Bibliographic study

Lyndon words, permutations and trees

Author Robert Fisch

Date 11.05.2004

Table of contents

Table of figures

1. Reading map

1.1. Authors

- Christophe Hohlweg [hohlweg@math.u-strasbg.fr]
- Christophe Reutenauer [christo@math.uquam.ca]

1.2. Reference

Lyndon words, permutations and trees

Theoretical Computer Science
Volume 307, Issue 1 (September 2003)
WORDS
Pages: 173 - 178
Year of Publication: 2003
ISSN: 0304-3975

Elsevier Science Publishers Ltd.

URL: http://www-irma.u-strasbg.fr/irma/publications/2002/02017.shtml
URL: http://portal.acm.org/citation.cfm?id=953045

1.3. Keywords

Lyndon, words, permutation, tree, suffix, binary, standardization, factorization, standard

1.4. Abstract

In the here presented article, its author sets up an algorithm for doing suffix standardization of a given Lyndon word in $O(n \cdot \log(n))$ time, using a binary, planary complete tree. After this he shows how the latter can be used to factorize any word into a decreasing product of Lyndon words.

1.5. Addendum

In the second part of this document I will give some reflections and personal remarks about factorization of Lyndon words.

2. Presentations

2.1. Projection

For better understanding of what follows, knowledge about projections of binary, planary trees is very important. For this reason the author gives the following example:

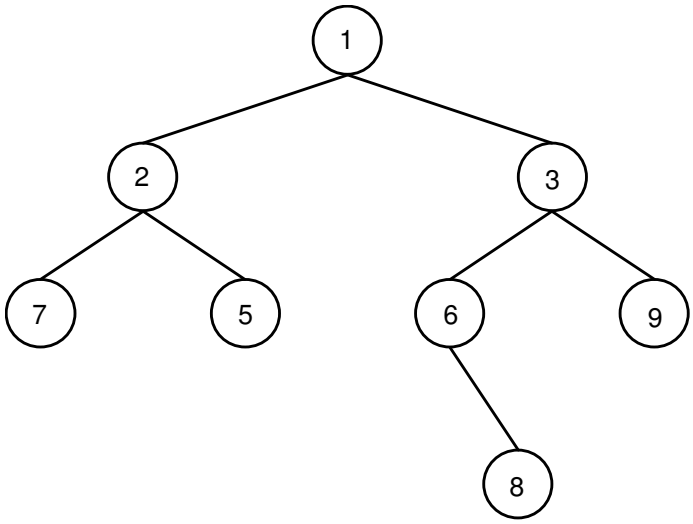


Figure 1: Binary, planary tree

The projection of this tree leads to the following sequence: `72516839`

In order to obtain this sequence, the "left - root - right" rule is applied recursively to each node, starting at the root node. This is equal to an infix-lecture of the tree. It may be represented by the following scheme:

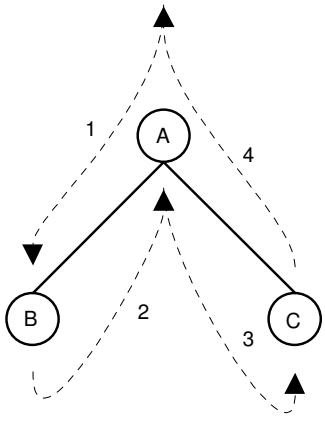


Figure 2: The «left - root - right» rule

The different steps are:

1. start at node A, go to node B and apply the entire process to the tree located at node B
2. read the content of node B, then go back to node A
3. read the content of node A, move down to node C and apply the entire process to the tree located at node C
4. read the content of C and move back to node A

2.2. Suffix standardization

Take a given word l and write down all its possible right factors into a list $[r_1..r_n]$, where $n = |l|$. Next this list has to be ordered in the lexicographically decreasing order, meaning for any element r_i of the list, except the last one, we have $r_i < r_{i+1}$. In fact r_1 is the smallest and $r_{|l|}$ the biggest right factor of l.

This list is what the author defines as being the suffix standard permutation σ. $\sigma(i)$ is the starting position of the i^{th} element of the list in the word l. σ^{-1} denotes the inverse function.

Let us now consider the following sequence: `babbacaca`

The ordered list of its right factor is the following:

$i = \sigma^{-1}(s)$ = list index	right factor	$s = \sigma(i)$ = starting position
1	a	9
2	abbacaca	1
3	aca	7
4	acaca	5
5	babbacaca	2
6	bacaca	4
7	bbacaca	3
8	ca	8
9	caca	6

Notice that for any Lyndon word, $\sigma(1) = 1$, because, by definition, a Lyndon word is the lexicographically smallest word among all its right factors.

The example sequence given by the author is: `aabaacab`

This is indeed a Lyndon word, so we will see in the list beneath that $\sigma(1) = 1$.

$i = \sigma^{-1}(s)$ = list index	right factor	$s = \sigma(i)$ = starting position
1	aabaacab	1
2	aacab	4
3	ab	7
4	abaacab	2
5	acab	5
6	b	8
7	baacab	3
8	cab	6

2.3. Infinite extension

Any Lyndon word l can be spitted into $l = p_i \cdot s_i$ so that s_i is a right factor of l and p_i is its corresponding left factor. Having to indexes i and j it can be proven that: $i < j \Rightarrow (s_i \cdot p_i)^\infty < (s_j \cdot p_j)^\infty$

Proof
As a matter of fact, for any given index x:

$$l = p_x \cdot s_x \Rightarrow (s_x \cdot p_x)^\infty = s_x \cdot (p_x \cdot s_x)^\infty \cdot p_x = s_x \cdot l^\infty \cdot p_x$$

This means that it will be enough to show that: $s_i < s_j \Rightarrow (s_i \cdot p_i)^\infty < (s_j \cdot p_j)^\infty$

If s_i is not a left factor of s_j, $s_i \notin P(s_j)$, the above statement is true. In the opposite case, having $s_i \in P(s_j)$, s_j can be written using s_i as follows: $s_j = s_i \cdot x$

As s_i already is a right factor of l, and x being a right factor of s_i, x is also a right factor of l. Having l being a Lyndon word, it is obvious that $l < x$.

