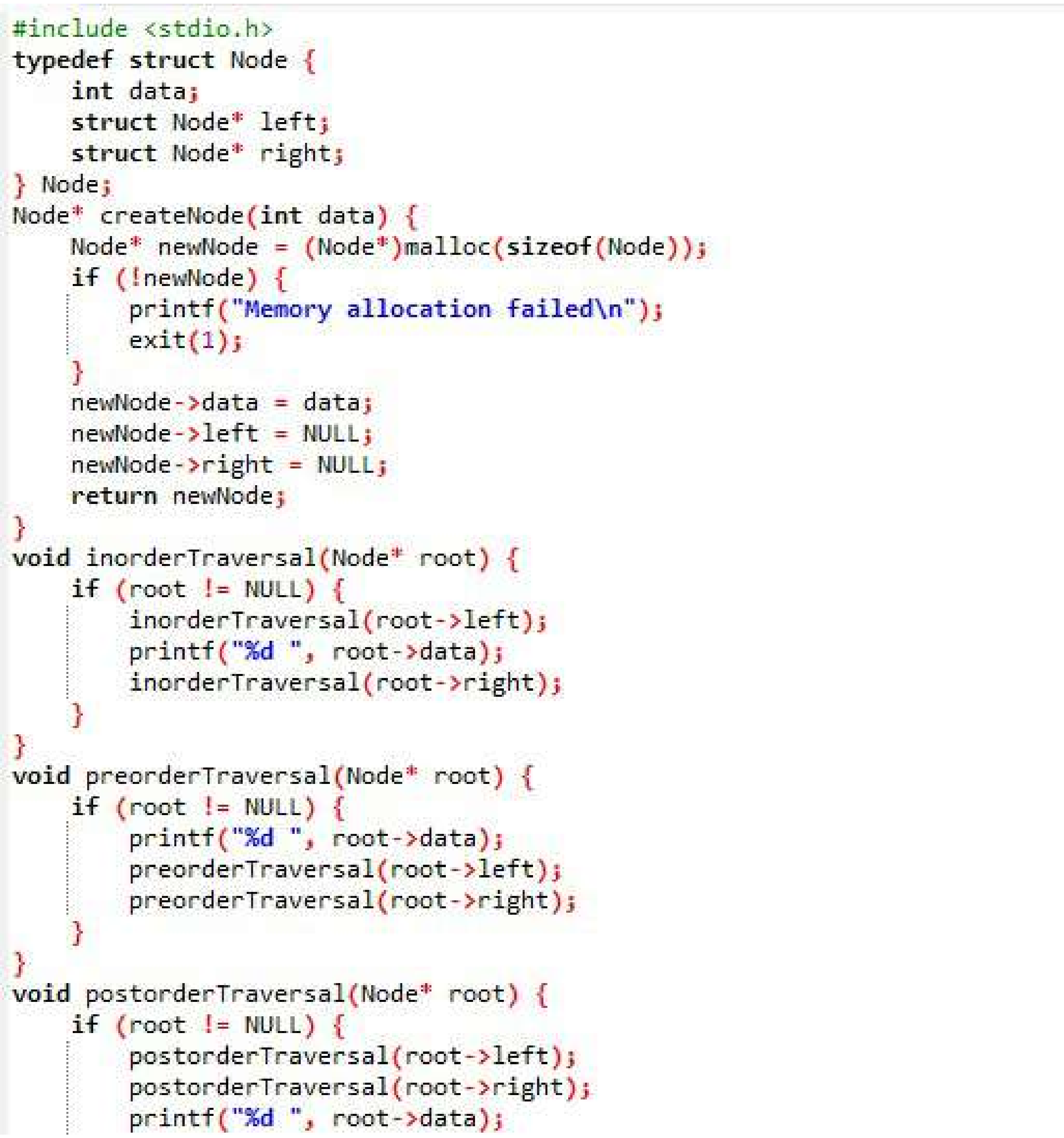
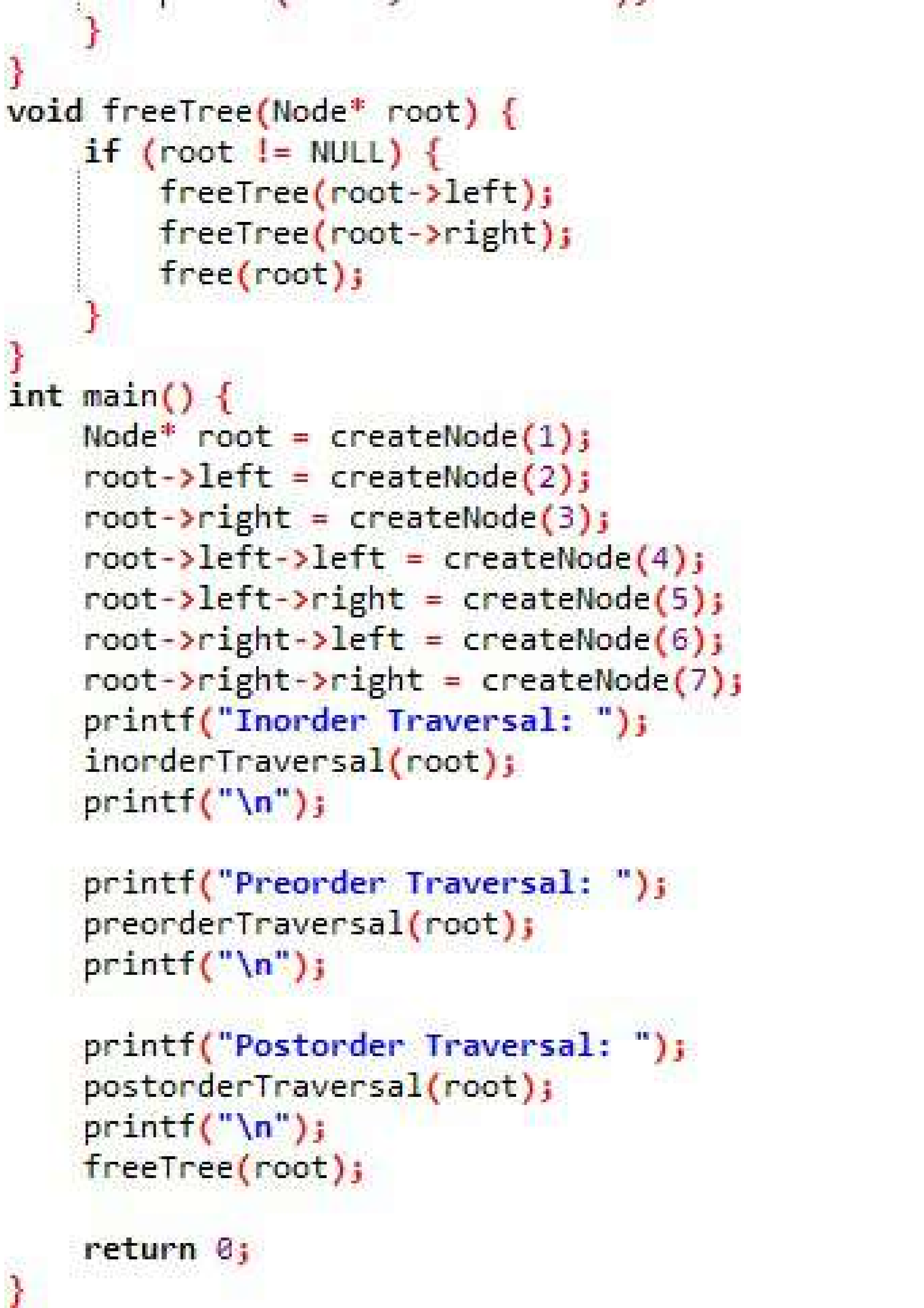
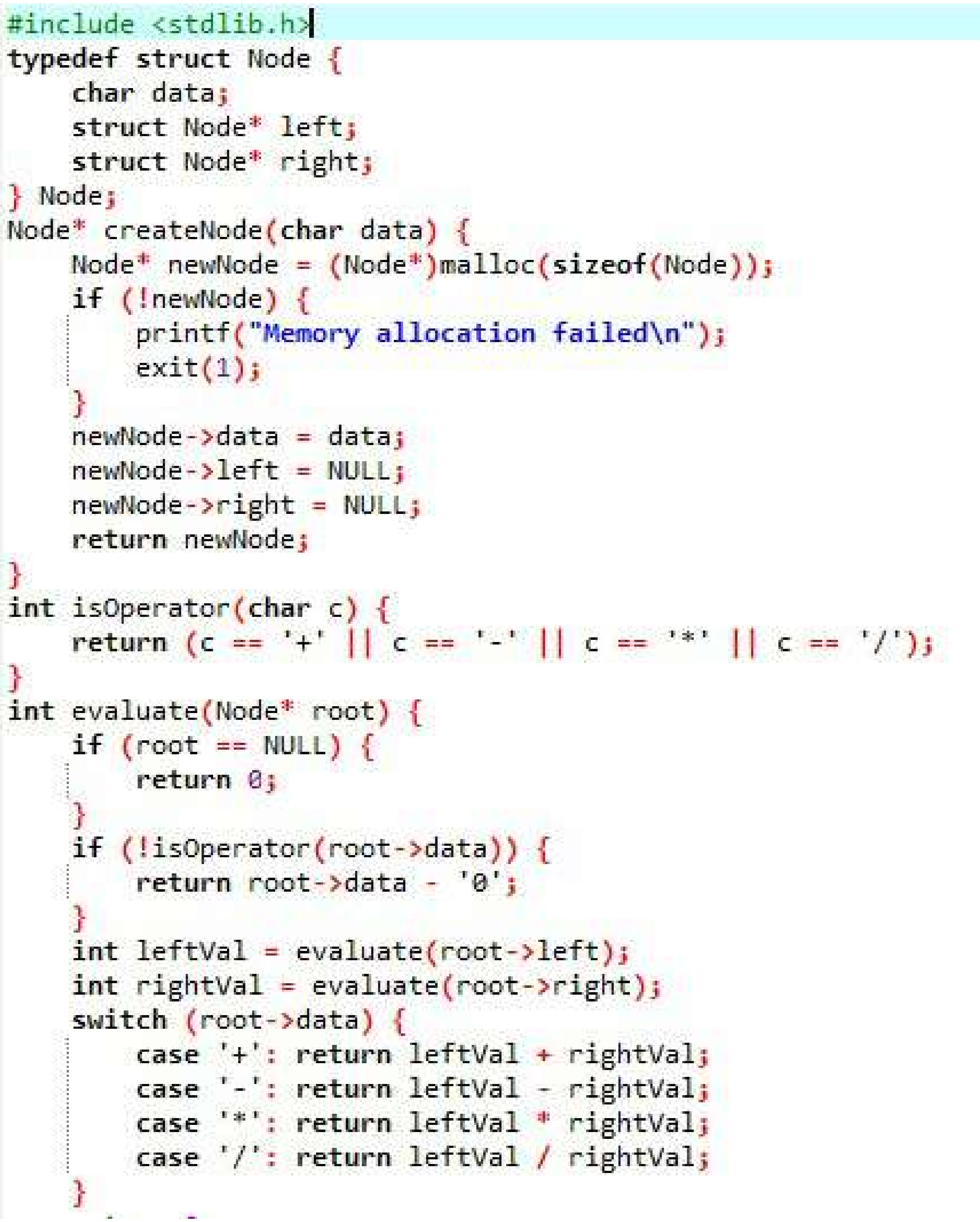


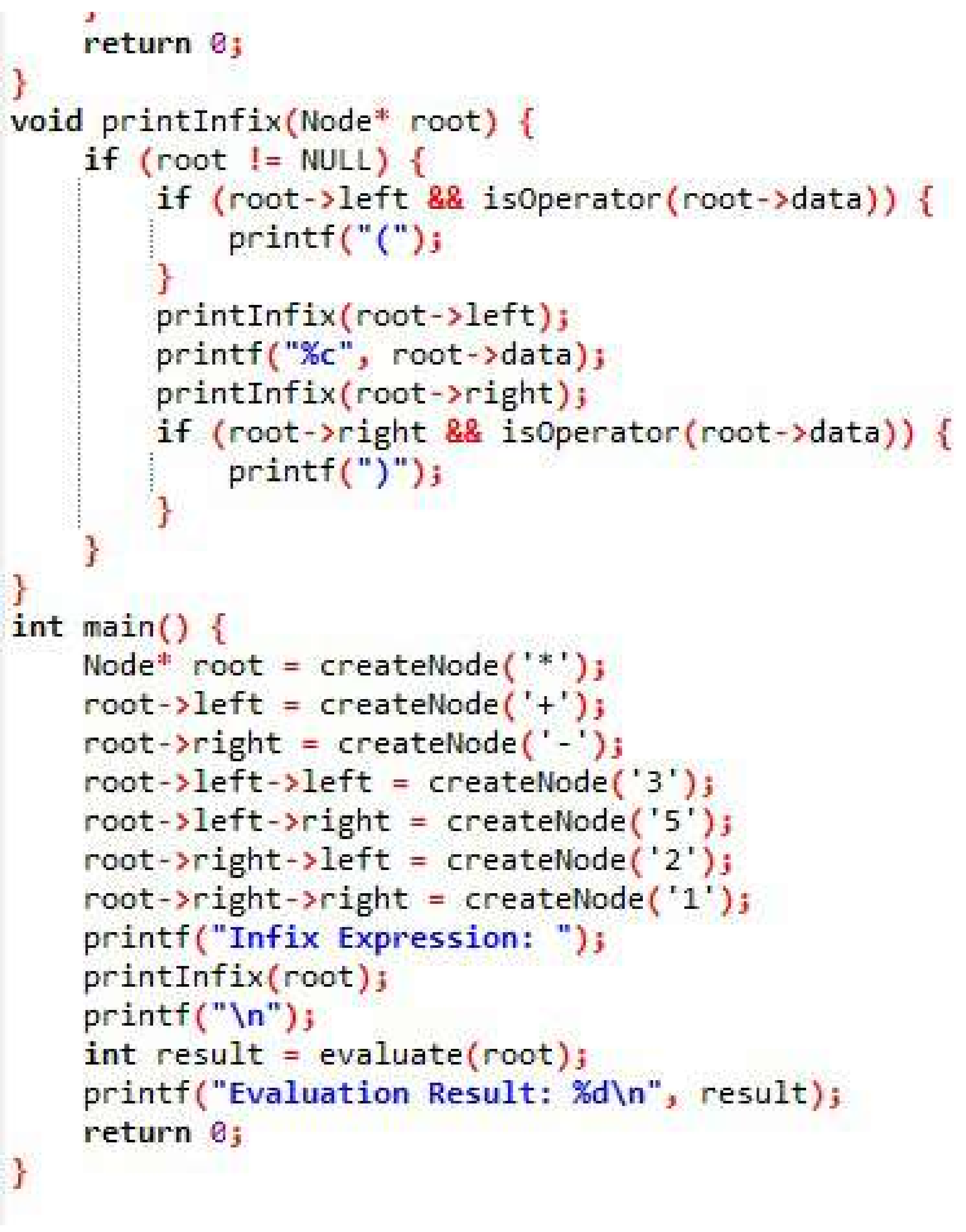
```c
#include <stdio.h>
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
void preorderTraversal(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
```



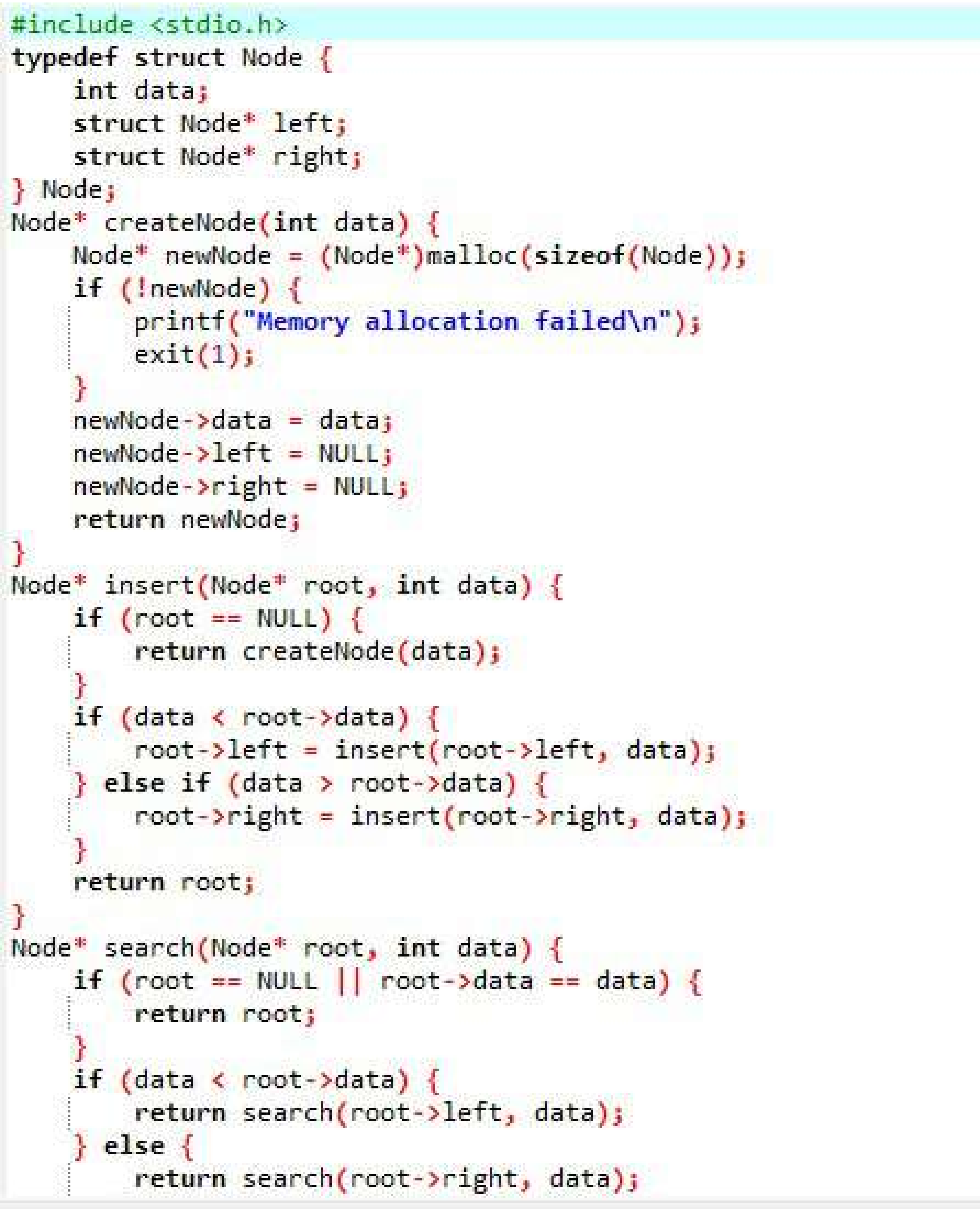
```c
    }
}
