Implementing Sets Efficiently in a Functional Language

Stephen Adams

CSTR 92-10

Department of Electronics and Computer Science

University of Southampton

Southampton S09 5NH

Abstract

Most texts describing data structures give imperative implementations. These are either difficult or tedious to convert to a functional (non side-effecting) form. This technical report describes the implementation of sets in the functional subset of Standard ML. The implementation is based on balanced binary trees. Tree balacing algorithms are usually complex. We show that this need not be the case—the trick is to abstract away from the rebalancing scheme to achieve a simple and efficient implementation. A complete implementation of sets is given, including the set$\times$set$\rightarrow$set operations union, difference and intersection. Finally, program transformation techniques are used to derive a more efficient union operation.

Contents

1 Introduction

Sets and finite maps (lookup tables) are fundamental data types. Many problems can be solved easily by applying these data types because the problem can be analysed into a small number of these mathematical types. Many specification languages use these concepts. The prevalence of these concepts underlines the important of having available data structures to implement sets and finite maps efficiently.

I make a distinction between data types and data structures. The data type, often referred to as the abstract data type, is a set of values and the operations provided on those values. In this paper I will be using a set of integers as the data type, with operations like union, intersection and testing for membership. The data structure is an implementation of the data type using some underlying data types, perhaps built-in types.

There can be many implementations of the abstract data type, and each different data structure used for the implementation will have different properties. These properties are reflected in the efficiency, in both time and space, of the abstract data type. Some data structures trade the efficiency of one operation against another. I have chosen balanced trees for the implementation because they provides a good all-round performance, with no operation being particularly inefficient.

Particular care has been taken to make the algorithms efficient, while trying to keep them clear. Program transformation techniques are used to derive a more efficient union operation by removing redundant operations.

2 Functional programming

Functional programming is a style of programming using functions with mathematical properties. One of the key properties of the style is that the same expression in the same context always has the same value. For example, the expression $\sin(x+1)$ always has the same value for a given value of x. This is called *referential transparency*. Functional programming languages usually enforce referential transparency by prohibiting side effects. Side effects include assigning values to variables and altering data structures. Functional programming does not use variables in the Pascal sense as a 'box' which contains a value which may be changed by assignment. Rather, variables are used in the mathematical sense to give a name to a value. Since the variable is the name of a value is no longer makes sense to assign to it. It only makes sense to use another value instead.

The practical benefit of referential transparency is that it eliminates a whole class of bugs that occur because something has mysteriously changed behind the programmer's back.

Without side effects, how is it possible to build and update data structures?

The quick and unhelpful answer is that it isn't. A better answer is that a similar effect is achieved by computing a new data structure that may be used in place of the old one. for example, suppose we have a database of telephone numbers:

```
phones = {("John",54987), ("Angie", 56221)}
```

Now `lookup(phones,"John")` should return 54987 and `lookup(phones,"Paul")` returns *error*. In a non-functional language we might add Paul to the phone list like this:

```
add(phones, "Paul", 23601)
```

In a functional language we would 'update' the phone list by computing a new phone list:

```
new_phones = update(phones, "Paul", 23601)
```

now

```
lookup(phones, "Paul")        ⟶ error
lookup(new_phones, "Paul")    ⟶ 23601
```

`phones` has not changed. It names the historical value of the database. This sequence of expressions is analogous to the simple arithmetic expressions:

```
x = 5
x            ⟶ 5
y = x+1
x            ⟶ 5
y            ⟶ 6
```

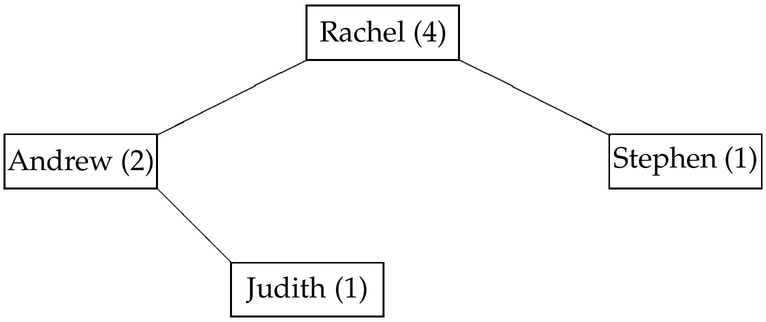
You would have been surprised if `x` had magically changed to 6, so why should you be any less surprised if the old phone number database changed when you added a new number?

3 Binary search tree data structure

A binary search tree is either empty or is a node with left and right subtrees and a data item of type `Element`. In addition we keep in each node a count of all the nodes in the tree rooted at that node. We declare such a structure in SML like this:

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

This declares `Tree` as a recursive type with two kinds of values. There are empty trees, written `E` and non-empty trees, written '`T(`*datum*`,`*count*`,`*left-subtree*`,`*right subtree*`)`'. `E` and `T` are constructor functions. This little tree of names



is represented as this tree data structure, where the element type is string:

```
T("Rachel", 4,
   T("Andrew", 2, E, T("Judith",1,E,E)),
   T("Stephen", 1, E, E))
```

The elements in the tree are ordered so that all of the elements stored in the left subtree of a node are less than the element stored in the node and all of the elements in the right subtree are greater. Duplicate elements are not allowed[1]. The ordering is supplied by a function `lt` from pairs of elements to booleans:

```
val lt : Element * Element -> bool
```

The ordering must be a total ordering, so that for all *a*, *b* and *c*

```
lt(a,b) and lt(b,c) implies lt(a,c)
not lt(a,b)  and  not lt(b,a)  implies  a=b
```

I use the function `lt` rather than the infix < operator only because < is a builtin overloaded operator which cannot be redefined for an arbitrary `Element` type. We only need one operation because all six relational operators can be expressed in terms of `lt`:

$a > b$	`lt(b,a)`
$a \geq b$	`not(lt(a,b))`
$a = b$	`not(lt(a,b)) andalso not(lt(b,a))`
etc	

In some parts of this paper I have written `a<b` or `a>b` where `lt(a,b)` or `lt(b,a)` appears in the real program because it looks better, and it still works for the builtin types `int`, `real` and `string`.

[1]If duplicates are required then the tree can be used to do this by storing a count of the duplicates at the node as well as the value.

4 Pattern matching and some simple tree operations

We use pattern matching against the tree constructors to extract information. A simple example of this is the `size` function which returns the number or elements in the tree. If the tree is empty then it contains no elements, otherwise this information is stored in the root node. Pattern matching is used to distinguish between the two cases and to identify subparts of the data structure that are of interest:

```
fun size E                  = 0
  | size (T(_,count,_,_)) = count
```

Note that the following invariant holds for the size of a tree. For a node n = `T(_,count,left,right)`

size n = count = 1 + size left + size right

We can ensure that this is always the case by using a *smart constructor*[2] `N` in the place of `T` which calculates the size for the new node:

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

The `member` function tests whether an element `x` is in the tree. The ordering of the elements is used to direct the search. From the element stored in the current node we can tell if `x` is in the left or right subtree.

```
fun member (x, E)   = false
  | member (x, T(v,_,left,right)) =
        if x<v then member(x,left)
        else if x>v then member(x,right)
        else true
```

Note that `true` is returned in the last else as `x` must be equal to `v` by the properties of the total ordering <.

Another example of pattern matching is the `min` function which returns the minimum value in the tree. The minimum value in a binary search tree is found by following the left subtrees until we reach a node that has an empty left subtree. The value at that node must be the minimum because all the values in the right subtree are greater than that value.

```
fun min (T(v,_,E,_))     = v
  | min (T(v,_,left,_)) = min left
  | min E                = raise Match
```

[2] I use the term *smart constructor* because the programmer would normally use `N` in place of the native constructor `T`. The distinction between a smart constructor and an ordinary function, which constructs a solution to a problem, is *level of abstraction*.

