MECS1023 ADSA Assignment

"BST-Based Solution of Pharmacy Inventory Control System"

Group Name	**: Group 1**	
Group Members	**: Lau Su Hui (Abby)**	**MEC245045**
	Nurul 'Aini Binti Hamdan	**MEC245046**
	Winnie Ngu	**MEC245065**
Semester	**: 20252026-1**	
Title	**: Task 2 - Solution 1 Code**	

PRESENTATION

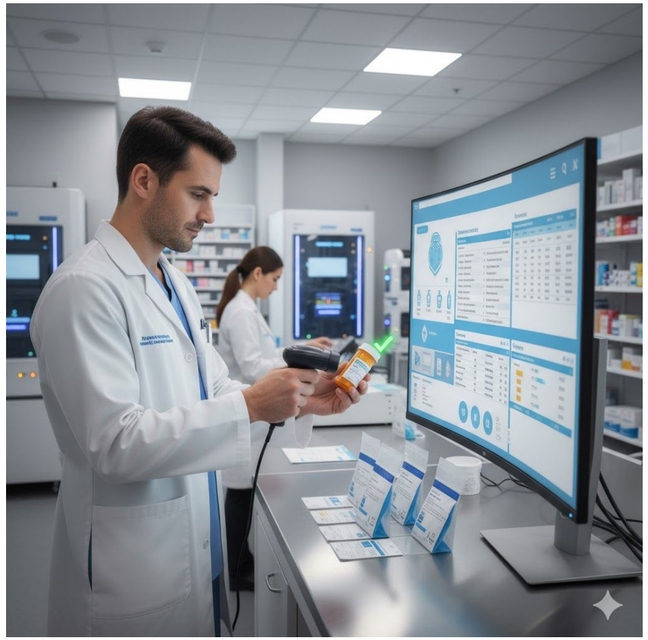
YEAR 2025

BST-based Medicine Management System

By: Abby

System Overview

Objective: To build an efficient console-based system for managing medicine batches using a Binary Search Tree (BST) implemented in C++.



Key Features:

- **Add:** Captures Batch ID, Expiry, Location, & Quantity.
- **Search:** Instant lookup via Batch ID.
- **Display:** Auto-sorted alphabetical inventory in a table view.
- **Delete:** Removes expired or empty batches.

Why BST?

- **Efficiency:** Faster Search/Insert (O(log n)) than linear arrays (O(n)).
- **Scalability:** Dynamic memory allocation (pointers) allows unlimited inventory growth.

Data Structure Design

💻 The Data Object (struct Medicine):

- **Key: batchID** (String) - Used for sorting.
- **Details: medicineName**, **quantity**.
- **Logistics: expiryDate**, **location** (e.g., "Shelf A01").

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

Data Structure Design

The Core Node (struct Node): The building block of the tree contains:

- **Data:** A **Medicine** object.
- **Left Pointer:** Points to a node with a smaller alphabetical Batch ID.
- **Right Pointer:** Points to a node with a larger alphabetical Batch ID.

Sorting Logic:

- The tree is organised strictly by **Batch ID**.
- Left Child < Parent/Root < Right Child.
- For example:
  1. Parent/Root: "B005"
  2. Left Child: "B001" (Smaller suffix)
  3. Right Child: "B009" (Larger suffix)

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

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

Key Functions - Insert

💻 Insert Function (insertInternal):

- **Logic:** Recursive comparison.
- **Flow:** 1. If the tree is empty, create the Root.
  2. If the new Batch ID is **smaller** than the current node, recurse **Left**.
  3. If the new Batch ID is **larger** than the current node, recurse **Right**.
  4. Duplicate Batch IDs are flagged as errors to prevent data corruption.

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

Key Functions - Search

💻 Search Function (searchInternal):

- **Logic:** Binary Search principle.
- **Flow:** Checks if the current node matches the Batch ID. If not, it eliminates half the tree instantly by choosing to go only Left or only Right.

```
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);
    }
}
```

