Fast Mergeable Integer Maps*

Chris Okasaki†
Department of Computer Science
Columbia University
cdo@cs.columbia.edu

Andrew Gill
Semantic Designs
Austin, TX
ajgill@semdesigns.com

Abstract

Finite maps are ubiquitous in many applications, but perhaps nowhere more so than in compilers and other language processors. In these applications, three operations on finite maps dominate all others: *looking up* the value associated with a key, *inserting* a new binding, and *merging* two finite maps. Most implementations of finite maps in functional languages are based on balanced binary search trees, which perform well on the first two, but poorly on the third. We describe an implementation of finite maps with integer keys that performs well in practice on all three operations. This data structure is not new—indeed, it is thirty years old this year—but it deserves to be more widely known.

1 Introduction

Finite maps are the workhorse data structure in every compiler. Imperative implementations typically use hash tables, whereas functional implementations typically use balanced binary search trees. Both are good at lookups and insertions, which—not coincidentally—are the two most common operations in these applications. However, a third operation on finite maps, that of *merging* two maps, is nearly as important, and on this operation, both hash tables and balanced binary search trees fare rather poorly. We describe an alternative implementation of finite maps with integer keys that performs extremely well in practice on all three operations. Our design is based on a thirty-year-old data structure known as a *Patricia tree* [3], which deserves to be more widely known.

Our implementation satisfies the following signature:

```
signature DICT =
sig
  type Key
  type 'a Dict
  val empty  : 'a Dict
  val lookup : Key * 'a Dict -> 'a option
  val insert : ('a * 'a -> 'a) -> Key * 'a * 'a Dict -> 'a Dict
  val merge  : ('a * 'a -> 'a) -> 'a Dict * 'a Dict -> 'a Dict
end
```

The first argument to `insert` and `merge` is a combining function that resolves collisions. It is applied whenever two bindings have the same key. Most of the time this function is either `fst` or `snd`, but other choices are possible.

*This paper is descended from material originally presented at the 1994 Glasgow Functional Programming Workshop under the title "A balancing algorithm that grows on trees". However, no paper was produced at that time.

†Supported during the summer of 1998 by the Department of Computing Science at the University of Glasgow, with funds from the Scottish Higher Education Development Council.

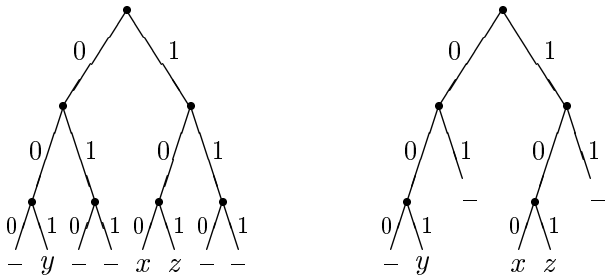

Figure 1: Two 3-bit binary tries for the map $\{1 \mapsto x, 4 \mapsto y, 5 \mapsto z\}$, (a) with empty subtrees included, and (b) with empty subtrees removed. Note that the bits are processed from least-significant to most-significant.

We restrict our attention to integer keys, which is reflected in Standard ML by specializing the signature using a `where` clause.

```
signature INTDICT = DICT where type Key = int
```

The restriction to integer keys is not a problem in practice. Compiler-generated identifiers are typically integers anyway, and it is usually a small matter to convert all source-level identifiers into integers during lexing.

We begin by gradually developing Patricia trees from a simpler data structure known as a *binary trie*. Then we reflect on the tradeoffs of using a big-endian binary encoding instead of a little-endian encoding. Finally, we report some preliminary benchmarks and conclude.

2 Binary Tries

A *binary trie* is a binary tree in which the placement of each key is controlled by its bits—each 0 means "go left at the next node" and each 1 means "go right at the next node". This is similar to a Braun tree [2, 4], except that in a Braun tree, data is stored at every node, whereas in a binary trie, data is stored only at the leaves.

