Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт № 8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Курсовой проект по курсу

«Дискретный анализ»

Студент: Павловский А.В.
Группа: М8О-201Б-21
Преподаватель: Макаров Н.К.
Оценка: ___
Дата:_____
Подпись: _____

Москва, 2022

Содержание

1. Цель работы

Реализовать персистентную структуру данных на Си++.

2. Постановка задачи

Вам дан набор горизонтальных отрезков, и набор точек. Для каждой точки определите сколько отрезков лежит строго над ней.
Ваше решение должно работать online, то есть должно обрабатывать запросы по одному после построения необходимой структуры данных по входным данным. Чтение входных данных и запросов вместе и построение по ним общей структуры запрещено.

3. Общие сведения о программе

Программа представляет из себя файл kp.cpp, собирается программа при помощи команд g++ kp.cpp запускается при помощи команды ./a.out

Формат ввода: В первой строке вам даны два числа n и m ($1 \le n, m \le 10^5$) — количество отрезков и количество точек соответственно. В следующих n строках вам заданы отрезки, в виде троек чисел l, r и h($-10^9 \le l < r \le 10^9, -10^9 \le h \le 10^9$) — координаты x левой и правой границ отрезка и координата y отрезка соответственно. В следующих m строках вам даны пары чисел x, y ($-10^9 \le x, y \le 10^9$) — координаты точек.

Формат вывода: Для каждой точки выведите количество отрезков над ней.

4. Общий метод и алгоритм решения

Сначала я считываю n отрезков из командной строки в структуру, затем я сортирую структуры по ключу h так, чтобы отрезки хранились от большего h к меньшему.
Я строю параллельно два персистентных дерева по r и по l каждого отрезка. После каждой вставки элемента в дерево создается новая версия, а старая версия дерева сохраняется. Таким образом, на k-ом шаге алгоритма я имею k версий моих двух деревьев.
Для подсчета ответа я храню размер каждого элемента дерева, он состоит из суммы размеров его поддеревьев.
Далее считаю ответ.

Исходный код

kp.cpp

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <vector>
#include <algorithm>
#include <set>

using namespace std;

struct Node
{
    int key;
    int x1;
    int x2;
    int height;
    int size;
    int count_same;
    Node* right;
    Node* left;

    Node(int key, int x1, int x2) : key(key), x1(x1), x2(x2) {
        height = 1;
        size = 1;
        count_same = 1;
        left = right = nullptr;
    }
};

int height(Node* target)
{
    if (target) return target->height;
    else return 0;
}

int calculate_diff(Node *target)
{
    return height(target->right)-height(target->left);
}

void recalculate_height(Node *target)
{
    int left_height = height(target->left);
    int right_height = height(target->right);
    target->height = (left_height>right_height ? left_height : right_height) + 1;
}

size_t calc_count(Node* target)
{
    if (!target) {
        return 0;
    }
    if (!target->left && !target->right) {
        return 1;
```

```
    }
    return calc_count(target->left) + calc_count(target->right) + 1;
}

Node* right_rotate(Node* target)
{
    Node *target_child = target->left;
    target->left = target_child->right;
    target_child->right = target;
    recalculate_height(target);
    recalculate_height(target_child);
    return target_child;
}

Node* left_rotate(Node* target)
{
    Node *target_child = target->right;
    target->right = target_child->left;
    target_child->left = target;
    recalculate_height(target);
    recalculate_height(target_child);
    return target_child;
}

void print_tree(Node* target, int level = 0)
{
    if (!target) return;

    print_tree(target->right, level + 1);

    for (int i = 0; i < level; i++)
    {
        cout << "      ";
    }

    cout << target->key << '(' << target->count_same << ")+(" << target->size << ')' <<
endl;

    print_tree(target->left, level + 1);
}

Node *rebalance(Node* target)
{
    recalculate_height(target);
    if (calculate_diff(target) == 2)
    {
        if (calculate_diff(target->right) < 0) {
            target->right = right_rotate(target->right);
        }
        return left_rotate(target);
    }
    if (calculate_diff(target) == -2)
    {
        if (calculate_diff(target->left) > 0) {
```

