Декартово дърво: Част 1. Описание, операции, приложения

**Декартово дърво** (cartesian tree, treap) — красива и лесна за реализация структура от данни, която с минимални усилия ще ви позволи да реализирате много скоростни операции над масивите с вашите данни.

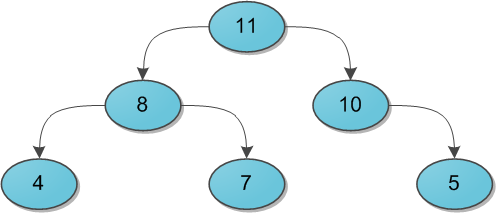
Ще се постарая да разкажа всичко, което ми е известно по темата — независимо от това, че то не е толкова много, все пак ще се наложи да го разделя на две, че и на три части. Всички алгоритми са илюстрирани на C# (а тъй като съм любител на функционалното програмиране, то на места в послесловията става дума и за F# — но това не е нужно да се чене :). И така, да започнем.

Въведение

Като въведение препоръчвам да прочетете [пост за двоични дървета за пре](http://habrahabr.ru/post/65617/)търсване на същия автор [winger](http://habrahabr.ru/users/winger/), тъй като без да разбирате, какво е дърво, дърво за претърсване, а също и без познаване на [оценка на сложност на алгоритъма](http://habrahabr.ru/post/78728/) голяма част от материала в дадената статия ще остане за вас като китайска грамота. Обидно, нали?

Следващата точка от нашата задължителна програма е **пирамида** (heap). Мисля даже, че това също е много известна структура от данни, но за всеки случай ще направя кратък обзор .

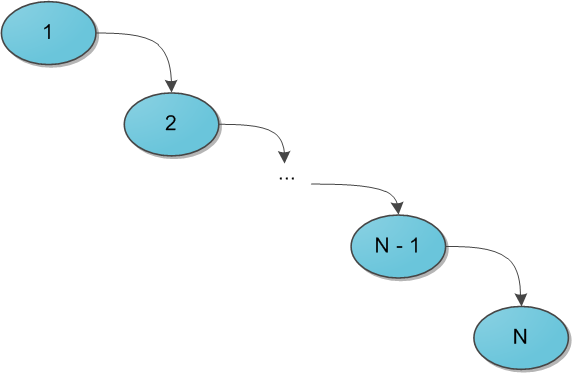
Представете си двоично дърво с някакви данни (ключове) във върховете. И за всеки връх изискваме следното: неговият ключ е строго по-голям, от ключовете на непосредствените му наследници. Ето неголям пример за конкретна пирамида:



Ще отбележа, че в никой случай не е задължително да приемаме пирамидата като структура, в която родителят е *по-голям*, от наследниците си. Никой не ви забранява да разгледате противоположния вариант и да приемете, че родителят е *по-малък от* наследниците си — важно е да е едно и също за цялото дърво. за целите на тази статия е по-удобно да се използва вариантът със знак «по-голямо».

За сега зад кадър остава вопросът, как се добавят и изтриват елементи от пирамидата. Първо, тези алгорити изискват отделно разглеждане, и второ, те все пак няма да ни потрябват.

Проблеми

Когато става дума за дърво за претърсване (вече прочетохте препоръчаната статия, нали?), основния въпрос, който стои пред структурата е скоростта на изпълнение на операциите, независимо от данните, които се съхраняват в нея, и последователността на тяхното постъпване. Така, двоичното дърво за претърсване дава гаранция, че търсенето на даден ключ в това дърво ще се изпълнява за O(H), където **H** е височината на дървото. Но каква може да бъде тази височина — един дявол знае. При неблагоприятни обстоятелства височината на дървото лесноо може да стане **N** (броя на елементите в него), и тогава дървото за претърсване се изражда в обикновен списък, което обезмисля всичко. За достигане до такава ситуация е достатъчно да елементите да се добавят в него в нарастващ ред – числата от 1 до N например, стандартния алгоритъм за добавяне в дървото ще се получи следната картинка:  