At its simplest, a binary trie is a complete binary tree of depth equal to the number of bits in the keys, where each leaf is either empty, indicating that the corresponding key is unbound, or full, in which case it contains the data to which the corresponding key is bound. This style of trie might be represented in Standard ML as

```
datatype 'a Dict =
    Empty
  | Lf of 'a
  | Br of 'a Dict * 'a Dict
```

To lookup a value in a binary trie, we simply read the bits of the key, going left or right as directed, until we reach a leaf.

```
fun lookup (k, Empty) = NONE
  | lookup (k, Lf x) = SOME x
  | lookup (k, Br (t0,t1)) =
      if even k then lookup (k div 2, t0)
                else lookup (k div 2, t1)
```

Figure 1(a) illustrates a binary trie holding the numbers 1, 4, and 5. Notice the placeholders for the missing numbers. The numbering might seem counter-intuitive because we process the bits

in *little-endian* order (i.e., from least-significant to most-significant). We will consider *big-endian* tries in Section 5.

Although this data structure is straightforward to create and index, it is hopelessly inefficient since the trees become huge even for keys with only a moderate number of bits. An obvious optimization is to collapse each empty subtree into a single `Empty` node. Figure 1(b) shows the effect of this optimization on the example trie from Figure 1(a). This representation can be enforced by constructing trees, not with the `Br` constructor directly, but rather with a helper function called a *smart constructor* [1].

```
fun br (Empty,Empty) = Empty
  | br (t0,t1) = Br (t0,t1)
```

The `lookup` function given above still works for this new representation.

Next, consider a map containing only a single binding. Such a map is represented as a long chain of `Br` nodes ending in a single `Lf`, where the other child of each of the `Br` nodes is `Empty`. A successful lookup in such a tree tests the bits of the query key one at a time. But, since there is only one bit pattern that can ultimately be successful, surely it would be more efficient to test the bits all at once! We accomplish this by collapsing each subtree containing only a single binding into a single `Lf` node, and storing the desired key in that node. Then we change the `lookup` function to check this key whenever a `Lf` node is found.

```
  | lookup (k, Lf (j,x)) = if j=k then SOME x else NONE
```

Note that the key we store in the `Lf` node is not the *actual* key, but only those bits that are not determined by the ancestors of the node. In other words, the key stored in a `Lf` node at depth d is the actual key divided by 2^d. This compensates for the way that `lookup` discards bits on the way down the tree (by dividing by 2). The smart constructor must add the discarded bits back into the key whenever it moves a `Lf` node upwards.

```
  | br (Lf (j,x),Empty) = Lf (2*j,   x)
  | br (Empty,Lf (j,x)) = Lf (2*j+1, x)
```

A reasonable alternative to the above scheme is for each `Lf` node to store its actual key, and for `lookup` to *not* discard bits of the key. This design is simplified if we store at each `Br` node the bit that is being branched on. We store this information as an integer with a single one bit (i.e., a power of 2), with the invariant that the power of 2 equals the depth of the node in the tree. The code then looks like

```
datatype 'a Dict =
    Empty
  | Lf of int * 'a
  | Br of int * 'a Dict * 'a Dict

fun lookup (k, Empty) = NONE
  | lookup (k, Lf (j,x)) = if j=k then SOME x else NONE
  | lookup (k, Br (m,t0,t1)) =
      if zeroBit (k,m) then lookup (k, t0)
                       else lookup (k, t1)

fun br (m,Empty,Empty) = Empty
  | br (m,Empty,t as Lf _) = t
  | br (m,t as Lf _,Empty) = t
  | br (m,t0,t1) = Br (m,t0,t1)
```

where `zeroBit` tests whether the desired bit is 0.

