Succinct Indexes for Strings, Binary Relations and Multi-labeled Trees

Jérémy Barbay *, Meng He *,
J. Ian Munro *, and S. Srinivasa Rao †

*David R. Cheriton School of Computer Science,
University of Waterloo, Canada
{jbarbay, mhe, imunro}@uwaterloo.ca
†Computational Logic and Algorithms group,
IT University of Copenhagen, Denmark
ssrao@itu.dk

Technical Report CS-2006-24
of the David R. Cheriton School of Computer Science

Abstract

We define and design succinct indexes for several abstract data types (ADTs). The concept is to design auxiliary data structures called succinct indexes that occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. As opposed to succinct encodings, The main advantage of succinct indexes is that we make assumptions only on the ADT through which the main data is accessed, rather than the way in which the data is encoded. This allows more freedom in the encoding of the main data. In this paper, we present succinct indexes for various data types, namely strings, binary relations and multi-labeled trees. Given the support for the interface of the ADTs of these data types, we can support various useful operations efficiently by constructing succinct indexes for them. When the operators in the ADTs are supported in constant time, our results are comparable to previous results, while allowing more flexibility in the encoding of the given data.

Using our techniques, we design the first succinct encoding that represents a string of length n over alphabet $[\sigma]$ using $nH_k + o(n \lg \sigma)$ bits [1] that support access / rank / select operations in $o((\lg \lg \sigma)^3)$ time. We also design the first succinct text index using $nH_k + o(n \lg \sigma)$ bits that supports pattern matching queries in $O(m \lg \lg \sigma + occ \lg^{1+\epsilon} n \lg \lg \sigma)$ time, for a given pattern of length m. Previous results on these two problems either have a $\lg \sigma$ factor instead of $\lg \lg \sigma$ in terms of running time, or are not compressible, but our results do not have such problems. More results are reported in the paper.

[1] We use $\lg n$ to denote $\lceil \log_2 n \rceil$.

1 Introduction

As a result of the rapid growth of large electronic data sets, a new trend of the design of data structures is to represent them succinctly. *Succinct data structures* were first proposed by Jacobson [12] to encode bit vectors, (unlabeled) trees and planar graphs in space close to the information-theoretic lower bound, while supporting efficient navigational operations. This technique was successfully applied to various other abstract data types, such as dictionaries, strings, binary relations [2] and labeled trees [2, 8]. In most of the previous results, researchers usually encode the given data, and use both the encoding and auxiliary data structures to support various operations. Therefore, these techniques usually require the given data to be stored in specific formats. We thus call this type of design *succinct encodings* of data structures.

The concept of *succinct indexes* was originally proposed to prove the lower bounds on the space required to encode some data structures: it constrains the definition of the encoding to the index [5, 13]. In this paper, we apply the idea to the design of succinct data structures. Given an ADT, our goal is to design auxiliary data structures (i.e. succinct indexes) that occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. Although succinct indexes and succinct encodings are closely related, they are different concepts. Succinct indexes make assumptions only on the ADT through which the given data is accessed, while succinct encodings represent data in specific formats. Succinct indexes are also more difficult to design: one can design a succinct encoding from a succinct index, but the converse is not true.

Although succinct indexes were previously presented only as a technical restriction to prove lower bounds, we argue than in fact they are more adequate to the design of a library of succinct tools for multiple usages than succinct encodings, and that they are even directly required in certain applications. Some of the advantages of succinct indexes over succinct encodings are:

1. A succinct encoding requires the given data to be stored in a specific format. However, a succinct index applies to any encoding of the given data that supports the ADT. Thus for succinct indexes, the given data can be either stored to achieve compression or to achieve optimal support of the operations defined in the ADT.
2. The existence of two succinct encodings supporting two non-identical sets of operations over the same ADT does not imply the existence of a single encoding supporting the union of the two sets of operations without storing the given data twice, because they may not store it in the same format. However, we can always combine two different succinct indexes for the same ADT to yield one index that supports the union of the two sets of the operations.
3. In some cases, we do not need to store the given data explicitly because they can be computed from different but related data and still support the operations defined in the ADT. Hence a succinct index is the only additional memory cost.

In this paper, we design succinct indexes for strings, binary relations and multi-labeled trees. Given the support for the interface of these ADTs, we can support an extended set of operations within a $o((\lg\lg\sigma)^3)$ factor of the running time of the operators in ADTs, where σ is the size of the alphabet of the characters or labels (See Section 2 for definitions). The succinct indexes occupy negligible space compared to the information-theoretic lower bound for representing the given data.

Based on the succinct indexes for strings, we design the first succinct encoding that represents a string of length n over alphabet $[\sigma]$ using $nH_k+o(n\lg\sigma)$ bits [2], which supports access / rank / select

[2] H_k denotes the k$^{\text{th}}$ order entropy of a given string [10].

operations in $o((\lg\lg\sigma)^3)$ time. We also design the first succinct text index using $nH_k + o(n\lg\sigma)$ bits that supports searching for a pattern of length m in $O(m\lg\lg\sigma + occ\lg^{1+\epsilon} n\lg\lg\sigma)$ time.

2 Background

Here we outline the design of succinct data structures for several abstract data types. We cite the results that we use in the design of succinct indexes, and the results which we improve upon.

2.1 Bit Vectors

A key structure we use is a bit vector B of size n that supports *rank* and *select* operations. We assume that the positions in B are numbered $1, 2, ..., n$. For $\alpha \in \{0, 1\}$, the operator $\texttt{bin_rank}_B(\alpha, x)$ returns the number of occurrences of α in B before position x, and the operator $\texttt{bin_select}_B(\alpha, r)$ returns the position of rth α in B. We omit the subscript B when it is clear from the context. Lemma 1 addresses the problem, in which part (a) is from [4, 12], and part (b) is from [16].

Lemma 1. *For a bit vector B of length n, we can support the access to each bit,* `bin_rank`*, and* `bin_select` *in $O(1)$ time using either: (a) $n + o(n)$ bits, or (b) $\lg\binom{n}{v} + O(n\lg\lg n/\lg n)$ bits, where v is the number of 1s in B .*

A less powerful version of $\texttt{bin_rank}(1, x)$, denoted $\texttt{bin_rank}'(1, x)$, returns the number of 1s in B before position x only if $B[x] = 1$. The following lemma addresses this problem.

