Compressed Suffix Trees: Design, Construction, and Applications

DISSERTATION

zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften
und Informatik der
Universität Ulm

vorgelegt von

SIMON GOG

aus Ehingen

August 2011
Amtierender Dekan: Prof. Dr. Klaus Dietmayer
Gutachter: Prof. Dr. Enno Ohlebusch
Prof. Dr. Jacobo Torán
Prof. Dr. Kunihiko Sadakane
Tag der Promotion: 24. 11. 2011
Acknowledgment

First of all I am grateful to my supervisor Enno Ohlebusch for the guidance during my research on this work here in Ulm. I also thank Uwe Schöning for providing me a research assistant position. I would also like to thank Kunihiko Sadakane and Jacobo Torán for refereeing my thesis.

I had a great time here in Ulm during my Ph.D. studies with my colleagues of the Institute. I enjoyed the daily discussions at lunch and the walk back around the building to the office. I am grateful to Adrian Kügel and Stefan Arnold for many discussions about algorithms, C++ template programming and LATEX. I also thank Adrian for proofreading this thesis and Markus Maucher for the suggestion to use the R programming language for evaluating my experimental results. I am also grateful to Timo Beller, who conducted the experiments which are depicted in Table 5.1, and to Rodrigo Cánovas for providing me a patched version of the libcds library.
# Contents

1 Introduction 2

2 Basic Concepts 5
   2.1 Notations .................................................. 5
      2.1.1 Strings .................................................. 5
      2.1.2 Trees .................................................... 5
      2.1.3 Machine Model, $O(\cdot)$-Notation, and Test Environment .................................. 6
      2.1.4 Text Compressibility .................................... 7
   2.2 The Suffix Tree ............................................ 8
   2.3 The Suffix Array ........................................... 10
   2.4 The Operations Rank and Select ............................ 14
   2.5 The Burrows-Wheeler Transform ............................. 15
   2.6 The Wavelet Tree .......................................... 17
      2.6.1 The Basic Wavelet Tree ................................. 17
      2.6.2 Run-Length Wavelet Tree .............................. 18
   2.7 The Backward Search Concept ................................ 21
   2.8 The Longest Common Prefix Array .......................... 24

3 Design 28
   3.1 The Big Picture ........................................... 28
   3.2 CST Overview ............................................... 31
   3.3 The Succinct Data Structure Library ....................... 35
   3.4 The vector Concept ......................................... 36
      3.4.1 int_vector and bit_vector ............................... 36
      3.4.2 enc_vector ............................................ 37
   3.5 The rank_support and select_support Concept ............. 38
   3.6 The wavelet_tree Concept .................................. 41
      3.6.1 The Hierarchical Run-Length Wavelet Tree .......... 41
   3.7 The csa Concept ............................................ 45
      3.7.1 The Implementation of csa_uncompressed ............. 47
      3.7.2 The Implementation of csa_sada ........................ 47
      3.7.3 The Implementation of csa_wt .......................... 48
      3.7.4 Experimental Comparison of CSA Implementations .... 48
   3.8 The bp_support Concept .................................... 54
      3.8.1 The New bp_support Data Structure .................. 56
      3.8.2 The Range Min-Max-Tree ............................... 68
3.8.3 Experimental Comparison of $bp_{\text{support}}$ Implementations ........................................... 70
3.9 The $cst$ Concept ................................................................................................................. 78
  3.9.1 Sadakane's CST Revisited ................................................................................................. 78
  3.9.2 Fischer et al.'s Solution Revisited ...................................................................................... 80
  3.9.3 A Succinct Representation of the LCP-Interval Tree ......................................................... 80
  3.9.4 Node Type, Iterators, and Members ..................................................................................... 86
  3.9.5 Iterators ................................................................................................................................ 88
3.10 The $lcp$ Concept .................................................................................................................. 90
  3.10.1 The Tree Compressed LCP Representation ......................................................................... 91
  3.10.2 An Advanced Tree Compressed LCP Representation ........................................................ 93
  3.10.3 Experimental Comparison of LCP Implementations ....................................................... 95
4 Experimental Comparison of CSTs ............................................................................................ 100
  4.1 Existing Implementations ....................................................................................................... 100
  4.2 Our Test Setup ...................................................................................................................... 101
  4.3 Our Test Results .................................................................................................................... 105
    4.3.1 The $\text{parent}(v)$ Operation ........................................................................................... 105
    4.3.2 The $\text{Sibling}(v)$ Operation .......................................................................................... 105
    4.3.3 The $\text{ith_child}(v, 1)$ Operation .................................................................................... 105
    4.3.4 Depth-First-Search Traversals .......................................................................................... 109
    4.3.5 Calculating Matching Statistics ....................................................................................... 110
    4.3.6 The $\text{child}(v, c)$ Operation .......................................................................................... 110
    4.3.7 Conclusions ..................................................................................................................... 114
  4.4 The Anatomy of Selected CSTs ............................................................................................... 116
5 Construction ................................................................................................................................. 119
  5.1 Overview ................................................................................................................................ 119
  5.2 CST Construction Revisited ................................................................................................... 119
    5.2.1 CSA Construction ............................................................................................................. 119
    5.2.2 LCP Construction ............................................................................................................ 120
    5.2.3 NAV Construction .......................................................................................................... 121
    5.2.4 Fast Semi-External Construction of CSTs ....................................................................... 121
  5.3 A LACA for CST construction ................................................................................................ 126
    5.3.1 Linear Time LACAs Revisited .......................................................................................... 126
    5.3.2 Ideas for a LACA in the CST Construction .................................................................... 128
    5.3.3 The First Phase .............................................................................................................. 129
    5.3.4 The Second Phase .......................................................................................................... 132
    5.3.5 Experimental Comparison of LACAs ............................................................................. 134
6 Applications ................................................................................................................................. 138
  6.1 Calculating the $k$-th Order Empirical Entropy ..................................................................... 139
  6.2 Succinct Range Minimum Query Data Structures ............................................................... 141
  6.3 Calculating Maximal Exact Matches ..................................................................................... 145
    6.3.1 Motivation ...................................................................................................................... 145
1 Introduction

In sequence analysis it is often advantageous to build an index data structure for large texts, as many tasks – for instance repeated pattern matching – can then be solved in optimal time. The most popular and important index data structure for many years was the suffix tree, which was proposed in the early 1970s by Weiner [Wei73] and was later improved by McCreight [McC76]. Gusfield showed in [Gus97] the power of the suffix tree by presenting over 20 problems which can be solved in optimal time complexity with ease by using the suffix tree.

Despite its good properties — the suffix tree can be constructed in linear time for a text of length $n$ over a constant size alphabet of size $\sigma$, and most operations can be performed in constant time — it has a severe drawback. Its space consumption is at least 17 times the text size in practice [Kur99]. In the early 1990s Gonnet et al. [GBYS92] and Manber and Myers [MM93] introduced another index data structure called suffix array, which only occupies $n \log n$ bits which equals four times the text size, when the text uses the ASCII alphabet and the size $n$ of the text is smaller than $2^{32}$. The price for the space reduction was often a $\log n$ factor in the time complexity of the solution. However with additional tables like the longest common prefix array (LCP) and the child table it is possible to replace the suffix tree by the so called enhanced suffix array [AKO04], which requires in the worst case 12 times the text size ($n \log n$ bits for each table).

In the last decade it was shown that the space of the suffix tree can be further improved by using so called succinct data structures, which have an amazing property: their representation is compressed but the defined operations can still be performed in an efficient way. That is we do not have to first decompress the representation to answer an operation. For many such data structures the operations can be even performed in the same time complexities as the operations of its uncompressed counterpart. Most succinct data structures contain an indexing part to achieve this. This indexing part — which we will call support structure — takes often only $o(n)$ bits of space. On example for a succinct data structure is the compressed suffix array (CSA) of Grossi and Vitter [GV00]. They showed that the suffix array can be represented in essentially $O(n \log \sigma) + o(n \log \sigma)$ bits (instead of $O(n \log n)$ bits) and the access to an element remains efficient (time complexity $O(\log^\varepsilon n)$ for a fixed $\varepsilon > 0$).

Sadakane proposed in [Sad02] a succinct data structure for LCP, which takes $2n + o(n)$ bits, and later [Sad07a] also a succinct structure for the tree topology which also includes navigation functionality (NAV) and takes $4n + o(n)$ bits. In combination with his compressed suffix array proposal [Sad00] he got the first compressed suffix tree (CST). In theory this compressed suffix tree takes $nH_0 + O(n \log \log \sigma) + 6n + o(n)$ bits and supports many operations in optimal constant time. Therefore the suffix tree was replaced by the CST in theory. The unanswered question is if the CST can also replace the suffix tree.
in practice. Where replace means that applications, which use a CST, can outperform applications, which use a suffix tree, not only in space but also in time.

Note that the term “in practice” implies different scenarios. The first and weakest one is (S1) that the suffix tree does not fit into the main memory but the CST does. The second and most challenging scenario is (S2) that both data structures fit into the main memory. Finally the last one arises from the current trend of elastic cloud computing, where one pays the used resources per hour. The price for memory usage typically increases linearly, cf. Table 1.1. This implies for instance that an application $A$ which takes $s_A$ MB of main memory and completes the task in $t_A$ hours is superior to an application $B$ which uses $2s_A$ MB of main memory and does not complete the task in $\leq \frac{1}{2} t_A$ hours. So scenario (S3) is that the application which uses a CST is more economical than the suffix tree application.

Välimäki et al. [VMGD09] were the first who implemented the proposal of Sadakane. The implementation uses much more space than expected: up to $35n$ bits for texts over the ASCII alphabet. Furthermore it turned out that the operations were so slow that the CST can only replace the suffix tree in the scenario S1.

In 2008, Russo et al. [RNO08] proposed another CST which uses even less space; essentially the size of a CSA plus $o(n)$ bits. Its implementation takes in practice only 4-6 bits but the runtime of most of its operations are orders of magnitudes slower than that of Sadakane’s suffix tree, but there are applications where this CST can also replace the suffix tree in the scenario S1.

Recently, Fischer et al. [FMN09] made another proposal for a CST, which also has a better theoretical space complexity than Sadakane’s original proposal but is faster than Russo et al.’s proposal. Cánovas and Navarro [CN10] showed that their implementation provides a very relevant time-space trade-off which lies between the implementation of Välimäki and Russo. Their implementation seems to replace the suffix tree not only in scenario S1 but also in S3. However we show that the runtime of some operations in their implementation is not yet robust which causes a slowdown up to two orders of magnitude for some inputs.

The goal of this thesis is to further improve the practical performance of CSTs. The thesis is organized as follows.

In Chapter 2 we present previous work which is relevant for the next chapters: A selection of support data structures, algorithms and important connection between

<table>
<thead>
<tr>
<th>Instance name</th>
<th>main memory</th>
<th>price per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro</td>
<td>613.0 MB</td>
<td>0.02 USD</td>
</tr>
<tr>
<td>High-Memory Extra Large</td>
<td>17.1 GB</td>
<td>0.50 USD</td>
</tr>
<tr>
<td>High-Memory Double Extra Large</td>
<td>34.2 GB</td>
<td>1.00 USD</td>
</tr>
<tr>
<td>High-Memory Quadruple Extra Large</td>
<td>68.4 GB</td>
<td>2.00 USD</td>
</tr>
</tbody>
</table>

Table 1.1: Pricing of Amazons Elastic Cloud Computing (EC2) service in July 2011.
Source: [Ama].
indexing and compression.

In Chapter 3 we first present a design of a CST which makes it possible to build up a CST using any combination of proposals for CSA, LCP, and NAV. We then present implementation variants of every data structure along with experimental results, to get an impression of their real-world performance. Furthermore we present our theoretical proposals for a CST, which was partly published in [OG09], [GF10], and [OFG10]. Sections 3.6.1 and 3.10 contain new and not yet published data structures.

In Chapter 4 we give the first fair experimental comparison of different CST proposals, since we compare CSTs which are built up from the same set of basic data structures.

In Chapter 5 we present a semi-external construction method for CSTs which outperforms previous construction methods in time and space. Parts of this chapter were published in [GO11]. This Chapter also shows, that there are CSTs which can be constructed faster than the uncompressed suffix tree.

In Chapter 6 we show that our implementation can replace the suffix tree in the S2 scenario in some applications. That is we get a faster running time and at the same time use less main memory than solutions which use uncompressed data structures. Parts of this chapter were published in [OGK10].

We would like to emphasize that all data structures, which are described in detail in this thesis, are available in a ready-to-use C++ library.
2 Basic Concepts

2.1 Notations

2.1.1 Strings

A string $T = T[0..n-1] = T_0T_1 \ldots T_{n-1}$ is a sequence of $n = |T|$ characters over an ordered alphabet $\Sigma$ (of size $\sigma$). We denote the empty string by $\varepsilon$. Each string of length $n$ has $n$ suffixes $s_i = T[i..n-1]$ ($0 \leq i < n$). We define the lexicographic order “$<$” on strings as follows: $\varepsilon$ is smaller than all other strings. Now $T < T'$ if $T_0 < T'_0$ or if $T_0 = T'_0$ and $s_1 < s'_1$.

In this thesis the alphabet size $\sigma$ will be most of the time be considered as constant. This reflects the practice, were we handle DNA sequences ($\sigma = 4$), amino acid sequences ($\sigma = 21$), ASCII strings ($\sigma = 128$), and so on. It is also often convenient to take the effective alphabet size, i.e. only the number of different symbols in the specific text. As the symbols are lexicographically ordered we will map the lexicographically smallest character to 0, the seconds smallest character to 1, and so on. This mapping is important when we us a character as index for an array. This mapping will be done implicitly in pseudo-code.

We assume that the last character of every string $T$ is a special sentinel character, which is the lexicographic smallest character in $T$. In the examples of this thesis we use the $\$-symbol as sentinel character, in practice the 0-byte is used.

We will often use a second string $P$ which is called pattern. The size $|P|$ is, if not stated explicitly, very small compared to $n$, i.e. $|P| \ll n$. Throughout the thesis we will use the string $T=umulmundumulmum$ as our running example.

2.1.2 Trees

We consider ordered trees in this thesis. That is the order of the children is given explicitly by an ordering function, e.g. the lexicographic order of the edge labels. Nodes are denoted by symbols $v, w, x, y, z$. An edge or the shortest sequence of edges from a node $v$ to $w$ is denoted by $(v,w)$. $L(v,w)$ equals the concatenation of edge labels on the shortest path from $v$ to $w$, when edge labels are present. Nodes which have no children are called leaf nodes and all other nodes are called inner nodes. A node with more than one child node is called branching node. A tree which has only branching nodes is called a compact tree.
2.1.3 Machine Model, $O(\cdot)$-Notation, and Test Environment

We use the RAM model with word size $\log n$, i.e. we can random access and manipulate $\log n$ consecutive bits in constant time. This RAM model is close to reality except for the time of the random access. In practice, the constants between the different levels in the memory hierarchy (see Figure 2.1) are so big, that we consider only the top levels (CPU registers, caches, and RAM) as practically constant. We also assume that the pointer size equals the word size and a implementation today should be written for a word size of 64 bits, as most of the recent CPUs support this size. This also avoid that the implementation is limited to string of size at most 2 or 4 GB, depended on the use of signed or unsigned integers. When we use integer divisions like $x/y$, the result is always rounded to the greatest integer smaller or equal to $x/y$, i.e. $\lfloor x/y \rfloor$. The default base of the logarithm function $\log$ is 2 throughout the thesis.

We avoid to use the $O(\cdot)$-notation for space complexities since constants are really important in the implementation of compressed or succinct data structures. For instance it is common to take the word size as a constant of 4 bytes or 32 bit and often the logarithm of the alphabet size as constant 1 byte or 8 bits. And therefore a compressed suffix tree could not be distinguished from an ordinary suffix tree by the $O(\cdot)$-notation. The proper calculation of constants often shows that milestones like “Breaking a Time-and-Space Barrier in Constructing Full-Text Indices”\cite{HSS03} are only of theoretical interest.

All experiments in this thesis were performed on the following machine: A Sun Fire\textsuperscript{TM} X4100 M2 server equipped with a Dual-Core AMD Opteron\textsuperscript{TM} 1222 processor which runs with 3.0 GHz on full speed mode and 1.0 GHz in idle mode. The cache size of the processor is 1 MB and the server contains 4 GB of main memory. Only one core of the processor cores was used for experimental computations. We run the 64 bit version of OpenSuse 10.3 (Linux Kernel 2.6.22) as operating system and use the gcc compiler version 4.2.1. All programs were compiled with optimization (option -O9) and in case of performance tests without debugging (option -DNDEBUG).
2.1 Notations

2.1.4 Text Compressibility

In this section we present some basic results about the compressibility of strings and its relation to the empirical entropy.

Let \(n_i\) be the number of occurrences of symbol \(\Sigma[i]\) in \(T\). The zero-order empirical entropy of \(T\) is defined as

\[
H_0(T) = \sum_{i=0}^{\sigma-1} \frac{n_i}{n} \log \frac{n}{n_i}
\]  

(2.1)

where \(0 \log 0 = 0\). We simply write \(H_0\), if the string \(T\) is clear from the context. If we replace each character of \(T\) by a fixed codeword (i.e. the codeword is not depended on the context of the character), then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.

Manzini, Makinen, and Navarro \([Man01, MN04]\) use another measure for the encoding size. If we are allowed to choose the codeword for a character \(T[i]\) depending on the context of the character, then \(nH_0\) bits is the smallest encoding size of \(T\). In practice the Huffman code\([Huf52]\) is used to get close to this bound. We get \(H_0(T) \approx 2.108\) bits for our example text \(T=umlumdumulumum\$\), since there are 6 \(u\), 5 \(m\), 2 \(1\), and three single characters \(\$, \d\), and \(n\). Therefore \(H_0 = \frac{3}{8} \log \frac{8}{3} + \frac{5}{16} \log \frac{16}{5} + \frac{1}{8} \log 8 + 3(\frac{1}{16} \log 16) \approx 2.108\)bits, i.e. we need at least \(n \cdot H_k = 16 \cdot 2.108 > 34\) bits to encode the example text. This is already much less than the uncompressed ASCII representation which takes \(16 \cdot 8 = 96\) bits. But it is possible to further improve the compression.
Table 2.1: \( H_k \) for the 200MB test cases of the Pizza\&Chili corpus. The values are rounded to 3 digits.

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\multicolumn{1}{|c|}{k} & \multicolumn{1}{|c|}{dblp.xml} & \multicolumn{1}{|c|}{dna} & \multicolumn{1}{|c|}{english} & \multicolumn{1}{|c|}{proteins} & \multicolumn{1}{|c|}{rand_k128} & \multicolumn{1}{|c|}{sources} \\
\hline
0 & 5.257 & 0.0000 & 1.974 & 0.0000 & 4.525 & 0.0000 & 4.201 & 0.0000 & 7.000 & 0.0000 & 5.466 & 0.0000 \\
1 & 3.479 & 0.0000 & 1.930 & 0.0000 & 3.620 & 0.0000 & 4.178 & 0.0000 & 7.000 & 0.0000 & 4.077 & 0.0000 \\
2 & 2.170 & 0.0000 & 1.920 & 0.0000 & 2.948 & 0.0001 & 4.156 & 0.0000 & 6.993 & 0.0001 & 3.102 & 0.0000 \\
3 & 1.434 & 0.0007 & 1.916 & 0.0000 & 2.422 & 0.0005 & 4.066 & 0.0001 & 5.979 & 0.0100 & 2.337 & 0.0012 \\
4 & 1.045 & 0.0043 & 1.910 & 0.0000 & 2.063 & 0.0028 & 3.826 & 0.0011 & 0.666 & 0.6939 & 1.852 & 0.0082 \\
5 & 0.817 & 0.0130 & 1.901 & 0.0000 & 1.839 & 0.0103 & 3.162 & 0.0173 & 0.006 & 0.9969 & 1.518 & 0.0250 \\
6 & 0.705 & 0.0265 & 1.884 & 0.0001 & 1.672 & 0.0265 & 1.502 & 0.1742 & 0.000 & 1.0000 & 1.259 & 0.0509 \\
7 & 0.634 & 0.0427 & 1.862 & 0.0001 & 1.510 & 0.0553 & 0.340 & 0.4506 & 0.000 & 1.0000 & 1.045 & 0.0850 \\
8 & 0.574 & 0.0598 & 1.834 & 0.0004 & 1.336 & 0.0991 & 0.109 & 0.5383 & 0.000 & 1.0000 & 0.867 & 0.1255 \\
9 & 0.537 & 0.0773 & 1.802 & 0.0013 & 1.151 & 0.1580 & 0.074 & 0.5588 & 0.000 & 1.0000 & 0.721 & 0.1701 \\
10 & 0.508 & 0.0955 & 1.760 & 0.0051 & 0.963 & 0.2292 & 0.061 & 0.5699 & 0.000 & 1.0000 & 0.602 & 0.2163 \\
\hline
\end{array}
\]

\( k = 5 \). If we have to store only 100 bits information per context, then the compression to \( n H_5 + 0.28n = 2.343 \) bits is better than the compression to \( n H_5 + 1.03n = 2.869 \).

Nevertheless \( H_k \) can be used to get a lower bound of the bits which are needed by an entropy compressor which uses a context of at most \( k \) characters. However, the calculation of \( H_k \) even for small \( k \) cannot be done straightforwardly for reasonable \( \sigma \) since this results in a time complexity of \( O(\sigma^k \cdot n) \) and it would have taken days to compute Table 2.1. Fortunately, the problem can be solved easily in linear time and space by using a suffix tree, which will be introduced in the next section. Readers which are already familiar with suffix trees are referred to Section 6.1 for the solution.

### 2.2 The Suffix Tree

In this section we will introduce the suffix tree data structure. We first give the definition which is adopted from [Gus97].

**Definition 1** A suffix tree of a string \( T \) of length \( n \) fulfills the following properties:

- It is a rooted directed ordered tree with exactly \( n \) leaves numbered 0 to \( n - 1 \).
- Each inner node has at least two children. We call such a node branching.
- Each edge is labeled with a nonempty substring of \( T \).
- No two edges out of a node can have edge-labels beginning with the same character.
- The concatenation of the edge-labels on the path from the root to a leaf \( i \) equals suffix \( i \ T[i..n - 1] \).

Figure 2.2 depicts a suffix tree of the string \( T = umil mund umil mum $ \). The direction of the edges is not drawn as we always assume that it is from top to bottom. The edges and the inner nodes are drawn blue. The children \( w_i \) of a node \( v \) are always ordered from
left to right according to the first character of the edge-label of \((v, w_i)\). This ordering implies that we get the lexicographically smallest suffix of \(T\) if we follow the path from the root to the leftmost leaf in the tree. Or more generally, we get the lexicographically \(i\)-th smallest suffix of \(T\) if we follow the path from the root to the \(i\)-th leftmost leaf.

The virtue of the suffix tree is that we can solve problems, which are hard to solve without an index data structure, very efficiently. The most prominent application is the substring problem. In its easiest version it is a decision problem. I.e. we have to decide if a string \(P\) is contained in \(T\). This problem can be solved in optimal \(O(|P|)\) time complexity with a suffix tree. We will sketch the solution with an example. Assume we want to determine if \(P = \text{mulm}\) occurs in the string \(T = \text{umulmundumulmum}$ (see Figure 2.2 for the suffix tree of \(T\)). All substrings of \(T\) can be expressed as a prefix of a suffix of \(T\). Therefore we try to find a prefix of a suffix of \(T\), which matches \(P\). This implies that we have to start at the root node \(v_0\) of the suffix tree. As \(P\) starts with character \(m\), we have to search for a child \(v_1\) of \(v_0\) for which \(L(v_0, v_1)\) starts with \(m\). Such a \(v_1\) is present in the example. Since the edge-label contains only one character, we can directly move to \(v_1\). Now we have to find a child \(v_2\) of \(v_1\) for which \(L(v_1, v_2)\) starts with the second character \(u\) of \(P\). Once again, such a \(v_2\) exists and since the edge-label contains only one character we move directly to \(v_2\). Now, \(v_2\) has also a child \(v_3\) for which \(L(v_2, v_3)\) starts with the next character \(l\) of \(P\). But this time the edge-label contains more than one character and therefore we have to check if all remaining characters of \(P\) match the characters of \(L(v_2, v_3)\). This is the case in our example and so we can answer that \(P\) occurs somewhere in \(T\). We can even give a more detailed answer. As the subtree rooted at \(v_3\) contains two leaves, we can answer that \(P\) occurs two times in \(T\); so we have solved the counting version of the substring problem. In addition to that we can also solve the locating version of the problem, since the leaf nodes contain the starting positions of the \(z\) occurrences of \(P\) in \(T\). Therefore the occurrences of \(P\) in \(T\) start at

![Figure 2.2: The suffix tree of string T=umulmundumulmum$.](attachment:image.png)
positions 9 and 1. The time complexity of the decision and counting problem depends on the representation of the tree. We will see that both problems can be solved also in optimal time complexities $O(|P|)$ and $O(|P| + z)$.

Let us now switch to the obvious drawbacks of the suffix tree. At first, the space complexity of the tree turns out to be $O(n^2 \log \sigma)$ bits if we store all edge-labels explicitly. E.g. the suffix tree of the string $a^{n-1}ba^{n-1}\$ (for even $n$) contains $\frac{n}{2}$ edge labels $ba^{n-1}\$. Fortunately, the space can be reduced to $O(n \log n)$ bits as follows: First we observe that the number of nodes is upper bounded by $2n - 1$, since there are exactly $n$ leaves and each inner node is branching. We can store the text $T$ explicitly and for every node $v$ a pointer to the starting position of one occurrence of the $\mathcal{L}(p,v)$ in $T$, where $p$ is the parent node of $v$. In addition to that, we also store the length $|\mathcal{L}(p,v)|$ at node $v$. Now, the space complexity is in $O(n \log n)$ bits. However the constant hidden in the $O(\cdot)$ notation is significant in practice. A rough estimate which considers the pointers for edge-labels, children, parents and in addition the text, lengths of the edge-labels, and leaf labels results in at least $n \log \sigma + 7 \log n$ bits. For the usual setup of text over ASCII alphabet and pointers of size 4 bytes the size of the suffix tree is about 25 times the size of the original text. Things are even worse, if we use a 64 bit architecture, have text of very small alphabet size like DNA sequences, or would like to support additional navigation features in the tree. We will show in this thesis how we can get a compressed representation of the suffix tree which at the same time allows time-efficient execution of navigational operations.

One might expect that the second disadvantage is the construction time of the suffix tree. Clearly, the naive way of doing that would be to sort all suffixes in $O(n^2 \log n)$ time and then insert the suffixes in the tree. Surprisingly, this problem can be solved in linear time. Weiner [Wei73] proposed a linear time algorithm in the early 1970s. Donald Knuth, who conjectured 1970, that an linear-time solution for the longest common substring problem of two strings would be impossible, characterized Weiner’s work as “Algorithm of the Year 1973”. Later, McCreight [McC76] and Ukkonen [Ukk95] proposed more space efficient construction algorithms which are also easier to implement than Weiner’s algorithm.

### 2.3 The Suffix Array

We have learned in the last chapter that the big space consumption is the major drawback of the suffix tree. Another index data structure which uses less space is the suffix array. Gonnet et al. [GBYS92] and Manber and Myers [MM93] were the first who discovered the virtue of this data structure in the context of string matching. The suffix array of a text is an array $\mathbf{SA}[0..n-1]$ of length $n$ for a text $T$ which contains the lexicographic ordering of all suffixes of $T$; i.e. $\mathbf{SA}[0]$ contains the starting position of the lexicographical smallest suffix of $T$, $\mathbf{SA}[1]$ contains the starting position of the lexicographical second smallest suffix, and so on. We have already seen a suffix array in Figure 2.2: It is the concatenation of all leaf node labels from left to right. Clearly, this data structure occupies only $n \log n$ bits of space and we will illustrate later that the substring problem can still be solved.
very fast. Before that let us focus on the construction algorithm. It should be clear that the suffix array can be constructed in linear time, since the suffix tree can be constructed in that time. However, the construction algorithm for suffix trees requires much space and therefore a direct linear solution was searched for a long time. In 2003 three groups [KS03, KA03, KSPP03] found independently and concurrently linear time suffix array construction algorithms (LACAs). However, the implementations of the linear time algorithms were slower on real world inputs than many optimized $O(n^2 \log n)$ worst case solutions. Puglisi et al. published an overview article which describes the development and the performance of LACAs until 2007. Today, there exists one linear time algorithm [NZC09] which can compete with most of the optimized $O(n^2 \log n)$ LACAs. We refer to Chapter 5 for more details.

Let us now solve the substring problem using only $SA$, $P$, and $T$. Figure 2.3 depicts the $SA$ for our running example and to the right of $SA$ we placed the corresponding suffixes. To find the pattern $P = \text{umu}$ we first determine all suffixes which start with $u$. This can be done by two binary searches on $SA$. The first one determines the minimum index $\ell$ in $SA$ with $T[SA[\ell]+0] = P[0]$, and the second determines the maximum index $r$ with $T[SA[r]+0] = P[0]$. So in our example we get the $SA$ interval $[10,15]$. Note that we call a $SA$ interval $[i,j]$ an $\omega$-interval, when $\omega$ is a substring of $T$ and $\omega$ is a prefix of $T[SA[k]..n-1]$ for all $i \leq k \leq j$, but $\omega$ is not a prefix of any other suffix of $T$. So $[10,15]$ is also called $u$-interval. In the second step of the search we search for all suffixes in the interval $[10,15]$, which start with $um$. One again this can be done by two binary searches. This time we have to do the binary searches on the second character of the suffixes. I.e. we determine the minimum (maximum) index $\ell'$ ($r'$) with $T[SA[\ell']+1] = P[1]$ and $\ell \leq \ell' \leq r' \leq r$. In our example we get $[\ell',r'] = [12,14]$. The next step does the binary searches on the third character of the suffixes and we finally get the $\text{umu}$-interval $[13,14]$.

It is easy to see, that the substring problem can be solved in $O(|P| \log n)$ with this procedure. Moreover, the counting problem can be solved in the same time complexity.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$SA$</th>
<th>$SA^{-1}$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>14</td>
<td>$$</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>6</td>
<td>dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>lnum$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>lmundumulmum$</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8</td>
<td>m$</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>15</td>
<td>mum$</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>9</td>
<td>mumulmundumulmum$</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>1</td>
<td>mum$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>13</td>
<td>mundumulmum$</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>5</td>
<td>ndumulmum$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>ulmum$</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>ulmundumulmum$</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>7</td>
<td>um$</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>12</td>
<td>umulmum$</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>4</td>
<td>umulmundulmum$</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>0</td>
<td>undulmum$</td>
</tr>
</tbody>
</table>

Figure 2.3: The suffix array of the text $T=\text{umulmundulmum}$.
as the size of the interval can be calculated in constant time. The presented matching
procedure is called \textit{forward search}, as we have matched the pattern from left to right
against the index.

Let us now switch to the basic building blocks of the compressed representation of the
suffix array. First of all, the \textit{Burrows-Wheeler Transform} $T^{\text{BWT}}$ of the string $T$. It can
be defined by $T$ and $SA$ as follows:

$$T^{\text{BWT}}[i] = T[SA[i] - 1 \mod n] \tag{2.3}$$

I.e. $T^{\text{BWT}}[i]$ equals the character which directly precedes suffix $SA[i]$ in the cyclic rotation
of $T$. In uncompressed form the Burrows-Wheeler Transform takes $n \log \sigma$ bits of memory,
which is exactly the size of the input string. However, $T^{\text{BWT}}$ is often better compressible$^1$, since adjacent elements in $T^{\text{BWT}}$ are sorted by their \textit{succeeding context} and we get many
runs of equal characters. Often the number of runs is denoted by $R$.

Mäkinen and Navarro [MN04, MN05] proved the following upper bound for the runs
in the $T^{\text{BWT}}$:

$$R \leq nH_k + \sigma^k \tag{2.4}$$

This inequality is used in many papers to get a entropy-bounded space complexity for
succinct data structures. However, in practice the upper bound is very pessimistic. A
typical example for that is the \textit{Pizza&Chili} corpus. For $k \leq 5$ only $H_k$ of the highly-
repetitive \texttt{dp1p.xml} test case is smaller than 1 while all other text have $H_k$ value greater
1, cf. Figure 2.1.

Even the small example in Figure 2.4 has three runs of length greater than or equal
to two. These runs can be compressed by using e.g. first move to front and then
Huffman coding. This results in a compression which is dependent on the entropy of
the input string (see the paper of [Man01]). Despite interesting compression variants of
the Burrows-Wheeler Transform, we will focus only on very simple and fast compression
schemata in this thesis which compress $T^{\text{BWT}}$ to the 0-order entropy. We postpone
a detailed discussion of the properties of the Burrows-Wheeler Transform to the next
section. The only thing we should bear in mind is that we get $T^{\text{BWT}}[i]$ in constant time
if $T$ and $SA[i]$ are present.

Before we will present the next two basic building blocks which are also closely related
to the Burrows-Wheeler Transform we have to introduce the inverse suffix array, called
ISA, which is the inverse permutation of $SA$. With help of ISA, we can answer the question
“what is the index of suffix $i$ in $SA$?” in constant time. Now we can express the following
building blocks in terms of $SA$ and ISA.

The \textit{LF-mapping} or \textit{LF-function} is defined as follows:

$$\text{LF}[i] = \text{ISA}[(SA[i] - 1) \mod n] \tag{2.5}$$

$^1$ Note: The popular software \texttt{bzip2} is based on the Burrows-Wheeler Transform.
2.3 The Suffix Array

I.e. we get for suffix \( j = SA[i] \) the index of suffix \( j - 1 \), which directly precedes suffix \( j \) in the cyclic rotation of \( T \), in \( SA \). The \( \Psi \)-function goes the other way around:

\[
\Psi[i] = ISA[(SA[i] + 1) \mod n]
\]

(2.6)

I.e. we get for suffix \( j = SA[i] \) the index of suffix \( j + 1 \), which directly succeeds suffix \( j \) in the cyclic rotation of \( T \), in \( SA \). So, in presence of \( SA \) and \( ISA \) both functions can be calculated in constant time. Also note that \( \Psi \) and \( LF \) are like \( SA \) and \( ISA \) inverse permutations of each other. Sometimes it is also useful to apply \( \Psi \) or \( LF \) \( k \) times on an initial value \( i \). This can be done faster by the following equations:

\[
\Psi^k[i] = ISA[(SA[i] + k) \mod n]
\]

(2.7)

\[
LF^k[i] = ISA[(SA[i] - k) \mod n]
\]

(2.8)

The correctness can be proven easily by induction.

Let us now switch back to the compressibility of \( \Psi \) and \( LF \). Both permutations can be decomposed into few sequences of strictly increasing integers. While \( \Psi \) consists of at most \( \sigma \) increasing sequences, \( LF \) consists of at most \( \mathcal{R} \) increasing sequences. We can check this in Figure 2.4. The \( \Psi \) entries \( \Psi[i], \Psi[i + 1], \ldots, \Psi[j] \) are always increasing if the first character of suffixes \( SA[i], SA[i + 1], \ldots, SA[j] \) are equal. E.g. take the range \([4,8]\). All suffixes start with character \( m \) and \( \Psi[4,8] = 0,10,11,12,15 \). The \( LF \) values increase in a range \([i,j]\) if all characters in \( T^{BW}[i,j] \) are equal. E.g. all characters in the range \([2,6]\) of the \( T^{BW} \) equal \( u \), therefore \( LF[2,6] = 10,11,12,13,14 \). Note that also the difference of two adjacent \( LF \) values is exactly one.

The first compressed suffix arrays of Grossi and Vitter [GV00] and an improved version of Sadakane [Sad00] is based on the \( \Psi \) function. They exploited the fact, that each of the
2 Basic Concepts

2.4 The Operations Rank and Select

In 1989 Jacobson published the seminal paper titled “Space-efficient Static Trees and Graphs” [Jac89]. From today’s point of view this paper is the foundation of the field of succinct data structures. We define a succinct data structure as follows: Let $D$ be an abstract data type and $X$ the underlying combinatorial object. A succinct data structure for $D$ uses space “close” to the information theoretical lower bound of representing $X$ while at the same time the operation on $D$ can still be performed “efficiently”. E.g. Jacobson suggested a succinct data structure for unlabeled static trees with $n$ leaves. There exists $\frac{1}{n-1} \binom{2n}{n}$ such trees and therefore the information theoretical lower bound to represent a trees is $\log \left( \frac{1}{n-1} \binom{2n}{n} \right) = 2n + o(n)$ bits. Jacobson presented a solution which takes $2n + o(n)$ space and supports the operations parent, first_child, and next_sibling in constant time. This solution gives also an intuition for the terms “close” and “efficient”. In almost all cases, the time complexities for the operations remain the same as in the uncompressed data structures. At least in theory. The term “close” means most of the time “with sub-linear extra space” on top of the information theoretical lower bound.

The core operations which enable Jacobson’s solution are the rank and the select operation. The rank operation $\text{rank}_A(i,c)$ counts the number of occurrences of a symbol $c$ in the prefix $A[0..i-1]$ of a sequence $A$ of length $n$. In most cases the context of the rank data structure is clear and we will therefore omit the sequence identifier and write simply $\text{rank}(i,c)$. Often we also omit the symbol $c$, since the default case is that the sequence $A$ is a bit vector and $c$ equals a 1-bit. It is not hard to see, that in the default case the rank query for $\text{rank}(i,0)$ can be calculated by $\text{rank}(i,0) = i - \text{rank}(i)$.

The select operation $\text{select}_A(i,c)$ returns the position of the $i$-th ($1 \leq i \leq \text{rank}_A(n,c)$) occurrence of symbol $c$ in the sequence $A$ of length $n$. Like in the rank case, we omit the subscript $A$ if the sequence is clear form the context. Once again, the default case is that $A$ is a bit vector but this time we cannot calculate $\text{select}(i,0)$ from $\text{select}(i)$. Therefore,
we have to build two data structures to support both operations.

It was shown in [Jac89, Cla96] how to answer rank and select queries for bit vectors in constant time using only sub-linear space by using a hierarchical data structure in combination with the Four Russians Trick [ADKF70]. The generalization to sequences of bigger alphabets is presented in Section 2.6.

2.5 The Burrows-Wheeler Transform

We have already defined the Burrows-Wheeler transformed string $T^{BWT}$ in Section 2.3. In this section, we will describe the relation of $T^{BWT}$ with the LF and $\Psi$ function. Before doing this, we first have a look at the interesting history of the Burrows-Wheeler transform. David Wheeler had the idea of the character reordering already in 1978. It then took 16 years and a collaboration with Michael Burrows until the seminal technical report at Digital Equipment Corporation [BW94] was published. The reader is referred to [ABM08] for the historical background of this interesting story. Today, the Burrows-Wheeler Transform plays a key role in text data compression. The most prominent example for a compressor based on this transformation is the bzip2\(^1\) application. However, one can not only compress the text but also index it. To show how this is possible, we have to present the relation between the LF and $\Psi$ function and $T^{BWT}$.

Figure 2.5 (a) shows once again LF, $T^{BWT}$ and the sorted suffixes of $T$. Now remember that the LF function at position $i$ tells us for suffix $SA[i]$ the previous suffix in the text, i.e. the position of suffix $SA[i] - 1$. We take for example suffix $SA[4] = 14$ which spells out $m\$$. Now, as $T^{BWT}[4] = u$, we know that suffix 13 starts with character $u$. I.e.

\begin{figure}[h]
\centering
\begin{tabular}{cccccccc}
\hline
\textbf{i} & \textbf{LF} & \textbf{$T^{BWT}$} & \textbf{F} & \textbf{i} & \textbf{LF} & \textbf{$T^{BWT}$} & \textbf{F} \\
\hline
0 & 4 & m & $\$u$ & 14 & 5 & m & $\$um\$ \\
1 & 9 & m & $\$um\$ & 13 & 6 & m & $\$ulmum\$ \\
2 & 10 & n & $\$um\$ & 7 & 11 & n & $\$ulmum\$ \\
3 & 11 & u & $\$ulmum\$ & 8 & 12 & u & $\$ulmum\$ \\
4 & 12 & u & $\$ulmum\$ & 9 & 13 & u & $\$ulmum\$ \\
5 & 13 & u & $\$ulmum\$ & 10 & 14 & u & $\$ulmum\$ \\
6 & 14 & u & $\$ulmum\$ & 11 & 15 & u & $\$ulmum\$ \\
7 & 2 & 1 & $\$um\$ & 12 & 1 & 1 & $\$um\$ \\
8 & 3 & 1 & $\$um\$ & 13 & 2 & 1 & $\$um\$ \\
9 & 15 & u & $\$ulmum\$ & 14 & 1 & 1 & $\$um\$ \\
10 & 5 & m & $\$ulmum\$ & 15 & 1 & 1 & $\$um\$ \\
11 & 6 & m & $\$ulmum\$ & 16 & 1 & 1 & $\$um\$ \\
12 & 7 & m & $\$ulmum\$ & 17 & 1 & 1 & $\$um\$ \\
13 & 1 & d & $\$um\$ & 18 & 1 & 1 & $\$um\$ \\
14 & 0 & $\$ & $\$um\$ & 19 & 1 & 1 & $\$um\$ \\
15 & 8 & m & $\$um\$ & 20 & 1 & 1 & $\$um\$ \\
\hline
\end{tabular}

(a) \hspace{1cm} (b)
\end{figure}

Figure 2.5: The arrows depict the (a) LF function and (b) the $\Psi$ function and how it can be expressed by $T^{BWT}$ and $F$ for the running example $T = umulmundumum$.  

\(^1\) http://www.bzip.org
suffix 13 lies in the u-interval. In our example this interval equals [10,15]. The amazing thing is that we can get the exact position of suffix 13. We only have to count the number of characters u in $T^{BWT}$ before position 4, i.e. $\text{rank}(4,u)$. In our example we get $\text{rank}(4,u) = 2$. If we add this number to the lower bound of the interval [10,15] we get the position $10 + 2 = 12 = LF[4]$ of suffix 13 in SA! We will now explain, why this is correct. Note that since all suffixes SA[j] in the u-interval start with the same character the lexicographical ordering is determined solely by the lexicographical ordering of the subsequent suffixes $T[SA[j] + 1..n]$ in the text. I.e. all characters u before position 4 in $T^{BWT}$ result in suffixes which lie in the u-interval and are lexicographically smaller than suffix 13.

Now, we analyze which information is needed to calculate LF in that way. Figure 2.5 depicts all sorted suffixes. We need only the first row of the sorted suffixes, called F, to determine the SA interval of a character c. However, storing F would be a waste of memory, since we can only store the $\sigma$ borders of the intervals. I.e. we store for each character c the first occurrence of c in F in an array C. For our example we get: $C[S] = 0$, $C[d] = 1$, $C[1] = 2$, $C[m] = 4$, $C[n] = 9$, $C[u] = 10$. Array C occupies only $\sigma \log n$ bits. In addition to C, we have to store $T^{BWT}$ and a data structure which enables rank queries on $T^{BWT}$. We postpone the presentation of such a rank data structure and first present the equation for LF:

$$LF[i] = C[T^{BWT}[i]] + \text{rank}_{BWT}(i, T^{BWT}[i])$$  (2.9)

I.e. the time to calculate LF mainly depends on the calculation of $T^{BWT}[i]$ and a rank query on $T^{BWT}$.

The remaining task is to explain the relation between the $\Psi$ function and $T^{BWT}$. Figure 2.5 (b) depicts $\Psi$, $T^{BWT}$, and F. The $\Psi$ function is shown two times in the figure. First as array and second as arrows pointing from position i to $\Psi(i)$. As LF and $\Psi$ are inverse functions the arrows in 2.5 (a) and 2.5 (b) point to the opposite direction. I.e. while LF[i] points to a suffix which is one character longer than suffix SA[i], $\Psi[i]$ points to a suffix which is one character shorter than SA[i].

We take position 7 to demonstrate the calculation process of $x = \Psi[7]$. Suffix SA[7] starts with character $F[i] = m$. Therefore, it holds that $T^{BWT}[x]$ has to be $m$. We also know that SA[7] is the lexicographically 4th smallest suffix which starts with $m$. I.e. the position of the 4th $m$ in $T^{BWT}$ corresponds to $x$. By replacing the access to F by a binary search on C we get the following formula for $\Psi$:

$$\Psi[i] = \text{select}_{BWT}(i - C[c] + 1, c), \text{where } c = \min\{c \mid C[c] \geq i\}$$  (2.10)

I.e. the time to calculate $\Psi$ depends mainly on calculating the \text{select} query on $T^{BWT}$ and the $O(\log \sigma)$ binary search on C.
2.6 The Wavelet Tree

2.6.1 The Basic Wavelet Tree

In the last section we have learned how to calculate LF and ψ with a data structure which provides random access, rank, and select on a string. Now we will introduce this data structure which is called wavelet tree [GGV03]. First we assign to each character $c \in \Sigma$ a unique codeword $\text{code}(c) \in \{0,1\}^\lceil \log_\sigma \rceil$. E.g. we can assign each character a fixed-length binary codeword in $\{0,1\}^\lceil \log_\sigma \rceil$ or use a prefix-code like the Huffman code [Hu52] or the Hu-Tucker code [HT71].\(^1\) The wavelet tree is now formed as follows; see Figure 2.6 (a) for an example. The root node $v$ is on level 0 and contains the text $S$ of length $n$. We store a bit vector $b_v$ of length $n$ which contains at position $i$ the most significant bit $\text{code}(S[i])[0]$ of character $S[i]$. Now the tree is built recursively: If a node $v$ contains a text $S_v$ which contains at least two different characters, then we create two child nodes $v_\ell$ and $v_r$. All characters which are marked with a 0 go to the left node $v_\ell$ and all other characters go to the right node. Thereby the order of the characters is preserved. Note, that we store in the implementation only the bit vectors, the rank and select data structures for the bit vectors, and two maps $c \to \text{leaf}$ and $\text{leaf} \to c$, which map each character $c$ to its leaf node and vice versa. By using the fixed-length code the depth of the tree is bounded by $\lceil \log_\sigma \rceil$. As each level, except the last one, contains $n$ bits, we get a space complexity of about $n \log_\sigma + 3\sigma \log n + o(n)$ bits. I.e. almost the same space as the original sequence for small alphabets.\(^2\)

By using a fixed-length code we get a time complexity of $O(\log_\sigma)$ for the three basic operations. Pseudo-code for rank, select, and random access is given in Algorithm 1, 2, and 3.

