**CS304 Assignment 4: Implementing TreeSort with a Custom BST**
**Due: Tuesday March 25th at 11:59 PM (Submit to moodle)**

---

**Objective**
In this assignment, you will:
1. **Implement a Binary Search Tree (BST) from scratch**, including the **Big Five** (Rule of Five).
2. **Implement the TreeSort algorithm** using a **in-order traversal** to store sorted elements in a vector.
3. **Compare the performance of TreeSort with C++'s std::sort** on large datasets.

---

**Part 1: Understanding TreeSort**
**How TreeSort Works**
TreeSort is a **comparison-based sorting algorithm** that leverages a **Binary Search Tree (BST)**:
1. **Insert all elements into a BST** (logically sorting them).
2. **Traverse the BST in-order** to extract elements in sorted order.
3. **Store the sorted elements in a vector**.

**Time Complexity**

| Operation | Average Case | Worst Case (Unbalanced Tree) |
|---|---|---|
| Insertion | `O(log n)` | `O(n)` |
| Traversal (Pre-order) | `O(n)` | `O(n)` |
| Overall Complexity | `O(n log n)` | `O(n²)` (degenerated tree) |

🔹 **Note:** To improve performance, you could use a **Self-Balancing BST (like AVL or Red-Black Tree)**, but for this assignment, implement a **standard unbalanced BST**.

---

**Part 2: Implementation Requirements**
**1. Implement a Binary Search Tree (BST) from Scratch**
✅ **You must define a BST class with the following:**
- **Node Structure (TreeNode)**
- **Insertion Method (insert)**
- **Pre-order Traversal (preOrderTraversal)**
- **Destructor, Copy Constructor, Copy Assignment, Move Constructor, Move Assignment (Big Five)**

**BST Class Skeleton:**
```
template<typename T>
class BST {
private:
    struct TreeNode {
        T data;
        TreeNode* left;
        TreeNode* right;
        TreeNode(const T& value) : data(value), left(nullptr), right(nullptr) {}
    };

    TreeNode* root;

    void insertHelper(TreeNode*& node, const T& value);
    void preOrderTraversalHelper(TreeNode* node, std::vector<T>& sortedVector);
    void destroyTree(TreeNode* node);
    TreeNode* copyTree(TreeNode* other);
```

```
public:
    BST();  // Constructor
    ~BST(); // Destructor
    BST(const BST& other); // Copy Constructor
    BST& operator=(const BST& other); // Copy Assignment
    BST(BST&& other) noexcept; // Move Constructor
    BST& operator=(BST&& other) noexcept; // Move Assignment

    void insert(const T& value);
    std::vector<T> preOrderTraversal();
};
```

---

## 2. Implement TreeSort using the BST

✅ **Steps to Implement TreeSort:**

1. **Insert all elements** from an unsorted vector into the BST.
2. **Extract elements using pre-order traversal** and store them in another vector.
3. **Return the sorted vector**.

**TreeSort Function:**

```
template<typename T>
std::vector<T> treeSort(std::vector<T>& arr) {
    BST<T> tree;
    for (const T& val : arr) {
        tree.insert(val);
    }
    return tree.preOrderTraversal();
}
```

---

## 3. Compare Performance with std::sort

✅ **Use the C++ <algorithm> library to compare execution time**

- **Generate large random datasets (10,000+ elements)**
- **Measure sorting time for both methods** using <chrono>

**Example Benchmarking Code:**

```
int main() {
    const int SIZE = 10000;
    std::vector<int> data(SIZE);

    // Fill with random values
    for (int& val : data) {
        val = rand() % 10000;
    }

    // Measure TreeSort time
    auto treeData = data;
    auto start = std::chrono::high_resolution_clock::now();
    std::vector<int> treeSorted = treeSort(treeData);
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "TreeSort Time: "
              << std::chrono::duration<double, std::milli>(end - start).count()
              << " ms\n";

    // Measure std::sort time
    auto stdData = data;
    start = std::chrono::high_resolution_clock::now();
    std::sort(stdData.begin(), stdData.end());
    end = std::chrono::high_resolution_clock::now();
    std::cout << "std::sort Time: "
              << std::chrono::duration<double, std::milli>(end - start).count()
```

```
            << " ms\n";

    return 0;
}
```

---

### Submission Requirements

- Submit a **C++ project (.cpp and .h files)** implementing:
  - **BST Class (with Big Five)**
  - **TreeSort Algorithm**
  - **Performance Comparison**
- Include a **README.md** with:
  - **Explanation of TreeSort and how it was implemented**
  - **Benchmark results comparing TreeSort vs. std::sort**
- Code should be **well-documented** and use **good object-oriented principles**.

---

### Grading Criteria

| Category | Points |
|---|---|
| BST correctly implemented with Big Five | 30 |
| TreeSort correctly implemented using BST | 30 |
| Pre-order traversal extracts sorted elements | 20 |
| Performance comparison with `std::sort` | 10 |
| Code readability, structure, and comments | 10 |
| **Total** | **100** |

---

### Bonus Challenge (+10 Points)

- Implement a **Self-Balancing BST (like AVL or Red-Black Tree)** and compare performance with the unbalanced BST for different inputs (sorted input, reverse-sorted input, randomized input).
- Analyze and **graph the performance difference** using different dataset sizes.

