Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №4

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Бинарное дерево поиска»

Выполнил:

Студент группы 23ВВВ2

Стрельцов А.П.

Федоров Б.М.

Приняли:

Юрова О. В.

Митрохин М. А.

Пенза 2024

**Цель работы:** поработать с динамическими списками, выполнить задания и усвоить материал.

**Лабораторное задание.**

**Задание 1**:

1. **Реализовать алгоритм поиска вводимого с клавиатуры значения в уже созданном дереве.**

**ЛИСТИНГ**

**laba 4-1.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Функция для создания дерева

struct Node\* CreateTree(struct Node\* root, int data) {

// Если дерево пустое, создаем новый узел

if (root == NULL) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Ошибка выделения памяти");

exit(0);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Если значение уже существует, не добавляем его

if (data == root->data) {

printf("Значение %d уже существует в дереве. Пропускаем добавление.\n", data);

return root;

}

// Рекурсивно добавляем в левое или правое поддерево

if (data < root->data) {

root->left = CreateTree(root->left, data);

}

else {

root->right = CreateTree(root->right, data);

}

return root;

}

// Функция для вывода дерева

void print\_tree(struct Node\* r, int l) {

if (r == NULL) {

return;

}

print\_tree(r->right, l + 1);

for (int i = 0; i < l; i++) {

printf(" ");

}

printf("%d\n", r->data);

print\_tree(r->left, l + 1);

}

// Функция для поиска значения в дереве

struct Node\* search(struct Node\* root, int value) {

if (root == NULL || root->data == value) {

return root;

}

if (value < root->data) {

return search(root->left, value);

}

else {

return search(root->right, value);

}

}

int main() {

setlocale(LC\_ALL, "");

int D, start = 1;

struct Node\* root = NULL;

printf("-1 - окончание построения дерева\n");

while (start) {

printf("Введите число: ");

scanf("%d", &D);

if (D == -1) {

printf("Построение дерева окончено\n\n");

start = 0;

}

else {

root = CreateTree(root, D);

}

}

print\_tree(root, 0);

printf("Введите значение для поиска: ");

scanf("%d", &D);

struct Node\* found = search(root, D);

if (found != NULL) {

printf("Значение %d найдено в дереве.\n", D);

}

else {

printf("Значение %d не найдено в дереве.\n", D);

}

return 0;

}

Результат работы программы

На рисунке 1 показана реализация задания №1.

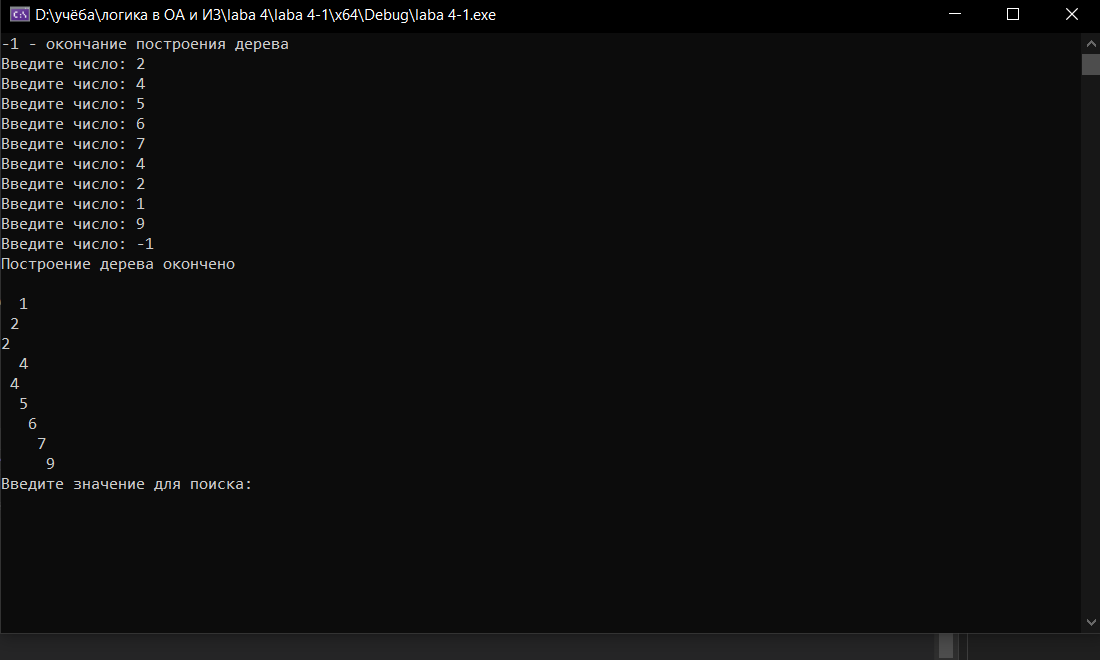


Рисунок 1 - Результат работы программы №1.

