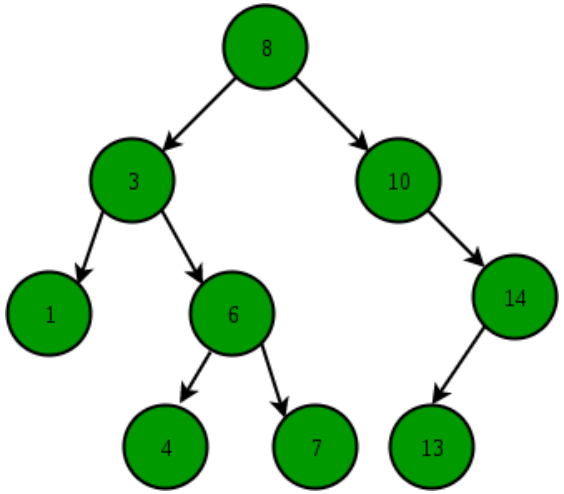
Data Structures and Algorithms

Trong-Hop Do

University of Information Technology, HCM city

Binary Search Tree



- 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.**

Binary search tree data 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
        = (struct node*)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

Searching a key

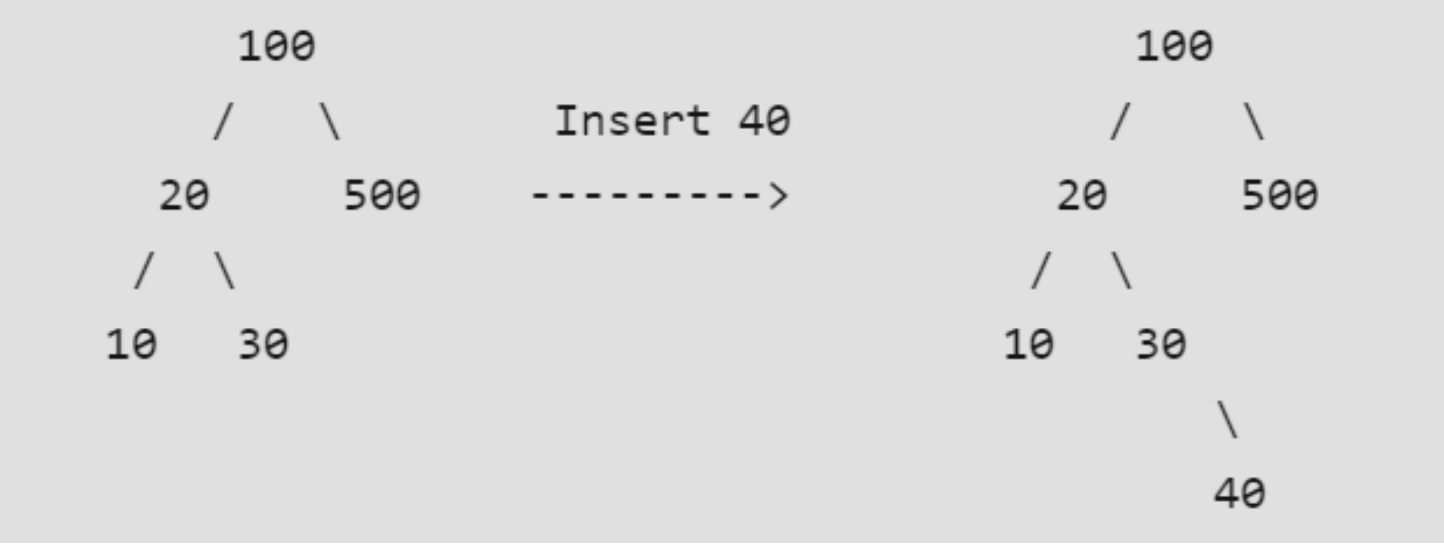
```
// C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // 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);
}
```

