Факультет программной инженерии и компьютерной техники

Алгоритмы и структуры данных

Структуры данных

Преподаватели: Косяков Михаил Сергеевич, Тараканов Денис Сергеевич

Выполнил: Кульбако Артемий Юрьевич

Р3212

# 1067. Disk Tree

#include <iostream>

#include <string>

#include <sstream>

#include <map>

using namespace std;

class Dir {

    private: map<string, Dir\*> childDirs;

    public:

        Dir() {}

        Dir\* getDir(string name) {

         if (childDirs.find(name) != childDirs.end()) return childDirs[name];

            else return createDir(name);

        }

        Dir\* createDir(string name) {

            childDirs[name] = new Dir();

            return childDirs[name];

        }

        void printTree(string separator = "") {

            string tabs = " ";

            tabs += separator;

            map<string, Dir\*> contents(childDirs.begin(), childDirs.end());

            for (auto it = contents.begin(); it != contents.end(); it++) {

                cout << separator << it->first << endl;

                it->second->printTree(tabs);

            }

        }

};

int main() {

    int n;

    cin >> n;

    Dir\* root = new Dir();

    for (int i = 0; i < n; i++) {

        Dir\* currentDir = root;

        string fullPath, name;

        cin >> fullPath;

        stringstream ss(fullPath);

        while (getline(ss, name, '\\'))

            currentDir = currentDir->getDir(name);

    }

    root->printTree();

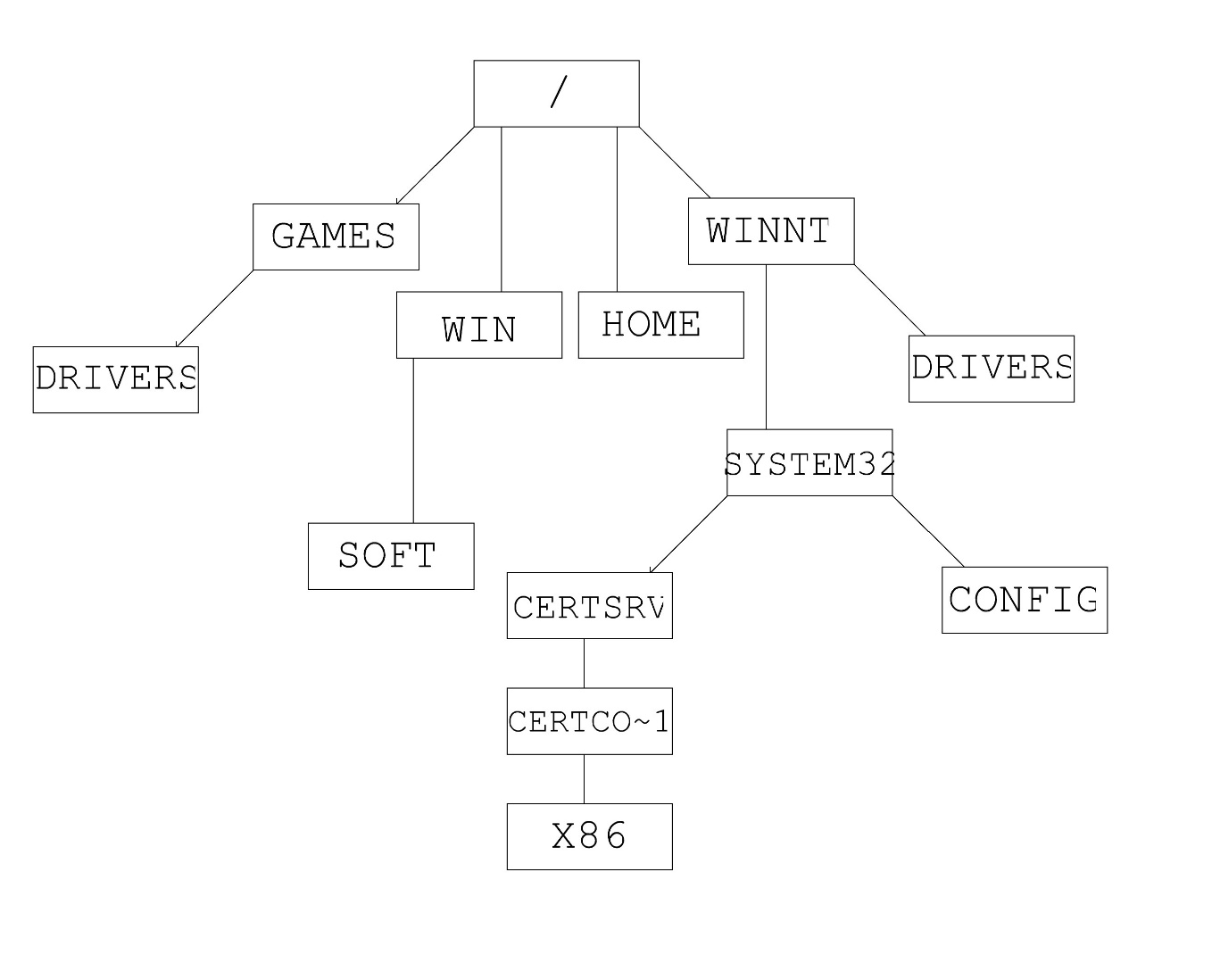
    return 0;

