Fortgeschrittene Datenstrukturen — Vorlesung 9

Schriftführer: Christian Schmitz

22.12.2011

1 Lowest Common Ancestor Queries

1.1 The LCA-Problem

Definition 1. *Let T be a rooted tree. The **lowest common ancestor** (LCA) of two nodes x and y is the deepest node which is an ancestor of both x and y and is denoted by $LCA_T(x, y)$.*

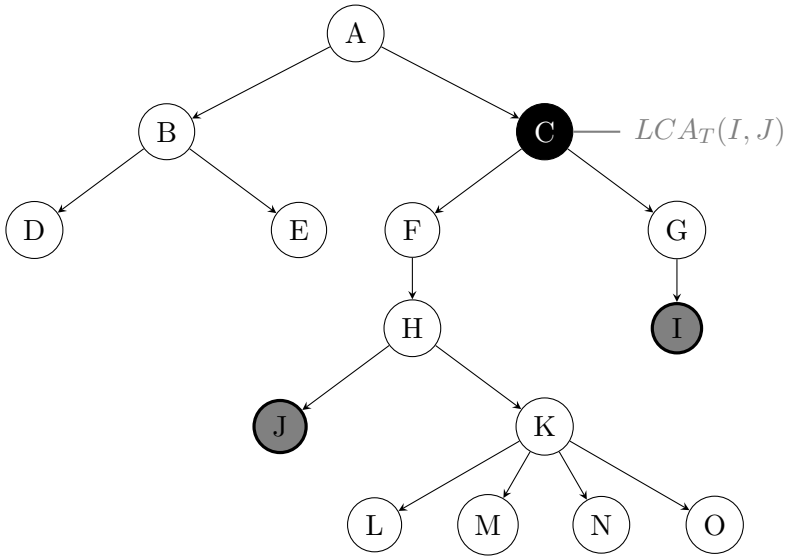


Figure 1: Example of a lowest common ancestor of two nodes I and J

According to this definition, the LCA of nodes I and J in the tree shown in figure 1 is C. In order to find the LCA for any node efficiently, we regard the LCA-problem. The *LCA-Problem* deals with the preprocessing of a static tree T on n nodes such that subsequent LCA-queries can be answered in constant time.

Naively, the LCA-problem can be solved either by storing the answer to all possible solutions or by saving only the tree and doing calculations on that stored information only. The first approach consumes $\Theta(n^2)$ space but the time for processing the query is constant. Nevertheless it is well known that the LCA-problem can be solved with linear space. The second approach only takes $\Theta(n)$ space but in the worst case $O(n)$ time is needed to find the LCA of two nodes. Therefore,

we need to find a trade-off between space and time consumption in order to compute the LCA efficiently. Our approach should be bound to $O(n)$ space consumption and $O(1)$ query time.

Slightly modifying the problem, we will search for a solution which requires the preprocessing algorithm to assign labels to each node. Using these labels the $LCA_T(x, y)$ can be found efficiently by just looking at the labels of nodes x and y. Additionally, we will restrict the problem by not allowing the usage for a global shared memory. Therefore, we work with a *distributed data structure*. This means that we cannot use a lookup table, for example. Hence, we do not consider the case in which we have to return a pre-defined label but a self-defined one.

1.2 Labelling Scheme

An intuitive way for a labelling scheme for the LCA-problem is to concat the Depth-First Search (DFS) number of a node with its parent's label, i.e. $label(v) = label(parent(v)) \circ DFS(v)$. Figure 2 shows this labelling for the graph shown in figure 1.

