Lempel Ziv Computation In Compressed Space (LZ-CICS)

Dominik Köppl[1] and Kunihiko Sadakane[2]

[1]Department of Computer Science, TU Dortmund, Germany
[2]Graduate School of Information Science and Technology, University of Tokyo, Japan

Abstract

We show that both the Lempel Ziv 77- and the 78-factorization of a text of length n on an integer alphabet of size σ can be computed in $\mathcal{O}(n \lg\lg\sigma)$ time (linear time if we allow randomization) using $\mathcal{O}(n \lg\sigma)$ bits of working space. Given that a compressed representation of the suffix tree is loaded into RAM, we can compute both factorizations in linear time using $\mathcal{O}(n)$ space. We achieve the same time bounds when considering the rightmost variant of the LZ77 problem, where the goal is to determine the largest possible referred position of each referencing factor.

1 Introduction

Centerpiece of most compression algorithms is the Lempel Ziv 77 (LZ77) [1] or the Lempel Ziv 78 (LZ78) [2] factorization. Although both factorizations are quite easy to understand, computing them in optimal time and space is still an unsolved problem. A variant of the LZ77 computation problem emerges from the encoding of the resulting factorization: Encoders commonly translate an LZ77 factor to its length and the offset to its referred position. The latter integer can be minimized when multiple referred positions (resulting in the same factor length) are available. This problem was recently addressed as the ***rightmost parsing*** problem in [3].

We tackle both problems in this paper: We show that the LZ77 and the LZ78 factorization of a text of length n on an integer alphabet of size σ can be done

- with $\mathcal{O}(n \lg\sigma)$ working space in either $\mathcal{O}(n)$ randomized or $\mathcal{O}(n \lg\lg\sigma)$ deterministic time, and
- with $\mathcal{O}(n)$ additional working space in linear time, given that we have access to the compressed suffix tree of the text.

We further achieve the rightmost parsing (for LZ77) in the same time.

2 Related Work

Due to the high abundance of publications, we sketch only recent achievements related to our approach.

We are aware of the following results for LZ77: The currently most space efficient algorithm is due to Kosolobov [4] whose algorithm runs in $\mathcal{O}(n(\lg\sigma + \lg\lg n))$ time and uses only εn bits of working space, provided that we have read-access to the text. A trade-off algorithm is given by Kärkkäinen et al. [5], using $\mathcal{O}(n/d)$ words of working space and $\mathcal{O}(dn)$ bits. By setting $d \leftarrow \log_\sigma n$ we get $\mathcal{O}(n \lg \sigma)$ bits and $\mathcal{O}(n \log_\sigma n)$ time. The algorithm of Belazzougui and Puglisi [6] derives its dominant terms in space and time from the same data structure [7] as we do; the LZ77 factorization algorithms of both papers work with the same space in the same time bounds. Their algorithm uses only the Burrows-Wheeler transform (BWT) [8] construction algorithm from [7]. By exchanging it with an improved version [9], we expect that their algorithm will run in deterministic linear time.

Despite the fact that the rightmost parsing achieves practically better results than other LZ77-variants [10], there exists much fewer theoretical analysis on this compression coding. For instance, Ferragina et al. [10] proposed an algorithm running in $\mathcal{O}(n + n \lg \sigma / \lg \lg n)$ time with $\mathcal{O}(n \lg n)$ bits of working space. Subsequently, Belazzougui and Puglisi [6] achieve the rightmost parsing in $\mathcal{O}\big(n(1 + \lg \sigma / \sqrt{\lg n})\big)$ time.

Since LZ78 factors are naturally represented in a trie, the so-called ***LZ trie***, improving LZ78 computation can be done, among others, by using sophisticated trie implementations [11, 12], or by superimposing the suffix tree with the suffix trie [13, 14]. We follow the latter approach. There, both Nakashima et al. [13] and Fischer et al. [14] presented a linear time algorithm, using $\mathcal{O}(n \lg n)$ and $(1+\varepsilon)n \lg n + \mathcal{O}(n)$ bits of space, respectively.

Based on the data structures of the compressed suffix tree, we derive our techniques from an approach [14] using the suffix tree topology with succinct representations of the suffix array, its inverse and the longest common prefix array. For both factorization variants, Fischer et al. [14] store the inverse suffix array and parts of the enhanced suffix array in $(1 + \varepsilon)n \lg n + \mathcal{O}(n)$ bits of space such that they can access leaves of the suffix tree in text order, and can compute the string depth of internal nodes, both in constant time. By overwriting the working space multiple times, and using a complicated counting for the LZ78 trie nodes, their algorithm looks more intricately than our approach.

3 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$ for some natural number n. Accessing a word costs $\mathcal{O}(1)$ time. We assume that the function *popcount*(w), counting the set bits in a word w, can be computed in constant time. [1]

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| \le n$. We call an element $T \in \Sigma^*$ a ***string*** or ***text***. Its length is denoted by $|T|$. The empty string is ϵ with $|\epsilon| = 0$. We access the j-th character of T with $T[j]$. Given $x, y, z \in \Sigma^*$ with $T = xyz$, then x, y and z are called a ***prefix***, a ***substring***, and a ***suffix*** of T, respectively. In particular, the suffix starting at position j of T is called the j***-th suffix*** of T.

Besides, for a string T and a character c, we are interested in answering the following queries:

- $T.\mathrm{rank}_c(j)$ counts the number of 'c's in $T[1, j]$, and

[1] Otherwise, we build a lookup-table like in [15] supporting *popcount* with two rank queries in constant time while fitting in our working space.

- $T.\mathrm{select}_c(j)$ gives the position of the j-th 'c' in T.

Regarding bit vectors, i.e., strings on a binary alphabet, we use a result due to Raman et al. [16]: There is a data structure using $o(|T|)$ space that can answer rank and select queries in constant time. We say that a bit vector has a ***rank-support*** and a ***select-support*** if it provides constant time access to rank and select, respectively. It can be constructed in time linear to $|T|$.

The zero-order entropy $H_0(n, z)$ of a bit vector of length n storing z ones is defined by $H_0(n, z)\, n = z \lg(n/z) + (n - z) \lg(n/(n - z))$. Such a bit vector can be compressed such that it consumes $H_0(n, z)\, n + o(n)$ bits, supporting access and rank in constant time (e.g., [16]).

