

# Assignment 1 - Set C: Theoretical Questions and Answers

---

## Binary Search Tree - Advanced Operations

---

### Question 1

---

**Write a C program which uses Binary search tree library and implements following two functions:**

- a) `int sumodd(T)` – returns sum of all odd numbers from BST
- b) `int sumeven(T)` – returns sum of all even numbers from BST
- c) `mirror(T)` – converts given tree into its mirror image.

### Answer

### a) Sum of Odd Numbers Function

```
int sumodd(Node *root) {  
    if (root == NULL) {  
        return 0;  
    }  
    int sum = (root->data % 2 != 0) ? root->data : 0;  
    return sum + sumodd(root->left) + sumodd(root->right);  
}
```

This function recursively traverses the tree and adds up all odd numbers.

### b) Sum of Even Numbers Function

```
int sumeven(Node *root) {  
    if (root == NULL) {  
        return 0;  
    }  
    int sum = (root->data % 2 == 0) ? root->data : 0;  
    return sum + sumeven(root->left) + sumeven(root->right);  
}
```

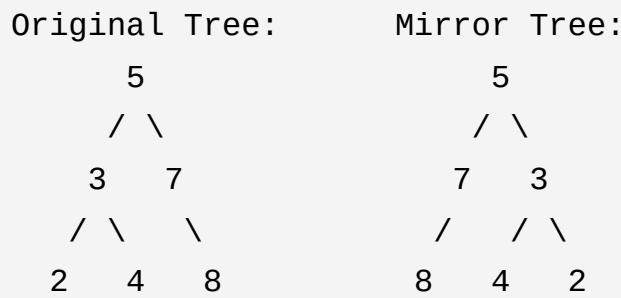
This function recursively traverses the tree and adds up all even numbers.

### c) Mirror Function

```
void mirror(Node *root) {  
    if (root == NULL) {  
        return;  
    }  
    Node *temp = root->left;  
    root->left = root->right;  
    root->right = temp;  
    mirror(root->left);  
    mirror(root->right);  
}
```

This function converts the tree into its mirror image by swapping left and right subtrees recursively.

### Example



## Question 2

---

**Write a function to delete an element from BST.**

### Answer

```
Node* findMin(Node *root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

Node* deleteNode(Node *root, int key) {
    if (root == NULL) {
        return NULL;
    }

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            Node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node *temp = root->left;
            free(root);
            return temp;
        }
    }
}
```

```
        return temp;
    }

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

## Explanation

The deletion algorithm handles three cases:

### 1. Node with no children (Leaf node):

Simply remove the node and return NULL.

### 2. Node with one child:

Remove the node and replace it with its child.

### 3. Node with two children:

Find the inorder successor (smallest node in right subtree), copy its value to the current node, and delete the successor.

**Time Complexity:** O(h) where h is the height of the tree

**Space Complexity:** O(h) due to recursion stack

## Question 3

**What modifications are required in search function to count the number of comparisons required?**

### Answer

To count the number of comparisons in the search function, we need to add a counter variable that increments with each comparison made during the search operation.

### Modified Search Function

```
int searchWithCount(Node *root, int key, int *comparisons) {  
    if (root == NULL) {  
        return 0; // Element not found  
    }  
  
    (*comparisons)++; // Increment comparison count  
  
    if (root->data == key) {  
        return 1; // Element found  
    }  
}
```

```
(*comparisons)++; // Increment for the next comparison

if (key < root->data) {
    return searchWithCount(root->left, key, comparisons);
} else {
    return searchWithCount(root->right, key, comparisons);
}
}
```

## Usage Example

```
int main() {
    Node *root = NULL;
    int comparisons = 0;
    int key = 10;

    // Build tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);

    // Search with comparison count
    if (searchWithCount(root, key, &comparisons)) {
        printf("Element %d found after %d comparisons\n", key, comparisons);
    } else {
        printf("Element %d not found after %d comparisons\n", key, comparisons);
    }
}
```

```
    return 0;  
}
```

## Key Modifications

1. Added a pointer parameter `comparisons` to track the count
2. Increment the counter before each comparison operation
3. Pass the counter by reference using pointer
4. Return the counter value to display number of comparisons

## Analysis

- **Best Case:**  $O(1)$  - Element at root (1 comparison)
- **Average Case:**  $O(\log n)$  - Balanced tree
- **Worst Case:**  $O(n)$  - Skewed tree (all nodes in one direction)