On the complexity of red-black trees for higher dimensions

Mikkel Engelbrecht Hougaard

February 2, 2015

Abstract

In this paper higher-dimensional trees that are balanced with a red-black balancing scheme are compared to ones implemented with a BB[α] balancing scheme. This research is motivated by results by George S. Lueker which show that an algorithm with fast operation costs for performing dynamic orthogonal range queries using a BB[α] balanced higher-dimensional tree exists. Whether similar results can be achieved using the red-black balancing scheme is interesting since it not only is simpler to implement correctly but also has relatively short path lengths that could make it perform better in terms of space cost. We will show how to algorithmically construct red-black trees that achieve maximum space cost for trees with some relatively small number of leaves n, as well as reason about general upper and lower bounds for enormous trees. We will also give a sequence of insert and delete operations that will perform poorly in terms of rebalancing cost.

Contents

1 Motivations and terminology

In this thesis we study special tree structures for storing points of higher dimensions. These trees which are a natural extension of ordinary binary search trees to higher dimensions store points only in its leaves and store other lower dimensional trees in its internal nodes. We shall refer to this structure as a *higher-dimensional* tree and define it formally as follows:

Definition 1.1. *A valid binary tree T where each internal node has exactly two children and points are stored in leaves is a higher-dimensional tree of dimension d if either $d = 1$, or $d > 1$ and each internal node v in T contains a higher-dimensional tree T_i of dimension $d-1$ and each point that is stored in a leaf in the sub-tree of v is also stored in a leaf in T_i.*

We will refer to a higher-dimensional tree of dimension d simply as a d-dimensional tree and we will refer to the tree contained in some node v as the internal tree of v. We refer to the single binary tree T_m in some d-dimensional tree T that is not an internal tree of any node in T_m as the main tree. Typically the points in a d-dimensional tree are points from $\mathbb{R}^d$, that are stored such that an inorder traversal of the d-dimensional tree T yields a list of points sorted on the d'th coordinate and an inorder traversals of any $d-1$-dimension internal tree will yield lists of points sorted on the $d-1$'th coordinate, see section 5 on range trees in [4]. We will not need to consider such restrictions in this paper though.

Since higher-dimensional trees are such a natural extension of ordinary binary trees they are of interest to study not only because of the practical results of their connection to orthogonal range searching, but also to establish theoretical results.

Lueker describes in [8] an algorithm for dynamically performing orthogonal range searches in higher dimensions. The dynamic version of the d-dimensional orthogonal range search problem is the the problem of reporting all d-dimensional points in a given set that are contained within a specified d-dimensional hyper cube while also allowing new points to be inserted into or removed from the set. This problem is ubiquitous in many areas of computer science and is a fundamental part of database theory and many geometrical problems. The algorithm proposed in the paper offers a $O(\log^d n)$ amortized time bound on inserting a new point into a set of n d-dimensional points and an $O(n \log^{d-1} n)$ bound on the space cost to store n d-dimensional points. The core part of the algorithm is the use of a higher-dimensional tree that uses a BB[α] rebalancing scheme. However higher-dimensional trees can be implemented using any rebalancing scheme, and it is of interest to see how these perform compared to the BB[α] case.

There are two restricting factors for how high the dimension of higher-dimensional trees can be before they stop being useful in practice: the cost of performing rebalancing in the tree and the space cost of storing the tree in memory. Interestingly these two costs are inversely related as we will show.

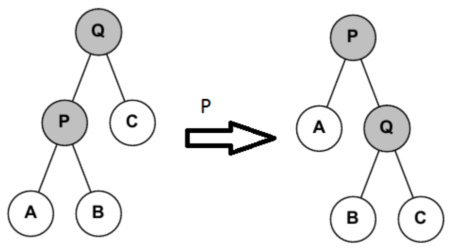

Figure 1: An example of an internal tree becoming invalid after a rotation. Recall by definition 1.1 that an internal tree must contain each point in the leaves of the sub-tree of the node it is contained in. The internal tree in P can be replaced with the internal tree from Q but no replacement exists for the new internal tree in Q.

1.1 Rebalancing higher-dimensional trees

As is the case with most dynamic tree structures, inserting into or removing points from higher-order may cause the tree to become invalid with respect to its rebalancing scheme and the tree must be rebalanced in some way. If a higher-dimensional tree is rebalanced with rotations however then the problem of invalidating an internal tree occurs.

Figure 1 shows the aftermath of a rotation in a binary tree. If the tree is a higher-dimensional tree then P now needs an internal tree that contains all the points in the sub-trees of A, B and C. Such a tree is readily available as the old internal tree in Q. Q however needs an internal tree that contains all the points in B and C and no internal tree with that combination of points exist. It is possible if the sub-trees of A and C contain a sufficiently small number of leaves to reuse the old internal structure from P and simply remove excess points while inserting the points from the sub-tree of C, but in the general case no better approach exists for Q than to simply rebuild the full internal tree from scratch.

Bentley and Friedman investigate in [1] the static case of orthogonal range searches also using higher-dimensional trees and note that a d-dimensional tree containing n points can be built statically in time $O(n \log^{d-1} n)$. This is a different result than Lueker since their statically built tree is near perfectly balanced, that is no path in the main tree or in the same internal tree will have length differing by more than one node. We can construct a list containing all the points sorted in the internal tree in any node by performing an in-order traversal. We need only merge two such lists to have a list in sorted order of all the points needed to rebuild an internal tree in a node, and we can thus build it statically using Bentley and Friedman's method. We thus infer that the total time cost of rebuilding the internal d-dimensional tree in a node with sub-tree containing n leaves is $O(n \log^{d-1} n)$.

If a balancing scheme performs many rotations on nodes that contain a large number of leaves in their sub-tree then the the rebuilding time of internal trees will dwarf the time it takes to search the tree when inserting and deleting points, and the tree becomes very ineffective.

1.2 Space cost of storing higher-dimensional trees

The other limiting factor as the dimensions of higher-dimensional trees grow is the space cost of storing the tree. Indeed any point in a leaf in a $d > 1$-dimensional tree will be stored in an internal tree in each of its ancestor nodes, these points will in turn be stored in internal trees of their ancestors and so forth. So for any higher-order tree that is balanced so its path lengths are logarithmic in the number of leaves it contains, it is easy to see that adding another dimension to the tree adds a factor of $O(\log n)$ to its space cost. The case is even worse if the d-dimensional tree is not balanced as a completely unbalanced tree with a path of length n will have space cost of the order $\Theta(n^d)$.

Finally we note that even though a d-dimensional tree of n leaves with $O(\log n)$ path lengths will have $O(n \log^{d-1} n)$ order space cost, the hidden coefficient in the cost may differ greatly. Indeed Lueker notes that the coefficient is dependent on d and as we shall see later, the difference between trees with n leaves and $O(\log n)$ path lengths may grow exponentially in d.

We previously described the merits a rebalancing scheme should have to be useful in the context of higher-dimensional trees. It must rebalance frequently enough that the space cost of storing the tree does not become prohibitive but not so frequently that the cost of inserting in or deleting points from the tree becomes too costly. We will briefly describe the two rebalancing schemes that are the focus of this thesis and list their pros and cons with respect to balancing higher-dimensional trees.

1.3 The BB[α] rebalancing scheme

The BB[α] rebalancing scheme keeps a binary tree balanced by imposing a restriction on how large a difference there can be in the number of nodes contained in the sub-tree children of a any node based on the parameter α. It was proposed by Nievergelt and Reingold [9] and has some interesting properties that fit well with higher-dimensional trees, although it is quite complex to implement and harbors some pitfalls. Indeed an error in the valid ranges of parameters in the original paper was not discovered until 8 years later by Blum, Norbert and Mehlhorn [2]. More recently an implementation of a variant of BB[α] trees in the standard library of the programming language Haskell was found to have invalid ranges too [7].

In a BB[α] tree a weight is associated with each internal node. The weight definition from the original paper states that the weight of an internal node v where n_l and n_r are the number of internal nodes plus the number of leaves in v's left and right sub-tree respectively is $w(v) = \frac{n_l+1}{n_l+1+n_r+1}$. The additional constants in the definition prevent the weight of any internal node from being 0 or 1 if an internal node is missing a child, but since internal nodes in this thesis always have exactly two children we can use a simpler definition:

Definition 1.2. *The weight of an internal node v where n_l and n_r are the number of leaves in v's left and right sub-tree child respectively is given by* $w(v) = \frac{n_l}{n_l+n_r}$.

Clearly the weight of any internal node v is a real valued number in the range $0 < w(v) < 1$. Using definition 1.2 a binary tree is valid with respect to the BB[α] rebalancing scheme if the following holds:

Definition 1.3. *A binary tree T is a BB[α] tree if for every internal node v in T it holds that $\alpha \leq w(v) \leq 1 - \alpha$.*

Note that by this definition a tree containing nothing but a leaf is valid since it has no internal nodes for the restriction to apply to. Any binary tree is valid with respect to BB[0] and only a fully balanced binary tree with a number of leaves that is a power of 2 is valid with respect to BB[1/2]. Values for α outside of this range are meaningless. If some insert or delete operation causes a BB[α]-tree to become invalid it is rebalanced using rotation.

One very desirable property of the BB[α] rebalancing scheme in the context of higher-dimensional trees is the fact that the change in weight of a node v after an insert or delete operation is smaller the more nodes there are in v's subtree. Since rotations can only occur when the weight of v is outside the valid range $[\alpha, 1-\alpha]$, if rebalancing occurs in a smart manner such that the weight of v is nearer the middle of the range after rebalancing, then it will take more operations to invalidate a large node than a small node. In the context of higher-order trees such a rebalancing would perform fewer costly rebuilds on large nodes compared to small nodes.

The value of α chosen influences how many rebalancing operations can potentially take place since the larger the range $[\alpha, 1-\alpha]$ is the more sub-tree structures are valid. Similarly more sub-trees with longer paths become valid too. Nievergelt and Reingold show that the path length in a BB[α] tree of n leaves is bounded by $C(\log n)$, but the that the constant C grows the smaller the value of α. Thus the choice of α directly influences the potential operations and space cost of higher-dimensional BB[α] trees. We explore these costs in section 2.2 and 4.1.

1.4 The red-black rebalancing scheme

The main focus of this thesis is establishing results for red-black higher-dimensional trees and comparing them with established results for BB[α] higher-dimensional trees. In [5] Guibas and Sedgewick proposes a rebalancing scheme that rebalances a tree based on its path lengths by associating a color with each internal node. The scheme is easy to understand and can be implemented with a series of simple near identical rebalancing cases. In a red-black binary tree each internal node is either colored red or black, and the following restrictions are imposed:

Definition 1.4. *A binary tree T is a red-black tree if the following four conditions are met:*

1 The root node in T is black.

2 If an internal node in T is red then its parent cannot also be red.

3 Any leaf in T has the same number of black internal node ancestors.

4 Any leaf in T is black.

Since each path in a red-black tree T must have the same number of black nodes, and red nodes cannot have red parents, it is easy to see that no path in T can exceed $2\log n$ in length. This is a desirable property in the context of higher-dimensional trees, as restrictions on path lengths are critical to lowering space costs of storing trees.

When inserting a point into a red-black tree T one black leaf v will be turned into a red internal node and the second property in definition 1.4 may be violated prompting rebalancing. Similarly deleting a point collapses an internal node to a black leaf which may violate property 3. In red-black trees rebalancing takes one of two forms, sometimes it is enough to recolor a number of nodes, which is of course no more costly in a higher-dimensional tree than in an ordinary binary tree, and sometimes rotations must occur. We will take an in depth look at rebalancing red-black trees in section 4.

2 Size coefficients in higher-dimensional trees

In this section we examine the space cost of higher-dimensional trees and compare space costs between implementations using red-black and BB[α] balancing schemes. The space cost is one of the key limiting factors as the cost rises exponentially as the dimension of the trees increase. Having small coefficients hidden in the asymptotic bound is therefore a very desirable property as the difference in space cost between two trees can be immense depending on how they are balanced. To stretch this point we will show that the coefficients in a fully balanced higher-dimensional tree may reduce the actual space cost of the tree significantly compared to the asymptotic result. But first we will formally define a function that maps a higher-dimensional tree to an integer that is a good representation of the space cost of the tree.

We define the size of a higher-order tree as the sum of the number of leaves in all of its internal 1-dimensional structures. So for example the size of a 1-dimensional tree is found by merely counting the number of leaves it contains, whereas the size of a 3-dimensional tree is found by summing the number of leaves in all 1-dimensional trees that are contained in all the 2-dimensional trees that are contained in the 3-dimensional tree.

