Декартово дерево: Часть 2. Ценная информация в дереве и множественные операции с ней

Оглавление (на данный момент)

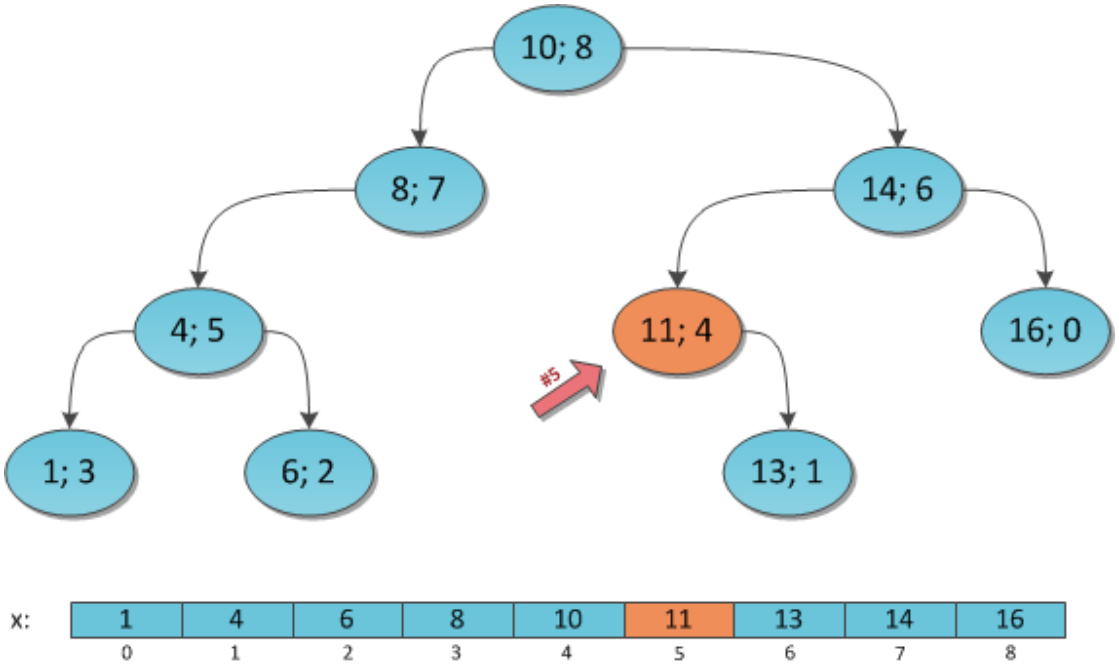
Тема сегодняшней лекции

В прошлый раз мы с вами познакомились — скажем прямо, очень обширно познакомились — с понятием декартового дерева и основным его функционалом. Только до сих мы с вами использовали его одним-единственным образом: как «квази-сбалансированное» дерево поиска. То есть пускай нам дан массив ключей, добавим к ним случайно сгенерированные приоритеты, и получим дерево, в котором каждый ключ можно искать, добавлять и удалять за логарифмическое время и минимум усилий. Звучит неплохо, но мало.

К счастью (или к сожалению?), реальная жизнь такими пустяковыми задачами не ограничивается. О чем сегодня и пойдет речь. Первый вопрос на повестке дня — это так называемая K-я порядковая статистика, или индекс в дереве, которая плавно подведет нас к хранению пользовательской информации в вершинах, и наконец — к бесчисленному множеству манипуляций, которые с этой информацией может потребоваться выполнять. Поехали.

Ищем индекс

В математике, *K-я порядковая статистика* — это случайная величина, которая соответствует K-му по величине элементу случайной выборки из вероятностного пространства. Слишком умно. Вернемся к дереву: в каждый момент времени у нас есть декартово дерево, которое с момента его начального построения могло уже значительно измениться. От нас требуется очень быстро находить в этом дереве K-й по порядку возрастания ключ — фактически, если представить наше дерево как постоянно поддерживающийся отсортированным массив, то это просто доступ к элементу под индексом K. На первый взгляд не очень понятно, как это организовать: ключей-то у нас в дереве N, и раскиданы они по структуре как попало.

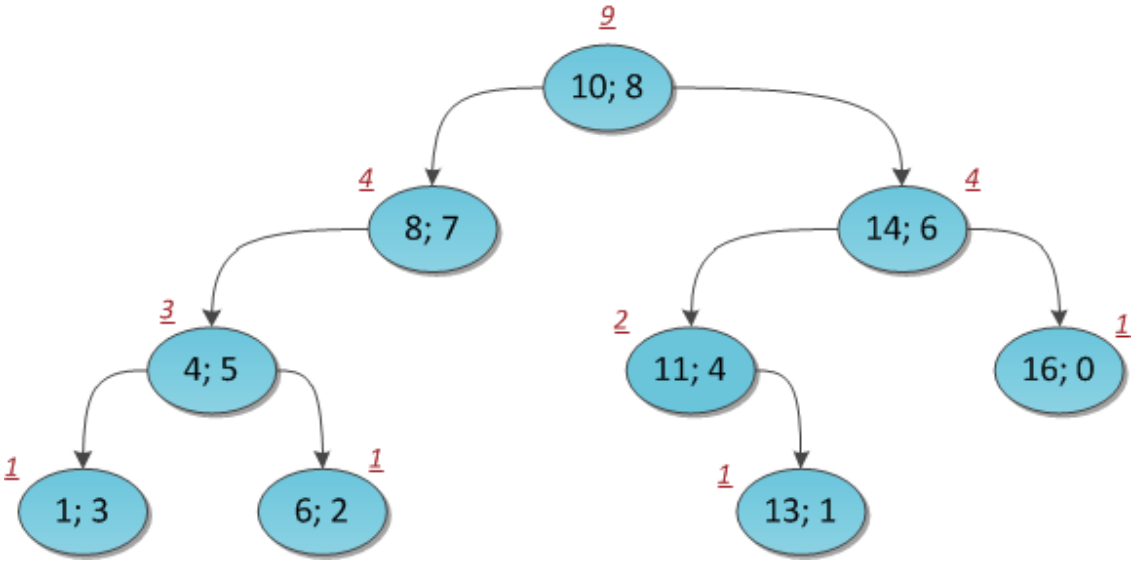


Хотя, скажем прямо, не как попало. Нам известно свойство дерева поиска — для каждой вершины её ключ больше всех ключей её левого поддерева, и меньше всех ключей правого поддерева. Стало быть, для выстраивания ключей дерева в отсортированном порядке достаточно провести по нему так называемый In-Order обход, то есть пройтись по всему дереву, выполняя принцип «сначала рекурсивно обойдем левое поддерево, потом ключ самого корня, а потом рекурсивно правое поддерево». Если каждый встреченный по пути таким образом ключ записывать в какой-нибудь список, то в конце концов мы получим полностью отсортированный список всех вершин дерева. Кто не верит, может провести указанную процедуру на дерамиде с рисунка чуть выше — получит массив с того же рисунка.

