MECS1023 ADSA Assignment

“Medicine Management System”

Student Name : Lau Su Hui (Abby)
Matric No. : MEC245045
Semester : 20252026-1
Title : Task 3 - Proposal & Analysis Report
Theme : Group 1 - Pharmacy Inventory Control System

PRESENTATION

YEAR 2025

Application Proposal

Project Synopsis

1) **Problem:** Manual medicine batch records are slow, error-prone, and hard to search, tracking in pharmacies is inefficient and slow.

2) **Solution:** A digital system "Medicine Management System" using search tree structures (BST & AVL) to manage thousands of medicine batches instantly.

3) **Goal:** To create an accessible, high-performance, organised, and reliable medicine management system that assists medical staff, allowing them focus on patient care.

Project Objectives

1) **Dynamic Scalability:** Replace static arrays with memory-efficient tree structures.

2) **Data Integrity:** Implement unique batch ID validation to prevent record duplicates.

3) **Optimised Search:** Ensure high-speed retrieval (O(log N)) for 10,000+ records.

4) **Performance Analysis:** Compare BST vs. AVL stability in pharmacy workflows.

5) **Practical Implementation:** Apply recursive logic and tree balancing to an efficient medicine management system

Solution 1: BST with ADT

1) **ADT Strategy:**
   - **Class:** MedicineManager encapsulates all logic.
   - **Private:** Node* root, insertInternal.
   - **Public:** addMedicine, findMedicine, removeMedicine.

2) **Characteristics:**
   - Uses standard recursive logic: Smaller items go Left, Larger items go Right.
   - **Pros:** Simple structure, slightly faster insertion for random data.
   - **Cons:** No self-balancing. Risks degrading to O(N) speed if data is sorted.

Solution 2: AVL with ADT

1) **ADT Strategy:**
   - **Class:** MedicineManager encapsulates all logic.
   - **Private:** Node* root, insertInternal, right/leftRotate.
   - **Public:** addMedicine, findMedicine, removeMedicine.

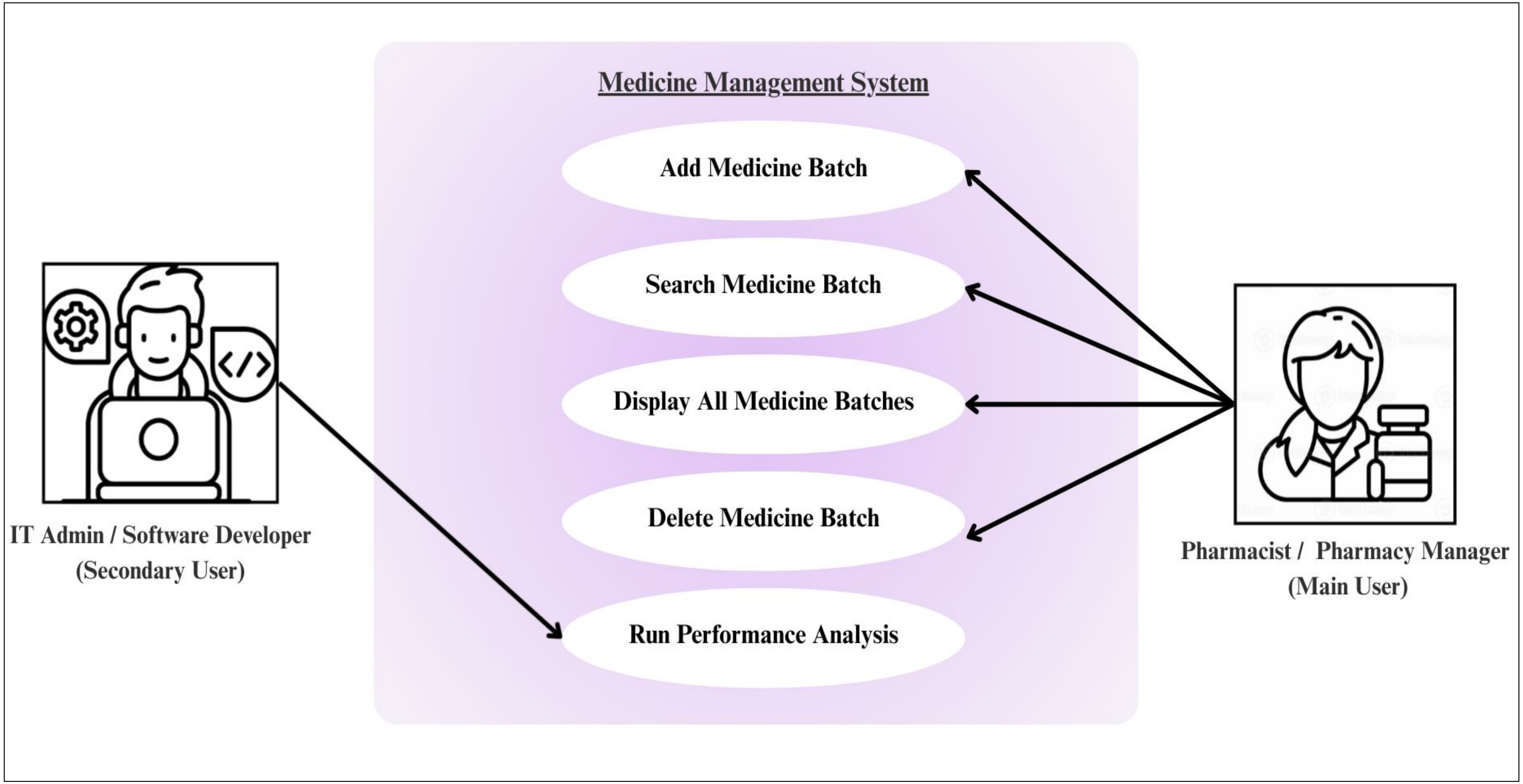
2) **Why AVL?:**
   - Standard BSTs degrade to O(N) speed if data is entered in sorted order.
   - AVL enforces balance using Height and Rotations, it can guarantee O(log N) speed.

Solution 2: AVL with ADT

3) AVL enforces balance using Height and Rotations*

```
    updateHeight(node);

    int balance = getBalance(node);

    if (balance > 1 && med.batchID < node->left->data.batchID)
        return rightRotate(node);

    if (balance < -1 && med.batchID > node->right->data.batchID)
        return leftRotate(node);

    if (balance > 1 && med.batchID > node->left->data.batchID) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && med.batchID < node->right->data.batchID) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}
```