Definition 2.1. *For any node v in a tree of dimension d, $D(v)$ is defined as follows:*

$d = 1:$

> $D(v) = 1$ *if v is a leaf.*
> $D(v) = D(v_l) + D(v_r)$ *if v is an internal node, where v_l and v_r are the left and right children of v.*

$d > 1:$

> $D(v) = 0$ *if v is a leaf.*
> $D(v) = D(v_i) + D(v_l) + D(v_r)$ *if v is an internal node, where v_i is the root of the $(d-1)$-dimensional internal tree in v and v_l and v_r are the left and right children of v.*

Definition 2.2. *The size of a tree T is the size of the root node of T.*

This definition is sensible since the number of leaves in the first order structures easily outgrows the number of leaves or internal nodes in all other internal trees. This is the case since the number of leaves in trees of dimension $d-1$ in a d-dimensional tree with n leaves is at least $n \log n$, and the number of internal nodes in any tree with n leaves is exactly $n-1$. It is also a very simple definition to work with, and thus well suited for establishing bounds on the space costs of higher-dimensional trees. It is trivial to see that the size of any 2-dimensional tree is exactly equal to the sum of the number of ancestors of each leaf in the tree, the so called external path length, and the size of trees can thus be thought of as a generalization of external path lengths to trees trees of higher dimensions which makes it an interesting characteristic to study.

Note that the size of a higher-dimensional tree is highly dependent on the lengths of the root to leaf paths in the tree. Any leaf will have its value stored in the lower dimensional internal trees of each of its ancestor internal nodes.

2.1 A strict lower bound on sizes of higher-dimensional trees

We will now use Definition 2.1 to prove a strict lower bound on the size of any higher-dimensional tree under a reasonable assumption about the size coefficients of minimum size higher-dimensional trees. We note that a fully balanced 2-dimensional tree with a number of leaves that is a power of 2 achieves a size coefficient of 1, which is clearly minimal for any 2-dimensional tree. We will generalize such trees to higher dimensions in the following way: a higher-dimensional tree T is fully balanced if for any node v in either its main tree or an internal tree in T, all paths from v to one of its descendant leaves have the same length. We will make an assumption that a result similar to the 2-dimensional case holds holds for higher-dimensional trees in general.

Assumption 2.3. *For any d-dimensional tree T with $2^x \leq n \leq 2^{x+1}$ leaves, the size coefficient of T is greater than or equal to the size coefficient of a fully balanced d-dimensional tree of either 2^x or 2^{x+1} leaves depending on whether the size coefficients of fully balanced d-dimensional trees increase or decrease with number of leaves.*

We will show that the sizes of fully balanced higher-dimensional trees converge. The size of a fully balanced higher-dimensional tree is given by the following recurrence.

Lemma 2.4. *The size of a fully balanced d-dimensional tree where each path from the root to a leaf in the main tree has length h is given by*
$F_d(h) = h2^h$ *if* $d = 2$.
$F_d(h) = \sum_{i=0}^{h-1} 2^i F_{d-1}(h-i)$ *if* $d > 2$.

Proof. For $d = 2$ it suffices to see that there are 2^h leaves and each leaf contributes its root distance, h, to $F_d(h)$. For $d > 2$ assume that the result holds for trees with dimension $d-1$ and look at the i'th layer corresponding to the i'th iteration of the sum. The layer contains 2^i nodes that all have the same number of leaves in their subtree, 2^{h-i}. Since the internal tree in each node is fully balanced the size of any node in the layer is given by $F_{d-1}(h-i)$ by the assumption. Thus the total size of the layer is given by $2^i F_{d-1}(h-i)$. Since the assumption holds for 2-dimensional trees it holds for $d > 2$-dimensional trees as well. □

Calculating a closed form of the recurrence in Lemma 2.4 for trees of dimension 3 to 7 gives the following list of functions for the size of fully balanced d-dimensional trees:

- $F_3(h) = \frac{1}{2}(h^2 + h)2^h$

- $F_4(h) = \frac{1}{6}(h^3 + 3h^2 + 2h)2^h$
- $F_5(h) = \frac{1}{24}(h^4 + 6h^3 + 11h^2 + 6h)2^h$
- $F_6(h) = \frac{1}{120}(h^5 + 10h^4 + 35h^3 + 50h^2 + 24h)2^h$
- $F_7(h) = \frac{1}{720}h(h+5)(h^4 + 10h^3 + 35h^2 + 50h + 24)2^h$

Observe that the coefficient of the dominating term in the polynomial is exactly $\frac{1}{(d-1)!}$ for the d'th size function. This suggests that the ratio of the size of a fully balanced d-dimensional tree with all root to leaf path lengths being h and $n = 2^h$ leaves and $\frac{1}{(d-1)!} n \log^{d-1} n$ will converge to 1 as h increases. This result is trivially true for 2-dimensional trees. We will use this assumption to inductively show its correctness for $d > 2$-dimensional trees.

Lemma 2.5. *If $F_d(h') = 2^{h'} \left(\frac{1}{(d-1)!} h'^{d-1} + P_{d-2}(h') \right)$ for all $h' \geq 2$ then*

$$F_{d+1}(h) = 2^h \left(\frac{1}{d!} h^d + P_{d-1}(h) \right),$$

where $P_d(h)$ is an arbitrary polynomial of h with degree d and coefficients depending only on d.

Proof. By Lemma 2.4 $F_{d+1}(h)$ is given by:

$$F_{d+1}(h) = \sum_{i=0}^{h-1} 2^i F_d(h-i).$$

Using the inductive hypothesis on $F_{d+1}(h)$ yields

$$F_{d+1}(h) = 2^h \left(\sum_{i=0}^{h-1} \frac{1}{(d-1)!}(h-i)^{d-1} + \sum_{i=0}^{h-1} P_{d-2}(h-i) \right).$$

Let S_1 be the left sum and S_2 the right sum in the above definition of $F_{d+1}(h)$. It is clear that both sums yield polyonomials of h with coefficients only depending on d. The degree of the polynomial S_2 is clearly $d-1$ so we need not investigate it but can merely write $S_2 = P_{d-1}(h)$. The degree of S_1 is d and the coefficient of the d'th term must be found.

$$S_1 = \sum_{i=0}^{h-1} \frac{1}{(d-1)!}(h-i)^{d-1}$$

is equivalent to

$$= \frac{1}{(d-1)!} \sum_{i=1}^{h} (i)^{d-1}.$$

Faulhaber's formula states that

$$\sum_{i=1}^{x} i^y = \frac{1}{y+1} \sum_{j=0}^{y} (-1)^y \binom{y+1}{j} B_j x^{y+1-j},$$

where B_i is the i'th Bernoulli Number. For a full topic on Faulhaber's formula and Bernoulli numbers see The Book of Numbers [3] by Conway et al. The sum in the formula is similar to S_1 with $y = d-1$ and $x = h$ so we may rewrite S_1 as

$$S_1 = \frac{1}{d!} \sum_{i=0}^{d-1} (-1)^i \binom{d}{i} B_i h^{d-i}.$$

Unrolling the first iteration of the sum yields the coefficient for the d degree term

$$S_1 = \frac{1}{d!} \left((-1)^0 \binom{d}{0} B_0 h^d + \sum_{i=1}^{d-1} (-1)^i \binom{d}{i} B_i h^{d-i} \right).$$

Since $(-1)^0 = \binom{d}{0} = B_0 = 1$ and since the i'th Bernoulli number is a function only of i, the remaining sum is a polynomial of degree $d-1$ in h with coefficients depending only on d. S_1 becomes

$$S_1 = \frac{1}{d!} h^d + P_{d-1}(h).$$

Inserting the found values for S_1 and S_2 into $F_{d+1}(h)$ now gives

$$F_{d+1}(h) = 2^h \left(\frac{1}{d!} h^d + P_{d-1}(h) \right)$$

which is the desired result and completes the proof. □

Since the assumption in Lemma 2.5 is trivially true for 2-dimensional trees the result holds for any d-dimensional tree where $d \geq 2$. Since the size coefficient of fully balanced higher-dimensional trees decreases as the number of leaves increases, Lemma 2.5 gives a lower bound for the size coefficient of any d-dimensional tree provided Assumption 2.3 is true.

Note that even though this result may seem to imply that the size of fully balanced trees may grow very slowly in practice, large values of h are needed to approach coefficient values near $\frac{1}{(d-1)!}$ which in turn increases h^{d-1}.

2.2 Maximum sizes of BB[α] higher-dimensional trees

In this section we will evaluate the coefficients in the worst case space cost of higher-dimensional trees implemented with a BB[α] rebalancing scheme. From Lueker [8] we know that the worst-case space cost of a d-dimensional BB[α] tree with n leaves is $O(n \log^{d-1} n)$ but with coefficients depending on d. We will calculate these coefficients using the size definition from definition 2.2.

Constructing a maximum size 2-dimensional BB[α] tree is simple since the the maximum allowed length of a path in some sub-tree depends on the number of leaves in the sub-tree, that is the more leaves are in a sub-tree, the greater the number of ancestors each leaf can potentially have. Thus, since the size contribution of a leaf in a 2-dimensional tree is exactly the number of ancestors the leaf has, a maximum size 2-dimensional BB[α]-tree T is a tree where the root of each sub-tree of n leaves in T has one child with exactly $\lceil \alpha n \rceil$ leaves and one child with exactly $\lfloor (1-\alpha)n \rfloor$ leaves.

Given some N and D we can easily calculate all sizes of d-dimensional BB[α] trees with $1 \leq n \leq N$ leaves and $1 \leq d \leq D$ where the main and all internal trees follow the structure of maximum size 2-dimensional trees. The algorithm uses two nested loops. The outermost loop iterates over the dimension of the tree, beginning with $d = 2$, and the innermost loops over number of leaves, note that no calculation needs be done for $d = 1$ since the size of any 1-dimensional tree is the number of leaves it contains.

Given that the maximum size of a tree of only one leaf is 0 for any dimension we can calculate at the $d'th$ innermost step and $i > 1$'th outermost step the maximum size of a tree of i leaves and dimension d as the sum of sizes of its two children and the size of the internal tree in its root. The number of leaves in the children are given by $n_1 = \lceil \alpha i \rceil$ and $n_2 = \lfloor (1-\alpha)i \rfloor$. Clearly $n_1, n_2 > 0$ for all $i > 1$ and the maximum sizes of these trees will already be calculated at the i'th innermost step. The internal tree will have dimension $d - 1$ and will have been calculated in the previous outermost step. This procedure can be completed in $O(DN)$ operations.

On Figure 2 size coefficients of BB[1/3] trees with up to 10^5 leaves and dimension of 2 to 6 calculated by the algorithm are shown. 1/3 is the parameter used in the algorithm by Lueker that achieves very favorable operation costs for performing dynamic orthogonal range searches. We note that the space coefficients are also quite small and that the 2-dimensional case appears nearly constant as the number of leaves increase. Higher dimensional trees have smaller coefficients that decrease very slowly in the plotted range.

Figure 3 shows size coefficients for BB[0.1] trees with same leaf and dimension parameters. We note that the difference in coefficients between the 0.1 and the 1/3 α case are relatively small for low dimensions but become more pronounced as this increases. We also note that the lower α value case is more volatile and the coefficients decrease more rapidly in higher dimensions as the number of leaves increase.

Whether the structure used here produces maximum size $BB[\alpha]$ trees for dimensions greater than 2 is not easily proved. However in chapter 4 we give an

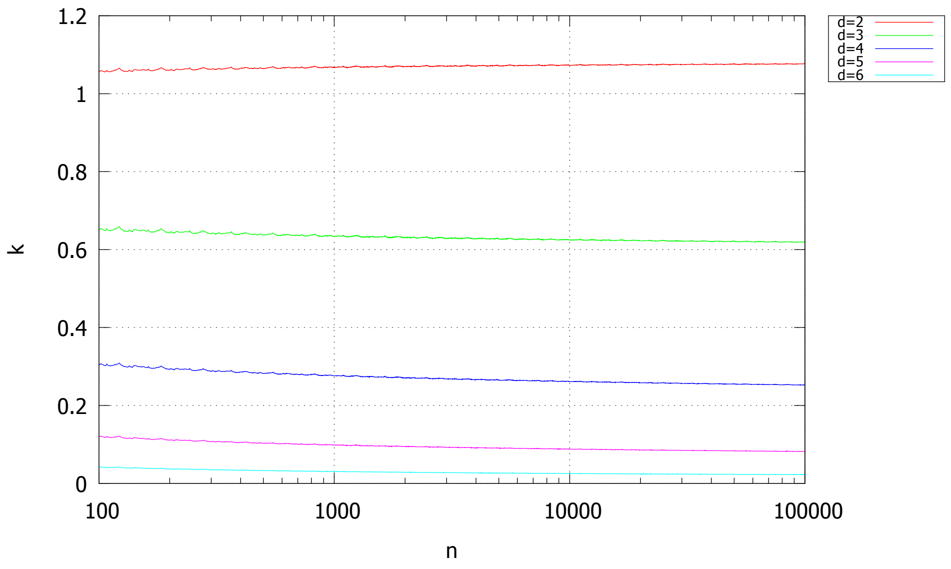

Figure 2: Size coefficients of assumed maximum size BB[1/3] d-dimensional trees with n leaves on logarithmic scale.

algorithm for computing actual maximum sizes of red-black trees. This algorithm can also calculate maximum size BB[α] trees if it is slightly modified to, instead of trying maximum size red-black candidate pairs, tries all maximum size BB[α] candidate pairs that do not invalidate the weight constraint of its root. Running this modified algorithm is slightly less computation intensive since the number of candidates is smaller, however it is still prohibitive for large numbers of leaves. We base the assumption that the maximum size 2-dimensional BB[α] structure provides maximum size trees for higher dimensions on the fact that the modified algorithm from chapter 4 and the algorithm presented here agree on values for all BB[α] trees of dimension $1 \leq d \leq 6$ and $1 \leq n\ 10^4$ leaves.

