J. Functional Programming **8** (6): 627–632, November 1998. Printed in the United Kingdom

FUNCTIONAL PEARLS

Diets for fat sets

MARTIN ERWIG
FernUniversität Hagen, Praktische Informatik IV. 58084 Hagen, Germany
(*e-mail:* erwig@fernuni-hagen.de)

1 Introduction

In this paper we describe the *discrete interval encoding tree* for storing subsets of types having a total order and a predecessor and a successor function. In the following, we consider for simplicity only the case for integer sets; the generalization is not difficult.

The discrete interval encoding tree is based on the observation that the set of integers $\{i \mid a \leq i \leq b\}$ can be perfectly represented by the closed interval $[a, b]$. The general idea is to represent a set by a binary search tree of integers in which maximal adjacent subsets are each represented by an interval. For example, inserting the sequence of numbers $6, 9, 2, 13, 8, 14, 10, 7, 5$ into a binary search tree, respectively, into a discrete interval encoding tree results in the tree structures shown in figure 1.

The efficiency of the interval representation, both in terms of space and time, improves with the density of the set, i.e. with the number of adjacencies between set elements. So what we propose is a 'diet' (discrete interval encoding tree) for 'fat' sets in the sense of 'the same amount of information with less nodes'.

In the next section, we define the discrete interval encoding tree with operations for inserting, deleting and searching for elements. An analysis is presented in section 3, and we comment upon some applications and actual running times in section 4.

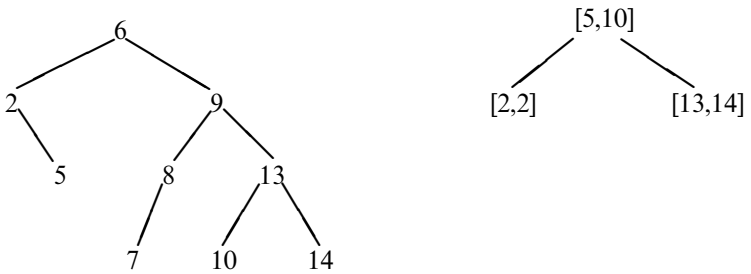

Fig. 1. Binary tree and discrete interval encoding tree.

https://doi.org/10.1017/S0956796898003116 Published online by Cambridge University Press

2 The diet

The nodes in a diet store intervals. This means that isolated elements will be represented by degenerate one-element intervals. This greatly simplifies the following definitions, and so we will use the following type:

```
datatype diet = EMPTY | NODE of int * int * diet * diet
```

All definitions are given in ML, the translation to other functional languages is obvious.[1] In an interval node the first integer gives the left border and the second integer gives the right border of the stored interval. This means that the integer pairs in a diet T are actually ordered:

$$\forall\ [x, y] \in T : x \leq y$$

An important invariant for any diet is that its intervals neither overlap nor touch. More precisely, between each two intervals of a diet T there is a gap of at least one element:

$$\forall\ [u, v], [x, y] \in T : v + 1 < x\ \vee\ y + 1 < u$$

Another view of this invariant is that intervals in a diet are always chosen maximally. All operations on diets must keep this invariant intact.

Testing for set membership in a diet is similar to binary search trees:

```
fun member (z,EMPTY) = false
 |  member (z,NODE (x,y,l,r)) =
    if z>=x andalso z<=y then true else
    if z<x then member (z,l) else
    (* z>y *)   member (z,r)
```

This definition is very simple, since the tree is not changed, and we thus need not be concerned about maintaining the invariant.

Inserting an element z into a diet T follows principally the insertion algorithm for binary search trees, but it has to consider the following special cases: if z extends an existing interval, i.e. if

$$\exists\ [x, y] \in T :\ \text{(i)}\ z = x - 1\ \vee\ \text{(ii)}\ z = y + 1,$$

then instead of creating a new node for z in T, the interval $[x, y]$ is changed to $[z, y]$ in case (i) or to $[x, z]$ in case (ii). Once such an interval extension has happened we have to be careful to maintain the tree invariant: there might now be an interval $[u, v]$ in the left subtree of $[z, y]$ (or in the right subtree of $[x, z]$) with $v + 1 = z$ (or with $z + 1 = u$). If this is the case, both intervals can and must be fused into one, i.e. into $[u, y]$, respectively, into $[x, v]$. The candidate interval for such an invariant violation is either the predecessor of $[z, y]$ or the successor of $[x, z]$ in T. These can be found as the maximum (minimum) of the left (right) subtree of $[z, y]$ ($[x, z]$).

