МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра «Вычислительная техника»

ОТЧЁТ

По лабораторной работе №4
По курсу «Логика и основы алгоритмизации в инженерных задачах»
На тему «Бинарное дерево поиска»

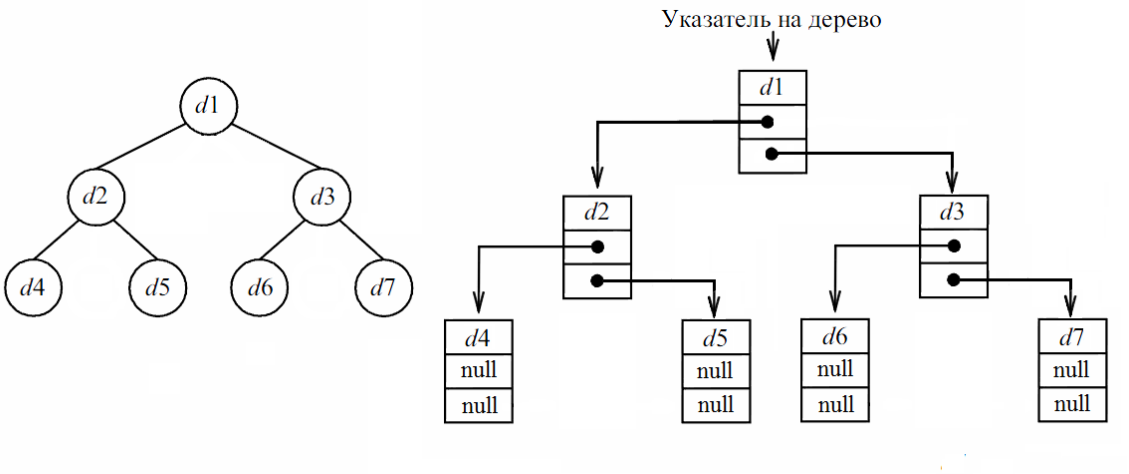
Выполнили
студенты
группы
23ВВВ4:
Святов И.Ю.
Епинин Д.В.

Приняли:
Юрова О.В.
Деев М.В.

Пенза 2024

Общие сведения:

Бинарные деревья – это деревья, у каждого узла которого возможно наличие только двух сыновей. Двоичные деревья являются упорядоченными.



Двоичное дерево можно представить в виде нелинейного связанного списка.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами:

– значение левого потомка меньше значения родителя;

– значение правого потомка больше значения родителя.

Такие структуры используются для сохранения данных в отсортированном виде.

В простейшем случае для создания элемента списка используется структура, в которой объединяются полезная информация и ссылка на следующий элемент списка:

```
struct Node {
int data;
struct Node *left;
struct Node *right;
};
```

Задание:

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

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

struct Node {
int data;
struct Node* left;
```

```
struct Node* right;
};
struct Node* root;
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->right = r;
	else
		root->left = r;
	return root;
}
if (data > r->data)
	CreateTree(r, r->right, data);
else
	CreateTree(r, r->left, data);
return root;
}
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* searchTree(struct Node* r, int key) {
// Базовый случай
if (r == NULL || r->data == key)
	return r;

// Если ключ меньше текущего, ищем в левом поддереве
if (key < r->data)
	return searchTree(r->left, key);

// Иначе ищем в правом поддереве
return searchTree(r->right, key);
}

int main() {
setlocale(LC_ALL, "");
int D, start = 1;
root = NULL;
printf(" '-1' - окончание построения дерева\n");
while (start) {
```

