**BST and Traversal using Stack and Queue**
**WRITEUP**

**1. COMPILATION AND EXECUTION**

**Compilation Command:** `g++ -std=c++11 -Wall -g driver.cpp -o bst_driver`
**Execution Command:** `./bst_driver`
**Memory Leak Verification:** `valgrind --leak-check=full ./bst_driver`
**File Structure:**

- bst.h - Class declarations and interface
- bst.cpp - Template implementations
- driver.cpp - Comprehensive test suite

**2. IMPLEMENTATION**
All five required methods have been successfully implemented and tested:

1. **insert()** - Recursive insertion with duplicate handling ✓
2. **retrieve()** - Recursive search with boolean return ✓
3. **pre_order_traversal()** - Iterative using single stack ✓
4. **in_order_traversal()** - Iterative using single stack ✓
5. **post_order_traversal()** - Iterative using single stack ✓

**Status: ALL METHODS WORKING CORRECTLY**

**3. TRAVERSAL METHODS USING STACKS**
**Pre-order Traversal (Root → Left → Right) Algorithm:**

```
stack<Node<Object>*> nodeStack;
nodeStack.push(startNode);

while (!nodeStack.empty()) {
    Node<Object>* current = nodeStack.top();
    nodeStack.pop();
    result.push(current->data);

    // Push right first, then left (LIFO order)
    if (current->right != nullptr)
        nodeStack.push(current->right);
    if (current->left != nullptr)
        nodeStack.push(current->left);
}
```

**Stack Behavior:**

- Process root immediately when popped
- Push right child before left child
- Left child processed next due to LIFO
- Example (4,2,1,3,6,5,7): Output: 4,2,1,3,6,5,7

**3.2 In-order Traversal (Left → Root → Right) Algorithm:**

```
stack<Node<Object>*> nodeStack;
Node<Object>* current = startNode;

while (current != nullptr || !nodeStack.empty()) {
    // Push all left children
    while (current != nullptr) {
        nodeStack.push(current);
        current = current->left;
    }

    // Process node and move to right subtree
    current = nodeStack.top();
    nodeStack.pop();
    result.push(current->data);
    current = current->right;
}
```

**Stack Behavior:**

- Push all left children until null
- Process node when popped from stack
- Move to right subtree after processing
- Produces sorted output for BST, Example (4,2,1,3,6,5,7): Output: 1,2,3,4,5,6,7

**3.3 Post-order Traversal (Left → Right → Root) Algorithm:**

**Step 1:** Initialize

- Set current = starting node
- Set lastVisited = nullptr
- Create empty stack

**Step 2**: While current != nullptr OR stack not empty:

a) If current != nullptr:
   - Push current onto stack
   - Move current to left child (go left)

b) Else (reached leftmost or returning from right):
   - Peek at top of stack (don't pop yet)
   - Check if right child exists AND has not been visited: -
     - If yes: Move current to right child (go right)
     - If no: Pop node from stack, process it, set lastVisited = popped node

**Step 3**: Return result queue

**Stack Behavior**:

- Uses SINGLE stack with lastVisited tracking variable
- The lastVisited pointer tracks the most recently processed node
- This tells us whether we've already visited the right subtree
- Process node only after BOTH children have been visited
- Most complex of the three iterative traversal implementations

- Requires careful tracking of which nodes have been processed

Why lastVisited is Needed: When we peek at a node on the stack, we need to know: "Have I already processed this node's right child?"

- If lastVisited == node's right child: Yes, process this node now
- If right child exists but lastVisited != right child: Go to right child first
- If no right child: Process this node now

Example with tree (4,2,1,3,6,5,7): Output: 1,3,2,5,7,6,4

**4. RECURSIVE METHODS**

**4.1 Insert Algorithm**

```
void insert(Node<Object>* &node, const
Object &value) {
    if (node == nullptr) {
        node = new Node<Object>(value);
    } else if (value < node->data) {
        insert(node->left, value);
    } else if (value > node->data) {
        insert(node->right, value);
    }
    // Duplicates ignored - no action
}
```

**Features:**

- Reference to pointer modifies parent's child directly
- Maintains BST property through recursion
- Duplicate values silently ignored
- Time Complexity: O(h) where h is tree height

