Simpler and Faster Lempel Ziv Factorization

Keisuke Goto and Hideo Bannai
Kyushu University

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

Example

$T =$ | f_1 a | f_2 b | f_3 a | f_4 a b a | f_5 b a b a | f_6 a a a | f_7 b | f_8 b a b a b |

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

Example

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

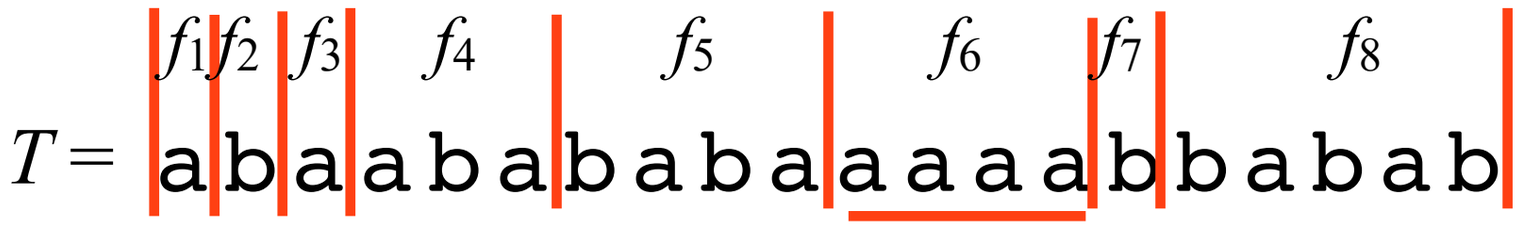
Example

$T =$ | f_1 | f_2 | f_3 | f_4 | f_5 | f_6 | f_7 | f_8 |
a b a a b a b a b a a a a a b b a b a b

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

Example



LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

Example

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

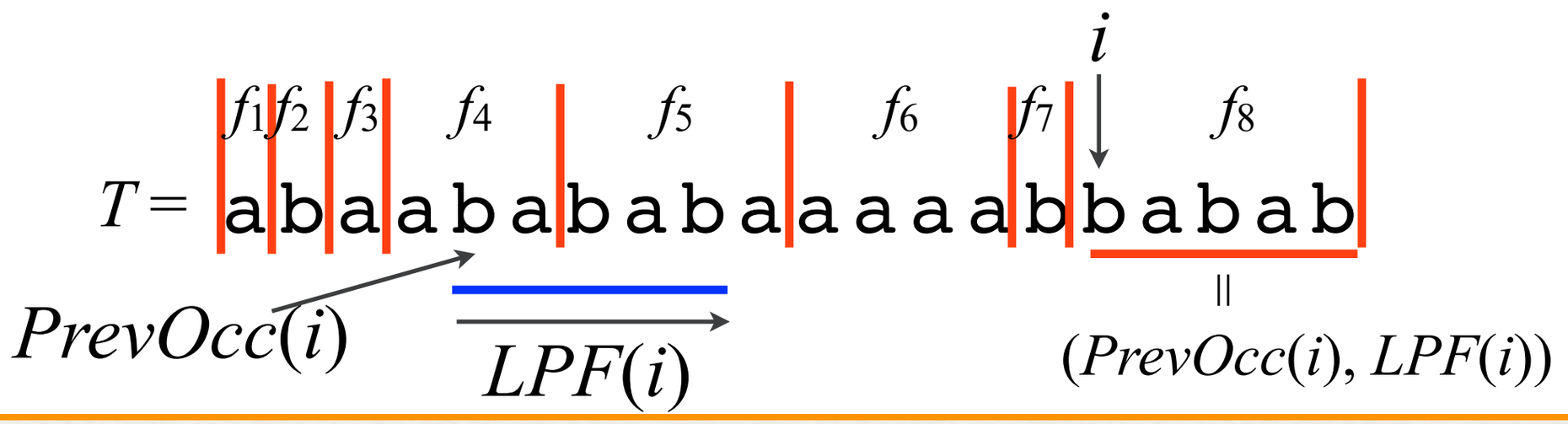
Example

$T =$ | f_1 a | f_2 b | f_3 a | f_4 a b a | f_5 b a b a | f_6 a a a | f_7 b | f_8 b a b a b |

LZ77 Factorization

- The LZ77 factorization of string T of size N is a factorization $T = f_1 f_2 \ldots f_z$ such that for $1 \leqq j \leqq z$
- For a factor f_j which starts at position i,
  - f_j is a single character $T[i]$ that does not occur before or
  - f_j is the longest prefix of $T[i \ldots N]$ that has at least one previous occurrence

We can represent each factor as a pair $(PrevOcc(i), LPF(i))$ where i is the position at which the factor starts



Previous Work

- Most recent LZ factorization algorithms use suffix array and additional arrays such as *LCP*, *LPF*, *PrevOcc* arrays.

algorithm	worst case	*SA*	*LCP*	*LPF, PrevOcc*	
Naive	$\Theta(N^2)$				
CI1	$\Theta(N)$	○		○	**[Crochemore and Ilie, 2011]**
CI2	$\Theta(N)$	○	○	○	**[Crochemore and Ilie, 2011]**
CPS1	$\Theta(N)$	○	○		**[Chen+, 2008]**
CIS	$\Theta(N)$	○	○	○	**[Crochemore+, 2008]**
CII	$\Theta(N)$	○	○	○	**[Crochemore+, 2009]**
LZ_OG	$\Theta(N)$	○		○	**[Ohlebusch and Gog, 2011]**

Algorithm by using *PrevOcc*, *LPF* arrays

f_1

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)

Algorithm by using *PrevOcc*, *LPF* arrays

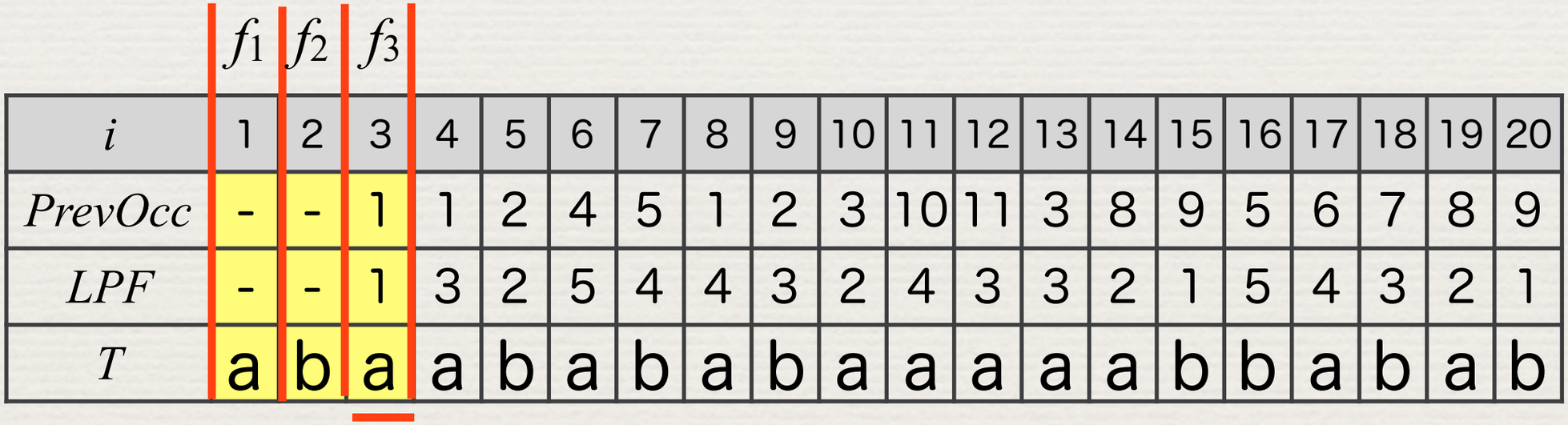
f_1 f_2

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)

(-, b)

Algorithm by using *PrevOcc*, *LPF* arrays



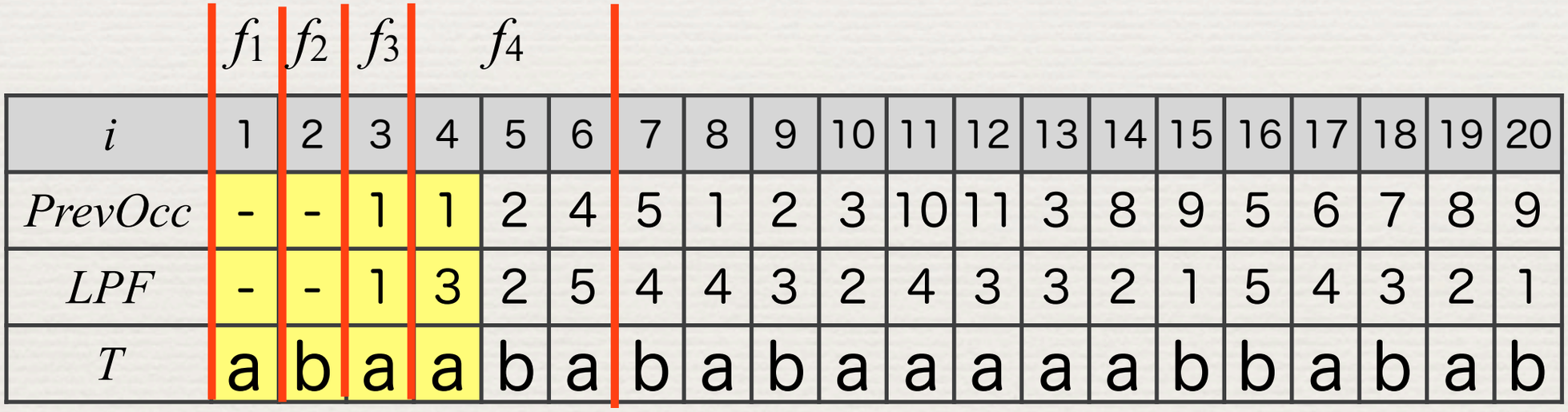
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)

(-, b)

(1, 1)

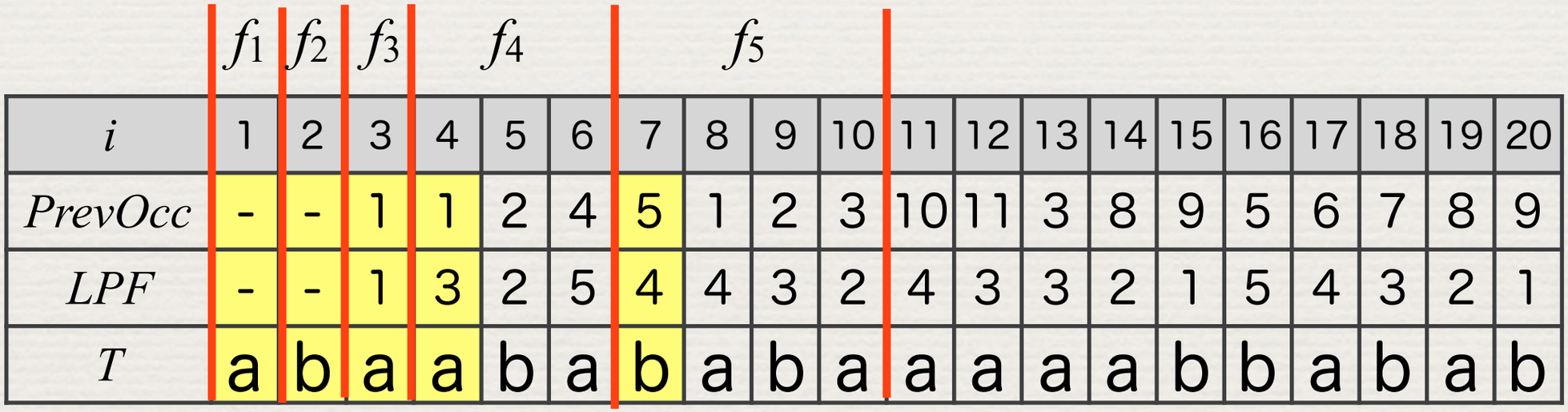
Algorithm by using *PrevOcc*, *LPF* arrays



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)
(-, b)
(1, 1)
(1, 3)

Algorithm by using *PrevOcc*, *LPF* arrays



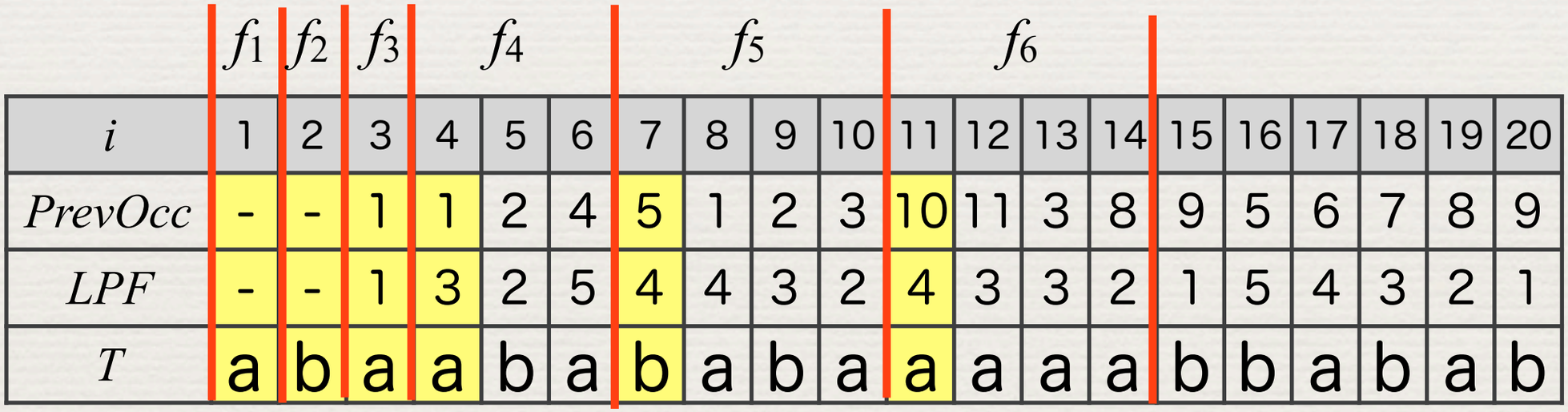
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)　　(1, 3)　(5, 4)

(-, b)

(1, 1)

Algorithm by using *PrevOcc*, *LPF* arrays



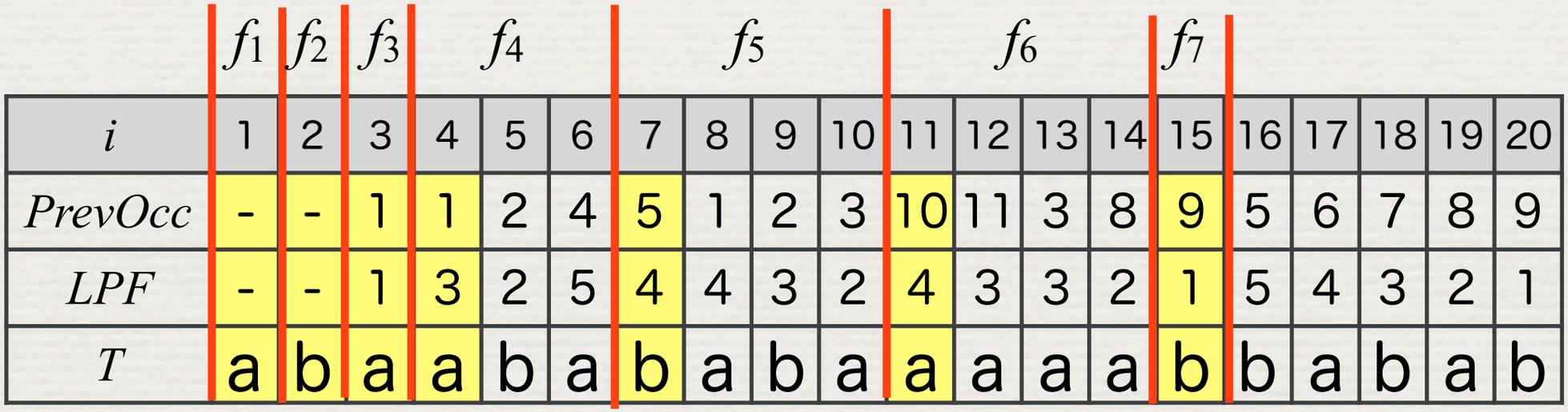
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)　　(1, 3)　(5, 4)　　(10, 4)

(-, b)

(1, 1)