An upper bound on sizes of the 2-dimensional case is given by Nievergelt and Wong [10]. Nievergelt and Wong state that if $|T|$ is the sum of the number of ancestors for each leaf and internal node in a BB[α] tree of n leaves and N internal nodes then $|T|$ is strictly bounded above by

$$|T| \leq \frac{1}{H(\alpha)}(n+N+1)\log(n+N+1) - 2(n+N)$$

where

$$H(\alpha) = -\alpha\log\alpha - (1-\alpha)\log(1-\alpha).$$

Since the size of a 2-dimensional tree is the sum of the number of ancestors for each leaf in the tree we can show that Nievergelt and Wong's result can also be used to bound the sizes of 2-dimensional BB[α] trees. We do this by showing that the size $D(T)$ of a tree T is proportional to $|T|$.

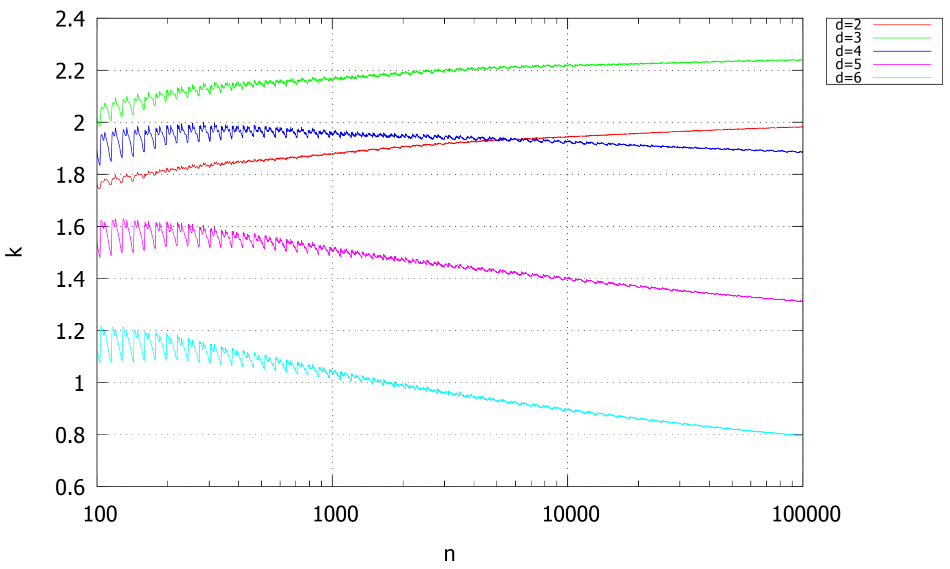

Figure 3: Size coefficients of assumed maximum size BB[0.1] d-dimensional trees with n leaves on logarithmic scale.

Lemma 2.6. *For any 2-dimensional tree T of n leaves and N internal nodes the sum of the number of ancestors for each leaf and internal node in T is related to the size of T by $|T| = 2D(T) - 2N$.*

Proof. The proof is trivially true for any tree of one leaf since $N = 0$ and $|T| = D(T) = 0$. We will prove the general case by structural induction on the trees. Assume the result holds for trees T_1 with n_1 leaves and N_2 internal nodes and T_2 with n_2 leaves and N_2 internal nodes. Joining these two trees together as children of a new internal root node produces a tree T with $n = n_1 + n_2$ leaves and $N = N_1 + N_2 + 1$ internal nodes. Now clearly every internal node except the new root and every leaf in T will have exactly one more ancestors than it had in T_1 or T_2 so

$$|T| = |T_1| + |T_2| + n + N - 1$$

and

$$D(T) = D(T_1) + D(T_2) + n.$$

Using the assumption $|T|$ becomes

$$|T| = 2D(T_1) - 2N_1 + 2D(T_2) - 2N_2 + n + N - 1$$

$$= 2D(T_1) + 2D(T_2) + n - N + 1$$

which can be rewritten using the value of $D(T)$ as

$$= 2D(T) - n - N + 1.$$

Since each internal node in a higher-dimensional tree has exactly two children $N_1 = n_1 - 1$, $N_2 = n_2 - 1$ and $N = n - 1$ and clearly $|T| = 2D(T) - 2N$ which is the desired result. Since the result holds for trees of one leaf any tree where internal nodes have exactly two children can be built and the result holds for any 2-dimensional tree. □

Theorem 2.7. *A strict upper bound on the size of any BB[α] 2-dimensional tree with n leaves is given by*

$$D(T) \leq \frac{1}{H(\alpha)} n(\log n + 1) - n$$

where

$$H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha).$$

Proof. From Nievergelt we know that if T has N internal nodes then

$$|T| \leq \frac{1}{H(\alpha)}(n + N + 1)\log(n + N + 1) - 2(n + N)$$

and from lemma 2.6 we know that $|T| = 2D(T) - 2N$. Since $N = n + 1$ in a 2-dimensional tree we can use these values to get

$$2D(T) - 2N \leq \frac{1}{H(\alpha)} 2n \log(2n) - 4n + 2$$

and so

$$\begin{aligned} D(T) &\leq \frac{1}{H(\alpha)} n \log(2n) - 2n + 1 + N \\ &= \frac{1}{H(\alpha)} n(\log n + 1) - n. \end{aligned}$$

This completes the proof. □

Theorem 2.7 shows that the size coefficient in a maximum size BB[α] 2-dimensional tree converges to $H(\alpha)$ as the number of leaves increase. We see that $H(1/3) = \frac{\log 8}{\log(27/4)} \approx 1.08897$ and that $H(0.1) \approx 2.13222$ which agrees with the data plots generated by the algorithm.

The general case where $d > 2$ is harder to reason about. While it is clear that the maximum sizes of a d-dimensional BB[α] tree of n leaves is given by the recurrence used in the algorithm

$$F_d(n, \alpha) = F_{d-1}(n, \alpha) + F_d(\lceil \alpha n \rceil, \alpha) + F_d(\lfloor n - \alpha n \rfloor, \alpha),$$

under the assumption that the structures of 2 and higher dimensional maximum size trees are the same, the formula admits no obvious closed form. Calculating a precise upper bound for this case is thus beyond the scope of this paper. However we note that for the rather strictly balanced case where $\alpha = 1/3$ the size coefficients appear to decrease as dimensions increase and that the 2-dimensional upper bound is therefore a non strict upper bound for any dimension. For $\alpha = 0.1$ we note that even though the $d = 3$ and $d = 4$ cases have trees with higher size coefficient than the $d = 2$ case, outside of these two the coefficients again decrease as d increases.

3 Sizes of red-black higher-dimensional trees

In a red black tree no path can have length equal to twice the length of the paths in a fully balanced tree. Initially this may seem like a stronger restriction on size of trees than the one imposed by $BB[\alpha]$ trees however unlike $BB[\alpha]$ trees where the majority of the leaves are concentrated near the top of the tree, no such guarantee is given for red-black trees. Recall that the potential of a leaf to contribute to the size of a higher-dimensional tree increases the further from the root the leaf is. When discussing higher-dimensional red-black trees we will use the notation that $T[n, h, d]$ means a d-dimensional red-black tree with n leaves where every path from root to a leaf in the main tree contains exactly h black internal nodes. We shall refer to h as the black-height of T.

It is not as trivial to construct a maximum size red-black higher-dimensional tree as a BB[α] higher-dimensional tree. Attempting to construct some maximum size red-black higher-dimensional tree with n leaves, we will want to place some large portion n' of n leaves at a deep layer i. However blindly increasing the value for i might drastically lower the value of n', since the black-height may need to be increased facilitating an increase of leaves that must be stored above i. Finding a balance between these two parameters is the focus of this chapter.

We can however give a proof that shows a trivial upper bound on the size coefficients in red-black higher-dimensional trees without taking the structure of these into account. If we use the property that a red-black tree of n leaves has at most $2 \log n$ layers, we can just take the sum of an upper bound on the sizes for each layer.

3.1 A trivial upper bound on sizes of red-black higher-dimensional trees

Lemma 3.1. *For all $d \geq 2$ the size, $D(T)$ of any red-black higher-dimensional tree $T[n, h, d]$ is bounded by $D(T) < 2^{d-1} n \log^{d-1} n$.*

Proof. For $d = 2$, it suffices that no path in T can have length greater than $2 \log n$ and therefore each of the n leaves can contribute no more than $2 \log n$ to $D(T)$.

For $d > 2$, assume the lemma holds for $d - 1$. Define the i'th layer in T to be all the nodes that have distance exactly i from the root, including the root itself. Again no path can be longer than $2 \log n$ and there can therefore at most be $2 \log n$ layers. The size of each layer will be given by $\sum_{n_j} D(n_j)$ where $\sum_{n_j} n_j \leq n$. By the induction hypothesis, $D(n_j) \leq 2^{d-2} n_j \log^{d-2} n_j$. Each layer therefore contributes less than $2^{d-2} n \log^{d-2} n$ to the size of T and the total size of T is bounded by

$$D(T) \leq 2 \log n \cdot 2^{d-2} n \log^{d-2} n = 2^{d-1} n \log^{d-1} n.$$

This concludes the proof. □

However it is not immediately clear how tight this bound is. To investigate we will gather data on size coefficients and structures of actual maximum size red-black higher-dimensional trees by proposing and implementing an algorithm for computing these values. The algorithm will build maximum trees by comparing all red-black higher-dimensional trees that have a possibility to be maximum in size. We will show that for a given number of leaves, the number of trees that have this possibility is small. The output from this algorithm will be size coefficients for red-black higher-dimensional trees that we compare against similar BB[α] higher-dimensional trees, as well as information about the structure of maximum size red-black higher-dimensional trees that will be used to prove bounds on the size coefficients when the number of leaves in trees grow to very large values.

3.2 Dynamically building maximum size higher-dimensional trees

We propose and implement a dynamic programming approach to construct actual maximum size higher-dimensional red-black trees. Any red-black higher-dimensional tree $T[n, h, d]$ can be constructed by creating a new black root node with an internal tree $T_i[n, h_i, d-1]$ and two sub-tree children $T_1[n_1, h-1, d]$ and $T_2[n_2, h-1, d]$ where $n = n_1 + n_2$. Such a tree is clearly a valid red-black higher-dimensional tree if the components it is made up of are valid red-black higher-dimensional trees and sub-trees and its size is given by

$$D(T) = D(T_i) + D(T_1) + D(T_2).$$

We will show that any such tree of n leaves that is maximum size will need to consider no more than $4n$ candidate triplets and that an algorithm that builds all maximum size trees with up to some n leaves can have all candidates calculated in advance for each tree.

3.3 Maximum size red-black higher-dimensional tree candidates

We show that to construct a maximum size higher-dimensional red-black tree by joining two existing red-black higher-dimensional trees with a new root, only two sets of tree candidates need to be considered: τ, the set of maximum size trees compared to all other trees with same black-height and number of leaves and ρ, the set of maximum size red-rooted sub-trees compared to all other red-rooted sub-trees with same black-height and number of leaves. Formally defined:

Definition 3.2. *A tree $T[n, h, d] \in \tau$ if for any tree $T'[n, h, d]$ it holds that $D(T) \geq D(T')$.*

Definition 3.3. *A red-rooted sub-tree $T[n, h, d] \in \rho$ if for any red-rooted sub-tree $T'[n, h, d]$ it holds that $D(T) \geq D(T')$.*

Note that any tree T can only belong to τ if it is a proper tree with root coloured black. It can only belong to ρ if it is a sub-tree with root coloured red. It can obviously never belong to both sets.

