[версия для печати]

Декартово дерево: Часть 3. Декартово дерево по неявному ключу

Оглавление (на данный момент)

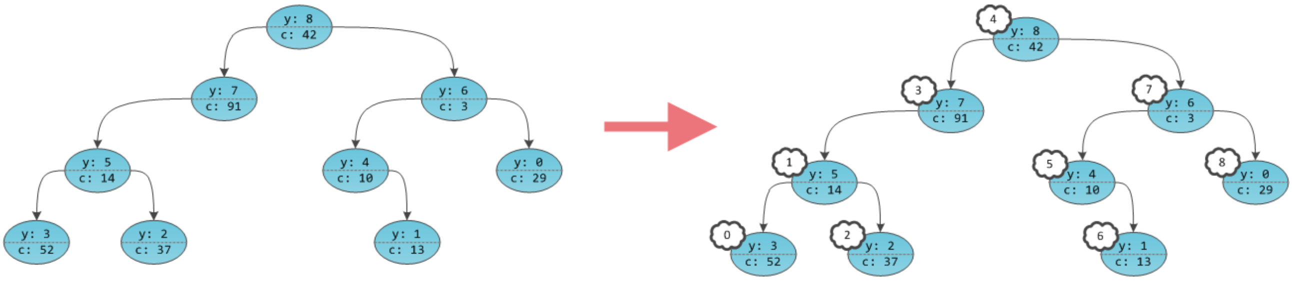
Очень сильное колдунство

После всей кучи возможностей, которые нам предоставило декартово дерево в предыдущих двух частях, сегодня я совершу с ним нечто странное и кощунственное. Тем не менее, это действие позволит рассматривать дерево в совершенно новой ипостаси — как некий усовершенствованный и мощный массив с дополнительными фичами. Я покажу, как с ним работать, покажу, что все операции с данными из второй части сохраняются и для модифицированного дерева, а потом приведу несколько новых и полезных.

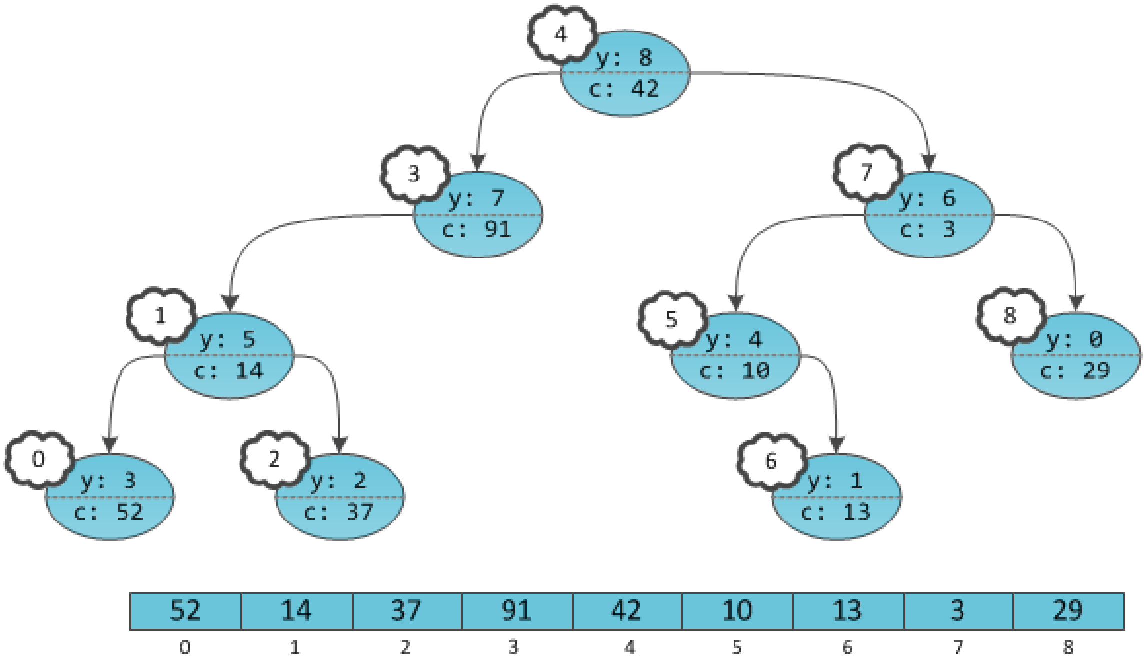
Вспомним-ка еще раз структуру дерамиды. В ней есть ключ **x**, по которому дерамида есть дерево поиска, случайный ключ **y**, по которому дерамида есть куча, а также, возможно, какая-то пользовательская информация **c** (cost). Давайте совершим невозможное и рассмотрим дерамиду… без ключей x. То есть у нас будет дерево, в котором ключа x нет вообще, а ключи y — случайные. Соответственно, зачем оно нужно — вообще непонятно :)

На самом деле расценивать такую структуру стоит как декартово дерево, в котором ключи x все так же где-то имеются, но нам их не сообщили.

