Succinct Representation of Search Trees on Trees

Seungbum Jo
Chungnam National University, South Korea

Nodari Sitchinava
University of Hawaii at Manoa, United States

Abstract

A search tree on trees (STT) is a data structure for performing a search for a target vertex in a reference tree. A standard binary search tree is a special case of an STT, where the reference tree is a path of totally ordered elements.

In this paper, we study the problem of succinct representation of STTs. We consider two cases: (1) general search trees on trees, and (2) Steiner-closed search trees on trees [Bose et al. TALG 2023]. For both cases, we present representations that can be constructed in polynomial time and achieve optimal space up to the lower-order additive terms.

We also present data structures for supporting fast traversals of both general and Steiner-closed STTs. For general STTs our data structure still takes optimal space up to lower-order additive terms.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Search trees on trees, Encoding data structures, Binary search trees

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2026.3

Related Version *Full Version*: `https://arxiv.org/abs/2609.21236`

Funding *Seungbum Jo*: This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2025-23963814)
Nodari Sitchinava: Supported by NSF grant 2432018.

Acknowledgements We thank Jean Cardinal, John Iacono and Stefan Langerman for helpful discussions.

© Seungbum Jo and Nodari Sitchinava;
licensed under Creative Commons License CC-BY 4.0
37th International Symposium on Algorithms and Computation (ISAAC 2026).
Editors: Lin Chen and Nicole Megow; Article No. 3; pp. 3:1–3:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Search on trees (and more generally *search on graphs*) has been extensively studied under various names in computer science and discrete mathematics, such as elimination trees [2,9,29, 40], ordered-colorings [23], edge rankings [15], vertex rankings [8,14,17], and tubings [12,31]. It is a generalization of binary search on a totally ordered set, which can be viewed as a search on a graph that is a path. (See Section 1.1 for related work on this topic.)

Recently, Bose et al. [10] took a data structuring approach to this problem and presented the *search tree on tree (STT)* data structure, which is a generalization of the classical *binary search tree (BST)*. Given an (unrooted) tree T, referred to as the *reference tree*, the root of the STT $\mathcal{T}$ on T corresponds to some vertex $x \in T$ and its subtrees are STTs on each of the connected components of $T \setminus x$. Therefore, BST $\mathcal{T}$ is a special case of STTs, where $\mathcal{T}$ is built on the reference tree T that is a path – the Hasse diagram of the totally ordered set P.

STT data structures (and their algorithmic analogues for search on trees and graphs) have been of increasing interest recently (see Section 1.1 for references). In this paper, we ask the next natural question in the study of STTs: how efficiently can we encode STTs?

1.1 Related work

Binary search on a totally ordered set is a classical algorithm. BSTs were considered independently by several researchers in the 1950s [25, Sec. 6.2.2], while the first balanced dynamic BST with worst-case $O(\log n)$ height – the AVL tree – was invented in 1962 [1].

The question of generalizing the idea of binary search from totally ordered sets to partially ordered sets (*posets*) was initiated by Linial and Saks [28]. They showed that $\Theta(\log n)$ queries are necessary and sufficient to find a target element in the worst case. Their result can be viewed as a search for a target vertex in a directed acyclic graph (DAG) – the Hasse diagram that represents the partial order relation between the elements of the poset. In this view, the search algorithm queries an oracle to determine which vertex of a directed edge (u, v) is closer to the target vertex t and proceeds in the corresponding subgraph (the subgraph induced by the set of vertices reachable from the vertex returned by the oracle). Hence, the result of Linial and Saks can be viewed as building a search tree on a DAG, with queries being performed on the edges. Alternatively, one can consider the search on DAGs where the queries are performed on the vertices. In this setting, a query on vertex u returns the neighbor v of u that is closer to target t than u.

Linial and Saks [28] showed that, just like in BSTs, the minimum height of a search tree on DAGs is $\Theta(\log n)$ in the worst case. However, unlike BSTs, the problem of finding minimum-height search trees on DAGs is known to be NP-hard [11, 15]. Therefore, the bulk of existing work is focused on search trees on trees rather than DAGs. The minimum-height STTs can be computed in linear time [27, 32, 39, 41], and this restriction for search on trees rather than DAGs is sufficient for many practical applications, such as file system synchronization [3, 32], parallel database queries [30], software testing [3, 32], asymmetric communication protocols [26], assembly planning [20], and mobile agent computing [7].

Search on trees and more generally search on graphs has been studied for decades under various names, such as elimination trees [2,9,29,40], ordered-colorings [23], and tubings [12,31] among others. The minimal height of a search tree on a graph G is also known as the *treedepth* of G [38, Ch. 6] and computing the search tree on trees is equivalent to the problems of *edge ranking* [15] (when queries are on the edges) and *vertex ranking* [8,14,17] (when queries are on the vertices). Because of different names in various research communities, several results have been rediscovered multiple times. For example, linear time algorithms for vertex rankings

was discovered by Schäffer [41] and for edge rankings by Lam and Yue [27] and rediscovered in the context of STTs by Mozes et al. [32] and Onak and Parys [39], respectively.

The question of minimizing the height of a search tree (worst-case optimality) is rather limited. Alternatively, one might ask for a search tree that minimizes the total query time across a given distribution of queries (also known as *optimal BST* [24]). In case of classical BSTs, the shape of such optimal BST can be found in $O(n^2)$ time, using the classic dynamic programming algorithm by Knuth [24]. On the other hand, finding the optimal search tree on graphs with edge queries is NP-hard even for trees of diameter at most 4 [13], although a constant-factor approximation can be computed in linear time [26].

Adaptive STTs. Recently, Bose et al. [10] considered the adaptive version of the STTs (with vertex queries). They proposed an extension of the BST rotation operation to the STT setting. They presented an analog of the Tango trees in the STT setting and showed that it is $O(\log \log n)$-competitive with an optimal STT on a distribution of queries that are not known a priori, i.e., that are produced online. Crucial to achieving this result is the concept of a restricted class of STTs, called *Steiner-closed* STTs, which they showed can be constructed from general STTs with a factor of at most 2 increase in the depth of every node.

Berendsohn and Kozma [6] presented *Splay tree on trees (SplayTT)* – a generalization of splay trees [42] to the STT setting. They showed that SplayTTs achieve static optimality, i.e., are constant-factor competitive with any optimal static STTs on the online sequence of queries. They also conjectured that SplayTTs are *dynamically optimal*, i.e., are constant-factor competitive with any optimal *dynamic* STT on the online sequence of queries, similar to how splay trees are conjectured to be dynamically optimal in the BST setting.

