Краткая теоретическая сводка

1. Декартово дерево

Будем далее считать, что читателю знакомы основы теории вероятностей, понятия бинарного дерева поиска и кучи.

1.1. Описание

Декартово дерево — это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу.
Декартово дерево — это структура данных, которая хранит пары (X, Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной кучей по y. Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0, Y_0), то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y < Y_0$.

Сделаем небольшое лирическое отступление и поговорим о том, зачем это вообще нужно. Дело в том, что с величиной X мы будем ассоциировать ключ соответствующий вершины. Читатель спросит, зачем нужен Y? Тут всё просто. Одними X ограничиться не получится. Мы уже знаем, что в бинарном дереве поиска вставка работает за $O(h)$, где h – высота дерева. Если мы ничего не сделаем, то наше дерево будет голым бинарным деревом поиска, и легко придумать пример, высота станет равной n. Такое положение вещей нас, конечно, не устраивает. Для этого-то и придуманы величины Y – приоритеты вершин. Как они помогут решить нашу проблему, мы увидим ниже.

1.2. Свойства приоритетов

"Декартово дерево". Декарт тут, разумеется, не причём. Дерево было придумано в 1989 году Сиделем (Siedel) и Арагоном (Aragon). Но почему же мы так его называем? Потому что это дерево очень хорошо визуализируется на плоскости с введёной в ней декартовой системой координат. Мы просто будем рисовать вершинки в точках (X, Y) и соединять их рёбрами. Пока оставим эту картину, но будем о ней переодически вспоминать.
Важно: об этом мы ещё не говорили, однако здесь и далее всюду будем подразумевать, что приоритеты попарно различны, то есть в дереве нет ни одной вершины с одинаковыми приоритеты. Хорошим упражнением для вас станет поиск самой первой ошибки в теоремах, если бы не наложили этого условия на приоритеты. Кроме того, для простоты рассуждений будем считать. что все ключи различны.

Лемма. *Пусть S — некоторый набор пар (x_i, y_i). Тогда декартово дерево T на наборе S определено однозначно.*

Доказательство. Будем вести рассуждения по индукции. Пусть для всех наборов размера не более $n-1$ (пустое дерево и дерево из одной вершины определены однозначно) утверждение верно. Докажем его для набора S размера n. Корнем дерева T называется вершина с наибольшим приоритетом (как это видно из определения), поэтому эта вершина $v = (x_0, y_0)$ будет определена однозначно. Его левое и правое поддеревья будут формироваться из наборов S_1 и S_2, равных

$$S_1 = \{(x, y) \big| (x, y) \in S \text{ и } (x < x_0) \text{ и } (y < y_0)\}$$

$$S_2 = \{(x, y) | (x, y) \in S \text{ и } (x > x_0) \text{ и } (y < y_0)\}$$

Но каждый из этих двух наборов по размеру меньше n, а значит левое и правое поддеревья будут определены однозначно (по предположению индукции). Лемма доказана. □

Приведём пример описания структуры вершины

```
struct Node {
    int key;
    int prior;
    Node* Left;
    Node* Right;
    ...
};
```

1.3. Split и Merge — начало великого пути.

Несмотря на весь тот громадный функционал, который представляет нам декартово дерево (мы увидим его ниже), вся его работа базируется на паре функций – split и merge. Остановимся на каждой поподробнее

1.3.1. Split

Функция split разбивает дерево T на деревья $T_{<x}$ и $T_{>x}$ по заданному ключу x. В дереве $T_{<x}$ содержатся все ключи из T, что меньше x, а в дереве $T_{>x}$ – большие x. Куда положить вершину x? На самом деле, принципиального значения это не имеет. Всё зависит от вашего желания. Главное, чтобы в одной и той же реализации, люди придерживались одной стратегии.

Теперь опишем принцип работы этой функции. В какое из деревьев $T_{<x}$ или $T_{>x}$ будет определён корень дерева T? Ответ очевиден, если ключ корня меньше x то в первое, иначе – во второе. Положим, что он меньше x. Тогда можно сразу сказать, что все элементы левого поддерева T также окажутся в $T_{<x}$ — их ключи ведь тоже все будут меньше x_0. Более того, корень T будет и корнем $T_{<x}$, поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с ключами, большими x_0, и вынести в дерево $T_{>x}$. А остаток ключей сохранить как новое правое поддерево $T_{<x}$. Эту процедуру запишем в программном коде

```
void split(Node* t, int key, Node* &l, Node* &r) {
    if (t == nullptr) //t -- empty tree
        l = r = nullptr;
    else if (key < t->key) {//t->root in T_{>x}
        split(t->Left, key, l, t->Left);
        r = t;
    } else { //t->root in T_{<x}
        split(t->Right, key, t->Right, r);
        l = t;
    }
}
```