Однако клянутся, что для них, как полагается, выполняется условие двоичного дерева поиска. Тогда можно представить, что эти неизвестные иксы суть числа от 0 до N-1 и *неявно* расставить их по структуре дерева:



Получается, что в дереве будто бы не ключи в вершинах проставлены, а сами вершины пронумерованы. Причем пронумерованы в уже знакомом с прошлой части порядке in-order обхода. Дерево с четко пронумерованными вершинами можно рассматривать как массив, в котором индекс — это тот самый неявный ключ, а содержимое — пользовательская информация `c`. Игреки нужны только для балансировки, это внутренние детали структуры данных, ненужные пользователю. Иксов *на самом деле* нет в принципе, их хранить не нужно.



В отличие от прошлой части, этот массив не приобретает автоматически никаких свойств, вроде отсортированности. Ведь на информацию-то у нас нет никаких структурных ограничений, и она может храниться в вершинах как попало.

Главные применения

Теперь стоит поговорить о том, зачем такая трактовка вообще нужна. Например, вам никогда не хотелось слить два массива? То есть просто приписать один из них в конец другого, при этом не копируя в цикле все элементы второго за O(N). С декартовым деревом по неявному ключу у вас такая возможность есть: ведь операцию Merge у нас никто не отбирал.

Стоп-стоп-стоп, но Merge ведь была написана для явного декартового дерева. Значит, её алгоритм придется здесь переработать? На самом деле нет. Посмотрите еще раз на её код.

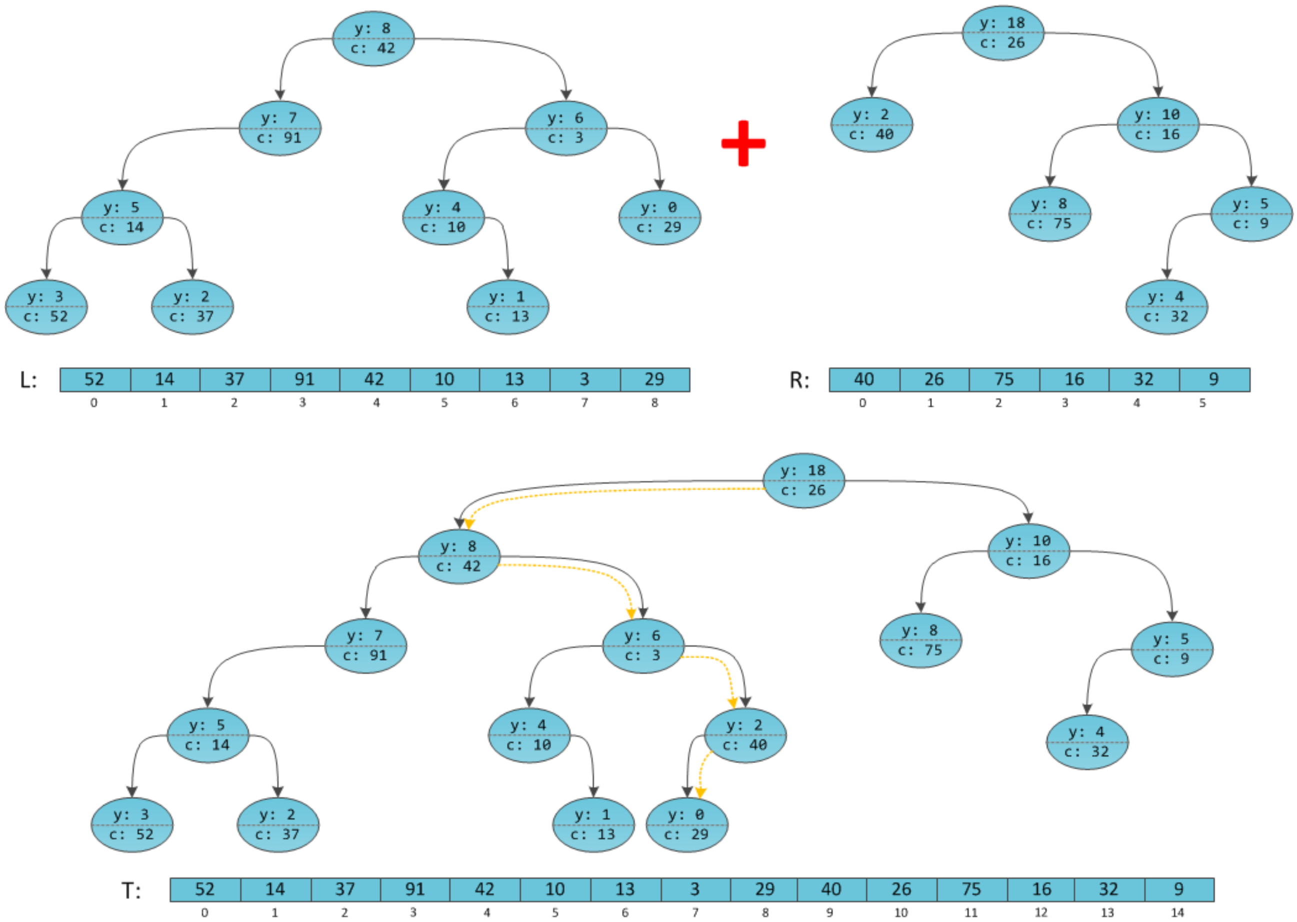
```
public static Treap Merge(Treap L, Treap R)
{
    if (L == null) return R;
    if (R == null) return L;

    if (L.y > R.y)
    {
        var newR = Merge(L.Right, R);
        return new Treap(L.x, L.y, L.Left, newR);
    }
    else
    {
        var newL = Merge(L, R.Left);
        return new Treap(R.x, R.y, newL, R.Right);
    }
}
```