In the rest of this paper, we take a read-only string T of length n, which is subject to LZ77 or LZ78 factorization. Let $T[n]$ be a special character appearing nowhere else in T, so that no suffix of T is a prefix of another suffix of T. We assume that Σ is the *effective* alphabet of T, i.e., each character of Σ appears in T at least once.

3.1 Lempel Ziv Factorization

A ***factorization*** of T with size z partitions T into z substrings $T = f_1 \cdots f_z$. These substrings are called ***factors***. In particular, we have:

Definition 3.1. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ77 factorization** *of T iff $f_x = \mathrm{argmax}_{S \in S_j(T) \cup \Sigma} |S|$ for all $1 \leq x \leq z$ with $j = |f_1 \cdots f_{x-1}| + 1$, where $S_j(T)$ denotes the set of substrings of T that start strictly before j (for $1 \leq j \leq |T|$).*

Definition 3.2. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ78 factorization** *of T iff $f_x = f'_x \cdot c$ with $f'_x = \mathrm{argmax}_{S \in \{f_y : y < x\} \cup \{\epsilon\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$.*

3.2 Suffix Tree

The ***suffix trie*** of T is the trie of all suffixes of T. The ***suffix tree (ST)*** of T, denoted by ST, is the tree obtained by compacting the suffix trie of T. We denote the root node of ST by root. In this paper we use a compressed representation of ST, consisting of

- the ψ-array [17], and
- a $4n + o(n)$-bit balanced parenthesis representation (BP) [18] of the tree topology [19], equipped with the minmax tree [20] for navigation.

Using the algorithm of Belazzougui [7], we can build the compressed suffix tree using $\mathcal{O}(n \lg \sigma)$ bits of space in either $\mathcal{O}(n)$ randomized time or $\mathcal{O}(n \lg \lg \sigma)$ deterministic time.

By the BP representation, each node of the suffix tree is uniquely identified by its pre-order number. Providing a rank and select data structure on the BP representation, a node can be addressed by its pre-order number in constant time. If the context is clear, we implicitly convert nodes of ST to their pre-order number, and

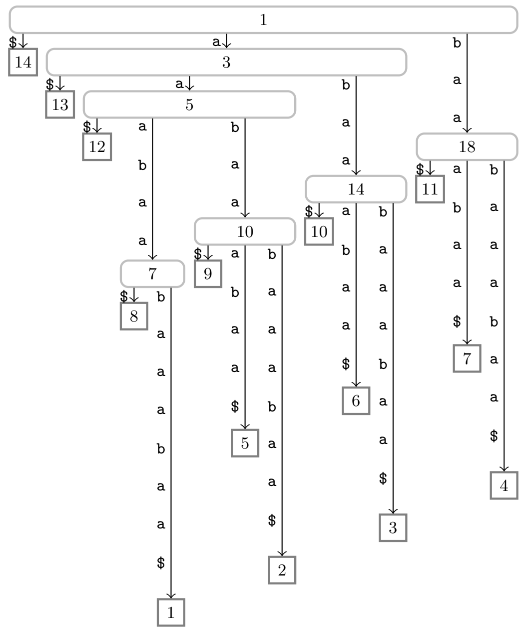

Figure 1: The suffix tree of `aaabaabaaabaa`\$. Internal nodes are labeled by their pre-order numbers, leaves by the text position where their respective suffix starts. The number of letters on an edge e is $c(e)$.

vice versa. Leaves are labeled *conceptionally* by the text position $1, \ldots, n$ where their corresponding suffix starts (see Figure 1). We write $\mathit{label}(\ell)$ for the label of a leaf ℓ. Reading the leaf labels from left to right returns the suffix array, which we denote by SA. We do neither store SA nor the leaf labels.

For description purposes, we define the conceptional function $c(e)$ returning, for each edge e, the length of e's label.

The BWT of T is the string BWT with $\mathsf{BWT}[j] = T[\mathsf{SA}[j] - 1]$ ($1 \leq j \leq n$, and $\mathsf{BWT}[j] = T[n]$ if $\mathsf{SA}[i] = 1$). Due to Belazzougui [9, Lemma 8], we can compute BWT, equipped with a rank structure, in $\mathcal{O}(n \lg \sigma)$ bits of space and linear time. The rank structure can answer $\mathsf{BWT}.\mathrm{rank}_c$ in constant time for a character $c \in \Sigma$ [9, Lemma 4].

We use the following methods on the ST topology that are well known to be computable in constant time (see [20]):

- $\mathit{level\_anc}(\ell, d)$ returns the ancestor of leaf ℓ at depth d,
- $\mathit{parent}(v)$ returns the parent of v,
- $v.\mathit{leaves}$ returns the maximal interval of leaves that are contained in the subtree rooted at v,
- $v.\mathit{child}(i)$ selects the i-th child of v, and
- $\mathit{lex}(\ell)$ returns the lexicographic order of the leaf ℓ, i.e., $\mathsf{root}.\mathit{leaves}[i] = \ell \Leftrightarrow \mathit{lex}(\ell) = i$.

Besides those basic tools, we need the following supplementary functions whose implementation details follow their descriptions:

- *head*(ℓ) retrieves the first character of the suffix whose starting position coincides with the label of the leaf ℓ. We can compute *head* in constant time by using a bit vector B_{H} of length n, marking the i-th bit with a one if $T[\mathsf{SA}[i]] \neq T[\mathsf{SA}[i-1]]$ for $i > 1$. Then $\mathit{head}(\ell) = B_{\mathrm{H}}.\operatorname{rank}_1(\mathit{lex}(\ell))$. (We assume that every character occurs at least once in T.) Having the text T and the ψ-array, we can compute B_{H} in linear time. Containing σ ones, B_{H} is compressible, taking $H_0(n, \sigma)\, n + o(n)$ bits.

- *smallest_leaf* returns the leaf with label 1. By a linear scan over the ψ-array, we can find the value $\mathsf{SA}^{-1}[1] =: \alpha$, so that $\psi^k[\alpha] = \mathsf{SA}^{-1}[k+1]$ for $0 \leq k \leq n-1$. We store α so that we can return *smallest_leaf* by $\mathsf{root}.\mathit{leaves}[\alpha]$.