> Проведенные рассуждения позволяют написать нам для декартова дерева итератор, чтобы стандартными средствами языка (`foreach` и т.п.) пройтись по его элементам в возрастающем порядке. В языке C# для этого можно написать просто функцию in-order обхода и задействовать оператор `yield return`, а в тех, где нет подобного мощного функционала, придется пойти одним из двух путей. Либо хранить в каждой вершине ссылку на «следующий» элемент в дереве, что дает дополнительный расход ценной памяти. Либо писать нерекурсивный обход дерева через собственный стек, что несколько сложнее, но зато улучшит как скорость работы программы, так и затраты на память.

Конечно, как полноценное решение задачи расценивать такой подход не стоит, ведь он требует O(N) времени, а это недопустимо. Зато можно попытаться его улучшить, если бы мы сразу знали, в какие части дерева заходить не стоит, ибо там гарантированно большие либо меньшие по порядку элементы. А это уже вполне реально реализовать, если запомнить для каждой вершины дерева, сколько элементов нам придется обойти при рекурсивном заходе в нее. То есть будем хранить в каждой вершине её *размер поддерева*.

На рисунке показано все то же дерево с размерами поддеревьев, проставленными у каждой вершины.



Тогда предположим, что положенное количество вершин в каждом поддереве мы честно подсчитали. Найдем теперь **K**-й элемент в индексации, начинающейся с нуля (!).

Алгоритм ясен: смотрим в корень дерева и на размер его левого

поддерева $\mathbf{S_L}$, размер правого даже не понадобится.

Если $S_L = K$, то искомый элемент мы нашли, и это — корень.

Если $S_L > K$, то искомый элемент находится где-то в левом поддереве, спускаемся туда и повторяем процесс.

Если $S_L < K$, то искомый элемент находится где-то в правом поддереве. Уменьшим K на число S_L+1, чтобы корректно реагировать на размеры поддеревьев справа, и повторим процесс для правого поддерева.

Промоделирую этот поиск для K = 6 на все том же дереве:

Вершина (10; 8), $S_L = 4$, K = 6. Идем вправо, уменьшая K на 4+1=5.

Вершина (14; 6), $S_L = 2$, K = 1. Идем влево, не меняя K.

Вершина (11; 4), $S_L = 0$ (нет левого сына), K = 1. Идем вправо, уменьшая K на 0+1=1.

Вершина (13; 1), $S_L = 0$ (нет левого сына), K = 0. Ключ найден.

Ответ: ключ под индексом 6 в декартовом дереве равен 13.

Весь этот проход в принципе очень напоминает простой поиск ключа в двоичном дереве поиска — все так же просто спускаемся по дереву, сравнивая искомый параметр с параметром в текущей вершине, и в зависимости от ситуации поворачиваем влево либо вправо. Скажу наперед — мы еще не раз встретимся с подобной ситуацией в разных алгоритмах, она совершенно типична для различных бинарных деревьев. Алгоритм обхода настолько типичен, что легко шаблонизируется. Как видно, здесь даже не нужна рекурсия, мы можем обойтись простым циклом с парочкой изменяющихся с каждой итерацией переменных.

> В функциональном языке можно написать решение и через рекурсию, и оно будет выглядеть и читаться гораздо красивее, при этом не теряя ни капли в производительности: рекурсия-то хвостовая, и компилятор ее тут же соптимизирует в тот же обычный цикл. Для тех, кто знает функциональное программирование, но сомневается в моих словах — код на Haskell:

```
data Ord a => Treap a = Null
                      | Node { key::a, priority::Int, size::Int,
left::(Treap a), right::(Treap a) }

sizeOf :: (Ord a) => Treap a -> Int
```

```
sizeOf Null           = 0
sizeOf Node {size=s} = s

kthElement :: (Ord a) => (Treap a) -> Int -> Maybe a
kthElement  Null _   = Nothing
kthElement (Node key _ _ left right) k
    | sizeLeft == k = Just key
    | sizeLeft <  k = kthElement left k
    | sizeLeft >  k = kthElement right (k - sizeLeft - 1)
        where sizeLeft = sizeOf left
```

Кстати, о производительности. Время выполнения поиска K-го элемента, очевидно, $O(\log_2 N)$, мы ведь просто спустились сверху вниз до глубины дерева.

