МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания 7-1

Тема: «Балансировка дерева поиска»

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Фамилия И. О.
Группа АААА-00-00

Москва 2024

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ.....3
1.1 Цель работы.....3
1.2 Индивидуальный вариант.....3
2 ХОД РАБОТЫ.....4
2.1 Анализ решения.....4
2.2 Код программы.....5
2.3 Код программы.....5
2.3 Результаты тестирования.....9
5 ВЫВОД.....10
6 ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ.....11

1 ВВЕДЕНИЕ

1.1 Постановка задачи

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Процедуры оформить в виде самостоятельных режимов работы созданного дерева. Выбор режимов производить с помощью пользовательского (иерархического ниспадающего) меню.

Провести полное тестирование программы на дереве размером n=10 элементов, сформированном вводом с клавиатуры. Тест-примеры определить самостоятельно. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием созданного дерева, принципов программной реализации алгоритмов работы с деревом, описанием текста исходного кода и проведенного тестирования программы.

1.2 Индивидуальный вариант

Таблица 1 — Индивидуальный вариант

Вариант	Тип значения узла	Алгоритмы
19	Строка - город	Вставка элемента (с балансировкой) Обратный обход Симметричный обход Найти длину пути от корня до заданного значения Найти высоту дерева

2 ХОД РАБОТЫ

2.1 Анализ решения

АВЛ-дерево — двоичное дерево поиска, ключи которого удовлетворяют свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм для поиска в двоичных деревьях поиска.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

Представим узел АВЛ-дерева в виде структуры в соответствии с индивидуальным вариантом. Код структуры представлен в листинге 1.

Листинг 1 - Структура узла

```
struct Node
{
    string key = "";
    unsigned char height;

    Node *left;
    Node *right;
    Node(string k)
    {
        key = k;
        left = right = 0;
        height = 0;
    }
};
```

Поле heght хранит высоту поддерева с корнем в данном узле. Значение данного поля используется при балансировке.

Для работы с высотой определим вспомогательную функцию, которая будет определять высоту равную 0 для нулевых указателей. Код функции представлен в листинге 2.

Листинг 2 - Вспомогательная функция получения высоты

```
unsigned char height(Node *p)
{
    return p ? p->height : 0;
}
```

Определим вторую вспомогательную функцию, вычисляющую фактор балансировки. Код представлен в листинге 3.

Листинг 3 - Функция вычисления фактора балансировки

```
int bfactor(Node *p)
{
    return height(p->right) - height(p->left);
}
```

И наконец для вычисления корректного значения высоты для заданного узла определим функцию fixheight. Код представлен в листинге 4.

Листинг 4 - Функция вычисления высоты для заданного узла

```
void fixheight(Node *p)
{
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl > hr ? hl : hr) + 1;
}
```

Для того, что бы свойства АВЛ-дерева соблюдали его свойства, необходимо производить балансировку узлов. Это необходимо делать в тех случаях когда фактор балансировки оказывается равным 2 или -2.

Для балансировки АВЛ-дерева используются простые повороты вокруг узлов дерева - вправо и влево. Коды реализующие повороты представлены в листинге 5.

Листинг 5 - Функции поворотов узла

```
Node *rotateright(Node *p)
{
    Node *q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
}

Node *rotateleft(Node *q)
{
```

```
    Node *p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}
```

Теперь реализуем саму функцию балансировки. Код представлен в листинге 6. Функция вычисляет высоту и фактор балансировки для заданного узла. Если фактор = 2 или -2, производится необходимый поворот, в зависимости от того же фактора, но для дочерних узлов.

Листинг 6 - Функции поворотов узла

```
Node *balance(Node *p)
{
    fixheight(p);
    if (bfactor(p) == 2)
    {
        if (bfactor(p->right) < 0)
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if (bfactor(p) == -2)
    {
        if (bfactor(p->left) > 0)
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p;
}
```

Реализуем функцию вставки нового узла в дерева. Для этого спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии выполняется балансировка текущего узла. Код представлен в листинге 7.

Листинг 7 - Функция вставки нового узла

```
Node *insert(Node *p, string k) // вставка ключа k в дерево с
корнем p
{
    if (!p)
        return new Node(k);
    if (k < p->key)
        p->left = insert(p->left, k);
```

```
    else
        p->right = insert(p->right, k);
    return balance(p);
}
```

Реализуем функцию поиска длины от корня до конкретного узла. Для этого применим такой же алгоритм поиска по дереву, какой использовался при вставке, и будем прибавлять значение высоты каждый проход рекурсии. При совпадении ключей вернём значение высоты. Код представлен в листинге 8.

Листинг 8 - Функция нахождения длины пути до узла

```
int find_height(const Node *p, const string key, int height=0)
{
    if (!p)
        return -1;
    if (p->key == key)
        return height;
    if (key < p->key)
        return find_height(p->left, key, height + 1);
    return find_height(p->right, key, height + 1);
}
```

Реализуем функции симметричного и обратного обхода. Код представлен в листинге 9.

Листинг 9 - Функции обходов

```
void symmetric_out(Node *p)
{
    if (!p) return;
    symmetric_out(p->left);
    cout << p->key << " ";
    symmetric_out(p->right);
}

void reverse_out(Node *p)
{
    if (!p) return;
    reverse_out(p->left);
    reverse_out(p->right);
    cout << p->key << " ";
}
```

2.2 Результаты тестирования

Проведём тестирование программы. Результаты тестирования представлены на рис. 1-3.

