$S_{\mathsf{SA}[m-1]}$ and $S_{\mathsf{SA}[m]}$ because $|\mathsf{lcp}(S_{\mathsf{SA}[m-1]}, S_{\mathsf{SA}[m]})| = \mathsf{LCP}[m] = \ell < \ell + 1 = |S[i..i+\ell]|$. Consequently, $S[i..i+\ell]$ is not a common prefix of $S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[j]}$. All in all, $|\mathsf{lcp}(S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[j]})| = \ell$. □

As in Chapter 3, we say that an algorithm has time complexity $\langle p(n), q(n)\rangle$ if its preprocessing time is $p(n)$ and its query is time $q(n)$.

Lemma 4.2.8 *There is an $\langle O(n), O(1)\rangle$-time algorithm for answering longest common prefix queries between two suffixes of a string S.*

Proof We must show that after a linear-time preprocessing, $|\mathsf{lcp}(S_i, S_j)|$ can be computed in constant time for all positions $1 \le i \le j \le n$. Given string S of length n, one can compute the corresponding arrays SA, ISA, and LCP in $O(n)$ time. Moreover, the LCP-array can be preprocessed in linear time so that range minimum queries can be answered in constant time; see Section 3.3. For $i = j$, we have $|\mathsf{lcp}(S_i, S_j)| = |S_i|$. Otherwise, for $i \ne j$, we have

$$|\mathsf{lcp}(S_i, S_j)| = \begin{cases} \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{ISA}[i]+1, \mathsf{ISA}[j])], & \text{if } \mathsf{ISA}[i] < \mathsf{ISA}[j] \\ \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{ISA}[j]+1, \mathsf{ISA}[i])], & \text{if } \mathsf{ISA}[j] < \mathsf{ISA}[i] \end{cases}$$

This is a direct consequence of Lemma 4.2.7 because the indices $i' = \mathsf{ISA}[i]$ and $j' = \mathsf{ISA}[j]$ satisfy $S_{\mathsf{SA}[i']} = S_i$ and $S_{\mathsf{SA}[j']} = S_j$. □

Corollary 4.2.9 *There is an $\langle O(n), O(1)\rangle$-time algorithm for answering longest common suffix queries between two prefixes of a string S.*

Proof Observe that $\mathsf{lcs}(S[1..i], S[1..j]) = \mathsf{lcp}(S^{rev}_{n-i+1}, S^{rev}_{n-j+1})$, where S^{rev} denotes the reverse string of S. This, in combination with the fact that S^{rev} can be obtained in linear time from S, implies that $\mathsf{lcs}(S[1..i], S[1..j])$ can also be computed in constant time after a linear-time preprocessing. □

4.3 The lcp-interval tree

Most concepts of this section originate from Abouelhoda et al. [1]. The idea to use RMQs in this context stems from Fischer and Heun [108].

To see the usefulness of lcp-intervals, let us have a second look at the enhanced suffix array of the string $S = ctaataatg$, which is replicated in Figure 4.13. By definition 4.1.3, the a-interval is the interval $[1..4]$, the aa-interval is $[1..2]$, and the aat-interval is also $[1..2]$. By contrast, there is no substring ω of S so that the interval $[1..3]$ is an ω-interval. The next definition allows us to identify such intervals solely by means of the LCP-array. The declarations $\mathsf{LCP}[1] = -1$ and $\mathsf{LCP}[n+1] = -1$ ensure that the definition also covers the interval $[1..n]$.

i	SA	ISA	LCP	$S_{\mathsf{SA}[i]}$
1	3	5	-1	*aataatg*
2	6	7	3	*aatg*
3	4	1	1	*ataatg*
4	7	3	2	*atg*
5	1	8	0	*ctaataatg*
6	9	2	0	*g*
7	2	4	0	*taataatg*
8	5	9	4	*taatg*
9	8	6	1	*tg*
10			-1	

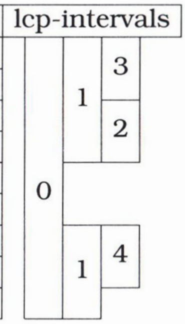

Figure 4.13: Enhanced suffix array and lcp-intervals of $S = ctaataatg$.

Definition 4.3.1 An interval $[i..j]$, $1 \leq i < j \leq n$, in an LCP-array is called an *lcp-interval of lcp-value* ℓ if and only if

1. $\mathsf{LCP}[i] < \ell$,
2. $\mathsf{LCP}[k] \geq \ell$ for all k with $i+1 \leq k \leq j$,
3. $\mathsf{LCP}[k] = \ell$ for at least one k with $i+1 \leq k \leq j$,
4. $\mathsf{LCP}[j+1] < \ell$.

We will also use the shorthand ℓ-$[i..j]$ for an lcp-interval $[i..j]$ of lcp-value ℓ, and $[i..j]$ will be called ℓ-interval. Every index k, $i+1 \leq k \leq j$, with $\mathsf{LCP}[k] = \ell$ is called ℓ-index (or lcp-index) of $[i..j]$. The set of all ℓ-indices of an ℓ-interval $[i..j]$ will be denoted by $\ell Indices(i,j)$. Furthermore, we will say that the lcp-interval ℓ-$[i..j]$ represents the string $\omega = S[\mathsf{SA}[i]..\mathsf{SA}[i]+\ell-1]$, where ω is the longest common prefix of the suffixes $S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[i+1]}, \ldots, S_{\mathsf{SA}[j]}$.

For ease of presentation, it is useful to ensure that the interval $[1..n]$ is always an lcp-interval of lcp-value 0. By Definition 4.3.1, this is the case if and only if there is at least one k with $2 \leq k \leq n$ so that $\mathsf{LCP}[k] = 0$. This in turn is the case if and only if the string S contains at least two different characters. Thus, we tacitly assume that \$ is appended to strings containing only one character.

As an example, consider Figure 4.13. $[1..4]$ is a 1-interval because $\mathsf{LCP}[1] = -1 < 1$, $\mathsf{LCP}[4+1] = 0 < 1$, $\mathsf{LCP}[k] \geq 1$ for all k with $2 \leq k \leq 4$, and $\mathsf{LCP}[3] = 1$. Furthermore, the lcp-interval 1-$[1..4]$ represents the string a and $\ell Indices(1,4) = \{3\}$. Similarly, the lcp-interval 3-$[1..2]$ represents the string aat. By definition, the string aa is not represented by an lcp-interval. This is because each lcp-interval $[i..j]$ only represents the longest common prefix of the suffixes $S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[i+1]}, \ldots, S_{\mathsf{SA}[j]}$. So the lcp-interval $[1..2]$ represents aat and not aa.

