The Monad.Reader/Issue4/On Treaps And Randomization

From HaskellWiki

< The Monad.Reader | Issue4

Contents

1 Treaps and Randomization in Haskell

by Jesper Louis Andersen <jlouis@mongers.org> for The Monad.Reader, Issue Four, 05/07 2005

We give an example implementation of treaps (tree heaps) in Haskell. The emphasis is partly on treaps, partly on the System.Random module from the hierarchical libraries. We show how to derive the code and explain it in an informal style.

1.1 Introduction

I have, a number of times, warned people that I ought to do a TMR article. The world had its way, and I had to wait until the Summer to be able to finish an article. So this is it. Originally, I considered playing around with the ALL-PAIRS-SHORTEST-PATH algorithms, but for some reason I was not really satisfied with it. Also, with the upcoming Matrix library in the hierarchical libraries, this might prove to be a better solution.

Instead I will provide a treatise on the *Treap* data structure, devised by Aragon and Seidel. I have much to thank them for in the following. Usually citations are at the back of an article, but I really advise you to read *Randomized Search Trees* (Algorithmica, 16(4/5):464-497, 1996). This document owes about 90% to the mentioned article.

I also advise you to check out Oleg Kiselyov's work on treaps for Scheme. He does a

number of optimizations on the data structure which I have skipped over here. Take a look at Oleg's Scheme Treap implementation (http://okmij.org/ftp/Scheme/lib/treap.scm)

1.2 Search trees

The classic problem of computer science is how to express and represent a finite map in a programming language. Formally a finite map is a function *f: K --> V*, which is said to map a finite set *K*, of keys, to a (thus also finite) set *V*, of values. The basic functions are: *lookup(f, k)*, which will return the value *f(k)* in *V*, associated with the value *k* in *K*; *insert(f, (k,v))* which extends or updates the finite map with a new key/value pair; and finally, *delete(f, k)*, which removes the association of *k* from *f*.

One such representation is the binary search tree (In much literature, the acronym BST is used). I assume most readers of TMR are familiar with binary search trees, and especially the pathological case degenerating the worst case search time bounds to *O(n)*.

There are a number of strategies for avoiding the degenerate case where the tree becomes a linked list in effect. One could be to add invariants to the tree, which ensures that it stays inside certain balance bounds. One example is the AVL tree, which maintains the following invariant: At each node, the child-subtrees differ in depth by at most 1. This ensures the AVL-tree is always balanced and the pathological case where a tree is actually a list is ruled out. As a side note, an AVL tree will never be worse in structure than a Fibonacci tree (Knuth's *The Art of Computer Programming* Volume 3 has a good treatment of this tree type).

Another famous example is the Red-Black tree, which provides a less strict balance invariant than the AVL tree. The invariant is harder to describe in a single paragraph -- but involves colouring nodes either red or black and adding invariants such that the tree always stays reasonably balanced. See Introduction to algorithms by Cormen, Leiserson, Rivest and Stein if you want to read the hard, incomprehensible imperative version of this data structure, or Purely Functional Data Structures by Chris Okasaki if you want the functional approach to this (the functional code is a mere 12 lines without the **delete** operation).

The **Data.Map** module in the hierarchical libraries of Haskell use another type of tree known as the *2-3 tree*. The *2-3 tree* is self-balancing but uses a different trick. A *2-3 tree* node contains either one or two *(k, v)*-pairs and thus has either 2 or 3 children. We call a node with 2 children a 2-node and a node with 3 children a 3-node. Insertions are always done into leaf nodes which can grow from 2-nodes to 3-nodes in a natural way. Growth of a 3-node is then done by splitting the node into two 2-nodes; the least *(k, v)*-pair, the greatest pair and the middle pair is inserted into the parent node (there is a reference inside the documentation of the Data.Map).

