The proposed measures are called the \emph{tree entropy} ($H_k(T)$) and the \emph{tree label entropy} ($H_k(L(T))$), respectively, i.e. $H_k(T) \leq H(T)$ and $H_k(L(T)) \leq H_k(L)$. Besides being theoretically superior, the new measures are significantly smaller in practice.

We also propose a new succinct representation of labeled trees which represents a tree $T$ using one of the following bounds: $|T| (H(T) + H_k(L(T)))$ or $|T| (H_k(T|L) + H_k(L))$. The representation is based on a new, simple method of partitioning the tree, which preserves both tree shape and node degrees. The previous state-of-the-art method of compressing the tree achieved $|T| (H(T) + H_k(L))$ bits, by combining the results of Ferragina at al. 2005 and Jansson et al. 2006; so proposed representation is not worse and often superior. Moreover, our representation supports standard tree navigation in constant time as well as more complex queries. Such a structure achieving this space bounds was not known before: aforementioned solution only worked for compression alone, our structure is the first which achieves $H_k(T)$ for $k > 0$ and supports such queries. Lastly, our data structure is fairly simple, both conceptually and in terms of the implementation, moreover it uses known tools, which is a counter-argument to the claim that methods based on tree-partitioning are impractical.
like aforementioned PPM or BWT with compression boosting, there are also compressed structures which consume such space, and moreover allow to perform operations on texts, like access [16], insertion/deletion [26] rank/select [4, 25] or even pattern matching [24].

**Tree Label entropy.** In case of labeled trees, 0-th order entropy of tree labels has a natural definition: it is a zeroth order entropy of string made by concatenation of labels of vertices. However, for k-th order entropy the situation is more involved, as we have to somehow define the context.

Ferragina et al. [13] proposed a definition of k-th order entropy of labeled trees. The context is defined as the k labels from the node to the root. Such definition of entropy takes into the account only the labels, and so we call it the k-th order entropy of labels. It is claimed to be good measure both in practice [14] (as similarly labeled nodes descend from similarly labeled contexts [13]) and in theory, as it is related to notion of tree sources [10].

Ferragina et al. [13] not only introduced the concept of k-th order entropy but also proposed a novel transform, XBWT, which works by generating a single string from tree labels by employing BWT-like transform on trees. It can be used to compress the tree using |T|((2 + H_k(T) + µ) + o(|T|)) bits [13], for a reasonable k; where the 2|T| bits are needed for encoding the tree shape and µ is a small constant, an artifact of used method of compression boosting (well-known in text compression [15]). The major drawback of this approach is that it works only for the case of compression alone, i.e. this approach does not generalize to a structure on which we can perform queries, as to this end rank/select structure over alphabet of size σ is required, which seems hard to combine with compression boosting. Dropping the compression boosting gives the structure which achieves H_0(T) and even then it supports only navigational queries (in particular, it does not support level_ancestor or depth), though it also support some basic label-related operations, as well as path pattern matching (i.e. pattern matching on strings made by concatenating labels of vertices on paths from root to leaves). The other XBWT-related result is by Ferragina et al. [14], who proposed a solution which encodes the labels of the tree using k-th order entropy of string produced by XBWT and support some navigational queries for k = o(log_σ |T|); the k-th order entropies of the tree and of string produced by XBWT seem loosely related and no formal relation is known between the two, but they argued that intuitively they are connected, because they both similarly cluster node labels. Still, this structure supports a very limited set of operations, has provably non-constant query time (due to the lower bounds on rank/select indices [4, 25]) and large additional space consumption.

**Tree entropy.** The (k-th order) entropy of labels ignores the shape of the tree and the information carried by it, and we still need to represent the tree structure somehow. A counting argument shows that 2|T| bits are needed for a random unlabeled tree [32], and indeed there are representations [32, 31, 6] and succinct structures achieving such bounds [42, 40, 12, 39, 43, 6, 31]. Yet, the 2|T| bits bound seems sub-optimal, as similarly as in the case of strings, real data is rarely a random tree drawn from the set of all trees, for example XML files are shallow and some tree shapes repeat, like in the Figure 1. For this reason Jansson et al. [33] introduced the notion of tree entropy with the idea that it takes into the account the probability of a node having a particular degree, i.e. it measures the number of trees under some degree distribution. Formally, it is defined as: |T|H(T) = −∑_i∈[|T|] d_i log d_i/|T|, where d_i is the number of vertices of degree i in T. Up to O(log |T|) additive summand, tree entropy is an information-theoretic lower bound on the number of bits needed to represent unlabeled tree that has some fixed degree distribution [33]. Similarly, as in the case of string
entropy, tree entropy refines the simpler estimations, in the sense that $|T|H(T) \leq 2|T|$ [33]. Jansson et al. [33] also showed a structure for unlabeled trees which supported most of the tree operations and use at most $|T|H(T)$ bits. This approach can be combined with aforementioned XBWT with compression boosting to compress the tree to roughly $|T|H(T) + |T|H_k(T)$ bits, but again it works only for the case of compression alone and needs a small additional linear factor, $\mu|T|$, same as when using only XBWT with compression boosting.

**Our contribution.** Label entropy and tree entropy treat labels and tree structure separately, and so did most of the previous approaches to labeled tree data structures [13, 33, 14, 28]. Yet, the two are most likely correlated: one can think of XML document representing the collection of different entities such as books, magazines etc., see Figure 1. Knowing that the label of some node is equal to “book”, and that each book has an author, a year and a title, determines degree of the vertex to be three (cf. tree grammar compression model, where we explicitly assume that the label uniquely defines the arity of node [9]). On the other hand, knowing the degree of the vertex can be beneficial for information on labels.

Motivated by this, we start by defining two new measures of entropy of trees, which take into the account both tree structure and tree labels: $H_k(T|L)$ and $H_k(L|T)$. Those measures lower bound previous measures, i.e tree entropy [33] and $H_k$ [13], respectively. We also devise the encoding achieving newly defined values. To this end, we propose a new way of partitioning the tree. In contrast to previous approaches, (i.e. succinct representations [22, 12] and tree grammar compression [19]), this partition preserves both the shape of the tree and the degrees of the nodes. We show that by applying an entropy coder to tree partition we can bound the size of the tree encoding by both $|T|H_k(L) + |T|H_k(L) + O(|T|k \log \sigma / \log_\sigma |T| + |T| \log \log_\sigma |T| / \log_\sigma |T|)$ and $|T|H_k(L|T) + |T|H(T) + O(|T|(k + 1) \log \sigma / \log_\sigma |T| + |T| \log \log_\sigma |T| / \log_\sigma |T|)$ bits. Note that the additional summands are $o(|T| \log \sigma)$ if $k = o(\log_\sigma |T|)$. This is the first method not based on XBWT achieving bounds related to $H_k$, moreover it does not need additional $\mu|T|$ bits.

Using standard techniques we can augment our tree encoding, at the cost of increasing the constants hidden in the $O$ notation (the overall complexity is the same and only constants in the “lower order terms” are slightly larger), so that most of the queries are supported in constant time, thus getting the first structure which achieves $H_k$ for trees and supports queries on compressed representation. Previous state-of-the-art methods based on tree grammars [18] and high-order compressed XBWT [13], do not have this property.

Then we show that we can further reduce the redundancy to $O(|T| \log \log |T| / \log_\sigma |T|)$ bits, at the cost of increasing query time to $O(\log |T| / \log \log |T|)$, assuming $k = \alpha \log_\sigma |S|$ where $\alpha < 1$. To this end we combine ideas from compression boosting [15] and compressed text representations [26].
Note that the high-order compressed structures for strings supporting access queries achieve analogous bit-size and query time: i.e. they either use $|S|H_k(S) + \mathcal{O}(|S|k \log \sigma \log \log |S|/\log |S|)$ bits and support queries in $\mathcal{O}(1)$ time or they use $|S|H_k(S) + \mathcal{O}(|S|k \log \log |S|/\log |S|)$ bits and have the query time of $\mathcal{O}(\log |S|/\log \log |S|)$ [16, 23, 26], and it is an open problem whether this trade-off is optimal. It is straightforward that improving our bounds even when the structure supports only preorder_rank/preorder_select operations yields the improvement for the high-order compressed structures for strings.

To compare, the original method of XBWT consumed $|T|H_0(L)$ bits for structure supporting navigation queries, also previously mentioned result by Ferragina et al. [14], which considered $k$-th order entropy of XBWT string instead $k$-th order label entropy, worked only for $k = o(\log \sigma |S|)$ and had a redundancy $o(|S| \log \sigma)$.

Moreover, we can use known tools to support label-related operations (like childrank($v, a$) or childselect($v, i, a$)). As these two are generalization of rank/select on large alphabets, the obtainable redundancy is larger than aforementioned bounds achieved by our structures, we also have to allow for a non-constant query time (see lower bounds in [4, 25]). In this case we get the same redundancy and query time for label operations as one of the previously mentioned results by Ferragina et al. [14], (it used space proportional to $H_k$ of XBWT string and supported only navigational queries), but in contrast to this result, our structure still supports all non-label related operations in constant time, thus we get the structure which outperforms previously known ones.

Our methods can also be applied to unlabeled trees to achieve tree entropy [33], in which case we get the same (best known) additional space as in [33] and our techniques in case of standard operations are less complex than other methods achieving tree entropy [33, 11].

Lastly, our structure allows to retrieve the tree in optimal, $\mathcal{O}(|T|/\log |T|)$ time (assuming machine words of size $\Theta(\log |T|)$), using $\mathcal{O}(1)$ memory, in contrast to XBWT-based methods.

The comparison of our results and previous results based on XBWT is in Table 2. Table 1 gives an empirical comparison of our measures with previous ones.

| Table 1 | Data is taken from XMLCompBench (http://xmlcompbench.sourceforge.net). The last four columns correspond to the bits per node used by compressed data structures (ignoring the smaller order terms); note that the proposed structure is the only one to (simultaneously) attain the last two. Note that sometimes the measures do not decrease when $k$ grows, this is because some files have shallow structure. |

| File          | $k$ | $H(T)$ | $H_k(L)$ | $H_k(L)|T|$ | $2H_k(L)$ | $H_k(T)$ | $H_k(T)L$ | $H_k(L)$ | $H_k(L)|T|$ |
|---------------|-----|--------|----------|-------------|------------|----------|-----------|----------|------------|
| EnWikiNew     | 0   | 0.233  | 0.537    | 0.031       | 0.335      | 2.537    | 0.77      | 0.568    | 0.568      |
|               | 1   | 0.233  | 0.311    | 0.031       | 0.216      | 2.311    | 0.544     | 0.342    | 0.45       |
|               | 2   | 0.233  | 0.311    | 0.031       | 0.216      | 2.311    | 0.544     | 0.342    | 0.45       |
| Nasa          | 0   | 0.33   | 0.849    | 0.06        | 0.579      | 2.849    | 1.178     | 0.909    | 0.909      |
|               | 1   | 0.33   | 0.232    | 0.056       | 0.174      | 2.232    | 0.562     | 0.288    | 0.503      |
|               | 2   | 0.33   | 0.23     | 0.056       | 0.172      | 2.23     | 0.559     | 0.286    | 0.501      |
| Treebank      | 0   | 1.812  | 4.616    | 0.692       | 3.495      | 6.616    | 6.428     | 5.308    | 5.308      |
|               | 1   | 1.812  | 3.234    | 0.656       | 2.259      | 5.234    | 5.046     | 3.89     | 4.072      |
|               | 2   | 1.812  | 3.073    | 0.64        | 2.109      | 5.073    | 4.886     | 3.713    | 3.922      |
|               | 4   | 1.812  | 2.957    | 0.626       | 1.997      | 4.957    | 4.77      | 3.584    | 3.809      |
2 Definitions

We denote the input alphabet by $\Sigma$, and size of the input alphabet as $\sigma = |\Sigma|$. For a tree $T$ we denote by $|T|$ the number of its nodes, the same applies to forests. We consider rooted (i.e. there is a designated root vertex), ordered (i.e. children of a given vertex have a left-to-right order imposed on them) $\Sigma$-labeled (i.e. each node has a label from $\Sigma$) trees; label does not determine node degree nor vice versa. We assume that bit sequences of length $\log |T|$ fit into $O(1)$ machine words and we can perform operations on them in $O(1)$ time.

