Succinct Ordinal Trees Based on Tree Covering

Meng He¹, J. Ian Munro¹, and S. Srinivasa Rao²

¹ Cheriton School of Computer Science, University of Waterloo, Canada, {mhe, imunro}@uwaterloo.ca
² Computational Logic and Algorithms Group, IT University of Copenhagen, Denmark, mrao@itu.dk

Abstract. Various methods have been used to represent a tree of \( n \) nodes in essentially the information-theoretic minimum space while supporting various navigational operations in constant time, but different representations usually support different operations. Our main contribution is a succinct representation of ordinal trees, based on that of Geary et al. [7], that supports all the navigational operations supported by various succinct tree representations while requiring only \( 2n + o(n) \) bits. It also supports efficient level-order traversal, a useful ordering previously supported only with a very limited set of operations [8].

Our second contribution expands on the notion of a single succinct representation supporting more than one traversal ordering, by showing that our method supports two other encoding schemes as abstract data types. In particular, it supports extracting a word (\( O(\lg n) \) bits) of the balanced parenthesis sequence [11] or depth first unary degree sequence [3, 4] in \( O(f(n)) \) time, using at most \( n/f(n) + o(n) \) additional bits, for any \( f(n) \) in \( O(\lg n) \) and \( \Omega(1) \).

1 Introduction

Trees are fundamental data structures in computer science. There are essentially two forms. An ordinal tree is a rooted tree in which the children of a node are ordered and specified by their rank, while in a cardinal tree of degree \( k \), each child of a node is identified by a unique number from the set \([k]\). In this paper, we mainly consider ordinal trees.

The straightforward representation of trees explicitly associates with each node the pointers to its children. Thus, an ordinal tree of \( n \) nodes is represented by \( \Theta(n \lg n) \) bits. This representation allows straightforward, efficient parent-to-child navigation in trees. However, as current applications often consider very large trees, such a representation often occupies too much space.

To solve this problem, various methods have been proposed to encode an ordinal tree of \( n \) nodes in \( 2n + o(n) \) bits, which is close to the information-theoretic minimum (as there are \( \binom{2n}{n} / (n + 1) \) different ordinal trees), while supporting various navigational operations efficiently. These representations are based on various traversal orders of the nodes in the tree: preorder in which each node is visited before its descendants, postorder in which each node is visited after its descendants, and DFUDS order in which all the children of a node are visited.
before its other descendants [3, 4]. However, different representations of trees usually support different sets of navigational operations. A new problem is to design a succinct representation that supports all the navigational operations of various succinct tree structures. We consider the following operations:

- \text{child}(x, i)$: the $i$th child of node $x$ for $i \geq 1$;
- \text{child rank}(x)$: the number of left siblings of node $x$ plus 1;
- \text{depth}(x)$: the depth of $x$, i.e., the number of edges in the rooted path to node $x$;
- \text{level anc}(x, i)$: the $i$th ancestor of node $x$ for $i \geq 0$ (given a node $x$ at depth $d$, its $i$th ancestor is the ancestor of $x$ at depth $d - i$);
- \text{ndesc}(x)$: the number of descendants of node $x$;
- \text{degree}(x)$: the degree of node $x$, i.e., the number of its children;
- \text{height}(x)$: the height of the subtree rooted at node $x$;
- \text{LCA}(x, y)$: the lowest common ancestor of nodes $x$ and $y$;
- \text{distance}(x, y)$: the number of edges of the shortest path between nodes $x$ and $y$;
- \text{leftmost leaf}(x)$ (\text{rightmost leaf}(x))$: the leftmost (or rightmost) leaf of the subtree rooted at node $x$;
- \text{leaf rank}(x)$: the number of leaves before node $x$ in preorder plus 1;
- \text{leaf select}(i)$: the $i$th leaf among all the leaves from left to right;
- \text{leaf size}(x)$: the number of leaves of the subtree rooted at node $x$;
- \text{node rank}_{\text{pre/post/dfs}}(x)$: the position of node $x$ in the preorder, postorder or DFS order traversal of the tree;
- \text{node select}_{\text{pre/post/dfs}}(r)$: the $r$th node in the preorder, postorder or DFS order traversal of the tree;
- \text{level leftmost}(i)$ (\text{level rightmost}(i))$: the first (or last) node visited in a preorder traversal among all the nodes whose depths are $i$;
- \text{level succ}(x)$ (\text{level pred}(x))$: the level successor (or predecessor) of node $x$, i.e., the node visited immediately after (or before) node $x$ in a preorder traversal among all the nodes whose depths are equal to \text{depth}(x)$.

