Succinct Representations of Binary Trees for Range Minimum Queries

Pooya Davoodi[1]*, Rajeev Raman[2], and S. Srinivasa Rao[3]

[1] Polytechnic Institute of New York University, United States. E-mail: pooyadavoodi@gmail.com
[2] University of Leicester, United Kingdom. E-mail: r.raman@leicester.ac.uk
[3] Seoul National University, South Korea. E-mail: ssrao@cse.snu.ac.kr

Abstract. We provide two succinct representations of binary trees that can be used to represent the Cartesian tree of an array A of size n. Both the representations take the optimal $2n + o(n)$ bits of space in the worst case and support range minimum queries (RMQs) in $O(1)$ time. The first one is a modification of the representation of Farzan and Munro (SWAT 2008); a consequence of this result is that we can represent the Cartesian tree of a random permutation in $1.92n + o(n)$ bits in expectation. The second one uses a well-known transformation between binary trees and ordinal trees, and ordinal tree operations to effect operations on the Cartesian tree. This provides an alternative, and more natural, way to view the 2D-Min-Heap of Fischer and Huen (SICOMP 2011). Furthermore, we show that the pre-processing needed to output the data structure can be performed in linear time using $o(n)$ bits of extra working space, improving the result of Fischer and Heun who use $n + o(n)$ bits working space.

1 Introduction

Given an array $A[1 \cdots n]$ of totally ordered values, the *range minimum query* (RMQ) problem is to preprocess A into a data structure such that given two indexes $1 \leq i \leq j \leq n$, we return the index of the minimum value in $A[i \cdots j]$; the aim is to minimize the time and space requirements of both the preprocessing and the data structure. This problem finds a variety of applications that deal with huge datasets, thus highly space-efficient solutions are of great interest. We consider the problem in the word RAM model with word size $\Theta(\log n)$ bits.

A standard approach to solve the RMQ problem is to use the *Cartesian tree* [20]. The Cartesian tree of an array $A[1 \cdots n]$ is a binary tree with nodes labeled by the indexes of A. The root has label i, where $A[i]$ is the minimum element in A. The left subtree of the root is the Cartesian tree of the subarray $A[1 \cdots i-1]$, and the right subtree of the root is the Cartesian tree of the

* Part of this research was done while the first author was a PhD student at MADALGO (supported by the Danish National Research Foundation), Aarhus University, Denmark.

subarray $A[i+1 \cdots n]$. Thus the answer for the query range $[i \cdots j]$ is the label of the lowest common ancestor (LCA) of the nodes labeled by i and j. The Cartesian tree of A can be constructed in $O(n)$ time [6]. A data structure that uses $O(n)$ words of space and finds the LCA of two nodes in a tree of size n in $O(1)$ time can be constructed in $O(n)$ time [10]—an apparently optimal solution.

In fact, the Cartesian tree of an array A completely characterizes A with respect to RMQs: the Cartesian tree of two arrays have different topology, iff there exists at least one query that has different answers over the two arrays. Since the information-theoretic lower bound for representing a binary tree on n nodes is $2n - \Theta(\log n)$ bits, there is a significant gap between this lower bound and the space usage of [6], which is $O(n)$ words, or $O(n \log n)$ bits. A fair amount of effort has gone into closing this gap, particularly since in several applications the array A need not be kept once we have enough information to answer RMQs.

It is known that binary trees can be represented *succinctly*, i.e. using space within a lower-order term of the information-theoretic lower bound. Specifically, an n-node binary tree can be represented in $2n+o(n)$ bits to support a number of operations, including LCA, in $O(1)$ time [12, 1–3]. Unfortunately, we cannot use these representations to solve the RMQ problem. The difficulty is that the label of a node in the Cartesian tree (the index of the corresponding array-element) is its rank in the *inorder* traversal of the Cartesian tree. However, succinct tree representations cannot label nodes in an arbitrary manner without blowing up the space usage, and support only a few numbering schemes, including level-order [12], preorder [3] and others, but *not* inorder.

