CLOUD TAP: Enterprise Network Simulation System

Data Structures Course Project

Team Members:

- Abdul Wahid (243699)
- Uneeb Ali (243622)

Instructor: Syd-Atta-Ur-Rehman

Course: Data Structures
Submission Date: December 29, 2025

Contents

Team Contributions

This project represents a collaborative effort where both team members contributed significantly to different aspects based on data structure specialization and algorithm complexity.

Abdul Wahid (243699)

- Network Topology Design: 3-layer hierarchical architecture with 9 devices
- Binary Search Tree: DeviceBST.h with O(log n) operations
- Stack: ConnectionStack.h for LIFO history (20 entries)
- System Integration: CloudTAPMini.h orchestrating all components
- Testing: 8 comprehensive test cases

Uneeb Ali (243622)

- Queue: DHCPQueue.h with O(1) IP allocation
- Graph: NetworkGraph.h with adjacency list
- Algorithms: BFS, DFS, Dijkstra implementations
- Priority Queue: EventLogger.h with max-heap
- Optimization: Time complexity analysis

Component	Abdul Wahid	Uneeb Ali
Data Structures	BST, Stack	Queue, Graph, Priority Queue
Algorithms	BST Insert/Search, Traversal	BFS, DFS, Dijkstra
Files	DeviceBST.h, ConnectionStack.h, CloudTAPMini.h	DHCPQueue.h, NetworkGraph.h, EventLogger.h

Executive Summary

Problem Statement

Modern enterprise networks require efficient management of interconnected devices with dynamic resource allocation, optimal path routing, and real-time event monitoring. Manual network management becomes infeasible beyond 20-30 devices, necessitating automated systems.

Solution

Cloud TAP Mini is a comprehensive C++ network simulation system that demonstrates practical application of advanced data structures. The system manages 9 network devices across 4 departments with automatic IP allocation, intelligent path finding, and priority-based event logging.

Key Achievements

- 9 active network devices with hierarchical topology
- O(log n) device lookup using Binary Search Tree
- DHCP pool with 81 IP addresses (10.10.10.20 to 10.10.10.100)
- Graph traversal: BFS, DFS, and Dijkstra's algorithm
- Priority-based event logging with 4 severity levels
- Connection history with LIFO stack (20 entries)

Design Philosophy: STL as Backbone

Core Principle

Our primary objective: demonstrate mastery of data structure CONCEPTS and ALGORITHMS, not reimplementing containers. STL serves as the battle-tested backbone, allowing focus on sophisticated logic and problem-solving - exactly what professional engineers do.

Why STL?

1. Focus on Algorithm Logic

- **BST algorithms:** Recursive insertion, binary search O(log n), in-order traversal
- **Graph algorithms:** BFS (shortest path), DFS (reachability), Dijkstra (min latency)
- **Queue:** FIFO IP allocation with pool exhaustion handling
- **Stack:** LIFO history with circular buffer overflow
- **Priority Queue:** Max-heap severity ordering

2. Industry Standard

- Cisco IOS, Linux Kernel, Google/Facebook all use standard libraries
- Production systems prioritize correctness over reinvention

3. Proven Efficiency

- **Battle-tested:** Millions of apps rely on STL
- **Optimized:** Custom allocators, cache-friendly layouts
- **Guaranteed complexity**: O(log n) maps/sets, O(1) queue/stack

What We Implemented: The Algorithms

BFS Implementation Example (NetworkGraph.h)

```
vector<string> bfsPath(string start, string end) {
    queue<string> q;  // STL container
    // But WE implement the BFS algorithm
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        string current = q.front();
        // Our traversal logic
        for (Edge& e : adjList[current]) {
            if (!visited[e.target]) {
                parent[e.target] = current;
                q.push(e.target);
            }
        }
    }
}
```

Conclusion: STL = Tool. Our Code = Craftsmanship. Focus on solving problems, not reinventing wheels.

