Position Heaps for Cartesian-tree Matching on Strings and Tries

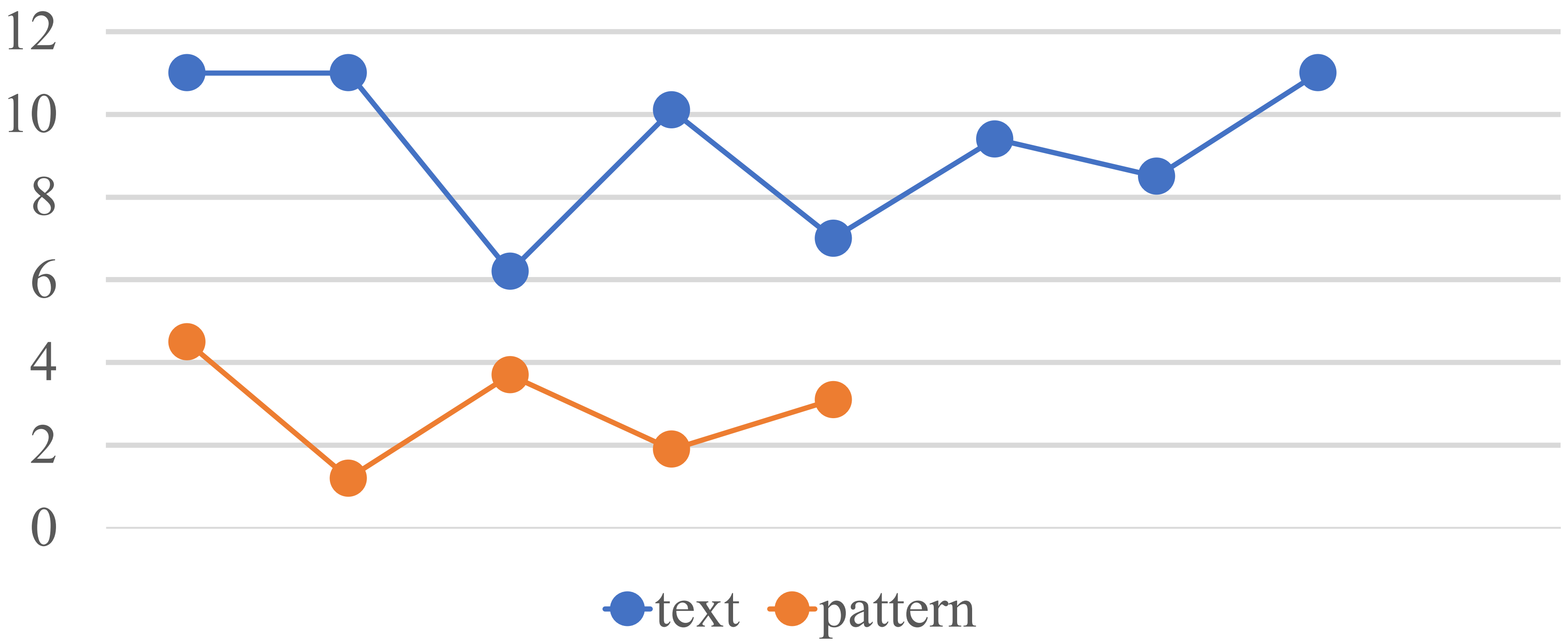
SPIRE 2021

Akio Nishimoto, Noriki Fujisato, Yuto Nakashima,
and Shunsuke Inenaga

Kyushu University, Japan

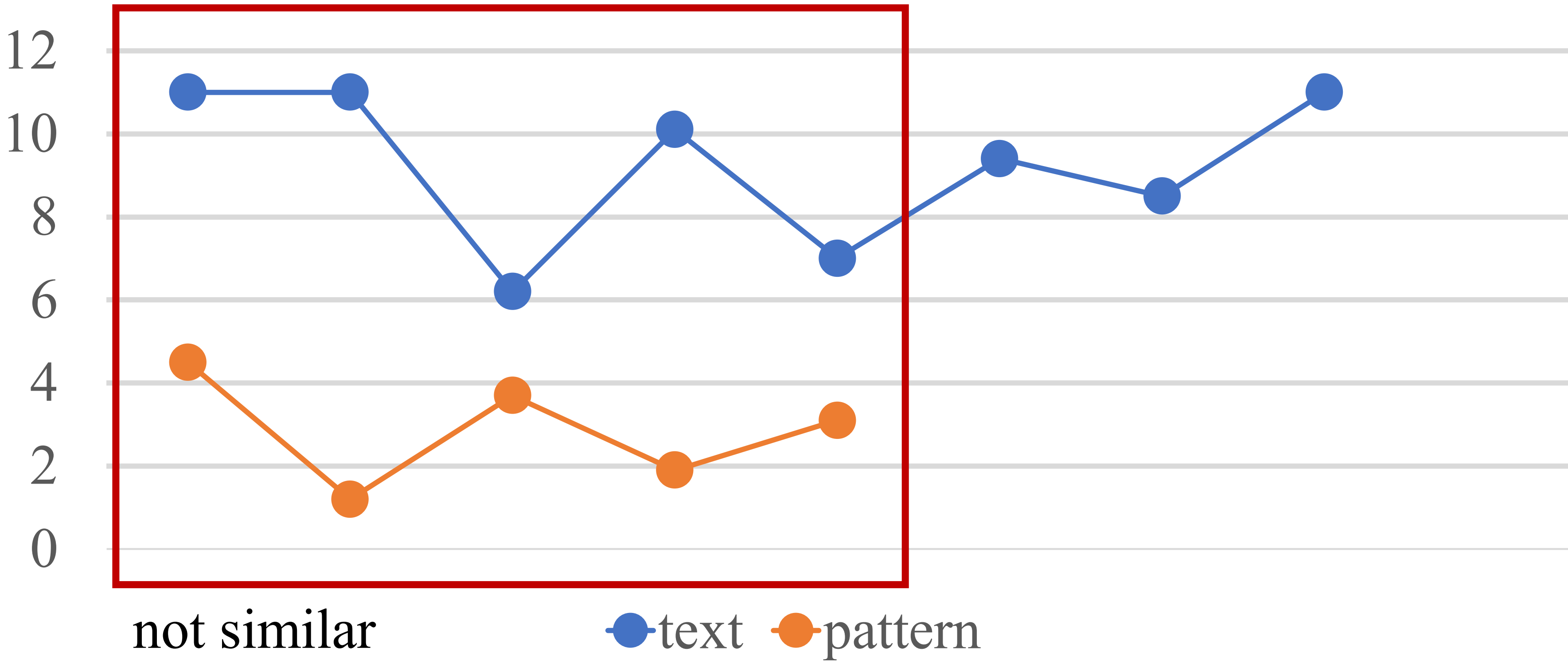
Background

We want to find substrings of a text that have similar structures as a pattern.



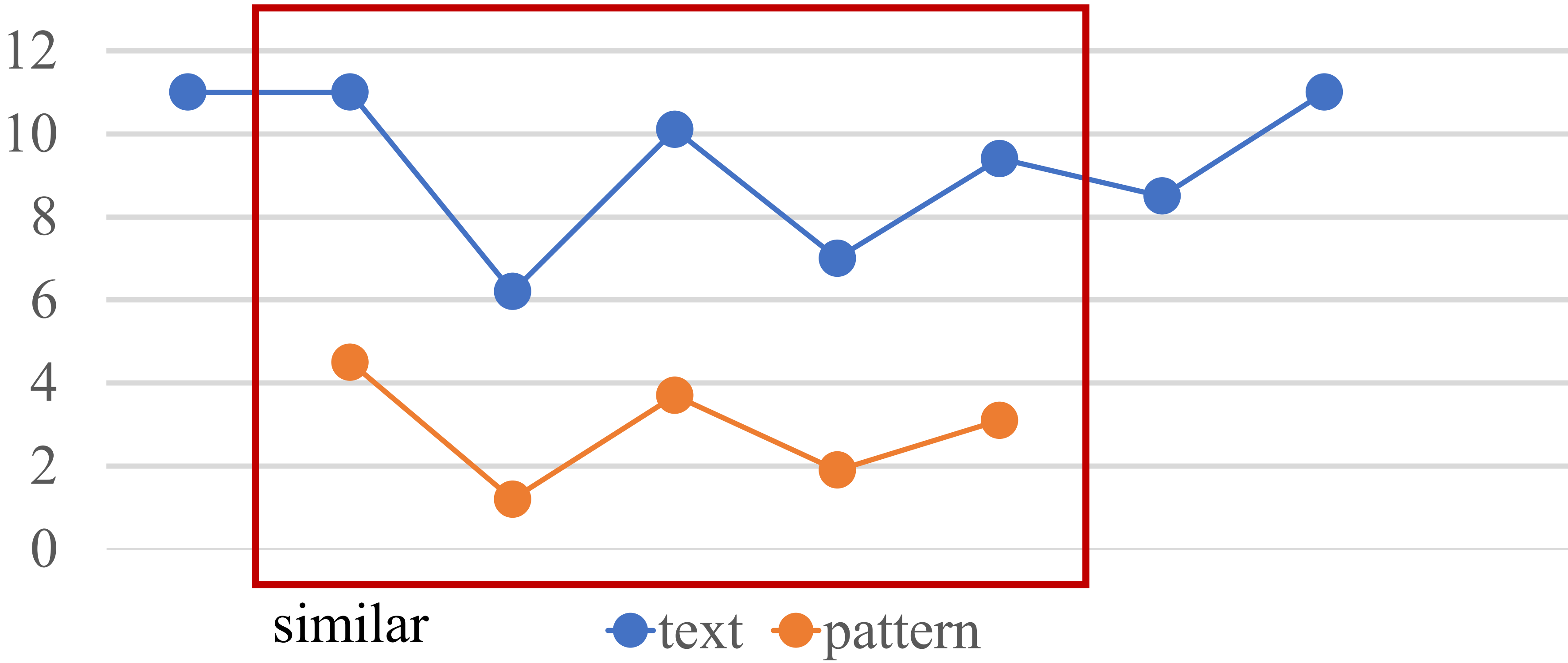
Background

We want to find substrings of a text that have similar structures as a pattern.



Background

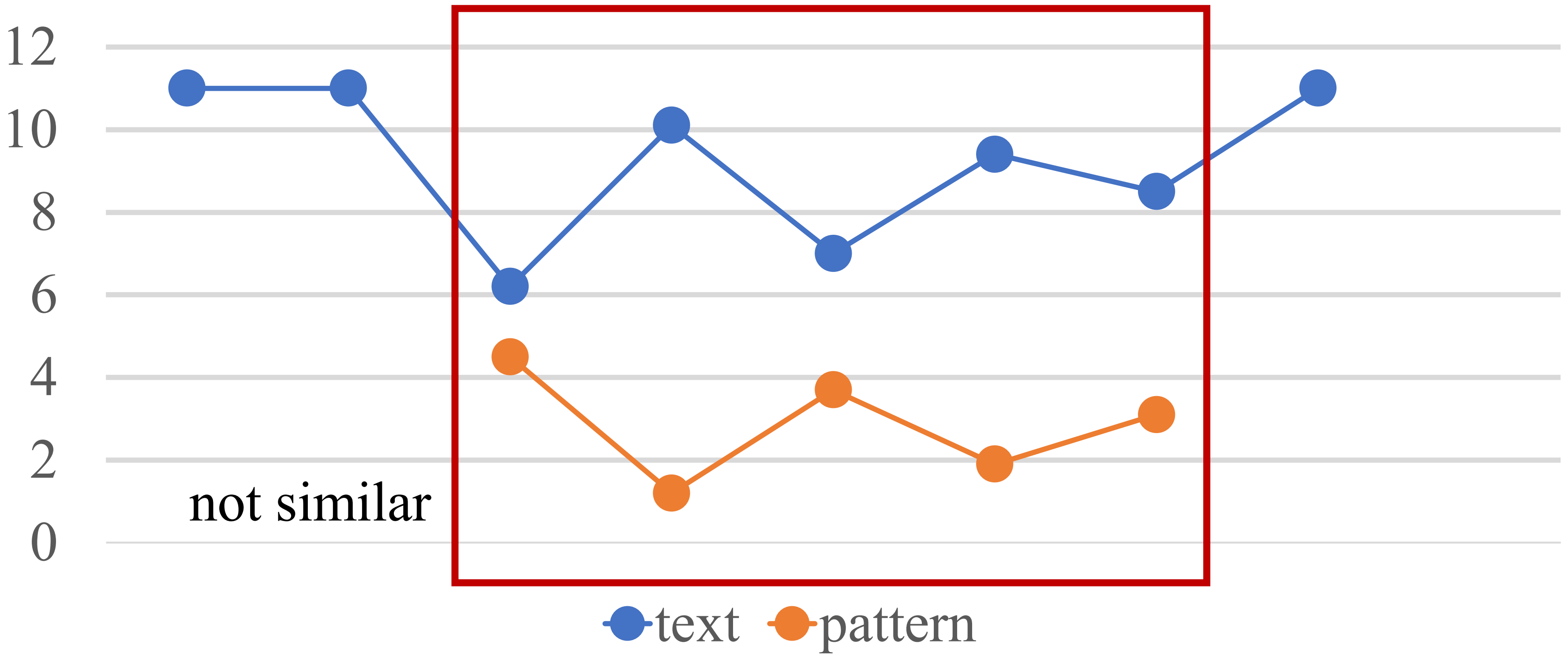
We want to find substrings of a text that have similar structures as a pattern.



This similar substring can be found by order-preserving matching [Kim et al. 2013].