Algorithm by using *PrevOcc*, *LPF* arrays



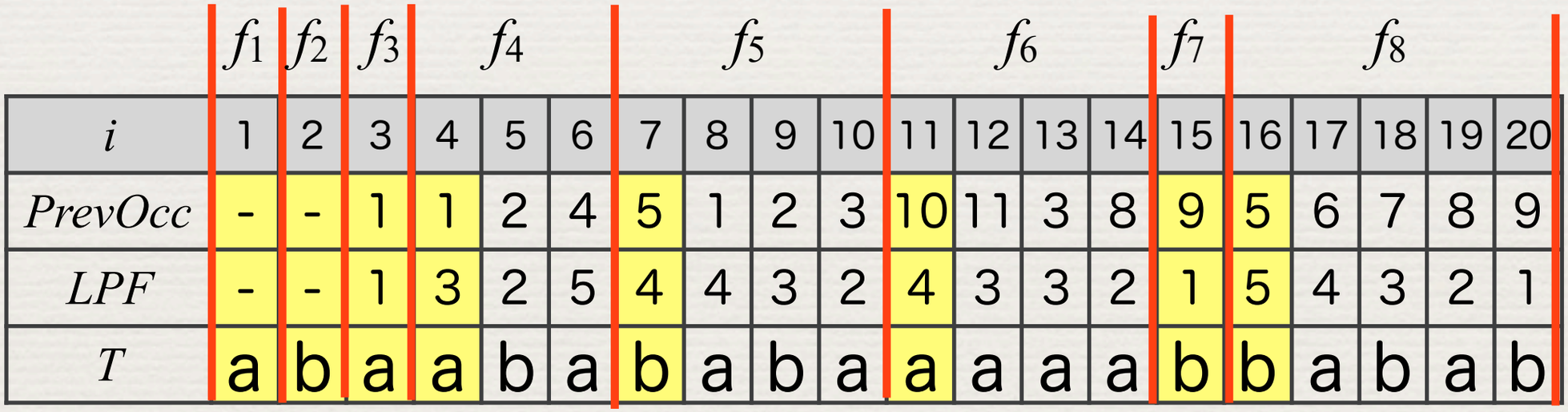
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)　　(1, 3)　(5, 4)　　(10, 4)　　(9, 1)

(-, b)

(1, 1)

Algorithm by using *PrevOcc*, *LPF* arrays



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)
(-, b)
(1, 1)
(1, 3)
(5, 4)
(10, 4)
(9, 1)
(5, 5)

Observation of the algorithm by *PrevOcc*, *LPF*

LPF[*i*], *PrevOcc*[*i*] are not used for these positions

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PrevOcc	-	-	1	1	2	4	5	1	2	3	10	11	3	8	9	5	6	7	8	9
LPF	-	-	1	3	2	5	4	4	3	2	4	3	3	2	1	5	4	3	2	1
T	a	b	a	a	b	a	b	a	b	a	a	a	a	a	b	b	a	b	a	b

(-, a)　(1, 3)　(5, 4)　(10, 4)　(9, 1)

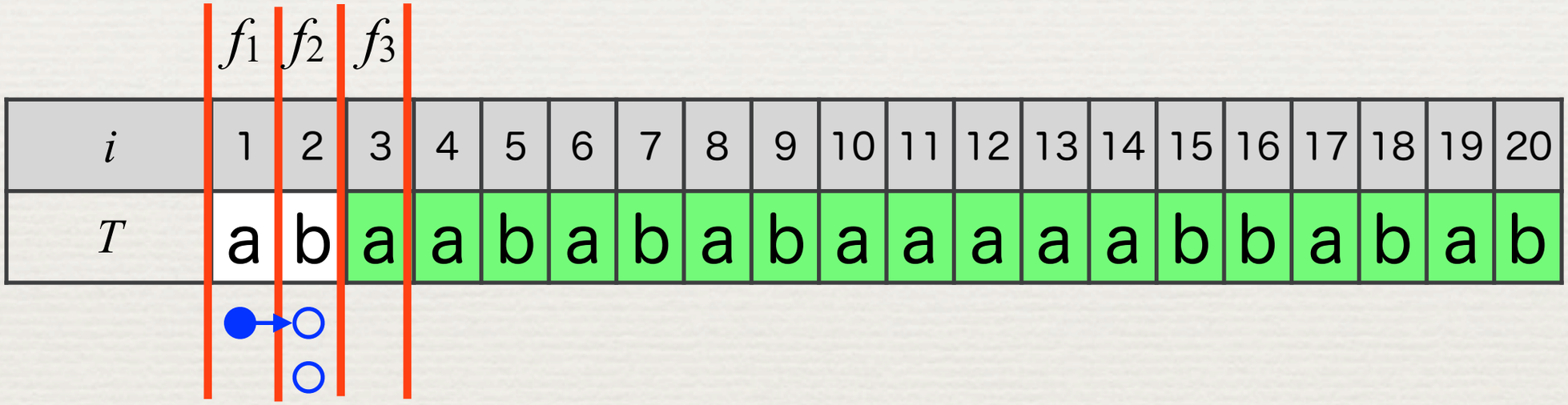
(-, b)　(5, 5)

(1, 1)

Our algorithm only computes *LPF*(*i*), *PrevOcc*(*i*) when required, and avoids construction of *LPF*, *PrevOcc* arrays

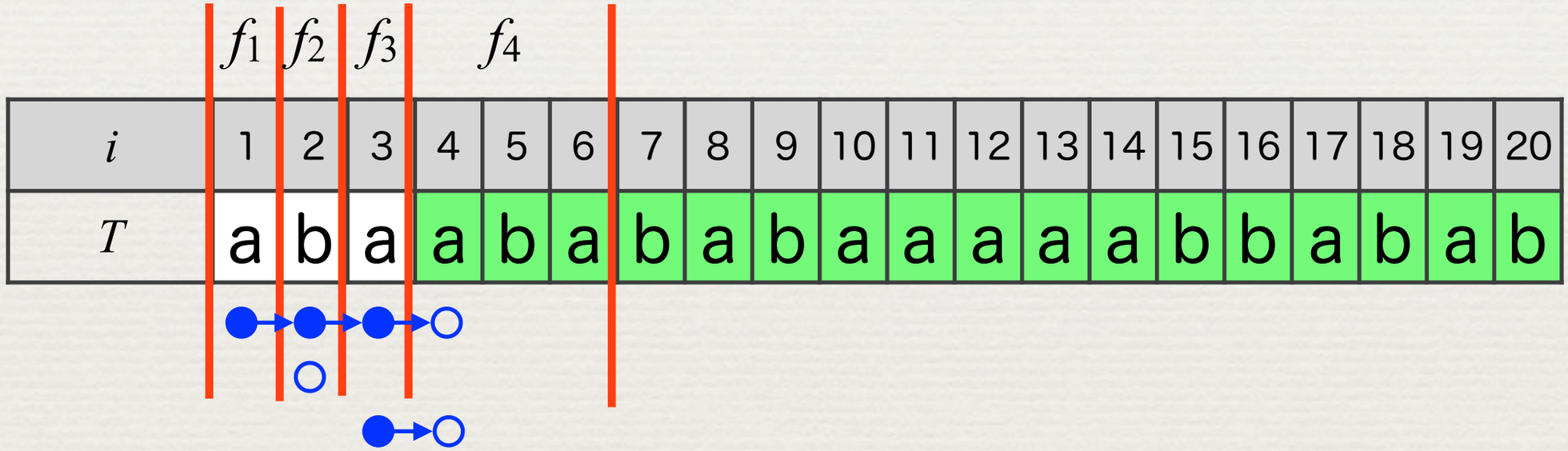
Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



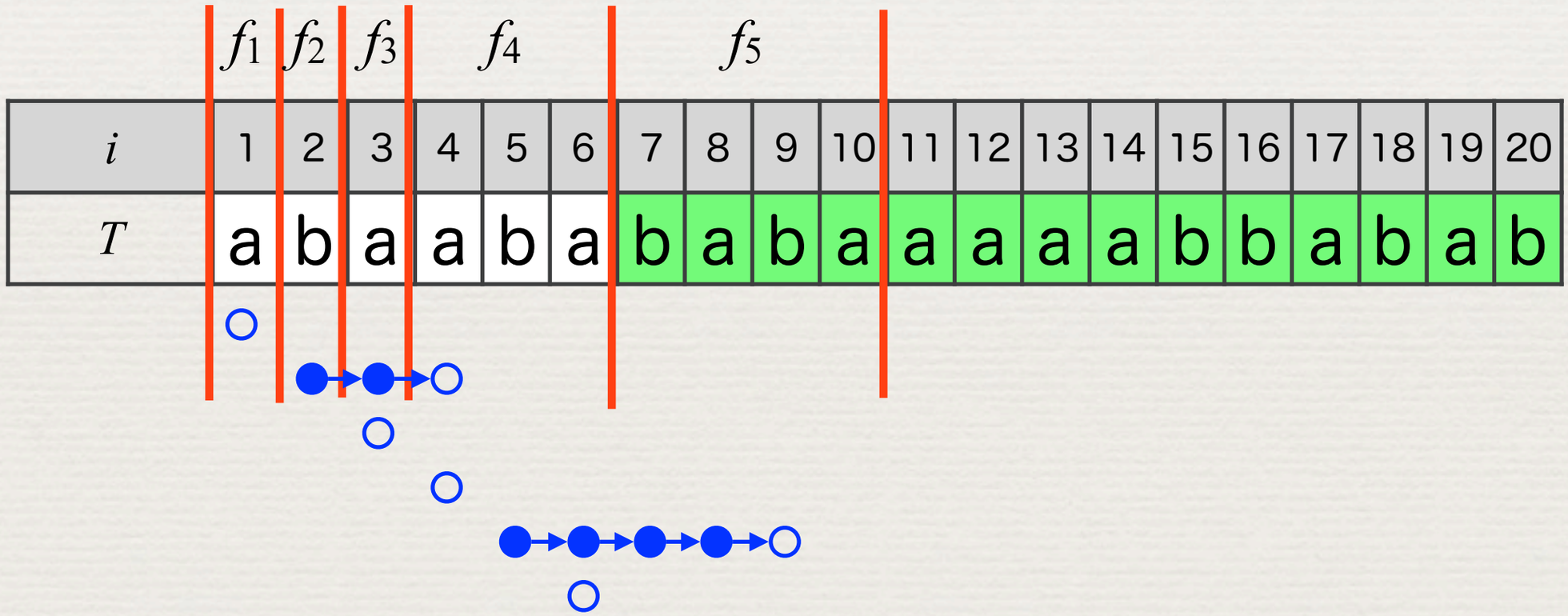
Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



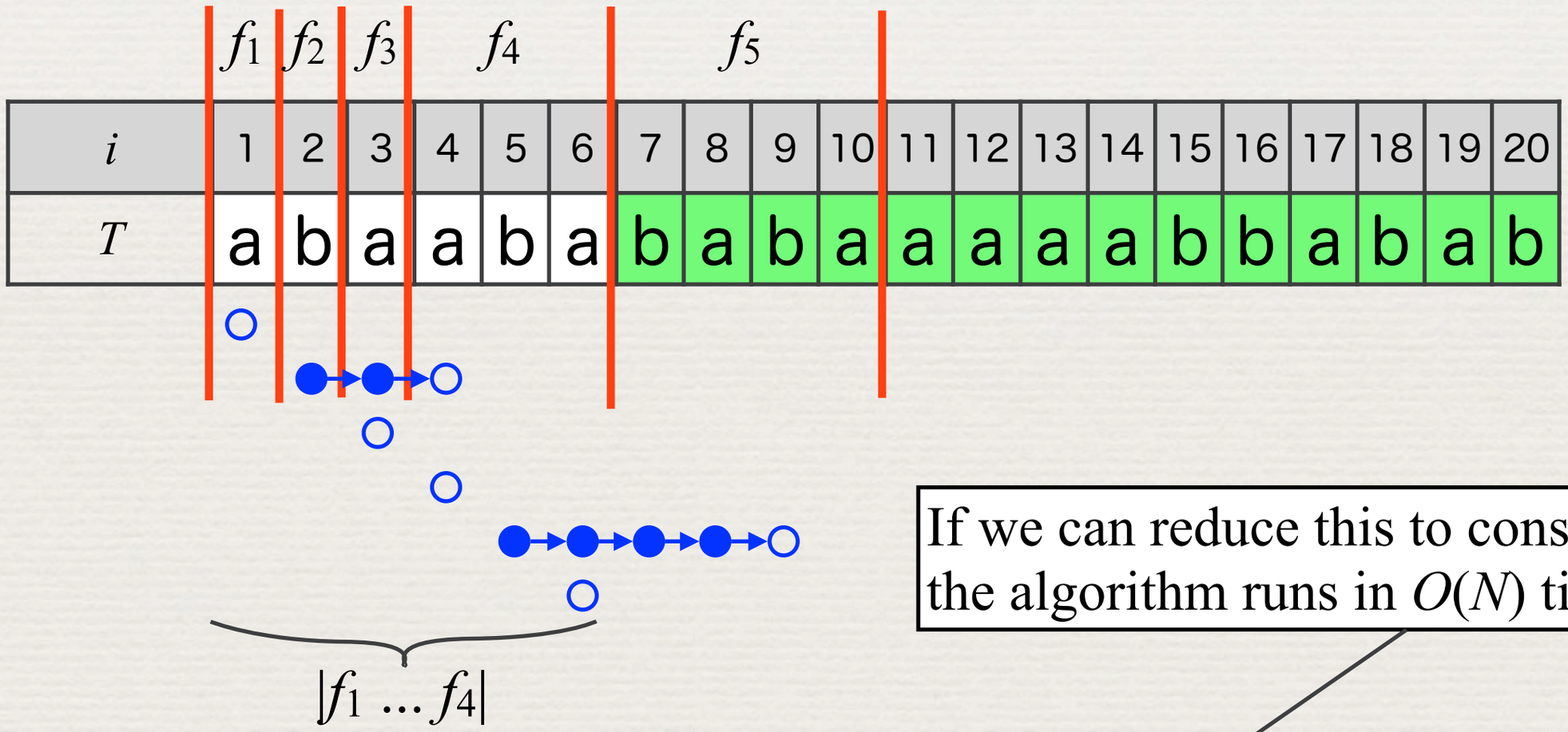
Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



Naive Algorithm

- compute $LPF(i)$, $PrevOcc(i)$ by naive character comparison between $T[i..N]$ and $T[k..N]$ for all $k < i$
- $i = i + LPF(i)$ and repeat



If we can reduce this to constant the algorithm runs in $O(N)$ time

For a factor f_j, naive algorithm needs at most $|f_1 \ldots f_{j-1}| \cdot |f_j|$ comparisons

Total # of comparison is $\Sigma\, |f_1 \ldots f_{j-1}| \cdot |f_j| \leqq N \Sigma\, |f_j| = O(N^2)$

Basic Idea of our algorithm

Lemma [Crochemore and Ilie, 2008]

The candidates of *PrevOcc*(*i*) can be reduced to 2 positions, which are lexicographic predecessor and successor of suffix *i*

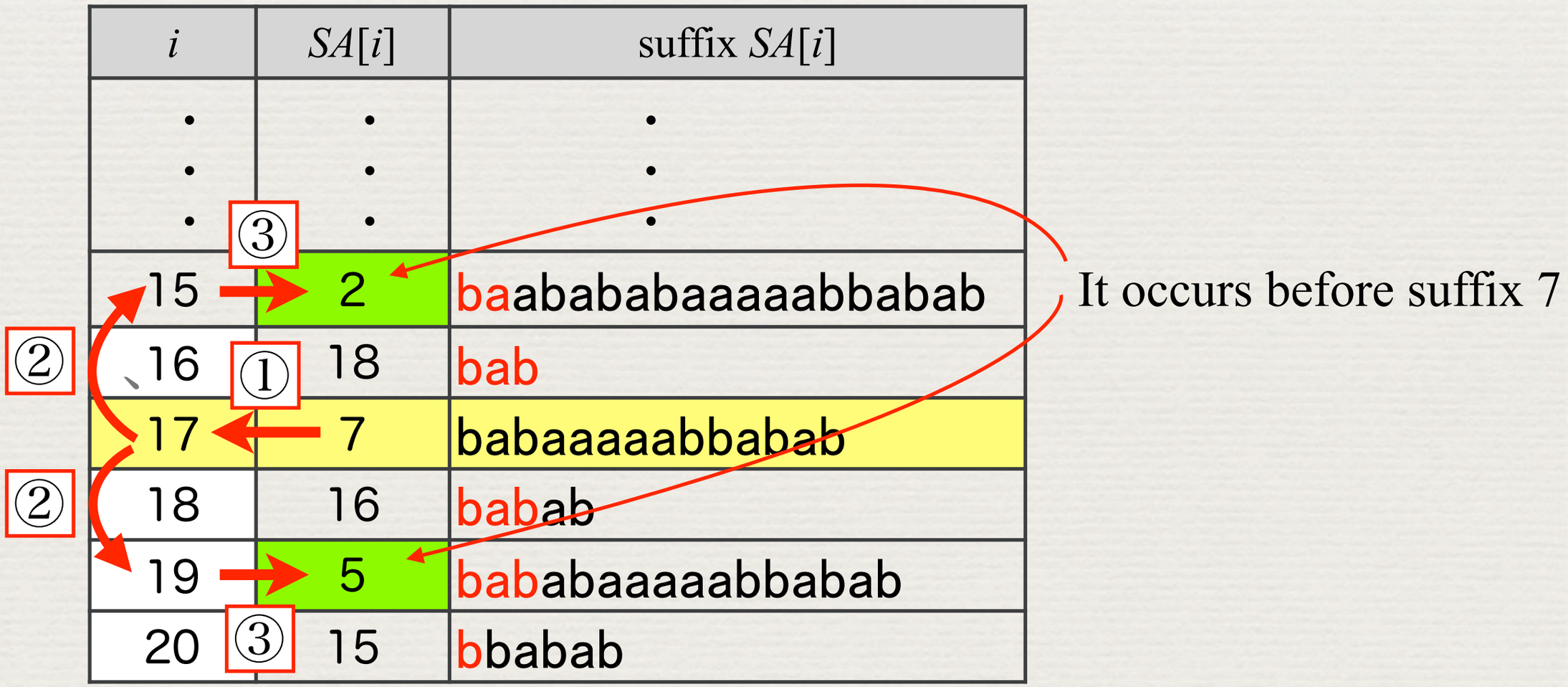
i	*SA*[*i*]	suffix *SA*[*i*]
.	.	.
.	.	.
.	.	.
15	2	baababababaaaaabbbabab
16	18	bab
17	7	babaaaaabbbabab
18	16	babab
19	5	bababaaaaabbbabab
20	15	bbabab