Lemma 2 ([16]). *A bit vector B of length n with v 1s can be represented using $\lg\binom{n}{v} + o(v) + O(\lg\lg n)$ bits to support access to each bit, $\texttt{bin_rank}'(1, x)$, and $\texttt{bin_select}(1, r)$ in $O(1)$ time.*

2.2 Strings and Binary Relations

Grossi *et al.* [10] generalized `bin_rank` and `bin_select` operators to strings (or sequences) over alphabets of arbitrary size σ, and the operations include: $\texttt{string_rank}(\alpha, x)$, which returns the number of occurrences of α before position x; $\texttt{string_select}(\alpha, r)$, which returns the position of the rth occurrence of α in the string if any, or ∞; and $\texttt{string_access}(x)$, which returns the character at position x in the string. They gave an encoding that takes $(n + o(n))\lg\sigma$ bits to support these three operators in $O(\lg\sigma)$ time, where n is the length of the string. Golynski *et al.* [9] gave an encodings that uses $n(\lg\sigma + o(\lg\sigma))$ bits and supports $\texttt{string_rank}(\alpha, x)$ and $\texttt{string_access}(x)$ in $O(\lg\lg\sigma)$ time, and $\texttt{string_select}(\alpha, r)$ in constant time.

Barbay *et al.* [2] extended the problem to the encoding of sequences of n objects where each object can be associated a subset of labels from $[\sigma]$, this association being defined by a binary relation of t pairs from $[n]\times[\sigma]$. The operations include: $\texttt{label_rank}(\alpha, x)$, which returns the number of objects labeled α preceding x; $\texttt{label_select}(\alpha, r)$, which returns the position of the rth object labeled α if any, or ∞; and $\texttt{label_access}(x, \alpha)$, which checks whether object x is associated with label α. Their representation supports `label_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_select` in constant time using $t(\lg\sigma + o(\lg\sigma))$ bits [3].

2.3 Ordinal Trees

An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their rank. The preorder and postorder traversals of trees are well-known. We introduce in this paper a different order on the traversal of tree nodes, namely `DFUDS`, which is the order associated with

[3] In this paper, we assume that each object is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. It is shown in [2] how to extend the results to other cases by simple reductions.

the *depth first unary degree sequence* [3] representation, where all the children of a node are listed before its other descendants (see Figure 2 in Appendix A for an example).

Various succinct data structures have been designed to represent ordinal trees ([3, 8, 12, 15]). Among them we list the most recent result by Geary *et al.* [8]. They proposed a structure that uses $2n + o(n)$ bits, which is close to the lower bound suggested by information theory ($2n - \Theta(\lg n)$ bits), and supports a rich set of navigational operations (we refer to each node by its preorder number):

- `child`(x, i), the ith child of node x for $i \geq 1$;
- `child_rank`(x), the number of left siblings of node x;
- `depth`(x), the depth of x, i.e. the number of edges in the rooted path to x;
- `level_anc`(x, i), the ith ancestor of node x for $i \geq 0$ (given a node x at depth d, its ith ancestor is the ancestor of x at depth $d - i$);
- `nbdesc`(x), the number of descendants of x;
- `degree`(x), the degree of x, i.e. the number of its children.

2.4 Labeled and Multi-Labeled Trees

A *Labeled tree* is a tree in which each node is associated with a label from a given alphabet $[\sigma]$, while in a *multi-labeled tree*, each node is associated with at least one label [4]. We use n to denote the number of nodes in a labeled / multi-labeled tree, and t to denote the total number of node-label pairs in a multi-labeled tree [5]. As we only consider ordinal trees, we assume that labeled / multi-labeled trees are ordinal trees in the rest of the paper.

Geary *et al.* extended the operators in Section 2.3 to support some operations on labeled trees in constant time, but their data structure uses $2n + n(\lg \sigma + O(\sigma \lg \lg \lg n / \lg \lg n))$ bits, which is much more than the asymptotic lower bound of $n\,(\lg \sigma - o(\lg \sigma))$ suggested by information theory when σ is large. Ferragina *et al.* [6] proposed another structure for labeled trees that locates the first child[6] of a given node x labeled α in constant time, but this structure does not support the retrieval of the ancestors or descendants by labels efficiently. It also uses $2n \lg \sigma + O(n)$ bits, which is almost twice the minimum space required to encode the tree. Barbay *et al.* [2] gave an encoding for labeled trees using $n\,(\lg \sigma + o(\lg \sigma))$ bits to support the retrieval of the ancestors or descendants by labels in $O(\lg \lg \sigma)$ time.

3 Succinct Indexes

In this section, we design succinct indexes for strings, binary relations, and multi-labeled trees. We present succinct indexes in two general steps: first, we define the interface of the ADTs; second, we design succinct indexes for the ADTs defined. Note that for some data structures, the definition of ADTs is trivial (such as strings), while for others, it may be not (such as multi-labeled trees). In any case, we always try to make our ADT as basic as possible.

3.1 Strings

We first design succinct indexes for a given string S of length n over alphabet $[\sigma]$. We adopt the common assumption that $\sigma \leq n$ (otherwise, we can reduce the alphabet size to the number of characters that occur in the string). We define the ADT of a string through the `string_access`

[4] We use $[n]$ to denote the set $\{1, 2, ..., n\}$.

[5] In this paper, we assume that each node of the tree is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. It is shown in [2] how to extend the results to other cases by simple reductions.