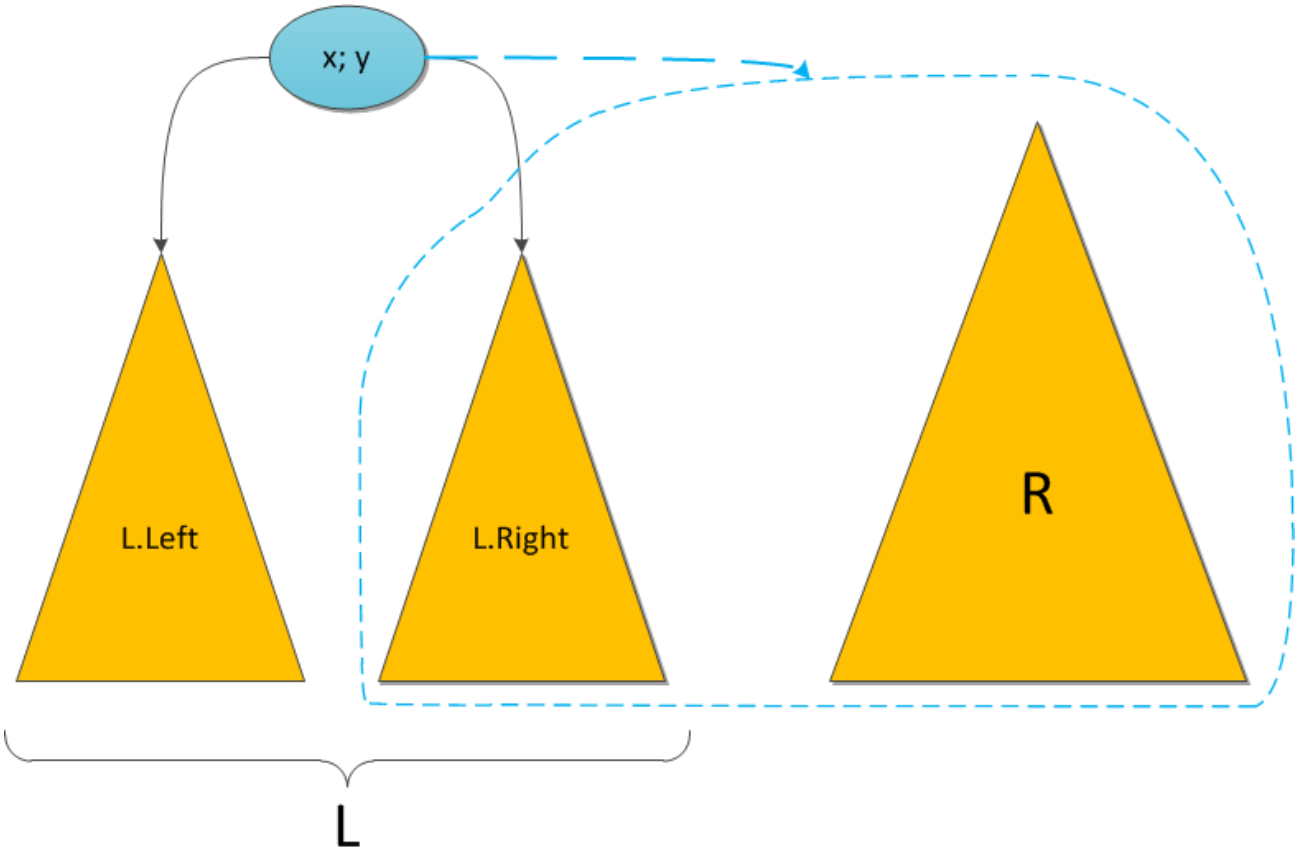
Чтобы не обижать читателей, кроме функционального приведу также и традиционный исходник на C#. В нем в стандартную заготовку класса `Treap`, приведенную в первой части, добавлено еще одно целочисленное поле `Size` — размер поддерева, а также полезную функцию `SizeOf` для его получения.

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

public int? KthElement(int K)
{
    Treap cur = this;
    while (cur != null)
    {
        int sizeLeft = SizeOf(cur.Left);

        if (sizeLeft == K)
            return cur.x;

        cur = sizeLeft > K ? cur.Left : cur.Right;
        if (sizeLeft < K)
            K -= sizeLeft + 1;
    }
    return null;
```

```
}
```

Ключевым вопросом все так же остается тот факт, что мы до сих пор не знаем, как поддерживать в дереве корректные значения `size`. Ведь после первого же добавления в дерево нового ключа все эти числа пойдут прахом, а пересчитывать их каждый раз заново — O(N)! Впрочем, нет. O(N) это заняло бы после какой-нибудь операции, которая полностью перекособочила дерево в структуру непонятной конструкции. А здесь добавление ключа, которое действует аккуратнее и не задевает все дерево, а только его маленькую часть. Значит, можно обойтись меньшей кровью.

Как вы помните, у нас с вами все делается, так сказать, через Split и Merge. Если приспособить эти две основные функции под поддержку дополнительной информации в дереве — в данном случае размеров поддеревьев — то все остальные операции автоматически станут выполняться корректно, ведь своих изменений в дерево они не вносят (за исключением создания элементарных деревьев из одной вершины, в которых `Size` нужно не забыть установить по дефолту в 1!).
Я начну с модификации операции Merge.



Вспомним процедуру выполнения Merge. Она выбирала сначала корень для нового дерева, а потом рекурсивно сливала одно из его поддеревьев с другим деревом и записывала результат на место убранного поддерева.

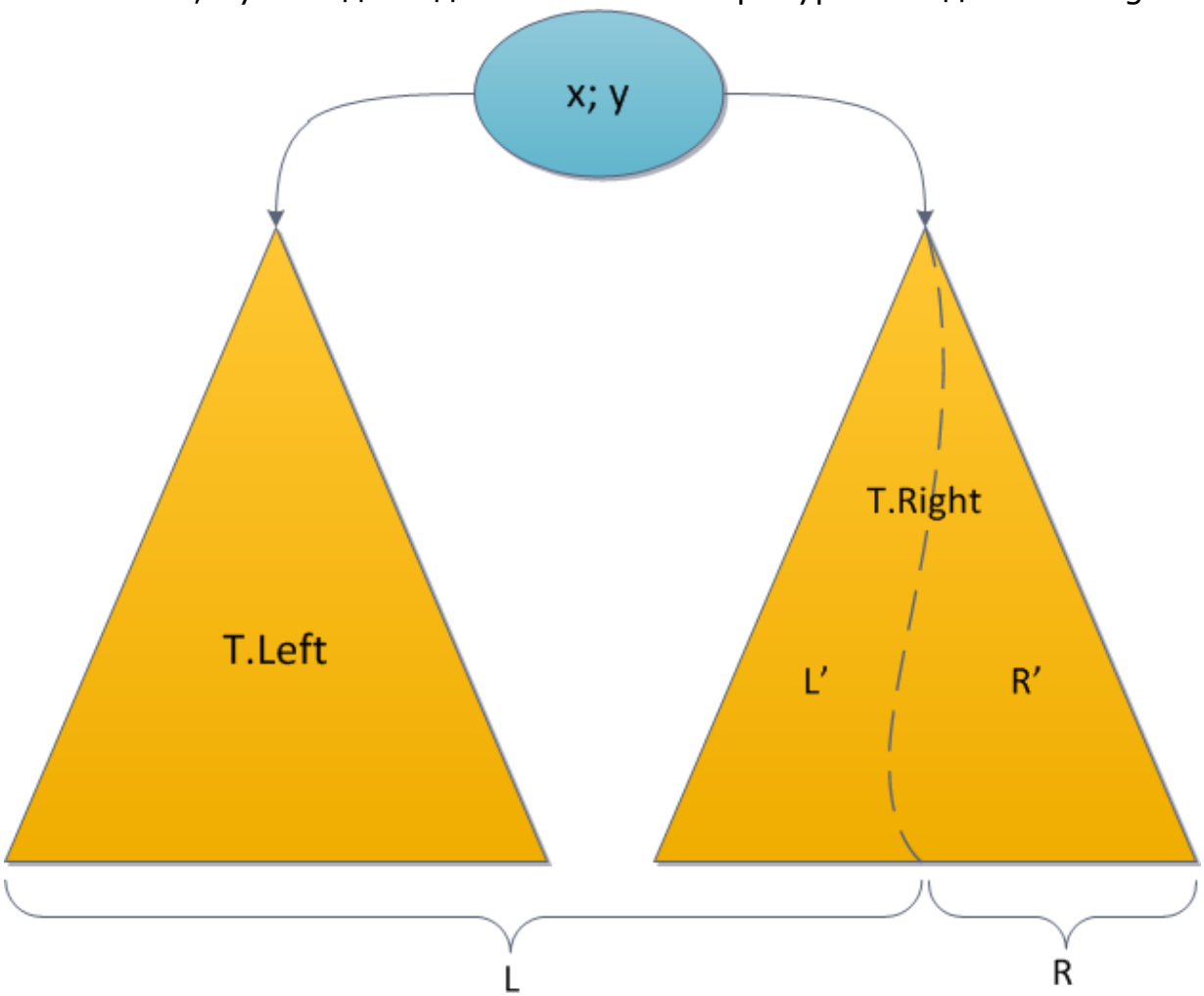
Я разберу случай, когда сливать нужно было правое поддерево, второй симметричен. Как и в прошлый раз, нам сильно поможет рекурсия.

Сделаем индукционное предположение: пускай после выполнения Merge на поддеревьях в них все уже подсчитано верно. Тогда имеем следующее положение вещей: в левом поддереве размеры подсчитаны верно, т.к. его никто не трогал; в правом тоже подсчитаны верно, т.к. это результат работы Merge. Восстановить справедливость осталось лишь в самом корне нового дерева! Ну так просто пересчитаем его перед завершением (`size = left.size + right.size + 1`), и теперь Merge полностью создала все новое дерево, в каждой вершине которого — правильный размер поддерева.

Исходный код Merge изменится незначительно: на строку пересчета размеров перед возвращением ответа. Она отмечена комментарием.

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

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

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

    answer.Recalc(); // пересчёт!
    return answer;
}
```

