Adams' Trees Revisited

Correct and Efficient Implementation

Milan Straka

fox@ucw.cz
Department of Applied Mathematics
Charles University in Prague, Czech Republic

Abstract. We present a correct proof of Adams' trees of bounded balance, which are used in Haskell to implement `Data.Map` and `Data.Set`. Our analysis includes the previously ignored `join` operation, and also guarantees trees with smaller depth than the original one. Because the Adams' trees can be parametrized, we use benchmarking to find the best choice of parameters. Finally, a saving memory technique based on introducing additional data constructor is evaluated.

1 Introduction

Adams' trees, or *trees of bounded balance* ω, shortly *BB-*ω trees, are binary search trees introduced in [1] and [2]. In order to guarantee an asymptotically logarithmic depth of these trees, the size of a subtree is stored in every node. This information is useful not only for rebalancing, but also for other operations – the `size` operation runs in constant time, we can locate the i-th smallest element of the tree in logarithmic time, and so on.

BB-ω trees are used in Haskell to implement the `Data.Map` and `Data.Set` modules, which are part of the standard data structure library `containers` [10]. BB-ω trees are also used in data structure libraries in Scheme and SML. According to the measurements in [9], their performance is comparable to other alternatives such as AVL trees [3] or red-black trees [4].

Although the implementation of a BB-ω tree is quite simple, proving its correctness is tricky. The original proof in [1] has several serious flaws – it wrongly handles `delete`, it does not consider `join` and does not use the fact that the sizes of subtrees are integers.

Our contributions are as follows:

- We present a correct proof of BB-ω trees, including the previously ignored `join` operation. This new analysis guarantees trees with lower depths than the original one.
- We investigate the depth of BB-ω trees.
- Because the BB-ω trees are parametrized, we perform several benchmarks to find the best choice of parameters.
- In order to save memory, we evaluate the technique of introducing additional data constructor representing a tree of size one. This allows us to save 20-30% of memory and even decreases the time complexity.

2 BB-ω trees

We expect the reader is familiar with binary search trees, see [6] for a comprehensive introduction.

Definition 1. *A binary search tree is a* tree of bounded balance ω*, denoted as* BB-ω tree*, if in each node the following balance condition holds:*

$$\begin{aligned} &\textit{size of the left subtree} \leq \omega \cdot \textit{size of the right subtree}\,, \\ &\textit{size of the right subtree} \leq \omega \cdot \textit{size of the left subtree}\,, \\ &\textit{if one subtree is empty, the size of the other one can be 1}\,. \end{aligned} \tag{1}$$

If the balance condition holds, the size of a tree decreases by at least the factor of $\frac{\omega}{\omega+1}$ at each level, so the maximum depth of a BB-ω tree with n nodes is bounded by

$$\log_{(\omega+1)/\omega} n = \frac{\log_2 n}{\log_2(1+1/\omega)}\,.$$

Detailed analysis is carried out in Section 6.

The exception for empty subtrees is not elegant, but from the implementation point of view is of no concern – empty subtrees are usually represented by a special data constructor and are treated differently anyway. Nevertheless, some modifications to the balance condition have been proposed to get rid of the special case – most notably to use the size of a subtree increased by one, which was proposed in [8]. We therefore define generalized version of balance condition, which comprises both these cases:

$$\begin{aligned} &\textit{size of the left subtree} \leq \max(1, \omega \cdot \textit{size of the right subtree} + \delta)\,, \\ &\textit{size of the right subtree} \leq \max(1, \omega \cdot \textit{size of the left subtree} + \delta)\,. \end{aligned} \tag{2}$$

The parameter δ is a nonnegative integer and if it is positive, the special case for empty subtrees is no longer necessary. Notice that the definition with sizes increased by one is equivalent to the generalized balance condition with $\delta = \omega - 1$.

An implementation of a BB-ω tree needs to store the size of a subtree of every node, which results in the following data-type:

```
data BBTree a = Nil           -- empty tree
              | Node          -- tree node
                  (BBTree a)   -- left subtree
                  Int          -- size of this tree
                  a            -- element stored in the node
                  (BBTree a)   -- right subtree
```

We also provide a function `size` and a smart constructor `node`, which constructs a tree using a left subtree, a key, and a right subtree. The balance condition is not checked, so it is upon the caller to ensure its validity.

```
size :: BBTree a -> Int
size Nil = 0
size (Node _ s _ _) = s

node :: BBTree a -> a -> BBTree a -> BBTree a
node left key right = Node left (size left + 1 + size right) key right
```

3 BB-ω tree operations

Locating an element in a BB-ω tree works as in any binary search tree:

```
lookup :: Ord a => a -> BBTree a -> Maybe a
lookup k Nil = Nothing
lookup k (Node left _ key right) = case k `compare` key of
                                     LT -> lookup k left
                                     EQ -> Just key
                                     GT -> lookup k right
```

When adding and removing elements to the tree, we need to ensure the validity of the balance condition. We therefore introduce another smart constructor `balance` with the same functionality as `node`, which in addition maintains the balance condition. To achieve effectiveness, certain conditions apply – the `balance` function can be used only on subtrees that previously fulfilled the balance condition and since then one element was inserted or deleted.