> Sadly, the above is not true. Data.Map has binary trees which are balanced according to the size of the left and right subtrees. If one subtree grows beyond a certain constant factor it is rebalanced -- Jesper Andersen

Yet another variant of the finite map is the splay tree. In the splay tree the rebalancing is done according to a simple heuristic which amortized over a certain number of operations yields *O(lg n)* worst case running time. Splay trees are not that good for

purely functional languages, since they change the tree for all operations, including **lookup**. Thus our type for a **lookup** function would be:

```
Splay_lookup :: key -> SplayTree key value -> (Maybe value, SplayTree)
```

As a consequence, the programmer has to *thread* his splay tree around where she wants to use it. This tends to clutter the code a great deal. Further, splay trees are not friendly to a cache or page hierarchy, since the constant updating of nodes tends to dirty more pages/cache lines than necessary - but it hurts an imperative language more than a functional one which already has a fair deal of copying to do, due to persistence.

- **Side Note**: a paging hierarchy can be seen as a cache

hierarchy, if you take the swap space as the lowest level, page table mapped pages as the second level and TLB mapped pages as the third (and fastest) level.

1.3 Heaps

A basic Queue (FIFO) is something I assume all know. A priority queue is a queue, where each element is assigned a priority from a totally ordered set P. Elements in the priority queue are extracted according to the order of the priorities. For the case where the order is increasing, the queue is often called a "min-priq", since the minimum priority element is extracted first. Of course, a "max-priq" is also possible.

Priority queues are often implemented as heaps. In a functional setting, a very simple heap to program is the pairing heap, which takes no more than 12 lines of Haskell. Unfortunately, this article is not about pairing heaps. Instead, we need the all familiar binary heap.

A binary heap is a binary tree, where each node is a queue element and a priority. For the min-priq case, each node in the tree has a priority less than the priorities of its children. If a node is placed at the leaf of such a tree, it can be *floated* up by comparing it and its parent, eventually exchanging their places until the priority invariant has been fulfilled. Similarly, a node can be floated down by comparing the children priorities to each other, and exchanging the node for the child with the least priority.

1.4 Treaps

So, why attempt another data structure for the finite map problem? One, it is fun. Two, this algorithm is so simple, it can be explained in a single, tiny(??), TMR article. Third, we need more TMR articles. Simplicity usually means a fast algorithm. Benchmarking treaps against **Data.Map** was my original idea and maybe a follow-up article will do carry out this benchmarking.

While the introduction mentions finite maps, we will explore the simpler case where V is the singleton {True} set. The map *f* then represents a set of keys *K*, since a key is either mapped to **True** or it is not, in which case we can return **False**. Thus, we do not even bother storing the singleton {True} set in the Treap structure. However, extending the

treap to also posses arbitrary value data at each node is trivial and left as an exercise to the (interested, practically oriented) reader.

Let *K* be a totally ordered space of keys. It is clear a binary search tree can be formed obeying this order. Formally, for each node, the left subtree contains keys less than the key at the node and the right subtree contains keys greater than the key at the node.

Let *P* be a totally ordered set of priorities. It is clear we can form a binary min-heap containing the elements of *P*. Formally, for each node, the subtrees contains keys ordering greater than the key at the node.

Associate with each key *k* in *K* a priority *p* in *P*. A **Treap** is then a binary tree obeying the binary search tree property with respect to the *K*s as well as the min-priq property of the *P*s. Now, if the priorities are chosen randomly, we will actually achieve a balanced tree (!). It might be wise to try to draw such a tree. In fact it is unique. To see this, construct the tree by inserting *K*s in increasing order of priorities, by using the binary search tree **insert** algorithm.