It occurs before suffix 7

Basic Idea of our algorithm

Lemma [Crochemore and Ilie, 2008]

The candidates of *PrevOcc*(*i*) can be reduced to 2 positions, which are lexicographic predecessor and successor of suffix *i*



Basic Idea of our algorithm

Lemma [Crochemore and Ilie, 2008]

$PrevOcc(i)$ is $SA[PSV_{lex}[SA^{-1}[i]]]$ or $SA[NSV_{lex}[SA^{-1}[i]]]$

$$PSV_{lex}[i] = \max\{j \mid j < i, SA[j] < SA[i]\}$$
$$NSV_{lex}[i] = \min\{j \mid j > i, SA[j] < SA[i]\}$$

i	$SA[i]$	suffix $SA[i]$
⋮	⋮	⋮
15	2	baababababaaaaabbbabab
16	8	bab
17	7	babaaaaabbbabab
18	16	babab
19	5	bababaaaaabbbabab
20	5	bbabab

$SA[15]$

$PSV_{lex}[17]$

$SA^{-1}[7]$

$NSV_{lex}[17]$

$SA[19]$

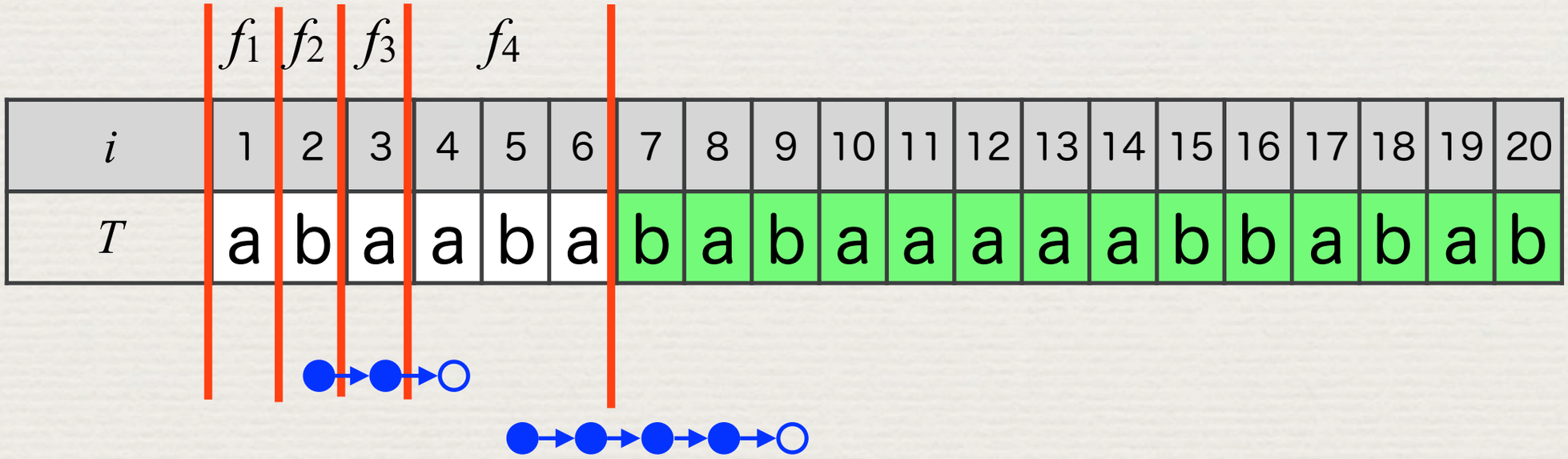
It occurs before suffix 7

Basic Idea of our algorithm

Lemma [Crochemore and Ilie, 2008]

$PrevOcc(i)$ is $SA[PSV_{lex}[SA^{-1}[i]]]$ or $SA[NSV_{lex}[SA^{-1}[i]]]$

$$PSV_{lex}[i] = \max\{j \mid j < i, SA[j] < SA[i]\}$$
$$NSV_{lex}[i] = \min\{j \mid j > i, SA[j] < SA[i]\}$$



For a factor f_j, the number of comparison is $2\,|f_j|$

Total # of comparison is $\Sigma\ 2\,|f_j| = O(N)$

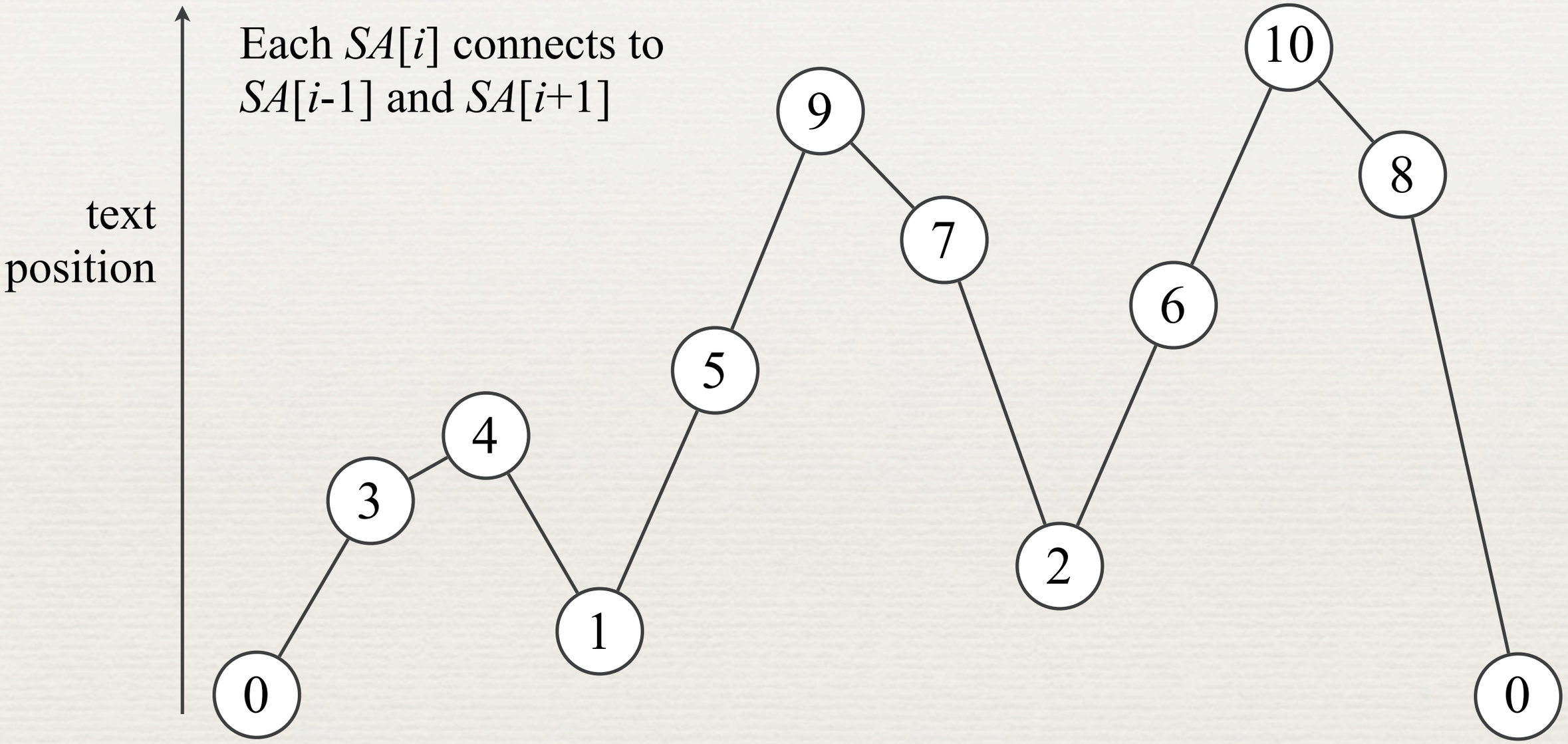
Compute *PSV*, *NSV* arrays

- We propose 3 variations of algorithm to compute *PSV*, *NSV* arrays

Algorithm	Computation of *PSV*, *NSV*	Order of $\perp$ SEP *PSV*, *NSV*	Space
BGS	stack	lex	$17N + 4\|S\|$ bytes
BGL	no stack	lex	$17N$ bytes
BGT	no stack	text	$13N$ bytes

$|S|$: stack size (in worst case $|S| = N$)

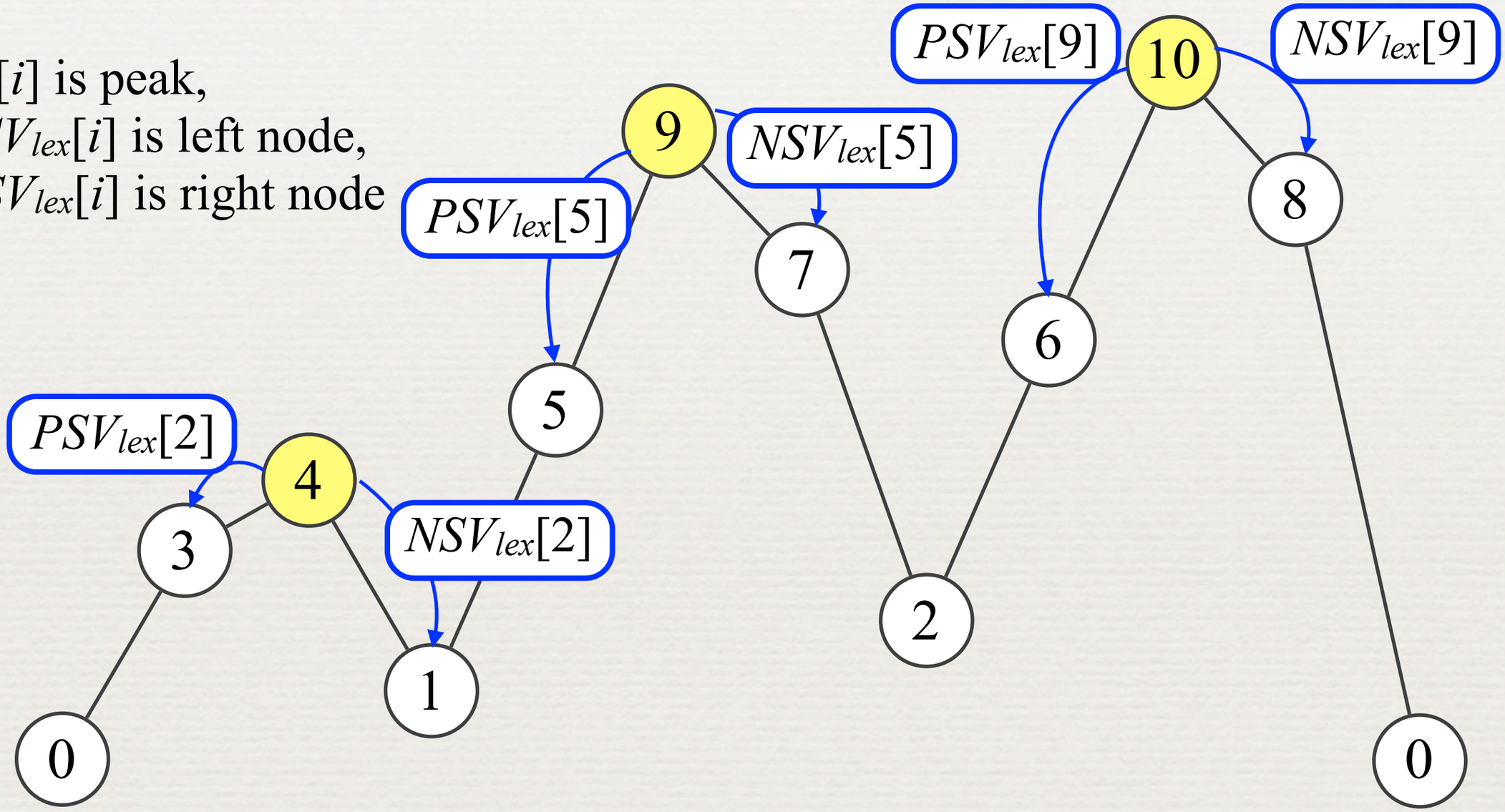
Peak Elimination (Crochemore and Ilie)



i	0	1	2	3	4	5	6	7	8	9	10	11
SA	0	3	4	1	5	9	7	2	6	10	8	0

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

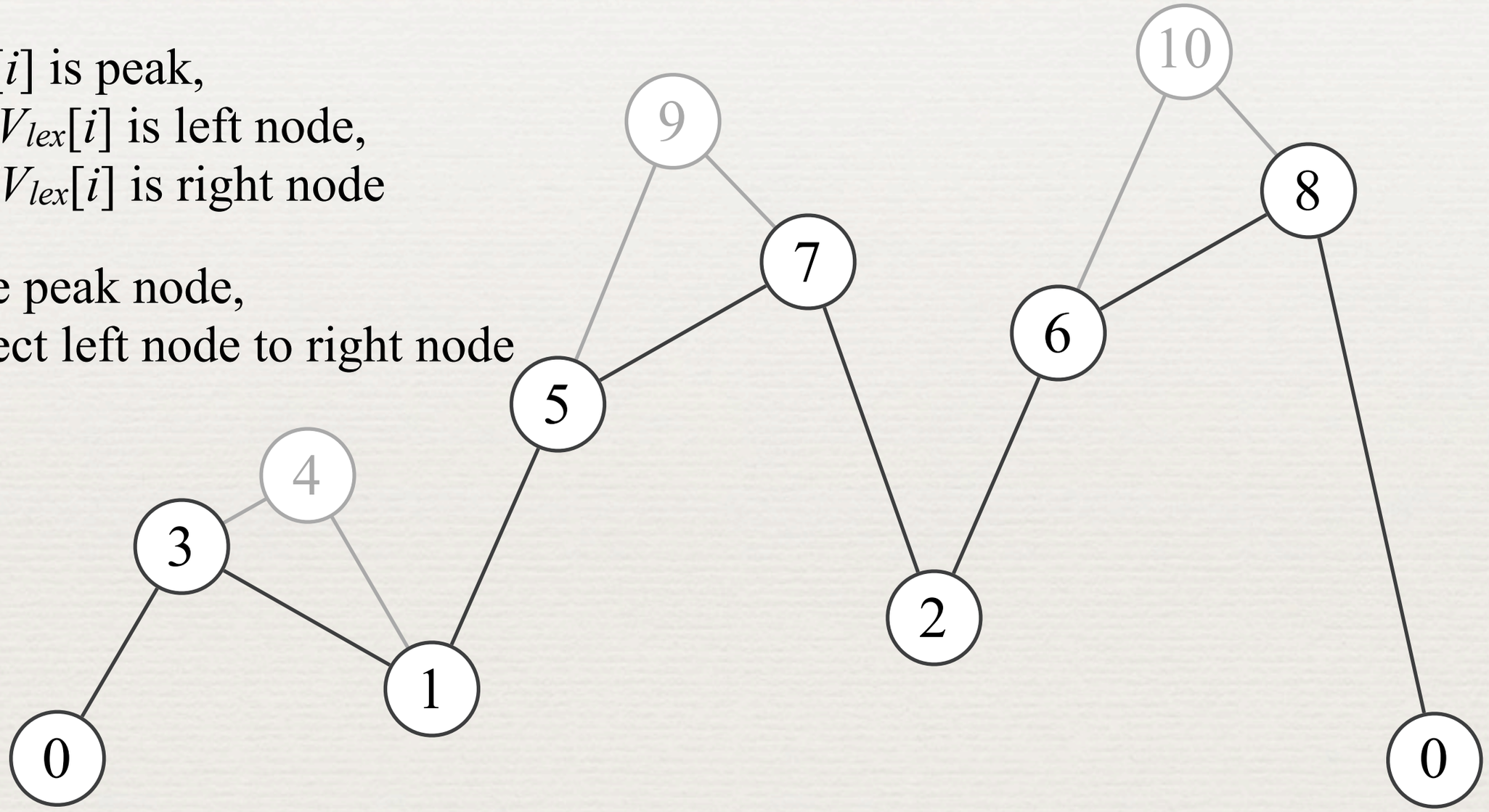


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}			1			4				8		
NSV_{lex}			3			6				10		