Операция Merge, как вы помните, полагается на то, что все ключи левого входного дерева **L** не превосходят ключей правого входного дерева **R**. В предположении, что это условие выполняется, она производит слияние, не обращая внимания на ключи в принципе: в процессе выполнения алгоритма сравниваются лишь приоритеты.

Получается, что операцию Merge мы с вами здесь обманываем самым наглым образом: она рассчитывает, что ей дадут деревья с упорядоченными ключами, а мы ей подсовываем деревья вообще без ключей :) Однако предположение что в деревьях есть явные ключи, и они упорядочены, заставит её слить деревья так, что ключи L окажутся по структуре дерева в некотором смысле раньше, чем ключи R — ведь условие дерева поиска должно выполняться. То есть: если в дереве L было N элементов, а в дереве R, соответственно, M элементов, то после слияния элементы дерева R автоматически приобретут неявные номера от N до N+M-1. По структуре дерева операция Merge их распределит автоматически надлежащим образом, а приоритеты, которые она учитывает, выполнят «квази-балансировку». Таким образом, «массив» R мы как бы приписали справа к «массиву» L.

Что касается исходников, нам потребуется лишь завести новый тип данных `ImplicitTreap` без ключа `x`, и для него соответствующий приватный конструктор. Весь код Merge останется таким же. Разумеется, это если учитывать, что здесь приведена версия без вычисления множественных запросов — функции «восстановления справедливости» и «проталкивания обещаний», реализованные во второй части, также останутся в Merge на своих старых местах.

Для полной ясности я возьму два случайных неявных декартовых дерева и приведу их на рисунке вместе с результатом слияния. Приоритеты выбраны случайно, поэтому реальная структура обоих деревьев и результата может сильно отличаться. Но это неважно — структура массива, т.е. порядок следования элементов `c`, всегда сохраняется. Оранжевая стрелка — путь следования рекурсии в Merge.



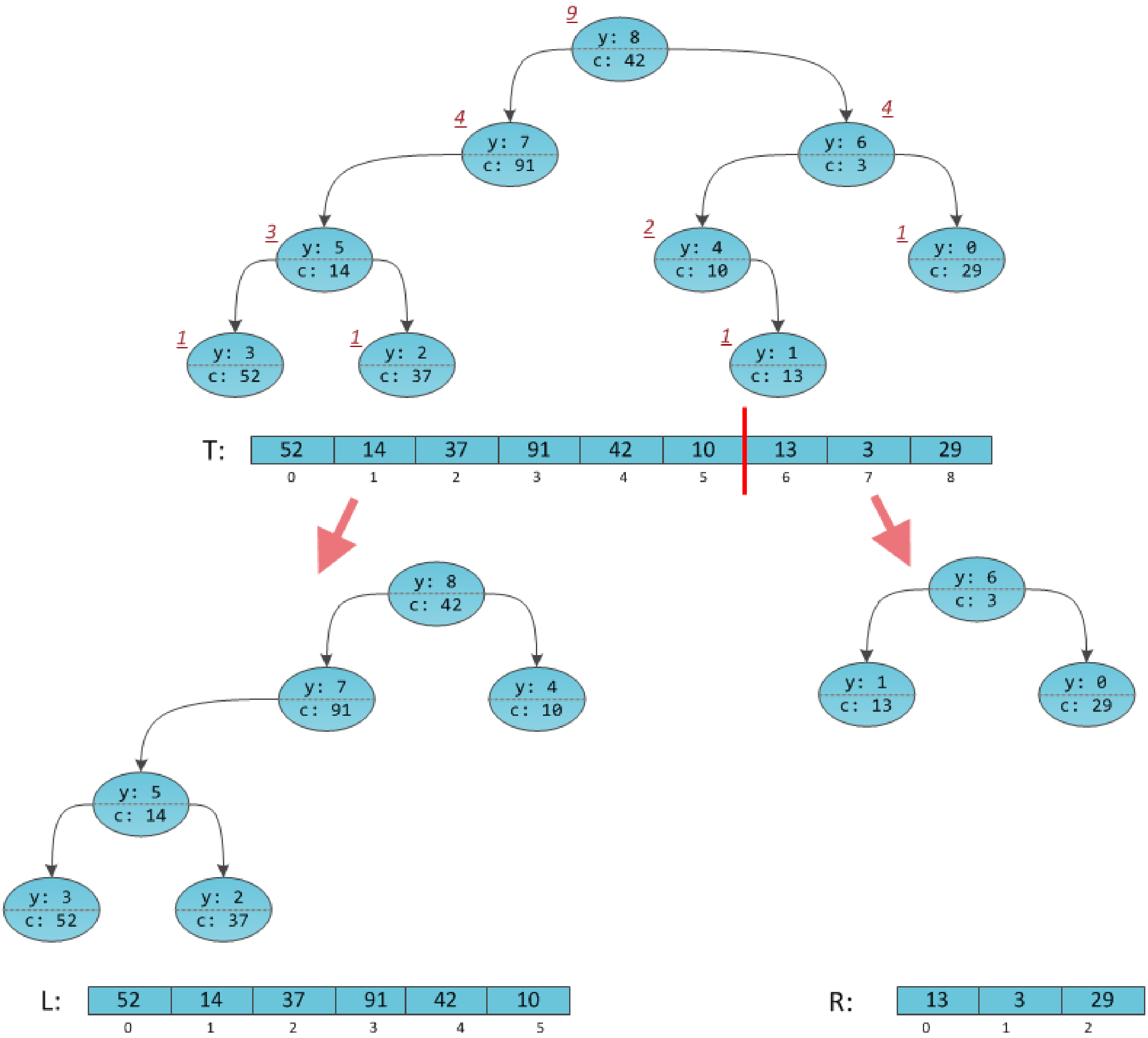
Теперь настала пора Split. Ее так просто обмануть уже не получится: операцию Split, наоборот, не интересуют приоритеты, она сравнивает лишь ключи. Придется задуматься, как она будет сравнивать вершины в неявном дереве. Хотя на самом деле проблема лежит выше: а что вообще выполняет в новой структуре данных операция Split? Раньше она разрезала дерево по ключу, но здесь у нас и ключей-то нет, по которым требуется разрезать.