Background

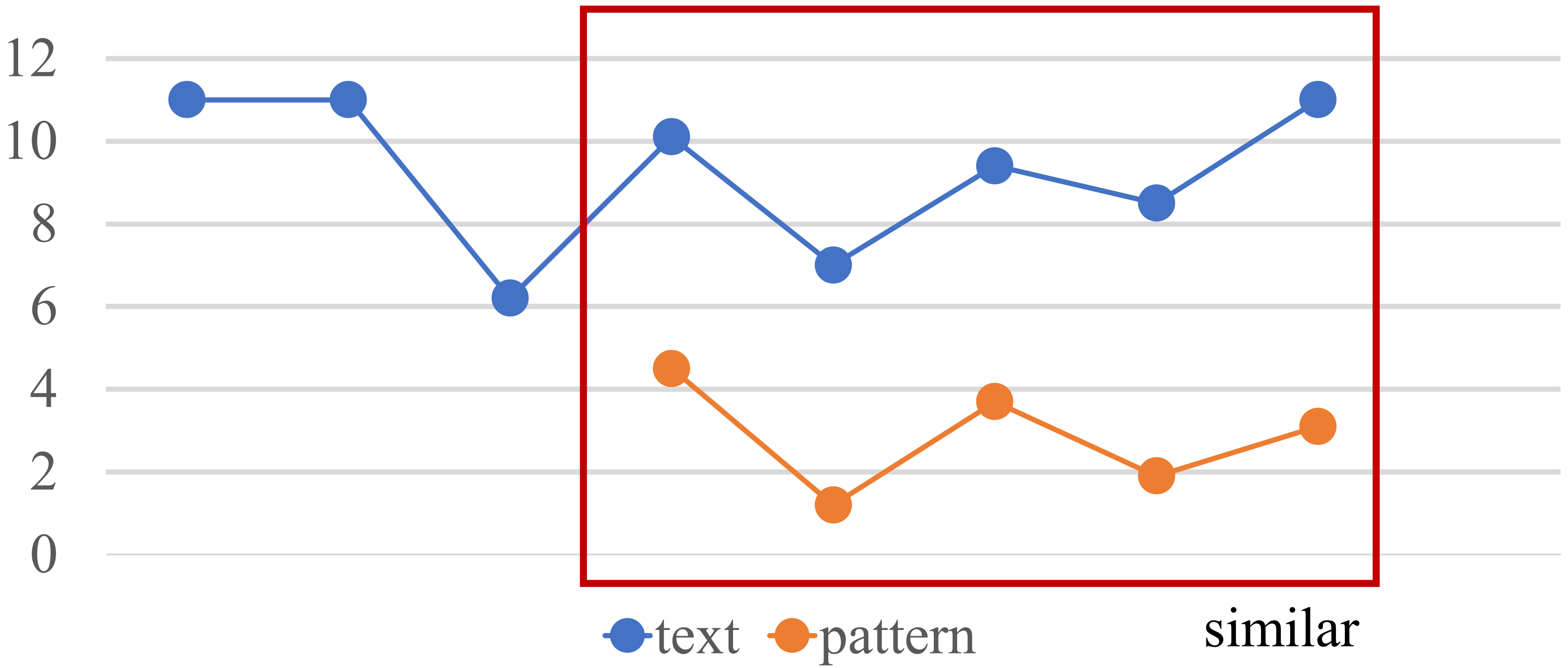
We want to find substrings of a text that have similar structures as a pattern.



We can find this substring for order-preserving matching.

Background

We want to find substrings of a text that have similar structures as a pattern.



We cannot find this substring for order-preserved matching.
But, we can use Cartesian-tree matching [Park et al., 2019].

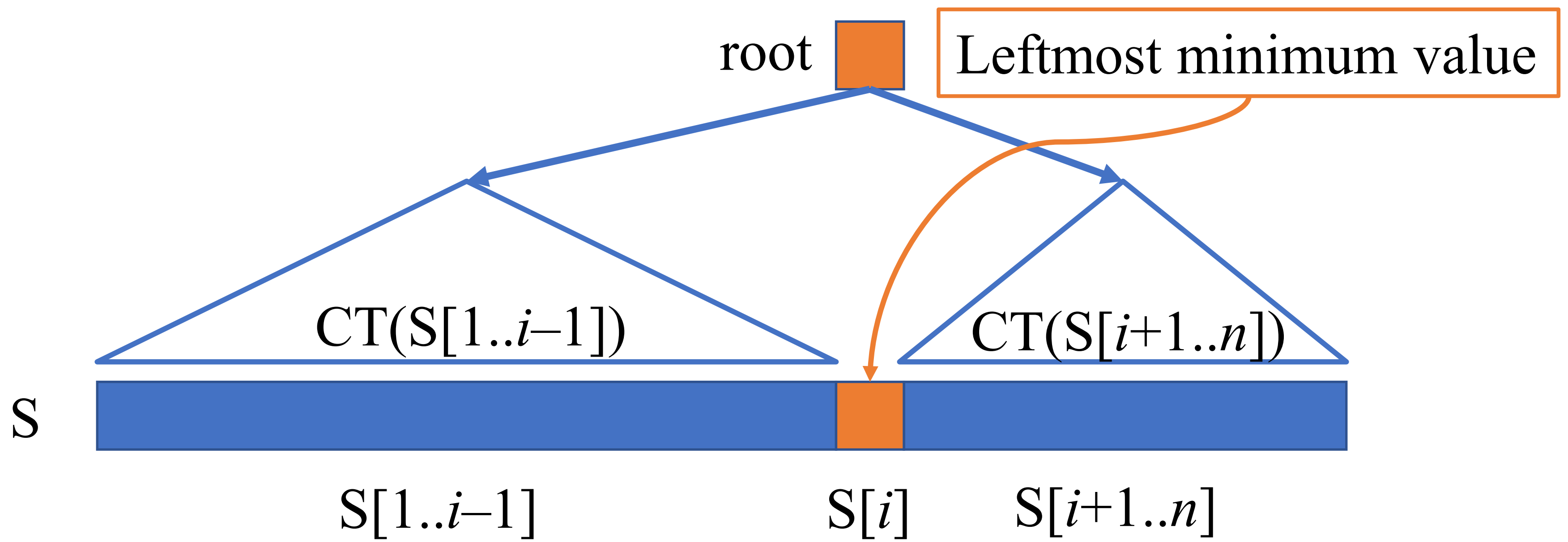
Definition : Cartesian-tree

The Cartesian-tree of a string S, denoted CT(S), is the rooted tree which is recursively defined as follows.

Each character in string S corresponds to a node in CT(S).

If S[i] is the leftmost minimum value of string S,

- S[i] is root node,
- the left subtree is CT(S[1..i–1]) and,
- the right subtree is CT(S[i+1..n]).



Example : Cartesian-tree

S = 2513164

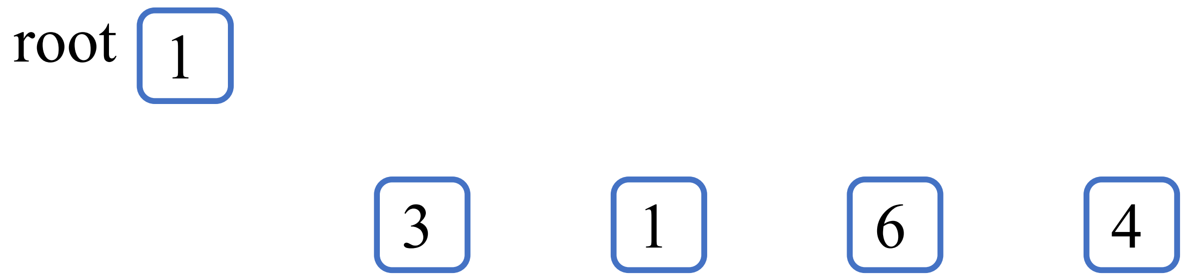
The minimum value of string S is 1.
We choose the leftmost 1.

2 5 1 3 1 6 4

Example : Cartesian-tree

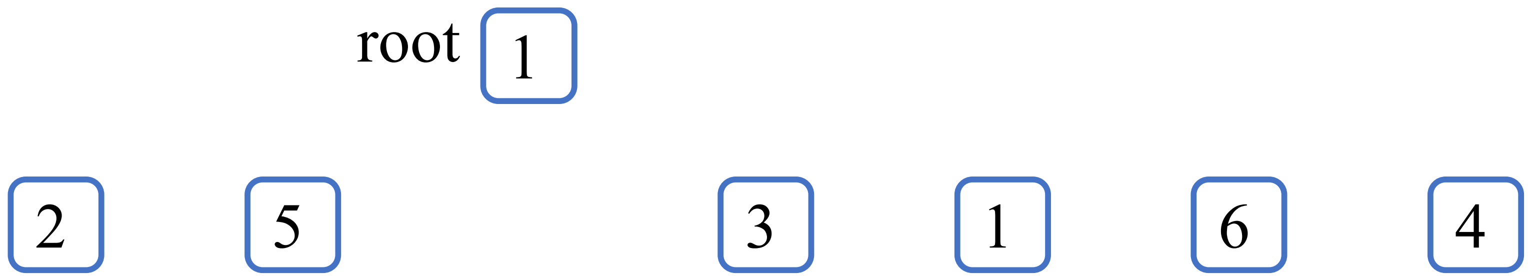
S = 2513164

The minimum value of string S is 1.
We choose the leftmost 1.



Example : Cartesian-tree

S = 2513164

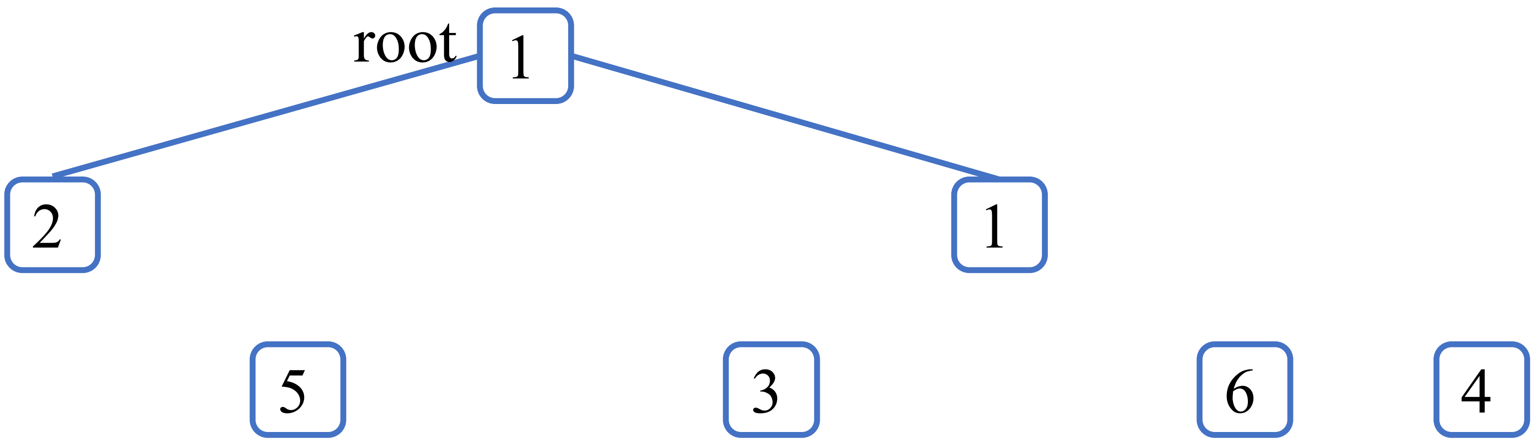


The minimum value
of string 25 is 2.
We choose the leftmost 2.

The minimum value
of string 3164 is 1.
We choose the leftmost 1.

Example : Cartesian-tree

S = 2513164



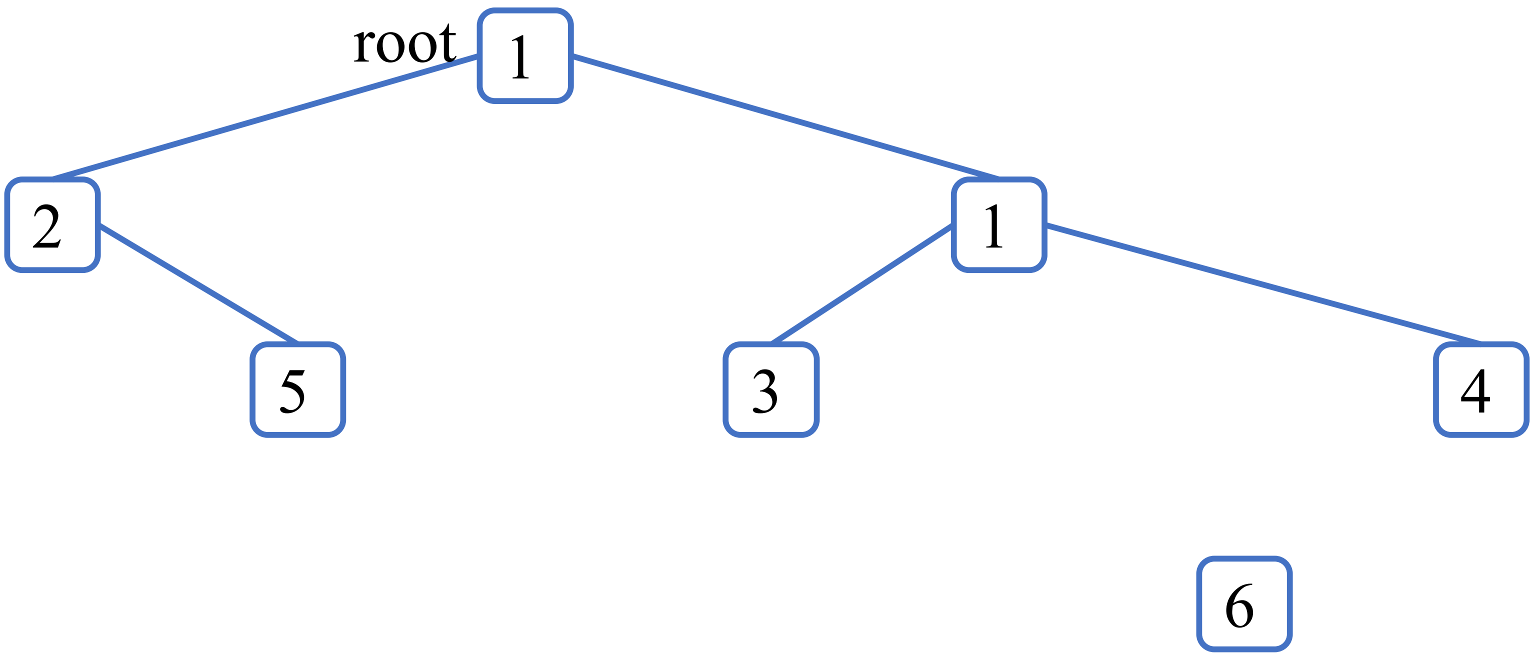
The minimum value of string 25 is 2. We choose the leftmost 2.

The minimum value of string 3164 is 1. We choose the leftmost 1.