The number of bits of all bit vectors $b_v$ can be further compressed if we use not a fixed-length code but a prefix-code, e.g. Huffman code. Let $n_c$ be the number of occurrences of character $c$ in $T$. Then Huffman code minimizes the sum $\sum_{c \in \Sigma} n_c \cdot |\text{code}(c)|$, i.e. we spend more bits for rare characters of $T$ and less for frequent ones. Figure 2.6 (b) shows the Huffman shaped wavelet tree for $T^{\text{BWT}}$ of our running example. Even in this small example the number of bits for the Huffman shaped tree (36) is smaller than the that of the balanced one (40). In general the number of used bits is close to $nH_0$, and never more than $nH_0 + n$. In practice the size of the Huffman shaped wavelet tree is smaller, except for random text, than that of the balanced one which uses $n \log_\sigma$ bits. The use of Huffman code also optimizes the average query time for $\text{rank}$, $\text{select}$, and the $[i]$-operator, which returns the symbol at position $i$ of the original sequence. Therefore in practice the Huffman code wavelet tree outperforms the balanced one in both query time and memory. We will see in the next section, that we can compress the wavelet tree further,

---

\(^1\) Note that in general the Huffman code does not preserve the lexicographical ordering of the original alphabet. However in some application of the wavelet tree, for instance bidirectional search [SOG10], the ordering has to be preserved. In this case Hu-Tucker code, which preserves the ordering, is one alternative to Huffman code.

\(^2\) We have also implemented a version for large alphabet sizes which eliminates the $3\sigma \log n$ part of the complexity by replacing pointers by rank queries.
2.6.2 Run-Length Wavelet Tree

The Huffman shaped wavelet tree uses $nH_0$ bits to represent the bit vectors $b_v$. In practice this simple method is enough to get a notable compression of the original text while the operations slow down only by a small factor. However, for texts with many repetitions like xml (where tags are repeated many times) or version control data (where many files are very similar) the compression is far from optimal.

Now, the remaining option is not only to compress single characters but whole runs...
2.6 The Wavelet Tree

Algorithm 2 The \textit{select}(i, c) method for the basic wavelet tree.

\begin{algorithm}
\begin{algorithmic}[0]
\State $v \leftarrow c_{\text{to_leaf}}[c]$ \Comment{\textit{c}_{\text{to_leaf}} is a mapping from characters to leaves.}
\State $j \leftarrow \text{depth}(v)$
\While{$j > 0$}
\State $j \leftarrow j - 1$
\State $v \leftarrow \text{parent}(v)$
\State $i \leftarrow \text{select}_{b_v}(i, \text{code}(c)[j]) + 1$
\EndWhile
\State \Return $i - 1$
\end{algorithmic}
\end{algorithm}

Algorithm 3 The $[i]$-operator for the basic wavelet tree.

\begin{algorithm}
\begin{algorithmic}[0]
\State $v \leftarrow \text{root}(\cdot)$
\While{not \text{is_leaf}(v)}
\If{$b_v[i] = 0$}
\State $i \leftarrow i - \text{rank}_{b_v}(i)$
\State $v \leftarrow v_t$
\Else
\State $i \leftarrow \text{rank}_{b_v}(i)$
\State $v \leftarrow v_r$
\EndIf
\EndWhile
\State \Return leaf_{to_c}[v]
\end{algorithmic}
\end{algorithm}

of characters. This approach is similar to compressors like \texttt{bzip2}, which first apply move-to-front encoding to $T^\text{BWT}$ and then Huffman coding on the result.

At first, suppose that we only want to support the $[i]$-operator. Then the almost obvious solution is as follows: We take a bit vector $b_l (= b_{\text{last}})$ of size $n$. Each element $b_l[i]$ indicates, if index $i$ is the first element of a run of equal characters. See Figure 2.7 for an example. To answer the $[i]$-operator it suffices to store only those character $T^\text{BWT}[i]$ which are the first in the run, i.e. all $T^\text{BWT}[i]$ with $b_l[i] = 1$. Lets call this new string $T^\text{BWT}'$. $T^\text{BWT}'$ consists of exactly $R = \text{rank}_{bl}(n)$ elements. $T^\text{BWT}[i]$ can be recovered by first calculating the number of character runs $i' = \text{rank}_{bl}(i + 1)$ up to position $i$ and then returning the corresponding character $T^\text{BWT}[i' - 1]$ of the run. If we store $T^\text{BWT}'$ uncompressed the whole operations take constant time. But in practice it is better to store $T^\text{BWT}'$ in a Huffman shaped wavelet tree $wt'$. This turns the average access time to $H_0$ but reduces space and adds support for rank and select on $T^\text{BWT}'$.

The problem with this representation is that it does not contain enough information to answer rank and select queries for $T^\text{BWT}$. Take e.g. rank$(13,m)$. A rank query rank$_{bl}(13 + 1)$ on $bl$ tells us that there are $r_u = 7$ runs up to position 13. Next we can determine that there are $c_{\text{runs}} = 2$ runs of $m$ up to position 13, since rank$_{wt'}(7,m) = 2$. But we cannot answer how many $m$ are in the two runs. Mákinen and Navarro [MN05]
showed how to handle this problem. They introduce a second bit vector $bf$ (= $b$ first) which contains the information of runs in another order. Figure 2.7 depicts $bf$ for our running example. It can be expressed with the LF-function: $bf[LF[i]] = bl[i]$ for all $0 \leq i < n$. I.e. all runs of a specific character $c$ are in a row inside the $c$-interval. E.g. take the three runs of $m$ of size $1$, $3$, and $1$ in $TBWT$. Now they are in a row inside the $m$-interval [4,8]. Finally, by adding a rank and select structure for $bf$ we can answer the rank $(13, m)$ query of our example. We have already counted two $m$-runs, we also know that we are not inside of an $m$-run, as $wt'[7 - 1] = d \neq m$. The number of $m$ in the first two runs in $TBWT$ can now be determined by selecting the position of the first 1 after the second 1 in the $m$-interval and subtracting the starting position of the $m$-interval.

Algorithm 4 shows pseudo-code. Things are slightly more complicated if the position $i$ is inside a run of cs. In this case we have to consider a correction term which requires also a select data structure on $bl$.

Since Mäkinen and Navarro proposed their data structure as Run-Length Encoded FM-Index (RLFM) they have not mentioned that select queries can also be supported, if a select structure for $wt'$ is present. We will complete the picture here and provide pseudo-code for select in Algorithm 5. RLFM occupies $2n$ bits for $bf$ and $bl$, $RH_0 + o(R)$ bits for $wt'$. Since $R$ is upper bounded by $n \cdot H_k + \sigma^k$ (see Equation 2.4 on page 12) we get a worst case space complexity of $n(H_k H_0 + 2) + O(\sigma^k) + o(n)$ bits. Sirén et. al [SVMN08] suggested to use a compressed representation [GHSV06] for $bl$ and $bf$ called binary searchable dictionary (BSD), and called the resulting structure RLFM+.

There exists another solution for compressing the wavelet tree to $nH_k + O(\sigma^k) + o(n \log \sigma)$
Algorithm 4 The rank\((i, c)\) method for the run-length wavelet tree.

01 \( \text{runs} \leftarrow \text{rank}_{bl}(i) \)
02 \( c_{\text{runs}} \leftarrow \text{rank}_{wt}(\text{runs}, c) \)
03 if \( c_{\text{runs}} = 0 \) then
04 return 0
05 if \( wt'[\text{runs} - 1] = c \) then
06 return \( \text{select}_{bf}(c_{\text{runs}} + \text{rank}_{bf}(C[c])) - C[c] + i - \text{select}_{bl}(\text{runs}) \)
07 else
08 return \( \text{select}_{bf}(c_{\text{runs}} + 1 + \text{rank}_{bf}(C[c])) - C[c] \)

Algorithm 5 The select\((i, c)\) method for the run-length wavelet tree.

01 \( c_{\text{runs}} \leftarrow \text{rank}_{bf}(C[c] + i) - \text{rank}_{bf}(C[c]) \)
02 \( \text{offset} \leftarrow C[c] + i - 1 - \text{select}_{bf}(c_{\text{runs}} + \text{rank}_{bf}(C[c])) \)
03 return \( \text{select}_{bl}(\text{select}_{wt}(c_{\text{runs}}) + 1) + \text{offset} \)

bits: The alphabet-friendly FM-Index (AFFM) of Ferragina et al. [FMMN04, FMMN07].

The main idea is to partition \( T^{BWT} \) in suitable chunks, i.e. intervals of contexts of not fixed length, and then compress each chunk with a Huffman shaped wavelet tree to zero order entropy. The task to partition \( T^{BWT} \) optimally, i.e. minimizing the size of the wavelet trees, can be solved by a cubic dynamic programming approach. Ferragina et al. [FNV09] presented a solution which is at most a factor\((1+\varepsilon)\) worse than the optimal partitioning while the running time is \( O(n \log_{1+\varepsilon} n) \) for a positive \( \varepsilon \).

The latest proposal for a run-length wavelet tree comes from Sirén et al. [SVMN08]. It is named RLWT. Here the idea is not to run-length compress the sequence itself but the concatenation of the bit vectors of the resulting wavelet tree. They also use the BSD data structure to get a space complexity which depends on \( H_k \).

We will propose a new run-length wavelet tree in Section 3.6.1.

2.7 The Backward Search Concept

We have learned in the last section how we can perform constant time rank and select queries for constant size alphabets\(^1\) by using only a wavelet tree. In this section we will show how Ferragina and Manzini [FM00] used this fact to solve the decision and counting version of the substring problem in \( O(|P|) \) when \( T^{BWT} \) is already present. Otherwise we have to do a linear time preprocessing to calculate \( T^{BWT} \). We will illustrate the

\(^1\) Constant time rank and select queries can even be done for alphabets of size in \( O(\text{polylog}(n)) \), cf. [FMMN07]
procedure called *backward search* by an example. Like in the example for forward search (see page 12) we take the pattern $P = \text{umu}$ and string $T = \text{umulmundumulmum}$. We also keep track of an SA interval. But this time the SA interval contains all suffixes which start with the current suffix of $P$ in contrast to forward search, which matches the current prefix of $P$! Initially, we have the empty suffix $\varepsilon$ of $P$ and every suffix of $T$ has $\varepsilon$ as prefix. Therefore the initial SA interval is $[i..j] = [0..n-1] = [0..15]$. In the first step we extend the suffix by one character to the left, i.e. to $u$. Therefore the new SA interval is included in the $u$ interval. In the first step it is clear that it even equals the $u$ interval, i.e. $[i..j] = [\text{C}[u]..\text{C}[u+1] - 1] = [10..15]$. When we extend the suffix one character further we get $\mu u$. This time we have to count how many of the $j+1$ suffixes in the range $[0..15]$, which are lexicographically equal to or smaller than the currently matched suffixes, are preceded by an $m$, and how many of the $i = 10$ lexicographically smaller suffixes in the range $[0..9]$ are preceded by a $m$. In Figure 2.8 (a) we get $s = 1$ suffix in $[0..9]$ and $e = 5$ suffixes in $[10..15]$. Therefore our new SA interval is $[i..j] = [\text{C}[m] + s..\text{C}[m] + e - 1] = [5..8]$. In the last step we extend the suffix to $\text{umu}$. Since there are $s = 3$ $u$ in $\text{T BWT}[0..4]$ and $e = 5$ in $\text{T BWT}[0..8]$, the final SA interval is $[i..j] = [\text{C}[u] + s..\text{C}[u] + e - 1] = [13..14]$; see Figure 2.8 (c). So we have determined, that there are 2 occurrences of the pattern $P = \text{umu}$ in $T$. If a pattern does not occur in $T$, we get an interval $[i..j]$ with $j = i - 1$. E.g. suppose we want to find $P = \text{mumu}$. In Figure 2.8 we have already matched the suffix $\text{umu}$. Now, there are 4 $m$ in $[0..12]$ and $[0..14]$, and therefore the new interval would be $[\text{C}[m] + e..\text{C}[m] + s - 1] = [8..7]$. Note, that we only need the wavelet tree of $\text{T BWT}$ and $C$ to calculate the decision and counting version of the substring problem, since $s$ and $e$ can be computed by a rank query on the wavelet tree. If we add SA we can also answer the locating version by returning every SA value in the range $[i..j]$.

Algorithm 6 depicts the pseudo-code for the whole matching process and Algorithm 7 shows one backward search step. We will see in Chapter 6 that the backward search concept cannot only be used to solve the different versions of the substring problem but

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{T BWT}$</th>
<th>$F$</th>
<th>$\text{T BWT}$</th>
<th>$F$</th>
<th>$\text{T BWT}$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>1</td>
<td>m</td>
<td>$$</td>
<td>u</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>$$</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>$$</td>
<td>g</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>4</td>
<td>u</td>
<td>$$</td>
<td>d</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td>$$</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>6</td>
<td>l</td>
<td>$$</td>
<td>d</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>7</td>
<td>l</td>
<td>$$</td>
<td>u</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>8</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>9</td>
<td>l</td>
<td>$$</td>
<td>u</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>10</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>11</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>12</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>13</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>14</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
<tr>
<td>15</td>
<td>l</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
<td>m</td>
<td>$$</td>
</tr>
</tbody>
</table>

Figure 2.8: Three steps of backward search to find the pattern $P = \text{umu}$. 


more complex tasks in sequence analysis, like the computation of maximal exact matches of two strings, very time and space efficiently.
Algorithm 6 Pattern matching by backward search $\text{backward}\_\text{search}(P)$.

01 $[i..j] \leftarrow [0..n-1]$
02 for $i \leftarrow |P| - 1$ downto 0 do
03 $[i..j] \leftarrow \text{backward}\_\text{search}(P[i],[i..j])$
04 if $j < i$
05 return ⊥
06 return $[i..j]$

Algorithm 7 Backward search step $\text{backward}\_\text{search}(c,[i..j])$.

01 $i' \leftarrow C[c] + \text{rank}_{BWT}(i,c)$
02 $j' \leftarrow C[c] + \text{rank}_{BWT}(j+1,c) - 1$
03 return $[i'..j']$

2.8 The Longest Common Prefix Array

The suffix array $SA$, the inverse suffix array $ISA$, $\Psi$ and LF contain only information about the lexicographic ordering of the suffixes of the string $T$. However, many tasks in sequence analysis or data compression need information about common substrings in $T$. The calculation of maximal repeats in $T$ or the Lempel-Ziv factorization are two prominent instances.

Information about common substrings in $T$ is stored in the longest common prefix (LCP) array. It stores for two lexicographically adjacent suffixes the length of the longest common prefix of the two suffixes. In terms of $SA$ we can express this by the following equation:

$$\text{LCP}[i] = \begin{cases} 
-1 & \text{for } i \in \{0,n\} \\
\max\{k \mid T[SA[i-1]..SA[i-1]+k] = T[SA[i]..SA[i]+k]\} & \text{otherwise}
\end{cases}$$

For $i \in \{0,n\}$

(2.11)

In the left part of Figure 2.9 we can see the LCP array for our running example. Note that the entry at position 0 is defined to be $-1$ only for technical reasons, i.e. to avoid case distinctions in the pseudo-code. Later, in the real implementation we will define $\text{LCP}[0] = 0$. In uncompressed form the LCP array takes $n \log n$ bits like the uncompressed suffix array. The naive calculation of the LCP array takes quadratic time by comparing each of the $n - 1$ pairs of adjacent suffixes from left to right, character by character. Kasai et al. [KLA+01] presented a linear time solution. We will explain the construction process in detail in Chapter 5.

It is easy to see that we can calculate the length of the longest common prefix $\text{lcp}(x,y)$ of two arbitrary suffixes $x$ and $y$ by taking the minimal LCP value of all lexicographic...
neighbouring suffixes between $x$ and $y$. For instance the longest common prefix between suffixes 9 and 4 in our running example (see Figure 2.9) equals 2, as the minimum LCP-value between suffixes 9 and 1, 1 and 12, and 12 and 4 equals 2. Clearly, the calculation of the longest common prefix between arbitrary suffixes in that way takes linear time. With an additional data structure we can reduce the time complexity to constant time.

A range minimum query $\text{RMQ}_A(i, j)$ on the interval $[i..j]$ of an array $A$ returns the index $k$ of the element with the smallest value in $A[i..j]$, i.e. $\text{RMQ}_A(i, j) = \min\{k \mid i \leq k \leq j \land A[k] \leq A[r] \forall r \in [i..j]\}$. We omit the subscript if $A$ is clear from the context.

We omit the subscript if $A$ is clear from the context.

Now we can express the length $|\text{lcp}(SA[i], SA[j])|$ of the longest common prefix of two arbitrary suffixes $SA[i]$ and $SA[j]$ with $i < j$ as follows:

$$|\text{lcp}(SA[i], SA[j])| = \text{LCP}[\text{RMQ}_{\text{LCP}}(i + 1, j)]$$ (2.12)

We will see that the following two arrays are also very useful. The next smaller value ($\text{NSV}_A$) and previous smaller value ($\text{PSV}_A$) array for an array $A$ of length $n$ are defined as follows:

$$\text{NSV}_A[i] = \min(\{n\} \cup \{j \mid i < j \leq n - 1 \land A[j] < A[i]\})$$ (2.13)

$$\text{PSV}_A[i] = \max(\{-1\} \cup \{j \mid 0 \leq j < i \land A[j] < A[i]\})$$ (2.14)

We omit the subscript $A$, if $A$ is clear from the context. Both arrays $\text{NSV}$ and $\text{PSV}$ can be computed from $A$ in linear time by scanning $A$ from right to left (left to right) and using a stack.

Now, we are ready to present the connection of the LCP array with the topology of

$$\begin{array}{cccc}
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
SA & 15 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & -1 & \varepsilon \\
LCPT & -1 & 2 & 1 & 5 & 3 & 6 & 4 & 1 & 3 & 2 & 1 & 0 & -1 & \varepsilon \\
BWT & \$ & \text{dumum}$ & \text{lum}$ & \text{umum}$ & \text{ul}$ & \text{lum}$ & \text{um}$ & \text{ul}$ & \text{um}$ & \text{um}$ & \text{um}$ & \text{um}$ & \text{um}$ & \text{um}$ & \text{um}$ & \text{um}$ \\
\end{array}$$

Figure 2.9: Left: The suffix array of the text $T=\text{umumulmumumum}$ and the corresponding longest common prefix (LCP) array. Right: The corresponding LCP-interval tree.
the suffix tree.

According to [AKO04], an interval \([i..j]\), where \(1 \leq i \leq n\), in an LCP-array is called an LCP-interval of LCP-value \(\ell\) (denoted by \(\ell-[i..j]\)) if:

1. \(\text{LCP}[i] < \ell\),
2. \(\text{LCP}[k] \geq \ell\) for all \(k \in [i+1..j]\),
3. \(\text{LCP}[k] = \ell\) for at least one \(k \in [i+1..j]\),
4. \(\text{LCP}[j+1] < \ell\).

Every index \(k\), \(i+1 \leq k \leq j\), with \(\text{LCP}[k] = \ell\) is called an \(\ell\)-index. Note that each LCP-interval has at most \(\sigma-1\) many \(\ell\)-indices.

An LCP-interval \(m-[p..q]\) is said to be embedded in an LCP-interval \(\ell-[i..j]\) if it is a subinterval of \([i..j]\) (i.e., \(i \leq p < q \leq j\)) and \(m \geq \ell\). The interval \([i..j]\) is then called the interval enclosing \([p..q]\). If \([i..j]\) encloses \([p..q]\) and there is no interval embedded in \([i..j]\) that also encloses \([p..q]\), then \([p..q]\) is called a child interval of \([i..j]\). The parent-child relationship constitutes a tree which we call the LCP-interval tree (without singleton intervals). An interval \([k..k]\) is called a singleton interval. The parent interval of such a singleton interval is the smallest LCP-interval \([i..j]\) which contains \(k\). Note that if we add the singleton intervals, the tree topology of the LCP-interval tree and the suffix tree coincide.

The child intervals of an LCP-interval can be determined as follows. Let \(i_1,i_2,\ldots,i_k\) be the \(\ell\)-indices of an LCP-interval \(\ell-[i..j]\) in ascending order, then the child intervals of \([i..j]\) are \([i..i_1-1]\), \([i_1..i_2-1]\), \ldots, \([i_k..j]\) (note that some of them may be singleton intervals); see [AKO04]. With RMQs on the LCP-array, \(\ell\)-indices can be computed easily [FH07]: \(k = \text{RMQ}(i+1,j)\) equals the smallest \(\ell\)-index \(i_1\), \(\text{RMQ}(i_1+1,j)\) yields the second \(\ell\)-index, etc. However, we will show in Section 3.9.3 how we can calculate \(\ell\)-indices without RMQs in constant time using only \(2n + o(n)\) bits. Note that also the calculation of the next sibling of a node can be done by finding the next \(\ell\)-index.

Also the parent interval of an LCP-interval can be determined in constant time by the NSV and PSV operations; cf. [FMN09]. It holds:

\[
\text{parent}([i..j]) = \begin{cases} 
\text{PSV}[i].\text{NSV}[i] - 1 & \text{if } \text{LCP}[i] \geq \text{LCP}[j+1] \\
\text{PSV}[j+1].\text{NSV}[j+1] - 1 & \text{if } \text{LCP}[i] < \text{LCP}[j+1]
\end{cases}
\quad (2.15)
\]

We will see in Section 3.9.3 how we can solve NSV, PSV, and RMQ queries in constant time using only \(3n + o(n)\) bits.

At last, we present the central observation from [KLA+01] which can be used to calculate the LCP-array in linear time and to compress the LCP-array to \(2n\) bits [Sad02]:

\[
\text{LCP}[i] \geq \text{LCP}[\text{ISA}][\text{SA}[i] - 1 \mod n] - 1
\quad (2.16)
\]

That is, the length of the longest common prefix of suffix \(x = \text{SA}[i]\) and its preceding suffix \(y = \text{SA}[i-1]\) in the suffix array (=\(\text{LCP}[x]\)) is as least as long as the longest common
prefix $\omega$ of the suffix $x - 1$ (at position $ISA[x - 1]$ in the suffix array) and its preceding suffix $z$ in the suffix array (= $LCP[x - 1]$) minus one. If $LCP[x - 1] \leq 0$ then the equation is true, since all $LCP$-values are greater than or equal to 0. If $LCP[x - 1] > 0$ then suffix $(x - 1) + 1 = x$ and $z + 1$ have at least a prefix of $LCP[x - 1] - 1$ characters in common, as we only omit the first character of $\omega$.

Now suppose that suffix $x = SA[i]$ and suffix $y = SA[i - 1]$ are preceded by the same character, i.e. $T^{BWT}[i] = T^{BWT}[i - 1]$ or $T[SA[LF[i]]] = T[SA[LF[i - 1]]]$ for $i \geq 1$. Then the longest common prefix of suffix $x - 1 = SA[LF[i]]$ and suffix $y - 1 = SA[LF[i - 1]]$ is one character longer than the longest common prefix of $x$ and $y$:

$$LCP[i] = LCP[SA[i] - 1 \mod n] - 1 \text{ iff } T^{BWT}[i] = T^{BWT}[i - 1]$$

(2.17)

Such an $LCP$-entry $i = SA[j]$ is called reducible in [KMP09] and if Equation 2.17 does not hold it is called irreducible. Note that the condition $T^{BWT}[i] = T^{BWT}[i - 1]$ is equivalent with $LF[i] = LF[i - 1] + 1$ (cf. Equation 2.9).

A special permutation of the $LCP$ array is called the permuted longest common prefix array ($PLCP$): It is defined as follows [KMP09]:

$$PLCP[SA[i]] = LCP[i] \text{ or equivalently } LCP[ISA[j]] = PLCP[j]$$

(2.18)

While the $LCP$ values are in order of the suffix array, which is also called $SA$-order, the $LCP$-values in $PLCP$ are ordered in text-order.

We get the analogon of Inequality 2.16 for $PLCP$ by first setting $i = ISA[j]$ in Inequality 2.16. So $LCP[ISA[j]] \geq LCP[ISA[j - 1 \mod n]] - 1$ and by using Equation 2.18 we get

$$PLCP[i] \geq PLCP[i - 1 \mod n] - 1$$

(2.19)

The definition of reducible $PLCP$ values is also analogous to $LCP$.

$$PLCP[i] = PLCP[i - 1 \mod n] - 1 \text{ if } T[i - 1] = T[\Phi[i] - 1]$$

(2.20)

where $\Phi[i] = SA[ISA[i] - 1]$. If Equation 2.20 holds for a entry $PLCP[i]$, we call $PLCP[i]$ reducible. Note, that sometimes it is also convenient to store all values of $\Phi$ in an array, which we call $\Phi$ array.

Clearly, $PLCP$ ($LCP$) can be constructed from $LCP$ ($PLCP$) in linear time when $SA$ is available. However, as $SA$ is not a very regular permutation, this process has a very bad locality of references and is therefore expensive in todays computer architectures. $PLCP$ can be computed faster than $LCP$ directly from $T$ and $SA$ [KMP09] and its use is favorable in some applications, see [OG11] for a very recent example.
3 Design

3.1 The Big Picture

We have already given the classical definition of a suffix tree in the previous chapter. Now we define the abstract data type of a CST by specifying a set of operations which should be provided and executed in an efficient manner by a CST. An abstract data type corresponds in our implementation in C++ to a C++ concept. In the following we will use the term of a “(C++) concept” instead of using the term “abstract data type”. Table 3.1 lists the 24 operations for the CST concept where symbol $i$ represents an integer, $c$ a character, and $v$ a node of the CST. Note that the concrete representation of the node is not yet specified. The listed operations are a superset of the set which are required for a full-functional (compressed) suffix tree due to the definition of Sadakane [Sad07a]. His full-functionality definition requires: root(), is_leaf($v$), child($v$, $c$), sibling($v$), parent($v$), edge($v$, $d$), depth($v$), lca($v$, $w$), and sl($v$). Some of the operations that we have added to the set are very handy when using the data structure in practice, albeit they were straightforward to implement and they were already present in uncompressed suffix trees; for instance size() or nodes().

Nevertheless, one should notice that most of the other added operations actually extend the functionality of uncompressed suffix trees, which is amazing since we would expect that data structures which use more space provide more functionality! All of these operations, which extend the functionality of an uncompressed suffix tree are based on the general design of a CST, which portions the information of a CST into three parts: The lexicographical information in a compressed suffix array (CSA), the information about common substrings in the longest common prefix array (LCP), and the tree topology in combination with a navigation structure (NAV). Figure 3.1 depicts the different parts in different colors which will be used consistently throughout the whole thesis. It should be clear that the size $|\text{CST}|$ of a compressed suffix trees depends on the size of the three parts:

$$|\text{CST}| = |\text{CSA}| + |\text{LCP}| + |\text{NAV}|$$

(3.1)

In the last years there were many proposals for each component of a CST. Like for the CST itself we will define a set of operations which should be implemented for each component, so that every combination of a CST, LCP, and NAV results in a fully-functional CST. And for this reason we can generate a myriad of different CSTs which offer a wide range of space-time trade-offs.

Sometimes the components depend on each other, e.g. an LCP structure of Sadakane [Sad02] depends on the CSA; i.e. an operation of the LCP structure requires a result
### 3.1 The Big Picture

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>Returns the number of leaves in the CST.</td>
</tr>
<tr>
<td>nodes()</td>
<td>Returns the number of nodes in the CST; i.e. the number of leaves plus the number of inner nodes.</td>
</tr>
<tr>
<td>root()</td>
<td>Returns the root node of the CST.</td>
</tr>
<tr>
<td>is_leaf(v)</td>
<td>Returns true if the node v is a leaf; i.e. we can decide if v is an inner node or not which is useful in many other operations like depth(v).</td>
</tr>
<tr>
<td>id(v)</td>
<td>Returns for each node v a unique number in the range [0..nodes() − 1]. This is useful in many suffix tree algorithms which have to store additional information at the nodes of the suffix tree.</td>
</tr>
<tr>
<td>lb(v)</td>
<td>Returns the left bound of the node v in the suffix array.</td>
</tr>
<tr>
<td>rb(v)</td>
<td>Returns the right bound of the node v in the suffix array.</td>
</tr>
<tr>
<td>depth(v)</td>
<td>Returns the length of L(v).</td>
</tr>
<tr>
<td>node_depth(v)</td>
<td>Returns the number of edges on the path from the root to node v.</td>
</tr>
<tr>
<td>degree(v)</td>
<td>Returns the number of child nodes of node v; e.g. for v = root() we get σ.</td>
</tr>
<tr>
<td>parent(v)</td>
<td>Returns the parent node of node v. If v = root(), v is returned.</td>
</tr>
<tr>
<td>sibling(v)</td>
<td>Returns the next sibling of v, if v is not the rightmost child of parent(v). Otherwise root() is returned.</td>
</tr>
<tr>
<td>sl(v)</td>
<td>Returns the suffix link of node v. If v = root(), v is returned. Note that the suffix link operation is also defined for leaf nodes in the CST, whereas it is not defined for leaf nodes in the classical suffix tree.</td>
</tr>
<tr>
<td>edge(v, d)</td>
<td>Returns the d-th character of L(v) with d ∈ [1..depth(v)].</td>
</tr>
<tr>
<td>child(v, c)</td>
<td>If there exists a child w of v with edge(w, depth(v) + 1) = c, w is returned. Otherwise, root() is returned.</td>
</tr>
<tr>
<td>ith_child(v, i)</td>
<td>Returns the i-th child node of v for i ∈ [1..degree(v)].</td>
</tr>
<tr>
<td>lca(v, w)</td>
<td>Returns the lowest common ancestor of nodes v and w in the CST.</td>
</tr>
<tr>
<td>node(lb, rb)</td>
<td>Returns the node in the CST which corresponds to the LCP-interval [lb,rb].</td>
</tr>
<tr>
<td>wl(v, c)</td>
<td>Returns the Weiner link of node v and character c or root() if the Weiner link does not exists.</td>
</tr>
<tr>
<td>ith_leaf(i)</td>
<td>Returns the i-th leaf in the CST with i ∈ [1..size()].</td>
</tr>
<tr>
<td>tlcp_idx(i)</td>
<td>Returns the postorder index of node v = lca(i, ith_leaf(i), ith_leaf(i+1)) for i ∈ [1..size() − 1] and the postorder index of root() for i = 0. For details see Section 3.10.1.</td>
</tr>
</tbody>
</table>

**Table 3.1:** Our **cst** concept requires the listed operations. Furthermore the operations leaves_in_the_subtree(v), leftmost_leaf_in_the_subtree(v), and rightmost_leaf_in_the_subtree(v) are required. These additional operations calculate the number of leaves, the leftmost, and the rightmost leaf in the subtree rooted at v.
Figure 3.1: General design of a CST: CSA information is colored in red, LCP information is colored in orange, and NAV is colored in blue.

of an operation of the CSA. Therefore, the time complexity for the operation on LCP depends on the CSA operation. This also implies that this LCP representation does not work standalone, and we have to add the size of the CSA to it, when we measure its space. But in the context of the CST the CSA is present anyway, and we can use it for answering LCP queries virtually for free. Therefore, it is better to substitute Equation 3.1 by the inequality:

\[ |\text{CST}| \leq |\text{CSA}| + |\text{LCP}| + |\text{NAV}| \]  

(3.2)

where \(|\cdot|\) this time denotes the size, if we consider the data structures as standalone. So we can say sometimes “the whole is less than the sum of its parts”.\(^1\) In the context of data compression we appreciate this fact.

We have added one operation to the CST concept which enables us to exploit the shared usage of information across the components: \(\text{tlcp, idx}(i)\). We will postpone details of this function to Section 3.10.

The abstraction of a concrete node representation forces the introduction of the operations \(\text{lb}(v)\), \(\text{rb}(v)\), and \(\text{node}(\text{lb}, \text{rb})\). \(\text{lb}(v)\) and \(\text{rb}(v)\) return the left and right bound of the SA interval which corresponds to node \(v\); \(\text{node}(\text{lb}, \text{rb})\) takes the SA interval bounds and returns the corresponding node. We will see in Chapter 6 that this transformation from the node type to a SA interval and back is extremely useful in many algorithms which use the backward search concept. Backward search is also naturally supported by the Weiner link operation \(\text{wl}(v, c)\) in our CST definition. Let us briefly explain the

\(^1\) in contrast to the famous sentence of Aristotle
3.2 CST Overview

Weiner link operation here: Let $s = L(v)$ be the path label from the root node to node $v$ in the tree, then the Weiner link $wl(v, c)$ equals the node $w$ which has the concatenation of $c$ and $s$ ($c \oplus s$) as a prefix of his path label. In the case that such a $w$ does not exists the root node is returned. For instance in Figure 3.1 the Weiner link of the node with path label $um$ and $c = m$ equals the leaf with label 12 and path label $mum\$.

We will see that we will get the Weiner link operation virtually for free from the CSA structure by using the LF function.

The suffix link operation $sl(v)$ goes into the other direction: The suffix link $w = sl(v)$ of a node $v$ with $L(v) = c \oplus \omega$ is a node $w$ with $L(w) = \omega$, where $\omega$ is a (possibly empty) substring of $T$.

Finally we provide an operation which simplifies the usage of a CST in applications. Many of the classical suffix tree algorithms store during their execution extra information at the nodes. However it is a very bad idea to extend the node data types each time for a specific problem. Therefore, we introduce an operation $id(v)$ which gives us for every node $v$ a unique integer in the range from 0 to the number of nodes minus one. Now we can allocate arrays of the specific data types for the extra information and read and write the extra information for node $v$ at index $id(v)$.

3.2 CST Overview

We have already mentioned that we can create different types of CST by using different combinations of CSA, LCP, and NAV data types. We will here give an overview which of these combinations were already presented as CSTs. They have all in common that they use an entropy compressed CSA, which we will present in detail in Section 3.7. An entropy compressed CSA uses about $nH_k + \sigma_k$ bits of space. In addition to that

- Sadakane [Sad07a] uses an LCP structure which takes $2n + o(n)$ bits and a NAV structure which represents the tree topology explicitly by using a balanced parentheses sequence which takes $2n$ bits for each nodes; i.e. in the worst case of $2n$ nodes, NAV takes in combination with a sub-linear support structure $4n + o(n)$ bits. So in total we have $6n + o(n)$ bits on top of the CSA.

- Fischer et al. [FMN09] use a run-length encoded version of Sadakane’s LCP encoding and showed that the size is upper bounded by $H_k(2n \log \frac{1}{H_k} + O(1)) + O(\frac{n \log \log n}{\log n})$ bits, which is in theory better than the $2n + o(n)$ bits of Sadakane. However experiments of Cánovas [Cán10] showed that in practice the difference is negligible. The NAV structure depends heavily on NSV, PSV, and RMQ operations on LCP. That is they do not store the tree topology explicitly but use for each of the three operations a sub-linear data structure, which answers the queries in sublogarithmic but not constant time.

- Russo et al. [RNO08] sample only a small subset of size $O(n/\delta)$ of nodes of the suffix tree, and use the CSA to recover the other nodes. Note that the tree which is formed by the remaining sampled nodes can be represent by any of the NAV structures in combination with a LCP proposal. In addition to NAV and LCP they
use $O(n/\delta \log n)$ bits to indicate the sampled nodes. So in total they get for a choice of $\delta = (\log \log n) \log n$ a space complexity of $o(n \log \sigma)$ bits and logarithmic time complexity for most of the operations.

- In our first proposal [OG09] we use any of the existing LCP solutions and our NAV depends like in the solution of Fischer et al. on NSV, PSV, and RMQ operations on LCP. We show, that using $2n + o(n) + |LCP|$ in total suffice to answer all operations in at most $\log \sigma$ time.

- In our second proposal [OFG10] we showed that $3n + o(n)$ bits suffice to represent NAV so that all operations can be answered in constant time and NAV does not depend on LCP! In addition several LCP proposals can be further compressed by using the additional information in NAV.

All of these proposals, except for the proposal of Russo et al. [RNO08], fit perfectly in our framework. We have put much effort into a very clean implementation of the CSTs. As a result, theory and practice of CSTs coincide in virtually all cases. Table 3.2 depicts the runtime dependencies of CST operations from the runtime of CST components. We have chosen three CST implementations: cst_sct3 corresponds to our last proposal from 2010, cst_sada to our optimized reimplementation of Sadakane’s proposal [Sad07a], and cstY to the implementation of the proposal of Fischer et al. [FMN09] by Cánovas [Cán10]. The dependencies of our first proposal are equal with those of cstY. Russo et al.’s CST can be realized with all of the listed CSTs. But since the unsampled nodes have to be recovered by CSA operations, the first column would be marked in all cases where the operation depends on a node $v$.

Note that Table 3.2 gives just a very rough impression. However, it should be clear that operations like child($v, c$) which depend on all three components are more expensive than operations which are only dependent on two components or just one component. Some operations like size() or root() do not have any dependency at all. This means that they are built-in operations; e.g. we store the answer for the root() operation in the specific CST. We can observe, that cst_sct3 minimizes the dependencies on LCP and sometimes it also avoids dependencies on NAV compared to cst_sada. In contrast cstY minimizes dependencies on NAV while new LCP dependencies are established.

In the next section we will define concepts for the components of the CST. But before doing this, we will first provide an overview of all the concepts and an example of how a CST implementation is built up from implementations of these concepts. The names of the nine concepts can be found in Figure 3.2. Each concept name corresponds to the header of a box. For instance the CST concept can be found as box header cst in the box at the bottom. The class names of concrete implementations, i.e. classes in our C++ library, which fulfill the concept are listed in the box of the concept. Figure 3.2 shows one possibility of how cst_sct3 can be built up from other classes. We have marked each class which is used to built up cst_sct3 with an ellipse. The class csa_wt which fulfills the concept csa was chosen as CSA type and the class lcp_support_tree2 which fulfills the concept lcp as LCP type. The NAV component of the CST consists of two parts.
Figure 3.2: Overview: Concepts (header of the boxes) and class names (content of the boxes) of implementations which fulfill the requirements of the specific concept.
<table>
<thead>
<tr>
<th>Operation</th>
<th>cst_sct3</th>
<th>cst_sada</th>
<th>cstY</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nodes()</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>root()</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>is_leaf(v)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>id(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lb(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rb(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>depth(v)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>node_depth(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>degree(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parent(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sibling(v)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>s(v)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>edge(v, d)</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>child(v, c)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ith_child(v, i)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lca(v, w)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>node(lb, rb)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>wl(v, c)</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>ith_leaf(i)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>tlcp_idx(i)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.2: Runtime dependency of the CST concept methods from the tree parts CSA, LCP, and NAV. A checkmark indicates the dependency, a minus independency, and a cross that the operation is not implemented.

The first is a balanced parentheses sequence, which is stored in a bit_vector.\(^1\) The second part is the class bp_support_sada which provides operations like find_close(i) and enclose(i) for the balanced parentheses sequence and therefore fulfills the bp_support\(^2\) concept. As one can see, the components of the CST itself are also built up from other components.

The class csa_wt for the CSA has one member whose type fulfills the concept of a wavelet tree called wavelet_tree. And the selected wavelet tree of type wt_huff again has members whose types fulfill the concept for a rank (rank_support) and a select data structure (select_support). Member types which fulfill these concepts are also required by our selected bp_support class. Since all of the selected classes have to store information, they all have a member of type int_vector.

\(^1\) bit_vector is a specialized version of the int_vector class.
\(^2\) bp_support is a abbreviation for balanced parentheses support.
In the remaining sections of the chapter we will present all of the depicted concepts. For each concept we will present the implemented classes along with experimental comparisons. We will start with the basic concepts `vector`, `rank_support`, `select_support`, and `wavelet_tree`, and will then present the more complex concepts `csa`, `lcp`, `bp_support`, and `rmq` whose implementations are based on the implementations of the basic concepts. All implemented classes are part of a C++ template library which will be presented in the following section.

3.3 The Succinct Data Structure Library

One goal of this thesis was to implement the described data structures and algorithms very efficiently. We therefore decided to first implement the basic data structures for the concepts `vector`, `rank_support`, `select_support`, `wavelet_tree` and assemble them in a library called succinct data structure library (sdsl). The sdsl was implemented in C++, because C++ allows to implement efficient low-level operations (e.g. bit parallelism on 64bit words), to control memory management, and to use high-level algorithm and data structures from other efficient libraries like the standard template library (STL). The STL served as prototype for the new library for the following reasons:

- It can be used very intuitively. For instance element $i$ of a container is accessed via the $[i]$-operator, elements are copied with the $=$-operator and compared with the $==$-operator and so on.

- It is very efficient, because it does not use inheritance for modeling but templates and concepts.

- STL concepts (like container, iterators,...) are used for the succinct data structures whenever it was appropriate (e.g. for arrays and trees).

The biggest differences between the STL and the sdsl are the following.

- While most types in the STL are mutable, the big part of types in the sdsl are immutable. The reason for that is, that in most data types it does not make sense to update values after the construction (take the suffix arrays for instance).

- Every class $X$ in the sdsl implements a method `serialize` which writes a object into a stream. The static method `util::get_size_in_bytes` takes an object of a class $X$ and returns the size in bytes of the representation of $X$ in memory. This functionality can be use to get an overview how many memory is spend for different part of a complex data structure. See Figures 4.9 and 4.10 for example.

- The data structures in the sdsl can be divided into two sets. The non-stand-alone ones and the stand-alone ones. Let $X$ be the index data structure and $D$ be the data structure for which we have built $X$. The index data structure $X$ can answer a specific set of operations about $D$. Now, $X$ is called non-stand-alone, if it has to access the data of $D$ to answer the operations; otherwise $X$ is called stand-alone.
In the last case, $X$ effectively replaces $D$. Systematic data structures can be easily recognized in the $sdsl$ by the substring `_support` in the class name; e.g. the class `rank_support_v` answers rank queries for a `bit_vector` $b$ in constant time, but has to access $b$ for every query. An example for a stand-alone data structure is the wavelet tree `wavelet_tree` which does not need the original sequence to answer all queries.

- There exist space-efficient semi-external construction algorithms for many data structures.

- There exists a class `bit_magic` that contains many static methods which calculate functions on 64bit words; e.g. functions to determine the number of bits set in a word, the position of the leftmost set bit in a word, the number of Fibonacci encoded numbers in a word, and so on.

3.4 The vector Concept

The vector concept requires exactly two operations. The first one is the $[i]$-operator, which should return the element at position $i$, and the second one is `size()`, which should return the number of elements stored in the vector. The `int_vector` and `bit_vector` are implemented mutable; i.e. we can also use the $[i]$-operator to assign new values to the element at position $i$. The `enc_vector` is implemented immutable.

3.4.1 int_vector and bit_vector

One of the most fundamental data structures is the vector. C++ and all other mainstream languages provide an implementation of a vector. To implement succinct data structures in practice one often needs vectors of integers of a specific word width; e.g. exactly $\log n$ or $\log \log n$ bits. In practice this cannot be realized with the C++ vector of the `STL`, as this vector can only be parametrized with an integer data type like `uint64_t`, `uint32_t`, ..., `uint8_t` or a boolean data type.

Therefore, we introduce a vector class for unsigned integers called `int_vector` in the `sdsl`. This class has one template parameter $w$ which specifies the fixed word width of the integers. If the parameter $w$ equals 2, 8, 16, 32, or 64 the `int_vector` equals a C array; i.e. the read and write access take the same time. In case of $w = 1$, the `int_vector<1>` is also called `bit_vector` and the read and write access take the same time as the `vector<bool>` of the `STL`. For $w = 0$, the word width can be changed during running time. More precisely we specify the word width in the constructor or change it after the construction of the vector. In many programs a C integer array or C++ integer vector can be replaced easily by an `int_vector` since the syntax for reading or writing an entry is exactly the same as it is realized with the $[i]$-operator. The syntax of the constructor of the `int_vector` equals the constructor of the `STL` vector except for a third argument, which specifies the word width.