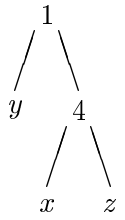

Figure 2: A Patricia tree for the map $\{1 \mapsto x, 4 \mapsto y, 5 \mapsto z\}$. The numbers in the branch nodes are the branching bits. The root has prefix 0 and the other branch node has prefix 1.

```
fun zeroBit (k,m) = (andb (k,m) = 0)
```

Here, `andb` is the bitwise-and of two integers.[1]

3 Patricia Trees

There is still one embarrassingly inefficient case remaining, which occurs when all the keys in a map of size two or more share a long common prefix of bits. In that case, the tree will begin with a long chain of "useless" `Br` nodes, each of which has an empty child, before finally reaching a "useful" `Br` node, which has two non-empty children. Just as before, a successful lookup on such a map will test the bits of the query key one at a time. But, again, since there is only one prefix that will successfully reach the useful node, surely it would be more efficient to test the bits of that prefix all at once! We accomplish this by eliminating all useless `Br` nodes and storing at each remaining `Br` node the longest common prefix of all keys in that subtree. Figure 2 shows the effect of this optimization on the trie from Figure 1.

We change the representation of `Br` nodes to

```
  | Br of int * int * 'a Dict * 'a Dict
```

where the first integer is the prefix and the second integer is the branching bit. Note that only the bits of the prefix below (i.e., less significant than) the branching bit are valid. We assume the invalid bits have been zeroed out.

The `Br` case of the `lookup` function then becomes

```
  | lookup (k, Br (p,m,t0,t1)) =
      if not (matchPrefix (k,p,m)) then NONE
      else if zeroBit (k,m) then lookup (k, t0)
                            else lookup (k, t1)
```

where

```
fun mask (k,m) = andb (k,m-1)
fun matchPrefix (k,p,m) = (mask (k,m) = p)
```

If most queries are expected to be successful, then it is slightly more efficient to eliminate the `matchPrefix` test.

```
  | lookup (k, Br (p,m,t0,t1)) =
      if zeroBit (k,m) then lookup (k, t0)
                       else lookup (k, t1)
```

[1] Actually, the Standard ML Basis Library defines `andb` over words, not integers, but converting between the two is easy.

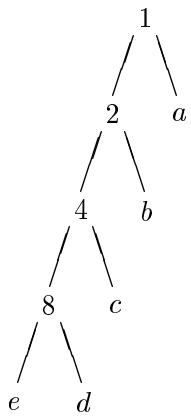

Figure 3: An unbalanced Patricia tree representing $\{1 \mapsto a, 2 \mapsto b, 4 \mapsto c, 8 \mapsto d, 16 \mapsto e\}$. The numbers in the branch nodes are the branching bits. All the branch nodes have prefix 0.

Bad keys will be detected when the search finally reaches a `Lf` node.

The `br` smart constructor now guarantees that no `Empty` node is ever a child of a `Br` node.

```
fun br (p,m,Empty,t) = t
  | br (p,m,t,Empty) = t
  | br (p,m,t0,t1) = Br (p,m,t0,t1)
```

This final design is called a *Patricia tree* [3]. It uses n `Lf` nodes and $n-1$ `Br` nodes to represent a finite map of size $n > 0$. The worst-case running time of `lookup` is $O(\min(n, W))$, where W is the word size of the `int` type. It is possible for these trees to get very unbalanced—for example, Figure 3 shows a tree with keys 1,2,4,8,16—but this worst case is both much less likely and much less devastating in practice than the worst case for, say, unbalanced binary search trees, which reach a depth of $O(n)$ (which can be much worse than $O(W)$!) for the extremely common case of ordered insertions. In the common case that keys are chosen contiguously, Patricia trees are perfectly balanced, regardless of the order of insertions.[2]

4 Insertions and Merges

So far we have described only one of the three major operations on finite maps. Now that we have the representation finalized, we turn our attention to the other two: insertions and merges.

