Balancing Weighted Trees in linear time

Frédérique Carrère
LaBRI, Université Bordeaux 1

carrere@labri.fr

20 février 2012

Content

- Introduction
- Previous work
  - Huffman trees, Minimax Trees
  - Optimal Alphabetic trees
  - Near Optimal Weighted Trees
- Definition of Balance for Weighted trees
- Some Properties
- Linear Algorithm to build Balanced Weighted Trees
  - Weight Sibling
  - Insertion Algorithm
  - Weight-Balance after insertion
- Experimental Results
- Conclusion

Introduction

Weighted Trees

- A weighted tree is a finite tree which stores weights in the leaves. Weights can be positive integers or reals.
- Let t be a weighted tree with n leaves.
  Let $w_1, w_2, \ldots w_n$ be the weights of the leaves,

  the weight of t, denoted $w(t)$, is the sum $\sum_{i=1}^{n} w_i$.
- The weight of a node v of t is the weight of the subtree $t{\downarrow}v$ rooted in this node :
  $w(v) = w(t{\downarrow}v)$.

Use of Weighted Trees

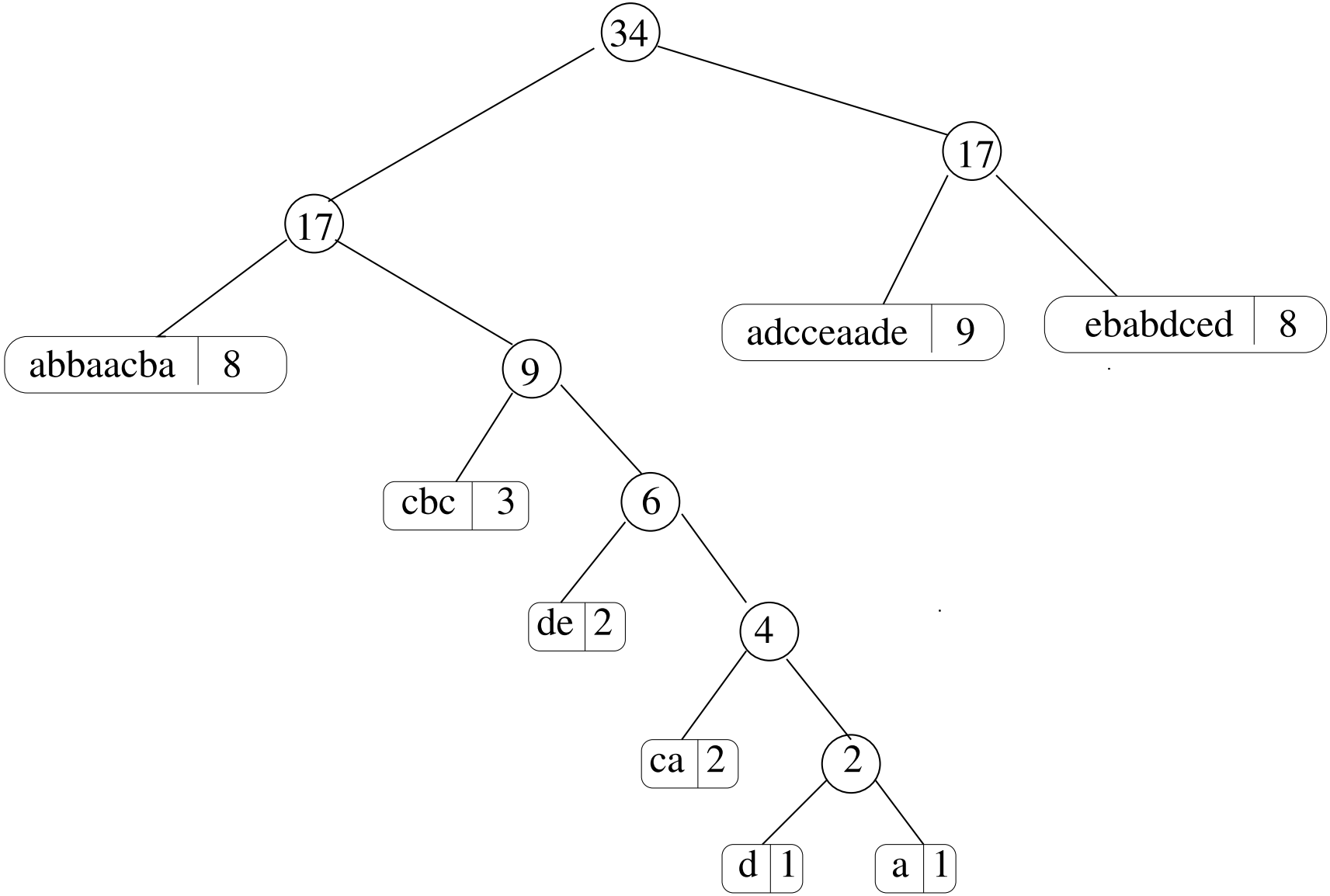
- Well-known Data Structures :
  AVL, B-trees, ...
- Information theory, Data Compression :
  Huffman trees, Minimax Trees.
- Analysis of Biological Structures :
  phylogenetic trees, neighbor-joining algorithm.
- Circuits Design :
  fanout trees (which represent connections between gates in a circuit).

Problem

- Given a weighted tree, define a notion of *weight-balance*.
- Given an ordered sequence of n weights (integers or reals) : $w_1, w_2, \ldots w_n$, build a *balanced weighted* binary tree which leaves hold the weights $w_1, w_2, \ldots, w_n$ from left to right,

  intuitively, a tree such that the value of any leaf's weight plus its depth is as small as possible.

A weight-balanced tree, which is not a balanced binary tree.



Application : Terms

- A term on signature (F, C) is a word :
  $f(a, h(b, g(f(b, b), a)))$, with $f, g, h \in F$, $a, b \in C$.
- It can be cut in factors (contexts or terms) :
  $f(a, \mid h(b, \mid g(f(b, b), \mid a)))$
  with weights 2, 2, 4, 1.
- Build a *balanced* weighted tree which leaves are the above factors with their respective weights.
- Replace each leaf with a tree which represents the factor (adding a rightmost leaf $\bullet$ for the contexts), label each internal node with a new function σ (substitution).
- We get a *balanced* term on the signature $(F \cup \{\sigma\}, C \cup \{\bullet\})$.

Notations

Let t be a binary tree. Let x be a node of t.

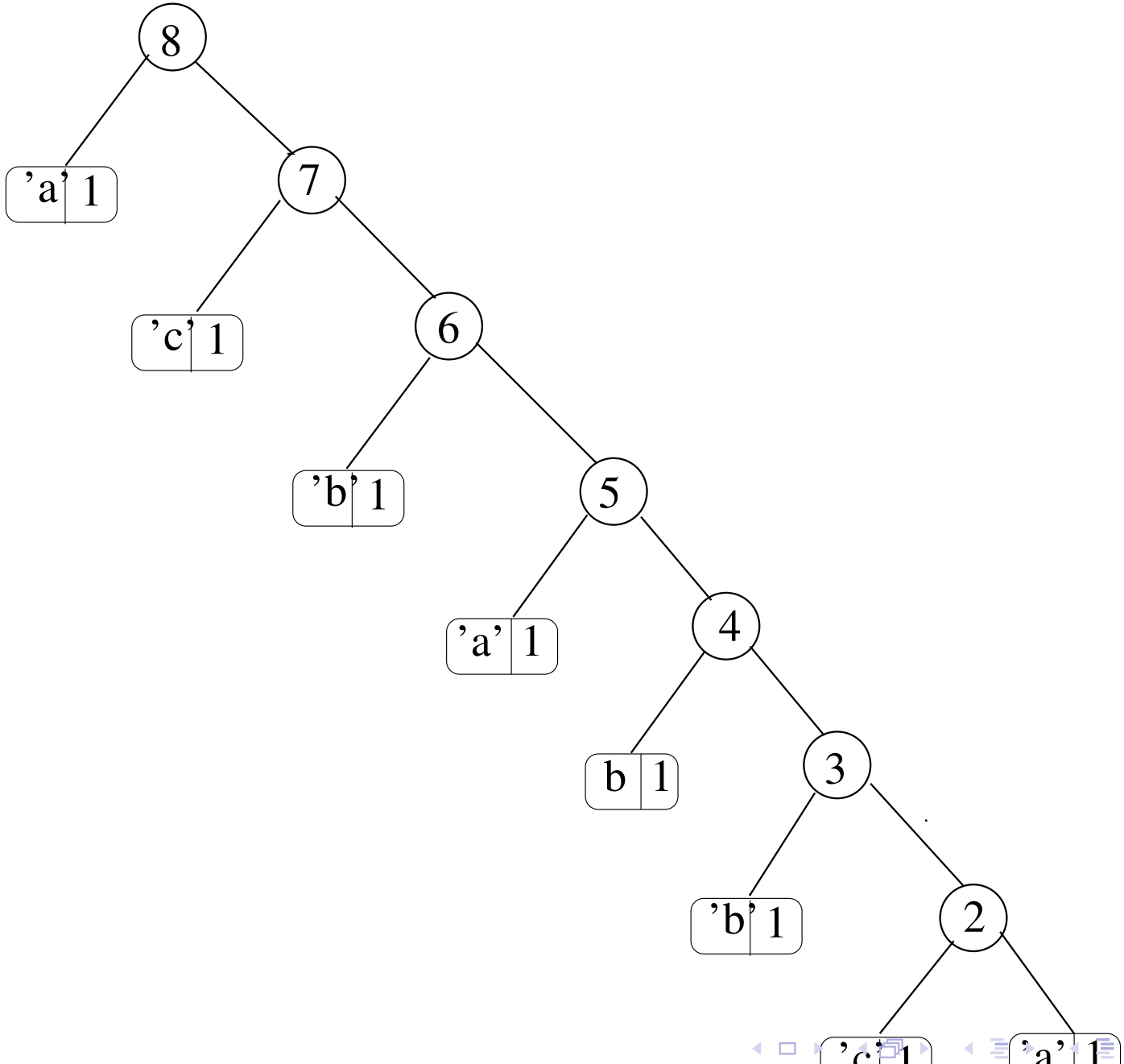
- The size $|t|$ is the number of nodes of t.
- The height $h(x)$ is the length of the longest downward path from x to a leaf. The height of a tree is the height of its root.
- The depth of x is the length of the path from x to the root.
- $t{\downarrow}x$ denotes the subtree rooted in x.
- $\mathit{left}(x)$ (resp. $\mathit{right}(x)$) denotes the left (resp. right) son of x, $\mathit{sibling}(x)$ denotes the sibling of x,
- $\mathit{parent}(x)$ denotes the parent of x, $\mathit{gparent}(x)$ the grandparent of x and $\mathit{ggparent}(x)$ the great grandparent of x (if it exists).

Content

Binary Search Trees

- The most widely used binary search trees are self-balancing.
- Balancing binary search trees statically : DSW Algorithm
  Day [76], improved by Stout-Warren [86].
- Build a "Vine" (list-like degenerated tree) from the binary search tree,
  then complete each level of a new binary tree from the "Vine".
- The algorithm is linear.
- All the leaves have the same weight.

Example of Vine :



Huffman Trees

- A well-known compression algorithm : Huffman Coding.
- Build a Huffman Tree, e.g. a tree which minimizes the weighted average of the leaves' depths :