The last case signals an error if `min` is applie to an empty tree. If the case had been omitted then the SML system would have inserted it and given us a warning that `min` does not match all trees. Writing the case explicitly avoids the warning and, more importantly, signals to someone reading the program that `min` is not intended to work for empty trees.

5 Bounded Balance Trees

A binary search tree is f-balanced if

1. Its subtrees are f-balanced
2. The left and right subtrees satisfy some balancing constraint f.

If a binary search tree is not balanced then it may become degenerate, long and thin, resembling a list more than a tree. If, for example, nearly every left subtree is empty, searches (like the `member` function) might have to inspect nearly all of the elements. If the tree is short and bushy then choosing to go left or right avoids inspecting the large number of elements in the other branch. Balancing a tree keeps it relatively short and bushy. Trees are usually balanced to guarantee that a search takes $O(\log n)$ operations, where n is the size of the tree.

Maintaining a balanced tree has two costs:

1. The tree has to be rebalanced after or during operations that add or remove elements from the tree.
2. If rebalancing is to be efficient it must be done using information local to a node. This requires that some information is stored at each node.

In a high level language this additional information requires an extra storage location per node. It makes sense to store additional information that is useful in implementing other operations. Bounded Balance trees use the size information to keep the tree balanced. The idea is to ensure that one subtree does not get substantially bigger than the other. The balancing constraint used in this paper is that one subtree is never more than a constant factor w larger than the other subtree. Other balancing schemes, like height-balanced trees, AVL trees [2] and Red-Black[3] trees store balancing information that is otherwise not very useful.

[3] For AVL trees and Red-Black trees the balancing information requires only a couple of bits of storage. It is often suggested that storage may be saved by hiding these bits in the pointer values used to represent the left and right subtrees. This kind of storage allocation requires knowledge of the representation of pointers in the machine so it is inherently non-portable. In high level languages like SML the pointers are hidden completely from the programmer so this technique is not available.

Nievergelt and Reingold [4] use a slightly different criterion that compares the size of a subtree to the size of the whole tree. For their purposes, which included producing analytical results on the behaviour of the tree, this criterion is cleaner. For our purposes it is simpler to compare the sizes of the left and right trees.

We abstract away from the balancing scheme we use. To do this it is necessary to be able to test if two subtrees would make a balanced tree by looking at the subtrees alone. This is possible if there is a cheap absolute measure of the tree that can be used to compare the trees, but not possible if this information is relative to some other part or point in the computation. AVL trees are an example of this because each node keeps the the height *difference* between the two subtrees. Taken in isolation it is not possible to tell immediately if the two subtrees would make a balanced node.

6 Balancing maintenance algorithms

We will always be considering balance maintenance with the goal of preventing an unbalanced tree from being created. Smart constructors are used which build a balanced tree. There is a hierarchy of constructors which construct balanced trees from progressively out-of-balance subtrees.

1. `T(v,n,l,r)`
   This is the data type constructor.

2. `N(v,l,r)`
   This is the smart constructor that ensures that the size of the tree is maintained correctly. The tree `(v,l,r)` must already be balanced.

3. `T'(v,l,r)`
   `T'` is used when the original tree was in balance and one of `l` or `r` has changed in size by at most one element, as in insertion or deletion of a single element.

4. `concat3(v,l,r)`
   `concat3` can join trees of arbitrary sizes. As with the other constructors, `v` must be greater than every element in `l` and less than every element in `r`.

The `T'` constructor works by comparing the sizes of the left and right subtrees. If the sizes of these trees are sufficiently close then a tree is constructed using the `N` constructor.

If one subtree is too heavy, (has a size greater than w times the other) then the tree that is built is a *rotation* of the tree that would be built by `N`. The idea of a rotation is to shift part of the heavy subtree over to the side of the lighter subtree. There are two kinds of rotation: single rotations and

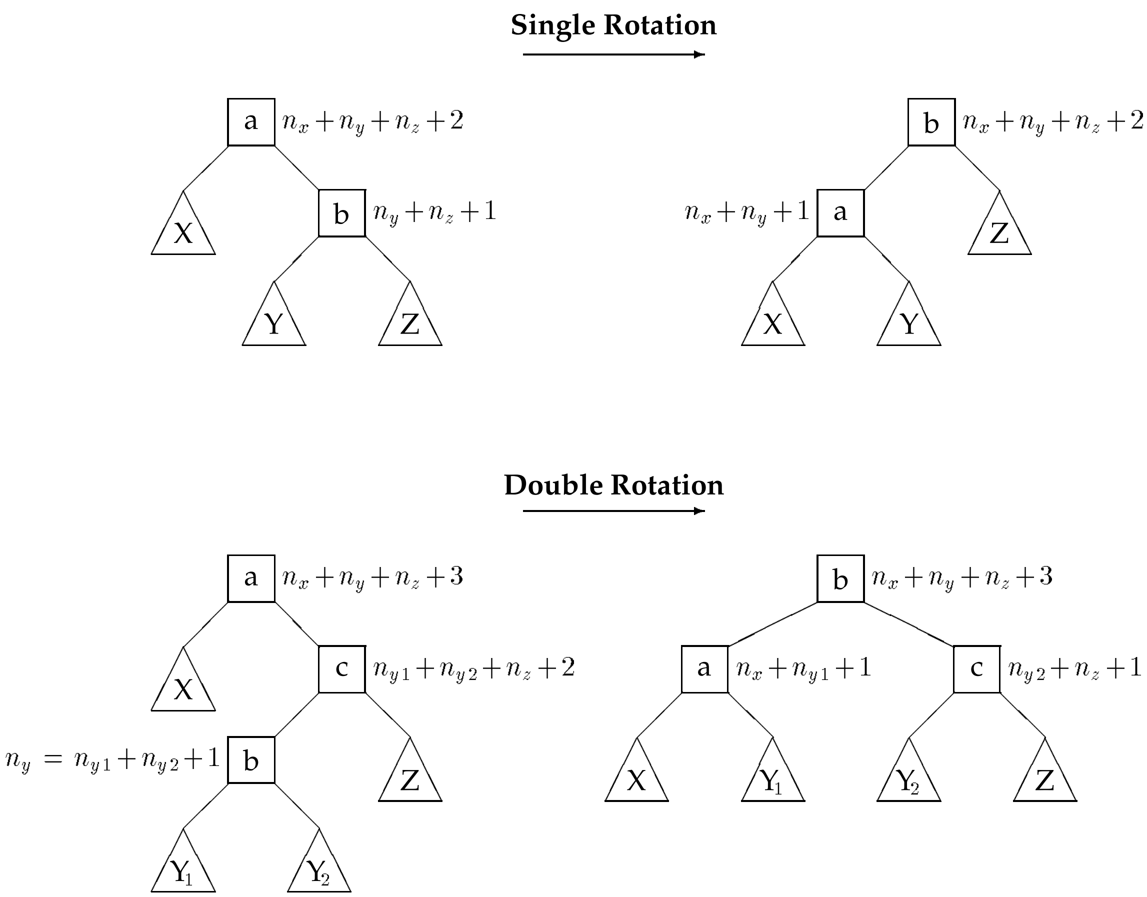


Fig. 1: Single and Double Left-Rotations. a<b<c are tree elements; X, Y and Z are trees of size n_x, n_y and n_z respectively.

double rotations. These are illustrated in figure 1. Each rotation has two symmetrical variants. The rotations are expressed very succinctly in SML:

```
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))
```

The difference between a single rotation and a double rotation is that a single (left) rotation moves the entire left subtree of the right side over to the left side whereas the double rotation only moves the left part of the left subtree.

```
fun T' (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>weight*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>weight*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
```

Fig. 2: The implementation of `T'`

The difficult part of implementing `T'` is deciding on what rotations, if any, to use. A careful analysis of the possible arguments leads directly to the solution given in figure 2.

