

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: Queue Using Two Stacks

[50 points]

Implement a `QueueUsingStacks` class that simulates a FIFO (first-in, first-out) queue using **exactly two stacks**. Your implementation should support standard queue operations while maintaining efficient performance.

Required Operations

Implement the following member functions:

<code>void enqueue(int x)</code>	Insert element <code>x</code> at the rear of the queue
<code>int dequeue()</code>	Remove and return the front element of the queue
<code>int front()</code>	Return (but do not remove) the front element
<code>bool empty()</code>	Return <code>true</code> if the queue is empty, <code>false</code> otherwise
<code>int size()</code>	Return the number of elements currently in the queue

Requirements

1. Use **only** stack operations: `push()`, `pop()`, `top()`, and `empty()`
2. Aim for **amortized** $O(1)$ time complexity for both `enqueue` and `dequeue` operations
3. Handle edge cases appropriately (e.g., dequeue from empty queue should throw an exception or return a sentinel value)
4. You may use `std::stack` from the C++ Standard Library

Hint

Use one stack for enqueueing operations and another stack for dequeuing operations. Transfer elements between stacks only when necessary to maintain the FIFO order.

Grading Criteria

Correct implementation of all operations	30 points
Efficient amortized time complexity	10 points
Proper edge case handling	5 points
Code quality and documentation	5 points

Sample Usage

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

int main() {
    QueueUsingStacks q;

    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    std::cout << "Front: " << q.front() << std::endl; // Output: 1
    std::cout << "Dequeue: " << q.dequeue() << std::endl; // Output: 1

    q.enqueue(4);
    q.enqueue(5);

    std::cout << "Size: " << q.size() << std::endl; // Output: 4
    std::cout << "Dequeue: " << q.dequeue() << std::endl; // Output: 2
    std::cout << "Dequeue: " << q.dequeue() << std::endl; // Output: 3
    std::cout << "Front: " << q.front() << std::endl; // Output: 4

    return 0;
}
```

Question 2: Binary Search Tree Range Sum

[50 points]

Given a Binary Search Tree (BST) with integer keys, implement a function that computes the sum of all keys that fall within a given range `[low, high]` (inclusive). Your solution should take advantage of the BST property to avoid visiting unnecessary nodes.

Node Structure

The BST is defined using the following structure:

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

Function Signature

Implement the following function:

```
int rangeSumBST(Node* root, int low, int high);
```

Requirements

1. Your solution must be **more efficient than $O(n)$** by pruning subtrees that cannot contain values in the specified range

2. Use the BST property: all keys in the left subtree are less than the node's key, and all keys in the right subtree are greater
3. Handle edge cases: empty tree, range outside tree values, invalid range (`low > high`), etc.
4. You may use either recursion or iteration

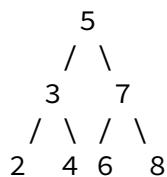
After implementing your solution, answer the following: *What is the time complexity of your solution in terms of:*

- k = number of keys within the range $[low, high]$
- h = height of the tree

Write your answer as a comment in your code.

Example

Consider a BST containing the keys: {5, 3, 7, 2, 4, 6, 8}



For `rangeSumBST(root, 4, 7)`, the function should return **22** because $4 + 5 + 6 + 7 = 22$.

Sample Test Cases

```

// Test Case 1: Normal range
int sum1 = rangeSumBST(root, 4, 7); // Expected: 22

// Test Case 2: Range includes all nodes
int sum2 = rangeSumBST(root, 1, 10); // Expected: 35 (sum of all nodes)

// Test Case 3: Range includes single node
int sum3 = rangeSumBST(root, 5, 5); // Expected: 5

// Test Case 4: Range outside tree values
int sum4 = rangeSumBST(root, 20, 30); // Expected: 0

// Test Case 5: Empty tree
int sum5 = rangeSumBST(nullptr, 1, 10); // Expected: 0
  
```

Grading Criteria

Correct implementation with proper BST traversal	25 points
Efficient pruning strategy (better than $O(n)$)	15 points
Proper edge case handling	5 points
Time complexity analysis	5 points

End of Exam
Good Luck!