### 1.1 Related Work

Jacobson’s succinct tree representation [8] was based on the level order unary degree sequence (LOUDS) of a tree, which lists the nodes in a level-order traversal of the tree and encode their degrees in unary. With this, Jacobson [8] encoded an ordinal tree in $2n + o(n)$ bits to support the selection of the first child, the next sibling, and the parent of a given node in $O(\lg n)$ time under the bit probe model. Clark and Munro [6] further showed how to support the above operations in $O(1)$ time under the word RAM model with $\Theta(\lg n)$ word size.

As LOUT only supports a very limited set of operations, various researchers have proposed different ways to represent ordinal trees using $2n + o(n)$ bits. There are three main approaches based on (see Table 1 for a complete list of operations that each of them supports):

- Balanced parenthesis sequence (BP) [11]. The BP sequence of a given tree can be obtained by performing a depth-first traversal, and outputting an

---

3 The ordering puts the root first, then all of its children, from left to right, followed by all the nodes at each successive level (depth).

4 We use $\lg n$ to denote $\lceil \log_2 n \rceil$. 
opening parenthesis each time a node is visited, and a closing parenthesis immediately after all its descendants are visited.

– Depth first unary degree sequence (DFUDS) [3, 4]. The DFUDS sequence represents a node of degree $d$ by $d$ opening parentheses followed by a closing parenthesis. All the nodes are listed in preorder (an extra opening parenthesis is added to the beginning of the sequence), and each node is numbered by its opening parenthesis in its parent’s description (DFUDS number).

– Tree covering TC [7]. There is an algorithm to cover an ordinal tree with a set of mini-trees, each of which is further covered by a set of micro-trees.

### 1.2 Our Results

Our first result is to extend the succinct ordinal trees based on tree covering by Geary et al. [7] to support all the operations on trees proposed in other work:

**Theorem 1.** An ordinal tree of $n$ nodes can be represented using $2n + o(n)$ bits to support all the operations in Section 1 in $O(1)$ time under the word RAM model with $\Theta(\lg n)$ word size.

We compare our results with existing results in Table 1, in which the columns for BP and DFUDS list the results of tree representations based on balanced parentheses and DFUDS, respectively, and the columns old TC and new TC list the results by Geary et al. [7] and our results, respectively.

Our second result deals with BP and DFUDS representations as abstract data types, showing that any operation to be supported by BP or DFUDS in the future can also be supported by TC efficiently:

**Theorem 2.** Given a tree represented by TC, any $O(\lg n)$ consecutive bits of its BP or DFUDS sequence can be computed in $O(f(n))$ time, using at most $n/f(n) + \ldots$
**Fig. 1.** An example of covering an ordinal tree with parameters $M = 8$ and $M' = 3$, in which the solid curves enclose mini-trees and dashed curves enclose micro-trees.

$O(n \log |\log n|/\log n)$ extra bits, for any $f(n)$ where $f(n) = O(|\log n|)$ and $f(n) = \Omega(1)$ under the word RAM model with $\Theta(|\log n|)$ word size.

### 2 Preliminaries

#### 2.1 Bit Vectors

A key structure for many succinct data structures is a bit vector $B[1..n]$ that supports rank and select operations. Operations $\text{rank}_1(B, i)$ and $\text{rank}_0(B, i)$ return the number of 1s and 0s in $B[1..i]$, respectively. Operations $\text{select}_1(B, i)$ and $\text{select}_0(B, i)$ return the positions of $i^{th}$ 1 and 0, respectively. Lemma 1 addresses the problem, in which part (a) is from [6, 8], and part (b) is from [14].

**Lemma 1.** A bit vector $B$ of length $n$ with $t$ 1s can be represented using either: (a) $n+o(n)$ bits, or (b) $\log \binom{n}{t} + O(n \log |\log n|/\log n)$ bits, to support the access to each bit, $\text{rank}_1(B, i)$, $\text{rank}_0(B, i)$, $\text{select}_1(B, i)$, and $\text{select}_0(B, i)$ in $O(1)$ time.

#### 2.2 Succinct Ordinal Tree Representation Based on Tree Covering

Geary *et al.* [7] proposed an algorithm to cover an ordinal tree by mini-trees (*the tier-1 cover*) of size $\Theta(M)$ for a given parameter $M$ (they choose $M = [\log^4 n]$). Any two mini-trees computed by this algorithm either do not intersect, or only intersect at their common root. They showed that the size of any mini-tree is at most $3M - 4$, and the size of any mini-tree that does not contain the root of the tree is at least $M$. They further use the same algorithm with the parameter $M' = [\log n/24]$ to cover each mini-tree by a set of micro-trees (*the tier-2 cover*). See Figure 1 for an example, in which all the nodes are numbered in preorder.