1. In the trivial case where the left and right trees are empty the tree will be balanced, so `T'` should call `N`.

2. If one subtree is empty and the other contains only one element the tree cannot be balanced, so `N` is called.

   Both of these last cases can be tested in one test: `ln+rn < 2`. As `N` takes the same type of parameter as `T'` I have used a layered pattern, '`p as (v,l,r)`' at the top of `T'`. SML functions take exactly one argument. Often this is a tuple of values, like `(v,l,r)`. Layered patterns allow the whole pattern to be named as well as its subparts. In `T'`, `p` is bound to the tuple of parameters[4]. `N` can be called directly with the same parameters like this: `N p`. The same idea is used later with the rotation functions.

3. If neither tree is too heavy then `N` is called. This case is determined in figure 2 by falling through to the final else clause.

4. If the right tree is too heavy (has more than w times the number of elements than the left tree) then a balanced tree must be constructed by moving part of the right tree to the left side. If the outer subtree on the right side is smaller that its sibling then a single rotation might leave the tree unbalanced. So we do a double rotation if the inner right subtree is the larger and a single otherwise.

   For some values of w is is not possible to build a balanced tree, for example $w = 1$ implies that the tree is always perfectly balanced, which is impossible.[5] The choice of a value for w is discussed in Appendix A. Here it is sufficient to say that $w > 3.75$ ensures that it is possible to create a balanced tree. It is convenient to use integral values for w, so w is chosen such that $w \geq 4$.

5. The case for a heavy left tree is symmetrical to the right tree.

We will return to discuss `concat3` in Section 9.

7 Insertion and deletion

The conventional terms 'insertion' and 'deletion' are a bit of a misnomer in functional programming, since referential transparency prevents us from altering the tree. The functional analogue to the imperative concept of

[4] That is why I called it p

[5] Execpt, of course, for trees containing exactly $2^n - 1$ elements for some n.

insertion is to return a new data structure that looks like old one except that it also contains the 'inserted' element.

Insertion works by finding a place on the fringe of the tree to put the new element. The place is determined by the ordering of the elements. The new element must be placed between the highest value less than it and the lowest value greater than it. This position is found by searching for the new element, then a new tree is constructed which has the new element in this place.

This is the implementation of `add`. The function application `add(aTree,x)` returns a new tree which contains `x`:

```
fun add (E,x) = T(x,1,E,E)
  | add (tree as T(v,_,l,r),x) =
    if lt(x,v) then T'(v,add(l,x),r)
    else if lt(v,x) then T'(v,l,add(r,x))
         else tree
```

The first case handles an empty tree and returns a tree containing one element. This case is used as a base case for the recursion in the second case. The second case navigates through the tree much like `member`, deciding to add the new element to either the left subtree or the right subtree. If, in the final analysis, the tree already has the new element at the root then there is no point in inserting it, to the original tree which already contains `x` is returned. It is more useful to change the last line to:

```
         else N(x,l,r)
```

This is more useful for when the tree contains elements which have information that is not used in the `lt` comparison, for example database records that are indexed on only one field. With this change, adding an element that already occurs in the tree but has different associated data has the effect of 'updating' the element's associated data. However, it does cost a little extra in storage and time to build the new node using `N` rather than reusing the old one.

It is interesting to note that the algorithm is the same as for inserting in an unbalanced tree, except we use the rebalancing constructor `T'` is used rather than the plain one `N`.

Deletion, or rather, computing a tree which looks like the old one but has a selected element missing, is a little harder. The basic idea is to navigate down the tree to find the element to delete. If the tree is empty then the job is trivial—there is nothing to delete. If the element is to the left or right then it is deleted from that subtree and the tree is rebuilt using the rebalancing constructor. The difficulty comes when the element is at the root of the current subtree, so that case is left to an auxiliary function, `delete'`:

```
fun delete (E,x) = E
```

```
  | delete (T(v,_,l,r),x) =
    if lt(x,v) then T'(v,delete(l,x),r)
       else if lt(v,x) then T'(v,l,delete(r,x))
            else delete'(l,r)
```

The auxiliary function `delete'` must build a tree containing all the elements of `l` and `r`. This could be done easily if there was an element of the right value available to use because we could then join `l` and `r` using `N`. This element is found by taking the minimum element from `r`. Note that the rebalancing constructor `T'` is used rather than `N` because `delete(r,min_elt)` is one element smaller than `r`, and so might not balance with `l`. There are also two special cases for when either `l` or `r` is empty. These cases can simply return the other tree.

```
and delete' (E,r) = r
  | delete' (l,E) = l
  | delete' (l,r) = let val min_elt = min r
                    in
                        T'(min_elt,l,delete(r,min_elt))
                    end
```

The efficiency of this code can be improved. We know that `min_elt` is the minimum element in `r`, so

- `min_elt` must be in `r`, so `delete(r,min_elt)` will never use the first case of `delete`.
- `min_elt` must be at the leftmost node in the tree `r`.

It is considerably more efficient to use a specialized function since no comparison is necessary. The improved algorithm is given below, with another auxiliary function which deletes the minimum element in a tree. This function, `delmin`, is structured like `min` and is useful in its own right, for example, in implementing priority queues. Since `min_elt` is only used in one place we dispense with the `let`.

```
and delete' (E,r) = r
  | delete' (l,E) = l
  | delete' (l,r) = T'(min r,l,delmin r)

and delmin (T(_,_,E,r)) = r
  | delmin (T(v,_,l,r)) = T'(v,delmin l,r)
  | delmin _ = raise Match
```

Again it is worth noting that this deletion algorithm is the same as for unbalanced trees except that `T'` is used to ensure balance.

8 Processing a tree

Insertion, deletion and membership tests (including lookup) allow trees to be used in a variety of applications, for example as a database. Sometimes it is useful to process all the elements in a tree, for example, to print them or sum them.

There are many ways of processing the elements in a tree. They may be processed from left to right or from right to left. The element at a node may be processed before, in-between or after processing the element in the subtrees—giving pre-order, in-order and post-order traversals respectively.

It is a good idea to capture all of this detail in a function. The SML standard environment and provides a function called `fold` for combining the elements of a list to build a result.[6] We copy this idea and present `inorder_fold`, which combines the elements from right-to-left and in-order:

```
fun inorder_fold(f,base,tree) =
  let fun fold'(base,E) = base
        | fold'(base,T(v,_,l,r)) = fold'(f(v,fold'(base,r)),l)
in
  fold'(base,tree)
end
```

Other traversals can be built by changing the order of the calls to `f` and `fold'`. This traversal is in-order because the call to `f` is between the two calls to `fold'`, and it is right to left because `r` is innermost, being processed before `v` and then `l`.

One simple use of `inorder_fold` is to make a list of all the elements in the tree.

```
fun members tree = inorder_fold(op::, [], tree)
```

This function works by starting with an empty list (`[]`) and working through the tree adding the elements on to the front of the list with the list constructor operator `::`. As `inorder_fold` performs a right-to-left in-order traversal the elements in the list are in order according to the ordering relation <. A tree-sort can be constructed by converting the list into a tree and back again:

```
fun reverse_add (element,tree) = add(tree,element)
fun tree_from_list aList = fold reverse_add aList E
fun treesort aList = members (tree_from_list aList)
```

[6] The list `fold` function in NJ-SML's standard environment has the type

```
val fold : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

The sort takes time $O(n \log n)$ which is, within the constant factor, as fast as is possible using only an ordering relation [1]. Note that `reverse_add` is necessary because the arguments for `add` are in the wrong order for the argument `f` of `inorder_fold`.

A interesting decision in the design of `inorder_fold` is to make it have the type

```
val inorder_fold : (Element * 'a -> 'a) * 'a * Tree -> 'a
```

Usually a fold-like function is written to be isomorphic to the structure which is is applied to:

```
val treefold : (Element * 'a * 'a -> 'a) * 'a * Tree -> 'a

fun treefold (f,base,E) = base
  | treefold (f,base,T(v,_,l,r)) =
      f(v, treefold(f,base,l), treefold(f,base,r))
```

This function can be thought of as substituting `base` for every empty tree and `f` for every node. `f` takes an element and the results of the computations from both subtrees. This is analogous to the list `fold` which substitutes a function for every cons cell and a value for the single empty list `[]` at the end.

If fold-like functions are isomorphic to the data structure then each fold-like function takes parameters with different types. When the data type is being used as a set or table this becomes poor for abstraction as the user of the data type has to know how it is implemented, e.g. as a list or a binary tree or some other structure[7]. As all the programmer wants is to collect all the elements together in a computation it makes sense to make several fold functions like `inorder_fold` which all take the same kinds of arguments.

9 Set×Set→Set operations

In this section we develop efficient functions for combining two trees. The operations are union, intersection and difference.

The union operation ($A \cup B$) is discussed in most detail as an exemplar of the principles involved. Asymmetric set difference ($A - B$) is described as its implementation has some important differences to union. Intersection is trivial since it can be defined in terms of difference, like this:

$$A \cap B = A - (A - B)$$

However, as it is more efficient to code intersection directly, we do this too. We already have a method for processing the elements stored in trees. We

[7]Things can get even worse—both binary trees and lists have only two constructors. Other data structures have more, requiring more functions as parameters—for example 2-3-4 trees.

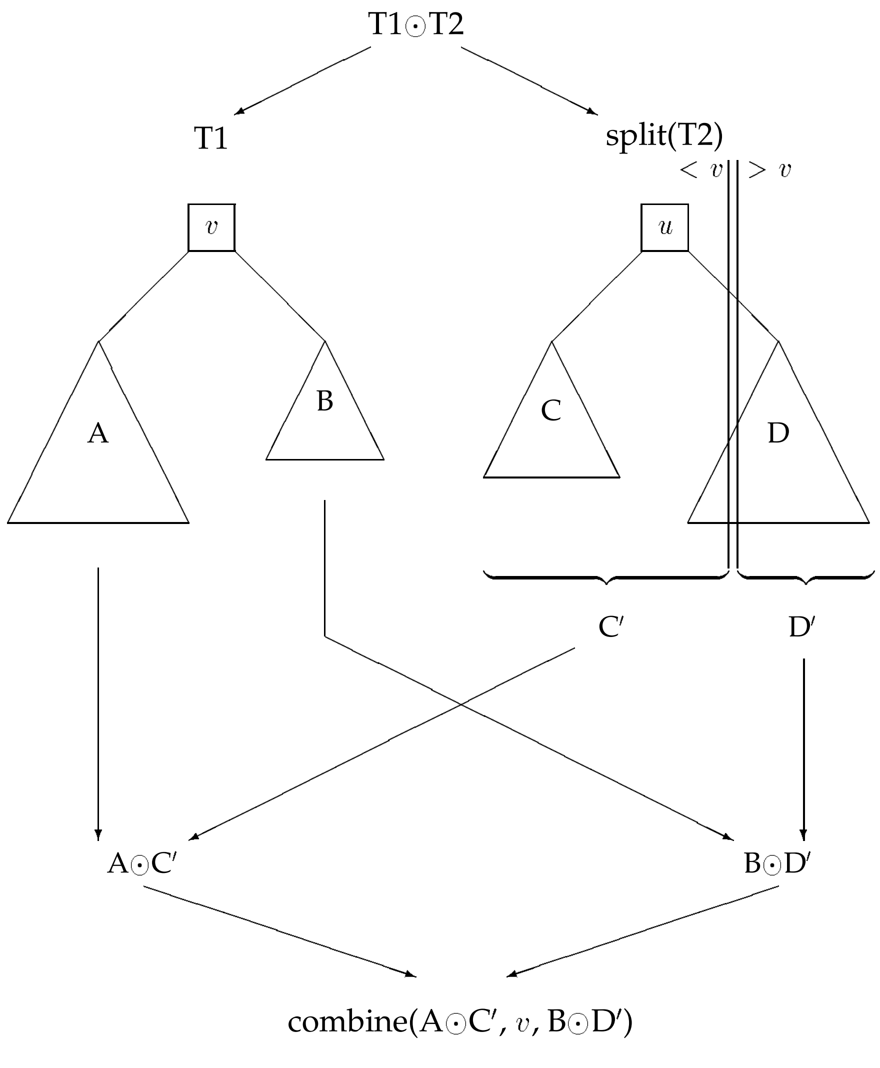


Fig. 3: Divide and conquer scheme for T1⊙T2.

could use `inorder_fold` to work through the second tree adding all of its elements to the first tree. The result is a new tree containing all of the element of both trees:

```
fun fold_union t1 t2 = inorder_fold(reverse_add,t1,t2)
```

This function is very elegant but not very efficient. It takes time $O(m\log(m+n))$ where n and m are the number of elements in `t1` and `t2` respectively. The problem is that each element of `t2` is inserted in a big tree, incurring the $O(\log(m+n))$ cost. The key to a more efficient version is to keep the trees small when inserting and to combine large trees efficiently. Fortunately it is possible to combine large trees very efficiently under certain conditions.

9.1 Divide and conquer

The efficiency of the operation depends on the size of the operands. Reducing the size of the operands increases efficiency. One strategy for reducing the size of the operands is *Divide and Conquer*.

Divide and Conquer works by breaking a big problem into small problems, solving the small problems, and then combining the solutions to the small problems to get the solution to the big problem. The small problems are usually just smaller versions of the big problem and are solved in the same way. Eventually the small problems must be so small that there is another easy way to solve them, so the dividing terminates.

Operations involving a single tree have an obvious structure on which to base the division—two subtrees[8]. The strategy for operations with two trees is to pick one tree to guide the control and to force the other tree into the right shape by cutting it into pieces and building two trees that are suitable to accompany the subtrees of the control tree. This is illustrated in figure 3.

9.2 Union

This is how `union` is implemented using the divide and conquer strategy:

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

The first two cases handle the trivial problem of combining a tree with an empty tree. The third case implements the divide and conquer steps.

[8] `treefold` is a simple divide and conquer algorithm.

- `split_lt` and `split_gt` return a tree containing all those elements in the original tree which are less than (or greater than) the cut element v. `l'` and `r'` corresponds to C′ and D′ in figure 3.
- `concat3` joins to trees using an element. It is the final member of the smart constructor hierarchy described in section 6.
- The *second* parameter is used as the control tree. The reason for this is that if both trees contain the same element then the final tree has the copy that originated from the control tree, matched against `v`. When a tree is used as a map `union` behaves like the map override operator $\oplus$[9]. This reason is the same as for the choice of the last case for the `add` on page 11, except that it costs no more than using the first parameter to guide the control. When the tree is to implement a set this does not matter.
- `split_lt` and `split_gt` can be implemented to take time $O(\log n)$ in the size of the tree.
- `concat(v,l,r)` can be implemented to take time $O(\log n - \log m)$ where the larger of `l` and `r` has size n and the smaller has size m.
- The entire `union` operation takes worst case time $O(n + m)$ where n and m are the sizes of the trees being combined.
- In cases where one tree is small or the trees have dense regions that do not overlap the operation is considerably faster.

The function `concat3` is used to join two trees with an element that is between the values in the left tree and the values in the right tree. If the left and right arguments would make a balanced tree then they can be joined immediately. If one tree is significantly larger then it is scanned to find the largest subtree on the side 'facing' the smaller tree that is small enough to balance with the smaller tree. The tree is joined at this position and the higher levels are rebalanced if necessary.