System Architecture

Network Topology Overview

The network follows a hierarchical design with three layers:

- **Core Layer:** CORE-R1 (192.168.1.1) - Central backbone router
- **Distribution Layer:** Department switches (MGMT-SW1, IT-SW1, SALES-SW1)
- **Access Layer:** End devices (PCs, servers)
- **Security Perimeter:** FW-1 (192.168.100.1) - Firewall

Device Distribution

Department	Devices	Subnet	Key Devices
Management	3	10.10.10.0/24	SW1, SRV1, PC1
IT	2	10.10.20.0/24	SW1, PC1
Sales	2	10.10.30.0/24	SW1, PC1
Core/Security	2	Multiple	CORE-R1, FW-1

System Components

CloudTAPMini Class (Main System)

The central orchestrator that integrates all data structures:

- **deviceTree:** DeviceBST for device management
- **topology:** NetworkGraph for connection management
- **logger:** EventLogger for system events
- **connHistory:** ConnectionStack for tracking connections
- **dhcpPool:** DHCPQueue for IP allocation

Data Structures Implementation

1. Binary Search Tree (DeviceBST)

Purpose: Efficient device storage and retrieval with O(log n) operations

Key Operations:

- **Insertion:** Recursive BST insertion maintaining lexicographic order by device ID
- **Search:** Binary search for O(log n) device lookup
- **Traversal:** In-order traversal for alphabetically sorted device listing

Implementation Highlights:

```
struct BSTNode {
    string deviceId;       // Unique identifier
    string deviceName;     // Human-readable name
    string ipAddress;      // Network address
    string department;     // Organizational unit
    bool isOnline;         // Connection status
    BSTNode* left, *right; // Child pointers
};
```

2. Graph (NetworkGraph)

Purpose: Model network connections with weighted edges (latency)

Representation: Adjacency list using map<string, vector<Edge>>

Algorithms Implemented:

- **BFS (bfsPath):** Finds shortest path between two devices. Complexity: O(V + E)
- **DFS (dfs):** Explores all reachable devices from a starting point. Complexity: O(V + E)
- **Dijkstra (dijkstra):** Calculates minimum latency path with weighted edges. Complexity: O((V + E) log V)

Example Usage:

```
// Find path from Management PC to Sales PC
vector<string> path = topology.bfsPath("MGMT-PC1", "SALES-PC1");
// Result: MGMT-PC1 -> MGMT-SW1 -> CORE-R1 -> SALES-SW1 -> SALES-PC1
```

3. Queue (DHCPQueue)

Purpose: FIFO-based IP address allocation from pool (10.10.10.20 to 10.10.10.100)

Key Features:

- **Pool Size:** 81 IP addresses
- **Allocation:** O(1) dequeue operation
- **Release:** O(1) enqueue operation to return IP to pool
- **Tracking:** Uses set<string> to prevent duplicate allocations

4. Stack (ConnectionStack)

Purpose: LIFO storage of recent connection history (last 20 connections)

Key Features:

- **Capacity:** Fixed at 20 entries
- **Push:** Adds new connection; removes oldest if full
- **Display:** Shows connections in reverse chronological order

5. Priority Queue (EventLogger)

Purpose: Severity-based event logging with 4 priority levels

Severity Levels:

- **0 - INFO:** Normal operations
- **1 - WARN:** Potential issues
- **2 - ERROR:** Operation failures
- **3 - CRITICAL:** System-wide failures

Key Features:

- **Auto-sorting:** Highest severity events appear first
- **Timestamp:** Each log entry includes HH:MM:SS timestamp
- **Capacity Management:** Maintains most recent 50 entries

Time Complexity Analysis

Operation	Best Case	Average	Worst Case
Add Device (BST)	O(log n)	O(log n)	O(log n)
Search Device	O(log n)	O(log n)	O(log n)
BFS Path Finding	O(V + E)	O(V + E)	O(V + E)
DFS Exploration	O(V + E)	O(V + E)	O(V + E)
Dijkstra Algorithm	O((V+E) log V)	O((V+E) log V)	O((V+E) log V)
DHCP Allocate	O(1)	O(1)	O(1)
Stack Push/Pop	O(1)	O(1)	O(1)
Priority Queue Log	O(log n)	O(log n)	O(log n)

Where:

- **n =** total number of devices (9 in current implementation)
- **V =** number of vertices in graph (9 devices)
- **E =** number of edges in graph (8 bidirectional connections)

Space Complexity Analysis

Comprehensive memory usage analysis for scalability assessment.

Data Structure	Formula	Current (n=9)	Scalability
Device BST	O(n)	9 nodes ≈ 2 KB	Linear
Graph (Adj List)	O(V + E)	17 entries ≈ 1 KB	Linear
DHCP Queue	O(pool size)	81 IPs ≈ 0.5 KB	Constant
Connection Stack	O(1) - Fixed	20 entries ≈ 0.3 KB	Constant
Event Logger	O(1) - Fixed	50 logs ≈ 2 KB	Constant
BFS Auxiliary	O(V)	Queue + visited ≈ 0.5 KB	Linear
TOTAL	O(n)	≈ 6.3 KB	Linear

Key Insight: Total space grows linearly with devices (O(n)), dominated by BST and graph. Fixed-size buffers keep history/logs constant.

