The compressed full-text index of the string S that is used throughout this book consists of the following four components:

1. the wavelet tree of the Burrows-Wheeler transformed string of S,
2. the sparse suffix array of S from Section 6.2.1,
3. the compressed LCP-array as explained in Section 6.2.2,
4. the balanced parentheses sequence BPS of the LCP-array that was introduced in Section 6.3.

We emphasize that each of the four components can be replaced with another component that has the same functionality. For example, the wavelet tree can be substituted by the compressed suffix array [135, 270] sketched in Section 6.2.1 because backward search can be done with the ψ-function; see Exercise 7.3.3. Further alternatives are described in [238]. However, the wavelet tree has many sophisticated properties that make it most suitable for many applications. Alternative compressed representations of the LCP-array are discussed in [126], among which is a representation that is based on the array LCP′ from Section 6.3.6. There are also alternatives to the BPS of the LCP-array, most notably the BPS_{pre} introduced in Section 6.1; cf. [231, 273]. We refer to [124] for an in-depth experimental study of the various incarnations of compressed full-text indexes.

7.2 The Burrows-Wheeler transform

The Burrows-Wheeler transform was introduced in a technical report written by David Wheeler and Michael Burrows [48]; see the historical notes in Adjeroh et al. [6]. In practice, the Burrows-Wheeler transformed string tends to be easier to compress than the original string; see e.g. [48, Section 3] and [215] for reasons why the transformed string compresses well.

Here we assume that the string S of length n is terminated by the sentinel character \$. Although this is not necessary for the Burrows-Wheeler transform to work correctly (cf. [48]), in virtually all practical cases the file to be compressed is terminated by a special symbol, the EOF (end of file) character. Moreover, it allows us to use a fast suffix sorting algorithm to compute the transformed string.

7.2.1 Encoding

The Burrows-Wheeler transform transforms a string S in three steps:

1. Form a conceptual matrix M' whose rows are the cyclic shifts of the string S.

$$ctatatat\$ \xrightarrow[shifts]{cyclic} \begin{array}{c} \\ \hline ctatatat\$ \\ tatatat\$c \\ atatat\$ct \\ tatat\$cta \\ atat\$ctat \\ tat\$ctata \\ at\$ctatat \\ t\$ctatata \\ \$ctatatat \\ \hline M' \end{array} \xrightarrow{sort} \begin{array}{ccc} F & & L \\ \hline \$ & ctatata & t \\ a & t\$ctata & t \\ a & tat\$cta & t \\ a & tatat\$c & t \\ c & tatatat & \$ \\ t & \$ctatat & a \\ t & at\$ctat & a \\ t & atat\$ct & a \\ t & atatat\$ & c \\ \hline & M & \end{array} \xrightarrow[column]{last} tttt\$aaac$$

Figure 7.1: The Burrows and Wheeler transform applied to the string $S = ctatatat\$$ yields the output $L = tttt\$aaac$.

2. Compute the matrix M by sorting the rows of M' lexicographically.

3. Output the last column L of M.

An example can be found in Figure 7.1. We next show that computing the Burrows-Wheeler transformed string of S boils down to sorting the suffixes of S, or more precisely, the output L of the Burrows-Wheeler transform can be derived in linear time from the suffix array SA. To this end, we define a string BWT and show that it coincides with L.

Definition 7.2.1 For a string S of length n having the sentinel character at the end (and nowhere else), the string $\mathsf{BWT}[1..n]$ is defined by $\mathsf{BWT}[i] = \$$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ if $\mathsf{SA}[i] \neq 1$.

Obviously, the string $\mathsf{BWT}[1..n]$ can be derived in linear time from the suffix array SA; see Algorithm 7.1.

Algorithm 7.1 Computing BWT from SA and the string S.

```
for $i \leftarrow 1$ to $n$ do
  if $\mathsf{SA}[i] = 1$ then $\mathsf{BWT}[i] \leftarrow \$$
  else $\mathsf{BWT}[i] \leftarrow S[\mathsf{SA}[i] - 1]$
```

If we truncate each string in the matrix M after the sentinel \$, then the truncated strings are still lexicographically ordered; see Figure 7.2. Since these truncated strings are exactly the suffixes of S, the string BWT coincides with the string L (this crucially relies on the fact that S is terminated by \$; see Exercise 7.2.2).

F		L
\$	ctatata	t
a	t\$ctata	t
a	tat\$cta	t
a	tatat\$c	t
c	tatatat	\$
t	\$ctatat	a
t	at\$ctat	a
t	atat\$ct	a
t	atatat\$	c

$\xrightarrow[\text{after } \$]{\text{truncate}}$

F		L
\$		t
a	t\$	t
a	tat\$	t
a	tatat\$	t
c	tatatat\$	\$
t	\$	a
t	at\$	a
t	atat\$	a
t	atatat\$	c

$\xrightarrow[L=\mathsf{BWT}]{\text{observe}}$

BWT	F	L
t	\$	t
t	at\$	t
t	atat\$	t
t	atatat\$	t
\$	ctatatat\$	\$
a	t\$	a
a	tat\$	a
a	tatat\$	a
c	tatatat\$	c

Figure 7.2: Truncate the strings (rows) after the sentinel character, and observe that $L = \mathsf{BWT}$.

i	1	2	3	4	5	6	7	8	9
$L[i]$	t	t	t	t	\$	a	a	a	c
$LF(i)$	6	7	8	9	1	2	3	4	5
$F[i]$	\$	a	a	a	c	t	t	t	t

Figure 7.3: LF maps the last column L to the first column F.