```
fun concat3 (v,E,r) = add(r,v)
  | concat3 (v,l,E) = add(l,v)
  | concat3 (v, l as T(v1,n1,l1,r1), r as T(v2,n2,l2,r2)) =
    if weight*n1 < n2 then T'(v2,concat3(v,l,l2),r2)
    else if weight*n2 < n1 then T'(v1,l1,concat3(v,r1,r))
         else N(v,l,r)
```

[9] defined as

$$\begin{aligned}(f \oplus g)(x) &= g(x) && \text{if } g(x) \text{ is defined}\\ &= f(x) && \text{otherwise}\end{aligned}$$

When the trees are nearly the same size, then concat3 does very little work.

A tree is split by discarding all the unwanted elements and subtrees, and joining together all the wanted parts using concat3.

```
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
```

split_lt takes time $O(\log n)$. At first it might be expected to take time $O(\log^2 n)$ as each of the $O(\log n)$ recursive calls might call concat3 which might take logarithmic time itself. This does not happen because the order of the calls to concat3 ensures that the small trees are joined together before being joined to the larger trees. So concat3 is usually called with comparably sized trees, and if concat3 is called with trees of greatly differing size this is compensated by the fact that it is called less often.

The running time of union is at worst $O(n + m)$. The result follows from the observation that at each node the operation takes time logarithmic in the size of the tree at that node. As the number of nodes increases exponentially with the logarithm of the size of the tree, this is the dominant factor, so the logarithmic operation of split_lt (and in some circumstances concat3) does not affect the order of the computation. There are so many more computations on small trees that are cheaper that is the dominant factor.

The running time of union is better for fortuitous inputs, for example, similar sized disjoint ranges or trees which differ greatly in size.

9.3 Difference and intersection

Asymmetric set difference also uses the divide and conquer strategy to achieve a linear time behaviour. The main difference compared with union is that difference must exclude certain values from the result. This is achived by making the second argument guide the control flow. Each element in the second argument is excluded because it does not appear in the split parts of the first tree and it is not included in final expression. This requires a function, concat, to concatenate two trees.

```
fun difference (E,s)  = E
  | difference (s,E)  = s
  | difference (s, T(v,_,l,r)) =
    let val l' = split_lt(s,v)
        val r' = split_gt(s,v)
    in
        concat(difference(l',l),difference(r',r))
    end
```

`concat` is easily coded in terms of `concat3`. As `concat` does not have the benefit of a 'glue element' one may be obtained by removing it from one of the parameters. This is much like how `delete'` worked:

```
fun concat (t1, E)  = t1
  | concat (t1, t2) = concat3(min t2, t1, delmin t2)
```

A slight improvement is to postpone the calls to `min` and `delmin` until the last possible moment. Then these functions operate on smaller trees. The rewritten `concat` then looks like `concat3`:

```
fun concat (E,  t2) = t2
  | concat (t1, E)  = t1
  | concat (t1 as T(v1,n1,l1,r1), t2 as T(v2,n2,l2,r2)) =
    if weight*n1 < n2 then T'(v2,concat(t1,l2),r2)
    else if weight*n2 < n1 then T'(v1,l1,concat(r1,t2))
         else T'(min t2,t1, delmin t2)
```

We can expect `difference` to be a little slower than `union` because `concat` always takes time $O(\log n)$.

A simple version of intersection relies on the identity $A \cap B = B - (B - A)$. The elements are removed from B so that the elements remaining come from B for uniformity with `union`.

```
fun intersection (a,b) = difference(b,difference(b,a))
```

This calls `difference` twice. It is faster to code `intersection` using divide and conquer like the previous operations:

```
fun intersection (E,_) = E
  | intersection (_,E) = E
  | intersection (s, T(v,_,l,r)) =
    let val l' = split_lt(s,v)
        val r' = split_gt(s,v)
    in
      if member(v,s) then
        concat3(v,intersection(l',l),intersection(r',r))
      else
        concat(intersection(l',l),intersection(r',r))
    end
```

The intersection contains only members occuring in both trees, so it is necessary to test the membership of `v` in the first tree.

10 Hedge_union

There is room for improvement in the performance of `union`. There is some inefficiency in the divide-and-conquer framework. At each level of recursion a tree is split into two subtrees, possibly leaving an element left over. This costs O($\log n$) time and space. At the next level of recursion the freshly constructed subtrees are split again. In this section we develop `hedge_union` which improves on the absolute efficiency of `union` by avoiding some of this cost.

Each of the subtrees produced by the splitting contains a subrange of the values in the original tree. We say that this set is the original set *restricted* to (i.e. intersected with) the subrange $(low, high)$. It is convenient to think of the original tree as being 'restricted' by the subrange $(-\infty, +\infty)$.

Instead of splitting the tree, we can just make a note of the subrange that restricts the tree. The triple $(tree, low, high)$ is used instead of the subtrees produced by the splitting. The splitting is deferred until the control tree (first argument of `union`) is empty, when we use `split_lt` and `split_gt` to extract the valid subrange. The result is `union'`:

```
fun union'  (E,(s2,lo,hi)) =
      split_gt(split_lt(s2,hi),lo)
  | union'  (T(v,_,l1,r1), (s2,lo,hi)) =
      concat3(v,
              union'(l1,(s2,lo,v)),
              union'(r1,(s2,v,hi)))
```

The astute reader will notice several problems with `union'`. Most important is that it runs in time $O(n \log n)$ which is worse than `union`'s $O(n)$ time. This is because for every one of the $O(n)$ empty subtrees in the first argument the entire second argument is split at $O(\log n)$ cost (remember `union` runs in $O(n)$ time because the vast majority of the splitting operations are performed on small trees).

A second problem with `union'` is that, as `s2` is always passed to the recursive calls unaltered, there is little point checking that it is empty. The operation $BigSet \cup SmallSet$ takes longer to compute than $SmallSet \cup BigSet$. The behavioural transparency of the algorithm has been lost.

Finally, the base case calls `split_gt` on the result of `split_lt`. A more efficient solution would be to implement the combined operation, but in fact this problem goes away when the previous two are solved.

The key observation is that if a tree is rooted at a node with value outside of the restricting range then only one of the subtrees can intersect that range. That subtree should be passed instead of the whole tree. We introduce the invariant that either the tree is empty or the root of the tree lies within the subrange. The invariant is established for any tree and subrange by descending the tree as in figure 4. The full implementation of `hedge_union`[10] is given in figure 5.

[10] The reason for thsi name is now apparent: the tree is bounded by the 'hedges' low and $high$ but these bounds are soft as bits of the tree may poke through them.

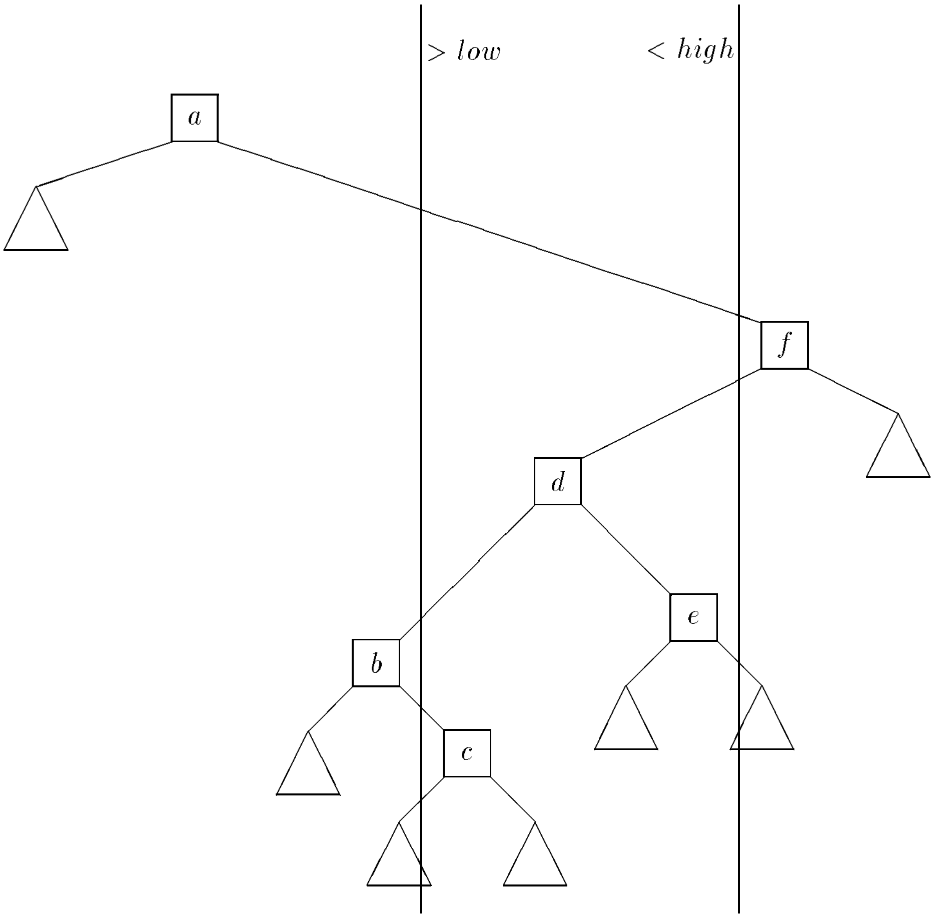


Fig. 4: Lazy subrange intersection. If the tree at a restricted to the exclusive range $(low, high)$ is equivalent to the tree at d restricted to the same range. The nodes a and f, and their left and right subtrees respectively may be discounted because they do not overlap the range $(low, high)$ and cannot overlap any subrange of $(low, high)$.

The `trim` function returns a (sub)tree satisfying the invariant. The heart of the algorithm is `uni_bd`, which cures the ills of `union'`. The first base case tests whether the second parameter is empty. This is worthwhile again because the invariant ensures that this parameter reduces in size in at least one of the recursive calls. In the second base case of `uni_bd` , when the first tree is empty, it is no longer necessary to call `split_gt` on the result of `split_lt` as we already know that `v` is between `lo` and `hi`. The subtrees can be trimmed as appropriate.

The algorithm starts with the range $(-\infty, +\infty)$. It is undesirable to program this explicity for a number of reasons. If values are chosen to represent the infinities, the algorithm will fail when those values appear in a set. There may be no such value, for example, it is always possible to find a greater alphabetic string by appending a 'z': 'z' < 'zz' < 'zzz' etc.

The solution used in figure 5 is specialise to the functions `trim` and `uni_bd` for the cases when one or both of `hi` $= +\infty$ and `lo` $= -\infty$. This accounts for all of the additional functions. Specialising a function is simple, if tedious:

- Copy the function and give it a new name.
- Remove the parameter, say `lo`.
- Subsititute $-\infty$ for `lo` in the body of the function.
- Use the properties of $-\infty$, like $-\infty$ |< lo| is alwats true, to remove all references to it.

The case where both `hi` and `lo` are infinities has the correct signature to replace `union`. Hedge versions of set difference and intersection may be derived in the same way.

The run time of `hedge_union` is $O(n)$, like `union`, but it is faster than `union` because it does fewer operations: the `split` functions are called only to build subtrees that are necessary. Test runs indicate that `hedge_union` is usually 20% faster than `union`. In the worst case, $SmallSet \cup BigSet$, it runs in about the same time. Whether this is worth the additional complexity depends on the application. Small constant factors are only worth this degree of effort when the program is a commonly used part of a library.

11 Additional operations: rank and indexing

Each tree node contains a count of the elements in the tree rooted at that node. This was justified in section 3—the information is used to ensure that only balanced trees were built, and is useful in its own right. Without the count data it would take $O(n)$ time to count the number of elements by traversing the tree instead of constant time.

The count data can also be used to determine the rank of an element and to retrieve an element by its rank. The elements are ranked according to the

```
fun trim (lo,hi,E) = E
  | trim (lo,hi,s as T(v,_,l,r)) =
    if  lt(lo,v)  then
       if  lt(v,hi)  then  s
       else  trim(lo,hi,l)
    else trim(lo,hi,r)

fun uni_bd (s,E,lo,hi) = s
  | uni_bd (E,T(v,_,l,r),lo,hi) =
      concat3(v,split_gt(l,lo),split_lt(r,hi))
  | uni_bd (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),lo,hi) =
      concat3(v,
              uni_bd(l1,trim(lo,v,s2),lo,v),
              uni_bd(r1,trim(v,hi,s2),v,hi))
     (* inv:  lo < v < hi *)

   (*all the other versions of uni and trim are
   specializations of the above two functions with
   lo=-infinity and/or hi=+infinity *)

fun trim_lo (_ ,E) = E
  | trim_lo (lo,s as T(v,_,_,r)) =
      if lt(lo,v) then s else trim_lo(lo,r)
fun trim_hi (_ ,E) = E
  | trim_hi (hi,s as T(v,_,l,_)) =
      if lt(v,hi) then s else trim_hi(hi,l)

fun uni_hi (s,E,hi) = s
  | uni_hi (E,T(v,_,l,r),hi) =
      concat3(v,l,split_lt(r,hi))
  | uni_hi (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),hi) =
      concat3(v,
              uni_hi(l1,trim_hi(v,s2),v),
              uni_bd(r1,trim(v,hi,s2),v,hi))

fun uni_lo (s,E,lo) = s
  | uni_lo (E,T(v,_,l,r),lo) =
      concat3(v,split_gt(l,lo),r)
  | uni_lo (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),lo) =
      concat3(v,
              uni_bd(l1,trim(lo,v,s2),lo,v),
              uni_lo(r1,trim_lo(v,s2),v))

fun hedge_union (s,E) = s
  | hedge_union (E,s2 as T(v,_,l,r)) = s2
  | hedge_union (T(v,_,l1,r1), s2 as T(v2,_,l2,r2)) =
      concat3(v,
              uni_hi(l1,trim_hi(v,s2),v),
              uni_lo(r1,trim_lo(v,s2),v))
```

Fig. 5: Hedge union

ordering relation <. The minimum element has rank 0, the next smallest has rank 1 and so on. By basing the rank on 0, the rank of an element is simply the number of elements to the left of that element. The rank is computed by summing the sizes of all the left-subtrees not taken on the path to the element. If the element doesn't appear in the tree then a `Subscript` exception is signalled.

```
exception Subscript

fun rank (E,x) = raise Subscript
  | rank (T(v,n,l,r), x) =
    if x<v then rank(l,x)
    else if x>v then rank(r,x) + size l + 1
         else size l
```

Indexing uses the size information to navigate to the element. At each node the index is checked against the size of the left subtree. If the index is smaller than the size of the subtree then the element must be somewhere in that subtree. If it is greater then it must be in the right subtree, but the number of elements in the left subtree and the one at current node must be discounted first. As always, if the element is in neither subtree then it must be at the current node.

