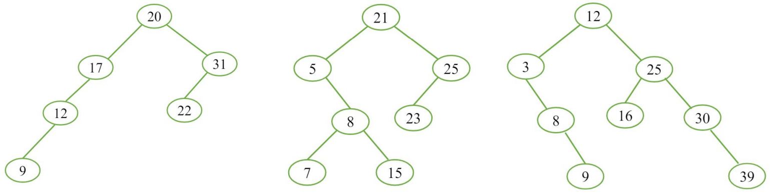
INTRODUCTION TO DATA STRUCTURES

Use iterative approach to perform an inorder traversal of a binary tree in C

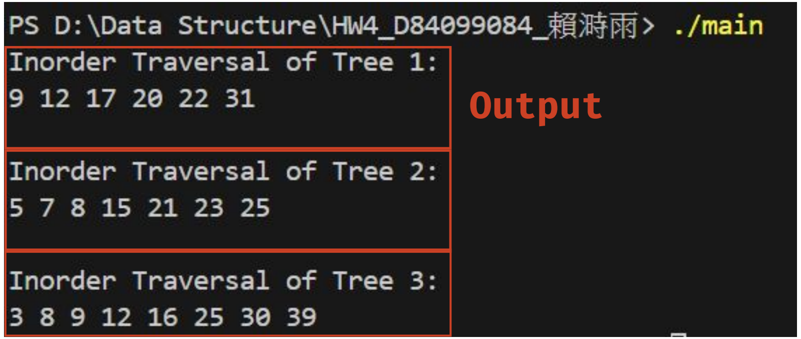
HW4_D84099084_賴溡雨

1.Evaluating postfix expressions

The C program is designed to perform operations on construct 3 binary trees, and use the stack to implement Inorder Traversal with the iterative method. It demonstrates an iterative approach to perform an inorder traversal of a binary tree without using recursion.

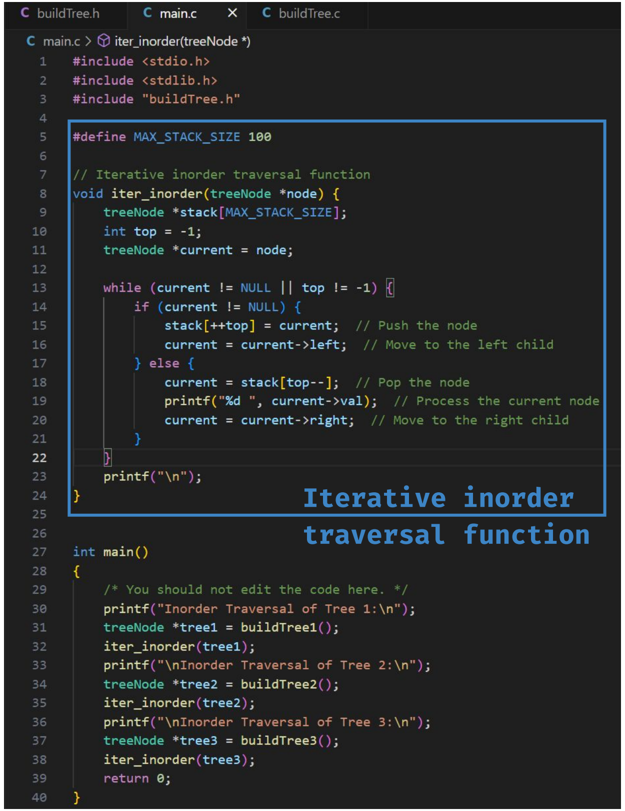


Tree 1 Tree 2 Tree 3



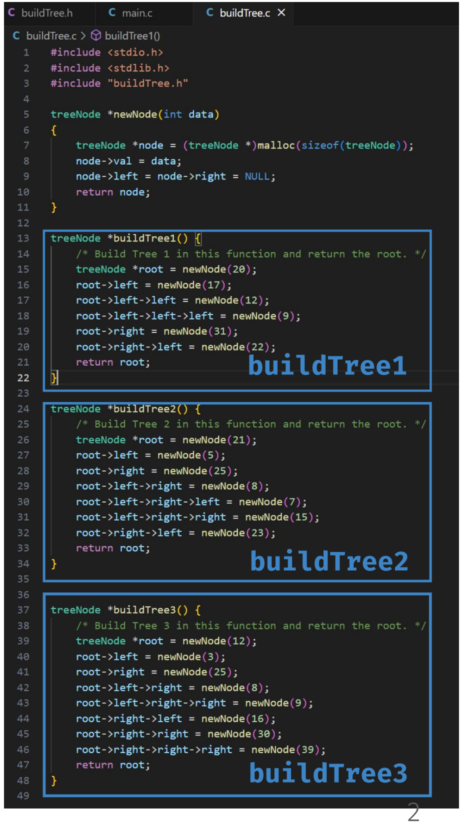
```
PS D:\Data Structure\HW4_D84099084_賴溡雨> ./main
Inorder Traversal of Tree 1:
9 12 17 20 22 31

Inorder Traversal of Tree 2:
5 7 8 15 21 23 25

Inorder Traversal of Tree 3:
3 8 9 12 16 25 30 39
```

