# Minimum Spanning Tree Verification In Linear Time Complexity

Valerie King

University of Victoria, Canada

Prepared by Iddo Bentov, Technion, 2009

## Introduction

We present an algorithm ([Ki95] based on [Ko85] and [SV88]) that verifies in linear time whether a given spanning tree $T$ of a graph $G = (V, E)$ is a minimum spanning tree. In turn, this algorithm can be used as a black box for a randomized algorithm ([KKT95]) that finds a minimum spanning forest in linear time with an exponentially small failure probability.

### Computational model

We require the following two assumptions:

- The number of nodes in the tree can be represented in a single word, e.g. if *wordsize* is 64 bits then there are no more than $2^{64}$ nodes in $T$.
- Unit cost RAM model, i.e. each machine instruction that operates on one or two *wordsize* operands takes $\mathcal{O}(1)$ time.

## Overview of the Komlós algorithm

We first discuss how to verify whether a full branching tree (i.e. a rooted tree with a branching factor of at least 2 and all leaves on the same level) is a MST by using $\mathcal{O}(|E|)$ weight comparisons.

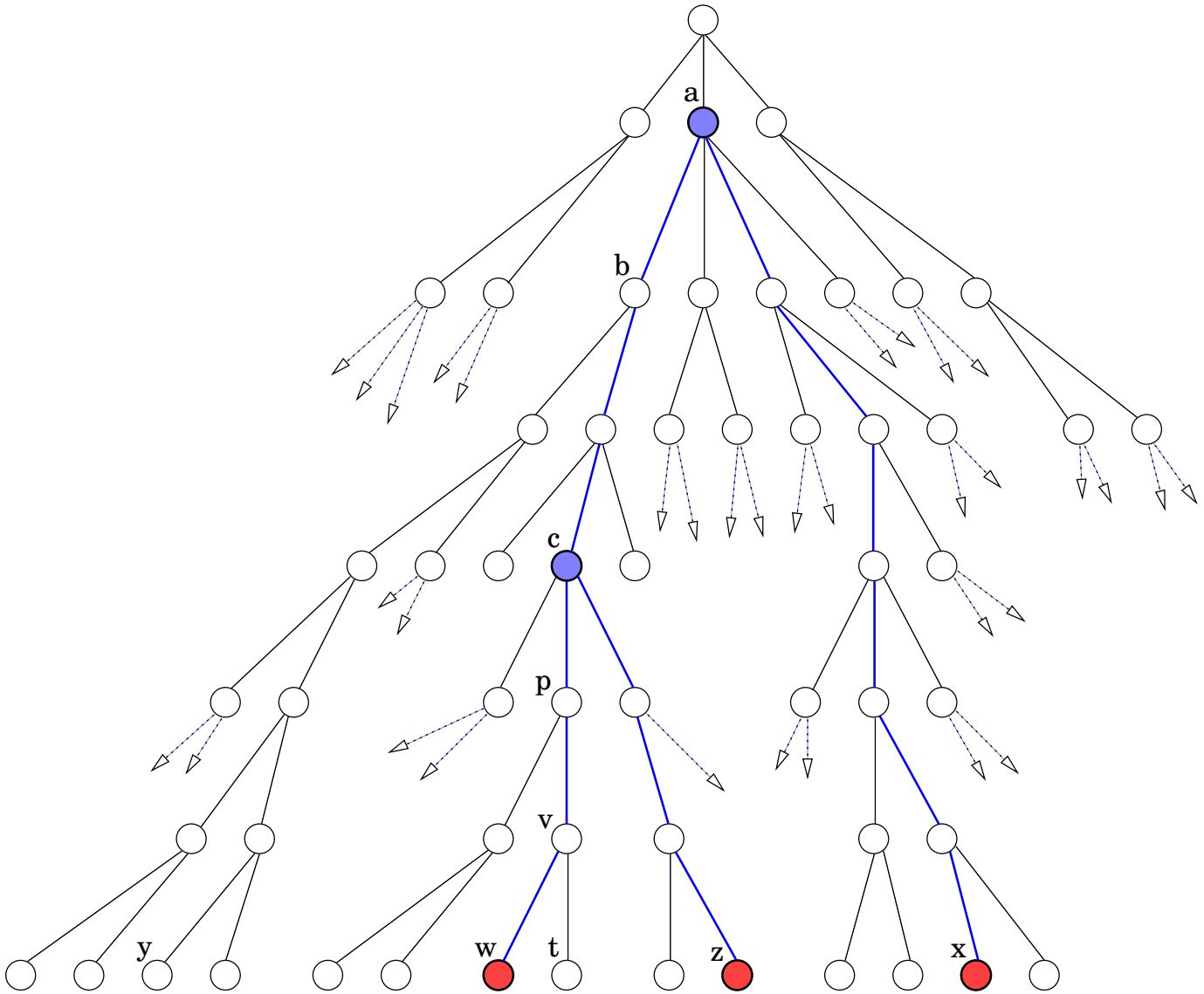
By the cycle property, it is enough to verify that each non-tree edge $e = (u, v)$ is at least as heavy as the heaviest edge in $treepath(u, v)$, where $treepath(u, v)$ is the set of edges that connect the nodes $u$ and $v$ in the tree.

The idea is that for every non-tree edge $(u, v)$ we look separately at the edges of the set $treepath(u, v)$ that are in the interval $[root, u]$, and the edges of the set $treepath(u, v)$ that are in the interval $[root, v]$, and find the heaviest edge in each of these two sets by descending down the tree and using binary search on the data that was already gathered in the previous tree level.

Let us illustrate this with the help of a simple example.

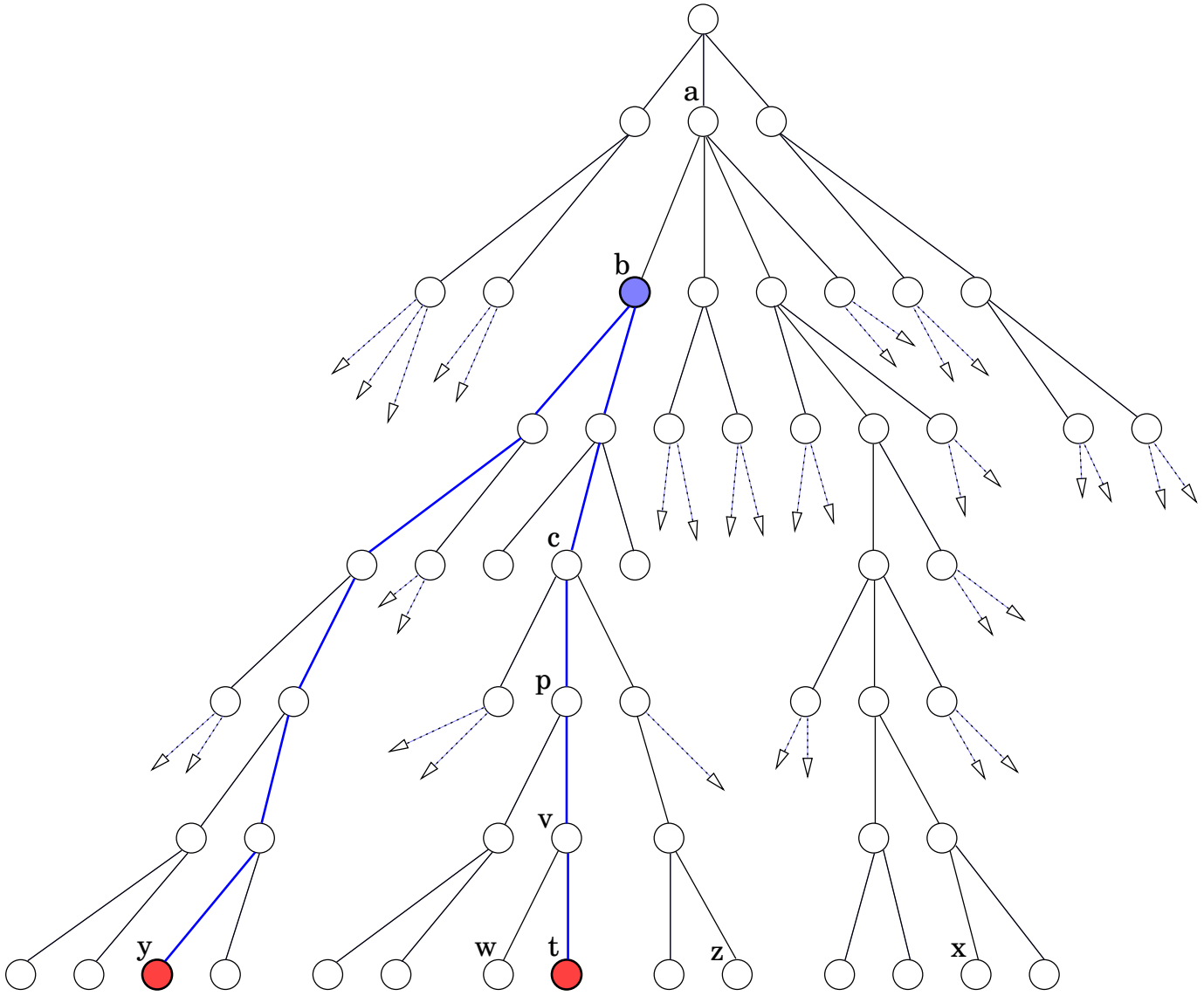
## *treepath*$(w, z)$ **and** *treepath*$(w, x)$

Suppose that $(w, z)$ and $(w, x)$ are non-tree edges.