Searching a key (non recursive)

```
TNode * searchNode(TREE Root, Data x){
        Node *p = Root;
        while (p != NULL){
                if(x == p->Key) return p;
                else
                if(x < p->Key) p = p->pLeft;
                else p = p->pRight;
        }
        return NULL;
}
```

Insertion of a key

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



Insertion of a key

```
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;
}
```

Insertion of a key

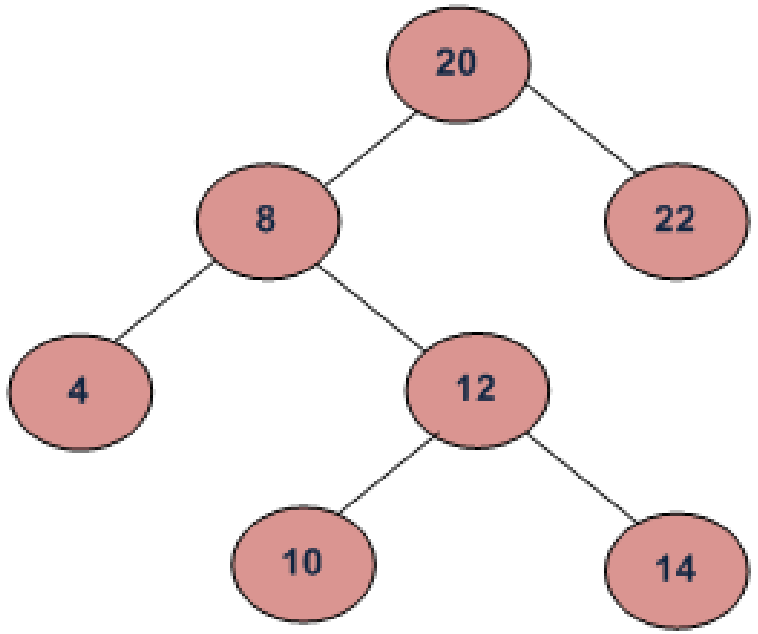
```
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);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

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

    return 0;
}
```

Inorder traversal of BST

```
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
```

Find the node with minimum value

```
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);
}
```



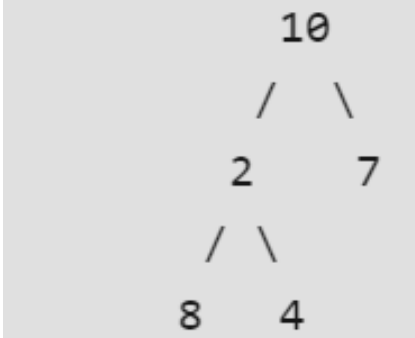
Find the node with maximum value

```
int maxValue(struct node* node)
{
    /* loop down to find the rightmost leaf */
    struct node* current = node;
    while (current->right != NULL)
        current = current->right;

    return (current->data);
}
```

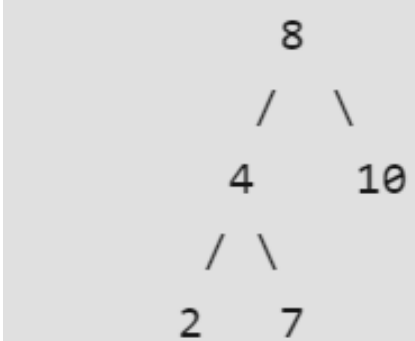
Binary Tree to Binary Search Tree Conversion

Example 1

Input:

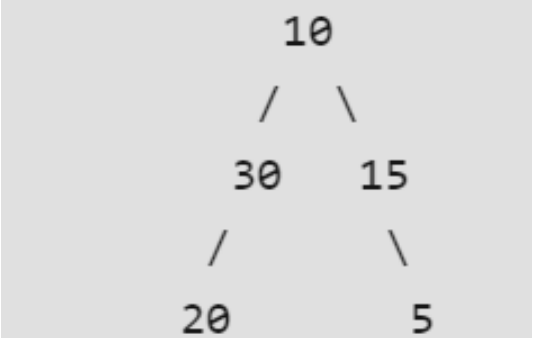


Output:

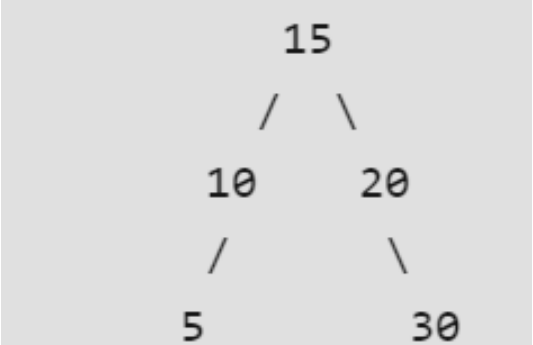


Example 2

Input:



Output:



Binary Tree to Binary Search Tree Conversion

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.

Binary Tree to Binary Search Tree Conversion

```
void binaryTreeToBST(struct node* root) {
    if (root == NULL)
        return;
    int n = countNodes(root);
    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;
}
```

Binary Tree to Binary Search Tree Conversion

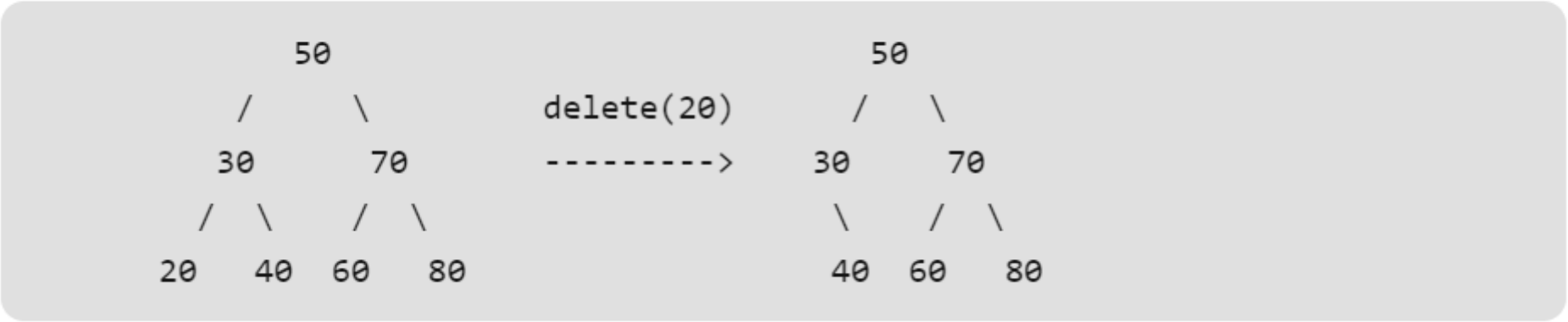
```
void storeInorder(struct node* node, int inorder[], int* index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* 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);
}
```

Binary Tree to Binary Search Tree Conversion

```
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);
}
```