Hence we can write:

$$(s_i \cdot p_j)^\infty < (s_j \cdot p_j)^\infty$$
$$\Leftrightarrow s_i \cdot l^\infty \cdot p_i < s_j \cdot l^\infty \cdot p_j$$
$$\Leftrightarrow s_i \cdot l^\infty \cdot p_i < s_i \cdot x \cdot l^\infty \cdot p_j$$
$$\Leftrightarrow l^\infty \cdot p_i < x \cdot l^\infty \cdot p_j$$

This is true because $l < x$.

qed

Example

For the string $l = aabaacab$, let us consider the right factors $s_1 = abaacab$ and $s_2 = acab$.

Obviously we have: $abaacab < acab$ and $s_1 \notin P(s_2)$

Thus also: $abaacab \cdot a < acab \cdot aaba$

because for any $u, v, x, y \in A^*$, we will have that: $\left.\begin{array}{c} u \le v \\ u \notin P(v) \end{array}\right\} \Rightarrow u \cdot x \le v \cdot y$

This gives us then:
$$(abaacab \cdot a)^\infty < (acab \cdot aaba)^\infty$$
$$\Leftrightarrow abaacab \cdot (a \cdot abaacab)^\infty \cdot a < acab \cdot (aaba \cdot acab)^\infty \cdot aaba$$
$$\Leftrightarrow abaacab \cdot l^\infty \cdot a < acab \cdot l^\infty \cdot aaba$$

2.4. Lyndon tree

This section describes what the author calls "The Lyndon Tree". Based upon a word l, a binary, planary complete tree can be build up, which, for the last given example, will look like this:

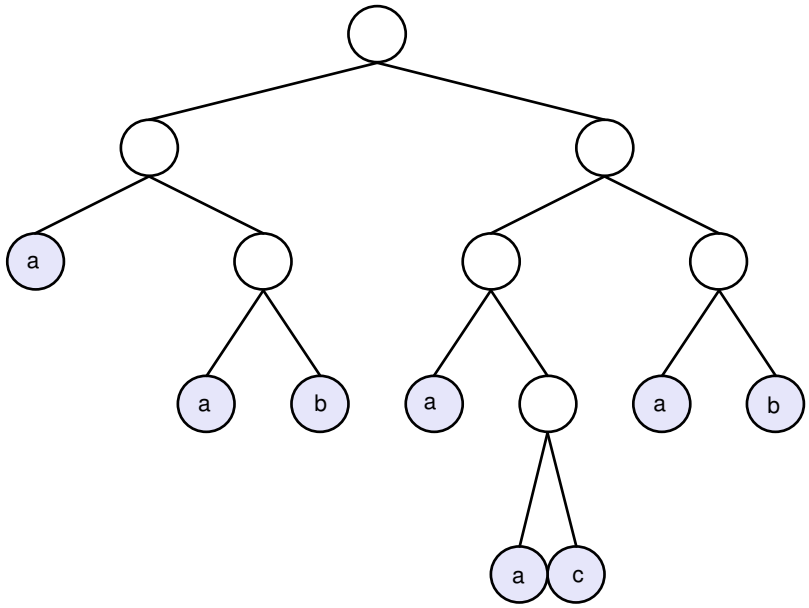


Figure 3: The "aabaacab" tree

2.5. Tree analyses

As said by the author of the article, for any Lyndon word l, which can be decomposed to $l = l' \cdot l''$, where l'' is the smallest proper right factor of l, l' and l'' are Lyndon words too.

If $T(l)$ denotes the Lyndon tree of the Lydon word l and having $l = l' \cdot l''$, then $T(l) = \left(T(l'), T(l'')\right)$.

This can be easy verified for the example string given above:

$$
\begin{aligned}
& aabaacab \\
&= aab \cdot aacab \\
&= (a \cdot ab) \cdot (aac \cdot ab) \\
&= \left[a \cdot (a \cdot b)\right] \cdot \left[(a \cdot ac) \cdot ab\right] \\
&= \left[a \cdot (a \cdot b)\right] \cdot \left\{\left[a \cdot (a \cdot c)\right] \cdot (a \cdot b)\right\}
\end{aligned}
$$

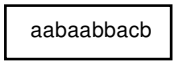
Each division has to be made such that the right factor of the divided word is its smallest proper right factor. As a matter of fact the last line represents well the infix notation of the tree in [Figure 3].

2.6. Offline tree construction

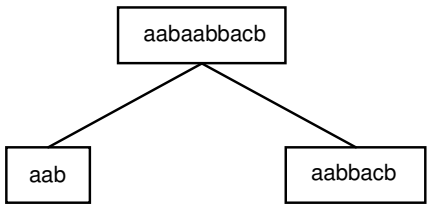
Using the above property, it is easy to set up an offline tree construction algorithm, meaning that the entire word has to be known to the algorithm before starting it.

Let us take again the string: `aabaabbacb`

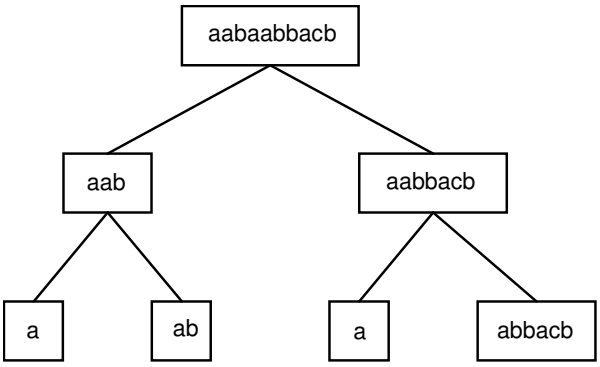
At the beginning there is only the root node, containing the entire string.