## *treepath*$(t, y)$

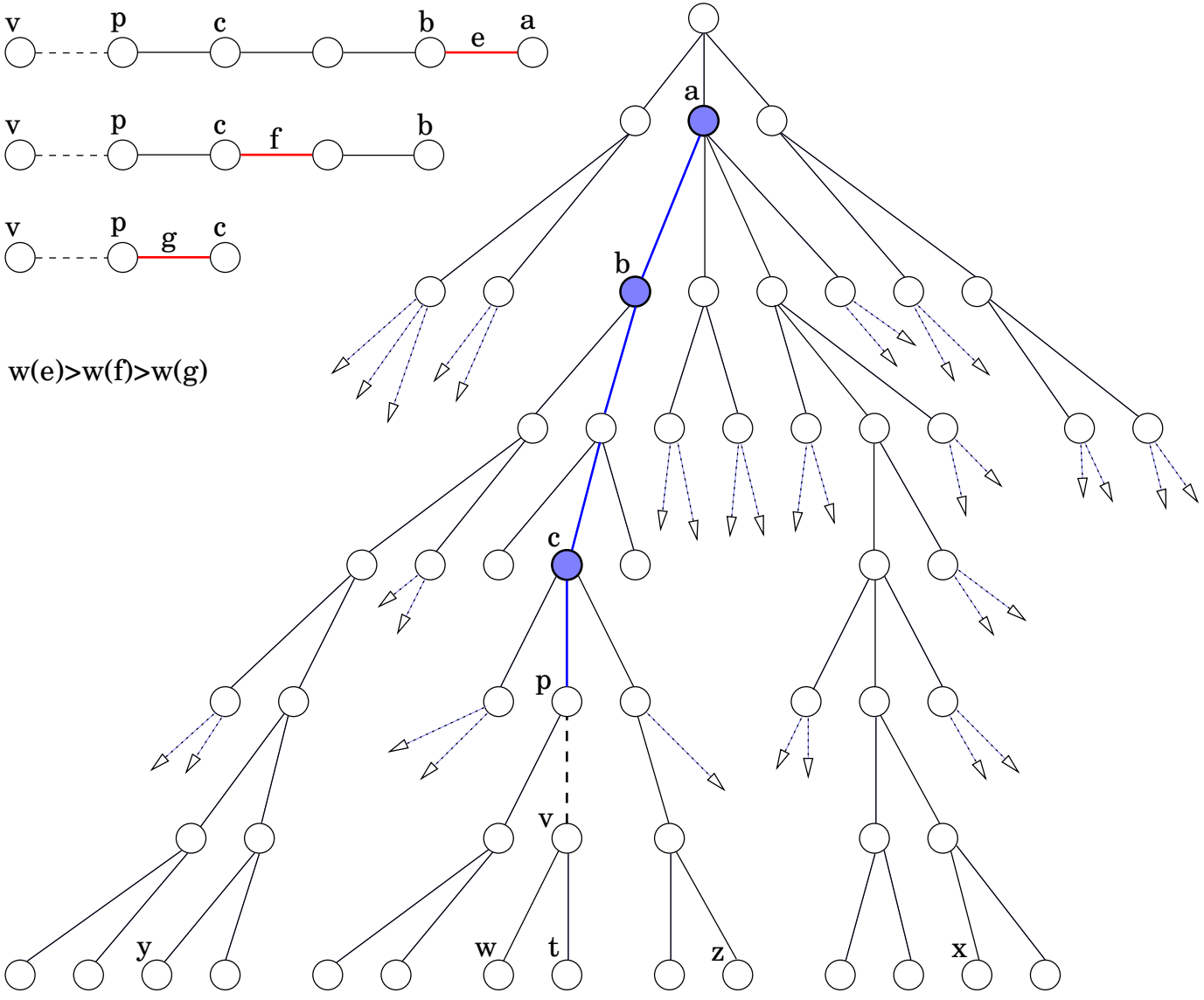
Suppose also that $(t, y)$ is the only other non-tree edge in $G$.



## The Komlós algorithm

Let us denote by $A(v)$ the set of tree-paths that contain the node $v$ and need to be queried, restricted to the interval $[root, v]$. In the example that we've just illustrated, $A(v)$ contains $treepath(v, a)$, $treepath(v, b)$ and $treepath(v, c)$.
Assume we already know the weight of the heaviest edge in each path of $A(p)$, where $p = parent(v)$. In our particular example, that means that we already know the heaviest weight in $treepath(p, a)$, $treepath(p, b)$ and $treepath(p, c)$. Since each path in $A(p)$ contains all the paths in $A(p)$ that are shorter than it, it means that the ordering of $A(p)$ according to the heaviest weight of each path is determined by the paths' length. Let $A(v|p)$ be the paths of $A(v)$ restricted to the interval $[root, p]$. Note that determining $A(v|p)$ doesn't involve weight comparisons, and note that in our example $A(v|p) = A(p)$. Since $A(v|p) \subseteq A(p)$, the weights order for $A(v|p)$ is also known. That means that we can use binary search to place $weight(v, p)$ among the weights of $A(v|p)$, and thereby determine the heaviest weights of $A(v)$. [illustration follows]

# The Komlós algorithm at node *v*



# Difficulties regarding the Komlós algorithm

## Computational overhead beyond the weight comparisons

Observe that in our example $A(w|v) \neq A(v) \neq A(t|v)$ because $A(w) = \{treepath(w, a), treepath(w, c)\}$, $A(t) = \{treepath(t, b)\}$. Or if e.g. there's another non-tree edge $(q, r)$ where $q \notin subtree(a)$ and $r \in subtree(p) \setminus subtree(v)$ then in that case $A(v|p) \neq A(p)$. A major obstacle to transforming the Komlós algorithm into an algorithm that runs in linear time is generating $A(child(p)|p)$ from $A(p)$ with linear overhead w.r.t. the weight comparisons. In other words, how to decide in $\mathcal{O}(|E|)$ time which comparisons to make.

## Historical note

In the original paper, Komlós also provides an algorithm for a general (not full branching) tree that performs a linear number of weight comparisons, but that result is rather involved. We will rely instead on a much simplified approach, due to Valerie King, that converts a general tree to a full branching tree in linear time.

## Analysis of the Komlós algorithm

If we start with $A(root) = \emptyset$ and descend down the tree, the number of weight comparisons needed for the binary search at each node $u$ is bounded by $\lceil \log_2(1 + |A(u)|) \rceil$.

