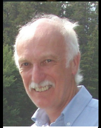
Extracting the Sparse Longest Common Prefix Array from the Suffix Binary Search Tree

Tomohiro I [1]



Robert W. Irving [2]

Dominik Köppl [3]

Lorna Love [2]

[1] Department of Artificial Intelligence, Kyushu Institute of Technology, Japan

[2] School of Computing Science, University of Glasgow, Glasgow, UK

[3] M&D Data Science Center, Tokyo Medical and Dental University, Japan

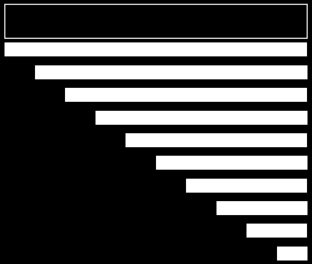
suffix sorting

$T = \boxed{}$

suffix sorting

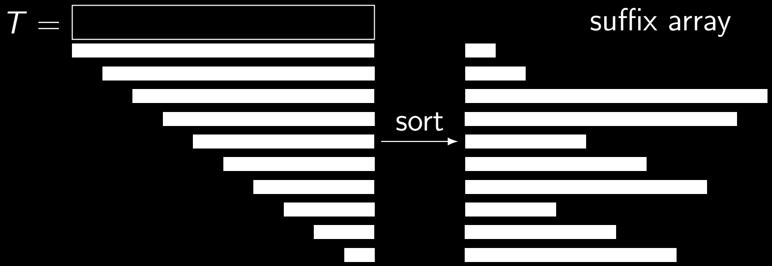
- sort *all* suffixes lexicographically

$T =$



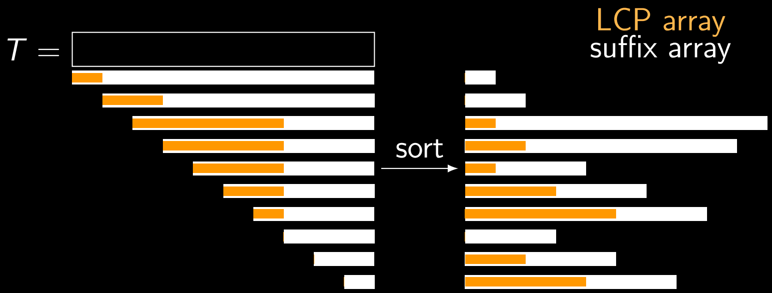
suffix sorting

- sort *all* suffixes lexicographically



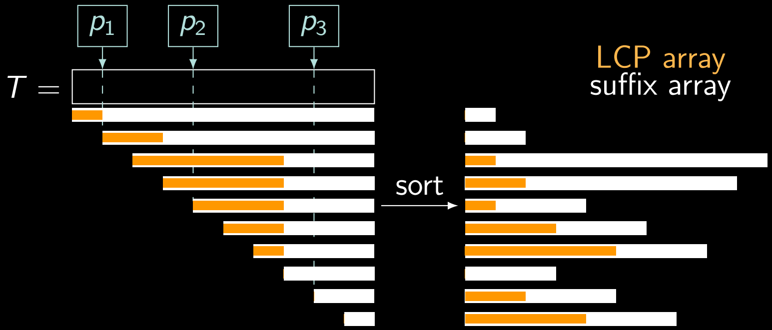
suffix sorting

- sort *all* suffixes lexicographically
- lengths of the longest common prefix (LCP) between adjacent suffixes.
- solved in $\mathcal{O}(n)$ time and words of space



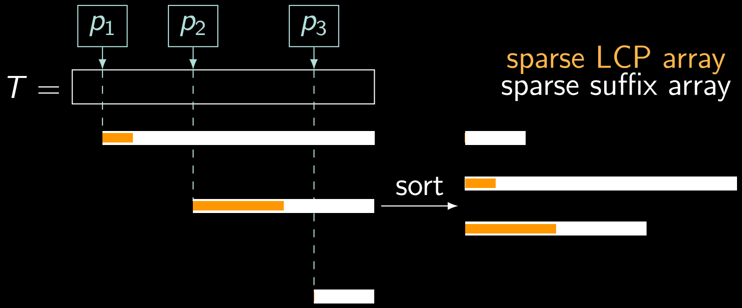
suffix sorting

- sort *all* suffixes lexicographically
- lengths of the longest common prefix (LCP) between adjacent suffixes.
- solved in $\mathcal{O}(n)$ time and words of space
- sometimes need only suffixes starting at $p_1, \ldots, p_m$



sparse suffix sorting

- sort *all* suffixes lexicographically
- lengths of the longest common prefix (LCP) between adjacent suffixes.
- solved in $\mathcal{O}(n)$ time and words of space
- sometimes need only suffixes starting at $p_1, \ldots, p_m$