In parallel, many succinct representations of *ordinal* trees (arbitrary rooted trees where the order of children matters) were developed. These take $2n + o(n)$ bits to represent n-node ordinal trees, and support a wide variety of operations including LCA queries (see e.g. [11, 2, 19]). However, ordinal trees do not distinguish between left and right children in a binary tree, so the structure of the Cartesian tree cannot be represented. Also, as above, we need to find a way to translate between the array indexes and the numbering scheme of the ordinal tree representation. To get around these problems, Sadakane [18] adds a new leaf to each node in a Cartesian tree of n nodes, and views the resulting tree of $n' = 2n$ nodes as an ordinal tree. He notes that array index i corresponds to the i-th leaf in the ordinal tree in preorder. Representing this ordinal tree succinctly, he answers RMQs in $O(1)$ time, but the space usage is $2n' + o(n') = 4n + o(n)$ bits—twice the optimal. A solution using $2n + o(n)$ bits that answers RMQs in $O(1)$ time was proposed by Fischer [4, 5], who defined the *2D-Min-Heap* of an array $A[1 \cdots n]$, an ordinal tree with $n+1$ nodes that stores information on prefix minima of sub-arrays of A, and represents this ordinal tree succinctly.

Our Results. We present two different techniques to represent Cartesian trees using $2n + o(n)$ bits, and both the representations support the inorder numbering scheme and LCA queries. An immediate consequence of each of these representations is a data structure of size $2n + o(n)$ bits that supports RMQs in $O(1)$ time. Although we do not improve the upper bounds of [5] for the RMQ problem in the worst case (as they were already optimal), we introduce more natural ways or simpler approaches to solve the RMQ problem.

In Section 2, we provide a new representation of binary trees that is a modification of the representation of [2], and to support the operations of converting between its numbering system and inorder numbering (so-called *node-rank*$_{\text{inorder}}$ and *node-select*$_{\text{inorder}}$) in $O(1)$ time. A consequence of this result is that we can represent the Cartesian tree of a *random* permutation of A in $1.92n + o(n)$ bits in expectation [9], and perform RMQs in $O(1)$ time.

In Section 3, we recall that there is a well-known transformation between binary trees and ordinal trees, which essentially converts inorder numbers in the binary tree to preoreder/postoreder numbers in the ordinal tree, and preserves the preorder/postorder numbers of the binary tree. Using this, we show another method to represent Cartesian trees: transform a Cartesian tree (a binary tree) into an ordinal tree, and then represent the ordinal tree. Using ordinal tree operations on the resulting tree, it is possible to represent the Cartesian tree in optimal space and perform RMQs in constant time. This provides an alternative and more natural way to view the 2D-Min-Heap of [5], as essentially the result of the above transformation of the Cartesian trees into ordinal trees. We also observe a connection between the above transformation and Jacobson's binary tree representation [12].

Finally, in Section 4, we show that constructing the data structure of Section 3 (outputting the data structure given an input array) can be done in linear time using only $O(\sqrt{n}\log n)$ bits of space, "improving" the result of [5] where $n + o(n)$ working space is used (the accounting of space is slightly different). This improvement is useful if the preprocessing and subsequent deployment of the data structure are done on the same machine.

Preliminaries. Given a bit vector, *rank*(i) returns the number of 1s up to the position i in the bit vector, and *select*(i) returns the position of the ith 1 in the bit vector. We use the following succinct representations of bit vectors.

Lemma 1. **[16]** *Given a bit vector of size m with n 1s, one can construct*

(a) an indexable dictionary *that uses* $\log\binom{m}{n} + o(n) + O(\log\log m)$ *bits, and supports rank, for only those positions where there is a 1 in the bit vector, and select queries in constant time, and*
(b) a fully indexable dictionary *that uses* $\log\binom{m}{n} + o(m)$ *bits, and supports rank and select queries in constant time.*

Given a sequence of balanced parentheses, we define the following operations: *find-close*(i) returns the position of the closing parenthesis that matches the open parenthesis at position i of the sequence (the *find-open*(i) opreation is analogous); *excess*(i) returns the difference between the number of open and closing parentheses from the beginning of the sequence up to position i. The operation *double-enclose*(i, j) returns the position of the pair of matching parentheses that tightly encloses two non-overlapping pairs of parentheses whose open parentheses respectively appear at positions i and j in the sequence. It is known that in an ordinal tree represented by its *balanced parenthesis representation (BP)*, the LCA of two nodes, whose open (closing) parentheses are in positions i and j, is equivalent to *double-enclose*(i, j) [14].