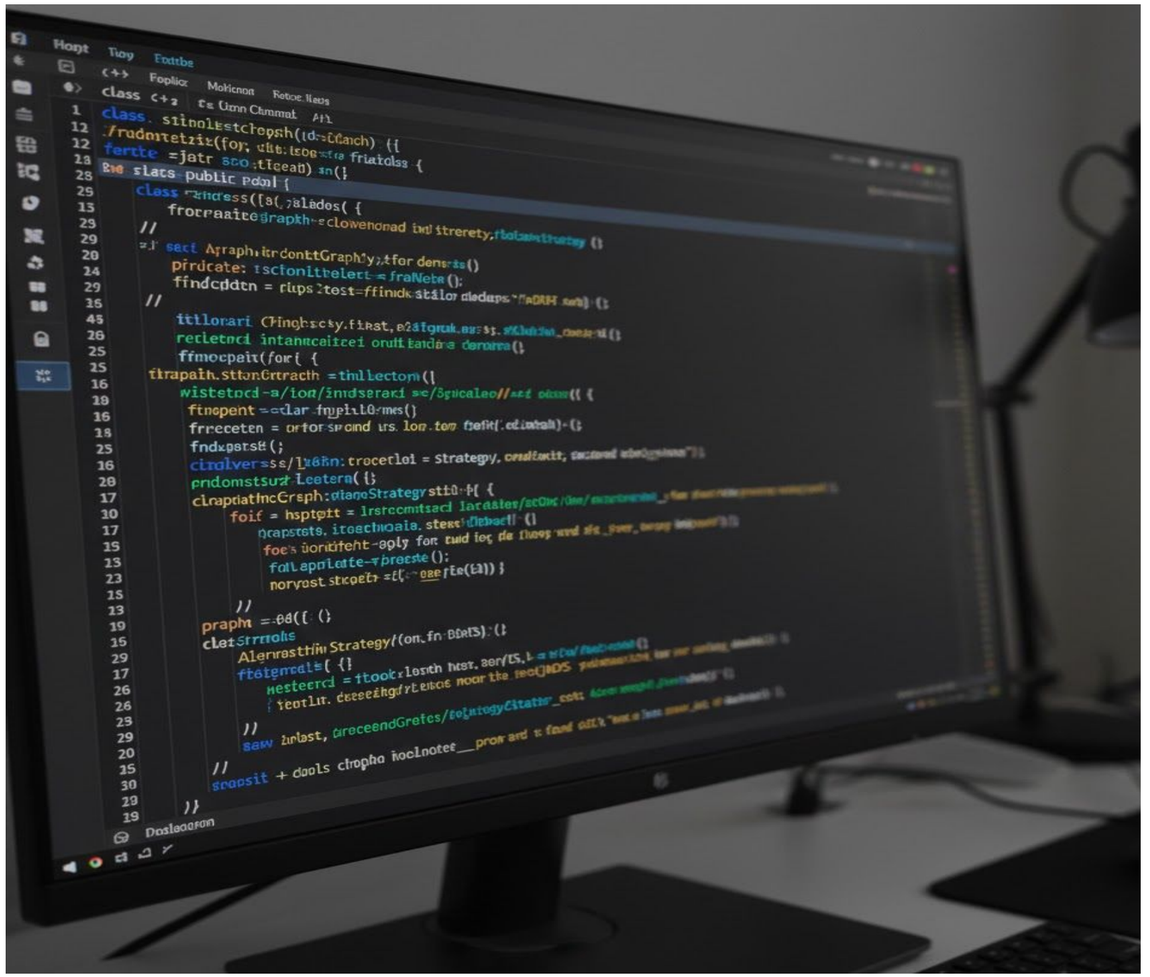
Key Functions - Delete

💻 **Logic: removeInternal** handles 3 cases to keep the tree sorted.

- **Leaf Node:** Delete directly (no children).
- **One Child:** Bypass node; link Parent → Child.
- **Two Children:** Replace with **In-Order Successor** (Right Min), then delete the successor.

```
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;
}
```

Demo of Working Functions



Improvements for Next Draft

- **Expiry Warning:** Utilise the expiryDate field to list batches that are expiring in < 30 days on a dashboard.
- **File Persistence:** Implement saveToFile() and loadFromFile() so data is not lost when the console is closed.
- **Category Grouping:** Sort medicines by therapeutic class (e.g., Antibiotics).

