MASTER THESIS

Adam Juraszek

Succinct encodings of trees

Department of Applied Mathematics

Supervisor of the master thesis: Mgr. Martin Mareš, Ph.D.
Study programme: Master of Computer Science
Study branch: Theoretical Computer Science

Prague 2016
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

Prague, July 22, 2016

signature of the author
Title: Succinct encodings of trees

Author: Adam Juraszek

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: We focus on space-efficient, namely succinct, representations of static ordinal unlabeled trees. These structures have space complexity which is optimal up to a lower-order term, yet they support a reasonable set of operations in constant time. This topic has been studied in the last 27 years by numerous authors who came with several distinct solutions to this problem. It is not only of an academic interest, the succinct tree data structures has been used in several data-intensive applications, such as XML processing and representation of suffix trees. In this thesis, we describe the current state of knowledge in this area, compare the many different approaches, and propose several either new or alternative algorithms for operations in the representations alongside.

Keywords: trees, succinct data structures, balanced parentheses, tree covering
I dedicate this thesis to my parents, who kept me motivated throughout the time of writing and provided me with moral support when I needed it most. I also want to mention my girlfriend, who waited until I finished the daily portion of work and kept me company on late nights. I would like to express my thanks to my advisor, Marin Mareš, for bringing this topic to my attention, and for his valuable insight during our meetings.
# Contents

## Introduction
- Organization of This Thesis .................................................. 4

## 1 Foundations
- 1.1 Model of Computation ......................................................... 5
  - 1.1.1 RAM for Bit Strings .............................................. 6
- 1.2 Space-Efficient Data Structures .......................................... 7
- 1.3 Storage .............................................................................. 8
- 1.4 Basic Techniques .............................................................. 9
  - 1.4.1 Division and Pointers ................................................. 9
  - 1.4.2 Precomputation .......................................................... 9
  - 1.4.3 Index Without Data .................................................... 10

## 2 Bit String Data Structures .................................................... 11
- 2.1 Operations on Bit Strings .................................................... 11
  - 2.1.1 \texttt{rank} ................................................................ 12
  - 2.1.2 \texttt{select} ............................................................. 13
- 2.2 Balanced Bit Strings ........................................................... 16
  - 2.2.1 Structure for \texttt{match} and \texttt{enclose} ..................... 18
  - 2.2.2 Block Queries ......................................................... 19
  - 2.2.3 Index for Range Minimum Query ................................ 23
- 2.3 Dictionaries ...................................................................... 27
  - 2.3.1 Sublinear Fully Indexed Dictionary ................................ 28
  - 2.3.2 Compressed Array ...................................................... 29
  - 2.3.3 Tiny Compressed Array ............................................... 30

## 3 Representations of Trees ....................................................... 32
- 3.1 Operations ....................................................................... 34
- 3.2 Representations ................................................................. 34
- 3.3 Level-Order Unary Degree Sequence ..................................... 35
  - 3.3.1 Navigation Operations ............................................... 36
  - 3.3.2 Depth and Level Queries .......................................... 36
  - 3.3.3 Leaf Operation .......................................................... 38
  - 3.3.4 Final Thoughts .......................................................... 39
- 3.4 Balanced Parentheses .......................................................... 39
  - 3.4.1 Navigation Operations ............................................... 40
  - 3.4.2 Other Native Operations .......................................... 41
  - 3.4.3 In-Order Rank and Select and Leaf Operations .......... 42
  - 3.4.4 Final Thoughts .......................................................... 43
- 3.5 Depth-First Unary Degree Sequence ...................................... 44
  - 3.5.1 Navigation Operations ............................................... 45
  - 3.5.2 Other Native Operations .......................................... 46
  - 3.5.3 Lowest Common Ancestor ....................................... 47
  - 3.5.4 Leaf operations .......................................................... 47
  - 3.5.5 Final Thoughts .......................................................... 48
Introduction

Tree data structures are one of the most commonly used in computer science, and as such they have been studied for decades. Many of the data structures are designed for a general purpose – they support most operations which their user may need. However, if more information is known about the tree and the operations which are required, then it is possible to come up with a specialized tree data structure which is better for the particular use-case than any general purpose one.

We will be focusing on one type of trees: ordinal unlabeled trees. Ordinal means that children of a vertex \( v \) are ordered and assigned numbers, to which we refer as child rank, starting with 1 which is assigned to the leftmost child, up to \( \text{degree}(v) \) which gets the rightmost child. Unlabeled in our case means that all vertices are treated equally by all operations.

Moreover we will focus on static data structures that support all operations in constant time and that are succinct – their space complexity is close to the optimal one. We will define what the restriction on space complexity means in the first chapter.

This problem of space-efficient data structures for representing trees was first studied by \[\text{Jac89}\]. Although their data structure was very limited, it brought attention to the topic, and lead to discovering three other distinct approaches. The hard part about succinct representations comes with adding support for operations which are not native for it. Many authors competed by introducing succinct indices with which they augmented their representation in order to support operations of the other representations. Quite recently (\[\text{FM08}, \text{SN10}\]) two data structures were proposed which can answer almost every operation.

Succinct representations of trees have found an applications in the following two areas: XML processing (\[\text{GRR06}, \text{LY08}, \text{DRR08}\]), and suffix trees (\[\text{JSS12}, \text{MRR01}\]). Both applications handle volumes of data for which the traditional data structures are inadequate. In the latter case, it not unusual to represent a single human genome, which on its own requires about 750 MB. Building a traditional suffix tree which uses multiple pointers per vertex increases the memory needed manyfold.

We continue in the saga of succinct representations of trees. We investigate and describe the main data structures which are dominant in this area. Not only do we summarize the current state, we also come up with own results which push the frontier even further. We propose two operations which have not been supported before by their respective representations. We also offer alternative implementations or small improvements of others.
Organization of This Thesis

In the first chapter we define the environment in which we present all results in the thesis. This includes the definition of the computation model which we use when we formulate algorithms, types of space-efficient data structures, and general techniques which we use in the following chapters.

The second chapter is dedicated to simpler data structures than trees. We present results about supporting various operations on general and balanced bit strings. Since bit strings can be seen as characteristic vectors of a subset of an interval $[0, N)$, we continue with results about representing sets and supporting the same operations on them. We also define a structure called compressed array which we use extensively in the last part of the thesis.

In the third chapter, we define a set of operations on trees which we consider later when introduce various tree data structures. We present three representations of trees which are based on a single bit string encoding the whole tree. We focus on operations which can be supported using the structures developed in the second chapter.

In the last chapter, we extend one of the representations described earlier into a data structure which supports nearly all possible operations. We also present the only data structure which decomposes the tree into subtrees instead of reducing it to a simple bit string. The last result in this thesis connects three representations into a single data structure, and thereby combines their advantages.

We conclude with a table of operations which each of the representations supports, and a summary of the contribution of our work.
1. Foundations

1.1 Model of Computation

We state all our results for word-RAM, a version of Random Access Machine, which resembles current computers. A word is a sequence of \( w \) consecutive bits which can be accessed and processed in one step of the computation. The constant \( w \) depends on the space complexity \( S \) in words of the problem which is being solved. A lower bound on \( w \geq \lceil \log S \rceil \) is necessary in order to allow access to any piece of memory which is used during the computation. We usually assume that \( w = \Theta(\log S) \) as working with larger words would make the model too powerful.

There are many definitions of word-RAMs, therefore we briefly describe the one which we are going to use.

RAM has access to two types of memory:

- **Main memory** which forms a sequence of words, which are the smallest addressable units. The sequence is potentially unbounded, however only a limited number of words can be accessed, depending on the word size.
  
  We assume (as in case of Turing machine), that the input is written in the main memory starting at position 0. The input can be encoded in various ways, for example prefixed with the number of words it requires. Except for the input, the memory is filled with zeros when the program starts.

- **Constant number of registers**; each of them stores a single word.

The instruction set of our model is inspired by the assembly language of current computers and the C language [KR06]. All single letter arguments can be either names of the registers or constants of size \( w \) bits.

**Flow control**

Instructions which are used for determining which instruction to process next.

- **Termination**: **halt**
  
  Stops the program immediately.

- **Jumps**: **label**: instruction, **goto** label, **goto** label if \( a \)
  
  Continues with execution of the instruction at the specified label. In the latter case, it goes to the label only if condition \( a \) is satisfied (true), otherwise it continues with the next instruction.

- **Functions**: **function** \( fn(args...) \), **return** \( a \)
  
  Defines a function \( fn \) with a constant number of arguments, possibly zero. A function has access to the main memory but not to any registers which were used by its caller. Each function ends with a return

\[1\] All logarithms are binary. We will usually assume the ceiling function of the logarithm while not explicitly writing it. This does not change the asymptotic time nor space complexity of the algorithms as it is only a lower order term.
instruction followed by a value. After a return, the program continues
with an instruction which follows the one which initiated the call.

**Memory access**

Instructions for accessing and modifying the main memory.

**Read:** \(\text{register} \leftarrow \text{Memory}[\text{position}]\)

Reads a word from the main memory and stores it in a named \text{register}.

**Write:** \(\text{Memory}[\text{position}] \leftarrow a\)

Overwrites the word stored in the main memory at the given \text{position} with the value \(a\).

**Value manipulation**

Instructions on values which are stored registers; the resulting value is always stored to a register: \(\text{register} \leftarrow \text{operation}\).

**Arithmetic:** \(a + b, a - b, a \cdot b, a / b, a \% b\)

Arithmetic operations interpret the values \(a\) and \(b\), and their result as numbers encoded as two’s complement. Only the least significant \(w\) bits of the result are used. The division is integral. \(\%\) denotes the operation remainder after division of \(a\) by \(b\).

**Bitwise:** \(a \& b, a \mid b, a \wedge b, \sim a, a \ll b, a \gg b\)

The last two instructions represent shift of the binary value \(a\) to left/right by \(b\) positions; the result is then padded by zeros.

**Boolean:** \(a \text{ and } b, a \text{ or } b, \text{not } a\)

Booleans are converted to integers using Iverson’s bracket convention:

\[
[a] = \begin{cases} 
1 & \text{if } a \text{ is true;} \\
0 & \text{otherwise.}
\end{cases}
\]

**Relational:** \(a < b, a \leq b, a > b, a \geq b, a = b, a \neq b\)

The result is a boolean value.

**Function call:** \(fn(args \ldots)\)

Calls a function \(fn\), provides it required number of arguments. The result is the value returned by the function.

### 1.1.1 RAM for Bit Strings

The general word-RAM is too restrictive for us as we will need to address individual bits or ranges of bits which are not aligned with words in the main memory.

We redefine the operations Read and Write to reflect our needs.

**Read:** \(\text{register} \leftarrow \text{Memory}[\text{start : end}]\)

Reads the \(end - start \leq w\) bits from the main memory starting with the bit at position \(start\), which is in \((start \% w)\)-th word and stores them padded with zeros from left in a \text{register}.

**Write:** \(\text{Memory}[\text{start : end}] \leftarrow a\)

Overwrites the \(end - start \leq w\) bits stored in the main memory at the given starting position with the \(end - start\) least significant bits of the value \(a\).
We also redefine $\text{Memory}[\text{start}]$ to be equal to $\text{Memory}[\text{start} : \text{start}+1]$. These redefined instructions can be emulated with a constant number of arithmetic, bitwise, and up to two original memory accessing or modifying instructions.

We define the precise way how the input is stored in the main memory. The first word (at position 0) contains the total number of bits – that can require up to $\log wS = \Theta(\log S)$ bits. Starting with the position 1 the bits of the input are stored consecutively filling the words from the most significant to the least significant bits. For convenience of querying ranges past the end of the input, we keep the one word after the end of the input free.

We prefer to express the space complexities in bits rather than words and except for the very last part we will not use words at all. All values that our algorithms will store in registers will fit into them.

### 1.2 Space-Efficient Data Structures

In a universe $U$, every object can be represented as a sequence of $\lceil \log |U| \rceil = N$ bits simply by ordering all objects in the universe and encoding their position within the order in binary. Although this encoding is optimal in terms of the space used, it is rarely useful for more advanced applications than iterations over the objects in the universe and reconstruction of the original object. Many operations cannot be supported unless the object is fully decoded into a more traditional form first; others can be processed on the encoded object, however their time complexity is much worse than optimal.

The goal of space-efficient data structures is to allow some additional space in exchange for more and faster operations. There are three major representatives depending on their space complexity. In this thesis, we are only interested in succinct data structures.

**Implicit data structure**

An *implicit data structure* for a universe $U$ is a data structure which has space complexity $\log |U| + O(\log \log |U|)$ bits of space; sometimes only $O(1)$ extra bits is allowed. The encoding of the input is important in case of the tighter definition as it does not allow storing the size of the input itself. An example of an implicit structure is a C string of characters which is terminated by a zero byte.

Another example are general directed labeled graphs on $n$ vertices which can be stored in $n(n - 1)$ bits – for pair of vertices $i \neq j$ a bit is stored whether the edge $(i,j)$ is in the graph or not – this is basically adjacency matrix without its diagonal. The only supported operation in constant time is determining whether two vertices are connected by an edge.

Only few implicit structures are known, since the restriction posed by the definition of an implicit data structure is often too strict.

**Succinct data structures**

A *succinct data structure* is a data structure whose space complexity is $\log |U| + o(\log |U|)$ bits. There exist succinct data structures for encoding some restricted classes of graphs, such as planar graphs, and trees. The rest of this thesis is about succinct structures.
Extending our example of a general graph by an operation for querying out-degrees of vertices results in a succinct data structure. We simply add an array of precomputed out-degrees which requires \( n \log n = o(n^2) \) bits.

Moreover, there is a succinct structure which supports adjacency testing and iteration over the list of neighbors and more in constant time (\cite{RRS07}, Theorem 6.1).

**Compact data structure**

A *compact data structure* is a data structure which uses \( O(\log |U|) \) bits.

A compact structure is often a byproduct of a succinct structure when the constants in space or time complexity prove to be too big for a real use. An example of such compact structure is \cite{GGMN05}.

We can easily extend the succinct structure for directed graphs into a compact structure for undirected graphs by simply using it twice: for the original graph and the graph with reversed edges.

## 1.3 Storage

There are two distinct approaches to representing data structures as bit strings depending on how the data are partitioned.

The *systematic approach* splits the encoding into two parts:

- the data sufficient for reconstruction of the structure. This could be an implicit structure, however a more verbose encoding is often used. For example in case of trees, it can be made by a traversal or a decomposition of the tree. It can often answer some local queries.
- auxiliary indices which help to speed up or even realize more advanced operations on the data structure. Indices store additional information which is intended to offer answers to queries which are non-local, such as depth of a vertex, or the lowest common ancestor.

The size of the indices is dominated by the size of the data. We call an *index succinct* if its size is \( o(N) \); recall that we defined \( \lceil \log |U| \rceil = N \).

Sometimes it is easier to think about *density* of an index (or its part) in bits per bit of data. The density of a succinct index is \( o(1) \).

In systematic approach, it is easy to add another index and thereby support new operations. It is also possible to drop indices which are not necessary for particular use-cases in order to make the encoded data structure a little smaller. It can sometimes be beneficial to save only the data while recomputing all of the indices when the data structure is loaded into memory.

The *non-systematic encoding* cannot be split into such two parts. The organization of data itself allows all kinds of operations to be supported in desired time.

Non-systematic representations are designed with the set of supported operations in one’s mind. It is not possible to remove support for any operation in order to spare some space. Adding support of a new operation which was not planned beforehand requires its composition from the already supported operations. It is also possible to add an additional index to a non-systematic data-structure, which results in a hybrid structure.
1.4 Basic Techniques

There are a few techniques which are used in nearly every data structure. The idea of most of them is simple, however when combined together they pose a strong tool.

1.4.1 Division and Pointers

Every data structure is only a string of bits and unless we can calculate offset in any other way (as in case of the implicit graphs \[1.2\]), we need to divide the bit string into parts and use pointers to navigate between them. In the following lemma we show that a small number of independent parts can be concatenated in one bit string.

**Lemma 1.** Let \( K = o\left(\frac{N}{\log N}\right) \) be the number of bit strings \( S_1, S_2, \ldots, S_K \) and \( N_1, N_2, \ldots, N_K \) their lengths which sum up to \( N \). Then we can combine all bit strings into a single bit string \( S \) while maintaining succinctness and constant-time access to the individual bit strings. Typically \( K = O(1) \).

**Proof.** The new bit string \( S \) starts with the number of parts in binary followed by a header in form of a table and continues with concatenated bit strings \( S_1, \ldots, S_K \) without any delimiters. The table contains \( K \) offsets of all the original bit strings within \( S \) ignoring the space taken by the header. Each offset is encoded as a binary number using \( \lceil \log(N) \rceil \) bits, which results in the table having size \( \log K + K \lceil \log(N) \rceil = o(N) \).

This lemma is used whenever we add indices to a systematic succinct data structure, in such cases we use it as a fact and do not emphasize it.

We store an offset – which we also call a pointer – relative to a certain position in the bit string rather than absolutely from its beginning. The size of the offset in bits depends on the range into which it points. The fact that the pointers can have different sizes allows us to decompose a big structure into small parts with restricted pointers between them. There will be a lot of pointers within individual parts, however they require only a small size, and pointers between parts, which are large in size but used only seldom.

1.4.2 Precomputation

The goal of most succinct data structures is to perform operations in constant time. Even if this was possible on high level, in the end the operation must be processed on data of size close to a machine word. The following lemma shows that we can precompute all queries of an operation which are restricted to a small block of data.

**Lemma 2.** Let \( O \) be an operation with some parameters which take \( P \) bits and returns \( R \) bits, such that its access to the memory is restricted to \( B \) consecutive bits which are the same for all values of the parameters. The operations \( O \) can be then solved in \( O(1) \) time with an index of additional \( 2^{B+P} R \) bits.

As long as \( B + P + \log R < \alpha \log N \) for \( 0 < \alpha < 1 \), this results in a succinct index.
Proof. We represent this as a table of consecutively stored precomputed answers to $2^{b+p}$ possible queries. The index of the table is the $B$ bits of data concatenated with $P$ bits of parameters, interpreted as a number.

Lookup is easy: $B$ and $P$ bits are concatenated and multiplied by $R$, which gives the offset of the result.

The bound for a succinct index follows from the total size of the table.

We can say that a look-up table captures the result of an algorithm with parameters of size $P$ accessing $B$ consecutive bits of the main memory and returning a result of $R$ bits.

We call the structure from the lemma a look-up table. All our look-up tables in this thesis will be succinct indices.

The look-up tables are sometimes used merely as a way to overcome the differences between the word-RAM and the real computers as there are instructions in modern computers which cannot be computed directly on the word-RAM.

If the algorithm captured by the look-up table can be split into several parts, then a single large look-up table can be replaced with two or more look-ups in substantially smaller tables. This can result in an implementation which is more considerate to caches in modern CPUs in exchange for a theoretical constant-time slowdown.

Example. We want to simulate an instruction POPCNT which returns the number of bits set to one. This instruction is provided by CPUs manufactured by Intel (and others) and it is described in the manual [Int09].

We use a look-up table which returns the number of set bits in a sequence of $b = \frac{\log n}{2}$ consecutive bits. The input of the operation (a word) is split into a constant number of blocks of length $b$ bits. The overall algorithm then iterates over blocks in the input and sums the answers of queries on the look-up table.

Note that setting $b = \frac{\log n}{4}$ extends the running time by factor of two, but at the same time lowers the space complexity from $\sqrt{n} \log \log n$ to $\frac{\sqrt{n}}{\sqrt{\log n}}$.

1.4.3 Index Without Data

Let’s assume that we want to support an operation on derived bit string $T$, however we cannot afford to store $T$. We show on an example how that it possible.

Example. Instead of counting ones, as in the previous example, we want to count the transitions from zero to one in the bit string $S$. We also assume that we do not want to store another look-up table, which would also be a solution.

We can define a new bit string $T$ which has a bit set for each such transition:

$$T = \sim S \& (S \ll 1).$$

Then the number of transitions in $S$ is equal to the number of set bits in $T$. Whenever the operation counting transitions needs to access the bit string $T$, it can only request $w$ bits at a time, which we can compute on the fly with a constant slowdown from the bit string $S$ using the given formula.

This technique is indeed general; any algorithm on word-RAM can access at most $w$ bits of the input in each step. We do not need to know how the operation works as long as we are able to provide it any chunk of $w$ bits of memory in constant time upon request.
2. Bit String Data Structures

In this chapter we present succinct data structures and succinct indices which are related to bit strings. We start with definitions of \texttt{rank} and \texttt{select} operations on a general bit string and show the succinct indexes for them.

Then we assume a sequence of parentheses which are correctly matched and naturally encoded as bit strings, which we call balanced. We define the operations \texttt{match} and \texttt{enclose}. Finally we define the problem of \textit{range minimum queries} and present its solution for (balanced) bit strings.

We leave the realm of bit strings and focus on dictionaries – a data structure for storing sets with a given number of elements. There are various types of dictionaries which differ by the operations which they support. We show a simple implementation of a fully indexable dictionary which we use in the introduction of a compressed array.

2.1 Operations on Bit Strings

We define the following two operations for a general bit string of size \(N\) bits. We use \(*\) as a placeholder for a value of a bit – either 0 or 1 (and later opening or closing parenthesis).

\[ \text{\texttt{rank}}_*(S, i) \rightarrow r \]

Returns the number \(r\) of \(*\) symbols in the bit string \(S\) on positions \([0, i]\).

For convenience we extend the definition of \texttt{rank} as 0 for \(i < 0\) and \(\text{\texttt{rank}}_*(S, N - 1)\) for \(i \geq N\).

\[ \text{\texttt{select}}_*(S, r) \rightarrow j \]

Returns the position \(j\) of the \(r\)-th symbol \(*\) in the bit string \(S\). If \(r \leq 0\), it returns \(-1\); if \(r > \text{\texttt{rank}}_*(S, N - 1)\), it returns \(N\).

Using these two operations we are able to derive more operations related to a given position:

\[ \text{\texttt{inspect}}(S, i) = S[i] \rightarrow \{0, 1\} \]

Returns the symbol at position \(i\) in the bit string \(S\). If for some reason the bit string \(S\) was not accessible, the operation can be implemented using RANKS:

\[ \text{\texttt{inspect}}(S, i) = \text{\texttt{rank}}_1(i) - \text{\texttt{rank}}_1(i - 1). \]

\[ \text{\texttt{pred}}_*(S, i) \rightarrow j \]

Returns the rightmost position \(j \leq l\) of the symbol \(*\):

\[ \text{\texttt{pred}}_*(S, i) = \text{\texttt{select}}_*(S, \text{\texttt{rank}}_*(S, i)). \]

\[ \text{\texttt{prev}}_*(S, i) \rightarrow j \]

Returns the rightmost position \(j < i\) of the symbol \(*\):

\[ \text{\texttt{prev}}_*(S, i) = \text{\texttt{select}}_*(S, \text{\texttt{rank}}_*(S, i - 1)). \]
\( \text{succ*}(S, i) \rightarrow j \)

Returns the leftmost position \( j \geq i \) of the symbol *:

\[
\text{succ*}(S, i) = \begin{cases} i & \text{if } S[i] = \ast; \\ \text{select*}(S, \text{rank*}(S, i) + 1) & \text{otherwise.} \end{cases}
\]

\( \text{next*}(S, i) \rightarrow j \)

Returns the leftmost position \( j > i \) of the symbol *:

\[
\text{next*}(S, i) = \text{select*}(S, \text{rank*}(S, i) + 1).
\]

Unless it is ambiguous, the first argument specifying the bit string on which the operation is performed can be omitted.

As all other operations can be derived from \text{RANK} and \text{SELECT}, it is sufficient to show succinct indices for these two. Indices for both operations are well studied. Many improvements were proposed in order to lower the theoretical space complexity, spare memory in the practical cases, speed up the real implementation. For more details we refer to [GGMN05, KNKP05, MN07].

2.1.1 RANK

We start our description \text{RANK} which is conceptually easier. We show the data in the form in which it was originally described by [Jac88].

It is sufficient to support only \text{RANK}1 because of the following identity:

\[
i = \text{rank}0(i) + \text{rank}1(i) - 1.
\]

We cover the bit string \( S \) by blocks and small blocks. An \( i \)-th block of size \( B \) starts at positions \( iB \) and ends at position \((i + 1)B - 1\). The small blocks are defined the same way, except they use the size \( b \). The sizes will be determined later.

First we design the small blocks so that we can precompute a look-up table \text{RANK} answering rank queries on them. The query in the look-up table has one parameter: the small-block-local offset \( o = i \% b \), and it returns the number of symbols * until \( o \). This is a very similar problem to the one which we showed in the example in section 1.4.2.

The data supplied to the look-up table are the \( b \) bits of the small block and the parameter \( o \); the answer is in range \([0, b]\). The algorithm captured by the look-up table simply iterates through the bits of the small block and adds them together. We set \( b = \frac{\log N}{2} \). The look-up table is a succinct index since \( b + \log b + \log(b + 1) < \log N \); the total size is \( O(\sqrt{N} \log N \log \log N) = o(N) \) bits.

The blocks represent parts of the bit string of size \( B \), which is assumed to be a multiple of \( b \). Each block \( k \) has an array \( \text{block}_k \) which for each small block \( l \) contains the numbers of * in the block before the beginning of \( l \):

\[
\text{block}_k[l] = \sum_{x=KB}^{kB+b-1} [S[x] = \ast].
\]
The sizes of all the block arrays sum up to $\frac{N}{B^b} B \log(B + 1)$. When $B$ is set to $\log N$, the size is asymptotically $O \left( \frac{N}{\log N} \log \log N \right) = o(N)$. As the arrays $\text{block}_k$ have the same size (with the exception of the last one), we can concatenate them without need of lemma 1.

We do the same on the level of the whole bit string by storing an array $\text{global}$:

$$\text{global}[k] = \sum_{x=0}^{kB-1} [S[x] = \ast].$$

The array $\text{global}$ has $\frac{N}{B}$ elements each of size $\log(N + 1)$ bits resulting in size asymptotically $O \left( \frac{N}{\log N} \right) = o(N)$.

**Algorithm**

We find out which block $k$ and which small block $l$ inside the block contains the position $i$, then we simply sum the precomputed values in arrays $\text{global}$, $\text{block}_k$ and the result of the look-up table for the small block:

```plaintext
function \text{RANK}(S, i):
    k ← \frac{i}{B}, \quad l ← \frac{i\%B}{b}, \quad s ← b_i^k
    return \text{global}[k] + \text{block}_k[l] + \text{rank}[S[s : s + B], i \% b]
```

**Implementation Details**

The look-up table could be substituted by an instruction `POPCNT` which is part of Advanced Bit Manipulation instruction set which is provided by many CPU architectures ([Int09]). Alternatively an algorithm with a theoretic running time $O(\log \log N)$ can be used, which leads in practice to a negligible slowdown, as the practical implementation still uses a fixed number of instructions.

2.1.2 SELECT

The operation SELECT is more complicated than RANK as it handles parts of the bit string differently depending on the local density of $\ast$. We show the index as it was proposed by [Cla98].

Two instances of the index are also necessary, since there is no identify which would allow a reduction from SELECT$_1$ to SELECT$_0$.

The SELECT operation also covers the bit string with multiple levels of blocks. The biggest difference from RANK is that the sizes of the blocks are not fixed – they are chosen by the number of symbols $\ast$ or their density in the bit string.

We cover the bit string by blocks and store offsets of the blocks within the bit string. If a block is sparse, we store a list of offsets of all symbols $\ast$ in it. Otherwise we process the block with a look-up table.

The bit string is “covered” by blocks beginning with $\ast$, each containing $B$ symbols $\ast$ with the exception of the last one. Note that not every symbol of the bit string is covered since there can be sequences of non-$\ast$ symbols outside of the blocks; this does not matter as they are not interesting for the queries. The $\text{global}$ array contains offset of all blocks.
We call a block \textit{sparse} if its size is greater than or equal to $K$ bits, otherwise it is called \textit{dense}. The value of $K$ will be determined later. This property can be easily tested from the difference of two consecutive offsets in \textit{global}.

If a block is sparse, we store the block-local offsets of all symbols $\ast$ in an array \textit{block\_enum}. We will deal with dense blocks later, once we discuss the constraints on $B$ and $K$ and set their values. For the space density of \textit{global}, the following must hold in order to obtain a succinct index:

\begin{align*}
\frac{N\log N}{N} &= \frac{\log N}{B} = o(1), \\
B &= \omega(\log N).
\end{align*}

Each offset in the enumeration of a sparse block can be in range $[0, N)$, resulting in up to $\log N$ bits of space. We have a lower bound on the size of a block ($K$) which allows us to phrase a constraint on the upper bound of the block density. We need:

\begin{align*}
\frac{B \log N}{K} &= o(1), \\
K &= \omega(B \log N).
\end{align*}

In order to have a strong bound for the dense blocks, we want to set $B$ and $K$ as small as possible. A sensible choice is to set:

\begin{align*}
B &= \log N \log \log N, \\
K &= (\log N \log \log N) \log N \log \log N = B^2.
\end{align*}

There is one more structure which needs to be discussed – an array \textit{blocks} of pointers to the beginnings of representations of blocks. Nevertheless, its space is the same as of the array \textit{global}.

We follow in a similar way with the index for a dense block. We cover it by \textit{small blocks}, each containing $b$ symbols $\ast$, with the exception of the last one. Each dense block has an array \textit{block} of block-local offsets of beginnings of small blocks. A small block is called \textit{sparse} if its size is greater than or equal to $k$ bits, and \textit{dense} otherwise.

If a small block is sparse, we store small-block-local offsets of all symbols $\ast$ in an array \textit{small\_block\_enum}. An array \textit{small\_blocks} pointing to the beginnings of the representations of small blocks exists again. The lower bound on the size of a dense block is $B$ bits. The constraint on $b$ from the definition of \textit{block} is:

\begin{align*}
\frac{B \log K}{B} &= \frac{\log K}{b} = o(1), \\
b &= \omega(\log K) = \omega(\log(\log N \log \log N)^2) = \omega(\log \log N).
\end{align*}

Each offset in the enumeration of a sparse small block can be in range $[0, K)$ resulting in up to $\log K$ bits of space. The lower bound on the size of a block used for calculating the index density is $k$:

\begin{align*}
\frac{b \log K}{k} &= o(1), \\
k &= \omega(b \log K).
\end{align*}
We choose:

\[ b = \log \log N \log \log \log N, \]
\[ k = (\log \log N \log \log \log N) \log(\log N \log \log N)^2 \log \log \log N = b^2. \]

It remains to discuss the constraint on the array \textit{small\_blocks} which contains \( \frac{B}{b} \) pointers of size \( l \). Since the representation of sparse small blocks is succinct, it is smaller than \( K \) per block. The size \( l \) of the pointer is \( \log K \), and therefore the size of the whole array poses the same constraint as the array \textit{block}.

We have been neglecting the case of dense small blocks; now we show how it is handled. Dense small blocks are limited to size \( k = o(\log N) \), which makes it possible to process them using a precomputed look-up table. Only one look-up table is sufficient because all dense small blocks can be padded from the right to the same size, while retaining the same result. No additional structure is necessary for small blocks; the pointer in the array \textit{small\_blocks} is a dummy value.

\textbf{Algorithm}

For a given parameter \( r \) we find in which block it is contained by division by \( B \) and note the reminder as a block-local index. In the array \textit{global} we find the offset of such block, and access its representation via the pointer in the array \textit{blocks}. If the block is sparse, we index the array \textit{block\_enum} of all occurrences to obtain the block-local offset; together with the block offset they form the answer.

If the block is dense, we find in which small block the \( r \)-th symbol \( * \) lies by division by \( b \) and note the small-block-local index. In the array \textit{block} we find the block-local offset of the small block. If the small block is sparse, we index the array \textit{small\_block\_enum} to obtain the small-block-local offset; together with the block offset and the small block offset they form the answer.

If the small block is dense, we use the look-up table to find the small-block-local offset. The answer is then computed in a similar fashion as in the previous case. We are guaranteed that there exists a constant \( c \) such that for all \( N \), the following inequality holds: \( k < c \cdot \frac{1}{2} \log N \), which means that we are done from the theoretical point of view.

In practice, when we are interested in all values \( N \) and not only those big enough for which \( c \leq 1 \), we need to iterate over chunks of \( s = \frac{\log N}{2} \) bits of the small block until the desired position is found. The look-up table is extended to return \(-1\) when the desired symbol does not lie in the current chunk. Note that the loop of the stated algorithm terminates, because we are guaranteed that the small block contains the desired symbol.
function \text{SELECT}(S, r):
    \begin{align*}
    a & \leftarrow r, \quad r' \leftarrow r \mod B \\
    \text{if} \ a \text{ is sparse:} & \quad \text{return } \text{global}[a] + \text{blocks}[a].\text{block enum}[r'] \\
    \text{else:} & \quad a' \leftarrow r', \quad r'' \leftarrow r' \mod b \\
    j & \leftarrow \text{global}[a] + \text{blocks}[a].\text{block} \\
    \text{if} \ a' \text{ is sparse:} & \quad \text{return } j + \text{blocks}[a].\text{small blocks}[a'].\text{small block enum}[r''] \\
    \text{else:} & \quad \text{while true:} \\
    & \quad \text{j' \leftarrow select}[S[j : j + s], r''] \\
    \text{if} \ j' = -1: & \quad r'' \leftarrow r'' - \text{rank}[S[j : j + s], s] \\
    & \quad j \leftarrow j + s \\
    \text{else:} & \quad \text{return } j + j'
    \end{align*}

Implementation Details
We can do the same trick as in \text{RANK}: we replace the theoretical look-up table with instructions of modern processors, namely \text{POPCNT} and \text{PDEP}. The later instruction is part of Bit Manipulation Instruction Set 2; details of this instructions are described in the Intel manual [Int09]. Using these instructions allows us to process dense small blocks by chunks of $s = 64$ bits.

The while loop leads to tens to hundreds of iterations which is the main reason for the bad performance of this algorithm. Better results can be achieved by using a different indices which despite not being necessarily succinct, tend to be smaller and faster in real situations. The details can be found in [KNKP05].

