Every element must be either smaller than min or
        // greater than max
        for (int i = size - 3; i >= 0; i--) {
            if (pre[i] < min) {
                min = pre[i];
            } else if (pre[i] > max) {
                max = pre[i];
            } else {
                return false;
            }
        }
        return true;
    }
```

```
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{8, 3, 5, 7, 6};
        int size = pre.length;
        if (tree.hasOnlyOneChild(pre, size) == true) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python3

```
 # Check if each internal
# node of BST has only one child
# approach 2

def hasOnlyOneChild(pre,size):

    # Initialize min and max
    # using last two elements
    min=0; max=0

    if pre[size-1] > pre[size-2] :
        max = pre[size-1]
        min = pre[size-2]
    else :
        max = pre[size-2]
        min = pre[size-1]

    # Every element must be
    # either smaller than min or
    # greater than max
    for i in range(size-3, 0, -1):
        if pre[i] < min:
            min = pre[i]
        elif pre[i] > max:
            max = pre[i]
        else:
            return False
    return True

# Driver program to
# test above function
if __name__ == "__main__":
```

```
    pre = [8, 3, 5, 7, 6]

    size = len(pre)
    if (hasOnlyOneChild(pre, size)):
        print("Yes")
    else:
        print("No")

# This code is contributed by
# Harshit Saini
```

Output:

```
Yes
```

Source

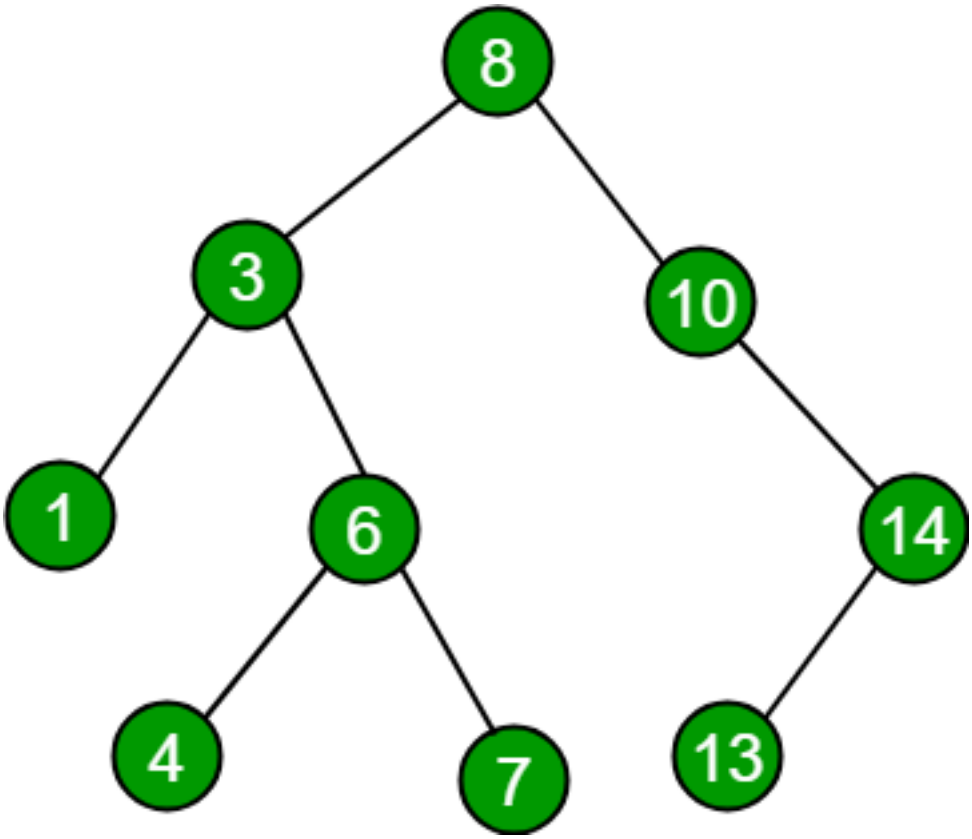
https://www.geeksforgeeks.org/check-if-each-internal-node-of-a-bst-has-exactly-one-child/

Chapter 19

Check if given sorted sub-sequence exists in binary search tree

Check if given sorted sub-sequence exists in binary search tree - GeeksforGeeks

Given a binary search tree and a sorted sub-sequence. the task is to check if the given sorted sub-sequence exist in binary search tree or not.



Examples:

```
// For above binary search tree
Input : seq[] = {4, 6, 8, 14}
Output: "Yes"

Input : seq[] = {4, 6, 8, 12, 13}
Output: "No"
```

A **simple solution** is to store **inorder traversal** in an auxiliary array and then by matching elements of sorted sub-sequence one by one with inorder traversal of tree , we can if sub-sequence exist in BST or not. Time complexity for this approach will be O(n) but it requires extra space O(n) for storing traversal in an array.

An **efficient solution** is to match elements of sub-sequence while we are traversing BST in **inorder** fashion. We take **index** as a iterator for given sorted sub-sequence and start inorder traversal of given bst, if **current node** matches with **seq[index]** then move **index** in forward direction by incrementing 1 and after complete traversal of BST if **index==n** that means all elements of given sub-sequence have been matched and exist as a sorted sub-sequence in given BST.

```
 // C++ program to find if given array exists as a
// subsequece in BST
#include<bits/stdc++.h>
using namespace std;

// A binary Tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

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

// A utility function to insert a given key to BST
struct Node* insert(struct Node* root, int key)
{
    if (root == NULL)
        return newNode(key);
    if (root->data > key)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

// function to check if given sorted sub-sequence exist in BST
// index --> iterator for given sorted sub-sequence
// seq[] --> given sorted sub-sequence
void seqExistUtil(struct Node *ptr, int seq[], int &index)
{
    if (ptr == NULL)
        return;

    // We traverse left subtree first in Inorder
    seqExistUtil(ptr->left, seq, index);

    // If current node matches with se[index] then move
    // forward in sub-sequence
    if (ptr->data == seq[index])
```

```
        index++;

    // We traverse left subtree in the end in Inorder
    seqExistUtil(ptr->right, seq, index);
}

// A wrapper over seqExistUtil. It returns true
// if seq[0..n-1] exists in tree.
bool seqExist(struct Node *root, int seq[], int n)
{
    // Initialize index in seq[]
    int index = 0;

    // Do an inorder traversal and find if all
    // elements of seq[] were present
    seqExistUtil(root, seq, index);

    // index would become n if all elements of
    // seq[] were present
    return (index == n);
}

// driver program to run the case
int main()
{
    struct Node* root = NULL;
    root = insert(root, 8);
    root = insert(root, 10);
    root = insert(root, 3);
    root = insert(root, 6);
    root = insert(root, 1);
    root = insert(root, 4);
    root = insert(root, 7);
    root = insert(root, 14);
    root = insert(root, 13);

    int seq[] = {4, 6, 8, 14};
    int n = sizeof(seq)/sizeof(seq[0]);

    seqExist(root, seq, n)? cout << "Yes" :
                            cout << "No";

    return 0;
}
```

Output:

```
Yes
```

Time complexity : O(n)

Improved By : break_it

Source

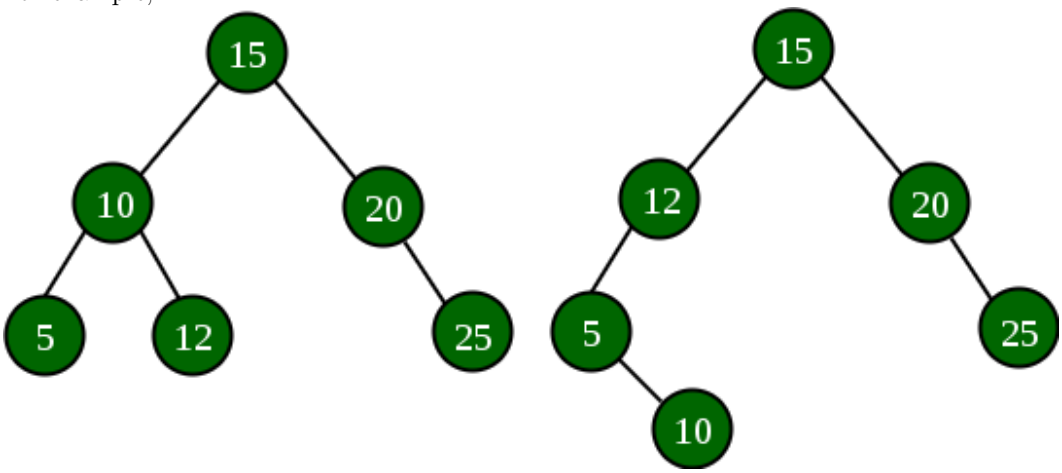
https://www.geeksforgeeks.org/check-if-given-sorted-sub-sequence-exists-in-binary-search-tree/

Chapter 20

Check if two BSTs contain same set of elements

Check if two BSTs contain same set of elements - GeeksforGeeks

Given two Binary Search Trees consisting of unique positive elements, we have to check whether the two BSTs contains same set or elements or not.
Note: The structure of the two given BSTs can be different.
For example,



The above two BSTs contains same set of elements {5, 10, 12, 15, 20, 25}

Method 1: The most simple method will be to traverse first tree and store its element in a list or array. Now, traverse 2nd tree and simultaneously check if the current element is present in the list or not. If yes, then mark the element in the list as negative and check for further elements otherwise if no, then immediately terminate the traversal and print No. If all the elements of 2nd tree is present in the list and are marked negative then finally traverse the list to check if there are any non-negative elements left. If Yes then it means that the first tree had some extra element otherwise the both tree consists same set of elements.

Time Complexity: O(n * n) , where n is the number of nodes in the BST.
Auxiliary Space: O(n).

Method 2: This method is an optimization of above approach. If we observe carefully, we will see that in the above approach, search for element in the list takes linear time. We can optimize this operation to be done in constant time using a hashmap instead of list. We insert elements of both trees in different hash sets. Finally we compare if both hash sets contain same elements or not.

Below is the complete implementation of above approach:

```
 // CPP program to check if two BSTs contains
// same set of elements
#include<bits/stdc++.h>
using namespace std;

// BST Node
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

// Utility function to create new Node
Node* newNode(int val)
{
    Node* temp = new Node;
    temp->data = val;
    temp->left = temp->right = NULL;
    return temp;
}

// function to insert elements of the
// tree to map m
void insertToHash(Node* root, unordered_set<int> &s)
{
    if (!root)
        return;
    insertToHash(root->left, s);
    s.insert(root->data);
    insertToHash(root->right, s);
}

// function to check if the two BSTs contain
// same set  of elements
bool checkBSTs(Node* root1, Node* root2)
{
    // Base cases
```

```
    if (!root1 && !root2)
        return true;
    if ((root1 && !root2) || (!root1 && root2))
        return false;

    // Create two hash sets and store
    // elements both BSTs in them.
    unordered_set<int> s1, s2;
    insertToHash(root1, s1);
    insertToHash(root2, s2);

    // Return true if both hash sets
    // contain same elements.
    return (s1 == s2);
}

// Driver program to check above functions
int main()
{
    // First BST
    Node* root1 = newNode(15);
    root1->left = newNode(10);
    root1->right = newNode(20);
    root1->left->left = newNode(5);
    root1->left->right = newNode(12);
    root1->right->right = newNode(25);

    // Second BST
    Node* root2 = newNode(15);
    root2->left = newNode(12);
    root2->right = newNode(20);
    root2->left->left = newNode(5);
    root2->left->left->right = newNode(10);
    root2->right->right = newNode(25);

    // check if two BSTs have same set of elements
    if (checkBSTs(root1, root2))
        cout << "YES";
    else
        cout << "NO";
    return 0;
}
```

Output:

```
YES
```

Time Complexity: O(n), where n is the number of nodes in the trees.
Auxiliary Space: O(n).

Method 3 : We know about an interesting property of BST that inorder traversal of a BST generates a sorted array. So we can do inorder traversals of both the BSTs and generate two arrays and finally we can compare these two arrays. If both of the arrays are same then the BSTs have same set of elements otherwise not.

```
 // CPP program to check if two BSTs contains
// same set of elements
#include<bits/stdc++.h>
using namespace std;

// BST Node
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

// Utility function to create new Node
Node* newNode(int val)
{
    Node* temp = new Node;
    temp->data = val;
    temp->left = temp->right = NULL;
    return temp;
}

// function to insert elements of the
// tree to map m
void storeInorder(Node* root, vector<int> &v)
{
    if (!root)
        return;
    storeInorder(root->left, v);
    v.push_back(root->data);
    storeInorder(root->right, v);
}

// function to check if the two BSTs contain
// same set  of elements
bool checkBSTs(Node* root1, Node* root2)
{
    // Base cases
    if (!root1 && !root2)
        return true;
    if ((root1 && !root2) || (!root1 && root2))
```

```
        return false;

    // Create two vectors and store
    // inorder traversals of both BSTs
    // in them.
    vector<int> v1, v2;
    storeInorder(root1, v1);
    storeInorder(root2, v2);

    // Return true if both vectors are
    // identical
    return (v1 == v2);
}

// Driver program to check above functions
int main()
{
    // First BST
    Node* root1 = newNode(15);
    root1->left = newNode(10);
    root1->right = newNode(20);
    root1->left->left = newNode(5);
    root1->left->right = newNode(12);
    root1->right->right = newNode(25);

    // Second BST
    Node* root2 = newNode(15);
    root2->left = newNode(12);
    root2->right = newNode(20);
    root2->left->left = newNode(5);
    root2->left->left->right = newNode(10);
    root2->right->right = newNode(25);

    // check if two BSTs have same set of elements
    if (checkBSTs(root1, root2))
        cout << "YES";
    else
        cout << "NO";
    return 0;
}
```

Output:

```
YES
```

Time Complexity: O(n).
Auxiliary Space: O(n).

Source

https://www.geeksforgeeks.org/check-two-bsts-contain-set-elements/

Chapter 21

Check whether BST contains Dead End or not

Check whether BST contains Dead End or not - GeeksforGeeks

Given a Binary search Tree that contains positive integer values greater then 0. The task is to check whether the BST contains a dead end or not. Here Dead End means, we are not able to insert any element after that node.

Examples:

```
Input :        8
             /   \
           5      9
         /   \
        2     7
       /
      1
Output : Yes
Explanation : Node "1" is the dead End because
         after that we cant insert any element.

Input :       8
            /   \
           7     10
         /      /   \
        2      9     13

Output : Yes
Explanation : We can't insert any element at
              node 9.
```

If we take a closer look at problem, we can notice that we basically need to check if there is leaf node with value x such that x+1 and x-1 exist in BST with exception of x = 1. For x = 1, we can't insert 0 as problem statement says BST contains positive integers only.

To implement above idea we first traverse whole BST and store all nodes in a hash_map. We also store all leaves in a separate hash to avoid re-traversal of BST. Finally we check for every leaf node x, if x-1 and x+1 are present in hash_map or not.

Below is C++ implementation of above idea .

```
 // C++ program check weather BST contains
// dead end or not
#include<bits/stdc++.h>
using namespace std;

// A BST node
struct Node
{
    int data;
    struct Node *left, *right;
};

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

/* 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->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);

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

// Function to store all node of given binary search tree
```

```
void storeNodes(Node * root, unordered_set<int> &all_nodes,
                          unordered_set<int> &leaf_nodes)
{
    if (root == NULL)
        return ;

    // store all node of binary search tree
    all_nodes.insert(root->data);

    // store leaf node in leaf_hash
    if (root->left==NULL && root->right==NULL)
    {
        leaf_nodes.insert(root->data);
        return ;
    }

    // recur call rest tree
    storeNodes(root-> left, all_nodes, leaf_nodes);
    storeNodes(root->right, all_nodes, leaf_nodes);
}

// Returns true if there is a dead end in tree,
// else false.
bool isDeadEnd(Node *root)
{
    // Base case
    if (root == NULL)
        return false ;

    // create two empty hash sets that store all
    // BST elements and leaf nodes respectively.
    unordered_set<int> all_nodes, leaf_nodes;

    // insert 0 in 'all_nodes' for handle case
    // if bst contain value 1
    all_nodes.insert(0);

    // Call storeNodes function to store all BST Node
    storeNodes(root, all_nodes, leaf_nodes);

    // Traversal leaf node and check Tree contain
    // continuous sequence of
    // size tree or Not
    for (auto i = leaf_nodes.begin() ; i != leaf_nodes.end(); i++)
    {
        int x = (*i);

        // Here we check first and last element of
```

```
        // continuous sequence that are x-1 & x+1
        if (all_nodes.find(x+1) != all_nodes.end() &&
            all_nodes.find(x-1) != all_nodes.end())
            return true;
    }

    return false ;
}

// Driver program
int main()
{
/*       8
       /   \
      5    11
     /  \
    2    7
     \
      3
       \
        4 */
    Node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 5);
    root = insert(root, 2);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 11);
    root = insert(root, 4);
    if (isDeadEnd(root) == true)
        cout << "Yes " << endl;
    else
        cout << "No " << endl;
    return 0;
}
```

Output:

```
Yes
```

Time Complexity : O(n)

Simple Recursive solution to check whether BST contains dead End

Source

https://www.geeksforgeeks.org/check-whether-bst-contains-dead-end-not/

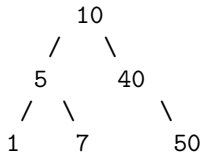
Chapter 22

Construct BST from given preorder traversal Set 1

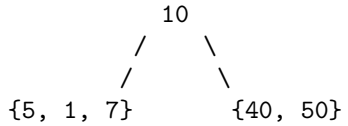
Construct BST from given preorder traversal Set 1 - GeeksforGeeks

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.



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.



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

C

```
 /* A O(n^2) program for construction of BST from preorder traversal */
#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 function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

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

    return temp;
}

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

    // The first node in preorder traversal is root. So take the node at
    // preIndex from pre[] and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    *preIndex = *preIndex + 1;

    // If the current subarry has only one element, no need to recur
    if (low == high)
        return root;

    // Search for the first element greater than root
    int i;
```

```
    for ( i = low; i <= high; ++i )
        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, preIndex, *preIndex, i - 1, size );
    root->right = constructTreeUtil ( pre, preIndex, i, high, size );

    return root;
}

// The main function to construct BST from given preorder traversal.
// This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, &preIndex, 0, size - 1, size);
}

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

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

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

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

    return 0;
}
```

Java

```
 // Java program to construct BST from given preorder traversal
```

```
// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class Index {

    int index = 0;
}

class BinaryTree {

    Index index = new Index();

    // A recursive function to construct Full from pre[]. preIndex is used
    // to keep track of index in pre[].
    Node constructTreeUtil(int pre[], Index preIndex,
            int low, int high, int size) {

        // Base case
        if (preIndex.index >= size || low > high) {
            return null;
        }

        // The first node in preorder traversal is root. So take the node at
        // preIndex from pre[] and make it root, and increment preIndex
        Node root = new Node(pre[preIndex.index]);
        preIndex.index = preIndex.index + 1;

        // If the current subarry has only one element, no need to recur
        if (low == high) {
            return root;
        }

        // Search for the first element greater than root
        int i;
        for (i = low; i <= high; ++i) {
            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, preIndex, preIndex.index, i - 1, size);
        root.right = constructTreeUtil(pre, preIndex, i, high, size);

        return root;
    }

    // The main function to construct BST from given preorder traversal.
    // This function mainly uses constructTreeUtil()
    Node constructTree(int pre[], int size) {
        return constructTreeUtil(pre, index, 0, size - 1, size);
    }

    // A utility function to print inorder traversal of a Binary Tree
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{10, 5, 1, 7, 40, 50};
        int size = pre.length;
        Node root = tree.constructTree(pre, size);
        System.out.println("Inorder traversal of the constructed tree is ");
        tree.printInorder(root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # A O(n^2) 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)

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

Output:

```
Inorder traveral of the constructed tree:
1 5 7 10 40 50
```

Time Complexity: O(n^2)

Method 2 (O(n) time complexity)
The idea used here is inspired from method 3 of thispost. 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}.
C

```
 /* A O(n) program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.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 function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct 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[].
struct node* constructTreeUtil( int pre[], int* preIndex, int key,
                                int min, int max, int size )
{
    // Base case
    if( *preIndex >= size )
        return NULL;

    struct 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()
struct 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 (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

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

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

    return 0;
}
```

Java

```
 // Java program to construct BST from given preorder traversal

// A binary tree node
class Node {
```

```
    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class Index {

    int index = 0;
}

class BinaryTree {

    Index index = new Index();

    // A recursive function to construct BST from pre[]. preIndex is used
    // to keep track of index in pre[].
    Node constructTreeUtil(int pre[], Index preIndex, int key,
            int min, int max, int size) {

        // Base case
        if (preIndex.index >= 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 = new Node(key);
            preIndex.index = preIndex.index + 1;

            if (preIndex.index < 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.index],
                        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.index],
                        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, index, pre[0], Integer.MIN_VALUE,
                Integer.MAX_VALUE, size);
    }

    // A utility function to print inorder traversal of a Binary Tree
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{10, 5, 1, 7, 40, 50};
        int size = pre.length;
        Node root = tree.constructTree(pre, size);
        System.out.println("Inorder traversal of the constructed tree is ");
        tree.printInorder(root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # 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)

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

Output:

```
Inorder traversal of Binary Tree:
1 5 7 10 40 50
```

Time Complexity: O(n)

We will soon publish a O(n) iterative solution as a separate post.

Source

https://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversa/

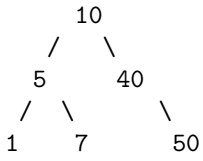
Chapter 23

Construct BST from given preorder traversal Set 2

Construct BST from given preorder traversal Set 2 - GeeksforGeeks

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.



We have discussed O(n^2) and O(n) recursive solutions in the previous post. Following is a stack based iterative solution that works in O(n) time.

1. Create an empty stack.

2. Make the first value as root. Push it to the stack.

3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.

4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.

5. Repeat steps 2 and 3 until there are items remaining in pre[].

C

```
 /* A O(n) iterative program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

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

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
```

```
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from pre[]
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );

    // The first element of pre[] is always root
    Node* root = newNode( pre[0] );

    // Push root
    push( stack, root );

    int i;
    Node* temp;

    // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
```

```
        while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
            temp = pop( stack );

        // Make this greater value as the right child and push it to the stack
        if ( temp != NULL)
        {
            temp->right = newNode( pre[i] );
            push( stack, temp->right );
        }

        // If the next value is less than the stack's top value, make this value
        // as the left child of the stack's top node. Push the new node to stack
        else
        {
            peek( stack )->left = newNode( pre[i] );
            push( stack, peek( stack )->left );
        }
    }

    return root;
}


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

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

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

    return 0;
}
```

Java

```
 // Java program to construct BST from given preorder traversal

import java.util.*;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    // The main function that constructs BST from pre[]
    Node constructTree(int pre[], int size) {

        // The first element of pre[] is always root
        Node root = new Node(pre[0]);

        Stack<Node> s = new Stack<Node>();

        // Push root
        s.push(root);

        // Iterate through rest of the size-1 items of given preorder array
        for (int i = 1; i < size; ++i) {
            Node temp = null;

            /* Keep on popping while the next value is greater than
             stack's top value. */
            while (!s.isEmpty() && pre[i] > s.peek().data) {
                temp = s.pop();
            }

            // Make this greater value as the right child and push it to the stack
            if (temp != null) {
                temp.right = new Node(pre[i]);
                s.push(temp.right);
            }

            // If the next value is less than the stack's top value, make this value
            // as the left child of the stack's top node. Push the new node to stack
            else {
```

```
                temp = s.peek();
                temp.left = new Node(pre[i]);
                s.push(temp.left);
            }
        }

        return root;
    }

    // A utility function to print inorder traversal of a Binary Tree
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int pre[] = new int[]{10, 5, 1, 7, 40, 50};
        int size = pre.length;
        Node root = tree.constructTree(pre, size);
        System.out.println("Inorder traversal of the constructed tree is ");
        tree.printInorder(root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
 1 5 7 10 40 50
```

Time Complexity: O(n). The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of constructTree(). Therefore, time complexity is O(n).

Source

https://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversal-set-2/

Chapter 24

Construct BST from its given level order traversal

Construct BST from its given level order traversal - GeeksforGeeks

Construct the BST (Binary Search Tree) from its given level order traversal.

Examples:

```
Input : arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}
Output : BST:
        7
       / \
      4   12
     / \  /
    3  6 8
   /  /   \
  1   5   10
```

The idea is to use a queue to construct tree. Every element of queue has a structure say **NodeDetails** which stores details of a tree node. The details are pointer to the node, and two variables **min** and **max** where **min** stores the lower limit for the node values which can be a part of the left subtree and **max** stores the upper limit for the node values which can be a part of the right subtree for the specified node in **NodeDetails** structure variable. For the 1st array value arr[0], create a node and then create a **NodeDetails** structure having pointer to this node and min = INT_MIN and max = INT_MAX. Add this structure variable to a queue. This Node will be the root of the tree. Move to 2nd element in arr[] and then perform the following steps:

1. Pop **NodeDetails** from the queue in **temp**.

2. Check whether the current array element can be a left child of the node in **temp** with the help of **min** and **temp.node** data values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, make this new node as the left child of temp's node and move to next element in arr[].
3. Check whether the current array element can be a right child of the node in **temp** with the help of **max** and **temp.node** data values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, make this new node as the right child of temp's node and move to next element in arr[].
4. Repeat steps 1, 2 and 3 until there are no more elements in arr[].

For a left child node, its **min** value will be its parent's 'min' value and **max** value will be the data value of its parent node. For a right child node, its **min** value will be the data value of its parent node and **max** value will be its parent's 'max' value.

Below is C++ implementation of above approach:

```
 // C++ implementation to construct a BST
// from its level order traversal
#include <bits/stdc++.h>

using namespace std;

// node of a BST
struct Node
{
    int data;
    Node *left, *right;
};

// to store details of a node like
// pointer to the node, 'min' and 'max'
// to obtain the range of values where
// node's left and right child's could lie
struct NodeDetails
{
    Node *ptr;
    int min, max;
};

// function to get a new node
Node* getNode(int data)
{
    // Allocate memory
    Node *newNode =
        (Node*)malloc(sizeof(Node));
```

```
    // put in the data
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function to construct a BST from
// its level order traversal
Node* constructBst(int arr[], int n)
{
    // if tree is empty
    if (n == 0)
        return NULL;

    Node *root;

    // queue to store NodeDetails
    queue<NodeDetails> q;

    // index variable to access array elements
    int i=0;

    // node details for the
    // root of the BST
    NodeDetails newNode;
    newNode.ptr = getNode(arr[i++]);
    newNode.min = INT_MIN;
    newNode.max = INT_MAX;
    q.push(newNode);

    // getting the root of the BST
    root = newNode.ptr;

    // until there are no more elements
    // in arr[]
    while (i != n)
    {
        // extracting NodeDetails of a
        // node from the queue
        NodeDetails temp = q.front();
        q.pop();

        // check whether there are more elements
        // in the arr[] and arr[i] can be left child
        // of 'temp.ptr' or not
        if (i < n && (arr[i] < temp.ptr->data &&
                          arr[i] > temp.min))
```

```
        {
            // Create NodeDetails for newNode
            /// and add it to the queue
            newNode.ptr = getNode(arr[i++]);
            newNode.min = temp.min;
            newNode.max = temp.ptr->data;
            q.push(newNode);

            // make this 'newNode' as left child
            // of 'temp.ptr'
            temp.ptr->left = newNode.ptr;
        }

        // check whether there are more elements
        // in the arr[] and arr[i] can be right child
        // of 'temp.ptr' or not
        if (i < n && (arr[i] > temp.ptr->data &&
                              arr[i] < temp.max))
        {
            // Create NodeDetails for newNode
            /// and add it to the queue
            newNode.ptr = getNode(arr[i++]);
            newNode.min = temp.ptr->data;
            newNode.max = temp.max;
            q.push(newNode);

            // make this 'newNode' as right child
            // of 'temp.ptr'
            temp.ptr->right = newNode.ptr;
        }
    }

    // root of the required BST
    return root;
}

// function to print the inorder traversal
void inorderTraversal(Node* root)
{
    if (!root)
        return;

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

// Driver program to test above
```

```
int main()
{
    int arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    Node *root = constructBst(arr, n);

    cout << "Inorder Traversal: ";
    inorderTraversal(root);
    return 0;
}
```

Output:

```
Inorder Traversal: 1 3 4 5 6 7 8 10 12
```

Time Complexity : O(n)

Source

https://www.geeksforgeeks.org/construct-bst-given-level-order-traversal/

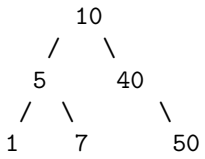
Chapter 25

Construct a Binary Search Tree from given postorder

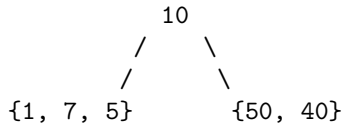
Construct a Binary Search Tree from given postorder - GeeksforGeeks

Given postorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {1, 7, 5, 50, 40, 10}, then following tree should be constructed and root of the tree should be returned.



Method 1 (O(n^2) time complexity)
The last element of postorder traversal is always root. We first construct the root. Then we find the index of last element which is smaller than root. Let the index be 'i'. The values between 0 and 'i' are part of left subtree, and the values between 'i+1' and 'n-2' are part of right subtree. Divide given post[] at index "i" and recur for left and right sub-trees.
For example in {1, 7, 5, 40, 50, 10}, 10 is the last element, so we make it root. Now we look for the last element smaller than 10, we find 5. So we know the structure of BST is as following.



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

Method 2 (O(n) time complexity)
The trick is to set a range {min .. max} for every node. Initialize the range as {INT_MIN .. INT_MAX}. The last 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 .. INT_MAX}.

Following code is used to generate the exact Binary Search Tree of a given post order traversal.

C

```
 /* A O(n) program for construction of BST from
   postorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

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

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

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

    return temp;
}

// A recursive function to construct BST from post[].
// postIndex is used to keep track of index in post[].
struct node* constructTreeUtil(int post[], int* postIndex,
                          int key, int min, int max, int size)
{
    // Base case
    if (*postIndex < 0)
        return NULL;
```

```
    struct node* root = NULL;

    // If current element of post[] is in range, then
    // only it is part of current subtree
    if (key > min && key < max)
    {
        // Allocate memory for root of this subtree and decrement
        // *postIndex
        root = newNode(key);
        *postIndex = *postIndex - 1;

        if (*postIndex >= 0)
        {

          // 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(post, postIndex, post[*postIndex],
                                           key, max, 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(post, postIndex, post[*postIndex],
                                         min, key, size );
        }
    }
    return root;
}

// The main function to construct BST from given postorder
// traversal. This function mainly uses constructTreeUtil()
struct node *constructTree (int post[], int size)
{
    int postIndex = size-1;
    return constructTreeUtil(post, &postIndex, post[postIndex],
                             INT_MIN, INT_MAX, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}
```

```
// Driver program to test above functions
int main ()
{
    int post[] = {1, 7, 5, 50, 40, 10};
    int size = sizeof(post) / sizeof(post[0]);

    struct node *root = constructTree(post, size);

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

    return 0;
}
```

Java

```
 /* A O(n) program for construction of BST from
   postorder traversal */

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

    Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

// Class containing variable that keeps a track of overall
// calculated postindex
class Index
{
    int postindex = 0;
}

class BinaryTree
{
    // A recursive function to construct BST from post[].
    // postIndex is used to keep track of index in post[].
    Node constructTreeUtil(int post[], Index postIndex,
            int key, int min, int max, int size)
    {
```

```
    // Base case
    if (postIndex.postindex < 0)
        return null;

    Node root = null;

    // If current element of post[] is in range, then
    // only it is part of current subtree
    if (key > min && key < max)
    {
        // Allocate memory for root of this subtree and decrement
        // *postIndex
        root = new Node(key);
        postIndex.postindex = postIndex.postindex - 1;

        if (postIndex.postindex > 0)
        {
            // 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(post, postIndex,
                    post[postIndex.postindex],key, max, 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(post, postIndex,
                    post[postIndex.postindex],min, key, size);
        }
    }
    return root;
}

// The main function to construct BST from given postorder
// traversal. This function mainly uses constructTreeUtil()
Node constructTree(int post[], int size)
{
    Index index = new Index();
    index.postindex = size - 1;
    return constructTreeUtil(post, index, post[index.postindex],
            Integer.MIN_VALUE, Integer.MAX_VALUE, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(Node node)
{
    if (node == null)
        return;
```

```
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        int post[] = new int[]{1, 7, 5, 50, 40, 10};
        int size = post.length;

        Node root = tree.constructTree(post, size);

        System.out.println("Inorder traversal of the constructed tree:");
        tree.printInorder(root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder traversal of the constructed tree:
1 5 7 10 40 50
```

Note that the output to the program will always be a sorted sequence as we are printing the inorder traversal of a Binary Search Tree.

Source

https://www.geeksforgeeks.org/construct-a-binary-search-tree-from-given-postorder/

Chapter 26

Construct all possible BSTs for keys 1 to N

Construct all possible BSTs for keys 1 to N - GeeksforGeeks

In this article, first count of possible BST (Binary Search Trees)s is discussed, then construction of all possible BSTs.

How many structurally unique BSTs for keys from 1..N?

```
For example, for N = 2, there are 2 unique BSTs
     1               2
      \             /
       2           1

For N = 3, there are 5 possible BSTs
  1              3        3         2      1
    \           /        /         /  \      \
     3        2         1         1    3      2
    /        /           \                     \
   2       1             2                     3
```

We strongly recommend you to minimize your browser and try this yourself first.

We know that all node in left subtree are smaller than root and in right subtree are larger than root so if we have ith number as root, all numbers from 1 to i-1 will be in left subtree and i+1 to N will be in right subtree. If 1 to i-1 can form x different trees and i+1 to N can from y different trees then we will have x*y total trees when ith number is root and we also have N choices for root also so we can simply iterate from 1 to N for root and another loop for left and right subtree. If we take a closer look, we can notice that the count is basically n'th Catalan number. We have discussed different approaches to find n'th Catalan number here.

How to construct all BST for keys 1..N?
The idea is to maintain a list of roots of all BSTs. Recursively construct all possible left and right subtrees. Create a tree for every pair of left and right subtree and add the tree to list. Below is detailed algorithm.

```
1) Initialize list of BSTs as empty.
2) For every number i where i varies from 1 to N, do following
......a)  Create a new node with key as 'i', let this node be 'node'
......b)  Recursively construct list of all left subtrees.
......c)  Recursively construct list of all right subtrees.
3) Iterate for all left subtrees
   a) For current leftsubtree, iterate for all right subtrees
        Add current left and right subtrees to 'node' and add
        'node' to list.
```

Below is C++ implementation of above idea.

```
 // A C++ prgroam to contrcut all unique BSTs for keys from 1 to n
#include <iostream>
#include<vector>
using namespace std;

//  node structure
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 =  new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do preorder traversal of BST
void preorder(struct node *root)
{
    if (root != NULL)
    {
        cout << root->key << " ";
        preorder(root->left);
```

```
        preorder(root->right);
    }
}

//  function for constructing trees
vector<struct node *> constructTrees(int start, int end)
{
    vector<struct node *> list;

    /*  if start > end   then subtree will be empty so returning NULL
        in the list */
    if (start > end)
    {
        list.push_back(NULL);
        return list;
    }

    /*  iterating through all values from start to end  for constructing\
        left and right subtree recursively  */
    for (int i = start; i <= end; i++)
    {
        /*  constructing left subtree   */
        vector<struct node *> leftSubtree  = constructTrees(start, i - 1);

        /*  constructing right subtree  */
        vector<struct node *> rightSubtree = constructTrees(i + 1, end);

        /*  now looping through all left and right subtrees and connecting
            them to ith root  below  */
        for (int j = 0; j < leftSubtree.size(); j++)
        {
            struct node* left = leftSubtree[j];
            for (int k = 0; k < rightSubtree.size(); k++)
            {
                struct node * right = rightSubtree[k];
                struct node * node = newNode(i);// making value i as root
                node->left = left;              // connect left subtree
                node->right = right;            // connect right subtree
                list.push_back(node);           // add this tree to list
            }
        }
    }
    return list;
}

// Driver Program to test above functions
int main()
{
```

```
    // Construct all possible BSTs
    vector<struct node *> totalTreesFrom1toN = constructTrees(1, 3);


    /*  Printing preorder traversal of all constructed BSTs   */
    cout << "Preorder traversals of all constructed BSTs are \n";
    for (int i = 0; i < totalTreesFrom1toN.size(); i++)
    {
        preorder(totalTreesFrom1toN[i]);
        cout << endl;
    }
    return 0;
}
```

Output:

```
Preorder traversals of all constructed BSTs are
1 2 3
1 3 2
2 1 3
3 1 2
3 2 1
```

This article is contributed by **Utkarsh Trivedi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/construct-all-possible-bsts-for-keys-1-to-n/

Chapter 27

Convert BST to Max Heap

Convert BST to Max Heap - GeeksforGeeks

Given a Binary Search Tree which is also a Complete Binary Tree. The problem is to convert a given BST into a Special Max Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Max Heap.

Examples:

```
Input :          4
               /   \
              2     6
            /  \   /  \
           1   3  5    7

Output :       7
             /   \
            3     6
          /   \  /   \
         1    2 4     5
The given BST has been transformed into a
Max Heap.
All the nodes in the Max Heap satisfies the given
condition, that is, values in the left subtree of
a node should be less than the values in the right
subtree of the node.
```

Pre Requisites: Binary Seach Tree Heaps

Approach
1. Create an array **arr**[] of size n, where n is the number of nodes in the given BST.
2. Perform the inorder traversal of the BST and copy the node values in the **arr**[] in sorted

order.
3. Now perform the postorder traversal of the tree.
4. While traversing the root during the postorder traversal, one by one copy the values from the array **arr**[] to the nodes.

```
 // C++ implementation to convert a given
// BST to Max Heap
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* getNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function prototype for postorder traversal
// of the given tree
void postorderTraversal(Node*);

// Function for the inorder traversal of the tree
// so as to store the node values in 'arr' in
// sorted order
void inorderTraversal(Node* root, vector<int>& arr)
{
    if (root == NULL)
        return;

    // first recur on left subtree
    inorderTraversal(root->left, arr);

    // then copy the data of the node
    arr.push_back(root->data);

    // now recur for right subtree
    inorderTraversal(root->right, arr);
}
```

```
void BSTToMaxHeap(Node* root, vector<int> arr, int* i)
{
    if (root == NULL)
        return;

    // recur on left subtree
    BSTToMaxHeap(root->left, arr, i);

    // recur on right subtree
    BSTToMaxHeap(root->right, arr, i);

    // copy data at index 'i' of 'arr' to
    // the node
    root->data = arr[++*i];
}

// Utility function to convert the given BST to
// MAX HEAP
void convertToMaxHeapUtil(Node* root)
{
    // vector to store the data of all the
    // nodes of the BST
    vector<int> arr;
    int i = -1;

    // inorder traversal to populate 'arr'
    inorderTraversal(root, arr);

    // BST to MAX HEAP conversion
    BSTToMaxHeap(root, arr, &i);
}

// Function to Print Postorder Traversal of the tree
void postorderTraversal(Node* root)
{
    if (!root)
        return;

    // recur on left subtree
    postorderTraversal(root->left);

    // then recur on right subtree
    postorderTraversal(root->right);

    // print the root's data
    cout << root->data << " ";
}
```

```
// Driver Code
int main()
{
    // BST formation
    struct Node* root = getNode(4);
    root->left = getNode(2);
    root->right = getNode(6);
    root->left->left = getNode(1);
    root->left->right = getNode(3);
    root->right->left = getNode(5);
    root->right->right = getNode(7);

    convertToMaxHeapUtil(root);
    cout << "Postorder Traversal of Tree:" << endl;
    postorderTraversal(root);

    return 0;
}
```

Output:

```
Postorder Traversal of Tree:
1 2 3 4 5 6 7
```

Time Complexity: O(n)
Auxiliary Space: O(n)
where, n is the number of nodes in the tree

Source

https://www.geeksforgeeks.org/convert-bst-to-max-heap/

Chapter 28

Convert BST to Min Heap

Convert BST to Min Heap - GeeksforGeeks

Given a binary search tree which is also a complete binary tree. The problem is to convert the given BST into a Min Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Min Heap.

Examples:

```
Input :          4
               /   \
              2     6
            /  \   /  \
           1   3  5    7

Output :        1
              /   \
             2     5
           /  \   /  \
          3   4  6    7

The given BST has been transformed into a
Min Heap.
All the nodes in the Min Heap satisfies the given
condition, that is, values in the left subtree of
a node should be less than the values in the right
subtree of the node.
```

1. Create an array **arr**[] of size **n**, where n is the number of nodes in the given BST.
2. Perform the inorder traversal of the BST and copy the node values in the **arr**[] in sorted order.

3. Now perform the preorder traversal of the tree.
4. While traversing the root during the preorder traversal, one by one copy the values from the array **arr**[] to the nodes.

```
 // C++ implementation to convert the given
// BST to Min Heap
#include <bits/stdc++.h>
using namespace std;

// structure of a node of BST
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* getNode(int data)
{
    struct Node *newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function prototype for preorder traversal
// of the given tree
void preorderTraversal(Node*);

// function for the inorder traversal of the tree
// so as to store the node values in 'arr' in
// sorted order
void inorderTraversal(Node *root, vector<int>& arr)
{
    if (root == NULL)
        return;

    // first recur on left subtree
    inorderTraversal(root->left, arr);

    // then copy the data of the node
    arr.push_back(root->data);

    // now recur for right subtree
    inorderTraversal(root->right, arr);
```

```
}

// function to convert the given BST to MIN HEAP
// performs preorder traversal of the tree
void BSTToMinHeap(Node *root, vector<int> arr, int *i)
{
    if (root == NULL)
        return;

    // first copy data at index 'i' of 'arr' to
    // the node
    root->data = arr[++*i];

    // then recur on left subtree
    BSTToMinHeap(root->left, arr, i);

    // now recur on right subtree
    BSTToMinHeap(root->right, arr, i);
}

// utility function to convert the given BST to
// MIN HEAP
void convertToMinHeapUtil(Node *root)
{
    // vector to store the data of all the
    // nodes of the BST
    vector<int> arr;
    int i = -1;

    // inorder traversal to populate 'arr'
    inorderTraversal(root, arr);

    // BST to MIN HEAP conversion
    BSTToMinHeap(root, arr, &i);
}

// function for the preorder traversal of the tree
void preorderTraversal(Node *root)
{
    if (!root)
        return;

    // first print the root's data
    cout << root->data << " ";

    // then recur on left subtree
    preorderTraversal(root->left);
```

```
    // now recur on right subtree
    preorderTraversal(root->right);
}

// Driver program to test above
int main()
{
    // BST formation
    struct Node *root = getNode(4);
    root->left = getNode(2);
    root->right = getNode(6);
    root->left->left = getNode(1);
    root->left->right = getNode(3);
    root->right->left = getNode(5);
    root->right->right = getNode(7);

    convertToMinHeapUtil(root);
    cout << "Preorder Traversal:" << endl;
    preorderTraversal(root);

    return 0;
}
```

Output:

```
Preorder Traversal:
1 2 3 4 5 6 7
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Source

https://www.geeksforgeeks.org/convert-bst-min-heap/

Chapter 29

Convert a BST to a Binary Tree such that sum of all greater keys is added to every key

Convert a BST to a Binary Tree such that sum of all greater keys is added to every key

Given a Binary Search Tree (BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all greater keys in BST.

Examples:

```
Input: Root of following BST
              5
            /   \
           2     13

Output: The given BST is converted to following Binary Tree
              18
            /   \
          20     13
```

Solution: Do reverse Inoorder traversal. Keep track of the sum of nodes visited so far. Let this sum be *sum*. For every node currently being visited, first add the key of this node to *sum*, i.e. *sum = sum + node->key*. Then change the key of current node to *sum*, i.e., *node->key = sum*.
When a BST is being traversed in reverse Inorder, for every key currently being visited, all keys that are already visited are all greater keys.

C

```
// Program to change a BST to Binary Tree such that key of a node becomes
```

```
// original key plus sum of all greater keys in BST
#include <stdio.h>
#include <stdlib.h>

/* A BST node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* 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;
    return (node);
}

// A recursive function that traverses the given BST in reverse inorder and
// for every key, adds all greater keys to it
void addGreaterUtil(struct node *root, int *sum_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    // Recur for right subtree first so that sum of all greater
    // nodes is stored at sum_ptr
    addGreaterUtil(root->right, sum_ptr);

    // Update the value at sum_ptr
    *sum_ptr = *sum_ptr + root->key;

    // Update key of this node
    root->key = *sum_ptr;

    // Recur for left subtree so that the updated sum is added
    // to smaller nodes
    addGreaterUtil(root->left, sum_ptr);
}

// A wrapper over addGreaterUtil().  It initializes sum and calls
// addGreaterUtil() to recursivel upodate and use value of sum
```

```
void addGreater(struct node *root)
{
    int sum = 0;
    addGreaterUtil(root, &sum);
}

// A utility function to print inorder traversal of Binary Tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->key);
    printInorder(node->right);
}

// Driver program to test above function
int main()
{
    /* Create following BST
              5
            /   \
           2     13  */
    node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(13);

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

    addGreater(root);

    printf("\n Inorder traversal of the modified tree\n");
    printInorder(root);

    return 0;
}
```

Java

```
 // Java program to convert BST to binary tree such that sum of
// all greater keys is added to every key

class Node {

    int data;
    Node left, right;
```

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

class Sum {

    int sum = 0;
}

class BinaryTree {

    static Node root;
    Sum summ = new Sum();

    // A recursive function that traverses the given BST in reverse inorder and
    // for every key, adds all greater keys to it
    void addGreaterUtil(Node node, Sum sum_ptr) {

        // Base Case
        if (node == null) {
            return;
        }

        // Recur for right subtree first so that sum of all greater
        // nodes is stored at sum_ptr
        addGreaterUtil(node.right, sum_ptr);

        // Update the value at sum_ptr
        sum_ptr.sum = sum_ptr.sum + node.data;

        // Update key of this node
        node.data = sum_ptr.sum;

        // Recur for left subtree so that the updated sum is added
        // to smaller nodes
        addGreaterUtil(node.left, sum_ptr);
    }

    // A wrapper over addGreaterUtil().  It initializes sum and calls
    // addGreaterUtil() to recursivel upodate and use value of sum
    Node addGreater(Node node) {
        addGreaterUtil(node, summ);
        return node;
    }

    // A utility function to print inorder traversal of Binary Tree
```

```
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test the above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(5);
        tree.root.left = new Node(2);
        tree.root.right = new Node(13);

        System.out.println("Inorder traversal of given tree ");
        tree.printInorder(root);
        Node node = tree.addGreater(root);
        System.out.println("");
        System.out.println("Inorder traversal of modified tree ");
        tree.printInorder(node);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
 Inorder traversal of the given tree
2 5 13
 Inorder traversal of the modified tree
20 18 13
```

Time Complexity: O(n) where n is the number of nodes in given Binary Search Tree.

Source

https://www.geeksforgeeks.org/convert-bst-to-a-binary-tree/

Chapter 30

Convert a normal BST to Balanced BST

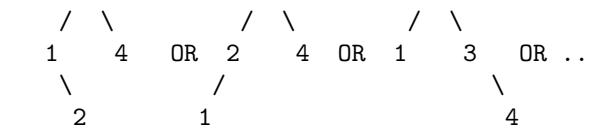
Convert a normal BST to Balanced BST - GeeksforGeeks

Given a BST (**B**inary **S**earch **T**ree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

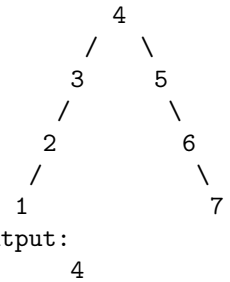
Examples :

```
Input:
       30
      /
     20
    /
   10
Output:
     20
   /   \
 10     30
```

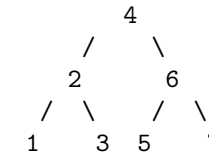
```
Input:
         4
        /
       3
      /
     2
    /
   1
Output:
      3            3           2
```



Input:



Output:



A **Simple Solution** is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is O(n Log n) and this solution doesn't guarantee

An **Efficient Solution** can construct balanced BST in O(n) time with minimum possible height. Below are steps.

1. Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2. Build a balanced BST from the above created sorted array using the recursive approach discussed here. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

Below is C++ implementation of above steps.
C++

```
 // C++ program to convert a left unbalanced BST to
// a balanced BST
#include <bits/stdc++.h>
using namespace std;

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

```
/* This function traverse the skewed binary tree and
   stores its nodes pointers in vector nodes[] */
void storeBSTNodes(Node* root, vector<Node*> &nodes)
{
    // Base case
    if (root==NULL)
        return;

    // Store nodes in Inorder (which is sorted
    // order for BST)
    storeBSTNodes(root->left, nodes);
    nodes.push_back(root);
    storeBSTNodes(root->right, nodes);
}

/* Recursive function to construct binary tree */
Node* buildTreeUtil(vector<Node*> &nodes, int start,
                    int end)
{
    // base case
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    Node *root = nodes[mid];

    /* Using index in Inorder traversal, construct
       left and right subtress */
    root->left  = buildTreeUtil(nodes, start, mid-1);
    root->right = buildTreeUtil(nodes, mid+1, end);

    return root;
}

// This functions converts an unbalanced BST to
// a balanced BST
Node* buildTree(Node* root)
{
    // Store nodes of given BST in sorted order
    vector<Node *> nodes;
    storeBSTNodes(root, nodes);

    // Constucts BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n-1);
}
```

```
// Utility function to create a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Function to do preorder traversal of tree */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver program
int main()
{
    /* Constructed skewed binary tree is
                10
               /
              8
             /
            7
           /
          6
         /
        5   */

    Node* root = newNode(10);
    root->left = newNode(8);
    root->left->left = newNode(7);
    root->left->left->left = newNode(6);
    root->left->left->left->left = newNode(5);

    root = buildTree(root);

    printf("Preorder traversal of balanced "
            "BST is : \n");
    preOrder(root);

    return 0;
}
```

Java

```
 // Java program to convert a left unbalanced BST to a balanced BST

import java.util.*;

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

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* This function traverse the skewed binary tree and
       stores its nodes pointers in vector nodes[] */
    void storeBSTNodes(Node root, Vector<Node> nodes)
    {
        // Base case
        if (root == null)
            return;

        // Store nodes in Inorder (which is sorted
        // order for BST)
        storeBSTNodes(root.left, nodes);
        nodes.add(root);
        storeBSTNodes(root.right, nodes);
    }

    /* Recursive function to construct binary tree */
    Node buildTreeUtil(Vector<Node> nodes, int start,
            int end)
    {
        // base case
        if (start > end)
            return null;

        /* Get the middle element and make it root */
```

```
    int mid = (start + end) / 2;
    Node node = nodes.get(mid);

    /* Using index in Inorder traversal, construct
       left and right subtress */
    node.left = buildTreeUtil(nodes, start, mid - 1);
    node.right = buildTreeUtil(nodes, mid + 1, end);

    return node;
}

// This functions converts an unbalanced BST to
// a balanced BST
Node buildTree(Node root)
{
    // Store nodes of given BST in sorted order
    Vector<Node> nodes = new Vector<Node>();
    storeBSTNodes(root, nodes);

    // Constucts BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n - 1);
}

/* Function to do preorder traversal of tree */
void preOrder(Node node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.left);
    preOrder(node.right);
}

// Driver program to test the above functions
public static void main(String[] args)
{
     /* Constructed skewed binary tree is
            10
           /
          8
         /
        7
       /
      6
     /
    5   */
    BinaryTree tree = new BinaryTree();
```

```
        tree.root = new Node(10);
        tree.root.left = new Node(8);
        tree.root.left.left = new Node(7);
        tree.root.left.left.left = new Node(6);
        tree.root.left.left.left.left = new Node(5);

        tree.root = tree.buildTree(tree.root);
        System.out.println("Preorder traversal of balanced BST is :");
        tree.preOrder(tree.root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

```
Preorder traversal of balanced BST is :
7 5 6 8 10
```

Source

https://www.geeksforgeeks.org/convert-normal-bst-balanced-bst/

Chapter 31

Count BST nodes that lie in a given range

Count BST nodes that lie in a given range - GeeksforGeeks

Given a Binary Search Tree (BST) and a range, count number of nodes that lie in the given range.

Examples:

```
Input:
        10
      /    \
    5       50
   /       /  \
 1       40   100
Range: [5, 45]

Output:  3
There are three nodes in range, 5, 10 and 40
```

The idea is to traverse the given binary search tree starting from root. For every node being visited, check if this node lies in range, if yes, then add 1 to result and recur for both of its children. If current node is smaller than low value of range, then recur for right child, else recur for left child.

