J. Functional Programming 3 (4): 553–561, October 1993

FUNCTIONAL PEARLS

Efficient sets—a balancing act

STEPHEN ADAMS

Electronics and Computer Science Department, University of Southampton, UK

Capsule Review

In late 1991 I organized an international programming competition for the Standard ML community. Each entrant implemented the 'set of integers' abstract data type, matching a signature that I provided. Prizes (donated by MIT Press) were awarded in two categories: fastest program (on a particular benchmark), and most elegant yet still efficient program.

More than a dozen entries were received; the top four entries in the speed category are listed here:

		$N = 10^5$	$N = 10^4$
1	Jon Freeman, U. of Pennsylvania	128 sec	10.1 sec
2	Stephen Adams, U. of Southampton	132	8.9
3	Thomas Yan & Sendhil Mullainathan, Cornell U.	189	16.7
4	Eugene Stark, State U. of New York at Stonybrook	226	18.5

The winner in the elegance category was Stephen Adams. His program was almost as fast as Freeman's for very large sets, and was faster for smaller sets.

In this 'functional pearl', Adams describes a generalization of his competition entry.

Andrew W. Appel

1 Introduction

We present an implementation of sets using balanced binary trees, written in Standard ML (SML). Binary trees are an important data structure, especially in the functional world where mutable data structures are not available. Unfortunately, to guarantee the nice properties of trees, like logarithmic lookup, it is necessary to keep the trees balanced. Balancing algorithms are usually complicated. We demonstrate that this need not be the case—the trick is to abstract away from the rebalancing scheme to achieve a simple and efficient implementation.

2 Specification

A general purpose set package should implement a wide range of set operations efficiently, including at least those listed in the signature `SET` in Fig. 1. This says that we have a type `Set` which represents sets of the type `Element`. The operations are

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

```
signature SET =
  sig
    type Element
    type Set
    val  empty : Set
    val  singleton : Element -> Set
    val  size : Set -> int
    val  member : Element * Set -> bool
    val  add : Element * Set -> Set
    val  delete :  Element * Set -> Set
    val  members : Set -> Element list
    val  union : Set * Set -> Set
    val  difference : Set * Set -> Set
    val  intersection : Set * Set -> Set
  end
```

Fig. 1. Common set operations.

meant to be obvious: `empty` is the empty set $\emptyset$, `singleton` creates a set containing exactly one element, `size` returns the cardinality of the set, `member` returns `true` iff the set contains the element. `add` and `delete` include or exclude an element, and `union`, `difference` and `intersection` perform the set operations $A \cup B$, $A - B$ and $A \cap B$. The operation `members` returns a list of the elements in a set. The elements in the list are in sorted order.

3 Tree representation

We assume that the type `Element` has a total ordering `lt`:

```
signature ORDER =
  sig
    type Element
    val lt : Element*Element -> bool
  end
```

The total ordering allows us to use an internal binary search tree, where the elements are stored in the nodes of the tree, and the elements in the left subtree of any node are all less than the element in the node and the elements in the right subtree are all greater.

To implement `size` efficiently we can store the count of the elements in a tree in the root node. Counting the nodes is just too expensive.

Nievergelt and Reingold (1973) show how the size of a tree can be used to keep it balanced. They call their trees *bounded balance binary trees*, and we use a similar idea. The efficiency these trees is within a few percent of AVL trees. Using the size is a bonus because it means that no extra information needs to be stored, for example the height as in height balanced trees, a height difference like in AVL trees or a 'colour' as in red-black trees. The saving in space is about 16% in New Jersey SML (NJ/SML). Small savings like this are often worth while because they have beneficial knock-on effects for object initialization and garbage-collection.

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

```
functor SizedTree(structure Order : ORDER) =
  struct
      open Order
      datatype  Tree = E  |  T  of  Element * int * Tree * Tree

      val empty = E

      fun size E = 0
        | size (T(v,n,l,r)) = n

      fun singleton item = T(item,1,E,E)

      fun member (x, E)   = false
        | member (x, T(v,n,left,right)) =
            if lt(x,v) then member(x,left)
            else if lt(v,x) then member(x,right)
                 else true

      fun  N(v,l,r) = T(v, 1 + size l + size r, l, r)

  end
```

Fig. 2. The basic tree data structure.

The tree data structure is an algebraic data type, either being empty or a node with constructor `T` containing the element, the size of the tree and the left and right subtrees:

```
datatype  Tree = E  |  T  of  Element * int * Tree * Tree
```

Now we can start on the implementation. The functor `SizedTree` (Fig. 2) collects together the operations that we can perform on trees with a stored size. The functor, like others in this paper, is parameterized with the properties of the type `Element` so that it can be reused for different element types.
The tree size obeys an invariant:

```
size (T(v,n,l,r))  =  n  =  1 + size l + size r
```

It is a good idea to use a *smart constructor* to build nodes to ensure that this invariant is never broken:

```
fun  N(v,l,r) = T(v, 1 + size l + size r, l, r)
```

`N` is really just an ordinary function but it will always be used in lieu of the 'native' constructor `T`. We still need to use `T` for pattern matching, however, so we cannot make this totally transparent.

