A Simple Representation of Tree Covering Utilizing Balanced Parentheses and Efficient Implementation of Average-Case Optimal RMQs

Kou Hamada
The University of Tokyo, Japan

Sankardeep Chakraborty
The University of Tokyo, Japan

Seungbum Jo
Chungnam National University, Daejeon, South Korea

Takuto Koriyama
The University of Tokyo, Japan

Kunihiko Sadakane
The University of Tokyo, Japan

Srinivasa Rao Satti
Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Tree covering is a technique for decomposing a tree into smaller sized trees with desirable properties, and has been employed in various succinct data structures. However, significant hurdles stand in the way of a practical implementation of tree covering: a lot of pointers are used to maintain the tree-covering hierarchy and many indices for tree navigational queries consume theoretically negligible yet practically vast space. To tackle these problems, we propose a simple representation of tree covering using a balanced-parenthesis representation. The key to the proposal is the observation that every micro tree splits into at most two intervals on the BP representation. Utilizing the representation, we propose several data structures that represent a tree and its tree cover, which consequently allow micro tree compression with arbitrary coding and efficient tree navigational queries. We also applied our data structure to average-case optimal RMQ by Munro et al. [ESA 2021] and implemented the RMQ data structure. Our RMQ data structures spend less than $2n$ bits and process queries in a practical time on several settings of the performance evaluation, reducing the gap between theoretical space complexity and actual space consumption. For example, our implementation consumes $1.822n$ bits and processes queries in 5 µs on average for random queries and in 13 µs on average for the worst query widths. We also implement tree navigational operations while using the same amount of space as the RMQ data structures. We believe the representation can be widely utilized for designing practically memory-efficient data structures based on tree covering.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Hypersuccinct trees, Succinct data structures, Range minimum queries, Binary trees

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.64

Related Version *Full Version*: https://arxiv.org/abs/2407.00573

Supplementary Material *Software (Source Code)*: https://github.com/zkou-ut/hyperrmq [16]
archived at swh:1:dir:5d624172e0f2ed764967a13d0dca00d30dc32613

1 Introduction

With the explosive growth of data volumes, data structures that process data quickly and memory-efficiently are of paramount interest. Data structures that asymptotically achieve the information-theoretic lower bound and support operations efficiently are called succinct data structures. Starting with the work by Jacobson [19], many succinct solutions have been proposed for various settings.

Trees have been thoroughly studied in the context of succinct representations due to their fundamental nature and wide applicability. One major subclass of trees is ordinal trees, which are rooted trees where the children of each node are ordered. The information-theoretic lower bound of representing ordinal trees with n nodes is $2n - \Theta(\log n)$[1] bits [22]. Researchers have devised many succinct data structures for ordinal trees, such as the level-order unary degree sequence (LOUDS) [19], the balanced parentheses (BP) [22], and the depth-first unary degree sequence (DFUDS) [4], to name a few. All these representations are efficient in practice [1, 24]. Another important subclass of trees is cardinal trees. Cardinal trees are rooted trees where every node has a fixed number of labeled slots and every slot has a child node or is empty. In this paper, we consider the cardinal trees whose nodes have two slots, which are called binary trees. Munro and Raman [22] employed a bijection between binary trees with n nodes and ordinal trees with $n + 1$ nodes to extend their BP representation for ordinal trees to binary trees, obtaining a succinct representation of binary trees using $2n + o(n)$ bits.

All the representations of trees described above convert trees into sequences and translate tree navigational queries into operations on the sequences. Another promising approach is tree covering [10], which decomposes trees into parts called micro trees. Since the decomposition is suited for look-up tables, tree covering can support a variety of queries in constant time [8, 10]. Also, its hierarchical structure is beneficial in designing succinct data structures involving trees [5, 6, 10, 17, 21, 24, 28]. For example, hypersuccinct trees [21], which we discuss in Sec. 2, encode micro trees with a Huffman code and achieve optimal compression for various tree sources.

Although tree covering powerfully facilitates designing succinct data structures for trees, the data structures based on tree covering tend to require numerous $o(n)$-bit indexes, such as pointers and look-up tables. While the data structure supports a wide range of tree navigational queries, the number of indexes needed to support the queries increases as well. Thus, straightforward implementation of data structures based on tree covering is unlikely to be efficient in practice. To the best of our knowledge, there is no practical implementation of succinct data structures based on tree covering. Thus, a practical design of tree covering suitable for implementation is extremely desirable.

We illustrate the need for practical tree-covering data structures by discussing their application to the Range Minimum Query (RMQ) problem. Given a static array A of length n consisting of totally ordered objects, an RMQ data structure supports the following queries efficiently: given two indices i, j with $1 \leq i \leq j \leq n$, return $\operatorname{argmin}_{i \leq k \leq j} A[k]$, i.e., the index of the minimum in the subarray of A from the i-th element to the j-th element. The problem appears as a subroutine in many real-world applications, such as auto-completion [18], data compression [7], and document retrieval [27].

Fischer and Heun [13] first designed a succinct RMQ data structure using $2n + o(n)$ bits, achieving the worst-case optimal space complexity. As for implementation, Ferrada and Navarro [11] provided a practical implementation of a succinct RMQ data structure by

[1] In this paper, log and lg denote the logarithm of base e and 2, respectively.

utilizing the range min-max tree [24]. Their implementation consumes $2.1n$ bits and takes 1–3 µs per query. Baumstark et al. [3] also gave another competitive implementation with a faster query time of about 1 µs.

On the other hand, if we assume that the input array for RMQ is a random permutation, the lower bound for the expected space consumption drops below $2n$ bits: this lower bound, also known as the expected effective entropy, is $1.736n + \mathrm{o}(n)$ bits [14, 20]. Although the asymptotic lower bound for the worst-case space complexity remains $2n$ bits, it opens up the possibility of designing RMQ data structures that consume less than $2n$ bits on average.

Davoodi et al. [8] proposed an RMQ data structure that uses $1.919n+\mathrm{o}(n)$ bits on average and supports constant-time queries. Munro et al. [21] designed hypersuccinct trees and applied them to Cartesian trees, obtaining an RMQ data structure that takes $1.736n + \mathrm{o}(n)$ bits on average and $2n + \mathrm{o}(n)$ bits in the worst case while supporting constant-time queries. They also found that their RMQ data structure uses $2 \lg \binom{n}{r} + \mathrm{o}(n)$ bits when applied to an array of length n with r increasing runs. The average space consumption of their RMQ data structure is asymptotically optimal in both cases.

While both RMQ data structures theoretically improve the average space consumption from the succinct solution by Fischer and Heun [13], they employ the tree-covering technique, making straightforward implementation inefficient in practice. To the best of our knowledge, there is no implementation of such an RMQ data structure, nor is there any implementation that consumes less than $2n$ bits. Therefore, a practical representation of tree covering may lead to a space-efficient RMQ data structure that is unprecedented.