Ключей нет, однако есть их неявное представление — индексы массива. Таким образом, суть разрезания несколько изменилась: мы хотим расщепить дерево на два так, чтобы в левом оказалось ровно $\mathbf{x_0}$ элементов, а в правом — все остальные. В «массивной» трактовке это означает отделение от массива x_0 элементов с начала в новый массив.

Как выполнить новую операцию Split? Она рассматривает, как и раньше, два случая: корень **T** окажется в левом результате **L** либо в правом **R**. В левом он окажется, если его индекс в массиве меньше x_0, в противном случае — в правом. А что такое индекс вершины дерева в массиве? Мы ведь с ним уже умеем работать со второй части: достаточно хранить в вершинах дерева *размеры поддеревьев*. Тогда процесс выбора легко восстанавливается.

Пусть $\mathbf{S_L}$ — размер левого поддерева (`T.Left.Size`).
Если $S_L+1 \le x_0$, то корень будет в левом результате. Значит, нужно рекурсивно разрезать правое поддерево. Но разрезать по другому ключу, по x_0-S_L-1, потому что S_L+1 элемент уже попал в искомый левый результат.
Если $S_L+1 > x_0$, то корень будет в правом результате. Теперь нужно рекурсивно разрезать левое поддерево. Этот случай не совсем симметричен, как раньше: разрезаем мы поддерево все по тому же ключу x_0, потому как на данном шаге рекурсии отщепляли элементы в правый результат, а не в левый.

На рисунке взято декартово дерево с проставленными размерами поддеревьев и расщеплено по $x_0 = 6$.



Исходный код нового Split, вместе с новой заготовкой класса — без ключа и с другим приватным конструктором.
Split даю снова пока что без множественных операций — читатель может восстановить эти строчки самостоятельно, они со своих старых мест никуда не подевались. Зато нельзя забывать о пересчете размеров поддеревьев.

```
private int y;
public double Cost;

public ImplicitTreap Left;
public ImplicitTreap Right;
public int Size = 1;

private ImplicitTreap(int y, double cost, ImplicitTreap left =
null, ImplicitTreap right = null)
{
```

```
    this.y = y;
    this.Cost = cost;
    this.Left = left;
    this.Right = right;
}

public static int SizeOf(ImplicitTreap treap)
{
    return treap == null ? 0 : treap.Size;
}

public void Recalc()
{
    Size = SizeOf(Left) + SizeOf(Right) + 1;
}

// теперь после заготовки - собственно Split

public void Split(int x, out ImplicitTreap L, out ImplicitTreap
R)
{
    ImplicitTreap newTree = null;
    int curIndex = SizeOf(Left) + 1;

    if (curIndex <= x)
    {
        if (Right == null)
            R = null;
        else
            Right.Split(x - curIndex, out newTree, out R);
        L = new ImplicitTreap(y, Cost, Left, newTree);
        L.Recalc();
    }
    else
    {
        if (Left == null)
            L = null;
        else
            Left.Split(x, out L, out newTree);
        R = new ImplicitTreap(y, Cost, newTree, Right);
        R.Recalc();
```

```
    }
}
```

Теперь, после того как мы написали для массива Split и Merge, работающие за логарифмическое время, настала пора их где-нибудь применить. Давайте поиграемся с массивом.