4 Keeping trees balanced

The standard algorithm for inserting an element in an unbalanced binary tree searches for the element to find the position to insert the element. If it is not found then a singleton tree is created containing the new element at the empty tree where

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

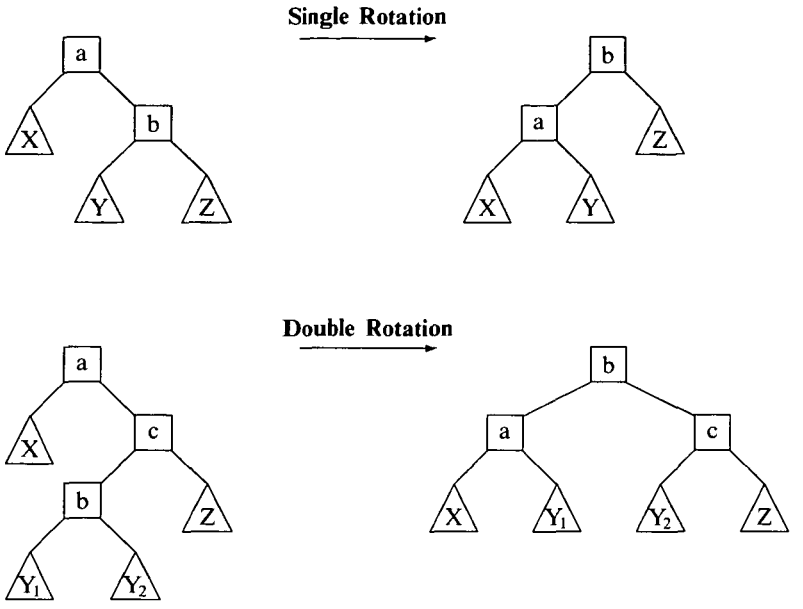

Fig. 3. Single and Double Left-Rotations. a<b<c are tree elements; X, Y and Z are trees. Note that rotations preserve the order of the elements.

the element ought to be. If it is found then the original tree is returned because it already contains the element.

```
fun add (x,E) = singleton x
  | add (x,tree as T(v,_,l,r)) =
          if lt(x,v)  then N(v,add(x,l),r)
      else if lt(v,x)  then N(v,l,add(x,r))
      else                  tree
```

The problem with this algorithm is that when it is used to insert an ordered sequence such as 1,2,3,... it produces a poorly balanced tree. The tree can be balanced if we replace the calls to `N` by a 'smarter' constructor that builds a balanced tree instead of the 'natural' tree. Such a function can only be devised if there is enough information to know if the tree would be unbalanced. Height-balanced and bounded balance trees have this information immediately available, but AVL trees (Adel'son-Vel'skii & Landis, 1962) do not.

The purpose of balancing is to keep the paths from the root to each element roughly the same length and short. The maximum path length in a balanced tree is logarithmic in the size of the tree. This is achieved by building *rotations* of the tree that place subtrees so that larger subtrees which tend to have longer path lengths are higher up than the shorter subtrees. There are two types of rotation which are illustrated in Fig. 3. These rotations and their mirror images can be expressed quite elegantly in SML as the operations `single_L`, `double_L`, etc., which construct the appropriate rotation of the tree that would be constructed by `N`.

An intuitive description of the rebalancing criteria is given here. More rigour is taken in the appendix of Adams (1992). A rotation must be used when the tree

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

```
functor BalancedTree(structure Order : ORDER) =
  struct
    structure TheTree = SizedTree(structure Order = Order)
    open TheTree

    fun single_L (a,x,T(b,_,y,z))            = N(b,N(a,x,y),z)
    fun double_L (a,x,T(c,_,T(b,_,y1,y2),z)) = N(b,N(a,x,y1),N(c,y2,z))
    fun single_R (b,T(a,_,x,y),z)            = N(a,x,N(b,y,z))
    fun double_R (c,T(a,_,x,T(b,_,y1,y2)),z) = N(b,N(a,x,y1),N(c,y2,z))

    val ratio = 5
    fun B (p as (v,l,r)) =
        let val ln = size l
            val rn = size r
        in
            if  ln+rn < 2  then  N p
            else if rn>ratio*ln then (*right is too big*)
                let val T(_,_,rl,rr) = r
                    val rln = size rl
                    val rrn = size rr
                in
                    if rln < rrn then  single_L p  else  double_L p
                end
            else if ln>ratio*rn then (*left is too big*)
                let val T(_,_,ll,lr) = l
                    val lln = size ll
                    val lrn = size lr
                in
                    if lrn < lln then  single_R p  else  double_R p
                end
            else  N p
        end

    fun add (x,E) = singleton x
      | add (x,tree as T(v,_,l,r)) =
          if lt(x,v) then B(v,add(x,l),r)
          else if lt(v,x) then B(v,l,add(x,r))
               else tree

    exception Homework
    fun delete (x,tree) = raise Homework
  end
```