```
fun index (E,_) = raise Subscript
  | index (T(v,_,l,r), i) =
    let val nl = size l
    in
        if i<nl then index(l,i)
        else if i>nl then index(r,i-nl-1)
             else v
    end
```

12 Summary

Very little in this report is new. Balanced binary search trees have been around for a long time. In particular, the methods of analysis and results, ($O(\log n)$ insertion, $O(n)$ union etc., are long established facts. This presentation, however, has several novel features:

- the functional programming style
- the abstraction away from the balancing algorithm
- the `hedge_union` algorithm

From this exercise we conclude that

- It is nearly as easy to implement balanced trees as unbalanced trees if the concept of 'rebalancing constructors' is taken on board.
- Rebalancing constructors need an absolute measure of the size or height of the tree. AVL trees which encode a height *difference* between left and right subtrees are not easy to code because the rebalancing depends on the *change* in height rather than an absolute measure.
- Bounded Balance binary trees are more useful than other type of binary tree.
- All the balanced tree schemes have logarithmic insert and delete, so when comparing them the constant factor is very important. One thing that affects the constant factor is the size of the node—every node has to be initialized and competes for space, putting presure on the memory allocation system. Small is beautifull.

The program described here was written in response to a competition put forward by Andrew Appel. Without this spur I would never have taken the time to finish the program, or write this report. I would like to thank Andy Gravell for the valuable discussions on the content and presentation of this report.

A Balance maintenance constraints

This section derives the constraints on the parameters used to maintain a balanced tree. The parameters are w and α. w is the bounded balance criterion, the maximum factor by which one subtree can outweigh its sibling. A tree is balanced if and only if either

1. Both subtrees have one or no elements, or

2. The number of elements in a subtree does not exceed w times the number of elements in the other subtree, i.e.

$$\begin{array}{rcl} size(left) & \leq & w \times size(right) \\ size(right) & \leq & w \times size(left) \end{array}$$

α is a decision variable. We will consider the case where a tree has been altered by the addition or deletion of a single element. Given a node which has two balanced subtrees, but the subtrees fail to satisfy the criteria (2) we will choose to apply a single or double rotation to restore balance. We choose a single or double rotation depending on the relative size of the subtrees of the heavier subtree of the unbalanced node. α measures the actual factor by which the outermost subtree exceeds the weight of the inner subtree. As this subtree is balanced, $1/w \leq \alpha \leq w$.

In the rest of this section is devoted to investigating relationship between w and α. We wish to know for what values of w do the tree rotations restore balance and what values of α should be used to choose a single or a double rotation. Only the case of the right subtree being one element too heavy is considered. This case is the same as the left tree being too light by one, and the other cases are generated by symmetry.

Single Rotation

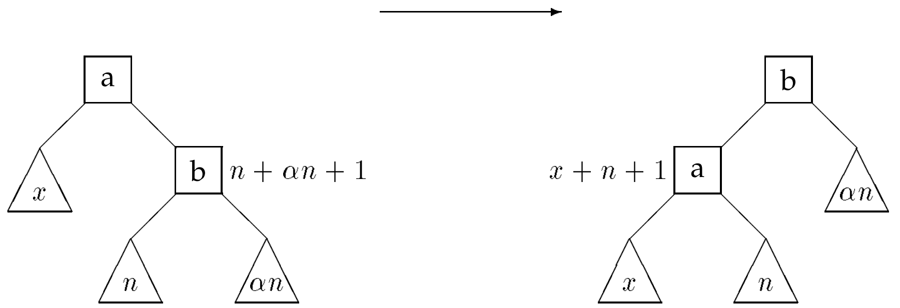


Fig. 6: Single rotation.

A.1 Single rotation

The transfer of elements in a single left rotation is characterized by figure 6. Before rotation the tree is out of balance at node a, so

$$n + \alpha n + 1 = wx + 1$$

giving

$$x = \frac{\alpha + 1}{w} n$$

We require that the rotated tree is balanced. Each each node the has to be balanced, with neither the left subtree nor the right subtree too heavy. This leads to the following constraints:

1. $x \leq w \cdot n$
2. $n \leq w \cdot x$
3. $x + n + 1 \leq w \cdot \alpha n$
4. $\alpha n \leq w \cdot (x + n + 1)$

Rearranging to constrain α in terms of w yields:

1. $\alpha \leq w^2 - 1$
2. $\alpha \geq 0$, which is ignored because it is weaker than $1/w \leq \alpha \leq w$.
3. $\alpha \geq (2w + 1)/(w^2 - 1)$
4. no constraint on α; $w \geq 0$

Some constraints, like case 3, express a constraint on α in terms of w and n. Since n can vary we pick the strongest constraint for any positive whole value of n. Case 3 is reduced as follows:

$$x + n + 1 \leq w \cdot \alpha n$$

Substituting for x and rearranging gives

$$\alpha \geq \frac{n(w + 1) + w}{n(w^2 - 1)}$$

If $n = 1$ this reduces to

$$\alpha \geq \frac{2w + 1}{w^2 - 1}$$

As $n \to \infty$ the single w in the numerator becomes irrelevant, so

$$\alpha \geq \frac{w + 1}{w^2 - 1}$$

The constraint for $n = 1$ is the stronger so it is chosen.

