Balancing weighted strings and trees in linear time

Frédérique Carrere

To cite this version:

Frédérique Carrere. Balancing weighted strings and trees in linear time. 2012. hal-00708337v2

HAL Id: hal-00708337
https://hal.science/hal-00708337v2

Preprint submitted on 3 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Balancing weighted strings and trees in linear time

Frédérique Carrère[1]

LaBRI, Université Bordeaux I,
33405 Talence Cedex, France

Abstract

In the present paper, we address the problem of building a binary tree which leaves carry weights in a given order and which is in some sense balanced. Such a tree is denoted as a *balanced alphabetic weighted tree*. If the leaves of the tree are labeled with letters, their concatenation (from left to right) gives a weighted string. The tree is balanced if it minimizes the maximum of all root-to-leaf paths weights. The root-to-leaf path weight for a given leaf is the sum of its depth and a function of its weight. There already exist linear algorithms to balance alphabetic weighted trees, using some extra hypothesis on the weights (for example if the number of distinct integer parts of the weights is bounded). The algorithm presented here is an online algorithm, which uses neither sorting nor extra data structures nor extra hypothesis on the weights. It applies to positive weights (either integers or reals). We use this algorithm to balance binary trees representing graphs in linear time.

Keywords: weighted string, weighted alphabetic tree, weight-balance,
tree-decomposable graph

1. Introduction

Given a finite set of positive weights $w_1, \dots ,w_n$ (integers or reals), we are interested in algorithms constructing a binary tree which leaves hold the given weights in left-to-right order and which is in some sense "balanced" relatively to the weights. In this paper we only deal with binary trees and we refer to a binary tree which leaves hold weights as a *weighted tree*. We say that a weighted tree is balanced if it minimizes the maximum of all root-to-leaf paths weights, where the path weight is the sum of the length of the path and a function of the leaf weight. In the present article, we choose the binary logarithm for the function. Intuitively a *balanced weighted tree* is such that the heaviest a leaf is, the smallest is its distance to the root.

Trees play a central role in various fields of Computer Science. A lot of text algorithms use trees (see [10], [15] and [20]). Weighted trees appear in the field of data structures (see [23]). A wild range of binary search trees are balanced weighted trees, for example

Email address: `frederique.carrere@labri.fr` (Frédérique Carrère)

[1] Laboratoire Bordelais de Recherche en Informatique (LaBRI) - CNRS: UMR 5800

Preprint submitted to Elsevier *February 14, 2013*

$BB[\alpha]$-trees ([25]), scapegoat trees ([13]), general-balanced trees ([1]). Their definitions use different notions of balance and the weights are based either on the size of subtrees or on the number of leaves of subtrees. These weighted trees are used to implement finite sets and finite maps in functional programming languages (Data.Set and Data.Map in Haskell, wttree.scm library in Scheme). In the case of $BB[\alpha]$-trees, the number of operations (rotations) needed for the insertion of n nodes, starting from the empty tree, is less than $c.n$, with c equal to 19.59 when α is equal to 1/3 (but Mehlhorn noticed in [22] that the experiments suggest a value of c close to one). Our algorithm to build balanced weighted trees is an incremental algorithm. It combines the idea of bottom-up search for the place of insertion, used in scapegoat trees and the technique of double rotation after insertion, used in $BB[\alpha]$-trees. Following our definition of the balance, it does not require that each node of the resulting tree satisfies a balance criterion. It proceeds by successive insertions starting from the empty tree, but it uses at most one double rotation after each insertion of a node. It applies to any sequence of positive weights, whereas the linear amortized complexity achieved for $BB[\alpha]$-trees is due to the fact that all the leaves have a weight equal to 1.

Building a balanced weighted tree with weights $w_1, \ldots, w_n$ can also be seen as a problem of clustering the nodes such that the clusters have near weights. This problem arises both in the field of information theory and in the field of bioinformatics. Both the well-known Huffman algorithm for lossless data compression (see [17], [26]) and the Neighbor-Joining algorithm which is widely used in genomics (see [24], [4]) are based on the principle of joining two nodes which minimize a certain weight. The weight can be a probability in the case of the Huffman coding or a distance in the case of the Neighbor-Joining algorithm. In this paper we are interested in trees such that the order of the leaves is relevant and can not be changed. Such trees are called alphabetic trees. The problem of building balanced alphabetic weighted trees has been investigated both for the optimization of lossless coding (see [11]) and for the optimization of circuits design (see [19]). Several algorithms adapt the principle of Huffman algorithm to the case of alphabetic trees (see [14], [16], [12]). These algorithms use either sorting or generalize selection and complex extra data structures. They run in linear time when the weights are integers or when the number of distinct integer parts of the weights (real weights) is bounded. Our algorithm uses neither sorting nor extra data structures nor extra hypothesis on the weights. It applies to positive weights (either integers or reals).

Our aim is to balance binary algebraic terms. Every binary term can be seen as a binary tree. In particular we are interested in binary terms representing graphs of bounded tree-width or clique-width ([3], [2], [8], [6] and [7]). Courcelle and Vanicat in [9] give fundamental results concerning queries on graphs of bounded tree-width or clique-width, provided that the graphs store small distributed data in their vertices. For these model-checking algorithms it is a crucial point to represent the graphs using binary trees of logarithmic height. In the same article Courcelle and Vanicat give an algorithm to balance a given term t (representing a graph), which runs in time $\mathcal{O}(|t|\log(|t|)$ where $|t|$ is the size of the term (or tree). In this paper we give a linear algorithm. The entry of the algorithm is the right branch of the tree. We associate a weight with each node of this branch. We get thus a sequence of weights. Therefore a linear algorithm to build a balanced weighted tree for a given sequence of weights is a key tool to build balanced decomposition trees for graphs. Dealing with balanced trees also enables to minimize the size of graph's representation. Meer and Rautenbach in [21] study the size of ordered

BDD for representing the adjacency function of a graph G on n vertices. Their algorithms use balanced tree-width or clique-width terms with depth $\mathcal{O}(log(n))$. The encoding of G's nodes used by the BDD has length $\mathcal{O}(h\log(k))$ if the graph has tree-width or clique-width k and if h is the depth of the decomposition tree. Balanced terms can also be used to solve constraints satisfaction problems if the constraint graph has bounded tree-width (see [5], [18]).

In the present paper we propose a dynamic linear algorithm for balancing alphabetic weighted strings and trees, with positive real weights. The paper is organized as follows. In the second section, we define the notion of (c, b)-balanced weighted tree, for positive integers c and b and we give a simple inductive algorithm in $\mathcal{O}(n\log(n))$ which builds a $(2, 2)$-balanced weighted tree. In the third section we give a dynamic linear algorithm which builds a $(2, 2)$-balanced weighted tree for any given sequence of positive weights. In the fourth section we use the linear algorithm to build 4-height-balanced binary trees representing binary terms. This result can apply to algebraic terms representing graphs and can be used for example in the well-known classes of graphs of bounded tree-width or bounded clique-width.

2. Weighted trees

In the present work, we consider only binary trees. A *weighted tree* is a tree which leaves hold weights. We consider only non negative weights.

Let t be a binary tree. The root of t is denoted $\mathrm{root}(t)$. The size (e.g., the number of nodes) of t is denoted $|t|$. The *height* of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. It is denoted $h(t)$. The *depth* of a node is the length of the path from that node to the root, which is called its *root path*. Every internal node x (e.g., not a leaf) has two child nodes denoted $\mathrm{left}(x)$ and $\mathrm{right}(x)$. The *left-weight* of x, denoted $w_{left}(x)$, is the weight of its left child. Every node x different from the root is the son of a node y called its *parent node* and denoted $\mathrm{parent}(x)$. The other son of y is the *sibling of* x. If y is not the root, the parent of y is denoted $\mathrm{gparent}(x)$. We denote $\mathrm{ggparent}(x)$ the parent of the parent of y, if it exists. Every node x of t is the root of a subtree consisting of the node x itself and all its descendants. This subtree is denoted $t{\downarrow}x$. The weight of any subtree of t is the sum of the weights of its leaves.

The following definition generalizes the notion of height-balanced tree. The logarithms are always in base 2.

