Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

По теме: Персистентные структуры данных

Студент:	Д. И. Артемьев
Преподаватель:	А. Н. Ридли
Группа:	М8О-306Б
Дата:	
Оценка:	
Подпись:	

Москва, 2021

Курсовой проект: Персистентные структуры данных

Задача:

Дан набор горизонтальных отрезков, и набор точек. Для каждой точки определите, сколько отрезков лежит строго над ней.

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.

Формат входных данных

В первой строке даны два числа n и m — количество отрезков и количество точек соответственно.

В следующих n строках заданы отрезки в виде троек чисел l, r и h — координаты x левой и правой границ отрезка и координата y отрезка соответственно.

В следующих m строках даны пары чисел x и y — координаты точек.

Формат результата

Для каждой точки вывести целое число — количество отрезков над ней.

1 Описание

Задача решается при помощи метода разделения плоскости на плитки после проведения вертикальных линий через точки всех отрезков (Dobkin-Lipton [1] “slabs”). Строим персистентное декартово дерево с использованием сканирующей прямой по точкам наших отрезков. Строим явное декартово дерева, которое является бинарным деревом поиска по значениям высоты. Если встречаем точку начала отрезка — добавляем значение её высоты, если встречаем точку конца — удаляем значение высоты из декартова дерева. Очередная добавленная или удалённая точка даёт нам новое состояние дерева.

При помощи бинарного поиска ищем по координате x, какой плитке принадлежит точка. Номер плитки даст нам номер состояние персистентного декартова дерева. Затем в декартовом дереве для этой плитки считаем количество вершин, имеющих значение высоты отрезка большее, чем координата y точки из запроса.

Достижение персистентности

Теперь нужно поговорить про реализацию персистентного декартова дерева.

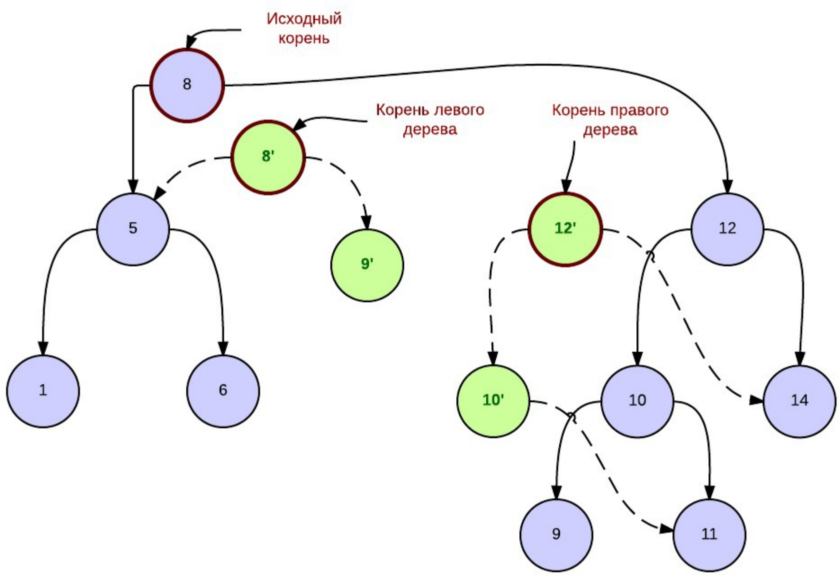
В статье Сарнака и Тарьяна 1986 года [2] описывается персистентное красно-чёрное дерево, поддерживающее операции split и merge. Однако сложность написания красно-чёрного дерева значительно выше сложности написания декартова дерева, которое было предложено спустя 10 лет с момента опубликования статьи. Также в ней описывается достижение персистентности с выделением в среднем $O(1)$ амортизированной памяти на очередное изменение дерева при сложности на запрос $O(\log n)$. Метод Сарнака и Тарьяна основывается на комбинировании двух методов для построения персистентной структуры: метода копирования путей и метода толстых узлов. Однако реализация их задумки несколько усложняет программу.