System Requirements



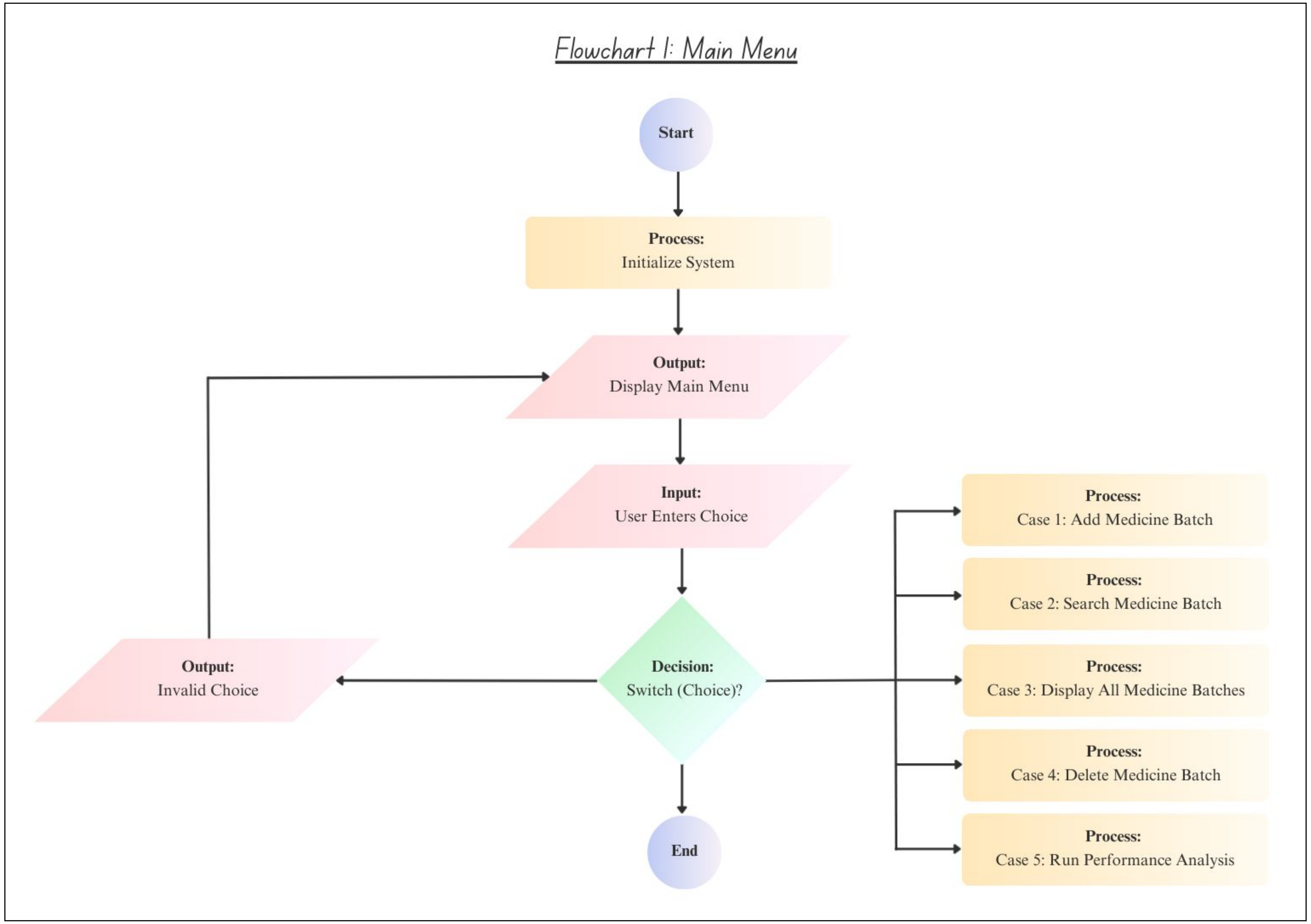
Use Case Diagram

System Requirements

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

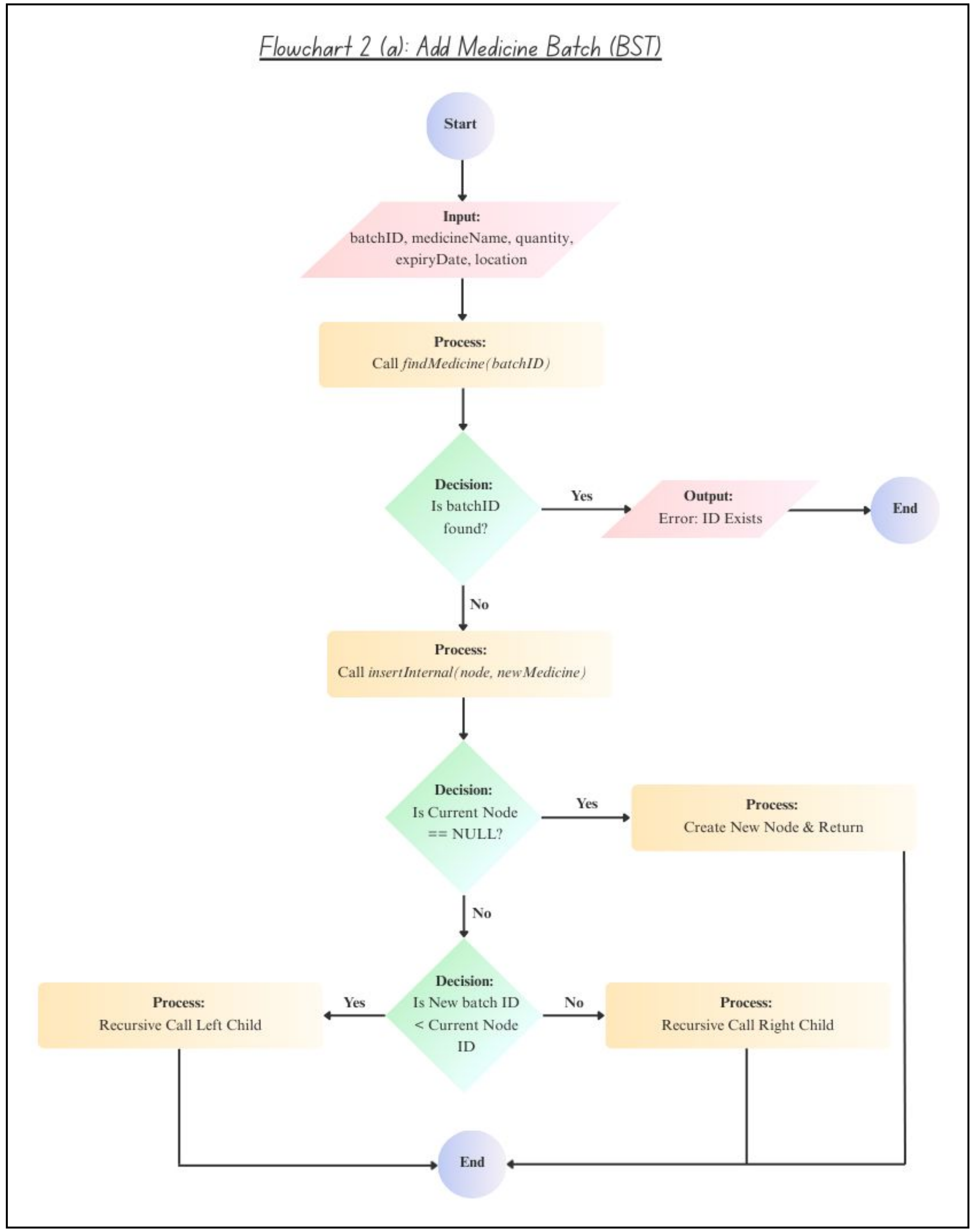
Actors	**Use Cases / Tasks**
Main User: Pharmacist / Pharmacy Manager	**a) Add medicine batch (Include: Check Duplicate ID)** = 1. Add New Medicine Batch➕ **b) Search medicine batch (Input: Batch ID)** = 2. Search by Batch ID🔎 **c) Display all medicine batches (Output: Sorted List)** = 3. Display All Batches💻 **d) Delete medicine batch (Input: Batch ID)** = 4. Delete Batch➖
Secondary User: IT Admin / Software Developer	**e) Run Performance Analysis (Generate Random Datasets & Report)** = 5. Analysis Report: Run Performance Experiment📊

