Сайт переезжает.

Большинство статей уже перенесено на новую версию.

Скоро добавим автоматические переходы, но пока обновленную версию этой статьи можно найти там.

Декартово дерево

Рене Декарт (фр. *René Descartes*) — великий французский математик и философ XVII века.

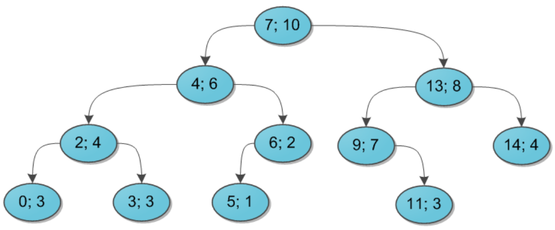
Рене Декарт не является создателем декартова дерева, но он является создателем декартовой системы координат, которую мы все знаем и любим.

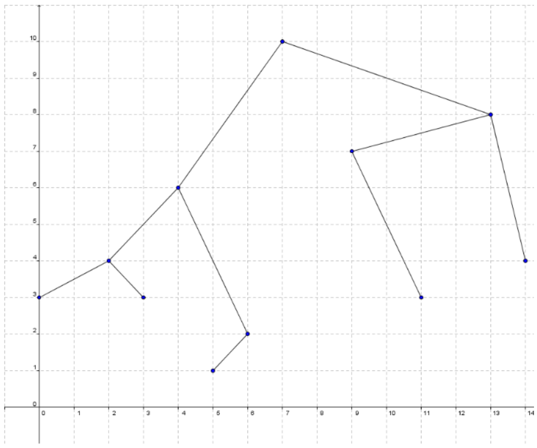
Декартово дерево же определяется и строится так:

- Нанесём на плоскость набор из n точек. Их x зачем-то назовем *ключом*, а y *приоритетом*.
- Выберем самую верхнюю точку (с наибольшим y, а если таких несколько — любую) и назовём её *корнем*.
- От всех вершин, лежащих слева (с меньшим x) от корня, рекурсивно запустим этот же процесс. Если слева была хоть одна вершина, то присоединим корень левой части в качестве левого сына текущего корня.
- Аналогично, запустимся от правой части и добавим корню правого сына.

Заметим, что если все y и x различны, то дерево строится однозначно.

Если нарисовать получившуюся структуру на плоскости, то получится действительно дерево — по традиции, корнем вверх:





Таким образом, декартово дерево — это одновременно *бинарное дерево* по x и *куча* по y. Поэтому ему придумали много альтернативных названий:

- Дерамида (дерево + пирамида)
- ПиВо (пирамида + дерево)
- КуРево (куча + дерево)
- Treap (tree + heap)

Бинарные деревья

С небольшими модификациями, декартово дерево умеет всё то же, что и любое бинарное дерево поиска, например:

- Добавить число x в множество.
- Определить, есть ли в множестве число x.
- Найти первое число, не меньшее x (`lower_bound`).
- Найти количество чисел в промежутке $[l, r]$.

При этом все операции — за $O(\log n)$.

На самом деле, бинарных деревьев очень много. В большинстве из них время выполнения операций пропорционально высоте дерева, поэтому в них придумываются разные инварианты, позволяющие эту высоту минимизировать до $O(\log n)$.

Приоритеты и асимптотика

В декартовом дереве логарифмическая высота дерева гарантируется не инвариантами и эвристиками, а законами теории вероятностей: оказывается, что если все приоритеты (y) выбирать случайно, то средняя глубина вершины будет логарифмической. Поэтому ДД ещё называют рандомизированным деревом поиска.

Теорема. Ожидание глубины вершины в декартовом дереве равно $O(\log n)$.

Доказательство. Введем функцию $a(x, y)$ равную единице, если x является предком y, и нулем в противном случае. Такие функции называются *индикаторами*.

Глубина вершины равна количеству её предков — прим. К. О. Таким образом, она равна

$$d_i = \sum_{j=1}^{n} a(j, i)$$

Её матожидание равно

$$E[d_i] = E[\sum_{j \neq i} a(j, i)] = \sum_{j \neq i} E[a(j, i)] = \sum_{j \neq i} E[a(j, i)] = \sum_{j \neq i} p(j, i)$$

где $p(x, y)$ это вероятность, что $a(x, y) = 1$. Здесь мы воспользовались важным свойством линейности: матожидание суммы чего угодно равна сумме матожиданий этого чего угодно.