Succinct tree representations. Search trees are widely used as data structures that organize and enable efficient access to data, as seen in applications such as text indexing [36] and information retrieval [34]. One way to access indexed data fast is to store the data within the nodes of the search tree. However, as data grows, the overhead associated with copying and storing it within the nodes of the tree becomes significant. Instead, one may want to store the shape of the search tree and access the data indexed by the tree directly from the original storage medium. This motivates space-efficient representations of tree structures that support efficient navigation while using minimal additional space.

Considering only the tree structure and ignoring the labels, a search tree can be represented as an ordered tree. A naive approach to storing an ordered tree with n nodes is to use $O(n)$ pointers, which requires $\Omega(n \log n)$ bits[1]. However, the information-theoretic space lower bound for storing such trees is $2n - O(\log n)$ bits. Thus, research in this area mainly focuses on reducing the gap between the naive representation and the information-theoretic lower bound. Since Jacobson's LOUDS representation [21], various *succinct* representations (i.e., representations of size $2n + o(n)$ bits) have been proposed for ordered and unlabeled trees, enabling efficient support for tree navigation queries (see [35] for earlier results on succinct tree representations). One of the most recent succinct representations, proposed by Navarro and Sadakane [37], supports a wide range of tree navigation queries in $O(1)$ time (see Table 1 in [37]).

Binary search trees (BSTs) can be considered as ordered and labeled trees where each node is labeled with a key from the totally ordered set, and all keys in the left (resp. right) subtree are smaller (resp. larger) than the key at the node. The data structure of Navarro

[1] We assume a word-RAM model with a word size of $\Theta(\log n)$ bits. All our logarithms are of base 2.

and Sadakane [37] provides a $(2n + o(n))$-bit representation of a static BST, supporting traversal in $O(1)$ time per node, as the label of any node can be obtained from its inorder number, while the keys are stored in a separate array, not counted in the space of BST.

1.2 Our results

In this paper, we study the problem of storing STTs using small space. Specifically, given a reference tree T with n vertices, which is accessible during decoding, our goal is to obtain a space-efficient representation of the STT $\mathcal{T}$ on T. Since we assume that T is accessible during decoding, we do not account for the space to store it. We show that we can significantly improve on the naive $O(n \log n)$-bit encoding of $\mathcal{T}$:

1. In Section 4, we present an $(\lceil n \log \ell \rceil + 2n)$-bit representation for $\mathcal{T}$, where ℓ denotes the number of leaves in T. Moreover, in the full version of this paper [22], we show that there exists T that requires at least $n \log \ell - O(\ell \log (n/\ell))$ bits to encode an STT on T, i.e., our representation is optimal up to $O(n)$ additive terms in general, and up to the *lower-order* additive terms when $\ell = \omega(1)$.
2. In Section 5 we show that if $\mathcal{T}$ is a Steiner-closed STT on T (see Section 2 for the formal definition), there exists a $(2n + \lceil \log n \rceil)$-bit representation of $\mathcal{T}$. Furthermore, in the full version of this paper [22], we prove that any Steiner-closed search tree on T requires at least $2n - O(\log n)$ bits, i.e., our representation is optimal up to lower-order additive terms. In particular, the upper and lower bounds coincide with those for BSTs on arrays of size n, up to lower-order additive terms. Since every STT can be transformed into a Steiner-closed STT at a cost of only doubling the height [10], i.e., without asymptotically affecting search time, the more efficient representation of Steiner-closed STTs is of particular interest.

Both of the above representations can be constructed in polynomial time. However, they do not support traversal of $\mathcal{T}$ efficiently. Instead, in the full version of this paper [22], we present additional representations of general and Steiner-closed STTs that support traversal from the current node to its i-th child, where the current node is identified by its preorder number, in the following time bounds:

1. An $(n \log \ell + O(n) + o(n \log \ell))$-bit representation that supports access to the i-th child in $O(\log \log \ell)$ time, and
2. An $O(n)$-bit representation that supports access to the i-th child in $O(1)$ time when $\mathcal{T}$ is Steiner-closed.

Note that, while the first of these two representations achieves optimal space up to lower order terms, the second one is optimal only asymptotically, because only $2n - O(\log n)$ bits are necessary to represent Steiner-closed STTs.

The rest of the paper is organized as follows. Section 2 introduces the necessary notations. Section 3 presents a high-level overview of the techniques used to obtain the results. Sections 4 and 5 present the succinct representations of general STTs and Steiner-closed STTs, respectively.

2 Preliminaries

Table 1 summarizes the notations used throughout the paper.

Symbol	Description	Symbol	Description
$T(v)$	The subtree of T rooted at v.	$CH_G(S)$	Convex hull of the vertex set S in G.
$G[S]$	Induced subgraph of G by the vertex set S.	$\mathsf{lca}_T(u, v)$	The lowest common ancestor (LCA) of u and v in the rooted tree T.
$P_G(u, v)$	The path from u to v in G.	$\mathsf{depth}_T(v)$	The depth of v in the rooted tree T.
$\mathsf{preorder}_T(v)$	The preorder number of v in the ordered tree T.	$\mathsf{inorder}_T(v)$	The inorder number of v in the binary tree T.

Table 1 Summary of the main notations used in this article

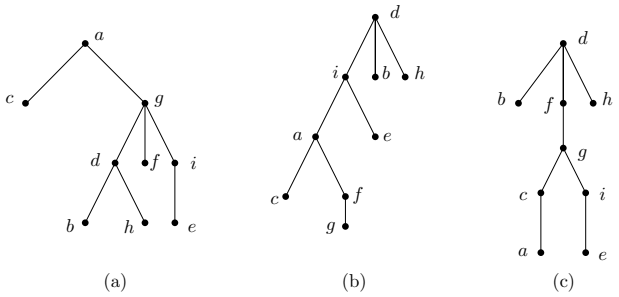


Figure 1 (a) Reference tree T, (b) search tree $\mathcal{T}$ on T, and (c) Steiner-closed search tree $\mathcal{T}$ on T.

Search trees on trees. Given a labeled tree $T = (V, E)$ with n vertices, a *search tree* $\mathcal{T}$ *on* T is defined as a rooted tree on T satisfying the following conditions:

- The root node r of $\mathcal{T}$ corresponds to an arbitrary vertex of T.
- The rooted subtrees of $\mathcal{T} \setminus r$ are search trees on the connected components of $T \setminus r$.

We assume the labels of T are totally ordered and unique, and given the label of a node u, we are able to access u and all relevant information stored at u in $O(1)$ time. For example, the label can be the memory address of where u is stored.