Below is the implementation of above idea.
C++

```
 // C++ program to count BST nodes withing a given range
#include<bits/stdc++.h>
using namespace std;
```

```
// A BST node
struct node
{
    int data;
    struct node* left, *right;
};

// Utility function to create new node
node *newNode(int data)
{
    node *temp = new node;
    temp->data  = data;
    temp->left  = temp->right = NULL;
    return (temp);
}

// Returns count of nodes in BST in range [low, high]
int getCount(node *root, int low, int high)
{
    // Base case
    if (!root) return 0;

    // Special Optional case for improving efficiency
    if (root->data == high && root->data == low)
        return 1;

    // If current node is in range, then include it in count and
    // recur for left and right children of it
    if (root->data <= high && root->data >= low)
         return 1 + getCount(root->left, low, high) +
                    getCount(root->right, low, high);

    // If current node is smaller than low, then recur for right
    // child
    else if (root->data < low)
         return getCount(root->right, low, high);

    // Else recur for left child
    else return getCount(root->left, low, high);
}

// Driver program
int main()
{
    // Let us construct the BST shown in the above figure
    node *root        = newNode(10);
    root->left        = newNode(5);
```

```
    root->right       = newNode(50);
    root->left->left  = newNode(1);
    root->right->left = newNode(40);
    root->right->right = newNode(100);
    /* Let us constructed BST shown in above example
          10
        /    \
      5       50
     /       /  \
    1       40   100   */
    int l = 5;
    int h = 45;
    cout << "Count of nodes between [" << l << ", " << h
         << "] is " << getCount(root, l, h);
    return 0;
}
```

Java

```
 // Java code to count BST nodes that
// lie in a given range
class BinarySearchTree {

    /* Class containing left and right child
    of current node and key value*/
    static class Node {
        int data;
        Node left, right;

        public Node(int item) {
            data = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Returns count of nodes in BST in
    // range [low, high]
    int getCount(Node node, int low, int high)
    {
        // Base Case
```

```
        if(node == null)
            return 0;

        // If current node is in range, then
        // include it in count and recur for
        // left and right children of it
        if(node.data >= low && node.data <= high)
            return 1 + this.getCount(node.left, low, high)+
                this.getCount(node.right, low, high);

        // If current node is smaller than low,
        // then recur for right child
        else if(node.data < low)
            return this.getCount(node.right, low, high);

        // Else recur for left child
        else
            return this.getCount(node.left, low, high);
    }

    // Driver function
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        tree.root = new Node(10);
        tree.root.left     = new Node(5);
        tree.root.right     = new Node(50);
        tree.root.left.left = new Node(1);
        tree.root.right.left = new Node(40);
        tree.root.right.right = new Node(100);
        /* Let us constructed BST shown in above example
          10
        /    \
      5       50
     /       /  \
    1       40   100   */
    int l=5;
    int h=45;
    System.out.println("Count of nodes between [" + l + ", "
                       + h+ "] is " + tree.getCount(tree.root,
                                                    l, h));
    }
}
// This code is contributed by Kamal Rawal
```

Output:

```
Count of nodes between [5, 45] is 3
```

Time complexity of the above program is O(h + k) where h is height of BST and k is number of nodes in given range.

This article is contributed by Gaurav Ahirwar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/count-bst-nodes-that-are-in-a-given-range/

Chapter 32

Count BST subtrees that lie in given range

Count BST subtrees that lie in given range - GeeksforGeeks

Given a Binary Search Tree (BST) of integer values and a range [low, high], return count of nodes where all the nodes under that node (or subtree rooted with that node) lie in the given range.

Examples:

```
Input:
        10
      /    \
    5       50
   /       /  \
 1       40   100
Range: [5, 45]
Output:  1
There is only 1 node whose subtree is in the given range.
The node is 40
```

```
Input:
        10
      /    \
    5       50
   /       /  \
 1       40   100
Range: [1, 45]
Output:  3
There are three nodes whose subtree is in the given range.
```

```
The nodes are 1, 5 and 40
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the given Binary Search Tree (BST) in bottom up manner. For every node, recur for its subtrees, if subtrees are in range and the nodes is also in range, then increment count and return true (to tell the parent about its status). Count is passed as a pointer so that it can be incremented across all function calls.

Below is C++ implementation of the above idea.

```
 // C++ program to count subtrees that lie in a given range
#include<bits/stdc++.h>
using namespace std;

// A BST node
struct node
{
    int data;
    struct node* left, *right;
};

// A utility function to check if data of root is
// in range from low to high
bool inRange(node *root, int low, int high)
{
    return root->data >= low && root->data <= high;
}

// A recursive function to get count of nodes whose subtree
// is in range from low to hgih.  This function returns true
// if nodes in subtree rooted under 'root' are in range.
bool getCountUtil(node *root, int low, int high, int *count)
{
    // Base case
    if (root == NULL)
       return true;

    // Recur for left and right subtrees
    bool l = getCountUtil(root->left, low, high, count);
    bool r = getCountUtil(root->right, low, high, count);


    // If both left and right subtrees are in range and current node
    // is also in range, then increment count and return true
    if (l && r && inRange(root, low, high))
```

```
    {
       ++*count;
       return true;
    }

    return false;
}

// A wrapper over getCountUtil(). This function initializes count as 0
//  and calls getCountUtil()
int getCount(node *root, int low, int high)
{
    int count = 0;
    getCountUtil(root, low, high, &count);
    return count;
}

// Utility function to create new node
node *newNode(int data)
{
    node *temp = new node;
    temp->data  = data;
    temp->left  = temp->right = NULL;
    return (temp);
}

// Driver program
int main()
{
    // Let us construct the BST shown in the above figure
    node *root        = newNode(10);
    root->left        = newNode(5);
    root->right       = newNode(50);
    root->left->left  = newNode(1);
    root->right->left = newNode(40);
    root->right->right = newNode(100);
    /* Let us constructed BST shown in above example
          10
        /    \
      5       50
     /       /  \
    1       40   100   */
    int l = 5;
    int h = 45;
    cout << "Count of subtrees in [" << l << ", "
         << h << "] is " << getCount(root, l, h);
    return 0;
}
```

Output:

```
Count of subtrees in [5, 45] is 1
```

This article is contributed by Gaurav Ahirwar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Rohan Agarwal 2

Source

https://www.geeksforgeeks.org/count-bst-subtrees-that-lie-in-given-range/

Chapter 33

Count greater nodes in AVL tree

Count greater nodes in AVL tree - GeeksforGeeks

In this article we will see that how to calculate number of elements which are greater than given value in AVL tree.
Examples:

```
Input : x = 5
        Root of below AVL tree
           9
          /  \
         1    10
        /  \     \
       0    5     11
      /    /  \
     -1   2    6
Output : 4

Explanation: there are 4 values which are
greater than 5 in AVL tree which are 6, 9,
10 and 11.
```

Prerequisites :

- Insertion in AVL tree
- Deletion in AVL tree

1. We maintain an extra field '**desc**' for storing the number of descendant nodes for every node. Like for above example node having value 5 has a desc field value equal to 2.

2. for calculating the number of nodes which are greater than given value we simply traverse the tree. While traversing three cases can occur-

I Case- x(given value) is greater than the value of current node. So, we go to the right child of the current node.
II Case- x is lesser than the value of current node. we increase the current count by number of successors of the right child of the current node and then again add two to the current count(one for the current node and one for the right child.). In this step first, we make sure that right child exists or not. Then we move to left child of current node.
III Case-x is equal to the value of current node. In this case we add the value of **desc** field of right child of current node to current count and then add one to it (for counting right child). Also in this case we see that right child exists or not.

Calculating values of desc field

1. **Insertion** – When we insert a node we increment one to child field of every predeccesor of the new node. In the leftRotate and rightRotate functions we make appropriate changes in the value of child fields of nodes.
2. **Deletion** – When we delete a node then we decrement one from every prdeccessor node of deleted node. Again, In the leftRotate and rightRotate functions we make appropriate changes in the value of child fields of nodes.

```
 // C program to find number of elements
// greater than a given value in AVL
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int key;
    struct Node* left, *right;
    int height;
    int desc;
};

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;
}

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; // initially added at leaf
    node->desc = 0;
    return (node);
}

// A utility function to right rotate subtree
// rooted with y
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;

    // calculate the number of children of x and y
    // which are changed due to rotation.
    int val = (T2 != NULL) ? T2->desc : -1;
    y->desc = y->desc - (x->desc + 1) + (val + 1);
    x->desc = x->desc - (val + 1) + (y->desc + 1);

    return x;
}

// A utility function to left rotate subtree rooted
// with x
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;

    // calculate the number of children of x and y
    // which are changed due to rotation.
    int val = (T2 != NULL) ? T2->desc : -1;
    x->desc = x->desc - (y->desc + 1) + (val + 1);
    y->desc = y->desc - (val + 1) + (x->desc + 1);

    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);
        node->desc++;
    }

    else if (key > node->key) {
        node->right = insert(node->right, key);
        node->desc++;
    }

    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 node becomes unbalanced, 4 cases arise
```

```
    // 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);
    root->desc = root->desc - 1;
}

// 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);
    root->desc = root->desc - 1;
}

// 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;
            free(temp);

        }
        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);
```

```
            root->desc = root->desc - 1;
        }
    }

    // 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, 4 cases arise

    // 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.
void preOrder(struct Node* root)
{
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
```

```
        preOrder(root->right);
    }
}

// Returns count of
int CountGreater(struct Node* root, int x)
{
    int res = 0;

    // Search for x. While searching, keep
    // updating res if x is greater than
    // current node.
    while (root != NULL) {

        int desc = (root->right != NULL) ?
                   root->right->desc : -1;

        if (root->key > x) {
            res = res + desc + 1 + 1;
            root = root->left;
        } else if (root->key < x)
            root = root->right;
        else {
            res = res + desc + 1;
            break;
        }
    }
    return res;
}

/* Driver program to test above function*/
int main()
{
    struct Node* root = NULL;
    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);
    printf("\nNumber of elements greater than 9 are %d",
           CountGreater(root, 9));

    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);
    printf("\nNumber of elements greater than 9 are %d",
           CountGreater(root, 9));
    return 0;
}
```

Output:

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Number of elements greater than 9 are 2
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
Number of elements greater than 9 are 1
```

Time Complexity: Time complexity of CountGreater function is O(log(n)) where n is number of nodes in avl tree, as we are basically searching for the given number in avl which takes O(log(n)) time.

Source

https://www.geeksforgeeks.org/count-greater-nodes-in-avl-tree/

Chapter 34

Count inversions in an array Set 2 (Using Self-Balancing BST)

Count inversions in an array Set 2 (Using Self-Balancing BST) - GeeksforGeeks

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

```
Two elements a[i] and a[j] form an inversion if
 a[i] > a[j] and i < j. For simplicity, we may
 assume that all elements are unique.

 Example:
 Input:  arr[] = {8, 4, 2, 1}
 Output: 6
 Given array has six inversions (8,4), (4,2),
 (8,2), (8,1), (4,1), (2,1).
```

We have already discussed Naive approach and Merge Sort based approaches for counting inversions.

Time Complexity of the Naive approach is O(n^2) and that of merge sort based approach is O(n Log n). In this post one more O(n Log n) approach is discussed. The idea is to use Self-Balancing Binary Search Tree like Red-Black Tree, AVL Tree, etc and augment it so that every node also keeps track of number of nodes in right subtree.

```
1) Create an empty AVL Tree.  The tree is augmented here
   such that every node also maintains size of subtree
   rooted with this node.
```

```
2) Initialize inversion count or result as 0.

3) Iterate from 0 to n-1 and do following for every
   element in arr[i]
     a) Insert arr[i] into the AVL Tree.  The insertion
        operation also updates result.  The idea is to
        keep counting greater nodes when tree is traversed
        from root to a leaf for insertion.

4) Return result.
```

More explanation for step 3.a:
1) When we insert arr[i], elements from arr[0] to arr[i-1] are already inserted into AVL Tree. All we need to do is count these nodes.
2) For insertion into AVL Tree, we traverse tree from root to a leaf by comparing every node with arr[i[]. When arr[i[is smaller than current node, we increase inversion count by 1 plus the number of nodes in right subtree of current node. Which is basically count of greater elements on left of arr[i], i.e., inversions.

Below is C++ implementation of above idea.

```
 // An AVL Tree based C++ program to count inversion in an array
#include<bits/stdc++.h>
using namespace std;

// An AVL tree node
struct Node
{
    int key, height;
    struct Node *left, *right;
    int size; // size of the tree rooted with this Node
};

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

// A utility function to size of the tree of rooted with N
int size(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->size;
```

```
}

/* 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 = new Node;
    node->key   = key;
    node->left   = node->right  = NULL;
    node->height = node->size = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
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;

    // Update sizes
    y->size = size(y->left) + size(y->right) + 1;
    x->size = size(x->left) + size(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
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;
```

```
    // Update sizes
    x->size = size(x->left) + size(x->right) + 1;
    y->size = size(y->left) + size(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);
}

// Inserts a new key to the tree rotted with Node. Also, updates
// *result (inversion count)
struct Node* insert(struct Node* node, int key, int *result)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
    {
        node->left  = insert(node->left, key, result);

        // UPDATE COUNT OF GREATE ELEMENTS FOR KEY
        *result = *result + size(node->right) + 1;
    }
    else
        node->right = insert(node->right, key, result);

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

    /* 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;
}

// The following function returns inversion count in arr[]
int getInvCount(int arr[], int n)
{
  struct Node *root = NULL;  // Create empty AVL Tree

  int result = 0;   // Initialize result

  // Starting from first element, insert all elements one by
  // one in an AVL tree.
  for (int i=0; i<n; i++)

     // Note that address of result is passed as insert
     // operation updates result by adding count of elements
     // greater than arr[i] on left of arr[i]
     root = insert(root, arr[i], &result);

  return result;
}

// Driver program to test above
int main()
{
    int arr[] = {8, 4, 2, 1};
```

```
    int n = sizeof(arr)/sizeof(int);
    cout << "Number of inversions count are : "
         << getInvCount(arr,n);
    return 0;
}
```

Output:

```
Number of inversions count are : 6
```

Time complexity of above solution is O(n Log n) as AVL insert takes O(Log n) time.

Counting Inversions using Set in C++ STL.

We will soon be discussingBinary Indexed Treebased approach for the same.

Source

https://www.geeksforgeeks.org/count-inversions-in-an-array-set-2-using-self-balancing-bst/

Chapter 35

Count pairs from two BSTs whose sum is equal to a given value x

Count pairs from two BSTs whose sum is equal to a given value x - GeeksforGeeks

Given two BSTs containing **n1** and **n2** distinct nodes respectively. Given a value **x**. The problem is to count all pairs from both the BSTs whose sum is equal to **x**.

Examples:

```
Input : BST 1:    5
                /   \
               3     7
              / \   / \
             2  4  6   8

        BST 2:    10
                /   \
               6     15
              / \   /  \
             3  8  11  18
        x = 16

Output : 3
The pairs are:
(5, 11), (6, 10) and (8, 8)
```

Method 1: For each node value **a** in BST 1, search the value **(x − a)** in BST 2. If value found then increment the **count**. For searching a value in BST, refer this post.

Time complexity: O(n1 * h2), here n1 is number of nodes in first BST and h2 is height of second BST.

Method 2: Traverse BST 1 from smallest value to node to largest. This can be achieved with the help of iterative inorder traversal. Traverse BST 2 from largest value node to smallest. This can be achieved with the help of reverse inorder traversal. Perform these two traversals simultaneously. Sum up the corresponding node's value from both the BSTs at a particular instance of traversals. If sum == x, then increment **count**. If x > sum, then move to the inorder successor of the current node of BST 1, else move to the inorder predecessor of the current node of BST 2. Perform these operations until either of the two traversals gets completed.

C++

```
 // C++ implementation to count pairs from two
// BSTs whose sum is equal to a given  value x
#include <bits/stdc++.h>
using namespace std;

// structure of a node of BST
struct Node {
    int data;
    Node* left, *right;
};

// function to create and return a node of BST
Node* getNode(int data)
{
    // allocate space for the node
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->left = new_node->right = NULL;
}

// function to count pairs from two BSTs
// whose sum is equal to a given value x
int countPairs(Node* root1, Node* root2, int x)
{
    // if either of the tree is empty
    if (root1 == NULL || root2 == NULL)
        return 0;

    // stack 'st1' used for the inorder
    // traversal of BST 1
    // stack 'st2' used for the reverse
    // inorder traversal of BST 2
    stack<Node*> st1, st2;
```

```
Node* top1, *top2;

int count = 0;

// the loop will break when either of two
// traversals gets completed
while (1) {

    // to find next node in inorder
    // traversal of BST 1
    while (root1 != NULL) {
        st1.push(root1);
        root1 = root1->left;
    }

    // to find next node in reverse
    // inorder traversal of BST 2
    while (root2 != NULL) {
        st2.push(root2);
        root2 = root2->right;
    }

    // if either gets empty then corresponding
    // tree traversal is completed
    if (st1.empty() || st2.empty())
        break;

    top1 = st1.top();
    top2 = st2.top();

    // if the sum of the node's is equal to 'x'
    if ((top1->data + top2->data) == x) {
        // increment count
        count++;

        // pop nodes from the respective stacks
        st1.pop();
        st2.pop();

        // insert next possible node in the
        // respective stacks
        root1 = top1->right;
        root2 = top2->left;
    }

    // move to next possible node in the
    // inoder traversal of BST 1
    else if ((top1->data + top2->data) < x) {
```

```
            st1.pop();
            root1 = top1->right;
        }

        // move to next possible node in the
        // reverse inoder traversal of BST 2
        else {
            st2.pop();
            root2 = top2->left;
        }
    }

    // required count of pairs
    return count;
}

// Driver program to test above
int main()
{
    // formation of BST 1
    Node* root1 = getNode(5); /*             5        */
    root1->left = getNode(3); /*           /   \      */
    root1->right = getNode(7); /*         3     7     */
    root1->left->left = getNode(2); /*    / \   / \    */
    root1->left->right = getNode(4); /*  2  4  6  8    */
    root1->right->left = getNode(6);
    root1->right->right = getNode(8);

    // formation of BST 2
    Node* root2 = getNode(10); /*           10         */
    root2->left = getNode(6); /*           /   \        */
    root2->right = getNode(15); /*        6     15      */
    root2->left->left = getNode(3); /*    / \   /  \     */
    root2->left->right = getNode(8); /*  3  8  11  18    */
    root2->right->left = getNode(11);
    root2->right->right = getNode(18);

    int x = 16;
    cout << "Pairs = "
         << countPairs(root1, root2, x);

    return 0;
}
```

Java

```
 // Java implementation to count pairs from two
// BSTs whose sum is equal to a given  value x
```

```
import java.util.Stack;
public class GFG {

    // structure of a node of BST
    static class Node {
        int data;
        Node left, right;

        // constructor
        public Node(int data) {
            this.data = data;
            left = null;
            right = null;
        }
    }

    static Node root1;
    static Node root2;
    // function to count pairs from two BSTs
    // whose sum is equal to a given value x
    static int countPairs(Node root1, Node root2,
                                          int x)
    {
        // if either of the tree is empty
        if (root1 == null || root2 == null)
            return 0;

        // stack 'st1' used for the inorder
        // traversal of BST 1
        // stack 'st2' used for the reverse
        // inorder traversal of BST 2
        //stack<Node*> st1, st2;
        Stack<Node> st1 = new Stack<>();
        Stack<Node> st2 = new Stack<>();
        Node top1, top2;

        int count = 0;

        // the loop will break when either of two
        // traversals gets completed
        while (true) {

            // to find next node in inorder
            // traversal of BST 1
            while (root1 != null) {
                st1.push(root1);
                root1 = root1.left;
            }
```

```
    // to find next node in reverse
    // inorder traversal of BST 2
    while (root2 != null) {
        st2.push(root2);
        root2 = root2.right;
    }

    // if either gets empty then corresponding
    // tree traversal is completed
    if (st1.empty() || st2.empty())
        break;

    top1 = st1.peek();
    top2 = st2.peek();

    // if the sum of the node's is equal to 'x'
    if ((top1.data + top2.data) == x) {
        // increment count
        count++;

        // pop nodes from the respective stacks
        st1.pop();
        st2.pop();

        // insert next possible node in the
        // respective stacks
        root1 = top1.right;
        root2 = top2.left;
    }

    // move to next possible node in the
    // inoder traversal of BST 1
    else if ((top1.data + top2.data) < x) {
        st1.pop();
        root1 = top1.right;
    }

    // move to next possible node in the
    // reverse inoder traversal of BST 2
    else {
        st2.pop();
        root2 = top2.left;
    }
}

// required count of pairs
return count;
```

```
    }

    // Driver program to test above
    public static void main(String args[])
    {
        // formation of BST 1
        root1 = new Node(5);       /*             5        */
        root1.left = new Node(3); /*           /   \      */
        root1.right = new Node(7); /*         3     7     */
        root1.left.left = new Node(2); /*    / \   / \    */
        root1.left.right = new Node(4); /*  2   4 6   8    */
        root1.right.left = new Node(6);
        root1.right.right = new Node(8);

        // formation of BST 2
        root2 = new Node(10);        /*           10         */
        root2.left = new Node(6); /*           /   \        */
        root2.right = new Node(15); /*        6     15      */
        root2.left.left = new Node(3); /*    / \   /  \     */
        root2.left.right = new Node(8); /*  3  8  11  18    */
        root2.right.left = new Node(11);
        root2.right.right = new Node(18);

        int x = 16;
        System.out.println("Pairs = "
             + countPairs(root1, root2, x));
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
Pairs = 3
```

Time Complexity: O(n1 + n2)
Auxiliary Space: O(n1 + n2)

Source

https://www.geeksforgeeks.org/count-pairs-from-two-bsts-whose-sum-is-equal-to-a-given-value-x/

Chapter 36

Data Structure for a single resource reservations

Data Structure for a single resource reservations - GeeksforGeeks

Design a data structure to do reservations of future jobs on a single machine under following constraints.
1) Every job requires exactly k time units of the machine.
2) The machine can do only one job at a time.
3) Time is part of the system. Future Jobs keep coming at different times. Reservation of a future job is done only if there is no existing reservation within k time frame (after and before)
4) Whenever a job finishes (or its reservation time plus k becomes equal to current time), it is removed from system.

Example:

```
Let time taken by a job (or k) be = 4

At time 0: Reservation request for a job at time 2 in
           future comes in, reservation is done as machine
           will be available (no conflicting reservations)
Reservations {2}

At time 3: Reservation requests at times 15, 7, 20 and 3.
           Job at 7, 15 and 20 can be reserved, but at 3
           cannot be reserved as it conflicts with a
           reserved at 2.
Reservations {2, 7, 15, 20}

At time 6: Reservation requests at times 30, 17, 35 and 45
           Jobs at 30, 35 and 45 are reserved, but at 17
```

```
           cannot be reserved as it conflicts with a reserved
           at 15.
Reservations {7, 15, 30, 35, 45}.
Note that job at 2 is removed as it must be finished by 6.
```

Let us consider different data structures for this task.

One solution is to keep all future reservations sorted in array. Time complexity of checking for conflicts can be done in O(Logn) using Binary Search, but insertions and deletions take O(n) time.

Hashingcannot be used here as the search is not exact search, but a search within k time frame.

The idea is to use Binary Search Tree to maintain set of reserved jobs. For every reservation request, insert it only when there is no conflicting reservation. While inserting job, do "within k time frame check". If there is a k distant node on insertion path from root, then reject the reservation request, otherwise do the reservation.

```
 // A BST node to store future reservations
struct node
{
    int time; // reservation time
    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->time = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* BST insert to process a new reservation request at
   a given time (future time).  This function does
   reservation only if there is no existing job within
   k time frame of new job  */
struct node* insert(struct node* root, int time, int k)
{
    /* If the tree is empty, return a new node */
    if (root == NULL) return newNode(time);

    // Check if this job conflicts with existing
    // reservations
    if ((time-k < root->time) && (time+k > root->time))
        return root;
```

```
    /* Otherwise, recur down the tree */
    if (time < root->time)
        root->left  = insert(root->left, time, k);
    else
        root->right = insert(root->right, time, k);

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

Deletion of job is simple BST delete operation.

A normal BST takes O(h) time for insert and delete operations. We can use self-balancing binary search trees like AVL, Red-Black, .. to do both operations in O(Log n) time.

This question is adopted from thisMIT lecture.

This article is contributed by **Rajeev**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/data-structure-for-future-reservations-for-a-single-resource/

Chapter 37

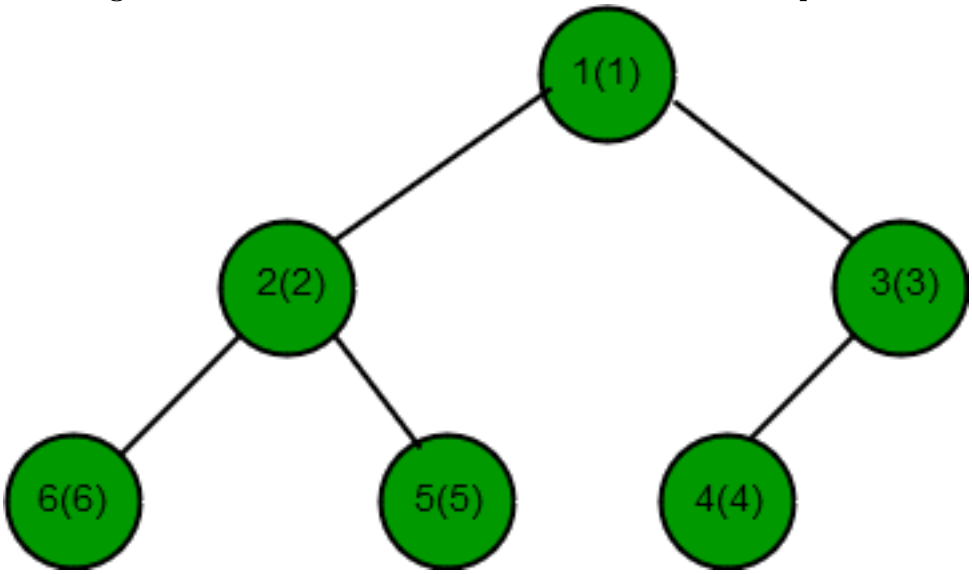
Euler Tour Subtree Sum using Segment Tree

Euler Tour Subtree Sum using Segment Tree - GeeksforGeeks

Euler tour tree (ETT) is a method for representing a rooted tree as a number sequence. When traversing the tree using Depth for search(DFS), insert each node in a vector twice, once while entered it and next after visiting all its children. This method is very useful for solving subtree problems and one such problem is **Subtree Sum.**

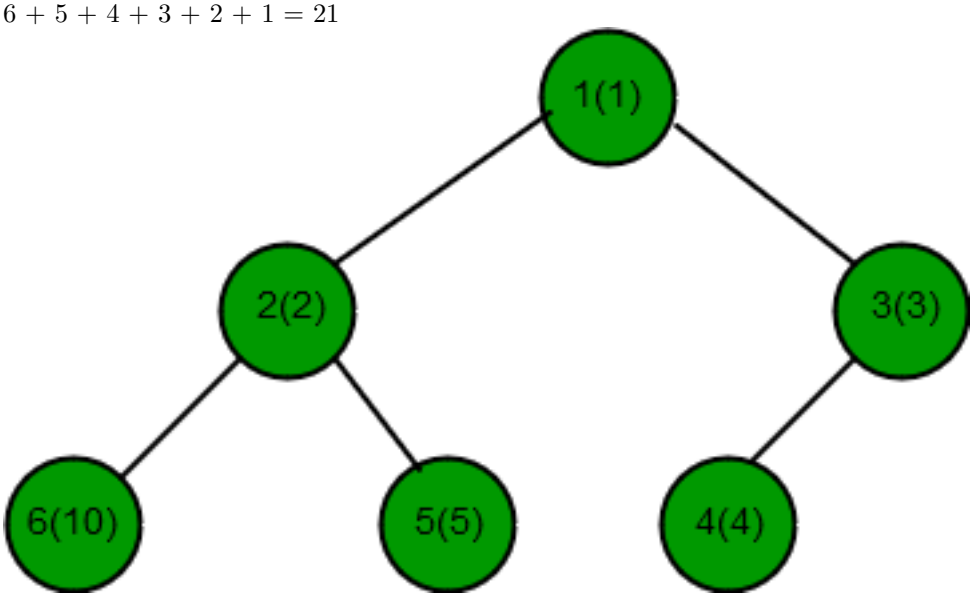
Prerequisite : Segment Tree(Sum of given range)

Naive Approach :
Consider a rooted tree with 6 vertices connected as given in the below diagram. Apply DFS for different queries.
The weight associated with each node is written inside the parenthesis.



Queries :
1. Sum of all the subtrees of node 1.
2. Update the value of node 6 to 10.
3. Sum of all the subtrees of node 2.

Answers :
1. 6 + 5 + 4 + 3 + 2 + 1 = 21

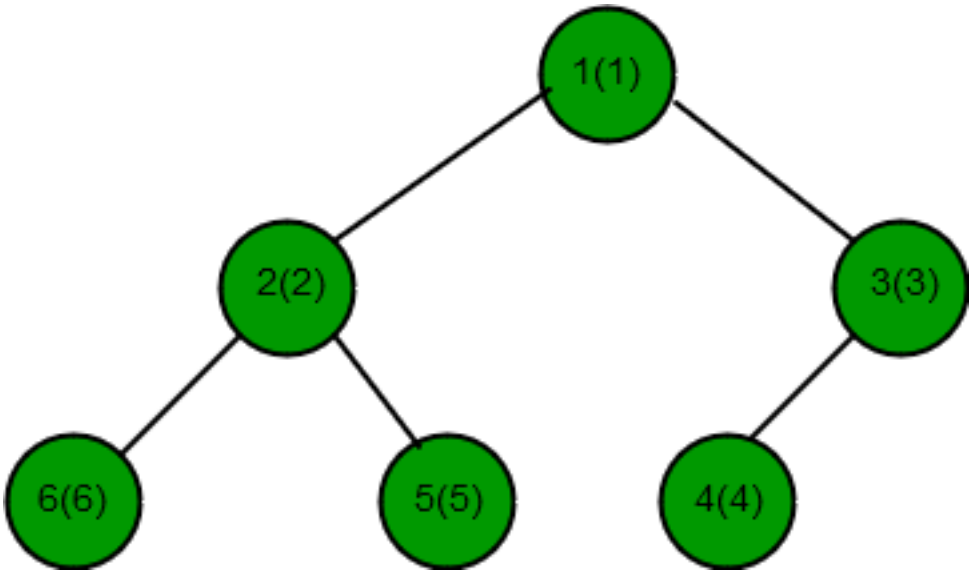


2.
3. 10 + 5 + 2 = 17

Time Complexity Analysis :
Such queries can be performed using depth for search(dfs) in **O(n)** time complexity.

Efficient Approach :
The time complexity for such queries can be reduced to **O(log(n))** time by converting the rooted tree into segment tree using Euler tour technique. So, When the number of queries are q, the total complexity becomes **O(q*5log(n))**.
Euler Tour :
In Euler tour Technique, each vertex is added to the vector twice, while descending into it and while leaving it.

Let us understand with the help of previous example :

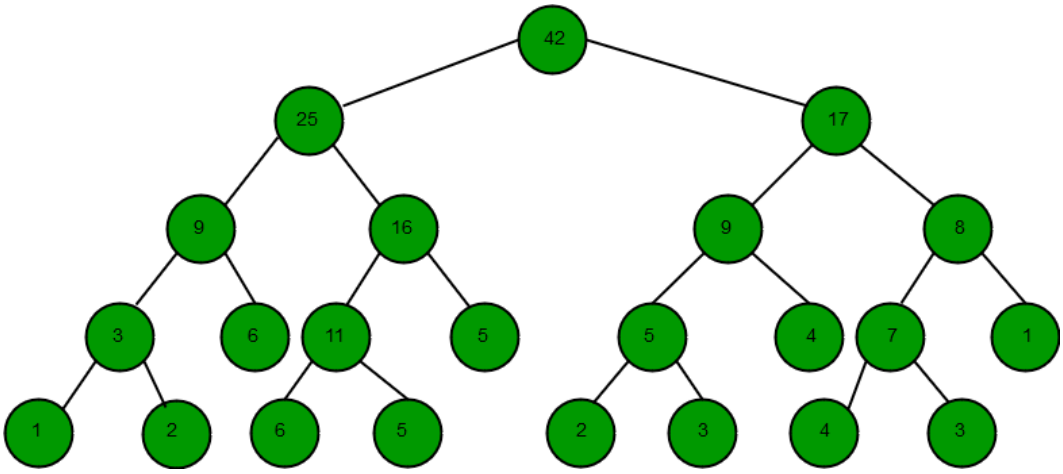


On performing depth for search(DFS) using euler tour technique on the given rooted tree, the vector so formed is :

```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}
```

```
 // DFS function to traverse the tree
int dfs(int root)
{
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i & lt; v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
}
```

Now, use vector s[ ] to **Create Segment Tree**.
Below is the representation of segment tree of vector s[ ].



For the output and update query, store the entry time and exit time(which serve as index range) for each node of the rooted tree.

```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}

Node  Entry time  Exit time
   1        1          12
   2        2          7
   3        8          11
   4        9          10
   5        5          6
   6        3          4
```

Query of type 1 :
Find the range sum on segment tree for output query where range is exit time and entry time of the rooted tree node. Deduce that the answer is always twice the expected answer because each node is added twice in segment tree. So reduce the answer by half.

Query of type 2 :
For update query, update the leaf node of segment tree at the entry time and exit time of the rooted tree node.
Below is the implementation of above approach :

```
 // C++ program for implementation of
// Euler Tour | Subtree Sum.
#include <bits/stdc++.h>
using namespace std;

vector<int> v[1001];
vector<int> s;
int seg[1001] = { 0 };
```

```
// Value/Weight of each node of tree,
// value of 0th(no such node) node is 0.
int ar[] = { 0, 1, 2, 3, 4, 5, 6 };

int vertices = 6;
int edges = 5;

// A recursive function that constructs
// Segment Tree for array ar[] = { }.
// 'pos' is index of current node
// in segment tree seg[].
int segment(int low, int high, int pos)
{
    if (high == low) {
        seg[pos] = ar[s[low]];
    }
    else {
        int mid = (low + high) / 2;
        segment(low, mid, 2 * pos);
        segment(mid + 1, high, 2 * pos + 1);
        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* Return sum of elements in range
   from index l to r . It uses the
   seg[] array created using segment()
   function. 'pos' is index of current
   node in segment tree seg[].
*/
int query(int node, int start,
          int end, int l, int r)
{
    if (r < start || end < l) {
        return 0;
    }

    if (l <= start && end <= r) {
        return seg[node];
    }

    int mid = (start + end) / 2;
    int p1 = query(2 * node, start,
                   mid, l, r);
    int p2 = query(2 * node + 1, mid + 1,
                   end, l, r);

    return (p1 + p2);
```

```
}

/* A recursive function to update the
   nodes which have the given index in
   their range. The following are
   parameters pos --> index of current
   node in segment tree seg[]. idx -->
   index of the element to be updated.
   This index is in input array.
   val --> Value to be change at node idx
*/
int update(int pos, int low, int high,
           int idx, int val)
{
    if (low == high) {
        seg[pos] = val;
    }
    else {
        int mid = (low + high) / 2;

        if (low <= idx && idx <= mid) {
            update(2 * pos, low, mid,
                   idx, val);
        }
        else {
            update(2 * pos + 1, mid + 1,
                   high, idx, val);
        }

        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* A recursive function to form array
    ar[] from a directed tree .
*/
int dfs(int root)
{
    // pushing each node in vector s
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i < v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
```

```
}

// Driver program to test above functions
int main()
{
    // Edges between the nodes
    v[1].push_back(2);
    v[1].push_back(3);
    v[2].push_back(6);
    v[2].push_back(5);
    v[3].push_back(4);

    // Calling dfs function.
    int temp = dfs(1);
    s.push_back(temp);

    // Storing entry time and exit
    // time of each node
    vector<pair<int, int> > p;

    for (int i = 0; i <= vertices; i++)
        p.push_back(make_pair(0, 0));

    for (int i = 0; i < s.size(); i++) {
        if (p[s[i]].first == 0)
            p[s[i]].first = i + 1;
        else
            p[s[i]].second = i + 1;
    }

    // Build segment tree from array ar[].
    segment(0, s.size() - 1, 1);

    // query of type 1 return the
    // sum of subtree at node 1.
    int node = 1;
    int e = p[node].first;
    int f = p[node].second;

    int ans = query(1, 1, s.size(), e, f);

    // print the sum of subtree
    cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

    // query of type 2 return update
    // the subtree at node 6.
    int val = 10;
    node = 6;
```

```
    e = p[node].first;
    f = p[node].second;
    update(1, 1, s.size(), e, val);
    update(1, 1, s.size(), f, val);

    // query of type 1 return the
    // sum of subtree at node 2.
    node = 2;

    e = p[node].first;
    f = p[node].second;

    ans = query(1, 1, s.size(), e, f);

    // print the sum of subtree
    cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

    return 0;
}
```

Output:

```
Subtree sum of node 1 is : 21
Subtree sum of node 2 is : 17
```

Time Complexity : O(q*log(n))

Source

https://www.geeksforgeeks.org/euler-tour-subtree-sum-using-segment-tree/

Chapter 38

Find a pair with given sum in BST

Find a pair with given sum in BST - GeeksforGeeks

Given a BST and a sum, find if there is a pair with given sum.

```
Input : sum = 28
        Root of below tree

Output : Pair is found (16, 12)
```

We have discussed different approaches to find a pair with given sum in below post.Find a pair with given sum in a Balanced BST

In this post, hashing based solution is discussed. We traverse binary search tree by inorder way and insert node's value into a set. Also check for any node, difference between given sum and node's value in set, if it is found then pair exists otherwise it doesn't exist.

```
 // CPP program to find a pair with
// given sum using hashing
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

Node* NewNode(int data)
{
    Node* temp = (Node*)malloc(sizeof(Node));
```

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

Node* insert(Node* root, int key)
{
    if (root == NULL)
        return NewNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

bool findpairUtil(Node* root, int sum,  unordered_set<int> &set)
{
    if (root == NULL)
        return false;

    if (findpairUtil(root->left, sum, set))
        return true;

    if (set.find(sum - root->data) != set.end()) {
        cout << "Pair is found ("
             << sum - root->data << ", "
             << root->data << ")" << endl;
        return true;
    }
    else
        set.insert(root->data);

    return findpairUtil(root->right, sum, set);
}

void findPair(Node* root, int sum)
{
    unordered_set<int> set;
    if (!findpairUtil(root, sum, set))
        cout << "Pairs do not exit" << endl;
}

// Driver code
int main()
{
    Node* root = NULL;
```

```
    root = insert(root, 15);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 12);
    root = insert(root, 16);
    root = insert(root, 25);
    root = insert(root, 10);

    int sum = 33;
    findPair(root, sum);

    return 0;
}
```

Output:

```
Pair is found (8, 25)
```

Time Complexity is O(n).

Source

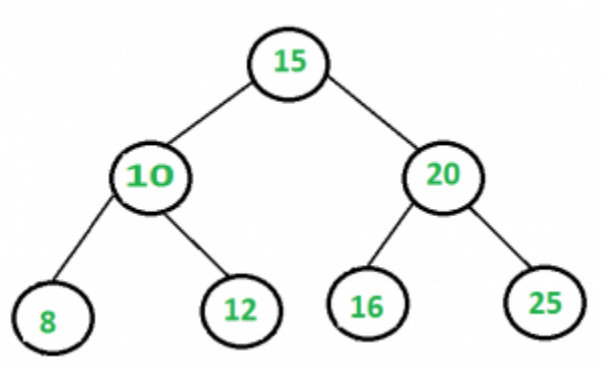
https://www.geeksforgeeks.org/find-pair-given-sum-bst/

Chapter 39

Find a pair with given sum in a Balanced BST

Find a pair with given sum in a Balanced BST - GeeksforGeeks

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is O(n) and only O(Logn) extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always O(Logn).



This problem is mainly extension of the previous post. Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be O(n^2).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in O(n) time (See this for details). This solution works in O(n) time, but requires O(n) auxiliary space.

```
// Java code to find a pair with given sum
```

```
// in a Balanced BST
import java.util.ArrayList;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree
{

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Inorder traversal of the tree
    void inorder()
    {
        inorderUtil(this.root);
    }

    // Utility function for inorder traversal of the tree
    void inorderUtil(Node node)
    {
        if(node == null)
            return;

        inorderUtil(node.left);
        System.out.print(node.data + " ");
        inorderUtil(node.right);
    }

    // This method mainly calls insertRec()
    void insert(int key) {
    root = insertRec(root, key);
    }
```

```
/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int data) {

    /* If the tree is empty, return a new node */
    if (root == null) {
        root = new Node(data);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (data < root.data)
        root.left = insertRec(root.left, data);
    else if (data > root.data)
        root.right = insertRec(root.right, data);

    return root;
}

// Method that adds values of given BST into ArrayList
// and hence returns the ArrayList
ArrayList<Integer> treeToList(Node node, ArrayList<Integer>
                                                    list)
{
    // Base Case
    if(node == null)
        return list;

    treeToList(node.left, list);
    list.add(node.data);
    treeToList(node.right, list);

    return list;
}

// method that checks if there is a pair present
boolean isPairPresent(Node node, int target)
{
    // This list a1 is passed as an argument
    // in treeToList method
    // which is later on filled by the values of BST
    ArrayList<Integer>a1 = new ArrayList<>();

    // a2 list contains all the values of BST
    // returned by treeToList method
    ArrayList<Integer> a2 = treeToList(node, a1);

    int start = 0; // Starting index of a2
```

```
        int end = a2.size() - 1; // Ending index of a2

        while(start < end)
        {

        if(a2.get(start) + a2.get(end) == target) // Target Found!
        {
            System.out.println("Pair Found: "+a2.get(start)+
                      " + "+a2.get(end)+" " + "= "+ target);
            return true;
        }

        if(a2.get(start) + a2.get(end)>target) // decrements end
        {
            end--;
        }

        if(a2.get(start) + a2.get(end)<target) // increments start
        {
            start++;
        }
    }

        System.out.println("No such values are found!");
        return false;
    }

    // Driver function
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();
        /*
                    15
                  /     \
                10      20
               / \     /  \
              8  12   16  25    */
        tree.insert(15);
        tree.insert(10);
        tree.insert(20);
        tree.insert(8);
        tree.insert(12);
        tree.insert(16);
        tree.insert(25);

        tree.isPairPresent(tree.root, 33);
        }
}
```

```
// This code is contributed by Kamal Rawal
```

Output :

```
Pair Found: 8 + 25 = 33
```

A **space optimized solution** is discussed in previous post. The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space, but it modifies the given BST.

The **solution discussed below takes O(n) time, O(Logn) space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of this for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false. Following is C++ implementation of this approach.

```
 /* In a balanced binary search tree isPairPresent two element which sums to
   a given value time O(n) space O(logn) */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* *array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
```

```
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{   return stack->top - 1 == stack->size;  }

int isEmpty(struct Stack* stack)
{   return stack->top == -1;   }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for normal inorder traversal using s1
    // done2, val2 and curr2 are used for reverse inorder traversal using s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
```

```
while (1)
{
    // Find next node in normal Inorder traversal. See following post
    // https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
    while (done1 == false)
    {
        if (curr1 != NULL)
        {
            push(s1, curr1);
            curr1 = curr1->left;
        }
        else
        {
            if (isEmpty(s1))
                done1 = 1;
            else
            {
                curr1 = pop(s1);
                val1 = curr1->val;
                curr1 = curr1->right;
                done1 = 1;
            }
        }
    }

    // Find next node in REVERSE Inorder traversal. The only
    // difference between above and below loop is, in below loop
    // right subtree is traversed before left subtree
    while (done2 == false)
    {
        if (curr2 != NULL)
        {
            push(s2, curr2);
            curr2 = curr2->right;
        }
        else
        {
            if (isEmpty(s2))
                done2 = 1;
            else
            {
                curr2 = pop(s2);
                val2 = curr2->val;
                curr2 = curr2->left;
                done2 = 1;
            }
        }
    }
```

```
        // If we find a pair, then print the pair and return. The first
        // condition makes sure that two same values are not added
        if ((val1 != val2) && (val1 + val2) == target)
        {
            printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
            return true;
        }

        // If sum of current values is smaller, then move to next node in
        // normal inorder traversal
        else if ((val1 + val2) < target)
            done1 = false;

        // If sum of current values is greater, then move to next node in
        // reverse inorder traversal
        else if ((val1 + val2) > target)
            done2 = false;

        // If any of the inorder traversals is over, then there is no pair
        // so return false
        if (val1 >= val2)
            return false;
    }
}

// A utility function to create BST node
struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left =NULL;
    return tmp;
}

// Driver program to test above functions
int main()
{
    /*
                    15
                  /     \
                10      20
               / \     /  \
              8  12   16  25    */
    struct node *root =  NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
```

```
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getchar();
    return 0;
}
```

Output:

```
 Pair Found: 8 + 25 = 33
```

This article is compiled by Kumarand reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

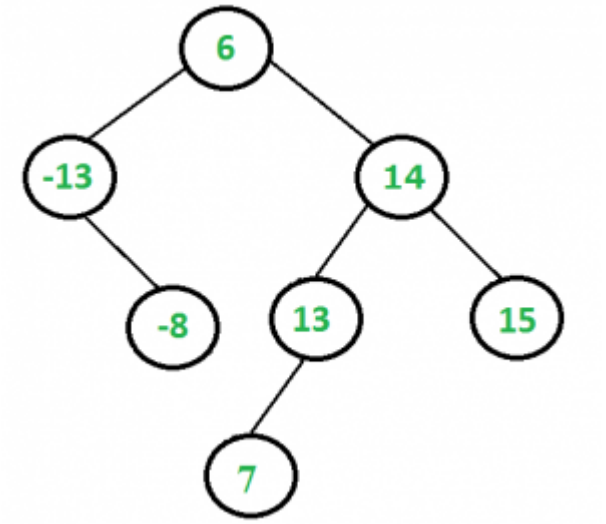
https://www.geeksforgeeks.org/find-a-pair-with-given-sum-in-bst/

Chapter 40

Find if there is a triplet in a Balanced BST that adds to zero

Three numbers in a BST that adds upto zero - GeeksforGeeks

Given a Balanced Binary Search Tree (BST), write a function isTripletPresent() that returns true if there is a triplet in given BST with sum equals to 0, otherwise returns false. Expected time complexity is O(n^2) and only O(Logn) extra space can be used. You can modify given Binary Search Tree. Note that height of a Balanced BST is always O(Logn)
For example, isTripletPresent() should return true for following BST because there is a triplet with sum 0, the triplet is {-13, 6, 7}.



The Brute Force Solution is to consider each triplet in BST and check whether the sum adds upto zero. The time complexity of this solution will be O(n^3).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data.

Once we have the Inorder traversal, we can use method 2 of thispost to find the triplet with sum equals to 0. This solution works in O(n^2) time, but requires O(n) auxiliary space.

Following is the solution that works in O(n^2) time and uses O(Logn) extra space:
1) Convert given BST to Doubly Linked List (DLL)
2) Now iterate through every node of DLL and if the key of node is negative, then find a pair in DLL with sum equal to key of current node multiplied by -1. To find the pair, we can use the approach used in hasArrayTwoCandidates() in method 1 of thispost.

```
 // A C++ program to check if there is a triplet with sum equal to 0 in
// a given BST
#include<stdio.h>

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// A function to convert given BST to Doubly Linked List. left pointer is used
// as previous pointer and right pointer is used as next pointer. The function
// sets *head to point to first and *tail to point to last node of converted DLL
void convertBSTtoDLL(node* root, node** head, node** tail)
{
    // Base case
    if (root == NULL)
        return;

    // First convert the left subtree
    if (root->left)
        convertBSTtoDLL(root->left, head, tail);

    // Then change left of current root as last node of left subtree
    root->left = *tail;

    // If tail is not NULL, then set right of tail as root, else current
    // node is head
    if (*tail)
        (*tail)->right = root;
    else
        *head = root;

    // Update tail
    *tail = root;

    // Finally, convert right subtree
```

```
    if (root->right)
        convertBSTtoDLL(root->right, head, tail);
}

// This function returns true if there is pair in DLL with sum equal
// to given sum. The algorithm is similar to hasArrayTwoCandidates()
// in method 1 of http://tinyurl.com/dy6palr
bool isPresentInDLL(node* head, node* tail, int sum)
{
    while (head != tail)
    {
        int curr = head->key + tail->key;
        if (curr == sum)
            return true;
        else if (curr > sum)
            tail = tail->left;
        else
            head = head->right;
    }
    return false;
}

// The main function that returns true if there is a 0 sum triplet in
// BST otherwise returns false
bool isTripletPresent(node *root)
{
    // Check if the given  BST is empty
    if (root == NULL)
       return false;

    // Convert given BST to doubly linked list.  head and tail store the
    // pointers to first and last nodes in DLLL
    node* head = NULL;
    node* tail = NULL;
    convertBSTtoDLL(root, &head, &tail);

    // Now iterate through every node and find if there is a pair with sum
    // equal to -1 * heaf->key where head is current node
    while ((head->right != tail) && (head->key < 0))
    {
        // If there is a pair with sum equal to  -1*head->key, then return
        // true else move forward
        if (isPresentInDLL(head->right, tail, -1*head->key))
            return true;
        else
            head = head->right;
    }
```

```
    // If we reach here, then there was no 0 sum triplet
    return false;
}

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

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

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);
    if (isTripletPresent(root))
        printf("Present");
    else
        printf("Not Present");
    return 0;
}
```

Output:

```
Present
```

Note that the above solution modifies given BST.

Time Complexity: Time taken to convert BST to DLL is O(n) and time taken to find triplet in DLL is O(n^2).

Auxiliary Space: The auxiliary space is needed only for function call stack in recursive function convertBSTtoDLL(). Since given tree is balanced (height is O(Logn)), the number of functions in call stack will never be more than O(Logn).

We can also find triplet in same time and extra space without modifying the tree. See nextpost. The code discussed there can be used to find triplet also.

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/find-if-there-is-a-triplet-in-bst-that-adds-to-0/

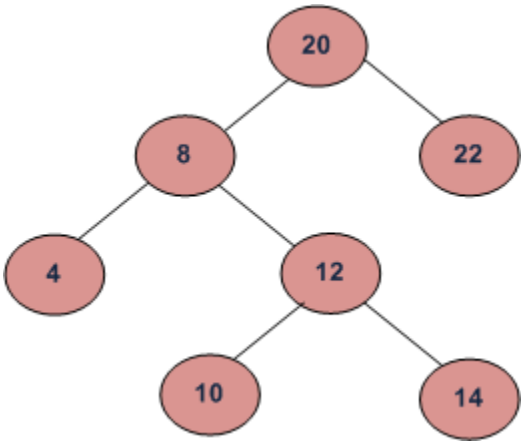
Chapter 41

Find k-th smallest element in BST (Order Statistics in BST)

Find k-th smallest element in BST (Order Statistics in BST) - GeeksforGeeks

Given root of binary search tree and K as input, find K-th smallest element in BST.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Method 1: Using Inorder Traversal.

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see this post). The idea is to keep track of popped elements which participate in the order statics. Hypothetical algorithm is provided below,

Time complexity: O(n) where n is total nodes in tree..

Algorithm:

```
/* initialization */
pCrawl = root
set initial stack element as NULL (sentinal)

/* traverse upto left extreme */
while(pCrawl is valid )
   stack.push(pCrawl)
   pCrawl = pCrawl.left

/* process other nodes */
while( pCrawl = stack.pop() is valid )
   stop if sufficient number of elements are popped.
   if( pCrawl.right is valid )
      pCrawl = pCrawl.right
      while( pCrawl is valid )
         stack.push(pCrawl)
         pCrawl = pCrawl.left
```

Implementation:

```
 #include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinal */
struct stack_t
{
    node_t*  base[ARRAY_SIZE(ele) + 1];
```

```
    int      stackIndex;
};

/* pop operation of stack */
node_t *pop(stack_t *st)
{
    node_t *ret = NULL;

    if( st && st->stackIndex > 0 )
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t *st, node_t *node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
```

```
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

node_t *k_smallest_element_inorder(stack_t *stack, node_t *root, int k)
{
```

```
    stack_t *st = stack;
    node_t *pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }

    /* pop off stack and process each node */
    while( pCrawl = pop(st) )
    {
        /* each pop operation emits one element
           in the order
        */
        if( !--k )
        {
            /* loop testing */
            st->stackIndex = 0;
            break;
        }

        /* there is right subtree */
        if( pCrawl->right )
        {
            /* push the left subtree of right subtree */
            pCrawl = pCrawl->right;
            while( pCrawl )
            {
                push(st, pCrawl);
                pCrawl = pCrawl->left;
            }

            /* pop off stack and repeat */
        }
    }

    /* node having k-th element or NULL node */
    return pCrawl;
}

/* Driver program to test above functions */
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t *kNode = NULL;
```

```
    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest elment for k = %d is %d", k, kNode->data);
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
    return 0;
}
```

Method 2: Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: O(h) where h is height of tree.