```
            target->left= left_rotate(target->left);
        }
        return right_rotate(target);
    }
    return target;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

void updateHeight(Node* node) {
    node->height = 1 + max(height(node->left), height(node->right));
}


Node* right_rotate_for_copy(Node* target)
{
    int height_first, height_second;

    Node *target_child = target->left;

    target->left = target_child->right;

    target_child->right = target;

    if (!target->left && !target->right) {
        // target->size = 1;
        target->size = target->count_same;
    } else if (target->left) {
        if (target->right) {
            target->size = target->left->size + target->right->size + target->count_same;
        } else {
            target->size = target->left->size + target->count_same;
        }
    } else {
        if (target->right) {
            target->size = target->right->size + target->count_same;
        }
    }

    if (target_child->left) {
        if (target_child->right) {
            target_child->size = target_child->left->size + target_child->count_same +
target_child->right->size;
        } else {
            target_child->size = target_child->left->size + target_child->count_same;
        }
    } else {
        if (target_child->right) {
            target_child->size = target_child->right->size + target_child->count_same;
        } else {
                    target_child->size = target_child->count_same;
        }
```

```
    }

    updateHeight(target);
    updateHeight(target_child);

    return target_child;
}

Node* left_rotate_for_copy(Node* target)
{
    Node *target_child = target->right; // target_child = 6
    target->right = target_child->left; // target->right = A
    target_child->left = target;

    if (!target->left && !target->right) {
        // target->size = 1;
        target->size = target->count_same;
    } else if (target->right) {
        if (target->left) {
            target->size = target->right->size + target->left->size + target->count_same;
        } else {
            target->size = target->right->size + target->count_same;
        }
    } else {
        if (target->left) {
            target->size = target->left->size + target->count_same;
        }
    }

    if (target_child->right) {
        if (target_child->left) {
            target_child->size = target_child->right->size + target_child->count_same +
target_child->left->size;
        } else {
            target_child->size = target_child->right->size + target_child->count_same;
        }
    } else {
        if (target_child->left) {
            target_child->size = target_child->left->size + target_child->count_same;
        } else {
                    target_child->size = target_child->count_same;
        }
    }

    updateHeight(target);
    updateHeight(target_child);

    return target_child;
}


Node* insert_node(Node *target, int key, int x1, int x2, int i)
{
    vector<pair<Node*, int>> path; // массив пути
```

```
    vector<Node*> copy; // массив скопированных путей
    if (i != 0) {

        Node* current = target;
        Node* befor_ins = nullptr;

        if (!target) return nullptr;

        while (current) {
            if (current->key > key) {
                path.push_back(make_pair(current, -1)); // заполнение массива путей
                befor_ins = current;
                current = current->left;
            } else if (current->key < key){
                path.push_back(make_pair(current, 1));
                befor_ins = current;
                current = current->right;
            } else {
                path.push_back(make_pair(current, 1));
                befor_ins = current;
                break;
            }

        }

        int count = 0;
        int curr_height = 1;



        for (int k = path.size() - 1; k >= 0; k--) { // снизу вверх копирование пути
            int flag_size = 0;
            if (k == path.size() - 1) {
                Node *old_node_before_ins = befor_ins;
                Node *new_copy_node = new Node(old_node_before_ins->key,
old_node_before_ins->x1, old_node_before_ins->x2);
                new_copy_node->height = 2;
                // curr_height++;
                new_copy_node->count_same = old_node_before_ins->count_same;

                Node *new_node = new Node(key, x1, x2);

                if (key < new_copy_node->key) {
                    new_copy_node->left = new_node;
                    new_copy_node->right = old_node_before_ins->right;

                    new_copy_node->height = 2;

                    if (new_copy_node->right) {
                        new_copy_node->size = new_copy_node->left->size + new_copy_node-
>right->size + new_copy_node->count_same;
                    } else {
                        new_copy_node->size = new_copy_node->left->size + new_copy_node-
>count_same;
```