By definition, T and its search tree $\mathcal{T}$ share the same set of nodes V. In the rest of the paper, to avoid ambiguity, we use the term *vertex* for elements of T and *node* for elements of $\mathcal{T}$. Furthermore, we refer to the vertex (resp., node) u as the vertex (resp., node) with label u. In general, T is an unrooted tree, and $\mathcal{T}$ is defined as an unordered rooted tree. In this paper, however, we assume that T is a rooted and ordered tree (the root of T and the ordering of siblings are defined in later sections). Moreover, we treat $\mathcal{T}$ as an ordered rooted tree by defining, for any non-leaf node d, the sibling order according to the *smallest* preorder numbers among the vertices in the corresponding connected components of $T \setminus \{d\}$.[2] For example, in T and $\mathcal{T}$ shown in Figure 1(a) and (b), respectively, the node i is the leftmost sibling among the children of d, because the connected component of T that corresponds to the subtree of $\mathcal{T}$ rooted at i contains vertex a, which has the smallest preorder number among all vertices in T.

As a special case of a search tree on T, Bose et al. [10] introduced the notion of a *Steiner-closed search tree* on T, defined as follows. Given a subset of vertices $S \subseteq V$ of a

[2] In general, the preorder traversal of an ordered tree visits the root r first and then recursively performs preorder traversal on the subtrees rooted at the children of r from left to right.

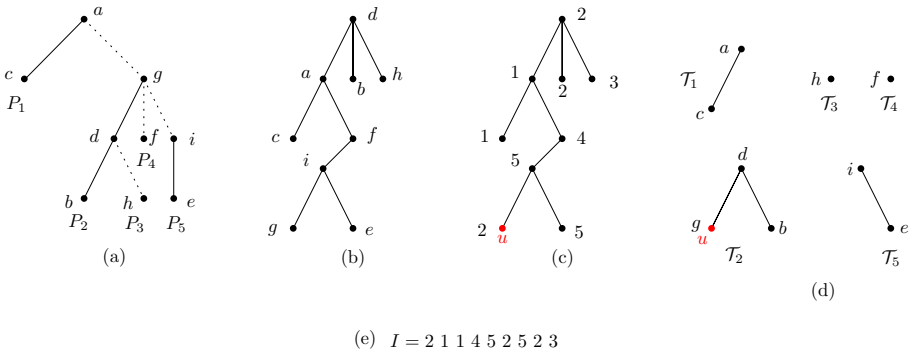


Figure 2 (a) A tree T with paths $P_1, \ldots, P_5$ (dotted lines indicate edges that are not part of any path in P), (b) a search tree $\mathcal{T}$ on T, (c) $\mathcal{T}'$, (d) the tree structures of $\mathcal{T}_1, \ldots, \mathcal{T}_5$, (e) the array I, which stores the labels of $\mathcal{T}'$ in preorder (this is used in the representation of Theorem 3).

graph G, the *convex hull* $CH_G(S)$ is the set of all vertices lying on the paths between any pair of vertices $u, v \in S$ in G. The set S is called *Steiner-closed with respect to* G if and only if every vertex in $CH_G(S) \setminus S$ has degree exactly two in the subgraph of G induced by $CH_G(S)$. A search tree $\mathcal{T}$ on T is Steiner-closed if and only if, for every node v, the set of nodes along the path from the root to v in $\mathcal{T}$ forms a Steiner-closed set with respect to T. For example, the tree in Figure 1(b) is a search tree on the tree shown in Figure 1(a), but it is *not* a Steiner-closed STT, because the set of nodes on the path from d to a in $\mathcal{T}$, i.e., $S = \{a, d, i\}$, does not form a Steiner-closed set with respect to T, as it omits the node g, which has degree 3 in the subgraph of T induced by $CH_T(S) = \{a, d, g, i\}$.

Supporting traversal of $\mathcal{T}$. To support the traversal of $\mathcal{T}$, we identify the current node by its preorder number k in $\mathcal{T}$. The search starts at the root, for which $k = 1$. Given k, our $\mathsf{access}(k)$ operation returns the label u of the corresponding node. The oracle $\mathcal{O}$, given the search key and u, returns the index i of the child from which the search should proceed. For example, when T is a path and $\mathcal{T}$ is a BST, the oracle query is the comparison query that returns the left child ($i = 0$) or the right child ($i = 1$) of u from which the search should proceed. Using the constant-time navigation operations of Navarro and Sadakane [37], we then obtain the preorder number k' of the i-th child of the current node. We continue the search with k' as the preorder number of the current node. Thus, the remaining problem is to support $\mathsf{access}(k)$, which maps the preorder number of a node in $\mathcal{T}$ to its label.

3 Technical Overview

In this section we present a high-level overview of our techniques.

Succinct representations of general STTs. Given a reference tree T with ℓ leaves (recall that we treat both T and $\mathcal{T}$ as rooted and ordered trees), we start by decomposing T into ℓ ancestor-to-leaf disjoint paths $P_1, \ldots, P_\ell$. This can be achieved via a preorder traversal of T.

Based on these paths, our representation of $\mathcal{T}$ is a labeled tree $\mathcal{T}'$ that has the same structure as the STT $\mathcal{T}$, where each node of $\mathcal{T}'$ is labeled by the index of the path that contains its corresponding vertex in T. Since there are at most ℓ distinct path labels, $\mathcal{T}'$ can be represented using $\lceil n \log \ell \rceil + 2n$ bits. Moreover, since $P_1, \ldots, P_\ell$ are ancestor-to-leaf paths, we can construct $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ (for each $i \in [\ell]$, $\mathcal{T}_i$ is constructed from the nodes of $\mathcal{T}$

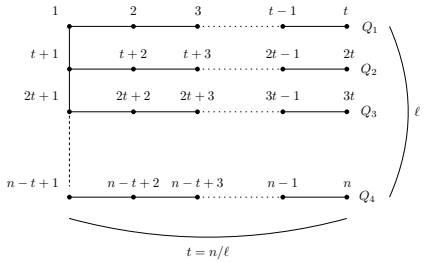


Figure 3 A comb-shaped tree used in proving the space lower bound for general STTs.

whose corresponding vertices lie in P_i) directly from T and $\mathcal{T}'$ without storing them explicitly. Hence, the size of the representation is $\lceil n \log \ell \rceil + 2n$ bits in total.

Using this representation, we can fully reconstruct $\mathcal{T}$: $\mathcal{T}'$ provides the tree structure of $\mathcal{T}$, and from the label of each node in $\mathcal{T}'$, we can identify its corresponding node in the binary trees $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$, whose labels can in turn be computed from T. For details of the representation and the reconstruction procedure, see Section 4.

Finally, we prove the space lower bound for storing $\mathcal{T}$ by showing that a *comb-shaped tree* (see Figure 3) requires $n \log \ell - O(\ell \log(n/\ell))$ bits to be represented by counting the number of possible STTs on such a tree. Details are presented in the full version of this paper [22].