2.2 Balanced Bit Strings

A balanced bit string is a bit string containing opening and closing parentheses (encoded by 1 and 0) such that each parenthesis has its matching one in the bit string. Since we aim for a systematic succinct data structure, we need to design the storage of the data and the indices separately.

The universe of balanced bit string with $n$ opening and closing parentheses has exactly $C_n = \frac{1}{n+1} \binom{2n}{n}$ elements, where $C_n$ represents the $n$-th Catalan number. The number of bits required for representation of elements in this universe is:

$$\log C_n \sim \log \frac{4^n}{n^2 \sqrt{n \pi}} = 2n - \log n - \frac{1}{2} \log \pi n \sim 2n - O(\log n).$$

Therefore, we can store the balanced bit string using the trivial encoding of parentheses as zeros and ones while retaining the succinctness of the data structure. We use $N = 2n$ to refer to the size of the bit string.

The operations and indices which we have shown for general bit strings stay the same with the minor difference that they are defined for parentheses instead
of bits. Because of the additional structure, which stems from the balanced property, we define more operations on balanced bit strings:

\( \text{find\_close}(i) \rightarrow j \)  
Returns the position \( j \) of the closing parenthesis which is paired with the opening one at position \( i \); such parenthesis is guaranteed to exist. If the parenthesis at the position \( i \) is not an opening one, the result is \( i \).

\( \text{find\_open}(i) \rightarrow j \)  
If \( S[i] = "(\) \), then \( i \) is returned; otherwise it returns the position \( j \) such that \( \text{find\_close}(j) = i \).

\( \text{match}(i) \rightarrow j \)  
Returns the position \( j \) of the parenthesis which is paired with the one on position \( i \). This operation is just a convenient wrapper around \text{FIND\_OPEN} and \text{FIND\_CLOSE} depending on the parenthesis at the position \( i \). It is defined as:

\[
\text{match}(i) = \begin{cases} 
\text{find\_close}(i) & \text{if } S[i] = "("; 
\text{find\_open}(i) & \text{otherwise.}
\end{cases}
\]

\( \text{excess}(i) \rightarrow d \)  
Returns the difference \( d \) between the number of opening and closing parentheses until the position \( i \):

\[
\text{excess}(i) = \text{rank}_1(i) - \text{rank}_1(i).
\]

\( \text{paren\_depth}(i) \rightarrow d \)  
The depth \( d \) of a parentheses pair \((i, \text{match}(i))\) is defined as:

\[
d = \text{paren\_depth}(i) = \text{excess}(\text{find\_open}(i)) - 1 = \text{excess}(i) - [S[i] = "("].
\]

In order to distinguish the depth of a parentheses pair and the depth of a vertex in a tree (which we will use often later), we call the operation \text{PAREN\_DEPTH}.

\( \text{enclose}(i) \rightarrow j \)  
Returns the position \( j \) of the opening parenthesis which tightly encloses the pair \((i, \text{match}(i))\). If the result of \text{ENCLOSE} is not defined, we set it to \(-1\):

\[
\text{enclose}(i) = -1 \text{ if } \text{paren\_depth}(i) = 0.
\]

In all the other cases, the following (in)equalities hold:

\[
k = \text{enclose}(i),
\text{excess}(k) = \text{excess}(\text{find\_open}(i)) - 1,
\text{match}(k) > \text{find\_close}(i).
\]

Sometimes \text{ENCLOSE} can be generalized to take two parameters:
enclose\((i_1, i_2)\) → \(j\)
Returns the position \(j\) of an opening parenthesis which tightly encloses the pairs \((i_1, \text{match}(i_1))\) and \((i_2, \text{match}(i_2))\):

\[
\begin{align*}
\text{excess}(j) &< \min(\text{excess}(\text{find\_open}(i_1)), \text{excess}(\text{find\_open}(i_2))) \\
\text{match}(j) &> \max(\text{find\_close}(i_1), \text{find\_close}(i_2)).
\end{align*}
\]

This operation is indeed a generalization of the one-parameter ENCLOSE:

\(\text{enclose}(i, i) = \text{enclose}(i)\).

The result does not have to exist in a general balanced bit string, however it will not be an issue in our case.

\(\text{rmqi}(i_1, i_2) \rightarrow j\)

\(\text{Range Minimum Query}\) – Returns the position \(i_1 \leq j \leq i_2\) such that:

\[
\text{excess}(\text{rmqi}(i_1, i_2)) \geq \text{excess}(k) \quad \forall i_1 \leq k \leq i_2
\]

If there are multiple positions with the same minimum excess, then the leftmost one is returned.

The value of the minimum is \(\text{rmq}(i_1, i_2) = \text{excess}(\text{rmqi}(i_1, i_2))\).

\(\text{RMQi}(i_1, i_2) \rightarrow j\)

\(\text{Range Maximum Query}\) – Returns the position with maximum excess.

We will restrict ourselves to balanced bit strings which contain only one pair of parentheses such that their paren\_depth is equal to zero. This restriction changes the size of the universe to \(C_{n-1}\) which in a negligible difference of 2 bits.

It is sufficient to show only the succinct indices for \text{find\_close}, \text{enclose} and \text{rmqi} since the others are either similar or derived from them. The first two are even handled by the same index. The index for the two-parameter \text{enclose} is then implemented using \text{rmqi}.

### 2.2.1 Structure for MATCH and ENCLOSE

We describe a succinct index which supports \text{find\_close}, \text{find\_open}, and \text{enclose} with one parameter. We follow the description given by [GRRR06]; other options are summarized in [ACNS10].

First, we cover the balanced bit string by blocks of size \(B = \frac{\log N}{2}\). For each position \(i\) we denote \(B(i)\) the block to which it belongs.

**Pioneers**

We provide definitions of special parentheses:

- **far, near**

  A parenthesis \(i\) is far if \(B(i) \neq B(\text{match}(i))\); otherwise we call it near. Note that the matching parenthesis of a far parenthesis is also a far parenthesis.

  We observe that each block contains first all closing far parentheses and then all opening far parentheses.
opening (closing) pioneer
An opening (closing) far parenthesis \( i \) is an opening (closing) pioneer if the matches of \( i \) and of a preceding opening (following closing) far parenthesis \( j \) are located in different blocks:

\[ j < i \text{ and } B(\text{match}(j)) \neq B(\text{match}(i)). \]

Note that a matching parenthesis of an opening (closing) pioneer does not have to be a closing (opening) pioneer.

pioneer
A pioneer is either an opening or closing parenthesis pioneer or its matching parenthesis. A pioneer pair is a pair of matching parentheses which are pioneers.

Note that pioneers form a subsequence of parentheses which are correctly matched. Also note that the first opening far parenthesis and the last closing far parenthesis in a block are pioneers.

Example. In the following bit strings \( S \) we mark the positions of the far parentheses (\texttt{focp}), opening pioneers (\texttt{o}), closing pioneers (\texttt{c}), and pioneers (\texttt{ocp}):

\[ S = (((() | (()()) | )))) \]

Lemma 3. There are \( O \left( \frac{N}{B} \right) \) pioneers.

Proof. For every pair of blocks there exists at most one pioneer pair. This is certainly true if the opening and closing pioneers are considered separately. Let’s assume there are two pioneer pairs between this pair of blocks: one with an opening pioneer, the other one with a closing pioneer. Because the pioneer parentheses are correctly matched, one pair is enclosed by the other one, and therefore the opening nor the closing parenthesis of the enclosed pair cannot be pioneers by definition.

Let’s consider a graph whose vertices are blocks of the bit string and edges are between the blocks which are connected by a pioneer pairs. Such graphs is an outerplanar graph with a bound on number of edges: \( |E| \leq 2|V| - 3 \) while \( |V| = O \left( \frac{N}{B} \right) \). There are at most \( E \) pioneer pairs, and therefore at most \( 2|E| \) pioneers.

\[ \square \]

2.2.2 Block Queries
We will use two similar look-up table to answer all queries within a block.

- \texttt{fwd\_search}[S, i, d, paren, far] returns the first position \( j \geq i \) for which holds that \( \text{excess}(S, j) = \text{excess}(S, i) + d \) and \( S[j] = \text{paren} \). If \( \text{far} = \text{true} \), then \( j \) must be a far parenthesis.
- \texttt{bwd\_search}[S, i, d, paren, far] is the same except for \( j \leq i \).

The look-up tables return \(-1\) if such position does not exist in the queried block.

There are two special cases which we address separately:
A block query for a matching parenthesis

We distinguish two cases depending on the parenthesis $S[i]$:

$$match(S, i) = \begin{cases} 
    \text{fwd_search}([S, i, -1, ""], false) & \text{if } S[i] = "("; \\
    \text{bwd_search}([S, i, 1, ""], false) & \text{otherwise.}
\end{cases}$$

A block query for an enclosing parenthesis

We assume that $S[i]$ is an opening parenthesis since the other case will never occur. We run two queries for which we return the first non-negative result:

1. $\text{bwd_search}([S, i, -1, ""], false)$ which returns $enclose(S, i)$,
2. and $\text{fwd_search}([S, i, -2, ""], false)$ returning $match(enclose(S, i))$.

We aim to reduce the queries from the original bit string $S$ to queries on a bit string $P$ consisting of pioneers. We are allowed to store the array $P$ since its size is $O\left(\frac{N}{\log N}\right) = o(N)$.

We will also use a structure which tells us the positions of pioneers in the bit string $S$. A naïve approach would be to use a bit string $P'$ marking the positions of pioneers: $P'[i] = 1 \iff i$ is pioneer. We equip $P'$ with indices for $\text{rank}_1$ and $\text{select}_1$. The problem is that the size of the bit string $P'$ is $N$ instead of $o(N)$. To address this problem, we use a fully indexable dictionary which we introduce in the next section.

Reduction of $\text{find_close}$

The operation $\text{find_close}(S, i)$ is performed using $\text{find_close}(P, i')$ as follows: If $S[i]$ is a close parenthesis, we return $i$. We use the look-up table to find out whether the answer exists in the block $B(i)$ and possibly return it.

Otherwise, $i$ is an opening far parenthesis. Either $i$ is pioneer or we find the preceding pioneer; we denote it $j$ in both cases. It must be an opening parenthesis because the first opening far parenthesis in a block is such and there cannot be another pioneer pair between $i'$ and $i$. We find the match of $i'$ as:

$$k = \text{find_close}(S, i') = \text{select}_1(P', \text{find_close}(P, \text{rank}_1(P', i'))).$$

If $i = i'$, then $k$ is the answer which we return.

Otherwise, we know that $B(k) = B(\text{find_close}(S, i')) = B(\text{find_close}(S, i))$; else $i$ would have been a pioneer. To find the answer within the block $B(k)$, we use a look-up table $\text{BWD_SEARCH}$ to find the far parenthesis $j$ with the right excess difference $d = \text{excess}(i) - \text{excess}(i')$. The parenthesis $j$ is guaranteed to exist.
**function** FIND_CLOSE(S, i):
    if S[i] = ")":
        return i
    else:
        j' ← fwd_search[S[B(i)B : (B(i) + 1)B], i, −1, ")", false]
        if j' ≠ −1:
            return B(i)B + j' ▷ Same block
        else:
            i' ← pred1(P', i)
            k ← select1(P', find_close(P, rank1(P', i')))) ▷ Recursion
            d ← excess(i) − excess(i')
            j' ← bwd_search[S[B(k)B : (B(k) + 1)B], k % B, d, ")", true]
            return B(k)B + j'

The operation FIND_OPEN is reduced similarly; together they provide the operation MATCH.

**Operation ENCLOSURE**

We show how to perform the operation enclose(S, i). We assume without loss of generality that i is an opening parenthesis.

If the answer to enclose(S, i) is within the block B(i), we use a look-up table to report it, and stop. If the answer to k = enclose(S, find_close(i)) is within the look-up table, we report it, and stop. The parenthesis found by the look-up table can be return a closing parenthesis, which for which we find its matching opening parenthesis before reporting the answer.

Otherwise we aim for recursion. The parenthesis pair (j, match(j)) tightly enclosing i is not contained in the block B(i), therefore both parentheses must be far. Since there exists an edge between blocks B(j) and B(match(j)), there must exist exactly one pioneer pair (f, match(f)) connecting these blocks. We find the parenthesis f depending on the nearest pioneer i' = succ1(P', i), which is enclosed by i and therefore also by j:

1. S[i'] is a closing parenthesis. Then its matching opening parenthesis is at position f = find_open(i') < i and f is the opening parenthesis of the pioneer pair for which we were looking. It cannot happen that f ≥ i because f is a pioneer and it would have otherwise been found instead of i' by the successor query.

2. S[i'] is an opening parenthesis. Then f < i ≤ i' < find_close(i) < match(f). At the same time f = enclose(i'), which we solve by recursion.

Once we have f, we continue in the similar way as in case of find_close — we use a look-up table to find the parenthesis with the right excess in the block B(f).
function ENCLOSE(S, i):
    if S[i] = "\n":
        i ← find_open(i)
        j' ← bwd_search[S[B(i)B : (B(i) + 1)B], i % B, -1, "\n", false]
        if j' ≠ -1:
            return B(i)B + j'  \开口在同一个块
        i'' ← find_close(i)
        j' ← fwd_search[S[B(i'')B : (B(i'') + 1)B], i'' % B, -2, "\n", false]
        if j' ≠ -1:
            return find_open(B(i'')B + j')  \闭合在同一个块
    i' ← succ1(P', i)
    if S[i'] = "\n":
        f ← find_open(i')
    else:
        f ← select1(P', enclose(P, rank1(P', i'))))  \递归
    d ← excess(i) − excess(f) − 1
    j' ← bwd_search[S[B(k)B : (B(k) + 1)B], k % B, d, "\n", true]
    return B(k)B + j'

Recursion

The recursion as it was defined reduces the query from a bit string of size \( N \) to one of size \( O\left(\frac{N}{\log N}\right) \). After \( t \) levels the bit string has size \( O\left(\frac{N}{\log^t N}\right) \); we could use \( t = O\left(\frac{\log N}{\log \log N}\right) \) to reduce the size to \( O(1) \) which would guarantee that the query fits in a single block. However that would result in a superconstant time complexity of the operation.

We instead require only a constant number of levels of the recursion, \( t = 2 \) is sufficient. For every position in a bit string of size \( O\left(\frac{N}{\log^2 N}\right) \) we can precompute the answers to both operations MATCH and ENCLOSE; such table \( T \) has size \( O\left(\frac{N}{\log^2 N} \log \frac{N}{\log^2 N}\right) = O\left(\frac{N}{\log N}\right) = o(N) \). Note that this table is not a universal look-up table which could be shared among multiple instances of the data structure.

We do not even need to represent the bit string \( P \) on the second level since we only need the index to the table \( T \) provided by the indexable dictionary \( P'_2 \).

All space complexities so far are:

- \( O\left(\frac{N}{\log N}\right) \) – the bit string \( P \) on the first level;
- \( O\left(\frac{N}{\log N} \log \log N\right) \) – the fully indexable dictionary \( P' \) on the first level;
- \( O\left(\frac{N}{\log^2 N} \log \log N\right) \) – the fully indexable dictionary \( P'_2 \) on the second level;
- \( O\left(\frac{N}{\log N}\right) \) – the precomputed table \( T \) on the second level;
• $O\left(\sqrt{N} \log^2 N \log \log N\right)$ – the universal look-up tables \textsc{fwd}\_search and \textsc{bwd}\_search.

Note that we have no special requirements on the operations supported by $S$ and $P$; we only need to access up to $B$ consecutive bits, which bit strings allow.

### 2.2.3 Index for Range Minimum Query

The problem of range minimum query has applications in many areas and therefore is well studied: \cite{BFC00,Fis10,Dur13}. In this section we first define a more general problem (as did \cite{SN10}) which we fully solve in a later chapter. We then show a simple succinct index which will still be sufficient in most cases and will be needed for the general solution.

Although range minimum query can be defined for an arbitrary bit string (or even an array of numbers), it will be useful only for balanced bit strings. That is the reason why we present it here.

#### $\pm 1$ Functions

We first describe all possible functions $g$ which are mapping the values of bits $\{0, 1\}$ into a set $\{-1, 0, 1\}$. We call all the functions $g$ as $\pm 1$ functions. They are:

- $\varphi(b) = b$,
- $\psi(b) = 1 - b$,
- $\pi(b) = 2b - 1$,
- inverses ($-\varphi(b), -\psi(b), -\pi(b)$) and constant functions mapping each value of $b$ to the same fixed value. They are mentioned only to clarify that there are $3^2 = 9$ functions in total. These functions will not be useful for us.

We define an array $G$ for a function $g$ as:

$$G[i] = \begin{cases} \sum_{k=0}^{i} g(S[i]) & \text{if } i \geq 0; \\ 0 & \text{otherwise}. \end{cases}$$

We redefine the operation \textsc{rmqi} and \textsc{RMQi} to use the array $G$ instead of excesses of the array $S$. Range Minimum Query $\textsc{rmqi}(G, i_1, i_2)$ returns $j$ such that $G[j] \leq G[k] \forall i_1 \leq k \leq i_2$ and $j$ is the smallest such position. Similarly we define Range Maximum Query $\textsc{RMQi}(G, i_1, i_2)$ returning the leftmost position of the maximum value of $G$ in range $[i_1, i_2]$. We also define function $\textsc{rmq}(G, i_1, i_2)$ and $\textsc{RMQ}(G, i_1, i_2)$ returning the value rather than the position.

We can see that only the function $\pi$ is useful as the others are monotonic and therefore the range minimum is at the position $i_1$ and maximum at the position $i_2$. An alternative definition of the array $G$ applied to $\pi$ (which we call $E$) uses the excess function:

$$E[i] = \sum_{k=0}^{i} \pi(S[i]) = \text{rank}_1(i) - \text{rank}_0(i) = \text{excess}(i).$$

Although we defined the array $G$ to be derived by an application of a function on individual bits of $S$, we only need that $O(\log N)$ consecutive elements of the
array $G$ can be derived from $O(\log N)$ consecutive bits of $S$. A function which we could use instead of EXCESS is for example PAREN\_DEPTH. It differs from EXCESS by one at positions of opening parentheses and it can be expressed as a function depending on two consecutive bits:

$$\delta(a, b) = \begin{cases} 
-1 & \text{if } a = 0 \text{ and } b = 0; \\
0 & \text{if } a = 0 \text{ and } b = 1; \\
0 & \text{if } a = 1 \text{ and } b = 0; \\
1 & \text{if } a = 1 \text{ and } b = 1.
\end{cases}$$

An array $D$ of depths of parentheses is then defined as:

$$D[i] = \sum_{k=0}^{i} \delta(S[i-1], S[i]) = \text{paren\_depth}(i).$$

The property that $|G[i] - G[i - 1]| \leq 1$ will be useful much later when we introduce a more complicated and also more versatile structure.

**A Simple Index**

We build a succinct index supporting range minimum/maximum queries for the array of excesses. We only show the case of RMQI as RMQI is realized the same way; also RMQ and RMQ can be implemented straightforward using RMQI as RMQI and EXCESS. The array $E$ is never stored explicitly.

When we solve queries on range $[i, j]$ on top of an array which is decomposed into blocks, we split it into three parts based on which blocks are fully contained in the range. Since the functions min and max are distributive over a concatenation of ranges, each part of the original range can be processed independently. We call the three parts, which together form the original range:

- **prefix**
  the non-full block containing $i$;

- **suffix**
  the non-full block containing $j$;

- **span**
  the interval of full blocks between $i$ and $j$.

Any of prefix, suffix or span (or more of them) can be empty in a query.

There is a special case of a range which fully is contained within a single block; we solve it separately using the following lemma.

**Lemma 4.** We can solve the operation $\text{rmqi}(E, i_1, i_2)$ in constant time whenever $i_2 - i_1 + 1 < \frac{\log N}{2}$.

**Proof.** We can express every element of $E$ as:

$$E[i_1 - 1] \sum_{k=0}^{i_2-i_1} \pi(S[i_1 + k]).$$
All the values depend only on the block $S[i_1 : i_2 + 1]$ and the value of $E[i - 1]$. Because the minimum is independent of the absolute value, the dependence on $E[i - 1]$ is irrelevant.

By the precondition of the lemma, the block has a size of at most $\frac{\log N}{2}$, which makes it possible to use it as an index to a look-up table RMQI. The look-up table is parametrized by the length of the block, which is the value of $j - i + 1$, which is encoded in $O(\log \log N)$ bits. The parameter is necessary because of the zero-padding of words in our definition of RAM. The table simply returns the position of the minimum.

We follow with a lemma which will solve the RMQI for spans. Although it has a large space complexity depending on the number of blocks, eventually, we will be able to lower it to $o(N)$ by the right choice of block sizes.

**Lemma 5.** Given an array $P$ of $p$ positions of minima in blocks on a lower level, we can solve the RMQI for span queries in constant time using $O(p \log^2 p)$ bits of memory.

**Proof.** We call $l = i_2 - i_1 + 1$ the number of blocks over which the query spans. We distinguish two cases:

1. $l = 2^k$ is a power of two. We simply use a value from a precomputed table $Tm$ which records the index of the block containing the minimum for each $i$ and $k$. This table of $O(p \log p)$ elements require $O(p \log^2 p)$ bits in total.

2. $l$ is not a power of two. We substitute the query on $l$ blocks by two queries each spanning $2^k$ blocks where $k = \lfloor \log l \rfloor$. The two queries overlap, however it does not cause any issue since we are only looking for a minimum of their answers. We gather both candidates and return the one which has the smaller value, while preferring the left one.

```python
function RMQI_SPAN(E, P, i_1, i_2):
    k ← \lfloor \log(i_2 - i_1 + 1) \rfloor
    p_1 ← Tm[i_1, k]
    p_2 ← Tm[i_2 - 2^k, k]
    if $E[P[p_1]] \leq E[P[p_2]]$:
        return $p_1$
    else:
        return $p_2$
```

The bit string $S$ is covered by blocks of size $B$ and small blocks of size $b = \frac{\log N}{2}$. For each block we store the position of its minimum in an array $P_1$, which requires $O\left(\frac{N}{B} \log B\right)$ bits of memory. We build the RMQI structure by lemma 5 on top of this array, which adds another $O\left(\frac{N}{B} \log^2 \frac{N}{B}\right)$ bits of memory.

For all small blocks in a block $i$ we precompute an array $P_2[i]$ of positions of their minima; this array uses $O\left(\frac{B}{b} \log \log N\right)$ bits per block. On top of this array we build again the RMQI structure, which uses $O\left(\frac{B}{b} \log^2 \frac{B}{b}\right)$ bits per block.

In order to keep all densities $o(1)$, we set $B = \log^3 N$. 

25
Algorithm

When we process the query, we first split the interval \([i_1, i_2]\) into:

1. up to one top-level span of blocks. The prefix and suffix are passed to the lower level.
2. up to two spans of small blocks, which form parts of the prefix and suffix from the top level;
3. up to two small blocks, which are not fully covered by the interval.

In the special case when the range is contained within a single block or a single small block, the range is passed to the lower level.

We gather the candidates for minimum in (1) and (2) by querying the RMQI structures and in (3) by using a look-up table as described by lemma 3. Once we have all candidates \(C\), we return the one with the lowest value:

\[
j = \arg \min_{c \in C} E[c].
\]

We omit the pseudo-code of the algorithm as it deals with many cases which are essentially the same.

Two-parameter ENCLOS

We use the operation RMQI to support the two-parameter ENCLOS.

Without loss of generality, we assume that both \(i_1\) and \(i_2\) are opening parentheses and that \(i_1 \leq i_2\). We first check if \(i_2\) is enclosed by \(i_1\); in such case we simply reduce the operation to a one-parameter ENCLOS.

Therefore the parentheses pair of \(i_1\) does not contain \(i_2\) and vice versa. By definition, we are looking for a parentheses pair \((j, \text{match}(j))\) which spans over the interval from \(\min(i_1, i_2)\) to \(\text{match}(\max(i_1, i_2))\). The parentheses pair of \(j\) contains two parentheses pairs \(p_1 < p_2\) such that each of them contains one of \(i_1, i_2\) and \(\text{excess}(p_1) = \text{excess}(p_2) = \text{excess}(j) + 1\).

We observe that the following properties hold for \(\text{find}_\text{close}(p_1)\):

- it is contained in the interval \([i_1, i_2]\);
- it has the minimum EXCESS on such interval;
- it is the leftmost parenthesis with such EXCESS.

The second property follows from the fact that it is a closing parenthesis and that:

\[
\text{excess}(k) \geq \text{excess}(j) \forall j \leq k < \text{match}(j).
\]

We also observe that \(\text{find}_\text{close}(p_1) + 1\) is an opening parenthesis of the next parentheses pair following \(p_1\). These properties allow us to find a parenthesis which is tightly enclosed by \(j\) and reduce the query to a one-parameter enclose:

\[
j = \text{enclose}(i_1, i_2) = \text{enclose}(\text{rmqi}(i_1, i_2) + 1).
\]
2.3 Dictionaries

The bit strings which we have been discussing so far can be seen from a different point of view: A bit string $S$ of size $N$ is a representation of a set $A$, which is a subset of $[0, N)$ such that a number $x \in A \iff S[x] = 1$.

The natural encoding is the one which we have used – representing the membership of each number in the universe $[0, N)$ by a bit in a characteristic vector. It is still true that such encoding is succinct as there are $|\mathcal{A}(N)| = 2^{|[0, N)|} = 2^N$ possible sets in the universe, and therefore $N$ bits is required.

However, we can restrict the subsets $A$ of $[0, N)$ by the number of elements $K$. There are $\binom{N}{K}$ sets in the parametrized universe requiring $\log \binom{N}{K}$ bits of memory. From this point of view of sets parametrized by the number of elements, the encoding by the membership of each element is not succinct.

We define structures supporting various operations, which we already know from the general bit strings.

**dictionary**

A dictionary is a data structure which stores $A$ and can answer membership queries $x \in A$.

**indexable dictionary**

An indexable dictionary (ID) is a dictionary which can answer RANK for elements of the set $A$ and SELECT queries.

$$\text{rank}(A, i) = |\{a \in A : a \leq i\}|$$

$$\text{select}(A, j) = i \in A \text{ and } \text{rank}(A, i) = j$$

**fully indexable dictionary**

A fully indexable dictionary (FID) is an indexable dictionary for both sets $A$ and its complement $\overline{A}$.

We can extend the $\text{rank}(A, i)$ for all $i \in [0, N)$:

$$\text{rank}(A, i) = \begin{cases} 
\text{rank}(A, i) & \text{if } i \in A; \\
 i - \text{rank}(\overline{A}, i) + 1 & \text{otherwise.}
\end{cases}$$

Moreover, all operations PRED, SUCC, PREV, NEXT are well defined for FID. FID is equivalent to a general bit string.

We will use the notation from general bit strings:

$$\text{rank}_1(A, i) = \text{rank}(A, i),$$

$$\text{rank}_0(A, i) = \text{rank}(\overline{A}, i),$$

$$\text{select}_1(A, j) = \text{select}(A, j),$$

$$\text{select}_0(A, j) = \text{select}(\overline{A}, j).$$

The first and only succinct indexable dictionary was developed by [RRS07] in Theorem 4.1. We state their result as a lemma without proof.

**Lemma 6.** There is a succinct indexable dictionary which uses $\log \binom{N}{K} + o(K) + O(\log \log N)$ bits.
2.3.1 Sublinear Fully Indexed Dictionary

We show a simple fully indexable dictionary of a size $\log \binom{N}{K} + o(N)$. In all cases when we use this structure, the space complexity will be $o(N)$, which is the reason for its name. Note that if $N = 2K$, then the size is $N - O(\log N)$; balanced bit strings are a special case of such setup.

This structure is based on Lemma 4.1 in [RRS07]. The difference is that we reuse the succinct indices for RANK and SELECT which we showed in section 2.1 instead of implementing them again.

This FID is also the missing piece in the structure for MATCH and ENCLOSE in section 2.2.1.

The main idea of the data structure is to provide an access to $B = \log \frac{N}{2}$ bits of the characteristic vector. Since any algorithm can access up to $w = O(S)$ bits in a single step, we can provide it with a constant slowdown.

We split the characteristic vector $S$ into blocks $S_1, S_2, \ldots$ of size $B$ with each of them having $K_1, K_2, \ldots$ ones. We extend $N$ so that the last block is full; this difference is negligible.

We represent each block $i$ implicitly by a binary number $[0, \left(\frac{B}{K_i}\right))$ encoded in $b_i \leq S$ bits. The space bound for storing all blocks consecutively as a bit string $S'$ follows from the generalized Chu-Vandermonde’s identity ([Bel]).

$$
\sum_{K_1 + K_2 + \cdots = K} \prod_i \binom{B}{K_i} = \binom{N}{K}
$$

$$
\sum_i \log \binom{B}{K_i} \leq \log \binom{N}{K}
$$

$$
\sum_i b_i \leq \log \binom{N}{K} + \frac{N}{B}
$$

We also store two arrays of size $O\left(\frac{N}{B} \log B\right)$: $L = [K_i]$, $C = [b_i]$. We can now restore any block $i$ into its characteristic vector by a simple look-up table provided that we know where the representation of the block starts. We group log $N$ blocks into macro-blocks and store where each macro-block starts in an array global using $O\left(\frac{N}{\log^2 N} \log N\right)$ bits. And for each macro-block we store the macro-block-local positions of beginning of blocks which it contains in an array block, using $O(\log N \log \log N)$ bits per macro-block. This is essentially the same as the partitioning which we used for RANK in section 2.1.1.

function CHARACTERISTIC_BLOCK($S', b$):

$m = b \log N$

$p \leftarrow \text{global}[m] + \text{block}[b \% \log N]$

return characteristic_block[$S'[p : p + C[b] + 1], L[b]$]

The implementation of the operations RANK, SELECT, and INSPECT, which are required by the definition of FID, is straightforward. The total size of the structure is $\binom{N}{K} + O\left(\frac{N \log \log N}{\log N}\right) = \binom{N}{K} + o(N)$. 

28
2.3.2 Compressed Array

In an array, a run is a consecutive sequence of identical elements.

Let’s assume that we have an array $A$ of $a$ elements of size $s$ which contains $r = o(a)$ runs. The goal will be to store $A$ in space $o(a)$.

We can compress such array $A$ by storing:

A_FID

A fully indexable dictionary containing the positions of the last elements of each run:

$$A_{\text{FID}} = [i : A[i + 1] \neq A[i]].$$

The FID contains $r$ values resulting in the space complexity $\log \binom{a}{r} + o(a)$.

A_elems

An array $A$ with each run reduced to a single (last) element:

$$A_{\text{elems}} = [A[i] : i \in A_{\text{FID}}].$$

The array contains $r$ elements of total size $rs$ bits.

A_before

An array containing the numbers of occurrences of the element $A$ before the current run:

$$A_{\text{before}} = [\{j : j < \text{prev}_1(A_{\text{FID}}, i) \text{ and } A[j] = A[i]\} : i \in A_{\text{FID}}].$$

The size of the array is $r \log a$.

We define several operations for this compressed array:

ELEMENT_INDEX

Returns the number of the run which contains position $i$:

$$\text{element_index}(A, i) = \text{rank}_1(A_{\text{FID}}, \text{succ}_1(A_{\text{FID}}, i)).$$

INSPECT, $A[i]$

It provides access to any element:

$$A[i] = A_{\text{elems}}[\text{element_index}(A, i)].$$

RUN_FIRST, RUN_LAST, RUN_LENGTH

It returns the position of the first and last elements of the run containing $i$. RUN_LENGTH returns the length of the run. They are defined as:

$$\text{run_first}(A, i) = \text{prev}_1(A_{\text{FID}}, i) + 1,$n$$

$$\text{run_last}(A, i) = \text{succ}_1(A_{\text{FID}}, i),$$

$$\text{run_length}(A, i) = \text{run_last}(A, i) - \text{run_first}(A, i) + 1.$$ 

RANK

Returns how many elements same as $A[i]$ there has been until the position $i$:

$$\text{rank}(A, i) = A_{\text{before}}[\text{element_index}(A, i)] + (i - \text{run_first}(A, i) + 1).$$

SIZE

Returns the total number of elements $a$. 

29
Extending Compressed Array

A compressed array can be built on top of several arrays $A_0, A_2, \ldots, A_{t-1}$ with sizes $a_i$ containing $r_i$ runs. We refer to the structure as a collection of compressed arrays. The total space complexity follows a simple summation: $a = \sum_i a_i$ and $r = \sum_i r_i$.

We extend the structure of the compressed array by the following field:

**A_parts**

An array containing $t + 1$ elements of positions where $A_i$ starts:

- $A_{\text{parts}}[0] = 0$
- $A_{\text{parts}}[i] = \sum_{j=0}^{i-1} |A_j|$

The size of **A_parts** is the same as the size of **A_BEFORE**, which was built on top of an array of the same size containing a single part.

The compressed array is then built on top of a concatenated array

$$A = A_0 \cdot A_2 \cdots A_{t-1}$$

with the exception that the first element of an array $A_i$ always starts a new run. This makes sure that runs do not extend over several parts, which would make it harder, yet not impossible, to handle. This is already incorporated in the total number of runs: $r = \sum_i r_i$.

Using this extra array, which has same space complexity as **A_BEFORE**, we can support the operations on part $p \in [0, t)$ which differs by an initial offsetting of $i$:

$$i' = A_{\text{parts}}[p] + i.$$ 

Note that we can also answer SIZE of each part, and check the bounds before an operation is commenced.

We define a way how to turn a partially filled array $A'$ with a characteristic vector $C$, which is usually obvious from the context and definition of $A'$, into a compressed array $A$ for a given operation $op$ which is one of PRED, SUCC, PREV, NEXT. The missing elements in the array are defined as:

$$A[i] = A'[\text{op}(C, i)].$$

### 2.3.3 Tiny Compressed Array

The compressed array, as we defined it using the FID, is going to be sufficient for most cases. However it is impractical in situations when the number of runs is very small since the size of the FID depends polynomially on the size of the universe $a$ rather than on the number of elements $r$ which are represented by the set. We are interested in developing a more space-efficient structure for $r = O(\log a)$.