Let $n = |V|$, and let $m$ be the number of tree-paths that need to be queried in accordance with the cycle property, i.e. $m = |E| - n + 1$. Let $D_i$ be the set of nodes at level $i$, and let $d_i = |D_i|$. Now,

$$\sum_{u \in D_i} \lceil \log(1+|A(u)|) \rceil < \sum_{u \in D_i} (1+\log(1+|A(u)|)) \overbrace{\leq}^{(*)} d_i + d_i \log \frac{d_i + 2m}{d_i}$$

$(*)$ By Jensen's inequality for the concave function $\log(\cdot)$,

$$\sum_{u \in D_i} \frac{1}{d_i} \log(1+|A(u)|) \leq \log\left(\sum_{u \in D_i} \frac{1}{d_i}(1+|A(u)|)\right) = \log \frac{d_i + \sum_{u \in D_i} |A(u)|}{d_i}$$

## Analysis of the Komlós algorithm (cont.)

Therefore the number of weight comparisons is bounded by

$$\sum_{i\geq 0} d_i + d_i \log \frac{d_i + 2m}{d_i} \leq n + \sum_i d_i \log\left(\frac{\mathbf{n+2m}}{d_i} \cdot \frac{n}{\mathbf{n}}\right) =$$

$$n + \sum_i d_i \log \frac{n+2m}{n} + \sum_i d_i \log \frac{n}{d_i} \overbrace{\leq}^{(**)} n + n \log \frac{n+2m}{n} + 2n$$

$(**)$ Because $T$ is a full branching tree, it means that $\sum_{k<i} d_k < d_i$

$$\Longrightarrow \sum_{i\geq 0} \frac{d_i}{n} \log_2 \frac{n}{d_i} \overbrace{<}^{\text{entropy}} 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 2$$

Now, since $\log \frac{n+2m}{n} \leq \frac{n+2m}{n}$, we obtained $3n + n \log \frac{n+2m}{n} \leq 4n + 2m$. Therefore, by performing 2 additional comparisons for each tree-path query that is needed in order to verify the cycle property, it holds that the total number of weight comparisons is $\mathcal{O}(n+m) + 2m = \mathcal{O}(n+m)$ □

## Transforming the Komlós algorithm

### What lies ahead...

Our remaining tasks are

## Transforming the Komlós algorithm

### What lies ahead...

Our remaining tasks are

- Presenting the Schieber-Vishkin algorithm for answering *LCA* queries in constant time, by using pre-processed tables that are prepared in linear time. **We will focus on certain modifications to the original algorithm that are suitable for our needs.**

## Transforming the Komlós algorithm

### What lies ahead...

Our remaining tasks are

- Presenting the Schieber-Vishkin algorithm for answering *LCA* queries in constant time, by using pre-processed tables that are prepared in linear time. **We will focus on certain modifications to the original algorithm that are suitable for our needs.**
- Adapting the Komlós algorithm to work in the case that the tree isn't a full branching tree.

## Transforming the Komlós algorithm

### What lies ahead...

Our remaining tasks are

- Presenting the Schieber-Vishkin algorithm for answering *LCA* queries in constant time, by using pre-processed tables that are prepared in linear time. **We will focus on certain modifications to the original algorithm that are suitable for our needs.**
- Adapting the Komlós algorithm to work in the case that the tree isn't a full branching tree.
- Combining these algorithms with a data structure that will allow us to achieve a linear overhead, meaning that while executing the Komlós algorithm, **at each node $u$ we will perform $\mathcal{O}(\log(1 + |A(u)|))$ machine instructions.**

## The preprocessing stage of the Schieber-Vishkin algorithm

We start by precomputing a few lookup tables in $\mathcal{O}(n)$ time and space. We generate $PREORDER(v)$, $SIZE(v)$ and $LEVEL(v)$ via preorder traversal of $T$. $SIZE(v)$ is the number of nodes in $subtree(v)$, the sub-tree rooted at $v$. $LEVEL(v)$ is the distance of $v$ from the root, so e.g. $LEVEL(root) = 0$. We then replace the $SIZE$ table with the $INLABEL$ table, where $INLABEL(v)$ is the $PREORDER$ number with maximum rightmost '0' bits of a node in $subtree(v)$, or in the interval $[a, b]$ where $a = PREORDER(v)$ and $b = PREORDER(v) + SIZE(v) - 1$. This is done by $i \leftarrow BSR((a-1) \text{ xor } b), INLABEL(v) \leftarrow (b >> i) << i$, where $BSR$ is the machine instruction "bit scan reverse" which is equivalent to $\lfloor \log_2 \rfloor$ and gives the index of the leftmost bit set to '1' (starting from the rightmost bit whose index is 0), which in our case is the index for the largest power of 2 between $a$ and $b$. We also generate the $ASCENDANT$ table, where the $i^{th}$ bit of $ASCENDANT(v)$ is set to '1' iff $i$ is the index of the rightmost '1' of the $INLABEL$ of an ancestor of $v$. We do that by traversing the tree starting with $ASCENDANT(root) \leftarrow 1 << BSR(n)$, and to every other node $v$ we set $ASCENDANT(v)$ to be $ASCENDANT(parent(v))|(INLABEL(v)\&(-INLABEL(v)))$.

[$(x\&(-x))$ sets only the rightmost '1' of $x$, by the properties of two's complement]

## Schieber-Vishkin preprocessing and the tag property

As we will see, the single word $ASCENDANT(v)$ can be used to construct from $INLABEL(v)$ the $INLABEL$ numbers of all the ancestors of $v$.

We define the tag of a tree edge $e = (u, v)$ to be $\mathcal{O}(\log \log n)$ bit string that consists of $< LEVEL(v), BSR(INLABEL(v)\&(-INLABEL(v))) >$, i.e. the level of $v$ and the index of the rightmost '1' of $INLABEL(v)$, where $v$ is the node farther from the root. For the MST verification algorithm, we depend on the following *tag property*: given the tag of any tree edge $e$ and $INLABEL(w)$, where $w$ is some node on the path from $e$ to some leaf, $e$ (or the weight of $e$) can be located in constant time.

To satisfy the *tag property*, we build the $HEAD$ table a little differently than in the original Schieber-Vishkin algorithm. First we build a temporary table that contains for each internal node $p$ the child (if it exists) $v$ of $p$ for which $INLABEL(p) = INLABEL(v)$. This is done by traversing the tree and updating the table entry of the node $p$ only when we descend from $p$ to a node $v$ that has the same $INLABEL$ number.

Note: the *PREORDER*, *LEVEL*, *INLABEL* and temporary tables can be built simultaneously in a single tree traversal, and the *ASCENDANT*, *ALL* and *HEAD* tables can then be built simultaneously in another traversal.

## Schieber-Vishkin preprocessing and the tag property (cont.)

We now generate the tables $ALL$ and $HEAD$ as follows. We traverse the tree in the order specified by the temporary table, i.e. we always descend first to the child with the same $INLABEL$ number (and afterwards to the other children in some arbitrary order). Starting with $i \leftarrow 0$, whenever we descend to any node $v$, we update $ALL(i)$ with $v$, and whenever we descend to a node $v$ whose $INLABEL$ is different than the $INLABEL$ of its parent (meaning that $v$ is the head of the $INLABEL(v)$ path), we update $HEAD(INLABEL(v))$ with $i$, and after we visit each node we increment $i \leftarrow i + 1$. Thus the table $ALL$ contains exactly $n$ elements that correspond to the tree nodes, but grouped according to the $INLABEL$ paths so that the nodes of each path appear in consecutive order in $ALL$ (the order among the paths is arbitrary). And the table $HEAD$ stores the placement of the head of each $INLABEL$ path in $ALL$.

We can now see that the *tag property* holds: given the tag $< d, k >$ of the edge $e = (u, v)$ and $INLABEL(w)$, we obtain $L_v = INLABEL(v)$ by $L_v \leftarrow ((INLABEL(w) >> k) \mid 1) << k$ and then $j \leftarrow HEAD(L_v)$ and then $v \leftarrow ALL(k - LEVEL(ALL(j)) + j)$, thus finding $e$ as the unique edge from $v$ to its parent.

# The Schieber-Vishkin algorithm

**Schieber-Vishkin algorithm for finding $z = LCA(x, y), d_z = LEVEL(z) \triangleq \tilde{LCA}(x, y)$**

1. if $INLABEL(x) = INLABEL(y)$ then return $d_z \leftarrow min(LEVEL(x), LEVEL(y))$
2. $i \leftarrow \begin{cases} BSR(INLABEL(x)\ \&\ (-INLABEL(y))) & \text{if } INLABEL(x) \geq INLABEL(y) \\ BSR(INLABEL(y)\ \&\ (-INLABEL(x))) & \text{if } INLABEL(x) < INLABEL(y) \end{cases}$
3. $i' \leftarrow ASCENDANT(x)\ \&\ ASCENDANT(y)\ \&\ (-(1 << i))$
4. $i \leftarrow BSR(i'\ \&\ (-i'))$
5. $L_z \leftarrow ((INLABEL(x) >> i) \mid 1) << i$
6. if $INLABEL(x) = L_z$ then $d_x \leftarrow LEVEL(x)$ else do steps 7 and 8
7. $k = BSR(ASCENDANT(x)\ \&\ ((1 << i) - 1))$
8. $x' \leftarrow ALL(HEAD(((INLABEL(x) >> k) \mid 1) << k)),\ d_x \leftarrow LEVEL(x') - 1$
9. if $INLABEL(y) = L_z$ then $d_y \leftarrow LEVEL(y)$ else do steps 10 and 11
10. $k = BSR(ASCENDANT(y)\ \&\ ((1 << i) - 1))$
11. $y' \leftarrow ALL(HEAD(((INLABEL(y) >> k) \mid 1) << k)),\ d_y \leftarrow LEVEL(y') - 1$
12. return $d_z \leftarrow min(d_x, d_y)$

For our needs, computing $\tilde{LCA}(x, y)$ is sufficient, but it is just as easy to compute $z$ itself ($z$ is $parent(x')$ or $parent(y')$ or $x$ or $y$).

