Abstract. A level-ancestor or LA query about a rooted tree $T$ takes as arguments a node $v$ in $T$, of depth $d_v$, say, and an integer $d$ with $0 \leq d \leq d_v$ and returns the ancestor of $v$ in $T$ of depth $d$. The static LA problem is to process a given rooted tree $T$ so as to support efficient subsequent processing of LA queries about $T$. All previous efficient solutions to the static LA problem work by reducing a given instance of the problem to a smaller instance of the same or a related problem, solved with a less efficient data structure, and a collection of small micro-instances for which a different solution is provided. We indicate the first efficient solution to the static LA problem that works directly, without resorting to reductions or micro-instances.

Keywords: LA problem, find-smaller queries, ladders, jump tables

1 Introduction

A level-ancestor or LA query about a rooted tree $T$ takes as arguments a node $v$ in $T$, of depth $d_v$, say, and an integer $d$ with $0 \leq d \leq d_v$ and returns the ancestor of $v$ in $T$ of depth $d$ (or, in some formulations, of depth $d_v - d$). LA queries have applications, e.g., to the computation of semigroup sums over paths in trees [5], the aggregation of minima over subtrees [11] and the recognition of breadth-first-search trees [7,8]. They are also considered part of the repertoire of operations that a well-endowed data structure for representing rooted trees should support (see, e.g., [9]).

One distinguishes between static and dynamic versions of the LA problem of supporting efficient LA queries. In dynamic versions of the problem [1,6,10], LA queries are interspersed with calls of operations that change the structure of the underlying tree. In the static version of the problem, which forms the focus of the present text, the tree is given once and for all, and the task is to preprocess it so that subsequent LA queries can be executed fast.

The static LA problem was first considered by Berkman and Vishkin [4], who reduced it to a problem about a sequence of integers. Specifically, suppose that a depth-first search (DFS) of a rooted tree $T$ appendes (the name of) a node $v$ in $T$ to an initially empty sequence whenever $v$ is visited for the first time or the DFS withdraws to $v$. This yields a sequence $P_1$ of $2n - 1$ nodes. The DFS can also mark each node $v$ with its depth in $T$ and with a position in $P_1$ in which $v$ occurs. Replacing each node in $P_1$ by its depth in $T$ yields a new sequence $P_2$ of integers. The ancestor in depth $d$ of a node $v$ in $T$ of depth $d_v$, where $0 \leq d \leq d_v$, can now be found as the node in $P_1$ in the same position as the first occurrence in $P_2$ of a number bounded by $d$ in or following the position that in $P_1$ contains an arbitrary occurrence of $v$. To answer LA queries about $T$, it therefore suffices to
be able to answer FS ("find-smaller") queries about $P_2$, where an FS query about a sequence $P = (d_0, \ldots, d_{m-1})$ of integers takes as arguments integers $i$ and $d$ and returns $\min\{j \in \{0, \ldots, m-1\} \mid j \geq i \text{ and } d_j \leq d\} \cup \{m\}$.

Say that a sequence $Y$ is a 1-difference sequence if $Y = (y_0, \ldots, y_{n-1})$ for integers $y_0, \ldots, y_{n-1}$ with the property that $|y_i - y_{i-1}| \leq 1$ for $i = 1, \ldots, n - 1$. Of course, $P_2$ above has this property. Berkman and Vishkin gave a family of parallel algorithms that input a 1-difference sequence $Y$ of length $n$ and output a data structure of $O(n)$ words that enables subsequent FS queries about $Y$ to be answered in constant time by a single processor. Here and in the following, when the space requirements of a data structure are expressed in terms of words, we use the common convention that a word consists of $\Theta(\log n)$ bits. For $k \geq 1$, the $k$th algorithm in the family works in $O(\log^k n)$ time using $O(n/\log^k n)$ processors, where $\log^k$ denotes the $k$-fold iterated logarithm function. A central idea is to equip each $i \in \{1, \ldots, n-1\}$ with precomputed answers to certain FS queries with first argument $i$, choosing the number of such precomputed answers to be proportional to the highest power of 2 that divides $i$. The result of Berkman and Vishkin implies the existence of a sequential algorithm to carry out the preprocessing for FS queries in $O(n)$ time. Such an algorithm, without the complications necessary in a parallel setting, was described by Ben-Amram [2].