Exercise 7.2.2 For a string S of length n without the sentinel character at the end, define $\mathsf{BWT}[i] = S[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = S[\mathsf{SA}[i]-1]$ if $\mathsf{SA}[i] \neq 1$. Find a string S (without sentinel) for which $\mathsf{BWT} \neq L$.

7.2.2 Decoding

It is not obvious how the string BWT can be retransformed into the original string S. The key to this back-transformation is the so-called LF-mapping.

Definition 7.2.3 Let F and L be the first and last column in the matrix M; cf. Figure 7.1. The function $LF : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is defined as follows: If $L[i] = c$ is the k-th occurrence of character c in L, then $LF(i) = j$ is the index so that $F[j]$ is the k-th occurrence of c in F.

The function LF is called last-to-first mapping because it maps the last column L to the first column F; see Figure 7.3 for an example. In the following, when we regard the LF- mapping as an array, we will the use the notation $LF[i]$ instead of $LF(i)$.

Algorithm 7.2 Computing LF from BWT and the C-array.

```
for all $c \in \Sigma$ do
    $count[c] \leftarrow C[c]$
for $i \leftarrow 1$ to $n$ do
    $c \leftarrow \mathsf{BWT}[i]$
    $count[c] \leftarrow count[c] + 1$
    $LF[i] \leftarrow count[c]$
```

Next, we develop a linear-time algorithm that computes LF. To achieve this goal, we must be able to find the k-th occurrence of a character $c \in \Sigma$ in F. Employing the C-array (if we consider all characters in Σ that are smaller than c, then $C[c]$ is the overall number of their occurrences in S), the index of the first occurrence of character c in the array F is $C[c] + 1$. Therefore, the k-th occurrence of c in F can be found at index $C[c] + k$.

Algorithm 7.2 shows the pseudo-code for the computation of LF. It scans the BWT from left to right and counts how often each character appeared already. The algorithm uses an auxiliary array $count$ of size σ. Initially, $count[c] = C[c]$. Each time character c appears during the scan of BWT, $count[c]$ is incremented by one. As discussed above, if the algorithm finds the k-th occurrence of character c at index i in BWT, then the k-th occurrence of character c in F appears at index $count[c] = C[c] + k$. In other words, the index $LF[i]$ we are searching for is $count[c]$.

It remains to compute the original string S from BWT and LF. Lemma 7.2.4 states the crucial property of the LF-mapping that makes this possible.

Lemma 7.2.4 *The first row of the matrix M contains the suffix $S_n = \$$. If row i, $2 \leq i \leq n$, of the matrix M contains the suffix S_j, then row $LF(i)$ of M contains the suffix S_{j-1}.*

Proof Since $\$$ is the smallest character in Σ, the first row of M contains $\$$, which is the n-th suffix of S. Let $c \neq \$$ be a character in S, and let $i_1 < i_2 < \cdots < i_m$ be all the indices with $\mathsf{BWT}[i_k] = c$, $1 \leq k \leq m$. (So if we would number the m occurrences of c in $L = \mathsf{BWT}$ as $c_1, c_2, \ldots, c_m$, then $\mathsf{BWT}[i_k] = c_k$.) Because the suffixes in M are ordered lexicographically, we have $S_{\mathsf{SA}[i_1]} < S_{\mathsf{SA}[i_2]} < \cdots < S_{\mathsf{SA}[i_m]}$. Obviously, this implies $cS_{\mathsf{SA}[i_1]} < cS_{\mathsf{SA}[i_2]} < \cdots < cS_{\mathsf{SA}[i_m]}$. (With the occurrence numbers as subscripts, $c_1S_{\mathsf{SA}[i_1]} < c_2S_{\mathsf{SA}[i_2]} < \cdots < c_mS_{\mathsf{SA}[i_m]}$.) By definition, $LF(i_k)$ is the index so that $F[LF(i_k)]$ is the k-th occurrence of c in F. Since $cS_{\mathsf{SA}[i_k]} = S_{\mathsf{SA}[i_k]-1}$, it follows that row $LF(i_k)$ of M contains the suffix $S_{\mathsf{SA}[i_k]-1}$. □

Theorem 7.2.5 *If $L = \mathsf{BWT}$ is the output of the Burrows-Wheeler transform applied to the string S, and LF is the corresponding last-to-first mapping, then Algorithm 7.3 computes S.*

Algorithm 7.3 Computing the string S from BWT and LF.

```
$S[n] \leftarrow \$$
$j \leftarrow 1$
**for** $i \leftarrow n-1$ **downto** 1 **do**
  $S[i] \leftarrow \mathsf{BWT}[j]$
  $j \leftarrow LF(j)$
```