- *next_leaf*(ℓ) returns the leaf labeled with $\mathit{label}(\ell) + 1$. We can compute it in constant time, since $\mathit{next\_leaf}(\ell) = \mathsf{root}.\mathit{leaves}[\psi[\mathit{lex}(\ell)]]$.

- *str_depth*(v) returns the string depth of an *internal* node. We use the ψ-array and the *head*-function to compute *str_depth*(v) in time proportional to the string depth (see Algorithm 1). Therefore, we take two different children of v (they exist since v is an internal node), and choose an arbitrary leaf in the subtree of each child. So we have two leaves representing two suffixes whose longest common prefix is the string depth of the lowest common ancestor (LCA) of both leaves. Our task is to compute the length of this prefix. To this end, we match the first characters of both suffixes by the *head*-function. If they match, we use ψ to move to the next pair of suffixes, and apply the *head*-function again. Informally, applying ψ strips the first character of both suffixes (like taking a suffix link). On a mismatch, we find the first pair of characters that does not belong to the path from the root to v (reading the labels of the edges along this path). We return the number of matched characters as the string depth.

4 Common Settings

We identify factors by text positions, i.e., we call a text position j the ***factor position*** of f_x ($1 \leq x \leq z$) iff the factor f_x starts at position j. A factor f_x may refer to either (LZ77) a previous text position j (called f_x's ***referred position***), or (LZ78) to a previous factor f_y (called f_x's ***referred factor***—in this case y is also called the ***referred index*** of f_x). If there is no suitable reference found for a given factor f_x with factor position j, then f_x consists of just the single letter $T[j]$. We call such a factor a ***fresh factor***. The other factors are called ***referencing factors***. Let z_{R} denote the number of referencing factors. An example is given in Figure 2.

Algorithm 1 Implementation of *str_depth*

Require: suffix tree node v
if v is an internal node **then**
  $\ell \leftarrow v.\mathit{child}(1)\,.\mathit{leaves.first}$
  $\ell' \leftarrow v.\mathit{child}(2)\,.\mathit{leaves.first}$ {$v.\mathit{child}(2)$ exists since v is an internal node}
  $m \leftarrow 0$
  while $\mathit{head}(\ell) = \mathit{head}(\ell')$ **do**
    $\ell \leftarrow \mathit{next\_leaf}(\ell)$
    $\ell' \leftarrow \mathit{next\_leaf}(\ell')$
    incr m
  end while
  return m
else if v is a leaf **then** {works only if $\mathit{label}(v)$ is available}
  return $n + 1 - \mathit{label}(v)$
end if

LZ77

	1	2	3	4	5	6
Factor	`a`	`aa`	`b`	`aabaa`	`abaa`	`$`
Coding	`a`	`1,2`	`b`	`2,5`	`3,4`	`$`

LZ78

	1	2	3	4	5	6	7
Factor	`a`	`aa`	`b`	`aab`	`aaa`	`ba`	`a$`
Coding	`a`	`1,a`	`b`	`2,b`	`2,a`	`3,a`	`1,$`

Figure 2: We parse the text `aaabaabaaabaa$` by both factorizations. The coding represents a fresh factor by a single character, and a referencing factor by a tuple with two entries. For LZ77, this tuple consists of the referred position and the number of characters to copy. For LZ78, it consists of the referred index and a new character.

4.1 Scaffold of the Algorithms

Common to our LZ77- and LZ78-factorization algorithms is the traversal of the compressed suffix tree. In more detail, they share a common framework, which we describe in the following by introducing some new keywords:

Witnesses. Witnesses are *internal* nodes that act as signposts for finding (LZ77) the referred position or (LZ78) the referred index of a factor. The number of witnesses z_{W} is at most the number of referencing factors z_{R}. We will enumerate the witnesses from 1 to z_{W} by a bit vector B_{W} on the BP of ST with a rank_1 support. So each witness has, along with its pre-order number, a so-called ***witness id*** (its B_{W}-rank).

Passes. Like the LZ77 algorithm in [14], we divide our algorithms in several passes. In a pass, we visit the leaves of ST in text position order. This is done by

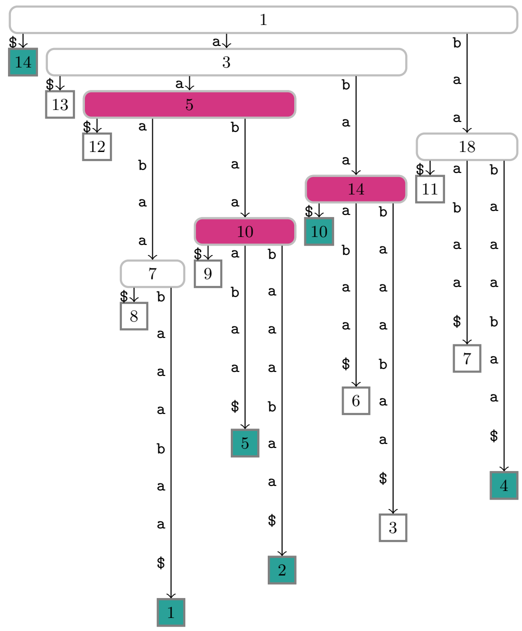

Figure 3: For the LZ77 parsing, we determine the witness nodes and the leaves corresponding to factors in Pass (a). Considering our running example, the witness nodes are the nodes with the pre-order numbers $5, 10$, and 14, and the leaves corresponding to factors have the labels $1, 2, 4, 5, 10$, and 14. Each witness w is the lowest ancestor of a leaf corresponding to a factor f with the property that the referred position of f is the label of a leaf contained in w's subtree. For instance, the leaf corresponding to the 5-th factor has the label 10. Its witness has pre-order number 14, leading to the leaf with label 3. So the referred position of the 5-th factor is 3. The length of the 5-th factor is the string depth of its witness.

using *smallest_leaf* and then calling successively *next_leaf*. The passes differ in how a leaf is processed. While processing a leaf ℓ, we want to access *label*(ℓ). Fortunately, we can track the label of the current leaf, since we start at the leaf with label 1.

