1. **Search a key in 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);

}

1. **Insertion 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;

}

1. **Deletion in BST**

/\* 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;

}

1. **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);

}

1. **Print all the keys in given range k1 to 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);

}

1. **Construct balanced BST from 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;

}

1. **Merge Two Balanced Binary Search Trees**

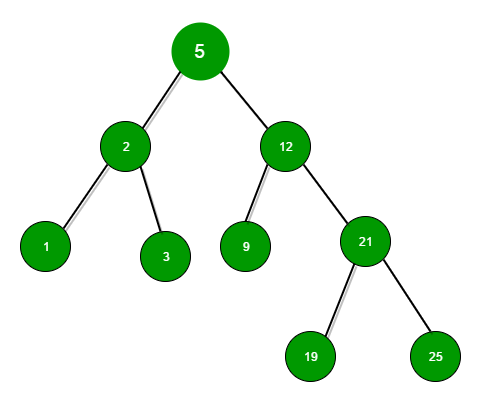
You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take O(m+n) time.

**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](https://www.geeksforgeeks.org/archives/17679)) 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. This step takes O(m+n) time.
5. **Largest number in BST which is less than or equal to N**



int findMaxforN(Node\* root, int N)

{

    /\* If leaf node reached and is greater than N\*/

    if (root->left == NULL && root->right == NULL && root->key > N)

        return -1;

    /\* If node's value is less than N and right value is NULL or greater than then return the node value\*/

    if ((root->key <= N && root->right == NULL) || (root->key <= N && root->right->key > N))

        return root->key;

    // if node value is greater than N search in the left subtree

    if (root->key >= N)

        return findMaxforN(root->left, N);

    // if node value is less than N search in the right subtree

    else

        return findMaxforN(root->right, N);

}