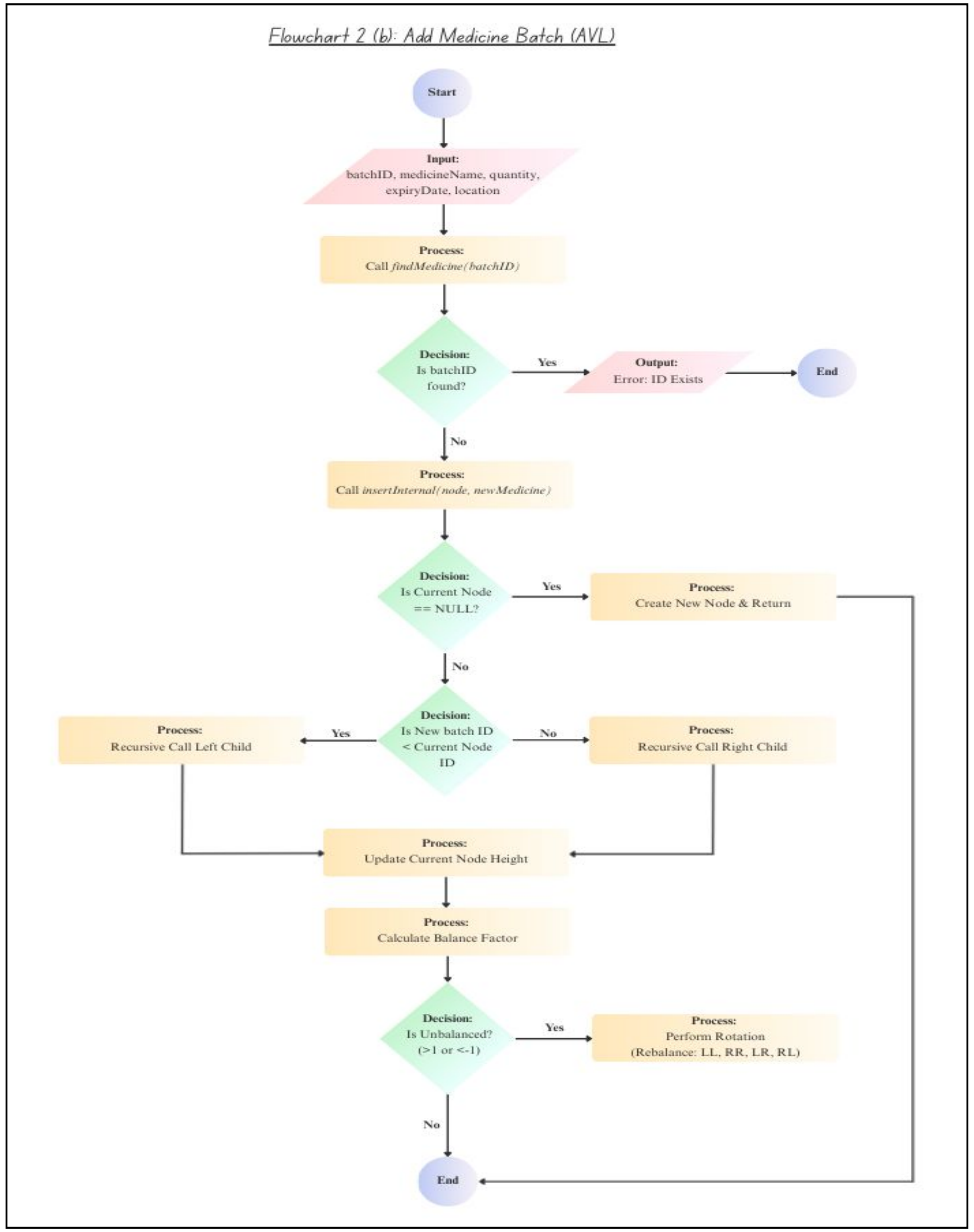
System Design



Flowchart 1: [Main Menu]