### Bonus Challenge 2 (+10 Points)

- The main bottleneck for treesort is the repeated use of 'new' to allocate heap memory whenever a new node is created. To avoid the repeated use of 'new' (which results in an expensive syscall), modify your tree to do the following:
  - If the list to be sorted is very small (<1000 elements), **use stack memory** to store the nodes.
  - If the list is larger, use a custom allocator that calls 'new' only once to allocate a **memory pool** of the exact size needed to store all the nodes in the tree, and access the memory pool (instead of calling 'new' repeatedly) as nodes are added to the tree.

---

### Expected Output Example

```
TreeSort Time: 35.2 ms
std::sort Time: 2.1 ms
```

👉 **std::sort should be significantly faster than TreeSort due to its optimized hybrid sorting approach.**

# Appendix (hints and additional information):

**In-order traversal (Left → Root → Right) prints the BST in sorted order.**
**Why?**

- In a **Binary Search Tree (BST)**, the **left subtree** contains values smaller than the root.
- The **right subtree** contains values larger than the root.
- **Visiting nodes in the order Left → Root → Right** means:
    - **First, visit the smallest elements (leftmost) first.**
    - **Then, visit the root (middle value).**
    - **Finally, visit the right subtree (larger values).**
- This results in a **sorted sequence**.

```
       8
      / \
     3   10
    / \    \
   1   6    14
      / \   /
     4   7 13
```

**In-order Traversal (Sorted Order)**
✅ **Output:**
1 3 4 6 7 8 10 13 14
**Pre-order Traversal (Root First)**
🚫 **Output (Not Sorted):**
8 3 1 6 4 7 10 14 13
🔹 **Conclusion:**
If you want to **print the BST in sorted order, use in-order traversal!**

**Hints for Implementing the <u>Big Five</u> in a Binary Search Tree (BST)**
When implementing a **Binary Search Tree (BST)** in C++, you need to carefully manage **dynamic memory** to avoid memory leaks and undefined behavior. This is where the **Big Five** come into play:

---

**1. Destructor (~BST()) – Freeing Allocated Memory**
**Hint:**

- Since your BST **allocates nodes dynamically (new)**, you must **delete all nodes** in the destructor.
- **Use a recursive helper function** to **delete the entire tree** from bottom to top.

**Guiding Question:**

- How do you safely delete an entire tree **starting from the leaves**?

🔹 **Think about a recursive post-order deletion method:**

```
void destroyTree(TreeNode* node) {
    if (node) {
        destroyTree(node->left);
        destroyTree(node->right);
        delete node;
    }
}
```

- Call this in the destructor.

---

**2. Copy Constructor (BST(const BST& other))**
**Hint:**

- If you create a new BST as a **copy of another BST**, you need to **deep copy** all nodes (not just copy pointers!).
- Use a **recursive function** to duplicate all nodes **from the original tree into the new tree**.

**Guiding Question:**

- How do you ensure that each node in the copied tree is a **newly allocated copy** of the corresponding node in the original tree?

◆ **Think about a recursive copy function:**

```
TreeNode* copyTree(TreeNode* otherNode) {
    if (!otherNode) return nullptr;
    TreeNode* newNode = new TreeNode(otherNode->data);
    newNode->left = copyTree(otherNode->left);
    newNode->right = copyTree(otherNode->right);
    return newNode;
}
```

- Call this in the copy constructor.

---

**3. Copy Assignment Operator (BST& operator=(const BST& other))**

**Hint:**

- The assignment operator needs to **copy data from an existing tree to an already existing tree**.
- Avoid **memory leaks** by first **deleting the old tree** before copying the new one.
- Handle **self-assignment** (if (this == &other) return *this;).

**Guiding Question:**

- What should you do first: **delete the current tree** or **copy the new tree**?

◆ **Think about three steps:**

1. **Check for self-assignment** (this != &other).
2. **Destroy the existing tree** (to prevent memory leaks).
3. **Copy the new tree recursively**.

---

**4. Move Constructor (BST(BST&& other) noexcept)**

**Hint:**

- When moving a BST, you want to **steal the pointer** to the other tree and **set the old tree to nullptr**.
- This avoids unnecessary deep copies and makes moving efficient.

**Guiding Question:**

- How do you **take ownership** of another tree without copying it?

◆ **Think about what happens to the root pointer:**

```
this->root = other.root;
other.root = nullptr;  // Leave the moved-from tree empty
```

- No need to delete anything in this case.

---

**5. Move Assignment Operator (BST& operator=(BST&& other) noexcept)**

**Hint:**

- The move assignment is similar to the move constructor but also needs to **clear any existing tree data first**.
- Just like in the copy assignment operator, **handle self-assignment**.

**Guiding Question:**

- What do you do before stealing the other tree's pointer?

◆ **Think about these steps:**

1. **Check for self-assignment (if (this == &other) return *this;).**
2. **Delete the existing tree**.
3. **Steal the pointer (this->root = other.root;)**.
4. **Set other.root to nullptr** to prevent double deletion.

---

**Final Thoughts**

- **Write the Destructor First** – It will help you with the other functions.
- **Write a Recursive copyTree() Helper –** It simplifies both the copy constructor and copy assignment.
- **Think About Ownership –** When moving, transfer the pointer instead of copying nodes.

Once you implement these correctly, your BST will **avoid memory leaks** and **follow proper C++ ownership semantics**. 🚀