Assignment 1

Dated 18th Nov, 2024

Problem Statement

Program in C to perform these on a binary tree:

1. Creation of a binary tree
2. Pre-order traversal
3. In-order traversal
4. Post-order traversal
5. Count no. of leaf nodes
6. Count no. of internal nodes
7. Find height of the tree

Algorithm

Input

Functions to get input from: `create`.

Output

Functions to provide output: `traverse_preorder`, `traverse_inorder`, `traverse_postorder`, `count_leaves`, `count_internal_nodes`, `height_tree`.

Data structure used

A binary tree data structure with left and right pointers to next nodes and a variable containing information.

Step 1: Start.

Step 2: Define the structure Node with the following members:

- A pointer lhs for the left child.
- A pointer rhs for the right child.
- An integer data to hold the node value.

Tree Creation (Function create):

Step 3: Display a message prompting the user to enter data for a node.

Step 4: Read the integer input and store it in variable x.

Step 5: Check if the input is invalid (not an integer). If true, display an error message and terminate the program.

Step 6: Check if the value of x is -1. If true, return NULL (base case of recursion).

Step 7: Allocate memory for a new Node.

Step 8: Check if memory allocation failed. If true, display an error message and terminate the program.

Step 9: Assign the value of x to the data member of the node.

Step 10: Display a message to create the left child of the node (Left child's of <x> =>).

Step 11: Recursively call the create function to construct the left subtree and assign it to the lhs member of the node.

Step 12: Display a message to create the right child of the node (Right child's of <x> =>).

Step 13: Recursively call the create function to construct the right subtree and assign it to the rhs member of the node.

Step 14: Return the created node.

[End of the create function]

Pre-order Traversal (Function traverse_preorder):

Step 15: Check if the current node p is NULL. If true, return.

Step 16: Display the data of the current node.

Step 17: Recursively traverse the left subtree by calling traverse_preorder(p->lhs).

Step 18: Recursively traverse the right subtree by calling traverse_preorder(p->rhs).

[End of the traverse_preorder function]

In-order Traversal (Function traverse_inorder):

Step 19: Check if the current node p is NULL. If true, return.

Step 20: Recursively traverse the left subtree by calling traverse_inorder(p->lhs).

Step 21: Display the data of the current node.

Step 22: Recursively traverse the right subtree by calling traverse_inorder(p->rhs).

[End of the traverse_inorder function]

Post-order Traversal (Function traverse_postorder):

Step 23: Check if the current node p is NULL. If true, return.

Step 24: Recursively traverse the left subtree by calling traverse_postorder(p->lhs).

Step 25: Recursively traverse the right subtree by calling traverse_postorder(p->rhs).

Step 26: Display the data of the current node.

[End of the traverse_postorder function]

Count Leaf Nodes (Function count_leaves):

Step 27: Check if the current node root is NULL. If true, return 0.

Step 28: Check if the current node is a leaf node (lhs and rhs are NULL). If true, return 1.

Step 29: Recursively count the leaf nodes in the left subtree and the right subtree.

Step 30: Return the sum of the leaf counts.

[End of the count_leaves function]

Count Internal Nodes (Function count_internal_nodes):

Step 31: Check if the current node root is NULL. If true, return 0.

Step 32: Check if the current node has at least one child (lhs or rhs is not NULL). If true, increment the count by 1.

Step 33: Recursively count the internal nodes in the left subtree and the right subtree.

Step 34: Return the sum of the counts.

[End of the count_internal_nodes function]

Calculate Tree Height (Function tree_height):

Step 35: Check if the current node root is NULL. If true, return 0.

Step 36: Recursively calculate the height of the left subtree and the right subtree.

Step 37: Add 1 to the maximum of the left and right subtree heights.

Step 38: Return the calculated height.

[End of the tree_height function]

Memory Deallocation (Function free_tree):

Step 39: Check if the current node root is NULL. If true, return.

Step 40: Recursively free the memory of the left subtree by calling free_tree(root->lhs).

Step 41: Recursively free memory of the right subtree by calling free_tree(root->rhs).

Step 42: Free the memory allocated for the current node.

[End of the free_tree function]

Main Program:

Step 43: Declare an integer option initialized to 0.