Lemma 3.4. *For any tree or sub-tree $T[n,h,d] \in \tau \cup \rho$ with sub-tree children $T_1[n_1,h_1,d]$ and $T_2[n_2,h_2,d]$ and internal tree $T_3[n,h_3,d-1]$ in the root, it holds that $T_1,T_2,T_3 \in \tau \cup \rho$.*

Proof. The size of T is given by:

$$D(T) = D(T_1) + D(T_2) + D(T_3).$$

Now assume $T_1 \notin \tau \cup \rho$. Then by definition 3.2 and 3.3: $\exists T_1^{'}[n_1,h_1,d]$ where $D(T_1^{'}) > D(T_1)$ and therefore $\exists T^{'}[n,h,d]$ where

$$D(T^{'}) = D(T_1^{'}) + D(T_2) + D(T_3) > D(T).$$

This contradicts $T \in \tau \cup \rho$ and therefore it must hold that $T_1 \in \tau \cup \rho$. A similar argument shows $T_2, T_3 \in \tau \cup \rho$. □

By Lemma 3.4 a tree $T[n,h,d] \in \tau$ can constructed if a $T_i[i,h-1,d] \in \tau$ and a $T_i^{'}[i,h-1,d] \in \rho$ is known for each $1 \leq i < n$ and a $T^{'}[n,h_3,d-1] \in \tau$ is known.

Similarly a sub-tree $T[n,h,d] \in \rho$ can be found if a $T_i[i,h,d] \in \tau$ is known for each $1 \leq i < n$ and a $T_3[n,h_3,d-1] \in \tau$ is known. Knowing more than one tree for any i from either of the two sets is unnecessary, since they will by definition all have the same size. Neither will the size be affected if the position of the two children are swapped. T can be found in linear time in n since there is only one candidate for the contained tree in the root, and for each i there are four possible candidate pairs. The number of leaves of one child sub-tree will be i and the other $n-i$. Let $T_i[n_i,h-1,d] \in \tau$ and $R_i[n_i,h-1,d] \in \rho$. The four possible candidate pairs are then: $\{T_i, T_{n-i}\}$, $\{T_i, R_{n-i}\}$, $\{R_i, T_{n-i}\}$ and $\{R_i, R_{n-i}\}$. T is found by trying the $O(n)$ candidate pairs and leeping track of which pair yielded the largest size.

Similarly a tree $T[n,h,d] \in \rho$ can be found in linear time in n by trying two candidate pairs from τ for each $i < n$.

3.4 An algorithm for constructing maximum size red-black higher-dimensional trees

This gives rise to algorithm 1 that given values N and D will compute the sizes of all maximum size red-black higher-dimensional trees with 2 to N leaves and dimension 1 to D.

The algorithm consists of three nested loops and a quick preprocessing phase that calculates the 1-dimensional case. The outermost loop loops over D, the middle loops over $\log N$ and the innermost over N. At the d,h,n'th state the algorithm first computes a maximum size red-black higher-dimensional tree $T_b[n, h, d]$ with black root and then a maximum size red-black higher-dimensional red rooted sub-tree $T_r[n, h, d]$ by comparing the sizes achieved by $O(N)$ different candidate triplets. Anytime a new tree $T_b[n, h, d]$ is computed it is tested against $T_{max}[n, h_{max}, d]$ which is the current highest computed size of any d-dimensional red-black higher-dimensional tree with n leaves and whatever black-height yielded the largest size and T_{max} is updated if needed.

To compute T_b the internal tree candidate is given by $T'[n, h', d-1]$ and will have already been calculated at the $d-1$'th step of the algorithm or the preprocessing phase. Let T_i be one of the two remaining candidates in the triplet, then T_i must be given by $T[i, h-1, d]$ and be either red or black rooted and will have been computed in the previous iteration of the middle loop.

To compute T_r the internal tree candidate has again been calculated at a previous outermost step or the preprocessing state. Let T_i be one of the two remaining candidates in the triplet and note that T_i is given by $T_i[i, h, d]$ and must have black root. This tree was calculated just prior in the algorithm.

We conclude that after the d'th iteration of the outermost loop all trees with dimension up to d and N leaves will have been calculated and that
$T_{max}[n, h_{max}, d]$ for any $2 \leq n \leq N$ will be a maximum size tree, and at the end of the entire algorithm all maximum size trees within the parameters D, N will have been computed.

To analyze the time cost of this algorithm we note that the outermost loop in the algorithm iterates D times, at each iteration values are calculated for $O(N \log N)$ trees where the computation for each tree takes $O(N)$ operations. This gives the algorithm a total running time of $O(DN^2 \log N)$.

A few small optimizations are used in the actual implementation of the algorithm, for instance no tree of dimension d and black-height h will be constructed from any trees with dimension $d' < d-1$ or with $d' = d$ and $h' < h-1$ and so these trees need no longer be stored. Also for trees with a given black-height h, no tree can exists with n leaves where $n < 2^h$ or $n > 2^{2h}$.

Algorithm 1 Maximum sizes of higher-dimensional red-black trees

```
procedure MaximumSize(N,D)
    Initialize all entries in $B, R, T_{max}$ to 0.
    for $n = 1$ to $N$ do
        $T_{max}[n, 1] \leftarrow n$
    for $d = 2$ to $D$ do
        $B[2, 1, d] \leftarrow T_{max}[2, d-1]$
        $B[3, 1, d] \leftarrow T_{max}[3, d-1] + T_{max}[2, d-1]$
        $B[4, 1, d] \leftarrow T_{max}[4, d-1] + T_{max}B[2, d-1] + T_{max}[2-d]$
        for $h = 2$ to $\log N + 1$ do
            for $n = 4$ to $N$ do
                for $i = 2$ to $n-2$ do
                    $B[n, h, d] \leftarrow Max\{B[n, h, d], T_{max}[n, d-1] + B[i, h-1, d] + B[n-i, h-1, d]\}$
                    $B[n, h, d] \leftarrow Max\{B[n, h, d], T_{max}[n, d-1] + B[i, h-1, d] + R[n-i, h-1, d]\}$
                    $B[n, h, d] \leftarrow Max\{B[n, h, d], T_{max}[n, d-1] + R[i, h-1, d] + R[n-i, h-1, d]\}$
                    $B[n, h, d] \leftarrow Max\{B[n, h, d], T_{max}[n, d-1] + R[i, h-1, d] + R[n-i, h-1, d]\}$
                    $R[n, h, d] \leftarrow Max\{R[n, h, d], T_{max}[n, d-1] + B[i, j, d] + B[n-i, h, d]\}$
                    $T_{max}[n, d] \leftarrow Max\{T_{max}[n, d], B[n, h, d]\}$
      return $T_{max}$
```

3.5 Red-black higher-dimensional maximum size coefficients in practice

In this section we study actual maximum size coefficients computed by Algorithm 1. We reason about the tightness of the upper bound in Lemma 3.1 and compare these results to the coefficients of BB[α] higher-dimensional trees obtained in section 2.2.

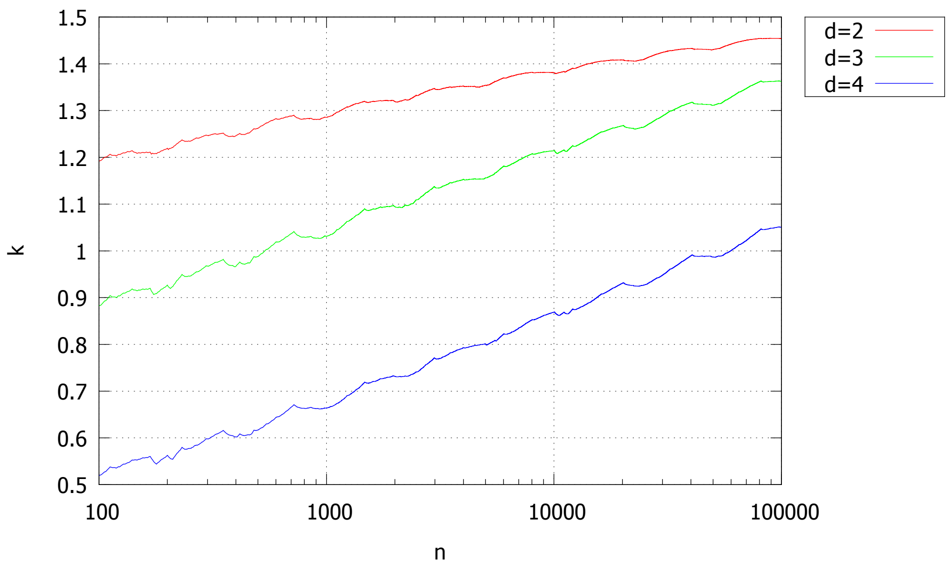

Figure 4: Size coefficients k of maximum size red-black 2,3 and 4-dimensional trees with n leaves on logarithmic scale.

Figure 4 shows coefficient values k of maximum size 2,3 and 4-dimensional red-black trees as a function of the number of leaves in the trees. We see that the growth of k appears to be decreasing and if k approaches the trivial upper bound 2 for the $d = 2$ case then it will most likely require trees with very large numbers of leaves to verify. We also note that contrary to the trivial upper bounds which increase as the dimensions of red-black trees increase, the coefficient values for all maximum size 3-dimensional red-black trees are lower than their 2-dimensional trees for every number of leaves in the range [100 : 100000]. We note that the coefficients are all still growing as the number of leaves increases. We will use these observations to motivate reinvestigating upper bounds on size coefficients for red-black higher-dimensional trees.

Finally we see that for practical values of d and n red-black d-dimensional trees of n leaves do not compare particularly favorable against BB[α] implementations. For the case where $\alpha = 1/3$ coefficients are marginally better for all values of d and n, and the discrepancies only increases with n. Even for a relaxedly balanced tree with $\alpha = 0.1$ the red-black tree is not much more than a factor 2 better and may not even outperform it for larger values of n.

3.6 Structures of maximum size red-black 2-dimensional trees

In order to further investigate maximum sizes of higher-dimensional red-black trees, we will try to determine the structure of such trees. Initially we reason about the simple 2-dimensional case and work up to the general d-dimensional case. First we show that any maximum size red-black 2-dimensional tree can be expressed using three critical parameters, and that these parameters can be found using Algorithm 1.

It is evident that the size contribution of a leaf in a red-black 2-dimensional tree is exactly the number of ancestors of the leaf. Let v be any node in a tree T, if there are ancestor nodes of v that have sub-tree children (excluding v) that are not minimally saturated as well as descendent nodes of v that have sub-tree children (again, excluding v) that are not maximally saturated then the size of T can be trivially increased by moving nodes around. Thus any maximum size 2-dimensional tree T must have a node v where each ancestor of v has one all-black sub-tree child and the sub-tree of v is as maximally saturated as the number of leaves in T permit.

We will formalize these possible candidates for maximum size red-black higher-dimensional trees, based on three parameters, the number of all black sub-trees on the path above v, the length a maximum path in the sub-tree of v and finally the number of leaves missing before the sub-tree of v is fully saturated.

Definition 3.5. *A tree T is a member of $\Gamma[H_1, H_2, \gamma]$ if T satisfies the following four requirements:*

- *1: The longest path P from the root to a leaf in T has length exactly $H_1 + H_2$.*
- *2: The H_1 internal nodes closest to the root (including the root itself) on P have one all-black sub-tree as child.*
- *3: The H_1'th internal node closest to the root on P has one sub-tree child containing $2^{H_2} - \gamma$ leaves.*
- *4: No value $H_1' > H_1$ exists such that the H_1' nodes closest to the root on P (including the root itself) have one all black sub-tree as child.*

Any maximum 2-dimensional red-black tree must be a member of $\Gamma[H_1, H_2, \gamma]$ for some value of H_1, H_2, γ and by the first and last requirement in definition 3.5 this triple of values must be unique. Note that two trees T_1, T_2 who are both members of $\Gamma[H_1, H_2, \gamma]$ will not necessarily have the same size unless $\gamma = 0$.

We will use Algorithm 1 to determine values of H_1, H_2, γ for trees of relatively few leaves in order to use these results for large trees. The hope is that these admit a pattern that can be generalized to trees with an arbitrary number of leaves. The procedure is as follows: First the algorithm builds all maximum size 2-dimensional trees with leaves up to N in time $O(N^2 \log N)$. Each tree $T[n, h, 2]$ is then traversed in the following way, we start by visiting the root node and at each visited node v the following step is performed: The left and right sub-tree child of v is traversed to determine whether it is all black, this can be done in time $O(2^h)$ keeping track of how many black nodes have been visited, h', and then verifying that no no path from the sub-tree root has length greater than $h - h'$ and no node on each path of length $h - h'$ from the sub-tree root is red. If exactly one sub-tree child is all-black then H_1 is increased by one and the sibling node of the all-black sub-tree root is visited. If both sub-trees are all black then H_1 is given the value of the path length of v to a an arbitrary leaf, as these paths are all same length. If neither sub-tree is all black then each node in the sub-tree of v is visited and the length of the longest path l_2 and the number of leaves n_2 is counted. H_2 then gets the value l_2 and γ gets the value $2^{H_2} - n_2$.

For each tree T we will attempt to discover if at most $O(h)$ sub-trees are all black, each requiring us to visit $O(2^h) nodes$, we will visit all nodes in one potentially large sub-tree of up to $O(n)$ nodes and thus the time to determine parameters for all N trees after is bounded by $O(N^2 \log N)$.

