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

Тема: НЕЛИНЕЙНЫЕ СТРУКТУРЫ

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент: Павлов Никита
группа ИКБО-50-23

Москва 2024

СОДЕРЖАНИЕ

1. БАЛАНСИРОВКА ДЕРЕВА ПОИСКА

1.1 ЦЕЛЬ РАБОТЫ

Освоить приёмы работы с нелинейной структурой дерева.

Персональный вариант: 24

Тип значения узла: Строка - город

Тип дерева: Бинарное дерево поиска

Используемые алгоритмы: Вставка элемента, прямой обход, симметричный обход, поиск среднего арифметического всех узлов, нахождение длины пути от корня до заданного значения.

1.2 ЗАДАНИЕ

1.2.1 Формулировка 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.

1.2.2 Математическая модель решения (описание алгоритма)

```
struct Node {
    string data;
    Node* left, * right;

    Node(string data) {
        this->data = data;
        left = right = nullptr;
    }
};
```

Рисунок 1 – Структура узла

1. **void BinarySearchTree::insert(string data)**

Цель: Вставляет новый узел с данными data в бинарное дерево поиска.

Алгоритм:

Вызывает рекурсивную функцию insert(root, data), которая вставляет новый узел в поддерево, начиная с корня.

2. **Node* BinarySearchTree::insert(Node* node, string data)**

Цель: Рекурсивная функция для вставки узла в бинарное дерево поиска.

Алгоритм:

Базовый случай: Если node равен nullptr (достигнут пустой узел), создается новый узел с данными data и возвращается указатель на этот новый узел.

Рекурсивный случай:

Если data меньше, чем данные в текущем узле (node->data), рекурсивно вызывается функция insert для левого поддерева (node->left).

Если data больше, чем данные в текущем узле, рекурсивно вызывается функция insert для правого поддерева (node->right).

После вставки узла в поддерево, функция возвращает node (указатель на текущий узел).

3. void BinarySearchTree::preorder()

Цель: Выполняет прямое обход (preorder traversal) бинарного дерева поиска и выводит данные узлов в консоль.

Алгоритм:

Вызывает рекурсивную функцию preorder(root), которая выполняет прямой обход, начиная с корня.

4. void BinarySearchTree::preorder(Node* node)

Цель: Рекурсивная функция для выполнения прямого обхода бинарного дерева поиска.

Алгоритм:

Базовый случай: Если node равен nullptr, функция ничего не делает.

Рекурсивный случай:

Выводит данные текущего узла (node->data) в консоль.

Рекурсивно вызывается функция preorder для левого поддерева (node->left).

Рекурсивно вызывается функция preorder для правого поддерева (node->right).

5. void BinarySearchTree::inorder()

Цель: Выполняет симметричный обход (inorder traversal) бинарного дерева поиска и выводит данные узлов в консоль.

Алгоритм:

Вызывает рекурсивную функцию inorder(root), которая выполняет симметричный обход, начиная с корня.

6. void BinarySearchTree::inorder(Node* node)

Цель: Рекурсивная функция для выполнения симметричного обхода бинарного дерева поиска.

Алгоритм:

Базовый случай: Если node равен nullptr, функция ничего не делает.

Рекурсивный случай:

Рекурсивно вызывается функция inorder для левого поддерева (node->left).

Выводит данные текущего узла (node->data) в консоль.

Рекурсивно вызывается функция inorder для правого поддерева (node->right).

7. double BinarySearchTree::average()

Цель: Вычисляет среднюю длину строк, хранящихся в узлах бинарного дерева поиска.

Алгоритм:

Базовый случай: Если дерево пусто (root равен nullptr), функция возвращает 0.

Рекурсивный случай:

Вычисляет суммарную длину всех строк в дереве с помощью функции sumNodes(root).

Вычисляет количество узлов в дереве с помощью функции countNodes(root).

Вычисляет среднюю длину строки, деля суммарную длину на количество узлов.

Возвращает полученное среднее значение.

8. int BinarySearchTree::sumNodes(Node* node)

Цель: Рекурсивная функция для вычисления суммарной длины строк, хранящихся в узлах бинарного дерева поиска.