1.1 Our contribution

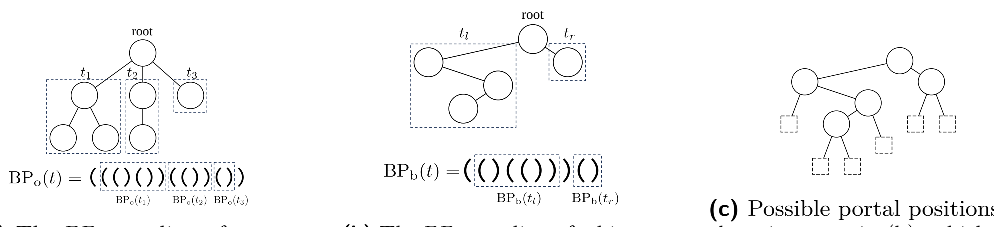
Our main contribution is the proposal of a simple representation of tree covering in the BP representation for both ordinal trees and binary trees. The representation is based on the observation that every micro tree splits into at most two intervals in the BP representation. Utilizing the representation, we propose several practical designs of succinct data structures for trees and their tree covers. Also, as an application, we present an optimized design of an average-case optimal RMQ data structure based on hypersuccinct trees. We also implement RMQ data structures using our tree-covering representation. In empirical evaluations, the implementations spend less than $2n$ bits and process queries in a practical time on several settings of the performance evaluation. Furthermore, we implement tree navigational operations while spending the same amount of space as the RMQ data structures.

The remainder of this paper is structured as follows: Sec. 2 introduces the prerequisite knowledge. Sec. 3 discusses a simple representation of tree-covering structure in the BP representation for ordinal trees and binary trees and presents practical design of data structures. Sec. 4 presents performance evaluation. Sec. 5 concludes the paper and provides future directions. Due to space constraints, we omit the proofs and skip various other details from the main part.

2 Preliminaries

2.1 Balanced Parentheses

We introduce a balanced sequence of parentheses and some operations on it [23], which will be useful in the Balanced-Parenthesis (BP) representation of trees. A sequence of parentheses is balanced if the following conditions are satisfied: (a) The sequence is of length $2n$, i.e., even, and consists of n opening parentheses and n closing parentheses. (b) The sequence



(a) The BP encoding of an ordinal tree.

(b) The BP encoding of a binary tree.

(c) Possible portal positions of the micro tree in (b), which are indicated by dashed squares.

Figure 1 Examples of the BP encodings of ordinal trees and binary trees and an example of micro-tree portals.

has n pairs of matching parentheses: each pair has an opening parenthesis on the left and a closing parenthesis on the right, and when considering intervals whose endpoints are matching parentheses, any two of the intervals are disjoint or one contains the other.

We also define some operations on a balanced sequence of parentheses as follows. (a) Given a closing parenthesis, the `open` operation returns the opening parenthesis that matches the given closing parenthesis. (b) Given an opening parenthesis, the `close` operation returns the closing parenthesis that matches the given opening parenthesis. (c) Given an index r, the $\texttt{rank}_{(}(r)$ and $\texttt{rank}_{)}(r)$ operations respectively count the number of opening and closing parentheses among the first r parentheses in the sequence. (d) Given a number j, the $\texttt{select}_{(}(j)$ and $\texttt{select}_{)}(j)$ operations respectively find the j-th opening and closing parenthesis and return the index of the parenthesis in the sequence.

2.2 BP Representations of Trees

Here, we define the BP representations of ordinal trees and binary trees [9, 21, 23] and present some properties of the representations.

▶ **Definition 1** (BP encoding of ordinal trees). *The BP encoding* $\mathrm{BP_o}(t)$ *of an ordinal tree* t *is defined recursively as follows:* $\mathrm{BP_o}(t) = (\cdot \mathrm{BP_o}(t_1) \cdots \mathrm{BP_o}(t_k) \cdot)$*. Here,* k *denotes the number of the children of the root, and* t_i *denotes the subtree rooted at the* i*-th child of the root, respectively.*

▶ **Definition 2** (BP encoding of binary trees). *The BP encoding* $\mathrm{BP_b}(t)$ *of a binary tree* t *is defined recursively as follows:*

$$\mathrm{BP_b}(t) = \begin{cases} \varepsilon & \textit{if } t \textit{ is empty;} \\ (\cdot \mathrm{BP_b}(t_l) \cdot) \cdot \mathrm{BP_b}(t_r) & \textit{otherwise.} \end{cases}$$

Here, t_l *and* t_r *denote the subtrees whose roots are the left and right children of the root, respectively.*

Examples of the BP encodings are shown in Fig. 1. In both encodings, nodes correspond to matching pairs of parentheses, and subtrees correspond to balanced intervals.

For both binary trees and ordinal trees, most of the tree navigational queries can be achieved by utilizing the BP operations described above. Here, we present a proposition for binary trees that enables converting a node, its preorder, and its inorder to one another.

▶ **Proposition 3** (Order of parentheses in BP of binary trees). *If we extract opening (resp. closing) parentheses from the BP sequence of a binary tree, then the corresponding nodes appear in the preorder (resp. inorder).*

2.3 The Farzan–Munro Algorithm

The Farzan–Munro tree-covering algorithm [10] decomposes an ordered tree into smaller sized trees, called *micro trees*, which may share root nodes and satisfy Prop. 4. When applied to binary trees, the micro trees become disjoint and additionally satisfy Prop. 5.

▶ **Proposition 4** (Properties of Tree Covers [10]). *For an ordinal tree with n nodes and a parameter $B \geq 1$, the Farzan–Munro algorithm produces a tree cover satisfying the following requirements.*

1. *The number of micro trees is $\mathrm{O}(n/B)$.*
2. *Each micro tree has less than $2B$ nodes.*[2]
3. *Each micro tree has at most one outgoing edge, apart from those from the root of the micro tree.*

▶ **Proposition 5** (Properties of Binary Tree Decomposition [10, 21]). *The binary tree decomposition obtained by the Farzan–Munro algorithm satisfies the following additional properties:*

1. *Contracting the micro trees into single nodes gives a binary tree, which we call a top-tier tree.*
2. *If a micro tree has two outgoing edges toward children, it consists of a single node.*

2.4 Hypersuccinct Trees

Hypersuccinct trees [21] asymptotically achieve average-case optimal compression for trees from various sources. Here, we present some of their results in binary trees as we later implement the data structure and benchmark it. Applying it to Cartesian trees of random permutations requires $1.736n + \mathrm{o}(n)$ bits on average, which is asymptotically optimal. Also, Cartesian trees of arrays of length n with r increasing runs can be encoded using $2 \lg \binom{n}{r} + \mathrm{o}(n)$ bits. Application of these results to the RMQ problem gives average-case optimal RMQ data structures for those input arrays.