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    freeTree(root);

    return 0;
}
```



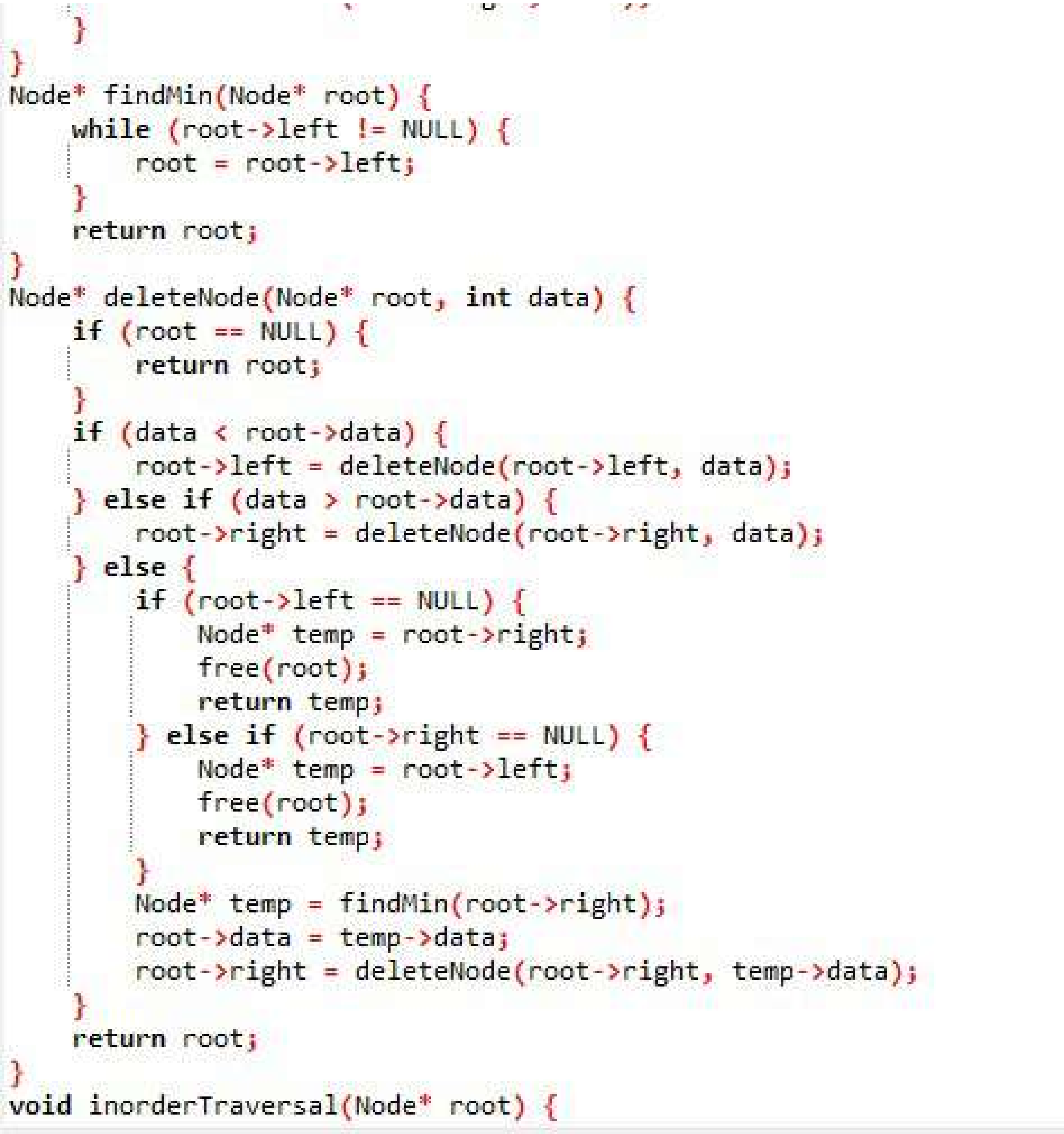
```c
#include <stdlib.h>
typedef struct Node {
    char data;
    struct Node* left;
    struct Node* right;
} Node;
Node* createNode(char data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}
int evaluate(Node* root) {
    if (root == NULL) {
        return 0;
    }
    if (!isOperator(root->data)) {