With such constructor, the implementation of `insert` and `delete` is straightforward. Assuming the `balance` constructor works in constant time, `insert` and `delete` run in logarithmic time.

```
insert :: Ord a => a -> BBTree a -> BBTree a
insert k Nil = node Nil k Nil
insert k (Node left _ key right) = case k `compare` key of
                                     LT -> balance (insert k left) key right
                                     EQ -> node left k right
                                     GT -> balance left key (insert k right)

delete :: Ord a => a -> BBTree a -> BBTree a
delete _ Nil = Nil
delete k (Node left _ key right) = case k `compare` key of
                                     LT -> balance (delete k left) key right
                                     EQ -> glue left right
                                     GT -> balance left key (delete k right)
  where glue Nil right = right
        glue left Nil = left
        glue left right
          | size left > size right = let (key', left') = extractMax left
                                     in node left' key' right
          | otherwise              = let (key', right') = extractMin right
                                     in node left key' right'

        extractMin (Node Nil _ key right) = (key, right)
        extractMin (Node left _ key right) = case extractMin left of
          (min, left') -> (min, balance left' key right)

        extractMax (Node left _ key Nil) = (key, left)
        extractMax (Node left _ key right) = case extractMax right of
          (max, right') -> (max, balance left key right')
```

When representing a set with a binary search tree, additional operations besides inserting and deleting individual elements are needed. Such an operation is join. The join operation is also a smart constructor – it constructs a tree using a key and left and right subtrees. However, it poses no assumptions on the sizes of given balanced subtrees and produces a balanced BB-ω tree.

In order to implement this operation we once again utilize the balance constructor, therefore changing its assumptions – the balance can be used on subtrees, that previously fulfilled the balance condition and since then one insert, delete or join operation was performed.

Having improved the balance operation, it is trivial to implement join. Once again, assuming balance works in constant time, join runs in logarithmic time.

```
join :: BBTree a -> a -> BBTree a -> BBTree a
join Nil key right = insertMin key right
  where insertMin key Nil            = Node Nil 1 key Nil
        insertMin key (Node l _ k r) = balance (insertMin key l) k r

join left key Nil = insertMax key left
  where insertMax key Nil            = Node Nil 1 key Nil
        insertMax key (Node l _ k r) = balance l k (insertMax key r)

join left@(Node ll ls lk lr) key right@(Node rl rs rk rr)
  | ls > omega * rs + delta = balance ll lk (join lr key right)
  | rs > omega * ls + delta = balance (join left key rl) rk rr
  | otherwise               = node left key right
```

4 Rebalancing BB-ω trees

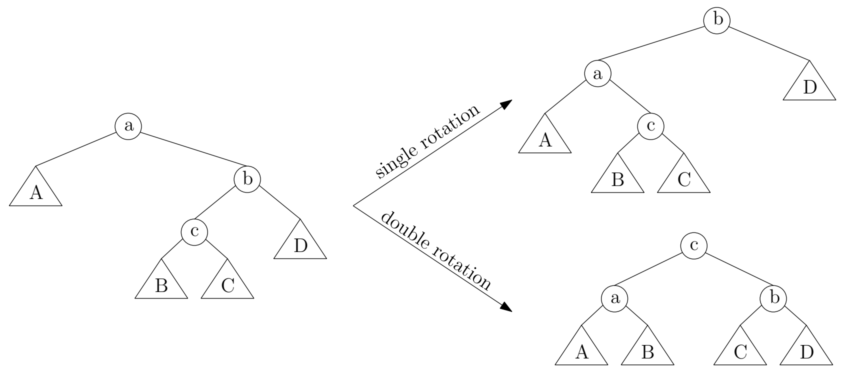
In order to restore balance, we use standard single and double rotations. These are depicted in Fig. 1. The code for these rotations is straightforward, the L or R suffix indicates the direction of the rotation (both rotations in the Fig. 1 are to the left).

The balance function restores balance using either a single or a double rotation – but a question is which one to choose. If we perform a left rotation as in Fig. 1, a single rotation leaves the left son of the right subtree unaffected, but a double rotation splits it into two subtrees. Therefore we choose the type of a rotation according to the size of the left son of the right subtree.

Formally, we use a parameter α[1], which we use as follows: When we want to perform a left rotation, we examine the right subtree. If its left son is strictly smaller than α-times the size of its right son, we perform a single rotation, and otherwise a double rotation. The implementation follows:

```
balance left key right
  | size left + size right <= 1 = node left key right
  | size right > omega * size left + delta = case right of
      (Node rl _ _ rr) | size rl<alpha*size rr -> singleL left key right
                       | otherwise             -> doubleL left key right
```

[1] Our α differs from [1], in the sense that our α is the inverse of α from [1].



```
singleL l k (Node rl _ rk rr) = node (node l k rl) rk rr
singleR (Node ll _ lk lr) k r = node ll lk (node lr k r)
doubleL l k (Node (Node rll _ rlk rlr) _ rk rr) =
  node (node l k rll) rlk (node rlr rk rr)
doubleR (Node ll _ lk (Node lrl _ lrk lrr)) k r =
  node (node ll lk lrl) lrk (node lrr k r)
```

Fig. 1. Single and double rotations.

```
  | size left > omega * size right + delta = case left of
      (Node ll _ _ lr) | size lr<alpha*size ll -> singleR left key right
                       | otherwise             -> doubleR left key right
  | otherwise = node left key right
```

5 Choosing the parameters ω, α and δ

We call the parameters (ω, α, δ) *valid*, if `balance` can always restore the balance condition after one `insert`, `delete` or `join` operation.

It would be best to fully characterize valid combination of parameters, but it difficult to do so. The reason is that the parameter validity heavily relies on the fact that small trees still have integral sizes – if the only available counterexamples are non-integral, the balance is always restored. However, because of this behaviour it is difficult to give generic characterization of parameter validity.

We therefore rule out parameters which are definitely not valid and then prove the validity only for several chosen parameters. It is easy to see that $\omega \geq 5$ and $\omega = 2$ are not valid for any α in the sense of the original balance condition, i.e., with $\delta = 0$: In the situation in Fig. 2 neither single nor double rotation can restore balance.