# Outline of the Schieber-Vishkin algorithm

Step 1 is appropriate since the *INLABEL* numbers partition the tree into paths.

Step 2 finds the index of the rightmost '1' of $b = LCA(INLABEL(x), INLABEL(y))$ in the sideway tree (smallest full binary tree with $n$ nodes, labeled by inorder traversal).

Step 3 sets the bit of the level of $INLABEL(z)$ in the sideway tree as the lowest bit turned on, by finding the nearest level above $b$ that is common to the levels recorded in $ASCENDANT(x)$ and $ASCENDANT(y)$. Step 4 gets the index of this bit. Correctness follows from the *descendance-preservation property*, i.e. for any node $v$ in $T$ the inlabel mapping sends descendants of $v$ to descendants of $INLABEL(v)$ in the sideway tree (so there are at most $\log n$ distinct *INLABEL*s among the ancestors of $v$).

This property follows from the observation that $s_1 = \overbrace{\square\square\ldots\square}^{r}1\overbrace{00\ldots0}^{t}$ is an ancestor of $s_2$ (which includes the case $s_1 = s_2$) in the sideway tree iff $s_2$ has the same $r$ leftmost bits and no more that $t$ rightmost '0' bits. Therefore, if $w \in subtree(v)$ in the tree $T$, since the $r$ leftmost bits of $INLABEL(v)$ are shared by the *INLABEL* numbers of all the nodes in $subtree(v)$, and none of these nodes have *INLABEL* with more than $t$ rightmost '0' bits, the *descendance-preservation property* holds.

Step 5 constructs $L_z = INLABEL(z)$ from the index (that was obtained in step 4) and the remaining leftmost bits of the *INLABEL* of $x$ (or $y$).

# Outline of the Schieber-Vishkin algorithm (cont.)

Steps 6,7,8 compute $\hat{x}$, the lowest ancestor of $x$ in the path defined by $INLABEL(z)$.

Step 7 gets the rightmost '1' of $x'$, the child of $\hat{x}$ that is an ancestor of $x$, by finding the leftmost '1' recorded in $ASCENDANT(x)$ which is below the level of $INLABEL(z)$.

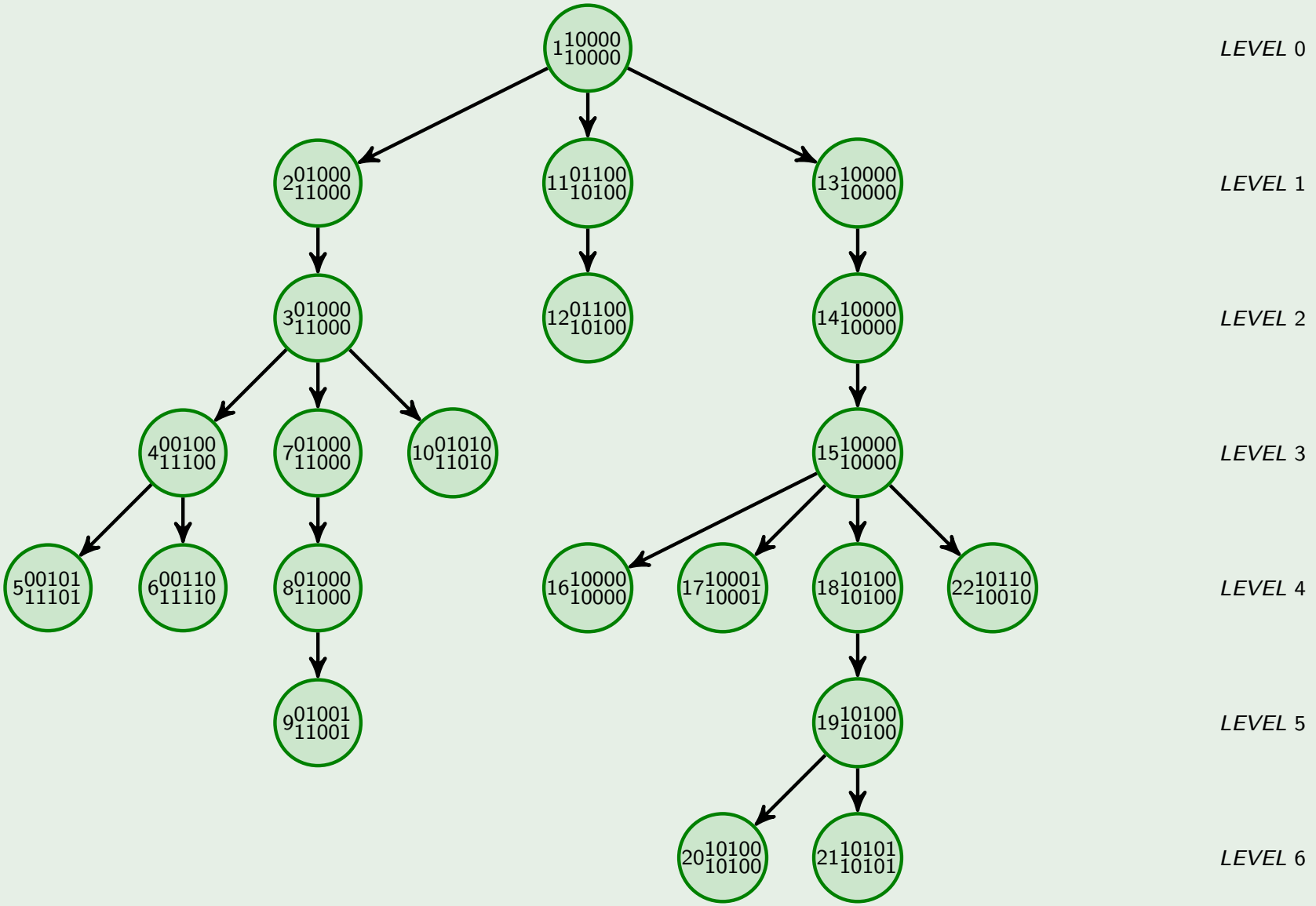
Step 8 constructs the *INLABEL* of this child by attaching the remaining leftmost bits of $x$, and then obtains the child itself via the *HEAD* table, which is merited since this child is the first node in its *INLABEL* path (as its parent, $\hat{x}$, has a different *INLABEL*).

Steps 9,10,11 similarly compute $\hat{y}$, the lowest ancestor of $y$ in the $INLABEL(z)$ path.

Step 12 completes the algorithm, because either $z = \hat{x}$ or $z = \hat{y}$. This follows from the observation that $INLABEL(z) = INLABEL(\hat{x}) = INLABEL(\hat{y})$, and therefore either $\hat{x}$ belongs to $subtree(\hat{y})$ or $\hat{y}$ belongs to $subtree(\hat{x})$ (otherwise we get a contradiction to fact that the *INLABEL* numbers partition the tree into paths). Thus both $x$ and $y$ belong either to $subtree(\hat{x})$ or to $subtree(\hat{y})$, and since $z$ is the *lowest* common ancestor of $x$ and $y$ (and both $\hat{x}$ and $\hat{y}$ aren't higher than $z$), the correctness follows.

