Курс c++

Царьков Олег

Описание тикета OTSN-13

Граф. Граф — множество вершин и ребер, соединяющих их.

От графа требуется следующее:

• конструктор, в который передается количество вершин и создается пустой граф без ребер.

• функция, добавляющая или убирающая ребро между двумя вершинами

• функция, способная быстро определить соседей вершины (соседи — те вершины, которые соединены с данной ребром, быстро — не перебирая все вершины, которые есть в графе)

Задание 1. Написать дерево.

std::set. std::set — встроенный контейнер, обозначающий множество объектов, никак не упорядоченных между собой. Его преимущество перед вектором — он может определять, есть ли в нем данный элемент за $ln(n)$ шагов.

Вот пример, объясняющий как им пользоваться:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> set;
    set.insert(1);
    set.insert(2);
    set.insert(100);
    std::cout << set.count(100) << std::endl;
    std::cout << set.count(0) << std::endl;
    set.erase(100);
    std::cout << set.count(100) << std::endl;
    return 0;
}
```

На экран выведется

```
1
0
0
```

Еще есть функция *set.clear()*, которая очистит полностью *set*, есть конструктор, принимающий два итератора и заполняющий *set* элементами между этими итераторами.

Поиск в глубину. Имеется задача: разбить граф на компоненты связности. Две вершины лежат в одной компоненте связности тогда и только тогда, когда из одной можно дойти до другой по ребрам.

Для того, чтобы выполнить эту задачу, нужно применить следующий алгоритм:

- мысленно представляем себе, что множество вершин разбито на три класса — белые, серые и черные. Изначально все вершины белые.
- берем любую белую вершину и называем ее черной.
- всех белых соседей черных вершин называем серыми
- всех серых называем черными

Вопрос: почему здесь не сказано "называем всех соседей черных черными зачем нужные серые вершины?

- идем к третьему шагу и выполняем все заново, пока процесс не остановится
- если у черных вершин больше нет белых соседей, то они образуют компоненту связности, отщепляем их, идем к первому шагу и работаем также с оставшимися белыми вершинами.

Задание 2. Написать программу, принимающую на вход дерево(вначале написано количество вершин, потом количество ребер, потом перечисляются пары вершин, которые надо соединить ребрами) и выдающую число, равное количеству его компонент связности.

std::map. std::map находится в файле map и является довольно-таки непростым контейнером.

Рассмотрим такой пример:

```
#include <iostream>
#include <map>

class Pair {
private:
    int a, b;
public:
    Pair()
    : a(0)
    , b(0)
    {}

    Pair(int a, int b)
    : a(a)
    , b(b)
    {}

    bool operator < (const Pair& other) const {
        return (a < other.a || (a == other.a && b < other.b));
    }

    friend std::ostream& operator << (std::ostream& out, const Pair& pair) {
        return out << "(" << pair.a << " " << pair.b << ")";
    }
};

int main() {
    std::map < Pair, int > map;

    map[Pair(3, 4)] = 1;
    map[Pair(5, 6)] = 1;

    for (std::map < Pair, int >::iterator it = map.begin(); it != map.end(); ++it) {
        std::cout << it->first << " " << it->second << std::endl;
    }
    std::cout << map.count(Pair(3, 4)) << std::endl;
    std::cout << map.count(Pair(3, 6)) << std::endl;
    map.erase(Pair(3, 4));
    std::cout << map.count(Pair(3, 4)) << std::endl;
```

```
    return 0;
}
```

Вывод программы таков:

```
(3 4) 1
(5 6) 1
1
0
0
```

Вначале написан класс $Pair$ – класс пар чисел. Хотя такой класс уже есть (std::pair), все равно для показательности нужно его написать самостоятельно.

$$std :: map < Pair,\ int >\ map;$$

Здесь объявляется контейнер map, который может сопоставлять объекту типа $Pair$ объект типа int. На самом деле map хранит в себе множество пар типа $std :: pair < Pair, int >$. Самое примечательное в этом контейнере, что он автоматически создает элементы внутри себя при написании квадратных скобок.

В строчке

$$map[Pair(3,\ 4)]\ =\ 1;$$

сразу создается пара (Pair(3, 4), 1) и запихивается в map.

Когда мы пробегаемся итератором по map и пытаемся напечатать все его элементы, то пишем

$$std :: cout\ <<\ it- > first\ <<\ \text{“ “}\ <<\ it- > second\ <<\ std :: endl;$$

потому что его элементы – пары типа std::pair, у этого класса есть открытые переменные $first$ и $second$. Получаем вывод

```
(3 4) 1
(5 6) 1
```

Для определения, что у map есть какой-то конкретный первый ключ, есть функция map.count, ее использование описано в программе.

Теперь очень важные замечания:

• map не будет работать, если в классе Pair не написать $operator <$, причем его определение должно быть именно таким, каким его хочет видеть map, если хоть где-то забыть $const$, или написать указатель вместо ссылки, или еще что-нибудь, то не надо удивляться что он выдаст страшную длинную ошибку, текст которой не читаем(хотя в середине можно увидеть что он ищет $operator <$)

• Этот оператор нельзя писать кое-как, вот пример программы:

```
#include <iostream>
#include <map>

class Pair {
private:
    int a, b;
public:
    Pair()
    : a(0)
    , b(0)
    {}

    Pair(int a, int b)
    : a(a)
    , b(b)
    {}

    bool operator < (const Pair& other) const {
        return a < other.a;
    }
};

int main() {
    std::map<Pair, int> map;
    map[Pair(3, 4)] = 1;
    std::cout << map.count(Pair(3, 4)) << std::endl;
    std::cout << map.count(Pair(3, 6)) << std::endl;
    return 0;
}
```

Неожиданно вывод таков:

```
1
1
```

То есть программа считает, что ключи $Pair(3,4)$ и $Pair(3,6)$ оба лежат в map. Почему? Программа начинает искать ключ $Pair(3,6)$. Вот она нашла $Pair(3,4)$ и проверяет:

1) $Pair(3,4) < Pair(3,6)$? да нет, не меньше
2) $Pair(3,6) < Pair(3,4)$? нет, опять не меньше.
3) Ну раз так, значит ключ найден.

map не пользуется operator== и operator!=, даже если их дописать в классе, эта ошибка никуда не устранится. Надо писать *operator* $<$ так, что если элементы не равны, то какой-то меньше какого-то.

Поиск в ширину. Дан граф, требуется найти наименьшую длину пути между вершинами с номерами 1 и 2.

Алгоритм: взять вершину 1, пометить ее нулем — это длина ее наименьшего пути до себя. Потом ее соседей пометить 1 – это их наименьшая длина пути до первой вершины(куда уж меньше), соседей этих вершин пометить 2 – для них это наименьший путь до первой вершины(так как если бы был путь меньше, они бы уже были помечены на каком-то их предыдущих шагов), и так далее. Остановиться, когда вершина 2 помечена каким-то числом, выдать это число в качестве ответа.

Задание 3. Реализовать алгоритм.

Граф с весами

Граф с весами – такой же граф + его ребрам сопоставлены числа.

Задагие 4. Написать граф с весами, в котором будут следующие фукнции:

- конструктор, принимающий число вершин и строящий дерево без ребер
- функция, принимающая номера вершин, которые надо соединить и вес, который надо приписать этому ребру.
- функция ввода с экрана, которая ожидает ввод размера дерева, количества его ребер и троек чисел “вершина 1, вершина 2, вес“.

Задание 5. Изучить и реализовать алгоритм Дейкстры поиска пути с наименьшей суммой весов ребер в нем.