Lemma 4.3.2 *Two lcp-intervals ℓ-$[i..j] \neq m$-$[p..q]$ cannot overlap, i.e., one of the following cases must hold:*

- *$[i..j]$ is a subinterval of $[p..q]$, i.e., $p \leq i < j \leq q$.*
- *$[p..q]$ is a subinterval of $[i..j]$, i.e., $i \leq p < q \leq j$.*
- *$[i..j]$ and $[p..q]$ are disjoint, i.e., $j < p$ or $q < i$.*

Proof Suppose to the contrary that $[i..j]$ and $[p..q]$ overlap, i.e., $i < p \leq j < q$ (the case $p < i \leq q < j$ is symmetric). By Definition 4.3.1, we have

1. $\mathsf{LCP}[i] < \ell$
2. $\mathsf{LCP}[p] \geq \ell$
3. $\mathsf{LCP}[j+1] < \ell$
4. $\mathsf{LCP}[p] < m$
5. $\mathsf{LCP}[j+1] \geq m$
6. $\mathsf{LCP}[q+1] < m$

The combination of (2) and (4) yields $\ell \leq \mathsf{LCP}[p] < m$, while the conjunction of (3) and (5) yields $m \leq \mathsf{LCP}[j+1] < \ell$. In summary, we obtain $\ell < m < \ell$. This contradiction shows the lemma. □

Definition 4.3.3 An m-interval $[p..q]$ is said to be *embedded* in an ℓ-interval $[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq p < q \leq j$) and $m > \ell$.[5] The ℓ-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 $[i..j]$ (vice versa, $[i..j]$ is the *parent interval* of $[p..q]$). This parent-child relationship constitutes a tree, which we call *lcp-interval tree.*

For instance, continuing the example of Figure 4.13, the child intervals of 1-$[1..4]$ are 3-$[1..2]$ and 2-$[3..4]$. The whole lcp-interval tree is shown in Figure 4.14. The root of an lcp-interval tree is always the 0-interval $[1..n]$. The lcp-interval tree of Figure 4.14 also contains singleton intervals, which are defined as follows.

[5]Note that we cannot have both $i = p$ and $j = q$ because $m > \ell$.

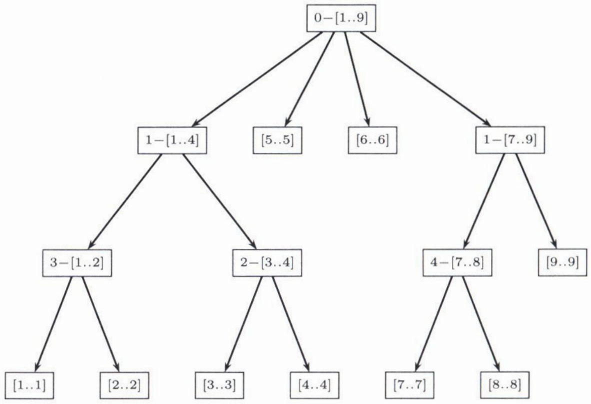

Figure 4.14: The lcp-interval tree for $S = ctaataatg$.

Definition 4.3.4 An interval $[k..k]$ is called *singleton interval.* The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ that contains k.

How much space does an lcp-interval tree occupy? Clearly, there are exactly n singleton-intervals, hence n leaves. As each internal node of an lcp-interval tree is branching, there can be at most $n-1$ internal nodes. Since the representation of a node needs at most three numbers, a node can be represented in constant space. It is readily seen that the number of edges is one less than the number of nodes. Consequently, there are at most $2n-2$ edges because there are at most $2n-1$ nodes in the lcp-interval tree. Since the edges are not labeled, we can surely represent each edge in constant space. To sum up, an lcp-interval tree requires only linear space. However, we will not construct this tree *explicitly*. As we shall see, it is possible to traverse this tree without constructing it.

4.3.1 Finding child and parent intervals

The next lemma shows how to determine child intervals.

Lemma 4.3.5 *Let $[i..j]$ be an ℓ-interval. If $i_1 < i_2 < \cdots < i_k$ are the ℓ-indices 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).*

Proof Let $[p..q]$ be a non-singleton interval out of the intervals $[i..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_k..j]$ and let $m = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(p + 1, q)]$. Since none of the indices $p+1, \ldots, q$ is an ℓ-index, it follows from Definition 4.3.1 that $\mathsf{LCP}[k] > \ell$ for all k with $p+1 \leq k \leq q$. Hence $m > \ell$. We claim that $[p..q]$ is an m-interval.

Note that $\mathsf{LCP}[p] < \ell$ if $p = i$ and $\mathsf{LCP}[p] = \ell$ if $p \neq i$. Analogously, $\mathsf{LCP}[q+1] < \ell$ if $q = j$ and $\mathsf{LCP}[q+1] = \ell$ if $q \neq j$. We have

1. $\mathsf{LCP}[p] \leq \ell < m$,
2. $\mathsf{LCP}[k] \geq m$ for all k with $p + 1 \leq k \leq q$,
3. $\mathsf{LCP}[k] = m$ for $k = \mathsf{RMQ}_{\mathsf{LCP}}(p + 1, q)$
4. $\mathsf{LCP}[q + 1] \leq \ell < m$.

By Definition 4.3.1, $[p..q]$ is an m-interval. To show that m-$[p..q]$ is a child interval of ℓ-$[i..j]$, we must prove that there is no lcp-interval embedded in $[i..j]$ that encloses $[p..q]$. For a proof by contradiction, suppose that the lcp-interval r-$[lb..rb]$ is embedded in $[i..j]$ and encloses $[p..q]$. We have $m > r > \ell$, and at least one of the following cases must hold: (a) $lb < p < q \leq rb$ or (b) $lb \leq p < q < rb$. We prove the lemma for case (a); the other case follows similarly. By Definition 4.3.1, it follows that $\mathsf{LCP}[k] \geq r > \ell$ for all k with $lb + 1 \leq k \leq rb$. In particular, $\mathsf{LCP}[p] > \ell$. This, however, contradicts the fact that $\mathsf{LCP}[p] \leq \ell$. Consequently, m-$[p..q]$ is a child interval of ℓ-$[i..j]$.

Now suppose that $[p..q]$ is a singleton interval, i.e., $p = q$. Obviously, at least one of the indices p and $p + 1$ must be an ℓ-index. That is, $\mathsf{LCP}[p] = \ell$ or $\mathsf{LCP}[p + 1] = \ell$ (or both). One can show that there is no lcp-interval $[lb..rb]$ that is embedded in $[i..j]$ and encloses $[p..p]$ (the indirect proof is verbatim the same as above). Therefore, ℓ-$[i..j]$ is the smallest lcp-interval that contains $[p..p]$, that is, ℓ-$[i..j]$ is the parent interval of $[p..q]$. □

As an example, we compute the child intervals of the lcp-interval 0-$[1..9]$ of the LCP-array from Figure 4.13. The 0-indices are (in ascending order) 5, 6, and 7. Thus, the child intervals of 0-$[1..9]$ are $[1..4]$, $[5..5]$, $[6..6]$, and $[7..9]$.

Exercise 4.3.6 Implement a procedure that takes an lcp-interval as input and returns the list of its child intervals.

We employ two auxiliary arrays $\mathtt{PSV}_{\mathsf{LCP}}$ and $\mathtt{NSV}_{\mathsf{LCP}}$ to explain how the parent interval of an lcp-interval can be determined.