Example : Cartesian-tree

S = 2513164

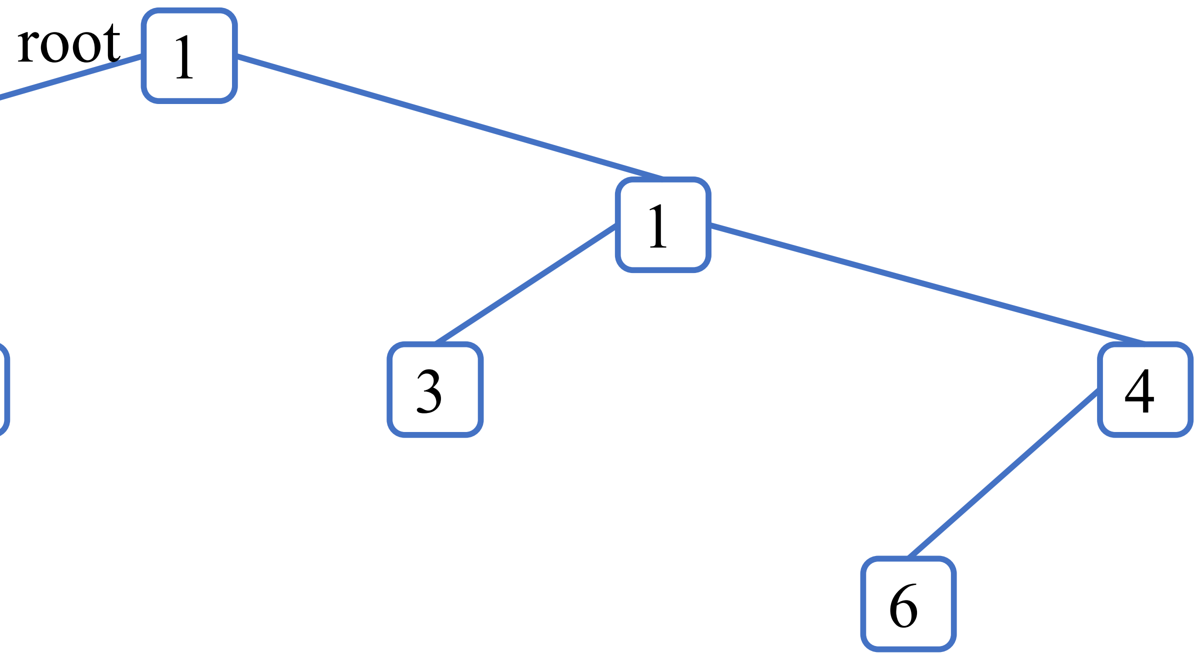
We continue recursively.



Example : Cartesian-tree

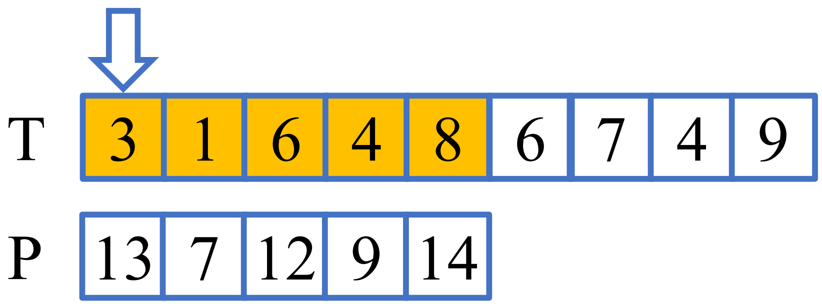
S = 2513164

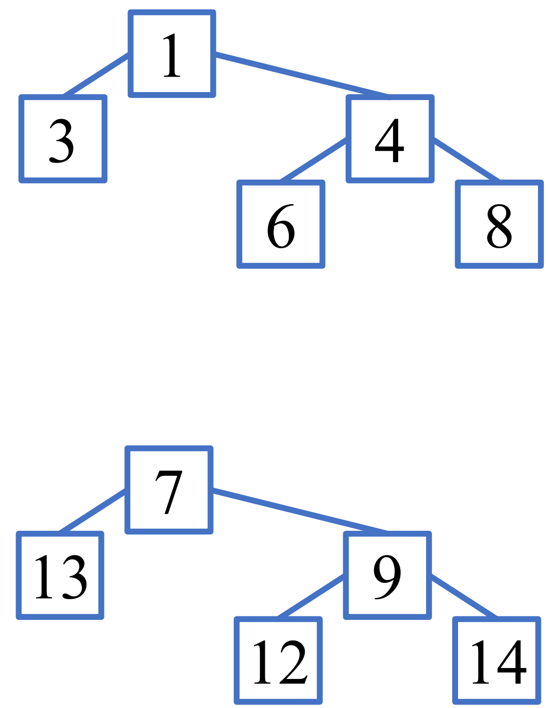
We continue recursively.



Cartesian-tree matching [Park et al., 2019]

Cartesian-tree matching is the problem of finding all positions i, such that the Cartesian-tree of substring $T[i,..,i + m - 1]$ and that of pattern P are isomorphic.

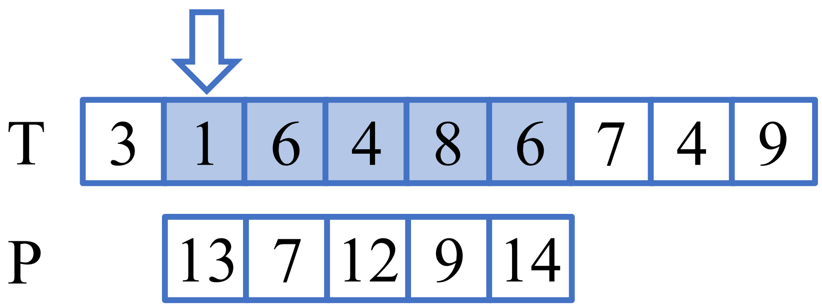




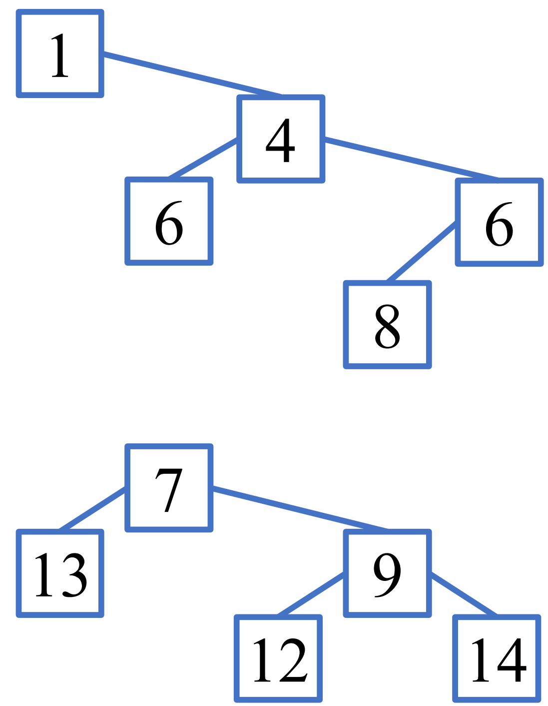
Output : 1

Cartesian-tree matching [Park et al., 2019]

Cartesian-tree matching is the problem of finding all positions i, such that the Cartesian-tree of substring T[i,..,$i + m - 1$] and that of pattern P are isomorphic.

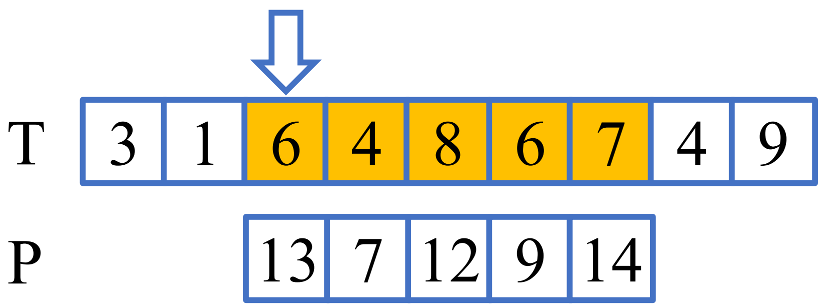


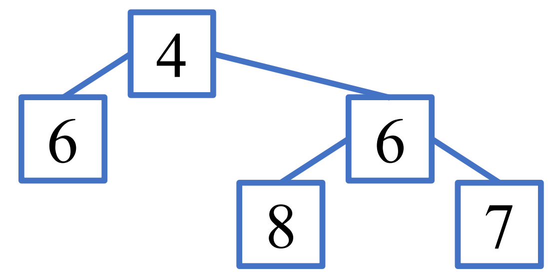
Output : 1



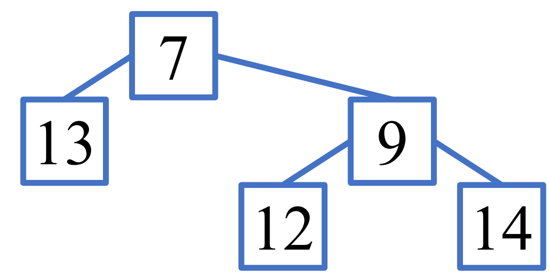
Cartesian-tree matching [Park et al., 2019]

Cartesian-tree matching is the problem of finding all positions i, such that the Cartesian-tree of substring T[i,..,$i + m - 1$] and that of pattern P are isomorphic.



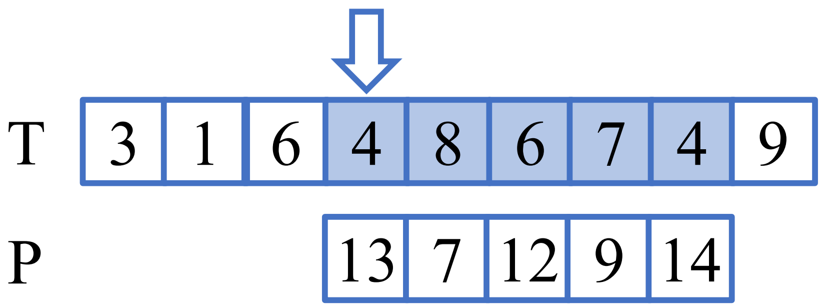


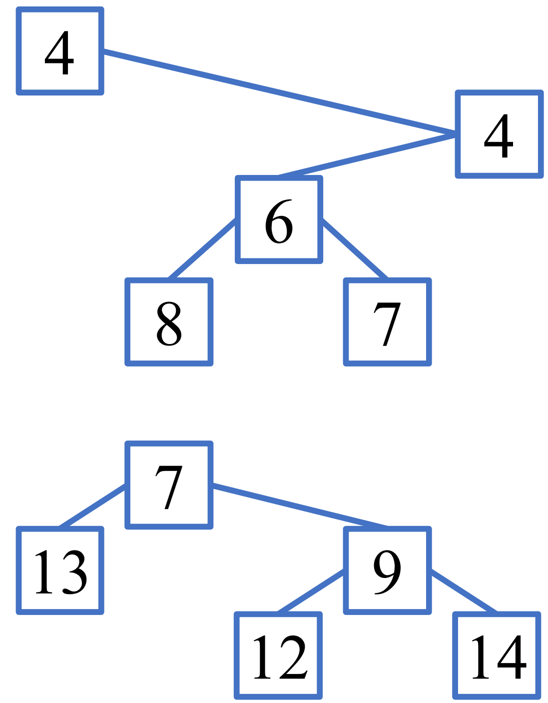
Output : 1,3



Cartesian-tree matching [Park et al., 2019]

Cartesian-tree matching is the problem of finding all positions i, such that the Cartesian-tree of substring T[i,..,$i + m - 1$] and that of pattern P are isomorphic.

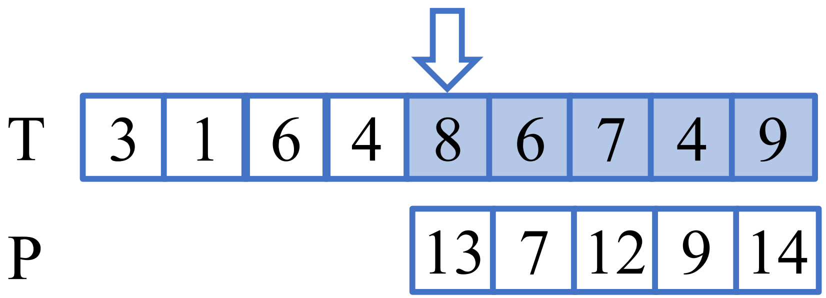


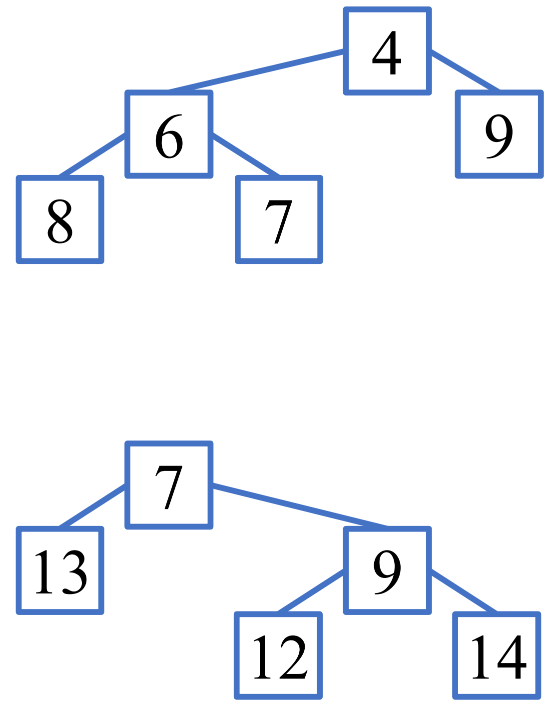


Output : 1,3

Cartesian-tree matching [Park et al., 2019]

Cartesian-tree matching is the problem of finding all positions *i*, such that the Cartesian-tree of substring T[i,..,$i+m-1$] and that of pattern P are isomorphic.





Output : 1,3

Existing indexing structures

Data structure	Const. time	Pattern locating time	space
Cartesian Suffix Tree [Park et al., 2020]	$O(n \log n)$	$O(m \log n + occ)$	$O(n)$ words
Succinct Index [Kim and Cho, 2021]	$O(n \log n)$	$O(m \cdot occ)$	$3n + o(n)$ bits

n is text length, σ is alphabet size, occ is number of pattern occurrences, m is pattern length.

Our Contribution 1

Data structure	**Const. time**	**Pattern locating time**	**space**
Cartesian Suffix Tree [Park et al., 2020]	$O(n \log n)$	$O(m \log n + occ)$	$O(n)$ words
Succinct Index [Kim and Cho, 2021]	$O(n \log n)$	$O(m \cdot occ)$	$3n + o(n)$ bits
Cartesian Position Heap [This work]	$\boldsymbol{O(n \log \sigma)}$	$\boldsymbol{O(m(\sigma + \log(\min(h, m))) + occ)}$	$\boldsymbol{O(n)}$ **words**

n is text length, σ is alphabet size, occ is number of pattern occurrences, m is pattern length , h is height of Cartesian Position Heap.