Точно та же ситуация нас ожидает и с операцией Split, которая тоже основа на рекурсивном вызове. Напомню: здесь в зависимости от значения ключа в корне дерева мы рекурсивно делили по указанному ключу либо левое, либо правое поддерево исходного, и один из результатов подвешивали обратно, а второй возвращали отдельно. Опять-таки, пускай для однозначности мы рекурсивно делим T.Right.



Знакомое индукционное предположение — пускай рекурсивные вызовы Split все подсчитали верно — поможет нам и в этот раз. Тогда размеры в T.Left корректны — их никто не трогал; размеры в L' корректны — это левый результат Split; размеры в R' корректны — это правый результат Split. Перед завершением нужно восстановить справедливость в корне (x; y) будущего дерева L — и ответ готов.

Исходный код нового Split, в котором две добавленные строчки пересчитывают значение `Size` в корне L либо R — в зависимости от варианта.

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

```
}

public void Split(int x, out Treap L, out Treap R)
{
    Treap newTree = null;
    if (this.x <= x)
    {
        if (Right == null)
            R = null;
        else
            Right.Split(x, out newTree, out R);
        L = new Treap(this.x, y, Left, newTree);
        L.Recalc(); // пересчёт в L!
    }
    else
    {
        if (Left == null)
            L = null;
        else
            Left.Split(x, out L, out newTree);
        R = new Treap(this.x, y, newTree, Right);
        R.Recalc(); // пересчёт в R!
    }
}
```

Обсуждаемое «восстановление справедливости» — пересчет величины в вершине — я выделил в отдельную функцию. Это базовая строчка, на которой держится функционал всего декартова дерева. И когда во второй половине статьи у нас пойдут многочисленные множественные операции, каждая со своим «восстановлением», гораздо удобнее будет его менять в одном-единственном месте, чем по всему коду класса. Такая функция обладает одним общим свойством: она предполагает, что все справедливо в левом сыне, все справедливо в правом сыне, и на основании этих данных пересчитывает параметр в корне. Таким образом можно поддержать любые дополнительные параметры, которые высчитываются из потомков, размеры поддеревьев — это лишь частный пример.

Welcome to the real world

Давайте вернемся в настоящую жизнь. В ней данные, которые

приходится хранить в дереве, не ограничиваются одним только ключом. И с этими данными постоянно приходится производить какие-то манипуляции. Я для примера в данной статье назову такое новое поле дучи `Cost`.

Итак, пусть нам на вход постоянно поступают (а порою удаляются) ключи `x`, и с каждым из них связана соответствующая цена — `Cost`. И вам нужно поддерживать во всей этой каше быстрые запросы на *максимум* цены. Можно спрашивать максимум во всей структуре, а можно только на каком-то её подотрезке: скажем, пользователя может интересовать максимальная цена за 2007 год (если ключи связаны со временем, то это можно интерпретировать как запрос максимума цены на множестве таких элементов, где **A** ≤ x < **B**).

Это не представляет никакого труда, потому что максимум тоже прекрасно подходит как кандидат на функцию «восстановления справедливости». Достаточно написать вот так:

```
public double Cost;

// Максимум
public double MaxTreeCost;
public static double CostOf(Treap treap)
{
    return treap == null ? double.NegativeInfinity :
treap.MaxTreeCost;
}

public void Recalc()
{
    MaxTreeCost = Math.Max(Cost, Math.Max(CostOf(Left),
CostOf(Right)));
}
```

Та же ситуация и с минимумом, и с суммой, и с какими-то булевыми характеристиками элемента (так называемой «окрашенностью» или «помеченностью»). Например:

```
// Сумма
public double SumTreeCost;
public static double CostOf(Treap treap)
```

```
{
    return treap == null ? 0 : treap.SumTreeCost;
}

public void Recalc()
{
    SumTreeCost = Cost + CostOf(Left) + CostOf(Right);
}
```

Или:

```
public bool Marked;

// Помеченность
public bool TreeHasMarked;
public static bool MarkedOf(Treap treap)
{
    return treap == null ? false : treap.TreeHasMarked;
}

public void Recalc()
{
    TreeHasMarked = Marked || MarkedOf(Left) || MarkedOf(Right);
}
```

