Reconstructing the suffix array

F. Franek & B. Smyth

Algorithms Research Group
Dept. of Computing & Software
McMaster University
Hamilton, Ontario

LSD'05, February 2005

Recently, we came across an interesting problem:
knowing the order of suffixes of a binary string, can one "easily" tell the order of suffixes of its complement?

"Obvious" answer --- just reverse it --- is wrong!

aaba $\quad$ **a < aaba < aba < ba**

$\overline{\textbf{aaba}}$ **= bbab** $\quad$ **ab < b < bab < bbab**

The reason: if $\mathbf{p_1} < \mathbf{p_2}$ are suffixes, either

- $\mathbf{p_1}$ is a prefix of $\mathbf{p_2}$, and then $\overline{\mathbf{p_1}} < \overline{\mathbf{p_2}}$
- $\mathbf{p_1}$ is not a prefix of $\mathbf{p_2}$, and then $\overline{\mathbf{p_1}} > \overline{\mathbf{p_2}}$

The problem can be phrased in a more general way:

knowing the order of suffixes of a string, if we reverse the order of the alphabet, can one "easily" tell the new order of suffixes?

What do we mean by "easily"? Usually a linear-time algorithm.

A simple answer:

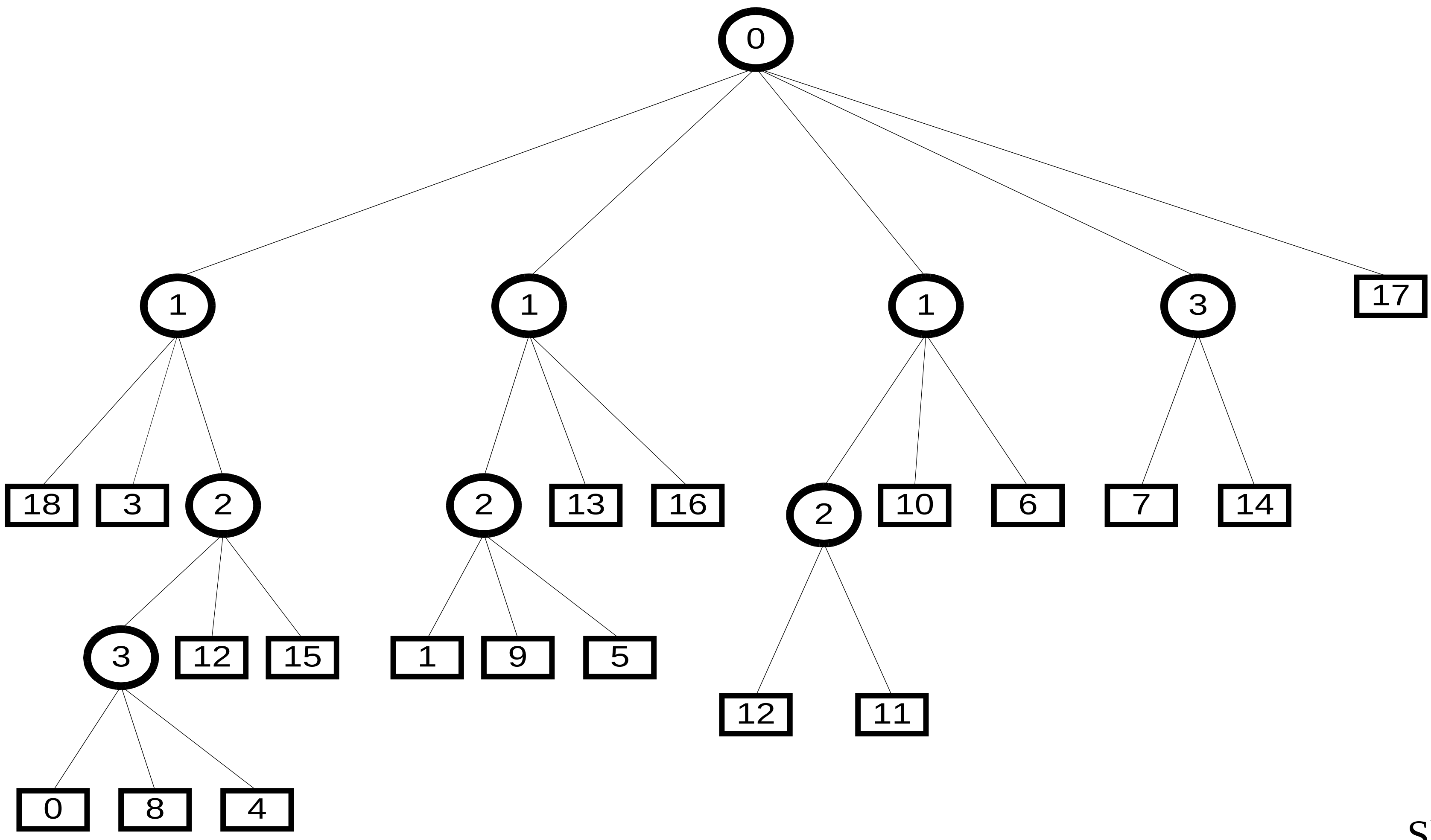
- knowing the order of suffixes, in linear time compute the lcp info (*Kasai et al*, *2001*) and build the suffix array
- having the suffix array, build the suffix tree
- invert the order of the links in every node of the suffix tree
- traverse the suffix tree in depth-first (inorder) fashion and “read out” the new order of suffixes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	**b**	**c**	**a**	**a**	**b**	**c**	**d**	**a**	**b**	**c**	**c**	**a**	**b**	**d**	**a**	**b**	**e**	**a**

18	a$
3	aabcdabccabdabea$
0	abcaabcdabccabdabea$
8	abccabdabea$
4	abcdabccabdabea$
12	abdabea$
15	abea$
1	bcaabcdabccabdabea$
9	bccabdabea$
5	bcdabccabdabea$
13	bdabea$
16	bea$
2	caabcdabccabdabea$
11	cabdabea$
10	ccabdabea$
6	cdabccabdabea$
7	dabccabdabea$
14	dabea$
17	ea$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	c	a	a	b	c	d	a	b	c	c	a	b	d	a	b	e	a

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
18	3	0	8	4	12	15	1	9	5	13	16	2	11	10	6	7	14	17
	1	1	3	3	2	2	0	2	2	1	1	0	2	1	1	0	3	0



The problem with this solution: a suffix tree requires between 5N to 10N words of memory (N the length of the string), on top of this you need some working memory.

So, what do we really mean by “easily”? A linear-time and a memory-efficient algorithm.

We designed a linear-time iterative (non-recursive) algorithm that requires a working memory of 2N words and that re-sorts suffixes of a string after reversing the order of the alphabet.

The algorithm is based on the observation that the order of suffixes will be reversed as well, with the exception of prefixes, for which the order will be preserved. So, we need to be able to determine for each pair of suffixes $\mathbf{p_1} < \mathbf{p_2}$ whether $\mathbf{p_1}$ is a prefix of $\mathbf{p_2}$ or not.

For that purpose we will compute in essence a ***reverse border array***:

rba[i]=j iff **x[i..N]** has **x[j..N]** as a maximal border.

A simple modification of the ***failure function algorithm*** (*Aho, Hopcroft, Ullman, 1974*) can compute **rba[ ]** in linear time.

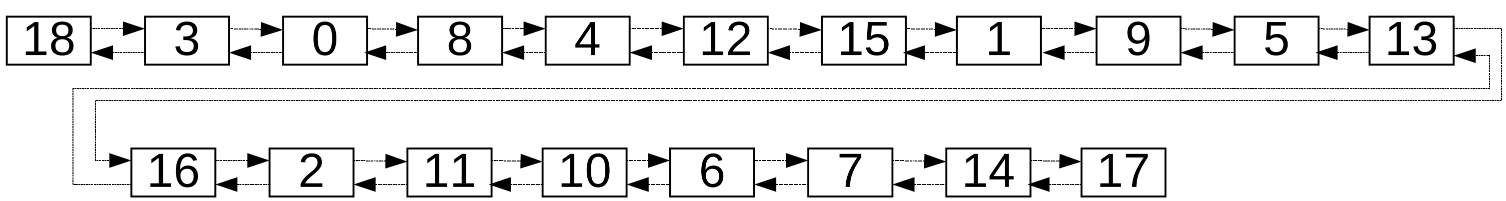
Algorithm 1

- let **s1..sN** be the suffixes sorted according to the original order of the alphabet
- let **t1..tk-**1 be **s1..sk-**1 sorted according to the reversed order of the alphabet, let **sk=x[p..**N**]**
- if **rba[p]=null** then move **sk** to the front of **t1..tk-1**
- if **rba[p] ≠null** then move **sk** right behind **tr** where **tr=**x**[rba[p]..**N**]**

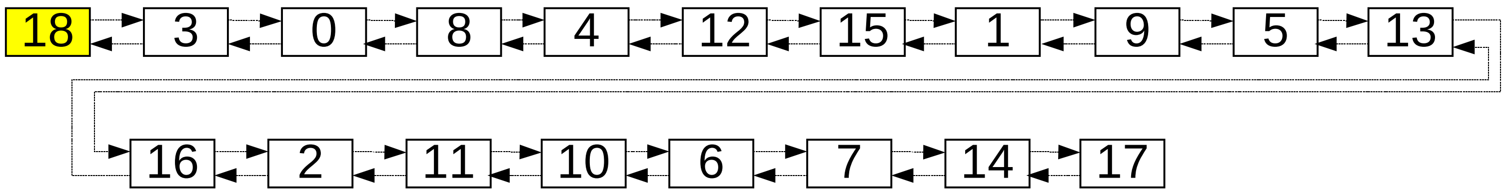
Let us run through an example for illustration:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	a	**b**	**c**	**a**	**a**	**b**	**c**	**d**	**a**	**b**	**c**	**c**	**a**	**b**	**d**	**a**	**b**	**e**	**a**
suff	18	3	0	8	4	12	15	1	9	5	13	16	2	11	10	6	7	14	17

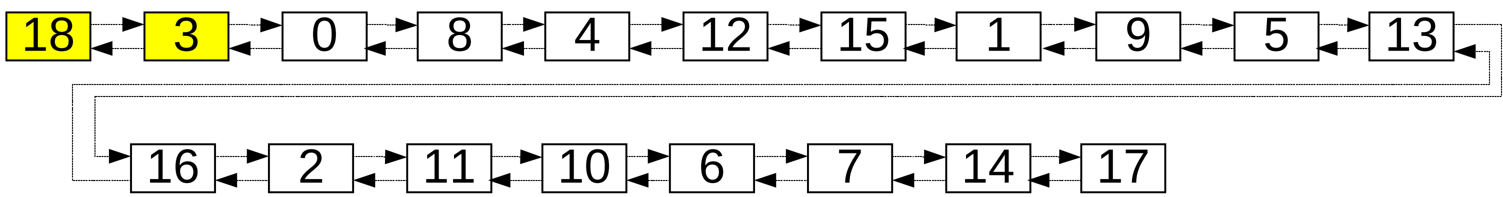
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
next	8	9	11	0	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
prev	3	15	16	18	8	9	10	6	0	1	11	2	4	5	7	12	13	14	Ø
ba	18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



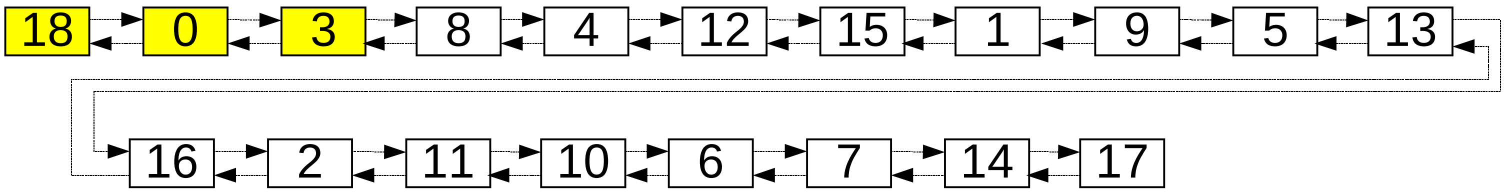
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
8	9	11	0	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
3	15	16	18	8	9	10	6	0	1	11	2	4	5	7	12	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



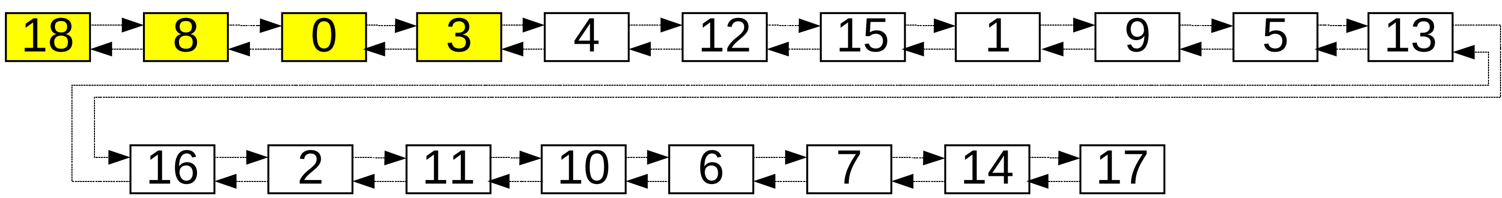
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
8	9	11	0	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
3	15	16	18	8	9	10	6	0	1	11	2	4	5	7	12	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



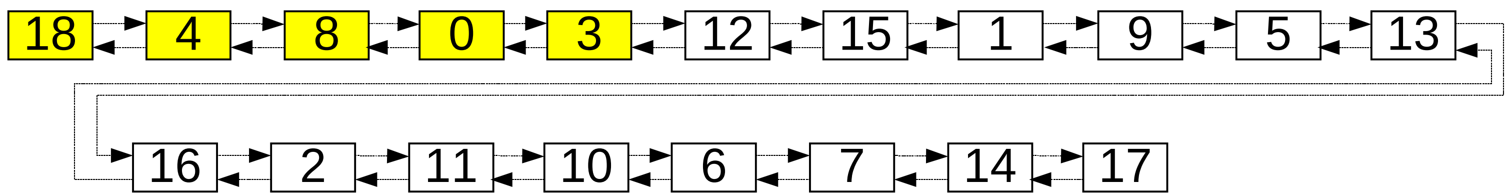
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	9	11	8	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	0
18	15	16	0	8	9	10	6	0	1	11	2	4	5	7	12	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