Игры с массивом

Вставка

Фокус №1 — мы вставим элемент внутрь массива на требуемую позицию **Pos** за $O(\log_2 N)$, а не за O(N), как обычно.

В принципе, мы это уже умеем делать с обычными декартовыми деревьями, только теперь на месте ключа стал индекс. А в остальном процедура не изменилась.

• Разрезаем массив T[0; N) по индексу Pos на массивы L[0; Pos) и R[Pos; N).
• Делаем из вставляемого элемента массив-дерево из одной вершины.
• Приписываем созданный массив справа к левому результату L, а к ним обоим — правый результат R.
• Получили массив T'[0; N+1), в котором на позиции Pos стоит искомый элемент, а остальная правая часть смещена.

Исходный код вставки не изменился совершенно.

```
public ImplicitTreap Add(int pos, double elemCost)
{
    ImplicitTreap l, r;
    Split(pos, out l, out r);
    ImplicitTreap m = new ImplicitTreap(rand.Next(), elemCost);
    return Merge(Merge(l, m), r);
}
```

Удаление

Фокус №2 — вырежем из массива элемент, стоящий на данной позиции

Pos.

Опять-таки, процедура та же, что и с обычными декартовыми деревьями.

• Разрезаем массив T[0; N) по индексу Pos на массивы L[0; Pos) и R[Pos; N).
• Правый результат R разрезаем по индексу 1 (единица!). Получаем массив M[Pos; Pos+1) из одного элемента (стоявшего ранее на позиции Pos), и массив R'[Pos+1; N).
• Сливаем массивы L и R'.

Исходный код удаления:

```
public ImplicitTreap Remove(int pos)
{
    ImplicitTreap l, m, r;
    Split(pos, out l, out r);
    r.Split(1, out m, out r);
    return Merge(l, r);
}
```

Множественные запросы на отрезке

Фокус №3: за O($\log_2$ N) можно также выполнять все те же множественные запросы на подотрезках массива (сумма/максимум/минимум/наличие или количество меток и т.д.).

Структура дерева не меняется с прошлой части: в вершине храним параметр, соответствующий искомой величине, вычисленной для всего подотрезка. В конце Merge и Split вставляется такой же вызов `Recalc()`, пересчитывающий значение величины в вершине на основании вычисленных параметров в ее потомках.

Запрос на отрезке [A; B) использует стандартный метод: вырезать из массива искомый отрезок (не забыв, что после первого разреза искомый индекс в правом результате уменьшился!) и вернуть значение параметра, хранящееся в его корне.
Исходный код — как пример, для максимума.

```
public double MaxTreeCost;
```

```
public static double CostOf(ImplicitTreap treap)
{
    return treap == null ? double.NegativeInfinity :
treap.MaxTreeCost;
}

public void Recalc()
{
    Size = SizeOf(Left) + SizeOf(Right) + 1;
    MaxTreeCost = Math.Max(Cost, Math.Max(CostOf(Left),
CostOf(Right)));
}

public double MaxCostOn(int A, int B)
{
    ImplicitTreap l, m, r;
    this.Split(A, out l, out r);
    r.Split(B - A, out m, out r);
    return CostOf(m);
}
```

Множественные операции на отрезке

Фокус №4: теперь за $O(\log_2 N)$ мы будем на подотрезках массива выполнять операции из второй части: прибавление константы, покраску, установку в единое значение и т.д.

Имея рабочие Merge и Split, реализация отложенных вычислений в декартовом дереве совершенно не меняется. Основный принцип работы тот же: перед выполнением любой операции «протолкнуть обещание» потомкам. Если дополнительно нужно поддерживать еще и множественные запросы из предыдущего раздела, после выполнения операции необходимо «восстановить справедливость».

Для выполнения операции на отрезке нужно этот отрезок сначала вырезать из дерева двумя вызовами Split, а потом снова вставить его двумя вызовами Merge.
Для ленивых привожу полный исходный код для прибавления, вместе с новыми реализациями Merge/Split (они отличаются аж на 1-2 строки), а

