Computing quasi suffix arrays

L.Baghdadi,F.Franek,W.F.Smyth, and X.Xiao

Algorithms Research Group
McMaster University

Motivation: in order to identify all substrings in a given string (needed for instance for data compression) or identifying all repeats (in pattern matching), or fast search for substrings, we have to "know" the structure of the string. Several data structures have been developed to allow sophisticated algorithms (usually of complexity ($O(n\ log\ n)$ or $O(n)$) to do so.

1968 *Morrison* introduced *Patricia trie.*

1972 *Weiner* introduced suffix trees, based on Patricia trie.

Suffix trees can be constructed effectively in $O(n \log n)$ time, e.g. 1992 *Ukkonen*. But both, the construction and the tree require a lot of space.

1997 *Farach* $\Theta(n)$ construction of suffix tree, with very significant increase in space complexity.

The space requirements both, for processing and the data structure, are so high that this approach becomes impractical for large strings

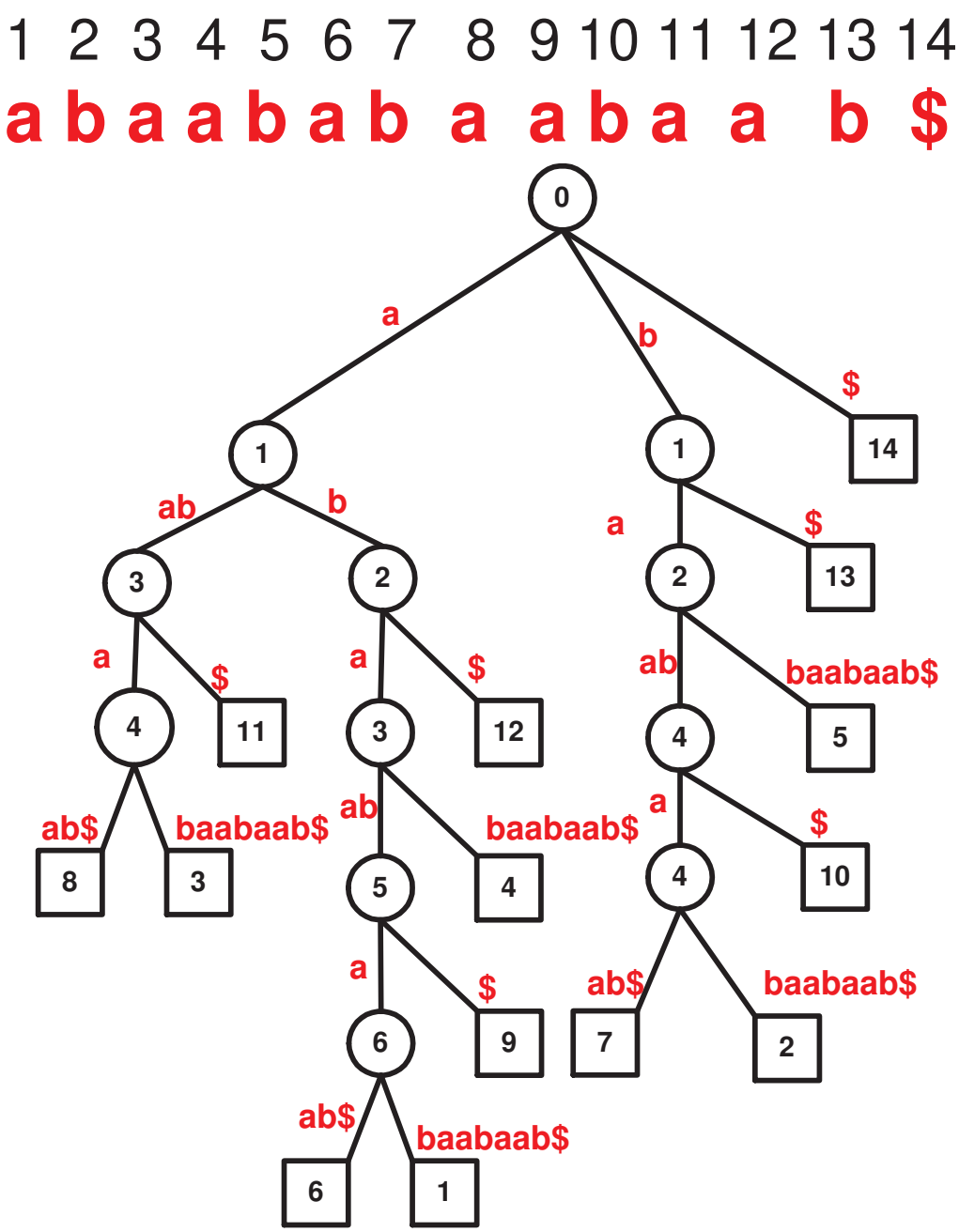
of tens of million or more characters.

1993 *Manber, Myers* introduced *suffix arrays* and presented an $O(n\ log\ n)$ algorithm to compute suffix array. Suffix array requires $2n$ integers for storage, and the algorithm itself requires $3n$ integers of additional memory to execute. Suffix array can be transformed to the corresponding suffix tree and vice versa in $\Theta(n)$ time.