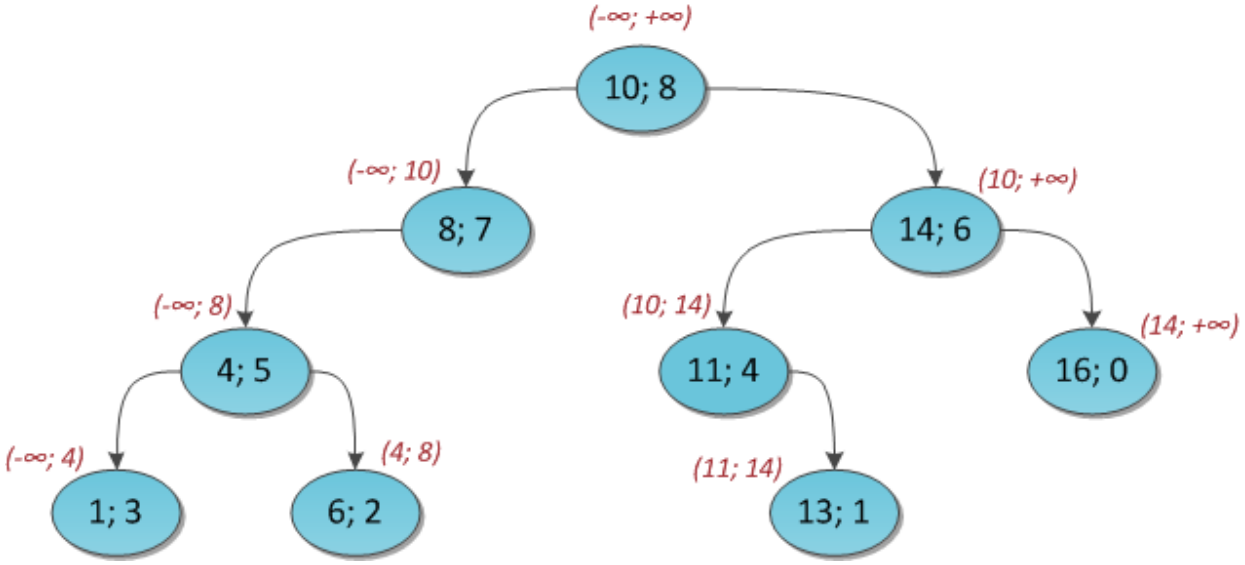
Единственное, что в таких случаях надо не забывать — инициализировать эти параметры в конструкторе при создании новых отдельных вершин.

> Как сказали бы в функциональном мире, мы с вами выполняем в какой-то мере *свертку* дерева.

Таким образом мы умеем запрашивать значение параметра для всего дерева — оно хранится в корне. Запросы на подотрезках тоже не представляют труда, если опять вспомнить свойство бинарного дерева поиска. Ключ каждой вершины больше всех ключей левого поддерева и меньше всех ключей правого поддерева. Поэтому можно виртуально считать, что с каждой вершиной ассоциирован какой-то интервал ключей, которые могут в теории встретиться в ней и в её поддереве. Так, у корня это интервал (-∞; +∞), у его левого сына (-∞; x), правого — (x; +∞), где x — значение ключа в корне. Все интервалы открытые с обоих сторон, икс среди ключей правого поддерева встретиться не

может. Если позволить в дереве одинаковые ключи и, как в первой части, заставить компаратор бросать их все в одно и то же поддерево — скажем, в левое, — то интервалом для левого сына станет (-∞; x].

Для ясности я покажу на рисунке соответствующий интервал для каждой вершины уже исследованного декартового дерева.



Теперь ясно, что параметр, хранящийся в вершине, отвечает за значение соответствующей характеристики (максимума, суммы и т.д.) для всей ключей на интервале этой вершины.

И мы можем отвечать на запросы по интервалам **[A; B)** (в C++ и ему подобных языках вообще удобнее оперировать полуоткрытыми интервалами, которые включают левый конец, и не включают правый; в дальнейшем я так и буду поступать).
Кто-то, знакомый с деревом отрезков, может подумать, что здесь стоит применить тот же рекурсивный спуск, локализируя постепенно текущую вершину до такой, которая полностью соответствует нужному интервалу или его кусочку. Но мы поступим проще и быстрее, сведя задачу к предыдущей.

Мастерски владея ножницами, разделим дерево сначала по ключу A-1. Правый результат Split, хранящий все ключи, большие либо равные A, снова разделим — на этот раз по ключу B. В середине мы получим дерево со всеми элементами, у которых ключи принадлежат искомому интервалу. Для выяснения параметра достаточно посмотреть на его значение в корне — ведь всю чёрную работу по восстановлению справедливости для каждого дерева функция Split уже сделала за нас :)

Время работы запроса, очевидно, $O(\log_2 N)$: два выполнения Split.

Исходный код для максимума:

```
public double MaxCostOn(int A, int B)
{
    Treap l, m, r;
    this.Split(A - 1, out l, out r);
    r.Split(B, out m, out r);
    return CostOf(m);
}
```

Сила отложенных вычислений

Высший пилотаж сегодняшнего дня — это возможность изменения пользовательской информации по ходу жизни дерева. Понятно, что после изменения значения `Cost` в какой-то вершине все наши прежние параметры, накопившие ответы для величин в своих поддеревьях, уже недействительны. Можно, конечно, пройтись по всему дереву и пересчитать их заново, но это снова O(N), и не лезет ни в какие ворота. Что делать?

Если речь идет о простом изменении `Cost` в одной-единственной вершине, то это не такая уж и проблема. Сначала мы, двигаясь сверху вниз, находим нужный элемент. Меняем в нем информацию. А потом, двигаясь обратно снизу вверх к корню, просто пересчитываем значения параметров стандартной функцией — ведь ни на какие другие поддеревья это изменение не повлияет, кроме как на те, которые мы посетили по пути от корня к вершине. Исходник приводить, полагаю, нет особого смысла, задача тривиальна: решайте ее хоть рекурсией, хоть двумя циклами, время работы все равно $O(\log_2 N)$.

Гораздо веселее становится жизнь, если надо поддерживать *множественные операции*. Пускай есть у нас декартово дерево, в каждой его вершине хранится пользовательская информация `Cost`. И мы хотим к каждому значению `Cost` в дереве (или поддереве — см. рассуждения об интервалах) прибавить какое-то одно и то же число A. Это пример множественной операции, в данном случае добавления константы на отрезке. И тут уже простым проходом к корню не обойдешься.

Давайте заведем в каждой вершине дополнительный параметр, назовем

его `Add`. Его суть следующая: он будет сигнализировать о том, что всему поддереву, растущему из данной вершине, *полагается* добавить константу, лежащую в `Add`. Получается такое себе запаздывающее прибавление: при необходимости изменить значения на A в некотором поддереве мы изменяем в этом поддереве только корневой `Add`, как бы давая обещание потомкам, что «когда-нибудь в будущем вам всем полагается еще дополнительное прибавление, но мы его пока выполнять фактически не будем, пока не потребуется».