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node

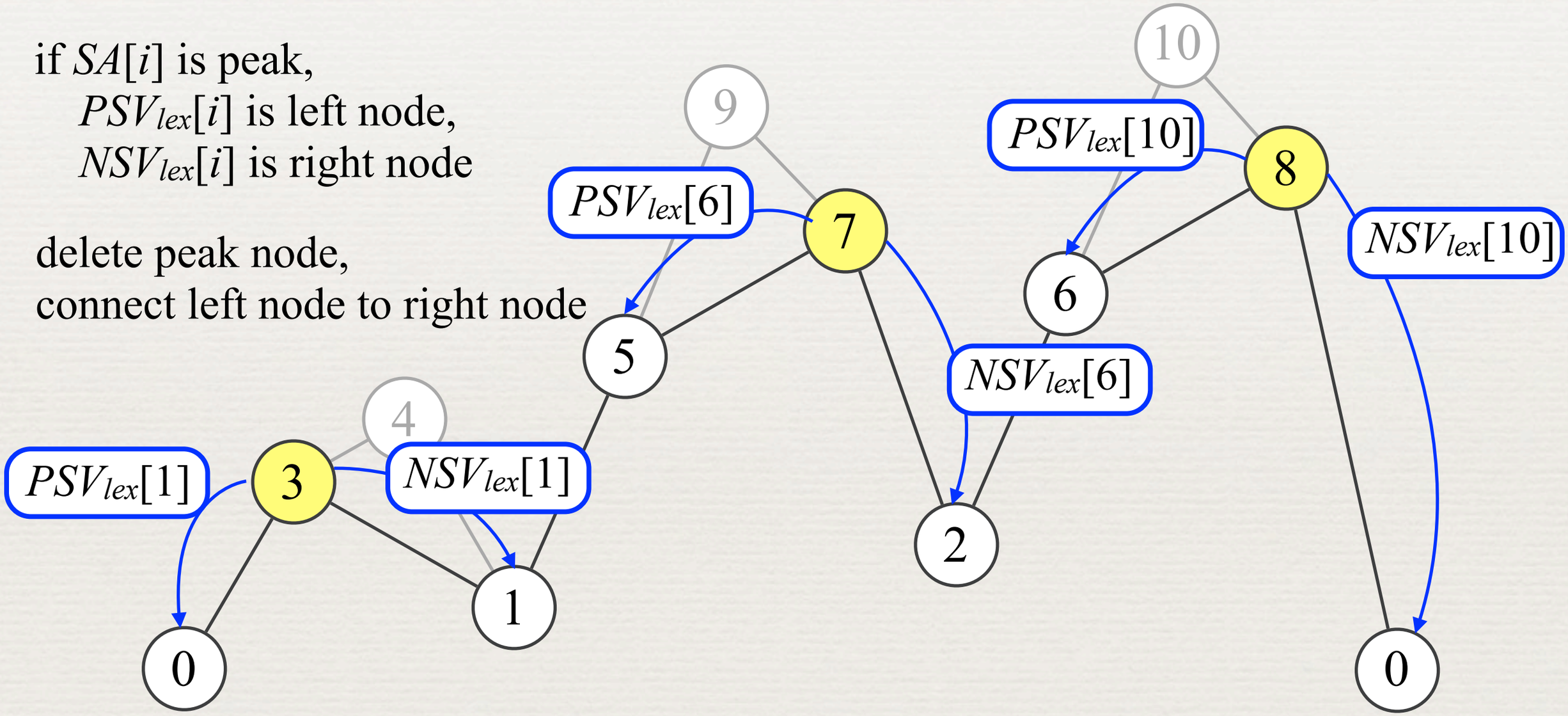


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}			1			4				8		
NSV_{lex}			3			6				10		

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node

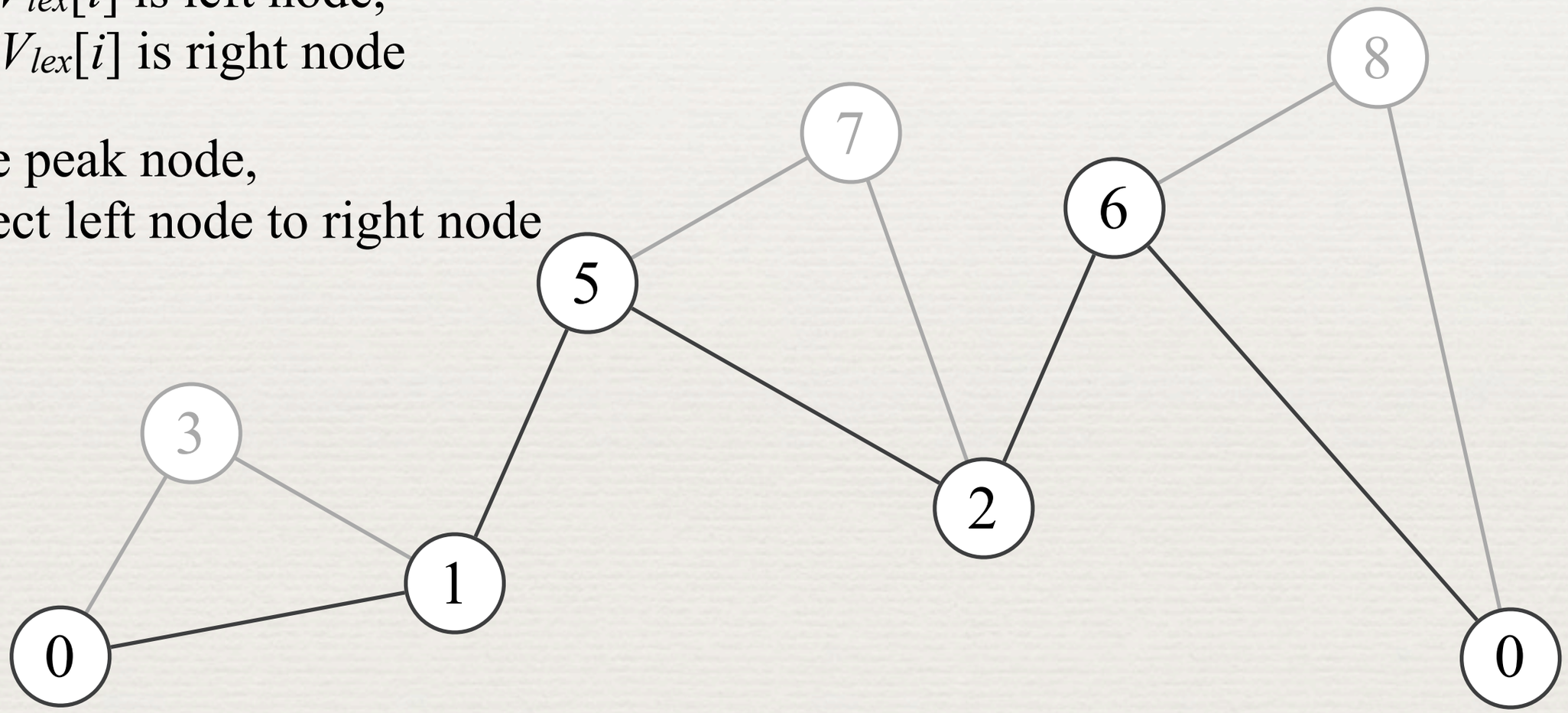


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1			4	4			8	8	
NSV_{lex}		3	3			6	7			10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node

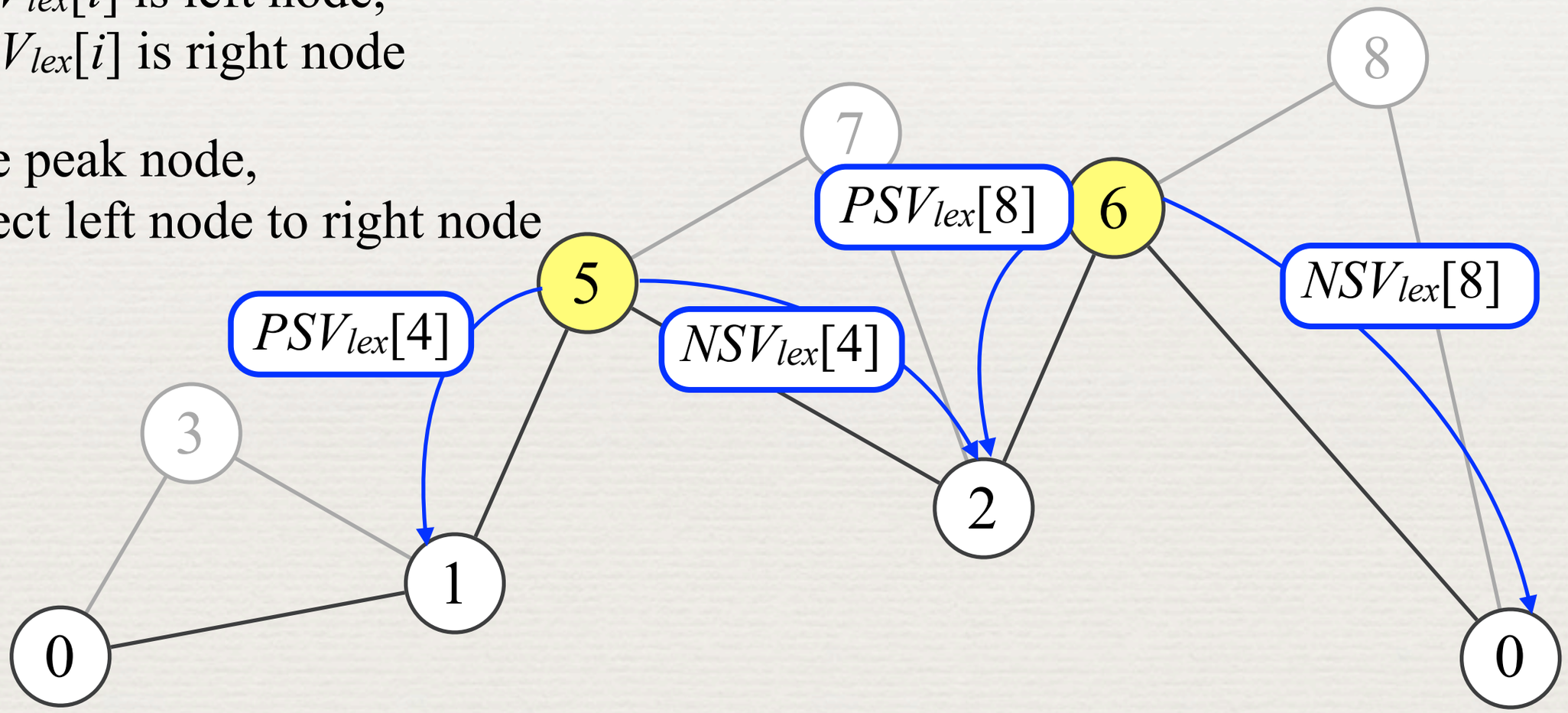


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1			4	4			8	8	
NSV_{lex}		3	3			6	7			10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node

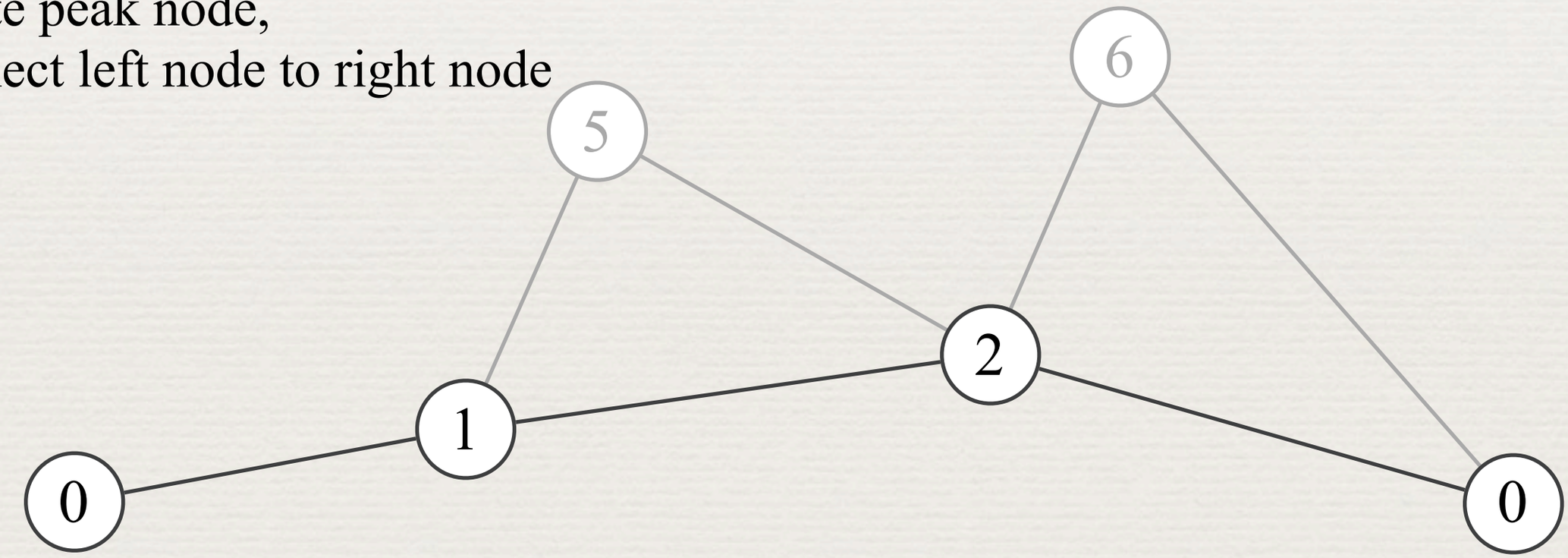


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1		3	4	4		7	8	8	
NSV_{lex}		3	3		7	6	7		11	10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node

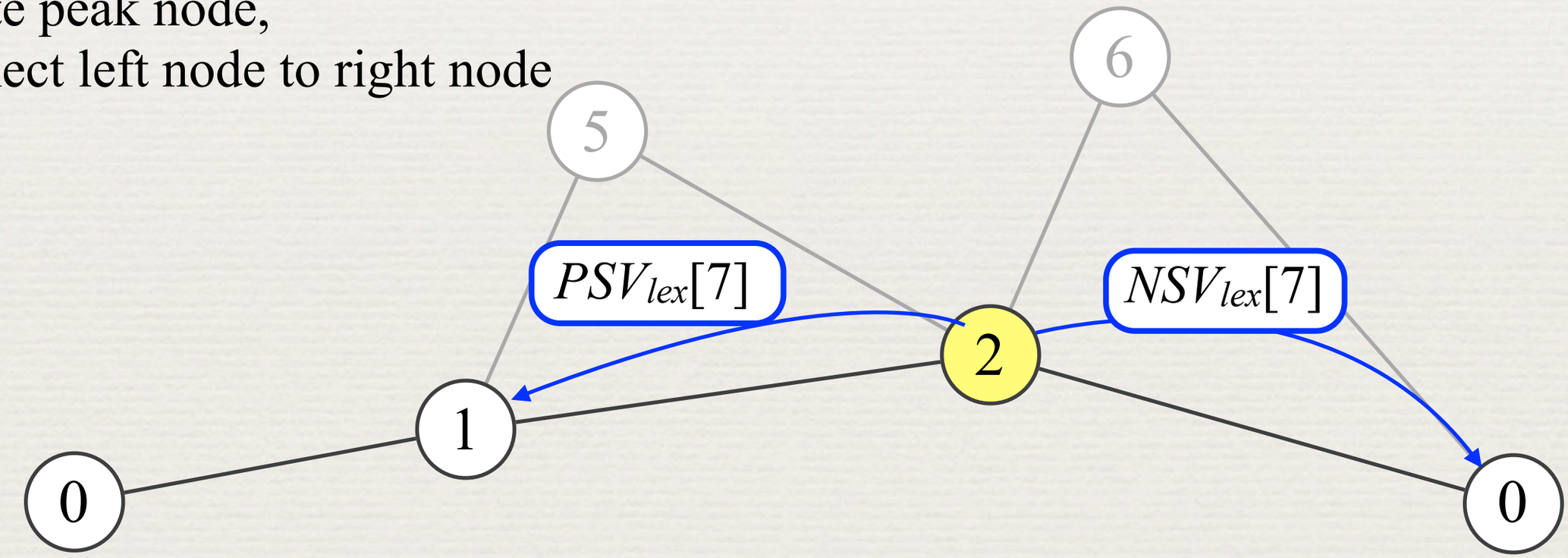


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1		3	4	4		7	8	8	
NSV_{lex}		3	3		7	6	7		11	10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
$PSV_{lex}[i]$ is left node,
$NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node



i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1		3	4	4	3	7	8	8	
NSV_{lex}		3	3		7	6	7	11	11	10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
    $PSV_{lex}[i]$ is left node,
    $NSV_{lex}[i]$ is right node