Tree Label entropy. The tree label entropy [13] defines the context of a node as the concatenations of $k$ labels from the node to the root. Similarly, as in the case of first $k$ letters in strings, this is undefined for nodes whose path to the root is of length less than $k$, which can be large, even when $k$ is small. There are two ways of dealing with this problem: in the first we allow the node to have the whole path to the root as its context (when this path is shorter than $k$); in the second we pad the too short context with some fixed letters. Our algorithms can be applied to both approaches, resulting in the same (asymptotic) redundancy; for the sake of the argument we choose the first one, as the latter can be easily reduced to the former.

The tree label entropy is formally defined as $|L|L_k(L) = -\sum_{v \in T} \log P(l_v|K_v)$, where $l_v$ is label of vertex $v$, $K_v$ is the word made by last $k$ labels of nodes on the path from root of $T$ to $v$ (or less if the path from the root to $v$ is shorter than $k$) and, as in the case of strings, $P(l_v|K_v)$ is the empirical probability of label $l_v$ conditioned that it occurs in context $K_v$.

Mixed entropy. We define the mixed entropies as follows:

- $|T|H_k(T|L) = -\sum_{v \in T} \log P(d_v|K_v, l_v)$, where $P(d_v|K_v, l_v)$ is the empirical probability of node $v$ having degree $d_v$ conditioned that it occurs in the context $K_v$ and the node degree is $d_v$, that is, $P(d_v|K_v, l_v) = \frac{t_{K,v,d_v}}{t_{K,d_v}}$, where $t_{K,v,d_v}$ is a number of nodes in $T$ with context $K$, having degree $d_v$ and a label $l_v$, and $t_{K,d_v}$ is number of nodes in $T$ preceded by the context $K$, and having degree $d_v$.

- $|T|H_k(L|T) = -\sum_{v \in T} \log P(l_v|K_v, d_v)$, where $P(l_v|K_v, d_v)$, is the empirical probability of node $v$ having degree $d_v$ and a label $l_v$, and $t_{K,d_v}$ is number of nodes in $T$ preceded by the context $K$, and having degree $d_v$.

We show that we can represent a tree using either $|T|H_k(T|L) + |T|H_k(L)$ or $|T|H_k(L|T) + |T|H(T)$ bits (plus some small order terms), see Section 5.

The new measures lower bound the old ones. This follows directly from log sum inequality: intuitively increasing number of contexts can only reduce the entropy.

Lemma 1. The following inequalities hold:

$$|T|H_k(T|L) \leq |T|H(T) \quad \text{and} \quad |T|H_k(L|T) \leq |T|H_k(L) .$$
Table 2 Comparison of the results, multiple values in the same operation set means that we can choose one of the possibilities, \( \rho_0 = O(k \log \sigma + \log \log \sigma |T|)/\log \sigma |T| \); \( \rho_b = O(\log \log |T|/\log \sigma |T|) \); \( g_0 = \text{polylog}(\sigma) \cdot \sigma^k \). Navigation = \{parent(v), firstchild(v), nextsibling(v), childrank(u), child(u,i)\}; Extended = \{lca(u,v) and depth(u)\}; Label = \{childrank(v,a) and childselect(v,i,a)\}, i.e., versions of childrank(u) and child(u,i) which take label into account; Level-Ancestor = \{level_ancestor(v,i)\}. XBWT(L) refers to the XBWT string [14]. \( T_0(\text{rank/select})/S_b(\text{rank/select}) \) and \( T_b(\text{rank/select})/S_b(\text{rank/select}) \) refer to the time and space required for structures supporting rank/select over strings over \( \sigma \)-sized alphabets, the first pair refers to query time/additional space required when the string is compressed with at most \( |S| H_0(S) \) bits, respectively the second refers to the case when string is compressed using \( |S| H_b(S) \) bits. Those values depends on the alphabet size, in general, for strings compressed with \( |S| H_b(S) \) bits either \( S_b(\text{rank/select}) \) is larger than \( |T|\rho_b \) and \( |T|\rho_b \) or query time is not constant [4].

<table>
<thead>
<tr>
<th>Operation set</th>
<th>Our structure</th>
<th>XBWT-based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>space</td>
</tr>
<tr>
<td>Compression</td>
<td>(</td>
<td>T</td>
</tr>
<tr>
<td>Navigation</td>
<td>(O(1))</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>(O\left(\frac{\log(T)}{\log \sigma}\right))</td>
<td>(</td>
</tr>
</tbody>
</table>

Extended as Navigation not supported

Label Depends on the alphabet size, see Section 7 as Navigation

3 Tree clustering

We present a new clustering method which preserves both node labels and vertex degrees.

Clustering. The idea of grouping nodes was used before in the context of compressed tree indices [22, 27], also some dictionary compression methods like tree-grammars or top-trees and other carry some similarities [8, 18, 29, 21]. Yet, from our perspective, their main disadvantage is that the node degrees in the internal representation were very loosely connected to the node degrees in the input tree, thus the tree entropy and mixed entropies are hardly usable in upper bounds on space usage. We propose a new clustering method, which preserves the node degrees and the tree structure much better: most vertices inside clusters have the same degree as in \( T \), and the rest have their degree zeroed. As we show, this property implies a bound on the used space both in terms of the label/tree entropy and mixed entropy of \( T \), when an entropy coder is used to compress the multiset of clusters.

The idea of our clustering technique is that we group nodes into clusters of \( \Theta(\log \sigma |T|) \) nodes, and collapse each cluster into a single node, thus obtaining a tree \( T' \) of \( O(|T|/\log \sigma |T|) \) nodes. We label its nodes so that the new label uniquely determines the cluster that it represents and separately store the description of the clusters.

The clustering uses a parameter \( m \), \( 2m - 1 \) is the maximum size of the cluster. Each node of the tree is in exactly one cluster and there are two types of nodes in a cluster: port and regular nodes. A port node is a leaf in a cluster and a non-leaf in \( T \), for a regular node
all its children in $T$ are also in the same cluster (in the same order as in $T$); in particular, its degree in the cluster is the same as in $T$. Observe that this implies that each node with degree larger than $2m$ will be a port node.

The desired properties of the clustering are:

1. (C1) there at least $\frac{|T|}{2m} - 1$ and at most $\frac{|T|}{m} + 1$ clusters;
2. (C2) each cluster is of size at most $2m - 1$;
3. (C3) each cluster is a forest of subtrees (i.e. connected subgraphs) of $T$, roots of trees in this forest are consecutive siblings in $T$;
4. (C4) each node in a cluster $C$ is either a port node or a regular node; each port node is a leaf in $C$ and non-leaf in $T$, each regular node has the same degree in $C$ as in $T$. In particular, if node $u$ belongs to some cluster $C$, then either all or none of its children are in $C$.

A clustering satisfying (C1–C4) can be found using a natural bottom-up greedy algorithm. Note that the previously known tree factorizations [22, 12] do not satisfy the above properties, especially the (C3) and (C4), which are crucial for our structure.

Lemma 2. Let $T$ be a labeled tree. For any $m \leq |T|$ we can construct in linear time a partition of nodes of $T$ into clusters satisfying conditions (C1–C4).

Figure 2 Clustering of tree for parameter $m = 3$ and tree created by replacing clusters with new nodes. Marked nodes are port nodes.

Building the cluster tree. We build the cluster tree out of the clustering satisfying (C1–C4): we replace each cluster with a new node and put edges between new nodes if there was an edge between some nodes in the corresponding clusters. To retrieve the original tree $T$ from the cluster tree and its labels we need to know the degree of each port node in the original tree (note that this depends not only on the cluster, but also on the particular cluster node, e.g. two clusters may have the same structure but can have different port node degrees in $T$).

Thus we store for a cluster node with $k$ ports a degree sequence $d_1, d_2, \ldots, d_k$, where $d_i$ is the number of clusters containing children of $i$-th port node. For an illustration, see cluster replaced by cluster node labeled $B$ in Figure 2, its degree sequence is $3, 2, 2$. In section 5 we show that degree sequence can be stored efficiently and along with cluster tree it is sufficient to retrieve and navigate $T$.  

STACS 2020
Using Statistical Encoding to Achieve Tree Succinctness Never Seen Before

Definition 3 (Cluster structure). For a labeled tree $T$ and parameter $m$ a cluster structure $C(T)$ consists of:

- Ordered, rooted, labeled tree $T'$ (called cluster tree) with $\Theta(|T|/m)$ nodes, where each node represents a cluster, different labels correspond to different clusters and the induced clustering of $T$ satisfies (C1–C4).

- For each node $v \in T'$, a degree sequence $d_{v,1}, \ldots, d_{v,j}$, where $d_{v,i}$ means that $i$-th node in left-to-right order on leaves of cluster represented by $v$ connects to $d_{v,i}$ clusters of $T'$.

- Look-up tables, which for a label of $T'$, allow to retrieve the corresponding cluster $C$.

Lemma 4. For a labeled tree $T$ and a parameter $m$ we can construct in time $O(|T|)$ cluster structure $C(T)$.

4 Entropy estimation

We show that entropy of labels of tree of $C(T)$ is upper bounded by the mixed entropy of the input tree, up to some small additive factor. Due to the technical details, in the last case the redundancy depends on $k+1$ and not on $k$.

Theorem 5. Let $T$ be a labeled tree and let $T'$ be a tree of cluster structure $C(T)$ from Lemma 4 for parameter $m$. Let $P$ be a string obtained by concatenation of labels of $T'$. Then all the following inequalities simultaneously hold:

$$|P|H_0(P) \leq |T|H(T) + |T|H_k(L) + O\left(\frac{|T|k\log \sigma}{m} + \frac{|T| \log m}{m}\right),$$

(1)

$$|P|H_0(P) \leq |T|H_k(T|L) + |T|H_k(L) + O\left(\frac{|T|k\log \sigma}{m} + \frac{|T| \log m}{m}\right),$$

(2)

$$|P|H_0(P) \leq |T|H(T) + |T|H_k(L|T) + O\left(\frac{|T|(k+1)\log \sigma}{m} + \frac{|T| \log m}{m}\right).$$

(3)

Moreover, if $m = \Theta(\log_\sigma |T|)$, the additional terms are bounded by:

$O\left(\frac{|T|k\log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$ and $O\left(\frac{|T|(k+1)\log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$, respectively. For $k = o(\log_\sigma |T|)$ both those values are $o(|T| \log \sigma)$.