[1] For ML and Haskell sources, see `http://www.fernuni-hagen.de/inf/pi4/erwig/diet`

https://doi.org/10.1017/S0956796898003116 Published online by Cambridge University Press

The adjustment that is possibly needed after an interval extension is realized by two functions `joinLeft` and `joinRight` that make use of two functions `splitMax` and `splitMin` for splitting off the maximum/minimum interval in the left/right subtree. Below we only show the implementation for `splitMax` and `joinLeft`, the case for `splitMin` and `joinRight` is analogous:

```
fun splitMax (NODE (x,y,l,EMPTY)) = (x,y,l)
 |  splitMax (NODE (x,y,l,r)) = let val (u,v,r') = splitMax r
                                 in (u,v,NODE (x,y,l,r')) end

fun joinLeft (t as NODE (x,y,EMPTY,r)) = t
 |  joinLeft (NODE (x,y,l,r)) =
    let val (x',y',l') = splitMax l
     in if y'+1=x then NODE (x',y,l',r)
                  else NODE (x,y,l,r)
    end
```

If the left subtree `l` of a left-extended interval, say i, is empty, `joinLeft` has nothing to do. Otherwise, `splitMax` yields the borders `x'` and `y'` of the maximal interval i' in `l` together with `l'` which is `l` with i' being removed. If i' is adjacent to i, i is extended further by changing its left border to `x'` and by replacing its left subdiet with `l'`. Otherwise, i remains unchanged.

Now the function `insert` proceeds much like for plain binary search trees, except for considering interval extensions:

```
fun insert (z,EMPTY) = NODE (z,z,EMPTY,EMPTY)
 |  insert (z,t as NODE (x,y,l,r)) =
    if z<x then
       if z+1=x then joinLeft (NODE (z,y,l,r))
                else NODE (x,y,insert (z,l),r) else
    if z>y then
       if z=y+1 then joinRight (NODE (x,z,l,r))
                else NODE (x,y,l,insert (z,r))
    else t (* z in [x,y] *)
```

Deleting an integer z from the tree T is again almost identical to the case for binary search trees. If T is empty, we are finished. Otherwise, we have to distinguish the following cases: let $[x, y]$ be the current interval under consideration. If $z \notin [x, y]$, deletion has to recursively move into the left or right subtree. If $z \in [x, y]$, we proceed as follows: if $x = y$, the node must be deleted, which means if both subtrees are not empty to replace it with the maximum of the left subtree. This is achieved by the function `merge` using `splitMax`. If $y > x$ and if $z = x$ or $z = y$, the interval can be simply reduced at the corresponding border. Otherwise, that is, when $x < z < y$, the interval must be split into two intervals $[x, z - 1]$ and $[z + 1, y]$. Both intervals are of size ≥ 1, and the invariant of a non-empty gap is also maintained.

https://doi.org/10.1017/S0956796898003116 Published online by Cambridge University Press

```
fun merge (l,EMPTY) = l
 |  merge (EMPTY,r) = r
 |  merge (l,r)     = let val (x,y,l') = splitMax l
                        in NODE (x,y,l',r) end

fun delete (z,EMPTY) = EMPTY
 |  delete (z,NODE (x,y,l,r)) =
    if z<x then NODE (x,y,delete (z,l),r) else
    if z>y then NODE (x,y,l,delete (z,r)) else
    if z=x then
       if x=y then merge (l,r)
              else NODE (x+1,y,l,r) else
    if z=y then NODE (x,y-1,l,r)
           else NODE (x,z-1,l,NODE (z+1,y,EMPTY,r))
```

Since the invariant cannot be violated by `delete`, no further adjustment as in the case of `insert` is needed. Note that the last line, which realizes the interval split case, might cause the diet to get out of balance. This can be improved by using

```
NODE (x,z-1,l,insertMin (z+1,y,r))
```

instead, where `insertMin` is the inverse of `splitMin`, i.e. a specialized version of `insert` placing the interval $[z+1, y]$ at the leftmost position in r.

3 Best, worst and average case analysis

In the best case, when there are no 'holes' in the stored set, the interval representation consists of just one single interval. This means $O(1)$ space requirement, and finding, inserting or deleting an element can be done in constant time. In the worst case, there are no two adjacent elements in the set, and the representation has the same number of nodes as an ordinary binary search tree (although with the simplified implementation described here each stored integer is duplicated). Thus, the space requirement of a diet is never worse, asymptotically, than for a binary tree, but is in many cases better. Concerning running time we observe that `insert`, `delete` and `member` work almost the same as for ordinary binary trees, except for the occasional necessity of joining adjacent intervals. The required number of steps is, however, bounded by the height differences of the two intervals to be joined. This means that a corresponding insertion into an ordinary binary tree requires the same number of steps anyhow. (Some tests have revealed that there is actually not much time spent on joining intervals.)