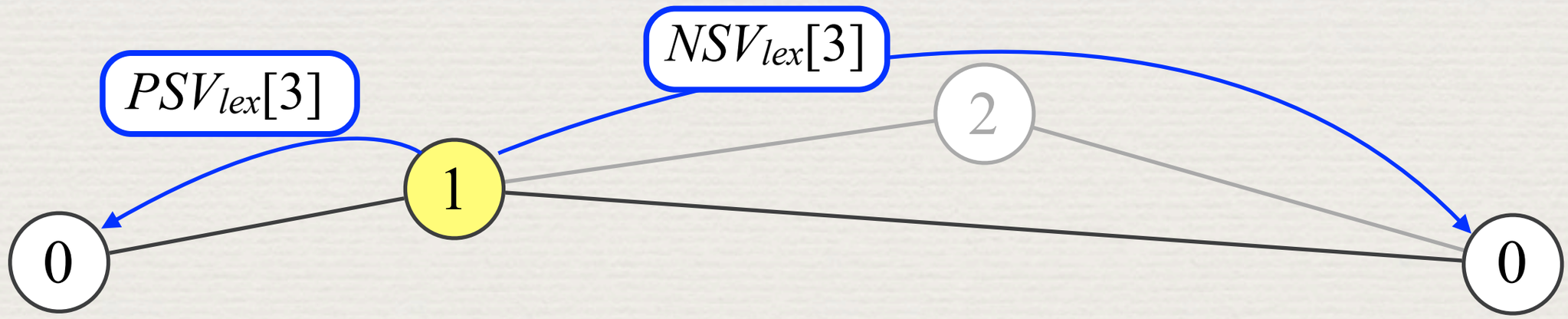
delete peak node,
connect left node to right node

i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1		3	4	4	3	7	8	8	
NSV_{lex}		3	3		7	6	7	11	11	10	11	

Peak Elimination (Crochemore and Ilie)

if $SA[i]$ is peak,
    $PSV_{lex}[i]$ is left node,
    $NSV_{lex}[i]$ is right node

delete peak node,
connect left node to right node



i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1	0	3	4	4	3	7	8	8	
NSV_{lex}		3	3	11	7	6	7	11	11	10	11	

Peak Elimination (Crochemore and Ilie)

After all peaks are deleted, all $PSV_{lex}[i]$, $NSV_{lex}[i]$ have been obtained

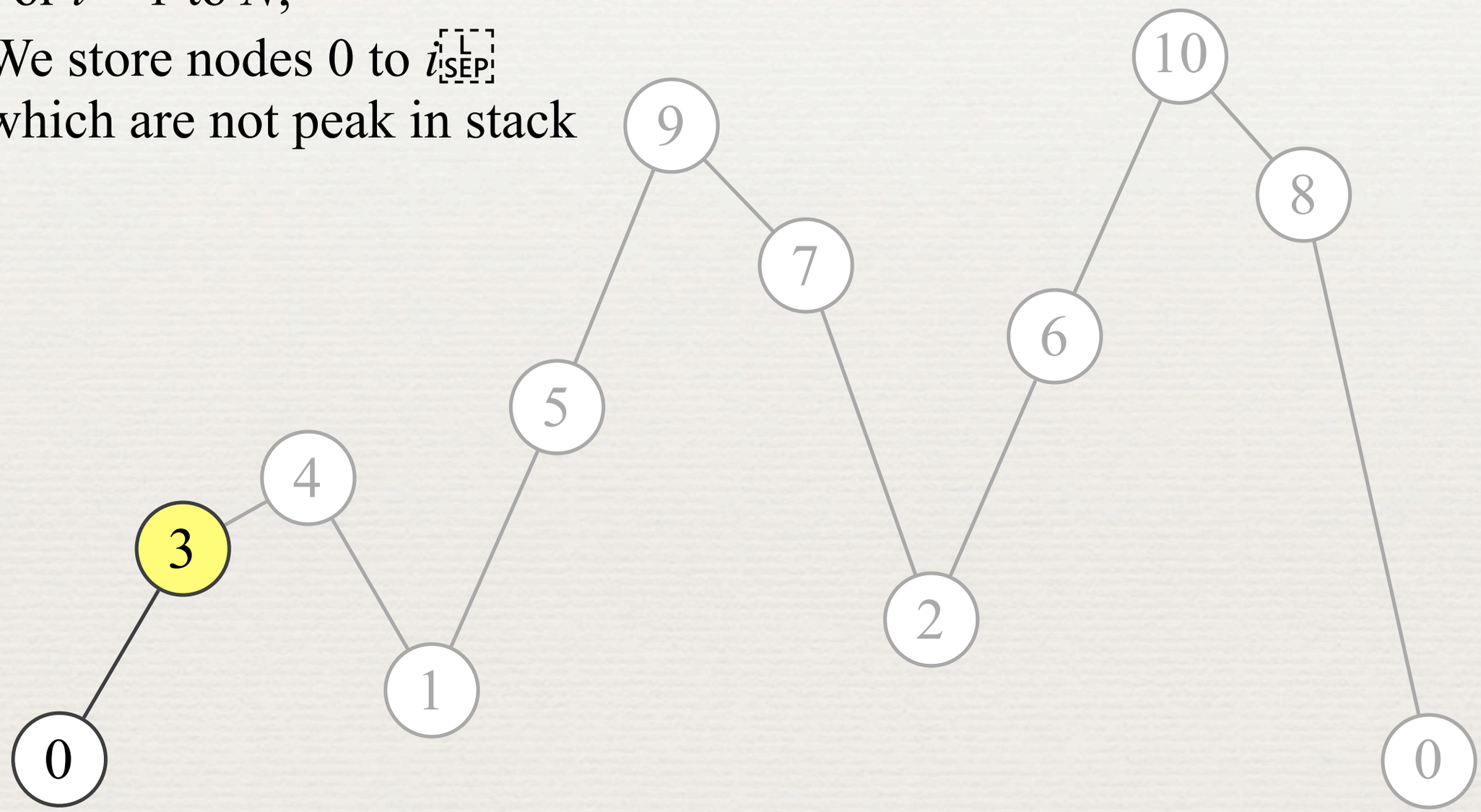
i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1	0	3	4	4	3	7	8	8	
NSV_{lex}		3	3	11	7	6	7	11	11	10	11	

LZ_BGS

For i = 1 to N,

We store nodes 0 to i_{SEP} which are not peak in stack

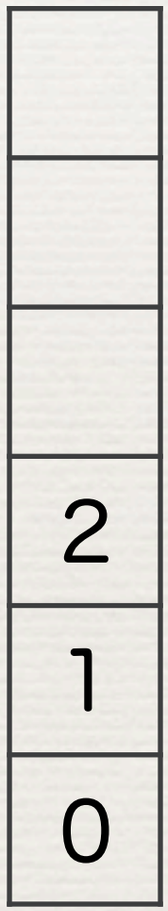
Stack S



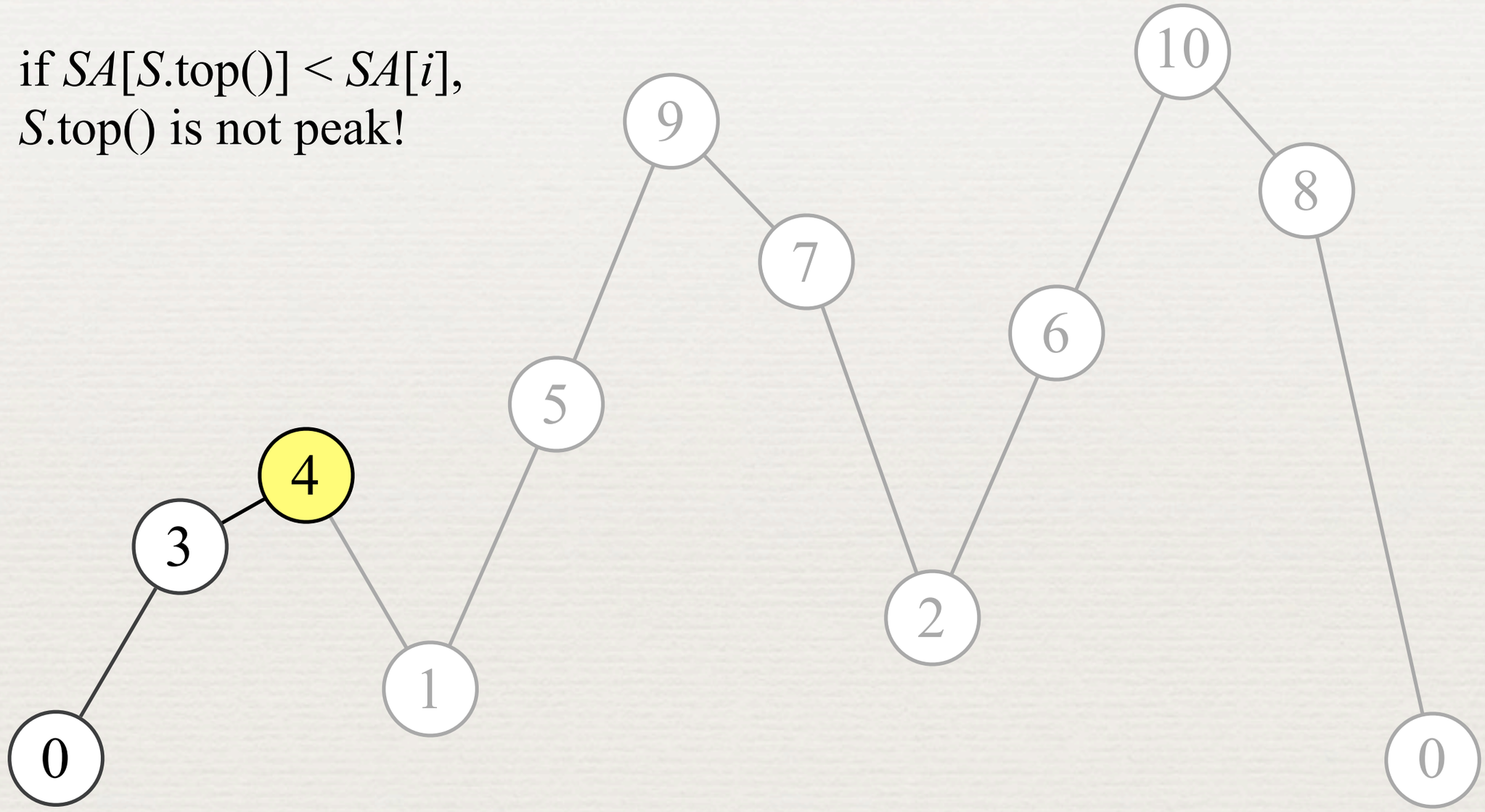
i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}												
NSV_{lex}												

LZ_BGS

Stack S



if $SA[S.\text{top}()] < SA[i]$,
S.top() is not peak!

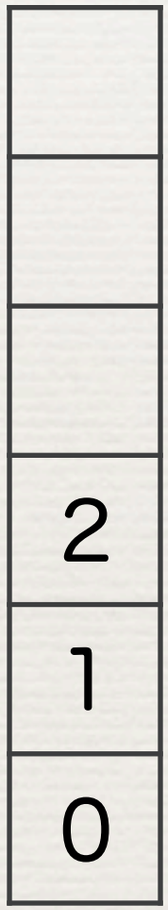


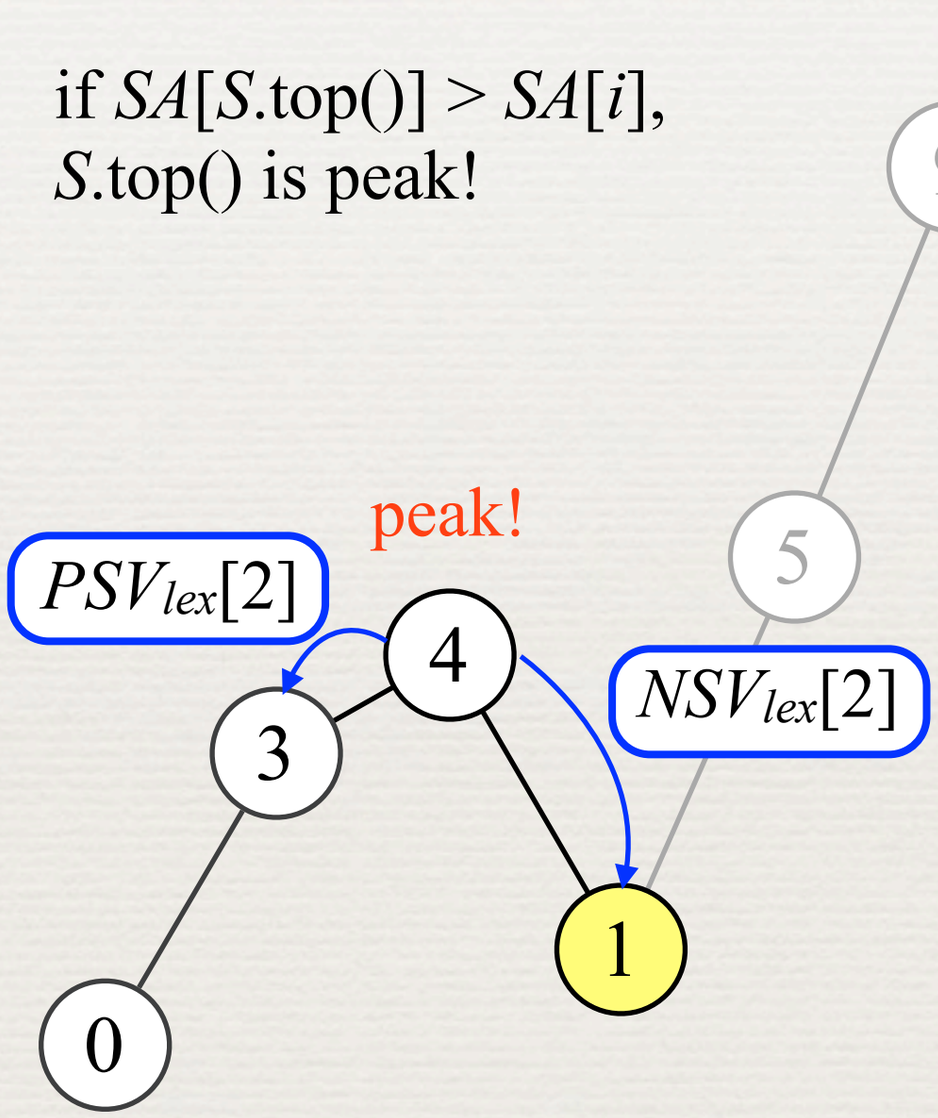
i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}												
NSV_{lex}												

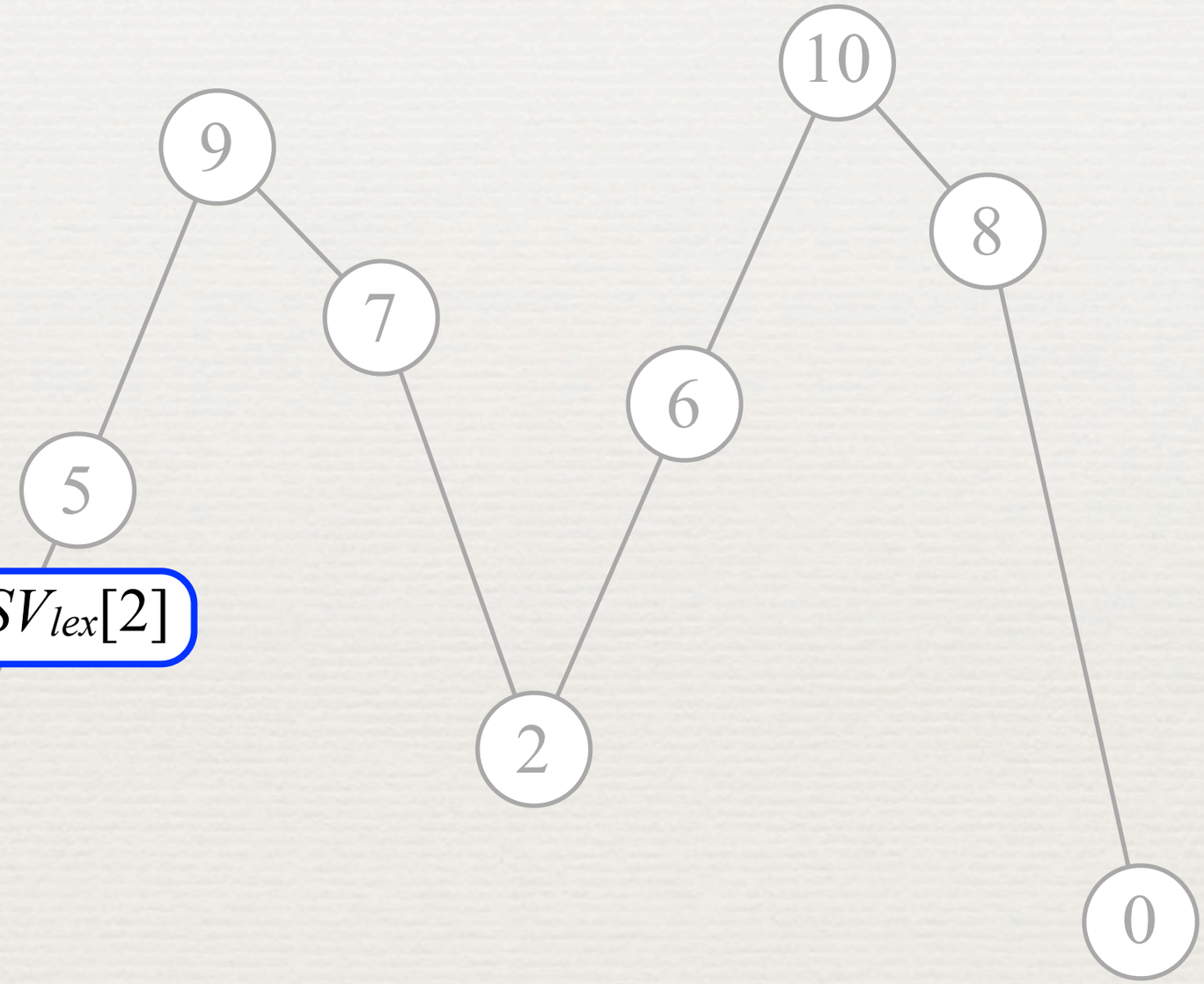
LZ_BGS

Stack S

if $SA[S.\text{top}()] > SA[i]$,
$S.\text{top}()$ is peak!