Lemma 2. [14, 17, 13] *Given a sequence of balanced parentheses of size n, there exists a data structure of size $n + o(n)$ bits that supports the operations* $\text{rank}_($*,* $\text{select}_($, $\text{rank}_)$, $\text{select}_)$*,* find-close*,* find-open*,* excess*, and* double-enclose *operations on the sequence all in $O(1)$ time.*

2 Representation Based on Tree Decomposition

We show a succinct representation of binary trees that supports multiple numberings (preorder, postorder, DFUDS order and inorder) on the nodes of the tree, plus a comprehensive list of operations suggested by [11, 2]. This data structure is essentially the same as the k-ary (cardinal) tree representation of Farzan and Munro [2] for the case when $k = 2$, with the additional support for two more operations, $\text{node-rank}_{\text{inorder}}$ and $\text{node-select}_{\text{inorder}}$. The first operation returns the inorder number of a node given its preorder number, and the second operation performs the inverse. We use the preorder numbers of the nodes to refer to them. Since we can support the *node-rank* and *node-select* operations with respect to inorder, postorder and DFUDS order, we can also use the numberings of the nodes in any of these three orders to refer to them in the operations.

We begin by outlining the succinct representation of Farzan and Munro [2]. Like the representations of [8, 11, 15], the representation of Farzan and Munro recursively decomposes the tree into sub-trees. A prominent property of their decomposition method is that each sub-tree, aside from its root, has at most one *boundary node* that connects the sub-tree to other sub-trees, and furthermore the boundary node has at most one child outside of the sub-tree. The following lemma states the result of the decomposition:

Lemma 3. [2, Theorem 1] *A tree with n nodes can be decomposed into $\Theta(n/L)$ subtrees, each of size at most L. The subtrees are disjoint aside from their roots. Moreover, aside from edges leaving root of subtrees, there is at most one edge in each subtree that connects a node of the subtree to its child in another subtree.*

The ordinal tree is first decomposed using Lemma 3 into $O(n/\log^2 n)$ *mini-trees* each of size $O(\log^2 n)$. Each mini-tree is further decomposed into $O(\log n)$ *micro-trees* of size at most $\lceil \frac{\log n}{2} \rceil$. Each micro-tree is represented with its size, and an index to a lookup table of size $o(n)$ bits, which stores answers of all queries asked within the micro-trees. The sum of the sizes of the representations of all the micro-trees (i.e., the total space for storing all the indexes to the lookup table) is $2n + o(n)$ bits in total, which is the dominating part of the space. Each mini-tree is represented by the explicitly stored list of pointers between the micro-trees within the mini-tree, where each pointer uses only $O(\log \log n)$ bits. The roots of micro-trees are represented using indexable dictionary structure of Lemma 1(a). The original tree that contains the mini-trees is represented analogously to the representation of mini-trees, by storing the list of pointers between the mini-trees. All the parts together take $2n+o(n)$ bits. The data structure can support the full set of navigational operations and queries in binary trees.

Lemma 4. [**2**] *A binary tree with n nodes can be represented using $2n+o(n)$ bits of space, while a full set of operations in [2, Table 2] including* LCA *can be supported in $O(1)$ time.*

We now show the main theorem of this section:

Theorem 1. *Given a binary tree with n nodes, there exists a data structure of size $2n+o(n)$ bits, that supports the following operations in $O(1)$ time:* node-rank$_{\text{inorder}}$*,* node-select$_{\text{inorder}}$*, and all the operations supported by the ordinal tree representation of Farzan and Munro [2] for Cardinal trees, including* LCA*.*