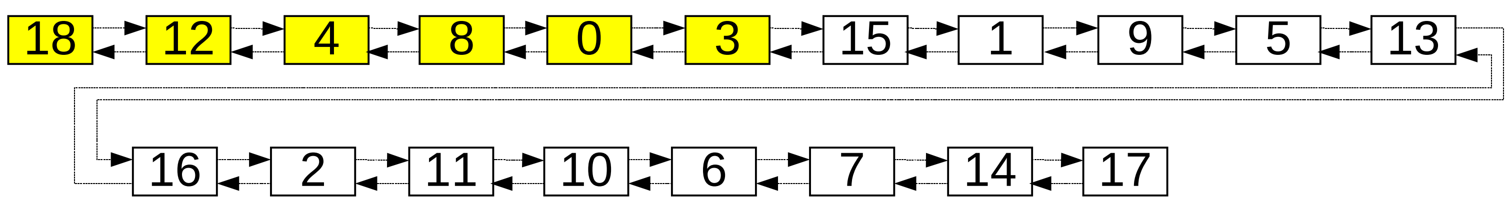
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	9	11	4	12	13	7	14	0	5	6	10	15	16	17	1	2	Ø	8
8	15	16	0	3	9	10	6	18	1	11	2	4	5	7	12	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



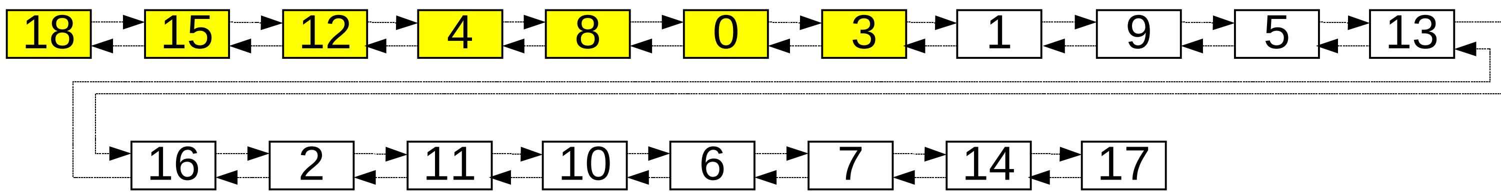
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	9	11	12	8	13	7	14	0	5	6	10	15	16	17	1	2	Ø	4
8	15	16	0	18	9	10	6	4	1	11	2	3	5	7	12	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



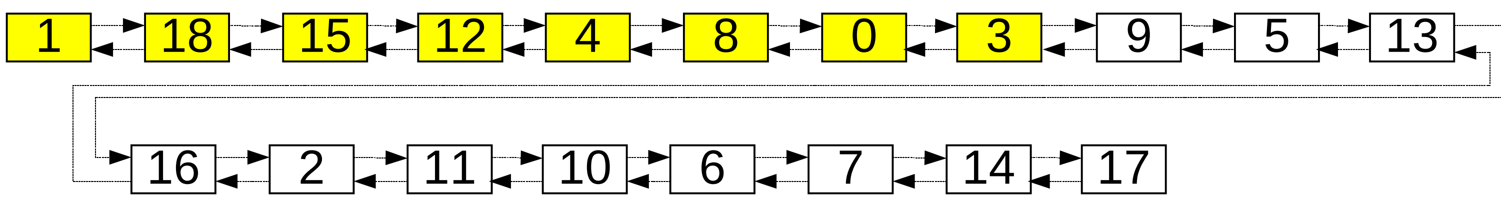
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	9	11	15	8	13	7	14	0	5	6	10	4	16	17	1	2	Ø	12
8	15	16	0	12	9	10	6	4	1	11	2	18	5	7	3	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



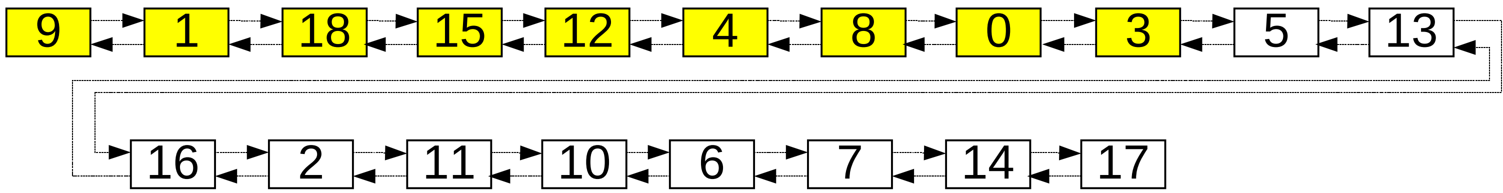
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	9	11	1	8	13	7	14	0	5	6	10	4	16	17	12	2	Ø	15
8	3	16	0	12	9	10	6	4	1	11	2	15	5	7	18	13	14	Ø
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



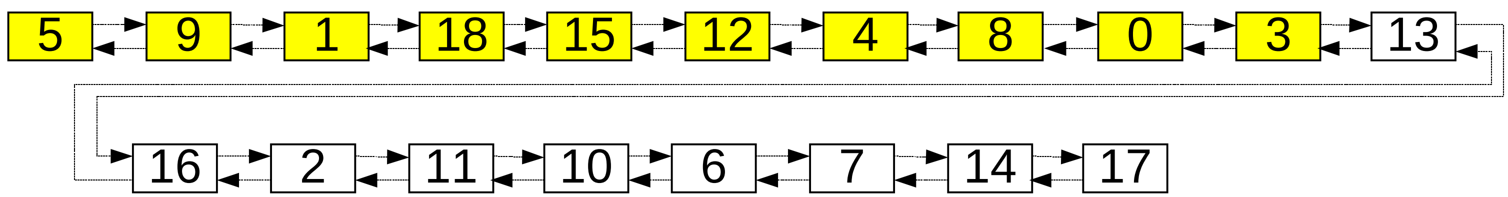
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	11	9	8	13	7	14	0	5	6	10	4	16	17	12	2	Ø	15
8	Ø	16	0	12	9	10	6	4	3	11	2	15	5	7	18	13	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



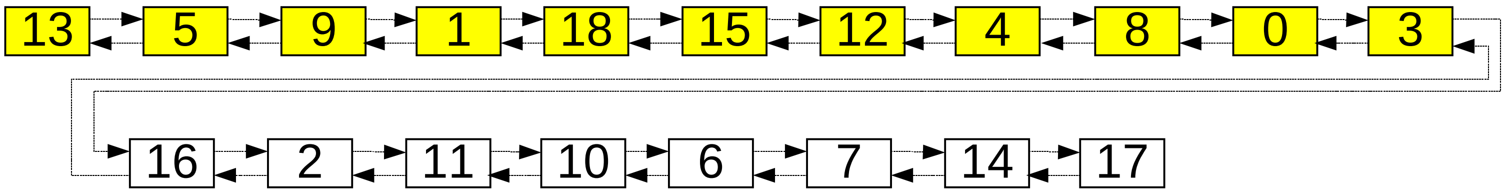
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	11	5	8	13	7	14	0	1	6	10	4	16	17	12	2	Ø	15
8	9	16	0	12	3	10	6	4	Ø	11	2	15	5	7	18	13	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



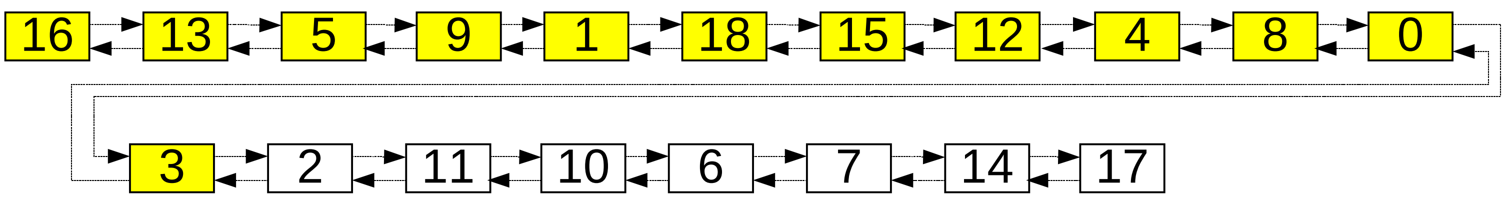
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	11	13	8	9	7	14	0	1	6	10	4	16	17	12	2	Ø	15
8	9	16	0	12	Ø	10	6	4	5	11	2	15	3	7	18	13	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



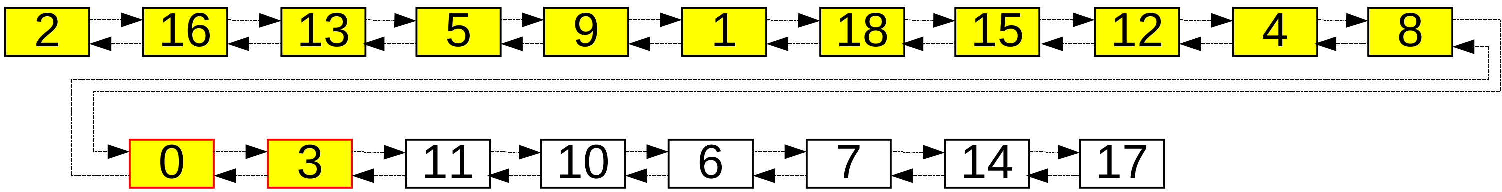
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	11	16	8	9	7	14	0	1	6	10	4	5	17	12	2	Ø	15
8	9	16	0	12	13	10	6	4	5	11	2	15	Ø	7	18	3	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



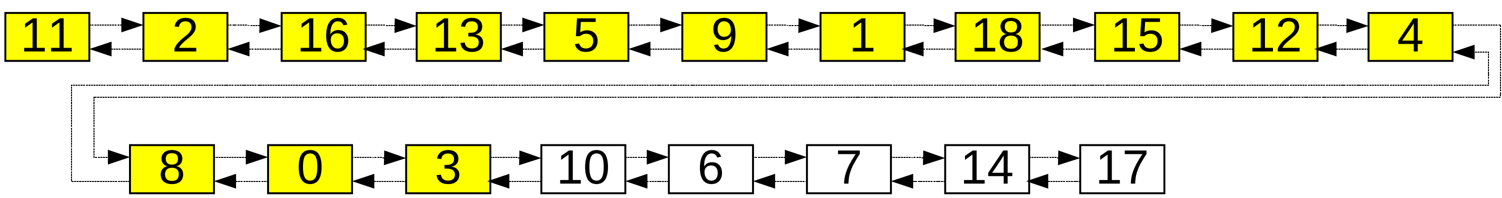
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	11	2	8	9	7	14	0	1	6	10	4	5	17	12	13	Ø	15
8	9	3	0	12	13	10	6	4	5	11	2	15	16	7	18	Ø	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



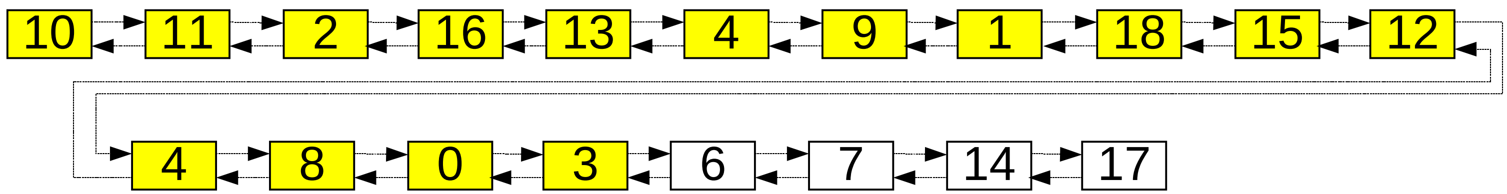
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	11	8	9	7	14	0	1	6	10	4	5	17	12	13	Ø	15
8	9	Ø	0	12	13	10	6	4	5	11	3	15	16	7	18	2	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



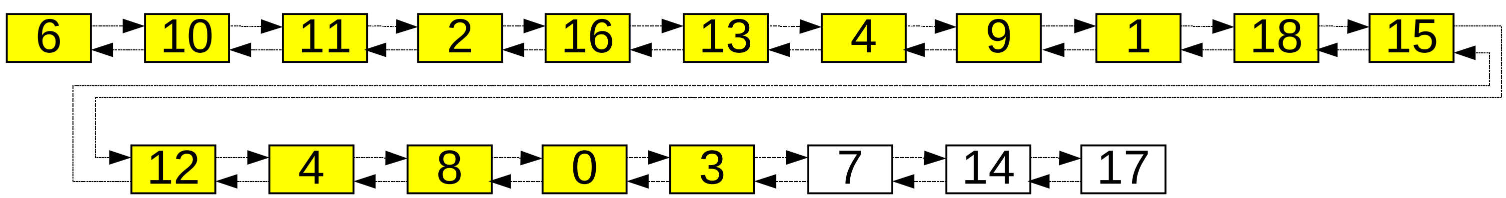
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	10	8	9	7	14	0	1	6	2	4	5	17	12	13	Ø	15
8	9	11	0	12	13	10	6	4	5	3	Ø	15	16	7	18	2	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



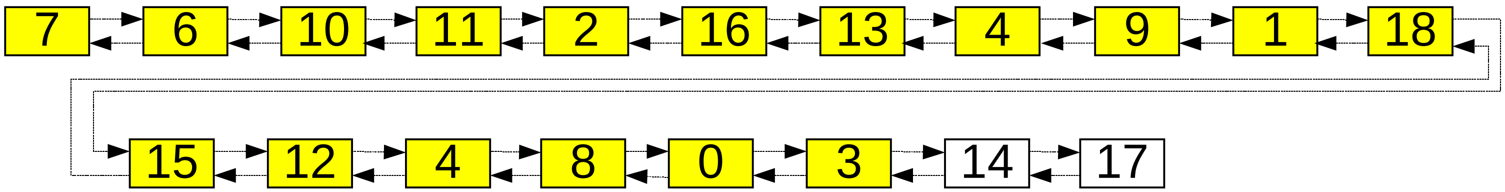
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	6	8	9	7	14	0	1	11	2	4	5	17	12	13	Ø	15
8	9	11	0	12	13	3	6	4	5	Ø	Ø	15	16	7	18	2	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