Succinct representations of Steiner-closed STTs. When $\mathcal{T}$ is a Steiner-closed STT, the Steiner-closed property of $\mathcal{T}$ allows us to find a specific decomposition of T into ancestor-to-leaf disjoint paths $P_1, \ldots, P_\ell$, such that each set of vertices V_i that forms path P_i also forms a connected component $\mathcal{T}[V_i]$ in $\mathcal{T}$ (such decomposition might not exist in general STTs). The fact that $\mathcal{T}[V_i]$ is a connected component in $\mathcal{T}$ allows us to reconstruct $\mathcal{T}$ only from the representations of $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ (the representations of the paths P_i) without the need for $\mathcal{T}'$. Instead it is sufficient to maintain a data structure that can answer *Range Minimum Queries (RMQ)* on a sequence of length ℓ that captures the information about how the paths $P_1, \ldots, P_\ell$ are formed from T. Because the paths $P_1, \ldots, P_\ell$ and their corresponding binary search trees $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ form connected subgraphs in both T and $\mathcal{T}'$, respectively, the answers to the RMQ queries provide a guide for merging $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ into a single tree $\mathcal{T}$.

We store the tree structures of $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ using only $2n - 2\ell$ bits in total, while the RMQ data structure requires only 2ℓ bits [5,43]. As a result, we obtain a $(2n + \lceil \log n \rceil)$-bit encoding of $\mathcal{T}$ (where the additional $\log n$ bits are used to store the root of $\mathcal{T}$). For details of the representation and the reconstruction procedure, see Section 5.

Finally, in the full version of this paper [22], we show that this is tight up to lower-order terms by observing that every STT on a path is Steiner-closed, and that STTs on a path of size n are in one-to-one correspondence with BSTs on an ordered set of the same size.

Supporting efficient traversals via access(k) queries. Consider the following procedure for performing the search for some key κ on STT $\mathcal{T}$. We identify the current node by its preorder number k in $\mathcal{T}$, starting from the root. Given k, access(k) returns the label u of the current node. The query oracle takes u together with the target key κ and determines the index i of the child of u in $\mathcal{T}$ from which the search should proceed. Thus, we would like to support efficient traversal from the current node to its i-th child, where the current node is identified by its preorder number.

The succinct representation of Navarro and Sadakane [37] supports a number of operations

on static trees in $O(1)$ time, including the operation of computing the preorder number k' of the i-th child of the node with preorder number k. Therefore, after each traversal step, it remains to obtain the label of the node whose preorder number in $\mathcal{T}$ is k'. We call such a query $\mathsf{access}(k)$.

Clearly, storing the mapping from each preorder number to the corresponding label naively would require $\Omega(n \log n)$ space. Instead, we describe space-efficient data structures that support answering $\mathsf{access}(k)$ queries on general and Steiner-closed STTs. When $\mathcal{T}$ is a general STT, the data structure is relatively simple: it stores a succinct representation of $\mathcal{T}$ together with $O(n)$-bit auxiliary structures that support efficient navigation on the trees in the representation (namely, $\mathcal{T}'$ and $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$).

However, there are additional challenges in obtaining an $O(n)$-bit data structure that supports access efficiently for a Steiner-closed STT $\mathcal{T}$. The main challenge lies in computing, for a query node, the index of the path that contains its corresponding vertex efficiently. In the general STT case, this index can be obtained directly from $\mathcal{T}'$, but $\mathcal{T}'$ cannot be stored using only $O(n)$ bits.

Instead, we introduce the *extended search tree* $\mathcal{T}^e$ of $\mathcal{T}$, which is constructed from $\mathcal{T}$ by adding at most n *copied nodes*. In addition to all core properties of $\mathcal{T}$, $\mathcal{T}^e$ has one extra property that enables us to compute the path indices efficiently: the relative order of (the first) occurrences of paths $P_1, \ldots, P_\ell$ in the preorder traversal of T is the same as the relative order of (the first) occurrences of paths $P_1, \ldots, P_\ell$ in the preorder traversal of $\mathcal{T}^e$. This property does not hold in Steiner-closed STTs in general.

After determining the path index j containing the corresponding vertex of the query node, we compute the two endpoint vertices of P_j in $O(1)$ time using $O(n)$ auxiliary bit sequences. Finally, we compute the label of the query node in $O(1)$ time by (1) locating the query node within the tree $\mathcal{T}_j$ using $\mathcal{T}^e$, and (2) computing the label of the node in $\mathcal{T}_j$ from P_j. Details of the data structures are presented in the full version of this paper [22].

4 Succinct Representation of General Search Trees on Trees

Consider a tree $T = (V, E)$ with n vertices and ℓ leaves. In this section we treat T as a rooted and ordered tree by assuming that (1) the root r of T is the vertex with the smallest label, and (2) for each non-leaf vertex $u \in T$, the children of u are ordered from left to right according to their labels. Since the labels are totally ordered, we can compute the preorder number of any vertex in T without using extra space.

The main idea of the representation of the STT $\mathcal{T}$ on T is to first decompose T into ℓ distinct paths, where each path connects a leaf vertex to one of its ancestors in T. For each path, we construct an STT on that path from the structure of $\mathcal{T}$. Since each such search tree can be viewed as a standard BST on an array (see Lemma 1), we exploit key structural properties of $\mathcal{T}$ from Lemma 2 to design a representation. In particular, we encode $\mathcal{T}$ by storing the tree structure of $\mathcal{T}$ (together with the index of the path containing each node) and the search trees on the paths. This representation uses $\lceil n \log \ell \rceil + 2n$ bits in total, and in the full version of this paper [22], we show that this representation is optimal up to an additive $O(n)$ term.

We now proceed to describe the representation in detail. We first prove a lemma showing that if T is a path, then any search tree on T can be treated as a standard BST.

▶ **Lemma 1.** *There exists a $(2n-2)$-bit representation of any search tree $\mathcal{T}$ on a path T.*

Proof. For $n = 1$, T and $\mathcal{T}$ are identical, so there is no need to maintain $\mathcal{T}$ separately. For $n > 1$, root T at an endpoint and order its vertices by depth; then any STT $\mathcal{T}$ on T is a BST

whose inorder order coincides with the preorder order of T; a single child is treated as the left or right child according to this order. Applying the first-child–next-sibling representation and BP (balanced parentheses sequence) [33] encoding gives $2n + 2$ bits, and omitting the first two opening and last two closing parentheses yields $2n - 2$ bits. ◀

Next, we describe how to decompose T into ℓ disjoint paths $P_1, \ldots, P_\ell$. For $i \in [\ell]$, let l_i be the i-th leaf vertex of T (leaves are ordered by their preorder numbers in T). First, define $P_1 = P_T(r, l_1)$, and for $i \in \{2, \ldots, \ell\}$, define P_i as the induced subgraph of T consisting of the vertices in $P_T(r, l_i) \setminus \left(\cup_{j=1}^{i-1} P_j\right)$. By construction, the sets of vertices on the paths $P_1, \ldots, P_\ell$ form a partition of V. The following lemma characterizes how the vertices in each path appear in $\mathcal{T}$.