To get more accurate idea, we evaluated validity of parameters on all trees up to size of 1 million – the results are displayed in Fig. 3. The code used to generate this figure is listed in Appendix A. When choosing the parameters, the value of ω is the most important, because it defines the height of the tree. On

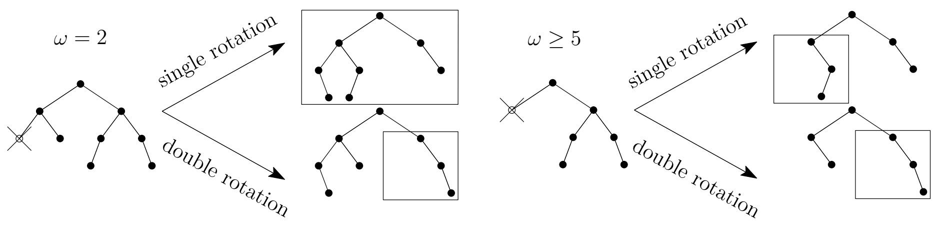


Fig. 2. Parameters $\omega = 2$ and $\omega \geq 5$ are not valid for any α and $\delta = 0$.

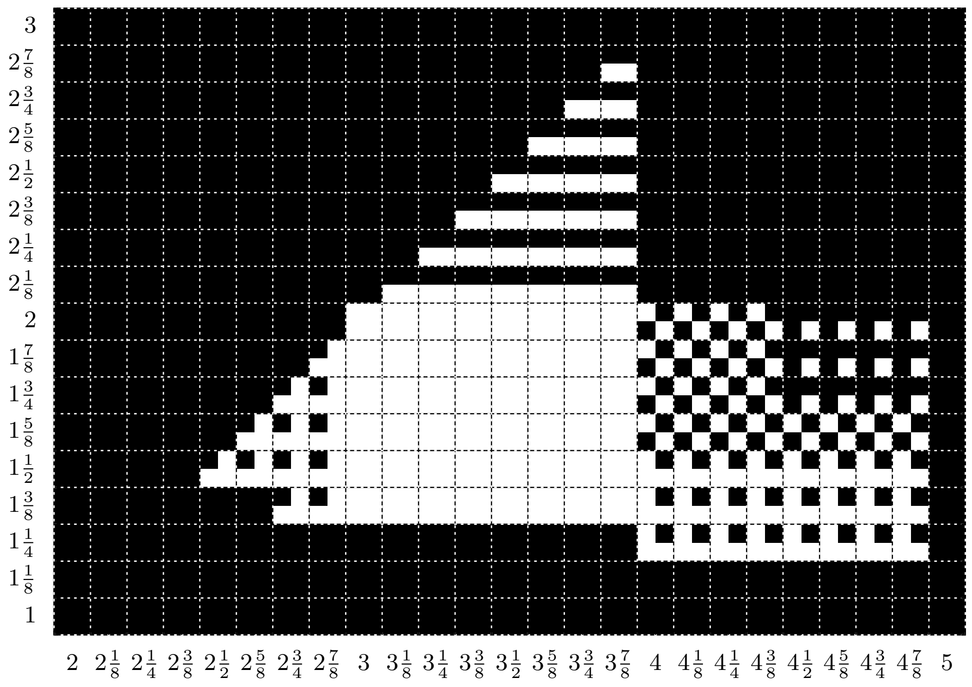


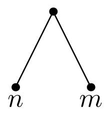
Fig. 3. The space of parameter (ω, α, δ). The values of ω and α are displayed on the x and y axis, respectively. Every dashed square consists of four smaller squares, which correspond to the δ values $\begin{smallmatrix}0 & 1\\ 2 & 3\end{smallmatrix}$. Black denotes non-valid parameters, white denotes parameters which are valid for trees of size up to 1 million. For example, when $\omega = 4$ and $\alpha = 2$, $\delta \in \{0, 3\}$ is valid and $\delta \in \{1, 2\}$ is not valid.

the other hand, the value of α is quite unimportant – it affects only the internal implementation of `balance`. The value of δ is kept as low as possible, since it increases imbalance of tree sizes.

After inspection of Fig. 3 we have chosen integer parameters ($\omega = 3, \alpha = 2, \delta = 0$) and ($\omega = 4, \alpha = 2, \delta = 0$) and also parameters ($\omega = 2.5, \alpha = 1.5, \delta = 1$), where the value of ω is the smallest possible. The last parameters are not integral, but we can perform multiplication by ω or α using right bit shift.

5.1 Validity of $w = 2.5$, $w = 3$ and $w = 4$

We now prove the validity of chosen parameters ($\omega = 2.5, \alpha = 1.5, \delta = 1$), ($\omega = 3, \alpha = 2, \delta = 0$) and ($\omega = 4, \alpha = 2, \delta = 0$). Because the values of α and δ are determined by ω, we identify these sets of parameters only by the value of ω.



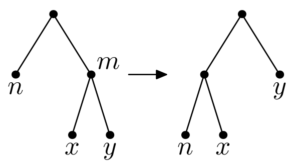
Consider performing `balance` after the balance is lost. Without loss of generality assume the right subtree is the bigger one and denote n and m the sizes of the left and right subtrees, respectively. We will use the notation of the tree size and the tree itself interchangeably.