Definition 2.1. Let t be a tree. Let c and b be integers, $c \geq 1$, $b \geq 0$. The tree t is (c, b)-*height-balanced* if and only if $h(t) \leq c\log(|t|) + b$.

We also define a notion of balance for weighted trees which involves the weights of the leaves. Let c and b be integers, $c \geq 1$, $b \geq 0$. The idea is that a leaf which weight is greater than $w(t)/p$ for some integer $p \geq 2$, must have a depth at most $c\log(p) + b$.

Definition 2.2. Let t be a weighted tree with leaves $u_1, u_2, \ldots, u_n$. Let c and b be integers, $c \geq 1$, $b \geq 0$. The tree t is (c, b)-*balanced* if and only if every leaf u_i of t $(1 \leq i \leq n)$ has a depth at most $c\log(w(t)/w(u_i)) + b$.

We shall say that a weighted tree is c-balanced when it is $(c, 0)$-balanced. We study now some properties of (c, b)-balanced weighted trees. First of all notice that if the weight of each leaf of t is 1, then t is a (c, b)-balanced weighted tree if and only if t is a (c, b)-height-balanced tree.

Recall that the root-to-leaf path weight for a given leaf is the sum of its depth and a function of its weight. Let us choose the function $f : x \rightarrow c \log(x)$. Then the first immediate consequence of the preceding definition is that if t is a (c, b)-balanced weighted tree then the maximum root-to-leaf path weight is bounded by $c \log(w(t)) + b$.

Proposition 2.1. *Let t be a weighted tree with leaves $u_1, u_2 \ldots, u_n$. If t is (c, b)-balanced, for some integers $c \geq 1$, $b \geq 0$, then the maximum root-to-leaf path weight, $max\{\, depth(u_i) + c \log(w(u_i)),\ 1 \leq i \leq n \,\}$, is less than $c \log(w(t)) + b$.*

The following property is interesting if we want to substitute a leaf of weight w_i with a binary tree of size w_i (this is the case in section 4).

Proposition 2.2. *Let t be a weighted tree with leaves $u_1, u_2 \ldots, u_n$. If t is (c, b)-balanced, for some integers $c \geq 1$, $b \geq 0$, then substituting any leaf u_i of t with an arbitrary (c, b')-height-balanced tree t_i of size $w(u_i)$ gives a $(c, b + b')$-height-balanced tree t'.*

Proof. Let t be a (c, b)-balanced weighted tree with leaves $u_1, u_2, \ldots, u_n$ holding the weights $w_1, w_2, \ldots, w_n$. By hypothesis every leaf u_i $(1 \leq i \leq n)$ has a depth at most $c \log(w(t)/w_i) + b$. Let t' be the tree obtained by substituting each u_i, $1 \leq i \leq n$, with a (c, b')-height-balanced tree t_i. By hypothesis the size of t_i is w_i, and the height of t_i is at most $c \log(w_i) + b'$. By definition the height of t is $max\{\, \mathrm{depth}(u_i),\ 1 \leq i \leq n \,\}$. So after the substitution, the height of t' is $max\{\, \mathrm{depth}(u_i) + h(t_i),\ 1 \leq i \leq n \,\}$. Thus $h(t')$ is bounded by $max\{c \log(w(t)/w_i) + b + c \log(w_i) + b',\ 1 \leq i \leq n\}$. Recall that $w(t)$ is the sum of the weights w_i, for $1 \leq i \leq n$. In this case it is the sum of the sizes of the t_i's and it is less than the size of t'. So we have $h(t') \leq c \log(|t'|) + b + b'$ and t' is $(c, b + b')$-height-balanced.

Let us now state a more technical lemma which is the basic tool for the inductive proves in the following sections.

Proposition 2.3. *Let t be a weighted tree. Let c and b be integers, $c \geq 1$, $b \geq 0$. If on every branch of t of length strictly greater than b, there exists a node y and some integers $p \geq 0$, $r \geq -b$ such that*

1. *$depth(y) \leq p \times c - r$*
2. *the subtree $t{\downarrow}y$ is $(c, b + r)$-balanced*
3. *$w(y) \leq w(t)/2^p$*

then t is (c, b)-balanced.

Proof. Let u_i be a leaf of t, $1 \leq i \leq n$. If u_i has a depth less than b then the property $\mathrm{depth}(u_i) \leq c \log(w(y)/w_i) + b$ is trivially true.
Otherwise, by hypothesis there exists an ancestor y of u_i at depth at most $p \times c - r$ such that $t{\downarrow}y$ is $(c, b + r)$-balanced and $w(y) \leq w(t)/2^p$.
If y is a proper ancestor of u_i then, by hypothesis (2) the depth of the leaf u_i in the subtree $t{\downarrow}y$ is less than $c \log(w(y)/w_i) + b + r$. It follows that the depth of u_i in t is

bounded by $\text{depth}(y) + c\log(w(y)/w_i) + b + r$. Hence, by hypothesis (1) and(3), we have $\text{depth}(u_i) \leq p \times c - r + c\log(1/2^p * w(t)/w_i) + b + r$. This gives the result.
If $y = u_i$ then by hypothesis (3), the weight w_i is less than $w(t)/2^p$. Thus we have $log(w(t)/w_i) \geq p$. By hypothesis (1), the depth of u_i is less than $p \times c - r$ with $-r \leq b$ then $\text{depth}(u_i) \leq p \times c + b$, which is less than $c\log(w(t)/w_i) + b$
Consequently t is (c, b)-balanced.

If t satisfies the hypothesis of lemma 2.3 with the values $c = 2$ and $b = 2$ and $p = 1$, then we have the following result:

Corollary 2.4. *Let t be a weighted tree. If on every branch of t of length strictly greater than 2, there exists a node y such that:*

- *$depth(y) \leq 2$*
- *the subtree $t{\downarrow}y$ is $(2, 2)$-balanced*
- *$w(y) \leq w(t)/2$*

then t is $(2, 2)$-balanced.

By Lemma 2.3, given a sequence of weights w_1, w_2, ... w_n, a straightforward inductive algorithm yields a $(2, 2)$-balanced weighted tree with leaves holding the weights $w_1, w_2, \ldots, w_n$.

Corollary 2.5. *For every sequence of weights $w_1, w_2, \ldots, w_n$, one can build in time $O(n\log(n))$ a $(2, 2)$-balanced weighted tree with leaves $w_1, w_2, \ldots, w_n$.*

Proof. Let w be the sum of the n weights. We use a simple inductive algorithm. The algorithm reads the sequence of weights from left to right, and stops on the smallest index i such that the sum of the weights $w_1, \ldots, w_i$ is more than $w(t)/2$. Then we inductively build a weighted tree denoted $first$ which leaves hold the weights $w_1, \ldots, w_{i-1}$ and a weighted tree denoted *last* which leaves hold the weights $w_{i+1}, \ldots, w_n$ (if $i = 1$ then $first$ is *null*; if $i = p$ then *last* is *null*). We also build a small tree denoted *current* with only one node with weight w_i. If none of this three trees is *null*, we build the weighted tree with $first$ as left son of the root and with a new binary node x as right son of the root. We hang *current* as left son of x and *last* as right son of x. If $first$ (resp. *last*) is null we simply build a binary tree with a new root node with sons *current* and *last* (resp. with sons $first$ and *current*).
We can easily prove Corollary 2.5 by induction on n.

Base Case. If the number of weights is less than 3, the depth of each leaf in the weighted tree is at most 2. So the weighted tree is trivially $(2, 2)$-balanced.

Induction Hypothesis. Let $m < n$, for any sequence of m weights the algorithm build a $(2, 2)$-balanced weighted tree.

Induction Step. Let i be the smallest index i such that the sum of the weights $w_1, \ldots, w_i$ is more than $w(t)/2$. Then clearly the sum of the weights w_1, ..., w_{i-1} is less than $w(t)/2$. Similarly the sum of the weights $w_{i+1}, \ldots, w_n$ is less than $w(t)/2$. By Induction Hypothesis the algorithm build two $(2, 2)$-weight-balanced trees $first$ and *last* for the sequences

$(w_1, \ldots, w_{i-1})$ and $(w_{i+1}, \ldots, w_n)$. Consequently, if new_node(t_1, t_2) is a function which creates a new binary node with sons root(t_1) and root(t_2), then by Corollary 2.4 with $c = b = 2$, the tree new_node($first$, new_node($w_i, last$)) is $(2, 2)$-balanced.