1.3.2. Merge

Функция merge в некотором смысле противоположна split – она получает на вход два дерева T_1 и T_2, после чего формирует новое дерево T, в котором содержатся все ключи из T_1 и T_2. Обязательное условие – все ключи в T_2 должны быть меньше всех ключей в T_1. Итак, какой элемент станет корнем нового дерева? Очевидно, что это будет элемент с наибольшим приоритетом. Кандидатов на максимальный приоритет у нас два — только корни двух исходных деревьев. Сравним их приоритеты; пускай для однозначности приоритет у левого корня больше, а ключ в нем равен x. Новый корень определен, теперь стоит подумать, какие же элементы окажутся в его правом поддереве, а какие — в левом. Читателю предлагается повторить те же самые рассуждения, что были приведены в описании операции split, применительно к merge. Мы сразу приведём примерную реализацию этой процедуры

```
void merge(Node* &t, Node* l, Node* r) {
    if (l == nullptr) {
        t = r;
        return;
    }
    if (r == nullptr) {
        t = l;
        return;
    }
    if (l->prior < r->prior) {
        merge(l->Right, l->Right, r);
        t = l;
    } else {
        merge(r->Left, l, r->Left);
        t = r;
    }
}
```

1.4. Реализация других операций

Рассмотрим, как выполнить операцию вставки новой пары (X, Y). Для этого нужно просто разбить наше дерево по ключу X на два поддерева T_1 и T_2, а затем вызовём последовательно merge от T_1 и дерева из одной вершины (X, Y), потом получившегося дерева с T_2. Таким образом, получим дерево, в котором есть нужная вершина. Попробуйте придумать как выполнить процедуру удаления, поиска вершины (пользуясь только split и merge). Тем не менее, декартово дерево является бинарным деревом поиска, поэтому реализовывать все эти операции можно и как в обычном BST.

1.5. Поддержка размера поддеревьев

Как в структуре поддерживать размеры поддеревьев? Для этого достаточно добавить соответствующее поле в структуру и при всех изменениях дерева (чаще всего это происходит на выходе из рекурсии, когда во всех поддеревьях всё построено верно) обновлять значение этого поля. Это поле пригодится нам в дальнейшем, а также при реализации такой функции, как поиск k-ого ключа в дереве. Для этого нужно просто осуществить спуск по дереву, каждый раз спускаясь в то поддерева, в котором достаточно элементов.

1.6. Хранение предка

Очень полезным бывает хранение в вершине указателя на предка. Для этого необходимо при каждом изменении вершины, правильно пересчитывать этот указатель.

1.7. Высота дерева

Внимательный читатель, конечно, заметил, что мы не описали алгоритм выбора приоритетов. Ключи задаются нам задачей, но откуда взять приоритеты. Сейчас мы покажем, что эффективным решением будет выбор приоритетов случайными.

Теорема. *В декартовом дереве на n вершинах, приоритеты у которого являются случайными величинами с равномерным распределением (как и прежде попарно различными), средняя глубина вершины равна* $O(\log n)$

Доказательство. Мы не будем выражать математическое ожидание максимальной глубины, а оценим именно среднюю глубину. Пусть

- x_k — вершина, с k-ым по величине ключом.
- индикаторная величина

$$A_{ij} = \begin{cases} 1 & x_i \text{ является предком } x_j; \\ 0 & \text{иначе.} \end{cases}$$

- $d(v)$ – глубина вершины v.

По определению

$$d(x_k) = \sum_{i=1}^{n} A_{ik}$$

Выразим математическое ожидание глубины конкретной вершины

$$\mathrm{E}(d(x_k)) = \sum_{i=1}^{n} \mathsf{P}(A_{ik} = 1)$$

Посчитаем теперь вероятность того, что x_i является предком x_k, то есть $\mathsf{P}(A_{ik} = 1)$. Введём новое обозначение

$$X_{ik} = \{x_{\min(i,k)}, x_{\min(i,k)+1}, \dots, x_{\max(i,k)}\}$$

То есть X_{ik} есть множество всех ключей x_j, где j пробегает от i до k (или от k до i, если $k < i$).

Лемма. *Для любых $i \neq k$, x_i является предком x_k, тогда и только тогда, когда x_i имеет наибольший приоритет среди X_{ik}.*

Доказательство. Если x_i является корнем, то по определению имеет наибольший приоритет среди всех вершин, а значит и среди X_{ik}. Если же корнем является x_k, то x_i не является его предком и имеет меньший приоритет, чем x_k, то есть среди X_{ik}. Пусть корнем является какая-то другая вершина x_m. Тогда если x_i и x_k лежат в разных поддеревьях, то

$$\min(i,k) < m < \max(i,k) \Rightarrow x_m \in X_{ik}$$

Получается, что x_i не является предком x_k и не имеет наибольшего приоритета. Если же, наконец, они лежат в одном поддереве, то доказательство применяется по индукции, где базой является пустое поддерево, а рассматриваемое поддерево является меньшим декартовым деревом □