}

Нужно построить и вывести дерево каталогов, по аналогии с командой tree. Самый простой вариант - сохранить все строки в массив, парсить по первому символу \ и создавать новую цепочку в связанном списке. Но это крайне тяжёлый и долгий для компьютера способ. Нам необходимо создавать нужную структуру каталогов "на лету" - сразу всё структуру для полученной строки. Так-как файловая система представляет из себя древовидную структуру, будет удобно использовать Map, где ключом выступит имя каталога, а значением - вложенные каталоги.

1. Создадим объект Dir. Он будет инкапсулировать методы создания, получения, вывода директорий.
2. Получим из исходной строки fullPath строку name, представляющую полный путь к каталогу и часть имени, до первого вхождения \.
3. Проверим, существует ли в текущем каталоге (изначально он будет корневым), директория с именем name. Если "нет" - создадим, если "да", сделаем её текущим каталогом и получим следующую часть fullPath до символа \.
4. Когда дерево для fullPath будет полностью заполнено, мы снова вернёмся в корень нашей "файловой системы", и повторим пункты 1-3 для новой fullPath.

Теперь надо вывести полученную структуру. Для этого, будем рекурсивно опускаться вглубь каждой директории, наращивая на каждом уровне количество пробельный отступов.



# 1494. Монобильярд

#include <iostream>

#include <stack>

using namespace std;

int main() {

    int n, max = 0;

    cin >> n;

    stack<int> balls;

    for (int i = 0; i < n; i++) {

        int currentBall;

        cin >> currentBall;

        if (currentBall > max) {

            for (int j = max + 1; j <= currentBall - 1; j++) balls.push(j);

            max = currentBall;

        } else {

            if (currentBall == balls.top()) balls.pop();

            else {

                cout << "Cheater" << endl;

                return 0;

            }

        }

    }

    cout << "Not a proof" << endl;

    return 0;

}

Основная сложность задачи - понять крайне запутанное условие. Возьмём множество шаров

Игроку нужно забивать шары строго в порядке возрастания номеров. Значит, ревизору нужно доставать их строго в обратном порядке, чтобы подтвердить честность игрока. При этом, ревизор может подойти в любой момент времени. Первый раз он подходит и достаёт шар и предполагает, что Чичиков забил все шары. Когда он подойдёт следующий раз и вытащит шар , есть 3 варианта развития событий:

Ревизор может предположить, что Чичиков забил все шары на промежутке между , и это не будет являться доказательством виновности.

За время отсутствия ревизора, Чичиков ничего не забил, а ревизор просто достал предыдущий шар. Обвинить Чичикова нельзя.

Значит ревизор достал шар, на несколько номером меньше, чем предыдущий взятый. Значит, за время отсутствия ревизора, Чичиков забил шар с меньшим номером, или не забивал вообще, а нарушил порядок раньше. Это будет являться доказательством его виновности.

    3 4 2 1 5 6

    Достаём шар 3 -> предполагаем, что Чичиков забил 1..3

    Достаём шар 4 -> Чичиков забил 4

    Достаём шар 2 -> Чичиков ничего не забил за время отсутствия ревизора

    Достаём шар 1 -> Чичиков ничего не забил за время отсутствия ревизора

    Достаём шар 5 -> Чичиков забил 5

    Достаём шар 6 -> Чичиков забил 6

    НЕТ ДОКАЗАТЕЛЬСТВ ВИНЫ

    3 1 2

    Достаём шар 3 -> предполагаем, что Чичиков забил 1..3

    Достаём шар 1 -> если Чичиков забивал 3, значит по правилам, до него должен был забит 2, а тут 1

    ЧИЧИКОВ ЖУЛЬНИЧАЕТ

Хорошее моделирование ситуации: <https://acm.timus.ru/forum/thread.aspx?id=42110&upd=636971955162244322>

Реализовать такую проверку ревизором можно через структуру данных - стек. Достанем и положим в стек, все шары, которые должен был забить Чичиков (), если является . Иначе проверим, является ли взятый шар ожидаемым, т.е. для шара , шар с вершины стека должен быть . Если условие выполняется, то всё в порядке, иначе вынесем обвинительный приговор.