We prove Theorem 5 in two steps. First, we devise a special representation of nodes of $T'$. The entropy of this representation is not larger than the entropy of $P$, thus we upper bound the entropy of $P$. To this end we use the following corollary from Gibbs’ inequality, see [1] for a simple proof.

Lemma 6 ([1]). Let $w \in \Gamma^*$ be a string and $q : \Gamma \rightarrow \mathbb{R}^+$ be a function such that $\sum_{s \in \Gamma} q(s) \leq 1$. Then $|w|H_0(w) \leq -\sum_{s \in \Gamma} t_s \log \log q(s)$, where $t_s$ is the number of occurrences of $s$ in $w$.

Lemma 6 should be understood as follows: we can assign each letter in a string a “probability” and calculate the “entropy” for a string using those “probabilities.” The obtained value is not smaller than the true empirical entropy. Thus, to upper-bound the entropy, it is enough to devise an appropriate function $q$.

Cluster representation. The desired property of the representation is that for each node $v$ in the cluster we can uniquely determine its context (i.e. labels of $k$ nodes on the path from $v$ to the root) in original tree $T$. 

Definition 7 (Cluster description). Given a cluster $C$ occurring in context $K$ and consisting of subtrees $T_1, \ldots, T_l$, the cluster description, denoted by $R_{K,C}$, is a triplet $(K,N_C,V_C)$, where

- $K$ is a context (of size at most $k$) preceding roots of the trees in cluster, i.e. for each root $r$ of tree in a cluster $K = K_r$ holds, where $K_v$ is the context of vertex $v$ in $T$; roots of trees in a cluster have the same context in $T$, as they have the same parent.
- $N_C$ is the total number of nodes in this cluster;
- $V_C$ is a list of descriptions of nodes of $C$, according to preorder ordering. If a node $v$ is a port then its description is $(1,l_v)$, if it is a regular node, then it is $(0,l_v,d_v)$, where $d_v$ is the degree of the node $v$ (in the cluster) and $l_v$ is its label.

Note that we do not store the degrees of port nodes, as they are always 0.

Example 8. The description for $k = 1$ and central node (the one which is labeled $B$ in cluster tree) from Figure 2 is: $(K = a, N = 4, V = \{(1,b),(0,a,2),(1,c),(1,a)\})$.

Lemma 9. Cluster description of a cluster $C$ uniquely defines a cluster $C$ and context $K_v$ in $T$ for each vertex $v$ in $C$.

To prove Theorem 5, instead of estimating the entropy of $P$ we will estimate the entropy of string where letters are descriptions of each cluster. To this end we employ Lemma 6: we assign each element of the description a separate value of $q$, which depends on both the element and previous elements of a description, then we multiply all such values. Note that storing, which nodes are port nodes inside a cluster description, uses additional $O(|T| \log m/m)$ bits, if done naively (e.g. in a separate structure) it would use the same memory.

Proof of Theorem 5. Let $P$ be a sequence of labels. Let $R \in \Gamma_R$ be a sequence of description of clusters of $T'$ in preorder ordering, so that description $R[i]$ of cluster $C$ corresponds to label $P[i]$. Then $H_0(P) \leq H_0(R)$, in particular $|P| H_0(P) \leq |R| H_0(R)$. To see this, note that each label of $P$ may correspond to a few descriptions from $R$, but not the other way around: if some nodes have different description, then they have different labels in $T'$.

Thus it is enough to upper-bound $|R| H_0(R)$. To this end we apply Lemma 6 for appropriately defined function $q$; different estimations require different variants of $q$. The function $q$ is defined on each $R[i]$. The assignment of values of $q$ can be thought as a procedure that starts with value 1 and then looks at each element of description and multiplies by a value which depends on this element in description (or some previous ones), i.e. as in adaptive arithmetic coding.

We begin with a proof for Case 2. Let $R_{K,C} = (K, N_C, V_C)$ be a description of the cluster $C$. We define $q(R_{K,C}) = q(K_C) \cdot q(N_C) \cdot q(V_C)$, where $q$ for each coordinate is defined as follows:

- $q(K_C) = 1/((k + 1) \cdot \sigma^{|K_C|})$
- $q(N_C) = 1/(2m)$
- $q(V_C) = \prod_{v \in V_C} q(v)$, where

$$q(v) = \begin{cases} \frac{1}{m} \cdot P(l_v|K_v) & \text{if } v = (1,l_v) \\ \frac{m-1}{m} \cdot P(l_v|K_v) \cdot P(d_v|K_v,l_v) & \text{if } v = (0,l_v,d_v) \end{cases}$$

Note that $K_v$ is the context of $v$ in original tree, i.e. $T$.

It is left to show that $q$ summed over all cluster descriptions is at most 1 as required by Lemma 6. To this end we will show that we can partition the interval $[0,1]$ into subintervals and assign each element of $\Gamma_R$ a subinterval of length $q(R_{K,C})$, such that for two symbols
of $\Gamma_R$ their intervals are pairwise disjoint. It is analogous to applying adaptive arithmetic coder. We start with interval $I = [0, 1]$ . We process description of cluster by coordinates, at each step we partition $I$ into disjoint subintervals and choose one as the new $I$:

- For $K_C$ we partition interval into $k + 1$ equal subintervals, each one corresponding to different context length, then we choose one corresponding to $|K_C|$. Then we partition the $I$ into $\sigma^{\left|K_C\right|}$ disjoint and equal subintervals, one for each different context of length $|K_C|$ and choose one corresponding to $K_C$. Clearly the length of the current $I$ at this point is $1/\left((|K_C| + 1) \cdot \sigma^{\left|K_C\right|}\right)$, also different contexts are assigned different intervals.

- For $N_C$ we partition the $I$ to $2m$ equal intervals, it is enough, as there are at most $2m$ different cluster sizes.

- Then we make a partition for each $v \in V_C$. We process vertex descriptions in $V_C$ according to the order they occur in list $V_C$, i.e. the order is the preorder ordering of corresponding vertices in $C$. We partition the interval into two, one of length $\left|I\right|/m$, second $\left|I\right|/(m-1)$. If $v = (1, l_v)$ we choose the first one, otherwise we choose the second one. Then we partition the $\left|I\right|$ into $\sigma$ different intervals (some may be of 0 length), one for each letter $a$ of original alphabet $\Sigma$; the subinterval for letter $a$ has length $\left|I\right| \cdot P(a|K_v)$. It is a proper partition, as all of the above values sum up to 1 by definition. We choose the one corresponding to the letter $l_v$. Lastly, if $i = (0, l_v, d_v)$ then we partition the interval into intervals corresponding to different degrees of nodes, again of lengths $\left|I\right| \cdot P(d_v|K_v, l_v)$.

By construction the interval assigned to $R_{K,C}$ has length $q(R_{K,C})$. Also, for two different clusters $C_1, C_2$, having different preceding contexts $K_{C_1}, K_{C_2}$, their intervals are disjoint. To see this consider the above procedure which assigns intervals and the first point where descriptions $R_{K_{C_1},C_1}, R_{K_{C_2},C_2}$ differ (there must be such point by Lemma 9). Observe that up to this point both clusters were assigned the same intervals. What is more already processed elements of $R_{K,C}$ uniquely define context (or label) for current vertex, i.e. when we want to partition by $P(l_v|K_v)$, then all nodes on the path to $v$ were already processed (this is due to the preorder ordering of nodes in description). This guarantees that, if the descriptions for two clusters $C_1, C_2$ were equal up to this point, then the interval will be partitioned in the same way for $C_1$ and $C_2$. At this point $R_{C_1}, R_{C_2}$ will be assigned different, disjoint intervals.

Now we are ready to apply Lemma 6. By $C_v$ and $K_v$ we denote the cluster represented by $v$ and context of this cluster in $T$, respectively; additionally let $n = \left|T\right|$.

\[|P|H_0(P) \leq |R|H_0(R)\]

\[\leq - \sum_{v \in T'} \log q(R_{K_v, C_v})\]

\[= - \sum_{v \in T'} \log (q(K_v) \cdot q(N_{C_v}) \cdot q(V_{C_v}))\]

\[\leq - \sum_{v \in T'} \log \left(\frac{1}{(k + 1)\sigma^k} \cdot \frac{1}{m} \cdot q(V_{C_v})\right)\]

\[= \left|T'\right| (k \log \sigma + \log(k + 1) + \log m) - \sum_{v \in T'} \log (q(V_{C_v}))\]

\[\leq - \sum_{v \in T'} \log q(V_{C_v}) + O\left(\frac{nk \log \sigma}{m} + \frac{n \log m}{m}\right).\]

Now:

\[- \sum_{v \in T'} \log q(V_{C_v}) = - \sum_{u \in T} \log q(u)\]
$$\begin{align*}
&= - \sum_{u \in T, u \text{ regular}} \log \left( \frac{1}{m} \cdot P(l_u | K_v) \right) - \sum_{u \in T, u \text{ port}} \log \left( \frac{m-1}{m} \cdot P(l_u | K_v) \cdot P(d_u | K_v, l_u) \right) \\
&\leq nH_k(L) + nH_k(T|\mathcal{L}) + \mathcal{O}(|T| \log m) + n \log \frac{m}{m-1} \\
&\leq nH_k(L) + nH_k(T|\mathcal{L}) + \mathcal{O} \left( \frac{n \log m}{m} \right) + \frac{n}{m-1} \log \left( 1 + \frac{1}{m-1} \right)^{m-1} \\
&\leq nH_k(L) + nH_k(T|\mathcal{L}) + \mathcal{O} \left( \frac{n \log m}{m} \right) + \mathcal{O} \left( \frac{n}{m} \right),
\end{align*}$$

which ends the proof for the Case 2. In the estimation we have used the fact that the total number of port nodes is at most $\mathcal{O}(n/m)$, since it cannot exceed the number of clusters.

The proof of Case 1 can be carried out in a similar manner, by replacing $P(d_v | K_v, l_v)$ with $\mathbb{P}(d_v | K_v)$; alternatively it follows from Lemma 1.

The Case 3 requires slight modification of assignment of $q$ to vertices, which reflects the different estimation:

$$q(v) = \begin{cases} 
\frac{1}{m} \cdot \frac{1}{\sigma} & \text{if } v = (1, l_v) \\
\frac{m-1}{m} \cdot P(d_v | K_v) \cdot P(l_v | K_v, d_v), & \text{if } v = (0, l_v, d_v) 
\end{cases}$$

Now, to show that values $q$ sum to at most one (i.e. they satisfy conditions of Lemma 6) the assignment of intervals must be changed, so that it reflects the current $q$: we first partition the interval by $\mathbb{P}(d_v | K_v)$ and then by $\mathbb{P}(l_v | K_v, d_v)$. The invariant is that when partitioning the interval for $\mathbb{P}(d_v | K_v)$ previous elements of $R_{K,C}$ uniquely determine $K_v$ and when for $\mathbb{P}(l_v | K_v, d_v)$ they uniquely determine $K_v, d_v$. However, this may not be true for port nodes: if $v$ is a port node then we cannot extract the information on original degree of $v$ from the cluster description alone. Hence we cannot partition the appropriate interval by $\mathbb{P}(d_v | K_v)$ for port nodes. This is why we have $q(v) = \frac{1}{m} \cdot \frac{1}{\sigma}$, instead of $\frac{m-1}{m} \cdot P(d_v | K_v)$ for port nodes. This adds additional $\mathcal{O} \left( \frac{n \log \sigma}{m} \right)$ term (since there are at most $\mathcal{O}(n/m)$ port nodes), hence we have $\mathcal{O} \left( \frac{n(k+1) \log \sigma}{m} \right)$ in the third case.