Вернёмся к доказательству теоремы. Итак, так как распределение равномерное, то каждая среди таких вершин X_{ik} может иметь максимальный приоритет с одинаковой вероятностью. Тогда

$$\mathsf{P}(A_{ik} = 1) = \begin{cases} \dfrac{1}{k-i+1} & k > i \\ 0 & k = i \\ \dfrac{1}{i-k+1} & k < i \end{cases}$$

Подставим последнее в выражение для математического ожидания, получим следующее

$$\mathrm{E}(d(x_k)) = \sum_{i=1}^{n} \mathsf{P}(A_{ik} = 1) = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1}$$

Воспользуемся суммой гармонического ряда. Тогда последнюю сумму можно ограничить величиной

$$\mathrm{E}(d(x_k)) \leqslant \ln(n) + \ln(n-k) + 2$$

Получается, что

$$\mathrm{E}(d(x_k)) \in O(\log n)$$

Доказательство теоремы может быть завершено. □

1.8. Построение

Придумайте как строить декартово дерево по заданным парам (x, y) за время $O(n \log n)$ (приоритеты могут быть выбраны неслучайно). Пусть теперь все ключи отсортированы, а приоритеты выбраны неслучайно. Предложите алгоритм построения дерева за $O(n)$ (*Hint*: воспользуйтесь стеком и графической визуализацией дерева). Существует ли алгоритм построения за $O(n)$ декартова дерева произвольного набора пар?

2. Декартово дерево по неявному ключу

2.1. Описание

Идея неявного ключа (Implicit Key) превращает нашу структуру из обычного бинарного дерева поиска в некоторую оболочку массива, создавая весьма обширный и многообразный интерфейс, позволяющий решать невероятное количество самых разных задач. Поговорим подробно об этой идее. Она заключается в том, что мы теперь не будем явно хранить ключ. Действительно, если мы храним в вершинах элементы массива, то в качестве ключей можно использовать индексы этих элементов. Более строго, неявный ключ для некоторой вершины t равен количеству вершин `cnt(t->Left)` в левом поддереве этой вершины плюс аналогичные величины `cnt(p->Left) + 1` для каждого предка p этой вершины, при условии, что t находится в правом поддереве для p. Ясно, как теперь быстро вычислять для текущей вершины её неявный ключ. Поскольку во всех операциях мы приходим в какую-либо вершину, спускаясь по дереву, мы можем просто накапливать эту сумму, передавая её функции. Если мы идём в левое поддерево - накапливаемая сумма не меняется, а если идём в правое - увеличивается на `cnt(t->Left)+1`.

2.2. Операции

2.2.1. Вставка элемента

Пусть нам надо вставить элемент в позицию `pos`. Разобьём декартово дерево на две половинки: соответствующую массиву $[0 \dots pos - 1]$ и массиву $[pos \dots n]$; для этого достаточно вызвать `split(t, l, r, pos)`. После этого мы можем объединить дерево `l` с новой вершиной; для этого достаточно вызвать `merge(l, l, new_item)` (нетрудно убедиться в том, что все предусловия для merge выполнены). Наконец, объединим два дерева l и r обратно в дерево t - вызовом `merge(t, l, r)`.

2.2.2. Удаление элемента

Здесь всё ещё проще: достаточно найти удаляемый элемент, а затем выполнить merge для его сыновей l и r, и поставить результат объединения на место вершины t. Фактически, удаление из неявного декартова дерева не отличается от удаления из обычного декартова дерева.

2.2.3. Функции на отрезке

Используя push-идею можно применять некоторые функции к целому отрезку; храня дополнительные поля в структуре, можно считать функции на отрезке. Придумайте как реализовывать переворот на отрезке.

2.3. Некоторые задачи и применения

2.3.1. Dynamic connectivity problem. Forests

В качестве мощного применения этой структуры можно привести пример следующей задачи. Пусть задано множество деревьев. В каждый момент времени добавляется новое ребро, удаляется существующее или даётся запрос проверить, лежат ли две вершины в одном дереве (гарантируется, что в каждый момент времени граф сохраняет структуру леса. Будем хранить Эйлеров обход каждого дерева в соответствующем декартовом дереве. Будем также поддерживать предка для каждой вершине. Операции добавления и удаления вершин можно свести к соответсвующим операциям на отрезке. Операцию проверки легко свести к тому, что достаточно подняться от двух вершин к их предку и проверить этих предков на равенство. Эта идея хранения эйлерова обхода в таком виде позволяет эффективно переподвешивать деревья и считать в них некоторую функцию.

2.3.2. Нетривиальные функции

Различными модификациями и идеями можно реализовывать тяжёлые функции. Например, пусть эффективно требуется выполнять запрос замены местами элементов, стоящих на чётных и нечётных позициях на отрезке, вставки и удаления элементов. Это можно сделать, храня два декартова дерева – для элементов на чётных и на нечётных позиций в массиве.