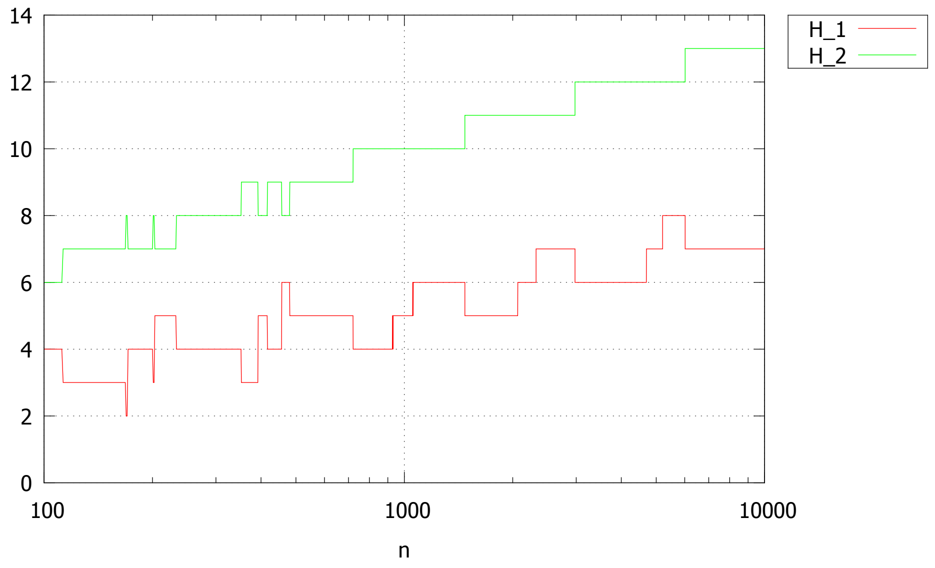

Figure 5: Values for H_1 and H_2 for maximum size 2-dimensional trees with n leaves on a logarithmic scale.

Using this procedure Figure 5 is generated showing values of H_1 and H_2 together with size coefficients for maximum size red-black higher-dimensional trees of up to 10000 leaves on a logarithmic first-axis. The values fluctuate quite a bit, however two tendencies seem to exist for H_1, H_2 pairs, they both appear to be growing logarithmically in the number of leaves in the tree and the gap between them appears to be widening.

We are interested in trees of the form $\Gamma[H_1, H_2, 0]$ as the sizes of these are easily calculatable even for very large trees. If such maximum size trees exist and if it can be determined for which values of H_1, H_2, then maximum size 2-dimensional trees with a near arbitrairy number of leaves can be constructed and data on how strict the trivial bound established is can be collected.

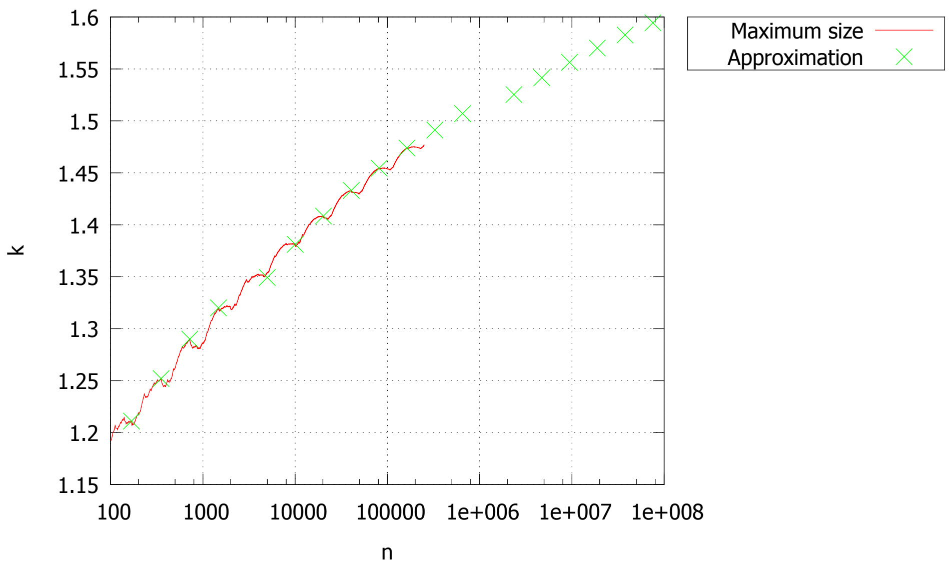

Figure 6: Approximated size coefficients compared to actual maximum size coefficients for trees with n leaves on a logarithmic scale.

The sizes of trees with different values for H_1, H_2 are tested against the size of actual maximum size trees, and a best fit is selected. It turns out that the approximation $\Gamma[h + \lceil \log h \rceil, h - \lceil \log h \rceil, 0]$ yields very promising results. In fact, of the first ten 2-order trees having this structure, only one is not a maximum size tree for its number of leaves. Figure 6 shows the size coefficients of these approximate maximum size trees plotted against the size coefficients of actual maximum size trees.

3.7 Investigating the tightness of the upper bound for size coeffients of 2-dimensional red-black trees

Lemma 3.6. *For any constant $\delta > 0$ there exists values n and h such that a 2-dimensional red-black tree $T[n, h, 2]$ exists where $D(T) \geq (2 - \delta)n \log n$.*

Proof. Let $T[n, h, 2] \in \Gamma[H_1, H_2, 0]$ where $H_1 = h - \lfloor \log h \rfloor, H_2 = h + \lfloor \log h \rfloor$ and let h be a power of 2. We will prove the result by showing that for a large enough h, a lower bound on $D(T)$ will outgrow an upper bound on $(2-\delta)\log n$.

First we note that T has n_1 leaves in the sub-trees of the top H_1 all-black sub-trees where n_1 is given by

$$n_1 = \sum_{i=1}^{\lceil H_1/2 \rceil} 2^{h-i} + \sum_{i=1}^{\lfloor H_1/2 \rfloor} 2^{h-i}.$$

Where the first sum is the contribution from the all black sub-tree children of the black nodes on the longest path P and the second sum is the contribution from the all black sub-tree children of the red nodes on P. This is a sum of two geometric series and therefore

$$n_1 \leq 2 \cdot 2^{h-1} + 2 \cdot 2^{h-1} = 2^{h+1}.$$

The number of leaves not located in an all-black sub-tree child of a node on P is given by $n_2 = 2^{H_2}$ by the definition of Γ. Thus the total number of leaves in T is bounded by

$$2^{H_2} \leq n \leq 2^{H_2} + 2^{h+1}.$$

A lower bound on the size of T is given by only considering the size contribution from the n_2 leaves in the maximum saturated sub-tree child of the H_1'th node on P, so clearly $D(T) \geq (2h)2^{H_2}$.

Now let $D'(T)$ be the right hand side of the inequality in the initial assumption, then

$$D'(T) = (2-\delta)n \log n.$$

Inserting the upper bound for n yields

$$D'(T) \leq 2(2^{h+1} + 2^{H_2}) \log(2^{h+1} + 2^{H_2}) - \delta n \log n,$$

for $h \geq 2$ which also implies $H_2 \geq h+1$, so

$$\begin{aligned} D'(T) &\leq 2(2^{h+1} + 2^{H_2})(H_2 + 1) - \delta n \log n \\ &= 2(2^{h+1})(H_2+1) + 2(2^{H_2})(H_2+1) - \delta n \log n. \end{aligned}$$

Split D' into parts so

$$D'(T) \leq D'_1 + D'_2 - \delta n \log n$$

where

$$\begin{aligned} D'_1 &= 2(H_2+1)2^{H_2} \\ D'_2 &= 2(H_2+1)2^{h+1}. \end{aligned}$$

To complete the proof, it must be shown that there exists a h large enough that subtracting the lower bound on $D(T)$ from the upper bound on $D'(T)$ yields zero or less. Now

$$D'(T) - D(T) \leq (D'_1 - D(T)) + D'_2 - \delta n \log n.$$

Inserting the value for H_2 into $D'_1 - D(T)$ yields

$$\begin{aligned} D'_1 - D(T) &\leq 2(h + \log h + 1)2^{H_2} - (2h)2^{H_2} \\ &= 2(\log h + 1)2^{H_2}. \end{aligned}$$

Using the value of H_2 in D'_2 yields

$$\begin{aligned} D'_2 &= \frac{4}{h}(H_2 + 1)2^{H_2} \\ &= (4 + \frac{4 \log h}{h} + \frac{4}{h})2^{H_2}. \end{aligned}$$

Finally, using the lower bound for n

$$\delta n \log n \geq \delta(h + \log h)2^{H_2}$$

will outgrow both $(D'_1 - D(T))$ and D'_2 as h increases so $D'(T) - D(T) \leq 0$ for sufficiently large h. This completes the proof. □

3.8 Structures of maximum size higher-dimensional red-black trees

Having proven tightness of bound for the 2-dimensional case, the general case is now considered. Note that it is not trivial to show that the same rules that apply to the structure of 2-dimensional trees apply to higher-dimensional trees. Indeed the size contribution of a leaf in a higher-dimensional tree is not linearly dependent on the number of ancestors the leaf has, but also depends on the structures of the internal trees in these ancestors. Therefore it is not clear that moving leaves further down a tree will always increase the size of the tree or whether there are sensible Γ representations for these.

However the procedure used to generate parameters H_1, H_2, γ can also be used in the d-dimensional case and trees with up to N leaves and up to D dimension can be calculated in time $O(DN^2 \log N)$.

Figure 7 shows that the values for H_1 and H_2 for maximum size trees of dimension $2, 3$ and 4 are nearly identical in the plotted interval. We will therefore attempt to use the maximum size approximation from the 2-dimensional size bound to improve on the upper bound for the d-dimensional case.

3.9 Improving the upper bound for maximum size d-dimensional trees

The proof requires us to bound the sizes of the internal $(d-1)$-dimensional trees. To do this we need the size coefficients of maximum size higher-dimensional red-black trees to be growing monotonically. This however is not a trivial result to prove or disprove and is beyond the scope of this thesis. Instead we will assume it to be the case and note that plotted data suggest this.

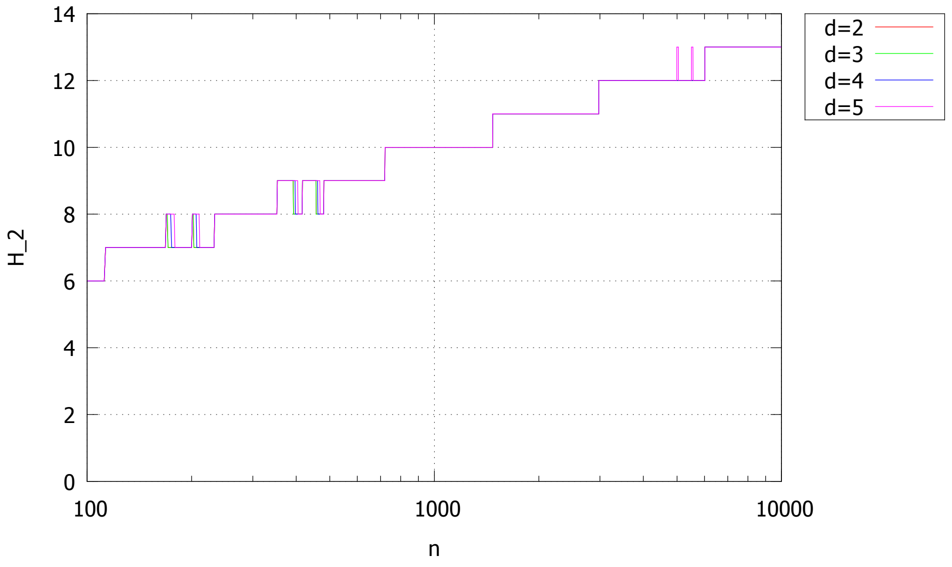

Figure 7: Values for H_1 and H_2 for trees with n leaves and dimension d on a logarithmic scale.

Assumption 3.7. *If for some $\delta > 0$ there exists a red-black tree $T[n,h,d]$ such that $D(T) \geq (k-\delta)n\log^{d-1} n$ then there exists an N and for any $n' \geq N$ there exists a tree $T'[n',h',d]$ where $D(T') \geq (k-\delta)n\log^{d-1} n$.*

Using the approximation of maximum size trees from the 2-dimensional proof and Assumption 3.7 we will show that the upper bound on maximum size coefficients of d-dimensional red-black trees can only increase with d.

