

演習3 今井研 第二回 04/04/28

第三回 04/05/12

Compressed Suffix Trees

Compressed Suffix Arrays

岡野原 大輔

調べる題材



- CST (Compressed Suffix Trees)
- CSA (Compressed Suffix Arrays)
 - CSA [Grossi, Vitter 00] [Sadakane 03]
[Grossi, Gupta, Vitter 03]
 - FM-index [Ferragina, Manzini 00]
- 括弧木の表現及び操作
- rank及びselectに関する話
全体を並行に調べていく

Suffix Trees、Suffix Arraysについて

- Suffix
 - $T[1\dots n]$ の時、 T の各部分列 $T_i = T[i\dots n]$ を T のSuffixと呼ぶ。
- Suffix Trees
 - T の全てのSuffixについての検索木。
 - 根を除く各nodeには必ず子が二つ以上あり、それぞれの枝には空でない部分文字列がつけられている。同じNodeからの枝からは必ず違う部分文字列がつけられている。葉には、そのSuffixのポジションがつけられている。
- Suffix Arrays
 - Suffixを辞書式順に並べた時のポジションを並べたもの (Suffix Treesの葉だけを順に並べたもの)
 - # 各Suffixが必ず違うように、文字列の最後に文字列中に現れない文字を挿入しておく。この文字を\$とする。

Suffix Trees, Suffix Arraysの例

元データ abracadabra

- Suffix

abracadabra\$

bracadabra\$

racadabra\$

acadabra\$

cadabra\$

adabra\$

dabra\$

abra\$

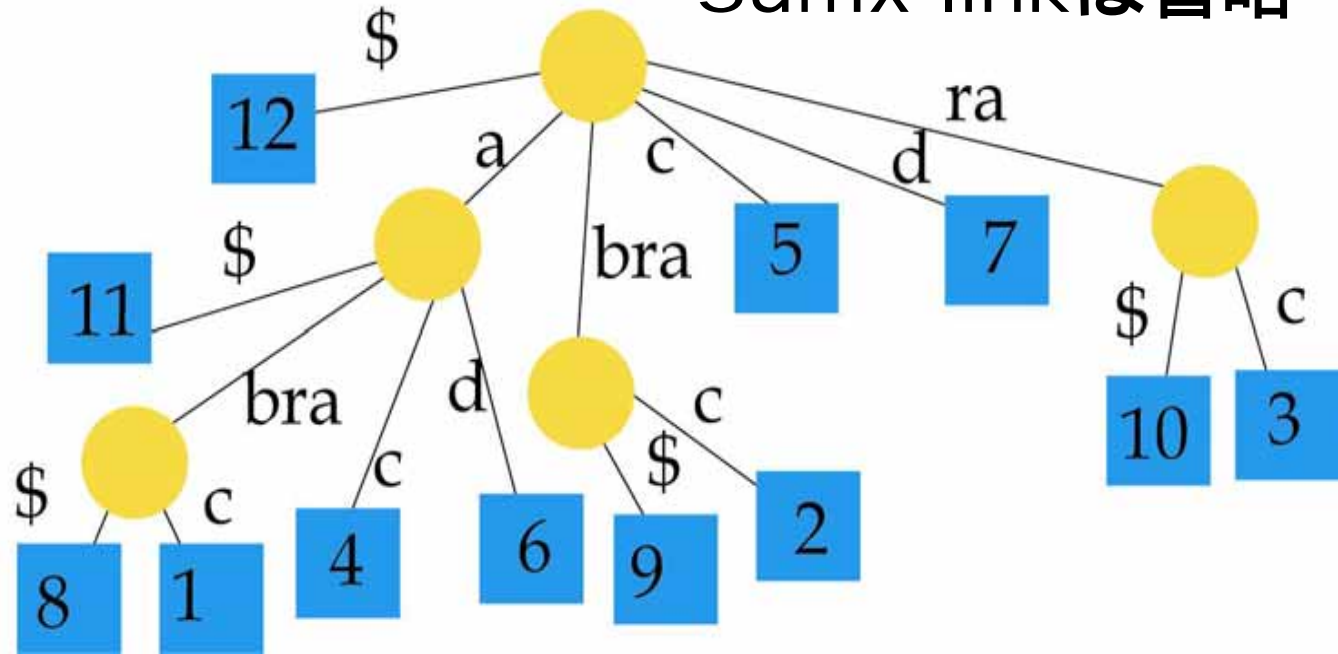
bra\$

ra\$

a\$

\$

Suffix linkは省略



Suffix Arrays : 13 11 8 1 4 6 12 2 5 7 10 3

Suffix Trees、Suffix Arraysの利用

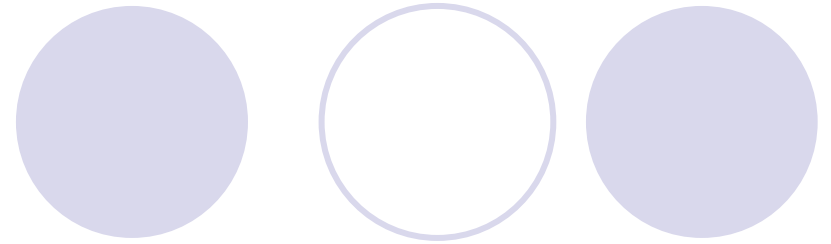
- Suffix Arrays

- 部分文字列の検索、列挙
- テキストの部分的な復元

- Suffix Trees

- Suffix Arraysで解ける問題
 - 2つの文字列の最長共通部分文字列
 - 繰り返し部分文字列の抽出
- 他にも非常に多数。

定義 rank select



- $\text{rank}_p(B, i)$
 - bit array B中のB[1..i]中のpの数
- $\text{select}_p(B, i)$
 - bit array B中のi番目のpの位置
- rank selectは $< O(n)$ の補助領域で定数時間で求められる。(証明は後述)

i	1	2	3	4	5	6	7
B	0	1	0	0	1	1	0
rank_1	0	1	1	1	2	3	3
select_1	2	5	6				

定義 Suffix Link

- Rootを除く、節点 v のラベルが x の時、(但し x は単一文字、 $sl(v)$ は空であるかもしれない部分文字列) $sl(v)$ はラベルが x である節点を返す。
 - 例 節点 v のラベルが、abcdeの時、 $sl(v)$ はbcdeのラベルを持つ節点を返す
- Suffix Treeを線形時間で構築するのに必要
- 二つの文字列中の共通部分文字列を線形時間で探すことなど、実用上でも重要