Definition 4.3.7 For any index $2 \leq i \leq n$, we define

$$\mathtt{PSV}_{\mathsf{LCP}}[i] = \max\{j \mid 1 \leq j < i \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$

and

$$\mathtt{NSV}_{\mathsf{LCP}}[i] = \min\{j \mid i < j \leq n+1 \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$

i	SA	LCP	$S_{\mathsf{SA}[i]}$	$\mathtt{PSV}_{\mathsf{LCP}}$	$\mathtt{NSV}_{\mathsf{LCP}}$
1	3	-1	*aaacatat*		
2	4	2	*aacatat*	1	3
3	1	1	*acaaacatat*	1	7
4	5	3	*acatat*	3	5
5	9	1	*at*	1	7
6	7	2	*atat*	5	7
7	2	0	*caaacatat*	1	11
8	6	2	*catat*	7	9
9	10	0	*t*	1	11
10	8	1	*tat*	9	11
11		-1			

Figure 4.15: The enhanced suffix array of the string $S = acaaacatat$ with the arrays $\mathtt{PSV}_{\mathsf{LCP}}$ and $\mathtt{NSV}_{\mathsf{LCP}}$.

PSV and NSV are acronyms for *previous smaller value* and *next smaller value*, respectively. Given the value $\mathsf{LCP}[i]$ at index i, among all indices j so that j is smaller than i and $\mathsf{LCP}[j]$ is smaller than $\mathsf{LCP}[i]$, $\mathtt{PSV}_{\mathsf{LCP}}[i]$ is the largest index. Analogously, among all indices j so that j is larger than i and $\mathsf{LCP}[j]$ is smaller than $\mathsf{LCP}[i]$, $\mathtt{NSV}_{\mathsf{LCP}}[i] = j$ is the smallest index. Figure 4.15 shows the arrays $\mathtt{PSV}_{\mathsf{LCP}}$ and $\mathtt{NSV}_{\mathsf{LCP}}$ of the string $S = acaaacatat$.

In this section, we will omit the subscript LCP, i.e., we will write PSV instead of $\mathtt{PSV}_{\mathsf{LCP}}$ and NSV instead of $\mathtt{NSV}_{\mathsf{LCP}}$.

Lemma 4.3.8 *Let* $2 \leq k \leq n$ *and* $\mathsf{LCP}[k] = \ell$. *Then* $[\mathtt{PSV}[k]..\mathtt{NSV}[k] - 1]$ *is an lcp-interval of lcp-value* ℓ.

Proof We have

1. $\mathsf{LCP}[\mathtt{PSV}[k]] < \ell$ (by the definition of $\mathtt{PSV}[k]$).

2. $\mathsf{LCP}[m] \geq \ell$ for all m with $\mathtt{PSV}[k] + 1 \leq m \leq \mathtt{NSV}[k] - 1$.

3. $\mathsf{LCP}[k] = \ell$ (note that $\mathtt{PSV}[k] + 1 \leq k \leq \mathtt{NSV}[k] - 1$).

4. $\mathsf{LCP}[\mathtt{NSV}[k]] < \ell$ (by the definition of $\mathtt{NSV}[k]$).

Consequently, $[\mathtt{PSV}[k]..\mathtt{NSV}[k] - 1]$ is an ℓ-interval. □

The following lemma explains how the parent interval $parent([i..j])$ of an lcp-interval $[i..j] \neq [1..n]$ can be determined with the help of the arrays LCP, PSV, and NSV.

Lemma 4.3.9 *Let* $[i..j] \neq [1..n]$ *be an lcp-interval* ($[i..j]$ *may be a singleton interval) with* $\mathsf{LCP}[i] = p$ *and* $\mathsf{LCP}[j+1] = q$.

- *If* $p = q$, *then*
  - *the parent interval of* $[i..j]$ *is the lcp-interval* $[\mathtt{PSV}[i]..\mathtt{NSV}[i]-1] = [\mathtt{PSV}[j+1]..\mathtt{NSV}[j+1]-1]$,
  - *the parent interval of* $[i..j]$ *has lcp-value* $p = q$,
  - i *and* $j+1$ *are consecutive* p*-indices of the parent interval of* $[i..j]$.
- *If* $p > q$, *then*
  - *the parent interval of* $[i..j]$ *is the lcp-interval* $[\mathtt{PSV}[i]..j]$,
  - *the parent interval of* $[i..j]$ *has lcp-value* p,
  - i *is the last* p*-index of the parent interval of* $[i..j]$.
- *If* $p < q$, *then*
  - *the parent interval of* $[i..j]$ *is the lcp-interval* $[i..\mathtt{NSV}[j+1]-1]$.
  - *the parent interval of* $[i..j]$ *has lcp-value* q,
  - $j+1$ *is the first* q*-index of the parent interval of* $[i..j]$.

Proof We proceed by case analysis.
Case $p = q$: According to Lemma 4.3.8, $[\mathtt{PSV}[i]..\mathtt{NSV}[i]-1]$ is an lcp-interval of lcp-value $p = q$. Clearly, i and $j+1$ are p-indices of that interval because $\mathsf{LCP}[i] = p$ and $\mathsf{LCP}[j+1] = p$. We claim that $\mathtt{PSV}[i] = \mathtt{PSV}[j+1]$ and $\mathtt{NSV}[i] = \mathtt{NSV}[j+1]$. This is certainly true if $[i..j]$ is a singleton interval. If $[i..j]$ is an lcp-interval of lcp-value ℓ, then $\mathsf{LCP}[m] \geq \ell$ for all m with $i+1 \leq m \leq j$ and $\ell > p = q$ prove the claim. Let $i_1 < i_2 < \cdots < i_k$ be the p-indices of the p-interval $[\mathtt{PSV}[i]..\mathtt{NSV}[i]-1]$ in ascending order. Since i and $j+1$ are two consecutive p-indices, it follows that $i = i_r$ and $j+1 = i_{r+1}$ for some $1 \leq r < k$. By Lemma 4.3.5, $[i..j]$ is a child interval of the p-interval $[\mathtt{PSV}[i]..\mathtt{NSV}[i]-1]$.
Case $p > q$: Again, by Lemma 4.3.8, $[\mathtt{PSV}[i]..\mathtt{NSV}[i]-1]$ is an lcp-interval of lcp-value p. Obviously, i is a p-index of that interval, but $j+1$ is not. Because $q < p$, we have $\mathtt{NSV}[i] = j+1$. Moreover, this implies that i is the last p-index of the p-interval $[\mathtt{PSV}[i]..j]$. According to Lemma 4.3.5, $[i..j]$ is the last child interval of $[\mathtt{PSV}[i]..j]$.
Case $p < q$: Similar to the previous case. □

As an example, consider Figure 4.15 and determine $parent([6..6])$. Since $p = \mathsf{LCP}[6] = 2 > 0 = \mathsf{LCP}[7] = q$, the second case of Lemma 4.3.9 applies, so that $parent([6..6]) = [\mathtt{PSV}[6]..6] = [5..6]$. Furthermore, the lcp-interval $[5..6]$ has lcp-value 2, and 6 is the last (in fact, the only) 2-index of $[5..6]$. As another example, we search for parent interval of $[1..2]$. In this case $p =$

$\mathsf{LCP}[1] = -1 < 1 = \mathsf{LCP}[3] = q$, so that $parent([1..2]) = [1..\mathtt{NSV}[2+1]-1] = [1..6]$. Furthermore, the parent interval of $[1..2]$ has lcp-value 1 and 3 is its first 1-index.