Алгоритм:

Базовый случай: Если node равен nullptr, функция возвращает 0.

Рекурсивный случай:

Возвращает длину строки в текущем узле (node->data.length()) плюс рекурсивные вызовы sumNodes для левого и правого поддеревьев.

9. int BinarySearchTree::countNodes(Node* node)

Цель: Рекурсивная функция для подсчета количества узлов в бинарном дереве поиска.

Алгоритм:

Базовый случай: Если node равен nullptr, функция возвращает 0.

Рекурсивный случай:

Возвращает 1 (текущий узел) плюс рекурсивные вызовы countNodes для левого и правого поддеревьев.

10. int BinarySearchTree::pathLength(string target)

Цель: Вычисляет длину пути от корня дерева до узла с заданным значением target.

Алгоритм:

Вызывает рекурсивную функцию pathLength(root, target, 0), которая выполняет поиск пути, начиная с корня.

11. int BinarySearchTree::pathLength(Node* node, string target, int length)

Цель: Рекурсивная функция для вычисления длины пути до узла с заданным значением target.

Алгоритм:

Базовый случай: Если node равен nullptr, функция возвращает -1 (узел не найден).

Рекурсивный случай:

Если данные в текущем узле (node->data) равны target, функция возвращает текущую длину пути length.

Иначе функция рекурсивно вызывается для левого и правого поддеревьев с увеличенной длиной пути (length + 1).

Если в левом поддереве найден узел с заданным значением, функция возвращает длину пути из левого поддерева.

Если в правом поддереве найден узел с заданным значением, функция возвращает длину пути из правого поддерева.

Если ни в левом, ни в правом поддереве узел не найден, функция возвращает -1.

1.2.3 Код программы

```
void BinarySearchTree::preorder() {
    preorder(root);
    cout << endl;
}

void BinarySearchTree::preorder(Node* node) {
    if (node != nullptr) {
        cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
    }
}

void BinarySearchTree::inorder() {
    inorder(root);
    cout << endl;
}

void BinarySearchTree::inorder(Node* node) {
    if (node != nullptr) {
        inorder(node->left);
        cout << node->data << " ";
        inorder(node->right);
    }
}
```

```
BinarySearchTree::BinarySearchTree() {
    root = nullptr;
}

void BinarySearchTree::insert(string data) {
    root = insert(root, data);
}

Node* BinarySearchTree::insert(Node* node, string data) {
    if (node == nullptr) {
        return new Node(data);
    }
    if (data < node->data) {
        node->left = insert(node->left, data);
    }
    else if (data > node->data) {
        node->right = insert(node->right, data);
    }
    return node;
}
```

Рисунок 2-3 – Код программы

```cpp
int BinarySearchTree::pathLength(string target) {
    return pathLength(root, target, 0);
}

int BinarySearchTree::pathLength(Node* node, string target, int length) {
    if (node == nullptr) {
        return -1;
    }
    if (node->data == target) {
        return length;
    }
    int leftLength = pathLength(node->left, target, length + 1);
    int rightLength = pathLength(node->right, target, length + 1);
    if (leftLength != -1) {
        return leftLength;
    }
    else if (rightLength != -1) {
        return rightLength;
    }
    return -1;
}
```

```cpp
double BinarySearchTree::average() {
    if (root == nullptr) {
        return 0;
    }
    int sum = sumNodes(root);
    int count = countNodes(root);
    return (double)sum / count;
}

int BinarySearchTree::sumNodes(Node* node) {
    if (node == nullptr) {
        return 0;
    }
    return node->data.length() + sumNodes(node->left) + sumNodes(node->right);
}

int BinarySearchTree::countNodes(Node* node) {
    if (node == nullptr) {
        return 0;
    }
    return 1 + countNodes(node->left) + countNodes(node->right);
```

```cpp
int main()
{
    setlocale(LC_ALL, "rus");
    system("chcp 1251");

    BinarySearchTree tree;
    int choice, length;
    string data, target;

    string test[] = { "50", "30", "70", "20", "40", "60", "80", "10", "25", "45" };
    for (int i = 0; i < 10; i++) {
        tree.insert(test[i]);
    }
```