Hypersuccinct trees first utilize the Farzan–Munro algorithm [10] described above to decompose a tree into micro trees, which are subsequently compressed with a Huffman code. Since the micro trees are the most dominant part of the space, compression with a Huffman code reduces the leading term in space complexity for various tree sources.

To explain the data structures of hypersuccinct binary trees, we define portals. To recover the original tree from the decomposed micro trees, it is insufficient just to store the micro trees and the top-tier tree; the connection between nodes of different micro trees cannot be recovered. Thus, we also store where the roots of the child micro trees are initially placed, called *portals*. Each micro tree may contain at most two portals; one is to the left-child micro tree and the other is to the right-child micro tree. Fig. 1 (c) shows an example of possible portal positions. The number of possible portal positions is one more than the number of nodes. The index of the portal from left to right is called the portal rank.

Thus, a hypersuccinct binary tree consists of micro trees, a top-tier tree, and portals. We consider the space consumption of these items when applied to Cartesian trees of random permutations. The parameter B is set to $\lceil (\lg n)/8 \rceil$ to make the data structure succinct.

The micro trees are dominant in space consumption: Huffman codes spend $1.736n + \mathrm{O}(n(\log B)/B)$ bits, while enumerating the BP representation of each code in a table consumes $\mathrm{O}(2^{4B} B)$ bits, i.e., $\mathrm{O}(\sqrt{n} \log n)$ bits. The top-tier tree and the portals both consume $\mathrm{o}(n)$

[2] The original paper [10] states that the micro-tree size is at most $2B$, but the algorithm does not yield a component of size $2B$.

bits. The top-tier tree has $O(n/B)$ nodes by Prop. 4, so representing the tree by the BP representation requires $O(n/B)$ bits. As for the portals, since each micro tree has less than $2B$ nodes by Prop. 4, it has at most $2B$ possible portal positions. Thus, each portal can be represented in $\lceil \lg 2B \rceil$ bits. Since the number of portals is twice the number of micro trees, the portals use $O(n(\log B)/B)$ bits.

3 BP Representation of Tree Covering

3.1 Ordinal Trees

In this section, we propose a simple representation of tree covering that leverages the BP sequence. At the core of the representation is that every micro tree corresponds to at most two intervals on the BP representation. This property motivates us to use the index of the BP sequence while preserving the tree-covering structure. We also give a data-structure design that isolates micro trees as a sequence and enables compressing micro trees with an arbitrary encoding.

3.1.1 Modified tree cover

We slightly modify the definition of the tree cover and give its simple representation using multi-type parentheses sequences. Then we show how to obtain the original tree cover from the modified tree cover.

First, we compute the original tree cover of an ordinal tree t, using the tree cover algorithm of Farzan and Munro [10]. Let v be a root of a micro tree μ consisting of multiple nodes. Then, we create a dummy node w and hang it from v. We change the parent of children of v in μ as w. We call the resulting tree t'. The modified tree cover is obtained by splitting μ into two micro trees: a singleton micro tree that contains only v and a micro tree that contains w and its children in μ.

The modified tree cover obtained by the procedure above consists of disjoint micro trees. As in the case of binary trees, we define a *top-tier tree* as a tree obtained by contracting the micro trees of the modified tree cover. The theorem below describes the properties of the modified tree cover.

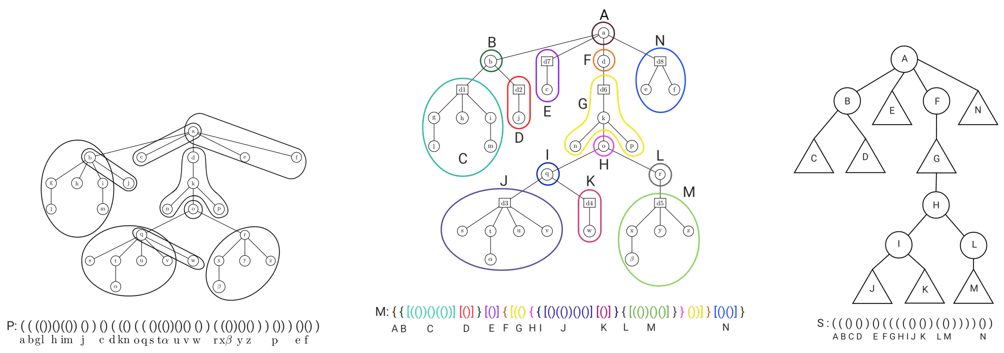
▶ **Theorem 6.** *For an ordinal tree with n nodes and a parameter $B \geq 1$, we can obtain a modified tree cover satisfying the following requirements.*

1. *The number of micro trees is $O(n/B)$.*
2. *Each micro tree either consists of a single node with an arbitrary number of child micro trees (called a singleton micro tree), or it consists of less than $2B$ nodes and has at most one child micro tree (called a non-singleton micro tree).*

We show how to represent the modified tree cover of a modified tree t' using multi-type parentheses sequence. We use three types: `()`, `{}`, and `[]`. Normal parentheses `()` represent non-root nodes of micro trees. Curly braces `{}` represent root nodes of singleton micro trees. Square brackets `[]` represent root nodes of non-singleton micro trees, i.e., the dummy nodes inserted in t. We simply construct a multi-type parentheses sequence M during a Depth-First Search (DFS) of t'.

This multi-type parentheses sequence M has the following good properties.

1. If we remove all normal parentheses from M and convert curly braces and square brackets into normal parentheses, it coincides with the BP sequence S of the top-tier tree.


P: (((())()(()) ()) () ((() ((()(())()() ()) ((())()())) ())) ()())
a bgl h im j c dkn oqs tα u v w rxβ y z p e f
M: { { [(())()(())] [()] } [()] { [(() { { [()(())()()] [()] } { [(())()()] } } ()] } [()()] }
AB C D E F G H I J K L M N
S : ((() ()) () ((((() ()) (())))) ())
ABCD EFGHIJ K LM N

(a) Tree cover with parameter $B = 5$. **(b)** Modified tree cover. **(c)** Top-tier tree.

Figure 2 Tree cover and modified tree cover of the tree.

2. If we remove all square brackets from M and convert curly braces into normal parentheses, it coincides with the BP sequence P of t.
3. For any micro tree μ, its BP representation is cut into at most two parts.

Let us briefly explain why the third property holds. There are two possible cases why a micro-tree BP splits: an outgoing edge emanates from a non-root node or another micro tree hangs to the micro-tree root and is between two children of the micro-tree root. The former case occurs at most once by Prop. 4. Due to the greedy strategy of the Farzan–Munro algorithm, the latter case rarely happens: it can be verified by examining the detail of the algorithm that the latter case occurs only when the micro-tree root has only one heavy child, which then implies that the micro tree has no other outgoing edge. Thus, the latter case occurs at most once and both the former and latter cases do not simultaneously happen.