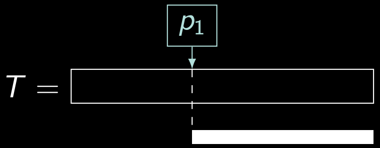


dynamic sparse suffix sorting

$T =$ [ ]

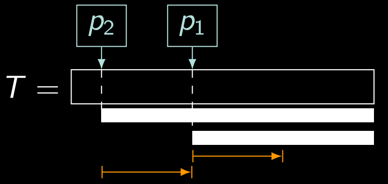
dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order



dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query



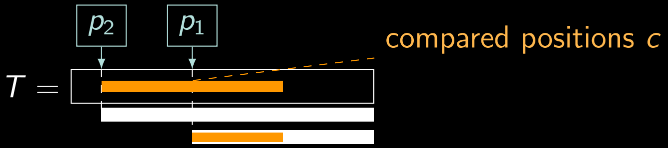
LCE query $\mathrm{lce}(p_1, p_2)$

dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c := \#$ characters to compare for sorting

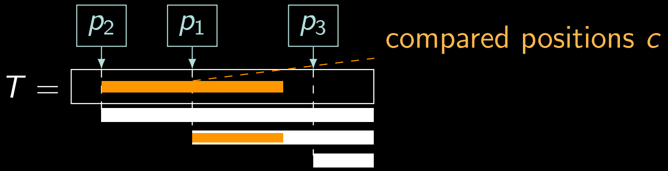
dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c :=$ # characters to compare for sorting



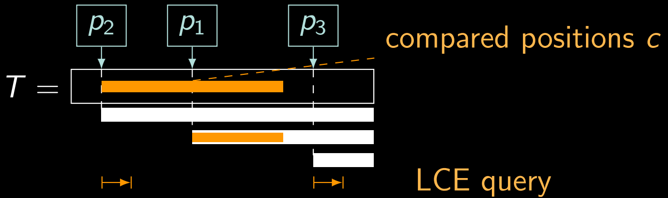
dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c := \#$ characters to compare for sorting



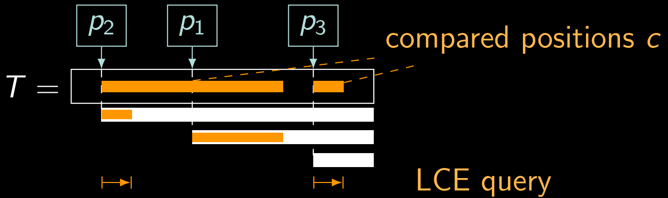
dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c := \#$ characters to compare for sorting



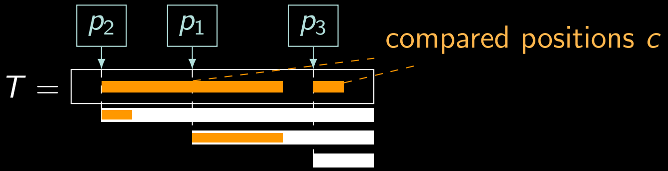
dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c := \#$ characters to compare for sorting

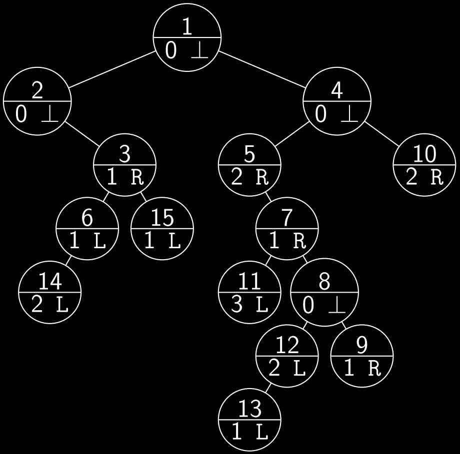


dynamic sparse suffix sorting

- $p_1, \ldots, p_m$: online, arbitrary order
- compare two suffixes with LCE query
- $c := \#$ characters to compare for sorting
- how to store their order?



suffix binary search tree (SBST)