Рисунок 4-6 – Код программы

1.2.4 Результаты тестирования

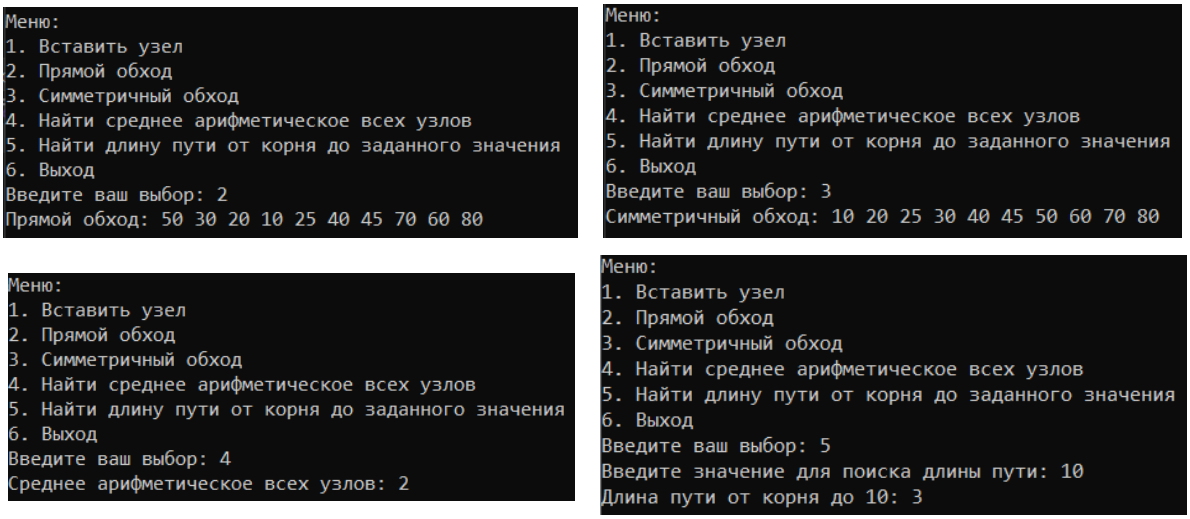

Рисунок 7-10 – Результаты тестирования

1.3 ВЫВОДЫ

Цель работы и сопутствующие задачи были успешно выполнены. Освоены приёмы работы с нелинейной структурой дерева. Реализован алгоритм построения бинарного дерева поиска и сопутствующий функции работы с ним.

2. ГРАФЫ: СОЗДАНИЕ, АЛГОРИТМЫ ОБХОДА, ВАЖНЫЕ ЗАДАЧИ ТЕОРИИ ГРАФОВ

2.1 ЦЕЛЬ РАБОТЫ

Освоить приёмы работы с графами: создание, алгоритмы обхода и прочие задачи.

Персональный вариант: 24

Алгоритм: Нахождение кратчайшего пути методом Беллмана Форда

Тестовый граф:

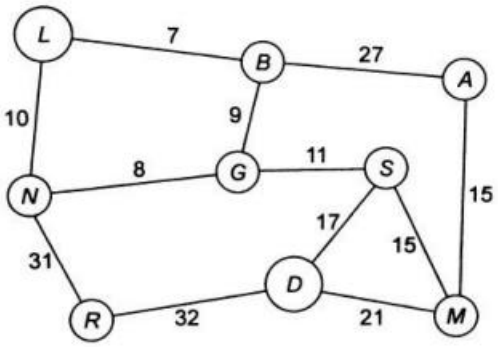

Рисунок 11 – Тестовый граф

2.2 ЗАДАНИЕ

2.2.1 Постановка задачи

Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания.

Самостоятельно выбрать и реализовать способ представления графа в памяти.