Because the balance is lost, we have now $\omega n + \delta < m$. The `insert` operation causes imbalance by exactly one element, so it is never worse than imbalance caused by a `delete` operation. Therefore we have to consider only two possibilities how the imbalance was caused – `delete` or `join` operation. If the last operation was `delete`, we know that $\omega n + \delta \geq m - \omega$. If the last operation was `join` with the subtree of size z, we know that $\omega n + \delta \geq m - z$. During the `join` operation the tree z was small enough to be recursively joined with subtree m, so we have $\omega z + \delta < n + 1 + (m - z)$, so $z < \frac{n+1+m-\delta}{\omega+1}$ and therefore $m - \frac{n+m+1-\delta}{\omega+1} < \omega n + \delta$, $m < \frac{\omega+1}{\omega}\left(\omega n + \delta + \frac{n+1-\delta}{\omega+1}\right)$, $m < \frac{\omega+1}{\omega}\left(\omega n + \frac{n+\omega\delta+1}{\omega+1}\right)$, $m < \left(\omega + 1 + \frac{1}{\omega}\right) n + \delta + \frac{1}{\omega}$. To summarize:

$$m \overset{(A)}{>} \omega n + \delta\,, \quad m - \omega \overset{(B_{del})}{\leq} \omega n + \delta\,, \quad m \overset{(B_{join})}{<} \left(\omega + 1 + \tfrac{1}{\omega}\right) n + \delta + \tfrac{1}{\omega}\,.$$

5.2 Correctness of a single rotation

Let x and y denote the subtrees of the tree m. We perform a single rotation iff $x < \alpha y$ and in that case we have the following inequalities:



$$\omega x + \delta \geq y \Rightarrow (\omega + 1)x + \delta \overset{(C)}{\geq} m - 1\,,$$
$$x < \alpha y \Rightarrow x \overset{(D)}{<} \tfrac{\alpha}{\alpha+1}(m-1),\ y \overset{(E)}{>} \tfrac{1}{\alpha+1}(m-1)\,.$$

At first we need to solve the cases where n, x or y are zero, as the balance condition is different in that case. All such cases are shown in Fig. 4.

In the case when all subtrees are nonempty, we need to validate the balance condition in each of the two new trees:

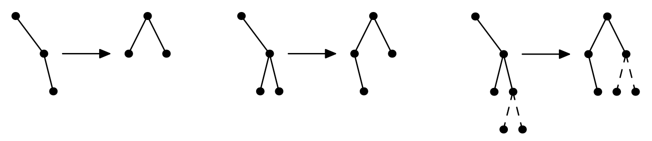


Fig. 4. Cases when n, x or y are zero and a single rotation is performed.

- $\omega n + \delta \geq x$ after `delete`: $x \overset{(D)}{<} \frac{\alpha}{\alpha+1}(m-1) \overset{(B_{del})}{\leq} \frac{\alpha}{\alpha+1}(\omega n + \delta + \omega - 1)$
- $\omega n + \delta \geq x$ after `join`: $x \overset{(D)}{<} \frac{\alpha}{\alpha+1}(m-1) \overset{(B_{join})}{<} \frac{\alpha}{\alpha+1}\left((\omega + 1 + \frac{1}{\omega})n + \delta + \frac{1}{\omega} - 1\right)$
- $\omega x + \delta \geq n$: $n \overset{(A)}{<} \frac{m-\delta}{\omega} \overset{(C)}{\leq} \frac{\omega+1}{\omega}x + \frac{1}{\omega}$
- $\omega(n+1+x) + \delta \geq y$: $y \leq \omega x + \delta$
- $\omega y + \delta \geq n+1+x$: $n+1+x = n+m-y \overset{(A)}{\leq} \frac{m-1}{\omega} + m - y = m\frac{\omega+1}{\omega} - y - \frac{1}{\omega} \overset{(E)}{<} ((\alpha+1)y+1)\frac{\omega+1}{\omega} - y - \frac{1}{\omega} = \frac{(\alpha+1)(\omega+1)-\omega}{\omega}y + 1$. Here we used the fact that when ω is an integer, $m \overset{(A)}{\geq} \omega n + \delta + 1$, so we have $m \overset{(A)}{\geq} \omega n + 1$.

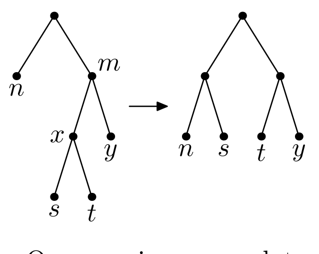
The third and the fourth inequalities obviously hold. To see that also the first, second and fifth inequalities hold, we evaluate the resulting inequalities and use the fact that the tree sizes are positive integers:

	$\omega n + \delta \geq x$ after `delete`	$\omega n + \delta \geq x$ after `join`	$\omega y + \delta \geq n + 1 + x$
$\omega = 2.5$	$x < \frac{3}{2}n + \frac{3}{2}$	$x < \frac{117}{50}n + \frac{6}{25}$	$n + 1 + x < \frac{5}{2}y + 1$
$\omega = 3$	$x < 2n + \frac{4}{3}$	$x < \frac{26}{9}n - \frac{4}{9}$	$n + 1 + x < 3y + 1$
$\omega = 4$	$x < \frac{8}{3}n + 2$	$x < \frac{7}{2}n - \frac{1}{2}$	$n + 1 + x < \frac{11}{4}y + 1$

The linear coefficients are always less or equal the required ones and it is simple to verify that all inequalities hold also for small integer sizes.

5.3 Correctness of a double rotation

When performing a double rotation, we have the following inequalities:



$$\text{any child } a \text{ of } b \Rightarrow (\omega+1)a + \delta \overset{(C)}{\geq} b - 1\,,$$
$$\text{any child } a \text{ of } b \Rightarrow (\omega+1)a \overset{(D)}{\leq} \omega(b-1) + \delta\,,$$
$$x \geq \alpha y \Rightarrow x \overset{(E)}{\geq} \frac{\alpha}{\alpha+1}(m-1),\ y \overset{(F)}{\leq} \frac{1}{\alpha+1}(m-1)\,.$$

Once again we need to solve the cases when n, y, s or t are zero – we enumerate these cases in Fig. 5.