[6] Ferragina *et al.*'s encoding also supports finding all the children of x labeled α in constant time per child.

operator that returns the element of the string at any given location. To generalize the operators on strings defined in Section 2.2 to include "negative" searches, we define the concept of patterns on labels as follows (we use the array notation for strings to refer to its characters and substrings):

Definition 1. *Consider a string $S[1..n]$ over $[\sigma]$. We say a position $x \in [n]$ matches a label-pattern $\alpha \in [\sigma]$ if $S[x] = \alpha$. A position $x \in [n]$ matches a label-pattern $\bar{\alpha}$ if $S[x] \neq \alpha$. We also define $[\bar{\sigma}]$ to be the set $\{\overline{1}, \ldots, \overline{\sigma}\}$.*

It would be quite reasonable to extend the notation of a label-pattern to be (at least) an arbitrary subset of $[\sigma]$, but we will not. Consider, then, the following operators:

Definition 2. *Consider a string $S \in [\sigma]^n$, a label-pattern $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and a position $x \in [n]$ in S. The* α-predecessor of position x, *denoted by* `string_pred`(α, x), *is the last position matching α before position x if it exists. Similarly, the* α-successor of position x, *denoted by* `string_succ`(α, x), *is the first position matching α after position x if it exists.*

To illustrate the operations above, consider the string *bbaaacdd*. We have `string_pred`$(a, 7) = 5$, as position 5 is the last position in the string before position 7 whose character is a. We also have `string_pred`$(\overline{a}, 5) = 2$, as position 2 is the last position before position 5 whose character is not a. With these definitions, we now begin to state our results.

Lemma 3. *Given support for* `string_access` *in $O(f(n, \sigma))$ time on a string $S \in [\sigma]^n$, using a succinct index of $n \cdot o(\lg \sigma)$ bits, we can support* `string_rank` *for any label-pattern in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time, and* `string_select` *for any label-pattern $\alpha \in [\sigma]$, in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time.*

Proof. We first observe that for the `string_rank` operator, as `string_rank`$(\overline{\alpha}, x) = x -$ `string_rank`$(\alpha, x) - 1$ for $\alpha \in [\sigma]$, we only need to show how to support it for $\alpha \in [\sigma]$.

We conceptually treat the given string S as an $n \times \sigma$ table E with rows indexed by $1, 2, ..., \sigma$ and columns by $1, 2, ..., n$. For any $\alpha \in [\sigma]$ and $x \in [n]$, Entry $E[\alpha][x] = 1$ if $S[x] = \alpha$, and $E[\alpha][x] = 0$ otherwise. When reading E in row major order, we have a conceptual bit vector A of length σn with exactly n 1s in it. As in [9], we divide A into blocks of size σ. The *cardinality* of a block is the number of 1s in it. In order to make use of `string_access` to support operators on blocks, we group blocks into chunks. To be specific, we conceptually divide S into chunks of length σ (we assume that n is divisible by σ for simplicity), denoted by $C_1, C_2, ..., C_{[n/\sigma]}$. We observe that $C_i[j] = S[(i-1)\sigma + j]$, where $i \in [n/\sigma]$ and $j \in [\sigma]$. We also observe that each chunk consists of exactly σ blocks, one for each row of the chunk. We denote the block corresponding to the αth row of C_i by $C_{i,\alpha}$, where $\alpha \in [\sigma]$. We store the following data structures:

- For the entire string, we construct a bit vector B which stores the cardinalities of all the blocks in unary, by the order they appear in A, i.e. $B = 1^{l_1}01^{l_2}0...1^{l_n}0$, where l_i is the cardinality of the ith block of A. The length of B is $2n$, as there are exactly n 1s in A, and n blocks. We store it using Part (a) of Lemma 1 in $2n + o(n)$ bits.
- For each chunk C_i, we construct a bit vector X_i that stores the cardinalities of the blocks in C_i in unary from top to bottom, i.e. $X_i = 01^{l_{i,1}}01^{l_{i,2}}0...1^{l_{i,\sigma}}0$, where $l_{i,\alpha}$ is the cardinality of block $C_{i,\alpha}$. We store it in in $2\sigma + o(\sigma)$ bits using Part (a) of Lemma 1 as its length is 2σ.
- For each chunk C_i, we construct an array R_i such that $R_i[j] =$ `bin_rank`$_D(1, j) \bmod \lg\sigma$, where D is block $C_{i,C_i[j]}$. Each element of R_i is an integer in the range $[0, \lg\sigma - 1]$, so R_i can be stored in $\sigma \lg\lg\sigma$ bits.

- For each chunk C_i, we construct an auxiliary structure P_i to support the computation of a certain permutation defined later in the proof;
- For each block $C_{i,\alpha}$, let $F_{i,\alpha}$ be a conceptual, "sparsified" bit vector for $C_{i,\alpha}$, in which only every $\lg\sigma^{\text{th}}$ 1 of $C_{i,\alpha}$ is present (i.e. $F_{i,\alpha}[j] = 1$ iff $C_{i,\alpha}[j] = 1$ and $\texttt{bin_rank}(1, j)$ on $C_{i,\alpha}$ is divisible by $\lg\sigma$). We construct a y-fast trie [19] over $F_{i,\alpha}$. This y-fast trie uses $O(l_{i,\alpha}/\lg\sigma \times \lg\sigma) = O(l_{i,\alpha})$ bits (as the trie is on universe $[\sigma]$, we use a word size of $O(\lg\sigma)$ for it). The σ y-fast tries corresponds to all the blocks in a given chunk thus occupies $O(\sigma)$ bits.

Using the bit vector B, we see that if we can compute $\texttt{bin_rank}(1, x)$ and $\texttt{bin_select}(1, r)$ for any given block, we can support `string_rank` and `string_select` operators in additional $O(1)$ time for $\alpha \in [\sigma]$ (See Section 2 of [9] for details).

To support rank/select on blocks, we first show how to support $\texttt{bin_rank}(1, j)$ on block $D = C_{i,C_i[j]}$ (i.e. to compute $\texttt{bin_rank}'(1, j)$ for a given block). With $F_{i,\alpha}$, we can compute $\lg\alpha\lfloor\texttt{bin_rank}_{C_{i,\alpha}}(1, j)/\lg\alpha\rfloor$ in $O(\lg\lg\sigma)$ time using the y-fast trie for $C_{i,\alpha}$, and we call it the *approximate rank* of j in $C_{i,\alpha}$. To support $\texttt{bin_rank}_D(1, j)$, we first compute $C_i[j]$ in $O(f(n, \sigma))$ time. Then we compute the approximate rank of j on D in $O(\lg\lg\sigma)$ time using the y-fast trie corresponding to D, and retrieve $R_i[j]$ in constant time. The sum of the above two is the result. Thus we can compute $\texttt{bin_rank}(1, j)$ on block D in $O(f(n, \sigma) + \lg\lg\sigma)$ time.