Output



```c
#include <stdio.h>
#include <stdlib.h>
#include "buildTree.h"

#define MAX_STACK_SIZE 100

// Iterative inorder traversal function
void iter_inorder(treeNode *node) {
    treeNode *stack[MAX_STACK_SIZE];
    int top = -1;
    treeNode *current = node;

    while (current != NULL || top != -1) {
        if (current != NULL) {
            stack[++top] = current;  // Push the node
            current = current->left;  // Move to the left child
        } else {
            current = stack[top--];  // Pop the node
            printf("%d ", current->val);  // Process the current node
            current = current->right;  // Move to the right child
        }
    }
    printf("\n");
}

int main()
{
    /* You should not edit the code here. */
    printf("Inorder Traversal of Tree 1:\n");
    treeNode *tree1 = buildTree1();
    iter_inorder(tree1);
    printf("\nInorder Traversal of Tree 2:\n");
    treeNode *tree2 = buildTree2();
    iter_inorder(tree2);
    printf("\nInorder Traversal of Tree 3:\n");
    treeNode *tree3 = buildTree3();
    iter_inorder(tree3);
    return 0;
}
```

Iterative inorder traversal function



```c
#include <stdio.h>
#include <stdlib.h>
#include "buildTree.h"

treeNode *newNode(int data)
{
    treeNode *node = (treeNode *)malloc(sizeof(treeNode));
    node->val = data;
    node->left = node->right = NULL;
    return node;
}

treeNode *buildTree1() {
    /* Build Tree 1 in this function and return the root. */
    treeNode *root = newNode(20);
    root->left = newNode(17);
    root->left->left = newNode(12);
    root->left->left->left = newNode(9);
    root->right = newNode(31);
    root->right->left = newNode(22);
    return root;
}

treeNode *buildTree2() {
    /* Build Tree 2 in this function and return the root. */
    treeNode *root = newNode(21);
    root->left = newNode(5);
    root->right = newNode(25);
    root->left->right = newNode(8);
    root->left->right->left = newNode(7);
    root->left->right->right = newNode(15);
    root->right->left = newNode(23);
    return root;
}

treeNode *buildTree3() {
    /* Build Tree 3 in this function and return the root. */
    treeNode *root = newNode(12);
    root->left = newNode(3);
    root->right = newNode(25);
    root->left->right = newNode(8);
    root->left->right->right = newNode(9);
    root->right->left = newNode(16);
    root->right->right = newNode(30);
    root->right->right->right = newNode(39);
    return root;
}
```

buildTree1

buildTree2

buildTree3

Shih Yu Lai

Implementation

Data Structures:

- The binary tree nodes are structured as defined in the treeNode structure, which includes:
  - **val**: an integer value stored in the node.
  - **left**: a pointer to the left child of the node.
  - **right**: a pointer to the right child of the node.
- **newNode(int data):**
  - It creates a new tree node with the given data, initializes the left and right child pointers to NULL, and returns the pointer to the new node.
- **buildTree1(), buildTree2(), and buildTree3()**:
  - These functions build specific tree structures by manually creating nodes and linking them appropriately to form a complete binary tree. Each function returns the root of the constructed tree.

Constants and Variables:

- `MAX_STACK_SIZE`: A constant to define the maximum size of the stack.
- `stack`: An array of treeNode pointers used as a stack to keep track of nodes during the traversal.
- `top`: An integer index representing the top of the stack.

Main Function:

- In `main()`, the program builds three different binary trees using `buildTree1()`, `buildTree2()`, and `buildTree3()`. For each tree, it calls `iter_inorder()` to perform and display the inorder traversal.

Algorithm- Iterative Inorder Traversal:

- Function: `iter_inorder(treeNode *node)`
- Purpose: To perform an inorder traversal (left, root, right) of a binary tree iteratively using a stack.
- Logic:
  - Initialize a stack to keep track of nodes and a pointer `current` to traverse the tree.
  - Continue the traversal while there are unvisited nodes (`current != NULL`) or there are nodes in the stack (`top != -1`).
  - If `current` is not `NULL`, push `current` onto the stack and move to the left child. This continues until the leftmost node.
  - If `current` is `NULL` and the stack is not empty, pop the top node from the stack, process it (print the value), and then move to its right child.
  - Repeat the above steps until all nodes are processed.