Тогда запрос `Cost` из корня дерева должен совершить еще одно дополнительное действие, прибавив к `Cost` корневой `Add`, и полученную сумму расценивать как фактический `Cost`, будто бы лежащий в корне дерева. То же самое с запросами дополнительных параметров, к примеру, суммы цен в дереве: у нас есть корректно (!) посчитанное значение `SumTreeCost` в корне, которое хранит сумму всех элементов дерева, но не учтя того, что ко всем этим элементам нам полагается прибавить некий `Add`. Для получения истинно правильного значения суммы с учетом всех отложенных операций достаточно прибавить к `SumTreeCost` значение `Add`, умноженное на `Size` — количество элементов в поддереве.

Пока что не очень понятно, что делать со стандартными операциями декартова дерева — Split и Merge — и когда нам потребуется все-таки выполнить обещание и добавить потомкам обещанный им `Add`. Сейчас рассмотрю эти вопросы.

Возьмемся снова за операцию Split. Поскольку исходное дерево разделяется на два новых, и исходное мы теряем, то отложенные прибавление придется частично выполнить. А именно: пускай рекурсивный вызов Split делит у нас правое поддерево T.Right. Тогда проведем такие манипуляции:

• Выполним обещание в корне, прибавим к корневому `Cost` значение корневого `Add`.

```
T.Cost += T.Add;
```

• «Спустим» обещание влево: всему левому поддереву тоже полагается `Add`. Но поскольку рекурсия влево не идет, то активно трогать это поддерево нам не нужно. Просто запишем обещание.

```
T.Left.Add += T.Add;
```

• «Спустим» обещание вправо: всему правому поддереву тоже

полагается `Add`. Это нужно сделать до рекурсивного вызова, чтобы операция Split манипулировала с корректным декартовым деревом. Дальнейшее обновление рекурсия сделает сама.

```
T.Right.Add += T.Add;
```

• Сделаем положенный рекурсивный вызов. Split вернула нам два корректных декартовых дерева.

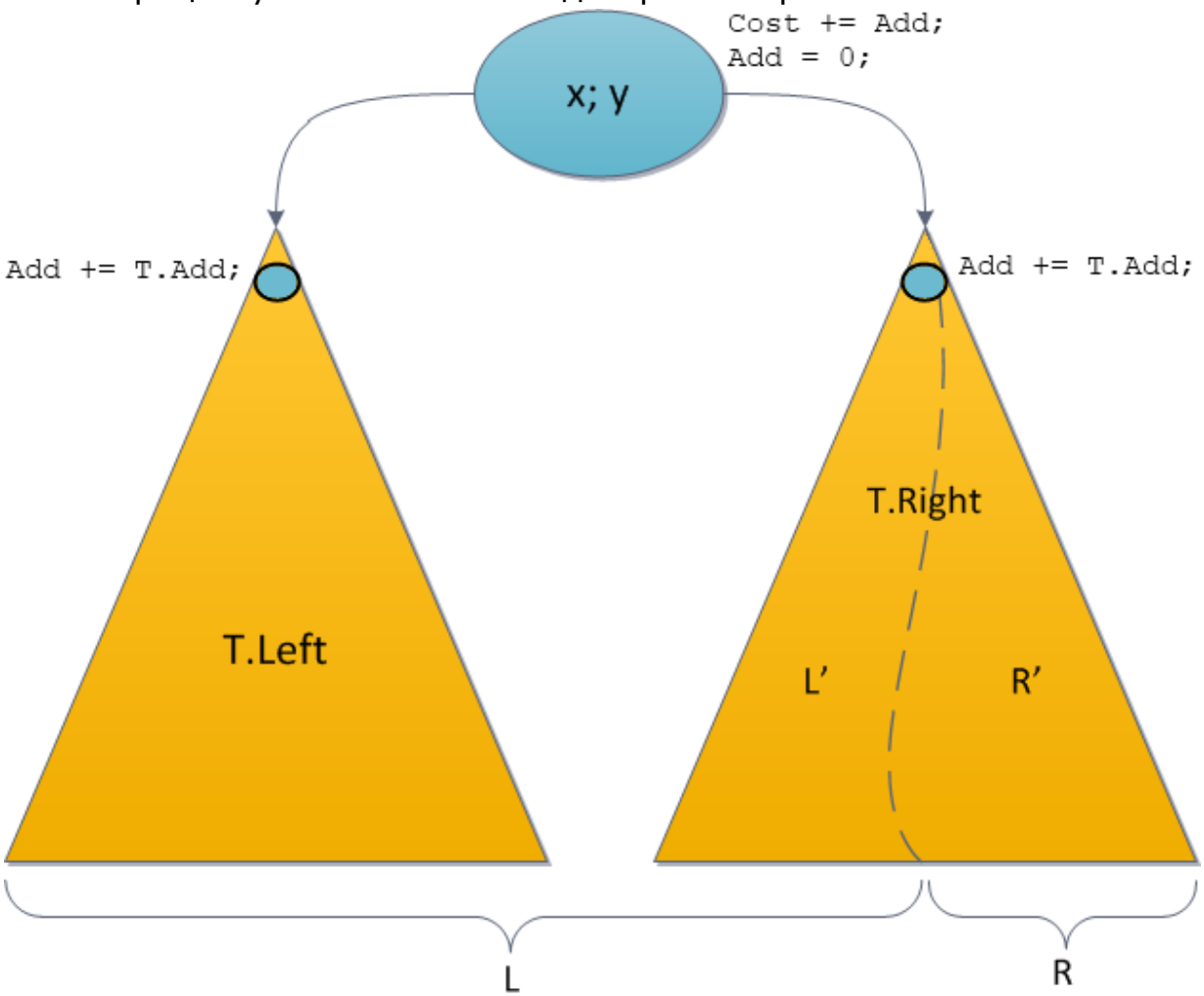
• Поскольку обещание в корне мы честно выполнили, а обещания для потомков честно записали в памяти, то корневой `Add` стоит обнулить.

```
T.Add = 0;
```

• Дальнейшие прикрепления поддеревьев в Split выполняем как обычно.

В итоге — два корректных декартовых дерева с актуальной информацией по обещаниям: где-то исполненным, где-то лишь отложенным, но актуальным.

Все операции указаны на новой диаграмме Split.



Заметим, что *фактические* обновления проводятся лишь в тех вершинах, по которым будет рекурсивно идти операция Split, для остальных же мы в лучшем случае спустим обещание чуть ниже. Та же ситуация будет и с Merge.