Our Contribution 2

Data structure	Const. time	Matching time	space
Cartesian Suffix Tree [Park et al., 2020]	$O(n \log n)$	$O(m \log n + occ)$	$O(n)$ words
Succinct Index [Kim and Cho, 2021]	$O(n \log n)$	$O(m \cdot occ)$	$3n + o(n)$ bits
Cartesian Position Heap [This work]	$\boldsymbol{O(n \log \sigma)}$	$\boldsymbol{O(m(\sigma + \log(\min(h, m))) + occ)}$	$\boldsymbol{O(n)}$ **words**

Cartesian Position Heap for trie [This work]	$\boldsymbol{O(N\sigma)}$	$\boldsymbol{O(m(\sigma^2 + \log(\min(h,m))) + occ)}$	$\boldsymbol{O(N\sigma)}$ **words**

n is text string length, σ is alphabet size, occ is number of pattern occurrences, m is pattern length, h is height of Cartesian Position Heap, N is text trie size.

Our Contribution 2

Data structure	Const. time	Matching time	space
Cartesian Suffix Tree [Park et al., 2020]	$O(n \log n)$	$O(m \log n + occ)$	$O(n)$ words
Succinct Index [Kim and Cho, 2021]	$O(n \log n)$	$O(m \cdot occ)$	$3n + o(n)$ bits
Cartesian Position Heap [This work]	$\boldsymbol{O(n \log \sigma)}$	$\boldsymbol{O(m(\sigma + \log(\min(h, m))) + occ)}$	$\boldsymbol{O(n)}$ **words**

Cartesian Position Heap for trie [This work]	$\boldsymbol{O(N\sigma)}$	$\boldsymbol{O(m(\sigma^2 + \log(\min(h,m))) + occ)}$	$\boldsymbol{O(N\sigma)}$ **words**

n is text string length, σ is alphabet size, occ is number of pattern occurrences, m is pattern length, h is height of Cartesian Position Heap, N is text trie size.

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[i] is the distance to the largest position in S[1..i–1] which has a value less than or equal to S[i].
- If such a position does not exist, then PD(S)[i] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)										

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[i] is the distance to the largest position in S[1..i–1] which has a value less than or equal to S[i].
- If such a position does not exist, then PD(S)[i] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)										

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value less than or equal to S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)	0									

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value less than or equal to S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)	0									

PD encoding [Park et al., 2019]

Definition of PD encoding

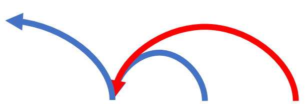
- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value **less than** or equal to S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)	0	1								

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value **less than** or equal to S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.

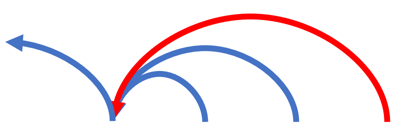


S	3	6	4	3	2	8	5	6	4	3
PD(S)	0	1	2							

PD encoding [Park et al., 2019]

Definition of PD encoding

- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value less than or **equal to** S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.



S	3	6	4	3	2	8	5	6	4	3
PD(S)	0	1	2	3						

PD encoding [Park et al., 2019]

Definition of PD encoding

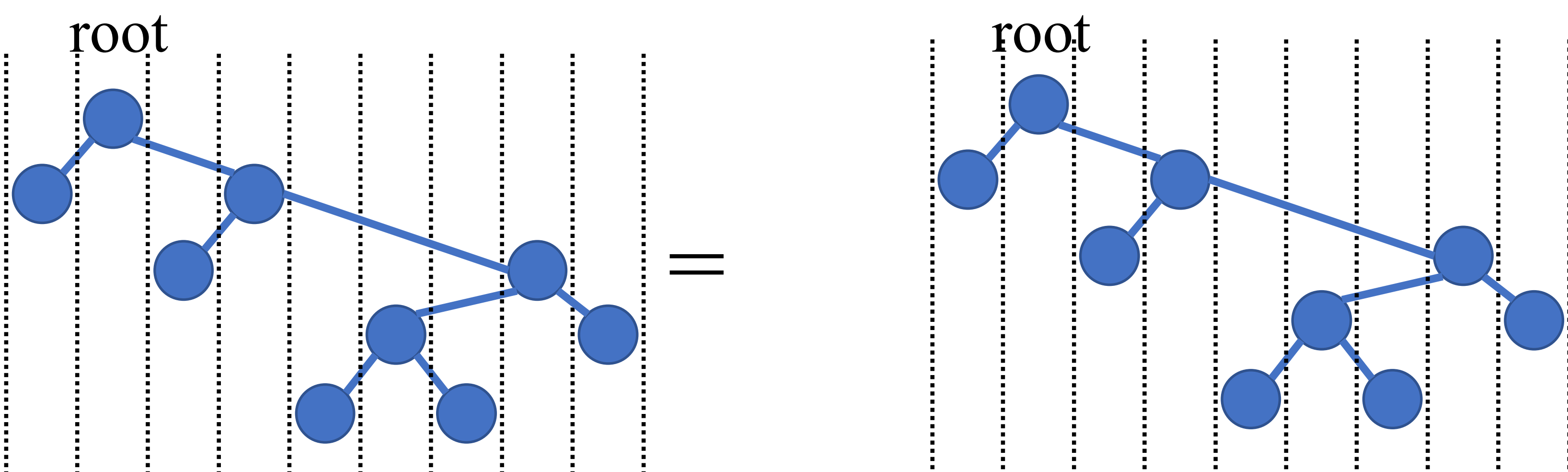
- PD(S)[*i*] is the distance to the largest position in S[1..*i*–1] which has a value less than or equal to S[*i*].
- If such a position does not exist, then PD(S)[*i*] = 0.

S	3	6	4	3	2	8	5	6	4	3
PD(S)	0	1	2	3	0	1	2	1	4	5

Relation between CT and PD

Lemma [Park et al., 2019]

For any strings S_1 and S_2 ,
$CT(S_1) = CT(S_2) \Leftrightarrow PD(S_1) = PD(S_2)$



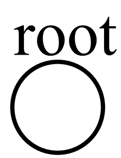
S_1	3	1	6	4	8	6	7	5	9
$PD(S_1)$	0	0	1	2	1	2	1	4	1

=

S_2	7	1	3	2	8	6	9	4	5
$PD(S_2)$	0	0	1	2	1	2	1	4	1

Cartesian Position Heap (CPH)

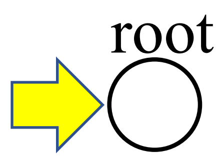
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

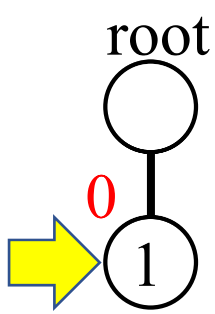
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

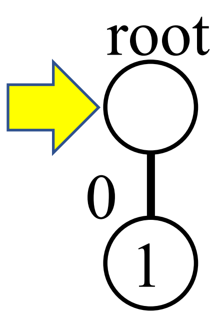
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

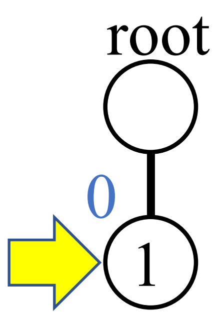
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

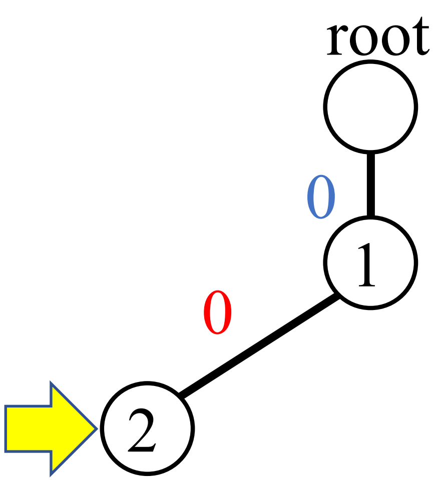
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with $w_k = PD(T_k)$ and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

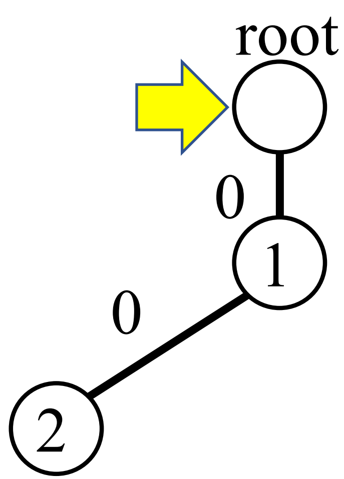
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with $w_k = PD(T_k)$ and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

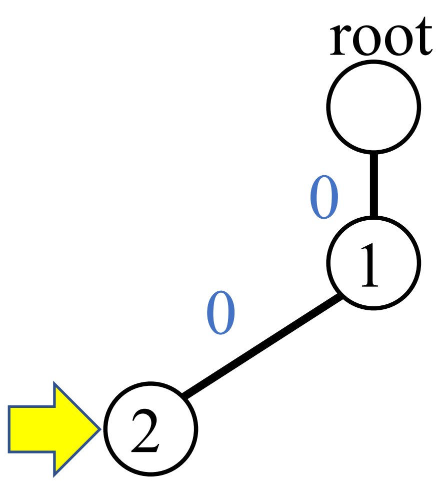
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

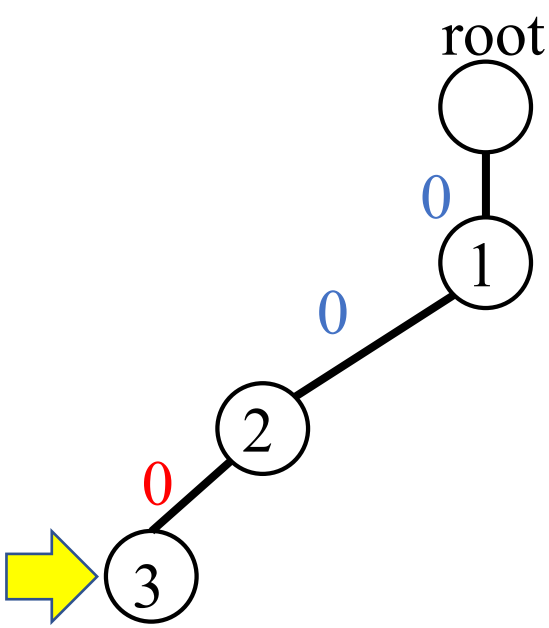
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

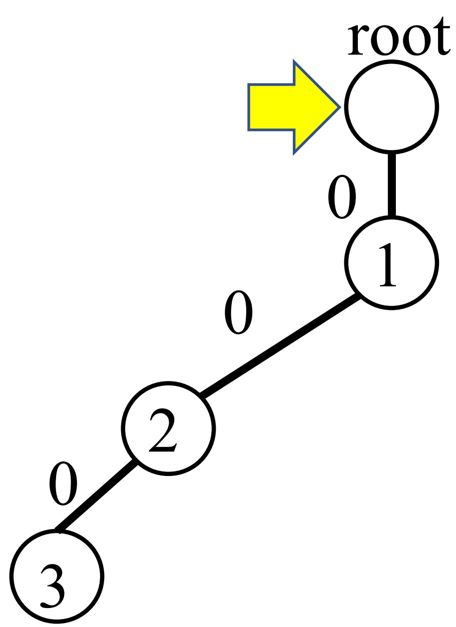
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

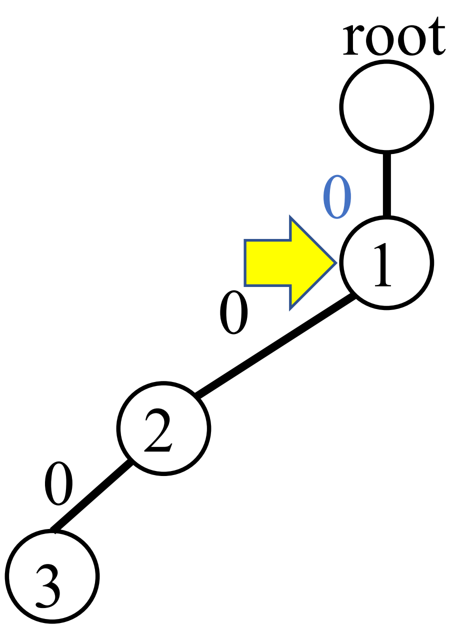
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

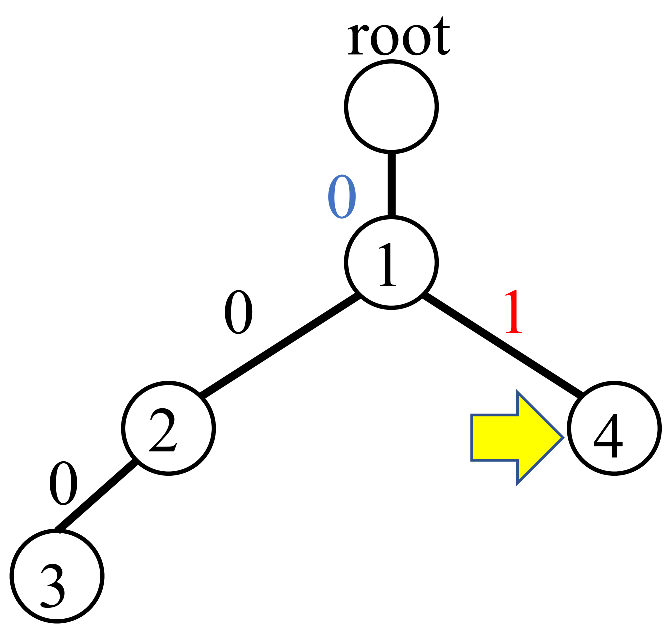
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

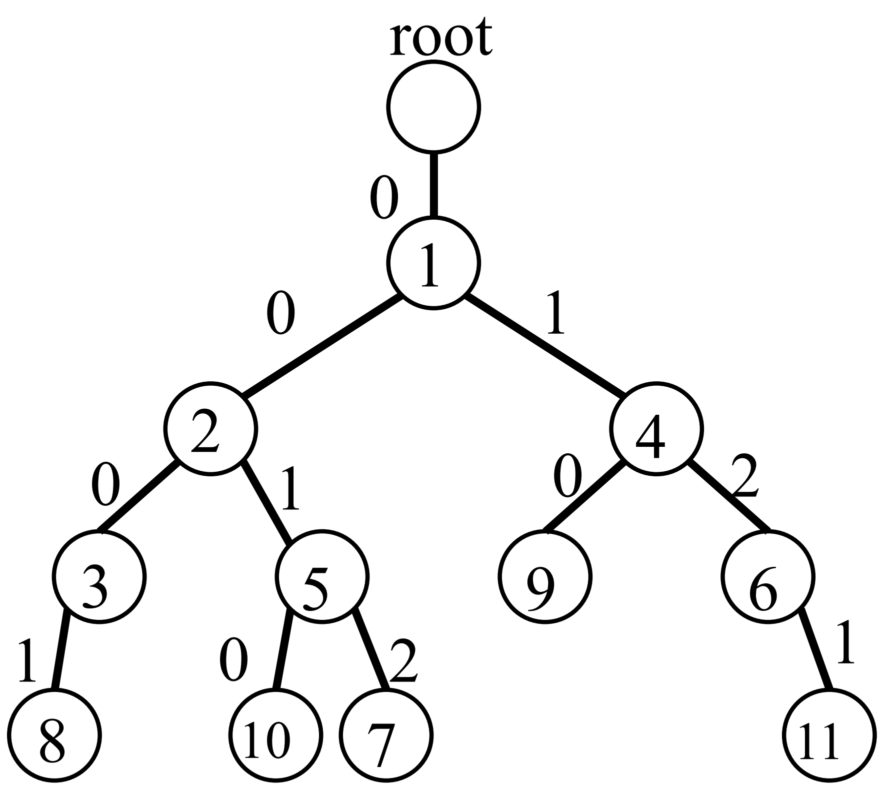
For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with $w_k = PD(T_k)$ and insert the next character after the traversal, where T_k is the suffix of T of length k.