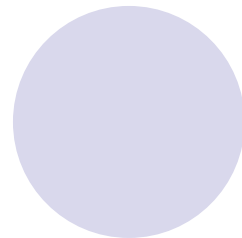
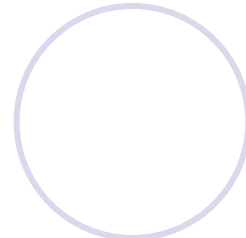
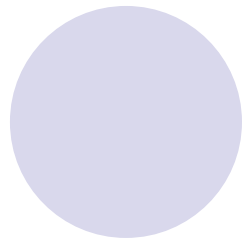
CSAの概要

- Suffix Arrays (以下SA)は[1..n]の permutationなので、そのままでは、圧縮は不可能
- $[i] = SA^{-1}[SA[i] + 1]$ をSAの代わりに保存し、これを用いてSA[i]を表現する
 - 但し $[n] = 0$
- は、各部分列において単調増加列という冗長性を持っている。これはSuffixの辞書式順序列という性質をうまく利用している。

i	SA		Suffix
1	11	3	a\$
2	8	6	abra\$
3	1	7	abracadabra\$
4	4	8	acadabra\$
5	6	9	adabra\$
6	9	10	bra\$
7	2	11	bracadabra\$
8	5	5	cadabra\$
9	7	2	dabra\$
10	10	1	ra\$
11	3	4	racadabra\$

元データ
abracadabra\$

i	SA		Suffix
1	11	3	a\$
2	8	6	abra\$
3	1	7	abracadabra\$
4	4	8	acadabra\$
5	6	9	adabra\$
6	9	10	bra\$
7	2	11	bracadabra\$
8	5	5	cadabra\$
9	7	2	dabra\$
10	10	1	ra\$
11	3	4	racadabra\$



は部分列で
単調増加。

一文字目が同
じ時、二文字
目で比較して
いるから。

CSA(1) [Grossi, Vitter 00]

- SAを一定間隔のみ保存し、 SA_k は全て保存。任意のSAを求める場合、保存されているSAまで、 SA_k を用いて移動
- レベルkでは、SAの $j = SA[i]$ のうち j が 2^k の倍数のものだけを暗黙的に $SA_k[1 \dots n/2^k] \wedge 2^k$ で割って保持。 $0, e, 2e, \dots, l$ レベルのみ保持
 $e = \lceil \varepsilon \log \log n \rceil, 1 \geq \varepsilon > 0, l = \lceil \log \log n \rceil$
- ビットベクトル $B_k[i]$ には、 $SA_k[i]$ が 2^k の倍数なら1、そうでないなら、0を保存
 - これより、 $B_k[i] = 1$ のとき (indexが1から始まる)
 $SA_k[i] = 2^e SA_{k+e}[\text{rank}(B_k, i)]$

CSA(2)

- $B_k[i]=0$ の時に、 を用いて、 $B_k[j]=1$ となるところまで移動する。

$$SA_k[i] = SA_k[\psi_i[i]] - 1$$

$$SA_k[i] = SA_k[\psi_i^v[i]] - v$$

- 実際に保存するのは、
 $SA_k, B_k, \psi_k (k = 0, e, 2e \dots l - e)$
この領域量は $O(n)$ である。(後述)
- $SA[i]$ を求める計算量は $O(\log n)$

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

	1	2
SA ₂	2	1

SA[5]を求める

$$\begin{aligned}
 SA[5] &= SA_0[\psi[5]] - 1 \\
 &= SA_0[9] - 1 \\
 &= SA_0[\psi[9]] - 2 \\
 &= SA_0[2] - 2 \\
 &= 2^2 SA_2[\text{rank}(B, 2)] - 2 \\
 &= 2^2 SA_2[1] - 2 \\
 &= 6
 \end{aligned}$$

青・実際に
保存する
= 1の時

CSA(3)

[Sadakane 03]

- が区分単調増加列になっているのをういて領域量を $O(nH_0 + n \log \log |A|)$ に抑える

$$d_k[i] = \begin{cases} \psi_k[i] - \psi_k[i-1] & T_k[SA_k[i]] = T_k[SA_k[i-1]] \\ \psi_k[i] & \textit{otherwise} \end{cases}$$

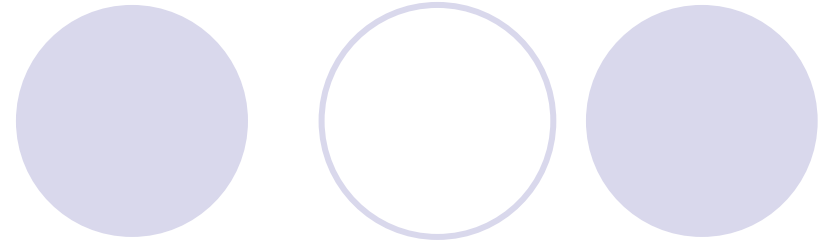
- $d_k[i]$ を $-code$ で符号化。
 $-code$ は自然数 r を次の bit 数で表現

$$1 + \lfloor \log r \rfloor + 2 \lfloor \log(1 + \lfloor \log r \rfloor) \rfloor$$

CSA(4) [sadakane 03]

- SA^{-1} をレベル(最深部)のみ保持しておく、
、 SA から任意の SA^{-1} を求めることができる。
- $i = SA_{k+e}^{-1}[q]$ がわかっているとす。この時、
 $B_k[j]=1$ かつ $i = \text{rank}(B_k, j)$ が成り立つ j が存在
し、 $j = \text{select}(B_k, i)$ で求められる。 $SA_k^{-1}[2^e q + v]$
は、 $v_k[j]$ により求められる。
- $SA_k^{-1}[t]$ を求めるには、 $t = 2^e q + v$ ($v = t \bmod 2^e$)
として、 q と v を求める。 $SA_{k+e}^{-1}[q] = i$ の時、
 $SA_k^{-1}[2^e q] = \text{select}(B_k, i)$
 $2^e q = SA_k[\text{select}(B_k, i)]$

CSA(5) SA⁻¹ 追加



● SA_k⁻¹[t]を求めるルゴリズム

○ $t = 2^e q + v$ ($v = t \bmod 2^e$) として、 q と v を求める。

○ $SA_{k+e}^{-1}[q] = i$ を求める。

$$SA_k^{-1}[2^e q] = \text{select}(B_k, i)$$

$$2^e q = SA_k[\text{select}(B_k, i)]$$

$$2^e q + v = SA_k[\psi^v(\text{select}(B_k, i))]$$

○ より

$$SA_k^{-1}[2^e q + v] = \psi^v(\text{select}(B_k, SA_{k+e}^{-1}[q]))$$

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

	1	2
SA ₂	2	1
SA ₂ ⁻¹	2	1

SA⁻¹[6]

$$i = \lfloor 6 / 4 \rfloor = 1$$

$$j = 6 \bmod 4 = 2$$

$$SA_2^{-1}[i] = SA_2^{-1}[1]$$

$$= 2$$

$$\text{select } (B_0, 2) = 4$$

$$\psi_0^j[4] = \psi_0^2[4]$$

$$= \psi_0[8]$$

$$= 5$$

CSA(5) 部分文字列復元

[sadakane 03]