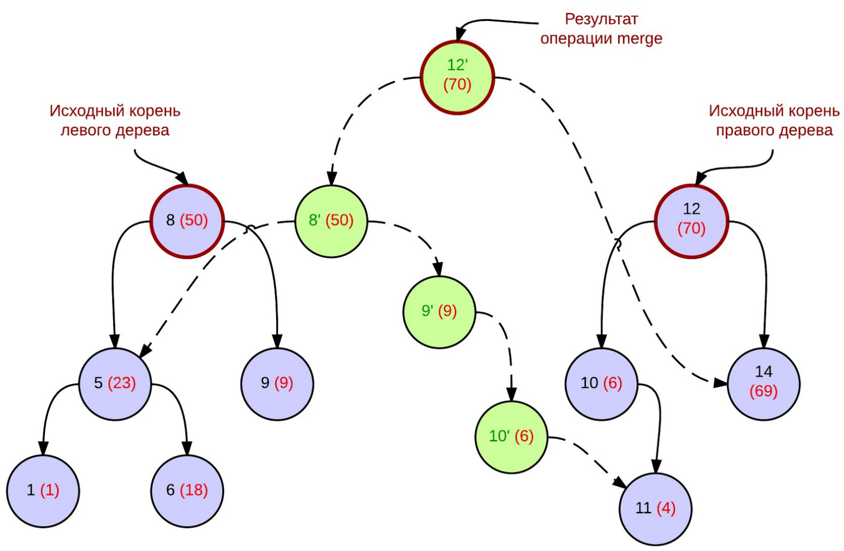
Отдельное использование метода толстых узлов даёт линейную память, однако сложность запросов в дереве становится равной $O(\log^2 n)$, так как для перехода к потомку, нужно использовать бинарный поиск для определения, по указателю для какого состояния нужно перейти. Я же отдал предпочтение скорости ответов на запросы.

В этом курсовом проекте для достижения персистентности будет использоваться метод копирования путей. Таким образом, если количество всех точек на плоскости, по которым мы строим декартово дерево, равно n, то сложность по памяти будет равна $O(n \log n)$, так как в дерево будут добавляться и удаляться узлы для каждой точки и на каждую операцию добавления и удаления будет выделяться $O(\log n)$ новых узлов. Уже из затрачиваемой памяти понятно, что сложность такого построения персистентного декартова дерева равна $O(n \log n)$.

Отдельные состояния будут храниться в векторе корней персистентного декартова дерева.

Привожу иллюстрации персистентных операций split и merge в неявном декартовом дереве со статьи на хабре [3]





Предобработка массива отрезков

Наверное, стоит уделить внимание использованию сканирующей прямой для предобработки точек отрезков. Здесь всё просто — создаём вектор пар, первый элемент пары будет содержать координату x точки какого-то из отрезков, а второй элемент будет содержать номер отрезка, которому эта точка принадлежала в исходном векторе отрезков. Затем этот вектор пар необходимо отсортировать по первой координате. И по уже отсортированному вектору нужно будет строить наше дерево.

Количество отрезков n, количество всех их точек $2n$. Следовательно предобработка с учётом сортировки будет иметь сложность $O(n \log n)$.

Осуществление запроса

Как было сказано ранее, сначала необходимо определить, в каком состоянии дерева необходимо будет делать подсчёт. Это делается при помощи одного бинарного поиска, сложность которого $O(\log n)$.

Запрос количества элементов, больших чем значение y заданной точки, в отдельном состоянии дерева будет осуществляться за $O(\log n)$. Для этого достаточно спуститься по дереву стандартным поиском элемента, добавляя в сумму для узла, значение высоты в котором больше значения y, количество элементов в правых поддеревьях плюс 1. Единица добавляется, так как нужно считать и сам узел, в котором условие выполнилось.

Таким образом, сложность одного запроса будет $O(\log n)$ на бинпоиск $+O(\log n)$ на поиск в декартовом дереве (которое в среднем почти сбалансированное). Значит общая сложность равна $O(\log n)$.