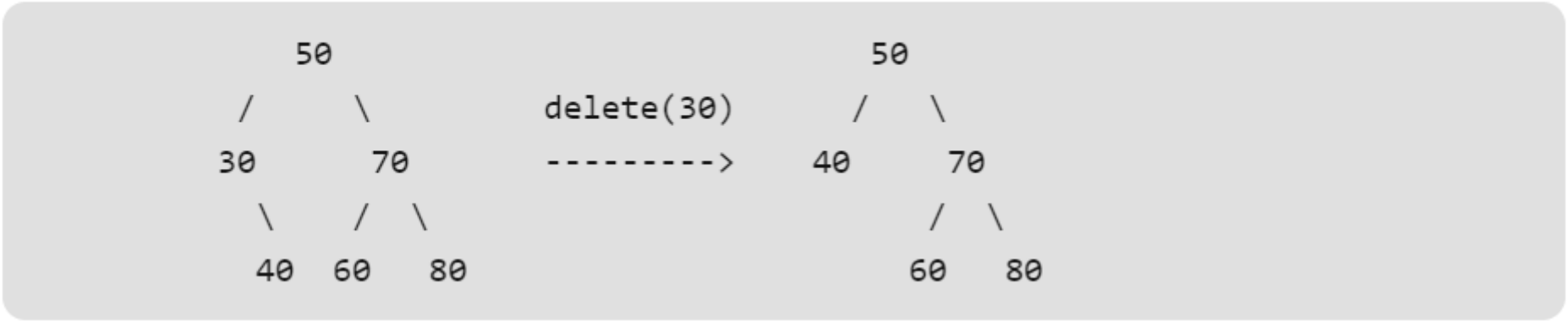
Delete a node in BST

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



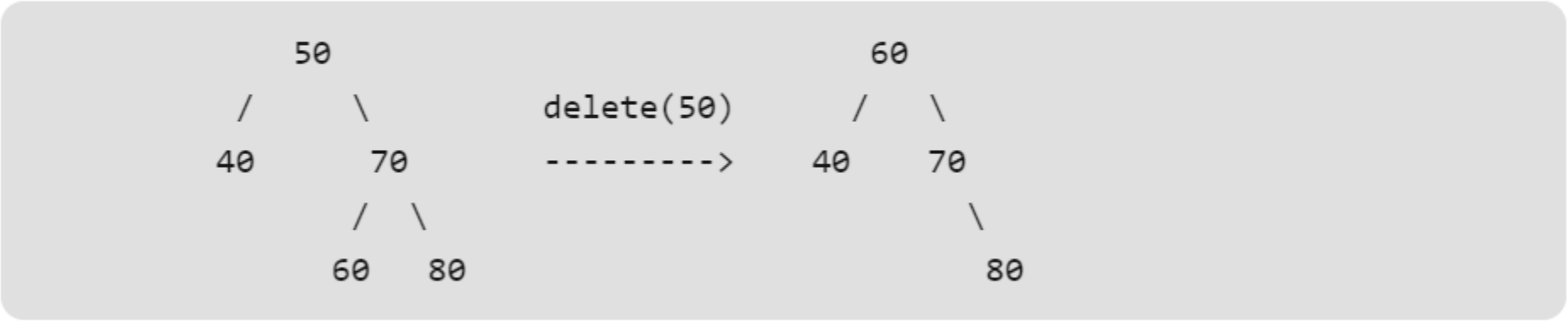
Delete a node in BST

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



Delete a node in BST

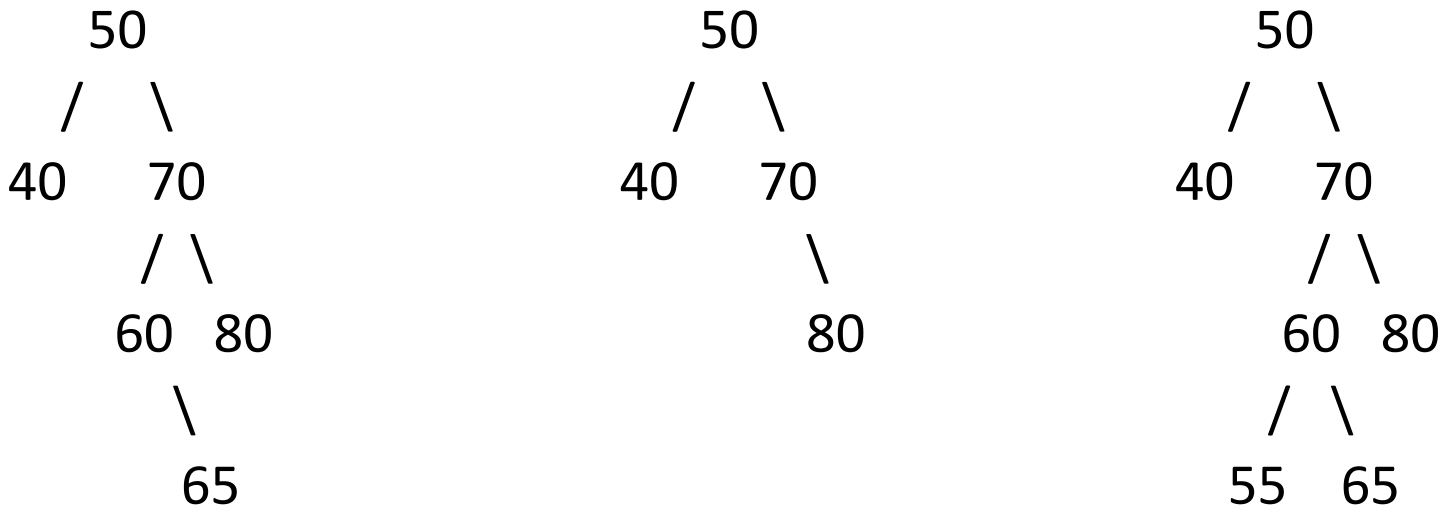
3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