также с функцией проталкивания `Push`:

```
public double Add;

public static void Push(ImplicitTreap treap)
{
    if (treap == null) return;
    treap.Cost += treap.Add;
    if (treap.Left != null) treap.Left.Add += treap.Add;
    if (treap.Right != null) treap.Right.Add += treap.Add;
    treap.Add = 0;
}

public void Recalc()
{
    Size = SizeOf(Left) + SizeOf(Right) + 1;
}

public static ImplicitTreap Merge(ImplicitTreap L, ImplicitTreap
R)
{
    // проталкивание!
    Push( L );
    Push( R );

    if (L == null) return R;
    if (R == null) return L;

    ImplicitTreap answer;
    if (L.y > R.y)
    {
        var newR = Merge(L.Right, R);
        answer = new ImplicitTreap(L.y, L.Cost, L.Left, newR);
    }
    else
    {
        var newL = Merge(L, R.Left);
        answer = new ImplicitTreap(R.y, R.Cost, newL, R.Right);
    }

    answer.Recalc();
```

```
    return answer;
}

public void Split(int x, out ImplicitTreap L, out ImplicitTreap
R)
{
    Push(this); // проталкивание!

    ImplicitTreap newTree = null;
    int curIndex = SizeOf(Left) + 1;

    if (curIndex <= x)
    {
        if (Right == null)
            R = null;
        else
            Right.Split(x - curIndex, out newTree, out R);
        L = new ImplicitTreap(y, Cost, Left, newTree);
        L.Recalc();
    }
    else
    {
        if (Left == null)
            L = null;
        else
            Left.Split(x, out L, out newTree);
        R = new ImplicitTreap(y, Cost, newTree, Right);
        R.Recalc();
    }
}

public ImplicitTreap IncCostOn(int A, int B, double Delta)
{
    ImplicitTreap l, m, r;
    this.Split(A, out l, out r);
    r.Split(B - A, out m, out r);
    m.Add += Delta;
    return Merge(Merge(l, m), r);
}
```

Небольшое отступление про разрешенные операции:

> Фактически, рекурсивная структура дерева и отложенные вычисления позволяют реализовать любую операцию моноида. **Моноид (http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%BE%**
> — множество с заданной на нем бинарной операцией ◦, которая обладает следующими свойствами:
> • Ассоциативность — для любых элементов a, b, c имеем (a ◦ b) ◦ c = a ◦ (b ◦ c).
> • Существование нейтрального элемента — в множестве есть такой элемент e, что для любого элемента a имеет место a ◦ e = e ◦ a = a.
> Тогда для такой операции возможно реализовать **дерево отрезков (http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D**
> , и по аналогичным причинам — декартово дерево.

Переворот массива

Фокус № 5 — переворот подотрезка, то есть перестановка его элементов в обратном порядке.

А на этом месте я остановлюсь чуть поподробнее. Разворот массива не является моноидальной операцией — откровенно говоря, это вообще не бинарная операция на каком бы то ни было множестве — но тем не менее. Уникальность этой задачи в том, что для неё можно придумать функцию проталкивания. А раз существует возможность протолкнуть операцию — значит, можно реализовать её как отложенную.

Итак, будем хранить в каждой вершине булево значение — бит перевернутости. Это будет отложенное обещание «данный отрезок массива в будущем необходимо развернуть». Тогда, в предположении что мы умеем проталкивать этот бит к потомкам своеобразной версией функции `Push`, дерево всегда остается в актуальном виде — перед любыми операциями доступа к элементам массива (поиск), а также в начале Merge и Split выполняется проталкивание. Осталось выяснить, как производить это «выполнение обещания».

Пусть в некоторой вершине **T** стоит обещание о переворачивании подотрезка. Чтобы его начать фактически выполнять, достаточно сделать следующее:
• Снять обещание в текущей вершине:

```
T.Reversed = false;
```

- Поменять местами его левого и правого сыновей.

```
temp = T.Left;
T.Left = T.Right;
T.Right = temp;
```

- Изменить обещание у потомков. Обратите внимание: не установить в true (мы не знаем, стоял ли этот бит у потомков до сих пор!), а изменить. Для этого используется операция ^.

```
T.Left.Reversed ^= true;
T.Right.Reversed ^= true;
```

Действительно, что такое «фактически перевернуть массив»? Возьмем два куска этого массива (поддеревья), поменяем их местами в реальности, и *пообещаем* перевернуть эти два подмассива в будущем. Нетрудно заметить, что, когда все обещания таки исполнятся до единого, элементы исходного массива окажутся в перевернутом порядке.

Заметьте — с обычными декартовыми деревьями такую махинацию проводить нельзя, поскольку мы нарушаем свойство дерева поиска — ключи в правом поддереве оказываются меньше ключей в левом. Но поскольку в неявном декартовом дереве ключей нет вообще, а для индексов свойство соблюдается всегда, то в дереве ничего не ломается.