\section{Application — succinct data structure for labeled trees}

We demonstrate how our tree clustering technique can be used for compressed representation of labeled trees. Given a tree $T$ we choose $m = \beta \log_\sigma |T|$, for some constant $\beta$ to be determined later. For appropriate $\beta$ the number of different clusters is $\mathcal{O}(|T|^{\frac{1-\alpha}{\alpha}})$ for some constant $\alpha > 0$, moreover for such $m$ the cluster tree $T'$ from $C(T)$ can be stored using any succinct representation (like balanced parentheses or DFUDS) in space $\mathcal{O}(|T|/\log_\sigma |T|) = o(|T|)$ (for small enough $\sigma$). At the same time clusters are small enough so that we can preprocess them and answer all relevant queries within the clusters in constant time, the needed space is also $o(|T|)$.

Let $T'$ denote the unlabeled tree of $C(T)$, i.e. the cluster tree (C1) stripped of node labels. Our structure consists of:

\begin{enumerate}[label=(T\arabic*)]
\item Unlabeled tree $T'$: $|T'| = \mathcal{O}(|T|/\log_\sigma |T|)$.
\item String $P$ obtained by concatenating labels of the cluster tree of $C(T)$ in preorder ordering.
\item Degree sequences for each node of $T'$.
\item Precomputed arrays for each operation, for each cluster (along with look-up table from $C(T)$ to decode cluster structure from labels).
\end{enumerate}

We encode each of (T1–T4) separately, using known tools.
Using Statistical Encoding to Achieve Tree Succinctness Never Seen Before

(T1): Encoding tree $T'$. There are many succinct representation for unlabeled trees which allow fast navigational queries [40, 23, 12]. They all use $2|T'| + o(|T'|)$ bits for tree of size $|T'|$, the exact function suppressed by the $o(|T'|)$ depends on the data structure. Since in our case the tree $T'$ is already of size $O(|T|/\log_\sigma |T|)$, we can use $O(|T'|)$ bits for the encoding of $T'$, so we do not care about exact function hidden in $o(|T'|)$. This is of practical importance, as the data structures with asymptotically smallest memory consumption, like [40], are very sophisticated, thus hard to implement and not always suitable for practical purposes. Thus we can choose theoretically inferior, but more practical data structure [2], we can even use a constant number of such data structures, as we are interested only in $O(|T'|)$ bound.

Choose one method, say [37], for the sake of argument. We use it to encode $T'$ on $O(|T'|)$ bits, this encoding supports the following queries in constant time: $\text{parent}(v)$ — parent of $v$; $\text{firstchild}(v)$ — leftmost child of $v$; $\text{nextsibling}(v)$ — right sibling of $v$; $\text{preorder-rank}(v)$ — preorder rank of $v$; $\text{preorder-select}(i)$ — returns a node whose preorder rank is $i$; $\text{lca}(u, v)$ — returns the lowest common ancestor of $u, v$; $\text{childrank}(v)$ — number of siblings preceding a node $v$; $\text{child}(v, j)$ — $j$-th child of $v$; $\text{preorder-rank}(v)$ — rank of node $v$ in preorder ordering of nodes; $\text{preorder-select}(i)$ — $i$-th node in preorder ordering of nodes; $\text{leaf_rank}(v)$ — number of leaves to the left of $u$ plus one; $\text{leaf_select}(i)$ — $i$-th leaf counting from the left; $\text{depth}(v)$ — distance from the root to $v$; $\text{level_ancestor}(v, i)$ — ancestor at distance $i$ from $v$.

(T2): Encoding preorder sequence of labels. By Theorem 5 it is enough to encode the sequence $P$ using roughly $|P|H_0(P)$ bits, in a way that allows for $O(1)$ time access to its elements. This problem was studied extensively, and many (also practical) solutions were developed [23, 16, 26, 24]. Most of these methods are not overly complex: they assign a prefix code to consecutive groups of elements, concatenate prefix codes and use some simple structure for storing information, where the code words begin/end.

However, we must take into the account that alphabet of $|P|$ can be large (though, as shown later, not larger than $|T|^{1-\alpha}$, $0 < \alpha < 1$). This renders some of previous results inapplicable, for example the simplest (and most practical) structure for alphabet of size $\sigma'$ need additional $O(|P| \log \log |P|/\log_\sigma |P|)$ which can be as large as $O(|P| \log \log |P|)$ [16]. As $|P| = |T|/\log_\sigma |T|$ this would be slightly above bound from Theorem 12: $O(|T| \log \log |T|/\log_\sigma |T|)$ vs $O(|T| \log \log_\sigma |T|/\log_\sigma |T|)$.

Still, there are structures achieving $|P|H_0(P) + o(|P|)$ bits for alphabets of size $|T|^{1-\alpha}$, for example the well-known one by Pătraşcu [41, Theorem 1].

(T3): Encoding the degree sequence. To navigate the tree, we need to know which children of a cluster belong to which port node. We do it by storing degree sequence for each cluster node of $T'$ and design a structure which, given a node $u'$ of $T'$ and index of a port node $u$ in the cluster represented by $u'$, returns the range of children of $u'$ which contain children of $u$ in $T$. We do it by storing rank/select structure for bitvector representing degrees of vertices of $T'$ in unary.

Lemma 10. We can encode all degree sequences of nodes of $T'$ using $O(|T'|)$ bits in total, such that given node $u$ of $T'$ and index of port node $v$ in cluster represented by $u$ the structure returns a pair of indices $i_1, i_2$, such that the children of $v$ (in $T$) are exactly the roots of trees in clusters in children $i_1, i_1 + 1, \ldots, i_2 - 1$ of $u$. Moreover we can answer reverse queries, that is, given an index $x$ of $x$-th child of $u \in T'$ find port node which connects to this child. Both operations take $O(1)$ time.
(T4): Precomputed tables. We want to answer all queries in each cluster in constant time, we start by showing that there are not many clusters of given size.

**Lemma 11.** There are at most \(2^{2m'} \sigma^m\) different clusters of size \(m'\).

It is sufficient to choose \(m\) in clustering as \(m = \frac{1}{2} \log_\sigma |T|\) (or 1 if this is smaller than 1) assuming that \(2 \leq \sigma \leq |T|^{1-\alpha}, \alpha > 0\), then the number of different clusters of size at most \(2m - 1\) is \(\sum_{i=1}^{2m-1} 2^i \sigma^i \leq O(|T|^{1-\alpha})\).

We precompute and store the answers for each query for each cluster. As every query takes constant number of arguments and each argument ranges over \(m\) values, this uses at most \(O(|T|^{1-\alpha}) \cdot O(\log_\sigma |T|) = o(|T|)\) bits, where \(c\) is a constant. Additionally we make tables for accessing \(i\)-th (in left-to-right order on leaves) port node of each cluster, as we will need this later to support more involved queries.

**Putting it all together.** The above structures can be combined into a succinct data structure for trees. Note that we used the fact that \(\sigma \leq |T|^{1-\alpha}\), we generalize for arbitrary \(\sigma\) in the full version.

**Theorem 12.** Let \(T\) be a labeled tree with labels from an alphabet of size \(\sigma \leq |T|^{1-\alpha}, \alpha > 0\). Then we can build in linear time a tree structure whose bit size is bounded by all of the below values:

\[
|T| H(T) + |T| H_k(L) + O\left(\frac{|T| k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)
\]

\[
|T| H_k(T) + |T| H_k(L) + O\left(\frac{|T| k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)
\]

\[
|T| H(T) + |T| H_k(L)/|T| + O\left(\frac{|T| (k+1) \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)
\]

It supports firstchild(u), parent(u), nextsibling(u), lca(u,v), childrank(u), child(u,i), depth(u), preorder_rank(u), preorder_select(i), leaf_rank(u) and leaf_select(i). operations in \(O(1)\) time; at the expense of additional \(O((|T|/\log \log |T|)^2)/\log_\sigma |T|\) bits it can support level_ancestor(v, i) query in \(O(1)\) time.

The main idea of Theorem 12 is that for each query if both arguments and the answer are in the same cluster then we can use precomputed tables, for other we can query the structure for \(T\) and reduce it to the former case using previously defined structures. A similar idea was used in other tree partition based structures like [22] (yet this solution used different tree partition method).

### 6 Even succincter structure

So far we have obtained the redundancy of \(O(|T| \log \log_\sigma |T|/\log_\sigma |T|) + O(|T| k \log \sigma / \log_\sigma |T|)\). As the recent lower bound for zeroth-order entropy coding of a string partition [20] (assuming certain partition properties) also applies to trees, we can conclude that our structure in worst case requires \(\Omega(|T|k \log \sigma / \log_\sigma |T|)\) additional bits. Yet, this lower bound only says that the above factor is necessary when zeroth-order entropy coder is used, not that this factor is required in general. Indeed, for strings, there are methods of compressing the text \(S\) (with fast random access) using \(|S| H_k(S) + o(|S|) + f(k, \sigma)\) bits [26, 38], also methods related to compression boosting achieve \(|S| H_k(S) + O(|S|) + f(k, \sigma)\) bits [15], where \(f(k, \sigma)\) is some function that depends only on \(k\) and \(\sigma\). Similarly, the method of compressing the tree
This data structure supports the following operations in \( O(|T| \log \log |T|) \) time:

- firstchild\( (u) \), parent\( (u) \), nextsibling\( (u) \), lca\( (u,v) \), child\( (v,i) \), childrank\( (u) \), depth\( (u) \), preorder_rank\( (u) \), preorder_select\( (i) \), leaf_rank\( (u) \) and leaf_select\( (i) \). It can support level_ancestor\( (v,i) \) query in \( O(|T|^{1/2} \log \log |T|) \) time with additional \( O(|T| (\log \log |T|)^2 / \log |T|) \) bits.

### 7 Label-related operations

Using additional memory we can support some label-related operations previously considered for succinct trees [44, 28]. Even though we do not support all of them, in all cases we support at least the same operations as XBWT, i.e. childrank\( (v,u) \) (which returns \( v \)'s rank among
children labeled with \( a \) and \( \text{childselect}(v,i,a) \) (which returns the \( i \)th child of \( v \) labeled with \( a \)). To this end we employ \( \text{rank/select} \) structures for large alphabets [3, 4]. Note, that we do not have constant time for every alphabet size and required additional space is larger than for other operations (i.e. \( o(|T| \log \sigma) \)), but this is unavoidable [4, 25]. Moreover, as the last point of Theorem 14 use structures which are not state-of-the-art, it is likely that better structures [4] are applicable, but they are more involved and it is not clear if they are compatible with our tree storage methods.

Theorem 14. We can augment structures from Theorem 12 and Theorem 13 so that:

- For \( \sigma = O(1) \) we can perform: \( \text{childrank}(v,a) \), \( \text{childselect}(v,i,a) \), \( \text{level_ancestor}(v,i,a) \), \( \text{depth}(v,a) \) for structure from Theorem 12 in \( O(1) \) time and for structure from Theorem 13 in \( O(\log |T|/\log \log |T|) \) time using asymptotically the same additional memory.