1. **Реализовать функцию подсчёта числа вхождений заданного элемента в дерево.**

**ЛИСТИНГ**

**laba 4-2.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Функция для создания дерева

struct Node\* CreateTree(struct Node\* root, struct Node\* r, int data) {

if (r == NULL) {

r = (struct Node\*)malloc(sizeof(struct Node));

if (r == NULL) {

printf("Ошибка выделения памяти");

exit(0);

}

r->left = NULL;

r->right = NULL;

r->data = data;

if (root == NULL) return r;

if (data > root->data) root->left = r;

else root->right = r;

return r;

}

if (data > r->data)

CreateTree(r, r->left, data);

else

CreateTree(r, r->right, data);

return root;

}

// Функция для вывода дерева

void print\_tree(struct Node\* r, int l) {

if (r == NULL) {

return;

}

print\_tree(r->right, l + 1);

for (int i = 0; i < l; i++) {

printf(" ");

}

printf("%d\n", r->data);

print\_tree(r->left, l + 1);

}

// Функция для подсчёта числа вхождений элемента в дереве

int count\_occurrences(struct Node\* root, int value) {

if (root == NULL) {

return 0;

}

int count = 0;

if (root->data == value) {

count = 1;

}

return count + count\_occurrences(root->left, value) + count\_occurrences(root->right, value);

}

int main() {

setlocale(LC\_ALL, "");

int D, start = 1;

struct Node\* root = NULL;

printf("-1 - окончание построения дерева\n");

while (start) {

printf("Введите число: ");

scanf("%d", &D);

if (D == -1) {

printf("Построение дерева окончено\n\n");

start = 0;

}

else {

root = CreateTree(root, root, D);

}

}

print\_tree(root, 0);

printf("Введите значение для подсчёта вхождений: ");

scanf("%d", &D);

int occurrences = count\_occurrences(root, D);

printf("Значение %d встречается в дереве %d раз(а).\n", D, occurrences);

return 0;

}

Результат работы программы

На рисунке 2 показана реализация задания №2.



Рисунок 2 - Результат работы программы №2.

1. **\* Изменить функцию добавления элементов для исключения добавления одинаковых символов.**

**ЛИСТИНГ**

**laba 4-3.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Функция для создания дерева

struct Node\* CreateTree(struct Node\* root, int data) {

if (root == NULL) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Ошибка выделения памяти");

exit(0);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Проверка на уникальность значения

if (data < root->data) {

root->left = CreateTree(root->left, data);

}

else if (data > root->data) {

root->right = CreateTree(root->right, data);

}

else {

printf("Значение %d уже существует в дереве.\n", data);

}

return root;

}

// Функция для вывода дерева

void print\_tree(struct Node\* r, int l) {

if (r == NULL) {

return;

}

print\_tree(r->right, l + 1);

for (int i = 0; i < l; i++) {

printf(" ");

}

printf("%d\n", r->data);

print\_tree(r->left, l + 1);

}

// Функция для поиска значения в дереве

struct Node\* search(struct Node\* root, int value) {

if (root == NULL || root->data == value) {

return root;

}

if (value < root->data) {

return search(root->left, value);

}

else {

return search(root->right, value);

}

}

int main() {

setlocale(LC\_ALL, "");

int D, start = 1;

struct Node\* root = NULL;

printf("-1 - окончание построения дерева\n");

while (start) {

printf("Введите число: ");

scanf("%d", &D);

if (D == -1) {

printf("Построение дерева окончено\n\n");

start = 0;

}

else {

root = CreateTree(root, D);

}

}

print\_tree(root, 0);

printf("Введите значение для поиска: ");

scanf("%d", &D);

struct Node\* found = search(root, D);

if (found != NULL) {

printf("Значение %d найдено в дереве.\n", D);

}

else {

printf("Значение %d не найдено в дереве.\n", D);

}

return 0;

}

Результат работы программы

На рисунке 3 показана реализация задания №3. Программа реализует базовые операции со стеком: добавление элементов, удаление элементов и просмотр содержимого стека.



Рисунок 3 - Результат работы программы №3.

1. **\* Оценить сложность процедуры поиска по значению в бинарном дереве.**

В данном коде реализованы три основные функции: создание дерева, вывод дерева и поиск значения в дереве. Давайте оценим сложность каждой из них.

Создание дерева (CreateTree):

В худшем случае, если дерево становится несбалансированным (например, элементы добавляются в порядке возрастания), сложность создания дерева будет O(n), где n — количество узлов в дереве. Это происходит потому, что для каждого нового элемента необходимо пройти по всем предыдущим узлам.

В сбалансированном дереве сложность будет O(log n), так как мы можем отбрасывать половину узлов на каждом уровне.

Вывод дерева (print\_tree):

Сложность этой функции составляет O(n), так как она обходит все узлы дерева, чтобы вывести их значения. Каждый узел посещается один раз.

Поиск значения (search):

Сложность поиска также зависит от структуры дерева. В сбалансированном дереве сложность будет O(log n), а в несбалансированном — O(n), как уже упоминалось ранее.

Общая сложность

Если рассматривать весь процесс, включая создание дерева, вывод и поиск, то общая сложность будет зависеть от того, как именно добавляются элементы в дерево:

Несбалансированное дерево:

Создание: O(n)

Вывод: O(n)

Поиск: O(n)

Итого: O(n)

Сбалансированное дерево:

Создание: O(log n)

Вывод: O(n)

Поиск: O(log n)

Итого: O(n) (вывод остается линейным, так как мы обходим все узлы)

Таким образом, в зависимости от структуры дерева, сложность операций может варьироваться, но в общем случае, для всех операций, связанных с деревом, можно сказать, что они имеют линейную сложность в худшем случае

**Вывод**

изучили новые функции, применили их в лабораторной работе.