The third property also holds for the original BP sequence P if we consider the partition of P into micro-tree BP sequences and apply the conversion described in the second property. This is equivalent to excluding the micro-tree root when considering the BP representation of a non-singleton micro tree. We assume this implicitly when we consider a micro-tree BP of P. Otherwise, a micro tree can correspond to more than two intervals of P. Also, for simplicity, we arbitrarily split a single interval corresponding to a whole micro tree into two intervals so that every micro tree corresponds to two intervals; we call the resulting two intervals *chunks*.

We observe that any micro tree in the modified tree cover either has a dummy node as its root or is a singleton micro tree. Let μ be a non-singleton micro tree in the modified tree cover. The original tree cover is obtained by simply merging μ with the parent node of the dummy root node of μ.

Note that the sequence M leads to a succinct representation of the tree t and its tree cover if B is super-constant with respect to n. Because the length of M is $2n + \mathrm{O}(n/B)$ and the number of curly braces and square brackets is $\mathrm{O}(n/B)$, we can store M in $2n + \mathrm{O}(n \log B/B + n \log\log n/\log n)$ bits by using sparse bitvectors [26].

3.1.2 Practical Designs

Although the sequence M leads to an intuitive and succinct representation of ordinal trees, there is still room for simplification in practice. In what follows, Theorem 7 first presents a practical design of tree-covering indexes for ordinal trees. Theorem 8 then discusses the replacement of the indexes to handle tree navigational queries. Theorem 9 finally describes another design that enables compressing micro trees with an arbitrary encoding, which is suitable for designing succinct representations for subclasses of ordinal trees.

We first describe the role of curly braces and square brackets when viewing M as a representation of the original tree cover:

1. Curly braces and square brackets together delimit the original BP sequence P based on the original tree cover.
2. The difference between curly braces and square brackets is whether the micro-tree BP is complete or not; a micro-tree BP enclosed with square brackets lacks the micro-tree root and needs to be enclosed with a pair of parentheses to obtain the complete micro-tree BP.

Thus, instead of inserting curly braces and square brackets into P, it suffices to store where to delimit the sequence P and whether each micro-tree BP is complete or not. Therefore, we present a practical succinct representation of ordinal trees based on this approach.

▶ **Theorem 7.** *Let $\mu_1, \ldots, \mu_m$ be the micro trees in the DFS order on the top-tier tree. The following indexes represent the tree cover if combined with the BP sequence P. The indexes consume $\mathrm{o}(n)$ bits if B is super-constant.*

- *A sparse bitvector V of the same length as P which marks the starting position of each chunk of P with 1. It handles rank and select operations in constant time and spends $\mathrm{O}(n \log B/B + n \log\log n/\log n)$ bits [26].*
- *A data structure that represents the BP sequence S of the top-tier tree. It handles basic operations on the BP sequence S in constant time and spends $\mathrm{O}(n/B)$ bits [24].*
- *A boolean array F whose i-th element indicates whether the BP of μ_i in P is complete or not, i.e., whether μ_i consists of a single node or not. It consumes $\mathrm{O}(n/B)$ bits.*

Given a position i of P, we can determine the micro tree containing i as follows. First, the index r of the chunk can be obtained by $\mathtt{rank}_1(V, i)$. Then, the range of the chunk on P is from $\mathtt{select}_1(V, r)$ to $\mathtt{select}_1(V, r+1)$. To fully restore the micro tree, we also need to retrieve the other chunk that is a part of the same micro tree, and the index of the other chunk can be obtained by using either `open` or `close` operation on S and r. Finally, we refer to the array F to see if the micro-tree root needs to be merged. The index of F can be obtained by $\mathtt{rank}_($ operation on the index of the left chunk of the micro tree.

One advantage of the indexes in Theorem 7 is that they are compatible with the indexes of the BP representation, especially with the range min-max tree [24]. We can replace V with the range min-max tree by modifying the algorithm: instead of fixed-length blocks in the original algorithm, we use variable-length blocks so that the leaves of the range min-max tree correspond to the chunks of P. Thereby, the data structure can handle BP operations on P and thus tree navigational queries on t.

▶ **Theorem 8.** *Consider the data structures in Theorem 7. Tree navigational queries can be supported by replacing the sparse bitvector V with a range min-max tree that uses variable-length blocks.*

Fig. 3 gives an example of the range min tree [15], which is a simplified variant of the range min-max tree. For each chunk, we store the local excess value e, that is, the number of opening parentheses minus the number of closing parentheses in the chunk. We also store the minimum excess value m in the chunk. In addition to these two values, we store the size s of the chunk because chunks have different lengths. The leaves of the range min tree store tuples (e, m, s). An internal node of the range min tree also stores the tuple for the BP sequence made by concatenating those for the children. It is shown [24] that, if the total length of the BP sequence for all the chunks is $m = \mathrm{polylog}(n)$, the range min-max tree can be stored in $\mathrm{O}(m \log\log n/\log n)$ bits and any basic tree operation can be done in

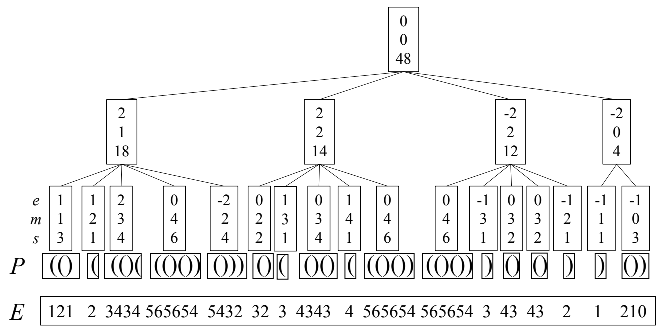


Figure 3 The range min tree of a BP sequence.

constant time. This can be easily extended for the case that chunks are of variable length of $\mathrm{O}(\log n)$. Since the `rank` and `select` operations on V can be simulated by using s on the range min-max tree, we can retrieve micro trees as in Theorem 7.

Another advantage of the indexes in Theorem 7 is that micro trees can be compressed with an arbitrary encoding by the following modified scheme.

▶ **Theorem 9.** *Let $\mu_1, \ldots, \mu_m$ be the original micro trees in the DFS order on the top-tier tree. The function $D(\mu_i)$ represents the code of the micro tree μ_i.*

Suppose we already have the indexes V and S described in Theorem 7. The sparse bitvector V can be replaced with the range min-max tree to support tree navigational queries as in Theorem 8. Then, the storage of P can be replaced with a variable-length cell array of codes $D(\mu_1), \ldots, D(\mu_m)$ supporting random access. Note that F is unnecessary since it can be obtained by checking if μ_i consists of a single node or not.

