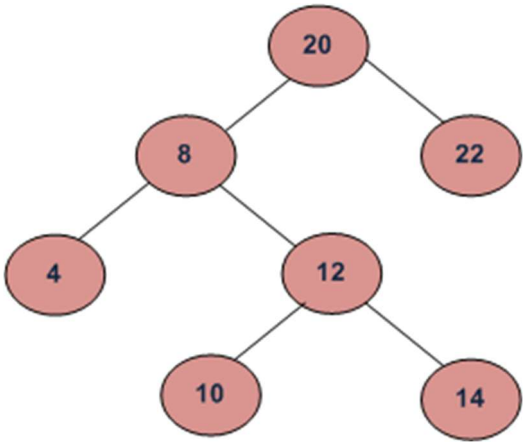
Lowest Common Ancestor in a Binary Search Tree

Given two values n1 and n2 in a Binary Search Tree, find
the **L**owest **C**ommon **A**ncestor (LCA). You may assume that both values exist in the tree.

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 [i.e., closest to n1 and n2].

Examples:

Input 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.*

Lowest Common Ancestor in a Binary Search Tree using **Recursion**:

To solve the problem follow the below idea:

For Binary search tree, while traversing the tree from top to bottom the first node which lies in between the two numbers n1 and n2 is the LCA of the nodes, i.e. the first node n with the lowest depth which lies in between n1 and n2 (n1<=n<=n2) n1 < n2.

So just recursively traverse the BST , if node's value is greater than both n1 and n2 then our LCA lies in the left side of the node, if it is smaller than both n1 and n2, then LCA lies on the right side. Otherwise, the root is LCA (assuming that both n1 and n2 are present in BST)

Follow the given steps to solve the problem:

- 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 the 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 the left subtree.
- If both the above cases are false then return the current node as LCA.

Below is the implementation of the above approach.

C++

```
// A recursive CPP program to find
// LCA of two nodes n1 and n2.
#include <bits/stdc++.h>
using namespace std;

class node {
public:
    int data;
    node *left, *right;
};

/* Function to find LCA of n1 and n2.
The function assumes that both
n1 and n2 are present in BST */
node* lca(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.*/
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = Node->right = NULL;
    return (Node);
}
```

```
/* Driver code*/
int main()
{
    // Let us construct the BST
    // shown in the above figure
    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);

      // Function calls
    int n1 = 10, n2 = 14;
    node* t = lca(root, n1, n2);
    cout << "LCA of " << n1 << " and " << n2 << " is "
         << t->data << endl;

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    cout << "LCA of " << n1 << " and " << n2 << " is "
         << t->data << endl;

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    cout << "LCA of " << n1 << " and " << n2 << " is "
```

```
        << t->data << endl;
    return 0;
}
```

Output

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time Complexity: O(H). where H is the height of the tree.
Auxiliary Space: O(H), If recursive stack space is ignored, the space complexity of the above solution is constant.

Below is the iterative implementation of the above approach:
C++

```
// A recursive CPP program to find
// LCA of two nodes n1 and n2.
#include <bits/stdc++.h>
using namespace std;

class node {
public:
    int data;
    node *left, *right;
};

/* Function to find LCA of n1 and n2.
The function assumes that both n1 and n2
are present in BST */
node* lca(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;
}

/* Helper function that allocates
a new node with the given data.*/
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = Node->right = NULL;
    return (Node);
}
```

```
/* Driver code*/
int main()
{
    // Let us construct the BST
    // shown in the above figure
    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);

      // Function calls
    int n1 = 10, n2 = 14;
    node* t = lca(root, n1, n2);
    cout << "LCA of " << n1 << " and " << n2 << " is "
         << t->data << endl;

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    cout << "LCA of " << n1 << " and " << n2 << " is "
         << t->data << endl;

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
```

```
    cout << "LCA of " << n1 << " and " << n2 << " is "
         << t->data << endl;
    return 0;
}
```

Output

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time Complexity: O(H). where H is the height of the tree
Auxiliary Space: O(1). The space complexity of the above solution is constant.

Lowest Common Ancestor in a Binary Search Tree using Morris traversal:

Follow the steps to implement the above approach:

1. Initialize a pointer curr to the root of the tree.
2. While curr is not NULL, do the following:
3. If curr has no left child, check if curr is either of the two nodes we are interested in. If it is, return curr. Otherwise, move curr to its right child.
4. If curr has a left child, find the inorder predecessor pre of curr by moving to the rightmost node in the left subtree of curr.
5. If the right child of pre is NULL, set it to curr and move curr to its left child.
6. If the right child of pre is curr, set it to NULL and restore the original tree structure. Then check if curr is either of the two nodes we are interested in. If it is, return curr. Otherwise, move curr to its right child.
7. If the two nodes we are interested in are not found during the traversal, return NULL

Below is the implementation of the above approach:

C++

```
// C++ code to implement the morris traversal approach

#include<bits/stdc++.h>

using namespace std;


struct TreeNode {

    int val;

    TreeNode *left, *right;

    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};


TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

    TreeNode* curr = root;

    while (curr != NULL) {

        if (curr->left == NULL) { // if the left subtree is empty, move to the right subtree

            if (curr == p || curr == q) return curr;

            curr = curr->right;

        }

        else {

            TreeNode* pre = curr->left;

            while (pre->right != NULL && pre->right != curr) pre = pre->right; // find the inorde
predecessor of the current node

            if (pre->right == NULL) { // if the predecessor's right child is NULL, make it point
the current node

                pre->right = curr;

                curr = curr->left;

            }

            else { // if the predecessor's right child is the current node, restore the tree
structure and move to the right subtree
```

```
                pre->right = NULL;
                if (curr == p || curr == q) return curr;
                curr = curr->right;
            }
        }
    }
    return NULL;
}
// Driver Code
int main() {
    /*
    Input Tree:
              5
            /    \
          4       6
           \       \
            3       7
                     \
                      8
    */
    TreeNode *root = new TreeNode(5);
    root->left = new TreeNode(4);
    root->left->right = new TreeNode(3);
    root->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->right->right = new TreeNode(8);
```

```
    TreeNode *p = root->left;
    TreeNode *q = root->left->right;

    TreeNode *lca1 = lowestCommonAncestor(root, p, q);
    cout <<" LCA of "<<p->val<<" and "<<q->val<<" is "<< lca1->val << endl;

     TreeNode  *x = root->right->right;
     TreeNode *y = root->right->right->right;

    TreeNode *lca2 = lowestCommonAncestor(root, x, y);
    cout <<" LCA of "<<x->val<<" and "<<y->val<<" is "<< lca2->val << endl;

    return 0;
}
```

Output

```
LCA of 4 and 3 is 4
LCA of 7 and 8 is 7
```

Time Complexity: O(N) , The time complexity of the Morris Traversal approach to find the lowest common ancestor of two nodes in a binary search tree is O(N), where N is the number of nodes in the tree.
Auxiliary Space: O(1) , The space complexity of the Morris Traversal approach is O(1), which is constant extra space.