- 任意の位置からの部分文字列を全体のテキストを復元することなく、復元可能である。
- Suffixの先頭文字を保持しておくことで可
 - 実際に保持するのではなく、これはrank及び、文字列表で実現 $O(n)$ 以下で保持できる
- により、復元を行う。
 - BWTの逆変換に似ている。

i	1	2	3	4	5	6	7	8	9	10	11
SA ₀	11	8	1	4	6	9	2	5	7	10	3
₀	3	6	7	8	9	10	11	5	2	1	4
B ₀	0	1	0	1	0	0	0	0	0	0	0
Text	a	a	a	a	a	b	b	c	d	r	r

T[sp .. sp + m] を復元

```

p = SA-1[SP]
for i = 0 to m
    putchar(Text[p])
    p = [p]

```

元データのT[5..8]を
復元する例

$$SA^{-1}[5] = 8$$

$$Text[8] = c$$

$$[8] = 5$$

$$Text[5] = a$$

$$[5] = 9$$

$$Text[9] = d$$

$$[9] = 2$$

$$Text[2] = a$$

Compressed Suffix Treesについて

- 「Compressed Suffix Trees with Full Functionality」 Kunihiro Sadakane 2004
を参照。
- CSAを基にし、Suffix Treesの各操作をSimulateする。
- データ長 n 、アルファベットが A の時、
必要領域量は $O(n \log n)$ から、 $O(n \log |A|)$
各操作計算量は $O(1)$ から、 $\text{polylog}(n)$

CSTの重要度

- Suffix Treesは多くの文字列検索処理を元データサイズによらず、パターン長にのみ依存する計算量で処理することが可能である。
(Suffix Arraysのみでは高度な処理ができない)
- 従来のSuffix Treesは必要領域量が $O(n \log n)$ なので、膨大なデータを処理するWeb、ゲノムマイニングにおいてSuffix Treesの構築、利用が難しかった。

→ 今回のCSTで初めて、on memory
で処理可能となる

CSTが備える操作

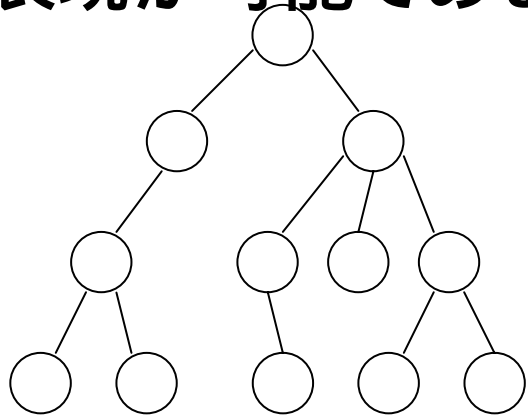
- SimulateするSuffix Tree(ST)の機能
 - root() STの根を返す
 - isleaf(v) vが葉かどうか
 - child(v,c) vの子wの中で、edgeがcであるものを返す
 - sibling(v) vの次のきょうだいを返す
 - parent(v) vの親を返す
 - edge(v,d) vを指している枝のd番目の文字を返す
 - depth(v) vの文字の深さ(長さ)を返す
 - lca(v,w) vとwのもっとも近い共通の親を返す
 - sl(v) vのSuffix Linkを返す

CSTの概要 [Sadakane 04]

- Suffix Treeの構成要素をCSA、括弧列P、Hgt(後述)、を用いてシミュレートする。
- CSAのサイズが $|CSA|$ とする。 $(|CSA|$ は元のテキストサイズより小さい。 nH_k [Grossi, Gupta, Vitter 03])
- 括弧列Pは $4n$ bit
- Hgtは $2n$ bit
- 他の補助領域は全て $o(n)$
→ 全体で $|CSA| + 6n + o(n)$ bit

括弧による木の表現

- 木をpreorderでtraverseしたとき、Nodeに入る時に“(“を、出る時に、)”“を書く、木構造が括弧により表現される。Node数がnの時、 $2n$ 個の括弧による表現が可能である。



((((()()))((()())(())())))

- 以降、括弧木をbit array Pで表現し、 $P[i]$ により、Pのi番目の要素をさすこととする。

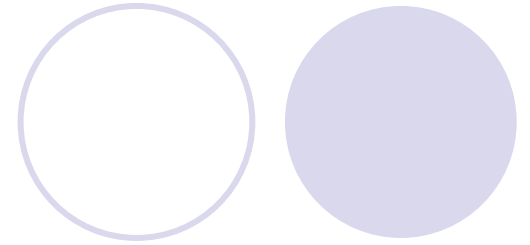
括弧木の各種操作(1)

- 以下の操作は全て定数時間で実行可能
[Munro Raman Rao 01]
 - $\text{rank}_p(i)$
 - i までのPattern P の出現回数
 - $\text{select}_p(i)$
 - Pattern P の i 番目の出現位置
 - $\text{findclose}(i)$
 - i の位置にある“(“に対応する“)”の位置を返す
 - $\text{enclose}(i)$
 - i の位置にある“(“を包む最小の括弧対の”(“の位置
- 例 ((()))() 赤の括弧のencloseは青の括弧

括弧木の各種操作(2)

- $\text{leftrank}(v)$ Node v に到達するまでpreorderで到達した葉の数
- $\text{leftmost}(v)$ v を根とする部分木の最も左側にある葉の位置
- $\text{rightmost}(v)$ v を根とする部分木の最も右側にある葉の位置

木における操作を 括弧列によりsimulateする例



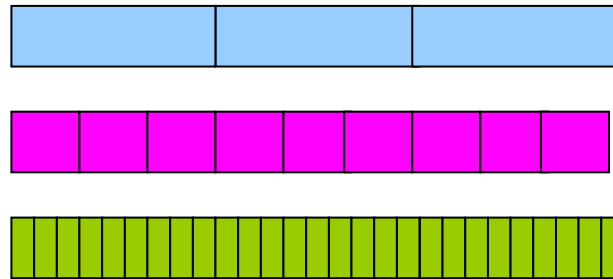
- $\text{root}() = 1$
- $\text{parent}(v) = \text{enclose}(v)$
- $\text{sibling}(w) = \text{findclose}(w) + 1$
- $\text{child}(v, c) := v$ の子で頭文字が c
 - v の最初の子供 w は $w = v + 1$ で求まる。 w と w の siblingを、labelを復元しながら c と合っているかチェックをして求める