Corresponding Leaves. We say that a leaf ℓ ***corresponds to*** the factor f if *label*(ℓ) is the beginning position of some factor f. During a pass, we keep track whether a visited leaf corresponds to a factor. To this end, for each leaf ℓ corresponding to a factor f, we compute the length of f while processing ℓ. This length tells us the number of leaves after ℓ (in *text order*) that do not correspond to a factor. By noting the next corresponding leaf, we know whether the current leaf is corresponding to a factor — remember that a pass selects leaves successively in *text order*, and *smallest_leaf* is always corresponding to the first factor.

Output Space. Given ST and ψ, we analyze our algorithms for both factorizations with respect to time and working space. We analyze both factorizations under the assumptions that either the output has to be stored explicitly in RAM, or that the algorithm must stream the output sequentially.

4.2 Loaded Data Structures

We need the ψ-array using $\mathcal{O}(n \lg \sigma)$ bits, B_{H} using $n + o(n)$ bits and ST's topology using $4n + o(n)$ bits. We spend $o(n)$ additional bits to create a $\mathrm{rank}_{()}$-support on the

Algorithm 2 LZ77 Pass (a)

```
ℓ ← smallest_leaf
p ← 1 {tracks next leaf corresponding to a factor}
repeat
   v ← parent(ℓ)
   while v ≠ root do
      if B_V[v] = 1 then {already visited?}
         if label(ℓ) = p then {if the current leaf corresponds to a factor}
            B_W[v] ← 1 {then this node is a witness}
            p ← p + str_depth(v) {determine next starting factor}
         end if
         break {on finding a visited node we stop}
      end if
      B_V[v] ← 1 {visit the node}
      v ← parent(v) {move upwards}
   end while
   incr p {factor is a single character}
   ℓ ← next_leaf(v)
until ℓ = smallest_leaf
```

BP sequence of ST for ranking leaves and internal nodes separately (a leaf in BP is represented by the sequence `()`). Additionally, the rightmost-parsing variant of the LZ77 factorization needs BWT. Our algorithms do *not* access the text T.

5 LZ77

We show that the LZ77 factorization can be computed with $(1 + H_0(n, z))n + z \lg n + o(n)$ additional bits of working space when streaming, or in $(1+H_0(n, z))n+2z \lg n+ o(n)$ additional bits of working space when storing the output. The rightmost parsing variant needs additionally $2n$ and n bits for streaming and storing the output, respectively.

LZ77 Passes. Common to all passes is the following procedure: For each visited leaf ℓ, we perform a leaf-to-top traversal, i.e., we visit every node on the path from ℓ to root. But we visit every node at most once, i.e., we break the leaf-to-top traversal on visiting an already visited node. Therefore, we use the bit vector B_{V} with which we mark a visited node. This bit vector is cleared before each pass. Since ST contains at most $n - 1$ internal nodes, a pass can be conducted in linear time.

5.1 Vanilla Algorithm

We do two passes:

(a) create B_{W} in order to determine the witnesses (see Algorithm 2), and

Algorithm 3 LZ77 Pass (b)

```
B_V.clear
B_W.add_rank_support
p ← 1 {tracks next leaf corresponding to a factor}
z_W ← B_W.rank_1(|W|)
W ← array of size z_W lg n {maps witness ids to text positions}
ℓ ← smallest_leaf
x ← 1 {count the x-th factors}
repeat
  v ← parent(ℓ)
  while v ≠ root do
    if B_V[v] = 1 then {Invariant: B_V[v] = 1 ∧ p = label(ℓ) ⇒ B_W(v) = 1}
      if label(ℓ) = p then {ℓ corresponds to a factor}
        output text position W[B_W.rank_1(v)]
        output factor length str_depth(v)
        p ← p + str_depth(v) {determine next starting factor}
        incr x
      end if
      break
    end if
    if B_W[v] = 1 then
      W[B_W.rank_1(v)] ← lex(ℓ)
    end if
    B_V[v] ← 1
    v ← parent(v)
  end while
  if lex(ℓ) = p then {We are currently processing a leaf of a fresh factor}
    output character head(ℓ)
    output factor length 1
    incr p
    incr x
  end if
  ℓ ← next_leaf(ℓ)
until ℓ = smallest_leaf
```

(b) stream the output by using an array mapping witness ids to text positions (see Algorithm 3).

Pass (a). We follow the approach from [14]. Determining the witnesses is done in the following way: Reaching the root from a leaf corresponding to a factor (while visiting only non-marked nodes) means that we found a fresh factor. Otherwise, assume that we visit an already visited node $u \neq \mathsf{root}$. If ℓ corresponds to a factor f, u *witnesses* the referred position of f. This means that there is a suffix starting before *label*(ℓ) having a prefix equal to the string read from the edge labels on the path from the root to u. Moreover, u is the lowest node in the set $\{\text{LCA of } \ell \text{ and } \ell' : B_{\text{V}}[\ell'] = 1\}$ comprising the lowest common ancestors of ℓ with all already visited leaves. So the factor corresponding to ℓ has to refer to a text position coinciding with the label of a leaf belonging to u's subtree. In order to find the referred position in the next pass, we mark u in B_{W}. Additionally, we compute the length of f with $\mathit{str\_depth}(u)$, and note the next factor position.

After this pass, we have determined the z_{W} witnesses by the '1's stored in B_{W}. We use the witnesses in the next pass to compute the referred positions (see Figure 3).

Pass (b). We clear B_{V}, create a rank-support on B_{W} and allocate an array W consuming $z_{\text{W}} \lg n$ bits. We use W to map a witness id to a text position (referred positions). Having this array as a working space, $W[w]$ becomes the text position of the leaf from which we visited the witness w in the first place. So we find the referred position of a referencing factor f in $W[w]$ when visiting w again from a different leaf corresponding to f. The length of f is the string depth of w. Since fresh factors consist of single characters, we can output a fresh factor by applying the *head*-function to its corresponding leaf.

Compressing B_{W}. Instead of directly marking nodes in B_{W}, we can allocate $z \lg n$ bits (we can use the space later for the array W) storing the pre-order numbers of all witnesses during Pass (a). Afterwards, we can create a compressed bit vector with constant time rank-support, representing B_{W}.