The complexity of the inductive algorithm is clearly $O(n \log(n))$, if n is the number of weights.

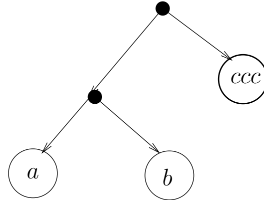

Figure 1: The weighted tree of Example1.

Example 2.1. let s be the word *abccc*. Assume that each occurrence of a letter has weight 1, then the weight of s is 5 Let $u_1 = a$, $u_2 = b$, $u_3 = ccc$ be three factors of s. The weights of the factors are respectively $w_1 = 1$, $w_2 = 1$ and $w_3 = 3$. The smallest index i such that the sum $w_1 + \ldots + w_i$ is less than $w(s)/2$ is 3. Hence we build the weighted tree $first$ with leaves $u_1 = a$, $u_2 = b$ and the weighted tree $last$ is $null$. Then the root of the weighted tree has $first$ as left son and a tree with a unique node label u_3 as right son. We obtain the weighted tree of figure 1.
ccurrence of a letter has weight 1, then the weight of s is 11 Let $u_1 = ab$, $u_2 = a$, $u_3 ccc$, $u_4 = aa$, $u_5 = bab$ be five factors of s. The weights of the factors are respectively $w_1 = 2$, $w_2 = 1$, $w_3 = 3$, $w_4 = 2$, $w_5 = 3$. The smallest index i such that the sum $w_1 + \ldots + w_i$ is less than $w(s)/2$ is 3. Hence we build the weighted tree $first$ with leaves $u_1 = ab$, $u_2 = a$ and the weighted tree $last$ with leaves $u_4 = aa$, $u_5 = bab$. We obtain the weighted tree of figure 2.

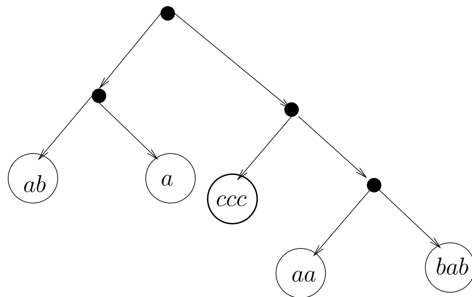

Figure 2: The weighted tree of Example2.

Example 2.2. let s be the word *abacccaabab*. Assume that each occurrence of a letter has weight 1, then the weight of s is 11 Let $u_1 = ab$, $u_2 = a$, $u_3 ccc$, $u_4 = aa$, $u_5 = bab$ be five factors of s. The weights of the factors are respectively $w_1 = 2$, $w_2 = 1$, $w_3 = 3$, $w_4 = 2$, $w_5 = 3$. The smallest index i such that the sum $w_1 + \ldots + w_i$ is less than

$w(s)/2$ is 3. Hence we build the weighted tree $first$ with leaves $u_1 = ab$, $u_2 = a$ and the weighted tree *last* with leaves $u_4 = aa$, $u_5 = bab$. We obtain the weighted tree of figure 2.

3. A linear algorithm to balance weighted trees.

In this paragraph we use the notion of *weight-balance* of a node in a tree which is used for scapegoat trees (see [13]).

Let t be a weighted tree. Let x be a node of t. The *weight-balance* of x is the ratio $w(\text{right}(x))/w(\text{left}(x))$.
Starting from a one node tree and applying a bottom-up schema of insertion from the rightmost leaf, we can build a $(2, 2)$-balanced weighted tree for any sequence of weights.

Theorem 1. *For every sequence of weights $w_1, w_2, \ldots, w_n$, one can build in time $\mathcal{O}(n)$ a $(2, 2)$-balanced weighted tree with leaves $w_1, w_2, \ldots, w_n$.*

Proof of Theorem 1. The balanced weighted tree is build by insertion starting from a one node tree as follows:

- create a leaf tree node t_1 holding the weight w_1.
- Assume that we have built a balanced tree t_{i-1} for the weights $w_1 \ldots w_{i-1}$, climb up from the rightmost leaf to the root, searching a point of insertion for a new node u_i holding the weight w_i.
- The point of insertion is the first node satisfying some conditions on its weight and the weights of its parent and great-parent. This node is called the *weight-sibling* of u_i.
- Assume that v is the weight-sibling of u_i and s is the parent of v in t_{i-1}. To insert u as sibling of v and keep the tree binary, we need to insert also another node c which becomes the new parent of both v and u_i in t_i. We insert c as right son of $s$and v becomes the left son of c (see figure 3). Notice that the node s, which was the parent of v in t_{i-1}, becomes the great parent of v in t_i.
  If we do not modify the subtree $t{\downarrow}c$ after the insertion, we build a $(3, 2)$-balanced weighted tree. If we perform in some cases either a double rotation at c (see figure 5) or a left rotation at v (see figure 4) after the insertion, we can build a $(2, 2)$-balanced weighted tree.

Let t be a weighted tree. Let u be a weighted node not in t.

Definition 3.1. The node v is a *weight-sibling* of the node u (u not in t) if and only if v belongs to the right branch of t, $\text{depth}_t(v) \geq 1$ and inserting u as sibling of v in t, gives a new tree t' such that:

- if $\text{depth}_{t'}(v) = 2$ then $\text{gparent}_{t'}(v)$ has weight-balance less than 1.
- if $\text{depth}_{t'}(v) \geq 3$ then $\text{gparent}_{t'}(v)$ and $\text{ggparent}_{t'}(v)$ have weight-balance less than 1.

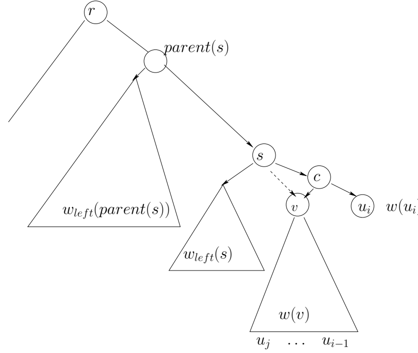

Figure 3: The insertion of a new right leaf u_i in t_{i-1}.

The linear algorithm to build the balanced weighted tree t can be decomposed in four functions:

- the function BUILDBALANCEDTREE$(u_1, u_2, \ldots, u_n)$ builds the tree t by a sequence of insertions: each u_i is inserted in t_{i-1} and $t_n = t$ (see algorithm 1).
- The function FINDWEIGHTSIBLING(t_{i-1}, p, u_i) looks for the point of insertion of u_i on the right branch of t_{i-1}, using a pointer p which moves bottom up from the right leaf of t_{i-1} to its root (see algorithm 2).
- The function ISWEIGHTSIBLING(p, u_i) performs the tests on the weights: v being the node pointed by p, the function tests if the weight-balance of the nodes $\text{gparent}_{t_i}(v)$ and $\text{ggparent}_{t_i}(v)$ (respectively s and s' on figure 3) will be less than one after the insertion of u_i (see algorithm 3).
- The function INSERT(t, u_i, p) performs the insertion of the leaf u_i in t as sibling of the node v pointed by p. This means that the function inserts both u_i and a new node c, which gets v and u_i as sons. If neither v nor its right son v_r are leaves and if the weight-balance of v is strictly greater than 1, then the function also performs either a simple or a double rotation (see algorithm 4). It performs a double rotation of the subtree rooted at c (see figure 5), if this double rotation gives a new subtree which root has weight-balance less than one, that is if the sum of the weights $w(\text{right}(v_r)) + w(u_i)$ is less than the sum of the weights $w(\text{left}(v_r)) + w(\text{left}(v))$ (see figure 5). Otherwise the function performs a left rotation at v (see figure 4) such that c gets a new left son, v_r, which has a weight-balance less than one.

Claim 2. *The complexity of* BUILDBALANCEDTREE$(u_1 \ldots u_n)$ *is* $\mathcal{O}(n)$.