Under the hood, the `int_vector` is implemented with an array of 64bit integers and an additional integer that stores the length of the vector in bits. Therefore, in practice the
size of this data structure is never greater than $n \cdot w + 127$ bits. Transier and Sanders call such a data structure bit-compressed ([TS10]). We call an \texttt{int\_vector} bit-compressed, if $w$ is greater than one and not a power of two. While such a representation reduces the space in some cases by a considerable amount (e.g. in the case when all integers fit in two bits, a char array would take four times the size of an \texttt{int\_vector}), the speed of a read and write operation of an bit-compressed \texttt{int\_vector} is slower than that of a C array. Figure 3.3 depicts benchmark results for an C integer array (i.e. a specialized \texttt{int\_vector}: \texttt{int\_vector<32>>}), a \texttt{bit\_vector}, a not specialized integer vector (\texttt{int\_vector<>}) with $w = 32$ bit integers and one with $w = 27$. The results can be explained as follows. Writing of a bit-compressed \texttt{int\_vector} forces one or two reads and writes on the underlaying 64bit integer array. This slows down the writing process. In contrast, the entries of an uncompressed \texttt{int\_vector} can be addressed directly, except for the \texttt{bit\_vector}.

In conclusion:

- The time of a random access read of one integer in a bit-compressed \texttt{int\_vector} is about 1.3-1.5 times the time of a random access read in a C integer array.

- The time of a random access write of one integer in a bit-compressed \texttt{int\_vector} is about two times the time of a random access write in a C integer array (the factor is 4 in the sequential case).

- Since the write access is really slow compared to the uncompressed array, one should avoid the use of the bit-compressed \texttt{int\_vector} in the construction of succinct data structures.

- Nevertheless the bit-compressed \texttt{int\_vector} is recommendable for the use in succinct data structures, since (1) the read access is not much slower than the read access to an C integer array and (2) we will see that the time is negligible compared to the time of the operations of other basic data structures like \texttt{rank\_support} or \texttt{select\_select} (see Section 3.5).

### 3.4.2 \texttt{enc\_vector}

The \texttt{enc\_vector} class is used to store $n$ integers in compressed form. Each integer is represented by its self-delimiting code. The self-delimiting code, like Elias-$\delta$, Elias-$\gamma$ [Eli75], or Fibonacci [Zec72], can be specified by a coder class (\texttt{coder::elias\_delta}, \texttt{coder::elias\_gamma}, \texttt{coder::fibonacci}) which is also provided in the sdsl. All self-delimiting encodings are stored one after another in a \texttt{bit\_vector} $z$. To provide random access to the \texttt{enc\_vector} we sample each $s_P$-th entry and store for each sample value a pointer to the next encoded value in $z$. The time for the $[i]$-operator therefore mainly depends on the decoding time for $s_P$ encoded integers. We have shown in [Gog09], that the choice of the code and the use of bit-parallel operations in a word can notably speed-up the decoding process. Also note, that in theoretic proposals one chooses the samples in such a way that the differences of the pointers to $z$ of two samples is not greater than $c \log n$ for a constant $c$. The decoding is then done in theory with a lookup
table in constant time and an additional rank_support is needed to indicate if an entry is sampled. However this approach has two drawbacks in practice. First, lookup tables for a reasonably small $c$ do not fit in cache and second the rank_support takes a considerable amount of memory and has slow access times compared to an int_vector, cf. Figure 3.4.

### 3.5 The rank_support and select_support Concept

Other fundamental succinct data structures are rank_support and select_support. Both data structures answer one type of query about a bit_vector $b$ of length $n$ in constant time after a linear time construction phase. In theory, both structures take only $o(n)$ bits of space. In the basic version, rank_support returns for a position $i \in [0..n]$ how many 1-bits are in the prefix $b[0..i-1]$ and select_support returns the position of the $j$-th 1-bit in $b$ (with $j \leq n$). The 1-bit represents a bit pattern of length 1. However, on word RAM model it is possible to generalize the operations and use every other bit pattern up to length $\log n$. Therefore, the data structures take two template parameters which specify (1) the bit pattern and (2) the length of the bit pattern. One example of the use of the generalized versions can be found in the implementation of cst_sada (see Section 3.9.1) which requires the bit pattern “10”.

In the sdsl we provide two classes that fulfill the rank_support concept: The first one takes 0.25$n$ bits of space on top of the original bit_vector $b$ and is called rank_support_v and the second one takes 0.0625$n$ bits and is called rank_support_v5. The implementations of both follow the proposal of Sebastiano Vigna ([Vig08]), which uses a 2-level schema that stores answers for large and medium blocks and then uses bit-parallel operations (e.g. bitmagic::b1Cnt for bit pattern “1”) to answer small block queries. The answers for large and medium blocks are stored interleaved to avoid cache misses. The difference between rank_support_v and rank_support_v5 is the block size $BS$ (in
3.5 The `rank_support` and `select_support` Concept

bits) of the small blocks. We use $BS = 64$ for `rank_support_v` and $BS = 6 \cdot 64 = 384$ for `rank_support_v5`. The time for a rank operation of `rank_support_v` at a random position is about $5 - 6$ times slower than the read access to a C array. Surprisingly, the time for a rank operation of `rank_support_v5` is only about $1.3$ times more than that of `rank_support_v` (see Figure 3.4) which occupies $4$ times more memory.

Clearly, a `select_support` can be realized by a binary search on a `rank_support` without adding extra space. But then each query takes logarithmic time. Therefore, we opted for an implementation of the proposal of David Clark (see [Cla96], page 30) which was adapted to a 64 bit computer architecture and enhanced by bit-parallel operations (e.g. `bitmagic::i1BP` equals the position of the $i$-th set bit in a 64 bit word). In theory the space for the data structure is in $o(n)$. In practice it is about $0.1 - 0.25n$ bits on top of the space of the original `bit_vector` $B$. The constant time queries are realized with a two level schema which stores position information of the ones. First, we divide the `bit_vector` in consecutive regions. Each region is called `superblock` and contains 4096 1-bits. We store for each superblock the position of the first 1-bit in an `int_vector` with integer width $\log n$; i.e. it takes $\frac{n}{4096} \log n$ bits.

A superblock is called a long superblock, if the difference $\Delta$ of the positions of the last and the first 1-bit in the superblock is greater than $\log^4 n$. In this case the 4096 positions of the 1-bits are stored plain in an `int_vector`, which takes $4096 \log n$ bits per long superblock. If $\Delta \leq 4096$ we call the superblock short and store for each 64-th 1-bit in the superblock the relative position to the first 1-bit in the superblock. This takes at most $64 \cdot \log \log^4 n = 256 \cdot \log \log n$ bits for each short superblock. The regions between these stored relative positions are called miniblocks. The answers for the miniblocks are calculated by scanning the blocks until the right 1 is selected. We use bit-parallel operations (in case of the bit pattern “1” we use `bitmagic::b1Cnt` to count the bits in a 64bit word and `bitmagic::i1BP` to get the position of the $i$-th 1-bit in a 64 bit word.

The time for a select operation of `select_support_mcl` corresponds to the time for 3 rank operations on `rank_support_v` or 9 integer array accesses. Therefore, it is clear that the binary search solution for select is much slower than the use of the `select_support_mcl`.

At last, we have a look at the construction time. For a `bit_vector` of size 100 MB it took about 0.1 seconds to construct a `rank_support_v` data structure and about 0.4 seconds to construct a `select_support_mcl` data structure. These times are remarkable: for comparison, a naive `bit by bit`-construction of `select_support_mcl` takes about 26.4 seconds. The trick to speed up the process in such a degree is to use the bit-parallel

---

1 We provide such an implementation called `select_support_bs` However, one should always mention the use of a non constant time `rank_support` in a data structure, as this may be the reason for a notable slowdown. For instance Arroyuelo et al. [ACNS10] implemented a proposal of Geary et al. [GRRR04], which they call RP in the paper, without using the suggested complex constant time select data structure and concluded that “the time performance of RP is clearly the worst”. We have implemented the suggested constant time select data structure of Geary et al. (called `nearest_neighbour_dictionary`) and cannot approve that RP is not competitive in query time to newer approaches.
Design operations mentioned above to select the borders of the mini- and superblocks.

Conclusion:

- Use these non-stand-alone data structures to answer rank and select queries on `bit_vector` which is not compressible.
- Rank queries are about 3 times slower than a random access on a integer array.
- Select queries are about 3 times slower than rank queries.
- The construction of the data structures is very fast.

We provide one standalone data structure for rank and select queries in the library. It was suggested by Geary et al. [GRRR04] and is called `nearest_neighbour_dictionary` in the `sdsl`. It can be used to replace a sparsely and uniformly populated `bit_vector` and at the same time add constant time rank and select functionality (see also Section 3.8.1).

![Figure 3.4:](image)

**Figure 3.4:** From left to right: Runtime of (1) accessing a random element of `int_vector`, (2) a random rank query on `rank_support_v`, (3) a random rank query on `rank_support_v5`, and a random select query on `select_support_mcl`. The times are average times of $10^5$ random queries on the data structures.
3.6 The wavelet_tree Concept

We have already seen in Section 2.6 that the wavelet_tree concept provides rank and select operations on a sequences with $\sigma > 2$. Table 3.3 shows all operations which should be implemented by a class that fulfills the wavelet_tree concept. We provide five wavelet_tree classes in the sdsl. The first four were already described in Section 2.6. The class $\text{wt}$ corresponds to the balanced wavelet tree for small alphabets with $\sigma \leq 256$. The class $\text{wt_int}$ corresponds to the balanced wavelet tree for integer alphabets. The Huffman shaped wavelet tree is represented by the class $\text{wt_huff}$. The run-length encoded wavelet tree described in Section 2.6.2 is implemented in the class $\text{wt_rlmn}$. Each implementation uses one bit_vector $b$ to store a level-wise concatenation of the bit vectors in the wavelet tree nodes. Each wavelet tree can be parametrized by a rank_support and select_support class which supports $b$. Apart from that, we can store information like a partial rank solution for each node in the tree. In the case of a constant $\sigma$, the memory consumption is negligible compared with the size of $b$ and the rank and select structure for $b$. But in the case of an integer alphabet, where we have up to $2n$ nodes, we do not store additional information, as this would double the space of the object. As we consider CSTs for strings over a constant size alphabet we will not come back to $\text{wt_int}$ in this chapter. Note however that $\text{wt_int}$ is very useful for 2d range search [NM07] or position restricted matching [MN06] applications. We will now present a new wavelet tree for a constant size alphabet which also uses run-length encoding to get a space reduction like $\text{wt_rlmn}$.

3.6.1 The Hierarchical Run-Length Wavelet Tree

First note, that in the following we use $T^{BWT}$ as input sequence for the wavelet tree, since this is our main application. It should be clear, that we can use any other character sequence $S$ where $R$ then corresponds to the number of equal character runs in $S$.

We have already presented three variants of run-length wavelet trees in Section 2.6.2. We now present a new proposal for a run-length wavelet tree, called hierarchical run-length wavelet tree, called $\text{wt_rlg}$ in the sdsl, whose space complexity is also upper bounded by $O(nH_k + \sigma^k)$. However, we can get a better constant with the new idea. Remember that we need two bit_vector for the run-length wavelet tree of Mäkinen and Navarro $\text{wt_rlmn}$. The first one is called $bl$ and indicates the beginnings of runs in $T^{BWT}$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>Returns the number of elements in $S$.</td>
</tr>
<tr>
<td>[i]-operator</td>
<td>Returns element $S[i]$ ($i \in [0..\text{size()} - 1]$).</td>
</tr>
<tr>
<td>rank(i, c)</td>
<td>Returns the number of occurrences of symbol $c$ in the prefix $S[0..i - 1]$.</td>
</tr>
<tr>
<td>select(i, c)</td>
<td>Returns the position of the $i$-th occurrences of symbol $c$ in $T^{BWT}$ ($i \in [1..\text{rank_bwt(size()),c}]$).</td>
</tr>
</tbody>
</table>

Table 3.3: Our wavelet_tree concept requires the listed operations.
The second one \(bf\) is a permutation of the first one, i.e. \(bf[i] = bl[LF[i]]\). We need \(2n\) bits if we represent them uncompressed in a \texttt{bit\_vector} and additional space for a rank and select data structure for these \(2n\) bits. We know that we can compress this part of the data structure with the ideas of [GHSV06] and get a new solution for this part which uses less memory. But in practice we have to pay with a longer runtime for the rank and select operations which makes the data structure unattractive compared to wavelet trees which do not use run-length encoding. Therefore, we use in both \texttt{wt\_rlmn} and \texttt{wt\_rlg} only uncompressed \texttt{bit\_vectors}.

In \texttt{wt\_rlmn} we occupy per run of character \(c\) of length \(m\) about \(2m + H_0 + o(m)\) bits. \(2m\) bits for the navigation part in \(bf\) and \(bl\), \(H_0\) bits for character \(c\) in a Huffman shaped wavelet tree \(wt'\), and \(o(m)\) bits for a part of the rank and select structure for all runs.

Our new proposal uses only about \(\frac{1}{g-1}m\) (for a fixed integer \(g > 1\)) bits for navigation and also \(H_0\) \(o(m)\) bits for character \(c\) and the rank and select structure. We will explain the approach by an example and parameter \(g = 2\). Figure 3.5 depicts on the top the input sequence for our wavelet tree: the Burrows-Wheeler transform \(T^{BWT} = T^{BWT_0}\) of our running example. We use a \texttt{bit\_vector} \(b_0\) to indicate if \(g = 2\) adjacent characters, called \(g\)-tuple, are equal and add a rank data structure to \(b_0\). We call a tuple \(x\)-run-tuple if every character in the tuple equals \(x\), and non-run-tuple otherwise. Now we form a string \(wt_0\) by concatenating all non-run-tuples. This string \(wt_0\) can be stored in a Huffman shaped wavelet tree. If there exist run-tuples we form a second (this time conceptional) string \(T^{BWT_1}\). It is the concatenation of the first characters of all run-tuples of the previous level. This process is now repeated until \(b_k\) consists only of zeros. The depth \(d\) of this hierarchical structure is limited by the logarithm to the base \(g\) of the number \(L\) of occurrences of the most frequent character in the text, i.e. \(\log_g L\). So in the worst case...
\[d = \log_g n.\]

We will now show, how we can answer the operations of the `wavelet_tree` efficiently. First we present the algorithm for the \([i]\)-operator. Algorithm 8 contains pseudo-code for the operation. Let's start with an example. Suppose we want to determine the character at position \(i = 11\) of \(T_{BWT}\). We start at level 0 by querying if the tuple of position 11 is a run-tuple or not. This information is stored in \(b_0\) at position \(i_0 = 5\). In our case the tuple is a run-tuple and therefore we have to proceed in the next level. \(T_{BWT}[^1]\) contains the character we search for now at position \(i' = \text{rank}_{b_0}(\frac{i}{2}) = 2\). This time the tuple at position \(i' = 2\) is a non-run-tuple and therefore we find our solution in \(wt_1\); to be precise at position 0, since index \(i'\) is the first position in a non-run-tuple. In the worst case we need \(\log_g L\) times a rank query to calculate the new index plus the access time to the wavelet tree to reconstruct the character, i.e. \(O(\log_g L + H_0)\). The correctness of the procedure can be proved by using the invariant that \(T_{BWT}_{\text{level}}[^i] = T_{BWT}_{\text{level}+1}[i']\) with \(i' = \text{rank}_{b_0}(\frac{i}{2})\) if \(b_0[\frac{i}{2}] = 1\) and \(T_{BWT}_{\text{level}}[^i] = wt_{\text{level}}[2 \cdot \text{rank}_{b_0}(\frac{i}{2}, 0) + i \mod 2]\) otherwise.

The answer of a rank query is slightly more complicated than the \([i]\)-operator. Algorithm 9 depicts pseudo-code again for the case \(g = 2\).\(^1\) The general idea is as follows: One character \(c\) in \(wt_{\text{level}}\) represents \(2^\text{level}\) occurrences of \(c\) in \(T_{BWT}\). In line 7 we add all occurrences left of the tuple to which \(i\) belongs in the current level. It now remains to add the occurrences inside the tuple. For doing that, we have to distinguish three cases:

- If \(i \equiv 1 \mod 2\) and the tuple of \(i\) is a \(x\)-run-tuple (line 9): In this case we cannot add occurrences of character \(c\), since we do not yet know if \(x = c\). And since we only represent one \(x\) of the \(x\)-run-tuple we increase \(i\) by one and calculate a correction term `added` which contains the number of \(x\) we have added virtually by the increasing of \(i\).

- If \(i \equiv 1 \mod 2\) and the tuple of \(i\) is a non-run-tuple (line 13): In this case we can determine the character left of position \(i\) in \(T_{BWT}_{\text{level}}\), i.e. \(T_{BWT}_{\text{level}}[^i - 1] = x\). If \(x = c\) we have to add \(2^\text{level}\) occurrences of \(c\) and subtract the correction term `added` from previous levels. In any case we then have to reset `added` to zero.

- If \(i \equiv 0 \mod 2\) (line 17): In this case we have already counted every occurrence left of position \(i\) in the current level in line 7. However, if we increased the position of \(i\) in previous levels and therefore `added` > 0 we have to subtract the correction term if \(T_{BWT}_{\text{level}}[^i - 1] = x = c\). In any case we then have to reset `added` to zero.

The time complexity for \(\text{rank}(i, c)\) is \(O(\log_g L \cdot H_0)\), since we have at most \(\log_g L\) levels and in each level we perform a constant numbers of accesses to the Huffman shaped wavelet tree \(wt_{\text{level}}\).

In practice we use only one wavelet tree \(wt\), which stores the concatenation of all non-run-tuples \(wt_0 \oplus wt_1 \oplus \cdots \oplus wt_{\log_g L}\), and one `bit_vector` \(b\) which stores the concatenation of all \(b_i\). In addition we need for each level a pointer into \(wt\) and \(b\) and results for partial results of rank. This representation does not only decrease the overhead from \(\log_g L\)

\(^1\) The general case \(g > 2\) has the same time complexity and is implemented in the `sdsl`.
Algorithm 8: The \( [i] \)-operator of \( \text{wt}_\text{rlg} \).

1. \( \text{level} \leftarrow 0 \)
2. \[ \text{while } b\text{level}[\frac{i}{2}] = 1 \text{ do} \]
3. \( i \leftarrow \text{rank}_{b\text{level}}(\frac{i}{2}) \)
4. \( \text{level} \leftarrow \text{level} + 1 \)
5. \[ \text{return } \text{wt}_{\text{level}}[2 \cdot \text{rank}_{b\text{level}}(\frac{i}{2}, 0) + i \mod 2] \]

single wavelet trees but also makes it easy to introduce an additional break condition for the while loop: A character \( c \) does not occur in deeper levels than the current one, if the last rank value equals the number of total occurrences of \( c \) in \( \text{wt} \). Finally, we have implemented the \( \text{select}(i,c) \) operation by a binary search which uses rank queries.

Experimental results of the different wavelet tree classes can be found in Section 3.7 where the wavelet trees are used as building blocks of CSAs.
Algorithm 9 The rank\((i,c)\) operation of wt_rlg.

```
01 res ← 0
02 level ← 0
03 added ← 0
04 while \(i > 0\) do
05 \(\text{ones} ← \text{rank}_{\text{bwt}_{\text{level}}} (\frac{1}{2})\)
06 \(\text{zeros} ← \frac{1}{2} - \text{ones}\)
07 \(\text{res} ← \text{res} + (\text{rank}_{\text{wt}_{\text{level}}} (2 \cdot \text{zeros}, c) \ll \text{level})\)
08 if \(i \mod 2 = 1\) then
09 \(\text{if } b_{\text{level}}[\frac{i}{2}] = 1 \text{ then}\)
10 \(i ← i + 1\)
11 \(\text{added} ← \text{added} + (1 \ll \text{level})\)
12 \(\text{ones} ← \text{ones} + 1\)
13 \(\text{else}\)
14 \(\text{if } c = \text{wt}_{\text{level}}[2 \cdot \text{zeros}] \text{ then}\)
15 \(\text{res} ← \text{res} + (1 \ll \text{level}) - \text{added}\)
16 \(\text{added} ← 0\)
17 \(\text{else}\)
18 \(\text{if } \text{added} > 0 \text{ and } b_{\text{level}}[\frac{i}{2} - 1] = 0 \text{ then}\)
19 \(\text{if } c = \text{wt}_{\text{level}}[2 \cdot \text{zeros} - 1] \text{ then}\)
20 \(\text{res} ← \text{res} - \text{added}\)
21 \(\text{added} ← 0\)
22 \(i ← \text{ones}\)
23 \(\text{level} ← \text{level} + 1\)
24 return res
```

3.7 The csa Concept

In this section we introduce the concept csa for compressed suffix arrays (CSAs). As in the case of a CST the term “compressed” suffix array is somewhat misleading, since the CSA provides more functionality than the (uncompressed) suffix array. All operations which have to be supported are listed in Table 3.4. The first operation is the size of the suffix array, which should be computed in constant time. The second is random access to suffix array entries by the \([i]\)-operator. The third is random access to the inverse suffix array entries by the \((i)\)-operator. The fourth is random access to the Burrows-Wheeler transformed string \(T^{\text{BWT}}\) of \(T\). This is implemented by the \([i]\)-operator of the const member \texttt{bwt}. The fifth and sixth operations are rank and select queries on \(T^{\text{BWT}}\). They can be performed by the member functions \texttt{rank\_bwt}(i,c) and \texttt{select\_bwt}(i,c). Finally, there should be random access to the \(\Psi\) function and its inverse (the LF mapping). This is implemented by the \([i]\)- and \((i)\)-operator of the const member \texttt{psi}.

Note, that our CST definition is more general than the original proposal given by Grossi
### Operation Description

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{size()}</td>
<td>Returns the number of elements of the suffix array.</td>
</tr>
<tr>
<td>\textit{[i]-operator}</td>
<td>Returns the suffix array entry at position ( i ) ((i \in [0..\text{size()} - 1])).</td>
</tr>
<tr>
<td>\textit{(i)-operator}</td>
<td>Returns the inverse suffix array entry at position ( i ) ((i \in [0..\text{size()} - 1])).</td>
</tr>
<tr>
<td>\textit{bwt[i]}</td>
<td>Returns the ( i )-th entry of ( T^{\text{BWT}} ) ((i \in [0..\text{size()} - 1])).</td>
</tr>
<tr>
<td>\textit{rank_bwt(i,c)}</td>
<td>Returns the number of occurrences of symbol ( c ) in the prefix ( T^{\text{BWT}}[0..i - 1] ) of ( T^{\text{BWT}} ).</td>
</tr>
<tr>
<td>\textit{select_bwt(i,c)}</td>
<td>Returns the position of the ( i )-th occurrences of symbol ( c ) in ( T^{\text{BWT}}(i \in [1..\text{rank_bwt(size(),c)}]) ).</td>
</tr>
<tr>
<td>\textit{psi[i]}</td>
<td>Returns the ( i )-th entry of the ( \Psi ) function ((i \in [0..\text{size()} - 1]))</td>
</tr>
<tr>
<td>\textit{psi(i)}</td>
<td>Returns the ( i )-th entry of the LF function ((i \in [0..\text{size()} - 1]))</td>
</tr>
</tbody>
</table>

Table 3.4: Our \texttt{csa} concept requires the listed operations.

and Vitter [GV00] and contains all functionality of the FM-Index of Ferragina and Manzini [FM00]. Note that with the \texttt{csa} concept it is also possible to provide fast random access to the original string \( T \), which is not stored explicitly, by \( T[i] = T^{\text{BWT}}[\text{ISA}[i] + 1 \mod \text{size()}] \). Therefore a CSA is also called \textit{self-index} in literature.

We will present here the implementation details of three CSA classes: \texttt{csa\_sada}, \texttt{csa\_wt}, and \texttt{csa\_uncompressed}. We would like to point out that the \texttt{sds} contains another class called \texttt{csa\_sada\_theo} which realizes the proposal of Grossi and Vitter [GV00] almost verbatim, i.e. it uses a hierarchical decomposition of the \( \Psi \)-function. Since its performance in practice could neither compete in space nor in time with Sadakane’s original CSA implementation\(^1\) we will omit it in this thesis.

Before we have a closer look at the details of each CSA class, we first present the common elements of the three CSA classes:

- The public member \texttt{sigma}, which contains the alphabet size \( \sigma \) of the CSA.
- A public member \texttt{comp2char[]}, which maps a character \( c \) to an corresponding integer \( i \in [0..\sigma - 1] \). The inverse mapping is given by the public member \texttt{char2comp[]}.
- A public member \texttt{C[]}, which corresponds to the character borders of the first column of the sorted suffixes of \( T \). E.g. for a CSA object \texttt{csa} and a character \( c \), \texttt{csa.C[csa.char2comp[c]]} returns the index of the first occurrence of character \( c \) in the first column of the sorted suffixes of \( T \).
- \texttt{SA} and \texttt{ISA} samples: Except for \texttt{csa\_uncompressed} all compressed suffix arrays have template parameters \( s_{\text{SA}} \) and \( s_{\text{ISA}} \) which specify that every \( s_{\text{SA}} \)-th suffix array and every \( s_{\text{ISA}} \)-th inverse suffix array value are stored in the CSA. The values can be accessed with the \([i]\)-operator of the public members \texttt{sa\_sample} and

\(^1\) available at http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/
isa_sample. E.g. for a CSA object csa the suffix array value at position $i \cdot s_{SA}$ equals $\text{csa.sa_sample}[i]$.

In some application it is useful to know $s_{SA}$ and $s_{ISA}$ (e.g. the child operation of the CST). We therefore can access the values by the members $\text{sa_sample_dens}$ and $\text{isa_sample_dens}$.

- The SA values are accessed via the $[i]$-operator and the ISA values are accessed via the $(i)$-operator.
- Access to the $\Psi$ and LF function is provided by a public member $\text{psi}$. The $\Psi$ values are accessed with the $[i]$-operator of $\text{psi}$ and the LF values with the $(i)$-operator.
- Access to the Burrows-Wheeler transformed string $T_{BWT}$ of the original string $T$ is provided by the $[i]$-operator of the public member $\text{bwt}$.

In practice, we need just a few kilobytes to store the information of the first three items in the list, if $\sigma$ is small like in the case of ASCII-alphabet. Usually, more space is needed for (1) the SA and ISA values (i.e. $(\frac{1}{s_{SA}} + \frac{1}{s_{ISA}}) n \log n$ bits) and (2) either the $\Psi$ function or a representation of $T_{BWT}$.

The main difference between the implementations is the way how non-sampled SA values, $\Psi$, LF, and $T_{BWT}$ values are calculated.

### 3.7.1 The Implementation of csa_uncompressed

The uncompressed CSA stores the complete suffix array and inverse suffix array in an $\text{int_vector}$ which takes $2n \log n$ bits. We use two helper classes of constant size to implement the $\text{psi}$ and $\text{bwt}$ members. The classes contain only a pointer $\text{m_csa}$ to the supported CSA object. These classes can be seen as strategy classes. The strategies are straightforward and return the result in constant time. The implementation of $\text{psi}[\cdot]$ and $\text{psi}()$ use Equations 2.6 and 2.5 to answer them in constant time.

The strategy class for $\text{bwt}$ first calculates $j = \text{LF}[i]$ and then uses a binary search on $\mathcal{C}$ to get the character interval to which position $j$ belongs, i.e. $\text{bwt}[\cdot]$ takes $O(\log \sigma)$ time. Finally, $\text{rank_bwt}(i,c)$ is realized by a binary search on $\Psi$ and $\text{select_bwt}(i,c)$ in constant time by using $\mathcal{C}$ and $\Psi$.

### 3.7.2 The Implementation of csa_sada

The $\text{csa_sada}$ class is a very clean reimplementation of Sadakane’s CSA[Sad02]. The class takes four template arguments. The first one is a vector type which is used to store the differences of the $\Psi$ function values, the second and third one are the sample rates $s_{SA}$ and $s_{ISA}$, and the last one is an integer width $w$ for the $\text{int_vector}$s in which the SA and ISA samples are stored. The default type for the first template parameter is the $\text{enc_vector}$ class (see Section 3.4.2) which takes a sampling parameter $s_{\Psi}$.

The size of a $\text{csa_sada}$ object mainly depends on the sampling parameters $s_{\Psi}$, $s_{SA}$, and $s_{ISA}$. A good choice for $s_{\Psi}$ in practice is about $64 - 128$. 

As we have constant time access to $\Psi$, all other operations are realized with this operation. The $[i]$-operator can be implemented by applying $\Psi$ $k$-times until a sampled SA value at position $x = \Psi^k[i]$ is found: $\text{SA}[i] = \text{SA}[x] - k$. The $(i)$-operator can be realized by first calculating the smallest index $i' \geq i$ and then apply $k = i \mod s_{\text{SA}}$ times $\Psi$, i.e. $\text{ISA}[i] = \Psi^k[\text{ISA}[i']]$. While the $(i)$-operator takes at most $s_{\text{ISA}}$ steps the $[i]$-operator takes more if the permutation $\Psi$ contains long sequences with $\text{SA}[\Psi^k(i)] \neq 0 \mod s_{\text{SA}}$. Therefore it is not surprising that in practice (see Figure 3.6) the $(i)$-operator is faster than the $[i]$-operator. The LF function is realized by Equation 2.5. Finally $\text{bwt}[i]$ can be calculated by LF and a binary search on C.

### 3.7.3 The Implementation of csa wt

The class `csa wt` is based on a wavelet tree, which is also the first template parameter of the class. The sample parameters $s_{\text{SA}}$ and $s_{\text{ISA}}$ are passed as second and third parameter. As we have already seen, $\Psi$ as well as LF can be calculated by Equations 2.10 and 2.9 which only depends on the used wavelet tree and on C. As LF depends on a rank query of the wavelet tree it can be answered faster than $\Psi$ which depends on a select query (see also Figure 3.4 for the running times). Therefore the $[i]$-operator this time is implemented by applying the LF function $k$ times until a sampled SA value at position $x = \text{LF}^k$ is found: $\text{SA}[i] = \text{SA}[x] + k$. The $(i)$-operator can be realized by first calculating the largest index $i' \leq i$ and then apply $k = i - i'$ times LF, i.e. $\text{ISA}[i] = \text{LF}^k[\text{ISA}[i']]$.

### 3.7.4 Experimental Comparison of CSA Implementations

In this section we will show how fast the operations of the CSA concept can be performed on selected CSA configurations. Note that all implementations are based on the same basic data structures and use the same basic operations. This should result in a fair comparison. We will present in total seven different CSA configurations. Four CSAs are

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>csa_uncompressed</code></th>
<th><code>csa_sada</code></th>
<th><code>csa_wt</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$[i]$-operator</td>
<td>$O(1)$</td>
<td>$O(s_{\text{SA}} \cdot t_{\Psi})$</td>
<td>$O(s_{\text{ISA}} \cdot t_{\Psi})$</td>
</tr>
<tr>
<td>$(i)$-operator</td>
<td>$O(1)$</td>
<td>$O(s_{\text{ISA}} \cdot t_{\Psi})$</td>
<td>$O(s_{\text{ISA}} \cdot t_{\Psi})$</td>
</tr>
<tr>
<td>$\text{bwt}[i]$</td>
<td>$O(\log \sigma)$</td>
<td>$O(t_{\Psi} \cdot \log \sigma)$</td>
<td>$O(t_{\Psi} \cdot \log \sigma)$</td>
</tr>
<tr>
<td>$\text{rank}_\text{bwt}[i,c]$</td>
<td>$O(\log n)$</td>
<td>$O(t_{\Psi} \cdot \log n)$</td>
<td>$O(t_{\Psi} \cdot \log n)$</td>
</tr>
<tr>
<td>$\text{select}_\text{bwt}[i,c]$</td>
<td>$O(1)$</td>
<td>$O(t_{\Psi})$</td>
<td>$O(t_{\Psi})$</td>
</tr>
<tr>
<td>$\text{psi}[i]$</td>
<td>$O(1)$</td>
<td>$O(t_{\Psi})$</td>
<td>$O(t_{\Psi})$</td>
</tr>
<tr>
<td>$\text{psi}(i)$</td>
<td>$O(1)$</td>
<td>$O(t_{\text{ISA}} + t_{\text{ISA}})$</td>
<td>$O(t_{\text{ISA}} + t_{\text{ISA}})$</td>
</tr>
</tbody>
</table>

**Table 3.5:** Theoretic worst case time complexities of different CSA implementations. We denote the time complexity of a basic $\text{rank}$ ($\text{select}$) operation by $t_{\Psi}$ ($t_{\Psi}$). The time for accessing the $\Psi$ (LF) function $t_{\Psi}$ ($t_{\text{ISA}}$) depends on the parametrization of the CSA. $t_{\text{ISA}}$ ($t_{\text{ISA}}$) denotes the time for the $[i]$-operator ($(i)$-operator) of the CSA.
based on different wavelet trees, two CSAs are based on the compressed $\Psi$ function, and on CSA is the bit-compressed one. Here are the details:

- **csa_wt<wt<>**: A CSA based on a balanced wavelet tree.
- **csa_wt<wt_huff<>**: A CSA based on a Huffman shaped wavelet tree.
- **csa_wt<wt_rlnn<>**: A CSA based on the run-length encoded wavelet tree of Mäkinen and Navarro.
- **csa_wt<wt_rlg<8>>**: A CSA based on the new run-length encoded wavelet tree with $g = 8$.
- **csa_sada<enc_vector<coder::elias_delta,128>>**: A CSA based on the $\Psi$ function, which is stored in an enc_vector with $s_\Psi = 128$ and the use of Elias-$\delta$ code. Abbreviated by csa_sada<$\delta$> in the figures.
- **csa_sada<enc_vector<coder::fibonacci,128>>**: A CSA based on the $\Psi$ function, which is stored in an enc_vector with $s_\Psi = 128$ and the use of Fibonacci code. Abbreviated by csa_sada<$\Phi$> in the figures.
- **csa_uncompressed**: The uncompressed version which uses about $2n \log n + n \log \sigma$ bits as a baseline. The $[i]$-operator and the $(i)$-operator take about 100 nanoseconds.

We have built the CSAs for the test cases of the *Pizza&Chili* corpus. As the results for the small test cases of size 50, 100, and 200 MB are almost equal we present only the last ones here. We measured time and space for each CSA. We have chosen different values for $s_\text{SA} = s_\text{ISA}$ to get a time-space trade-off for each CSA. Figure 3.6 contains the results for the runtime of the $[i]$-operator. Remember that the uncompressed CSA takes about 100 nanoseconds per access but also 56 bits per input character. The best CSA, which is cst_sada parametrized with $s_\text{SA} = 4$ for this operation takes about 10 times longer while using only 16 – 25 bits per input character. The CSA based on wavelet trees are clearly slower than csa_sada, except for the case of DNA data. This was expected since the theoretic time complexity of csa_sada is $O(s_\text{SA} \cdot t_\Psi)$ and the time complexity of the wavelet tree based solutions are $O(s_\text{SA} \cdot t_r \log \sigma)$ or $O(s_\text{SA} \cdot t_r H_0)$. For a complete list of theoretic time complexities see Table 3.5. Figure 3.6 also shows the difference of the last two time complexities in practice. The csa_wt based on the Huffman shaped wavelet tree is always as good or better than the csa_wt based on a balanced wavelet tree. The most notable improvement is achieved in the case of the DNA sequence, which has an alphabet size of 17, since it contains some occurrences of special chars besides the four bases $a$, $c$, $g$, and $t$. In this case, csa_wt<wt_huff<> also beats csa_sada, since $H_0$ is only about 2.

Now we have a closer look at the compression ratio of the CSAs. We can observe that CSAs which are based on run-length compressed wavelet trees use notably less space in the test cases dblp.xml, english, and sources. These test cases contain many repeats (which can be also deduced from Table 2.1 where $H_3$ is at most the half of $H_0$) and therefore the $nH_0$ compression of csa_wt<wt_huff<> is not optimal. The dblp.xml
test case shows that CSAs based on run-length encoded wavelet trees can provide a better time-space trade-off. Both csa_wt<wt_rlmn<> and csa_wt<wt_rlg<8>> use only 4.5 bits to answer the \( [i] \)-operator in about 100 microseconds, while csa_wt<wt_huff<> uses 8 bits.

Finally, we have a look at the compression which is reached by setting \( s_{SA} \) to 64. Both CSAs which do not contain a select structure, i.e. csa_sada and csa_wt<wt_rlg<8>>, dominate this category. However, we will see in the next paragraph that this makes them poor alternatives for supporting other operations fast.

Figure 3.7 contains the running times of three additional operations on our CSTs. We use the same time and space limits for the diagrams as in Figure 3.6 to make it easy to compare the operation times. Let’s start with csa_wt<wt<>: The access times for LF, \( \Psi \), and bwt[] do not change for different values of \( s_{SA} \), since they solely depend on the wavelet tree, which has always equal size in our scenario. The LF and bwt[] operations which correspond to rank operations are about 3 times faster than the \( \Psi \) operation which corresponds to select operations in a wavelet tree. This perfectly matches our expectation, since a rank data structure is about 3 times faster than a select structure (see Figure 3.4).

The same observations can also be made for csa_wt<wt_huff<>.. Therefore, we directly switch to csa_wt<wt_rlmn<>.. Here the ratio between LF and \( \Psi \) is smaller. This can be explained as follows: The LF method requires a rank operation of wt_rlmn<>. The pseudo-code for this rank operation is depicted in Algorithm 4. It contains multiple rank and select queries on bit vectors and a Huffman shaped wavelet tree \( wt' \). The select operation of wt_rlmn<>, which is required to calculate \( \Psi \), is composed of multiple rank and select queries as well (see Algorithm 5). Let \( t_r \) be the time for a rank query on a bit_vector and \( t_s \) the time for a select query. If we count the number of basic rank and select operations for LF and \( \Psi \) we get the times: \( (3H_0 + 2)t_r + 2t_s \) and \( 3t_r + (H_0 + 2)t_s \), i.e. for our observed ratio for \( 3t_r = t_s \) the times are almost the same: \( (3H_0 + 8)t_r \) vs. \( (3H_0 + 9)t_r \). Since the bwt[] operation with the same calculation requires only \( H_0 + 1 \) basic rank operations, it is expected that it takes at most third of the time of LF or \( \Psi \). This matches the empirical results.

The CSA based on the new wavelet tree wt_rlg is competitive with wt_rlmn only for the bwt[] operation. The bwt[] operation takes \( H_0 + \log_8 n \) basic rank operations in the worst case and \( H_0 \) in the best case. Since LF and \( \Psi \) have worst case time complexities of \( H_0 \cdot \log_8 n \) and \( H_0 \cdot \log_8 n \cdot \log n \), the empirical results for these operations are also not surprising.

Finally, we consider the results of csa_sada. Since csa_sada uses an enc_vector to store \( \Psi \) it is clear that the random access to \( \Psi \) can be performed fast by a constant time random access on enc_vector. Sadakane describes implicitly in Section 4 of [Sad02] how also the LF operation can be performed in constant time by adding a data structure for rank and predecessor queries on \( \Psi \). Unfortunately, the promised \( H_0 \)-factor can only be realized with the hierarchical implementation of the CSA, which we have discarded due

1 Note that our definition of \( H_0 \) corresponds to \( H_1 \) in [Sad02].
Figure 3.6: The runtime of the \([i]\)-operator of different CSA implementations included in the \textit{sdsl}. Different time-space trade-offs are achieved by setting \(s_{\text{SA}} = s_{\text{ISA}}\) to 4, 8, 16, 32, and 64. The runtime of accessing the inverse suffix array with the \((i)\)-operator is at the same sample rate about the half of the runtime of the \([i]\)-operator.
Design to a big space overhead. Therefore, we have implemented LF by using Equation 2.5 and \texttt{bwt[\ldots]} by using LF and C. For this reason the running time of both operations depends on \(s_{SA}\) and \(s_{ISA}\).

Conclusion:

- \texttt{csa\_sada} is the best choice, if we only need fast access to \(SA\), \(ISA\), and \(\Psi\).

- The \textit{Swiss Army knife} of CSAs is \texttt{csa\_wt<wt\_huff>} since it offers good time-space trade-offs for all operations and even outperforms \texttt{csa\_sada} on DNA sequences. Possibly the performance can be further improved by using a 4-ary Huffman tree for the wavelet tree.

- A CSA based on run-length encoded wavelet trees like \texttt{wt\_rlmn} or \texttt{wt\_rlg} can be used to further compress strings with many repeats. While \texttt{wt\_rlg} uses less space than \texttt{wt\_rlmn}, the latter provides better runtimes on every operation.
Figure 3.7: The runtime for a random access to $\Psi$ (i.e. $\text{psi[]}$), LF (i.e. $\text{psi()}$), and the Burrows-Wheeler transform of the original string $T$ (i.e. $\text{bwt[]}$) of different CSA implementations included in the sdsl. Different time-space trade-offs are achieved by setting $s_{SA} = s_{ISA}$ to 4, 8, 16, 32, and 64. Lines with a ◦ correspond to $\Psi$, lines with a △ correspond to LF, and lines with a + correspond to $\text{bwt[]}$. 
3.8 The bp_support Concept

The NAV structure of a CST consists of two parts. First, a balanced parentheses sequence which represents the tree topology and second a helping data structure which is called bp_support in the sdsl and supports the operations listed in Table 3.6.

One possibility of representing a tree (of \(n\) nodes) as sequence of balanced parentheses is the following. Let \(v\) be a node of the tree. Remember that we denote the numbers of children of \(v\) by degree\((v)\) and the \(i\)-th child by ith_child\((v,i)\). The representation rep\((v)\) is then defined recursively as:

\[
\text{rep}(v) = \left( \text{rep}(\text{ith}_1 \text{child}(v,1)) \ldots \text{rep}(\text{ith}_{\text{degree}(v)} \text{child}(v,\text{degree}(v))) \right)
\]

With this definition we get for example the representation \(((())(()()))()(()())\) for the tree which is depicted in Figure 3.8. The representation can be constructed from the uncompressed tree representation by a depth first search traversal. We write an opening parenthesis before we process the child nodes of the node and write a closing parenthesis after we have processed all child nodes. We denote the resulting balanced parentheses sequence with BPS\(_{dfs}\). While the time complexity for the construction is optimal, the space complexity is not, since we need in the worst case at least \(n \log n\) bits for the stack of the depth first search traversal. We will see in Chapter 5 how to do this with only \(3n + o(n)\) bits.

The representation occupies \(2n\) bits as we write two parentheses for each node. The space is optimal in terms of the information-theoretic lower bound, since there are \(C_n = \frac{1}{2n+1} {2n+1 \choose n} = 2^{2n}/\Theta(n^2)\) different binary trees [GKP94]. Thus we need \(2n - o(n)\) bits to distinguish binary trees, and thus \(2n\) bits for general trees is asymptotically optimal.

Sadakane represents the tree topology of his CST by the BPS\(_{dfs}\) of size \(4n\), since a suffix tree of a string of length \(n\) has \(2n\) nodes in the worst case. He showed how to solve the navigational operations of the CST concept like root\(), parent\((v)\), ith_leaf\((i)\), sibling\((v)\), etc. by using the operations of the bp_support concept. But before we can solve navigational operations we first have to define how a node is represented. Sadakane identifies every node with the position of its corresponding opening parenthesis in BPS\(_{dfs}\). Therefore, root equals 0 in his representation and the parent\((v)\) operation equals the enclose. We will not further discuss details here, as we will do this later when we present our solution in comparison to Sadakane’s. The only thing you should keep in mind is that we can express all navigational operations of the CST with the operations of Table 3.6. Now we show how we can perform all the operations in constant time while only using at most \(o(n)\) bits additional space on top of the balanced parentheses sequence.

In the late 1980s Jacobson [Jac89] engineered a data structure which answers excess, find_close, and find_open. The naive implementation of this data structure takes \(10n + o(n)\) bits of space. Munro and Raman [MR97] suggested in 1997 another data structure for
balanced parentheses which supports additionally the \textit{enclose} operation in constant time. They also proposed a solution for the \textit{double_enclose} operation in the proceedings version of their paper. However, the solution does not work in general and so it was removed in the journal version. In 2004, Geary et al. \cite{GRRR04} suggested a practical version of Jacobson’s data structure and a new hierarchical data structure, which supports \textit{excess}, \textit{find_close}, \textit{find_open} and \textit{enclose} in constant time. We showed in \cite{GF10} how we can extend the data structure of Geary et al. to support the \textit{rr_enclose} and \textit{double_enclose} operation. At the same time Sadakane and Navarro \cite{SN10} published another data structure, which also solves all operations in constant time and \(o(n)\) space. It is based on a tree which stores the minimum and the maximum excess values and therefore also called \textit{range min-max-tree}. In the \textit{sdsl} we provide two versions of our proposal (classes \texttt{bp_support\_g} and \texttt{bp_support\_gg}) and a reimplementation of the range min-max-tree enhanced with bit-parallel tricks (class \texttt{bp_support\_sada}).