T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Cartesian Position Heap (CPH)

For increasing $k = 1, \ldots, n$, traverse CPH(T_{k-1}) with w_k = PD(T_k) and insert the next character after the traversal, where T_k is the suffix of T of length k.

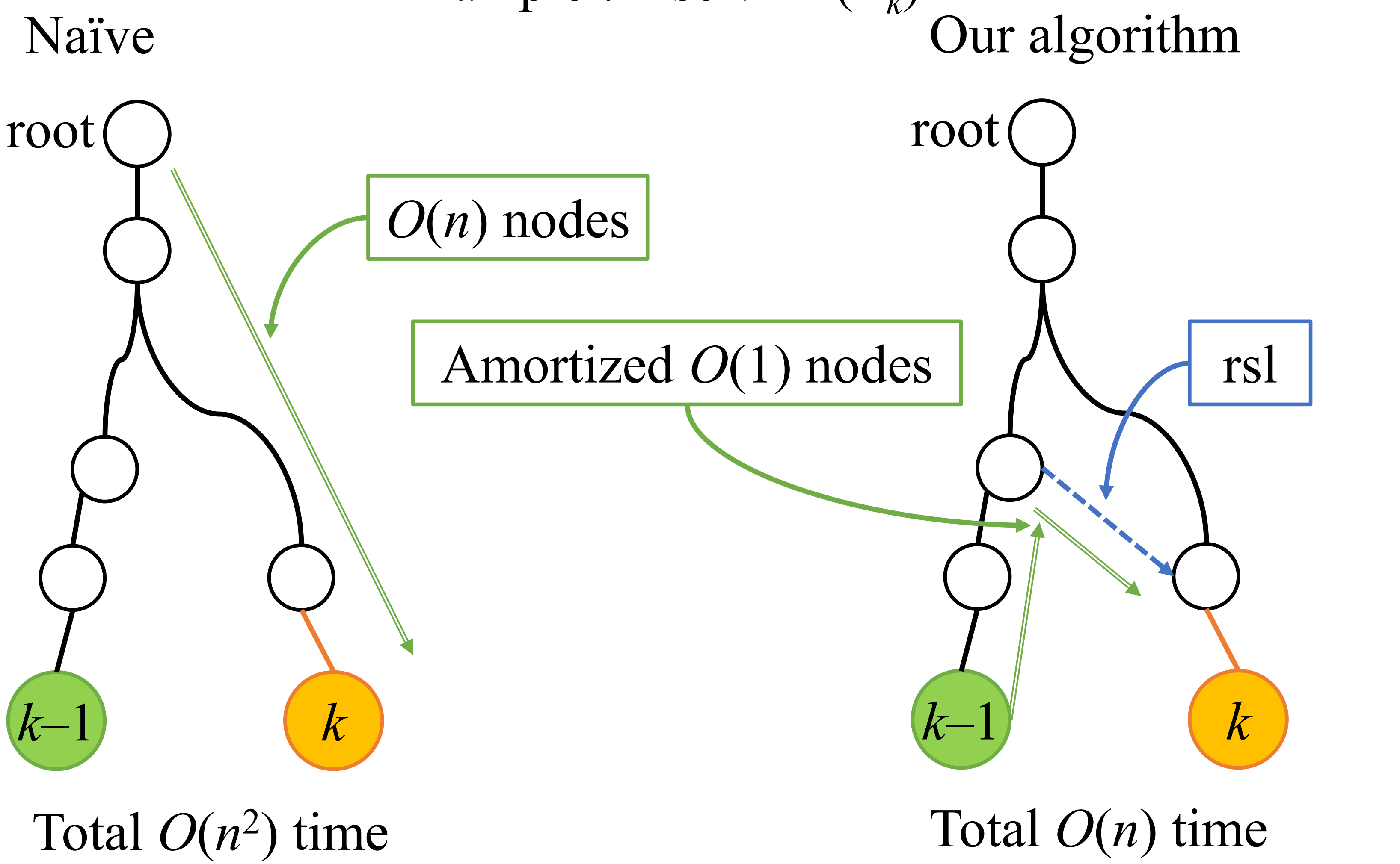


T	27584365741
w_1	0
w_2	00
w_3	000
w_4	0100
w_5	00100
w_6	012140
w_7	0012140
w_8	00012140
w_9	010012140
w_{10}	0010012140
w_{11}	01214512140

Construction of CPH on string

We use reverse suffix link (rsl) instead of naïve traversal.

Example : insert PD(T_k)



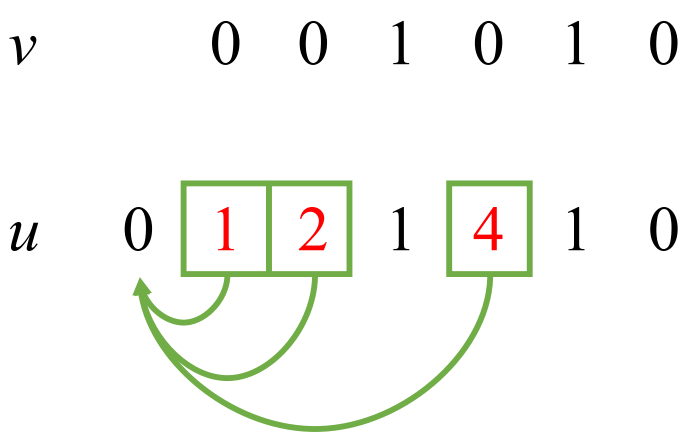
reverse suffix link on CPH

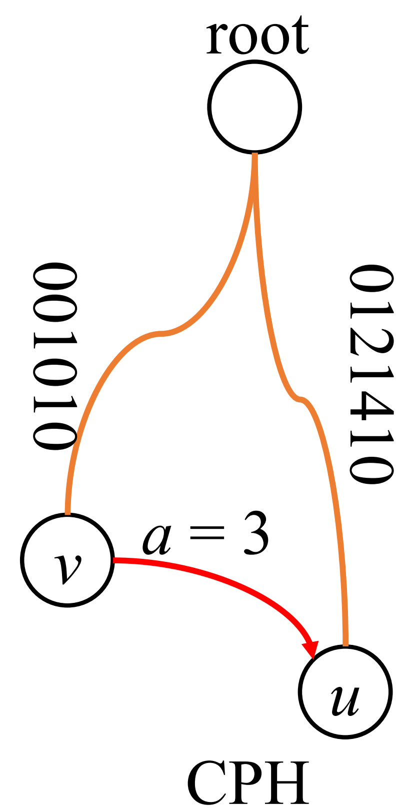
Let u be a node of CPH, and let a be the number of the pointers representing the PD encoding which point to $u[1]$.

Then, there exists a node v such that v is obtained by removing $u[1]$ from u and chaining the first a 0's in $u[2..|u|]$.

We set rsl from v to u with label a.

Example : $a = 3$





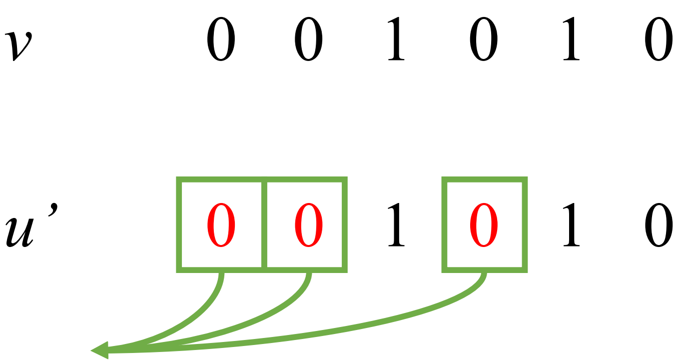
reverse suffix link on CPH

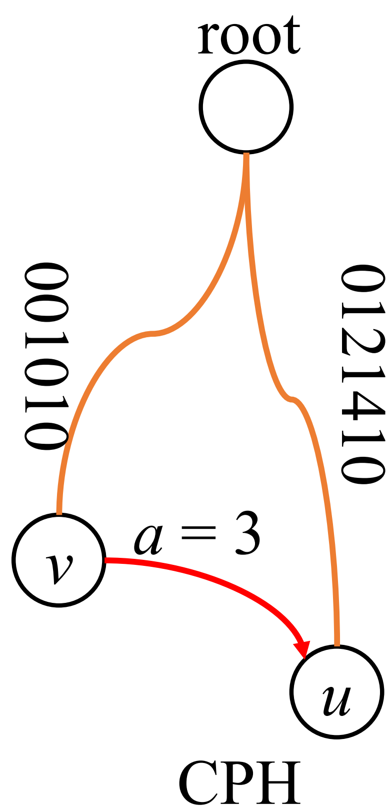
Let u be a node of CPH, and let a be the number of the pointers representing the PD encoding which point to $u[1]$.

Then, there exists a node v such that v is obtained by removing $u[1]$ from u and chaining the first a 0's in $u[2..|u|]$.

We set rsl from v to u with label a.

Example : $a = 3$





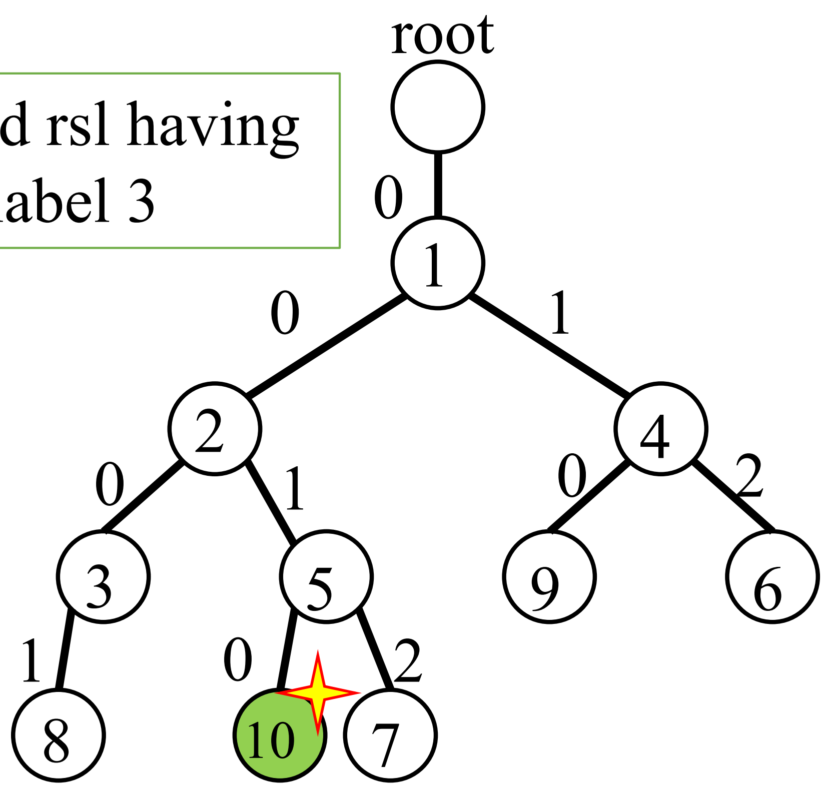
Constructing CPH for string ($k \geq 2$)

We traverse CPH(T_{k-1}) from node k–1 towards the root and find the deepest node which has rsl with appropriate label a.

$k = 11$

T	27584365741
w_{10}	0010012140
w_{11}	01214512140

To find rsl having label 3

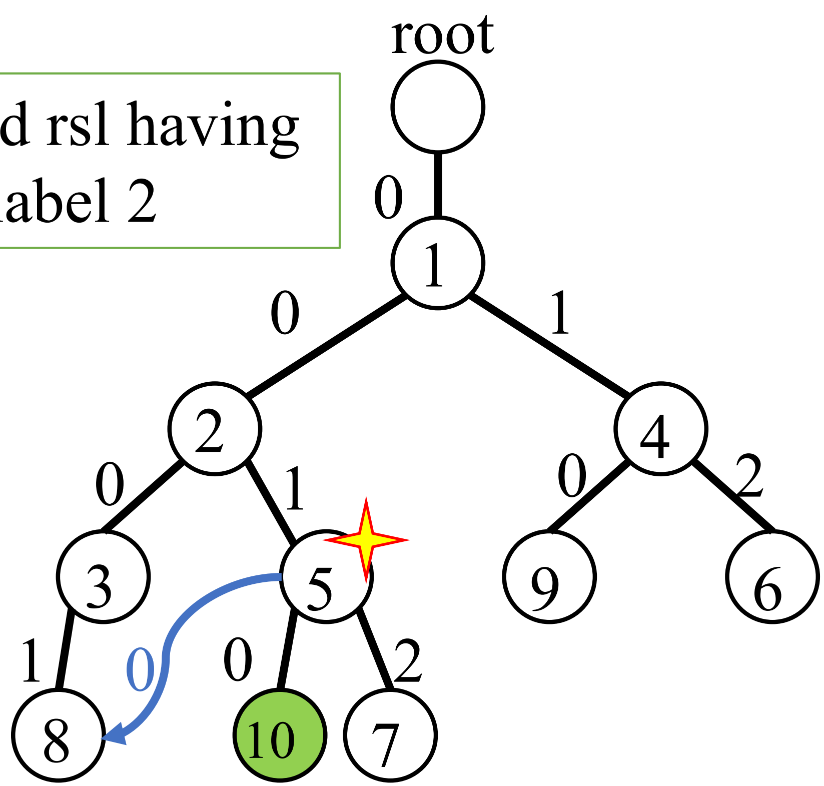


The number of changed positions is 3.