A different approach to the LA problem was initiated by Dietz [6]. Here a main idea is to decompose the given tree into a set of paths, to provide complete ancestor information within each path in an array and to introduce a mechanism that allows an LA query to find its relevant path (the one that contains the node to be returned) in constant time. A simpler data structure based on the same idea was described by Bender and Farach-Colton [3]. Yet another data structure, intermediate in complexity, was proposed by Alstrup and Holm [1]. All three data structures occupy $O(n)$ words and can be constructed in $O(n)$ time, just as the data structure of Ben-Amram.

All of the data structures discussed above have at their core a less efficient data structure, and they work by reducing a given instance of the LA or FS problem to a smaller instance of the same or a related problem, which is handled with the less efficient data structure, and a collection of small micro-instances, for which a different solution is provided. More concretely, in the case of the sequential solutions [1,2,3,6], what we call the basic data structure has a logarithmic overhead and needs $\Theta(n \log n)$ preprocessing time and $\Theta(n \log n)$ words of space to handle input instances of size $n$. The tree-based solutions [1,3,6] partition the given tree into a collection of micro-trees of $O(\log n)$ nodes each, “held together” by a macro-tree with $O(n/\log n)$ nodes, the macro-tree is stored in an instance of the basic data structure, the micro-trees are handled with table lookup, and it is shown how to process a top-level query with a constant number of queries in the macro-tree and in micro-trees. In the data structure of Ben-Amram [2], the separation between the original instance and a macro-instance is less clear-cut, but there are still micro-instances of size $O(\log n)$ handled with table lookup.

We describe a new data structure for the static LA problem that works directly, without resorting to reductions or micro-instances, and is the first solution to the
LA problem with this property. In order to highlight what sets the new structure off from its predecessors, we call it the one-level structure. Like the data structures of Berkman and Vishkin [4] and Ben-Amram [2], the one-level structure actually solves the more general FS problem for 1-difference sequences. While our result does not allow us to prove any new asymptotic bounds, we expect the one-level structure to be easier to program and to perform better in practice than the known data structures with the same guaranteed resource bounds. The paper by Bender and Farach-Colton [3] has been cited more than a hundred times, according to Google Scholar. It seems likely that most or all of the applications of solutions to the LA problem described in the scientific literature can benefit from the results developed here.

In fact, we prefer to phrase the discussion in terms of FL ("find-larger") queries defined in complete analogy with FS queries, i.e., an FL query about a sequence \( Y = (y_0, \ldots, y_{n-1}) \) of integers inputs integers \( x \) and \( y \) and returns \( FL(x, y) = \min\{i \in \{0, \ldots, n-1\} \mid i \geq x \text{ and } y_i \geq y \} \cup \{\perp\} \), where \( \perp \) is a default value, considered larger than \( n-1 \), to be returned when the query has no natural answer. Correspondingly, we speak of the one-level FL structure. We believe that the choice of FL over FS leads to more natural intuition and terminology, human beings generally having more experience being above hill sides than being below cave sides. Informally, let us associate with a sequence \( (y_0, \ldots, y_{n-1}) \) of integers the sequence \( ((0, y_0), \ldots, (n-1, y_{n-1})) \) of points in the Euclidean plane and imagine these connected with the line segments \( ((i-1, y_{i-1}), (i, y_i)) \), for \( i = 1, \ldots, n-1 \), to create the contour of a landscape. If an FL query with arguments \( x \) and \( y \), which we will write simply as \( (x, y) \), is nontrivial, i.e., if \( 0 \leq x < n \) and \( y \geq y_x \leq \max_{0 \leq i < n} y_i \), then it can be answered by reporting the \( x \) coordinate of the point in the landscape visible by looking horizontally to the right from the point \( (x, y) \) (\( \perp \) if there is no such point).

2 The One-Level FL Structure

The new one-level FL structure combines ideas of the basic data structures of Ben-Amram [2] and Bender and Farach-Colton [3], even though this may not be apparent at a first inspection. It operates with the notion of valleys. Given a sequence \( (y_0, \ldots, y_{n-1}) \) of \( n \) arbitrary integers and a pair \( (x, y) \) of integers with \( 0 \leq x < n \) and \( y \geq y_x \), informally, the valley of \( (x, y) \) is the \( x \) coordinate of the rightmost deepest point that one can reach from \( (x, y) \) while moving only downwards and to the left and staying above the contour of the landscape. Formally, say that a point \( (x, y) \) is down-left reachable from \( (x, y) \) if \( x \leq i < x \) and \( y_i < y \) for all integers \( i \) with \( x \leq i < x \). Of course, \( (x, y) \) is down-left reachable from itself. To define the valley of \( (x, y) \), where \( x \) and \( y \) are integers with \( 0 \leq x < n \) and \( y \geq y_x \), let \( y \) be minimal such that some point of the form \( (x, y) \) is down-left reachable from \( (x, y) \). Then the valley of \( (x, y) \) is the largest \( x \in \{0, \ldots, x\} \) with \( y_{\overline{x}} = \overline{y} \).
2.1 Initialization