BST-based Pharmacy Stock System

By: Winnie

System Overview

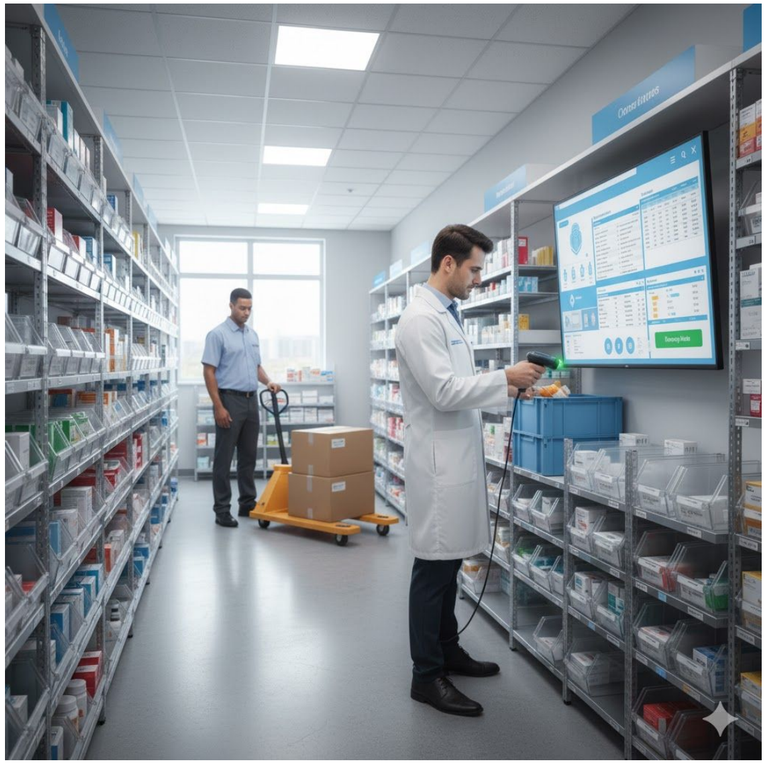
Objective: To build an efficient console application for managing stock in pharmacy.

Key Features:

- Add new medicines (ID, Name, Qty, Price).
- Search instantly by ID.
- Update stock details.
- Delete records dynamically.
- Display sorted inventory.

Why BST?

- Faster Search/Insert than Arrays (O(log n) vs O(n)).
- Dynamic memory (no fixed size limit).



Data Structure Design

Structure `Medicine` contains:

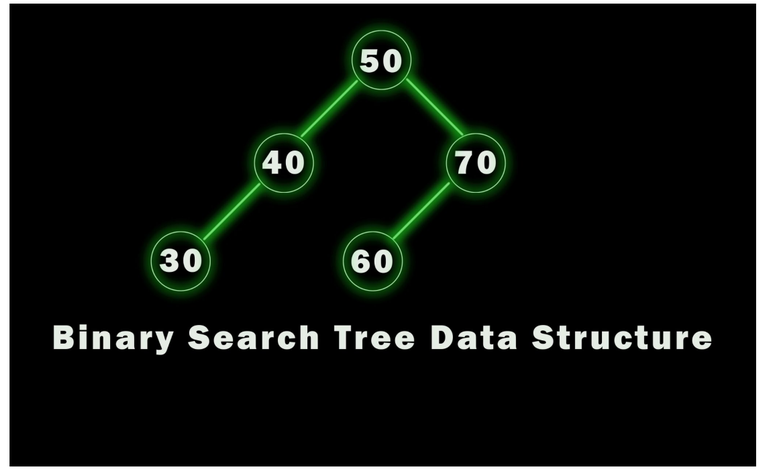
- Medicine unique ID, name
- Quantity
- Price

```
struct Medicine {
    string id;
    string name;
    int quantity{};
    double price{};
};
```

Data Structure Design

The Core Node:

- Structure Node contains:
  - Medicine data (Struct)
  - Node* left (Pointer to smaller IDs)
  - Node* right (Pointer to larger IDs)



Sorting Logic:

- The tree is organized by **Medicine ID** (String comparison).
- Left Child < Parent < Right Child.

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

    Node(const Medicine& m) : data(m), left(nullptr), right(nullptr) {}
};
```

Key Functions - Insert & Search

Insert: Recursive logic. Smaller IDs go Left, Larger go Right. Handles duplicates by updating.

Search: Recursive traversal based on ID comparison.

```
Node* insert(Node* node, const Medicine& med) {
    if (node == nullptr) {
        return new Node(med);
    }
    if (med.id < node->data.id) {
        node->left = insert(node->left, med);
    } else if (med.id > node->data.id) {
        node->right = insert(node->right, med);
    } else {
        //Update record if same medicine exists
        cout << "Medicine with ID " << med.id
             << " already exists. Updating record...\n";
        node->data.name = med.name;
        node->data.quantity = med.quantity;
        node->data.price = med.price;
    }
    return node;
}
```