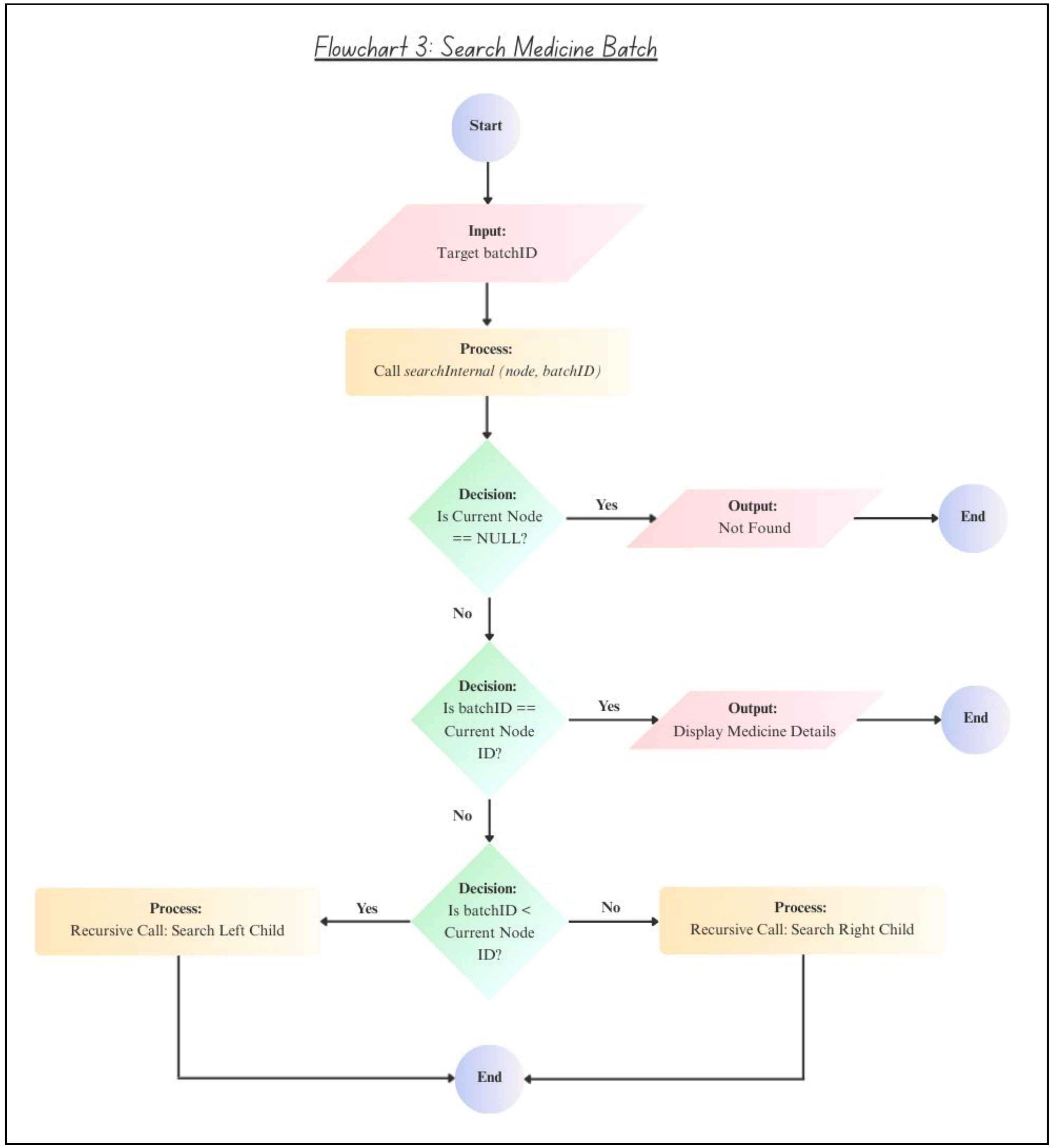
System Design





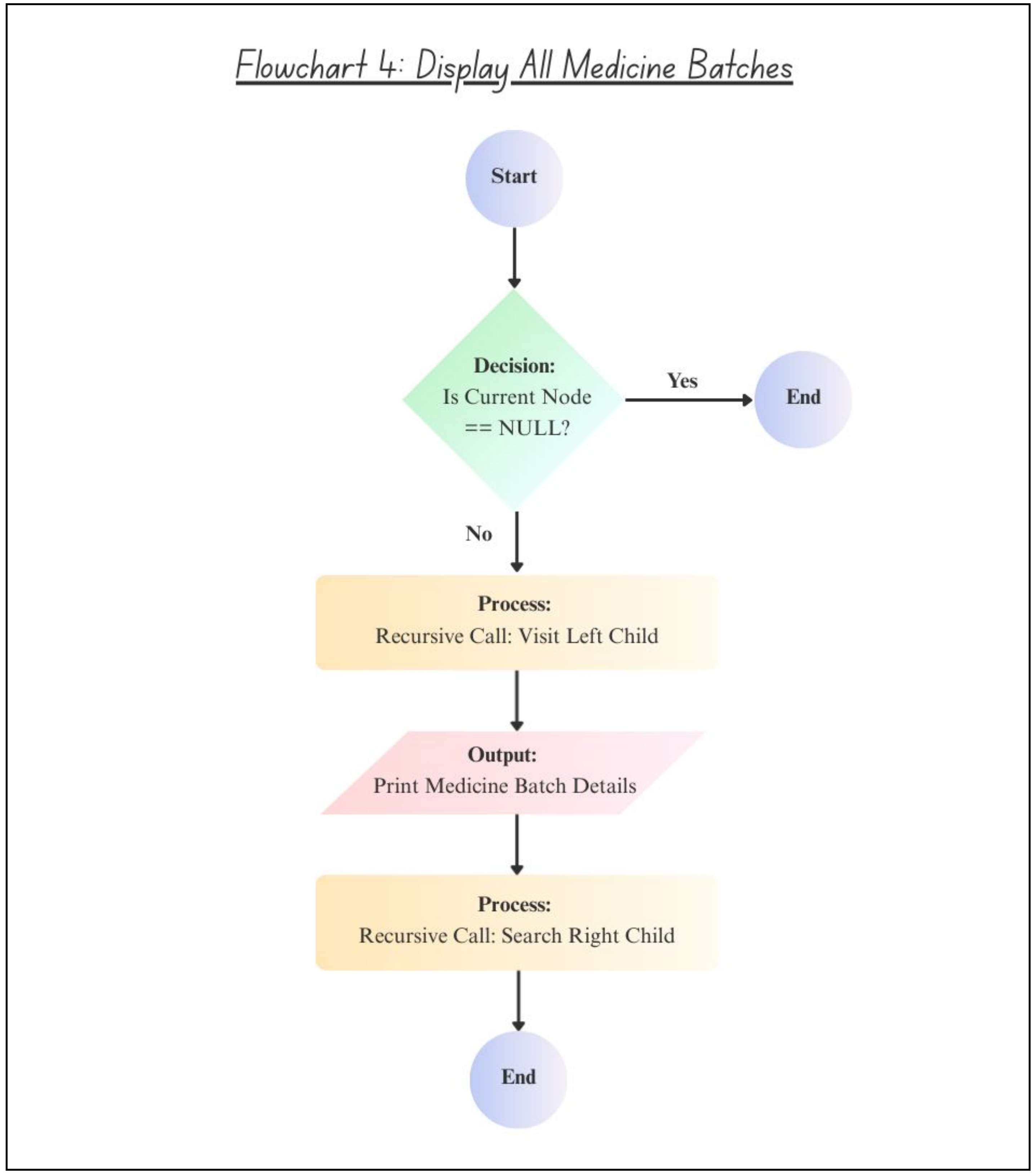
Flowchart 2: Add Medicine Batch (Insertion)

System Design



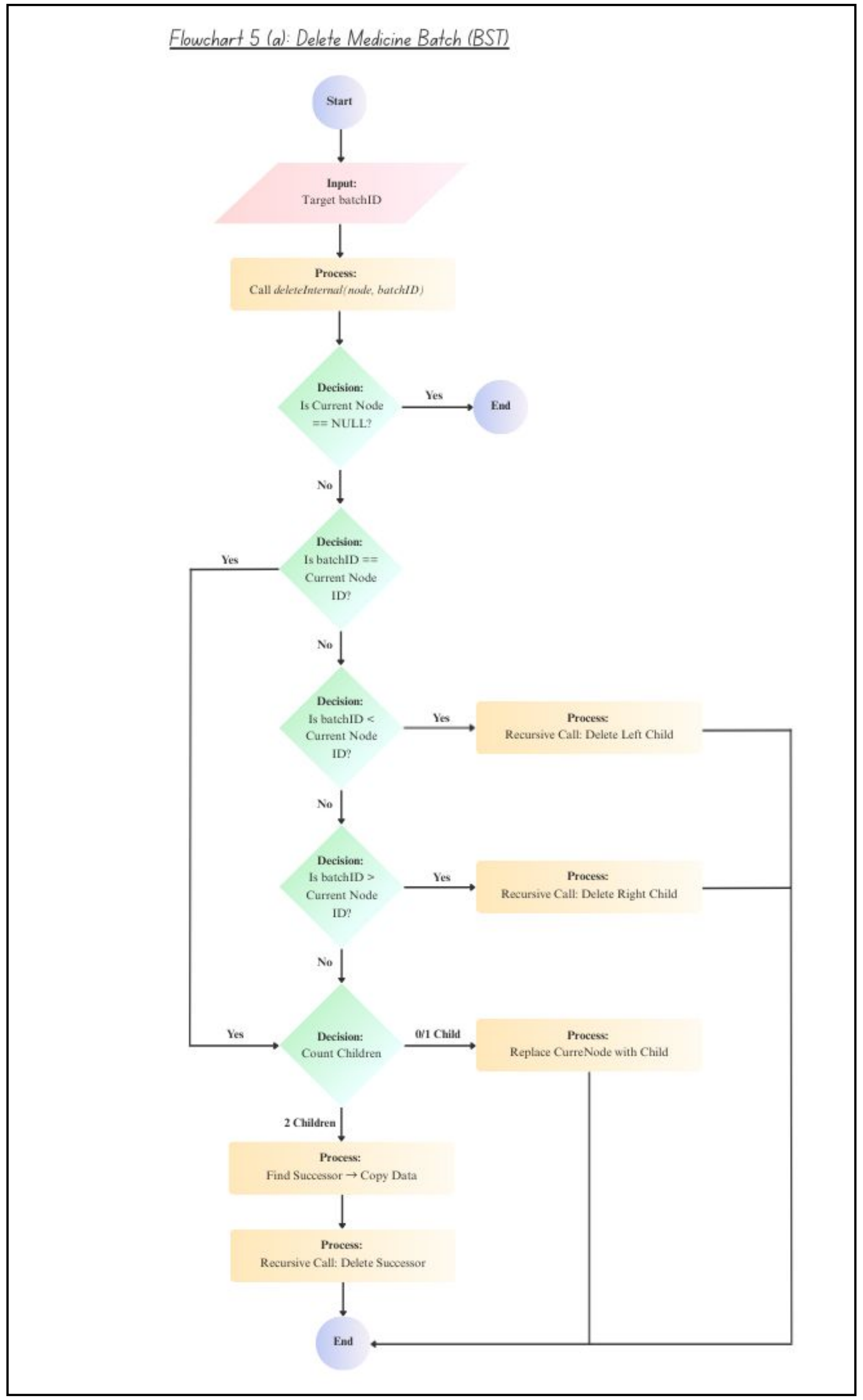
Flowchart 3: Search Medicine Batch (Search)