Algorithm:

```
start:
if K = root.leftElement + 1
   root node is the K th node.
   goto stop
else if K > root.leftElements
   K = K - (root.leftElements + 1)
   root = root.right
   goto start
else
   root = root.left
   goto start

stop:
```

Implementation:

```
 #include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t* left;
    node_t* right;
};

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* We are branching to left subtree
               increment node count */
            pTraverse->lCount++;
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }
```

```
    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
        new_node->lCount = 0;
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
```

```
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )
            {
                ret = pTraverse->data;
                break;
            }
            else if( k > pTraverse->lCount )
            {
                /*  There are less nodes on left subtree
                    Go to right subtree */
                k = k - (pTraverse->lCount + 1);
                pTraverse = pTraverse->right;
            }
            else
            {
                /* The node is on left subtree */
                pTraverse = pTraverse->left;
            }
        }
    }

    return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
    int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
    int i;
    node_t* root = NULL;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    /*  It should print the sorted array */
    for(i = 1; i <= ARRAY_SIZE(ele); i++)
    {
        printf("\n kth smallest elment for k = %d is %d",
                 i, k_smallest_element(root, i));
    }
```

```
    getchar();
    return 0;
}
```

Thanks to **Venki** for providing post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Dheerain Jain

Source

https://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/

Chapter 42

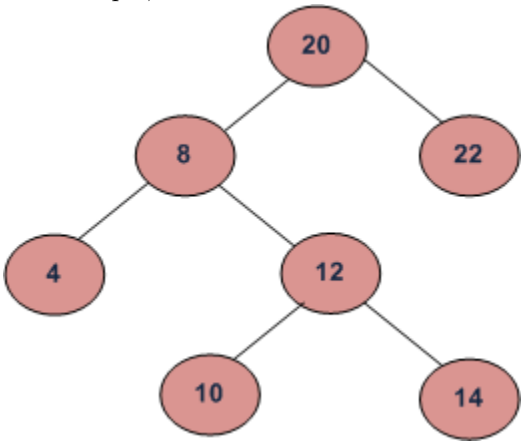
Find median of BST in O(n) time and O(1) space

Find median of BST in O(n) time and O(1) space - GeeksforGeeks

Given a Binary Search Tree, find median of it.

If no. of nodes are even: then median = ((n/2th node + (n+1)/2th node) /2
If no. of nodes are odd : then median = (n+1)th node.

For example, median of below BST is 12.



More Examples:

```
Given BST(with odd no. of nodes) is :
                  6
                /    \
               3      8
             /   \    /  \
```

```
            1     4  7    9

Inorder of Given BST will be : 1, 3, 4, 6, 7, 8, 9
So, here median will 6.

Given BST(with even no. of nodes) is :
                    6
                  /    \
                 3       8
               /   \    /
              1     4  7

Inorder of Given BST will be : 1, 3, 4, 6, 7, 8
So, here median will  (4+6)/2 = 5.
```

Asked in : Google

To find the median, we need to find the Inorder of the BST because its Inorder will be in sorted order and then find the median i.e.

The idea is based on K'th smallest element in BST using O(1) Extra Space

The task is very simple if we are allowed to use extra space but Inorder traversal using recursion and stack both uses Space which is not allowed here. So, the solution is to do Morris Inorder traversal as it doesn't require any extra space.

```
Implementation:
1- Count the no. of nodes in the given BST
   using Morris Inorder Traversal.
2- Then Perform Morris Inorder traversal one
   more time by counting nodes and by checking if
   count is equal to the median point.
   To consider even no. of nodes an extra pointer
   pointing to the previous node is used.

 /* C++ program to find the median of BST in O(n)
   time and O(1) space*/
#include<iostream>
using namespace std;

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

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

/* 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->data)
        node->left  = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);

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

/* Function to count nodes in a  binary search tree
   using Morris Inorder traversal*/
int counNodes(struct Node *root)
{
    struct Node *current, *pre;

    // Initialise count of nodes as 0
    int count = 0;

    if (root == NULL)
        return count;

    current = root;
    while (current != NULL)
    {
        if (current->left == NULL)
        {
            // Count node if its left is NULL
            count++;

            // Move to its right
            current = current->right;
```

```
        }
        else
        {
            /* Find the inorder predecessor of current */
            pre = current->left;

            while (pre->right != NULL &&
                   pre->right != current)
                pre = pre->right;

            /* Make current as right child of its
               inorder predecessor */
            if(pre->right == NULL)
            {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in if part to
               restore the original tree i.e., fix
               the right child of predecssor */
            else
            {
                pre->right = NULL;

                // Increment count if the current
                // node is to be visited
                count++;
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */

    return count;
}


/* Function to find median in O(n) time and O(1) space
   using Morris Inorder traversal*/
int findMedian(struct Node *root)
{
   if (root == NULL)
        return 0;

    int count = counNodes(root);
    int currCount = 0;
    struct Node *current = root, *pre, *prev;
```

```
while (current != NULL)
{
    if (current->left == NULL)
    {
        // count current node
        currCount++;

        // check if current node is the median
        // Odd case
        if (count % 2 != 0 && currCount == (count+1)/2)
            return prev->data;

        // Even case
        else if (count % 2 == 0 && currCount == (count/2)+1)
            return (prev->data + current->data)/2;

        // Update prev for even no. of nodes
        prev = current;

        //Move to the right
        current = current->right;
    }
    else
    {
        /* Find the inorder predecessor of current */
        pre = current->left;
        while (pre->right != NULL && pre->right != current)
            pre = pre->right;

        /* Make current as right child of its inorder predecessor */
        if (pre->right == NULL)
        {
            pre->right = current;
            current = current->left;
        }

        /* Revert the changes made in if part to restore the original
          tree i.e., fix the right child of predecssor */
        else
        {
            pre->right = NULL;

            prev = pre;

            // Count current node
            currCount++;

            // Check if the current node is the median
```

```
                if (count % 2 != 0 && currCount == (count+1)/2 )
                    return current->data;

                else if (count%2==0 && currCount == (count/2)+1)
                    return (prev->data+current->data)/2;

                // update prev node for the case of even
                // no. of nodes
                prev = current;
                current = current->right;

            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */
}

/* 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);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    cout << "\nMedian of BST is "
         << findMedian(root);
    return 0;
}
```

Output:

```
Median of BST is 50
```

Reference:
https://www.careercup.com/question?id=4882624968392704

Source

https://www.geeksforgeeks.org/find-median-bst-time-o1-space/

Chapter 43

Find pairs with given sum such that pair elements lie in different BSTs

Find pairs with given sum such that pair elements lie in different BSTs - GeeksforGeeks

Given two Binary Search Trees (BST) and a given sum. The task is to find pairs with given sum such that each pair elements must lie in different BST.

Examples:

```
Input : sum = 10
     8                     5
   /   \                 /   \
  3     10              2    18
 /  \      \           /   \
1    6      14       1     3
    / \     /               \
   5   7   13               4
Output : (5,5), (6,4), (7,3), (8,2)
In above pairs first element lies in first
BST and second element lies in second BST
```

A **simple solution** for this problem is to store Inorder traversal of one tree in auxiliary array then pick element one by one from the array and find its pair in other tree for given sum. Time complexity for this approach will be $O(n^2)$ if total nodes in both the trees are equal.

An **efficient solution** for this solution is to store Inorder traversals of both BSTs in two different auxiliary arrays **vect1**[] and vect2[]. Now we follow **method1** of this article. Since Inorder traversal of BST is always gives sorted sequence, we don not need to sort our arrays.

- Take iterator **left** and points it to the left corner vect1[].
- Take iterator **right** and points it to the right corner vect2[].
- Now if **vect11[left] + vect2[right] < sum** then move left iterator in vect1[] in forward direction i.e; **left++**.
- Now if **vect1[left] + vect2[right] > sum** then move right iterator in vect[] in backward direction i.e; **right–**.

Below is implementation of above idea.

```
 // C++ program to find pairs with given sum such
// that one element of pair exists in one BST and
// other in other BST.
#include<bits/stdc++.h>
using namespace std;

// A binary Tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

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

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

// store storeInorder traversal in auxiliary array
void storeInorder(Node *ptr, vector<int> &vect)
```

```
{
    if (ptr==NULL)
        return;
    storeInorder(ptr->left, vect);
    vect.push_back(ptr->data);
    storeInorder(ptr->right, vect);
}

// Function to find pair for given sum in different bst
// vect1[]  --> stores storeInorder traversal of first bst
// vect2[]  --> stores storeInorder traversal of second bst
void pairSumUtil(vector<int> &vect1, vector<int> &vect2,
                                                    int sum)
{
    // Initialize two indexes to two different corners
    // of two vectors.
    int left = 0;
    int right = vect2.size() - 1;

    // find pair by moving two corners.
    while (left < vect1.size() && right >= 0)
    {
        // If we found a pair
        if (vect1[left] + vect2[right] == sum)
        {
            cout << "(" << vect1[left] << ", "
                 << vect2[right] << "), ";
            left++;
            right--;
        }

        // If sum is more, move to higher value in
        // first vector.
        else if (vect1[left] + vect2[right] < sum)
            left++;

        // If sum is less, move to lower value in
        // second vector.
        else
            right--;
    }
}

// Prints all pairs with given "sum" such that one
// element of pair is in tree with root1 and other
// node is in tree with root2.
void pairSum(Node *root1, Node *root2, int sum)
{
```

```
    // Store inorder traversals of two BSTs in two
    // vectors.
    vector<int> vect1, vect2;
    storeInorder(root1, vect1);
    storeInorder(root2, vect2);

    // Now the problem reduces to finding a pair
    // with given sum such that one element is in
    // vect1 and other is in vect2.
    pairSumUtil(vect1, vect2, sum);
}

// Driver program to run the case
int main()
{
    // first BST
    struct Node* root1 = NULL;
    root1 = insert(root1, 8);
    root1 = insert(root1, 10);
    root1 = insert(root1, 3);
    root1 = insert(root1, 6);
    root1 = insert(root1, 1);
    root1 = insert(root1, 5);
    root1 = insert(root1, 7);
    root1 = insert(root1, 14);
    root1 = insert(root1, 13);

    // second BST
    struct Node* root2 = NULL;
    root2 = insert(root2, 5);
    root2 = insert(root2, 18);
    root2 = insert(root2, 2);
    root2 = insert(root2, 1);
    root2 = insert(root2, 3);
    root2 = insert(root2, 4);

    int sum = 10;
    pairSum(root1, root2, sum);

    return 0;
}
```

Output:

```
(5,5),(6,4),(7,3),(8,2)
```

Time complexity : O(n)

Auxiliary space : O(n)

We have another **space optimized** approach to solve this problem. The idea is to convert bst into doubly linked list and apply above method for doubly linked list.See this article.

Time complexity : O(n)
Auxiliary Space : O(1)

Source

https://www.geeksforgeeks.org/find-pairs-with-given-sum-such-that-pair-elements-lie-in-different-bsts/

Chapter 44

Find postorder traversal of BST from preorder traversal

Find postorder traversal of BST from preorder traversal - GeeksforGeeks

Given an array representing preorder traversal of BST, print its postorder traversal.

Examples:

```
Input : 40 30 35 80 100
Output : 35 30 100 80 40

Input : 40 30 32 35 80 90 100 120
Output : 35 32 30 120 100 90 80 40
```

Prerequisite: Construct BST from given preorder traversal

Simple Approach: A simple solution is to first construct BST from given preorder traversal as described in this post. After constructing tree, perform postorder traversal on it.

Efficient Approach: An efficient approach is to find postorder traversal without constructing the tree. The idea is to traverse the given preorder array and maintain a range in which current element should lie. This is to ensure that BST property is always satisfied. Initially the range is set to {minval = INT_MIN, maxval = INT_MAX}. In preorder traversal, the first element is always the root and it will certainly lie in initial range. So store the first element of the preorder array. In postorder traversal, first left and right subtrees are printed and then root data is printed. So first recursive call for left and right subtrees are performed and then the value of root is printed. For left subtree range is updated to {minval, root->data} and for right subtree range is updated to {root->data, maxval}. If current preorder array element does not lie in the range specified for it, then it does not belong to a current subtree, return from recursive calls until correct position of that element is not found.

Below is implementation of above approach:

```
 // C++ program for finding postorder
// traversal of BST from preorder traversal
#include <bits/stdc++.h>
using namespace std;

// Function to find postorder traversal from
// preorder traversal.
void findPostOrderUtil(int pre[], int n, int minval,
                       int maxval, int& preIndex)
{

    // If entire preorder array is traversed then
    // return as no more element is left to be
    // added to post order array.
    if (preIndex == n)
        return;

    // If array element does not lie in range specified,
    // then it is not part of current subtree.
    if (pre[preIndex] < minval || pre[preIndex] > maxval) {
        return;
    }

    // Store current value, to be printed later, after
    // printing left and right subtrees. Increment
    // preIndex to find left and right subtrees,
    // and pass this updated value to recursive calls.
    int val = pre[preIndex];
    preIndex++;

    // All elements with value between minval and val
    // lie in left subtree.
    findPostOrderUtil(pre, n, minval, val, preIndex);

    // All elements with value between val and maxval
    // lie in right subtree.
    findPostOrderUtil(pre, n, val, maxval, preIndex);

    cout << val << " ";
}

// Function to find postorder traversal.
void findPostOrder(int pre[], int n)
{

    // To store index of element to be
    // traversed next in preorder array.
    // This is passed by reference to
```

```
    // utility function.
    int preIndex = 0;

    findPostOrderUtil(pre, n, INT_MIN, INT_MAX, preIndex);
}

// Driver code
int main()
{
    int pre[] = { 40, 30, 35, 80, 100 };

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

    // Calling function
    findPostOrder(pre, n);
    return 0;
}
```

Output:

```
35 30 100 80 40
```

Time Complexity: O(N), where N is the number of nodes.
Auxiliary Space: O(N) (Function call stack size)

Source

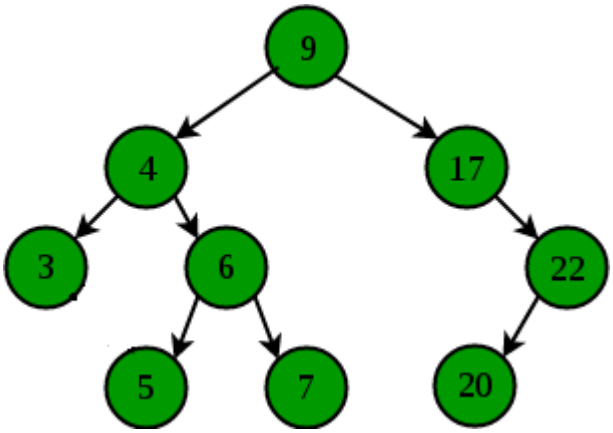
https://www.geeksforgeeks.org/find-postorder-traversal-of-bst-from-preorder-traversal/

Chapter 45

Find the closest element in Binary Search Tree

Find the closest element in Binary Search Tree - GeeksforGeeks

Given a binary search tree and a target node K. The task is to find the node with minimum absolute difference with given target value K.



Examples:

```
// For above binary search tree
Input  :  k = 4
Output :  4

Input  :  k = 18
Output :  17

Input  :  k = 12
```

```
Output :  9
```

A **simple solution** for this problem is to store Inorder traversal of given binary search tree in an auxiliary array and then by taking absolute difference of each element find the node having minimum absolute difference with given target value K in linear time.

An **efficient solution** for this problem is to take advantage of characteristics of BST. Here is the algorithm to solve this problem :

- If target value K is present in given BST, then it's the node having minimum absolute difference.
- If target value K is less than the value of current node then move to the left child.
- If target value K is greater than the value of current node then move to the right child.

```
17
```

Time complexity : O(h) where h is height of given Binary Search Tree.

Reference :
http://stackoverflow.com/questions/6209325/how-to-find-the-closest-element-to-a-given-key-value-in-a-binary-sea

Source

https://www.geeksforgeeks.org/find-closest-element-binary-search-tree/

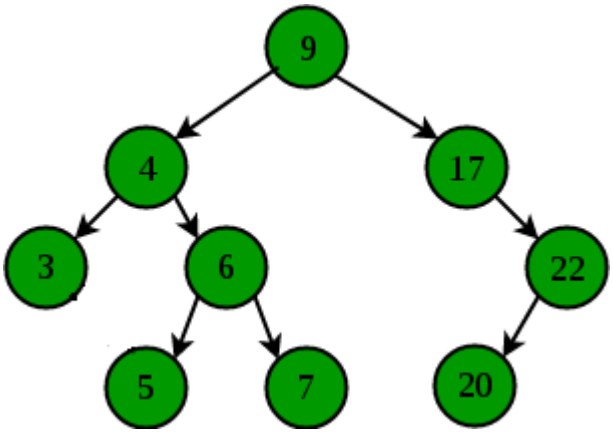
Chapter 46

Find the closest element in Binary Search Tree Space Efficient Method

Find the closest element in Binary Search Tree Space Efficient Method - GeeksforGeeks

Given a binary search tree and a target node K. The task is to find the node with the minimum absolute difference with given target value K.

NOTE: The approach used should have constant extra space consumed O(1). No recursion or stack/queue like containers should be used.



Examples:

```
Input:  k = 4
Output:  4
```

```
Input:  k = 18
Output:  17
```

A simple solution mentioned in this post uses recursion to get the closest element to a key in Binary search tree. The method used in the above mentioned post consumes O(n) extra space due to recursion.

Now we can easily modify the above mentioned approach using Morris traversal which is a space efficient approach to do inorder tree traversal without using recursion or stack/queue in constant space O(1).

Morris traversal is based on Threaded Binary trees which makes use of NULL pointers in a tree to make them point to some successor or predecessor nodes. As in a binary tree with n nodes, n+1 NULL pointers waste memory.

In the algorithm mentioned below we simply do inorder tree traversal and while doing inorder tree traversal using Morris Traversal we check for differences between the node's data and the key and maintain two variables 'diff' and 'closest' which are updated when we find a closer node to the key. When we are done with the complete inorder tree traversal we have the closest node.

Algorithm :

```
1) Initialize Current as root.

2) Initialize a variable diff as INT_MAX.

3)initialize a variable closest(pointer to node) which
  will be returned.

4) While current is not NULL:

  4.1) If the current has no left child:
     a) If the absolute difference between current's data
        and the key is smaller than diff:
       1) Set diff as the absolute difference between the
          current node and the key.
       2) Set closest as the current node.

     b)Otherwise, Move to the right child of current.

  4.2) Else, here we have 2 cases:

   a) Find the inorder predecessor of the current node.
      Inorder predecessor is the rightmost node
      in the left subtree or left child itself.

   b) If the right child of the inorder predecessor is NULL:
      1) Set current as the right child of its inorder
```

```
         predecessor(Making threads between nodes).
      2) Move current node to its left child.

   c) Else, if the threaded link between the current node
      and it's inorder predecessor already exists :

      1) Set right pointer of the inorder predecessor node as NULL.

      2) If the absolute difference between current's data and
         the key is smaller than diff:
        a) Set diff variable as the absolute difference between
           the current node and the key.
        b) Set closest as the current node.

      3) Move current to its right child.

5)By the time we have traversed the whole tree, we have the
  closest node, so we simply return closest.
```

Below is the implementation of above approach:

```
 // CPP program to find closest value in
// a Binary Search Tree.
#include <iostream>
#include <limits.h>
using namespace std;

// Tree Node
struct Node {
    int data;
    Node *left, *right;
};

// Utility function to create a new Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to find the Node closest to the
// given key in BST using Morris Traversal
Node* closestNodeUsingMorrisTraversal(Node* root,
                                      int key)
{
    int diff = INT_MAX;
```

```
Node* curr = root;
Node* closest;

while (curr) {
    if (curr->left == NULL) {

        // updating diff if the current diff is
        // smaller than prev difference
        if (diff > abs(curr->data - key)) {
            diff = abs(curr->data - key);
            closest = curr;
        }

        curr = curr->right;
    }

    else {

        // finding the inorder predecessor
        Node* pre = curr->left;
        while (pre->right != NULL &&
               pre->right != curr)
            pre = pre->right;

        if (pre->right == NULL) {
            pre->right = curr;
            curr = curr->left;
        }

        // threaded link between curr and
        // its predecessor already exists
        else {
            pre->right = NULL;

            // if a closer Node found, then update
            // the diff and set closest to current
            if (diff > abs(curr->data - key)) {
                diff = abs(curr->data - key);
                closest = curr;
            }

            // moving to the right child
            curr = curr->right;
        }
    }
}

return closest;
```

```
}

// Driver Code
int main()
{
    /* Constructed binary tree is
          5
        /   \
       3     9
     /  \   /  \
    1    2  8    12 */
    Node* root = newNode(5);
    root->left = newNode(3);
    root->right = newNode(9);
    root->left->left = newNode(1);
    root->left->right = newNode(2);
    root->right->left = newNode(8);
    root->right->right = newNode(12);

    cout << closestNodeUsingMorrisTraversal(root, 10)->data;

    return 0;
}
```

Output:

```
9
```

Time Complexity: O(n)
Auxillary Space : O(1)

Source

https://www.geeksforgeeks.org/find-the-closest-element-in-binary-search-tree-space-efficient-method/

Chapter 47

Find the largest BST subtree in a given Binary Tree Set 1

Find the largest BST subtree in a given Binary Tree Set 1 - GeeksforGeeks

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

```
Input:
      5
    /  \
   2    4
 /  \
1    3

Output: 3
The following subtree is the maximum size BST subtree
   2
 /  \
1    3


Input:
       50
     /    \
  30       60
 /  \     /  \
5   20   45    70
             /  \
```

```
           65    80
Output: 5
The following subtree is the maximum size BST subtree
      60
     /  \
   45    70
        /  \
      65    80
```

Method 1 (Simple but inefficient)
Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.

```
 /*
  See https://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/ for implementa

  See Method 3 of https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not
  implementation of isBST()

  max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
   if (isBST(root))
     return size(root);
   else
    return max(largestBST(root->left), largestBST(root->right));
}
```

Time Complexity: The worst case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

Method 2 (Tricky and Efficient)
In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.
1) Whether the subtree itself is BST or not (In the following code, is_bst_ref is used for this purpose)
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.

3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)

max_ref is used for passing the maximum value up the tree and min_ptr is used for passing minimum value up the tree.

C

```
 #include <stdio.h>
#include <stdlib.h>
#include <limits.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;
};

/* 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);
}

int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                             int *max_size_ref, bool *is_bst_ref);

/* Returns size of the largest BST subtree in a Binary Tree
  (efficient version). */
int largestBST(struct node* node)
{
  // Set the initial values for calling largestBSTUtil()
  int min = INT_MAX;  // For minimum value in right subtree
  int max = INT_MIN;  // For maximum value in left subtree

  int max_size = 0;  // For size of the largest BST
  bool is_bst = 0;

  largestBSTUtil(node, &min, &max, &max_size, &is_bst);
```

```
  return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree.   Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                           int *max_size_ref, bool *is_bst_ref)
{

  /* Base Case */
  if (node == NULL)
  {
     *is_bst_ref = 1; // An empty tree is BST
     return 0;    // Size of the BST is 0
  }

  int min = INT_MAX;

  /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
  bool left_flag = false;

  /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
  bool right_flag = false;

  int ls, rs; // To store sizes of left and right subtrees

  /* Following tasks are done by recursive call for left subtree
    a) Get the maximum value in left subtree (Stored in *max_ref)
    b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
    c) Get the size of maximum size BST in left subtree (updates *max_size) */
  *max_ref = INT_MIN;
  ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data > *max_ref)
     left_flag = true;

  /* Before updating *min_ref, store the min value in left subtree. So that we
     have the correct minimum value for this subtree */
  min = *min_ref;

  /* The following recursive call does similar (similar to left subtree)
    task for right subtree */
  *min_ref =  INT_MAX;
  rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data < *min_ref)
```

```
    right_flag = true;

  // Update min and max values for the parent recursive calls
  if (min < *min_ref)
     *min_ref = min;
  if (node->data < *min_ref) // For leaf nodes
     *min_ref = node->data;
  if (node->data > *max_ref)
     *max_ref = node->data;

  /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
  if(left_flag && right_flag)
  {
     if (ls + rs + 1 > *max_size_ref)
         *max_size_ref = ls + rs + 1;
     return ls + rs + 1;
  }
  else
  {
    //Since this subtree is not BST, set is_bst flag for parent calls
     *is_bst_ref = 0;
     return 0;
  }
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
          50
       /      \
     10        60
    /  \       /  \
   5   20    55    70
            /     /  \
          45     65    80
  */

  struct node *root = newNode(50);
  root->left        = newNode(10);
  root->right       = newNode(60);
  root->left->left  = newNode(5);
  root->left->right = newNode(20);
  root->right->left  = newNode(55);
  root->right->left->left  = newNode(45);
  root->right->right = newNode(70);
```

```
  root->right->right->left = newNode(65);
  root->right->right->right = newNode(80);

  /* The complete tree is not BST as 45 is in right subtree of 50.
     The following subtree is the largest BST
        60
      /  \
    55    70
   /     /  \
 45     65    80
  */
  printf(" Size of the largest BST is %d", largestBST(root));

  getchar();
  return 0;
}
```

Java

```
 // Java program to find largest BST subtree in given Binary Tree

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class Value {

    int max_size = 0; // for size of largest BST
    boolean is_bst = false;
    int min = Integer.MAX_VALUE;  // For minimum value in right subtree
    int max = Integer.MIN_VALUE;  // For maximum value in left subtree

}

class BinaryTree {

    static Node root;
    Value val = new Value();

    /* Returns size of the largest BST subtree in a Binary Tree
     (efficient version). */
```

```
int largestBST(Node node) {

    largestBSTUtil(node, val, val, val, val);

    return val.max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
 subtree.   Also, if the tree rooted with node is non-empty and a BST,
 then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(Node node, Value min_ref, Value max_ref,
        Value max_size_ref, Value is_bst_ref) {

    /* Base Case */
    if (node == null) {
        is_bst_ref.is_bst = true; // An empty tree is BST
        return 0;    // Size of the BST is 0
    }

    int min = Integer.MAX_VALUE;

    /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
    boolean left_flag = false;

    /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
    boolean right_flag = false;

    int ls, rs; // To store sizes of left and right subtrees

    /* Following tasks are done by recursive call for left subtree
     a) Get the maximum value in left subtree (Stored in *max_ref)
     b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
     c) Get the size of maximum size BST in left subtree (updates *max_size) */
    max_ref.max = Integer.MIN_VALUE;
    ls = largestBSTUtil(node.left, min_ref, max_ref, max_size_ref, is_bst_ref);
    if (is_bst_ref.is_bst == true && node.data > max_ref.max) {
        left_flag = true;
    }

    /* Before updating *min_ref, store the min value in left subtree. So that we
     have the correct minimum value for this subtree */
    min = min_ref.min;

    /* The following recursive call does similar (similar to left subtree)
     task for right subtree */
    min_ref.min = Integer.MAX_VALUE;
```

```
    rs = largestBSTUtil(node.right, min_ref, max_ref, max_size_ref, is_bst_ref);
    if (is_bst_ref.is_bst == true && node.data < min_ref.min) {
        right_flag = true;
    }

    // Update min and max values for the parent recursive calls
    if (min < min_ref.min) {
        min_ref.min = min;
    }
    if (node.data < min_ref.min) // For leaf nodes
    {
        min_ref.min = node.data;
    }
    if (node.data > max_ref.max) {
        max_ref.max = node.data;
    }

    /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
    if (left_flag && right_flag) {
        if (ls + rs + 1 > max_size_ref.max_size) {
            max_size_ref.max_size = ls + rs + 1;
        }
        return ls + rs + 1;
    } else {
        //Since this subtree is not BST, set is_bst flag for parent calls
        is_bst_ref.is_bst = false;
        return 0;
    }
}

public static void main(String[] args) {
    /* Let us construct the following Tree
            50
         /      \
        10        60
       /  \       /  \
      5   20    55    70
     /     /  \
    45   65    80
     */

    BinaryTree tree = new BinaryTree();
    tree.root = new Node(50);
    tree.root.left = new Node(10);
    tree.root.right = new Node(60);
    tree.root.left.left = new Node(5);
```

```
        tree.root.left.right = new Node(20);
        tree.root.right.left = new Node(55);
        tree.root.right.left.left = new Node(45);
        tree.root.right.right = new Node(70);
        tree.root.right.right.left = new Node(65);
        tree.root.right.right.right = new Node(80);

        /* The complete tree is not BST as 45 is in right subtree of 50.
         The following subtree is the largest BST
             60
           /  \
          55    70
          /     /  \
        45     65   80
        */
        System.out.println("Size of largest BST is " + tree.largestBST(root));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

Largest BST in a Binary Tree Set 2

Source

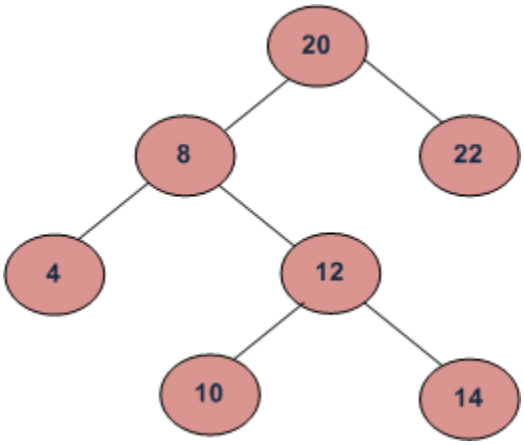
https://www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/

Chapter 48

Find the node with minimum value in a Binary Search Tree

Find the node with minimum value in a Binary Search Tree - GeeksforGeeks

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

C

```
 #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;
};

/* 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);
}

/* Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
struct node* insert(struct node* node, int data)
{
  /* 1. If the tree is empty, return a new,
      single node */
  if (node == NULL)
    return(newNode(data));
  else
  {
    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
        node->left  = insert(node->left, data);
    else
        node->right = insert(node->right, data);

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

/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
```

```
to be searched. */
int minValue(struct node* node) {
  struct node* current = node;

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

/* Driver program to test sameTree function*/
int main()
{
  struct node* root = NULL;
  root = insert(root, 4);
  insert(root, 2);
  insert(root, 1);
  insert(root, 3);
  insert(root, 6);
  insert(root, 5);

  printf("\n Minimum value in BST is %d", minValue(root));
  getchar();
  return 0;
}
```

Java

```
 // Java program to find minimum value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
```

```
 inserts a new node with the given number in
 the correct place in the tree. Returns the new
 root pointer which the caller should then use
 (the standard trick to avoid using reference
 parameters). */
Node insert(Node node, int data) {

    /* 1. If the tree is empty, return a new,
     single node */
    if (node == null) {
        return (new Node(data));
    } else {

        /* 2. Otherwise, recur down the tree */
        if (data <= node.data) {
            node.left = insert(node.left, data);
        } else {
            node.right = insert(node.right, data);
        }

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

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

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }
    return (current.data);
}

// Driver program to test above functions
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    Node root = null;
    root = tree.insert(root, 4);
    tree.insert(root, 2);
    tree.insert(root, 1);
    tree.insert(root, 3);
    tree.insert(root, 6);
```

```
        tree.insert(root, 5);

        System.out.println("The minimum value of BST is " + tree.minvalue(root));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to find the node with minimum value in bst

# A binary tree node
class Node:

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

""" Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). """
def insert(node, data):

    # 1. If the tree is empty, return a new,
    # single node
    if node is None:
        return (Node(data))

    else:
        # 2. Otherwise, recur down the tree
        if data <= node.data:
            node.left = insert(node.left, data)
        else:
            node.right = insert(node.right, data)

        # Return the (unchanged) node pointer
        return node

""" Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. """
```

```
def minValue(node):
    current = node

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

    return current.data

# Driver program
root = None
root = insert(root,4)
insert(root,2)
insert(root,1)
insert(root,3)
insert(root,6)
insert(root,5)

print "\nMinimum value in BST is %d"  %(minValue(root))

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

Time Complexity: O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

References:
http://cslibrary.stanford.edu/110/BinaryTrees.html

Source

https://www.geeksforgeeks.org/find-the-minimum-element-in-a-binary-search-tree/

Chapter 49

Floor and Ceil from a BST

Floor and Ceil from a BST - GeeksforGeeks

There are numerous applications we need to find floor (ceil) value of a key in a binary search tree or sorted array. For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

A) Root data is equal to key. We are done, root data is ceil value.

B) Root data $<$ key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.

C) Root data $>$ key value, the ceil value *may be* in left subtree. We may find a node with is larger data than key value in left subtree, if not the root itself will be ceil node.

Here is the code for ceil value.

C

```
 // Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* 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;
    return(node);
}

// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(node *root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
    return (ceil >= input) ? ceil : root->key;
}

// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d  %d\n", i, Ceil(root, i));
```

```
    return 0;
}
```

Java

```
 // Java program to find ceil of a given value in BST

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node root;

    // Function to find ceil of a given input in BST. If input is more
    // than the max key in BST, return -1
    int Ceil(Node node, int input) {

        // Base case
        if (node == null) {
            return -1;
        }

        // We found equal key
        if (node.data == input) {
            return node.data;
        }

        // If root's key is smaller, ceil must be in right subtree
        if (node.data < input) {
            return Ceil(node.right, input);
        }

        // Else, either left subtree or root has the ceil value
        int ceil = Ceil(node.left, input);
        return (ceil >= input) ? ceil : node.data;
    }

    // Driver program to test the above functions
    public static void main(String[] args) {
```

```
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(8);
        tree.root.left = new Node(4);
        tree.root.right = new Node(12);
        tree.root.left.left = new Node(2);
        tree.root.left.right = new Node(6);
        tree.root.right.left = new Node(10);
        tree.root.right.right = new Node(14);
        for (int i = 0; i < 16; i++) {
            System.out.println(i + " " + tree.Ceil(root, i));
        }
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to find ceil of a given value in BST

# A Binary tree node
class Node:

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

# Function to find ceil of a given input in BST. If input
# is more than the max key in BST, return -1
def ceil(root, inp):

    # Base Case
    if root == None:
        return -1

    # We found equal key
    if root.key == inp :
        return root.key

    # If root's key is smaller, ceil must be in right subtree
    if root.key < inp:
        return ceil(root.right, inp)

    # Else, either left subtre or root has the ceil value
    val = ceil(root.left, inp)
    return val if val >= inp else root.key
```

```
# Driver program to test above function
root = Node(8)

root.left = Node(4)
root.right = Node(12)

root.left.left = Node(2)
root.left.right = Node(6)

root.right.left = Node(10)
root.right.right = Node(14)

for i in range(16):
    print "%d %d" %(i, ceil(root, i))

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

Output:

```
0  2
1  2
2  2
3  4
4  4
5  6
6  6
7  8
8  8
9  10
10  10
11  12
12  12
13  14
14  14
15  -1
```

Exercise:

1. Modify above code to find floor value of input key in a binary search tree.

2. Write neat algorithm to find floor and ceil values in a sorted array. Ensure to handle all possible boundary conditions.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/floor-and-ceil-from-a-bst/

Chapter 50

Floor in Binary Search Tree (BST)

Floor in Binary Search Tree (BST) - GeeksforGeeks

Given a Binary Search Tree and a number x, find floor of x in the given BST.

```
Input : x = 14 and root of below tree
            10
           /  \
          5    15
              /  \
             12    30
Output : 12

Input : x = 15 and root of below tree
            10
           /  \
          5    15
              /  \
             12    30
Output : 15
```

A **simple solution** is to traverse the tree using (Inorder or Preorder or Postorder) and keep track of closest smaller or same element. Time complexity of this solution is **O(n)** where n is total number of Nodes in BST.

We can **efficiently** find closes in **O(h)** time where h is height of BST. Algorithm to find the floor of a key in a binary search tree (BST):

```
1 Start at the root Node.
2 If root->data == key,
```

```
     floor of the key is equal
     to the root.
3 Else if root->data > key, then
     floor of the key must lie in the
     left subtree.
4 Else floor may lie in the right subtree
  but only if there is a value lesser than
  or equal to the key.If not, then root is
  the key.
```

For finding ciel of BST you can refer to this article.

```
 // CPP code to find floor of a key in BST
#include <bits/stdc++.h>
using namespace std;

/*Structure of each Node in the tree*/
struct Node {
    int data;
    Node* left, * right;
};

/*This function is used to create and
initialises new Nodes*/
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->left = temp->right =  NULL;
    temp->data = key;
    return temp;
}

/* This function is used to insert
 new values in BST*/
Node* insert(Node* root, int key)
{
    if (!root)
        return newNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

/*This function is used to find floor of a key*/
int floor(Node* root, int key)
{
```

```
    if (!root)
        return INT_MAX;

    /* If root->data is equal to key */
    if (root->data == key)
        return root->data;

    /* If root->data is greater than the key */
    if (root->data > key)
        return floor(root->left, key);

    /* Else, the floor may lie in right subtree
      or may be equal to the root*/
    int floorValue = floor(root->right, key);
    return (floorValue <= key) ? floorValue : root->data;
}

int main()
{
    /* Let us create following BST
              7
            /    \
           5      10
         /  \    /  \
        3    6   8   12 */
    Node* root = NULL;
    root = insert(root, 7);
    insert(root, 10);
    insert(root, 5);
    insert(root, 3);
    insert(root, 6);
    insert(root, 8);
    insert(root, 12);
    cout << floor(root, 9) << endl;
    return 0;
}
```

Output:

```
8
```

Source

https://www.geeksforgeeks.org/floor-in-binary-search-tree-bst/

Chapter 51

Given n appointments, find all conflicting appointments

Given n appointments, find all conflicting appointments - GeeksforGeeks

Given n appointments, find all conflicting appointments.

Examples:

```
Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}}
Output: Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
```

An appointment is conflicting, if it conflicts with any of the previous appointments in array.

We strongly recommend to minimize the browser and try this yourself first.

A **Simple Solution** is to one by one process all appointments from second appointment to last. For every appointment i, check if it conflicts with i-1, i-2, … 0. The time complexity of this method is $O(n^2)$.

We can use **Interval Tree** to solve this problem in O(nLogn) time. Following is detailed algorithm.

```
1) Create an Interval Tree, initially with the first appointment.
2) Do following for all other appointments starting from the second one.
   a) Check if the current appointment conflicts with any of the existing
     appointments in Interval Tree.  If conflicts, then print the current
     appointment.  This step can be done O(Logn) time.
```

```
   b) Insert the current appointment in Interval Tree. This step also can
      be done O(Logn) time.
```

Following is C++ implementation of above idea.

```
 // C++ program to print all conflicting appointments in a
// given set of appointments
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i;  // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree
// Node. This is similar to BST Insert.  Here the low value
//  of interval is used tomaintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval
    //  goes to left subtree
    if (i.low < l)
```

```
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
bool doOVerlap(Interval i1, Interval i2)
{
    if (i1.low < i2.high && i2.low < i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i
// in a given Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOVerlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child
    // is greater than or equal to given interval, then i may
    // overlap with an interval is left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

// This function prints all conflicting appointments in a given
// array of apointments.
void printConflicting(Interval appt[], int n)
{
     // Create an empty Interval Search Tree, add first
     // appointment
```

```
    ITNode *root = NULL;
    root = insert(root, appt[0]);

    // Process rest of the intervals
    for (int i=1; i<n; i++)
    {
        // If current appointment conflicts with any of the
        // existing intervals, print it
        Interval *res = overlapSearch(root, appt[i]);
        if (res != NULL)
           cout << "[" << appt[i].low << "," << appt[i].high
                << "] Conflicts with [" << res->low << ","
                << res->high << "]\n";

        // Insert this appointment
        root = insert(root, appt[i]);
    }
}


// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval appt[] = { {1, 5}, {3, 7}, {2, 6}, {10, 15},
                        {5, 6}, {4, 100}};
    int n = sizeof(appt)/sizeof(appt[0]);
    cout << "Following are conflicting intervals\n";
    printConflicting(appt, n);
    return 0;
}
```

Output:

```
Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
```

Note that the above implementation uses simple Binary Search Tree insert operations. Therefore, time complexity of the above implementation is more than O(nLogn). We can use Red-Black Tree or AVL Tree balancing techniques to make the above implementation O(nLogn).

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/given-n-appointments-find-conflicting-appointments/

Chapter 52

How to check if a given array represents a Binary Heap?

How to check if a given array represents a Binary Heap? - GeeksforGeeks

Given an array, how to check if the given array represents a Binary Max-Heap.

Examples:

```
Input:  arr[] = {90, 15, 10, 7, 12, 2}
Output: True
The given array represents below tree
       90
     /    \
   15      10
  /  \     /
 7    12  2
The tree follows max-heap property as every
node is greater than all of its descendants.

Input:  arr[] = {9, 15, 10, 7, 12, 11}
Output: False
The given array represents below tree
       9
     /    \
   15      10
  /  \     /
 7    12  11
The tree doesn't follows max-heap property 9 is
smaller than 15 and 10, and 10 is smaller than 11.
```

A **Simple Solution** is to first check root, if it's greater than all of its descendants. Then check for children of root. Time complexity of this solution is $O(n^2)$

An **Efficient Solution** is to compare root only with its children (not all descendants), if root is greater than its children and same is true for for all nodes, then tree is max-heap (This conclusion is based on transitive property of > operator, i.e., if x > y and y > z, then x > z).

The last internal node is present at index (2n-2)/2 assuming that indexing begins with 0.

Below is C++ implementation of this solution.

```
 // C program to check whether a given array
// represents a max-heap or not
#include <stdio.h>
#include <limits.h>

// Returns true if arr[i..n-1] represents a
// max-heap
bool isHeap(int arr[], int i, int n)
{
   // If a leaf node
   if (i > (n - 2)/2)
       return true;

   // If an internal node and is greater than its children, and
   // same is recursively true for the children
   if (arr[i] >= arr[2*i + 1]  &&  arr[i] >= arr[2*i + 2] &&
       isHeap(arr, 2*i + 1, n) && isHeap(arr, 2*i + 2, n))
       return true;

   return false;
}

// Driver program
int main()
{
    int arr[] = {90, 15, 10, 7, 12, 2, 7, 3};
    int n = sizeof(arr) / sizeof(int);

    isHeap(arr, 0, n)? printf("Yes"): printf("No");

    return 0;
}
```

Output:

```
Yes
```

Time complexity of this solution is O(n). The solution is similar to preorder traversal of Binary Tree.

Thanks to Utkarsh Trivedi for suggesting the above solution.

An **Iterative Solution** is to traverse all internal nodes and check id node is greater than its children or not.

```
 // C program to check whether a given array
// represents a max-heap or not
#include <stdio.h>
#include <limits.h>

// Returns true if arr[i..n-1] represents a
// max-heap
bool isHeap(int arr[],  int n)
{
    // Start from root and go till the last internal
    // node
    for (int i=0; i<=(n-2)/2; i++)
    {
        // If left child is greater, return false
        if (arr[2*i +1] > arr[i])
                return false;

        // If right child is greater, return false
        if (arr[2*i+2] > arr[i])
                return false;
    }
    return true;
}

// Driver program
int main()
{
    int arr[] = {90, 15, 10, 7, 12, 2, 7, 3};
    int n = sizeof(arr) / sizeof(int);

    isHeap(arr, n)? printf("Yes"): printf("No");

    return 0;
}
```

Output:

```
Yes
```

Thanks to Himanshu for suggesting this solution.

Source

https://www.geeksforgeeks.org/how-to-check-if-a-given-array-represents-a-binary-heap/

Chapter 53

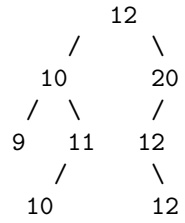
How to handle duplicates in Binary Search Tree?

How to handle duplicates in Binary Search Tree? - GeeksforGeeks

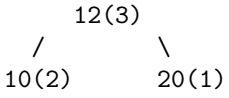
In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Treeby definition has distinct keys.

How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

A **Simple Solution** is to allow same keys on right side (we could also choose left side). For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree



A **Better Solution** is to augment every tree node to store count together with regular fields like key, left and right pointers.
Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.



```
    /    \
 9(1)   11(1)

Count of a key is shown in bracket
```

This approach has following advantages over above simple approach.

1) Height of tree is small irrespective of number of duplicates. Note that most of the BST operations (search, insert and delete) have time complexity as O(h) where h is height of BST. So if we are able to keep the height small, we get advantage of less number of key comparisons.

2) Search, Insert and Delete become easier to do. We can use same insert, search and delete algorithms with small modifications (see below code).

3) This approach is suited for self-balancing BSTs (AVL Tree, Red-Black Tree, etc) also. These trees involve rotations, and a rotation may violate BST property of simple solution as a same key can be in either left side or right side after rotation.

Below is C implementation of normal Binary Search Tree with count with every key. This code basically is taken from code for insert and delete in BST. The changes made for handling duplicates are highlighted, rest of the code is same.

```
 // C program to implement basic operations (search, insert and delete)
// on a BST that handles duplicates by storing count with every node
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    int count;
    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;
    temp->count = 1;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
```

```
        inorder(root->left);
        printf("%d(%d) ", root->key, root->count);
        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);

    // If key already exists in BST, icnrement count and return
    if (key == node->key)
    {
       (node->count)++;
        return node;
    }

    /* 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->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function
   deletes a given key and returns root of modified tree */
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
else
{
    // If key is present more than once, simply decrement
    // count and return
    if (root->count > 1)
    {
       (root->count)--;
       return root;
    }

    // ElSE, delete the node

    // 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
             12(3)
           /        \
       10(2)      20(1)
       /   \
    9(1)  11(1)   */
    struct node *root = NULL;
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 9);
    root = insert(root, 11);
    root = insert(root, 10);
    root = insert(root, 12);
    root = insert(root, 12);

    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 12\n");
    root = deleteNode(root, 12);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 9\n");
    root = deleteNode(root, 9);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Output:

```
Inorder traversal of the given tree
9(1) 10(2) 11(1) 12(3) 20(1)
```

```
Delete 20
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(3)
Delete 12
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(2)
Delete 9
Inorder traversal of the modified tree
10(2) 11(1) 12(2)
```

We will soon be discussing AVL and Red Black Trees with duplicates allowed.

This article is contributed by **Chirag**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/how-to-handle-duplicates-in-binary-search-tree/

Chapter 54

How to implement decrease key or change key in Binary Search Tree?

How to implement decrease key or change key in Binary Search Tree? - GeeksforGeeks

Given a Binary Search Tree, write a function that takes following three as arguments:
1) Root of tree
2) Old key value
3) New Key Value
The function should change old key value to new key value. The function may assume that old key value always exists in Binary Search Tree.

Example:

```
Input: Root of below tree
             50
          /     \
         30      70
        /  \    /  \
       20   40  60   80
     Old key value:  40
     New key value:  10

Output: BST should be modified to following
             50
          /     \
         30      70
        /       /  \
       20      60   80
       /
```

```
     10
```

We strongly recommend you to minimize your browser and try this yourself first

The idea is to call delete for old key value, then call insert for new key value. Below is C++ implementation of the idea.

```
 // C program to demonstrate decrease  key operation on 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->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;
}

// Function to decrease a key value in Binary Search Tree
struct node *changeKey(struct node *root, int oldVal, int newVal)
{
    //  First delete old key value
    root = deleteNode(root, oldVal);

    // Then insert new key value
    root = insert(root, newVal);

    // Return new root
    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);

    root = changeKey(root, 40, 10);

    /* BST is modified to
              50
            /     \
           30      70
          /       /  \
        20      60   80
        /
      10     */
    printf("\nInorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Inorder traversal of the modified tree
10 20 30 50 60 70 80
```

Time complexity of above changeKey() is O(h) where h is height of BST.

Source

https://www.geeksforgeeks.org/how-to-implement-decrease-key-or-change-key-in-binary-search-tree/

Chapter 55

In-place Convert BST into a Min-Heap

In-place Convert BST into a Min-Heap - GeeksforGeeks

Given a Binary Search Tree, convert it into a Min-Heap containing the same elements in O(n) time. Do this in-place.

```
Input: Binary Search Tree
         8
      /    \
     4      12
   /  \     /  \
  2    6   10  14


Output - Min Heap
        2
      /    \
    4        6
  /  \     /   \
 8    10  12   14
[Or any other tree that follows Min Heap
 properties and has same keys]
```

If we are allowed to use extra space, we can perform inorder traversal of the tree and store the keys in an auxiliary array. As we're doing inorder traversal on a BST, array will be sorted. Finally, we construct a complete binary tree from the sorted array. We construct the binary tree level by level and from left to right by taking next minimum element from sorted array. The constructed binary tree will be a min-Heap. This solution works in O(n) time, but is not in-place.

How to do it in-place?
The idea is to convert the binary search tree into a sorted linked list first. We can do this by traversing the BST in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, to maintain sorted order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.

Finally we convert the sorted linked list into a min-Heap by setting the left and right pointers appropriately. We can do this by doing a Level order traversal of the partially built Min-Heap Tree using queue and traversing the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue. As the linked list is sorted, the min-heap property is maintained.