        return root->data - '0';
    }
    int leftVal = evaluate(root->left);
    int rightVal = evaluate(root->right);
    switch (root->data) {
        case '+': return leftVal + rightVal;
        case '-': return leftVal - rightVal;
        case '*': return leftVal * rightVal;
        case '/': return leftVal / rightVal;
    }
```



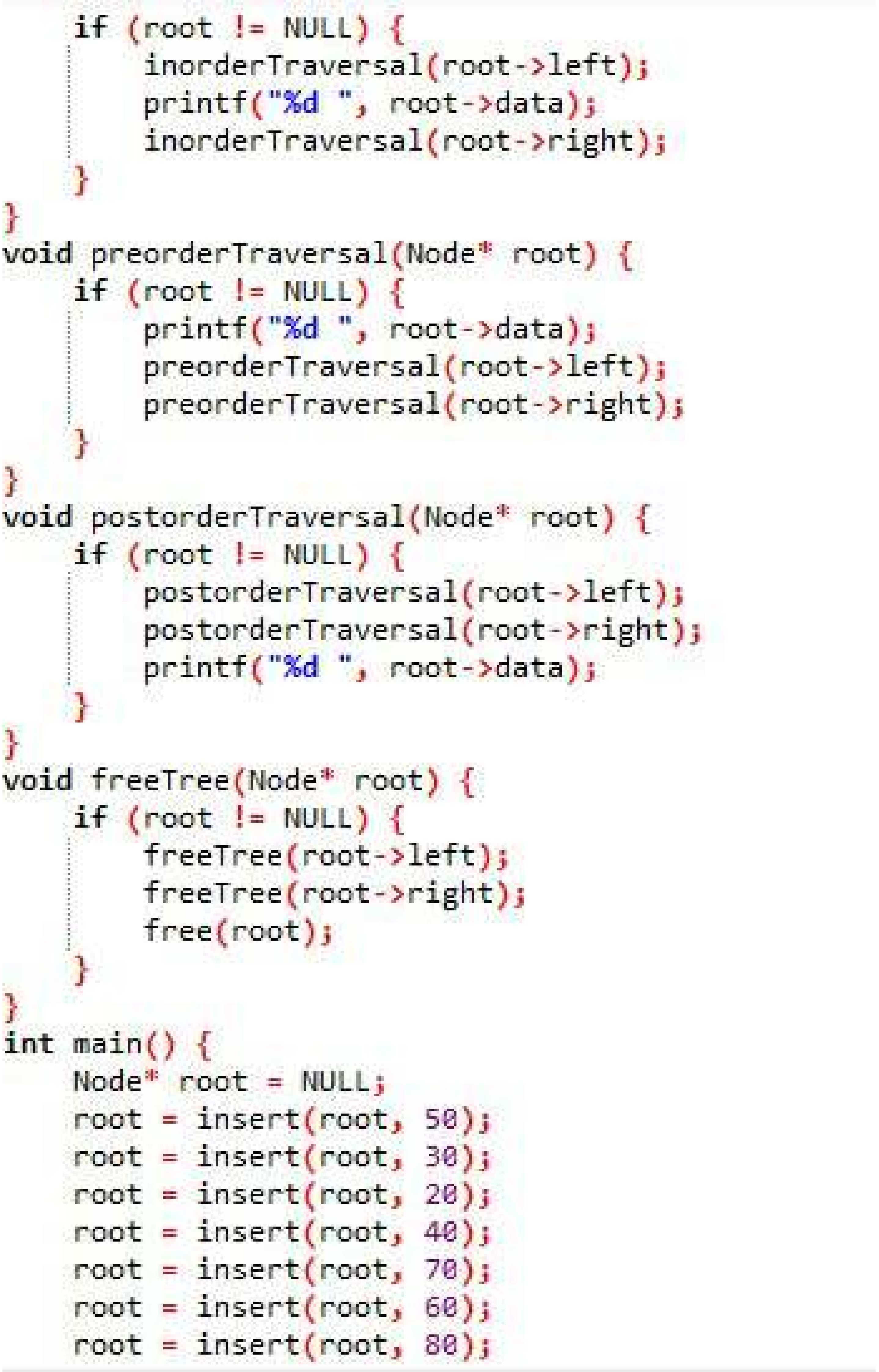
```c
    }
    return 0;
}