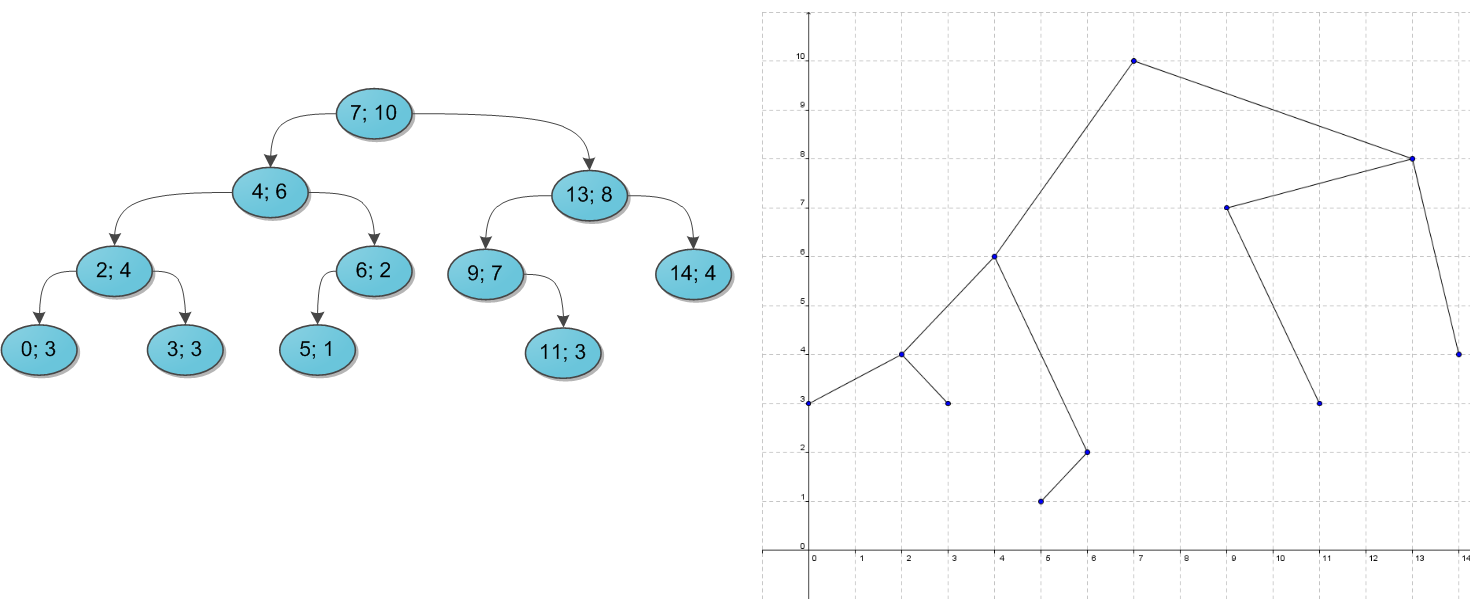
Съществуват огромно количество тъй наречени балансирани дървета за претърсване — грубо казане, такива, в които по време а съществуването на дървото, при всяка операция над него се поддържа оптималност на максималната дълбочина на дървото. Оптималната дълбочина е от порядъка на O(log2 N) — тогава това е скоростта на изпълнение на всяка операция в дървото. Структури данни, поддържащи такава дълбочина, има много, като най-известните са червено-черно дърво или AВЛ-дърво. Тяхна отличителна черта в общия случай е трудната реализация, базирана на размера на огромен брой случаеи, в които можеш да сгрешиш. Декартовото дърво се отличава със своята простота и красота, дори още повече, то предоставя по определен начин същото логаритмично време, но с достаточно висока вероятност… впрочем, за такива детайли и тънкости – по-късно.

Определение

И така имаме данните на дървото — ключ **x** (тук и по-нататък се предполага, че ключът е и самата информация, която съхраняваме в дървото; когато впоследствие се наложи да се отделя по смисъл потребителската информация от ключа, ще го обясня подробно). Да добавим към ключа още един параметър — **y**, и да го наречем *приоритет*. Сега ще построим едно такова вълшебно дърво, което съхранява във всеки връх по два параметъра, и при това *по ключовете то е дърво за търсене, а по приоритетите — пирамида*. Такова дърво за напред ще наричаме декартово.

Ще отбележим, че в англоезичната литература е особено популярно наименованието *treap*, което нагледно показва същността на структурата: tree + heap. В рускоезичната може да се срещне наименование, съставено на същия принцип: *дерамида* (дерево + пирамида) или *дуча* (дерево + куча).

Защо тези дървета се наричат декартови ще стане ясно, веднага след като се опитаме да го нарисуваме. Да вземем произволен набор двойки «ключ-приоритет» и да разположим в координатната мрежа съответстващите им точки (x, y). Нека след това свържем съответстващите върхове с линии, образувайки дърво. По този начин, декартовото дърво отлично се разполага в равнината благодарение на своите ограничения, а двата негови основни параметъра — ключ и приоритет — в известен смисъл се явяват координати. Резултатът от построението е показан на рисунката: вляво в стандартната нотация на дърво, в дясно — на декартова равнина.



За сега все още не е съвсем ясно, за какво ни е това. А разгадаването е просто, и се базира на следното твърдение. Първо, нека са дадени множество ключове: коректни дървета за търсене от тях могат да се построят по много различни начини, в това число и списъкоподобни. От тях след добавянето на приоритетите от дадените ключове може да се построи вече само едно-единствено дърво, независимо от реда на постъпване на ключовете. Това е доста очевидно.

Второ, нека сега да направим нашите приоритети случайни. Т. е. просто да свържем с всеки ключ случайно число от достатъчно голям диапазон, и именно то и ще служи за съответстващия y. Тогава полученото декартово дърво с много висока, стремяща се към 100% вероятност, ще има височена, не надминаваща 4 log2 N. (Ще оставим този факт тук без доказателство.) Това означава, че макар и то да не е идеално балансирано, времето за търсене на ключа в такова дърво все пак ще бъде от порядъка на O(log2 N), към което ние всъщност се стремяхме.