Below is C++ implementation of above idea –

```
 // Program to convert a BST into a Min-Heap
// in O(n) time and in-place
#include <iostream>
#include <queue>
using namespace std;

// Node for BST/Min-Heap
struct Node
{
    int data;
    Node *left, *right;
};

// Utility function for allocating node for BST
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility function to print Min-heap level by level
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order traversal
    queue<Node *> q;
    q.push(root);

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

```
    {
        int nodeCount = q.size();
        while (nodeCount > 0)
        {
            Node *node = q.front();
            cout << node->data << " ";
            q.pop();
            if (node->left)
                q.push(node->left);
            if (node->right)
                q.push(node->right);
            nodeCount--;
        }
        cout << endl;
    }
}

// A simple recursive function to convert a given
// Binary Search tree to Sorted Linked List
// root     --> Root of Binary Search Tree
// head_ref --> Pointer to head node of created
//              linked list
void BSTToSortedLL(Node* root, Node** head_ref)
{
    // Base cases
    if(root == NULL)
        return;

    // Recursively convert right subtree
    BSTToSortedLL(root->right, head_ref);

    // insert root into linked list
    root->right = *head_ref;

    // Change left pointer of previous head
    // to point to NULL
    if (*head_ref != NULL)
        (*head_ref)->left = NULL;

    // Change head of linked list
    *head_ref = root;

    // Recursively convert left subtree
    BSTToSortedLL(root->left, head_ref);
}

// Function to convert a sorted Linked
// List to Min-Heap.
```

```
// root --> Root of Min-Heap
// head --> Pointer to head node of sorted
//              linked list
void SortedLLToMinHeap(Node* &root, Node* head)
{
    // Base Case
    if (head == NULL)
        return;

    // queue to store the parent nodes
    queue<Node *> q;

    // The first node is always the root node
    root = head;

    // advance the pointer to the next node
    head = head->right;

    // set right child to NULL
    root->right = NULL;

    // add first node to the queue
    q.push(root);

    // run until the end of linked list is reached
    while (head)
    {
        // Take the parent node from the q and remove it from q
        Node* parent = q.front();
        q.pop();

        // Take next two nodes from the linked list and
        // Add them as children of the current parent node
        // Also in push them into the queue so that
        // they will be parents to the future nodes
        Node *leftChild = head;
        head = head->right;        // advance linked list to next node
        leftChild->right = NULL; // set its right child to NULL
        q.push(leftChild);

        // Assign the left child of parent
        parent->left = leftChild;

        if (head)
        {
            Node *rightChild = head;
            head = head->right; // advance linked list to next node
            rightChild->right = NULL; // set its right child to NULL
```

```
            q.push(rightChild);

            // Assign the right child of parent
            parent->right = rightChild;
        }
    }
}

// Function to convert BST into a Min-Heap
// without using any extra space
Node* BSTToMinHeap(Node* &root)
{
    // head of Linked List
    Node *head = NULL;

    // Convert a given BST to Sorted Linked List
    BSTToSortedLL(root, &head);

    // set root as NULL
    root = NULL;

    // Convert Sorted Linked List to Min-Heap
    SortedLLToMinHeap(root, head);
}

// Driver code
int main()
{
    /* Constructing below tree
                8
              /   \
             4     12
           /  \   /  \
          2    6 10   14
     */

    Node* root = newNode(8);
    root->left = newNode(4);
    root->right = newNode(12);
    root->right->left = newNode(10);
    root->right->right = newNode(14);
    root->left->left = newNode(2);
    root->left->right = newNode(6);

    BSTToMinHeap(root);

    /* Output - Min Heap
                2
```

```
             /   \
            4     6
          /  \   /  \
         8   10 12   14
     */

    printLevelOrder(root);

    return 0;
}
```

Output :

```
2
4 6
8 10 12 14
```

Source

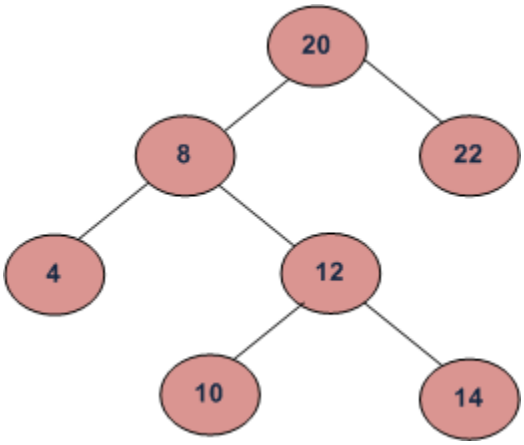
https://www.geeksforgeeks.org/in-place-convert-bst-into-a-min-heap/

Chapter 56

Inorder Successor in Binary Search Tree

Inorder Successor in Binary Search Tree - GeeksforGeeks

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inoorder traversal.
In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Method 1 (Uses Parent Pointer)
In this method, we assume that every node has parent pointer.

The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.
output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.
2) If right sbtree of *node* is NULL, then *succ* is one of the ancestors. Do following.
Travel up using the parent pointer until you see a node which is left child of it's parent. The parent of such a node is the *succ*.

Implementation
Note that the function to find InOrder Successor is highlighted (with gray background) in below code.
C

```
 #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;
    struct node* parent;
};

struct node * minValue(struct node* node);

struct node * inOrderSuccessor(struct node *root, struct node *n)
{
  // step 1 of the above algorithm
  if( n->right != NULL )
    return minValue(n->right);

  // step 2 of the above algorithm
  struct node *p = n->parent;
  while(p != NULL && n == p->right)
  {
     n = p;
     p = p->parent;
  }
  return p;
}

/* Given a non-empty binary search tree, return the minimum data
    value found in that tree. Note that the entire tree does not need
    to be searched. */
struct node * minValue(struct node* node) {
```

```
  struct node* current = node;

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

/* 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;
  node->parent = NULL;

  return(node);
}

/* Give a binary search tree and a number, inserts a new node with
    the given number in the correct place in the tree. Returns the new
    root pointer which the caller should then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
  /* 1. If the tree is empty, return a new,
      single node */
  if (node == NULL)
    return(newNode(data));
  else
  {
    struct node *temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
    {
         temp = insert(node->left, data);
         node->left  = temp;
         temp->parent= node;
    }
    else
    {
        temp = insert(node->right, data);
        node->right = temp;
```

```
        temp->parent = node;
    }

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

/* Driver program to test above functions*/
int main()
{
  struct node* root = NULL, *temp, *succ, *min;

  //creating the tree given in the above diagram
  root = insert(root, 20);
  root = insert(root, 8);
  root = insert(root, 22);
  root = insert(root, 4);
  root = insert(root, 12);
  root = insert(root, 10);
  root = insert(root, 14);
  temp = root->left->right->right;

  succ =  inOrderSuccessor(root, temp);
  if(succ !=  NULL)
    printf("\n Inorder Successor of %d is %d ", temp->data, succ->data);
  else
    printf("\n Inorder Successor doesn't exit");

  getchar();
  return 0;
}
```

Java

```
 // Java program to find minimum value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right, parent;

    Node(int d) {
        data = d;
        left = right = parent = null;
    }
}
```

```
class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
     inserts a new node with the given number in
     the correct place in the tree. Returns the new
     root pointer which the caller should then use
     (the standard trick to avoid using reference
     parameters). */
    Node insert(Node node, int data) {

        /* 1. If the tree is empty, return a new,
         single node */
        if (node == null) {
            return (new Node(data));
        } else {

            Node temp = null;

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data) {
                temp = insert(node.left, data);
                node.left = temp;
                temp.parent = node;

            } else {
                temp = insert(node.right, data);
                node.right = temp;
                temp.parent = node;
            }

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

    Node inOrderSuccessor(Node root, Node n) {

        // step 1 of the above algorithm
        if (n.right != null) {
            return minValue(n.right);
        }

        // step 2 of the above algorithm
        Node p = n.parent;
        while (p != null && n == p.right) {
```

```
            n = p;
            p = p.parent;
        }
        return p;
    }

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

        /* loop down to find the leftmost leaf */
        while (current.left != null) {
            current = current.left;
        }
        return current;
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        Node root = null, temp = null, suc = null, min = null;
        root = tree.insert(root, 20);
        root = tree.insert(root, 8);
        root = tree.insert(root, 22);
        root = tree.insert(root, 4);
        root = tree.insert(root, 12);
        root = tree.insert(root, 10);
        root = tree.insert(root, 14);
        temp = root.left.right.right;
        suc = tree.inOrderSuccessor(root, temp);
        if (suc != null) {
            System.out.println("Inorder successor of " + temp.data +
                                                    " is " + suc.data);
        } else {
            System.out.println("Inorder successor does not exist");
        }
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to find the inroder successor in a BST

# A binary tree node
```

```
class Node:

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

def inOrderSuccessor(root, n):

    # Step 1 of the above algorithm
    if n.right is not None:
        return minValue(n.right)

    # Step 2 of the above algorithm
    p = n.parent
    while( p is not None):
        if n != p.right :
            break
        n = p
        p = p.parent
    return p

# Given a non-empty binary search tree, return the
# minimum data value found in that tree. Note that the
# entire tree doesn't need to be searched
def minValue(node):
    current = node

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

    return current


# Given a binary search tree and a number, inserts a
# new node with the given number in the correct place
# in the tree. Returns the new root pointer which the
# caller should then use( the standard trick to avoid
# using reference parameters)
def insert( node, data):

    # 1) If tree is empty then return a new singly node
    if node is None:
        return Node(data)
```

```
    else:

        # 2) Otherwise, recur down the tree
        if data <= node.data:
            temp = insert(node.left, data)
            node.left = temp
            temp.parent = node
        else:
            temp = insert(node.right, data)
            node.right = temp
            temp.parent = node

        # return  the unchanged node pointer
        return node


# Driver progam to test above function

root  = None

# Creating the tree given in the above diagram
root = insert(root, 20)
root = insert(root, 8);
root = insert(root, 22);
root = insert(root, 4);
root = insert(root, 12);
root = insert(root, 10);
root = insert(root, 14);
temp = root.left.right.right

succ = inOrderSuccessor( root, temp)
if succ is not None:
    print "\nInorder Successor of %d is %d " \
            %(temp.data , succ.data)
else:
    print "\nInorder Successor doesn't exist"

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

Output of the above program:
Inorder Successor of 14 is 20

Time Complexity: O(h) where h is height of tree.

Method 2 (Search from root)
Parent pointer is NOT needed in this algorithm. The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.
output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.
2) If right sbtree of *node* is NULL, then start from root and us search like technique. Do following.
Travel down the tree, if a node's data is greater than root's data then go right side, otherwise go to left side.

```
 struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    struct node *succ = NULL;

    // Start from root and search for successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
           break;
    }

    return succ;
}
```

Thanks to *R.Srinivasan* for suggesting this method.

Time Complexity: O(h) where h is height of tree.

References:
http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm

Source

https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/

Chapter 57

Inorder predecessor and successor for a given key in BST

Inorder predecessor and successor for a given key in BST - GeeksforGeeks

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 ;
};
```

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

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
```

Following is C++ implementation of the above algorithm:

C++

```
 // 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

```
 # 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 fucntion 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"

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

Output:

```
Predecessor is 60
Successor is 70
```

Another Approach :
We can also find the inorder successor and inorder predecessor using inorder traversal . Check if the current node is smaller than the given key for predecessor and for successor, check if it is greater than the given key . If it is greater than the given key then, check if it is smaller than the already stored value in successor then, update it . At last, get the predecessor and successor stored in q(successor) and p(predecessor).

```
 // CPP code for inorder succesor
```

```
// and predecessor of tree
#include<iostream>
#include<stdlib.h>

using namespace std;

struct Node
{
    int data;
    Node* left,*right;
};

// Function to return data
Node* getnode(int info)
{
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = info;
    p->right = NULL;
    p->left = NULL;
    return p;
}

/*
since inorder traversal results in
ascending order visit to node , we
can store the values of the largest
no which is smaller than a (predecessor)
and smallest no which is large than
a (succesor) using inorder traversal
*/
void find_p_s(Node* root,int a,
              Node** p, Node** q)
{
    // If root is null return
    if(!root)
        return ;

    // traverse the left subtree
    find_p_s(root->left, a, p, q);

    // root data is greater than a
    if(root&&root->data > a)
    {

        // q stores the node whose data is greater
        // than a and is smaller than the previously
        // stored data in *q which is sucessor
        if((!*q) || (*q) && (*q)->data > root->data)
```

```
                *q = root;
    }

    // if the root data is smaller than
    // store it in p which is predecessor
    else if(root && root->data < a)
    {
        *p = root;
    }

    // traverse the right subtree
    find_p_s(root->right, a, p, q);
}

// Driver code
int main()
{
    Node* root1 = getnode(50);
    root1->left = getnode(20);
    root1->right = getnode(60);
    root1->left->left = getnode(10);
    root1->left->right = getnode(30);
    root1->right->left = getnode(55);
    root1->right->right = getnode(70);
    Node* p = NULL, *q = NULL;

    find_p_s(root1, 55, &p, &q);

    if(p)
        cout << p->data;
    if(q)
        cout << " " << q->data;
    return 0;
}
```

Output :

```
50 60
```

Thanks **Shweta** for suggesting this method.

This article is contributed by **algoLover**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

Chapter 58

Inorder predecessor and successor for a given key in BST Iterative Approach

Inorder predecessor and successor for a given key in BST Iterative Approach - GeeksforGeeks

Given a BST and a key. The task is to find the inorder successor and predecessor of the given key. In case, if either of predecessor or successor is not present, then print -1.

Examples:

```
Input:          50
               /  \
              /    \
            30      70
           / \      / \
          /   \    /   \
         20   40 60    80
            key = 65
Output: Predecessor : 60
        Successor : 70

Input:          50
               /  \
              /    \
            30      70
           / \      / \
          /   \    /   \
         20   40 60    80
            key = 100
Output: predecessor : 80
```

```
        successor : -1
Explanation: As no node in BST has key value greater
than 100 so -1 is printed for successor.
```

In the previous post, a recursive solution has been discussed. The problem can be solved using an iterative approach. To solve the problem, the three cases while searching for the key has to be dealt with which are as described below:

1. **Root is the given key**: In this case, if the left subtree is not NULL, then predecessor is the rightmost node in left subtree and if right subtree is not NULL, then successor is the leftmost node in right subtree.
2. **Root is greater than key**: In this case, the key is present in left subtree of root. So search for the key in left subtree by setting root = root->left. Note that root could be an inorder successor of given key. In case the key has no right subtree, the root will be its successor.
3. **Root is less than key**: In this case, key is present in right subtree of root. So search for the key in right subtree by setting root = root->right. Note that root could be an inorder predecessor of given key. In case the key has no left subtree, the root will be its predecessor.

Below is the implementation of above approach:

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

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

// Function that finds predecessor and successor of key in BST.
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    if (root == NULL)
        return;

    // Search for given key in BST.
    while (root != NULL) {

        // If root is given key.
        if (root->key == key) {

            // the minimum value in right subtree
```

```
            // is predecessor.
            if (root->right) {
                suc = root->right;
                while (suc->left)
                    suc = suc->left;
            }

            // the maximum value in left subtree
            // is successor.
            if (root->left) {
                pre = root->left;
                while (pre->right)
                    pre = pre->right;
            }

            return;
        }

        // If key is greater than root, then
        // key lies in right subtree. Root
        // could be predecessor if left
        // subtree of key is null.
        else if (root->key < key) {
            pre = root;
            root = root->right;
        }

        // If key is smaller than root, then
        // key lies in left subtree. Root
        // could be successor if right
        // subtree of key is null.
        else {
            suc = root;
            root = root->left;
        }
    }
}

// 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 << "-1";

    if (suc != NULL)
        cout << "Successor is " << suc->key;
    else
        cout << "-1";
    return 0;
```

```
}
```

Output:

```
Predecessor is 60
Successor is 70
```

Time Complexity: O(N)
Auxiliary Space: O(1)

Related Article: https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

Source

https://www.geeksforgeeks.org/inorder-predecessor-and-successor-for-a-given-key-in-bst-iterative-approach/

Chapter 59

Iterative searching in Binary Search Tree

Iterative searching in Binary Search Tree - GeeksforGeeks

Given a binary search tree and a key. Check the given key exist in BST or not without recursion.

Please refer binary search tree insertion for recursive search.

```
 // C++ program to demonstrate searching operation
// in binary search tree without recursion
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *left, *right;
};

// Function to check the given key exist or not
bool iterativeSearch(struct Node *root, int key)
{
    // Traverse untill root reaches to dead end
    while (root != NULL)
    {
        // pass right subtree as new tree
        if (key > root->data)
            root = root->right;

        // pass left subtree as new tree
        else if (key < root->data)
```

```
            root = root->left;
        else
            return true;// if the key is found return 1
    }
    return false;
}

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

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

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

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

// 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);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
```

```
    insert(root, 60);
    insert(root, 80);
    if (iterativeSearch(root, 15))
        cout << "Yes";
    else
        cout << "No";
    return 0;
}
```

Output:

```
No
```

Source

https://www.geeksforgeeks.org/iterative-searching-binary-search-tree/

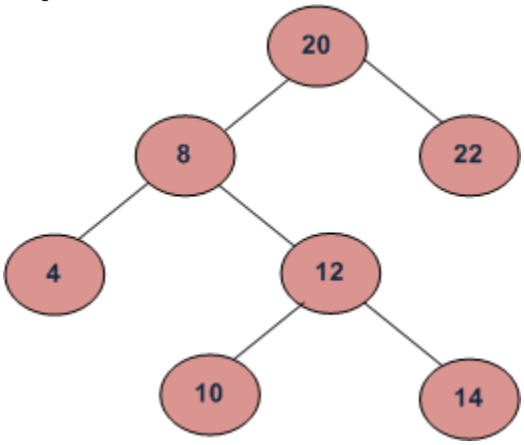
Chapter 60

K'th Largest Element in BST when modification to BST is not allowed

K'th Largest Element in BST when modification to BST is not allowed - GeeksforGeeks

Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



We have discussed two methods in thispost. The method 1 requires O(n) time. The method 2 takes O(h) time where h is height of BST, but requires augmenting the BST (storing count of nodes in left subtree with every node).

Can we find k'th largest element in better than O(n) time and no augmentation?

In this post, a method is discussed that takes O(h + k) time. This method doesn't require any change to BST.

The idea is to do reverse inorder traversal of BST. The reverse inorder traversal traverses all nodes in decreasing order. While doing the traversal, we keep track of count of nodes visited so far. When the count becomes equal to k, we stop the traversal and print the key.
C++

```
 // C++ program to find k'th largest element in BST
#include<iostream>
using namespace std;

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

// 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 function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
    if (c == k)
    {
        cout << "K'th largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
```

```
    kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    kthLargestUtil(root, k, c);
}

/* A utility function to insert a new node with given key in BST */
Node* insert(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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver Program to test above functions
int main()
{
    /* 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);

    int c = 0;
```

```
    for (int k=1; k<=7; k++)
        kthLargest(root, k);

    return 0;
}
```

Java

```
 // Java code to find k'th largest element in BST

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // function to insert nodes
    public void insert(int data)
    {
        this.root = this.insertRec(this.root, data);
    }

    /* A utility function to insert a new node
    with given key in BST */
    Node insertRec(Node node, int data)
    {
        /* If the tree is empty, return a new node */
        if (node == null) {
            this.root = new Node(data);
            return this.root;
```

```
    }

    if (data == node.data) {
        return node;
    }

    /* Otherwise, recur down the tree */
    if (data < node.data) {
        node.left = this.insertRec(node.left, data);
    } else {
        node.right = this.insertRec(node.right, data);
    }
    return node;
}

// class that stores the value of count
public class count {
    int c = 0;
}

// utility function to find kth largest no in
// a given tree
void kthLargestUtil(Node node, int k, count C)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (node == null || C.c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    this.kthLargestUtil(node.right, k, C);

    // Increment count of visited nodes
    C.c++;

    // If c becomes k now, then this is the k'th largest
    if (C.c == k) {
        System.out.println(k + "th largest element is " +
                                            node.data);
        return;
    }

    // Recur for left subtree
    this.kthLargestUtil(node.left, k, C);
}

// Method to find the kth largest no in given BST
```

```
    void kthLargest(int k)
    {
        count c = new count(); // object of class count
        this.kthLargestUtil(this.root, k, c);
    }

    // Driver function
    public static void main(String[] args)
    {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        for (int i = 1; i <= 7; i++) {
            tree.kthLargest(i);
        }
    }
}

// This code is contributed by Kamal Rawal
```

```
K'th largest element is 80
K'th largest element is 70
K'th largest element is 60
K'th largest element is 50
K'th largest element is 40
K'th largest element is 30
K'th largest element is 20
```

Time complexity: The code first traverses down to the rightmost node which takes O(h) time, then traverses k elements in O(k) time. Therefore overall time complexity is O(h + k).

This article is contributed by **Chirag Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-bst-is-not-allowed/

Chapter 61

K'th Largest element in BST using constant extra space

K'th Largest element in BST using constant extra space - GeeksforGeeks

Given a binary search tree, task is to find Kth largest element in the binary search tree.

```
Input :  k = 3
         Root of following BST
            10
           /    \
          4      20
         /      /   \
        2      15     40
Output : 15
```

The idea is to use Reverse Morris Traversal which is based on Threaded Binary Trees. Threaded binary trees use the NULL pointers to store the successor and predecessor information which helps us to utilize the wasted memory by those NULL pointers.

The special thing about Morris traversal is that we can do Inorder traversal without using stack or recursion which saves us memory consumed by stack or recursion call stack.

Reverse Morris traversal is just the reverse of Morris traversal which is majorly used to do Reverse Inorder traversal with constant O(1) extra memory consumed as it does not uses any Stack or Recursion.

To find Kth largest element in a Binary search tree, the simplest logic is to do reverse inorder traversal and while doing reverse inorder traversal simply keep a count of number of Nodes visited. When the count becomes equal to k, we stop the traversal and print the data. It uses the fact that reverse inorder traversal will give us a list sorted in descending order.

Algorithm

```
1) Initialize Current as root.
2) Initialize a count variable to 0.
3) While current is not NULL :
   3.1) If current has no right child
   a) Increment count and check if count is equal to K.
      1) If count is equal to K, simply return current
         Node as it is the Kth largest Node.
   b) Otherwise, Move to the left child of current.

   3.2) Else, here we have 2 cases:
   a) Find the inorder successor of current Node.
      Inorder successor is the left most Node
      in the right subtree or right child itself.
   b) If the left child of the inorder successor is NULL:
      1) Set current as the left child of its inorder
         successor.
      2) Move current Node to its right.
   c) Else, if the threaded link between the current Node
      and it's inorder successor already exists :
      1) Set left pointer of the inorder successor as NULL.
      2) Increment count and check if count is equal to K.
           a) If count is equal to K, simply return current
              Node as it is the Kth largest Node.

      3) Otherwise, Move current to it's left child.
```

```
 // CPP code for finding K-th largest Node using O(1)
// extra memory and reverse Morris traversal.
#include <iostream>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// helper function to create a new Node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->right = temp->left = NULL;
    return temp;
}
```

```
Node* KthLargestUsingMorrisTraversal(Node* root, int k)
{
    Node* curr = root;
    Node* Klargest = NULL;

    // count variable to keep count of visited Nodes
    int count = 0;

    while (curr != NULL) {
        // if right child is NULL
        if (curr->right == NULL) {

            // first increment count and check if count = k
            if (++count == k)
                Klargest = curr;

            // otherwise move to the left child
            curr = curr->left;
        }

        else {

            // find inorder successor of current Node
            Node* succ = curr->right;

            while (succ->left != NULL && succ->left != curr)
                succ = succ->left;

            if (succ->left == NULL) {

                // set left child of successor to the
                // current Node
                succ->left = curr;

                // move current to its right
                curr = curr->right;
            }

            // restoring the tree back to original binary
            //  search tree removing threaded links
            else {

                succ->left = NULL;

                if (++count == k)
                    Klargest = curr;
```

```
                // move current to its left child
                curr = curr->left;
            }
        }
    }

    return Klargest;
}

int main()
{
    // Your C++ Code
    /* Constructed binary tree is
          4
        /   \
       2     7
     /  \   /  \
    1    3  6    10 */

    Node* root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
    root->right->left = newNode(6);
    root->right->right = newNode(10);

    cout << "Finding K-th largest Node in BST : "
         << KthLargestUsingMorrisTraversal(root, 2)->data;

    return 0;
}
```

Output:

```
Finding K-th largest Node in BST : 7
```

Time Complexity :

O(n)

Auxiliary Space :

O(1)

Source

https://www.geeksforgeeks.org/kth-largest-element-bst-using-constant-extra-space/

Chapter 62

K'th largest element in a stream

K'th largest element in a stream - GeeksforGeeks

Given an infinite stream of integers, find the k'th largest element at any point of time.

Example:

```
Input:
stream[] = {10, 20, 11, 70, 50, 40, 100, 5, ...}
k = 3

Output:    {_,   _, 10, 11, 20, 40, 50,  50, ...}
```

Extra space allowed is O(k).

We have discussed different approaches to find k'th largest element in an array in the following posts.
K'th Smallest/Largest Element in Unsorted Array Set 1
K'th Smallest/Largest Element in Unsorted Array Set 2 (Expected Linear Time)
K'th Smallest/Largest Element in Unsorted Array Set 3 (Worst Case Linear Time)

Here we have a stream instead of whole array and we are allowed to store only k elements.

A **Simple Solution** is to keep an array of size k. The idea is to keep the array sorted so that the k'th largest element can be found in O(1) time (we just need to return first element of array if array is sorted in increasing order)
How to process a new element of stream?
For every new element in stream, check if the new element is smaller than current k'th largest element. If yes, then ignore it. If no, then remove the smallest element from array and insert new element in sorted order. Time complexity of processing a new element is O(k).

A **Better Solution** is to use a Self Balancing Binary Search Tree of size k. The k'th largest element can be found in O(Logk) time.

How to process a new element of stream?
For every new element in stream, check if the new element is smaller than current k'th largest element. If yes, then ignore it. If no, then remove the smallest element from the tree and insert new element. Time complexity of processing a new element is O(Logk).

An **Efficient Solution** is to use Min Heap of size k to store k largest elements of stream. The k'th largest element is always at root and can be found in O(1) time.
How to process a new element of stream?
Compare the new element with root of heap. If new element is smaller, then ignore it. Otherwise replace root with new element and call heapify for the root of modified heap. Time complexity of finding the k'th largest element is O(Logk).

```
 // A C++ program to find k'th smallest element in a stream
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void buildHeap();
    void MinHeapify(int i);  //To minheapify subtree rooted with index i
    int parent(int i)  { return (i-1)/2;  }
    int left(int i)    { return (2*i + 1);  }
    int right(int i)   { return (2*i + 2);  }
    int extractMin();  // extracts root (minimum) element
    int getMin()       {  return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(int x) { harr[0] = x; MinHeapify(0); }
};

MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
}

void MinHeap::buildHeap()
{
    int i = (heap_size - 1)/2;
```

```
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the minimum vakue.
    int root = harr[0];

    // If there are more than 1 items, move the last item to root
    // and call heapify.
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        MinHeapify(0);
    }
    heap_size--;

    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
```

```
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element from input stream
void kthLargest(int k)
{
    // count is total no. of elements in stream seen so far
    int count = 0, x;  // x is for new element

    // Create a min heap of size k
    int *arr = new int[k];
    MinHeap mh(arr, k);


    while (1)
    {
        // Take next element from stream
        cout << "Enter next element of stream ";
        cin >> x;

        // Nothing much to do for first k-1 elements
        if (count < k-1)
        {
            arr[count] = x;
            count++;
        }

        else
        {
          // If this is k'th element, then store it
          // and build the heap created above
          if (count == k-1)
          {
              arr[count] = x;
              mh.buildHeap();
          }

          else
          {
               // If next element is greater than
               // k'th largest, then replace the root
               if (x > mh.getMin())
                  mh.replaceMin(x); // replaceMin calls
                                    // heapify()
          }
```

```
            // Root of heap is k'th largest element
            cout << "K'th largest element is "
                 << mh.getMin() << endl;
            count++;
        }
    }
}

// Driver program to test above methods
int main()
{
    int k = 3;
    cout << "K is " << k << endl;
    kthLargest(k);
    return 0;
}
```

Output

```
K is 3
Enter next element of stream 23
Enter next element of stream 10
Enter next element of stream 15
K'th largest element is 10
Enter next element of stream 70
K'th largest element is 15
Enter next element of stream 5
K'th largest element is 15
Enter next element of stream 80
K'th largest element is 23
Enter next element of stream 100
K'th largest element is 70
Enter next element of stream
CTRL + C pressed
```

This article is contributed by **Shivam Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/kth-largest-element-in-a-stream/

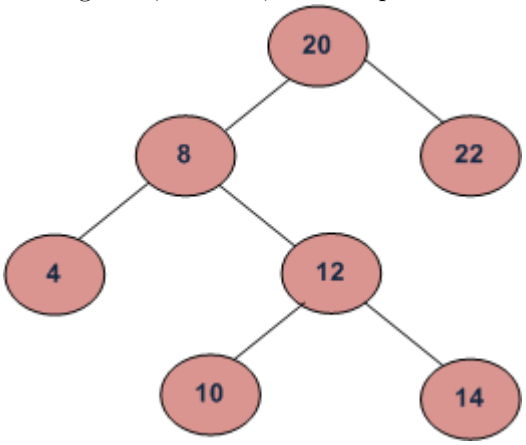
Chapter 63

K'th smallest element in BST using O(1) Extra Space

K'th smallest element in BST using O(1) Extra Space - GeeksforGeeks

Given a Binary Search Tree (BST) and a positive integer k, find the k'th smallest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



We have discussed two methods in thispost and one method in thispost. All of the previous methods require extra space. How to find the k'th largest element without extra space?

The idea is to use Morris Traversal. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree. See thisfor more details.

Below is C++ implementation of the idea.

```
 // C++ program to find k'th largest element in BST
#include<iostream>
```

```
#include<climits>
using namespace std;

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

// A function to find
int KSmallestUsingMorris(Node *root, int k)
{
    // Count to iterate over elements till we
    // get the kth smallest number
    int count = 0;

    int ksmall = INT_MIN; // store the Kth smallest
    Node *curr = root; // to store the current node

    while (curr != NULL)
    {
        // Like Morris traversal if current does
        // not have left child rather than printing
        // as we did in inorder, we will just
        // increment the count as the number will
        // be in an increasing order
        if (curr->left == NULL)
        {
            count++;

            // if count is equal to K then we found the
            // kth smallest, so store it in ksmall
            if (count==k)
                ksmall = curr->key;

            // go to current's right child
            curr = curr->right;
        }
        else
        {
            // we create links to Inorder Successor and
            // count using these links
            Node *pre = curr->left;
            while (pre->right != NULL && pre->right != curr)
                pre = pre->right;

            // building links
```

```
            if (pre->right==NULL)
            {
                //link made to Inorder Successor
                pre->right = curr;
                curr = curr->left;
            }

            // While breaking the links in so made temporary
            // threaded tree we will check for the K smallest
            // condition
            else
            {
                // Revert the changes made in if part (break link
                // from the Inorder Successor)
                pre->right = NULL;

                count++;

                // If count is equal to K then we found
                // the kth smallest and so store it in ksmall
                if (count==k)
                    ksmall = curr->key;

                curr = curr->right;
            }
        }
    }
    return ksmall; //return the found value
}

// 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 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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver Program to test above functions
int main()
{
    /* 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);

    for (int k=1; k<=7; k++)
       cout << KSmallestUsingMorris(root, k) << " ";

    return 0;
}
```

Output:

```
20 30 40 50 60 70 80
```

This article is contributed by Abhishek Somani. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/kth-smallest-element-in-bst-using-o1-extra-space/

Chapter 64

Largest BST in a Binary Tree Set 2

Largest BST in a Binary Tree Set 2 - GeeksforGeeks

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

```
Input:
      5
    /  \
   2    4
 /  \
1    3

Output: 3
The following subtree is the
maximum size BST subtree
   2
 /  \
1    3


Input:
       50
     /    \
  30       60
 /  \     /  \
5   20   45    70
```

```
             /  \
           65    80
Output: 5
The following subtree is the
maximum size BST subtree
      60
     /  \
   45    70
        /  \
      65    80
```

We have discussed a two methods in below post.
Find the largest BST subtree in a given Binary Tree Set 1

In this post a different O(n) solution is discussed. This solution is simpler than the solutions discussed above and works in O(n) time.

The idea is based on method 3 of check if a binary tree is BST article.

A Tree is BST if following is true for every node x.

1. The largest value in left subtree (of x) is smaller than value of x.
2. The smallest value in right subtree (of x) is greater than value of x.

We traverse tree in bottom up manner. For every traversed node, we return maximum and minimum values in subtree rooted with it. If any node follows above properties and size of

```
 // C++ program to find largest BST in a
// Binary Tree.
#include <bits/stdc++.h>
using namespace std;

/* 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;
};

/* 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 = new Node;
```

```
    node->data = data;
    node->left = node->right = NULL;

    return(node);
}

// Information to be returned by every
// node in bottom up traversal.
struct Info
{
    int sz; // Size of subtree
    int max; // Min value in subtree
    int min; // Max value in subtree
    int ans; // Size of largest BST which
    // is subtree of current node
    bool isBST; // If subtree is BST
};

// Returns Information about subtree. The
// Information also includes size of largest
// subtree which is a BST.
Info largestBSTBT(Node* root)
{
    // Base cases : When tree is empty or it has
    // one child.
    if (root == NULL)
        return {0, INT_MIN, INT_MAX, 0, true};
    if (root->left == NULL && root->right == NULL)
        return {1, root->data, root->data, 1, true};

    // Recur for left subtree and right subtrees
    Info l = largestBSTBT(root->left);
    Info r = largestBSTBT(root->right);

    // Create a return variable and initialize its
    // size.
    Info ret;
    ret.sz = (1 + l.sz + r.sz);

    // If whole tree rooted under current root is
    // BST.
    if (l.isBST && r.isBST && l.max < root->data &&
            r.min > root->data)
    {
        ret.min = min(l.min, min(r.min, root->data));
        ret.max = max(r.max, max(l.max, root->data));

        // Update answer for tree rooted under
```

```
        // current 'root'
        ret.ans = ret.sz;
        ret.isBST = true;

        return ret;
    }

    // If whole tree is not BST, return maximum
    // of left and right subtrees
    ret.ans = max(l.ans, r.ans);
    ret.isBST = false;

    return ret;
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
        60
       /  \
      65  70
     /
    50 */

    struct Node *root = newNode(60);
    root->left = newNode(65);
    root->right = newNode(70);
    root->left->left = newNode(50);
    printf(" Size of the largest BST is %d\n",
           largestBSTBT(root).ans);
    return 0;
}

// This code is contributed by Vivek Garg in a
// comment on below set 1.
// www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/
```

Output:

```
Size of largest BST is 2
```

Time Complexity : O(n)

Source

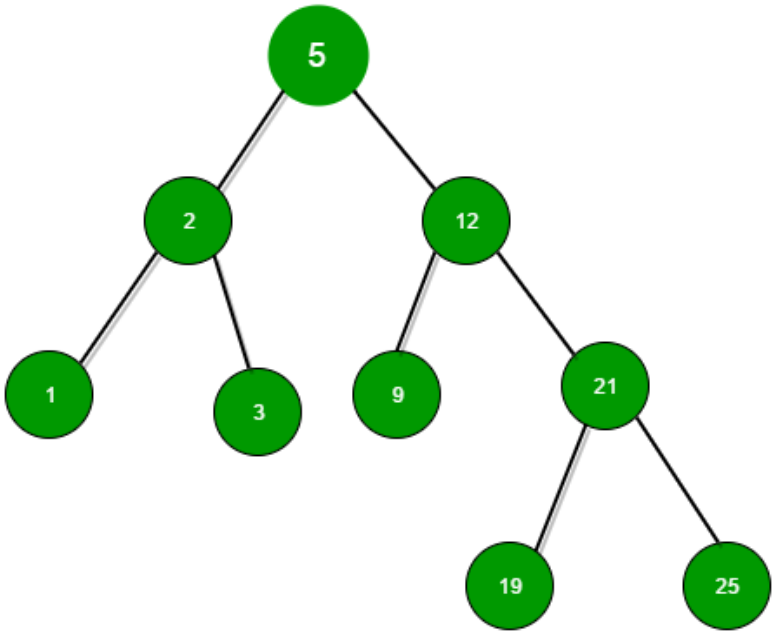
https://www.geeksforgeeks.org/largest-bst-binary-tree-set-2/

Chapter 65

Largest number in BST which is less than or equal to N

Largest number in BST which is less than or equal to N - GeeksforGeeks

We have a binary search tree and a number N. Our task is to find the greatest number in the binary search tree that is less than or equal to N. Print the value of the element if it exists otherwise print -1.



Examples:For the above given binary search tree-

```
Input : N = 24
Output :result = 21
(searching for 24 will be like-5->12->21)

Input  : N = 4
Output : result = 3
(searching for 4 will be like-5->2->3)
```

We follow recursive approach for solving this problem. We start searching for element from root node. If we reach a leaf and its value is greater than N, element does not exist so return -1. Else if node's value is less than or equal to N and right value is NULL or greater than N, then return the node value as it will be the answer.
Otherwise if node's value is greater than N, then search for the element in the left subtree else search for the element in the right subtree by calling the same function by passing the left or right values accordingly.

```
 // C++ code to find the largest value smaller
// than or equal to N
#include <bits/stdc++.h>
using namespace std;

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

// To create new BST Node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// To insert a new node in BST
Node* insert(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return newNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
```

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

// function to find max value less then N
int findMaxforN(Node* root, int N)
{
    // Base cases
    if (root == NULL)
        return -1;
    if (root->key == N)
        return N;

    // If root's value is smaller, try in rght
    // subtree
    else if (root->key < N) {
        int k = findMaxforN(root->right, N);
        if (k == -1)
            return root->key;
        else
            return k;
    }

    // If root's key is greater, return value
    // from left subtree.
    else if (root->key > N)
        return findMaxforN(root->left, N);
}

// Driver code
int main()
{
    int N = 4;

    // creating following BST
    /*
                  5
               /   \
             2     12
           /  \    /  \
         1   3   9   21
                    /   \
                   19   25  */
    Node* root = insert(root, 25);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
```

```
    insert(root, 12);
    insert(root, 9);
    insert(root, 21);
    insert(root, 19);
    insert(root, 25);

    printf("%d", findMaxforN(root, N));

    return 0;
}
```

Output:

```
3
```

Time complexity = O(h) where h is height of BST.

Reference :
https://www.careercup.com/question?id=5765237112307712

Improved By : abhishektayal0

Source

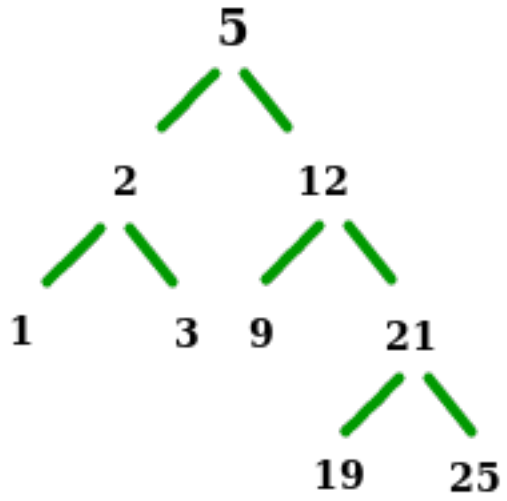
https://www.geeksforgeeks.org/largest-number-bst-less-equal-n/

Chapter 66

Largest number less than or equal to N in BST (Iterative Approach)

Largest number less than or equal to N in BST (Iterative Approach) - GeeksforGeeks

We have a binary search tree and a number N. Our task is to find the greatest number in the binary search tree that is less than or equal to N. Print the value of the element if it exists otherwise print -1.



Examples:For the above given binary search tree-

```
Input : N = 24
Output :result = 21
```

```
(searching for 24 will be like-5->12->21)

Input  : N = 4
Output : result = 3
(searching for 4 will be like-5->2->3)
```

We have discussed recursive approach in below post.
Largest number in BST which is less than or equal to N

Here an iterative approach is discussed. We try to find the predecessor of the target. Keep two pointers, one pointing to the current node and one for storing the answer. If the current node's data > N, we move towards left. In other case, when current node's data is less than N, the current node can be our answer (so far), and we move towards right.

```
 // C++ code to find the largest value smaller
// than or equal to N
#include <bits/stdc++.h>
using namespace std;

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

// To create new BST Node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// To insert a new node in BST
Node* insert(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return newNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // return the (unchanged) node pointer
```

```
    return node;
}

// Returns largest value smaller than or equal to
// key. If key is smaller than the smallest, it
// returns -1.
int findFloor(Node* root, int key)
{
    Node *curr = root, *ans = NULL;
    while (curr) {
        if (curr->key <= key) {
            ans = curr;
            curr = curr->right;
        }
        else
            curr = curr->left;
    }
    if (ans)
        return ans->key;
    return -1;
}

// Driver code
int main()
{
    int N = 25;

    Node* root = insert(root, 19);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 12);
    insert(root, 9);
    insert(root, 21);
    insert(root, 19);
    insert(root, 25);

    printf("%d", findFloor(root, N));

    return 0;
}
```

Output:

```
25
```

Source

https://www.geeksforgeeks.org/largest-number-less-equal-n-bst-iterative-approach/

Chapter 67

Leaf nodes from Preorder of a Binary Search Tree

Leaf nodes from Preorder of a Binary Search Tree - GeeksforGeeks

Given a Preorder traversal of a Binary Search Tree. The task is to print leaf nodes of the Binary Search Tree from the given preorder.

Examples:

```
Input : preorder[] = {890, 325, 290, 530, 965};
Output : 290 530 965
Explanation : Tree represented is,
      890
     /   \
  325    965
  /  \
290   530

Input : preorder[] = { 3, 2, 4 };
Output : 2 4
```

Method 1: (Simple)

The idea is to find Iorder, then traverse the tree in preorder fashion (using both inorder and postorder traversals) and while traversing print leaf nodes.

How to traverse in preorder fashion using two arrays representing inorder and preorder traversals?
We iterate the preorder array and for each element find that element in the inorder array. For searching, we can use binary search, since inorder traversal of binary search tree is always sorted. Now, for each element of preorder array, in binary search we set the range [L, R].

And when L == R, leaf node is found. So, initially, L = 0 and R = n – 1 for first element (i.e root) of preorder array. Now, to search for element on the left subtree of root, set L = 0 and R = index of root – 1. Also, for all element of right subtree set L = index of root + 1 and R = n -1.
Recursively, follow this, until L == R.

Below is C++ implementation of this approach:

```
 // C++ program to print leaf node from
// preorder of binary search tree.
#include<bits/stdc++.h>
using namespace std;

// Binary Search
int binarySearch(int inorder[], int l, int r, int d)
{
    int mid = (l + r)>>1;

    if (inorder[mid] == d)
        return mid;

    else if (inorder[mid] > d)
        return binarySearch(inorder, l, mid - 1, d);

    else
        return binarySearch(inorder, mid + 1, r, d);
}

// Function to print Leaf Nodes by doing preorder
// traversal of tree using preorder and inorder arrays.
void leafNodesRec(int preorder[], int inorder[],
                  int l, int r, int *ind, int n)
{
    // If l == r, therefore no right or left subtree.
    // So, it must be leaf Node, print it.
    if(l == r)
    {
        printf("%d ", inorder[l]);
        *ind = *ind + 1;
        return;
    }

    // If array is out of bound, return.
    if (l < 0 || l > r || r >= n)
        return;

    // Finding the index of preorder element
    // in inorder array using binary search.
    int loc = binarySearch(inorder, l, r, preorder[*ind]);
```

```
    // Incrementing the index.
    *ind = *ind + 1;

    // Finding on the left subtree.
    leafNodesRec(preorder, inorder, l, loc - 1, ind, n);

    // Finding on the right subtree.
    leafNodesRec(preorder, inorder, loc + 1, r, ind, n);
}

// Finds leaf nodes from given preorder traversal.
void leafNodes(int preorder[], int n)
{
    int inorder[n];  // To store inorder traversal

    // Copy the preorder into another array.
    for (int i = 0; i < n; i++)
        inorder[i] = preorder[i];

    // Finding the inorder by sorting the array.
    sort(inorder, inorder + n);

    // Point to the index in preorder.
    int ind = 0;

    // Print the Leaf Nodes.
    leafNodesRec(preorder, inorder, 0, n - 1, &ind, n);
}

// Driven Program
int main()
{
    int preorder[] = { 890, 325, 290, 530, 965 };
    int n = sizeof(preorder)/sizeof(preorder[0]);

    leafNodes(preorder, n);
    return 0;
}
```

Output:

```
290 530 965
```

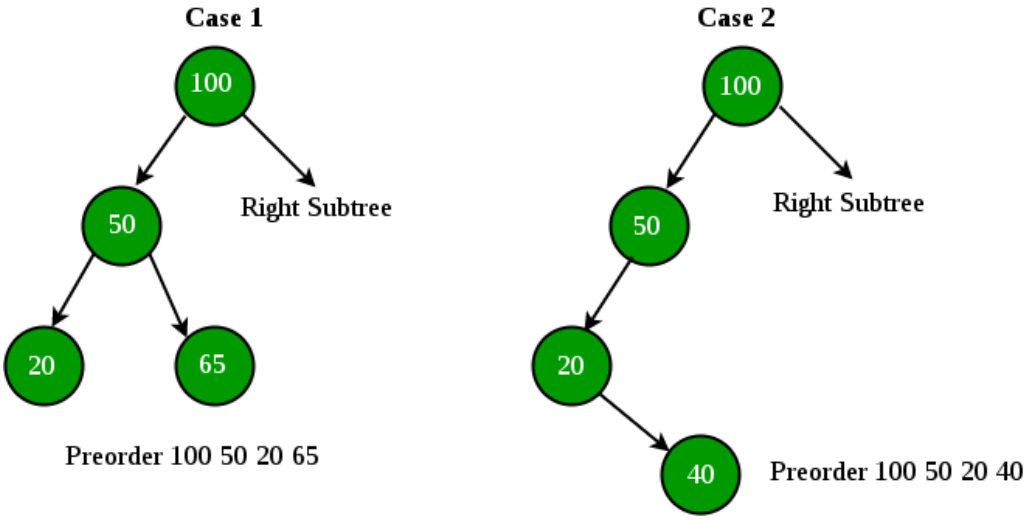
Time Complexity: O(n log n)
Auxiliary Space: O(n)

Method 2:(using Stack)

The idea is to use the property of the Binary Search Tree and stack.
Traverse the array using two pointer i and j to the array, initially i = 0 and j = 1. Whenever a[i] > a[j], we can say a[j] is left part of a[i], since preorder traversal follows Visit -> Left -> Right. So, we push a[i] into the stack.
For those points voilating the rule, we start to pop element from the stack till a[i] > top element of the stack and break when it doesn't and print the corresponding j^{th} value.
Algorithm:

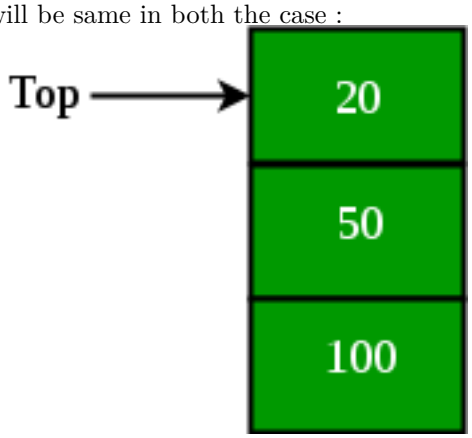
```
1. Set i = 0, j = 1.
2. Traverse the preorder array.
3. If a[i] > a[j], push a[i] to the stack.
4. Else
   While (stack is not empty)
     if (a[j] > top of stack)
       pop element from the stack;
       set found = true;
     else
       break;
5. if (found == true)
     print a[i];
```

How this algorithm works?
Preorder traversal traverse in the order: Visit, Left, Right.
And we know left node of any node in BST is always less than node. So preorder traversal will first traverse from root to leftmost node. Therefore, preorder will be in decreasing order first. Now, after decreasing order there may be node which is greater or which break the decreasing order. So, there can be case like this :



In case 1, 20 is leaf node whereas in case 2, 20 is not the leaf node.
So, our problem is how to identify if we have to print 20 as leaf node or not?
This is solved using stack.

While running above algorithm on case 1 and case 2, when i = 2 and j = 3, state of stack will be same in both the case :



So, node 65 will pop 20 and 50 from the stack. This is because 65 is the right child of a node which is before 20. This information we store using found varible. So, 20 is a root node.
While in case 2, 40 will not able to pop any element from the stack. Because 40 is right node of a node which is after 20. So, 20 is not a leaf node.

Note: In the algorithm, we will not be able to check condition of leaf node of rightmost node or rightmost element of the preorder. So, simply print the rightmost node because we know this will always be leaf node in preorder traversal.

Below is C++ implementation of this approach:

```
 // Stack based C++ program to print leaf nodes
// from preorder traversal.
#include<bits/stdc++.h>
using namespace std;

// Print the leaf node from the given preorder of BST.
void leafNode(int preorder[], int n)
{
    stack<int> s;
    for (int i = 0, j = 1; j < n; i++, j++)
    {
        bool found = false;

        if (preorder[i] > preorder[j])
            s.push(preorder[i]);

        else
        {
            while (!s.empty())
            {
                if (preorder[j] > s.top())
                {
```

```
                    s.pop();
                    found = true;
                }
                else
                    break;
            }
        }

        if (found)
            cout << preorder[i] << " ";
    }

    // Since rightmost element is always leaf node.
    cout << preorder[n - 1];
}

// Driver code
int main()
{
    int preorder[] = { 890, 325, 290, 530, 965 };
    int n = sizeof(preorder)/sizeof(preorder[0]);

    leafNode(preorder, n);
    return 0;
}
```

```
290 530 965
```

Time Complexity: O(n)

Source

https://www.geeksforgeeks.org/leaf-nodes-preorder-binary-search-tree/

Chapter 68

Left Leaning Red Black Tree (Insertion)

Left Leaning Red Black Tree (Insertion) - GeeksforGeeks

Prerequisites : Red – Black Trees.

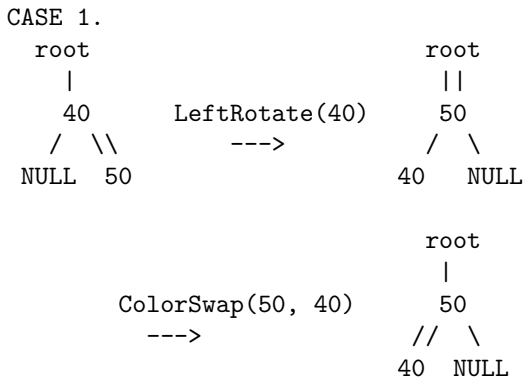
A left leaning Red Black Tree or **(LLRB)**, is a variant of red black tree, which is a lot easier to implement than Red black tree itself and guarantees all the search, delete and insert operations in O(logn) time.

Which nodes are RED and Which are Black ?
Nodes which have double incoming edge are RED in color.
Nodes which have single incoming edge are BLACK in color.

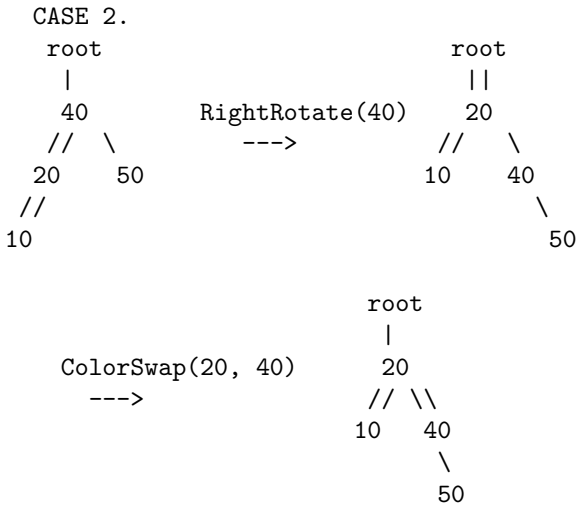
Characteristics of LLRB
1. Root node is Always BLACK in color.
2. Every new Node inserted is always RED in color.
3. Every NULL child of a node is considered as BLACK in color.
Eg : only 40 is present in tree.

```
 root
   |
   40 <-- as 40 is the root so it
  /  \    is also Black in color.
NULL  NULL <-- Black in color.
```

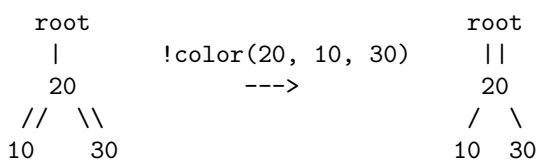
4. There should not be a node which has RIGHT RED child and LEFT BLACK child(or NULL child as all NULLS are BLACK), if present Left rotate the node, and swap the colors of current node and its **LEFT** child so as to maintain consistency for rule 2 i.e., new node must be RED in color.



5. There should not be a node which has LEFT RED child and LEFT RED grandchild, if present Right Rotate the node and swap the colors between node and it's **RIGHT** child to follow rule 2.



6. There should not be a node which has LEFT RED child and RIGHT RED child, if present Invert the colors of all nodes i. e., current_node, LEFT child, and RIGHT child.

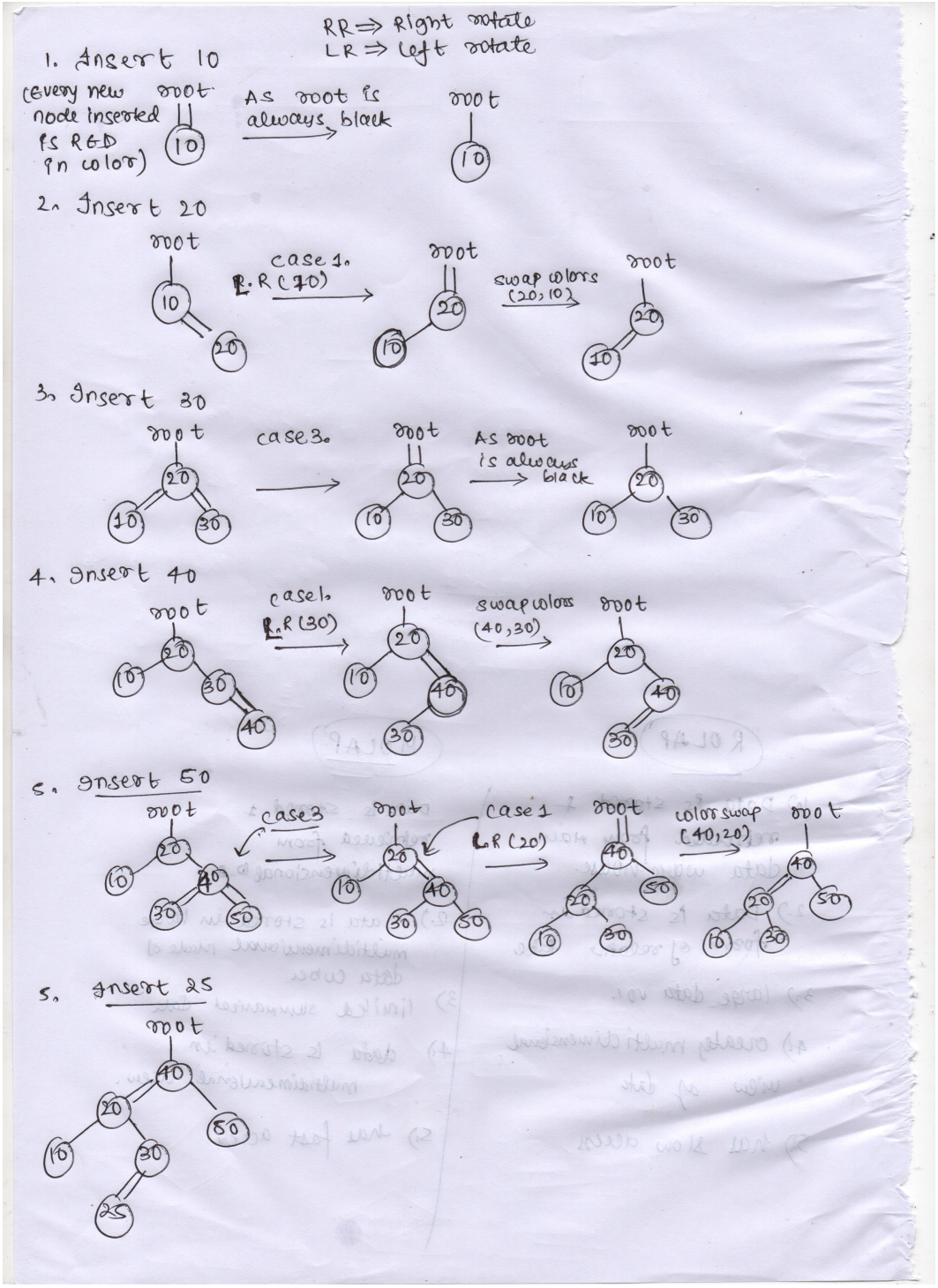


```
                                  root
        As the root is always black    |
                ---->                 20
                                     /  \
                                   10   30
```

Why are we following the above mentioned rules? Because by following above characteristics/rules we are able to simulate all the red-black tree's properties without caring about the complex implementation of it.
Example:

```
Insert the following data into LEFT LEANING RED-BLACK
TREE and display the inorder traversal of tree.
Input : 10 20 30 40 50 25
Output : 10 20 30 40 50 25
        root
         |
         40
       //  \
      20    50
     /  \
   10    30
        //
       25
```

Approach :
Insertions in the LLRB is exactly like inserting into a Binary search tree. The difference is that that After we insert the node into the tree we will retrace our steps back to root and try to enforce the above rules for LLRB.
While doing the above rotations and swapping of color it may happen that our root becomes RED in color so we also. We have to make sure that our root remains always BLACK in color.


