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 Кузнецов Л. А.
группа ИКБО-20-23

Москва 2024

СОДЕРЖАНИЕ

ЧАСТЬ 7.2

Условие

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

Вариант

В данной работе я буду взаимодействовать с графом 18 с произвольными значениями.

Метод решения

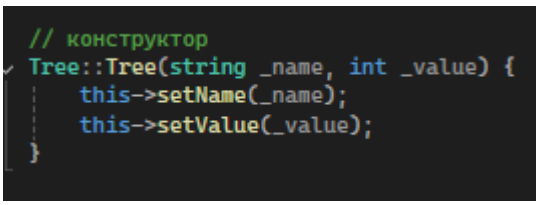
Для начала реализуем создание дерева, его объектов, а также методы для работы с ним (рис. 1).

```
class Tree {
private:
    string name; // уникальное имя листа
    int value; // значение листа
    bool part = false; // является ли объект частью дерева Прима
protected:
    void setName(string _name);// здесь защищённый тип данных, так как не желательно позволять пользователю после задания изменять значение поля name
    void setPart(bool _part) { this->part = _part; }
    bool getPart() { return this->part; }
public:
    bool visited = false; // посещён ли был данный элемент
    vector<Tree*> elements; // список связяннах элементов листа



    //геттеры и сеттеры приватных данных
    void setValue(int _value);
    int getValue();

    string getName();

    // добавить элемент в дерево
    void addElement(string _name, int _value, vector<string> elements_names);

    // переопределение для простоты
    void addElement(Tree* newElement, string headName);

    // конструктор
    Tree(string _name, int _value);

    // вывод дерева
    void out(Tree* node);

    // делаем дерево доступным для изменений
    void reset();

    // делаем из нашего дерева остовное методом Прима
    Tree* makePrim(int finalParts);
};
```

Рисунок 1 – Вёрстка всех методов для работы с поставленной задачей

В поля добавим имя (name), значение (value) ячейки и то, является ли она частью нового дерева (part). Поле visited нужно для работы с деревом, а elements для хранения в нём связанных с ним элементов.

Рассмотрим методы нашего дерева. Начнём с конструктора (рис.2).



```
// конструктор
Tree::Tree(string _name, int _value) {
    this->setName(_name);
    this->setValue(_value);
}
```

Рисунок 2 – Конструктор Tree

Интересным будет отметить тот факт, что в данной программе понадобилось повторное определение добавления элементов для двух случаев (когда мы строим обычное дерево, и когда мы строим остовное дерево по методу Прима) (рис. 3-4).

```
// добавить элемент в дерево
void Tree::addElement(string _name, int _value, vector<string> elements_names) {
    Tree* newElement = new Tree(_name, _value);

    vector<Tree*> tempArray;
    this->visited = true; // делаем элемент посещённым
    tempArray.push_back(this);
    while (tempArray.size() != 0) {
        Tree* tempElement = tempArray[tempArray.size() - 1]; // берём элемент на рассмотрение

        tempArray.pop_back();
        for (Tree* obj : tempElement->elements) {
            if (!obj->visited){
                obj->visited = true;
                tempArray.push_back(obj);
            }
        }

        for (string searched_name : elements_names) // ищем заданные элементы на связь
            if (tempElement->getName() == searched_name) {
                tempElement->elements.push_back(newElement);
                newElement->elements.push_back(tempElement);
                break;
            }
    }

    this->reset();
}
```

Рисунок 3 – 1-ое определение метода добавления элементов в дерево

Проблема с первым определением заключалась в том, что при работе с ним мы знали, куда мы будем подсоединять элемент, однако при создании дерева методом Прима такой информации нет, из-за чего пришлось создать второй метод с похожим функционалом, но совершенно другим подходом к выполнению поставленной задачи.

```cpp
void Tree::addElement(Tree* newElement, string headName) {
    Tree* elem = new Tree(newElement->getName(), newElement->getValue());
    Tree* tempElem;
    vector<Tree*> array;
    array.push_back(this);
    this->visited = true;
    while (array.size() != 0)
    {

        tempElem = array[array.size() - 1];
        tempElem->visited = true;
        array.pop_back();

        if (tempElem->getName() == headName)
        {
            tempElem->elements.push_back(elem);
            elem->elements.push_back(tempElem);
            break;
        }

        for (Tree* obj : tempElem->elements) {
            if (!obj->visited){
                obj->visited = true;
                array.push_back(obj);
            }
        }
    }

    this->reset();
}
```