We present something similar to suffix array, that's why we call it *quasi suffix array*. It carries sufficient structural information about the string, allows a construction of

the corresponding quasi suffix tree in $O(n\ log\ n)$ (but most likely in $O(n)$) time and thus may be used in problems that do not require full strength of suffix array or suffix tree (most of pattern matching problems qualify). The advantage may be the speed with which quasi suffix array can be computed compared to computation of suffix array.

We will discuss several simple, fast, and space efficient algorithms to compute quasi suffix arrays.



A string and its suffix tree.

The corresponding suffix array:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$\pi =$	$\cdot$	4	3	1	6	5	3	2	0	5	4	2	1
$\lambda =$	8	3	11	6	1	9	4	12	7	2	10	5	13

The λ array is the array of lexicographically sorted suffixes, where $\lambda[i] = j$ represents the suffix of x from the position j, i.e. $x[j..13]$.

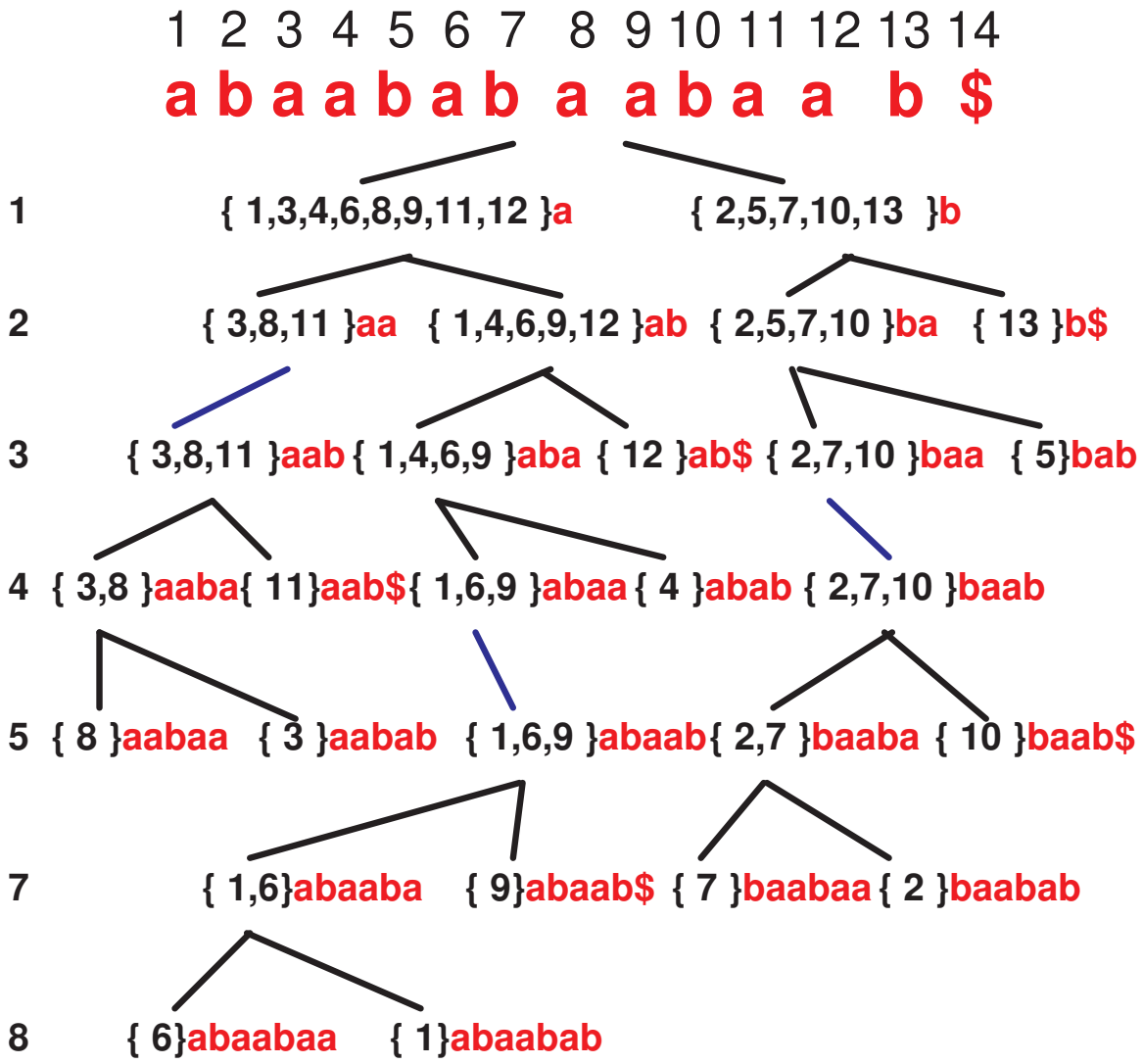
The π array is the *lcp* array, i.e. $\pi[i] =$ length of $lcp(\lambda[i-1], \lambda[i])$.