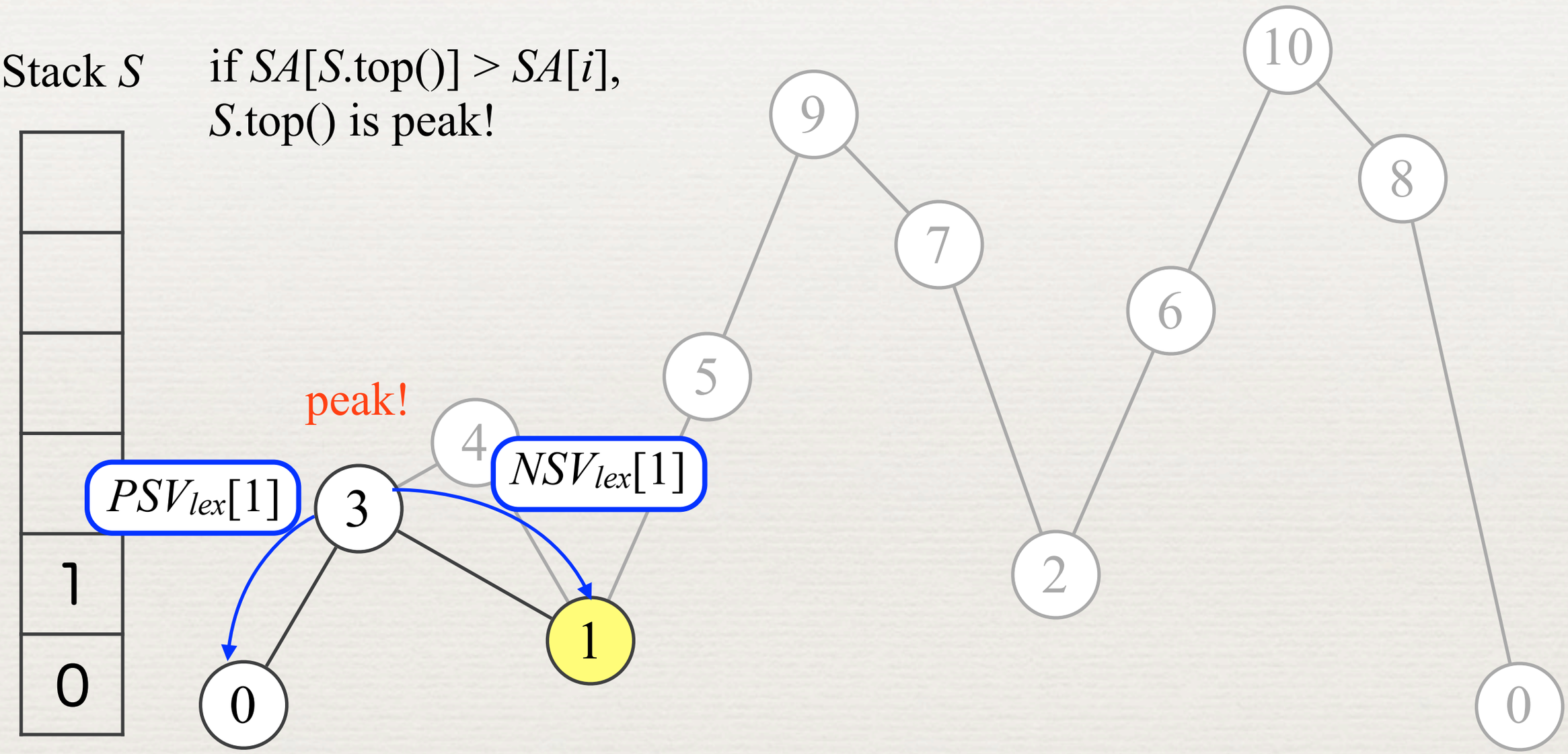


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}			1									
NSV_{lex}			3									

LZ_BGS

Stack S

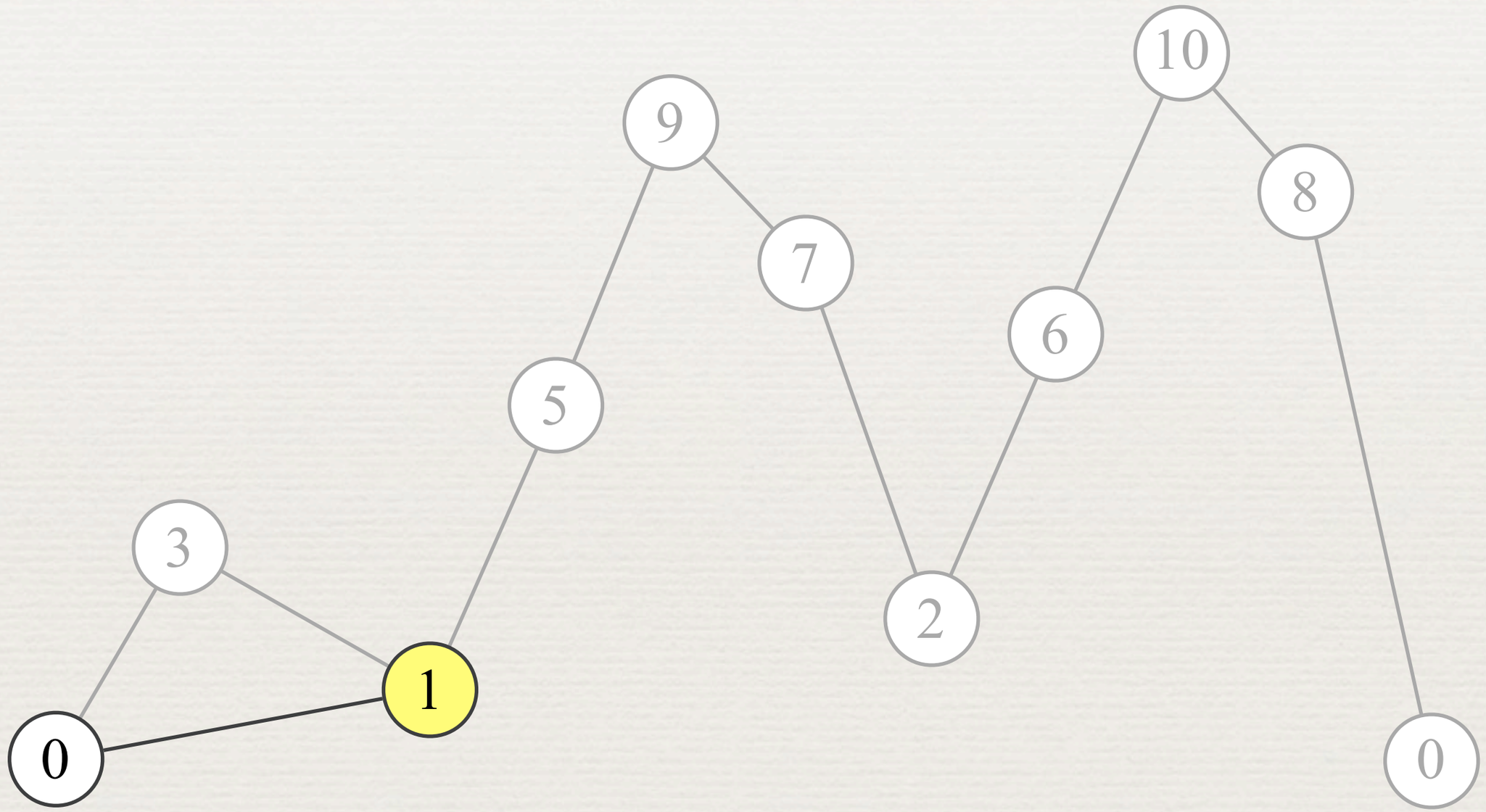
if $SA[S.\text{top}()] > SA[i]$,
S.top() is peak!



i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}		3	3									

LZ_BGS

Stack S



i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}		3	3									

Overview of LZ_BGS

requires T

Compute SA and SA^{-1} arrays

requires SA, stack S

Compute PSV_{lex} and NSV_{lex} arrays by Peak Elimination

requires T, SA, SA^{-1} PSV_{lex}, NSV_{lex}

Compute LZ factorization by naive character comparison for two candidates of $PrevOcc$

$PrevOcc(i)$ is $SA[PSV_{lex}[SA^{-1}[i]]]$ or $SA[NSV_{lex}[SA^{-1}[i]]]$

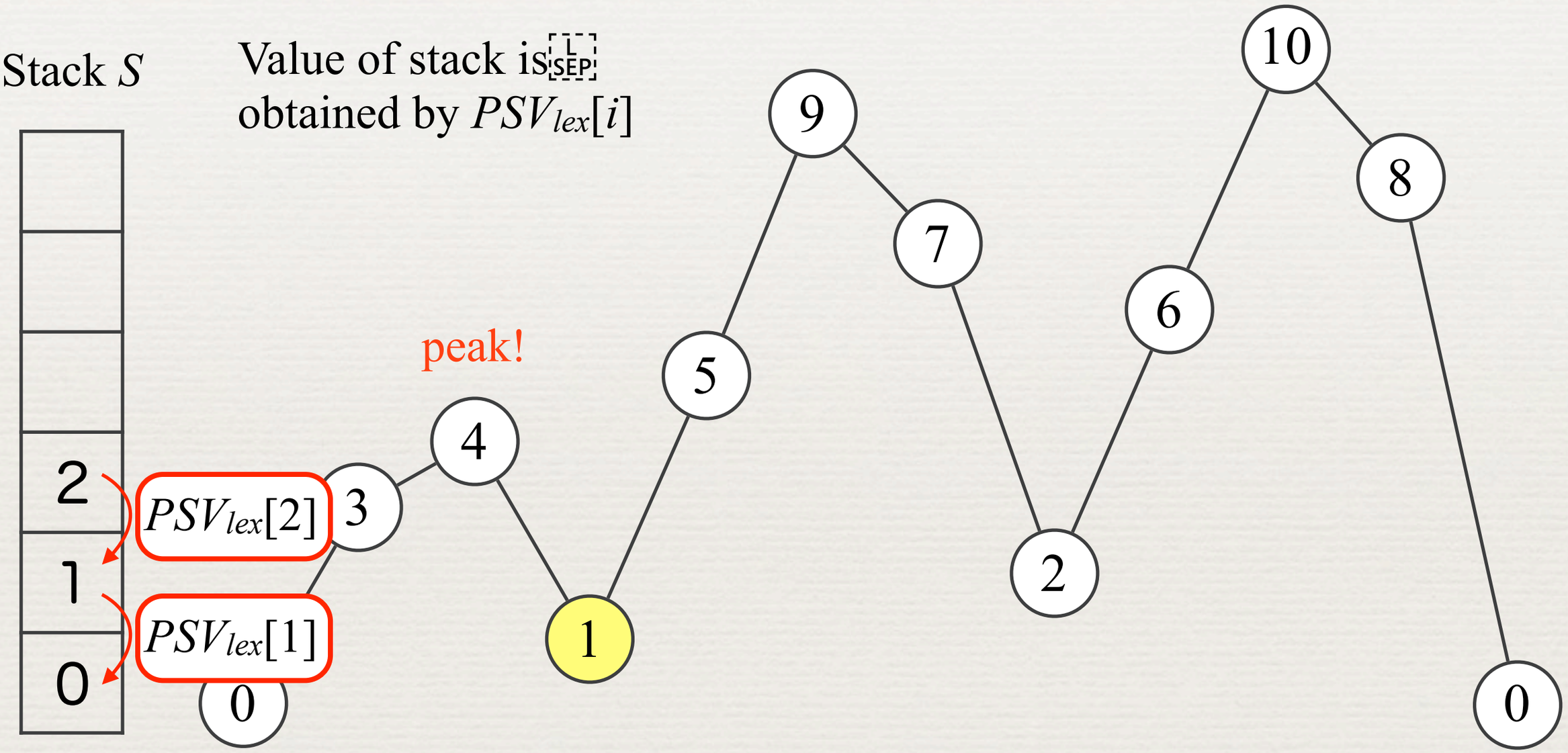
LZ_BGS runs in $O(N)$ time and $17N + 4|S|$ bytes space