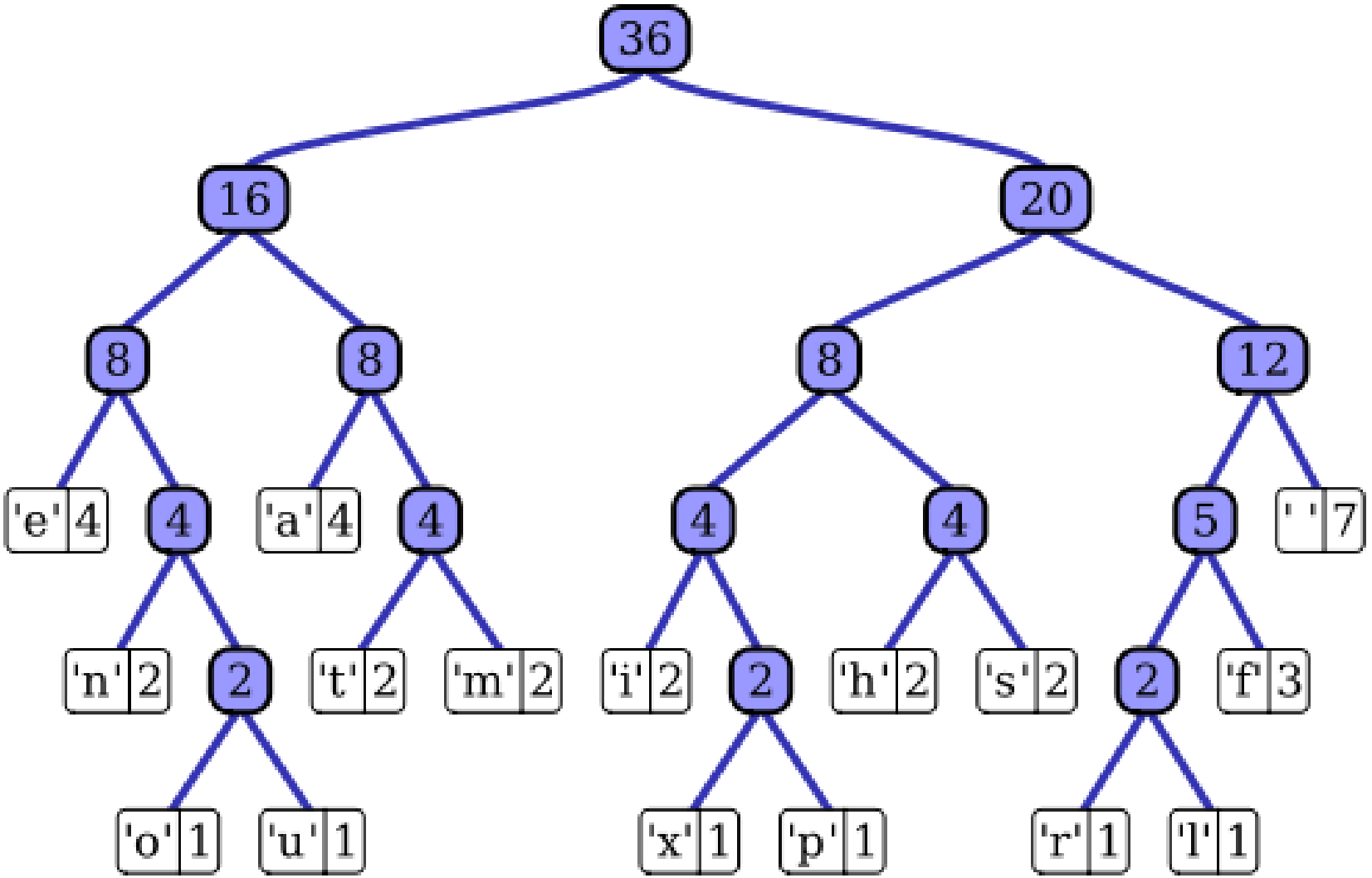
  $$\sum_{i=1}^{n} w_i \cdot \ell_i$$

  where ℓ_i is the length of the path from the root to the i-th leaf.
- Huffman [52] : an $\mathcal{O}(n \log n)$-time algorithm, improved in $\mathcal{O}(n)$-time if the weights are sorted.

Huffman Algorithm

- Construct a forest of singleton trees, each per weight.
- Repeatedly choose the two trees of least weights and add them as left and right sons of a new root node.
- The weight of the any new node is the sum of the weights of its two sons.
- To improve the complexity, first build a sorted list of the given weights.

Huffman Tree : an exemple



Minimax Trees

- a Minimax Tree is a tree which minimizes the maximum of any leaf's weight plus its depth :
  $max_{1 \leq i \leq n}\{ w_i + \ell_i \}$.
- Golumbic [76] : introduced minimax trees and gave a Huffman-like, $\mathcal{O}(n \log n)$-time algorithm for building them (with real weights).
- Drmota-Szpankowski [02] : gave another $\mathcal{O}(n \log n)$-time algorithm, which takes $\mathcal{O}(n)$-time when the weights are integers or are sorted by their fractional parts.
- Gagie-Gawrychowski [09] : gave an $\mathcal{O}(n.d)$-time algorithm for building minimax trees for unsorted real weights, where d is the number of distincts integer parts of the weights.

Ordering of leaves

- Huffman trees, Minimax trees :
  do not preserve the ordering of the leaves.

- Optimal Alphabetic Trees :

  - minimizes a cost function depending on the leaves weights and their depth,

  - preserve the ordering of the leaves.

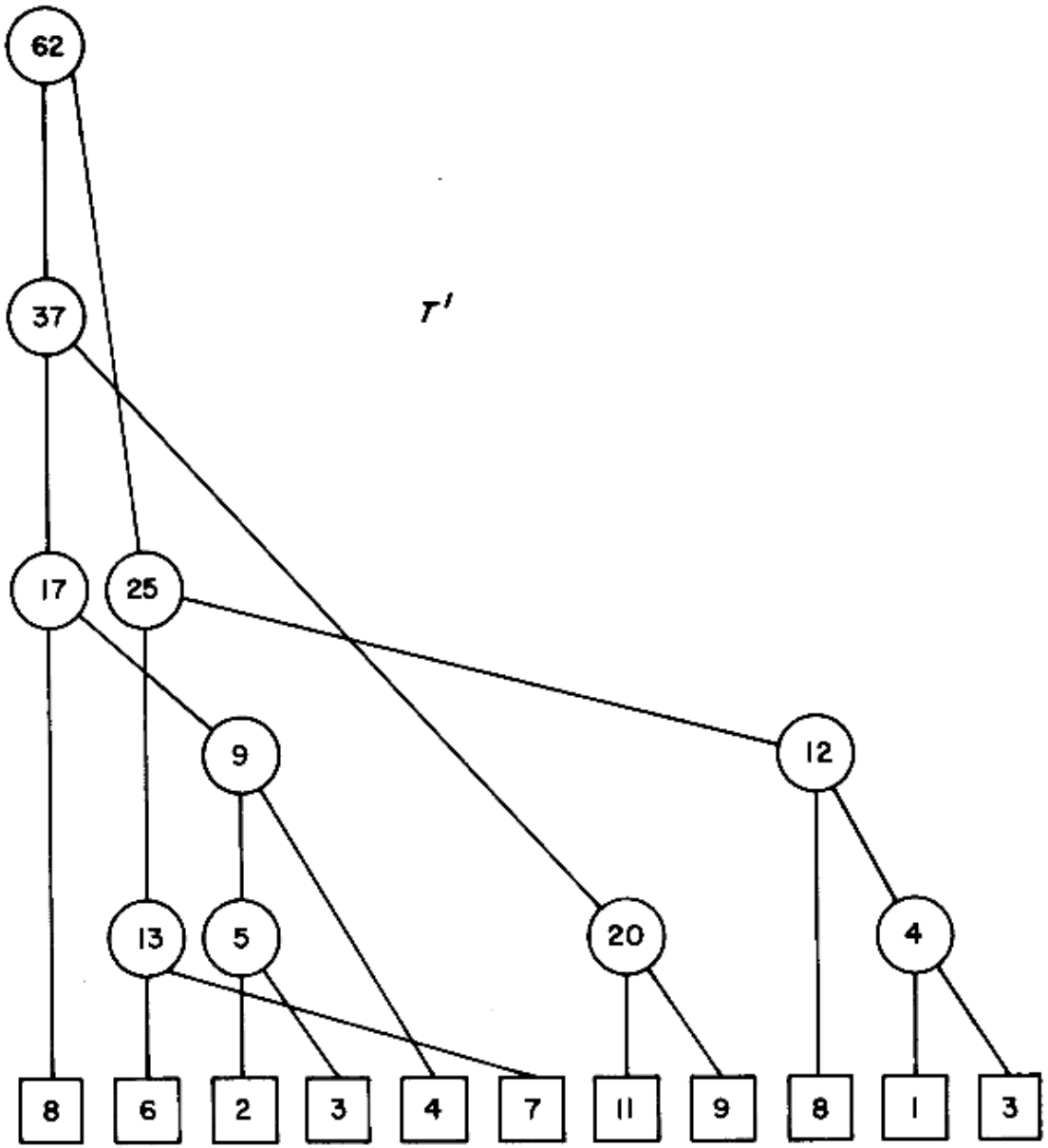
Optimal Alphabetic Trees

- Hu-Tucker [71] and Garsia-Wachs [77] : gave an $\mathcal{O}(n \log n)$-time algorithm (using priority queues) for building alphabetic Huffman trees with real weights.
- Hu-Larmore-Morgenthaler [05] : gave an $\mathcal{O}(n)$-time algorithm for building alphabetic Huffman trees if the weights are integers or can be sorted in linear time.
- Kirkpatrick-Klawe [85], Coppersmith-Klawe-Pippenger [86] : gave an $\mathcal{O}(n)$-time algorithm for building alphabetic minimax trees with integer weights.
- Gagie [09] : gave an $\mathcal{O}(d.n \log(\log n))$-time algorithm for building alphabetic minimax trees where d is the number of distincts integer parts of the weights.

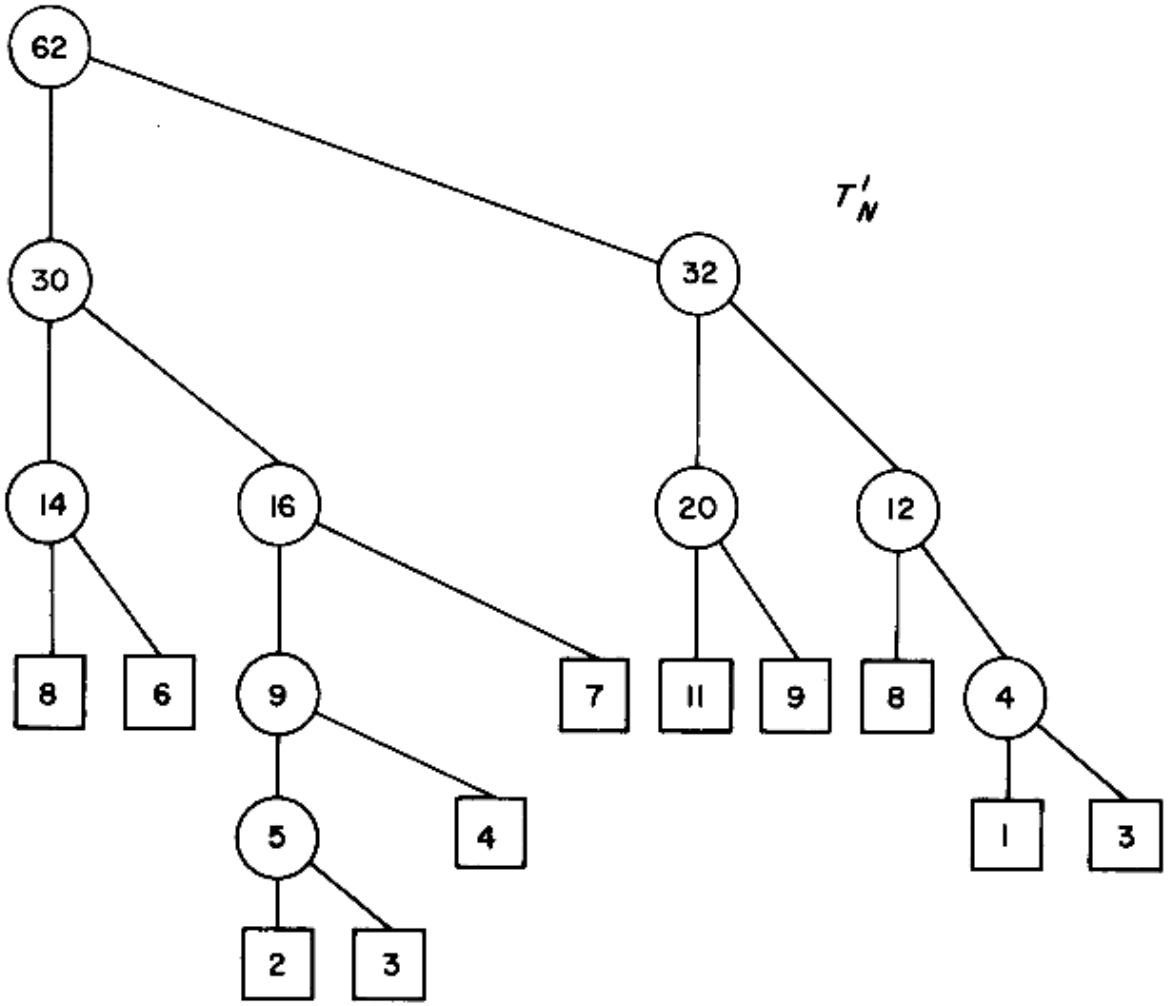
Garsia-Wachs algorithm

- Construct a forest of singleton trees, each per weight.
- A *locally minimal pair* (lmp) in the forest is a pair (t_i, t_{i+1}) which weight is less than the weight of the preceding pair and strictly less than the weight of the following pair.
- Find the leftmost lmp $(i, i+1)$, add the two nodes of the pair as left and right sons of a new node x, replace (t_i, t_{i+1}) by t_x in the forest.
- Move t_x in the forest to be the predecessor of the nearest right tree of larger or equal weight.
- Exchange some leaves, wich weights are the same, in the resulting forest to obtain an alphabetic mimimal tree.

Hu-Tucker : an exemple



Hu-Tucker : an exemple



Hu-Larmore-Morgenthaler

- Use of new data structures which are associated with certain parts of the sequence, which are called mountains and valleys

  valleys $\longrightarrow$ local minima (cf. Garsia-Wachs)

  montains $\longrightarrow$ local maxima.

- Nodes are sorted within the different data structures, nodes can be move from one data structure to another,.

- Processing of data stuctures in constant amortized time per node, then the next locally minimal pair can be found in constant time per node

  $\Rightarrow$ linear amortized time.