Corollary 4.3.10 *Let $[i..j] \neq [1..n]$ be an lcp-interval ($[i..j]$ may be a singleton interval). The parent interval of $[i..j]$ has lcp-value* $\max\{\mathsf{LCP}[i], \mathsf{LCP}[j+1]\}$.

Proof This is a direct consequence of Lemma 4.3.9. □

We have seen that child intervals can be determined with RMQs, while parent intervals can be determined with PSV and NSV values. As a matter of fact, it is also possible to determine the LCA of two lcp-intervals by means of RMQ, PSV, and NSV. However, this is left as an exercise for the reader because lowest common ancestors are not needed in the applications dealt with in this book.

Exercise 4.3.11 Give an algorithm in pseudo-code that takes two lcp-intervals $[i..j]$ and $[p..q]$ as input and returns their lowest common ancestor in the lcp-interval tree.
Hint: If $j < p$, then their LCA is the lcp-interval $[\mathtt{PSV}[k]..\mathtt{NSV}[k]-1]$, where $k = \mathsf{RMQ}(j+1, p)$.

In this chapter, we merely use the arrays PSV and NSV in proofs but not in algorithms. Nevertheless, we show here how to compute them in linear time. In the pseudo-code of Algorithm 4.5, the elements on the stack are pairs $\langle idx, lcp\rangle$, where $lcp = \mathsf{LCP}[idx]$. The procedures *push* (pushes an element onto the stack) and *pop*() (pops an element from the stack and returns that element) are the usual stack operations, while *top*() provides a pointer to the topmost element of the stack. Moreover, *top*().*idx* denotes the first component of the topmost element of the stack, while *top*().*lcp* denotes the second component.

Initially, Algorithm 4.5 pushes the pair $\langle 1, -1\rangle$ consisting of the first index and its lcp-value onto the stack, and sets $\mathtt{PSV}[1]$ to $\perp$ (so $\mathtt{PSV}[1]$ does not exist). The following invariant is maintained in the for-loop of the algorithm: for every element e on the stack, $\mathtt{PSV}[e.idx]$ is set correctly. In the while-loop, the algorithm tests whether the lcp-value of the current index k is strictly smaller than the lcp-value of the topmost element of the stack. If this is the case, then the next smaller lcp-value of the topmost element can be found at the current index k. Consequently, the assignment $\mathtt{NSV}[pop().idx] \leftarrow k$ pops the topmost element from the stack and sets the corresponding NSV-entry to k. After the while-loop, one has $\mathsf{LCP}[k] \geq top().lcp$. If the lcp-value of the topmost element of the stack is strictly smaller than that of the current index k, then the previous smaller lcp-value of the current index k is the index of the topmost element. Hence

Algorithm 4.5 Construction of the PSV and NSV arrays.

```
$push(\langle 1, -1 \rangle)$        /* an element on the stack has the form $\langle idx, lcp \rangle$ */
PSV[1] $\leftarrow \perp$
**for** $k \leftarrow 2$ **to** $n+1$ **do**
    **while** LCP[$k$] $<$ $top().lcp$ **do**
        NSV[$pop().idx$] $\leftarrow k$
    **if** LCP[$k$] $>$ $top().lcp$ **then**
        PSV[$k$] $\leftarrow top().idx$
    **else**
        PSV[$k$] $\leftarrow$ PSV[$top().idx$]
    $push(\langle k, \text{LCP}[k] \rangle)$
```

the assignment PSV[k] $\leftarrow$ $top().idx$ does the job. Otherwise, the equality LCP[k] $=$ $top().lcp$ holds. In this case, the indices k and $top().idx$ have the same previous smaller lcp-value. By the loop-invariant, PSV[$top().idx$] has been set correctly in a previous iteration of the for-loop. Therefore, PSV[k] $\leftarrow$ PSV[$top().idx$] assigns the correct value to PSV[k]. Finally, the pair $\langle k, \text{LCP}[k] \rangle$ is pushed onto the stack. Because PSV[k] was set correctly, the loop-invariant also holds before the next execution of the for-loop.

4.3.2 Bottom-up traversal

In this section, we are going to describe a linear-time algorithm that traverses the lcp-interval tree in a bottom-up fashion with the help of a stack. We shall satisfy ourselves with the lcp-interval tree *without* singleton intervals. However, it is not difficult to modify the algorithm so that it also incorporates singleton intervals. To demonstrate the full capabilities of the method, we first show that the lcp-interval tree can be constructed in a bottom-up fashion. However, in applications we will not construct this tree *explicitly*. As we shall see, it is possible to traverse this tree without constructing it.

Pseudo-code for the bottom-up construction of the lcp-interval tree can be found in Algorithm 4.6. The elements on the stack are lcp-intervals represented by quadruples $\langle lcp, lb, rb, childList \rangle$, where lcp is the lcp-value of the interval, lb is its left boundary, rb is its right boundary, and $childList$ is a list of its child intervals. Furthermore, $add(list, c)$ appends the element c to the list $list$ and returns the result. Algorithm 4.6 traverses the lcp-interval tree by scanning the LCP-array from left to right (or, in many illustrations, from top to bottom). At each index k, the while-loop tests whether lcp-intervals on the stack end with the right boundary $k-1$, and new lcp-intervals are detected in the penultimate if-statement.

Algorithm 4.6 Bottom-up traversal of the lcp-interval tree based on the LCP-array.

```
$lastInterval \leftarrow \bot$
$push(\langle 0, 1, \bot, [\,] \rangle)$
**for** $k \leftarrow 2$ **to** $n + 1$ **do**
  $lb \leftarrow k - 1$
  **while** $\mathsf{LCP}[k] < top().lcp$ **do**
    $top().rb \leftarrow k - 1$
    $lastInterval \leftarrow pop()$
    $process(lastInterval)$
    $lb \leftarrow lastInterval.lb$
    **if** $\mathsf{LCP}[k] \leq top().lcp$ **then**
      $top().childList \leftarrow add(top().childList, lastInterval)$
      $lastInterval \leftarrow \bot$
  **if** $\mathsf{LCP}[k] > top().lcp$ **then**
    **if** $lastInterval \neq \bot$ **then**
      $push(\langle \mathsf{LCP}[k], lb, \bot, [lastInterval] \rangle)$
      $lastInterval \leftarrow \bot$
    **else** $push(\langle \mathsf{LCP}[k], lb, \bot, [\,] \rangle)$
```