We construct a permutation π_i for each chunk C_i. To obtain the sequence of π_i, for $\alpha = 1, 2, ..., \sigma$, if α appears in C_i, we write down the positions (relative to the starting position of the chunk) of all its occurrences in increasing order. We use π_i^{-1} to denote its inverse. We see that $\pi_i^{-1}(k)$ is equal to the sum of the following two values: the number of characters smaller than $C_i[k]$ in C_i, and $\texttt{bin_rank}(1, k) + 1$ on block D. The first value can be easily computed using X_i in constant time, and we have already showed how to compute the second value in $O(f(n, \sigma) + \lg\lg\sigma)$ time. Therefore, we can compute any element of π_i^{-1} in $O(f(n, \sigma) + \lg\lg\sigma)$ time. For chunk C_i, using the auxiliary structure (P_i) that occupies $O(\lg\sigma/\lg\lg\lg\sigma)$ bits of space, we can compute any element of π_i in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time [14] (note that the $f(n, \sigma) + \lg\lg\sigma$ term in the above claim comes from the time required to retrieve a given element of π_i^{-1}).

Golynski *et al.* [9] showed how to compute `string_select` by a single access to π_i plus a few constant-time operations. When combined with our approach, we can support `string_select` for any label $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time. It was also shown in [9] how to compute `string_rank` by calling `string_select` $O(\lg\lg\sigma)$ times. Thus we can support operator `string_rank` in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time.

We finally calculate the space cost of our index. As there are n/σ chunks, the sum of space cost of the auxiliary structures constructed for all the chunks is clearly $O(n\lg\sigma/\lg\lg\sigma)$ bits. The overall space cost of all the auxiliary structures is therefore $n \cdot o(\lg\sigma)$. ☐

Lemma 4. *Using at most* $2n+o(n)$ *additional bits, the index of Lemma 3 also supports* `string_pred` *and* `string_succ` *in* $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ *time for label-pattern* α.

Proof. We only show how to support `string_pred`, and `string_succ` can be supported similarly. As $\texttt{string_pred}(\alpha, x) = \texttt{string_select}(\alpha, \texttt{string_rank}(\alpha, x))$ for $\alpha \in [\sigma]$, we only need to show how to support $\texttt{string_pred}(\alpha, x)$ when $\alpha \in [\bar{\sigma}]$.

We require another auxiliary structure. In the bit vector A, there are n 1s, so there are at most n runs of consecutive 1s. Assume that there are u runs and their lengths are $p_1, p_2, ..., p_u$, respectively. We store these lengths in unary using a bit vector D, i.e. $D = 1^{p_1}01^{p_2}0...1^{p_u}0$. The length of D is at most $2n$, and we store it using Part (a) of Lemma 1 in at most $2n + o(n)$ bits.

$$E=\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

```
COLUMNS = 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1
ROWS    = 3, 4,    1, 4,    3,    1, 2, 3,    4
```

Figure 1: An example of the encoding of a binary relation.

Now we show how to support $\texttt{string_pred}(\alpha, x)$ when $\alpha \in [\bar{\sigma}]$. Assume that c is the character such that $\alpha = \bar{c}$. We first retrieve $S[x-1]$ in $O(f(n,\sigma))$ time. If $S[x-1] \neq c$, then we return $x-1$. Otherwise, we need to compute the number of 1's before position $(c-1)\sigma + x - 1$ in A (this position in A corresponds to the $(x-1)^{\text{th}}$ position in the cth row in table E). We denote the result by j. As $j = \texttt{bin_rank}_B(1, \texttt{bin_select}_B(0, (c-1)n/\sigma)) + \texttt{string_rank}(c, x-1)$, we can compute j in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time. $v = \texttt{bin_select}_D(1, j+1)$ is the position in D that corresponds to the $(x-1)^{\text{th}}$ position in the cth row in table E. Thus $q = v - \texttt{bin_select}_D(0, \texttt{bin_rank}_D(0, v))$ is the number of consecutive 1s preceding position v in D (including the 1 at position v). If $q \geq x-1$, then there is no 0 in front of position $x-1$ in row c of table E, so we return $-\infty$. Otherwise, we return $x-q-1$ as the result. All the above operations take $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time. □

Combining Lemma 3 and Lemma 4, we have our first main result:

Theorem 1. *Given support for* `string_access` *in $O(f(n,\sigma))$ time on a string $S \in [\sigma]^n$, using a succinct index of $n \cdot o(\lg\sigma)$ bits, we can support* `string_rank`, `string_pred` *and* `string_succ` *for any label-pattern in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time, and* `string_select` *for any label-pattern $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time.*

3.2 Binary Relations

We define the interface of the ADT of a binary relation through the following operator: $\texttt{object_access}(x, i)$, which returns the ith label associated with x in lexicographic order, and returns $+\infty$ if no such label exists. We have the following result:

Theorem 2. *Given support for* `object_access` *in $f(n,\sigma,t)$ time on a binary relation on t pairs from an object set $[n]$ and a label set $[\sigma]$, using a succinct index of $t \cdot o(\lg\sigma)$ bits, we can support* `label_rank` *for any label-pattern and* `label_access` *for any label $\alpha \in [\sigma]$ in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ time, and* `label_select` *for any label-pattern $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ time.*

Proof. As with strings, we also conceptually treat a binary relation as an $n \times \sigma$ table E, and entry $E[\alpha][x] = 1$ iff object x is associated with label α. A binary relation on t pairs from $[n] \times [\sigma]$ can be stored as follows as in [2] (See Figure 1 for an example):

- a string `ROWS` of length t drawn from alphabet $[\sigma]$, such that the ith label of `ROWS` is the label of the ith pair in the column-major order traversal of E;
- a bit vector `COLUMNS` of length $n+t$ which encodes the number of labels associated with each object in unary.