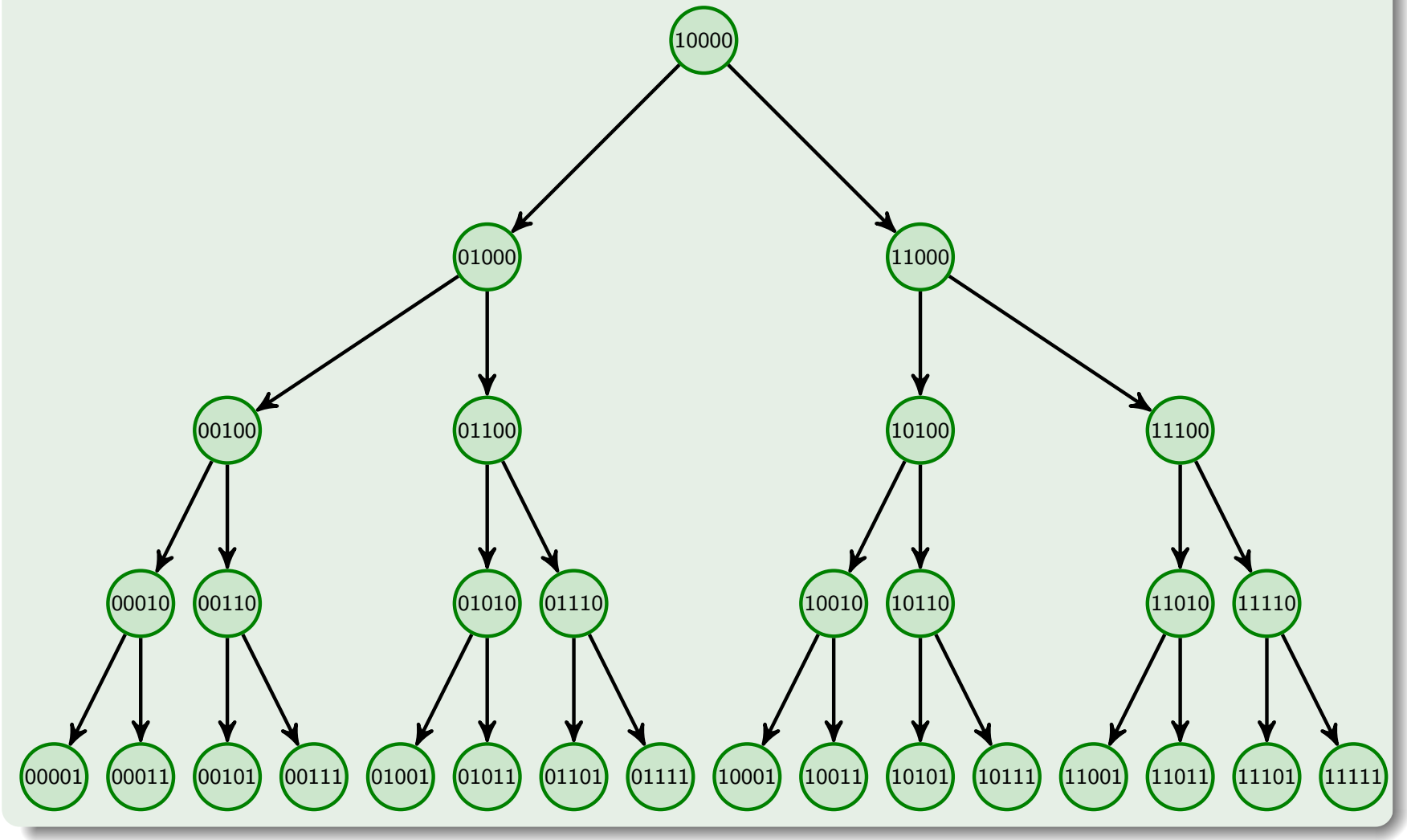
# An illustration of the Schieber-Vishkin algorithm

Tree example: *PREORDER* on left, *INLABEL* on top, *ASCENDANT* on bottom



# An illustration of the Schieber-Vishkin algorithm (cont.)

## The sideway tree



## Converting the tree to a full branching tree

We construct a full branching tree $B$ by applying the Borůvka MST algorithm to the given spanning tree $T$. Namely, we start with $n$ blue trees that correspond to the nodes of $T$, and repeat the following until only one blue tree (i.e. $T$) remains: for each blue tree, select the minimum weight edge incident to it and color this edge blue.
Each repetition will be referred to as a phase. We update $B = (W, F)$ after each phase so that all the blue trees that are created correspond 1-1 to the nodes $W$. At start we create the leaf $f(v) \in W$ for every $v \in V$. Let $H$ be some set of blue trees that are joined into one blue tree $t$, then we add the new node $f(t)$ to $W$ and $\{(f(h), f(t)) | h \in H\}$ to $F$. The weight of $(f(h), f(t))$ is the weight of the edge that $h$ selects to connect to $t$ with.

Thus $B$ is a full branching tree that is built in $\mathcal{O}(n)$ time, because the number of blue trees drops by a factor of at least two after each phase, and the running time of each phase is proportional to the number of uncolored edges (each uncolored edge is inspected twice, i.e. by the two blue trees incident to it). Note that the number of uncolored edges is one less than the number of blue trees at that phase (because $T$ is a tree).

## Converting the tree to a full branching tree (cont.)

We now prove that for any two nodes $x, y$ in $T$, the weight of the heaviest edge in $treepath(x, y)$ equals the weight of the heaviest edge in $treepath(f(x), f(y))$. In other words, verifying the cycle property for $T$ can be done by using $B$ instead. So we can dispose of $T$ and use just $B$.

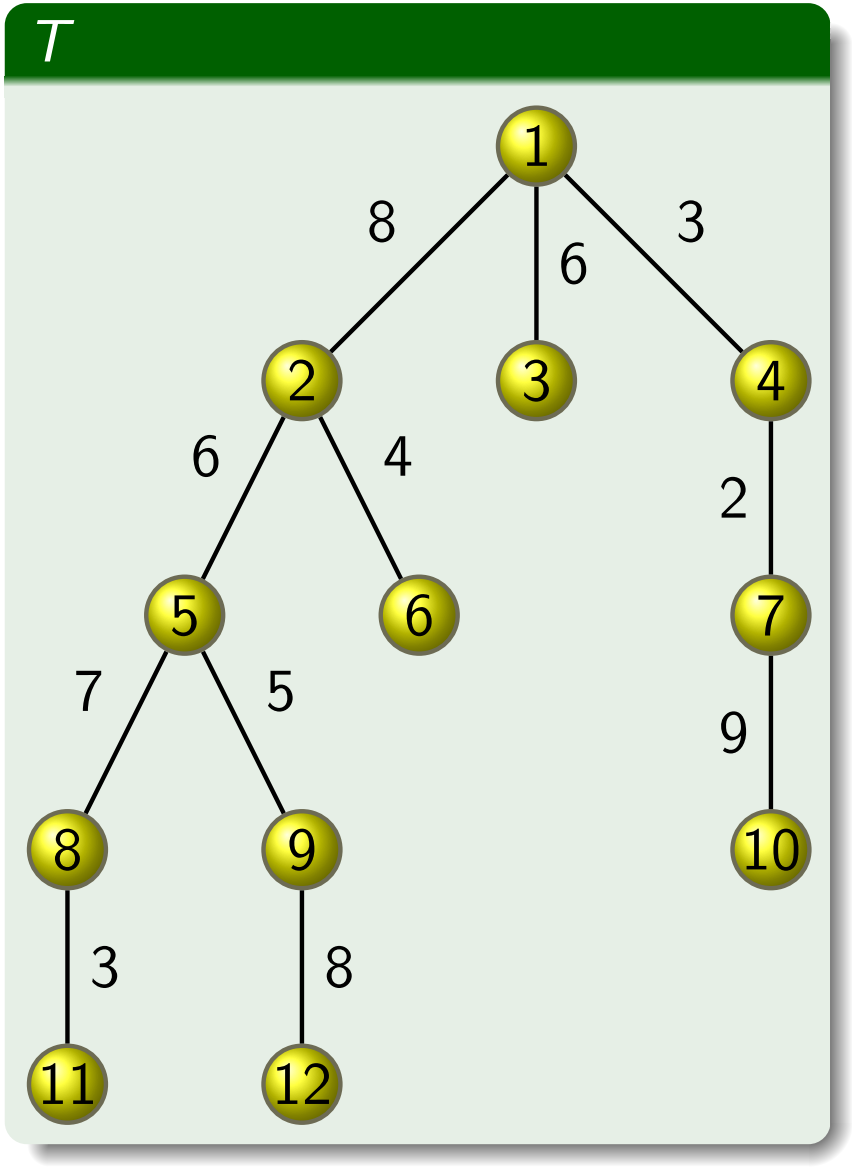
$\boxed{\Longleftarrow}$ Let $e \in treepath(f(x), f(y))$, then there exists $e' \in treepath(x, y)$ such that $weight(e') \geq weight(e)$.