Geary *et al.* [7] list the mini-trees $t_1, t_2, \ldots$ in an order such that in a preorder traversal of $T$, either the root of $t_i$ is visited before that of $t_{i+1}$, or if these two mini-trees share the same root, the children of the root in $t_i$ are visited before the children of the root in $t_{i+1}$. The $i^{th}$ mini-tree in the above sequence is denoted by $\mu^i$. All the micro-trees in a mini-tree are also listed in the same order, and
the $j^{th}$ micro-tree in mini-tree $\mu^i$ is denoted by $\mu^i_j$. When the context is clear, they also refer to this micro-tree using $\mu^i$. A node is denoted by its preorder number from an external viewpoint. To further relate a node to its position in its mini-tree and micro-tree, they define the $\tau$-name of a node $x$ to be a triplet $\tau(x) = <\tau_1(x), \tau_2(x), \tau_3(x) >$, which means that node $x$ is the $\tau_3(x)^{th}$ node visited in a preorder traversal of micro-tree $\mu^i_{\tau_1(x)}$. For a node that exists in more than one mini-tree and/or micro-tree, its lexicographically smallest $\tau$-name is its canonical name, and the corresponding copy of the node is its canonical copy. To enable the efficient retrieval of the children of a given node, Geary et al. [7] proposed the notion of extended micro-trees. To construct extended micro-trees, each micro-tree is initially made into an extended micro-tree, and each of its nodes is an original node. Every node $x$ that is the root of a micro-tree is promoted into the extended micro-tree to which its parent, $y$, belongs. See [7] for the details of promoting a node, and the properties of extended micro-trees. The main data structures by Geary et al. [7] are designed to represent each individual extended micro-tree. They take $2n + o(n)$ bits in total and can be used as building blocks to construct the tree representation at a higher level.

3 New Operations Based on Tree Covering (TC)

We now extend the succinct tree representation proposed by Geary et al. [7] to support more operations on ordinal trees, by constructing $o(n)$-bit auxiliary data structures in addition to their main data structures that use $2n + o(n)$ bits. As the conversion between the preorder number and the $\tau$-name of a given node can be done in constant time [7], we omit the steps of performing such conversions in our algorithms (e.g., we may return the $\tau$-name of a node directly when we need to return its preorder number). We use $T$ to denote the (entire) ordinal tree.

3.1 Height

Definition 1. Node $x$ is a tier-1 (or tier-2) preorder changer if $x = 1$, or if nodes $x$ and $(x - 1)$ are in different mini-trees (or micro-trees).

For example, in Figure 1, nodes 1, 2, 16, 22, and 30 are tier-1 preorder changers. Nodes 16, 17, 20, 22, 26 and others are tier-2 preorder changers. It is obvious that all the tier-1 preorder changers are also tier-2 preorder changers. By Lemma 2.2 in [7], each tier-1 (or tier-2) preorder changer can be mapped to a unique tier-1 (or tier-2) boundary node defined by Geary et al. [7]. As each mini-tree (or micro-tree) has at most 2 tier-1 (or tier-2) boundary nodes:

Lemma 2. The total number of tier-1 (or tier-2) preorder changers is at most twice the number of mini-trees (or micro-trees).

To compute height($x$), we compute $x$’s number of descendants, $d$, using nbdesc (from [7], see Section 1). Then all the descendants of $x$ are nodes $x + 1, x + 2, \cdots, x + d$. We have the formula: height($x$) = max_{i=1}^{d}(depth(x + i)) −
\text{depth}(x) + 1$. Therefore, the computation of $\text{height}(x)$ can be reduced to answering the range maximum query (i.e., retrieving the leftmost maximum value among the elements in a given range of the array) on the conceptual array $D[1..n]$, where $D[j] = \text{depth}(j)$ for $1 \leq j \leq n$. For any node $j$, we can compute $\text{depth}(j)$ in $O(1)$ time [7]. We now show how to support the range maximum query on $D$ using $o(n)$ additional bits without storing $D$ explicitly.

