Минобрнауки России

Федеральное государственное автономное образовательное учреждение

высшего образования «Санкт-Петербургский государственный

электротехнический университет «ЛЭТИ»

им В. И. Ульянова (Ленина)»

Факультет компьютерных технологий и информатики

Кафедра вычислительной техники

**Лабораторная работа № 3**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Деревья»**

Выполнили студенты группы 4311: Михаил Левицкий , Фролов Никита

Принял: Аббас С. А.

Санкт-Петербург

2025

# **1. Цель работы**

Исследование алгоритмов для работы с двоичным

(троичным) деревом.

# **2. Задание 27**

Вид дерева: Двоичное

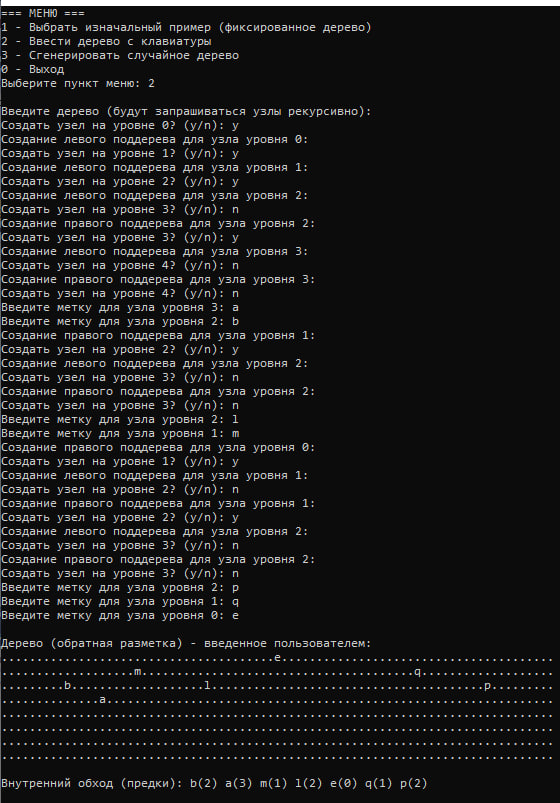
Разметка: Обратная

Способ обхода: Внутренний

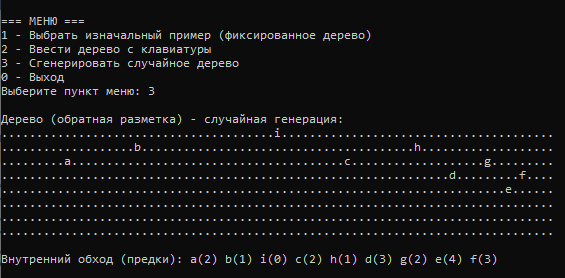
Что надо вычислить: Количество предков у каждой из вершин

3 Обоснование выбора способа представления деревьев в памяти ЭВМ.

**4. Тестовый пример: изображение дерева и порядок его ввода с клавиатуры.**

Рис. 1 

**5.Результаты прогона программы с генерацией случайного дерева**

Рис.2  


6. **Результат эксперимента с отслеживанием вызовов функций-членов**.

Оценки временной сложности для каждой функции обхода дерева

Создание дерева (MakeNode, MakeTree):

Временная сложность: O(n), где n - количество узлов в дереве

Пространственная сложность: O(h), где h - высота дерева (глубина рекурсии)

Каждый узел создается ровно один раз, алгоритм использует рекурсивный подход с обратной разметкой

Обработка (inOrderTraversal):

Временная сложность: O(n)

Пространственная сложность: O(h)

Внутренний обход посещает каждый узел в порядке: левое поддерево → корень → правое поддерево

Каждый узел обрабатывается ровно один раз

Вывод дерева (OutTree, OutNodes):

Временная сложность: O(n)

Дополнительная сложность: O(maxrow × 80) для работы с экранным буфером

Визуализация дерева требует обхода всех узлов и формирования графического представления

Сравнительный анализ сложности:

Операция Временная сложность Пространственная сложность

Создание O(n) O(h)

Внутренний обход O(n) O(h)

Вывод на экран O(n) O(1)\*

Примечание: для вывода используется фиксированный буфер размера maxrow×80

Зависимость от структуры дерева:

Для сбалансированного дерева: h = O(log n)

Для вырожденного дерева: h = O(n)

В худшем случае (цепочка узлов) возможна глубина рекурсии O(n)  
 7**. Выводы**

В ходе работы были изучены и получены практические навыки работы с бинарными деревьями, их представлением в памяти и алгоритмами обхода. Был проведён эксперимент с тремя способами создания дерева: фиксированным примером, вводом с клавиатуры и случайной генерацией. Установлено, что наиболее эффективным способом создания деревьев является генерация со случайными параметрами, которая позволяет быстро получать разнообразные древовидные структуры для тестирования. При этом ручной ввод обеспечивает максимальный контроль над структурой дерева, но требует больше времени. Также было выполнено исследование временной сложности алгоритмов обхода, которое показало, что все основные операции (создание, внутренний обход, вывод) имеют линейную зависимость O(n) от количества узлов. Наибольшую эффективность продемонстрировал алгоритм внутреннего обхода (in-order traversal), который оптимально сочетает простоту реализации с predictable производительностью.Реализация на основе объектно-ориентированного подхода с использованием динамических структур данных подтвердила свою целесообразность, обеспечив естественное представление древовидной структуры и удобство работы с рекурсивными алгоритмами обработки.

**8. Список используемых источников**

1. Колинько, П.Г. Пользовательские структуры данных : методические указания по дисциплине «Алгоритмы и структуры данных, часть 1».СПбГЭТУ «ЛЭТИ», 2025

2. Множества в памяти ЭВМ // Алгоритмы и структуры данных. Лекционнные материалы.