SBST of Irving and Love'03:
binary search tree representation

each node

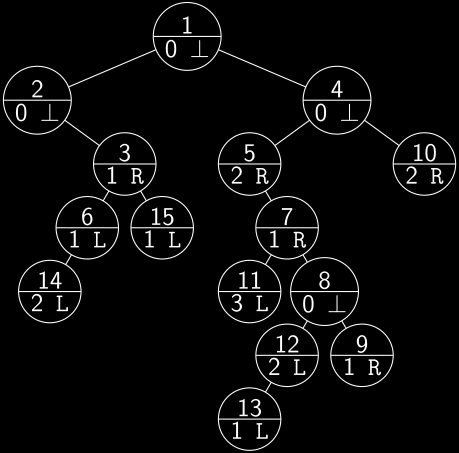
- represents a position p_i
- stores a flag $\in \{\mathtt{L}, \mathtt{R}, \perp\}$
- the LCE with an ancestor

running example

- ISA : inverse suffix array
- SA : suffix array
- LCP : LCP array

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$T[i]$	c	a	a	t	c	a	c	g	g	t	c	g	g	a	c
ISA[i]	6	1	4	14	7	3	9	12	13	15	8	11	10	2	5

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
LCP[r]	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2



problem definition

- obtain SA from in-order traversal in $\mathcal{O}(m)$ time.
- how to obtain LCP?

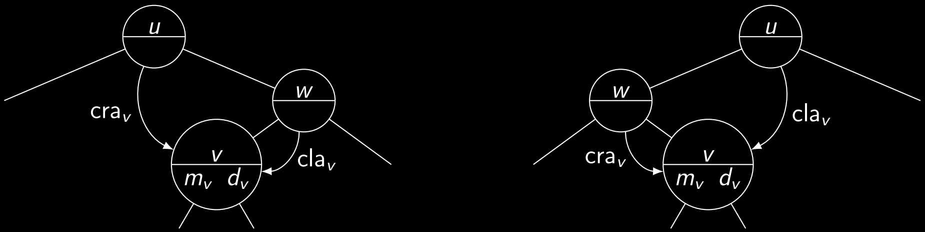
r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
LCP[r]	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2

closest left/right ancestors

let v be a node

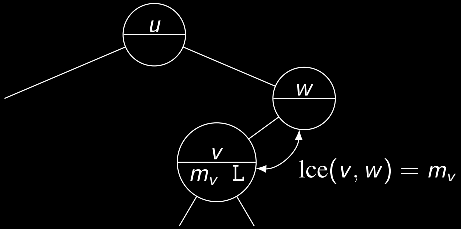
- cla_v : lowest node having v as a descendant in its left subtree
- cra_v : lowest node having v as a descendant in its right subtree

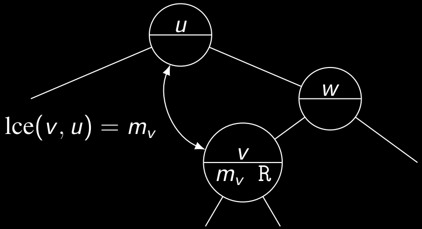
$\Rightarrow$ either cla_v or cra_v is v's parent

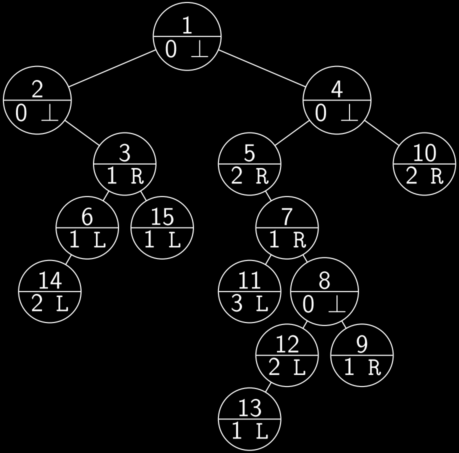


LCE value m_v and flag d_v

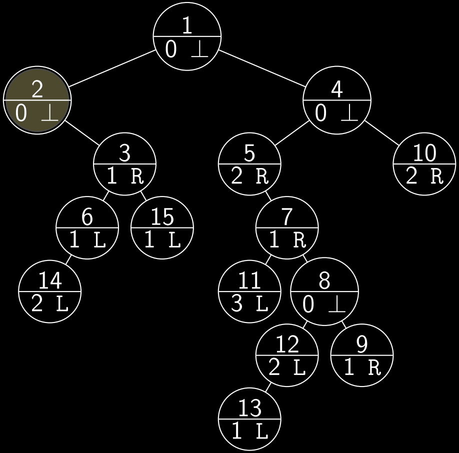
- $\mathsf{ca}_v := \mathrm{argmax}_{u \in \{\mathsf{cla}_v, \mathsf{cra}_v\}} \mathrm{lce}(v, u)$
- if $\mathsf{ca}_v = \mathsf{cla}_v$, then $m_v = \mathrm{lce}(v, \mathsf{cla}_v)$, $d_v = \mathtt{L}$.
- if $\mathsf{ca}_v = \mathsf{cra}_v$, then $m_v = \mathrm{lce}(v, \mathsf{cra}_v)$, $d_v = \mathtt{R}$.
- if ca_v is undefined, then $m_v = 0$, $d_v = \perp$.