When all subtrees are nonempty we create three new trees, so we have to check six inequalities:

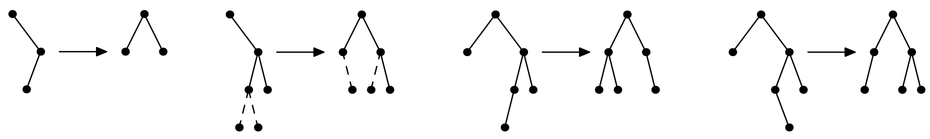


Fig. 5. Cases when n, y, s or t are zero and a double rotation is performed.

- $\omega n+\delta \geq s$ after `delete`: $s \overset{(D)}{\leq} \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \overset{(D)}{\leq} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) \overset{(B_{del})}{\leq} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(\omega n+\delta+\omega-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) = \frac{\omega^3}{(\omega+1)^2}n+\frac{\omega^3+\delta\omega^2-\omega^2+\delta\omega}{(\omega+1)^2}+\frac{\delta-\omega}{\omega+1}$
- $\omega n+\delta \geq s$ after `join`: $s \overset{(D)}{\leq} \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \overset{(D)}{\leq} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) \overset{(B_{join})}{<} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}((\omega+1+\frac{1}{\omega})n+\delta+\frac{1}{\omega}-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) = \frac{\omega^3+\omega^2+\omega}{(\omega+1)^2}n+\frac{\delta\omega^2-\omega^2+\delta\omega+\omega}{(\omega+1)^2}+\frac{\delta-\omega}{\omega+1}$
- $\omega s+\delta \geq n$: $n \overset{(A)}{<} \frac{1}{\omega}(m-\delta) \overset{(E)}{\leq} \frac{1}{\omega}(\frac{\alpha+1}{\alpha}x+1-\delta) \overset{(C)}{\leq} \frac{1}{\omega}(\frac{\alpha+1}{\alpha}((\omega+1)s+\delta+1)+1-\delta) = \frac{\omega+1}{\omega}\frac{\alpha+1}{\alpha}s+\frac{\delta+1}{\omega}\frac{\alpha+1}{\alpha}+\frac{1-\delta}{\omega}$
- $\omega t+\delta \geq y$: $y \leq \frac{x}{\alpha} \overset{(C)}{\leq} \frac{\omega+1}{\alpha}t+\frac{\delta+1}{\alpha}$
- $\omega y+\delta \geq t$: $t \overset{(D)}{\leq} \frac{\omega(x-1)+\delta}{\omega+1} \leq \frac{\omega(\omega y+\delta-1)+\delta}{\omega+1} = \frac{\omega^2}{\omega+1}y+\delta-\frac{\omega}{\omega+1}$
- $\omega(n+1+s)+\delta \geq t+1+y$ after `delete`: $\omega(n+1+s)+\delta \geq \omega(n+1)+t \overset{(B_{del})}{\geq} m-\delta+t \geq x-\delta+1+y+t$
- $\omega(n+1+s)+\delta \geq t+1+y$ after `join`: $t+1+y \leq \omega s+\delta+1+y \overset{(F)}{\leq} \omega s+\delta+1+\frac{m-1}{\alpha+1} \overset{(B_{join})}{<} \omega s+\delta+1+\frac{(\omega+1+\frac{1}{\omega})n+\delta+\frac{1}{\omega}-1}{\alpha+1} = \frac{\omega^2+\omega+1}{\omega(\alpha+1)}n+1+\frac{\omega(\delta-1)+1}{\omega(\alpha+1)}+\omega s+\delta$
- $\omega(t+1+y)+\delta \geq n+1+s$: $n+1+s \overset{(A)}{<} \frac{m}{\omega}+1+s \leq \frac{m}{\omega}+1+\omega t+\delta \overset{(C)}{\leq} \omega t+\delta+1+\frac{(\omega+1)y+\delta+1}{\omega} = \omega t+\frac{\omega+\delta+1}{\omega}+\frac{\omega+1}{\omega}y+\delta$

All but the first three inequalities obviously hold for positive integral sizes. In order to prove that the first three inequalities hold, we again evaluate the resulting inequalities and use the fact that the sizes are positive integers:

	$\omega n+\delta \geq s$ after `delete`	$\omega n+\delta \geq s$ after `join`	$\omega s+\delta \geq n$
$\omega = 2.5$	$s < \frac{125}{98}n+\frac{103}{98}$	$s < \frac{195}{98}n-\frac{1}{49}$	$n < \frac{7}{3}s+\frac{4}{3}$
$\omega = 3$	$s < \frac{27}{16}n+\frac{3}{8}$	$s < \frac{39}{16}n-\frac{9}{8}$	$n < 2s+\frac{5}{6}$
$\omega = 4$	$s < \frac{64}{25}n+\frac{28}{25}$	$s < \frac{84}{25}n-\frac{32}{25}$	$n < \frac{15}{8}s+\frac{5}{8}$

The linear coefficients are less or equal than the required ones and for small positive integral sizes the resulting inequalities imply the required ones, which concludes the proof.

6 BB-ω trees height

If the balance condition holds and $\delta \leq 1$, we know that the size of a tree decreases by at least a factor of $\frac{\omega}{\omega+1}$. Therefore the maximum height of a tree is $\frac{1}{\log_2(1+1/\omega)} \log_2 n$. But this is merely an upper bound – it is frequently not possible for the balance condition to be tight, because the tree sizes are integers.