Constructing CPH for string ($k \geq 2$)

We traverse CPH(T_{k-1}) from node k–1 towards the root and find the deepest node which has rsl with appropriate label a.

To find rsl having label 2



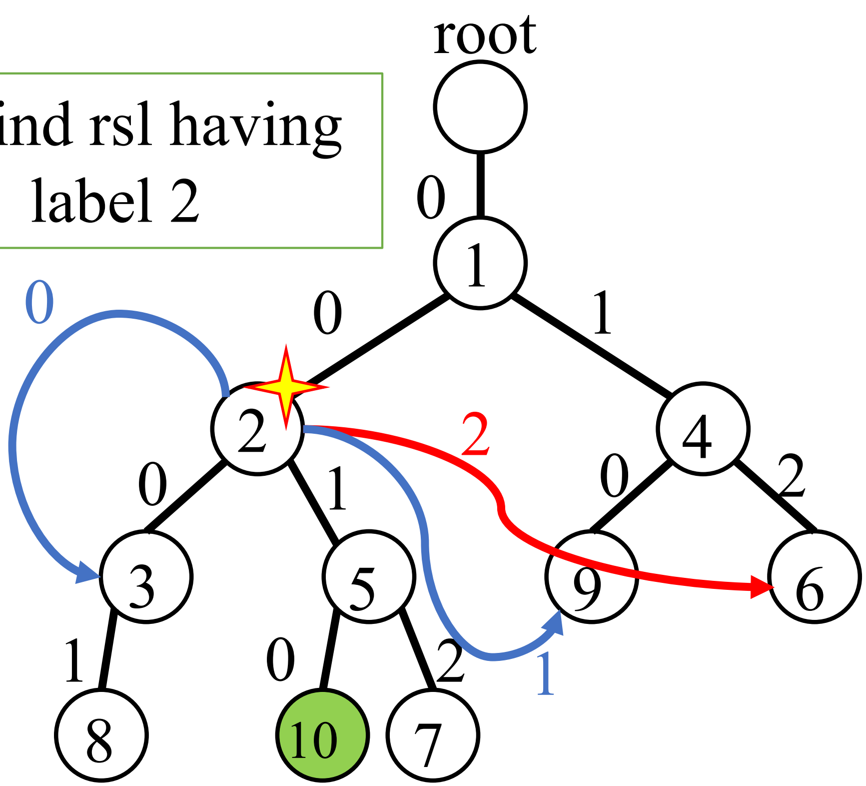
T	27584365741
w_{10}	0010012140
w_{11}	01214512140

The number of changed positions is 2.

Constructing CPH for string ($k \geq 2$)

We traverse CPH(T_{k-1}) from node k–1 towards the root and find the deepest node which has rsl with appropriate label a.

To find rsl having label 2

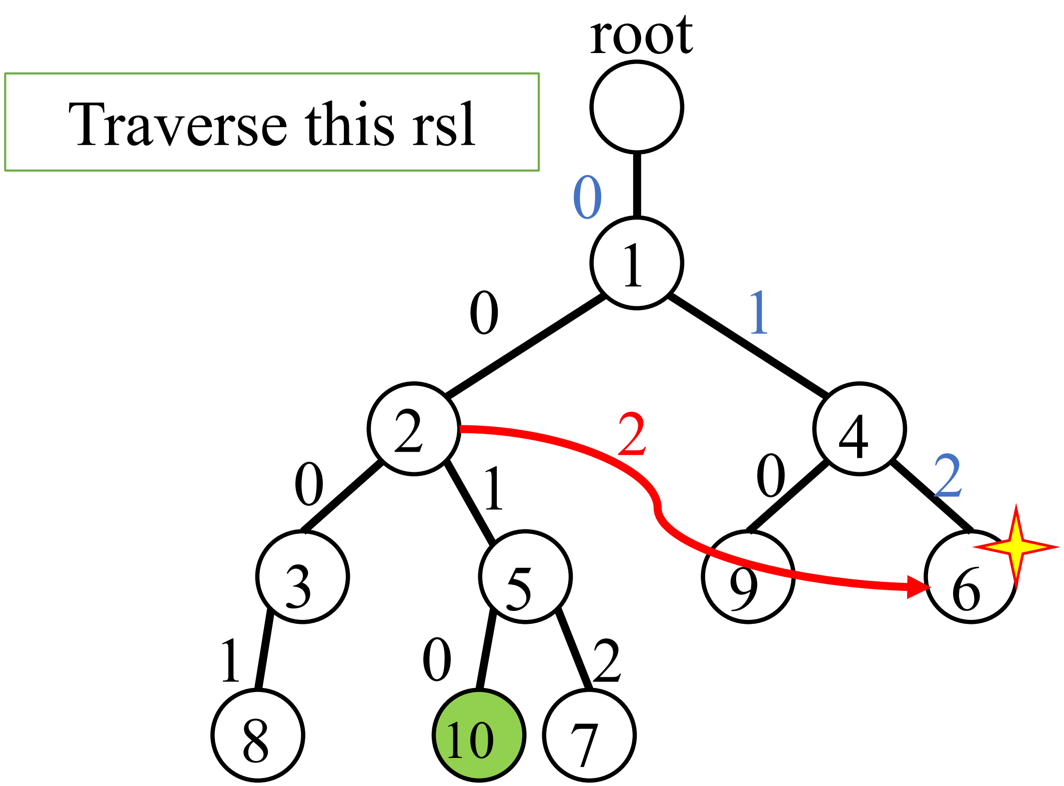


T	27584365741
w_{10}	0010012140
w_{11}	01214512140

The number of changed positions is 2.

Constructing CPH for string ($k \geq 2$)

We traverse CPH(T_{k-1}) from node k–1 towards the root and find the deepest node which has rsl with appropriate label a.

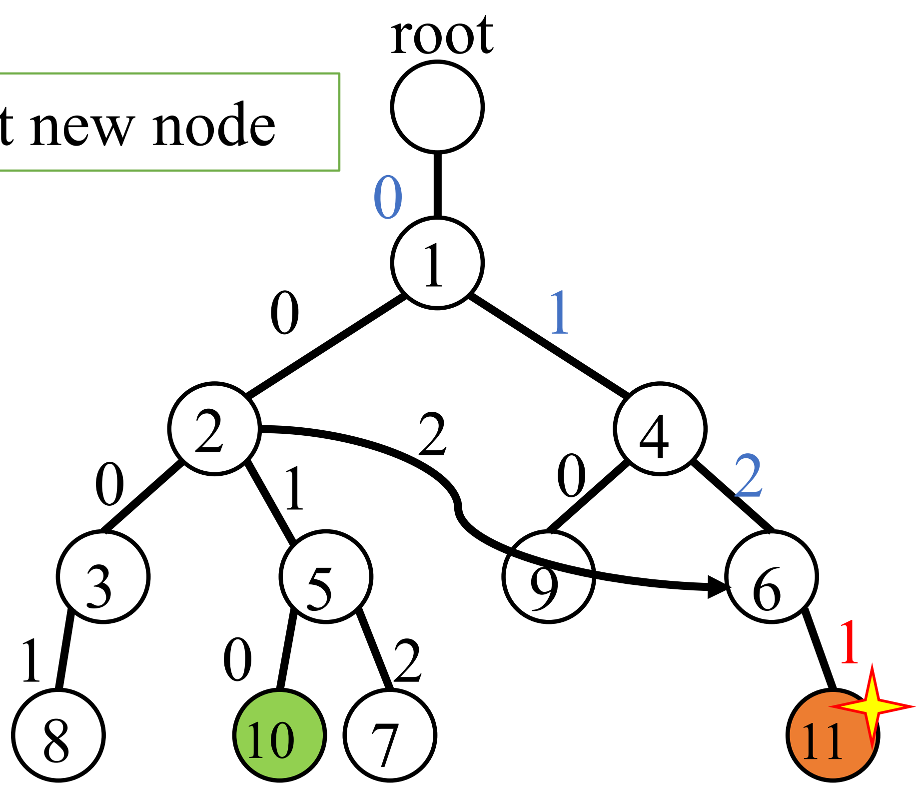


T	27584365741
w_{10}	0010012140
w_{11}	01214512140

Constructing CPH for string ($k \geq 2$)

We traverse the rsl and insert next character and a new rsl, after traversal.

Insert new node

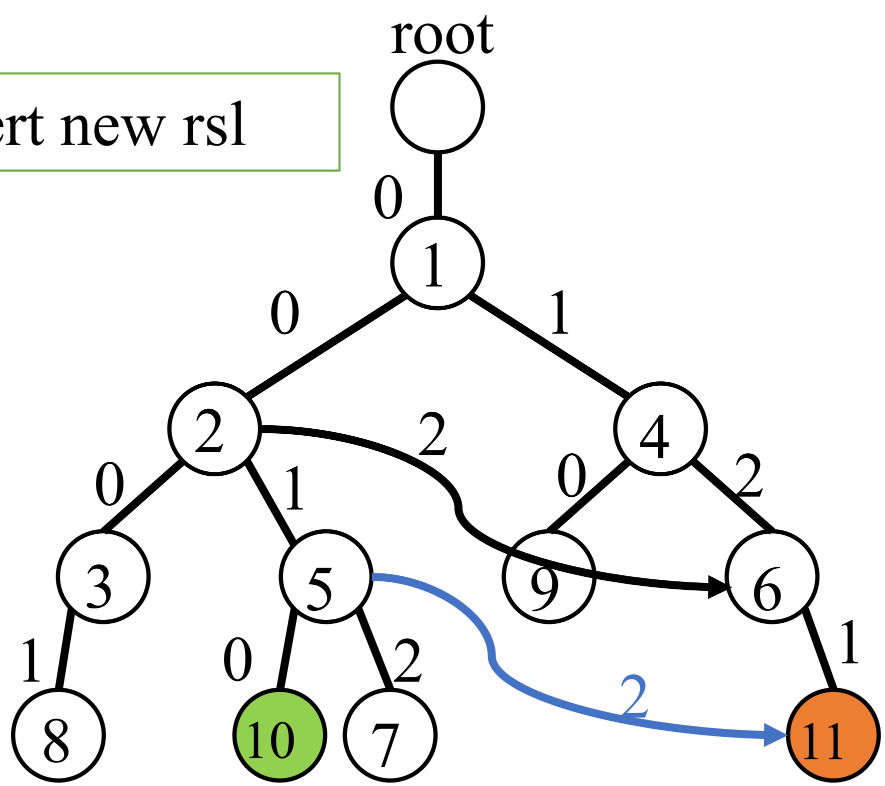


T	27584365741
w_{10}	0010012140
w_{11}	01214512140

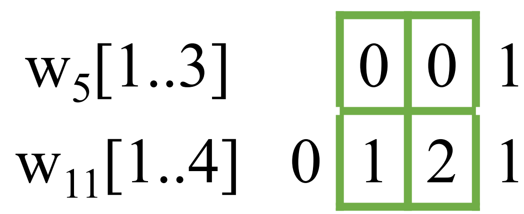
Constructing CPH for string ($k \geq 2$)

We traverse the rsl and insert next character with new rsl.

Insert new rsl



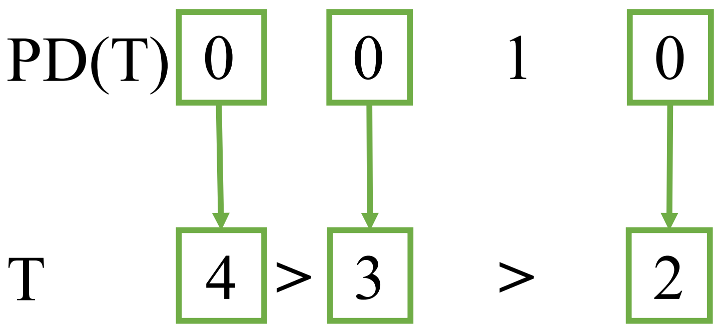
T	27584365741
w_5	00100
w_{11}	01214512140

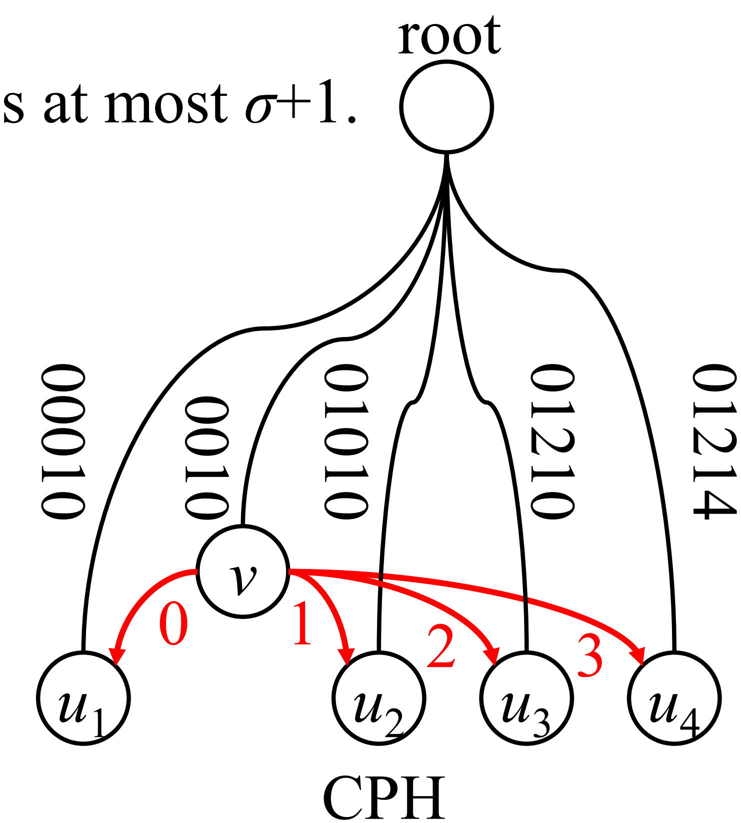


Number of rsl's from each node

Lemma
The number of rsl's from each node in CPH is at most σ+1.

By the definition of PD, the number of 0's in PD is at most σ.
So, the range of label of rsl is [0, …, σ].
The number of rsl which any nodes have is at most σ+1.