Новый Merge в принципе поступает аналогично. Пускай ему нужно

рекурсивно объединить правое поддерево L.Right с правым входным деревом R. Тогда выполняем следующее:

• Выполним обещание в корне L.

```
L.Cost += L.Add;
```

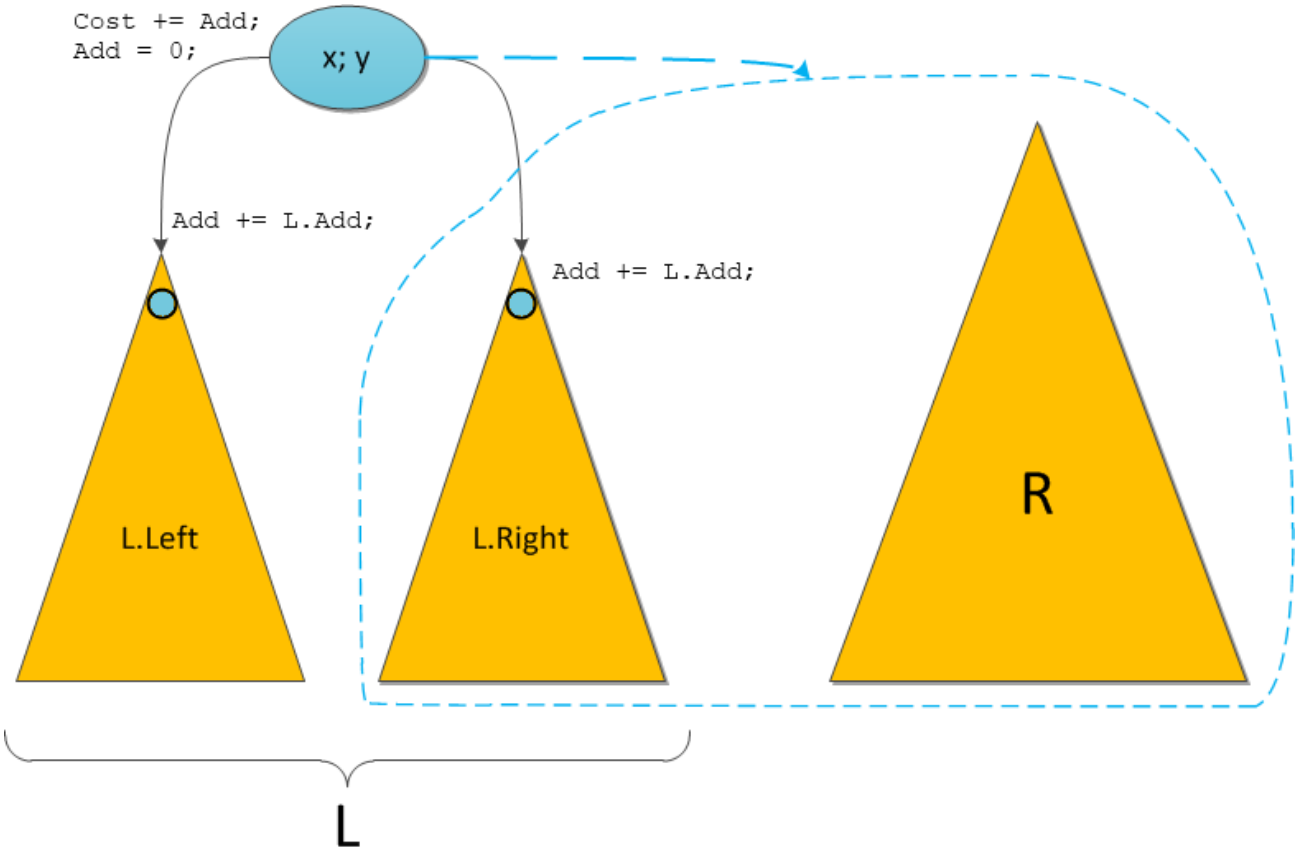
• «Спустим» обещание потомкам L — на будущее.

```
L.Left.Add += L.Add;
L.Right.Add += L.Add;
```

• Обещание в корне выполнено — можно про него забыть.

```
L.Add = 0;
```

• Делаем нужный рекурсивный вызов Merge(L.Right, R), ведь теперь оба её аргумента — корректные декартовы деревья. И вернет она нам тоже корректное дерево.
• Подвешиваем возвращенное дерево правым сыном, как и раньше. В итоге — снова декартово дерево с актуальной информацией по обещаниям.



Теперь, когда мы умеем делать запросы в корне и менять на всем дереве, не представляет труда промасштабировать это решение для подотрезков, просто применив тот же принцип, что и несколькими абзацами выше.
Сделаем два вызова Split, выделив поддерево, соответствующее нужному интервалу ключей, в отдельное дерево.

Увеличим у этого дерева корневое обещание `Add` на данное A. Получим корректное дерево с отложенным прибавлением.
Сольем снова все три дерева вместе двумя вызовами `Merge` и запишем на месте исходного. Теперь ключи на заданном интервале честно живут с обещанием, что всем им когда-нибудь в будущем полагается добавить A. Исходный код данной манипуляции можно оставить как упражнение читателю :)

> Не забывайте после всех операций восстанавливать справедливость в корнях вызовом нового варианта функции `Recalc`. Этот новый вариант должен учитывать отложенные прибавления, как уже было описано при рассказе о запросах.

Напоследок замечу, что множественные операции, конечно же, не ограничиваются одним только прибавлением на отрезке. Можно также на отрезке «красить» — устанавливать всем элементам булевый параметр, изменять — устанавливать все значения `Cost` на отрезке в одно значение, и так далее, что только придумает фантазия программиста. Главное условие на операцию — чтобы ее можно было за O(1) протолкнуть вниз от корня к потомкам, передав отложенное обещание чуть ниже по дереву, как мы и поступали с Merge и Split. Ну и, конечно же, информация должна легко восстанавливаться из обещания во время запроса, иначе нет смысла и огород городить.

Резюме

Мы с вами научились поддерживать в декартовом дереве различную пользовательскую дополнительную информацию, и проводить с ней огромное количество множественных манипуляций, при этом каждый запрос или изменение, будь то в единственной вершине либо на отрезке, выполняется за логарифмическое время. Вкупе с элементарными свойствами декартового дерева, рассказанными в первой части, это уже представляет огромный простор для использования его на практике в любой системе, где требуется хранить значительные объемы данных, и периодически запрашивать у этих данных какие-нибудь статистики.