В программе предусмотреть ввод с клавиатуры произвольного графа. В вариантах построения основного дерева также разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

2.2.2 Математическая модель решения

1. **Graph::Graph(int numVertices) : table(numVertices){}**

Цель: Конструктор класса `Graph`. Инициализирует граф с заданным количеством вершин.

Алгоритм:

Создает вектор `table` с заданным количеством элементов, где каждый элемент представляет собой пару (вершина, вектор ребер). Изначально все вершины пустые, а вектор ребер пуст.

2. **void Graph::addEdge(string source, string destination, int weight)**

Цель: Добавляет ребро в граф.

Алгоритм:

Проходит по всем элементам вектора `table`.

Если в текущем элементе `table[i].first` хранится вершина `source`, то добавляет ребро (`destination`, `weight`) в вектор ребер для этой вершины `table[i].second`.

Если в `table[i].first` хранится пустая строка (`""`), то добавляет вершину `source` в `table[i].first` и добавляет ребро (`destination`, `weight`) в вектор ребер для этой вершины `table[i].second`.

Выходит из цикла, если вершина `source` была добавлена.

3. void Graph::printGraph()

Цель: Выводит информацию о графе в консоль.

Алгоритм:

Проходит по всем элементам вектора `table`.

Для каждой вершины `table[i].first` выводит имя вершины и список ребер, хранящихся в `table[i].second`, в формате "(destination, weight)".

4. vector<pair<string, int>> Graph::bellmanFord(string source)

Цель: Вычисляет кратчайшие пути от заданной вершины `source` до всех других вершин в графе с помощью алгоритма Беллмана-Форда.

Алгоритм:

Создает вектор `distances` с парами (вершина, расстояние) для всех вершин.

Инициализирует расстояния до `INF` для всех вершин, кроме начальной вершины `source`, для которой расстояние устанавливается в `0`.

Релаксация ребер:

Выполняет `table.size() - 1` итераций.

На каждой итерации проходит по всем вершинам в `table`.

Для каждой вершины проходит по всем ее исходящим ребрам `table[j].second`.

Для каждого ребра проверяет, можно ли уменьшить расстояние до конечной вершины `table[j].second[k].destination` через текущую вершину `table[j].first`.

Если расстояние до конечной вершины может быть уменьшено, обновляет значение `distances` для соответствующей вершины.

Проверка на наличие отрицательных циклов:

Проходит по всем вершинам в `table`.

Для каждой вершины проходит по всем ее исходящим ребрам.

Если найдено ребро, которое может уменьшить расстояние до конечной вершины после `table.size() - 1` итераций, то в графе присутствует отрицательный цикл.

Если обнаружен отрицательный цикл, выводит сообщение об ошибке и возвращает пустой вектор.

Возвращает вектор `distances`, содержащий пары (вершина, минимальное расстояние).

2.2.3 Код программы

```
const int INF = numeric_limits<int>::max();

struct Edge {
    string destination;
    int weight;
};

class Graph
{
private:
    vector<pair<string, vector<Edge>>> table;

public:
    Graph(int numVertices);
    void addEdge(string sourse, string destination, int weight);
    void printGraph();
    vector<pair<string, int>> bellmanFord(string source);
};
```

```
Graph::Graph(int numVertices) : table(numVertices){}

void Graph::addEdge(string source, string destination, int weight) {
    for (int i = 0; i < table.size(); i++) {
        if (table[i].first == source) {
            table[i].second.push_back({ destination, weight });
            break;
        }
        else if (table[i].first.empty()) {
            table[i].first = source;
            table[i].second.push_back({ destination, weight });
            break;
        }
    }
}

void Graph::printGraph() {
    for (int i = 0; i < table.size(); i++) {
        cout << "Вершина " << table[i].first << ":";
        for (auto edge : table[i].second) {
            cout << " (" << edge.destination << ", " << edge.weight << ")";
        }
        cout << endl;
    }
}
```