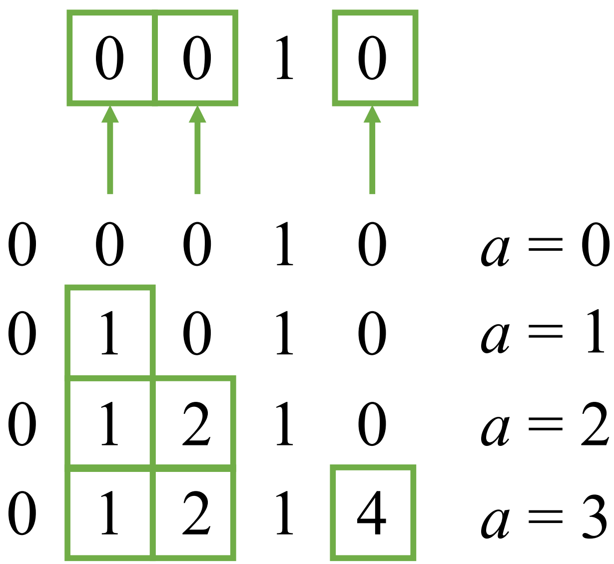


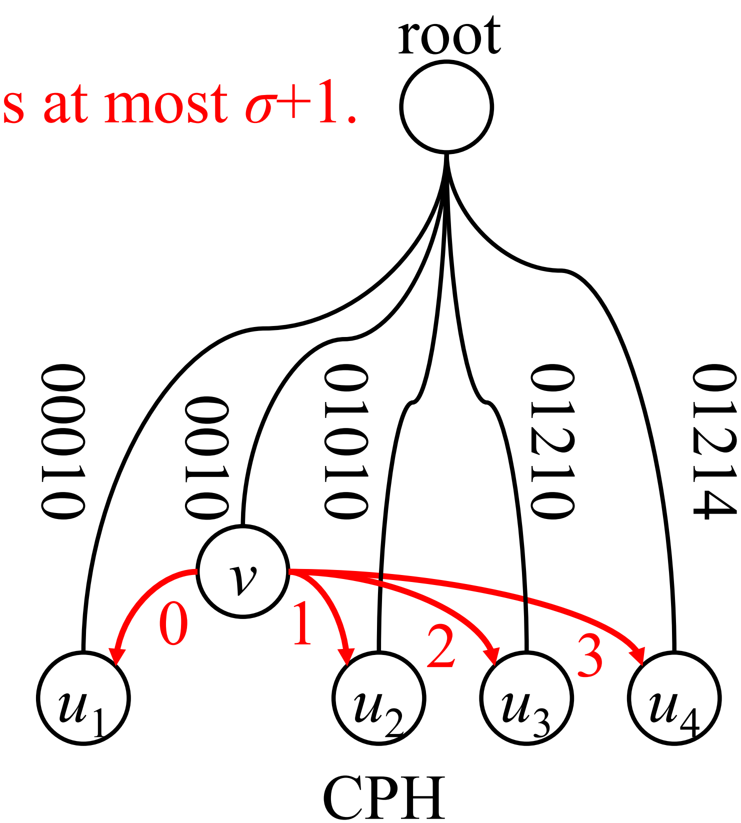


Number of rsl's from each node

Lemma
The number of rsl's from each node in CPH is at most σ+1.

By the definition of PD, the number of 0's in PD is at most σ.
So, the range of label of rsl is [0, …, σ].
The number of rsl which any nodes have is at most σ+1.



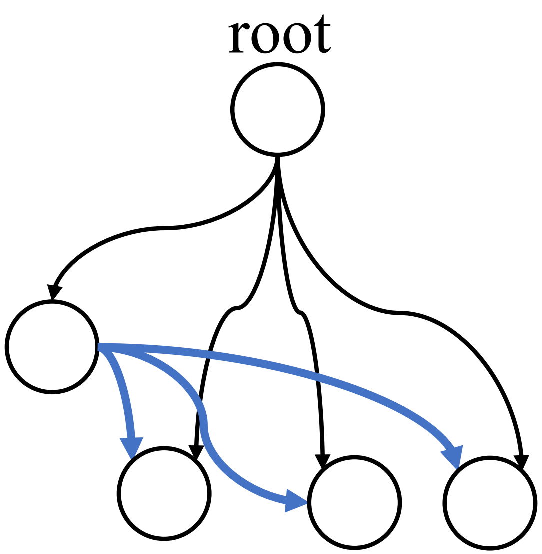


Construction time for CPH

- By using a standard amortization analysis, we can show that the total number of traversed nodes for all steps is $O(n)$.
- Since there are at most σ+1 reversed suffix links at each node, searching for the objective rsl at each node takes $O(\log \sigma)$ time.

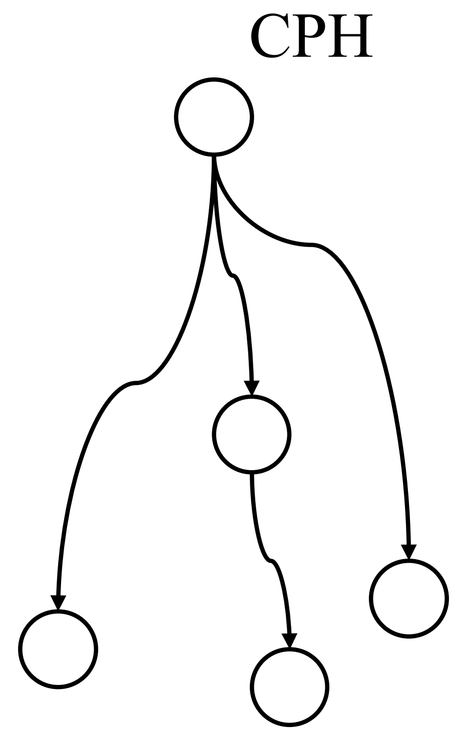
➔ We can construct CPH in $O(n \log \sigma)$ total time.

We use binary search to find rsl.



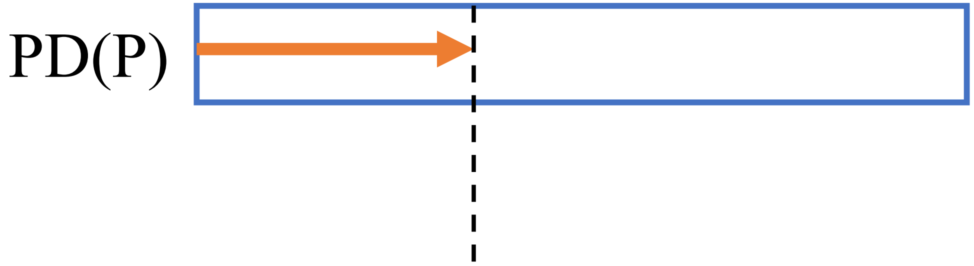
Pattern Matching with CPH

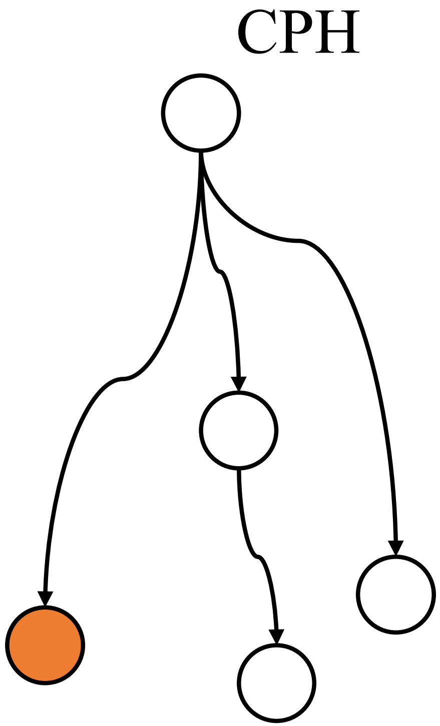
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.



Pattern Matching with CPH

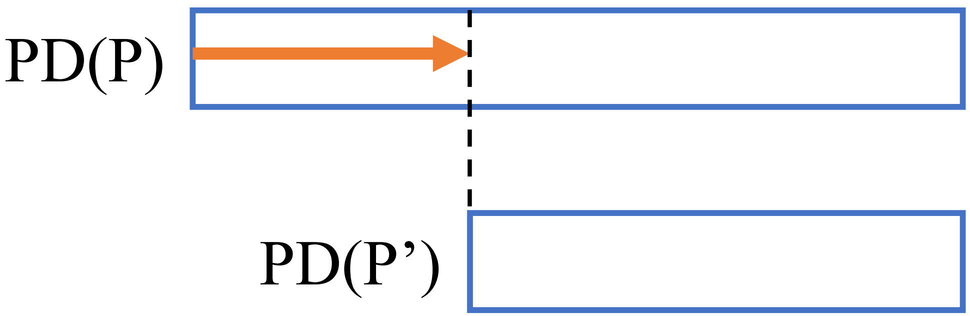
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.

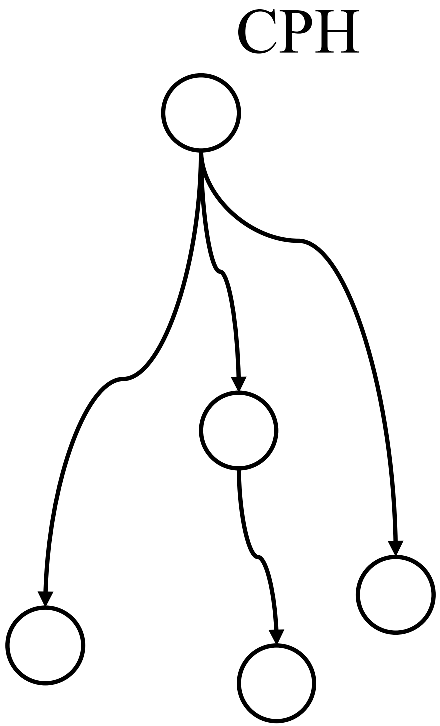




Pattern Matching with CPH

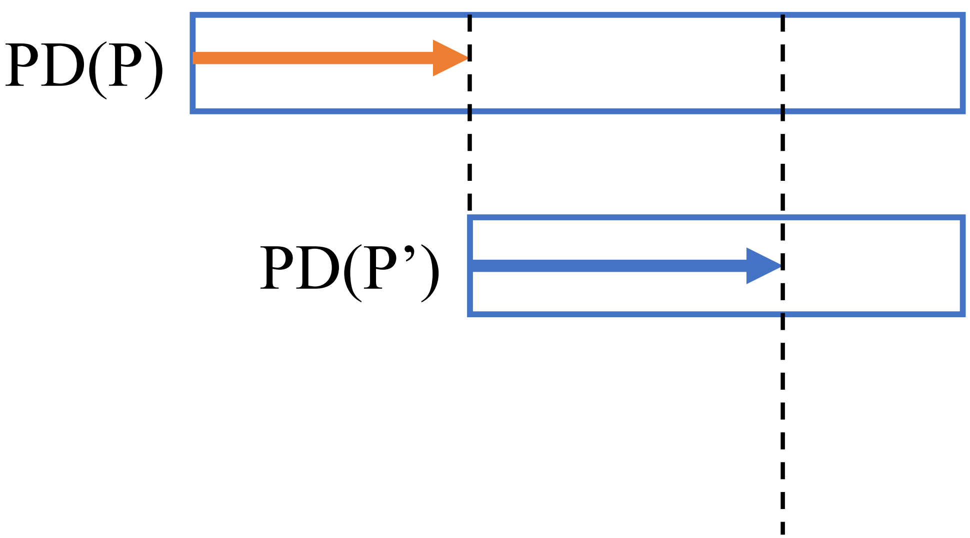
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.

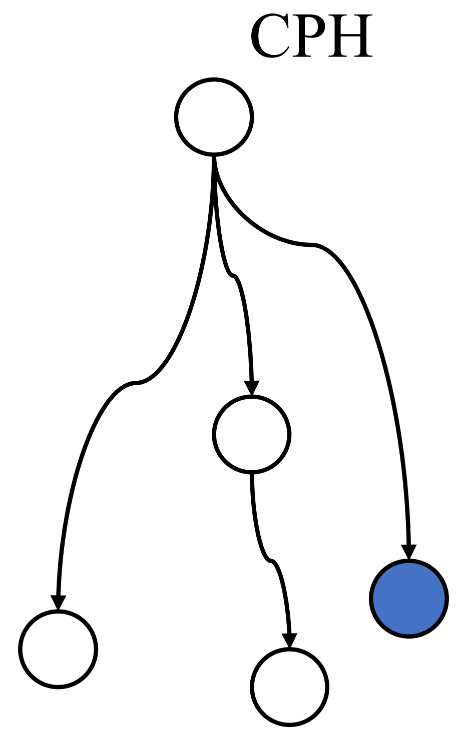




Pattern Matching with CPH

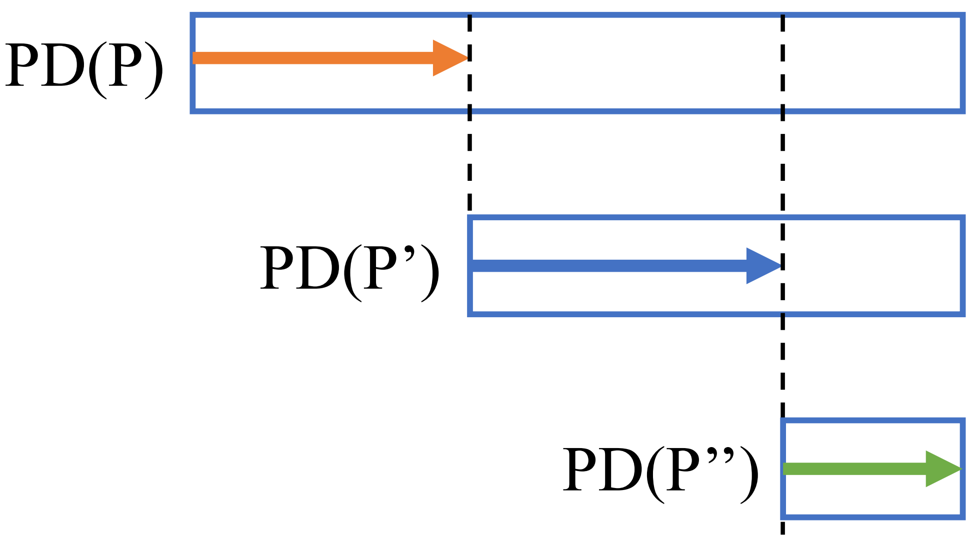
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.