To compute the amortized complexity, we charge a fix amount of credits for each elementary operation. If an operation is cheap and if we have charge more than necessary, we save up credits for later use. If an operation is expensive, which appends only occasionally, we use the credits saved to pay for it.
Let i, $1 \leq i \leq n-1$. Assume that:

- we have built a balanced tree t_{i-1} for the weights $w_1 \ldots w_{i-1}$,
- we have charged $5i - 5$ credits to construct t_{i-1},

Algorithm 1 BUILDBALANCEDTREE($u_1 \dots u_p$)

Require: A list of weighted nodes $u_1, u_2, \dots u_n$.
Ensure: A $(3,2)-$balanced weighted tree t.

1: $t = u_1$
2: $p = u_1$
3: **for** $i = 2$ to n **do**
4:     /* move the pointer p bottom up from u_{i-1} */
5:     $findWeightSibling(t, p, u_i)$
6:     /* after the insertion, p moves on u_i */
7:     $t = insert(t, u_i, p)$
8: **end for**
9: **return** t

Algorithm 2 FINDWEIGHTSIBLING(t_{i-1}, p, u_i)

Require: A weighted tree t_{i-1}, a pointer p on its right leaf u_{i-1}, a weighted node u_i.
Ensure: A pointer on an ancestor of u_{i-1}, which is a $weight - sibling$ of u_i.

1: $weight = w(u_{i-1})$
2: $height = 0$
3: **while** parent(p) $<>$ $null$ **do**
4:     $height = height + 1$
5:     $parentWeight = weight + w(\text{left}(\text{parent}(p)))$
6:     **if** $isWeithSibling(p, u_i)$ **then**
7:         **return** p
8:     **end if**
9:     $p = \text{parent}(p)$
10:     $weight = parentWeight$
11: **end while**

Algorithm 3 ISWEIGHTSIBLING(p, u_i)

Require: A pointer p on the right branch of t_{i-1}, a weighted node u_i.
Ensure: True iff p points on a $weight - sibling$ of u_i.

1: $q = \text{parent}(p)$
2: $qBalance = (w(p) + w(u_i)) \ / \ w(\text{left}(q))$
3: **if** parent(q) $==$ $null$ **then**
4:     **return** $qBalance \leq 1$
5: **else**
6:     $r = \text{parent}(q)$
7:     $rBalance = (w(\text{left}(q)) + w(p) + w(u_i)) \ / \ w(\text{left}(r))$
8:     **return** ($qBalance \leq 1$ **and** $rBalance \leq 1$)
9: **end if**

- at this step we have at least k credits saved on an account, where k is the depth of u_{i-1},

Algorithm 4 INSERT(t, p, u_i)

Require: A leaf u_i to insert at the position pointed by p .
Ensure: A new tree t_i with the pointer p on its rightmost leaf u_i.

```
1:  q = parent(p)
2:  c = new_node(p, u_i)
3:  pBalance = w(right(p)) / w(left(p))
4:  if !leaf(p) and !leaf( right(p) ) and pBalance > 1 then
5:      v_r = right(p)
6:      A = w(left(p))
7:      B = w(left(v_r))
8:      C = w(right(v_r))
9:      if C + w(u_i) ≤ A + B then
10:         c = double_rotation(c)
11:     else
12:         c → left = left_rotation(p)
13:     end if
14: end if
15: p = u_i
16: if q <> null then
17:     q → right = c
18: else
19:     t = c
20: end if
21: return  t
```

We want to charge 5 credits to construct t_i from t_{i-1}.
We work on the rightmost branch of t_{i-1}, with a pointer p on its leaf u_{i-1}. Let u_i be the new leaf node holding the weight w_i. To add u_i in t_{i-1} we move p towards the root until we find a weight-sibling of u_i. There are three cases:

- If u_{i-1} is a weight-sibling of u_i, we insert u_i as sibling of u_{i-1} and move the pointer p to u_i. The cost of the test is 2 credits (we look at the weight of gparent(u_{i-1})), the cost of the insertion is 1 credit and the excess 2 credits are saved into an account.
- If the weight-sibling of u_i is a proper ancestor x of u_{i-1} with depth$(x) = k'$, $k' < k$, we move from u_{i-1} to x and insert u_i as sibling of x. The cost of the move is $k - k'$ credits, the cost of the test is 2 credits (we look at the weight of gparent(x)), the cost of the insertion is 1 credit, so the total cost is $k - k' + 3$ credits.
- Suppose we move from u_{i-1} to the root and insert u_i as sibling of the root. The cost of the move and the insertion is $k + 1$ credits.

In each case we have at least $k' + 2$ credits saved on an account, where $k' + 2$ is the depth of u_i on the rightmost branch of t_i (recall that we insert both u_i and a node c, parent of u_i).

Claim 3. *We claim that the tree* $t = linBalance(u_1 \dots u_p)$ *is* $(2, 2)$*-weight-balanced.*

Figure 4: Left rotation at v, after inserting u_i in t_{i-1}.

Figure 5: Double rotation at c, after inserting u_i in t_{i-1}.

We prove that for every node x of t the subtree $t{\downarrow}x$ is $(2, 2)$-weight-balanced. The proof is done by induction on the size of $t{\downarrow}x$.

Base Case. If $|t{\downarrow}x| \leq 5$ then $h(t{\downarrow}x) \leq 2$ the condition of $(2, 2)$-balance is trivially satisfied.

Let m be an integer such that $m \leq h(t)$.

Induction Hypothesis. Every subtree of t of size strictly less than m is $(2, 2)$-balanced.

Induction Step. Let x be a node of t such that $|t{\downarrow}x| = m$.
Recall that the weight of $t{\downarrow}x$ is $w(x)$. By Corollary 2.4 with $c = 2$ and $b = 2$, it is sufficient to prove that on every branch of $t{\downarrow}x$ of length at least 3, there exists a node y such that $\text{depth}_{t{\downarrow}x}(y) \leq 2$ and $w(y) \leq w(x)/2$. By inductive hypothesis the subtree $t{\downarrow}y$ is $(2, 2)$-balanced. Then the tree $t{\downarrow}x$ is $(2, 2)$-balanced.
We need some preliminary lemmas to prove the existence of such a node y on every branch of $t{\downarrow}x$ of length at least 3. We first focus on the right branch of $t{\downarrow}x$. Let x_1 be the right son of x. The following lemma and corollary prove that both sons of x_1 have a weight less than half the weight of x (see figure 6).

Lemma 4. *Let x be a node of t such that the length of the right branch of $t{\downarrow}x$ is at least 2. Let $x_1 = right(x)$ and $x_2 = right(x_1)$, then we have:*

1. $w_{left}(x_1) \leq w_{left}(x)$,

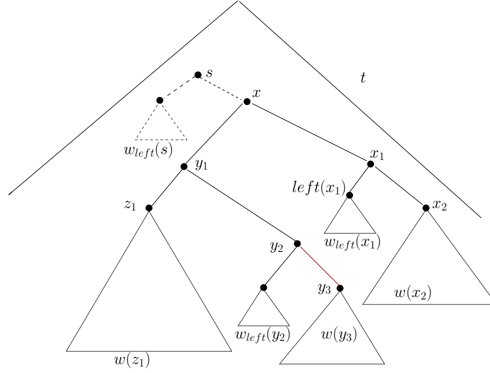

Figure 6: A subtree $t{\downarrow}x$ of t.

2. $w_{left}(x_2) \leq w_{left}(x)/2$,
3. $w(x_2) \leq w_{left}(x) + w_{left}(x_1)$.

Before proving lemma 4, we can state the following corollary:

Corollary 5. *Let x be a node of t such that the length of the right branch of $t{\downarrow}x$ is at least 2. Let $x_1 = right(x)$ and $x_2 = right(x_1)$. We have* $w_{left}(x_1),\ w(x_2) \leq w(x)/2$.

Let A and B be the left and right members of inequalities (1) (resp. (3)) of Lemma 4. Note that A and B are positive numbers such that $A + B \leq w(x)$ and $A \leq B$ then one has $A \leq w(x)/2$. Hence the corollary holds.

Proof of Lemma 4. 1) Let x, x_1, x_2 be as stated in the lemma. Let u_ℓ be the leftmost leaf of $t{\downarrow}x_2$ (if x_2 is a leaf then u_ℓ is x_2). There are three cases.