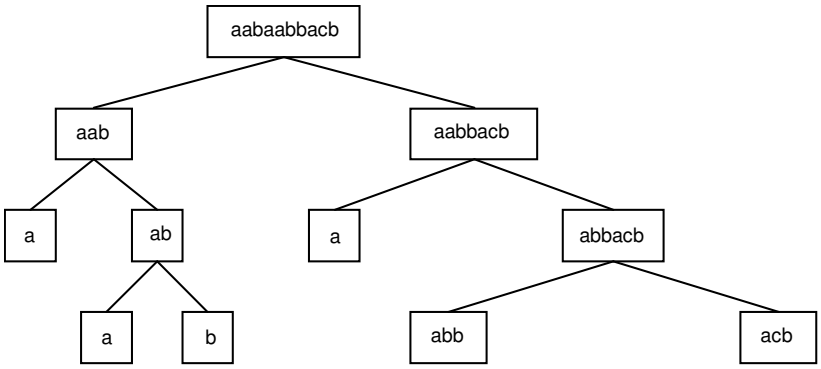
The above property is applied in order to split it into two new Lyndon words, namely: `aab + aabbacb`



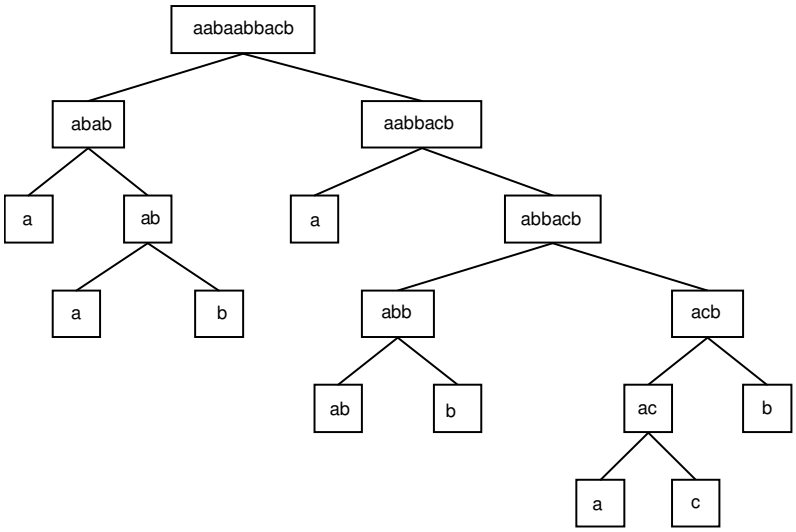
Next the procedure is applied to all leaves.



If any leaf contains a single character, there is no need to apply the procedure on it.



In this step the left part of the tree is completely done. Only the right part has to be considered now.



Finally the following tree is obtained, which is identical to the one obtained by the online construction [Figure 6]:

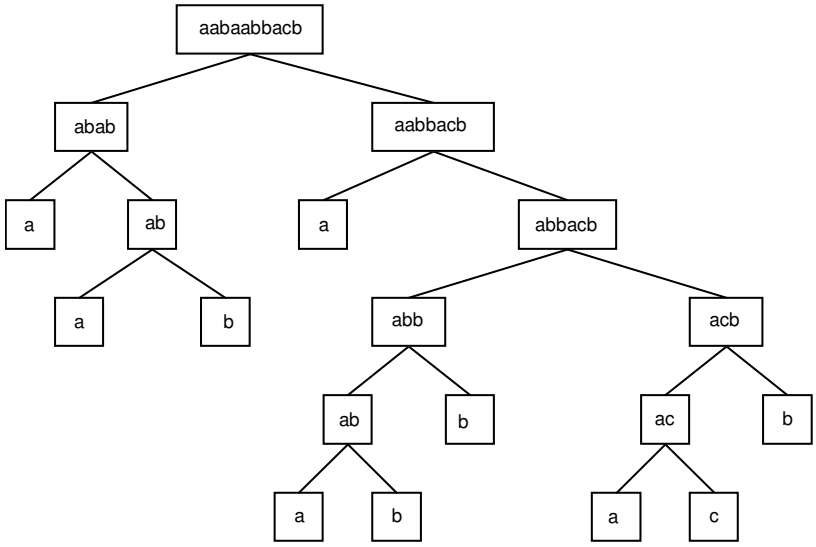


Figure 4: Offline «aabaabbacb» tree construction

2.7. Reading a Lyndon tree

A Lyndon tree, like the one in [Figure 3] or [Figure 4], can be red by using the projection described in section [2.1.]. Depending on the starting position, it is possible to determine de different right factors of the word.

On the figure below, starting the infix-lecture at the third leaf will output the right factor: `baabbacb`

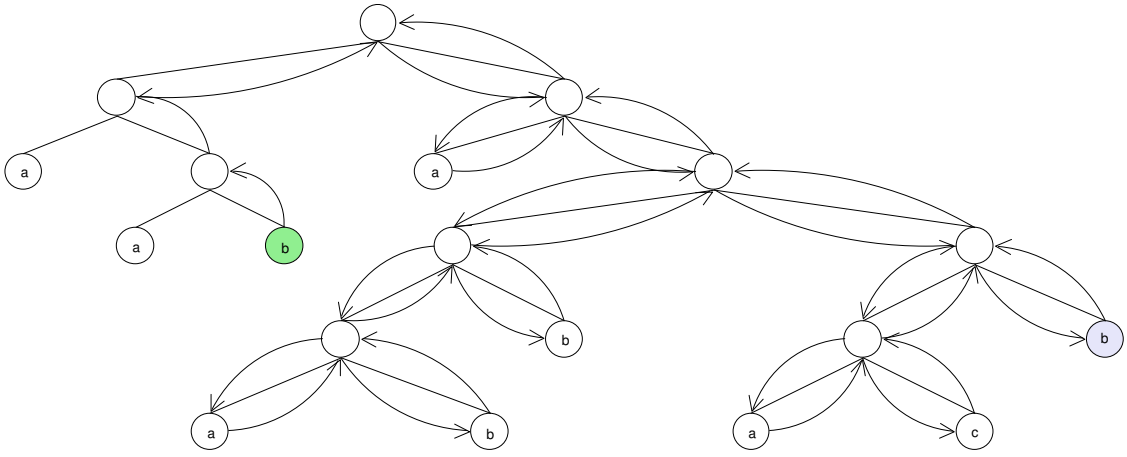


Figure 5: Reading a right factor

Instead of starting the lecture at a given leaf, we may also start it at a precise internal node, beginning by reading its right descent. By doing so, we can affect a distinct right factor to each internal node:

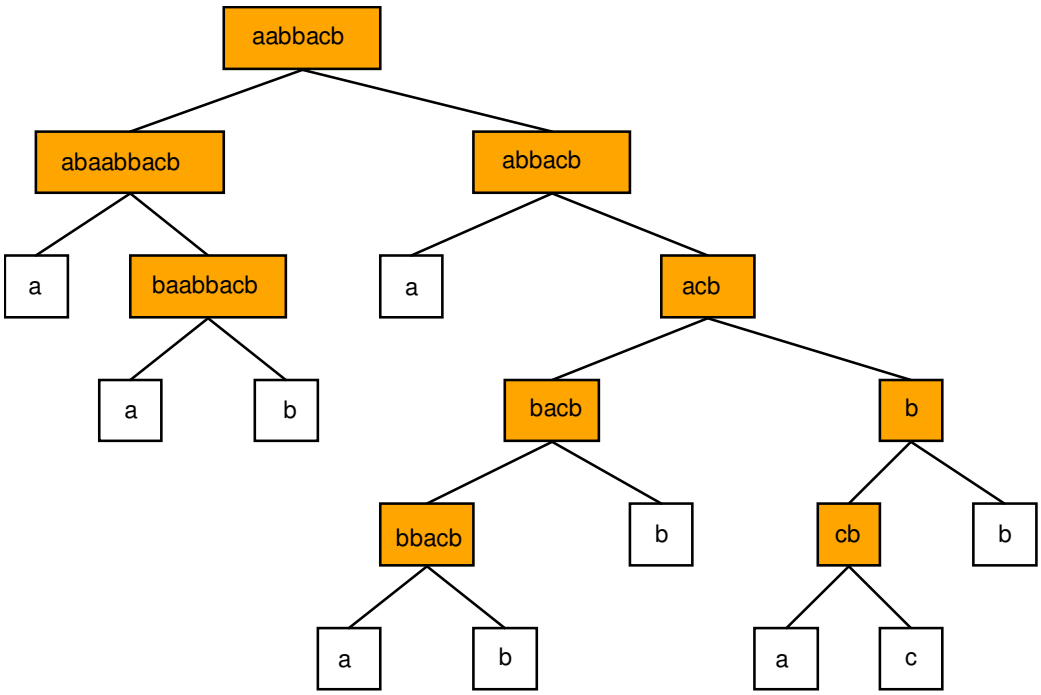


Figure 6: Reading right factors

By cutting out all leaves, we will obtain the increased binary Lyndon tree, which is no longer complete.

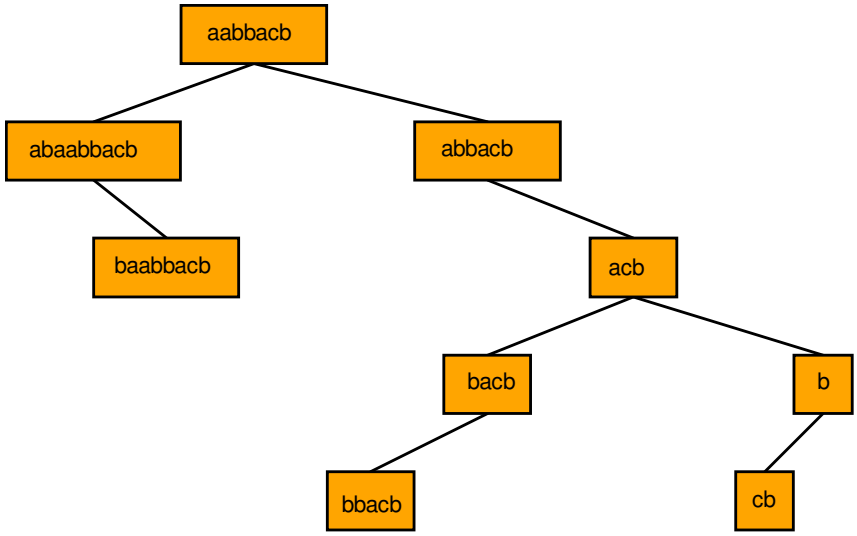


Figure 7: Increased Lyndon tree

Notice that, when reading this tree using an infix-lecture and starting at the root node, we are obtaining the list of the proper right factors of the word:

```
abaabbacb
baabbacb
aabbacb
abbacb
bbacb
bacb
acb
cb
b
```

Another property to underline is that the content of any given node is lexicographically smaller than any of its descents.

At this point we may now also replace the nodes' content with its relative starting position. These are the numeric values of the example table of the suffix standardization [2.2.].

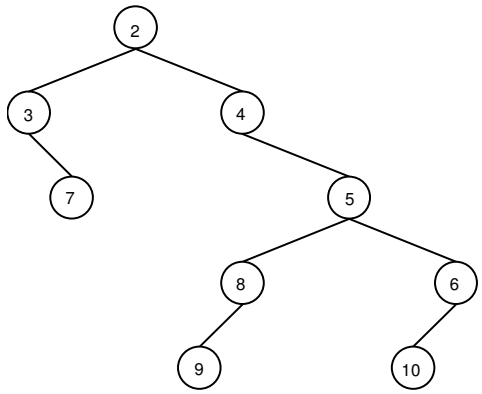


Figure 8: Increased position Lyndon tree

2.8. Factorizing any word

The same procedure can be applied to any given word. For example the sequence: `dabodabdab`

Its Lyndon tree will look like this:

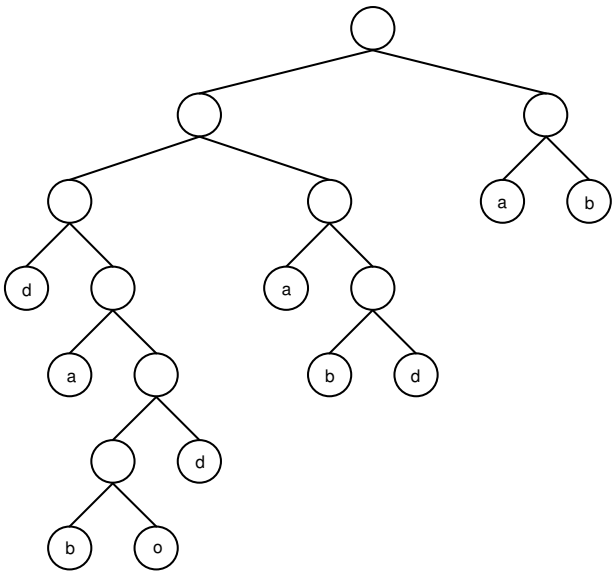


Figure 9: Lyndon tree for «dabodabdab»

By remodeling the given tree a bit, as may be seen on the next figure, the following things can easily be noticed:

- On the way from the root to the leftmost node, each right sub-tree builds some kind of cluster.
- The projection of each of these sub-trees gives a Lyndon word.
- Reading the sub-trees associated projections from the topmost sub-tree to the one at the bottom, the different words appear in decreasing order.

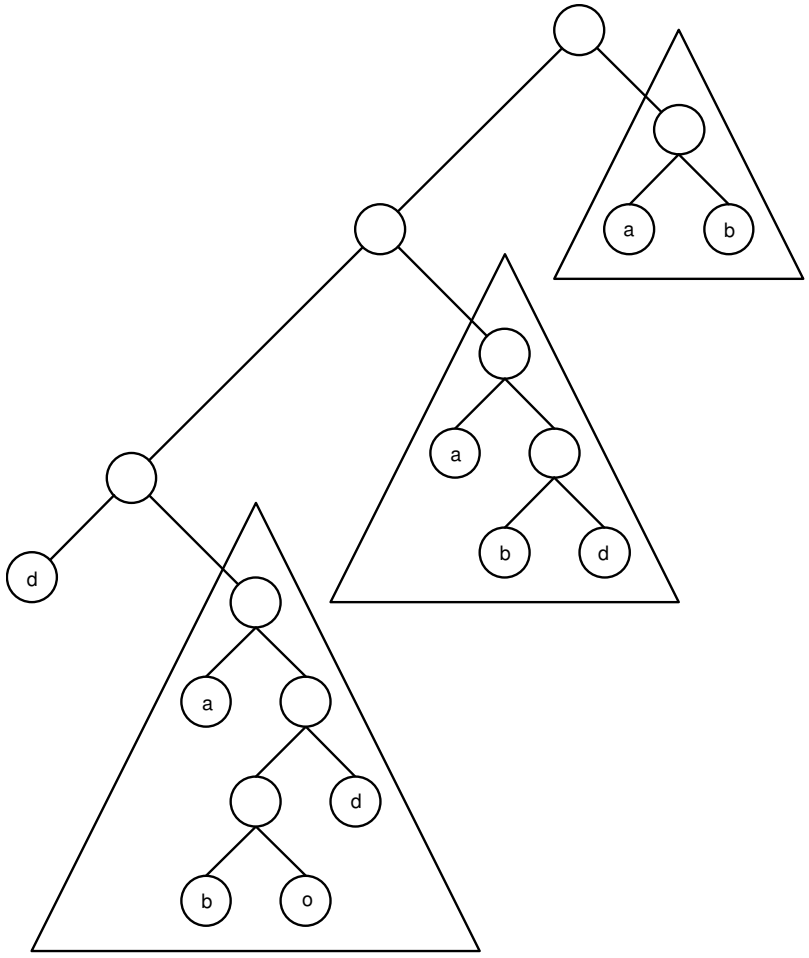


Figure 10: Lyndon tree factorization

3. Addendum

While working with Lyndon words and reading some articles, I noticed some interesting properties about Lyndon words:

3.1. Right extension

Taking $u,v,x,y \in A^*$, we will obtain that: $\left.\begin{array}{l} u \le v \\ u \notin P(v) \end{array}\right\} \Rightarrow u \cdot x \le v \cdot y$

This is true, especially for the case where $u = a \in A$, $v = b^k \mid b \in A$ $(k > 0)$ and $y = a \cdot x$. Thus we will get: $\left.\begin{array}{l} a \le b^k \\ a \ne b \end{array}\right\} \Leftrightarrow a \cdot x \le b^k \cdot a \cdot x$

3.2. Definition of a crescent word

A word $\omega = x_1 \cdots x_{|\omega|}$ is *crescent* if either $\forall 1 \le i \le |\omega| - 1 \rightarrow x_i \le x_{i+1}$ or $|\omega| = 1$.

Each *crescent word* is necessarily a Lyndon word too because a *crescent word* is trivially the smallest among its right factors.

3.3. Definition of a strictly crescent word

A word $\omega = x_1 \cdots x_{|\omega|}$ is *strictly crescent* if $|\omega| = 1$ or if $|\omega| > 1$ such that ω is *crescent* and $\exists k \mid x_k < x_{k+1}$, having $1 \le k \le |\omega| - 1$

3.4. Decomposition into crescent words

Any word $\omega \in A^+$ can be decomposed into *crescent words*.

Proof

Let be $\omega \in A^+$, then ω is composed of $|\omega|$ elements of A. Hence each $a \in A$ is a *crescent word* (by definition), thus ω can be decomposed into *crescent words*.

qed

3.5. Definition of a maximal crescent word

A factor f of any given word ω is defined as being *maximal crescent*, if it can't be extended neither at left nor at right such that it stays *crescent*.