括弧列の定数時間操作の概要

[Jacobson 89] [Munro, Raman 01]

- Pを三段階の大きさのブロックに再帰的に分ける。

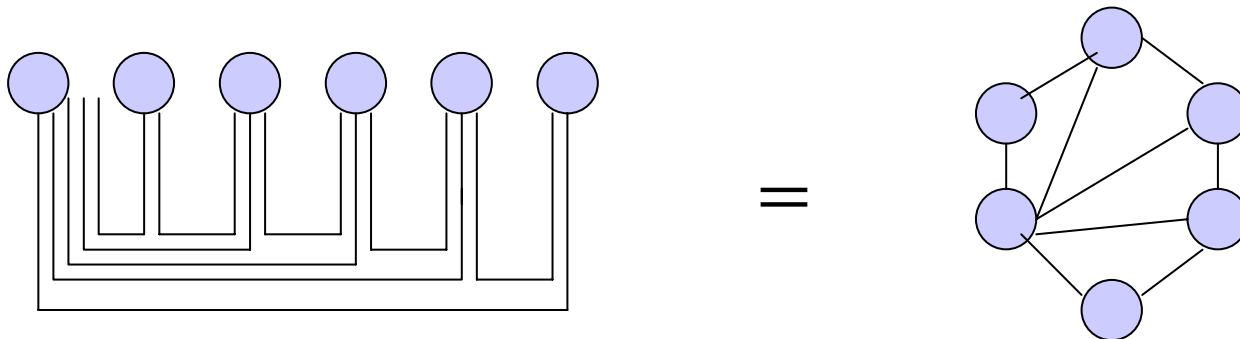
○ big small tiny



- far 対応する括弧が同じブロック内に無い
- pioneer farの中で、直前のfarの括弧に対応する括弧の入っているブロックとは違う場所(より前のブロック)に対応する括弧がある。

pioneerの数が $2B-3$ であることの証明

- 各ブロックを B 個の点に対応させ、pioneerが存在するブロックと、それに対応する括弧が存在するブロックを線で結ぶ。この時、木構造であることから、各線は交差せず、さらに全ての点は同一平面を共有している。よって、最大の線数は、 $B + B - 3 = 2B - 3$



対応する括弧を求める

- 全てのpioneerに対して、対応する括弧の位置を完全Hashで保持
- farである括弧pについては、直前のpioneerを探す。
(直前のpioneerの対応する括弧が存在するブロックとpの対応する括弧が存在するブロックは同じブロック)
 - 各ブロックには、直前のpioneerの位置を保持。
- 各ブロックについて、左の始めからブロックの開始地点までの“(“の数 - “)”の数をleft excessとする。また、右の終わりからブロックの終わりまでの”)” - “(“の数をright excessと呼ぶ。これらを各ブロックで保持

farの括弧に対応する括弧を求める

- 直前のpioneerまで到達し、そこでの、左からのexcessを求める。そして、対応するブロック内での右からのexcessが一致する一番左側の閉括弧を調べる。



Hgtを用いたSuffix TreeのTraverse

[Kasai et al 01]

- $Hgt[i] = lcp(T_{SA[i]}, T_{SA[i+1]})$ ただし $i = n$ の時 0
 - $lcp(a, b)$ は a と b の 最長共通接頭辞 (Longest Common Prefix) の長さ。
例 $lcp(abcde, abcee) = 3$
- 1 から n まで順に Hgt を Scan しながら、 Hgt が大きくなっている列では Stack に Push し、 Hgt が小さくなっているところでは Pop して Node を報告
- Node は、Suffix Arrays 上の範囲として報告する。

定数時間のrank操作

- 長さ n のbit列を $\log^2 n$ 個ずつのblockに区切り、それぞれのblockの先頭のrankを保持

$$\frac{n}{\log^2 n} \lg n = \frac{n}{\log n} \text{ (bit)}$$

- 大きさ $\log n$ のblockの先頭のrankを保持

$$\frac{n}{\log n} \lg(\log^2 n) = \frac{2n \lg \log n}{\log n} \text{ (bit)}$$

- $(\log n) / 2$ のパターンのrankを全て表で保持

$$2^{\frac{\log n}{2}} \frac{\log n}{2} \lg\left(\frac{\log n}{2}\right) = \sqrt{n} \frac{\log n}{2} \lg\left(\frac{\log n}{2}\right) \text{ (bit)}$$

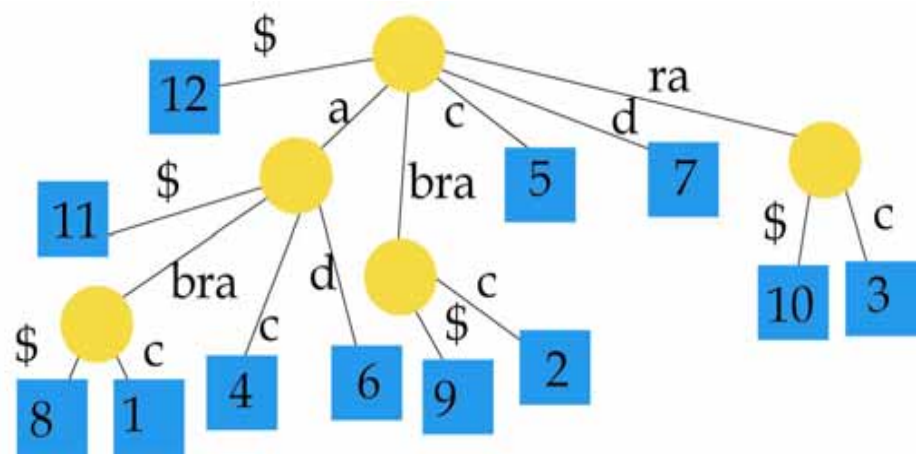
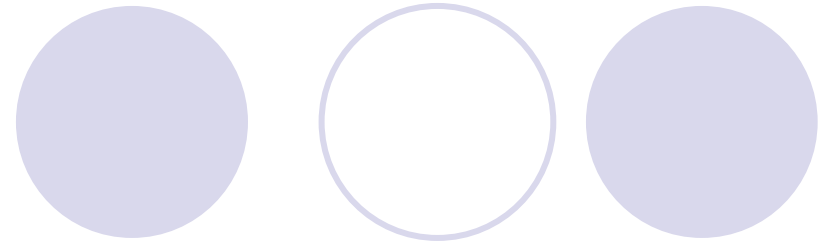
- 記憶領域は n 未満。定数時間でのselect