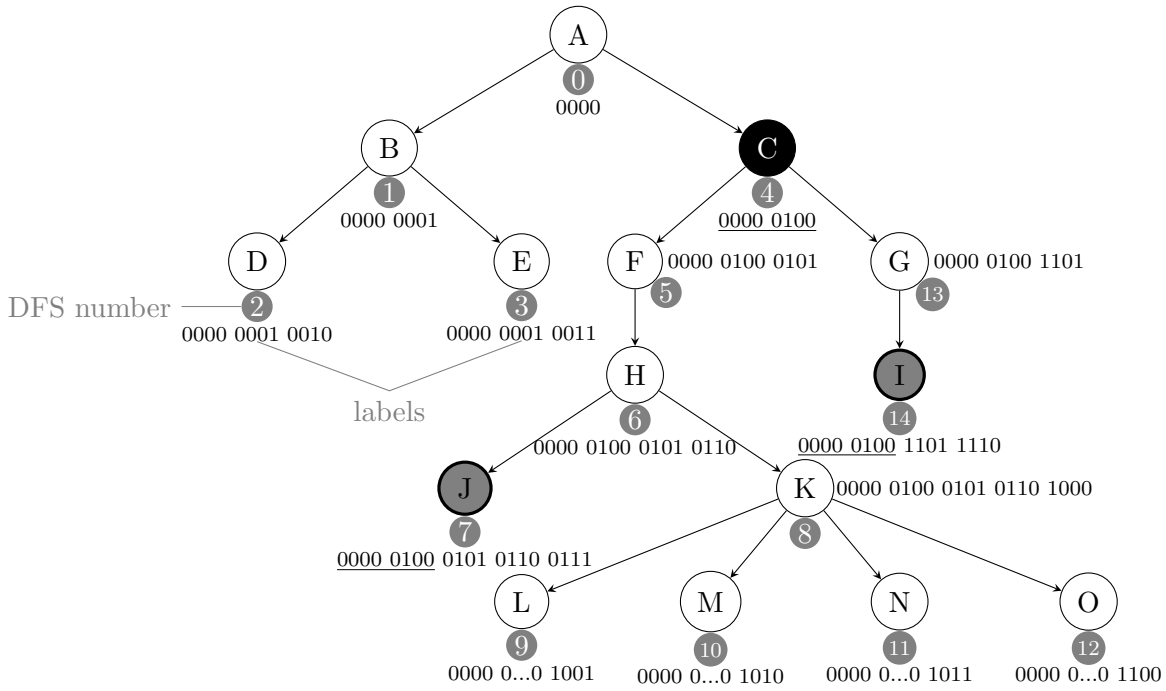


Figure 2: Naive labelling scheme for the LCA-problem

As shown in this figure, the $LCA_T(I, J)$ can be found by getting the node whose label equals the longest common prefix (LCP) of the labels of I and J, i.e. $label(LCA_T(I, J)) \approx LCP(label(I), label(J))$. Though, the label length increases for every child node leading to a size for the longest label between $\lg^2 n$ and $n \lg n$ bits. This is not very space efficient. Hence, the aim is to minimize the label lengths, which can be as small as $O(\lg n)$ bits.

For a more efficient way of labelling we have to introduce some new node and path characteristics. Let T_v denote T's *subtree* rooted at v and $|T_v|$ the *number of nodes* in T_v. Let $parent(v)$ be the parent of v and $children(v)$ the set of v's children. Using these definitions a node can be

classified as either light or heavy.

Definition 2. *For every internal node v exactly one child $u \in children(v)$ with $|T_u| = \max\{|T_w| : w \in children(v)\}$ is called **heavy**, the other children are called **light**. The root is always called light.*

If two children have the same subtree size only one of them is called heavy. The choice, which one this is, is not restricted to a certain pattern.

Definition 2 is visualized in figure 3. All children of v_0 are light nodes but v_2. Its subtree is the biggest one of all subtrees of $children(v_0)$. Therefore v_2 is a heavy node.

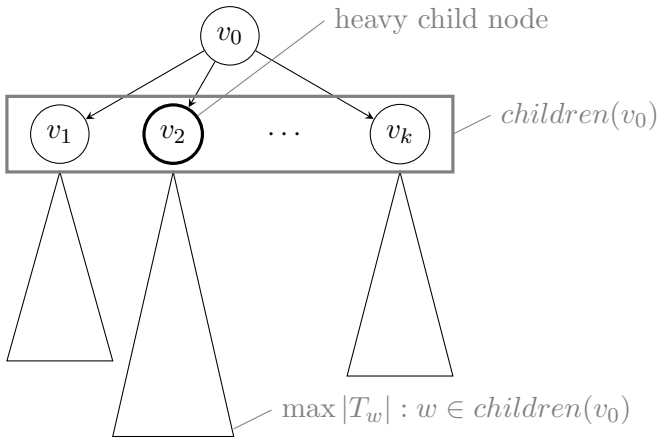


Figure 3: Visualization of the heavy node concept

The heavy nodes divide T into disjoint *heavy paths*. A heavy path can be found by following the heavy nodes starting at a light node. Let $\langle v_1, ..., v_k \rangle$ be a heavy path. Node v_1 is called the apex of all nodes on that heavy path, devoted by $apex(v_i)$. The $apex(v_i)$ of any heavy path is therefore always a light node. Evidently, all heavy paths in T can be found recursively. Starting at the root (a light node) the heavy path whose apex it is can be determined by following its heavy ancestors. In a next step for all light children of the nodes on the just found heavy path the heavy paths are determined the same way. The heavy path with $apex(v_i)$ being the root of T is the longest heavy path in this tree. $HP(v)$ describes the set of all nodes on the same heavy path as node v. A light leaf is its own apex.