Example
If $\omega = abcaabbcacbd$ then $f = aabbc$ is a *maximal crescent factor*, but $f' = aab$ is not.

3.6. Decomposition into maximal crescent words

Each word $\omega \in A^+$ can be decomposed into *maximal crescent factors*.

Proof
Let $\omega \in A^+$ be a word. If ω is already a *crescent word*, then the above proposition is true.

If ω is not a *crescent word*, then we can define: $K = \left\{2 \le i \le |\omega| \,\middle|\, \omega_{i-1} > \omega_i\right\} \neq \phi$

Putting $m = \min(K)$, we have that ω can be written as $\omega = c_1 \cdot r_1$, such that c_1 is a *maximal crescent factor.* This gives us: $c_1 = \omega_1 \cdot\cdot \omega_{m-1}$ and $r_1 = \omega_m \cdot\cdot \omega_{|\omega|}$.

By recursion on $|K|$, each word can be decomposed into *maximal crescent factors*.

qed

3.7. Unique decomposition into maximal crescent words

The above mentioned decomposition of a word $\omega \in A^+$ into *maximal crescent factors* is unique.

Proof
Suppose that it is not unique and that ω could be written under the form $\omega = c_1 \cdot\cdot c_n = c_1' \cdot\cdot c_k'$, having $k \neq n$.

Without loss of generality, we can put $c_1 \ge c_1' \Rightarrow c_1' \notin P(c_1)$.

This means that the *maximal crescent factor* c_1' is a prefix of the *maximal crescent factor* c_1, but having $c_1 > c_1'$, c_1' can be extended in order to obtain

a longer *crescent factor*. This is contradictory to the fact that $c_1^{'}$ is a *maximal crescent factor* of ω.

By induction on the indexes of c, we will get $k = n$ and $\forall 1 \le i \le n \rightarrow c_i = c_i^{'}$

qed

3.8. Notice about decomposition of a Lyndon word #1

Let $\omega \in \mathcal{L}$ be a *Lyndon* word and $c_1 \cdot \cdot c_n$ its unique decomposition into *maximal crescent factors*. In this case we will have: $\forall 2 \le i \le n \rightarrow c_1 \le c_i$

Proof

Suppose that $\exists i | c_1 \ge c_i$. Then, for the given i, we will have $c_1 \notin P(c_i)$, because $c_1 \ge c_i$. Using [Property 4.1.], we will obtain: $c_1 \cdot x \ge c_i \cdot y$, especially for $x = \alpha \cdot c_i \cdot \beta | c_1 \cdot x = \omega$ and $y = \beta$.

Thus: $c_1 \cdot \alpha \cdot c_i \cdot \beta \ge c_i \cdot \beta \Leftrightarrow \omega \ge s_j$

Hence $\omega \in \mathcal{L}$, implying $\forall 1 \le i \le |\omega| - 1 \rightarrow \omega < s_i$, which is contradictory to the hypotheses made above, thus the proposition is true.

qed

3.9. Notice about decomposition of a Lyndon word #2

Let p_i be a prefix of length i of a *Lyndon* word $\omega \in \mathcal{L}$ and let m be the length of the longest *crescent factor* of ω.

Then $\exists k | \forall k \le j \le m \rightarrow p_j$ is *strictly crescent.* We can also say that the first *crescent factor* of any *Lyndon* word is necessarily *strictly crescent.*

Proof

Let p_i be the left factor of length i of ω and s_j its right factor of length j.

As $\omega \in \mathcal{L}$ is a *Lyndon* word, by definition, ω is strictly smaller than any of its proper right factors s_i, having $\forall 1 \le i \le |\omega| - 1$. We may put $\omega = x_1 .. x_{|\omega|}$.

Considering the word $p_1 = x_1$, if $|\omega| = 1$, then p_1 is a *Lyndon* word and p_1 is *strictly crescent* by definition. If $|\omega| > 1$, we can suppose that for a given $i \geq 1$ we will have $x_1 = x_2 = \ldots = x_i$ and thus $p_i = x_1 \cdot x_2 \cdot \ldots \cdot x_i$.

Different cases for $p_{i+1} = p_i + x_{i+1}$ must be analyzed:

- $i+1 = |\omega|$

  - if $x_{i+1} < x_i$, then ω is not a *Lyndon* word, because $x_{i+1} = s_1 < \omega$!

  - if $x_{i+1} = x_i$, then ω is not a *Lyndon* word, because $x_{i+1} = s_1 < \omega$!

  - if $x_{i+1} > x_i$, then ω is a *Lyndon* word and $\omega = p_{i+1}$ is *strictly crescent* because $x_{i+1} > x_i$.

- $i+1 < |\omega|$

  - $x_{i+1} < x_i$, then ω is not a *Lyndon* word, because, having $p_i = x_1 \cdot x_2 \cdot \ldots \cdot x_i$ with $x_1 = x_2 = \ldots = x_i$, [Property 4.1.] can be applied to obtain:

$$\left.\begin{array}{l} x_{i+1} < p_i \\ x_{i+1} \neq x_i \end{array}\right\} \Rightarrow x_{i+1} \cdot s_{|\omega|-i-1} = s_{|w|-i} < p_i \cdot x_{i+1} \cdot s_{|\omega|-i-1} = \omega$$

  - $x_{i+1} = x_i$, in this case no conclusion is possible until the next step's character is known.

  - $x_{i+1} > x_i$, then ω_{i+1} is a *strictly crescent* factor.

Thus, by recursion on i, there will always be a i such that ω_i will be *strictly crescent*.

qed

3.10. Online tree construction

The above properties can be used to set up an algorithm for building a binary complete tree which's leafs are labeled by the letters of the word. This tree will have special characteristics. The algorithm I will present is an online algorithm, meaning that at each step a character is being added and that the entire word has not to be known in advance.

The rules to build up the tree are the followings:

1. If the actual node's content is less or equal than the character that should be added, then two new nodes have to be added to the current one. The left child will get the old content of the current node and the right one will contain the newly inserted character.