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Operation & Description \\
\hline
\texttt{size()} & Returns the size of the supported \texttt{bit\_vector} \texttt{b}. \\
\texttt{rank(i)} & Returns the number of 1-bits in \texttt{b[0..i]}, where \(0 \leq i < \texttt{size}()\). \\
\texttt{select(i)} & Returns the position of the \texttt{i}-th 1-bit in \texttt{b}, with \(i \in [1..\texttt{rank(size())} - 1]\). \\
\texttt{excess(i)} & Number of 1 bits minus the number of 0-bits in \texttt{b[0..i]}, where \(0 \leq i < \texttt{size}()\). \\
\texttt{find\_close(i)} & The position of the matching 0-bit to a 1-bit at position \(i\), where \(0 \leq i \leq \texttt{size}()\) and \(b[i] = 1\). Formally: \(\texttt{find\_close(i)} = \min \{ j \mid j > i \land \texttt{excess}(j) = \texttt{excess}(i) - 1 \}\). \\
\texttt{find\_open(i)} & The position of the matching 0-bit to a 0-bit at position \(i\), where \(0 \leq i < \texttt{size}()\) and \(b[i] = 0\). Formally: \(\texttt{find\_open(i)} = \max \{ j \mid j < i \land \texttt{excess}(j) = \texttt{excess}(j) + 1 \}\). \\
\texttt{enclose(i)} & Returns the maximal position \(j < i\) of an 1-bit which together with its matching 0-bit encloses the 1-bit at position \(i\) and the 0-bit at position \(\texttt{find\_close(i)}\). Formally: \(\texttt{enclose}(i) = \max \{ j \mid j < i \land \texttt{find\_close}(j) > \texttt{find\_close}(i) \}\). \\
\texttt{double\_enclose(i,j)} & Returns for two matching bit pairs at positions \((i,\texttt{find\_close}(i))\) and \((j,\texttt{find\_close}(j))\) with \(\texttt{find\_close}(i) < j\) a bit pair which tightest encloses both bit pairs. Formally: \(\texttt{double\_enclose}(i,j) = \max \{ k \mid k < i \land \texttt{find\_close}(k) > \texttt{find\_close}(j) \}\). \\
\texttt{rr\_enclose(i,j)} & Returns for two matching bit pairs at position \((i,\texttt{find\_close}(i))\) and \((j,\texttt{find\_close}(j))\) with \(\texttt{find\_close}(i) < j\) a bit pair whose opening parenthesis lies leftmost in the range \([\texttt{find\_close}(i) + 1..j - 1]\) and its matching parenthesis to the right of \(\texttt{find\_close}(j)\). Formally: \(\texttt{rr\_enclose}(i,j) = \min \{ k \mid k \in [\texttt{find\_close}(i) + 1..j - 1] \land \texttt{find\_close}(k) > \texttt{find\_close}(j) \}\). \\
\hline
\end{tabular}
\caption{Our \texttt{bp\_support} concept requires the listed operations.}
\end{table}
In the rest of this section, we first present a detailed description of our new bp_support data structure, then we present a very brief description of the range min-max-tree, and finally we will compare the different proposals.

3.8.1 The New bp_support Data Structure

Subdivide and Conquer

In the following, let \( bp \) be the sequence of \( n \) balanced parentheses pairs. All succinct data structures for balanced parentheses sequences have a common core structure. First, \( bp \) is subdivided into blocks of a fixed size, say \( \text{block size} \). Then, we store additional information for each block and for each supported operation.

In theory, the block size is chosen as \( c \log n \) (\( c \) is a constant) and we therefore get at most \( 2^c \log n = n^c \) different kinds of blocks. One can then calculate all query answers for each operation in advance; that is we use the well known Four Russians Trick [ADKF70]. If \( c < 1 \), the lookup tables for the query answers take only \( o(n) \) bits of space and therefore we have a succinct data structure. In practice, we choose \( c \in \{4,8,16\} \) and use lookup tables for smaller blocks of sizes 8 or 16 (so that they fit in cache). The answer for one block is then composed of several table lookups on these small tables. It turned out that this approach is the fastest in practice.

A first example for a succinct data structure for balanced sequences is depicted in Figure 3.9. The balanced parentheses sequence is subdivided into blocks of block size = 8. In the following we denote the parenthesis at index \( i \) as parenthesis \( i \) and \( \beta(i) \) is the block number of the block in which parenthesis \( i \) lies. Notice that there are two kinds of parentheses: Near and far parentheses. The matching parenthesis \( \mu(i) \) for a near parenthesis \( i \) lies in the same block as \( i \), i.e. \( \beta(\mu(i)) = \beta(i) \). All other parentheses are called far. To answer \( \text{find close}(i) \) in our example we construct a lookup table, called \( \text{findclose table} \), of size \( 2^8 \times 8 \) (number of different blocks \( \times \) number of query positions). The answer for \( \text{find close}(9) \) is then calculated as follows. The block number is \( \beta(9) = 1 \) and in this block we have to answer the query on index \( 9 \mod 8 = 1 \). Therefore the answer is \( \beta(9) \cdot 8 + \text{findclose table}[())(([])[1] = 8 + 4 = 12 \). The lookup table can also be used to identify far parentheses. For this purpose we store a special value \( \perp \) for all queries which result in the answer that no near parenthesis exists. Unfortunately we cannot store all answers for far parentheses as there can be up to \( 2n \) far parentheses in \( bp \) and so it would take \( \mathcal{O}(n \log n) \) bits of space to store all the answers.

Pioneers Help to Conquer

Jacobson [Jac89] was the first who introduced pioneers to solve the operations \( \text{find close}(i) \) and \( \text{find open}(i) \) with a succinct data structure. Later, Geary et al. [GRRR04] slightly changed the definition of Jacobson and got a very nifty data structure. So, we now present the definition of [GRRR04]. Let \( M_{far} = \{ i \mid \beta(i) \neq \beta(\mu(i)) \} \) be the set of far parentheses in \( bp \). The subset

\[
M_{pio} = \{ i \mid i \in M_{far} \land \beta j : (\beta(j) = \beta(i) \land \beta(\mu(j)) = \beta(\mu(i)) \land \text{excess}(j) < \text{excess}(i)) \}
\]
of $M_{far}$ is called the set of pioneers of $bp$. It has three nice properties:

(a) The set is symmetric with regard to opening and closing pioneers. That is if parenthesis $i$ is a pioneer, $\mu(i)$ is also a pioneer.

(b) The parentheses sequence consisting only of the parentheses from $M_{pio}$ is again balanced. We call this sequence pioneer $bp$.

(c) The size of $M_{pio}$ depends on the number of blocks, say $b$, in $bp$. There are at most $4b - 6$ pioneers if $b > 1$. Proof sketch from [Jac89]: The pioneer graph $PG$, which consists of nodes $1, \ldots, b$ and edges $(\beta(i), \beta(\mu(i)))$ ($i \in M_{pio}$), is outerplanar and has no multiple edges. We know from graph theory that $PG$ then has at most $2n - 3$ edges.

So in contrast to the set of far parentheses whose size cannot be bounded by a multiple of $b$, the set of pioneers is bounded by $4b$. Now we present how to answer $\text{find\_close}(i)$ queries for far parentheses. First we add a bitmap pioneer bitmap of size $2n$ to the data structure which indicates whether a parenthesis is in the set $M_{pio}$ or not (see Fig. 3.9). Let parenthesis $i$ be opening and far. If it is a pioneer itself, we return $\mu(i)$. Otherwise, we determine the maximal pioneer $p < i$. $p$ is also an opening pioneer. (Otherwise, if $p$ would be closing, we can show that $i$ is itself an opening pioneer.) We determine $\mu(p)$, which lies per definition in the same block as $\mu(i)$. As all excess values in the range $[i, \mu(i) - 1]$ are greater than $\text{excess}(i) - 1$, we have to find the minimal position, say $k$, in $\beta(\mu(p))$ with $\text{excess}(k) = \text{excess}(i) - 1$. This can be done either by lookup tables or a left-to-right scan.
Jacobson’s Original Data Structure Revisited

To support \texttt{find.close}(i) and \texttt{find.open}(i), Jacobson used the following data structures:

- \texttt{pioneer bitmap} and a \texttt{rank support} and \texttt{select support} for it ($2n + o(n)$ bits).

- An array \texttt{matches}, which stores for each pioneer $p$ the matching block $\beta(\mu(p))$. This takes $4n/\text{block size} \cdot \log n$ bits.

- An array \texttt{excess}, which stores the first excess value in each block. This takes $n/\text{block size} \cdot \log n$ bits.

The \texttt{excess} array can be easily replaced by a $o(n)$ rank support for $bp$. So, for a block size of $\frac{1}{2} \log n$ this data structure occupies $10n + o(n)$ bits. Even more space is occupied, if \texttt{enclose}(i) has to be supported.

Geary et al.’s Data Structure Revisited

Geary et al. use property (b) of the pioneers to construct a succinct data structure. Figure 3.10 shows an example for a sequence of length 20 and \texttt{block size} = 4. On each level there is a conceptual balanced parentheses sequence and a bitmap that stores the position of the pioneers. In the original paper, only two levels are stored and on the third level the answers for each query type are stored explicitly. Therefore, the query time for \texttt{find.close}(i), \texttt{find.open}(i), etc. remains constant as there are only three levels with constant time operations. However, the presentation of the structure is easier if one proceeds with the recursion until the size of the sequence is less than or equal to \texttt{block size}. That is we neither have to count on which level we are in a procedure nor handle special cases. Let us first calculate how many levels we need for this representation. Let $n_i$ be the number of parentheses in level $i$. So $n_0 = 2n$. In the first level we have $n_1 = 4\frac{n_0}{\text{block size}} - 6$ (see property (c) for the factor 4) and on the second $n_2 = 4\frac{n_1}{\text{block size}} - 6 = 4^2\frac{n_0}{\text{block size}^2} - 6 \cdot 4 - 6$. In general, we have $n_k = 4^k\left(\frac{2n}{\text{block size}} - 2\right) + 2$. The recursion ends when $n_k \leq \text{block size}$ holds. If we choose the block size as $\log n$ we get about $k = \frac{\log n}{\log \log n}$ levels. It follows that the time complexity for \texttt{find.close}(i), \texttt{find.open}(i), etc. lies in $O(\frac{\log n}{\log \log n})$ if we only use constant time operations on each level.

\[
bp = ((((( ))())((())(())()))) \\
pioneer\_bmp(bp) = \begin{array}{cccccc}
1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \\
pioneer\_bmp(\text{pioneer}_bp(bp)) = \begin{array}{cc}
1 & 0 & 0 & 0 & 1 \\
\end{array} + \text{rank/select} \\
pioneer\_bp(\text{pioneer}_bp(bp)) = 0 & 1 + \text{rank/select}
\]

\textbf{Figure 3.10:} Hierarchical data structure for balanced parentheses. The block size is set to four.
We now calculate the space complexity of the data structure. As one can see in Fig. 3.10, we store on each level (1) a balanced parentheses sequence and (2) a bitmap to mark the pioneers. This bitmap is supported by a rank\_support and select\_support. The size of the balanced parentheses sequence at level \( k \) is at most \( n_k = 4^k \left( \frac{2n}{\text{block\_size}^k} - 2 \right) + 2 \). Summing up the sizes of all levels (as geometric sum), we get at most \( 2n \) bits. Rank and select can be supported in \( o(n) \) space by constructing one rank\_support and one select\_support for the concatenation of all balanced parentheses sequences of all levels and storing pointers to the beginning of each balanced parentheses sequence of each level. This requires at most \( o(2n) + \Theta \left( n \log \frac{n}{\log \log n} \right) \) bits.

If we would store the bitmaps for the pioneers explicitly on each level, we would also get \( 2n + o(n) \) bits. But as there are only few ones in the bitmaps, we use a gap encoded dictionary which support rank/select queries on a bitmap, instead of storing the bitmap itself. Geary et al. presented such a gap encoded dictionary, called a simple nearest neighbour dictionary for uniform sparse sets, that occupies \( \Theta \left( n \log \log n \right) = \Theta(n_k) \) bits for a bitmap of size \( n_k \). Summing up the dictionaries for each level, we get \( o(n) \) bits of space. So in total the data structure takes \( 2n + o(n) \) bits (including the \( 2n \) bits for the supported parentheses) or \( o(n) \) bits for the supporting data structure itself. In practice, our implementation takes about \( 0.4 \) – \( 0.5 \) bits per supported parentheses, where most of the space is occupied for the rank and select functionality. So for a balanced parentheses sequence of length \( 2n \) we need \( 2.8n - 3.0n \) bits.

The find\_close Operation in Detail

The code of the find\_close operation is given in Algorithm 10. The method takes a balanced parentheses sequence \( \text{bp} \), a parenthesis \( i \), and the block size as input. We have defined \( \text{find\_close}(i) = i \) if \( i \) is already a closing parenthesis. This is reflected by the condition at line 1. The method has one recursive call (see \( \ominus \)) and one base case which is solved by the subroutine near\_find\_close (see \( \circ \)).

Example Let us now explain the code by an example. We take the balanced parentheses sequence \( \text{bp} = (((((())(((())()))))))) \), \( \text{block\_size} = 5 \), and ask for the solution of find\_close(1). See Figure 3.11 for the original parentheses sequence grouped into blocks of size 5 and the pioneers which are marked bold. It also contains the recursively constructed parentheses sequence of the pioneers of \( \text{bp} \). The method first checks whether \( i = 1 \) is a closing parenthesis. In our case it is an opening parenthesis, so we continue with step \( \bigcirc \). That is the subroutine near\_find\_close, which scans the block \( \beta(1) \) for a near matching closing parenthesis. Unfortunately, no such parenthesis is found and the subroutine returns 1, which leads us to step \( \ominus \). A rank query on the pioneer\_bitmap returns the position \( p' = 0 \) of the maximal pioneer \( p \leq i \) in pioneer\_bp. Recall, that pioneer\_bp was the recursively constructed balanced parentheses sequence of the pioneers of \( \text{bp} \). Next, we get the answer to find\_close on pioneer\_bp for \( p' = 0 \). This is \( mp' = 5 \). A select query (see \( \circ \)) on the pioneer\_bitmap returns the corresponding closing parentheses for \( p \) in \( \text{bp} \). In our case \( mp \) is set to 19 at line 10. Parenthesis 19 lies in block \( \beta(19) = 3 \). We finally scan in step \( \bigcirc \) from the left end of block 3 and search for the first parenthesis \( mi \) with
excess(mi) = excess(1) − 1 = 1. This results in mi = 16.

Correctness We have to show that a call of find_close(i) for an opening parenthesis i results in the minimal closing parenthesis, say mi, with mi > i and excess(mi) = excess(i) − 1. Note that there is always a valid solution if the parentheses sequence is balanced. We prove the correctness by induction on the length n of the parentheses sequence. If n ≤ block_size the answer lies in the same block; i.e. the subroutine near_find_close (see step 1⃝) finds mi. If n > block_size and near_find_close does not find mi, it follows that β(mi) > β(i).

In steps 2⃝ we get the position p′ of the maximal pioneer p ≤ i in pioneer_bp. Note that such a pioneer p always exists, since i is a far parenthesis and would be itself a pioneer if no pioneer exists left to i. Furthermore, p has to be an opening pioneer, for if p would be closing we would get the following contradictions:

1. If p = i, p is closing but i is opening.
2. If p < i and p is a closing pioneer, there cannot be an opening pioneer left to i. It follows, that i itself is an opening pioneer.

Step 3⃝ correctly calculates by induction the result for find_close(p′) on pioneer_bp. In step 4⃝ we then get the position of the matching parentheses mp of p. By the definition of the pioneers, we know that mi lies in the same block as mp. So scanning from left to right in block β(mp) for the first parenthesis with excess value equal to excess(i) − 1 in step 5⃝ correctly results in mi, as all excess values between i and mi − 1 are greater than excess(i) − 1.

Runtime The runtime for steps 1⃝ and 5⃝ is constant if we use lookup tables of size O(block_size · 2block_size). The rank and select queries in steps 2⃝ and 3⃝ can also be performed in constant time. Since there are at most O(log n/ log log n) recursive calls, the time complexity for Algorithm 10 is O(log n/ log log n). Note, that this can be improved to constant time, if one uses only 3 levels where the last level stores the answers explicitly.

The find_open Operation in Detail

The code of the find_open operation is given in Algorithm 11. The method takes a balanced parentheses sequence bp, a parentheses i, and the block size as input. We have defined find_open(i) = i if i is already an opening parenthesis. This is reflected by the condition at line 1. As in the find_close(i) operation, we search for a near answer in step 1⃝. In step 2⃝, we get the minimal pioneer p ≥ i and this time we get a closing pioneer. The recursive call (1⃝) and the calculation of mp in step 4⃝ is analogous to the steps in
Algorithm 10 Pseudo code for the find_close operation.

\[
\text{find_close}(bp, i, \text{block\_size})
\]

01 if \(bp[i] = ')'\)  
02 return \(i\)  
03 else  
04 \(mi \leftarrow \text{near\_find\_close}(bp, i, \text{block\_size})\)  
05 if \(mi > i\)  
06 return \(mi\)  
07 else  
08 \(p' \leftarrow \text{rank1}(\text{pioneer\_bitmap}, i + 1) - 1\)  
09 \(mp' \leftarrow \text{find\_close}(\text{pioneer\_bp}, p', \text{block\_size})\)  
10 \(mp \leftarrow \text{select1}(\text{pioneer\_bitmap}, mp' + 1)\)  
11 \(mi \leftarrow \beta(mp) \cdot \text{block\_size}\)  
12 while \(\text{excess}(mi) + 1 \neq \text{excess}(i)\)  
13 \(mi \leftarrow mi + 1\)  
14 return \(mi\)

\textit{find\_close}(i). We then calculate the last position of the block in which \(mp\) lies and scan from right to left for the first opening parenthesis \(mi\) which has excess value \(\text{excess}(i) + 1\) and therefore is the matching opening parenthesis to \(i\).

**Correctness** The proof is similar to the proof of the \textit{find\_close}(i) algorithm. In step \(\circ\) we get the position \(p'\) of the pioneer \(p > i\) in \textit{pioneer\_bp}. Note that such a pioneer \(p\) always exists, since \(i\) is a far parenthesis and would be itself a pioneer if no pioneer exists right to \(i\). Furthermore, \(p\) has to be a closing pioneer, for if \(p\) would be opening we would get the following contradictions:

1. If \(p = i\), \(p\) is opening but \(i\) is closing.
2. If \(p > i\) and \(p\) is an opening pioneer, \(i\) would be itself a closing pioneer and the minimal pioneer \(\geq i\).

The rest of the proof is straightforward.

**Runtime** Analogously to \textit{find\_close}(i), we get \(O(\log n / \log \log n)\) or constant time if we use 3 levels and store the answers for the last level explicitly.

**The enclose(i) Operation in Detail**

The code of the \textit{enclose}(i) operation, which is given in Algorithm 12, bears a striking similarity to the code of the \textit{find\_open} operation. This is due to the similar structure of the problems. While \textit{find\_open}(i) takes a closing parenthesis as argument and asks
Algorithm 11 Pseudo code for the $\text{find\_open}$ operation.

```plaintext
\begin{algorithmic}
\STATE $\text{find\_open}(bp, i, \text{block\_size})$
\IF {$bp[i] = \text{'('}$
\STATE \textbf{return} $i$
\ELSE
\STATE $mi \leftarrow \text{near\_find\_open}(bp, i, \text{block\_size})$
\IF {$mi < i$
\STATE \textbf{return} $mi$
\ELSE
\STATE $p' \leftarrow \text{rank1}(\text{pioneer\_bitmap}, i)$
\STATE $mp' \leftarrow \text{find\_open}(\text{pioneer\_bp}, p', \text{block\_size})$
\STATE $mp \leftarrow \text{select1}(\text{pioneer\_bitmap}, mp' + 1)$
\STATE $mi \leftarrow (\beta(mp) + 1) \cdot \text{block\_size} - 1$
\WHILE {$\text{excess}(mi) \neq \text{excess}(i) + 1 \text{ or } bp[mi] = \text{'('}$
\STATE $mi \leftarrow mi - 1$
\STATE \textbf{return} $mi$
\ENDWHILE
\ENDIF
\ENDIF
\ENDIF
\end{algorithmic}
```

for the maximal opening parenthesis, say $\ell$, with $\ell < i$ and $\text{excess}(\ell) = \text{excess}(i) + 1$, $\text{enclose}(i)$ takes an opening parenthesis as argument and asks for the maximal opening parenthesis, say $k$, with $k < i$ and $\text{excess}(k) + 1 = \text{excess}(i)$.

Examples Let us now explain by two examples how the algorithm works. In the first example we take the balanced parentheses sequence $bp = (((())())(()()))$ which is also depicted in Figure 3.12. We set $\text{block\_size} = 5$ and would like to know the answer for parenthesis $12$, i.e. $\text{enclose}(12)$. The algorithm first checks whether parenthesis $i = 12$ is closing. If it is, we have a special case, which will be explain later. As parenthesis 12 is opening, we check whether its excess value equals 1, if this is the case, $\text{enclose}$ returns $\perp$ because there is no opening parenthesis $k$ with $\text{excess}(k) = 0$. We get $\text{excess}(12) = 3 > 1$ and get to line 6 of the algorithm. The subroutine $\text{near\_enclose}$ scans the block of $i$ from position $i - 1 = 11$ to the left and returns the maximal opening parenthesis $k$ with $\text{excess}(k) + 1 = \text{excess}(i)$ in the scanned interval or $i$ if no such $k$ exists. In the example $\text{near\_enclose}$ returns 12, as there is no such parenthesis in the block (see ① in Figure 3.12). Therefore, we get to line 10, which calculates the position $p'$ of the pioneer $p \geq i$ in the recursive balanced parentheses sequence of the pioneers of $bp$. (This sequence, called $\text{pioneer\_bp}$ is depicted at the bottom of Figure 3.12.) Conceptionally, this calculation is a rank query on the $\text{pioneer\_bitmap}$. The next pioneer is at position 14 and the rank
query results in \( p' = 3 \). Now, the recursive call \( \textcircled{1} \) is answered by near_enclose and results in \( ep' = 0 \). A select query gives us the corresponding index \( ep \) in \( bp \) of parenthesis \( ep' \) in \( pioneer_bp \). We get parenthesis \( ep = 0 \). Note that this parenthesis pair \((0,19)\) already encloses parenthesis pair \((12,13)\). But to be sure that there is no parenthesis \( k > 0 \) in the block which also encloses pair \((12,13)\) we have to scan the block in which \( ep = 0 \) lies from right-to-left (see \( \textcircled{5} \)). In this example, this scan results in \( ei = 1 \) and therefore returns the pair \((1,18)\) which encloses \((12,13)\) most tightly.

The second example illustrates the above mentioned special case in which enclose equals \texttt{find_open}. We take the balanced parentheses sequence \( bp = (((())()))(())()) \) and \texttt{block_size} = 5.

This time we would like to know the answer to \texttt{enclose}(11). As \( i = 11 \) is an opening parenthesis and \( excess(i) = 4 \) > 1 we first scan for a near answer (see \( \textcircled{2} \) in Figure 3.13). Unfortunately, we get no valid answer and have to go to line 10. This time, the pioneer \( p = 14 \geq i \) is a closing parenthesis and we get its position \( p' = 4 \) in \( pioneer_bp \) by a rank query on the \texttt{pioneer_bitmap}. The recursive call yields a call of \texttt{find_open} and we get \( ep' = 1 \). We get the corresponding position \( ep = 1 \) in \( bp \) by a select query on the \texttt{pioneer_bitmap}. Again the pair \((1,14)\) encloses \((11,12)\). But as in the example before, the scan in block \( \beta(1) \) yields the pair \((4,13)\), which most tightly encloses \((11,12)\).

**Correctness of the \texttt{enclose} operation** We have to show that a call of \texttt{enclose}(i) for an opening parenthesis \( i \) results either in the maximal opening parenthesis, say \( k \), with \( k < i \) and \( excess(k) + 1 = excess(i) \) or \( \bot \) if no such \( k \) exists.

The test, if there exists an enclosing pair for \((i,\mu(i))\) is easy. The first parenthesis of the sequence is opening and has \( excess(0) = 1 \) and all excess values from 1 to \( excess(i) \) appear between position 0 and \( i \). Therefore, there always exists a \( k < i \) with \( excess(k) + 1 = excess(i) \) if \( excess(i) > 1 \).

If \( k \) lies in the same block \( \beta(i) \) as \( i \), \texttt{enclose} correctly returns \( k \) because at first the subroutine near_enclose searches for this \( k \). Otherwise, \( k \) lies in a block left to block \( \beta(i) \). Therefore, the pair \((k,\mu(k))\) is a far parenthesis as \( \mu(k) > \mu(i) \).

Now, let \( p \) be the minimal pioneer right to \( i \), i.e. \( p \geq i \). We continue with a case analysis:

**Case 1:** \( p \) is a closing pioneer. First, we notice that \( \mu(p) < i \), since otherwise \( \mu(p) \) lies in the interval \([\mu(i) + 1, p - 1]\) and \( p \) would not be the minimal pioneer right to \( i \). So \((\mu(p),p)\) encloses (but not necessarily most tightly) \((i,\mu(i))\). We will now show that \( k \) lies in the same block as \( \mu(p) \). First of all, \( k \) cannot lie in a block \( b \) with \( b < \beta(\mu(p)) \), as \( \mu(p) > k \) and \((\mu(p),p)\) encloses \((i,\mu(i))\) more tightly than \((k,\mu(k))\) does. Now suppose that \( k \) lies in a block \( b \) with \( \beta(\mu(p)) < b < \beta(i) \). As \( k \) encloses
(i, µ(i)) and is enclosed by (µ(p), p) it follows that µ(k) ∈ [µ(i) + 1, p − 1]. If k is a pioneer itself we get a contradiction to the assumption that p is the minimal pioneer right to i. Otherwise there exists a pioneer ˆp in b, and the pair (ˆp, µ(ˆp)) encloses (k, µ(k)) and is itself enclosed by (µ(p), p); i.e. µ(ˆp) ∈ [µ(k), p − 1] and we get a contradiction to the fact that p is the minimal pioneer right to i. So k lies in block β(µ(p)) and is found by the algorithm in step (i) which scans this block.

Case 2: p is an opening pioneer. We claim that

(a) excess(p) = excess(i) and
(b) the answer to p is the right answer to i.

If p = i, the claim holds. So now assume that p > µ(i). To prove (a), first assume that excess(p) < excess(i) = excess(µ(i)) + 1. This implies that there exists a minimal closing parenthesis j in the interval [µ(i) + 1..p − 1] with excess(j) < excess(p) and therefore µ(j) < i. So (µ(j), j) encloses (i, µ(i)) and j is a far parenthesis. If j is a pioneer we get a contradiction to the assumption that p is the minimal pioneer right to i. Otherwise if j is no pioneer, there exists a pioneer pair (p′, µ(p′)) that encloses (µ(j), j) more tightly. It is easy to see that µ(p′) ∈ [j + 1, p − 1] and thus we get again a contradiction to the assumption that p is the minimal pioneer right to i. Now assume that excess(p) > excess(i). It follows that there exists an maximal opening parenthesis p′ with excess(p′) = excess(p) − 1 in the interval [µ(i) + 1..p − 1]. This p′ encloses p and hence has to be a pioneer as well. This is a contradiction to the assumption that p is the minimal pioneer right to i.

To prove (b), let k′ be the answer to enclose(p). As all excess values of opening parentheses in the interval [µ(i) + 1..p − 1] are greater than excess(i) = excess(p) it follows that k′ < i, i.e. (k′, µ(k′)) also encloses (i, µ(i)). As all closing parentheses in the interval [µ(i) + 1..p − 1] are greater than excess(i − 1), the closing parenthesis for the answer k to enclose(i) is greater than µ(p) and therefore k = k′.

Time complexity The time complexity of the enclose(i) operation depends on the time for find_open, excess, rank, select and steps (i) and (ii). We have already shown that find_open, excess, rank, and select can be answered in constant time O(1). So again, we get a time complexity of O(log n/ log log n) and constant time, if we only take 3 levels and store the answers for enclose explicitly on the last level.

The rr_enclose Operation in Detail

The code for rr_enclose is given in Algorithm 13. Recall that the range restricted enclose method takes two opening parentheses i, j (i < j) and calculates the minimal opening parentheses k ∈ [µ(i) + 1, j − 1] such that (k, µ(k)) encloses (j, µ(j)) or ⊥ if no such parentheses pair exists. We get a nice recursive formulation of the method, if we change the parameters slightly. The min_excess_position method takes two (not necessarily opening)
Algorithm 12 Pseudo code for the \textit{enclose}(i) operation.

\begin{verbatim}
enclose(bp, i, block_size)
01 if bp[i] = ')' return find_open(i) \footnote{1}
02 else if excess(i) = 1 return ⊥ \footnote{2}
03 else ei ← near_enclose(bp, i, block_size) \footnote{3}
04 if ei < i // true, if we have found a near answer
05 return ei
06 else p′ ← rank1(pioneer_bitmap, i) \footnote{4}
07 ep′ ← enclose(pioneer_bp, p′, block_size) \footnote{5}
08 ep ← select1(pioneer_bitmap, ep′ + 1) \footnote{6}
09 ei ← (β(ep) + 1) · block_size − 1
10 while excess(ei) + 1 ≠ excess(i) or bp[ei] = ')' return ei
11 end while
12 ei ← ei − 1
13 return ei
\end{verbatim}

parentheses $ℓ, r$ ($ℓ < r$) and calculates the minimal opening parenthesis $k \in [ℓ, r - 1]$ with $μ(k) ≥ r$. If no $k \in [ℓ, r - 1]$ exists, $⊥$ is returned. See Algorithm 14 for the code of this method.

Algorithm 13 Pseudo code for the \textit{rr_enclose} operation.

\begin{verbatim}
rr_enclose(bp, i, j, block_size)
01 return min_excess_position(bp, find_close(i) + 1, j, block_size)
\end{verbatim}

Example Let us now explain the algorithm for \textit{rr_enclose} and \textit{min_excess_position} respectively by an example. We take \(bp = (())(((()())((())(((())()()())))))\), a block size of 5, and we would like to know the answer to \(rr enclose(bp, 1, 28, 5)\). The balanced parentheses, its pioneers, and the recursive constructed balanced parentheses sequence of pioneers is depicted in Figure 3.14. In the example, the \textit{rr_enclose} operation calculates the matching closing parentheses $μ(i)$ for $i$ and calls \textit{min_excess_position}(bp, $ℓ = μ(i) + 1, r = j, 5$) with $ℓ = 3$ and $r = 28$. \textit{min_excess_position} first tests whether the interval $[ℓ, r - 1]$ is empty, and therefore no solution exists. If so, it returns $⊥$ to handle this case. Otherwise, we initialize the solution $k$ with $⊥$. Next, we test in line 4 whether $ℓ$ and $r$ lie in the same block. If so, we call a subroutine \textit{near min_excess_position}, which calculates the result of \textit{min_excess_position} for one block. In our example, $β(3) = 0 ≠ 5 = β(28)$ and so we have
Algorithm 14 Pseudo code for the \texttt{min\_excess\_position}-operation.

\begin{verbatim}
min\_excess\_position(bp, \ell, r, block\_size)
01    \textbf{if} \ \ell \geq r
02        \textbf{return} \ \bot
03    k \leftarrow \bot
04    \textbf{if} \ \beta(\ell) = \beta(r)
05        k \leftarrow \text{near\_min\_excess\_position}(bp, \ell, r, block\_size)\footnote{1}
06    \textbf{else}
07        \texttt{min\_ex} \leftarrow \text{excess}(r) + 2 \cdot (bp[r] = '\)')\footnote{2}
08        p'_{\ell} \leftarrow \text{rank1}(pioneer\_bitmap, \ell) \footnote{3}
09        p'_{r} \leftarrow \text{rank1}(pioneer\_bitmap, r)
10    k' \leftarrow \text{min\_excess\_position}(pioneer\_bp, p'_{\ell}, p'_{r}, block\_size)\footnote{4}
11    \textbf{if} \ k' \neq \bot
12        k \leftarrow \text{select1}(pioneer\_bitmap, k' + 1)\footnote{4'}
13        \texttt{min\_ex} \leftarrow \text{excess}(k)
14    \textbf{else}
15        k' \leftarrow \text{near\_min\_excess\_position}(bp, \beta(r) \cdot block\_size, r, block\_size)
16        \textbf{if} \ k' \neq \bot
17            k \leftarrow k'
18        \texttt{min\_ex} \leftarrow \text{excess}(k)
19    k' \leftarrow \text{near\_min\_excess\_position}(bp, \ell, (\beta(\ell)+1) \cdot block\_size, block\_size)
20    \textbf{if} \ k' \neq \bot \textbf{and} \text{excess}(k') < \texttt{min\_ex}
21        k \leftarrow k'
22        \texttt{min\_ex} \leftarrow \text{excess}(k)
23    \textbf{return} \ k
\end{verbatim}

to search for the answer in block $\beta(28) = 5$ (see step \footnote{2}), in block $\beta(3) = 1$ (see step \footnote{1}) or in between these blocks (see steps \footnote{3}-\footnote{4}). We first calculate the value \texttt{min\_excess} which equals $\text{excess}(28) + 0 = 5$ in our example. The excess value of a valid solution has to be lower than \texttt{min\_excess}. After that, we calculate with a rank query in step \footnote{3} the position $p'_{\ell}$ ($p'_{r}$) of the minimal pioneer $p_{\ell} \geq \ell$ ($p_{r} \geq r$) in pioneer\_bp. We get $p'_{\ell} = 1$ and $p'_{r} = 8$ and solve the problem recursively on pioneer\_bp which results in $k' = 3$. Since $k' \neq \bot$, we get the position $k$ of the pioneer in $bp$ by a select query in step \footnote{4'}. So $k = 13$, and \texttt{min\_excess} is set to $\text{excess}(13) = 4$.

As we get a solution from the set of pioneers, we do not have to execute step \footnote{5} because the solution from the pioneer would enclose the solution found in block 5. So, at last, we have to check the block 0 in step \footnote{6}. The \texttt{near\_min\_excess\_position} method returns $k' = 3$, and, as $\text{excess}(3) = 2 < 4 = \text{excess}(13)$, we set $k = 3$. 

Correctness of the \textit{min\_excess\_position} method \quad To prove the correctness, we have to show that a call of \textit{min\_excess\_position}(bp, ℓ, r, block\_size) results in the minimal parenthesis $k \in [\ell, r - 1]$ with $\mu(k) \geq r$ or $\perp$ if no such $k$ exists. We will prove this by induction on the length $n$ of the balanced parentheses sequence $bp$.

If $n \leq \text{block\_size}$ then it follows that $\ell$ and $r$ lie in the same block and the subroutine \textit{near\_min\_excess\_position} will find the correct answer by either scanning the block for the answer or making a look-up in precomputed tables of size $O(\text{block\_size}^2 \cdot 2^{\text{block\_size}})$.

Otherwise $n > \text{block\_size}$ and $\beta(\ell) \neq \beta(r)$. Let $\hat{k}$ be the right solution for the call of \textit{min\_excess\_position}(bp, ℓ, r, block\_size). We divide the interval $I = [\ell, r - 1]$ in three subintervals $I_\ell = [\ell, (\beta(\ell) + 1) \cdot \text{block\_size} - 1]$, $I_m = [(\beta(r) + 1) \cdot \text{block\_size}, \beta(\ell) \cdot \text{block\_size} - 1]$, and $I_r = [\beta(\ell) \cdot \text{block\_size}, r - 1]$; see Figure 3.15.

We will show that if $\hat{k}$ lies in $I_m$, it follows that $\hat{k}$ is a pioneer.

To prove that, suppose that $\hat{k}$ is the right solution, i.e. $\mu(\hat{k}) \geq r$ and $\hat{k}$ is minimal in $I_m$ but no pioneer. As $\hat{k} \in I_m$ and $\mu(\hat{k}) \geq r \in I_r$, it follows that $\hat{k}$ has to be a far parenthesis. Since $\hat{k}$ is a far parenthesis but no pioneer, there has to be an pioneer $k < \hat{k}$ in the same block. Consequently, $(\hat{k}, \mu(\hat{k}))$ encloses $(\hat{k}, \mu(\hat{k}))$. This is a contradiction to the assumption that $\hat{k}$ is the right solution.

We know by induction, that if $k' \neq \perp$ after step (4), the corresponding pioneer $k$ in $bp$ (calculated by a select query in step (4)) lies in $I$ and $\mu(k) \geq p'_r \geq r$.

The scan in the interval $I_\ell$ (see step (2)) for a solution is unnecessary if a pioneer $k$ is found in step (2). To show this, let $k_r \in I_r$ be the solution from step (5). If $k_r$ is not a pioneer, $(k_r, \mu(k_r))$ is enclosed by $(k, \mu(k))$, otherwise $k_r$ is a pioneer and so was already considered in step (3).

At last, we have to check the interval $I_\ell$ for the solution $\hat{k}$, because if $\hat{k}$ lies in $I_\ell$ it does not necessarily follow that $\hat{k}$ is a pioneer. This is due to the fact that there can be a pioneer in $\beta(\hat{k}) < \ell$. The example in Figure 3.14 shows such a situation.

- $\text{bp} = \begin{array}{c}
\text{excess}(\text{bp}) = 12123 43454 54345 45454 56565 45454 3210
\end{array}$

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{example}
\caption{Example for the calculation of \texttt{rr\_enclose(bp, 1, 28, 5)} and \texttt{min\_excess\_position(bp, 3, 28, 5)} respectively.}
\end{figure}
Figure 3.15: Proof of the correctness of the \textit{min\_excess\_position} method: Split the interval between $i$ and $j$ into three subintervals.

Time and space complexity  
Like in the previously considered methods, all operations in the algorithm are constant time operations. It follows, that we get a time complexity of $O(\log n / \log \log n)$ for the simple structure with $O(\log n / \log \log n)$ levels and constant time if we take two levels and store the answers for the third level explicitly.

This time, we have to take care about the occupied space in the latter case, as there are $O(n^2)$ possible queries on the parentheses sequence on the third level of size $n^2$. Since $n^2 \in O\left(\frac{n^2}{\log n}\right)$ we would occupy $O\left(\frac{n^2}{\log n}\right) \cdot \log n^2 = O\left(\frac{n^2}{\log n}\right) \cdot \log n^2$ bits of space, which is clearly not in $o(n)$. To get a succinct solution, we use the already stored answers for $\mu$ in the second level and a data structure that supports constant time range maximum queries (RMQ) on that array. Note that the answer for \textit{min\_excess\_position}($bp, \ell, r$) is equal to the parenthesis $k \in [\ell..r-1]$ with maximal matching parenthesis $\mu(k)$: $k = \arg \max_{i \in [\ell..r-1]} \{\mu(i) \mid \mu(i) \geq r\}$. We set $k = \bot$ if the set is empty. There are constant time solutions for RMQ which occupy $O(n_2 \log n_2)$ or $O(n_2)$ words (see e.g. [BFC00]). The second one gives us a $O(n_2 \log n_2) = O\left(\frac{n}{\log \log n}\right) \cdot \log n_2 = O\left(\frac{n}{\log \log n}\right)$ bits solution. Therefore, we need only $o(n)$ bits to support constant time queries on the third level.

The \textit{double\_enclose} Operation

We can express the \textit{double\_enclose} operation with a combination of $rr\_enclose$ and $enclose$. See Algorithm 15 for the detailed solution. The proof of the correctness is easy and therefore left out here. Since the \textit{double\_enclose} operation in the \textit{BPS}$_{dfs}$ corresponds to the lowest common ancestor (lca) in the corresponding tree we now support the whole set of navigational operations in a CST. Furthermore the \textbf{bp\_support} classes can be used to easily create a succinct data structure for range minimum queries on an array. We refer the interested reader to Section 6.2.

3.8.2 The Range Min-Max-Tree

The \textit{range-min-max tree} [SN10] is also a hierarchical data structure. Figure 3.16 depicts its structures. It is built over the virtual array of excess values of the balanced parentheses sequence as follows: First the excess array is grouped into $\lceil n/s \rceil$ small blocks of size $s = \frac{\log n}{2}$. The minimum and maximum value of each small block is stored in a corresponding leaf node of the min-max-tree, which forms a complete $k$-ary tree. Each inner node of
Algorithm 15 Pseudo code for the double_enclose operation.

\begin{verbatim}
  double_enclose(bp, i, j, block_size)
  01  k = rr_enclose(bp,i,j,block_size)
  02  if k ≠ ⊥
  03    return enclose(bp, k, block_size)
  04  else
  05    return enclose(bp, j, block_size)
\end{verbatim}

the min-max-tree contains the minimum (maximum) of the minima (maxima) of its child nodes. To get a theoretical \(o(n)\) space complexity the minimum and maximum value are relative to the excess value at the beginning of each block. Therefore each of the \(O(n/s)\) nodes of the tree takes \(2 \log s\) bits and in total we spent \(O(n \log \frac{k}{2}) = o(n)\) bits. In practice, we can take absolute minimum and maximum values of \(\log n\) bits, since it turns out that \(s\) can be chosen much larger than \(\frac{\log n}{2}\) without increasing the runtime of the operations.

All operations of the \texttt{bp\_support} concept are reduced to the following two operations

\[
\text{fwd\_excess}(i, d) = \min\{j \mid j > i \land \text{excess}(j) = \text{excess}(i) + d\}
\]

\[
\text{bwd\_excess}(i, d) = \max\{j \mid j < i \land \text{excess}(j) = \text{excess}(i) + d\}
\]

which search for the leftmost (rightmost) position \(j > i \ (j < i)\), such that \(\text{excess}(i) = \text{excess}(j) + d\) for an integer \(d\). Note that both operations can be trivially answered in constant time, when \(i\) and the corresponding answer lie in the same block \(bk\), since we can use a lookup table of size \(o(n)\). Otherwise, things are more complicated. Roughly spoken, we have (1) to go up in the range min-max-tree until an inner node encloses the desired excess value \(\text{excess}(i) + d\) in its minimum and maximum label, and (2) following the leftmost (rightmost) path down which leads to a block which is right (left) to \(bk\) and all nodes in the path enclose the desired excess value. We refer the interested reader to the implementation of the class \texttt{bp\_support\_sada} for details or to the original paper [SN10].

The point is that the operations now take \(O(\log_k n)\) time, even when each operation at a node is performed with lookup tables in constant time.

Sadakane and Navarro [SN10] solved this problem as follows. For trees of “moderate” size \(n = \text{polylog}(\log n)\) the depth is constant if we chose \(k \in \Theta(\frac{\log n}{\log \log n})\). They also present a solution for larger \(n\) which also provides constant time access by using \(O(n/\text{polylog}(n))\) space. But the solution is very complex and was not implementable in reasonable time. The implemented version for the “moderate” size uses a degree of \(k = 32\) and therefore the tree has in practice an almost constant depth.

It remains to show how the main operations of the \texttt{bp\_support} concept were reduced
Figure 3.16: Example of the range min-max-tree. At the bottom: $BPS_{dfs}$ of the suffix tree of our running example colored in blue. The excess value for each position $i$ in $BPS_{dfs}$ is shown above parenthesis $i$. The excess values are grouped into small blocks of size $s = 7$. Each leaf of the range min-max-tree contains the minimum and maximum of the corresponding small block. The degree $k$ of the range min-max-tree equals $k = 3$ in our example. Each inner node contains the minimum (maximum) of the minima (maxima) of its child nodes.

to the base operations:

$$
\begin{align*}
\text{find\_close}(i) & \equiv \text{fwd\_excess}(i, -1) \\
\text{find\_open}(i) & \equiv \text{bwd\_excess}(i, 0) + 1 \\
\text{enclose}(i) & \equiv \text{bwd\_excess}(i, -2) + 1 
\end{align*}
$$

The remaining operations $rr\_enclose$ and $double\_enclose$ are more complex and cannot be expressed with $fwd\_excess$ or $bwd\_excess$ and we refer the interested reader another time to the original paper [SN10] or to our implementation for details.

### 3.8.3 Experimental Comparison of $bp\_support$ Implementations

In the $sdsl$ we provide three classes for the $bp\_support$ concept:

- **$bp\_support\_g$**: Corresponds to the new data structure with only two levels of recursion. All answers to $find\_close$, $find\_open$, and $enclose$ of the third level are stored uncompressed. The answers to $rr\_enclose$ are calculated with the RMQ data structure of Bender and Farach-Colton [BFC00] (called $rmq\_support\_sparse\_table$ in the $sdsl$).

- **$bp\_support\_gg$**: Corresponds to the new data structure whereas the recursion stops if the number of parenthesis is smaller than the chosen block size.

- **$bp\_support\_sada$**: Our implementation of the range min-max-tree proposed in [SN10].

All three implementations can be parametrized with the data types which should be used to realize the $rank$ and $select$ operations. In the experiments we have parametrized all
data types with the same structures to get a fair comparison. Furthermore, we use the same lookup tables and the same scanning routines for the near answers.

\texttt{bp\_support\_g} and \texttt{bp\_support\_gg} can be also parametrized with a block size, and \texttt{bp\_support\_sada} with an additional parameter medium block size, which corresponds to the degree $k$ of the range min-max-tree. In the experiments we used block sizes of $2^4, 2^5, \ldots, 2^{12}$ and for \texttt{bp\_support\_sada} a fixed medium block size of $k = 32$ to get a time-space trade-off for the different operations.

Figures 3.17, 3.18, 3.19 and 3.21 show the results of our experiments. We have depicted the memory consumption of the \texttt{bp\_support} data structure per parenthesis of the input without considering the memory for the used \texttt{rank\_support} and \texttt{select\_support}. The inputs are balanced parentheses sequences of the suffix tree topology of the \textit{Pizza&BChili} corpus. For each operation we have chosen $2^{20}$ random queries in advance and then repeatedly queried the data structure until we have answered $10^7$ queries in total. The average runtime is then plotted in the figures.

In Figure 3.17 we have depicted the results for the \texttt{find\_close} operation. For all test cases the best runtime of all three implementations lie between 100 and 200 nanoseconds, while using less then 0.1 bits per parenthesis. The fact that the runtime and the space is decreasing for all implementations can be contributed to the many near answers which can be answered quickly by simply scanning to the right. Smaller block sizes and therefore a higher memory consumption yield a slower running times since the navigation in the hierarchical structures is expensive compared to scanning which has a good locality of references.