Weight-balanced Binary Search Trees

- Different definition of Weights :
  the leaves do not hold weights, the weight of a subtree is either its size or the number of its leaves.
- Different rebalancing algorithms, some of them have linear amortized complexity.
- Implemented (in haskell, scheme) for dictionaries.

Two well-known classes :

→ $BB[\alpha]$-trees (Trees of Bounded Balance) :
Nievergelt-Reingold [73], Mehlhorn [84].

→ Scapegoat trees : Galperin-Rivest [93].

Rebalancing schemes

- A rebalancing scheme = a definition of balance + a rebalancing algorithm applied after each insertion.
- $BB[\alpha]$-trees, with $0 < \alpha < 1/2$:
  - for any node x, $\alpha \leq w(right(x)) / w(x) \leq 1 - \alpha$,
  - rebalancing algorithm : bottom-up from inserted node x, processes every unbalanced node using rotations,
  - $\mathcal{O}(n)$ amortized complexity.
- Scapegoat trees (with parameter ω) :
  - for any node x, $w(left(x)) \leq \omega \,.\, w(right(x))$ and $w(right(x)) \leq \omega \,.\, w(left(x))$
  - rebalancing algorithm : bottom-up, searches the first unbalanced node y, preforms a total rebuilding of the subtree $t{\downarrow}y$,
  - $\mathcal{O}(n \log n)$ amortized complexity.

$BB[\alpha]$-trees

- $BB[\alpha]$-trees t are *c-height-balanced* :

Theorem (Mehlhorn - 89)

There exists a constant $c > 1$ such that for every $BB[\alpha]$-tree t, $h(t) \leq c.log(|t|)$.

For $\alpha = 1/3$, $c = 1.70$.
For $\alpha = 1/4$, $c = 2.40$.

- Linear amortized complexity : adding up costs for several operations $\Rightarrow$ fast on average.

Theorem (Mehlhorn - 89)

For $1/4 \leq \alpha \leq 1 - \sqrt{2}/2$, there exists a constant K such that the number of rotations during a sequence of m insertions from the empty tree is $\leq K.m$.

For $\alpha = 3/11$, $K = 19$.

Content

- Introduction
- Previous work
  - Huffman trees, Minimax Trees
  - Optimal Alphabetic trees
  - Near Optimal Weighted Trees
- Definition of Balance for Weighted trees
- Some Properties
- Linear Algorithm to build Balanced Weighted Trees
  - Weight Sibling
  - Insertion Algorithm
  - Weight-Balance after insertion
- Experimental Results
- Conclusion

Another Balancing scheme

We define the *weight-balance of a node* as the ratio of right to left weights.

Definition

Let t be a weighted tree. Let x be a node of t. The *weight-balance* of x is the ratio $w(right(x))/w(left(x))$.

We define the *balance* of a weighted tree comparing leaves depth and ratio of weights.

Definition

Let t be a weighted tree which leaves hold the weights $w_1, w_2, \ldots w_n$. The tree t is (c, b)-*balanced* for some integers $c \geq 1$, $b \geq 0$, if and only if for all i, $1 \leq i \leq n$, the i-th leaf of t has a depth at most $c.log(w(t)/w_i)+b$.

(We omit b when it is zero)

Lemma (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 substituting any leaf u_i of t with an arbitrary (c, b')-height-balanced binary tree t_i of size $w(u_i)$ gives a $(c, b + b')$-height-balanced binary tree t'.

The height of t is $max\{depth(u_i),\ 1 \leq i \leq p\}$.

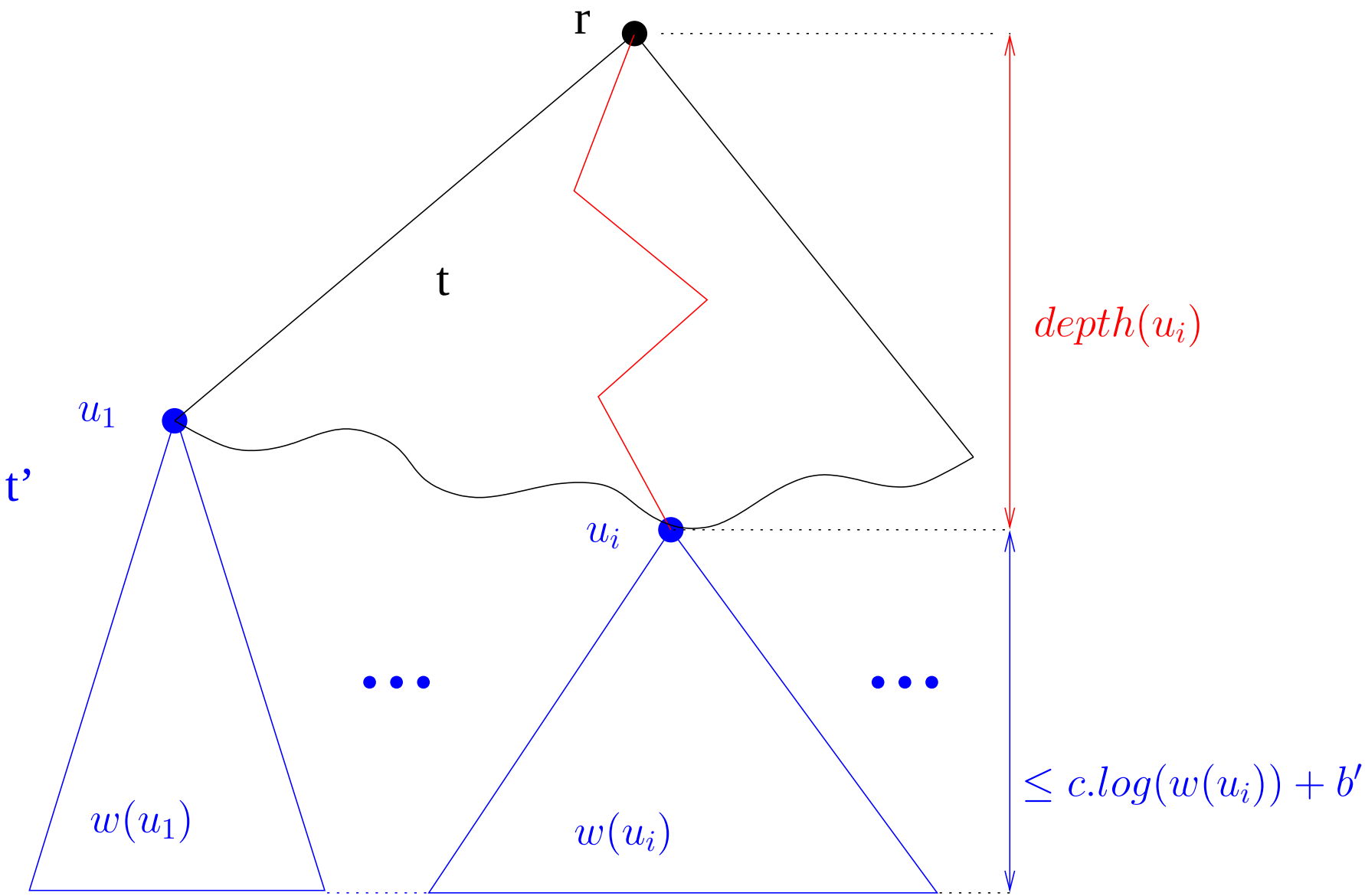
The height of t' is $max\{depth(u_i) + h(t_i),\ 1 \leq i \leq p\}$.

By hypothesis,

$$h(t') \leq max\{\ c.log(w(s)/w(u_i)) + b + c.log(w(u_i)) + b'\ ,\ \ 1 \leq i \leq p\}.$$

$$\Rightarrow \quad h(t') \leq c.log(w(s)) + b + b' \leq c.log(|t'|) + b + b'\ .$$

t' is $(c, b + b')$-height-balanced


r
t
$depth(u_i)$
u_1
t'
u_i
$w(u_1)$
$w(u_i)$
$\leq c.log(w(u_i)) + b'$

Lemma (2)

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 at least c, there exists a node y such that :

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

then t is (c, b)-balanced.

Proof.

Let us consider a branch of length greater than c.
Let u_i be the leaf on this branch and $w_i = w(u_i)$.
By hypothesis, there exists a node y at depth at most c such that $t{\downarrow}y$ is (c, b)-balanced and $w(t{\downarrow}y) \leq w(t)/2$.

Either y ancestor of w_i or $y = w_i$.

- if y is an ancestor of u_i :

$$\Rightarrow \quad depth_{t\downarrow y}(u_i) \leq c.log(\ w(t\downarrow y)/w_i\ ) + b$$

$$\begin{aligned} \Rightarrow \quad depth(u_i) &\leq depth(y) + c.log(\ w(t\downarrow y)/w_i\ ) + b \\ &\leq c + c.log(\ 1/2 \times w(t)/w_i) + b \\ &\leq c.log(\ w(t)/w_i) + b. \end{aligned}$$

- if $y = u_i$ with $w(t\downarrow y) = w_i \leq w(t)/2$

$$\Rightarrow log(w(t)/w_i) \geq 1$$

$$\Rightarrow \quad depth(u_i) = depth(y) \leq c \leq c.log(w(t)/w_i)$$

So t is (c, b)-weight-balanced.

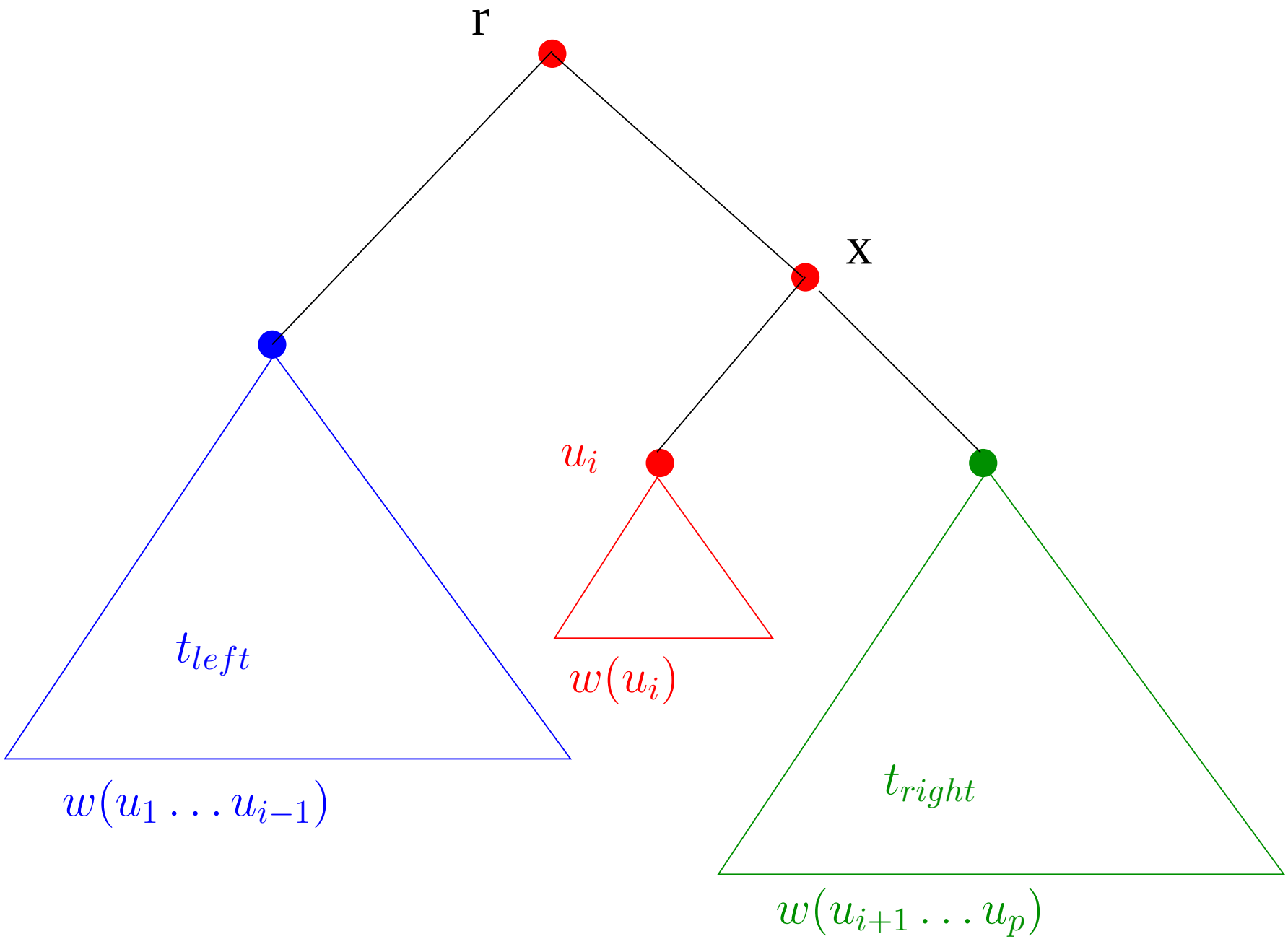
Inductive algorithm

Corollary (3)

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 $w_1, w_2, \ldots, w_n$.

The Algorithm : Let $w = \sum_{k=1}^{n} w_k$.