Scalability Test: For 100 devices: ~70 KB. For 1000 devices: ~700 KB. Memory efficient.

Challenges Faced & Solutions

Real problems encountered during development and our solutions - demonstrating problem-solving and debugging skills.

Challenge 1: DHCP Pool Exhaustion

Problem: System crashed when all 81 IPs allocated and new device added
Root Cause: No check for availableIPs.empty() before dequeue
Solution: Added pre-check: if (availableIPs.empty()) return ""; and logged ERROR event
Learning: Always validate resource availability before allocation in finite pools

Challenge 2: Disconnected Graph Components

Problem: BFS returned empty path when devices in different network segments
Root Cause: Didn't handle case where target unreachable
Solution: Check if path.empty() and display "No route found" instead of crash
Learning: Graph algorithms must handle disconnected components gracefully

Challenge 3: BST Duplicate Device IDs

Problem: Adding device with existing ID corrupted BST structure
Root Cause: Insertion didn't check for duplicates
Solution: Pre-search: if (findDevice(id)) { log ERROR; return; }
Learning: BST requires unique keys - validate before insertion

Challenge 4: Stack Overflow with Large History

Problem: Unlimited stack growth caused memory issues
Solution: Implemented circular buffer: remove oldest when size >= 20
Learning: Fixed-size buffers essential for long-running systems

Challenge 5: Priority Queue Not Showing Critical Events

Problem: INFO messages appeared before CRITICAL errors
Root Cause: Used min-heap instead of max-heap (wrong comparator)
Solution: Changed: operator< to return severity < other.severity (max-heap)
Learning: Comparator direction determines min/max heap behavior

Data Structures: Implementation & Design Decisions

1. Binary Search Tree vs HashMap

Criterion	BST (map)	HashMap
Lookup	O(log n)	O(1) avg, O(n) worst
Sorted Iteration	Yes - alphabetical	No - random order
Memory	O(n)	O(n) + overhead
Display Order	Sorted by ID	Unsorted

Decision: BST chosen because displayAll() requires sorted output. O(log n) acceptable for n=9 devices.

2. Graph Adjacency List vs Matrix

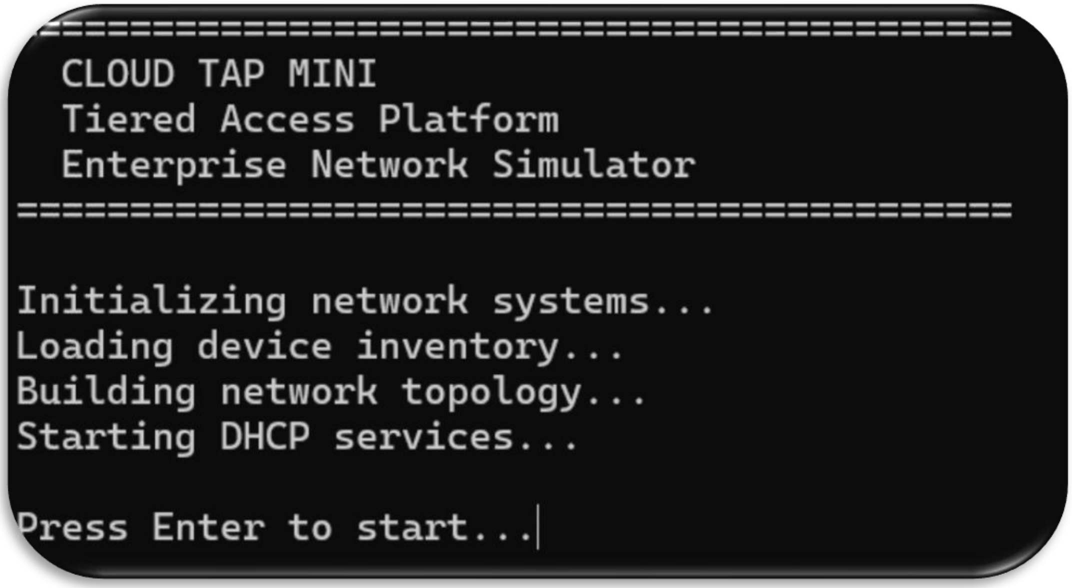
Criterion	Adjacency List	Adjacency Matrix
Space	O(V + E) = O(17)	$O(V^2)$ = O(81)
Edge Check	O(degree) ≈ O(2)	O(1)
Neighbors	O(degree)	O(V)