Lemma 3.8. *If for any constant $\delta_d > 0$ there exists an n_d and h_d such that a d-dimensional red-black tree $T_d[n_d,h_d,d]$ exists where $D(T_d) \geq (k - delta_d)n_d\log^{d-1} n_d$ and Assumption 3.7 is true then for any constant $\delta > 0$ there exists an n and h such that a $(d+1)$-dimensional red-black tree $T[n,h,d+1]$ exists where $D(T) \geq (k-\delta)n\log^d n$.*

Proof. As in the proof for Lemma 3.6, let $T[n,h,d+1] \in \Gamma[H_1,H_2,0]$, $H_1 = h - \lfloor \log h \rfloor$, $H_2 = h + \lfloor \log h \rfloor$ and let h be a power of 2. The number of leaves in T is again bounded by

$$2^{H_2} \leq n \leq 2^{H_2} + 2^{h+1}.$$

Let v be any of the the H_1 largest nodes in T. Per the assumptions in the lemma, h can be picked large enough that there exists a tree $T_i[i,h',d]$ with

$$D(T_i) \geq (k-\delta')i\log^{d-1} i,$$

for all $i \geq 2^{h+\log h} = 2^{H_2}$. Therefore the d-dimensional internal tree in v can be picked such that

$$D(v) \geq (k - \delta')2^{H_2} \log^{d-1} 2^{H_2},$$

provided that $k \geq \delta'$. The size of T is then bounded by

$$\begin{aligned} D(T) &\geq H_1(k - \delta')2^{H_2} \log^{d-1} 2^{H_2} \\ &= (k - \delta')H_1(H_2)^{d-1}2^{H_2}. \end{aligned}$$

Let D' be the right hand side in the lemma, to complete the proof it must be shown that $D(T) - D' \leq 0$. Using the upper bound for n yields

$$D' \leq k(2^{H_2} + 2^{h+1}) \log^d(2^{H_2} + 2^{h+1}) - \delta n \log^d n,$$

which, if $h \geq 2$ is

$$\leq k(2^{H_2} + 2^{h+1}) \log^d(2^{H_2} \cdot 2) - \delta n \log^d n$$

$$= k(2^{H_2} + 2^{h+1})(H_2 + 1)^d - \delta n \log^d n.$$

Split D' into parts where

$$D' \leq D'_1 + D'_2 - D'_3$$

and

$$\begin{aligned} D'_1 &= k(H_2 + 1)^d 2^{H_2} \\ D'_2 &= k(H_2 + 1)^d 2^{h+1} \\ D'_3 &= \delta n \log^d n. \end{aligned}$$

Subtracting $D(T)$ from D' yields

$$D' - D(T) \leq (D'_1 - D(T)) + D'_2 - D'_3.$$

To complete the proof, all that is needed is to show that D'_3 will outgrow both $D'_1 - D(T)$ and D'_2 as h becomes sufficiently large.

1) D'_3 will outgrow $D'_1 - D(T)$

Inserting value for H_1 into $D(T)$ gives

$$D(T) \geq k(h - \log h)(H_2)^{d-1}2^{H_2} - \delta' H_1(H_2)^{d-1}2^{H_2}.$$

Inserting values for H_2 into D_1' yields

$$\begin{aligned} D_1' &= k(h + \log h + 1)(H_2 + 1)^{d-1}2^{H_2} \\ &= kh(H_2+1)^{d-1}2^{H_2} + k(\log h + 1)(H_2+1)^{d-1})2^{H_2} \end{aligned}$$

which, using the binomial thorem, can be rewritten as

$$= kh \left(\sum_{i=0}^{d-1} \binom{d-1}{i} (H_2)^i \right) 2^{H_2} + k(\log h + 1)(H_2+1)^{d-1}2^{H_2}$$

and since $\binom{d-1}{d-1} = 1$ becomes

$$\begin{aligned} &= kh \left((H_2)^{d-1} + \left(\sum_{i=0}^{d-2} \binom{d-1}{i} (H_2)^i \right) \right) 2^{H_2} \\ &+ k(\log h + 1)(H_2+1)^{d-1}2^{H_2}. \end{aligned}$$

Finally subtracting $D(T)$ from D_1' gives

$$\begin{aligned} D_1' - D(T) \le\, & kh \left(\sum_{i=0}^{d-2} \binom{d-1}{i} (H_2)^i \right) 2^{H_2} \\ &+ k(\log h + 1)(H_2+1)^{d-1}2^{H_2} \\ &+ k(\log h)(H_2)^{d-1}2^{H_2} \\ &+ \delta' H_1 (H_2)^{d-1} 2^{H_2}. \end{aligned}$$

Now to show that D_3' will outgrow each of these four terms to an arbitrairy factor. $D_3' = \delta n \log^d n \ge \delta 2^{H_2}(H_2)^d$ has as coefficient a polynomial of H_2 with degree d with coefficients depending only on the constants d and δ. The first three terms are of the order no larger than $\log h H_2^{d-1}$ with coefficients dependent only on the constants k and d, clearly these will be outgrown arbitrarily by a d order polynomial. The last term is also a polynomial in h of order d, but it is clearly inferior to D_3' for any $\delta' < \delta$. Since the size of δ' can be picked arbitrarily and independently of δ if h is sufficiently large, the last term can also be outgrown arbitrarily.

2) D_3' will outgrow D_2'

Inserting value for H_2 into D_2'

$$D_2' = (h + \log h + 1)(H_2+1)^{d-1}2^{h+1}$$

using the value of H_2 to change the exponent

$$= \frac{2}{h}(h + \log h + 1)(H_2 + 1)^{d-1} 2^{H_2}.$$

A similar argument to the one in 1) shows that the coefficient of D'_2 is a polynomial of order $H_2^{d-1} \log h$ with coefficients dependent only on the constants d and k and will be outgrown by D'_3 for sufficiently large h. This completes the proof. □

We now have the Lemmas to show our main result on maximum sizes of red-black higher-dimensional trees. An upper and lower bound on sizes.

Theorem 3.9. *The size of any red-black d-dimensional tree of n leaves where $d > 1$ is bounded by $\frac{1}{(d-1)!} n \log^{d-1} n < D(T) < kn \log^{d-1} n$ where k is at least 2 and at most 2^{d-1}.*

Proof. The lower bound is given by Lemma 2.5 provided Assumption 2.3 is true, the lower bound for k is given by Lemma 3.6 together with Lemma 3.8 provided Assumption 3.7 is true. The upper bound on k is given by lemma 3.1. □

3.10 Approximating large maximum size 3-dimensional red-black trees

As a final note on the sizes of red-black higher-dimensional, all it takes to increase the value of k in the upper bound is to find one example of a d-dimensional tree with size coefficient higher than $k' > 2$ and every red-black tree of dimension greater than d will have coefficient at least k' provided Assumption 3.7 is true. We will show that constructing such a tree is most likely not possible using any tree $T[n, h, d]$ of the form $\Gamma[h - \lfloor \log h \rfloor, h + \lfloor \log h \rfloor, 0]$. To do this we give a way to calculate efficiently a lower bound on the sizes of such 3-dimensional trees for very large values of h.

First we note that the size of any 2-dimensional red-black tree in some $\Gamma[H_1, H_2, 0]$ can be calculated by summing $H_1 + 1$ sub-tree sizes that can each be calculated as a simple function without having to traverse the sub-tree. This is the case since the closest H_1 nodes on the longest path will have one all-black sub-tree child and the remaining sub-tree in the H_1'th node will be maximally saturated. 2-dimensional trees where $\gamma > 0$ cannot be calculated in this way and if H_2 is large the computation will get prohibitively expensive. Unfortunately the 3-dimensional trees that we are approximating will not necessarily have internal 2-dimensional trees of a number of leaves that permit γ to be 0. However if we instead of calculating the sizes of these internal trees, we only calculate the sizes of 2-dimensional trees T_h on the form $\Gamma[h - \lfloor \log h \rfloor, h + \lfloor \log h \rfloor, 0]$, we can calculate a reasonable lower bound for any maximum size 2-dimensional tree of n leaves by using the tree T_h of n_h leaves where $n - n_h$ is minimized and still positive. A reasonable lower bound on T is then given by $D(T_h) + (n - n_h)H_1$. Similarly we can calculate a reasonable upper bound on the size of T by finding a T_h of n_h leaves such that $n_h - n$ is minimized and positive and using the size approximation $D(T_h) - (n_h - n)H_1$.

In a 3-dimensional tree $T[n, h, 3]$ on the form $\Gamma[h - \lfloor \log h \rfloor, h + \lfloor \log h \rfloor, 0]$ the majority of the size contribution will be from the H_1 large nodes near the root if h is very large, if we calculate the size of the approximate maximum size 2-dimensional tree in each of these we get a lower bound on the size of T where the size coefficient hopefully grows large and that can be calculated by computing sizes of only $O(h)$ internal tree approximations.

Unfortunately this approach leads to no trees with a size coefficient higher than or equal to 2, even when the large approximations are used for the 2-dimensional trees the size approximation of a 3-dimensional tree with black-height $h = 8192$ and $n > 2^{8205}$ leaves yield a coefficient of $k \approx 1.99026..$

This leads to several plausible conclusions: either trees $T[n, h, d]$ of the form $\Gamma[h - \lfloor \log h \rfloor, h + \lfloor \log h \rfloor, 0]$ are not strong enough to model maximum size higher-dimensional trees or the lower bound used on these is too low or the case where $d = 3$ is special and somehow equivalent to the 2-dimensional case or the maximum size coefficients of all $d > 1$-dimensional trees converge to exactly 2. Exploring any of these possibilities further is beyond the scope of this thesis however.

4 Worst case rebalancing cost in higher-dimensional trees

In this section we will compare worst case rebalancing costs between higher-dimensional trees implemented using a BB[α] and a red-black rebalancing scheme. A theorem by Lueker gives an amortized upper bound on rebalancing costs in higher-dimensional BB[α] trees, we will merely sketch this proof and refer to [8] for details. For red-black trees we will give a recipe for constructing a sequence of input and delete operations that has very high rebuilding cost, we will not show an actual strict upper bound but merely note that it is much larger than that of the BB[α] case.

4.1 Worst case rebalancing of BB[α] higher-dimensional trees

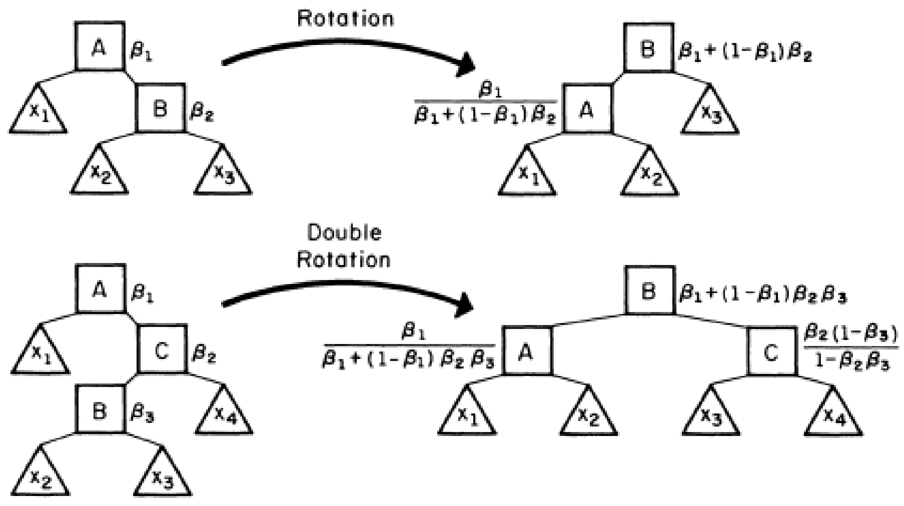
Rotation
A
β_1
B
β_2
x_1
x_2
x_3
B
$\beta_1 + (1-\beta_1)\beta_2$
$\frac{\beta_1}{\beta_1 + (1-\beta_1)\beta_2}$
A
x_1
x_2
x_3
Double Rotation
A
β_1
C
β_2
B
β_3
x_1
x_2
x_3
x_4
B
$\beta_1 + (1-\beta_1)\beta_2\beta_3$
$\frac{\beta_1}{\beta_1 + (1-\beta_1)\beta_2\beta_3}$
A
C
$\frac{\beta_2(1-\beta_3)}{1-\beta_2\beta_3}$
x_1
x_2
x_3
x_4

Figure 8: From Nievergelt and Reingold [9]. Rotations and double rotations in BB[α] trees.

Nievergelt and Reingold give the following rules for rebalancing a BB[α] tree in [9]. If after some insertion or deletion in a sub-tree with root v the weight constraint of v has been violated such that $w(v) < \alpha$ and v_r is the right child of v then a single rotation is performed on v if $w(v) < \frac{1-2\alpha}{1-\alpha}$ and a double rotation on v is performed othewise. If $w(v) > (1-\alpha)$ then the symmetrical variant of the rotation is performed and a single or double rotation will depend on the weight of v's right child. Blum and Mehlhorn show in [2] that performing this rebalancing will restore the weight of v to within its allowed range for any BB[α] tree where $\frac{2}{11} \leq \alpha \leq 1 - \frac{1}{2}\sqrt{2}$.