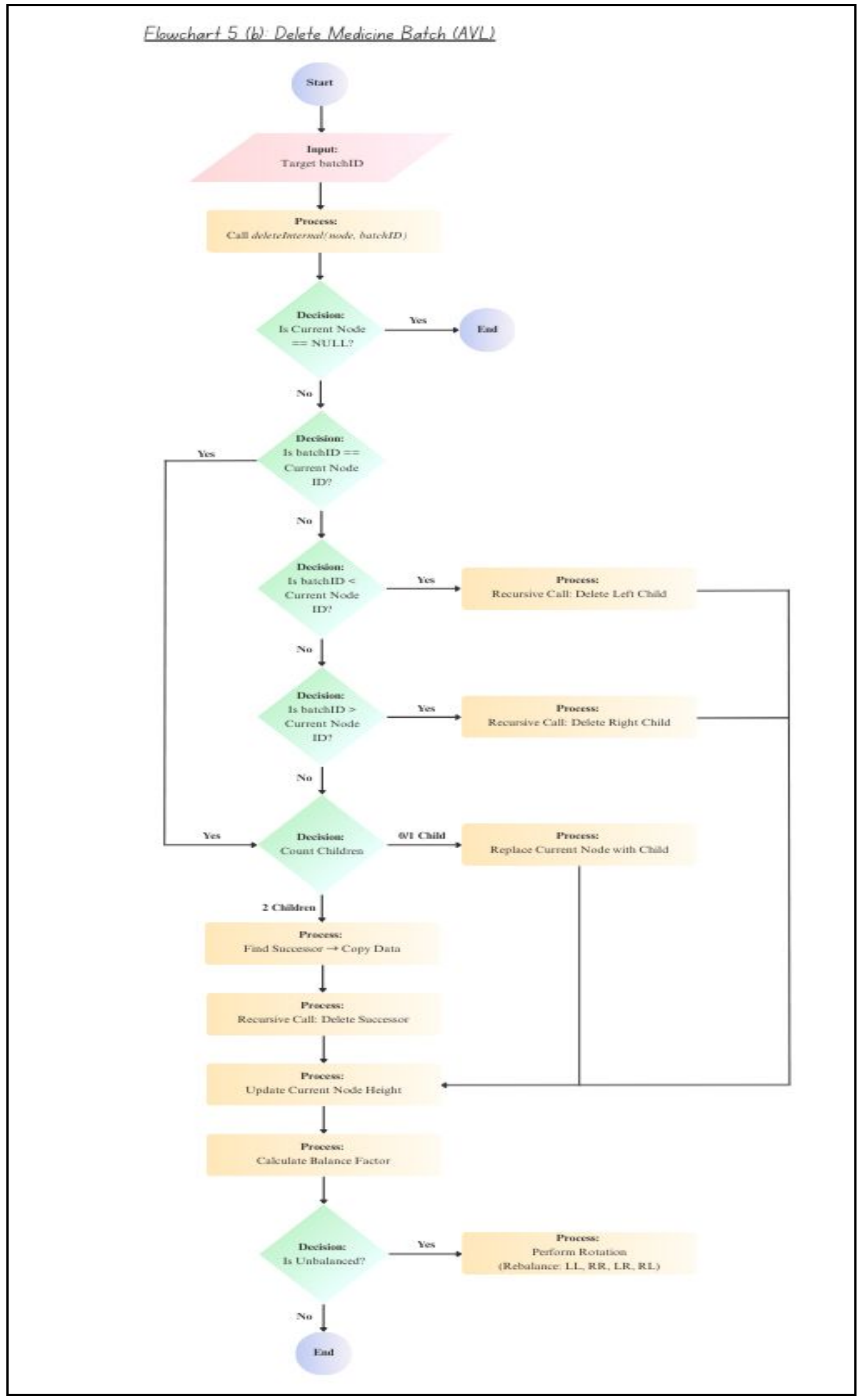
System Design



Flowchart 4: Display All Medicine Batches

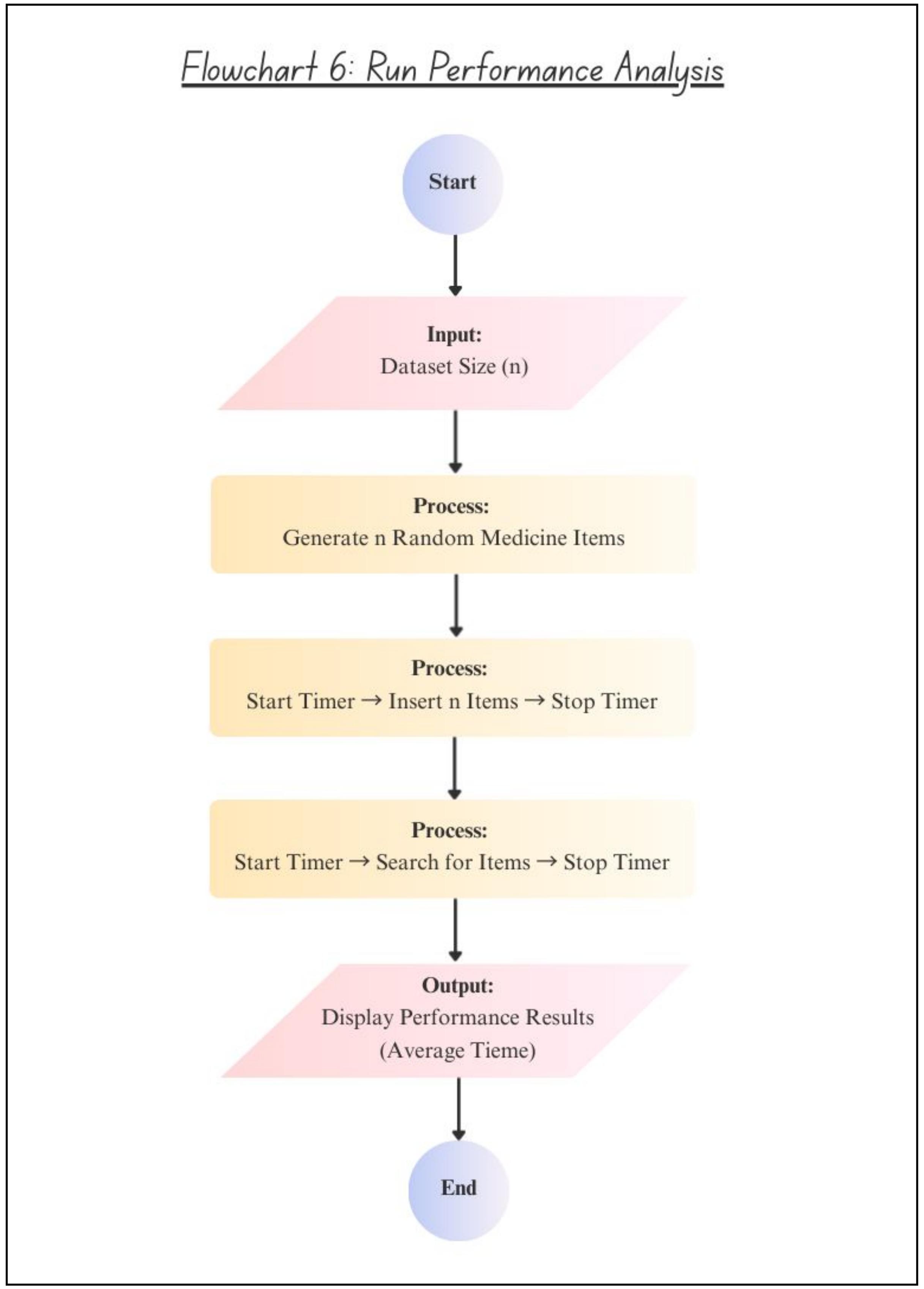
System Design





Flowchart 5: Delete Medicine Batch (Deletion)

