BST and Traversal using Stack and Queue
DESIGN

1. SPECIFICATION

1.1 Problem Statement

Implement a templated Binary Search Tree class with recursive insertion and iterative traversals using explicit stack and queue data structures. The BST must maintain the invariant that left children contain smaller values and right children contain larger values than their parent.

1.2 Clarifications and Assumptions

- *Duplicate Handling*: Duplicate values during insertion are silently ignored. The tree structure remains unchanged when duplicate values are inserted.
- *Traversal Start Node*: If the starting node for traversal is not found, an empty queue is returned. No exceptions are thrown for missing nodes.
- *Template Requirements*: The template type Object must support comparison operators (<, >) and assignment operator (=). Compilation will fail for incompatible types.
- *Memory Management*: The BST class owns all Node memory and provides complete cleanup via destructor. No external memory management is required.
- *Iterative Traversal Requirement*: All three traversals must use explicit stack data structures; recursive traversal implementations are not acceptable.

1.3 Input and Output Specifications

Input Sources:

- Method calls from driver programs
- Values passed by const reference for efficiency
- Template instantiation determines data type

Valid Input:

- Any comparable type supporting required operators
- Insertion values that maintain BST property
- Existing node values for traversal starting points

Invalid Input Handling:

- Non-existent traversal nodes: return empty queue
- Memory allocation failures: propagate std::bad_alloc
- Invalid types: compile-time errors

Output Formats:

- retrieve(): boolean indicating value existence
- traversal methods: std::queue<Object> with sequence
- Empty queues for invalid traversal requests

2. ERROR DETECTION AND HANDLING

2.1 Error Conditions

- *Traversal node not found*: Return empty queue
- *Memory allocation failure*: Propagate std::bad_alloc exception
- *Empty tree operations*: Return appropriate empty/false values
- *Null pointer access*: Check before dereference operations

2.2 Error Prevention

- All pointer operations include nullptr checks
- Stack operations protected by emptiness verification
- Template constraints enforced at compile time
- Memory safety through RAII and deterministic destruction

3. PROGRAM DESIGN

3.1 Class Structure

```
BST<Object> ——composition——▶ Node<Object>
     |
     |–uses–▶ stack<Node<Object>*> (traversal algorithms)
     |
     └–produces–▶ queue<Object> (traversal results)
```

3.2 Node Class

Purpose: Fundamental tree element storing data and structure
Attributes:
- data: Object - templated data value
- left: Node* - pointer to left child
- right: Node* - pointer to right child

Operations:
- Node(const Object& value) - constructor initializing data and null children

3.3 BST Class

Purpose: Manages tree operations, memory, and public interface
Attributes:
- root: Node<Object>* - root pointer, initially nullptr

Public Operations:
- BST() - constructor initializing empty tree
- ~BST() - destructor performing complete cleanup
- insert(const Object& value) - insert value (ignore duplicates)
- retrieve(const Object& value) const - check value existence
- pre_order_traversal(const Object& value) - Root→Left→Right
- in_order_traversal(const Object& value) - Left→Root→Right
- post_order_traversal(const Object& value) - Left→Right→Root

Private Helper Operations:
- insert(Node*& node, const Object& value) - recursive insertion
- retrieve(Node* node, const Object& value) const - recursive search
- findNode(Node* node, const Object& value) const - locate subtree root
- clear(Node* node) - recursive destruction helper

4. IMPLEMENTATION AND TEST PLAN

Phase 1: Foundation, implementing node structure, BST constructor/destructor, clear helper
Unit Tests:
- Empty tree creation and destruction
- Memory leak verification with Valgrind

- Single node allocation/deallocation
  Success Criteria: No memory leaks, proper initialization

Phase 2: Core Operations, implementing recursive insert and retrieve methods
Unit Tests:
- Insert sequence: 4,2,1,3,6,5,7 - verify BST structure
- Duplicate insertion: insert(4) twice - verify ignore behavior
- Retrieval: retrieve(7)=true, retrieve(8)=false
- Skewed tree: 1,2,3,4,5 - verify BST property maintained
  Success Criteria: Correct BST structure, accurate retrieval

Phase 3: Traversal Foundation, implementing findNode helper, iterative inorder traversal
Unit Tests:
- findNode for root and subtree nodes
- inorder_traversal(4) → {1,2,3,4,5,6,7} (sorted)
- inorder_traversal(6) → {5,6,7} (subtree)
- inorder_traversal(9) → empty queue (not found)
  Success Criteria: Correct sorted output, proper error handling

Phase 4: Complete Traversals*,* implementing iterative preorder and postorder traversals
Integration Tests:
- preorder_traversal(4) → {4,2,1,3,6,5,7}
- postorder_traversal(4) → {1,3,2,5,7,6,4}
- String template: insert "D","B","A","C","F","E","G"
- inorder_traversal("D") → {"A","B","C","D","E","F","G"}
  Success Criteria: All traversals correct, template functional

Test Data Sets with Balanced Tree: 4,2,1,3,6,5,7
- Inorder: 1,2,3,4,5,6,7
- Preorder: 4,2,1,3,6,5,7
- Postorder: 1,3,2,5,7,6,4

Edge Cases:
- Empty tree: all operations return empty/false
- Single node: traversals return single element
- Skewed trees: maintain traversal correctness
- Non-existent nodes: return empty queues

Template Verification:
- Integer types: numerical comparison
- String types: lexicographical ordering
- Custom types: require operator<, operator>

Reference
AI assistance was provided by Claude (Anthropic's Claude Sonnet 4.5) for grammatical review, structural organization, and documentation formatting. All technical content, design decisions, algorithm specifications, and implementation strategies are the original work of the Scott Elliott. Original prompt to Claude: "Review this BST and Traversal design using Stack and Queue for grammatical errors and suggest improvements to document structure."