- For \( \sigma = O(\log^{1+\omega(1)}|T|) \) and \( \sigma = \omega(1) \) we can perform: \( \text{childrank}(v,a) \) and \( \text{childselect}(v,i,a) \) for structure from Theorem 12 in \( O(1) \) time and for structure from Theorem 13 in \( O(\log |T|/\log \log |T|) \) time using additional \( o(|T| \log \sigma) \) bits.

- For arbitrary \( \sigma \) we can perform: \( \text{childrank}(v,a) \) and \( \text{childselect}(v,i,a) \) for structure from Theorem 12 in \( O(\log \log^{1+\epsilon} \sigma) \) time and for structure from Theorem 13 in \( O((\log \log^{1+\epsilon} \sigma) \log |T|/\log \log |T|) \) time; using additional \( o(|T| \log \sigma) \) bits.

8 Open problems

There are a few open questions. First, can our analysis be applied to recently developed dictionary compression methods for trees like Top-Trees/Top-Dags [8, 29] or other dictionary based methods on trees? Related is the problem of finding good compression measures for repetitive trees, for instance for the case of text we have LZ77 and BWT-run, for which we can build efficient structures (like text indices) based on this representations [36, 17] and find relation with information-theoretic bounds (like \( k \)-th order entropy) or even show that they are close in information-theoretic sense to each other [35]. Even though we have tree representation like LZ77 for trees [21] or tree grammars [34] we do not know relation between them nor how they correspond to tree entropy measures (with the exception of recent paper on tree grammars [30], yet this approach works only in the restricted case of binary trees and the presented definition of tree entropy is different and only valid for the binary trees).

One could also measure tree repetitiveness with number of runs (i.e. number of phrases in run-length encoding) in string generated by some linearization of the tree. The XBWT seems to not be a good choice in this case, because, as mentioned before, it does not capture the tree shape, moreover even on repetitive tree (i.e. trees with many repeated subtrees) XBWT does not contain many long runs. Still, it would be interesting to develop new linearization techniques or improve XBWT so it would apply for repetitive trees.

Second, we do not support all of the label-related operations, moreover we do not achieve optimal query times. The main challenge is to support more complex operations, like labeled \( \text{level_ancestor} \), while achieving theoretical bounds considered in this work. Previous approaches partitioned (in a rather complex way) the tree into subtrees, by node labels (i.e. one subtree contained only nodes with same label) and achieved at most zeroth-order entropy of labels [44, 28], it may be possible to combine these methods with ours.

Next, it should be possible that the presented structure can be made to support dynamic trees, as all of the used structures have their dynamic equivalent [26, 7, 40]. Still, it is not entirely trivial as we need to maintain the clustering.
Next, as Ferragina et al. [13] mentioned, in some applications nodes can store strings, rather than single labels, where the context for a letter is defined by labels of ancestor nodes and previous letters in a node. It seems that our method should apply in this scenario, contrary to XBWT.

Finally, can the additional space required for level_ancestor query be lowered? We believe that this is the case, as our solution did not use the fact that all weights sum up to $|T|$; moreover it should be possible to apply methods from [40] to obtain $O(|T| \log \log |T|/ \log_\sigma |T|)$ additional space for level_ancestor.

References


Using Statistical Encoding to Achieve Tree Succinctness Never Seen Before

A Additional proofs for Section 2

Proof of Lemma 1. The proof follows by straightforward application of the log sum inequality. To prove the case for $|T|H_k(T|L)$ we use the fact that $\sum_l \sum_K t_{K,l} = |T|$ and that for a fixed $d$ we have $\sum_K \sum_l t_{K,l,d} = t_d$.

$$|T|H_k(T|L) = - \sum_{v \in T} \log \mathbb{P}(d_v|K_v,l_v)$$

$$= - \sum_{v \in T} \log \frac{t_{K_v,l_v,d_v}}{t_{K,l}}$$

$$= - \sum_d \sum_K \sum_l \log \frac{t_{K,l,d}}{t_{K,l}}$$

$$\leq - \sum_d t_d \log \frac{t_d}{|T|}$$

The proof for the case $|T|H_k(L|T)$ is analogous. \hfill $\blacktriangle$
B Additional proofs for Section 3

Proof of Lemma 2. We build the clustering by a simple dfs-based method, starting at the root r.

For a node v, the procedure returns a tree c_v rooted in v along with its size (and also creates some clusters). The actions on v are as follows: we recursively call the procedure on v’s children v_1,...,v_j, let the returned trees be c_1,...,c_j and their sizes s_1,...,s_j. We have two possibilities:

\[ \sum_{i=1}^{j} s_i < m, \] then return a tree rooted at v with all of the returned trees c_1,...,c_j rooted at its children.

\[ \sum_{i=1}^{j} s_i \geq m, \] then we group returned trees greedily: We process trees from left to right, at the beginning we create cluster containing C = \{c_1\}, while \(|C| < m\) we add consecutive c_i’s to C (recall that |C| denotes the number of nodes in trees in C). Then at some point we must add c_j such that |C| \geq m. We output the current cluster C, set C = \{c_{j+1}\} and continue grouping the trees. At the end we return tree containing just one vertex: v.

Finally, we make a cluster of the tree returned by the root of \(T\), regardless of its size.

We now show that the above algorithm satisfies (C1–C4).

Observe that the above procedure always returns a tree, its size is at most m: either it is only a vertex v (in the second case) or the sum of sizes of subtrees is at most m – 1, plus 1 for the v (in the first case). This implies (C2): when we make a cluster out of c_1,...,c_{j+1} then \( \sum_{i=1}^{j} s_i \) is at most m – 1 by the algorithm and \( s_{j+1} \leq m \) by the earlier observation.

It is easy to see from the algorithm that a cluster contains trees rooted at the consecutive siblings, so (C3) holds.

By an easy induction we can also show that in each returned tree the degree of the node v is either 0 (in the second case, when we return tree which consists of only v) or equal to the degree in the input tree \(T\) (in the first case, when the tree contains v and all of its children), thus (C4) holds.

Concerning the total number of clusters, first recall that they are of size at most \(2m-1\), thus there are at least \( \frac{|T|}{2m-1} \geq \frac{|T|}{2m} - 1 \) clusters. To upper bound the number of clusters observe that by the construction the clusters are of size at least m except two cases: the cluster rooted at the root of the whole tree and the clusters that include the last child of the node (but not the node, i.e. they are created in the second case). For the former, there is at most one such a cluster. For the latter, before creating such cluster we created at least one other cluster from trees rooted in siblings whose size is at least m, thus for each cluster of size less than m there must exist corresponding (previously created) cluster with size at least m. Hence the number of clusters of size smaller than m is at most half the number of total clusters plus one, thus there are at most \(2\frac{|T|}{m} + 1\) clusters, as claimed in (C1).

Proof of Lemma 4. Given a tree \(T\) we build a cluster of its nodes using procedure from Lemma 2, create a node for each cluster and add an edge between two nodes u and v if and only if in \(T\) there was an edge from some vertex from cluster \(C_u\) to some vertex in \(C_v\). We label the cluster nodes consistently, i.e. u and v get the same label if and only if their clusters are identical, also in the sense which nodes are port nodes (note though, that the port nodes can have different degree in the input tree). For each label representing the cluster we store its cluster. For simplicity, we assume that the assigned labels are from an ordered set (i.e. set of numbers).

As mentioned before, we store the previously defined degree sequence.
C Additional proofs for Section 4

Proof of Lemma 9. It is a known fact that from sequence of degrees in preorder ordering we can retrieve shape of the tree (we can do it by simple dfs-procedure, which first creates node with given degree, then calls itself recursively; when we recurse back we know which node is next, etc.). From cluster description we can retrieve the sequence of its degrees in preorder ordering. We also can retrieve labels and information which nodes are port. Now for each vertex \( v \) in cluster we know its original context \( K_v \) in \( T \), as we explicitly store context for roots of trees, and other nodes have their context either fully in cluster, or their context is concatenation of some suffix of \( K \) and some path in the cluster. ▶

D Additional material for Section 5

Proof of Lemma 10. First we concatenate degree sequences for each node in \( T' \), according to preorder ordering, obtaining a sequence \( D = d_{v_1,1}, \ldots, d_{v_1,j_1}, d_{v_2,1}, \ldots, d_{v_2,j_2}, d_{v_{|T'|},1}, \ldots, d_{v_{|T'|},|T'|} \). Sum of all \( d_{v,j} \)'s in the sequence is bounded by \( |T'| \), as each \( d_{v,j} \) corresponds to \( d_{v,j} \) edges. We encode each number in the sequence in unary: \( D_u = 0^{d_{v_1,1}} \cdot 1 \ldots 0^{d_{v_{|T'|},|T'|}} \cdot 1 \). Then we build a separate sequence \( B_u \), which marks the borders between nodes in the degree sequence, i.e. \( B_u[z] = 1 \) if and only if at index \( z \) starts the unary degree sequence of some node.

Consider the following example: for nodes \( a, b, c, d, e \) and corresponding degree sequences \( (0), (3, 1), (2), (1, 2), (2, 2) \) we have (the vertical lines \( | \) denote borders of degree sequences and are added for increased readability):

\[
D_u = 1|000101|001|01001|001001 \\
B_u = 1|100000|100|10000|100000
\]

We now construct rank/select data structure for \( D_u \) and \( B_u \). There are multiple approaches that, for static bitvectors, use \( O(|D_u| + |B_u|) = O(|T'|) \) bits and allow both operations in \( O(1) \) time [42].

We describe how to answer a query for node \( u \in T' \) and port node \( v \) in the cluster. Let \( p_u \) be preorder index of \( u \).

Let \( j \) be the point in \( D_u \) where the degree sequence of \( u \) starts, i.e. \( j = select_1(p_u, B_u) \). Let \( p_v \) be a port index of \( v \) in the \( u \)-cluster (recall that we ordered port nodes in each cluster in left-to-right order on leaves). We want to find the beginning of unary description of \( d_{a,p_v} \) (plus one) in \( D_u \): this is the \( p_v - 1 \)-th 1 starting from \( j \)-th element in \( D_u \). The next 1 corresponds to the end of unary description. Let \( u_1, u_2 \) be the beginning and end of this unary description, we can find them in the following way: \( u_1 = select_1(rank_1(j, D_u) + p_v - 1, B_u) + 1 \) and \( u_2 = select_1(rank_1(j, D_u) + p_v, B_u) - 1 \). Observe now that number of 0's in \( D_u \) between \( j \) and \( u_1 \) (with \( j \) and \( u_1 \) is equal to \( i_j \)). Similarly we can get \( i_j \) by counting zeroes between \( u_1 \) and \( u_2 \). Thus \( i_j = rank_0(u_1, D_u) - rank_0(j) \) and \( i_j = rank_0(u_2, D_u) - rank_0(u_1, D_u) - 1 \).

We now proceed to the second query. We find the beginning of description in unary, denoted \( j \) as above. We find position \( u_x \) of \( x \)-th 0 counting from \( j \), we do it by calling \( u_x = select_1(rank_0(j) + x) \). Now we calculate the numbers of 1's between \( j \) and \( u_x \) and simply return this value (+1). That is, we return \( rank_1(u_x) - rank_1(j) + 1 \). ▶

Proof of Lemma 11. Each cluster can be represented by: a number of nodes in the cluster, written as a unary string of length \( m' + 1 \), a bitvector indicating which nodes are port nodes of length \( m' \), balanced parentheses representation of cluster structure, string of labels of length \( m' \). ▶
We give more general version of Theorem 12.

