Binary Tree

```
#include <stdio.h>
#include <stdlib.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 == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
Node* insertNode(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    } else {
        root->right = insertNode(root->right, data);
    }
    return root;
}
```

```
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;
}
Node* searchNode(Node* root, int data) {
```

```
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return searchNode(root->left, data);
    } else {
        return searchNode(root->right, data);
    }
}
void freeTree(Node* root) {
    if (root == NULL) return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}
int main() {
    Node* root = NULL;
    int choice, value;
    while (1) {
        printf("\nBinary Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insertNode(root, value);
```

```
            printf("Value %d inserted.\n", value);
            break;
        case 2:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            printf("Value %d deleted.\n", value);
            break;
        case 3:
            printf("Enter value to search: ");
            scanf("%d", &value);
            Node* found = searchNode(root, value);
            if (found) {
                printf("Value %d found.\n", value);
            } else {
                printf("Value %d not found.\n", value);
            }
            break;
        case 4:
            freeTree(root);
            return 0;
        default:
            printf("Invalid choice, please try again.\n");
    }
  }
}
```

Output:

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to insert: 50

Value 50 inserted.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to insert: 30

Value 30 inserted.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to insert: 70

Value 70 inserted.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to insert: 20

Value 20 inserted.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 3

Enter value to search: 30

Value 30 found.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 2

Enter value to delete: 30

Value 30 deleted.

Binary Tree Operations:

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 4

Binary Search Tree (Insertion , Deletion and Searching)

```
#include <stdio.h>
#include <stdlib.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 == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
Node* insertNode(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    } else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    return root;
}
Node* searchNode(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return searchNode(root->left, data);
```

```
    } else {
        return searchNode(root->right, data);
    }
}
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) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
void freeTree(Node* root) {
    if (root == NULL) return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}
int main() {
    Node* root = NULL;
    int choice, value;
    while (1) {
        printf("\nBinary Search Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. In-order Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
```

```
            root = insertNode(root, value);
            printf("Value %d inserted.\n", value);
            break;
        case 2:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            printf("Value %d deleted.\n", value);
            break;
        case 3:
            printf("Enter value to search: ");
            scanf("%d", &value);
            Node* found = searchNode(root, value);
            if (found) {
                printf("Value %d found.\n", value);
            } else {
                printf("Value %d not found.\n", value);
            }
            break;
        case 4:
            printf("In-order traversal: ");
            inorderTraversal(root);
            printf("\n");
            break;
        case 5:
            freeTree(root);
            return 0;
        default:
            printf("Invalid choice, please try again.\n");
    }
}
```

}

Output:

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 50

Value 50 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 30

Value 30 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 70

Value 70 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 20

Value 20 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 40

Value 40 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 1

Enter value to insert: 60

Value 60 inserted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 3

Enter value to search: 40

Value 40 found.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 3

Enter value to search: 90

Value 90 not found.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 2

Enter value to delete: 20

Value 20 deleted.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 4

In-order traversal: 30 40 50 60 70

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. In-order Traversal

5. Exit

Enter your choice: 5

Binary Tree Traversal (In order, Pre order, Post order)

```
#include <stdio.h>
#include <stdlib.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 == NULL) {
      printf("Memory allocation failed\n");
      exit(1);
   }
   newNode->data = data;
```

```
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
void preorderTraversal(Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
void inorderTraversal(Node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
void postorderTraversal(Node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}
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("Pre-order traversal: ");
```

```
    preorderTraversal(root);
    printf("\n");
    printf("In-order traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Post-order traversal: ");
    postorderTraversal(root);
    printf("\n");
    free(root->left->left);
    free(root->left->right);
    free(root->right->left);
    free(root->right->right);
    free(root->left);
    free(root->right);
    free(root);
    return 0;
}
```

Output:

Pre-order traversal: 3 2 4 5 3 6 7

In-order traversal: 4 2 5 3 6 3 7

Post-order traversal: 4 5 2 6 7 3 3