First, we explore the case that gives these operations their speed. Suppose we have two non-empty trees t_0 and t_1, with longest common prefixes p_0 and p_1, respectively. Further, suppose that p_0 and p_1 disagree, that is, neither prefix is contained in the other. Then, no matter how large t_0 and t_1 are, we can merge them simply by creating a new `Br` node that has t_0 and t_1 as children! This is accomplished by the auxiliary function `join`.

```
fun join (p0,t0,p1,t1) =
  let val m = branchingBit (p0,p1)
  in if zeroBit (p0,m) then Br (mask (p0,m), m, t0, t1)
                       else Br (mask (p0,m), m, t1, t0)
  end
```

where `branchingBit` finds the first bit at which p_0 and p_1 disagree. Note that the first bit at which p_0 and p_1 disagree is the lowest one bit in their bitwise exclusive-or, i.e.,

[2] In fact, these trees are always insensitive to the order of insertions.

```
fun insert c (k,x,t) =
  let fun ins Empty = Lf (k,x)
        | ins (t as Lf (j,y)) =
            if j=k then Lf (k,c (x,y))
            else join (k,Lf (k,x),j,t)
        | ins (t as Br (p,m,t0,t1)) =
            if matchPrefix (k,p,m) then
              if zeroBit (k,m) then Br (p,m,ins t0,t1)
                               else Br (p,m,t0,ins t1)
            else join (k,Lf (k,x),p,t)
  in ins t end
```

Figure 4: The `insert` function.

```
fun branchingBit (p0,p1) = lowestBit (xorb (p0,p1))
```

Now, `lowestBit` could be calculated with a loop, as in

```
fun lowestBit x = if odd x then 1 else 2 * lowestBit (x div 2)
```

but a faster solution takes advantage of a curious property of the twos-complement representation of integers.

```
fun lowestBit x = andb (x, ~x)
```

Recall that in twos-complement representations, `~x` equals `notb x + 1`. Now, suppose the (little-endian) binary representation of `x` is

$$000001bbbbb$$

Then, the bitwise negation of `x` is

$$111110\overline{bbbbb}$$

Adding one yields

$$000001\overline{bbbbb}$$

Finally, taking the bitwise-and with `x` yields

$$00000100000$$

which is the desired answer.

With these functions in hand, we can now write `insert` and `merge`, which are shown in Figure 4 and Figure 5. There are many cases to consider, but each is straightforward. Note that in the case

```
        | mrg (t, Lf (k,x)) = insert (c o swap) (k,x,t)
```

the combining function `c` is modified to take its arguments in the opposite order, where

```
fun swap (x,y) = (y,x)
```

In the worst case, `insert` runs in $O(\min(n, W))$ time and `merge` runs in $O(n_0 + n_1)$ time, but in practice both are quite fast. (See Section 6.)

Many other finite map and set operations on these trees are equally fast and easy to implement, such as `delete`, `intersection`, and `difference`. (These are the functions that would typically use the `br` smart constructor to eliminate empty nodes.)

```
fun merge c (s,t) =
  let fun mrg (Empty, t) = t
        | mrg (t, Empty) = t
        | mrg (Lf (k,x), t) = insert c (k,x,t)
        | mrg (t, Lf (k,x)) = insert (c o swap) (k,x,t)
        | mrg (s as Br (p,m,s0,s1), t as Br (q,n,t0,t1)) =
            if m=n andalso p=q then
                (* The trees have the same prefix. Merge the subtrees. *)
                Br (p,m,mrg (s0,t0),mrg (s1,t1))
            else if m<n andalso matchPrefix (q,p,m) then
                (* q contains p.  Merge t with a subtree of s. *)
                if zeroBit (q,m) then Br (p,m,mrg (s0,t),s1)
                                 else Br (p,m,s0,mrg (s1,t))
            else if m>n andalso matchPrefix (p,q,n) then
                (* p contains q.  Merge s with a subtree of t. *)
                if zeroBit (p,n) then Br (q,n,mrg (s,t0),t1)
                                 else Br (q,n,t0,mrg (s,t1))
            else (* The prefixes disagree. *)
                join (p,s,q,t)
  in mrg (s,t) end
```

Figure 5: The `merge` function.