void printInfix(Node* root) {
    if (root != NULL) {
        if (root->left && isOperator(root->data)) {
            printf("(");
        }
        printInfix(root->left);
        printf("%c", root->data);
        printInfix(root->right);
        if (root->right && isOperator(root->data)) {
            printf(")");
        }
    }
}
int main() {
    Node* root = createNode('*');
    root->left = createNode('+');
    root->right = createNode('-');
    root->left->left = createNode('3');
    root->left->right = createNode('5');
    root->right->left = createNode('2');
    root->right->right = createNode('1');
    printf("Infix Expression: ");
    printInfix(root);
    printf("\n");
    int result = evaluate(root);
    printf("Evaluation Result: %d\n", result);
    return 0;
}
```



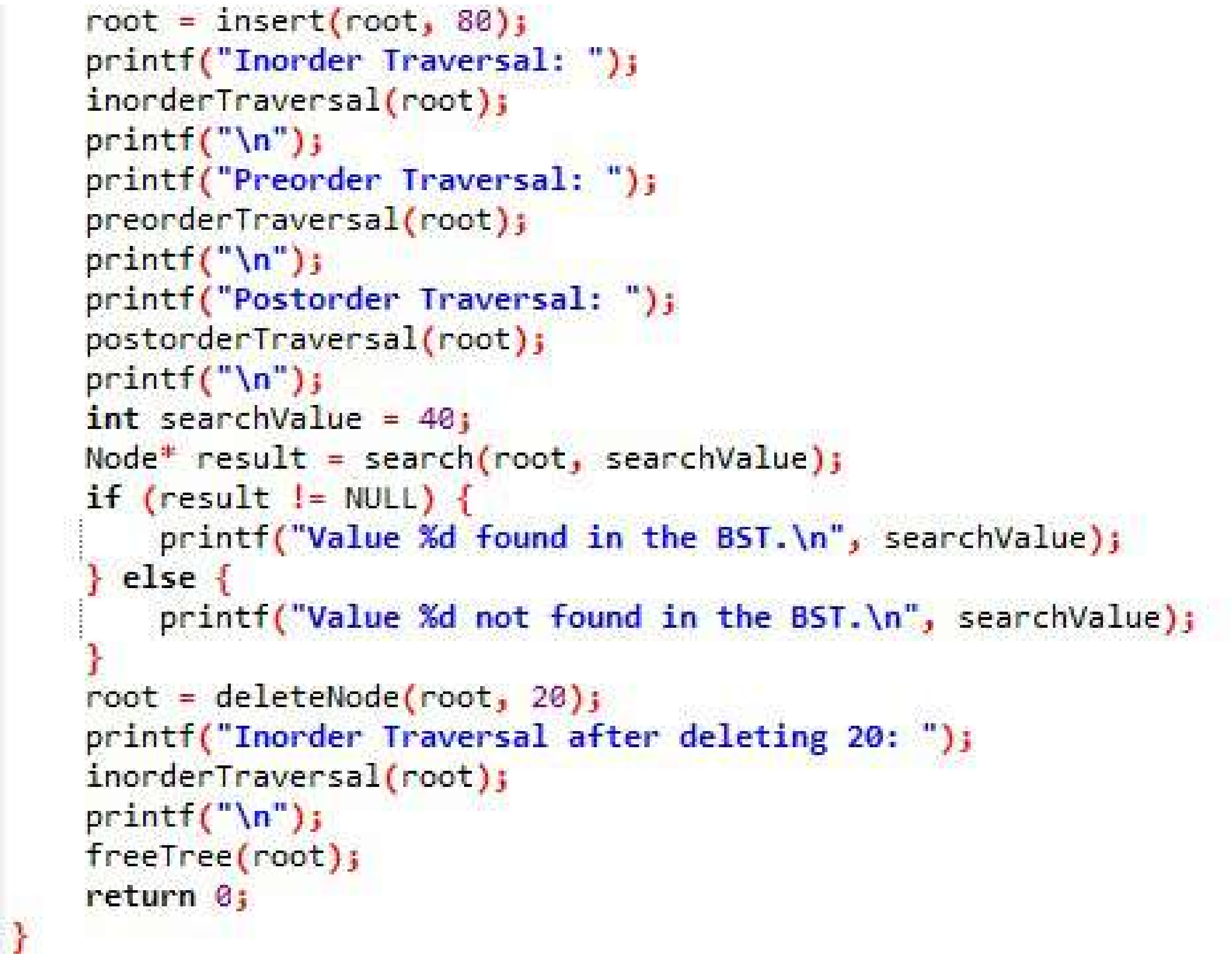
```c
#include <stdio.h>
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
Node* insert(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}
Node* search(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
```



```c
    }
}
Node* findMin(Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
Node* deleteNode(Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        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;
        }
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inorderTraversal(Node* root) {
```



```c
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
void preorderTraversal(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
int main() {
    Node* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
```



```c
    root = insert(root, 80);
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    int searchValue = 40;
    Node* result = search(root, searchValue);
    if (result != NULL) {
        printf("Value %d found in the BST.\n", searchValue);
    } else {
        printf("Value %d not found in the BST.\n", searchValue);
    }
    root = deleteNode(root, 20);
    printf("Inorder Traversal after deleting 20: ");
    inorderTraversal(root);
    printf("\n");
    freeTree(root);
    return 0;
}
```