Construct a balanced BST from the given keys

Given an unsorted integer array that represents binary search tree (BST) keys, construct a height-balanced BST from it. For each node of a height-balanced tree, the difference between its left and right subtree height is at most 1.

For example,

```
Input: keys = [15, 10, 20, 8, 12, 16, 25]

Output:

        15
      /    \
     10     20
```

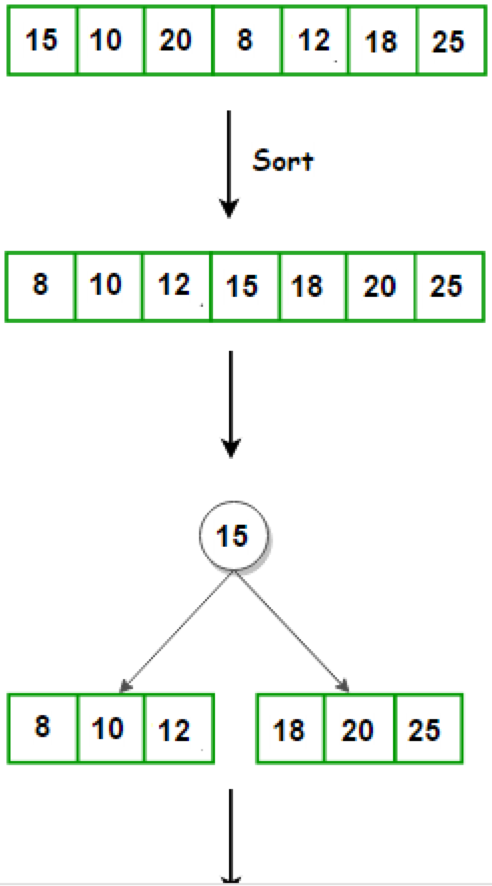
```
  8   12 16  25

OR

         12
       /    \
      10    20
    /     /  \
  8     16  25
       /
      15

OR

Any other possible representation.
```

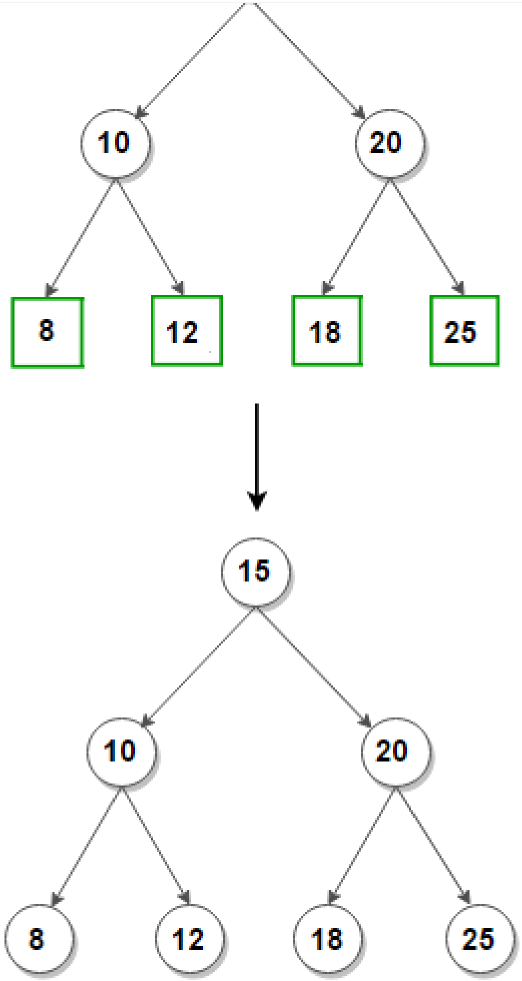
Practice this problem

We have already discussed how to insert a key into a BST. The height of such BST in the worst-case can be as much as the total number of keys in the BST. The worst case happens when given keys are sorted in ascending or descending order, and we get a skewed tree where all the nodes except the leaf have one and only one child.

better than the linear time required to find items by key in an (unsorted) array or unbalanced trees.

We can easily modify the solution to get height-balanced BSTs if all keys are known in advance. The idea is to sort the given keys first. Then the root will be the middle element of the sorted array, and we recursively construct the left subtree of the root by keys less than the middle element and the right subtree of the root by keys more than the middle element. For example,





Following is the C++, Java, and Python implementation of the idea:

C++ Java Python

```
#include <algorithm>
using namespace std;

// Data structure to store a BST node
struct Node
{
    int data;
    Node* left = nullptr, *right = nullptr;

    Node() {}
    Node(int data): data(data) {}
};

// Function to perform inorder traversal on the tree
void inorder(Node* root)
{
    if (root == nullptr) {
        return;
    }

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

// Recursive function to insert a key into a BST
Node* insert(Node* root, int key)
{
    // if the root is null, create a new node and return it
    if (root == nullptr) {
        return new Node(key);
    }

    // if the given key is less than the root node, recur for the left subtree
    if (key < root->data) {
        root->left = insert(root->left, key);
    }

    // if the given key is more than the root node, recur for the right subtree
```

```
    return root;
}

// Function to construct balanced BST from the given sorted array.
// Note that the root of the tree is passed by reference here
void convert(vector<int> const &keys, int low, int high, Node* &root)
{
    // base case
    if (low > high) {
        return;
    }

    // find the middle element of the current range
    int mid = (low + high) / 2;

    // construct a new node from the middle element and assign it to the root
    root = new Node(keys[mid]);

    // left subtree of the root will be formed by keys less than middle element
    convert(keys, low, mid - 1, root->left);

    // right subtree of the root will be formed by keys more than the middle element
    convert(keys, mid + 1, high, root->right);
}

// Function to construct balanced BST from the given unsorted array
Node* convert(vector<int> keys)
{
    // sort the keys first
    sort(keys.begin(), keys.end());

    // construct a balanced BST
    Node* root = nullptr;
    convert(keys, 0, keys.size() - 1, root);

    // return root node of the tree
    return root;
}
```

```
    // input keys
    vector<int> keys = { 15, 10, 20, 8, 12, 16, 25 };

    // construct a balanced binary search tree
    Node* root = convert(keys);

    // print the keys in an inorder fashion
    inorder(root);

    return 0;
}
```

Download Run Code

Output:

```
8 10 12 15 16 20 25
```

The time complexity of the above solution is `O(n.log(n))`, where `n` is the size of the BST, and requires space proportional to the tree's height for the call stack.

BST, Sorting

Amazon, Easy, Recursive