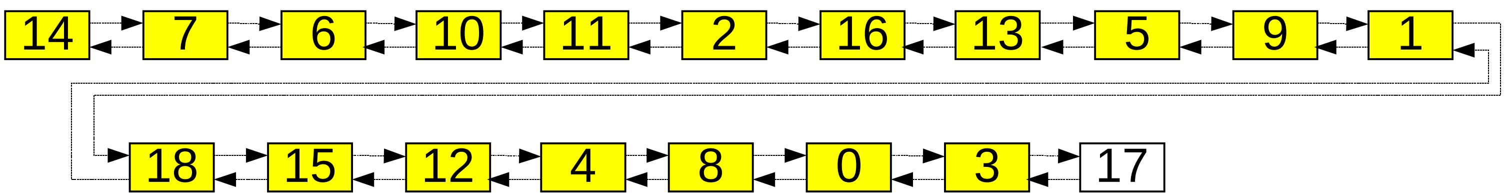
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	7	8	9	10	14	0	1	11	2	4	5	17	12	13	Ø	15
8	9	11	0	12	13	Ø	3	4	5	6	Ø	15	16	7	18	2	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



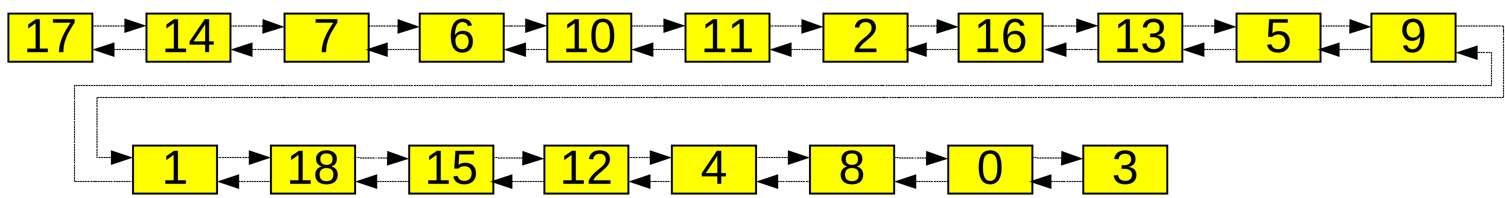
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	14	8	9	10	6	0	1	11	2	4	5	17	12	13	Ø	15
8	9	11	0	12	13	7	Ø	4	5	6	Ø	15	16	3	18	2	14	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	17	8	9	10	6	0	1	11	2	4	5	7	12	13	Ø	15
8	9	11	0	12	13	7	14	4	5	6	Ø	15	16	Ø	18	2	3	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	18	16	Ø	8	9	10	6	0	1	11	2	4	5	7	12	13	14	15
8	9	11	0	12	13	7	14	4	5	6	Ø	15	16	17	18	2	Ø	1
18	Ø	Ø	18	18	Ø	Ø	Ø	18	Ø	Ø	Ø	18	Ø	Ø	18	Ø	Ø	Ø



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
17	14	7	6	10	11	2	16	13	5	9	1	18	15	12	4	8	0	3

What about an arbitrary permutation of the order of the alphabet?

There we run into several problems. Again, for a suffix tree all we have to worry about is the permutation itself, i.e. how complicated it is to sort each “family” of suffixes (resorting the families of links may not be linear). More about the complexity of permutations later.

If we start with a suffix array of a string, rather than just an ordered sequence of suffixes as we did for the inversion of the alphabet, we can identify and sort “families” as in suffix tree. This “extra” work can be done in linear-time with <=2N words of working memory.

The iterative (non-recursive) algorithm relies on four steps repeated until the end of processing. These four steps are: identify-and-extract family, sort the family, flatten the family, and verticalize the family. The following example will illustrate these procedures:

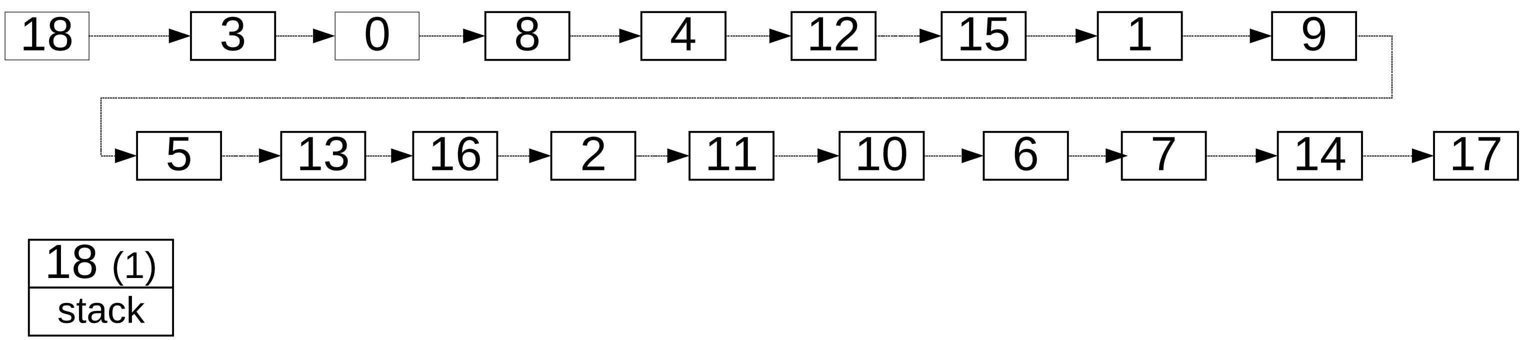
We are using again the same string as in Slide 6 . The permutation of the alphabet for this examples is defined by

p[a]=a
p[b]=c
p[c]=d
p[d]=b
p[e]=e

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	c	a	a	b	c	d	a	b	c	c	a	b	d	a	b	e	a

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
18	3	0	8	4	12	15	1	9	5	13	16	2	11	10	6	7	14	17
	1	1	3	3	2	2	0	2	2	1	1	0	2	1	1	0	3	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
lcp	3	2	2	1	2	1	0	3	3	2	1	1	2	1	0	0	0	Ø	1
next	8	9	11	0	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
tail	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	2	1	0	3	3	2	1	1	2	1	0	0	0	Ø	1
8	9	11	0	12	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 0 → 8 → 4 → 12 → 15 → 1 → 9 → 5 → 13 → 16 → 2 → 11 → 10 → 6 → 7 → 14 → 17

0 (3)
18 (1)
stack

identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	2	1	0	3	3	2	1	1	2	1	0	0	0	Ø	1
8	9	11	0	Ø	13	7	14	4	5	6	10	15	16	17	1	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 0 → 8 → 4   12 → 15 → 1 → 9 → 5 → 13 → 16 → 2 → 11 → 10 → 6 → 7 → 14 → 17

0 (3)
18 (1)
stack

3-family

3	3	2
0	8	4

0 (3)
18 (1)
stack

sort the family

3-family

3	3	2
0	8	4

→

3	3	2
0	4	8

lcp "sort"
1. last is preserved
2. every >= k preserved
3. everything else is k

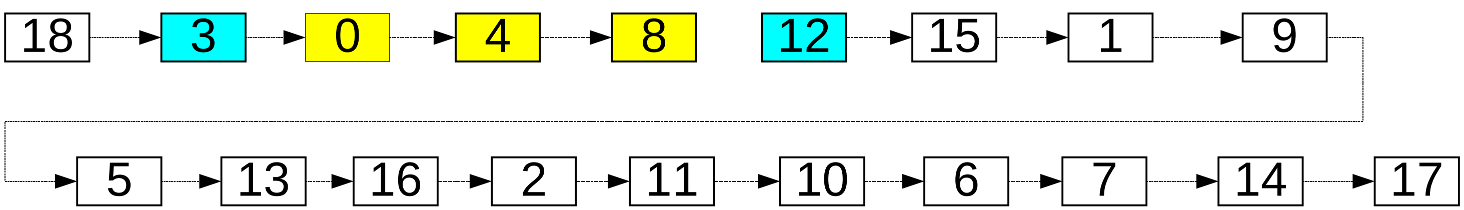
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	0	8	13	7	14	Ø	5	6	10	15	16	17	1	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 0 → 4 → 8   12 → 15 → 1 → 9 →
→ 5 → 13 → 16 → 2 → 11 → 10 → 6 → 7 → 14 → 17

flatten the family

nothing to flatten, it is already flat

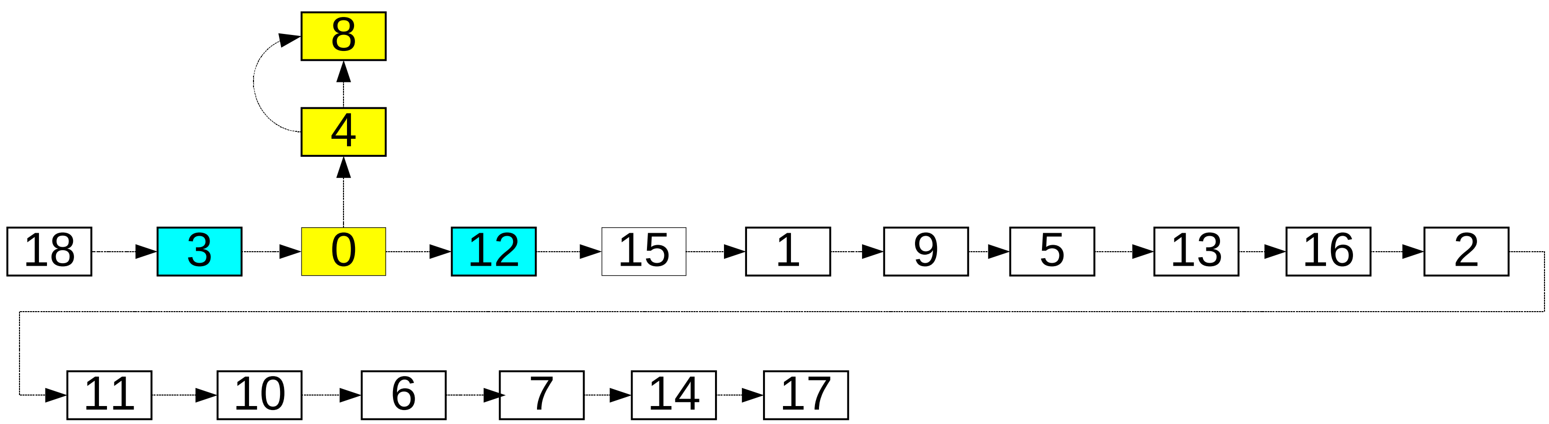
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	0	8	13	7	14	Ø	5	6	10	15	16	17	1	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø



verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
12	9	11	12	8	13	7	14	Ø	5	6	10	15	16	17	1	2	Ø	3
4	Ø	Ø	Ø	8	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

tailnext tailend



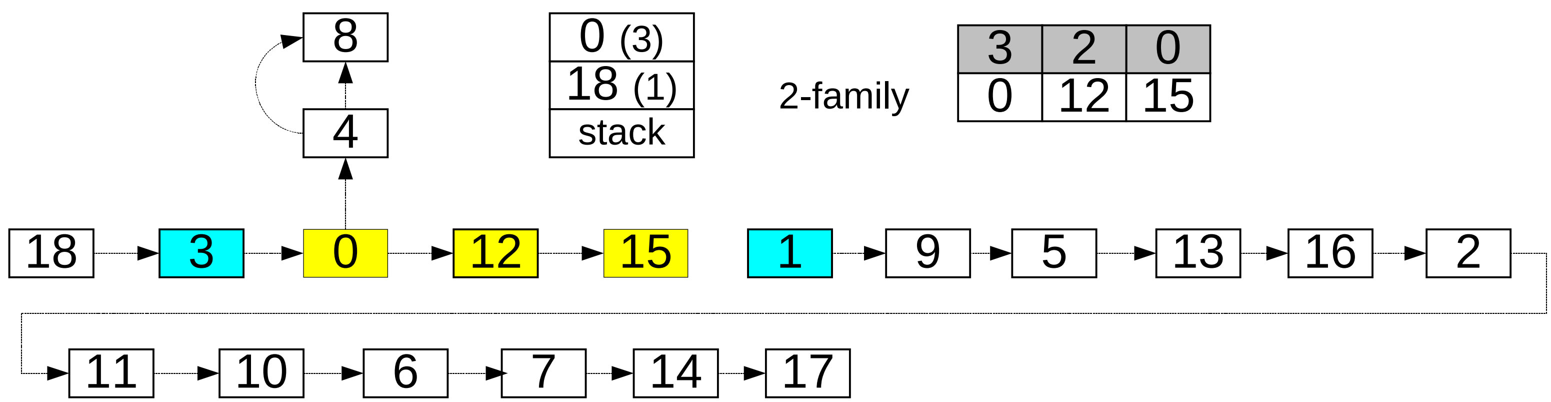
identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
12	9	11	12	8	13	7	14	Ø	5	6	10	15	16	17	Ø	2	Ø	3
4	Ø	Ø	Ø	8	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

tailnext tailend

2-family

3	2	0
0	12	15



sort the family

0 (3)
18 (1)
stack

2-family

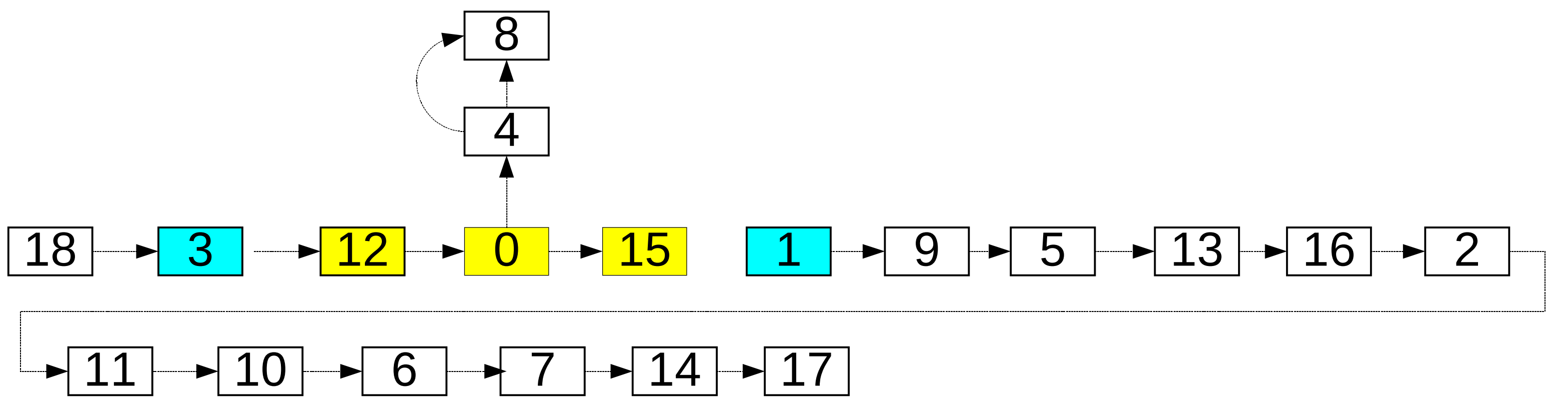
3	2	0
0	12	15

→

2	3	0
12	0	15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
15	9	11	12	8	13	7	14	Ø	5	6	10	0	16	17	Ø	2	Ø	3
4	Ø	Ø	Ø	8	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

tailnext tailend



flatten the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	0	16	17	Ø	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 12 → 0 → 4 → 8 → 15   1 → 9 →

