1. Определите АТД для хранения информации о таблице базы данных: столбцы (количество и названия столбцов), строки со значениями (все значения в строке таблицы имеют тип string), ключи для получения строк таблицы в некотором порядке (для упрощения ключ включает только один столбец, т.е. ключ - это имя или номер столбца, а строки таблицы можно получать последовательно по одной в порядке возрастания значения в указанном столбце). Перечислите методы АТД, обеспечивающие последовательный доступ к информации и её изменение, аргументы и возвращаемые значения каждого метода с комментариями.

Методы структуры данных:

* Получить Количество столбцов.
* Получить Количество строк.
* Получить название всех столбцов - Возвращает список названий столбцов в соответствующем порядке.
* Добавить столбец (название столбца).
* Удалить столбец (Название столбца или номер).
* Добавить строку (список значений).
* Удалить строку (номер строки).
* Получить всю строку (номер строки) - Возвращает список значений в соответствующей строке.
* Получить значение у указанной колонки и строки (название или номер столбца, номер строки) - возвращает значение в соответствующем поле.

1. Предложите структуры данных для представления АТД из задания 3. Перечислите поля, их типы и комментарии к каждому полю. Укажите оценку эффективности (амортизированную или среднюю) для каждого метода с учетом использованных структур данных. Хранимая в структуре информация не должна дублироваться. Это отдельная задача. Не объединяете с задачей 3! Не пишите реализацию методов, нужно указать только оценку эффективности.

Пусть n - количество строк, m - количество столбцов (для удобства)

**Медленная реализация**

Массиве массивов строк (vector<vector<string>>),

для этого нужно будет хранить ещё массив названий столбцов vector<string>.

* Получить Количество столбцов O(1)
* Получить Количество строк O(1)
* Получить название всех столбцов O(1)
* Добавление столбца O(n) (амортизированная)

Добавление столбца будет работать так: вставим название столбца в конец массива названий столбцов O(1), после этого в конец каждой строки добавим элемент для хранения значения в соответствующем столбце O(n \* 1) (O(1) вставка в вектор амортизированная)

* Удалить столбец O(n \* m)

Сначала находим наш столбец в списке столбцов O(m), либо O(1) в случае если аргумент - номер столбца, после этого удаляем элемент по найденному индексу в массиве O(m). Далее нужно удалить по такому же индексу элементы в массивах строк.

В общем удаление столбца займет O(m или 1) + O(m) + O(n \* m) ~ O(n \* m)

* Добавить строку - O(1) (амортизированная)
* Удалить строку - O(n) (Удалить значение в vector по соотв индексу)
* Получить всю строку - O(1)
* Получить значение у указанной колонки и строки - O(m) (если столбец задан номером то O(1))

**Быстрая реализация**

Реализация на хеш-таблицах (unordered\_map<string, unordered\_map<int, string>>), в таком случае у нас появляется возможность хранить id строк (этого не было в задании, но мне кажется это правильно), для того чтобы хранить максимальное id нужна будет дополнительная переменная

* Получить Количество столбцов O(1)
* Получить Количество строк O(1)
* Получить название всех столбцов O(m) (собрать все названия в вектор и вернуть, можно хранить этот вектор и возвращать за O(1), но тогда нужно будет O(m) памяти)
* добавление столбца - O(1)

Создадим пустой unordered\_map по ключу - название столбца, заполнять пустыми значениями не будем, будет заполняться само при обращении к пустым элементам

* Удаление столбца O(m) (если не учитывать время которое тратится на удаление хеш-таблицы)
* Добавить строку - O(m)
* Удалить строку - O(m) - Удалить значения по соответствующему id в unordered\_map каждого столбца
* Получить всю строку - O(m) - собрать все значения в вектор и вернуть
* Получить значение у указанной колонки и строки - O(m) (если столбец задан строкой то O(1))

11. Определить АТД Разреженная матрица, обеспечивающий метод get(i, j) для получения элемента матрицы и set(i, j, v) для изменения (добавления) ненулевого элемента. В конструкторе задаются размеры матрицы.