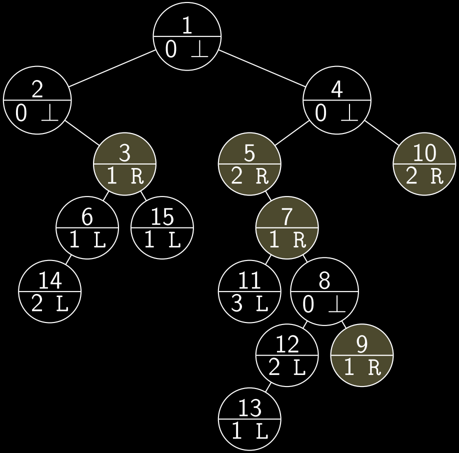
r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
LCP[r]	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2



rules:

e : neither left child nor cra_v exists
$\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e														
	0														

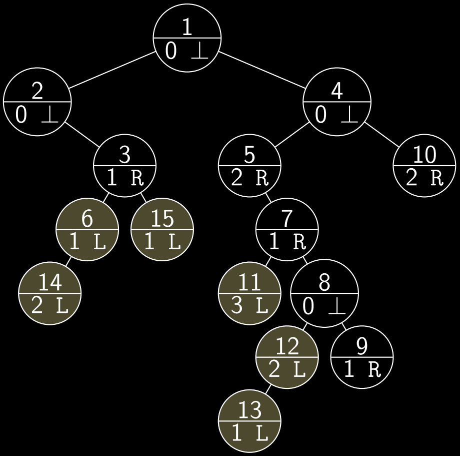


rules:

e : neither left child nor cra_v exists $\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$

r : $d_v = \texttt{R} \Rightarrow \text{LCP}[\text{ISA}[v]] \geq m_v$

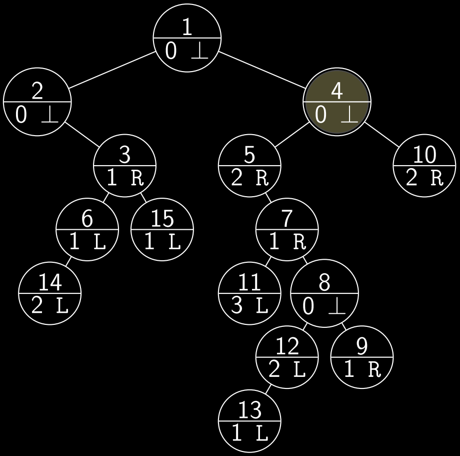
r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e			r			r		r				r		r
	0			1			2		1				1		2



rules:

- e : neither left child nor cra_v exists $\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$
- r : $d_v = \mathtt{R} \Rightarrow \text{LCP}[\text{ISA}[v]] \geq m_v$
- l : $d_v = \mathtt{L}$ and right subtree of v is empty $\Rightarrow \text{LCP}[\text{ISA}[v]+1] = m_v$

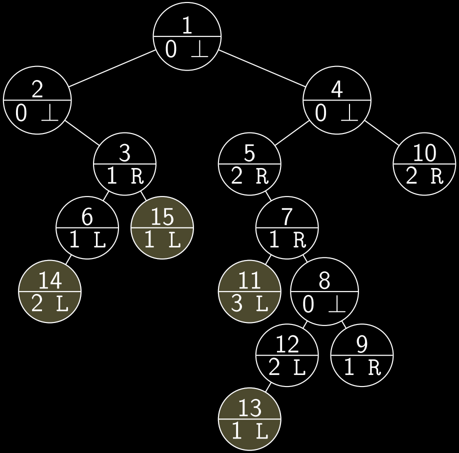
r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e	l	l	r	l		r	l	r	l	l		r		r
	0		2	1		1	2		3		1	2	1		2



rules:

- e : neither left child nor cra_v exists $\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$
- r : $d_v = \texttt{R} \Rightarrow \text{LCP}[\text{ISA}[v]] \geq m_v$
- l : $d_v = \texttt{L}$ and right subtree of v is empty $\Rightarrow \text{LCP}[\text{ISA}[v] + 1] = m_v$
- d : v has left child $u \Rightarrow$ rightmost node in u's subtree determines $\text{LCP}[\text{ISA}[v]]$

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e	l	l	r	l		r	l	r	l	l		r	d	r
	0		2	1		1	2		3		1	2	1	0	2



rules:

- e : neither left child nor cra_v exists $\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$
- r : $d_v = \mathtt{R} \Rightarrow \text{LCP}[\text{ISA}[v]] \geq m_v$
- l : $d_v = \mathtt{L}$ and right subtree of v is empty $\Rightarrow \text{LCP}[\text{ISA}[v]+1] = m_v$
- d : v has left child $u \Rightarrow$ rightmost node in u's subtree determines $\text{LCP}[\text{ISA}[v]]$
- a : otherwise: cra_v determines $\text{LCP}[\text{ISA}[v]]$