Case 1. Assume that the leaf u_ℓ is inserted in $t_{\ell-1}$ without rotation. Then the node left(x_1) is a weight-sibling of u_ℓ and by definition 3.1 the weight-balance of x, which is the great parent of left(x_1), is less than 1 in t_ℓ. So we have $w(u_\ell) + w_{left}(x_1) \leq w_{left}(x)$.

Case 2. Assume that after the insertion of u_ℓ we performed a left rotation in the left subtree of x_1 (see figure 4 with $c = x_1$). Then we know that before the rotation the weight-balance of x was less than 1 and the rotation in the left subtree of x_1 does not change the weight-balance of x.

Case 3. Assume that after the insertion of u_ℓ, we performed a double rotation which result is $t{\downarrow}x$ (see figure 5). By hypothesis, a double rotation occurs only if it gives a subtree which root x has weigh-balance less than 1. This implies that $w_{left}(x_1) + w(u_\ell) \leq w_{left}(x)$.

Thus in the three cases, the first statement of the lemma holds.

Proof of Lemma 4. 2) Let u_j be the leftmost leaf of $t{\downarrow}$right(x_2). There are three cases.

Case 1. Assume that the leaf u_j is inserted in t_{j-1} without rotation. Then the node left(x_2) is a weight-sibling of u_j and by definition 3.1 the nodes x_1 and x, which are

respectively the great parent and the great great parent of left(x_2), have a weight-balance less than 1 in t_j. This gives the two following inequalities:
$w(u_j) + w_{left}(x_2) \leq w_{left}(x_1)$ and $w(u_j) + w_{left}(x_2) + w_{left}(x_1) \leq w_{left}(x)$.
Adding this two inequalities we also get that $w(u_j) + w_{left}(x_2) \leq w_{left}(x)/2$.

Case 2. Assume that after the insertion of u_j we performed a left rotation in the left subtree of x_2 (see figure 4 with $c = x_2$). Then we know that before the rotation the weight-balance of x_1 and x was less than 1 and the rotation in the left subtree of x_2 does not change the weight-balance of x_1 and x.

Case 3. Assume that after the insertion of u_j we performed a double rotation which result is $t{\downarrow}x_1$ (see figure 5). By hypothesis, a double rotation occurs only if it gives a subtree which root x_1 has weigh-balance less than 1. This implies that $w_{left}(x_2)+w(u_j) \leq w_{left}(x_1)$. On the other hand, the weight-balance of x is not changed by the double rotation and we know that it is less than 1. So we also have $w(u_j)+w_{left}(x_2)+w_{left}(x_1) \leq w_{left}(x)$. The two inequalities give the result, as in case (1).

Thus in the three cases, the second statement of the lemma holds.

Proof of Lemma 4. 3) If x_2 is a leaf, we proved (see Proof of lemma 4 (1)) that $w(x_2) + w_{left}(x_1) \leq w_{left}(x)$. Thus the third part of the lemma holds.
Otherwise let $x, x_1, x_2, \ldots, x_m$ and u be the nodes on the right branch of $t{\downarrow}x$, where u is the leaf. Note that the weight $w(x_2)$ is the sum of the left weights on the right branch of $t{\downarrow}x_2$, that is $w_{left}(x_2) + \ldots + w_{left}(x_m) + w(u)$. From Lemma 4 (2) it follows that for every i, $2 \leq i \leq m-1$, we have $w_{left}(x_i) \leq w_{left}(x_{i-2})/2$. Using this inequality we can get the following result:

- if i is even, $i = 2p$, then for every k, $1 \leq k \leq p$, we have:
  $w_{left}(x_i) \leq \frac{1}{2} w_{left}(x_{i-2}) \ldots \leq \frac{1}{2^k} w_{left}(x_{i-2k}) \ldots \leq \frac{1}{2^p} w_{left}(x_0)$,
- if i is odd, $i = 2p+1$, then for every k, $1 \leq k \leq p$, we have:
  $w_{left}(x_i) \leq \frac{1}{2} w_{left}(x_{i-2}) \ldots \leq \frac{1}{2^k} w_{left}(x_{i-2k}) \ldots \leq \frac{1}{2^p} w_{left}(x_1)$.

Since gparent(x_m) and ggparent(x_m) have weight-balance less than 1, we also get that $w_{left}(x_m)+w(u) \leq w_{left}(x_{m-2})/2$. From this inequality we deduce the following result:

- if m is even, $m = 2p$, then we have $w_{left}(x_m) + w(u) \leq \frac{1}{2^p} w_{left}(x_0)$,
- if m is odd, $m = 2p+1$, then we have $w_{left}(x_m) + w(u) \leq \frac{1}{2^p} w_{left}(x_1)$.

As the weight of x_2 is the sum of the left-weights on the right branch of $t \downarrow x_2$, we obtain the following result:

$$\begin{aligned} w(x_2) = \sum_{i=2}^{i=m} w_{left}(x_i) + w(u) &= \sum_{i=2}^{i=m-1} w_{left}(x_i) + w_{left}(x_m) + w(u) \\ &\leq (w_{left}(x_0) + w_{left}(x_1)) \ \left(\frac{1}{2} + \frac{1}{4} \ldots + \frac{1}{2^p}\right) \ \leq \ w_{left}(x_0) + w_{left}(x_1). \end{aligned}$$

Thus the lemma 4 holds.

It remains to consider the left branch of $t{\downarrow}x$. Let y_1 be the left son of x, the following lemma proves that both sons of y_1 have a weight less than half the weight of x (see figure 6).

Lemma 6. *Let x be a node of t such that the length of the left branch of $t{\downarrow}x$ is at least 2. Let $y_1 = left(x)$, we have $w_{left}(y_1) \leq w(x)/2$. Let $y_2 = right(y_1)$ and assume that y_2 is not a leaf, then we have $w(y_2) \leq w(x)/2$.*

Proof of Lemma 6. Let $x_1 = \text{right}(x)$. Let u_i be the leftmost leaf of $t{\downarrow}x_1$. There are three cases.

Case 1. Assume that the leaf u_i is inserted in t_{i-1} without rotation. As u_i is not inserted in $t{\downarrow}y_1$, we can deduce that none of the nodes on the right branch of $t{\downarrow}y_1$ is a weight-sibling of u_i. In particular y_2 is not a weight-sibling of u_i. To apply the definition of weight-sibling we must know the depth of y_2 in t_i.
Suppose that the node x is the right son of a node s in t. Then the depth of y_2 in t_i is at least 3. By the algorithm of insertion, the node $\text{left}(x)$ is a weight-sibling of u_i. This implies that the node s, which is the great parent of $\text{left}(x)$, has a weight-balance less than 1. The node s is also the great great parent of y_2. On the other hand, recall that y_2 is not a weight-sibling of u_i. Then necessarily (see Definition 3.1), the great parent of y_2, namely y_1, has a weight-balance strictly greater than 1. This gives $w_{left}(y_1) < w(u_i)+w(y_2)$. Since we also trivially have $w_{left}(y_1)+w(y_2)+w(u_i) \leq w(x)$, adding these two inequalities we get that $w_{left}(y_1) \leq w(x)/2$. So the first inequality of the lemma holds.
Suppose that x is the left son of a node s in t. This means that x belongs to the left branch of t. Necessarily y_1 was the root of t_{i-1} and the tree t_i has been constructed by adding a new root x with left son y_1 and right son u_i. Then the depth of y_2 in t_i is 2. Recall that y_2 is not a weight-sibling of u_i. Then by Definition 3.1 the weight-balance of the great parent of y_2, which is y_1, is greater than 1. This gives $w_{left}(y_1) < w(u_i) + w(y_2)$. Since we also trivially have $w_{left}(y_1) + w(y_2) + w(u_i) \leq w(x)$, adding these two inequalities we get that $w_{left}(y_1) \leq w(x)/2$. So the first inequality of the lemma holds.
On the other hand, the fact that we perform no rotation after the insertion, assuming that y_2 is not a leaf, implies that the weight-balance of y_1 is less than 1. This means that $w(y_2) \leq w_{left}(y_1)$. Since we trivially have that $w(y_2) + w_{left}(y_1) \leq w(x)$, it follows that $w(y_2) \leq w(x)/2$. So the second inequality of the lemma holds.