BottomUpTraverse



1. Stack S を空に初期化
2. for $k = 1$ to $n+1$ do
3. $v := \text{Hgt}[k - 1]$
4. while ($\text{depth}(\text{top}(S)) > \text{depth}(v)$) do
5. $v := \text{Pop}(S)$; report v
6. if ($\text{depth}(\text{top}(S)) < \text{depth}(v)$) then
7. $\text{Push}(v, S)$

H	Suffix
0	\$
1	a\$
4	abra\$
1	abracadabra\$
1	acadabra\$
0	adabra\$
3	bra\$
0	bracadabra\$
0	cadabra\$
0	dabra\$
2	ra\$
0	racadabra\$



lcpの効率的な保存方法

[Sadakane 03]

- lcp、つまりHgtは大抵小さい値だが、n-1までなりうることもある(例 入力が全て同じ文字) そのため、Hgtはそのまま保存すると $O(n \log n)$ bit 必要である。
- HgtもSA同様、変換を行い、圧縮されやすい形に変換する。そのためには次の補題を用いる。

$$Hgt[\psi[i]] \geq Hgt[i] - 1$$

補題の証明

[Kasai 01]

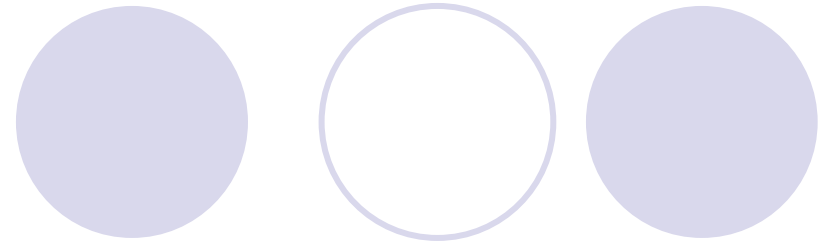
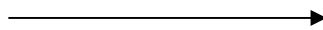
$$Hgt[\psi[i]] \geq Hgt[i] - 1$$

- $Hgt[i] = lcp(T_{SA[i]}, T_{SA[i+1]})$
 - $T[SA[i]] \neq T[SA[i+1]]$ の時、 $Hgt[i] = 0$ より成立
 - $T[SA[i]] = T[SA[i+1]]$ の時、 $[i] < [i+1]$ より $[i] + 1 = j \leq [i+1]$ となる j が存在。
 j は、 $[i] + 1$ と、 $[i+1]$ の間にあり、 $T_{SA[j]}$ 、 $T_{SA[i+1]}$ はそれぞれ、 $Hgt[i]-1$ の共通接頭辞を持っているので、 $T_{SA[j]}$ も $T_{SA[i]}$ と少なくとも、 $Hgt[i]-1$ の共通接頭辞を持っている。
よって $Hgt[j] \geq Hgt[i] - 1$

補題の例

i abcdef

i+1 abcdegh



[i] bcdef

j bcde.....

[i+1] bcdegh

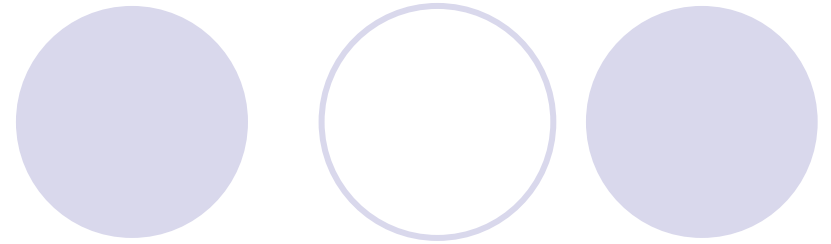
$$\text{Hgt}[i] = 5$$

$$\text{Hgt}[[i]] \geq 4$$

Hgtの変換

- $p = SA^{-1}[1]$ としたとき
 $Hgt[p] \leq Hgt[SA[p]] + 1$
 $\leq Hgt[SA^2[p]] + 2$
.... $\leq Hgt[SA^{n-1}[p]] + n - 1 = n - 1$
という $[0.. n-1]$ の単調増加列が構築できる。
 - 最後の等式は $SA[SA^{n-1}[p]] = n$ であり、 $T[n]$ はunique文字であるから、 $Hgt[SA^{n-1}[p]] = 0$ から成り立つ
- Hgtをそのまま保持する代わりに、この単調増加列を保存すればよい。
- では、どのようにして、この並び替えた列から、 $Hgt[i]$ を得ることができるか。

Hgtの変換(2)



- Hgt[i]を求めたいとき、 $i = k[p]$ となる、kは以下のようにして求められる
$$SA[i] = SA[k[p]] = SA[p] + k = k + 1$$
より、 $k = SA[i] - 1$
- $[0..n-1]$ の範囲にある、単調増加列のn個の整数は $2n + o(n)$ bitで定数時間でaccessできる。
 - 隣の数との差をtとした時、t個の0を書いた後に1をおく。このとき、i番目の要素の値はselect[i]によって求められ、 $2n$ bitあれば十分である。また、定数時間によるselectには $o(n) < O(n)$ bitの補助領域があれば可能である

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

$$p = SA^{-1}[1] = 3$$

- 1 = Hgt[3]
- 1 = Hgt[[3]] + 1 = Hgt[7] + 1
- 2 = Hgt[[7]] + 2 = Hgt[11] + 2
- 4 = Hgt[[11]] + 3 = Hgt[4] + 3
- 4 = Hgt[[4]] + 4 = Hgt[8] + 4
- 5 = Hgt[[8]] + 5 = Hgt[5] + 5
- 6 = Hgt[[5]] + 6 = Hgt[9] + 6
- 11 = Hgt[[9]] + 7 = Hgt[2] + 7
- 11 = Hgt[[2]] + 8 = Hgt[6] + 8
- 11 = Hgt[[6]] + 9 = Hgt[10] + 9
- 11 = Hgt[[10]] + 10 = Hgt[1] + 10

1 1 2 4 4 5 6 11 11 11 11

上のbitによる表現

110100110101000001111

lcpの計算 [sadakane 03]