Реализовать АТД через словарь по ключам map<pair<int, int>, double>.

class SparseMatrix

{

    int n, m;

    map<pair<int, int>, double> matrix;

public:

    SparseMatrix(int n, int m) : n(n), m(m) {}

    double get(int i, int j)

    {

        if (i < 0 || i >= n || j < 0 || j >= m)

            throw runtime\_error("Invalid index");

        if (matrix.count({i, j}))

            return matrix[{i, j}];

        return 0;

    }

    void set(int i, int j, int v)

    {

        if (i < 0 || i >= n || j < 0 || j >= m)

            throw runtime\_error("Invalid index");

        if (v == 0)

            matrix.erase({i, j});

        else

            matrix[{i, j}] = v;

    }

};

Определить эффективность операций + и \* в зависимости от количества ненулевых элементов K.

Операция + будет реализован двумя указателями и будет работать за O(K1 + K2)

Операция \* будет делать полный перебор и использовать метод get который работает за O(log2(K)), поэтому общее время работы алгоритма будет:

O(n \* m \* (n + m) \* log2(K1 \* K2))

Для каждого элемента исходной матрицы мы должны посчитать значение, это значит что их придется перебрать O(n\*m). Для каждого такого элемента мы должны перебрать строку и столбец O(n + m) и для каждой итерации такого перебора должны сделать get из первой и второй матриц это O(log2(K1) + log2(K2)).

12. Определить АТД Матрица, обеспечивающий метод [*i, j*] для доступа к элементам матрицы. В конструкторе задаются размеры матрицы.

Реализовать матрицу через vector размером N⋅M. Определить операцию +. Сравнить время сложения матриц размером 1000×1000, меняя порядок циклов (строки/столбцы и столбцы/строки) для уровня оптимизации O3. Результаты записать в таблицу, в которой будет указан порядок выполнения циклов и время выполнения в мкс.

class Matrix

{

    int n, m;

    vector<double> v;

public:

    Matrix(int n, int m) : n(n), m(m)

    {

        v.resize(n \* m);

    }

    double &operator[](pair<int, int> index)

    {

        auto [i, j] = index;

        if (i < 0 || i > n || j < 0 || j > m)

            throw runtime\_error("Invalid Index");

        return v[i \* n + j];

    }

    Matrix operator+(const Matrix &other)

    {

        if (n != other.n || m != other.m)

            throw runtime\_error("Invalid Sizes");

        Matrix answer(n, m);

        for (int i = 0; i < n; i++)

            for (int j = 0; j < m; j++)

                answer[{i, j}] = v[i \* n + j] + other.v[i \* n + j];

        return answer;

    }

};

|  |  |
| --- | --- |
| for(i){ for(j) } | for(j){ for(i) } |
| 36760 микросекунд | 38213 микросекунд |

(Средние за 100 попыток)

13. Используя поиск в глубину, определите число компонент связности в графе, задаваемом следующим образом:

Как набор кругов с центром (Xi,Yi) диаметром D. Круги считаются связными, если они накладываются друг на друга или соприкасаются краем.

class Circle

{

    double x, y, r;

public:

    Circle(double x, double y, double r) : x(x), y(y), r(r) {}

    bool isTouch(const Circle &other)

    {

        return hypot(other.x - x, other.y - y) <= r + other.r;

    }

};

int main()

{

    vector<Circle> circles;

    vector<vector<int>> graph;

    int n;

    cin >> n;

    circles.reserve(n);

    graph.resize(n);

    for (int i = 0; i < n; i++)

    {

        double x, y, r;

        cin >> x >> y >> r;

        Circle circle(x, y, r);

        for (int j = 0; j < circles.size(); j++)

            if (circles[j].isTouch(circle))

            {

                graph[i].push\_back(j);

                graph[j].push\_back(i);

            }

        circles.push\_back(circle);

    }

    vector<bool> visited(n);

    int cnt = 0;

    deque<int> q;

    for (int i = 0; i < n; i++)

    {

        if (!visited[i])

        {

            cnt++;

            q.push\_back(i);

            while (q.size())

            {

                int c = q.front();

                q.pop\_front();

                visited[c] = true;

                for (int i = 0; i < graph[c].size(); i++)

                    if (!visited[graph[c][i]])

                        q.push\_back(graph[c][i]);

            }

        }

    }

    cout << cnt << '\n';

    return 0;

}

16. На предприятии есть n𝑛 (2 ≤ n ≤ 10000) отделов, между которыми нужно проложить сеть. Есть m (2 ≤ m ≤ 100000) вариантов для соединения отделов между собой, описываемых тройкой ai, bi, di, где ai, bi - номера отделов, di - длина кабеля. Для построения сети необходимо, чтобы все отделы прямо или косвенно через другие отделы были соединены между собой.  
Укажите какой алгоритм построения минимального остовного дерева является более эффективным для решения этой задачи и обоснуйте свой выбор.