Proof Initially, the algorithm assigns \$ to $S[n]$. This is correct because \$ is the last character of S. Since \$ is the smallest character in Σ, row $j = 1$ of the matrix M contains the suffix $S_n = \$$. Now $L[1] = \mathsf{BWT}[1] = S[n-1]$ implies that the $(n-1)$-th character of S is correctly decoded in the first iteration of the for-loop. After the assignment $j \leftarrow LF(j)$, row j contains the suffix S_{n-1}. In the second iteration of the for-loop, the $(n-2)$-th character of S is correctly decoded because $L[j] = \mathsf{BWT}[j] = S[n-2]$, and so on. □

Exercise 7.2.6 Extend Algorithm 7.3 so that it also computes the suffix array of the string S. Is it possible to overwrite the LF-array with the suffix array? (This would save space if the LF-array is no longer needed.)

An alternative way to retransform the BWT into the original string S uses the ψ-function instead of the LF-mapping. We are already familiar with the ψ-function: For a string of length n (without the sentinel character \$ at the end), $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i]+1]$ for all i with $\mathsf{SA}[i] < n$; see Definition 5.5.4. Here, we assume that the string under consideration is terminated by \$. If S is a string of length n having the sentinel character at the end (and nowhere else), then $\mathsf{SA}[1] = n$ because \$ is the lexicographically smallest suffix of S. So with the previous definition of the ψ-function, the value $\psi(1)$ is undefined. Definition 7.2.7 provides a value for $\psi(1)$ so that ψ becomes a permutation.

Definition 7.2.7 The function $\psi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is defined by $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i]+1]$ for all i with $2 \le i \le n$ and $\psi(1) = \mathsf{ISA}[1]$.

The next two lemmata reveal the close relationship between the functions LF and ψ.

Lemma 7.2.8 *We have* $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$ *for all* i *with* $\mathsf{SA}[i] \neq 1$ *and* $LF(i) = 1$ *for the index* i *so that* $\mathsf{SA}[i] = 1$.

Proof If $\mathsf{SA}[i] = 1$, then $\mathsf{BWT}[i] = \$$. Since \$ occurs at index 1 in the array F, we have $LF(i) = 1$. Now suppose that $\mathsf{SA}[i] \neq 1$. According to Lemma 7.2.4, if $\mathsf{SA}[i] = j$, then $\mathsf{SA}[LF(i)] = j-1$. So the equation $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ holds true. Thus, $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$. □

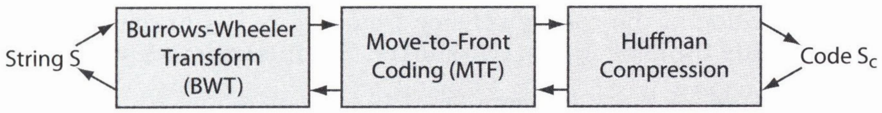

Figure 7.4: The main phases of the bzip2 compression program.

Lemma 7.2.9 *The functions LF and ψ are inverse of each other.*

Proof We will show $LF(\psi(i)) = i$ for all i with $1 \leq i \leq n$. (The equality $\psi(LF(i)) = i$ similarly follows.) If $i = 1$, then $\psi(1) = \mathsf{ISA}[1]$ is the index so that $\mathsf{SA}[\mathsf{ISA}[1]] = 1$. Hence $LF(\psi(1)) = 1$ by Lemma 7.2.8. For $i > 1$, it follows from Lemma 7.2.8 and Definition 7.2.7 that $LF(\psi(i)) = \mathsf{ISA}[\mathsf{SA}[\psi(i)] - 1] = \mathsf{ISA}[\underbrace{\mathsf{SA}[\mathsf{ISA}}_{cancel}[\mathsf{SA}[i] + 1]] - 1] = \mathsf{ISA}[\mathsf{SA}[i] + 1 - 1] = i$. □

Exercise 7.2.10 This exercise makes clear that LF can be replaced with ψ in BWT-decoding.

- Modify Algorithm 7.2 so that it computes the ψ-array from BWT. You may assume that the index index_of_\$, at which the character \$ occurs in the string BWT, is known (it can easily be computed during the Burrows-Wheeler transform).
- Modify Algorithm 7.3 so that it computes the string S from BWT, index_of_\$, and ψ.
- Show how to compute the suffix array SA from BWT, index_of_\$, and ψ. Is it possible to overwrite the ψ-array with the suffix array? (This would save space if the ψ-array is no longer needed.)

7.2.3 Data compression

The Burrows-Wheeler transform is used in many lossless data compression programs, of which the best known is Julian Seward's bzip2. Figure 7.4 shows bzip2's main phases. (Its ancestor bzip used arithmetic coding [267] instead of Huffman coding [158]. The change was made because of a software patent restriction.) It is possible to further use a run-length encoder (RLE) in between move-to-front (MTF) and Huffman coding, or to replace MTF with RLE. As a matter of fact, many more variations of the coding scheme are possible. The reader is referred to Adjeroh et al. [6] for a detailed introduction to the current state of knowledge about data compression with the Burrows-Wheeler transform.