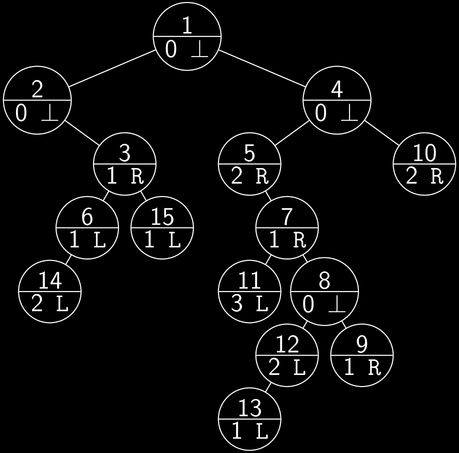
r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e	a	l	r	a		r	a	r	a	l		r	d	r
LCP[r]	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2

- rules e, r, l can be computed in constant time per node.
- how to compute rules d and a?

rules:

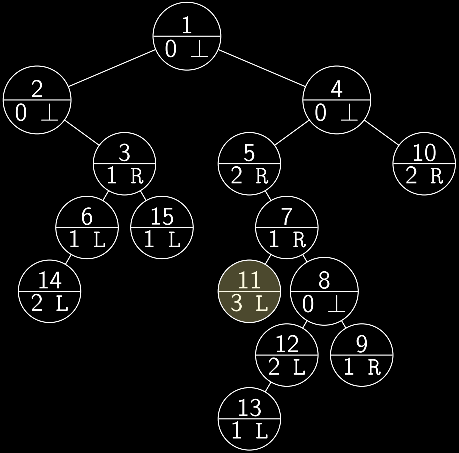
- e : neither left child nor cra_v exists $\Rightarrow \text{LCP}[\text{ISA}[v]] = 0$
- r : $d_v = \texttt{R} \Rightarrow \text{LCP}[\text{ISA}[v]] \geq m_v$
- l : $d_v = \texttt{L}$ and right subtree of v is empty $\Rightarrow \text{LCP}[\text{ISA}[v] + 1] = m_v$
- d : v has left child $u \Rightarrow$ rightmost node in u's subtree determines $\text{LCP}[\text{ISA}[v]]$
- a : otherwise: cra_v determines $\text{LCP}[\text{ISA}[v]]$

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA[r]	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
rules	e	a	l	r	a		r	a	r	a	l		r	d	r
LCP[r]	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2



task

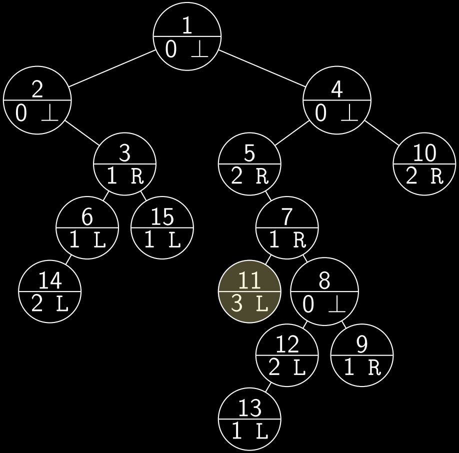
compute LCP[11]



task

compute LCP[11]

$= \mathrm{lce}(\mathrm{cra}_{11}, 11)$ since 11 has no left child

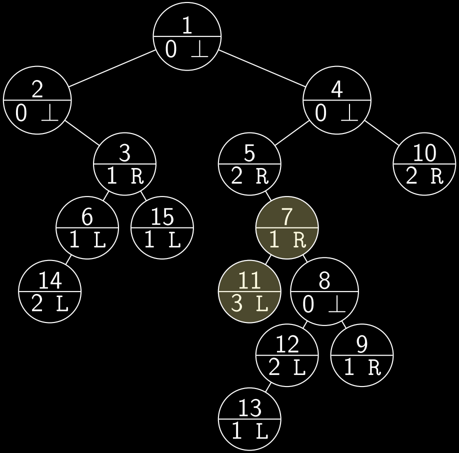


task

compute LCP[11]

$= \mathrm{lce}(\mathsf{cra}_{11}, 11)$ since 11 has no left child

$= \mathrm{lce}(\mathsf{cra}_{11}, \mathsf{cla}_{11})$ since $d_{11} = \mathtt{L}$ (proof later)