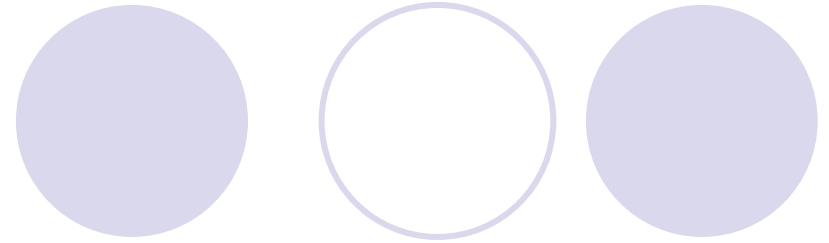
- $T_{SA[l]}$ $T_{SA[r]}$ の最長共通接頭辞 (longest common prefix) を求めるのは $Hgt[l]$ $Hgt[l+1] \dots Hgt[r-1]$ の最小値とそのindexを求める問題に帰着。
- 最小値とそのindexを定数時間で求めるために $o(n)$ の補助領域を前もって構築
[Bender et al 00] のアルゴリズムを改良

CSTの各節点、葉へのアクセス

- P上のindex v と番号付け i の相互変換
- preorder (行きがけ順) によるindexを用いたアクセス
 - preorderは深さ優先探索順であり、括弧列P中のindex v と、番号の変換は以下の通り。
$$i = \text{rank}_l(v) \quad v = \text{select}_l(i)$$
- inorder (通りがけ順) によるindexを用いたアクセス
 - inorderは内部節点のみ定義される。複数の番号が定義される時はもっとも小さい数を用いる。
$$i = \text{rank}_l(\text{findclose}(v + 1))$$

$$v = \text{parent}(\text{select}_l(i) + 1)$$

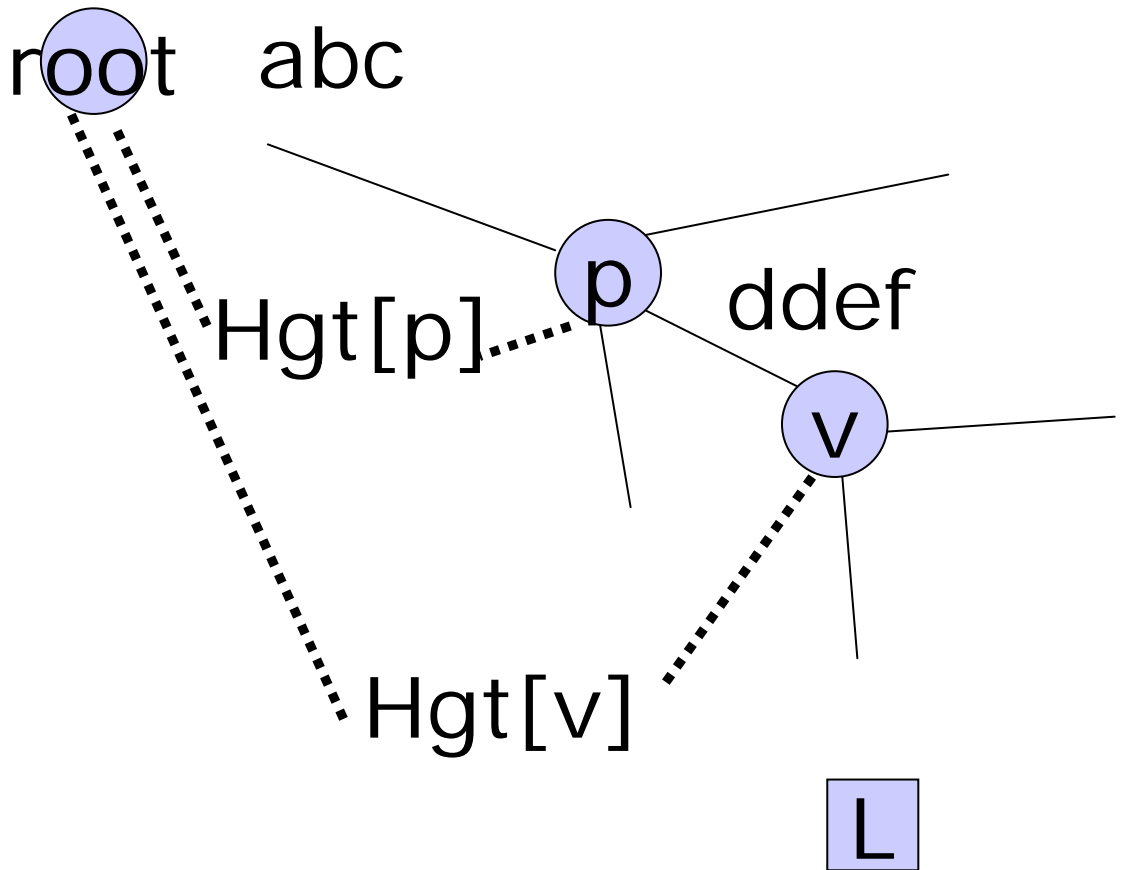
CSTの各操作(2)



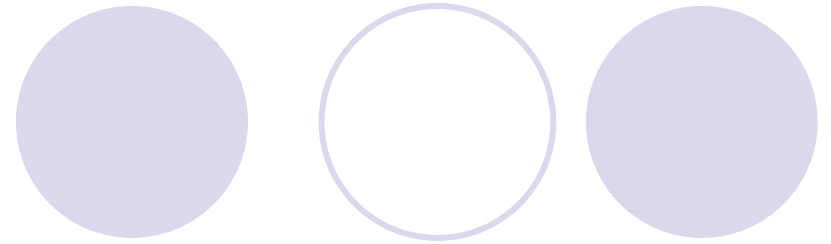
- edgeの文字列の復元 v を節点。 $\text{parent}(v)$ と v の間のedgeを復元したい。
 - $i = \text{inorder}(v)$
 - $d1 = \text{Hgt}[\text{inorder}(\text{parent}(v))]$
 - $d2 = \text{Hgt}[i]$
- edgeの部分列は
 $T[\text{SA}[i] + d1 \dots \text{SA}[i] + d2 - 1]$
- $\text{edge}(v,d) = T[\text{SA}[i] + d1 + d - 1]$

edgeの例

vのinorderをiとすると、SA[i]は葉root LのSuffixであり、[SA[i] ..]はabcddef...となる。そして、T[SA[i] + d1 ... SA[i] + d2 - 1]は、ddefをさす。

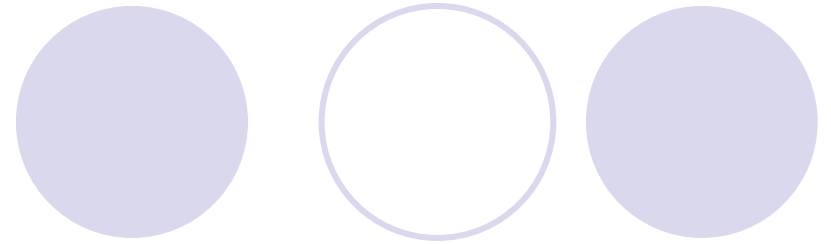


CSTの各操作(3)