To get an accurate estimate, we compute the maximum heights of BB-ω trees up to size of 1 million. We used the following recursive definition:

```
-- Returns the list [ max height of BB-w tree with n elements | n <- [1..] ].
heights :: Ratio Int -> Int -> [Int]
heights w d = result
 where
   result = 1 : 2 : compute_heights 3 1 result
   compute_heights n r rhs@(rhs_head : rhs_tail)
     | w*((n-1-(r+1))%1) + d%1 >= (r+1)%1 = compute_heights n (r+1) rhs_tail
     | otherwise = 1 + rhs_head : compute_heights (n+1) r rhs
```

The function compute_heights gets the size of the tree n, the size of the its right subtree r and also a list of maximum heights of BB-ω trees of r and more elements. It constructs the highest tree of size n by using the largest possible right subtree, and then using the highest tree of such size.

The resulting heights are presented in Fig. 6. The heights are divided by $\lceil \log_2 n \rceil$, so the optimal height is 1. Notice that the height of a BB-2.5 tree is always smaller than 2 for less than million elements – such height is better than the height of a red-black tree of the same size.

size of BB-ω tree	height divided by $\lceil \log_2 n \rceil$		
	$\omega = 2.5$	$\omega = 3$	$\omega = 4$
10	1.33	1.33	1.33
100	1.57	1.67	1.86
1 000	1.70	1.90	2.30
10 000	1.84	2.00	2.54
100 000	1.86	2.13	2.63
1 000 000	1.90	2.16	2.70
upper bound	2.06	2.41	3.11

Fig. 6. Maximum heights of BB-ω trees with $w = 2.5$, $w = 3$ and $w = 4$.

7 The performance of BB-2.5, BB-3 and BB-4 trees

With various possible ω to use, a search for the optimum value is in order. Is some value of ω universally the best one or does different usage patterns call for specific ω values?

We know that smaller values of ω result in lower trees. That seems advantageous, because the time complexity of many operations is proportional to the tree height.

In order to compare different values of ω, we measured the number of invocations of `balance` function. We inserted and then deleted $10^{\{1..6\}}$ elements, in both ascending and uniformly random order, and measured the number of invocations of `balance` during each phase. The results are displayed in Fig. 7.

	insert			delete		
	$w = 2.5$	$w = 3.0$	$w = 4.0$	$w = 2.5$	$w = 3.0$	$w = 4.0$
consecutive 10 elements	25	25	26	11	12	10
random 10 elements	23	23	23	12	12	12
consecutive 10^2 elements	617	657	769	362	349	302
random 10^2 elements	542	549	562	377	376	413
consecutive 10^3 elements	10245	11439	13997	6554	6116	5500
random 10^3 elements	8700	8753	8953	7162	7177	7377
consecutive 10^4 elements	143685	163261	206406	94865	88487	79938
random 10^4 elements	121192	121623	124204	105251	105854	108362
consecutive 10^5 elements	1852582	2133997	2722419	1251621	1175569	1042398
random 10^5 elements	1554230	1562168	1595269	1395871	1402939	1434371
consecutive 10^6 elements	22701321	26336469	33878677	15492747	14429384	12974950
random 10^6 elements	18956075	19074599	19476673	17367930	17480730	17856278

Fig. 7. The number of `balance` calls during inserting and deleting elements.

In case of ascending elements, smaller ω values perform better during insertion – the difference between $\omega = 2.5$ and $\omega = 4$ is nearly 50% for large number of elements. On the other hand, higher ω values perform better during deletion, although the difference is only 18% at most. In case of random elements, lower values of ω are always better, but the difference is less noticeable in this case.

We also performed the benchmark of running time of `insert`, `lookup` and `delete` operations. We used the `criterion` package [11], a commonly used Haskell benchmarking framework. All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 7.0.1. The benchmarking process works by calling the benchmarked method on given input data and forcing the evaluation of the result. Because the benchmarked method can take only microseconds to execute, the benchmarking framework repeats the execution of the method until it takes reasonable time (imagine 50ms) and then divides the elapsed time by the number of iterations. This process is repeated 100 times to get the whole distribution of the time needed, and the mean and the confidence interval are produced.

The benchmarks are similar to our previous experiment – we insert, locate and delete $10^{\{1..6\}}$ elements of type `Int`, which are both in ascending and uniformly random order. We used the implementation of `balance` from the `containers` package – we already improved this implementation in [9]. The resulting execution times are normalised with respect to one of the implementations and presented as percentages. The overview is in Fig. 8. (Ignore the trees with `One` subscript, they are explained in the next section.) Here the geometric mean of running times for all input sizes 10^1 to 10^6 is displayed. The detailed results and the benchmark itself are available on the author's website `http://fox.ucw.cz/papers/bbtree`.