Аккуратная работа с памятью

Для того, чтобы в программе точно не возникало утечек памяти, было решено использовать умные указатели. std::unique_ptr не подошли, потому как в методе копирования путей образуется множесво указателей на одни и те же узлы. Поэтому в программе используются std::shared_ptr, которые идеально подходят под нужды задачи. Таким образом, нет нужды писать свой деструктор для всей персистентной структуры.

2 Исходный код

```cpp
// main.cpp
#include "persistent_treap.hpp"
#include <iostream>
#include <limits>

int main() {
    int n, m;
    std::cin >> n >> m;

    // preprocessing
    std::vector<base::TSegment> segments(n + 1);
    segments.front() = {std::numeric_limits<int>::min(),
        std::numeric_limits<int>::max(),
        std::numeric_limits<int>::min()};

    std::vector<std::pair<int, int>> segments_info(n * 2 + 2);
    segments_info[0] = {segments.front().l, 0};
    segments_info[1] = {segments.front().r, 0};

    for (int i = 1; i < n + 1; i++) {
        std::cin >> segments[i].l >> segments[i].r >> segments[i].h;

        if (segments[i].l > segments[i].r) {
            std::swap(segments[i].l, segments[i].r);
        }

        segments_info[i * 2] = {segments[i].l, i};
        segments_info[i * 2 + 1] = {segments[i].r + 1, i}; // delete after point
    }

    std::sort(segments_info.begin(), segments_info.end());

    base::TPersistentTreap tree(segments, segments_info);

    for (const std::pair<int, int> &point : segments_info) {
        int segment_id = point.second;
        base::TSegment current = segments[segment_id];
        if (current.l == point.first) {
            tree.Insert(current.h);
        } else {
            tree.Remove(current.h);
        }
    }

    // online query answering
    for (int i = 0; i < m; i++) {
        int x, y;
```

```
        std::cin >> x >> y;
        std::cout << tree.CntUpperSegments(x, y) << std::endl;
    }
}
```

```
// persistent_treap.hpp
#pragma once
#include <algorithm>
#include <memory>
#include <random>
#include <vector>

namespace base {

struct TSegment {
    int l, r, h;
};

struct Node {
    int size;
    int height;

    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;

public:
    Node() = delete;
    Node(const int height_)
        : size(1), height(height_), left(nullptr), right(nullptr) {}

    Node(const Node &other) {
        this->size = other.size;
        this->height = other.height;

        if (other.left != nullptr) {
            this->left = other.left;
        }
        if (other.right != nullptr) {
            this->right = other.right;
        }
    }
};

using ptr_node = std::shared_ptr<Node>;

struct TPersistentTreap {
    TPersistentTreap() = delete;
    TPersistentTreap(const std::vector<TSegment> &segments_,
```

```
                    const std::vector<std::pair<int, int>> &segments_info_) {
        this->segments = segments_;
        this->segments_info = segments_info_;
    }

    void Insert(int height);
    void Remove(int height);
    int CntUpperSegments(int x, int y);
    int Size() { return root.size(); }

private:
    ptr_node CopyNode(ptr_node &prev);
    int Size(const ptr_node &node);
    void UpdateNode(ptr_node &node);
    bool GoLeft(int left, int right);
    void Merge(ptr_node &node, ptr_node &left, ptr_node &right);
    void SplitHeight(ptr_node &node, ptr_node &left, ptr_node &right, int key);
    void SplitSize(ptr_node &node, ptr_node &left, ptr_node &right, int key);
    int CntUpperSegments(ptr_node &node, int y);

    std::vector<TSegment> segments;
    std::vector<std::pair<int, int>> segments_info;
    std::vector<ptr_node> root;
};

} // namespace base
```

```
// persistent_treap.cpp
#include "persistent_treap.hpp"

namespace base {
using ptr_node = std::shared_ptr<Node>;

std::random_device rd;
std::mt19937 rand_number(rd());

int TPersistentTreap::CntUpperSegments(int x, int y) {
    if (root.empty()) {
        return 0;
    }
    int idx =
        std::lower_bound(segments_info.begin(), segments_info.end(), x,
                         [](const std::pair<int, int> &a, const int b) {
                             return a.first < b;
                         }) -
        segments_info.begin();

    if (idx >= static_cast<int>(root.size())) {
        idx -= 1;
    }
```

```cpp
    if (segments_info[idx].first > x) {
        idx -= 1;
    }

    return CntUpperSegments(root[idx], y);
}

int TPersistentTreap::CntUpperSegments(ptr_node &node, int height) {
    if (node == nullptr) {
        return 0;
    }
    if (node->height <= height) {
        return CntUpperSegments(node->right, height);
    }
    else {
        return CntUpperSegments(node->left, height) + Size(node->right) + 1;
    }
}

void TPersistentTreap::Insert(int height) {
    if (root.empty()) {
        root.push_back(std::shared_ptr<Node>(new Node(height)));
        return;
    }

    ptr_node left;
    ptr_node right;
    ptr_node new_node = std::shared_ptr<Node>(new Node(height));
    SplitHeight(root.back(), left, right, height);

    root.resize(root.size() + 1);
    ptr_node tmp;
    Merge(tmp, left, new_node);
    Merge(root.back(), tmp, right);
}

void TPersistentTreap::Remove(int height) {
    if (root.empty()) {
        return;
    }

    ptr_node left;
    ptr_node right;
    ptr_node mid;
    SplitHeight(root.back(), left, right, height);
    SplitSize(right, mid, right, 1);

    root.resize(root.size() + 1);
    Merge(root.back(), left, right);
```