**4.2 Retrieve Algorithm**

```
bool retrieve(Node<Object>* node,
const Object &value) const {
    if (node == nullptr) return
false;
    else if (value < node->data)
return retrieve(node->left, value);
    else if (value > node->data)
return retrieve(node->right, value);
    else return true; // value found
}
```

**Features:**

- Recursive search following BST structure
- Returns boolean existence indicator
- Time Complexity: O(h) same as insertion

**5. TESTING RESULTS**

**5.1 Comprehensive Test Summary**

| Test Category | Methods Tested | Result | Issues |
|---|---|---|---|
| Integer BST | All 5 methods | PASS | None |
| String Template | insert, retrieve, traversals | PASS | None |
| Edge Cases | All methods with edge conditions | PASS | None |
| Const Correctness | const traversal methods | PASS | None |
| Memory Management | Destructor, leak check | PASS | No leaks |

**5.2 Integer BST Tests**

- **Insertion**: Sequence 4,2,1,3,6,5,7 - correct structure
- **Duplicates**: insert(4) called twice - correctly ignored
- **Retrieval**: retrieve(7)=true, retrieve(8)=false
- **Traversals**: All three orders match expected sequences
- **Subtree**: Traversal from node 2 produces correct partial sequences

**5.3 String Template Tests**

- **Insertion**: "D","B","A","C","F","E","G" - correct structure
- **Sorting**: inorder produces alphabetical sequence
- **Retrieval**: retrieve("G")=true, retrieve("H")=false

**5.4 Edge Case Tests**

- **Empty Tree**: retrieve() returns false, traversals return empty queues
- **Single Node**: All methods work correctly with one element
- **Skewed Trees**: Right-skewed (1,2,3,4,5) maintains correct traversals
- **Non-existent Nodes**: Traversals return empty queues

**5.5 Const Correctness Test**

- **Const Traversal:** `in_order_traversal()` via const reference produces correct sorted output
- **Const Compatibility:** All traversal methods compile without error when invoked on const objects
- **Immutable Access:** Tree structure remains unmodified during const traversal operations

**5.6 Memory Validation:** `valgrind --leak-check=full ./bst_driver`

**Result**:

- All heap blocks freed - no leaks possible
- 25 allocs, 25 frees, 73,848 bytes allocated
- No memory errors detected

**6. TIME COMPLEXITY:**

- **Insert/Retrieve**: O(h) where h = tree height, **Traversals**: O(n) for all three methods
- **Best Case**: O(log n) for balanced tree, **Worst Case**: O(n) for degenerate tree

**7. KEY DESIGN DECISIONS**

**7.1 Algorithm Choices**

**Traversals:** Iterative with stacks to meet requirements

- **Preorder:** Single stack, push right then left child
- **Inorder:** Single stack, traverse left, process, go right
- **Postorder:** Single stack with lastVisited tracking (most complex)

**Memory Management: Recursive post-order destruction**

- Ensures children deleted before parent
- Prevents dangling pointers

**7.2 Error Handling**

- **Non-existent nodes**: Return empty queues (no exceptions)
- **Duplicate values**: Silent ignore (common BST convention)
- **Memory allocation**: Propagate std::bad_alloc

**7.3 Template Implementation**

- Header file with included implementation
- Support for any comparable type
- Verified with int and string types

**8. CONCLUSION**

**Implementation Status: COMPLETE AND FULLY FUNCTIONAL**

All five required methods are working correctly with no known issues. The implementation successfully demonstrates:

- **Recursive Algorithms**: Efficient insertion and retrieval maintaining BST properties
- **Iterative Traversals**: All three orders implemented with explicit stacks
- **Template Support**: Works with multiple data types including int and string
- **Memory Safety**: Zero leaks with complete resource management
- **Robust Error Handling**: Graceful handling of edge cases and invalid inputs

**References**

DeepSeek AI. (2025). [Large language model]. https://www.deepseek.com/. Assisted with debugging BST traversal implementations, clarifying stack-based algorithms, and improving documentation structure.

Microsoft Corporation. (2025). Visual Studio Code [IDE]. Used code completion and auto-documentation during BST template development.

*Note: All BST algorithms, iterative traversal implementations, and design decisions are original work. AI tools were used only for technical clarification and documentation support.*