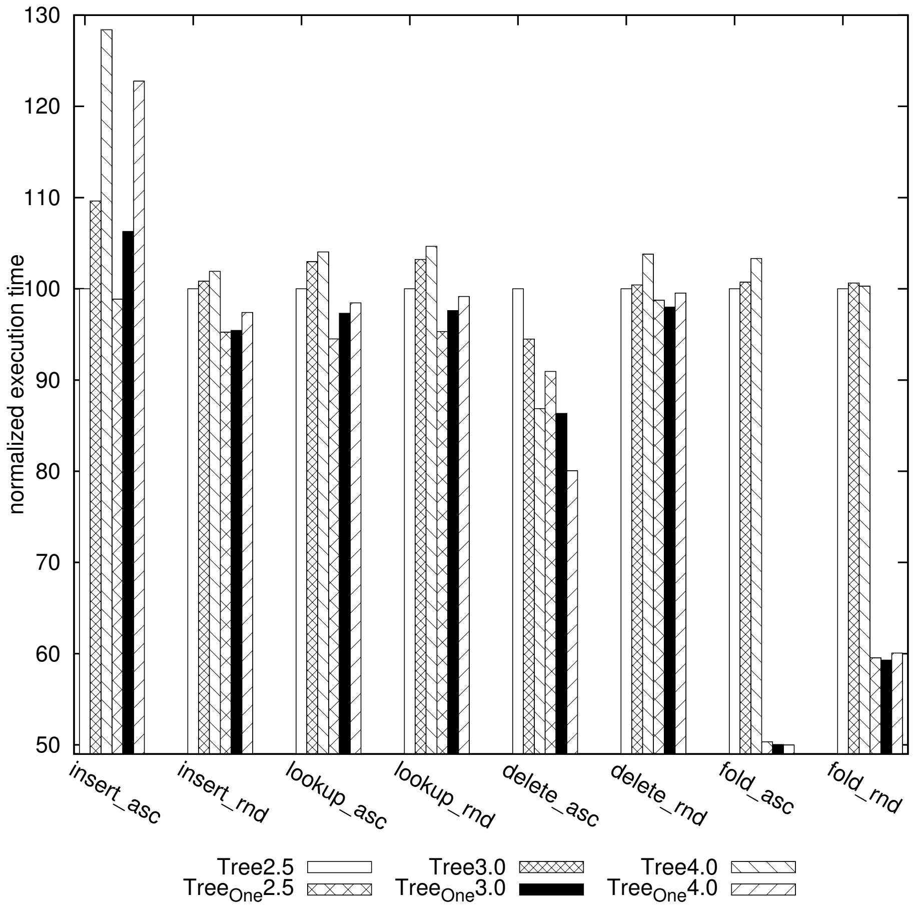


Fig. 8. The normalized execution times of BB-ω trees with various ω.

The findings are similar to the previous experiment – if the elements are in random order, the value of ω makes little difference, and smaller values perform slightly better. In case of ascending elements, smaller ω are better when inserting and larger when deleting. As expected, the `lookup` operation runs faster for smaller values of ω, independently on the order of elements.

8 Reducing memory by introducing additional data constructor

The proposed representation of a BB-ω tree provides room for improvements in terms of memory efficiency – if the tree contains n nodes, there are $n + 1$ `Nil` constructors in the whole tree, because every `Node` constructors contains two subtrees. We can improve the situation by introducing additional data constructor for a tree of size one:

```
data BBTree a = Nil           -- empty tree
              | One a         -- tree of size one
              | Node          -- tree node
                  (BBTree a)    -- left subtree
                  Int           -- size of this tree
                  a             -- element stored in the node
                  (BBTree a)    -- right subtree
```

Leaves are represented efficiently with this data-type. However, the trees of size 2 still require one `Nil` constructor.

To determine the benefit of the new data constructor we need to bound the number of `Nil` constructors in the tree. A `Nil` constructor appears in a tree of size 2 and if there are t trees of size 2, there need to be at least $(t-1)$ internal `Nodes` for these t trees to be reachable from the root. Therefore, there can be at most $n/3$ `Nil` constructors in the tree. This implies that the number of `One` constructors is between $n/3$ and $n/2$. Experimental measurements presented in Fig. 9 show that a tree created by repeatedly inserting ascending elements contains $n/2$ `One` and no `Nil` constructors, and a tree created by inserting uniformly random elements contains approximately $0.43n$ `One` and $0.14n$ `Nil` constructors.

	$T_{One}2.5$	$T_{One}3.0$	$T_{One}4.0$
any number of consecutive elements	50.0%	50.0%	50.0%
random 10 elements	45.5%	45.5%	45.5%
random 10^2 elements	43.6%	43.6%	43.6%
random 10^3 elements	43.0%	43.0%	42.8%
random 10^4 elements	43.0%	43.0%	43.0%
random 10^5 elements	42.8%	42.8%	42.9%
random 10^6 elements	42.9%	42.9%	42.9%

Fig. 9. The percentage of `One` constructors in a BB-ω tree.

Considering the memory representation used by the GHC compiler, the `Node` constructor occupies 5 words and `One` constructor occupies 2 words, so the new representation takes 20-30% less memory. The time complexity of the new representation is also better as shown in Fig. 8. Especially note the speedup of the `fold` operation, which is the result of decreased number of `Nil` constructors in the tree. The only disadvantage is the increase of the code size – but this affects the library author only.

We could also add a fourth data constructor to represent a tree of size 2. That would result in no `Nil` constructors in a nonempty tree. The disadvantage is further code size increase and also a noticeable time penalty – on 32bit machines GHC uses pointer tagging to distinguish data constructors without the pointer dereference, which is described in detail in [7]. This technique works with types with at most three data constructors (and up to 7 different constructors on 64bit machines), so it is not advantageous to add a fourth data constructor.

8.1 The order of data constructors

When implementing the data-type with the `One` constructor, we found out that the order of data constructors in the definition of the data-type notably affects the performance. On Fig. 10 you can see the time improvements in the benchmark from the previous section, when we reordered the constructors to the following order: `Node` first, then `One` and `Nil` last.

	$T_{One}2.5$	$T_{One}3.0$	$T_{One}4.0$
insert_asc	5.1%	6.8%	6.6%
insert_rnd	4.5%	5.2%	5.0%
lookup_asc	7.4%	6.1%	6.2%
lookup_rnd	6.1%	5.4%	5.4%
delete_asc	5.3%	8.4%	8.5%
delete_rnd	4.4%	4.8%	5.0%
fold_asc	8.9%	9.5%	13.1%
fold_rnd	10.1%	10.5%	9.4%

Fig. 10. The improvements of time complexity after reordering the data constructors.