We can recover P from the indexes: for every micro tree, we can identify where it is placed on P as in Theorem 7, and we can retrieve the micro-tree BP by accessing the corresponding code in a similar way as accessing F in Theorem 7. Also, random access on P can be efficiently supported by this method.

One application of Theorem 9 is optimal compression of unordered trees. The optimal compression of unordered trees needs to fit into $1.564n+\mathrm{o}(n)$ bits [25]. Farzan and Munro [10] achieve this by creating a look-up table T_k that lists all the possible micro-tree shapes with k nodes and encoding each micro tree into its size s and its index o of the table T_s. This technique is compatible with the scheme in Theorem 9, giving another succinct representation of unordered trees with fewer indexes.

Another application is hypersuccinct trees [21], which achieves optimal compression for various tree distributions by encoding micro trees with a Huffman code. Using a Huffman code in Theorem 9, we obtain a practical design of hypersuccinct trees.

3.2 Binary Trees

3.2.1 Extension from Ordinal Trees to Binary Trees

There are several differences to consider when applying the BP representation of tree covering to binary trees. First, the micro trees do not share nodes when the Farzan–Munro algorithm is applied to binary trees. Thus, we do not need to modify tree covering as in Sec. 3.1. Also, as introduced in Sec. 2, the BP representation of a binary tree differs from that of ordinal trees. Thus, how the tree-covering hierarchy appears in the BP sequence should be again investigated.

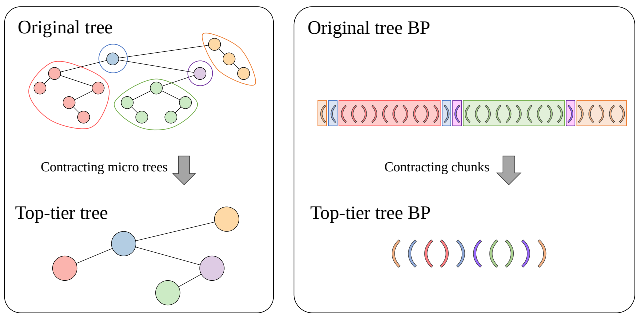


Figure 4 Overview of the relationship between the original tree and the top-tier tree. Each micro tree corresponds to one or two intervals of the BP sequence. Just as contracting micro trees yields the top-tier tree, so contracting chunks into parentheses yields the BP representation of the top-tier tree.

We note the advantages of using the BP representation for binary trees. First, since the representation distinguishes a single left child and a single right child, we do not need to tailor our scheme to full binary trees. Also, we can directly handle queries involving inorder values on the BP sequence, which is especially beneficial when applying to Cartesian trees to implement average-case optimal RMQ data structures.

In the remainder of this section, we first show that every micro tree corresponds to at most two intervals on the BP representation, which enables adopting the data structures in Theorems 7–9 to binary trees. Then we introduce a *split rank*, which corresponds to portals but is more versatile and compact. Finally, we discuss the data-structure design utilizing the observations.

3.2.2 Top-Tier Tree

Here, we discuss that each micro-tree BP appears as one or two intervals in the BP representation of the original tree, and replacing each one or two intervals with a pair of parentheses gives the BP of the top-tier tree. Fig. 4 summarizes the overview.

▶ **Proposition 10.** *Each micro tree corresponds to one or two intervals of the BP representation of the original tree.*

Let us briefly prove Prop. 10. Since a micro-tree BP splits by its child micro trees, we only need to consider the case if it has two child micro trees. Then, Prop. 5 implies it consists of a single node, deriving Prop. 10.

For simplicity, if a micro tree corresponds to a single interval, we virtually split it into two intervals so that every micro tree corresponds to two intervals, which we call *chunks*.

▶ **Definition 11.** *A single interval representing a whole micro tree is split into two intervals: a left interval is defined to be the original interval, followed by a right interval of zero width. Thus, every micro tree corresponds to two intervals, which we call chunks.*

The reason for this extreme division is that it works well with split ranks, which are discussed later. Where to split a single interval does not affect the theorem below.

▶ **Theorem 12** (BP of top-tier tree)**.** *Contracting each left and right chunk into a left and right parenthesis yields the BP of the top-tier tree.*

The theorem follows by focusing on the fact that an edge between a parent p and a child c exists if and only if the opening parenthesis of c is immediately right next to either of the matching parentheses of p. This adjacency is preserved before and after the contraction.

Note that the above theorem defines whether a single child in the top-tier tree is a left- or right-child, which is not discussed in the definition of the top-tier tree. It also shows that the original BP can be obtained by properly replacing pairs of matching parentheses in the top-tier tree BP with the corresponding micro-tree BP sequences.

3.2.3 Split Rank

Theorem 12 enables recovery of the original BP by appropriately replacing each matching pair of parentheses in the top-tier tree BP with the micro-tree BP. To achieve the appropriate replacement, we need to remember where to split each micro-tree BP as in Theorem 9. As for binary trees, we find a versatile and compact value that determines where to cut a micro-tree BP, named a *split rank*. It naturally corresponds to portals and is useful for queries involving inorder, while consuming one bit less than naively storing where to split a micro-tree BP. In addition, storing split ranks consumes only half as much space as storing portals, since only one split rank is needed per micro tree, whereas two portals are needed per micro tree. We start with a lemma that enables more efficient storage than naively maintaining where to split the micro-tree BP.

▶ **Lemma 13.** *Non-empty right chunks begin with closing parentheses.*

This is because, if a right chunk begins with an opening parenthesis, the node corresponding to the opening parenthesis hangs below the child micro tree that splits the micro-tree BP, which is a contradiction. By the above lemma, the number of closing parentheses in the left chunk suffices to recover where to split the micro-tree BP. We define the quantity plus one as a *split rank* so that it corresponds to the portal ranks, as discussed in Prop. 15.

▶ **Definition 14** (Split rank)**.** *The split rank of a micro tree is defined to be the number of closing parentheses in the left chunk plus one.*

Calling `select`$_)$ with the split rank returns the starting index of the right chunk. Assuming that `select` operation returns the last index plus one when searching for a non-existent key, it also works well when the micro-tree BP appears as a single interval in the original tree BP. Also, the split rank corresponds to the rank of the leftmost portal.

▶ **Proposition 15.** *The split rank equals the rank of the portal to the leftmost child if it exists.*

This proposition can be verified by the correspondence between inserting a new leaf at the left portal and inserting a pair of parentheses between the chunks.