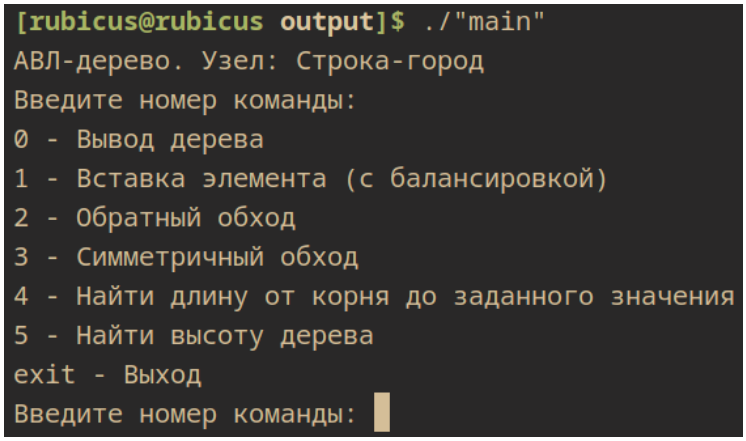

Рисунок 1 - Главное меню

Рисунок 2 - Ввод 10-и узлов и вывод дерева

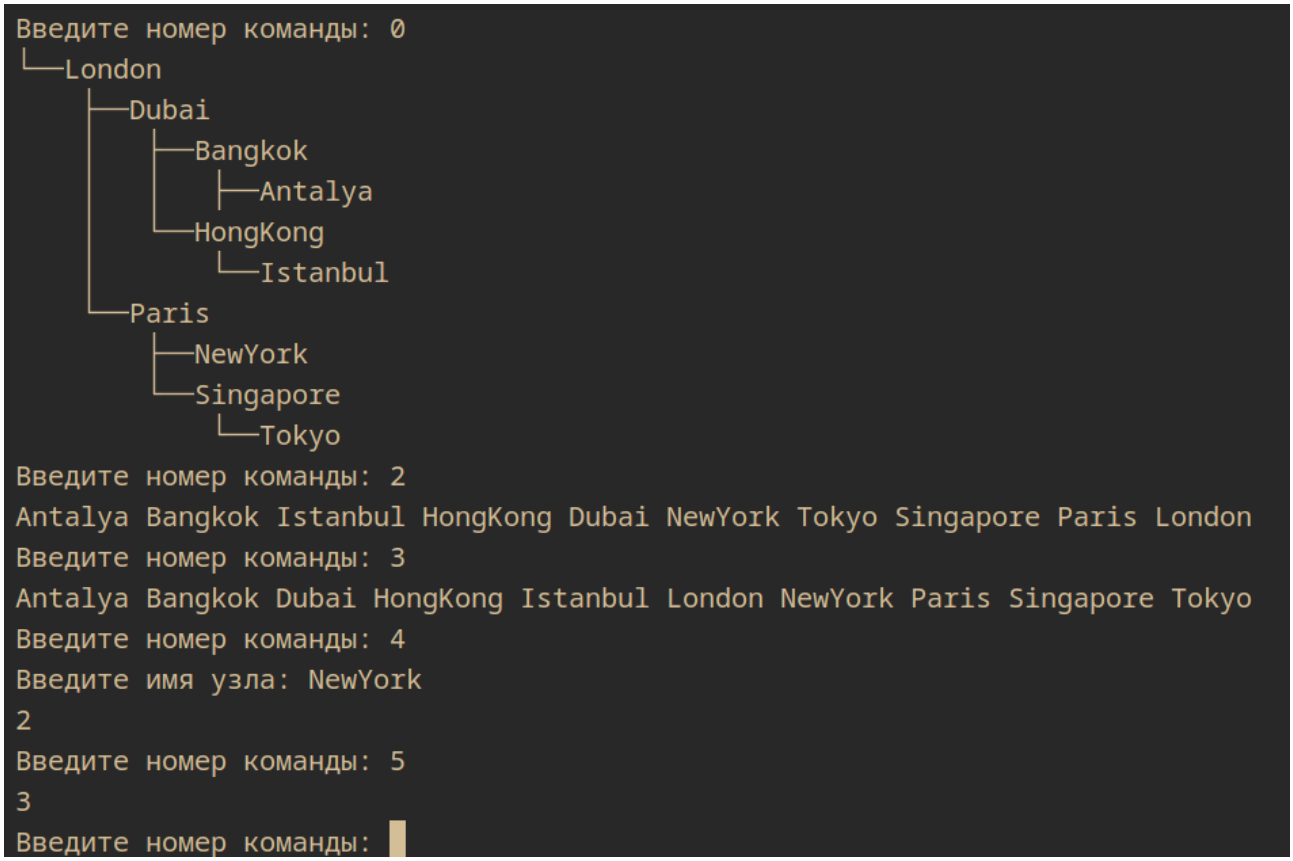

Рисунок 3 - Тестирование остальных функций

5 ВЫВОД

Были освоены приёмы по реализации и применения алгоритмов для работы с АВЛ-деревом.

6 ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Структуры и алгоритмы обработки данных (часть 2): Лекционные материалы / Рысин М. Л. МИРЭА — Российский технологический университет, 2022/23. – 82 с.