Теперь осталось посчитать эти вероятности и сложить. Но сначала нам понадобится вспомогательное утверждение.

Лемма. Вершина x является предком y, если у неё приоритет больше, чем у всех вершин из полуинтервала $(x, y]$ (без ограничения общности, будем считать, что $x < y$).

Необходимость. Если это не так, то где-то между x и y есть вершина с большим приоритетом, чем x. Она не может быть потомком x, а значит x и y будут разделены.

Достаточность. Если справа будет какая-то вершина с большим приоритетом, то её левым сыном будет какая-то вершина, которая будет являться предком x. Таким образом, всё, что справа от y, ни на что влиять не будет.

У всех вершин на любом отрезке одинаковая вероятность иметь наибольший приоритет. Объединяя этот факт с результатом леммы, мы можем получить выражение для искомых вероятностей:

$$p(x, y) = \frac{1}{y - x + 1}$$

Теперь, чтобы найти матожидание, эти вероятности надо просуммировать:

$$E[d_i] = \sum_{j \neq i} p(j, i) = \sum_{j \neq i} \frac{1}{|i - j| + 1} \leq \sum_{i=1}^{n} \frac{1}{n} = O(\log n)$$

Перед последним переходом мы получили сумму гармонического ряда.

Примечательно, что ожидаемая глубина вершин зависит от их позиции: вершина из середины должна быть примерно в два раза глубже, чем крайняя.

Упражнение. Выведите из этого доказательства асимптотику `quicksort`.

Реализация

Декартово дерево удобно писать на указателях и структурах.

Создадим структуру `Node`, в которой будем хранить ключ и приоритет, а также указатели на левого и правого сына. Указателя на корень дерева достаточно для идентификации всего дерева. Поэтому, когда мы будем говорить «функция принимает два дерева» на самом деле будут иметься в виду указатели на их корни. К нулевому указателю же мы будем относиться, как к «пустому» дереву.

```
struct Node {
    int key, prior;
    Node *l = 0, *r = 0;
    Node (int _key) { key = _key, prior = rand(); }
};
```

Объявим две вспомогательные функции, изменяющие структуру деревьев: одна будет разделять деревья, а другая объединять. Как мы увидим, через них можно легко выразить почти все функции, которые нам потом понадобятся.

Merge

Принимает два дерева (два корня, L и R), про которые известно, что в левом все вершины имеют меньший ключ, чем все в правом. Их нужно объединить в одно дерево так, чтобы ничего не сломалось: по ключам это всё ещё дерево, а по приоритетами — куча.

Сначала выберем, какая вершина будет корнем. Здесь всего два кандидата — левый корень L или правый R — просто возьмем тот, у кого приоритет больше.

Пусть, для однозначности, это был левый корень. Тогда левый сын корня итогового дерева должен быть левым сыном L. С правым сыном сложнее: возможно, его нужно смерджить с R. Поэтому рекурсивно сделаем `merge(l->r, r)` и запишем результат в качестве правого сына.

```
Node* merge (Node *l, Node *r) {
    if (!l) return r;
    if (!r) return l;
    if (l->prior > r->prior) {
        l->r = merge(l->r, r);
        return l;
    }
    else {
        r->l = merge(l, r->l);
        return r;
    }
}
```

Split

Принимает дерево и ключ x, по которому его нужно разделить на два: L должно иметь все ключи не больше x, а R должно иметь все ключи больше x.

В этой функции мы сначала решим, в каком из деревьев должен быть корень, а потом рекурсивно разделим его правую или левую половину и присоединим, куда надо:

```
typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int x) {
    if (!p) return {0, 0};
    if (p->key <= x) {
        Pair q = split(p->r, x);
        p->r = q.first;
        return {p, q.second};
    }
    else {
        Pair q = split(p->l, x);
        p->l = q.second;
        return {q.first, p};
    }
}
```

Пример: вставка

`merge` и `split` сами по себе не очень полезные, но помогут написать все остальное.

Вот так, например, будет выглядеть код, добавляющий x в дерево.

```
Node *root = 0;

void insert (int x) {
    Pair q = split(root, x);
    Node *t = new Node(x);
    root = merge(q.first, merge(t, q.second));
}
```

Пример: модификация для суммы на отрезке

Иногда нам нужно написать какие-то модификации для более продвинутых операций.

Например, нам может быть интересно иногда считать сумму чисел на отрезке. Для этого в вершине нужно хранить также своё число и сумму на своем «отрезке».

```
struct Node {
    int val, sum;
    // ...
};
```

При `merge` и `split` надо будет поддерживать эту сумму актуальной.