3.2.4 Counterparts of Practical Designs

In what follows, we consider the counterparts of Theorems 7–9. In the counterpart of Theorem 7, we do not need F since micro trees are disjoint. Thus, it just marks the starting points of the chunks with V, which is a natural way to store the tree cover. We also note that when V is used to indicate the starting positions, we should not adopt the split positions in Def. 11 since the starting points may collide and V needs to be replaced with a multiset. Here, we can adopt the rule of cutting the intervals at some internal positions so that every chunk has a nonzero width and the start points become distinct.

Theorems 8 and 9 directly apply to binary trees. It is now possible to use zero-width chunks. The data structures can exploit both the BP sequence and the tree-covering structure to efficiently handle tree navigational queries. For example, they support efficient inorder conversion by using the BP sequence, which is conventionally solved by reduction to several queries [8] in tree covering. They also enable efficient depth query that is difficult with the $\mathrm{BP_b}$ representation but is easy if we use the tree-covering structure.

3.2.5 Minimal Design

Here, we present a more efficient design that leverages the hierarchical structure of the BP sequence and the limited use of BP operations due to tailoring to binary trees. To support tree navigational queries, Theorem 9 employs the range min-max tree built on P to support the BP operations on P. However, if we limit the BP operations to ones necessary for the tree navigational queries, we can simplify the data structure. In particular, we can still support the tree navigational queries if we remove the minimum excess value m in the nodes of the range min-max tree built on P, i.e., if we replace the range min-max tree with prefix sum data structures [26].

▶ **Theorem 16.** *Among the data structures in Theorem 9, we can replace the range min-max tree constructed on P with two arrays O and C which stores the number of opening and closing parentheses in each chunk and supports efficient prefix sum queries. More formally, the following data structures are sufficient to support the tree navigational queries that are possible with the BP representation for binary trees.*

- *A variable-length cell array of codes $D(\mu_1), \ldots, D(\mu_m)$ supporting random access.*
- *A data structure that represents the BP sequence S of the top-tier tree. It handles basic operations on the BP sequence S in constant time and spends $\mathrm{O}(n/B)$ bits [24].*
- *Two arrays O and C of the same length as the number of chunks, defined as follows: the i-th element of O and C stores the number of opening and closing parentheses in the i-th chunk, respectively. The arrays O and C support queries involving prefix sum [26].*

There are two main roles of O and C. One is to convert a node, its preorder, and its inorder to one another. The other is to indicate the length of each chunk, which can be obtained by adding the elements of O and C. Note that the values of e and s in each chunk in the range min-max tree can be easily recovered from the values of O and C.

▶ Remark 17. We mainly use the above mechanism in the later implementation, but we modify the mechanism to further reduce the space consumption: it removes the array O and replaces the array C with the array of split ranks in the DFS order of the top-tier tree and sparse sampling of the elements of C. By removing O, we need to give up converting a node and its preorder to each other. On the other hand, since we can find where a micro-tree BP splits on the BP sequence P by using the number of micro-tree nodes and the split rank, we can still recover the length of each chunk. Also, the elements of C can be efficiently recovered from the array of split ranks. In particular, the number of closing parentheses in the left chunk equals its split rank minus one by definition, and that in the right chunk can be obtained by subtracting that of the left chunk from the number of micro-tree nodes. Thus, we can support the conversion of a node and its inorder to each other.

3.2.6 Design Comparison

We briefly note the advantage of designs in Theorems 9 and 16. The advantage of the design in Theorem 16 is that it is space efficient compared to the other design. This is especially crucial when we use a Huffman code to encode micro trees and a small value of B. The advantage of the design in Theorem 9 is that it can be used to represent some subclasses of ordinal trees. The $\mathrm{BP_o}$ sequences of some subclasses coincide with the $\mathrm{BP_b}$ sequences of compressible subclasses of binary trees: LRM-trees [2] of random permutations correspond to Cartesian trees of random permutations and ordinal trees with r leaves correspond to Cartesian trees of arrays with r increasing runs. These subclasses of binary trees can be efficiently compressed with hypersuccinct binary trees. Also, the mechanism in Theorem 9 can easily support some extra BP operations that are difficult with the mechanism in Theorem 16, which do not benefit the $\mathrm{BP_b}$ representation but can be utilized to implement tree navigational queries for ordinal trees using the $\mathrm{BP_o}$ representation. Thus, Theorem 9 can be exploited to represent certain subclasses of ordinal trees while supporting efficient tree navigational queries.

4 Performance Evaluation

In this section, we present the performance evaluation of the RMQ data structures that utilize hypersuccinct binary trees and our representations of tree covering. Although we also implemented tree navigational queries on the hypersuccinct binary trees, we only explain the results regarding RMQ here and omit the detailed results of tree navigational queries; broadly speaking, most tree navigational queries finished in microseconds.

4.1 Optimization of Hypersuccinct RMQ

Here, we consider the space consumption of the data structures. Decoding a Huffman code requires storing a table of the corresponding BP sequences, which consumes $\mathrm{O}(2^{4B}B)$ bits when encoding micro trees. The reason for taking $B = \lceil(\lg n)/8\rceil$ is to suppress the term to $\mathrm{o}(n)$. However, B is significantly small when applied to a tree of practical size: if $n = 10^9$, we have $(\lg n)/8 \sim 3.7$. We can increase B to $(\lg n)/(4 + \varepsilon)$ for an arbitrarily small positive number ε to keep the space complexity asymptotically negligible. However, it is still only up to $B = 7$ for $n = 10^9$. Later numerical experiments also confirm that taking a value of B larger than $(\lg n)/(4 + \varepsilon)$ leads to non-negligible space consumption.

Since we consume $\mathrm{O}(n(\log B)/B)$ bits for storing split ranks and $\mathrm{O}(n/B)$ bits for the top-tier tree BP, taking a small value of B also leads to a significant consumption of space. When choosing B to optimize space consumption, there is a trade-off between terms $\mathrm{O}(2^{4B}B)$ and $\mathrm{O}(n(\log B)/B)$, which are both theoretically negligible but practically significant.

To eliminate the trade-offs between the space-complexity terms, we can employ a *depth-first arithmetic code* [21] instead of a Huffman code, which is a technical tool used to show that the Huffman code achieves optimal space consumption. Although the depth-first arithmetic code takes a longer time to decode and lacks universality, the depth-first arithmetic code does not need a decoding table and achieves optimal space consumption for random permutations. Thus, we can take an arbitrarily large value of B to reduce space consumption. The drawback of taking a large value of B is that the decoding time becomes long: the average code length of each micro tree is $\mathrm{O}(B)$ bits and decoding takes time linear in code length. However, we later experimentally show that it remains efficient if we take a large value of B.

4.2 Implementation Details

We give details of implementation. We implemented the minimal mechanism discussed in Theorem 16 and Remark 17. We used a simple variant of the range min tree with $\mathrm{O}(\log n)$ query time. We employed a Huffman code and a *breadth-first arithmetic code* to encode the micro trees. The breadth-first arithmetic code is a variant of a depth-first arithmetic code, which speeds up the computation of RMQ when combined with branch pruning.