▶ **Lemma 2.** *For any vertices $u, v \in P_i$, the following statements hold:*

1. *$\mathcal{T}(u)$ has at most two child subtrees that contain at least one node in P_i.*
2. *Let $\mathcal{T}(u_l)$ and $\mathcal{T}(u_r)$ be the two non-empty child subtrees of $\mathcal{T}(u)$ each containing at least one node in P_i. If u_l is a left sibling of u_r, then $\mathcal{T}(u_l)$ (resp., $\mathcal{T}(u_r)$) contains all nodes $u' \in P_i \cap \mathcal{T}(u)$ with $\mathsf{depth}_T(u') < \mathsf{depth}_T(u)$ (resp., $\mathsf{depth}_T(u') > \mathsf{depth}_T(u)$).*
3. *$\mathsf{lca}_{\mathcal{T}}(u, v) \in P_i$.*

Proof. By the construction of P_i, the vertices in P_i have distinct depths in T, which directly implies statement 1 from the definition of a search tree on T. Statement 2 also follows from the definition of the ordering among siblings in $\mathcal{T}$. For any vertex $x \in P_i$, there are at most two connected components in $T \setminus \{x\}$ that contain at least one vertex in P_i: C^u, the connected component containing all proper ancestors of x, and C^d, the connected component containing all proper descendants of x. Moreover, the smallest preorder number in C^d is greater than the preorder number of x in T. Thus, the subtree of $\mathcal{T}$ corresponding to C^u must appear as the left subtree of the subtree of $\mathcal{T}$ corresponding to C^d. To prove statement 3, suppose the node $z = \mathsf{lca}_{\mathcal{T}}(u, v)$ lies in P_j with $i \neq j$. Since $z \neq u$ and $z \neq v$, the vertices u and v must belong to distinct connected components of $T \setminus \{z\}$, implying $z \in P_j \cap P_T(u, v)$. This contradicts the fact that P_i and P_j are disjoint, as $P_T(u, v)$ is a subgraph of P_i from the construction. ◀

Note that the paths $P_1, \ldots, P_\ell$ can be identified by a single preorder traversal of T (i.e., a lexicographical DFS from the vertex r): during the traversal, whenever we visit an adjacent vertex of a vertex v other than the first one, we create a new path. Next, for each $i \in [\ell]$, we define $\mathcal{T}_i$ as follows (see Figure 2(d) for an example): (1) for any leaf node l, pick the edge $e = (l, u)$ in $\mathcal{T}$ where u is the parent of l; (2) if both l and u are in P_i, keep the edge; otherwise, contract the edge and assign the label of the merged node to either l or u (whichever belongs to P_i; if neither does, use l); (3) repeat steps (1) and (2) recursively until all nodes are in P_i. Then, by Lemma 2, $\mathcal{T}_i$ is a binary search tree on P_i, and given the tree structures of $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$, their labels can be decoded by referring to P_i using Lemma 1. As in the proof of Lemma 1, if a node in $\mathcal{T}_i$ has a single child, we treat that child as either the left or right child of its parent, depending on their preorder numbers in T.

The representation of $\mathcal{T}$ is a single tree $\mathcal{T}'$, obtained from $\mathcal{T}$ by replacing each node's label with the index of the path it belongs to. We store the tree structure of $\mathcal{T}'$ using BP, requiring $2n$ bits, and an array I that stores the labels of $\mathcal{T}'$ in preorder. Using the representation of Dodis et al. [16], I takes $\lceil n \log \ell \rceil$ bits, so the total space for (2) is $\lceil n \log \ell \rceil + 2n$ bits.

To decode the label of a node $u \in \mathcal{T}$ from the representation, we first identify the path P_i that contains u using $\mathcal{T}'$. Then, we reconstruct P_i from T and $\mathcal{T}_i$ from $\mathcal{T}'$. The label of u is obtained from P_i and $\mathcal{T}_i$ using Lemma 1.

Finally, we analyze the construction time of the representation. We first construct a perfect hash table [19] that stores the mapping from the labels of the nodes in $\mathcal{T}$ to their position in $\mathcal{T}$. This can be done in $O(n)$ time by traversing $\mathcal{T}$ once. Next, as described earlier, we identify the paths $P_1, \ldots, P_\ell$ in $O(n)$ time by a single preorder traversal of T. During this traversal, we can also construct $\mathcal{T}'$ by locating, for each visited vertex of T, its corresponding node in $\mathcal{T}$ in $O(1)$ time using the hash table. Moreover, a BP of any tree of size s can be built in $O(s)$ time, which implies that the bit string containing the BP of $\mathcal{T}'$ can be constructed in $O(n)$ time. Finally, the array I can be constructed in $O(n)$ time after constructing $\mathcal{T}'$ via the preorder traversal. We summarize the result in the following theorem.

▶ **Theorem 3.** *Given a tree T with n vertices and ℓ leaves, any search tree $\mathcal{T}$ on T can be encoded using at most $\lceil n \log \ell \rceil + 2n$ bits. The representation can be constructed in $O(n)$ time.*

▶ **Example 4.** Consider how to decode the label of the node u in Figure 2(c) using the representation of Theorem 3. Given that u is labeled 2 in $\mathcal{T}'$, we reconstruct P_2 from T, and the tree structure of $\mathcal{T}_2$ from $\mathcal{T}'$. Since u is the first node in the inorder traversal of $\mathcal{T}_2$, the answer is the label of the first node in P_2 under the preorder traversal of T, which is g.

5 Succinct Representation of Steiner-Closed Search Trees on Trees

In this section we present a representation of a Steiner-closed search tree $\mathcal{T}$ on $T = (V, E)$ with n vertices and ℓ leaves. First, we treat T as a rooted and ordered tree as follows: (1) the root r of T is set to be the root of $\mathcal{T}$, and (2) the siblings in T are ordered according to their labels. Therefore, by using $O(\log n)$ additional bits to indicate r, we can regard T as a rooted and ordered tree.

The overall idea of the representation follows a strategy similar to the representation of Theorem 3, where we decompose T into ℓ distinct paths $P_1, \ldots, P_\ell$ and construct the search trees $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ on these paths by contracting edges of $\mathcal{T}$. However, unlike Theorem 3, where the paths are defined independently of $\mathcal{T}$, here each path is determined based on the structure of $\mathcal{T}$. In Lemma 8, we show that when $\mathcal{T}$ is Steiner-closed, the subgraph of $\mathcal{T}$ induced by the nodes in P_i forms a connected subgraph. Using this property, we show that $\mathcal{T}$ can be reconstructed from $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ without storing the path index of each node in $\mathcal{T}$, which is required in the representation of Theorem 3. Before introducing the detailed decomposition procedure, we first present the following proposition, which shows that, when $\mathcal{T}$ is a Steiner-closed STT, the sibling order in $\mathcal{T}$ coincides with the order induced by the preorder numbers of the corresponding vertices in T.