**Theorem 15 (Full version of Theorem 12).** Let $T$ be a labeled tree with labels from an alphabet of size $\sigma \leq |T|^{1-\alpha}, \alpha > 0$. Then we can build a tree structure which requires a number of bits bounded by all of the chosen value from the list below:

1. $|T|H(T) + |T|H_k(L) + O\left(\frac{|T|k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$;
2. $|T|H_k(T|L) + |T|H_k(L) + O\left(\frac{|T|k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$;
3. $|T|H(T) + |T|H_k(L|T) + O\left(\frac{|T|(k+1) \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$.

For general $\sigma$ we can build the structure which size is bounded by any of the values below:

1. $|T|H(T) + |T|H_k(L) + O\left(\frac{|T|k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$;
2. $|T|H_k(T|L) + |T|H_k(L) + O\left(\frac{|T|k \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$;
3. $|T|H(T) + |T|H_k(L|T) + O\left(\frac{|T|(k+1) \log \sigma}{\log_\sigma |T|} + \frac{|T| \log \log_\sigma |T|}{\log_\sigma |T|}\right)$.

It supports firstchild($u$), parent($u$), nextsibling($u$), lca($u,v$), childrank($u$), child($u,i$) and depth($u$) operations in $O(1)$ time; moreover with additional $O(|T| \log \log_\sigma |T|^2)$ bits it can support level-ancestor($v, i$) query in $O(1)$ time.

**Proof of Theorem 12 (and 15).** We start by proving the part for operations firstchild($u$), parent($u$), nextsibling($u$), lca($u,v$), as the rest requires additional structures.

First we consider the case for $\sigma \leq |T|^{1-\alpha}$.

To bound the memory consumption we sum needed space for (T1–T4). (T1), (T3) and (T4) take at most $O(|T|) = O\left(\frac{|T|}{\log_\sigma |T|}\right)$ bits. We bound space for (T2) by Theorem 5, observe that this theorem gives us the same bound as in the claim, moreover this summand dominates bounds for (T1), (T3) and (T4).

One of the crucial part while performing operations on our compressed tree structures is that we can perform preorder-rank and preorder-select in constant time, this allows us to retrieve tree node labels from preorder sequence $P$ of $C(T)$ (and thus to retrieve the cluster), given node of $T'$, in constant time.

We now give the description of operations, let $u$ denote the node and $u'$ the name of its cluster. If the answer can be calculated using only the cluster of $u$ (i.e. when $u$ and the answer is in the same cluster) we return the answer using precomputed tables. Thus in the following we give the description when the answer cannot be computed within the cluster $u'$ alone.

**firstchild($u$):** Using the structure for degree sequence (T3) we find index $i$ of child of $u'$ which represents cluster containing first child of $u$. We call child($u,i$) on the structure for unlabeled tree $T'$ (T1), to get this cluster, the answer is the root of the first tree in this cluster.

**parent($u$):** We call childrank($u'$) on structure for $T'$. This gives us index $i$ such that $u'$ is $i$-th node of node $u'$ representing the cluster containing parent($u$). Now we query degree sequence structure, as it supports also reverse queries (see Lemma 10) obtaining index of port node. Finally, we use precomputed table, i.e. we query the table which for given index of port node and given cluster returns this port node.

**nextsibling($u$):** We call nextsibling($u'$) on structure for $T'$ (T1) and take root of the first tree in the cluster and verify that it has the same parent as $u$.

**lca($u,v$):** let $v'$ be the cluster of $v$. We use the structure for $T'$ (T1): the answer is in the cluster which is represented by node $l = lca(u',v')$ of $T'$ but we still need to determine the actual node inside the cluster. To this end we find nodes $u'', v''$ of $T'$ such that: both are
children of \( l \), they are ancestors of \( u' \) and \( v' \) respectively, and \( u'' \) and \( v'' \) connect to some (port) nodes \( x, y \) such that \( x, y \) are in the cluster represented by node \( l \) and \( \text{lca}(x, y) = \text{lca}(u, v) \). They can be computed as follows: \( u'' = \text{level\_ancestor}(u', \text{depth}(\text{lca}(u', v')) - \text{depth}(u') - 1) \), the case for \( v'' \) is analogous. Having \( u'' \) and \( v'' \) we can, as in the case for parent\( (v) \), reverse query the structure for degree sequence (T3) obtaining indices of \( x \) and \( y \). Finally we use precomputed table (as we want to find lowest common a for port nodes with indices \( x \) and \( y \) in given cluster).

Note that most tree structures allow to find \( u'', v'' \) without calling depth and level\_ancestor, as they are rank/select structures on balanced parenthesis, and it is easy to express this operation using such structures [40].

For general \( \sigma \) we need to only slightly modify our solution.

To encode labels the string \( P \) we use results from [16], which achieves \( |P|H_0(P) + |P| \log \log |P| \) bits. As \( |P| = \mathcal{O}(|\mathcal{T}| / \log_\sigma |\mathcal{T}|) \) this gives required bound.

Additionally, if \( \sigma = \Omega(|\mathcal{T}|) \) we do not have to use precomputed tables, as every cluster has constant number of nodes in it, thus we can perform operations inside the clusters in constant time.

In other case we use tables of size \( \mathcal{O}(|\mathcal{T}|) \), this is still within bounds, as this is dominated by \( \mathcal{O}(|\mathcal{T}| \log \log |\mathcal{T}| / \log_\sigma |\mathcal{T}|) \).

Note, that even Theorem 12 for the case \( \sigma = |\mathcal{T}|^{1-\alpha} \), the guarantee on the redundancy is \( \mathcal{O}(n) \), which is of the same magnitude as the size of the encoding of the tree using parentheses, so for large alphabets this dominates tree entropy.

Also, in Theorem 12 for the case for arbitrary \( \sigma \) we get a slightly worse redundancy, i.e. we have \( \mathcal{O}(|\mathcal{T}| \log \log |\mathcal{T}| / \log_\sigma |\mathcal{T}|) \) factor instead of \( \mathcal{O}(|\mathcal{T}| \log \log_\sigma |\mathcal{T}| / \log_\sigma |\mathcal{T}|) \). Still, even in the case of \( \sigma = \omega(n^{1-\alpha}) \) we can get better bounds (more precisely: \( \mathcal{O}(|\mathcal{T}| \log \log_\sigma |\mathcal{T}| / \log_\sigma |\mathcal{T}|) \), or \( \mathcal{O}(n) \) for large alphabets) by encoding string of labels \( P \) using structure like [4]; but at the cost that operations are slower than \( \mathcal{O}(1) \).

We now prove the rest of Theorem 12, that is we can add even more operations, for more complex operations we will need more involved data structures.

**Succinct partial sums.** To realize more complex operations we make use of structure for succinct partial sums. This problem was widely researched, also in dynamic setting [7]. For our applications, however, it is enough to use a basic, static structure by Raman et al. (RRR) [42].

**Lemma 16.** For a table \( |\mathcal{T}| = n_t \) of nonnegative integers such that \( \sum_i T[i] \leq n \) and \( \sum_i T[i] \leq \mathcal{O}(\log^c n) \), for some constant \( c \), we can construct a structure which answers the following queries in constant time: \( \text{sum}(i, j) : \sum_{y=i}^j T[y]; \text{find}(x) : \text{find first } i \text{ such that } \sum_{y=1}^j T[y] \geq x; \) and consumes \( \mathcal{O}(n_t \log n_t / \log c) \) bits.

**Proof of Lemma 16.** We exploit the fact that all the numbers sum up to \( n \). This allows us to store \( T \) unary, i.e. as string \( 0^{|T|} 1 \ldots 0^{|T|} 1 \). Now using rank/select we can realize desired operations, see [42] for details. For a string with \( n \) zeros and \( n_t \) ones this structure takes \( \log \binom{n+n_t}{n} + o(n_t) \) bits. This can be estimated as: \( \log \binom{n+n_t}{n} + o(n_t) \leq n_t \log(e(n + n_t)/n_t) + o(n) = \mathcal{O}(n_t \log n_t) \), as claimed.
childrank\((v)\), child\((v, i)\). Observe first that if \(v\) and its parent are in the same cluster then childrank\((v)\) can be answered in constant time, as we preprocess all clusters. The same applies to \(v\) and its children in case of child\((v, i)\). Thus in the following we consider only the case when \(v\) is a root of a tree in a cluster (for childrank\((v)\)) or it is a port (for child\((v, i)\)).

The problem with those operations is that one port node \(p\) can connect to multiple clusters, and each cluster can have multiple trees. We solve it by storing for each port node \(p\) a sequence \(T_p = t_{p,1}, \ldots, t_{p,j}\), where \(t_{p,j}\) is the number of children of \(p\) in the \(j\)-th (in left to right order) cluster connecting to \(p\).

Observe that all sequences \(T_p\) contain in total \(|T'| - 1\) numbers, as each number corresponds to one cluster. To make a structure we first concatenate all sequences according to preorder of nodes in \(T'\), and if multiple nodes are in some cluster, we break the ties by left-to-right order on port nodes. Call the concatenated sequence \(T\). Using structure from Lemma 10, for port node \(p\) we can find indices \(i_1, i_2\) which mark where the subsequence corresponding to \(T_p\) starts and ends in \(T\), i.e. \(T[i_1 \ldots i_2 - 1] = T_p\).

We build the structure from Lemma 16 for \(T\), this takes \(O(|T| \log \log_\sigma |T|/\log_\sigma |T|)\) bits, as \(|T| = \mathcal{O}\left(\frac{|T'|}{\log_\sigma |T'|}\right)\) and all elements in \(T\) sum up to at most \(|T|\). We realize childrank\((v)\) as follows: let \(v' \in T'\) be a node representing cluster containing \(v\) (by the assumption: as a root). First we find indices \(i_1, i_2\) corresponding to subsequence \(T_p\), where \(p\) is port node which connects to \(v'\). Let \(j = \text{childrank}(v')\) in \(T'\). Now it is enough to get sum\((i_1, j - 1)\), as this corresponds to number of children in first \(j - i_1\) clusters connected to \(p\), and add to the result the rank of \(v\) in its cluster, the last part is done using look-up tables.

The child\((v, i)\) is analogous: we find indices \(i_1, i_2\) corresponding to \(T_p\). Then we call find\((i + \text{sum}(1, i_1 - 1))\) to get cluster containing child\((v, i)\), as we are interested in first index \(j\) such that \(T[i_1 + \ldots + T[j]] > i\). This way we reduced the problem to find \(i'\)-th node in given cluster, this can be done using precomputed tables.

depth\((v)\). The downside of clustering procedure is that we lose information on depth of vertices. To fix this, we assign to each edge a non-negative natural weight in the following way: Let \(v \in T'\) be any vertex and \(p\) be a port node in cluster represented by parent\((v)\). For an edge \((v, \text{parent}(v))\) we assign depth of \(p\) in cluster represented by parent\((v)\). For example in Figure 2 for edge \((D, B)\) we assign 1, and for edge \((I, B)\) we assign 2. In this way the depth of the cluster \(C\) (alternatively: depth of roots of trees in \(C\)) in \(T\) is the sum of weights of edges from root to node representing \(C\).