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.

Delete a node in BST

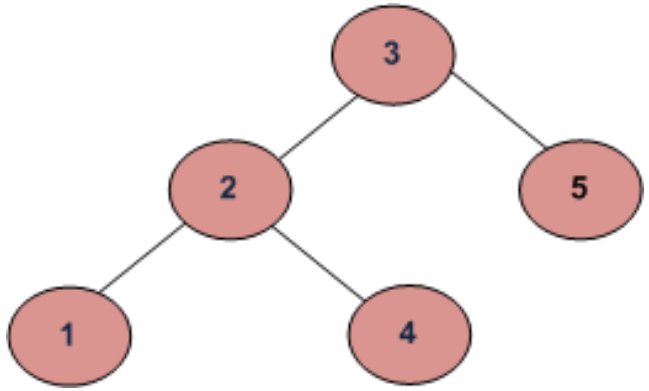
3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



```
Node* deleteNode(Node* root, int k)
{
    // Base case
    if (root == NULL)
        return root;

    // Recursive calls for ancestors of node to be deleted
    if (root->key > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->key < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    // We reach here when root is the node to be deleted.

    // If one of the children is empty
    if (root->left == NULL) {
        Node* temp = root->right;
        //CONNECT THE ROOT'S PARENT TO TEMP NODE
        //delete root;
        return temp;
    }
    else if (root->right == NULL) {
        Node* temp = root->left;
        //CONNECT THE ROOT'S PARENT TO TEMP NODE
        //delete root;
        return temp;
    }

    // If both children exist
    else {

        Node* succParent = root;

        // Find successor
        Node* succ = root->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }


        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;

        // Copy Successor Data to root
        root->key = succ->key;

        // Delete Successor and return root
        delete succ;
        return root;
    }
}
```

Check if a binary tree is BST or not

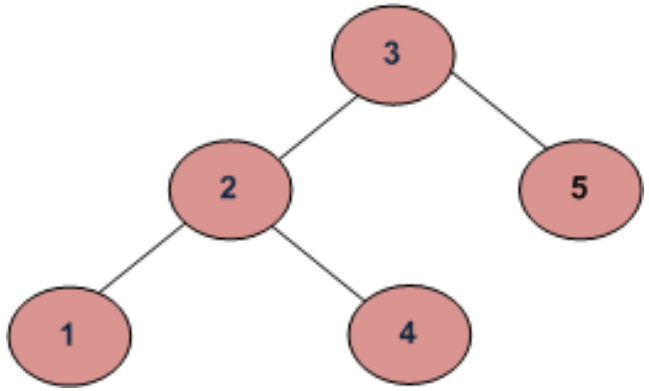
- Method 1 (simple but wrong)




```
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;
}
```