The polynomial dependency of FID on the size of the universe $a$ is not only problem of our quite simple data structure. In fact there is no known FID which
does not suffer with this problem. The two state of the art data structures [Pat08, GORR09] have a similar polynomial dependency.

We overcome the problem by replacing the FID by two more advanced data structures which together provide the same result under this restricted $r$. The resulting space complexity will be $O(\log^2 a)$ bits.

The first structure which we need is a fusion tree which was described by [FW93] and proven optimal for our case by [PT06]. We state their result as lemma which we do not prove.

**Lemma 7.** A fusion tree stores $n$ 1-bit integers and supports `pred` and `succ` operations in time $O(\log n)$ requiring space $O(nl)$ bits.

The compressed array requires only operations `prev_1`, `succ_1`, and `rank_1`; the fusion tree provides the first two. Applied to our problem of a set containing $O(\log a)$ values, we uses $O(\log^2 a)$ bits and answers the operations in time $O(1)$.

In order to support the `rank_1`, we use the indexable dictionary from lemma 6. Although, its `rank` is not universal – it is restricted to elements of the dictionary, in combination the `pred` operation of the fusion tree, it can still be supported.

It is worth noting, that the tiny compressed array is a theoretical data structure as it can be replaced by a simple sorted array. The operations `rank`, `select`, `pred`, and `succ` can be then processed in time $O(\log \log a)$, which for all practical purposes ($\log \log a \leq 6$) is negligible.
3. Representations of Trees

There are many ways to represent static ordinal trees in memory. The traditional way, which most people know and use, is to have a structure for each vertex pointing to all its children and its parent, and sometimes storing an additional value. Each edge of the tree is stored twice as a pointer of size $w$. Even if we did not need any other space to manage the vertex structures, the space occupied by the edges takes $\Omega(n \log n)$ bits.

In our discussion we assume that no additional value is present in vertices, and we are therefore interested in the structure of the tree itself. If values are present, we could move them to an external table which is indexed by the number of the vertex to which it belongs. This is one of the reasons why we will be interested in the $*_\text{rank}$ operations on tree. This transformation maintains the same space complexity.

Let $T(n)$ be a universe of all ordinal trees with $n$ vertices. The size of the universe can be expressed using Catalan numbers:

$$|T(n)| = C_{n-1} = \frac{1}{n} \binom{2n - 2}{n - 1}.$$  

We saw a similar bound for balanced bit string, that is not a coincidence, as there exists a bijection between trees and balanced bit strings. The difference of $-1$ stems from the fact that we are representing a tree with $n$ vertices rather than a forest of trees containing $n$ vertices in total.

The optimal number of bits for representing an ordinal tree is:

$$\log |T(n)| = \log \frac{1}{n} \binom{2n - 2}{n - 1}$$
$$\sim \log \frac{1}{n} \frac{4^{n-1}}{\sqrt{(n-1)}\pi}$$
$$= 2(n-1) - \frac{1}{2} \log (n-1)\pi - \log n$$
$$\sim 2n - O(\log n).$$

We used Stirling’s approximation for the binomial coefficient. When we design a succinct data structure for ordinal trees, we are limited by the space given by the leading term: $2n + o(n)$ bits.

In this and in the next chapter we show several succinct data structures which overcome the logarithmic factor introduced by using pointers, and get close to the optimum space complexity. We start with defining operations which are desired to be supported by the tree data structures. The rest of this chapter is then dedicated to simple representations which encode the whole tree in $\sim 2n$ bits and support various operations via additional indices, which we developed in chapter 2. More advanced structures which do not have the limitations of the simple structures are shown in the next chapter.
General operations which we grouped together since it is usually desired to support them all. The operations are defined for various parts of the tree. In all algorithms we assume that the bounds are checked before the operation commences.

If \(*\text{RANK}\), \(*\text{SELECT}\), and \(*\text{SIZE}\) are available, all the others can be expressed using them. Alternatively, an iteration over vertices is possible if \(*\text{PREV}\), \(*\text{NEXT}\), \(*\text{FIRST}\), \(*\text{LAST}\) are defined.

The \text{RANK} and \text{SELECT} operations come in several variants based on method of the tree traversal and when the vertices are assigned their numbers.

- \text{PREV} is defined when DFS visits a vertex from its parent; post-order when DFS leaves a vertex to its parent; \text{in-order} when DFS returns to a vertex from its child and leaves to another of its children; \text{dfuds} when a parent is first visited.
- \text{FIRST} and \text{ROOT} complete the set of the navigation operations.

Ancestral operations are usually the hardest to support; they usually require specialized indices. We define the level ancestor to be the \text{d}-th ancestor of \(i\). The operation \text{LCA} stands for Lowest Common Ancestor.

Table 3.1: List of operations defined for ordinal trees

<table>
<thead>
<tr>
<th>op (avg)</th>
<th>(\rightarrow)</th>
<th>(\text{Description})</th>
</tr>
</thead>
<tbody>
<tr>
<td>*\text{rank}(i) \rightarrow j</td>
<td></td>
<td>General operations which we grouped together since it is usually desired to support them all. The operations are defined for various parts of the tree. In all algorithms we assume that the bounds are checked before the operation commences. If (<em>\text{RANK}), (</em>\text{SELECT}), and (<em>\text{SIZE}) are available, all the others can be expressed using them. Alternatively, an iteration over vertices is possible if (</em>\text{PREV}), (<em>\text{NEXT}), (</em>\text{FIRST}), (*\text{LAST}) are defined. The \text{RANK} and \text{SELECT} operations come in several variants based on method of the tree traversal and when the vertices are assigned their numbers. - \text{PREV} is defined when DFS visits a vertex from its parent; post-order when DFS leaves a vertex to its parent; \text{in-order} when DFS returns to a vertex from its child and leaves to another of its children; \text{dfuds} when a parent is first visited. - \text{FIRST} and \text{ROOT} complete the set of the navigation operations. Ancestral operations are usually the hardest to support; they usually require specialized indices. We define the level ancestor to be the \text{d}-th ancestor of (i). The operation \text{LCA} stands for Lowest Common Ancestor.</td>
</tr>
</tbody>
</table>
3.1 Operations

The operations are defined in the table 3.1. The desired running time of all operations is $O(1)$.

Navigation operations are those which allow traversal of the tree. Since the operations \textsc{child\_prev}, \textsc{child\_next}, \textsc{child\_first} and \textsc{child\_last} can be emulated by \textsc{child\_rank}, \textsc{child\_select} and \textsc{degree}, the support of the later ones is preferred. In order to navigate up in the tree \textsc{parent} and \textsc{is\_root} are required.

In the definitions, we use $i$ and $j$ to refer to the internal representation of the vertex. They are not meant to be used for anything else except for being passed to an operation as an argument. If the number of the vertex matters, one of the supported ranking operations, which map the vertices into an interval $[0, n)$, shall be used.

The in-order \textsc{rank} is different from the others. Only vertex $i$ such that $\text{degree}(i) \geq 2$ is assigned in-order number; moreover it is assigned $\text{degree}(i) - 1$ numbers corresponding to transitions from one child to another one. When a \textsc{in\_rank} is used, the smallest assigned in-order number is returned, or $-1$ if a vertex was not assigned any.

3.2 Representations

An ordinal tree can be stored in a single bit string in several different ways. Here we describe how it is possible to store a tree in a bit string of size $N \sim 2n$ bits\footnote{We use $N$ to refer to size of the bit string, and $n$ to refer to number of vertices.}.

**Level-Order Unary Degree Sequence (LOUDS)**

LOUDS is the oldest representation with the most limitations on supported operations, for which it compensates by its simplicity as only \textsc{rank} and \textsc{select} operations are required. It stores the data in a heap-like way by levels, which results in subtrees being spread throughout the representation. The consequence of the storage is that it lacks all subtree-restricted and ancestral operations.

**Balanced Parentheses (BP)**

With \textsc{match} and \textsc{enclose} operations available on balanced bit strings, the BP representation was invented. It uses the natural mapping between ordinal trees and balanced bit strings. It supports many operations which were not possible before, however the price for it is that multiple indices are necessary. The \textsc{child\_rank} and \textsc{child\_select} operations are not supported natively.

**Depth-First Unary Degree Sequence (DFUDS)**

The DFUDS representation is a hybrid representation of the previous ones using an alternative mapping to a balanced bit string. The navigation queries are fully supported while it retains the locality of the subtrees.

Authors often compete and equip their DFUDS and BP structures with more indices that support operations which the other representation made available first.
3.3 Level-Order Unary Degree Sequence

The first method of representing an ordinal tree succinctly was first described by [Jac89]. The encoding of the tree is based on BFS traversal. The vertices report their degrees in the unary system; we call it the degree sequence. The representation is prefixed by "10".

```
function LOUDS_representation(R):
    Q ← [], enqueue(Q, R)  # The queue is initialized with the tree root.
    output("10")
    while V ← dequeue(Q):
        for all C ← children(V):
            output("1")
            enqueue(Q, C)
        output("0")
```

Lemma 8. There are $n$ ones and $n + 1$ zeros in the LOUDS representation of a tree with $n$ vertices.

Proof. The "10" corresponds to an artificial supervertex which has a single child – the root of the tree. Each vertex of the tree is a child of a vertex and therefore was accounted for by a one in the degree sequence of its parent. The number of ones is therefore the same as the number of all vertices in the tree. Each vertex is responsible for its own zero, plus there is one extra zero for the supervertex.

The whole bit string, which we call $S$, has $N = 2n + 1$ bits.

As there is a one and a zero belonging to each vertex, we can associate the $i$-th vertex with the $i$-th one and the $(i + 1)$-th zero. Note that the one is actually in the degree sequence of the parent of $i$, and the zero is terminating its degree sequence. This follows from the algorithm LOUDS_representation: the $i$ vertex assigned one when it is added into the queue, and zero when its degree sequence is outputted.

We use the position of the associated one for the internal representation of a vertex. A zeros-based structure is also possible as was showed by [RR06].

Example. The LOUDS representation of the tree in figure 3.1 is:

```
S = 10 | 1110 | 0 110 0 10 | 0 10 0 | 0
```

Levels are delimited by vertical bars, unary degree sequences of individual vertices by spaces. There is a vertex for each one. The underlined zeros are boundaries defined in section 3.3.2.
3.3.1 Navigation Operations

We equip the bit string $S$ storing the representation of the tree with \textsc{rank} and \textsc{select} indices. Because of the correspondence between ones and zeros, we can easily change between them using the following two functions. The extra function \textsc{to\_beginning} returns the position where the degree sequence starts.

\begin{verbatim}
function TO\_ONES(S, i):
    return select$_1$(S, rank$_0$(S, i) − 1)

function TO\_ZEROS(S, i):
    return select$_0$(S, rank$_1$(S, i) + 1)

function TO\_BEGINNING(S, i):
    return select$_0$(S, rank$_1$(S, i)) + 1
\end{verbatim}

Only one vertex numbering schema is supported by this representation – the level-order numbering. The operations \textsc{lo\_rank} and \textsc{lo\_select} are simply \textsc{rank}$_1$(S, i) and \textsc{select}$_1$(S, r).

The navigation operations are relatively easy. We also define an optimized version of the operation \textsc{child\_last} since we will need it later.

\begin{verbatim}
function IS\_ROOT(S, i):
    return i = 0

function IS\_LEAF(S, i):
    return to\_zeros(S, i) = to\_beginning(S, i) \triangleright Degree sequence is empty

function PARENT(S, i):
    return to\_ones(S, succ$_0$(S, i)) \triangleright Conversion from zeros to ones

function DEGREE(S, i):
    return to\_zeros(S, i) − to\_beginning(S, i) \triangleright Length of its degree sequence

function CHILD\_RANK(S, i):
    return i − pred$_0$(S, i) \triangleright Distance from the previous zero

function CHILD\_SELECT(S, i, k):
    return to\_beginning(S, i) + k − 1

function CHILD\_LAST(S, i): \triangleright CHILD\_FIRST is merely child\_select(S, i, 1).
    return to\_zeros(S, i) − 1
\end{verbatim}

The data structure can support more operations when augmented with more succinct indices.

3.3.2 Depth and Level Queries

We propose a new index for the LOUDS representation which allows us to query the depth of a vertex and which indirectly supports all level queries (without restriction on a subtree). This was not possible before.

We call a \textit{level boundary} the terminal zero of the last vertex in each level, including the zero which belongs to the supervertex. We define a bit string $D$ which contains 1 at the position of level boundaries.
The bit string $D$ has size $N$ bits which does not make it a succinct index; we will discuss this in the next section. We equip the bit string $D$ with succinct indices for $\text{RANK}_1$ and $\text{SELECT}_1$, which we described in the section 2.1. Using $D$, we can support $\text{LEVEL}_\text{RANK}$, $\text{LEVEL}_\text{SELECT}$ and $\text{LEVEL}_\text{SIZE}$ (and thereby all $\text{LEVEL}_*\text{ operations}$), and the operation $\text{DEPTH}$.

```
function \text{LEVEL}_\text{RANK}(S, i):
    z \leftarrow \text{prev}_1(D, i)
    \text{return } \text{rank}_1(S, i) - \text{rank}_1(S, z)
```

```
function \text{LEVEL}_\text{SELECT}(S, i, l):
    z \leftarrow \text{select}_1(D, l)
    \text{return } \text{select}_1(S, \text{rank}_1(S, z) + i)
```

```
function \text{LEVEL}_\text{SIZE}(S, l):
    z \leftarrow \text{select}_1(D, l)
    z' \leftarrow \text{select}_1(D, l + 1)
    \text{return } \text{rank}_1(S, z) - \text{rank}_1(S, z')
```

```
function \text{DEPTH}(S, i):
    \text{return } \text{rank}_1(D, i)
```

**Succinct Index for Depth**

We cannot store the bit string $D$ directly, we cannot even store it as a FID. If the tree has height $\Omega(n)$, then there is the same number of boundaries – ones in the bit string $D$, which leads to $\Omega(N)$ bits from purely combinatorial reasons. We work around the problem by utilizing the fact that if there are many levels, they must be short. They are in fact short enough to precompute the level boundaries in a look-up table.

We split the bit string $D$ into blocks of size $b = \log \frac{N}{2}$; for each block $i$ we remember only the position of the first 1, or lack thereof in a table $D'[i]$. The table $D'$ requires $\frac{N}{b} \log b = O \left( \frac{N \log \log N}{\log N} \right) = o(N)$ bits.

**Lemma 9.** Using the table $D'$, we can restore any block $i$ of $b$ consecutive bits of the original bit string $D$ in constant time.

**Proof.** If there is no one in the block $i$, there is also no level boundary and the block contains only zeros. Otherwise, there is a 1 at a position $p$. If $p = 2n$, then it is the last boundary which marks the end of the last level; the rest of the block is filled with zeros. From the properties of BFS it follows that the parent of the last vertex in a level is the last inner vertex in the previous level. An inner vertex contains 1 in its degree sequence. By finding the last 1 in the bit string $S$ before the boundary at position $p$, we find the last inner vertex in the previous level. Switching to the zero of its last child brings us to the position of the boundary $q$ for the level $i + 1$. We iterate this process until we get out of the block.

The crucial observation is that except for the first iteration, only bits from the block $i$ are needed. We can process the first iteration manually to obtain $q$ and then pass $p$ and $q$ to a look-up table $\text{BLOCK}_\text{OF}_\text{D}$ which determines the rest of the boundaries within the block. Given the boundaries of a single level, the
boundaries of the next level can be determined by counting the same number of zeros as there were ones in the previous level.

\[
\text{functions block}\_\text{of}\_\text{D}(S, i):
\]

\[
p' \leftarrow D'[i]
\]

if \( p' = -1 \) \( \triangleright \) No 1 in the block

return 0

else:

\[
p \leftarrow \text{ib} + p'
\]

if \( p = 2n \)

\( \triangleright \) The end of the last level

return \( 1 \ll (b - p' - 1) \)

\( \triangleright 1 \) at position \( p' \)

else:

\[
q \leftarrow \text{to_zeros}(S, \text{pred}_1(S, p))
\]

if \( q \geq (i + 1)b \) \( \triangleright \) The level ends outside of the block \( i \).

return \( 1 \ll (b - p' - 1) \)

else:

\[
\text{return block}\_\text{of}\_\text{D}[S[\text{ib} : (i + 1)\text{b}], p', q \mod b]
\]

In a constant number of queries we can generate any \( w \) bits of the original bit string \( D \), which is the requirement for using the technique of an index without data from section 1.4.3.

3.3.3 Leaf Operation

Tree data structures often allow a direct iteration of their leaves. Although the usual order of leaves corresponds to DFS traversal, LOUDS supports an access to the leaves in level-order in accordance with the its only ranking operation – LOU_RANK. We also parametrize the operations by the desired depth of the leaves, by upper and lower bounds \( u \leq l \).

Lemma 10. Whenever there are two consecutive zeros in the LOUDS representation, the latter one is associated with a leaf.

Proof. There is always a zero immediately preceding the representation of every vertex. A leaf is a vertex with its degree equal to zero; the degree sequence consists of only a single zero. A leaf can therefore be localized by looking for two consecutive zeros; this gives us its zeros-based number.

The original idea for this index comes from \cite{MRR98}. We have seen a solution to a similar problem in 1.4.3.

We derive a bit string \( L \) such that \( L[i] = [S[i - 1] = 0 \) and \( S[i] = 0] \):

\[
L = \sim S \& \sim (S \gg 1).
\]

We build RANK and SELECT indices for it.

All the functions require a correction by one if the first vertex of the level \( l \) is a leaf.

\[
\text{functions leaf}\_\text{size}(S, l, u):
\]

\[
f \leftarrow \text{level}\_\text{first}(S, l)
\]

\( \triangleright \) The first vertex in the range

\[
f_0 \leftarrow \text{to_zeros}(S, f)
\]

\[
z_0 \leftarrow \text{to_zeros}(S, \text{level}\_\text{last}(S, u))
\]

\[
\text{return rank}_1(L, z_0) - \text{rank}_1(L, f_0) + \text{is}\_\text{leaf}(S, f)
\]
function LEAF_RANK(S, l, i): ▷ We assume i is a leaf at the admissible level.
f ← level_first(S, l)                      ▷ The first vertex in the range
f_0 ← to_zeros(S, f)
return rank_1(L, to_zeros(i)) − rank_1(L, f_0) + is_leaf(S, f)

function LEAF_SELECT(S, l, i):
f ← level_first(S, l)                      ▷ The first vertex in the range
f_0 ← to_zeros(S, f)
o ← rank_1(L, f_0) − is_leaf(S, f)        ▷ Offset
return to_ones(S, select_1(L, o + i))

For the sake of completeness, we look at the other patterns:

"01"
The symbol one in this pattern corresponds to the beginning of a degree sequence of an inner vertex (it has at least one child).

"10"
The symbol zero in this pattern corresponds directly to an inner vertex; its degree sequence contains at least one zero.

"11"
It is part of a degree sequence of vertices which have at least two children.

3.3.4 Final Thoughts
The main problem of the LOUDS representation appears to stem from the non-locality of the near vertices. The unsupported operations are more concerned about ancestors and subtrees than siblings and levels. We developed a new index for the operation DEPTH and LEVEL_* which greatly expands the possibilities. Its simple implementation requiring only three types of indices (RANK, SELECT, BLOCK_OF_D) make it suitable for cases when there are not many requirements set on the supported operations.

3.4 Balanced Parentheses
In the previous section, we have seen that it is possible to build a succinct data structure for storing trees which requires essentially only the RANK and SELECT operations on the underlying bit string. Here we focus on a different schema which exploits the property that the two symbols associated with a vertex are correctly matched in the bit string. We represent them as parentheses in a balanced bit string, for which we utilize the MATCH and ENCLOSURE operations which we developed earlier in 2.2.1.

The representation is based on Depth-First Search and it follows two rules:

1. When a vertex is entered from its parent, the vertex is opened and an opening parenthesis is outputted in the representation.
2. When the algorithm returns from a vertex to its parent, the vertex is closed and a closing parenthesis is outputted.
Although the root vertex does not have a parent, it is opened and closed as if it had one.

**function BP_REPRESENTATION(R):**

\[ S \leftarrow [], \quad \text{push}(S, R) \quad \triangleright \text{The stack is initialized with the root of the tree.} \]

\[ \text{while } V \leftarrow \text{pop}(S): \]

\[ \text{if } V.\text{state} = \text{"UNSEEN"}: \]

\[ \text{push}(S, V) \]

\[ V.\text{state} \leftarrow \text{"OPEN"} \]

\[ \text{output("(")} \]

\[ \text{for all } C \in \text{reverse(children}(V)): \]

\[ \text{push}(S, C) \]

\[ \text{else if } V.\text{state} = \text{"OPEN"}: \]

\[ \text{All children has been processed.} \]

\[ V.\text{state} \leftarrow \text{"CLOSED"} \]

\[ \text{output(")")} \]

In the algorithm, the reverse is needed only to maintain the order of children. An alternative definition of this schema can be expressed recursively:

1. A leaf is encoded with a pair of parentheses.
2. An inner vertex is encoded as a concatenated string of encodings of its children surrounded by a pair of parentheses.

We can see from this definition that each subtree forms a continuous substring.

We associate each vertex with its opening parenthesis. By using the operation MATCH we can easily switch between these two.

**Example.** The BP representation of the tree in figure 3.1 including vertex association is:

\[ S = ((())(((())())())) \]

The spaces separate representations of children of the root \( a \).

The operations PRE_RANK, PRE_SELECT, POST_RANK, and POST_SELECT are immediately supported by opening-based and closing-based RANKS and SELECTs on the underlying bit string \( S \). We first show implementation of the operations which do not need any auxiliary index and which can be expressed using only the primitives provided by \( S \).

**function IS_ROOT(S, i):**

\[ \text{return } i = 0 \]

**function IS_LEAF(S, i):**

\[ \text{return } S[i + 1] = \text{"} \]

\[ \triangleright \text{Is the parenthesis immediately closed?} \]

### 3.4.1 Navigation Operations

**function PARENT(S, i):**

\[ \text{return } \text{enclose}(S, i) \]

Neither DEGREE nor CHILD_RANK nor CHILD_SELECT are available without an additional index; we therefore resort to the direct implementation of the rest of the CHILD_* operations:
function CHILD_FIRST(S, i):
    return i + 1

function CHILD_LAST(S, i):
    return find_open(find_close(S, i) - 1)

function CHILD_NEXT(S, i):
    return find_close(S, i) + 1

function CHILD_PREV(S, i):
    return find_open(S, i - 1)

3.4.2 Other Native Operations

The following natively supported operations use the property that a subtree of $v$ is fully contained in the representation of $v$.

function DEPTH(i):
    return paren_depth(i)

function IS_ANCESTOR(i₁, i₂):
    return $i_1 \leq i_2$ and find_close($i_1$) $\geq i_2$

function SUBTREE_SIZE(i):
    return rank((find_close(i) − rank(i) + 1)

The group of operations consisting of LCA, DISTANCE, DEEPEST_VERTEX, and HEIGHT can be solved by ENCLOSE and maximum range queries.

function LCA(i₁, i₂):
    if is_ancestor(i₁, i₂):
        return i₁
    else if is_ancestor(i₂, i₁):
        return i₂
    else:
        return enclose(i₁, i₂)

function DISTANCE(i₁, i₂):
    $a \leftarrow \text{lca}(i_1, i_2)$
    return $\text{depth}(i_1) + \text{depth}(i_2) − 2\text{depth}(a)$

function DEEPEST_VERTEX(i):
    return RMQi(i, match(i))

function HEIGHT(i):
    return $\text{depth}(\text{deepest_vertex}(i)) - \text{depth}(i)$
3.4.3 In-Order Rank and Select and Leaf Operations

An in-order number is assigned to a vertex for each two children in a row. This situation can be detected in the bit string \( S \) by searching for the pattern ")(" which means that one vertex ends and its right sibling starts. Each occurrence of the pattern is accounted as an in-order number being assigned to their parent.

We recall how the parentheses are actually stored: ")(" = 0. With that in our mind, we derive a bit string \( I \) which contains 1 at the position of the opening parenthesis which is preceded by a closing parenthesis: \( I[i] = [S[i] = 1 \text{ and } S[i - 1] = 0] \):

\[
I = \sim S & (S \gg 1).
\]

The operations \texttt{IN\_RANK} and \texttt{IN\_SELECT} are then defined on this derived bit string.

Note that the operation \texttt{IN\_RANK} returns the smallest assigned in-order number, and that \texttt{IN\_SELECT} find the vertex by any in-order number which it has been assigned. Also, we cannot use the operation \texttt{DEGREE} in the implementation since it is not supported.

\begin{verbatim}
function IN_RANK(S, i):
    if is_leaf(S, i) or child_first(S, i) = child_last(S, i):
        return -1
    else:
        return rank_1(I, child_first(S, i))

function IN_SELECT(S, r):
    s ← select_1(I, r)
    if s = N:  ▷ SELECT is out of range; we need an explicit check.
        return -1
    else:
        return parent(S, s)
\end{verbatim}

We use a similar technique as in 3.3.3 to support the leaf operations, this time restricted to a subtree rather than a level. A vertex is a leaf if its opening parenthesis is immediately closed. We are looking for a pattern "O", for which we derive a bit string \( L \) such that \( L[i] = [S[i] = 1 \text{ and } S[i + 1] = 0] \):

\[
L = S & (\sim S \ll 1).
\]

\begin{verbatim}
function LEAF_SIZE(S, a):
    if is_leaf(a):
        return 1
    else:
        return rank_1(L, find_close(a)) − rank_1(L, a)

function LEAF_RANK(S, a, i):  ▷ We assume i is in the subtree of a.
    if is_leaf(a):
        return 1
    else:
        return rank_1(L, i) − rank_1(L, a)
\end{verbatim}
function LEAF_SELECT(S, a, i):
    if is_leaf(a):
        return a
    else:
        return select₁(L, rank₁(L, a) + i)

We may be interested in the two other patterns:

"((" If we focus on the first symbol of the pattern, it corresponds to a vertex which has a child. We could therefore derive similar operations to LEAF_* which would be defined for inner vertices.

The second symbol corresponds to the first child of an inner vertex; which can also be obtained by CHILD_FIRST operation applied to its parent.

"))" Again, it can be used to query inner vertices or last children. It is less useful, since the same can be done by the previous pattern without the need of an extra call of FIND_OPEN.

3.4.4 Final Thoughts

Three more indices have been developed for the BP representation:

DEGREE
An index by [CLL05].

CHILD_RANK, CHILD_SELECT
An index by [LY08]. Together with the index for DEGREE, the indices complete the set of navigation operation and make the structure useful for a wider set of use cases.

LEVEL_ANCESTOR, LEVEL_PREV, LEVEL_NEXT
An index by [MRRR12].

We did not show these indices as they are rather complicated. We are going to show in the next chapter a different structure based on the same BP representation, which supports all these operations (including LEVEL_*) natively.

The operations which remain unsupported are: LEVEL_* operations (only LEVEL_PREV, LEVEL_NEXT are supported); DFUDS_RANK and DFUDS_SELECT (that is the numbering schema which is native to the representation that we introduce next).

Its main advantage of the BP representation is the simplicity and the support for many operations including LCA and the DEPTH, for which it pays by hard, yet not impossible, CHILD_RANK, CHILD_SELECT and DEGREE operations. If we restrict degrees of the vertices to constants, the BP representation suddenly becomes a very good choice.
3.5 Depth-First Unary Degree Sequence

The Depth-First Unary Degree Sequence (DFUDS) is an encoding which tries to combine the advantages from both LOUDS (child queries are natively supported) and BP (each subtree is stored locally instead of being spread throughout the encoding). It was originally proposed by [BDMR99].

The encoding is defined recursively:

1. A leaf is encoded as a closing parenthesis.
2. An inner vertex is encoded as degree opening parentheses followed by a single closing parenthesis (which we again call the degree sequence) and concatenated encodings of its children.

For convenience, one opening parenthesis is prepended to make the sequence of parentheses balanced.

function DFUDS_representation(R):

\( S \leftarrow [\,], \quad \text{push}(S,R) \quad \triangleright \text{The stack is initialized with the root of the tree.} \)

\( \text{output}("\)"), \text{while } V \leftarrow \text{pop}(S): \)

\( \text{for all } C \leftarrow \text{reverse}(\text{children}(V)): \)

\( \text{output}("\)"), \text{push}(S,C) \)

\( \text{output}(\)"

Lemma 11. The DFUDS encoding provides a balanced bit string.

Proof. When we prepend an opening parenthesis in front of an encoding of any subtree, we obtain a balanced bit string.

The proof goes by induction on height of a subtree. The claim holds for a leaf, which is only a closing parenthesis, which misses its matching opening one.

We assume, that the claim holds for all \( d = \text{degree}(v) \) subtrees \( T_1, \ldots, T_d \) of a vertex \( v \). The degree sequence of the vertex \( v \) contains \( d \) opening and one closing parenthesis. The closing one matches the last opening one within the degree sequence, and \( d - 1 \) of them are left to be matched with the excessive parentheses in the subtrees. All parentheses in \( T_1, \ldots, T_{d-1} \) get matched, leaving one closing parenthesis in \( T_d \) unmatched, which is the one that the claim requires.

The only unmatched parenthesis of the representation of the whole tree is matched in the end with the artificially prepended one.

Alternative, we can associate it with the terminal zero of its degree sequence (we will indeed define functions that switch between them). The association with the opening parenthesis in the degree sequence of its parent (similar to LOUDS) is also possible as we show later by defining \( \text{dfuds_rank} \) and \( \text{dfuds_select} \).

Example. The DFUDS representation of the tree in figure 3.1 including vertex association is:

\[ S = \left( \left( \left( ( ( ( a \ b \ c \ f \ g \ i \ d \ e \ h ) ) ( ) \ ) \ ) \ ) \right) \right) \]

The spaces separate the unary degree sequences of the vertices.
### 3.5.1 Navigation Operations

It is easy to test whether a vertex is the root or a leaf.

**Function** `IS_ROOT(S, i)`:
- `return i = 1`  
  ▷ Mind the artificially prepended parenthesis.

**Function** `IS_LEAF(S, i)`:
- `return S[i] = "\)"`  
  ▷ Is the parenthesis a closing one?

The crucial observations for the following operations are:

1. The representation of a vertex is preceded by a closing parenthesis with the exception of the root.
2. The opening parenthesis in the degree sequence matches the closing parenthesis immediately preceding the representation of a child.
3. In a subtree of a vertex \( v \), the last closing parenthesis, which represents the last leaf, matches a parenthesis in the degree sequence of the parent of \( v \).

The observations follow from the same induction as in lemma [11](#).

We first define two helper functions which navigate from any position in a vertex degree sequence to its beginning (which is the symbol associated with the vertex) and to its end (which is always a closing parenthesis). We also define a function which navigates to the last parenthesis of the vertex representation.

**Function** `TO_BEGINNING(S, p)`:
- `if p = 0:`
  - `return −1`  
    ▷ The representation of the root starts at position 1.
- `else if prev\(_1\)(S, p) < 0`:
  - `return 1`  
    ▷ Correction for the root
- `else`:
  - `return prev\(_1\)(S, p) + 1`

**Function** `TO_END(S, p)`:
- `if p = 0:`
  - `return −1`
- `else`:
  - `return succ\(_1\)(S, p)`

**Function** `TO_LAST(S, p)`:
- `if is_leaf(S, p)`:  
  - `return p`
- `else`:
  - `return find_close(S, enclose(S, to_beginning(S, p)))`

The navigation operations can be immediately defined.

**Function** `PARENT(S, i)`:
- `p ← find_open(S, i − 1)`  
  ▷ Inside parent’s degree sequence
- `return to_beginning(S, p)`

**Function** `DEGREE(S, i)`:
- `return to_end(S, i) − i`
function CHILD_RANK(S, i):
    p ← find_open(S, i - 1)
    return to_end(S, p) - p

function CHILD_SELECT(S, i, k):
    p ← to_end(S, i) - k
    return find_close(S, p) + 1

3.5.2 Other Native Operations

function PRE_RANK(S, i):
    return rank(S, to_end(S, i))

function PRE_SELECT(S, r):
    return to_beginning(S, select(S, i))

In order to support DFUDS_RANK and DFUDS_SELECT, we will need another helper function TO_SYMMETRIC which navigates from k-th opening parenthesis in the degree sequence of vertex v to \((\text{degree}(v) - k - 1)\)-th one.

function TO_SYMMETRIC(S, p):
    b ← to_beginning(S, p)
    return b + degree(S, b) - (p - b) - 1

function DFUDS_RANK(S, i):
    if is_root(S, i):
        return 1
    else:
        p ← find_open(S, i - 1)
        return rank(S, to_symmetric(S, p))

function DFUDS_SELECT(S, r):
    if r = 1:
        return 1
    else:
        p ← to_symmetric(S, select(S, i))
        return find_close(S, p) + 1

And finally there are two simple subtree-oriented operations which use the helper function TO_LAST.

function IS_ANCESTOR(i_1, i_2):
    if is_leaf(i_1):
        return i_1 = j_2
    else:
        return i_1 ≤ i_2 ≤ to_last(S, i_1)

function SUBTREE_SIZE(i):
    if is_leaf(S, i):
        return 1
    else:
        return rank(S, to_last(S, i)) - rank(S, i)
3.5.3 Lowest Common Ancestor

The operation LCA is surprisingly similar to the one for BP 3.4.2, however it needs to be shown. The proof is by [Fis10].

Let’s assume that neither \( i_1 \) nor \( i_2 \) is an ancestor of the other one; in that case \( \min(i_1, i_2) \) would be the answer. We further assume that \( i_1 < i_2 \) and that their lowest common ancestor is \( j = \text{lca}(i_1, i_2) \). The vertex \( j \) is the smaller from both \( i_1, i_2 \) because it is assigned a lower pre-order number than any of its descendants. We call (like in section [2.2.3]) \( p_1 < p_2 \) the distinct children of \( j \) whose subtrees contain \( i_1 \) and \( i_2 \).