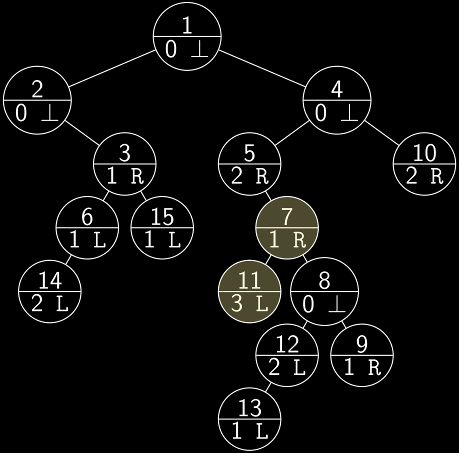
task

compute LCP[11]

$= \mathrm{lce}(\mathrm{cra}_{11}, 11)$ since 11 has no left child

$= \mathrm{lce}(\mathrm{cra}_{11}, \mathrm{cla}_{11})$ since $d_{11} = \mathtt{L}$ (proof later)

$= \mathrm{lce}(\mathrm{cra}_{11}, 7) = m_7 = 1$ since $\mathrm{cra}_{11} = \mathrm{cra}_7$.



task

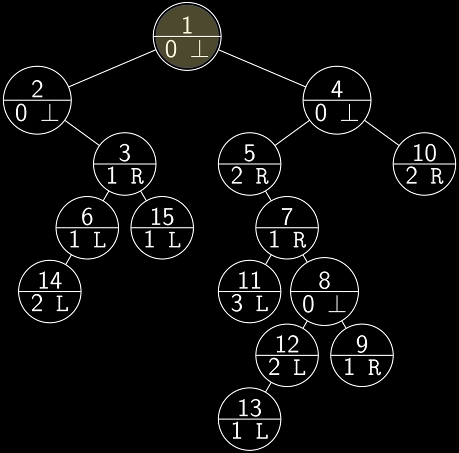
compute LCP[11]

$= \mathrm{lce}(\mathrm{cra}_{11}, 11)$ since 11 has no left child

$= \mathrm{lce}(\mathrm{cra}_{11}, \mathrm{cla}_{11})$ since $d_{11} = \mathtt{L}$ (proof later)

$= \mathrm{lce}(\mathrm{cra}_{11}, 7) = m_7 = 1$ since $\mathrm{cra}_{11} = \mathrm{cra}_7$.

- goal: maintain $\mathrm{lce}(\mathrm{cra}_v, \mathrm{cla}_v)$ for each node v to process



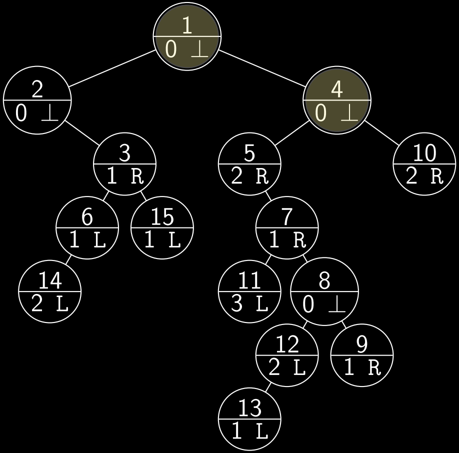
stack S

maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathsf{cla}_u, \mathsf{cra}_u)$ of all ancestors u of v.

$S = \{$

$\quad \mathrm{lce}(\mathsf{cra}_1, \mathsf{cla}_1) = 0,$

$\}$



stack S

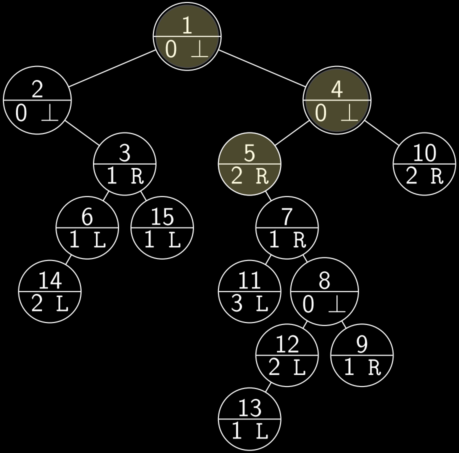
maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathsf{cla}_u, \mathsf{cra}_u)$ of all ancestors u of v.