```
                    }

                    flag_size = 1;
                } else if (key > new_copy_node->key) {
                    new_copy_node->right = new_node;
                    new_copy_node->left = old_node_before_ins->left;

                    new_copy_node->height = 2;

                    if (new_copy_node->left) {
                        new_copy_node->size = new_copy_node->right->size + new_copy_node-
>left->size + new_copy_node->count_same;
                    } else {
                        new_copy_node->size = new_copy_node->right->size + new_copy_node-
>count_same;
                    }

                    flag_size = 1;
                } else {
                    new_copy_node->count_same = old_node_before_ins->count_same + 1;
                    new_copy_node->left = old_node_before_ins->left;
                    new_copy_node->right = old_node_before_ins->right;

                    if (new_copy_node->left != nullptr || new_copy_node->right != nullptr)
{
                        new_copy_node->height = 2;
                    } else {
                        new_copy_node->height = 1;
                    }

                    if (new_copy_node->left) {
                        if (new_copy_node->right) {
                            new_copy_node->size = new_copy_node->left->size +
new_copy_node->count_same + new_copy_node->right->size ;
                        } else {
                            new_copy_node->size = new_copy_node->left->size +
new_copy_node->count_same;
                        }
                    } else {
                        if (new_copy_node->right) {
                            new_copy_node->size = new_copy_node->count_same +
new_copy_node->right->size ;
                        } else {
                            new_copy_node->size = new_copy_node->count_same;
                        }
                    }
                }

                copy.push_back(new_node); // новая версия
                count++;
                copy.push_back(new_copy_node);
                count++;
            } else {
                Node *old_node_before_ins = path[k].first;
                Node *new_copy_node = new Node(old_node_before_ins->key,
```

```
old_node_before_ins->x1, old_node_before_ins->x2);

                new_copy_node->height = curr_height + 1;
                curr_height++;

                new_copy_node->count_same = old_node_before_ins->count_same;


                int left_child_height, right_child_height;


                if (path[k+1].first->key < path[k].first->key) {
                    new_copy_node->left = copy[count-1];
                    new_copy_node->right = old_node_before_ins->right;


                    if (new_copy_node->right) {
                        new_copy_node->size = new_copy_node->left->size + new_copy_node-
>right->size + new_copy_node->count_same;
                    } else {
                        new_copy_node->size = new_copy_node->left->size + new_copy_node-
>count_same;
                    }


                }
                if (path[k+1].first->key > path[k].first->key) {
                    new_copy_node->right = copy[count-1];
                    new_copy_node->left = old_node_before_ins->left;


                    if (new_copy_node->left) {

                        new_copy_node->size = new_copy_node->right->size + new_copy_node-
>left->size + new_copy_node->count_same;
                    } else {
                        new_copy_node->size = new_copy_node->right->size + new_copy_node-
>count_same;

                    }

                }


                copy.push_back(new_copy_node);
                count++;

                recalculate_height(copy[count-1]);
                if (calculate_diff(copy[count-1]) == 2)
                {
                    if (calculate_diff(copy[count-1]->right) < 0) {

                        copy[count-1]->right = right_rotate_for_copy(copy[count-1]->right);
                    }
```

```
                    copy[count-1] = left_rotate_for_copy(copy[count-1]);
                }
                if (calculate_diff(copy[count-1]) == -2)
                {
                    if (calculate_diff(copy[count-1]->left) > 0) {
                        copy[count-1]->left = left_rotate_for_copy(copy[count-1]->left);
                    }
                    copy[count-1] = right_rotate_for_copy(copy[count-1]);
                }

            }
        }

        return copy[count-1];


    } else { // добавляем корень
        if (!target) {
            return new Node(key, x1, x2);
        }

        return rebalance(target);
    }

}

int calculateResult_x1(Node* target, int x) {
    if (!target) {
        return 0;
    }

    int count = 0;