An application of the coding scheme from Figure 7.4 to the string $S = ctatatat\$$ yields the code $S_c = 0111100010111$. The intermediate steps are

$$S = ctatatat\$ \overset{\text{BWT}}{\Longrightarrow} L = tttt\$aaac \overset{\text{MTF}}{\Longrightarrow} R = 300012003 \overset{\text{Huffman}}{\Longrightarrow} S_c = 011110000011101$$

We have already seen how the Burrows-Wheeler transform works, so we now turn to the other two steps: move-to-front and Huffman coding.

Move-to-front coding

Bentley et al. [36] introduced the *move-to-front* transform in 1986 but the method was already described in 1980 by Ryabko; see [269]. The MFT is an encoding of a string designed to improve the performance of entropy encoding techniques of compression like Huffman coding [158] and arithmetic coding [267]. The idea is that each character in the string is replaced by its rank in a list of recently used characters. After a replacement, the character is moved to the front of the list of characters. Algorithm 7.4 makes this precise.

Algorithm 7.4 Move-to-front coding of a string $L \in \Sigma^n$.

```
Initialize a $list$ containing the characters from $\Sigma$ in increasing order.
for $i \leftarrow 1$ to $n$ do
    $R[i] \leftarrow$ number of characters preceding character $L[i]$ in $list$
    move character $L[i]$ to the front of $list$
```

Figure 7.5 shows the application of Algorithm 7.4 to the string $L = tttt\$aaac$; note that '0' occurs more often in the resulting string R than t or a do in L. As you can see, every *run* (a run is a substring of identical characters) is replaced by a sequence of zeros (except for the first rank). Because a Burrows-Wheeler transformed string usually has many runs, the proportion of zero ranks after MTF has been applied is relatively high.

Pseudo-code for the decoding of the rank vector R is shown in Algorithm 7.5, and Figure 7.6 illustrates the behavior of this algorithm applied to the rank vector $R = 300012003$.

Algorithm 7.5 Move-to-front decoding of R.

```
Initialize a $list$ containing the characters from $\Sigma$ in increasing order.
for $i \leftarrow 1$ to $n$ do
    $L[i] \leftarrow$ character at position $R[i] + 1$ in $list$ (numbering elements from 1)
    move character $L[i]$ to the front of $list$
```

i	*list*	$L[i]$	$R[i]$
1	$\$act$	t	3
2	$t\$ac$	t	0
3	$t\$ac$	t	0
4	$t\$ac$	t	0
5	$t\$ac$	$\$$	1
6	$\$tac$	a	2
7	$a\$tc$	a	0
8	$a\$tc$	a	0
9	$a\$tc$	c	3

Figure 7.5: Move-to-front coding of a string $L = tttt\$aaac$.

i	*list*	$R[i]$	$L[i]$
1	$\$act$	3	t
2	$t\$ac$	0	t
3	$t\$ac$	0	t
4	$t\$ac$	0	t
5	$t\$ac$	1	$\$$
6	$\$tac$	2	a
7	$a\$tc$	0	a
8	$a\$tc$	0	a
9	$a\$tc$	3	c

Figure 7.6: Move-to-front decoding of $R = 300012003$ with $\Sigma = \{\$, a, c, t\}$.

character	'0'	'1'	'2'	'3'
frequency	5/9	1/9	1/9	2/9
Huffman code	1	000	001	01
fixed-length code	00	01	10	11

Figure 7.7: Encoding $R = 300012003$ with a Huffman code takes 15 bits. Encoding it with a fixed-length code would require 18 bits.

Huffman coding

Huffman coding is an encoding algorithm developed by David A. Huffman [158], which is used for lossless data compression. The algorithm works by creating the so-called Huffman tree and Huffman code in a bottom-up fashion as follows:

1. Initially, there are only leaf nodes, one for each character appearing in the string (file) to be encoded. Besides a character c, a leaf node contains a weight, which equals the frequency of c in the string.
2. The algorithm repeatedly creates a new node whose left child has the smallest weight, whose right child has the second smallest weight (from that point on, these two nodes are no longer considered), and whose weight is the sum of the weights of its children. This is done until only one node remains, the root of the Huffman tree.
3. The codeword of a character c can be read off the path from the root to the leaf that contains c: a 0 means "follow the left child" and a 1 means "follow the right child."

In this way, a variable-length code—a Huffman code—for encoding characters from the string is obtained. As an example, consider the string $R = 300012003$ on the alphabet $\{'0','1','2','3'\}$ and Figure 7.7. First, leaf $'1'$ (with weight $1/9$) becomes the left child and leaf $'2'$ (with weight $1/9$) becomes the right child of a new node v_1, whose weight is $2/9$. Second, the node, v_1 (with weight $2/9$) becomes the left child and leaf $'3'$ (with weight $2/9$) becomes the right child of another new node, v_2, whose weight is $4/9$. Third, the algorithm creates the root of the Huffman tree, whose left child is v_2 and whose right child is the leaf $'0'$. Figure 7.8 shows the resulting Huffman tree and Figure 7.7 shows the codewords.