Case 2. Assume that after the insertion of u_i we performed a left rotation which result is the subtree $t{\downarrow}y_1$ (see figure 4). This implies that z_1 is not a leaf. Let $A = w(\text{left}(z_1))$, $B = w(\text{right}(z_1))$ and $C = w(y_2)$. The fact that we do not perform a double rotation implies that $C+w(u_i) > A+B$ (see algorithm 4). Since we trivially have $A+B+C+w(u_i) \leq w(x)$, it follows that $A + B \leq w(x)/2$, that is $w(\text{left}(y_1)) \leq w(x)/2$. On the other hand, by Lemma 4 (3) applied to the subtree $t{\downarrow}\text{left}(x))$ before the rotation, we know that $C \leq A + B$. Since we trivially have $A + B + C \leq w(x)$, it follows that $C \leq w(x)/2$, that is $w(y_2) \leq w(x)/2$.

Case 3. Assume that after the insertion of u_i we performed a double rotation which result is the subtree $t{\downarrow}x$ (see figure 5). Let u_ℓ be the leftmost leaf of $t{\downarrow}x_2$. Let $A = w(z_1)$, $B = w(y_2)$ and $C = w(\text{left}(x_1))$. By Lemma 4 (1) applied to the subtree $t{\downarrow}\text{left}(x))$ before the rotation, we know that $B \leq A$. Since we trivially have $A + B \leq w(x)$, it follows that $B \leq w(x)/2$, that is $w(y_2) \leq w(x)/2$. On the other hand, the proof of case (1) applied to $t{\downarrow}x$ before the double rotation gives that $A \leq w(x)/2$, that is $w_{left}(y_1) \leq w(x)/2$.

Thus in the three cases the lemma holds.

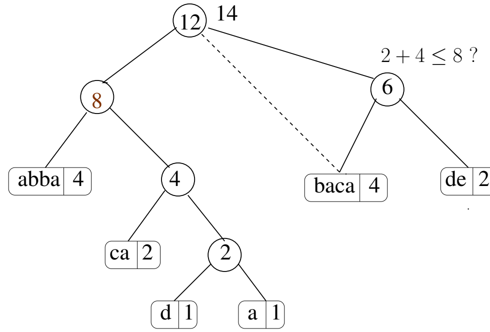

Figure 7: Insertion of a new factor u_6.

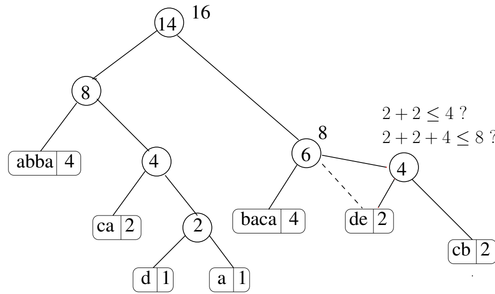

Figure 8: Insertion of a new factor u_7.

In conclusion, on every branch of $t{\downarrow}x$ of length at least 3, there exists a node, namely left(x_1) or x_2 or left(y_1) or y_2, which depth is less than 2, which weight is less than $w(x)/2$, and which is the root of a $(2,2)$-balanced subtree (by inductive hypothesis). So by Corollary 2.4 (with $c = 2$ and $b = 2$) the tree $t{\downarrow}x$ is $(2,2)$-balanced.

This is true for any node x of t, including the root. Consequently the tree t is $(2,2)$-balanced.

Example 3.1. let s be the word *abbacadabacdecbadabeded*. Assume that each occurrence of a letter has weight 1, then the weight of s is 23. Let $u_1 = abba$, $u_2 = ca$, $u_3 = d$, $u_4 = a$, $u_5 = bac$, $u_6 = de$, $u_7 = cb$, $u_8 = ad$, $u_9 = a$, $u_{10} = b$, $u_{11} = ed$, $u_{12} = ed$ be twelve factors of s. The weights of the factors are respectively $w_1 = 4$, $w_2 = 2$, $w_3 = 1$, $w_4 = 1$, $w_5 = 3$, $w_6 = 2$, $w_7 = 2$, $w_8 = 2$, $w_9 = 1$, $w_{10} = 1$, $w_{12} = 2$, $w_{12} = 2$. Assuming that we have build a balanced tree for u_1, u_2, u_3, u_4, u_5, we show in figures 7 and 8 the insertion of u_6 and u_7, then in figures 9, 10 and 11 we show the insertion of u_{10}, u_{11}, u_{12}.

4. Application to terms.

Let F be a set of binary operation symbols. Let C be a set of constants symbols. The set $T(F,C)$ is the set of *terms* over the signature (F,C). There is a classical bijection between the terms of $T(F,C)$ and the binary trees with labels of internal nodes in F and labels of leaves in C and we shall identify terms and trees.

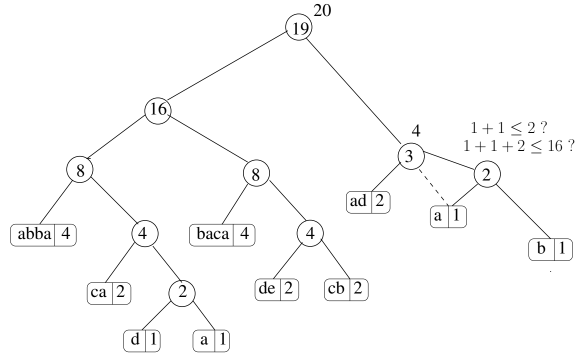

Figure 9: Insertion of a new factor u_{10}.

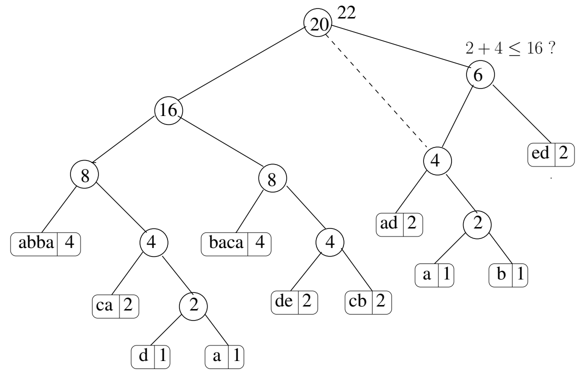

Figure 10: Insertion of a new factor u_{11}.

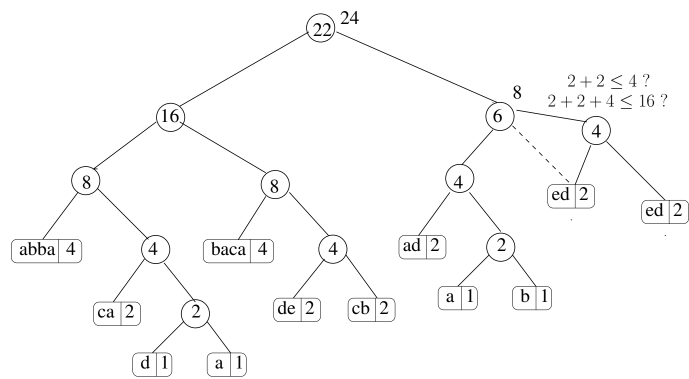

Figure 11: Insertion of a new factor u_{12}.

Let t be a tree in $T(F,C)$. Let x be a node of t. The *context* of x in t is the tree obtained by replacing x in t with a node labeled by a special symbol u not in C, which has no successor (all the descendants of x are deleted). If x is the root of t, the context of x is the trivial context reduced to one node u. The *contexts* are trees in $T(F, C \cup \{u\})$, containing a unique occurrence of the constant u. The notions of depth, height, size, defined for trees are defined similarly for contexts. The set $Ctxt(F,C)$ denotes the set of all the contexts.

Let t be a tree in $T(F,C)$, we define $c \circ t = c\,[u := t]$ as the tree of $T(F,C)$ obtained by replacing u with the tree t in the context c. Let c' be a context in $Ctxt(F,C)$, we define $c \circ c' = c[u := c']$ as the context obtained by replacing u with the context c' in c. Note that for any tree t and any contexts c and c', if f belongs to F then $f(c,t)$ and $f(t,c)$ are contexts (they contain a unique occurrence of u), but $f(c,c')$ is not a context (it contains two occurrences of u). Using these new function symbol $\circ$ and this new constant symbol u, we can build a new class of terms.