5 Big-endian Patricia Trees

Until now, we have been processing the bits of each key in *little-endian* order (i.e., from least-significant to most-significant). We could also process the bits in *big-endian* order (i.e., from most-significant to least-significant). The only changes required are to the `mask` and `branchingBit` functions.

The `mask` function now sets the indicated bit to zero and all the lesser bits to one.

```
fun mask (k,m) = andb (orb (k,m-1), notb m)
```

This is because the valid bits are now those above the indicated bit, rather than below it. We could simply set all the invalid bits to zero, but this more complicated version of `mask` has an interesting advantage. Provided we either exclude negative keys or use unsigned arithmetic, every key in the left subtree of a `Br` node is now less than or equal to the prefix in that node, and every key in the right subtree is greater. In other words, our trees can now masquerade as binary search trees! If desired, we could replace most of the `zeroBit` tests with comparisons, e.g., rewriting the `Br` case of `lookup` as

```
    | lookup (k, Br (p,m,t0,t1)) =
        if k <= p then lookup (k, t0)
                  else lookup (k, t1)
```

Unfortunately, the changes to the `branchingBit` function are more substantial. The problem is that there does not appear to be a clever "bit-twiddling" solution to calculate the highest one bit in a number, as there was for the lowest one bit. Instead, we find the highest one bit using a loop that searches upward from some initial guess. We can write this loop either as

```
fun highestBit (x,m) =
  let val x' = andb (x,notb (m-1))  (* zero all bits below m *)
      fun highb (x,m) =
        if x=m then m else highb (andb (x,notb m), 2*m)
  in highb (x',m) end
```

or as

```
fun highestBit (x,m) =
  let val x' = andb (x,notb (m-1))  (* zero all bits below m *)
      fun highb x =
        let val m = lowestBit x
        in if x=m then m else highb (x-m) end
  in highb x' end
```

The first version checks every bit between the initial guess and the answer, whereas the second version checks only the one bits between the initial guess and the answer.

The initial guess can be calculated from the lengths of the two prefixes. Recall that we only call `branchingBit` when two prefixes disagree. The bit at which the two prefixes disagree must be somewhere within the shorter prefix, so we can begin searching at the least-significant valid bit in the shorter prefix. If the most-significant *invalid* bits of p_0 and p_1 are m_0 and m_1, respectively, where m_i is the branching bit in the `Br` node for p_i (or 0 if p_i is actually a key from a `Lf` node), then the initial guess should be

$$\max\left(1, 2 \cdot \max\left(m_0, m_1\right)\right)$$

Thus, `branchingBit` can be implemented as

```
fun branchingBit (p0,m0,p1,m1) =
      highestBit (xorb (p0,p1), max (1, 2 * max (m0,m1)))
```

The `join` function and the calls to `join` must also be modified to pass along m_0 and m_1.

Although the `branchingBit` computation is considerably less efficient, big-endian Patricia trees offer several significant advantages over little-endian Patricia trees.

- *Potentially cheaper branch tests.* As already noted, many `zeroBit` tests can be replaced by simple comparisons. For some compilers and machine architectures, the comparison will be a few instructions cheaper than the `zeroBit` test (although often the two will generate exactly the same instruction count).

- *Better locality.* Keys that are close together are likely to be placed close together in big-endian trees, whereas they are likely to be placed far apart in little-endian trees. This results in substantially better cache behavior for big-endian trees when keys are accessed sequentially.

- *Easy to process keys in order.* Because big-endian trees double as binary search trees, it is easy to iterate over the keys in order. This is quite difficult to do in little-endian trees.

- *Faster merges in common cases.* It is common in practice for a map to contain contiguous or nearly contiguous blocks of keys. Big-endian trees will tend to have large common prefixes for such blocks, whereas little-endian trees will tend to have small common prefixes. Large prefixes are good because they offer more opportunities for prefixes to disagree, which lets us use the fast `join` function during insertions and merges. For example, merging trees containing the keys $1 \ldots 5$ and $6 \ldots 9$ takes 8 steps using little-endian trees, but only 3 steps using big-endian trees. (Of course, it is easy to come up with examples where the reverse is true—just consider merging trees with the keys $2, 4, 8, 10, 12$ and $1, 6, 9, 14$—but these examples seem less likely to arise in practice.)