We look at excess values of children of \( j \). Each child and its subtree forms a sequence of matching parentheses with the exception of the last one, which is a closing parenthesis and stays unmatched (lemma [11]). From this, it follows that \( \text{rmqi}(S, \text{to}_\text{last}(S, i)) = \text{to}_\text{last}(S, i) \).

We look into the properties of \( \text{rmqi}(S, i_1, i_2 - 1) \). The last parenthesis of the representation of a child \( c \) of the vertex \( j \) which is contained in the range sets a new minimum. There may be an issue with the subtree of \( p_2 \), which is not fully contained in the range. If it contained the minimum elsewhere, then we cannot simply navigate to the vertex \( j \).

It cannot be the last parenthesis of \( p_2 \), because we end the interval just before it. It also will not be any other parenthesis \( p \geq p_2 \):

\[
\begin{align*}
\text{excess}(S, \text{to}_\text{last}(S, p_2)) + 1 \\
= \text{excess}(S, \text{to}_\text{last}(S, \text{child_prev}(S, p_2)))
\end{align*}
\]

\[
\leq \text{excess}(S, p).
\]

In case of equality, the RMQI returns the leftmost occurrence.

The \( \text{rmqi}(S, i_1, i_2 - 1) \) returns position of the last parenthesis of the previous sibling of \( p_2 \) and therefore \( p_2 = \text{rmqi}(S, i_1, i_2 - 1) + 1 \). The result is then \( \text{lca}(S, i_1, i_2) = j = \text{parent}(S, p_2) \).

\[
\text{function } \text{LCA}(i_1, i_2): \\
\quad \text{if } \text{is_ancestor}(i_1, i_2): \\
\quad \quad \text{return } i_1 \\
\quad \quad \text{else if } \text{is_ancestor}(i_2, i_1): \\
\quad \quad \quad \text{return } i_2 \\
\quad \quad \text{else:} \\
\quad \quad \quad p_2 \leftarrow \text{rmqi}(i_1, i_2 - 1) + 1 \\
\quad \quad \quad \text{return } \text{parent}(p_2)
\]

3.5.4 Leaf operations

The operations on leaves can be defined similarly to those for the LOUDS and BP representation. We are looking for the second of two consecutive closing parentheses – ")")". This follows from the property that the degree sequence of every vertex is preceded by a closing parenthesis, and the degree sequence of leaves is only the closing parenthesis. Note that this does not work in case when the tree root is a leaf, however this special case is easy to handle, too.

The other patterns are the same as in case of the LOUDS representation in section [3.3.3].
3.5.5 Final Thoughts

Two indices were proposed to extend the set of supported operations beyond what we have shown.

**DEPT**
An index by [JSS12]. Supporting the operation DEPT makes also possible to query DISTANCE between vertices.

**LEVEL_ANCESTOR**
An index also by [JSS12].

DFUDS is the opposite of BP: it is less intuitive, it supports all the basic navigation queries, however it lacks an easy support for the DEPT operation. It is a good choice if the PRE_RANK and CHILD_* operations are important for the use case.
4. Advanced Data Structures

In the previous chapter, we showed how three different bit string encodings of ordinal trees can be turned into data structures which support wide varieties of operations, provided that they are equipped with the RANK and SELECT (and later MATCH and ENCLOSURE) indices. Here we present data structures which are fundamentally different in either the primitive operations they use, or that they require more advanced encoding instead of being based on a single bit string.

**Fully-Functional (FF)**

A data structure built on top of the BP representation which replaces the set of traditional indices by new more general index. This results in a structure which supports not only all operations which BP did – the old primitive operations are special cases of the new ones, but also allows to implement those operations which either required a specialized index, or they simply were not possible.

**Tree Covering (TC)**

Tree covering is a data structure based on a recursive decomposition of the tree into mini-trees and micro-trees which are then encoded. Each level of the decomposition is stored and processed in a different way. Since it is tree structure rather than bit string oriented, it is easier to augment it with small pieces of information which are required by various operations. This is a big difference from (1) discovering, more than inventing, the inner rules of the representations, which we did with ranking and selecting; (2) or developing general indices which cover the bit string in various blocks on multiple levels.

**Universal Succinct Representation (USR)**

An attempt to develop a universal data structure which diminishes the differences between several representations. It provides a view to the BP and DFUDS bit string representation as well as decomposition into micro-trees. Its main advantage is that it automatically benefits from any indices which are proposed for either BP, DFUDS, or TC representations.

4.1 Fully Functional Representation

The BP representation which we described in the previous chapter was based on using RANK and SELECT, and MATCH and ENCLOSURE operations as the main tool. There were several operations which required an additional index (CHILD_* and LEVEL_ANCESTOR), and several operations which were not supported at all (LEVEL_*). All the operations which we have just listed have one thing in common: they can all be expressed using more general operations. We did not show the indices for LEVEL_ANCESTOR in the BP and DFUDS representations for a reason: they were unnecessarily complicated, whereas here in the Fully Functional data structure, it is going to be one of the easiest operations.

The data structure was developed by [SN10]. The name Fully Functional refers to the set of operations, that the structure offers full range of functionality.
We define a new set of primitive operations for which we then design a single index, incorporating even the range operation. The index will be general – any of the representations (LOUDS, BP, DFUDS) can use it, however only BP will benefit from it.

4.1.1 New Operations

We follow on the section 2.2.3 and define more operations on the bit string $S$ parametrized by the function $g$ which is to be applied.

The sum operation

Since RANK will no longer be a primitive operation, we cannot use it to compute $G[i]$. It is therefore replaced by an operation SUM which is equivalent to $G[j] - G[i - 1]$, which we used before in 2.2.3. It is defined as:

$$\text{sum}(S, g, i, j) = \sum_{k=i}^{j} g(S[k]).$$

Linear search operations

We replace SELECT in all forms by linear searching operations.

FWD SEARCH

Returns the first position $j$ after a given position $i$ such that the values sum up to $d$:

$$\text{fwd\_search}(S, g, i, d) = \min_{j > i} \{ j : \text{sum}(S, g, i, j) = d \}.$$

BWD SEARCH

Works the same way as FWD SEARCH except that it searches towards the beginning of the bit string:

$$\text{bwd\_search}(S, g, i, d) = \max_{j < i} \{ j : \text{sum}(S, g, j, i) = d \}.$$

Note that despite having used the same names in section 2.2.1, their definition here is different. They refer to operations defined for all $i \in [0, N)$, rather than only to look-up tables.

Range operations

We generalize the range operations for any function $g$. We also extend the original operation RMQI into RMQ_SELECT, so that it does not only return the first occurrence of the minimum, but any occurrence in general.

RMQ

Returns the value of the minimum in the given range.

$$\text{rmq}(S, g, i, j) = \text{sum}(S, g, 0, i - 1) + \min_{i \leq k \leq j} \text{sum}(S, g, i, k)$$

RMQ_SIZE

Returns how many occurrences of minimum there are in the given range:

$$\text{rmq\_size}(S, g, i, j) = |\{i \leq k \leq j : \text{sum}(S, g, 0, k) = \text{rmq}(S, g, i, j)\}|.$$
RMQ_RANK
Returns how many occurrences of minimum there are in the given range up to position \( r \). This can be directly implemented using RMQ and RMQ_SIZE:

\[
\text{function RMQ_RANK}(S, g, i, j, r):
\]
\[
\text{if } \text{rmq}(S, g, i, j) \neq \text{rmq}(S, g, i, r):
\]
\[
\text{return } 0
\]
\[
\text{else}:
\]
\[
\text{return } \text{rmq} \_\text{size}(S, g, i, r)
\]

RMQ_SELECT
Returns the position \( p \) of the \( r \)-th minimum in the range \( i, j \):

\[
\text{rmq} \_\text{select}(S, g, i, j, r) = \min_{p \geq i} \{ p : \text{rmq} \_\text{rank}(S, g, i, j, p) \geq r \}.
\]

RMQ, RMQ_SIZE, RMQ_RANK, RMQ_SELECT
The maximum variants of the range operations are defined similarly.

We show how it is possible to realize all operations which we used as primitives in the previous succinct data structures only by sums, searches and range operations on calculated arrays. We immediately get a data structure equivalent to those in the previous chapter in terms of the operations which they support. Later we will show that even more operations become feasible for the FF representation.

We recall the definitions of the functions \( \varphi, \psi, \pi \) from the section 2.2.3:

\[
\varphi(b) = b, \\
\psi(b) = 1 - b, \\
\pi(b) = 2b - 1.
\]

Operations on general bit strings

\[
\text{rank}_1(i) = \text{sum}(S, \varphi, 0, i) \\
\text{select}_1(n) = \text{fwd} \_\text{search}(S, \varphi, 0, n) \\
\text{rank}_0(i) = \text{sum}(S, \psi, 0, i) \\
\text{select}_0(n) = \text{fwd} \_\text{search}(S, \psi, 0, n)
\]

Operations on balanced bit strings

\[
\text{find} \_\text{close}(i) = \text{fwd} \_\text{search}(S, \pi, i, 0) \\
\text{find} \_\text{open}(i) = \text{bwd} \_\text{search}(S, \pi, i, 0) \\
\text{enclose}(i) = \text{bwd} \_\text{search}(S, \pi, i, 2) \\
\text{enclose}(i_1, i_2) \text{ stays the same} \\
\text{rmqi}(E, i, j) = \text{rmq} \_\text{select}(S, \pi, i, j, 1) \\
\text{RMQi}(E, i, j) = \text{RMQ} \_\text{select}(S, \pi, i, j, 1)
\]

Therefore, while designing the index, we can focus only on the new operations.
4.1.2 The Succinct Index

As we are describing a universal index which is independent of its specific usage for representing trees, we again use \( N \) to denote the size of the bit string. We also parametrize all operations by the function \( g \) which will be fixed for the rest of this section.

The structure of the index follows the general idea of dividing into blocks and then into small blocks. Here the blocks will be of roughly polylogarithmic size and small blocks of size \( b = \frac{\log N}{2} \) in order to be processed by using look-up tables. If the query spans multiple small blocks within a single block, the queries are handled by min-max trees. The queries spanning multiple blocks are answered by a macro structure.

The small blocks are small enough to be used as indices of precomputed look-up tables, which answer all queries in constant time. We use the following look-up tables which directly implement the primitive operations on small blocks:

**SUM**
For a given range \( i, j \), it returns the sum \( v \).

**FWD\_SEARCH, BWD\_SEARCH**
For a given position \( i \) and value \( d \), they return the first next and previous position \( p \) where the difference is \( d \). If the answer is not present, a special value is returned.

**RMQ, RMQ**
For a given range \( i, j \), they return the local value \( v \) of minimum and maximum.

**RMQ\_SIZE, RMQ\_SIZE**
For a given range \( i, j \), they return the number \( r \) of occurrences of their minimum and maximum.

**RMQ\_SELECT, RMQ\_SELECT**
For a given range \( i, j \) and \( r \), they return the position \( p \) of \( r \)-th minimum and maximum in the small block.

The values of \( i, j, r \) are in range \([0, b-1]\); the values \( d, v \) are in range \([-b, b]\); the value of \( p \) is in range \([0, b-1]\) plus the special value indicating that such position does not exist in the small block. Together we have 9 precomputed tables; each of them requires \( O(\sqrt{N} \log N \log \log N) \) bits of memory.

The tables are used to answer all range operations whenever \( i \) and \( j \) are in the same small block. In case of the searches, they also determine that the answer is not present in the small block.

We also use the look-up tables to handle the prefix and suffix of a range query, or at the beginning or the end of a search query.
4.1.3 Min-Max Tree

Each block contains \( k^c \) small blocks for \( k = \frac{\log N}{\log \log N} \) and an arbitrary integer constant \( c \geq 1 \). The value of the constant \( c \) will be discussed at the end. The purpose of the min-max tree is to lift the operations from \( b \) bits to \( bk^c \) while retaining their constant running time.

The \( l \)-th block covers an interval of \( B = bk^c \) bits spanning from the position \( lB \) to \( (l+1)B - 1 \). To simplify the description we isolate the block by shifting the offsets so that it covers the range \([0, B - 1]\) by setting \( l = 0 \). We also assume that the values in the array \( G \) are in range \([-B, B]\); the global values can be computed simply by adding the value \( G[lB - 1] \). This does not have any impact on the correctness of the operations on the block level.

We extend the bit string \( S \) to the nearest multiple of \( B \) in order to ensure that each block is full. The extra space caused by the extension is negligible. Only the \textsc{fwd-search} operation could return an answer from the extended part, however it can happen only if the answer was not found earlier – this case is easy to handle in the end.

We build a perfect \( k \)-ary tree on top of the sequence of the small blocks (belonging to the block); we call it the \textit{min-max tree}. The tree has a constant height equal to \( c \). Its leaves represent the information from the small blocks provided by the look-up tables; the inner nodes aggregate the information from their children. We store the following values in each node which spans over the range \( i, j \):

\[
\begin{align*}
e &= \text{sum}(S, g, 0, j) \\
&\quad \text{The value at the end of the range.} \\
m &= \text{rmq}(S, g, i, j), ms = \text{rmq}_\text{size}(S, g, i, j) \\
&\quad \text{The minimum value and how many times it occurs.} \\
M &= \text{RMQ}(S, g, i, j), Ms = \text{RMQ}_\text{size}(S, g, i, j) \\
&\quad \text{The maximum value and how many times it occurs.}
\end{align*}
\]

The aggregation in the inner node is the obvious one: the last value for \( e \); minimum for \( m \), maximum for \( M \); sum for \( ms \) and \( Ms \). All values within a node require only \( O(\log b) \) bits each.

As it is a perfect \( k \)-ary tree, we store the values of the nodes in a heap-like fashion in arrays \( e[] \), \( m[] \), \( ms[] \), \( M[] \), \( Ms[] \). We can navigate in this tree from an node with number \( n \) to the parent (\( \lfloor n/k \rfloor \)), first child \( n \cdot k \), last child \( n \cdot k + k - 1 \), previous sibling \( n - 1 \), next sibling \( n + 1 \).

All children of a node are stored together in consecutive \( O(k \log \log N) = O(\log N) = c'b \) bits of memory, for a constant \( c' \). There are \( \frac{k^{c+1}}{c'} = O(k^c) \) nodes, each of them requires \( O(\log b) \) bits. The density of the tree structure built on top of the sequence of the small blocks is \( O \left( \frac{k^c \log b}{k^c b} \right) = O \left( \frac{\log b}{b} \right) = o(1) \).

We show how to support the operations on the block level using traversal of the min-max tree and possibly delegation to small blocks. By \( b(i) \) we denote the index of the small block which contains the position \( i \). Contrary to the defined numbering of nodes, in order to simplify the notation, we assume in the algorithms that the node numbers of leaves are the same as indices of the small blocks with
which they are associated. We will use subscripts to refer to bits of $S$ represented by a small block $n$: $S_n = S^{[nb : (n+1)b]$

```plaintext
function SUM_BLOCK(S, g, i, j):
    x ← e[b(i − 1) − 1] + sum[Sb(i−1), g, 0, (i − 1) % b]
    y ← e[b(j) − 1] + sum[Sb(j), g, 0, j % b]
    return y − x
```

**Search Operations**

We say that the small block $j$ covers a value $v$ if $m[j] \leq v \leq M[j]$.

**Lemma 12.** The answer to a search query starting at $i$ and looking for a difference $d$ (such that it is not answered in the small block $b(i)$) is in the first small block $j$ such that covers $v = \text{sum}(S, g, 0, i) + d$.

No small block in the subtree of a min-max tree node not covering the value $v$ contains the answer.

**Proof.** Let us recall an important property of the $\pm 1$ functions $g$: $|G[i] − G[i − 1]| \leq 1$.

From that it follows that each small block contains all values between its minimum and maximum. The first small block which covers the desired value $v$ has to contain it.

Moreover, if $v < \text{rmq}[S_{b(i)}, (i + 1) \% b, b − 1]$ (less than minimum of the rest of the small block), then it is sufficient to find the first small block $j$ such that $m[j] \geq v$ because $M[j] \geq m[j − 1]$. A similar statement holds for $v$ being greater than the maximum.

The extremes of a leaf are the same as of the small block with which it is associated. If a small block $j$ contains the answer, then the value $v$ is covered by the leaf $j$. Because of the aggregation method of inner vertices, all ancestors of $j$ cover $v$ too. An inner vertex covers $v$ only if a leaf in its subtree covers $v$.  

When we process the search operations, we first check the small block containing $i$ for an answer. If the answer is present there, we return the position, and stop.

Otherwise, we compute the desired value $v$ and traverse the tree up until we find the answer or until we reach the root. If we get to root, we signal that this block does not contain the answer by returning a special value. When we are in a node $a$, we check the extremes of its siblings in the direction in which we are searching. This check can be performed in time $c'$ using a precomputed look-up table as all siblings are stored in consecutive $c'b$ bits. Only one-sided checks are necessary as we noted in the lemma 12.

When we find a sibling covering $v$, we move to it and start descending to its first child covering the value $v$. Finding such child again requires time $c'$ and another set of precomputed tables. Once we descend to a leaf $j$, we compute the value $d' = v − e[j − 1]$ which we are going to look for in the associated small block using a look-up table. As the height of the tree is $c$, the whole operation takes at most $O(2cc') = O(1)$ steps.
We show the pseudocode for the operation FWD_SEARCH. By an asterisk superscript we denote a repeated application of a look-up table on the list of consecutive nodes in the tree.

```plaintext
function FWD_SEARCH_BLOCK(S, g, i, d):
  p ← fwd_search[Sb(i), g, i % b, d]
  if p ≠ −1:
    return p + b(i)b
  else:
    v ← e[b(i) − 1] + sum[Sb(i), g, 0, i % b] + d
    n ← b(i)  ▷ The current node, initialized to be a leaf
    while (j ← node_search*[right_siblings(n), v]) = −1:   ▷ Sibling search
      n ← parent(n)
      if is_root(n):
        return −1
    while not is_leaf(j):
      j ← node_search*[children(j), v]  ▷ The first child covering v
      d' ← v − e[j − 1]  ▷ The remaining difference
    return jb + fwd_search′(Sj, g, 0, d')  ▷ d' in the small block j
```

Note that the final FWD_SEARCH′ is the standard FWD_SEARCH altered to allow the answer 0, which is prohibited by the original definition. This can be handled with a simple check for $\text{sum}[S_j, g, 0, 0] = d'$.

**Range Operations**

The range operations work similarly: they traverse the tree up gathering information on the way, and in case of range select operations, they descend down. If the query is fully contained within a small block, the answer is found in a look-up table. In order to solve the other queries, we use two helper functions. By the same argument as for the search operations, the running time is $O(2cc′) = O(1)$.

The helper function RMQ_INFO which returns aggregated information for all siblings of a given node. The function returns a tuple containing: the number of the first node, the number of the last node, the minimum, the maximum, the number of occurrences of the minimum, the number of occurrences of the maximum. It iterates over $\frac{c}{2}$ nodes in each step using look-up tables, which results in the running time $O(c′)$.

Another helper function RMQ_LIST gathers all aggregated info for subranges between $i$ and $j$. It starts with the leaves $u = b(i)$ and $v = b(j)$, and keeps two lists of partial answers, one for each starting point. We first ask the small blocks for their answers and remember them in their respective lists.

In a loop until $u$ and $v$ merge, we get the answers from the right siblings of $u$ and the left siblings of $v$, we remember them in the lists and move to their parents. In the last iteration the left siblings and the right siblings overlap; therefore we store the answers in only one of the lists. Each list contains at most $c$ values as the table look-ups aggregate the answers. We combine the lists into a single list and find the answer to the queries in it.
function RMQ_LIST(S, g, i, j):
    L ← [], R ← []  \( \triangleright b(i) \neq b(j) \)
    \( \triangleright \) Initialization of the empty lists
    \( i_m \leftarrow \text{rmq}[S_{b(i)}, g, i \% b, b - 1] + e[b(i) - 1] \)
    \( \triangleright \) Similarly  \( j_m, i_M, j_M \)
    \( i_{ms} \leftarrow \text{rmq}_\text{size}[S_{b(i)}, g, i \% b, b - 1] \)
    \( \triangleright \) Similarly  \( j_{ms}, i_{Ms}, j_{Ms} \)
    append(L, \{ f: b(i), l: b(i), m: i_m, ms: i_{ms}, M: i_M, Ms: i_{Ms} \})  \( \triangleright \) Info for  \( b(i) \)
    prepend(R, \{ f: b(j), l: b(j), m: j_m, ms: j_{ms}, M: j_M, Ms: j_{Ms} \})  \( \triangleright \) Info for  \( b(j) \)
    while parent(u) \( \neq \) parent(v):
        \( \triangleright u, v \) in different subtrees
        append(L, \text{rmq}_\text{info}(S, g, u, "r"))
        prepend(R, \text{rmq}_\text{info}(S, g, v, "l"))
    append(L, \text{rmq}_\text{info}(S, g, u, v))  \( \triangleright \) Between  \( u \) and  \( v \)
    concatenate(L, R)
    return L

For RMQ we return the minimum of the list obtained from RMQ_LIST.

function RMQ_BLOCK(S, g, i, j):
    if  \( b(i) = b(j) \):
        return \( \text{rmq}[S_{b(i)}, g, i \% b, j \% b] + e[b(i) - 1] \)
    else:
        \( m \leftarrow \infty \)
        for all  \( I \leftarrow L: \)
            \( m \leftarrow \min(m, I.m) \)
        return \( m \)

For RMQ_SIZE_BLOCK we sum  \( ms \) for those info-nodes whose minimum is equal to the global one calculated by RMQ.

function RMQ_SIZE_BLOCK(S, g, i, j):
    if  \( b(i) = b(j) \):
        return \( \text{rmq}_\text{size}[S_{b(i)}, g, i \% b, j \% b] \)
    else:
        \( m \leftarrow \text{rmq}_\text{block}(G, i, j) \)
        \( L \leftarrow \text{rmq}_\text{list}(G, i, j) \)
        \( ms \leftarrow 0 \)
        for all  \( I \leftarrow L: \)
            if  \( I.m = m: \)
                \( ms \leftarrow ms + I.ms \)
        return \( ms \)

In case of RMQ_SELECT, we find the first node  \( n \) in the list such that the prefix sum of  \( ms \) is greater than or equal to  \( r \). We descend in such node into the first child whose left siblings including itself sum up to  \( r \), and repeat until we get to a leaf. Once we are in a leaf, we use a look-up table to solve the select there.
function RMQ_SELECT_BLOCK(S, g, i, j, r):
   \[ \triangleq \text{Assuming } r\text{-th min exists} \]
   if \( b(i) = b(j) \):
      \( p \leftarrow \text{rmq_select}[S_{b(i)}, g, i \% b, j \% b, r] \)
      return \( b(i)b + p \)
   else:
      \( m \leftarrow \text{rmq_block}(G, i, j) \)
      \( L \leftarrow \text{rmq_list}(G, i, j) \)
      \( ms \leftarrow 0 \)
      for all \( I \leftarrow L \):
         if \( I.m = m \):
            if \( ms + I.ms \geq r \):
               break
            \( ms \leftarrow ms + I.ms \)

When the loop is broken, \( I \) contains the information about list of siblings node with one of them containing the desired minimum. Using look-up tables which are similar to NODE_SEARCH, we descend into the right child. The table also returns the new remaining number of occurrences by subtracting those in nodes before \( p \).

We first search the restricted range of nodes provided by \( I \), keeping track of the number of the occurrences which we are interested in.

\[
(p, r) \leftarrow \text{min_search}^*[\text{node_range}(I.f, I.l), r - I.ms] \\
\text{while not is_leaf}(p): \\
   (p, r) \leftarrow \text{min_search}^*[\text{children}(p), r] \\
   x \leftarrow \text{max}(i, pb) \% b, \quad y \leftarrow \text{min}(j, (p + 1)b - 1) \% b \\
\text{return } b(p)b + \text{rmq_select}[S_p, g, x, y] 
\]

There are seven look-up tables necessary for traversal and processing the tree. The tables have an additional (not mentioned) parameter which restricts the size of the block to be processed. Whenever an index of a node is returned, it is turned into a node number by adding the number of the leftmost node involved in the query.

**FWD_SEARCH**

The table is looking for the node which contains the first occurrence of \( v \) in the block of consecutive nodes. BWD_SEARCH requires another table returning the position of the last occurrence \( v \).

**RMQ_INFO**

Four tables aggregating information for consecutive nodes are necessary; the tables compute minimum, maximum, and sum conditioned by the value of minimum or maximum of the other nodes in the given range.

**RMQ_SELECT**

One table which is looking for the \( r \)-th occurrence of the minimum. It returns the index of the node and the number of occurrences of the minimum in its preceding siblings. RMQ_SELECT requires a similar table for occurrences of the maximum.

All tables are defined for blocks of \( b \) bits with \( O(1) \) parameters of size \( O(\log k^c) = O(\log \log N) \).
4.1.4 Macro Structure

The macro structure starts with five arrays which summarize the results from the underlying blocks (the root nodes of their min-max trees).

\( e[i] \)  
The value at the end of the block \( i \); in comparison to the block level, here we store the absolute values (the same for minima and maxima). Each element has size of \( O(\log N) \) bits.

\( m[i], M[i] \)  
The minimum and maximum values of the block \( i \).

\( ms[i], Ms[i] \)  
The number of their occurrences in the block \( i \). Each element is of size \( O(\log \log N) \) bits.

We denote \( B(i) \) to the index of the block containing the position \( i \).

The algorithm for \( \text{sum} \) is very similar to \( \text{sum\_block} \).

\[
\text{function } \text{sum}(S,g,i,j) :
\begin{align*}
x & \leftarrow e[B(i-1) - 1] + \text{sum\_block}(S_{B(i-1)}, g, 0, (i-1) \% B) \\
y & \leftarrow e[B(j) - 1] + \text{sum\_block}(S_{B(j)}, g, 0, j \% B) \\
\text{return} y - x
\end{align*}
\]

Search Operations

For each block \( i \) we define an array of left-to-right minima \( lrm_i \) such that

\[
\begin{align*}
lrm_i[0] & = i \\
lrm_i[j+1] & = \min\{k : k > lrm_i[j] \text{ and } m[k] > m[lrm_i[k]]\}
\end{align*}
\]

Similarly we define left-to-right maxima arrays \( LRM_i \), and for backward searching we define the arrays \( rlm_i \) and \( RLM_i \). We focus only on left-to-right direction.

The arrays \( lrm \) and \( LRM \) can be used to implement the global operation \( \text{fwd\_search}(S, g, i, d) \). Let’s assume that the result is not in block \( B(i) \), if it was, we can solve it on the block level. We are looking for a block \( j \geq B(i) + 1 = n \) such that \( j \) covers the value \( v = \text{sum}(S, g, 0, i) + d \). There are three possibilities of how to find the block \( j \):

1. If \( n \) covers \( v \), then the first block after \( B(i) \) contains the answer.
2. If \( v < m[n] \), we search for the block in the array \( lrm \).
3. Otherwise, we search in the array \( LRM \).

It can happen that the value \( v \) is not covered by any block \( j \), in which case we report a failure.
function FWD_SEARCH(S, g, i, d):
p ← fwd_search_block(S_B(i), g, i % B, d)  ▷ Block operation offset
if p ≠ −1:
    return B(i)B + p
else:
v ← sum(S, g, 0, i) + d
n ← B(i) + 1
if v < m[n]:
j ← lrm_search(n, v)
else if v > M[j]:
j ← LRM_search(n, v)
else:
j ← n
if j = −1:  ▷ A search reported that v is not covered.
    return −1
else:
d′ ← v − sum(S, g, 0, B(j)B − 1)
return B(j)B + fwd_search_block′(S_B(j), g, 0, d′)

Note that the final call of FWD_SEARCH_BLOCK’ is the standard block operation FWD_SEARCH_BLOCK altered to allow the answer 0. The same was necessary for the search on the block level.

It remains to show a constant time algorithm for the operation LRM_SEARCH (LRM_SEARCH is similar). The naïve approach would be to store each lrm_i as an array A turned into a compressed array using operation succ. The array A is defined partially for all j in lrm_i: A[M[i] − m[j]] = j. The problem with this approach is that, the compressed arrays for all i can contain \( O\left(\left(\frac{N}{B}\right)^2\right) \) runs in total, which is too much.

We observe that the arrays lrm_i are alike.

Lemma 13. Let a block a be in two different arrays lrm_i and lrm_j:

\[ lrm_i[x_i] = a = lrm_j[x_j], \]

then all following values in the arrays are the same:

\[ lrm_i[x_i + k] = lrm_j[x_j + k], \quad \forall k \geq 0. \]

Proof. It follows from the definition of the array lrm.

We define a tree \( T_{lrm} \) which is built as a trie for reversed lrm arrays. We add an artificial root, in order to make it a tree; it will not be used in algorithms.

Example. Let \( m = [6, 4, 9, 7, 4, 1, 8, 5] \), then the arrays lrm_i are:

\[
\begin{align*}
lrm_0 &= [0, 1, 6] & lrm_1 &= [1, 6] & lrm_2 &= [2, 3, 4, 6] \\
\end{align*}
\]

The tree \( T_{lrm} \) built from the arrays is shown in the figure 4.1. The values of m are in parentheses.
The properties of the tree are:

1. \( m[\text{parent}(i)] < m[i] \leq m[\text{parent}(i)] + B \);
2. \( i < \text{parent}(i) \);
3. the tree has \( \frac{N}{B} + 1 \) nodes.

The search for the value \( v \) in \( lrm_i \) is transformed to a search for an ancestor \( j \) of \( i \) in the tree \( T_{lrm} \) such that \( m[j] \leq v \). We split the search into two parts:

1. a search in \( lrm_i \) restricted to powers of two;
2. a search in \( lrm_i \) with a bounded distance.

We use the \( (O(n \log n); O(1)) \) algorithm for querying level ancestor introduced in [BFC04]. Instead of searching through the whole array \( lrm_i \), we limit the number of elements to \( \log N \):

\[
lrm'_i[j] = lrm_i[2^j].
\]

This array is turned into a tiny compressed array of jumps \( J_i \) requiring only \( O(\log^2 N) \) bits of space, which is \( O\left(\frac{N}{B} \log^2 N\right) = o(N) \) for \( c \geq 2 \).

By using the tiny compressed array, we move from node \( i \) to a node \( j' \) of \( T_{lrm} \) for which we bounds on depth and height, and the number of ancestors which we have to search through.

\[
\begin{align*}
\text{depth}(j') - \text{depth}(j) &< \text{depth}(i) - \text{depth}(j') \\
\text{height}(j') &\geq \text{depth}(i) - \text{depth}(j') \\
\text{depth}(j') - \text{depth}(j) &< \text{height}(j')
\end{align*}
\]

Note that if we tried to be more clever and represent nodes \( j \) which cover values \( m[i] - m[j] \) instead of those at distance \( 2^j \), then the bounds would not hold.

The tree gets decomposed iteratively to paths; in each step the longest path \( p \) from a node \( \text{start}(p) \) to a leaf \( \text{end}(p) \) is removed. If the path contains \( l = \text{length}(p) \) nodes, we prepend it with \( l \) more ancestors (or less if the root is reached) and call it a ladder. There are as many ladders as leaves of the tree \( T_{lrm} \), and all ladders together contain \( \leq 2^\frac{N}{B} \) nodes. Note that the root of the tree \( T_{lrm} \) is not represented in the ladders.
The previous bound guarantees that the final answer \( j \) is in the ladder which was extended from a path containing the node \( j' \), to which we got by using the tiny compressed dictionary \( J_i \). We represent each ladder by an array of block indices (which are nodes of \( T_{brm} \)) covering the value \( M[\text{start}(p)] - v \) (using maximum makes sure that the whole block \( \text{start}(p) \) is represented). We combine all ladders together in a single compressed array \( L \) which leads to \( 2N \) runs in total; as the difference of \( m[i] - m[\text{parent}(i)] \leq B \) is bounded, the total number of elements is \( O(N) \). The space complexity of the compressed array \( L \) is \( O(NB \log N) = o(N) \) bits.

For each block \( i \), we also store the index \( l \) of the ladder, which is also the index of the part in the compressed dictionary \( L \), in an array \( \text{ladder} \). The size of the array is \( O(NB \log N) = o(N) \) bits.

**function** LRM_SEARCH\((i, v)\):
\[
\begin{align*}
  j' &\leftarrow J_i[M[i] - v] \quad \text{▶ Jump by power of two} \\
  l &\leftarrow \text{ladder}[j'] \quad \text{▶ The ladder containing } j' \\
  \text{end} &\leftarrow L[l, 0] \\
  j &\leftarrow L[l, M[\text{end}] - v] \quad \text{▶ If the index is out of the range, } -1 \text{ is returned} \\
\end{align*}
\]
\text{return } j

**Range Operations**

We split the queried range into a prefix, suffix, and span. The solution for the prefix and suffix is solved on the block level; we focus on supporting the operations for spans of the queries.

We use a helper function \( \text{RMQL\_SPAN} \) which was defined in lemma 5 with two minor differences:

1. It is defined directly for the array \( m[\cdot] \) instead of \( E \) and \( P \);
2. It returns indices of the first and the last block containing the minimum;
3. Contrary to \( \text{RMQI} \) shown in 2.2.3, it is sufficient to use only one level provided that we set \( c \geq 2 \).

**function** RMQ\((S, g, i, j)\):
\[
\begin{align*}
  m_1 &\leftarrow e[B(i) - 1] + \text{rmq\_block}(S_{B(i)}, g, \text{prefix}(i, j)) \\
  m_2 &\leftarrow e[B(j) - 1] + \text{rmq\_block}(S_{B(j)}, g, \text{suffix}(i, j)) \\
  (f, l) &\leftarrow \text{rmq\_span}(m, \text{span}(i, j)) \\
  m_3 &\leftarrow m[f] \\
\end{align*}
\]
\text{return } \min(m_1, m_2, m_3)

The structure for \( \text{RMQL\_SPAN} \) cannot be extended for querying the number of occurrences of the minimum nor selecting the \( r \)-th one. If we augmented each precomputed interval of size \( 2^k \) with the number of occurrences, we would still fit in the same space. The problem stems for the inability of an easily combination of information from the two intervals, like the \( \text{min} \) function does. In this case, the inclusion-exclusion principle would have to be used: adding the numbers of occurrences in both intervals and subtracting the number of occurrences in the overlapping part. Since the size of the overlapping part is not a power of two in general, and so its number of occurrences is not precomputed, it leads to recursion and running time \( O(\log N) \).
Structures for RMQ_SIZE and RMQ_SELECT

We store the several indexable and fully indexable dictionaries which use the properties of the minima of the blocks. The idea is to reorder the blocks, which are represented by elements in \( m[\cdot] \), so that all blocks containing the same minimum are stored consecutively.