Note that the λ array must be lexicographically sorted; to produce an array of suffixes is trivial, $\lambda[i] = i$ will do.

Note that depth-first traversal of the square nodes of the suffix tree produces the λ array, while the traversal of the round nodes produces the π array. In reverse we can use the two arrays and construct the suffix tree.

Even though not originally intended to be so, *Crochemore's* repetitions $O(n\ log\ n)$ algorithm (1981) can be used to construct suffix tree or suffix array.

The algorithms to compute quasi suffix array are based on the main ideas of Crochemore's algorithm concerning refinement of certain classes of equivalence.


1 2 3 4 5 6 7 8 9 10 11 12 13 14
a b a a b a b a a b a a b $
1 { 1,3,4,6,8,9,11,12 }a { 2,5,7,10,13 }b
2 { 3,8,11 }aa { 1,4,6,9,12 }ab { 2,5,7,10 }ba { 13 }b$
3 { 3,8,11 }aab { 1,4,6,9 }aba { 12 }ab$ { 2,7,10 }baa { 5}bab
4 { 3,8 }aaba { 11}aab$ { 1,6,9 }abaa { 4 }abab { 2,7,10 }baab
5 { 8 }aabaa { 3 }aabab { 1,6,9 }abaab { 2,7 }baaba { 10 }baab$
7 { 1,6}abaaba { 9}abaab$ { 7 }baabaa { 2 }baabab
8 { 6}abaabaa { 1}abaabab

To obtain $O(n\ log\ n)$ complexity, the refinement of the classes is done via so-called "small" classes.

The problem is that Crochemore's algorithm requires extensive data structures to handle the classes and their refinement, generally of the order of $20n$ integers. With some effort and some penalty in processing speed (about 25-30%), it can be implemented using "only" $10n$ (as we did).

We shall first discuss the naive algorithm DIST1 to introduce the main ideas and to introduce the quasi suffix array.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$c_1 =$	0	0	1	3	2	4	5	6	8	7	9	11	10
$c_2 =$			0	1	2	4	5	3	6	7	8	9	0
$c_3 =$				1	0	4	2	3	6	7	8	0	
$c_4 =$				0		1	2	3	6	7	0		
$c_5 =$						1	2	0	6	0			
$c_6 =$						1	0		0				
$c_7 =$						0							

Setting c_1 is the same as in Crochemore's algorithm. Its complexity depends on the alphabet.

We again build two arrays, $\hat{\pi}$ and $\hat{\lambda}$ as for suffix arrays.

$\hat{\pi}[i] =$ the first level p at which $c_p[i]$ becomes zero.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\hat{\pi} =$	0	0	1	3	2	6	5	4	5	4	3	2	1

$\hat{\lambda}[i] = c_{p-1}[i]$ when $c_p[i]$ becomes zero.

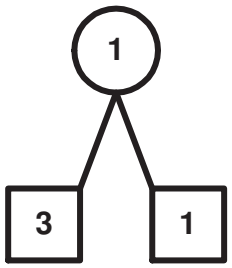
	1	2	3	4	5	6	7	8	9	10	11	12	13
$\hat{\lambda} =$	0	0	1	1	2	1	2	3	6	7	8	9	10

Thus
$\hat{\pi}[i] =$ the length of $lcp(i, \hat{\lambda}[i])$.

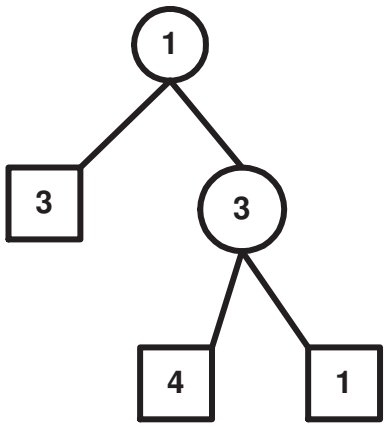
Compare the quasi suffix array with suffix array. First we note that $\hat{\pi}$ is a permutation of π, i.e. we get the same sizes of lcp's. But do we get the same substrings of x as lcp's? Checking the suffix tree we see that indeed we get them:

8:3=8:3 3:11=11:8 11:6=3:1
6:1=6:1 1:9=9:6 9:4=4:1
4:12=12:9 12:7=? 7:2=7:2
2:10=10:7 10:5=5:2 5:13=13:10

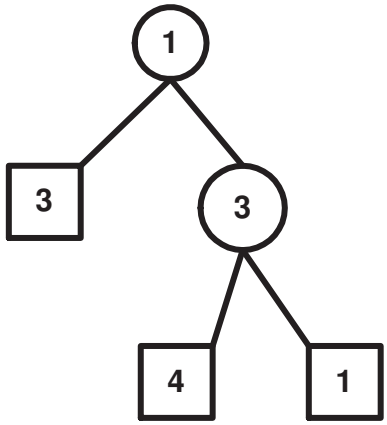
So we have the same *lcp*'s expressed using different suffixes. And, of course, we have a different set of explicit suffixes. We can build the corresponding quasi suffix tree using the same technique as for building suffix tree from suffix array:

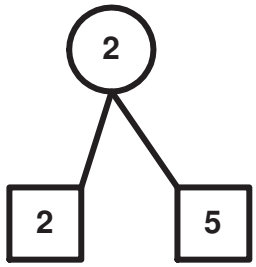


3:1

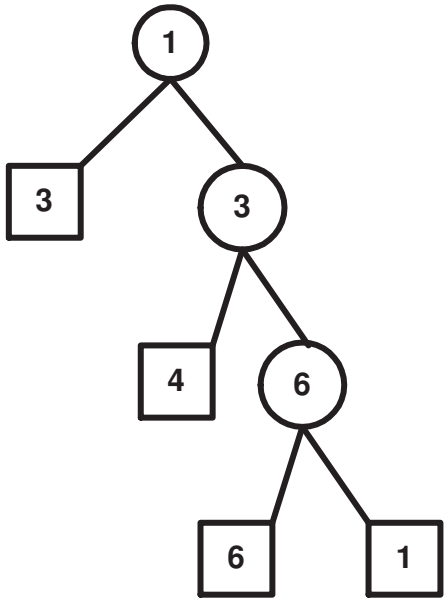


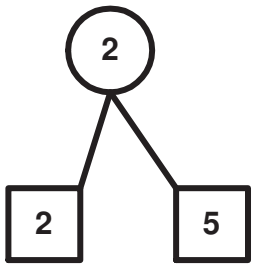
4:1



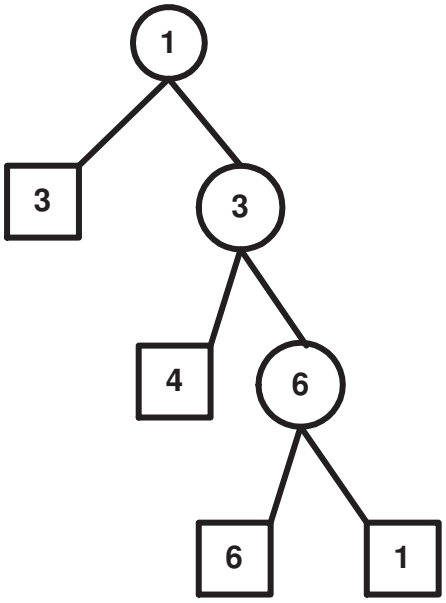


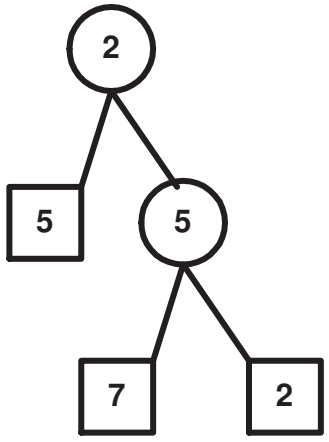
5:2



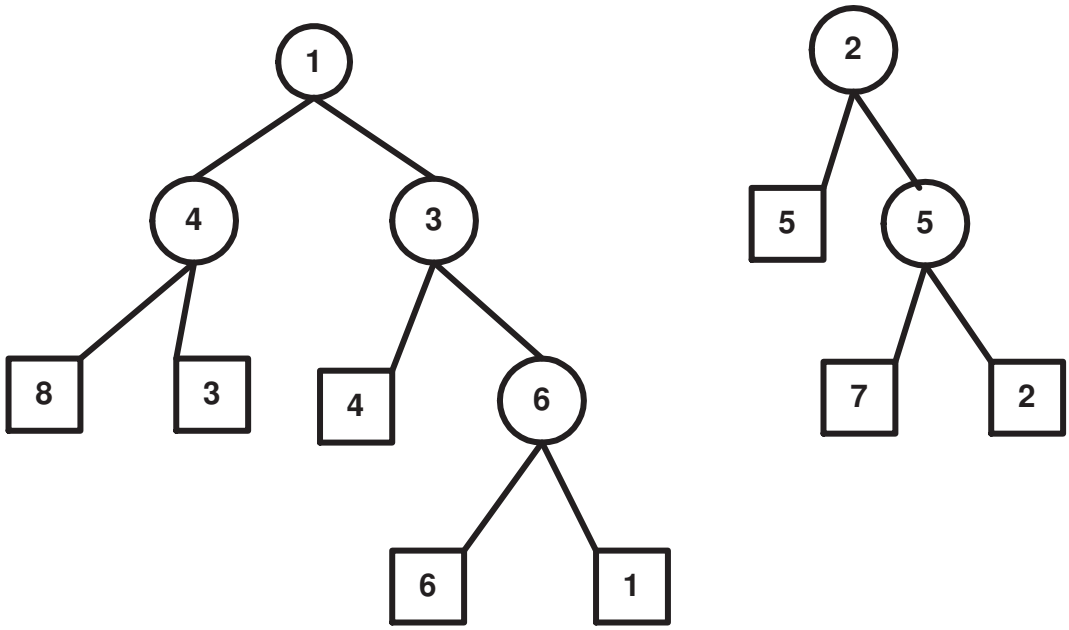


6:1

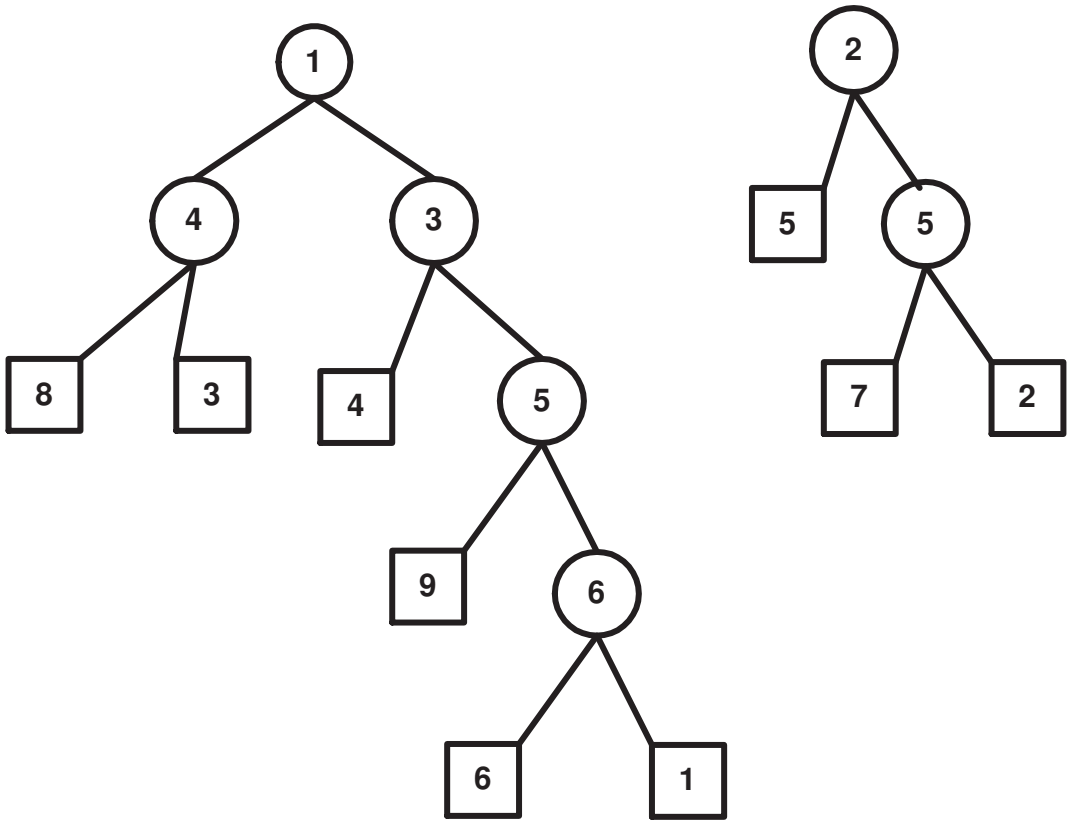




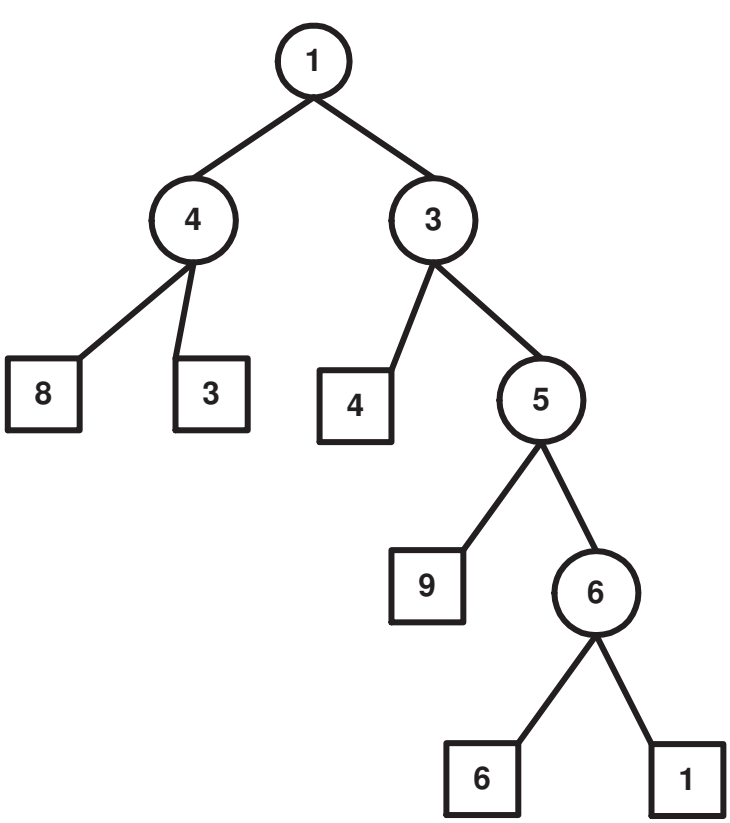
7:2

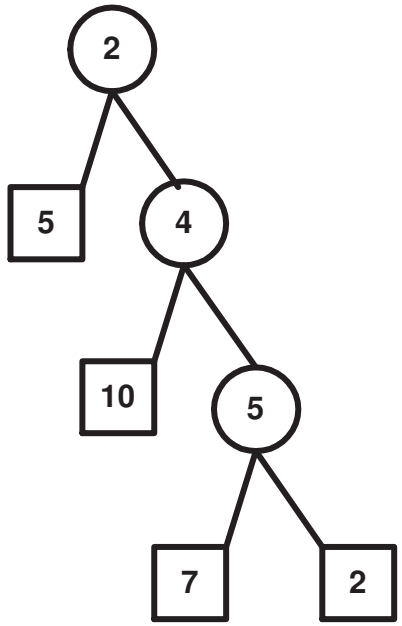


8:3

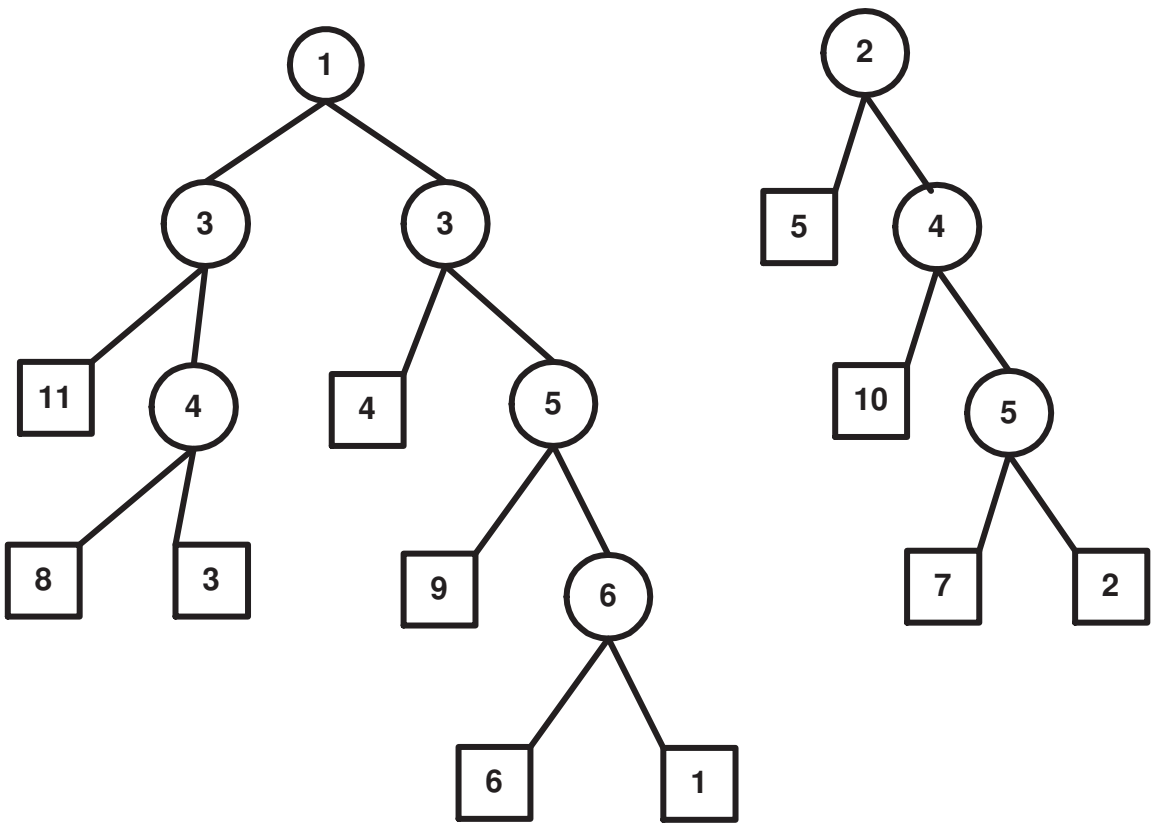


9:6

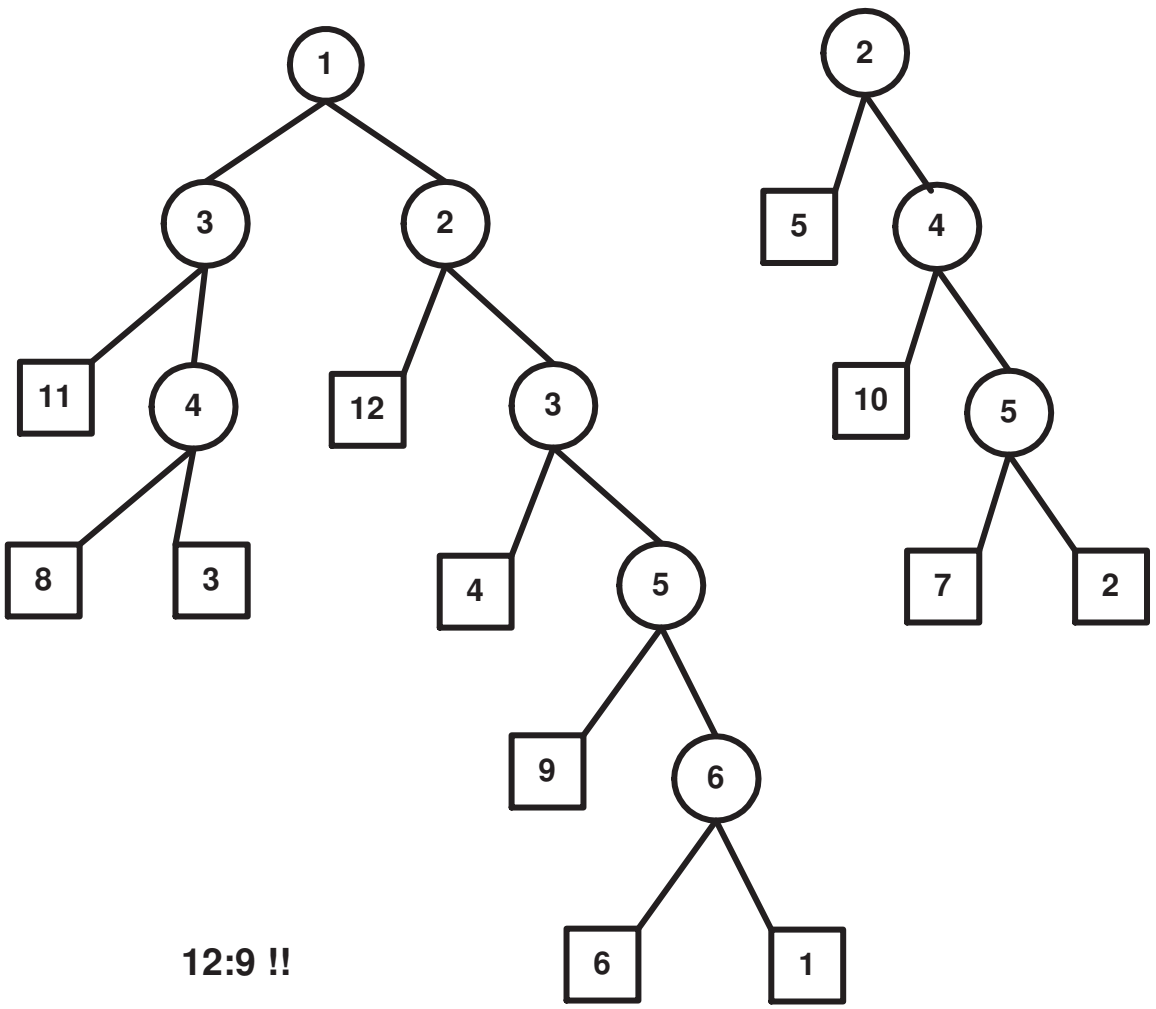




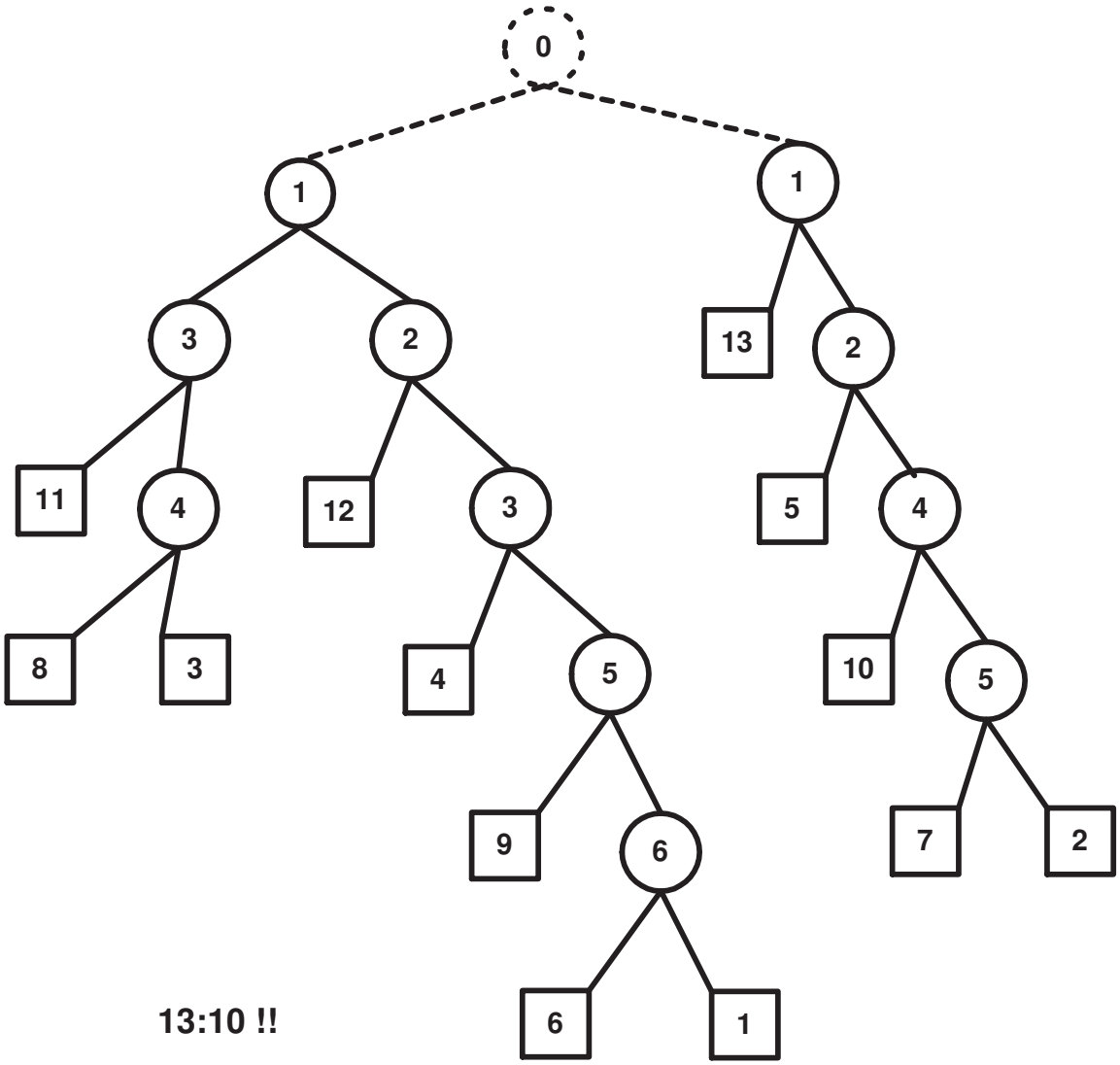
10:7



11:8



12:9 !!