Proof. As mentioned above, in our data structure, to perform any operation on a node, except node-select operations, we need to give the preorder number of the node to the operation, and the operation also returns a node in the form of its preorder number (this is not the case if the operation does not return a node at all such as depth). Thus, in circumstances in which we are asked to perform an operation on a node referred to by its inorder number, we first need to compute the preorder number of the node and then perform the operation as usual. Similarly, when we are asked to return the result of an operation in the form of an inorder number, we need to compute the inorder number of the node from the preorder number returned by the operation. These two tasks are performed by the operations node-select$_{\text{inorder}}$ and node-rank$_{\text{inorder}}$ respectively.

node-rank$_{\text{inorder}}$. For a node v, we want to compute the inorder number of v, given its preorder number. We count the following; c_1: the number of nodes that are visited before v in inorder traversal but visited after v in preorder traversal; c_2: the number of nodes that are visited after v in inorder traversal but visited before v in preorder traversal. It is not hard to see that the inorder number of v is equal to its preorder number $+ c_1 - c_2$.

The nodes counted in c_1 are all the nodes located in the left subtree of v, which can be counted by subtree size of the left child of v. The nodes counted in c_2 are all the ancestors of v of which left child is on the v-to-root path. We compute c_2 in a way similar to computing the depth of a node as follows. At the root of each mini-tree, we store c_2 of that root, which requires $O((n/\log^2 n)\log n) = o(n)$ bits. At the root r_μ of each micro-tree, we store the local-c_2 of r_μ, that is, the number of ancestors of r_μ, only up to the root of the mini-tree containing r_μ, where their left child is on the r_μ-to-root path. The local-c_2 of v is analogously defined for the ancestors of v within its micro-tree, which can be computed using table lookup. To calculate c_2 of v, we clearly take the sum of the following: c_2 of the root of the mini-tree containing v, local-c_2 of the root of the micro-tree containing v, and local-c_2 of v, all computed in $O(1)$ time.

node-select$_{\text{inorder}}$. For a node v, we want to compute the preorder number of v, given its inorder number. Notice that a node that is visited before v in preorder traversal is the root r_m of the mini-tree containing v. The preorder number of v can be expressed as the sum of two quantities: (1) preorder number of r_m; and (2) the number of nodes that are visited after r_m and before v in preorder traversal, which may include nodes both within and outside the mini-tree. In the following, we explain how to compute these two quantities.

(1) The preorder number of r_m is stored with the mini-tree representation, and thus we only need to find the mini-tree containing the node v. We number all the mini-trees in some arbitrary order, counting from zero up to $n_m - 1$, where $n_m = O(n/\log^2 n)$ is the number of mini-trees. We call these numbers, *names* of the mini-trees. Starting with an empty bit vector A, we traverse the tree in inorder, and after visiting each new node, we append a bit to A as follows: during the traversal when we enter a mini-tree from another mini-tree, we append a 1 to A, and while we are traversing within a mini-tree we append a 0 to A. During the traversal when we enter a mini-tree from another mini-tree, we also write down the name of the current mini-tree in another array B. At the beginning of A we write 1 corresponding to the first visited node (the root), and at the beginning of B, we write the name of the first visited mini-tree (containing the root). Thus, at the end of the traversal, A is a bit vector of length n. We observe that the i-th node in the inorder traversal of the tree belongs to the mini-tree with name $B[j]$ where j to be the number of 1s before $A[i + 1]$ (i.e., $j = rank_A(i + 1)$).

We store B explicitly as it only requires $O(n_m \cdot \log n) = o(n)$ bits due to the following. The length of B is at most $2 \cdot n_m$ because the traversal can enter a mini-tree at most two times (each mini-tree has at most one edge leaving the mini-tree aside from its root; see Lemma 3), and thus its name can be written in B at most two times. For the same reason, the number of 1s in A is at most $2 \cdot n_m$. We represent A using the FID structure of Lemma 1(b) which uses $O(\log \binom{n}{n_m}) = o(n)$ bits and supports rank operation on A in constant time.