Based on Bender and Farach-Colton’s algorithm [2], Sadakane [15] showed how to support the range minimum/maximum query in $O(1)$ time on an array of $n$ integers using $o(n)$ additional bits. As we do not explicitly store $D$, we cannot use this approach directly (Sadakane [15] combined this approach with an array of integers encoded in the form of balanced parentheses). However, these $o(n)$-bits auxiliary data structures provide constant-time support for the range minimum/maximum query without accessing the array, when the starting and ending positions of the range are multiples of $\lg n$ (i.e., the given range is of the form $[k \lg n \ldots lg n]$, where $k$ and $l$ are integers). We thus construct these $o(n)$-bit auxiliary structures to support this special case on the conceptual array $D$. To support the general case, we construct (we assume that the $i$th tier-1 and tier-2 preorder changers are numbered $y_i$ and $z_i$, respectively):

- A bit vector $B_1[1..n]$, where $B_1[i] = 1$ iff node $i$ is a tier-1 preorder changer;
- A bit vector $B'_1[1..n]$, where $B'_1[i] = 1$ iff node $i$ is a tier-2 preorder changer;
- An array $C_1[1..n]$ ($l_i$ denotes the number of tier-1 preorder changers), where $C_1[i] = \tau_1(y_i)$;
- An array $C'_1[1..n]$ ($l'_i$ denotes the number of tier-2 preorder changers), where $C'_1[i] = \tau_2(z_i)$;
- An array $E[1..n]$, where $E[i]$ is the $\tau_3$-name of the node, $e_i$, with maximum depth among the nodes between $z_i$ and $z_{i+1}$ (including $z_i$ but excluding $z_{i+1}$) in preorder (we also consider the conceptual array $E'[1..n']$, where $E'[i] = \text{depth}(e_i)$, but we do not store $E'$ explicitly);
- A two-dimensional array $M$, where $M[i, j]$ stores the value $\delta$ such that $E'[i + \delta]$ is the maximum between and including $E'[i]$ and $E'[i + 2^{j'}]$, for $1 \leq i < l'_1$ and $1 \leq j \leq \lfloor \lg \lg n \rfloor$;
- A table $A_1$, in which for each pair of nodes in each possible micro-tree, we store the $\tau_3$-name of the node of the maximum depth between (inclusive) this pair of nodes in preorder. This table could be used for several trees.

There are $O(n/\lg^4 n)$ tier-1 and $O(n/\lg n)$ tier-2 preorder changers, so $B_1$ and $B'_1$ can be stored in $o(n)$ bits using Part (b) of Lemma 1. $C_1$, $C'_1$ and $E$ can also be stored in $o(n)$ bits (each element of $E$ is $O(\lg \lg n)$ bits). As $M[i, j] \leq 2^{\lg \lg n - 1}$, we can store each $M[i, j]$ in $\lg \lg n$ bits, so $M$ takes $O(n/\lg n \times \lg \lg n \times \lg \lg n) = o(n)$ bits. As there are $n^{1/4}$ possible micro-trees, and a micro-tree has at most $\lfloor \lg n/8 \rfloor$ nodes, $A_1$ occupies $o(n)$ bits. Thus these auxiliary structures occupy $o(n)$ bits.

To support the range maximum query on $D$, we divide the given range $[i, j]$ into up to three subranges: $[i, i/\lg n \lceil \lg n \rceil], [i/\lg n \lceil \lg n \rceil, j/\lfloor \lg n \rfloor \lceil \lg n \rceil]$ and $[j/\lfloor \lg n \rfloor \lceil \lg n \rceil, j]$. The result is the largest among the maximum values of these three subranges. The range maximum query on the second subrange is supported
by Sadakane’s approach (see above), so we consider only the first (the query on
the third one, and the case where \([i,j]\) is indivisible using this approach can be
supported similarly).

To support range maximum query for the range \([i, \lfloor i / \lg n \rfloor \lg n]\), we first use
\(B_1^i\) to check whether there is a tier-2 preorder changer in this range. If not, then
all the nodes in the range is in the same micro-tree. We further use \(B_1^i, B_1^{i'}, C_1\) and \(C_1^{i'}\) to retrieve the micro-tree by performing rank/select operations, and
then use \(A_1\) to compute the result in constant time.