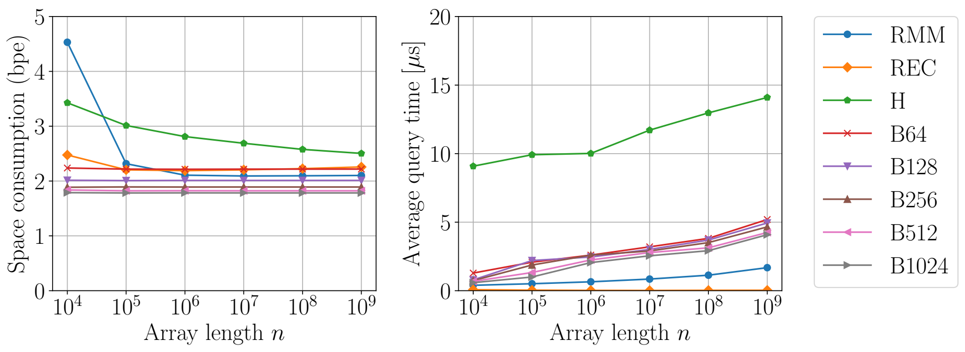


Figure 5 Comparison of the space consumption and the average query time when varying the array length n from 10^4 to 10^9. In the legend, RMM refers to the previous implementation by Ferrada and Navarro [11] and REC refers to the implementation by Baumstark et al. [3]. The other labels indicate our implementations: H means that micro trees are encoded with Huffman codes, and B followed by an integer means that micro trees are encoded with a breadth-first arithmetic code with tree-covering parameter B set to the shown integer.

We implemented the data structures in C++17, compiled it with GCC 11.4.0, and optimized it with the `-O3` option. We evaluated the program on a laptop equipped with an 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz with 16 GB memory.

4.3 Random RMQ Comparison

Here, we compare our implementations with previous implementations by Ferrada and Navarro [11] and Baumstark et al. [3] by benchmarking on random permutations. We varied the length of the permutation from 10^4 to 10^9 by a factor of 10 and measured the space consumption and the average query time of the data structures. We prepared the RMQ data structure using a Huffman code with B set to the integer that minimizes the space consumption, i.e., the nearest integer to $(\lg n)/4$ for $n = 10^4, \ldots, 10^8$ and 8 for $n = 10^9$. We also created the RMQ data structures using a breadth-first arithmetic code with B varying from 64 to 1024 by a factor of 2. Fig. 5 shows the comparison of the space consumption and the average query time. The RMQ data structure using a breadth-first arithmetic code uses a nearly constant space per element, regardless of the number of elements. It is a natural consequence of the data-structure design, as the space complexity is almost linear to n when fixing B. The data structure using a Huffman code is slow, and we consider that this is due to the space optimization technique of packing multiple codes into a single variable-length cell.

We also fixed the length of the array as $n = 10^9$, varied the query width from 2^1 to 2^{29} by a factor of 2, and measured the query time for each query width. We prepared 10^5 random queries for each width. Fig. 6 shows the average query time of each query width. We observe that the graph shapes of breadth-first arithmetic codes are similar even if we vary the tree-covering parameter B. Also, we see that breadth-first arithmetic codes with $B \leq 512$ work faster than Huffman codes while consuming less space. Although breadth-first arithmetic codes have longer query times when query widths are small, if the original array is available, we expect this disadvantage to be compensated by sequentially scanning the original array.

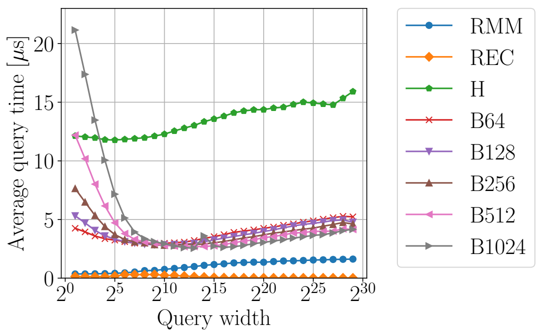


Figure 6 Comparison of the average query time when varying query width. The length of the permutation is set to 10^9 and each point is an average of 10^5 queries. The meaning of each label in the legend is described in the caption of Fig. 5.

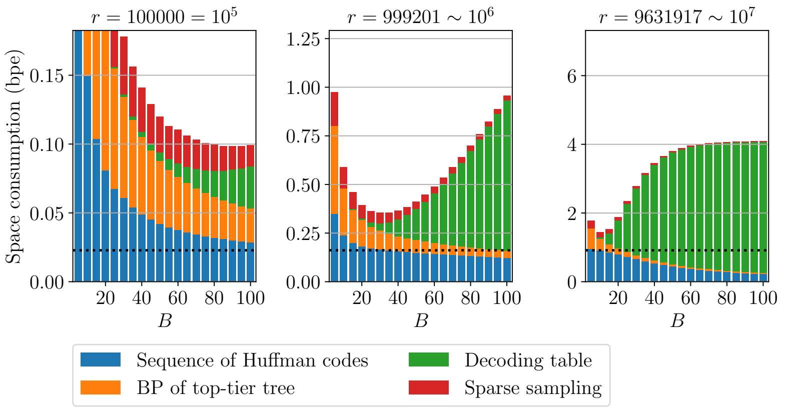


Figure 7 Space consumption breakdown when varying the number of increasing runs r while fixing the array length to $n = 10^8$. Space consumption is expressed as bits per element (bpe). The dotted lines indicate $2\lg\binom{n}{r}/n$, which is the asymptotical upper bound of the space consumption of the Huffman code sequence.

4.4 RMQ with Runs

We evaluate the performance of the RMQ data structure when applied to a permutation with increasing runs. Hypersuccinct binary trees encode the Cartesian tree of a permutation of length n with r increasing runs using $2\lg\binom{n}{r} + o(n)$ bits [21]. We fixed the array length n to 10^8 and varied B from 5 to 100 by steps of 5. We generated an array with approximately r' runs as follows: (1) Shuffle the permutation. (2) Split the permutation into r' parts: different r' splitting points are uniformly sampled from $n-1$ possible splitting positions. (3) Sort each part and concatenate them. The number of runs in the obtained permutation r is possibly fewer than r', but it works well when r' is small.