Още един интересен подход – няма да правим приоритетите случайни, а ще си спомним за това, че имаме огромно количество някаква дополнителна потребителска информация, която, като правило, се налага да се съхранява във върховете на дървото. Ако има основание да се смята, че тази информация по своята същност е достатъчно случайна (рожденния ден на потребителя, например), то може да пробваме да я използваме с користни цели. Да вземем в качеството на приоритет или непосредствено информацията, или резултатът от някаква нейна функция (само, че тогава функцията трябва да бъде обратима, за да се възстанавява при необходимост информацията от приоритетите). Впрочем, тук се налага да се действа на свой страх и риск — ако дървото след известно време силно се разбалансира и цялата програма започне чувствително да бави, ще се наложи спешно да мъдрим нещо за спасение от ситуацията.

По нататък, за простота на изложението ще предположим, че всички ключове и всички приоритети в дървото са различни. В същност возможността за равенство на ключовете не създава никакви особени проблеми, вие просто трябва точно да определите, къде ще слагате елемените, равни на дадено x — или *само* в лявото, или *само*  в дясното му поддърво. Равенството на приоритетите по идея също не е особен проблем, освен замърсяване на доказателството и разсъжденията с частни случаи, но на практика е по-добре да го избягваме. Случайната генерация на цели приоритети напълно подхожда в повечето случаи, а реалните числа между 0 и 1 — почти във всички случаи.

Преди началото на разказа за операциите ще приведа заготовка на клас на C#, който ще реализира нашето декартово дърво.

public class Treap

{

public int x;

public int y;

public Treap Left;

public Treap Right;

private Treap(int x, int y, Treap left = null, Treap right = null)

{

this.x = x;

this.y = y;

this.Left = left;

this.Right = right;

}

// здесь будут операции...

}

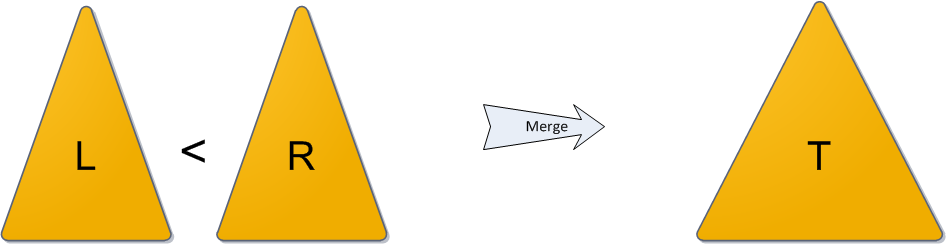
За простота на изложението ще използвам x и y от тип int, но е ясно, че на тяхно място може да бъде произволен тип, елементите на който могат да се сравняват — т.е. произволна, реализация на IComparable или IComparable<T> в терминах C#. В С++ това е произволен тип, за който са предефинирани операциите за сравнение - <, >, ==.

Магията на лепилото и ножницата

Да отговорим на насъщния въпрос — как се работи с декартови дървета. Въпросът, как въобще да го построим от безразборния набор от ключове, ще отложим за малко, а за сега да предположим, че някоя добра душа вече ни е построила дървото, и сега ние само ще го променяме.

Цялата работа с декартови дървета се свежда до две основни операции: **Merge** и **Split**. С тяхна помощ, елементарно се представят всички останали операзции.

Операцията **Merge** приема на вход две декартови дървета **L** и **R**. Тя трябва да ги слее в едно, също декартово дърво **T**. Да отбележим, че да направим операция **Merge** може не с кои да е две дървета, а само с тези, за които всички ключове от лявото дърво ( L ) не превишават ключовете от дясното ( R ).

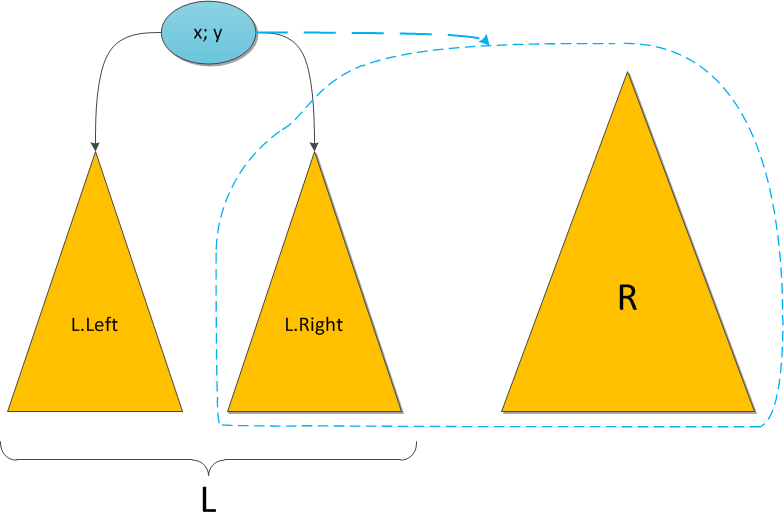


Алгоритъмът на работа на **Merge** е много прост. Кой элемент ще стане корен на бъдещото дърво? Очевидно, този с най-голям приоритет. Кандидатите за елементи с максимален приоритет са два — само корените на двете изходни дървета. Да сравним техните приоритети; нека, за еднозначност приоритетът на левия корен е по-голям, а ключът му е равен на x. Новият корен е определен, но сега трябва да помислим, кои елементи са в неговото дясно поддърво, и кои — в лявото.