The set of *well-formed trees* $\widehat{T}(F,C)$ is the subset of $T(F \cup \{\circ\}, C \cup \{u\})$ generated by the following grammar with axiom S:
$S \longrightarrow T \mid X$
$T \longrightarrow f(T,T) \mid \circ(X,T) \mid a \quad \forall f \in F, \forall a \in C$
$X \longrightarrow f(T,X) \mid \circ(X,X) \mid u \quad \forall f \in F.$
The set of extended contexts $\widehat{Ctxt}(F,C)$ is the subset of $T(F \cup \{\circ\}, C \cup \{u\})$ generated by the preceding rules with axiom X. Notice that the symbol u may occur more than one time in an extended context.

We define an equivalence relation on well-formed trees and extended contexts. Two well-formed trees (resp. two extended contexts) are equivalent if and only if carrying out (in any order) all the substitutions denoted by $\circ$ in these two well-formed trees (resp. extended contexts) gives the same tree of $T(F,C)$ (resp. the same context of $Ctxt(F,C)$).

Theorem 7. *For every tree in $T(F,C)$ of size n, one can build in time $\mathcal{O}(n)$ an equivalent tree in $\widehat{T}(F,C)$ which is 4-height-balanced.*

We can assume without loss of generality that every tree t in $T(F,C)$ is such that for any node x of t, the size of $t\!\downarrow\!\mathrm{left}(x)$ is less than the size of $t\!\downarrow\!\mathrm{right}(x)$. If the symbols of F represent commutative functions this can always be achieve. Otherwise we apply the algorithm as if the functions were commutative and in the resulting tree we proceed to the permutations of the left and right sons when necessary.

Proof of Theorem 7. Actually we prove a stronger result. For any tree $\widehat{t}$ in $\widehat{T}(F,C)$, let us use a different definition of the size. We define $s(\widehat{t}\,)$ as the number of nodes of $\widehat{t}$ which belong to $F \cup C$ (i.e. which are different from the two symbols $\circ$ and u). Then we clearly have that if a well-formed tree $\widehat{t}$ is equivalent to a tree t in $T(F,C)$, then it has a size $s(\widehat{t}\,)$ which is equal to $|t|$. We define similarly the size $s(c)$ of an extended context c in $\widehat{Ctxt}(F,C)$. We prove that for every tree t in $T(F,C)$ of size n, one can build in time $\mathcal{O}(n)$ an equivalent tree $\widehat{t}$ in $\widehat{T}(F,C)$ such that the height of $\widehat{t}$ is less than $4\log(s(\widehat{t}\,))$.

To achieve a complexity in $\mathcal{O}(n)$ we cannot go down and up along the rightmost branch of the tree as often as we would do with an inductive algorithm. We can use at most a fix number of runs along the righmost branch. Therefore we balance the

tree using two runs along the rightmost branch. If a_1, a_2, …, a_k are the nodes of the rightmost branch of t, we first balance the left subtree hanging on a node a_i, for any i, $1 \leq i \leq k-2$. The subtree hanging on a_{k-1} is a small subtree t_{k-1} of size 3, which is trivially 4-height-balanced. Thus we obtain a sequence of 4-height-balanced subtrees $\widehat{T}_1$, $\widehat{T}_2$, …, $\widehat{T}_{k-2}$, t_{k-1}, which respective sizes are $s(\widehat{T}_1)$, $s(\widehat{T}_2)$, …, $s(\widehat{T}_{k-2})$, $|t_{k-1}|$. Then we use Algorithm of Section 3 to build in linear time a balanced weighted tree t_0 with leaves holding the weights $s(\widehat{T}_1)+1$, $s(\widehat{T}_2)+1$, …, $s(\widehat{T}_{k-2})+1$, $|t_{k-1}|+1$ (we add 1 to $|t_{k-1}|$ for some technical reason) . The final step of the algorithm consists in labeling internal nodes of t_0 with the substitution operator $\circ$ and hanging on the leaves of t_0 the subtrees $a_1(\widehat{T}_1, u)$, …, $a_{k-2}(\widehat{T}_{k-2}, u)$ and t_{k-1}. At the end we get a 4-height-balanced binary tree, that is a tree $\widehat{t}$ which height is less than 4 times the logarithm of its size $s(\widehat{t})$.
The proof of the theorem is done by induction on the size of t.

Base Case. If $|t| \leq 5$, then the height of t is 2 and t is clearly 4-height-balanced, then $\widehat{t}$ is equal to t (and $s(\widehat{t})$ is equal to $|t|$).

Induction Hypothesis. For every tree t' of size m strictly less than $|t|$, one can build in time $\mathcal{O}(m)$ an equivalent tree $\widehat{t'}$ in $\widehat{T}(F,C)$ such that the height of $\widehat{t'}$ is less than $4\log(s(\widehat{t'}))$.

Induction Step. Let $a_1, a_2, \ldots a_k$ be the nodes on the rightmost branch of t, from the root to the leaf. If k is less than 3, since we assume that for any node x of t, the size of $t{\downarrow}\mathrm{left}(x)$ is less than the size of $t{\downarrow}\mathrm{right}(x)$, we necessarily have a tree t of size at most 7 and the result is trivial. Let k be at least 4. Assume that for all i, $1 \leq i \leq k-2$, we cut the right branch of t between each pair of consecutive nodes, a_i and a_{i+1}. Then we get the contexts $a_1(t{\downarrow}\mathrm{left}(a_1), u)$, $a_2(t{\downarrow}\mathrm{left}(a_2), u)$,…, $a_{k-2}(t{\downarrow}\mathrm{left}(a_{k-2}), u)$. The subtrees $t{\downarrow}\mathrm{left}(a_1)$, $t{\downarrow}\mathrm{left}(a_2)$, …, $t{\downarrow}\mathrm{left}(a_{k-2})$ have a size strictly less than $|t|$. By induction hypothesis one can build equivalent well-formed trees $\widehat{T}_1$, $\widehat{T}_2$, …, $\widehat{T}_{k-2}$ in $\widehat{T}(F,C)$ such that for any i, $1 \leq i \leq k-2$, the height of $\widehat{T}_i$ is less than $4\log(s(\widehat{T}_i))$. Thus for any i, $1 \leq i \leq k-2$, the context $a_i(t{\downarrow}\mathrm{left}(a_i), u)$, is equivalent to the extended contexts $a_i(\widehat{T}_i, u)$, where $\widehat{T}_i$ is a 4-height-balanced binary tree. Each $\widehat{T}_i$, $1 \leq i \leq k-2$, can be constructed in time $\mathcal{O}(|t{\downarrow}\mathrm{left}(a_i)|)$. So we can build the sequence $\widehat{T}_1$, …, $\widehat{T}_{k-2}$, in time $\mathcal{O}(n)$. The last subtree t_{k-1} is a small 4-height-balanced tree with 3 nodes.
Let us consider now the nodes $a_1, a_2, \ldots a_{k-2}$ and a_{k-1} as weighted nodes with respective weights $w_1 = s(\widehat{T}_1)+1$, $w_2 = s(\widehat{T}_2)+1$, …, $w_{k-2} = s(\widehat{T}_{k-2})+1$ and $w_{k-1} = |t{\downarrow}a_{k-1}|+1$. We use the linear algorithm of section 3 applied to the nodes a_1, a_2,…, a_{k-1}, holding the weights w_1,w_2,…,w_{k-1} to build a weighted tree. The sequence of weights w_1, w_2 … w_{k-1}, satisfies the following assertion.

Claim 8. *For any i, $1 \leq i \leq k-2$, w_i is less than $\sum_{j=i+1}^{j=k-1} w_j$.*

Recall that we assume that for any node x of t, the size of $t{\downarrow}\mathrm{left}(x)$ is less than the size of $t{\downarrow}\mathrm{right}(x)$. Thus for any i, $1 \leq i \leq k-2$, the size $|t{\downarrow}\mathrm{left}(a_i)|$ is less than $\sum_{j=i+1}^{j=k-2}(\,|t{\downarrow}\mathrm{left}(a_j)|\,+1\,) + |t{\downarrow}a_{k-1}|$. Then it follows that for any i, $1 \leq i \leq k-2$, the weight of a_i, $s(\widehat{T}_i)+1$, which is also equal to $|t{\downarrow}\mathrm{left}(a_i)|+1$ (by the definition of s), is less than $\sum_{j=i+1}^{j=k-2} w_j + w_{k-1}$.