Check if a binary tree is BST or not

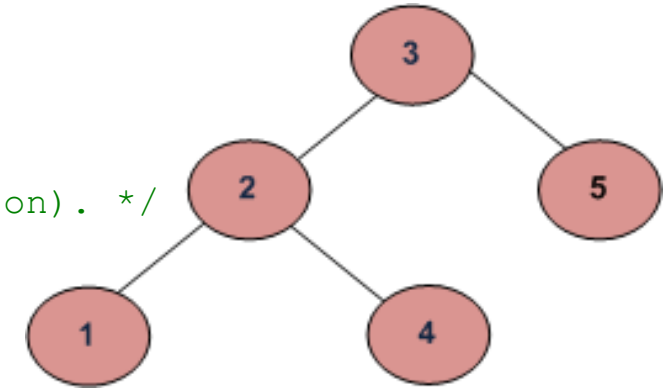
- Method 2 (correct but not efficient)




```
int isBST(struct node* node){
  if (node == NULL)
    return 1;
  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return 0;
  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < 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;
}
```

Check if a binary tree is BST or not

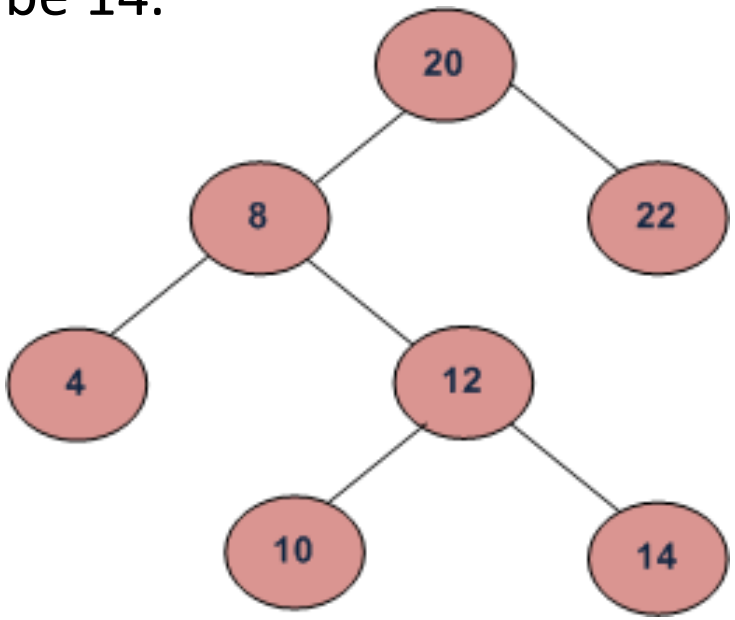
- Method 3 (correct and efficient)




```
/* 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
}
```

Find k-th smallest element in BST

- Given the root of a binary search tree and K as input, find K-th smallest element in BST.
- For example, in the following BST, if k = 3, then the output should be 10, and if k = 5, then the output should be 14.



Find k-th smallest element in BST

- Method 1: Using Inorder Traversal (O(n) time and O(h) auxiliary space)

```
Node* kthSmallest(Node* root, int& k){
    // base case
    if (root == NULL)
        return NULL;
    // search in left subtree
    Node* left = kthSmallest(root->left, k);
    // if k'th smallest is found in left subtree, return it
    if (left != NULL)
        return left;
    // if current element is k'th smallest, return it
    k--;
    if (k == 0)
        return root;
    // else search in right subtree
    return kthSmallest(root->right, k);
}
```

Find k-th smallest element in BST

- Method 2: Augmented Tree Data Structure (O(h) Time Complexity and O(h) auxiliary space)
- The idea is to maintain the rank of each node. We can keep track of elements in the left subtree of every node while building the tree. Since we need the K-th smallest element, we can maintain the number of elements of the left subtree in every node.
- Assume that the root is having ‘lCount’ nodes in its left subtree. If K = lCount + 1, root is K-th node. If K < lCount + 1, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > lCount + 1, we continue our search in the right subtree for the (K – lCount – 1)-th smallest element. Note that we need the count of elements in the left subtree only.

