Зимняя компьютерная школа 2015. Лекция группы B про декартово дерево

Дмитрий Иващенко, Константин Семенов

Оглавление

Введение

Многие, наверное, знают о существовании структуры данных std::set и ее времени работы. Кто-то, наверное, знает и о ее внутреннем устройстве: красно-черном дереве. Нашей задачей будет построить и научиться быстро и безошибочно реализовывать структуру данных, не уступающую по силе стандартному контейнеру. Кратко опишем, что мы научимся делать:

- Упорядоченное по возрастанию множество объектов. Поиск и вставка элементов за $O(\log n)$.
- Вычисление какой-либо функции (например, суммы) от всех чисел в множестве, лежащих от l до r за $O(\log n)$.
- Поиск k-го по величине элемента в множестве за $O(\log n)$.
- Реализация структуры данных «супермассив» с возможностью произвольного разрезания, склеивания, перестановки и переворота отдельных его частей за $O(\log n)$.
- Решение задач на отрезках (RMQ) в таком массиве за $O(\log n)$ на запрос.
- Сохранение всех версий дерева в процессе работы с ним. Произвольное обращение к любой из предыдущих версий (read/write persistentcy). Каждая операция за $O(\log n)$ времени и дополнительной памяти. (To be done...)

Сразу сделаем оговорку, что наша структура будет существенно опираться на случайность, и мы получим асимптотические оценки времени работы только в среднем случае. Однако в реальности ее быстродействие будет вполне приемлемым, хоть оно и будет немного уступать аккуратной реализации дерева отрезков. Взамен этого мы научимся решать некоторые задачи, которые дереву отрезков недоступны.

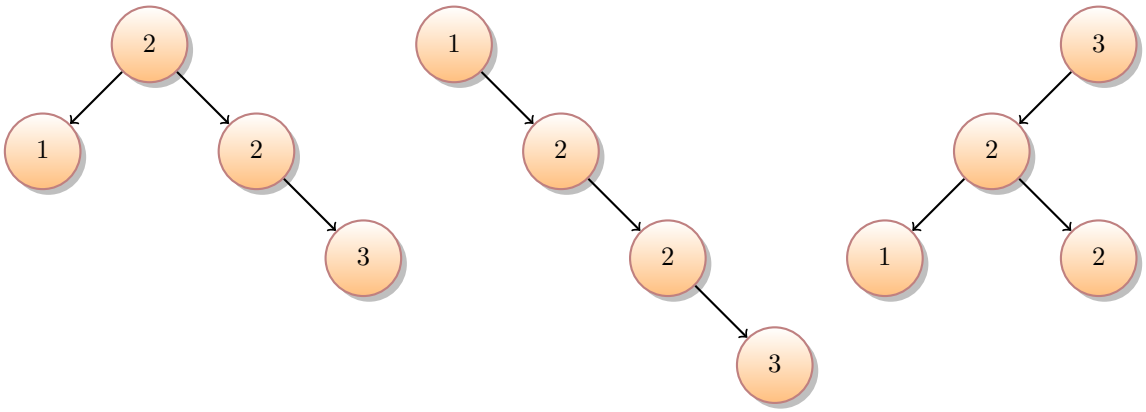
Бинарное дерево поиска

Определение. *Бинарным деревом поиска* называется подвешенное бинарное дерево, в вершинах которого расставлены так называемые *ключи*. Как правило это числа, однако, вместо них можно использовать произвольный тип данных, который позволяет осуществлять сравнение двух экземпляров. Далее мы будем везде считать, что ключи — это целые числа, сравнение их выполняется за $O(1)$ операций.

Заметим также, что дополнительно в вершине может храниться и другая информация. Например, если ключом является какое-то число, то дополнительно может храниться количество раз, которое оно входит в наше множество, а точнее мультимножество. Также, например, если мы храним в бинарном дереве какие-нибудь отрезки на прямой, то ключом может являться левая координата, а правая будет храниться как дополнительная информация.

Распределение ключей в дереве поиска не может быть произвольным, выполняется *инвариант дерева поиска*: для произвольной вершины v все ключи в ее левом поддереве меньше, чем ключ вершины v, а все ключи в правом поддереве больше или равны.