□

Let t_0 be the tree obtained by the linear algorithm of section 3 applied to the nodes a_1, a_2,..., a_{k-1}, holding the weights w_1, w_2, ..., w_{k-1}. The tree t_0 is $(2,2)$-balanced. If we hang on each leaf a_i a subtree which is itself $(2,2)$-balanced, then by Lemma 2.2 we get a $(2,4)$-balanced tree and the induction failed. Therefore we first need to prove that t_0 is a $(4,0)$-balanced weighted tree.

Claim 9. *The tree t_0 is a 4-balanced weighted tree.*

By definition of a $(2,2)$-balanced weighted tree, any leaf a_i, $1 \leq i \leq k-1$, of t_0 has a depth smaller than $2\log(w(t_0)/w(a_i)) + 2$. From Claim 8 we can deduce that for any i, $1 \leq i \leq k-2$, $w(a_i)$ is smaller than $w(t_0)/2$. This implies that $log(w(t_0)/w(a_i))$ is greater than 1. Thus for any i, $1 \leq i \leq k-2$, $2\log(w(t_0)/w(a_i)) + 2$ is smaller than $4\log(w(t_0)/w(a_i))$. For the last leaf a_{k-1} with weight 4, either the tree t is small or $w(a_{k-1})$ is less than $w(t_0)/2$. In both case the depth of a_{k-1} is less than $4\log(w(t_0)/w(a_{k-1}))$. So t_0 is 4-balanced.

□

We build the 4-height-balanced tree $\widehat{t}$ equivalent to t by replacing for any i, $1 \leq i \leq k-2$, the leaf a_i of t_0 with the subtree $a_i(\widehat{T}_i, u)$ and the node a_{k-1} with t_{k-1} (a 4-height-balanced subtree of size 3). We denote the substitution as follows: $\widehat{t} = t_0[\, a_1 := a_1(\widehat{T}_1, u), \ldots, a_{k-2} := a_{k-2}(\widehat{T}_{k-2}, u)\,,\, a_{k-1} := t_{k-1}]$.
By hypothesis, for any i, $1 \leq i \leq k-2$, the tree $\widehat{T}_i$ is a 4-height-balanced tree and the tree t_{k-1} is a 4-height-balanced tree of size 3. By Claim 9 the tree t_0 is a 4-balanced weighted tree. Then by Lemma 2.2 we obtain a tree $\widehat{t}$ which is 4-height-balanced and this proves the theorem.

We conjecture that using the fact that we deal here with trees representing terms (internal nodes are labeled with the substitution operator), we can improve the algorithm of Section 3 to build a $(3,0)$-balanced weighted tree t'_0. This would lead to a 3-height balanced binary tree $\widehat{t}$.

5. Conclusion

In this paper we give a linear algorithm to build a $(2,2)$-balanced weighted tree for a given sequence of weights. This algorithm handles strictly positive weights (integers or reals). It is an online algorithm. Furthermore it does not need any additional storage and the weights do not need to be sorted. This algorithm can be used for weighted strings, where the letters of the strings carry weights. It builds in linear time a balanced weighted tree which leaves are either the letters or some given factors of the initial string. We obtain in linear time a tree which is in most of the cases the well-known Huffman tree.

We can also use our linear algorithm to balance trees representing graphs. In particular, we consider the classes of graphs of bounded clique-width or bounded tree-width, which are widely studied. A graph in one of these classes is represented by a term (or a tree). Using the linear algorithm which builds balanced weighted trees, we are able to balance the decomposition trees of such graphs (on a signature containing a new function symbol and a new constant symbol to deal with contexts). Using balanced decomposition trees yields more compact representations of the graphs and dealing with these balanced

trees often leads to algorithms which have a logarithmic complexity. In particular, building balanced decomposition trees is a crucial point to obtain labeling of graphs with short labels from which one can answer queries expressed in monadic second order logic (with quantification on sets of variables).

[1] A. Anderson, General Balanced Trees, *Journal of Algorithms* 30 (1999) 1-18.

[2] H. L. Bodlaender, Discovering treewidth, *Proceedings of 31st Conf. on Current Trends in Theory and Practice of Comp. Sci.* (2005) 1-16.

[3] R. B. Borie, R. G. Parker, C. A. Tovey, Algorithms on Recursively Constructed Graphs, *CRC Handbook of Graph Theory*, 2003, pp. 1046-1066.

[4] G. S. Brodal, R. Fagerberg, T. Mailund, C. NS Pedersen, D. Phillips, Recrafting the neighbor-joining method, *BMC Bioinformatics* (2006) 7-29.

[5] H. Chen, Quantified constraint satisfaction and bounded treewidth, *ECAI'2004*, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, (R. L. de Mantaras and L. Saitta, ed.), 2004, pp.161-165.

[6] D. G. Corneil, M. Habib, J. M. Lanlignel, B. A. Reed, U. Rotics, Polynomial time recognition algorithm of clique width ≤ 3 graphs, *LATIN'00*, *Lect. Notes in Comp. Sci.* 1776 (2000) 126-134.

[7] B. Courcelle, J. Engelfriet, Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach, *Cambridge University Press*, 2012.

[8] B. Courcelle, S. Olariu, Upper bounds to clique-width of graphs, *Discrete Applied Mathematics* 101 (2000) 77-114.

[9] B. Courcelle, R. Vanicat, Query efficient implementations of graphs of bounded clique-width, *Discrete Applied Mathematics* 131 (2003) 129-150.

[10] M. Crochemore and W. Rytter, Text Algorithms, *Oxford University Press* , New York, 1994.

[11] M. Drmota, W. Szpankowski, Generalized Shannon Code Minimizes the Maximal Redundancy *LATIN'02*, *Lect. Notes in Comp. Sci.* 2286 (2002) 306-318.

[12] T. Gagie, A New Algorithm for Building Alphabetic Minimax Trees *Fundamenta Informaticae* 97 (2009) 321-329.

[13] I. Galperin, R. R. Rivest, Scapegoat trees, *Proc. 4th Ann. ACM-SIAM Symp. Discrete Algorithms* 1993, pp. 165-174.

[14] A. M. Garsia and M. L. Wachs, A New algorithm for minimal binary search trees, *SIAM Journal of Computing* 6 (1977) 622-642.

[15] D. Gusfield, Algorithms on Strings, Trees, and Sequences, *Cambridge University Press*, 1997.

[16] T.C. Hu , L. L. Larmore, J. D. Morgenthaler, Optimal Integer Alphabetic Trees in Linear Time, *ESA 2005*, *Lect. Notes in Comp. Sci.* 3669 (2005) 226-237.

[17] D. A. Huffman, A method for the construction of minimum redundancy codes, *Proc. IRE 40* 10 (1952) 1098-1101.

[18] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen, Solving partial constraint satisfaction problems with tree decomposition. *Networks* 40(3) (2002) 170-180.

[19] T. Lengauer, Combinatorial algorithms for integrated circuits layout, *Wiley-Teubner* , Berlin, 1990.

[20] M. Lothaire, Combinatorics on Words, *Cambridge University Press*, second edition, 1997.

[21] K. Meer, D. Rautenbach, The OBDD size for graphs of bounded Tree- and Clique-width, *IWPEC 2006 Lect. Notes in Comp. Sci.* 4169 (2006) 13-15.

[22] K. Mehlhorn, Sorting and Searching, Data Structures and Algorithms, vol. 1, *Springer-Verlag*, 1986.

[23] D. Mehta and S. Sahni (editors), Handbook on Data Structures and Applications, *CRC Press*, 2005.

[24] M. Nei, N. Saitou, The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees, *Mol Biol Evol* 4(4) (1987) 406-425.

[25] J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM J. on Computing* 2(1) (1973) 33-43.

[26] D. Salomon, Data Compression, The Complete Reference , *Springer-Verlag*, 1998.