Encoding a string is very simple: just replace each character in the string by its codeword. For instance, $R = 300012003$ is encoded by $S_c = 011110000011101$. A Huffman code is a prefix-free code, that is, the codeword representing some particular character is never a prefix of the codeword representing any other character. (Instead of the more accurate

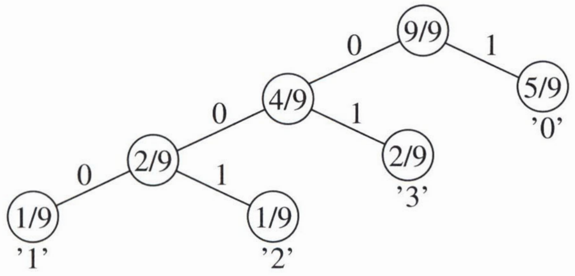

Figure 7.8: Huffman tree and code of $R = 300012003$.

term “prefix-free code,” the term “prefix code” is standard in the literature.) This property makes decoding also simple: the character that is represented by the initial codeword of the encoded string can be read off the path from the root to a leaf in the Huffman tree, where 0 means “go to the left child” and 1 means “go to the right child.” Then, the initial codeword is removed from the encoded string and the decoding process is repeated on the remainder of the encoded string.

For example, the decoding process for $S_c = 011110000011101$ starts at the root of the Huffman tree, goes to the left (since $S_c[1] = 0$) and then to the right (since $S_c[2] = 1$). Because the leaf '3' is encountered, '3' is the character that is represented by the codeword 01. Then, the decoding process continues with 1110000011101, the rest of S_c.

Huffman codes are so important because they are optimal in the sense of Definition 7.2.11; see e.g. [61] for a proof of this fact.

Definition 7.2.11 A prefix-free binary code pc for an alphabet Σ and a frequency function $f : \Sigma \to [0 \ldots 1]$ with $\sum_{c\in\Sigma} f(c) = 1$ is *optimal* if its expected codeword length

$$\sum_{c\in\Sigma} f(c)\ |pc(c)|$$

is minimum among all prefix-free binary codes for Σ and f.

7.2.4 Direct construction of the BWT

In the last few years, several algorithms have been proposed that construct the BWT either directly or by first constructing the suffix array and then deriving the BWT in linear time from it; see e.g. [173, 206, 254, 293]. The latter approach has a major drawback: all known SACAs require at least $n \log n + n \log \sigma$ bits of main memory ($n \log n$ bits for the suffix array and $n \log \sigma$ bits for the string S). If one has to deal with large datasets, it is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	i	m	i	m	m	m	i	s	i	s	m	i	s	i	s	s	i	i	p	i	$
type	S	L	S	L	L	L	S	L	S	L	L	S	L	S	L	L	S	S	L	L	S
			*				*		*			*		*			*				*

	$	i									m					p	s				
LMS	21				17	3	7	12	9	14											
L-type suffixes	21	20			17	3	7	12	9	14	2	6	11	5	4	19	16	8	13	10	15
S-type suffixes	21	20	17	1	17	3	7	12	9	14	2	6	11	5	4	19	16	8	13	10	15
					3	18	7	12	9	14											

Figure 7.9: Phase II of the induced sorting algorithm.

therefore advantageous to construct the BWT more space efficiently.[1] For example, Okanohara and Sadakane [254] have shown that the induced sorting algorithm devised by Nong et al. [244] (the SACA from Section 4.1.2) can be modified so that it directly constructs the BWT in linear time.

In this section, we discuss (a variant of) the algorithm presented in [254], which we call algorithm BWTbyIS (direct computation of the BWT by induced sorting). Algorithm BWTbyIS shares the same structure with the induced sorting algorithm: it is also divided in two phases. First, we explain how phase II of algorithm BWTbyIS works. We briefly recall phase II of the induced sorting algorithm because this will be the basis of our explanation. The example of Figure 7.9 will serve as an illustration (this is the same example as in Section 4.1.2).

Phase II of the induced sorting algorithm starts with the sequence of sorted LMS-positions (i.e., the order corresponds to the increasing lexicographic order of the suffixes starting at these LMS-positions). In the example of Figure 7.9, this is the sequence $21, 17, 3, 7, 12, 9, 14$.

In step 1 of phase II, the sequence of sorted LMS-positions is scanned from right to left (hence in decreasing order) and the positions are moved to their buckets in such a way that they appear in increasing order in the S-type regions of the buckets; see Figure 7.9.

In step 2 of phase II, the array A is scanned from left to right. If, for an element $A[i]$, the position $A[i]-1$ is of type L (i.e., $T[A[i]-1] = \text{L}$), then $A[i]-1$ is moved to the current front of its bucket. In the example of Figure 7.9, what happens when the LMS-position 21 is encountered? The position 20 is moved to the current front of the i-bucket because $T[20] = \text{L}$ and $S[20] =$ i. When position 20 is reached during the scan, position 19 is moved to the current front of the p-bucket because $T[19] = \text{L}$ and $S[19] =$ p. So step

[1]To deal with massive data, one has to resort to external-memory algorithms; see e.g. [30, 99].

2 handles the transitions from LMS-positions to L-positions (LMS→L) as well as the transitions from L-positions to L-positions (L→L). Within a bucket, L→L transitions are dealt with before LMS→L transitions because in a left-to-right scan the L-type region appears before the S-type region.