Step 44: Call the create function to construct the binary tree and assign its root to root.

Step 45: Repeat steps 46 to 54 in a loop until the user enters 0.

Step 46: Display the menu:

- Traversal: [1] Pre-order, [2] In-order, [3] Post-order
- Count: [4] Leaf nodes, [5] Internal nodes, [6] Height
- [0] Exit

Step 47: Read the user input into option.

Step 48: Check if the input is invalid. If true, display an error and break the loop.

Step 49: Check the value of option and perform the corresponding action:

- **Case 1**: Display Pre-order Traversal: and call traverse_preorder(root).
- **Case 2**: Display In-order Traversal: and call traverse_inorder(root).
- **Case 3**: Display Post-order Traversal: and call traverse_postorder(root).
- **Case 4**: Display the total number of leaf nodes by calling count_leaves(root).

- **Case 5**: Display the total number of internal nodes by calling count_internal_nodes(root) and adding 1 (for the root).
- **Case 6**: Display the height of the tree by calling tree_height(root) and subtracting 1.
- **Case 0**: Display Exiting... and exit the loop.
- **Default**: Display an error message prompting the user to choose a valid option.

  [End of the switch-case block in step 49]

Step 50: End the loop when the user selects the Exit option.

Step 51: Call free_tree(root) to free the memory allocated for the tree.

Step 52: Terminate the program.

[End of the main function]

Code

```
#include <stdio.h>
#include <stdlib.h>

// Binary Tree Node Definition
typedef struct BTree {
    struct BTree* lhs;
    struct BTree* rhs;
    int data;
} Node;

Node* create(void)
{
    int x = 0;

    printf("Data (-1 to quit): ");
    if (scanf("%d", &x) != 1) {
        fprintf(stderr, "error: Invalid input.\n");
        exit(1);
    }

    if (x == -1) {
        return NULL;
    }

    // Allocate memory for a new node
    Node* node = malloc(sizeof(Node));
    if (node == NULL) {
```