System Design



Flowchart 6: Run Performance Analysis

Data Structure Design

```cpp
struct Medicine {
    string batchID;
    string medicineName;
    int quantity;
    string expiryDate;
    string location;
};

struct Node {
    Medicine data;
    Node *left;
    Node *right;

    Node(Medicine med) {
        data = med;
        left = nullptr;
        right = nullptr;
    }
};
```

BST

```cpp
struct Medicine {
    string batchID;
    string medicineName;
    int quantity;
    string expiryDate;
    string location;
};

struct Node {
    Medicine data;
    Node *left;
    Node *right;
    int height;

    Node(Medicine med) {
        data = med;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};
```

AVL

Key Functions: Insert

```
Node *insertInternal(Node *current, Medicine med, bool showErrors) {
  if (current == nullptr) {
    return new Node(med);
  }

  if (med.batchID < current->data.batchID) {
    current->left = insertInternal(current->left, med, showErrors);
  } else if (med.batchID > current->data.batchID) {
    current->right = insertInternal(current->right, med, showErrors);
  } else {
    if (showErrors) {
      cout << "Error: Batch ID " << med.batchID << " already exists.🚫" << endl;
    }
  }
  return current;
}
```

BST

```
Node *insertInternal(Node *current, Medicine med, bool showErrors) {
  if (current == nullptr) {
    return new Node(med);
  }

  if (med.batchID < current->data.batchID) {
    current->left = insertInternal(current->left, med, showErrors);
  } else if (med.batchID > current->data.batchID) {
    current->right = insertInternal(current->right, med, showErrors);
  } else {
    if (showErrors) {
      cout << "Error: Batch ID " << med.batchID << " already exists.🚫" << endl;
    }
    return current;
  }

  updateHeight(current);

  int balance = getBalance(current);

  if (balance > 1 && med.batchID < current->left->data.batchID)
    return rightRotate(current);

  if (balance < -1 && med.batchID > current->right->data.batchID)
    return leftRotate(current);

  if (balance > 1 && med.batchID > current->left->data.batchID) {
    current->left = leftRotate(current->left);
    return rightRotate(current);
  }

  if (balance < -1 && med.batchID < current->right->data.batchID) {
    current->right = rightRotate(current->right);
    return leftRotate(current);
  }
```

AVL

Key Functions: Search

```cpp
Node *searchInternal(Node *current, string batchID) {
  if (current == nullptr || current->data.batchID == batchID) {
    return current;
  }
  if (batchID < current->data.batchID) {
    return searchInternal(current->left, batchID);
  } else {
    return searchInternal(current->right, batchID);
  }
}
```

BST

```cpp
Node *searchInternal(Node *current, string batchID) {
  if (current == nullptr || current->data.batchID == batchID) {
    return current;
  }
  if (batchID < current->data.batchID) {
    return searchInternal(current->left, batchID);
  } else {
    return searchInternal(current->right, batchID);
  }
}
```

AVL

Key Functions: Delete

UTM UNIVERSITI TEKNOLOGI MALAYSIA

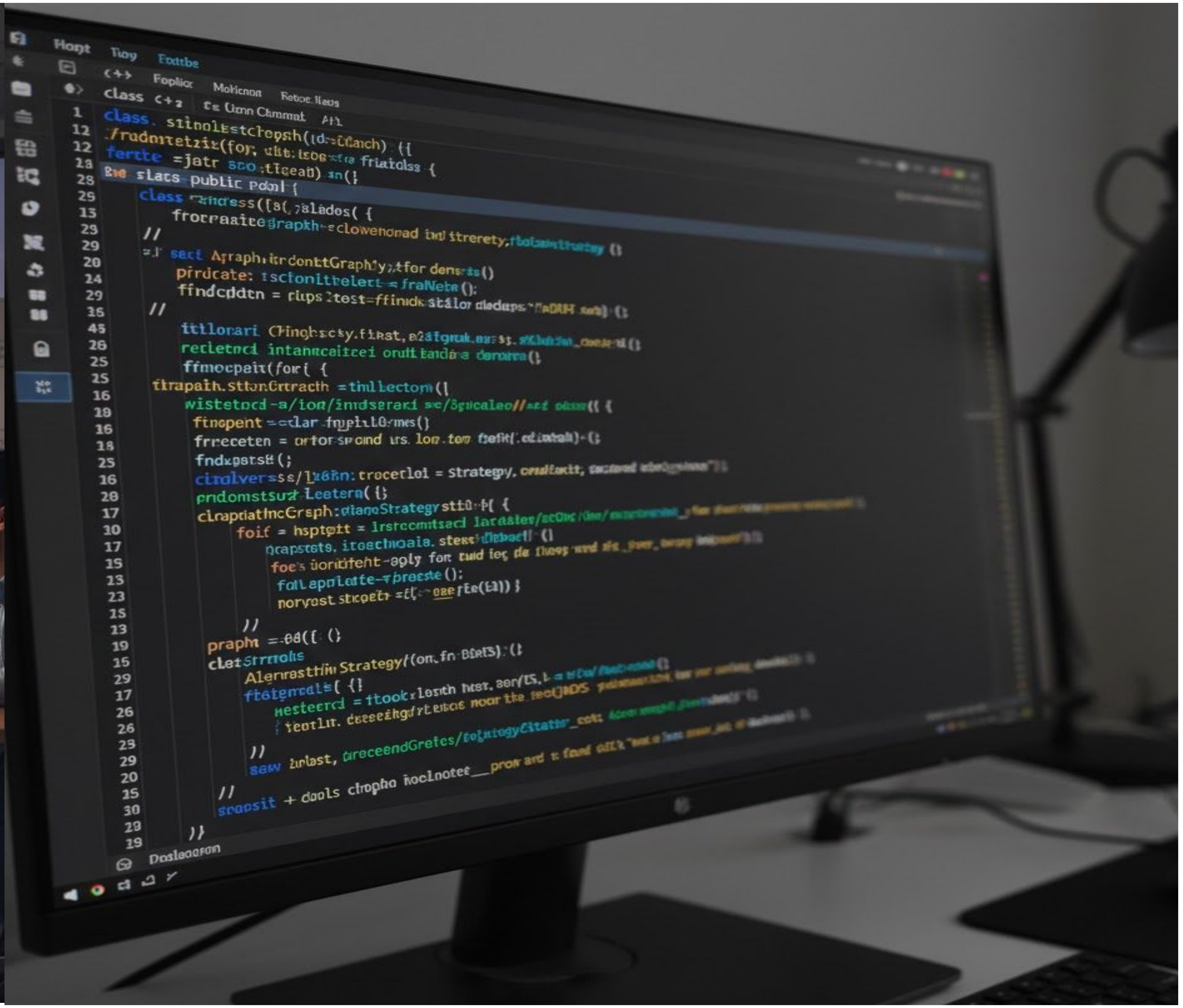
```cpp
Node *removeInternal(Node *current, string batchID) {
  if (current == nullptr)
    return current;
  if (batchID < current->data.batchID) {
    current->left = removeInternal(current->left, batchID);
  } else if (batchID > current->data.batchID) {
    current->right = removeInternal(current->right, batchID);
  } else {
    if (current->left == nullptr) {
      Node *temp = current->right;
      delete current;
      return temp;
    } else if (current->right == nullptr) {
      Node *temp = current->left;
      delete current;
      return temp;
    }
    Node *temp = findMinNode(current->right);
    current->data = temp->data;
    current->right = removeInternal(current->right, temp->data.batchID);
  }
  return current;
}
```