Note: We suppose that a character takes 1 byte and integer takes 4 bytes

Idea of LZ_BGL

Stack S

Value of stack is SEP
obtained by $PSV_{lex}[i]$

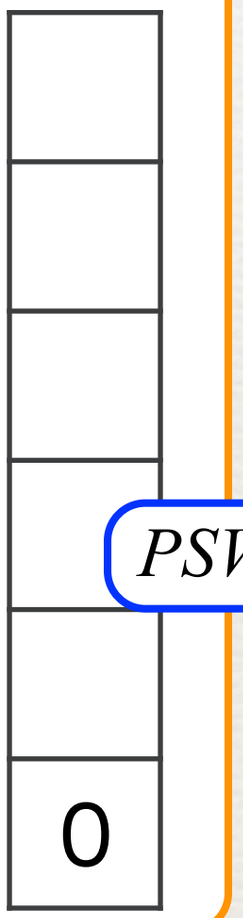


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}												

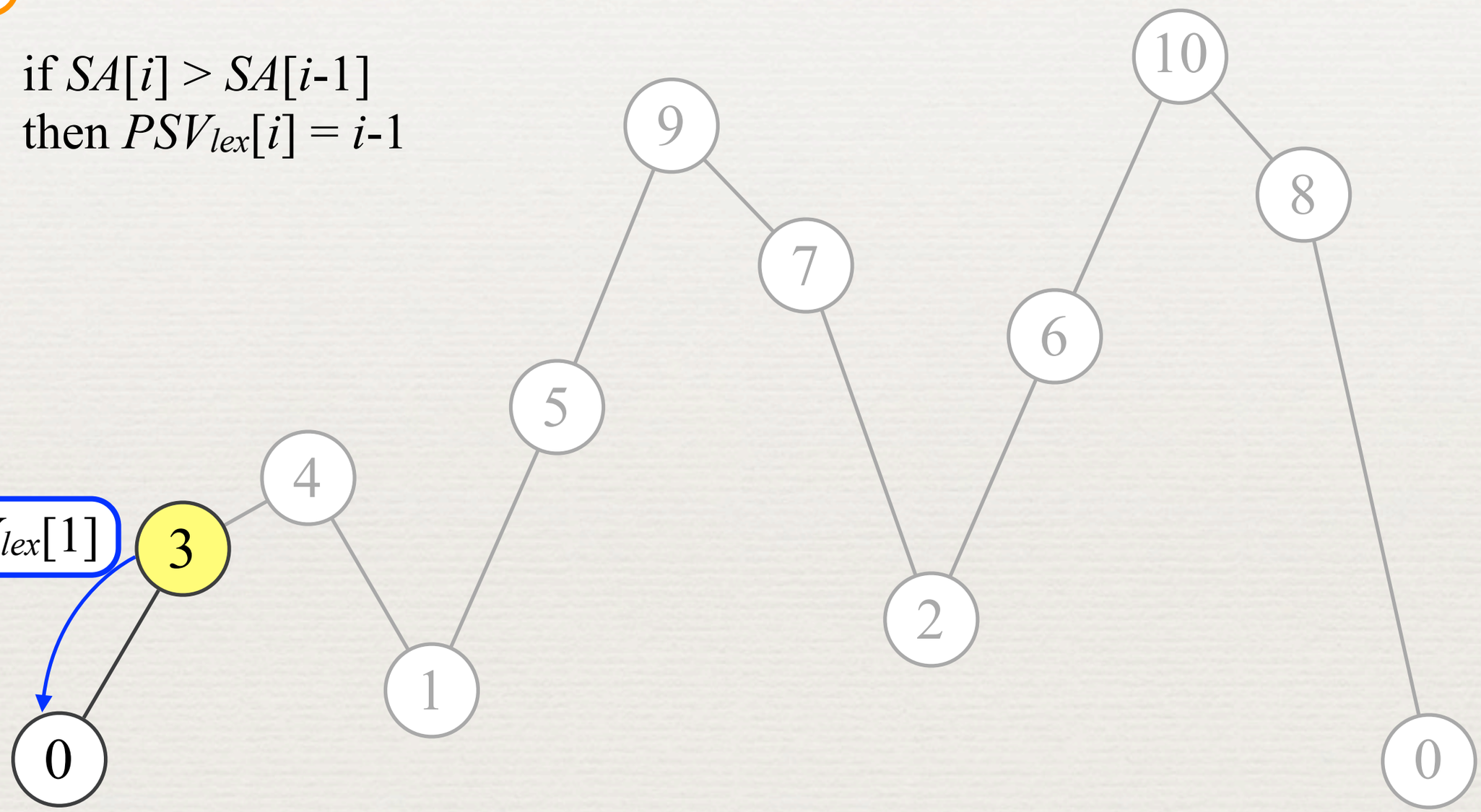
LZ_BGL

Conceptual

Stack S



if $SA[i] > SA[i\text{-}1]$
then $PSV_{lex}[i] = i\text{-}1$

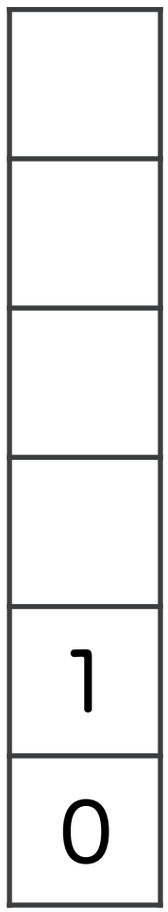


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0										
NSV_{lex}												

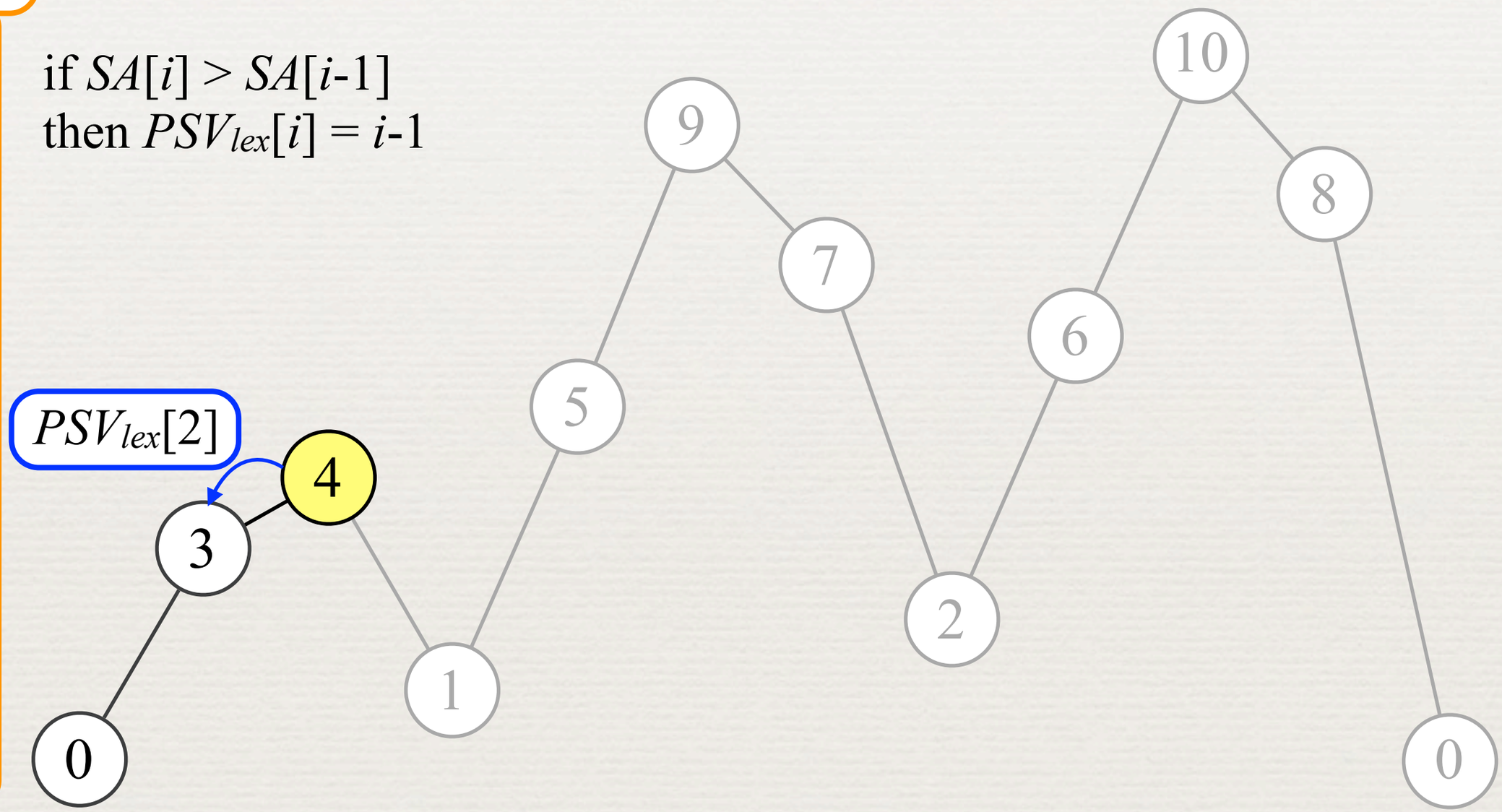
LZ_BGL

Conceptual

Stack S



if $SA[i] > SA[i\text{-}1]$
then $PSV_{lex}[i] = i\text{-}1$

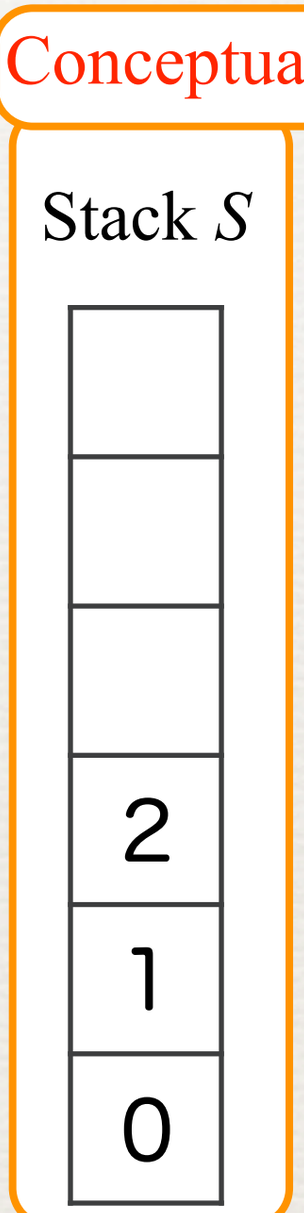


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}												

LZ_BGL

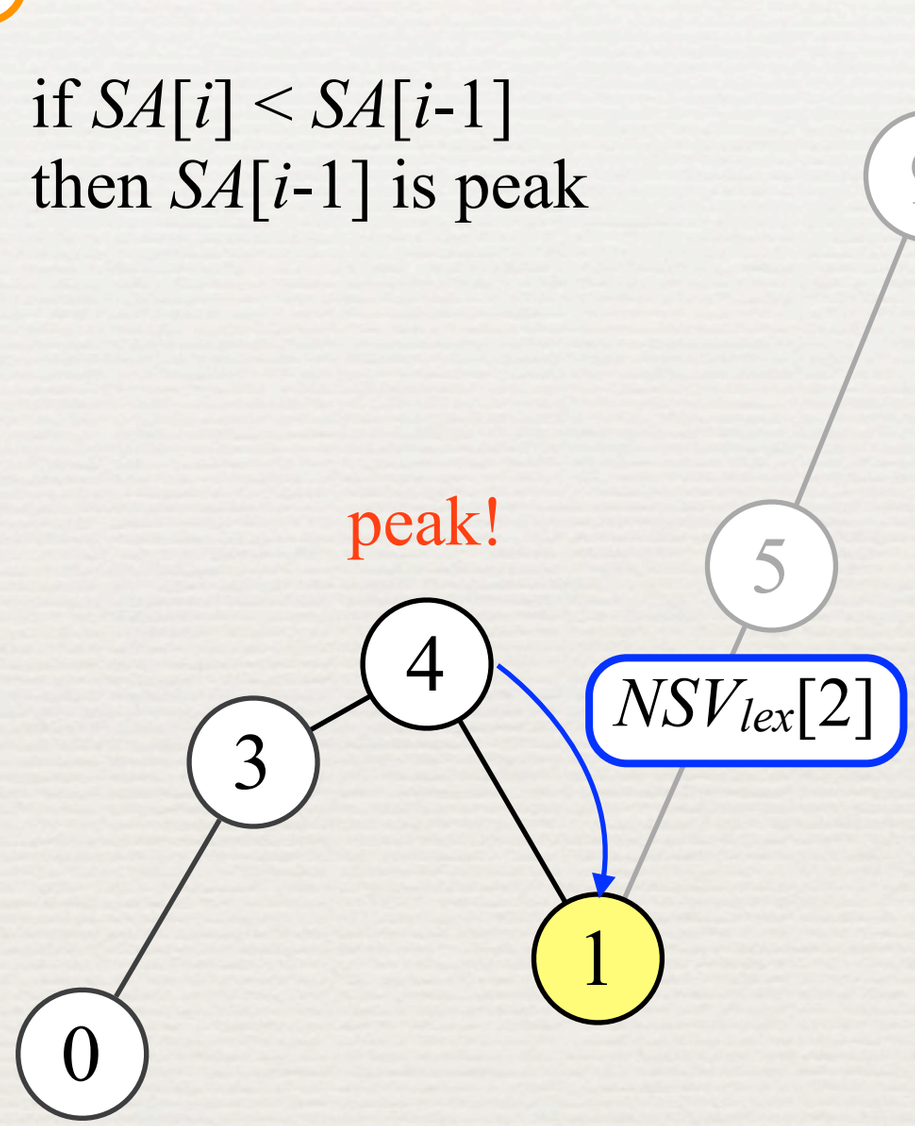
Conceptual

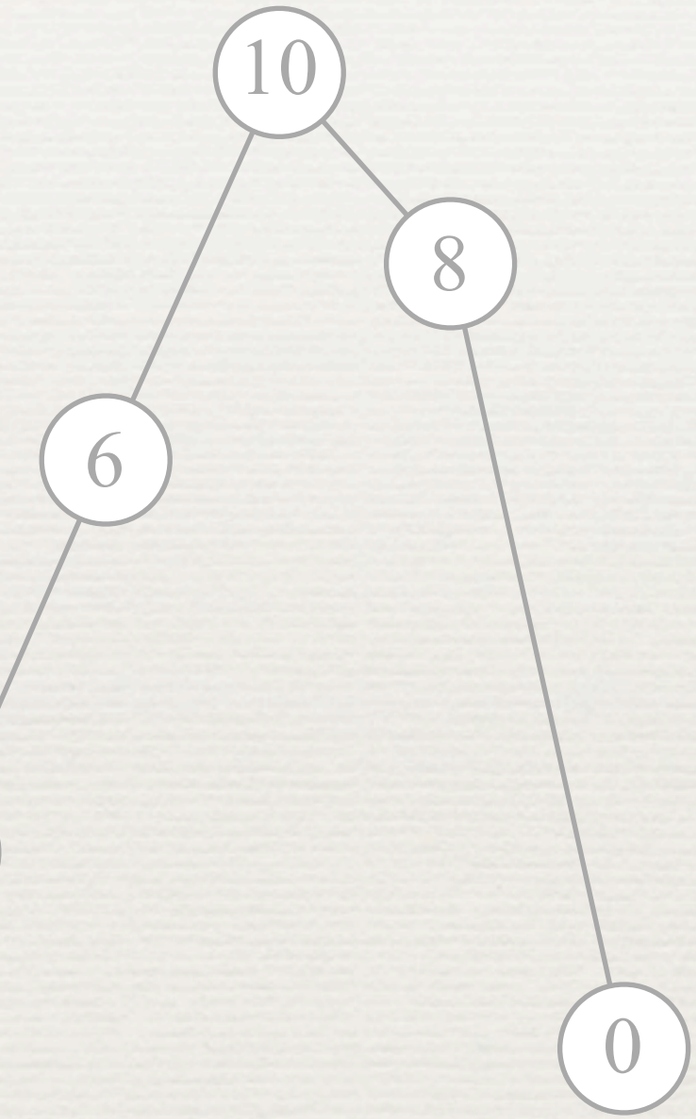
Stack S



if $SA[i] < SA[i\text{-}1]$
then $SA[i\text{-}1]$ is peak

peak!