As an example, consider the execution of Algorithm 4.6 applied to the LCP-array of the string $S = ctaataatg$, shown in Figure 4.16. First, the 0-interval $[1..\bot]$ is pushed onto the stack. In the first iteration ($k = 2$) of the for-loop, the next lcp-interval 3-$[1..\bot]$ is detected because $\mathsf{LCP}[2] = 3 > 0 = top().lcp$. Consequently, it is pushed onto the stack; see Figure 4.17. In the next iteration ($k = 3$) the while-loop detects the end of this 3-interval because $\mathsf{LCP}[3] = 1 < 3 = \mathsf{LCP}[2]$. Thus, its right boundary $rb = k - 1 = 2$ is set, it is popped from the stack, and processed. Then, the if-statement inside the while-loop tests by $\mathsf{LCP}[k] \leq top().lcp$ whether this 3-interval is a child of the lcp-interval 0-$[1..\bot]$, which now lies on top of the stack. If so, it would be added to the child list of the topmost interval. Since $\mathsf{LCP}[3] = 1 \not\leq 0 = top().lcp$, however, this is not the case. Thereafter, the while-loop is left and the lcp-interval 1-$[1..\bot]$ is detected and pushed onto the stack. Because it is the parent interval of the "dangling" 3-interval, its child list must contain the interval 3-$[1..2]$. The remaining part of the LCP-array is processed analogously.

i	1	2	3	4	5	6	7	8	9	10
$\mathsf{LCP}[i]$	-1	3	1	2	0	0	0	4	1	-1

Figure 4.16: The LCP-array of the string $S = ctaataatg$; cf. Figure 4.13.

k	contents of the stack
	$\langle 0, 1, \perp, [\,]\rangle$
2	$\langle 3, 1, \perp, [\,]\rangle$
	$\langle 0, 1, \perp, [\,]\rangle$
3	$\langle 1, 1, \perp, [\langle 3, 1, 2, [\,]\rangle]\rangle$
	$\langle 0, 1, \perp, [\,]\rangle$
4	$\langle 2, 3, \perp, [\,]\rangle$
	$\langle 1, 1, \perp, [\langle 3, 1, 2, [\,]\rangle]\rangle$
	$\langle 0, 1, \perp, [\,]\rangle$
5	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle]\rangle$
6	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle]\rangle$
7	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle]\rangle$
8	$\langle 4, 7, \perp, [\,]\rangle$
	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle]\rangle$
9	$\langle 1, 7, \perp, [\langle 4, 7, 8, [\,]\rangle]\rangle$
	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle]\rangle$
10	

Figure 4.17: Contents of the stack during the run of Algorithm 4.6. $\langle 0, 1, 9, [\langle 1, 1, 4, [\langle 3, 1, 2, [\,]\rangle, \langle 2, 3, 4, [\,]\rangle]\rangle, \langle 1, 7, 9, [\langle 4, 7, 8, [\,]\rangle]\rangle]\rangle$ is the last interval that is processed (when $k = 10$). As a matter of fact, it is the whole lcp-interval tree corresponding to the LCP-array of Figure 4.16. The construction of the lcp-interval tree can be avoided by implementing the procedure *process* in Algorithm 4.6 accordingly: after *process* has processed $lastInterval$ (the parameter of the procedure), the child list of $lastInterval$ must be emptied of its contents by the assignment $lastInterval.childList \leftarrow [\,]$; cf. Algorithm 4.7.

The correctness of Algorithm 4.6 is a direct consequence of Theorem 4.3.12.

Theorem 4.3.12 *Consider the for-loop of Algorithm 4.6 for some index k. Let top be the topmost interval on the stack and top_{-1} be the interval directly beneath it (note that $top_{-1}.lcp < top.lcp$). If $\mathsf{LCP}[k] < top.lcp$, then before top will be popped from the stack in the while-loop, the following holds:*

1. *If $\mathsf{LCP}[k] \leq top_{-1}.lcp$, then top is the child interval of top_{-1}.*
2. *If $\mathsf{LCP}[k] > top_{-1}.lcp$, then top is the first child interval of the lcp-interval with lcp-value $\mathsf{LCP}[k]$ that contains k. To be precise, top is the first child interval of $[top.lb..\mathtt{NSV}[k] - 1]$.*

Proof (1) First, we show that top is embedded in top_{-1}. The following invariant is maintained in the for-loop of Algorithm 4.6: If $\langle \ell_1, lb_1, rb_1\rangle, \ldots, \langle \ell_m, lb_m, rb_m\rangle$ are the intervals on the stack, where $top = \langle \ell_m, lb_m, rb_m\rangle$, then $lb_i \leq lb_j$ and $\ell_i < \ell_j$ for all $1 \leq i < j \leq m$. Furthermore, because $\langle \ell_j, lb_j, rb_j\rangle$ will be popped from the stack before $\langle \ell_i, lb_i, rb_i\rangle$, it follows that $rb_j \leq rb_i$. Thus, the ℓ_j-interval $[lb_j..rb_j]$ is embedded in the ℓ_i-interval $[lb_i..rb_i]$. In particular, top is embedded in top_{-1}.

If top was not the child interval of top_{-1}, then there would be an lcp-interval $\langle lcp', lb', rb'\rangle$ so that top is embedded in $\langle lcp', lb', rb'\rangle$ and $\langle lcp', lb', rb'\rangle$ is embedded in top_{-1}. This, however, can only happen if $\langle lcp', lb', rb'\rangle$ is an interval on the stack that is above top_{-1}. This contradiction proves the claim.

(2) We have $\mathsf{LCP}[top.lb] = top_{-1}.lcp < \mathsf{LCP}[k] < top.lcp$ and $top.rb = k-1$. By the third case of Lemma 4.3.9, it follows that (a) the parent interval of top is the lcp-interval $[top.lb..\mathtt{NSV}[k]-1]$, (b) the parent interval of top has lcp-value $\ell = \mathsf{LCP}[k]$, and (c) k is the first ℓ-index of the parent interval of top. Thus, the lemma follows. □

In Algorithm 4.6, the lcp-interval tree is traversed in a bottom-up fashion by a linear scan of the LCP-array, while storing information on a stack. Whenever an ℓ-interval is processed by the generic procedure $process$, only its child intervals have to be known. These are determined solely from the lcp-information, i.e., we do not need explicit parent-child pointers in our framework. It should be stressed that the algorithm exhibits strong locality of reference because of the sequential access to the LCP-array.

It is possible to solve several problems merely by specifying the procedure $process$ in Algorithm 4.6; an example is given below. Other applications may require slight modifications of the algorithm; see Chapter 5.