Setting $r' = 10^5$, 10^6, and 10^7, the generated permutation had 100000, 999201, and 9631917 runs, respectively. We created the RMQ data structures for these permutations and evaluated the memory performance. Fig. 7 shows the results. The right plot of the figure shows that when $r \sim 0.1n$, i.e., when the average length of the run is about 10, the space consumption can be less than $2n$ bits: choosing $B = 10$ results in a space consumption of $1.449n$ bits. When $r \sim 0.01n$, the space consumption takes its minimum value of $0.354n$ bits by choosing $B = 35$. When $r = 0.001n$, the space consumption takes its minimum value of $0.098n$ bits at $B = 95$. Although these values are a few times larger than $2\lg\binom{n}{r}$, they are significantly smaller than $2n$. As for time efficiency, we prepared 10^6 random queries and found that the average query time is 14 µs.

Table 1 Average space consumption and query time of RMQ data structures over the LCP arrays of texts *Pizza&Chilli* corpus [12]. It simulates DFS on the suffix trees by RMQs. The meaning of each label in the leftmost column is described in the caption of Fig. 5.

RMQs	Average space consumption (bpe)				Average query time [ns]			
	dblp	dna	english	sources	dblp	dna	english	sources
RMM	2.102	2.117	2.113	2.168	202	200	203	215
REC	2.229	2.230	2.231	2.242	61	61	63	68
H	2.402	2.428	2.490	2.574	10051	10047	10081	10388
B64	2.517	2.251	2.462	2.496	2324	2323	2260	2344
B128	2.336	2.051	2.300	2.340	2657	2690	2561	2640
B256	2.232	1.936	2.208	2.244	3301	3320	3155	3346
B512	2.174	1.872	2.161	2.192	4657	4642	4390	4676
B1024	2.143	1.835	2.136	2.169	7375	7397	6948	7443

4.5 LCP

As a practical setting, we consider the numerical experiment by Baumstark et al. [3]: DFS on suffix trees by using RMQ data structures over the Longest Common Prefix (LCP) array of real-world texts retrieved from *Pizza&Chilli* corpus [12].

The results are shown in Table 1. Although there is no theoretical guarantee, our average-case optimal RMQ data structures tailored to random permutations spend less than $2n$ bits for the DNA text. On the other hand, our RMQ data structures consume more space for other texts. As for query time, our RMQ implementations are considerably slower than previous implementations, probably due to the large number of queries with small query widths during the traversal.

5 Conclusion

In this paper, we proposed a simple representation of tree covering in the BP sequence for both ordinal trees and binary trees. Utilizing the representation, we presented several efficient designs of succinct data structures for trees. Our designs not only reflect the hierarchy but also isolate micro trees as a sequence in the data-structure design, thus enabling compressing micro trees with an arbitrary encoding. We believe the representation can be widely utilized for designing practically memory-efficient data structures based on tree covering.

We also addressed the implementation of average-case optimal RMQ data structures with hypersuccinct trees. By leveraging our proposed scheme and optimizing the RMQ data structures, our RMQ implementations used less than $2n$ bits and processed queries in a practical time on several settings of the performance evaluation.

Future work includes the method for determining the value of B. We manually adjusted the tree-covering parameter B in the present study, but it was unclear how to appropriately choose B when applied to arrays with increasing runs. Also, the original array may be accessible in some applications, and it would be interesting to design a variant of our work for that case. Implementation and performance evaluation of the versatile mechanism as well as our scheme for ordinal trees is another future issue.

References

1 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In Guy E. Blelloch and Dan Halperin, editors, *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97. SIAM, 2010. doi:10.1137/1.9781611972900.9.

2 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/j.tcs.2012.08.010.

3 Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2017.12.

4 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, December 2005. doi:10.1007/s00453-004-1146-6.

5 Sankardeep Chakraborty, Seungbum Jo, Kunihiko Sadakane, and Srinivasa Rao Satti. Succinct data structures for series-parallel, block-cactus and 3-leaf power graphs. In *COCOA 2021, Tianjin, China, December 17-19, 2021, Proceedings*, volume 13135 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2021. doi:10.1007/978-3-030-92681-6_33.

6 Sankardeep Chakraborty, Seungbum Jo, Kunihiko Sadakane, and Srinivasa Rao Satti. Succinct data structures for small clique-width graphs. In *2021 Data Compression Conference (DCC)*, pages 133–142, 2021. doi:10.1109/DCC50243.2021.00021.

7 Gang Chen, Simon J. Puglisi, and W. F. Smyth. Lempel–ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, June 2008. doi:10.1007/s11786-007-0024-4.

8 Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S Srinivasa Rao. Encoding range minima and range top-2 queries. *Philos. Trans. A Math. Phys. Eng. Sci.*, 372(2016):20130131, May 2014. doi:10.1098/rsta.2013.0131.

9 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11(2):177–189, June 2017. doi:10.1007/s11786-017-0294-4.

10 Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, January 2014. doi:10.1007/s00453-012-9664-0.

11 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017. doi:10.1016/j.jda.2016.09.002.

12 Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM J. Exp. Algorithmics*, 13, February 2009. doi:10.1145/1412228.1455268.

13 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

14 Mordecai J. Golin, John Iacono, Danny Krizanc, Rajeev Raman, Srinivasa Rao Satti, and Sunil M. Shende. Encoding 2d range maximum queries. *Theor. Comput. Sci.*, 609:316–327, 2016. doi:10.1016/j.tcs.2015.10.012.

15 Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM J. Exp. Algorithmics*, 19(1), January 2014. doi:10.1145/2656332.

16 Kou Hamada. zkou-ut/hyperrmq. Software, swhId: swh:1:dir:5d624172e0f2ed764967a13d0dca00d30dc32613 (visited on 2024-07-09). URL: https://github.com/zkou-ut/hyperrmq.

17 Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, December 2014. doi:10.1007/s00453-014-9894-4.

18 Bo-June Paul Hsu and Giuseppe Ottaviano. Space-efficient data structures for top-k completion. In Daniel Schwabe, Virgílio A. F. Almeida, Hartmut Glaser, Ricardo Baeza-Yates, and Sue B. Moon, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, WWW '13, pages 583–594, New York, NY, USA, 2013. International World Wide Web Conferences Steering Committee / ACM. doi:10.1145/2488388.2488440.

19 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

20 John C. Kieffer, Enhui Yang, and Wojciech Szpankowski. Structural complexity of random binary trees. In *2009 IEEE International Symposium on Information Theory*, pages 635–639, 2009. doi:10.1109/ISIT.2009.5205704.

21 J. Ian Munro, Patrick K. Nicholson, Louisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct Trees - New Universal Tree Source Codes for Optimal Compressed Tree Data Structures and Range Minima. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 70:1–70:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2021.70.

22 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

23 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, New York, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

24 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3), May 2014. doi:10.1145/2601073.

25 Richard Otter. The number of trees. *Annals of Mathematics*, 49(3):583–599, 1948. doi:10.2307/1969046.

26 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

27 Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/j.jda.2006.03.011.

28 Dekel Tsur. Representation of ordered trees with a given degree distribution, 2018. doi:10.48550/arXiv.1807.00371.