$$S = \{$$

$$\mathrm{lce}(\mathsf{cra}_1, \mathsf{cla}_1) = 0,$$

$$\mathrm{lce}(\mathsf{cra}_4, \mathsf{cla}_4) = 0,$$

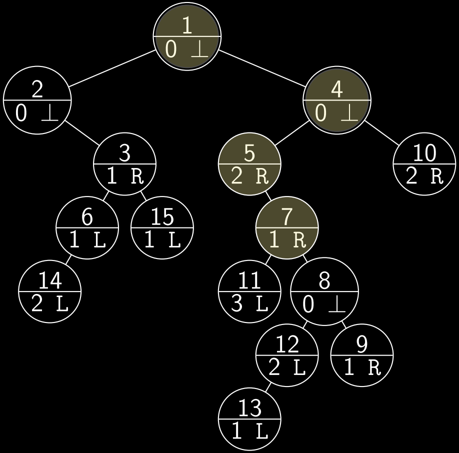
$$\}$$



stack S

maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathsf{cla}_u, \mathsf{cra}_u)$ of all ancestors u of v.

$$S = \{ \quad \mathrm{lce}(\mathsf{cra}_1, \mathsf{cla}_1) = 0, \quad \mathrm{lce}(\mathsf{cra}_4, \mathsf{cla}_4) = 0, \quad \mathrm{lce}(\mathsf{cra}_5, \mathsf{cla}_5) = 0, \quad \}$$



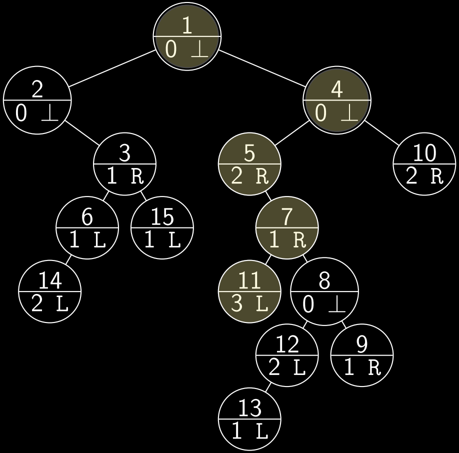
stack S

maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathsf{cla}_u, \mathsf{cra}_u)$ of all ancestors u of v.

$$S = \{$$

$$\mathrm{lce}(\mathsf{cra}_1, \mathsf{cla}_1) = 0,$$
$$\mathrm{lce}(\mathsf{cra}_4, \mathsf{cla}_4) = 0,$$
$$\mathrm{lce}(\mathsf{cra}_5, \mathsf{cla}_5) = 0,$$
$$\mathrm{lce}(\mathsf{cra}_7, \mathsf{cla}_7) = 0,$$

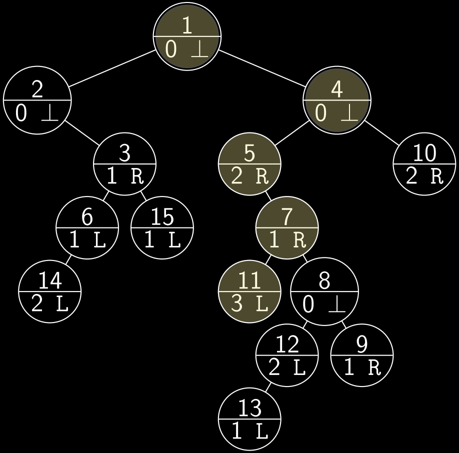
$$\}$$



stack S

maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathsf{cla}_u, \mathsf{cra}_u)$ of all ancestors u of v.

$$S = \{ \mathrm{lce}(\mathsf{cra}_1, \mathsf{cla}_1) = 0, \mathrm{lce}(\mathsf{cra}_4, \mathsf{cla}_4) = 0, \mathrm{lce}(\mathsf{cra}_5, \mathsf{cla}_5) = 0, \mathrm{lce}(\mathsf{cra}_7, \mathsf{cla}_7) = 0, \mathrm{lce}(\mathsf{cra}_{11}, \mathsf{cla}_{11}) = 1 \}$$



stack S

maintain stack S of LCE values such that, on visiting node v, S stores $\mathrm{lce}(\mathrm{cla}_u, \mathrm{cra}_u)$ of all ancestors u of v.

$$S = \{$$

$$\mathrm{lce}(\mathrm{cra}_1, \mathrm{cla}_1) = 0,$$

$$\mathrm{lce}(\mathrm{cra}_4, \mathrm{cla}_4) = 0,$$

$$\mathrm{lce}(\mathrm{cra}_5, \mathrm{cla}_5) = 0,$$

$$\mathrm{lce}(\mathrm{cra}_7, \mathrm{cla}_7) = 0,$$

$$\mathrm{lce}(\mathrm{cra}_{11}, \mathrm{cla}_{11}) = 1$$

$$\}$$

- why helpful?
- how computable?

known facts

1. $u, v, w \in [1..n]$ with $T[u..] \prec T[v..] \prec T[w..]$
   $\Rightarrow \mathrm{lce}(u, w) = \min(\mathrm{lce}(u, v), \mathrm{lce}(v, w))$
2. $T[\mathrm{cra}_v..] \prec T[v..] \prec T[\mathrm{cla}_v..]$ (assume cla_v and cra_v exist)

