ase Study: Implementing a Binary Search Tree for a University Student Databas

1. Introduction

This case study explores the development of a Binary Search Tree (BST) to manage student records in a university's database. Efficient data organization is critical in large databases, where operations such as insertion, deletion, and searching must be fast and scalable. Binary Search Trees provide an optimal solution for handling ordered data, allowing logarithmic time complexity for operations, making them suitable for the university's requirements.

2. Background and Requirements

The university requires a system to manage student records, each identified by a unique student ID. Operations required include: insertion of new records, searching by ID, and deletion, ensuring data integrity and quick access. A Binary Search Tree (BST) is chosen for its efficient performance in managing ordered keys.

3. Design of the Binary Search Tree (BST)

A BST is structured with nodes, each containing a key (student ID) and pointers to left and right children. The left child has a smaller key, and the right child has a larger key, facilitating efficient operations. Key operations include:

- **Insertion**: Insert nodes based on ID, moving left if the ID is smaller, right if larger.
- **Search**: Start at the root, moving left or right based on comparisons with the target ID.
- **Deletion**: Handle cases based on the node having zero, one, or two children, using in-order successor replacement when necessary to maintain structure.

4. Implementation (in C)

4.1 Insertion Code

```
// Structure of a BST Node
struct Node {
    int id;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int id) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->id = id;
    node->left = node->right = NULL;
    return node;
}

// Function to insert a node in the BST
struct Node* insert(struct Node* node, int id) {
    if (node == NULL) return newNode(id);

    if (id < node->id)
        node->left = insert(node->left, id);
    else if (id > node->id)
        node->right = insert(node->right, id);

    return node;
}
```

4.2 Search Code

```
// Function to search a given ID in BST
struct Node* search(struct Node* root, int id) {
    if (root == NULL || root->id == id)
        return root;

    if (id < root->id)
        return search(root->left, id);

    return search(root->right, id);
}
```

4.3 Deletion Code

```
// Function to find the minimum value node
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a node in BST
struct Node* deleteNode(struct Node* root, int id) {
    if (root == NULL) return root;

    if (id < root->id)
        root->left = deleteNode(root->left, id);
```

```
    else if (id > root->id)
        root->right = deleteNode(root->right, id);
    else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = minValueNode(root->right);
        root->id = temp->id;
        root->right = deleteNode(root->right, temp->id);
    }
    return root;
}
```

5. Time Complexity Analysis

Each operation has different time complexities depending on tree balance:

- **Insertion**: O(log n) in balanced trees; O(n) in the worst case (unbalanced).
- **Search**: O(log n) average; O(n) if unbalanced.
- **Deletion**: Same complexity as search, with additional restructuring as needed.

Balancing ensures logarithmic time for most operations, ideal for large student databases.

6. Results and Discussion

A BST enables efficient student record management through optimized search, insertion, and deletion. Limitations include potential unbalance, which increases operation time. Consider self-balancing BSTs (AVL, Red-Black) if data is sequentially ordered.

7. Conclusion

This case study demonstrates how a BST fulfills the university's requirements for an efficient, scalable student records system. It provides consistent performance, with further improvements possible through self-balancing techniques.