Лесно е да се разбере, че цялото дърво R ще се окаже в дясното поддърво на новия корен, защото неговите ключове са по-големи от x по условие. По същия начин, лявото поддърво на стария корен L.Left съдържа всички ключове, по-малки от x, и трябва да останат в лявото поддърво, а дясното поддърво L.Right… а тук дясното трябва по същите съображения да се окаже отдясно, обаче отново е неясно, къде да поставем неговите елементи и къде елементите на дървото R.

Стоп, защо да не е ясно? Имаме две дървета, ключовете наедното са по-малки от ключовете в другото, и трябва някак да ги обединим и получения резултат да прикачим към новия корен като дясно поддърво. Просто рекурсивно извикваме **Merge** за L.Right и дървото R, и върнатото от нея дърво използваме като ново дясно поддърво. Резултатът е налице.

На рисунката със син цвят е показано дясното поддърво, което се получава от операцията Merge и връзката от новия корен към това дърво.

  
Симметричния случай — когато приоритетът в корена на дървото R е по-голям — се обработва аналогично. И, разбира се, не трябва да забравяме граничния случай, който в нашия случай е, ако някое от дърветата L и R, или и двете се окажат празни.

Исходный код Merge:

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

{

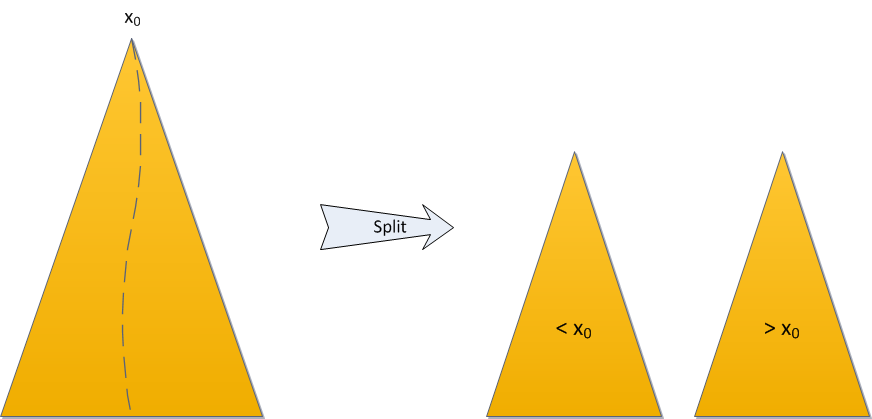
var newL = Merge(L, R.Left);

return new Treap(R.x, R.y, newL, R.Right);

}

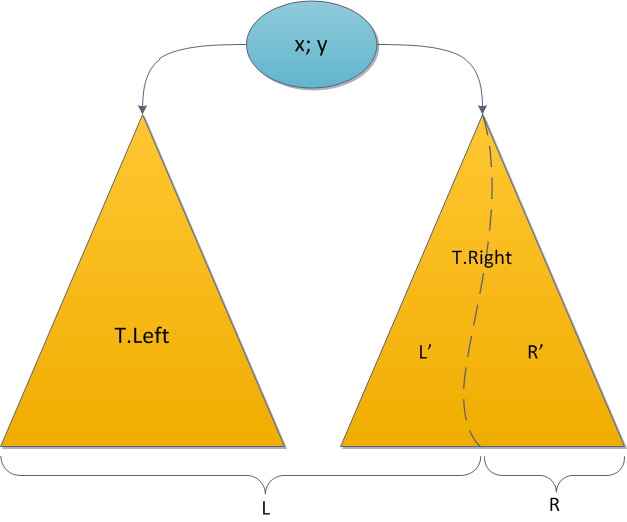
}

Сега за операцията Split. Тя приема като параметри коректно декартово дърво T и някакъв ключ **x0**. Задачата на тази операция е да раздели дървото на две така, че в едното от тях ( L ) да се окажат всички елементи от изходното дърво с ключове, по-малки от x0, а в другото ( R ) — с по-големи. Няма особени ограничения за дървото.

  
  
Ще разсъждаваме по подобен начин. Къде ще отиде коренът на дървото T? Ако неговият ключ е по-малък от x0, то в L, иначе в R. Нека отново, за определеност, да предположим, че ключът на корена е по-малък от x0.