If there are one or more tier-2 preorder changes in \([i, \lfloor i / \lg n \rfloor \lg n]\), let node
\(z_u\) be the first one and \(z_v\) be the last. We further divide this range into three
subranges: \([i, z_u]\), \([z_u, z_v]\) and \([z_v, \lfloor i / \lg n \rfloor \lg n]\). We can compute the maximum
values in the first and the third subranges using the method described in the
last paragraph, as the nodes in either of them are in the same micro-tree. To
perform range maximum query on \(D\) with the range \([z_u, z_v]\), by the definition of
\(E'\), we only need to perform the query on \(E'\) with range \([u, v-1]\), we observe that
\([u,v) = [u,u+2^s \cap [v-2^s, v]\), where \(s = \lceil \lg (v-1-u) \rceil\). As \(v-u < z_v - z_u < \lg n,\)
s \(\leq \lceil \lg (\lg n) \rceil\). Thus using \(M[u,s]\) and \(M[v-2^s, s]\), we can retrieve from \(E\) the
\(\tau_J\)-names of the nodes corresponding to the maximum values of \(E'\) in \([u, u+2^s]\)
and \([v-2^s, v]\), respectively. We further retrieve these two nodes using \(B_1, B_1', C_1\) and \(C_1'\), and the node with the larger depth is the one with the maximum
depth in range \([z_u, z_v]\).

### 3.2 LCA and distance

We pre-compute a tier-1 macro tree as follows. First remove any node that is
not a mini-tree root. For any two remaining nodes \(x\) and \(y\), there is an edge
from \(x\) to \(y\) if and among the remaining nodes, \(x\) is the nearest ancestor of \(y\) in
\(T\). Given a tree of \(n\) nodes, Bender et al. [2] showed how to support LCA using
\(O(n \lg n)\) additional bits\(^5\) for any tree representation. We store the tier-1 macro
tree using the representation by Geary et al. [7] and then build the structures by
Bender et al. [2] to support LCA and the operations listed in the column old \(T_C\) of
Table 1 in \(O(1)\) time. As the tier-1 macro tree has \(O(n/ \lg^4 n)\) nodes, this costs
\(O(n/ \lg^4 n \times \lg n) = o(n)\) bits. Similarly, for each mini-tree, we pre-compute a
tier-2 macro tree for the micro-tree roots, and store it using the same approach so
that all the tier-2 macro trees occupy \(o(n)\) bits in total. We also construct a table
\(A_2\) to store, for each possible micro-tree and each pair of nodes in it (indexed by
their \(\tau_J\)-names), the \(\tau_J\)-name of their lowest common ancestor. Similarly to the
analysis in Section 3.1, \(A_2\) occupies \(o(n)\) bits.

Figure 2 presents the algorithm to compute LCA. The correctness is straight-
forward and it clearly takes \(O(1)\) time. With the support for LCA and depth, the
support for distance is trivial.

---

\(^5\) They did not analyze the space cost in [2], but it is easy to verify that the cost is
\(O(n \lg n)\) bits.
Algorithm LCA($x, y$)

1. If $x$ and $y$ are in the same micro-tree, retrieve their LCA using a constant-time lookup on $A_2$ and return.
2. If $x$ and $y$ are not in the same micro-tree, but are in the same mini-tree, retrieve the roots, $u$ and $v$, of the micro-trees that $x$ and $y$ are in, respectively.
3. If $u = v$, return $u$ as the result.
4. If $u \neq v$, retrieve their lowest common ancestor, $w$, in the tier-2 macro tree.
5. Retrieve the two children, $i$ and $j$, of $w$ in the tier-2 macro tree that are ancestors of $x$ and $y$, respectively using depth and level_num. Then retrieve the parents, $k$ and $l$, of $i$ and $j$ in $T$, respectively.
6. If $k$ and $l$ are in two different micro-trees, return $w$ as the result. Otherwise, return LCA($k, l$).
7. If $x$ and $y$ are in two different mini-trees, retrieve the roots, $p$ and $q$, of the two different mini-trees, respectively.
8. If $p = q$, return $p$ as the result. Otherwise, similarly to Steps 4 and 5, retrieve two nodes $a$ and $b$, such that $a$ and $b$ are the children of the lowest common ancestor, $c$, of $p$ and $q$ in the tier-1 macro tree, and they are also the ancestors of $p$ and $q$, respectively. Retrieve the parents, $r$ and $s$, of $p$ and $q$ in $T$, respectively.
9. If $r$ and $s$ are in two different mini-trees, return $c$. Otherwise, return LCA($r, s$).

Fig. 2. An algorithm for computing LCA.

3.3 leaf_rank, leftmost_leaf, leaf_size and leaf_select

Definition 2. Each leaf of a mini-tree (or micro-tree) is a pseudo leaf of the original tree $T$. A pseudo leaf that is also a leaf of $T$ is a real leaf. Given a mini-tree (or micro-tree), we mark the first real leaf and each real leaf visited after an edge that leaves the mini-tree (or micro-tree). These nodes are called tier-1 (or tier-2) marked leaves.

