МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЕТ

по лабораторной работе № 3

по дисциплине «Алгоритмы и структуры данных»

ТЕМА: «МНОЖЕСТВО КАК ОБЪЕКТ»

Студенты гр. 3311 _____ Локтионов Т. И.

_____ Осипова Е. Р.

Преподаватель _____ Манирагена Валенс

Санкт-Петербург

2024

Цель работы {

исследование алгоритмов для работы с двоичным деревом.

}

Вариант 25 {

Вид дерева: **Двоичное**

Разметка: **Обратная**

Способ обхода: **Внутренний**

Что надо вычислить: **Количество вершин, имеющих не более двух потомков**

}

Обоснование выбора представления в памяти (в виде классов) {

Инкапсуляция

- Скрытие данных: Классы позволяют скрывать детали реализации и предоставлять только тот интерфейс, который необходим. Это предотвращает случайные изменения состояния объекта из внешнего кода.
- Упрощение доступа: Доступ к данным осуществляется через методы, что позволяет контролировать, как и когда данные могут быть изменены.

Упрощение работы с памятью

- Динамическое выделение памяти: Классы позволяют удобно управлять динамическим выделением и освобождением памяти, что полезно для создания сложных структур данных, таких как деревья, списки и т.д.
- Автоматическое управление ресурсами: с помощью деструкторов можно автоматически освобождать ресурсы, когда объект выходит из области видимости.

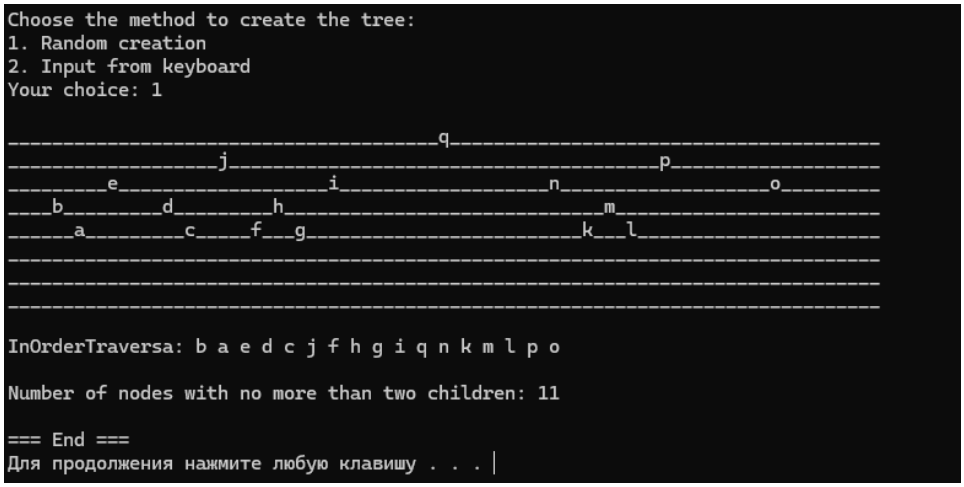
Использование классов для представления данных в памяти упрощает разработку, делает код более структурированным и облегчает поддержку и расширение программы. Это особенно важно в больших проектах, где поддержка и управление сложными структурами данных становятся критически важными для успеха.

}

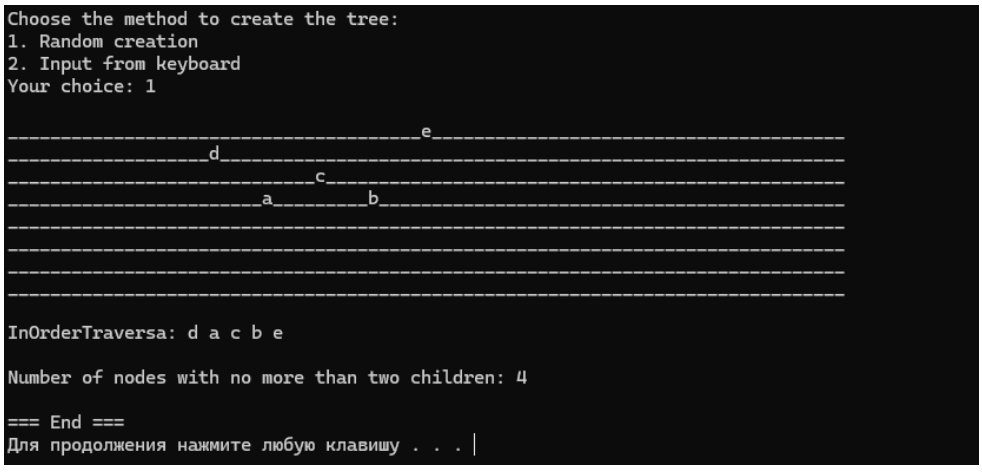
Тесты {

Случайная генерация {

1.