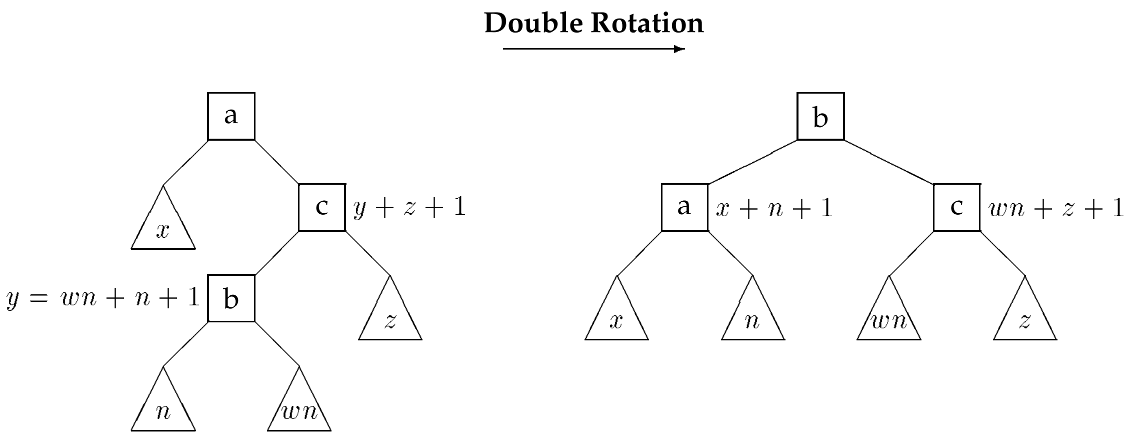


Fig. 7: Double rotation case (a).

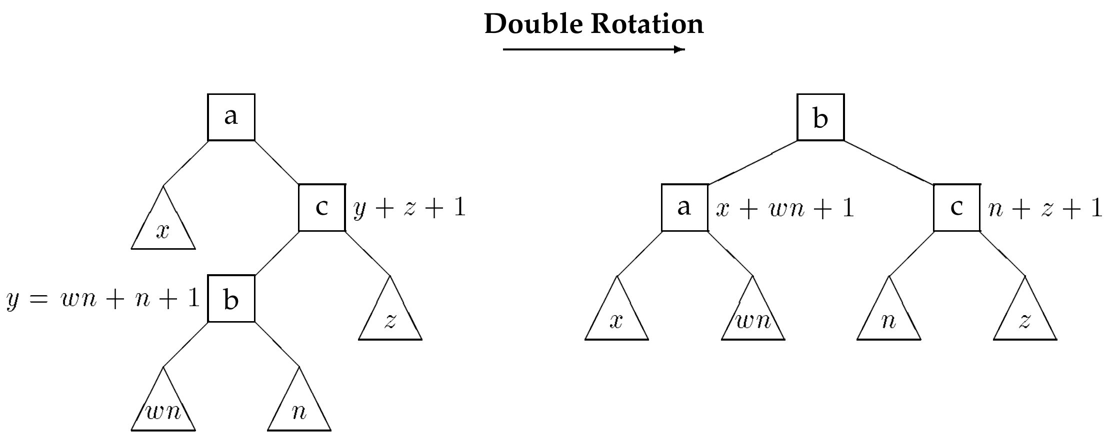


Fig. 8: Double rotation case (b).

A.2 Double rotation

There are two cases to be considered. The inner tree at b may be right-heavy (figure 7) or left-heavy (figure 8). These two cases are sufficient as it is our concern that the rotation will fail to balance the tree by leaving too little or adding too much at a and c in the final tree.

For the first case constraints are

1. $x \leq w \cdot n$, which reduces to

$$\alpha \leq \frac{n(w^2 - w - 1) + 1}{n(w+1) + 1}$$

As the numerator is quadratic in w, we have to choose the strongest constraint ($n = 1$ or $n \to \infty$) depending on w:

$$\begin{array}{ll} \alpha \leq w(w-1)/(w+2) & \text{if } w \geq 1+\sqrt{3} \text{ (when } n=1) \\ \alpha \leq (w^2-w-1)/(w+1) & \text{if } w \leq 1+\sqrt{3} \text{ (when } n\to\infty) \end{array}$$

2. $n \leq w \cdot x$ trivially yields $\alpha \geq 0$
3. $wn \leq w \cdot z$ yields $\alpha \geq 1/(w+1)$
4. $z \leq w \cdot wn$ yields $\alpha \leq w^2/(w+2)$
5. $x+n+1 \leq w \cdot (wn+z+1)$ yields $\alpha \geq -(w^2-w-1)/(w^2-1)$ which is subsumed by $\alpha \geq 0$ for $w^2-w-1 \geq 0$, i.e. $w \geq 1.6181$
6. $wn+x+1 \leq w \cdot (x+n+1)$, which does not constrain α.

For the second case the constraints are calculated as for the first case. There is only one case that is not weaker than other constraints that we have already encountered:

7. $z \leq w \cdot n$, which yields
$$\alpha \leq \frac{w}{w+2}$$

A.3 The (w, α) space

The graph in figure 9 illustrates the constraints that affect a practical[11] choice of w and α.

The graph may be interpreted as follows

- Single rotations restore balance in the clear region above the dashed line.
- Double rotations restore balance in the clear region below the dotted line.
- In the shaded region it is not possible to balance the tree. There are two reasons:
  1. There are no trees in the region above the line $\alpha = w$ and below $\alpha = 1/w$ because the right subtree must already be balanced.
  2. In the remainder of the shaded region both single and double rotations fail to restore balance.
- To guarantee that it is possible to restore balance w must be chosen to the right of the 'cusp' at (w_c, α_c) where $w_c \approx 3.745$ and $\alpha_c \approx 0.652$.

[11] Double rotation constraint 1 would nip the corner off the clear area near $(2,\frac{1}{2})$ but that does not affect the choice.

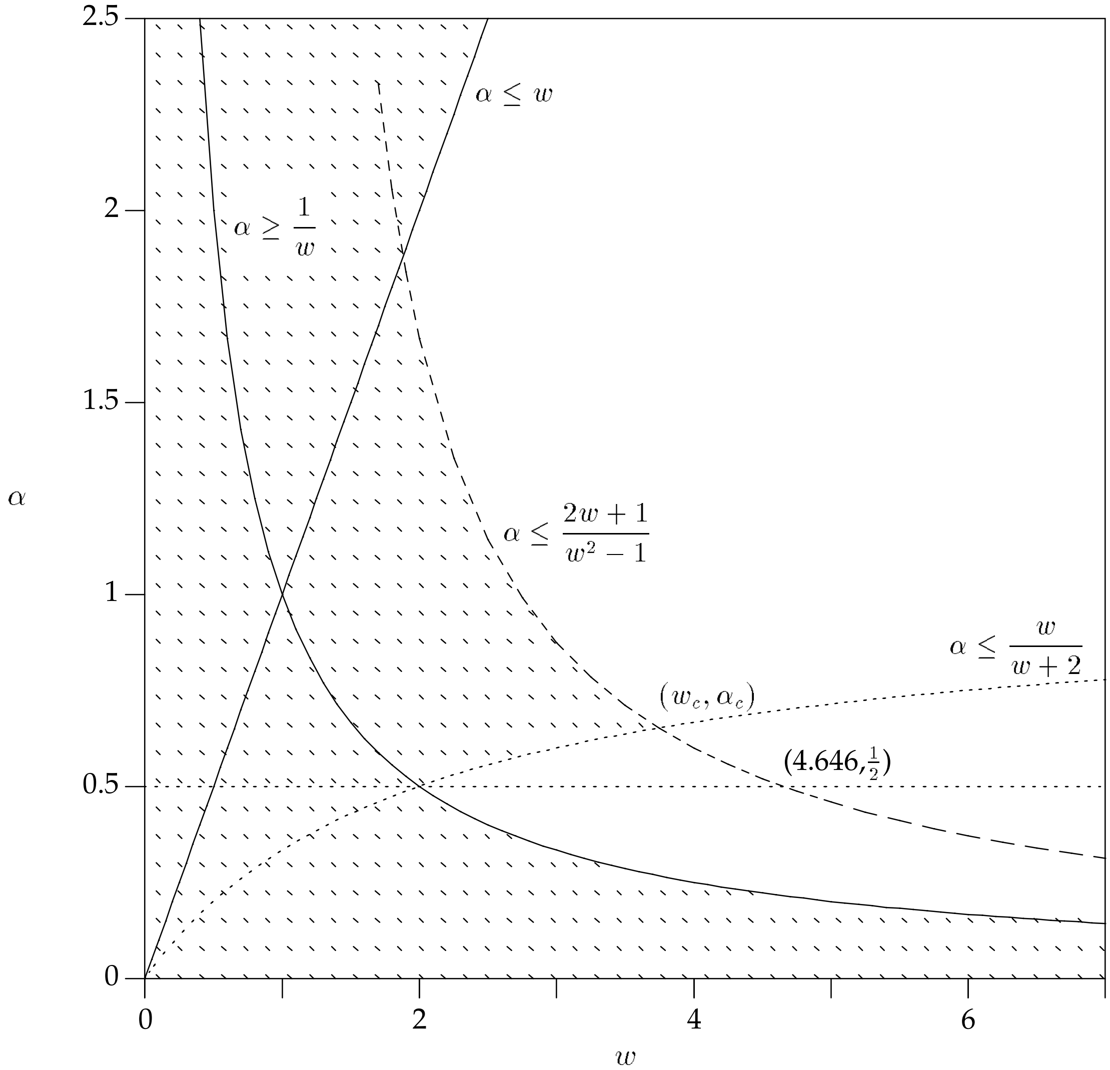


Fig. 9: The (w, α) parameter space. Values of w and α should be chosen to lie in the region below the dotted curve $\alpha = w/(w+2)$ and above the dashed curve $\alpha = (2w+1)/(w^2-1)$.

- If $w = w_c$ then the rebalancing algorithm must calculate α for the tree. If this $\alpha < \alpha_c$ then a double rotation must be applied, otherwise a single rotation must be applied.
- If a more convenient $\alpha = \frac{1}{2}$ is used this constrains us to use $w > 4.646$

B A simple way to reason about the trees

The analysis presented in the previous section is detailed and difficult. This appendix notes a less rigorous and easier way to reason about bounded balance trees.

The important factor determining the speed of operations on a tree are

- The maximum path length from the root node to an element. Balancing ensures that this height is logarithmic in the size of the tree.
- The extra cost of operations to detect and correct an imbalance.

A rotation is used to build a tree when otherwise the tree would be unbalanced. The criterion that no tree should have more than w times the number of elements than its sibling is roughly equivalent to saying that, since the height of a tree is logarithmic in its size, one tree should never be more than a fixed amount higher than its sibling. The rotations (figure 1) must be applied to lift the larger (hence taller) trees at the expense of the smaller trees. This is exactly what `T'` does.

Both analytical and empirical evidence ([4],[3]) suggests that there is little to choose between various balancing schemes, and that balancing is unnecessary on random data. Balancing is only necessary to prevent poor worst case behaviour and the evidence suggests that almost any scheme to avoid poor behaviour in the pathological cases will produce acceptable results.

Bibliography

[1] Aho, Hopcroft & Ullman, 1974. The Design and Analysis of Computer Algorithms. Addison-Wesley.

[2] Adelśon-Velśkii, G. M. and Y. M. Landis, 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.

[3] Guibas, L. J. and R. Sedgewick, 1978. "A dichromatic framework for balanced trees", Proc. 9th ACM Symposium on the Theory of Computing, 49–60.

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