1.5 Show me the code!

Enough talk. Haskell! A module representing treaps is first defined:

```
module Treap (
 RTreap
 , empty
 , null
 , insert
 , delete
 , member
 , stdGenTreap
 , splitTreap
 , joinTreap
) where

import System.Random
import Prelude hiding (null)
```

A treap is a binary search tree, where each node has a key and a priority:

```
data Treap k p = Leaf | Branch (Treap k p) k p (Treap k p)
 deriving (Show, Read)
```

The empty tree and the null predicate are simple. They are copied verbatim from the binary search tree:

```
treap_Empty :: Treap k p
treap_Empty = Leaf

treap_Null :: Treap k p -> Bool
treap_Null Leaf = True
treap_Null _ = False
```

Insertion into a treap works by inserting the node, as if inserting into a binary search

tree. Then we use the famous left- and right-rotations to float the node up, until it fulfills the heap-property on its priority. If you are not familiar with left and right rotations, they are just restructurings of a binary search tree, maintaining the ordering property. What is important is they alter the heights of the subtrees and so can help balance the tree more. They are easily definable in Haskell by pattern matching. Drawing them on paper is a good exercise:

```
rotateLeft :: Treap k p -> Treap k p
rotateLeft (Branch a k p (Branch b1 k' p' b2)) =
 Branch (Branch a k p b1) k' p' b2
rotateLeft _ = error "Wrong rotation (rotateLeft)"

rotateRight :: Treap k p -> Treap k p
rotateRight (Branch (Branch a1 k' p' a2) k p b) =
 Branch a1 k' p' (Branch a2 k p b)
rotateRight _ = error "Wrong rotation (rotateRight)"

treap_Insert :: (Ord k, Ord p) => k -> p -> Treap k p -> Treap k p
treap_Insert k p Leaf = Branch Leaf k p Leaf
treap_Insert k p (Branch left k' p' right) =
 case compare k k' of
 EQ -> Branch left k' p' right -- Node is already there, ignore
 LT -> case Branch (treap_Insert k p left) k' p' right of
 (t @ (Branch (Branch l' k p r') k' p' right)) ->
 if p' > p
 then rotateRight t
 else t
 t -> t
 GT -> case Branch left k' p' (treap_Insert k p right) of
 (t @ (Branch left k' p' (Branch l' k p r'))) ->
 if p' > p
 then rotateLeft t
 else t
 t -> t
```

When coding structures based upon binary trees it can be convenient to *forget* the deletion case. It is often the hardest case to grasp and it can be quite hard to maintain invariants associated with the tree such as the AVL-tree or Red/Black-tree. Not so for Treaps, however. We just locate the node by a binary tree search and then float it down by rotations until the node is a leaf using the heap-properties and operations. Then we cut off the leaf (Notice the nice metaphors, please).

```
treap_Delete :: (Ord k, Ord p) => k -> Treap k p -> Treap k p
treap_Delete k treap = recDelete k treap
 where recDelete k Leaf = error "Key does not exist in tree (delete)"
 recDelete k (t @ (Branch left k' p right)) =
 case compare k k' of
 LT -> Branch (recDelete k left) k' p right
 GT -> Branch left k' p (recDelete k right)
 EQ -> rootDelete t
 priorityCompare Leaf (Branch _ _ _ _) = False
 priorityCompare (Branch _ _ _ _) Leaf = True
 priorityCompare (Branch _ _ x _) (Branch _ _ y _) = x < y
 rootDelete Leaf = Leaf
 rootDelete (Branch Leaf _ _ Leaf) = Leaf
 rootDelete (t @ (Branch left k p right)) =
 if priorityCompare left right
 then let Branch left k p right = rotateRight t
 in Branch left k p (rootDelete right)
 else let Branch left k p right = rotateLeft t
 in Branch (rootDelete left) k p right
```

We must not forget the **member** function. This is simple, as it is nothing but the original binary search tree function:

```
treap_Member :: (Ord k, Ord p) => k -> Treap k p -> Bool
treap_Member e Leaf = False
treap_Member e (Branch left k _ right) =
 case compare e k of
 LT -> treap_Member e left
 GT -> treap_Member e right
 EQ -> True
```

1.6 Providing random priorities

The premise of the Treap algorithm is the provision of a good random number generator. If the priorities are randomly assigned, the tree will be balanced well (with a high probability). So, our next quest is to assign priorities randomly to each node. The random assignment also makes it impossible for an evil adversary to unbalance the structure.