Замечание. Стоит заметить, что бинарных деревьев поиска с одним и тем же набором ключей бывает несколько. Ниже приведены примеры деревьев поиска на множестве ключей $\{1, 2, 2, 3\}$



Зачем же нужны бинарные деревья поиска? Ответ явствует из названия: они позволяют искать произвольный ключ простым спуском по дереву. Ясно, что если искомый ключ не находится в текущей вершине, то одним сравнением мы можем определить, куда именно нужно спуститься. Также несложно реализовать вставку элемента в дерева на какое-нибудь подходящее место. Для примера приведем код, иллюстрирующий, как мы будем хранить дерево и как будем производить операции.

```
struct tree_node
{
    tree_node *left, *right;
    int key;

    tree_node(int new_key)
    {
        key = new_key;
        left = right = nullptr;
    }
};
```

Структура для хранения дерева выглядит весьма просто. Так удобно хранить и декартово дерево тоже. Предка мы хранить не будем, в нашей реализации он не понадобится.

Для удобства сделаем конструктор, создающий вершину с данным ключом и не имеющую детей.

```
bool search(tree_node *v, int key)
{
    if (v == nullptr) return false;
    if (v->key == key) return true;
    if (key < v->key)
        return search(v->left, key);
    else
        return search(v->right, key);
}
```

Функция поиска элемента в поддереве принимает вершину и искомый ключ.

Реализация прозрачна: если дерево пусто, то ключа в нем, нет. Также тривиален случай, если ключ текущей вершины равен искомому. Иначе нам нужно перейти налево или направо, в зависимости от значения искомого ключа.

```
void insert(tree_node &*v, int key)
{
    if (v == nullptr)
    {
        v = new node(key);
        return;
    }
    if (key < v->key)
        insert(v->left, key);
    else
        insert(v->right, key);
}
```

Функция вставки принимает корень поддерева и ключ. Обратите внимание, корень нужно передавать по ссылке, чтобы иметь возможность изменить его вне функции или в рекурсивном вызове.

Опять, вставка в пустое дерево тривиальна, в противном случае мы смотрим, в какое поддерево можно вставить, не нарушив инварианта дерева и переходим в выбранного сына.

Ясно, что все время работы всех операций с деревом напрямую зависит от его высоты. Однако, наша реализация дерева поиска может порождать деревья, вырожденные в цепочки, поиск в которых будет работать как линейный поиск. Во избежании этого используют различные техники *балансировки* дерева. Одна из таких техник используется и в декартовом дереве.

Приоритеты. Инвариант декартова дерева

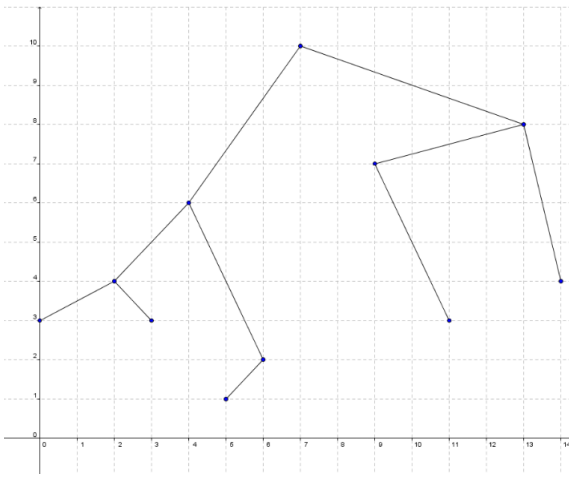
Определение бинарного дерева поиска допускает много различных конфигураций для одних и тех же ключей. Среди этих конфигураций встречаются как «хорошие», небольшой высоты, так и «плохие», вытянутые в какую-либо сторону. Интуитивно ясно, что в случайно выбранной конфигурации высота будет не очень высокой. Оказывается, это утверждение можно формализовать и доказать, а это соображение и составляет основную идею декартова дерева.

Определение. Назначим каждой вершине *приоритет* — некоторое число, уникальное для каждой вершины. Потребуем теперь помимо обычного инварианта дерева поиска выполнения следующего *инварианта декартова дерева*: приоритет вершины больше, чем приоритет ее сыновей. Иными словами, мы требуем, чтобы по ключам наше дерево было *двоичным деревом поиска*, а по приоритетам *кучей* (это дало структуре несколько жаргонных названий как в русском, там и в английском языках: treap, дуча, дерамида). Следующее утверждение поясняет смысл такого определения.

Утверждение. Конфигурация декартова дерева с различными приоритетами определена однозначно.

Доказательство. В самом деле, легко заметить, что корень дерева определен однозначно: это просто вершина с наибольшим приоритетом. В левом его поддереве будут находится все вершины с меньшими ключами, в правом — с большими либо с равными. Так, проводя рассуждение по индукции, мы однозначно определим, как устроено декартово дерево с данными ключами и приоритетами. □

Замечание. В свете этого утверждения структуру дерева можно легко изобразить на плоскости, отложив по оси Ox ключи, а по оси Oy приоритеты, отчего его и называют декартовым.