RR ⇒ Right rotate
LR ⇒ Left rotate
1. Insert 10
(every new node inserted is RED in color)
root
10
As root is always black
root
10
2. Insert 20
root
10
20
case 1.
R. R (10)
root
20
10
swap colors (20, 10)
root
20
10
3. Insert 30
root
20
10
30
case 3.
root
20
10
30
As root is always black
root
20
10
30
4. Insert 40
root
20
10
30
40
case 1.
R. R (30)
root
20
10
40
30
swap colors (40, 30)
root
20
10
40
30
5. Insert 50
root
20
10
40
30
50
case 3
root
20
10
40
30
50
case 1
R. R (20)
root
40
20
50
10
30
color swap (40, 20)
root
40
20
50
10
30
5. Insert 25
root
40
20
50
10
30
25

C

```
 // C program to implement insert operation
// in Red Black Tree.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct node
{
    struct node *left, *right;
    int data;

    // red ==> true, black ==> false
    bool color;
} node;



// utility function to create a node.
node* createNode(int data, bool color)
{
    node *myNode = (node *) malloc(sizeof(node));
    myNode -> left = myNode -> right = NULL;
    myNode -> data = data;

    // New Node which is created is
    // always red in color.
    myNode -> color = true;
    return myNode;
}

// utility funciton to rotate node anticlockwise.
node* rotateLeft(node* myNode)
{
    printf("left rotation!!\n");
    node *child = myNode -> right;
    node *childLeft = child -> left;

    child -> left = myNode;
    myNode -> right = childLeft;

    return child;
}

// utility function to rotate node clockwise.
node* rotateRight(node* myNode)
{
    printf("right rotation\n");
    node *child = myNode -> left;
```

```
    node *childRight =  child -> right;

    child -> right = myNode;
    myNode -> left = childRight;

    return child;
}

// utility funciton to check whether
// node is red in color or not.
int isRed(node *myNode)
{
    if (myNode == NULL)
       return 0;
    return (myNode -> color == true);
}

// utility function to swap color of two
// nodes.
void swapColors(node *node1, node *node2)
{
    bool temp = node1 -> color;
    node1 -> color = node2 -> color;
    node2 -> color = temp;
}

// insertion into Left Leaning Red Black Tree.
node* insert(node* myNode, int data)
{
    // Normal insertion code for any Binary
    // Search tree.
    if (myNode == NULL)
        return createNode(data, false);

    if (data < myNode -> data)
        myNode -> left = insert(myNode -> left, data);

    else if (data > myNode -> data)
        myNode -> right = insert(myNode -> right, data);

    else
        return myNode;


    // case 1.
    // when right child is Red but left child is
    // Black or doesn't exist.
    if (isRed(myNode -> right) && !isRed(myNode -> left))
```

```
    {
        // left rotate the node to make it into
        // valid structure.
        myNode = rotateLeft(myNode);

        // swap the colors as the child node
        // should always be red
        swapColors(myNode, myNode -> left);

    }

    // case 2
    // when left child as well as left grand child in Red
    if (isRed(myNode -> left) && isRed(myNode -> left -> left))
    {
        // right rotate the current node to make
        // it into a valid structure.
        myNode = rotateRight(myNode);
        swapColors(myNode, myNode -> right);
    }


    // case 3
    // when both left and right child are Red in color.
    if (isRed(myNode -> left) && isRed(myNode -> right))
    {
        // invert the color of node as well
        // it's left and right child.
        myNode -> color = !myNode -> color;

        // change the color to black.
        myNode -> left -> color = false;
        myNode -> right -> color = false;
    }

    return myNode;
}

// Inorder traversal
void inorder(node *node)
{
    if (node)
    {
        inorder(node -> left);
        printf("%d ", node -> data);
        inorder(node -> right);
    }
}
```

```
// Driver function
int main()
{
    node *root = NULL;
    /* LLRB tree made after all insertions are made.

    1. Nodes which have double INCOMING edge means
       that they are RED in color.
    2. Nodes which have single INCOMING edge means
       that they are BLACK in color.

        root
         |
         40
        //  \
       20    50
      /  \
    10    30
         //
        25    */

    root = insert(root, 10);
    // to make sure that root remains
    // black is color
    root -> color = false;

    root = insert(root, 20);
    root -> color = false;

    root = insert(root, 30);
    root -> color = false;

    root = insert(root, 40);
    root -> color = false;

    root = insert(root, 50);
    root -> color = false;

    root = insert(root, 25);
    root -> color = false;

    // display the tree through inorder traversal.
    inorder(root);

    return 0;
}
```

Java

```
 // Java program to implement insert operation
// in Red Black Tree.

class node
{
    node left, right;
    int data;

    // red ==> true, black ==> false
    boolean color;

    node(int data)
    {
        this.data = data;
        left = null;
        right = null;

        // New Node which is created is
        // always red in color.
        color = true;
    }
}

public class LLRBTREE {

    private static node root = null;

    // utility funciton to rotate node anticlockwise.
    node rotateLeft(node myNode)
    {
        System.out.printf("left rotation!!\n");
        node child = myNode.right;
        node childLeft = child.left;

        child.left = myNode;
        myNode.right = childLeft;

        return child;
    }

    // utility function to rotate node clockwise.
    node rotateRight(node myNode)
    {
        System.out.printf("right rotation\n");
        node child = myNode.left;
        node childRight = child.right;
```

```
    child.right = myNode;
    myNode.left = childRight;

    return child;
}

// utility funciton to check whether
// node is red in color or not.
boolean isRed(node myNode)
{
    if (myNode == null)
        return false;
    return (myNode.color == true);
}

// utility function to swap color of two
// nodes.
void swapColors(node node1, node node2)
{
    boolean temp = node1.color;
    node1.color = node2.color;
    node2.color = temp;
}

// insertion into Left Leaning Red Black Tree.
node insert(node myNode, int data)
{
    // Normal insertion code for any Binary
    // Search tree.
    if (myNode == null)
        return new node(data);

    if (data < myNode.data)
        myNode.left = insert(myNode.left, data);

    else if (data > myNode.data)
        myNode.right = insert(myNode.right, data);

    else
        return myNode;


    // case 1.
    // when right child is Red but left child is
    // Black or doesn't exist.
    if (isRed(myNode.right) && !isRed(myNode.left))
    {
```

```
            // left rotate the node to make it into
            // valid structure.
            myNode = rotateLeft(myNode);

            // swap the colors as the child node
            // should always be red
            swapColors(myNode, myNode.left);

        }

        // case 2
        // when left child as well as left grand child in Red
        if (isRed(myNode.left) && isRed(myNode.left.left))
        {
            // right rotate the current node to make
            // it into a valid structure.
            myNode = rotateRight(myNode);
            swapColors(myNode, myNode.right);
        }


        // case 3
        // when both left and right child are Red in color.
        if (isRed(myNode.left) && isRed(myNode.right))
        {
            // invert the color of node as well
            // it's left and right child.
            myNode.color = !myNode.color;

            // change the color to black.
            myNode.left.color = false;
            myNode.right.color = false;
        }

        return myNode;
    }


    // Inorder traversal
    void inorder(node node)
    {
        if (node != null)
        {
            inorder(node.left);
            System.out.print(node.data + " ");
            inorder(node.right);
        }
    }
```

```
    public static void main(String[] args) {
    /* LLRB tree made after all insertions are made.

    1. Nodes which have double INCOMING edge means
        that they are RED in color.
    2. Nodes which have single INCOMING edge means
        that they are BLACK in color.

        root
        |
        40
        // \
    20 50
    / \
    10 30
        //
        25 */

    LLRBTREE node = new LLRBTREE();

    root = node.insert(root, 10);
    // to make sure that root remains
    // black is color
    root.color = false;

    root = node.insert(root, 20);
    root.color = false;

    root = node.insert(root, 30);
    root.color = false;

    root = node.insert(root, 40);
    root.color = false;

    root = node.insert(root, 50);
    root.color = false;

    root = node.insert(root, 25);
    root.color = false;

    // display the tree through inorder traversal.
    node.inorder(root);

    }

}
```

```
// This code is contributed by ARSHPREET_SINGH
```

Output:

```
left rotation!!
left rotation!!
left rotation!!
10 20 25 30 40 50
```

Time Complexity : O(log n)

Source

https://www.geeksforgeeks.org/left-leaning-red-black-tree-insertion/

Chapter 69

Lowest Common Ancestor in a Binary Search Tree.

Lowest Common Ancestor in a Binary Search Tree. - GeeksforGeeks

Given values of two values n1 and n2 in a Binary Search Tree, find the **L**owest **C**ommon **A**ncestor (LCA). You may assume that both the values exist in the tree.

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Following is definition of LCA from Wikipedia:
Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the

node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

C

```
 // A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

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

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data  = data;
    node->left  = node->right = NULL;
    return(node);
}

/* Driver program to test lca() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root        = newNode(20);
    root->left               = newNode(8);
    root->right              = newNode(22);
    root->left->left         = newNode(4);
```

```
    root->left->right        = newNode(12);
    root->left->right->left  = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    getchar();
    return 0;
}
```

Java

```
 // Recursive Java program to print lca of two nodes

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to find LCA of n1 and n2. The function assumes that both
       n1 and n2 are present in BST */
    Node lca(Node node, int n1, int n2)
    {
        if (node == null)
            return null;
```

```
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (node.data > n1 && node.data > n2)
            return lca(node.left, n1, n2);

        // If both n1 and n2 are greater than root, then LCA lies in right
        if (node.data < n1 && node.data < n2)
            return lca(node.right, n1, n2);

        return node;
    }

    /* Driver program to test lca() */
    public static void main(String args[])
    {
        // Let us construct the BST shown in the above figure
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(20);
        tree.root.left = new Node(8);
        tree.root.right = new Node(22);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(12);
        tree.root.left.right.left = new Node(10);
        tree.root.left.right.right = new Node(14);

        int n1 = 10, n2 = 14;
        Node t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

        n1 = 14;
        n2 = 8;
        t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

        n1 = 10;
        n2 = 22;
        t = tree.lca(tree.root, n1, n2);
        System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # A recursive python program to find LCA of two nodes
# n1 and n2
```

```
# A Binary tree node
class Node:

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

# Function to find LCA of n1 and n2. The function assumes
# that both n1 and n2 are present in BST
def lca(root, n1, n2):

    # Base Case
    if root is None:
        return None

    # If both n1 and n2 are smaller than root, then LCA
    # lies in left
    if(root.data > n1 and root.data > n2):
        return lca(root.left, n1, n2)

    # If both n1 and n2 are greater than root, then LCA
    # lies in right
    if(root.data < n1 and root.data < n2):
        return lca(root.right, n1, n2)

    return root

# Driver program to test above function

# Let us construct the BST shown in the figure
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

n1 = 10 ; n2 = 14
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2 , t.data)
```

```
n1 = 10 ; n2 = 22
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

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

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time complexity of above solution is O(h) where h is height of tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
 /* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
         // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
           root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
           root = root->right;

        else break;
    }
    return root;
}
```

See thisfor complete program.

You may like to see below articles as well :

Lowest Common Ancestor in a Binary Tree

LCA using Parent Pointer

Find LCA in Binary Tree using RMQ

Exercise
The above functions assume that n1 and n2 both are in BST. If n1 and n2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n1 or n2 or both not present in BST.

Source

https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/

Chapter 70

Maximum Unique Element in every subarray of size K

Maximum Unique Element in every subarray of size K - GeeksforGeeks

Given an array and an integer K. We need to find the maximum of every segment of length K which has no duplicates in that segment.

Examples:

```
Input : a[] = {1, 2, 2, 3, 3},
          K = 3.
Output : 1 3 2
For segment (1, 2, 2), Maximum = 1.
For segment (2, 2, 3), Maximum = 3.
For segment (2, 3, 3), Maximum = 2.

Input : a[] = {3, 3, 3, 4, 4, 2},
          K = 4.
Output : 4 Nothing 3
```

A **simple solution** is to run two loops. For every subarray find all distinct elements and print maximum unique element.

An **efficient solution** is to use sliding window technique. We maintain two structures in every window.
1) A hash table to store counts of all elements in current window.
2) A self balancing BST (implemented using set in C++ STL and TreeSet in Java). The idea is to quickly find maximum element and update maximum element.

We process first K-1 elements and store their counts in hash table. We also store unique elements in set. Now we one by one process last element of every window. If current element is unique, we add it to set. We also increase its count. After processing last element, we

print maximum from set. Before starting next iteration, we remove first element of previous window.

```
 // C++ code to calculate maximum unique
// element of every segment of array
#include <bits/stdc++.h>
using namespace std;

void find_max(int A[], int N, int K)
{
    // Storing counts of first K-1 elements
    // Also storing distinct elements.
    map<int, int> Count;
    for (int i = 0; i < K - 1; i++)
        Count[A[i]]++;
    set<int> Myset;
    for (auto x : Count)
        if (x.second == 1)
            Myset.insert(x.first);

    // Before every iteration of this loop,
    // we maintain that K-1 elements of current
    // window are processed.
    for (int i = K - 1; i < N; i++) {

        // Process K-th element of current window
        Count[A[i]]++;
        if (Count[A[i]] == 1)
            Myset.insert(A[i]);
        else
            Myset.erase(A[i]);

        // If there are no distinct
        // elements in current window
        if (Myset.size() == 0)
            printf("Nothing\n");

        // Set is ordered and last element
        // of set gives us maximum element.
        else
            printf("%d\n", *Myset.rbegin());

        // Remove first element of current
        // window before next iteration.
        int x = A[i - K + 1];
        Count[x]--;
        if (Count[x] == 1)
            Myset.insert(x);
```

```
        if (Count[x] == 0)
            Myset.erase(x);
    }
}

// Driver code
int main()
{
    int a[] = { 1, 2, 2, 3, 3 };
    int n = sizeof(a) / sizeof(a[0]);
    int k = 3;
    find_max(a, n, k);
    return 0;
}
```

Output:

```
1
3
2
```

Time Complexity : O(N Log K)

Source

https://www.geeksforgeeks.org/maximum-unique-element-every-subarray-size-k/

Chapter 71

Maximum element between two nodes of BST

Maximum element between two nodes of BST - GeeksforGeeks

Given an array of **N** elements and two integers **A, B** which belongs to the given array. Create a Binary Search Tree by inserting element from arr[0] to arr[n-1]. The task is to find the maximum element in the path from A to B.

Examples :

```
Input : arr[] = { 18, 36, 9, 6, 12, 10, 1, 8 },
        a = 1,
        b = 10.
Output : 12


Path from 1 to 10 contains { 1, 6, 9, 12, 10 }. Maximum element is 12.
```

The idea is to find Lowest Common Ancestor of node 'a' and node 'b'. Then search maximum node between LCA and 'a', also find maximum node between LCA and 'b'. Answer will be maximum node of two.

C++

```
 // C++ program to find maximum element in the path
// between two Nodes of Binary Search Tree.
#include <bits/stdc++.h>
using namespace std;

struct Node
{
```

```
    struct Node *left, *right;
    int data;
};

// Create and return a pointer of new Node.
Node *createNode(int x)
{
    Node *p = new Node;
    p -> data = x;
    p -> left = p -> right = NULL;
    return p;
}

// Insert a new Node in Binary Search Tree.
void insertNode(struct Node *root, int x)
{
    Node *p = root, *q = NULL;

    while (p != NULL)
    {
        q = p;
        if (p -> data < x)
            p = p -> right;
        else
            p = p -> left;
    }

    if (q == NULL)
        p = createNode(x);
    else
    {
        if (q -> data < x)
            q -> right = createNode(x);
        else
            q -> left = createNode(x);
    }
}

// Return the maximum element between a Node
// and its given ancestor.
int maxelpath(Node *q, int x)
{
    Node *p = q;

    int mx = INT_MIN;

    // Traversing the path between ansector and
    // Node and finding maximum element.
```

```
    while (p -> data != x)
    {
        if (p -> data > x)
        {
            mx = max(mx, p -> data);
            p = p -> left;
        }
        else
        {
            mx = max(mx, p -> data);
            p = p -> right;
        }
    }

    return max(mx, x);
}

// Return maximum element in the path between
// two given Node of BST.
int maximumElement(struct Node *root, int x, int y)
{
    Node *p = root;

    // Finding the LCA of Node x and Node y
    while ((x < p -> data && y < p -> data) ||
        (x > p -> data && y > p -> data))
    {
        // Checking if both the Node lie on the
        // left side of the parent p.
        if (x < p -> data && y < p -> data)
            p = p -> left;

        // Checking if both the Node lie on the
        // right side of the parent p.
        else if (x > p -> data && y > p -> data)
            p = p -> right;
    }

    // Return the maximum of maximum elements occur
    // in path from ancestor to both Node.
    return max(maxelpath(p, x), maxelpath(p, y));
}


// Driver Code
int main()
{
    int arr[] = { 18, 36, 9, 6, 12, 10, 1, 8 };
```

```
    int a = 1, b = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Creating the root of Binary Search Tree
    struct Node *root = createNode(arr[0]);

    // Inserting Nodes in Binary Search Tree
    for (int i = 1; i < n; i++)
        insertNode(root, arr[i]);

    cout << maximumElement(root, a, b) << endl;

    return 0;
}
```

Output :

```
12
```

Time complexity : O(h) where h is height of BST

Improved By : sanjeetkumarSingh

Source

https://www.geeksforgeeks.org/maximum-element-two-nodes-bst/

Chapter 72

Merge Two Balanced Binary Search Trees

Merge Two Balanced Binary Search Trees - GeeksforGeeks

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/C++

```
 #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;
}
```

Java

```
 import java.io.*;
import java.util.ArrayList;

// A binary tree node
class Node {
```

```
    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree
{

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Inorder traversal of the tree
    void inorder()
    {
        inorderUtil(this.root);
    }

// Utility function for inorder traversal of the tree
void inorderUtil(Node node)
{
    if(node==null)
        return;

    inorderUtil(node.left);
    System.out.print(node.data + " ");
    inorderUtil(node.right);
}


    // A Utility Method that stores inorder traversal of a tree
    public ArrayList<Integer> storeInorderUtil(Node node, ArrayList<Integer> list)
    {
        if(node == null)
            return list;

        //recur on the left child
        storeInorderUtil(node.left, list);
```

```
    // Adds data to the list
    list.add(node.data);

    //recur on the right child
    storeInorderUtil(node.right, list);

    return list;
}

// Method that stores inorder traversal of a tree
ArrayList<Integer> storeInorder(Node node)
{
    ArrayList<Integer> list1 = new ArrayList<>();
    ArrayList<Integer> list2 = storeInorderUtil(node,list1);
    return list2;
}

// Method that merges two ArrayLists into one.
ArrayList<Integer> merge(ArrayList<Integer>list1, ArrayList<Integer>list2, int m, int n)
{
    // list3 will contain the merge of list1 and list2
    ArrayList<Integer> list3 = new ArrayList<>();
    int i=0;
    int j=0;

    //Traversing through both ArrayLists
    while( i<m && j<n)
    {
        // Smaller one goes into list3
        if(list1.get(i)<list2.get(j))
        {
            list3.add(list1.get(i));
            i++;
        }
        else
        {
            list3.add(list2.get(j));
            j++;
        }
    }

    // Adds the remaining elements of list1 into list3
    while(i<m)
    {
        list3.add(list1.get(i));
        i++;
    }
```

```
    // Adds the remaining elements of list2 into list3
    while(j<n)
    {
        list3.add(list2.get(j));
        j++;
    }
    return list3;
}

// Method that converts an ArrayList to a BST
Node ALtoBST(ArrayList<Integer>list, int start, int end)
{
    // Base case
    if(start > end)
        return null;

    // Get the middle element and make it root
    int mid = (start+end)/2;
    Node node = new Node(list.get(mid));

    /* Recursively construct the left subtree and make it
    left child of root */
    node.left = ALtoBST(list, start, mid-1);

    /* Recursively construct the right subtree and make it
    right child of root */
    node.right = ALtoBST(list, mid+1, end);

return node;
}

// Method that merges two trees into a single one.
Node mergeTrees(Node node1, Node node2)
{
    //Stores Inorder of tree1 to list1
    ArrayList<Integer>list1 = storeInorder(node1);

    //Stores Inorder of tree2 to list2
    ArrayList<Integer>list2 = storeInorder(node2);

    // Merges both list1 and list2 into list3
    ArrayList<Integer>list3 = merge(list1, list2, list1.size(), list2.size());

    //Eventually converts the merged list into resultant BST
    Node node = ALtoBST(list3, 0, list3.size()-1);
    return node;
}
```

```
    // Driver function
    public static void main (String[] args)
    {

        /* Creating following tree as First balanced BST
                100
                / \
                50 300
                / \
               20 70
        */

        BinarySearchTree tree1 = new BinarySearchTree();
        tree1.root = new Node(100);
        tree1.root.left = new Node(50);
        tree1.root.right = new Node(300);
        tree1.root.left.left = new Node(20);
        tree1.root.left.right = new Node(70);

        /* Creating following tree as second balanced BST
                80
                / \
              40 120
        */

        BinarySearchTree tree2 = new BinarySearchTree();
        tree2.root = new Node(80);
        tree2.root.left = new Node(40);
        tree2.root.right = new Node(120);


        BinarySearchTree tree = new BinarySearchTree();
        tree.root = tree.mergeTrees(tree1.root, tree2.root);
        System.out.println("The Inorder traversal of the merged BST is: ");
        tree.inorder();
    }
}
// This code has been contributed by Kamal Rawal
```

Output:

```
Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300
```

Method 3 (In-Place Merge using DLL)
We can use a Doubly Linked List to merge trees in place. Following are the steps.

1) Convert the given two Binary Search Trees into doubly linked list in place (Refer this post for this step).
2) Merge the two sorted Linked Lists (Refer this post for this step).
3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer this post for this step)

Time complexity of this method is also O(m+n) and this method does conversion in place.

Thanks to Dheeraj and Ronzii for suggesting this method.

Source

https://www.geeksforgeeks.org/merge-two-balanced-binary-search-trees/

Chapter 73

Merge two BSTs with limited extra space

Merge two BSTs with limited extra space - GeeksforGeeks

Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is O(m+n) where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is O(height of the first tree + height of the second tree).

Examples:

```
First BST
       3
    /     \
   1       5
Second BST
    4
  /   \
2       6
Output: 1 2 3 4 5 6
```

```
First BST
          8
         / \
        2   10
       /
      1
Second BST
          5
         /
```

```
        3
       /
      0
Output: 0 1 2 3 5 8 10
```

Source: Google interview question

A similar question has been discussed earlier. Let us first discuss already discussed methods of the previous post which was for Balanced BSTs. The method 1 can be applied here also, but the time complexity will be O(n^2) in worst case. The method 2 can also be applied here, but the extra space required will be O(n) which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 can't be done in O(n) for an unbalanced BST.

Thanks to Kumar for suggesting the following solution.

The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

```
 #include<stdio.h>
#include<stdlib.h>

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..................... START OF STACK RELATED STUFF...................
// A stack node
struct snode
{
    struct node  *t;
    struct snode *next;
};

// Function to add an elemt k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
```

```
}

// Function to pop an element t from stack
struct node *pop(struct snode **s)
{
    struct  node *t;
    struct snode *st;
    st=*s;
    (*s) = (*s)->next;
    t = st->t;
    free(st);
    return t;
}

// Fucntion to check whether the stack is empty or not
int isEmpty(struct snode *s)
{
    if (s == NULL )
        return 1;

    return 0;
}
//..................... END OF STACK RELATED STUFF.....................


/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A utility function to print Inoder traversal of a Binary Tree */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// The function to print data of two BSTs in sorted order
void  merge(struct node *root1, struct node *root2)
```

```
{
    // s1 is stack to hold nodes of first BST
    struct snode *s1 = NULL;

    // Current node of first BST
    struct node  *current1 = root1;

    // s2 is stack to hold nodes of second BST
    struct snode *s2 = NULL;

    // Current node of second BST
    struct node  *current2 = root2;

    // If first BST is empty, then output is inorder
    // traversal of second BST
    if (root1 == NULL)
    {
        inorder(root2);
        return;
    }
    // If second BST is empty, then output is inorder
    // traversal of first BST
    if (root2 == NULL)
    {
        inorder(root1);
        return ;
    }

    // Run the loop while there are nodes not yet printed.
    // The nodes may be in stack(explored, but not printed)
    // or may be not yet explored
    while (current1 != NULL || !isEmpty(s1) ||
          current2 != NULL || !isEmpty(s2))
    {
        // Following steps follow iterative Inorder Traversal
        if (current1 != NULL || current2 != NULL )
        {
            // Reach the leftmost node of both BSTs and push ancestors of
            // leftmost nodes to stack s1 and s2 respectively
            if (current1 != NULL)
            {
                push(&s1, current1);
                current1 = current1->left;
            }
            if (current2 != NULL)
            {
                push(&s2, current2);
                current2 = current2->left;
```

```
    }

}
else
{
    // If we reach a NULL node and either of the stacks is empty,
    // then one tree is exhausted, ptint the other tree
    if (isEmpty(s1))
    {
        while (!isEmpty(s2))
        {
            current2 = pop (&s2);
            current2->left = NULL;
            inorder(current2);
        }
        return ;
    }
    if (isEmpty(s2))
    {
        while (!isEmpty(s1))
        {
            current1 = pop (&s1);
            current1->left = NULL;
            inorder(current1);
        }
        return ;
    }

    // Pop an element from both stacks and compare the
    // popped elements
    current1 = pop(&s1);
    current2 = pop(&s2);

    // If element of first tree is smaller, then print it
    // and push the right subtree. If the element is larger,
    // then we push it back to the corresponding stack.
    if (current1->data < current2->data)
    {
        printf("%d ", current1->data);
        current1 = current1->right;
        push(&s2, current2);
        current2 = NULL;
    }
    else
    {
        printf("%d ", current2->data);
        current2 = current2->right;
        push(&s1, current1);
```

```
                current1 = NULL;
            }
        }
    }
}

/* Driver program to test above functions */
int main()
{
    struct node  *root1 = NULL, *root2 = NULL;

    /* Let us create the following tree as first tree
            3
          /  \
         1    5
     */
    root1 = newNode(3);
    root1->left = newNode(1);
    root1->right = newNode(5);

    /* Let us create the following tree as second tree
            4
          /  \
         2    6
     */
    root2 = newNode(4);
    root2->left = newNode(2);
    root2->right = newNode(6);

    // Print sorted nodes of both trees
    merge(root1, root2);

    return 0;
}
```

Time Complexity: O(m+n)
Auxiliary Space: O(height of the first tree + height of the second tree)

Source

https://www.geeksforgeeks.org/merge-two-bsts-with-limited-extra-space/

Chapter 74

Minimum Possible value of ai + aj – k for given array and k.

Minimum Possible value of ai + aj - k for given array and k. - GeeksforGeeks

You are given an array of n integer and an integer K. Find the number of total unordered pairs {i, j} such that absolute value of (ai + aj – K), i.e., ai + aj – k is minimal possible where i != j.

Examples:

```
Input : arr[] = {0, 4, 6, 2, 4},
            K = 7
Output : Minimal Value = 1
         Total  Pairs = 5
Explanation : Pairs resulting minimal value are :
              {a1, a3}, {a2, a4}, {a2, a5}, {a3, a4}, {a4, a5}

Input : arr[] = {4, 6, 2, 4}  , K = 9
Output : Minimal Value = 1
         Total Pairs = 4
Explanation : Pairs resulting minimal value are :
              {a1, a2}, {a1, a4}, {a2, a3}, {a2, a4}
```

A **simple solution** is iterate over all possible pairs and for each pair we will check whether the value of (ai + aj – K) is smaller then our current smallest value of not. So as per result of above condition we have total of three cases :

1. *abs(ai + aj – K) > smallest* : do nothing as this pair will not count in minimal possible value.
2. *abs(ai + aj – K) = smallest* : increment the count of pair resulting minimal possible value.

3. *abs(ai + aj – K) < smallest* : update the smallest value and set count to 1.

C++

```
 // CPP program to find number of pairs  and minimal
// possible value
#include<bits/stdc++.h>
using namespace std;

// function for finding pairs and min value
void pairs(int arr[], int n, int k)
{
    // initialize smallest and count
    int smallest = INT_MAX;
    int count=0;

    // iterate over all pairs
    for (int i=0; i<n; i++)
        for(int j=i+1; j<n; j++)
        {
            // is abs value is smaller than smallest
            // update smallest and reset count to 1
            if ( abs(arr[i] + arr[j] - k) < smallest )
            {
                smallest = abs(arr[i] + arr[j] - k);
                count = 1;
            }

            // if abs value is equal to smallest
            // increment count value
            else if (abs(arr[i] + arr[j] - k) == smallest)
                count++;
        }

        // print result
        cout << "Minimal Value = " << smallest << "\n";
        cout << "Total Pairs = " << count << "\n";
}

// driver program
int main()
{
    int arr[] = {3, 5, 7, 5, 1, 9, 9};
    int k = 12;
    int n = sizeof(arr) / sizeof(arr[0]);
    pairs(arr, n, k);
    return 0;
}
```

Java

```
 // Java program to find number of pairs
// and minimal possible value
import java.util.*;

class GFG {

    // function for finding pairs and min value
    static void pairs(int arr[], int n, int k)
    {
        // initialize smallest and count
        int smallest = Integer.MAX_VALUE;
        int count=0;

        // iterate over all pairs
        for (int i=0; i<n; i++)
            for(int j=i+1; j<n; j++)
            {
                // is abs value is smaller than
                // smallest update smallest and
                // reset count to 1
                if ( Math.abs(arr[i] + arr[j] - k) <
                                    smallest )
                {
                    smallest = Math.abs(arr[i] + arr[j]
                                        - k);
                    count = 1;
                }

                // if abs value is equal to smallest
                // increment count value
                else if (Math.abs(arr[i] + arr[j] - k)
                                    == smallest)
                    count++;
            }

            // print result
           System.out.println("Minimal Value = " +
                                    smallest);
           System.out.println("Total Pairs = " +
                                      count);
    }

    /* Driver program to test above function */
    public static void main(String[] args)
    {
        int arr[] = {3, 5, 7, 5, 1, 9, 9};
```

```
        int k = 12;
        int n = arr.length;
        pairs(arr, n, k);
    }
}
// This code is contributed by Arnav Kr. Mandal.
```

C#

```
 // C# program to find number
// of pairs and minimal
// possible value
using System;

class GFG
{

// function for finding
// pairs and min value
static void pairs(int []arr,
                  int n, int k)
{
    // initialize
    // smallest and count
    int smallest = 0;
    int count = 0;

    // iterate over all pairs
    for (int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
        {
            // is abs value is smaller
            // than smallest update
            // smallest and reset
            // count to 1
            if (Math.Abs(arr[i] +
                arr[j] - k) < smallest )
            {
                smallest = Math.Abs(arr[i] +
                                    arr[j] - k);
                count = 1;
            }

            // if abs value is equal
            // to smallest increment
            // count value
            else if (Math.Abs(arr[i] +
                              arr[j] - k) ==
```

```
                                smallest)
                count++;
        }

    // print result
    Console.WriteLine("Minimal Value = " +
                                smallest);
    Console.WriteLine("Total Pairs = " +
                                 count);
}

// Driver Code
public static void Main()
{
    int []arr = {3, 5, 7,
                 5, 1, 9, 9};
    int k = 12;
    int n = arr.Length;
    pairs(arr, n, k);
}
}

// This code is contributed
// by anuj_67.
```

PHP

```
 <?php
// PHP program to find number of
// pairs and minimal possible value

// function for finding pairs
// and min value
function pairs($arr, $n, $k)
{

    // initialize smallest and count
    $smallest = PHP_INT_MAX;
    $count = 0;

    // iterate over all pairs
    for ($i = 0; $i < $n; $i++)
        for($j = $i + 1; $j < $n; $j++)
        {

            // is abs value is smaller than smallest
            // update smallest and reset count to 1
            if ( abs($arr[$i] + $arr[$j] - $k) < $smallest )
```

```
            {
                $smallest = abs($arr[$i] + $arr[$j] - $k);
                $count = 1;
            }

            // if abs value is equal to smallest
            // increment count value
            else if (abs($arr[$i] +
                     $arr[$j] - $k) == $smallest)
                $count++;
        }

        // print result
        echo "Minimal Value = " , $smallest , "\n";
        echo "Total Pairs = ", $count , "\n";
}

    // Driver Code
    $arr = array (3, 5, 7, 5, 1, 9, 9);
    $k = 12;
    $n = sizeof($arr);
    pairs($arr, $n, $k);

// This code is contributed by aj_36
?>
```

Output:

```
Minimal Value = 0
Total Pairs = 4
```

An **efficient solution** is to use a self balancing binary search tree (which is implemented in set in C++ and TreeSet in Java). We can find closest element in O(log n) time in map.

C++

```
 // C++ program to find number of pairs
// and minimal possible value
#include<bits/stdc++.h>
using namespace std;

// function for finding pairs and min value
void pairs(int arr[], int n, int k)
{
    // initialize smallest and count
    int smallest = INT_MAX, count = 0;
    set<int> s;
```

```
    // iterate over all pairs
    s.insert(arr[0]);
    for (int i=1; i<n; i++)
    {
        // Find the closest elements to  k - arr[i]
        int lower = *lower_bound(s.begin(),
                                 s.end(),
                                 k - arr[i]);

        int upper = *upper_bound(s.begin(),
                                 s.end(),
                                 k - arr[i]);

        // Find absolute value of the pairs formed
        // with closest greater and smaller elements.
        int curr_min = min(abs(lower + arr[i] - k),
                           abs(upper + arr[i] - k));

        // is abs value is smaller than smallest
        // update smallest and reset count to 1
        if (curr_min < smallest)
        {
            smallest = curr_min;
            count = 1;
        }

        // if abs value is equal to smallest
        // increment count value
        else if (curr_min == smallest )
            count++;
        s.insert(arr[i]);

    }        // print result

    cout << "Minimal Value = " << smallest <<"\n";
    cout << "Total Pairs = " << count <<"\n";
}

// driver program
int main()
{
    int arr[] = {3, 5, 7, 5, 1, 9, 9};
    int k = 12;
    int n = sizeof(arr) / sizeof(arr[0]);
    pairs(arr, n, k);
    return 0;
}
```

Output:

```
Minimal Value = 0
Total Pairs = 4
```

Time Complexity : O(n Log n)

Improved By : jit_t, vt_m

Source

https://www.geeksforgeeks.org/minimum-possible-value-ai-aj-k-given-array-k/

Chapter 75

Next Greater Element Set-2

Next Greater Element Set-2 - GeeksforGeeks

Given an array, print theNext Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

```
Input: arr[] = {4, 5, 2, 25}
Output:
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1
```

Approach: In this post, we will be discussing how to find the Next Greater Element using C++ STL(set). Given below are the steps to find the Next Greater Element of every index element.

- Insert all the elements in the Multi-Set, it will store all the elements in an increasing order.
- Iterate on the array of elements, and for each index, find the upper_bound of the current index element. The upper_bound() returns an iterator which can point to the following position.
    1. If the iterator is pointing to a position past the last element, then there exists no NGE to the current index element.
    2. If the iterator points to a position referring to an element, then that element is the NGE to the current index element.

- Find the position of current index element at every traversal and remove it from the multiset using lower_bound() and erase() functions of multiset.

Below is the implementation of the above approach.

```
 // C++ program to print the
// NGE's of array elements using
// C++ STL
#include <bits/stdc++.h>
using namespace std;

// Function to print the NGE
void printNGE(int a[], int n)
{

    multiset<int> ms;

    // insert in the multiset container
    for (int i = 0; i < n; i++) {
        ms.insert(a[i]);
    }

    cout << "Element   "<< "NGE";

    // traverse for all array elements
    for (int i = 0; i < n; i++) {

        // find the upper_bound in set
        auto it = ms.upper_bound(a[i]);

        // if points to the end, then
        // no NGE of that element
        if (it == ms.end()) {
            cout << "\n   " << a[i]
                 << " ----> " << -1;
        }

        // print the element at that position
        else {
            cout << "\n   " << a[i]
                 << " ----> " << *it;
        }

        // find the first occurence of
        // the index element and delete it
        it = ms.lower_bound(a[i]);

        // delete one occurence
```

```
        // from the container
        ms.erase(it);
    }
}

// Driver Code
int main()
{
    int a[] = { 4, 5, 2, 25 };
    int n = sizeof(a) / sizeof(a[0]);

    // Function call to print the NGE
    printNGE(a, n);
    return 0;
}
```

Output:

```
Element   NGE
   4 ----> 5
   5 ----> 25
   2 ----> 25
   25 ----> -1
```

Time Complexity: O(N log N)

Note that the time complexity of the previous solution is O(N). The idea of this article is to demonstrate an alternate approach to solve the problem.

Source

https://www.geeksforgeeks.org/next-greater-element-set-2/

Chapter 76

Number of elements smaller than root using preorder traversal of a BST

Number of elements smaller than root using preorder traversal of a BST - GeeksforGeeks

Given a preorder traversal of a BST. The task is to find the number of elements less than root.

Examples:

```
Input: preorder[] = {3, 2, 1, 0, 5, 4, 6}
Output: 3

Input: preorder[] = {5, 4, 3, 2, 1}
Output: 4
```

For a binary search tree, a preorder traversal is of the form:

> root, { elements in left subtree of root }, { elements in right subtree of root }

Simple approach:

1. Traverse the given preorder.
2. Check if the current element is greater than root.
3. If yes then return the **indexOfCurrentElement − 1** as the no. elements smaller than root will be all the elements occurs before the currentelement except root.

C++

```
 // C++ implementation of above approach
#include <iostream>
using namespace std;

// Function to find the first index of the element
// that is greater than the root
int findLargestIndex(int arr[], int n)
{
    int i, root = arr[0];

    // Traverse the given preorder
    for(i = 0; i < n-1; i++)
    {
        // Check if the number is greater than root
        // If yes then return that index-1
        if(arr[i] > root)
           return i-1;
    }

}

// Driver Code
int main()
{
    int preorder[] = {3, 2, 1, 0, 5, 4, 6};
    int n = sizeof(preorder) / sizeof(preorder[0]);

     cout << findLargestIndex(preorder, n);

    return 0;
}
```

Java

```
 // Java implementation of
// above approach

class GFG
{
// Function to find the first
// index of the element that
// is greater than the root
static int findLargestIndex(int arr[],
                            int n)
{
    int i, root = arr[0];

    // Traverse the given preorder
```

```
    for(i = 0; i < n - 1; i++)
    {
        // Check if the number is
        // greater than root
        // If yes then return
        // that index-1
        if(arr[i] > root)
        return i-1;
    }
    return 0;
}

// Driver Code
public static void main(String ags[])
{
    int preorder[] = {3, 2, 1, 0, 5, 4, 6};
    int n = preorder.length;

    System.out.println(findLargestIndex(preorder, n));
}
}

// This code is contributed
// by Subhadeep Gupta
```

C#

```
 // C# implementation of above approach
using System;

class GFG
{

// Function to find the first
// index of the element that
// is greater than the root
static int findLargestIndex(int []arr,
                            int n)
{
    int i, root = arr[0];

    // Traverse the given preorder
    for(i = 0; i < n - 1; i++)
    {
        // Check if the number is
        // greater than root. If yes
        // then return that index-1
        if(arr[i] > root)
```

```
        return i - 1;
    }
    return 0;
}

// Driver Code
static public void Main()
{
    int []preorder = {3, 2, 1, 0, 5, 4, 6};
    int n = preorder.Length;

    Console.WriteLine(findLargestIndex(preorder, n));
}
}

// This code is contributed
// by Subhadeep Gupta
```

Output:

```
3
```

Time complexity: O(n)

Efficient approach (Using Binary Search): Here the idea is to make use of an extended form of binary search. The steps are as follows:

1. Go to mid. Check if the element at mid is greater than root. If yes then we recurse on the left half of array.
2. Else if the element at mid is lesser than root and element at mid+1 is greater than root we return mid as our answer.
3. Else we recurse on the right half of array to repeat the above steps.

Below is the implementation of the above idea.
C++

```
 // C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// Function to count the smaller elements
int findLargestIndex(int arr[], int n)
{
    int root = arr[0], lb = 0, ub = n-1;
    while(lb < ub)
```

```
    {

        int mid = (lb + ub)/2;

        // Check if the element at mid
        // is greater than root.
        if(arr[mid] > root)
            ub = mid - 1;
        else
        {
          // if the element at mid is lesser
          //  than root and element at mid+1
          // is greater
           if(arr[mid + 1] > root)
              return mid;
            else lb = mid + 1;
        }
     }
     return lb;
}

// Driver Code
int main()
{
    int preorder[] = {3, 2, 1, 0, 5, 4, 6};
    int n = sizeof(preorder) / sizeof(preorder[0]);

     cout << findLargestIndex(preorder, n);

    return 0;
}
```

Java

```
 // Java implementation
// of above approach
import java.util.*;

class GFG
{

// Function to count the
// smaller elements
static int findLargestIndex(int arr[],
                            int n)
{
    int root = arr[0],
        lb = 0, ub = n - 1;
```

```
    while(lb < ub)
    {

        int mid = (lb + ub) / 2;

        // Check if the element at
        // mid is greater than root.
        if(arr[mid] > root)
            ub = mid - 1;
        else
        {

            // if the element at mid is
            // lesser than root and
            // element at mid+1 is greater
            if(arr[mid + 1] > root)
                return mid;
            else lb = mid + 1;
        }
    }
    return lb;
}

// Driver Code
public static void main(String args[])
{
    int preorder[] = {3, 2, 1, 0, 5, 4, 6};
    int n = preorder.length;

    System.out.println(
           findLargestIndex(preorder, n));
}
}

// This code is contributed by Arnab Kundu
```

Output:

```
3
```

Time Complexity: O(logn)

Improved By : tufan_gupta2000, andrew1234

Source

https://www.geeksforgeeks.org/number-of-elements-smaller-than-root-using-preorder-traversal-of-a-bst/

Chapter 77

Optimal Binary Search Tree DP-24

Optimal Binary Search Tree DP-24 - GeeksforGeeks

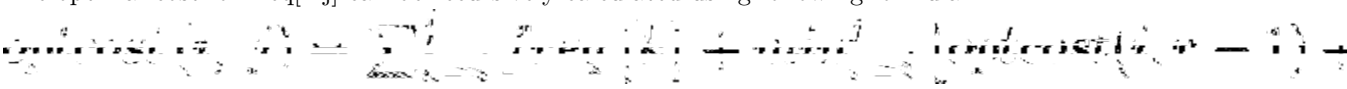
Given a sorted array *keys[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches to *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

```
Example 1
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                       12
          \                     /
           12                 10
          I                     II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118

Example 2
Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10               12                 20         10              20
      \             /   \               /            \            /
      12          10    20            12              20         10
        \                             /               /          \
         20                         10               12           12
     I              II              III             IV             V
```

```
Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

1) Optimal Substructure:
The optimal cost for freq[i..j] can be recursively calculated using following formula.



We need to calculate ***optCost(0, n-1)*** to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.
We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.
C/C++

```
 // A naive recursive implementation of optimal binary
// search tree problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal
// binary search tree
int optCost(int freq[], int i, int j)
{
   // Base cases
   if (j < i)      // no elements in this subarray
     return 0;
   if (j == i)     // one element in this subarray
     return freq[i];

   // Get sum of freq[i], freq[i+1], ... freq[j]
   int fsum = sum(freq, i, j);

   // Initialize minimum value
   int min = INT_MAX;

   // One by one consider all elements as root and
   // recursively find cost of the BST, compare the
   // cost with min and update min if needed
```

```
   for (int r = i; r <= j; ++r)
   {
       int cost = optCost(freq, i, r-1) +
                  optCost(freq, r+1, j);
       if (cost < min)
          min = cost;
   }

   // Return minimum value
   return min + fsum;
}

// The main function that calculates minimum cost of
// a Binary Search Tree. It mainly uses optCost() to
// find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
     // Here array keys[] is assumed to be sorted in
     // increasing order. If keys[] is not sorted, then
     // add code to sort keys, and rearrange freq[]
     // accordingly.
     return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
                optimalSearchTree(keys, freq, n));
    return 0;
}
```

Java

```
 // A naive recursive implementation of optimal binary
```

```
// search tree problem
public class GFG
{
    // A recursive function to calculate cost of
        // optimal binary search tree
    static int optCost(int freq[], int i, int j)
    {
       // Base cases
       if (j < i)      // no elements in this subarray
         return 0;
       if (j == i)     // one element in this subarray
         return freq[i];

       // Get sum of freq[i], freq[i+1], ... freq[j]
       int fsum = sum(freq, i, j);

       // Initialize minimum value
       int min = Integer.MAX_VALUE;

       // One by one consider all elements as root and
           // recursively find cost of the BST, compare the
           // cost with min and update min if needed
       for (int r = i; r <= j; ++r)
       {
           int cost = optCost(freq, i, r-1) +
                          optCost(freq, r+1, j);
           if (cost < min)
              min = cost;
       }

       // Return minimum value
       return min + fsum;
    }

    // The main function that calculates minimum cost of
        // a Binary Search Tree. It mainly uses optCost() to
        // find the optimal cost.
    static int optimalSearchTree(int keys[], int freq[], int n)
    {
         // Here array keys[] is assumed to be sorted in
             // increasing order. If keys[] is not sorted, then
             // add code to sort keys, and rearrange freq[]
             // accordingly.
         return optCost(freq, 0, n-1);
    }

    // A utility function to get sum of array elements
        // freq[i] to freq[j]
```

```
    static int sum(int freq[], int i, int j)
    {
        int s = 0;
        for (int k = i; k <=j; k++)
           s += freq[k];
        return s;
    }

    // Driver code
    public static void main(String[] args) {
        int keys[] = {10, 12, 20};
        int freq[] = {34, 8, 50};
        int n = keys.length;
        System.out.println("Cost of Optimal BST is " +
                           optimalSearchTree(keys, freq, n));
    }
}
// This code is contributed by Sumit Ghosh
```

C#

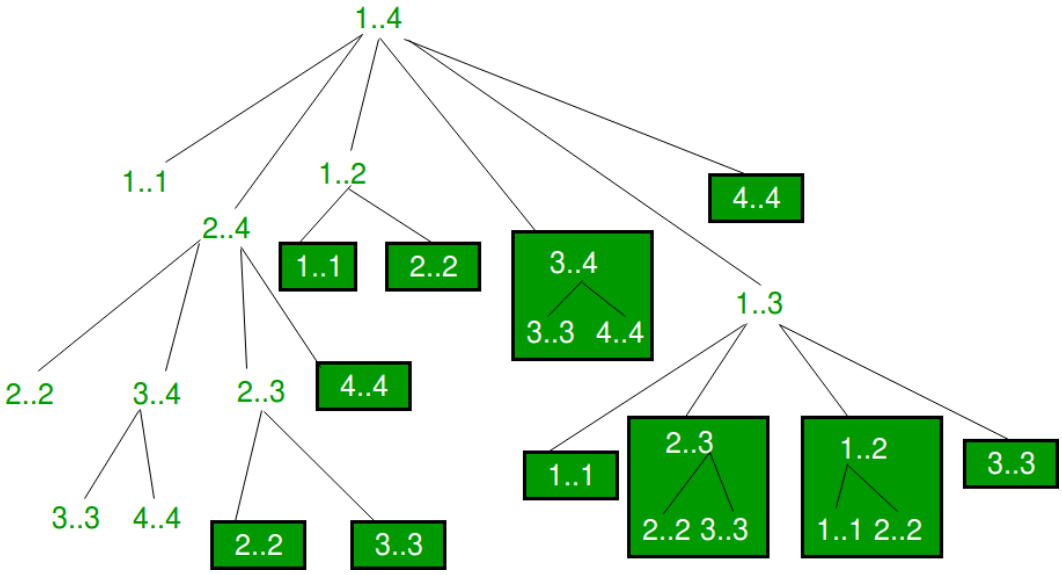
```
 // A naive recursive implementation of optimal binary
// search tree problem
using System;

class GFG
{
    // A recursive function to calculate cost of
    // optimal binary search tree
    static int optCost(int []freq, int i, int j)
    {

    // Base cases
    // no elements in this subarray
    if (j < i)
        return 0;

    // one element in this subarray
    if (j == i)
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = sum(freq, i, j);

    // Initialize minimum value
    int min = int.MaxValue;

    // One by one consider all elements as root and
```

```
    // recursively find cost of the BST, compare the
    // cost with min and update min if needed
    for (int r = i; r <= j; ++r)
    {
        int cost = optCost(freq, i, r-1) +
                        optCost(freq, r+1, j);
        if (cost < min)
            min = cost;
    }

    // Return minimum value
    return min + fsum;
    }

    // The main function that calculates minimum cost of
    // a Binary Search Tree. It mainly uses optCost() to
    // find the optimal cost.
    static int optimalSearchTree(int []keys, int []freq, int n)
    {
        // Here array keys[] is assumed to be sorted in
        // increasing order. If keys[] is not sorted, then
        // add code to sort keys, and rearrange freq[]
        // accordingly.
        return optCost(freq, 0, n-1);
    }

    // A utility function to get sum of array elements
    // freq[i] to freq[j]
    static int sum(int []freq, int i, int j)
    {
        int s = 0;
        for (int k = i; k <=j; k++)
        s += freq[k];
        return s;
    }

    // Driver code
    public static void Main()
    {
        int []keys = {10, 12, 20};
        int []freq = {34, 8, 50};
        int n = keys.Length;
        Console.Write("Cost of Optimal BST is " +
                        optimalSearchTree(keys, freq, n));
    }
}

// This code is contributed by Sam007
```

Output:

```
Cost of Optimal BST is 142
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].



Since same suproblems are called again, this problem has Overlapping Subprolems property. So optimal BST problem has both properties (see thisand this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom up manner.

Dynamic Programming Solution
Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner> The idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

C/C++

```
 // Dynamic Programming code for Optimal Binary Search
// Tree Problem
#include <stdio.h>
#include <limits.h>
```

```
// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates
  minimum cost of a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results
      of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree
       that can be  formed from keys[i] to keys[j].
       cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and
            // chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
               // c = cost when keys[r] becomes root of this subtree
               int c = ((r > i)? cost[i][r-1]:0) +
                       ((r < j)? cost[r+1][j]:0) +
                       sum(freq, i, j);
               if (c < cost[i][j])
                  cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements
```

```
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
                 optimalSearchTree(keys, freq, n));
    return 0;
}
```

Java

```
 // Dynamic Programming Java code for Optimal Binary Search
// Tree Problem
public class Optimal_BST2 {

    /* A Dynamic Programming based function that calculates
        minimum cost of a Binary Search Tree.  */
    static int optimalSearchTree(int keys[], int freq[], int n) {

        /* Create an auxiliary 2D matrix to store results of
           subproblems */
        int cost[][] = new int[n + 1][n + 1];

        /* cost[i][j] = Optimal cost of binary search tree that
           can be formed from keys[i] to keys[j]. cost[0][n-1]
           will store the resultant cost */

        // For a single key, cost is equal to frequency of the key
        for (int i = 0; i < n; i++)
            cost[i][i] = freq[i];

        // Now we need to consider chains of length 2, 3, ... .
        // L is chain length.
        for (int L = 2; L <= n; L++) {

            // i is row number in cost[][]
            for (int i = 0; i <= n - L + 1; i++) {
```

```
                // Get column number j from row number i and
                // chain length L
                int j = i + L - 1;
                cost[i][j] = Integer.MAX_VALUE;

                // Try making all keys in interval keys[i..j] as root
                for (int r = i; r <= j; r++) {

                    // c = cost when keys[r] becomes root of this subtree
                    int c = ((r > i) ? cost[i][r - 1] : 0)
                            + ((r < j) ? cost[r + 1][j] : 0) + sum(freq, i, j);
                    if (c < cost[i][j])
                        cost[i][j] = c;
                }
            }
        }
        return cost[0][n - 1];
    }

    // A utility function to get sum of array elements
    // freq[i] to freq[j]
    static int sum(int freq[], int i, int j) {
        int s = 0;
        for (int k = i; k <= j; k++) {
            if (k >= freq.length)
                continue;
            s += freq[k];
        }
        return s;
    }

    public static void main(String[] args) {

        int keys[] = { 10, 12, 20 };
        int freq[] = { 34, 8, 50 };
        int n = keys.length;
        System.out.println("Cost of Optimal BST is "
                + optimalSearchTree(keys, freq, n));
    }

}
//This code is contributed by Sumit Ghosh
```

C#

```
 // Dynamic Programming C# code for Optimal Binary Search
// Tree Problem
```

```
using System;

class GFG
{
    /* A Dynamic Programming based function that calculates
    minimum cost of a Binary Search Tree. */
    static int optimalSearchTree(int []keys, int []freq, int n) {

        /* Create an auxiliary 2D matrix to store results of
        subproblems */
        int [,]cost = new int[n + 1,n + 1];

        /* cost[i][j] = Optimal cost of binary search tree that
        can be formed from keys[i] to keys[j]. cost[0][n-1]
        will store the resultant cost */

        // For a single key, cost is equal to frequency of the key
        for (int i = 0; i < n; i++)
            cost[i,i] = freq[i];

        // Now we need to consider chains of length 2, 3, ... .
        // L is chain length.
        for (int L = 2; L <= n; L++) {

            // i is row number in cost[][]
            for (int i = 0; i <= n - L + 1; i++) {

                // Get column number j from row number i and
                // chain length L
                int j = i + L - 1;
                cost[i,j] = int.MaxValue;

                // Try making all keys in interval keys[i..j] as root
                for (int r = i; r <= j; r++) {

                    // c = cost when keys[r] becomes root of this subtree
                    int c = ((r > i) ? cost[i,r - 1] : 0)
                            + ((r < j) ? cost[r + 1,j] : 0) + sum(freq, i, j);
                    if (c < cost[i,j])
                        cost[i,j] = c;
                }
            }
        }
        return cost[0,n - 1];
    }

    // A utility function to get sum of array elements
    // freq[i] to freq[j]
```

```
    static int sum(int []freq, int i, int j) {
        int s = 0;
        for (int k = i; k <= j; k++) {
            if (k >= freq.Length)
                continue;
            s += freq[k];
        }
        return s;
    }

    public static void Main() {

        int []keys = { 10, 12, 20 };
        int []freq = { 34, 8, 50 };
        int n = keys.Length;
        Console.Write("Cost of Optimal BST is "
                + optimalSearchTree(keys, freq, n));
    }
}
// This code is contributed by Sam007
```

Output:

```
Cost of Optimal BST is 142
```

Notes
1) The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.