For example, in Figure 1, nodes 6, 11 and 15 are pseudo leaves of micro-tree $\mu_1^2$, among which nodes 6 and 15 are real leaves, while node 11 is not. Nodes 23 and 29 are tier-2 marked leaves. We observe the following property.

Property 1. Given a mini-tree (or micro-tree) and a pair of its tier-1 (or tier-2) marked leaves such that there is no marked leaf between them in preorder, the real leaves visited in preorder between these two leaves (including the left one but excluding the right) have the property that, when listed from left to right, their leaf ranks are consecutive integers. The real leaves that are to the right of (and including) the rightmost marked leaf have the same property.

Observe that each tier-1 marked leaf (except perhaps the first real leaf in a mini-tree) corresponds to an edge that leaves its mini-tree (which in turn corresponds to a unique mini-tree root). Therefore, the number of tier-1 marked leaves is at most twice as many as the number of mini-trees, which is $O(n \log n)$. We thus store for each mini-tree $\mu_i$, the ranks of its tier-1 marked leaves. Similarly, the number of tier-2 marked leaves is at most twice the number of micro-trees. For each tier-2 marked leaf $x$, let $y$ be the last tier-1 marked leaf visited before...
\( x \) during a preorder traversal (if \( x \) is a tier-1 marked leaf then we set \( y = x \)). We compute the difference of \( \text{leaf} \_\text{rank}(x) \) and \( \text{leaf} \_\text{rank}(y) \) (bounded by the number of nodes in \( x \)'s mini-tree) and store for each micro-tree, the list of such values for its tier-2 marked leaves. With additional bit vectors to mark the positions of these marked leaves, and an \( o(n) \)-bit table to look up the number of nodes between two pseudo leaves in a micro-tree, we can support \( \text{leaf} \_\text{rank} \) in constant time by Property 1. We omit the details.

To support \( \text{leftmost} \_\text{leaf} \) and \( \text{rightmost} \_\text{leaf} \), given a node \( x \) with preorder number \( i \), postorder number \( j \), and \( m \) descendants, we observe that the postorder number of its leftmost leaf is \( j - m \), and the preorder number of its rightmost leaf is \( i + m \). Thus the support of these two operations follows the constant-time support for \( \text{node} \_\text{rank} \_\text{PRE}, \text{node} \_\text{rank} \_\text{POST}, \text{node} \_\text{select} \_\text{PRE} \) and \( \text{node} \_\text{select} \_\text{POST} \). With the constant-time support for \( \text{leaf} \_\text{rank}, \text{leftmost} \_\text{leaf} \) and \( \text{rightmost} \_\text{leaf} \), we can also support \( \text{leaf} \_\text{size} \) in constant time.

To support \( \text{leaf} \_\text{select} \), we observe that given a leaf \( x \) that is not a tier-1 (or tier-2) marked leaf, if the closest tier-1 (or tier-2) marked leaf to the left is node \( y \) (or node \( z \)), then \( \tau_1(x) = \tau_1(y) \) (or \( \tau_2(x) = \tau_2(z) \)). This property allows us to support \( \text{leaf} \_\text{select} \) in \( O(1) \) time with \( o(n) \) additional bits, using an approach similar to that of Section 3.1. We omit the details.

### 3.4 node\_rank\_DFUDS and node\_select\_DFUDS

We use the following formula proposed by Barbay et al. [1]:

\[
\text{node} \_\text{rank} \_\text{DFUDS}(x) = \begin{cases} 
\text{child} \_\text{rank}(x) - 1 + \text{node} \_\text{rank} \_\text{DFUDS} \left( \text{child} \_\text{parent}(x), 1 \right) & \text{if } \text{child} \_\text{rank}(x) > 1, \\
\text{node} \_\text{rank} \_\text{DFUDS}(x) + \sum_{y \in \text{anc}(x) \setminus r} \left( \text{degree}(\text{parent}(y)) - \text{child} \_\text{rank}(y) \right) & \text{otherwise},
\end{cases}
\]

where \( \text{parent}(x) = \text{level} \_\text{anc}(x, 1) \), \( \text{anc}(x) \) is the set of ancestors of \( x \), and \( r \) is the root of \( T \). This formula reduces the support of \( \text{node} \_\text{rank} \_\text{DFUDS} \) to the support of computing \( S(x) = \sum_{y \in \text{anc}(x) \setminus r} \text{degree}(\text{parent}(y)) - \text{child} \_\text{rank}(y) \) for any given node \( x \) [1]. We use \( u(x) \) and \( v(x) \) to denote the roots of the mini-tree and micro-tree of node \( x \), respectively. Then we compute \( S(x) \) as the sum of the following three values as suggested by Barbay et al. [1]:

\[
S_1(x) = S(u(x)), \quad S_2(x) = S(v(x)) - S(u(x)), \quad S_3(x) = S(x) - S(v(x)).
\]