Ну и наконец модифицируем структуру вершины дерева под наши нужды:

```
struct tree_node
{
    tree_node *left, *right;
    int key, priority;

    tree_node(int new_key)
    {
        key = new_key;
        priority = rand();
        left = right = nullptr;
    }
};
```

Замечание. Будьте осторожны на платформах, где rand() возвращает 15 случайных бит. В дальнейших рассуждениях мы рассчитываем на то, что все приоритеты различны. Нарушения этого правила редки, если генерировать случайные числа порядка 10^9, несколько случайных совпадений на практике не влияют на быстродействие. Однако, если генерировать числа порядка 10^4, то коллизии случаются часто и это может повлиять на время работы.

Так мы избавились от одной проблемы — неоднозначности построения. Однако, остается вопрос, каким образом выбрать приоритеты, чтобы дерево получилось достаточно сбалансированным. Оказывается, хорошим выбором приоритетов является случайный выбор. Чтобы не быть голословными, сформулируем это в виде следующей теоремы. Если читатель не слишком силен в теории вероятностей, в утверждение в принципе можно просто поверить, однако для полноты картины мы все же ее докажем.

Теорема. *В декартовом дереве из n вершин, приоритеты которого являются равномерно распределенными случайными величинами математическое ожидание высоты (средняя высота) вершины есть $O(\log n)$.*

Доказательство. Пусть ключи без потери общности — это числа от 1 до n, причем для удобства i для $1 \leqslant i \leqslant n$ — вершина с ключом, равным i. Введем случайную величину

$$P_{i,j} = \begin{cases} 1, & i \text{ предок } j \\ 0, & \text{иначе} \end{cases}$$

Глубину вершины $d(i)$ теперь можно расписать как количество ее предков: $d(i) = \sum\limits_{j=1}^{n} P_{j,i}$.

Теперь нам нужно понять, когда вершина i является предком вершины j в терминах приоритетов.

Оказывается, это происходит, если вершина i имеет наибольший приоритет среди вершин отрезка $[i \dots j]$ (если $i > j$ эта запись обозначает отрезок $[j \dots i]$).

Если i имеет наибольший приоритет на отрезке, то если это корень, то i действительно предок j. Если i не корень, то i и j обязательно лежат в разных поддеревьях, так как иначе корень дерева лежит на отрезке $[i \dots j]$ и имеет больший приоритет. Тогда мы можем перейти в поддерево, где лежат i и j и продолжить рассуждения в нем по индукции.

Наоборот, если i является предком j, то если j находится в левом поддереве ($j < i$), то и все вершины отрезка $[i \dots j]$ находятся слева, а значит умеют приоритет меньше. Аналогично, если j лежит справа.

Тогда мы можем посчитать вероятность того, что i является предком j. Пользуясь полученным критерием, отметим, что все вершины отрезка $[i \dots j]$ равновероятно окажутся максимальными на нем, поэтому $Pr\left(P_{i,j} = 1\right) = \frac{1}{len([i \dots j])}$.

Теперь вычислим математическое ожидание глубины вершины: $Ed(i) = E\left(\sum\limits_{j=1}^{n} P_{j,i}\right)$. В силу линейности математического ожидания, получаем

$$Ed(i) = \sum_{j=1}^{n} EP_{j,i} = \sum_{j=1}^{n} Pr\left(P_{j,i} = 1\right) = \sum_{j=1}^{i-1} \frac{1}{i-j+1} + 1 + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \leqslant \ln\left(i\right) + \ln\left(n-i\right) + 3$$

В последнем неравенстве использован факт о сумме гармонического ряда: $\sum\limits_{i=1}^{n} \frac{1}{i} \leqslant 1 + \ln n$ (можно использовать более слабую и простую оценку с двоичным логарифмом). Итак, мы показали, что $Ed(i) \leqslant 3 + 2\ln n = O(\log n)$. $\square$

Операции с декартовым деревом: split и merge.

Мы показали существование, единственность нашей структуры, даже поняли, как нужно выбрать приоритеты. Однако, пока непонятно, например, как эффективно вставить элемент в дерево. Здесь удобно выразить все операции через две основные: операцию разделения *split* и обратную ей операцию *merge*.

Операция $split(T, x)$ принимает два аргумента — дерево T и некоторое значение x. Результатом работы этой операции являются два дерева L и R, первое из которых построено на всех вершинах T с ключом меньше x, а второе на всех остальных.

