[*] Untitled1

```c
BINARY TREE TRANSVERSAL
#include <stdio.h>
#include <stdlib.h>
typedef struct TreeNode {
    int value;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

INORDER
void inOrderTraversal(TreeNode* root) {
    if (root == NULL) return;
    inOrderTraversal(root->left);
    printf("%d ", root->value);
    inOrderTraversal(root->right);
}

PREORDER
void preOrderTraversal(TreeNode* root) {
    if (root == NULL) return;
    printf("%d ", root->value);
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

POSTORDER
void postOrderTraversal(TreeNode* root) {
    if (root == NULL) return;
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    printf("%d ", root->value);
}
```

[*] Untitled1

```
BINARY TREE EXPRESSION
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
typedef struct ExprNode {
    char value;
    struct ExprNode *left;
    struct ExprNode *right;
} ExprNode;
ExprNode* createNode(char value) {
    ExprNode* newNode = (ExprNode*)malloc(sizeof(ExprNode));
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
void inorderTraversal(ExprNode* root) {
    if (root == NULL) return;
    if (root->left && !isdigit(root->left->value) && root->left->value != '(') printf("(");
    inorderTraversal(root->left);
    printf("%c", root->value);
    inorderTraversal(root->right);
    if (root->right && !isdigit(root->right->value) && root->right->value != ')') printf(")");
}
int evaluate(ExprNode* root) {
    if (root == NULL) return 0;
    if (isdigit(root->value)) {
        return root->value - '0';
    }
    int leftEval = evaluate(root->left);
    int rightEval = evaluate(root->right);
    switch (root->value) {
        case '+': return leftEval + rightEval;
        case '-': return leftEval - rightEval;
        case '*': return leftEval * rightEval;
        case '/': return leftEval / rightEval;
```

```
        case '/': return leftval / rightval;
        default: return 0;
    }
}
void freeTree(ExprNode* root) {
    if (root == NULL) return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}
int main() {
    ExprNode* root = createNode('*');
    r 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('8');
    printf("Expression: ");
    inorderTraversal(root);
    printf("\n");
    printf("Evaluation Result: %d\n", evaluate(root));
    freeTree(root);
    return 0;
}
```

```
    }
}
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
#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
    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;
}
```