There are at most \( N \cdot B \) distinct minima in range \([-N, N] \). We store the minima in an indexable dictionary \( mr \) offsetted by \( N \) to become a range \([0, 2N] \). The indexable dictionary uses \( O\left(\frac{N}{B} \log B\right) \) bits and it supports RANK on the minima in constant time. This is used for referring to \( k \)-th smallest minimum in the following structures. We omit the correction of an off-by-one error caused by RANK as the smallest minimum has \( k = 1 \) instead of desired \( k = 0 \).

For each block minimum \( m \) (which is the \( k \)-th smallest), we define a set containing all blocks which have minimum \( m \): \( \{ i : m[i] = m \} \). We represent the set as an indexable dictionary \( mi_k \). Because of the inequality derived from the generalized Chu-Vandermonde’s identity \([\text{Bel}]\), the space complexity of all \( mi_m \) together can be bounded by:

\[
\sum_m S(mi_k) = \sum_m \left( \log \left( \frac{N}{\left| mi_k \right|} \right) + o(\left| mi_k \right|) + O(\log \log N) \right) \\
\leq \log \left( \left( \frac{N}{\frac{N}{B}} \right)^2 \right) + o \left( \frac{N}{B} \right) + O \left( \frac{N}{B} \log \log N \right) \\
= O \left( \frac{N}{B} \log N \right) = o(N).
\]

Finally we define an array \( mp_k \) for each block minimum \( m \) (which is again the \( k \)-th smallest):

\[
mp_k \left[ \sum_{j \in mi_k, \ j \leq i} ms[j] \right] = i \ \forall i \in mi_k.
\]

We turn all these arrays into a single compressed array \( mp \) using the operation SUCC. The property of this array, from which it was also derived, connects it to a block \( t \) which contains the \( r \)-th occurrence of the \( k \)-th smallest minimum:

\[
t = \text{select}_1(mi_k, mp[k, r]).
\]

All the structures require only \( o(N) \) bits of memory.

As we restricted our queries to their spans, and from RMQI_SPAN we know the indices of the first and last block containing the minimum, it is easy to find how many occurrences there were before the any block.
function RMQ_SIZE(S, g, i, j):
    m₁ ← 0, m₂ ← 0, m₃ ← 0
    m ← rmq(S, g, i, j)
    if e[B(i) − 1] + rmq_block(S[B(i)], g, prefix(i, j)) = m:  ▷ In prefix?
        m₁ ← rmq_size_block(S[B(i)], g, prefix(i, j))
    if e[B(j) − 1] + rmq_block(S[B(j)], g, suffix(i, j)) = m:  ▷ In suffix?
        m₂ ← rmq_size_block(S[B(j)], g, suffix(i, j))
    (f, l) ← rmq_span(m, span(i, j))
    if m[f] = m:  ▷ In span?
        k ← rank1(mr, m + N)  ▷ k-th smallest minimum
        x ← run_first(mp, k, rank1(mi_k, f))  ▷ Before the run of f
        y ← run_first(mp, k, rank1(mi_k, l))  ▷ The block l is not accounted
        m₃ ← y − x + ms[l]
    return m₁ + m₂ + m₃

The operation RMQ_SELECT first looks for the minimum in the prefix, then in the span, and finally in the suffix. The idea of the search in span is to use the property of mp.

function RMQ_SELECT(S, g, i, j, r):
    m ← rmq(S, g, i, j)
    if e[B(i) − 1] + rmq_block(S[B(i)], g, prefix(i, j)) = m:  ▷ In prefix?
        m₁ ← rmq_size_block(S[B(i)], g, prefix(i, j))  ▷ Prefix size
        if m₁ ≥ r:
            return B(i)B + rmq_select_block(S[B(i)], g, prefix(i, j), r)
        else:
            r ← r − m₁
    (f, l) ← rmq_span(m, span(i, j))
    if m[f] = m:  ▷ In span?
        m₃ ← rmq_size(S, g, span(i, j))
        if m₃ ≥ r:
            k ← rank1(mr, m + N)  ▷ k-th smallest minimum
            x ← run_first(mp, k, rank1(mi_k, f))  ▷ Before the run of f
            t' ← mp[k, r + x]  ▷ s \in \text{suffix}
            t ← select1(mi_k, t')  ▷ Block containing the desired occurrence
            y ← run_first(mp, k, t')  ▷ Elements before the run of block t
            return tB + rmq_select_block(S, g, 0, B − 1, r − y)
        else:
            r ← r − m₃
    if e[B(j) − 1] + rmq_block(S[B(j)], g, suffix(i, j)) = m:  ▷ In suffix?
        m₂ ← rmq_size_block(S[B(j)], g, suffix(i, j))  ▷ Prefix size
        if m₂ ≥ r:
            return B(j)B + rmq_select_block(S[B(j)], g, suffix(i, j), r)
        else:
            ▷ This cannot happen as the bounds of r has been checked
4.1.5 ±1 Functions Revisited

We follow in discussion of ±1 functions which we began in section 2.2.3.

Although there are 9 functions \( g \) mapping \( \{0, 1\} \rightarrow \{-1, 0, 1\} \), we only use 3 of them in our index: \( \pi, \varphi, \psi \). Answers to all operations except for searches with parameters \( \varphi \) and \( \psi \) can be answered using the structure for \( \pi \).

For the operation \( \text{sum} \), the following identities hold:

\[
\text{sum}(S, \pi, i, j) = \text{sum}(S, \varphi, i, j) - \text{sum}(S, \psi, i, j), \\
\text{sum}(S, \varphi, i, j) + \text{sum}(S, \psi, i, j) = j - i + 1.
\]

The explicit formula for sums follow from the previous equations:

\[
\text{sum}(S, \varphi, i, j) = \frac{(j - i + 1) + \text{sum}(S, \pi, i, j)}{2}, \\
\text{sum}(S, \psi, i, j) = \frac{(j - i + 1) - \text{sum}(S, \pi, i, j)}{2}.
\]

Because the sums of \( g \in \{\varphi, \psi\} \) are monotonic, we can reduce all range operations to sums and searches. In the range \( i, j \), the minimum occurs for the first time at \( i \) and maximum occurs for the last time at \( j \). All occurrences are continuous.

\[
\text{rmq}(S, g, i, j) = \text{sum}(S, g, 0, i) \\
\text{RMQ}(S, g, i, j) = \text{sum}(S, g, 0, j) \\
\text{rmq} \_\text{size}(S, g, i, j) = \min(j + 1, \text{fwd} \_\text{search}(S, g, i, 1)) - i \\
\text{RMQ} \_\text{size}(S, g, i, j) = j - \max(i - 1, \text{bwd} \_\text{search}(S, g, j, 1)) \\
\text{rmq} \_\text{select}(S, g, i, j, r) = i + r - 1 \\
\text{RMQ} \_\text{select}(S, g, i, j, r) = j - \text{RMQ} \_\text{size}(S, g, i, j) + r
\]

Search Operations

The operation \text{BWD\_SEARCH} can be the reduced to \text{FWD\_SEARCH} and \text{SUM}.

\text{function BWD\_SEARCH}(S, g, i, d): 
  if \( d > 0 \):
    \( v \leftarrow \text{sum}(S, g, 0, i) - d + 1 \)
    if \( v \geq 2 \) or (\( v = 1 \) and \( g(S[0]) \neq 1 \)):
      return \text{fwd\_search}(S, g, 0, v)
    else if \( v = 1 \):
      return 0
    else:
      return \(-1\)
  else if \( d = 0 \) and \( g(S[i]) = 0 \) and \( g(S[i - 1]) = 0 \):
    return \( i - 1 \)
  else:
    return \(-1\)

The only non-trivial operation is \text{FWD\_SEARCH}. We review the structures which are used in its index and note places which can be simplified.
No simplification is possible for small block (the look-up tables) and block (min-max tree) levels. In the macro structure, only the tree $T_{LRM}$ is necessary ($T_{lrm}$ is not because of the monotonicity of $g$). The tree is also degenerated into a path. The tiny compressed array of jumps are not necessary as the whole path forms a ladder; the search in the ladder stays the same.

The operation FWD_SEARCH can be used to implement SELECT, which makes the index for FWD_SEARCH an alternative to the index which we showed in the section 2.1.2.

function SELECT$_1(S, i)$:
  if $i \geq 2$ or ($i = 1$ and $S[0] \neq 1$):
    return fwd_search($S, \varphi, 0, i$) ▷ In case of SELECT$_0$: ̸= 0
  else if $i = 1$:
    return 0
  else:
    return −1

4.1.6 Extension of BP

The index which we developed in the previous section can be used for any representation, however only the BP representation benefits from it as many operations can be immediately supported.

Child operations, mainly DEGREE, CHILD_RANK, and CHILD_SELECT, which are important for basic navigation in the tree, were only available with a specialized index. They were also the reason for developing the DFUDS representation, which supports them natively. Using the generalized RMQ operations, we present an alternative implementation to the one described by [SN10]. The key observation is that in a representation of a subtree of a vertex $i$ with omitted boundary parentheses, the occurrences of minimum correspond with the terminal parentheses of the children of $i$.

function DEGREE($i$):
  return rmq_size($S, \pi, i + 1, \text{find}_\text{close}(S, i) - 1$)

function CHILD_RANK($i$):
  $p \leftarrow$ parent($i$)
  return rmq_rank($S, \pi, p + 1, \text{find}_\text{close}(S, p) - 1, \text{find}_\text{close}(S, i)$)

function CHILD_SELECT($i, k$):
  return find_open($S, \text{rmq}_\text{select}(S, \pi, i + 1, \text{find}_\text{close}(i) - 1, k)$)

The operation LEVEL_ANCESTOR is now a straightforward generalization of ENCLOSE.

function LEVEL_ANCESTOR($i, d$):
  if $d = 0$:
    return $i$
  else:
    return bwd_search($S, \pi, i, d + 1$)

It is also possible to realize some of the LEVEL_* operations with the restriction to a subtree of a vertex $a$. The most general ones: LEVEL_SIZE, LEVEL_RANK, and LEVEL_SELECT are however not supported.
function LEVEL_FIRST(a, l):
    if depth(S, a) = l:
        return a
    else:
        return fwd_search(S, π, a, l - depth(S, a))

function LEVEL_LAST(a, l):
    if depth(S, a) = l:
        return a
    else:
        return find_open(S, bwd_search(S, π, find_close(S, a), depth(S, a) - l))

function LEVEL_NEXT(a, i):
    return fwd_search(S, π, find_close(S, i), 0)

function LEVEL_PREV(a, i):
    return find_open(S, bwd_search(S, π, i, 0))

4.1.7 Final Thoughts

The constant \( c \) needs to be set to \( c \geq 2 \) because of the level-ancestor and range minimum query macro structure.

The single index which we presented is more general than all the indices which used the BP representation. As a consequence more operations are supported, in fact more than any of the previous representations.

The only unsupported operations by this index are (1) the RANK and SELECT in respect to lo-order and dfuds-order numberings of vertices; (2) arbitrary access for level queries through LEVEL_SIZE, LEVEL_RANK, and LEVEL_SELECT operations.

All of the listed operations would benefit from a better handling of the level structure of the tree, an operation like FWD_SEARCH which is parametrized by the number of the occurrence. If these operations are crucial, the only data structure which supports them is the LOUDS.

4.2 Tree Covering

All tree data structures which we have shown so far have one thing in common; they solve their operations on two or three different levels:

(1) On the lowest level, they are decomposed into small blocks of size \( b \) which is less than \( \log n \) bits. This size makes it possible to precompute results to all possible queries which are contained in a small block.

(2) Then on an intermediate level, several small blocks are grouped together to form blocks of size \( B = O(\log^c n) \) bits. The purpose of this level is to lower the number of blocks for the next level.

(3) Finally, the macro level which spans the whole structure connects individual blocks. It usually uses data structures which are inspired by the traditional ones which are allowed to use \( O\left(\frac{n}{B} \log^{c'} n\right) \) bits where \( c' < c \).
So far, the small blocks and blocks were chunks of a bit string representation of the tree. They did not have any connection to the structure of the tree; all operations were implemented in their most generality for an arbitrary (balanced) bit string. The representation of a single vertex, let alone the sequence of its children or the whole subtree, could have been covered by multiple (small) blocks.

The tree covering approach respects the structure of the tree by its decomposition into components of bounded sizes. Not dissimilar to the other representations, it also uses three levels of scale with look-up tables being utilized on the lowest one.

The basic operations (CHILD_*, DEGREE, DEPTH, SUBTREE_SIZE, PRE_*, POST_*) were described by [GRR06]. They were also the first ones who supported the operation LEVEL_ANCESTOR.

The more advanced operations (HEIGHT, LCA, DISTANCE, LEAF_*, DFUDS_*, LEVEL_*) were introduced by [HMR07].

Many of the operation were later simplified by [FM14] with introduction of the better algorithm for tree decomposition.

We propose the operations DEEPEST_VERTEX and IN_. We also show an alternative solutions to HEIGHT and LEVEL_FIRST and LEVEL_LAST. We further explicitly restrict the operations to the subtree of a given vertex.

4.2.1 Decomposition Algorithm

Instead of decomposing a bit string of the encoded tree, we decompose the tree and then we encode it. Two decomposition algorithms have been proposed; they are parametrized by a target number of vertices $B$ to be present in a component. The first algorithm [GRR06] decomposes the tree into connected components of size $[B, 3B - 2]$ with the exception of the component containing the root, which can be undersized. Such decomposition cannot exist unless the components are allowed to overlap; more specifically to overlap in their common root. It follows from the bounds that there exist $O\left(\frac{n}{B}\right)$ components.

This decomposition was used and it leads to a succinct data structure which at the time of its introduction supported more operations than BP (child operations) or DFUDS (depth). The problem of the decomposition was that the components were connected together in too many ways and the structure had to handle many cases.

Later a different decomposition was proposed in [FM08] with more details published in [FM14]. It removes the lower bound on the sizes of the components while retaining their asymptotic number in exchange for restriction on the ways how the components can be connected with each other. At most one edge from a non-root vertex connecting a different component is allowed. Note that a stronger claim of no such edge existing does not provide a decomposition in a general case (e.g., a tree with depth greater than $2B$).

We present the latter decomposition since it results in components satisfying stronger properties. A vertex is called heavy if its subtree contains at least $B$ vertices. All ancestors of a heavy vertex are heavy and therefore heavy vertices form a subtree $T_{\text{heavy}}$ of the tree $T$. A heavy vertex is called a branching vertex.
if it has at least two heavy children, and branching edges are called the edges connecting a branching vertex with its heavy children.

**Lemma 14.** There are $O\left(\frac{n}{B}\right)$ branching vertices and branching edges in a tree $T$ with $n$ vertices.

*Proof.* The tree $T_{\text{heavy}}$ has at most $\frac{n}{B}$ leaves (we call them heavy leaves) as each leaf is a root of a subtree of $T$ which contains at least $B$ vertices. Each branching vertex connects at least two heavy subtrees containing each at least one heavy leaf; therefore their number must be less than the number of all heavy leaves. The number of branching edges is the same as the number of heavy leaves plus the number of branching vertices minus one which can be seen after contraction of non-branching edges in the tree $T$. \hfill \square

The decomposition is defined by an algorithm which processes the tree in DFS post-order; it first recursively processes all children of a vertex $v$ before solving $v$ itself. A leaf starts its own temporary component – a component which has not been declared permanent. A set of permanent and at most one temporary component is returned from the recursion. The way how the components are processed in $v$ depends on the number of its heavy children.

We distinguish three cases:

1. If a vertex $v$ does not have any heavy children, then all children are parts of temporary components. Children are processed from left to right; their temporary components are merged with $v$ and potentially with the components of their right siblings. When the size of the current component is at least $B$, we declare it permanent and start the same process with the next temporary component if there is any. If at least one component was declared permanent, we declare all of them as permanent even though the last one can remain undersized.

2. If a vertex $v$ has exactly one heavy child $u$, then we process it in a similar way as the case (1). If $u$ is part of a temporary component, nothing changes; otherwise we simply skip it.

3. If a vertex $v$ has two or more heavy children ($v$ is a branching vertex), the temporary components containing the heavy children are declared permanent no matter what size they are. All non-heavy children are split into intervals delimited by the heavy children. Each interval is processed as in case (1) with the exception that all components are declared as permanent. If the vertex $v$ does not have any non-heavy children, then it forms a permanent component of size one.

Several invariants hold during the course of the decomposition algorithm:

- The size of a permanent component is less than $2B - 2$; the size of a temporary component is less than $B$. A temporary component is only merged with another temporary components; it is declared permanent when its size is at least $B$. Because we merge a temporary component first with the parent and then with its right siblings one by one, its size will never be greater than $(B - 2) + 1 + (B - 1)$, at which point it is declared permanent.
• If a vertex is shared among multiple components, it is their common root. When such situation happens in the algorithm, all of them are declared as permanent and they are never dealt with again.

• Whenever a vertex \( v \) is being processed and a component containing \( v \) is declared permanent, then \( v \) is a heavy vertex. In case (2) and (3) it is heavy from the fact that it is a parent of heavy children. In case (1) the first component is declared permanent if the components of children plus \( v \) exceed \( B \). If the first component is not declared permanent, none is.

• There is at most one edge leaving a component from a non-root vertex. A root \( u \) of a permanent component which is connected by such edge is heavy from the previous invariant. Therefore, when the parent \( v \) of \( u \) is being processed, \( u \) is its heavy child. If a component of another child of \( v \) already contained such edge, then \( v \) has at least two heavy children and case (3) applies. All components end with \( v \) or its children, and so \( u \) is connected to a root vertex instead of a non-root one.

• The number of all components is \( O(\frac{n}{B}) \). We charge \( O(1) \) undersized components to regular-sized components, branching edges, and branching vertices. The bound then follows from the lemma [14]. If an undersized component was declared permanent in (1) or (2), then another component of a regular size was declared too. In (3), it was either connected with a branching edge, or it happened once per interval of non-heavy children, which is delimited by at least one branching edge, or it is the branching vertex itself.

We propose a modification of the algorithm which makes the resulting decomposition satisfy one more constraint. In case (2), it could happen that the component of the heavy child \( u \) is permanent, and some permanent components containing \( v \) are created. In such situation a left and a right sibling of \( u \) could be in the same component, and thereby make the sequence of children of \( v \) discontinuous in terms of component to which they belong. Moreover, a heavy temporary component is forcefully declared permanent in case (3), which would lead to the same result.

We prevent this situation in case (2) to happen by splitting the children of \( v \) into two intervals (one can be empty) and solving them separately similar to the case (3). In case (3) we revise the temporary heavy component and split it in the same way; for that we return from recursion the vertex just before \( u \). This can result in at most two permanent undersized components which we charge to the permanent regular-sized component which was not created. When the root is processed, a temporary component may need to be split.

**Lemma 15.** The components created by the altered decomposition algorithm satisfy the following property. Let \( C \) be a sequence of components to which the children of a root of a component belong, then each component occurs in at most one run in \( C \).

**Proof.** The proof follows from the stated algorithm. \( \Box \)
function decomposeSubtree(v, B):
    if is_leaf(v):
        return [], v, −1  # No permanent, itself as temporary, heavy children
    else:
        P ← [], T ← []  # All permanent and temporary from children
        R ← [], h ← 0  # Split vertices in children, number of heavy comps
        d ← 0  # Total size of temporary components
        for all u ← children(v):
            (p, t, r′) = decomposeSubtree(u, B)  # Process child
            concatenate(P, p)  # Gather permanent
            append(T, t), append(R, r′), d ← d + |t|  # Store and count temp
        if subtree_size(u) ≥ B:  # Count number of heavy children
            h ← h + 1
        s ← [v], i ← 0, r ← −1  # Working component
        for all (u, t, r′) ← zip(children(v), T, R):
            if subtree_size(u) ≥ B and |t| > 0 and h ≥ 2:  # Heavy, temp, c. 3
                if r′ ≠ −1:
                    s′ ← []  # Revise the component
                    for all u′ ← t:
                        append(s′, u′)
                        if u′ = r′:
                            append(P, s′), s′ = [u]  # Declare permanent, reset
                            append(P, s′)  # Declare permanent
                    else:
                        append(P, t)  # Declare permanent
                t ← []  # t has been processed
            else if |t| = 0 and |s| > 1 and d > B:  # Heavy, perm, case 2, large
                append(P, s), s ← [v], i ← i + 1  # Declare permanent
            else if |t| = 0:  # Heavy, permanent, case 2, small
                if child_first(v) ≠ u and child_last(v) ≠ u:
                    r ← child_prev(u)  # Mark prev for potential splitting
                if |t| = 0 and |s| > 1 and h ≥ 2:  # Permanent, non-empty s, case 3
                    append(P, s), s ← [v]  # Declare permanent, reset
            else if |t| ≠ 0:  # Temporary
                concatenate(s, t)  # Merge t into s
            if |s| ≥ B:
                append(P, s), s ← [v], i ← i + 1  # Regular sized
                if |s| = 1:
                    s ← []  # Reset before returning
                else if i ≥ 1 or h ≥ 2:  # At least one regular sized or case 3
                    append(P, s), s ← []  # Declare permanent
                if h = degree(v) and h ≥ 2:  # All are heavy, case 3
                    append(P, [v])  # Declare v as permanent
        return P, s, r
function \textsc{decompose}(v, B):
\( (P, t, r) \leftarrow \textsc{decomposeSubtree}(v, B) \)
\( \triangleright \) Decompose the subtree of root
\( \text{if } |t| \neq 0: \)
\( \text{if } r = -1: \)
\( \text{append}(P, t) \) \( \triangleright \) A temporary component exists
\( \text{else:} \)
\( s \leftarrow [\ ] \)
\( \text{for all } u \leftarrow t: \)
\( \text{append}(s, u) \)
\( \text{if } u = r: \)
\( \text{append}(P, s), \quad s = [u] \) \( \triangleright \) Declare permanent
\( \text{append}(P, s) \) \( \triangleright \) There is a split vertex
\( \text{if } u = -1: \)
\( \text{append}(P, t) \) \( \triangleright \) Declare permanent
\( \text{else:} \)
\( \text{for all } u \leftarrow t: \)
\( \text{append}(s, u) \)
\( \text{if } u = r: \)
\( \text{append}(P, s), \quad s = [u] \) \( \triangleright \) Declare permanent
\( \text{append}(P, s) \) \( \triangleright \) Declare permanent

\textbf{4.2.2 The Structure}

We run the decomposition algorithm twice; the first time with \( B = \log^c n \) for \( c \geq 2 \) which will be specified later. The second time we decompose the components into small components with parameter \( b = \frac{\log n}{8} \). The components form connected subtrees in the tree, which we call mini-trees; similarly we call the subtrees in small components micro-trees.

There are three different connections between mini-trees (and similarly between micro-trees):

(1) Two mini-trees \( s, t \) share their root vertex. The common root vertex can be shared among multiple mini-trees:

\[ \text{root}(s) = \text{root}(t). \]

(2) A parent of a root of a mini-tree \( s \) is a root of a different mini-tree \( t \). Either of the roots can be shared with other mini-trees:

\[ \text{parent}(\text{root}(s)) = \text{root}(t). \]

(3) A parent of a root of a mini-tree \( s \) (bottom) is a non-root vertex in a different mini-tree \( t \) (top):

\[ \text{parent}(\text{root}(s)) \in t \text{ and } \text{parent}(\text{root}(s)) \neq \text{root}(t). \]

If this type of connection appears more than once in a mini-tree, then all bottom components share a common root.

We define a set of terms for root related structures, which we use to refer to them without ambiguities. A \textit{mini-tree root} is the root of a mini-tree. A \textit{root mini-tree} is any mini-tree which contains the mini-tree root. Similarly we define the terms for micro-trees where \textit{root micro-tree} refers to a micro-tree which contains a micro-tree root. We also use negations, for example “a micro-tree root of a non-root mini-tree”.

71
Vertex Naming

We order the mini-trees based on their pre-order number in the original tree. For each pair of mini-trees $s, t$, the following properties hold:

1. $\text{root}(s) \leq \text{root}(t)$;
2. If $\text{root}(s) = \text{root}(t)$, then their first children in the mini-trees are compared: $\text{child}\_\text{first}(\text{root}(s)) < \text{child}\_\text{first}(\text{root}(t))$. Note that even though there are mini-trees with a single vertex, their roots are not shared.

Each mini-tree is then assigned number $\tau_1$ based on this order. The first mini-tree (it contains the tree root) is assigned the number 0. Similarly we assign a number $\tau_2$ to each micro-tree in a mini-tree $\tau_1$ and name the micro-tree $(\tau_1, \tau_2)$. We continue with individual vertices in the same way; they are given a name $(\tau_1, \tau_2, \tau_3)$.

The size of $\tau_1$ is $O(\log n)$; the sizes of $\tau_2$ and $\tau_3$ are $O(\log \log n)$. A vertex can be assigned multiple names when it is a common root of several micro-trees, which could even be in different mini-trees. A canonical name of a mini-tree, a micro-tree, or a vertex is the lexicographically smallest one. The interface of all operations on the tree assumes that all names of vertices are canonical; non-canonical and partial names are used only for internal purposes of the data structure.

In order to handle the case (3) connection, we alter the process of decomposition of the tree. After the mini-trees are identified by the decomposition algorithm, we subdivide each edge realizing the connection of type (3) by introducing a dummy vertex, to which we refer as mini-tree dummy vertex. This new vertex is then added to the top mini-tree, which increases its upper bound by one to $2^B - 1$. Then we follow with the second level of the decomposition (mini-trees into micro-trees) and again introduce dummy vertices, to which we refer as micro-tree dummy vertices.

The total number of dummy vertices introduced is $O\left(\frac{n}{B} + \frac{n}{\log n}\right) = O\left(\frac{n}{\log n}\right)$. The dummy vertex gets its name as if it was a normal vertex in the top mini-tree or micro-tree. The canonical name of a dummy vertex is the canonical name of the vertex to which it is connected in the bottom mini-tree or micro-tree, which is its root. The depth of a dummy vertex in a mini-tree or a micro-tree is at least 2, because it has a parent which is not a root.

We call primary the mini-tree $\tau_1$ and the micro-tree $(\tau_1, \tau_2)$ which contains a root with a canonical name $(\tau_1, \tau_2, 0)$. We also call primary the micro-tree $(\tau_1, 0, 0)$ so that every micro-tree has its primary micro-tree within the same mini-tree. A root micro-tree can have two primary micro-trees: one within the same mini-tree, the other one in its primary mini-tree. In the latter case we use the phrase “in the primary mini-tree”.

The same way we numbered mini-trees we number primary mini-trees; we assign them $\sigma_1$ names. We do the same for micro-trees with $\sigma_2$ names. These names are only a technicality which is used to index structures which are stored in the tree structure instead of each primary mini-tree.
Representation

We use compressed arrays in two different settings throughout the whole data structure:

• A single compressed array for all vertices in a mini-tree or a micro-tree with a bounded number of runs.

  Specifically, if we store $\tau_1$ names of all vertices of the tree requiring $O(1)$ runs per name, then the space is $O\left(\frac{B}{\epsilon} \log n \right) + o(n) = o(n)$ bits. Similarly for $\tau_2$ names of vertices of a given mini-tree, the space complexity is $O\left(\frac{B}{\epsilon} \log B \right) + o(B) = o(B)$ bits.

• A collection of $O(r)$ compressed arrays $A_i$, containing $a$ elements of size $s$ in $r$ runs in total.

  If each $\tau_1$ name occurs in $O(1)$ arrays in $O(1)$ runs, and there is at most one record for each vertex of the tree, then the space complexity is the same as in the previous case. The same applies for $\tau_2$ names within a given mini-tree.

For each micro-tree we store several pieces of information:

Identity of the micro-tree

Information about the position of the micro-tree within the mini-tree.

$\tau_2$  Its $\tau_2$ name; if the name equals to 0, it is a root micro-tree.

$offset$  The offset of this structure from the beginning of the structure of the mini-tree in bits. While $\tau_1$ name is too big, the offset is at most $B + o(B)$, which results in $\log \log n$ bits. In order to move to the mini-tree structure, we subtract the $offset$ from the position where the micro-tree structure begins.

$primary_2$  The $\tau_2$ name of the primary micro-tree; it can be the micro-tree itself.

$\sigma_2$  The $\sigma_2$ name of the primary micro-tree.

Parent and the dummy vertex

Each micro-tree can have a parent micro-tree and can contain a dummy vertex which leads to another mini-tree or micro-tree.

$parent_2$  The $\tau_2$ name of the primary micro-tree which contains its parent. The parent could in theory be stored only in the structure of its primary micro-tree, however, it is small enough to keep a copy in all of them.

$type_3$  A boolean denoting whether it is connected to its parent micro-tree by a type (3) connection.

$dummy_2$  The $\tau_3$ name of the dummy vertex which represents the type (3) connection to a different micro-tree. The value $-1$ means that the micro-tree does not have a dummy vertex.

$bottom_2$  The $\tau_2$ name of the primary micro-tree to which the dummy vertex
leads. If $dummy_2 \neq -1$ and $bottom_2 = -1$, then the dummy vertex leads to a different mini-tree, which is handled by the mini-tree structure.

**Children**

Since the structure storing children is a collection of compressed arrays, it is stored on a mini-tree level. Here we store only fields which are useful for determining `CHILD_RANK`.

- $children\_index_2$
  - The index of the first occurrence of $\tau_2$ within the compressed array of the children for the primary micro-tree.
- $children\_parent_2$
  - The index of the first occurrence of $\tau_2$ within the compressed array of the children for the parent’s micro-tree. This index is $-1$ if $type3 = true$ because in such case the micro-tree root is not a child of parent’s micro-tree root.

**Representation**

All look-up tables will take the following two fields as their arguments.

- $size$  The number of vertices $k$ of the micro-tree; this includes the possibly shared root and the dummy vertex.
- $rep$  The succinct (or even implicit) representation of the micro-tree which uses $2k$ bits.

All fields except for $rep$ require only $O(\log \log n)$ bits per micro-tree. The succinct representations $rep$ of all micro-trees together contain all $n$ vertices of the tree together with $o(n)$ dummy vertices and $o(n)$ shared root vertices. The representations require $2n + o(n)$ bits in total and are the dominant part of the structure.

Each micro-tree contains $k$ vertices, with an upper bound of $k \leq 2b - 1 < \frac{\log n}{4}$ vertices. We can represent this subtree succinctly using $2k < \frac{\log n}{2}$ bits, which is small enough to be used as an index to a look-up table. All our look-up tables will require space $o(n)$.

The structure for a mini-tree is similar to the micro-tree structure. The only difference is that it contains the collection of compressed arrays for roots of micro-trees.

**Identity of the mini-tree**

- $\tau_1$  Its $\tau_1$ name.
- $primary_1$
  - The $\tau_1$ name of the primary mini-tree; it can be the mini-tree itself.
- $\sigma_1$  The $\sigma_1$ name of the primary mini-tree.

**Parent and the dummy vertex**

- $parent_1$
  - The $\tau_1$ name of the mini-tree which contains the parent of the root of this mini-tree.
dummy<sub>i</sub>

The $\tau_2$ name of the micro-tree which contains the dummy vertex which was introduced for edges between mini-trees. If there is no mini-tree dummy vertex, then it is $-1$.

bottom<sub>i</sub>

The $\tau_1$ name of the mini-tree to which the micro-tree dummy vertex leads, or $-1$.

**Children**

$children_2$

A collection of compressed arrays containing one array per a primary micro-tree. Each array contains $\tau_2$ names of children of the micro-tree root restricted to the current mini-tree. Individual parts of the collection are accessed by $\sigma_2$ names of the micro-trees.

$children\_index_i$

The index of the first occurrence of $\tau_1$ within the compressed array of the children for the primary mini-tree.

$children\_parent_i$

The index of the first occurrence of $\tau_1$ within the compressed array of the children for the parent’s mini-tree. This index is $-1$ if $type3 = true$.

**Representation**

$micro\_tree\_offsets$

A table of offsets of the micro-tree structures, as in lemma $\surd$.

$micro\_trees$

The micro-tree structures stored consecutively.

All fields except for $children_2$ and the representation, require $O(\log n)$ bits of space. The collection of compressed arrays $children_2$ satisfies the property stated earlier since: (1) each vertex (and each micro-tree) is a child of at most one root; (2) all occurrences of $\tau_2$ form a single run which follows from our modification of the decomposition algorithm.

At the global level, we only need a very simple structure:

**Children**

$children_1$

The same structure as $children_2$ in mini-trees: each array contains $\tau_1$ names of children of a mini-tree root.

**Representation**

$mini\_tree\_offsets$

A table of offsets of the mini-tree structures, as in lemma $\surd$.

$micro\_trees$

The mini-tree structures stored consecutively.
4.2.3 Navigation Operations

With the structure which we have just described, the navigation operations can already be supported. In the presented algorithms we identify the structures with their names provided that we can navigate to them:

- **global → mini-tree \(\tau_1\)**
  - using \(\text{mini\_tree\_offsets}\) and \(\text{mini\_trees}\);

- **mini-tree \(\tau_1\) → micro-tree \((\tau_1, \tau_2)\)**
  - using \(\text{micro\_tree\_offsets}\) and \(\text{micro\_trees}\);

- **micro-tree \((\tau_1, \tau_2)\) → mini-tree \(\tau_1\)**
  - using \(\text{offset}\);

- **mini-tree \(\tau_1\) → global**
  - it simply starts at position 0.

Since individual vertices do not have a structure on their own, then we use a vertex name \(i = (\tau_1, \tau_2, \tau_3)\) to access the micro-tree \((\tau_1, \tau_2)\). We assume that mini-trees have access to all global fields, and all micro-trees have access to all mini-tree and global fields.