5.2 Trade-Off Variant

We can reduce W to $\varepsilon z \lg z$ by performing both passes $1/\varepsilon$ times. Therefore, we prematurely stop Pass (a) after counting ε many witnesses. We store the label j of the last visited leaf in order to resume Pass (a) after outputting the found factors. Since z_{W} is now ε, we need not modify Pass (b). When running Pass (a) again to capture the next ε witnesses (some may be the same), we suppress the marking in B_{W} when visiting leaves corresponding to referencing factors whose factor positions are at most j.

Since we run both passes $1/\varepsilon$ times we get $\mathcal{O}(n/\varepsilon)$ time overall.

5.3 Rightmost Parsing with Streaming

The referred position of a referencing factor f is not uniquely determined by Definition 3.1. In terms of the suffix tree topology, we can choose the label of all already visited leaves belonging to the subtree of f's witness. Looking at Figure 3, the text positions 3 and 6 are valid referred positions for the 5-th factor. From all possible referred positions, our algorithm chooses the smallest one. Since many encoders use the position difference between f's factor position and f's referred position, it is practically a good idea to choose the largest of the referred positions, also called the ***rightmost*** one. We can modify our algorithm to select the rightmost positions. To this end, during Pass (a), we additionally collect in a bit vector B_{C} the *label* of each leaf corresponding to a referencing factor. Further, we exchange Pass (b) with two passes:

(c) mark each leaf whose label is the (rightmost) referred position of a factor, and

(d) stream each factor to the output while maintaining the marked leaves in a helper array.

Pass (c) is performed in *reversed* text order. To reverse the order, we need the function $\mathit{prev\_leaf}(\ell)$ that returns the leaf with label $\mathit{lex}(\ell) - 1$. We can compute $\mathit{prev\_leaf}(\ell)$ by $B_{\mathrm{H}}.\mathrm{select}_1(c) + \mathsf{BWT}.\mathrm{rank}_c(\mathit{label}(\ell))$ in constant time, where $c = \mathit{head}(\ell)$. By exchanging *next_leaf* with *prev_leaf*, we can conduct a pass in reverse text order. Tracking the label of the current leaf is done similarly to a regular pass, since we start by taking *smallest_leaf* and applying *prev_leaf*. We can check whether a leaf corresponds to a factor by accessing $B_{\mathrm{C}}[\mathit{label}(\ell)]$.

A detailed description of both passes follows:

Pass (c). Before starting the pass, we create a bit vector B_{L} to mark each leaf whose lexicographic order is a referred position. We allocate an array W with $z_{\mathrm{W}} \lg n$ bits, storing a leaf (by its lexicographic order) corresponding to a referencing factor for each witness id. W and B_{L} are computed in the following way: Assume that we visit a witness w from a leaf ℓ. If $W[w]$ is empty and ℓ corresponds to a (referencing) factor, then set $W[w] \leftarrow \ell$. If $W[w]$ is non-empty, then mark ℓ in B_{L}; if ℓ corresponds to a factor, then set $W[w] \leftarrow \ell$, otherwise clear $W[w]$. In the end, B_{L} marks all leaves whose labels are the rightmost positions.

Pass (d). We clear W before starting the final pass in *regular* text order. This time, W is used to map a witness id to a text position. When accessing a witness w by a leaf ℓ marked by B_{L}, we set $W[w] \leftarrow \mathit{label}(\ell)$, possibly overwriting an old entry.

Assume now that we visit a witness w from a leaf ℓ. If ℓ corresponds to a referencing factor, then $W[w]$ is non-empty (otherwise, ℓ would be the first leaf from which we explored w, a contradiction for ℓ corresponding to a *referencing* factor). In other words, the referred position of the factor corresponding to ℓ is $W[w]$. So we can output all factors in text order.

5.4 Rightmost Parsing With Output

Similar to Section 5.3, we do two passes:

(e) compute the referred positions, and

(f) store the factor lengths.

We create the bit vector B_{C} like in Section 5.3.

Pass (e). The pass is conducted in *reverse* order, allocating the same array W as in Section 5.3. We additionally use an array O with $z_{\mathrm{R}} \lg n$ bits to store the referred position of each referencing factor. We now show how to compute O: Assume that we visit the witness w from a leaf ℓ. If $W[w]$ is empty and ℓ corresponds to a referencing factor (marked by B_{C}), then we set $W[w] \leftarrow B_{\mathrm{C}}.\mathrm{rank}_1(\mathit{label}(\ell))$. If $W[w] = u$ is non-empty, then the u-th leaf (rank with respect to B_{C}) corresponds to a referencing factor f whose witness is w. Because ℓ's label is the referred position of f, we write $O[u] \leftarrow \mathit{label}(\ell)$. We set $W[w] \leftarrow B_{\mathrm{L}}.\mathrm{rank}_1(\mathit{label}(\ell))$ if ℓ corresponds to a referencing factor, otherwise we clear $W[w]$.

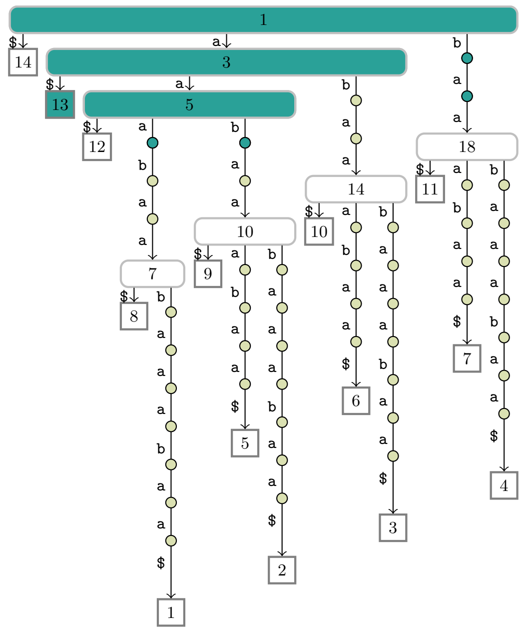

Figure 4: The small rounded nodes in the depicted tree represent the implicit suffix trie nodes. Dark colored nodes represent the nodes of the LZ78 trie, which is a subtree of the suffix trie.

Having random access to an additional output space with $z \lg z$ bits, we can store the factor lengths during Pass (e). Otherwise, we retrieve the factor lengths by a supplementary pass (Pass (f)), during which we can store these values in W (extending W to $z \lg n$ bits).