Тогава може веднага да се каже, че всички елементи на лявото поддърво на T ще се окажат в L — тъй като техните ключове ще бъдат пао-малки от x0. Освен това, коренът T ще бъде и корен на L, тъй като неговият приоритет е най-големия в цялото дърво. Лявото поддърво на корена остава без промяна, но дясното ще се намали — от него ще трябва да се изключат елементите с ключове, по-големи от x0, и да се преместят в дървото R. А остатъкът от ключовете да се съхрани като ново дясно поддърво на L. Отново виждаме идентичната задача и отново се налага използване на рекурсия!

Да вземем дясното поддърво и рекурсивно да го разрежем по същия ключ x0 на две дървета L' и R'. След това става ясно, че L' ще стане новото дясно поддърво на дървото L, а R' и е всъщност дървото R — то се състои от тези и само от тези елементи, които са по-големи от x0.



Симетричния случай, при който ключът на корена е по-голям от x0, също е съвършенно идентичен. Основата на рекурсията тук е случаят, в който някое от поддърветата се окаже празно. Ето и изходния код на функцията:

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);

}

else

{

if (Left == null)

L = null;

else

Left.Split(x, out L, out newTree);

R = new Treap(this.x, y, newTree, Right);

}

}

Обърнете внимание: дърветата, върнати на изхода на операцията Split, са подховящи за входни данни за операцията Merge: всички ключове на лявото дърво не надвишават ключовете в дясното. Това ценно обстоятелство ще ни послужи след няколко абзаца.

Последен въпрос — това е времето за работа на Merge и Split. От описанието на алгоритъма става ясно, че Merge за свяка итерация на рекурсията намалява общата височина на двете сливащи се дървета минимум с единица, така че общото време за работа не надвишава 2H, т. е. O(H). А със Split всичко е съвсем просто — работим с единствено дърво, чиято височина се умалява при всяка итерация също най-малко с единица, и тогава сложността на работа на операцията е от порядъка на O(H). А тъй като декартовото дърво е със случайни приоритети, както вече стана ясно, с голяма вероятност има близка до логаритмична височина, то Merge и Split работят за желаното O(log2 N), и това ни дава потресаващ простор за тяхното приложение.

Операции с деревом

Сега, когато до съвършенство владеем лепилото и ножницата, не представлява никакъв труд само с помощта им да реализираме най-необходимите действия с декартови дървета: добавяне на елемент в дърво и изтриване на елемент от дърво. Ще покажем най-простия вариант за тяхната реализация, основани *изцяло* на Merge и Split. Той ще работи за логаритмично врем, но все пак ще има разлика, както казват ACM-олимпийците, с голяма константа: т. е. порядъка на зависимостта на времето на работа от размера на дървото ще бъде все така O(log2 N), но *точното* време за работа се различава няколко пъти. Примерно, 4 log2 N срещу просто log2 N. На практика това различие почти не се усеща, докато размера на дървото не достигне наистина галактически размери.

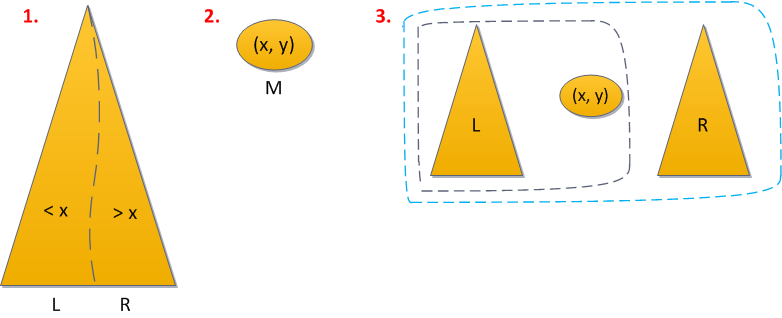
Съществуват и оптимални реализации както на добавяне и изтриване, така и други нужни операции в дерамидите, чиято константа е значително малка. Тези реализации са разгледани в една от следващите части от този цикъл лекции. Там става дума и за подводните камъни, свързани с използването на по-бързия вариант. Подводните камъни на първо място са свързани с необходимостта от поддръжка на дополнителни заявки към дървото и съхраняването в него на особена информация… впрочем, сега не е време да ви занимаваме с това, до множествените операции с дървото (извънредно важни функции!) има още време.

И така, нека е дадено декартово дърво и някакъв елемент **x**, кото трябва да се включи в дървото (да припомним, че в контекста на тази статия се предполага, че всички елементи са различн и **x** още го няма в дървото). В първия момент идеята за вмъкване е стандартната: спускаме са по дървото по ключовете, избилайки всеки път да вървим наляво или надясно, докато не намерим мястото, където може да вмъкнем нашият x, и да го допишем. Но това решение е неправилно, тъй като забравихме за приоритетите. Мястото, където алгоритъма на дървото за претарсване ще реши да добави новия връх, еднозначно удовлетворява ограничението на дървото за търсене по x, но може да наруши ограничението за пирамида по y.