Пользовательская функция переворачивания отрезка работает по неизменному принципу, как и любая другая операция: вырезать требуемый отрезок, установить в его корне обещание, и вклеить отрезок обратно. Вот исходный код функций проталкивания и переворачивания:

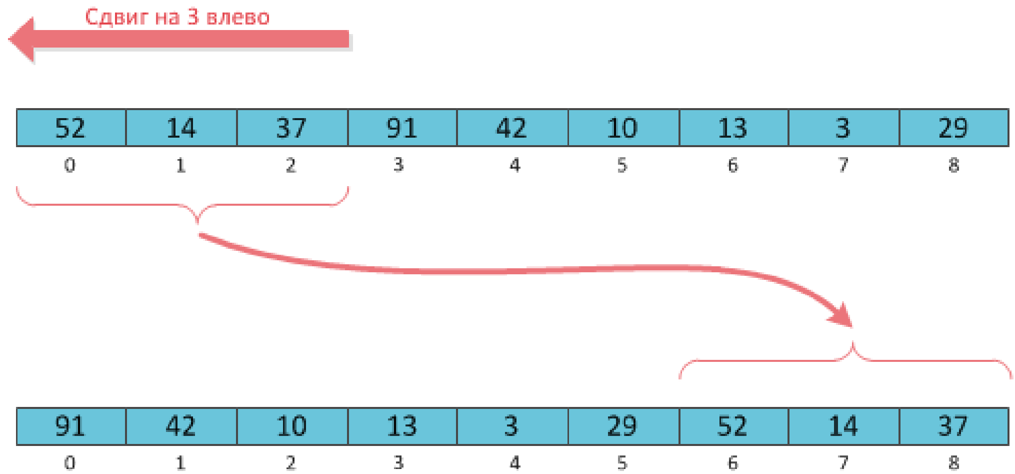
```
public bool Reversed;

public static void Push(ImplicitTreap treap)
{
    if (treap == null) return;
    // не установленное обещание - не проталкивается
    if (!treap.Reversed) return;

    var temp = treap.Left;
    treap.Left = treap.Right;
    treap.Right = temp;

    treap.Reversed = false;
```

```
    if (treap.Left != null) treap.Left.Reversed ^= true;
    if (treap.Right != null) treap.Right.Reversed ^= true;
}

public ImplicitTreap Reverse(int A, int B)
{
    ImplicitTreap l, m, r;
    this.Split(A, out l, out r);
    r.Split(B - A, out m, out r);
    m.Reversed ^= true;
    return Merge(Merge(l, m), r);
}.
```

Теперь вы можете решать классическое задание на собеседованиях принципиально новым способом :)

Циклический сдвиг массива

Фокус №6: циклический сдвиг. Суть этой операции, для тех, кто не знает, проще пояснить на рисунке.



Разумеется, циклический сдвиг всегда можно выполнить за O(N), но, реализовав массив как неявное декартово дерево, вы сможете сдвигать его за $O(\log_2 N)$. Процедура сдвига влево на **K** банальна: разрезаем дерево по индексу K, и склеиваем его в обратном порядке. Сдвиг вправо симметричен, только разрезать нужно по индексу N-K.

```
public ImplicitTreap ShiftLeft(int K)
{
```

```
    ImplicitTreap l, r;
    this.Split(K, out l, out r);
    return Merge(r, l);
}
```

Опять-таки: операция уникальная для декартовых деревьев по неявному ключу, поскольку склеивать два результата Split, будь они обычными декартовыми деревьями, недопустимо: ведь Merge ожидает от нас упорядоченные деревья, а мы скармливаем ему аргументы в неверном порядке.

Резюме

Декартово дерево по неявному ключу — простое представление массива в виде дерева, которое позволяет производить с ним и с его подмассивами кучу операций за логарифмическое время. Памяти при этом тратится все так же O(N). Конечно, использовав O-нотацию, я здесь слегка покривил душой, ведь в реальной жизни важна фактическая память, занимаемая деревом, а декартово дерево славится своим overhead`ом. Судите сами: N на информацию, N на приоритеты, 2N на ссылки на потомков, N на размеры поддеревьев — это прожиточный минимум, а если добавить еще и множественные запросы, то получим еще N на каждую операцию. Можно чуточку улучшить себе жизнь, **создавая приоритеты из информации (http://habrahabr.ru/blogs/algorithm/102006/#comment_3178416)** , однако это, во-первых, капля в море (всего лишь минус N), а во-вторых, чревато последствиями с точки зрения безопасности: если некто выяснит функцию, с помощью которой вы создаете приоритеты, то он будет потенциально способен подавать вам новые записи для создания в зловредном порядке, чтобы сильно разбалансировать декартово дерево. В конечном итоге возможна ситуация, когда все ваши данные будут заметно притормаживать — хотя такие случаи, конечно, считанные и требуют недюжинной работы. От опасности можно избавляться, используя разные простые числа P для разных вершин дерева… но это уже тема для отдельного научного исследования. Лично для меня возможности декартового дерева и простота его кода — преимущества, превосходящие проблему большого расхода памяти. Хотя, конечно же, программа программе рознь.

Из интересных фактов: в западной литературе, статьях и Интернете ни одного упоминания о декартовом дереве по неявному ключу мне найти не удалось. Конечно, никто и не знает, как оно должно называться в английской терминологии — однако вопросы на форумах и StackOverflow тоже ни к чему не привели. В русской практике спортивного программирования ACM ICPC эта структура была использована впервые в 2000 году, придуманная членом команды Kitten Computing Николаем Дуровым — многочисленным победителем международных олимпиад (впрочем, рунет больше знает его **брата Павла (http://ru.wikipedia.org/wiki/%D0%94%D1%83%D1%80%D0%BE%D0** , а также их совместное **творение (http://vkontakte.ru/)**).