Lueker gives an upper bound on the cost of performing i mixed insert or delete operations on an initially empty BB[α] d-dimensional tree as $O(i \log^d i)$.

Theorem 4.1. *The rebalancing cost of inserting or deleting i points on an initially empty BB[α] d-dimensional tree where* $\frac{2}{11} \leq \alpha \leq 1 - \frac{1}{2}\sqrt(2)$ *is worst case* $O(i \log^d i)$.

Proof. We will sketch Lueker's proof here and refer to [8] for full details. First define $\beta(v)$ for a node v as the distance of $w(v)$ from the range $[1/3, 2/3]$. That is $\beta(v)$ is 0 if $w(v)$ is in the range and otherwise it is the smaller of the two values it takes to increase $w(v)$ to 1/3 or decrease $w(v)$ to 2/3. Let the imbalance $I(T)$ of a d-dimensional BB[α] tree T be defined as the sum of $\beta(x)2n' \log^{d'-1} i$ for each internal node v in the main tree or any internal tree in T where the sub-tree of v contains n' leaves and has dimension d'.The proof uses amortization and uses the decrease in $I(T)$ after a rebalance to cover the cost of the insertions. Lueker proves by induction that a single insert into T can increase the value of $I(T)$ by at most $O(log^d i)$. Although not explicitly stated in the paper, a similar argument holds for the increase after a delete operation. Lueker then proves that any time a rotation takes place at a node v'' in T with sub-tree of dimension d'' containing n'' leaves, $I(T)$ is decreased by at least $(\alpha - \frac{1}{3})2n'' \log^{d''} i$.

Since rebuilding a node v with sub-tree of dimension d containing n leaves can be done in time $O(n \log^{d-1} n)$ Lueker concludes that rebalancing covers the cost of the insert and delete operations if a suitable constant based on α is used to charge these. □

Thus, since path lengths in BB[α] trees are logarithmic in lengths in the number of leaves in the tree, a search in a BB[α] d-dimensional tree T of n leaves visits at most one internal tree for each node on any path it traverses we note that the time to search T is bounded by $O(log^d n)$. This is a remarkable property of BB[α] trees that even though a single rebuild operation may take as much as $O(n \log^{d-1} n)$ time, the amortized rebuilding time is no worse than the time it takes to search the tree.

4.2 Rebalancing cost of red-black higher-dimensional trees

Red-black trees are balanced by recoloring nodes and performing rotations. Figure 9 shows the rules used for rebalancing red-black trees in this paper. Note that rebalancing a red black tree of n leaves after an insert or delete operation requires at most one single- or double rotation and since path lengths in red-black trees are logarithmic, at most the recoloring of $O(\log n)$ nodes. We note that cases exist where both delete rule 4.1 and 4.2 can be used to correctly balance a red-black tree, however in this thesis we will have rule 4.1 take precedence over rule 4.2 unless otherwise stated. Refer to [5] for proof that these rules correctly rebalance any red-black tree after an insert or delete operation.

As mentioned in section 1 Bentley and Friedman show in [1] that a d-dimensional tree where path lengths differ by at most one for the main tree and each internal tree can be built from n points in time $O(n \log^{d-1} n)$. Such a tree can easily be be colored into a valid red-black tree if either it contains 2 or fewer leaves by coloring all internal nodes and leaves black or if this is not the case then by coloring all internal nodes on the lowest layer in the tree red and all other leaves and internal nodes black.

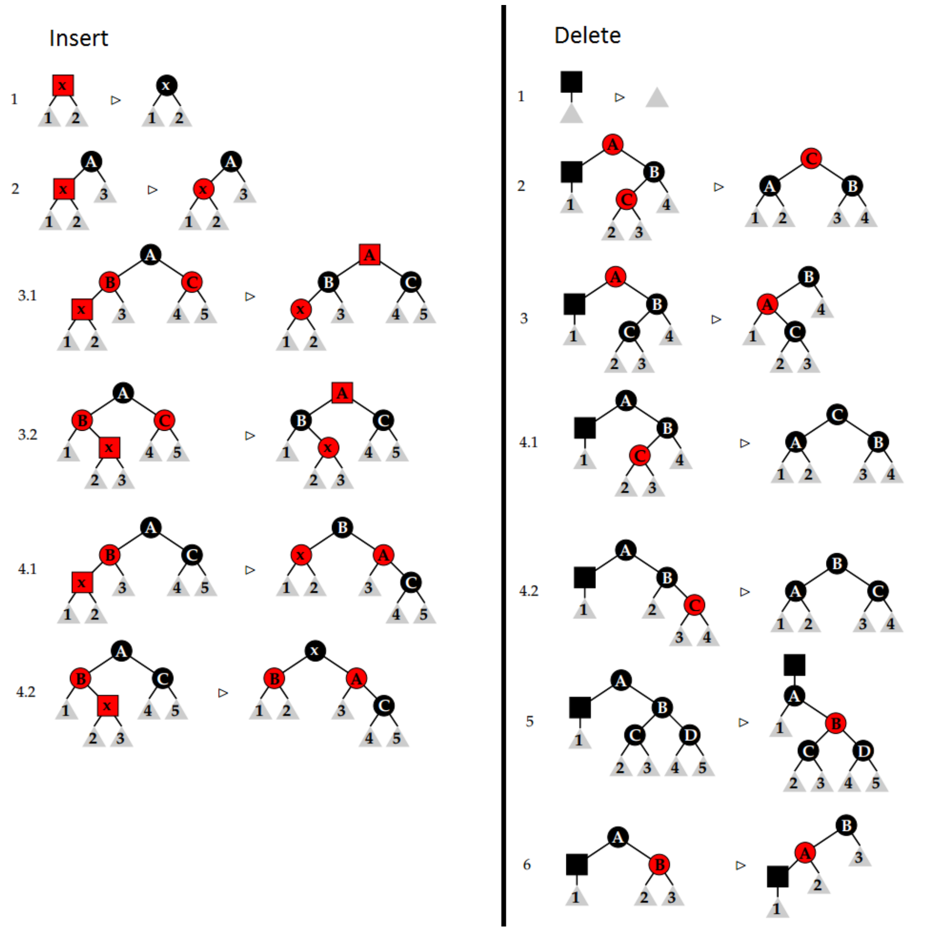

Figure 9: Visual representation of rebalancing rules for red-black trees from Hansen and Schmidt's Transition Systems [6]. Squares represent the node that is currently being rebalanced, if no squares are present in the result of a rebalancing rule then the rule terminates the rebalancing.

Unlike in a $BB[\alpha]$ higher-dimensional tree where an amortization argument shows that the time spent rebuilding trees can be paid for by the number of insert or delete operations needed to unbalance large nodes, no such argument is easily available to red-black trees. It is therefore of interest to consider sequences of insert and delete operations that require the rebuilding of many large internal trees. We will be working mostly with the main tree of the red-black higher-dimensional trees in this section, and when we refer to a point being larger or smaller than some other point, we mean in the context of the coordinate with the specific index that is being used to compare points in this main tree.

4.3 Constructing a red-black higher-dimensional tree with high rebuilding cost

We will show that if a red-black tree with a specific structure Y is constructed, then a sequence of k insert and delete operations into Y can be repeated indefinitely and each repetition will require rotation on a node with sub-tree containing a very large number of leaves.

Definition 4.2. *A a higher-dimensional red-black tree $T[n, h, d]$ is a member of Y if the main tree of T satisfies the following properties.*

- *1 The left sub-tree child of the root is all black.*
- *2 The right child of the root v_r is black and the right child of v_r is also black.*
- *3 The left sub-tree child of v_r is all black.*
- *4 The left and right sub-tree children of the right child of v_r are maximally saturated, with each path being alternating red-black.*

We will give a general recipe to construct trees of a chosen black-height h that are memberes of Y.

4.4 Constructing a red-black higher-dimensional Y tree

Since reasoning about the results of the different rebalancing rules leads to very many tedious cases, we will rely on results generated by our computer program that implements red-black trees using the rules specified in figure 9. We note that the implementation contains a function that given some tree validates that every restriction imposed by the red-black rebalancing scheme is indeed met. We have run this function on generated red-black trees containing hundreds of thousands of leaves and have thus far found no invalid trees.

The following sequence constructs a red-black tree of black-height h in Y. First the skeleton of the tree T_0 is created by inserting $2^{h-1} - 1$ increasing points into an initially empty red-black tree. Except for the first two inserts, all inserts will be rebalanced using some combination of the mirrors of insert rule 3.1 and 4.1. These two rules will only color nodes on the right-leaning path or the left child of the root red, and simulation shows that if the number of points inserted in this value is a power of 2 then the end result is a tree with one right-leaning black and red alternating path where each left sub-tree child on the path is all black. See figure 13 in appendix $A2$ for an example of a tree T_0 generated in this manner.

To construct the next step, T_1, we first insert decreasing points into the left sub-tree child of the root where each point p_i is smaller than the point stored in the left-most leaf in the sub-tree. This will eventually color all right siblings on the left most path in the sub-tree red, until rebalancing reaches the left child of the root's left child where rule 3.1 colors the root's right child black. This is done with less than 2^{h-2} inserts and afterwards the same number of points is deleted so that the left sub-tree child of the root is again all black. See figure 14 in appendix $A2$ for an example of a tree T_1 generated in this manner.

Note that we leave the right child of the root's right child black, which seems counterintuitive as it decreases the number of leaves in its sub-tree, however the reason is that otherwise both delete rule 4.1 and 4.2 would be able to balance the tree in the step where we perform the rotation that causes the large rebuilding. If the precedence of these rules were for instance chosen at random the structure of the tree could end up in a configuration where resetting it to be a member of Y required rebuilding it from scratch.

Finally, inserting values into the sub-tree of the root's right child's right child until every path in this sub-tree is alternating red-black and the sub-tree is maximally saturated generates the finished tree $T \in Y$ and the preprocessing ends. This can trivially be done by finding a leaf in the sub-tree that has a minimum number of ancestors and inserting a point that is slightly larger or smaller than the stored point in the leaf. See figure 15 in appendix $A2$ for an example of a tree T that is a member of Y.

Once the tree is completed, the following sequence of operations on Y will require costly rebuilding if Y is the main tree in a higher-dimensional red-black tree. Let v_r be the right child of the root's right child. The sub-tree of v_r contains 2^{2h-2} leaves and performing a left rotation on its parent will invalidate its internal tree causing very costly rebuilding.

4.5 Performing a high cost sequence of operations on a Y tree.

Let v_l be the left child of the root's right child and let p_l be the point in the left-most leaf in v_l. Inserting $2^{h-1} - 1$ decreasing points p_i where $p_i < p_l$ for each i into the sub-tree of v_l causes rebalancing by insert rule 3.1 and 4.2 and after the final insert the otherwise all black sub-tree will have an alternating red-black left leaning path and v_l will be red. An example of the resulting tree can be seen on figure 16 in appendix A2.

Now, deleting the point contained in the left-most leaf in y_1 triggers recoloring up the left-most path in the all black left sub-tree child of the root by delete rule 5. Eventually the rule reaches the left child of the root where no recoloring is possible. Instead a double rotation by delete rule 4.1 is performed. Note that the position of v_r in the tree remains unchanged after the double rotation, but the left sub-tree of its parent is replaced by the right sub-tree child of v_l and so its internal tree is invalidated and must be rebuilt. The result of this sequence is shown on figure 17 in appendix A2.

The new left child of the right child of the root, v_l' is now the old right sub-tree child of v_l which is an all black tree. The old left sub-tree child of v_l has been moved into the left sub-tree child of the left child of the root. Recall that this sub-tree, the old v_l had a red parent and was all black except for one red-black alternating path and as such it contains exactly $2^{h-1}-1$ leaves. Thus this sub-tree can be returned to an all black sub-tree of black-height $h-2$ by deleting $2^{h-2}-1$ points, and these points can be found by repeatedly deleting the point in a leaf in the sub-tree that has a maximum number of ancestors. Similarly the left sub-tree child of the left child of the root is all black with one red-black alternating path with one missing leaf, the one that was deleted to prompt the double rotation. This sub-tree has $2^{h-1}-2$ leaves and can thus be made all black in a similar way by deleting $2^{h-2}-2$ points. After the deletion of these points, the tree is structurally similar to the original tree Y, and the sequence can be repeated.

Theorem 4.3. *Under the assumption that our implementation of a red-black tree is correct, a d-dimensional red-black tree $T[n,h,d]$ that is a member of Y can be constructed in $O(n \log^d n)$ time and a sequence of insert and delete operations exists into T where the average cost per operation is $O(\sqrt{n} \log^d n)$.*

Proof. Building only the main structure of the tree requires inserting and deleting $O(2^h) = O(\sqrt{n})$ points a constant number of times before finally filling up a sub-tree with $O(2^{2h}) = O(n)$ inserts. This can trivially be done in $O(n \log n)$ time. After constructing the main tree we can use Bentley and Friedman's method to fill in the internal trees and the total time to construct the tree is at most $O(n \log^d n)$.