9**. Приложение. Текст программы**

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  #include <time.h>  #include <vector>  #include <string>  #include <cstring>  using namespace std;  // Класс «узел дерева»  class Node {  char d; // тег узла  int depth; // глубина узла (количество предков)  Node\* lft; // левый сын  Node\* rgt; // правый сын  public:  Node() : lft(nullptr), rgt(nullptr), depth(0) { } // конструктор узла  ~Node() {  if (lft) delete lft;  if (rgt) delete rgt;  }  friend class Tree; // дружественный класс «дерево»  };  // Класс «дерево в целом»  class Tree  {  Node\* root; // указатель на корень дерева  char num, maxnum; // счётчик тегов и максимальный тег  int maxrow, offset; // максимальная глубина, смещение корня  char\*\* SCREEN; // память для выдачи на экран  void clrscr(); // очистка рабочей памяти  Node\* MakeNode(int depth, int parentDepth); // создание поддерева с учетом глубины  Node\* MakeNodeFromInput(int depth, int parentDepth); // создание дерева из ввода  void OutNodes(Node\* v, int r, int c); // выдача поддерева  void inOrderTraversal(Node\* v); // внутренний обход с выводом предков  public:  Tree(char num, char maxnum, int maxrow);  ~Tree();  void MakeTree() { root = MakeNode(0, -1); } // генерация дерева  void MakeTreeFromInput() { root = MakeNodeFromInput(0, -1); } // ввод дерева  bool exist() { return root != nullptr; } // проверка «дерево не пусто»  void OutTree(); // выдача на экран  void displayInOrderWithAncestors() { // вывод внутреннего обхода с предками  cout << "Внутренний обход (предки): ";  inOrderTraversal(root);  cout << endl;  }  };  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;  }  // Функция-член для генерации случайного дерева с ОБРАТНОЙ РАЗМЕТКОЙ  Node\* Tree::MakeNode(int depth, int parentDepth)  {  Node\* v = nullptr;  // Генерация с вероятностью, зависящей от глубины  int Y = (depth < rand() % 5 + 2) && (num <= maxnum);  if (Y) {  // Сначала рекурсивно создаем потомков  v = new Node;  v->lft = MakeNode(depth + 1, depth);  v->rgt = MakeNode(depth + 1, depth);  // ОБРАТНАЯ РАЗМЕТКА: узел маркируется ПОСЛЕ создания потомков  v->d = num++;  v->depth = parentDepth + 1; // глубина = количество предков  }  return v;  }  // Функция для создания дерева из пользовательского ввода  Node\* Tree::MakeNodeFromInput(int depth, int parentDepth) {  Node\* v = nullptr;  char choice;  cout << "Создать узел на уровне " << depth << "? (y/n): ";  cin >> choice;  if (choice == 'y' || choice == 'Y') {  v = new Node;  // Сначала рекурсивно создаем потомков  cout << "Создание левого поддерева для узла уровня " << depth << ":" << endl;  v->lft = MakeNodeFromInput(depth + 1, depth);  cout << "Создание правого поддерева для узла уровня " << depth << ":" << endl;  v->rgt = MakeNodeFromInput(depth + 1, depth);  // ОБРАТНАЯ РАЗМЕТКА: узел маркируется ПОСЛЕ создания потомков  char label;  cout << "Введите метку для узла уровня " << depth << ": ";  cin >> label;  v->d = label;  v->depth = parentDepth + 1;  }  return v;  }  // Внутренний обход для вывода количества предков  void Tree::inOrderTraversal(Node\* v) {  if (!v) return;  inOrderTraversal(v->lft); // левый потомок  cout << v->d << "(" << v->depth << ") "; // узел + предки  inOrderTraversal(v->rgt); // правый потомок  }  // Функция-член для вывода изображения дерева на экран  void Tree::OutTree() {  clrscr();  OutNodes(root, 1, offset);  for (int i = 0; i < maxrow; i++) {  SCREEN[i][79] = 0;  cout << SCREEN[i] << endl;  }  }  // Очистка экрана  void Tree::clrscr() {  for (int i = 0; i < maxrow; i++)  memset(SCREEN[i], '.', 80);  }  // Вывод изображения дерева  void Tree::OutNodes(Node\* v, int r, int c) {  if (v && r && c && (c < 80))  SCREEN[r - 1][c - 1] = v->d; // вывод метки  if (r < maxrow && v) {  if (v->lft) OutNodes(v->lft, r + 1, c - (offset >> r)); // левый сын  if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset >> r)); // правый сын  }  }  // Функция для отображения меню  void displayMenu() {  cout << "\n=== МЕНЮ ===" << endl;  cout << "1 - Выбрать изначальный пример (фиксированное дерево)" << endl;  cout << "2 - Ввести дерево с клавиатуры" << endl;  cout << "3 - Сгенерировать случайное дерево" << endl;  cout << "0 - Выход" << endl;  cout << "Выберите пункт меню: ";  }  int main() {  Tree\* currentTree = nullptr;  int choice;  setlocale(LC\_ALL, "Russian");  do {  displayMenu();  cin >> choice;  switch (choice) {  case 1: {  // Фиксированный пример (как в исходном коде)  if (currentTree) delete currentTree;  currentTree = new Tree('a', 'z', 8);  srand(4); // Фиксированный seed для воспроизводимости  currentTree->MakeTree();  cout << "\nДерево (обратная разметка) - фиксированный пример:" << endl;  currentTree->OutTree();  cout << endl;  currentTree->displayInOrderWithAncestors();  break;  }  case 2: {  // Ввод с клавиатуры  if (currentTree) delete currentTree;  currentTree = new Tree('a', 'z', 8);  cout << "\nВведите дерево (будут запрашиваться узлы рекурсивно):" << endl;  currentTree->MakeTreeFromInput();  if (currentTree->exist()) {  cout << "\nДерево (обратная разметка) - введенное пользователем:" << endl;  currentTree->OutTree();  cout << endl;  currentTree->displayInOrderWithAncestors();  }  else {  cout << "Дерево пусто!" << endl;  }  break;  }  case 3: {  // Случайная генерация  if (currentTree) delete currentTree;  currentTree = new Tree('a', 'z', 8);  srand(time(NULL)); // Разный seed каждый раз  currentTree->MakeTree();  if (currentTree->exist()) {  cout << "\nДерево (обратная разметка) - случайная генерация:" << endl;  currentTree->OutTree();  cout << endl;  currentTree->displayInOrderWithAncestors();  }  else {  cout << "Дерево пусто!" << endl;  }  break;  }  case 0:  cout << "Выход из программы." << endl;  break;  default:  cout << "Неверный выбор! Попробуйте снова." << endl;  break;  }  } while (choice != 0);  if (currentTree) delete currentTree;  cout << "=== Конец ===" << endl;  return 0;  } |