The results for the \texttt{find\_open} operation (see Figure 3.18) look very similar. This is not surprising since (1) we used as input the matching closing parentheses of the \texttt{find\_close} input and (2) the only difference between \texttt{find\_open} and \texttt{find\_close} is the scanning direction. This time its from right to left.

The results for the \texttt{enclose} operations are depicted in Figures 3.19 and 3.20. The results of Figure 3.19 look very similar to those of the \texttt{find\_open} operation. This can be explained as follows. We use random opening parentheses as input, and therefore many near answers can be answered by only scanning to the left without navigate in the hierarchical structure. In Figure 3.20 we start at a random opening parentheses and call \texttt{enclose} as long as \texttt{enclose} does not return 0, i.e. this can be interpreted as calling the \texttt{parent}(v) until we are at the root of the corresponding tree. Therefore, we get more far answers which require the navigation in the hierarchical structures. So the best running time is about 600-800 nanoseconds which is reached by using about 0.1 – 0.2 bits per character for \texttt{bp\_support\_gg} and \texttt{bp\_support\_sada}. The minimum runtime of \texttt{bp\_support\_g} is reached by more than 1 bit per input character. For smaller memory consumptions the running time increases fast and therefore not all time-space trade-offs are depicted in Figure 3.20.

Finally, we results for the \texttt{double\_enclose} operation are depicted in Figure 3.21. We choose pairs of random opening parentheses as input and therefore get another time many far answers. This time, \texttt{bp\_support\_g} provides the fastest runtime. This is due to the use of the \texttt{RMQ} structure \texttt{bp\_support\_sparse\_table} on the third recursion level. However the time-space trade-off of \texttt{bp\_support\_gg} and \texttt{bp\_support\_sada} for very little
space (< 0.1 bits per parenthesis) is better than that of \texttt{bp\_support\_g}.

Conclusion:

- The time-space trade-off of a \texttt{bp\_support} depends on the chosen block size. We recommend to use block sizes of several hundreds to get a good time-space trade-off.

- \texttt{bp\_support\_gg} and \texttt{bp\_support\_sada} show similar results, where the runtime of \texttt{bp\_support\_sada} is always slightly better. Since we provide also a very fast construction of \texttt{bp\_support\_sada}, we will use it as default \texttt{bp\_support} type.

- \texttt{bp\_support\_g} is the right solution when we have enough main memory and need faster answers for the \texttt{double\_enclose} operation.
Figure 3.17: Experimental results for the \textit{find_close} operation of three \texttt{bp_support} data structures on randomly chosen opening parentheses.
Figure 3.18: Experimental results for the \textit{find open} operation of three \texttt{bp_support} data structures on randomly chosen closing parentheses.
Figure 3.19: Experimental results for the \textit{enclose} operation of three \texttt{bp\_support} data structures on randomly chosen opening parentheses.
Figure 3.20: Experimental results for the *enclose* operation of three *bp_support* data structures.
3.8 The bp_support Concept

Figure 3.21: Experimental results for the double_enclose operation of three bp_support data structures.
3.9 The cst Concept

We have already introduced the operations of the cst concept in Table 3.1. In this section we will focus on the representation of the NAV structure which distinguishes the different CST proposals.

3.9.1 Sadakane’s CST Revisited

Munro et al. [MRR01] and Sadakane [Sad07a] proposed to use the balanced parentheses sequence $BPS_{dfs}$ (see page 54) of the suffix tree to represent its topology in the CST. A node of the CST is represented by the position of the opening parenthesis of the corresponding parentheses pair, i.e. in the rest of this section a node $v$ corresponds to an integer in the range from 0 to $size() - 1$ and in the implementation the node_type of class cst_sada corresponds to an unsigned integer. In this representation the root node $w = \text{root}()$ equals 0. Most tree operations can now be solved easily. For instance the \textit{is leaf}($v$) operation: A node $v$ is a leaf if it has no children and therefore is immediately followed by a closing parenthesis, i.e. checking whether $BPS_{dfs}[v+1] = 0$ answers the query. For other operations we have to add supporting data structures:

- A \textbf{select support} on the pattern “10” is required to answer $ith\_leaf(i)$ as it holds that $ith\_leaf(i) = select_{BPS_{dfs}}(i, 10)$.

- A \textbf{bp support}, which includes a \textbf{rank support} and \textbf{select support} for the pattern “1” is required by the operations \textit{node depth}($v$), \textit{leaves in the subtree}($v$), \textit{rightmost leaf in the subtree}($v$), \textit{leftmost leaf in the subtree}($v$), \textit{rb}($v$), \textit{parent}($v$), \textit{sibling}($v$), \textit{child}($v$, $c$), $ith\_child(v, i)$, \textit{lca}($v$, $w$), \textit{sl}($v$), \textit{wl}($v$, $c$), \textit{id}($v$), \textit{degree}($v$), and \textit{tlcp idx}($i$). With a \textbf{bp support} data structure the implementation of these operations is straightforward, e.g. the \textit{lb}($v$) operation. Here we have to count all leaf nodes which were already visited in the depth-first traversal before we process node $v$. This can be translated in $lb(v) = rank_{BPS_{dfs}}(v, 10)$. The \textit{rb}($v$) operation counts all leaf nodes that are visited in the depth first traversal until all nodes of the subtree of $v$ are processed, i.e. $rb(v) = rank_{BPS_{dfs}}(find\_close_{BPS_{dfs}}(v), 10)$. The answer of \textit{leaves in the subtree}($v$) equals the difference of $rb$ and $lb$.

Figure 3.22 depicts the dependencies of Sadakane’s proposal from other data structures. The main difference to the following proposals is the \textit{rank} and \textit{select} data structure for the bit pattern “10”.

3.9 The cst Concept

Figure 3.22: One possible configuration for cst_sada. Note that cst_sada has a template parameter for a rank support and a select support. The first one determines the concrete type for the rank data structure for the bit pattern “10” and the second one the concrete type for the select data structure for the bit pattern “10”. This is the main difference to other CSTs, cf. Figure 3.2 of the CST which we will present in Section 3.9.3.
3.9.2 Fischer et al.’s Solution Revisited

In Fischer et al.’s CST [FMN09] the navigation is based on (1) the representation of nodes as intervals (like in the LCP-interval tree presented in Section 2.8) and (2) the use of NSV, PSV, and RMQ queries on the LCP-array to answer navigational queries. For the NSV, PSV, and RMQ queries they introduce a data structure which takes sub-linear space and solves the queries in sublogarithmic time. However, the query time also depends on the time complexity of the \([\cdot]\)-operator of the used LCP-array. Therefore, from the practical point of view the suggested entropy compressed CST configuration, which uses an entropy compressed LCP-array (see page 90) which was also suggested in the paper, was no improvement to the CST of Russo et al. [RNO08]. To achieve a better runtime in practice, we adopted the suggested theoretic framework which use NSV, PSV, and RMQ to answer the navigational operations and focused on designing a data structure which use little (but not sub-linear) space in practice and answers the NSV, PSV, and RMQ queries faster. It is also quite interesting that in some cases we actually do not need a RMQ to answer operations.

3.9.3 A Succinct Representation of the LCP-Interval Tree

In Section 2.8 we have presented the LCP-interval tree and how to calculate parent\((v)\), sibling\((v)\), and \(i\)th child\((v, i)\) in terms of RMQ, NSV, and PSV. Now we will show how to calculate the operations using only \(2n + o(n)\) or \(3n + o(n)\) bits of space. The first result was published in [OG09] and the second in [OFG10]. The main tool is a sequence of balanced parentheses which is generated from the LCP array and which we will call \(\text{BPS}_{sct}\). In [OG09] we have shown how \(\text{BPS}_{sct}\) is related to the Super-Cartesian tree of the LCP array and therefore it has the subscript \(sct\). Since we can prove the correctness of the solution for the operations NSV, PSV, . . . by arguing only on \(\text{BPS}_{sct}\) we will omit the presentation of the Super-Cartesian tree here. We will define the \(\text{BPS}_{sct}\) by its construction.

Construction of \(\text{BPS}_{sct}\)

The \(\text{BPS}_{sct}\) of an LCP array of size \(n\) consists of exactly \(2n\) parentheses. Every element is represented by an opening parenthesis and a corresponding closing parenthesis. Algorithm 16 shows pseudo-code for the construction. We first initialize \(\text{BPS}_{sct}\) in line 00 to a sequence of \(2n\) closing parentheses. On the next line we set the current position in \(\text{BPS}_{sct}\) to 0 and initialize a stack \(s\). Next we scan the LCP array from left to right. For every element \(\text{LCP}[i]\) we check if there exists previous elements on the stack \(s\) which are greater than \(\text{LCP}[i]\). If this is the case we pop them from the stack and increment the current position in \(\text{BPS}_{sct}\). As we have initialized \(\text{BPS}_{sct}\) with closing parentheses this action equals of writing a closing parentheses at position \(ipos\). After the while loop the stack contains only positions of elements which are smaller or equal than \(\text{LCP}[i]\). We write an opening parenthesis for element \(\text{LCP}[i]\) at position \(ipos\) and push the position of \(\text{LCP}[i]\) onto the stack. Note that the produced balanced parentheses sequence is balanced as we only write closing parentheses for elements which where pushed onto the stack and we
have therefore written an opening parenthesis before we write the matching closing one. Also all elements which remain on the stack after the n-th iteration have corresponding closing parentheses, since we have initialized \( BPS_{sect} \) with closing parentheses. The whole construction algorithm has a linear running time since in total we cannot enter the while loop more than \( n \) times. The \( BPS_{sect} \) for the LCP-array of our running example is depicted in Figure 3.23.

Note that Fischer’s 2d-min-heap data structure presented in [Fis10a] is very similar to \( BPS_{sect} \). In [OFG10] we have shown that they are essentially the reversed string of each other. Therefore the construction algorithm is also essentially the same. Therefore, we can use Fischer’s result to get a very space efficient construction of the \( BPS_{sect} \).

The elements on the stack are at any time a strictly increasing sequence of integers in the range from \(-1\) to \( n-1 \). Fischer showed in [Fis10a] that such a stack which supports the \( \text{top}() \), \( \text{pop}() \), and \( \text{push}(i) \) operations in constant time can be realized by only using \( n + o(n) \) bits, where the \( o(n) \) term is due to \text{rank support} and \text{select support} data structures. We will present an advanced solution\(^1\) which gets rid of the \text{rank support} and \text{select support} structure and is therefore faster in practice. In the following we denote the word width by \( w = \lceil \log n \rceil \). First, we initialize a \text{bit_vector} \( bv \) which contains \( n+1 \) bits and a \text{bit_vector} \( \text{far} \) which contains \( n+1 \) \( w \) bits. The \( \text{push}(i) \) operation is then realized by setting bit \( bv[i+1] \) to 1. To realize the \( \text{top}() \) operation we introduce a pointer \( p \), which contains a pointer to the rightmost set bit in \( bv \). Each time we do a \( \text{push}(i) \) operation, we set \( p \) to \( i+1 \). But before doing that we keep track of the \( p \) value if the difference of \( i+1 - p \) is large. We divide \( bv \) into blocks of size \( w \) bits. If \( p \) is not contained in the block of \( i+1 \) and not in the block \( bl \) left to the block of \( i+1 \), i.e. \( p < w \cdot (i+1) - 1 \), then we set the \( \text{far} \) bit of \( bl \): \( \text{far}[i+1] = 1 \). Note that \( bl \) does not contain any set bit. Therefore we store \( p \) in \( bl \). Finally, the \( \text{pop}() \) operation is realized as follows. We delete the topmost set bit \( bv[p] = 0 \). Next we check in constant time if there exists a set bit in the block which contains \( p \). If there exists a set bit, we set \( p \) to the rightmost set bit. Otherwise, we check if the block \( bl \) left to the block containing \( p \) contains the pointer to the next smaller element on the stack by

![Figure 3.23](image)

**Figure 3.23:** The LCP-array of our running example and the corresponding balanced parentheses sequence \( BPS_{sect} \). We have added arrows. Each arrow points from a closing parenthesis to its corresponding opening one. Note that we have placed the \( i \)-th opening parentheses directly above its corresponding LCP entry \( \text{LCP}[i] \).

\(^1\) It is called \text{sorted_stack_support} in the \text{sdsl}.
Algorithm 16 Construction of $\text{BPS}_{\text{act}}$ of the LCP array.

\begin{verbatim}
00  BPS_{act}[0..2n-1] ← [0,0,\ldots,0]
01  ipos ← 0
02  s ← stack()
03  for i ← 0 to n − 1 do
04      while s.size() ≥ 0 and LCP[s.top()] > LCP[i] do
05          s.pop()
06      ipos ← ipos + 1
07  BPS_{act}[ipos] ← 1
08  s.push(i)
09  ipos ← ipos + 1
\end{verbatim}

checking if $\text{far}[\frac{p}{2} - 1] = 1$. When this is the case, we set $\text{far}[\frac{p}{2} - 1] = 0$ and $p$ to the content of $bl$. Otherwise the new $p$ value is contained in $bl$ and we set $p$ to the rightmost set bit of $bl$. Note that all operations can be performed in constant time. We would like to mention that finding the rightmost bit\footnote{In the \textit{sdsl} the operation is called \texttt{bit\_magic::r1BP(x)}.} in a word can be solved very elegantly by a trick which uses a multiplication and a De Bruijn sequence; for details see [Knu08].

Mapping between LCP and $\text{BPS}_{\text{act}}$

After the construction we can map $\text{LCP}[i]$ to the position of an opening parenthesis by $\text{ipos = select}_{\text{BPS}_{\text{act}}}(i + 1)$. It is also possible to map an opening parenthesis $\text{ipos}$ to its corresponding element $\text{LCP}[i]$ by $i = \text{rank}_{\text{BPS}_{\text{act}}}(\text{ipos})$. Note that we have defined $\text{rank}(\text{ipos})$ to return the number of ones up to but not including $\text{ipos}$. As indexing in $\text{LCP}$ starts at 0 we get the right answer.

Calculating $\text{NSV}[i]$\footnote{In the \textit{sdsl} the operation is called \texttt{bit\_magic::r1BP(x)}.}

To calculate the next smaller value $\text{NSV}[i]$ we first map the element $\text{LCP}[i]$ to $\text{ipos}$ in $\text{BPS}_{\text{act}}$. In the construction algorithm of $\text{BPS}_{\text{act}}$ the corresponding closing parenthesis of $\text{ipos}$ is not written until we reach the next element $\text{LCP}[j]$ with $j > i$ and $\text{LCP}[j] < \text{LCP}[i]$, i.e. the next smaller value! Therefore, the corresponding opening parenthesis $\text{ipos}$ of $\text{LCP}[j]$ is the first opening parenthesis which follows the closing parenthesis of $\text{LCP}[i]$ at position $\text{cipos} = \text{find\_close}_{\text{BPS}_{\text{act}}}(\text{ipos})$. Since there are only closing parentheses between $\text{cipos}$ and $\text{ipos}$ a call of $\text{rank}_{\text{BPS}_{\text{act}}}(\text{cipos})$ returns the index $\text{NSV}[i] = j$. Pseudo-code is given in Algorithm 17.
To calculate the previous smaller value \( PSV[i] \) we again first map the element \( LCP[i] \) to \( ipos \) in \( BPS_{act} \). The opening parenthesis \( jpos \) of a previous element \( LCP[j] \) with \( j < i \) has to be left of \( ipos \). The closing parenthesis \( cjpos \) of a previous element \( LCP[j] \) which is smaller than or equal to \( LCP[i] \) has to be right of \( cipos \), since we close only parentheses of elements which are greater than \( LCP[i] \) before \( cipos \). Therefore the parentheses pair of a previous smaller value enclous the pair \((ipos, cipos)\). The rightmost \( j \) with \( j < i \) and \( LCP[j] \leq LCP[i] \) is represented by the pair \((jpos, cjpos)\) which encloses \((ipos, cipos)\) most tightly. Fortunately we have the \( enclose \) operation, which will return \( jpos \). Unfortunately, the corresponding \( LCP[j_0 = j] \) entry is only the previous smaller or equal value, and not the previous smaller value. Therefore, if \( LCP[j_0] \) is not a smaller value we have to repeatedly call \( enclose \) until we get a \( LCP[j_k] \) which is smaller than \( LCP[i] \). See the while-loop in Algorithm 18. This algorithm has a running time which is linear in the number of equal LCP entries between \( i \) and \( PSV[i] \). In the LCP array this number is bounded by the alphabet size \( \sigma \), since each LCP value is also an \( \ell \)-index in the LCP-interval tree and there exists at most \( \sigma - 1 \) \( \ell \)-indices in one interval. Therefore, this implementation of \( PSV[i] \) has a time complexity of \( O(\sigma \cdot t_{LCP}) \), where \( t_{LCP} \) equals the time for an access to an LCP entry.

**Algorithm 18** Calculating \( PSV[i] \) with \( BPS_{act} \) in linear time.

\[
\begin{align*}
00 & \quad ipos \leftarrow \text{select}_{BPS_{act}}(i + 1) \\
01 & \quad jpos \leftarrow \text{enclose}_{BPS_{act}}(ipos) \\
02 & \quad j \leftarrow \text{rank}_{BPS_{act}}(jpos) \\
03 & \quad \textbf{while } LCP[j] = LCP[i] \textbf{ do} \\
04 & \quad \quad jpos \leftarrow \text{enclose}_{BPS_{act}}(jpos) \\
05 & \quad \quad j \leftarrow \text{rank}_{BPS_{act}}(jpos) \\
06 & \quad \textbf{return } j
\end{align*}
\]

A crucial observation is that closing parentheses of the equal \( LCP[j_t] \) entries \( (0 \leq t < k) \) form a contiguous sequence of closing parentheses, since in the construction they are all removed from the stack in the same execution of the while loop in Algorithm 16. Therefore we can also implement the \( PSV \) operation by a binary search on the closing parentheses and get a time complexity of \( O(\log \sigma \cdot t_{LCP}) \).

We will now show how we can achieve a constant time solution for the \( PSV[i] \) operation
and how we get rid of the $t_{LCP}$ term in the time complexity. We add another bit_vector of length $n$ which we call $rc$ (abbreviation for rightmost closing) to our solution. The entry $rc[k]$ indicates whether the parenthesis at position $cipos = select_{BPS_{set}}(k + 1, 0)$

(1) is not followed by another closing parenthesis at position $cipos = cipos + 1$, i.e. $cipos \geq 2n$ or $BPS_{set}[cipos + 1] = 1$, or

(2) it is followed by another closing parenthesis at position $cipos$ and the corresponding LCP values at position $i = rank_{BPS_{set}}(find_{open}BPS_{set}(cipos))$ and $j = rank_{BPS_{set}}(find_{open}BPS_{set}(cipos))$ are different.

Note that we can construct $rc$ in linear time by slightly modifying Algorithm 16. First, we add a counter for the number of closing parentheses which is incremented each time when we pop an element from the stack and write a closing parenthesis. Second, we check each time when we pop an element from the stack if it is equal to the element which we have popped before.

With $rc$ supported by a rank_support and select_support data structure we can formulate Algorithm 19 for answering PSV[i] in constant time. In line 00 we first map $i$ to the corresponding opening parenthesis $ipos$ in $BPS_{set}$. Then we calculate the corresponding closing parenthesis $cipos$. Now, we can skip enclosing parentheses pairs which have an equal LCP value by using information of $rc$. In line 02, we calculate the number of closing parentheses up to position $cipos$ and therefore the index of the closing parenthesis in $rc$. In line 03, we calculate the leftmost index $last_{eq} \geq rc_{idx}$ with $rc[last_{eq}] = 1$. As all $rc$ values between $rc_{idx}$ and $last_{eq} - 1$ equal 0, we can conclude that the corresponding closing parenthesis of $last_{eq}$ in $BPS_{set}$ is at position $cipos = cipos + last_{eq} - rc_{idx}$ and that between $cipos$ and $rc_{idx}$ are only closing parentheses which all correspond to the same LCP value. A call of $enclose$ to the corresponding opening parentheses of $cipos$ will finally return a proper previous smaller value of $i$.

**Algorithm 19** Calculating PSV[i] with $BPS_{set}$ and $rc$ in constant time.

\begin{verbatim}
00  ipos ← select_{BPS_{set}}(i + 1)
01  cipos ← find_{close}BPS_{set}(ipos)
02  rc_{idx} ← rank_{BPS_{set}}(cipos, 0)
03  last_{eq} ← select_{rc}(rank_{rc}(rc_{idx}) + 1)
04  return  j ← enclose_{BPS_{set}}(find_{open}BPS_{set}(cipos + last_{eq} - rc_{idx}))
\end{verbatim}

Note that in our presented solution we have not considered border cases. That is when there exists no previous (next) smaller value for an entry LCP[i]. In this case the methods return 0 ($n - 1$). However, it is easy to check if LCP[0] (LCP[n-1]) is really a smaller value.

Now have a look at Equation 2.15 on page 26 for calculating the parent of an LCP-interval. The only piece that is missing for a constant time solution which is only based on $BPS_{set}$ and $rc$ is to determine if LCP[i] ≥ LCP[j + 1]. In the special case of an LCP-interval
this is easy, since all LCP-values in \([i + 1..j]\) are greater than \(\text{LCP}[i]\) and \(\text{LCP}[j + 1]\). Therefore it follows that all these elements are popped from the stack before \(i\) is popped from the stack. If \(\text{LCP}[j + 1]\) is greater than or equal to \(\text{LCP}[i]\) it is also popped from the stack before \(i\) is popped and therefore it holds for an LCP-interval \([i..j]\)

\[ \text{cipos} > \text{cjpos} \Leftrightarrow \text{LCP}[i] \leq \text{LCP}[j + 1] \] (3.5)

The direct consequence is the following theorem:

**Theorem 1** The parent\((v)\) operation can be computed in constant time using a data structure which takes \(3n + o(n)\) bits of space.

In the next paragraph we will show how to support the \(i\text{th}_\text{child}(v, i)\) also in constant time by calculating each \(\ell\)-index of a LCP-interval in constant time.

**Calculation the \(k\)-th \(\ell\)-index**

Remember that the \(\ell\)-indices of an LCP-interval \(\ell - [i..j]\) are the indices \(i < k_r < j + 1\) with \(\text{LCP}[k_r] = \ell\), where \(\ell\) is the smallest value which occurs in the interval. We find those indices with the help of \(\text{BPS}_{\text{ct}}\) as follows. We make a case distinction which is also schematically depicted in Figure 3.24: First let \(\text{LCP}[i] \leq \text{LCP}[j + 1]\). In this case the closing parentheses of the minimal elements in \([i + 1..j]\) can only be located directly left of the opening parenthesis of element \(j + 1\), as \(\text{LCP}[j + 1] < \text{LCP}[k_r]\) and therefore the closing parentheses of the minimal elements are popped from the stack directly before we push \(j + 1\) and write the corresponding opening parenthesis.

Second let \(\text{LCP}[i] > \text{LCP}[j]\). In this case the closing parentheses of the minimal elements in \([i + 1..j]\) can only be located directly left to the closing parenthesis of element \(i\), as \(\text{LCP}[i] < \text{LCP}[k_r]\) and therefore the closing parentheses of the minimal elements are popped from the stack directly before we pop element \(i\). Also note, that element \(i\) is

\[
\begin{array}{cccccccccc}
\text{LCP-index} & i & k & k_{j+1} & j+1 & i \\
\text{BPS}_{\text{ct}}\text{-index} & \text{ipos} & \text{kpos} & \text{cjpos} & \text{cipos} & \text{cipos} \\
\end{array}
\]

\[
\cdots ( \cdots ( \cdots ) ) ( \cdots ) \cdots \approx \text{LCP}[i] \leq \text{LCP}[j + 1]
\]

\[
\begin{array}{cccccccccc}
\text{LCP-index} & i & k & k_{j+1} & j+1 & i \\
\text{BPS}_{\text{ct}}\text{-index} & \text{ipos} & \text{kpos} & \text{cjpos} & \text{cipos} & \text{cipos} \\
\end{array}
\]

\[
\cdots ( \cdots ( \cdots ) ) ( \cdots ) \cdots \approx \text{LCP}[i] > \text{LCP}[j + 1]
\]

**Figure 3.24:** The two cases which have to be handled for the calculation of the \(\ell\)-indices of an LCP-interval \(\ell - [i..j]\).
directly popped before we push element \( j + 1 \) onto the stack. Therefore, we calculate in both cases the positions of the \( \ell \)-indices in the \( \text{BPS}_{sct} \) relative to the opening parenthesis of element \( j + 1 \). This can also be seen in the pseudo-code in Algorithm 20. Note that we have omitted in Algorithm 20 a check, if the \( k \)-th \( \ell \)-index exists, for the purpose of clarity. However, this check can be done as follows: If the parenthesis at position \( ckpos \) is not a closing one, then the \( k \)-th \( \ell \)-index does not exist. Otherwise we can check in constant time with \( rc \) if the closing parenthesis at position \( ckpos \) and \( x \) have the same value (\( = \ell \)-value) and therefore the \( k \)-th \( \ell \)-index exists.

**Algorithm 20** Calculating the \( k \)-th \( \ell \)-index with \( \text{BPS}_{sct} \) in constant time.

```plaintext
00  cipos ← find_close_{\text{BPS}_{sct}}(select_{\text{BPS}_{sct}}(i + 1))
01  cjpos ← find_close_{\text{BPS}_{sct}}(select_{\text{BPS}_{sct}}(j + 2))
02  if cipos > cjpos then  // LCP\[i\] ≤ LCP\[j + 1\]
 03      x ← select_{\text{BPS}_{sct}}(j + 2) - 1
04  else
05      x ← select_{\text{BPS}_{sct}}(j + 2) - 2
06  ckpos ← x - (k - 1)
07  return rank_{\text{BPS}_{sct}}(\text{find open}_{\text{BPS}_{sct}}(ckpos))
```

A direct consequence of the constant time computation of the \( k \)-th \( \ell \)-index is the following theorem:

**Theorem 2** The \( \text{ith}_\text{child}(v, i) \) operation of a CST can be computed in constant time using data structures which takes \( 3n + o(n) \) bits of space.

In combination with a CSA which answers its \( [] \)- and \( () \)-operations in constant time and a LCP array which answers its \( [] \)-operation in constant time we can compute the \( \text{child}(v, c) \) operation in \( \mathcal{O}(\log \sigma) \) time by a binary search for the right starting character \( c \) on the edges to its child nodes.

Since the \( \text{sibling}(v) \) boils down to finding the next \( \ell \)-index it can computed in constant time with the \( 3n + o(n) \) data structure. Also note that the \( \text{degree}(v) \) operation can be calculated in constant time using rank and select queries on \( rc \).

A complete list of the resulting time complexities for all operations of all CSTs of the last three sections is given in Figure 3.7. Sadakane’s solution is labeled with \( \text{cst}_\text{sada} \), Fischer et al.’s solution is labeled with \( \text{cstY} \), and our new proposal is labeled with \( \text{cst}_{sct3} \). Note that we have omitted the \( \mathcal{O}(\cdot) \) in all cases and have broken down the time complexity to the time of basic operations. \( t_r \) is the time for \( \text{rank} \), \( t_s \) for \( \text{select} \), \( t_{fc} \) for \( \text{find close} \), and so on.

### 3.9.4 Node Type, Iterators, and Members

#### Node Type

We have seen that different CST proposals led to different node representations in a CST. For example in \( \text{cst}_\text{sada} \) nodes of the CST are represented by integers in the range
<table>
<thead>
<tr>
<th>Operation</th>
<th>cst_sct3</th>
<th>cst_sada</th>
<th>cstY</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nodes()</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>root()</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>is_leaf(v)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>id(v)</td>
<td>(t_r)</td>
<td>(t_r)</td>
<td></td>
</tr>
<tr>
<td>lb(v)</td>
<td>(t_r)</td>
<td>(t_r)</td>
<td></td>
</tr>
<tr>
<td>rb(v)</td>
<td>1</td>
<td>(t_r)</td>
<td>1</td>
</tr>
<tr>
<td>depth(v)(^{(1)})</td>
<td>(t_r + t_{SA})</td>
<td>(t_{SA})</td>
<td>(t_{RMQ} + t_{LCP})</td>
</tr>
<tr>
<td>node_depth(v)</td>
<td>(d \cdot t_{parent})</td>
<td>(t_R)</td>
<td>(d \cdot t_{parent})</td>
</tr>
<tr>
<td>degree(v)</td>
<td>(t_r)</td>
<td>(t_r + t_{SA})</td>
<td>(t_{RMQ} + t_{NSV} + t_{PSV})</td>
</tr>
<tr>
<td>parent(v)</td>
<td>(t_r + t_s + t_{ec} + t_{fc} + t_{fo})</td>
<td>(t_{ec})</td>
<td>(t_{LCP} + t_{NSV})</td>
</tr>
<tr>
<td>sibling(v)</td>
<td>(t_{fo})</td>
<td>(t_{fc})</td>
<td>(t_{LCP} + t_{NSV})</td>
</tr>
</tbody>
</table>
| sl(v)             | \(s \cdot t_{

Table 3.7: Detailed time complexities of the CST methods depending on basic functions. Note that the times \(t_{RMQ}\), \(t_{PSV}\), and \(t_{NSV}\) depend on \(t_{LCP}\) in the implementation of Cánovas (cstY) Blank cells in the table indicate that the corresponding methods are not implemented. Remarks: (1) The time complexity on top is valid for leaf nodes, the one at the bottom for inner nodes. (2) \(s = \min(d, s_{SA} + s_{ISA})\). (3) \(d = \text{depth}(v)\) and \(s = \min(d, s_{SA} + s_{ISA})\).
[0..\text{nodes}()], in \text{cstY} nodes are represented by intervals [lb..rb], and in \text{cst\_sct2} nodes are represented by the corresponding opening parentheses of lb and rb + 1 in BPS\text{\_sct}.

Each CST has to offer a type definition for its node representation. This type definition is called \text{node\_type}. When we use this type definition in client code, we can replace the CST implementation afterwards without any adjustment in our client code and we get rid of the concrete implementation of a node. A very simple code example for the use of this type definition is given in Listing 3.1: We wrote a function \text{output\_node} which takes a reference to a CST of type \text{Cst} and a node v of type \text{node\_type}. Since node arguments of all operations of the CST are always defined to be of type \text{node\_type} this function will work for every CST class which fulfills the \text{cst} concept. Therefore, we have a function which transforms a node of a CST into the corresponding LCP-interval and outputs the result. We have also used this generic form of coding for all the experiments which we will present in Chapter 4.

3.9.5 Iterators

Another useful tool in generic programming is an iterator. The CST concept offers two constant size iterators which are implemented by using the methods \text{parent}(v), \text{sibling}(v), and \text{i\_th\_child}(v, i) or \text{left\_most\_leaf\_in\_the\_subtree}(v). Since each of these operations takes constant time, the incrementation of each iterator also takes constant time.

The first iterator traverses the CST in depth-first order (dfs). Each inner node v is visited two times: the first time when we have visited already all nodes left to v, and the second time when we have visited all nodes in the subtree of v. Leaf nodes are only visited once. The dfs iterator is used for example to build the parentheses sequence BPS\text{dfs} of Sadakane’s CST out of the NAV structure of \text{cst\_sct3}. That is we can build BPS\text{dfs} with only 3n + o(n) bits of extra space in linear time. A code example for using the dfs iterator is given in Listing 3.2.

The second iterator, traverses the CST in bottom up order. That is we start at the leftmost node in the tree and visit each node v after all nodes in its subtree have been visited. In contrast to the dfs iterator, each node is visited exactly once. The method \text{begin\_bottom\_up()} of a CST returns the iterator for the first node of the traversal and \text{end\_bottom\_up()} the last one.

Listing 3.1: Example for using the \text{node\_type} definition of the CST concept.

```cpp
#include <sdsl/suffixtrees.hpp>
#include <iostream>

// Formatted output of a CST node v

template<
    Cst>
void output_node(const Cst &cst, typename Cst::node_type v)
{
    std::cout << cst.depth(v) << "-[
    << cst.lb(v) << "," << cst.rb(v) << std::endl;
}
```
3.9 The cst Concept

Listing 3.2: Example for using the dfs iterator of the CST concept.

```cpp
#include <sds1/suffix_trees.hpp>
#include <iostream>

// Output all nodes of a CST
template <Cst>
void output_all_node(const Cst &cst) {
    typedef typename Cst::const_iterator iterator_type;
    for (iterator_type it = cst.begin(); it != cst.end(); ++it) {
        if (it.visit() == 1) // if it is the first visit
            output_node(cst, *it); // output the node: see previous listing
    }
}
```

Many problems can be solved with a bottom up traversal of a CST; cf [Gus97]. One example is the calculation of all maximal repeated pairs of a specific length of a text.

A breadth-first search (bfs) iterator was not implemented, since this iterator would take linear space.

Members

Each CST in the sds1 can be parametrized with a CSA, a LCP array, and support structures for the specific NAV structure of the CST. Since the CST does not provide all operations of the underlying CSA and LCP array, we decided to introduce const references to the CSA and LCP array for the convenient use of these structures. Listing 3.3 shows how intuitive it is to output information like the Burrows Wheeler transformed text, $\Psi$, LF, SA, ISA, and LCP with a minimum amount of code.
Listing 3.3: Example for using members of the CST.

```cpp
template<Cst>
void cst_information(const Cst &cst)
{
    cout << "Alphabet/uni2423size/uni2423of/uni2423the/uni2423CST: " << endl;
    cout << cst.csa.sigma << endl;
    for (typename Cst::size_type i=0; i < cst.size(); ++i){
        cout << i << "// Output: 
        << cst.csa[i] << ' ' // suffix array
        << cst.csa(i) << ' ' // inverse suffix array
        << cst.lcp[i] << ' ' // lcp array
        << cst.csa.psi[i] << ' ' // psi function
        << cst.csa.psi(i) << ' ' // LF function
        << cst.csa.bwt[i] << endl; // Burrows Wheeler transform
    }
}
```

3.10 The lcp Concept

In this Section we will present a concept for LCP-arrays and present a new compressed representation for LCP-arrays, which can be used in combination with a CST. The lcp concept itself is very simple to explain. The only operation which has to be supported is the $[i]$-operator which returns the length of the longest common prefix between suffixes $SA[i-1]$ and $SA[i]$.

There are three main types of compressed LCP representations:

(a) The representation in suffix array order, i.e. we store at position $i$ the length of the longest common prefix of the suffixes $SA[i-1]$ and $SA[i]$. Examples for this representation are the bit-compressed LCP array, the solution of Kurtz [Kur99], the direct accessible code solution of Brisaboa et al. [BLN09] and the wavelet tree solution.

(b) The representation in text order. The length of the longest common prefix of the suffixes $SA[i-1]$ and $SA[i]$ is stored at position $SA[i]$. This representation is also known under the term PLCP (see Equation 2.19 on page 27). To recover the LCP entry at position $i$, we have to query the suffix array at position $i$. This query is very slow on a CST compared to rank queries (which are used in (a)) or the calculation of $\Psi$ or LF values (which will be useful in (c)). However, the advantage of this ordering of LCP values is that it uses only a fraction of the space of the representations in (a). An example for the PLCP representation is the solution proposed by Sadakane [Sad02]. This solution takes only $2n + o(n)$ bits and has to perform only one select operation on top of the suffix array access. Fischer et al. [FMN09] further compress the size for this kind of representation to at most $n \cdot H_k \cdot (2 \log \frac{1}{H_k} + O(1))$ bits. However, experiments of Cánovas ([Cán10], Section
5.7) show, that the space consumption for different kinds of texts is not smaller than the implementation of the \(2n + o(n)\) bits solution of Sadakane, but the query time is surprisingly more or less equal. In [Fis10b] Fischer proposes another data structure which is basically the select data structure of Sadakane, but needs the original text to work fast in practice.

(c) The tree compressed representation. Here the idea is to use the topology information of the CST to compress the LCP information. Since every LCP value corresponds to the depth value of the lowest common ancestor of two adjacent leaves, it suffices to store the depth values of the \(q\) inner nodes of the CST.

In practice \(q\) is usually in the range from \(0.6n\) to \(0.8n\) ([BEH89] get \(q \approx 0.62\) as a theoretical average for random strings) and therefore we get a notable compression. We call the array, which contains the remaining \(q\) entries \(LCP'\).

To further compress the representation we can apply a type (a) representation to \(LCP'\). This will reduce the space, if there are many small values in \(LCP'\). Otherwise we propose to store only the small values with a type (a) representation and store only a small subset of samples\(^1\) of the large values in \(LCP'\). Large values which are not sampled can then be reconstructed by using the LF function. This two combined compressions lead to a space-efficient solution which is close to or even better (for some special test cases) than type (b) but has a faster access time in practice.

To complete the picture of representations we like to mention the one of Puglisi and Turpin [PT08] which is based on the idea of using difference covers and RMQ queries (see also [KS03]).

3.10.1 The Tree Compressed LCP Representation

The main idea of the tree compressed representation can be explained with Figure 3.25. On the left we have depicted the CST, the SA, and the LCP-array of our running example. On the right, we have only depicted the tree topology and the depth value of each inner node. Now remember that \(LCP[i] = \text{depth}(lca(i\text{th leaf}(i - 1), i\text{th leaf}(i)))\), i.e. \(LCP[i]\) can be stored at an inner node \(v = lca(i\text{th leaf}(i - 1), i\text{th leaf}(i))\). So it suffice to store only the depth values at the \(q\) inner nodes of the CST to answer \(LCP[i]\) for any \(i\). We will now show how we can store the \(q\) depth values along with a CST.

**Definition 2** For each inner node \(v\) of a suffix tree \(ST\), the postorder index \(\text{po_idx}(v)\) equals the number of inner nodes that are printed before \(v\) is printed in a postorder tree walk through \(ST\).

Figure 3.25 depicts all postorder indices of our example CST.

\(^1\) The idea of using a sampled (P)LCP array was already presented by Sirén in [Sir10]. We use a very simple sampling approach here, which will not necessarily meet the time bound of Sirén’s paper.
Figure 3.25: Left: CST, SA, and LCP-array. Right: The tree topology of the CST and each node $v$ is labeled with the pair $(\text{depth}(v), \text{po_idx}(v))$. On the bottom we have depicted $LCP'$ which is stored in a type (c) representation.

**Definition 3** For each index $i \in [0..n-1]$ of the LCP array the tree-lcp index denoted with tlcp_idx($i$) corresponds to the postorder index of the lowest common ancestor of the ($i+1$)-th and ($i+2$)-th leaf of the suffix tree.

The pseudo-code for the calculation of tlcp_idx($i$) in `cst_sada` is given in Algorithm 21 and Algorithm 22 depicts the pseudo-code for `cst_sct3`. We can observe that in both cases we only need a constant number of constant time operations. Therefore we get the following theorem.

**Theorem 3** The postorder index and the tree-lcp index can be computed in constant time in the CSTs `cst_sada` and `cst_sct3`.

We will now prove the correctness. In the case of `cst_sada` the `select` operations in line 02 return the positions of leaf $i$ and leaf $i+1$. The position of the opening parenthesis of the lowest common ancestor $v$ of leaf $i$ and $i+1$ is then calculated by a `double_enclose` operation. Finally, the postorder index of $v$ is determined by jumping to the closing parenthesis of $v$ and counting the number of inner nodes that were visited before the last visit of $v$, which corresponds to the postorder index of $v$.

In the case of `cst_sct3`, line 01 corresponds to jumping to an opening parenthesis which corresponds to an $\ell$-index of the lowest common ancestor $v$ of leaf $i-1$ and leaf $i$. The `find_close` on line 02 skips all nodes in the CST which are located in the subtrees of $v$. In line 03 we first determine the number $x$ of closing parentheses which are left to $\text{ipos}$, i.e. $x = \text{rank}_{\text{BPS}_{\text{st}}}(\text{ipos},0)$. However, we do not have to make a `rank` query, since
Algorithm 21 Calculating $tlcp_{idx}(i)$ in $cst_{sada}$.

01 if $i > 0$
02     $jpos \leftarrow double_{enclose}_{BPS_{ds}}(select_{BPS_{ds}}(i,'10'), select_{BPS_{ds}}(i+1,'10'))$
03 else
04     $jpos \leftarrow 0$
05 $cjpos \leftarrow find_{close}_{BPS_{ds}}(jpos)$
06 return $(cjpos - rank_{BPS_{ds}}(cjpos,'1') - rank_{BPS_{ds}}(cjpos,'10'))
\quad \text{rank}_{BPS_{ds}}(cjpos,'0')$

we have calculated enough information in the previous two lines:

$$x = rank_{BPS_{sct}}(ipos,0) = rank_{BPS_{sct}}(ipos,0) + rank_{BPS_{sct}}(cipos,0) - rank_{BPS_{sct}}(ipos,0)$$

The $select$ query in line 01 tells us that there are $i$ ones left of position $ipos$ and therefore $ipos - i$ zeros. In line 02, we have calculated the matching closing parenthesis for $ipos$, so we know that $BPS_{sct}[ipos..cipos]$ is a balanced parentheses sequence. So the number of zeros in $BPS_{sct}[ipos..cipos-1]$ is exactly $\frac{cipos-ipos-1}{2}$.

Finally a call to $rank_{rc}(x)$ results in the number of nodes which are left of $v$ or in the subtree of $v$, i.e. results in the postorder index of $v$!

Algorithm 22 Calculating $tlcp_{idx}(i)$ in $cst_{sct3}$.

01 $ipos \leftarrow select_{BPS_{sct}}(i+1)$
02 $cipos \leftarrow find_{close}_{BPS_{sct}}(ipos)$
03 return $\frac{rank_{rc}(ipos+cipos-1) - i}{2}$

The tree compressed data structure $lcp_{support\_tree}$ can now be realized as follows. First, we store the $q$ depth values in postorder in an array $LCP'$. Note that we can easily integrate the construction of $LCP'$ into Algorithm 16 and therefore need only $n + o(n)$ additional bits for the succinct stack. Second, the $|i|$-operator is implemented as follows. We calculate $j = tlcp_{idx}(i)$ and return $LCP'[j]$. Note that we can parametrize $lcp_{support\_tree}$ with any $LCP$ data structure of type (a) for $LCP'$.

3.10.2 An Advanced Tree Compressed LCP Representation

The space consumption of $lcp_{support\_tree}$ depends on the number $q$ of inner nodes of the CST as well as the type (a) schema of LCP representation which is used for $LCP'$. All
type (a) representations have one common drawback. They cannot compress large values, as they do not use additional informations from the CST. In our advanced solution we will take a Huffman shaped wavelet tree (wt_huff) to store a modified version of LCP called LCP_SMALLrc.

$$\text{LCP\_SMALLrc}[j] = \begin{cases} \text{LCP}'[j] & \text{if } \text{LCP}'[j] < 254 \\ 254 & \text{if } \text{LCP}'[j] \geq 254 \text{ and } \text{LCP}'[j] \text{ is not sampled} \\ 255 & \text{if } \text{LCP}'[j] \geq 254 \text{ and } \text{LCP}'[j] \text{ is sampled} \end{cases}$$

(3.6)

In the first case of Equation 3.6 answering LCP[i] is done exactly as in the implementation of lcp_support_tree: We calculate \(j = tlcp\_idx(i)\) and return \(\text{LCP}'[j] = \text{LCP\_SMALLrc}[j]\). See also lines 01-04 in Algorithm 23. Now we store the values of the \(p < q\) sampled large values in a bit-compressed int_vector of size \(p\) called LCP_BIGrc.

When \(\text{LCP\_SMALLrc}[j] = 255\) indicates that there is a sampled large value, we can determine the corresponding index in LCP_BIGrc by a \text{rank} query which is supported naturally by the Huffman shaped wavelet tree LCP_SMALLrc. Note, that if there are many large values in LCP, then the \text{rank} query in a Huffman shaped wavelet tree is faster, as we use less bits for the code of 255 in the wavelet tree. Therefore the runtime of the operations adapts to the input data!

Before we can explain the rest of the algorithm, we have to state how the sampling is done. First we choose a sampling parameter \(s_{\text{LCP}} \geq 1\). All large values \(x = \text{LCP}'[j]\) with \(x \equiv 0 \mod s_{\text{LCP}}\) are sampled. Next, we check for each large value \(x = \text{LCP}'[j]\) with \(x \not\equiv 0 \mod s_{\text{LCP}}\) if we can “easily reconstruct” it from other values in LCP’ by using the LF function. That is, we take for every \(\text{LCP}'[j]\) all indices \(i_k\) \((0 \leq k < t)\) with \(j = tlcp\_idx(i_k)\) and check, if \(T^{\text{BWT}}[i_k] = T^{\text{BWT}}[i_k - 1]\) for all \(k \in [0..t-1]\). In other words, we check if all \(\text{LCP}[i_k]\) are reducible. If this is true, we can calculate \(\text{LCP}[i_k]\) as follows: \(\text{LCP}[i_k] = \text{LCP}[\text{LF}[i_k]] - 1\) (see also Equation 2.17 on page 27).

Therefore, storing a large value for index \(j\) in LCP’ is not necessary and we mark that with \(\text{LCP\_SMALLrc}[j] = 254\); all other large values \(\text{LCP}[j]\) are added to the sampled values by setting \(\text{LCP\_SMALLrc}[j] = 255\). It is worth mentioning, that the size of the set of large values which were not “easy to reconstruct” from LF was very small in all our experiments.1

Note that we can again use Algorithm 16 to calculate all the informations for \(\text{LCP\_SMALLrc}\) in linear time and only \(n + o(n)\) extra space for the succinct stack, since \(T^{\text{BWT}}\), which is needed to decide, if values are sampled or not, can be streamed from disk.