The number of leaves in the sub-tree of the root's right child is clearly $n_r = O(2^{2h}) = O(n)$, and so rebuilding its internal structure takes time $O(n \log^d n)$. After the tree has been constructed we insert $2^{h-1}-1$ points to recolor the root's right child's left child, then delete one point to trigger the rotation and delete 2^{h-1} points to restore the tree to its original structure. Thus the amortized time cost of performing an insert or delete in this sequence is $O(\sqrt{n} \log^d n)$. □

We end this section by discussing the merits of implementing
higher-dimensional trees with red-black rebalancing. We have regrettably not found any way to prove how wide the gap between a worst-case sequence of operations and the sequence in theorem 4.3 is. Indeed a much worse sequence may exist and it could be much more complex than simply performing a single rotation on a node with a large internal trees and then resetting the structure of the main tree. We note, however, that the performance cost of the found sequence is bad enough to render red-black higher-dimensional trees impractical for use in any real-world scenario and any such sequence would be interesting only as a theoretical result. Indeed the advantage of performing orthogonal range queries using higher-dimensional trees over for instance *kd*-trees are purely in the the lower costs of searching the tree as the space cost of storing higher-dimensional trees is an order of magnitude higher.

5 Conclusion

In this thesis we explored the two interesting properties of higher-dimensional trees, their space cost and their operations cost, implemented with a red-black balancing scheme and compared them with their well documented BB[α] counterparts. The main motivation for this research was the paper [8] by Lueker that describes an algorithm for efficiently performing dynamic orthogonal range queries using a higher-dimensional BB[α] tree where it was stated that the main limiting factor was the space cost increase as the dimensions of the trees increase. Since the more ancestors some leaf has the more internal structures its point can also be contained in it seemed natural to attempt to see if the relatively short path lengths in a red-black tree could alleviate this problem in any way. Another reason to investigate an alternative to BB[α] trees is that these seemed very complex to implement correctly and examples were given on mistakes in the original paper [2] as well as in commercial software [7].

We noted that asymptotically the two tree implementations had same space cost, but that the hidden coefficients would depend on the dimensions of the tree. We motivated our research into investigating these coefficients by showing that for fully balanced trees of dimension d the coefficient drops exponentially as d increases.

When comparing these coefficients of red-black ahd BB[α] higher-dimensional trees we first showed that the sizes of what we assumed were maximum size BB[α] higher-dimensional trees are about equal to those implemented with red-black rebalancing for trees with a number of leaves that is feasible to use in a real-world setting. Even when the value for α was chosen so that the trees would have long paths, the BB[α] versions still performed about as well as the red-black versions and the discrepancies decreased as the number of leaves increased. Indeed the size coefficients for red-black trees increased in the computed range as the number of leaves increase whereas the BB[α] trees were near constant or decreasing for large enough dimensions.

To generate values for maximum sizes of BB[α] trees we used the simple assumption that the structure of the main tree of a d-dimensional case is the same as that of the 2-dimensional case. This gave a simple recurrence that could be computed efficiently for trees with a decent number of leaves. We later implemented a more rigorous algorithm that computed maximum sizes without relying on any assumptions and noted that the two generated the same output for all tested values. Since we found that no simple structure of maximum size red-black higher-dimensional trees were readily available we had to rely on a rigorous algorithm to generate data.

In order to explore theoretical results we used patterns in the structures of the trees generated by the algorithm to construct a moderately successful approximation of maximum size red-black higher-dimensional trees. We used this approximation to show that coefficients of maximum size 2-dimensional red-black trees will get within any arbitrary constant of 2 but never reach it exact. Generalizing this approximation to d-dimensional trees we proved that given the assumption of monotonic growth of maximum size coefficients then the size of any red-black $d > 1$-dimensional tree with n leaves is bounded by $\frac{1}{(d-1)!} n \log^{d-1} n \leq D(T) < kn \log^{d-1} n$, where k is at least 2 and at most 2^{d-1}.

The assumption used was that if it holds that given some constant depending on the dimension of the tree, k', and a d-dimensional tree exists where the size coefficient is greater than $k' - \delta$ for any positive constant δ then there exists a number N such that a red-black higher-dimensional tree T_n with n leaves exist for any $n \geq N$ where the size coefficient of T is also higher than $k' - \delta$. No proof was presented for this, but we noted that the data plotted by the algorithm supported it being true.

We used the results of Nievergelt and Reingold to show that the upper bound on sizes of 2-dimensional BB[α] trees is about half of the red-black case for $\alpha = 1/3$ and slightly larger for $\alpha = 0.1$. As a final note on sizes of red-black trees we attempted to construct an approximation of a 3-dimensional tree where the size coefficient was above 2, but found no such tree even as the number of leaves in the tree became astronomically high.

On cost of performing operations in higher-dimensional trees we noted that for a single operation the dominating factor was potentially the time it takes to rebuild internal trees after rotations occur. A proof in Lueker showed that amortized this cost could be absorbed by the cost of the number of inserts and deletes it takes before the rotation is needed. We showed that no such cost absorption exists in red-black higher-dimensional trees and we gave a recipe for constructing a sequence of insert and delete operations of arbitrary length into a specific red-black higher-dimensional tree that makes rebalancing cost dominate all other operation costs for the series. This cost is a factor of $\sqrt(n)$ larger for a tree with n leaves than the BB[α] cost and we noted that even though the red-black higher-dimensional trees performed no better or worse to make any real difference than the BB[α] higher-dimensional trees in terms of space cost in the computed range, and that BB[α] trees are indeed complex to implement, red-black higher-dimensional trees turned out to have very little purpose in any real-world setting.

5.1 Future work

As the research done in this thesis indicates, red-black higher-dimensional trees have little use in any real-world sense. However a number of theoretical results may still be of interest. A number of assumptions are made in this thesis that, although supported by data, are not rigorously proven. For one we assume that the sizes of maximum size and minimum size higher-dimensional trees all follow the same structure as the 2-dimensional case, that is a minimum size tree is as close to perfectly balanced as possible and a maximum size tree has as many leaves as possible positioned as far down in the tree as possible. Verifying or discarding these assumptions may be an interesting result to pursue.

We also show that if these assumptions are indeed correct then the size coefficient of a maximum sized d-dimensional tree converges to some k that is at least 2 and at most 2^{d-1}. Examining how tight this bound is is another possible area of interest. The algorithm we give for constructing maximum size red-black higher-dimensional trees is super polynomial in the number of leaves, if a better algorithm could be found perhaps more information about the maximum sizes of red-black trees can be learned.

A final result on sizes of higher-dimensional trees that could be of interest is solving the recurrence presented for maximum sizes of BB[α] trees and uncovering a bound on the coefficients for these.

Finally we show that a sequence of inputs into a red-black higher-dimensional tree of n leaves exists that causes average rebalancing cost of $O(\sqrt{n}\log^d n)$. We do not prove whether a worse sequence exists, and finding an example of such or disproving it may also be a venue to explore.

6 Appendix

6.1 A1

Output generated from Algorithm 1 shows structures of maximmum size red-black 2-dimensional trees with $38, 39$ and 40 leaves. Circles represent internal nodes and boxes represent sub-trees where all paths have length h. A red box is a fully saturated sub-tree and a black box is a minimally saturated sub-tree.

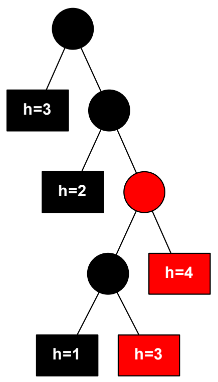

Figure 10: Maximum size red-black 2-dimensional tree with 38 leaves in $\Gamma[2,5,6]$.

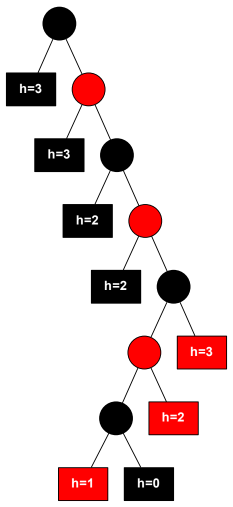

Figure 11: Maximum size red-black 2-dimensional tree with 39 leaves in $\Gamma[4,4,1]$.

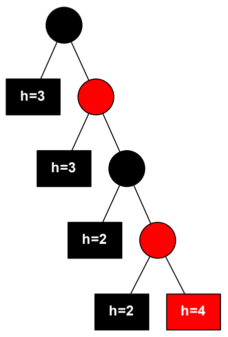

Figure 12: Maximum size red-black 2-dimensional tree with 40 leaves in $\Gamma[4,4,0]$.

6.2 A2

Various stages in generating trees in Y and performing the high rebalance cost sequence on these generated by a computer program implementing red-black trees.

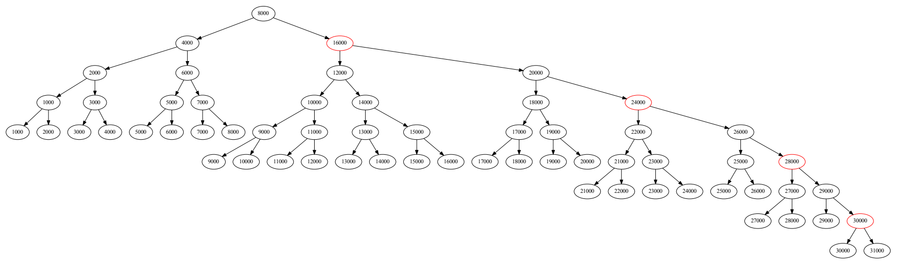

Figure 13: T_0, the initial stage in generating a tree in Y with black-height 5.

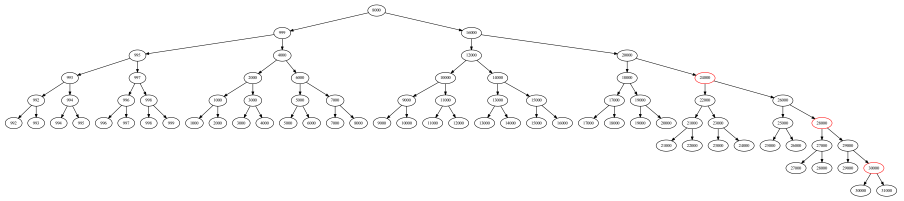

Figure 14: T_1, the second stage in generating a tree in Y with black-height 5.

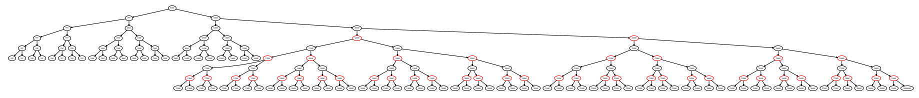

Figure 15: T, the final stage in generating a tree in Y with black-height 5.

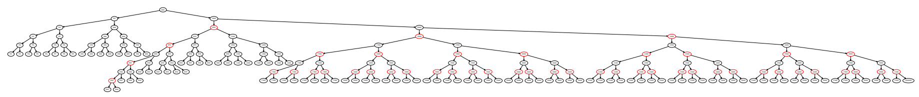

Figure 16: The result of performing insertions in the sub-tree of v_l in a tree in Y.

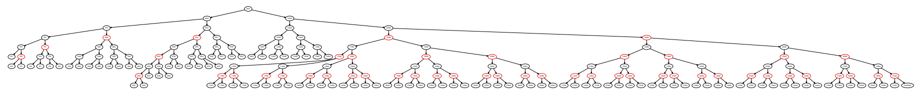

Figure 17: The result of deleting the smallest point in a tree and performing a double rotation that causes high rebuilding cost. Note that if excess points are deleted in the left sub-tree of the root then the tree returns to being in Y.

References

[1] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, December 1979.

[2] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theor. Comput. Sci.*, 11:303–320, 1980.

[3] John Horton Conway and Richard K Guy. *The book of numbers.* Copernicus, 1996.

[4] Mark de Berg, Marc van Kreveld, Mark Overmars, and OtfriedCheong Schwarzkopf. Computational geometry. In *Computational Geometry.* Springer Berlin Heidelberg, 2000.

[5] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:8–21, 1978.

[6] Mikkel Nygaard Hansen and Erik Meineche Schmidt. *Transition Systems: Algorithms and Data Structures.* Matematisk Institut, Aarhus Universitet, Datalogisk Afdeling, 2004.

[7] YOICHI HIRAI and KAZUHIKO YAMAMOTO. Balancing weight-balanced trees. *Journal of Functional Programming*, 21:287–307, 5 2011.

[8] George S. Lueker. A data structure for orthogonal range queries. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 28–34, Oct 1978.

[9] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

[10] C. K. Wong and J. Nievergelt. Upper bounds for the total path length of binary trees. *J. ACM*, 20(1):1–6, January 1973.