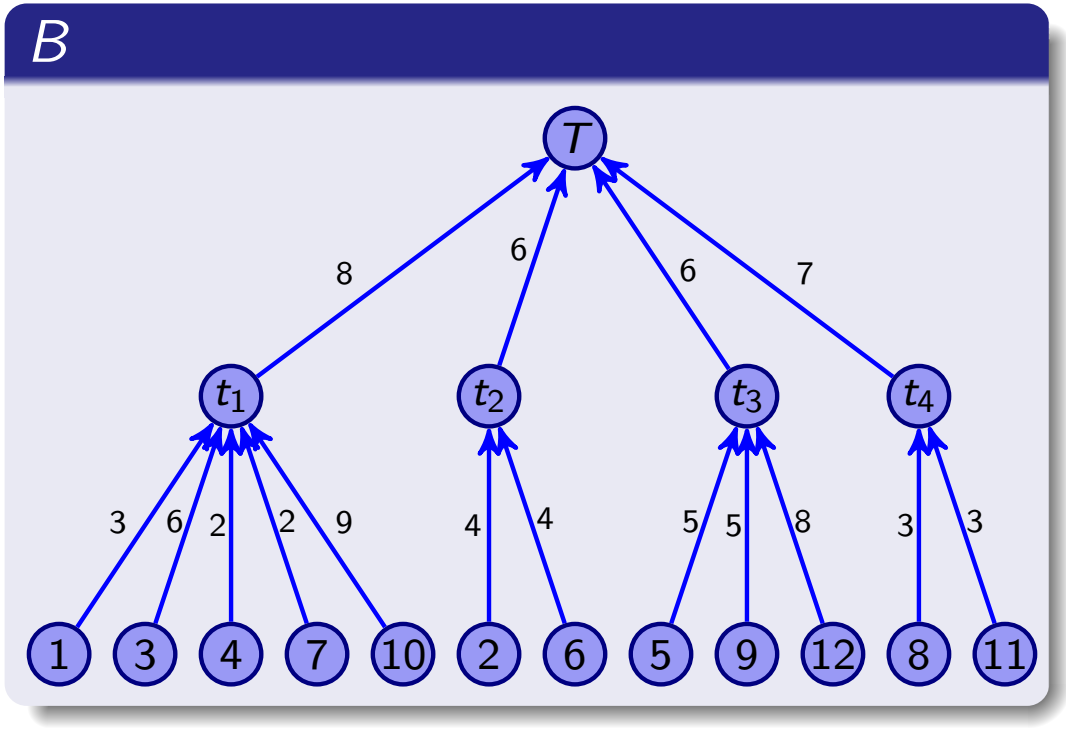
**Proof:** Suppose $e = (f(t), f(z))$ where $f(t)$ is the node farther from the root, then $t$ contains either $x$ or $y$, but not both. Let $e'$ be the edge in $treepath(x, y)$ with exactly one endpoint in $t$. Since $t$ had the option to connect to the rest of the tree via $e'$, it means that $weight(e') \geq weight(e)$ due to the minimality condition of Borůvka.

$\boxed{\Longrightarrow}$ Let $e$ be a heaviest edge in $treepath(x, y)$, then there is an edge with the same weight in $treepath(f(x), f(y))$.

**Proof:** Note that $e$ must be selected by some blue tree. If $e$ is selected by a blue tree that contains $x$ or $y$ then we are done. Assume the contrary, i.e. that $e$ is selected by some blue tree $t$ that contains neither $x$ nor $y$. Thus at the phase in which $e$ is selected, $t$ had the option to choose $e$ and extend towards (w.l.o.g.) $x$, and also the option to choose some other $e' \in treepath(x, y)$ and extend towards $y$ (because choosing $e$ doesn't lead towards $y$, and $y$ isn't in $t$). If $weight(e') < weight(e)$ then the fact that $t$ chose $e$ contradicts the minimality condition of Borůvka. And if $weight(e') = weight(e)$ then $e'$ is also a heaviest edge in $treepath(x, y)$, and unlike $e$ it is still uncolored (since $t$ chose $e$), so we repeat the same argument with $e'$.

# An illustration of the full branching tree $B$ that is constructed from $T$



## Lookup tables for the MST verification algorithm

We prepare a $LCA$ table, where for each tree node $v$ the $i^{th}$ bit of $LCA(v)$ is '1' iff there is a path in $A(v)$ whose upper endpoint is at distance $i$ from the root. This is done by using Schieber-Vishkin to compute $L\tilde{C}A(u, w)$ for each non-tree edge $(u, w)$ out of the $m$ non-tree edges, in $\mathcal{O}(1) \cdot m = \mathcal{O}(m)$ time. We then form $LCA(x)$ for each leaf $x$ by computing $LCA(x)|(1 << L\tilde{C}A(w, x))$ for every edge $(w, x)$ that is incident to $x$, in $\mathcal{O}(n + m)$ time. And for every internal node $v$ at distance $i$ from the root we compute $LCA(v)$ by ORing the $LCA$ values of its children and ANDing the result with $((1 << i) - 1)$ in order to set to '0' all the bits at distance $i$ or higher, in $\mathcal{O}(n)$ time. We store the $L\tilde{C}A(u, w)$ values for later use.

Let *subword* be *tagsize* bits, where *tagsize* is the size of an edge tag, which is $\mathcal{O}(\log \log n)$. Let *subwnum* be $\lfloor \frac{wordsize}{tagsize} \rfloor$. We require a few unconventional functions that we precompute in $\mathcal{O}(n)$ time and store in lookup tables, in order to allow access in $\mathcal{O}(1)$ time.

# Lookup tables for the MST verification algorithm

## Unconventional functions that we store in lookup tables

The input for the following functions is $k$ bits, and the output is *wordsize* bits. Let us assume $wordsize = 16$ and $tagsize = 5$ in our examples.
Let $subw1$ be the constant word in which every subword equals 1, so in our case $subw1 = 0{,}00001{,}00001{,}00001$.

**hweight$_k$** returns the number of bits set to '1', i.e. the Hamming weight of the input (also known as the population count).
Example: $hweight_{16}(1111{,}1111{,}1111{,}1001) = 0000{,}0000{,}0000{,}1110$
It can be easily computed via a lookup table in the same way that we show for the next functions, though many architectures already have a *POPCNT* machine instruction (same goes for *BSR* that is used in Schieber-Vishkin).

**index$_k$** takes a bit string that contains no more than *subwnum* '1' bits, and outputs a list of subwords containing the indices of the '1' bits.
Example: $index_{16}(1000{,}0000{,}0000{,}1000) = 0{,}00000{,}01111{,}00011$
Let us demonstrate how to build the $index_k$ lookup table. Suppose we have already built $index_{k/2}$, so now we go over all the $2^k$ strings and compute the output for each string $x$ in $\mathcal{O}(1)$ time, as follows.

# Lookup tables for the MST verification algorithm

## Unconventional functions that we store in lookup tables (cont.)

Let $x_1 \leftarrow (x\&((1 << (k/2)) - 1))$, $x_2 \leftarrow (x >> (k/2))$ and let $z_1 \leftarrow index_{k/2}(x_1)$, $z_2 \leftarrow index_{k/2}(x_2)$, i.e. we look up the output for the first half and second half of $x$. Next we add $\frac{k}{2}$ to each subword in the second half, by $z_2 \leftarrow z_2 + subw1 * (k/2)$, and finally we concatenate the first $hweight(x_1)$ subwords (i.e. $tagsize \cdot hweight(x_1)$ bits) of $z_1$ with the first $hweight(x_2)$ subwords of $z_2$, by shifting and masking, and thereby form $index_k(x)$.
Thus for any $k$ the total time needed to build the $index_k$ table is bounded by $\mathcal{O}(2^k + 2^{k/2} + 2^{k/4} + \ldots) = \mathcal{O}(2^k)$. For our needs $k = \lceil \log n \rceil \leq wordsize$, hence the table is built in linear time. Note that illegal inputs (with more than *subwnum* '1' bits) don't come up in the MSTV algorithm, so it's of no concern.

**$\mathbf{select_k}$** takes as input $X \cdot Y$ where $X$ and $Y$ are two bit strings of $k$ bits each, and outputs a list of bits of $Y$ which are "selected" by $X$, i.e. if $< k_1, k_2, \ldots >$ is the ordered list of indices of the '1' bits of $X$, then the output represents the ordered list $< j_{k_1}, j_{k_2}, \ldots >$, where $j_{k_i}$ is the value of the $k_i^{th}$ rightmost bit of $Y$.

Example: $select_8(11110101, 01101110) = select_8(< 1, 3, 5, 6, 7, 8 >, 01110110) =< 0, 1, 0, 1, 1, 0 >= 00011010$
We build the $select_k$ lookup table similarly to the way we built the $index_k$ table. Let us use this example to demonstrate. Suppose we already have $select_4$, i.e. we know that $select_4(0101, 1110) = 0010$ and that $select_4(1111, 0110) = 0110$.