Algorithm 23 depicts the whole pseudo-code for the \(\lfloor i \rfloor\)-operator of lcp_support_tree2. The variable offset which is defined in line 05 is used to keep track of how many times we have skipped a large LCP-value and used LF to go to the next larger value. When we finally reach a sampled large value in line 11, we have to subtract offset from the

---

1 We made a very similar observation on page 137: There ratio of large irreducible LCP-values is very small.
3.10 The lcp Concept

LCP-value to get the correct answer.

Algorithm 23 Calculation of LCP[i] in the class lcp_support_tree2.

\begin{verbatim}
3.10.3 Experimental Comparison of LCP Implementations
In this section we will give an experimental analysis of seven proposals of compressed LCP representations. To get a fair comparison we use in all data structures the same basic data structures like rank_supports, select_supports, and so on. Here are details of the implementations:

- **lcp_uncompressed** The bit-compressed representation which uses \( n \log n \) bits for each entry.

- **lcp_wt** We use a Huffman shaped wavelet tree to store the LCP values.

- **lcp_dac** We use the direct accessible code (presented in [BLN09]) with different values for the time-space trade-off parameter \( b \) (\( b \in \{4, 8, 16\} \)). The compression work as follows: The \( b \) least significant bits of each LCP value are stored in an int_vector\( <b> \) \( d_0 \) of size \( n_0 = n \). We store an additional bit_vector \( overflow_0 \) of size \( n_1 \), which indicates for each entry \( LCP[i] \), if it is greater than \( 2^b - 1 \). For each such \( LCP[i] \), we store the next \( b \) least significant bits in another int_vector\( <b> \) \( d_1 \) of length \( n_1 = rank_{overflow_0}(n_0) \) at position \( rank_{overflow_0}(i) \) and store an additional bit_vector \( overflow_1 \) of size \( n_1 \), which indicates if \( LCP[i] > 2^b - 1 \) and so on until \( n_k = 0 \). So accessing an LCP value \( x \) takes about \( \log(x + 1)/b \) rank operations.

- **lcp_kurtz** We use one byte for values < 255 and store values \( LCP[i] \geq 255 \) as a pair \((i, LCP[i])\) in a sorted array, i.e. the value at position \( i \) can be recovered in \( \log n \) time [Kur99].

- **lcp_support_sada** The solution of Sadakane which uses \( 2n + o(n) \) on top of the suffix array. We use a CSA based on a Huffman shaped wavelet tree and set the sample
values for the SA and ISA values to 32, i.e. the class \texttt{csa\_wt<wt\_huff<>\x{32},32>}.  

- \texttt{lcp\_support\_tree} was presented in Section 3.10.1. We use \texttt{lcp\_wt} as container for LCP’ and \texttt{cst\_sct3} as CST parametrized with \texttt{csa\_wt<wt\_huff<>\x{32},32>}.  

- \texttt{lcp\_support\_tree2} was presented in Section 3.10.2. We set $s_{\text{LCP}} \in \{2,4,8,16,32\}$ to get a time-space trade-off. For LCP\_SMALL\_rc we use \texttt{wt\_huff} and for LCP\_BIG\_rc we use an \texttt{int\_vector}.  

We measure only the space of the real LCP information, i.e. we do not take into account the space for the CSA or the CST for \texttt{lcp\_support\_sada}, \texttt{lcp\_support\_tree}, and \texttt{lcp\_support\_tree2}. Figure 3.26 shows the results of the following experiment: We have performed $10^7$ queries to $10^6$ random positions of the LCP array of each \texttt{Pizza\&Chili} test case. The diagrams show the average time for one query and the space occupied by the structures. The bit-compressed \texttt{int\_vector} of the class \texttt{lcp\_uncompressed} takes the most space ($\lceil \log(200 \cdot 2^{20}) \rceil = 28$ bit) and takes about 100 nanoseconds in all cases. This is about 1.3 times the runtime which we have determined for a random access on a bit-compressed \texttt{int\_vector} in Figure 3.3 on page 38. For test cases with many small values < 255 like \texttt{dna.200MB} or \texttt{rank\_k128.200MB}, the alternatives \texttt{lcp\_kurtz} and \texttt{lcp\_dac} perform even better with only a fraction of the space. However, for the test cases \texttt{english.200MB} and \texttt{sources.200MB} where we have many large values $\geq 255$ both solutions slow down due to the special handling of the large values. The most space saving solution for structured texts is clearly \texttt{lcp\_support\_sada}. It occupies only a little bit more than $2n$ bits but has a runtime between 10000 and 40000 nanoseconds. This is about 100 – 400 times slower than \texttt{lcp\_uncompressed}! We can observe that the running time of \texttt{lcp\_support\_sada} depends mainly on the underlying CSA and not on the additional select query, which takes only about 400 nanoseconds. Let us have a look at Figure 3.6 on page 51 for the runtime of the CSA. We have chosen to use a \texttt{csa\_wt<wt\_huff<>\x{32},32>} for \texttt{lcp\_support\_sada}, this corresponds to the second point from the left on the line for the \texttt{cst\_wt<wt\_huff<> \x{32},32>} class. Its runtime is always about the runtime of \texttt{lcp\_support\_sada} for all test cases. As a consequence, we can conclude that we can get a runtime for \texttt{lcp\_support\_sada} that is 10 times faster if we use a less compressed CSA but then we have to pay this with much additional space for the CSA.  

Our new solutions \texttt{lcp\_support\_tree} and \texttt{lcp\_support\_tree2} take about 0.2 – 5 bits for all test cases. \texttt{lcp\_support\_tree} takes about 1000 – 2000 nanoseconds per operation. This was expected, since the sum of the runtime of the operations in \texttt{tlcp\_idx(i)(see Algorithm 22, 1× select, 1× find\_close, and (1+8)× rank)} is about 1000–2000 nanoseconds. So we have a solution which is about 10 times slower than \texttt{lcp\_dac}, but reduces the space consumption significantly in all cases.  

\texttt{lcp\_support\_tree2} further reduces space by only storing about every $s_{\text{LCP}}$-th large LCP-value for $s_{\text{LCP}} \in \{2,4,8,16,32\}$. Clearly, for test cases with many small values (\texttt{dna.200MB}, \texttt{dblp.200MB} and \texttt{rank\_k128.200MB}) the solution shows no time-space trade-off, as there are no large values which can be omitted. Otherwise, like in the case of \texttt{english.200MB}, our solution closes the gap between \texttt{lcp\_support\_sada} and \texttt{lcp\_dac}.  

...
Figure 3.26: Average runtime of one random LCP access for different LCP representations included in the sdsl.
We provide a solution which takes about 3.5 bits but is about 10 times faster than \texttt{lcp\_support\_sada}.

In the special case of the random text over an alphabet of size 128, our solution takes very little space. This is due to the fact, that the CST has only few inner nodes (about 0.2\text{n}) and we store only LCP-values at inner nodes.

In some applications the LCP-array is accessed in sequential order, e.g. in the CST of Fischer et al. [FMN09]. Therefore we performed a second experiment which takes a random position and then queries the next 32 positions.\footnote{This is exactly the same experiment which was performed by Cánovas in [Cán10].} In total we performed $32 \cdot 10^7$ queries and took the average runtime for Figure 3.27. We can observe that the type (a) LCP representations are 10 times faster compared to the previous experiment. \texttt{lcp\_support\_sada} does not profit form sequential access, since it has not stored the information in sequential order. Our solutions \texttt{lcp\_support\_tree} and \texttt{lcp\_support\_tree2} do profit from the sequential access, since we store the values in postorder of the CST and therefore get a good locality in some cases. For instance, one case is a sequence of equal values in the LCP-array. We can observe that our solutions are at least two times faster compared to the previous experiment.

Conclusions

- \texttt{lcp\_dac} is always a good replacement for \texttt{lcp\_uncompressed}. Its runtime is slightly better for arrays with many small values and for the parameter $b = 8$ its runtime is about two times the runtime of \texttt{lcp\_uncompressed}, but \texttt{lcp\_dac} occupies only around $8 - 10$ bits for real world texts. The access is about 10 times faster if the queries are in sequential order.

- \texttt{lcp\_support\_tree} and \texttt{lcp\_support\_tree2} are about 10 times slower than \texttt{lcp\_dac} but occupy only around 3 – 7 bits. The runtime profits from sequential access by a factor of 2.

- \texttt{lcp\_support\_sada} is for reasonably small CSAs about 10 times slower than \texttt{lcp\_support\_tree} and uses in all cases a little bit more than $2\text{n}$ bits. The runtime does not profit from a sequential access order.
3.10 The lcp Concept

![Graphs showing average runtime time for different LCP representations](image)

**Figure 3.27:** Average runtime time of one random sequential LCP access for different LCP representations included in the *sdsl*. 
4 Experimental Comparison of CSTs

In this chapter we will compare the performance of our implementation with other state of the art implementations. We will hereby explain our implementation in more detail and illustrate for some operations why other implementations result in a bad runtime.

4.1 Existing Implementations

In 2007 Välimäki et al. [VMGD09] presented the first implementation of a CST which we will call cstV. They follow closely the proposal of Sadakane [Sad07a] and therefore use a CSA based on a Huffman shaped wavelet tree, the $2n + o(n)$ bits-LCP array, the $\text{BPS}_{dfs}$ plus a sub-linear structure for the navigation.

From today’s perspective, the implementation has several drawbacks:

- The $\Psi$ function was implemented inefficiently, i.e. the running time equals $t_\Psi = t_{SA} = O(s_{SA} \cdot \log \sigma)$, while we get $t_\Psi = \log \sigma$ by using Equation 2.10 and a wavelet tree. Therefore, all operations which use the $\Psi$ function (e.g. $sl(v)$) cannot compete with recent implementations.

- The operations $\text{find\_close}(i)$ and $\text{enclose}(i)$ do not run in constant time.

- They use $2n + o(n)$ extra bits for the rank data structure for the “10” bit pattern, while $o(n)$ suffice to do that.

- The construction algorithm for the balanced parentheses sequence is very slow $(n \log n \log \sigma)$ and takes $n \log n$ bits in the worst case.

- The memory consumption is about $3 - 3.5$ bytes per input character and the sub-linear ($o(n)$) data structures take about half of the memory of their CST. They stated that lowering the memory consumption of the sub-linear structures is therefore the major task for further research.

In 2008 Russo et al. [RNO08] presented the fully-compressed suffix tree. While this CST takes only sub-linear space in theory, it still performs the navigational operations in logarithmic time. An implementation of the data structure is available on request from Luís Russo.

In 2010 Cánovas and Navarro [CN10, Cán10] presented another implementation of a CST which is based on the paper of Fischer et al. [FMN09]. They also present an experimental study which compares their implementation to cstV and the one from Russo. They thereby show that
4.2 Our Test Setup

- \texttt{cstV} takes about $3 - 4.3$ bytes per input character for the \textit{Pizza&Chili} test cases and the running times of the operations — except for the parent operation — are slower than that of the implementation of Cánovas.

- the CST implementation of Russo et al. \cite{RNO08} takes only $0.5 - 1.1$ bytes per input character, which is remarkable. Unfortunately, the running times of the operations are orders of magnitudes slower than that of the other implementations. The running times of the operations lie between $0.1$ milliseconds and $10$ milliseconds.

- they can get several CSTs with different time-space trade-offs (1) by using different LCP arrays and (2) using a time-space trade-off parameter in their implementation of the PSV, NSV, and RMQ structures. The CSTs can be grouped by (2) into two families: The “large and fast” ones which use between 1.6 and 2.6 bytes per input character and take between 1 and 10 microseconds for most of the operation \((\text{edge}(v, d)\) and \(\text{child}(v, c)\) take more time). These CSTs use the direct accessible code compression (see \cite{BLN09}, in the \textit{sdsl} we provide the class \texttt{lcp\_dac} for it) for the LCP array. In contrast, the “small and slow” CSTs use either the LCP representation of Sadakane \cite{Sad02} \((\text{lcp\_sada} \text{ in the } \textit{sdsl})\) or the one of Fischer et al. \cite{FMN09}. This results in CSTs of sizes between 1.0 and 2.0 bytes per input character and operation times between 50 and 500 microseconds for all operations except \text{edge}(v, d)\ and \text{child}(v, c), which take more time.

In summary, the implementation of Cánovas and Navarro is the one which outperforms all other existing implementations in terms of query times and at the same time provides a good compression ratio. Therefore, we chose the “large and fast” CSTs as baseline for our experimental comparison.

4.2 Our Test Setup

In this section we present the first experimental comparison of a wide range of CSTs, which all rely on the same implementations of basic data structures, and all components (CSA, LCP, NAV) use also this basic data structures. Moreover, all implementations of CSTs use the same optimizations like bit-parallelism in words. Since we want to show the practical impacts of our contributions we decide to configure eight different CSTs, which are built up from new proposed data structures like \texttt{lcp\_support\_tree2} and \texttt{cst\_sct3} and old ones like \texttt{lcp\_dac} and \texttt{cst\_sada}. We number the configurations from 0 to 7, where

\begin{align*}
0 & \triangleq \texttt{cst\_sada<csa\_sada<>, lcp\_dac>}
\end{align*}

\begin{align*}
1 & \triangleq \texttt{cst\_sada<csa\_sada<>, lcp\_support\_tree2>}
\end{align*}

\begin{align*}
2 & \triangleq \texttt{cst\_sada<csa\_wt<>, lcp\_dac>}
\end{align*}

\begin{align*}
3 & \triangleq \texttt{cst\_sada<csa\_wt<>, lcp\_support\_tree2>}
\end{align*}

\begin{align*}
4 & \triangleq \texttt{cst\_sct3<csa\_sada<>, lcp\_dac>}
\end{align*}
All configurations use \texttt{bp\_support\_sada} as \texttt{bp\_support} data structure which is the default value for the third template parameter; see Section 3.8.3. The components \texttt{csa\_sada} and \texttt{csa\_wt} are parametrized with the sample parameters $s_{\text{SA}} = 32$ and $s_{\text{ISA}} = 64$ which results in a reasonable time-space trade-off for the CSAs; see Figure 3.6. \texttt{csa\_sada} is parametrized with \texttt{enc\_vector} $<\psi = 128>$, since this value is also used in [CN10]. In \texttt{lcp\_support\_tree2} we set the sampling parameter $s_{\text{LCP}} = 4$ as larger values do not significantly reduce the space but increase the runtime; cf. Figure 3.26.

The “large and fast” implementation of Cánovas and Navarro are part of the the \texttt{libcds} library and denoted by \texttt{cst\_Y} or “8” in the diagrams. We got a patched version of the \texttt{libcds} library directly by Rodrigo Cánovas, since we found a couple of bugs in the latest available version 1.08 during our experiments. Unfortunately, the implementation does not work for random text over the alphabet of size 128.

To get an impression of the performance of the nine CSTs, we build CSTs for all \texttt{Pizza&Chili} test cases of sizes 50 MB, 100 MB, and 200 MB. Since the results are similar for each size we only include the results for the test cases of 200 MB. However, testing 9 different CSTs on $18 = 3 \cdot 6$ test cases was very useful, since we discovered some bugs in the implementations. We measure the average time for each CST operation by executing the operation between $10^5$ and $10^9$ times ($10^5$ times for slow, and $10^9$ times for fast operations) on different input values, i.e. nodes $v$ and $w$.

To get a meaningful runtime of the operations we use the following sets of nodes, which in that form occur in algorithms which use a suffix tree, as input values for the operations:

- \(V_1(x)\) We take \(x\) random leaves of the CST. For each random leaf $v$ we add all nodes of the path from $v$ to the root to $V_1(x)$; used to measure \texttt{parent}(v), \texttt{child}(v,c)^2, \texttt{ith\_child}(v,1), \texttt{sibling}(v), \texttt{id}(v), \texttt{depth}(v)$.

- \(V_2(x)\) We take \(x\) random leaves of the CST. For each parent $v$ of a random leaf we call the suffix link operation until we reach the root and add all these nodes to $V_2(x)$; used to measure \texttt{sl}(v).

- \(V_3(x)\) We take \(x\) random leaf pairs; use to measure \texttt{lca}(v,w).

Before we take a tour through the detailed results of each operation, we first have a look at Figure 4.1. It contains two boxplots. The first one depicts the runtime for each operation on all test cases of the \texttt{cst\_sada} based CSTs, i.e. configurations (0)-(3), while the second depicts it for the \texttt{cst\_sct3} based CSTs, i.e. configurations (4)-(7).

\footnotetext[1]{The library is available under http://libcds.recoded.cl.}
\footnotetext[2]{Here we have also chosen characters $c$ from random positions in $T$.}
Since the runtime of different operations differs by order of magnitudes, but we still want to compare the runtime of operations which are in the same order of magnitude, we use two y-axis. The blue one on the top is for the fast operation, for which the boxes are also drawn in blue, and the black one on the bottom is for the slow operations. The runtimes of the first three operations are almost equal, since both configurations types (0)-(3) and (4)-(7) use the same CSA and LCP structures an equal number of times. All other operations depend on the NAV structure of the CSTs and we therefore can observe differences in the runtime. The main differences occur at the navigational operations parent\((v)\), sibling\((v)\), \(\text{ith}_{-}\text{child}(v, i)\), and \(\text{child}(v, c)\). The first three are dominated by \texttt{cst\_sada} while \texttt{cst\_sct3} is slightly better for the \(\text{child}(v, c)\) operation. We will now discuss the experimental results for these operations in detail.
Runtime of \texttt{cst\_sada} operations

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cst_sada_runtime}
\caption{Runtime of \texttt{cst\_sada} operations on the Pizza\&Chili corpus. We use two box-plots to provide an overview of the runtime of \texttt{cst\_sada} and \texttt{cst\_sct3}. Both were parametrized with different CSA and LCP structures. The description of all operations except of \texttt{depth}(v), \texttt{dfs} and \texttt{id}(v), \texttt{dfs} and \texttt{depth}(v), and \texttt{mstats}, can be found in Table 3.1. While \texttt{depth}(v) labels the runtime of the depth function for arbitrary nodes v, \texttt{depth}(v)^* labels the time only for inner nodes. We measured the average runtime per operations in a traversal in depth-first-search order that calls the \texttt{id}(v) or \texttt{depth}(v) operation for each node or the average time per step for the calculation of matching statistics by backward search. Note that the blue drawn operations are an order of magnitude faster than the black ones. Their runtime can be determined with the blue y-axis on the top.}
\end{figure}
4.3 Our Test Results

4.3.1 The parent\((v)\) Operation

Figure 4.2 shows the time-space trade-off of the different CST implementations for the parent\((v)\) operation. As expected the CSTs based on cst\_sada are the fastest, since a parent operation boils down to an \textit{enclose} operation in BPS\_dfs, which is performed in about 800 – 1000 nanoseconds (see Figure 3.20). Since the other solutions depend on calculating NSV (see Equation 17) and PSV (see Algorithm 19), which both require multiple calls of \textit{rank}, \textit{select}, \textit{find\_open}, \textit{find\_close}, and \textit{enclose} (see also Table 3.7), they are at least 2-3 times slower than the \textit{cst\_sada} based CSTs. The \textit{cst\_sct3} solutions (4)-(7) and the different version of \textit{cstY}(8) achieve similar results, but for instance implementation (5) uses clearly less space than \textit{cst\_sct3}.

4.3.2 The sibling\((v)\) Operation

Figure 4.3 shows the time-space trade-off of the different CST implementations for the sibling\((v)\) operation. Like for the parent\((v)\) operation, the CSTs based on \textit{cst\_sada} are the fastest, since a sibling operation boils down to a \textit{find\_close} operation in BPS\_dfs, which can be performed in about 200 – 400 nanoseconds (see Figure 3.17). Since we have many far answers for \textit{find\_close} in $V_1(100000)$, the running time tends to be slower than the 200 nanoseconds which were the result of many near answers. Another time the \textit{cst\_sct3} implementations (5) and (7), which use the new \textit{lcp\_support\_tree2} provide a better time-space trade-off than \textit{cst\_sct3}. This can be contributed to the used rank data structure. Other experiments show that using \textit{rank\_support\_v5} in NAV of 5 and 7 slow down the times to about 2300 nanoseconds per operation. But here we use \textit{rank\_support\_v} which in total only adds 3% of memory to the CST but provides a significant acceleration (about 30%) of navigation operations.

4.3.3 The \textit{ith\_child}\((v,1)\) Operation

Figure 4.4 shows the time-space trade-off of the different CST implementations for the \textit{ith\_child}\((v,1)\) operation, which returns the first or leftmost child of a node $v$. The \textit{cst\_sada} versions take only about 20 nanoseconds to answer the question. This equals the time for one random read on a \textit{bit\_vector} (cf. Figure 3.3) which is exactly the time for the most expensive operation in the implementation: We need this read to determine if $v$ is a leaf or an inner node. The runtime of the \textit{cst\_sct3} version is an order of magnitude slower — which is indicated by the prime after the configuration number — as the calculation of the first child boils down to find the first $\ell$-index (see Algorithm 20), which requires a call of \textit{find\_open} and \textit{rank} which takes at least 200 and 150 nanoseconds (see Figure 3.18 and 3.4). As the operation contains also conditional branches and many far answers for \textit{find\_open}, it is not unexpected that it takes around 800 nanoseconds. We observe a significant increase in runtime for \textit{cstY} when the sample parameter of their NSV-PSV-RMQ data structure is increased from 8 to 32.
Figure 4.2: Time and space for the \textit{parent}(v) operation of different CSTs. Note that the runtime has to be multiplied by 10 for each prime after the configuration number.
4.3 Our Test Results

![Graphs showing time and space for different operations of different CSTs.](image)

**Figure 4.3:** Time and space for the sibling(v) operations of different CSTs. Note that the runtime has to be multiplied by 10 for each prime after the configuration number.
Figure 4.4: Time and space for the \( \text{ith}_\text{child}(v, 1) \) operation of different CSTs. Note that the runtime has to be multiplied by 10 for each prime after the configuration number.
4.3.4 Depth-First-Search Traversals

The presented operations \textit{ith\_child}(v, 1), \textit{sibling}(v), and \textit{parent}(v) are the main ingredients for a constant space implementation of tree iterators. The depth-first-search traversal iterator just needs an additional status bit which indicates if the traversal should continue up- or downward the tree. To test the speed of the iterator we have set up two scenarios. In both scenarios we start with the first element in the traversal (which can be calculated with \texttt{it= cst\_begin()} in the \texttt{sdsl} implementation) and then increment the iterator 100000 times (\texttt{++it} in the \texttt{sdsl}). In the first — and less practical — scenario we calculate for each visited node \(v\) the very fast \textit{id}(v) operation, which takes only about 50 – 150 nanoseconds (see Figure 4.1). The results are depicted in Figure 4.5. We can observe that the average time per traversed node is much smaller than the minimum of the average times of the operations \textit{ith\_child}(v, 1), \textit{sibling}(v), and \textit{parent}(v). This can be explained as follows. We have a much higher locality of references in the depth-first-search traversal than at the generated node set \(V_{1}(100000)\). The effect is especially notable in the case of \texttt{cst\_sada}, since \texttt{BPS\_sect} is exactly constructed in depth-first-search order. Note that we have not included \texttt{cst\_Y} here, as it does not support \textit{id}(v).

In the second — more practical — scenario we calculate the \textit{depth}(v) operation for each inner node \(v\) during the traversal. The results are depicted in Figure 4.6. While in the previous scenario the runtime was only dependent on the choice of the \textit{CST} type — configurations (4)-(7) and (0)-(3) have the same runtime — it is now also dependent on the \texttt{LCP} structure.

Therefore every odd configuration \(i\), which uses \texttt{lcp\_support\_tree2}, takes now longer than the counterpart configuration \(i-1\), which uses \texttt{lcp\_dac}. For test cases with many small \texttt{LCP} values, like \texttt{rand\_k128}, the difference is barely notable, on \texttt{dblp.xml}, \texttt{dna} and \texttt{xml} where we have more big \texttt{LCP} values, which have to be recovered by using the \texttt{LF} function, the difference of the runtime increases up to a factor of 10. However, note that the runtime of both sets of configurations is clearly faster than \texttt{cst\_Y}, which also uses an implementation of the \texttt{LCP} array which is based on direct accessible codes.

For the test cases \texttt{english} and \texttt{proteins} the configurations (1) and (5) perform extremely bad, since they take between 2500 and 5000 nanoseconds per operation. This can be explained as follows: First, we have to look into Figure 3.27, which depicts the runtime for \texttt{lcp\_support\_tree2}. For our configuration with \(s\_LCP = 4\) we would get a runtime of about 1000 nanoseconds. But in Figure 3.27 we use \texttt{lcp\_support\_tree2} in combination with the \textit{CST} \texttt{cst\_wt\textlt wt\textgt\_>}, which provides fast access to LF (about 100 nanoseconds, see Figure 3.7)! In configuration (1) and (5) we use the class \texttt{cst\_sada} which takes about 2000 nanoseconds per LF entry and therefore slows down the runtime of the \texttt{lcp\_support\_tree2} by a remarkable factor.

We can conclude that the combination of the classes \texttt{csa\_sada} and \texttt{lcp\_support\_tree2} should never be used when we need fast access for any type of text while the combination of \texttt{csa\_wt} and \texttt{lcp\_support\_tree2} is relative robust in terms of runtime.
4.3.5 Calculating Matching Statistics

We will show in Chapter 6 how we can calculate matching statistics with a CST. The algorithm depends on the speed of the LF function and the parent\((v)\) operation. For details see [OGK10]. Our results for calculating the matching statistics of a text against its inverse are depicted in Figure 4.7. As in the previous experiment the runtime is clustered in two groups. The configurations which use a cst\_wt – i.e. (2),(3),(6), and (7) – and therefore answer LF fast and the configurations which use a cst\_sada – i.e. (0),(1),(4), and (5) – and therefore perform slower. Since parent\((v)\) is supported in faster time by cst\_sada the blue configurations perform better than their cst\_sct counterparts in red.

4.3.6 The \(child(v, c)\) Operation

The \(child(v, c)\) operation is the most complex operation in the suffix tree and the most important one for string matching with suffix trees\(^1\). The time consumption of the \(child(v, c)\) operation mainly depends on two algorithms:

(a) The recovering of the character \(c\) and

(b) the search of the child whose edge starts with character \(c\).

Configurations (1)-(4) use a linear search\(^2\) for (b), which first calls \(w_1 = ith\_child(v, 1)\), and then \(w_{i+1} = sibling(w_i)\) for at most \(\sigma\) siblings. Configurations (5)-(8) use a binary search on the children. We can see in Figure 4.8 that on small alphabets like DNA it does not matter if we use a linear or binary search, but for larger alphabets like in the dblp.xml test case it does matter. Here the cst\_sct solutions are always faster than the corresponding cst\_sada solutions which use the same parameters, i.e. (4) vs. (0), (5) vs. (1), (6) vs. (2), and (7) vs. (3).

You might wonder in the rand\_k128 case that configuration (3) is faster than (7). This is not true, since each prime sign after a number indicates that we have to multiply the depicted running time by 10. For example the running time of 3 in the rand\_k128 test case equals about 200 microseconds while that of 5 equals 20 microseconds.

A even bigger gap between the implementations can be found in the english and sources test cases. The implementation of [CN10] takes about 1 millisecond (!) while the sdsl implementations take only 7 – 30 microseconds. This can be explained by the implementation of part (a) which is basically the edge\((w_i, depth(v))\) operation. [CN10] decided to implement edge\((w_i, depth(v))\) by applying \(d = depth(v)\) times \(\Psi\) to \(lb(w_i)\) to get the position of the character in \(SA\) and then use a binary search on \(C\) to answer the question, i.e. they have a time complexity of \(O(\sigma \cdot t_{sibling} + \log \sigma \cdot (t_{edge} + \log \sigma))\) which depends linearly on \(d\)! So \(child(v, c)\) can only be answered fast when \(d\) is very small. You can see in Figure 4.8 that this is only the case for the dna.200MB test case.

---

1 Note, that we can do string matching in our CST also by backward search on the included CSA.

2 Note that we also get a logarithmic solution if an succinct implementation of median range minimum queries would be available.
Figure 4.5: Time and space for a depth-first-search traversal of the CSTs. The time is the average time per visited node. We have calculated at each node \( v \) the \( id(v) \) operation.
Figure 4.6: Time and space for a depth-first-search traversal of the CSTs. The time is the average time per visited node. We have calculated at each node $v$ the $\text{depth}(v)$ operation. Note that the runtime has to be multiplied by 10 for each prime after the configuration number.
4.3 Our Test Results

Figure 4.7: Time and space for computing matching statistics with different CSTs. The time corresponds to the average time per calculated matching statistics entry.
We opt for a robust variant of $\text{edge}(v, d)$: If $d > s_{SA} + s_{ISA}$ it is cheaper to calculate $\Psi^k[\ell b(w_i)]$ with Equation 2.7 in $s_{SA} + s_{ISA}$ constant operations. So in this case it pays off that we have introduced public members for $s_{SA}$ and $s_{ISA}$ in the $\text{csa}$ concept which now can be used from the CST implementation to optimize the speed of the edge operation. In summary we get a time complexity of $O(t_\Psi \cdot \min(s_{SA} + s_{ISA}, \text{depth}(v)))$ for calculating the edge operation in $\text{child}(v, c)$.

We would also like to mention that [CN10] uses a binary search to find the right child in their implementation as in our $\text{cst_sct3}$ solution, but before doing that they calculate all $\sigma$ children. This is another point where they can optimize their implementation.

4.3.7 Conclusions

Our first conclusion is quite contrary to that of [CN10], and can be formulated in one sentence: *Sadakane’s CST strikes back!* All presented configurations of $\text{cst_sada}$ beat the configurations of $\text{cst_sct3}$ and $\text{cstY}$ in runtime on almost all operations. Especially the navigational operations are very fast. This was not unexpected, since $\text{parent}(v)$, $\text{Sibling}(v)$, and $\text{ith}_\text{child}$ in $\text{cst_sada}$ correspond to core operations of $\text{bp_support}$, while in $\text{cst_sct}$ they require multiple calls to that helping data structure. The detailed time complexities in Figure 3.7 also show this.

Second, we can conclude that the new LCP representation $\text{lcp_support_tree2}$ in combination with $\text{cst_sct3}$ – e.g. configuration (7) – is of practical relevance, since it reduces significantly the space of the CST and beats $\text{cstY}$ (which uses more space for $\text{lcp}$) in common application scenarios (see Figure 4.6) in time and space.

Third, we would like to emphasize that this experimental study contains *nothing unexpected*, since the runtime of each operation is almost exactly the summation of the times of its operation calls to the subcomponents. This also implies that a runtime improvement of a subcomponent directly results in a faster runtime of the dependent CST operations. Some subcomponents – like $\text{rank_support_v}$ – have a very optimized runtime and we do not expect that we can further improve it. But it is still a challenge to optimize more complex structures like a $\text{wavelet_tree}$, which plays a key roll in CSTs. It would be very interesting if an implementation of the $k$-ary wavelet tree proposed in [FMMN07] can further improve the runtime of CST operations.

Finally, we remark that we can also beat the “small and slow” versions of $\text{cstY}$. We therefore first parametrize our CSTs by $\text{lcp_support_sada}$ to get the same space as the small version of $\text{cstY}$. Since the runtime of the navigation does not depend on $t_{LCP}$, we beat $\text{cstY}$ by one or two orders of magnitude in navigational operations. Only the time for the $\text{depth}$ operation lies in the same order of magnitude.
Figure 4.8: Time and space for the \textit{child}(v, c) operation for different CSTs. Note that the runtime has to be multiplied by 10 for each prime after the configuration number.
4.4 The Anatomy of Selected CSTs

In the last section we learned that the total space consumption of a suffix tree implementation which performs most operations in a few microseconds is about $1.5 - 3$ times the original text size. Now we will show how much space each component contributes to the total space of the CST. Since we also break down the space of each component to basic sdsl data structures we can also answer the question, how much space is occupied by the sub-linear data structures.

First, we inspect the two CST configurations (0) and (7) from the last section (see page 101). We have chosen to inspect these configurations, since they use many different substructures.

Figure 4.9 contains the information for configuration (7), which is a `cst_sct3` parametrized with a Huffman shaped wavelet tree and our new LCP data structure `lcp_support_tree2`. In total the CST takes 366.9 MB for the original text `english.200MB`, that is 183\% of the original text size. The CSA part (colored red) takes about half of the space (180 MB), the underlying Huffman shaped wavelet tree `wt_huff` takes 148 MB and the samples for SA and ISA values take 21.9 and 10.9 MB for a sample rate of $s_{SA} = 32$ and $s_{ISA} = 64$. The 148.0 MB of `wt_huff` can be divided into the `bit_vector` (114.0 MB), the `rank_support` (7.1 MB), and the `select_support` (21.9 MB) for the `select_support` data structures which support $b$.

The LCP component (colored yellow) takes only 96.1 MB of space. The member `LCP_SMALL rc`, which stores the small LCP values and is also realized by a Huffman

![Figure 4.9: Space consumption of the components of the CST configuration `cst_sct3<csa_wt<> lcp_support_tree2<>` for the text `english.200MB`. The CST part is colored in red, the LCP part is colored in orange, and the NAV part is colored in blue.](image-url)

```plaintext
<table>
<thead>
<tr>
<th>Component</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSA (csa_wt)</td>
<td>180.8 MB</td>
</tr>
<tr>
<td><code>wt_huff</code></td>
<td>148 MB</td>
</tr>
<tr>
<td><code>bit_vector</code></td>
<td>114 MB</td>
</tr>
<tr>
<td><code>rank_support_v5&lt;1&gt;</code></td>
<td>7.1 MB</td>
</tr>
<tr>
<td><code>select_support_mcl&lt;1&gt;</code></td>
<td>14.4 MB</td>
</tr>
<tr>
<td><code>select_support_mcl&lt;0&gt;</code></td>
<td>12.5 MB</td>
</tr>
<tr>
<td><code>sa_sample</code></td>
<td>21.9 MB</td>
</tr>
<tr>
<td><code>isa_sample</code></td>
<td>10.9 MB</td>
</tr>
<tr>
<td>LCP (<code>lcp_support_tree2</code>)</td>
<td>96.1 MB</td>
</tr>
<tr>
<td><code>LCP_SMALL rc (wt_huff)</code></td>
<td>72.1 MB</td>
</tr>
<tr>
<td><code>bit_vector</code></td>
<td>67.9 MB</td>
</tr>
<tr>
<td><code>rank_support_v5&lt;5&gt;</code></td>
<td>4.2 MB</td>
</tr>
<tr>
<td><code>LCP_BIG rc</code></td>
<td>24 MB</td>
</tr>
<tr>
<td><code>NAV</code></td>
<td>90.0 MB</td>
</tr>
<tr>
<td>BPS_{sct} (bit_vector)</td>
<td>50 MB</td>
</tr>
<tr>
<td><code>bp_support_sada</code></td>
<td>13.4 MB</td>
</tr>
<tr>
<td><code>small block</code></td>
<td>3.5 MB</td>
</tr>
<tr>
<td><code>medium block</code></td>
<td>0.8 MB</td>
</tr>
<tr>
<td><code>rank_support_v5&lt;1&gt;</code></td>
<td>3.1 MB</td>
</tr>
<tr>
<td><code>select_support_mcl&lt;1&gt;</code></td>
<td>5.9 MB</td>
</tr>
<tr>
<td><code>rc (bit_vector)</code></td>
<td>25 MB</td>
</tr>
<tr>
<td><code>rc rank (rank_support_v5&lt;1&gt;)</code></td>
<td>1.4 MB</td>
</tr>
</tbody>
</table>
```
shaped wavelet tree, takes only 72.1 MB. This is less then half of the space of the naive method, which stores each of the \( n \) small values in one byte and therefore needs 200 MB! This impressively shows how good the combination of using a Huffman shaped wavelet tree and storing only LCP values of inner nodes of the CST is. Note that we only need the rank functionality of the wavelet tree. Therefore, we parametrize the wavelet tree with two binary search select supports which use no additional space on top of the rank support. Since the rank functionality is only needed to index into the vector of sampled big values (LCP-big\(_{rc}\), 24 MB), and this is usually not often the case, we use a small rank support data structure, which takes only 4.2 MB.

Finally, the NAV data structure (colored blue) takes 90 MB. We spend 50 MB for BPS\(_{sct}\) and the bp support data structure for BPS\(_{sct}\) takes 13.4 MB. Most of the space of bp support sada, which supports BPS\(_{sct}\), is taken by the rank and select functionality (3.1 + 5.9 MB). The additional information for \texttt{find_close}, \texttt{find_open} and so on takes only 4.3 MB. Finally the \( n \) bits of \texttt{rc} occupy 25 MB, and the rank support v5 for \texttt{rc} takes 1.4 MB. Note, that we do not use a select support data structure for \texttt{rc}, although the PSV calculation in Algorithm 19 on page 84 uses a select query in line 03. However, this select query is only used to find the next set bit in \texttt{rc} left to position \texttt{rc_idx}. In case of Algorithm 19 we know that this position is at most \( \sigma \) positions left to \texttt{rc_idx} and so we can determine it very fast with bit-parallelism in words in practice.

Now we analyse the space consumption of the sub-linear structures in the CST. In Figure 4.9 the shaded areas correspond to rank and select data structures. The three select supports sum up to 32.8 MB and the four rank supports sum up to 15.8 MB. In addition to that we have 4.3 MB for the bp support data structure. So in total the sub-linear structures take only about 14.5 \% of the whole CST, which is a huge improvement compared to the implementation of Välimäki et al. We believe that the select support structure can be further optimized, i.e. we can reduce the space while the query time remains constant. The query time for rank support is already very close to the time for accessing an int vector and bit vector (cf. Figure 3.4) and therefore not much space for further optimizations is left.

Now let's switch to our second example. Figure 4.10 depicts a csa sada which is parametrized with the CSA class csa sada and with the LCP class lcp dac. The whole CST takes 232\% of the original text \texttt{english.200MB}. The CSA portion is this time the smallest one. It takes only 120.3 MB. Like in the previous example, we have chosen a sample rate of \( s_{SA} = 32 \) and \( s_{ISA} = 64 \) and therefore get the same space (21.9 + 10.9 MB) for the SA and ISA samples in the CSA. The rest of the CSA consists of the compressed \( \Psi \) function which is stored in an enc vector, which uses Elias-\( \delta \) code to encode differences of \( \Psi \) values in a bit vector \( z \) of size 75.7 MB. The second component of the enc vector consists of absolute samples of the \( \Psi \) function and pointers to \( z \) which takes for a sample rate of \( s_{\Psi} = 128 \) 11.7 MB. Note that csa sada does neither use a rank support nor a select support data structure.

The LCP component takes in this case the biggest portion of space: 215.0 MB. The key idea of lcp dac<4> is to use only 4 bits for values \( < 2^4 \), and 8 bits for values \( < 2^8 \), and so on. So every value \( x \) is divided into blocks of 4 bits. Then we only store blocks which contain set bits or which are followed by a block which contains set bits (170.4 MB). We
Figure 4.10: Space consumption of the components of the CST configuration \texttt{cst\_sada<csa\_sada<, lcp\_dac<> >} for the text \texttt{english.200MB}. Note that the space of each part T can be easily determined by the \texttt{sdsl} function \texttt{get\_size\_in\_bytes(T)}.

store for each block an overflow bit which indicates if the next block is stored. The space for this overflow indication is 41.9 MB, and the \texttt{rank\_support}, which tells us where the next block is stored takes only 2.6 MB.

The NAV component consists of the \texttt{BPS\_dfs}(80.9 MB), which is not much bigger than the 75.0 MB for \texttt{BPS\_sct} and \texttt{rc} of the previous CST. However the \texttt{bp\_support} structure is 60 \% bigger (21.9 vs. 13.4) since in the last example it was only built for the 50 MB of \texttt{BPS\_sct} and now we have to build it for the 80.9 MB of \texttt{BPS\_dfs}. In addition to that, we also have two additional rank and select data structures for the bit pattern “10” in \texttt{BPS\_dfs}. Since the rank query is used often in methods of \texttt{cst\_sada}, we opted for the use of the fastest \texttt{rank\_support} which takes 20.2 MB. So in total the NAV structure takes 129.2 MB, which is 43.5 \% more than the NAV structure of the previous CST.

Finally, we note that the space for the sub-linear data structures is again very small. We only spend about 10 \% of the whole CST space for sub-linear structures.
5 Construction

5.1 Overview

We have learned in the previous chapter that the space consumption of a CST usually lies between 0.5 and 3.5 times the text size. Therefore, we can now handle gigabytes of texts on a commodity personal computer with an in-memory CST whereas we can only handle a few hundred megabytes with the most space efficient suffix tree implementations. So the CST can be used in domains where the amount of data, which should be mined and processed, increases every day by an extensive rate. One example domain is the analysis of server log data, another is biological sequencing: In 2009, Voelkerding et al. [VDD09] reported that the output of one next-generation sequencing (NGS) platform is above one giga base pairs per day and very recently Bauer et al. [BCR11] stated that data sets of 100 giga base pairs or larger have become commonplace in NGS!

However, before we can use a CST to solve problems on such large sequences, we have to construct it! The naive way of doing that is to first compute the three components SA, LCP, and NAV in uncompressed form and then compress each component. This approach can be implemented in optimal linear time, since each component can be constructed in linear time for a constant alphabet size. However, we are faced with a real problem: the memory consumption of the algorithms. We will show this in the following quick tour through the state-of-the art of CST construction.

5.2 CST Construction Revisited

5.2.1 CSA Construction

The implementations of the first linear time suffix array construction algorithms (SACA) [KS03, KA03] in 2003 take about 10n bytes, i.e. in the worst case 20 times the size of the resulting CST. After that much work was done to improve the practical running time and space for SACAs. In 2007, Puglisi et al. [PST07] published a survey article which reviews 20 different SACAs which were proposed until 2007. The fastest in practice was at that time the implementation of Maniscalco and Puglisi which uses 5 – 6n bytes but has a theoretical worst case time complexity of $O(n^2 \log n)$. Algorithms which use only 5n bytes are also called lightweight algorithms. This term was coined by Manzini and Ferragina when they presented their first lightweight (only the text, SA, and some extra space are in memory) SACA [MF04]. Today the SACA implementation of Yuta Mori
called **libdivsufsort**\(^1\) is the fastest. This implementation has a theoretical worst case performance of \(O(n \log n)\) and uses \(5n\) bytes. Note that today there also exists a fast implementation of a worst case linear time SACA which is implemented by Yuta Mori as well and was proposed by Nong et al. [NZC09]. It also uses only \(5n\) bytes of memory and takes about twice the time of the **libdivsufsort** implementation.

After the construction of the **SA** array, we store **SA** to disk, as in the following we only access **SA** in sequential order and therefore can stream **SA** from disk. Next, we construct the corresponding **CSA** by first calculating \(T^{BWT}\) from **SA** using Equation 2.3 (note that we need only \(T\) in memory when we write \(T^{BWT}[i]\) directly to disk in each step) and second either (1) calculate the compressed \(\Psi\) function (for **csa_sada**) or (2) a wavelet tree (for **csa_wt**). Task (1) needs \(\Psi\) and \(T\) in main memory while \(T^{BWT}\) is streamed from disk\(^2\). Task (2) can also be done with constant space by streaming; or in space two times the text size in a straightforward implementation.\(^3\)

A **CSA** is then constructed by streaming **SA** another time and taking every \(s_{SA}\)-th entry for the **SA** samples and collecting the **ISA** entries when \(SA[i] \equiv 0 \mod s_{ISA}\).

An alternative way to construct the **CSA** is to first construct \(T^{BWT}\) directly from \(T\). Okanohara and Sadakane [OS07] recently presented an algorithm for this task which needs about \(2n\) bytes of space. Experiments in [BGOS11] showed that the runtime of their implementation is about twice the runtime of the fastest LACA for small (\(\leq 2\) GB) inputs of equal size. Second, one can construct the wavelet tree of the \(T^{BWT}\) and then use \(LF\) (or \(\Psi\)) to generate the **SA** and **ISA** samples in \(O(n \cdot t_{LF})\) (or \(O(n \cdot t_{\Psi})\)) time.

### 5.2.2 LCP Construction

Kasai et al. [KLA\(^+\)01] proposed the first linear time LCP-array construction algorithm (LACA), which does not use the suffix tree. Their algorithm uses \(T\), **SA**, **ISA**, and of course the **LCP**-array. So, in total \(13n\) bytes. The main advantage of their algorithm is that it is simple (it is based on Inequality 2.16, pseudo-code for a semi-external version is given in Algorithm 24) and uses at most \(2n\) character comparisons. But its poor locality behavior results in many cache misses, which is a severe disadvantage on current computer architectures. Mäkinen [Mäk03] and Manzini [Man04] reduced the space occupancy of Kasai et al.’s algorithm to \(9n\) bytes and Manzini showed that his solution has a slowdown of about \(5\% – 10\%\) compared to the \(13n\) bytes solution. He also proposed an even more space-efficient (but slower) algorithm that overwrites the suffix array. Recently, Kärkkäinen et al. [KMP09] proposed another variant of Kasai et al.’s algorithm which computes PLCP, i.e. the LCP array in text order. This algorithm takes only \(5n\) bytes and is much faster than Kasai et al.’s algorithm because it has a better locality of references. However, in virtually all applications LCP-values are required to be in suffix array order, so that in a final step the PLCP-array must be converted into the LCP-array. Although

---


2 In detail: we initialize a counter array \(cnt\) with \(C\) and then iteratively compute \(\Psi[cnt[T^{BWT}[i]]++] = i\) and after that stream \(\Psi\) to construct the compressed version using a self-delimiting coder.