```
	printf("Введите число: ");
	scanf_s("%d", &D);
	if (D == -1) {
		printf("Построение дерева окончено\n\n");
		start = 0;
	}
	else
		root = CreateTree(root, root, D);
}
print_tree(root, 0);
//поиск в дереве
// Поисковая часть
printf("Введите число для поиска в дереве: ");
scanf("%d", &D);  // Изменил на scanf
struct Node* result = searchTree(root, D);
if (result != NULL) {
	printf("Элемент %d найден в дереве.\n", D);
}
else {
	printf("Элемент %d не найден в дереве.\n", D);
}
return 0;
}
```

2. Реализовать функцию подсчёта числа вхождений заданного элемента в дерево.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

struct Node {
int data;
int count; //счетчик вхождений
struct Node* left;
struct Node* right;
};
struct Node* root;

struct Node* CreateTree(struct Node* root, int data) {
if (root == NULL) {
	root = (struct Node*)malloc(sizeof(struct Node));
	if (root == NULL) {
		printf("Ошибка выделения памяти");
		exit(0);
	}
	root->left = NULL;
	root->right = NULL;
	root->data = data;
	root->count = 1; // Инициализация счетчика
	return root;
}
if (data == root->data) {
	root->count++; // Увеличиваем счетчик, если такое значение уже есть
```

```
}
else if (data < root->data) {
	root->left = CreateTree(root->left, data);
}
else {
	root->right = CreateTree(root->right, data);
}
return root;
}

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);
}
//поиск значения в дереве
struct Node* SearchTree(struct Node* root, int data) {
if (root == NULL || root->data == data) {
	return root;
}
if (data < root->data) {
	return SearchTree(root->left, data);
}
else {
	return SearchTree(root->right, data);
}
}

int CountOccurrences(struct Node* root, int data) {
    struct Node* foundNode = SearchTree(root, data);
    if (foundNode != NULL) {
        return foundNode->count; // Возврат счетчика найденного узла
    }
    return 0; // Если узел не найден, возвращаем 0
}

int main(){
setlocale(LC_ALL, "");
int D, start = 1;
root = NULL;
printf(" '-1' - окончание построения дерева\n");
while (start){
	printf("Введите число: ");
	scanf_s("%d", &D);
	if (D == -1){
		printf("Построение дерева окончено\n\n");
		start = 0;
	}
	else
		root = CreateTree(root, D);
}
print_tree(root, 0);
```

```
//поиск в дереве
printf("Введите значение для поиска: ");
scanf_s("%d", &D);
int occurrences = CountOccurrences(root, D);
if (occurrences > 0) {
	printf("Значение %d найдено в дереве %d раз.\n", D, occurrences);
}
else {
	printf("Значение %d не найдено в дереве.\n", D);
}
return 0;
}
```

3. *Изменить функцию добавления элементов для исключения добавления одинаковых символов.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

struct Node {
    int data;
    int count; // Счетчик вхождений (в этом контексте будет всегда равен 1)
    struct Node* left;
    struct Node* right;
};

struct Node* root;

struct Node* CreateTree(struct Node* root, int data) {
    if (root == NULL) {
        root = (struct Node*)malloc(sizeof(struct Node));
        if (root == NULL) {
            printf("Ошибка выделения памяти");
            exit(0);
        }
        root->left = NULL;
        root->right = NULL;
        root->data = data;
        root->count = 1; // Инициализация счетчика
        return root;
    }
    if (data == root->data) {
        // Если значение уже существует, просто игнорируем его добавление
        printf("Значение %d уже существует в дереве и не будет добавлено.\n", data);
    }
    else if (data < root->data) {
        root->left = CreateTree(root->left, data);
    }
    else {
        root->right = CreateTree(root->right, data);
    }
    return root;
```

```
}

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);
}

struct Node* SearchTree(struct Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return SearchTree(root->left, data);
    }
    else {
        return SearchTree(root->right, data);
    }
}

int CountOccurrences(struct Node* root, int data) {
    struct Node* foundNode = SearchTree(root, data);
    if (foundNode != NULL) {
        return foundNode->count;
    }
    return 0; // Если узел не найден, возвращаем 0
}

int main() {
    setlocale(LC_ALL, "");
    int D, start = 1;
    root = NULL;
    printf(" '-1' - окончание построения дерева\n");
    while (start) {
        printf("Введите число: ");
        scanf_s("%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_s("%d", &D);
    int occurrences = CountOccurrences(root, D);
    if (occurrences > 0) {
        printf("Значение %d найдено в дереве %d раз.\n", D, occurrences);
    }
    else {
```