Как реализовать такую операцию? Как обычно, мы выражаем все операции рекурсивно. В данном случае мы точно знаем, в какое из двух деревьев попадет корень T. Пусть для определенности его ключ меньше x и корень попадает в дерево L. Но тогда все его левое поддерево можно оставить без изменения, потому что оно также должно лежать в дереве L. Значит нам нужно разобраться только с правым поддеревом. А разобраться с ним очень просто — достаточно рекурсивно вызвать процедуру *split* от него и подвесить левое из полученных деревьев к корню, правое же и есть целиком дерево R.

Случай, если корень попадает в правое поддерево полностью аналогичен. Это приводит нас к короткой и простой реализации. Мы приводим два варианта, предоставляя читателю выбрать наиболее понравившийся.

```
void split(tree_node *root, tree_node &*left, tree_node &*right, int key)
{
    if (root == nullptr)
    {
        left = right = nullptr;
        return;
    }
    if (root->key < key)
    {
        split(root->right, root->right, right, key);
        left = root;
    }
    else
    {
        split(root->left, left, root->left, key);
        right = root;
    }
}
```

Вторая реализация отличается от первой тем, что возвращает пару из полученных деревьев. Это может быть удобно, если вы хотите поддерживать предка в декартовом дереве и, возможно, такой код проще читать.

```
pair<tree_node*, tree_node*> split(tree_node *root, int key)
{
    if (root == nullptr)
        return make_pair(nullptr, nullptr);
    if (root->key < key)
    {
        pair<tree_node*, tree_node*> splitted = split(root->right, key);
        root->right = splitted.first;
        return make_pair(root, splitted.second);
    }
    else
    {
        pair<tree_node*, tree_node*> splitted = split(root->left, key);
        root->left = splitted.second;
        return make_pair(splitted.first, root);
    }
}
```

Поскольку рекурсивный вызов всегда только один, причем каждый раз мы спускаемся на один уровень в декартовом дереве, то сложность такой операции *split* пропорциональна высоте дерева и составляет $O(\log n)$ в среднем.

Теперь выразим обратную операцию *merge*, которая принимает два дерева L и R, все ключи первого из которых меньше или равны, чем все ключи второго, и строит по ним одно общее дерево (это действительно обратная операция, если применить ее к результату функции *split*, то мы получим обратно то же самое дерево).

Здесь мы поступим аналогично предыдущему случаю: мы точно знаем, какая из вершин будет новым корнем — это либо корень дерева L, либо корень дерева R, причем из них надо выбрать вершину с большим приоритетом. Пусть для определенности корнем дерева оказался корень R. Тогда правого сына R можно оставить в покое и просто слить L и левого сына R, подвесив результат к корню слева.

Код получается также короткий и простой. Приводим снова две реализации. Первая принимает *root*, $left$, *right* и кладет результат слияния $left$ и *right* в *root*.

```
void merge(tree_node &*root, tree_node *left, tree_node *right)
{
    if (left == nullptr || right == nullptr)
    {
        root = right == nullptr ? left : right;
        return;
    }
    if (left->priority > right->priority)
    {
        merge(left->right, left->right, right);
        root = left;
    }
    else
    {
        merge(right->left, left, right->left);
        root = right;
    }
}
```

Во второй функция *merge* возвращает дерево, полученное слиянием $left$ и $right$.

```
tree_node* merge(tree_node *left, tree_node *right)
{
    if (left == nullptr || right == nullptr)
        return right == nullptr ? left : right;
    if (left->priority > right->priority)
    {
        left->right = merge(left->right, right);
        return left;
    }
    else
    {
        right->left = merge(left, right->left);
        return right;
    }
}
```

Время работы этой операции точно также можно оценить суммой высот деревьев $left$ и $right$, то есть слияние деревьев размеров n и m произойдет в среднем за $O(\log n + \log m)$.

Замечание. Обратите внимание на ограничения, налагаемые функцией *merge* на свои аргументы. Они весьма принципиальны. К сожалению, слить произвольные декартовы деревья таким образом или какими-то небольшими модификациями не получится.

Замечание. Далее в примерах кода мы будем использовать и ту, и ту реализацию функций *split* и *merge*, коротко обозначая их просто «первая» и «вторая». Далее каждый пример кода мы будем приводить для какой-нибудь одной реализации, оставляя читателю в качестве упражнения модифицирование кода для другого подхода.