In figure 4 the heavy paths for the example tree is shown. The heavy paths are indicated by thicker arrows and heavy nodes by thicker circles. The longest heavy path is $\langle A, C, F, H, K, L \rangle$. A is the apex of all nodes on that path.

In order to apply our final labelling scheme, we firstly have to assign heavy and light labels to the nodes in the tree. The heavy label $hl(v)$ is assigned to every node v in the tree. It is different for two nodes in one heavy path. The constraint on the heavy labels is that for a heavy path $\langle v_1, ..., v_k \rangle$ and two nodes v_i and v_j on that heavy path it can be determined in $O(1)$ time which of the nodes v_i and v_j is an ancestor of the other. The heavy label of a light leaf is always 0. The light labels $ll(v)$ are assigned to light nodes v only. Light nodes with the same parent node should be distinguishable by this label. The root is assigned by an empty string (indicated by ε).

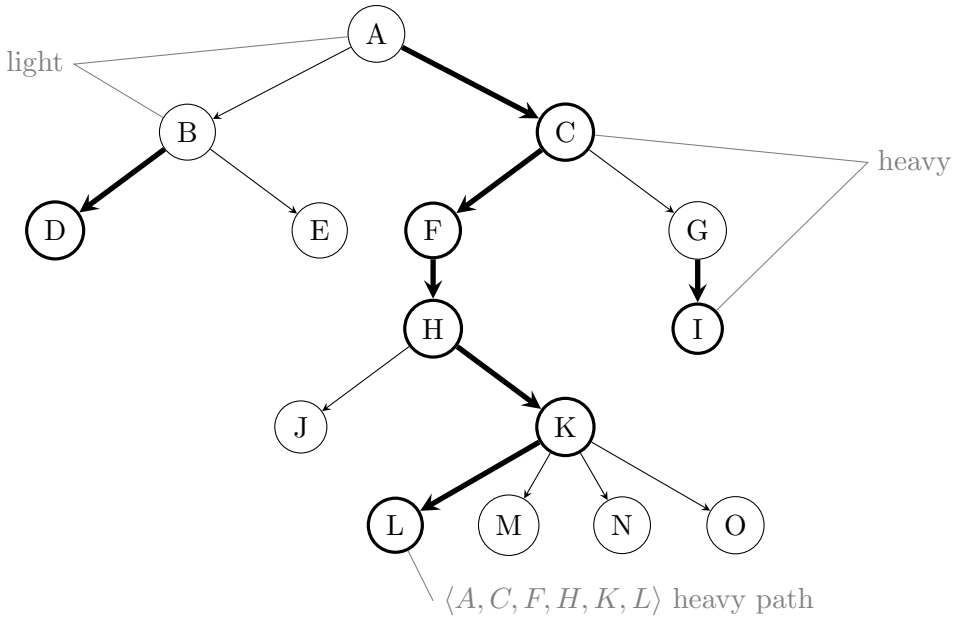


Figure 4: Tree with its heavy paths

Definition 3. *The label $l(v)$ of a node v is then defined as:*

$$l(v) := l(parent(apex(v))) \circ ll(apex(v)) \circ hl(v)$$

The first part of this label scheme allows a recursive climbing upwards the tree. The second part is needed to decide whether two nodes are in the same branch. The third part, at last, distinguishes two nodes in the same heavy path. For the label of the parent of the root also an empty string (i.e. ε) is assigned. Therefore, for all nodes on the longest heavy path the heavy label equals the actual label since $l(parent(apex(root))) = \varepsilon$ as well as $ll(apex(root)) = \varepsilon$.

Example 1. *Figure 5 shows the application of the labelling scheme in which the label concatination is visualized as the following* $l(parent(apex(v)))\,|\,ll(apex(v))\,|\,hl(v)$ *For nodes in $\langle A, C, F, H, K, L\rangle$ the label of the node is the heavy label. In the following the compositions of the labels for nodes B, D and N are explained.*

The label of node B is 000 0 0. $hl(B) = 0$ because it is the apex of a heavy path with two nodes. Therefore the two nodes can be distinguished by only one bit. In order to meet the ordering constraint the bit is 0 while the heavy label bit of its only child in the heavy path (i.e. node D) is 1. Since B is its own apex $ll(apex(B)) = ll(B) = 0$. It consists only of one bit because there are no other light nodes for $children(parent(B)) = children(A)$. 000 is the part of the label representing $l(parent(apex(B)))$. Since B is its own apex and A is $parent(B)$ this part equals the label of A.