Let us address the problem of finding all substrings of S having at least p and at most q occurrences in S, where $1 \leq p \leq q$. The goal is to give a linear-time algorithm that solves the problem. However, if $p = 1$ and

Algorithm 4.7 To find all substrings of S having at least p and at most q occurrences in S, where $2 \leq p \leq q$, plug this implementation of the procedure *process* in Algorithm 4.6.

```
$process(lastInterval)$
  for each $\langle \ell, i, j, [\,] \rangle$ in $lastInterval.childList$ do
    if $p \leq (j-i+1)$ and $(j-i+1) \leq q$ then
      output $(lastInterval.lcp, \ell, [i..j])$
  $lastInterval.childList \leftarrow [\,]$        /* empty childList */
```

$q = n$, then the algorithm must output all substrings of S, and there are $O(n^2)$ substrings of S. In other words, a linear-time algorithm is impossible if every substring is output explicitly. For this reason, the algorithm must use an implicit representation of the output. Here, we will give a solution for the case $p \geq 2$. Exercise 4.3.15 asks you to solve the problem for the case $p = 1$. As in Algorithm 4.6, the lcp-interval tree of S is traversed in a bottom-up fashion. Suppose that the lcp-interval m-$[lb..rb]$ is going to be processed by the procedure *process*. At this point, all its child intervals are known. Let ℓ-$[i..j]$ be one of those. Let the lcp-intervals m-$[lb..rb]$ and ℓ-$[i..j]$ represent the strings u and ω, respectively; see Definition 4.3.1. Clearly, $\omega = uv$ for some string v of length $\ell - m$. The key observation is that every substring uv', where v' is a non-empty prefix of v, occurs exactly $(j-i+1)$ times in S. Thus, procedure *process* tests whether $p \leq (j-i+1) \leq q$ is true. If so, it outputs $(m+1, \ell, [i..j])$; meaning that every prefix of $\omega = S[\mathsf{SA}[i]..\mathsf{SA}[i]+\ell-1]$ having a length in between $m+1$ and ℓ occurs at least p times and at most q times in S, namely at the positions $\mathsf{SA}[i], \ldots, \mathsf{SA}[j]$. Algorithm 4.7 implements this approach. Note that its last assignment $lastInterval.childList \leftarrow [\,]$ empties the *childList* of $lastInterval$. This ensures that the lcp-interval tree is not constructed during the bottom-up traversal.

Exercise 4.3.13 Show that Algorithm 4.6 takes only linear time and space.

Exercise 4.3.14 Modify Algorithm 4.6 so that it also incorporates singleton intervals.

Exercise 4.3.15 Give a linear-time solution to the problem of finding all substrings of S having at most $q \geq 1$ occurrences in S.

Exercise 4.3.16 A string ω is called a *prefix tandem repeat* of string S if ω is a prefix of S and has the form uu for some string u. Give a linear-time algorithm to find the longest prefix tandem repeat of S.

Algorithm 4.8 *BuildTopDown*($[i..j]$) recursively constructs the subtree of the lcp-interval tree rooted at the lcp-interval $[i..j]$, using the LCP-array and RMQs thereon.

```
if $i = j$ then return $\langle \bot, i, i, [\,] \rangle$          /* singleton interval */
$childList \leftarrow [\,]$
$k \leftarrow i$
$m \leftarrow \mathsf{RMQ}(i+1, j)$          /* first $\ell$-index of $[i..j]$ */
$\ell \leftarrow \mathsf{LCP}[m]$
repeat
  $subtree \leftarrow BuildTopDown([k..m-1])$
  $add(childList, subtree)$
  $k \leftarrow m$
  if $k = j$ then
    break
  else
    $m \leftarrow \mathsf{RMQ}(k+1, j)$
until $\mathsf{LCP}[m] \neq \ell$
$subtree \leftarrow BuildTopDown([k..j])$
$add(childList, subtree)$
return $\langle \ell, i, j, childList \rangle$
```

4.3.3 Top-down traversal

According to Lemma 4.3.5, determining the child intervals of an ℓ-interval $[i..j]$ boils down to finding the ℓ-indices of $[i..j]$ in ascending order. With range minimum queries (see Chapter 3) on the LCP-array this is easy: $\mathsf{RMQ}(i+1, j)$ yields the first ℓ-index i_1, $\mathsf{RMQ}(i_1+1, j)$ yields the second ℓ-index i_2, etc.

We use this to construct the lcp-interval tree from the LCP-array in a top-down fashion. The pseudo-code of the procedure *BuildTopDown* can be found in Algorithm 4.8; it takes an lcp-interval $[i..j]$ as input and returns the subtree of the lcp-interval tree rooted at node $[i..j]$. Hence *BuildTopDown*($[1..n]$) yields the desired lcp-interval tree. As in Algorithm 4.6, nodes (i.e., lcp-intervals) in the lcp-interval tree are represented by quadruples $\langle lcp, lb, rb, childList \rangle$, where lcp is the lcp-value of the interval (this value is $\bot$ in singleton intervals), lb is its left boundary, rb is its right boundary, and $childList$ is the list of its child intervals.

Let us have a closer look at Algorithm 4.8. The first line contains the base case of the recursion: If $i = j$, then $[i..j]$ is a singleton interval, and the lcp-interval tree rooted at node $[i..j]$ consists solely of the node $\langle \bot, i, i, [\,] \rangle$. Otherwise, $i < j$ and the lcp-interval $[i..j]$ is not a singleton interval. Its child list is initialized to the empty list and k is set to the left boundary of the lcp-interval $[i..j]$. Furthermore, m is set to the first lcp-index of the lcp-interval $[i..j]$ (note that every lcp-interval has at least one lcp-index and this can be obtained by the range minimum query $\mathsf{RMQ}(i+1,j)$) and ℓ is set to the lcp-value of $[i..j]$. When the repeat-until-loop is entered, the interval $[k..m-1]$ is the first child interval of $[i..j]$ by Lemma 4.3.5. Consequently, $\mathit{BuildTopDown}$ is called recursively with this child interval and it returns the subtree of the lcp-interval tree rooted at node $[k..m-1]$. This subtree is added to the child list. Thereafter, the current ℓ-index is stored in variable k. The loop will be executed as long as $k < j$ and $\mathsf{LCP}[\mathsf{RMQ}(k+1,j)] = \ell$, i.e., it will be executed as long as $[k..j]$ is not a singleton interval and there is another ℓ-index of the lcp-interval $[i..j]$, namely the index $m = \mathsf{RMQ}(k+1,j)$. In this case, $\mathit{BuildTopDown}$ is called recursively with the child interval $[k..m-1]$ (cf. Lemma 4.3.5), the returned subtree is added to the child list, and k is set to the current ℓ-index m.

After the loop is done, there are two possibilities.

- $k = j$: In this case, the recursive call $\mathit{BuildTopDown}([k..j])$ yields the subtree consisting of one node, viz. the singleton interval $[k..j]$, and this subtree is added to the child list.

- $k < j$ and $\mathsf{LCP}[\mathsf{RMQ}(k+1,j)] \neq \ell$: In this case, k is the last ℓ-index of the lcp-interval $[i..j]$ and, by Lemma 4.3.5, $[k..j]$ is the last child interval of the lcp-interval $[i..j]$. Thus, $\mathit{BuildTopDown}$ is called recursively with this child interval and the returned subtree is added to the child list.