# 1521. Военные учения 2

#include <iostream>

using namespace std;

class FlavicksTree {

    private:

        pair<int, int>\* soldiers;

        int z = 1;

    public:

        FlavicksTree(int n) {

            soldiers = new pair<int, int>[4 \* n];

    }

        void createNode(int v, int tl, int tr) {

            if (tl == tr) {

                soldiers[v] = make\_pair(1, z++);

                return;

            }

            int tm = (tl + tr) / 2;

            createNode(2 \* v, tl, tm);

            createNode(2 \* v + 1, tm + 1, tr);

            soldiers[v].first = soldiers[2 \* v].first + soldiers[2 \* v + 1].first;

            soldiers[v].second = -1;

        }

        int modify(int v, int tl, int tr, int n) {

            if (tl == tr) {

                --soldiers[v].first;

                return soldiers[v].second;

            }

            int tm = (tl + tr) / 2;

            soldiers[v].first--;

            if (soldiers[2 \* v].first >= n) modify(2 \* v, tl, tm, n);

            else modify(2 \* v + 1, tm + 1, tr, n - soldiers[2 \* v].first);

        }

};

int main() {

    int n, k;

    cin >> n >> k;

    FlavicksTree\* tree = new FlavicksTree(n);

    tree->createNode(1, 1, n);

    int current = k;

    for (int i = 0; i < n; i++) {

        int dead = tree->modify(1, 1, n, current);

        cout << dead << " ";

        if (i == n - 1) break;

        current = (current - 1 + k) % (n - 1 - i);

        if (current == 0) current += n - 1 - i;

    }

    return 0;

}

Это модификация задачи Иосифа Флавия, с той лишь разницей, что вывести нужно не номер выжившего, а номера всех в порядке умерщвления. Для решения, попытаемся найти зависимость ответа от n (количества солдат) и k (разница между номера убийцы и убиваемого).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n\k | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 |
| 3 | 3 | 3 | 2 | 2 | 1 | 1 | 3 | 3 | 2 | 2 |
| 4 | 4 | 1 | 1 | 2 | 2 | 3 | 2 | 3 | 3 | 4 |
| 5 | 5 | 3 | 4 | 1 | 2 | 4 | 4 | 1 | 2 | 4 |
| 6 | 6 | 5 | 1 | 5 | 1 | 4 | 5 | 3 | 5 | 2 |
| 7 | 7 | 7 | 4 | 2 | 6 | 3 | 5 | 4 | 7 | 5 |
| 8 | 8 | 1 | 7 | 6 | 3 | 1 | 4 | 4 | 8 | 7 |
| 9 | 9 | 3 | 1 | 1 | 8 | 7 | 2 | 3 | 8 | 8 |
| 10 | 10 | 5 | 4 | 5 | 3 | 3 | 9 | 1 | 7 | 8 |

Видна закономерность решения:

Значит, рекурсивная реализация, возвращающая номер выжившего, будет выглядеть так (асимптотика ):

int solve(int n, int k) {return n > 1 ? (joseph (n-1, k) + k - 1) % n + 1 : 1;}

Так-как нам нужно вывести не выжившего, а последовательность убывания, задача становится значительно сложнее. Всех солдат нужно хранить в какой-либо структуре данных. Нам придётся n раз проходить по кругу из солдат, из-за чего обычный массив отработает за . Попробуем уменьшить количество операций алгоритма, воспользовавшись структурой данных - дерево отрезков, которое даст нам . Дерево можно реализовать в виде массиве, где каждая ячейка является узлом. На нулевом уровне дерева запросом затрагивается единственная вершина — корень дерева. Дальше на первом уровне рекурсивный вызов в худшем случае разбивается на два рекурсивных вызова, но важно здесь то, что запросы в этих двух вызовах будут соседствовать, т.е. число запроса во втором рекурсивном вызове будет на единицу больше числа запроса в первом рекурсивном вызове. Отсюда следует, что на следующем уровне каждый из этих двух вызовов мог породить ещё по два рекурсивных вызова, но в таком случае половина этих запросов отработает нерекурсивно, взяв нужное значение из вершины дерева отрезков. Таким образом, всякий раз у нас будет не более двух реально работающих ветвей рекурсии (можно сказать, что одна ветвь приближается к левой границе запроса, а вторая ветвь — к правой), а всего число затронутых отрезков не могло превысить высоты дерева отрезков, умноженной на четыре, т.е. оно есть число .

Каждый узел дерева будет хранить номер убийцы, и номер убиваемого. При модификации будем получать номер нового убиваемого, и изменять узлы дерева. Сделать это нужно n раз, финальная сложность = .