6 LZ78

A natural representation of the LZ78 factors is a trie, the so-called ***LZ trie***. Each node in the trie represents a factor and is labeled by its index. If the x-th factor refers to the y-th factor, then there is a node u having a child v such that u and v have the unique labels y and x, respectively. The edge (u, v) is labeled by the last character of the x-th factor (the newly introduced character). A node with a label x is the child of the root iff the x-th factor is a fresh factor.

The LZ trie representation is used in the algorithms presented below. While the streaming algorithm computes the topology of the LZ trie, the storing variant explicitly creates the LZ trie for querying. We can compute the factorization with $5n + z \lg z + o(n)$ additional bits of working space when streaming the output, or with $6n + z(\lg \sigma + \lg z + 3) + o(n)$ additional bits of working space when storing the LZ trie explicitly, labeling each node of the trie by a factor index.

6.1 Superimposition

Main idea is the superimposition of the suffix trie on the suffix tree, borrowed from Nakashima et al. [13]: Since the LZ trie is a subtree of the suffix trie, we want to represent the LZ (trie) nodes during the LZ78 parsing (see Figure 4). Regarding the suffix tree, the LZ nodes are either already represented by an ST node (explicit), or lie on an ST edge (implicit). To ease explanation, we identify each edge $e = (u, v)$ of ST uniquely with its ending node v, i.e., we implicitly convert between the edge e and its in-going node v (each node except root is associated with an edge). In order to address all LZ nodes, we keep track of how far an edge on the suffix tree got explored during the parsing. To this end, for an edge $e = (u, v)$, we define the ***exploration counter*** $0 \leq n_v \leq c(e)$ storing how far e is explored. If $n_v = 0$, then the factorization has not (yet) explored e, whereas $n_v = c(e)$ tells us that we have already reached v. Unfortunately, storing n_v in an integer array for all edges costs us $2n \lg n$ bits. We therefore represent these values differently, based on the number of leaves in the subtree rooted at v: If v's subtree has more than $\lg n$ leaves, we call v ***giant***, otherwise ***small***. We mark each giant node v by a bit vector B_{G}, and store n_v in an integer array G. The array G consumes at most $(n/\lg n) \lg n = n$ bits, since ST has n leaves.

The exploration counters of the small nodes are stored implicitly by a bit vector marking some leaves (corresponding to factors) and the *popcount* function: During a pass, when exploring a new factor on the in-going edge of a small node v, we mark the currently accessed leaf in a bit vector B_{C}. By *popcount*, we can count how many leaves had been accessed belonging to the subtree rooted at v. This number is exactly n_v. Moreover, we can compute n_v in constant time, since the cardinality of $v.\mathit{leaves}$ is at most $\lg n$. After fully exploring the edge of v, we erase the area in B_{C} representing the leaves contained in v's subtree. By doing so, the counter n_u of every child u of v is reset.

Applying this procedure during a pass, we can collect n_v of each giant node v, and n_v of each small node v whose in-going edge got partially explored (since we clear parts of B_{C} after full exploration).

6.2 Streaming Variant

We do two passes:

(a) create B_{W} so we can address the witnesses (see Algorithm 4), and

(b) stream the output by using a helper array mapping witness ids to factor indices.

We explain the passes in detail, after introducing their commonality and a helpful lemma:

LZ78 Passes. Since referencing factors address factor indices (z options) instead of text positions (n options), we are only interesting in the leaves corresponding to a factor. Starting with *smallest_leaf*, which corresponds to the first factor, we can compute the length of the factor corresponding to the current accessed leaf so that we know how many leaves we will skip.

Algorithm 4 LZ78 Pass (a)

$B_\mathrm{G}[v] \leftarrow 1$ for all nodes v having more than $\log n$ leaves in its subtree
$G \leftarrow$ integer array with $n/\lg n$ entries, i.e., n bits.
$\ell \leftarrow$ *smallest_leaf*
repeat
  $v \leftarrow$ root
  $d \leftarrow 0$ {node depth}
  repeat {find first edge on path from root to ℓ that is not fully explored}
    incr d
    $v \leftarrow$ *level_anc*(ℓ, d)
  until $B_\mathrm{V}[v] = 0$
  $B_\mathrm{W}[v] \leftarrow 1$ {v is a witness}
  $u \leftarrow$ *parent*(v) {new factor is on the edge (u, v)}
  if $B_\mathrm{G}[v] = 1$ **then**
    $m \leftarrow G[B_\mathrm{G}.\operatorname{rank}_1(v)]$
  else
    $m \leftarrow$ *popcount*$(B_\mathrm{C}[v.\mathit{leaves}])$ {call *lex* for the left- and rightmost leaf}
  end if
  Let $\ell' \neq \ell$ be a leaf whose LCA with ℓ is v
  $s \leftarrow$ *str_depth*(u)
  $\ell = (\mathit{next\_leaf})^{m+s+1}[\ell]$
  $\ell' = (\mathit{next\_leaf})^{m+s+1}[\ell']$
  if *head*$(\ell) \neq$ *head*(ℓ') **then** {edge (u, v) now fully explored}
    $B_\mathrm{V}[v] \leftarrow 1$ {set v as fully explored}
    $B_\mathrm{C}.\mathit{clear}[\mathrm{v}.\mathit{leaves}]$ {reset the counting so that we can work with the children of v}
  else if $B_\mathrm{G}[v] \neq 1$ **then** {edge (u, v) has at least one character unexplored}
    $B_\mathrm{C}[\ell] \leftarrow 1$ {increment n_e for the small node v}
  end if
  if $B_\mathrm{G}[v] = 1$ **then** {increment n_e for the giant node v}
    incr $G[B_\mathrm{G}.\operatorname{rank}_1(v)]$
  end if
until $\ell =$ *smallest_leaf*

Lemma 6.1 ([14, Lemma 4])**.** *Let $e = (u, v)$ be an* ST *edge, and u the parent of the node v. Then $n_v \leq \min(c(e), l)$, where l is the number of leaves of the subtree rooted at v.*

Pass (a). Main goal of this pass is to determine the topology of the LZ trie with respect to the superimposition. Starting with an LZ trie consisting only of the root, we build the LZ trie successively by filling up the exploration values. If the exploration value of an edge is filled up, we mark its in-going node in a bit vector B_V.