Fig. 4. Balanced tree operations.

that would otherwise be built is unbalanced. In particular, we should not build a tree that has more than w times the number of elements in one subtree than the other. This is equivalent to saying that one subtree must never be more than a fixed amount higher than its sibling. Figure 3 shows the effect of rotating a tree that has a right subtree that is too big. We call w the *weight ratio.*

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

The important difference between the two rotations is that the single rotation lifts Z relative to X and Y, whereas the double rotation lifts Y. So a good choice of rotation is to pick a single rotation when Z is heavier and a double if Y is heavier. This algorithm is implemented by the smarter constructor, called `B`, for balance, which is the centrepiece of the functor `BalancedTree` in Fig. 4.

The first test ensures that the two subtrees are big enough to achieve anything by rebalancing. Note also that the triple of parameters is named as `p` by using a *layered pattern*. The parameters are passed on to the other functions simply as `p`, without the possibility of getting them mixed up.‡ The algorithm preserves the invariant $\texttt{ln}/w \leq \texttt{rn} \leq \texttt{ln} * w$ provided that the two subtrees are not too far out of balance. Plug `B` into the code for `add` in place of `N` and *voilà*: a rebalancing implementation of `add`. The identical trick works with the textbook version of `delete`, which we have left as a homework exercise.

5 Set×Set→Set operations

Crane (1972) shows how to concatenate two balanced trees and, the inverse operation, how extract an substring from a tree. For concatenation all of the values in the first tree must be less than any of the values in the second tree. Concatenation is a special case of union. Crane's algorithm finds the position in the larger tree where the smaller tree can be inserted without unbalancing the tree at that position. The higher parts of the new tree may then need rebalancing just as in `add`. We implement this idea using the balancing factor w to find the position and `B` to rebalance the higher parts of the tree. The function `concat3`, in Fig. 5, is the natural successor to `B` and `N` in a hierarchy of smart constructors. This function forms a balanced tree from two non-overlapping trees of arbitrary size and a third item, an element which lies between the values in the first tree and those in the second tree. It runs in time proportional to the difference in the height of the trees, so it is very fast for similar sized trees, and degenerates to an insert when joining a tiny tree to a big tree. This property of `concat3` is used in an $O(\log n)$ algorithm to split a tree selecting all the elements less than a certain value, corresponding to the set operation $\texttt{split\_lt}(S,a) = \{x \in S | x < a\}$. The splitting algorithm cuts out all the unwanted subtrees and recombines the fragments left over. It takes time only $O(\log n)$ because `concat3` combines small fragments of tree to make bigger trees before combining the result with the bigger fragments, so its arguments are usually similar in size. The symmetrical operation `split_gt` is similar.

We can use these tools to devise a divide-and-conquer algorithm that will compute the union $A \cup B$ of two arbitrary sets. We use the following properties of union

- $A \cup \emptyset = \emptyset \cup A = A$.
- $a \in A \quad \Rightarrow \quad A \cup B = \begin{cases} & \{x \in A | x < a\} \ \cup \ \{x \in B | x < a\} \\ \cup & \{a\} \\ \cup & \{x \in A | x > a\} \ \cup \ \{x \in B | x > a\} \end{cases}$

‡ This might also be faster on a naive system or an interpreter.

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

```
functor BalancedTreeCollection(structure Order : ORDER) =
  struct
    structure TheTree = BalancedTree(structure Order = Order)
    open TheTree

    fun concat3 (v,E,r) = add(v,r)
      | concat3 (v,l,E) = add(v,l)
      | concat3 (v, l as T(v1,n1,l1,r1), r as T(v2,n2,l2,r2)) =
          if ratio*n1 < n2 then B(v2,concat3(v,l,l2),r2)
          else if ratio*n2 < n1 then B(v1,l1,concat3(v,r1,r))
               else N(v,l,r)

    fun split_lt (E,x) = E
      | split_lt (T(v,_,l,r),x) =
          if lt(x,v) then split_lt(l,x)
          else if lt(v,x) then concat3(v,l,split_lt(r,x))
               else l

    fun split_gt (E,x) = E
      | split_gt (T(v,_,l,r),x) =
          if lt(v,x) then split_gt(r,x)
          else if lt(x,v) then concat3(v,split_gt(l,x),r)
               else r

    fun union (E,tree2)  = tree2
      | union (tree1,E)  = tree1
      | union (tree1, T(a,_,l,r)) =
          let val l' = split_lt(tree1,a)
              val r' = split_gt(tree1,a)
          in
              concat3(a, union(l',l), union(r',r))
          end

    fun difference (tree1,tree2) = raise Homework
    fun intersection (tree1,tree2) = raise Homework
end
```