Обязательную программу по декартовому дереву я на этом заканчиваю. Скорее всего, попозже будет еще как минимум одна или две части — об альтернативных реализациях операций дерева, и о функциональной его реализации, — однако уже написанные три в принципе составляют достаточный боекомплект для полноценного использования дерамиды в жизни. Спасибо всем, кто честно продирался со мной сквозь строки этого туториала :) Надеюсь, вам было интересно.

декартово дерево, **cartesian tree**, **treap**, **дерамида**, **дуча**, **структуры данных**, **двоичные деревья**, **тег который никто не читает**, **бинарные деревья**, **сбалансированные деревья**, **декартово дерево по неявному ключу**, **C**

+73	23 августа 2010, 11:15	115	Skiminok

комментарии (16)

witeb, 23 августа 2010, 11:17 –3

andreycha, 23 августа 2010, 11:57 +2

Александр, респектище вам за отличные статьи! Keep on!

Rastler, 23 августа 2010, 13:45 +1

Серия статей просто супер, тока что-то не особо много комментариев :)

maxshopen, 23 августа 2010, 14:28 +2

А всё настолько круто, что по существу то и нечего комментировать. Мне вот не понятно, кто минусы таким статьям ставит. Чтобы это делать — нужно иметь веские аргументы, но что то их не видно. Неужели на хабре кому-то неугодны такие статьи? удивительное дело

Rastler, 23 августа 2010, 14:35 **+1**

ну минусуют те, кто видимо не понял в чем суть

maxshopen, 23 августа 2010, 14:49 **0**

Ну так есть же кнопочка — воздержаться (она же посмотреть результат). Ну мимо пройти на худой конец можно, раз ничего не понял. Может промахиваются мышкой? последняя надежда на адекватный вариант ;)

Rastler, 23 августа 2010, 14:50 **0**

сам порой недоумеваю зачем так делать

wickedweasel, 23 августа 2010, 16:46 **0**

Предпоследний абзац расстроил :)
А статьи просто супер. Кстати, чем картинки рисованы?

Commaster, 23 августа 2010, 16:58 **0**

В основном — Visio

Skiminok, 23 августа 2010, 17:11 **0**

Microsoft Visio 2010.
Одна геометрическая иллюстрация в первой части создана с помощью **GeoGebra**.

wickedweasel, 24 августа 2010, 12:24 **0**

Спасибо, присмотрюсь

fuCtor, 24 августа 2010, 07:46 **0**

Упоминанием Дурова? Или тем что за бугром ничего не знают по данной теме (возможно и знают но под другим именем)?

wickedweasel, 24 августа 2010, 12:23 **0**

И тем, и другим

kotehok, 24 августа 2010, 22:36 **+1**

Да, есть ещё одна очень интересная и imho полезная, малоисследованная операция — обмен кусками между декартовыми деревьями, представляющими различные массивы. В принципе, это почти то же самое (массивы можно было бы склеить), но так удобнее понимать.

Пример задачи: реализовать операцию «обменять на отрезке все элементы на чётных позициях с элементами на нечётных».
Тогда будем хранить все чётные элементы в одном дереве, а нечётные — в другом.
Вырежем куски этого отрезка из каждого из деревьев:
A1 | A2 | A3 — нечётные элементы
B1 | B2 | B3 — чётные элементы

а теперь склеим по пути A1 — B2 — A3 и B1 — A2 — B3 — получим нужный обмен.
Очень забавная картинка :) Ну конкретно эта задача не очень осмысленна (просто олимпиадная задачка с чемпионата СПбГУ :), но, помнится, похожий приём применяли в какой-то гораздо более ценной задаче, у которой, правда, было более быстрое решение с помощью нескольких сдвинутых деревьев отрезков (мне кажется, это задача E с четвертьфинала ACM 2009, северный подрегион).
Возможно, где-то ещё ему найдётся применение :)

kotehok, 24 августа 2010, 22:40 **0**

PS. Да, буду очень благодарен, если кто-то всё-таки найдёт и выложит куда-нибудь/пришлёт мне/каким-либо иным способом доведёт до общественности информацию о всех известных в западной и вообще в научной литературе подобных структурах.

Я слышал, что есть структура данных **Rope**, которая позволяет делать склеивание за O(1), однако я ничего не знаю про разрезание и реальную эффективность использования на практике — если кто знает, поделитесь, plz.
Если кто ещё что-то такое знает, тоже пишите :)

ttim, 10 ноября 2010, 06:09 **0**

Сейчас реализовывая понял еще одно огромное преимущество дерева с неявным ключом: оно полностью immutable. Можно переделать все поля на final вычисляя size сразу (непонятно зачем рекалк нужен) и тогда сразу видно полная immutable этого дерева.

А если соответсвенно сделать лист — получится полностью immutable лист. И очень быстрый. Очень круто.