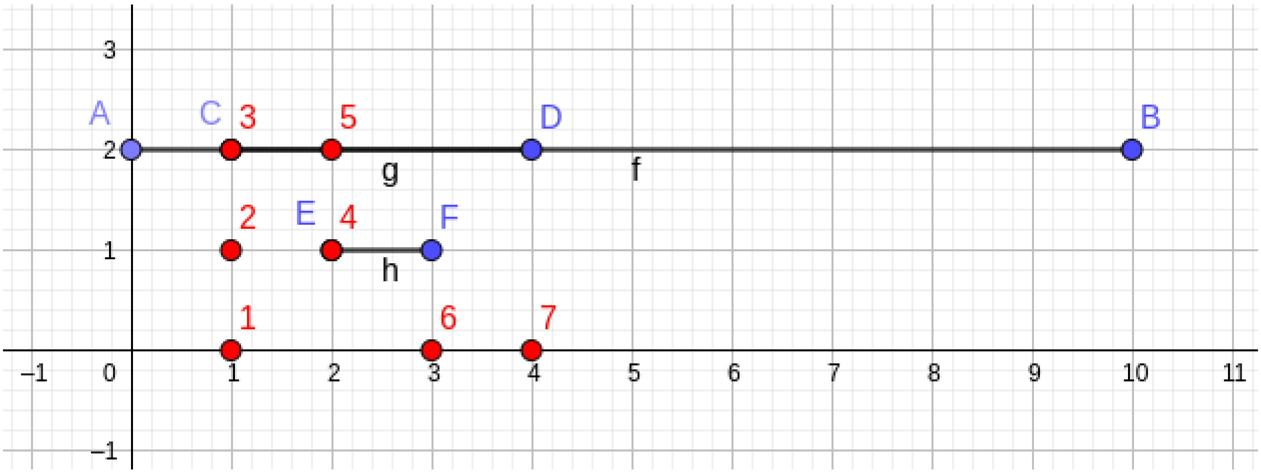
```
}

ptr_node TPersistentTreap::CopyNode(ptr_node &prev) {
    if (prev == nullptr) {
        return nullptr;
    }

    ptr_node new_node = std::shared_ptr<Node>(new Node(*prev));
    return new_node;
}

int TPersistentTreap::Size(const ptr_node &node) {
    if (node == nullptr) {
        return 0;
    }
    return node->size;
}

void TPersistentTreap::UpdateNode(ptr_node &node) {
    if (node == nullptr) {
        return;
    }
    node->size = Size(node->left) + Size(node->right) + 1;
}

bool TPersistentTreap::GoLeft(int left, int right) {
    return static_cast<int>(rand_number()) % (left + right) < left;
}

void TPersistentTreap::Merge(ptr_node &node, ptr_node &left, ptr_node &right) {
    if (left == nullptr) {
        node = CopyNode(right);
        return;
    }
    if (right == nullptr) {
        node = CopyNode(left);
        return;
    }

    if (GoLeft(Size(left), Size(right))) {
        node = CopyNode(left);
        Merge(node->right, left->right, right);
    } else {
        node = CopyNode(right);
        Merge(node->left, left, right->left);
    }

    UpdateNode(node);
}
```

```

void TPersistentTreap::SplitHeight(ptr_node &node, ptr_node &left,
                                  ptr_node &right, int key) {
    if (node == nullptr) {
        left = nullptr;
        right = nullptr;
        return;
    }

    if (node->height < key) {
        left = CopyNode(node);
        SplitHeight(node->right, left->right, right, key);
        UpdateNode(left);
    } else {
        right = CopyNode(node);
        SplitHeight(node->left, left, right->left, key);
        UpdateNode(right);
    }
}

void TPersistentTreap::SplitSize(ptr_node &node, ptr_node &left, ptr_node &right,
                                int key) {
    if (node == nullptr) {
        left = nullptr;
        right = nullptr;
        return;
    }

    if (Size(node->left) + 1 <= key) {
        left = CopyNode(node);
        SplitSize(left->right, left->right, right, key - Size(node->left) - 1);
        UpdateNode(left);
    } else {
        right = CopyNode(node);
        SplitSize(right->left, left, right->left, key);
        UpdateNode(right);
    }
}
} // namespace base
```

