

Instructions:

- This exam consists of two programming questions.
- Read each question carefully before starting to code.
- Write clean, well-commented code with proper error handling.
- Test your implementation with the provided examples.
- You may use standard C++ library containers (vector, stack, queue, etc.).
- Manage your time wisely – each question is designed to take approximately 35–40 minutes.

## Question 1: Min-Stack Implementation

[50 points]

Design a stack data structure that supports standard stack operations (**push**, **pop**, **top**) as well as retrieving the minimum element in **constant time**. Implement a **MinStack** class with all operations running in  $O(1)$  time complexity.

Implement the following member functions:

<b>void push(int x)</b>	Push element <b>x</b> onto the stack
<b>void pop()</b>	Remove the top element from the stack
<b>int top()</b>	Return the top element without removing it
<b>int getMin()</b>	Return the minimum element in the stack in $O(1)$ time
<b>bool empty()</b>	Return <b>true</b> if the stack is empty
<b>int size()</b>	Return the number of elements in the stack

All operations must run in  $O(1)$  time complexity. The space complexity should be  $O(n)$  where  $n$  is the number of elements.

Your implementation must handle all edge cases (**pop** from empty stack, **getMin** on empty stack, etc.) You may use **std::stack** or implement your own using vectors

### Hint

Consider maintaining auxiliary information alongside each element to track the minimum value at each level of the stack. One approach is to store pairs of (value, current\_min) or maintain a separate stack that tracks minimums.

### Grading Criteria

Correct implementation of all operations	<b>30 points</b>
$O(1)$ time complexity for all operations	<b>12 points</b>
Proper edge case handling	<b>5 points</b>
Code quality and documentation	<b>3 points</b>

## Sample Usage

```
#include "MinStack.h"
#include <iostream>

int main() {
    MinStack ms;

    ms.push(5);
    ms.push(2);
    ms.push(7);
    ms.push(1);

    std::cout << "Min: " << ms.getMin() << std::endl; // Output: 1
    std::cout << "Top: " << ms.top() << std::endl; // Output: 1

    ms.pop(); // Remove 1

    std::cout << "Min: " << ms.getMin() << std::endl; // Output: 2
    std::cout << "Top: " << ms.top() << std::endl; // Output: 7

    ms.pop(); // Remove 7
    ms.pop(); // Remove 2

    std::cout << "Min: " << ms.getMin() << std::endl; // Output: 5
    std::cout << "Size: " << ms.size() << std::endl; // Output: 1

    return 0;
}
```

## Example Trace

Operation	Stack Contents	getMin()	Explanation
push(5)	[5]	5	5 is the only element
push(2)	[5, 2]	2	2 is now the minimum
push(7)	[5, 2, 7]	2	2 remains the minimum
push(1)	[5, 2, 7, 1]	1	1 is now the minimum
pop()	[5, 2, 7]	2	After removing 1, min is 2
pop()	[5, 2]	2	Min is still 2

## Question 2: BST Validation and Path Sum

[50 points]

Implement two functions for Binary Search Trees: (i) validate if a binary tree is a valid BST, and (ii) check if there exists a root-to-leaf path with a given sum.

The binary tree is defined using the following structure:

```
struct Node {
    int key;
    Node *left, *right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};
```

**Part A: BST Validation [25 points]**

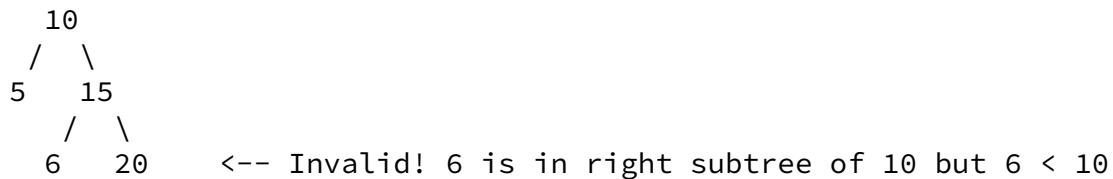
Implement the following function:

```
bool isValidBST(Node* root);
```

A valid BST must satisfy:

- All nodes in the left subtree have keys **strictly less than** the node's key
- All nodes in the right subtree have keys **strictly greater than** the node's key
- Both left and right subtrees are also valid BSTs

**Important:** The following tree is **NOT** a valid BST even though each node's immediate children satisfy the BST property:



**Hint:** Pass valid range constraints (min, max) down the recursion to ensure all descendants satisfy the BST property relative to their ancestors.

**Test Cases for Part A**

```
// Test Case 1: Valid BST
//      5
//      / \
//     3   7
//    / \
//   2   4
Node* tree1 = new Node(5);
tree1->left = new Node(3);
tree1->right = new Node(7);
tree1->left->left = new Node(2);
tree1->left->right = new Node(4);
// isValidBST(tree1) should return true

// Test Case 2: Invalid BST
//      5
//      / \
//     3   7
//    / \
//   2   6  <-- Invalid! 6 > 5 but in left subtree
Node* tree2 = new Node(5);
tree2->left = new Node(3);
tree2->right = new Node(7);
tree2->left->left = new Node(2);
tree2->left->right = new Node(6);
// isValidBST(tree2) should return false
```

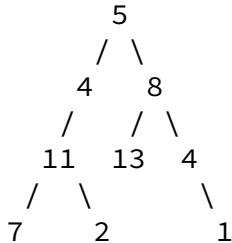
## Part B: Root-to-Leaf Path Sum [25 points]

Implement the following function:

```
bool hasPathSum(Node* root, int targetSum);
```

Given a binary tree and a target sum, return **true** if the tree has a **root-to-leaf** path such that adding up all the values along the path equals the target sum. A leaf is a node with no children.

### Example for Part B



- `hasPathSum(root, 22)` returns **true** (path: 5 → 4 → 11 → 2)
- `hasPathSum(root, 26)` returns **true** (path: 5 → 8 → 13)
- `hasPathSum(root, 18)` returns **true** (path: 5 → 8 → 4 → 1)
- `hasPathSum(root, 10)` returns **false** (no such path exists)

**Important:** The path must end at a leaf node. The path 5 → 4 (sum = 9) does not count because node 4 is not a leaf.

### Test Cases for Part B

```
// Using the tree from the example above
bool result1 = hasPathSum(root, 22); // Expected: true
bool result2 = hasPathSum(root, 26); // Expected: true
bool result3 = hasPathSum(root, 10); // Expected: false
bool result4 = hasPathSum(nullptr, 0); // Expected: false (empty tree)

// Single node tree
Node* single = new Node(5);
bool result5 = hasPathSum(single, 5); // Expected: true
bool result6 = hasPathSum(single, 10); // Expected: false
```

### Requirements and Grading

Both functions should use recursion. Your implementation must handle edge cases (empty tree, single node, negative values, etc). For both parts, the time complexity should be  $O(n)$ , where  $n$  is the number of vertices.

<code>isValidBST</code> and <code>hasPathSum</code> correct implementation	<b>22+22 points</b>
Edge case handling for <code>isValidBST</code>	<b>4 points</b>
Proper leaf node checking in <code>hasPathSum</code>	<b>4 points</b>
Code quality and documentation	<b>3+3 points</b>

**End of Exam**  
*Good Luck!*