```
        printf("Значение %d не найдено в дереве.\n", D);
    }
    return 0;
}
```

4. * Оценить сложность процедуры поиска по значению в бинарном дереве.

Ответ:

Процедура поиска по значению в бинарном дереве имеет временную сложность, которая зависит от структуры дерева:

1. Сбалансированное бинарное дерево:

   - В лучшем и среднем случае – O(log n), где n – количество узлов в дереве. Это связано с тем, что каждый проход по дереву делит оставшуюся часть пополам.

2. Несбалансированное бинарное дерево:

   - В худшем случае – O(n). Это происходит, когда дерево становится линейным (например, если добавлять элементы в порядке возрастания), и в таком случае необходимо пройти через все узлы дерева для поиска значения.

Таким образом, сложность поиска в бинарном дереве может варьироваться от O(log n) до O(n), в зависимости от его сбалансированности.

Задание было: кол-во ввода и строки вводить.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

struct Node {
    char* data; // Строка для хранения данных
    int count;  // Счетчик вхождений (в этом контексте будет всегда равен 1)
    struct Node* left;
    struct Node* right;
};

// Глобальная переменная для корня дерева
struct Node* root;

// Функция для создания дерева
struct Node* CreateTree(struct Node* root, const char* data) {
    if (root == NULL) {
        root = (struct Node*)malloc(sizeof(struct Node));
        if (root == NULL) {
            printf("Ошибка выделения памяти");
            exit(0);
        }
```

```
        root->left = NULL;
        root->right = NULL;
        root->data = (char*)malloc(strlen(data) + 1); // Выделение памяти для
строки
        if (root->data == NULL) {
            printf("Ошибка выделения памяти для строки");
            exit(0);
        }
        strcpy(root->data, data); // Копирование строки
        root->count = 1; // Инициализация счетчика
        return root;
    }

    if (strcmp(data, root->data) == 0) {
        // Если значение уже существует, просто игнорируем его добавление
        printf("Значение '%s' уже существует в дереве и не будет добавлено.\n",
data);
    }
    else if (strcmp(data, root->data) < 0) {
        root->left = CreateTree(root->left, data);
    }
    else {
        root->right = CreateTree(root->right, data);
    }
    return root;
}

// Функция для печати 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("%s\n", r->data);
    print_tree(r->left, l + 1);
}

// Функция для поиска значения в дереве
struct Node* SearchTree(struct Node* root, const char* data) {
    if (root == NULL || strcmp(root->data, data) == 0) {
        return root;
    }
    if (strcmp(data, root->data) < 0) {
        return SearchTree(root->left, data);
    }
    else {
        return SearchTree(root->right, data);
    }
}

// Функция для подсчета вхождений
int CountOccurrences(struct Node* root, const char* data) {
    struct Node* foundNode = SearchTree(root, data);
    if (foundNode != NULL) {
        return foundNode->count;
    }
    return 0; // Если узел не найден, возвращаем 0
}
```

```
int main() {
    setlocale(LC_ALL, "");
    char D[100]; // Буфер для ввода строки
    int start = 1;
    int count = 0; // Счетчик вводов
    const int max_c = 10; // Максимальное количество вводов
    root = NULL;

    printf("Введите '-1' для окончания построения дерева\n");
    while (start) {
        if (count >= max_c) {
            printf("Максимальное количество вводов достигнуто. Построение дерева
окончено.\n\n");
            break;
        }

        printf("Введите строку: ");
        scanf("%s", D); // Используем %[^\n] для считывания строки с пробелами,
если это необходимо
        if (strcmp(D, "-1") == 0) {
            printf("Построение дерева окончено\n\n");
            start = 0;
        }
        else {
            root = CreateTree(root, D);
            count++;
        }
    }

    printf("\nДерево:\n");
    print_tree(root, 0);

    printf("Введите строку для поиска: ");
    scanf("%s", D);
    int occurrences = CountOccurrences(root, D);
    if (occurrences > 0) {
        printf("Значение '%s' найдено в дереве %d раз.\n", D, occurrences);
    }
    else {
        printf("Значение '%s' не найдено в дереве.\n", D);
    }

    // Освобождаем выделенную память
    // Здесь необходимо реализовать функцию очистки памяти для всего дерева

    return 0;
}
```

Вывод: В ходе лабораторной работы научились работать с двоичным бинарным деревом. Сделали поиск значений по дереву, реализовали счетчик, также не допустили повторений одинаковых значений в дерево.