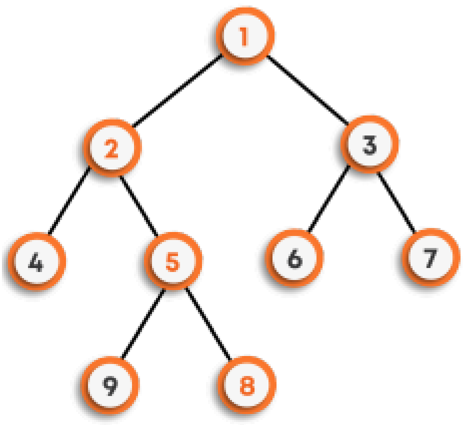
# BST-II

## Root to node path in a binary tree

**Problem statement**: Given a Binary tree, we have to return the path of the root node to the given node.

**For Example**: Refer to the image below.



Path from root to 8: [1 2 5 8]

**Approach:**

1. Start from the root node and compare it with the target value. If matched, then simply return, otherwise, recursively call over left and right subtrees.
2. Continue this process until the target node is found and return if found.
3. While returning, you need to store all the nodes that you have traversed on the path from the root to the target node in a vector.
4. Now, in the end, you will be having your solution vector.

Code:

```
vector<int>* getRootToNodePath(BinaryTreeNode<int>* root, int data) {
        if (root == NULL) {                      // Base Case:
                return NULL;
        }

        if (root->data == data) {           // Small calculation part: when node found
                vector<int>* output = new vector<int>();
                output->push_back(root->data); // inserted the node in solution vector
                return output;
        }
        // getting output vector out of left subtree
        vector<int>* leftOutput = getRootToNodePath(root->left, data);
        if (leftOutput != NULL) {
                leftOutput->push_back(root->data);
                return leftOutput;
        }
        // getting output vector out of right subtree
        vector<int>* rightOutput = getRootToNodePath(root->right, data);
        if (rightOutput != NULL) {
                        rightOutput->push_back(root->data);
                        return rightOutput;
        } else {
                return NULL;
        }
}
```

Now try to code the same problem with BST instead of a binary tree.

## Pair Sum in Bst

**Problem statement**: Given a Binary Search Tree (BST) and a target sum, find two elements in the BST that add up to the given target sum. You need to return both elements as an array in ascending order.

**Approach:**

1. Perform an in-order traversal of the BST, which will give you elements in sorted order.
2. Initialize two pointers, one at the beginning (smallest element) and one at the end (largest element) of the sorted array.

3. While the left pointer is less than the right pointer:

a. Calculate the sum of elements at the left and right pointers.
b. If the sum is equal to the target, return these two elements.
c. If the sum is less than the target, increment the left pointer.
d. If the sum is greater than the target, decrement the right pointer.

If no such pair is found, return an empty array.

C++ Code:

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

vector<int> findTarget(TreeNode* root, int k) {
    vector<int> result;
    if (!root) return result;

    stack<TreeNode*> leftStack, rightStack;
    TreeNode* left = root;
    TreeNode* right = root;

    while (left) {
        leftStack.push(left);
        left = left->left;
    }
    while (right) {
        rightStack.push(right);
        right = right->right;
    }
    while (!leftStack.empty() && !rightStack.empty() && leftStack.top()
!= rightStack.top()) {
        int sum = leftStack.top()->val + rightStack.top()->val;
        if (sum == k) {
            result.push_back(leftStack.top()->val);
            result.push_back(rightStack.top()->val);
            return result;
        }
        else if (sum < k) {
            left = leftStack.top()->right;
            leftStack.pop();
            while (left) {
                leftStack.push(left);
                left = left->left;
            }
        }
        else {
            right = rightStack.top()->left;
            rightStack.pop();
            while (right) {
                rightStack.push(right);
                right = right->right;
            }
        }
    }

    return result;
}
```

## Types of BST:

The two commonly used types of BSTs we are going to cover are

- AVL Trees (also known as self-balancing BST, uses rotation to balance)
- Red-Black Trees

First of all let’s look at what are balanced BSTs

**For a balanced BST:**

```
|Height_of_left_subtree - Height_of_right_subtree| <= 1
```

This equation must be valid for each and every node present in the BST. By mathematical calculations, it was found that the height of a Balanced BST is log(n), where n is the number of nodes in the tree. This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in O(logn).

## AVL Trees:

An AVL tree, named after its inventors Adelson-Velsky and Landis, is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of every node differ by at most one. This property ensures that the tree remains approximately balanced, preventing it from becoming skewed and, as a result, maintaining efficient search, insertion, and deletion operations.

**Here are some key points about AVL trees:**

**Balanced Structure**: AVL trees are self-balancing, meaning that after each insertion or deletion operation, the tree is adjusted to maintain its balance. The balance factor (the height difference between the left and right subtrees) of every node in an AVL tree is restricted to be at most 1.
**Efficient Operations**: Due to their balanced nature, AVL trees provide efficient operations like searching, insertion, and deletion, all with an average time complexity of O(log n), where "n" is the number of nodes in the tree.

**Where to Use:**

- **Fast Lookup**: AVL trees are ideal for scenarios where you need fast lookup operations. For example, they are used in database indexing, symbol tables in compilers, and in various data structures and algorithms.
- **Range Queries:** They are particularly useful in situations where you need to perform range queries efficiently, such as finding all elements within a specified range.
- **Ordered Data**: When you need to maintain data in sorted order and have frequent insertions and deletions, AVL trees are a good choice.

**Drawbacks**: While AVL trees provide efficient, balanced operations, they come with a slight overhead in terms of balancing the tree after each insertion or deletion, which can make these operations slightly slower than in simpler data structures like binary search trees. For scenarios with mostly static data or infrequent updates, simpler data structures might be more efficient.

## Red-Black Trees:

Red-Black Trees are another type of self-balancing binary search tree. They were developed by Rudolf Bayer in 1972 and further refined by his student, Edward McCreight, in 1978. Red-Black Trees have similar use cases as AVL trees, but they use a different balancing scheme, which involves coloring the nodes to ensure the tree's balance.

Here are some key points about Red-Black Trees:

**Balanced Structure**: Red-Black Trees are balanced binary search trees where each node is colored either red or black, and they must adhere to a set of rules to maintain balance. The primary rule is that no path from the root to any leaf node should have more than twice the height of any other path.

**Efficient Operations**: Red-Black Trees provide efficient search, insertion, and deletion operations with an average time complexity of O(log n), where "n" is the number of nodes in the tree. This makes them suitable for applications that require balanced trees, but they tend to be slightly faster than AVL trees due to fewer balancing restrictions.

**Where to Use:**

- **Dynamic Ordered Data**: Red-black trees are commonly used in scenarios where you need a self-balancing data structure for maintaining ordered data. They are preferred over AVL trees when you want to optimize for insertions and deletions.
- **Map and Set Implementations**: Red-Black Trees are used as the underlying data structure for many programming language implementations of map and set data structures, such as C++'s std::map and std::set.

**Drawbacks**: While Red-Black Trees are more permissive than AVL trees when it comes to balancing, they still require additional logic to maintain the color properties, which may lead to slightly slower operations than simple binary search trees for some use cases.