NAME:IBRAHIM SAID ABDALLA

REG NO:BCS/18/23/020/TZ

GIT USERNAME:Astro-67

REPO:https://github.com/Astro-67/Cyclesheet1Assigniment.git

Operation 1: Search for a Key in the BST (10 Marks) Write a C function `search_bst(Node *root, int key)` that searches for a key in a binary search tree. The function should return `true` if the key is found and `false` otherwise. In addition to this basic search, you should also:

- Track the path traversed during the search and print it as a list of node values.
- Return the number of nodes visited during the search.

ANSWER:

- First I define binary search tree node with data to contain a value and pointer to the left child and the right pointer to the right node.
- Then Function which contain 4 argument root the starting node of the tree the key which is searched for path to array to record node traversed and count to count number of node visited
- Also while loop for traversal the tree unless root is not equal to one which count ,record path and transverse to the left and right of the tree

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


bool search_bst(struct node *root, int key, int *path, int *count) {
   *count = 0;
   while (root != NULL) {
       *count++;
      path[*count] = root->data;
      if (key == root->data) {
         return true;
      } else if (key < root->data) {
         root = root->left;
      } else {
         root = root->right;
```

```
        }
    }
    return false;
}
```

Operation 2: Find the Height and Diameter of the Binary Tree (10 Marks) Write a C function `find_height_and_diameter(Node *root, int *diameter)` to compute the height of the binary tree and the diameter of the tree.

- The height of a tree is the length of the longest path from the root to a leaf.

- The diameter of a tree is the length of the longest path between any two nodes in the tree, which may or may not pass through the root.

ANSWER:

- First I create a max function to calculate the max and return the max value
- find height and diameter and height calculate function if root is null then it will return 0
- left height for recursion of left diameter and right height for recursion of right diameter which I store in variable and to call max function to calculate the diameter and return max.

```
int max(int a, int b) {
if (a>b){
   return a;
} else{
   return b;
}
}
int find_height_and_diameter(struct node *root, int *diameter) {
    if (root == NULL) return 0;

    int leftHeight = find_height_and_diameter(root->left, diameter);
    int rightHeight = find_height_and_diameter(root->right, diameter);

    *diameter = max(*diameter, leftHeight + rightHeight);

    return max(leftHeight, rightHeight);
}
```

Operation 3: Iterative Pre-order Traversal with Subtree Sum (10 Marks) Write a C function `pre_order_iterative_with_sum(Node *root, int *sum)` to perform an iterative pre-order traversal and calculate the sum of all node values during the traversal.

- The traversal should use a stack.
- While performing the traversal, calculate the sum of all node values visited.

ANSWER:

- first I define stack to for traversal of tree the push funciton to push an item and pop function to pop an item and pre order function for performing pre order operation

```
#define SIZE 100
struct Stack {
    struct node* array[SIZE];
    int top;
}s;

void push(struct Stack* stack, struct node* item) {
    if (stack->top < SIZE - 1) {
        stack->array[++stack->top] = item;
    }
}

struct node* pop(struct Stack* stack) {
    return (stack->top >= 0) ? stack->array[stack->top--] : NULL;
}

int pre_order_iterative_with_sum(struct node *root) {
    if (root == NULL) return 0;
    s.top = -1;
    push(s, root);
    int sum = 0;

    while (s.top != -1) {
        struct node *current = pop(s);
        sum += current->data;

        if (current->right) push(s, current->right);
        if (current->left) push(s, current->left);
    }
    return sum;
}
```

Operation 4: Iterative In-order Traversal with Balanced Tree Check (10 Marks) Write a C function `in_order_iterative_with_balance_check(Node *root)` to perform an iterative in-order traversal while checking whether the tree is balanced.

- A balanced tree is defined as one where the heights of the left and right subtrees of any node differ by at most 1.
- Use two stacks: one for the traversal and one for tracking balance checks.

ANSWER:

```
bool is_balanced(struct node *root) {
    if (root == NULL) return true;
    int leftHeight = find_height_and_diameter(root->left);
    int rightHeight = find_height_and_diameter(root->right);
    if (left > right + 1 || right > left + 1) return false;
    return true;
}
```

Operation 5: Post-order Traversal with Node Deletion Counter (10 Marks) Write a C function `post_order_iterative_with_deletion_count(Node *root, int *delete_count)` to perform an iterative post-order traversal while counting the number of leaf nodes deleted.

- Use two stacks to implement the iterative post-order traversal.
- Count how many leaf nodes are deleted during the traversal.

ANSWER:

```
int post_order_iterative_with_deletion_count(struct node *root) {
    if (root == NULL) return 0;
    struct Stack s1, s2;
    s1.top = -1;
    s2.top = -1;
    push(&s1, root);
    int delete_count = 0;
   while (s1.top != -1) {
        struct node *current = pop(&s1);
        push(&s2, current);
        if (current->left) push(&s1, current->left);
        if (current->right) push(&s1, current->right);
    }

    while (s2.top != -1) {
        struct node *node = pop(&s2);
        if (node->left == NULL && node->right == NULL) {
            delete_count++;
        }
        free(node);
    }
    return delete_count;
}
```

Operation 6: Delete a Node from the Binary Tree with Rotation (10 Marks) Write a C function `delete_node_with_rotation(Node *root, int key)` to delete a node from the binary search tree while performing tree rotations when necessary.

- You need to handle the three cases of deletion:

1. No children (leaf node).
2. One child.
3. Two children (use the in-order successor and possibly perform rotations).

ANSWER:

```
struct node* delete_node_with_rotation(struct node *root, int key) {
    if (root == NULL) return root;

    if (key < root->data) {
        root->left = delete_node_with_rotation(root->left, key);
    } else if (key > root->data) {
        root->right = delete_node_with_rotation(root->right, key);
    } 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 = findMin(root->right);
        root->data = temp->data;
        root->right = delete_node_with_rotation(root->right, temp->data);
    }
    return root;
}
```