- $\text{root}(v) = 1 \quad O(1)$
- $\text{sibling}(w) = \text{findclose}(P, w) + 1 \quad O(1)$
- $\text{child}(v, c) :=$ 最初の子は $v + 1$ 後はsiblingを繰り返す。 $O(|A|t_{SA})$
もしくはCSAのbinaryserch $O(\lg nt_{SA})$
- $\text{parent}(w) = \text{enclose}(w)$

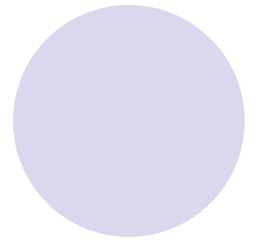
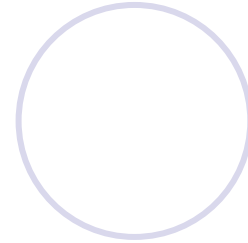
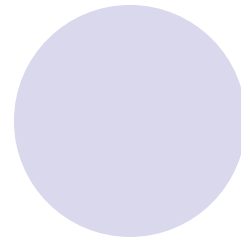
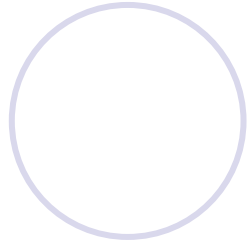
CSTの各操作(4)



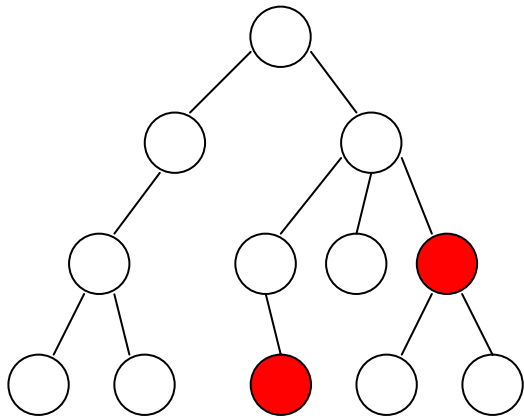
● lca

- $lca(v, w) = \text{parent}(\text{RMQ}_{P'}(v, w) + 1)$
ただし、 $P'[i] = \text{rank}_l(i) - \text{rank}_r(i)$
 $\text{RMQ}_{P'}(v, w)$ は $P'[v \dots w]$ 中の最も小さい数を持つindexを返す。(最小値を持つindexが複数あるときはもっとも小さい数)
- RMQにより返された値を m とすると、 $P[m] = \text{“} \text{”}$ 、 $P[m+1] = \text{“} (\text{“}$ である。 $P[m+1]$ の $\text{“} (\text{“}$ は、lcaの子供を意味しているなので、その親がlcaである。

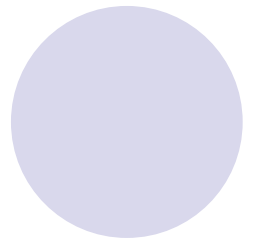
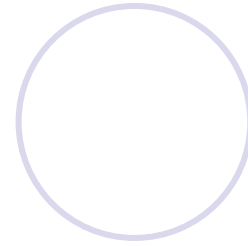
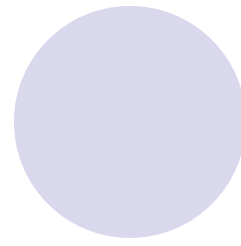
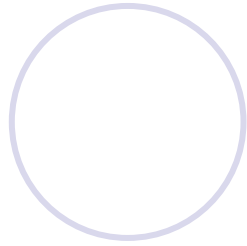
Icaの例



$((((()())))((()())()()()))$
1 2 3 4 3 4 3 2 1 2 3 4 3 2 3 2 3 4 3 4 3 2 1 0



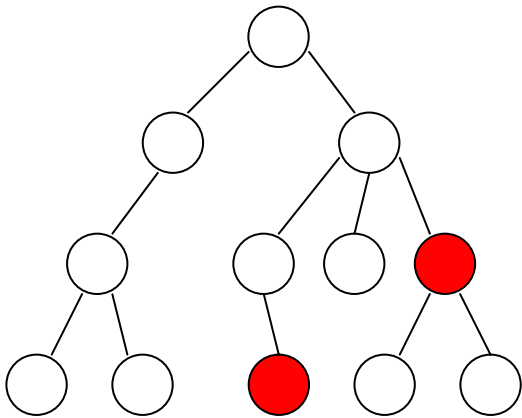
Icaの例



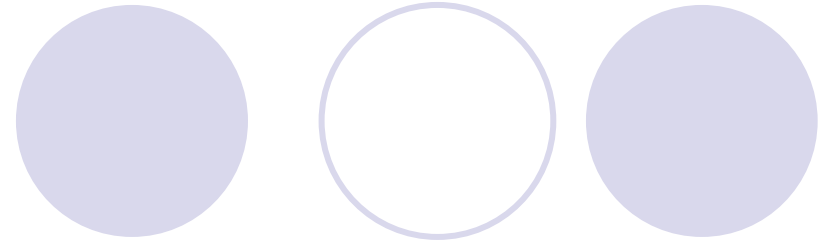
((((() ())) ((()) () (() ()))))
1 2 3 4 3 4 3 2 1 2 3 4 3 2 3 4 3 4 3 2 1 0



RMQ



CSTの各操作(5)



- suffix links

- $x = \text{rank}_0(v - 1) + 1$

- これは、 $\text{leftmost}(v)$ を表している

- $y = \text{rank}_0(\text{findclose}(v))$

- これは $\text{rightmost}(v)$ を表している

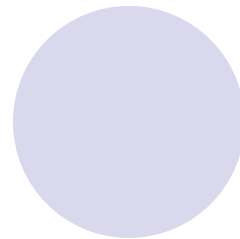
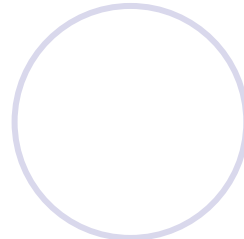
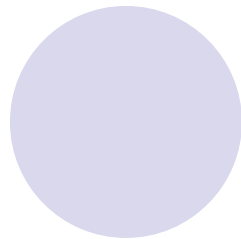
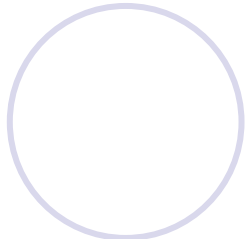
- $x' = [x]$