Node D is labelled 000 0 1. As explained before $hl(D) = 1$ in order to distinguish and to order the two nodes in $\langle B, D\rangle$. The apex of D is B. Therefore $ll(apex(D)) = ll(B) = 0$. For the first part of the label the label of the parent of the apex of D is taken which is as for B the label of node A.

Node N is a light leaf in the tree. Hence $hl(N)$ is encoded as 0. The first part of its label is its parent label. Though, N has to be distinguishable from its light node siblings M and O. This

is the reason why its light label consists of the two-bit string 01 which is part of the node's label as $ll(apex(N))$.

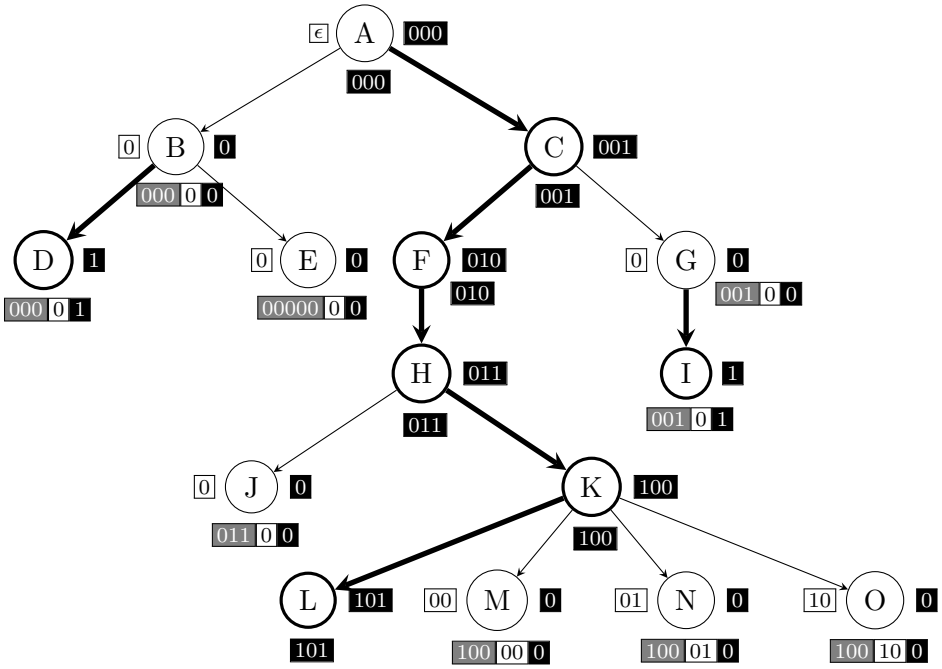


Figure 5: Tree with final labelling scheme applied to nodes

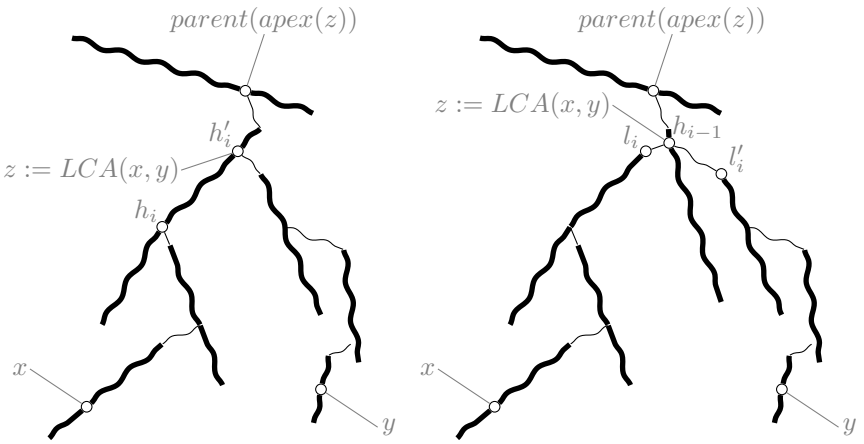
1.3 Query Answering

From the structure of the labelling scheme it follows that $l(v)$ is a concatenation of alternating heavy and light labels, i.e. $l(v) = h_0 \circ l_1 \circ h_1 \circ ... \circ l_t \circ h_t$. This structure is used to answer LCA queries for two nodes. Given two nodes x and y and their corresponding labels $l(x)$ respectively $l(y)$ the $LCA(x, y)$ can be inferred by computing the LCP of $l(x)$ and $l(y)$. In order to infer the label of $LCA(x, y)$ the first mismatch has to be considered. As shown in figure 6 the first mismatch can occur either in a heavy (a) or a light label (b).