Assume that we visit a leaf ℓ. We want to find the first edge on the path from root to ℓ that is either unexplored or partially explored. By invoking level ancestor queries, we traverse from the root to an edge $e = (u, v)$, where $n_v < c(e)$ and u is

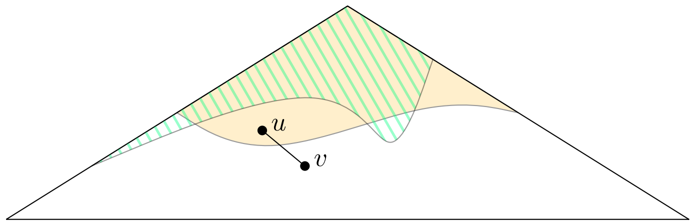

Figure 5: Our LZ78 algorithm divides ST by two boundaries: Nodes having at least $\lg n$ leaves in their subtree (hatched upper cone), and (partially) explored edges belonging to the LZ trie (colored upper cone). The nodes whose exploration values are stored implicitly are directly below the fringe of the LZ trie. During the passes, we always search for an ST edge (u, v) crossing the boundary of the already discovered part.

(already) represented as a node in the LZ trie. If $n_v = 0$, we mark v by B_{W}, i.e., we make v a witness (the idea is that the edge e is *superimposed* by some LZ nodes). Regardless that, we add a new factor by incrementing n_v. If the edge e now got fully explored, we additionally mark v in B_{V}. Whether the edge e got fully explored, can be determined with the *next_leaf* function: First, if v is a leaf, the edge (u, v) can be explored at most once (by Lemma 6.1). Otherwise, we choose a leaf ℓ' such that the LCA of ℓ and ℓ' is v. The idea is that $\mathit{str\_depth}(u)$ is the longest common prefix of two suffixes corresponding to two leaves (e.g., ℓ and ℓ') having v as their LCA. So we can compare the l-th character of both respective suffixes by applying *next_leaf* l-times on both leaves before using the *head*-function. With $l := \mathit{str\_depth}(u) + n_v + 1$ we can check whether the edge (u, v) got fully explored. Additionally, we can determine the label of the next corresponding factor. Although we apply *next_leaf* as many times as the factor length, we still get linear time overall, because concatenating all factors yields the text T.

Pass (b). This pass is nearly identical to the first one. We explore the LZ trie nodes again, but this time we already have the witnesses. So we keep B_{W}, but clear the exploration counters and B_{V}.

For finding the referred indices, we create an array W with $z_{\mathrm{W}} \lg z$ bits to store a factor index for each witness id. The witness ids are determined by B_{W}, the factor indices by a counting variable tracking the number of visited corresponding leaves, i.e., the number of processed factors.

Assume that we visit the leaf ℓ corresponding to the x-th factor, i.e., ℓ is the x-th visited corresponding leaf. Again by level ancestor queries, we determine the edge $e = (u, v)$ on the path from the root to ℓ, where u is in B_{V} and v not. We look at the value $y := W[B_{\mathrm{W}}.\mathrm{rank}_1(v)]$. If y is undefined, then x is the index of a fresh factor. Otherwise, the x-th factor refers to the y-th factor. In any case, we set $W[B_{\mathrm{w}}.\mathrm{rank}_1(v)] \leftarrow x$, and increment n_v (thus exploring the LZ trie like before).

Like before, when v's in-going edge got fully explored, we mark v in B_{V}. But this time, if the child of v on the path from v to ℓ is a witness, we additionally set $W[B_{\mathrm{W}}.\mathrm{rank}_1(\mathit{level\_anc}(\ell, \mathit{depth}(v)+1))] \leftarrow x$. This prepares the referred index of the *first factor* on the path from v to ℓ (if it exists) — with first factor we mean that the LZ node representing this factor is the *first* node that superimposes the edge from v

to $\mathit{level\_anc}(\ell, \mathit{depth}(v)+1)$.

So far, we can output the referred index of the x-th factor, if it exists. We get the new character of the x-th factor (i.e., the last letter of the factor) by accessing the leaf ℓ' that is (with respect to text order) *before* the leaf corresponding to the $(x+1)$-th factor; then we can output the new character by $\mathit{head}(\ell')$.

6.3 Explicitly storing the LZ trie

After Pass (a), we can construct the LZ trie topology: While we already store the exploration counters for all *partial* discovered edges, we can compute the length of the *fully* discovered edges with *str_depth*.

We construct the LZ trie by a depth first search (DFS) of ST starting at root. While traversing ST we generate a BP representation of the LZ trie. We mark an explicit LZ node in a bit vector B_{E} of length z. For each edge $e=(u,v)$ we create $c(e)$ LZ trie nodes, if v belongs to B_{V}, or create n_v LZ nodes otherwise.

Assume that we create the LZ nodes u' and v' on the ST edge (u,v), where u' is the parent of v'. We can obtain the label of the edge (u',v') consisting of a character by invoking *next_leaf* and *head*: Given that l is the string depth of v' in the LZ trie, applying l-times *next_leaf* to a leaf ℓ in the subtree rooted at v returns a leaf ℓ' whose $\mathit{head}(\ell')$-value is the label in question.

Overall, we can compute the LZ trie in $\mathcal{O}(n)$ time, storing its BP representation along with the edge labels in $2z + z\lg\sigma$ bits.

Annotating the LZ nodes with the factor indices is done by an additional pass. Therefore, we need to jump from the fully-explored ST nodes to their LZ trie node representations: We identify the nodes marked in B_{V} with the explicit LZ trie nodes, and use this mapping to access the implicit LZ nodes efficiently. More precisely, the LZ trie nodes marked by B_{E} can be mapped isomorphic via rank / select to the ST nodes marked by B_{V}. We copy B_{V} to a new bit vector B_{U}, add a rank-support on B_{U}, and a select-support on B_{E}. Now assume that there is a B_{U} marked ST node v having an out-going edge with a label comprising k letters. The LZ trie contains a node v' marked with B_{E} such that $v' = B_{\mathrm{E}}.\,\mathrm{select}_1(B_{\mathrm{U}}.\,\mathrm{rank}_1(v))$ (conceptionally $B_{\mathrm{E}}.\,\mathrm{rank}_1(v') = B_{\mathrm{U}}.\,\mathrm{rank}_1(v)$). Moreover, there are k implicit nodes below v' forming a unary tree (also called linear graph or path graph). By the BP representation we can address them with $v'.\,\mathit{descendant}(j)$ for $0 \le j \le k$ in constant time (with $v'.\,\mathit{descendant}(0) = v'$).