- find the least index i such that $\sum_{k=1}^{i} w_k > w/2$,
- run the algorithm to build $(2,2)$-balanced trees
  - for $w_1, \ldots, w_{i-1}$: we get t_{left},
  - for $w_{i+1}, \ldots, w_n$: we get t_{right},
- build a new node x with left son w_i and right son t_{right},
- build a new root r with left son t_{left} and right son x.


r
x
u_i
t_{left}
$w(u_i)$
t_{right}
$w(u_1 \dots u_{i-1})$
$w(u_{i+1} \dots u_p)$

Content

- Introduction
- Previous work
  - Huffman trees, Minimax Trees
  - Optimal Alphabetic trees
  - Near Optimal Weighted Trees
- Definition of Balance for Weighted trees
- Some Properties
- Linear Algorithm to build Balanced Weighted Trees
  - Weight Sibling
  - Insertion Algorithm
  - Weight-Balance after insertion
- Experimental Results
- Conclusion

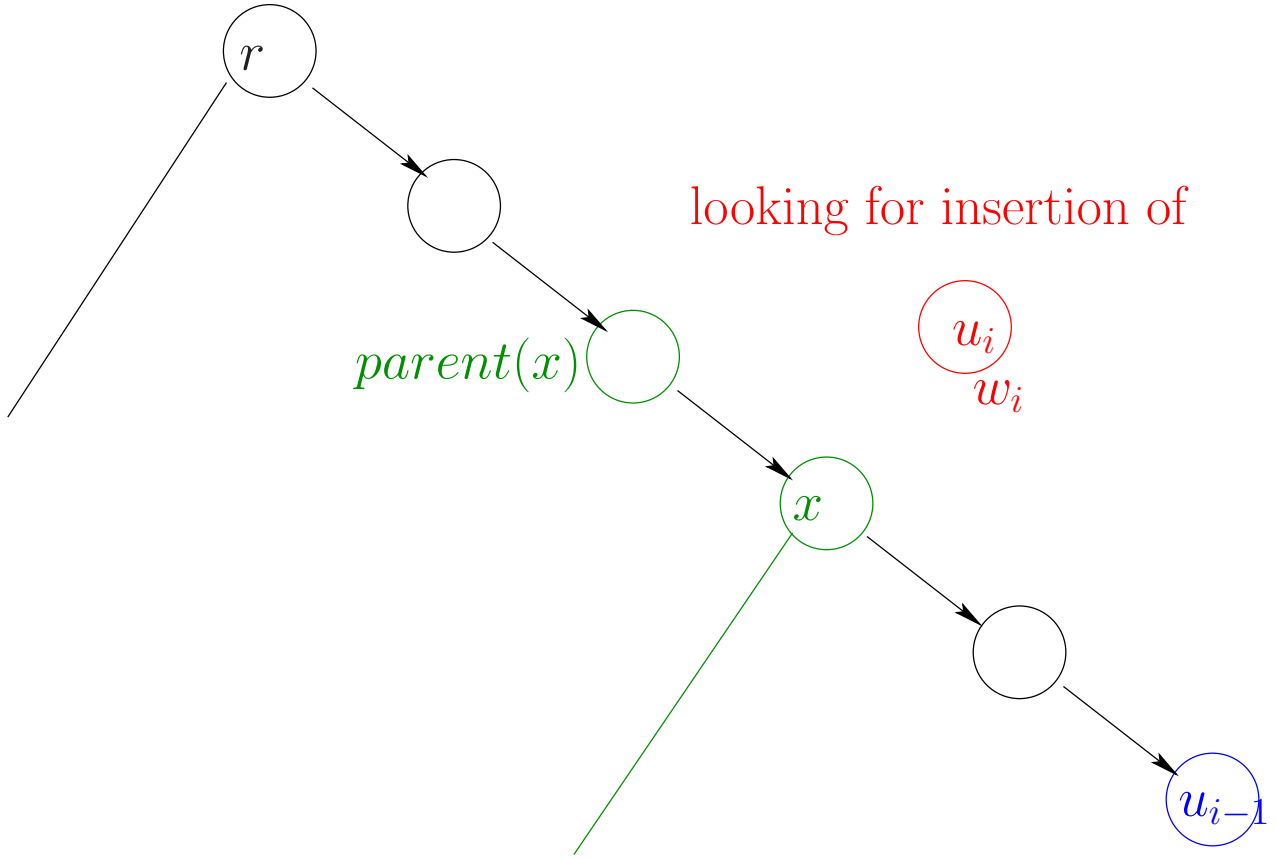
Theorem

For every sequence of weights $w_1, w_2, \ldots, w_n$, one can build in time $O(n)$ a $(3,2)$-balanced weighted tree with leaves $w_1, w_2, \ldots, w_n$.

The Algorithm :

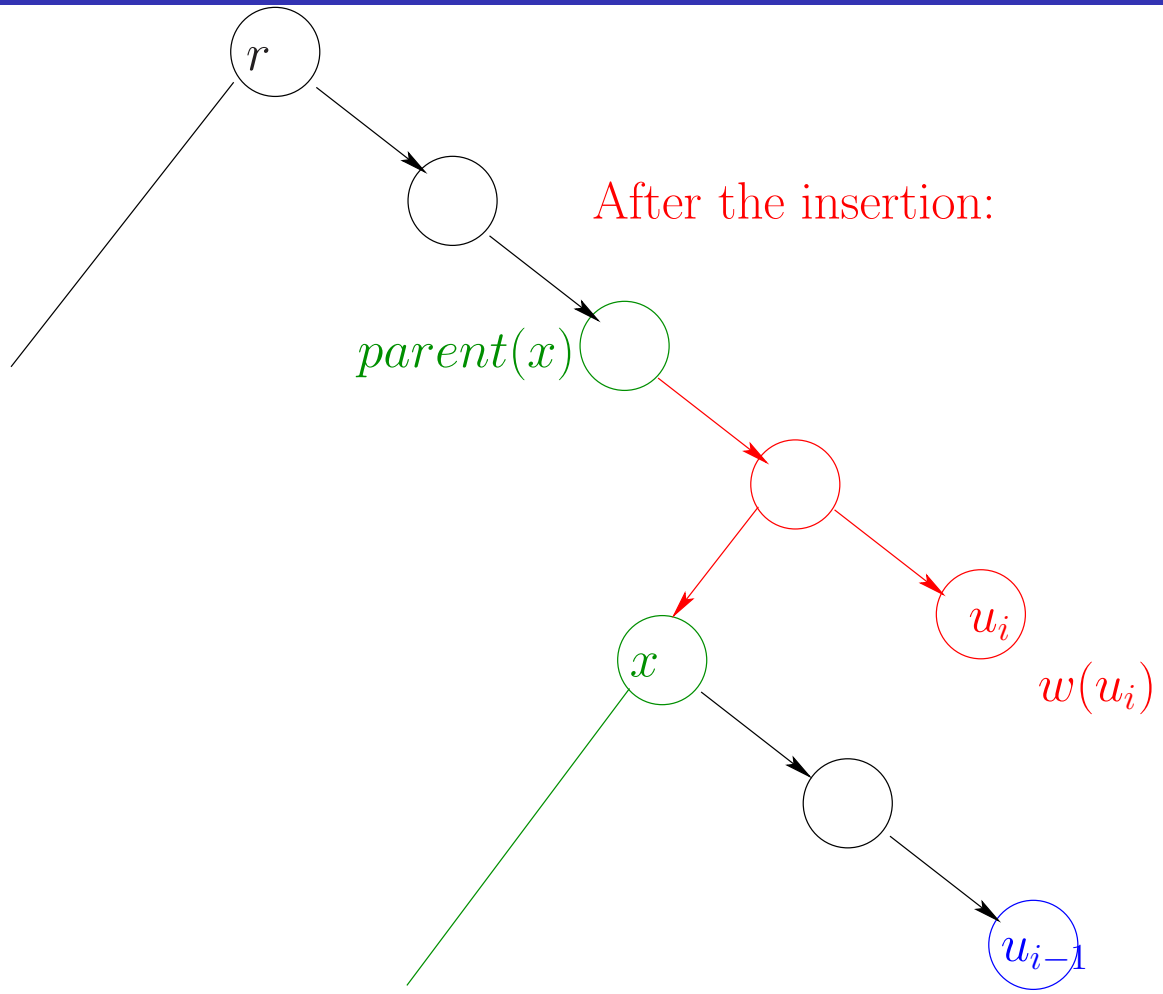
- create a leaf tree node t_1 holding the weight w_1.
- assume that we have built a balanced tree t_{i-1} for the weights $w_1 \ldots w_{i-1}$, climb up from the rightmost leaf to the root, searching a point of insertion for a new node u_i holding the weight w_i.
- the point of insertion is the first 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.

Insertion of a new node



The insertion : If x is the weight-sibling of u_i, insert u_i between x and its parent.

Insertion of a new node



The insertion : If x is the weight-sibling of u_i, insert u_i between x and its parent.

Weight-Sibling of a node

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

Recall that the weight-balance of a node y is the ratio $w(right(y))/w(left(y))$.

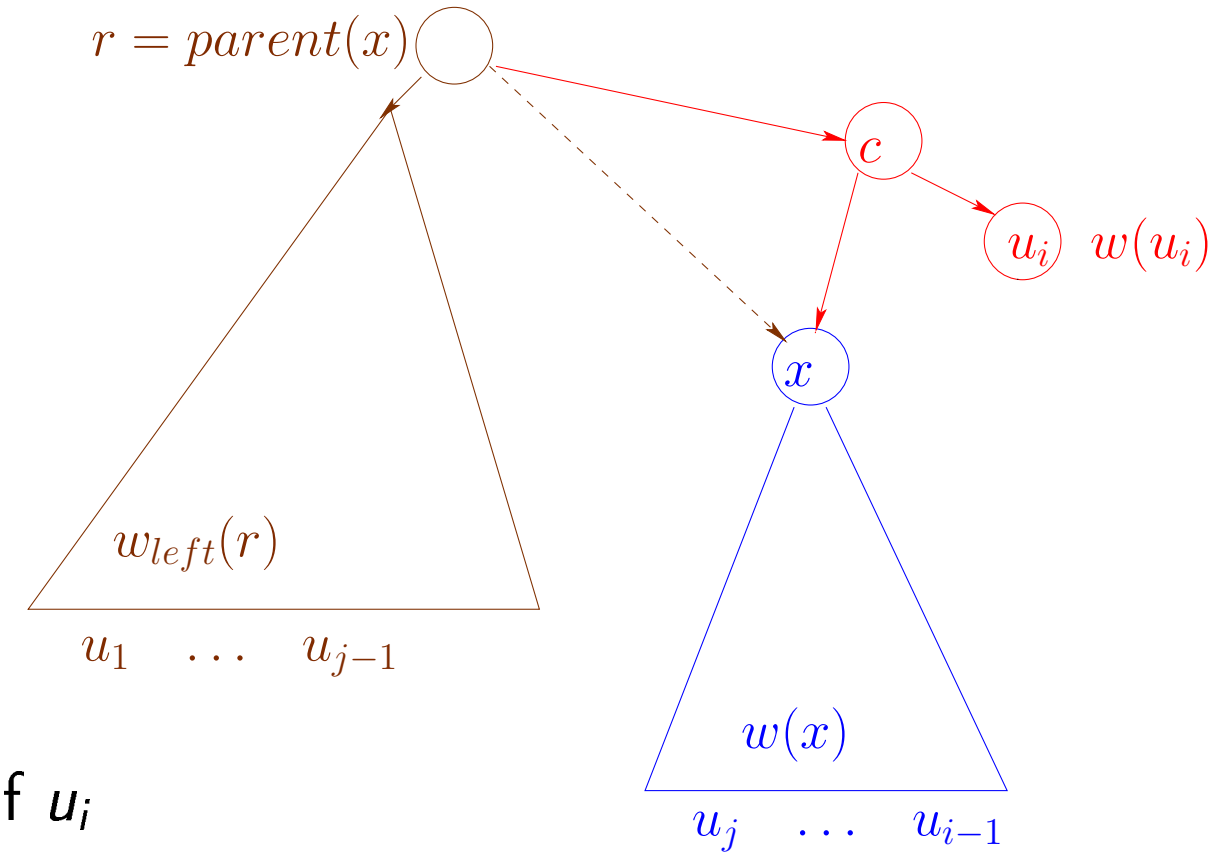
Definition

The node x is a weight-sibling of the node u (u not in t) if and only if x belongs to the right branch of t, $depth_t(x) \geq 1$ and inserting u as sibling of x in t, gives a tree t' such that :

- if $depth_{t'}(x) = 2$ then $gparent_{t'}(x)$ has weight-balance less than 1.
- if $depth_{t'}(x) \geq 3$ then $gparent_{t'}(x)$ and $ggparent_{t'}(x)$ have weight-balance less than 1.