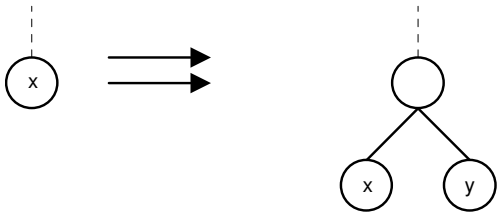


Figure 11: Node splitting transformation

2. If the actual node's content is greater than the character to be added, then a new node is being added on top of the root which's left descent will be the old tree and which's right child is becoming the node with the new character to add.

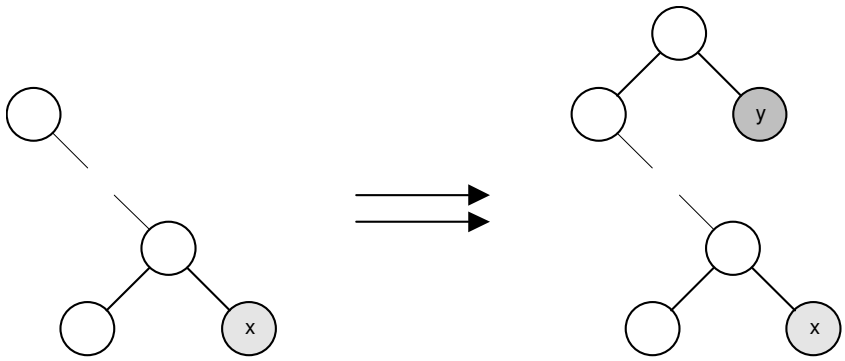
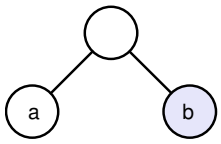


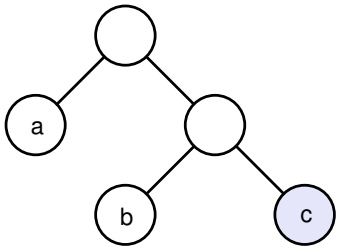
Figure 12: Root extension

For the example string $\omega = abcaabbca$ the different steps will be:

A simple "a" node:

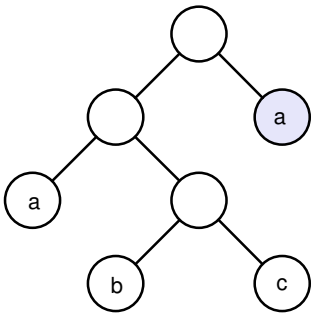
Adding a "b" by splitting the node:



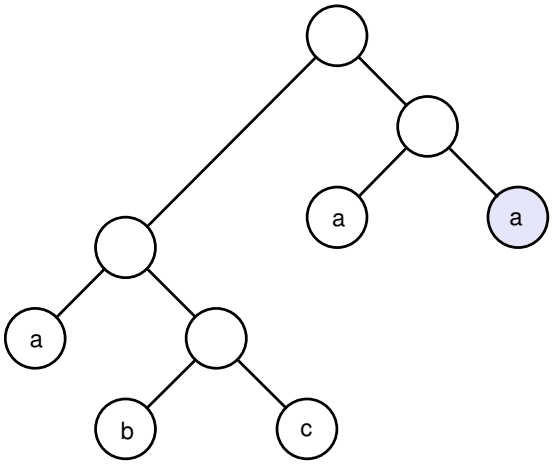


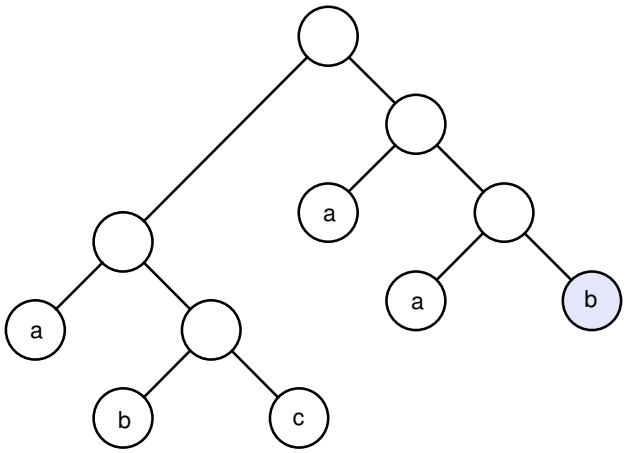
Next a "c" is being added, using the same rule:

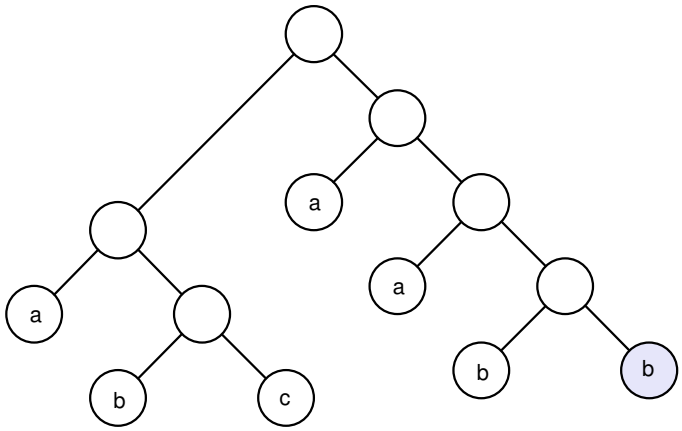
To add the next "a" will, the second rule is required:



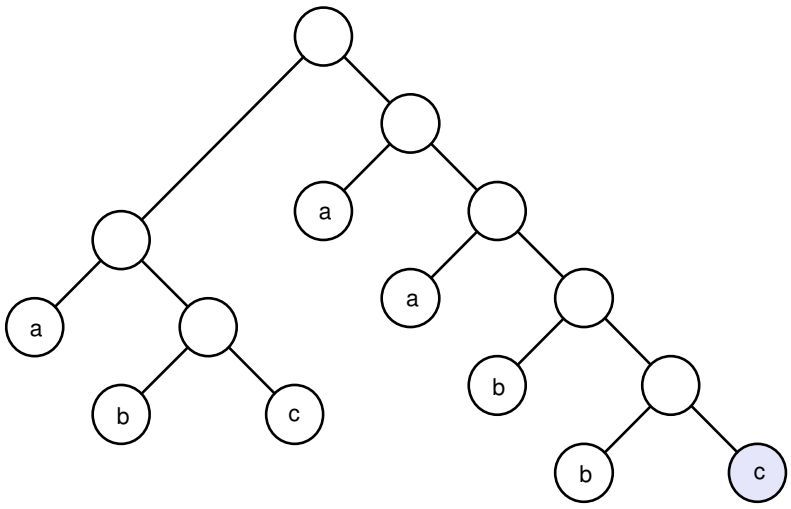
The next few letters, namely "aabbc" will be added using the first rule:







After having added the "c", the tree will look like this:



Finally we have to add the last "a" so that the final tree will become:

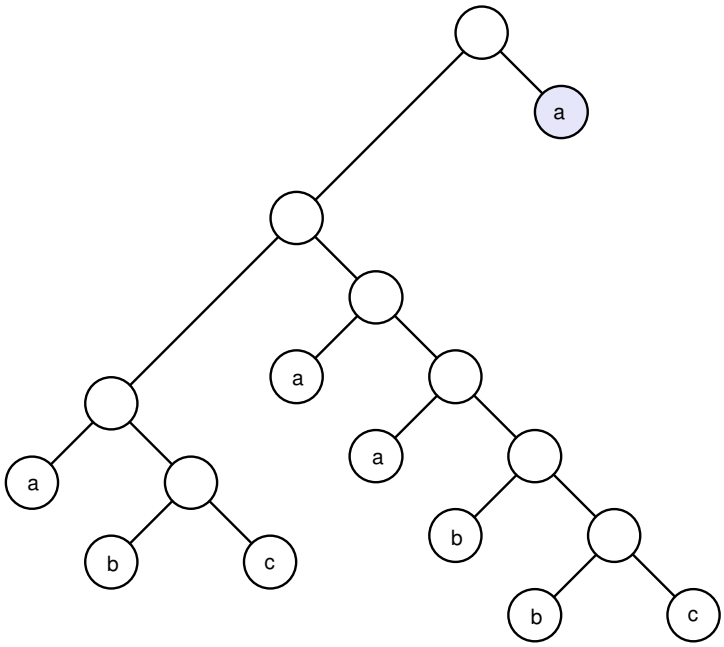


Figure 13: Crescent word factorization tree

By remodeling the previous tree, one can easily see that it can be split into different sub-trees, where each sub-tree represents a maximal crescent factor of the starting string.

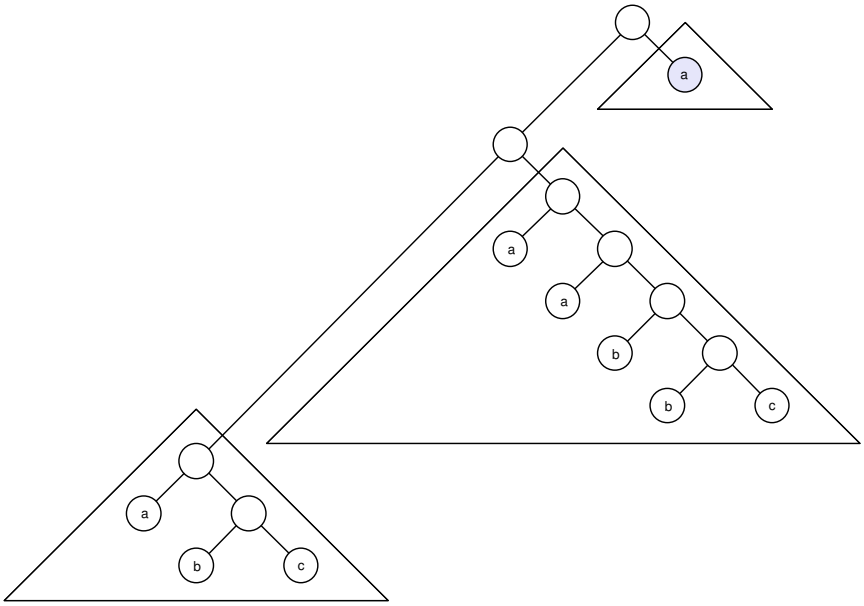


Figure 14: Tree decomposition

By holding at each step a reference to the root node of the tree as well as to the last added node, it seems evident that such a tree can be build up in linear time $O(n)$.

This algorithm is equal to a simple loop algorithm which steps through the word and at each time it locates a character which is lexicographically smaller than the previous one, it stores the memorized word to a list and reinitialized it with the newly found letter. When the found character is greater or equal than the last one, it is just being added to the memorized string.

The pseudo code for such an algorithm will look like this:

```
function splitCrescent(_w:String)
var
 list: Array of String;
 tmp: String;
 i: integer;

begin

 tmp:=_w[1]; // init vars

 for i = 2 to length(_w) do // stop through the string
 begin
  if(_w[i]<_w[i-1]) then
  begin
   list.add(tmp);
   tmp:=_w[i];
  end
  else tmp:=tmp+_w[i];
 end;
 list.add(tmp);

 result:=list;
end;
```

4. Bibliography

[LOT97] LOTHAIRE M.: *Combinatorics on Words*, Encyclopedia of Mathematics, edition 1983, Vol. 17, Addison-Wesley, reprinted in 1997 by Cambridge University Press, with only minor corrections, in the Cambridge Mathematical Library, [onlie] http://www-igm.univ-mlv.fr/~berstel/Lothaire/

[LOT02] LOTHAIRE M.: *Algebraic Combinatorics on Words*, Cambridge University Press, May 2002, ISBN: 0521812208, [online] http://www-igm.univ-mlv.fr/~berstel/Lothaire/