When initialized with a 1-difference sequence \( Y = (y_0, \ldots, y_{n-1}) \), the one-level FL structure first computes an array \( \text{Valley}[0 \ldots n] \) such that \( \text{Valley}[x] \) is the valley of \((x, y_x)\) for all \( x \in \{0, \ldots, n-1\} \) and \( \text{Valley}[n] \) has the artificial value \( n-1 \). Even if \( Y \) is a sequence of \( n \) arbitrary integers (or real numbers, given a suitably generalized definition of valleys), this can be done in \( O(n) \) time with a sweep over \( 0, \ldots, n-1 \) shown in Fig. 1. When the sweep is at some \( x \in \{0, \ldots, n-1\} \), its state is given by the sorted sequence of all valleys of points of the form \((x, y)\), where \( y \geq y_x \), with each valley \( \overline{x} \) represented by the triple \((\overline{x}, y_{\overline{x}}, \max\{y_i \mid \overline{x} \leq i \leq x\})\). The triples, whose components, in the order from left to right, are referred to using the field names \( x \), \( \text{low} \) and \( \text{high} \) in the code, are stored in order in an array \( S \), preceded by the dummy triple \((0, -\infty, \infty)\). \( S \) is manipulated as a stack, except that the sweep occasionally inspects the \( \text{high} \) component of the triple just below the top triple. The computation of valleys corresponds roughly to the decomposition of the given tree into paths in the algorithm of Bender and Farach-Colton [3].

\[
\begin{align*}
top & := 0; \quad (* \text{stack pointer of } S *) \\
S[top] & := (0, -\infty, \infty); \quad (* \text{dummy sentinel; will never be popped } *) \\
\text{for } x & := 0 \text{ to } n - 1 \text{ do } (* \text{sweep from left to right } *) \\
\text{while } S[top].\text{low} \geq y_x & \text{ do } top := top - 1; \quad (* \text{pop valleys no deeper } *) \\
\text{if } S[top].\text{high} \geq y_x & \text{ then } (* \text{cannot reach even first valley from } (x, y_x) *) \\
& \quad \text{Valley}[x] := x; (* \text{cannot go deeper from } (x, y_x) *) \\
\text{if } S[top].\text{high} > y_x & \text{ then } (* \text{a deeper valley can be reached from } (x, y_x) *) \\
& \quad \text{top} := top + 1; \quad (* \text{push } *) \\
& \quad S[top] := (x, y_x, y_x); (* x \text{ is a new valley } *) \\
\text{else } (* \text{a deeper valley can be reached from } (x, y_x) *) \\
\text{while } S[top - 1].\text{high} < y_x & \text{ do } top := top - 1; \quad (* \text{pop until before } \geq *) \\
& \quad \text{Valley}[x] := S[top].x; \quad (* \text{deepest reachable valley } *) \\
\text{if } S[top - 1].\text{high} > y_x & \text{ then } \text{S[top].high} := y_x; \quad (* \text{have to pass here } *) \\
\text{else } top := top - 1; \quad (* \text{top valley not deepest once sweep continues } *) \\
\text{Valley}[n] & := n - 1; \quad (* \text{special convention } *)
\end{align*}
\]

Fig. 1: The computation of the array \( \text{Valley} \) for a sequence \((y_0, \ldots, y_{n-1})\) of integers.