In step 3 of phase II, the array A is scanned from right to left. If, for an element $A[i]$, the position $A[i]-1$ is of type S (i.e., $T[A[i]-1] =$ S), then $A[i]-1$ is moved to the current end of its bucket. In the example of Figure 7.9, what happens when position 19 is encountered? Position 18 is moved to the current end of the i-bucket because $T[18] =$ S and $S[18] =$ i. When position 18 is reached during the scan, position 17 is moved to the current end of the i-bucket because $T[17] =$ S and $S[17] =$ i. So step 3 handles the transitions from L-positions to S-positions (L→S) as well as the transitions from S-positions to S-positions (S→S). Within a bucket, S→S transitions are dealt with before L→S transitions because in a right-to-left scan the S-type region appears before the L-type region.

Algorithm BWTbyIS does not work with LMS-positions but with LMS-substrings. It starts with the sequence of LMS-substrings *ending* at the sorted LMS-positions. In our example, this is the sequence

iipi\$, issi, \$imi, immmi, ismi, isi, isi

The reader may wonder why the string \$imi appears in this sequence, and not the string imi. Strictly speaking, the string $S[1..3]$ ending at the first LMS-position 3 is not an LMS-string. For reasons that will become clear below, we prepend \$ to this string and say that the resulting string is the LMS-string ending at the first LMS-position of S (i.e., we interpret S as a cyclic string).

In step 1 of phase II, the sequence of sorted LMS-substrings is scanned from right to left. In contrast to the induced sorting algorithm, algorithm BWTbyIS uses queues instead of the buckets. To be precise, for each character $c \in \Sigma$, there is one queue LMS-$queue[c]$ (which is initially empty). When an LMS-substring uc is encountered during the right-to-left scan, the string u is added to the queue LMS-$queue[c]$. In our example, we have LMS-$queue[\$] =$ [iipi] and LMS-$queue[i] =$ [iss, \$im, immm, ism, is, is] (the remaining three queues are empty).

Since positions are not available any more, transitions cannot be detected by looking up the types of positions in the type array T, but they can be inferred from the LMS-substrings. To exemplify the idea, let us have a look on the string iipi. Because it is in LMS-$queue[\$]$, we know that iipi\$ is an LMS-string and that the position at which the character \$ appears is an LMS-position. In what follows, we say that a character is of type L (S, respectively) if the position at which it occurs is of type L (S, respectively). So the character \$ is of type S and the character preceding it is of type L. Given a character c of type L, it is possible to infer the type

transition	implementation
LMS→L	remove from an LMS-*queue*, add to an L-*queue*
L→L	remove from an L-*queue*, add to an L-*queue*
L→S	remove from an L-*stack*, add to an S-*queue*
S→S	remove from an S-*queue*, add to an S-*queue*

Figure 7.10: Implementation of the four types of transitions.

of the preceding character b: if $b < c$, then b is of type S; otherwise it is of type L. In our example, this yields

i	i	p	i	$
	S	L	L	LMS

Once an L→S transition is observed, it is clear from the type structure of an LMS-substring that the types of the remaining characters must be S.

To simulate the transitions in steps 2 and 3 of the induced sorting algorithm, algorithm BWTbyIS employs the following data structures. For for each character $c \in \Sigma$, there are two queues L-$queue[c]$ and S-$queue[c]$ as well as a stack L-$stack[c]$ (all of which are initially empty). An LMS→L (L→L, respectively) transition in step 2 is implemented by dequeuing an element from an LMS-*queue* (L-*queue*, respectively) and enqueuing an element to an L-*queue*. When an L→S transition is detected in step 2, it must be postponed to step 3. Furthermore, since step 2 scans from left to right, but step 3 scans from right to left, the order in which L→S transitions are processed must be reversed. That is why L→S transitions are stored in a stack, and not in a queue. In step 3, an L→S (S→S, respectively) transition is implemented by popping an element from an L-*stack* (dequeuing an element from an S-*queue*, respectively) and enqueuing an element to an S-*queue*. Figure 7.10 summarizes the implementation of the transitions.

Now we have all the ingredients to achieve the main goal, namely to compute the BWT. In the algorithm BWTbyIS, the BWT is implemented as an array with the bucket structure known from the induced sorting algorithm. When a character c of type L is processed in step 2, its preceding character b is moved to the current front of the c-bucket of the array BWT (initially, the front of the c-bucket is the index $C[c] + 1$) and the current front of the c-bucket is shifted by one position to the right. Analogously, when a character c of type S is processed in step 3, its preceding character b is moved to the current end of the c-bucket of the array BWT (initially, the end of the c-bucket is the index $C[c + 1]$) and the current end of the c-bucket is shifted by one position to the left. There is one caveat though: when a last character of type S is reached (this is the leftmost character of an initial LMS-substring), the preceding character is not available. The solution to this problem relies on the fact that the position of such a character is an LMS-position. Recall that phase II of the induced sorting algorithm starts with sorted LMS-positions, and that the relative order of these positions remains unchanged in step 3. In the example of Figure 7.9, the sequence of sorted LMS-positions is $21, 17, 3, 7, 12, 9, 14$, and the sequence of characters at the preceding positions $20, 16, 2, 6, 11, 8, 13$ is i,s,m,m,m,s,s. In the right-to-left scan of step 3, these characters must be accessed in the reverse order, that is why they are stored on the stack *char-stack*.