3 Пример использования

Программа была протестирована на сгенерированных тестах, здесь же приводится простой тестовый пример.

```
amder@amder-pc $ g++ -O2 -std=c++17 main.cpp persistent_treap.cpp -o main
amder@amder-pc $ cat input.txt
3 7
1 4 2
2 3 1
0 10 2
1 0
1 1
1 2
2 1
2 2
3 0
4 0
amder@amder-pc $ ./main <input.txt
2
2
0
2
0
3
2
```



4 Выводы

В данном курсовом проекте выполнен базовый вариант задачи. Продвинутый предполагает определение, какому многоугольнику принадлежит данная точка на плоскости в режиме онлайн. Для доведения текущей программы до решения продвинутой задачи есть всё необходимое: в статье Сарнака и Тарьяна [2] описан метод нахождения, какой области принадлежит точка на плоскости, на которой отображён планарный граф. Очевидным образом этот метод сводится к нахождению многоугольника, которому принадлежит точка в случае непересекающихся выпуклых многоугольников. Далее необходимо добавлять дополнительные атрибуты структуры отрезков для масштабирования задачи на непересекающиеся многоугольники без самопересечений. Следующим этапом является решение задачи и для пересекающихся многоугольников. В таком случае ответом может быть несколько многоугольников. Для решения этой задачи потребуется также строить вертикальные прямые через все точки пересечений отрезков и производить более аккуратную обработку.

Решение продвинутой задачи может найти применение в компьютерной графике, программах ГИС, системах компьютерного проектирования CAD.

В базовом варианте, который был реализован, существует также решение через персистентное дерево отрезков. Базовая идея разделения на вертикальные плитки здесь остаётся. Далее, при добавлении новой точки в последнее состояние дерева нужно будет добавлять единицу в позицию, соответствующую координате y. При удалении единицу нужно будет вычитать. Затем, при ответе на запрос также находим соответствующее состояние персистентного дерева отрезков, после чего нужно будет считать сумму всех элементов, правее y точки из запроса. Сложность по памяти при таком варианте будет $O(n \log n)$, время на построение $O(n \log n)$, сложность ответа на запрос также, как и в моей программе, $O(\log n)$.

Список литературы

[1] Dobkin. D., and Lipton. R.J. *Multidimensional SIAM* J. Compuf. 5, 2 (June 1976). 181-186.

[2] Neil Sarnak and Robert E. Tarjan *Planar Point Location Using Persistent Search Trees* Communications of the ACM, July 1986, Volume 29, Number 7

[3] Персистентное декартово дерево по неявному ключу: https://habr.com/ru/post/240519/