Я знаю всего 2 алгоритма для построения остовного дерева – Алгоритм Прима и Крускала.

Алгоритм Прима более эффективен, когда граф насыщен (то есть количество ребер ~ квадрату количества вершин), а Алгоритм Крускала наоборот более эффективен на разреженных графах (то есть количество ребер ~ количества вершин).

В данном случае лучше использовать Алгоритм Крускала потому что граф разрежен.

17. Укажите какой алгоритм нужно использовать для решения задачи и обоснуйте свой выбор. Из каких вершин и ребер граф будет состоять?  
О последовательности A1, ..., AN (1 ≤ N ≤ 200), в которой каждое число от 1 до N встречается ровно один раз, известно M (0 ≤ M ≤ 40000) ограничений вида:  
1 i j x – наибольшее число в позициях между i и j (включительно) равно x  
2 i j y – наименьшее число в позициях между i и j (включительно) равно y  
Выведите любой вариант для такой последовательности.

Будем использовать дерево отрезков, в вершинах будем хранить максимальный и минимальный элемент на отрезке с l по r. В ребрах будем хранить “обещание” на изменение максимального либо минимального элемента на подотрезке. Если будем приходить в вершину то всегда сначала выполним это обещание и передадим его “детям” вершины.

Когда будем заполнять массив, который будет ответом, сначала заполним используя минимальные и максимальные элемента помечая их “посещенными” потом заполним оставшиеся элементы рандомными которые подойдут под ограничения и так же будем отмечать их “посещенными” чтобы не использовать повторно.

Сложность алгоритма O(M \* logN + N)

20. Сравните время сортировки с помощью sort, stable\_sort, make\_heap/sort\_heap для вектора из 106 случайных чисел. Результаты оформить в виде таблицы.

Код для тестов:

#include <iostream>

#include <chrono>

#include <algorithm>

#include <vector>

#include <ctime>

using namespace std;

vector<int> generate\_vector()

{

    vector<int> ans;

    for (int i = 0; i < 1e6; i++)

        ans.push\_back(rand());

    return ans;

}

int test(vector<int> &arr)

{

    std::chrono::steady\_clock::time\_point begin, end;

    begin = std::chrono::steady\_clock::now();

    sort(arr.begin(), arr.end());

    // stable\_sort(arr.begin(), arr.end());

    // make\_heap(arr.begin(), arr.end());

    // sort\_heap(arr.begin(), arr.end());

    end = std::chrono::steady\_clock::now();

    return chrono::duration\_cast<std::chrono::microseconds>(end - begin).count();

}

int main()

{

    int sum = 0;

    for (int i = 0; i < 10; i++)

    {

        vector<int> arr = generate\_vector();

        int cnt = test(arr);

        sum += cnt;

    }

    cout << "Avarage: " << sum / 10 << '\n';