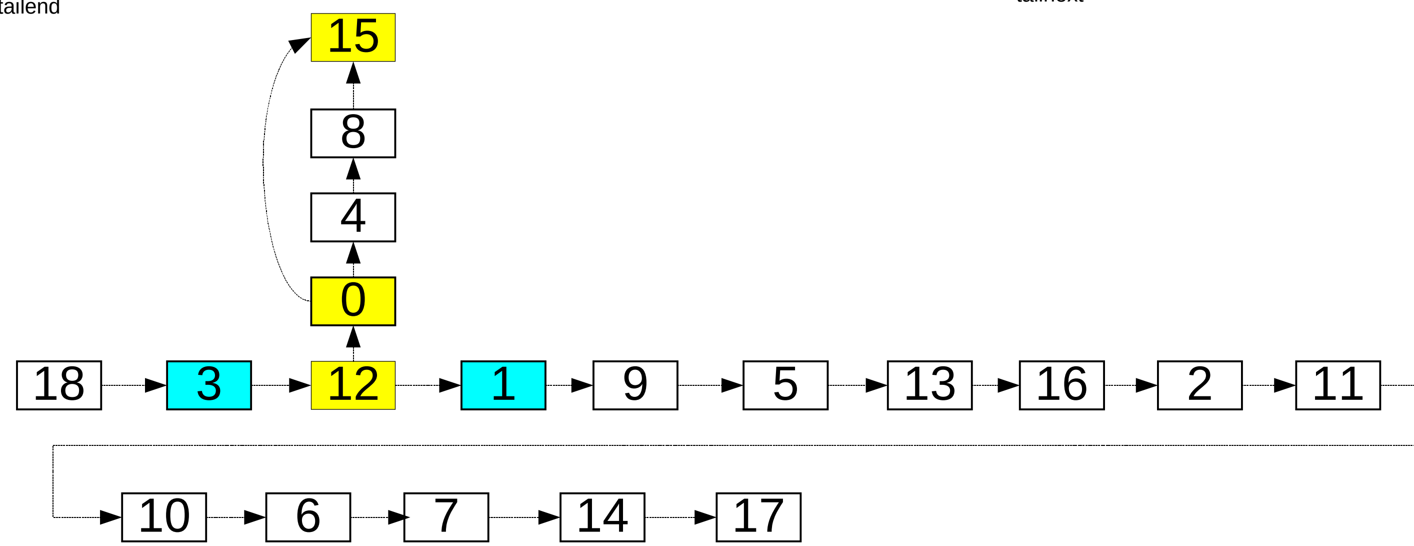
→ 5 → 13 → 16 → 2 → 11 → 10 → 6 → 7 → 14 → 17

verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	1	16	17	Ø	2	Ø	3
15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	0	Ø	Ø	Ø	Ø	Ø	Ø

tailend

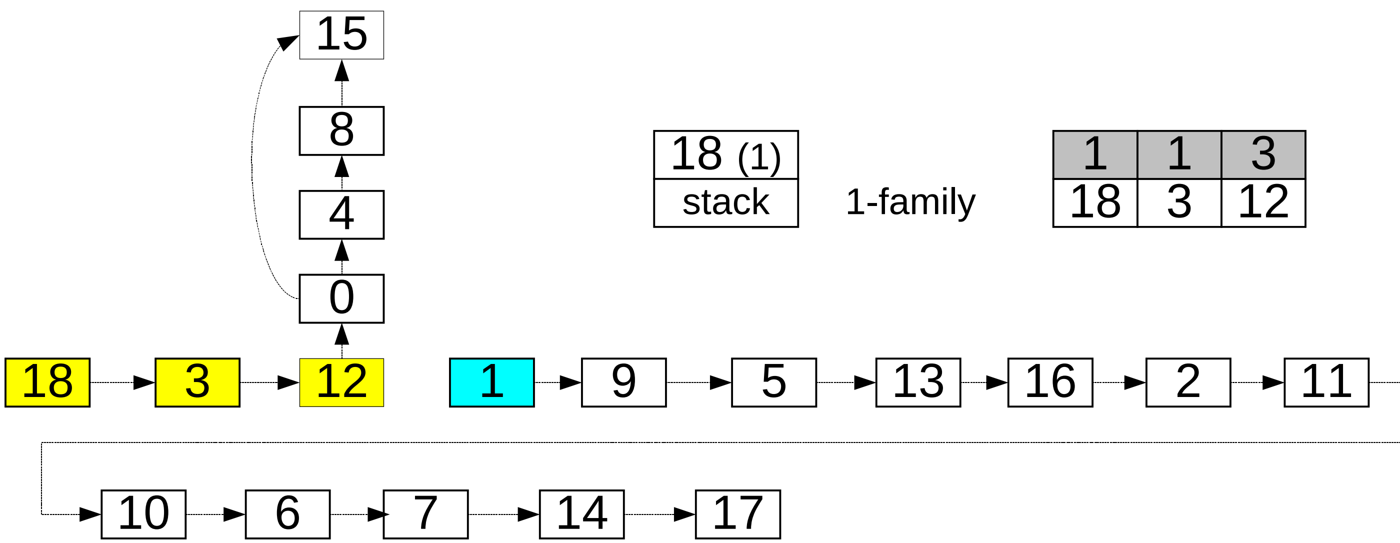
tailnext



identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	Ø	16	17	Ø	2	Ø	3
15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	0	Ø	Ø	Ø	Ø	Ø	Ø

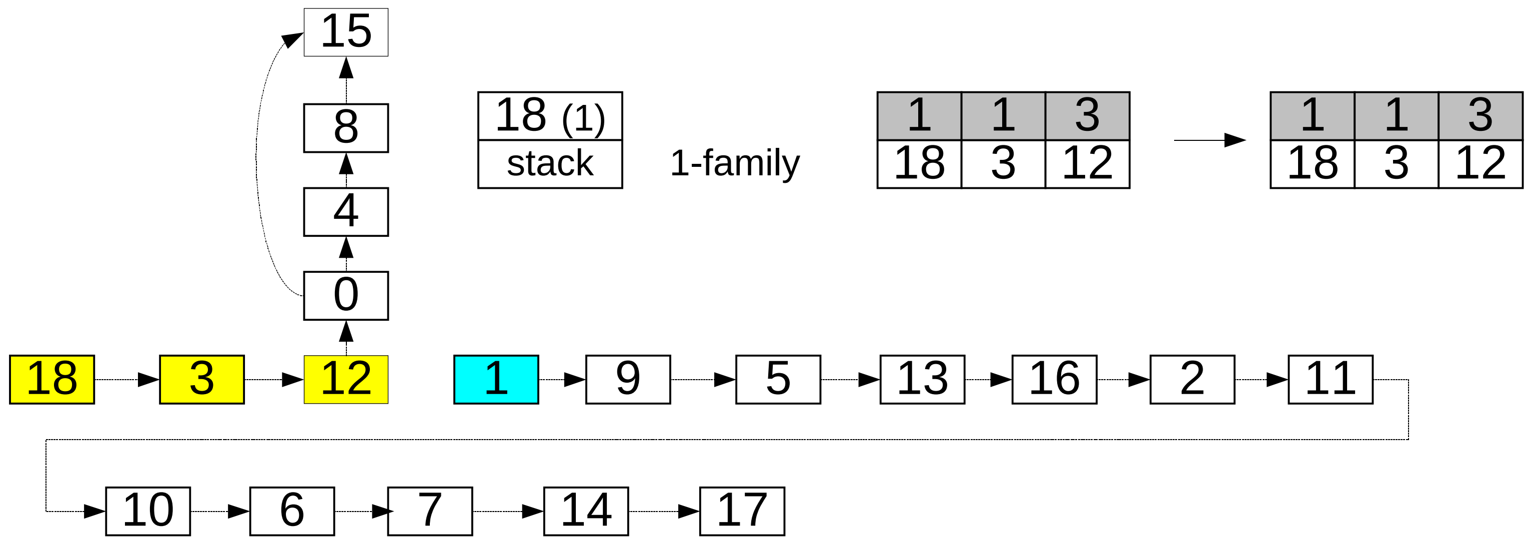
tailend tailnext



sort the family

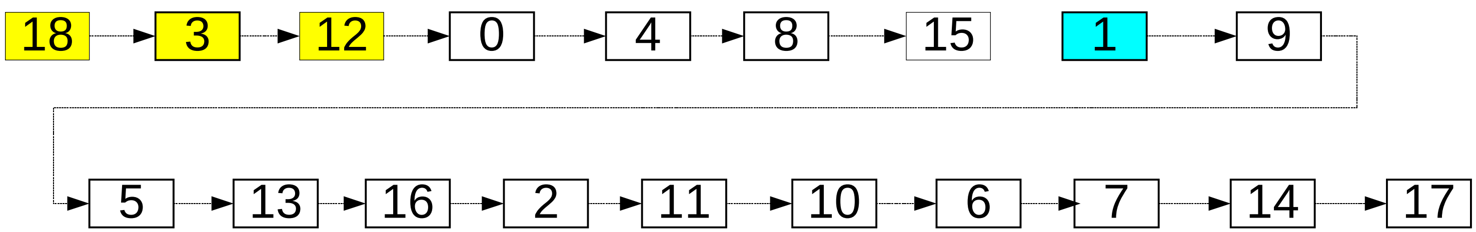
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	Ø	16	17	Ø	2	Ø	3
15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	0	Ø	Ø	Ø	Ø	Ø	Ø

tailend tailnext



flatten the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	0	16	17	Ø	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

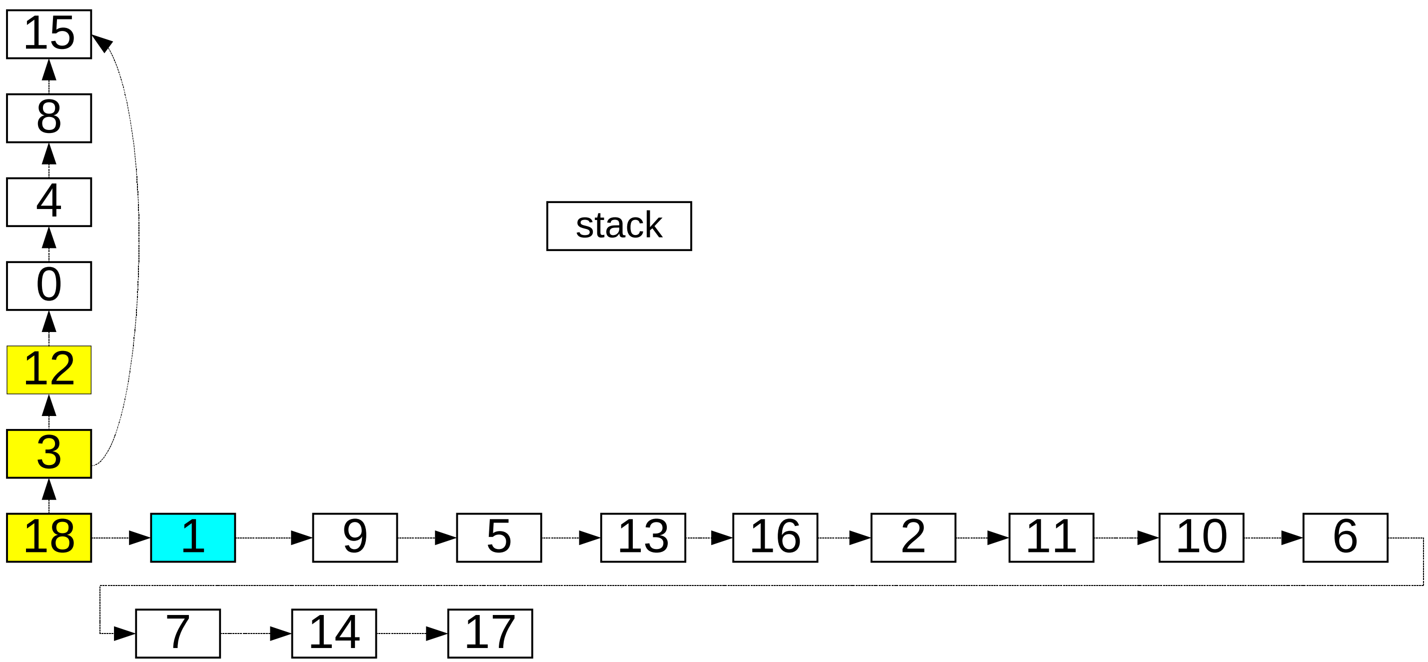


verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	13	7	14	15	5	6	10	0	16	17	Ø	2	Ø	1
Ø	Ø	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3