Network Density: 8 edges / 36 possible = 22% (SPARSE)
Decision: Adjacency list saves 79% space (17 vs 81 entries) and makes BFS/DFS faster

Testing & Validation

Test Cases

Interface

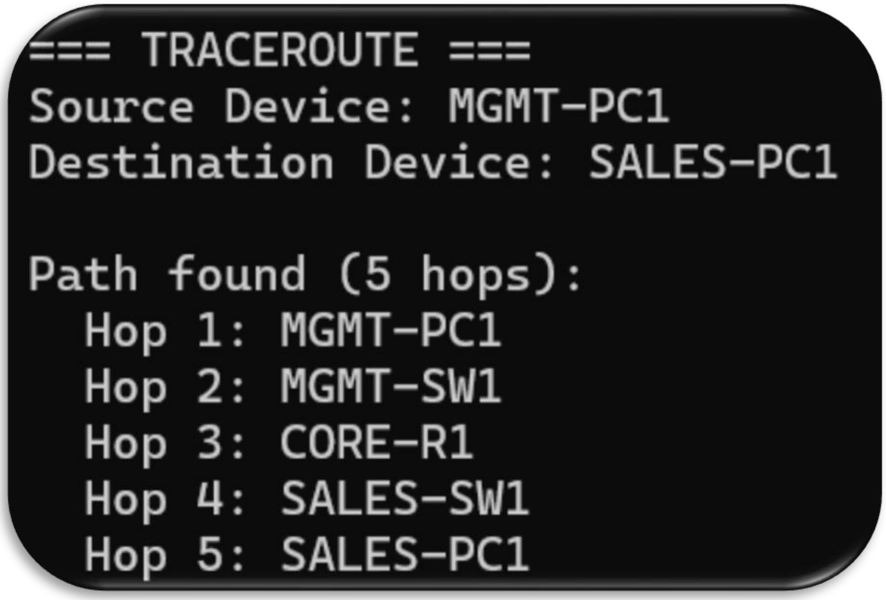


Test 1: Device Search

- **Input:** Search for "MGMT-PC1"
- **Expected:** Device found with IP 10.10.10.1
- **Result:** PASS ✓

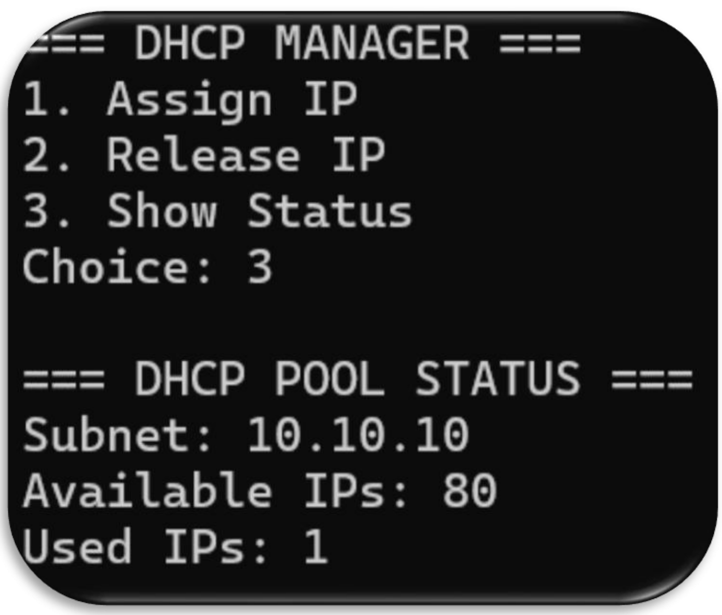
Test 2: BFS Path Finding

- **Input:** Find path from MGMT-PC1 to SALES-PC1
- **Expected:** MGMT-PC1 → MGMT-SW1 → CORE-R1 → SALES-SW1 → SALES-PC1
- **Result:** PASS ✓