It is trivial to support the computation of \( S_1(x) \) in constant time: for each mini-tree root \( i \), we simply precompute and store \( S(i) \) using \( O(n/\log^2 n) = o(n) \) bits. However, we cannot do the same to support the computation of \( S_2(x) \). The approach of Barbay et al. [1] does not solve the problem, either, because Property 1 in [1] does not hold. To address this problem, we extend the mini-trees using the same method used to extend micro-trees. As with the Proposition 4.1 in [7], we have that except for the roots of mini-trees, all the other original nodes have the property that (at least a promoted copy of) each of their children is in the same extended mini-tree as themselves. With this we can support \( \text{node} \_\text{rank} \_\text{DFUDS} \). We omit the details.

To support \( \text{node} \_\text{select} \_\text{DFUDS} \), we first define the \( \tau^* \)-name of a node and show how to convert \( \tau^* \)-names to \( \tau \)-names. Then we show how to convert \( \text{DFUDS} \) numbers to \( \tau^* \)-names.
Definition 3. Given a node $x$ whose $\tau$-name is $\tau(x) = \langle \tau_1(x), \tau_2(x), \tau_3(x) \rangle$, its $\tau^*$-name is $\tau^*(x) = \langle \tau_1(x), \tau_2(x), \tau_3(x) \rangle$, if $x$ is the $\tau_3(x)^{th}$ node of its micro-tree in DFUDS order.

For example, in Figure 1, node 29 has $\tau$-name $<3,1,5>$ and $\tau^*$-name $<3,1,4>$. To convert the $\tau^*$-name of a node to its $\tau$-name, we only need convert its $\tau_3^*$-name to its $\tau_3$-name, in constant time using a table of size $o(n)$.

As with the algorithm in Section 4.3.1 of [7] supporting node select pass, the idea of computing the $\tau^*$-name given a DFUDS number is to store the $\tau^*$-names of some of the nodes, and compute the $\tau^*$-names of the rest using these values.

Definition 4. List the nodes in DFUDS order, numbered $1, 2, \ldots, n$. The $i^{th}$ node in DFUDS order is a tier-$1$ (or tier-$2$) DFUDS changer if $i = 1$, or if the $i^{th}$ and $(i - 1)^{th}$ nodes in DFUDS order are in different mini-trees (or micro-trees).

For example, in Figure 1, nodes $1, 2, 16, 3, 17, 22$ and $30$ are tier-$1$ DFUDS changers, and nodes $2, 16, 3, 6, 7, 11, 4$ and others are tier-$2$ DFUDS changers. It is obvious that all the tier-$1$ DFUDS changers are also tier-$2$ DFUDS changers. We have the following lemma (we omit its proof because of the space constraint).

Lemma 3. The number of tier-$1$ (or tier-$2$) DFUDS order changers is at most four times the number of mini-trees (or micro-trees).

This allows us to convert DFUDS numbers to $\tau^*$-names with $o(n)$ additional bits, using an approach similar to that of Section 3.1. We omit the details.

### 3.5 level_leftmost, level_rightmost, level_succ and level_pred

We define the $i^{th}$ level of a tree to be the set of nodes whose depths are equal to $i$. To support level_leftmost (level_rightmost can be supported similarly), we first show how to compute the $\tau_1$-name, $u$, of the node level_leftmost($i$). Let $h$ be the height of $T$. We construct a bit vector $B[1..h]$, in which $B[j] = 1$ iff the nodes level_leftmost($j$) and level_leftmost($j$) are in two different mini-trees, for $1 < j < h$ (we set $B[1] = 1$). Let $m$ be the number of 1s in $B$, and we construct an array $C[1..m]$ in which $C[k]$ stores the $\tau_1$-name of the node level_leftmost(select$_1(B, k)$). As the $\tau_1$-name of the node level_leftmost($i$) is the same as that of the node level_leftmost(select$_1(B, \text{rank}$_1$B, i)$), we have that $u = C[\text{rank}$_1$B, i]$. To analyze the space cost of $B$ and $C$, we observe that if a given value, $p$, occurs in $i$ times in $C$, then the mini-tree $\mu^p$ has at least $q-1$ edges that leave $\mu^p$. If we map the first occurrence of $p$ to mini-tree $\mu^p$, and each of the rest of the occurrences of $p$ to a unique edge of $\mu^p$ that leaves $\mu^p$, then we can map each element of $C$, to either to a unique mini-tree, or to a unique edge that leaves a mini-tree. Thus $m$ is at most the number of mini-trees plus the number of edges that leave a mini-tree, which is $O(n/\log^4 n)$. Hence $B$ and $C$ occupy $O(n)$ additional bits. Similarly, we can support the computation of the $\tau_2$-name, $v$, of the node level_leftmost($i$) in constant time using $o(n)$-bit table.

