جامـــــعة المـــــستـقبـل
AL MUSTAQBAL UNIVERSITY

قســـم الأمـــــن الـــــــــسيبرانـــــــــي

DEPARTMENT OF CYBER SECURITY

SUBJECT:

SEARCHING AND SORTING ALGORITHMS

CLASS:

SECOND

LECTURER:

M.SC.MUNTATHERALMUSSAWEE

LECTURE: (9-10)

BINARY SEARCH TREE: INSERTION & DELETION

Binary Search Tree (BST) Insertion

```cpp
#include <iostream>
using namespace std;

// Node definition
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert into BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value); // base case
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
// insert in left subtree
    } else if (value > root->data) {
        root->right = insert(root->right, value);
// insert in right subtree
    }
    // if value == root->data, we do nothing (no
duplicates)

    return root;
}

// Inorder traversal (sorted order)
void inorderTraversal(Node* root) {
```

```
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert values into BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    // Display BST in inorder (sorted)
    cout << "Inorder Traversal: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Code Explanation

✅ 1. Node Structure

```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

This defines a node in the tree:

- `data`: holds the value.
- `left`: pointer to the left child.
- `right`: pointer to the right child.

🟦 2. Create a Node

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}
```

- Creates a new node with the given value.
- Initializes its left and right children as `nullptr`.

🌳 3. Insert a Value in BST

```
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}
```

- If the tree is empty → the new node becomes the root.
- Otherwise:
  - If the value is **less than** current node → go **left**.
  - If the value is **greater than** current node → go **right**.
  - If it's equal → we skip (no duplicates).
- Returns the (possibly updated) root.

🔍 4. Inorder Traversal (Print the Tree)

```
void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}
```

- **Inorder traversal** of BST prints values in **sorted order**:
  - Left subtree → Root → Right subtree

📝 5. Main Function

```
int main() {
    Node* root = nullptr;

    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    cout << "Inorder Traversal: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

- Starts with an empty tree (`root = nullptr`).
- Inserts values into the BST.
- Prints the tree in sorted order using `inorderTraversal()`.

🍵 Output:

```
Inorder Traversal: 20 30 40 50 60 70 80
```

Binary Search Tree (BST) Deletion

```cpp
#include <iostream>
using namespace std;

// Node definition
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert a value in BST
Node* insert(Node* root, int value) {
    if (root == nullptr)
        return createNode(value);

    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);

    return root;
}

// Find minimum value node in a BST (used for
deletion)
Node* findMin(Node* node) {
    while (node && node->left != nullptr)
        node = node->left;
    return node;
}
```

```
// Delete a node from BST
Node* deleteNode(Node* root, int value) {
    if (root == nullptr)
        return root;

    if (value < root->data) {
        root->left = deleteNode(root->left,
value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right,
value);
    } else {
        // Node to be deleted found
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right,
temp->data);
    }

    return root;
}

// Inorder traversal
void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Main function
```

```
int main() {
    Node* root = nullptr;

    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    cout << "Inorder before deletion: ";
    inorderTraversal(root);
    cout << endl;

    // Delete some nodes
    root = deleteNode(root, 20);  // Leaf
    root = deleteNode(root, 30);  // One child
    root = deleteNode(root, 50);  // Two children

    cout << "Inorder after deletion: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Code Explanation

1. Node Definition

```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

This defines the structure of a tree node:

- `data`: stores the node's value.

- `left` and `right`: pointers to child nodes.

🔷 2. Create a New Node

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}
```

Allocates memory for a new node and returns it.

🔷 3. Insertion into BST

```
Node* insert(Node* root, int value)
```

Inserts a value by:

- Going **left** if value < current node.
- Going **right** if value > current node.

🔷 4. Find Minimum Node

```
Node* findMin(Node* node) {
    while (node && node->left != nullptr)
        node = node->left;
    return node;
}
```

Used in deletion when the node has **two children**. It finds the **smallest value in the right subtree** (inorder successor).

🔹 5. Deletion Function

```
Node* deleteNode(Node* root, int value)
```

➤ *Case 1: Node Not Found*

```
if (root == nullptr)
    return root;
```

➤ *Case 2: Search for the Node*

```
if (value < root->data) ...
else if (value > root->data) ...
```

➤ *Case 3: Node Found*

```
if (root->left == nullptr) ...
else if (root->right == nullptr) ...
```

- If left or right is `nullptr`, return the non-null child.

➤ *Case 4: Node Has Two Children*

```
Node* temp = findMin(root->right);
root->data = temp->data;
root->right = deleteNode(root->right, temp-
>data);
```

- Find the inorder successor.
- Replace current node's value with successor's value.
- Delete the successor.

🔹 6. Inorder Traversal

```
void inorderTraversal(Node* root)
```

Prints tree values in **sorted order**:
Left → Root → Right

🔹 7. Main Function

```
int main() {
    ...
    root = deleteNode(root, 20); // Leaf
    root = deleteNode(root, 30); // One child
```

```
    root = deleteNode(root, 50); // Two children
}
```

Test cases that cover all 3 deletion scenarios.

✅ Sample Tree Structure

Before:

markdown

```
        50
       /  \
     30    70
    / \    / \
  20  40  60  80
```

After Deletions:

- 20 (leaf) is deleted
- 30 (one child) is deleted
- 50 (two children) is replaced by 60

Final Tree:

markdown

```
        60
       /  \
     40    70
             \
             80
```

Inorder Traversal Output:

yaml

```
Inorder before deletion: 20 30 40 50 60 70 80
Inorder after deletion: 40 60 70 80
```