In practice, these advantages tend to outweigh the extra cost of the `branchingBit` computation, especially since `branchingBit` is called at most once per insertion. (It may be called more frequently during merges, however.)

time (sec)		Little	Big	Adams	RedBlack	Splay
lookup	seq	1.85	1.26	0.72	**0.70**	2.44
	rand	1.90	1.89	1.13	**1.04**	4.04
insert	seq	5.00	2.83	5.92	3.61	**0.25**
	rand	4.91	6.23	5.41	3.31	**3.21**
merge	seq	8.72	**1.58**	6.14	14.97	5.54
	rand	6.90	**5.73**	15.82	15.06	6.42

Table 1: Preliminary benchmark results. Experiments were run on a 266MHz Pentium II using SML/NJ v110.8 under Linux.

6 Preliminary Measurements

Table 1 summarizes the performance of several implementations of finite maps on benchmarks designed to stress lookups, inserts, and merges under both sequential and random access patterns. Besides little- and big-endian Patricia trees, we tested splay trees [5], red-black trees [6], and balanced binary search trees derived from Adams' implementation [1]. All the implementations were tuned to approximately the same degree.

The results are mixed. Splay trees are lightning fast on insertions (especially sequential insertions), and also perform quite well on merges. Unfortunately, splay trees are very slow on lookups—at least in SML/NJ—because SML/NJ severely penalizes assignments, which splay trees use for a kind of memoization. Another drawback to splay trees is that they are guaranteed to be efficient only when used in a single-threaded manner.

Red-black trees support the fastest lookups and very good insertions, but are terrible at merges.[3] Adams' balanced binary search trees also support fast lookups, but are rather slow on merges. Surprisingly, they are also rather slow on insertions.

Little-endian Patricia trees fare much worse than big-endian Patricia trees on all the sequential benchmarks, but hold their own on the random benchmarks. This is not surprising since sequential accesses are close to the worst case for little-endian trees but close to the best case for big-endian trees. A different set of benchmarks would paint the opposite picture, but sequential accesses seem much more likely in practice than the best cases for little-endian trees.

Big-endian trees perform extremely well on merges and on sequential insertions. They also provide reasonably fast lookups and random merges, but they fare rather poorly on random insertions, where the relatively expensive `highestBit` calculation is at its slowest.

Overall, the best choices for most applications are probably

- *red-black trees* for applications that use `lookup` and `insert` heavily, but `merge` only rarely (if at all), or
- *big-endian Patricia trees* for applications that also use `merge` heavily.

Splay trees would also be an excellent choice for single-threaded applications that do not lookup individual elements much, but rather process trees with bulk operations such as maps and folds.

[3] With a minor change in representation, it is possible to speed up the merges, but at the cost of slowing down the insertions.

7 Conclusions

We have presented an implementation of integer finite maps that supports fast merges. We expect this data structure to be widely useful in compilers and similar applications written in functional languages. In fact, the Glasgow Haskell compiler has used a similar implementation for several years.

References

[1] Adams, S. Efficient sets—a balancing act. *Journal of Functional Programming 3*, 4 (Oct. 1993), 553–561.

[2] Hoogerwoord, R. R. A logarithmic implementation of flexible arrays. In *Conference on Mathematics of Program Construction* (July 1992), vol. 669 of *LNCS*, Springer-Verlag, pp. 191–207.

[3] Morrison, D. R. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4 (Oct. 1968), 514–534.

[4] Okasaki, C. Three algorithms on Braun trees. *Journal of Functional Programming 7*, 6 (Nov. 1997), 661–666.

[5] Okasaki, C. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[6] Okasaki, C. Red-black trees in a functional setting. *Journal of Functional Programming* (1998). To appear.