Algorithm 7.6 shows pseudo-code of phase II of algorithm BWTbyIS, which runs in linear time. It needs $n \log \sigma$ bits for all LMS-substrings of S, $n \log \sigma$ bits for the BWT, and some auxiliary data structures. The stack *char-stack* can be emulated by using the S-type regions of the array BWT; see Exercise 7.2.13. In step 2, the algorithm merely needs the *front* pointers, the LMS-*queues*, the L-*queues*, and the L-*stacks*. This is the amount of extra memory (besides the $2n \log \sigma$ bits) needed by the algorithm because step 3 requires less memory.

Exercise 7.2.12 Apply Algorithm 7.6 to the example of Figure 7.9.

Exercise 7.2.13 Show how to emulate the stack *char-stack* in Algorithm 7.6 by using the S-type regions of the array BWT. This saves memory. (Note that only the L-type regions of BWT are filled in step 2, and that the S-type regions of BWT are filled from right to left in step 3.)

We now discuss phase I of algorithm BWTbyIS. As in the induced sorting algorithm, the lexicographic names of LMS-substrings can be computed by an application of (a modified version of) Algorithm 7.6 to unsorted LMS-substrings; see Exercise 7.2.17. In contrast to the induced sorting algorithm, however, it is not straightforward to obtain the string $\overline{S}$ (the string that is obtained from S by replacing each LMS-substring with its lexicographic name; recall that the induced sorting algorithm proceeds recursively with $\overline{S}$ unless the LMS-substrings are pairwise distinct). This is because the positions at which the LMS-substrings start are not available. A possible solution would be to store the LMS-substrings in an array ILN so that ILN$[k]$ is the k-th lexicographically smallest LMS-substring. Then, one determines the LMS-substrings of S from left to right. For each LMS-substring ω encountered, a binary search (see Section 5.1.3) yields the index k with ILN$[k] = \omega$. This index is the lexicographic name of ω, and it is appended to the growing string $\overline{S}$. However, this possible solution has a non-linear runtime. Using algorithm engineering techniques, it is possible to implement a space-efficient linear-time algorithm, but we will present a conceptually simpler approach.

Algorithm 7.6 Phase II of the induced sorting algorithm.

```
initialize an empty stack $char\text{-}stack$
for each $c$ in $\Sigma$ do
  initialize empty queues LMS-$queue[c]$, L-$queue[c]$, and S-$queue[c]$
  initialize an empty stack L-$stack[c]$
  $front[c] \leftarrow C[c] + 1$          /* front of the $c$-bucket */
  $end[c] \leftarrow C[c+1]$          /* end of the $c$-bucket */

/* step 1: right-to-left scan */
Scan the sorted sequence of LMS-substrings ending at LMS-positions
from right to left. For each LMS-substring $uc$ encountered in the scan,
enqueue the string $u$ to the queue LMS-$queue[c]$.
/* If $S[i..j] = uc$, then $T[i-1] = $ L, $T[i..j-1] = [$S, ..., S, L, ..., L$]$, $T[j] = $ S */

/* step 2: left-to-right scan, L-type characters before S-type characters */
for each $c$ in $\Sigma$ do          /* in increasing order */
  while L-$queue[c]$ is not empty do          /* $c$ is L-type */
    $\omega b \leftarrow dequeue($L-$queue[c])$ /* $b$ is the last character of the string */
    BWT$[front[c]] \leftarrow b$
    $front[c] \leftarrow front[c] + 1$
    if $b < c$ then          /* $b$ is S-type */
      $push($L-$stack[c], \omega b)$          /* store L→S transition */
    else          /* $b$ is L-type */
      $enqueue($L-$queue[b], \omega)$          /* L→L transition */
  while LMS-$queue[c]$ is not empty do          /* $c$ is S-type */
    $\omega b \leftarrow dequeue($LMS-$queue[c])$  /* $b$ is L-type */
    $push(char\text{-}stack, b)$          /* store character left of LMS-position */
    $enqueue($L-$queue[b], \omega)$          /* LMS→L transition */

/* step 3: right-to-left scan, S-type characters before L-type characters */
for each $c$ in $\Sigma$ do          /* in decreasing order */
  while S-$queue[c]$ is not empty do          /* $c$ is S-type */
    $v \leftarrow dequeue($S-$queue[c])$
    if $v = \varepsilon$ then          /* $v$ is the empty string */
      $b \leftarrow pop(char\text{-}stack)$          /* $b$ precedes $c$ in $S$, $b$ is L-type */
    else
      $\omega b \leftarrow v$          /* decompose $v$, its last character $b$ is S-type */
      $enqueue($S-$queue[b], \omega)$          /* S→S transition */
    BWT$[end[c]] \leftarrow b$
    $end[c] \leftarrow end[c] - 1$
  while L-$stack[c]$ is not empty do          /* $c$ is L-type */
    $\omega b \leftarrow pop($L-$stack[c])$          /* $b$ is S-type */
    $enqueue($S-$queue[b], \omega)$          /* L→S transition */
```

Phase I:

1. In a left-to-right scan of the string S and the type array T, successively determine the LMS-substrings of S and incrementally build the trie of all LMS-substrings (in which the outgoing edges of a node are ordered alphabetically); see Figure 7.11. Mark nodes at which an LMS-substring ends.[2] As shown in Section 2.5, this takes $O(n)$ time. Furthermore, store the first LMS-position j_1 and count how many different LMS-substrings appear in S; let m denote this number. Initialize an array LMS-$array[1..m]$ (in the end, this array will contain the sequence of LMS-substrings with which phase II starts).