Втория вариант за решение е следния: представяме новия ключ като дърво с единствен връх (със случаен приоритет y), след което го сливаме с изходното с помощта на **Merge**. Това отново не е вярно: в изходното дърво могат да бъдат върхове с ключове, по-големи от x, и тогава ще нарушим обещанието, дадено на функцията **Merge** за взаимоотношенията между нейните входни дървета.

Проблемът може лесно да се реши. Универсалността на операциите Split/Merge, позволява да реализираме елегантно решение на този проблем.

1. Разделяме (split) дървото по ключа x на дърво L, с ключове по-малки от **x**, и дърво R, с ключове по-големи от **x**.
2. От дадения ключ създаваме дърво M от единствен връх (x, y), където y е току що генериран случаен приоритет.
3. Объединяваме (merge) по ред L с M, и полученото дърво — с R.

Ето илюстрация на всички стъпки на алгоритъма.  
  
Тук 1. - приложението на Split, и 2 приложението на Merge — общото време за работа е O(log2 N). Ето кода на функцията.

public Treap Add(int x)

{

Treap l, r;

Split(x, out l, out r);

Treap m = new Treap(x, rand.Next());

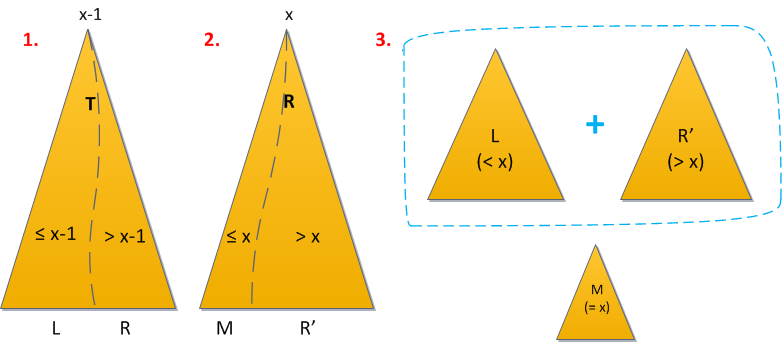
return Merge(Merge(l, m), r);

}

При изтриването също не възникват никакви въпроси. Нека да се налага да изтрием от декартовото дърво елемент с ключ x. Предполагам, че вече сте наясно с равенството на ключовете, ако оставим ключът **x** в лявото поддърво, то в дясното поддърво със сигурност нама да има елемент с ключ **x**. Тогава да извършим следната последовательност от действия:

1. Отначало разделяме дървото по ключ x-1. Всички елементи, по-малки или равни на x-1, отиват в лявото поддърво, което означава, че търсеният елемент е в дясното.
2. След това разделяме десния резултат по ключа x (тук разделянето е коректно да се направи с равенство!). В новия десен резултат отиват всички елементи с ключове, по-големи от **x**, а в «средниая» (левия от десния) — всички по-малки или равни на x. Но пъй като строго по-малките след първото стъпка бява отсяти, то средното дърво съдържа единствено търсения елемент.
3. Сега просто обединяваме отново лявото дърво с дясното, без средното, и дерамидата остава без ключа **x**.

Сега стана ясно, защо постоянно акцентирах вниманието ви на факта, че все пак е необходимо да се отчита равенството на ключовете. Наистина, ако вашият компаратор приеме, че елементите с равни ключове трябва да се изпратят в дясното подърво, то на первата стъпка ще се наложи да разделяте по ключа x, а на втората — по x+1. Ако не бяхме конкретизирали този въпрос, процедурата за изтриване в дадения вариант не би могла да бъде изпълнена — след втората стъпка средното дърво щеше да остане празно, а търсеният елемент някъде е избягал или наляво, или надясно, и върви го търси вече.



Времето за работа на операциите е все още O(log2 N), тъй като прилагаме 2 пъти Split и 1 път Merge.

Изходен код:

public Treap Remove(int x)

{

Treap l, m, r;