**Helper Functions**

We define three helper functions which traverse the edge of type (3) and canonize the name of a vertex. As they are not part of the interface of the data structure, the argument \(i\) is allowed to be a non-canonical name.

- **function \textsc{dummy\_up}(i):**
  - if \(i.\tau_3 = 0\) and \(i.\text{type3}\):
    - ▶ If root and type (3) connected
    - if \(i.\tau_2 \neq 0\):
      - ▶ Micro-tree dummy vertex
        - return \((i.\tau_1, \text{parent}_2, \text{parent}_2.\text{dummy}_2)\)
    - else:
      - ▶ Mini-tree dummy vertex
        - return \((i.\text{parent}_1, i.\text{parent}_1.\text{dummy}_1, i.\text{parent}_1.\text{dummy}_1.\text{dummy}_2)\)
  - else:
    - return \(i\)

- **function \textsc{dummy\_down}(i):**
  - if \(i.\tau_3 = i.\text{dummy}_2\):
    - ▶ This is a dummy vertex
      - if \(i.\text{bottom}_2 \neq -1\):
        - ▶ Micro-tree dummy vertex
          - return \((i.\tau_1, i.\text{bottom}_2, 0)\)
      - else:
        - ▶ Mini-tree dummy vertex
          - return \((i.\text{bottom}_1, 0, 0)\)
  - else:
    - return \(i\)

- **function \textsc{canonize}(i):**
  - if \(i.\tau_3 \neq 0\):
    - ▶ Potentially dummy vertex
      - return \textsc{dummy\_down}(i)
  - else if \(i.\text{primary}_2 \neq 0\):
    - ▶ Not a root micro-tree
      - return \((i.\tau_1, i.\text{primary}_2, 0)\)
  - else:
    - ▶ Potentially shared with different mini-tree
      - return \((j.\text{primary}_1, 0, 0)\)
We add four more functions which navigate to special vertices in a given micro-tree or a mini-tree. Their names alias with the fields in the representation, however they can always be distinguished by the parentheses. The function can also be called with mini-tree name or a micro-tree name. The last four functions assume that such dummy vertex exists.

\[
\begin{align*}
\text{root}_1(i) &= \text{canonize}(i.\tau_1, 0, 0) \\
\text{root}_2(i) &= \text{canonize}(i.\tau_1, i.\tau_2, 0) \\
\text{dummy}_1(i) &= (i.\tau_1, i.\text{dummy}_1, i.\text{dummy}_1, i.\text{dummy}_2) \\
\text{dummy}_2(i) &= (i.\tau_1, i.\tau_2, i.\text{dummy}_2)
\end{align*}
\]

\[
\begin{align*}
\text{bottom}_1(i) &= \text{dummy}_\text{down}(\text{dummy}_1(i)) \\
\text{bottom}_2(i) &= \text{dummy}_\text{down}(\text{dummy}_2(i))
\end{align*}
\]

The operations \texttt{is_ROOT} and \texttt{is_LEAF} use the property that \(i\) is a canonical name. If \(i\) is a root of the tree, it is part of the primary root mini-tree, which in pre-order numbering has the number 0. We apply the same reasoning on micro-trees inside the mini-tree and to the vertex inside the micro-tree.

Similarly, because of the canonicity, \(i\) cannot be the name of a dummy vertex inside the top micro-tree. The \texttt{DEGREE} look-up table used in \texttt{is_LEAF} can therefore be oblivious of the existence of dummy vertices.

\begin{verbatim}
function \texttt{is_ROOT}(i):
    return \(i = (0, 0, 0)\)

function \texttt{is_LEAF}(i):
    return \(\text{degree}[i.\text{size}, i.\text{rep}, i.\tau_3] = 0\)
\end{verbatim}

**Parent**

A vertex can potentially use a connection of type (3) to its parent; we handle that by utilizing the \texttt{dummy_up} function. Then we continue differently in cases of a non-root, a micro-tree root but not a mini-tree root, a mini-tree root, and the tree root. This pattern of four cases will keep recurring in most of our algorithms.

If the \(i\) is not a micro-tree root \((\tau_3 \neq 0)\), we use a look-up table to obtain the answer. If it is a micro-tree root \((\tau_3 = 0)\) but not a mini-tree root \((\tau_2 \neq 0)\), then the parent is present in a micro-tree \(\text{parent}_2\) within the same mini-tree. If the vertex is a mini-tree root, but not the root of the whole tree \((\tau_2 = 0 \text{ and } \tau_1 \neq 0)\), the answer is the root of the parent mini-tree. The last case of the tree root vertex \((0, 0, 0)\) simply leads to a failure.

\begin{verbatim}
function \texttt{PARENT}(i):
    \(i \leftarrow \text{dummy_up}(i)\) \hfill \triangleright \text{Handles type (3) connection}
    \quad \text{if } i.\tau_3 \neq 0:\n    \quad \quad \text{return } (i.\tau_1, i.\tau_2, \text{parent}[i.\text{size}, i.\text{rep}, i.\tau_3]) \hfill \triangleright \text{Non-root}
    \quad \text{else if } i.\tau_2 \neq 0:\n    \quad \quad \text{return } (i.\tau_1, i.\text{parent}_2, 0) \hfill \triangleright \text{Micro-tree root but not mini-tree root}
    \quad \text{else if } i.\tau_1 \neq 0:\n    \quad \quad \text{return } (i.\text{parent}_1, 0, 0) \hfill \triangleright \text{Mini-tree root but not tree root}
    \quad \text{else:}
    \quad \quad \text{return } -1 \hfill \triangleright \text{Tree root}
\end{verbatim}
Children

We solve all operations on children of a vertex $i$ (DEGREE, CHILD_RANK and CHILD_SELECT) using the compressed arrays $children_1$ and $children_2$.

**function** DEGREE($i$):
   if $i.\tau_3 \neq 0$:
      return $degree[i.size, i.rep, i.\tau_3]$
   else if $i.\tau_2 \neq 0$:
      return $size(i.children_2, i.\sigma_2)$
   else:
      return $size(i.children_1, i.\sigma_1)$

The CHILD_RANK algorithm starts with identifying where the current micro-tree is located in the compressed array of children of $p$; for this we have stored indices $children_{index}\{1, 2\}$ and $children_{parent}\{1, 2\}$ in the representation. Their alternative meaning is that they represent the number of children before the current mini-tree or micro-tree. Note that we use the property from lemma 15.

Two cases need to be distinguished: (1) the parent $p$ is within the same micro-tree, (2) or its is in a different micro-tree. If $p$ is a mini-tree root, we follow the same procedure first on a mini-tree level, then on a micro-tree level. Finally, we use a look-up table to find the CHILD_RANK within a micro-tree.

**function** CHILD_RANK($i$):
   $p \leftarrow parent(i)$
   $i \leftarrow dummy_{up}(i)$
   \quad \triangleright Handling of type (3) connection
   if $p.\tau_3 \neq 0$:
      return $child_rank[p.size, p.rep, i.\tau_3]$
   else if $p.\tau_2 \neq 0$:
      if $i.\text{primary}_2 = p.\tau_2$:
         return $i.children_{index}_2 + child_rank[i.size, i.depth, i.\tau_3]$
      else:
         return $i.children_{parent}_2 + 1$
   else if $p.\tau_1 \neq 0$:
      if $i.\text{primary}_1 = p.\tau_1$:
         \quad \triangleright Connection type (1) between mini-trees
         if $i.\text{primary}_2 = p.\tau_2$:
            \quad \triangleright Connection type (1) between micro-trees
            $r \leftarrow child_rank[i.size, i.depth, i.\tau_3]$
            return $i.children_{index}_1 + i.children_{index}_2 + r$
         else:
            \quad \triangleright Connection type (2) between micro-trees
            return $i.children_{index}_1 + i.children_{parent}_2 + 1$
         else:
            return $i.children_{parent}_1 + 1$
      else:
         return $-1$

The algorithm CHILD_SELECT introduces a pattern which will be often used in other operations. When $i$ is a mini-tree root, we first find the mini-tree which contains the $k$-th child, and the offset $k'$ within the mini-tree. We follow in the same way with micro-trees. Finally we solve the query within a micro-tree using a look-up table.
function CHILD_SELECT(i, k):
  if i.τ_3 ≠ 0:
    d ← child_select[i.size, i.rep, i.τ_3, k]
    return dummy_down(d)
  else if i.τ_2 ≠ 0:
    d ← i.children_2[k - 1]
    k' ← rank(i.children_2, k - 1)
    if d.primary_2 = i.τ_2:  # Connection type (1) between micro-trees
      return child_select[d.size, d.rep, 0, k']
    else:
      return (i.τ_1, d, 0)
  else:
    d_1 ← i.children_1[k - 1]
    k' ← rank(i.children_1, k - 1)
    if d_1.primary_1 = i.τ_1:  # Connection type (1) between mini-trees
      d_2 ← d_1.children_2[k' - 1]
      k'' ← rank(d_1.children_2, k' - 1)
      if d_2.primary_2 = i.τ_2:  # Connection type (1) between micro-trees
        return child_select[d_2.size, d_2.rep, 0, k'']
      else:
        return (d_1, d_2, 0)
    else:
      return (d_1, 0, 0)  # Connection type (2) between mini-trees

4.2.4 Depth, Deepest Vertex, Height, Subtree-Size

In order to support more operations we need to augment the structure with more information. Here we focus on operations which return information about vertices.

Micro-tree

The micro-tree structure is augmented with the following fields:

- depth_2
  The depth of the micro-tree root within the mini-tree, which is the distance from the mini-tree root.

- dummy_ancestor
  A boolean flag denoting whether the micro-tree root is an ancestor of a mini-tree dummy vertex.

- subtree_size_2
  The size of the subtree of the micro-tree root excluding any type (3) connected mini-tree.

- deepest_vertex_2
  The (τ_2, τ_3) name of the deepest vertex within the subtree restricted to the current mini-tree. It is often a mini-tree dummy vertex.

Mini-tree

The mini-tree structure contains information about the mini-tree root and its subtree.
The depth of the mini-tree root within the whole tree.

The full size of the subtree.

The full name of the deepest vertex within the subtree.

The operation \texttt{depth} is straightforward; it sums the micro-tree-local depth of the vertex, mini-tree-local depth of the micro-tree root, and the global depth of the mini-tree root.

\textbf{function} \texttt{depth}(i):
\begin{itemize}
\item if \(i.\tau_3 \neq 0\):
  \begin{itemize}
  \item return \texttt{depth}(\text{root}_2(i)) + \texttt{depth}[\text{size}, \text{rep}, i.\tau_3]
  \end{itemize}
\item else if \(i.\tau_2 \neq 0\):
  \begin{itemize}
  \item return \texttt{depth}(\text{root}_1(i)) + i.\text{depth}_2
  \end{itemize}
\item else:
  \begin{itemize}
  \item return \texttt{i.depth}_1
  \end{itemize}
\end{itemize}

The operation \texttt{deepest_vertex} is more complex than \texttt{depth}.

If \(i\) is a non-root, we find the deepest vertex using a look-up table. We also consider the alternative that the deepest vertex is present in the subtree which is connected via a dummy vertex, if \(i\) is its ancestor. The recursive call reduces the search to one of the following cases.

If \(i\) is a micro-tree root but not a mini-tree root, then if it is not a dummy vertex, we answer with \texttt{deepest_vertex}_2, otherwise we follow the connection to a mini-tree root and solve the query there. If \(i\) is a mini-tree root, we answer immediately with the stored vertex \texttt{deepest_vertex}_1.

\textbf{function} \texttt{deepest_vertex}(i):
\begin{itemize}
\item if \(i.\tau_3 \neq 0\):
  \begin{itemize}
  \item \(d \leftarrow (i.\tau_1, i.\tau_2, \texttt{deepest_vertex}[i.\text{size}, i.\text{rep}, i.\tau_3])\) \quad \triangleright \text{Within micro-tree}
  \item if \texttt{is_ancestor}[i.\text{size}, i.\text{rep}, i.\tau_3, i.\text{dummy}_2]: \quad \triangleright \text{Type (3) connection?}
    \begin{itemize}
    \item \(d' \leftarrow \texttt{deepest_vertex}(\text{bottom}_2(i))\) \quad \triangleright \text{Outside}
    \end{itemize}
  \item if \(d.\tau_3 = i.\text{dummy}_2:\)
    \begin{itemize}
    \item return \(d'\)
    \end{itemize}
  \item else:
    \begin{itemize}
    \item return if \texttt{depth}(d) > \texttt{depth}(d') then \(d\) else \(d'\) \quad \triangleright \text{The deeper one}
    \end{itemize}
  \end{itemize}
\item else:
  \begin{itemize}
  \item return \texttt{d}
  \end{itemize}
\item else if \(i.\tau_2 \neq 0\):
  \begin{itemize}
  \item \(d \leftarrow (i.\tau_1, i.\texttt{deepest_vertex}_2.\tau_2, i.\texttt{deepest_vertex}_2.\tau_3)\)
  \item if \(d.\tau_3 = d.\text{dummy}_2:\) \quad \triangleright \text{Mini-tree dummy vertex?}
    \begin{itemize}
    \item return \texttt{deepest_vertex}(\text{dummy}_\text{down}(d)) \quad \triangleright \text{Recursion}
    \end{itemize}
  \item else:
    \begin{itemize}
    \item return \(d\)
    \end{itemize}
  \end{itemize}
\item else:
  \begin{itemize}
  \item return \texttt{i.deepest_vertex}_1
  \end{itemize}
\end{itemize}

The operation \texttt{deepest_vertex} is also used for supporting of \texttt{height}.

80
function \textsc{height}(i):
    \hspace{1em} d \leftarrow \textsc{deepest\_vertex}(i)
    \hspace{1em} \textbf{return} \hspace{0.5em} \text{depth}(d) - \text{depth}(i)

The operation \textsc{subtree\_size} is similar to \textsc{deepest\_vertex}. The answer is known for a mini-tree root. In case of a micro-tree root, we know the subtree size restricted to the mini-tree; we sum it with the subtree size of a potentially type (3) connected mini-tree if it exists in the subtree of \(i\). Finally, in case of a non-root, we use a look-up table, plus we add the size of anything connected by a dummy vertex, and we subtract 1 for the dummy vertex.

function \textsc{subtree\_size}(i):
    \hspace{1em} \textbf{if} \hspace{0.5em} i.\tau_3 \neq 0:
        \hspace{1em} s \leftarrow \textsc{subtree\_size}[i.\text{size},i.\text{rep},i.\tau_3]
        \hspace{1em} \textbf{if} \hspace{0.5em} \text{is\_ancestor}[i.\text{size},i.\text{rep},i.\tau_3,1]:
            \hspace{1em} \hspace{1em} \textbf{return} \hspace{0.5em} s - 1 + \textsc{subtree\_size}(\text{bottom}_2(i))
        \hspace{1em} \textbf{else}:
            \hspace{1.5em} \textbf{return} \hspace{0.5em} s
    \hspace{1em} \textbf{else if} \hspace{0.5em} i.\tau_2 \neq 0:
        \hspace{1em} \textbf{if} \hspace{0.5em} i.\text{dummy\_ancestor}:
            \hspace{1em} \hspace{1em} \textbf{return} \hspace{0.5em} i.\text{subtree\_size}_2 + \textsc{subtree\_size}(i.\text{bottom}_1,0,0)
        \hspace{1em} \textbf{else}:
            \hspace{1.5em} \textbf{return} \hspace{0.5em} i.\text{subtree\_size}_2
    \hspace{1em} \textbf{else}:
        \hspace{1.5em} \textbf{return} \hspace{0.5em} i.\text{subtree\_size}_1

4.2.5 Vertex and Leaf Ranks and Selects

We again augment the global structure and the structures of mini-trees and micro-trees with additional data to support new operations. We consider four distinct vertex numbering schemas and later use the same techniques to support operations on leaves restricted to a subtree.

It is crucial to account every vertex of the original tree once; we do so by introducing a relation \textit{to belong to}. We define the relation differently for each numbering schema.

**Pre-Order Numbering**

A vertex \(i\) with a canonical name \((\tau_1, \tau_2, \tau_3)\) belongs to a mini-tree \(\tau_1\) and a micro-tree \((\tau_1, \tau_2)\). Each vertex of the original tree belongs to exactly one mini-tree and one micro-tree.

A vertex in a mini-tree or a micro-tree is called the \textit{first} if it belongs to the mini-tree or micro-tree and has the lexicographically smallest name. We can find it by asking if the root belongs to the mini-tree or the micro-tree; if not, we set \(\tau_3 = 1\). Note that there can exist a micro-tree consisting of a single vertex, which is its root, that is shared with another micro-trees; in such case, the first vertex is undefined.

We store the explicit pre-order number \texttt{pre\_first}_1 of the first vertex of each mini-tree. We store the offset of the pre-order number \texttt{pre\_first}_2 from \texttt{pre\_first}_1
for the first vertex of each micro-tree while skipping the possible mini-tree connected via the type (3) connection.

When we compute the PRE_RANK, we need to determine which vertex is the first one in a mini-tree (micro-tree); this is achieved by comparing the canonical name of its root with the name of the name of the mini-tree (micro-tree).

If \( i \) is the first vertex in a mini-tree, we answer the query directly. If \( i \) is the first vertex in a micro-tree, we add its offset to the first vertex in the mini-tree, and also the subtree size of a possible type (3) connected mini-tree if it is before the current micro-tree. Otherwise, we determine the offset of \( i \) from the first vertex within the micro-tree and add compensate for the dummy vertex and its subtree if it is in the micro-tree before \( i \).

We state the algorithm in a more general form which does not assume that the PRE_RANK corresponds to the names of vertices; it will make the discussion of POST_RANK easier. The look-up table PRE_RANK has one additional argument which makes sure that the rank is counted from the first vertex in the micro-tree.

**function** PRE_RANK(\( i \)):

\[
\begin{align*}
\text{first}_1 &\leftarrow (i.\tau_1, 0, i.\tau_1 \neq i.\text{primary}_1) \\
\text{first}_2 &\leftarrow (i.\tau_1, i.\tau_2, i.\tau_2 \neq i.\text{primary}_2 \text{ or } (i.\tau_1 \neq i.\text{primary}_1 \text{ and } i.\tau_2 = 0)) \\
\text{if } i &= \text{first}_1: \quad \triangleright \text{ First in mini-tree} \\
& \quad \quad r \leftarrow i.\text{pre}_{\text{first}_1} \\
\text{else if } i &= \text{first}_2: \quad \triangleright \text{ First in micro-tree} \\
& \quad \quad r \leftarrow i.\text{first}_2 \\
& \quad \quad \text{if } i.\text{dummy}_1 \neq -1 \text{ and } i.\text{dummy}_1.\text{pre}_{\text{first}_2} < r: \quad \triangleright \text{ Type (3) connected mini-tree} \\
& \quad \quad \quad d \leftarrow \text{bottom}_1(i) \\
& \quad \quad \quad r \leftarrow r + \text{subtree}_\text{size}(d) \\
& \quad \quad \text{else:} \quad \triangleright \text{ Non-first} \\
& \quad \quad \quad r \leftarrow \text{pre}_{\text{rank}}[i.\text{size}, i.\text{rep}, i.\tau_3, \text{first}_2.\tau_3] \\
& \quad \quad \quad \text{if } i.\text{dummy}_2 \neq -1 \text{ and } \text{pre}_{\text{rank}}[i.\text{size}, i.\text{rep}, i.\text{dummy}_2, \text{first}_2.\tau_3] < r: \quad \triangleright \text{ Type (3) connection} \\
& \quad \quad \quad \quad d \leftarrow \text{bottom}_2(i) \\
& \quad \quad \quad \quad r \leftarrow r + \text{subtree}_\text{size}(d) \\
& \quad \quad \quad r \leftarrow r + \text{pre}_{\text{rank}}(\text{first}_2) \\
\text{return } r
\end{align*}
\]

We store a compressed array \( \text{pre}_\text{vertices}_1 \) of \( \tau_1 \) names for all vertices of the tree in pre-order; each vertex reports the \( \tau_1 \) name of the mini-tree to which it belongs. The space complexity of \( o(n) \) bits comes from the following lemma and the discussion in the beginning of the representation.

**Lemma 16.** Each \( \tau_1 \) name is the compressed array \( \text{pre}_\text{vertices}_1 \) occurs in at most three runs.

**Proof.** We describe all situation which can lead to the sequence being split into multiple runs.

1. A root vertex of a mini-tree can have type (2) connected mini-tree which appear before all its children.
2. A dummy vertex introduces an alien subtree inside the current one.
We store a similar compressed array $\text{pre}_{\text{vertices}}_2$ of $\tau_2$ names for each mini-tree; only the vertices belonging to the mini-tree are reported. The space complexity is $o(B)$ bits per mini-tree.

In the search, we first find the correct mini-tree and the remaining $r'$ within it; then the correct micro-tree and the remaining $r''$. In the end, we use a look-up table in the micro-tree with correction for the non-root first vertex and the possible dummy vertex. We can ignore type (3) connections as they were taken care of implicitly by the compressed arrays.

```plaintext
function PRE_SELECT(r):
    \( d_1 \leftarrow \text{pre}_{\text{vertices}}_1[r - 1] \) \( \triangleright \) Name of the mini-tree
    \( r' \leftarrow \text{rank}(\text{pre}_{\text{vertices}}_1, r - 1) \)
    \( d_2 \leftarrow d_1.\text{pre}_{\text{vertices}}_2[r' - 1] \) \( \triangleright \) Name of the micro-tree
    \( r'' \leftarrow \text{rank}(d_1.\text{pre}_{\text{vertices}}_2, r' - 1) \)

    \( \text{first}_2 \leftarrow (i.\tau_1, i.\tau_2, i.\tau_2 \neq i.\text{primary}_1 \text{ or } (i.\tau_1 \neq i.\text{primary}_1 \text{ and } i.\tau_2 = 0)) \)

    if \( \text{pre}_{\text{rank}}[d_2.\text{size}, d_2.\text{rep}, d_2.\text{dummy}_2, \text{first}_2.\tau_3] \leq r'':
        \( r'' \leftarrow r'' + 1 \) \( \triangleright \) Add the dummy vertex because of the table look-up
    return canonize(\( d_1, d_2, \text{pre}_{\text{select}}[d_2.\text{size}, d_2.\text{rep}, r'', \text{first}_2.\tau_3] \))
```

Ancestor Checking

We have already use a look-up table called IS_ANCESTOR which solved this query for vertices in a micro-tree. This table was used in the operation SUBTREE_SIZE which is used in PRE_RANK which is used for defining the general IS_ANCESTOR operation.

```plaintext
function IS_ANCESTOR(\( i_1, i_2 \)):
return \( \text{pre}_{\text{rank}}(i_1) \leq \text{pre}_{\text{rank}}(i_2) \leq \text{pre}_{\text{rank}}(i_1) + \text{subtree}_\text{size}(i_1) - 1 \)
```

Post-Order Numbering

The POST_RANK and POST_SELECT are very similar to PRE_*, however there are several differences which make them more complicated.

We use a different definition of belonging – a vertex belongs to the mini-tree and micro-tree with the lexicographically biggest name instead of the smallest one; we call them the *terminary mini-tree and micro-tree*. We introduce an analogy of the fields primary$_1$ and primary$_2$ called terminary$_1$ and terminary$_2$ which are used to navigate to the terminary mini-tree and micro-tree from the primary ones.

Since the first vertex which is assigned a post-order number in a mini-tree or micro-tree is its first leaf which could also be a dummy vertex, we store the explicit \( \text{post}_{\text{last}}_1 \) or offsetted \( \text{post}_{\text{last}}_2 \) post-order ranks of the last vertices in the mini-tree or a micro-tree. The value of \( \text{post}_{\text{last}}_2 \) is non-positive. Variables \( \text{last}_1 \) and \( \text{last}_2 \) are defined similarly to \( \text{first}_1 \) and \( \text{first}_2 \) using the definition of belonging.

We omit the algorithms here because they differ only in details.
DFUDS-Order Numbering

Lemma 17. The DFUDS-order number of a vertex can be determined by a recursive formula:

\[
\text{dfuds}_\text{anc}(i) = \sum_{a \in \text{ancestors}(i) \setminus \{i\}} \text{right}_\text{siblings}(a),
\]

\[
\text{dfuds}_\text{rank}(i) = \begin{cases} 
1 & \text{if } \text{is}_\text{root}(i); \\
\text{pre}_\text{rank}(i) + \text{dfuds}_\text{anc}(i) & \text{if } \text{child}_\text{rank}(i) = 1; \\
\text{dfuds}_\text{rank} (\text{child}_\text{first}(\text{parent}(i))) + \text{child}_\text{rank}(i) - 1 & \text{otherwise.}
\end{cases}
\]

Finding the \text{dfuds}_\text{rank} of a non-first child is easy and does not require any special consideration. In case of the first child, we already know its \text{pre}_\text{rank} so the only thing left is to compute \text{dfuds}_\text{anc} in constant time.

Let \( v_1 \) and \( v_2 \) be ancestors of \( i \) such that:

\[
\begin{align*}
v_1 \in \text{ancestors}(i) \text{ and } & \text{parent}(v_1) = \text{root}_1(i), \\
v_2 \in \text{ancestors}(i) \text{ and } & \text{parent}(v_2) = \text{root}_2(i).
\end{align*}
\]

We split the precomputed \( \text{dfuds}_\text{anc}(i) \) into several parts, which are in the most general case:

\[
\text{dfuds}_\text{anc}(i) = \text{dfuds}_\text{anc}(\text{parent}(v_1)) + \text{dfuds}_\text{anc}(v_1) - \text{dfuds}_\text{anc}(\text{parent}(v_1)) + \text{dfuds}_\text{anc}(v_2) - \text{dfuds}_\text{anc}(\text{parent}(v_2)) + \text{dfuds}_\text{anc}(i) - \text{dfuds}_\text{anc}(v_2).
\]

If \( v_1 = v_2 \), then \( C \) and \( D \) do not exist.

When we refer to the mini-tree structure, it is \( i.\tau_1 \); the micro-tree structure is \( (i.\tau_1, i.\tau_2) \).

\begin{itemize}
\item \text{A} It is the answer for a mini-tree root; it is stored in the mini-tree structure.
\item \text{B} Reduced to the number of right siblings of a mini-tree root; stored in the mini-tree structure.
\item \text{C} Since the right siblings of \( v_1 \) can span over multiple mini-trees, we split \( C \) into two parts \( C_1 + C_2 = C \) such that:

\[
C_1 = \sum_{a \in \cdots} |\{s : s \in \text{right}_\text{siblings}(a) \text{ and } s.\tau_1 \neq a.\tau_1\}|,
\]

\[
C_2 = \sum_{a \in \cdots} |\{s : s \in \text{right}_\text{siblings}(a) \text{ and } s.\tau_1 = a.\tau_1\}|.
\]

\( C_1 \) is stored in the mini-tree structure; \( C_2 \) in the micro-tree structure.
\item \text{D} Reduced to the number of right siblings of a micro-tree root; stored in a micro-tree structure. As the micro-tree root is not a mini-tree root, all its right siblings are in the same mini-tree.
\end{itemize}
The value $E$ cannot be fully computed by a look-up table since it account for vertices which can be outside the micro-tree and even outside the mini-tree. We split it into three parts $E_1 + E_2 + E_3 = E$ such that:

$$E_1 = \sum_{a \in \ldots} |\{s : s \in \text{right}_\text{siblings}(a) \text{ and } s.\tau_1 \neq a.\tau_1\}|,$$

$$E_2 = \sum_{a \in \ldots} |\{s : s \in \text{right}_\text{siblings}(a) \text{ and } s.\tau_1 = a.\tau_1 \text{ and } s.\tau_2 \neq a.\tau_2\}|,$$

$$E_3 = \sum_{a \in \ldots} |\{s : s \in \text{right}_\text{siblings}(a) \text{ and } s.\tau_1 \neq a.\tau_1 \text{ and } s.\tau_2 = a.\tau_2\}|.$$

$E_1$ is stored in the mini-tree structure; $E_2$ in the micro-tree structure; $E_3$ is provided by a look-up table. Note that $E_1$ needs to be stored only for root micro-trees, and that it has the same value for all of them within the same mini-tree.

The algorithm for the operation $\text{DFUDS\_RANK}$ consists mostly of the computation of $\text{DFUDS\_ANC}$. Six cases of the position of the vertex $i$ in the tree has to be considered.

Function $\text{DFUDS\_RANK}(i)$:

if $\text{is\_root}(i)$:
  return 1
else if $\text{child\_rank}(i) > 1$:
  return $\text{dfuds\_rank}(\text{child\_first(parent(i))) + child\_rank(i)} - 1$
else:
  $s \leftarrow \text{pre\_rank}(i)$
  if $i.\tau_3 = 0$:
    $p \leftarrow \text{parent}(i)$
    if $p.\tau_3 \neq 0$:
      $r \leftarrow \text{root2}(i)$  \(\triangleright\) Descendant of a micro-tree root
      $e \leftarrow i.E_2 + E_3[i.\text{size}, i.\text{rep}, i.\tau_3]$  
      return $s + i.A + i.B + i.C_1 + i.C_2 + i.D + e$
    else:
      \(\triangleright\) Descendant of a mini-tree root
      $e \leftarrow i.E_1 + i.E_2 + E_3[i.\text{size}, i.\text{rep}, i.\tau_3]$  
      return $s + i.A + i.B + e$
  else if $p.\tau_2 \neq 0$:
    \(\triangleright\) Child of a micro-tree root
    return $s + i.A + i.B + i.C_1 + i.C_2 + i.D$
  else:
    \(\triangleright\) Child of a mini-tree root
    return $s + i.A + i.B$
else if $i.\tau_2 \neq 0$:
  \(\triangleright\) Micro-tree root
  return $s + i.A + i.B + i.C_1 + i.C_2$
else:
  \(\triangleright\) Mini-tree root
  return $s + i.A$

For the $\text{DFUDS\_SELECT}$ operation, we reuse the definitions of the pre-order belonging and the first vertex. We store a compressed array $\text{dfuds\_vertices}_1$, which is similar to the pre-order case; it contains the $\tau_1$ names of all vertices of the tree in DFUDS-order.
Lemma 18. The array $\text{dfuds}\_\text{vertices}_1$ contains at most four runs of each $\tau_1$ name.

Proof. For all vertices $v$ of a mini-tree except for a constant number of exceptions, it holds that the immediately preceding vertex in the array has the same $\tau_1$ name. The exceptions are:

1. the mini-tree root; it is the first vertex in DFUDS-order;
2. the $(\tau_1, 0, 1)$, unless the root was the only child of its parent;
3. the vertex after the mini-tree dummy vertex; the dummy vertex is canonized and the $\tau_1$ name of the connected mini-tree is used instead;
4. the vertex after last vertex of the connected mini-tree via a dummy vertex.

Each of these vertices start a run, therefore there are at most four runs. □

The space complexity of the compressed array is $o(n)$ bits. We do the same on the mini-tree level with $\text{dfuds}\_\text{vertices}_2$ which requires $o(B)$ bits.

The operation $\text{DFUDS}\_\text{SELECT}$ is the same $\text{PRE}\_\text{SELECT}$, including the corrections in micro-trees.

In-Order Numbering

Lemma 19. We can establish a recursive formula for IN_RANK, which is inspired by DFUDS_RANK:

$$\text{in}_\text{anc}(i) = \sum_{a \in \text{ancestors}(i)} \sum_{l \in \text{left}_\text{siblings}(a)} (1 + \text{in}_\text{size}(l)),$$

$$\text{in}_\text{rank}(i) = \begin{cases} -1 & \text{if } \text{degree}(i) \leq 1; \\ \text{in}_\text{anc}(i) + \text{in}_\text{size}(\text{child}_\text{first}(i)) + 1 & \text{otherwise}. \end{cases}$$

IN_SIZE returns the total number of in-order numbers assigned in a subtree of a given vertex.

Proof. An in-order number is assigned to a vertex which has a degree greater than 1; we can therefore ignore vertices of degree 0 or 1.

We calculate how many in-order numbers have been assigned until vertex $i$ is given its first in-order number. The subtrees of left siblings of its ancestors have been fully processed, as has been the subtree of the first child of $i$. The $+1$ in IN_ANC is for in-order numbers already assigned to parent$(a)$ for its left children (left siblings of $a$). □

The function IN_SIZE is implemented the same way as SUBTREE_SIZE is. There are precomputed values for mini-tree and micro-tree roots; a look-up table answers queries within a micro-tree. The only difference is that subtraction of 1 for a dummy vertex is not needed as leaves are not assigned an in-order number.

We focus on the function IN_ANC which we again split into several parts for which we store their precomputed value in the min-tree or micro-tree structures.

Let $v_1$ and $v_2$ be ancestors of $i$ such that:

$$v_1 \in \text{ancestors}(i) \text{ and } \text{parent}(v_1) = \text{root}_1(i),$$

$$v_2 \in \text{ancestors}(i) \text{ and } \text{parent}(v_2) = \text{root}_2(i).$$
We split the precomputed $\text{in}_\text{anc}(i)$ into several parts, which are in the most general case:

\[
\text{in}_\text{anc}(i) = \text{in}_\text{anc}(\text{parent}(v_1)) + \text{in}_\text{anc}(v_1) - \text{in}_\text{anc}(\text{parent}(v_1)) + \text{in}_\text{anc}(\text{parent}(v_2)) - \text{in}_\text{anc}(v_1) + \text{in}_\text{anc}(v_2) - \text{in}_\text{anc}(\text{parent}(v_2)) + \text{in}_\text{anc}(i) - \text{in}_\text{anc}(v_2).
\]

(A)

(B)

(C)

(D)

(E)

If $v_1 = v_2$, then $C$ and $D$ do not exist.

When we refer to the mini-tree structure, it is $i.\tau_1$; the micro-tree structure is $(i.\tau_1, i.\tau_2)$.

A A precomputed value for a mini-tree root; it is stored in a mini-tree structure.