There are numerous possibilities, but the one shining most is the **System.Random** library. The library provides us with 2 type classes **Random``Gen** and **Random**. The **Random``Gen** class are those types *g*, which can be used as random number generators. The **Random** class on the other hand are those types *a*, from which random values can be drawn. That is, given a type of class **Random``Gen**, any value with a type of class **Random** can be drawn from it.

The **System.Random** library also provides a standard random number generator. For our purpose it has the disadvantage of being wrapped inside the **IO** monad and having to rely on a monad for our treap operations is bad since we then have to thread the monad around with us.

Thus the plan is the following: Initialize a treap as a random number generator and the structure above. Then maintain the random number generator while running operations in the treap. We call this structure an **RTreap**:

```
newtype RTReap g k p = RT (g, Treap k p)
 deriving (Show, Read)
```

The empty treap is then an initialization of the random number generator, as said. The null predicate is just a simple re-usage of the function above:

```
empty :: RandomGen g => g -> RTreap g k p
empty g = RT (g, treap_Empty)

null :: RandomGen g => RTreap g k p -> Bool
null (RT (g, t)) = treap_Null t
```

Insertion into the treap is done by requesting a new random number from our supply and using this for the node in question. Delete and member are just the same from above with some added structure.

Note we draw random values in a bounded area, such that we have a value less than

every random priority in the treap and a value greater than every random priority in the heap. There are certain tricks which can be pulled with these values.

```
insert :: (RandomGen g, Ord k, Ord p, Num p, Random p)
 => k -> RTreap g k p -> RTreap g k p
insert k (RT (g, tr)) =
 let (p, g') = randomR (-2000000000, 2000000000) g
 in RT (g', treap_Insert k p tr)

delete :: (RandomGen g, Ord k, Ord p) => k -> RTreap g k p
 -> RTreap g k p
delete k (RT (g, tr)) = RT (g, treap_Delete k tr)

member :: (RandomGen g, Ord k, Ord p) => k -> RTreap g k p
 -> Bool
member k (RT (g, tr)) = treap_Member k tr
```

The initialization of the **RTreap** will then be something like:

```
stdGenTreap :: Int -> RTreap StdGen k p
stdGenTreap = (empty . mkStdGen)
```

The *Int* type one has to provide is an initialization seed. We can get one such inside an **IO** monad when starting our program and then use it to seed the Treaps we need afterwards. The functions needed are defined inside the **System.Random** module.

1.7 Cool additions

If we wish to split a treap at a certain node k in K, we can do so, by inserting k with the minimum priority. Assuming p are in the **Bounded** class:

```
splitTreap :: (RandomGen g, Bounded p, Ord k, Ord p)
 => k -> RTreap g k p -> (RTreap g k p, RTreap g k p)
splitTreap k (RT (g, tr)) =
 let (g', g'') = split g
 Branch left _ _ right = treap_Insert k minBound tr
 in (RT (g', left), RT (g'', right))
```

Similarly to join two *disjoint* treaps with key spaces K1 and K2, where the keys in K1 are smaller than the keys in K2 (formally: max K1 < min K2), we can choose a key k not in the union (K1, K2) and form the tree where k is the root and the treaps are left and right children. We then proceed by deleting the node k:

```
joinTreap :: (Bounded p, Ord p, Ord k)
 => k -> RTreap g k p -> RTreap g k p -> RTreap g k p
joinTreap k (RT (g, tr1)) (RT (_, tr2)) =
 RT (g, (treap_Delete k (Branch tr1 k maxBound tr2)))
```

1.8 Optimizations

I will simply direct people to the article by Oleg pointed at in the introduction. There are certain optimizations possible, which he thoroughly discusses. Implementing these is an

exercise.

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue4/On_Treaps_And_Randomization"

Category: Article