Regarding the first case that the first mismatch occurs in a heavy label the labels of x and y look as follows:

$$\begin{array}{rcl} l(x) & = & \overbrace{h_0 \circ l_1 \circ h_1 \circ ... \circ h_{i-1}}^{l(parent(apex(z)))} \circ l_i \circ \quad h_i \quad \circ ... \\ l(y) & = & h_0 \circ l_1 \circ h_1 \circ ... \circ h_{i-1} \circ l_i \circ \quad \underbrace{h'_i}_{mismatch} \quad \circ ... \end{array}$$

If the heavy label is the first mismatch it can be concluded that l_i is the light label of the apex of the two nodes with heavy label h_i and h'_i respectively. Furthermore $h_0 \circ l_1 \circ h_1 \circ ... \circ h_{i-1}$ is the label of the parent of this apex. Therefore, those nodes are on the same heavy path. Since heavy



(a) First mismatch at h_i (b) First mismatch at l_i

Figure 6: Graphs indicating the two possibilities for a first mismatch for two labels $l(x)$ and $l(y)$

labels are defined to distinguish two heavy nodes on one heavy path h_i and h'_i now determine whose corresponding node is $z := LCA(x, y)$. The determination of the LCA of two nodes is evidently feasible in $O(1)$ by returning the node labelled $h_0 \circ l_1 \circ h_1 \circ ... \circ h_{i-1} \circ l_i \circ h'_i$ if the node with heavy label h'_i is the ancestor of the node with heavy label h_i, i.e. h'_i is lexicographically smaller than h_i. In other words the LCA is $z = h_0 \circ l_1 \circ h_1 \circ ... \circ h_{i-1} \circ l_i \circ \min_{lex}\{h_i, h'_i\}$. In the case of h_0 being the first mismatch the LCA is $z = \min_{lex}\{h_i, h'_i\}$.

In the second case that a light label is the first mismatch the labels of x and y look as follows:

$$\begin{array}{rcl} l(x) & = & \overbrace{h_0 \circ l_1 \circ h_1 \circ ... \circ l_{i-1} \circ h_{i-1}}^{l(LCA(x,y))} \circ \quad \underbrace{l_i}_{} \quad \circ ... \\ l(y) & = & \underbrace{h_0 \circ l_1 \circ h_1 \circ ...}_{l(parent(apex(z)))} \circ l_{i-1} \circ h_{i-1} \circ \quad \underbrace{l'_i}_{mismatch} \quad \circ ... \end{array}$$

Light labels are used to distinguish different light children for a heavy parent. A mismatch in a light label then means that at a heavy node in the tree the branch for x follows a different light node than the one for y. Therefore that heavy node is $z := LCA(x, y)$. The label of the heavy node is $l(LCA(x, y)) = h_0 \circ l_1 \circ h_1 \circ ... \circ l_{i-1} \circ h_{i-1}$. Knowing this the LCA query can be answered in $O(1)$.

A special case of the light label mismatch is the case when the comparison reaches the end of one label while no mismatch has occurred yet, i.e. l_i and ε differ. This case might happen if the $LCA(x, y)$ is looked for with y being a node in a subtree of one light child of x. Then the label of x is the prefix of the label of y and $x = LCA(x, y)$.

In order to decide whether a bit of a label belongs to a heavy or a light label a helper label $h(v)$ of the same size as $l(v)$ is needed. For every specific bit in $l(v)$ $h(v)$ saves whether this bit is part of a heavy or a light label by 0 or 1 respectively.

1.4 Space Analysis

In the following we will look for a code for the labels which is space efficient for our means. Before we do so, though, the following definition is needed.

Definition 4. *The light size of v is defined as*

$$lsize(v) = |T_v| - |T_w|,$$

for w being the heavy child of v. The light size of a leaf is defined as 1.