B The value $B$ is further split into three parts $B_1 + B_2 + B_3 = B$ such that:

\[
B_1 = \sum_{l \in \text{left}_\text{siblings}(v_1)} \sum_{l.\tau_1 \neq v_1.\tau_1} (1 + \text{in}_\text{size}(l))
\]

\[
B_2 = \sum_{l \in \text{left}_\text{siblings}(v_1)} \sum_{l.\tau_1 = v_1.\tau_1, l.\tau_2 \neq v_1.\tau_2} (1 + \text{in}_\text{size}(l))
\]

\[
B_3 = \sum_{l \in \text{left}_\text{siblings}(v_1)} \sum_{l.\tau_1 = v_1.\tau_1, l.\tau_2 = v_1.\tau_2} (1 + \text{in}_\text{size}(l)).
\]

$B_1$ is stored in the mini-tree structure; $B_2$ is stored in the micro-tree structure. If $v_1 = v_2$ then $B_3$ is computed by a look-up table, otherwise it is stored in the micro-tree structure.

C The value $C$ is stored in a micro-tree structure.

D $D$ is split into two parts $D_2 + D_3 = D$, similar to how $B$ is:

\[
D_2 = \sum_{l \in \text{left}_\text{siblings}(v_2)} \sum_{l.\tau_1 = v_2.\tau_1, l.\tau_2 \neq v_2.\tau_2} (1 + \text{in}_\text{size}(l))
\]

\[
D_3 = \sum_{l \in \text{left}_\text{siblings}(v_2)} \sum_{l.\tau_1 = v_2.\tau_1, l.\tau_2 = v_2.\tau_2} (1 + \text{in}_\text{size}(l)).
\]

$D_2$ is stored in the micro-tree structure, $D_3$ is handled by a look-up table. As the micro-tree root is not a mini-tree root, all its right siblings are in the same mini-tree.

E It stored in a look-up table.

Note that the in\_size of any type (3) connected mini-tree is not accounted in the fields stored in micro-trees, and any type (3) connected micro-tree is not
accounted by the look-up table. They can easily be detected by checking the pre-order number of the dummy vertices, and added later.

function IN_RANK(i):
    if degree(i) ≤ 1:
        return -1
    else:
        s ← in_size(child_first(i)) + 1
        if i.τ3 ≠ 0:
            p ← parent(i)
            if p.τ3 ≠ 0:
                r ← root_2(i)
                if r.τ2 ≠ 0:
                    s ← s + in_size(bottom_2(i)) + 1
                else:
                    p ← parent(i)
                    if p.τ2 ≠ 0:
                        d ← i.D2 + D_3[i.size, i.rep, i.τ3], e ← E[i.size, i.rep, i.τ3]
                        s ← s + i.A + i.B1 + i.B2 + i.B3 + i.C + d + e
                    else:
                        e ← E[i.size, i.rep, i.τ3]
                        s ← s + i.A + i.B1 + i.B2 + B_3[i.size, i.rep, i.τ3] + e
            else if p.τ2 ≠ 0:
                s ← s + i.A + i.B1 + i.B2 + B_3[i.size, i.rep, i.τ3]
            else:
                s ← s + i.A
                if i.dummy_2 ≠ −1 and i.dummy_1 ≠ i.τ2:
                    if pre_rank(bottom_2(i)) < pre_rank(i):
                        s ← s + in_size(bottom_2(i))
                    else:
                        s ← s + i.A + i.B1 + i.B2 + i.B3 + i.C
                else if i.τ2 ≠ 0:
                    s ← s + i.A + i.B1 + i.B2 + i.B3 + i.C
                else:
                    s ← s + i.A
                    if i.dummy_1 ≠ −1:
                        if pre_rank(bottom_1(i)) < pre_rank(i):
                            s ← s + in_size(bottom_1(i))
                        else:
                            s ← s + i.A
    if i.dummy_2 ≠ −1 and i.dummy_1 ≠ i.τ2:
        Add it back
    else:
        Micro-tree root
        Mini-tree root
        Add it back

The IN_SELECT is similar to PRE_SELECT; it has two levels of compressed arrays which we use to navigate to the correct micro-tree. The array in_vertices contains τ_1 names for all vertices of the tree which have been assigned an in-order number in all their instances, therefore a vertex v appears degree(v) − 1 times.

We define more precisely which τ_1 name is stored in the compressed array for each instance of v. If v is a non-root, the v.τ_1 is stored.

If v is a micro-tree root, then we look closer at the mini-trees which contain the children of v:

(1) If v has children, it is a root of several type (1) connected mini-trees; an extreme case is that v is not shared, in which case there is only one such mini-tree. The sequences of children of v in these mini-trees are uninterrupted by children in other mini-trees. There can be type (2) connected mini-trees, each of them contributing with one child to v:
(a) before the first type (1) connected mini-tree; The name \(v.\tau_1\) is reported for all visits of \(v\) from a child in such mini-tree.

(b) between two type (1) connected mini-trees; The name of the preceding type (1) connected mini-tree is reported.

(c) after the last type (1) connected mini-tree. The name of the preceding type (1) connected mini-tree is reported.

For visits of \(v\) which result in an in-order number being assigned to \(v\) from type (2) connected mini-trees of \(v\), we store \(v.\tau_1\) in the case (a), and the \(\tau_1\) name of the preceding type (1) connected mini-tree in the cases (b) and (c). In the cases (a), we also store the number of them in a field \(left\_type2_1\) in the primary mini-tree; the value is 0 for non-primary mini-trees.

(2) If the mini-tree containing \(v\) does not contain any other vertex, then all its children are roots of type (2) connected distinct mini-trees. The name \(v.\tau_1\) is reported and it is treated as the case (c) in the algorithm.

The name reported for visits from children in a type (1) connected mini-tree \(t\) is simply \(t\).

**Lemma 20.** The number of runs in the compressed array \(in\_vertices_1\) is \(O\left(\frac{n}{B}\right)\).

**Proof.** There are only the following cases when a vertex \(v\) has a different \(\tau_1\) name than its immediately preceding vertex \(u\) in the array \(in\_vertices_1\).

- \(v\) is the first vertex in its mini-tree which is assigned an in-order number. This happens at most once per mini-tree.
- \(v\) is a parent of a mini-tree dummy vertex which leads to a subtree containing \(u\). The whole subtree is exhausted before DFS continues with the mini-tree containing \(v\), therefore this happens at most once per mini-tree.
- \(v\) is a root and it was visited from a root of a type (2) connected mini-tree. There are at most \(O\left(\frac{n}{B}\right)\) type (2) connections, and so are these visits.
- \(v\) is a root and it was visited from the first child in a non-primary type (1) connected mini-tree. This happens once per a non-primary mini-tree, whose number is bounded by \(O\left(\frac{n}{B}\right)\).

As the total number of runs is bounded by \(O\left(\frac{n}{B}\right)\), the size required to store such array is \(o(n)\) bits.

Using the compressed array \(in\_vertices_1\) we can navigate to the mini-tree which contains the vertex with the in-order number \(r\), however it might not be the primary mini-tree. It can happen that \(\tau_1\) was used in \(in\_vertices_1\) in one of the following cases, which we handle separately before searching for a micro-tree.

- If the mini-tree has the only one vertex, we simply return its name. We can therefore assume that the mini-tree root has children in the same mini-tree.
- If the mini-tree is the primary one, then the first \(left\_type2_1\) occurrences of \(\tau_1\) in \(in\_vertices_1\) are due to type (2) connections. From this we know that they refer to the mini-tree root.

89
If the mini-tree \( t \) has \( k \) type (b) or (c) children, then the last \( k \) occurrences of \( t \) in \( \text{in}_\text{vertices}_1 \) are due to visits of them. If there exists a mini-tree \( s > t \) such that it shares its root with \( t \), then one extra occurrence is due to visit of \( s \). All these occurrences correspond to the root being assigned an in-order number.

The search continues in a compressed array \( \text{in}_\text{vertices}_2 \) which is defined similarly to \( \text{in}_\text{vertices}_1 \). All occurrences are due to in-order numbers which were assigned as the result of returning from and diving into vertices in the mini-tree. The size of the array is therefore bounded, as is the number of runs; the space complexity is \( o(B) \) bits.

Using the compressed arrays \( \text{in}_\text{vertices}_1 \) and \( \text{in}_\text{vertices}_2 \), we navigate to the correct micro-tree. We handle the first two special cases in the micro-tree the same we as in the mini-tree. The last case is handled by comparison with the result of the \( \text{IN}_\text{SIZE} \) look-up table; this look-up table comes from the implementation of the \( \text{IN}_\text{SIZE} \) function. The final vertex is then found by a look-up table.

In order to check the bounds of \( \text{IN}_\text{SELECT} \) before it is processed, we use the field \( \text{in}_\text{size} \) which is stored in the root mini-tree and contains the number of all in-order numbers assigned to the vertices of the tree.

```plaintext
function \text{IN}_\text{SELECT}(r):
    \( d_1 \leftarrow \text{in}_\text{vertices}_1[r - 1] \)
    \( r' \leftarrow \text{rank}(\text{in}_\text{vertices}_1, r - 1) \)
    if \( |d_1\text{.micro_tree_offsets}| = 1 \) and \( d_1[0]\text{.size} = 1 \):
        \( \triangleright \) A single vertex
        \text{return root}_1(d_1)
    else if \( r' \leq d_1\text{.left_type}_2 \):
        \( \triangleright \) Left type (2) connections
        \text{return root}_1(d_1)
    else:
        \( r' \leftarrow r' - d_1\text{.left_type}_2 \)
        if \( r' \geq \text{size}(d_1\text{.left_type}_2) \):
            \( \triangleright \) Right type (2) or type (1) connection
            \text{return root}_1(d_1)
        else:
            \( \triangleright \) The same for micro-trees
            \( d_2 \leftarrow d_1\text{.in}_\text{vertices}_2[r' - 1] \)
            \( r'' \leftarrow \text{rank}(d_1\text{.in}_\text{vertices}_1, r' - 1) \)
            if \( d_2\text{.size} = 1 \):
                \( \triangleright \) A single vertex micro-tree
                \text{return root}_2(d_1, d_2)
            else if \( r'' \leq d_2\text{.left_type}_2 \):
                \( \triangleright \) Left type (2) connections
                \text{return root}_2(d_1, d_2)
            else:
                \( r'' \leftarrow r'' - d_2\text{.left_type}_2 \)
                if \( r'' \geq \text{in}_\text{size}[d_2\text{.size}, d_2\text{.rep}, 0] \):
                    \( \triangleright \) Right (2) or right (1)
                    \text{return root}_2(d_1, d_2)
                else:
                    \text{return canonize}(d_1, d_2, \text{in}_\text{select}[d_2\text{.size}, d_2\text{.rep}, r''])
```

90
Leaf Operations

All the LEAF_* operations are analogies of operations which we have shown before. The global LEAF_RANK and LEAF_SELECT operations are similar to PRE_RANK and PRE_SELECT. We extend them to be parametrized by the root of a subtree to which they are restricted.

We support the parametrized operations LEAF_SIZE the same way we support SUBTREE_SIZE, and LEAF_FIRST which returns the first leaf in the subtree of a given vertex the same way as DEEPEST_VERTEX.

We can then restrict the leaf operations to a subtree:

\[
\text{function LEAF_RANK}(a, i) := \text{leaf_rank}(i) - \text{leaf_rank}(\text{leaf_first}(a)) + 1
\]

\[
\text{function LEAF_SELECT}(a, i) := \text{leaf_select}(i +\text{leaf_rank}(\text{leaf_first}(a)) - 1)
\]

4.2.6 Ancestral Operations

We define a tree \(T_1\) whose nodes are roots of primary mini-trees. A node \(u\) is a parent of node \(v\) if \(v.\text{parent}_1 = u\). The tree \(T_1\) has \(O\left(\frac{n}{B}\right)\) nodes.

Similarly we define a tree \(T_2\) for each mini-tree \(t\) which consists of nodes corresponding to primary micro-tree roots within the mini-tree \(t\). A node \(u\) is a parent of node \(v\) in \(T_2\) if \(v.\text{parent}_1 = u\). This tree \(T_1\) has \(O\left(\frac{B}{b}\right)\) nodes.

Lowest Common Ancestor

The operation LCA will be solved on three levels: micro-tree, mini-tree and the whole tree. If \(i_1\) and \(i_2\) are within the same micro-tree we use a look-up table. Otherwise we use an LCA structure which uses the techniques from lemma 5.

Lemma 21. We can solve the LCA operation on the trees \(T_1\) and \(T_2\) using indices of \(o(n)\) and \(o(B)\) bits.

Proof. We first assume a general tree \(T\) with \(p\) nodes. We define an array \(E\) which contains nodes as they are visited during an Eulerian tour starting in the root. The array \(E\) contains \(2p - 1\) nodes, which corresponds to every vertex being visited from each of its neighbors plus one for the root where the tour starts and ends. The answer to \(lca(u, v)\) is the shallowest node between the first occurrence of \(u\) and the first occurrence of \(v\) in the array \(E\). We can therefore reduce the task of finding the lowest common ancestor to task of finding the minimum value in the given range in an array \(D[i] = \text{depth}(E[i])\). The position of the minimum can be found by a range minimum query using a precomputed (not look-up) table as in lemma 5.

In each node of the tree, we store its depth and the index of its the first occurrence in the array \(E\), which we do not store anywhere. The precomputed table contains directly names of the nodes; it requires requires \(p \log^2 p\) bits. In the case of two overlapping intervals we simply compare the depths of the two candidates.

In case of \(T_1\), we use the field \(\text{depth}_1\) since the relation of one node being an ancestor of the other is preserved. The name of the node is its \(\text{primary}_1\) name.
The tree \( T_1 \) has \( O \left( \frac{n}{B} \right) \) nodes, so the space required is \( O \left( \frac{n}{B} \log^2 n \right) = o(n) \) bits for \( c \geq 3 \). (c is the constant from definition of the size \( B \).)

In case of \( T_2 \), we use \( \text{depth}_2 \) for determining the depth and \( \text{primary}_2 \) as the name of a node. The tree \( T_2 \) has \( O \left( \frac{B}{b} \right) \) nodes, which results in space \( O \left( \frac{B}{b} \log^2 B \right) = o(B) \) bits.

If the vertices are in the same mini-tree, we solve it by querying \( T_2 \) for the lowest common ancestor using the lemma 21 (function \( \text{ANC SEARCH}_1 \)); in case when they are in different mini-trees, we query the tree \( T_1 \) (function \( \text{ANC SEARCH}_2 \)).

There are two special cases which need to be addressed in the algorithm.

1. If the vertices \( i_1 \) and \( i_2 \) are in mini-trees or micro-trees which share their root, the answer is the root.

2. If the vertices are in unaffiliated mini-trees, the reduction to LCA of their roots might not be correct if the answer was the root of \( i_1 \), without loss of generality. We need to check if the path between the vertices \( i_1, i_2 \) uses the mini-tree dummy vertex in the mini-tree of \( i_1 \), which we test by the operation \( \text{is ancestor} \). In such case, we replace \( i_2 \) by the dummy vertex and proceed as before.

The same applies for reduction to roots of micro-trees. However, we are only interested in micro-tree dummy vertices.

```python
function \( \text{LCA}(i_1, i_2) \):
    if \( i_1.\tau_1 = i_2.\tau_1 \):
        \( \triangleright \) The same mini-tree
        if \( i_1.\tau_2 = i_2.\tau_2 \):
            \( \triangleright \) The same micro-tree
            return \( \text{canonize}(i_1.\tau_1, i_1.\tau_2, \text{lca}[i_1.\text{size}, i_1.\text{rep}, i_1.\tau_3, i_2.\tau_3]) \)
        else if \( i_1.\text{primary}_2 = i_2.\text{primary}_2 \):
            \( \triangleright \) Type (1) connected micro-trees
            return \( \text{root}_2(i_1) \)
        else:
            \( \triangleright \) Unaffiliated micro-trees
            if \( i_1.\text{dummy}_2 \neq -1 \) and \( i_1.\tau_2 \neq i_1.\text{dummy}_1 \):
                if \( \text{is ancestor}(\text{bottom}_2(i_1), i_2) \):
                    \( \triangleright \) Test type (3) connection
                    return \( \text{lca}(i_1, \text{dummy}_2(i_1)) \)
            if \( i_2.\text{dummy}_2 \neq -1 \) and \( i_2.\tau_2 \neq i_2.\text{dummy}_1 \):
                \( \triangleright \) Symmetrical
                return \( \text{lca}(i_2, \text{dummy}_2(i_2)) \)
        return \( \text{root}_2(i_1.\tau_1, \text{lca}(i_1.\tau_1, i_1.\text{primary}_2, i_2.\text{primary}_2)) \)
    else if \( i_1.\text{primary}_1 = i_2.\text{primary}_1 \):
        \( \triangleright \) Type (1) connected micro-trees
        return \( \text{root}_1(i_1) \)
        \( \triangleright \) Mini-tree root
    else:
        \( \triangleright \) Unaffiliated mini-trees
        if \( i_1.\text{dummy}_1 \neq -1 \):
            if \( \text{is ancestor}(\text{bottom}_1(i_1), i_2) \):
                \( \triangleright \) Test type (3) connection
                return \( \text{lca}(i_1, \text{dummy}_1(i_1)) \)
        if \( i_2.\text{dummy}_2 \neq -1 \):
            \( \triangleright \) Symmetrical
            return \( \text{lca}(i_2, \text{dummy}_1(i_2)) \)
        return \( \text{root}_1(\text{lca}(i_1.\text{primary}_1, i_2.\text{primary}_1)) \)
```

92
**Level Ancestor**

If the desired vertex \( j \in \text{ancestors}(i) \) and \( \text{depth}(j) = \text{depth}(i) - d \) is not contained in the current mini-tree, we use the tree \( T_1 \) to find the correct mini-tree. We follow the same way for micro-trees, and finally find the vertex \( j \) using a look-up table in the correct micro-tree.

When we search the tree \( T_1 \), we not only want to know the name of the mini-tree \( t \) which contains \( j \), but also the the lowest ancestor \( j' \) in it. The lowest ancestor \( j' \) in a mini-tree \( t \) can be: (1) its root if \( i \) is in a subtree of a type (2) connected mini-tree; (2) parent of its dummy vertex if \( i \) is a subtree of a type (3) connected mini-tree. Since each mini-tree has at most one mini-tree dummy vertex, it is easy to check which option occurs in constant time.

We use the same technique as for \( \text{LRM}_{-}\text{SEARCH} \) in section 4.1.4 to find the correct mini-tree. Each node \( i \) has already been assigned its minimum depth in field \( \text{depth}_1 \). We first use a tiny compressed array \( J_i \) and then a collection of compressed arrays \( L \).

The same structure is built for the tree \( T_2 \) where the depths are \( \text{depth}_2 \). We call the search functions \( \text{ANC}_{-}\text{SEARCH}_1 \) and \( \text{ANC}_{-}\text{SEARCH}_2 \).

```
function \text{LEVEL\_ANCESTOR}(i, d):
  v ← \text{depth}(i) − d
  if \( v < 0 \):
    return −1
  if \( v < \text{depth}(\text{root}_1(i)) \):
    \( t ← \text{anc\_search}_1(\text{root}_1(i), v) \) ▷ Wrong mini-tree
    if \( \text{is\_ancestor}(\text{bottom}_1(t), i) \):
      \( i ← \text{parent}(\text{dummy}_1(t)) \) ▷ Type (3) connection
    else:
      \( i ← \text{root}_1(t) \) ▷ Type (2) connection
  if \( v < \text{depth}(\text{root}_2(i)) \):
    \( t ← \text{anc\_search}_2(i.\tau_1, \text{root}_2(i), v) \) ▷ Wrong micro-tree
    if \( \text{is\_ancestor}(\text{bottom}_2(t), i) \):
      \( i ← \text{parent}(\text{dummy}_2(t)) \) ▷ Type (3) connection
    else:
      \( i ← \text{root}_2(t) \) ▷ Type (2) connection
  return \text{canonize}(i.\tau_1, i.\tau_2, \text{level\_ancestor}[i.\text{size}, i.\text{rep}, i.\tau_3, \text{depth}(i) − v])
```

**4.2.7 Level Operations**

**Level First**

We first show how to support the operation \( \text{LEVEL\_FIRST} \) which is restricted to the subtree of a vertex \( i \). The operation \( \text{LEVEL\_LAST} \) is analogous, and we address the differences at the end.

For each mini-tree root \( i \) we define an array \( lfirst_1 \):

\[
i.\text{lfirst}_1[j] = \arg\min_{\text{pre\_rank}(v) \geq \text{pre\_rank}(i), \text{depth}(v) = \text{depth}(i) + j} \text{pre\_rank}(v).
\]

93
This array contains level-first descendants of $i$ for $j \leq \text{height}(i)$ as it has the correct depth and it is the first one such.

We cannot afford to store the arrays $lfirst_i$, however we observe the following fact: If for two mini-tree roots $i$, $j$ and level $d$ holds:

$$i.lfirst_i[d - \text{depth}(i)] = j.lfirst_i[d - \text{depth}(j)],$$

then for $k \geq 0$ holds that:

$$i.lfirst_i[d - \text{depth}(i) + k] = j.lfirst_i[d - \text{depth}(j) + k].$$

This is basically the same property which holds for the arrays $brm_i$ in section 4.1.4. Before we use it, we formulate and prove the following lemma.

**Lemma 22.** There are at most three runs for each $\tau_1$ name in the array $lfirst_i$.

**Proof.** Let’s assume that the root $v$ of a mini-tree $t$ is a level-first descendant of $i$, then $v.\tau_1$ is in the array. We also assume that $v$ has children in the same mini-tree. There are two possibilities how a run of level-first descendants can be interrupted:

1. We look at the first child of $v$. If it is not in the same mini-tree, then $v$ must have some type (2) connected mini-trees $L$ to the left. If $h = \max_{l \in L} \text{height}(l)$ is less than the height of $t$ (restricted to $t$), then the $v.\tau_1$ will appear in the array once all subtrees of $L$ end.

2. If a level-first descendant of $v$ is a mini-tree dummy vertex, then the level-first descendants of its mini-tree $s$ will be in the array. If the $h = \text{depth}(s) - \text{dep}(v) + \text{height}(s)$ is less than the height of the mini-tree $t$ (restricted to $t$), then the $v.\tau_1$ will appear in the array once the subtree of $s$ ends.

These two causes of interruption can overlap, however they can cause at most three runs of $v.\tau_1$.

If the root was not the level-first descendant but another vertex of the mini-tree $t$ occurs in the array, we can apply the same reasoning about the second type of interruption. That proves that such $\tau_1$ can appear in at most two runs. \hfill $\square$

We build a tree $T_1$ as a trie from the reversed arrays $lfirst_i$ with an additional artificial root. We compress the tree by contracting paths between nodes which are either:

1. The artificial root node.

2. Nodes corresponding to mini-tree roots. All leaves of $T_1$ are mini-tree roots, however not all mini-tree roots are leaves in $T_1$.

3. Branching nodes in the tree $T_1$. Their number is bounded by the number of leaves of $T_1$.

4. Nodes corresponding to vertices in the arrays $lfirst_i$ such that their predecessor has a different $\tau_1$ name.
Not counting branching nodes, there are at most three vertices for each mini-tree, which follows from the previous lemma. The number of nodes in the compressed tree is \( O \left( \frac{n}{B} \right) \).

We associate each node \( n \) with the depth of the vertex which it represents in the original tree. We store the information about the nodes in an array; each node knows its \( \tau_1 \) name, its depth, and its pre-order number \( k \) in \( T_1 \). We also store the pre-order number \( k \) of the node which corresponds with the mini-tree root as a field \( \text{lfirst}_k \) in a mini-tree structure.

Since this tree has increasing values on paths from leaves to the root, we use the same structures as for \( T_{LRM} \) in section 4.1.4.

- A tiny compressed array \( J_n \) for each node \( n \). The \( J \) contains pre-order numbers \( k \) of the nodes at distances which are powers of two. Contrary to the definition in section 4.1.4, we use the operation \( \text{pred} \) for definition of the compressed array.
- A collection of compressed arrays \( L \) storing all ladders. Similarly to \( J \), the operation \( \text{pred} \) is used.
- An array \( \text{ladder} \) which contains information about to which ladder each node belongs.

These structures take \( O \left( \frac{n}{B} \log^2 n \right) = o(n) \) bits for \( c \geq 3 \).

We use the algorithm \( \text{LRM\_SEARCH} \) which returns the first ancestor node \( j \) in the tree \( T_1 \) such that \( \text{depth}(j) \) less than or equal to the desired one. In our case, the function returns the \( \tau_1 \) name of the node. Note that in a general case, we should check whether the node which we found is a descendant of the queried node, which can be done by \( \text{is\_ANCESTOR} \). However, in the algorithm as we formulate it, we check the bounds beforehand, so it is not necessary. This concludes the search for the correct mini-tree.

We build a similar structure for querying level-first descendants of micro-tree roots for each mini-tree. The searches are processed by functions \( \text{LD\_SEARCH}_1 \) and \( \text{LD\_SEARCH}_2 \). Finally, we use a look-up table to find the answer within the correct micro-tree.

The algorithm tests two alternatives and chooses the one with smaller pre-order number.

1. If the query starts with a mini-tree root, the correct mini-tree is found, then the correct micro-tree is found, and finally the correct vertex is found.

2. If the query starts with a micro-tree root, a micro-tree is found within a potentially incorrect mini-tree, then a vertex is found. The mini-tree dummy vertex is check, as it could contain the correct answer; a recursive call causes triggers the option (1).

3. If the query starts with a non-root, a vertex is found in a look-up table. The dummy vertex in the micro-tree is checks; a recursive call leads to options (1) or (2).

The algorithm uses recursion of a constant depth.
function `LEVEL_FIRST`\(i, d\):
   if \(d < \text{depth}(i)\) and \(d > \text{depth}(i) + \text{height}(i)\):
      return \(-1\) \(\triangleright\) Answer does not exist
   c ← \(-1\) \(\triangleright\) Alternative candidate
   if \(i.\tau_2 = 0\) and \(i.\tau_3 = 0\):
      \(i \leftarrow \text{root}_1(\text{ld}\_\text{search}_1(i.\text{lfirst}_k_1, d))\) \(\triangleright\) Find mini-tree (root)
   if \(i.\tau_3 = 0\): \(\triangleright\) Micro-tree root
      if \(i.\tau_2 \neq 0\) and \(i.\tau_3 = 0\) and \(i.\text{dummy\_ancestor}\):
         \(c \leftarrow \text{level\_first} (\text{bottom}_1(i))\) \(\triangleright\) Check mini-tree dummy vertex
         \(i \leftarrow \text{root}_2(i.\tau_1, \text{ld}\_\text{search}_2(i.\tau_1, i.\text{lfirst}_k_2, d))\) \(\triangleright\) Find micro-tree (root)
   if \(i.\tau_3 \neq 0\) and \(\text{is\_ancestor}[i.\text{size}, i.\text{rep}, i.\tau_3, i.\text{dummy}_2]\):
      \(c \leftarrow \text{level\_first} (\text{bottom}_2(i))\) \(\triangleright\) Check dummy vertex
      \(i \leftarrow \text{level\_first}[i.\text{size}, i.\text{rep}, i.\tau_3, d - \text{depth}(i)]\) \(\triangleright\) Find vertex
   return if \(c \neq -1\) and \(\text{pre\_rank}(c) < \text{pre\_rank}(i)\) then \(c\) else \(i\)

**Level Last**

The operation `LEVEL_LAST` differs from `LEVEL_FIRST` in minor details:

- We define an arrays `llast_1` (and `llast_2`):
  \[
  i.\text{llast}_1[j] = \arg\max_{\text{pre\_rank}(v) \leq \text{pre\_rank}(i) + \text{subtree\_size}(i) - 1} \text{pre\_rank}(v) \\
  \text{depth}(v) = \text{depth}(i) + j
  \]

- There is one more cause of interruption in the `LEVEL_LAST` version of the lemma [22]. Type (1) connected mini-trees pose the same reason for interruption as the type (2) connections. This is because the canonical names are reported.

- We select the candidate with highest rank at the last line of the algorithm.

**Level Next and Level Previous**

We focus on `LEVEL_NEXT`; the operation `LEVEL_PREV` is symmetrical.

We distinguish four cases:

1. the vertex \(i\) is the level-last vertex in the subtree of \(a\). Then there is no next vertex on the level.
2. the vertex \(i\) is not the level-last vertex in its micro-tree. We use a look-up table to handle this case. We also need to check for type (3) connected subtree which could contain the answer. This check can be easily done by comparing pre-order rank of \(i\) and \(\text{dummy}_2(i)\).
3. the vertex \(i\) is the level-last vertex in its micro-tree, then we search the micro-tree \(t_2\) which is “to the right” from \(i\). We find its \(\tau_2\) name using a structure which we describe later. We distinguish two cases:
   - If the micro-tree \(t_2\) contains a dummy vertex and \(i\) is in its subtree,
     then we solve the search in \(t_2\) using a look-up table `LEVEL_NEXT` searching for the first vertex \(v\) on the desired level with \(\text{pre\_rank}(v) > \text{pre\_rank}(\text{bottom}_2(t_2))\).
• Otherwise, it is the level-first vertex in the micro-tree $t$.

(4) The vertex $i$ is the level-last vertex in its mini-tree. We need to find vertex in the mini-tree $t_1$ which is “to the right” from the current one. As before, we distinguish two cases:

• If $i$ is a descendant of the dummy vertex in $t_1$, then we need to find the micro-tree which contains the vertex $v$ such that $\text{pre_rank}(v) > \text{pre_rank}(\text{bottom}_1(t_1))$.

As there is at most one dummy vertex in a mini-tree, we store a compressed array \textit{dummy_next} which contains $\tau_2$ names of micro-trees which contain the answer for all admissible levels. This compressed array contains at most $B$ elements in $O(B)$ runs and therefore uses $O\left(\frac{B}{\tau} \log B\right) = o(B)$ bits of space. The number of runs follows from the constant bound in lemma 22 and the fact that there are $O\left(\frac{B}{\tau}\right)$ micro-trees in a mini-tree.

Let $t_2$ be the micro-tree which is obtained from the compressed array \textit{dummy_next}. We finish the query as in the previous case since \textit{dummy}_2(t_2) could be an ancestor of $i$.

• Otherwise, the answer is the level-first vertex in the mini-tree $t_1$.

We define a graph $G$ whose nodes are mini-trees and each two nodes $x, y$ are connected whenever there is a vertex $u$ in $x$ and a vertex $v$ in $y$ such that $\text{level_next}(i, u) = v$.

\textbf{Lemma 23.} The graph $G$ has $O\left(\frac{n}{B}\right)$ edges.

\textit{Proof.} The graph is planar, therefore there is a linear bound on number of edges. \hfill $\Box$

For each mini-tree $i$, we define an array \textit{lnext}:

$$\text{lnext}_1[j] = \text{level_next}(\text{root}_1(0), \text{level_last}(i, \text{depth}(i) + j)).\tau_1$$

If there is no next vertex, than we use the name $-1$.

We build a collection of compressed arrays out of all \textit{lnext}_1 arrays, which will be stored in the global structure. They contain $O(n)$ elements in total as each vertex except for mini-tree roots is the rightmost one in at most one mini-tree, and the number of mini-tree roots is $O\left(\frac{n}{B}\right)$. There are at most $O\left(\frac{n}{B}\right)$ runs in all arrays, which follows from the lemmas 23 and 22 there are only $O(1)$ ways how three and three runs can be combined together. The extra “no next vertex” names do not matter as we they form a single run until the end of the array. The collection of compressed arrays requires $O\left(\frac{n}{B} \log n\right) = o(n)$ bits of space.

We construct a similar structure \textit{lnext}_2 for all micro-trees, storing the \tau_2 name of the micro-trees containing the level-next vertex. This requires $O\left(\frac{B}{\tau} \log B\right) = o(B)$ bits of memory, and it is stored in each mini-tree structure.
function LEVEL_NEXT(a, i):
    if i = level_last(a, depth(i)):
        return −1
    j ← level_next[i.size, i.rep, i.τ2]  # Search in micro-tree
    if i.dummy2 ≠ −1 and (j = −1 or pre_rank(bottom2(i)) < pre_rank(j)):
        f ← level_first(bottom2(i), depth(i))  # Check dummy vertex
        if f ≠ −1:
            return f
    else if j ≠ −1:
        return i.τ1, i.τ2, j  # Answer in micro-tree
    t2 ← i.lnext2[i.τ2, depth(i) − depth(root2(i))]  # Search for micro-tree
    if t2 ≠ −1 and t2.dummy2 ≠ −1 and is_ancestor(bottom2(t2), i):
        return level_next'[t2.size, t2.rep, depth(i) − depth(root2(t2))]
    else if t2 ≠ −1:
        return level_first(t2, depth(i))
    t1 ← lnext1[i.τ1, depth(i) − depth(root2(i))]  # Search for mini-tree
    if t1 ≠ −1 and t1.dummy1 ≠ −1 and is_ancestor(bottom1(t1), i):
        t2 ← t1.dummy_next[depth(i) − depth(dummy1(t1))]
        if t2.dummy2 ≠ −1 and is_ancestor(bottom2(t2), i):
            return level_next'[t2.size, t2.rep, depth(i) − depth(root2(t2))]
        else:
            return level_first(t2, depth(i))
    else if t1 ≠ −1:
        return level_first(t1, depth(i))
    else:
        return −1

4.2.8 Final Thoughts

In the beginning we defined the size B dependent on the constant c ≥ 2. We can set c = 2 as long as we do not require the ancestral and level operations in the form we have shown (they require c ≥ 3). However, it is not true that c > 2 is required as [FM14] shown the same structure with slightly more complicated operations for which c = 2 is sufficient. We chose the simpler options as they allowed us to reuse structures which we defined in other parts of this work.

The TC data structure supports all operations of FF; it is better in ranking operations (POST_RANK, POST_SELECT, DFUDS_RANK, DFUDS_SELECT) which are not supported by the former one. The advantage of TC is that it is easier to extend it by adding small bits to an existing skeleton. On the other hand, some operations are significantly harder to implement.
4.3 Universal Succinct Representation

All of the previous structures can be split into two parts (recalling the definition of systematic data structures):

**data**

In case of BP (and FF – the same representation with different indices) and DFUDS, it is clearly the bit string $S$ of size $2n$ bits.