2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen 'r' in the innermost loop.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : aradhya95

Source

https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/

Chapter 78

Overview of Data Structures Set 2 (Binary Tree, BST, Heap and Hash)

Overview of Data Structures Set 2 (Binary Tree, BST, Heap and Hash) - GeeksforGeeks

We have discussed Overview of Array, Linked List, Queue and Stack. In this article following Data Structures are discussed.

5. Binary Tree
6. Binary Search Tree
7. Binary Heap
9. Hashing

Binary Tree
Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.
A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

Binary Tree Representation: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

A Binary Tree can be traversed in two ways:
Depth First Traversal: Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)
Breadth First Traversal: Level Order Traversal

Binary Tree Properties:

```
The maximum number of nodes at level 'l' = 2l-1.

Maximum number of nodes = 2h - 1.
Here h is height of a tree. Height is considered
as is maximum number of nodes on root to leaf path

Minimum possible height =  ceil(Log2(n+1))

In Binary tree, number of leaf nodes is always one
more than nodes with two children.

Time Complexity of Tree Traversal: O(n)
```

Examples : One reason to use binary tree or tree in general is for the things that form a hierarchy. They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing heirarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

Binary Search Tree
In Binary Search Tree is a Binary Tree with following additional properties:
1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Time Complexities:

```
Search :  O(h)
Insertion : O(h)
Deletion : O(h)
Extra Space : O(n) for pointers

h: Height of BST
n: Number of nodes in BST

If Binary Search Tree is Height Balanced,
then h = O(Log n)

Self-Balancing BSTs such as AVL Tree, Red-Black
Tree and Splay Tree make sure that height of BST
remains O(Log n)
```

BST provide moderate access/search (quicker than Linked List and slower than arrays).
BST provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists).

Examples : Its main use is in search application where data is constantly entering/leaving and data needs to printed in sorted order. For example in implementation in E- commerce

websites where a new product is added or product goes out of stock and all products are lised in sorted order.

Binary Heap
A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

```
Get Minimum in Min Heap: O(1) [Or Get Max in Max Heap]
Extract Minimum Min Heap: O(Log n) [Or Extract Max in Max Heap]
Decrease Key in Min Heap: O(Log n)  [Or Extract Max in Max Heap]
Insert: O(Log n)
Delete: O(Log n)
```

Example : Used in implementing efficient priority-queues, which in turn are used for scheduling processes in operating systems. Priority Queues are also used in Dijstra's and Prim's graph algorithms.
The Heap data structure can be used to efficiently find the k smallest (or largest) elements in an array, merging k sorted arrays, median of a stream, etc.
Heap is a special data structure and it cannot be used for searching of a particular element.

HashingHash Function: A function that converts a given big input key to a small practical integer value. The mapped integer value is used as an index in hash table. A good hash function should have following properties
1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

Chaining:The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

```
Space : O(n)
Search    : O(1) [Average]    O(n) [Worst case]
Insertion : O(1) [Average]    O(n) [Worst Case]
Deletion  : O(1) [Average]    O(n) [Worst Case]
```

Hashing seems better than BST for all the operations. But in hashing, elements are unordered and in BST elements are stored in an ordered manner. Also BST is easy to implement but hash functions can sometimes be very complex to generate. In BST, we can also efficiently find floor and ceil of values.

Example : Hashing can be used to remove duplicates from a set of elements. Can also be used find frequency of all items. For example, in web browsers, we can check visited urls using hashing. In firewalls, we can use hashing to detect spam. We need to hash IP addresses. Hashing can be used in any situation where want search() insert() and delete() in O(1) time.

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Rahul1421

Source

https://www.geeksforgeeks.org/overview-of-data-structures-set-2-binary-tree-bst-heap-and-hash/

Chapter 79

Print BST keys in given Range O(1) Space

Print BST keys in given Range O(1) Space - GeeksforGeeks

Given two values n1 and n2 (where n1 < n2) and a root pointer to a Binary Search Tree. Print all the keys of tree in range n1 to n2. i.e. print all nodes n such that n1<=n<=n2 and n is a key of given BST. Print all the keys in increasing order.

Prerequisites : Morris traversal Threaded binary trees

Inorder traversal uses recursion or stack/queue which consumes O(n) space. But there is one efficient way to do inorder tree traversal using Morris Traversal which is based in Threaded Binary trees. Morris traversal uses no recursion or stack/queue and simply stores some important information in the wasted NULL pointers. Morris traversal consumes constant extra memory O(1) as it uses no recursion or stack/queue. Hence we will use Morris traversal to do inorder traversal in the algorithm presented in this tutorial to print keys of a BST in a given range, which is efficient memory wise.

The concept of Threaded Binary trees is simple that they store some useful information in the wasted NULL pointers. In a normal binary tree with n nodes, n+1 NULL pointers waste memory.

Approach : Morris Traversal is a very nice memory efficient technique to do tree traversal without using stack or recursion in constant memory O(1) based on Threaded Binary Trees. Morris traversal can be used in solving problems where inorder tree traversals are used especially in **order statistics** eg-Kth largest element in BST, Kth smallest in BST etc. Hence, this is where Morris traversal would come handy as a more efficient method to do inorder traversal in constant O(1) space without using any stack or recursion.

Algorithm

```
1) Initialize Current as root.
```

```
2) While current is not NULL :

  2.1) If current has no left child

   a) Check if current lies between n1 and n2.
      1)If so, then visit the current node.

   b)Otherwise, Move to the right child of current.

  3) Else, here we have 2 cases:
   a) Find the inorder predecessor of current node.
      Inorder predecessor is the right most node
      in the left subtree or left child itself.

   b) If the right child of the inorder predecessor is NULL:
      1) Set current as the right child of its inorder predecessor.
      2) Move current node to its left child.

   c) Else, if the threaded link between the current node
      and it's inorder predecessor already exists :
      1) Set right pointer of the inorder predecessor as NULL.
      2) Again check if current node lies between n1 and n2.
        a)If so, then visit the current node.

      3)Now move current to it's right child.
```

Below is the implementation of above approach.

```
 // CPP code to print BST keys in given Range in
// constant space using Morris traversal.
#include <iostream>

using namespace std;

struct node {

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

// Function to print the keys in range
void RangeTraversal(node* root,
                    int n1, int n2)
{
    if (!root)
        return;

    node* curr = root;
```

```
while (curr) {

    if (curr->left == NULL)
    {
        // check if current node
        // lies between n1 and n2
        if (curr->data <= n2 &&
            curr->data >= n1)
        {
            cout << curr->data << " ";
        }

        curr = curr->right;
    }

    else {
        node* pre = curr->left;
        // finding the inorder predecessor-
        // inorder predecessor is the right
        // most in left subtree or the left
        // child, i.e in BST it is the
        // maximum(right most) in left subtree.
        while (pre->right != NULL &&
               pre->right != curr)
                    pre = pre->right;

        if (pre->right == NULL)
        {
            pre->right = curr;
            curr = curr->left;
        }

        else {
            pre->right = NULL;

            // check if current node lies
            // between n1 and n2
            if (curr->data <= n2 &&
                curr->data >= n1)
            {
                cout << curr->data << " ";
            }

            curr = curr->right;
        }
    }
}
```

```
}

// Helper function to create a new node
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->right = temp->left = NULL;

    return temp;
}

// Driver Code
int main()
{

    /* Constructed binary tree is
          4
        /   \
       2     7
     /  \   /  \
    1    3  6    10
*/

    node* root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
    root->right->left = newNode(6);
    root->right->right = newNode(10);

    RangeTraversal(root, 4, 12);

    return 0;
}
```

Output:

```
4 6 7 10
```

Time Complexity : O(n)
Auxiliary Space : O(1)

Source

https://www.geeksforgeeks.org/print-bst-keys-in-given-range-o1-space/

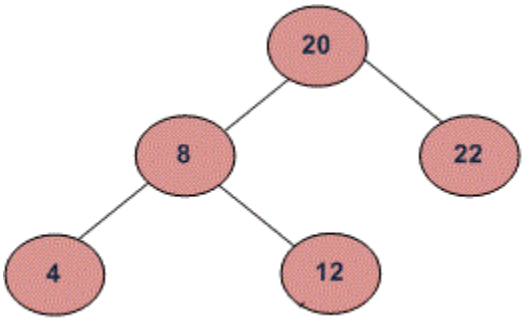
Chapter 80

Print BST keys in the given range

Print BST keys in the given range - GeeksforGeeks

Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Print all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Print all the keys in increasing order.

For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.



Algorithm:
1) If value of root's key is greater than k1, then recursively call in left subtree.
2) If value of root's key is in range, then print the root's key.
3) If value of root's key is smaller than k2, then recursively call in right subtree.

Implementation:
C

```
 #include<stdio.h>

/* A tree node structure */
struct node
```

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

/* The functions prints all the keys which in the given range [k1..k2].
    The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
   /* base case */
   if ( NULL == root )
      return;

   /* Since the desired o/p is sorted, recurse for left subtree first
      If root->data is greater than k1, then only we can get o/p keys
      in left subtree */
   if ( k1 < root->data )
     Print(root->left, k1, k2);

   /* if root's data lies in range, then prints root's data */
   if ( k1 <= root->data && k2 >= root->data )
     printf("%d ", root->data );

  /* If root->data is smaller than k2, then only we can get o/p keys
      in right subtree */
   if ( k2 > root->data )
     Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int k1 = 10, k2 = 25;

  /* Constructing tree given in the above figure */
```

```
  root = newNode(20);
  root->left = newNode(8);
  root->right = newNode(22);
  root->left->left = newNode(4);
  root->left->right = newNode(12);

  Print(root, k1, k2);

  getchar();
  return 0;
}
```

Java

```
 // Java program to print BST in given range

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node root;

    /* The functions prints all the keys which in the given range [k1..k2].
     The function assumes than k1 < k2 */
    void Print(Node node, int k1, int k2) {

        /* base case */
        if (node == null) {
            return;
        }

        /* Since the desired o/p is sorted, recurse for left subtree first
         If root->data is greater than k1, then only we can get o/p keys
         in left subtree */
        if (k1 < node.data) {
            Print(node.left, k1, k2);
        }
```

```
        /* if root's data lies in range, then prints root's data */
        if (k1 <= node.data && k2 >= node.data) {
            System.out.print(node.data + " ");
        }

        /* If root->data is smaller than k2, then only we can get o/p keys
         in right subtree */
        if (k2 > node.data) {
            Print(node.right, k1, k2);
        }
    }


    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int k1 = 10, k2 = 25;
        tree.root = new Node(20);
        tree.root.left = new Node(8);
        tree.root.right = new Node(22);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(12);

        tree.Print(root, k1, k2);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to find BST keys in given range

# A binary tree node
class Node:

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

# The function prints all the keys in the gicven range
# [k1..k2]. Assumes that k1 < k2
def Print(root, k1, k2):

    # Base Case
    if root is None:
        return
```

```
    # Since the desired o/p is sorted, recurse for left
    # subtree first. If root.data is greater than k1, then
    # only we can get o/p keys in left subtree
    if k1 < root.data :
        Print(root.left, k1, k2)

    # If root's data lies in range, then prints root's data
    if k1 <= root.data and k2 >= root.data:
        print root.data,

    # If root.data is smaller than k2, then only we can get
    # o/p keys in right subtree
    if k2 > root.data:
        Print(root.right, k1, k2)

# Driver function to test above function
k1 = 10 ; k2 = 25 ;
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)

Print(root, k1, k2)

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

Output:

```
12 20 22
```

Time Complexity: O(n) where n is the total number of keys in tree.

Source

https://www.geeksforgeeks.org/print-bst-keys-in-the-given-range/

Chapter 81

Print Common Nodes in Two Binary Search Trees

Print Common Nodes in Two Binary Search Trees - GeeksforGeeks

Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Example:

```
Input: root1  (Root of below tree)
           5
         /   \
        1     10
       / \    /
      0   4  7
              \
               9

      root2 (Root of below tree)
          10
        /   \
       7     20
      / \
     4   9

Output: 4 7 9 10
```

Method 1 (Simple Solution) A simple way is to one by once search every node of first tree in second tree. Time complexity of this solution is O(m * h) where m is number of nodes in first tree and h is height of second tree.

Method 2 (Linear Time) We can find common elements in O(n) time.
1) Do inorder traversal of first tree and store the traversal in an auxiliary array ar1[]. See

sortedInorder() here.
2) Do inorder traversal of second tree and store the traversal in an auxiliary array ar2[]
3) Find intersection of ar1[] and ar2[]. See thisfor details.

Time complexity of this method is O(m+n) where m and n are number of nodes in first and second tree respectively. This solution requires O(m+n) extra space.

Method 3 (Linear Time and limited Extra Space) We can find common elements in O(n) time and O(h1 + h2) extra space where h1 and h2 are heights of first and second BSTs respectively.
The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

```
 // Iterative traversal based method to find common elements
// in two BSTs.
#include<iostream>
#include<stack>
using namespace std;

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

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

// Function two print common elements in given two trees
void printCommon(Node *root1, Node *root2)
{
    // Create two stacks for two inorder traversals
    stack<Node *> stack1, s1, s2;

    while (1)
    {
        // push the Nodes of first tree in stack s1
        if (root1)
        {
            s1.push(root1);
            root1 = root1->left;
        }
```

```
// push the Nodes of second tree in stack s2
else if (root2)
{
    s2.push(root2);
    root2 = root2->left;
}

// Both root1 and root2 are NULL here
else if (!s1.empty() && !s2.empty())
{
    root1 = s1.top();
    root2 = s2.top();

    // If current keys in two trees are same
    if (root1->key == root2->key)
    {
        cout << root1->key << " ";
        s1.pop();
        s2.pop();

        // move to the inorder successor
        root1 = root1->right;
        root2 = root2->right;
    }

    else if (root1->key < root2->key)
    {
        // If Node of first tree is smaller, than that of
        // second tree, then its obvious that the inorder
        // successors of current Node can have same value
        // as that of the second tree Node. Thus, we pop
        // from s2
        s1.pop();
        root1 = root1->right;

        // root2 is set to NULL, because we need
        // new Nodes of tree 1
        root2 = NULL;
    }
    else if (root1->key > root2->key)
    {
        s2.pop();
        root2 = root2->right;
        root1 = NULL;
    }
}
```

```
        // Both roots and both stacks are empty
        else  break;
    }
}

// A utility function to do inorder traversal
void inorder(struct Node *root)
{
    if (root)
    {
        inorder(root->left);
        cout<<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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver program
int main()
{
    // Create first tree as shown in example
    Node *root1 = NULL;
    root1 = insert(root1, 5);
    root1 = insert(root1, 1);
    root1 = insert(root1, 10);
    root1 = insert(root1,  0);
    root1 = insert(root1,  4);
    root1 = insert(root1,  7);
    root1 = insert(root1,  9);

    // Create second tree as shown in example
    Node *root2 = NULL;
    root2 = insert(root2, 10);
```

```
    root2 = insert(root2, 7);
    root2 = insert(root2, 20);
    root2 = insert(root2, 4);
    root2 = insert(root2, 9);

    cout << "Tree 1 : ";
    inorder(root1);
    cout << endl;

    cout << "Tree 2 : ";
    inorder(root2);

    cout << "\nCommon Nodes: ";
    printCommon(root1, root2);

    return 0;
}
```

Output:

```
4 7 9 10
```

This article is contributed by **Ekta Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/print-common-nodes-in-two-binary-search-trees/

Chapter 82

Rank of an element in a stream

Rank of an element in a stream - GeeksforGeeks

Given a stream of integers, lookup the rank of a given integer x. Rank of an integer in stream is "Total number of elements less than or equal to x (not including x)".

If element is not found in stream or is smallest in stream, return -1.
Examples:

```
Input :  arr[] = {10, 20, 15, 3, 4, 4, 1}
               x = 4;
Output : Rank of 4 in stream is: 3
There are total three elements less than
or equal to x (and not including x)

Input : arr[] = {5, 1, 14, 4, 15, 9, 7, 20, 11},
           x = 20;
Output : Rank of 20 in stream is: 8
```

A relatively easy way to implement this is to use an Array that holds all the elements in sorted order. When a new element is inserted we would shift the elements. Then we perform binary search on the array to get right most index of x and return that index. getRank(x) would work in O(log n) but insertion would be costly.

An **efficient way** is to use a Binary Search Tree. Each Node will hold the data value and size of its left subtree.

We traverse the tree from root and compare the root values to x.

1. If root->data == x, return size of left subtree of root.
2. If x data, return getRank(root->left)
3. If x > root->data, return getRank(root->right) + size of letSubtree + 1.

Below is C++ solution.

```
 // CPP program to find rank of an
// element in a stream.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
    int leftSize;
};

Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->leftSize = 0;
    return temp;
}

// Inserting a new Node.
Node* insert(Node*& root, int data)
{
    if (!root)
        return newNode(data);

    // Updating size of left subtree.
    if (data <= root->data) {
        root->left = insert(root->left, data);
        root->leftSize++;
    }
    else
        root->right = insert(root->right, data);

    return root;
}

// Function to get Rank of a Node x.
int getRank(Node* root, int x)
{
    // Step 1.
    if (root->data == x)
        return root->leftSize;

    // Step 2.
    if (x < root->data) {
```

```
        if (!root->left)
            return -1;
        else
            return getRank(root->left, x);
    }

    // Step 3.
    else {
        if (!root->right)
            return -1;
        else {
            int rightSize = getRank(root->right, x);
            return root->leftSize + 1 + rightSize;
        }
    }
}

// Driver code
int main()
{
    int arr[] = { 5, 1, 4, 4, 5, 9, 7, 13, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 4;

    Node* root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, arr[i]);

    cout << "Rank of " << x << " in stream is: "
         << getRank(root, x) << endl;

    x = 13;
    cout << "Rank of " << x << " in stream is: "
         << getRank(root, x) << endl;

    return 0;
}
```

Output:

```
Rank of 4 in stream is: 3
Rank of 13 in stream is: 8
```

Source

https://www.geeksforgeeks.org/rank-element-stream/

Chapter 83

Reallocation of elements based on Locality of Reference

Reallocation of elements based on Locality of Reference - GeeksforGeeks

Consider a problem where same elements are likely to be searched again and again. Implement search operation efficiently.

Examples :

```
Input : arr[] = {12 25 36 85 98 75 89 15 63 66
                                64 74 27 83 97}
          q[] = {63, 63, 86, 63, 78}
Output : Yes Yes No Yes No
We need one by one search items of q[] in arr[].
The element 63 is present, 78 and 86 are not present.
```

The idea is **simple**, we move the searched element to front of the array so that it can be searched quickly next time.

```
 // C++ program to implement search for an item
// that is searched again and again.
#include <bits/stdc++.h>
using namespace std;

// A function to perform sequential search.
bool search(int arr[], int n, int x)
{
    // Linearly search the element
    int res = -1;
    for (int i = 0; i < n; i++)
```

```
        if (x == arr[i])
           res = i;

    // If not found
    if (res == -1)
        return false;

    // Shift elements before one position
    int temp = arr[res];
    for (int i = res; i > 0; i--)
        arr[i] = arr[i - 1];

    arr[0] = temp;
    return true;
}

// Driver Code
int main()
{
    int arr[] = { 12, 25, 36, 85, 98, 75, 89, 15,
                  63, 66, 64, 74, 27, 83, 97 };
    int q[] = {63, 63, 86, 63, 78};
    int n = size(arr)/sizeof(arr[0]);
    int m = sizeof(q)/sizeof(q[0]);
    for (int i=0; i<m; i++)
      search(arr, n, q[i]?  cout << "Yes "
                         :  cout << "No  ";

    return 0;
}
```

Further Thoughts : We can do better by using a linked list. In linked list, moving an item to front can be done in O(1) time.
The best solution would be to use Splay Tree (a data structure designed for this purpose). Splay tree supports insert, search and delete operations in O(Log n) time on average. Also, splay tree is a BST, so we can quickly print elements in sorted order.

Source

https://www.geeksforgeeks.org/reallocation-of-elements-based-on-locality-of-reference/

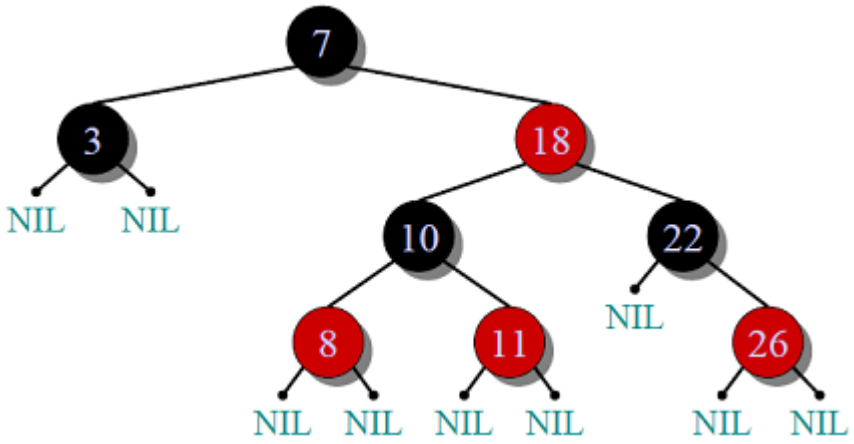
Chapter 84

Red Black Tree vs AVL Tree

Red Black Tree vs AVL Tree - GeeksforGeeks

In this post we will compare Red Black Tree and AVL Tree.

Red Black Tree:



Properties:

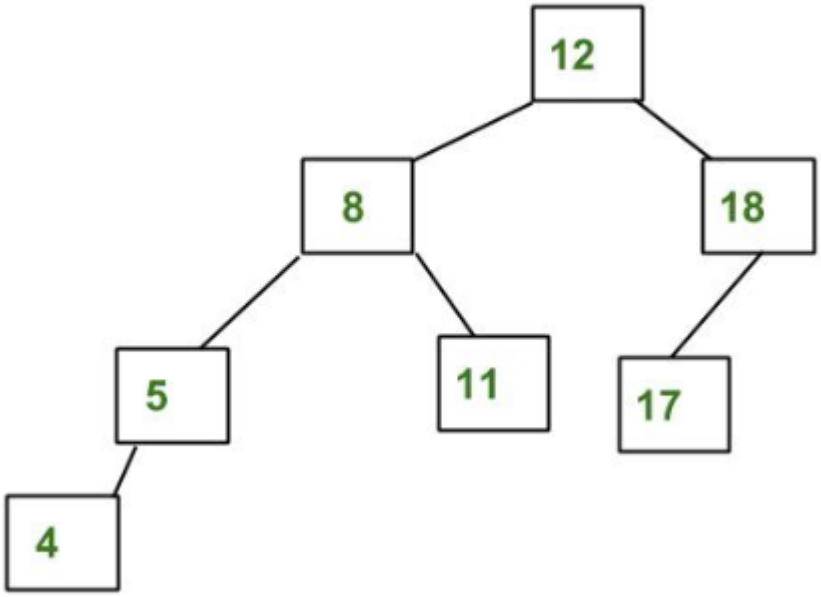
1. Self-Balancing is provided by painting each node with one two colors(Red or Black).
2. When Tree is modified, new tree is subsequently rearranged and repainted.
3. It requires 1 bit of color information for each node in tree.

***Constraints* maintained by Red Black Tree**:

1. Root is always black.
2. All NULL leaves are black, both children of red node are black and vice-versa.

3. Every simple path from a given node to any of its descendant leaves contains the same number of black
nodes.
4. Path form root to farthest leaf is no more than twice as long as path from root to nearest leaf.

AVL(Adelson-Velskii and Landis) Tree



Properties:

1. Height difference of left and right subtree of node should be less than 2.
2. Re-balancing is done when heights of two child subtrees of a node differ by more than one.
3. Faster retrievals as strictly balanced.

***Difference*:**

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** for each node, thus requires **O(N) extra space** whereas Red Black Tree requires only 1 bit of information per node, thus require **O(1) extra space**.

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.

Source

https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/

Chapter 85

Red-Black Tree Set 2 (Insert)

Red-Black Tree Set 2 (Insert) - GeeksforGeeks

In the previous post, we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

InAVL 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

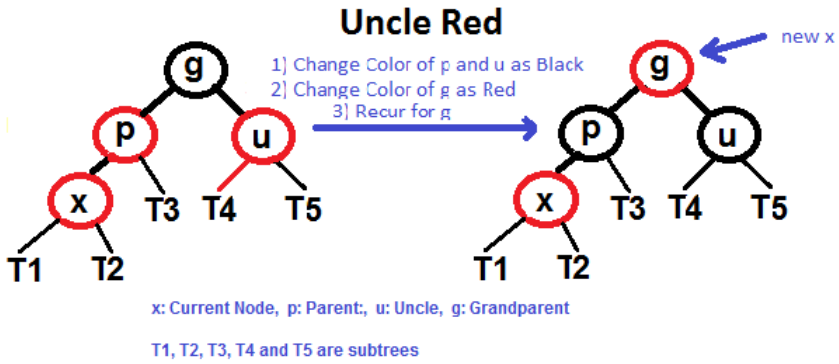
We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

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).

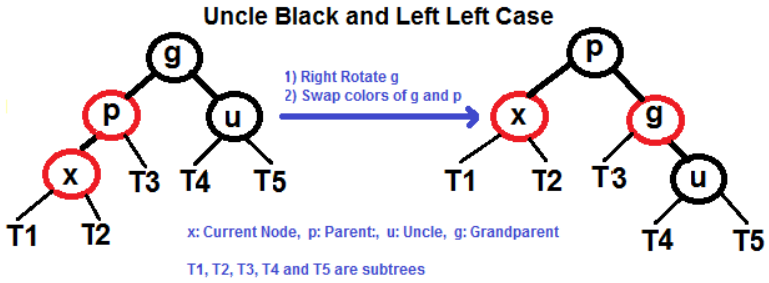
3) Do following if color of x's parent is not BLACK or x is not root.
….**a) If x's uncle is RED** (Grand parent must have been black from property 4)
……..**(i)** Change color of parent and uncle as BLACK.
……..**(ii)** color of grand parent as RED.
……..**(iii)** Change x = x's grandparent, repeat steps 2 and 3 for new x.



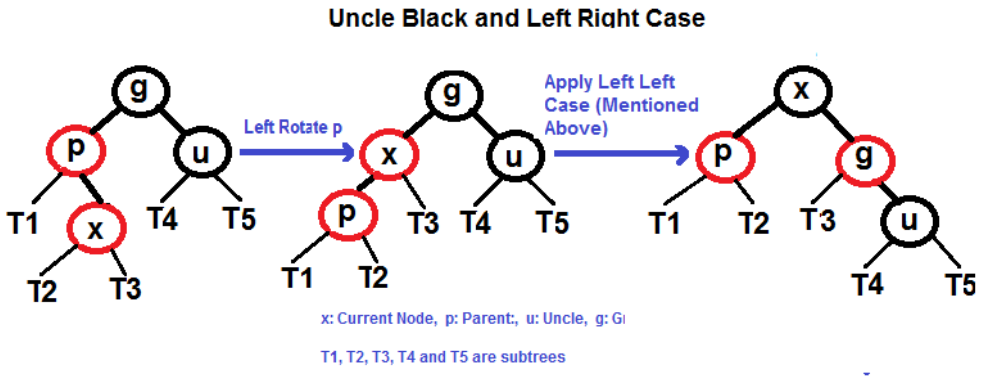
....**b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar toAVL Tree)
.....**i)** Left Left Case (p is left child of g and x is left child of p)
.....**ii)** Left Right Case (p is left child of g and x is right child of p)
.....**iii)** Right Right Case (Mirror of case a)
.....**iv)** Right Left Case (Mirror of case c)

Following are operations to be performed in four subcases when uncle is BLACK.

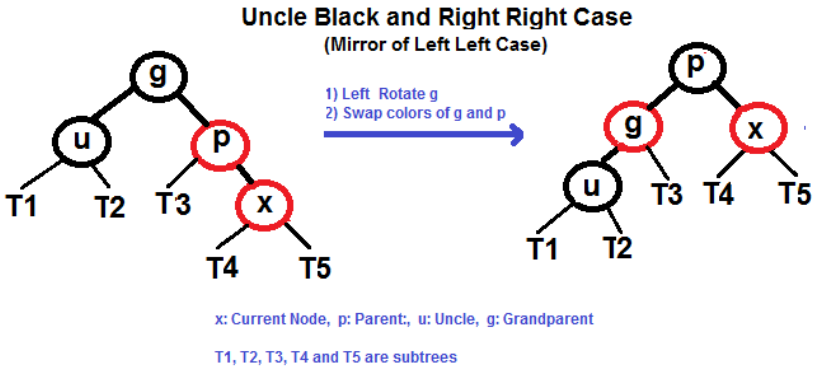
Left Left Case (See g, p and x)



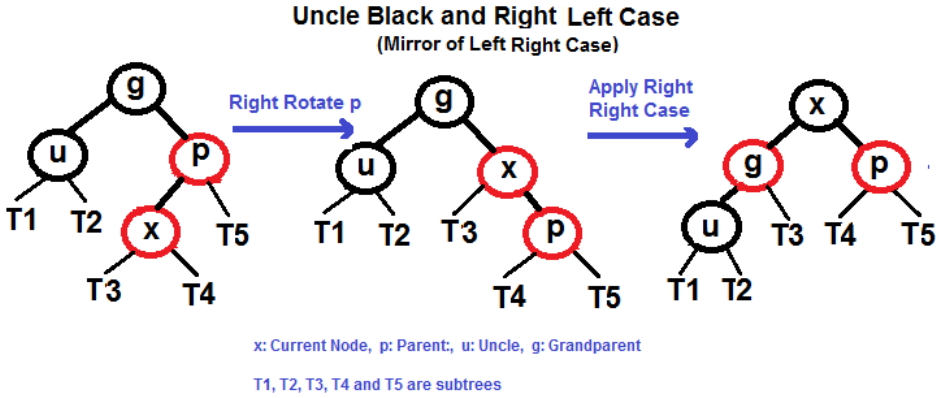
Left Right Case (See g, p and x)



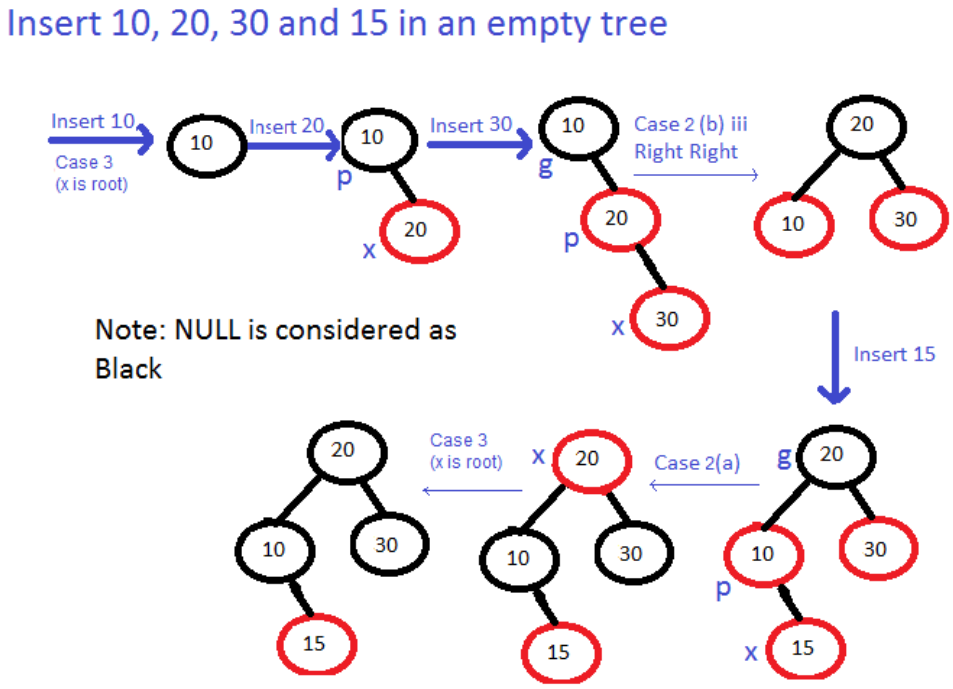
Right Right Case (See g, p and x)



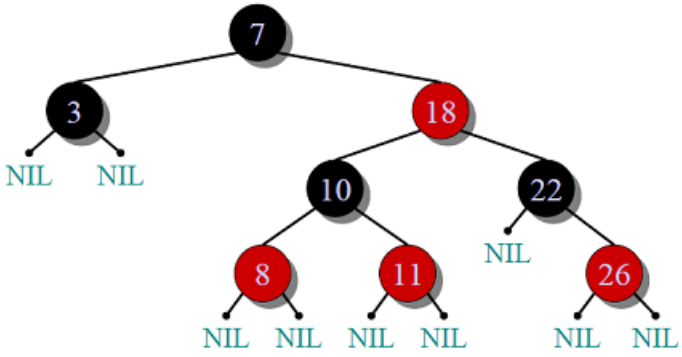
Right Left Case (See g, p and x)



Examples of Insertion



Exercise:
Insert 2, 6 and 13 in below tree.



Please refer **C Program for Red Black Tree Insertion** for complete implementation of above algorithm.

Red-Black Tree Set 3 (Delete)

Source

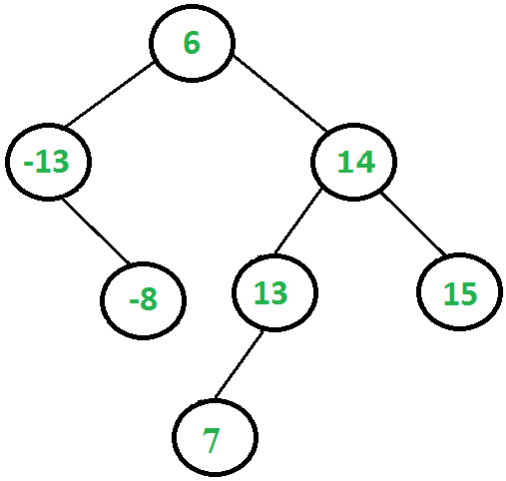
https://www.geeksforgeeks.org/red-black-tree-set-2-insert/

Chapter 86

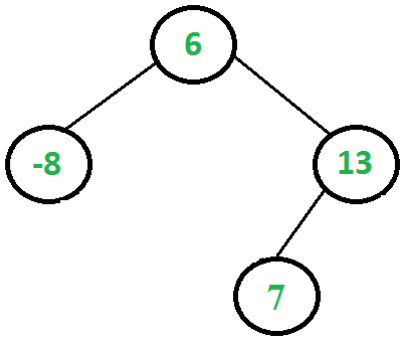
Remove BST keys outside the given range

Remove BST keys outside the given range - GeeksforGeeks

Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].



The given tree should be changed to following. Note that all keys outside the range [-10, 13] are removed and modified tree is BST.



There are two possible cases for every node.
1) Node's key is outside the given range. This case has two sub-cases.
……**a)** Node's key is smaller than the min value.
……**b)** Node's key is greater that the max value.
2) Node's key is in range.

We don't need to do anything for case 2. In case 1, we need to remove the node and change root of sub-tree rooted with this node.
The idea is to fix the tree in Postorder fashion. When we visit a node, we make sure that its left and right sub-trees are already fixed. In case 1.a), we simply remove root and return right sub-tree as new root. In case 1.b), we remove root and return left sub-tree as new root.

Following is implementation of the above approach.
C++

```
 // A C++ program to remove BST keys outside the given range
#include<stdio.h>
#include <iostream>

using namespace std;

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// Removes all nodes having value outside the given range and returns the root
// of modified tree
node* removeOutsideRange(node *root, int min, int max)
{
   // Base Case
   if (root == NULL)
      return NULL;
```

```
   // First fix the left and right subtrees of root
   root->left =  removeOutsideRange(root->left, min, max);
   root->right =  removeOutsideRange(root->right, min, max);

   // Now fix the root.  There are 2 possible cases for toot
   // 1.a) Root's key is smaller than min value (root is not in range)
   if (root->key < min)
   {
       node *rChild = root->right;
       delete root;
       return rChild;
   }
   // 1.b) Root's key is greater than max value (root is not in range)
   if (root->key > max)
   {
       node *lChild = root->left;
       delete root;
       return lChild;
   }
   // 2. Root is in range
   return root;
}

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

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

// Utility function to traverse the binary tree after conversion
void inorderTraversal(node* root)
{
```

```
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->key << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);

    cout << "Inorder traversal of the given tree is: ";
    inorderTraversal(root);

    root = removeOutsideRange(root, -10, 13);

    cout << "\nInorder traversal of the modified tree is: ";
    inorderTraversal(root);

    return 0;
}
```

Java

```
 // Write Java code here
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

class Node
{
    int key;
    Node left;
    Node right;
}

class GFG
```

```
{
    // Removes all nodes having value
    // outside the given range and
    // returns the root of modified tree
    private static Node removeOutsideRange(Node root,
                                           int min, int max)
    {
        // BASE CASE
        if(root == null)
        {
            return null;
        }

        // FIRST FIX THE LEFT AND
        // RIGHT SUBTREE OF ROOT
        root.left = removeOutsideRange(root.left,
                                       min, max);
        root.right = removeOutsideRange(root.right,
                                        min, max);

        // NOW FIX THE ROOT. THERE ARE
        // TWO POSSIBLE CASES FOR THE ROOT
        // 1. a) Root's key is smaller than
        // min value(root is not in range)
        if(root.key < min)
        {
            Node rchild = root.right;
            root = null;
            return rchild;
        }

        // 1. b) Root's key is greater than
        // max value (Root is not in range)
        if(root.key > max)
        {
            Node lchild = root.left;
            root = null;
            return lchild;
        }

        // 2. Root in range
        return root;
    }

    public static Node newNode(int num)
    {
        Node temp = new Node();
        temp.key = num;
```

```
    temp.left = null;
    temp.right = null;
    return temp;
}

public static Node insert(Node root,
                          int key)
{
    if(root == null)
    {
        return newNode(key);
    }
    if(root.key > key)
    {
        root.left = insert(root.left, key);
    }
    else
    {
        root.right = insert(root.right, key);
    }
    return root;
}

private static void inorderTraversal(Node root)
{
    if(root != null)
    {
        inorderTraversal(root.left);
        System.out.print(root.key + " ");
        inorderTraversal(root.right);
    }
}

// Driver code
public static void main(String[] args)
{
    Node root = null;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);

    System.out.print("Inorder Traversal of " +
                     "the given tree is: ");
    inorderTraversal(root);
```

```
        root = removeOutsideRange(root, -10, 13);

        System.out.print("\nInorder traversal of " +
                         "the modified tree: ");
        inorderTraversal(root);
    }
}

// This code is contributed
// by Divya
```

Output:

```
Inorder traversal of the given tree is: -13 -8 6 7 13 14 15
Inorder traversal of the modified tree is: -8 6 7 13
```

Time Complexity: O(n) where n is the number of nodes in given BST.

Improved By : 9divya5

Source

https://www.geeksforgeeks.org/remove-bst-keys-outside-the-given-range/

Chapter 87

Remove all leaf nodes from the binary search tree

Remove all leaf nodes from the binary search tree - GeeksforGeeks

We have given a binary search tree and we want to delete the leaf nodes from the binary search tree.

Examples:

```
Input : 20 10 5 15 30 25 35
Output : Inorder before Deleting the leaf node
         5 10 15 20 25 30 35
         Inorder after Deleting the leaf node
         10 20 30

        This is the binary search tree where we
        want to delete the leaf node.
              20
            /    \
           10     30
          /  \    /  \
         5    15 25   35

      After deleting the leaf node the binary
      search tree looks like
             20
           /    \
          10     30
```

We traverse given Binary Search Tree in preorder way. During traversal we check if current node is leaf, if yes, we delete it. Else we recur for left and right children. An important

thing to remember is, we must assign new left and right children if there is any modification in roots of subtrees.

```
 // C++ program to delete leaf Node from
// binary search tree.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* left,
    struct Node* right;
};

// Create a newNode in binary search tree.
struct Node* newNode(int data)
{
    struct Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Insert a Node in binary search tree.
struct Node* insert(struct Node* root, int data)
{
    if (root == NULL)
        return newNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}

// Function for inorder traversal in a BST.
void inorder(struct Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// Delete leaf nodes from binary search tree.
struct Node* leafDelete(struct Node* root)
{
```

```
    if (root->left == NULL && root->right == NULL) {
        free(root);
        return NULL;
    }

    // Else recursively delete in left and right
    // subtrees.
    root->left = leafDelete(root->left);
    root->right = leafDelete(root->right);

    return root;
}

// Driver code
int main()
{
    struct Node* root = NULL;
    root = insert(root, 20);
    insert(root, 10);
    insert(root, 5);
    insert(root, 15);
    insert(root, 30);
    insert(root, 25);
    insert(root, 35);
    cout << "Inorder before Deleting the leaf Node." << endl;
    inorder(root);
    cout << endl;
    leafDelete(root);
    cout << "INorder after Deleting the leaf Node." << endl;
    inorder(root);
    return 0;
}
```

Output:

```
Inorder before Deleting the leaf node.
5 10 15 20 25 30 35
INorder after Deleting the leaf node.
10 20 30
```

Source

https://www.geeksforgeeks.org/remove-leaf-nodes-binary-search-tree/

Chapter 88

Replace every element with the least greater element on its right

Replace every element with the least greater element on its right - GeeksforGeeks

Given an array of integers, replace every element with the least greater element on its right side in the array. If there are no greater element on right side, replace it with -1.

Examples:

```
Input: [8, 58, 71, 18, 31, 32, 63, 92,
         43, 3, 91, 93, 25, 80, 28]
Output: [18, 63, 80, 25, 32, 43, 80, 93,
         80, 25, 93, -1, 28, -1, -1]
```

A naive method is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the smallest element greater than the picked element on its right side. Finally the outer loop will replace the picked element with the element found by inner loop. The time complexity of this method will be $O(n^2)$.

A tricky solution would be to use Binary Search Trees. We start scanning the array from right to left and insert each element into the BST. For each inserted element, we replace it in the array by its inorder successor in BST. If the element inserted is the maximum so far (i.e. its inorder successor doesn't exists), we replace it by -1.

Below is C++ implementation of above idea –

```
 // C++ program to replace every element with the
// least greater element on its right
#include <iostream>
using namespace std;

// A binary Tree node
```

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

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

    return temp;
}

/* A utility function to insert a new node with
   given data in BST and find its successor */
void insert(Node*& node, int data, Node*& succ)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        node = newNode(data);

    // If key is smaller than root's key, go to left
    // subtree and set successor as current node
    if (data < node->data)
    {
        succ = node;
        insert(node->left, data, succ);
    }

    // go to right subtree
    else if (data > node->data)
        insert(node->right, data, succ);
}

// Function to replace every element with the
// least greater element on its right
void replace(int arr[], int n)
{
    Node* root = NULL;

    // start from right to left
    for (int i = n - 1; i >= 0; i--)
    {
        Node* succ = NULL;
```

```
        // insert current element into BST and
        // find its inorder successor
        insert(root, arr[i], succ);

        // replace element by its inorder
        // successor in BST
        if (succ)
            arr[i] = succ->data;
        else    // No inorder successor
            arr[i] = -1;
    }
}

// Driver Program to test above functions
int main()
{
    int arr[] = { 8, 58, 71, 18, 31, 32, 63, 92,
                  43, 3, 91, 93, 25, 80, 28 };
    int n = sizeof(arr)/ sizeof(arr[0]);

    replace(arr, n);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output:

```
18 63 80 25 32 43 80 93 80 25 93 -1 28 -1 -1
```

Worst case time complexity of above solution is also $O(n^2)$ as it uses BST. The worst case will happen when array is sorted in ascending or descending order. The complexity can easily be reduced to O(nlogn) by using balanced trees like red-black trees.

Source

https://www.geeksforgeeks.org/replace-every-element-with-the-least-greater-element-on-its-right/

Chapter 89

Reverse a path in BST using queue

Reverse a path in BST using queue - GeeksforGeeks

Given a binary search tree and a key, your task to reverse path of the binary tree.

Prerequisite : Reverse path of Binary tree

Examples :

```
Input :       50
            /     \
           30      70
         /  \    /  \
       20   40  60   80
k = 70
Output :
Inorder before reversal :
20 30 40 50 60 70 80
Inorder after reversal :
20 30 40 70 60 50 80

Input :       8
            /     \
           3       10
         /  \       \
       1    6         14
           /  \      /
          4    7    13
k = 13
Output :
Inorder before reversal :
```

```
1 3 4 6 7 8 10 13 14
Inorder after reversal :
1 3 4 6 7 13 14 10 8
```

Approach :
Take a queue and push all the element till that given key at the end replace node key with queue front element till root, then print inorder of the tree.

Below is the implementation of above approach :

```
 // CPP code to demonstrate insert
// operation in binary search tree
#include <bits/stdc++.h>
using namespace std;

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 = new 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);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// reverse tree path using queue
void reversePath(struct node** node,
            int& key, queue<int>& q1)
{
    /* If the tree is empty,
    return a new node */
    if (node == NULL)
```

```
    return;

// If the node key equal
// to key then
if ((*node)->key == key)
{
    // push current node key
    q1.push((*node)->key);

    // replace first node
    // with last element
    (*node)->key = q1.front();

    // remove first element
    q1.pop();

    // return
    return;
}

// if key smaller than node key then
else if (key < (*node)->key)
{
    // push node key into queue
    q1.push((*node)->key);

    // recusive call itself
    reversePath(&(*node)->left, key, q1);

    // replace queue front to node key
    (*node)->key = q1.front();

    // performe pop in queue
    q1.pop();
}

// if key greater than node key then
else if (key > (*node)->key)
{
    // push node key into queue
    q1.push((*node)->key);

    // recusive call itself
    reversePath(&(*node)->right, key, q1);

    // replace queue front to node key
    (*node)->key = q1.front();
```

```
        // performe pop in queue
        q1.pop();
    }

    // return
    return;
}

/* 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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node* root = NULL;
    queue<int> q1;

    // reverse path till k
    int k = 80;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
```

```
    insert(root, 80);

    cout << "Before Reverse :" << endl;
    // print inoder traversal of the BST
    inorder(root);

    cout << "\n";

    // reverse path till k
    reversePath(&root, k, q1);

    cout << "After Reverse :" << endl;

    // print inorder of reverse path tree
    inorder(root);

    return 0;
}
```

Output:

```
Before Reverse :
20 30 40 50 60 70 80
After Reverse :
20 30 40 80 60 70 50
```

Source

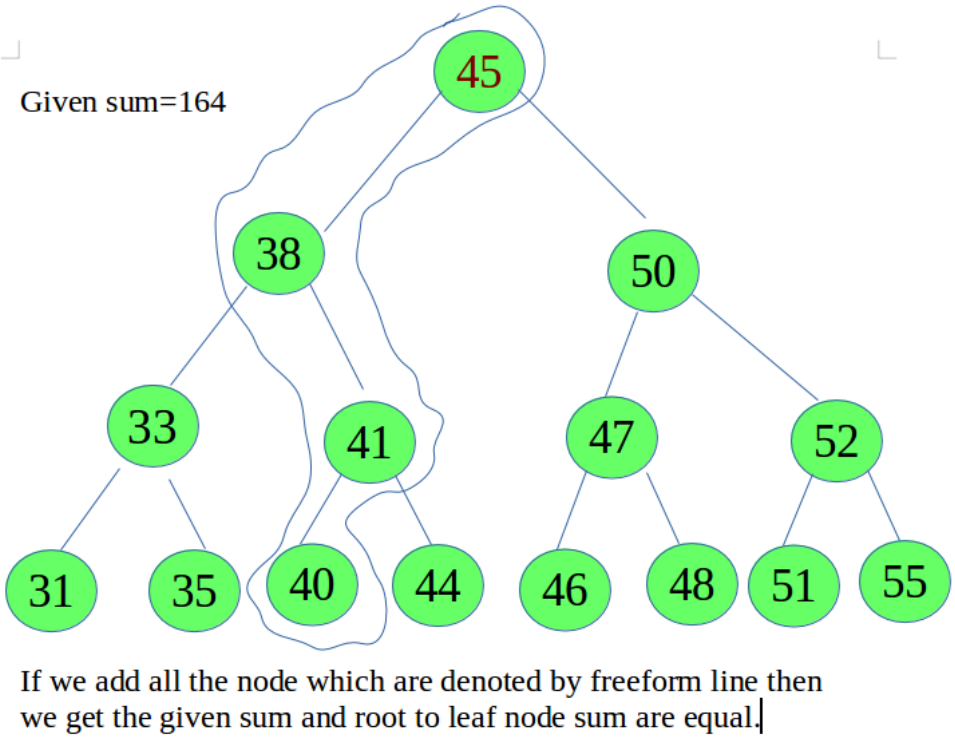
https://www.geeksforgeeks.org/reverse-path-bst-using-queue/

Chapter 90

Root to leaf path sum equal to a given number in BST

Root to leaf path sum equal to a given number in BST - GeeksforGeeks

Given a BST and a number. The task is to check whether the given number is equal to the sum of all the node from root leaf across any of the root to leaf paths in the given Binary Search Tree.



Approach: The idea is to traverse from root to all leaves in top-down fashion maintaining a path[] array to store current root to leaf path. While traversing, store data of all nodes of current path in the array path[]. Whenever a leaf node is reached, calculate the sum of all of the nodes on the current path using the array path[] and check if it is equal to the given sum.

Below is the implementation of above approach:

Source

https://www.geeksforgeeks.org/root-to-leaf-path-sum-equal-to-a-given-number-in-bst/

C++

```
 // CPP program to check if root to leaf path
// sum to a given number in BST

#include<bits/stdc++.h>
using namespace std;

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

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to check if root to leaf path
// sum to a given number in BST
int checkThesum(struct Node *root, int path[], int i, int sum)
{
    int sum1 = 0, x, y, j;

    if(root == NULL)
        return 0;

    // insert the data of a node
    path[i] = root->data;
```

```
    // if the node is leaf
    // add all the element in array
    if(root->left==NULL&&root->right==NULL)
    {
        for(j = 0; j <= i; j++)
            sum1 = sum1 + path[j];

        // if the sum of root node to leaf
        // node data is equal then return 1
        if(sum == sum1)
            return 1;
        else
            return 0;
    }

    x = checkThesum(root->left, path, i+1, sum);

    // if x is 1, it means the given sum is matched
    // with root to leaf node sum
    if(x==1)
        return 1;
    else
    {
        return checkThesum(root->right, path, i+1, sum);
    }
}

// Driver code
int main()
{
    int path[100], sum = 164;

    Node *root = newNode(45);
    root->left = newNode(38);
    root->left->left = newNode(33);
    root->left->left->left = newNode(31);
    root->left->left->right = newNode(35);
    root->left->right = newNode(41);
    root->left->right->left = newNode(40);
    root->left->right->right = newNode(44);
    root->right = newNode(50);
    root->right->left = newNode(47);
    root->right->left->left = newNode(46);
    root->right->left->right = newNode(48);
    root->right->right = newNode(52);
    root->right->right->left = newNode(51);
    root->right->right->right = newNode(55);
```

```
    if(checkThesum(root, path, 0, sum)==1)
        cout<<"YES\n";
    else
        cout<<"NO\n";

    return 0;
}
```

Java

```
// Java program to check if
// root to leaf path sum to
// a given number in BST
class GFG
{

// BST node
static class Node
{
int data;
Node left, right;
}

/* Helper function that
allocates a new node */
static Node newNode(int data)
{
Node node = new Node();
node.data = data;
node.left = node.right = null;
return (node);
}

// Function to check if root
// to leaf path sum to a
// given number in BST
static int checkThesum(Node root, int path[],
int i, int sum)
{
int sum1 = 0, x, y, j;

if(root == null)
return 0;

// insert the data of a node
path[i] = root.data;

// if the node is leaf
// add all the element in array
if(root.left == null &&
root.right == null)
```

{
for(j = 0; j <= i; j++) sum1 = sum1 + path[j]; // if the sum of root node to leaf // node data is equal then return 1 if(sum == sum1) return 1; else return 0; } x = checkThesum(root.left, path, i + 1, sum); // if x is 1, it means the // given sum is matched with // root to leaf node sum if(x == 1) return 1; else { return checkThesum(root.right, path, i + 1, sum); } } // Driver code public static void main(String args[]) { int path[] = new int[100], sum = 164; Node root = newNode(45); root.left = newNode(38); root.left.left = newNode(33); root.left.left.left = newNode(31); root.left.left.right = newNode(35); root.left.right = newNode(41); root.left.right.left = newNode(40); root.left.right.right = newNode(44); root.right = newNode(50); root.right.left = newNode(47); root.right.left.left = newNode(46); root.right.left.right = newNode(48); root.right.right = newNode(52); root.right.right.left = newNode(51); root.right.right.right = newNode(55); if(checkThesum(root, path, 0, sum) == 1) System.out.print("YES\n"); else System.out.print("NO\n"); } } // This code is contributed by Arnab Kundu [tabbyending]

Output:

```
YES
```

Improved By : andrew1234

Chapter 91

Second largest element in BST

Second largest element in BST - GeeksforGeeks

Given a Binary Search Tree(BST), find the second largest element.