▶ **Proposition 5.** *For any non-root node u and its left sibling v in a Steiner-closed STT $\mathcal{T}$ of T, $\mathsf{preorder}_T(v) < \mathsf{preorder}_T(u)$.*

Proof. Suppose that v is a left sibling of u in $\mathcal{T}$, but $\mathsf{preorder}_T(v) > \mathsf{preorder}_T(u)$, and let w be the parent of u and v in $\mathcal{T}$. Let v' be the node in $\mathcal{T}(v)$ with the smallest preorder number in T. Since v is the left sibling of u, the definition of the sibling ordering in STTs implies that $\mathsf{preorder}_T(v') < \mathsf{preorder}_T(u)$. Hence, $v' \neq v$, and u cannot be an ancestor of v in T (otherwise, $v' \notin \mathcal{T}(v)$). Furthermore, the two vertices $z = \mathsf{lca}_T(u, v)$ and $z' = \mathsf{lca}_T(v, v')$ satisfy that z' is an ancestor of z in T. Otherwise, suppose that z' is a proper descendant of z. Then there are two cases. If $\mathsf{preorder}_T(u) < \mathsf{preorder}_T(z')$, then $\mathsf{preorder}_T(u) < \mathsf{preorder}_T(z') < \mathsf{preorder}_T(v')$, contradicting the assumption that $\mathsf{preorder}_T(v') < \mathsf{preorder}_T(u)$. On the other hand, if

$\mathsf{preorder}_T(u) > \mathsf{preorder}_T(z')$, then $\mathsf{preorder}_T(u)$ is greater than the preorder numbers of all vertices in $T(z')$, and in particular, $\mathsf{preorder}_T(u) > \mathsf{preorder}_T(v)$, contradicting the assumption that $\mathsf{preorder}_T(u) < \mathsf{preorder}_T(v)$. Moreover, $w \neq z$ and $w \in P_T(z, u)$ (otherwise, from the properties of z and z', $v' \notin \mathcal{T}(v)$), which implies that $z \in CH_T(\{r, v, w\})$.

Now let H be the set of nodes on the path $P_{\mathcal{T}}(r, v)$. Then $v, w \in H$, but $z, z' \notin H$ (otherwise, $v' \notin \mathcal{T}(v)$). Thus, $CH_T(H) \setminus H$ contains the vertex z, which has degree at least 3 in the subtree of T induced by $CH_T(H)$. This contradicts the assumption that $\mathcal{T}$ is Steiner-closed. ◀

Decomposition of T into paths. As described earlier, we decompose T into ℓ distinct paths $P_1, \ldots, P_\ell$, which are defined based on the structure of $\mathcal{T}$. The decomposition procedure consists of two main steps: (i) *Initial decomposition*, and (ii) *Modification procedure*. Briefly, the initial decomposition partitions T into paths by traversing $\mathcal{T}$ in preorder of $\mathcal{T}$, and the modification procedure refines these paths so that each path contains exactly one leaf of T and all vertices in a path have distinct depths, as in the paths defined in Section 4.

The initial decomposition is performed as follows. First, we initialize the vertex sets $V_1 = V_2 = \cdots = V_\ell = \emptyset$, and traverse $\mathcal{T}$ in preorder. Whenever we traverse the node $u \in \mathcal{T}$, we find the smallest index $i \in [\ell]$ such that the induced subgraph of T on $CH_T(V_i \cup \{u\})$ is a path and does not include any vertices belonging to sets other than V_i, and add u to V_i. After the traversal, we define $P_i = T[V_i]$. By construction, $V_1, \ldots, V_\ell$ form a partition of V. The following three lemmas show that the initial decomposition is well-defined and that each $\mathcal{T}[V_i]$ forms a connected subgraph of $\mathcal{T}$ (the proofs are presented in the full version of this paper [22]).

▶ **Lemma 6.** *After the initial decomposition, for every $i \in [\ell]$: P_i is a path in T.*

▶ **Lemma 7.** *After the initial decomposition, for every path P_i in T, every vertex in P_i that is not a leaf in T has at least one child in T that also lies in P_i. Consequently, P_i contains at least one (and at most two) leaves of T for all $i \in [\ell]$.*

▶ **Lemma 8.** *$\mathcal{T}[V_i]$ is a connected subgraph of $\mathcal{T}$ for all $i \in [\ell]$.*

After the initial decomposition, we apply the following modification procedure to the paths to ensure that (a) P_i contains exactly one leaf node l_i of T (recall that l_i is the i-th leaf in T, ordered by its preorder number in T), and (b) all vertices in P_i have distinct depths in T, as in the decomposition of Section 4.

We decompose each path P_i that contains two leaves of T into two paths as follows. Let M_L and M_R be the sets of vertices of P_i in the left and right subtrees of $T(h_i)$, respectively, where h_i is the LCA of the two leaves. Let u be the first node in V_i according to the preorder traversal of $\mathcal{T}$. If $u = h_i$, we arbitrarily assign h_i to V_L. Thus, if $u = h_i$ or $u \in M_L$, we define $V_L = M_L \cup \{h_i\}$ and $V_R = M_R$. Otherwise, we define $V_L = M_L$ and $V_R = M_R \cup \{h_i\}$.

Finally, we decompose P_i into two paths: $T[V_L]$ and $T[V_R]$. Once all paths contain only one leaf, we relabel them so that each P_i denotes the path in T containing l_i.