Insertion of u_i as sibling of a node x of t_{i-1}

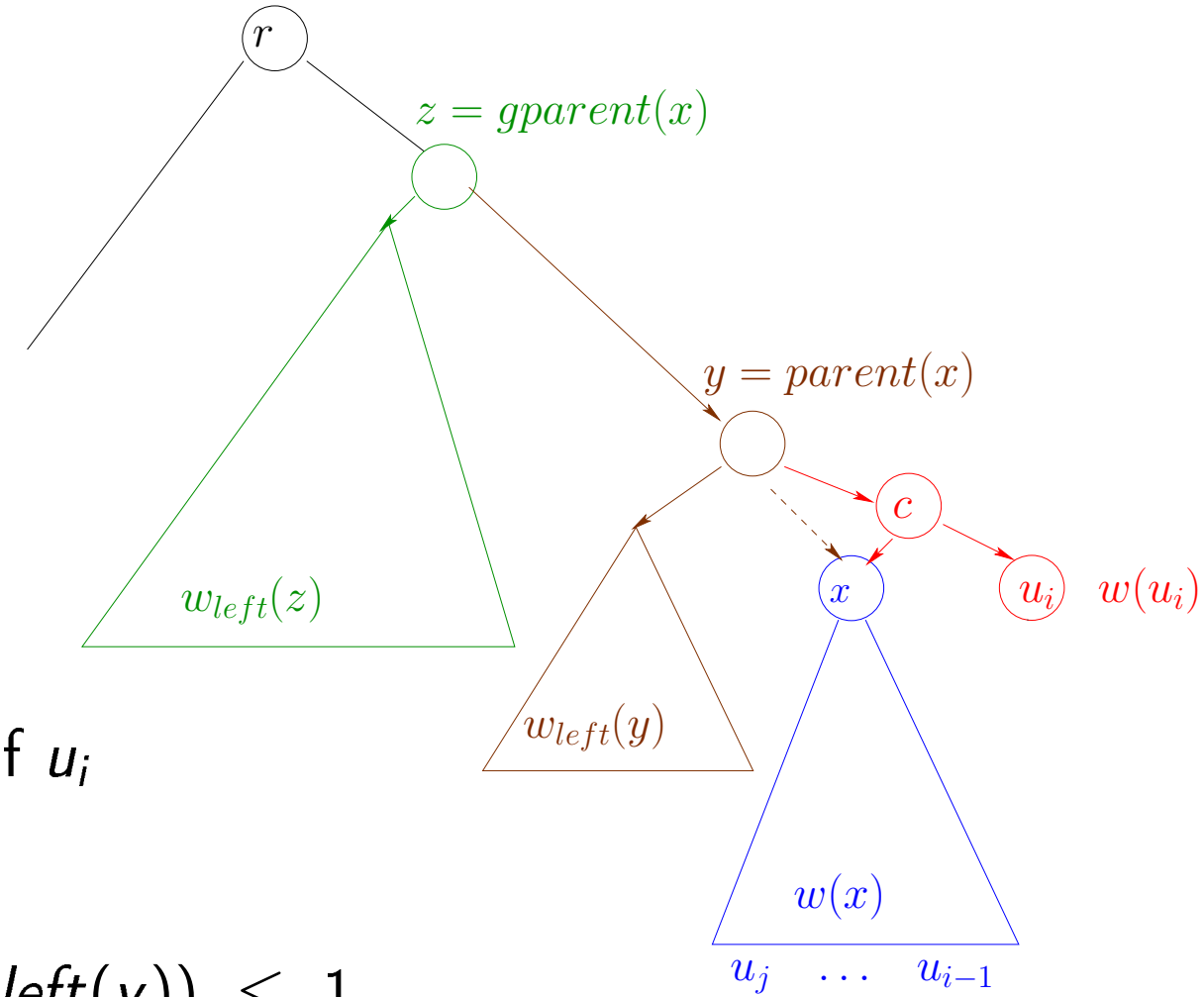
Case $depth t_{t_i}(x) = 2$:



x weight-sibling of u_i
if and only if
after insertion, $w(right(r))/w(left(r)) \leq 1$

Insertion of u_i as sibling of a node x of t_{i-1}

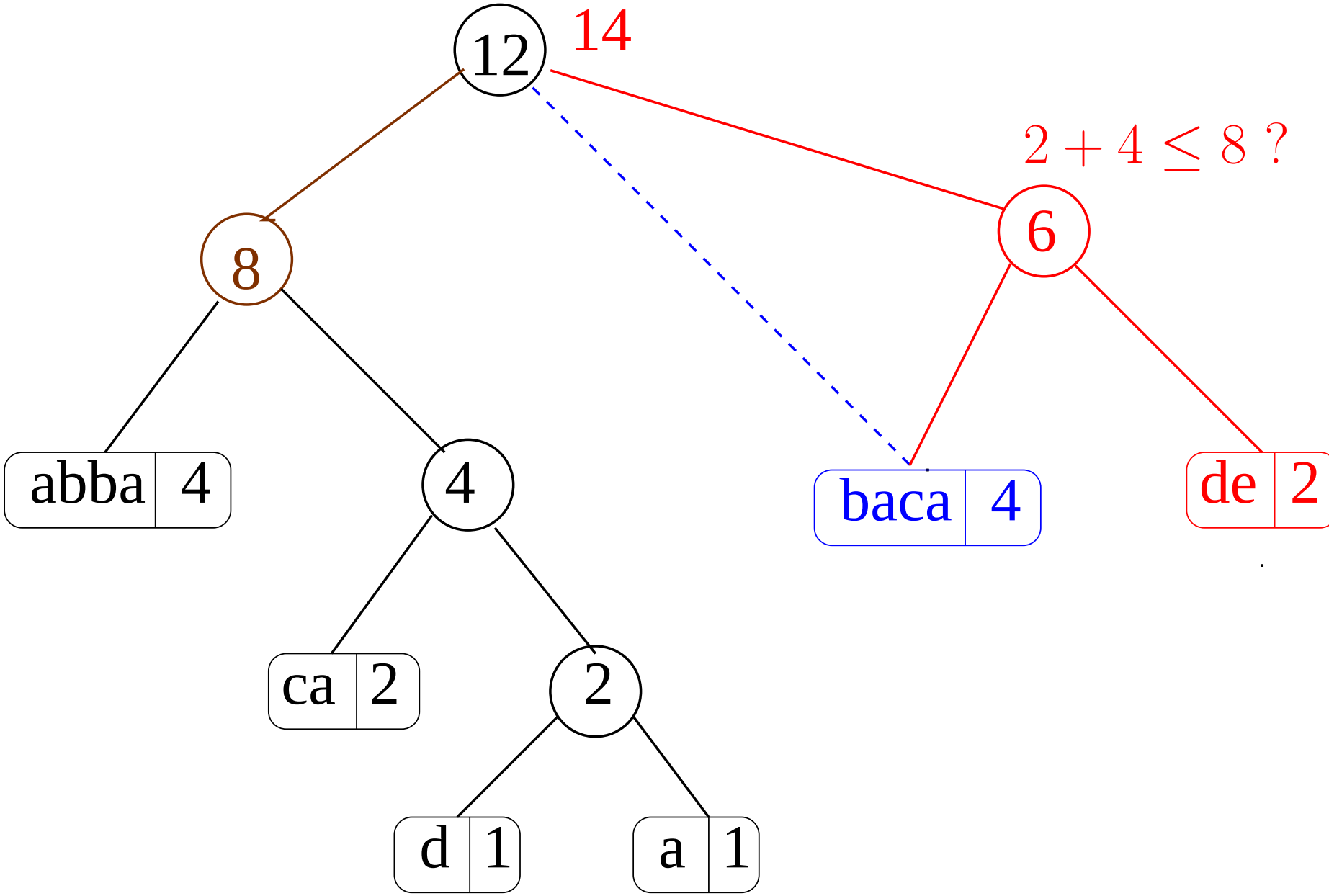
Case $depth_{t_i}(x) \geq 3$:



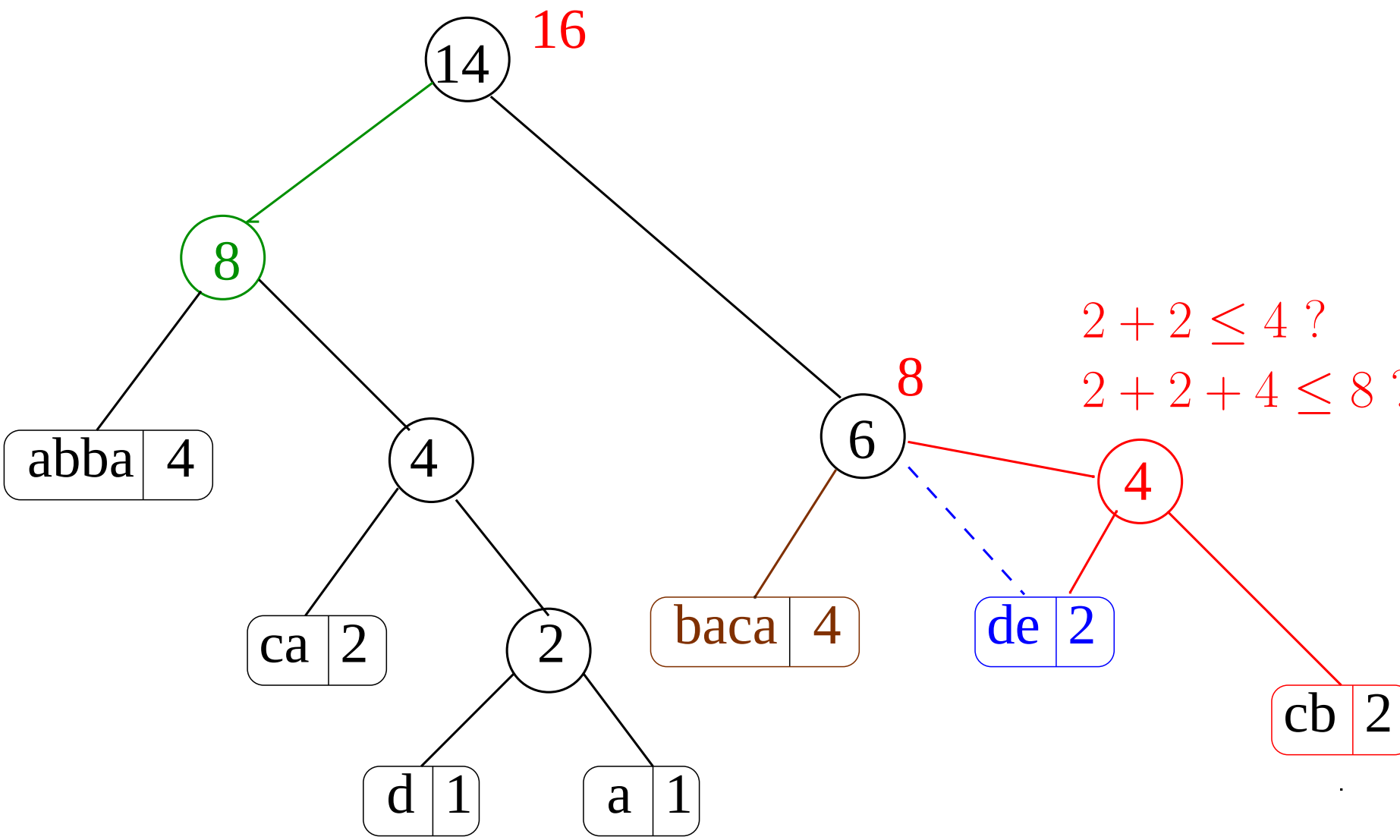
x weight-sibling of u_i
if and only if
after insertion,