3 See the implementation of the \(wt_*\) classes in the \(sdsl\) for details.
this final step suffers (again) from a poor locality behavior, the overall algorithm is still faster than Kasai et al.’s. They also proposed two semi-external versions of their algorithm which use only $5n + \frac{1}{d}n$ and $n + \frac{1}{d}$ bytes of space\(^1\) for an integer $d \geq 1$ and a running time of $O(nd)$. In a different approach, Puglisi and Turpin [PT08] tried to avoid cache misses by using the difference cover method of Kärkkäinen and Sanders [KS03]. The worst case time complexity of their algorithm is $O(nv)$ and the space requirement is $n + O(n/\sqrt{v} + v)$ bytes, where $v$ is the size of the difference cover. Experiments showed that the best run-time is achieved for $v = 64$, but the algorithm is still slower than Kasai et al.’s. This is because it uses a succinct data structure for constant time range minimum queries, which take considerable time in practice, cf. data structure $\text{succ}$ in Figure 6.2. To sum up, the currently fastest lightweight LACA is that of Kärkkäinen et al. [KMP09]. We propose in Section 5.3 a new lightweight linear time algorithm which is specially designed for the use in the CST construction, since it reuses temporary results of previous calculations in the CST construction process.

5.2.3 NAV Construction

The construction of NAV was a real problem in old implementations. First the construction of $\text{BPS}_{dfs}$ requires a traversal of the suffix tree. However, the explicit construction of the suffix tree would take about 20 times the size of the text. Therefore, Välimäki et al. [VMGD09] solved the problem by using a modified version of a suffix tree construction algorithm (the suffix-insertion algorithm) which constructs $\text{BPS}_{dfs}$ but not the suffix tree. However, the naive version of this algorithm uses $n \log n + o(n)$ bits. Hon and Sadakane [HS02] and Välimäki et al. suggested solutions to overcome this problem. But in practice these methods slow down the construction process. A second problem was that back in 2007 no real space economical $\text{bp\_support}$ data structure was available. Today, we can first construct $\text{BPS}_{sct}$ and then use the $\text{dfs\_iterator}$ of $\text{cst\_sct}$ to construct $\text{BPS}_{dfs}$.

5.2.4 Fast Semi-External Construction of CSTs

In this section we show how we can improve the running time and space consumption of the construction process by only using (1) the latest algorithms for the uncompressed data structures combined with streaming the uncompressed results, (2) the fast construction of the support data structures, and (3) the construction of $\text{BPS}_{dfs}$ through $\text{BPS}_{sct}$. Figure 5.1 contains the left the construction profile of Välimäki et al.’s solution [VMGD09], which we call $\text{cstV}^2$, and on the right the construction profile of our solution for 200 MB of data. Both construction processes generate a CST which consists of a CSA based on a wavelet tree, the LCP structure of Sadakane, and the $\text{BPS}_{dfs}$. The construction in Välimäki et al.’s implementation takes about 5000 seconds or 1 hour and 23 minutes while

---

1 Unfortunately, details of the implementation of these two versions were not covered in the publication and the source code was only available by request. Therefore, we first implemented the proposal not optimally, which first led to a biased comparison in [GO11].

2 Available under http://www.cs.helsinki.fi/group/suds/cst/cst_v_1_0.tar.gz.
ours takes about 400 seconds or 6 minutes. The following list will answer the question why our construction outperforms the old implementation in that extent.

- We use a newer and faster suffix array construction algorithm than [VMGD09]. They use the algorithm of Schürmann and Stoye [SS05], we use Yuta Mori’s libdivsufsort library.

- We use bit-parallelism in words to speed up the construction of rank and select data structures (see also page 39). The parts which most profit from this are the wavelet tree for the CSA and the \texttt{bp}_\texttt{support} for the NAV structure.

- We construct the LCP array by a semi-external algorithm, which streams the uncompressed SA array. Therefore the access to SA is fast. They also use a space-efficient algorithm, but they access \texttt{SA}[i] through the CSA which is really slow, cf. Section 3.7.4.

- Both the construction of \texttt{BPS}_\texttt{dfs} and \texttt{BPS}_\texttt{sct} can be done in linear time. We generate \texttt{BPS}_\texttt{sct} by streaming the LCP array and by only using a stack which uses at most

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{construction_profile.png}
\caption{Construction profile for a \texttt{cst}_\texttt{sada} of the \texttt{Pizza&Chili} test \texttt{english.200MB}. Left: Implementation of Välimäki et al., which we call \texttt{cstV}. Right: Implementation in the \texttt{sdsl}. We have colored each phase of the construction process to get an impression on which part it is worth to optimize.}
\end{figure}
5.2 CST Construction Revisited

$n + o(n)$ bytes during the construction (the stack was proposed by Fischer in [Fis10a]). $BPS_{dfs}$ can be generated from $BPS_{sct}$ and a corresponding $bp\_support$ in linear time. Välimäki et al. opted for a solution which simulates the suffix-insertion algorithm and accesses the compressed LCP array and therefore have a time complexity of $O(n\log n)$.

- We use $bp\_support\_sada$, which is our implementation of the range min-max-tree (see Section 3.8.2), since it can be constructed very fast by using bit-parallelism on words$^1$ for calculating the excess values. In contrast, detecting the pioneers in $bp\_support\_gg$ is expensive, since we have to use a (succinct) stack and we have conditional branches inside the for-loop. In addition, we also have to build the sparse $rank\_support$ and $select\_support$ for pioneers.

Figure 5.1 also shows that the construction process of the NAV structure is the most time consuming part in our solution. However, if we build not $cst\_sada$ but $cst\_sct3$, we can again accelerate the construction. Figure 5.2 shows that the construction of $cst\_sct3$ can be done in about half of the time of $cst\_sada$, i.e. in about 2 minutes and 30 seconds. This is over 20 times faster than the implementation of Välimäki et al. which was also optimized for speed! The speed-up can be mainly attributed to the faster construction of the NAV structure and the faster construction of the LCP array. We will present the new LCP construction algorithm in Section 5.3.

The next step is to compare our implementation with the construction of an uncompressed suffix tree. The MUMMER 3 software package$^2$ contains a space-efficient version of a suffix tree implemented by Kurtz [Kur99], which we call ST. Kurtz uses a linear time construction which uses no extra space on top of the resulting tree.

We have tested the construction on all Pizza&Chili files. We have taken prefixes of size 20 MB, 50 MB, 100 MB, 200 MB, ..., 500 MB to get an impression how the running times develop. The result for the prefixes of the english test case are depicted in Figure 5.3. The behavior for the other test cases is similar and therefore omitted here. We can see that the $cst\_sct3$ construction is faster than the construction of ST! This was not the case for all test cases, but on average the construction process is about equally fast. But the CST has a clear advantage: We can build trees for large texts while the uncompressed suffix tree implementation works only for test cases up to 300 MB on our test machine which is equipped with 4 GB of RAM. The process for the 400 MB test case ran quickly out of memory and was not finished after 7 days, which we set as time limit for the algorithm.

An interesting question is how fast suffix tree algorithms are, depending on the concrete suffix tree implementation. To get an impression of the answer, we have implemented an algorithm which contains typical elements of a suffix tree algorithm: It traverses the suffix tree in depth-first search order and calculates the depth at each inner node. So we execute the operations $ith\_child(v, 1)$, $sibling(v)$, $depth(v)$, and $parent(v)$. We executed

---

$^1$ For instance we use the operation `bit_magic::min_max_byte_excess`.

$^2$ The software is available under `http://mummer.sourceforge.net`. 
Figure 5.2: Construction profile for different CSTs for the Pizza&Chili test english.200MB. Left: Detailed version of the cst_sada construction which is also depicted in the right plot of Figure 5.1. A semi-external version of the LACA of Kasai et al. is used. In the middle and right plot we use our new LACA algorithm which is presented in Section 5.3. Right: We construct cst_sct3 instead of cst_sada.

1000000 steps for prefixes of the english.1000MB file of the Pizza&Chili corpus. Figure 5.4 contains the result of our experiment. The cstV takes about 2 microseconds per operation for small inputs and about 3 microseconds for large inputs. By using cst_sct3, the running time is about 10 times faster and the running time does not increase for larger test cases. The big gap between cst_sct3 and cstV is due to the used LCP representation. cst_sct3 is parametrized with lcp_dac in this example, while cstV uses the solution of Sadakane, which is at least an order of magnitude slower than lcp_dac (see Figure 3.26). Finally, the algorithm is about two times faster when ST is used instead of cst_sct3 while ST fits in main memory. So the message is clear: As long as there is enough main memory you can use ST or another uncompressed index data structure like an enhanced suffix array to get maximal speed. Otherwise you should configure a CST which fits in main memory and use it to solve your problem.
5.2 CST Construction Revisited

Figure 5.3: Construction time of four different suffix tree implementations on prefixes of the `english.1024MB` file of the Pizza&Chili corpus. Note that the construction for the uncompressed suffix tree for texts of size \( \geq 400 \) MB was not finished after 7 days on our PC equipped with 4 GB of RAM, since the data structure does not fit into RAM.

Figure 5.4: Average visit time of a node in a depth-first-search order traversal which includes the calculation of the current depth of the node. We use prefixes of the Pizza&Chili file `english.1024MB` as input. Note the logarithmic scale of the y-axis and that ST operates very slowly when its memory consumption exceeds the main memory size.
5.3 A LACA for CST construction

Before we present our new LACA, which is designed for the use in the CST construction process, we first inspect previous linear time LACAs which calculate LCP from SA and T.

5.3.1 Linear Time LACAs Revisited

We have already mentioned, that the key idea in the LACA of Kasai et al. [KLA+01] is to use Inequality 2.16 to calculate all values in amortized linear time. Therefore, they first calculate ISA from SA; see line 01 in the Algorithm 24. This is the first part of the algorithm, which is expensive since the write accesses are not in sequential order. In the main loop (lines 03-10) they calculate sequentially for each suffix \( i \) the lexicographically preceding suffix \( j = SA[ISA[i] - 1] \) (i.e. \( j = \Phi[i] \)) and store it in their original implementation to LCP[ISA[i]] (line 09). That is, we have to perform a random read access to SA and a random write access to LCP in each iteration. Furthermore, the first read access to T at position \( j = \Phi[i] \) is only cheap, when the value of \( j' = \Phi[i - 1] \) in the previous iteration is \( j - 1 \) and therefore \( T[j] \) is cached.

An important observation, which will help us to explain the practical runtime of the algorithm on different kind of texts, is the following: if LCP[ISA[i]] = PLCP[i] is reducible, then \( j = \Phi[i] = \Phi[i - 1] + 1 = j' + 1 \) and hence \( T[j] \) is cached.

**Algorithm 24** Kasai et al.’s LACA [KLA+01]. With streaming we get a 5n byte semi-external version, without streaming we get the 9n byte solution.

\[
\begin{align*}
01 & \textbf{for } i \leftarrow 0 \text{ to } n - 1 \text{ do } \text{ISA}[ISA[i]] \leftarrow i \text{ // stream SA from disk} \\
02 & \ell \leftarrow 0 \\
03 & \textbf{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
04 & \text{ISA}_i \leftarrow ISA[i] \text{ // stream ISA from disk} \\
05 & \textbf{if } ISA_i > 0 \text{ do} \\
06 & \quad j \leftarrow ISA[ISA_i - 1] \\
07 & \quad \textbf{while } T[i + \ell] = T[j + \ell] \\
08 & \quad \ell \leftarrow \ell + 1 \\
09 & \quad SA[ISA_i - 1] \leftarrow \ell \text{ // overwrite } SA[ISA_i - 1] \text{ with LCP[ISA_i]} \\
10 & \ell \leftarrow \max(\ell - 1, 0) \\
11 & SA[n - 1] \leftarrow -1 \\
12 & \text{// store overwritten SA as result to disk and shift indices by 1}
\end{align*}
\]

In 2009, Kärkkäinen et al. [KMP09] improved the practical runtime of Kasai et al.’s algorithm, essentially by removing the random write access to LCP. Algorithm 25 depicts how this is achieved. In line 01, they first compute all values of \( \Phi \). Note that this is as expensive as calculating ISA in Algorithm 24. In the main loop (lines 03-07) the read accesses to T are exactly the same as in Algorithm 24. However, the resulting LCP-values
are stored this time in sequential (text-)order to PLCP, i.e. we have sequential write accesses to PLCP, in contrast to Algorithm 24, where we have random write accesses to LCP. Finally, in line 09 LCP is constructed sequentially by \( n \) random read accesses on PLCP. Since Algorithm 24 has also \( n \) additional random read accesses to SA in line 06, we can conclude that the difference of the practical runtime of Algorithm 24 and Algorithm 25 is due to the removal of the random write access to LCP in Algorithm 25.

Algorithm 25 Kärkkäinen et al.’s LACA [KMP09]. With streaming and buffered writing to disk we get the \( 5n \) byte semi-external version, without streaming and using the same array for PLCP and \( \Phi \) we get the fast \( 9n \) byte version.

```plaintext
for \( i \leftarrow 0 \) to \( n - 1 \) do
  \( \Phi[SA[i]] \leftarrow SA[i - 1] \) // stream SA from disk

\( \ell \leftarrow 0 \)

for \( i \leftarrow 0 \) to \( n - 1 \) do
  \( \ell \leftarrow \max(\ell - 1, 0) \)
  while \( T[i + \ell] = T[\Phi[i] + \ell] \) do
    \( \ell \leftarrow \ell + 1 \)
  \( \text{PLCP}[i] \leftarrow \ell \)

for \( i \leftarrow 0 \) to \( n - 1 \) do
  \( \text{LCP}[i] \leftarrow \text{PLCP}[\text{SA}[i]] \) // stream SA from disk, write LCP buffered to disk
```

Kärkkäinen et al. [KMP09] also show how to reduce the space consumption and get a lightweight algorithm. The key idea is to first calculate LCP-values for a smaller set of suffixes. To be precise, all suffixes \( x \) with \( x \equiv 0 \mod q \) for a \( q \geq 1 \). In a second step, each remaining LCP value is calculated with at most \( q \) character comparisons. So the overall time complexity is \( O(nq) \). We present pseudo-code for the approach in Algorithm 26. In lines 01-03 a sparse \( \Phi \) array \( \Phi_q \) of length \( n/q \) is initialized. Then all LCP values of suffixes \( x \) with \( x \equiv 0 \mod q \) are calculated in lines 04-09. Note that the difference of the LCP value of suffix \( i - q \) and suffix \( q \) is always greater than or equal to \(-q\), since Inequality 2.19 is applied \( q \) times and we get

\[
\text{PLCP}[i] \geq \text{PLCP}[i - q] - q
\] (5.1)

So in each of the \( n/q \) iterations we make at most \( q - 1 \) more character comparisons than in the corresponding iteration in Algorithm 25. So in total the time for initializing \( \text{PLCP}_q \) is in \( O(n) \) and only \((4/q + 1)n\) bytes of space are required when we stream SA from disk.

Finally, in lines 10-16, LCP is calculated in sequential order as follows: For each pair of suffixes \( y = SA[i] \) and \( x = SA[i - 1] \) the leftmost sampled suffix \( y' \leq y \) is calculated by \( y = q[y/q] \). The LCP value of \( y' \) is stored in \( \text{PLCP}_q[y/q] \) and it is at most \( y - y' \) characters smaller than the LCP value for suffix \( y' \). So we make in each iteration at most \( q - 1 \) additional character comparisons than in the corresponding iteration in Algorithm 25. So we get a time complexity of \( O(qn) \) for the last step and the space remains at \((4/q + 1)n\) bytes when LCP is written buffered to disk. It is worth mentioning, that all of the additional character comparisons are in sequential order and therefore very fast in
practice. Hence the practical runtime does not depend linearly on \( q \).

Algorithm 26 Kärkkäinen et al.’s sparse LACA [KMP09]. It requires \( 4(1 + \frac{n}{q}) + n \) bytes when the same array is used for \( SA \) and LCP; i.e. \( SA \) is overwritten by LCP. The semi-external version, which streams \( SA \) and writes LCP buffered to disk, requires only \( n(1 + \frac{q}{4}) \) bytes.

```plaintext
1. for i ← 0 to n − 1 do
2.   if SA[i] mod q = 0 then
3.     \( \Phi_q[SA[i]/q] \leftarrow SA[i] \)

4. \( \ell \leftarrow 0 \)
5. for i ← 0 to \( \lfloor (n − 1)/q \rfloor \) do
6.   while \( T[iq + \ell] = T[\Phi_q[i] + \ell] \) do
7.     \( \ell \leftarrow \ell + 1 \)
8. \( \text{PLCP}_q[i] \leftarrow \ell \quad \text{// overwrite } \Phi_q \text{ with } \text{PLCP}_q \)
9. \( \ell \leftarrow \max(\ell − q, 0) \)

10. \((x, y) \leftarrow (0, 0)\)
11. for i ← 0 to n − 1 do
12.   \( (x, y) \leftarrow (y, SA[i]) \)
13.   \( \ell \leftarrow \max(0, \text{PLCP}_q[y/q] − (y − \lfloor y/q \rfloor \cdot q)) \)
14. while \( T[y + \ell] = T[x + \ell] \) do
15.     \( \ell \leftarrow \ell + 1 \)
16. \( \text{LCP}[i] \leftarrow \ell \)
```

5.3.2 Ideas for a LACA in the CST Construction

In the previous section we have seen that all LACAs have to make expensive random accesses to arrays. Our first observation is that in the context of CST construction some of the expensive random accesses have already been made before we construct the LCP array. One example is the calculation of \( T^{\text{BWT}} \) with Equation 2.3 from \( SA \) and \( T \), where we make \( n \) random accesses to \( T \). We can store \( T^{\text{BWT}} \) to disk and reuse this information in sequential order in our new algorithm. Note that we also get sequential access to \( LF \) for free, when \( T^{\text{BWT}} \) is present. We will also see, that we can deduce some LCP values with \( LF \) and \( \text{RMQs} \), which can be answered very fast in our algorithm in practice. Finally, we observe that the naive approach of calculating LCP values by comparing suffixes \( SA[i] \) and \( SA[i] − 1 \) for each \( i \) is fast when LCP\( [i] \) is small. Therefore, we have two phases in the algorithm. In the first one, we calculate all small LCP values \( \leq m \), and in the second phase, we insert the remaining big values.
The pseudo-code of the first phase of our LACA can be found in Algorithm 27. The main loop (lines 03-15) calculates the LCP array in sequential SA-order, but the naive computation of LCP[i] by comparing suffixes SA[i] with SA[i − 1] in line 10 has not to be done for every suffix! To exemplify that we take a look at Figure 5.5, which illustrates the application of the algorithm on our running example. In iteration j = 4 we calculate LCP[3] = 3 by comparing suffix SA[i − 1] = 11 with suffix SA[i] = 3. But since the characters before suffixes 11 and 3 are equal — i.e. $T^{BWT}[i − 1] = T^{BWT}[i]$ — we can deduce that LCP[11] = LCP[LF[3]] = 4 and insert that value in the same iteration into LCP. Note that this write access is not a random access, since LF points to at most σ regions in LCP! We will now show, that we can insert a second LCP entry in each iteration i in which LF[i] > i.

To do this, we first define a function prev by

$$\text{prev}(i) = \max\{j \mid 0 \leq j < i \text{ and } T^{BWT}[j] = T^{BWT}[i]\}$$

where $\text{prev}(i) = −1$ if the maximum is taken over an empty set. Intuitively, if we start at index $i − 1$ and scan the $T^{BWT}$ upward, then $\text{prev}(i)$ is the first smaller index at which the same character $T^{BWT}[i]$ occurs.

The following equation holds

$$\text{LCP}[\text{LF}[i]] = \begin{cases} 
0, & \text{if } \text{prev}(i) = −1 \\
1 + \text{LCP}[\text{RMQ}(\text{prev}(i) + 1, i)], & \text{otherwise}
\end{cases} \quad (5.2)$$

If $\text{prev}(i) = −1$, then $s_{\text{LF}[i]} = T[\text{LF}[i]..σ − 1]$ is the lexicographically smallest suffix among all suffixes having $T^{BWT}[i]$ as first character. Hence $\text{LCP}[\text{LF}[i]] = 0$. Otherwise,
We first push \( LF[prev(i)] = LF[i] - 1 \). In this case, it follows that

\[
\begin{align*}
  LCP[LF[i]] &= lcp(s_{SA}[LF[i]-1],s_{SA}[LF[i]]) = lcp(s_{SA}[LF[prev(i)]],s_{SA}[LF[i]]) \\
  &= 1 + lcp(T_{SA[prev(i)]},T_{SA[i]}) = 1 + LCP\[RMQ(prev(i) + 1,i)]
\end{align*}
\]

We will now show that Algorithm 27 correctly computes the LCP-array.

Under the assumption that all entries in the LCP-array in the first \( i-1 \) iterations of the for-loop have been computed correctly, we consider the \( i \)-th iteration and prove:

1. If \( LF[i] = \bot \), then the entry \( LCP[i] \) will be computed correctly.
2. If \( LF[i] > i \), then the entry \( LCP[LF[i]] \) will be computed correctly.

(1) If the if-condition in line 6 is not true, then \( s_{SA[i-1]} \) and \( s_{SA[i]} \) are compared character by character (lines 10-11) and \( LF[i] \) is assigned the correct value in line 12. Otherwise, if the if-condition in line 6 is true, then \( \ell \) is set to \( \max\{LCP[LF[i]] - 1, 0\} \). We claim that \( \ell \leq LCP[i] \). This is certainly true if \( \ell = 0 \), so suppose that \( \ell = LCP[LF[i]] - 1 > 0 \). According to (the proof of) Equation 5.2, \( LCP[LF[i]] - 1 = lcp(s_{SA[prev(i)]},s_{SA[i]}) \). Obviously, \( lcp(s_{SA[prev(i)]},s_{SA[i]}) \leq lcp(s_{SA[i-1]},s_{SA[i]}) = LCP[i] \), so the claim follows.

Now, if \( T_{BWT}[i] \neq T_{BWT}[i - 1] \), then \( s_{SA[i-1]} \) and \( s_{SA[i]} \) are compared character by character (lines 10-11), but the first \( \ell \) characters are skipped because they are identical. Again, \( LCP[i] \) is assigned the correct value in line 12. Finally, if \( T_{BWT}[i] = T_{BWT}[i - 1] \), then \( prev(i) = i - 1 \). This, in conjunction with Equation 5.2, yields \( LCP[LF[i]] - 1 = lcp(s_{SA[prev(i)]},s_{SA[i]}) = lcp(s_{SA[i-1]},s_{SA[i]}) = LCP[i] \). Thus, \( \ell = LCP[LF[i]] - 1 \) is already the correct value of \( LCP[i] \). So lines 10-11 can be skipped and the assignment in line 12 is correct.

(2) In the linear scan of the LCP-array, we always have \( last_\text{occ}[T_{BWT}[i]] = prev(i) \). Therefore, it is a direct consequence of Equation 5.2 that the assignment in line 14 is correct.

We still have to explain how the index \( j = \text{RMQ}(last_\text{occ}[T_{BWT}[i]] + 1,i) \) and the LCP-value \( LCP[j] \) in line 14 can be computed efficiently. To this end, we use a stack \( K \) of size \( O(\sigma) \). Each element on the stack is a pair consisting of an index and an LCP-value.

We push \((0,-1)\) onto the initially empty stack \( K \). It is an invariant of the for-loop that the stack elements are strictly increasing in both components (from bottom to top).

In the \( i \)-th iteration of the for-loop, before line 13, we update the stack \( K \) by removing all elements whose LCP-value is greater than or equal to \( LCP[i] \). Then, we push the pair \((i,LCP[i])\) onto \( K \). Clearly, this maintains the invariant. Let \( x = last_\text{occ}[T_{BWT}[i]] + 1 \).

The answer to \( \text{RMQ}(x,i) \) is the pair \((j,\ell)\) where \( j \) is the minimum of all indices that are greater than or equal to \( x \). This pair can be found by an inspection of the stack.

Moreover, the LCP-value \( LCP[x] + 1 \) we are looking for is \( \ell + 1 \). To meet the \( O(\sigma) \) space condition of the stack, we check after each \( \sigma \)-th update if the size of \( K \) is greater than \( \sigma \). If so, we can remove \( s - \sigma \) elements from \( K \) because there are at most \( \sigma \) possible queries. With this strategy, the stack size never exceeds \( 2\sigma \) and the amortized time for the updates is \( O(1) \). Furthermore, an inspection of the stack takes \( O(\sigma) \) time.
Algorithm 27 Construction of the LCP-array.

01 last_occ[0..σ-1] ← [-1, -1, ..., -1]
02 LCP[0] ← 0; LCP[LF[0]] ← 0
03 for i ← 1 to n - 1 do
04   if LCP[i] = ⊥ then // LCP[i] is undefined
05     ℓ ← 0
06     if LF[i] < i then
07       ℓ ← max{LCP[LF[i]] - 1, 0}
08       if T[BWT[i]] = T[BWT[i-1]] then
09         continue at line 12
10     while T[SA[i] + ℓ] = T[SA[i-1] + ℓ] do
11       ℓ ← ℓ + 1
12     LCP[i] ← ℓ
13     if LF[i] > i then
14       LCP[LF[i]] ← LCP[RMQ(last_occ[T[BWT[i]] + 1, i])] + 1
15 last_occ[T[BWT[i]]] ← i

practice, this works particularly well when there is a run in the \(T[BWT]\) because then the element we are searching for is on top of the stack.

Algorithm 27 has a quadratic run time in the worst case, consider e.g. the string \(T = ababab...ab\$.

At first glance, Algorithm 27 does not have any advantage over Kasai et al.’s algorithm because it holds \(T, SA, LF, T[BWT],\) and LCP in main memory. A closer look, however, reveals that the arrays \(SA, LF,\) and \(T[BWT]\) are accessed sequentially in the for-loop. So they can be streamed from disk. We cannot avoid the random access to \(T,\) but we can avoid the random access to LCP as we shall show next.

Most problematic are the “jumps” upwards (line 7 when \(LF[i] < i\)) and downwards (line 14 when \(LF[i] > i\)). The key idea is to buffer LCP-values in queues (FIFO data structures) and to retrieve them when needed.

First, one can show that the condition \(LCP[i] = ⊥\) in line 4 is equivalent to \(i ≥ C[F[i]] + rank(i + 1, T[BWT][i])\). The last condition can be evaluated in constant time and space (in detail: by \(σ\) counters \(c[t][0..σ - 1]\), one for each character, and we increment counter \(c[t][T[BWT][i]]\) in iteration \(i\), so it can replace \(LCP[i] = ⊥\) in line 4. This is important because in case \(j = LF[i] > i\), the value \(LCP[j]\) is still in one of the queues and has not yet been written to the LCP-array. In other words, when we reach index \(j\), we still have \(LCP[j] = ⊥\) although \(LCP[j]\) has already been computed. Thus, by the test \(i ≥ C[F[j]] + rank(i + 1, T[BWT][j])\) we can decide whether \(LCP[j]\) has already been computed or not.

Second, \(LF[i]\) lies in between \(C[T[BWT][i]]\) and \(C[T[BWT][i]] + rank(n, T[BWT][i])\), the interval of all suffixes that start with character \(T[BWT][i]\). Note that there are at most \(σ\) different such intervals. We exploit this fact in the following way. For each character \(c ∈ Σ\) we
use a queue $Q_c$. During the for-loop we add (enqueue) the values $\text{LCP}[C[e]], \text{LCP}[C[e] + 1], \ldots, \text{LCP}[C[e] + \text{rank}(n,c)]$ in exactly this order to $Q_c$. In iteration $i$, an operation $\text{enqueue}(Q_c, x)$ is done for $c = T^{\text{BWT}}[i]$ and $x = \text{LCP}[\text{RMQ}([\text{last_occ}[T^{\text{BWT}}[i]]], 1, i)] + 1$ in line 14 provided that $LF[i] > i$, and in line 12 for $c = F[i]$ and $x = \ell$. Also in iteration $i$, an operation $\text{dequeue}(Q_c)$ is done for $c = T^{\text{BWT}}[i]$ in line 7 provided that $LF[i] < i$. This dequeue operation returns the value $\text{LCP}[LF[i]]$ which is needed in line 7. Moreover, if $i < C[F[i]] + \text{occ}(T^{\text{BWT}}[i], i)$, then we know that $\text{LCP}[i]$ has been computed previously but is still in one of the queues. Thus, an operation $\text{dequeue}(Q_c)$ is done for $c = F[i]$ immediately before line 13, and it returns the value $\text{LCP}[i]$.

The space used by the algorithm now only depends on the size of the queues. We use constant size buffers for the queues and read/write the elements to/from disk if the buffers are full/empty (this even allows to answer an RMQ by binary search in $O(\log(\sigma))$ time). Therefore, only the text $T$ remains in main memory and we obtain an $n$ bytes semi-external algorithm.

5.3.4 The Second Phase

Our experiments showed that even a careful engineered version of Algorithm 27 does not always beat the currently fastest LACA [KMP09]. For this reason, we will now present another algorithm that uses a modification of Algorithm 27 in its first phase. This modified version computes each LCP-entry whose value is smaller than or equal to $m$, where $m$ is a user-defined value. (All we know about the other entries is that they are greater than $m$.) It can be obtained from Algorithm 27 by modifying lines 8, 10, and 14 as follows:

\begin{center}
\begin{verbatim}
08 if $T^{\text{BWT}}[i] = T^{\text{BWT}}[i - 1]$ and $\ell < m$ then
10 while $S[SA[i] + \ell] = S[SA[i - 1] + \ell]$ and $\ell < m + 1$ do
14 $\text{LCP}[LF[i]] \leftarrow \min\{\text{LCP}[\text{RMQ}([\text{last_occ}[T^{\text{BWT}}[i]]], 1, i)] + 1, m + 1\}$
\end{verbatim}
\end{center}

In practice, $m = 254$ is a good choice because LCP-values greater than $m$ can be marked by the value 255 and each LCP-entry occupies only one byte. Because the string $T$ must also be kept in main memory, this results in a total space consumption of $2n$ bytes.

Let $I = \{ i \mid 0 \leq i < n \text{ and } \text{LCP}[i] \geq m \}$ be an array containing the indices at which the values in the LCP-array are $\geq m$ after phase 1. In the second phase we have to calculate the remaining $n_I = |I|$ many LCP-entries, and we use Algorithm 28 for this task. In essence, this algorithm is a combination of two algorithms presented in [KMP09] that compute the PLCP-array: (a) the linear time $\Phi$-algorithm and (b) the $O(n \log n)$ time algorithm based on the concept of irreducible LCP-values. We have already introduced the term irreducible in terms of LCP and PLCP, cf. Equations 2.17 and 2.20.

Inequality 2.19 and Equation 2.20 have two consequences:

- If we compute an entry $\text{PLCP}[j]$ (where $j$ varies from 1 to $n - 1$), then $\text{PLCP}[j - 1]$ many character comparisons can be skipped. This is the reason for the linear run time of Algorithm 28; cf. [KLA+01, KMP09].
If we know that PLCP\(_{\text{SA}[i]}\) is reducible, then no further character comparison is needed to determine its value. At first glance this seems to be unimportant because the next character comparison will yield a mismatch anyway. At second glance, however, it turns out to be important because the character comparison may result in a cache miss!

Algorithm 28 Phase 2 of the construction of the LCP-array. (In practice \(SA[n-1]\) can be used for the undefined value \(\perp\) because the entries in the \(\Phi\)-array are of the form \(SA[i-1]\), i.e., \(SA[n-1]\) does not occur in the \(\Phi\)-array.)

```
01 \(b[0..n-1] \leftarrow [0,0,\ldots,0]\)
02 for \(i \leftarrow 0\) to \(n-1\) do
03 if \(LCP[i] > m\) then
04 \(b[SA[i]] \leftarrow 1\) // the \(b\)-array can be computed in phase 1 already
05 \(\Phi'[0..n] - 1] \leftarrow [\perp,\perp,\ldots,\perp]\)
06 initialize a rank data structure for \(b\)
07 for \(i \leftarrow 0\) to \(n-1\) do // stream \(SA\), \(LCP\), and \(T\) from disk
08 if \(LCP[i] > m\) and \(T[BWT][i] \neq T[BWT][i-1]\) then // PLCP[\(SA[i]\)] is irreducible
09 \(\Phi'[\text{rank}_b(SA[i])] \leftarrow SA[i-1]\)
10 \(j_i \leftarrow 0\)
11 \(\ell \leftarrow m + 1\)
12 PLCP'[0..n] - 1] \leftarrow [0,0,\ldots,0]\)
13 for \(j \leftarrow 0\) to \(n-1\) do // left-to-right scan of \(b\) and \(T\), but random access to \(T\)
14 if \(b[j] = 1\) then
15 if \(j \neq 0\) and \(b[j-1] = 1\) then
16 \(\ell \leftarrow \ell - 1\) // at least \(\ell - 1\) characters match by Equation 2.20
17 else
18 \(\ell \leftarrow m + 1\) // at least \(m + 1\) characters match by phase 1
19 if \(\Phi'[j_i] \neq \perp\) then //PLCP'[\(j_i]\] is irreducible
20 while \(T[j + \ell] = T[\Phi'[j_i] + \ell]\) do
21 \(\ell \leftarrow \ell + 1\)
22 PLCP'[\(j_i]\] \(\leftarrow \ell\) // if PLCP'[\(j_i]\] is reducible, no character comparison was needed
23 \(j_i \leftarrow j_i + 1\)
24 for \(i \leftarrow 0\) to \(n-1\) do // stream \(SA\) and \(LCP\) from disk
25 if \(LCP[i] > m\) then
26 \(LCP[i] \leftarrow \text{PLCP'[\text{rank}_b(SA[i])]}\)
```

Algorithm 28 uses a bit_vector \(b\), where \(b[SA[i]] = 0\) if \(LCP[i]\) is known already (i.e., \(b[j] = 0\) if PLCP\([j]\) is known) and \(b[SA[i]] = 1\) if \(LCP[i]\) still must be computed (i.e., \(b[j] = 1\) if PLCP\([j]\) is unknown); see lines 1–4 of the algorithm. In contrast to the
Φ-algorithm [KMP09], our algorithm does not compute the whole Φ-array (PLCP-array, respectively) but only the $n_I$ many entries for which the LCP-value is still unknown (line 5). So if we would delete the values $\Phi[j]$ (PLCP$[j]$, respectively) for which $b[j] = 0$ from the original Φ-array (PLCP-array, respectively) [KMP09], we would obtain our array $\Phi'[0..n_I - 1]$ (PLCP$'[0..n_I - 1]$, respectively). We achieve a direct computation of $\Phi'[0..n_I - 1]$ with the help of a rank data structure, like rank$\_support\_v$, for the bit array $b$. The for-loop in lines 7–9 fills our array $\Phi'[0..n_I - 1]$ but again there is a difference to the original Φ-array: reducible values are omitted! After initialization of the counter $j_I$, the number $\ell$ (of characters that can be skipped), and the PLCP$'$ array, the for-loop in lines 13–23 fills the array PLCP$'[0..n_I - 1]$ by scanning the $b$-array and the string T from left to right. In line 14, the algorithm tests whether the LCP-value is still unknown (this is the case if $b[j] = 1$). If so, it determines the number of characters that can be skipped in lines 15–18.1 If PLCP$'[j_I]$ is irreducible (equivalently, $\Phi'[j_I] \neq \perp$) then its correct value is computed by character comparisons in lines 20–21. Otherwise, PLCP$'[j_I]$ is reducible and PLCP$'[j_I] = \text{PLCP}'[j_I - 1] - 1$ by Equation 2.20. In both cases PLCP$'[j_I]$ is assigned the correct value in line 22. Finally, the missing entries in the LCP-array (LCP-values in SA-order) are filled with the help of the PLCP$'$-array in lines 24–26.

Clearly, the first phase of our algorithm has a linear worst-case time complexity. The same is true of the second phase as explained above. Thus, the whole algorithm has a linear run-time.

5.3.5 Experimental Comparison of LACAs

We have implemented semi-external versions of the algorithm presented in [KLA+01] and [KMP09] in the $sdsl$. In the whole construction process of a CST we use bit-compressed int\_vectors since we want to avoid to double the construction space for files which are slightly larger than 4 GB. Unfortunately it turned out, that the use of bit-compressed int\_vectors in the algorithms of [KLA+01] and [KMP09] results in a slowdown of about a factor of 2, while this is not the case for our new algorithm. Therefore, we now compare the implementation of our algorithm, which is called go-Φ, with the original implementations of [KMP09] and a reimplementation of [KLA+01]. Both implementations use 32 bit (signed) integer arrays and therefore are limited to input files of 2 GB. The implementation of the algorithm of [KLA+01] is denoted with KLAAP and the various implementations of [KMP09] are all prefixed with Φ. Here are details:

- Φ-optimal: The fastest version of the Φ algorithm, which uses $9n$ bytes; see Algorithm 25 for pseudo-code.

---

1 Note that $\ell$ in line 18 in the $j$-th iteration of Algorithm 28 is at least $m + 1$, which is greater than $\ell$ in line 04 in the $j$-th iteration of Algorithm 25. Therefore, Algorithm 28 has also a linear time complexity.

2 The code was provided by Juha Kärkkäinen.
5.3 A LACA for CST construction

- **Φx**: Worst case time $O(xn)$ version of the Φ algorithm, which takes only $4n/q + n$ bytes of space, since SA is in memory and is overwritten by LCP; see Algorithm 26.

- **Φx-semi**: Same approach as Φx, but this time in the last step SA is streamed from disk and the result is written buffered to disk; see Algorithm 26.

We used inputs from the *Pizza&Chili* corpus and Giovanni Manizini’s text corpus\(^1\) and in addition some larger DNA sequences. Since the performance of lightweight LACAs is only interesting for large test cases, we present only the results for the 200 MB of the *Pizza&Chili* text corpus and a DNA sequence of 1828 MB. For small test cases like in Manzini’s test corpus the $n$-bytes variant of the Φ algorithm is always the fastest variant.

Since the main memory of our test server was smaller than the space consumption of most of the LACAs for the large genome sequences, we used another server for the experiments: It was equipped with a six-core AMD Opteron\(^\text{TM}\) processor 2431, which runs on 2.4 GH on full speed mode, and 32 GB of RAM. The programs were compiled using gcc version 4.4.3 with options -O9 -DNDEBUG under a 64 bit version of Ubuntu 10.4 (Kernel 2.6.32).

Table 5.1 contains the results of the experiments. The first two rows serve as baselines: The first row depicts the time for the SA construction by using libdivsufsort, the second row depicts the time for calculating $T^{\text{BWT}}$ from SA by Equation 2.3. We replaced the expensive modulo operation by a boolean expression and a lookup table of size 2. Furthermore only SA is in memory and the result is written buffered to disk. Therefore the space consumption is about 1.0 byte per input symbol.

Now we first put our attention to the space-consuming algorithms Φ-optimal, Φ1, and KLAAP. We first observe that Φ-optimal outperforms KLAAP. It is about 1.5 times faster in all cases. This can be explained by the observations which we have already made in Section 5.3.1: In KLAAP (see Algorithm 24) we have $n$ write accesses in ISA-order in line 09, which are substituted in Φ-optimal (see Algorithm 25) by $2n$ sequential write accesses in line 06 and 09. So in total, we save computing time since a sequential write access is an order of magnitude faster than a random write access, cf. Figure 3.3. Moreover, Φ-optimal has a better locality in its main loop than KLAAP, since only 2 arrays are accessed when we use the same array for Φ and PLCP. In case of KLAAP three arrays are accessed in the main loop. In terms of Algorithm Engineering we can say that the two loops in lines 03-07 and 08-09 in Algorithm 25 are a *loop fission*\(^2\) of the main-loop in Algorithm 24.

The next observation is that Φ-optimal and KLAAP both perform significantly better for the test cases *dblp.xml.200MB* and *sources.200MB* compared to the rest of the *Pizza&Chili* files. This can be explained as follows: Both algorithms compare in each iteration of their main loop the (a) first character of suffix $i + \ell$ with the (b) first character of suffix $j = \Phi[i] + \ell$ (cf. line 07 in Algorithm 24 and line 04 in Algorithm 25). Since the value of $\ell$ increases only rarely in consecutive iterations of the loop, the text is accessed

---

2. See [KW02] for Algorithm Engineering techniques.
Table 5.1: Runtime in seconds and peak memory consumption in bytes per input symbol for different construction algorithms and inputs. The runtime is the average runtime of 10 runs of the construction algorithms. \textit{O. Anatinus} denotes the DNA sequences of the species Ornithorhynchus Anatinus (we have removed all N-symbols from the sequence).

Almost in sequential order in case (a). For (b) the access in two consecutive iterations \(i - 1\) and \(i\) is in sequential order, when \(\Phi[i] = \Phi[i - 1] - 1\) is true, which is equivalent to \(\text{T}_{\text{BWT}}[i] = \text{T}_{\text{BWT}}[i - 1]\) and therefore \(\text{PLCP}[i]\) is reducible. We have depicted the ratio of reducible values in the test cases at the bottom of Table 5.1. We can see that test cases with a high ratio of reducible values are processed faster than test cases with a low ratio.

The same observation is also true for the sparse variants of the \(\Phi\)-algorithms. Also note that \(\Phi_1\) takes about double the time of \(\Phi\)-optimal, since many expensive division and modulo operations are used in the sparse version of the algorithm. Therefore, the calculation of the \(\Phi_1\) array and PLCP\(_1\) array in \(\Phi_1\) – i.e., the initialization phase in lines 01-09 in Algorithm 26 – takes about as long as \(\Phi\)-optimal in total.

Now let us switch to the lightweight algorithms which use \(2n\) bytes or less for the computation. Since our new solution uses the \(\text{T}_{\text{BWT}}\) for the construction process, we introduced two rows. One includes the time for the \(\text{T}_{\text{BWT}}\) computation (\(\text{go-}\Phi + \text{T}_{\text{BWT}}\)) and one excludes the time, as in our special case of CST construction the \(\text{T}_{\text{BWT}}\) is calculated anyway for the construction of the CSA. Note that the computation of the \(\text{T}_{\text{BWT}}\) results in almost the same runtime for texts of equal size. The explanation of the runtime of the new algorithm for the test cases is more complex, since we use many techniques to speed up the process. In the main loop of the first phase (see Algorithm 27) the first accesses to the text in line 10 are expensive, as they are in \textit{SA}-order. However, line 10 is only reached in 32 – 41 percent of the iterations for the test cases, since we either have already computed the LCP value at position \(i\) (see line 04) or \(\text{LF}[i] < i\) and the value is reducible. The runtime of the first phase is also correlated with the number of character comparisons: The algorithm performs about 20 comparisons per input character for the test cases \textit{proteins.200MB} and \textit{english.200MB}, 6 for \textit{dna.200MB}, and 13 for the others.
So it is not surprising, that the first phase takes about 32 seconds for proteins.200MB and english.200MB and between 24 – 26 for the remaining test cases.

The left plot in Figure 5.6 depicts the ratio of small values, which are computed during the first phase of the algorithm. In the second phase, the initialization of the arrays b and \( \Phi \) in lines 01-09 takes between 7 – 10 seconds. It might be possible to further optimize that runtime for test cases where only a small fraction of entries are left. The runtime of the remaining part of the algorithm is more input-dependent and takes less time for inputs with smaller \( n_f \).

We can conclude that our new solution performs significantly better compared to the \( \Phi \)-semi variants when the percentage of reducible values is not very high, which is often the case for DNA or protein sequences. Note that our algorithm outperforms even the \( \Phi \)-optimal algorithm when the DNA sequences are large enough, as the DNA sequence of Ornithorhynchus Anatinus in Table 5.1. So in case of the CST construction, where we do not count the time for \( T_{BWT} \) construction, our new approach takes only 431 seconds while \( \Phi \)-optimal uses 4.5 times of our space and 676 seconds! This shows impressively the results of our Algorithm Engineering efforts.

We believe that a further improvement of runtime is possible. An interesting observation, which was made during the analysis of the new algorithm, has not yet been exploited: The percentage of irreducible LCP values of value \( \geq i \) drops from 61 – 14\% for \( i = 0 \) to below 1\% for \( i \in [20..100] \) in all test cases. That is, a possible fast strategy is to calculate the small irreducible LCP values with the naive method, and the reducible LCP values are then filled in with the \( \Phi \) approach.

![Graph](image.png)

**Figure 5.6**: Density of the LCP values for the Pizza\&Chili test cases of size 200 MB.
6 Applications

In Chapter 3 we provide a clear concept for a CST and we can easily configure myriads of CSTs with different time-space trade-offs which all provide full functionality. The full functionality combined with features like constant time and space depth-first-search and bottom-up iterators makes it easy to implement algorithms (even complex ones), which rely on a suffix tree, almost verbatim. So the sdsl is an ideal tool for rapid prototyping of algorithms in domains like sequence analysis and data compression. We provide a small example in Section 6.1 for this feature. It is also easy to replace a suffix tree in an existing application by a CST to reduce the space usage immediately. In most cases this will slow down the application, except if we have the case that the CST fits in main memory but the suffix tree does not. However, the ultimate goal of our research is to replace suffix trees by CSTs and at the same time improve the runtime of the application.