Finally, Algorithm 4.8 returns the lcp-interval tree rooted at the lcp-interval $[i..j]$ in form of the quadruple $\langle \ell, i, j, \mathit{childList} \rangle$.

Exercise 4.3.17 Show that Algorithm 4.8 takes only linear time and space. Modify the algorithm so that

- it returns the lcp-interval tree rooted at node $[i..j]$ without singleton intervals,

- it returns the list of all child intervals of $[i..j]$ instead of the lcp-interval tree rooted at node $[i..j]$.

We stress that in applications it is not necessary to actually construct the lcp-interval tree of a string. Slight modifications to Algorithm 4.8 suffice to obtain algorithms that traverse the lcp-interval tree in a top-down fashion without constructing it. Below, we provide two applications. The first one uses a *depth-first* traversal (similar to Algorithm 4.8), while the second one uses a *breadth-first* traversal of the lcp-interval tree.

In our first application, for each non-singleton lcp-interval ℓ-$[i..j]$ we wish to compute a value $\mathit{val}([i..j])$ defined as follows: For a non-empty string ω, let $\mathit{occ}_\omega(S)$ denote the number of occurrences of ω in S. Let u

i	SA	LCP	$S_{\mathsf{SA}[i]}$	VAL
1	3	-1	$aaacatat$	
2	4	2	$aacatat$	8
3	1	1	$acaaacatat$	6
4	5	3	$acatat$	10
5	9	1	at	6
6	7	2	$atat$	8
7	2	0	$caaacatat$	0
8	6	2	$catat$	4
9	10	0	t	0
10	8	1	tat	2
11		-1		

Figure 4.18: The enhanced suffix array of $S = acaaacatat$ with VAL array.

be the string that is represented by the lcp-interval ℓ-$[i..j]$ (i.e., u is the longest common prefix of the suffixes $S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[i+1]}, \ldots, S_{\mathsf{SA}[j]}$), and define

$$val([i..j]) = \sum_{\omega \sqsubset u} occ_\omega(S)$$

where $\omega \sqsubset u$ means that ω is a non-empty prefix of u. In words, $val([i..j])$ is the number of all occurrences of all non-empty prefixes of u in S. In Section 5.7.2, the importance of these values will become clear. As an example, consider the lcp-interval 3-$[3..4]$ in Figure 4.18. This lcp-interval represents the string $u = aca$. The prefixes a, ac, and aca of u have 6, 2, and 2 occurrences in S. Hence $val([3..4]) = 10$.

Our algorithm is based on the following lemma.

Lemma 4.3.18 *Let q-$[lb..rb]$ be a child interval of the lcp-interval ℓ-$[i..j]$. Then*

$$val([lb..rb]) \quad = \quad val([i..j]) + (rb - lb + 1)\ (q - \ell).$$

Proof Let u be the string that is represented by $[i..j]$. This implies that $[lb..rb]$ represents a string uv, where $v \neq \varepsilon$. Let ω be a substring of S so that $\omega \sqsubset uv$ but $\omega \not\sqsubset u$. The key observation is that the ω-interval coincides with the uv-interval. In other words, ω occurs as often in S as uv does, namely

$(rb - lb + 1)$ times. Thus,

$$\begin{aligned} val([lb..rb]) &= \sum_{\omega \sqsubset uv} occ_\omega(S) \\ &= \sum_{\omega \sqsubset u} occ_\omega(S) + \sum_{\omega \sqsubset uv, \omega \not\sqsubset u} occ_\omega(S) \\ &= val([i..j]) + \sum_{\omega \sqsubset uv, \omega \not\sqsubset u} (rb - lb + 1) \\ &= val([i..j]) + (rb - lb + 1) \sum_{\omega \sqsubset uv, \omega \not\sqsubset u} 1 \\ &= val([i..j]) + (rb - lb + 1)\ (q - \ell) \end{aligned}$$

□

Again, consider the lcp-interval 3-[3..4] and its parent interval 1-[1..6]; see Figure 4.18. We have $val([1..6]) = 6$ because a occurs six times in S. According to the previous lemma,

$$val([3..4]) = val([1..6]) + (4 - 3 + 1)\ (3 - 1) = 6 + 2 \cdot 2 = 10$$

To have constant-time access to the values, we store them in an additional array VAL. For an lcp-interval ℓ-$[i..j]$, the value $val([i..j])$ can be stored at several locations. Among the options are

1. the first ℓ-index of $[i..j]$,
2. all ℓ-indices of $[i..j]$,
3. the home index of $[i..j]$, defined by

$$home([i..j]) = \begin{cases} i & \text{if } \mathsf{LCP}[i] \geq \mathsf{LCP}[j+1] \\ j & \text{otherwise} \end{cases}$$

In what follows, we will use the second possibility; see Figure 4.18 for an example. The uniqueness of the alternative location $home([i..j])$ is due to Strothmann [302]; cf. Exercise 4.3.19.

The procedure $ValTopDown(\ell\text{-}[i..j], idx, val)$ of Algorithm 4.9 takes an lcp-interval $[i..j]$ of lcp-value ℓ, its first ℓ-index idx, and $val = val([i..j])$ as input and recursively computes the VAL array of the lcp-interval tree rooted at $[i..j]$. The lcp-value ℓ and the first lcp-index idx of the lcp-interval $[i..j]$ are supplied as parameters to the procedure because this avoids superfluous recomputations of these values. In order to get the whole VAL array, the procedure is called with the root interval 0-$[1..n]$, its first 0-index $\mathsf{RMQ}(2, n)$ and $val = 0$. In Algorithm 4.9, the value $val([lb..rb])$ of a child interval q-$[lb..rb]$ of ℓ-$[i..j]$ is computed by a generic function $computeValue$. Here,

Algorithm 4.9 $\mathit{ValTopDown}(\ell\text{-}[i..j], idx, val)$ recursively computes the VAL array of the lcp-interval tree rooted at the lcp-interval ℓ-$[i..j]$, where idx is the first ℓ-index of $[i..j]$ and $val = val([i..j])$. It uses the LCP-array and RMQs thereon.

```
$k \leftarrow i$          /* $k$ stores the left boundary of the current child interval */
$m \leftarrow idx$          /* $m$ stores the current $\ell$-index */
repeat
  $\mathsf{VAL}[m] \leftarrow val$
  if $k \neq m-1$ then          /* $[k..m-1]$ is a non-singleton child of $[i..j]$ */
    $childIdx \leftarrow \mathsf{RMQ}(k+1, m-1)$          /* first lcp-index of $[k..m-1]$ */
    $q \leftarrow \mathsf{LCP}[childIdx]$          /* $q$ is the lcp-value of $[k..m-1]$ */
    $childVal \leftarrow computeValue(\ell, val, q, k, m-1)$
    $ValTopDown(q\text{-}[k..m-1], childIdx, childVal)$
  $k \leftarrow m$          /* $k$ is left boundary of the next child interval */
  if $k = j$ then
    return          /* there is no more non-singleton child interval */
  else
    $m \leftarrow \mathsf{RMQ}(k+1, j)$          /* $m$ is the next $\ell$-index unless $\mathsf{LCP}[m] \neq \ell$ */
until $\mathsf{LCP}[m] \neq \ell$
/* $[k..j]$ is the last non-singleton child interval of $[i..j]$ */
/* and $m$ is the first lcp-index of $[k..j]$ */
$q \leftarrow \mathsf{LCP}[m]$          /* $q$ is the lcp-value of $[k..j]$ */
$childVal \leftarrow computeValue(\ell, val, q, k, j)$
$ValTopDown(q\text{-}[k..j], m, childVal)$
```

$computeValue(\ell, val, q, lb, rb) = val + (rb - lb + 1)\ (q - \ell)$ by Lemma 4.3.18. We shall see in Exercise 4.3.20 and in Section 5.7.2 that it is sufficient to modify *computeValue* to solve related problems.