To design a succinct index for binary relations, we explicitly store the bit vector `COLUMNS` using Part (a) of Lemma 1 in $n + t + o(n+t)$ bits. We now show how to support `string_access` on `ROWS` using `object_access`. To compute the ith character in `ROWS`, we need to compute the object it corresponds to (we call it x), and the rank of the label it corresponds to among all

the labels associated with x (we call it r). The position of the 0 in COLUMNS corresponds to the ith character in ROWS is $l = \texttt{bin_select}_{\texttt{COLUMNS}}(0, i)$, so $\alpha = \texttt{bin_rank}_{\texttt{COLUMNS}}(1, l) + 1$, and $r = l - \texttt{bin_select}_{\texttt{COLUMNS}}(1, \alpha - 1)$ if $\alpha > 1$ ($r = l$ otherwise). Thus we can support string_access by calling object_access once, with additional operations in constant time.

We store a succinct index for ROWS using Theorem 1 in $t \cdot o(\lg \sigma)$ bits. As we can support string_access on ROWS using object_access, the index can support string_rank and string_select on ROWS for any label $\alpha \in [\sigma]$. Using the approaches in [2] (Barbay *et al.* [2] showed how to support label_rank, label_select, and label_access operations on binary relations using rank / select on ROWS and COLUMNS), we can support the operators listed in this theorem. The running time of the algorithms can be easily computed from Theorem 1 and [2]. The space of the index is the sum of space cost of storing COLUMNS and the index for ROWS, which is $t \cdot o(\lg \sigma)$. □

3.3 Multi-Labeled Trees

We define the interface of the ADT of a multi-labeled tree to include the navigational operators defined in Section 2.3 and the following operator: node_label(x, i), which returns the ith label associated with node x in lexicographic order. Recall that we refer to nodes by their preorder numbers (i.e. node x is the xth node in the preorder traversal). In this ADT, node_label(x, i) supports operations on labels, while the other operators merely support navigational operations over the tree structure. One may argue that the ADT includes too many navigational operators to make the index applicable to various types of tree representation. However, this is not a major issue. First, for multi-labeled trees, researchers mainly concentrate on supporting more powerful queries on labels over the trees, rather than pure navigational operations, which makes it reasonable to assume to have a rich set of navigational operators available. Second, there are ways to represent ordinal trees and support the above navigational operations using $2n + o(n)$ bits [8].

We now present the definition of permutations on binary relations and a related lemma that we need to design succinct indexes for multi-labeled trees.

Definition 3. *Given a permutation π on $[n]$ and a binary relation $R \in [\sigma] \times [n]$, the* permuted binary relation *$\pi(R)$ is the relation such that $(x, \alpha) \in \pi(R)$ if and only if $(\pi^{-1}(x), \alpha) \in R$.*

Lemma 5. *Consider a permutation π on $[n]$, such that access to $\pi(i)$ and $\pi^{-1}(i)$ is supported in $O(1)$ time. Given a binary relation $R \in [\sigma] \times [n]$ of cardinality t, and support for* object_access *on R in $f(n, \sigma, t)$ time, using a succinct index of $t \cdot o(\lg \sigma)$ bits, we can support* label_rank *and* label_access *in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time, and* label_select *in $O(\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time, on* both *R and $\pi(R)$ for any label-pattern $\alpha \in [\sigma]$.*

Proof. In the proof of Theorem 2, we showed how to support string_access on ROWS using object_access on R and bin_rank / bin_select on COLUMNS. We then designed a succinct index for R using a succinct index for ROWS and the bit vector COLUMNS. We denote ROWS′ and COLUMNS′ to be the string and bit vector corresponding to $\pi(R)$, and store COLUMNS′ using using Part (a) of Lemma 1 using $n + t + o(n + t)$ bits. Thus to design a succinct index to support efficient retrieval for both R and $\pi(R)$, we only need to show how to support string_access on the string ROWS′.

To support string_access(i) on ROWS′, We first compute the object x corresponding to the ith element of ROWS′ using $x = \texttt{bin_rank}_{\texttt{COLUMNS}'}(1, i) + 1$. For $r = i - \texttt{bin_select}_{\texttt{COLUMNS}'}(1, x) - 1$, we see that the ith element of ROWS′ corresponds to the rth label of x in $\pi(R)$. As object x corresponds to object $y = \pi^{-1}(x)$ in ROWS, we have that $\texttt{string_access}_{\texttt{ROWS}'}(i) = \texttt{object_access}_R(y, r)$. Thus we can support $\texttt{string_access}_{\texttt{ROWS}'}(i)$ in $O(f(n, \sigma, t))$ time. □

It is not clear how to support the list of the children of a given node based on preorder traversal. But as seen in Figure 2 in Appendix A, the children of any given node are consecutive in the `DFUDS` order traversal of the tree. To make full use of the fact, we present a method to perform the conversions of the preorder and `DFUDS` order numbers of a given node (See Appendix A for the proof):

Lemma 6. *Using the* `DFUDS` *representation of an ordinal tree (Benoit* et al. *[3]) in* $2n + o(n)$ *bits, we can support the following operations in* $O(1)$ *time:*

- `find_dfuds`(i)*, which returns the rank in* `DFUDS` *order of the* i^{th} *node in preorder;*
- `find_pre`(i)*, which returns the rank in preorder of the* i^{th} *node in* `DFUDS` *order.*

To efficiently find *all* the α-ancestors of any given node, for each node and for each of its labels α we encode the number of α-ancestors of x. To measure the maximum number of such ancestors, we define the *recursivity* of a node, motivated by the notion of *document recursion level* of a given XML document [20].

Definition 4. *The* recursivity ρ_α of a label α *in a multi-labeled tree is the maximum number of occurrences of* α *on any rooted path of the tree. The* average recursivity ρ of a multi-labeled tree *is the average recursivity of the labels weighted by the number of nodes associated with each label* α *(denoted by* t_α*):* $\rho = \frac{1}{t}\sum_{\alpha\in[\sigma]}(t_\alpha \rho_\alpha)$.