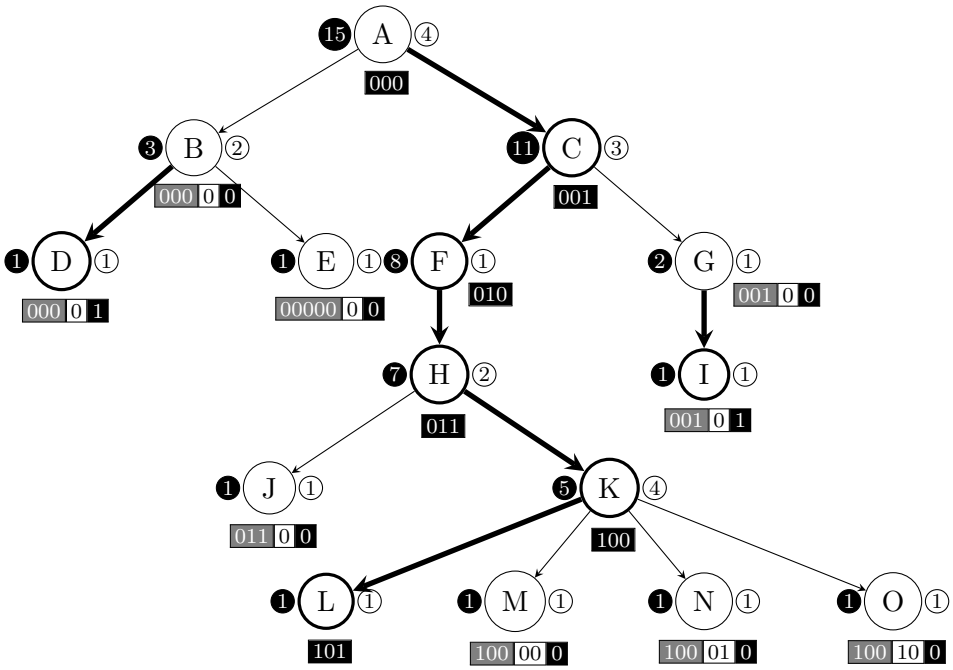


Figure 7: Tree with final labelling scheme and size assigned to the left and light size to the right of the nodes

The assignment of $lsize(v)$ to v is a bottom-up procedure. Starting at the leafs of a tree and knowing the subtree size $|T_v|$ for every node v in T the light size is determined. Going up the tree the light size for the predecessor nodes is calculated as given by definition 4.

Knowing $|T_v|$ and $lsize(v)$ for every node v provides knowledge about the amount of occurrences of parts of $label(v)$. A heavy label $hl(v)$ occurs in all descendants of v apart from the ones under the heavy child of v, i.e. $hl(v)$ is reused in all nodes below $w \in children(v) \setminus \{heavy\ child\ of\ v\}$. Therefore it occurs $lsize(v)$ times in T. A light label $ll(v)$ occurs in all nodes below v and hence $|T_v|$ times.

The sizes and the derived light sizes for the example tree are shown in figure 7. As can be seen for C the light size is 3. Following its only light path there are two descendants whose label reuse the label of C. Counting into the label of C as well, its label is used three times in different labels. For B $|T_B|$ is 3. Its light label is used three times in T, namely in B and all its children D and E.

We conclude that the sizes of the labels should be chosen adaptively in order to minimize the overall label length. Hence, for heavy labels it makes sense to assign short labels to nodes with large $lsize$ while for light labels nodes with larger subtree size $|T_v|$ should be assigned with shorter light labels.

A suitable code for the application in the LCA-Problem should therefore make use of these label length requirements on the one hand. On the other hand the code must not lose the lexicographical ordering. A satisfying code is developed in the following. Let $\langle y_1, ..., y_k \rangle$ be a sequence of positive integers with $\sum y_i = s$. $\langle c_1, ..., c_k \rangle$ is then a sequence of binary strings and an alphabetical sequence for $\langle y \rangle_k$.

For $\langle c_1, ..., c_k \rangle$ we demand that

$$c_1 \leq_{lex} c_2 \leq_{lex} ... \leq_{lex} c_k \tag{1}$$

$$|c_i| \leq \lg s - \lg y_i + O(1) \tag{2}$$

In equation (1) it is defined that a lexicographically constant comparison must be possible. Equation (2) bounds the length of a single code.