Pass (c). We create an array W with $z\lg z$ bits to store a factor index for each LZ node. We recompute B_{V} and the exploration values. We count the current factor index with a variable x. Assume that we visit the leaf ℓ corresponding to the x-th factor. By *level_anc*, we find the edge $e=(u,v)$ in ST on the path from root to ℓ, where u belongs to B_{V}, but v not. We set $W[B_{\mathrm{E}}.\,\mathrm{select}_1(B_{\mathrm{V}}.\,\mathrm{rank}_1(u)).\,\mathit{descendant}(n_v)] \leftarrow x$.

Acknowledgements

We thank Veli Mäkinen for outlining us the differences between [7] and [9], and Johannes Fischer for some helpful comments.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[2] ——, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[3] M. Crochemore, A. Langiu, and F. Mignosi, "The rightmost equal-cost position problem," in *Data Compression Conference, 2013*, March 2013, pp. 421–430.

[4] D. Kosolobov, "Faster Lightweight Lempel-Ziv Parsing," in *MFCS*. Springer Berlin Heidelberg, 2015, vol. 9235, p. 432–444.

[5] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lightweight Lempel-Ziv Parsing," in *Experimental Algorithms*, 2013, p. 139–150.

[6] D. Belazzougui and S. J. Puglisi, "Range predecessor and lempel-ziv parsing," *ArXiv CoRR*, vol. abs/1507.07080, 2015.

[7] D. Belazzougui, "Linear time construction of compressed text indices in compact space," ser. STOC '14. ACM, 2014, pp. 148–193.

[8] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep., 1994.

[9] D. Belazzougui, "Linear time construction of compressed text indices in compact space," *ArXiv CoRR*, vol. abs/1401.0936, 2015.

[10] P. Ferragina, I. Nitto, and R. Venturini, "On the bit-complexity of lempel-ziv compression," *SIAM J. Comput.*, vol. 42, no. 4, pp. 1521–1541, 2013.

[11] J. Fischer and P. Gawrychowski, "Alphabet-Dependent String Searching with Wexponential Search Trees," in *CPM*, 2015, p. 160–171.

[12] J. Jansson, K. Sadakane, and W. Sung, "Linked dynamic tries with applications to LZ-compression in sublinear time and space," *Algorithmica*, vol. 71, no. 4, pp. 969–988, 2015.

[13] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda, "Constructing LZ78 tries and position heaps in linear time for large," *Inform. Process. Lett.*, 2015.

[14] J. Fischer, T. I, and D. Köppl, "Lempel ziv computation in small space (LZ-CISS)," in *CPM*, 2015, pp. 172–184.

[15] J. I. Munro, "Tables," in *Proc. FSTTCS*, ser. LNCS, vol. 1180. Springer, 1996, pp. 37–42.

[16] R. Raman, V. Raman, and S. R. Satti, “Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets,” *ACM Trans. Algorithms*, vol. 3, no. 4, Nov. 2007.

[17] R. Grossi and J. S. Vitter, “Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract),” ser. STOC. New York, NY, USA: ACM, 2000, pp. 397–406.

[18] G. J. Jacobson, “Space-efficient static trees and graphs,” in *Proc. FOCS*. IEEE Computer Society, 1989, pp. 549–554.

[19] K. Sadakane, “Compressed Suffix Trees with Full Functionality.” *Theory of Computing Systems*, vol. 41, no. 4, pp. 589–607, 2007.

[20] G. Navarro and K. Sadakane, “Fully Functional Static and Dynamic Succinct Trees,” *ACM Trans. Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, May 2014.

A Appendix

A.1 Overview of used data structures

While describing both factorization algorithms, we used several data structures, among others bit vectors, some with rank or select-support, to achieve the small space bounds. We give here an overview (see also Table 1). We denote bit vectors with B_α for some letter α. We use for both factorizations

- B_{H} for the *head*-function,
- B_{W} marking all witness nodes,
- the array W mapping witness ids to
  - (LZ77) text positions, or
  - (LZ78) factor indices.

In LZ77 we use

- B_{V} marking visited nodes, and

In the LZ77 rightmost parsing variant we use

- B_{C} marking leaves corresponding to referencing factors.
- B_{L} marking leaves whose labels are referred positions.

In LZ78 we use

- B_{C} marking corresponding leaves (is used as a counter),
- B_{G} marking nodes with at least $\lg n$ leaves in its subtree,
- B_{V} marking suffix tree nodes represented in the LZ trie (their ingoing edges got fully explored),
- the array G storing $n/\lg n$ numbers binary,
- B_{E} marking explicit LZ trie nodes, and
- B_{U} as a copy of B_{V}.

Common

Name	bits	rank	select	compress
B_{W}	$H_0(n, z)n + o(n)$	○		○
B_{H}	$H_0(n, \sigma)n + o(n)$	○	○	○
ST	$4n + o(n)$			
ψ	$\mathcal{O}(n \lg \sigma)$			

LZ77

Name	bits	(a)	(b)	(c)	(d)	(e)	rank	select	compress
B_{V}		○	○	○	○	○			
B_{C}	$H_0(n, z)n$			○	○	○			○
B_{L}				○					○
W	$z \lg n$	○	○	○	○	○			
BWT	$\mathcal{O}(n \lg \sigma)$			○	○	○	○		
O	$z \lg n$					○			

LZ78

Name	bits	(a)	(b)	(c)	rank	select
B_{C}		○	○	○		
B_{G}	$n + o(n)$	○	○	○	○	
B_{V}		○	○	○		
B_{E}	$z + o(z)$			○		○
B_{U}				○		
G		○	○	○		
W	$z \lg z$	○	○	○		

Table 1: Additional data structures used while computing a LZ77/78 factorization. The letters written in brackets represent a pass (e.g., (d) refers to Pass (d) of LZ77). The number of bits is omitted if it is exactly n. Circles symbolize that the data structure is used during a pass, or that it is used with a rank or select structure, or that the data structure is compressible.