Note that ρ is usually small in practice, especially for XML trees. Zhang *et al.* observed that in practice the document recursion level (when translated to our more precise definition, it is the maximum value of all ρ_α's minus one, which can be easily used to bound ρ) is often very small: in their data sets, it was never larger than 10 [20, Table 2]. With this definition, we have:

Theorem 3. *Consider a multi-labeled tree on* n *nodes and* σ *labels, associated in* t *relations, of average recursivity* ρ*. Given support for the navigational operations in* $O(1)$ *time, and* `node_label` *in* $O(f(n, \sigma, t))$ *time, using a succinct index of* $t(\lg \rho + o(\lg(\rho\sigma)))$ *bits, we can support (for a given node* x*) the enumeration of:*

- *the set of* α*-descendants of* x *(denoted by* D*) in* $O(|D| \lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ *time;*
- *the set of* α*-children of* x *(denoted by* C*) in* $O(|C| \lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ *time;*
- *the set of* α*-ancestors of* x *(denoted by* A*) in* $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma) + |A|(\lg\lg\rho_\alpha + \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma)))$ *time.*

Proof. (sketch) The sequence of nodes referred by their preorder numbers, and the sequence of nodes referred by their `DFUDS` numbers form two different binary relations with their associated label sets respectively. Lemma 6 provides constant-time conversions between these two sequences, and `node_label`(x, i) supports `object_access` on the binary relation between the set of nodes in preorder and the set of labels. By Lemma 5, we can construct an succinct index R for these two binary relations using $t \cdot o(\lg\sigma)$ bits, and support `label_rank`, `label_select` and `label_access` operations on either of them efficiently.

The support of the relations between nodes in preorder and the labels enables us to enumerate of all descendants of a node x matching label α in $O(|D| \lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ time using the techniques in Barbay *et al.* [2]. The same technique, used with the `DFUDS` order, enables us to enumerate all children of a node x matching α in $O(|C| \lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ time, as the `DFUDS` order traversal lists the children of any given node consecutively.

As there is no order in which the ancestors of each node are consecutive, we store for each label α of a node x the number of ancestors of x matching α. To be specific, for each label α such that $\rho_\alpha > 1$, we represent those numbers in one string $S_\alpha \in [\rho_\alpha]^{t_\alpha}$, where the ith number of S_α corresponds to the ith node labeled α in preorder. As the lengths of the strings $(S_\alpha)_{\alpha\in[\sigma]}$ are implicitly encoded in R, we just need to encode for each label α its recursivity ρ_α in unary, using at most $t+\sigma$ bits. We use the encoding of Golynski *et al.* [9] to encode each string S_α in $t_\alpha(\lg\rho_\alpha + o(\lg\rho_\alpha))$ bits to support `string_rank` and `string_access` in $O(\lg\lg\rho_\alpha)$ time and `string_select` in constant time. The total space used by these strings is $\sum_{\alpha\in[\sigma]} t_\alpha(\lg\rho_\alpha + o(\lg\rho_\alpha))$. By concavity of the lg function, this is at most $\left(\sum_{\alpha\in[\sigma]} t_\alpha\right)\left(\lg\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right) + o\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right)\right) = t(\lg\rho + o(\lg\rho))$.

To support the enumeration of all the α-ancestors of a node x, we first find from R the number of α-nodes preceding x in preorder (denoted by p_x) using `label_rank`. Then we initialize i to 1 and iterate as follows: find the position p_i in S_α of the character i immediately preceding position p_x: it corresponds to the ${p_i}^{\text{th}}$ α-node in preorder (this can be located using `label_select` on R). If this node is an ancestor of x (this can be checked using `depth` and `level_anc` in constant time), output it, increment i and iterate, otherwise stop. Each iteration contains a `label_select` on R and some rank and select operations on S_α, so each is performed in $O(\lg\lg\rho_\alpha + \lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time. Hence it takes $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma)+|A|(\lg\lg\rho_\alpha+\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma)))$ time to enumerate the set A. The overall space used is $t(\lg\rho + o(\lg(\rho\sigma)))$ bits. □

We can also support the retrieval of the first α-descendant (children, or ancestor) of node x that appears after node y in preorder:

Corollary 1. *Using an additional $2n+o(n)$ bits, we can also support (for any two given nodes x and y) the selection of:*

- *the first α-descendant of x after y in preorder in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time;*
- *the first α-children of x after y in preorder in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time;*
- *the first α-ancestor of x after y in preorder in $O(\lg\lg\rho_\alpha + \lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time.*

Proof. (sketch) Using the index in Theorem 3, we can easily support the first two operations. To support the search for the first α-ancestor of x after y, we encode the structure of the tree and support the `LCA` operation (i.e. computing the lowest common ancestor of two given nodes) in constant time using $2n+o(n)$ bits [17]. We assume that y precedes x in preorder (otherwise the operator return ∞), and that y is an ancestor of x (if not, the problem can be reduced to the search for the first α-ancestor of node x after node $\texttt{LCA}(x,y)$).

Using rank and select on the relation R and some navigational operators, we can find the first α-descendant z of y in preorder in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time. The node z is not necessarily an ancestor of x, but it has the same number of α-ancestors (we denote the number by i) as the node we are looking for. We can retrieve i from the string S_α in $O(\lg\lg\rho_\alpha)$ time. Finally, the first α-ancestor of x after y is the α-node corresponding to the value i just before the position corresponding to x in S_α, found in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma)+\lg\lg\rho_\alpha)$ time. □

Remark: The operations on multi-labeled trees are important for the support of XPath queries for XML trees [1, 2]. The main idea of our algorithms is to construct indexes for binary relations for different orders to traverse the trees. Note that without succinct indexes, we need to encode different binary relations separately and waste a lot of space.

4 Applications

4.1 High-Order Entropy-Compressed Succinct Encodings for Strings

Given a string S of length n over alphabet $[\sigma]$, we now design a high-order entropy-compressed succinct encoding for it that supports `string_access`, `string_rank`, and `string_select` efficiently. Theorem 2.2 in [9] claims that one can represent S in $nH_k + o(n \lg \sigma)$ bits and support the above operations. However, the claim was based on the assumption that by storing π^{-1} for all the chunks, the string S is stored explicitly. As the substring corresponding to each chunk is not π^{-1}, the above assumption is false. With our succinct indexes, we can now achieve such a goal.