tailend

tailnext

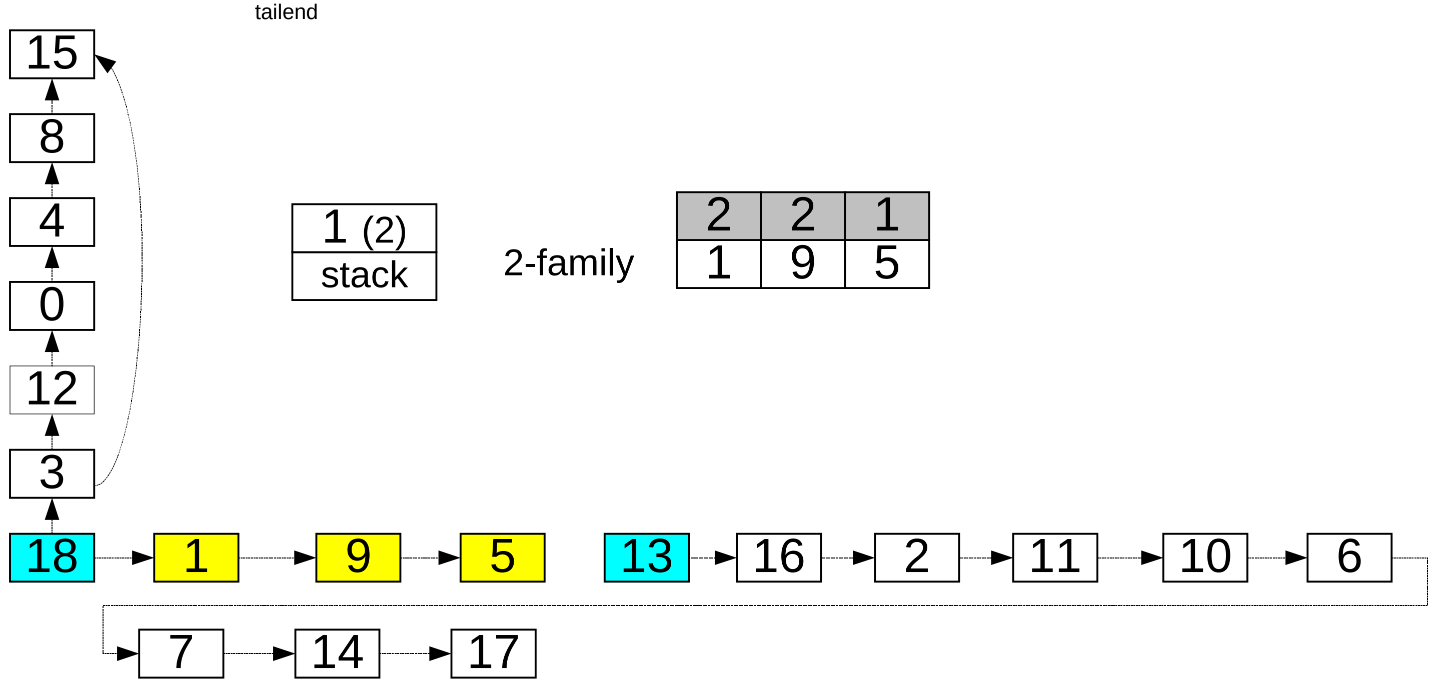


identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	1	0	3	2	2	1	1	2	1	0	0	0	Ø	1
4	9	11	12	8	Ø	7	14	15	5	6	10	0	16	17	Ø	2	Ø	1
Ø	Ø	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3

tailend

tailnext



1 (2)
stack

2-family

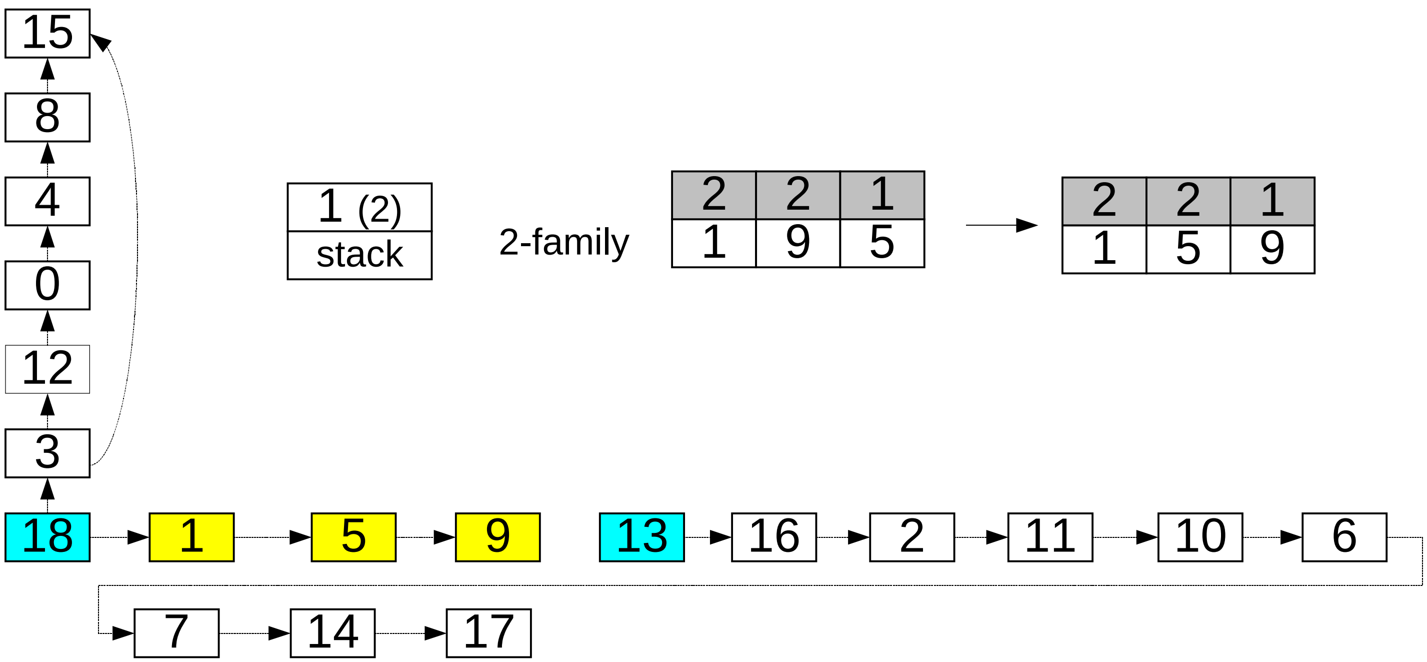
2	2	1
1	9	5

sort the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	Ø	6	10	0	16	17	Ø	2	Ø	1
Ø	Ø	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3

tailend

tailnext



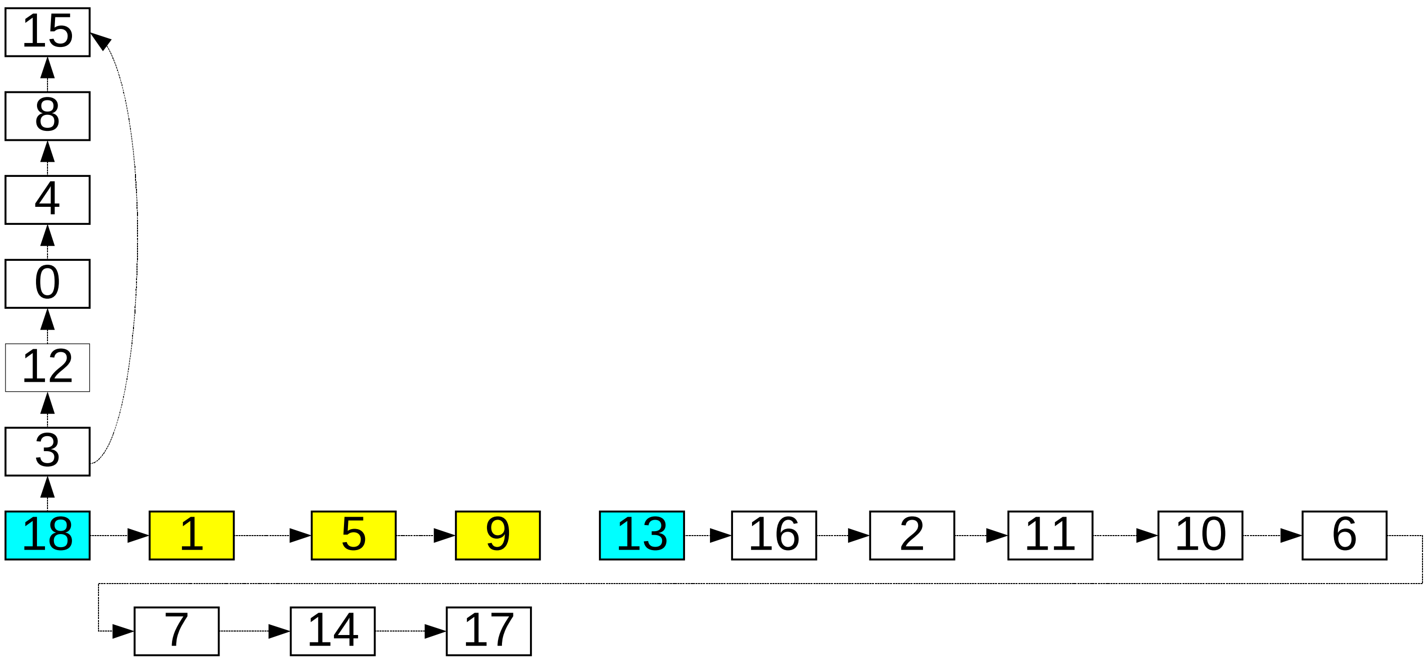
flatten the family

nothing to flatten, it is already flat

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	Ø	6	10	0	16	17	Ø	2	Ø	1
Ø	Ø	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3

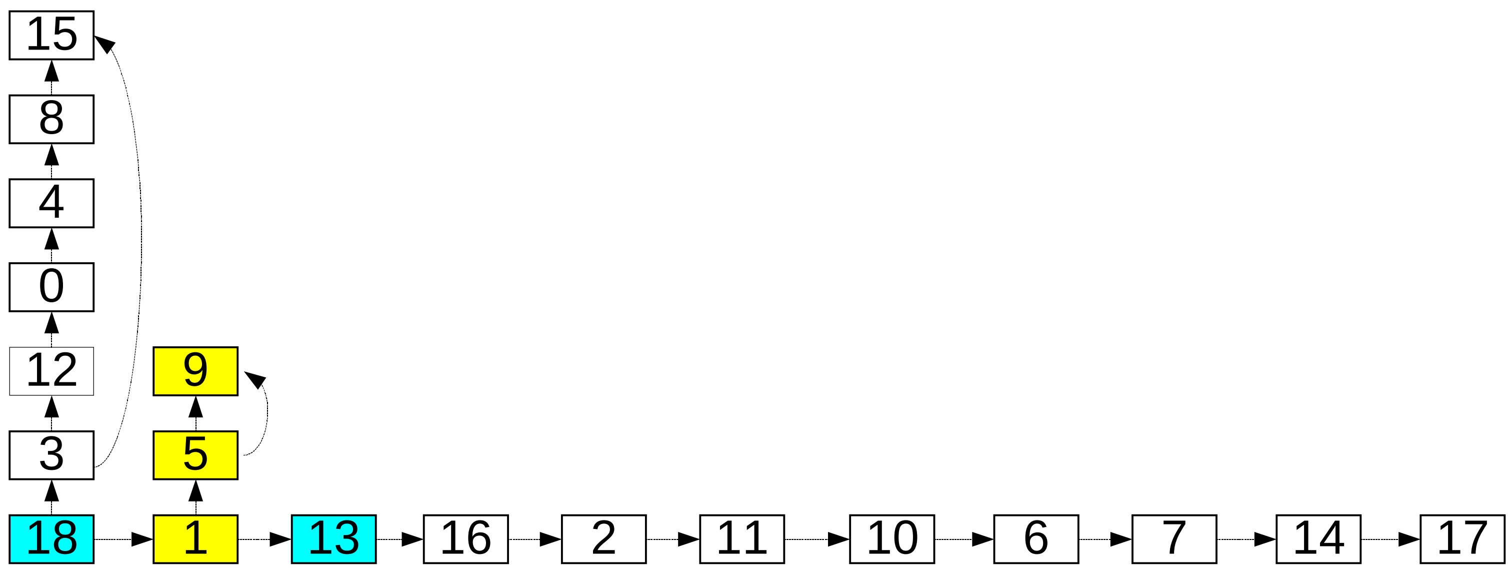
tailend

tailnext



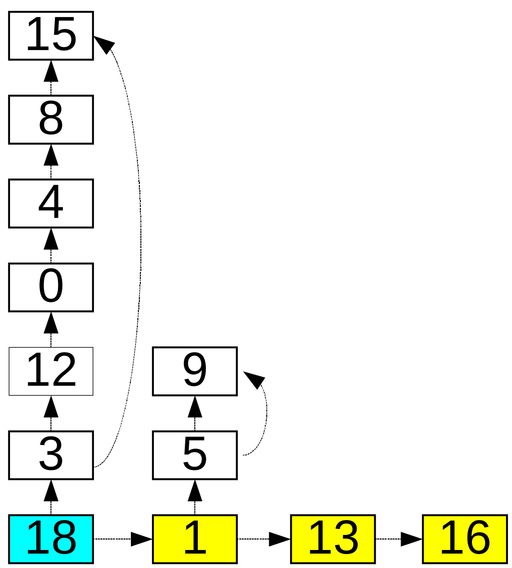
verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	13	11	12	8	9	7	14	15	Ø	6	10	0	16	17	Ø	2	Ø	1
Ø	5	Ø	15	Ø	9	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3
	tailnext		tailend		tailend													tailnext



identify and extract family

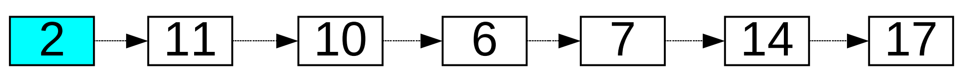
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	13	11	12	8	9	7	14	15	Ø	6	10	0	16	17	Ø	Ø	Ø	1
Ø	5	Ø	15	Ø	9	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3
	tailnext		tailend		tailend													tailnext