# Lookup tables for the MST verification algorithm

## Unconventional functions that we store in lookup tables (cont.)

Let $x = \overbrace{1111}^{b_2}\overbrace{0101}^{a_2},\overbrace{0110}^{b_1}\overbrace{1110}^{a_1}$. Our goal is to compute $select_8(x)$ in $\mathcal{O}(1)$ time. We form $a_1 \leftarrow x\&15, b_1 \leftarrow (x >> 4)\&15, a_2 \leftarrow (x >> 8)\&15, b_2 \leftarrow (x >> 12)\&15$, then look up $a \leftarrow select_4(a_1|(a_2 << 4)) = 0010, b \leftarrow select_4(b_1|(b_2 << 4)) = 0110$, then assign $w \leftarrow hweight(a_2) = hweight(0101) = 2$ and form the $select_8(x)$ output by $(b << w)|a = (0110 << 2)|0010 = 00011010$.

**selectS$_k$** takes as input $X \cdot Y$ where $X$ and $Y$ are two bit strings of $k$ bits each, with $X$ having at most *subwnum* '1' bits, and outputs the list of subwords of $Y$ that are "selected" by $X$, i.e. if $< k_1, k_2, \ldots >$ are the indices of '1' bits of $X$, then the output represents $< j_{k_1}, j_{k_2}, \ldots >$, where $j_{k_i}$ is the $k_i^{th}$ subword of $Y$.

Example: $selectS_{16}(0000000000000101, 0110111111100100) = 0,00000,11011,00100$

We build $selectS_k$ similarly to $select_k$, e.g. to compute $selectS_{32}(x)$ where half of $x$ consists of this example, we shift by $tagsize \cdot hweight(101) = 5 \cdot 2 = 10$ and mask the selected subwords from the other half of $x$ after we look up the value.

**Note that unlike $index_{wordsize}$, building the tables for $select_{wordsize}$ and $selectS_{wordsize}$ takes superlinear time. The input size for $select_{wordsize}$ is $2\lceil \log n \rceil$, so going over all inputs gives $\mathcal{O}(n^2)$. However, by building tables for $select_{wordsize/2}$ and $selectS_{wordsize/2}$ we can look up any needed value in $\mathcal{O}(1)$ time, as demonstrated.**

## Overview and data structure of the MST verification algorithm

As with the Komlós algorithm, we descend from the root and determine $A(v)$ for every node $v$. If $|A(v)| > \mathit{subwnum}$ then we define $v$ as big, otherwise as small. While traversing the tree we maintain a pointer to nearest big ancestor of the current node.

- If $v$ is big we generate $A(v|p)$ in $\mathcal{O}(\log\log n)$ time and store $A(v)$ in an array of $\lceil \frac{|A(v)|}{\mathit{subwnum}} \rceil = \mathcal{O}(\log\log n)$ words that is denoted by $\mathit{bigList}(v)$. This size bound holds because each edge tag is stored in a single subword, and $|A(v)| \leq \log n$. Observe that the overhead w.r.t. the weight comparisons is linear, because $|A(v)| \geq \frac{\mathit{wordsize}}{\mathit{tagsize}} = \Omega(\frac{\log n}{\log\log n})$ and the number of weight comparisons is $\log |A(v)| \geq \log \Omega(\frac{\log n}{\log\log n}) \geq \Omega(\log\log n - \log\log\log n) \geq \Omega(\log\log n - \frac{1}{2}\log\log n) \geq \Omega(\log\log n)$
- In case $v$ is small we generate $A(v|p)$ in $\mathcal{O}(1)$ time and store $A(v)$ in $\mathit{smallList}(v)$, which is a single word that contains one subword per tree-path of $A(v)$. The overhead is constant.

# The MST verification algorithm - generating $smallList(v)$

Let $v$ be the current node reached while traversing the tree, let $p = parent(v)$, and let $z$ be the nearest big ancestor of $v$.

- **Step 1:** compute $size_{A(v)} \leftarrow hweight(LCA(v))$ and determine whether $v$ is small or big. [if $size_{A(v)} = 0$ then $smallList(v) \leftarrow 0$ and descend from $v$]
- **Step 2.1.1:** if $v$ is small and $p$ is small, create $smallList(v|p)$ from $smallList(p)$ by:
  - $L \leftarrow select(LCA(p), LCA(v))$
  - $smallList(v|p) \leftarrow selectS(L, smallList(p))$

illustration, $w(a) \geq w(b) \geq \ldots$

```
  A(p)=(    f e d c   b a  )
LCA(p)=(0 0 1 1 1 1 0 1 1 0)
LCA(v)=(0 1 1 1 0 1 0 1 0 0)
select=(    1 1 0 1   1 0  )
A(v|p)=(    f e   c   b    )
```

The *select* operation gives a bit list of $A(p)$ elements that belong to $A(v|p)$. Here we can directly copy all of the selected subwords from $smallList(p)$ to $smallList(v|p)$ with a single *selectS* operation.

**Example:** $LCA(p) = 0000000000010110$, $LCA(v) = 0000000000110100$, $L =< 0, 1, 1 >= 0000000000000110$
$smallList(p) =< 10001, 10101, 11111 >= 0111111010110001$, $selectS(< 0, 1, 1 >, 0111111010110001) = 0{,}00000{,}11111{,}10101$

## The MST verification algorithm - generating $smallList(v)$

- **Step 2.1.2:** if $v$ is small and $p$ is big, create $smallList(v|p)$ from $LCA(v)$ and $LCA(p)$, without accessing $bigList(p)$, by:
  - $smallList(v|p) \leftarrow index(select(LCA(p), LCA(v)))$

illustration, $w(a) \geq w(b) \geq \ldots$

```
  A(p)=(    e d c   b a  )
LCA(p)=(0 0 1 1 1 0 1 1 0)
LCA(v)=(0 0 1 1 0 0 1 0 0)
select=(    1 1 0   1 0  )
A(v|p)=(    5 4     2    )
```

Here we don't copy the selected subwords from $bigList(p)$, but instead just store the $bigList(p)$ indices of these subwords. The reason is that copying each of the subwords would take $\Omega(|A(v|p)|)$ operations (after the single *index* operation, an extra $\mathcal{O}(1)$ arithmetic and mask instructions per element of $A(v|p)$ would be needed), and our objective is to operate on $v$ in $\mathcal{O}(\log |A(v)|)$ time.

**Example:** $LCA(p) = 0010110110110110$, $LCA(v) = 0100000110000100$, $select =< 2, 5, 6 >= 0000000000110010$
$smallList(v|p) = index(< 2, 5, 6 >) = index(0000000000110010) = 0,00101,00100,00001$

## The MST verification algorithm - generating $smallList(v)$

- **Step 2.2:** if $v$ is small, create $smallList(v)$ by using binary search to find the index $i$ for which the $i^{th}$ and all subsequent elements of $smallList(v|p)$ weigh less than $weight(v,p)$.

  Namely, let $S_{v|p} \leftarrow select(LCA(p), LCA(v))$, $i_2 \leftarrow hweight(S_{v|p})$-1, $i_1 \leftarrow 0$, $i \leftarrow \lfloor \frac{1}{2} i_2 \rfloor$ and retrieve the $k^{th}$ tag of $bigList(z)$ by $(bigList_z[d] >> (tagsize * (k - d * subwnum)))\&((1 << tagsize) - 1)$ where $k$ is the content of the $i^{th}$ subword of $smallList(v|p)$ and $d \leftarrow \lfloor k/subwnum \rfloor$, then use the *tag property* to obtain the weight for this tag (we elaborated on how to do it in $\mathcal{O}(1)$ time when we discussed Schieber-Vishkin), then compare the obtained weight with $weight(v,p)$ and continue with half the range by updating $i, i_1, i_2$. Let $j$ be the index of the last element of $A(v)$, i.e. $j \leftarrow size_{A(v)} - 1$. Now compose $tag(v)$ from $INLABEL(v)$ and $LEVEL(v)$, and compute the word $w \leftarrow subw1 * tag(v)$ in which every subword consists of $tag(v)$.

## The MST verification algorithm - generating $smallList(v)$ (cont.)

- Finally, create $smallList(v)$ from $w$ and $smallList(v|p)$ by $\mathcal{O}(1)$ shifts and masks according to the $i$ and $j$ offsets, so that $smallList(v)$ consists of the first $i-1$ subwords of $smallList(v|p)$ followed by the $i$ through $j$ subwords of $w$.
  Notice that because we insert $tag(v)$ to subwords of $smallList(v)$, in subsequent invocations of this step some subwords of $smallList(v|p)$ may contains tags instead of indices of $bigList(z)$. But since we always copy $tag(v)$ consecutively until the last element, we just need to maintain for each $smallList$ the index of its first tag.
  So along with each $smallList$ we store $smallList_{idx}$ in another word, and during the binary search if $i$ is greater or equal to the position of the first tag in $smallList(v|p)$ (this position can be obtained by masking $S_{v|p}$ with $smallList_{idx}(p)$ and getting the $hweight$ of the result), then we retrieve the tag directly from the content of the $i^{th}$ subword of $smallList(v|p)$, instead of accessing $bigList(z)$.
  Thus the time of step 2.2 is $\mathcal{O}(1) \cdot \lceil \log |A(v|p)| \rceil \leq \mathcal{O}(\log |A(v)|)$.