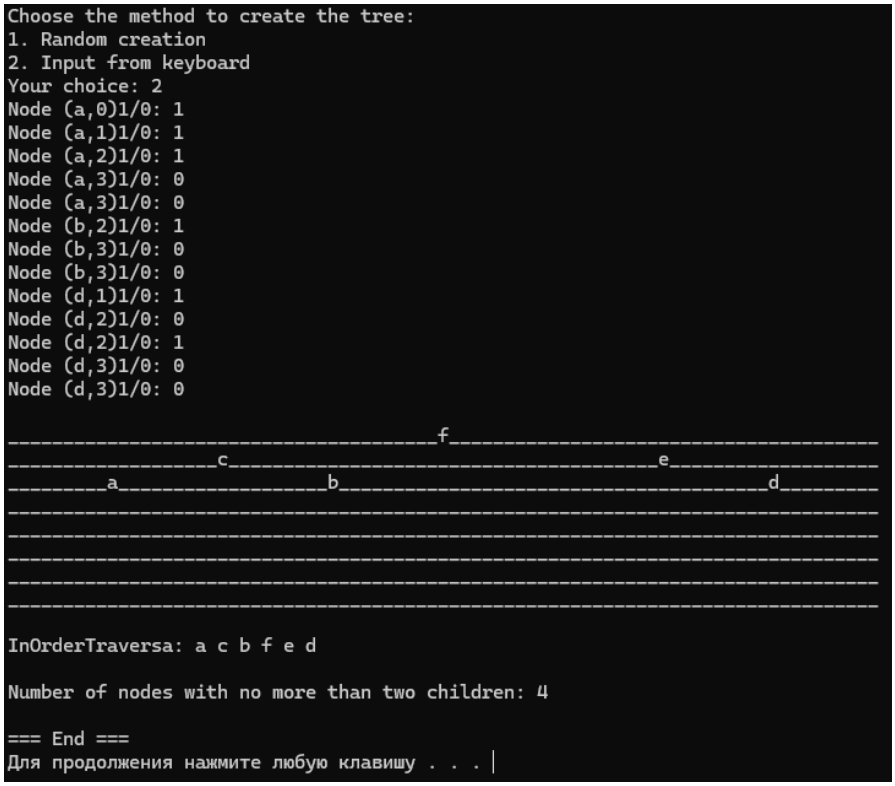
2.



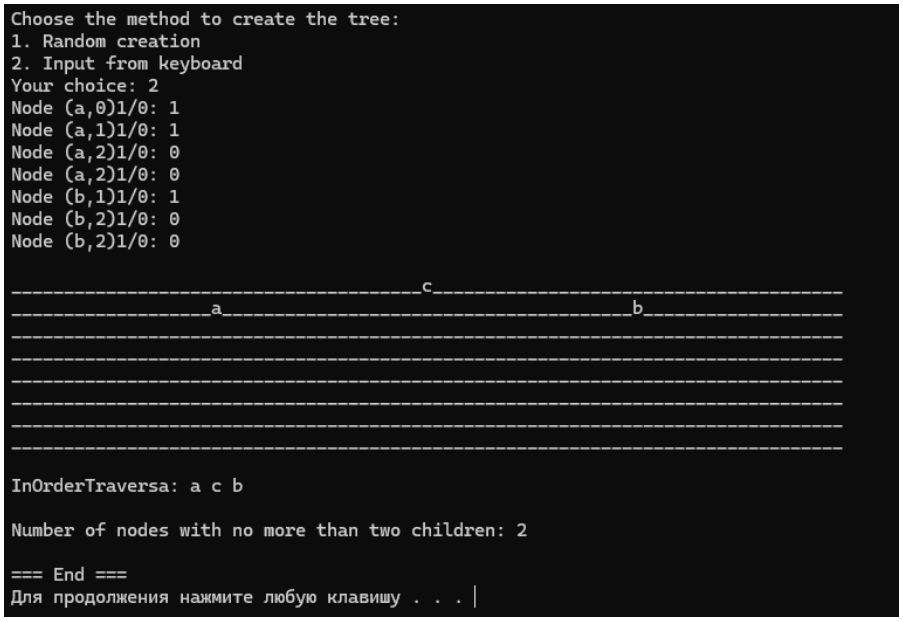
}

Ввод с клавиатуры {

1.