For every integer \( x \geq 1 \), let \( \pi(x) \) be the largest power of 2 that divides \( x \). The idea of using \( \pi \) in the solution of the FS or FL problem goes back to Berkman and Vishkin [4] and Ben-Amram [2], but it is crucial to our approach to use \( \pi \) in a different way. The one-level FL structure is parameterized by an integer constant \( \kappa \geq 3 \). Its initialization proceeds to compute two arrays \( \text{Weight}[0 \ldots n - 1] \) and \( \text{Jump}[0 \ldots n - 1] \) such that \( \text{Weight}[\overline{x}] = \{|x \in \{\overline{x}, \ldots, n - 1\} : \text{Valley}[x] = \overline{x}\| \) for all \( \overline{x} \in \{0, \ldots, n - 1\} \) and \( \text{Jump}[\hat{\overline{x}}] = \text{Valley}[\text{FL}(\hat{\overline{x}}, y_{\hat{\overline{x}}} + (\kappa - 2)\pi(\hat{\overline{x}}))] \) for all \( \hat{\overline{x}} \in \{1, \ldots, n - 1\} \), while \( \text{Jump}[0] \) is set to the artificial value 0. Following Bender and Farach-Colton [3], we define a ladder of height \( h \) located at \( x \), where \( h \) and \( x \) are integers with \( h \geq 0 \) and \( 0 \leq x < n \), to be an array with index set \( \{y_x + 1, \ldots, y_x + h\} \) that maps each \( y \in \{y_x + 1, \ldots, y_x + h\} \) to \( \text{FL}(x, y) \). The initialization of the structure is finished by equipping each \( x \in \{1, \ldots, n - 2\} \) with a ladder \( L_x \) located...
at $x$ and of height $\min\{\max\{\kappa - 1, \kappa’(\text{Weight}[x] - 1) - 2\}, y_{\max} - y_x\}$, where $\kappa’ = \lceil (2k + 2)/(\kappa - 2) \rceil$ and $y_{\max} = \max\{y_0, \ldots, y_{n-1}\}$, and each $x \in \{0, n-1\}$ with a ladder $L_x$ of height $y_{\max} - y_x$. This is easy to do in a second sweep over $0, \ldots, n-1$, this time from right to left. The complete initialization of the one-level FL structure for a 1-difference sequence $(y_0, \ldots, y_{n-1})$ is shown in Fig. 2, which assumes that the default value $\perp$ is chosen as $n$.

2.2 Processing of Queries

For $x \geq 1$, let $\|x\| = 2^{\lceil \log_2 x \rceil}$, i.e., $\|x\|$ is the largest power of 2 no larger than $x$. To answer a nontrivial query $(x, y)$, the one-level FL structure computes $t = y - y_x$ and, if $t < \kappa$, returns $L_x[y]$, which is obviously correct. If $t \geq \kappa$, it returns $L_{\text{Jump}[x]}[y]$, where $\hat{x}$ is the largest integer with $1 \leq \hat{x} \leq x$ and $\pi(\hat{x}) = \|t/\kappa\|$ if there is such an integer, and $\hat{x} = 0$ if not. This procedure, augmented with instructions to handle trivial queries, is shown in Fig. 3.

Fig. 3. The execution of a query $(x, y)$ in the one-level FL structure.
there to the “foot” \((\vec{x}, y_{\vec{x}})\) of the ladder at \(\vec{x} = \text{Jump}[\vec{x}]\). The red arrow “shunts out” the value \(x' = \text{FL}(\vec{x}, y_{\vec{x}}+ (\kappa-2)p)\), which is hinted at with dashed red arrows that pass via \((x', y_{x'})\). The ladder at \(\vec{x}\) is shown in green, and the entry consulted in the ladder and the information provided by the ladder are symbolized by two green arrows. A magenta cross, finally, marks the point \((\vec{x}, y_{\vec{x}})\), where \(\vec{x} = \text{FL}(x, y)\).

The ladders not used by the example queries are hinted at in pale green. Figs. 5 and 6 show, using similar drawing conventions, how the same queries are executed in data structures derived from the basic data structures of Ben-Amram [2] and Bender and Farach-Colton [3] by translating them to our setting and streamlining them where possible. The ladders of [2] are of total height \(\Theta(n \log n)\), and the jump tables of [3] (shown as columns of red or pale red dots in Fig. 6) hold a total of \(\Theta(n \log n)\) entries, which explains why these earlier data structures are less efficient.

Fig. 4: The execution of two example queries in the one-level FL structure with \(\kappa = 5\).