Рисунок 4 - 2-ое определение метода добавления элементов в дерево

Далее идёт уже скорее вспомогательный метод reset (рис. 5), который нужен для обновления состояния дерева для корректной работы с ним.

```cpp
// делаем дерево доступным для изменений
void Tree::reset() {
    vector<Tree*>tempArray;

    tempArray.push_back(this);
    this->visited = false;
    while (tempArray.size() != 0) {
        Tree* tempElement = tempArray[tempArray.size() - 1]; // берём элемент на рассмотрение

        tempArray.pop_back();

        for (Tree* obj : tempElement->elements) {
            if (obj->visited) {
                obj->visited = false;
                tempArray.push_back(obj);
            }
        }
    }
}
```

Рисунок 5 – Метод обновления состояния дерева

Предпоследний метод – вывод полученного дерева на экран (рис.6). Так как было довольно затруднительно реализовать корректный вывод дерева, чьи элементы могли бы пересекаться друг с другом, я реши изменить вывод таким образом, что поочерёдно показываются все элементы дерева и связанные с ними ячейки.

```
// вывод дерева
void Tree::out(Tree* node) {

    cout << "|-" + node->getName() << endl;
;
    for (Tree* obj : node->elements)
        cout << "   |-" + obj->getName() << endl;

    cout << "  ----  " << endl;
    for (int i = 0; i < node->elements.size(); i++)
    {
        if (!node->elements[i]->visited){
            node->elements[i]->visited = true;
            out(node->elements[i]);
        }
    }
}
```

Рисунок 6 – Метод вывода дерева на экран

И последний метод, реализованный в дереве Tree, - это метод создания остовного дерева при помощи метода Прима (рис. 7). Текущее дерево изменить слишком проблематично, да и при подобном подходе придётся пойти на многие проверки и уступки ради достижения цели, а создание уже нового дерева способно решить подобную проблему.

```
// делаем из нашего дерева остовное методом Прима
Tree* Tree::makePrim(int finalParts) {
    Tree* newTree = new Tree(this->getName(), this->getValue());
    this->setPart(true);
    int countParts = 1;
    vector<Tree*> parts;
    parts.push_back(this);
    Tree* currentPart;
    Tree* futurePart;
    while (countParts != finalParts) {
        currentPart = new Tree("ERROR", 999);
        futurePart = new Tree("ERROR", 0);
        for (Tree* part : parts) {
            for (Tree* obj : part->elements) {
                if (  !obj->getPart() && (abs(part->getValue() - obj->getValue()) < abs(currentPart->getValue() - futurePart->getValue())))
                {
                    futurePart = obj;
                    currentPart = part;
                }
            }
        }

        futurePart->setPart(true);
        newTree->addElement(futurePart, currentPart->getName());

        countParts++;
        parts.push_back(futurePart);
    }

    return newTree;
}
```

Рисунок 7 – Метод создания дерева методом Прима

Мы постепенно проходимся по дереву и выбираем элементы согласно методу Прима.

Осталось только реализовать пользовательский интерфейс и поставленная задача, считай, выполнена (рис. 8).

```
int main() {
    setlocale(LC_ALL, "ru");

    Tree* body = new Tree("a", 87);
    int finalParts = 1;
    int value;
    string name;
    vector<string> elements;
    string tempName = "";
    cout << "Начинаем создание графа" << endl;
    while (name!= "- 1"){
        elements.clear();
        cout << "Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):" << endl;
        cin >> name;
        if (name == "-1")
            break;
        cin >> value;
        finalParts++;
        cout << "Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):" << endl;


        while (true) {
            cin >> tempName;
            if (tempName == "-1")
                break;
            elements.push_back(tempName);
        }
        tempName = "";

        body->addElement(name, value, elements);
    }
    body->visited = true;
    body->out(body);
    body->reset();

    cout << " ||||| " << endl;

    Tree* bodyPrima = body->makePrim(finalParts);
    bodyPrima->visited = true;
    bodyPrima->out(bodyPrima);
    bodyPrima->reset();
```

Рисунок 8 – Пользовательский интерфейс

Сначала мы просим пользователя ввести все необходимы значения, выводим дерево, а после этого преобразуем в остовное дерево при помощи метода Прима. Дабы не быть голословным, проведём тесты соответственно варианту.

Тестирование

Проведём тесты со следующими значениями a=87, b=82, c=82, d=84, e=89, f-83, построив дерево, как на рисунке 9.