A data structure for calculating depths is built using a structure for partial sums: Consider balanced parentheses representation of \(T'\). Then we assign each opening parenthesis to node \(v\) weight \(w(v, \text{parent}(v))\) and each closing parenthesis weight \(-w(v, \text{parent}(v))\). This creates the sequence of numbers, for example, for tree from Figure 2 we have:

\[
\begin{array}{cccccccccccccccc}
& & & & & & & & & & & & & & & & \\
0 & 1 & 1 & -1 & 1 & -1 & 1 & -1 & 2 & -2 & 2 & -2 & 2 & -2 & 2 & -2 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 0
\end{array}
\]

Then we can calculate depth of a cluster by calculating the prefix sum. Observe that our partial sums structure does not work on negative numbers, but we can solve that by creating two structures, one for positive and one for negative number and subtract the result. Finally we use look-up table to find the depth in the cluster.

The total memory consumption is bounded by \(O(|T| \log \log_\sigma |T|/\log_\sigma |T|)\), as all weights sum to at most \(|T|\).
**preorder_rank**(*v*), **preorder_select**(*i*). We again use the structure for succinct partial sums. We create two sequences: First, for each port node consider its preorder rank in *T*, and arrange these ranks in the sequence according to the preorder ordering: \( S_p = \text{preorder}_\text{rank}(p_1), \text{preorder}_\text{rank}(p_2), \ldots, \text{preorder}_\text{rank}(p_{#ports}) \). Now consider the sequence of the increments, i.e.: \( \text{preorder}_\text{rank}(p_1), \text{preorder}_\text{rank}(p_2) - \text{preorder}_\text{rank}(p_1), \ldots, \text{preorder}_\text{rank}(p_{#ports}) - \text{preorder}_\text{rank}(p_{#ports-1}) \). With the structure for succinct partial sums created for this sequence we are able to get the **preorder_rank** in *T* for any port node assuming that we know its position in the sequence *S*\(_p\). We can get this position in the same way as we did for the **childrank** operation: we store additional partial sum structure where elements are number of port nodes in given cluster (see the **childrank** operation description for details). Next, we create the sequence of cluster sizes \( S_c = |C_1|, |C_2|, \ldots, |C_{|T'|}| \), where the order of clusters is determined by the preorder ordering of nodes of *T’* (recall that each node of *T’* corresponds to some cluster).

Now, for a given *v* to get the **preorder_rank**(*v*) we sum up three values: preorder rank in *T’* of port node which is a connected to cluster *C*\(_v\) containing *v*, preorder rank of the node *v* in cluster *C*\(_v\) and the sum of sizes of clusters *C*\(_j\) forming subtrees which are connected to the port nodes *p* in *C*\(_v\) such that *p* is before *v* in preorder ordering of nodes in *C*\(_v\). To get the first summand, we use previously described structure, to get the second summand we use precomputed tables. Now observe the sum of sizes of clusters *C*\(_j\) form consecutive interval in *S*\(_c\). Thus it is enough to get the preorder rank in *T’* of first and last such cluster. This is straightforward when the structure for *T’* supports **preorder_rank** and **rightmost_leaf** operation (e.g. we can use the structure from [40]): let *p* and *p’* be first and last port nodes which are before *v* in *C*\(_v\), we find the preorder rank of the leftmost child of *p* and preorder rank of rightmost cluster in subtree connected to *p*. As the elements in sequences sum up to at most |*T’|, the space is bounded by \( \mathcal{O}(|T| \log \log_\sigma |T| / \log_\sigma |T|) \).

The **preorder_select**(*i*) operation is more involved. For each cluster *C* we define \( p_r(C) \) — the position in preorder ordering of leftmost root of *C* (note that *C* can be a forest) according to preorder ordering of vertices of *T*. Consider the sequence of increments \( D = p_r(C_1), p_r(C_2) - p_r(C_1), p_r(C_3) - p_r(C_2), \ldots, p_r(C_{|T’|-1}) - p_r(C_{|T’|}) \), where the clusters are ordered to preorder ordering according to *T’*.

Now, by using succinct partial sums structure on the sequence *D*, for a given *i* we can find two (consecutive in preorder ordering on *T’*) clusters *C*\(_j\), *C*\(_{j+1}\) such that \( p_r(C_j) \leq i < p_r(C_{j+1}) \). Observe that there are two possibilities: either *C*\(_j\) is parent of *C*\(_{j+1}\) in *T’* and *C*\(_{j+1}\) is the first child of *C*\(_j\), or *C*\(_j\) is the rightmost node of subtree rooted in parent *C*\(_p\) of *C*\(_{j+1}\) in *T’*. If *C*\(_j\) is a parent of *C*\(_{j+1}\), it is sufficient to find a node with preorder rank of \( i - p_r(C_{j+1}) \), as *C*\(_{j+1}\) is the first child of *C*\(_j\), for this we can use precomputed tables.

If *C*\(_j\) is a rightmost node of subtree rooted in *C*\(_p\) (and hence a leaf) we have three possibilities: either the requested node is in *C*\(_j\) or in *C*\(_p\) or in parent of *C*\(_p\). If \( |C_j| + p_r(C_j) \leq i \) the node is in *C*\(_j\) and so we use the precomputed tables as in former case.

Otherwise we find two (not necessarily different) port nodes in *C*\(_p\), *n*\(_j\) and *n*\(_{j+1}\), which are the nodes on the path from *C*\(_j\) to *C*\(_p\) and *C*\(_{j+1}\) to *C*\(_p\), respectively. Let *n*\(_{r_j}\) and *n*\(_{r_{j+1}}\) be the roots of subtrees of *C*\(_p\) which contain *n*\(_j\) and *n*\(_{j+1}\), respectively. Now the searched node is either a parent of *n*\(_{r_j}\) and *n*\(_{r_{j+1}}\) (observe that they must share a parent), or in one of the subtrees in *C*\(_p\) rooted at either *n*\(_{r_j}\) or *n*\(_{r_{j+1}}\). We check the former case by simply calling **preorder_rank** for the parent of *n*\(_{r_j}\). For the latter case it is enough to get the node which is \( i - (p_r(C_j) + |C_j| - 1) \) positions after *n*\(_j\) in *C*\(_p\) according to preorder ordering of nodes in *C*\(_p\). For this we can use precomputed tables (i.e. we store tables which allow to call **preorder_rank** and **preorder_select** for vertices of a given cluster).
Consider the sequence of leaves of \( T \), \( L = v_1, v_2, \ldots, v_{|L|} \) where leaves are ordered from left to right. Now consider the grouping of leaves such that two leaves are in the same group if and only if they are in the same cluster in \( T' \). We obtain sequence of groups \( L_G = G_1, G_2, \ldots, G_{|L|} \). We build the succinct partial sum structure for sequence of group sizes, \( LS_G = |G_1|, |G_2|, \ldots, |G_{|L|}| \).

Assuming that our structure for \( T' \) supports \( \text{leaf}_{\text{rank}}(u) \) and \( \text{leaf}_{\text{select}}(j) \) (this can be achieved by using the structure from \([40]\) for \( T' \)) we can realize the operations as follow: For \( \text{leaf}_{\text{rank}}(v) \) we first call the \( \text{leaf}_{\text{rank}}(u) \), where \( u \) is vertex in \( T' \) representing cluster \( C_v \), this way we know how many clusters containing leaves are to the left of \( C_v \) in \( T' \). We use the structure for partial sums to get the number of leaves in clusters to the left of \( v \), for the vertices in \( C_v \) to the left of \( v \) we use precomputed tables. Analogously, we realize \( \text{leaf}_{\text{select}}(i) \) by first using our structure for partial sums, this allows to identify the index \( i' \) in \( LS_G \), then we call \( \text{leaf}_{\text{select}}(i') \) on structure for \( T' \), we also use precomputed tables to identify the leaf inside the cluster.

**Lemma 17.** Let \( T', |T'| = t \) be a tree where each edge is assigned a weight of at most \( O(\log n) \), for some \( n \). We can build structure which consumes \( O(t(\log \log |T|)^2 / \log_3 |T|) \) bits of memory and allows to answer weighted level ancestor queries in \( O(1) \) time.

With the structure from Lemma 17 the query \( \text{level}_{\text{ancestor}}(v, i) \) is easy: first we check if the answer is in the same cluster using preprocessed array. If not we find cluster containing the answer, we do it by asking for \( \text{level}_{\text{ancestor}}(C, i \text{-depth}(C, v)) \), where \( \text{depth}(C, v) \) is depth of \( v \) in cluster \( C \) containing \( v \). There is similar problem as in the case of lca query, that is we also need to find the port node on path from given vertex \( v \) to its \( i \)-th ancestor. This may be solved in the same manner as in the case for lca.

Now we give the construction for weighted level ancestor structure. Note that there are multiple ways of doing this \([22, 40, 33]\). We use the tree partitioning approach, yet the one that operates on sequence of numbers, as in case for depth, should also be applicable, \([33]\) shows similar method (and uses same additional space), yet for simplicity we choose to stick with solution which partition the tree into subtrees as we already defined most of the required machinery. Note that tree partitioning method \([22]\), which we refer to, partitioned the tree a few times, we do it once and use stronger result \([40]\) for the simplicity of proof. Also, the partitioning from \([22]\) may be used instead of our method.

**Proof of Lemma 17.** We use the idea from \([22]\). We first cluster the tree according to Lemma 2 with \( m = \Theta(\log^3 n) \). We obtain a smaller tree \( T'' \) of size \( |T''| = O \left( \frac{1}{\log^3 n} \right) \). We store labels of \( T'' \) and the descriptions of clusters naively, without the entropy coder. We also store additional structure for navigation of \( T'' \), including degree sequences, observe that it takes at most \( O(t) \) bits.
Similarly, as in the case for depth, we assign weights to edges of $T''$ but we have to remember that the input tree is weighted as well. Let $v \in T''$ be any vertex and $p$ be a port node in a cluster represented by $\text{parent}(v)$, call the cluster $C_p$. For an edge $(v, \text{parent}(v))$ we assign the sum of the weights on the path from $p$ to the root of the tree containing $p$ in $C_p$ (i.e. the weighted depth of $p$ in $C_p$).

Observe that the weights in the $T''$ are of order $O(\log^4 n)$. For such weighted tree we can build structure which supports $\text{level\_ancestor}$ queries in $O(1)$ time and use $O(|T''|\log^2 n) = o(n)$ bits, using result from [5].

Now for each smaller tree we can build structure from [40]. For a tree of size $t'$ and with weights limited by $O(\log^4 n)$ this structure takes $O(t' \log t' \log(t' \log^4 n)) = O(t' \log \log n)^2$ bits. Summing over all trees, we get $O(t(t \log \log n)^2)$, as claimed. For each such tree we additionally store the information, which nodes are port nodes in a way that allow to retrieve $i$-th port node. This can be achieved by storing the bitmap for each small tree (in which each $j$-th element indicates if $j$ leaf is port node or not) and applying rank/select structure (this consumes $O(t)$ bits for all trees). Observe that we can even explicitly list all of the port nodes: we do not have to use space efficient solution, as there are at most $|T''|$ such nodes, so even consuming $O(\log^2 n)$ bits for port node is sufficient.