In practice, it is more interesting to figure out the performance in the average case than just knowing the best and worst case of the structure. As a measure for the complexity of tree operations, we can take the number of nodes in the tree because the tree height, which gives an upper bound for the time spent by all three presented operations, grows at least logarithmically with the number of nodes. Compared with an ordinary binary search tree (balanced or not), the number of nodes is reduced

https://doi.org/10.1017/S0956796898003116 Published online by Cambridge University Press

by exactly the number of adjacencies among the stored elements where two distinct integers x and y are adjacent iff $|x-y| = 1$. More precisely, we call a pair (x, y) with $x + 1 = y$ an *adjacency*.

Let us consider the representation of an arbitrary subset $S \subseteq U = \{1, \ldots, n\}$ with $|S| = k$, and let $N(k, n)$ denote the average number of nodes in a diet for S. It is clear that the (average) number of nodes in an ordinary binary tree is always k, and thus $N(k, n)$ is given by subtracting from k the average number of adjacencies in S. This average number of adjacencies can be obtained by determining $A(k, n)$, the number of all the adjacencies in all k-subsets, and dividing by the number of such subsets, $\binom{n}{k}$. A formula for $A(k, n)$ is obtained by the following consideration:

We observe that there are exactly $n-1$ adjacencies in U, namely $(1, 2)$, $(2, 3)$, …, $(n-1, n)$. Now let us find out how many times each adjacency can occur in any k-subset ($k \geq 2$), i.e. let

$$C(x, y, k, n) = |\{S \in 2^U : |S| = k \wedge \{x, y\} \subseteq S\}|$$

All the k-subsets S that contain x and y can be constructed by fixing x and y and choosing the missing $k-2$ elements from the remaining $n-2$ elements of U. This means that $C(x, y, k, n) = \binom{n-2}{k-2}$. Hence the number of all adjacencies is given by the sum of $C(x, y, k, n)$ for all possible adjacencies (x, y) of which there are exactly $n-1$. Thus:

$$A(k, n) = \binom{n-2}{k-2}(n-1) = \binom{n-1}{k-1}(k-1)$$

The average number of adjacencies is therefore

$$\begin{aligned} A(k, n)/\binom{n}{k} &= \binom{n-1}{k-1}(k-1)/\binom{n}{k} \\ &= \frac{(n-1)!(k-1)k!(n-k)!}{(k-1)!(n-k)!n!} \\ &= \frac{(k-1)k}{n} \end{aligned}$$

The average number of nodes in a diet is thus

$$N(k, n) = k - \frac{(k-1)k}{n} = k\left(1 - \frac{(k-1)}{n}\right)$$

Since k/n is a measure for the density of the elements in the stored set, this means that the number of nodes in a diet reduces linearly with growing density of the set.

4 Applications and some test results

The diet is not meant to be a general purpose set representation. Here we have only presented a minimal set of operations (also known as a dictionary data type). Although a true set data type with union, difference, etc. is also possible, it requires a lot of effort to keep the interval invariant intact without sacrificing efficiency. Most of the additional operations are amenable to divide-and-conquer solutions,

https://doi.org/10.1017/S0956796898003116 Published online by Cambridge University Press

however, we have, for example, to consider and separately deal with all 13 possible cases of interval overlapping in the implementation of set difference. Another way of generalizing sets is to extend them to a mapping data type. This is, in principle, also possible for diets, but the opportunities for exploiting adjacencies will be infrequent because this is only possible for adjacent integers that are mapped to the same value. (The same comment applies to the diet representation of bags.)

However, for certain application areas the interval representation performs quite well. For example, for the part of Andrew Appel's integer set benchmark[2] doing only (random) insertions, deletions and retrievals, diets are about 45% faster than the winning implementation of Adams (1993). This picture will be certainly different if we compare a general purpose extension of diets, in particular, if we work with sorted input (although balancing strategies can also, in principle, be applied to diets as well).

A particularly well-suited application area is given by graph algorithms: many algorithms move through the graph and have to remember the nodes they have already visited. In a functional setting this means to thread a set of visited nodes through function calls. What makes this particularly interesting for diets is that these node sets are typically growing until they are completely filled. For example, depth first search on sparse graphs (of degree 4) takes 30% longer when using the balanced trees of Adams (1993) than with using diets. It is evident that the payoff of building intervals rises with the number of lookups, and so, as expected, for denser graphs the advantage of diets is even larger, e.g. for a node degree of 25 depth first search with diets is twice as fast as with balanced trees.

Acknowledgement

I would like to thank Richard Bird for giving me valuable comments on an earlier version of this paper. Thanks also to Andrew Appel for putting his benchmarks at my disposal.

References

Adams, S. (1993) Efficient Sets – A Balancing Act. *J. Functional Programming*, **3**(4), 553–561.

[2] This benchmark was used in 1992 in a competition to find a good functional integer set implementation in ML.