Examples:

```
Input: Root of below BST
    10
   /
  5

Output:  5


Input: Root of below BST
        10
      /   \
    5      20
             \
              30

Output:  20
```

Source: Microsoft Interview

The idea is similar to below post.
K'th Largest Element in BST when modification to BST is not allowed

The second largest element is second last element in inorder traversal and second element in reverse inorder traversal. We traverse given Binary Search Tree in reverse inorder and keep track of counts of nodes visited. Once the count becomes 2, we print the node.

Below is the implementation of above idea.
C++

```
 // C++ program to find 2nd largest element in BST
#include<iostream>
using namespace std;

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

// 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 function to find 2nd largest element in a given tree.
void secondLargestUtil(Node *root, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= 2)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    secondLargestUtil(root->right, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the 2nd largest
    if (c == 2)
    {
        cout << "2nd largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
    secondLargestUtil(root->left, c);
}

// Function to find 2nd largest element
void secondLargest(Node *root)
```

```
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    secondLargestUtil(root, c);
}

/* A utility function to insert a new node with given key in BST */
Node* insert(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 if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver Program to test above functions
int main()
{
    /* 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);

    secondLargest(root);

    return 0;
}
```

Java

```
 // Java code to find second largest element in BST

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // function to insert new nodes
    public void insert(int data)
    {
        this.root = this.insertRec(this.root, data);
    }

    /* A utility function to insert a new node with given
    key in BST */
    Node insertRec(Node node, int data)
    {
        /* If the tree is empty, return a new node */
        if (node == null) {
            this.root = new Node(data);
            return this.root;
        }

        /* Otherwise, recur down the tree */
        if (data < node.data) {
            node.left = this.insertRec(node.left, data);
        } else {
            node.right = this.insertRec(node.right, data);
        }
```

```
    return node;
}

// class that stores the value of count
public class count {
    int c = 0;
}

// Function to find 2nd largest element
void secondLargestUtil(Node node, count C)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (node == null || C.c >= 2)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    this.secondLargestUtil(node.right, C);

     // Increment count of visited nodes
    C.c++;

    // If c becomes k now, then this is the 2nd largest
    if (C.c == 2) {
        System.out.print("2nd largest element is "+
                                          node.data);
        return;
    }

     // Recur for left subtree
    this.secondLargestUtil(node.left, C);
}

// Function to find 2nd largest element
void secondLargest(Node node)
{
    // object of class count
    count C = new count();
    this.secondLargestUtil(this.root, C);
}

// Driver function
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
```

```
            50
          /    \
         30     70
        /  \    /  \
      20   40  60   80 */

        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        tree.secondLargest(tree.root);
    }
}

// This code is contributed by Kamal Rawal
```

Output:

```
2nd largest element is 70
```

Time complexity of the above solution is O(h) where h is height of BST.

This article is contributed by **Ravi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/second-largest-element-in-binary-search-tree-bst/

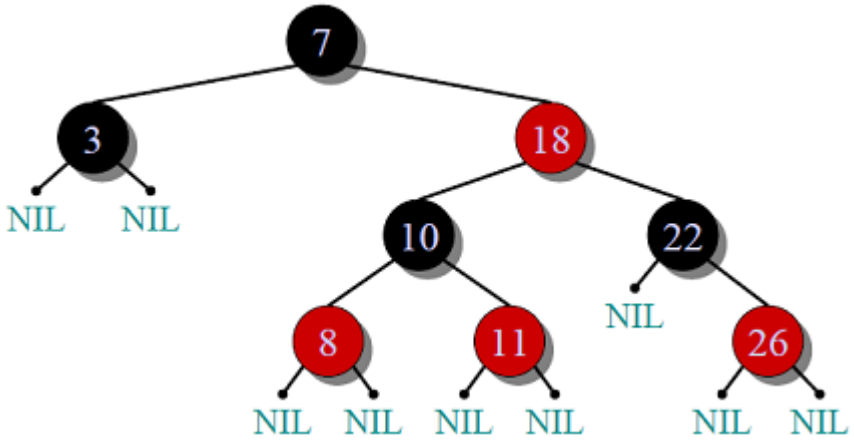
Chapter 92

Self-Balancing-Binary-Search-Trees (Comparisons)

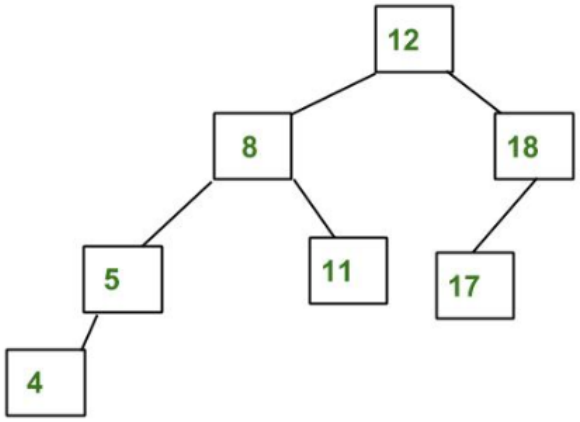
Self-Balancing-Binary-Search-Trees (Comparisons) - GeeksforGeeks

Self-Balancing Binary Search Trees are ***height-balanced*** binary search trees that automatically keeps height as small as possible when insertion and deletion operations are performed on tree. The height is typically maintained in order of Log n so that all operations take O(Log n) time on average.

Examples :
Red Black Tree



AVL Tree:



How do Self-Balancing-Tree maintain height?
A typical operation done by trees is rotation. 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.
```

We have already discussed AVL tree, Red Black Tree and Splay Tree. In this acrticle, we will compare the efficiency of these trees:

Metric	RB Tree	AVL Tree
worst case	**O(1)**	**O(logn)**
of tree	**2*log(n)**	**1.44*log(n)**
worst case		
Moderate		
Faster		
Slower		

Metric	RB Tree	AVL Tree
Efficient Implementation requires node no extra information	**Three pointers with color bit per node**	
worst case	**O(logn)**	**O(logn)**
Mostly used	**As universal data structure**	**When frequent lookups**
retrieved again and again		
Real world Application	**Multiset, Multimap, Map, Set, etc.**	**Database Transactions**

Source

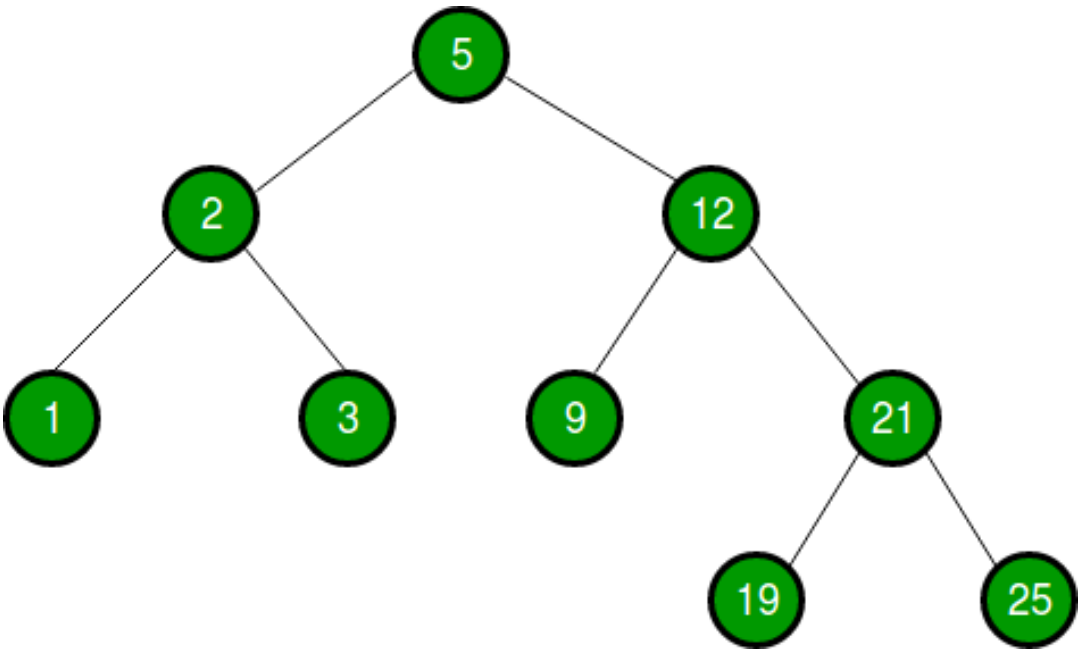
https://www.geeksforgeeks.org/self-balancing-binary-search-trees-comparisons/

Chapter 93

Shortest distance between two nodes in BST

Shortest distance between two nodes in BST - GeeksforGeeks

Given a Binary Search Tree and two keys in it. Find the distance between two nodes with given two keys. It may be assumed that both keys exist in BST.




```
Input :  Root of above tree
```

```
         a = 3, b = 9
Output : 4
Distance between 3 and 9 in
above BST is 4.

Input :  Root of above tree
         a = 9, b = 25
Output : 3
Distance between 9 and 25 in
above BST is 3.
```

We have discusseddistance between two nodes in binary tree. The time complexity of this solution is O(n)

In case of BST, we can find distance faster. We start from root and for every node, we do following.

1. If both keys are greater than current node, we move to right child of current node.
2. If both keys are smaller than current node, we move to left child of current node.
3. If one keys is smaller and other key is greater, current node is Lowest Common Ancestor (LCA) of two nodes. We find distances of current node from two keys and return sum of the distances.

```
 // CPP program to find distance between
// two nodes in BST
#include <bits/stdc++.h>
using namespace std;

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

struct Node* newNode(int key)
{
    struct Node* ptr = new Node;
    ptr->key = key;
    ptr->left = ptr->right = NULL;
    return ptr;
}

// Standard BST insert function
struct Node* insert(struct Node* root, int key)
{
    if (!root)
        root = newNode(key);
```

```
    else if (root->key > key)
        root->left = insert(root->left, key);
    else if (root->key < key)
        root->right = insert(root->right, key);
    return root;
}

// This function returns distance of x from
// root. This function assumes that x exists
// in BST and BST is not NULL.
int distanceFromRoot(struct Node* root, int x)
{
    if (root->key == x)
        return 0;
    else if (root->key > x)
        return 1 + distanceFromRoot(root->left, x);
    return 1 + distanceFromRoot(root->right, x);
}

// Returns minimum distance beween a and b.
// This function assumes that a and b exist
// in BST.
int distanceBetween2(struct Node* root, int a, int b)
{
    if (!root)
        return 0;

    // Both keys lie in left
    if (root->key > a && root->key > b)
        return distanceBetween2(root->left, a, b);

    // Both keys lie in right
    if (root->key < a && root->key < b) // same path
        return distanceBetween2(root->right, a, b);

    // Lie in opposite directions (Root is
    // LCA of two nodes)
    if (root->key >= a && root->key <= b)
        return distanceFromRoot(root, a) +
               distanceFromRoot(root, b);
}

// This function make sure that a is smaller
// than b before making a call to findDistWrapper()
int findDistWrapper(Node *root, int a, int b)
{
   if (a > b)
     swap(a, b);
```

```
   return distanceBetween2(root, a, b);
}

// Driver code
int main()
{
    struct Node* root = NULL;
    root = insert(root, 20);
    insert(root, 10);
    insert(root, 5);
    insert(root, 15);
    insert(root, 30);
    insert(root, 25);
    insert(root, 35);
    int a = 5, b = 55;
    cout << findDistWrapper(root, 5, 35);
    return 0;
}
```

Output:

```
2
```

Time Complexity : O(h) where h is height of Binary Search Tree.

Source

https://www.geeksforgeeks.org/shortest-distance-between-two-nodes-in-bst/

Chapter 94

Simple Recursive solution to check whether BST contains dead end

Simple Recursive solution to check whether BST contains dead end - GeeksforGeeks

Given a Binary Search Tree that contains positive integer values greater than 0. The task is to check whether the BST contains a dead end or not. Here Dead End means, we are not able to insert any integer element after that node.

Examples:

```
Input :        8
             /   \
           5      9
         /   \
        2     7
       /
      1
Output : Yes
Explanation : Node "1" is the dead End because
         after that we cant insert any element.

Input :       8
            /   \
           7     10
         /      /   \
        2      9     13

Output :Yes
Explanation : We can't insert any element at
```

```
node 9.
```

We have discussed a solution in below post.

Check whether BST contains Dead End or not

The idea in this post is based on method 3 of Check if a binary tree is BST or not.

First of all, it is given that it is a BST and nodes are greater than zero, root node can be in the range [1, ∞] and if root val is say, val, then left sub-tree can have the value in the range [1, val-1] and right sub-tree the value in range [val+1, ∞].
we need to traverse recursively and when the the min and max value of range coincided it means that we cannot add any node further in the tree.
Hence we encounter a dead end.

Following is the simple recursive solution to the problem.
C++

```
 // CPP Program to check if there is a dead end
// in BST or not.
#include <bits/stdc++.h>
using namespace std;

// A BST node
struct Node {
    int data;
    struct Node *left, *right;
};

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

/* 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->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
```

```
        node->right = insert(node->right, key);

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

// Returns true if tree with given root contains
// dead end or not. min and max indicate range
// of allowed values for current node. Initially
// these values are full range.
bool deadEnd(Node* root, int min=1, int max=INT_MAX)
{
    // if the root is null or the recursion moves
    // after leaf node it will return false
    // i.e no dead end.
    if (!root)
        return false;

    // if this occurs means dead end is present.
    if (min == max)
        return true;

    // heart of the recursion lies here.
    return deadEnd(root->left, min, root->data - 1) ||
           deadEnd(root->right, root->data + 1, max);
}

// Driver program
int main()
{
    /*   8
       /   \
      5    11
     /  \
    2    7
     \
      3
       \
        4 */
    Node* root = NULL;
    root = insert(root, 8);
    root = insert(root, 5);
    root = insert(root, 2);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 11);
    root = insert(root, 4);
    if (deadEnd(root) == true)
```

```
        cout << "Yes " << endl;
    else
        cout << "No " << endl;
    return 0;
}
```

Java

```
 // Java Program to check if there
// is a dead end in BST or not.
class BinarySearchTree {

    // Class containing left and right
    // child of current node and key value
    class Node {
        int data;
        Node left, right;

        public Node(int item) {
            data = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // This method mainly calls insertRec()
    void insert(int data) {
    root = insertRec(root, data);
    }

    // A recursive function
    // to insert a new key in BST
    Node insertRec(Node root, int data) {

        // If the tree is empty,
        // return a new node
        if (root == null) {
            root = new Node(data);
            return root;
        }
```

```
        /* Otherwise, recur down the tree */
        if (data < root.data)
            root.left = insertRec(root.left, data);
        else if (data > root.data)
            root.right = insertRec(root.right, data);

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

// Returns true if tree with given root contains
// dead end or not. min and max indicate range
// of allowed values for current node. Initially
// these values are full range.
boolean deadEnd(Node root, int min, int max)
{
    // if the root is null or the recursion moves
    // after leaf node it will return false
    // i.e no dead end.
    if (root==null)
        return false;

    // if this occurs means dead end is present.
    if (min == max)
        return true;

    // heart of the recursion lies here.
    return deadEnd(root.left, min, root.data - 1)||
                deadEnd(root.right, root.data + 1, max);
}


    // Driver Program
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        /*       8
               /   \
              5    11
             /  \
            2    7
             \
              3
               \
                4 */
        tree.insert(8);
        tree.insert(5);
        tree.insert(2);
```

```
        tree.insert(3);
        tree.insert(7);
        tree.insert(11);
        tree.insert(4);

        if (tree.deadEnd(tree.root ,1 ,
                Integer.MAX_VALUE) == true)

        System.out.println("Yes ");
        else
        System.out.println("No " );
    }
}

// This code is contributed by Gitanjali.
```

Output:

```
Yes
```

Improved By : vishal22091998

Source

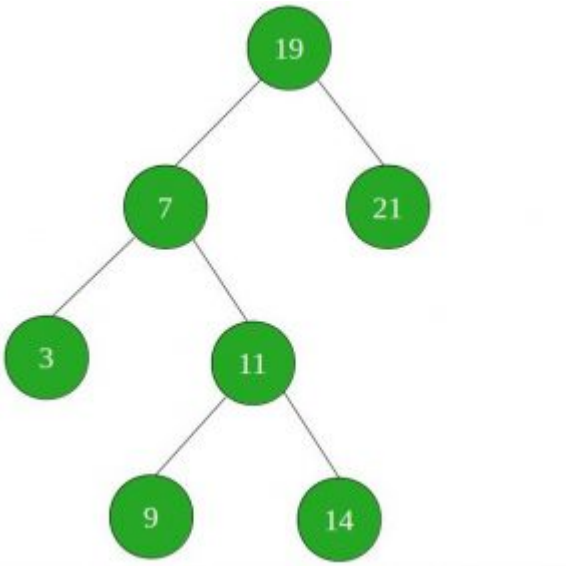
https://www.geeksforgeeks.org/simple-recursive-solution-check-whether-bst-contains-dead-end/

Chapter 95

Smallest number in BST which is greater than or equal to N

Smallest number in BST which is greater than or equal to N - GeeksforGeeks

Given a Binary Search Treeand a number N, the task is to find the smallest number in the binary search tree that is greater than or equal to N. Print the value of the element if it exists otherwise print -1.



Examples:

Input: N = 20
Output: 21
Explanation: 21 is the smallest element greater than 20.

Input: N = 18
Output: 19
Explanation: 19 is the smallest element greater than 18.

Approach:
The idea is to follow the recursive approach for solving the problem i.e. start searching for the element from the root.

- If there is a leaf node having a value less than N, then element doesn't exist and return -1.
- Otherwise, if node's value is greater than or equal to N and left child is NULL or less than N then return the node value.
- Else if node's value is less than N, then search for the element in the right subtree.
- Else search for the element in the left subtree by calling the function recursively according to the left or right value.

```
 // C++ program to find the smallest value
// greater than or equal to N
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

// To create new BST Node
Node* createNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;

    return temp;
}

// To add a new node in BST
Node* add(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return createNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
```

```
    if (key < node->data)
        node->left = add(node->left, key);
    else if (key > node->data)
        node->right = add(node->right, key);

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

// function to find min value less then N
int findMinforN(Node* root, int N)
{
    // If leaf node reached and is smaller than N
    if (root->left == NULL && root->right == NULL
                                && root->data < N)
        return -1;

    // If node's value is greater than N and left value
    // is NULL or smaller then return the node value
    if ((root->data >= N && root->left == NULL)
        || (root->data >= N && root->left->data < N))
        return root->data;

    // if node value is smaller than N search in the
    // right subtree
    if (root->data <= N)
        return findMinforN(root->right, N);

    // if node value is greater than N search in the
    // left subtree
    else
        return findMinforN(root->left, N);
}

// Drivers code
int main()
{
    /*    19
        /    \
       7     21
     /   \
    3     11
         /   \
         9    14
          */

    Node* root = NULL;
    root = add(root, 19);
```

```
    root = add(root, 7);
    root = add(root, 3);
    root = add(root, 11);
    root = add(root, 9);
    root = add(root, 13);
    root = add(root, 21);

    int N = 18;
    cout << findMinforN(root, N) << endl;

    return 0;
}
```

Output:

```
19
```

Source

https://www.geeksforgeeks.org/smallest-number-in-bst-which-is-greater-than-or-equal-to-n/

Chapter 96

Sorted Array to Balanced BST

Sorted Array to Balanced BST - GeeksforGeeks

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Examples:

```
Input:  Array {1, 2, 3}
Output: A Balanced BST
     2
   /  \
  1    3

Input: Array {1, 2, 3, 4}
Output: A Balanced BST
      3
    /  \
   2    4
 /
1
```

Algorithm
In the previous post, we discussed construction of BST from sorted Linked List. Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the array and make it root.
2) Recursively do same for left half and right half.
      a) Get the middle of left half and make it left child of the root
```

```
          created in step 1.
      b) Get the middle of right half and make it right child of the
          root created in step 1.
```

Following is the implementation of the above algorithm. The main code which creates Balanced BST is highlighted.

C

```
 #include<stdio.h>
#include<stdlib.h>

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);

/* A function that constructs Balanced Binary Search Tree from a sorted array */
struct TNode* 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 TNode *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;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct TNode* newNode(int data)
{
```

```
    struct TNode* node = (struct TNode*)
                         malloc(sizeof(struct TNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    /* Convert List to BST */
    struct TNode *root = sortedArrayToBST(arr, 0, n-1);
    printf("n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Java

```
 // Java program to print BST in given range

// A binary tree node
class Node {

    int data;
    Node left, right;

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

```
class BinaryTree {

    static Node root;

    /* A function that constructs Balanced Binary Search Tree
     from a sorted array */
    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;
        Node node = new Node(arr[mid]);

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

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

        return node;
    }

    /* A utility function to print preorder traversal of BST */
    void preOrder(Node node) {
        if (node == null) {
            return;
        }
        System.out.print(node.data + " ");
        preOrder(node.left);
        preOrder(node.right);
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int arr[] = new int[]{1, 2, 3, 4, 5, 6, 7};
        int n = arr.length;
        root = tree.sortedArrayToBST(arr, 0, n - 1);
        System.out.println("Preorder traversal of constructed BST");
        tree.preOrder(root);
    }
}
```

```
// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python code to convert a sorted array
# to a balanced Binary Search Tree

# binary tree node
class Node:
    def __init__(self, d):
        self.data = d
        self.left = None
        self.right = None

# function to convert sorted array to a
# balanced BST
# input : sorted array of integers
# output: root node of balanced BST
def sortedArrayToBST(arr):

    if not arr:
        return None

    # find middle
    mid = (len(arr)) / 2

    # make the middle element the root
    root = Node(arr[mid])

    # left subtree of root has all
    # values <arr[mid]
    root.left = sortedArrayToBST(arr[:mid])

    # right subtree of root has all
    # values >arr[mid]
    root.right = sortedArrayToBST(arr[mid+1:])
    return root

# A utility function to print the preorder
# traversal of the BST
def preOrder(node):
    if not node:
        return

    print node.data,
    preOrder(node.left)
    preOrder(node.right)
```

```
# driver program to test above function
"""
Constructed balanced BST is
    4
/ \
2 6
/ \ / \
1 3 5 7
"""

arr = [1, 2, 3, 4, 5, 6, 7]
root = sortedArrayToBST(arr)
print "PreOrder Traversal of constructed BST ",
preOrder(root)

# This code is contributed by Ishita Tripathi
```

Time Complexity: O(n)
Following is the recurrance relation for sortedArrayToBST().

```
  T(n) = 2T(n/2) + C
  T(n) -->  Time taken for an array of size n
   C   -->  Constant (Finding middle of array and linking root to left
                      and right subtrees take constant time)
```

The above recurrence can be solved using Master Theorem as it falls in case 1.

Improved By : IshitaTripathi

Source

https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/

Chapter 97

Sorted Linked List to Balanced BST

Sorted Linked List to Balanced BST - GeeksforGeeks

Given a Singly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Linked List.

Examples:

```
Input:  Linked List 1->2->3
Output: A Balanced BST
     2
   /  \
  1    3


Input: Linked List 1->2->3->4->5->6->7
Output: A Balanced BST
        4
      /   \
     2     6
   /  \   / \
  1   3  5   7

Input: Linked List 1->2->3->4
Output: A Balanced BST
      3
    /  \
   2    4
 /
1
```

```
Input:  Linked List 1->2->3->4->5->6
Output: A Balanced BST
      4
    /   \
   2     6
 /  \   /
1   3  5
```

Method 1 (Simple)
Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
       a) Get the middle of left half and make it left child of the root
          created in step 1.
       b) Get the middle of right half and make it right child of the
          root created in step 1.
```

Time complexity: O(nLogn) where n is the number of nodes in Linked List.

Method 2 (Tricky)
The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we allocate memory for root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.
While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

C

```
 #include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct LNode
{
    int data;
    struct LNode* next;
};
```

```
/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);
int countLNodes(struct LNode *head);
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n);


/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of linked list
       n  --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) which is n-n/2-1*/
    root->right = sortedListToBSTRecur(head_ref, n-n/2-1);
```

```
    return root;
}



/* UTILITY FUNCTIONS */

/* A utility function that returns count of nodes in a given Linked List */
int countLNodes(struct LNode *head)
{
    int count = 0;
    struct LNode *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct LNode** head_ref, int new_data)
{
    /* allocate node */
    struct LNode* new_node =
        (struct LNode*) malloc(sizeof(struct LNode));
    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct LNode *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Helper function that allocates a new node with the
```

```
   given data and NULL left and right pointers. */
struct TNode* newNode(int data)
{
    struct TNode* node = (struct TNode*)
                         malloc(sizeof(struct TNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct LNode* head = NULL;

    /* Let us create a sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6->7 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given Linked List ");
    printList(head);

    /* Convert List to BST */
    struct TNode *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Java

```
class LinkedList {

   /* head node of link list */
   static LNode head;

   /* Link list Node */
   class LNode
   {
       int data;
       LNode next, prev;

       LNode(int d)
       {
           data = d;
           next = prev = null;
       }
   }

   /* A Binary Tree Node */
   class TNode
   {
       int data;
       TNode left, right;

       TNode(int d)
       {
           data = d;
           left = right = null;
       }
   }

   /* This function counts the number of nodes in Linked List
      and then calls sortedListToBSTRecur() to construct BST */
   TNode sortedListToBST()
   {
       /*Count the number of nodes in Linked List */
       int n = countNodes(head);

       /* Construct BST */
       return sortedListToBSTRecur(n);
   }

   /* The main function that constructs balanced BST and
      returns root of it.
      n  --> No. of nodes in the Doubly Linked List */
   TNode sortedListToBSTRecur(int n)
```

```
{
    /* Base Case */
    if (n <= 0)
        return null;

    /* Recursively construct the left subtree */
    TNode left = sortedListToBSTRecur(n / 2);

    /* head_ref now refers to middle node,
       make middle node as root of BST*/
    TNode root = new TNode(head.data);

    // Set pointer to left subtree
    root.left = left;

    /* Change head pointer of Linked List for parent
       recursive calls */
    head = head.next;

    /* Recursively construct the right subtree and link it
       with root. The number of nodes in right subtree  is
       total nodes - nodes in left subtree - 1 (for root) */
    root.right = sortedListToBSTRecur(n - n / 2 - 1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a
   given Linked List */
int countNodes(LNode head)
{
    int count = 0;
    LNode temp = head;
    while (temp != null)
    {
        temp = temp.next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginging of
   the Doubly Linked List */
void push(int new_data)
{
    /* allocate node */
    LNode new_node = new LNode(new_data);
```

```
    /* since we are adding at the begining,
       prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = head;

    /* change prev of head node to new node */
    if (head != null)
        head.prev = new_node;

    /* move the head to point to the new node */
    head = new_node;
}

/* Function to print nodes in a given linked list */
void printList(LNode node)
{
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(TNode node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.left);
    preOrder(node.right);
}

/* Drier program to test above functions */
public static void main(String[] args) {
    LinkedList llist = new LinkedList();

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 7->6->5->4->3->2->1 */
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
```

```
        llist.push(1);

        System.out.println("Given Linked List ");
        llist.printList(head);

        /* Convert List to BST */
        TNode root = llist.sortedListToBST();
        System.out.println("");
        System.out.println("Pre-Order Traversal of constructed BST ");
        llist.preOrder(root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Given Linked List 1 2 3 4 5 6 7
 PreOrder Traversal of constructed BST 4 2 1 3 6 5 7
```

Time Complexity: O(n)

Improved By : Sulav Timsina

Source

https://www.geeksforgeeks.org/sorted-linked-list-to-balanced-bst/

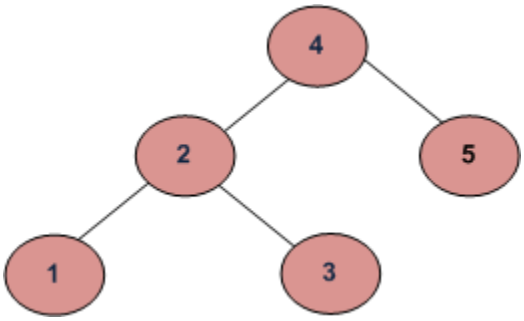
Chapter 98

Sorted order printing of a given array that represents a BST

Sorted order printing of a given array that represents a BST - GeeksforGeeks

Given an array that stores a complete Binary Search Tree, write a function that efficiently prints the given array in ascending order.

For example, given an array [4, 2, 5, 1, 3], the function should print 1, 2, 3, 4, 5



Solution:

Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in standard Inorder Tree Traversal.

Implementation:
C

```
 #include<stdio.h>

void printSorted(int arr[], int start, int end)
{
  if(start > end)
    return;
```

```
  // print left subtree
  printSorted(arr, start*2 + 1, end);

  // print root
  printf("%d  ", arr[start]);

  // print right subtree
  printSorted(arr, start*2 + 2, end);
}

int main()
{
  int arr[] = {4, 2, 5, 1, 3};
  int arr_size = sizeof(arr)/sizeof(int);
  printSorted(arr, 0, arr_size-1);
  getchar();
  return 0;
}
```

Java

```
 // JAVA Code for Sorted order printing of a
// given array that represents a BST
class GFG{

private static void printSorted(int[] arr, int start,
                                        int end) {
        if(start > end)
            return;

        // print left subtree
        printSorted(arr, start*2 + 1, end);

        // print root
        System.out.print(arr[start] + " ");

        // print right subtree
        printSorted(arr, start*2 + 2, end);
        }

    // driver program to test above function
    public static void main(String[] args) {
        int arr[] = {4, 2, 5, 1, 3};

        printSorted(arr, 0, arr.length-1);
    }
}
```

```
// This code is contributed by Arnav Kr. Mandal.
```

Output:

```
1 2 3 4 5
```

Time Complexity: O(n)

Source

https://www.geeksforgeeks.org/sorted-order-printing-of-an-array-that-represents-a-bst/

Chapter 99

Special two digit numbers in a Binary Search Tree

Special two digit numbers in a Binary Search Tree - GeeksforGeeks

Given a Binary Search Trees, task is to count the number of nodes which are having special two digit numbers.

Prerequisite : Special Two Digit Number Binary Search Tree

Examples :

```
Input : 15 7 987 21
Output : 0

Input : 19 99 57 1 22
Output : 2
```

Algorithm : Iterate through each node of tree recursively with a variable count, and check each node's data for special two digit number. If it is then increment the variable count. At the end, return count.

```
 // C program to count number of nodes in
// BST containing two digit special number
#include <stdio.h>
#include <stdlib.h>

// A Tree node
struct Node
{
    struct Node *left;
    int info;
```

```
    struct Node *right;
};

// Function to create a new node
void insert(struct Node **rt, int key)
{
    if(*rt == NULL)
    {
        (*rt) = (struct Node *)malloc(sizeof(struct Node));
        (*rt) -> left = NULL;
        (*rt) -> right = NULL;
        (*rt) -> info = key;
    }
    else if(key < ((*rt) -> info))
        insert(&((*rt) -> left), key);
    else
        insert(&(*rt) -> right, key);
}

// Function to find if number
// is special or not
int check(int num)
{
    int sum = 0, i = num, sum_of_digits, prod_of_digits ;

    // Check if number is two digit or not
    if(num < 10 || num > 99)
        return 0;
    else
    {
        sum_of_digits = (i % 10) + (i / 10);
        prod_of_digits = (i % 10) * (i / 10);
        sum = sum_of_digits + prod_of_digits;
    }

    if(sum == num)
        return 1;
    else
        return 0;
}

// Function to count number of special two digit number
void countSpecialDigit(struct Node *rt, int *c)
{
    int x;
    if(rt == NULL)
        return;
    else
```

```
    {
        x = check(rt -> info);
        if(x == 1)
            *c = *c + 1;
        countSpecialDigit(rt -> left, c);
        countSpecialDigit(rt -> right, c);
    }
}

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

    // Initialize result
    int count = 0;

    // Function call to insert() to insert nodes
    insert(&root, 50);
    insert(&root, 29);
    insert(&root, 59);
    insert(&root, 19);
    insert(&root, 53);
    insert(&root, 556);
    insert(&root, 56);
    insert(&root, 94);
    insert(&root, 13);

    // Function call, to check each node for
    // special two digit number
    countSpecialDigit(root, &count);
    printf("%d", count);

    return 0;
}
```

Output:

```
3
```

Time Complexity : O(N), where N is the number of nodes in Tree.

Source

https://www.geeksforgeeks.org/special-two-digit-numbers-in-a-binary-search-tree/

Chapter 100

Subarray whose sum is closest to K

Subarray whose sum is closest to K - GeeksforGeeks

Given an array of positive and negative integers and an integer K. The task is to find the subarray which has its sum closest to k. In case of multiple answers, print any one.
Note: Closest here means abs(sum-k) should be minimal.

Examples:

> **Input:** a[] = { -5, 12, -3, 4, -15, 6, 1 }, K = 2
> **Output:** 1
> The subarray {-3, 4} or {1} has sum = 1 which is the closest to K.
>
> **Input:** a[] = { 2, 2, -1, 5, -3, -2 }, K = 7
> **Output:** 6
> Here the output can be 6 or 8
> The subarray {2, 2, -1, 5} gives sum as 8 which has abs(8-7) = 1 which is same as that of the subarray {2, -1, 5} which has abs(6-7) = 1.

A **naive approach** is to check for all possible subarray sum using prefix sum. The complexity in that case will be $O(N^2)$.

An **efficient solution** will be to use C++ STL set and binary search to solve the following problem. Follow the below algorithm to solve the above problem.

- Initially insert the first element in the set container.
- Initiallize the answer *sum* as first element and *difference* as abs(A_0-k).
- Iterate for all array elements from 1 to N and keep adding the elements to prefix sum at each step to the set container.

- At every iteration, since the prefix sum is already there, we just need to subtract the sum of some elements from beginning to get the sum of any subarray. The greedy way will be to subtract the sum of the subarray which takes the sum closest to K.
- Using binary search (lower_bound() function can be used) find the sum of subarray from beginning which is closest to (prefix-k) as the subtraction of that number from prefix sum will give the subarray sum which is closest to K till that iteration.
- Also check for the index before which lower_bound() returns, since the sum can either be greater or lesser than K.
- If the lower_bound returns no such element, then the current prefix sum is compared and updated if it was lesser than the previous computed sum.

Below is the implementation of the above approach.

```
 // C++ program to find the
// sum of subarray whose sum is
// closest to K
#include <bits/stdc++.h>
using namespace std;

// Function to find the sum of subarray
// whose sum is closest to K
int closestSubarraySumToK(int a[], int n, int k)
{

    // Declare a set
    set<int> s;

    // initially consider the
    // first subarray as the first
    // element in the array
    int presum = a[0];

    // insert
    s.insert(a[0]);

    // Initially let this difference
    // be the minimum
    int mini = abs(a[0] - k);

    // let this be the sum
    // of the subarray
    // to be searched initially
    int sum = presum;

    // iterate for all the array elements
    for (int i = 1; i < n; i++) {
```

```
// calculate the prefix sum
presum += a[i];

// find the closest subarray
// sum to by using lower_bound
auto it = s.lower_bound(presum - k);

// if it is the first element
// in the set
if (it == s.begin()) {

    // get the prefix sum till start
    // of the subarray
    int diff = *it;

    // if the subarray sum is closest to K
    // than the previous one
    if (abs((presum - diff) - k) < mini) {

        // update the minimal difference
        mini = abs((presum - diff) - k);

        // update the sum
        sum = presum - diff;
    }
}

// if the difference is
// present in between
else if (it != s.end()) {

    // get the prefix sum till start
    // of the subarray
    int diff = *it;

    // if the subarray sum is closest to K
    // than the previous one
    if (abs((presum - diff) - k) < mini) {

        // update the minimal difference
        mini = abs((presum - diff) - k);

        // update the sum
        sum = presum - diff;
    }

    // also check for the one before that
```

```
            // since the sum can be greater than
            // or less than K also
            it--;

            // get the prefix sum till start
            // of the subarray
            diff = *it;

            // if the subarray sum is closest to K
            // than the previous one
            if (abs((presum - diff) - k) < mini) {

                // update the minimal difference
                mini = abs((presum - diff) - k);

                // update the sum
                sum = presum - diff;
            }
        }

        // if there exists no such prefix sum
        // then the current prefix sum is
        // checked and updated
        else {

            // if the subarray sum is closest to K
            // than the previous one
            if (abs(presum - k) < mini) {

                // update the minimal difference
                mini = abs(presum - k);

                // update the sum
                sum = presum;
            }
        }

        // insert the current prefix sum
        s.insert(presum);
    }

    return sum;
}

// Driver Code
int main()
{
    int a[] = { -5, 12, -3, 4, -15, 6, 1 };
```

```
    int n = sizeof(a) / sizeof(a[0]);
    int k = 2;

    cout << closestSubarraySumToK(a, n, k);
    return 0;
}
```

Time Complexity: O(N log N)

Source

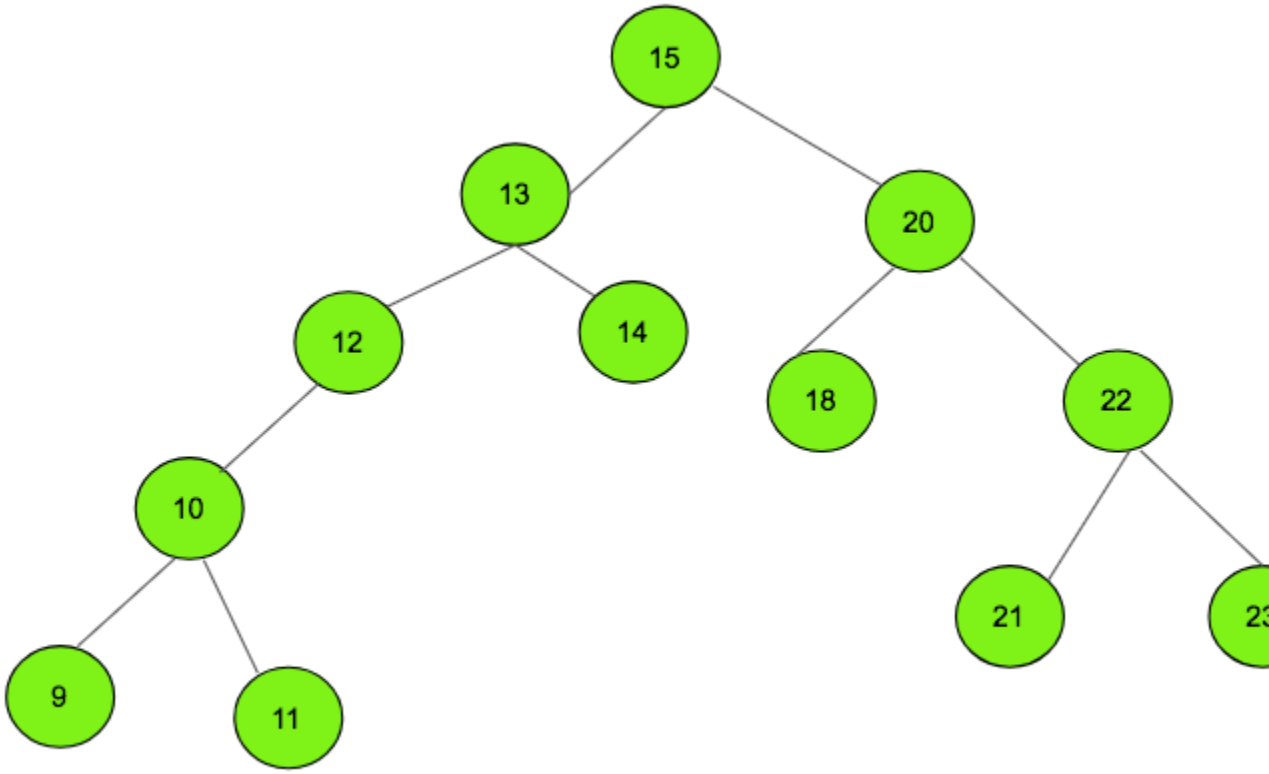
https://www.geeksforgeeks.org/subarray-whose-sum-is-closest-to-k/

Chapter 101

Sum of cousin nodes of a given node in a BST

Sum of cousin nodes of a given node in a BST - GeeksforGeeks

Given a binary search tree and a number N, the task is to find the sum of the cousins of the given node N if a node with given value 'N' is present in the given BST otherwise print -1.



Examples:

```
Input: Node = 12
Output: 40
Cousins are 18 and 22

Input: 19
Output: -1
```

Approach: Given below is the algorithm to solve the problem.

- Find the parent of the given node, if the node is not present return -1.
- Traverse in the tree, find the level of each node while traversal.
- If the level is the same as the given node. Check for the parent of that node, if the parent is different then add the node to the sum.

Below is the implementation of above approach:

```
 // C++ program to find the sum of cousins
// of a node of a given BST
#include <bits/stdc++.h>
using namespace std;

// structure to store the binary tree
struct Tree {
    int data;
    struct Tree *left, *right;
};

// insertion of node in the binary tree
struct Tree* newNode(int data)
{
    // allocates memory
    struct Tree* node = (struct Tree*)malloc(sizeof(struct Tree));

    // initializes data
    node->data = data;

    // marks the left and right
    // child as NULL
    node->left = node->right = NULL;

    // Return the node after allocating memory
    return (node);
};

// Function which calculates the sum of the cousin Node
int SumOfCousin(struct Tree* root, int p,
                int level1, int level)
{
    int sum = 0;
    if (root == NULL)
        return 0;

    // nodes which has same parent
    // as the given node will not be
    // taken to count for calculation
    if (p == root->data)
        return 0;

    // if the level is same
    // then it is a cousin
    // as parent checking has been
    // done above
    if (level1 == level)
        return root->data;
```

```
    // traverse in the tree left and right
    else
        sum += SumOfCousin(root->left, p, level1 + 1, level) + SumOfCousin(root->right, p, level1

    return sum;
}

// Function that returns the parent node
int ParentNode(struct Tree* root, int NodeData)
{
    int parent = -1;

    // traverse the full Binary tree
    while (root != NULL) {

        // if node is found
        if (NodeData == root->data)
            break;

        // if less than move to left
        else if (NodeData < root->data) {
            parent = root->data;
            root = root->left;
        }

        // if greater than move to rigth
        else {
            parent = root->data;
            root = root->right;
        }
    }

    // Node not found
    if (root == NULL)
        return -1;
    else
        return parent;
}

// Function to find the level of the given node
int LevelOfNode(struct Tree* root, int NodeData)
{
    // calculate the level of node
    int level = 0;
    while (root != NULL) {

        // if the node is found
```

```
        if (NodeData == root->data)
            break;

        // move to the left of the tree
        if (NodeData < root->data) {
            root = root->left;
        }

        // move to the right of the tree
        else {
            root = root->right;
        }

        // increase the level after every traversal
        level++;
    }

    // return the level of a given node
    return level;
}

// Driver Code
int main()
{

    // initialize the root as NULL
    struct Tree* root = NULL;

    // Inserts node in the tree
    // tree is the same as the one in image
    root = newNode(15);
    root->left = newNode(13);
    root->left->left = newNode(12);
    root->left->right = newNode(14);
    root->right = newNode(20);
    root->right->left = newNode(18);
    root->right->right = newNode(22);

    // Given Node
    int NodeData = 12;
    int p, level, sum;

    // fuction call to find the parent node
    p = ParentNode(root, NodeData);

    // if given Node is not present then print -1
    if (p == -1)
        cout << "-1\n";
```

```
    // if present then find the level of the node
    // and call the sum of cousin function
    else {

        // fuction call to find the level of that node
        level = LevelOfNode(root, NodeData);

        // sum of cousin nodes of the given nodes
        sum = SumOfCousin(root, p, 0, level);

        // print the sum
        cout << sum;
    }
    return 0;
}
```

Output:

```
40
```

Source

https://www.geeksforgeeks.org/sum-of-cousin-nodes-of-a-given-node-in-a-bst/

Chapter 102

Sum of k largest elements in BST

Sum of k largest elements in BST - GeeksforGeeks

Given a BST, the task is to find the sum of all elements greater than and equal to kth largest element.

Examples:

```
Input : K = 3
              8
            /   \
           7     10
         /      /   \
        2      9     13
Output : 32
Explanation: 3rd largest element is 9 so sum of all
             elements greater than or equal to 9 are
             9 + 10 + 13 = 32.

Input : K = 2
           8
         /   \
        5    11
      /  \
     2    7
      \
       3
Output : 19
Explanation: 2nd largest element is 8 so sum of all
             elements greater than or equal to 8 are
             8 + 11 = 19.
```

Approach:
The idea is to traverse BST in **Inorder traversal** in a **reverse** way (Right Root Left). Note that Inorder traversal of BST accesses elements in a sorted (or increasing) order, hence the reverse of inorder traversal will be in a sorted order(**decreasing**). While traversing, keep track of the count of visited Nodes and keep adding Nodes until the count becomes k.

```
 // C++ program to find Sum Of All Elements larger
// than or equal to Kth Largest Element In BST
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

// utility function new Node of BST
struct Node* cNode(int data)
{
    Node* node = new Node;
    node->left = NULL;
    node->right = NULL;
    node->data = data;
    return node;
}

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

    // Otherwise, recur down the tree
    if (root->data > key)
        root->left = add(root->left, key);

    else if (root->data < key)
        root->right = add(root->right, key);

    // return the (unchanged) Node pointer
    return root;
}

// function to return sum of all elements larger than
// and equal to Kth largest element
int klargestElementSumUtil(Node* root, int k, int& c)
```

```
{
    // Base cases
    if (root == NULL)
        return 0;
    if (c > k)
        return 0;

    // Compute sum of elements in right subtree
    int ans = klargestElementSumUtil(root->right, k, c);
    if (c >= k)
        return ans;

    // Add root's data
    ans += root->data;

    // Add current Node
    c++;
    if (c >= k)
        return ans;

    // If c is less than k, return left subtree Nodes
    return ans + klargestElementSumUtil(root->left, k, c);
}

// Wrapper over klargestElementSumRec()
int klargestElementSum(struct Node* root, int k)
{
    int c = 0;
    klargestElementSumUtil(root, k, c);
}

// Drivers code
int main()
{
    /*   19
        /    \
       7     21
     /   \
    3     11
         /   \
        9    13
          */

    Node* root = NULL;
    root = add(root, 19);
    root = add(root, 7);
    root = add(root, 3);
    root = add(root, 11);
```

```
    root = add(root, 9);
    root = add(root, 13);
    root = add(root, 21);

    int k = 2;
    cout << klargestElementSum(root, k) << endl;
    return 0;
}
```

Output:

```
40
```

Source

https://www.geeksforgeeks.org/sum-of-k-largest-elements-in-bst/

Chapter 103

Sum of k smallest elements in BST

Sum of k smallest elements in BST - GeeksforGeeks

GivenBinary Search Tree. The task is to find sum of all elements smaller than and equal to Kth smallest element.

Examples:

```
Input :  K = 3
              8
            /   \
           7     10
         /      /   \
        2      9     13
Output : 17
Explanation : Kth smallest element is 8 so sum of all
              element smaller then or equal to 8 are
              2 + 7 + 8

Input : K = 5
           8
         /   \
        5    11
      /  \
     2    7
      \
       3
Output :  25
```

Method 1 (Does not changes BST node structure)

The idea is to traverse BST in inorder traversal. Note that Inorder traversal of BST accesses elements in sorted (or increasing) order. While traversing, we keep track of count of visited Nodes and keep adding Nodes until the count becomes k.

```
 // c++ program to find Sum Of All Elements smaller
// than or equal to Kth Smallest Element In BST
#include <bits/stdc++.h>
using namespace std;

/* Binary tree Node */
struct Node
{
    int data;
    Node* left, * right;
};

// utility function new Node of BST
struct Node *createNode(int data)
{
    Node * new_Node = new Node;
    new_Node->left = NULL;
    new_Node->right = NULL;
    new_Node->data = data;
    return new_Node;
}

// A utility function to insert a new Node
//  with given key in BST and also maintain lcount ,Sum
struct Node * insert(Node *root, int key)
{
    // If the tree is empty, return a new Node
    if (root == NULL)
        return createNode(key);

    // Otherwise, recur down the tree
    if (root->data > key)
        root->left = insert(root->left, key);

    else if (root->data < key)
        root->right = insert(root->right, key);

    // return the (unchanged) Node pointer
    return root;
}

// function return sum of all element smaller than
// and equal to Kth smallest element
int ksmallestElementSumRec(Node *root, int k, int &count)
```

```
{
    // Base cases
    if (root == NULL)
        return 0;
    if (count > k)
        return 0;

    // Compute sum of elements in left subtree
    int res = ksmallestElementSumRec(root->left, k, count);
    if (count >= k)
        return res;

    // Add root's data
    res += root->data;

    // Add current Node
    count++;
    if (count >= k)
      return res;

    // If count is less than k, return right subtree Nodes
    return res + ksmallestElementSumRec(root->right, k, count);
}

// Wrapper over ksmallestElementSumRec()
int ksmallestElementSum(struct Node *root, int k)
{
   int count = 0;
   ksmallestElementSumRec(root, k, count);
}

/* Driver program to test above functions */
int main()
{

    /*    20
        /    \
       8     22
     /   \
    4     12
         /   \
        10    14
          */
    Node *root = NULL;
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 4);
    root = insert(root, 12);
```

```
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 22);

    int k = 3;
    cout <<  ksmallestElementSum(root, k) <<endl;
    return 0;
}
```

Output :

```
22
```

Time complexity : O(k)

Method 2 (Efficient and changes structure of BST)

We can find the required sum in O(h) time where h is height of BST. Idea is similar to Kth-th smallest element in BST. Here we use augmented tree data structure to solve this problem efficiently in O(h) time [h is height of BST] .