Theorem 4. *Given a string S of length n over alphabet $[\sigma]$, we can represent it in $nH_k + o(n \lg \sigma)$ bits for any positive integer k such that $k + \lg \sigma = o(\lg n)$, and support* `string_access` *in $O(1)$ time. The representation also supports* `string_rank`, `string_pred` *and* `string_succ` *for any label-pattern in $O((\lg \lg \sigma)^2 \lg \lg \lg \sigma)$, and* `string_select` *for any label-pattern $\alpha \in [\sigma]$ $O(\lg \lg \sigma \lg \lg \lg \sigma))$ time.*

Proof. We use the approach presented by Grossi and Sadakane [18] to store S in $nH_k + O(n \lg n \lg \sigma / \lg \lg n)$ (H_k denotes the kth order entropy of S) for any positive integer k such that $k + \lg \sigma = o(\lg n)$. This representation allows us to retrieve $S[i]$ in $O(1)$ time (i.e. operator `string_access` can be supported in $O(1)$ time). We store a succinct index for S using Theorem 1, and the support for the above operations immediately follows. The overall space is $nH_k + O(n \lg \sigma \lg n / \lg \lg n) + O(n \lg \sigma / \lg \lg \lg \sigma)$. The last two terms sum up to $O(n \lg \sigma (\lg \lg n / \lg n + 1/ \lg \lg \lg \sigma)) = o(n \lg \sigma)$. [7] □

Remark: Using similar approaches, we can design succinct encodings for binary relations and multi-labeled trees based on our succinct indexes, and compress the underlying strings (recall that we reduce the operations on binary relations and multi-labeled trees to rank/select on strings and bit vectors) to high-order entropies of the strings.

4.2 High-Order Entropy-Compressed Text Indexes for Large Alphabets

Text indexes are data structures that facilitate text searching. Given a text string T of length n and a pattern string P of length m, whose symbols are drawn from the same fixed alphabet Σ, the goal is to look for the occurrences of P in T. We consider three types of queries: existential queries, cardinality queries, and listing queries. An *existential query* returns a boolean value that indicates whether P is contained in T. A *cardinality query* returns the number of occurrences of P in T (*occ* denotes the result). A *listing query* lists all the positions of occurrences of P in T.

We now apply our index to design space-efficient suffix arrays. We first present the following lemma to encode strings in 0th order entropy while supporting rank and select:

Lemma 7. *Given a string S of length n over alphabet $[\sigma]$, we can represent it in $n(H_0 + o(\lg \sigma))$ bits to support* `string_access` *and* `string_rank` *for any label-pattern α in $O(\lg \lg \sigma)$ time, and* `string_select` *for any label-pattern $\alpha \in [\sigma]$ in $O(1)$ time.*

For the proof of Lemma 7, see Appendix B. It is known that we can represent suffix arrays by encoding the Burrows-Wheeler transformed string of the raw text (denoted by T^{bwt}) appropriately [11] [7]. Ferragina*et al.* [7] also presented how to design high-order entropy-compressed suffix array given an encoding of T^{bwl} that occupies space in 0th order entropy plus an appropriate lower order term. When combine these results with Lemma 7, the following theorem immediately follows:

[7] If $\sigma < \sqrt{\lg n}/2$, we can support all the operations in constant using table lookups. When $\sigma \geq \sqrt{\lg n}/2$, we can bound n in terms of σ, and hence this term is $o(n \lg \sigma)$.

Theorem 5. *A text string T of length n over alphabet $[\sigma]$ can be stored using $nH_k + o(n \lg \sigma))$ bits (for any $k \leq \beta \lg_\sigma n$ and $0 < \beta < 1$), where H_k denotes the kth order entropy of T. Using this, given a pattern P of length m, we can answer existential and cardinality queries in $O(m \lg \lg \sigma)$ time, list each occurrence in $O(\lg^{1+\epsilon} n \lg \lg \sigma)$ time for any ϵ where $0 < \epsilon < 1$, and output a substring of length l in $O((l + \lg n) \lg \lg \sigma)$ time.*

Remark: Grossi *et al.*[10] designed the first text index that uses $nH_k + o(n) \lg \sigma$ bits, and support existential and cardinality queries in $O(m \lg \sigma + \texttt{polylog}(n))$ time. However, the $\lg \sigma$ factor is not good for texts over large alphabets. Golynski *et al.* [9] reduced this factor to a $\lg \lg \sigma$, but their index is not compressible. Our text index has the advantages of both these indexes.

5 Concluding Remarks

In this paper, we define succinct indexes for the design of data structures. We show their advantages (listed in Section 1) by presenting succinct indexes for strings, binary relations and multi-labeled trees, and applying them to various applications. We expect that the concept of succinct indexes will influence the design of succinct data structures.

References

[1] Jérémy Barbay. Adaptive search algorithm for patterns, in succinctly encoded XML. Technical Report CS-2006-11, University of Waterloo, Ontario, Canada, 2006.

[2] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 24–35. Springer-Verlag LNCS 4009, 2006.

[3] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 169–180. Springer-Verlag LNCS 1663, 1999.

[4] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[5] Erik D. Demaine and Alejandro Lopez-Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 289–294, 2001.

[6] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[7] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*, pages 150–160. Springer-Verlag LNCS 3246, 2004.

[8] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 2004.

[9] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete algorithm*, pages 368–373, 2006.

[10] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 841–850, 2003.

[11] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete algorithm*, pages 23–32, 2005.

[12] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[13] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 11–12, 2005.

[14] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 345–356, 2003.

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

[16] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 233–242, 2002.

[17] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 225–232, 2002.

[18] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm*, pages 1230–1239, 2006.

[19] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[20] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proceedings of the 22nd International Conference on*

Data Engineering, pages 61–72, 2006.

Appendix A Proof of Lemma 6

In this proof, we use the operators defined in [3].