We now claim that in each decomposition each $\mathcal{T}[V_L]$ and $\mathcal{T}[V_R]$ are connected subgraphs of $\mathcal{T}$, thus, maintaining the property of Lemma 8. Suppose not, and without loss of generality assume that $u \in V_L$. If $u = h_i$, the claim holds immediately. Otherwise, there exist a descendant $v \in V_R$ of u in $\mathcal{T}$ and, without loss of generality, let v be the node with the minimum preorder number in $\mathcal{T}$ among all nodes in V_R. If $P_{\mathcal{T}}(r, v)$ contains h_i, then h_i must be the last node in V_L according to the preorder traversal of $\mathcal{T}$; otherwise, by the definition

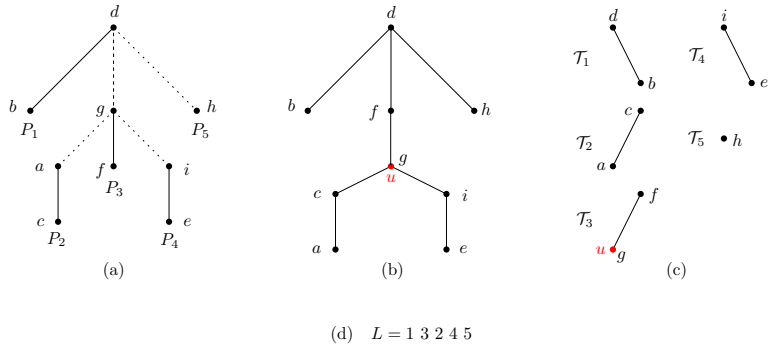


Figure 4 (a) A T with paths $\{P_1, \ldots, P_5\}$ (dotted lines indicate edges that are not part of any path in P), (b) Steiner-closed search tree $\mathcal{T}$ on T, (c) the shapes of $\mathcal{T}_1, \ldots, \mathcal{T}_5$, (d) an array L. The representation of $\mathcal{T}$ given in Theorem 10 consists of the shapes of $\mathcal{T}_1, \ldots, \mathcal{T}_5$ (without the labels), the RMQ encoding of L, and the position of the root node in T (T is initially an unrooted tree).

of T, $\mathcal{T}[V_L]$ would contain no nodes from V_R. In this case, both $\mathcal{T}[V_L]$ and $\mathcal{T}[V_R]$ are connected subgraphs of T. Otherwise, $P_{\mathcal{T}}(r, v)$ contains u but not h_i, which contradicts the Steiner-closed property of $\mathcal{T}$ (recall that r is also the root of T). Furthermore, by Lemma 7, both $T[V_L]$ and $T[V_R]$ contain exactly one leaf of T.

▶ **Example 9.** Consider a tree T and its Steiner-closed search tree $\mathcal{T}$ in Figure 4(a) and (b), respectively. We first define the root of T as d, which is a root of $\mathcal{T}$. The initial decomposition decomposes T into five disjoint paths: $V_1 = \{d, b, f, g\}$, $V_2 = \{c, a\}$, $V_3 = \{i, e\}$, $V_4 = \{h\}$, and $V_5 = \emptyset$. Next, the modification procedure partitions P_1 into two paths: one containing $\{d, b\}$ and the other containing $\{f, g\}$, and relabels the paths $P_1 \ldots, P_5$, as shown in Figure 4(a).

Representation of $\mathcal{T}$. We define $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ as the trees derived from the paths $P_1, \ldots, P_\ell$, respectively, as in Section 4. By Lemmas 6 and 8, $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ are search trees on their corresponding paths and form connected subgraphs of $\mathcal{T}$ (note that this latter property does not generally hold for arbitrary search trees on trees).

Now we describe the representation of $\mathcal{T}$. Let s_i be the vertex in P_i with the minimum distance from the root of T, i.e., $P_i = P_T(s_i, l_i)$. We first store the tree structures of $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ using the representation of Lemma 1. By concatenating these representations into a single bit string in the order of the preorder numbers of s_i's in T, this requires $2n - 2\ell$ bits in total.

Unlike the paths defined in Section 4, we cannot decode the paths from T without additional information. However, using the properties of Lemmas 6, 7 and 8, we can decode them using only a small extra space as follows. We first explicitly store the position of the root r of $\mathcal{T}$ (which is also the root of T) using $\lceil \log n \rceil$ bits. Next, imagine an array $L[1, \ldots, \ell]$ of size ℓ, where $L[i] = j$ if and only if s_i is the j-th entry in the sequence of vertices $S = (s_1, \ldots, s_\ell)$, ordered by the preorder traversal of T (see Figure 4(d) for an example). Storing array L would require too much space. Instead let us show that by performing *Range Minimum Queries (RMQ)* on L,[3] we can determine all paths $P_1, \ldots, P_\ell$.

First, we reconstruct $P_{\min} = P(r, l_{\min})$ by computing $\min = \mathsf{RMQ}(1, \ell)$. Let S_L (resp., S_R) be the set of all left (resp., right) siblings of the vertices in $P_{\min}$. Since all paths are disjoint,

[3] The RMQ is defined as follows: for $1 \le i \le j \le \ell$, $\mathsf{RMQ}(i, j)$ returns the position of the minimum value in the subarray $L[i, \ldots, j]$

$S_L \subseteq \{s_1, \ldots, s_{\min -1}\}$ and $S_R \subseteq \{s_{\min +1}, \ldots, s_\ell\}$. Let s_{left} be the vertex in $\{s_1, \ldots, s_{\min -1}\}$ with the smallest preorder number in T, and define s_{right} analogously. Then since $s_{left} \in S_L$ and $s_{right} \in S_R$, we can reconstruct P_{left} by computing $left = \mathsf{RMQ}(1, \min -1)$ and P_{right} by computing $right = \mathsf{RMQ}(\min +1, \ell)$. We repeat this procedure recursively until all paths $P_1, \ldots, P_\ell$ are determined.

Thus, we do not need to store L explicitly, but rather we just need a data structure that answers RMQ on L. Bender and Farach-Colton [5] showed that RMQ queries can be answered via LCA queries on the Cartesian tree of L. Therefore, we obtain the RMQ data structure by storing the 2ℓ-bit BP representation of the shape of the Cartesian tree on L [43].

After reconstructing the paths, we reconstruct the search trees $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ with their labels by reading the bit string of their representations sequentially. Finally, we merge the search trees into a single tree $\mathcal{T}$. By Lemma 8 and the definition of the sibling ordering in $\mathcal{T}$, we can merge them according to the preorder numbers of the nodes in T.

From the above construction it follows that the total space required for the representation of a Steiner-closed STT is $(2n - 2\ell) + \lceil \log n \rceil + 2\ell = 2n + \lceil \log n \rceil$ bits.

This representation can be constructed in $O(n\ell)$ time as follows: (1) construct a mapping between the vertices in T and the nodes in $\mathcal{T}$ in $O(n)$ time by traversal; (2) perform the initial decomposition and modification procedure on T in $O(n\ell)$ time, since for each node visited during the decomposition we can determine its vertex set in $O(\ell)$ time by checking each set in $O(1)$ time using LCA on T; (3) construct L (which can be built during the modification procedure) and $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ in $O(n)$ time; and (4) construct the representations of the structures in (3) in $O(n)$ time [18].