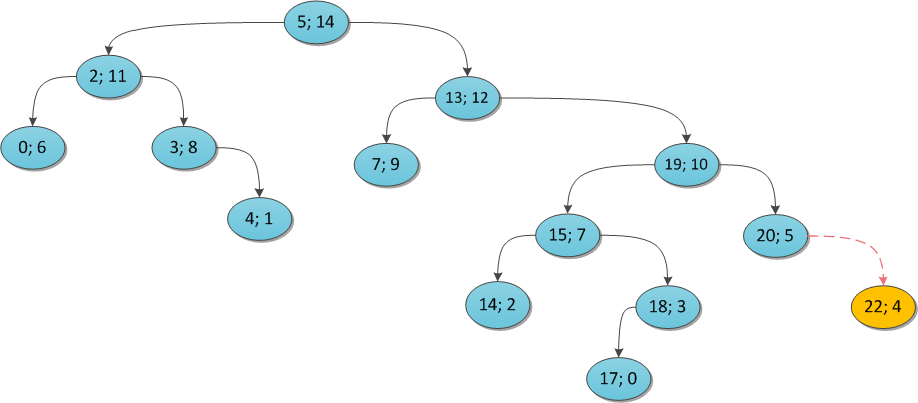
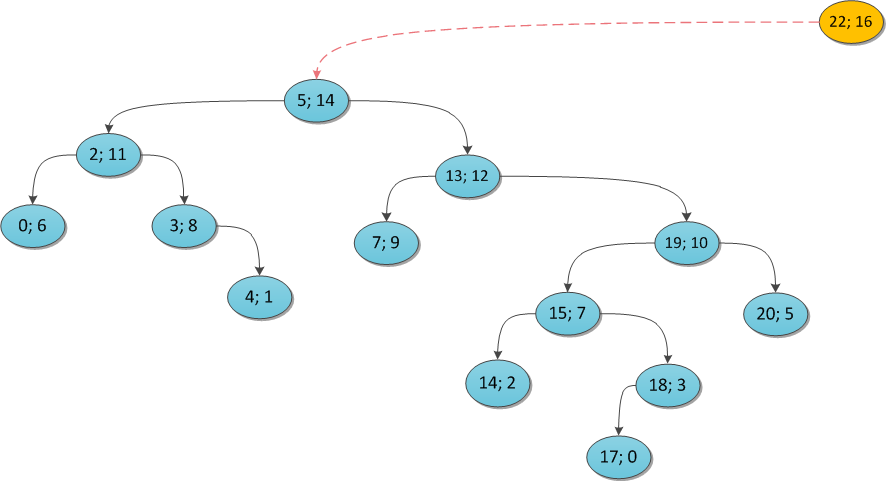
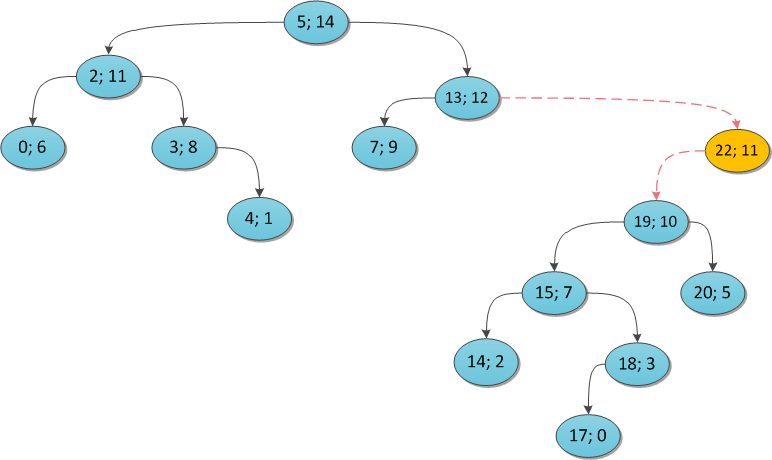
Split(x - 1, out l, out r);

r.Split(x, out m, out r);

return Merge(l, r);

}

Актът на създаването

Теперь, зная алгоритм добавления элемента в готовое декартово дерево, мы можем привести простейший способ построить дерево из поступающего набора ключей: просто добавлять их по очереди стандартным алгоритмом, начав с дерева из одной вершины — первого ключа. Помня, что операция добавления выполняется за логарифмическое время, мы получим общее время выполнения полного построения дерева — O(N log2 N).  
  
Интересно, а быстрее можно?  
  
Как оказалось, в некоторых случаях — да. Давайте представим, что ключи нам на вход поступают в возрастающем порядке. Такое в принципе вполне может произойти, если это какие-то свежесоздающиеся идентификаторы с auto increment. Так вот, в таком случае существует несложный алгоритм построения дерева за O(N). Правда, нам придется заплатить за это временным overhead`ом по памяти: хранить для каждой вершины строящегося дерева ссылку на ее предка (на самом деле даже не обязательно для каждой, но это уже тонкости).  
  
Будем хранить ссылку на последнюю добавленную вершину в дереве. По совместимости она будет в нем самой правой — ведь ключ у нее наибольший из всех ключей дерева, построенного на данный момент. Теперь допустим, что на вход поступает следующий ключ **x** с каким-то приоритетом **y**. Куда его поместить?  
  
Последняя вершина суть самая правая, следовательно, правого сына у нее нет. Если ее приоритет больше, чем y добавляемой, то можно просто приписать новую вершину правым сыном и с чистой совестью переходить к следующему ключу на входе. В противном же случае надо подумать. У новой вершины все равно наибольший ключ, так что в конечном итоге она точно станет самой правой вершиной дерева. Стало быть, искать место для ее вставки где-либо, кроме как по самой правой ветви, смысла не имеет. Найти же нам нужно всего лишь место, где приоритет вершины больше, чем y. Итак, поднимемся от самой правой вверх по ветви, каждый раз проверяя приоритет текущей осматриваемой вершины. В конце концов мы либо придем в корень, либо остановимся где-то посреди ветви.  
  