We will briefly explain Algorithm 4.9. As we have seen in Lemma 4.3.5, the child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_k..j]$, where $i_1 < i_2 < \cdots < i_k$ are the ℓ-indices of $[i..j]$. In Algorithm 4.9, the variables k and m store the left boundary of the current child interval and the current ℓ-index, respectively. Initially, k is set to the left boundary of the interval $[i..j]$ and m is set to the first ℓ-index idx of $[i..j]$. Hence the first child interval is $[k..m - 1]$. The body of the repeat-until-loop stores val in the VAL array at the current ℓ-index m and then deals with the current child interval $[k..m - 1]$ provided it is a non-singleton. In this case, the first lcp-index $childIdx$ of $[k..m - 1]$ is determined by the range minimum query $\mathsf{RMQ}(k + 1, m - 1)$. Therefore, $q = \mathsf{LCP}[childIdx]$ is the lcp-value of $[k..m - 1]$. According to Lemma 4.3.18, $childVal = val([k..m - 1])$ is computed by $computeValue(\ell, val, q, lb, rb) = val + (rb - lb + 1)\ (q - \ell)$. The computation proceeds recursively with the procedure call $ValTopDown(q\text{-}[k..m - 1], childIdx, childVal)$. Subsequently, k becomes the left boundary of the next child interval, which is m, and the next ℓ-index of $[i..j]$ must be determined. If $k = j$, then certainly there is no more ℓ-index and the last child interval of $[i..j]$ is the singleton interval $[j..j]$. In this case, the repeat-until-loop is left and the procedure terminates. Otherwise, m is set to $\mathsf{RMQ}(k + 1, j)$. Now there are two possibilities: Either $\mathsf{LCP}[m] = \ell$, in which case m is the next ℓ-index of $[i..j]$, or $\mathsf{LCP}[m] \neq \ell$, in which case $[k..j]$ is the last (non-singleton) child interval of $[i..j]$ and m is the first lcp-index of $[k..j]$. In the first case, the loop is repeated, i.e., the next iteration of the loop sets $\mathsf{VAL}[m]$ to val and deals with the next child interval $[k..m - 1]$. In the second case, Algorithm 4.9 deals with the last child interval $[k..j]$ of $[i..j]$ as with the previous child intervals. Figure 4.18 depicts the VAL array of our example.

Exercise 4.3.19 For an lcp-interval $[i..j]$, define

$$home([i..j]) = \begin{cases} i & \text{if } \mathsf{LCP}[i] \geq \mathsf{LCP}[j + 1] \\ j & \text{otherwise} \end{cases}$$

to be the *home index* of $[i..j]$. Prove that for any two lcp-intervals $[i..j]$ and $[p..q]$, the equality $home([i..j]) = home([p..q])$ implies $[i..j] = [p..q]$.

Exercise 4.3.20 Modify Lemma 4.3.18 and the function *computeValue* in such a way that Algorithm 4.9 computes

$$val([lb..rb]) = \sum_{\omega \sqsubset uv} |\omega| \cdot occ_{\omega}(S)$$

where uv is the string represented by the lcp-interval $[lb..rb]$.

As a second application of the top-down traversal, we will briefly describe how to find all shortest unique substrings. This is relevant in the design of primers for DNA sequences; for details see Section 5.6.5.

Definition 4.3.21 A substring $S[i..j]$ of S is *unique* if it occurs exactly once in S. The *shortest unique substring problem* is to find all shortest unique substrings of S.

For example, ca is the shortest unique substring of $acac$. If S consists solely of a's, i.e., $S = a^n$, then the shortest unique substring is S itself. In the following, we assume that S contains at least two different characters.

It is readily verified that a substring u of S is unique if and only if it is prefix of exactly one suffix of S, say of $S_{\mathsf{SA}[k]}$. In terms of lcp-intervals, this can be rephrased as: u is unique if and only if there is exactly one suffix $S_{\mathsf{SA}[k]}$ so that u is prefix of $S_{\mathsf{SA}[k]}$ but not of $S[\mathsf{SA}[k]..\mathsf{SA}[k]+\ell-1]$, where ℓ-$[i..j]$ is the parent interval of the singleton interval $[k..k]$. Moreover, the length of all the shortest unique substrings of S is $m+1$, where m is the smallest lcp-value of all lcp-intervals having a singleton child interval. Using this observation, the shortest unique substring problem can be solved by a breadth-first traversal of the lcp-interval tree, using a queue. Initially, the queue contains only the root interval 0-$[1..n]$. During the traversal, more lcp-intervals may be added to the queue. Besides the queue, the algorithm maintains a set M that contains all the shortest unique substrings detected so far (represented by their start position in S) and a variable min that stores their length. Initially, M is empty and $min = \infty$.

Suppose that ℓ-$[i..j]$ is removed from the front of the queue, i.e., it is the lcp-interval that is processed next during the breadth-first traversal. The algorithm computes all its child intervals. If a singleton child interval $[k..k]$ of $[i..j]$ is detected, then the length $\ell+1$ prefix of $S_{\mathsf{SA}[k]}$ is a unique substring of S. Thus, if M is empty or $min > \ell+1$, then M is set to $\{\mathsf{SA}[k]\}$ and min is set to $\ell+1$. If M is not empty and $min = \ell+1$, then $\mathsf{SA}[k]$ is added to M. Otherwise, M and min remain unchanged. If ℓ-$[i..j]$ has no singleton child interval, then every child interval q-$[lb..rb]$ of ℓ-$[i..j]$ satisfying $q+1 \leq min$ is added to the back of the queue. Then, the algorithm proceeds with the next lcp-interval at the front of the queue, as described above, until the queue is empty. Finally, the algorithm outputs min and M. It is not difficult to see that the algorithm takes time proportional to the number of processed lcp-intervals. In the worst case, this is $O(n)$. However, in practice only a small number of lcp-intervals is processed.

Exercise 4.3.22 Given a string S on an alphabet Σ, a string $\omega \in \Sigma^+$ is called an *absent word* if it is not a substring of S. Develop an algorithm that takes a string S as input, and outputs all shortest absent words (suppose that the alphabet Σ consists of the characters appearing in S).