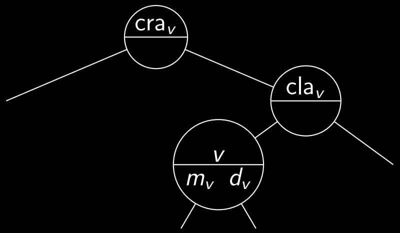
lemma

given

- $\mathrm{lce}(\mathrm{cla}_v, \mathrm{cra}_v)$ and
- $m_v = \mathrm{lce}(v, \mathrm{ca}_v)$,

we can compute

- $\mathrm{lce}(v, \mathrm{cla}_v)$ and
- $\mathrm{lce}(v, \mathrm{cra}_v)$ in constant time.



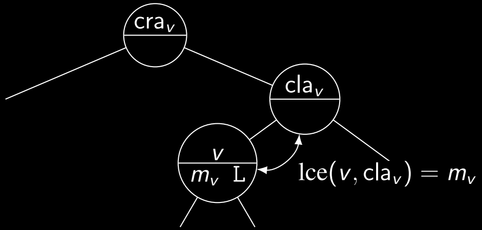
proof of lemma

- wlog., $d_v = \mathtt{L}$, and cla_v and cra_v exist

$\Rightarrow$ $\mathsf{ca}_v = \mathsf{cla}_v$

hence:

- $\mathrm{lce}(v, \mathsf{cla}_v) = \mathrm{lce}(v, \mathsf{ca}_v) = m_v$
- $\mathrm{lce}(v, \mathsf{cra}_v) = \mathrm{lce}(\mathsf{cla}_v, \mathsf{cra}_v)$



the latter is because of Facts 1 and 2:

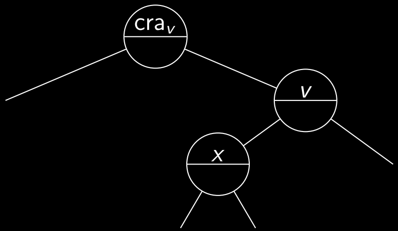
$$\begin{aligned}\mathrm{lce}(\mathsf{cra}_v, \mathsf{cla}_v) &= \min(\mathrm{lce}(v, \mathsf{cra}_v), \mathrm{lce}(v, \mathsf{cla}_v)) \\ &= \mathrm{lce}(v, \mathsf{cra}_v) \leq \mathrm{lce}(v, \mathsf{cla}_v)\end{aligned}$$

$\square$

corollary: how to compute stack S

given:

- value $\mathrm{lce}(\mathsf{cla}_v, \mathsf{cra}_v)$
- x: v's left child



then:

- $\mathsf{cla}_x = v$ and $\mathsf{cra}_x = \mathsf{cra}_v$

$\Rightarrow$ $\mathrm{lce}(\mathsf{cla}_x, \mathsf{cra}_x) = \mathrm{lce}(v, \mathsf{cra}_v)$ computable in constant time by lemma

(right child analogously by symmetry)

$\Rightarrow$ can maintain stack S during a top-down traversal in constant time per node.

subarray extraction

can compute $\mathrm{SLCP}[\ell, r]$ in $\mathcal{O}(h + (r - \ell))$ time, where h is the tree's height.

- augment tree with subtree sizes
- can find node ℓ by top-down traversal (while maintaining S)
- can start in-order traversal at node ℓ
- stop traversal when arriving at node r
- number of visited nodes is $\mathcal{O}(h) + r - \ell$, and each node is processed in constant time.

summary

suffix binary search tree by Irving and Love'03

- maintains ranks of m suffixes
- $\mathcal{O}(m)$ space (each node stores 2 integers + 1 bit)
- construction needs $\mathcal{O}(mh)$ LCE queries (h: height)
- can be made balanced ($h = \mathcal{O}(\lg m)$)
- used for sparse suffix sorting by Fischer+'20
  - $\mathcal{O}(c(\sqrt{\lg \sigma} + \lg \lg n) + m \lg m \lg n \lg^* n)$ time
  - c: lower bound on number of characters needed to compare
  - $\mathcal{O}(m)$ space

(n : text length, σ: alphabet size)

our contribution: can extract

- $\mathsf{SSA}[i..i+\ell-1]$ and
- $\mathsf{SLCP}[i..i+\ell-1]$ in $\mathcal{O}(h+\ell)$ time

any questions are always very welcome!

open problems

- memory-efficient representations of suffix binary search trees?
- time-efficient implementation via B trees
  - balanced by construction
  - B+ variants have good memory locality
- can we merge two trees efficiently?