To achieve this goal we cannot simply consider the CST as a black box, i.e. as a data structure which answers all its operations somehow efficiently from the viewpoint of a theoretician, i.e. in appealing time complexities like $O(1), O(\log n)$ and so on. We have to take care about the constants in each operation of our application! But before doing so, we have to know the constants. Fortunately Chapter 3 and 4 provide information about the practical runtime of almost all data structures and their operations (see for example Figure 4.1 on page 104). So the general plan is to avoid using slow operations like the $[i]$-operator of the CSA or the suffix link operation $sl(v)$ of the CST and try to redesign algorithms in such a way that we can use faster operations like LF or $\Psi$ of the CSA or $parent(v)$ of the CST.\footnote{Note that we follow the approach by designing lcp\_support\_sada in Section 3.10.2.} Sometimes we can also speed up an algorithm by spending a little more memory for a part of the CST, that is heavily used. For instance, it was often the case that the replacement of the $2n + o(n)$ bit LCP representation lcp\_support\_sada by a more space consuming version (e.g. lcp\_kurtz) was enough to be competitive with another solution.

We have shown in [OGK10] that this approach works for

- the computation of matching statistics,
- and the computation of maximum exact matches.

Many other problems, for instance finding longest palindromes of length $\geq \ell$, can be also tackled with this approach. We will exemplify the approach by presenting the solution
for the calculation of maximum exact matches in this chapter. But first we show two applications which can be solved effortlessly by using data structures of the sdsl library.

6.1 Calculating the \( k \)-th Order Empirical Entropy

The first application is to efficiently generate Table 2.1 on page 8 which contains the \( k \)-th order empirical entropy of the Pizza&Chili test cases. Let us first recall the definition of the \( k \)-th order entropy (see Equation 2.2):

\[
H_k(T) = \frac{1}{n} \sum_{W \in \Sigma^k} |W_T| H_0(W_T)
\]

where \( W_T \) is a concatenation of all symbols \( T[j] \) such that \( T[j-k, \ldots, j-1] = W \). A naive calculation of this sum results in a time complexity of \( O(\sigma^k n) \). We will now show how to get the optimal \( O(n) \) solution.

First note that \( H_k(T) \) does not depend on the order of the characters in \( W_T \), as the order neither affects the length of \( W_T \) nor the zeroth order entropy \( H_0(W_T) \), i.e. it suffice to determine the distribution of characters in each \( W_T \).

The calculation of \( H_k \) can be divided into three task. First, we build the (compressed) suffix tree in linear time. Second, we have to find all contexts \( W \) of length \( k \) in the suffix tree and then we have to calculate \( |W_T| \) and \( H_0(W_T) \). Finding all contexts can be done by selecting all nodes \( v \) with \( \text{depth}(v) \geq k \) and no other node \( w \) on the path from the root to \( v \) was also selected. Figure 6.1 depicts the selected nodes for the calculation of \( H_1 \) in our running example. Note, that the subtrees rooted at the selected nodes cover all leaves in the tree. Now we are ready for the final step, which computes for each selected node its contribution to the final result \( H_k \). We distinguish two cases for the calculation of the contribution, which corresponds to the zero order entropy of the context \( W_T \) of the node.

Case (a): \( \text{depth}(v) = k \). In this case we can determine \( |W_T| \) and the character distribution of \( W_T \) in the following way. \( |W_T| \) corresponds to the size of the subtree rooted at \( v \). The number of different characters in \( W_T \) corresponds to the number \( j \) of children \( v_1, v_2, \ldots, v_j \) of \( v \). The distribution for \( H_0 \) consists of the subtree sizes of \( v_1, v_2, \ldots, v_j \). Note that we can compute the number of children and each subtree size in constant time.

Case (b): \( \text{depth}(v) > k \) and \( \text{depth}(w) < k \) for all other nodes \( w \) on the path from the root to \( v \). In this case the context of length \( k \) is always followed by the same character. E.g. in Figure 6.1 take node \( u \) which is at the end of the path labeled with \( \text{lmu} \) and is marked as selected node. \( \text{depth}(u) = 3 \) and the context \( W \) of length \( k = 1 \) equals \( 1 \). \( W = 1 \) is always (two times to be precise) followed by character \( u \) in the text. Therefore \( H_0(11) = 0 \). This observation is also true for all other nodes with \( \text{depth}(v) > k \) and therefore all such \( v \)s do not contribute to the sum of \( H_k \).

The resulting algorithm can be implemented efficiently in a few lines using the sdsl. Listing 6.1 shows the code. We will briefly explain it. Lines 1-15 contains the main method, which expects a CST and the parameter \( k \). We use the depth-first-search iterator
Figure 6.1: Calculation of $H_1(T)$. The green nodes correspond to the contexts. 

$$H_1(T) = \frac{1}{10}(5H_0([1,4]) + 6H_0([2,3,1])) = \frac{1}{10}(5(1.5 \log 5 + 1.5 \log 4) + 6(3 \log 3 + 1.5 \log 6)) \approx 0.773$$

of the CST in line 3 to traverse through the tree. In line 4, we check if it is the first visit of the node, and if this is true we check if the depth $d$ of the node is greater or equal than $k$. If it is equal to $k$ we are in case (a) and have to calculate the contribution of the current node to the result. $|W_T|$ is calculated by the leaves in the subtree $(v)$ and $H_0(W_T)$ by the method $H_0$, which expects a reference to the current node $v$ and the CST (see lines 17-31). If $v$ is a leaf, then $H_0$ is 0 by definition. Otherwise, we use the operations $ith\_child(v, 1)$, $leaves\_in\_the\_subtree(v)$, and $sibling(w)$ to calculate $H_0(W_T)$.

The overall time complexity is $O(n)$, since the traversal takes at most $2n$ steps, the selected nodes have at most $n$ child nodes, and each performed operation takes constant time.

Note that it is also possible to use a bottom-up traversal of the CST to calculate $H_k$ in linear time. However by using a depth-first-search order traversal we can skip many subtrees (cf. line 9 in Listing 6.1) if $k$ is small. This is not possible in the bottom-up traversal.
Listing 6.1: Linear time computation of $H_k$ using the sdsl.

```cpp
double Hk(const tCST &cst, size_type k){
    double hk = 0;
    for(const_iterator it = cst.begin(); it != cst.end(); ++it){
        size_type d = cst.depth(*it);
        if( d >= k ){
            if( d == k )
                hk += cst.leaves_in_the_subtree(*it) * H0(*it, cst);
            it.skip_subtree();
        }
    }
    hk /= cst.size();
    return hk;
}

double H0(const node_type &v, const tCST &cst){
    if( cst.is_leaf(v) ){
        return 0;
    } else{
        double h0=0;
        size_type n = cst.leaves_in_the_subtree(v);
        node_type w = cst.ith_child(v, 1);
        do{
            double p = ( (double)cst.leaves_in_the_subtree(w))/n;
            h0 -= p*log2(p);
            w = cst.sibling(w);
        } while( w != cst.root() );
        return h0;
    }
}
```

6.2 Succinct Range Minimum Query Data Structures

In this section, we will show how easy it is to create a stand-alone succinct data structure which can answer range minimum queries (RMQs) on an array $A$ in constant time by using a bp_support data structure. The succinct data structure will take only $2n + o(n)$ bits of space and we compare its implementation with the implementation of previous succinct RMQ solutions of Fischer [FH07, FHS08, Fis10a] and Sadakane [Sad07b].

We will briefly present the history of succinct RMQ data structures. In 2002 Sadakane quietly introduced a stand-alone\footnote{See page 35 for the definition of the terms stand-alone and non-stand-alone.} $4n + o(n)$ data structure in the conference version...
Applications of [Sad07a] and this solution is also described in [Sad07b]. In 2007, Fischer and Heun [FH07] presented a non-stand-alone $2n + o(n)$ solution and Fischer provided a practical implementation which uses $7n$ bits on top of the input array $A$ and also requires $A$ to answer queries. In 2008, Fischer et al. [FHS08] improved the space of their solution to $nH_k + o(n) + |A|$ bits. However in practice (e.g. for the LCP array of the test case english) the space was not reduced compared to [FH07], since the lower order terms dominate the space consumption. However, we will see in the experiment that the use of sdslib data structures in their solution lowers the space consumption significantly. In 2008, Fischer introduced a stand-alone $2n + o(n)$ bits solution which is based on the 2d-min-heap data structure [Fis10a]. We will see that its practical implementation still needs about $4n$ bits. Finally in 2010, Sadakane and Navarro [SN10] and Gog and Fischer [GF10] published their bp_support_data structure, which can be used to realize the following solution.

First we build the $BPS_{set}$ of $A$ and then construct a bp_support for $BPS_{set}$. Remember that this can be done in linear time by using only $n + o(n)$ bits on top of $A$ and $BPS_{set}$. The pseudo-code for answering the query $RMQ(i, j)$ is depicted in Algorithm 29. Note that it consists only of constant time queries to the bp_support structure, and therefore stand-alone! So the whole solution takes about $3n$ bits in practice ($2n$ bits for $BPS_{set}$, 0.5n bits for $rank_{support_v}$, and 0.5n bits for $select_{support_mcl}$).

Algorithm 29 Calculation of $RMQ(i, j)$ in constant time using $BPS_{set}$.

```
01 if $i = j$ then
02    return $i$
03  $ipos \leftarrow select_{BPS_{set}}(i + 1)$
04  $jpos \leftarrow select_{BPS_{set}}(j + 1)$
05  $cpos \leftarrow find_{close}_{BPS_{set}}(ipos)$
06  if $jpos < cpos$ then
07    return $i$
08  else
09    $ec \leftarrow rr_{enclose}_{BPS_{set}}(ipos, jpos)$
10   if $ec = \bot$ then
11      return $j$
12  else
13    return $rank_{BPS_{set}}(ec)$
```

It is not hard to show the correctness of Algorithm 29: If $i$ equals $j$ then the answer is trivially $i$. Otherwise (if $j > i$) we select the positions $ipos$ and $jpos$ of the opening parentheses which represent element $i$ and $j$ in the array (line 3 and 4) and get the matching closing parentheses $cpos$ of $ipos$. Next we check, if $cpos$ is right of $jpos$. If yes, then all elements between $i$ and $j$ are greater than or equal to $A[i]$ (from the definition of $BPS_{set}$), and therefore we return $i$. Otherwise we call $ec = rr_{enclose}(ipos, jpos)$.
6.2 Succinct Range Minimum Query Data Structures

Remember that $ec \neq \perp$ equals the leftmost opening parenthesis in the range $[i+1, \ldots, j-1]$ whose matching parenthesis is greater than $j$. The corresponding element at array position $k = \text{rank}(ec)$ is the smallest element in $A[i..j]$ since (1) all elements in $A[k+1..j]$ are greater than or equal to $A[k]$, because the parentheses pair of $A[k]$ encloses all these elements and (2) all elements in $A[i..k-1]$ are greater than $A[k]$, as otherwise the parentheses pair of a smaller element $k' \in [i+1..k-1]$ would enclose the pair of $k$, which is a contradiction to the result of $\text{rr_enclose}(i\text{pos}, j\text{pos})$.

In the case that $ec = \perp$ holds, there is no $k \in [i+1..j-1]$ whose parentheses pair encloses the pair of $j$. Therefore all these elements are greater than $A[j]$ and $j$ is the right answer.

In the sdsl, the class \textit{rmq_succinct_sct} implements the Algorithm 29. The class can be parametrized with a \textit{bp_support} data structure and therefore profits directly from improvements on the \textit{bp_support} implementation.

Before we come to the experimental results, we first have to shortly review the succinct solutions of Sadakane [Sad07b] and Fischer [Fis10a]. Both solutions reduce the RMQ query to a so called $\pm 1$RMQ. This is a special case of RMQ which expects that the difference of two consecutive elements in $A$ is either $+1$ or $-1$. So exactly the property of the virtual array of excess values of a balanced parentheses sequence! Both use a $o(n)$ data structure for answering these $\pm 1$RMQs on top of their balanced parentheses sequences. However, they also use extra $o(n)$ data structures for answering other basic queries like \textit{rank}, \textit{select}, and \textit{find_open}. It turned out, that these two $o(n)$ data structures contain redundant information. Now the good message is that it also turned out that the $\pm 1$RMQ corresponds to the $\text{rr_enclose}$ operation and therefore we now only need one $o(n)$ data structure — a \textit{bp_support} — to answer RMQs! With this in mind, we present the experimental results.

The implementation of the non-stand-alone solution of Fischer and Heun [FH07] is called \textit{succ}, and its compressed version [FHS08] which was improved with sdsl data structures [GF10] is called \textit{compr}. Furthermore, \textit{2dmin} denotes the implementation of the stand-alone data structures of Fischer [Fis10a] and \textit{sada_old} the implementation of Sadakane’s proposal [Sad07b]. Finally, \textit{sct} equals \textit{rmq_succinct_sct} parametrized with \textit{bp_support}\_\textit{gg}.

We ran tests on random integer arrays and on LCP arrays of all texts of size 50 MB and 200 MB from the Pizza&Chili corpus. Since the results were all rather similar, we only display those for two LCP arrays of the test case \textit{english.200MB}.

The left bar plot of Figure 6.2 shows the final space of the solution. For a deeper analysis of the stand-alone schemes, we partition the bars into different components:

\textbf{2dmin}: From bottom to top, the boxes correspond to the $2n$ bits of the balanced parentheses sequence of the 2d-Min-Heap [Fis10a], a \textit{bp_support} structure for navigation in the 2d-Min-Heap, and $\pm 1$RMQ-information on the excess sequence of the balanced parentheses sequence.

\textbf{sada_old}: The boxes, again from bottom to top, represent the space for: BPS\textit{dfs}, \textit{rank_support_v5} for the pattern ‘(’) and ‘)’, \textit{select_support_mcl} for the pattern ‘(’), and a $\pm 1$RMQ structure on the excess sequence.
Figure 6.2: Final memory consumption of succinct RMQ structures (left) and during the construction (right). Note that the non-stand-alone data structures succ and compr require access to array $A$ to answer queries. In our case $A$ is the the LCP array of the text `english.200MB` which has a size of 800 MB and is drawn in gray. Due to clarity the bars for the LCP arrays are cut off.

**sct**: Because we replace ±1RMQ by `rr_enclose`, it consists of only two parts (again from bottom to top): BPS$_{sct}$ and bp_support.

The final size of the stand-alone schemes highly depends on the size of the balanced parentheses sequence (depicted as the lowest box of the stacks in Fig. 6.2). However, the space for the $o(n)$-terms is non-negligible, and we can indeed observe a clear advantage of sct, which uses only one $o(n)$ structure for all operations. Note again that we can also reduce the space of 2dmin and sada_old by replacing the ±1RMQ data structure by a bp_support data structure. Compared with the two non-standalone schemes (succ and compr), sct is almost as small as compr$_1$ without adding the original array $A$ which is needed at query time! However for a fair comparison of both types of data structures we have to add $A$ to the non-stand-alone data structures and so sct needs only a tenth of memory of the best non-stand-alone solution compr.

The running times for $10^7$ random queries are also shown on the left of Figure 6.2. The solutions 2dmin and sada_old are the slowest, as they do two calls on different $o(n)$ data structures for each query. The non-stand-alone scheme succ is fastest, which is not surprising given that it does not use any compressed data structures (all arrays are byte-aligned). The other two schemes (sct, and compr) are about equally fast, showing that `rr_enclose` is also competitive in time.

The construction space (we measured the resident size) is depicted in the right plot of

---

1 The reason that the space for the compressed scheme compr is significantly lower than that of succ is that we now use much smaller data structures for rank and select than in the original implementation [FHS08], where both implementation use about the equal amount of space for this input.
6.3 Calculating Maximal Exact Matches

In this section we will consider a problem which we can solve faster and with less memory than previous solutions by using a CST of the sds1. This was achieved (1) by designing a new algorithm, which uses fast operations of the CST and (2) by choosing the right CST configuration. In the rest of this section, we will first give a motivation for the considered application, then we present the new algorithm, and finally show experimental results.

6.3.1 Motivation

A very active field of research in life sciences is comparative genomics, i.e. the study of the relationship of genome structure and function across different biological species. Since for most mammalian genomes the length of the DNA sequences is more than a billion base pair, the comparison of two such strings cannot be handled by classical approaches like the quadratic time edit distance calculation. Most of the comparative genomics software-tools overcome this obstacle by first computing exact matches between two DNA sequences which satisfy certain criteria. Large regions of similarity are then identified and aligned by extending or chaining these exact matches; see [TM06] for a classification of genome comparison tools. Exact matches between two strings \( T^1 \) and \( T^2 \) fall into three basic categories: (a) common \( k \)-mers (common substrings of a fixed length \( k \)), (b) maximal unique matches (these occur only once in \( T^1 \) and \( T^2 \)), (c) maximal exact matches (these cannot be extended in either direction towards the beginning or end of \( T^1 \) and \( T^2 \) without allowing for a mismatch). In our opinion, maximal exact matches are most suitable for genome comparisons; see e.g. [AKO08] for a discussion.

The bottleneck in large-scale applications like whole-genome comparisons is often the space requirement of the software-tools. If the index structure (e.g. a suffix array) does not fit into main memory, then it is worthwhile to use a compressed index structure instead; see [Lip05] for a discussion of the time-space trade-off. Lippert [Lip05] used compressed index structures of both strings \( T^1 \) and \( T^2 \) to compute common \( k \)-mers.
Hon and Sadakane [HS02] showed how maximal unique matches between $T^1$ and $T^2$ can be computed on a compressed version of the generalized suffix tree. However, their algorithms use relatively slow operations of a CST (suffix links, access to the inverse suffix array, etc.) and no publically available implementation exists. Recently, Khan et al. [KBKS09] showed how maximal exact matches between $T^1$ and $T^2$ can be computed by matching $T^2$ in forward direction against a sparse suffix array of $T^1$, extending the approach of Kurtz [KPD+04, AKO06] for uncompressed suffix arrays. The algorithm presented in this section uses $T^2$ and the CST of $T^1$ to compute maximal exact matches. It turned out, that we only need the backward search method of the underlying CSA (see Algorithm 6 on page 24 for details) and the (fast) parent, depth and node operation of the CST.

6.3.2 The New Algorithm

We will first define the term of a maximal exact match between two strings $T^1$ and $T^2$ of lengths $n_1$ and $n_2$ more formally: An exact match between two strings $T^1$ and $T^2$ is a triple $(k,p_1,p_2)$ such that $T^1[p_1..p_1+k−1] = T^2[p_2..p_2+k−1]$. An exact match is called right maximal if $p_1 + k = n_1$ or $p_2 + k = n_2$ or $T^1[p_1+k] \neq T^2[p_2+k]$. It is called left maximal if $p_1 = 0$ or $p_2 = 0$ or $T^1[p_1−1] \neq T^2[p_2−1]$. A left and right maximal exact match is called maximal exact match (MEM). In genome comparison, one is merely interested in MEMs that exceed a user-defined length threshold $\ell$.

To compute MEMs between $T^1$ and $T^2$, we construct a CST for the string $T^1$. Then we match the string $T^2$ backward against the CST and keep track of the longest matching path; see Algorithm 30. We only sketch here how the algorithm works. The full description and a proof of the correctness is given in [OGK10]. In the first iteration of the outer while-loop, the algorithm computes the longest suffix of $T^2[0..n_2−1]$ that matches a substring of $T^1$ by applying backward search character by character. In the inner while-loop, whenever a character is detected, the number $k$ (initially $k = 0$) of character matches is increased by one. Once the number $k$ of character matches is greater than or equal to the length threshold $\ell$, the algorithm adds the triple $(k,[lb..rb],[p_2])$ to the current list path (which is initially empty). At that moment, $[lb..rb]$ is the $\omega$-interval of the string $\omega = T^2[p_2..p_2+k−1]$. When the inner while-loop is left, we know that the string $T^2[p_2..n_2−1]$ does not match a substring of $S^1$ and that $[i..j]$ is the $\omega$-interval of the string $\omega = T[p_2+1..n_2−1]$. In the subsequent for-loop, each element of path is processed. Obviously, for each position $p_2'$ in $T^2$, at most one triple $(k',[lb'..rb'],[p_2'])$ appears in a matching path. It can be shown that each match $(k',SA[q],p_2')$ is a longest right maximal match at position $p_2'$ in $T^2$, where $lb \leq q \leq rb$. Now, Algorithm 30 tests left maximality by $T^1[SA[q]−1] = T^BWT[q] \neq T^2[p_2'−1]$. If $(k',SA[q],p_2')$ is left maximal, then it is a maximal exact match between $T^1$ and $T^2$ with $k \geq \ell$, and the algorithm outputs it. After that, it considers the parent LCP-interval of $[lb'..rb']$. Let us denote this parent interval by $k'' = [lb''..rb'']$. For each $q$ with $lb'' \leq q < lb'$ or $rb' < q \leq rb''$, the triple $(k'',SA[q],p_2')$ is a right maximal match because $T^1[SA[q]..SA[q]+k''−1] = T^2[p_2'..p_2'+k''−1]$ and $T^1[SA[q]+k''−1] = T^2[p_2'+k''−1]$ and $T^1[SA[q]+k''−1] = T^2[p_2'+k''−1]$ and $T^1[SA[q]+k''−1] = T^2[p_2'+k''−1]$. So if $k'' \geq \ell$ and $T^BWT[q] \neq T^2[p_2'−1]$, then the algorithm outputs $(k'',SA[q],p_2')$. Then it considers the parent LCP-interval of $[lb''..rb'']$ and so on.
6.3 Calculating Maximal Exact Matches

Algorithm 30 Computing maximal exact matches of length $\geq \ell$ by backward search.

```
01 $p_2 \leftarrow n_2 - 1$; $k \leftarrow 0$; $[i..j] \leftarrow [0..n_1 - 1]$ 
02 while $p_2 \geq 0$ do
03     $path \leftarrow [\ ]$
04     $[lb..rb] \leftarrow \text{backward_search}(T^2[p_2],[i..j])$
05     while $[lb..rb] \neq \bot$ or $p_2 \geq 0$ do
06         $k \leftarrow k + 1$
07         if $k \geq \ell$ then
08             add($path,(k,[lb..rb],p_2)$)
09         $[i..j] \leftarrow [lb..rb]$
10         $p_2 \leftarrow p_2 - 1$
11     $[lb..rb] \leftarrow \text{backward_search}(T^2[p_2],[i..j])$
12     for each $(k',[lb'..rb'],p'_2)$ in $path$ do
13         $[lb..rb] \leftarrow \bot$
14         while $k' \geq \ell$ do
15             for each $q \in [lb'..rb'] \setminus [lb..rb]$ do
16                 if $p'_2 = 0$ or $T^{BWT}[q] \neq T^2[p'_2 - 1]$ then
17                     output $(k',\text{SA}(q),p'_2)$
18             $[lb..rb] \leftarrow [lb'..rb']$
19             $k'',[lb'..rb'] \leftarrow \text{cst1.parent}(\text{cst1.node}(lb',rb'))$
20         if $[i..j] = [1..n_1 + 1]$ then
21             $k \leftarrow 0$
22         $p_2 \leftarrow p_2 - 1$
23     else
24     $k-[i..j] \leftarrow \text{cst1.parent}(\text{cst1.node}(i,j))$
```

If all elements in the current list $path$ have been processed, the algorithm computes the next longest matching path, this time with the current position $p_2$ instead of $n_2$ and the parent interval of the $T^2[p_2 + 1..n_2]$-interval $[i..j]$, yielding another list $path$. Then, each element in $path$ is processed, and so on. The process of computing longest matching paths continues until $p_2 = -1$. To sum up, Algorithm 30 checks every right maximal exact match exceeding the length threshold $\ell$ for left maximality. It follows as a consequence that it detects every maximal exact match of length $\geq \ell$. The time complexity of the Algorithm 30 is $O(n_2 \cdot t_{LF} + z + \text{occ} \cdot t_{SA})$, where $\text{occ} (z)$ is the number of (right) maximal exact matches of length $\geq \ell$ between the strings $T^1$ and $T^2$, since the $\text{parent}$ and $\text{node}$ operation can be performed in constant time.

To exemplify the algorithm, we match the string $T^2 = ulmul$ backward against the CST of our running example $T^1 = umulmundumulmum$; cf. Figure 2.9. Starting backward search with the last characters of $T^2$, we find that the strings $T^2[4..4] = 1$, $T^2[3..4] = ul$, $T^2[2..4] = mul$ match a substring of $T^1$ but $T^2[1..4] = 1mul$ does not. For the length
threshold \( \ell = 2 \), the matching path is
\[(2, [10..11], 3), (3, [5..6], 2)\]

The triple \((2, [10..11], 3)\) yields no output, since all matches are not left maximal. The parent interval of \([10..11]\) is not considered since its LCP value is \(< \ell \). The algorithm then outputs the MEMs \((3, SA[5] = 9, 2)\) and \((3, SA[6] = 1, 2)\) for the triple \((3, [5..6], 2)\). The parent interval of \([5..6]\) is \([5..8]\) and therefore we have to consider all suffixes in \([7..8]\).

We get no output, since the exact matches \((2, 12, 2)\) and \((2, 4, 2)\) are not left maximal. Now all triples in the matching path have been considered, and the algorithm computes the next longest matching path starting at position \(p_2 = 1\) and the parent interval of \(2 - [5..8]\) of \([5..6]\). The new matching path is \((3, [2..3], 1), (4, [10..11], 0)\). Triple \((3, [2..3], 1)\) produces no output, since all suffixes in it are not left maximal. The parent interval of \([2..3]\) is \(0 - [0..15]\) and not processed since \(0 < \ell\). Finally, triple \((4, [10..11], 0)\) yields the output of the MEMs \((4, 10, 0)\) and \((4, 2, 0)\). As the LCP value of the parent interval of \([10..11]\) is \(1 \leq \ell\) no other MEMs of length \(\geq \ell\) exists.

6.3.3 Implementation and Experiments

The computation of MEMs always consists of two phases: In the first phase, an index is constructed and in the second phase this index is used to actually calculate the MEMs. Since we want to show the performance of the new MEM algorithm and not of the construction process (which was done in Chapter 5), we solely focus on the second phase.

We use the sdlsl class cst_sct for the CST. This CST class was not yet mentioned since it does not fulfill all requirements of the CST concept (for instance an efficient implementation of the \(tlcp_idx(i)\) operation is not possible and therefore we can not use all LCP representations in combination with \(cst_sct\)) but its functionality suffices to be used in Algorithm 30. We configured the class as follows:

- \(cst_sct\) served as CSA, since it provides a fast access time (\(O(\log \sigma)\)) to LF and \(T_{BWT}\) and only takes \(n_1 \cdot \log \sigma + \frac{n_2}{T} \log n\) bits. From todays perspective the choice of \(cst_sct<wt_huff>\), \(k>\) would provide an even better time-space trade-off, cf. Figure 3.7 on page 53.

- We used \(lcp_kurtz\) as LCP representation, since most LCP values are small (< 255) and \(lcp_kurtz\) provides a fast constant access time in this case, cf. Figure 3.26 on page 97. So \(lcp_kurtz\) takes about 8 bits of memory.

- \(bp_support_gg<>\) served as \(bp_support\) data structure. Since we have parametrized it with \(rank_support_v\) and \(select_support_mcl\) it takes about 3\(n\) bits of memory. Note that we need the \(bp_support\) structure only for answering the \(parent\) operation in the algorithm to determine the parent interval of \([i..j]\). (Alternatively, one could decrement \(i\) until an index \(i'\) with \(LCP[i'] < x\) is found, and increment \(j\) until an index \(j'\) with \(LCP[j'] < x\) is found, where \(x = \max\{LCP[i], LCP[j]\}\). The interval \([i'..j']\) is the parent interval of \([i..j]\); see Section 2.8. This alternative
approach has a worse worst-case time complexity, but might work faster in practice. We did not implement it yet.)

In total our program occupies (for inputs \( \leq 4 \) GB) about \( n_1^2 + (1.375 + \frac{3}{4})n_1 \) bytes of memory. Note that we can further reduce the space by \( n_2 \) bytes, since the second string \( T^2 \) can be streamed from disk.

In order to compare our implementation, called \texttt{backwardMEM}, with other software-tools computing MEMs, we chose the one developed by Khan et al. [KBKS09], called \texttt{sparseMEM}.

Their method computes MEMs between \( T^1 \) and \( T^2 \) by matching \( T^2 \) in forward direction against a \textit{sparse} suffix array of \( T^1 \), which stores every \( K \)th suffix of \( T^1 \), where \( K \) is a user-defined parameter. (The reader should be aware of the difference between a \textit{sparse} and a \textit{compressed} suffix array: A \textit{sparse} suffix array stores each \( K \)th suffix of \( T^1 \), while a \textit{compressed} suffix array stores each \( k \)-th entry of the suffix array of \( T^1 \).) Our choice is justified by the fact that the sequential version of \texttt{sparseMEM} beats the open-source software-tool 	extit{MUMmer} [KPD+04] and is competitive with the closed-source software-tool \textit{vmatch}.

For a fair comparison, we used the sequential version of \texttt{sparseMEM} and the same input and output routines as \texttt{sparseMEM}. In the experiments, we used the program parameters \texttt{-maxmatch -n -l \ell}, which set the length threshold on the MEMs to \( \ell \). We ran both programs on a test set of DNA sequences of different species which was also

| \( S^1 \) | \( |S^1| \) | \( S^2 \) | \( |S^2| \) | \( \ell \) | \( K = 1 \) | \( K = 4 \) | \( K = 8 \) |
|---|---|---|---|---|---|---|---|
| \texttt{sparseMEM} | Mbp | Mbp | \( K = 1 \) | \( K = 4 \) | \( K = 8 \) |
| \texttt{A.fumigatus} | 29.8 | \texttt{A.nidulans} | 30.1 | 20 | 23s | 307 | 3m59s | 108 | 6m13s | 74 |
| \texttt{M.musculus16} | 35.9 | \texttt{H.sapiens21} | 96.6 | 50 | 1m15s | 430 | 10m52s | 169 | 19m56s | 163 |
| \texttt{H.sapiens21} | \texttt{M.musculus16} | 35.9 | 50 | 32s | 957 | 5m08s | 362 | 10m34s | 255 |
| \texttt{D.simulans} | 139.7 | \texttt{D.sechellia} | 168.9 | 50 | 2m17s | 1489 | 21m09s | 490 | 49m34s | 326 |
| \texttt{D.melanogaster} | 170.8 | \texttt{D.sechellia} | 168.9 | 50 | 2m37s | 1861 | 28m49s | 588 | 55m43s | 386 |
| \texttt{D.melanogaster} | \texttt{D.yakuba} | 167.8 | 50 | 2m49s | 1860 | 32m57s | 587 | 61m39s | 384 |
| \texttt{backwardMEM} | Mbp | Mbp | \( k = 1 \) | \( k = 8 \) | \( k = 16 \) |
| \texttt{A.fumigatus} | 29.8 | \texttt{A.nidulans} | 30.1 | 20 | 43s | 187 | 49s | 89 | 50s | 82 |
| \texttt{M.musculus16} | \texttt{H.sapiens21} | 35.9 | 50 | 2m09s | 261 | 2m09s | 142 | 2m16s | 134 |
| \texttt{H.sapiens21} | \texttt{M.musculus16} | 35.9 | 50 | 51s | 576 | 59s | 258 | 57s | 235 |
| \texttt{D.simulans} | \texttt{D.sechellia} | 139.7 | 50 | 5m42s | 859 | 17m35s | 399 | 32m39s | 366 |
| \texttt{D.melanogaster} | \texttt{D.sechellia} | 168.9 | 50 | 4m33s | 1074 | 11m19s | 504 | 20m38s | 464 |
| \texttt{D.melanogaster} | \texttt{D.yakuba} | 167.8 | 50 | 3m50s | 1068 | 5m18s | 502 | 7m35s | 463 |

**Table 6.1:** For each pair of DNA sequences, the time (in minutes and seconds) and space consumption in MB of the programs are shown (without the construction phase). We tested different values of \( K \) and \( k \) to demonstrate the time-space trade-off of the algorithms. The value \( \ell \) is the length threshold on the MEMs.

---

1 The program and the test cases are available under http://compbio.cs.princeton.edu/mems.

2 The software-tool \textit{vmatch} is available under http://www.vmatch.de/.
used in [KBKS09]. In the uncompressed case \((k = K = 1)\), the memory consumption of \(\text{backwardMEM}\) is smaller than that of \(\text{sparseMEM}\), but \(\text{sparseMEM}\) is faster. In the compressed cases, \(\text{backwardMEM}\) performs quite impressively, in most cases much better than \(\text{sparseMEM}\). For example, \(\text{backwardMEM}\) takes only 57s (using 235 MB for \(k = 16\)) to compare the human chromosome 21 with the mouse chromosome 16, whereas \(\text{sparseMEM}\) takes 10m34s (using 255 MB for \(K = 8\)); see Table 6.1.

The space consumption of \(\text{sparseMEM}\) decreases faster with \(K\) as that of \(\text{backwardMEM}\) with \(k\), but its running time also increases faster. While the experiments show a clear space-time trade-off for \(\text{sparseMEM}\), this is fortunately not the case for \(\text{backwardMEM}\). Sometimes its running time increases with increasing compression ratio, and sometimes it does not. This is because the algorithm is output-sensitive. More precisely, before a MEM \((q, \text{SA}[i], p_2)\) can be output, the algorithm first has to determine the value \(\text{SA}[i]\). While this takes only constant time in an uncompressed suffix array, it takes \(t_{\text{SA}}\) time in a compressed suffix array, and the value of \(t_{\text{SA}}\) crucially depends on the compression ratio; cf. Figure 3.6. It is future work to try if alternative approaches, which omit the access to the CSA, perform better in practice. For instance, we can store all suffix array indexes \(i\) to disk, sort them with an external memory algorithm, and finally stream the uncompressed \(\text{SA}\) array to get all \(\text{SA}[i]\) values.
7 Conclusion

The main goal of this thesis was to create a very space and time efficient compressed suffix tree implementation, which should — in the optimal case — replace uncompressed suffix trees in different real-world applications.

As already mentioned, a compressed suffix tree is a complex data structure, which can be logically divided into the three components CSA, LCP, and NAV. We have seen that there exists many theoretical proposals for each component. We have implemented the most promising ones and have evaluated their practical performance. Furthermore, we have proposed an additional data structure for the CSA component \((\text{csa}_\text{wt}<\text{wt}_\text{rlg}>\))\(^1\), two data structures for the LCP component \((\text{lcp}_\text{support}_\text{tree} \text{ and } \text{lcp}_\text{support}_\text{tree2})\)\(^1\), and two data structures for the NAV component \((\text{bp}_\text{support}_\text{g} \text{ and } \text{bp}_\text{support}_\text{gg})\)\(^1\). The experiments showed that all of them provide relevant time-space trade-offs.

The presented CST design made it possible to get a myriad of different CST configurations, which provide a wide range of time-space trade-offs. Our experimental results show that the first implementation of Sadakane’s CST by Välimäki at al. [VMGD09] was far from optimal and that our reimplementation uses only \(1n - 2.5n\) bytes, which is half of the space of the old implementation. In addition the operations are now performed faster and we have even presented applications where the new implementation outperforms the uncompressed suffix trees.

The implementation of our theoretical proposal [OFG10] takes less space and can compete with Sadakane’s proposal in applications which have to map between SA-intervals and nodes in the tree (for instance in the case of calculating matching statistics by backward search). The experiments also show that our implementations outperform other recent implementations [Cán10].

A second goal of our research was to accelerate the construction process of the CST. We have proposed a semi-external construction method which first constructs CSA, then LCP, and finally NAV. With this method it is possible to construct some CST variants faster than uncompressed suffix trees, even in the scenario when both data structures fit in main memory. One reason for this advance in the construction time is the use of a new lightweight LCP construction algorithm which is especially favorable for large texts.

Finally, we have presented applications which can be solved by exploiting the functionality of the new data structures. We have made all implementations available in a ready-to-use C++ library called sdsl to encourage people to solve more problems with this powerful set of space-efficient data structures.
Appendix
<table>
<thead>
<tr>
<th>Concept</th>
<th>sdsl class</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>int_vector</td>
<td>Mutable vector 3.4.1.</td>
</tr>
<tr>
<td></td>
<td>bit_vector</td>
<td>Specialization of int_vector.</td>
</tr>
<tr>
<td></td>
<td>enc_vector</td>
<td>Immutable vector which uses a self-delimiting code to compress integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequences; see Section 3.4.2.</td>
</tr>
<tr>
<td>rank_support</td>
<td>rank_support_v</td>
<td>Uses $0.25n$ bits to support a bit_vector of length $n$. Fast construction. See Section 3.5.</td>
</tr>
<tr>
<td></td>
<td>rank_support_v5</td>
<td>Uses $0.05n$ bits to support a bit_vector of length $n$. Fast construction. See Section 3.5.</td>
</tr>
<tr>
<td>select_support</td>
<td>select_support_mcl</td>
<td>Constant time select. Fast construction. See Section 3.5.</td>
</tr>
<tr>
<td></td>
<td>select_support_bs</td>
<td>Logarithmic time select. See Section 3.5.</td>
</tr>
<tr>
<td>wavelet_tree</td>
<td>wt</td>
<td>Balanced wavelet tree.</td>
</tr>
<tr>
<td></td>
<td>wt_int</td>
<td>Balanced and for integer alphabets.</td>
</tr>
<tr>
<td></td>
<td>wt_huff</td>
<td>Huffman shaped tree, $H_0$ compression.</td>
</tr>
<tr>
<td></td>
<td>wt_r1lm</td>
<td>Run-length wavelet tree, $H_k$ compression.</td>
</tr>
<tr>
<td></td>
<td>wt_r1lg</td>
<td>Hierarchical run-length wavelet tree, $H_k$ compression.</td>
</tr>
<tr>
<td>csa</td>
<td>csa_uncompressed</td>
<td>Uses $n \log n$ bits.</td>
</tr>
<tr>
<td></td>
<td>csa_sada</td>
<td>See Section 3.7.2.</td>
</tr>
<tr>
<td></td>
<td>csa_wt</td>
<td>See Section 3.7.3.</td>
</tr>
<tr>
<td>bp_support</td>
<td>bp_support_g</td>
<td>$2n + o(n)$ bits. See Section 3.8.1.</td>
</tr>
<tr>
<td></td>
<td>bp_support_gg</td>
<td>$2n + o(n)$ bits. See Section 3.8.1.</td>
</tr>
<tr>
<td></td>
<td>bp_support_sada</td>
<td>$2n + o(n)$ bits. Provides a very fast construction.</td>
</tr>
<tr>
<td>rmq</td>
<td>rmq_succinct_sada</td>
<td>Uses about $6n$ bits in practice. [Sad07b]</td>
</tr>
<tr>
<td></td>
<td>rmq_succinct_sct</td>
<td>Uses about $3n$ bits in practice. See Section 6.2.</td>
</tr>
<tr>
<td></td>
<td>rmq_support_sparse_table</td>
<td>Uses $n \log^2 n$ bits [BFC00].</td>
</tr>
<tr>
<td>lcp</td>
<td>lcp_uncompressed</td>
<td>Uses $n \log n$ bits.</td>
</tr>
<tr>
<td></td>
<td>lcp_dac</td>
<td>Uses direct accessible codes for compression [BLN09].</td>
</tr>
<tr>
<td></td>
<td>lcp_wt</td>
<td>Uses $H_0$ compression.</td>
</tr>
<tr>
<td></td>
<td>lcp_kurtz</td>
<td>Uses 1 byte for small entries, 2 words for large ones [Kur99].</td>
</tr>
<tr>
<td></td>
<td>lcp_support_sada</td>
<td>Uses $2n + o(n)$ bits [Sad02].</td>
</tr>
<tr>
<td></td>
<td>lcp_support_tree</td>
<td>See section 3.10.1.</td>
</tr>
<tr>
<td></td>
<td>lcp_support_tree2</td>
<td>See section 3.10.2.</td>
</tr>
<tr>
<td>cst</td>
<td>cst_sada</td>
<td>See Section 3.9.1.</td>
</tr>
<tr>
<td></td>
<td>cst_sct3</td>
<td>See Section 3.9.3.</td>
</tr>
</tbody>
</table>

Table 7.1: Description of concepts and classes in the sdsl.
Bibliography


Bibliography


[Sad00] Kunihiko Sadakane, Compressed text databases with efficient query algorithms based on the compressed suffix array, ISAAC, 2000, pp. 410–421.


## List of Figures

2.1 Computer memory hierarchy ........................................ 6  
2.2 The suffix tree of string $T=umulmundumulmum$. ................ 9  
2.3 The suffix array of the text $T=umulmundumulmum$. .......... 11  
2.4 The $\mathbf{T}^{\text{BWT}}$, $\Psi$, and $\mathbf{LF}$ of the text $T=umulmundumulmum$. ... 13  
2.5 Dependency of $\mathbf{LF}$, $\Psi$, and $\mathbf{T}^{\text{BWT}}$ ............... 15  
2.6 Balanced and Huffman shaped wavelet trees ................... 18  
2.7 The run-length encoded wavelet tree ............................. 20  
2.8 Backward search for $P = \text{umu}$ .............................. 22  
2.9 The longest common prefix array ................................ 25  

3.1 General design of a CST ............................................ 30  
3.2 Overview: Concepts in the $\text{sdsl}$ ............................. 33  
3.3 Runtime of different vector implementations ................... 38  
3.4 Runtime comparison of rank and select .......................... 40  
3.5 Hierarchical structure of $\text{wt_rlg}$ ............................ 42  
3.6 Runtime of the $[i]$-operator of different CSAs .................. 51  
3.7 Runtime of the other operations of the CSAs .................... 53  
3.8 An example of an ordered tree .................................... 54  
3.9 Jacobson’s balanced parentheses structure ...................... 57  
3.10 Example of Geary et al.’s balanced parentheses structure ..... 58  
3.11 Example for a $\text{find\_close}(i)$ call .......................... 60  
3.12 First example for an $\text{enclose}(i)$ call ....................... 62  
3.13 Second example for an $\text{enclose}(i)$ call ...................... 63  
3.14 Example for a call of $\text{rr\_enclose}$ ............................ 67  
3.15 Correctness of the $\text{min\_excess\_position}$ implementation ....... 68  
3.16 Example of the range min-max-tree ............................. 70  
3.17 Runtime of the $\text{find\_close}$ operation ....................... 73  
3.18 Runtime of the $\text{find\_open}$ operation ......................... 74  
3.19 Runtime of the $\text{enclose}$ operation (random) ................. 75  
3.20 Runtime of the $\text{enclose}$ operation ........................... 76  
3.21 Runtime of the $\text{double\_enclose}$ operation .................. 77  
3.22 Dependencies of $\text{cst\_sada}$ ................................. 79  
3.23 An example for $\text{BPS\_set}$ .................................... 81  
3.24 Two cases for determine the $\ell$-indices ........................ 85  
3.25 Post-order-index and depth value in a CST ..................... 92  
3.26 Random access times of different LCP structures .............. 97
3.27 Sequential random access times of different LCP structures . . . . . . 99
4.1 Runtime overview of cst_sada and cst_sct3 operations . . . . . . . 104
4.2 Time and space for the parent(v) operation . . . . . . . . . . . . 106
4.3 Time and space for the sibling(v) operation . . . . . . . . . . . 107
4.4 Time and space for the ith_child(v, 1) operation of different CSTs . . 108
4.5 Time for a depth-first-search traversal of different CSTs . . . . . . . 111
4.6 Time for a depth-first-search traversal of different CSTs . . . . . . . 112
4.7 Time for computing matching statistics with different CSTs . . . . . 113
4.8 Time and space for the child(v, c) operation . . . . . . . . . . . 115
4.9 Anatomy of one cst_sct3 configuration . . . . . . . . . . . . . . . 116
4.10 Anatomy of one cst_sada configuration . . . . . . . . . . . . . 118
5.1 CST construction profiles . . . . . . . . . . . . . . . . . . . . . . . . 122
5.2 Detailed CST construction profiles of the sdsl . . . . . . . . . . . . . 124
5.3 Suffix tree construction times . . . . . . . . . . . . . . . . . . . . . 125
5.4 Suffix tree performance comparison . . . . . . . . . . . . . . . . . . 125
5.5 Example of the first phase of the go-Φ algorithm . . . . . . . . . . . 129
5.6 Density of the LCP values for the Pizza&Chili test cases of size 200 MB. . 137
6.1 Calculation of H_k by using a CST . . . . . . . . . . . . . . . . . 140
6.2 Performance of succinct RMQ structures . . . . . . . . . . . . . . 144
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Pricing of Amazons Elastic Cloud Computing (EC2) service</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>The $k$-th order entropies of the <em>Pizza&amp;Chili</em> corpus</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Operation set of the <code>cst</code> concept</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Runtime of CST operations depending on CST components</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Operations set of the <code>wavelet_tree</code> concept</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Operation set of the <code>csa</code> concept</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>Worst case time complexities of CSAs</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Operation set of the <code>bp_support</code> concept</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>Detailed time complexities of CST operations</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Runtime of different LACA</td>
<td>136</td>
</tr>
<tr>
<td>6.1</td>
<td>Time-space trade-off for computing MEMs</td>
<td>149</td>
</tr>
<tr>
<td>7.1</td>
<td>Description of concepts and classes in the <code>sdsl</code></td>
<td>153</td>
</tr>
</tbody>
</table>