Algorithm :

```
BST Node contain to extra fields : Lcount , Sum

For each Node of BST
lCount : store how many left child it has
Sum    : store sum of all left child it has

Find Kth smallest element
[ temp_sum store sum of all element less than equal to K ]

ksmallestElementSumRec(root, K, temp_sum)

  IF root -> lCount == K + 1
      temp_sum += root->data + root->sum;
      break;
  ELSE
     IF k > root->lCount   // Goto right sub-tree
        temp_sum += root->data + root-> sum;
        ksmallestElementSumRec(root->right, K-root->lcount+1, temp_sum)
     ELSE
        // Goto left sun-tree
        ksmallestElementSumRec( root->left, K, temp_sum)
```

Below is C++ implementation of above algo :

```
 // C++ program to find Sum Of All Elements smaller
// than or equal t Kth Smallest Element In BST
#include <bits/stdc++.h>
using namespace std;

/* Binary tree Node */
struct Node
{
    int data;
    int lCount;
    int Sum ;
    Node* left;
    Node* right;
};

//utility function new Node of BST
struct Node *createNode(int data)
{
    Node * new_Node = new Node;
    new_Node->left = NULL;
    new_Node->right = NULL;
    new_Node->data = data;
    new_Node->lCount = 0 ;
    new_Node->Sum = 0;
    return new_Node;
}

// A utility function to insert a new Node with
// given key in BST and also maintain lcount ,Sum
struct Node * insert(Node *root, int key)
{
    // If the tree is empty, return a new Node
    if (root == NULL)
        return createNode(key);

    // Otherwise, recur down the tree
    if (root->data > key)
    {
        // increment lCount of current Node
        root->lCount++;

        // increment current Node sum by adding
        // key into it
        root->Sum += key;

        root->left= insert(root->left , key);
    }
    else if (root->data < key )
```

```
        root->right= insert (root->right , key );

    // return the (unchanged) Node pointer
    return root;
}

// function return sum of all element smaller than and equal
// to Kth smallest element
void ksmallestElementSumRec(Node *root, int k , int &temp_sum)
{
    if (root == NULL)
        return ;

    // if we fount k smallest element then break the function
    if ((root->lCount + 1) == k)
    {
        temp_sum += root->data + root->Sum ;
        return ;
    }

    else if (k > root->lCount)
    {
        // store sum of all element smaller than current root ;
        temp_sum += root->data + root->Sum;

        // decremented k and call right sub-tree
        k = k -( root->lCount + 1);
        ksmallestElementSumRec(root->right , k , temp_sum);
    }
    else // call left sub-tree
        ksmallestElementSumRec(root->left , k , temp_sum );
}

// Wrapper over ksmallestElementSumRec()
int ksmallestElementSum(struct Node *root, int k)
{
    int sum = 0;
    ksmallestElementSumRec(root, k, sum);
    return sum;
}

/* Driver program to test above functions */
int main()
{
    /*    20
        /    \
       8     22
     /   \
```

```
    4     12
         /   \
        10    14
          */
    Node *root = NULL;
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 22);

    int k = 3;
    cout <<  ksmallestElementSum(root, k) << endl;
    return 0;
}
```

Output:

```
 22
```

Source

https://www.geeksforgeeks.org/sum-k-smallest-elements-bst/

Chapter 104

Threaded Binary Search Tree Deletion

Threaded Binary Search Tree Deletion - GeeksforGeeks

A threaded binary tree node looks like following.

```
 struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to predecessor
    // in Inorder Traversal
    bool rthread;
};
```

We have already discussed Insertion of Threaded Binary Search Tree
In deletion, first the key to be deleted is searched, and then there are differenct cases for deleting the Node in which key is found.

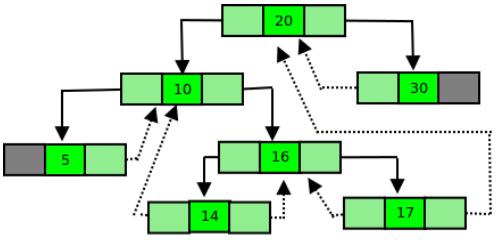
```
 // Deletes a key from threaded BST with given root and
// returns new root of BST.
struct Node *delThreadedBST(struct Node* root, int dkey)
{
    // Initialize parent as NULL and ptrent
    // Node as root.
    struct Node *par = NULL, *ptr = root;
```

```
// Set true if key is found
int found = 0;

// Search key in BST : find Node and its
// parent.
while (ptr != NULL)
{
    if (dkey == ptr->info)
    {
        found = 1;
        break;
    }
    par = ptr;
    if (dkey < ptr->info)
    {
        if (ptr->lthread == false)
            ptr = ptr -> left;
        else
            break;
    }
    else
    {
        if (ptr->rthread == false)
            ptr = ptr->right;
        else
            break;
    }
}

if (found == 0)
    printf("dkey not present in tree\n");

// Two Children
else if (ptr->lthread == false && ptr->rthread == false)
    root = caseC(root, par, ptr);

// Only Left Child
else if (ptr->lthread == false)
    root = caseB(root, par, ptr);

// Only Right Child
else if (ptr->rthread == false)
    root = caseB(root, par, ptr);

// No child
else
    root = caseA(root, par, ptr);
```

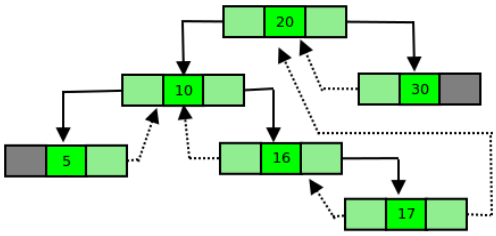
```
    return root;
}
```

Case A: Leaf Node need to be deleted
In BST, for deleting a leaf Node the left or right pointer of parent was set to NULL. Here instead of setting the pointer to NULL it is made a thread.
If the leaf Node is to be deleted is left child of its parent then after deletion, left pointer of parent should become a thread pointing to its predecessor of the parent Node after deletion.

```
par -> lthread = true;
par -> left = ptr -> left;
```

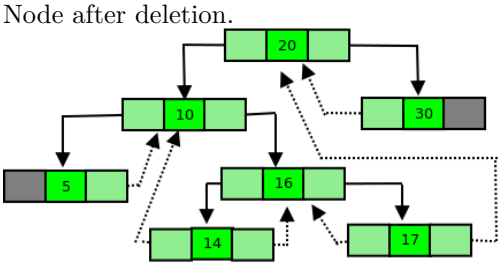


Delete 14
Inorder: 5 10 14 16 17 20 30

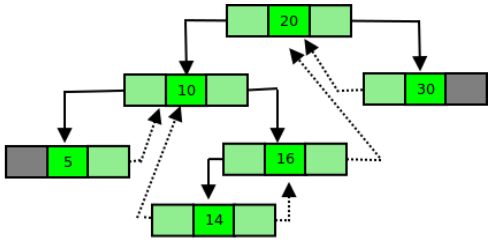


Inorder: 5 10 16 17 20 30

If the leaf Node to be deleted is right child of its parent then after deletion, right pointer of parent should become a thread pointing to its successor. The Node which was inorder successor of the leaf Node before deletion will become the inorder successor of the parent Node after deletion.



Delete 17
Inorder: 5 10 14 16 17 20 30



Inorder: 5 10 14 16 20 30

```
 // Here 'par' is pointer to parent Node and 'ptr' is
// pointer to current Node.
struct Node *caseA(struct Node *root, struct Node *par,
                                      struct Node *ptr)
{
    // If Node to be deleted is root
    if (par == NULL)
        root = NULL;

    // If Node to be deleted is left
```

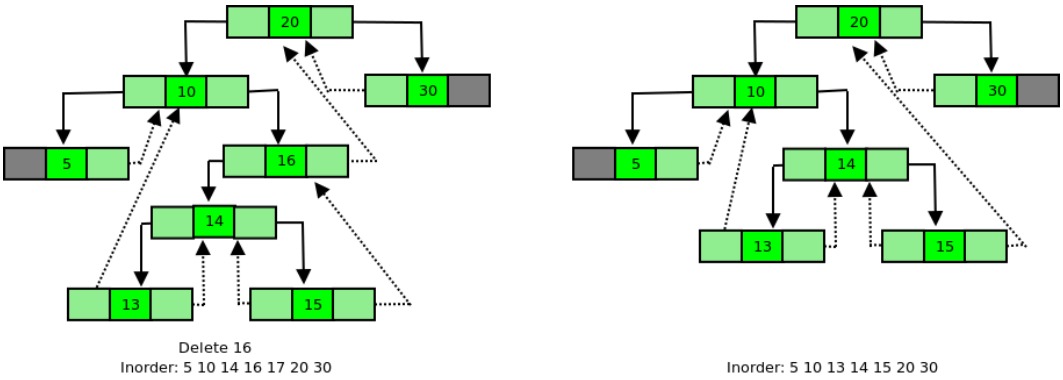
```
    // of its parent
    else if (ptr == par->left)
    {
        par->lthread = true;
        par->left = ptr->left;
    }
    else
    {
        par->rthread = true;
        par->right = ptr->right;
    }

    // Free memory and return new root
    free(ptr);
    return root;
}
```

Case B: Node to be deleted has only one child
After deleting the Node as in a BST, the inorder successor and inorder predecessor of the Node are found out.

```
s = inSucc(ptr);
p = inPred(ptr);
```

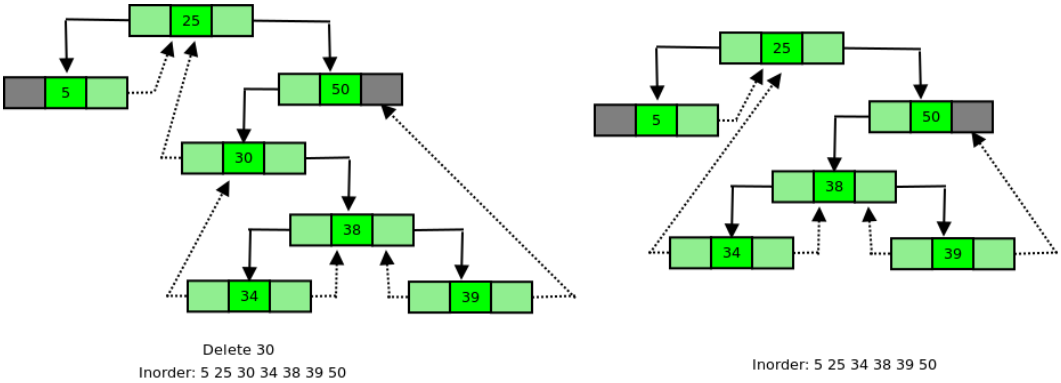
If Node to be deleted has left subtree, then after deletion right thread of its predecessor should point to its successor.

```
p->left = s;
```



Before deletion 15 is predecessor and 2 is successor of 16. After deletion of 16, the Node 20 becomes the successor of 15, so right thread of 15 will point to 20.
If Node to be deleted has right subtree, then after deletion left thread of its successor should point to its prredecessor.

```
s->left = p;
```



Before deletion of 25 is predecessor and 34 is successor of 30. After deletion of 30, the Node 25 becomes the predecessor of 34, so left thread of 34 will point to 25.

```
 // Here 'par' is pointer to parent Node and 'ptr' is
// pointer to current Node.
struct Node *caseB(struct Node *root, struct Node *par,
                                        struct Node *ptr)
{
    struct Node *child;

    // Initialize child Node to be deleted has
    // left child.
    if (ptr->lthread == false)
        child = ptr->left;

    // Node to be deleted has right child.
    else
        child = ptr->right;

    // Node to be deleted is root Node.
    if (par == NULL)
        root = child;

    // Node is left child of its parent.
    else if (ptr == par->left)
        par->left = child;
    else
        par->right = child;

    // Find successor and predecessor
    Node *s = inSucc(ptr);
    Node *p = inPred(ptr);

    // If ptr has left subtree.
    if (ptr->lthread == false)
        p->right = s;
```

```
    // If ptr has right subtree.
    else
    {
        if (ptr->rthread == false)
            s->left = p;
    }

    free(ptr);
    return root;
}
```

Case C: Node to be deleted has two children
We find inorder successor of Node ptr (Node to be deleted) and then copy the information of this successor into Node ptr. After this inorder successor Node is deleted using either Case A or Case B.

```
 // Here 'par' is pointer to parent Node and 'ptr' is
// pointer to current Node.
struct Node *caseC(struct Node *root, struct Node *par,
                                        struct Node *ptr)
{
    // Find inorder successor and its parent.
    struct Node *parsucc = ptr;
    struct Node *succ = ptr -> right;

    // Find leftmost child of successor
    while (succ->left != NULL)
    {
        parsucc = succ;
        succ = succ -> left;
    }

    ptr->info = succ->info;

    if (succ->lthread == true && succ->rthread == true)
        root = caseA(root, parsucc, succ);
    else
        root = caseB(root, parsucc, succ);

    return root;
}
```

Below is Complete C++ code:

```
 // Complete C++ program to demonstrate deletion
// in threaded BST
```

```
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to predecessor
    // in Inorder Traversal
    bool rthread;
};

// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
    // Searching for a Node with given value
    Node *ptr = root;
    Node *par = NULL; // Parent of key to be inserted
    while (ptr != NULL)
    {
        // If key already exists, return
        if (ikey == (ptr->info))
        {
            printf("Duplicate Key !\n");
            return root;
        }

        par = ptr; // Update parent pointer

        // Moving on left subtree.
        if (ikey < ptr->info)
        {
            if (ptr -> lthread == false)
                ptr = ptr -> left;
            else
                break;
        }

        // Moving on right subtree.
        else
        {
            if (ptr->rthread == false)
                ptr = ptr -> right;
```

```
            else
                break;
        }
    }

    // Create a new Node
    Node *tmp = new Node;
    tmp -> info = ikey;
    tmp -> lthread = true;
    tmp -> rthread = true;

    if (par == NULL)
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))
    {
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
    }
    else
    {
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
    }

    return root;
}

// Returns inorder successor using left
// and right children (Used in deletion)
struct Node *inSucc(struct Node *ptr)
{
    if (ptr->rthread == true)
        return ptr->right;

    ptr = ptr -> right;
    while (ptr->right)
        ptr = ptr->left;

    return ptr;
}
```

```
// Returns inorder successor using rthread
// (Used in inorder)
struct Node *inorderSuccessor(struct Node *ptr)
{
    // If rthread is set, we can quickly find
    if (ptr -> rthread == true)
        return ptr->right;

    // Else return leftmost child of right subtree
    ptr = ptr -> right;
    while (ptr -> lthread == false)
        ptr = ptr -> left;
    return ptr;
}

// Printing the threaded tree
void inorder(struct Node *root)
{
    if (root == NULL)
        printf("Tree is empty");

    // Reach leftmost Node
    struct Node *ptr = root;
    while (ptr -> lthread == false)
        ptr = ptr -> left;

    // One by one print successors
    while (ptr != NULL)
    {
        printf("%d ",ptr -> info);
        ptr = inorderSuccessor(ptr);
    }
}

struct Node *inPred(struct Node *ptr)
{
    if (ptr->lthread == true)
        return ptr->right;

    ptr = ptr->left;
    while (ptr->rthread);
        ptr = ptr->right;
    return ptr;
}


// Here 'par' is pointer to parent Node and 'ptr' is
```

```
// pointer to current Node.
struct Node *caseA(struct Node *root, struct Node *par,
                                        struct Node *ptr)
{
    // If Node to be deleted is root
    if (par == NULL)
        root = NULL;

    // If Node to be deleted is left
    // of its parent
    else if (ptr == par->left)
    {
        par->lthread = true;
        par->left = ptr->left;
    }
    else
    {
        par->rthread = true;
        par->right = ptr->right;
    }

    // Free memory and return new root
    free(ptr);
    return root;
}

// Here 'par' is pointer to parent Node and 'ptr' is
// pointer to current Node.
struct Node *caseB(struct Node *root, struct Node *par,
                                        struct Node *ptr)
{
    struct Node *child;

    // Initialize child Node to be deleted has
    // left child.
    if (ptr->lthread == false)
        child = ptr->left;

    // Node to be deleted has right child.
    else
        child = ptr->right;

    // Node to be deleted is root Node.
    if (par == NULL)
        root = child;

    // Node is left child of its parent.
    else if (ptr == par->left)
```

```
        par->left = child;
    else
        par->right = child;

    // Find successor and predecessor
    Node *s = inSucc(ptr);
    Node *p = inPred(ptr);

    // If ptr has left subtree.
    if (ptr->lthread == false)
        p->right = s;

    // If ptr has right subtree.
    else
    {
        if (ptr->rthread == false)
            s->left = p;
    }

    free(ptr);
    return root;
}

// Here 'par' is pointer to parent Node and 'ptr' is
// pointer to current Node.
struct Node *caseC(struct Node *root, struct Node *par,
                                        struct Node *ptr)
{
    // Find inorder successor and its parent.
    struct Node *parsucc = ptr;
    struct Node *succ = ptr -> right;

    // Find leftmost child of successor
    while (succ->left != NULL)
    {
        parsucc = succ;
        succ = succ -> left;
    }

    ptr->info = succ->info;

    if (succ->lthread == true && succ->rthread == true)
        root = caseA(root, parsucc, succ);
    else
        root = caseB(root, parsucc, succ);

    return root;
}
```

```
// Deletes a key from threaded BST with given root and
// returns new root of BST.
struct Node *delThreadedBST(struct Node* root, int dkey)
{
    // Initialize parent as NULL and ptrent
    // Node as root.
    struct Node *par = NULL, *ptr = root;

    // Set true if key is found
    int found = 0;

    // Search key in BST : find Node and its
    // parent.
    while (ptr != NULL)
    {
        if (dkey == ptr->info)
        {
            found = 1;
            break;
        }
        par = ptr;
        if (dkey < ptr->info)
        {
            if (ptr->lthread == false)
                ptr = ptr -> left;
            else
                break;
        }
        else
        {
            if (ptr->rthread == false)
                ptr = ptr->right;
            else
                break;
        }
    }

    if (found == 0)
        printf("dkey not present in tree\n");

    // Two Children
    else if (ptr->lthread == false && ptr->rthread == false)
        root = caseC(root, par, ptr);

    // Only Left Child
    else if (ptr->lthread == false)
        root = caseB(root, par, ptr);
```

```
    // Only Right Child
    else if (ptr->rthread == false)
        root = caseB(root, par, ptr);

    // No child
    else
        root = caseA(root, par, ptr);

    return root;
}

// Driver Program
int main()
{
    struct Node *root = NULL;

    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 16);
    root = insert(root, 14);
    root = insert(root, 17);
    root = insert(root, 13);

    root = delThreadedBST(root, 20);
    inorder(root);

    return 0;
}
```

Output :

```
10 13 14 16 17 5 30
```

Source

https://www.geeksforgeeks.org/threaded-binary-search-tree-deletion/

Chapter 105

Threaded Binary Tree Insertion

Threaded Binary Tree Insertion - GeeksforGeeks

We have already discuss the Binary Threaded Binary Tree.

Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

C representation of Binary Threaded Node:

```
struct Node
{
  struct Node *left, *right;
  int info;

  // True if left pointer points to predecessor
  // in Inorder Traversal
  boolean lthread;

  // True if right pointer points to successor
  // in Inorder Traversal
  boolean rthread;
};
```

In the following explanation, we have considered Binary Search Tree (BST) for insertion as insertion is defined by some rules in BSTs.

Let **tmp be the newly inserted node**. There can be three cases during insertion:

Case 1: Insertion in empty tree
Both left and right pointers of tmp will be set to NULL and new node becomes the root.

```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

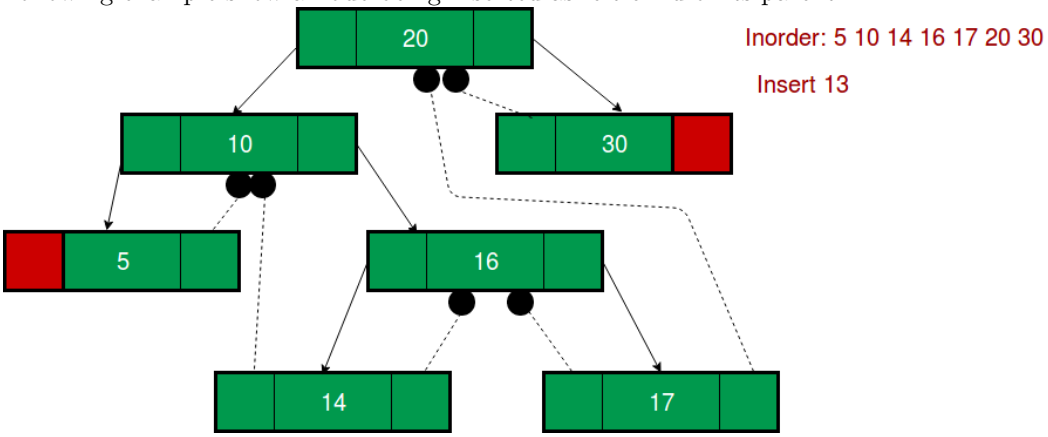
Case 2: When new node inserted as the left child
After inserting the node at its proper place we have to make its left and right threads points to inorder predecessor and successor respectively. The node which was inorder successor. So the left and right threads of the new node will be-

```
tmp -> left = par ->left;
tmp -> right = par;
```

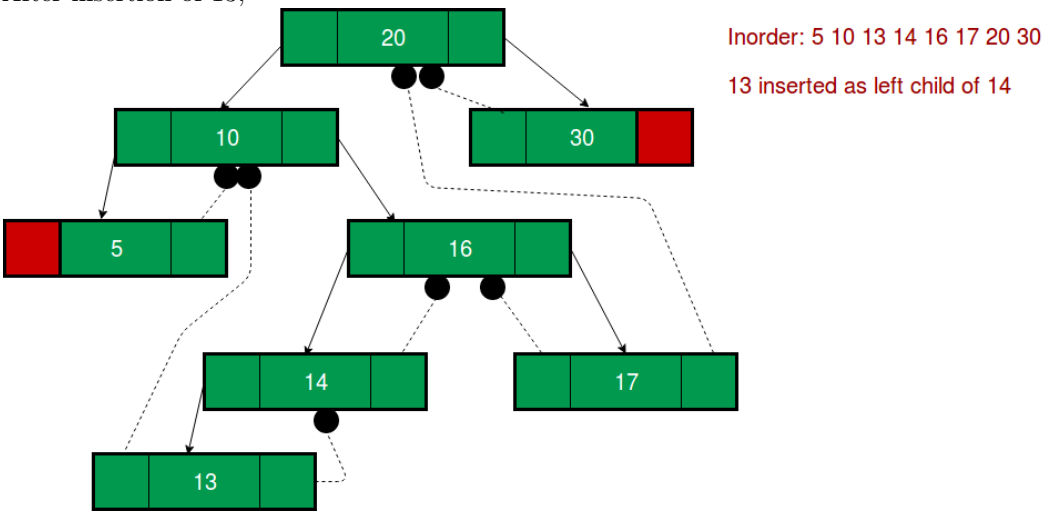
Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par -> lthread = false;
par -> left = temp;
```

Following example show a node being inserted as left child of its parent.



After insertion of 13,



Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.

Successor of 13 is 14, so right thread of 13 points to left child which is 13.
Left pointer of 14 is not a thread now, it points to left child which is 13.

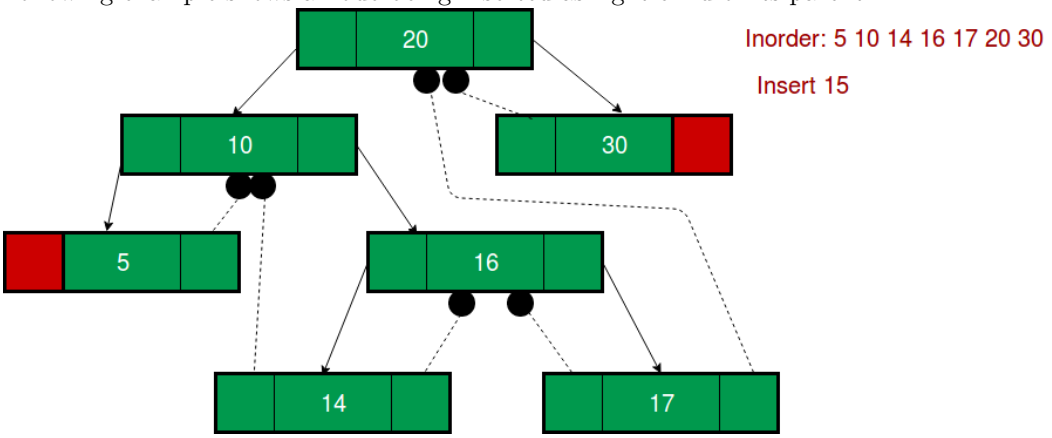
Case 3: When new node is inserted as the right child
The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

```
tmp -> left = par;
tmp -> right = par -> right;
```

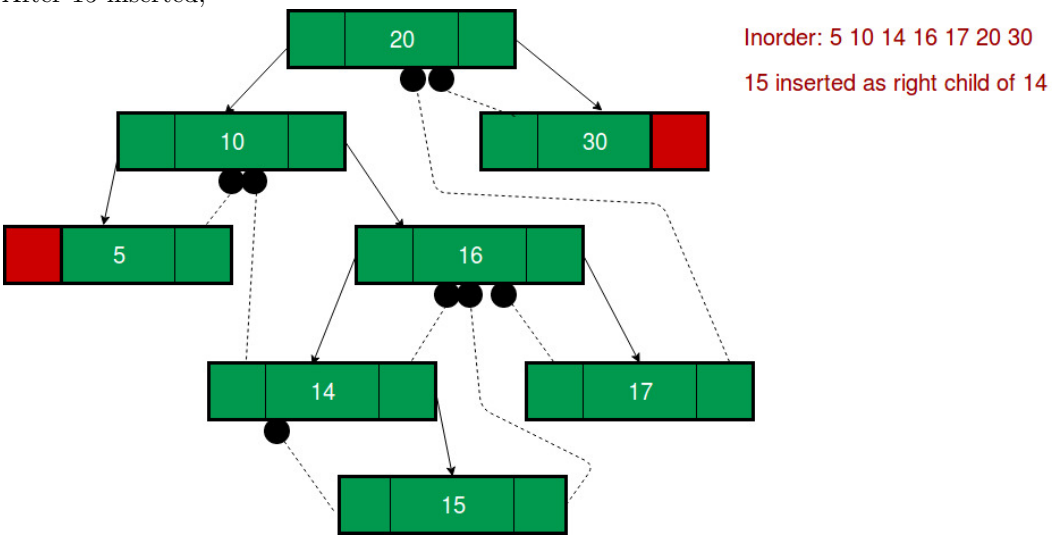
Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par -> rthread = false;
par -> right = tmp;
```

Following example shows a node being inserted as right child of its parent.



After 15 inserted,



Successor of 14 becomes the successor of 15, so right thread of 15 points to 16
Predecessor of 15 is 14, so left thread of 15 points to 14.
Right pointer of 14 is not a thread now, it points to right child which is 15.

C++ implementation to insert a new node in Threaded Binary Search Tree:
Like standard BST insert, we search for the key value in the tree. If key is already present, then we return otherwise the new key is inserted at the point where search terminates. In BST, search terminates either when we find the key or when we reach a NULL left or right pointer. Here all left and right NULL pointers are replaced by threads except left pointer of first node and right pointer of last node. So here search will be unsuccessful when we reach a NULL pointer or a thread.

```
 // Insertion in Threaded Binary Search Tree.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to predecessor
    // in Inorder Traversal
    bool rthread;
};

// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
    // Searching for a Node with given value
    Node *ptr = root;
    Node *par = NULL; // Parent of key to be inserted
    while (ptr != NULL)
    {
        // If key already exists, return
        if (ikey == (ptr->info))
        {
            printf("Duplicate Key !\n");
            return root;
        }

        par = ptr; // Update parent pointer

        // Moving on left subtree.
        if (ikey < ptr->info)
```

```
        {
            if (ptr -> lthread == false)
                ptr = ptr -> left;
            else
                break;
        }

        // Moving on right subtree.
        else
        {
            if (ptr->rthread == false)
                ptr = ptr -> right;
            else
                break;
        }
    }

    // Create a new node
    Node *tmp = new Node;
    tmp -> info = ikey;
    tmp -> lthread = true;
    tmp -> rthread = true;

    if (par == NULL)
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))
    {
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
    }
    else
    {
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
    }

    return root;
}

// Returns inorder successor using rthread
```

```
struct Node *inorderSuccessor(struct Node *ptr)
{
    // If rthread is set, we can quickly find
    if (ptr -> rthread == true)
        return ptr->right;

    // Else return leftmost child of right subtree
    ptr = ptr -> right;
    while (ptr -> lthread == false)
        ptr = ptr -> left;
    return ptr;
}

// Printing the threaded tree
void inorder(struct Node *root)
{
    if (root == NULL)
        printf("Tree is empty");

    // Reach leftmost node
    struct Node *ptr = root;
    while (ptr -> lthread == false)
        ptr = ptr -> left;

    // One by one print successors
    while (ptr != NULL)
    {
        printf("%d ",ptr -> info);
        ptr = inorderSuccessor(ptr);
    }
}

// Driver Program
int main()
{
    struct Node *root = NULL;

    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 16);
    root = insert(root, 14);
    root = insert(root, 17);
    root = insert(root, 13);

    inorder(root);
```

```
    return 0;
}
```

Output:

```
5 10 13 14 16 17 20 30
```

Source

https://www.geeksforgeeks.org/threaded-binary-tree-insertion/

Chapter 106

Total number of possible Binary Search Trees and Binary Trees with n keys

Total number of possible Binary Search Trees and Binary Trees with n keys - GeeksforGeeks

Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = (2n)!/(n+1)!*n!

For n = 0, 1, 2, 3, … values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …. So are numbers of Binary Search Trees.

Total number of possible Binary Trees with n different keys (countBT(n)) = countBST(n) * n!

Below is code for finding count of BSTs and Binary Trees with n numbers. The code to find n'th Catalan number is taken from here.

C++

```
 // See https://www.geeksforgeeks.org/program-nth-catalan-number/
// for reference of below code.

#include <bits/stdc++.h>
using namespace std;

// A function to find factorial of a given number
unsigned long int factorial(unsigned int n)
{
    unsigned long int res = 1;

    // Calculate value of [1*(2)*---*(n-k+1)] / [k*(k-1)*---*1]
    for (int i = 1; i <= n; ++i)
```

```
    {
        res *= i;
    }

    return res;
}

unsigned long int binomialCoeff(unsigned int n, unsigned int k)
{
    unsigned long int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}


// A Binomial coefficient based function to find nth catalan
// number in O(n) time
unsigned long int catalan(unsigned int n)
{
    // Calculate value of 2nCn
    unsigned long int c = binomialCoeff(2*n, n);

    // return 2nCn/(n+1)
    return c/(n+1);
}

// A function to count number of BST with n nodes
// using catalan
unsigned long int countBST(unsigned int n)
{
    // find nth catalan number
    unsigned long int count = catalan(n);

    // return nth catalan number
    return count;
}
```

```
// A function to count number of binary trees with n nodes
unsigned long int countBT(unsigned int n)
{
    // find count of BST with n numbers
    unsigned long int count = catalan(n);

    // return count * n!
    return count * factorial(n);
}

// Driver Program to test above functions
int main()
{

    int count1,count2, n = 5;

    // find count of BST and binary trees with n nodes
        count1 = countBST(n);
        count2 = countBT(n);

    // print count of BST and binary trees with n nodes
    cout<<"Count of BST with "<<n<<" nodes is "<<count1<<endl;
        cout<<"Count of binary trees with "<<n<<" nodes is "<<count2;

    return 0;
}
```

Java

```
 // See https://www.geeksforgeeks.org/program-nth-catalan-number/
// for reference of below code.
import java.io.*;

class GFG
{

// A function to find
// factorial of a given number
static int factorial(int n)
{
    int res = 1;

    // Calculate value of
    // [1*(2)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for (int i = 1; i <= n; ++i)
    {
        res *= i;
```

```
    }

    return res;
}

static int binomialCoeff(int n,
                         int k)
{
    int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of
    // [n*(n-1)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}


// A Binomial coefficient
// based function to find
// nth catalan number in
// O(n) time
static int catalan( int n)
{

    // Calculate value of 2nCn
    int c = binomialCoeff(2 * n, n);

    // return 2nCn/(n+1)
    return c / (n + 1);
}

// A function to count number of
// BST with n nodes using catalan
static int countBST( int n)
{
    // find nth catalan number
    int count = catalan(n);
```

```
    // return nth catalan number
    return count;
}

// A function to count number
// of binary trees with n nodes
static int countBT(int n)
{
    // find count of BST
    // with n numbers
    int count = catalan(n);

    // return count * n!
    return count * factorial(n);
}

// Driver Code
public static void main (String[] args)
{
    int count1, count2, n = 5;

    // find count of BST and
    // binary trees with n nodes
    count1 = countBST(n);
    count2 = countBT(n);

    // print count of BST and
    // binary trees with n nodes
    System.out.println("Count of BST with "+
                            n +" nodes is "+
                                    count1);
    System.out.println("Count of binary " +
                              "trees with "+
                          n + " nodes is " +
                                    count2);
}
}

// This code is contributed by ajit
```

C#

```
 // See https://www.geeksforgeeks.org/program-nth-catalan-number/
// for reference of below code.
using System;

class GFG
{
```

```
// A function to find
// factorial of a given number
static int factorial(int n)
{
    int res = 1;

    // Calculate value of
    // [1*(2)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for (int i = 1; i <= n; ++i)
    {
        res *= i;
    }

    return res;
}

static int binomialCoeff(int n,
                         int k)
{
    int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of
    // [n*(n-1)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

// A Binomial coefficient
// based function to find
// nth catalan number in
// O(n) time
static int catalan(int n)
{

    // Calculate value
    // of 2nCn
```

```
    int c = binomialCoeff(2 * n, n);

    // return 2nCn/(n+1)
    return c / (n + 1);
}

// A function to count
// number of BST with
// n nodes using catalan
static int countBST(int n)
{
    // find nth catalan number
    int count = catalan(n);

    // return nth catalan number
    return count;
}

// A function to count number
// of binary trees with n nodes
static int countBT(int n)
{
    // find count of BST
    // with n numbers
    int count = catalan(n);

    // return count * n!
    return count * factorial(n);
}

// Driver Code
static public void Main ()
{
    int count1, count2, n = 5;

    // find count of BST
    // and binary trees
    // with n nodes
    count1 = countBST(n);
    count2 = countBT(n);

    // print count of BST and
    // binary trees with n nodes
    Console.WriteLine("Count of BST with "+
                           n +" nodes is "+
                                   count1);
    Console.WriteLine("Count of binary " +
                            "trees with "+
```

```
                        n + " nodes is " +
                                    count2);
    }
}

// This code is contributed
// by akt_mit
```

PHP

```
 <?php
// See https://www.geeksforgeeks.org/program-nth-catalan-number/
// for reference of below code.
// A function to find factorial
// of a given number
function factorial($n)
{
    $res = 1;

    // Calculate value of
    // [1*(2)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for ($i = 1; $i <= $n; ++$i)
    {
        $res *= $i;
    }

    return $res;
}

function binomialCoeff($n, $k)
{
    $res = 1;

    // Since C(n, k) = C(n, n-k)
    if ($k > $n - $k)
        $k = $n - $k;

    // Calculate value of
    // [n*(n-1)*---*(n-k+1)] /
    // [k*(k-1)*---*1]
    for ($i = 0; $i < $k; ++$i)
    {
        $res *= ($n - $i);
        $res = (int)$res / ($i + 1);
    }

    return $res;
```

```
}

// A Binomial coefficient
// based function to find
// nth catalan number in
// O(n) time
function catalan($n)
{
    // Calculate value of 2nCn
    $c = binomialCoeff(2 * $n, $n);

    // return 2nCn/(n+1)
    return (int)$c / ($n + 1);
}

// A function to count
// number of BST with
// n nodes using catalan
function countBST($n)
{
    // find nth catalan number
    $count = catalan($n);

    // return nth
    // catalan number
    return $count;
}

// A function to count
// number of binary
// trees with n nodes
function countBT($n)
{
    // find count of
    // BST with n numbers
    $count = catalan($n);

    // return count * n!
    return $count *
           factorial($n);
}

// Driver Code
$count1;
$count2;
$n = 5;

// find count of BST and
```

```
// binary trees with n nodes
$count1 = countBST($n);
$count2 = countBT($n);

// print count of BST and
// binary trees with n nodes
echo "Count of BST with " , $n ,
     " nodes is ", $count1,"\n";

echo "Count of binary trees with " ,
           $n ," nodes is ",$count2;

// This code is contributed by ajit
?>
```

Output:

```
Count of BST with 5 nodes is 42
Count of binary trees with 5 nodes is 5040
```

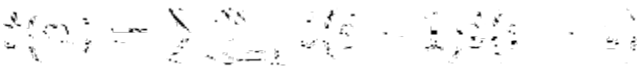
Proof of Enumeration

Consider all possible binary search trees with each element at the root. If there are n nodes, then for each choice of root node, there are n – 1 non-root nodes and these non-root nodes must be partitioned into those that are less than a chosen root and those that are greater than the chosen root.

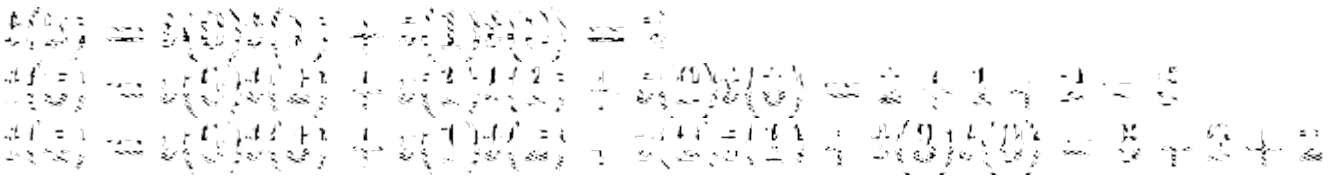
Let's say node i is chosen to be the root. Then there are i – 1 nodes smaller than i and n – i nodes bigger than i. For each of these two sets of nodes, there is a certain number of possible subtrees.

Let t(n) be the total number of BSTs with n nodes. The total number of BSTs with i at the root is t(i – 1) t(n – i). The two terms are multiplied together because the arrangements in the left and right subtrees are independent. That is, for each arrangement in the left tree and for each arrangement in the right tree, you get one BST with i at the root.

Summing over i gives the total number of binary search trees with n nodes.



The base case is t(0) = 1 and t(1) = 1, i.e. there is one empty BST and there is one BST with one node.



Also, the relationship countBT(n) = countBST(n) * n! holds. As for every possible BST, there can have n! binary trees where n is the number of nodes in BST.

Improved By : jit_t

Source

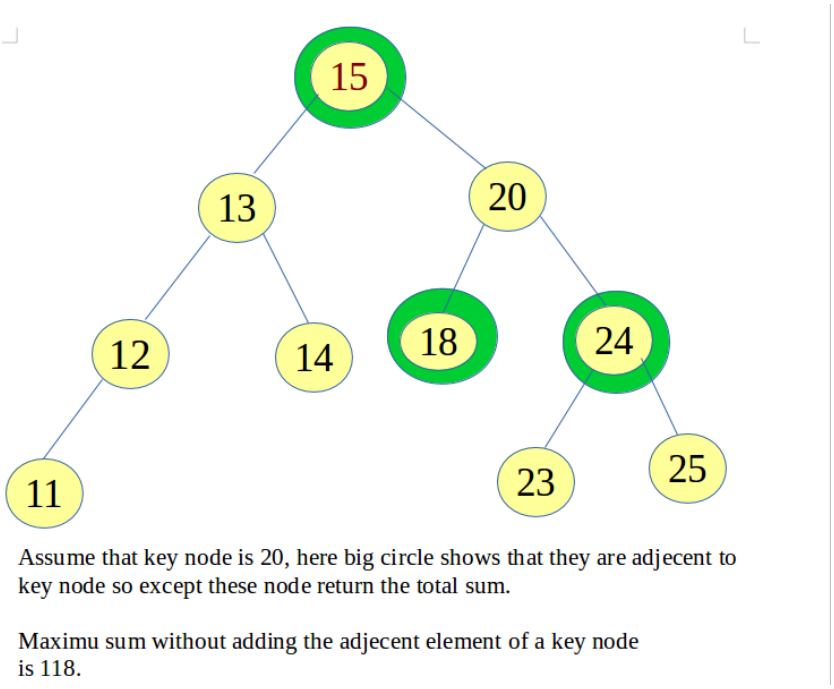
https://www.geeksforgeeks.org/total-number-of-possible-binary-search-trees-with-n-keys/

Chapter 107

Total sum except adjacent of a given node in BST

Total sum except adjacent of a given node in BST - GeeksforGeeks

Given a BST and a key Node, find the total sum in BST, except those Node which are adjacent to key Node.
Examples:



1:-First find the total sum of BST
2:-Search the key Node and trace its parent Node.

3:-If the key Node is present then, subtract the sum of its adjacent Node from total sum
4:-If key is not present in BST then return -1.

```
 // C++ program to find total sum except a given Node in BST
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// insertion of Node in Tree
Node* getNode(int n)
{
    struct Node* root = new Node;
    root->data = n;
    root->left = NULL;
    root->right = NULL;
    return root;
}

// total sum of bst
int sum(struct Node* root)
{
    if (root == NULL)
        return 0;

    return root->data + sum(root->left) + sum(root->right);
}

// sum of all element except those which are adjecent to key Node
int adjSum(Node* root, int key)
{

    int parent = root->data;

    while (root != NULL) {
        if (key < root->data) {
            parent = root->data;
            root = root->left;
        }
        else if (root->data == key) // key Node matches
        {
            // if the left Node and right Node of key is
            // not null then add all adjecent Node and
            // substract from totalSum
            if (root->left != NULL && root->right != NULL)
```

```
                return (parent + root->left->data +
                                 root->right->data);

            // if key is leaf
            if (root->left == NULL && root->right == NULL)
                return parent;

            // If only left child is null
            if (root->left == NULL)
                return (parent + root->right->data);

            // If only right child is NULL
            if (root->right == NULL)
                return (parent + root->left->data);
        }

        else {
            parent = root->data;
            root = root->right;
        }
    }

    return 0;
}

int findTotalExceptKey(Node *root, int key)
{
    return sum(root) - adjSum(root, key);
}

// Driver code
int main()
{
    struct Node* root = getNode(15);
    root->left = getNode(13);
    root->left->left = getNode(12);
    root->left->left->left = getNode(11);
    root->left->right = getNode(14);
    root->right = getNode(20);
    root->right->left = getNode(18);
    root->right->right = getNode(24);
    root->right->right->left = getNode(23);
    root->right->right->right = getNode(25);
    int key = 20;
    printf("%d ", findTotalExceptKey(root, key));
    return 0;
}
```

Output:

```
118
```

Time Complexity : O(n) + O(h) where n is number of nodes in BST and h is height of BST. We can write time complexity as O(n).

Source

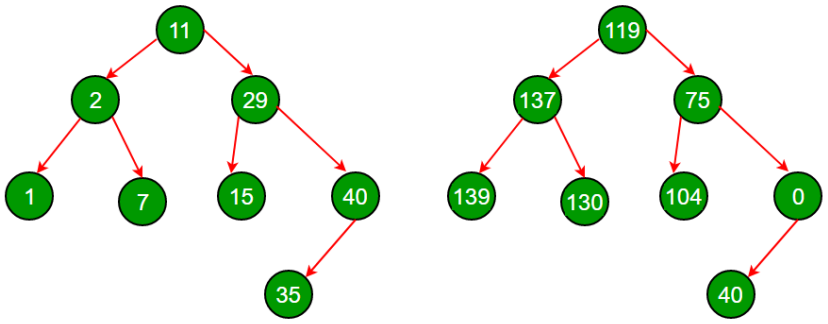
https://www.geeksforgeeks.org/total-sum-except-adjacent-of-a-given-node-in-bst/

Chapter 108

Transform a BST to greater sum tree

Transform a BST to greater sum tree - GeeksforGeeks

Given a BST, transform it into greater sum tree where each node contains sum of all nodes greater than that node.



We strongly recommend to minimize the gbrowser and try this yourself first.

Method 1 (Naive):
This method doesn't require the tree to be a BST. Following are steps.
1. Traverse node by node(Inorder, preorder, etc.)
2. For each node find all the nodes greater than that of the current node, sum the values. Store all these sums.
3. Replace each node value with their corresponding sum by traversing in the same order as in Step 1.
This takes O(n^2) Time Complexity.

Method 2 (Using only one traversal)
By leveraging the fact that the tree is a BST, we can find an O(n) solution. The idea is to traverse BST in reverse inorder. Reverse inorder traversal of a BST gives us keys in

decreasing order. Before visiting a node, we visit all greater nodes of that node. While traversing we keep track of sum of keys which is the sum of all the keys greater than the key of current node.

```
 // C++ program to transform a BST to sum tree
#include<iostream>
using namespace std;

// A BST node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp =  new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to transform a BST to sum tree.
// This function traverses the tree in reverse inorder so
// that we have visited all greater key nodes of the currently
// visited node
void transformTreeUtil(struct Node *root, int *sum)
{
   // Base case
   if (root == NULL)  return;

   // Recur for right subtree
   transformTreeUtil(root->right, sum);

   // Update sum
   *sum = *sum + root->data;

   // Store old sum in current node
   root->data = *sum - root->data;

   // Recur for left subtree
   transformTreeUtil(root->left, sum);
}

// A wrapper over transformTreeUtil()
void transformTree(struct Node *root)
```

```
{
    int sum = 0; // Initialize sum
    transformTreeUtil(root, &sum);
}

// A utility function to print indorder traversal of a
// binary tree
void printInorder(struct Node *root)
{
    if (root == NULL) return;

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

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(11);
    root->left = newNode(2);
    root->right = newNode(29);
    root->left->left = newNode(1);
    root->left->right = newNode(7);
    root->right->left = newNode(15);
    root->right->right = newNode(40);
    root->right->right->left = newNode(35);

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    transformTree(root);

    cout << "\n\nInorder Traversal of transformed tree\n";
    printInorder(root);

    return 0;
}
```

Output:

```
Inorder Traversal of given tree
1 2 7 11 15 29 35 40

Inorder Traversal of transformed tree
139 137 130 119 104 75 40 0
```

Time complexity of this method is O(n) as it does a simple traversal of tree.

This article is contributed by **Bhavana**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/transform-bst-sum-tree/

Chapter 109

Two nodes of a BST are swapped, correct the BST

Two nodes of a BST are swapped, correct the BST - GeeksforGeeks

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

```
Input Tree:
         10
        /  \
       5    8
      / \
     2   20

In the above tree, nodes 20 and 8 must be swapped to fix the tree.
Following is the output tree
         10
        /  \
       5    20
      / \
     2   8
```

The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiilary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is O(nLogn) and auxiliary space needed is O(n).

We can solve this in O(n) time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

```
For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5
```

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

```
For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25
```

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? *We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node. In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.*

Following is the implementation of the given code.
C++

```
// Two nodes in the BST's swapped, correct the BST.
#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, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
/* 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);
}

// This function does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last.  If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}
```

```
// A function to fix a given BST where two nodes are swapped.  This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the poiters to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inoder traversal */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    /*   6
        /  \
       10    2
      / \   / \
     1   3 7  12
     10 and 2 are swapped
    */

    struct node *root = newNode(6);
    root->left        = newNode(10);
    root->right       = newNode(2);
    root->left->left  = newNode(1);
    root->left->right = newNode(3);
```

```
    root->right->right = newNode(12);
    root->right->left = newNode(7);

    printf("Inorder Traversal of the original tree \n");
    printInorder(root);

    correctBST(root);

    printf("\nInorder Traversal of the fixed tree \n");
    printInorder(root);

    return 0;
}
```

Java

```
 // Java program to correct the BST
// if two nodes are swapped
import java.util.*;
import java.lang.*;
import java.io.*;

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{
    Node first, middle, last, prev;

    // This function does inorder traversal
    // to find out the two swapped nodes.
    // It sets three pointers, first, middle
    // and last. If the swapped nodes are
    // adjacent to each other, then first
    // and middle contain the resultant nodes
    // Else, first and last contain the
    // resultant nodes
    void correctBSTUtil( Node root)
    {
        if( root != null )
```

```
    {
        // Recur for the left subtree
        correctBSTUtil( root.left);

        // If this node is smaller than
        // the previous node, it's
        // violating the BST rule.
        if (prev != null && root.data <
                            prev.data)
        {
            // If this is first violation,
            // mark these two nodes as
            // 'first' and 'middle'
            if (first == null)
            {
                first = prev;
                middle = root;
            }

            // If this is second violation,
            // mark this node as last
            else
                last = root;
        }

        // Mark this node as previous
        prev = root;

        // Recur for the right subtree
        correctBSTUtil( root.right);
    }
}

// A function to fix a given BST where
// two nodes are swapped. This function
// uses correctBSTUtil() to find out
// two nodes and swaps the nodes to
// fix the BST
void correctBST( Node root )
{
    // Initialize pointers needed
    // for correctBSTUtil()
    first = middle = last = prev = null;

    // Set the poiters to find out
    // two nodes
    correctBSTUtil( root );
```

```
    // Fix (or correct) the tree
    if( first != null && last != null )
    {
        int temp = first.data;
        first.data = last.data;
        last.data = temp;
    }
    // Adjacent nodes swapped
    else if( first != null && middle !=
                                null )
    {
        int temp = first.data;
        first.data = middle.data;
        middle.data = temp;
    }

    // else nodes have not been swapped,
    // passed tree is really BST.
}

/* A utility function to print
 Inoder traversal */
void printInorder(Node node)
{
    if (node == null)
        return;
    printInorder(node.left);
    System.out.print(" " + node.data);
    printInorder(node.right);
}


// Driver program to test above functions
public static void main (String[] args)
{
    /*   6
        / \
       10  2
      / \ / \
     1  3 7 12

    10 and 2 are swapped
    */

    Node root = new Node(6);
    root.left = new Node(10);
    root.right = new Node(2);
    root.left.left = new Node(1);
```

```
        root.left.right = new Node(3);
        root.right.right = new Node(12);
        root.right.left = new Node(7);

        System.out.println("Inorder Traversal"+
                           " of the original tree");
        BinaryTree tree = new BinaryTree();
        tree.printInorder(root);

        tree.correctBST(root);

        System.out.println("\nInorder Traversal"+
                            " of the fixed tree");
        tree.printInorder(root);
    }
}
// This code is contributed by Chhavi
```

Output:

```
Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12
```

Time Complexity: O(n)

See thisfor different test cases of the above code.

Source

https://www.geeksforgeeks.org/fix-two-swapped-nodes-of-bst/

Chapter 110

set vs unordered_set in C++ STL

set vs unordered_set in C++ STL - GeeksforGeeks

Pre-requisite : set in C++, unordered_set in C++

Differences :

```
                |     set             | unordered_set
-------------------------------------------------------
Ordering        | increasing  order   | no ordering
                | (by default)        |

Implementation  | Self balancing BST  | Hash Table
                | like Red-Black Tree |

search time     | log(n)              | O(1) -> Average
                |                     | O(n) -> Worst Case

Insertion time  | log(n) + Rebalance  | Same as search

Deletion time   | log(n) + Rebalance  | Same as search
```

Use set when

- We need ordered data.
- We would have to print/access the data (in sorted order).
- We need predecessor/successor of elements.

- Since set is ordered, we can use functions like binary_search(), lower_bound() and upper_bound() on set elements. These functions cannot be used on unordered_set().
- See advantages of BST over Hash Table for more cases.

Use unordered_set when

- We need to keep a set of distinct elements and no ordering is required.
- We need single element access i.e. no traversal.

Examples:

```
set:
Input  :  1, 8, 2, 5, 3, 9
Output :  1, 2, 3, 5, 8, 9

Unordered_set:
Input  : 1, 8, 2, 5, 3, 9
Output : 9 3 1 8 2 5
```

If you want to look at implementation details of set and unordered_set in c++ STL, see Set Vs Map. Set allows to traverse elements in sorted order whereas Unordered_set doesn't allow to traverse elements in sorted order.

```
 // Program to print elements of set
#include <bits/stdc++.h>
using namespace std;

int main()
{
    set<int> s;
    s.insert(5);
    s.insert(1);
    s.insert(6);
    s.insert(3);
    s.insert(7);
    s.insert(2);

    cout << "Elements of set in sorted order: \n";
    for (auto it : s)
        cout << it << " ";

    return 0;
}
```

Output:

```
Elements of set in sorted order:
1 2 3 5 6 7
```

```
 // Program to print elements of set
#include <bits/stdc++.h>
using namespace std;

int main()
{
    unordered_set<int> s;
    s.insert(5);
    s.insert(1);
    s.insert(6);
    s.insert(3);
    s.insert(7);
    s.insert(2);

    cout << "Elements of unordered_set: \n";
    for (auto it : s)
        cout << it << " ";

    return 0;
}
```

Output:

```
Elements of unordered_set:
2 7 5 1 6 3
```

Predecessor/Successor in Set:
Set can be modified to find predecessor or successor whereas Unordered_set doesn't allow to find predecessor/Successor.

```
 // Program to print inorder predecessor and inorder successor
#include <bits/stdc++.h>
using namespace std;

set<int> s;

void inorderPredecessor(int key)
{
    if (s.find(key) == s.end()) {
```

```
        cout << "Key doesn't exist\n";
        return;
    }

    set<int>::iterator it;
    it = s.find(key); // get iterator of key

    // If iterator is at first position
    // Then, it doesn't have predecessor
    if (it == s.begin()) {
        cout << "No predecessor\n";
        return;
    }

    --it; // get previous element
    cout << "predecessor of " << key << " is=";
    cout << *(it) << "\n";
}

void inorderSuccessor(int key)
{
    if (s.find(key) == s.end()) {
        cout << "Key doesn't exist\n";
        return;
    }

    set<int>::iterator it;
    it = s.find(key); // get iterator of key
    ++it; // get next element

    // Iterator points to NULL (Element does
    // not exist)
    if (it == s.end())
    {
        cout << "No successor\n";
        return;
    }
    cout << "successor of " << key << " is=";
    cout << *(it) << "\n";
}

int main()
{
    s.insert(1);
    s.insert(5);
    s.insert(2);
    s.insert(9);
    s.insert(8);
```

```
    inorderPredecessor(5);
    inorderPredecessor(1);
    inorderPredecessor(8);
    inorderSuccessor(5);
    inorderSuccessor(2);
    inorderSuccessor(9);

    return 0;
}
```

Output:

```
predecessor of 5 is=2
No predecessor
predecessor of 8 is=5
successor of 5 is=8
successor of 2 is=5
No successor
```

Source

https://www.geeksforgeeks.org/set-vs-unordered_set-c-stl/