Note: because we always insert $tag(v)$ starting at an index computed in accordance with the $LCA$ bit lists, it's possible to save some work by masking $tag(v)$ until the last subword (instead of until $j$).

## The MST verification algorithm - generating $bigList(v)$

- **Step 3.1:** if $v$ is big and $z$ exists, create $bigList(v|z)$ from $bigList(z)$, $LCA(v)$ and $LCA(z)$ by:
  - $L \leftarrow select(LCA(z), LCA(v))$
  - construct $\mathcal{O}(\log\log n)$ words by partitioning $L$ into strings $L_i$ of $subwnum$ consecutive bits each, namely for $i = 0, 1, 2, 3, \ldots$ do $L_i \leftarrow (L >> (i * subwnum))\&((1 << subwnum) - 1)$
  - for each $i$ do $R_i \leftarrow selectS(L_i, bigList_z[i])$
  - concatenate the rightmost $tagsize \cdot hweight(L_i)$ bits from each $R_i$ and thereby form the array $bigList(v|z)$.

We iterate $\mathcal{O}(\log\log n)$ times and perform $\mathcal{O}(1)$ operations in each iteration, thus the total time is bounded by $\mathcal{O}(\log\log n)$.

**Example**

$LCA(z) = 0010110110110110$, $LCA(v) = 0100010010110100$, $L =< 2, 3, 4, 5, 7 >= 0000000001011110$
$L_0 = 0000000000000110$, $L_1 = 0000000000000011$, $L_2 = 0000000000000001$
$bigList_z[0] = 0{,}11111{,}10101{,}01010$, $bigList_z[1] = 0{,}11011{,}00010{,}01110$, $bigList_z[3] = 0{,}10111{,}01111{,}01011$
$R_0 = 0{,}00000{,}11111{,}10101$, $R_1 = 0{,}00000{,}00010{,}01110$, $R_2 = 0{,}00000{,}00000{,}01011$
$bigList_{v|z}[0] = 0{,}01110{,}11111{,}10101$, $bigList_{v|z}[1] = 0{,}00000{,}01011{,}00010$
$bigList(v|z) =< 10101, 11111, 01110, 00010, 01011 >$

## The MST verification algorithm - generating $bigList(v)$

- **Step 3.2:** if $v$ is big and $p \neq z$, create $bigList(v|p)$ from $bigList(v|z)$ and $smallList(p)$ by:
  - insert the subwords of $smallList(p)$ at positions $smallList_{idx}(p)$ and higher into $bigList(v|z)$ starting also from this position.

Rationale: If $v$ is big and $p$ is small, it means that $|A(v)| = 1 + |A(p)|$ where the one extra element is $treepath(v, p)$, and in this case $A(v|p) = A(p)$. However, we cannot use $smallList(p)$ directly and just insert the one extra tag in order to form $bigList(v)$, because $smallList(p)$ might contains indices of $bigList(z)$ instead of tags, and we rely on having only tags in the $bigList$ of any node. Observe that all the paths of $A(v|p)$ before position $smallList_{idx}(p)$ have their heaviest weight in $treepath(root, z)$, and we already computed their tags in $bigList(v|z)$, and all the (less heavy) paths starting at position $smallList_{idx}(p)$ have their heaviest weight in $treepath(z, p)$, and their tags are stored in $smallList(p)$. Therefore we just need to concatenate the $smallList_{idx}(p) - 1$ subwords of $bigList(v|z)$ with the subwords of $smallList(p)$ starting at $smallList_{idx}(p)$. Since $|A(v|z)| \leq |A(v|p)| = |A(p)|$, both $smallList(p)$ and $bigList(v|z)$ are stored in a single word, and therefore this step can be done in $\mathcal{O}(1)$ time by masking the $tagsize \cdot smallList_{idx}(p)$ and higher bits of $smallList(p)$ into $bigList_{v|z}[0]$.

## The MST verification algorithm - generating $bigList(v)$

- **Step 3.3:** if $v$ is big, create $bigList(v)$ by using binary search in order to insert $weight(v, p)$ into $bigList(v|p)$ at the appropriate positions.

  This is done similarly to step 2.2, i.e. binary search with $i, i_1, i_2$, and in fact it's easier because we always retrieve a tag and not an index from the content of the $i^{th}$ subword of $bigList(v|p)$.

  Let us mention that we initialize $bigList(v|z)$ as an empty array of $size_{A(v)}$ subwords as its allocated size (meaning $\lceil \frac{size_{A(v)}}{subwnum} \rceil$ words), and in this step we transform this array into $bigList(v)$ by inserting $weight(v, p)$ at the appropriate positions, as in step 2.2.

  The binary search takes $\mathcal{O}(1) \cdot \lceil \log |A(v|p)| \rceil$, and the overhead for inserting $weight(v, p)$ into consecutive words is $\mathcal{O}(\log \log n)$, thus the time for this step is $\mathcal{O}(\log |A(v)|) + \mathcal{O}(\log \log n) \leq \mathcal{O}(\log \log n)$.

## The MST verification algorithm - verifying the cycle property

- **Step 4:** after $A(v)$ was built for all nodes, we now verify that the cycle property holds for each of the $m$ tree-path queries.

  Verify that each non-tree edge $(u, w)$ is as heavy as $treepath(u, w)$, by:

  1. $pos_u \leftarrow hweight(LCA(u)\ \&\ ((1 << L\tilde{C}A(u, w)) - 1))$
  2. if $u$ is small:
     $tag_u \leftarrow (smallList(u) >> (tagsize * pos_u))\ \&\ ((1 << tagsize) - 1)$
  3. retrieve the weight from $tag_u$ and compare it to $weight(u, w)$
  4. $pos_w \leftarrow hweight(LCA(w)\ \&\ ((1 << L\tilde{C}A(u, w)) - 1))$
  5. if $w$ is big:
     $i \leftarrow \lfloor pos_w / subwnum \rfloor$, $d \leftarrow tagsize * (pos_w - i * subwnum)$
     $tag_w \leftarrow (bigList_w[i] >> d)\ \&\ ((1 << tagsize) - 1)$
  6. retrieve the weight from $tag_w$ and compare it to $weight(u, w)$

The idea is to compute the index of $treepath(LCA(u, w), u)$ in $A(u)$, by taking the level of $LCA(u, w)$ and using *hweight* to check how many '1' bits exist before this level in the bit list $LCA(u)$. We then demonstrated how to obtain the tag stored at this index of $smallList(u)$ or $bigList(w)$, as examples. Retrieving the weight of the edge tag is done via the *tag property* in $\mathcal{O}(1)$ time, therefore step 4 runs in $\mathcal{O}(1) \cdot m = \mathcal{O}(m)$ total time.

## Analysis of the MST verification algorithm

### Concluding analysis of the V. King MST verification algorithm

We have proved that generating $smallList(v)$ or $bigList(v)$ for each node $v$ is done in $\mathcal{O}(\log(1 + |A(v)|))$ time. As with the analysis of the Komlós algorithm, this implies $\mathcal{O}(m + n)$ total time and space to generate *smallList* or *bigList* for all the nodes in the tree.

After the *smallList* and *bigList* data has been built for all the nodes, verifying the cycle property in step 4 takes $\mathcal{O}(m)$ time.

We have also seen that precomputing the $LCA$ table is done in $\mathcal{O}(m + n)$ time. The $LCA(v)$ data is then stored in $\mathcal{O}(n)$ space, and the $L\tilde{C}A(u, w)$ data is stored in $\mathcal{O}(m)$ space.
The preprocessing for the unconventional functions that we use takes $\mathcal{O}(n)$ time and space.

Thus the total time for the MST verification algorithm is linear.

## References



D. Karger, P. N. Klein and R. E. Tarjan: *A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees*.
JACM 42 (1995), 321-328


V. King: *A Simpler Minimum Spanning Tree Verification Algorithm*.
LNCS 955 (1995), 440-448
Algorithmica 18 (1997), 263-270


J. Komlós: *Linear Verification for Spanning Trees*.
Combinatorica 5 (1985), 57-65


B. Schieber and U. Vishkin: *On Finding Lowest Common Ancestors: Simplification and Parallelization*.
SIAM J. Comput. 17 (1988), 1253-1262