We summarize the result in the following theorem:

▶ **Theorem 10.** *Given a labeled and ordered tree T with n vertices and ℓ leaves, any Steiner-closed search tree $\mathcal{T}$ on T can be encoded using $2n + \lceil \log n \rceil$ bits. The representation can be constructed in $O(n\ell)$ time.*

▶ **Example 11.** Consider how to decode the label of the node u in Figure 4(b) by reconstructing $\mathcal{T}$ from the representation. First, we decode the path $P_{\min} = P_1$. Here, $S_L = \emptyset$, $S_R = \{g, h\}$, and $right = \mathsf{RMQ}(2, 5) = 3$, which implies $P_3 = P(g, f)$. By repeating this procedure, we construct all paths $P_1, \ldots, P_5$. Next, we decode the tree structures with the labels of $\mathcal{T}_1, \ldots, \mathcal{T}_5$ and merge them into a single tree $\mathcal{T}$. For example, to merge $\mathcal{T}_1$ and $\mathcal{T}_3$, we first check the preorder numbers of b and f in T, and since $\mathsf{preorder}_T(b) < \mathsf{preorder}_T(f)$, we attach the root of $\mathcal{T}_3$ (the node f) as the child of d and as the right sibling of b, following the definition of the sibling ordering in $\mathcal{T}$. After fully reconstructing $\mathcal{T}$, we can determine that the label of u is g.

6 Conclusions

In this paper, we present space-efficient representations of general and Steiner-closed STTs that can be constructed in polynomial time and achieve optimal space up to lower-order additive terms, as well as space-efficient data structures supporting fast traversals.

Natural directions include extending our techniques to k-cut trees [6] and designing space-efficient data structures for SplayTT [6] that support rotations using tiny pointers [4]. Another interesting direction is to obtain instance-optimal succinct representations for general STTs whose space depends on the topology of T, since the $n \log \ell$ bound need not be tight for every T.

References

1. Georgii M. Adel'son-Vel'skii and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
2. Bengt Aspvall and Pinar Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT*, 34(4):484–509, dec 1994. doi:10.1007/BF01934264.
3. Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal search in trees. *SIAM Journal on Computing*, 28(6):2090–2102, 1999. doi:10.1137/S009753979731858X.
4. Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In *SODA*, pages 477–508. SIAM, 2023.
5. Michael A Bender and Martin Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer, 2000.
6. Benjamin Aram Berendsohn and László Kozma. Splay trees on trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1875–1900, 2020. doi:10.1137/1.9781611977073.75.
7. Lucas Boczkowski, Uriel Feige, Amos Korman, and Yoav Rodeh. Navigating in trees with permanently noisy advice. *ACM Trans. Algorithms*, 17(2), jun 2021. doi:10.1145/3448305.
8. Hans L. Bodlaender, Jitender S. Deogun, Klaus Jansen, Ton Kloks, Dieter Kratsch, Haiko Müller, and Zsolt Tuza. Rankings of graphs. *SIAM Journal on Discrete Mathematics*, 11(1):168–181, 1998. doi:10.1137/S0895480195282550.
9. H.L. Bodlaender, J.R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995. doi:10.1006/jagm.1995.1009.
10. Prosenjit Bose, Jean Cardinal, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Competitive online search trees on trees. *ACM Trans. Algorithms*, 19(3):25:1–25:19, 2023. doi:10.1145/3595180.
11. Renato Carmo, Jair Donadelli, Yoshiharu Kohayakawa, and Eduardo Laber. Searching in random partially ordered sets. *Theoretical Computer Science*, 321(1):41–57, 2004. Latin American Theoretical Informatics. doi:10.1016/j.tcs.2003.06.001.
12. Michael Carr and Satyan L. Devadoss. Coxeter complexes and graph-associahedra. *Topology and its Applications*, 153(12):2155–2168, 2006. doi:10.1016/j.topol.2005.08.010.
13. Ferdinando Cicalese, Tobias Jacobs, Eduardo Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, 412(50):6879–6896, 2011. doi:10.1016/j.tcs.2011.08.042.
14. Jitender S. Deogun, Ton Kloks, Dieter Kratsch, and Haiko Müller. On vertex ranking for permutations and other graphs. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '94, page 747–758, Berlin, Heidelberg, 1994. Springer-Verlag.
15. Dariusz Dereniowski. Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, 156(13):2493–2500, 2008. Fifth International Conference on Graphs and Optimization. doi:10.1016/j.dam.2008.03.007.
16. Yevgeniy Dodis, Mihai Patrascu, and Mikkel Thorup. Changing base without losing space. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 593–602, 2010.
17. Guy Even and Shakhar Smorodinsky. Hitting sets online and unique-max coloring. *Discrete Applied Mathematics*, 178:71–82, 2014. doi:10.1016/j.dam.2014.06.019.
18. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
19. Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.
20. Ananth V. Iyer, H.Donald Ratliff, and G. Vijayan. Optimal node ranking of trees. *Information Processing Letters*, 28(5):225–229, 1988. doi:10.1016/0020-0190(88)90194-9.
21. Guy Jacobson. Space-efficient static trees and graphs. In *FOCS 1989*, pages 549–554, 1989.

22 Seungbum Jo and Nodari Sitchinava. Succinct representation of search trees on trees, 2026. URL: https://arxiv.org/abs/2609.21236, arXiv:2609.21236.

23 Meir Katchalski, William McCuaig, and Suzanne Seager. Ordered colourings. *Discrete Mathematics*, 142(1):141–154, 1995. doi:10.1016/0012-365X(93)E0216-Q.

24 Donald E. Knuth. Optimum binary search trees. *Acta Inf.*, 1(1):14–25, 1971. doi:10.1007/BF00264289.

25 Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, Redwood City, CA, USA, 1998.

26 Eduardo Laber and Marco Molinaro. An approximation algorithm for binary searching in trees. *Algorithmica*, 59(4):601–620, apr 2011.

27 Tak Wah Lam and Fung Ling Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001. doi:10.1007/S004530010076.

28 Nathan Linial and Michael Saks. Searching ordered structures. *Journal of Algorithms*, 6(1):86–103, 1985. doi:10.1016/0196-6774(85)90020-3.

29 Joseph W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990. doi:10.1137/0611010.

30 Kazuhisa Makino, Yushi Uno, and Toshihide Ibaraki. On minimum edge ranking spanning trees. *Journal of Algorithms*, 38(2):411–437, 2001. doi:10.1006/jagm.2000.1143.

31 Thibault Manneville and Vincent Pilaud. Graph properties of graph associahedra. *Séminaire Lotharingien de Combinatoire*, 73, 2014.

32 Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, page 1096–1105, USA, 2008. Society for Industrial and Applied Mathematics.

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

34 Shanmugavelayutham Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666, 2002.

35 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

36 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2–es, 2007.

37 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

38 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity: Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-27875-4.

39 Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 379–388, 2006. doi:10.1109/FOCS.2006.32.

40 Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990. doi:10.1137/0611030.

41 A. A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, nov 1989. doi:10.1016/0020-0190(89)90161-0.

42 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

43 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.