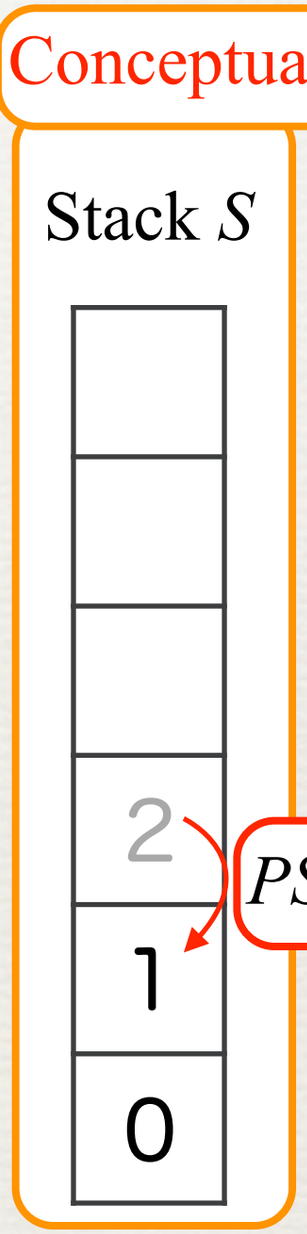


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}			3									

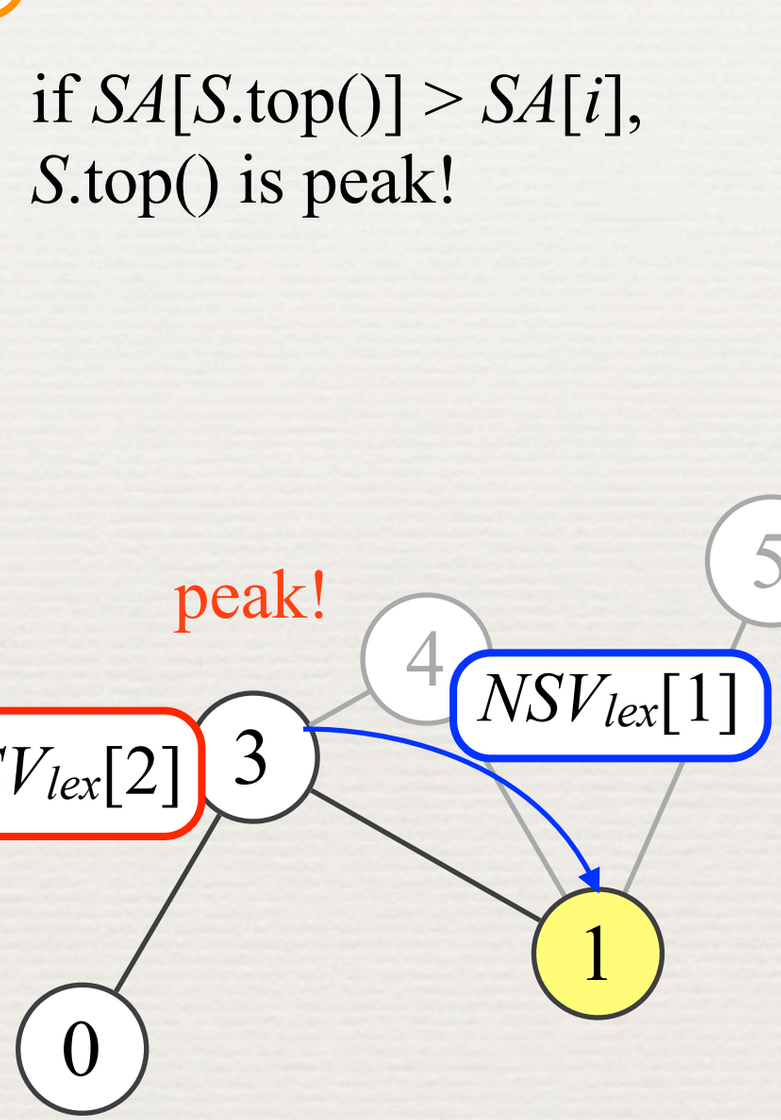
LZ_BGL

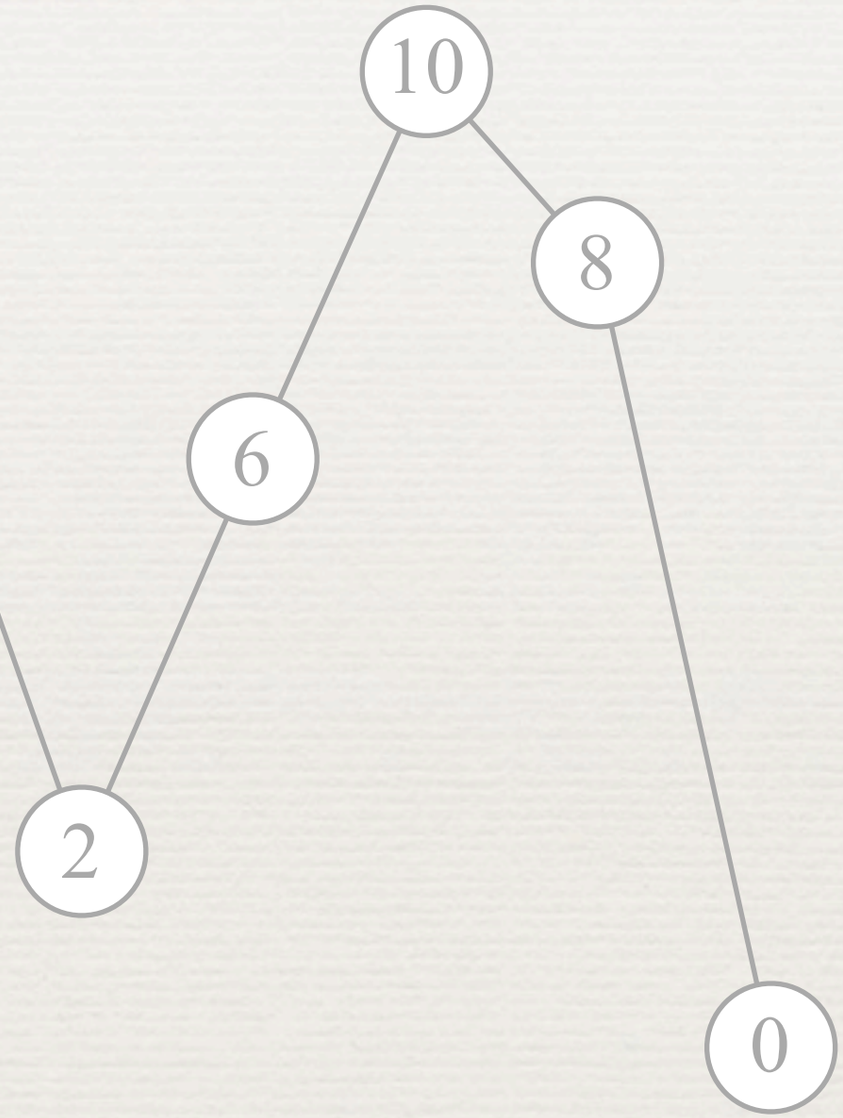
Conceptual

Stack S



if $SA[S.\text{top}()] > SA[i]$,
S.top() is peak!



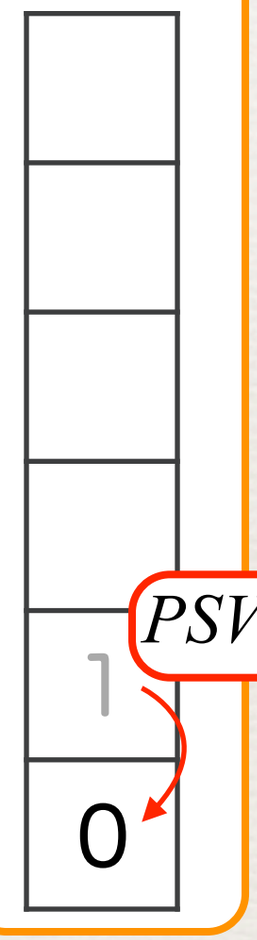


i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1									
NSV_{lex}		3	3									

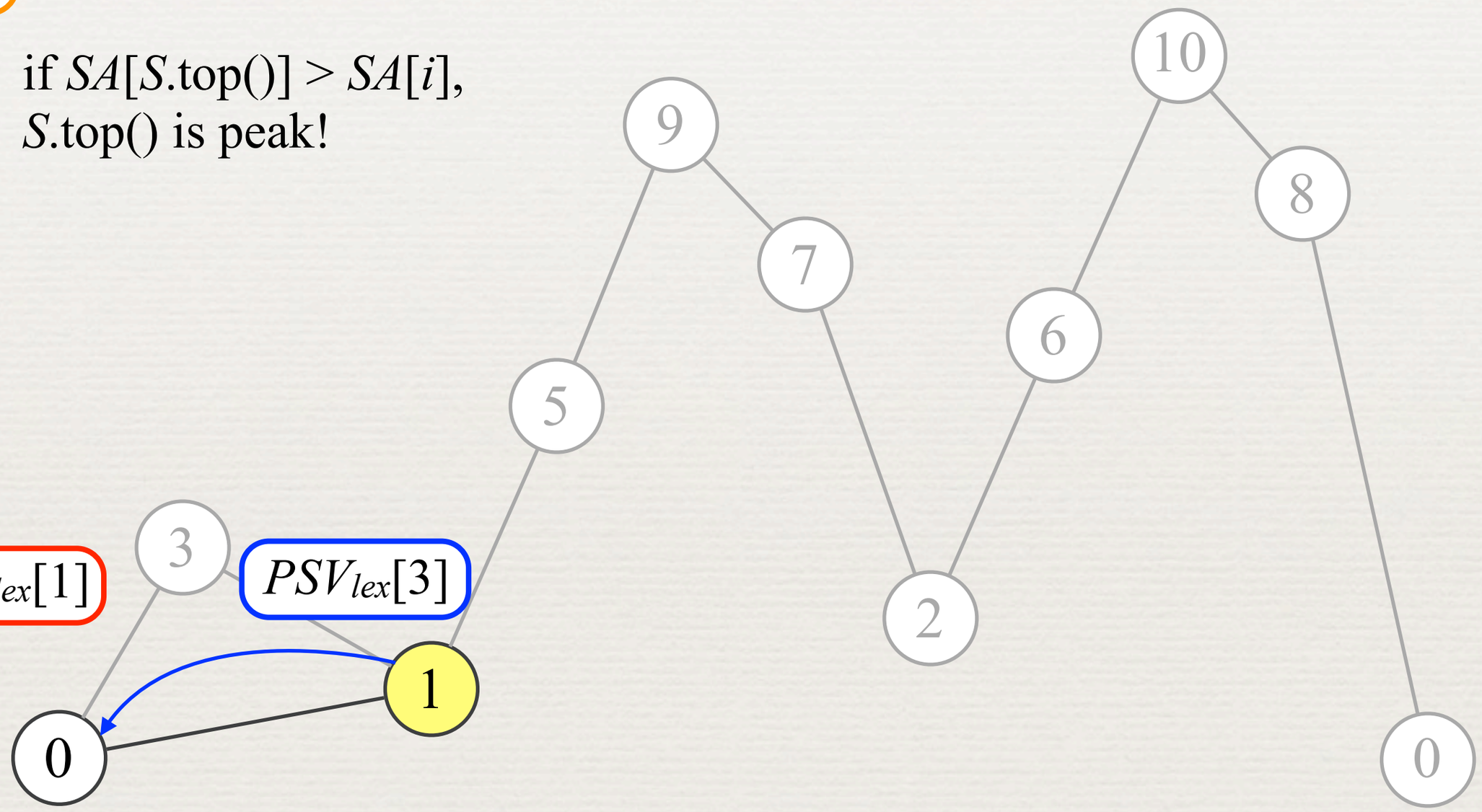
LZ_BGL

Conceptual

Stack S



if $SA[S.\text{top}()] > SA[i]$,
S.top() is peak!



i	0	1	2	3	4	5	6	7	8	9	10	11
PSV_{lex}		0	1	0								
NSV_{lex}		3	3									

Overview of LZ_BGL

requires T

Compute SA and SA^{-1} arrays

requires SA, ~~stack S~~

Compute PSV_{lex} and NSV_{lex} arrays by Peak Elimination

requires T, SA, SA^{-1} PSV_{lex}, NSV_{lex}

Compute LZ factorization by naive character comparison for two candidates of $PrevOcc$

$PrevOcc(i)$ is $SA[PSV_{lex}[SA^{-1}[i]]]$ or $SA[NSV_{lex}[SA^{-1}[i]]]$

LZ_BGL runs in $O(N)$ time and $17N$ ~~$+ 4|S|$~~ bytes space

Note: we suppose that a character takes 1 byte and integer takes 4 bytes

LZ_BGT

- LZ_BGT uses PSV_{text}, NSV_{text} rather than PSV_{lex}, NSV_{lex}

$PrevOcc(i)$ is $PSV_{text}[i]$ or $NSV_{text}[i]$

$$PSV_{text}[SA[i]] = SA[PSV_{lex}[i]]$$
$$NSV_{text}[SA[i]] = SA[NSV_{lex}[i]]$$

i	$SA[i]$	suffix $SA[i]$
⋮	⋮	⋮
1	2	baababababaaaaabbbabab
18		bab
3	7	babaaaaabbbabab
16		babab
5	5	bababaaaaabbbabab
6	15	bbabab

$PSV_{text}[7]$

$NSV_{text}[7]$

It occurs before suffix 7

PSV_{text}, NSV_{text} can be computed in $O(N)$ time in similar way of LZ_BGL by using Φ array ($\Phi[SA[i]] = SA[i\text{-}1]$)

Overview of LZ_BGT

requires T

Compute Φ arrays

requires Φ

Compute PSV_{text} and NSV_{text} arrays by Peak Elimination

requires T, PSV_{text}, NSV_{text}

Compute LZ factorization by naive character comparison for two candidates of *PrevOcc*

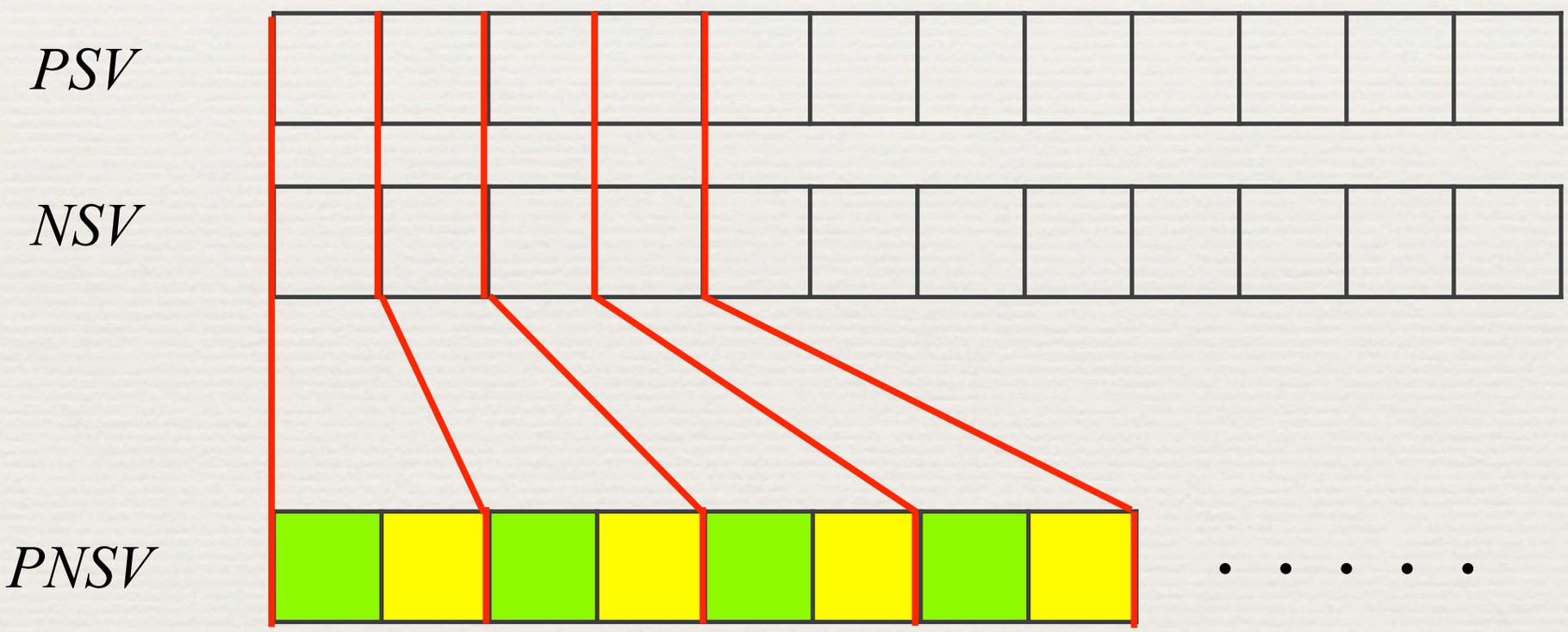
$PrevOcc(i)$ is $PSV_{text}[i]$ or $NSV_{text}[i]$

LZ_BGT runs in $O(N)$ time and $13N$ bytes space

Note: we suppose that a character takes 1 byte and integer takes 4 bytes

Interleaving Array

- In order to reduce cache misses for accessing *PSV* and *NSV*, we use interleaving array.



$$PNSV[2i] = PSV[i]$$
$$PNSV[2i+1] = NSV[i]$$

Summary of our algorithm

Algorithm	Computation of *PSV*, *NSV*	Order of L SEP *PSV*, *NSV*	Space
BGS, iBGS	stack	lex	$17N + 4\|S\|$ bytes
BGL, iBGL	no stack	lex	$17N$ bytes
BGT, iBGT	no stack	text	$13N$ bytes

iBGX: interleaving version

$|S|$: stack size (in worst case $|S| = N$)

Computational Experiment

- All variants of the algorithm constantly out perform LZ_OG and LZ_iOG, and some cases can be up to 2 to 3 times faster

	OG	**iOG**	**BGS**	**iBGS**	**BGL**	**iBGL**	**BGT**	**iBGT**
	13N	13N	17N + 4|S|	17N + 4|S|	17N	17N	13N	13N
E.coli	0.64	0.58	0.26	**0.23**	0.33	0.29	0.45	0.37
bible	0.37	0.34	0.20	**0.19**	0.25	0.22	0.27	0.24
chr19.dna4	10.05	9.25	4.40	**4.00**	5.33	4.71	7.64	6.54
chr22.dna4	5.37	4.91	2.27	**2.06**	2.77	2.44	4.09	3.45
fib_s5702887	0.18	0.18	0.15	0.16	0.16	0.15	0.15	**0.14**
howto	4.20	3.91	2.30	**2.15**	2.79	2.51	3.28	2.91
mozilla	5.30	4.95	3.19	**3.13**	3.91	3.65	4.31	3.86
p32Mb	5.58	5.02	2.47	**2.44**	3.14	3.03	4.43	3.74

bold: fastest running time

red: fastest running time for the variants use 13N space

corpus: http://www.cas.mcmaster.ca/~bill/strings

Summary

- We proposed faster and simpler LZ factorization algorithms which avoid construction of *LPF*, *PrevOcc* arrays

- All of our algorithms outperform the previous fastest algorithm LZ_OG

- Algorithms using similar idea were proposed very recently
  These results show that our idea is very effective

  [Kempa and Puglisi, ALENEX 2013]

  the algorithm is not linear time, but is very fast in practice

  [K¨arkk¨ainen+, arXiv:1212.2952]

  linear time, the algorithm runs in $9N$ bytes space