(2) The number of nodes that are visited after r_m and before v in preorder, is computed by taking the sum of the following quantities: (i) the number of such nodes that are outside the mini-tree; (ii) the number of such nodes that are within the micro-tree t_μ containing v; and (iii) the number of such nodes that are within the mini-tree and outside t_μ (visited after r_m and before the root of t_μ). To compute these three quantities, we first need to find t_μ among the other micro-trees within the same mini-tree. We utilize the same method as we used in (1) as follows Each mini-tree plays the role of the original tree in (1) and its micro-trees play the role of mini-trees in (1). That is, we give a name to each micro-tree in the mini-tree; we traverse the mini-tree in inorder; we make the arrays A and B; and we use an FID to encode A. Applying the same analysis provides $o(n)$ bits space.

The nodes in (i) only exist if the mini-tree has a boundary node which is visited before the root of t_μ. The nodes in (i) are in fact all the nodes in a subtree of such a boundary node, and thus the subtree size of the child of the boundary node which is outside of the mini-tree determines the quantity in (i).

The quantity in (ii) is computed using table lookup. The number in (iii) is the local-preorder number of the root of t_μ. We store the local-preorder number of the root of each micro-tree in $O(\log\log n)$ bits which requires $o(n)$ bits in total. This completes the proof of Theorem 1. $\square$

The following theorem gives a slight generalization of Theorem 1, which uses entropy coding to exploit any differences in frequency between the four node types (Theorem 1 corresponds to choosing all the α_is to be 1/4):

Theorem 2. *For any positive constants $\alpha_0, \alpha_L, \alpha_R$ and α_2, such that $\alpha_0 + \alpha_L + \alpha_R + \alpha_2 = 1$, a binary tree with n_0 leaves, n_L (n_R) nodes with only a left (right) child and n_2 nodes with both children can be represented using $\left(\sum_{i\in\{0,L,R,2\}} n_i \log_2 1/\alpha_i\right) + o(n)$ bits of space, while a full set of operations [2, Table 2] including* LCA *can be supported in $O(1)$ time.*

Proof. We proceed as in the proof of Theorem 1, but if $\alpha = \min_{i\in\{0,L,R,2\}} \alpha_i$, we choose the size of the micro-trees to be at most $\mu = \frac{\log n}{2\log_2(1/\alpha)}$. Then, given a micro-tree with μ_i nodes of type i, for $i \in \{0, L, R, 2\}$ we encode it by writing the node types in level order (cf. [12]) and encoding this string using arithmetic coding with the probability of a node of type i taken to be α_i. The size of this micro tree is $\left\lceil \sum_{i\in\{0,L,R,2\}} \mu_i \log_2 1/\alpha_i \right\rceil$, from which the theorem follows. $\square$

Corollary 1. *If A is a random permutation over $\{1, \ldots, n\}$, then RMQ queries on A can be answered using $1.92n + o(n)$ bits in expectation.*

Proof. Choose $\alpha_0 = \alpha_2 = 1/3$ and $\alpha_R = \alpha_L = 1/6$. The claim follows from [9, Theorem 1]. $\square$

3 Transforming Binary Trees into Ordinal Trees

We now give a succinct representation of binary trees based upon a well-known transformation between binary trees and ordinal trees. We show that this transformation not only supports inorder numbering, but also permits navigational and LCA operations by using the relevant operations on the ordinal tree.

Theorem 3. *A binary tree on n nodes can be represented in $2n + o(n)$ bits to support* left-child, right-child, parent, subtree-size *and* LCA *in $O(1)$ time, where the nodes are referred to by any of the inorder, preorder, or postorder numbers.*

Proof. We first describe two (related) transformations between binary trees and ordinal trees, and then describe how binary tree operations can be performed. Let t_b be a binary tree with n nodes that we want to transform to an ordinal tree, and let t_1 and t_2 be the ordinal trees resulting from the first and second transformation respectively. Each of t_1 and t_2 has $n+1$ nodes, where each node corresponds to a node in t_b, except the root which is dummy. In the first transformation, the root of t_b corresponds to the first child of the dummy root of t_1;