Fig. 5. Splitting and combining trees

This gives us the algorithm for `union` (Fig. 5). `union` runs in worst case time $O(n+m)$ where n and m are the sizes of the inputs. The logarithmic cost of `split_lt` (and possibly `concat3`) is out-powered by the exponential growth of recursive calls to `union` on smaller trees. On fortuitous inputs, like sets taken from disjoint ranges, or huge with tiny sets, the performance of `union` is considerably better and degrades gracefully to the worst case.

Asymmetric set difference and intersection may also be implemented using the divide-and-conquer framework. We have left them as an exercise here, but full implementations and analysis are given in Adams (1992).

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

6 Iterating over sets

It is a common practice to provide a function to gather the elements of a collection together in a computation. The standard environment provides `fold` which does this for lists. We copy this idea in `tree_fold` which combines the elements from right-to-left and in-order:

```
fun tree_fold  f  base  E            = base
  | tree_fold  f  base  (T(v,n,l,r)) =
      tree_fold  f  (f (v,tree_fold f base r))  l
```

This traversal is in-order because the call to `f` is between the two recursive calls to `tree_fold`, and it is right-to-left because `r` is innermost, being processed before `v` and then `l`. Other traversals can be built by changing the order of the calls to `f` and `tree_fold`. An in-order traversal is perhaps the most useful because it can make a list of all the elements in the tree, in ascending order:

```
fun members tree = tree_fold  (op ::)  []  tree
```

An important feature of `tree_fold` is that it has a type similar to that of the standard list `fold`. This hides the tree implementation and allows list-based programs to be easily modified to use to trees. Since these functions are independent of the shape of the tree they belong in the `SizedTree` functor.

7 Finishing touches

At this point we have balanced trees that do all that is needed to implement the specified set operations. All that remains is to use the implemention to provide the abstract data type specified by the signature `SET`:

```
functor OrderedSet(structure Order : ORDER) : SET =
  struct
    structure Implementation =
        BalancedTreeCollection(structure  Order = Order)
    open Implementation
    type Set = Tree
  end
```

This last step is important: it separates the concrete implementation from the abstract type, leaving the implementation free for use in another application. At this point we may correct any small discrepancies. Only a new name was needed in this case.

8 Conclusion

Balanced binary trees are a difficult subject, often left as an exercise in data structure courses. Standard texts usually present complex and ugly algorithms. It has been the aim of this paper to show that balanced trees can be implemented more easily by abstracting away from the rebalancing using 'smart constructors'. It is even possible to upgrade programs that use unbalanced trees, provided sufficient information is available at each node to determine the size or height of the tree.

https://doi.org/10.1017/S0956796800000885 Published online by Cambridge University Press

Standard texts, like Knuth or Aho *et al.*, describe only concrete manipulations on balanced trees. The small but important step of combining these operations to implement abstract set operations is usually omitted. We have rectified this common omission.

The size information is useful for determining the cardinality of a set and for rebalancing the tree. The size has applications outside sets, for example, in determining the rank of an element in an ordered collection or the *n*th element in an ordered collection (Knuth). Size-balanced binary trees are a particularly versatile data structure.

Acknowledgements

The original form of this program was written in response to Andrew Appel's challenge to the SML community to write a fast integer set package. Without the bait of a competition I doubt that I would have taken the time to finish the program. I would like to thank Andy Gravell, Richard Bird and Robert Harper for their comments on the presentation of this material.

References

Adams, S. R. (1992) An efficient functional implementatation of sets. Report CSTR 92-10, Electronics and Computer Science, University of Southampton.

Adel'son-Vel'skii, G. M. and Landis, Y. M. (1962) An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR* **146**, 263–266 (in Russian). English translation in *Soviet Math. Dokl.* **3**, 1962, 1259–1262.

Aho, A. V, Hopcroft, J. E. and Ullman, J. D. (1974) The Design and Analysis of Computer Algorithms. Addison-Wesley.

Crane, C. A. (1972) Linear lista and priotity queues as balanced binary trees. PhD Thesis, Stanford University.

Knuth, D. E. (1973) The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley.

Nievergelt, J. and Reingold, E. M. (1973) Binary search trees of bounded balance, *SIAM J. Computing* **2**(1).