Выражение всех операций через split и merge

Возможно, это пока еще не очень понятно, но на самом деле мы уже обрели могущество. Давайте теперь поймем, как вставить новое значение x в дерево T. Для этого создадим дерево P с одной вершиной с ключом x. Теперь достаточно просто сделать $split(T, x)$, а потом сделать $merge$ полученных деревьев L, P, R именно в таком порядке. Проиллюстрируем операцию вставки следующим кодом (пользуемся второй реализацией).

```
void insert(tree_node &*root, int key)
{
    pair<tree_node*, tree_node*> splitted = split(root, key);
    merge(merge(splitted.first, new node(key)), splitted.second);
}
```

Есть альтернативный способ вставки вершины. Он слегка экономит время, потому что использует меньшее количество спусков по дереву. Мы просто будем спускаться по декартову дереву как по обычному дереву поиска до тех пор, пока не поймем, что приоритет текущей вершины меньше нашего и мы не можем вставить нашу вершину где-то ниже. Поддерево этой вершины обладает приоритетами меньше, чем приоритет вставляемый, поэтому мы можем просто вызвать функцию $split$ от него. Код получается немногим больше (используется первая реализация).

```
void insert(tree_node &*root, tree_node *vertex)
{                                                   Обратите внимание, фукнция
    if (root == nullptr)                            принимает уже готовую
    {                                               вершину, а не просто ключ.
        root = vertex;                              Важно, чтобы это была
        return;                                     вершина без детей.
    }
    if (root->priority > vertex->priority)          Вставка в пустое дерево
    {                                               тривиальна. Иначе, если
        if (vertex->key < root->key)                можно пойти вниз, то
            insert(root->left, vertex);             вставим вершину в нужного
        else                                        сына. Иначе, придется
            insert(root->right, vertex);            применить операцию split,
        return;                                     которая разобьет и подвесит
    }                                               полученные части дерева
    split(root, vertex->left, vertex->right);       к нашей вершине.
    root = vertex;
}
```

Научимся удалять элемент x из дерева. Для этого достаточно просто разделить дерево на три: L, M и R, так чтобы в L были ключи меньше x, в R ключи больше x, а в M только равные x. Тогда нам нужно удалить любой лист M или, например, сделать $M = merge(M.left, M.right)$.

После этого нужно не забыть склеить деревья обратно. Если же мы уверены, что все ключи уникальны, то достаточно будет просто вернуть $merge(L, R)$. Приведем пример кода (используется первая реализация).

```
void erase(tree_node &*root, int key)
{
    tree_node *left = nullptr, *middle = nullptr, *right = nullptr;
    split(root, left, middle, key);
    split(middle, middle, right, key + 1);
    merge(middle, middle->left, middle->right);
    merge(root, left, middle);
    merge(root, root, right);
}
```

Замечание. Обратите внимание, здесь нам пришлось сделать *split* по ключу $key + 1$. К сожалению, в случае сложного ключа не всегда очевидно, как получить «следующий» элемент, поэтому может понадобиться «нестрогая» реализация функции *split*, отправляющая в левое дерево все меньше или равные элементы. Также эту проблему можно обойти, если в дереве *right* пройти в самую левую вершину — это и будет наш ключ (мы предполагаем, что он есть в дереве). Мы можем легко удалить эту вершину, так как она является листом.

Точно также как и в прошлый раз, можно слегка ускорить удаление, не используя операции *split*, если спуститься в дереве до нужного элемента, а потом сделать для него $T = merge(T.left, T.right)$. Приводим код (для второй реализации).

```
void erase(tree_node &*root, int key)
{
    assert(root != nullptr);
    if (key < root->key)
        erase(root->left, key);
    else if (key > root->key)
        erase(root->right, key);
    else
        root = merge(root->left, root->right);
}
```

Замечание. *assert* в первой строчке указывает на то, что мы уверены, что удаляемый ключ есть в дереве. Если же операции удаления несуществующего ключа нужно просто пропускать, то в этой строке нужно просто выйти из функции без сообщения об ошибке.

Вот мы и реализовали все необходимые для начала операции, получив тем самым полный аналог std::set.

Замечание. Мы не реализовали пока что только одну вещь: итераторы std::set. Однако, это не очень сложно. Например, в качестве итератора можно использовать собственно указатель на вершину, где лежит наше значение, благо значение никогда не переходит из вершину в вершину. Чтобы пройти по всем числам можно использовать простой рекурсивный обход дерева — сперва рекурсивно идем в левого сына, следующее значение лежит в корне, далее нужно рекурсивно перейти направо.