Но в следующей части я сделаю декартовому дереву маленькую модификацию, которая превратит его в инструмент поистине колоссальной мощи для повседневных нужд. Именоваться это чудо будет

«декартово дерево по неявному ключу».

Всем спасибо за внимание.

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

+68 18 августа 2010, 20:11 132 Skiminok

комментарии (14)

Lendis, 18 августа 2010, 21:32 +4

ха, а я прочитал (я про тег, а не про статью :)).

Skiminok, 18 августа 2010, 21:33 +1

Ух ты. За трое суток первый человек заметил пасхалку.

Lendis, 18 августа 2010, 21:41 0

в статьях такого рода первым делом нужно читать теги), а то после уже не до них будет. А за статью спасибо, вкурил с трудом, но интересно.

IDMan, 20 августа 2010, 23:31 0

В первой части тоже такой был.

Krovosos, 18 августа 2010, 21:40 0

Прикольная штука. Кто их придумал?

Skiminok, 18 августа 2010, 21:47 0

В сегодняшнем полном варианте — Зайдель и Арагон, в 1996 году (**link**). Идеи о том, как «квази-балансировать» дерево поиска приоритетами существовали и раньше, применения отложенных вычислений на отрезках — еще раньше, с успехом работали в одноименном «дереве отрезков».
После 1996 года главная модификация, которая произошла — переделка под неявный ключ. Что характерно, вроде бы сделана в России ACM-олимпийцами, потому что в западной литературе упоминаний об этой махинации — ноль =)

Voviandr, 18 августа 2010, 22:37 0

благодарю автора, чудесная статья. обожаю изучать новые алгоритмы и структуры данных, буду ждать продолжения.

Krovosos, 18 августа 2010, 22:39 **+1**

Понятно.

Напомнило как я однажды «изобрел» Б-деревья. Как-то я размышлял над таблицами и списками и решил, что их нужно творчески объединить, чтобы избавиться от недостатков (долгая вставка в таблицу и долгий поиск в списке). Полученный «список таблиц», разумеется, постепенно превратился в дерево, состоящее из «списков таблиц». Долго я сражался с реализацией и получил-таки работающий вариант. Радости моей не было предела и после оптимизаций я решился помериться производительностью с деревьями из stl библиотеки, на которых основан map.

Увы, в среднем моя структура отставала на 50-70%. Хотя временами, для отдельных случаев — опережала. Тогда-то я и решил, наконец, поизучать вопрос поподробнее и узнал про красно-черные и про Б-деревья… :-)

> **Krovosos**, 18 августа 2010, 22:40 **0**
>
> Собственно, декартовы деревья поражают простотой кода модификации, вот я к чему :-)

MikhailEdoshin, 19 августа 2010, 07:34 **0**

Слушайте, замечательная структура эти декартовы деревья. Спасибо большое :) — хотя придётся еще раз (и не раз, чувствую) перечитать обе статьи. Заинтригован упоминаниями о третьей части.

grep0, 19 августа 2010, 10:15 **+1**

Вообще-то, для классических сбалансированных деревьев, например, AVL, тоже не представляет никакой трудности держать в узле вычисленные параметры поддерева — тот же размер левого потомка или max(cost). И преобразование этих величин при элементарных преобразованиях (для AVL это вставка/удаление листьев и поворот) ничуть не сложнее.

AlMag, 22 августа 2010, 13:07 **+1**

Небольшим недостатком декартового дерева есть то, что игреки нигде не используются практически, кроме балансировки дерева.
На самом деле, можно вместо Y брать реальную информацию, тоесть не случайно сгенерированные числа. Если после этого умножить все Y на некоторое простое число P и взять по модулю 2^64 (что очень удобно и быстро), то можно доказать, что Y будут распределены случайным образом. Для того, что бы при необходимости узнать Y, нужно будет просто домножить на обратный элемент к P по модулю 2^64.
Таким образом, память у декартового дерева не будет тратиться попусту и в ней можно хранить некоторую информацию, помимо ключей.

kotehok, 24 августа 2010, 22:26 **+1**

Да, этот метод действительно может быть практически полезен.
Однако у него есть одна серьезная уязвимость, а именно, если кто-то сможет каким-то образом выяснить, какое число вы используете в качестве простого, то он сможет ваше дерево сделать очень некрасивым (с серьезными последствиями для работы вашей программы).

Мне рассказывали, что недавно к нам приезжал Тарьян, и рассказывал грустную историю про людей, которые использовали подобные методы в красно-черном дереве, бездоказательно, и у них залажался какой-то серьезный проект, и их за это засудили и отсудили кучу денег…

Поэтому реально гораздо более сильны следующие методы:
1) использование счётчиков для балансировки
— если у вас в дереве всё равно хранятся счётчики, то можно, используя их, предсказать вероятность, с которой новый «y» будет больше текущего. Этим методом любила пользоваться ACM-команда SPb SU Drink Less, пока на очередном чемпионате СПбГУ им не попалась задачка, на которой они не смогли пробить TL с помощью такой балансировки. Действительно, если аккуратно написать формулу, то видно, что там нужно так или иначе взятие по модулю, которое не самая быстрая операция. Конечно, есть способы с этим бороться, но домножение на обратное было быстрее.
2) апгрейд исходного метода, придуманный мной совсем недавно, выглядит так:
пусть у нас в дереве нет никакой дополнительной информации, кроме x
тогда, казалось бы, на простое (по модулю) придётся домножать x
но легко доказать, что какая бы функция f(x) ни была, найдётся такая последовательность x длины не менее sqrt (M), где M — модуль, по которому происходит домножение на простое, что пары (x, f(x)) будут совместно монотонны — таким образом, используя x в качестве прототипа для y, мы никак не сможем добиться хорошей формы дерева
однако есть ещё одно непредусмотренное место, в которое легко засунуть рандом — это, собственно, указатель на структурку, в которой и хранится узел дерева
тут открывается простор для фантазии, как выделять структурки, чтобы указатель на них был случаен :) так или иначе с этой задачей можно справиться, и никакой y не нужен
ну или можно по первому методу использовать p в качестве дополнительных данных, если не боитесь злобных хакеров ;) или линейную комбинацию из x и p со случайными коэффициентами, так вернее :)

IDMan, 28 августа 2010, 00:16 **0**

Вы не пробовали писать книгу? У вас хороший стиль, как для технических текстов — не сухо, и в то же время не художественный рассказ. Картинки очень понравились. Даже если что не понятно в тексте, то после них вопросов не остается :).