Теперь же осуществим ввод значений (рис. 10).

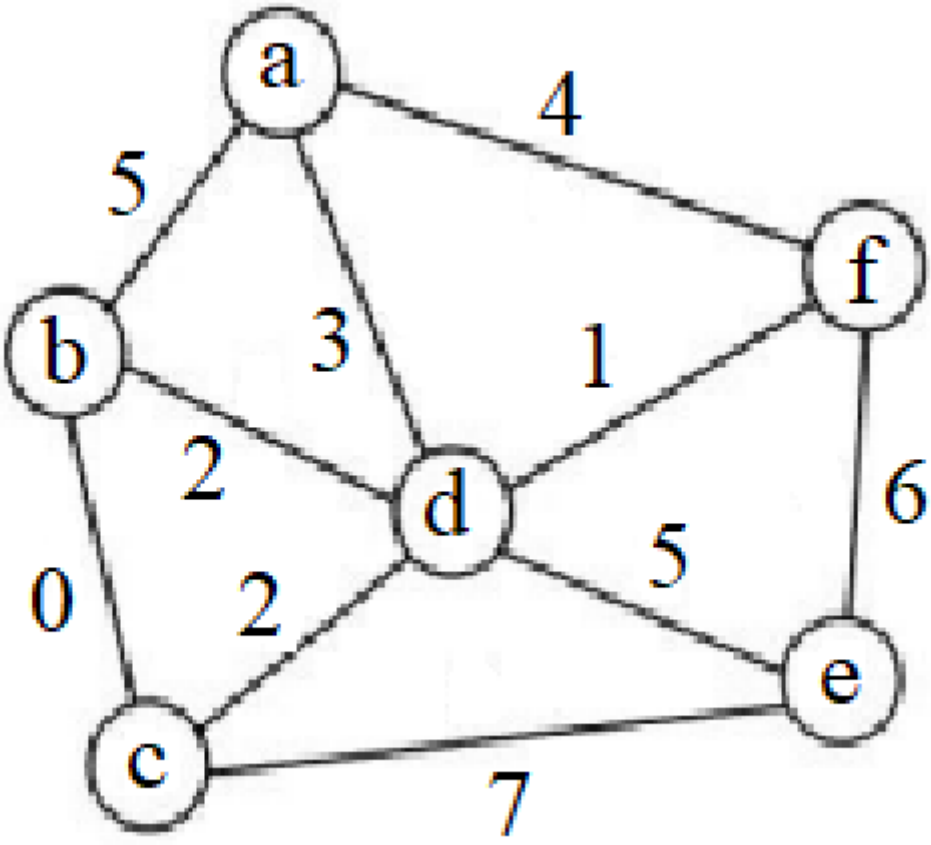


Рисунок 9 – Заданный граф

```
Начинаем создание графа
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
b 82
Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):
a -1
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
c 82
Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):
b -1
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
d 84
Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):
a b c -1
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
e 89
Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):
c d -1
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
f 83
Введите через пробел элементы, которые должны будут быть связаны с этой ячейкой (для остановки введите -1):
a d e -1
Введите имя и значение ячейки через пробел (для остановки работы программы введите -1):
-1
```

Рисунок 10 – Ввод заданных значений

Далее ознакомимся с результатами, приведёнными на рисунке 11 и удостоверимся в правильности как полученного дерева, так и самого графа.

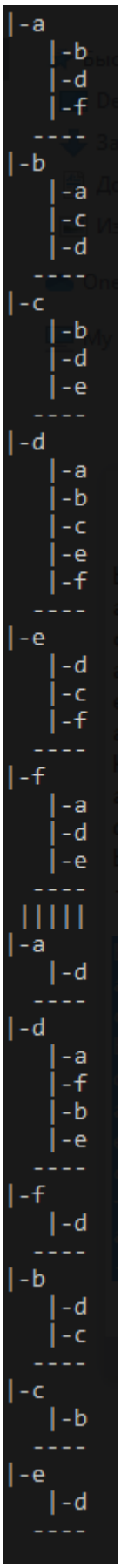


Рисунок 11 – Полученный результат

Как видно из рисунка 11 ожидаемый результат совпал с действительным, поэтому можно смело заявить, что программа работает корректно.

ВЫВОД

Ознакомились с различными видами графов и методами построения разнообразных деревьев, закрепив материал на практике при помощи создания своего дерева по определённой методике.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

2. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).