```
//Search
Node* search(Node* node, const string& id) const {
    if (node == nullptr) return nullptr;
    if (id == node->data.id) return node;
    if (id < node->data.id) return search(node->left, id);
    return search(node->right, id);
}
```

Key Functions - Delete

Case 1: Leaf Node (Delete).

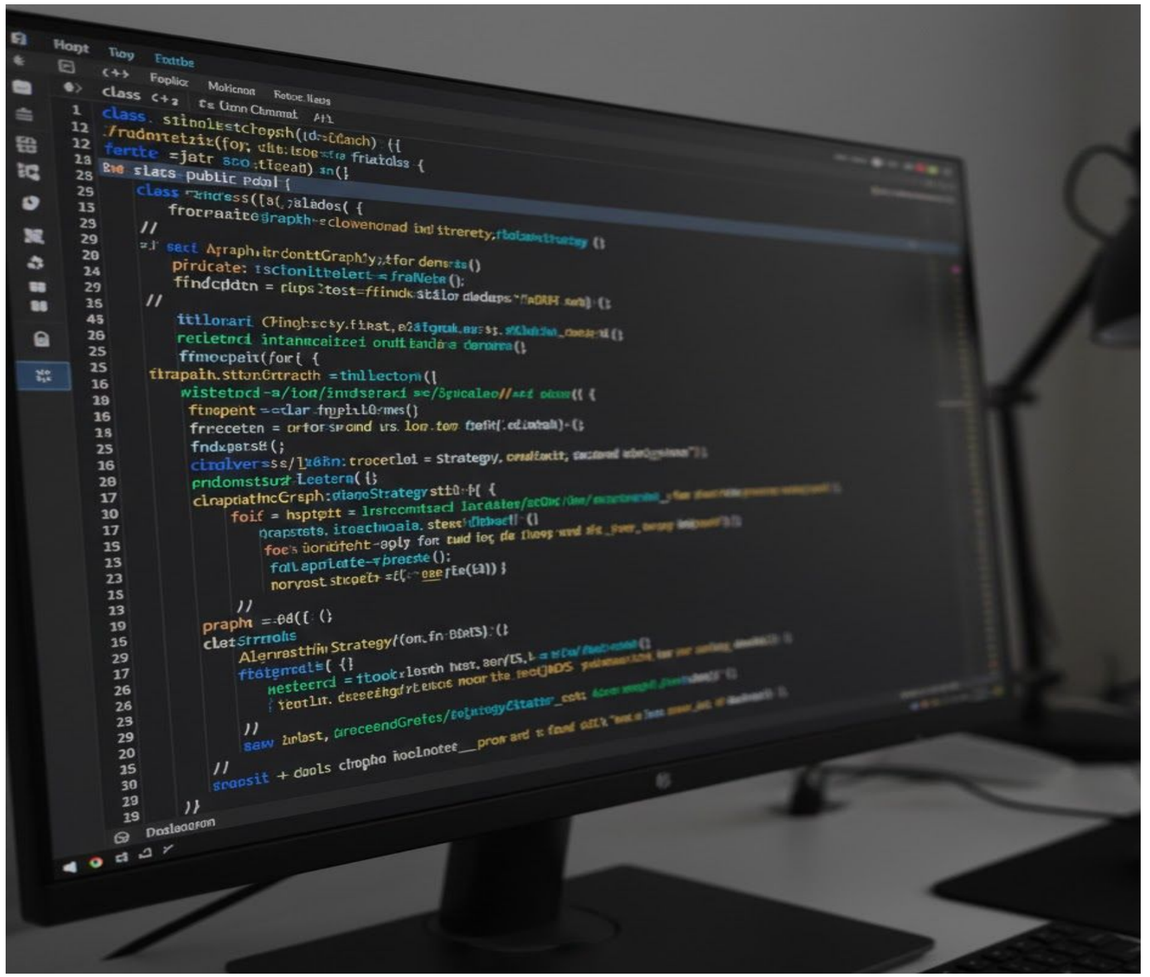
Case 2: One Child (Bypass/Link Parent to Child).

Case 3: Two Children (Replace with **In-Order Successor**).