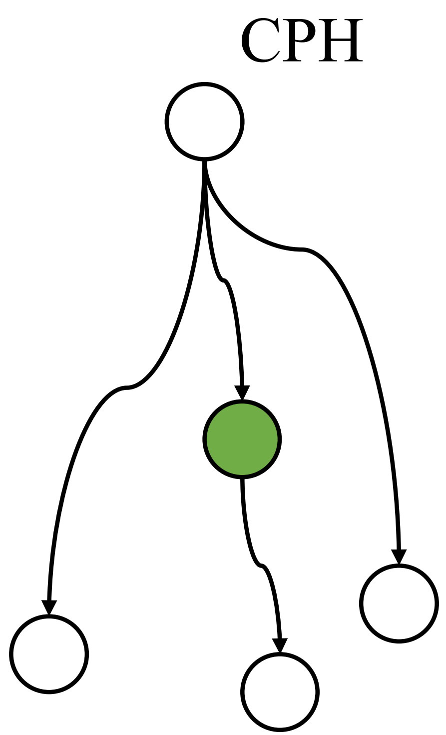




Pattern Matching with CPH

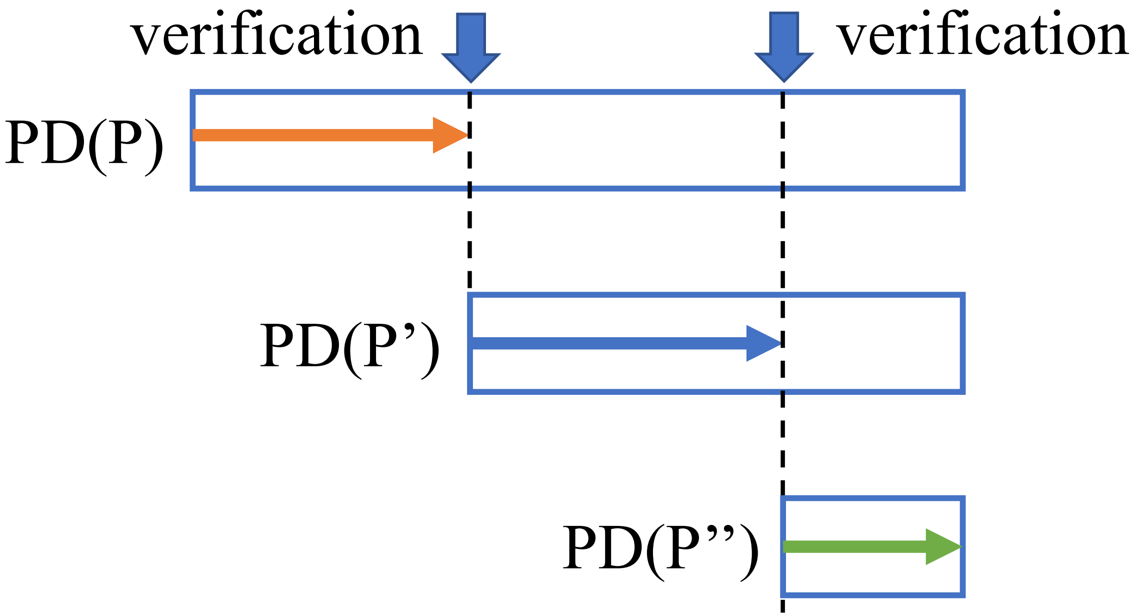
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.

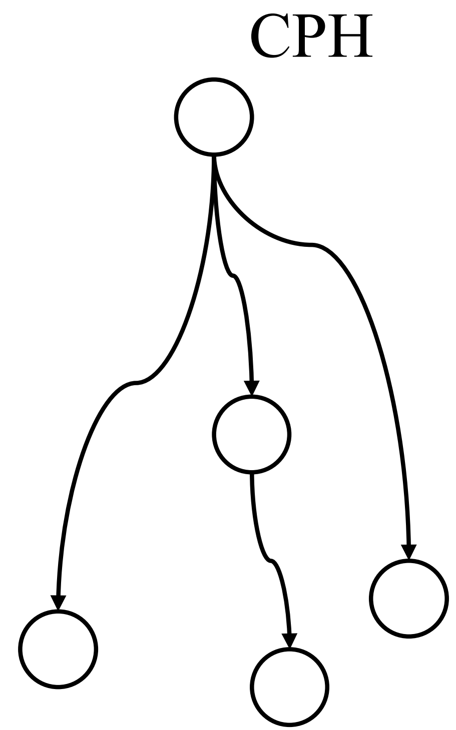




Pattern Matching with CPH

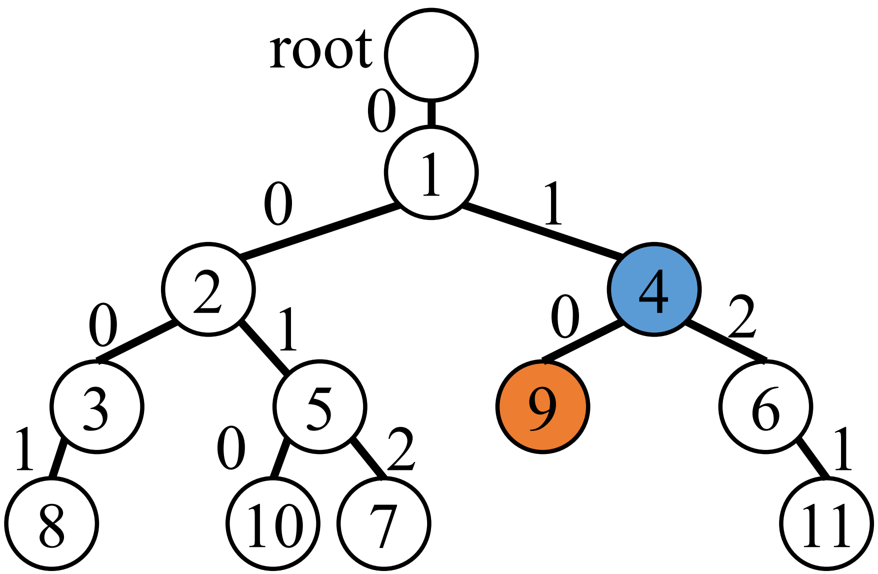
1. Traverse CPH(T) with PD(P).
2. Split P after the traversal.
3. Continue traversing CPH(T) with remainder of PD(P), and go to 2. If the remainder is nil, go to 4.
4. Verify the pattern.





Examples for verification

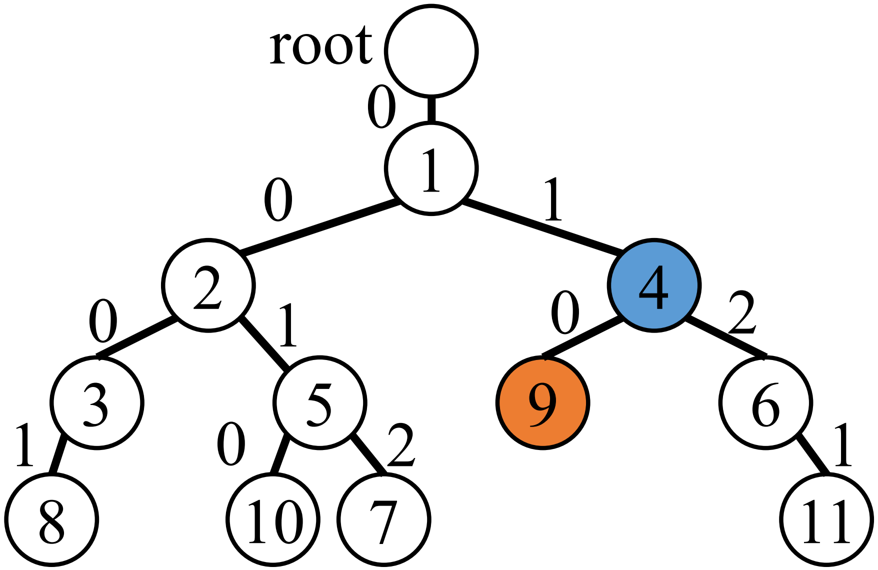
P	23145	
PD(P)	01011	
	010	01



T	27584365741
PD(T_9)	010012140
w_6	012140
w_9	010012140

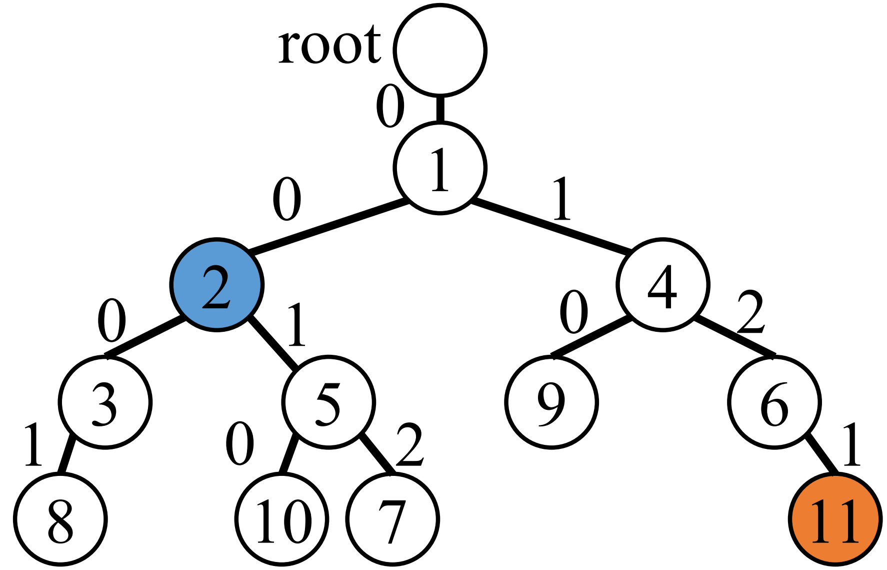
Examples for verification

P	23145	
PD(P)	01011	
	010	01



T	27584365741
PD(T_9)	010012140
w_6	012140
w_9	010012140

P'	154532	
PD(P')	012145	
	0121	00



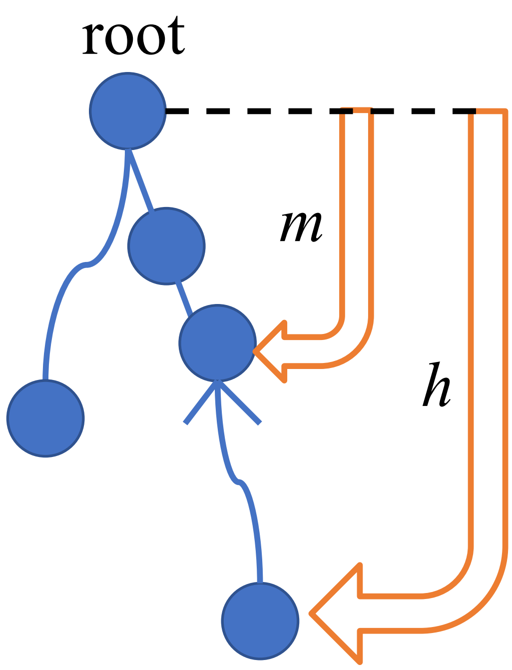
T	27584365741
PD(T_{11})	01214512140
w_7	0012140
w_{11}	01214512140

Pattern Matching Time

Pattern matching with CPH takes $O(m(\sigma+\log(\min(h,m)))+occ)$ time, where h is the height of CPH.

When pattern P spilt $P=P_1P_{2\ldots}P_k$,let m_i is length of P_i.

- The traversal take $O(m_i \log(\min(h,m)))$ time.
- Verification take $O(m_i\sigma)$ time.



For length m_i, $\Sigma_{i=1}{}^{k} m_i = m$,
matching takes $O(m(\sigma+\log(\min(h,m)))+occ)$ time using maximal reach pointers (details omitted).

Lemma
The number of edges from each node in CPH is at most the node depth.

Number of edges from each node

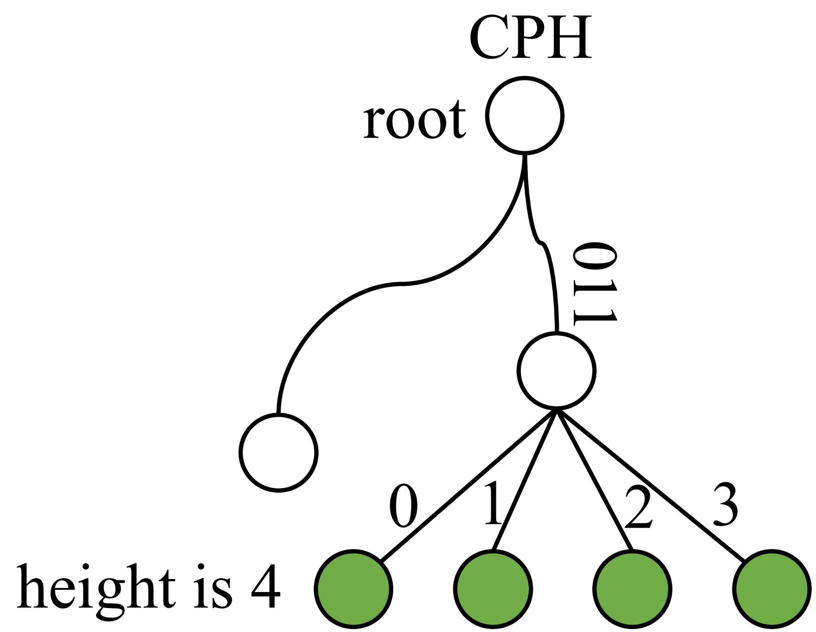
Lemma
The number of edges from each node in CPH is at most the node depth.

By the definition of PD, the maximum value in PD of length l is $l-1$. So, the number of edge which any nodes have in CPH is at most the node depth.

Example : $l = 4$

PD(T) 0 1 1 0

PD(T) 0 1 1 3



Our Contributions

Data structure	Const. time	Matching time	space
Cartesian Suffix Tree [Park et al., 2020]	$O(n \log n)$	$O(m \log n + occ)$	$O(n)$ words
Succinct Index [Kim and Cho, 2021]	$O(n \log n)$	$O(m \cdot occ)$	$3n + o(n)$ bits
Cartesian Position Heap [This work]	$\boldsymbol{O(n \log \sigma)}$	$\boldsymbol{O(m(\sigma + \log(\min(h, m))) + occ)}$	$\boldsymbol{O(n)}$ **words**

Cartesian Position Heap for trie [This work]	$\boldsymbol{O(N\sigma)}$	$\boldsymbol{O(m(\sigma^2 + \log(\min(h,m))) + occ)}$	$\boldsymbol{O(N\sigma)}$ **words**

n is text string length, σ is alphabet size, occ is number of pattern occurrences, m is pattern length, h is height of Cartesian Position Heap, N is text trie size.