BST

```cpp
Node *removeInternal(Node *root, string batchID) {
  if (root == nullptr)
    return root;

  if (batchID < root->data.batchID) {
    root->left = removeInternal(root->left, batchID);
  } else if (batchID > root->data.batchID) {
    root->right = removeInternal(root->right, batchID);
  } else {

    if ((root->left == nullptr) || (root->right == nullptr)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == nullptr) {
        temp = root;
        root = nullptr;
      } else {
        *root = *temp;
      }
      delete temp;
    } else {
      Node *temp = findMinNode(root->right);
      root->data = temp->data;
      root->right = removeInternal(root->right, temp->data.batchID);
    }
  }
  if (root == nullptr)
    return root;

  updateHeight(root);

  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

  if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}
```

AVL*

System Prototype & Demonstration



Analysis Report

Experiment Setup

Mode	Random Mode
Input Sizes	N = 1,000, 5,000, and 10,000.
Justification	N = 10,000 was selected as the maximum to simulate a realistic, busy pharmacy inventory.
Optimal Tree Order	AVL Tree
Reason	AVL maintained a height of ≈14 levels for 10,000 items (calculated via O(log N)), whereas the BST height varied unpredictably.

Performance Results

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 1000

Creating 1000 random medicines...

--- BST Analysis Report Data (Size: 1000) ---
Avg Insert Time: 34.662 microseconds
Avg Search Time: 1.127 microseconds
Total Time: 34662 (Insert) / 1127 (Search)
```

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 5000

Creating 5000 random medicines...

--- BST Analysis Report Data (Size: 5000) ---
Avg Insert Time: 116.663 microseconds
Avg Search Time: 1.9392 microseconds
Total Time: 583314 (Insert) / 9696 (Search)
```

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 10000

Creating 10000 random medicines...

--- BST Analysis Report Data (Size: 10000) ---
Avg Insert Time: 73.287 microseconds
Avg Search Time: 1.9223 microseconds
Total Time: 732870 (Insert) / 19223 (Search)
```

BST Results

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 1000

Creating 1000 random medicines...

--- AVL Analysis Report Data (Size: 1000) ---
Avg Insert Time: 36.678 microseconds
Avg Search Time: 3.833 microseconds
Total Time: 36678 (Insert) / 3833 (Search)
```

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 5000

Creating 5000 random medicines...

--- AVL Analysis Report Data (Size: 5000) ---
Avg Insert Time: 108.605 microseconds
Avg Search Time: 1.4742 microseconds
Total Time: 543023 (Insert) / 7371 (Search)
```

```
--- 📊Performance Experiment📊 ---
Enter dataset size N (e.g. 1000, 5000, 10000): 10000

Creating 10000 random medicines...