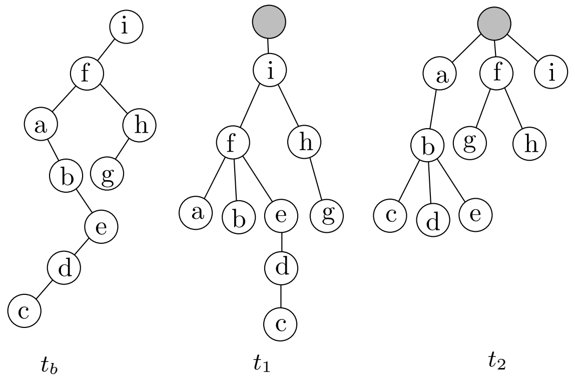

Fig. 1. An example for the transformations: t_b is a binary tree, t_1 and t_2 are the ordinal trees obtained by applying the first and second transformations respectively to t_b. The gray nodes are dummy and do not correspond to any node in t_b.

the left child of a node in t_b corresponds to the first child of the corresponding node in t_1; and the right child of a node in t_b corresponds to the next sibling of the corresponding node in t_1. In the second transformation, the root of t_b corresponds to the last child of the dummy root of t_2; the left child of a node in t_b corresponds to the previous sibling of the corresponding node in t_2; and the right child of a node in t_b corresponds to the last child of the corresponding node in t_2 (see Fig. 1).

These two transformations have a useful property which allows us to use the inorder number as the interface of the operations. In the first transformation, the inorder number of a node in t_b is equal to the postorder number of its corresponding node in t_1. In the second transformation, the inorder number of a node in t_b is equal to the preorder number of its corresponding node in t_2. Furthermore, the preorder number of a node in t_b is equal to the preorder number of its corresponding node in t_1, and the postorder number of a node in t_b is equal to the postorder number of its corresponding node in t_2.

The first and second transformations can be modified to make a third and fourth transformation, respectively, by reversing the order of all siblings in the ordinal tree. That is, if some node is the ith child out of the k children of its parent in the ordinal tree, then in the reverse order, it will be the $k - i + 1$th child of its parent (the new tree can be seen as the mirror image of the original ordinal tree). The third transformation is used in Section 4.

Taking advantage of the transformations and using the known ordinal tree representations that use preorder or postorder numbers as the interface of the operations, we obtain binary trees representations that use the inorder numbers as the interface of the operations. To represent a binary tree, we transform it into an ordinal tree using either of the transformations, and then we represent the ordinal tree by utilizing one of the known succinct representations that supports at least the operations ith-child, parent, next-sibling, previous-sibling, subtree-size, leftmost leaf, rightmost leaf, LCA, level-ancestor, and depth. In the following, we only show how to support the operations in the binary tree using the first transformation. Supporting the operations on the second transformation is analogous. Given a node v in a binary tree t_b, let v_{t_1} denote the corresponding node in t_1, the transformed binary tree using the first transformation.

left-child, right-child, parent. The left child of a node v in t_b is the first child of the node v_{t_1}, which can be determined using the operation ith-child on t_1. The right child of a node v in t_b is the next sibling of v_{t_1}, which can be determined by the operation next-sibling on t_1. For the parent of a node v in t_b there are two cases: 1) if v_{t_1} is the first child of its parent, then the answer is the parent; 2) if v_{t_1} is not the first child of its parent, then the answer is the previous sibling of v_{t_1}. These also can be determined using the operations ith-child, parent, and previous-sibling.

subtree-size. It is not difficult to see that the subtree size of v is equal to the sum of the subtree size of v_{t_1} and the subtree sizes of all the siblings to the right of v_{t_1}. Let ℓ be the right-most leaf in the subtree of the parent of v_{t_1}. To obtain the above sum, we only subtract the preorder number of v_{t_1} from the preorder number of ℓ.