2.3 Correctness

Assume that the procedure of Fig. 3 is carried out for a nontrivial query \((x, y)\) and define \(p\) and \(\hat{x}\) as in the procedure. Let \(t = y - y_x\) and observe that \(\kappa p \leq t \leq 2\kappa p-1\) and that \(|x - \hat{x}| \leq 2p-1\). Since \(y - y_x = t > |x - \hat{x}|\), it is clear that no integer \(i\) with \(\hat{x} \leq i \leq x\) can have \(y_i \geq y\), so \(\text{FL}(x, y) = \text{FL}(\hat{x}, y)\) (informally, nothing blocks the sight between \((x, y)\) and \((\hat{x}, y)\)). If \(\hat{x} = 0\), we have \(\text{Jump}[\hat{x}] = 0\) (by the special convention regarding \(\text{Jump}[0]\)), the nontriviality of the query shows that \(y\) belongs to the index set \(\{y_0 + 1, \ldots, y_{\max}\}\) of \(L_0\), and the procedure correctly returns \(L_0[y]\). Assume from now on that \(\hat{x} > 0\), so that \(\pi(\hat{x}) = p\). We shall need the following bounds on \(y - y_{\hat{x}}\).

\[
y - y_{\hat{x}} = t + (y_x - y_{\hat{x}}) \geq t - |x - \hat{x}| \geq \kappa p - (2p - 1) = (\kappa - 2)p + 1 \quad \text{and} \quad y - y_{\hat{x}} = t + (y_x - y_{\hat{x}}) \leq t + |x - \hat{x}| \leq (2\kappa p - 1) + (2p - 1) = (2\kappa + 2)p - 2.
\]
Define $x' = FL(\hat{x}, y_{\hat{x}} + (\kappa - 2)p)$ and $\overline{x} = Jump[\hat{x}] = Valley[x']$. In order to demonstrate that the data structure operates correctly in the remaining cases, we must show that $y$ belongs to the index set of $L_\overline{x}$ and that $FL(x, y) = FL(\overline{x}, y)$.

Assume first that $x' = n (= \bot)$. Then we also have $FL(\hat{x}, y) = n$ (because $y \geq y_{\hat{x}} + (\kappa - 2)p$), $\overline{x} = n - 1$ (by the special convention regarding $Valley[n]$), $FL(\overline{x}, y) = n$ (because $\hat{x} \leq \overline{x}$), and $y > y_{\overline{x}}$ (because $y > y_i$ for $i = \hat{x}, \ldots, n - 1$). Since $L_{n-1}$ is of height $y_{\max} - y_{n-1}$, it is clear that $y$ belongs to the index set of $L_\overline{x}$ and that the query returns the correct value, namely $FL(\overline{x}, y) = FL(\hat{x}, y) = FL(x, y) = n$.

Assume from now on that $x' < n$ and therefore that $y_{x'} = y_{\overline{x}} + (\kappa - 2)p < y$.

Because $x' = FL(\hat{x}, y_{x'})$, $(\hat{x}, y_{\overline{x}})$ is down-left reachable from $(x', y_{x'})$. Since $\overline{x} = Valley[x']$, this shows that $y_{\overline{x}} \leq y_{x'}$. Another consequence of the relation $\overline{x} = Valley[x']$ is that for every $j \in J = \{y_{\overline{x}}, y_{\overline{x}} + 1, \ldots, y_{x'}\}$, the set $I_j = \{i \mid \overline{x} \leq i \leq x' \text{ and } y_i = j\}$ is nonempty and $Valley[\min I_j] = \overline{x}$. It follows that
Weight[\pi] \geq |J| = y_{x'} - y_\pi + 1. Since \kappa' \geq 1, we now find
\[
y \leq y_x + (2\kappa + 2)p - 2 \leq y_x + \kappa'(\kappa - 2)p - 2 = y_\pi + \kappa'(y_{x'} - y_\pi) - 2 \
\leq y_\pi + \kappa'(y_{x'} - y_\pi) - 2 \leq y_\pi + \kappa'(\text{Weight}[\pi] - 1) - 2.
\]
Because \(y > y_\pi \geq y\), this shows that \(y\) belongs to the index set of \(L_\pi\). Finally observe that the relation \(y_i \leq y_{x'} < y\) holds both for \(\pi \leq i \leq x'\) (because \(\pi = \text{Valley}[x']\)) and for \(\pi \leq i \leq x'\) (because \(x' = \text{FL}(\pi, y_{x'})\)). Thus \(y_i < y\) is satisfied for all integers \(i\) between \(\pi\) and \(\hat{x}\), inclusive, so \(\text{FL}(\pi, y) = \text{FL}(\hat{x}, y) = \text{FL}(x, y)\).

Therefore the query returns the correct result also in this final case.