```
//Remove
Node* remove(Node* node, const string& id) {
    if (node == nullptr) return nullptr;

    if (id < node->data.id) {
        node->left = remove(node->left, id);
    } else if (id > node->data.id) {
        node->right = remove(node->right, id);
    } else {
        // Found node to delete
        if (node->left == nullptr && node->right == nullptr) {
            // Leaf node
            delete node;
            return nullptr;
        } else if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        } else {
            // Two children- replace with inorder successor
            Node* succ = findMin(node->right);
            node->data = succ->data;
            node->right = remove(node->right, succ->data.id);
        }
    }
```

Demo of Working Functions



Improvements for Next Draft

- **Low Stock Alert:** Identifies medicines falling below a specific threshold (e.g., less than 10 units).
- **Range Search (ID):** Finds all medicines within a specific ID range (e.g., from `M002` to `M008`).
- **Total Asset Valuation:** Calculates the total monetary value of the entire stock (Price × Quantity).

BST-based Supplier and Vendor Management System

By: Aini

System Overview

Objective: To centralize and streamline the management of supplier relationships to maximize value, minimize risks, and ensure efficient procurement and high-quality supply chain performance

Key Features:

- Add new supplier/vendor
- Search supplier/vendor by ID
- Delete supplier/vendor by ID
- Update supplier/vendor information
- Display supplier/vendor sorted list

```
class Supplier:
    def __init__(self, supplier_id
        self.supplier_id = supplie
        self.supplier_name = suppl
        self.supplier_contact = su
        self.supplier_product = su
        self.supplier_rating = sup
```

Why BST?

- Provide efficient average-case insertion and search operations with time complexity of O(log n).
- Returns data sorted by keys without needing extra sorting.
- Enables intuitive hierarchical data organization and recursive operations for tasks

Data Structure Design

Class `Supplier` contains:

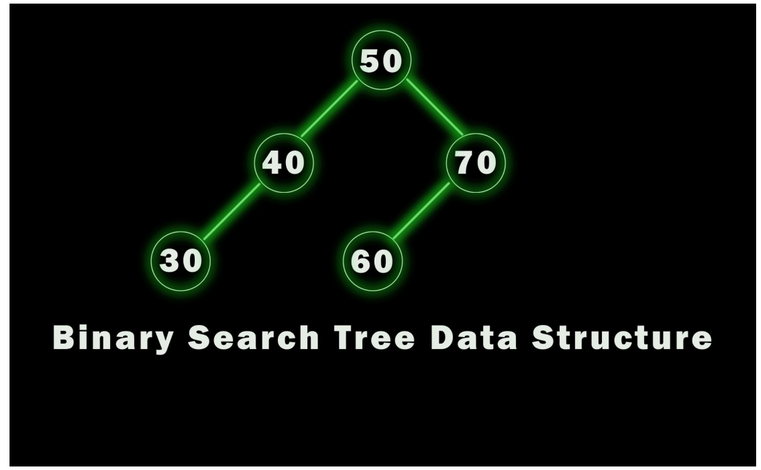
- Supplier ID, name, contact
- Type of products supplied
- Rating of supplier

```
class Supplier:
    def __init__(self, supplier_id
        self.supplier_id = supplie
        self.supplier_name = suppl
        self.supplier_contact = su
        self.supplier_product = su
        self.supplier_rating = sup
```

Data Structure Design

The Core Node:

- Class BSTNode contains:
  - self.supplier (store supplier object)
  - self.left (Pointer to smaller IDs)
  - self.right (Pointer to larger IDs)



Sorting Logic:

- The tree is organized by **Supplier ID** (Numeric comparison).

```
class BSTNode:
    def __init__(self, supplier):
        self.supplier = supplier  # Store supplier object
        self.left = None  # Pointer to left child (smaller ID)
        self.right = None  # Pointer to right child (larger ID)
```

Key Functions - Insert & Search

Insert: Recursive logic. Smaller IDs go Left, Larger go Right.

Search: Recursive traversal based on ID comparison.

```python
# Insert new supplier into BST based on ID
def insert(self, supplier):
    if supplier.supplier_id < self.supplier.supplier_id:
        if self.left is None:
            self.left = BSTNode(supplier)
        else:
            self.left.insert(supplier)
    else:
        if self.right is None:
            self.right = BSTNode(supplier)
        else:
            self.right.insert(supplier)
```