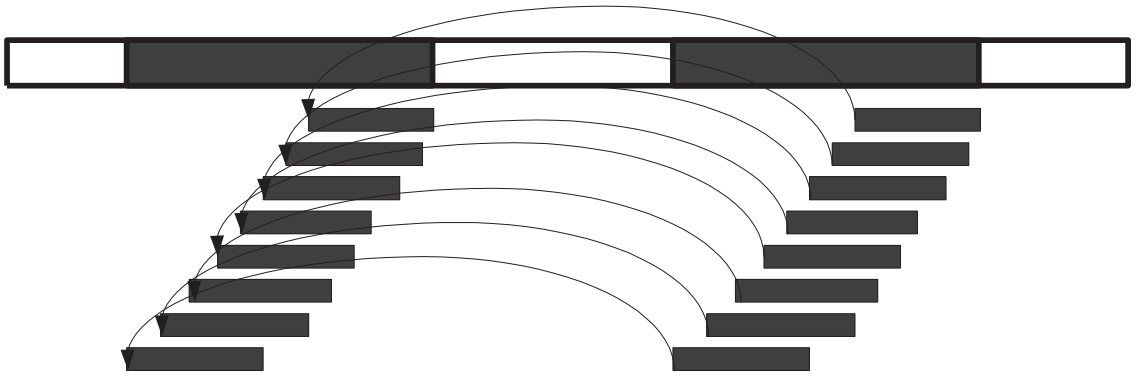
Thus, essentially (but the alphabetical ordering) we have the same structural info as with the suffix tree.

Of course, the worst-case complexity of DIST1 s at least order of n^2. The biggest problem is the repeated traversal (skips) of the classes during refinement.

In DIST2 we introduce some additional data structures (a bit array S to flag what has been visited, a stack of locations to be able to retrace a class, an array of alphabet symbols to keep the current rightmost location, a stack S' of distinct tail-letters that occurred