# 1650. Миллиардеры

*#include* <map>

*#include* <iostream>

*#include* <string>

*#include* <set>

*#include* <unordered\_map>

using namespace std;

int main() {

    unordered\_map<string, long long> cityAndMoney;*// Город и его капитализированные деньги*

    unordered\_map<string, long long> richmanAndMoney;*// Богач и его сумма денег*

    unordered\_map<string, string> richmanAndCity;*// Богач и его местоположение*

    map<long long, set<string>> moneyAndCitiesSet;*// Деньги и города, с такой суммой*

    map<string, int> citiesRank;*// Финальный топ городов*

    long long money;

    string name, city;

    int n;

    cin >> n;

*for* (int i = 0; i < n; i++) {

        cin >> name >> city >> money;

        richmanAndMoney[name] = money;*//заполнение карты связью {Богач-Деньги}*

        richmanAndCity[name] = city;*//заполнение карты связью {Богач-Город}*

*if* (cityAndMoney.find(city) != cityAndMoney.end()) {

*/\**

*Если город city существует, значит над ним мы проводим операцию не первый раз, значит его*

*капитализация изменится. Отчистим в карте {Капиталиция-Города} связь (её больше не будет существовать).*

*\*/*

            long long sum = cityAndMoney[city];

            moneyAndCitiesSet[sum].erase(city);

*//Если города с такой капитализацией перестали существовать, отчистим и ячейку с множеством городов.*

*if* (moneyAndCitiesSet[sum].size() == 0) moneyAndCitiesSet.erase(sum);

        }

        cityAndMoney[city] += money;*//увеличение капитализации города*

        moneyAndCitiesSet[cityAndMoney[city]].insert(city);*//заносим город в множество городов с такой капитализацией*

    }

    int days, movements, day, prevDay, currDay = 0;

    cin >> days >> movements;

*for* (int i = 0; i <= movements; i++) {*//вычислим капитализации для каждого из городов в разные дни*

        prevDay = currDay;

*//Обрабатываем ситуацию последнего дня, когда ещё нужно менять параметры сущностей, но данные вводить уже не надо.*

*if* (i == movements) day = days;

*else* cin >> day >> name >> city;

        currDay = day;

        map<long long, set<string>>:: reverse\_iterator it = moneyAndCitiesSet.rbegin();

*/\**

*Если в отсортированном по ключам контейнере {Деньги-Список городов} в ячейке, соответствующей наибольшему ключу*

*только один элемент в {set<string>}, значит сейчас существует единственный город с максимальной суммой, и ему*

*нужно добавить дни в карту {Город-Количество дней в топе}.*

*\*/*

*if* (currDay != prevDay && it->second.size() == 1) citiesRank[\*(it->second.begin())] += currDay - prevDay;

*if* (i < movements) {

*/\**

*Старое местоположение богача: если он уехал из города, необохдимо поменять соответствующую*

*информацию во всех картах.*

*\*/*

            string oldLocation = richmanAndCity[name];

            long long oldMoney = cityAndMoney[oldLocation];

            moneyAndCitiesSet[oldMoney].erase(oldLocation);

*if* (moneyAndCitiesSet[oldMoney].size() == 0) moneyAndCitiesSet.erase(oldMoney);

            cityAndMoney[oldLocation] -= richmanAndMoney[name];

            moneyAndCitiesSet[cityAndMoney[oldLocation]].insert(oldLocation);

*/\**

*Новое местоположение богача: если он приехал в новый город, необходимо поменять соответствующую*

*информацию во всех картах.*

*\*/*

            long long newMoney = cityAndMoney[city];

            moneyAndCitiesSet[newMoney].erase(city);

*if*(moneyAndCitiesSet[newMoney].size() == 0) moneyAndCitiesSet.erase(newMoney);

            cityAndMoney[city] += richmanAndMoney[name];

            moneyAndCitiesSet[cityAndMoney[city]].insert(city);

            richmanAndCity[name] = city;

        }

    }

*for* (const auto& c: citiesRank) cout << c.first << " " << c.second << endl;*//вывод топа городов*

*return* 0;

}

Как таковой сложности в разрабатываемом алгоритме в задаче нет. Нам просто нужно аккуратно манипулировать всеми данными и поддерживать связь между ними. Первое делается в лоб в одном цикле по мере чтения данных, а вот второе уже сложнее. Связи должны поддерживается между всеми сущностями: {Деньгами}, {Богачами}, {Городами}. Для них подходит структуры данных map или unordered\_map, где одна из сущностей будет ключом, а другая - значением. Таких карт должно быть несколько, для всех необходимых связей.