```python
# Search for supplier by ID in BST
def search(self, supplier_id):
    if supplier_id == self.supplier.supplier_id:
        return self.supplier
    elif supplier_id < self.supplier.supplier_id and self.left:
        return self.left.search(supplier_id)
    elif supplier_id > self.supplier.supplier_id and self.right:
        return self.right.search(supplier_id)
    else:
        return None
```

Key Functions - Delete

Case 1: Node with No Children (Leaf Node)

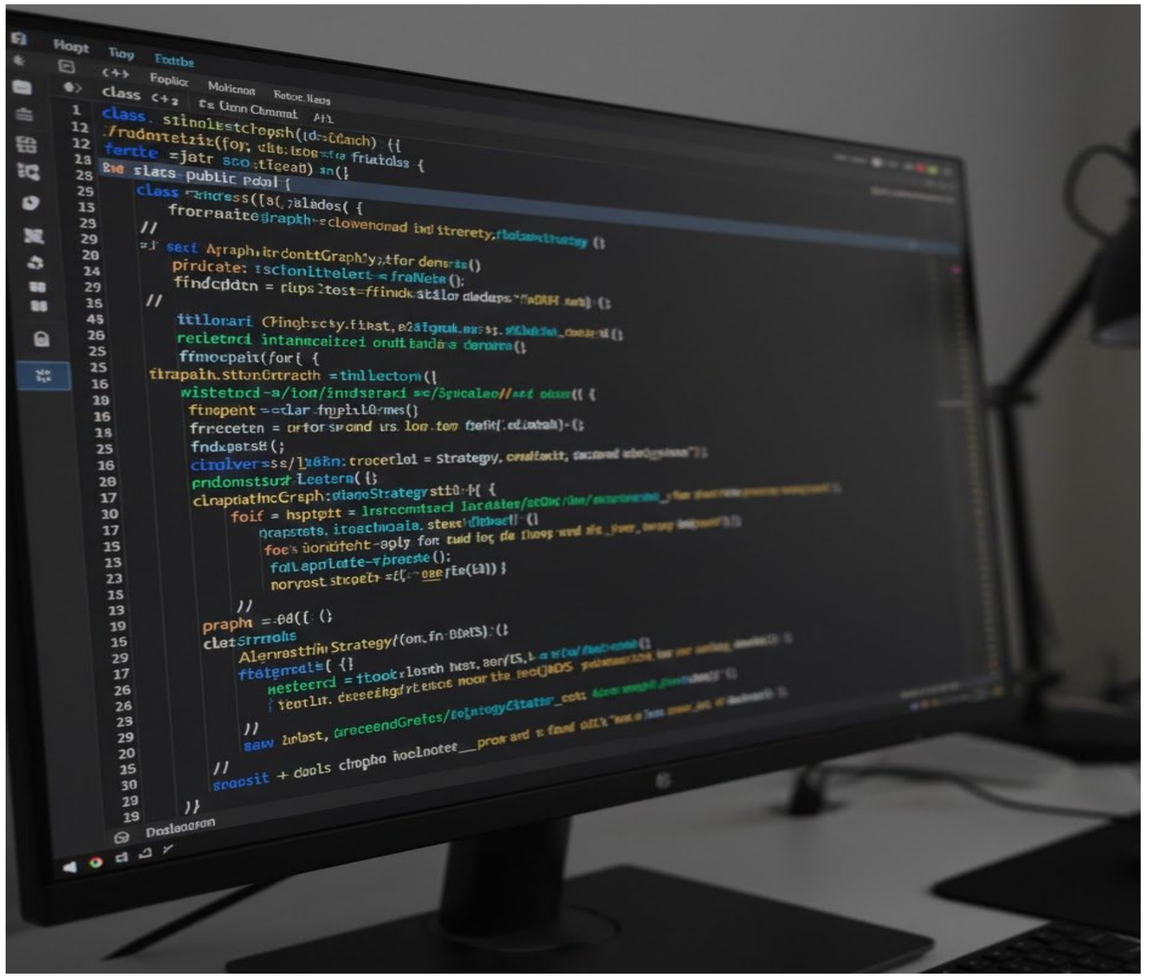
Case 2: Node with One Child (bypass node and connect the parent directly to grandchild)

Case 3: Node with Two Children (in-order successor using smallest node in the right subtree.

```
# Delete supplier from BST
def delete(self, supplier_id):
    if supplier_id < self.supplier.supplier_id:
        if self.left:
            self.left = self.left.delete(supplier_id)
    elif supplier_id > self.supplier.supplier_id:
        if self.right:
            self.right = self.right.delete(supplier_id)
    else:
        # Node to be deleted found
        if self.left is None:
            return self.right
        elif self.right is None:
            return self.left

        # Node with two children: get inorder successor
        temp = self.right
        while temp.left:
            temp = temp.left
        self.supplier = temp.supplier
        self.right = self.right.delete(temp.supplier.supplier_id)
```

Demo of Working Functions



Improvements for Next Draft

- **Supplier Category:** Add product categories for better organization.
- **Audit Trail:** Track all changes made to supplier records.
- **Search Enhancements:** Add search by name, product, contact or rating range.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Thank You

Questions?

PRESENTATION

2025