We believe the reason for the performance improvement is the following: When matching data constructors, a conditional forward jump is made if the constructor is not the first one from the data-type definition. Then another conditional forward jump is made if the constructor is not the second one from the data-type definition. In other words, it takes $i - 1$ conditional forward jumps to match the i-th constructor from the data-type definition, and these forward jumps are usually mispredicted (forward jumps are expected not to be taken). It is therefore most efficient to list the data constructor in decreasing order of their frequency.

9 Conclusions

We described balanced trees and explicitly proved their correctness for several representative parameter combinations. For these parameters we also measured their runtime performance. The resulting implementation is comparable to other available on Hackage (this work started already in [9]). We also focused on memory complexity and improved it by changing the data-type representation. During this process we discovered the effect of the data constructors order in the data-type definition on the performance.

Several goals remain for future work. In our further efforts, we will incorporate the improvements described here in the `containers` package. We will also benchmark the effect of reordering data constructors of other data structures from the `containers` package – especially the `IntMap`, `IntSet`, `HashMap` and `HashSet`, which all use three data constructors. Also the benchmark of BB-ω trees could be extended to include set operations like `union`, `intersection` and others. We already described a benchmark with a `union` operation in [9].

9.1 Related work

The original weight balanced trees were described in [8], with two parameters with values $1+\sqrt{2}$ and $\sqrt{2}$. Because these are not integers, the resulting algorithm is not very practical. Adams created a variant of balanced trees, the BB-ω trees, and described them in papers [1] and [2]. Unfortunately, the proof is erroneous – the paper concludes that for $\alpha = 2$ the valid parameters are $\omega \geq 4.646$.

The error in the proof was known by several people, but in 2010 a bug was also found in the Haskell implementation – in the `Data.Set` and `Data.Map` modules from the `containers` package. The recent paper [5] deals with the correctness of the original weight balanced trees (equivalent to setting $\delta = \omega - 1$ in our definition) and proves in Coq, that for $\delta = \omega - 1$ the only integral valid parameters are $\omega = 3$ and $\alpha = 2$. Our proof on the other hand is explicit, and proves validity of only some chosen parameters. It covers both the original weighted trees and Adams' trees.

References

1. Adams, S.: Implementing sets efficiently in a functional language (Technical Report CSTR 92-10) (1992)
2. Adams, S.: Efficient sets – a balancing act. J. Funct. Program. 3(4), 553–561 (1993)
3. Adelson-Velskii, G.M., Landis, E.M.: An algorithm for the organization of information. Dokladi Akademia Nauk SSSR (146) (1962)
4. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. Foundations of Computer Science, Annual IEEE Symposium on 0, 8–21 (1978)
5. Hirai, Y., Yamamoto, K.: Balancing weight-balanced trees. Journal of Functional Programming (to appear)
6. Knuth, D.: The Art of Computer Programming, vol. 3: Sorting and Searching, chap. 6.2.2: Binary Tree Searching, pp. 426–458. Addison-Wesley, third edn. (1997)
7. Marlow, S., Yakushev, A.R., Jones, S.P.: Faster laziness using dynamic pointer tagging. In: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. pp. 277–288. ICFP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1291151.1291194

8. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Proceedings of the fourth annual ACM symposium on Theory of computing. pp. 137–142. STOC '72, ACM, New York, NY, USA (1972), http://doi.acm.org/10.1145/800152.804906
9. Straka, M.: The performance of the haskell containers package. In: Proceedings of the third ACM Haskell symposium on Haskell. pp. 13–24. Haskell '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1863523.1863526
10. The `containers` package, http://hackage.haskell.org/package/containers
11. The `criterion` package, http://hackage.haskell.org/package/criterion

A Generating the Fig. 3

When generating the Fig. 3 of valid parameters for all trees up to size of 1 million, we used the following code:

```
max_n = 1000000
find_min x p | p x       = last $ x : takeWhile p [x-1, x-2 .. 0]
             | otherwise = head $ dropWhile (not . p) [x+1, x+2 ..]

test w a d = and [delete n m && join n m | n <- [0 .. max_n],
                                           let m = flr $ max 1 (w * n + d)]
 where
  delete n m = n == 0 || rebalance (n-1) m
  join n m = rebalance n (m+increment)
    where increment = max 1 $ ceil ((n+m+1-d) / (w+1) - 1)

  rebalance n m = and [rebalance' n m x | x <- nub [x_min, x_mid - 1,
                                                    x_mid, m - 1 - x_min]]
    where x_min = find_min (flr $ m     / (w+1)) (\x -> balanced x (m-1-x))
          x_mid = find_min (flr $ m * a / (a+1)) (\x -> x >= a * (m-1-x))

  rebalance' n m x
    | x < a * y = balanced n x && balanced (n + 1 + x) y
    | otherwise = balanced n s && balanced t y && balanced (n+1+s) (t+1+y) &&
                  balanced n t && balanced s y && balanced (n+1+t) (s+1+y)
   where(y,s,t)=(m-1-x,find_min (flr$x/(w+1)) (\s->balanced s (x-1-s)),x-1-s)

  balanced n m = max 1 (w * n + d) >= m && n <= max 1 (w * m + d)

  flr, ceil :: Double -> Double
  flr = fromInteger . floor
  ceil = fromInteger . ceiling

results = [(w, a, d, test w a d) | w <- [2, 2.125 .. 5],
                                   a <- [1, 1.125 .. 3], d <- [0 .. 3]]
```

It relies on the fact that when there is a tree which cannot be balanced, there also exists a counterexample with a subtree as large as the balance condition allows. Therefore, for a fixed value of n it is enough to try the largest possible m and for a fixed value of m it is enough to verify that the balance condition is restored when considering the smallest and the largest subtree causing a single rotation and the smallest and the largest subtree causing a double rotation.