LCA. Let w be the LCA of two nodes u and v in t_b that we want to compute. Notice that the LCA of u_{t_1} and v_{t_1} is a node z_{t_1}, that is a child of w_{t_1} and an ancestor of u_{t_1}, assuming that u is to the left of v in t_b. Thus, we only need to find the ancestor of u_{t_1} at level i, where $i-1$ is the depth of z_{t_1}. To compute this, we utilize the operations LCA, depth, and level-ancestor on t_1. □

We now observe an interesting connection between the above transformation and the binary tree representation of Jacobson [12]. Given a binary tree, we first add external nodes wherever there is a missing child, and label the internal nodes with an open parenthesis, and the external nodes with a closing parenthesis. We then traverse the tree in preorder and write down the labels of the nodes visited in the traversal order (this is similar to Jacobson's [12] encoding, except that he visits the tree in level-order). If the original tree has n nodes, the sequence so obtained has length $2n+1$ (as $n+1$ external nodes are added to the tree). It is easy to show that by adding an extra open parenthesis at the beginning, we get a balanced parenthesis sequence S of length $2n+2$. See Fig. 2 for an example. Note that in the depth-first search, if we switch the order in which the children of a node are visited (i.e., visit the right child before the left child), then the resulting sequence obtained is the balanced parenthesis sequence of the tree obtained by applying the first transformation to the given binary tree.

Furthermore, each open parenthesis in S, except the extra parenthesis that is added, and its matching closing parenthesis, are (conceptually) associated with a node in the given tree that was visited when the parenthesis is added to the sequence. It is easy to verify that the open parentheses in S from left to right correspond to the nodes in preorder, and the closing parentheses from left to right correspond to the nodes in inorder.

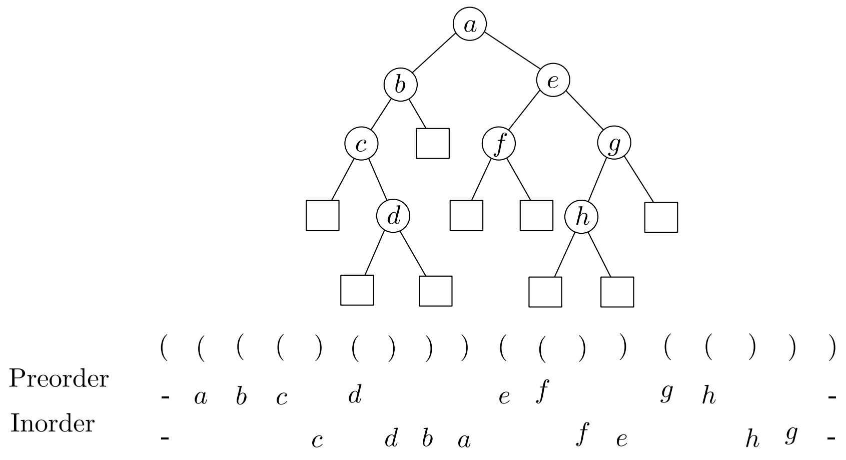

Fig. 2. Illustrating the connection between Jacobson's approach to representing binary trees and the representation of Theorem 3 to a binary tree. Note that the open/closing parentheses from left to right are in the same order as a preorder/inorder traversal of the nodes respectively.

4 Cartesian Tree Construction in $o(n)$ Working Space

We show how to construct the succinct representation of Section 3, using only $o(n)$ bits during the construction. A straightforward way to construct a succinct representation of a Cartesian tree is to construct the standard pointer-based representation of the Cartesian tree from the given array in linear time [6], and then construct the succinct representation using the pointer-based representation. The drawback of this approach is that the space used during the construction is $O(n \log n)$ bits, although the final structure uses only $O(n)$ bits. Fischer and Heun [5] show that the construction space can be reduced to $n + o(n)$ bits. In this section, we show how to improve the construction space to $o(n)$ bits.