For the derivation of $\langle c_1, ..., c_k \rangle$ let $s_i = \sum_{j=1}^{i} y_j$ and $s_0 = 0$ for non-null integers y_i. Furthermore let $f_i = \lfloor \lg y_i \rfloor$ and $I = [0, s)$ be an integer interval split up k-times by $I_i = [s_{i-1}, s_{i-1} + y_i)$. In the interval I_i of length $y_i \geq 2^{f_i}$ there must be an integer z_i such that $z_i \mod 2^{f_i} = 0$, i.e. the f_i rightmost bits in its binary representation are 0. If $s_{i-1} \mod 2^{f_i} = 0$ then $z_i = s_{i-1}$, otherwise $z_i = s_{i-1} - (s_{i-1} \mod 2^{f_i}) + 2^{f_i}$. Hence, z_i can be represented in a word with $w = \lceil \lg s \rceil$ bits having the f_i less significant bits set to 0. Taking the $w - f_i$ most significant bits of z_i as c_i results in lexicographically increasing numbers of size:

$$\begin{aligned} |c_i| &= w - f_i \\ &= \lceil \lg s \rceil - \lfloor \lg y_i \rfloor \\ &= \lg s - \lg y_i + O(1). \end{aligned}$$

This shortened binary representation of z_i is sufficient to distinguish I_i from all other I_j lexicographically.

Example 2. *For a given $\langle y \rangle_k$ we can compute $\langle s \rangle_k$, $\langle f \rangle_k$ and $\langle z \rangle_k$ as defined:*

$$\begin{aligned} \langle y \rangle_k &= \langle 3, 5, 3, 4, 1 \rangle \\ \langle s \rangle_k &= \langle 3, 8, 11, 15, 16 \rangle \\ \langle f \rangle_k &= \langle 1, 2, 1, 2, 0 \rangle \\ \langle z \rangle_k &= \langle 0, 4, 8, 12, 15 \rangle \end{aligned}$$

$s = s_k = 16$ and hence $w = \lceil \lg s \rceil = 4$ bits are needed to get the binary representation of the s integers in I. The binary representation is shown in the table below.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
0	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*
0	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*
0	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*

In order to get c_i the binary representation of z_i is taken and the last f_i bits are discarded.
$\Rightarrow \langle c \rangle_k = \langle 000, 01, 100, 11, 1111 \rangle$

As can be seen in example 2 as well, the bigger y_i gets compared to s the fewer bits are used for c_i. Therefore, this code is suitable for our purpose.

We stated that light labels of nodes with larger subtree size $|T_v|$ should be assigned with shorter light labels. Therefore we set $y_i = |T_{v_i}|$ for all $v_i \in children(v)$. s in this case includes all nodes of all subtrees of those v_i. The amount of these nodes is smaller than $lsize(parent(v_i))$. Hence, $|ll(v)| \leq \lg lsize(parent(v)) - \lg |T_v| + O(1)$.

For the heavy labels short labels should be assigned to nodes with large $lsize$. This means that we define $y_i = lsize(v)$ for all v on the same heavy path. s is then defined as $s = \sum_{w \in HP(v)} lsize(w)$. Therefore it can be concluded that $|hl(v)| \leq \lg \sum_{w \in HP(v)} lsize(w) - \lg lsize(v) + O(1)$.

Knowing these facts we can now prove that $|l(v)| = O(\lg n)$. First of all, we can ignore the $O(1)$-terms in the analysis. This can be done because there are at most $\lg n$ light nodes on a path from a leaf to the root. Therefore all these additive $O(1)$-terms add up to $O(\lg n)$ at most.

By induction on the depth of the apices we show that $|l(v)| \leq \lg n - \lg lsize(v)$. For a node v with $apex(v)$ being the root we have $l(v) = hl(v)$ and hence

$$|l(v)| = \lg \sum_{w \in HP(v)} lsize(w) - \lg lsize(v) \qquad \leq \lg n - \lg lsize(v).$$

So the statement holds. For a node v with $apex(v)$ not being the root we have $l(v) = l(parent(apex(v))) \circ ll(apex(v)) \circ hl(v)$. Hence

$$\begin{aligned}
|l(v)| &= \underbrace{|l(parent(apex(v)))|}_{\leq \lg n - \lg lsize(parent(apex(v)))} + \overbrace{|ll(apex(v)|}^{\leq \lg lsize(parent(apex(v))) - \lg |T_{apex(v)}|} + \underbrace{|hl(v)|}_{\leq \lg \sum_{v \in HP(v)} lsize(v) - \lg lsize(v)} \\
&\leq \lg n - \lg |T_{apex(v)}| + \lg \underbrace{\sum_{v \in HP(v)} lsize(v)}_{\leq |T_{apex(v)}|} - \lg lsize(v) \\
&\leq \lg n - \lg lsize(v)
\end{aligned}$$

□

Therefore we have n labels with a size of $O(\lg n)$ bits. In sum this means $O(n)$ words are needed, since one word consists at least of $\lg n$ bits.