2.



}

}

Оценка сложности {

Создание дерева (Tree::MakeNode):

- В процессе создания дерева функция MakeNode вызывается рекурсивно для создания каждого узла, включая левых и правых потомков.
- Если дерево содержит N узлов, то каждый узел вызывается один раз, что означает, что сложность создания дерева составляет O(N), где N — это общее количество узлов.
- Генерация дерева зависит от того, случайно ли оно создается или вводится вручную. В обоих случаях количество вызовов функции равно количеству узлов в дереве.

Обход In-Order (Tree::InOrderTraversel):

- Функция InOrderTraversel реализует рекурсивный обход дерева "in-order" (левый потомок — корень — правый потомок). Она посещает каждый узел дерева один раз.
- Временная сложность обхода in-order для дерева с N узлами также составляет O(N), так как каждый узел посещается один раз.

Подсчет вершин с не более чем двумя потомками (Tree::countTwoChildren):

- Функция countTwoChildren выполняет обход каждого узла дерева и проверяет наличие его потомков. Она также рекурсивно вызывает саму себя для каждого потомка.
- Сложность этой функции O(N), так как каждый узел проверяется один раз, и функция рекурсивно спускается по дереву.

Отображение дерева (Tree::OutTree и Tree::OutNodes):

- Функция OutNodes также рекурсивно проходит по каждому узлу дерева и выводит его в массив SCREEN. Каждый узел обрабатывается один раз, включая позиционирование на экране.
- Временная сложность этой функции также составляет O(N), где N — количество узлов в дереве.

}

Заключение {

При проведении испытаний было подтверждено, что, при обратной разметке, узлом является наибольший узел из всей подгруппы, следовательно, корнем дерева является наибольший узел всего дерева.

Так же была замечена разница между обратным и симметричным обходами:
последовательность при обратном обходе представляет собой:

левый --> правый --> родитель

последовательность при симметричном обходе пр. собой:

левый --> родитель --> правый

}

Текст программы:

```
#include <iostream>
#include <ctime>
#include <cstring>
using namespace std;

// Класс «узел дерева»
class Node {
    char d;         // тег узла
    Node *lft;      // левый потомок
    Node *rgt;      // правый потомок
public:
    Node() : lft(nullptr), rgt(nullptr) {} // конструктор узла
    ~Node() { // деструктор (уничтожает поддерево)
        if (lft) delete lft;
        if (rgt) delete rgt;
    }
    friend class Tree; //  --> позволяет иметь доступ ко всем членам Node (включая
приватные)
};

// Класс «дерево в целом»
class Tree {
    Node *root;  // указатель на корень дерева
    char num, maxnum; // счетчики тегов
    int maxrow, offset; // максимальная глубина, смещение корня
    char **SCREEN; // память для отображения дерева
    void clrscr(); // очистка рабочей памяти
    Node* MakeNode(int depth, bool isRandom); // создание поддерева
    void OutNodes(Node *v, int r, int c); // отображение поддерева
    int countTwoChildren(Node *v); // подсчет вершин с <= 2 потомками
    void InOrderTraversel(Node *v); // явный внутренний обход
public:
    Tree() = delete;
    Tree(char num, char maxnum, int maxrow); // конструктор
    ~Tree(); // деструктор
    Tree(const Tree&) = delete;
    Tree& operator=(const Tree&) = delete;
    void MakeTree(bool isRandom) { root = MakeNode(0, isRandom); } // создание дерева

    bool exist() { return root != nullptr; } // проверка дерева
    void OutTree(); // отображение дерева
    int CountNodesWithLessThenTwoChildren(); // подсчет вершин с < 2 потомками
    void OutIOT();
};
```

```cpp
// offset - смещение корня в середину
Tree::Tree(char nm, char mnm, int mxr)
    : num(nm), maxnum(mnm), maxrow(mxr), offset(40), root(nullptr), SCREEN(new
char*[maxrow]) {
    for (int i = 0; i < maxrow; i++)
        SCREEN[i] = new char[80];
}

Tree::~Tree() {
    for (int i = 0; i < maxrow; i++)
        delete[] SCREEN[i];
    delete[] SCREEN;
    delete root;
}

void Tree::clrscr() {
    for (int i = 0; i < maxrow; i++)
        memset(SCREEN[i], '_', 80); // заполнения массива символами
}



Node* Tree::MakeNode(int depth, bool isRandom) {
    Node *v = nullptr; // сам узел (родитель)
    int Y;

    if (isRandom) {
        Y = (depth < rand() % 6 + 1) && (num <= 'z'); // случайная генерация
    } else {
        cout << "Node (" << num << ',' << depth << ")1/0: "; // запрос у пользователя
        cin >> Y;
    }

    if (Y) { // создание узла
        v = new Node; // непосредственно выделение памяти + вызов конструктора

        v->lft = MakeNode(depth + 1, isRandom); // создание левого потомка
        v->rgt = MakeNode(depth + 1, isRandom); // создание правого потомка
        v->d = num++; // разметка узла (увеличение переменной num)

    }
    return v;
}

void Tree::OutTree() {
```

```
    clrscr();
    OutNodes(root, 1, offset);
    for (int i = 0; i < maxrow; i++) {
        SCREEN[i][79] = 0;
        cout << '\n' << SCREEN[i];
    }
    cout << '\n';
}

void Tree::OutNodes(Node *v, int r, int c) { // r -- номер ряда; c -- offset (смещение)
    if (r && c && (c < 80)){
        SCREEN[r - 1][c - 1] = v->d; // вывод метки (-1 так как по факту всё считается с
0)
    }

    if (r < maxrow) {
        if (v->lft) OutNodes(v->lft, r + 1, c - (offset >> r)); // левый потомок
        if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset >> r)); // правый потомок
    }
}

void Tree::InOrderTraversel(Node *v) {
    if (v) {
        InOrderTraversel(v->lft);
        cout << v->d << ' ';
        InOrderTraversel(v->rgt);
    }
}

int Tree::countTwoChildren(Node *v) {
    if (!v) return 0;
    int count = 0;
    if ((!v->lft && !v->rgt) || (v->lft && !v->rgt) || (!v->lft && v->rgt)) {
        count = 1; // вершина с не более чем 2 потомками
    }
    return countTwoChildren(v->lft) + count + countTwoChildren(v->rgt);
}

int Tree::CountNodesWithLessThenTwoChildren() {
    return countTwoChildren(root);
}

void Tree::OutIOT() {
    InOrderTraversel(root);
}
```

```cpp
int main() {
    // Проверка конструкторов по умолчанию должна вызывать ошибку --> по сути они не нужны;
    // Tree t1; // Ошибка: конструктор по умолчанию запрещен;

    // Проверка конструктора копирования — должна вызвать ошибку
    Tree t2('a', 'z', 8); // Корректное создание дерева
    // Tree t3 = t2; // Ошибка: конструктор копирования запрещён

    // Проверка оператора присваивания — должна вызвать ошибку
    Tree t4('b', 'y', 7); // Корректное создание другого дерева
    // t4 = t2; // Ошибка: оператор присваивания запрещён


    Tree Tr('a', 'z', 8); // основное дерево для главного задания
    srand(27); // для тестов, чтобы увидеть разницу
    // srand(time(nullptr));


    int choice;
    cout << "Choose the method to create the tree:\n";
    cout << "1. Random creation\n";
    cout << "2. Input from keyboard\n";
    cout << "Your choice: ";
    cin >> choice;

    if (choice == 1) {
        // Случайное создание дерева
        Tr.MakeTree(true);
    } else if (choice == 2) {
        // Ввод с клавиатуры
        Tr.MakeTree(false);
    } else {
        cout << "Invalid choice!" << endl;
        return 1; // Выход при неправильном выборе
    }

    if (Tr.exist()) {
        Tr.OutTree();
        cout << endl;
        cout << "InOrderTraversa: ";
        Tr.OutIOT();
        cout << endl;
        int count = Tr.CountNodesWithLessThenTwoChildren();
        cout << "\nNumber of nodes with no more than two children: " << count << endl;
    } else {
```

```
        cout << "The tree is empty!" << endl;
    }
    cout << "\n=== End ===" << endl;


    system("pause");
    return 0;
}
```