    Node *current = target;
    while(current) {
        if (current->key <= x) {
            if (current->right) {
                count += current->size - current->right->size; // 3
                current = current->right; //вправо
            } else {
                count += current->size;
                break;
            }
        } else {
            current = current->left;
        }
    }

    return count;
}
int calculateResult_x2(Node* target, int x) {
    if (!target) {
        return 0;
```

```
    }

    int count = 0;

    Node *current = target;
    while(current) {
        if (current->key < x) {
            if (current->right) {
                count += current->size - current->right->size; // 3
                current = current->right; //вправо
            } else {
                count += current->size;
                break;
            }
        } else {
            current = current->left;
        }
    }

    return count;
}

struct point {
    int point_x, point_y;
};

struct intresting_point {
    int x1, x2, y;
};

bool compare(const intresting_point& a, const intresting_point& b) {
    return (a.y > b.y);
}

int find_version(vector<intresting_point>& intresting_points, int n, int curr_y) {
    int count = -1;
    for (int i = 0; i < n; i++) {
        if (intresting_points[i].y > curr_y) {
            count++;
        } else break;
    }
    return count;
}

int main() {
    ios:: sync_with_stdio(false);
    cin.tie(0);

    int n, m;

    cin >> n;
    cin >> m;
```

```
    vector<intresting_point> intresting_points(n);
    int first_digit, second_digit, third_digit;

    for (int i = 0; i < n; i++) {
        cin >> first_digit;
        intresting_points[i].x1 = first_digit;

        cin >> second_digit;
        intresting_points[i].x2 = second_digit;

        cin >> third_digit;
        intresting_points[i].y = third_digit;
    }

    sort(intresting_points.begin(), intresting_points.end(), compare);


    Node* root = nullptr;

    vector<Node*> roots(n);
    vector<Node*> starts(n);
    vector<Node*> ends(n);

    for (int i = 0; i < n; i++) {

        if (i == 0) {
            starts[0] = insert_node(root, intresting_points[i].x1, intresting_points[i].y,
intresting_points[i].x2, i);
        }

        if (i > 0) {
            starts[i] = insert_node(starts[i-1], intresting_points[i].x1,
intresting_points[i].y, intresting_points[i].x2, i);
        }

        if (i == 0) {
            ends[0] = insert_node(root, intresting_points[i].x2, intresting_points[i].x1,
intresting_points[i].y, i);
        }

        if (i > 0) {
            ends[i] = insert_node(ends[i-1], intresting_points[i].x2,
intresting_points[i].x1, intresting_points[i].y, i);
        }

    }

    point curr_point;


    while (m > 0) {
        m--;
        int curr_x = 0;
```

```
            int curr_y = 0;
            cin >> curr_x;
            cin >> curr_y;

            int version = find_version(intresting_points, n, curr_y);

            if (version == -1) {
                cout << 0 << "\n";
            } else {
                int result = 0;

                result = calculateResult_x1(starts[version], curr_x) -
calculateResult_x2(ends[version], curr_x);

                cout << result << "\n";
            }
        }
        return 0;
}
```

7. Сложность работы

Название функции	Сложность
int main()	O(nlogn + m) – где n количество отрезков (встроенная сортировка), m – количество точек.
Node* insert_node(Node *target, int key, int x1, int x2, int i)	O(logn) – где n количество элементов в дереве.
int calculateResult_x1(Node* target, int x)	O(logn) – где n количество элементов в дереве.
int calculateResult_x2(Node* target, int x)	O(logn) – где n количество элементов в дереве.
int find_version(vector<intresting_point>& intresting_points, int n, int curr_y)	O(n) – где n количество отрезков
void recalculate_height(Node *target)	O(1)

8.Вывод

В ходе выполнения данного курсового проекта я научился реализовывать персистентную структуру данных, также познакомился с методом копирования пути и с методом сканирующей прямой. В ходе курсовой работы я сталкивался с трудностями реализации, ведь задача имеет несколько методов решения, но каждый из них имеет свои интересные моменты, которые заставляют задуматься.