Подсчет функции в поддереве

Зададимся следующей задачей: есть мультимножество чисел, поступают запросы на удаление и вставку, а также запросы «посчитать gcd всех чисел, значение которых лежит от a до b». Тут нам приходит на помощь техника подсчета функции в поддереве.

Добавим в нашу структуру для вершины еще одно значение *subtree_gcd*. Осталось научиться его нормально поддерживать. Поскольку все наши операции выражаются через *split* и *merge*, достаточно, чтобы они корректно пересчитывали все значения. Напишем функцию $update(v)$, которая в предположении, что информация у сыновей посчитана правильно, обновляет функцию в вершине v. Ее код на примере этой задачи выглядит так:

```
int get_gcd(tree_node *v)
{                                                   Для удобства реализуем функцию,
    return v == nullptr ? 0 : v->subtree_gcd;       возвращающую 0 в случае пустого
}                                                   дерева, иначе его substree_gcd

void update(tree_node *v)
{                                                   Функция обновления также будет
    if (v == nullptr) return;                       игнорировать запросы к пустым
    v->substree_gcd = gcd(get_gcd(v->left),         деревьям. Так как gcd(0, x) = x,
                          get_gcd(v->right));       то значение будет пересчитано
}                                                   корректно
```

Теперь будем поддерживать инвариант, что возвращаемые функциями *split* и *merge* деревья актуальны, то есть информация в них подсчитана корректно. Его и в самом деле легко поддержать: нужно просто после каждого рекурсивного вызова вызывать функцию *update*. Для всех вырожденных случаев пустых поддеревьев и вовсе не надо ничего делать. Приведем в качестве примера модифицированный код одной из операций каждой реализации.

```
void split(tree_node *root, tree_node &*left, tree_node &*right, int key)
{
    if (root == nullptr)
    {
        left = right = nullptr;
        return;
    }
    if (root->key < key)
    {
        split(root->right, root->right, right, key);
        left = root;
    }
    else
    {
```

```
        split(root->left, left, root->left, key);
        right = root;
    }
    update(root);
}
tree_node* merge(tree_node *left, tree_node *right)
{
    if (left == nullptr || right == nullptr)
        return right == nullptr ? left : right;
    if (left->priority > right->priority)
    {
        left->right = merge(left->right, right);
        update(left);
        return left;
    }
    else
    {
        right->left = merge(left, right->left);
        update(right);
        return right;
    }
}
```

Поддержка размеров поддеревьев

Раз уж мы научились считать любые (ассоциативные) функции в поддереве, то можем и реализовать самую простую из них: количество вершин в поддереве. Что это нам дает? Например, рассмотрим такую задачу: необходимо поддерживать множество чисел с добавлением и удалением, а также говорить k-е по величине число в множестве.

После того, как мы имеем всегда актуальный размер поддерева, мы можем эффективно реализовать последнюю операцию. В самом деле, она вырождается в простой спуск по дереву в ту часть, где расположено искомое число (как раз для определения, в какой части оно лежит нам и нужны посчитанные размеры поддеревьев).

```
int get_size(tree_node *v)
{
    return v == nullptr ? 0 : v->size;
}

void update(tree_node *v)
{
    if (v == nullptr) return;
    v->size = get_size(v->left) + get_size(v->right) + 1;
}

int nth_element(tree_node *v, int n)
{
    assert(v != nullptr);
    int root_number = get_size(v->left);
    if (root_number == n)
        return v->key;
    if (n < root_number)
        return nth_element(v->left, n);
    else
        return nth_element(v->right, n - root_number - 1);
}
```

Замечание. Функция *nth_element* принимает n в 0-индексации. Также, если необходимо, можно убрать *assert* в первой строке, заменив его на какой-либо другой обработчик отсутствия достаточного количества элементов во множестве.

Декартово дерево по неявному ключу

Давайте проведем такой эксперимент: неявное декартово дерево на ключах $\{0, \dots, n-1\}$ и каких-нибудь приоритетах и подсчитаем размеры всех поддеревьев. А теперь сотрем все ключи. Можно ли будет их восстановить? Да, конечно. Мы, пользуясь лишь структурой дерева можем пройти по всем числам в порядке возрастания и восстановить их ключ. Это наводит нас на странную мысль, что ключ иногда можно не хранить.

Практически полезным это оказывается при решении задачи нахождения минимума на отрезке (RMQ). Нам нужно находить минимум на отрезке массива $[l; r]$. Давайте построим декартово дерево на ключах $\{0, \dots, n-1\}$, причем в i-й вершине сохраним значение исходного массива $a[i]$. Тогда, подсчитав минимум в поддереве (надо не забыть добавить все необходимые обновления в функцию *update*), мы можем за две операции *split* получить дерево с ключами от l до r и узнать минимум в нем. Также, если мы знаем размеры поддеревьев, то мы можем легко найти в дереве вершину по конкретному индексу и, например, обновить значение в ней (не забыв пересчитать всех ее предков).