--- AVL Analysis Report Data (Size: 10000) ---
Avg Insert Time: 73.7873 microseconds
Avg Search Time: 1.6463 microseconds
Total Time: 737873 (Insert) / 16463 (Search)
```

AVL Results

Performance Results

Performance Comparison (BST vs. AVL)

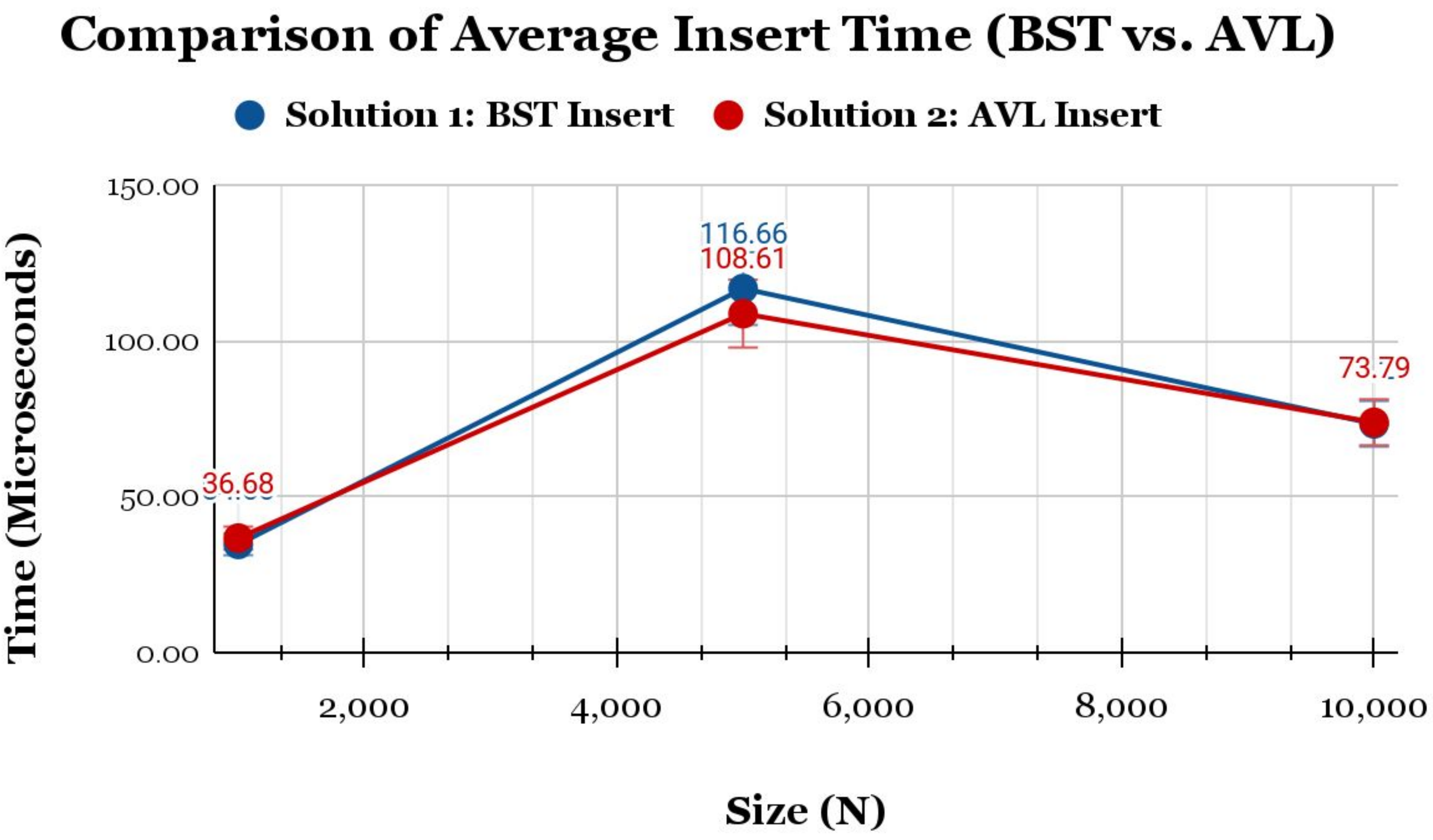
Dataset Size (N)	Average Times	Solution 1 (BST)	Solution 2 (AVL)	Difference	Winner
N = 1,000	Insertion	34.66 µs	36.68 µs	+2.02 µs	BST
	Search	1.13 µs	3.83 µs	+2.70 µs	BST
N = 5,000	Insertion	116.66 µs	108.61 µs	-8.05 µs	AVL
	Search	1.94 µs	1.47 µs	-0.47 µs	AVL
N = 10,000	Insertion	73.29 µs	73.79 µs	+0.50 µs	Tie (~Equal)
	Search	1.92 µs	1.65 µs	-0.27 µs	AVL

Performance Results: Insert Time

BST & AVL's Average Insert Time Results

Size (N)	Solution 1: BST Insert	Solution 2: AVL Insert
1,000	34.66 µs	36.68 µs
5,000	116.66 µs	108.61 µs
10,000	73.29 µs	73.79 µs

Performance Chart: Insert Time

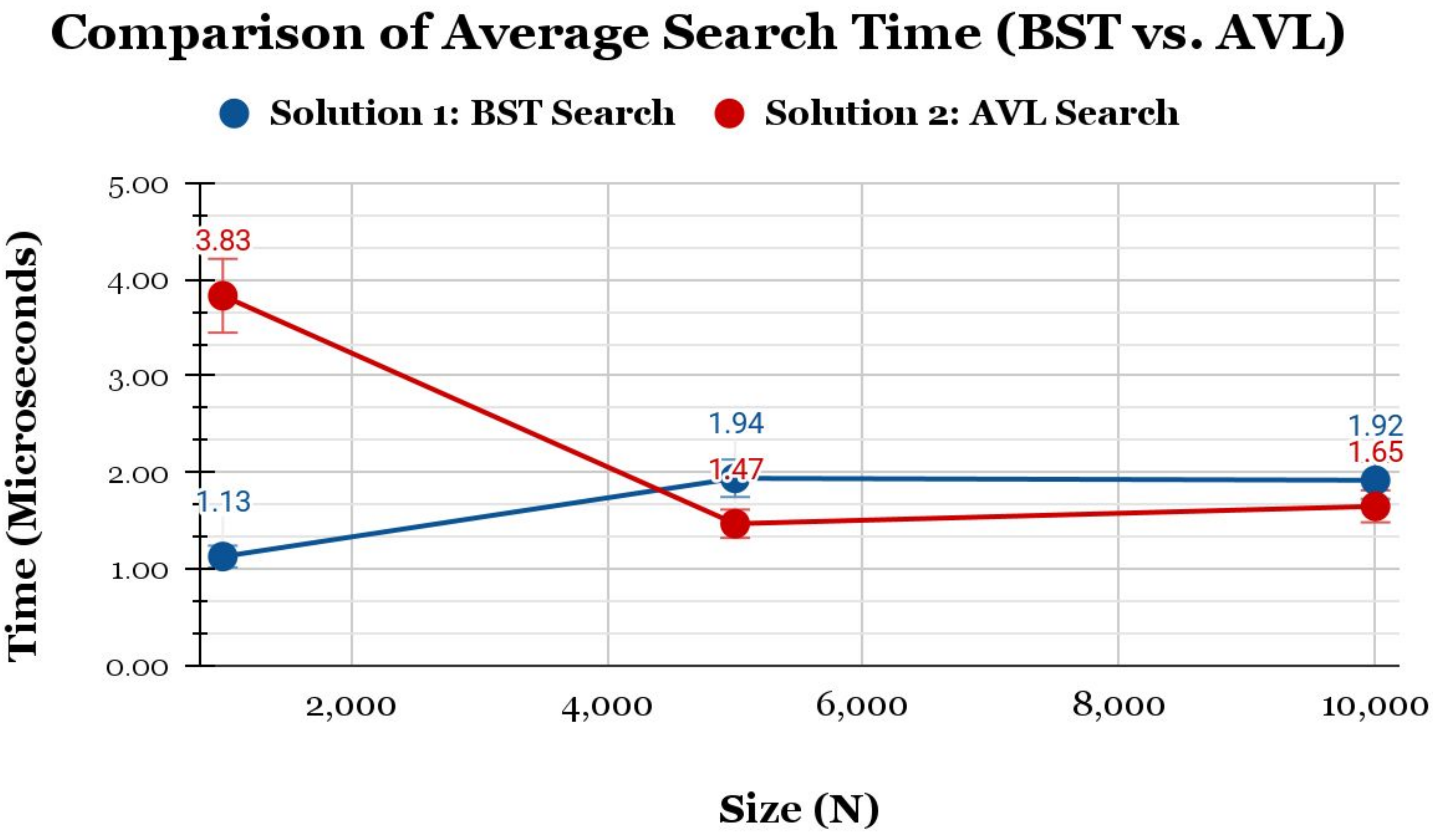


Performance Results: Search Time

BST & AVL's Average Search Time Results

Size (N)	Solution 1: BST Search	Solution 2: AVL Search
1,000	1.13 µs	3.83 µs
5,000	1.94 µs	1.47 µs
10,000	1.92 µs	1.65 µs

Performance Chart: Search Time



Insertion Analysis: The Integrity Cost

1) **Duplicate Check:** We do not just "add" data, also search for duplicates first. The Data Integrity Check takes the most time & does increase the insertion time.

2) **The “Maths” Overhead:** AVL is slightly slower to insert because it has extra steps for rotations and height updates to stay balanced.

3) **The N = 5,000 Anomaly**: In our tests, AVL (108.61 µs) was actually faster than BST (116.66 µs). This is because the BST became a little unbalanced, making searches slower, while the AVL tree stayed short and balanced, so checking for duplicate values was faster.

4) **Result:** The extra work (rebalancing cost) is worthwhile trade-off to keep searches fast for the user during the high-volume data entry.

Search Analysis: The Scalability Factor

1) **The Tipping Point:** At small sizes (N = 1,000), BST is faster. But as the database grows to N = 10,000, AVL takes the lead and wins. AVL is 14% faster (1.65 μs vs 1.92 μs) than BST.

2) **Guaranteed Height:** AVL limits the search path at ≈14 levels. This ensures logarithmic retrieval efficiency regardless of dataset size.

3) **Stability & Reliability:** Automatically prevents the O(N) "linear slowdown" that occurs if medicine batches are entered in sorted order. BST risks becoming "skewed" (slow) if data is entered in order. AVL never slows down.

4) **Predictable Scalability:** Doubling the data (from N = 5,000 to N = 10,000) only increased search time by a tiny 12.2%.

Recommendation: AVL - The Best Way

1) **Recommendation:** Deploying the AVL Tree, if expecting the system to reach N = 10,000 records

2) **Reasons:**
   a) Guaranteed search speed & predictable scalability
   b) Prevents system slowdowns/lag
   c) Optimised for read-heavy usage
   d) Reliable data integrity checks

Conclusion & Appendix

Conclusion

1) **Performance Benchmarks: N = 10,000**
- **Search:** AVL (1.65 µs) vs. BST (1.92 µs) → proves logarithmic efficiency.
- **Insertion:** BST (73.29 µs) vs. AVL (73.79 µs) → shows minimal balancing cost.
- **Integrity:** Pre-insertion search used to block duplicate Batch IDs.

2) **Recommendation: AVL Tree**
- **Consistency:** Guarantees ≈14 levels for stable, predictable speed.
- **Risk Mitigation:** Prevents O(N) linear degradation common in standard BSTs.
- **Verdict:** High-speed retrieval is more critical for pharmacists than minor insertion latency.

3) **Future Improvement:**
- **Edit Medicine Batch Feature:** Replace the full delete-and-re-add process with direct, in-place updates to medicine details.

Appendix

1) **Source Code (BST & AVL in C++):** GitHub Link
2) **Use Case Diagram:** Use Case Diagram
3) **Flowcharts:** Flowchart 1, Flowchart 2, Flowchart 3, Flowchart 4, Flowchart 5, Flowchart 6
4) **Report:** PDF Link
5) **Slides:** PDF Link
6) **Video:** YouTube Link

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Thank You

Questions?

Email: lausuhui@graduate.utm.my

PRESENTATION

2025