In the balanced parentheses representation of the tree in [3], each node corresponds to an opening parenthesis and a closing parenthesis. We observe that in the sequence, the opening parentheses correspond to DFUDS order, while the closing parentheses correspond to the preorder. For example, in Figure [3], the 6th node in DFUDS order (which is the 5th node in preorder) corresponds to the 6th opening parenthesis, and the 5th closing parenthesis.

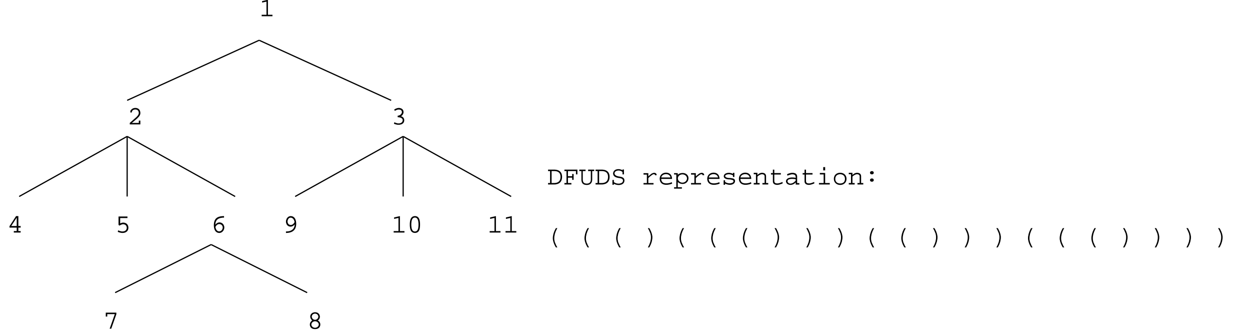

Figure 2: An ordinal tree (where each node is assigned its rank in the DFUDS order) and its DFUDS representation in [3].

With this observation, $\texttt{find_dfuds}(i)$ means that for the node that corresponds to the ith closing parenthesis (denoted by x), we need to compute the rank of the corresponding opening parenthesis among opening parentheses. To compute this value, we first find the opening parenthesis that matches the closing parenthesis that comes before node x. Its position in the sequence is: $j = \texttt{find_open}(\texttt{select}_{\texttt{close}}(i-1))$. With j, we can compute the position of the parent of x, which is $p = \texttt{select}_{\texttt{close}}(\texttt{rank}_{\texttt{close}}(j)) + 1$, and $\texttt{child_rank}(x)$ (denoted by r), which is $r = \texttt{select}_{\texttt{close}}(\texttt{rank}_{\texttt{close}}(p) + 1) - j$. Finally, $\texttt{rank}_{\texttt{open}}(p + r - 1)$ is the result.

The computation of $\texttt{find_pre}(i)$ is exactly the inverse of the above process. $\square$

Appendix B Proof of Lemma 7

We divide string S and its corresponding conceptual table E into chunks and blocks, and store bit vectors B and X_i as in the proof of Theorem 1. We also store the same set of y-fast tries for chunks.

Each row of E is a bit vector, and we denote the αth row by $E[\alpha]$ for $\alpha \in [\sigma]$. For each $\alpha \in [\sigma]$, we store $E[\alpha]$ using Lemma 2 in $\lg \binom{n}{n_\alpha} + o(n_\alpha) + O(\lg \lg n) \approx n_\alpha \lg \frac{en}{n_\alpha} + o(n_\alpha) + O(\lg \lg n)$ bits. Summing the space cost of these dictionaries, the last two terms clearly sum to $n \cdot o(\lg \sigma)$. The first term sums to $nH_0 + n \lg e$. Therefore, the total space cost is $n(H_0 + o(\lg \sigma))$ bits.

With the rows of E stored in the above way, $\texttt{string_select}$ can be supported in $O(1)$ time, as $\texttt{string_select}(\alpha, i) = \texttt{bin_select}_{E[\alpha]}(1, i)$, where $\alpha \in [\sigma]$. With the support for $\texttt{string_select}$ on S, we can easily support $\texttt{string_select}$ on any chunk in $O(1)$ time. Using the method in [9], we can further support $\texttt{string_rank}$ on any chunk in $O(\lg \lg \sigma)$ time. Similarly to the proof of Theorem 1, we can support $\texttt{string_rank}$ on S in $O(\lg \lg \sigma)$ time.

Now we need to provide support for $\texttt{string_access}$. We first design data structures to support π_i and π_i^{-1} (see the proof of Theorem 1 for the definition of π_i and π_i^{-1}). From the definition of π_i, we have that $\pi_i(j) = \texttt{bin_select}_{C_{i,\alpha}}(1, r)$, where $\alpha = \texttt{bin_rank}_{X_i}(0, \texttt{bin_select}_{X_i}(1, j))$, and

$r = \texttt{bin_select}_{X_i}(1, j) - \texttt{bin_select}_{X_i}(0, \alpha)$. α and r can clearly be computed in $O(1)$ time, and because $\texttt{bin_select}_{C_{i,\alpha}}(1, r) = \texttt{bin_select}_{E[\alpha]}(1, r + z)$, where z is the number of 1s in the αth row of E before the $(i-1)\lg\sigma^{\text{th}}$ position in the row (we can compute z by performing rank / select operations on B in constant time), we can compute $\pi_i(j)$ in O(1) time. For chunk C_i, by adding an auxiliary structure that occupies $O(\lg\sigma/\lg\lg\sigma)$ space, we can compute any element of π_i^{-1} in $O(\lg\lg\sigma)$ time [14]. Finally, we use the method in [9] to compute $C_i[j]$ in $O(\lg\lg\sigma)$ time, which in turn can be used to support `string_access` in $O(\lg\lg\sigma)$ time as $\texttt{string_access}[l] = C_v[l - v\sigma]$ where $v = \lfloor l/\sigma \rfloor$.

Similar to the proof of Theorem 1, we observe that the above auxiliary data structures (B, X_i's, y-fast tries, and auxiliary structures for π_i's) occupy $n \cdot o(\lg\sigma)$ bits, while $E[i]$'s occupies $n(H_0 + o(\lg\sigma))$ bits. Therefore, the overall space cost is $n(H_0 + o(\lg\sigma))$ bits. □