We perform $\text{level\_ancestor}$ operation in the same manner as previously described when applying Lemma 17, i.e. we first check if the answer is in the same tree, if yes we can output the answer as we can perform operations on small trees in $O(1)$ time [40], if not we use combination of depth and $\text{level\_ancestor}$ queries on structure for $T''$ in the same way as we did for lca query (see proof of Theorem 12). It is possible as structure [5] supports all of the required operations (or can be easily adapted to support by adding additional tree structure, as the structure for $T''$ can consume up to $O(\log^2 n)$ bits per node, in particular this means that we can even preprocess all answer for depth queries for $T''$).

The only nontrivial thing left is that we would like to not only find a node in our structure but also find corresponding node in structure for $T'$. To this end we show that we can return preorder position of given node in $T'$, this is sufficient as structure for $T'$ has $\text{preorder\_select}$ operation.

To this end we explicitly store preorder and subtree size for each port node, observe that this consumes at most $O(T'' \log t) = O(t)$ bits. Now given a node $v$ in some cluster to find preorder rank of this node in $T'$ we sum the following values: preorder rank of $v$ in cluster $C$ containing $v$, the rank of port node $p_v$ which is connected to cluster $C$, and the sizes of subtrees $T_i$ which are connected to port nodes $c_i$ of $C$, such that $c_i$ precedes $v$ in preorder ordering in $C$. To find the sum of sizes of $c_i$ we first find how many $c_i$'s precede $v$ in preorder ordering in $C$, to this end we use rank/select structure for binary vector $B_C$ where $B_C[i] = 1$ if and only if $i$-th node according to preorder ordering in $C$ is a port node. Then we use the structure for partial sums (again, we do not have to use succinct structure as there are at most $O(|T''|)$ elements in total).

---

**E Additional material for Section 6**

The following Lemmas says that if we partition the clusters into groups according to their $k$-letter contexts and encode each group separately with zeroth-order entropy we can get better estimation than encoding them together.
Lemma 18. Let $T'$ be a labeled cluster tree from Lemma 4 for parameter $m$, obtained from $T$. For each $k$-letter context $K_i$, let $P_{K_i}$ be a concatenation of labels of $T'$ which are preceded by this context (i.e. each root $v$ in each cluster is preceded by the context $K_i$ in $T$). Then all of the following inequalities hold:

1. $\sum_i |P_{K_i}| H_0(P_{K_i}) \leq |T| H(T) + |T| H_k(L) + O\left(\frac{|T| \log m}{m}\right)$;
2. $\sum_i |P_{K_i}| H_0(P_{K_i}) \leq |T| H_k(T|L) + |T| H_k(L) + O\left(\frac{|T| \log m}{m}\right)$;
3. $\sum_i |P_{K_i}| H_0(P_{K_i}) \leq |T| H(T) + |T| H_k(L|T) + O\left(\frac{|T| \log m}{m} + |T| \log m\right)$.

Proof of Lemma 18. The proof is very similar to the proof of Theorem 5. For each $P_{K_i}$ we apply the Lemma 6: we use almost the same values of $q$ function for each cluster but we do not need to multiply it by $q(K_C)$, i.e. we define $q(C) = q(N_C) \cdot q(V_C)$. For detailed definition of $q$ see proof of Theorem 5. It is easy to check that without this factor we arrive at the claim.

Lemma 19. Let $T'$ be a labeled cluster tree from Lemma 4 for parameter $m$, obtained from $T$. Let $P$ be a string obtained by concatenation of labels of $T'$. Then we can encode $P$ in a way, that, given context $K_{P[i]}$, we can retrieve $P[i]$ in constant time. The encoding is bounded by the following values:

1. $|T| H(T) + |T| H_k(L) + O\left(\frac{|T| \log m+\log|T|}{m} + \sigma^{k+m} \cdot 2^m \cdot \log |T|\right)$;
2. $|T| H_k(T|L) + |T| H_k(L) + O\left(\frac{|T| \log m+\log|T|}{m} + \sigma^{k+m} \cdot 2^m \cdot \log |T|\right)$;
3. $|T| H(T) + |T| H_k(L|T) + O\left(\frac{|T| \log \sigma}{m} + \frac{|T| \log \log|T|}{m} + \sigma^{k+m} \cdot 2^m \cdot \log |T|\right)$.

Proof of Lemma 19. We use Lemma 18. For each $P_{K_i}$ we generate codes using Huffman encoding, this allows us to encode each $P_{K_i}$ using $|P_{K_i}| H_0(P_{K_i}) + |P_{K_i}|$ bits, as we lose at most 1 bit per code (see [16] for example), plus additional $O(2^m \sigma^m \log |T'|)$ bits for Huffman dictionary. Summing this over all contexts $K_i$ yields the bound, by Lemma 18.

Denote $c_v$ as Huffman code for vertex $v$ of $T'$. Let $T_C$ be the concatenation of all codes $c_v$ according to order in string $P$. As each code is of length at most $O(\log |T'|)$, given its start and end in $T_C$ we can decode it in constant time. We store the bitmap of length $|T_C|$ where $T_C[j] = 1$ if and only if at $j$-th is the beginning of some code. This bitmap has length at most $|T_C| \log |T|$ and has $|T'|$ ones. For such a bitmap we build rank/select structure, using result by Raman et al. [42] this takes $O(|T| \log \log |T|)$ bits. Using rank/select we can retrieve the starting position of $i$-th code by simply calling select($i$) (same idea was used in [23]).

Now we would like to simply apply Lemma 19, which says that we can encode labels of $T'$ more efficiently, yet there is one major difficulty: the Lemma states that to decode $P[i]$ we need to know the context. The idea is that we choose $|T'| / d$ nodes for which we store the context, for rest we can retrieve the contexts in time $O(d)$ by traversing $T'$ and decoding them on the way.

Proof of Theorem 13. We use almost the same structures as in the simpler case, i.e. the structure from Theorem 12, the only difference is that instead encoding preorder sequence $P$ with structure using space proportional to zeroth order entropy we apply Lemma 19. We choose $m = \beta \log \sigma |T|$ so that $2 \beta + \alpha < 1$ and $\beta < \frac{1}{2}$, so that the precomputed tables use $o(|T|)$ space. As each operation in Theorem 12 accessed elements of $P$ constant number
of times it is sufficient to show how to access it in time $O(\log |\mathcal{T}| / \log \log |\mathcal{T}|)$ time. By Lemma 19 this leaves us with problem of finding context for each node in aforementioned complexity.

Let $d = \lceil \log |\mathcal{T}| / \log \log |\mathcal{T}| \rceil$. We choose at most $O((|\mathcal{T}'| \log \log |\mathcal{T}|) / \log |\mathcal{T}|) = O(|\mathcal{T}'| / d)$ nodes, for which we store the context explicitly, in the following way: We store the context for the root using $[k \log \sigma]$ bits. We partition the nodes into $d$ classes $C_i$ depending on their depth modulo $d$, i.e. in the class $C_i$ there are nodes at depth $dj + i$, $j \geq 0$. Then there is a class having at most $\mathcal{T}' / d$ such nodes, we choose all nodes in this class and store their contexts.

Now we show, assuming we know the contexts for chosen nodes, that given a node we can retrieve its context in $O(d)$ time. If we want to compute the context for some node $v$ we first check whether it is stored explicitly. If not, we look at nodes on path from $v$ to the root, until we find a node $u$ which has its context stored. Call the visited nodes $v, v_1, v_2, \ldots, v_t, u$. Observe that we visited at most $O(d)$ nodes that way. As we know the context for $u$, now we can decode the node $u$, and determine the context for $v_i, v_{i-1}, \ldots, v_1, v$. To read the labels in $u$ which precede $v_i$ in constant time we first find port node which connects to $u$ (as in the proof of Theorem 12) and use the precomputed tables.

The only nontrivial thing left to explain is how to store the contexts for chosen $O(|\mathcal{T}'| / d)$ nodes and check which nodes have their contexts stored. We concatenate all contexts for chosen $O(|\mathcal{T}'| / d)$ nodes according to their order in preorder ordering. On top of that we store binary vector $B$ which satisfies $B[i] = 1$ if and only if $i$-th node of $\mathcal{T}'$ in preorder ordering has its context stored. We build rank/select structure for $B$, as we have preorder-rank and preorder-select operation for $\mathcal{T}'$ in constant time, for a given node we can check in constant time if the node have its index stored or not. As each context has the same bit-length to decode context for node which is $j$-th in preorder ordering we look at position $(j - 1)[k \log \sigma]$.

The total space for storing the context is $O(|\mathcal{T}'| k \log \sigma / d) = O(|\mathcal{T}'| \log \log |\mathcal{T}|)$, summing that up with space bound from Lemma 19 yields the claim.

## Additional material for Section 7

### Proof of Theorem 14.

The first part of the theorem is easy: if $\sigma$ is constant we can construct a separate structure for each letter. For each letter $a$ we build a separate degree sequence, level ancestor structure and depth structure; observe that all of those structures support the weighted case when we assign each vertex weight of 0 or 1, so it is sufficient to assign nodes labeled with $a$ value 1 and for the rest value 0. Similar idea was mentioned in [44, 28, 22].

For the next two parts we show how to adapt rank/select structures over large alphabets to support childrank/childselect queries.

For the second part, when $\sigma = O(\log^{1+o(1)} |\mathcal{T}|)$, we use result by Belazzougui et al. [4]. They show (at discussion at above Theorem 5.7 in [4]) how for the sequence $S$ divided into $O(|S| / m)$ blocks of length at most $m$, for some $m$, construct rank/select structure for large alphabets, assuming that we can answer queries in time $O(t)$ in blocks, such that it takes $O(t)$ time for query and consumes additional $|S| \log \sigma / m + O\left(\frac{|S|}{m} + \frac{\sigma |S| / m}{\log \sigma |S| / m}\right)$ bits [in [4] the assumption is that we can answer queries in blocks in $O(1)$ time but as the operations on additional structure cost constant time, our claim also holds]. The solution uses only succinct bitmaps by Raman et al. [43] and precomputed tables for additional data. Now we can define sequence $S$ as concatenation of labels of roots of cluster, where clusters are ordered by preorder ordering, this gives our blocked sequence (where blocks correspond to clusters). Observe that labeled childrank/childselect operations can easily be reduced to
labeled rank/select in string $S$, all we need to do is to know where the sequence for children of a given vertex $v$ begins in $S$. Fortunately, this can be done in same manner as in Lemma 10, that is, we use structure for degree sequence. As in our case $m = \log_\sigma |T|$, we can store precomputed tables to answer rank/select queries for each cluster. The additional space is $o(|S| \log \sigma)$ and clearly $|S| \leq |T|$.

For the last part we use Lemma 3 from [3]. The lemma states that for a string $|S|$ if, for a given $i$, we can access $i$-th element in time $O(t)$ then we can, using additional $o(|S| \log \sigma)$ bits, support labeled rank/select operations in time $O(t \log \log^{1+\epsilon} \sigma)$. We use the same reduction as in the case for $\sigma = O(\log^{1+o(1)} |S|)$, i.e. we set $S$ as concatenation of labels of roots of clusters, where clusters are ordered by preorder ordering. As in previous case, we use node degree sequence and tree structure to retrieve $i$-th character in $|S|$ (i.e. we first find appropriate cluster and use precomputed tables).