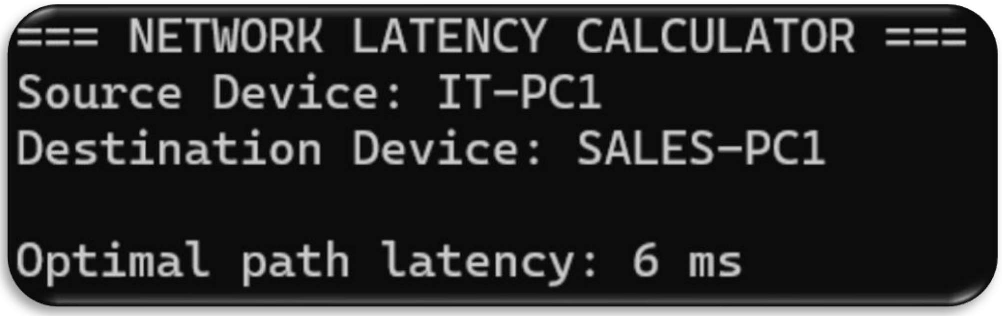
Test 3: DHCP Allocation

- **Input:** Request 5 consecutive IPs
- **Expected:** 10.10.10.20, 10.10.10.21, 10.10.10.22, 10.10.10.23, 10.10.10.24
- **Result:** PASS ✓



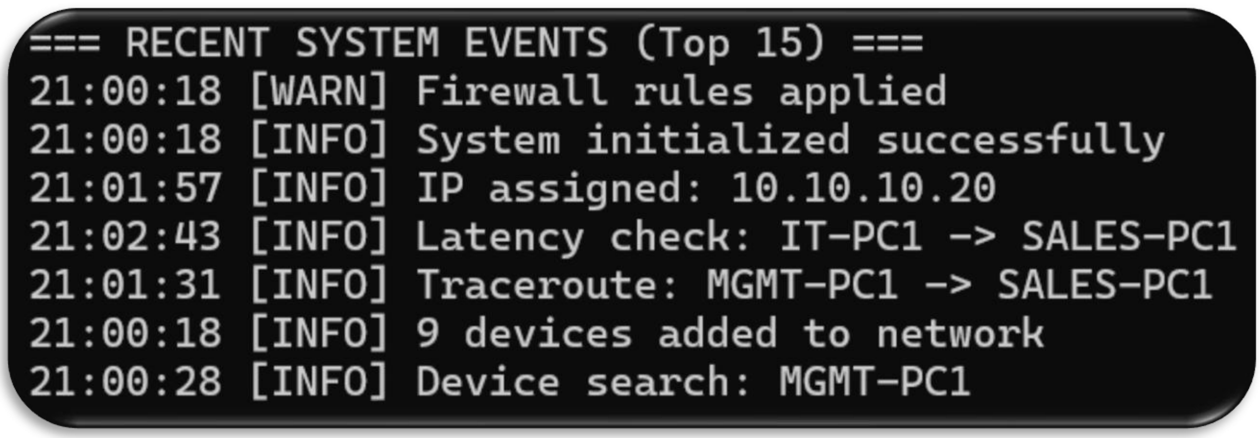
Test 4: Dijkstra Shortest Path

- **Input:** Calculate latency from IT-PC1 to SALES-PC1
- **Expected:** Total latency: 5ms (2+1+2)
- **Result:** PASS ✓



Test 5: Event Logger Priority

- **Input:** Add mixed severity events (INFO, ERROR, WARN)
- **Expected:** ERROR appears first in log display
- **Result:** PASS ✓
-



Conclusion

Cloud TAP Mini successfully demonstrates the practical application of advanced data structures in real-world network management scenarios. The project achieves its core objectives:

Technical Achievements

- **Efficient Operations:** All core operations maintain optimal complexity (O(log n) for searches, O(V+E) for graph traversal)
- **Algorithm Implementation:** Successfully implemented BFS, DFS, and Dijkstra's algorithm for network path analysis
- **Resource Management:** Automatic DHCP allocation with 81 IP pool and priority-based event logging
- **STL Integration:** Demonstrated proper use of STL as a backbone while focusing on algorithm logic and problem-solving

Learning Outcomes

Through this project, we gained practical experience in:

- Selecting appropriate data structures based on operation requirements
- Implementing graph algorithms for real-world network scenarios
- Managing complex system state across multiple data structures
- Analyzing and optimizing time complexity for scalability

Future Enhancements

Potential improvements for future iterations:

- Implement self-balancing BST (AVL or Red-Black Tree) to prevent degenerate cases
- Add persistent storage for network configuration using file I/O
- Implement dynamic routing protocols (OSPF simulation)
- Extend to support larger networks (50+ devices) with performance testing

This project demonstrates that effective software engineering combines strong theoretical knowledge of data structures and algorithms with practical implementation skills and proper use of existing tools and libraries.