$$w(right(y))/w(left(y)) \leq 1$$
$$w(right(z))/w(left(z)) \leq 1$$

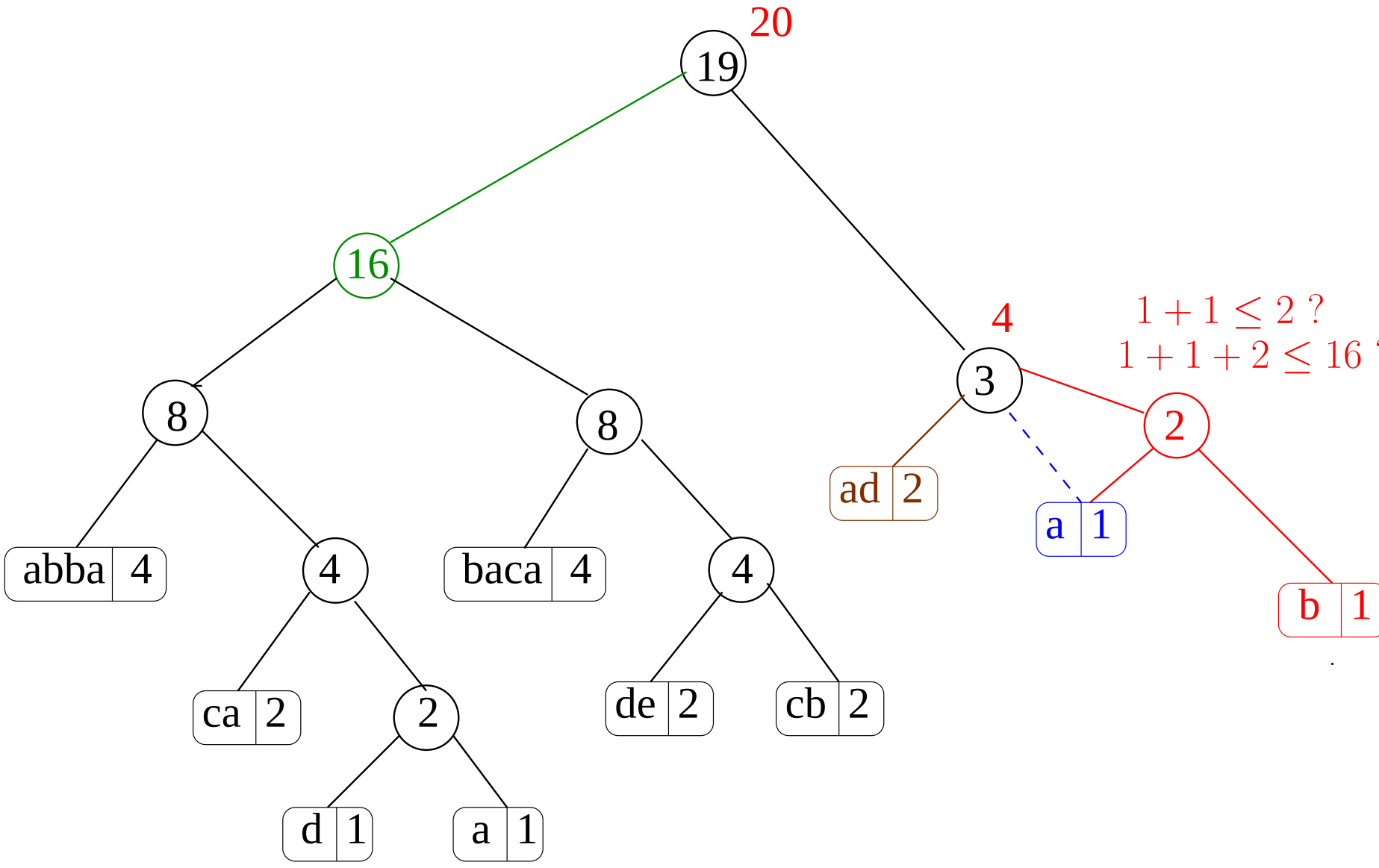
Insertion : an exemple



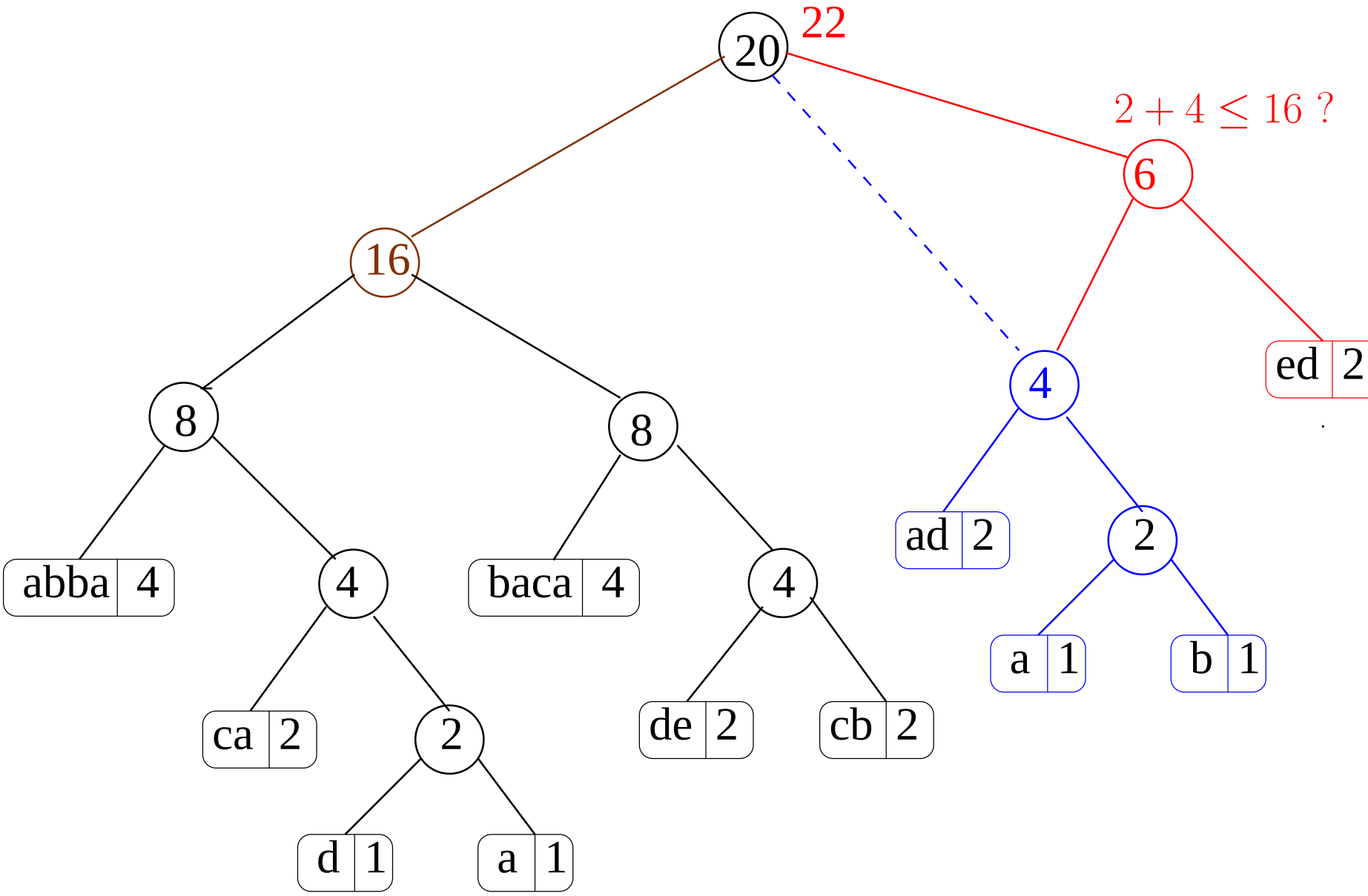
Insertion : an exemple



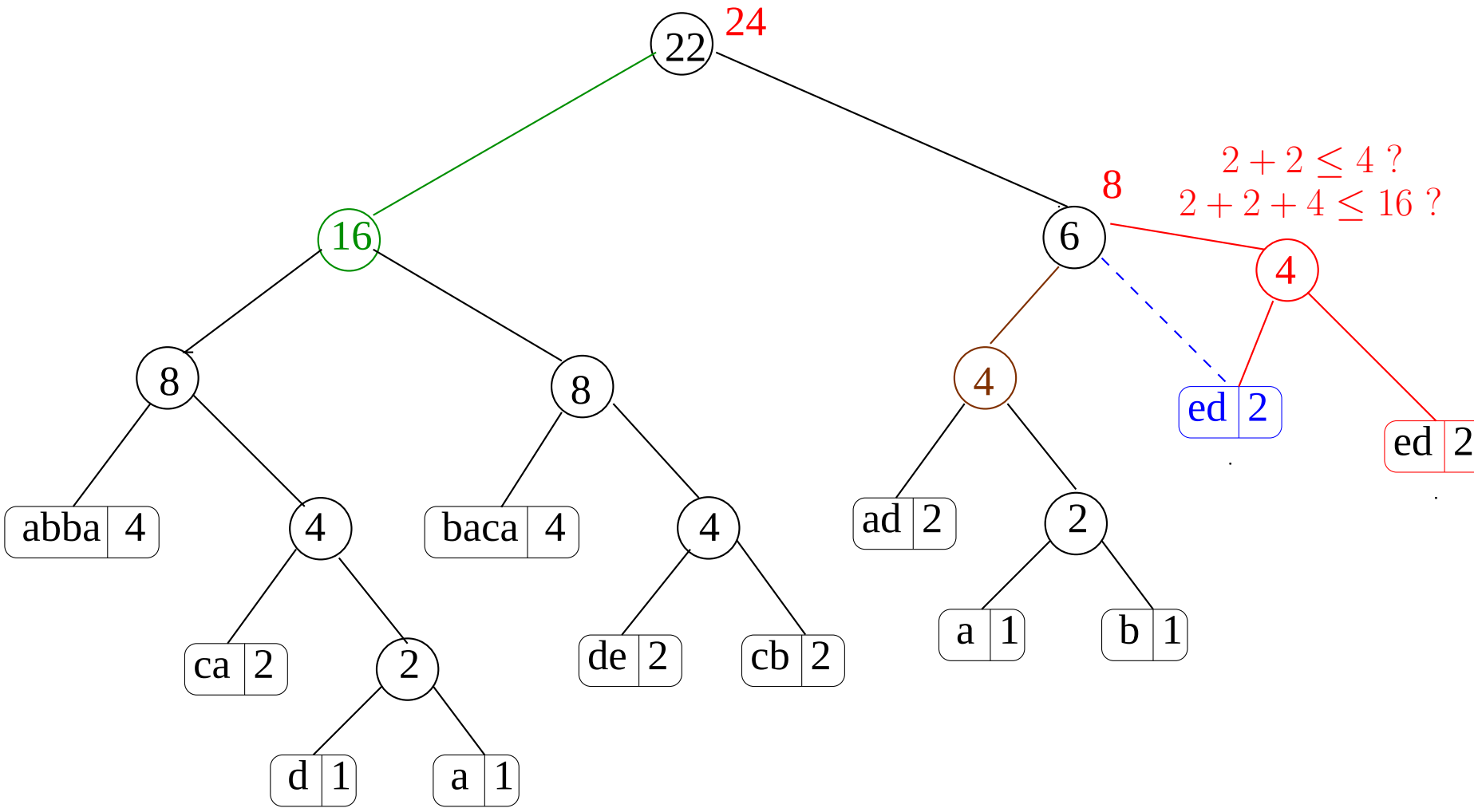
Insertion : an exemple



Insertion : an exemple



Insertion : an exemple



$buildBalancedTree(u_1, u_2, \ldots, u_n)$

Input : A **list** of weighted nodes $u_1, u_2, \ldots u_n$.
Output : A $(3,2)-$balanced weighted tree t.

```
1  t = u_1
2  p = u_1
3  for i = 2 to n do
4      p = findWeightSibling(t, p, u_i)
5      /* the insertion procedure moves p on u_i */
6      if ( parent(p) <> null ) then
7          t = insert(u_i, parent(p))
8      else
9          t = insert(u_i, root(t))
10 end for
```

$FindWeightSibling(t_{i-1}, p, u_i)$

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

```
1  weight = w_{i-1}
2  height = 0
3  while ( parent(p) <> null ) do
4      height = height + 1
5      parentWeight = weight + w(sibling(p))
6      if ( isWeithSibling(p, u_i) ) then
7          return p
8      p = parent(p)
9      weight = parentWeight
10 end while
```

$isWeightSibling(p, u_i)$

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

```
1  q = parent(p)
2  qBalance = (w(p) + w(u_i)) / w(left(q))
3  if ( parent(q) == null ) then
4      return (qBalance ≤ 1)
5  else
6      r = parent(q)
7      rBalance = (w(left(q) + w(p) + w(u_i) / w(left(r))
8      return (qBalance ≤ 1)
9                  && (rBalance ≤ 1)
```

Complexity of $buildBalancedTree(w_1 \dots w_n)$

Claim 1.

The amortized complexity of $buildBalancedTree(u_1 \dots u_n)$ is $\mathcal{O}(n)$.

Amortized complexity = average complexity on a sequence of operations in the worst case.

To compute the amortized complexity :

- 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.

Amortized Complexity

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},
- we work on the righmost branch of t_{i-1}, with a pointeur p on its leaf u_{i-1},
- we have at least k credits saved, where k is the depth of u_{i-1},
- we charge 5 credits to construct t_i from t_{i-1}.

Let u_i be a leaf node holding the weight w_i. We climb up from u_{i-1} to 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 between u_{i-1} and $parent(u_{i-1})$ and move the pointer p to u_i. Cost of test and insertion 3 credits (we look at $gparent(u_{i-1})$), and excess 2 credits saved into account.

- If the weight-sibling of u_i is a proper ancestor x of u_{i-1}, such that $depth(x) = k' < k$, we move from u_{i-1} to $parent(x)$ and insert u_i as son of $parent(x)$. Cost of move, test, and insertion $k - k' + 3$ credits (we look at $w(gparent(x))$).

- Suppose we move from u_{i-1} to the root and insert u_i as sibling of the root. Cost of move and insertion $k + 1$ credits.

In each case we have at least $k' + 2$ credits saved, where $k' + 2$ is the depth of u_i on the righmost branch of t_i.

Balance

Claim 2.

Let $t = buidBalancedTree(u_1 \dots u_n)$, every subtree $t{\downarrow}x$ of t is $(3,2)$-balanced.