Example 3. *In this example the heavy and light labels for our example tree are calculated. The result is shown in figure 8.*

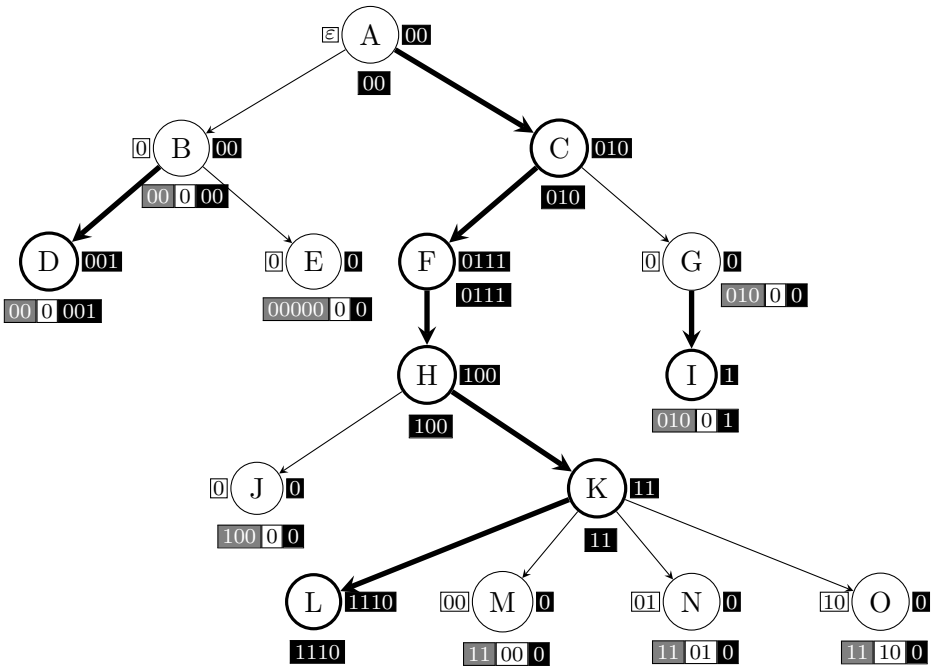


Figure 8: Tree with final coding applied

Exemplified the calculation for the heavy labels of all nodes on $HP(A)$ and for light labels for nodes M, N and O are shown.

$$
\begin{aligned}
\langle v \rangle_k &= \langle A, C, F, H, K, L \rangle \\
\langle y \rangle_k &= \langle lsize(A), lsize(C), lsize(F), lsize(H), lsize(K), lsize(L) \rangle = \langle 4, 3, 1, 2, 4, 1 \rangle \\
\langle s \rangle_k &= \langle 4, 7, 8, 10, 14, 15 \rangle \\
\langle f \rangle_k &= \langle 2, 1, 0, 1, 2, 0 \rangle \\
\langle z \rangle_k &= \langle 0, 4, 7, 8, 12, 14 \rangle \\
s &= \sum_{w \in HP(A)} lsize(w) = 4 + 3 + 1 + 2 + 4 + 1 = 15 \\
w &= \lceil \lg s \rceil = 4
\end{aligned}
$$

For the label of A: $hl(A) = 00$, since the last two bits are left out from the binary representation of 0 with 4 bits. Analogously, $hl(C) = 010$, $hl(F) = 0111$, $hl(H) = 100$, $hl(K) = 11$ and $hl(L) = 1110$.

The determination of the light labels for M, N and O is shown in the following:

$$\begin{aligned}
\langle v\rangle_k &= \langle M, N, O\rangle \\
\langle y\rangle_k &= \langle |T_M|, |T_N|, |T_O|\rangle = \langle 1, 1, 1\rangle \\
\langle s\rangle_k &= \langle 1, 2, 3\rangle \\
\langle f\rangle_k &= \langle 0, 0, 0\rangle \\
\langle z\rangle_k &= \langle 0, 1, 2\rangle \\
s &= lsize(parent(M)) = 3 \\
w &= \lceil \lg s \rceil = 2
\end{aligned}$$

This leads to light labels $ll(M) = 00$, $ll(N) = 01$ and $ll(O) = 10$.