Theorem 4. *Given an array A of n values, we can build a $2n + o(n)$-bit representation of its Cartesian tree in $O(n)$ time using $o(n)$ bits of auxiliary space.*

Proof. The proof assumes that the array A is present in read-only memory and it is possible to randomly access A. The algorithm reads A from left to right, and outputs a parenthesis sequence as follows: if, having completed the preprocessing for $A[1], \ldots, A[i]$, for some $i \geq 0$, when processing the $A[i+1]$, we compare $A[i+1]$ with all the suffix minima of $A[1..i]$—if $A[i+1]$ is smaller than $j \geq 0$ suffix minima, then we output the string $)^j($. This is so far a restatement of the algorithm of [6] for constructing a Cartesian tree, and it is not hard to see that the string output is balanced, by adding j closing parentheses to the end, where j is number of suffix minima of $A[1..n]$. This sequence is in fact the reverse of the DFUDS sequence of the ordinal tree obtained by applying the third transformation of Section 3 to the Cartesian tree. While the straightforward approach would be to maintain a linked list of the locations of the current suffix minima, this list could contain $\Theta(n)$ locations and could take $\Theta(n \log n)$ bits.

Our approach is to use the output string itself to encode the positions of the suffix minima. It is not hard to see that if the output string is created by the above process, it will be of the form $b_0(b_1(...(b_k($ where each b_i is a (possibly empty) maximal balanced parenthesis string – the remaining parentheses are called *unmatched*. It is not hard to see that the unmatched parentheses encode the positions of the suffix minima in the sense that if the unmatched parentheses are the $i_1, i_2 \ldots, i_k$-th opening parentheses in the current output sequence then the position $i_1, \ldots, i_k$ are precisely the suffix minima positions. Our task is therefore to sequentially access the next unmatched parenthesis, starting from the end, when adding the new element $A[i+1]$. We conceptually break the string into blocks of size $\lfloor \sqrt{n} \rfloor$. For each block that contains at least one unmatched parenthesis, store the following info:

- it's block number (in the original paren string) and the total number of open parenthesis in the current output string before the start of the block.
- the position p of the rightmost paren in the block, and the number of open parentheses before it in the block.
- a pointer to the next block with at least one unmatched parenthesis.

This takes $O(\log n)$ bits per block, which is $O(\sqrt{n} \log n)$ bits.

- For the rightmost block (in which we add the new parens), keep positions of all the unmatched parens: the space for this is also $O(\sqrt{n} \log n)$ bits.

When we process the next element of A, we compare it with unmatched parens in the rightmost block, which takes $O(1)$ time per unmatched paren that we compared the new element with, as in the algorithm of [6]. Updating the last block is also trivial. Suppose we have compared $A[i+1]$ and found it smaller than all suffix maxima in the rightmost block. Then, using the linked list, we find the rightmost unmatched paren (say at position p) in the next block in the list, which takes $O(1)$ time, and compare with it (this is also $O(1)$ time). If $A[i+1]$ is smaller, then sequentially scan this block leftwards starting at position p, skipping over a maximal BP sequence to find the next unmatched paren in that block. The time for this sequential scan is $O(n)$ overall, since we never sequentially scan the same paren twice. Updating the blocks is straightforward. Thus, the creation of the output string can be done in linear time using $O(\sqrt{n} \log n)$ bits. For constructing the auxiliary structures for the DFUDS in linear time see [7]. $\square$

References

1. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
2. A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th Scandinavian Workshop on Algorithm Theory*, pages 173–184. Springer-Verlag, 2008.

3. A. Farzan, R. Raman, and S. S. Rao. Universal succinct representations of trees? In *Proc. 36th International Colloquium on Automata, Languages and Programming*, pages 451–462. Springer, 2009.
4. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *LNCS*, pages 158–169. Springer-Verlag, 2010.
5. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
6. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th annual ACM symposium on Theory of computing*, pages 135–143. ACM Press, 1984.
7. R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.
8. R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.
9. M. J. Golin, J. Iacono, D. Krizanc, R. Raman, and S. S. Rao. Encoding 2D range maximum queries. In *Proc. ISAAC 2011*, volume 7074 of *LNCS*, pages 180–189. Springer-Verlag, 2011.
10. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
11. M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th International Colloquium on Automata, Languages and Programming*, pages 509–520. Springer-Verlag, 2007.
12. G. Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1989.
13. H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4(3), 2008.
14. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
15. J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 529–536. SIAM, 2001.
16. R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
17. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Symposium on Discrete Algorithms*, pages 225–232, 2002.
18. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
19. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 134–149. SIAM, 2010.
20. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.