The support for level_succ and level_pred is based on the above approach and other techniques. We omit the details.
4 Computing a subsequence of BP and DFUDS

We use $\text{BP}[1..2n]$ to denote the BP sequence. Recall that each opening parenthesis in BP corresponds to the preorder number of a node, and each closing parenthesis corresponds to the postorder. Thus the number of opening parentheses corresponding to tier-1 (or tier-2) preorder changers is $O(n/\lg^2 n)$ (or $O(n/\lg n)$), and we call them tier-1 (or tier-2) marked opening parentheses.

We first show how to compute the subsequence of BP starting from a tier-2 marked opening parenthesis up to the next tier-2 marked opening parenthesis. We use $j$ and $k$ to denote the positions of these two parentheses in BP, respectively, and thus our goal is to compute $\text{BP}[j..k-1]$. We construct auxiliary data structures $B_k$, $B'_k$, $C_\ell$ and $C'_\ell$, which are similar to the structures in Section 3.1, except that they store $\tau$-names of the nodes that correspond to marked opening parentheses. We also construct a table $A_\ell$, in which for each possible micro-tree and each one of its nodes, we store the subsequence of the balanced parenthesis sequence of the micro-tree, starting from the opening parenthesis corresponding to this node, to the end of this sequence, and we also store the length of such a subsequence. These auxiliary data structures occupy $O(n \lg \lg n / \lg n)$ bits. To compute $\text{BP}[j..k-1]$, we first compute, in constant time, the $\tau$-names of the tier-2 preorder changers, $x$ and $y$, whose opening parenthesis are stored in $\text{BP}[j]$ and $\text{BP}[k]$, respectively, using $B_k$, $B'_k$, $C_\ell$ and $C'_\ell$. If $x$ and $y$ are in the same micro-tree, then we can perform a constant-time lookup on $A_\ell$ to retrieve the result. Otherwise, $\text{BP}[j..k-1]$ is the concatenation of the following two sequences: the subsequence of the balanced parenthesis sequence of $x$'s micro-tree, starting from the opening parenthesis corresponding to $x$, to the end of this sequence, and zero or more closing parentheses. Thus, $\text{BP}[j..k-1]$ can either be computed in constant time using table lookup on $A_\ell$ if its length is at most $\lg n/4$, or any $\lg n$ subsequence of it can be computed in constant time.

To compute any $O(\lg n)$-bit subsequence of BP, we conceptually divide BP into blocks of size $\lg n$. As any $O(\lg n)$-bit subsequence spans a constant number of blocks, it suffices to support the computation of a block. For a given block with $u$ tier-2 marked opening parentheses, we can run the algorithm described in the last paragraph at most $u + 1$ times to retrieve the result. To facilitate this, we choose a function $f(n)$ where $f(n) = O(\lg n)$ and $f(n) = \Omega(1)$. We explicitly store the blocks that have $2f(n)$ or more tier-2 marked opening parentheses, which takes at most $2n/(\lg n \times 2f(n)) \times \lg n = n/f(n)$ bits. Thus, a block explicitly stored can be computed in $O(1)$ time, and a block that is not can be computed in $O(f(n))$ time as it has at most $2f(n)$ tier-2 marked opening parentheses.

To support the computation of a word of the DFUDS sequence, recall that the DFUDS sequence can be considered as the concatenation of the DFUDS subsequences of all the nodes in preorder (see Section 1.1). Thus the techniques used above can be modified to support the computation of a subsequence of DFUDS. The main difference is that we need the notion of extended micro-trees, as they have the property that the children of each non-root original node of an extended micro-tree are all in the same extended micro-tree.
5 Open Problems

The first open problem is whether we can compute any $O(\lg n)$-bit subsequence of BP or DFUDS in constant time using $o(n)$ additional bits for TC. Our result in Theorem 2 is in the form of time/space tradeoff and we do not know whether it is optimal. Other interesting open problems include the support of the operations that are not previously supported by BP, DFUDS or TC. One is to support rank/select operations on the level-order traversal of the tree. Another one is to support level_leftmost (level_rightmost) on an arbitrary subtree of $T$.

References