Казалось бы, какое-то другое решение задачи RMQ, деревья отрезков все равно быстрее и проще, однако, прелесть этого подхода в том, что можно забыть о существовании ключей. Если бы мы могли реализовать *split* и *merge* так, чтобы они работали без ключей, то, например, если мы имеем дерево L для массива a и дерево R для массива b, то чему будет соответствовать дерево $merge(L, R)$? Оказывается, что это будет просто конкатенация (приписывание) ab. В самом деле, функция *merge* считает, что все ключи L меньше, чем все ключи R, то есть можно условно назначить дереву L ключи от 0 до $n-1$, а R — от n до $n+m-1$, тогда действительно получится дерево с ключами от 0 до $n+m-1$ причем первые n элементов соответствуют элементам из a, остальные — элементам из b.

Аналогично, $split(T, x)$ будет разбивать дерево T, соответствующее нашему массиву a, на два дерева L и R, разбив тем самым массив a на части $[0 \dots x-1]$ и $[x \dots n-1]$. Получается довольно мощная структура, мы можем, взяв массив, любым образом разбивать его на части и произвольно соединять их.

Осталось реализовать *split* и *merge*. Однако *merge* вообще никак не использует ключи, значит нам нужно только придумать, как делать операцию *split*, если ключи явно не хранятся. Это очень просто: ведь для того, чтобы понять, куда спускаться, нам нужно знать только ключ корня дерева. А ключ корня дерева это просто размер его левого поддерева.

Теперь если корень попадает в дерево L, то мы должны разбить дерево R по ключу $x - key(root) - 1$, а иначе нужно разбить дерево L по ключу x. Приведем модифицированную операцию *split* (для первой реализации).

```
void split(tree_node *root, tree_node &*left, tree_node &*right, int key)
{
    if (root == nullptr)
    {
        left = right = nullptr;
        return;
    }
    int root_key = get_size(root->left);
    if (root_key < key)
    {
        split(root->right, root->right, right, key - root_key - 1);
        left = root;
    }
    else
    {
        split(root->left, left, root->left, key);
        right = root;
    }
    update(root);
}
```

В качестве классического применения полученной структуры можно указать задачу о циклическом сдвиге на подотрезке: требуется поддерживать массив с обращением к произвольному элементу и уметь применять произвольные циклические сдвиги к любому его подотрезку. В самом деле, пользуясь описанной техникой, операцию сдвига можно выразить через одну операцию *split* и две операции *merge*. Конкретную реализацию оставим читателю в качестве упражнения.

Бывают также и более интересные идеи, эксплуатирующие декартово дерево. К примеру, нужно построить структуру данных, которая может находить сумму на отрезке, а также осуществлять операцию перестановки следующего вида: на отрезке четной длины $[l; r]$ поменять местами элементы l и $l + 1$, $l + 2$ и $l + 3$, $\ldots$, $r - 1$ и r.

Для решения этой задачи предлагается поддерживать два декартовых дерева по неявному ключу. В первом дереве мы будем хранить элементы с четными индексами, а во втором с нечетными. Чем удобно такое хранение? Тем, что теперь запрос обмена на отрезке $[l; r]$ можно осуществить, просто вырезав из четного дерева часть с $\lceil \frac{l}{2} \rceil$ до $\lfloor \frac{r}{2} \rfloor$, из нечетного дерева часть с $\lfloor \frac{l}{2} \rfloor$ по $\lfloor \frac{r-1}{2} \rfloor$ и обменять их местами. В самом деле, легко отследить, что таким образом мы просто поменяем четность нужной группы элементов.

Сумму на отрезке легко разбить на два запроса: сумма элементов с четными индексами на $[l; r]$ и с нечетными. Так, мы получили еще одно довольно интересное применение декартова дерева.

Групповые операции и неявный ключ

Сейчас мы сделаем отложенные операции обновления на отрезках и декартово дерево полностью догонит дерево отрезков по функциональности. Идея та же самая: пусть в вершине хранится какая-то информация *post_update*, которая будет лениво проталкиваться вниз по дереву по необходимости. Опять же важно уметь быстро пересчитывать значение на отрезке, к которому применено преобразование, а также уметь вычислять композицию двух отложенных преобразований (также как в дереве отрезков). Все, что нам нужно сделать, это написать операцию $push(v)$, которая обновляет значение в вершине сообразно отложенной операции и проталкивает ее детям, если они существуют.