Find k-th smallest element in BST

- Method 2: Augmented Tree Data Structure (O(h) Time Complexity and O(h) auxiliary space)

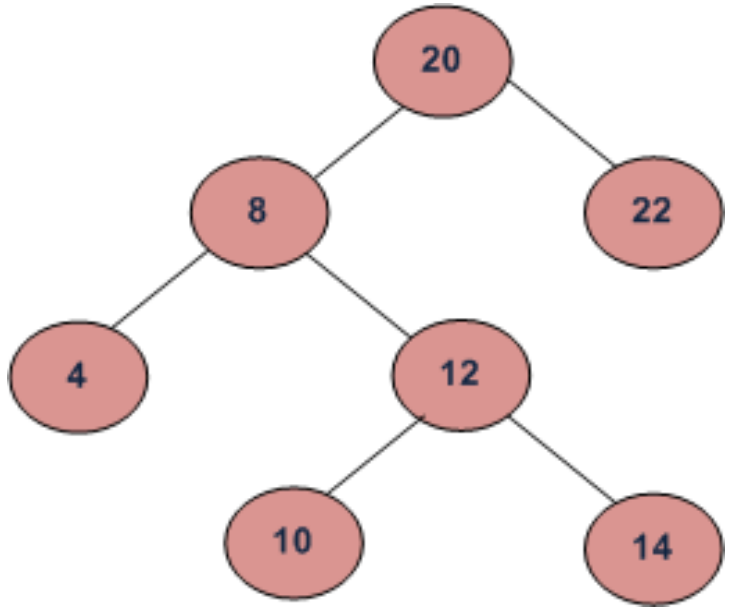
```
// Function to find k'th largest element in BST
// Here count denotes the number of nodes processed so far
Node* kthSmallest(Node* root, int k){
    // base case
    if (root == NULL)
        return NULL;
    int count = root->lCount + 1;
    if (count == k)
        return root;
    if (count > k)
        return kthSmallest(root->left, k);
    // else search in right subtree
    return kthSmallest(root->right, k - count);
}
```

Lowest Common Ancestor in a Binary Search Tree.

- Input: LCA of 10 and 14
- Output: 12
- Explanation: 12 is the closest node to both 10 and 14 which is a ancestor of both the nodes.

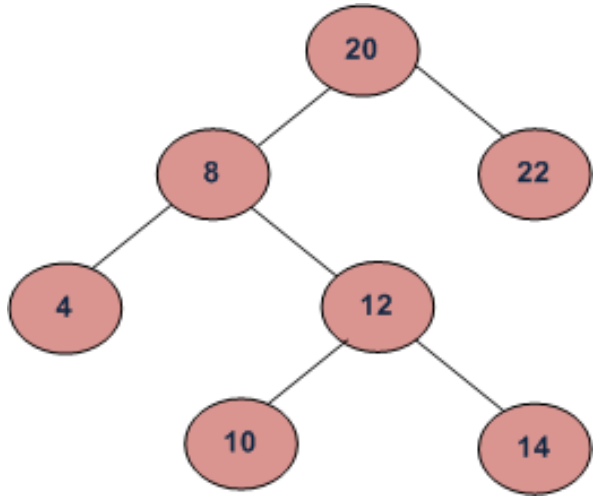
- Input: LCA of 8 and 14
- Output: 8
- Explanation: 8 is the closest node to both 8 and 14 which is a ancestor of both the nodes.

- Input: LCA of 10 and 22
- Output: 20
- Explanation: 20 is the closest node to both 10 and 22 which is a ancestor of both the nodes.



Lowest Common Ancestor in a Binary Search Tree.

- Algorithm:
- Create a recursive function that takes a node and the two values n1 and n2.
- If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for thr right subtree.
- If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for thr left subtree.
- If both the above cases are false then return the current node as LCA.



Lowest Common Ancestor in a Binary Search Tree.

```
/* 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;
}
```