during traversal of a class). The purpose of these structures is to traverse each class just once during refinement.

Since S and S' can share the same memory, we need $2n+2$ extra integers and n extra bits. The average (expected) complexity of DIST2 is $O(n\ log\ n)$.

There are some other possibilities how to improve the performance of the algorithm. The algorithm processes the array from right to left touching upon every entry in the array. There are though two different "blocks" that could be "hopped". One is clearly a block of consecutive zeroes (zero-hop) and one is a "triangle block" (triangle-hop). A triangle block consists of consecutive entries decreasing by 1.



DIST4 is a variant of DIST1 in which zero-hops and triangle-hops are managed at the expense of $2n$ integers of extra memory. Clearly, if a string has a lot of repeats, then triangle-hopping pays of well, if it does not, then the depth of processing is small. Our hope is to prove that DIST4 has a linear expected complexity.

Preliminary testing and comparison of the various forms of DIST

by one of the authors (Xiao) point strongly in that direction.

To conclude, we are working on "fusing" all improvements together in DIST5 that would do both, non-repeated traversal of classes (as DIST2) and hopping (as DIST4).

The open problems we intend to tackle in our future research are: what is the complexity of constructing quasi suffix tree from a quasi suffix array, how "expensive" is to transform quasi suffix tree (array) to suffix tree (array), and what is the worst-case complexity of DIST4 and what is its average case complexity.

Papers related to this topic, including these slides, can be viewed at the web site

www.cas.mcmaster.ca/∼franek