Adams’ Trees Revisited
Correctness Proof and Efficient Implementation

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

Abstract. We present a correctness 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.

Keywords: Data structures, balanced binary search trees, Haskell.

1 Introduction
Adams’ trees, or trees of bounded balance ω, shortly BB-ω trees, are binary search trees introduced in [1] and [2]. These trees are a popular choice for implementing purely functional search structures: They 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].

Every node of BB-ω tree has subtrees of relative size bounded by ω. This balance condition guarantees logarithmic depth, which is asymptotically optimal. The only correctness proof (published in [1]) has several serious flaws – it wrongly handles delete and it does not consider join. Recently a proof of a tree similar to BB-ω tree appeared in [5], presented using Coq Proof Assistant.

Our contributions are as follows:

− We present a correctness proof of BB-ω trees. In particular, we investigate the space of parameters and prove correctness for several chosen parameters: for all integral parameters and also for parameters that guarantee trees with smallest depth. Our analysis guarantees trees with lower depths than the original one and also considers previously ignored join operation.
− We show that the depth of BB-ω trees is better than the known upper bound.
− 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 to be 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:

\[
\begin{align*}
\text{size of the left subtree} & \leq \omega \cdot \text{size of the right subtree}, \\
\text{size of the right subtree} & \leq \omega \cdot \text{size of the left subtree},
\end{align*}
\]

if one subtree is empty, the size of the other one is at most 1. (1)

Consider a BB-ω tree of size n. The size of its left subtree is ω times the size of its right subtree, so the size of the left subtree is at most \(\omega^{\frac{1}{\omega+1}}n\). Therefore the size of a BB-ω tree decreases by at least a factor of \(\omega^{\frac{1}{\omega+1}}\) at each level, which implies that the maximum depth of a BB-ω tree with n nodes is bounded by

\[
\log_{\omega^{1/\omega}} n = \frac{1}{\log_2(1+1/\omega)} \log_2 n.
\]

Detailed analysis is carried out in Section 6.

The exception for empty subtrees in the definition of balance condition is not elegant, but from the implementation point of view it 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 a generalized version of the balance condition, which comprises both cases:

\[
\begin{align*}
\text{size of the left subtree} & \leq \max(1, \omega \cdot \text{size of the right subtree} + \delta), \\
\text{size of the right subtree} & \leq \max(1, \omega \cdot \text{size of the left subtree} + \delta).
\end{align*}
\]

The parameter \(\delta\) 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:

```haskell
data BBTree a = Nil -- empty tree
               | Node (BBTree a) -- 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.

```haskell
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
```