Вместо того, чтобы модифицировать и `merge`, и `split` под наши хотелки, напишем вспомогательные функцию `upd`, которую будем вызывать при обновлении детей вершины.

```
void sum (Node* v) { return v ? v->sum : 0; }
// обращаться по пустому указателю нельзя -- выдаст ошибку

void upd (Node* v) { v->sum = sum(v->l) + sum(v->r) + v->val; }
```

В `merge` и `split` теперь можно просто вызывать `upd` перед тем, как вернуть вершину, и тогда ничего не сломается:

```
Node* merge (Node *l, Node *r) {
    // ...
    if (...) {
        l->r = merge(l->r, r);
        upd(l);
        return l;
    }
    else {
        // ...
    }
}
```

```
typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int x) {
    // ...
    if (...) {
        // ...
        upd(p);
        return {p, q.second};
    }
    else {
        // ...
    }
}
```

Тогда при запросе суммы нужно просто вырезать нужный отрезок и запросить эту сумму:

```
int sum (int l, int r) {
    Pair rq = split(root, r);
    Pair lq = split(rq.first, l);
    int res = sum(lq.second);
    root = merge(lq.first, merge(lq.second, rq.second));
    return res;
}
```

Неявный ключ

Обычное декартово дерево — это структура для множеств, каждый элемент которых имеет какой-то ключ. Эти ключи задают на этом множестве какой-то порядок, и все запросы к ДД обычно как-то привязаны к этому порядку.

Но что, если у нас есть запросы, которые этот порядок как-то нетривиально меняют? Например, если у нас есть массив, в котором нужно уметь выводить сумму на произвольном отрезке и «переворачивать» произвольный отрезок. Если бы не было второй операции, мы бы просто использовали индекс элемента в качестве ключа, но с операцией переворота нет способа их быстро поддерживать актуальными.

Решение такое: выкинем ключи, а вместо них будем поддерживать информацию, которая поможет неявно восстановить ключ, когда он нам будет нужен. А именно, будем хранить вместе с каждой вершиной размер её поддерева:

```
struct Node {
    int key, prior, size = 1;
    //               ^ размер поддерева
    Node *l = 0, *r = 0;
    Node (int _key) { key = _key, prior = rand(); }
};
```

Размеры поддеревьев будем поддерживать по аналогии с суммой — напишем вспомогательную функцию, которую будем вызывать после каждого структурного изменения вершины.

```
int size (Node *v) { return v ? v->size : 0; }

void upd (Node *v) { v->size = 1 + size(v->l) + size(v->r); }
```

`merge` не меняется, а вот в `split` нужно использовать позицию корня вместо его ключа.

Про `split` теперь удобнее думать как “вырежи первые `k` элементов”.

```
typedef pair<Node*, Node*> Pair;

Pair split (Node *p, int k) {
    if (!p) return {0, 0};
    if (size(p->l) + 1 <= k) {
        Pair q = split(p->r, k - size(p->l) - 1);
        //                  ^ правый сын не знает количество вершин слева от него
        p->r = q.first;
        upd(p);
        return {p, q.second};
    }
    else {
        Pair q = split(p->l, k);
        p->l = q.second;
        upd(p);
        return {q.first, p};
    }
}
```

Всё. Теперь у нас есть клёвая гибкая структура, которую можно резать как угодно.

Пример: ctrl+x, ctrl+v

```
Node* ctrlx (int l, int r) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l);
    root = merge(q2.first, q1.second);
    return q2.second;
}
```

```
void ctrlv (Node *v, int k) {
    Pair q = split(root, k);
    root = merge(q.first, merge(v, q.second));
}
```

Пример: переворот

Нужно за $O(\log n)$ обрабатывать запросы переворота произвольных подстрок: значение a_l поменять с a_r, a_{l+1} поменять с a_{r-1} и т. д.

Будем хранить в каждой вершине флаг, который будет означать, что её подотрезок перевернут:

```
struct Node {
    bool rev;
    // ...
};
```

Поступим по аналогии с ДО — когда мы когда-либо встретим такую вершину, мы поменяем ссылки на её детей, а им самим передадим эту метку:

```
void push (node *v) {
    if (v->rev) {
        swap(v->l, v->r);
        if (v->l)
            v->l->rev ^= 1;
        if (v->r)
            v->r->rev ^= 1;
    }
    v->rev = 0;
}
```

Аналогично, эту функцию будем вызывать в начале `merge` и `split`.