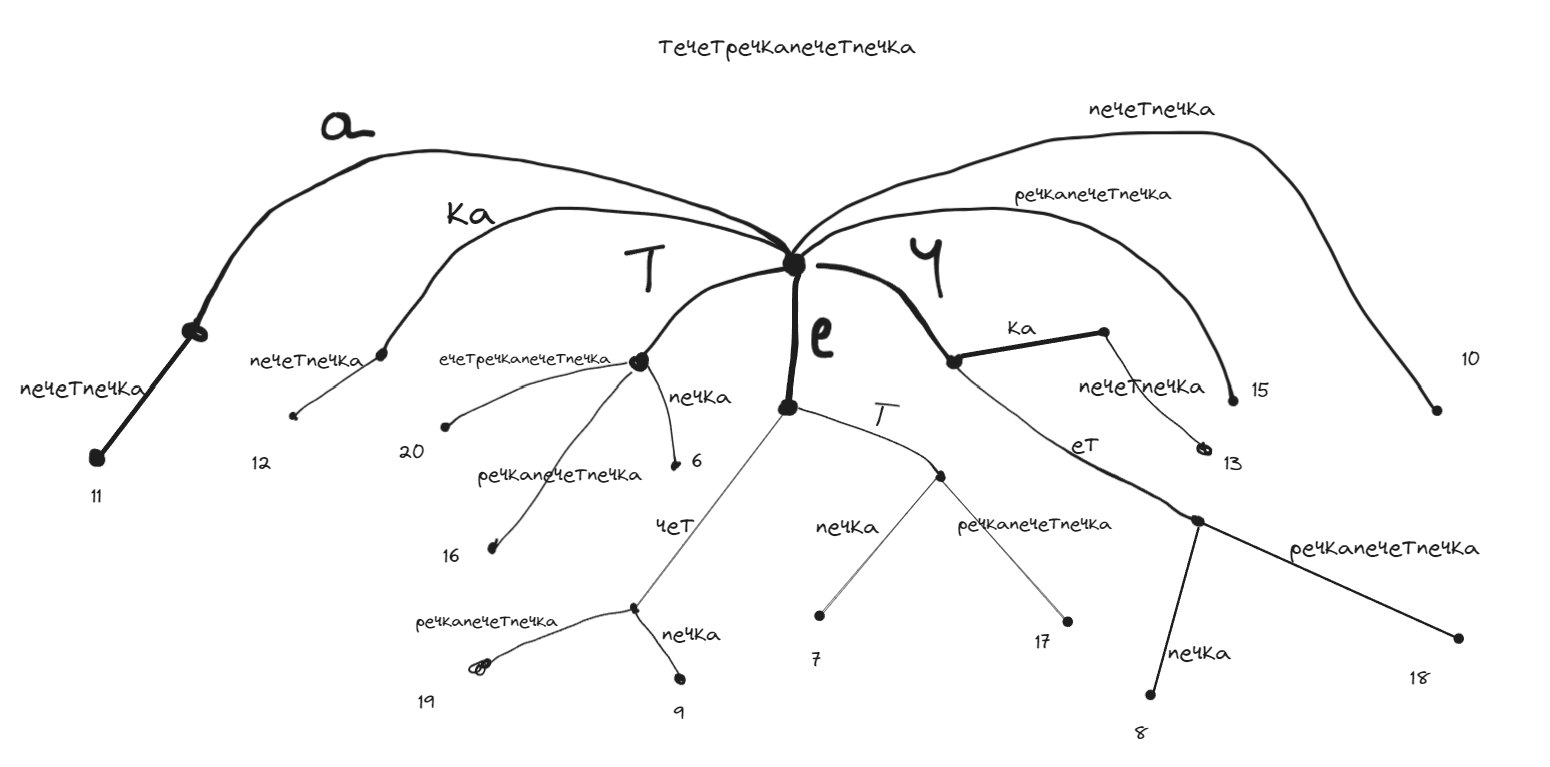
    return 0;

}

Результаты в микросекундах:

|  |  |  |
| --- | --- | --- |
| sort | stable\_sort | make\_heap/sort\_heap |
| 241206 | 278574 | 542811 |

22. Постройте сжатое суффиксное дерево для строки "течетречкапечетпечка" и найдите количеcтво различных подстрок в этой строке. Объясните способ подсчета с использованием суффиксного дерева.



Количество подстрок: 181

Как считаем: идём по каждому ребру и к результату прибавляем длину строки в этом ребре. Фактически получается что мы считаем количество подстрок начинающихся на уникальную подстроку.

23. Определите необходимые геометрические объекты и напишите следующую функцию  
В декартовой системе координат на плоскости заданы координаты концов отрезка и ещё одной точки. Определить расстояние от точки до отрезка.

#include <iostream>

#include <cmath>

using namespace std;

struct Point

{

    double x, y;

    double len() const { return hypot(x, y); }

    Point operator-(Point p) const

    {

        return {x - p.x, y - p.y};

    }

    double operator\*(Point p) const

    {

        return x \* p.x + y \* p.y;

    }

    double operator^(Point p) const

    {

        return x \* p.y - y \* p.x;

    }

};

double distance(Point A, Point B, Point C)

{

    Point AB = B - A;

    Point BC = C - B;

    Point AC = C - A;

    double AB\_BC = (AB \* BC);

    double AB\_AC = (AB \* AC);

    if (AB\_BC > 0)

        return (C - B).len();

    else if (AB\_AC < 0)

        return (C - A).len();

    return abs((AB ^ AC) / (AB \* AC));

}

int main()

{

    Point A, B, C;

    cin >> A.x >> A.y;

    cin >> B.x >> B.y;

    cin >> C.x >> C.y;

    cout << distance(A, B, C);

    return 0;

}

AB – Отрезок, C – точка