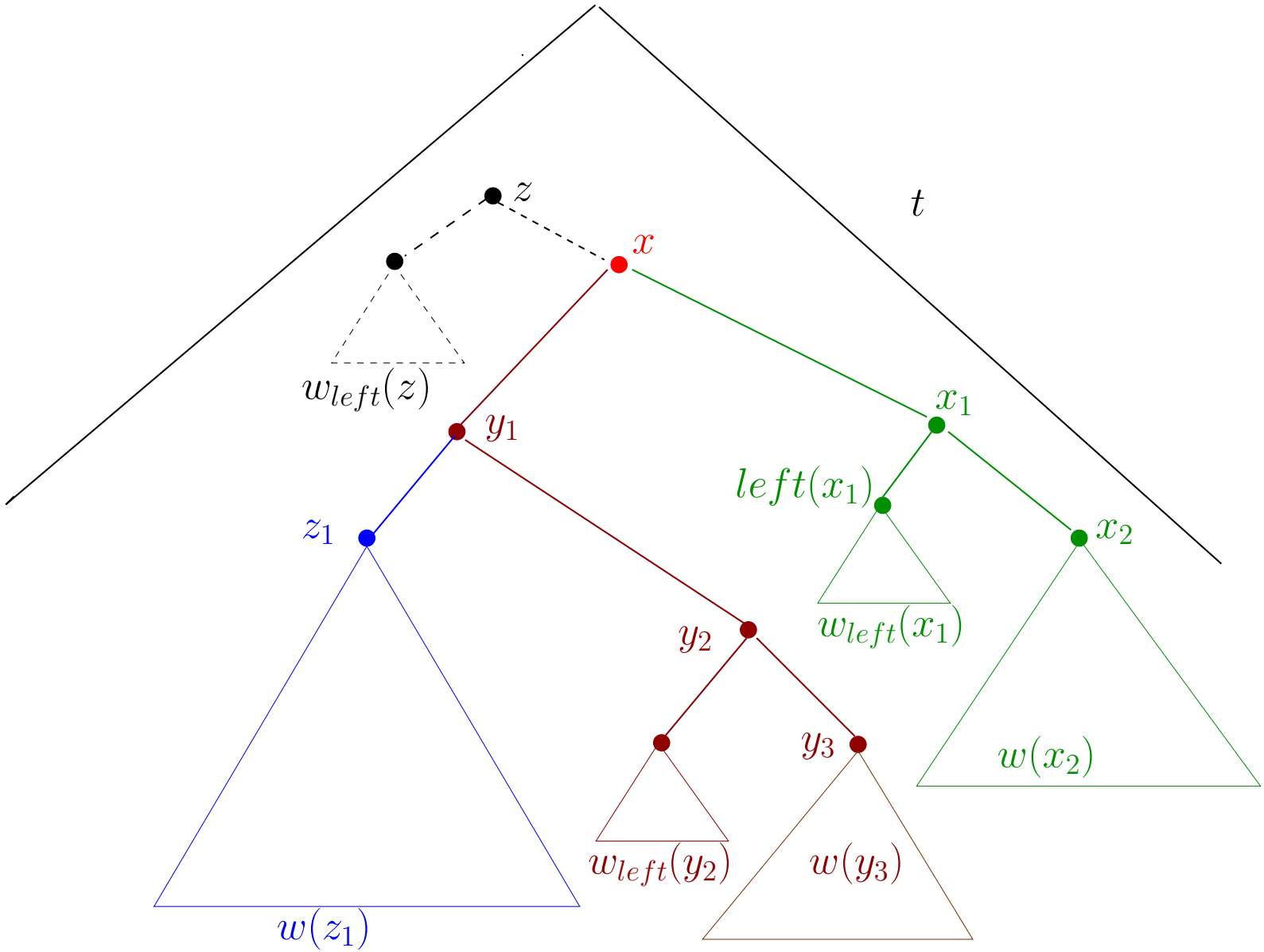
Proof. Induction on the size of the subtree $t{\downarrow}x$.

Basis : If $|t{\downarrow}x| \leq 5$ then $h(t{\downarrow}x) \leq 2$ and the result holds.

Let x be such that $|t{\downarrow}x| = m$, with $m > 5$.
Induction hypothesis : every subtree of t of size strictly less than m is $(3,2)$-balanced.

By Lemma 2, it is sufficient to prove that on every branch of $t{\downarrow}x$ of length at least 3, there exists a node y at depth at most 3, such that : $w(y) \leq w(x)/2$,
(by ind. hyp. $t{\downarrow}y$ is $(3,2)$-balanced).

We prove that $w_{left}(x_1)$, $w(x_2)$, $w_{left}(y_2))$, $w(y_3)$ and $w(z_1)$ are not greater than $w(x)/2$.



Lemma (3)

Let $t = buildBalancedTree(u_1 \dots u_p)$.
Let x *be a node of* t *such that* x *is the right son of a node* z *and the distance between* x *and a leaf of* t *is at least* 2.
Let $x_1 = right(x)$, $x_2 = right(x_1)$, $y_1 = left(x)$ *and* $y_2 = right(y_1)$.
Then one has :

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

Corollary (3)

Let x be a node of t with $left(x) = y_1$.

- *If the length of the right branch of $t{\downarrow}x$ is at least 2, let $x_1 = right(x)$, $x_2 = right(x_1)$. One has :*
  $w_{left}(x_1),\ w(x_2) \leq w(x)/2,$

- *If the length of the right branch of $t{\downarrow}y_1$ is at least 2, let $y_2 = right(y_1)$ and $y_3 = right(y_2)$. One has :*
  $w_{left}(y_2),\ w(y_3) \leq w(x)/2.$

Let A and B be the left and right members of inequalities (1) and (2) of Lemma 1.
Note that A, B are positive numbers such that

$A + B \leq w(x)$ and $A \leq B$

then one has $A \leq w(x)/2$.

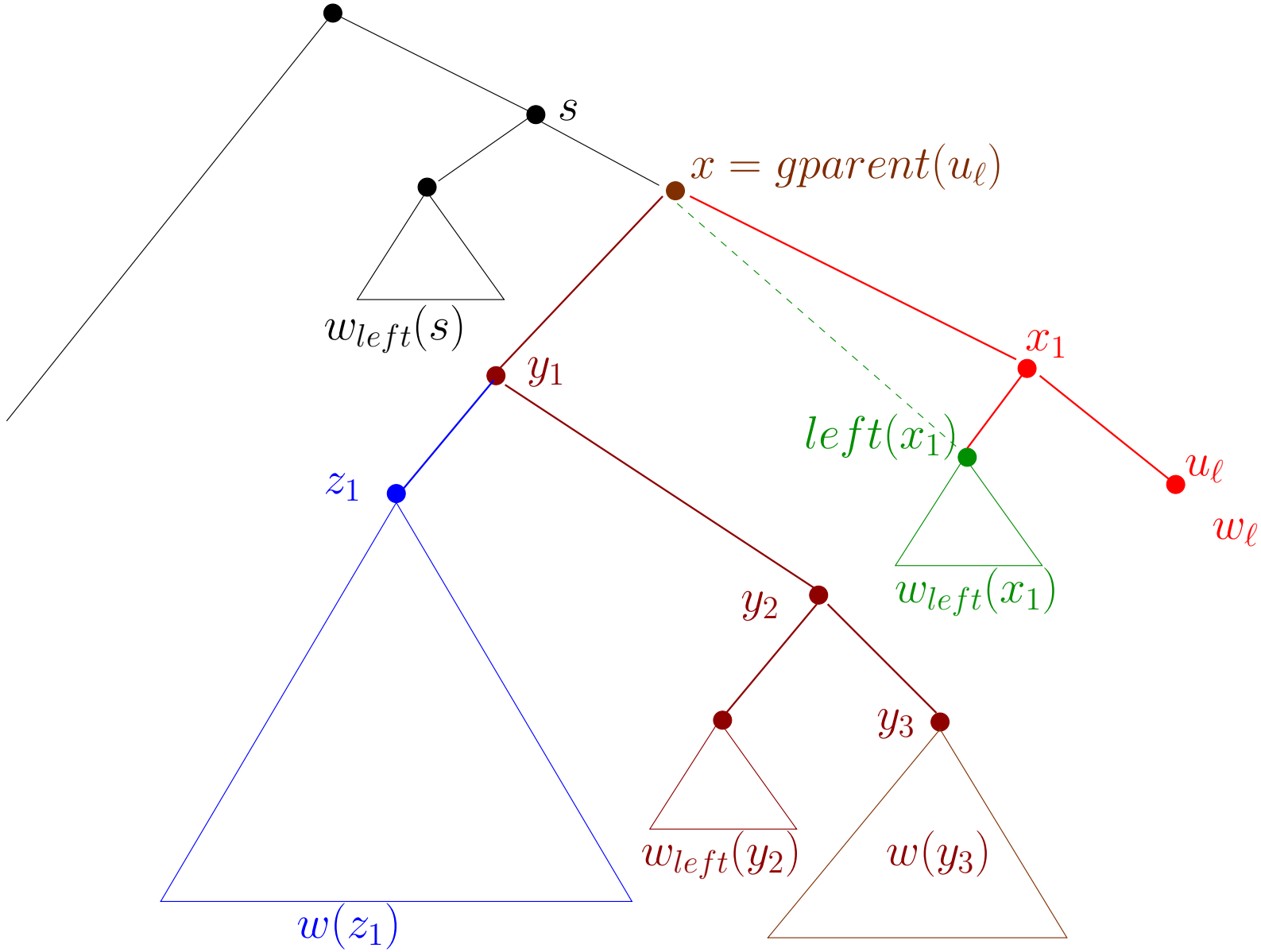
Proof of Lemma 3.

Recall that $x_1 = right(x)$, $x_2 = right(x_1)$.
The tree t is built by successive insertion of leaves.

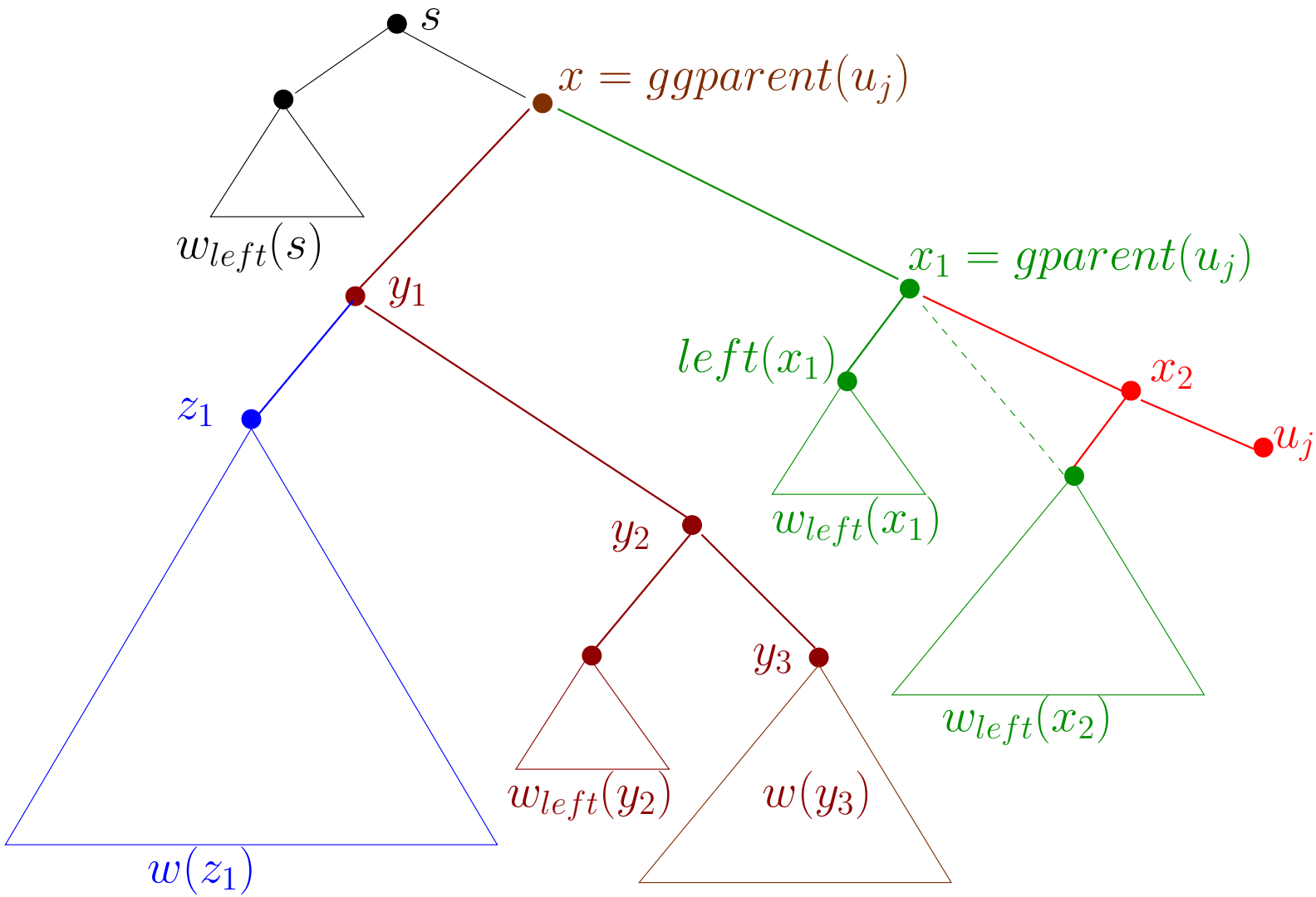
- Let u_ℓ be the leftmost leaf of $t{\downarrow}x_2$, u_ℓ is inserted as right leaf in $t_{\ell-1}$.
  When u_ℓ is inserted as right leaf in $t_{\ell-1}$,
  $gparent_{t_\ell}(u_\ell) = x$ has a weight-balance less than 1.