In case of TC, it is the representations of micro-trees in form of a sequence of pairs $(\text{size}, \text{rep})$. Although it is not enough to restore the original tree – for that we would need to store at least the number of mini-trees in the tree, number of micro-trees in a mini-tree, parents, primaries and dummy vertices, we do not consider it here.

**index**

Any structure or structures of size $o(n)$ bits which speed up the queries; for BP and DFUDS they were indices for RANK, SELECT, MATCH, RMQI. In case of TC, the global, mini-tree and micro-tree structures without what we defined as data take only $o(n)$ bits and will be considered as an index.

Any algorithm for word-RAM can access only $w = \Theta(\log n)$ consecutive bits of memory in one step. Since all the operations which we have shown were formulated for word-RAM, at each step at most $w$ bits of the bit string $S$ or at most one micro-tree representation $(\text{size}, \text{rep})$ can be accessed in the memory. We can replace the data part of each data structure by simulation of the memory access in time $O(1)$. As long as the structure providing the simulation uses only $2n + o(n)$ bits, the original data structure is still succinct.

We show a data structure for a universal succinct representation of trees which was described by [FRR09].

The data structure provides the following three operations in running time $O(1)$:

$\text{bp}\_\text{substring}(i, s) \rightarrow S$

It returns $s \leq w$ bits of BP representation staring at position $i$.

$\text{dfuds}\_\text{substring}(i, s) \rightarrow S$

It returns $s \leq w$ bits of DFUDS representation staring at position $i$.

$\text{tc}\_\text{microtree}(\tau_1, \tau_2) \rightarrow (\text{size}, \text{rep})$

Returns the micro-tree representation by its $(\tau_1, \tau_2)$ name. We assume that the decomposition and the naming schema which we described in the section 4.2 is used.

Although the parameter $s$ can be in range $[1, w]$, we will restrict it to a fraction of $\log n$; the exact value depends on the target representation and on a constant which we leave as a parameter of the data structure. We also constrain $i$ to be a multiple of $s$. The answer to a query with unrestricted $i$ and $s$ can still be obtained by $O(1)$ queries of the restricted type.

Assuming that such data structure exists, we can equip it with all indices (for BP, DFUDS and TC) in order to obtain a succinct data structure which supports the union of all operations of individual representations. Although it may seem that TC is superior to any other representation, there are still advantages:
• If there are more ways to support the same operation, the fastest one or the one requiring the smallest index (both in terms of a real implementation) can be chosen.
• If any new operation becomes supported by either of the representations, the universal data structure can benefit from it.

4.3.1 Restriction to Mini-Trees

We start with the same two-level decomposition as in the Tree Covering representation. The mini-trees are the primary part of the data structure, which is where the operations BP_SUBSTRING and DFUDS_SUBSTRING are supported. The micro-trees will be used much later since they are required only to support the operation TC_MICROTREE.

Almost every parenthesis in the bit strings $S$ of BP and DFUDS representation can be associated with a vertex:

• an opening parenthesis in BP directly represents a vertex;
• a closing parenthesis in BP represents the same vertex as the matching opening one;
• the first parenthesis in DFUDS is unassociated;
• the first parenthesis following a closing one or parenthesis at position 1 directly represents a vertex;
• any other parenthesis in DFUDS represents the same vertex as the preceding one.

Contrary to the original proposal in [FRR09], we use a different association in the DFUDS representation. The advantage of our approach is that the algorithms become simpler and we are able to generate bigger blocks of bits.

We define two compressed arrays $bp$ and $dfuds$ which contain the $\tau_1$ name in the TC representation of the vertex associated with the parenthesis at every position.

Lemma 24. Every $\tau_1$ name occurs in at most 4 runs in the compressed array $bp$ and 3 runs in $dfuds$.

Proof. The definitions of the BP and DFUDS representations are based on preorder traversal of the tree, and they can be expressed recursively. All vertices between the first and the last occurrence of the $\tau_1$ name of a mini-tree $t$ are in the subtree of the mini-tree root $v$ of $t$. We call own the children of $v$ which are contained in the mini-tree $t$.

There are three types of connections which connect other mini-trees to $t$. All type (1) and type (2) connections involve the root $v$; some subtrees rooted in children of $v$ can belong to other mini-trees. The own children of $v$ form an interval which is not interrupted by any type (1) nor type (2) connection.

In the array $bp$, the subtree of $v$ consists of:

(1) the opening parenthesis of the root $v$,
(2) subtrees of children preceding own children (the first interruption),

100
(3) subtrees of own children,
(4) subtrees of children following own children (the second interruption),
(5) the closing parenthesis of the root \( v \).

In the array \( dfuds \), the subtree of \( v \) is slightly different:

(1) the degree sequence of the root \( v \),
(2) subtrees of children preceding own children (the only interruption),
(3) subtrees of own children,
(4) subtrees of children following own children.

Any of the parts in both arrays can be empty, which can only decrease the number of interruptions. Each mini-tree can further have up to one type (3) connection which can interrupt once the run containing own children.

The number of runs follows from the number of interruptions. □

According to the lemma, there are at most \( O(1) \) runs of each name, therefore the size of the compressed arrays \( bp \) and \( dfuds \) is \( o(n) \).

We immediately solve all queries which span over multiple \( \tau_1 \) names in the arrays \( bp \) and \( dfuds \), and also all queries which involve any bit of the representation of any mini-tree root in the DFUDS case.

For each run, we store the last and the following \( w \) bits of both representations. In case of DFUDS and a primary mini-tree, we also store the first \( w \) bits starting at the position of the closing parenthesis in the degree sequence of the root. This requires \( O \left( \frac{n}{B} w \right) = o(n) \) bits.

We can detect such queries in \( O(1) \) time by the following two conditions:

\[
\text{run\_last}(i) - i < b, \\
\text{dfuds}[i].\text{primary}_i = \text{dfuds}[i] \quad \text{and} \quad \text{rank}(\text{dfuds}, i) \leq \text{dfuds}[i].\deg + 1.
\]

In the first case we simply answer the query using the parts which we have stored; the number of the run can be found using the function \text{ELEMENT\_INDEX}. In case of the root in DFUDS representation, we first generate the right number of opening parentheses and then append the stored part. All this can be implemented using arithmetic and bitwise operations on word-RAM.

From now on, we can assume that all queries span a single run. They can therefore be restricted to a query on the BP or DFUDS representation of a single mini-tree.

We do not have to handle specially the case of existence of a mini-tree dummy vertex since its encounter causes an interruption in the arrays \( bp \) and \( dfuds \), and therefore such query must have already been solved. It is still necessary to keep it in order to calculate the degree of its parent correctly in the DFUDS representation.

We store the offset of the dummy vertex in the BP and DFUDS representation of the mini-tree. The dummy vertex is represented by the sequence "() in BP and by ")" in DFUDS representation. The position \( i \) is transformed to a position \( i' \) inside the BP and DFUDS representations of the mini-tree.

101
function BP.REDUCE(i):
    \( i' \leftarrow \text{rank}(bp, i) - 1 \)
    if \( bp[i].τ_1 \neq bp[i].\text{primary}_1 \):
        \( i' \leftarrow i' + 1 \) ▷ Not primary
    if \( bp._\text{dummy} \neq -1 \) and \( i' \geq bp._\text{dummy} \):
        \( i' \leftarrow i' + 2 \) ▷ Dummy vertex is before

function DFUDS.REDUCE(i):
    \( i' \leftarrow \text{rank}(dfuds, i) - 1 \)
    if \( dfuds[i].τ_1 = dfuds[i].\text{primary}_1 \):
        \( i' \leftarrow i' - dfuds[i].\text{degree} - 1 \) ▷ Primary
        \( i' \leftarrow i' + dfuds[i].\text{degree}_\text{own} + 1 \) ▷ Remove root’s degree sequence
    if \( dfuds._\text{dummy} \neq -1 \) and \( i' \geq dfuds._\text{dummy} \):
        \( i' \leftarrow i' + 1 \) ▷ Dummy vertex is before

So far we needed the following fields to be stored:

Global

\( bp \) The compressed array which maps positions of parentheses of the BP representation to mini-trees.

\( dfuds \) The compressed array which maps positions of parentheses of the DFUDS representation to mini-trees.

\( bp._\text{last}, bp._\text{following}, dfuds._\text{last}, dfuds._\text{following} \) Arrays containing words of the last and the following \( w \) bits for each run and for both representations.

Mini-tree

\( τ_1, \text{primary}_1 \)
They are the same as in the TC data structure.

\( \text{degree}, \text{degree}_\text{own} \)
The full degree of the mini-tree root, and its root when restricted to children in this mini-tree.

\( bp._\text{dummy}, dfuds._\text{dummy} \) Positions of the dummy vertex in the BP and DFUDS representations of the mini-tree.

\( dfuds._\text{start} \) Contains the \( w \) bits of the DFUDS representation starting at the position of the closing parenthesis in the degree sequence of the root.

From now on, we restrict our description to a single mini-tree; we ignore all other mini-trees, including those connected via the dummy vertex.

We call a vertex significant if its subtree size is larger than \( d' = \frac{\log n}{d} \), for a constant \( d \). A skeleton of a mini-tree is the subtree induced by significant vertices. The skeleton must be connected since each ancestor of a significant vertex is also significant.

There are at most \( O\left(\frac{B}{\log n}\right) \) leaves in the skeleton as the subtree of each of them contains at least \( d' \) vertices. We can assume that there is also at least one significant vertex as otherwise the size of the representation of the mini-tree is less than \( w \), and so every query has already been solved.
4.3.2 Skinny Mini-Trees

We call a mini-tree skinny if its skeleton is a path. We solve this special case first, and then generalize it to all skeletons.

We use the same notation as the authors in [FRR09]: Let $P$ by the skeleton, which is a path, $u$ its leaf and $v$ the last child of $u$. Let us define the following sets:

\[
S = \{s : \text{parent}(s) \in P \text{ and } s \notin P\}
\]
\[
S_D = \{s \in S : \text{pre}_\text{rank}(s) \leq \text{pre}_\text{rank}(v)\}
\]
\[
S_U = \{s \in S : \text{pre}_\text{rank}(s) > \text{pre}_\text{rank}(v)\}
\]

We represent a skinny mini-tree by four bit strings:

**Path down, $P_D$**

The concatenation of the unary degree sequences of vertices in the skeleton, assuming only their children in $S_D$. They are stored in the order from the root to the leaf $u$.

**Path up, $P_U$**

The same for children in $S_U$ and direction from $u$ to the root.

**Trees on path down, $T_D$**

Subtrees of $S_D$ stored consecutively according to their pre-order numbers. We call the trees left dangling trees. The trees can be stored in any self-delimiting representation which requires at most $2k - 1$ bits for a tree with $k$ vertices. We can either use BP without the first parenthesis, which is an opening one, or DFUDS without the artificially prepended opening parenthesis.

**Trees on path up, $T_U$**

The same for subtrees of vertices $S_U$ ordered by their pre-order numbers. We call them right dangling trees.

These four bit strings together use exactly 2 bits per every vertex of the mini-tree. Each opening parenthesis in $P_D$ and $P_U$ can be paired with the one missing in representations of the dangling trees in $T_D$ and $T_U$. Vertices belonging to the skeleton are represented by closing parentheses in $P_D$ and $P_U$, one in each.

We can restore the original mini-tree from this representation:

function restore():

\[ v \leftarrow \text{null} \]

\[ \text{while } P_D \text{ has more symbols:} \]

\[ v \leftarrow \text{new child of } v \]

\[ \text{while Read a symbol in } P_D; \text{ is it an opening parenthesis:} \]

\[ \text{Read a tree from } T_D, \text{ decode it and add it as a child to } v. \]

\[ \text{while } P_U \text{ has more symbols:} \]

\[ \text{while Read a symbol in } P_U; \text{ is it an opening parenthesis:} \]

\[ \text{Read a tree from } T_U, \text{ decode it and add it as a child to } v. \]

\[ v \leftarrow \text{parent}(v) \]
Generation of the BP and DFUDS Representations

We recall the meaning of the constant $d$ from the definition of a significant vertex. A dangling tree contains less than $d' = \log_2 n$ vertices.

In the following two lemmas we show how it is possible to generate $q$ bits of BP and DFUDS representation, where $q$ is a constant dependent on the representation and constrained by $d$.

**Lemma 25.** Using an index of size $o(B)$, we can query $q' = \log_2 n / q$ bits of BP representation of a skinny mini-tree provided that $i$ is a multiple of $q'$. The constant $q$ is restricted to:

$$\frac{2}{q} + \frac{4}{d} < 1.$$ 

**Proof.** We first formulate an algorithm which outputs the BP representation of the whole mini-tree. The algorithm is based on RESTORE; we split it into two functions BP_SKINNY_DOWN and BP_SKINNY_UP:

**function** BP_SKINNY_DOWN():

\[ p_D \leftarrow 0, \quad t_D \leftarrow 0 \]

while $p_D < |P_D|:

\[ \text{output}("\)"
\]

\[ \text{while } P_D[p_D] = "\)"
\]

\[ t \leftarrow \text{read_tree}(t_D), \quad t_D \leftarrow t_D + |t|, \quad p_D \leftarrow p_D + 1 \]

\[ \text{output(bp_rep[t])} \]

\[ p_D \leftarrow p_D + 1 \quad \triangleright \text{Consume the closing parenthesis} \]

**function** BP_SKINNY_UP():

\[ p_U \leftarrow 0, \quad t_U \leftarrow 0 \]

while $p_U < |P_U|:

\[ \text{while } P_U[p_U] = "\)"
\]

\[ t \leftarrow \text{read_tree}(t_U), \quad t_U \leftarrow t_U + |t|, \quad p_U \leftarrow p_U + 1 \]

\[ \text{output(bp_rep[t])} \]

\[ p_U \leftarrow p_U + 1 \quad \triangleright \text{Consume the closing parenthesis} \]

\[ \text{output("")} \]

For each $j$ we store the state of the algorithm at the point when it is about to access the arrays in order to produce the $i$-th bit of the representation:

- which function $f$ is being processed, 1 bit;
- the instruction pointer $ip$, $O(1)$ bits;
- the values of all local variables, $O(\log \log n)$ bits.

Using this index we can restart the course of the algorithm from any position which, related to the output, is either a parenthesis of a vertex in $P$ or the beginning of a dangling tree. In order to support any position of the output, we add to the index an offset $o$ which allows us to skip the first $o$ bits of which we do not need.

In order to generate any $s$ bits of the BP representation, either function needs to read at most $s$ bits from $P_*$ as well as up to $s + 4d'$ bits from $T_*$. The term $4d'$ follows from accessing two dangling trees more than is needed.
We can therefore implement the algorithms solely as two precomputed look-up tables $\text{BP_SKINNY_DOWN}$ and $\text{BP_SKINNY_UP}$ which get all information from the index and return the needed bits. The tables return correctly aligned bits ready for the output together with number of bits which they could not generate. The table has an additional argument $s$ which specifies the number of right-aligned bits to return. The total size of the index of the look-up table is $2q' + 4d' + O(\log \log n)$, which is from where the restriction on value of $q$ comes.

We state the algorithm in a more general form which allows to generate $s \leq q$ bits; it also returns the number of bits which it could not produce. Neither of these features is useful for skinny mini-trees as we always need $q$ bits, and the query cannot reach the end, because such query has already been solved.

function $\text{BP_SUBSTRING_SKINNY}(i, s)$:

\[
\begin{align*}
  r &\leftarrow 0 & \triangleright \text{Representation buffer} \\
  (f, ip, p, t, o) &\leftarrow \text{bp_index}\left[\frac{i}{q}\right] \\
  \text{if } f = 0: & \triangleright \text{Function for going down} \\
  S &\leftarrow (P_D[p : p + q'], T_D[t : t + q' + 4d'], |P_D|, |T_D|) \\
  (r, s) &\leftarrow \text{bp_skinny_down}[S, ip, p, t, o, s] \\
  ip &\leftarrow 0, \quad p &\leftarrow 0, \quad t &\leftarrow 0, \quad o &\leftarrow 0 \\
  \text{if } s > 0: & \triangleright \text{Function for going up} \\
  S &\leftarrow (P_U[p : p + q'], T_U[t : t + q' + 4d'], |P_U|, |T_U|) \\
  (r', s) &\leftarrow \text{bp_skinny_up}[S, ip, p, t, o, s] \\
  r &\leftarrow r \mid r' & \triangleright \text{Bitwise or} \\
  \text{return } r, s
\end{align*}
\]

Lemma 26. Using an index of size $o(B)$, we can query $q' = \frac{\log n}{q}$ bits of BP representation of a skinny mini-tree provided that $i$ is a multiple of $q'$. The constant $q$ is restricted to:

\[
\frac{3}{q} + \frac{4}{d} < 1.
\]

Proof. We use the same technique as in the previous lemma. The algorithm is a little more complicated as it needs to go through $P_D$ twice: the first time to report the degree, and the second time when the left dangling trees are reported. It also go through $P_U$ twice: the first time from the end to the beginning treating closing parentheses as delimiters, and the second time to output the right dangling trees.
function DFUDS_SKINNY_DOWN():
  \( p_D \leftarrow 0, \quad p_U \leftarrow |P_U| - 2, \quad t_D \leftarrow 0 \)
  while \( p_D < |P_D| \):
    \( p_D \leftarrow p_D \) \hfill \( \triangleright \) First we output the degree sequence
    while \( P_D[p_D] = "(" \):
      \( p_D \leftarrow p_D + 1 \) \hfill \( \triangleright \) Left dangling children
      output(""")
    \( p_D \leftarrow p_D + 1 \) \hfill \( \triangleright \) Consume the closing parenthesis
    if \( p_D < |P_D| \):
      \( p_D \leftarrow p_D \) \hfill \( \triangleright \) Child in the path
      output(""")
    while \( P_U[p_U] = "(" \) and \( p_U \geq 0 \):
      \( p_U \leftarrow p_U - 1 \) \hfill \( \triangleright \) Right dangling children
      output(""")
    \( p_U \leftarrow p_U - 1 \) \hfill \( \triangleright \) Consume the closing parenthesis
    output(""")
    \( p_U \leftarrow p_U - 1 \) \hfill \( \triangleright \) End of unary degree sequence
  while \( P_D[p_D] = "(" \):
    \( p_D \leftarrow p_D + 1 \) \hfill \( \triangleright \) Then we output the left dangling children
    \( t \leftarrow \text{read_tree}(t_D), \quad t_D \leftarrow t_D + |t|, \quad p_D \leftarrow p_D + 1 \)
    output(dfuds_rep[t])
  function DFUDS_SKINNY_DOWN():
    \( p_U \leftarrow 0, \quad t_U \leftarrow 0 \)
    while \( p_U < |P_U| \):
      \( p_U \leftarrow p_U \) \hfill \( \triangleright \) Output the right dangling trees
      if \( P_U[p_U] = "(\):
        \( t \leftarrow \text{read_tree}(t_U), \quad t_U \leftarrow t_U + |t| \)
        output(dfuds_rep[t])
      \( p_U \leftarrow p_U + 1 \) \hfill \( \triangleright \) Consume the closing parenthesis

The index contains record about three local variables. The look-up table \( \text{DFUDS\_SKINNY\_DOWN} \) requires access to \( q' \) bits of \( P_U \) and \( P_D \), and to \( q' + 4d' \) bits of \( T_D \), while \( \text{DFUDS\_SKINNY\_DOWN} \) needs \( q' \) bits of \( P_U \), and \( q' + 4d' \) bits of \( T_U \). The maximum total size of the index is \( 3q' + 4d' + O(\log \log n) \), which is from where the restriction on value of \( q \) comes.

The algorithm \( \text{DFUDS\_SUBSTRING\_SKINNY} \) for the DFUDS representation is similar to \( \text{BP\_SUBSTRING\_SKINNY} \). The difference is only in the data which are provided to the look-up tables.

The better calculation of the size of the index proved to be beneficial. If we set \( d = 16 \), which was the value originally proposed in [FRR09], we set \( q = 3 \) for BP and to \( q = 5 \) for DFUDS representation. Both values are significantly better than the former \( q = 8 \) for BP and \( q = 24 \) for DFUDS representation.
4.3.3 General Mini-Trees

If the skeleton is a general tree, we decompose it iteratively into paths. We first remove the leftmost path connecting the root with the first skeleton leaf; we call it a left-leaning path. For each tree in the rest of the skeleton, we remove its rightmost path connecting the root with the last skeleton leaf; we call it a right-leaning path. We repeat this until every vertex of the skeleton is in a left-leaning or a right-leaning path.

The decomposition keeps subtrees of non-significant vertices connected to their parent, which is now in a path; they are again called left and right dangling trees. A path together with its dangling trees can be viewed as a skinny mini-tree. There are \( O\left(\frac{B}{\log n}\right) \) paths as each of them ends in a leaf. We assign each path a number starting with 0.

The paths are connected with each other in three possible ways:

- A left-leaning path can have left-leaning and right-leaning paths connected to the right. If there are more of them connected to one vertex, we store only the leftmost one.
- A right-leaning path can have left-leaning paths connected to the left. Again, if there are more of them connected to the same vertex, we store only the leftmost one.
- All paths except for one have a next path. The next path is either the path containing the right sibling of its root if it exists, or the path containing the parent of its root. The only path which does not have a next path is the path containing the mini-tree root.

In the skinny mini-trees we could uniquely reference any bit of the representation by the state of the algorithm and the offset \( o \). In the case of general mini-trees, we extend the index by the number of the path. The size of the index record is \( O(\log \log n) \) and we call it a reference.

Path Structures

First, we add opening parentheses to the bit strings \( P_D \) and \( P_U \) for all connected paths. Without this change reporting the vertex degree in the DFUDS representation would be wrong. This change adds \( O\left(\frac{B}{\log n}\right) \) bits because the each path is connected to at most one other path. The root of a path (except for the path containing the mini-tree root) is represented by 3 bits in total.

For each path we define arrays \( C_D \) and \( C_U \) which mark the positions of the parentheses which we have just added. Note that only one of the arrays is non-empty since left-leaning paths have connections only to the right – in \( C_U \), and right-leaning paths only to the left – in \( C_D \). In order to find the position of the closest connection, we turn the arrays \( C_D \) and \( C_U \) into a collection of compressed arrays using the operation \( \text{succ} \). The values in the arrays are the references denoting where to continue; we store two versions of them: for BP and for DFUDS representation.

There are \( O(B) \) elements in the each collection of compressed arrays in total since they contain the same number of elements as the arrays \( P_* \). Each reference
in the compressed arrays refers to the root of a connected path, therefore each of them occurs in a single run, and the total number of them is bounded by the number of paths. Each collection of compressed arrays takes $o(B)$ bits of space.

**Path**

The path structure contains the following fields:

- $k$: The number of the path.
- $P_D, P_U, T_D, T_U$
  - As before.
- $bp_{next}, dfuds_{next}$
  - References to the next path for both BP and DFUDS representations.

**Mini-tree**

The fields which are stored for the whole mini-tree.

- $bp_{index}, dfuds_{index}$
  - These are the indices from the algorithms BP_SUBSTRING_SKINNY and DFUDS_SUBSTRING_SKINNY generalized to incorporate the path number. They contain a reference for each multiple of $q'$. 
- $bp_{CD}, bp_{CU}, dfuds_{CD}, dfuds_{CU}$
  - The collections of compressed arrays which contain references to connected paths for all paths.

**Paths**

Concatenated structures for paths together with a table of their offsets.

**Algorithm for the General Mini-Trees**

We extend the algorithm for skinny mini-trees by several modifications:

- It is provided the reference (the state of the algorithm) instead of finding it in the index itself.
- It finds the position $c$ of the closest connected path in the direction of traversal and provides it to the look-up table.
- Whenever the algorithm captured by the look-up table encounters the position $c$, it shall stop and return the reason.
- The look-up tables return more information:
  - the number of bits which they could not generate;
  - the buffer containing the generated bits;
  - the situation which lead to the end:
    - (1) the requested number of bits was generated;
    - (2) a turn from direction down to up is requested;
    - (3) a switch to a connected path is necessary;
    - (4) a switch to the next path is requested.
  - If the reason is a switch (3) or (4), a recursive call with the number of bits left to be generated is made.
We present an algorithm returning a block of $q'$ bits of the BP representations starting at a position which is a multiple of $q'$. The same restrictions to $q$ applies as in the lemma [25]. A similar algorithm is for DFUDS representation (with its constraint on the value of $q$).

**Function** `bp_substring(I, s)`:
- $I$ is the record in the global index $(k, f, ip, p, t, o) \leftarrow I$
- $f = 0$: Function for going down
  - $S \leftarrow (k.P_D[p : p + q'], k.T_D[t : t + q' + 4d'], \|k.P_D\|, \|k.T_D\|)$
  - $c \leftarrow \text{run_last}(\text{bp}_C_D, k, p)$, $C \leftarrow \text{bp}_C_D[k, c]$
  - $(r, s, e) \leftarrow \text{bp}_\text{skinny_down}(S, I, s, c)$
- $f = 1$: Function for going up
  - $S \leftarrow (k.P_U[p : p + q'], k.T_U[t : t + q' + 4d'], \|k.P_U\|, \|k.T_U\|)$
  - $c \leftarrow \text{run_last}(\text{bp}_C_U, k, p)$, $C \leftarrow \text{bp}_C_U[k, c]$
  - $(r, s, e) \leftarrow \text{bp}_\text{skinny_up}(S, I, s, c)$
- $e = 2$: Switch direction
  - $(r', s) \leftarrow \text{bp}_\text{substring}((k, 1, 0, 0, 0, 0), s)$
  - $r \leftarrow r | r'$
- $e = 3$: Switch to connection
  - $(r', s) \leftarrow \text{bp}_\text{substring}(C, s)$
  - $r \leftarrow r | r'$
- $e = 4$ and $k.bp_next \neq -1$: Switch to next
  - $(r', s) \leftarrow \text{bp}_\text{substring}(k.bp_next, s)$
  - $r \leftarrow r | r'$
- **Return** $r, s$

It only remains to argue that the depth of the recursion is constant. We inspect the graph of transitions between paths.

![Figure 4.2: Graph of the transitions between paths and directions of the traversal](image)

First we observe that all cycles in the graph contain the edges labeled “Turn”. We focus on the transitions of the direction of a path traversal from “down” to “up”. 

109
Lemma 27. When the direction of the path traversal changes from “down” to “up”, at least 2$d'$ bits of the representation are generated.

Proof. As the initial reference in the global index can lead anywhere, we start the analysis after the first transition. All references leading to a left-leaning path traversed down target its root. In case of the right-leaning path traversed down, there are three possible references to it:

- If the path was referenced as a right connection, then its root was the target.
- If the path was referenced as the next from a sibling of its root, then its root was targeted.
- If the path was referenced as the next from other vertex, then it was from the root of its left connection. Because of the path decomposition algorithm, this left-leaning path cannot be connected to the skeleton leaf.

In all these cases the whole subtree of the skeleton leaf is processed. Since the subtree contains $d'$ vertices, at least 2$d'$ bits of representation are generated.

From the lemma and from the lengths of the cycles follows that at most $3 \frac{d}{q} + 2 = O(\frac{d}{q}) = O(1)$ recursive calls are required. The additional 2 is the number of recursive calls before the first cycle.

Since we want to generate as many bits as possible, we want to set $q$ small, however that results in deeper recursion and longer running time. Setting $q \geq \frac{d}{2}$ ensures that a single “Turn” generates enough bits.

4.3.4 Micro-Tree Representation

The mini-tree is decomposed into micro-trees as it is usual for the Tree Covering representation. We reuse the lemma 24 which in this case claims that the BP representation of a micro-tree forms at most four runs in the BP representation of a mini-tree. We prefer the BP representation over DFUDS because no degree adjustment of the root is necessary.

For each micro-tree $(\tau_1, \tau_2)$ we store positions of the beginning and the end of each run of $\tau_2$ withing the BP representation of the mini-tree $\tau_1$. If the micro-tree contains a dummy vertex, an adjustment of the run boundaries by 1 is needed. The length of each run is bounded by the size of the micro-tree, which is less then $\frac{\log n}{2}$, therefore $O(1)$ queries of the BP representation are sufficient. If the micro-tree is not a primary one, an opening and a closing parentheses are added.

The size of BP representation of a micro-tree can be calculated from the stored offsets of the runs and its primarity. Because of the small size of the micro-tree BP representation, we can use a look-up table to transcode it into any other representation which is native for the TC data structure.

4.3.5 Final Thoughts

The universal succinct representation provides a way how to combine several data structures which we have shown in the previous sections. As two of the three representations (BP/FF, DFUDS, TC) support nearly the same set of operations, its benefits consist in allowing to choose the most space or time efficient implementation of an operation out of several alternatives.
Conclusion

The aim of the thesis was to explore the existing succinct data structures for representing trees, to compare them, and possibly to improve them. We have succeeded in all three tasks.

To summarize this thesis, we have presented five ways how a static ordinal tree with \( n \) vertices can be encoded using only \( 2n + o(n) \) bits of space. The first three use a direct encoding in a bit string of \( 2n + 1 \) or \( 2n \) bits. The later two representations build on top of them by either replacing the index, or moving the idea of decomposition from bit strings to subtrees. Finally, we showed a way how the data part of three different representations can be shared among them.

The representations differ in the set of operations which the support. We proceeded from the very simples ones which do not offer much, to representations which support almost every operation. The comparison of the supported operation can be found in the table 5.1. The universal succinct representation is not shown as it only provides the union of the last four columns. Our improvements, which are in more detail described in the next section, are typeset in bold.

All structures and algorithms which we present in this thesis are also described with two exceptions:

**Fusion tree**
A fusion tree is required for the tiny compressed array. This decision can be advocated by the fact that it is only of a theoretical interest.

**Succinct indexable dictionary**
The succinct indexable dictionary appears twice in our work: first in the definition of the tiny compressed array, and then also in the solution of the generalized \textsc{rmq\_size} and \textsc{rmq\_select}. We did not incorporate the description of this structures as it uses technique which are too different from those which we used for representing trees.

We simplified some operations by not using the state of the art data structures in order to provide a description of a reasonable complexity.
<table>
<thead>
<tr>
<th>Operation</th>
<th>LOUDS</th>
<th>BP</th>
<th>DFUDS</th>
<th>FF</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>*_RANK, *_SELECT</td>
<td>LO_*</td>
<td>PRE_<em>, POST_</em>, IN_*</td>
<td>PRE_<em>, DFUDS_</em></td>
<td>PRE_<em>, POST_</em>, DFUDS_<em>, IN_</em> (new result)</td>
<td></td>
</tr>
<tr>
<td>PARENT, IS_ROOT, IS_LEAF</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CHILD_*, DEGREE</td>
<td>Yes</td>
<td>Yes, CHILD_RANK, CHILD_SELECT, DEGREE not-shown</td>
<td>Yes</td>
<td>Yes (alternative)</td>
<td>Yes</td>
</tr>
<tr>
<td>DEPTH</td>
<td>Yes, new result</td>
<td>Yes</td>
<td>Yes, not-shown</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LEVEL_* (restrictable)</td>
<td>Yes</td>
<td>Only LEVEL_PREV, LEVEL_NEXT, not-shown</td>
<td>No</td>
<td>Only iteration (subtree)</td>
<td>Only iteration (alternative) (subtree)</td>
</tr>
<tr>
<td>LEAF_* (restrictable)</td>
<td>Yes, in level-order (level range)</td>
<td>Yes (subtree)</td>
<td>Yes (subtree)</td>
<td>Yes (subtree)</td>
<td>Yes (subtree)</td>
</tr>
<tr>
<td>IS_ANCESTOR</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LEVEL_ANCESTOR</td>
<td>No</td>
<td>Yes, not-shown</td>
<td>Yes, not-shown</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LCA, DISTANCE</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HEIGHT, Deepest_VERTEX</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SUBTREE_SIZE</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of operations supported by various representations
Our Contribution

Our first result is the definition of the compressed array in section 2.3.2. Although this technique has been widely used, we wrapped it into a structure which provides clearly defined operations. The use of the compressed array simplified the notation in many of the presented algorithms.

Our major result is the proposal of an index which supports the depth and level* operations in the LOUDS representation. The LOUDS representation being discovered first, however the attention moved fast to the other representation since the set of operations supported by LOUDS seemed to be very restricted. It is now the only representation which supports the operations level_rank and level_select.

For the first time, we have defined the operation rmq_select which generalizes rmq by allowing to specify the number of the desired occurrence of minimum. Until now, only a fixed occurrence, such as the first or the last, could be found. We also defined operations rmq_size, rmq_rank which return the number of all occurrences of the minimum in range, or in a part of it. We used the generalized range operations to propose an alternative algorithm for the degree, child_rank, child_select operations in the FF representation.

We modified the decomposition algorithm for the TC representation, which leads to the components satisfying an extra condition. This simplifies the data structure since there is one more case which needs to be handled in a special way.

We solved the operation in_rank and in_select for the TC representation. Although the solution uses the same techniques as dfuds_rank and dfuds_select, which have already been published, the in-order operations are more complex. This stems from the property that one vertex can be assigned multiple in-order numbers.

One more of our results is related to the TC representation. We propose alternative algorithms for the operations level_first and level_last which are restricted to a subtree of a given vertex. The query is reduced to a problem of finding a level ancestor, which we also describe at two different occasions.

The last of our results involves the universal succinct representation where we achieved two improvements. First, we changed ownership of parenthesis in the DFUDS representation which lowered the number of runs in this case to only three. It also simplified the algorithm captured by the look-up table generating the bits of the DFUDS representation.

The other improvement is that we analyzed carefully the constant in the definitions of a significant vertex. We used this constant and one other to formulate inequalities which describe more precisely the behavior of the BP and DFUDS generating algorithms. We increased the lower bound on the number of bits generated in the skinny mini-tree case from \( \log \frac{n}{8} \) and \( \log \frac{n}{24} \) bits for the BP and DFUDS representations to \( \log \frac{n}{4} \) and \( \log \frac{n}{8} \) bits.
Bibliography