1(2)
stack

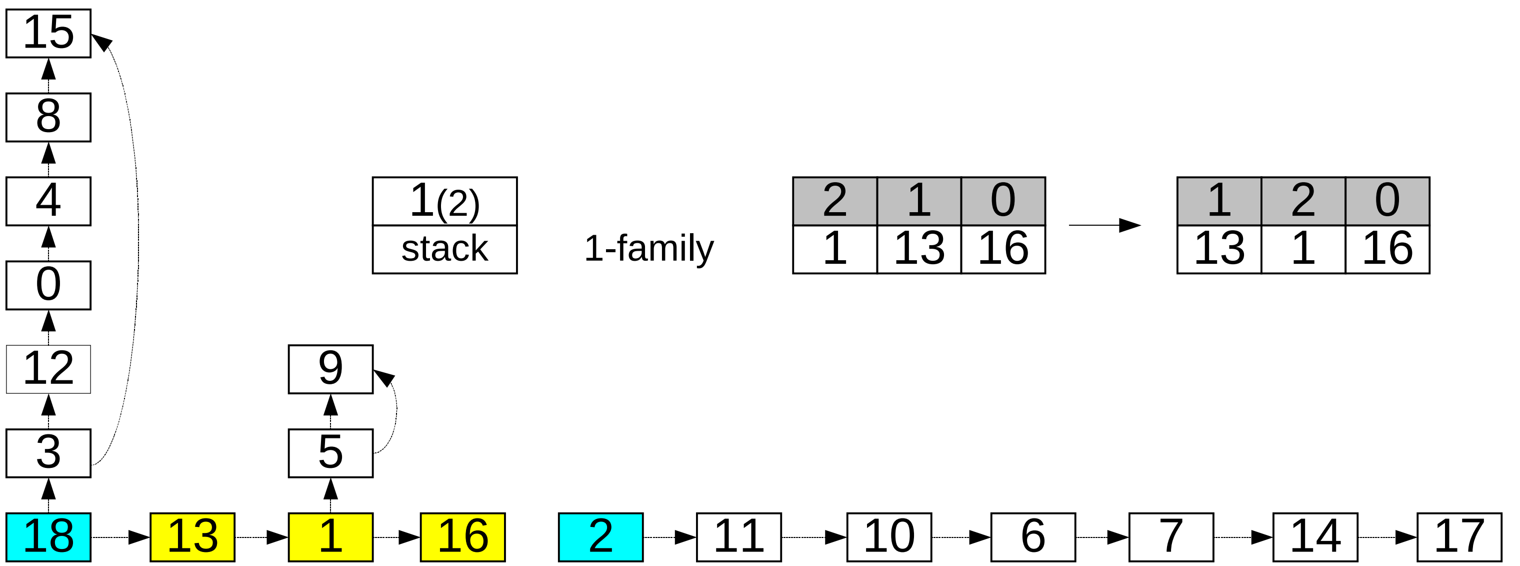
1-family

2	1	0
1	13	16



sort the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	16	11	12	8	9	7	14	15	Ø	6	10	0	1	17	Ø	Ø	Ø	13
Ø	5	Ø	15	Ø	9	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3
	tailnext		tailend		tailend													tailnext

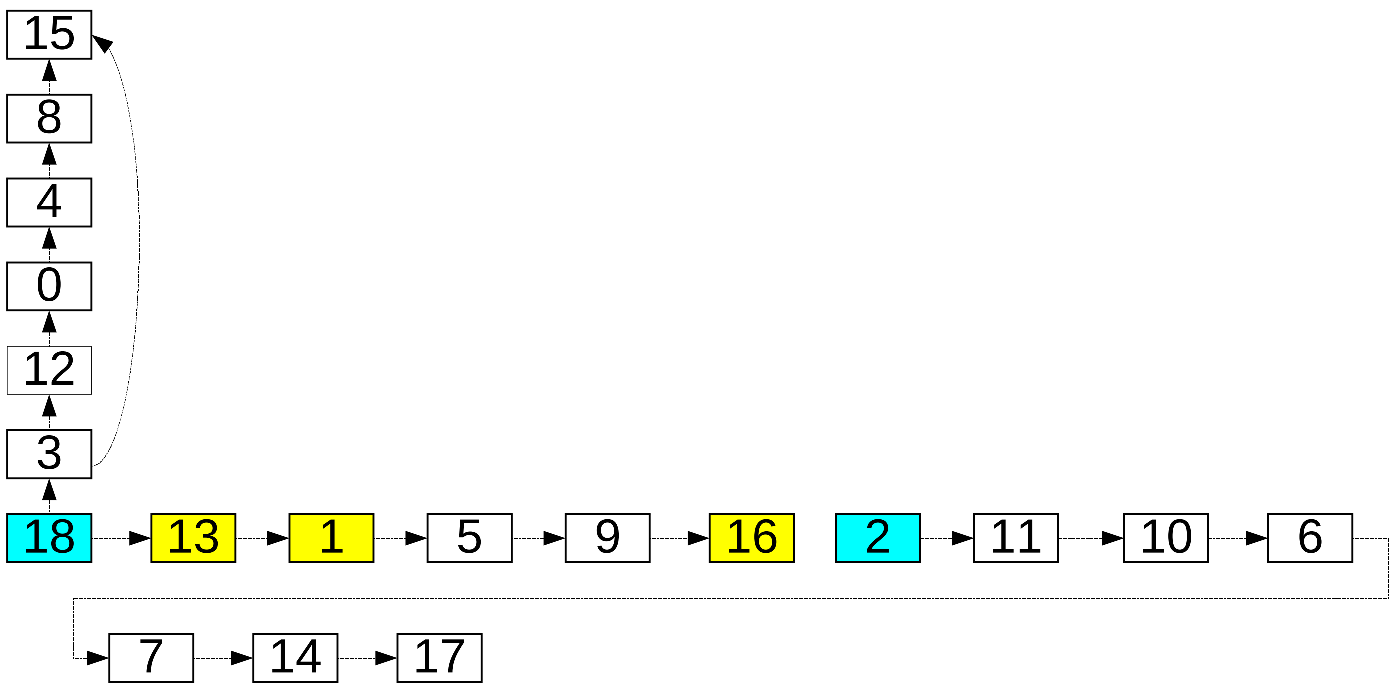


flatten the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	16	6	10	0	1	17	Ø	Ø	Ø	13
Ø	Ø	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	3

tailend

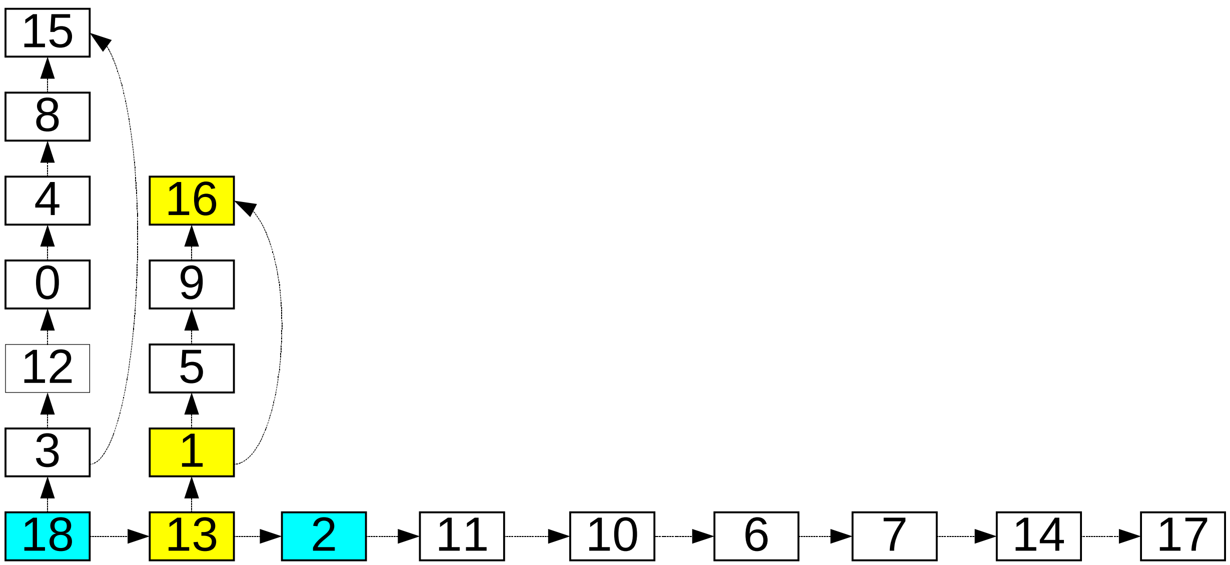
tailnext



verticalize the family

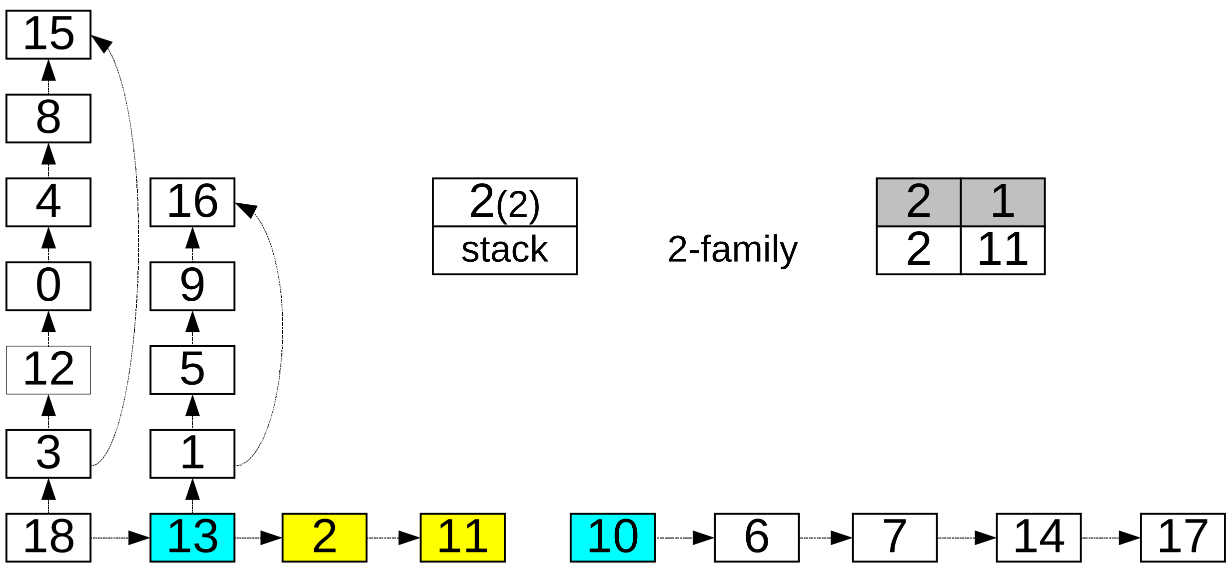
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	16	6	10	0	2	17	Ø	Ø	Ø	13
Ø	16	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3

tailend tailend tailnext tailnext



identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	16	6	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend		tailend										tailnext					tailnext