### 2.4 Time and Space Requirements

The function \(x \mapsto \|x\|\) can be evaluated in constant time. E.g., this can be done by using the \texttt{bsr} instruction supported by modern CPUs or by lookup in tables that can be constructed in \(O(\sqrt{n})\) time and occupy \(O(\sqrt{n})\) words. Similarly, it is easy to compute \(\pi(x)\) for \(x = 1, \ldots, n - 1\) in average constant time per value by inspecting the bits in the binary representation of \(x\) in the order from right to left until a 1 is encountered. It is now obvious that the initialization of the 1-level FL structure takes \(O(n)\) time and that it answers every query in constant time. In addition to a small number of simple variables, the data structure must store the array \textit{Jump}, of \(n\) entries, and the ladders \(L_0, \ldots, L_{n-1}\). The two ladders \(L_0\) and \(L_{n-1}\) are of height at most \(y_{\max} - y_{\min} \leq n\) each, and the ladders \(L_1, \ldots, L_{n-2}\) are of total height at most \(\sum_{x=1}^{n-2} \max\{\kappa - 1, \kappa' (\text{Weight}[x] - 1) - 2\} \leq (\kappa - 1 + \kappa') n\), where the inequality follows from the fact that \(\sum_{x=0}^{n-1} \text{Weight}[x] = n\). The factor \(\kappa - 1 + \kappa'\) is a constant, for every fixed \(\kappa\), that takes on its minimum value of 8 for \(\kappa \in \{4, 5\}\). Thus it is clear that the data structure occupies \(O(n)\) words or \(O(n \log n)\) bits. During its construction \(O(n)\) additional words are needed for the arrays \(S[0 \ldots n], \text{Valley}[0 \ldots n], \text{Weight}[0 \ldots n-1]\) and \(\text{RightSight}[y_{\min} \ldots y_{\max} + 1]\). It is easy to reduce the space requirements of the finished data structure by a constant factor at the price of a somewhat higher (but still constant) query time. E.g., for all integers \(x\) and \(x'\) with \(0 \leq x \leq x' < n\) and all integers \(y\), \(\text{FL}(x, y) = \text{FL}(x', y)\) unless \(y_i \geq y\) for some \(i \in \{x, \ldots, x'\}\), so it is possible to do away with the bottom \(k\) entries of every except every \(\ell\)th ladder for arbitrary fixed positive integers \(k\) and \(\ell\) with \(k + \ell \leq \kappa\). Another possibility is to equip each element of \(\{1, \ldots, n - 1\}\) with two “jump values”, rather than one. We have reproved the following result of [4][6][13][2].

**Theorem 2.1.** Given a 1-difference sequence \(Y\) of length \(n\), a data structure that answers FL queries about \(Y\) in constant time and occupies \(O(n \log n)\) bits can be constructed in \(O(n)\) time. Given an \(n\)-node rooted tree \(T\), a data structure that answers LA queries about \(T\) in constant time and occupies \(O(n \log n)\) bits can be constructed in \(O(n)\) time.

### References

1. Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th International Colloquium on Automata, Languages and Program-
2. Amir M. Ben-Amram. The Euler path to static level-ancestors. *Computing Research Repository (CoRR)*, abs/0909.1030 [cs.DS], 2009. Available from: http://arxiv.org/abs/0909.1030.
3. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. Available from: https://doi.org/10.1016/j.tcs.2003.05.002.
4. Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994. Available from: https://doi.org/10.1016/S0022-0000(05)80002-9.
5. Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987. Available from: https://doi.org/10.1007/BF01840366.
6. Paul F. Dietz. Finding level-ancestors in dynamic trees. In *Proc. 2nd Workshop on Algorithms and Data Structures (WADS 1991)*, volume 519 of *LNCS*, pages 32–40. Springer, 1991. Available from: https://doi.org/10.1007/BFb0028247.
7. Torben Hagerup and Manfred Nowak. Recognition of spanning trees defined by graph searches, 1985. Tech. Rep. A 85/08, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany.
8. Udi Manber. Recognizing breadth-first search trees in linear time. *Inform. Process. Lett.*, 34(4):167–171, 1990. Available from: https://doi.org/10.1016/0020-0190(90)90155-Q.
9. Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.
10. Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. Available from: https://doi.org/10.1145/2601073.
11. Hao Yuan and Mikhail J. Atallah. Efficient data structures for range-aggregate queries on trees. In *Proc. 12th International Conference on Database Theory (ICDT 2009)*, pages 111–120. ACM, 2009. Available from: https://doi.org/10.1145/1514894.1514908.