Представим, что мы пишем неявное декартово дерево для минимума на отрезке (поле *value* для хранения значения вершины и *min_value* для хранения минимума в поддереве) с отложенной операцией прибавления на отрезке (поле *post_update*). Тогда операция *push* будет выглядеть примерно так

```
void update(tree_node *v)
{
    if (v == nullptr) return;
    v->size = get_size(v->left) + 1 +
              get_size(v->right);                  В update мы обязаны пересчитать
    v->min_value = min(v->value,                   и размер, и min_value
                   min(get_min_value(v->left),
                       get_min_value(v->right));
}

void push(tree_node *v)
{                                                  Операция push игнорирует пустые
    if (v == nullptr) return;                      деревья. Если дерево непусто, то
    if (v->left != nullptr)                        мы должны протолкнуть обновление
        v->left->post_update += v->post_update;    потомкам и пересчитать значение
    if (v->right != nullptr)                       в корне дерева. Важно не забыть
        v->right->post_update += v->post_update;   последнее присваивание, которое
    v->min_value += v->post_update;                говорит, что отложенных
    v->post_update = 0;                            обновлений у вершины больше нет.
}
```

Осталось только применить операцию *push* ко всем входным параметрам в начале функций *split* и *merge*.

Стоит рассмотреть одну интересную групповую операцию, которую не умеет дерево отрезков — это переворот подотрезка. Для этого в качестве *post_update* мы храним булевский флаг — надо ли переворачивать наш подотрезок. Единственная хитрость заключается в том, чтобы проталкивать информацию детям: если наш отрезок нужно перевернуть, то наших детей нужно перевернуть и при этом поменять местами. Операция *push* будет выглядеть так

```
void push(tree_node *v)
{
    if (v == nullptr || !v->inversed) return;
    if (v->left != nullptr)
        v->left->inversed ^= true;
    if (v->right != nullptr)
        v->right->inversed ^= true;
    swap(v->left, v->right);
    v->inversed = false;
}
```

Хранение предка

В самом начале мы сказали, что не будем хранить предка — действительно, до сих пор мы всегда ходили по дереву только вниз, но иногда пригождается и умение ходить вверх. Например, рассмотрим такую задачу: есть граф, в котором происходят добавления и удаления ребер, при этом гарантируется, что степень вершины никогда не превосходит 2. Необходимо отвечать на запросы кратчайшего пути между двумя вершинами.

Для начала вспомним, что граф, где степени вершин не превосходят 2, представляет собой объединение путей и циклов. Будем хранить пути и циклы в декартовом дереве по неявному ключу. Путь храним в естественном порядке, а цикл с какого-нибудь произвольного места, причем сохраним отметку о том, что это цикл. Сохраним также для каждой вершины графа указатель на соответствующую вершину дерева.

Для каждой вершины теперь можно вычислить ее ключ: достаточно подняться от нее вверх и посчитать суммарный размер поддеревьев левее данной вершины. Тогда легко реализовать запрос расстояния — нужно вычислить позицию каждой из вершин в своем дереве, заодно проверив в одном ли дерево они находятся (сравнив указатели на корень).

Запросы добавления и удаления ребра тоже легко выражаются: удаление ребра разбивает путь на два, что соответствует одной операции *split*. Добавление ребра может либо замкнуть путь в цикл (если соединяемые вершины лежат в одном дереве), либо объединить два пути в один (одна операция *merge*). Удаление ребра из цикла превращает его в путь (нужно не забыть циклически сдвинуть его, чтобы концы пути были первым и последним элементом дерева).

Технически, в этих операциях нет ничего нового, поэтому приведем здесь лишь код, описывающий, как реализовать хранение предка в операциях *split* и *merge* (во второй реализации).

```
pair<tree_node*, tree_node*> split(tree_node *root, int key)
{
    if (root == nullptr)
        return make_pair(nullptr, nullptr);
    if (root->key < key)
    {
        pair<tree_node*, tree_node*> splitted = split(root->right, key);
        root->right = splitted.first;
        splitted.first->parent = root;
        splitted.second->parent = nullptr;
        return make_pair(root, splitted.second);
    }
    else
    {
        pair<tree_node*, tree_node*> splitted = split(root->left, key);
        root->left = splitted.second;
        splitted.second->parent = root;
        splitted.first->parent = nullptr;
        return make_pair(splitted.first, root);
```

```
    }
}
```