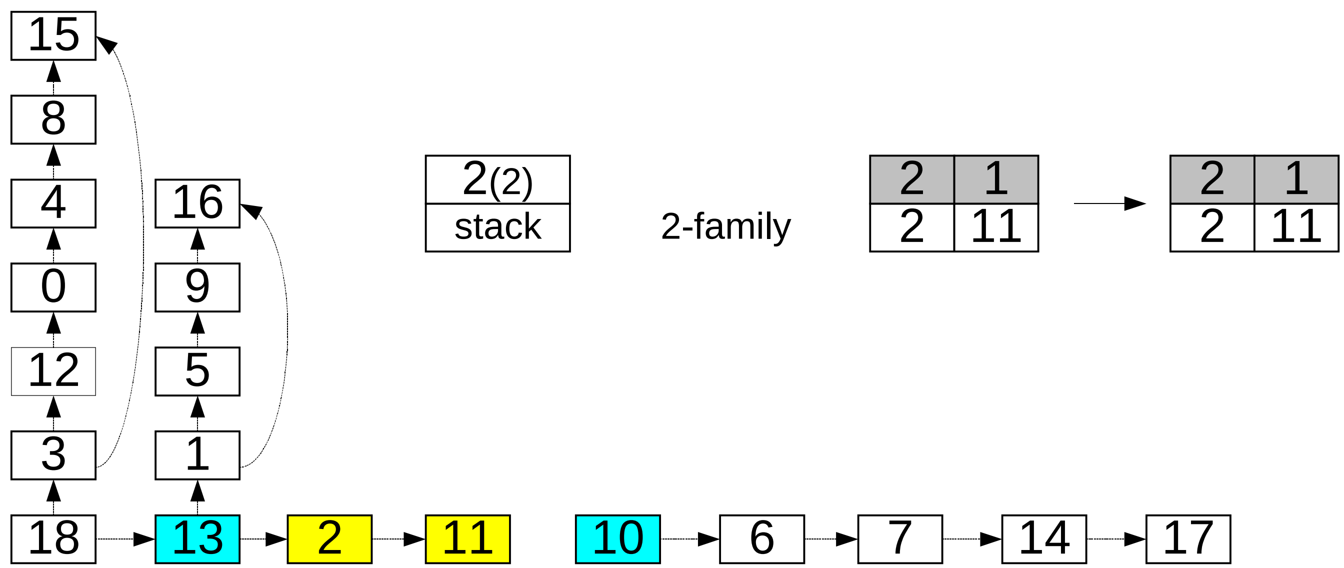
2(2)
stack

2-family

2	1
2	11

sort the family

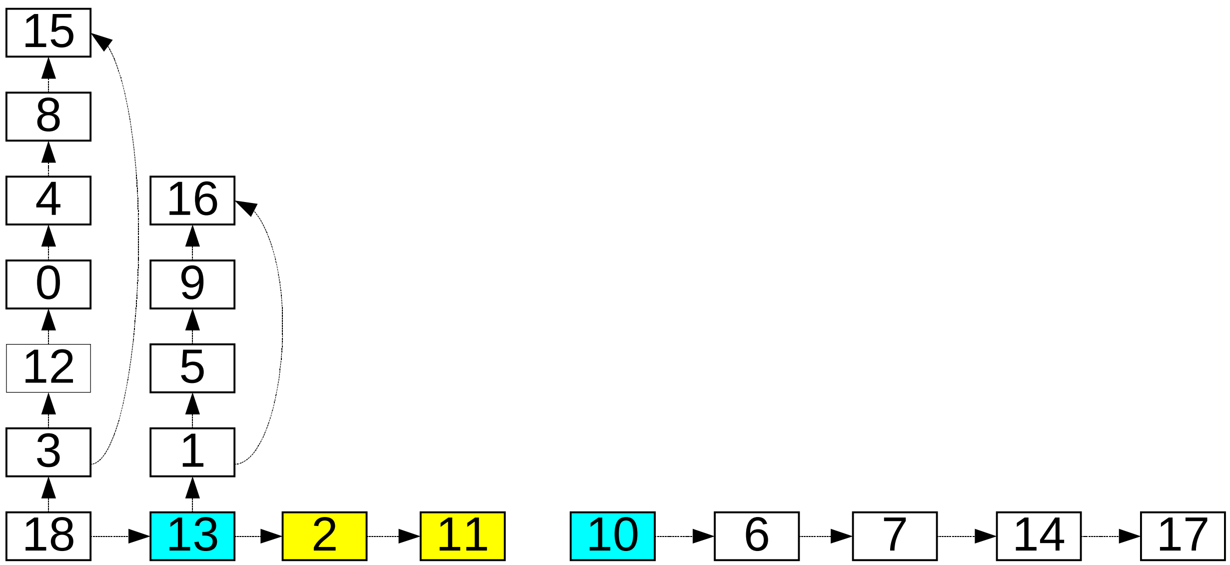
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	16	6	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend		tailend										tailnext					tailnext



flatten the family

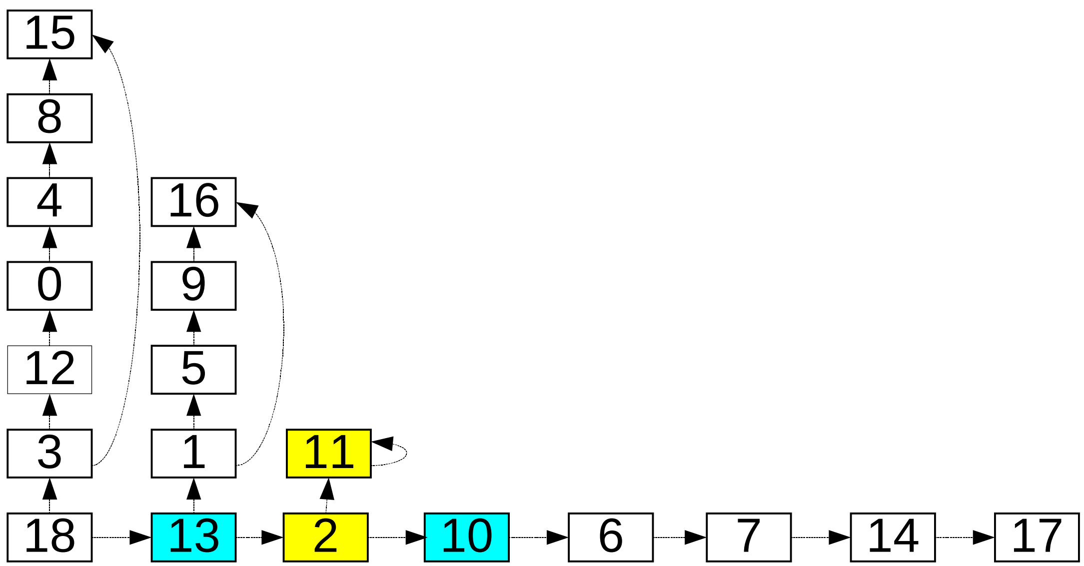
nothing to flatten, it is already flat

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	7	14	15	16	6	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend		tailend										tailnext					tailnext



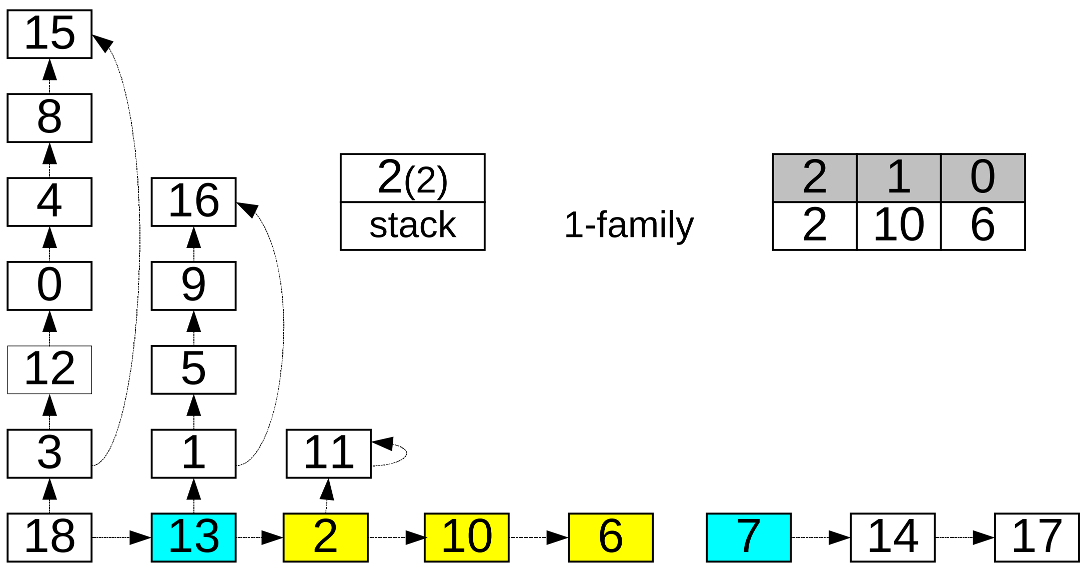
verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	10	12	8	9	7	14	15	16	6	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	0	3	2	1	1	1	2	1	0	0	0	Ø	1
4	5	10	12	8	9	Ø	14	15	16	6	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



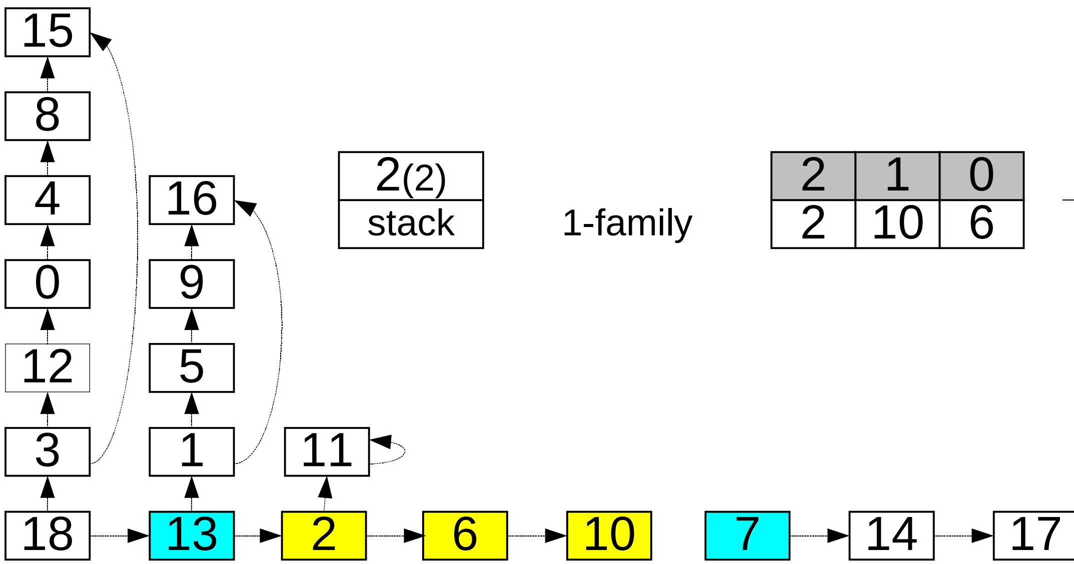
2(2)
stack

1-family

2	1	0
2	10	6

sort the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	1	0	2	1	0	0	0	Ø	1
4	5	6	12	8	9	10	14	15	16	Ø	Ø	0	2	17	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



2(2)
stack

1-family

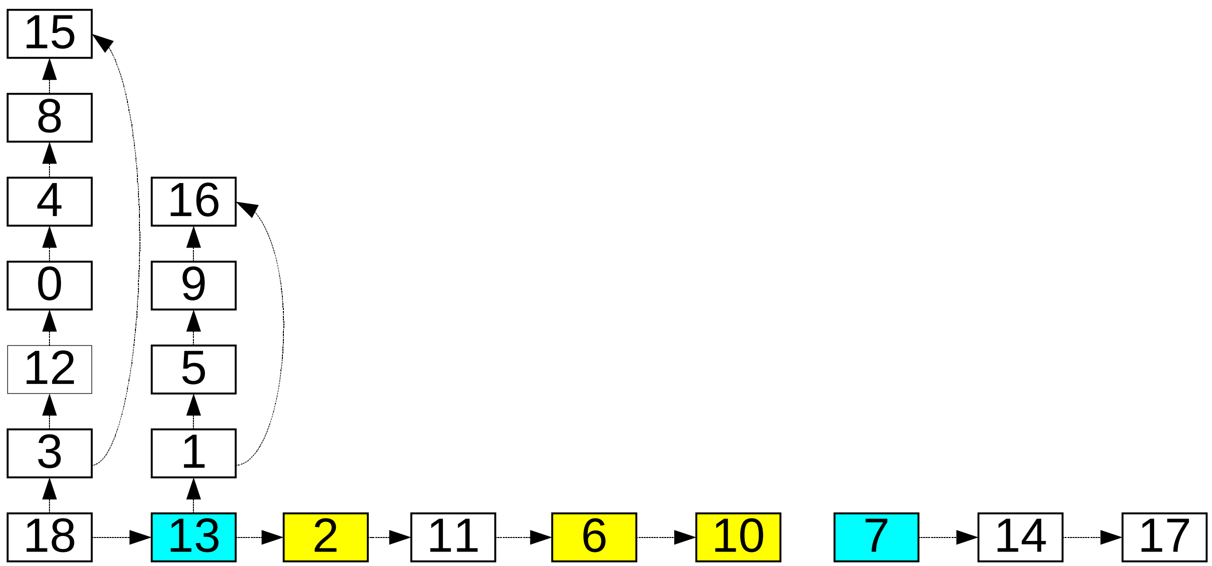
2	1	0
2	10	6

→

2	1	0
2	6	10

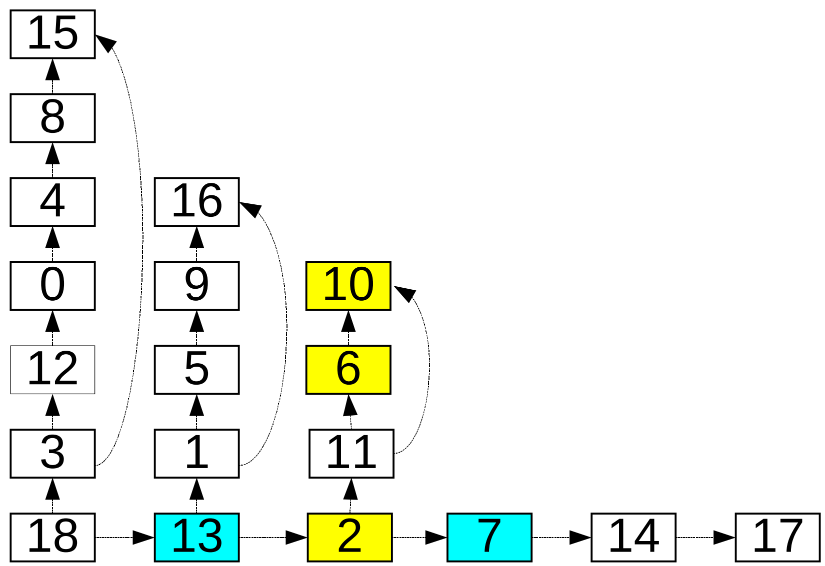
flatten the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	10	14	15	16	Ø	6	0	2	17	Ø	Ø	Ø	13
Ø	16	Ø	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	3
	tailend		tailend										tailnext					tailnext