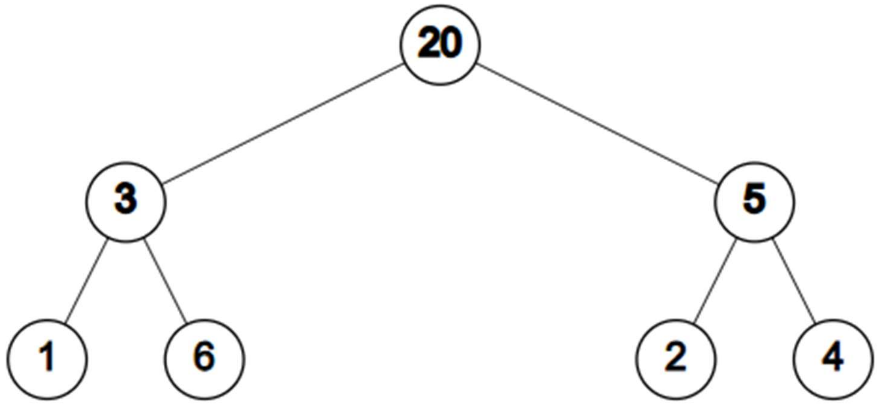
```
        fprintf(stderr, "error: malloc() failed.\n");
        exit(1);
    }

    node->data = x;

    // Recursively create left and right subtrees
    printf("Left child's of %d => ", x);
    node->lhs = create();

    printf("Right child's of %d => ", x);
    node->rhs = create();

    return node;
}

// Free the Allocated Memory for Tree
void free_tree(Node* root)
{
    if (root == NULL) {
        return;
    }

    free_tree(root->lhs);
    free_tree(root->rhs);
    free(root);
}

// Preorder Traversal
void traverse_preorder(Node* p)
{
    if (p == NULL) {
        return;
    }
    printf("%d ", p->data);
    traverse_preorder(p->lhs);
    traverse_preorder(p->rhs);
}

// Inorder Traversal
void traverse_inorder(Node* p)
{
    if (p == NULL) {
        return;
    }

    traverse_inorder(p->lhs);
    printf("%d ", p->data);
    traverse_inorder(p->rhs);
}
```

```c
// Postorder Traversal
void traverse_postorder(Node* p)
{
    if (p == NULL) {
        return;
    }

    traverse_postorder(p->lhs);
    traverse_postorder(p->rhs);
    printf("%d ", p->data);
}

int count_leaves(Node* root)
{
    if (root == NULL) {
        return 0;
    } else if (root->lhs == NULL && root->rhs == NULL) {
        return 1;
    }

    return count_leaves(root->lhs) + count_leaves(root->rhs);
}

int count_internal_nodes(Node* root)
{
    if (root == NULL) {
        return 0;
    } else if (root->lhs != NULL || root->rhs != NULL) {
        return 1;
    }

    return count_leaves(root->lhs) + count_leaves(root->rhs);
}

int tree_height(Node* root)
{
    if (root == NULL) {
        return 0;
    }

    int height_lhs = tree_height(root->lhs) + 1;
    int height_rhs = tree_height(root->rhs) + 1;

    return (height_lhs > height_rhs) ? height_lhs : height_rhs;
}

int main(void)
{
    int option = 0;
```

```
    Node* root = create();

    do {
        printf("Traversal: [1] Pre-order\t[2] In-order\t[3] Post-order\n");
        printf("Count: [4] Leaf nodes\t[5] Internal nodes\t[6] Height\t: ");

        // Input validation
        if (scanf("%d", &option) != 1) {
            fprintf(stderr, "error: Invalid input.\n");
            break;
        }

        switch (option) {
        case 1:
            printf("Pre-order Traversal: ");
            traverse_preorder(root);
            printf("\n");
            break;
        case 2:
            printf("In-order Traversal: ");
            traverse_inorder(root);
            printf("\n");
            break;
        case 3:
            printf("Post-order Traversal: ");
            traverse_postorder(root);
            printf("\n");
            break;
        case 4:
            printf("\nLeaf nodes: %d\n\n", count_leaves(root));
            break;
        case 5:
            printf("\nInternal nodes: %d\n\n", count_internal_nodes(root) + 1);
            break;
        case 6:
            printf("\nHeight of the tree: %d\n\n", tree_height(root) - 1);
            break;
        case 0:
            printf("Exiting...\n");
            break;
        default:
            fprintf(stderr, "error: Please choose a valid option (0-3).\n");
        }
    } while (option != 0);

    // A good practice: Free-up the allocated memory
    free_tree(root);

    return 0;
}
```

Output

Structure of a sample binary tree:



Input

```
C:\Windows\System32\cmd.e

C:\Users\rohan\Code>a.exe
Data (-1 to quit): 20
Left child's of 20 => Data (-1 to quit): 3
Left child's of 3 => Data (-1 to quit): 1
Left child's of 1 => Data (-1 to quit): -1
Right child's of 1 => Data (-1 to quit): -1
Right child's of 3 => Data (-1 to quit): 6
Left child's of 6 => Data (-1 to quit): -1
Right child's of 6 => Data (-1 to quit): -1
Right child's of 20 => Data (-1 to quit): 5
Left child's of 5 => Data (-1 to quit): 2
Left child's of 2 => Data (-1 to quit): -1
Right child's of 2 => Data (-1 to quit): -1
Right child's of 5 => Data (-1 to quit): 4
Left child's of 4 => Data (-1 to quit): -1
Right child's of 4 => Data (-1 to quit): -1
Traversal: [1] Pre-order        [2] In-order     [3] Post-order
Count: [4] Leaf nodes    [5] Internal nodes      [6] Height       :
```

Operations

```
Left child's of 4 => Data (-1 to quit): -1
Right child's of 4 => Data (-1 to quit): -1
Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 1
Pre-order Traversal: 20 3 1 6 5 2 4
Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 2
In-order Traversal: 1 3 6 20 2 5 4
Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 3
Post-order Traversal: 1 6 3 2 4 5 20
Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 4

Leaf nodes: 4

Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 5

Internal nodes: 2

Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 6

Height of the tree: 2

Traversal: [1] Pre-order        [2] In-order    [3] Post-order
Count: [4] Leaf nodes   [5] Internal nodes      [6] Height      : 0
Exiting...

C:\Users\rohan\Code>
```

Teacher's signature