Саму функцию `reverse` реализуем так: вырезать нужный отрезок, поменять флаг.

```
void reverse (int l, int r) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l)
    q2.second->rev ^= 1;
    root = merge(q2.first, merge(q2.second, q1.second));
}
```

Небольшой рефакторинг

Реализация большинства операций всегда примерно одинаковая — вырезаем отрезок с l по r, что-то с ним делаем и склеиваем обратно.

Дублирующийся код — это плохо. Давайте используем всю мощь плюсов и определим функцию, которая принимает другую функцию, которая уже делает полезные вещи на нужном отрезке.

```
auto apply (int l, int r, auto f) {
    Pair q1 = split(root, r);
    Pair q2 = split(q1.first, l)
    q2.second = f(q2.second);
    root = merge(q2.first, merge(q2.second, q1.second));
}

void reverse (Node *v) {
    if (v)
        v->rev ^= 1;
}
```

Применять её нужно так:

```
apply(l, r, reverse);
```

Это работает в плюсах, начиная с `g++14`.

Для простых операций можно даже написать лямбду:

```
apply(l, r, [](Node *v){
    if (v)
        v->rev ^= 1;
});
```

Персистентность

Так же, как и с ДО, персистентной версией ДД можно решать очень интересные задачи.

> Дана строка. Требуется выполнять в ней копирования, удаления и вставки в произвольные позиции.

Построим персистентное ДД. Тогда просто вызвав два `split`-а, мы можем получить копию любой подстроки (указатель вершину), которую потом можно вставлять куда угодно, при этом оригинальную подстроку мы не изменим.

> Дана строка. Требуется выполнять в ней копирования, удаления, вставки в произвольные позиции **и сравнение произвольных подстрок**.

Можно в вершинах хранить **полиномиальный хэш** соответствующей подстроки. Тогда мы можем проверять равенство подстрок сравниванием хэшей вершин, полученных теми же двумя сплитами.

Чтобы полноценно сравнивать стоки лексикографически, можно применить бинарный поиск: перебрать длину совпадающего суффикса, и, когда она найдется, посмотреть на следующий символ.

Реализация почти такая же, как и для всех персистентных структур на ссылках — перед тем, как идти в какую-то вершину, нужно создать её копию и идти в неё. Создадим для этого вспомогательную функцию `copy`:

```
Node* copy (Node *v) { return new Node(*v); }
```

Во всех методах мы будем начинать с копирования всех упоминаемых в ней вершин. Например, персистентный `split` начнётся так:

```
Pair split (Node *p, int x) {
    p = copy(p);
    // ...
}
```

В ДО просто создавать копии вершин было достаточно. Этого обычно достаточно для всех детерминированных структур данных, но в ДД всё сложнее. Оказывается существует тест, который «валит» приоритеты: можно раскопировать много версий одной вершины, а все остальные — удалить. Тогда у всех вершин будет один и тот же приоритет, и дерево превратится в «бамбук», в котором все операции будут работать за линию.

У этой проблемы есть очень элегантное решение — избавиться от приоритетов, и делать теперь следующее переподвешивание: если размер левого дерева равен L, а размер правого R, то будем подвешивать за левое с вероятностью $\frac{L}{L+R}$, иначе за правое.

Теорема. Такое переподвешивание эквивалентно приоритетам.

Доказательство. Покажем, что все вершины всё так же имеют равную вероятность быть корнем. Докажем по индукции:

- Лист имеет вероятность 1 быть корнем себя (база индукции)
- Переход индукции — операция `merge`. Любая вершина левого дерева была корнем с вероятностью $\frac{1}{L}$ (по предположению индукции), а после слияния она будет корнем всего дерева с вероятностью $\frac{1}{L} \cdot \frac{L}{L+R} = \frac{1}{L+R}$. С вершинами правого дерева аналогично.

Получается, что при таком переподвешивании всё так же каждая вершина любого поддерева равновероятно могла быть его корнем, а на этом основывалось наше доказательство асимптотики ДД.

```
Node* merge (Node *l, Node *r) {
    if (!l) return r;
    if (!r) return l;
    l = copy(l), r = copy(r);
    if (rand() % (size(l) + size(r)) < size(l)) {
        // ...
    }
    else {
        // ...
    }
}
```

Философский вопрос: можно ли декартово дерево называть декартовым, если из него удалить и x, и y?