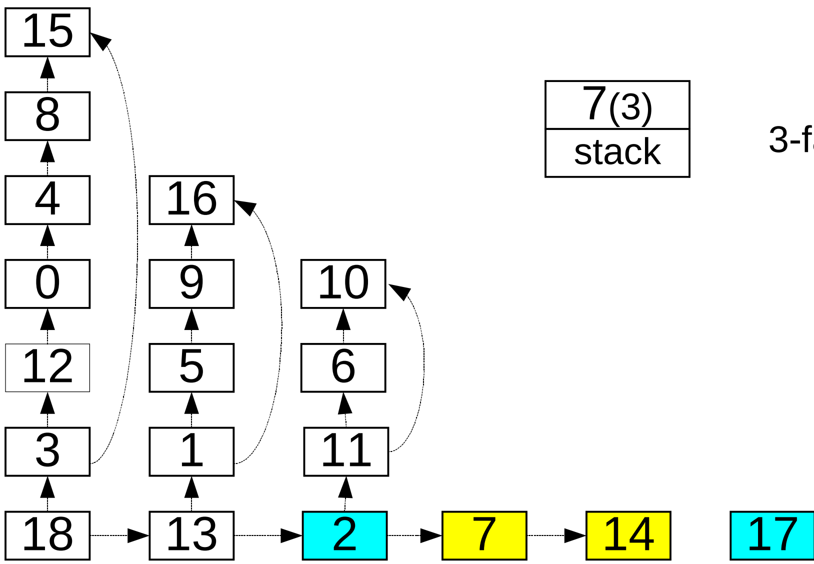
verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	1	0	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	14	15	16	Ø	6	0	2	17	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	14	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext

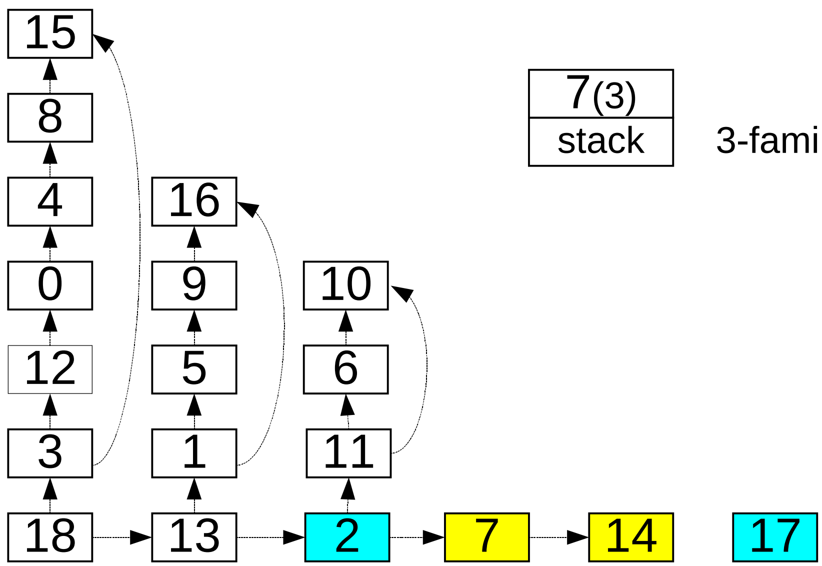


3-family

3	0
7	14

sort the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	14	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



7(3)
stack

3-family

3	0
7	14

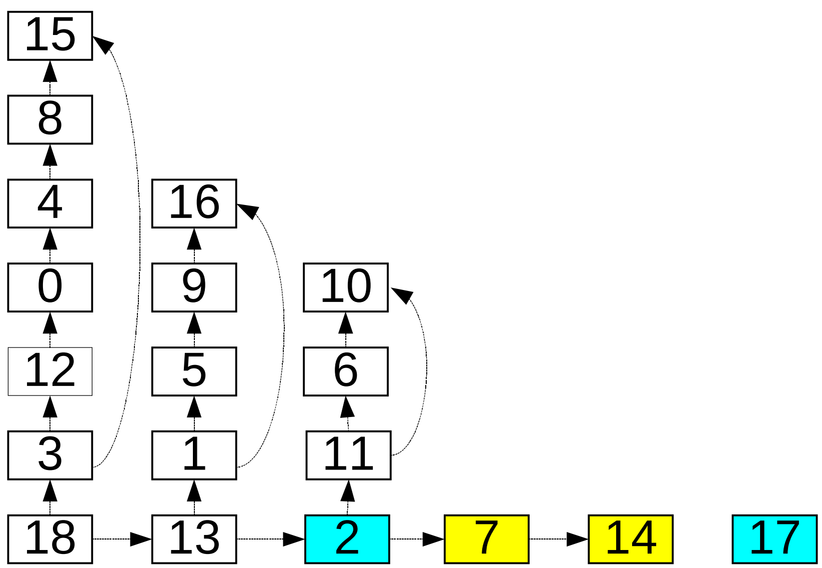
→

3	0
7	14

flatten the family

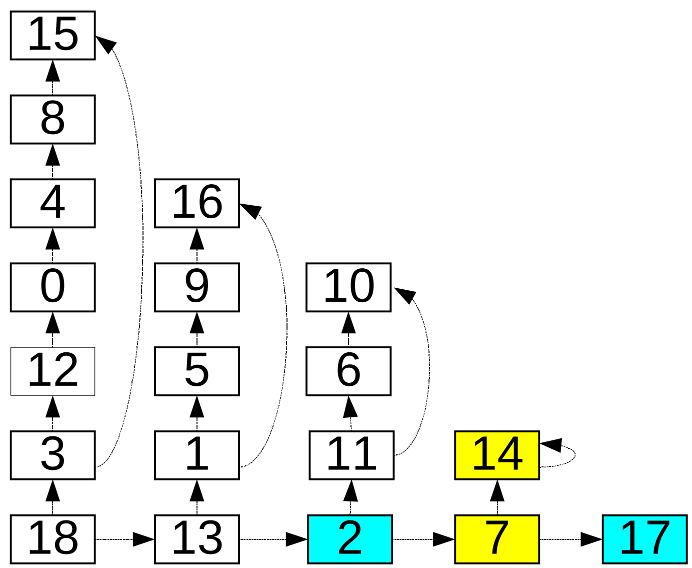
nothing to flatten, it is already flat

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	14	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend								tailend		tailnext					tailnext



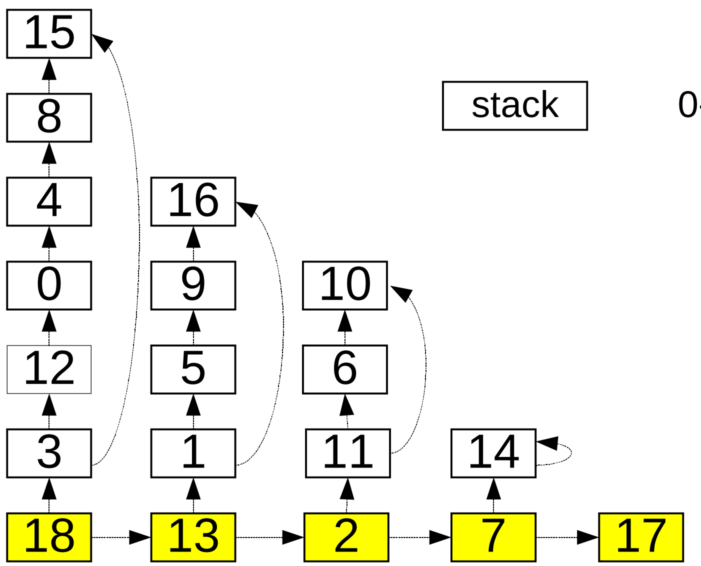
verticalize the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	17	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	14	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend				tailnext				tailend		tailnext	tailend				tailnext



identify and extract family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	17	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	14	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend				tailnext				tailend		tailnext	tailend				tailnext



stack

0-family

1	1	2	3	Ø
18	13	2	7	17

sort the family

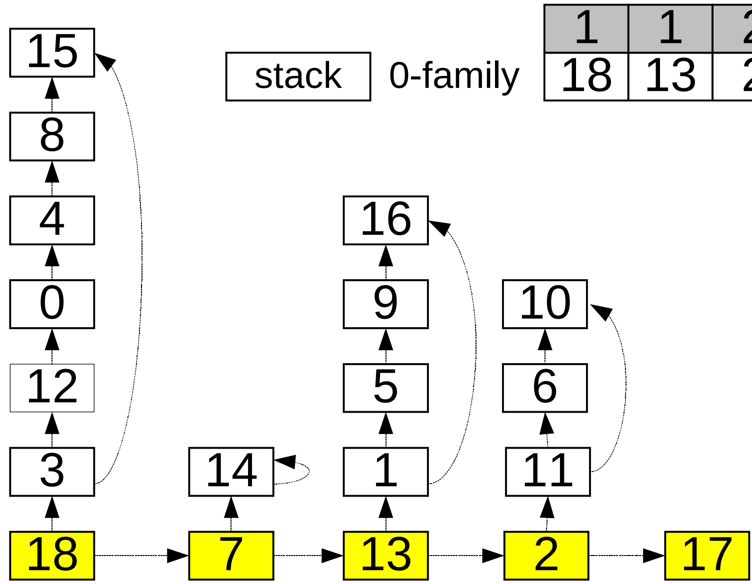
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	7	12	8	9	10	17	15	16	Ø	6	0	2	Ø	Ø	Ø	Ø	13
Ø	16	11	15	Ø	Ø	Ø	14	Ø	Ø	Ø	10	Ø	1	Ø	Ø	Ø	Ø	3
	tailend	tailnext	tailend				tailnext				tailend		tailnext	tailend				tailnext

stack 0-family

1	1	2	3	Ø
18	13	2	7	17

→

1	3	1	2	Ø
18	7	13	2	17



flatten the family

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	10	14	15	16	17	6	0	1	13	7	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 12 → 0 → 4 → 8 → 15 → 7 → 14 → 13 →

→ 1 → 5 → 9 → 16 → 2 → 11 → 6 → 10 → 17

stop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	2	2	1	3	2	1	3	2	1	0	1	2	1	0	0	0	Ø	1
4	5	11	12	8	9	10	14	15	16	17	6	0	1	13	7	2	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	Ø	Ø	Ø	Ø	Ø	Ø	Ø

18 → 3 → 12 → 0 → 4 → 8 → 15 → 7 → 14 → 13 → 1 → 5 → 9 → 16 → 2 → 11 → 6 → 10 → 17

transform to suffix array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
18	3	12	0	4	8	15	7	14	13	1	5	9	16	2	11	6	10	17
	1	1	2	3	3	2	0	3	0	1	2	2	1	0	2	1	1	0

The algorithm just presented shows yet again, that for most of applications suffix array is as efficient as suffix tree, yet requiring significantly less memory.

So, when the resorting process is linear?

1. Of course, when the alphabet is fixed, then the re-sorting is linear.

2. When the permutation is not too complex, then the resorting will also be linear.

Let us introduce the suborder complexity β of a permutation $\boldsymbol{p}$ of length N: $\beta(\boldsymbol{p}) = \boldsymbol{min}\ \beta$ *so that for any* $2 \le \boldsymbol{k} \le N$, *it takes at most* $\beta \boldsymbol{k}$ *steps to order any subset of* N *of size* $\boldsymbol{k}$.

Note: $\beta(\boldsymbol{p}) \le \boldsymbol{log}\ N$ as any subset of N of size $\boldsymbol{k}$ can be sorted in $\le \boldsymbol{k}\ \boldsymbol{log}\ \boldsymbol{k}$ steps and $\boldsymbol{k}\ \boldsymbol{log}\ \boldsymbol{k} \le \boldsymbol{k}\ \boldsymbol{log}\ N$

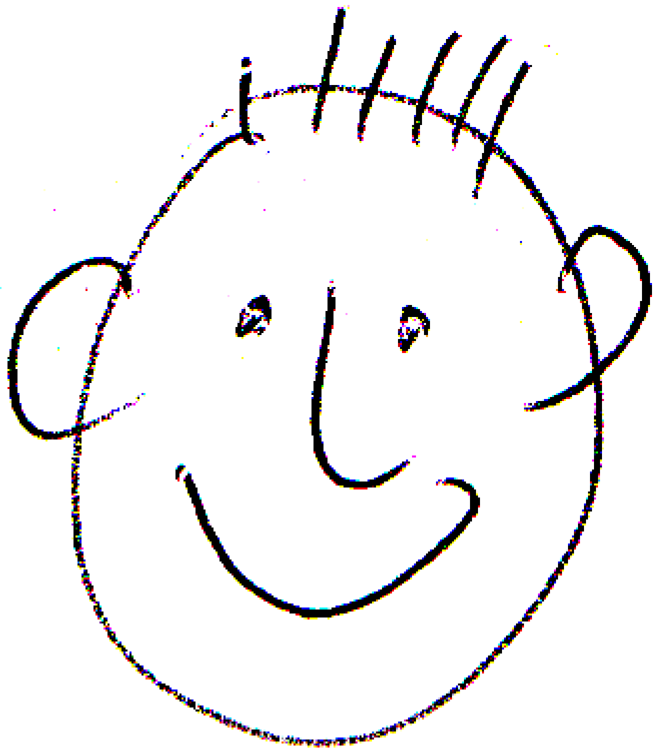
For any permutation with suborder complexity β, the suffix array of a string can be re-ordered in a $O(\beta N)$ time, where N is the length of the input string.

- For instance, the inversion has suborder complexity of 1.
- Any rotation has suborder complexity of 1.
- Any permutation with β transpositions has suborder complexity of β
- Let $\boldsymbol{p}$ be a "mild" permutation, i.e. $|\boldsymbol{p}(\boldsymbol{i}) - \boldsymbol{i}| \leq \beta$. Then $\boldsymbol{p}$ has suborder complexity of 2β.
- Let $\boldsymbol{p}_1$ on $\boldsymbol{N}_1$ have suborder complexity β_1 and let $\boldsymbol{p}_2$ on $\boldsymbol{N}_2$ have suborder complexity β_2, then $\boldsymbol{p}_1 \oplus \boldsymbol{p}_2$ will have suborder complexity $max(\beta_1,\beta_2)$.

So, there are quite a few of permutations that allow us to re-sort the suffix array or the suffix tree of a string in linear time.
May be, it is of independent interest to study the suborder complexity of permutations.

It will be also interesting to see, if it is more efficient to simply re-sort the suffixes, or if re-sorting it our way is more efficient. If our approach turns out to be more efficient, it may be conceivable to compute efficiently a suffix array according to some other order of the alphabet more conducive to the task and then re-sort it according to the natural order.

This is one of the possibilities we will be pursuing in our quest for a non-recursive linear-time and memory efficient algorithm to sort suffixes.



http://www.cas.mcmaster.ca/~franek