Рисунок 12-13 – Код программы

```
vector<pair<string, int>> Graph::bellmanFord(string source) {
    vector<pair<string, int>> distances;
    for (int i = 0; i < table.size(); i++) {
        if (table[i].first == source) {
            distances.push_back({ table[i].first, 0 });
        }
        else {
            distances.push_back({ table[i].first, INF });
        }
    }
    //релаксация ребер
    for (int i = 0; i < table.size() - 1; i++) {
        for (int j = 0; j < table.size(); j++) {
            for (int k = 0; k < table[j].second.size(); k++) {
                int d, w = table[j].second[k].weight;
                for (int u = 0; u < distances.size(); u++) {
                    if (distances[u].first == table[j].second[k].destination) {
                        d = u;
                        break;
                    }
                }
                if (distances[j].second != INF && distances[j].second + w < distances[d].second) {
                    distances[d].second = distances[j].second + w;
                }
            }
        }
    }
    //проверка на наличие отрицательных циклов
    for (int i = 0; i < table.size(); i++) {
        for (int j = 0; j < table[i].second.size(); j++) {
            int d, w = table[i].second[j].weight;
            for (int k = 0; k < distances.size(); k++) {
                if (distances[k].first == table[i].second[j].destination) {
                    d = k;
                    break;
                }
            }
            if (distances[i].second != INF && distances[i].second + w < distances[d].second) {
                cout << "Граф содержит отрицательный цикл!" << endl;
                return vector<pair<string, int>>();
            }
        }
    }
    return distances;
}
```

Рисунок 14 – Код программы

```
void main() {
    setlocale(LC_ALL, "rus");
    system("chcp 1251");

    int numVertices, numEdges;
    cout << "Введите количество вершин: ";
    cin >> numVertices;
    cout << "Введите количество направленных ребер: ";
    cin >> numEdges;
    Graph graph(numVertices);
    string source, destination;
    int weight;
    cout << "Введите ребра в формате <начало> <конец> <вес>: " << endl;
    for (int i = 0; i < numEdges; i++) {
        cin >> source >> destination >> weight;
        graph.addEdge(source, destination, weight);
    }
    graph.printGraph();

    string sourceVertex;
    cout << "Введите начальную вершину для поиска кратчайших путей: ";
    cin >> sourceVertex;
    vector<pair<string, int>> distances = graph.bellmanFord(sourceVertex);
    if (!distances.empty()) {
        cout << "Кратчайшие расстояния от вершины " << sourceVertex << ":" << endl;
        for (int i = 0; i < 9; i++) {
            if (distances[i].second == INF) {
                cout << "Вершина " << distances[i].first << ": Достижима" << endl;
            }
            else {
                cout << "Вершина " << distances[i].first << ": " << distances[i].second << endl;
            }
        }
    }
}
```

Рисунок 15 – Код программы

2.2.4 Результаты тестирования

```
Вершина A: (B, 27) (M, 15)
Вершина B: (A, 27) (G, 9) (L, 7)
Вершина M: (A, 15) (S, 15) (D, 21)
Вершина G: (B, 9) (S, 11) (N, 8)
Вершина L: (B, 7) (N, 10)
Вершина S: (G, 11) (D, 17) (M, 15)
Вершина N: (G, 8) (L, 10) (R, 31)
Вершина D: (S, 17) (M, 21) (R, 32)
Вершина R: (D, 32) (N, 31)
Введите начальную вершину для поиска кратчайших путей: A
Кратчайшие расстояния от вершины A:
Вершина A: 0
Вершина B: 27
Вершина M: 15
Вершина G: 36
Вершина L: 34
Вершина S: 30
Вершина N: 44
Вершина D: 36
Вершина R: 68
```

Рисунок 16 – Результаты тестирования

2.3 ВЫВОДЫ

Цель работы и сопутствующие задачи были успешно выполнены. Освоены приёмы работы с нелинейной структурой графа. Успешно реализована функция нахождения кратчайшего пути методом Беллмана Форда и протестирована на тестовом графе.

СПИСОК ИНФОРМАЦТОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных. Часть 1. Сложность алгоритмов. Сортировки. Линейные структуры данных. Поиск в таблице. : учеб. пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова ; МИРЭА – Российский технологический университет, 2022. – 109 с. – ISBN 978-5-7339-1612-5.

2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01-01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с.