- Let u_j be the leftmost leaf of $t{\downarrow}right(x_2)$.
  When u_j is inserted as right leaf in t_{j-1},
  $gparent_{t_j}(u_j) = x_1$ has weight-balance less than 1,
  $ggparent_{t_j}(u_j) = x$ has weight-balance less than 1.

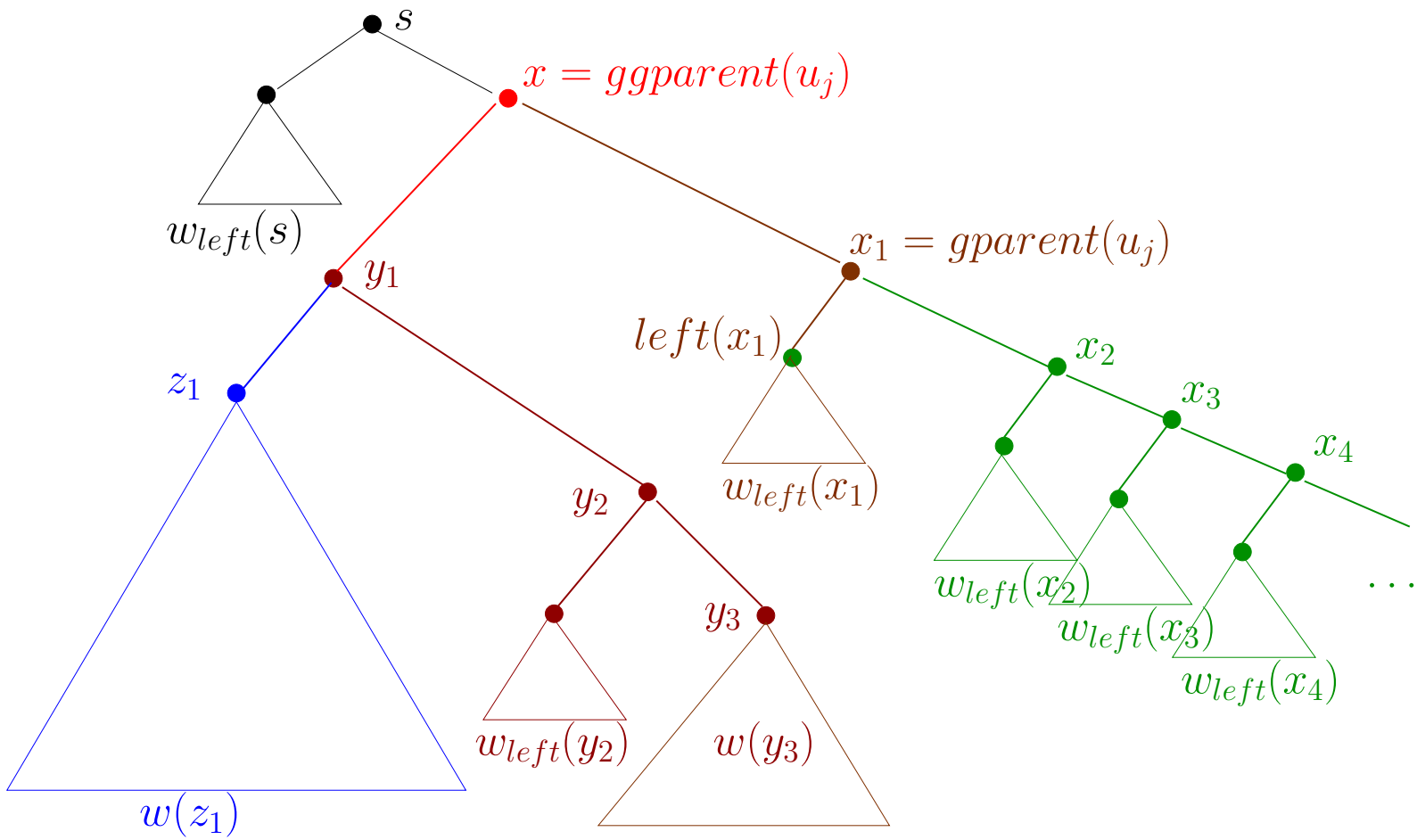
In t_ℓ we have $w_{right}(x) = w_\ell + w_{left}(x_1) \leq w_{left}(x)$.



In t_j we have $w_j + w_{left}(x_2) \leq w_{left}(x_1)$
and $w_j + w_{left}(x_2) \leq w_{left}(x)/2$,



For any i, $2 \leq i \leq m-1$, $w_{left}(x_i) \leq w_{left}(x_{i-2})/2$,



Proof of Lemma 3.3

On the right branch of $t\downarrow x$, we have :
$w(x_2) = w_{left}(x_2) + w_{left}(x_3) + \ldots w_{left}(x_m) + w(a)$.

We prove that $w(x_2) \leq w_{left}(x_1) + w_{left}(x)$.
We know that

for any i, $2 \leq i \leq m-1$, $w_{left}(x_i) \leq w_{left}(x_{i-2})/2$,

Assume that $m \geq 3$ and $2 \leq i \leq m-1$.

- If i is even, $i = 2k$, we get :

$$w_{left}(x_i) \leq \frac{1}{2} w_{left}(x_{i-2}) \ldots \leq \frac{1}{2^j} w_{left}(x_{i-2j}) \ldots \leq \frac{1}{2^k} w_{left}(x_0)$$

- If i is odd, $i = 2k+1$, we get :

$$w_{left}(x_i) \leq \frac{1}{2} w_{left}(x_{i-2}) \ldots \leq \frac{1}{2^j} w_{left}(x_{i-2j}) \ldots \leq \frac{1}{2^k} w_{left}(x_1)$$

Since $gparent(x_m)$ and $ggparent(x_m)$ have left balance 1, we get :

$$w_{left}(x_m) + w(a) \leq w_{left}(x_{m-2})/2$$

Assume that $m \geq 3$.

- If m is even, $m = 2p$, one has :
$$w_{left}(x_m) + w(a) \leq \frac{1}{2^p} w_{left}(x_0)$$
- If m is odd, $m = 2p + 1$, one has :
$$w_{left}(x_m) + w(a) \leq \frac{1}{2^p} w_{left}(x_1)$$

Consequently we have :

- $$\begin{aligned} w(x_2) &= w_{left}(x_2) + w_{left}(x_3) + \dots w_{left}(x_m) + w(a) \\ &= \sum_{i=2}^{i=m-1} w_{left}(x_i) + (w_{left}(x_m) + w(a)) \\ &\leq (w_{left}(x_0) + w_{left}(x_1)) \left(\frac{1}{2} + \frac{1}{4} \dots + \frac{1}{2^p}\right) \\ &\leq w_{left}(x_0) + w_{left}(x_1) \end{aligned}$$

Let $t = buildBalanceTree(u_1 \dots u_n)$.

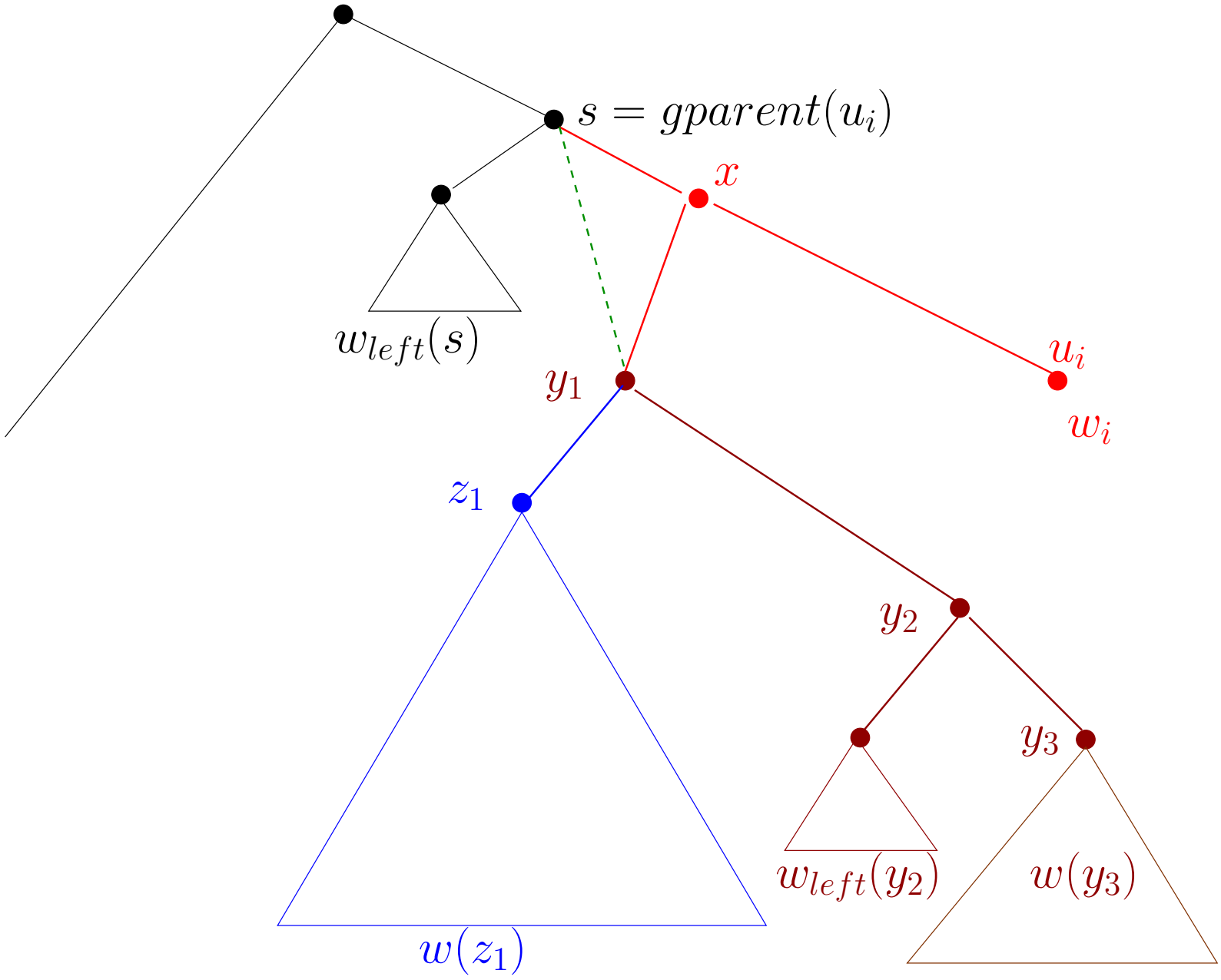
Lemma (4)

Let x be a node of t such that the length of the left branch of $t{\downarrow}x$ is at least 2,

Let $y_1 = left(x)$ and $z_1 = left(y_1)$.

One has $w(z_1) \leq w(x)/2$.

We have $w_i + (w(y_2) + w_{left}(y_1)) \leq w_{left}(s)$.



Proof of Lemma 4.

First case : if $parent(x)$ is not null, let $s = parent(x)$.
Let u_i be the left leaf of $t{\downarrow}x_1$,

In t_{i-1}, $s = gparent_{t_i}(u_i)$ has a weight-balance less than 1
$\Leftrightarrow \quad w_i + (w(y_2) + w_{left}(y_1)) \leq w_{left}(s)$.

- Suppose that $w_i + w(y_2) \leq w_{left}(y_1)$
  then y_2 is weight-sibling of u_i and u_i should not have been inserted as sibling of y_1.
- Suppose that $w_{left}(y_1) \leq w_i + w(y_2)$,
  since we have $w_{left}(y_1) + w_i + w(y_2) \leq w(x)$,
  adding these two inequalities, we get :
  $w(z_1) = w_{left}(y_1) \leq w(x)/2$.

Proof of Lemma 4.

Second case : if $parent(x) = null$,

then $gparent_{t_{i-1}}(y_2) = null$

Since u_i is inserted as sibling of y_1,
we deduce that y_2 is not weight-sibling of u_i :

$$\Rightarrow \quad w_{right}(y_1) = w_i + w(y_2) > w_{left}(y_1)$$

Recall that we have $w_i + w(y_2) + w_{left}(y_1) \leq w(x)$.
Adding the two inequalities, we get :

$$w(z_1) = w_{left}(y_1) \leq w(x)/2.$$

Experimental results

size of term	unbalanced height	balanced height
127	15	10
363	31	12
549	98	13
769	384	11
835	46	15
1407	115	15
2183	131	16
2595	121	16

Conclusion

- We define a notion of balance for weighted trees, comparing the depths of the leaves with the ratio of their weights to the total weight.
- By a simple inductive algorithm, we easily get a $(2, 2)$-balanced alphabetic weighted trees in time $\mathcal{O}(n \log n)$.
- We give an online linear algorithm to balance alphabetic weighted trees. The result is a $(3, 2)$-balanced weighted tree. There is no need of extra data structures. There is no need of sorting the weights.
- Future work : use the algorithm to balance binary terms in linear time.