2a. If all the LMS-substrings are pairwise distinct (in this case m equals the number of LMS-substrings), then proceed as follows: In a postorder traversal of the trie, number the marked nodes from 1 to m in the order of their appearance. In a left-to-right scan of the string S, walk through the trie and compute LMS-substrings as follows: Start with the root of the trie and follow the edge whose label is the first character $S[j_1]$ of the first LMS-substring, then the edge whose label is the second character $S[j_1+1]$ and so on, until a marked node in the trie is reached. Let k_1 be its number. Note that the concatenation of the edge labels on the path from the root to node k_1 spells out the first LMS-substring $S[j_1 \dots j_2]$ of S. Set LMS-$array[k_1] = \$S[1 \dots j_1]$.[3] Then, start again with the root of the trie and follow the path corresponding to a prefix of $S[j_2 \dots n]$ until a marked node k_2 in the trie is reached. Clearly, the concatenation of the edge labels on the path from the root to node k_2 spells out the second LMS-substring $S[j_2 \dots j_3]$ of S. Set LMS-$array[k_2] = S[j_1 \dots j_2]$. Again, start at the root of the trie and follow the path corresponding to a prefix of $S[j_3 \dots n]$ until a marked node k_3 in the trie is reached, set LMS-$array[k_3] = S[j_2 \dots j_3]$, etc. Upon termination of this process, the LMS-$array$ contains the sequence of LMS-substrings with which phase II starts.

2b. If not all LMS-substrings are pairwise distinct, then—as in phase I of the induced sorting algorithm—the algorithm must be applied recursively to the string $\overline{S}$, which is obtained from S by replacing each LMS-substring with its lexicographic name. This string $\overline{S}$ can be computed as follows: Initialize an array ILN$[1..m]$ (in the end, this will be the inverse of the lexicographic naming, i.e., of the array LN). In a postorder traversal of the trie, number the marked nodes from 1 to m in the order of their appearance (see Figure 7.11) and simultaneously fill the array ILN: if the concatenation of the edge labels on the

[2]In fact, it suffices to mark the internal nodes at which an LMS-substring ends. For ease of presentation, however, we also mark leaves.

[3]The first LMS-substring is a special case.

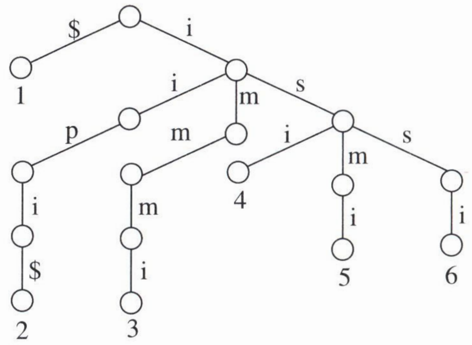

Figure 7.11: The trie of the LMS-substrings \$, iipi\$, immmi, isi, ismi, and issi with the lexicographic names.

current path from the root to node k spells out the LMS-substring ω, then set $\mathsf{ILN}[k] = \omega$ because the lexicographic name of ω is k. In a left-to-right scan of the string S (starting at the first LMS-position j_1), walk through the trie and compute LMS-substrings as in step (2a). Whenever a marked node is encountered during this process, append its number to the string $\overline{S}$ (initially, $\overline{S} = \varepsilon$). Then, recursively compute the Burrows-Wheeler transform $\overline{\mathsf{BWT}}$ of the string $\overline{S}$. Using $\overline{\mathsf{BWT}}$, fill the LMS-*array*: for i from 1 to n do

$$\text{LMS-}array[i] = \begin{cases} \$S[1\ldots j_1] & \text{if } \mathsf{ILN}[\overline{\mathsf{BWT}}[i]] = \$ \\ \mathsf{ILN}[\overline{\mathsf{BWT}}[i]] & \text{otherwise} \end{cases}$$

Exercise 7.2.14 Show that algorithm BWTbyIS runs in linear time.

Exercise 7.2.15 Apply phase I of algorithm BWTbyIS to the example from Figure 7.9 (page 292).

Exercise 7.2.16 Explain why the marked nodes of the trie are numbered in a postorder traversal (and not in a preorder traversal).
Hint: Use an example in which an LMS-substring is a proper prefix of another LMS-substring.

Exercise 7.2.17 Modify Algorithm 7.6 in such a way that it computes the lexicographic names of (initially unsorted) LMS-substrings.