Предположим, мы пришли в корень. Тогда оказывается, что y больше, чем все приоритеты в дереве. У нас не остается другого выбора, кроме как сделать (x, y) новым корнем и привесить к ней старое дерево левым сыном.  
  
Если же в корень мы не пришли — ситуация похожая. В некоторой вершине правой ветви (x0, y0) имеем y0 > y. А у ее непосредственного правого потомка приоритет меньше y. Чтобы сохранить структуру дерева поиска по ключам, и сделать (x, y) самой правой в дереве, мы подвешиваем её как нового правого сына к (x0, y0), а все старое правое поддерево становится левым поддеревом (x, y).  
  
Чтобы было понятнее, я проиллюстрирую на примерах. Возьмем некоторое декартово дерево и покажем, что произойдет, если в него пытаться добавлять те или иные вершины. Ключ везде тот же (22), а приоритет поварьируем.  
y = 4:  
  
y = 16:  
  
y = 11:  
  
  
Почему этот алгоритм работает за O(N)? Заметьте, что каждую вершину дерева вы в ее жизни посетите максимум два раза:

* при ее непосредственном добавлении;
* возможно, при добавлении какой-то другой, пока она будет оставаться в правой ветви. Сразу после этого из правой ветви она уйдет и более посещаться не будет.

Таким образом, общее количество переходов не превосходит 2N, и ассимптотика построения — O(N).  
  
На закуску — исходный код построения по заданному массиву ключей и приоритетов. Здесь предполагается, что у каждой вершины дерева есть еще свойство Parent, а также что опциональный параметр с таким же именем есть у уже использовавшегося приватного конструктора вершины (пятый по счету).

public static Treap Build(int[] xs, int[] ys)

{

Debug.Assert(xs.Length == ys.Length);

var tree = new Treap(xs[0], ys[0]);

var last = tree;

for (int i = 1; i < xs.Length; ++i)

{

if (last.y > ys[i])

{

last.Right = new Treap(xs[i], ys[i], parent: last);

last = last.Right;

}

else

{

Treap cur = last;

while (cur.Parent != null && cur.y <= ys[i])

cur = cur.Parent;

if (cur.y <= ys[i])

last = new Treap(xs[i], ys[i], cur);

else

{

last = new Treap(xs[i], ys[i], cur.Right, null, cur);

cur.Right = last;

}

}

}

while (last.Parent != null)

last = last.Parent;

return last;

}

Резюме

Мы с вами построили древовидную структуру данных с такими свойствами:

* обладает почти гарантированно логарифмической высотой относительно количества своих вершин;
* позволяет за логарифмическое время искать любой ключ в дереве, добавлять его и удалять;
* исходный код всех её методов не превышает 20 строк, они легко понимаются и в них крайне сложно ошибиться
* содержит некоторый overhead по памяти, сравнительно с истинно самобалансирующимися деревьями, на хранение приоритетов.

В принципе, результат довольно-таки мощный. Однако некоторым может быть все же неясно, стоило ли ради него городить настолько большой огород с такой кучей текста. Стоило. Фишка в том, что возможности декартового дерева и потенциал его применения далеко не ограничиваются функциями, описанными в этой статье. Это лишь предисловие.  
  
В следующих частях:

1. Множественные операции над декартовым деревом (ищем за O(log2 N) сумму, максимум и т.д.)
2. Декартово дерево по неявному ключу (или как усовершенствовать обычный массив)
3. Ускоренные реализации функций декартового дерева (и их проблемы)
4. Функциональная реализация декартового дерева на F#.

Источники

Источники указываю раз и навсегда, они одни и те же для всех планируемых статей.  
  
В первую очередь при написании этих статей я основываюсь на лекции Виталия Гольдштейна, рассказанной на Харьковской зимней школе по программированию ACM ICPC в 2010 году. Ее можно загрузить из видеогалереи [школы](http://olimp.sc170.kharkov.ua/) (год 2010, день 2), как только она снова заработает, потому что в последние дни сервер что-то катастрофически барахлит.  
  
[Сайт](http://e-maxx.ru/) Максима «e-maxx» Иванова — богатый кладезь информации по разным алгоритмам и структурам данных, использующихся в спортивном программировании. В частности, есть на нем и [статья про декартово дерево](http://e-maxx.ru/algo/treap).  
  
В знаменитой книге Кормен, Лейзерсон, Ривест, Штайн «Алгоритмы: построение и анализ» можно найти доказательство того, что матожидание высоты случайного двоичного дерева поиска есть O(log2 N), хотя его определение случайного дерева поиска и отличается от того, что мы здесь использовали.  
  
Впервые дерамиды были предложены в статье Seidel, Raimund; Aragon, Cecilia R. (1996),[«Randomized Search Trees»](http://people.ischool.berkeley.edu/~aragon/pubs/rst96.pdf). В принципе там можно найти полный объем информации по теме.  
  
Пока что все. Надеюсь, вам было интересно :)