- $y' = [y]$

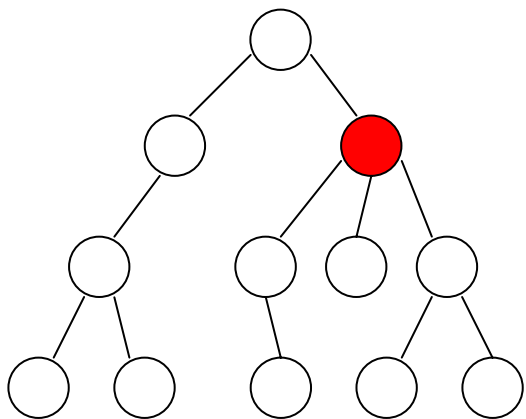
- $w = \text{lca}(\text{select}_0(x'), \text{select}_0(y'))$

- がNodeのslになっているので、節点wに対するslを求めるのに、葉の を経由する。具体的にはwの一番左側と右側の子供の葉の をそれぞれ求め、その親としてslを求める

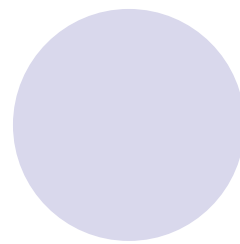
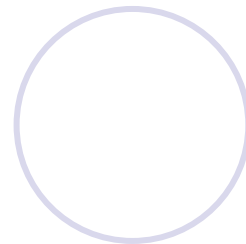
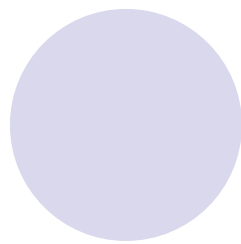
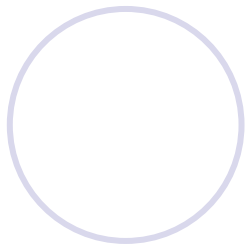
slの例



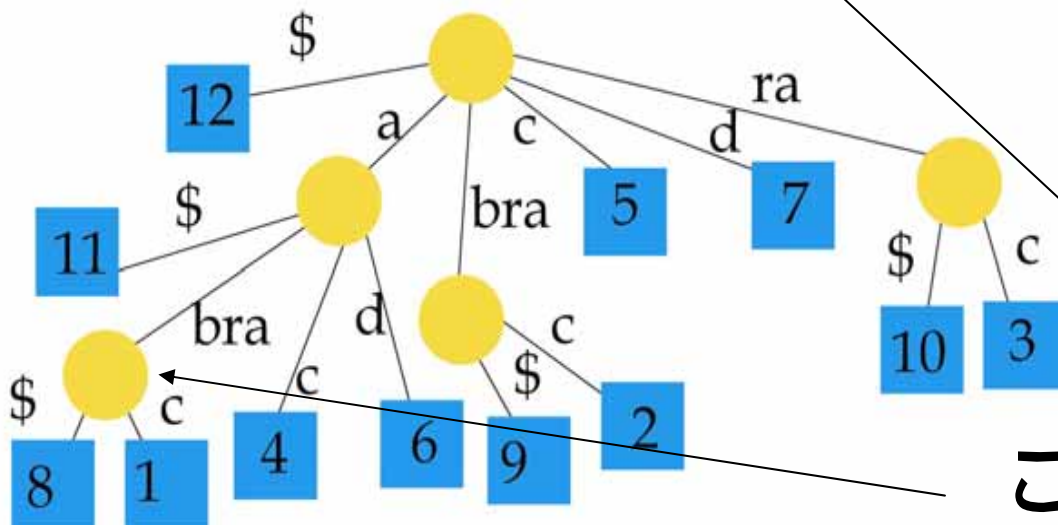
((((()))) ((()) () (() ()))))



slの例

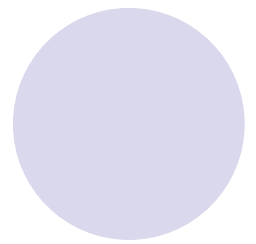
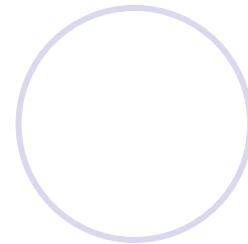
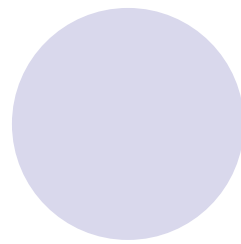
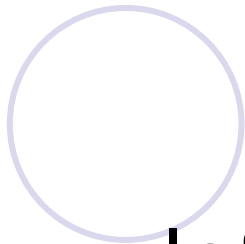


$((0(0(00)00)(00)00(00)))$



これらのslを求める

slの例

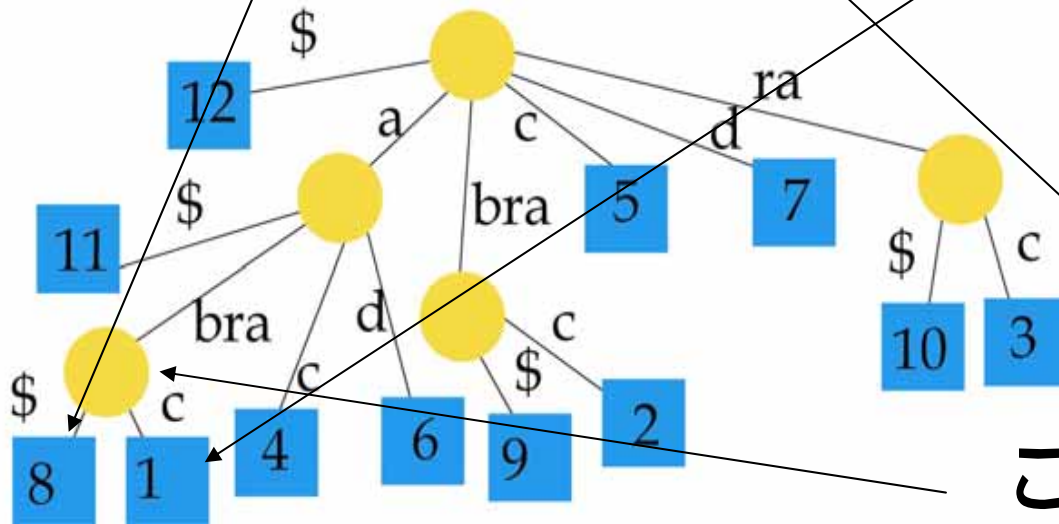


leftmost



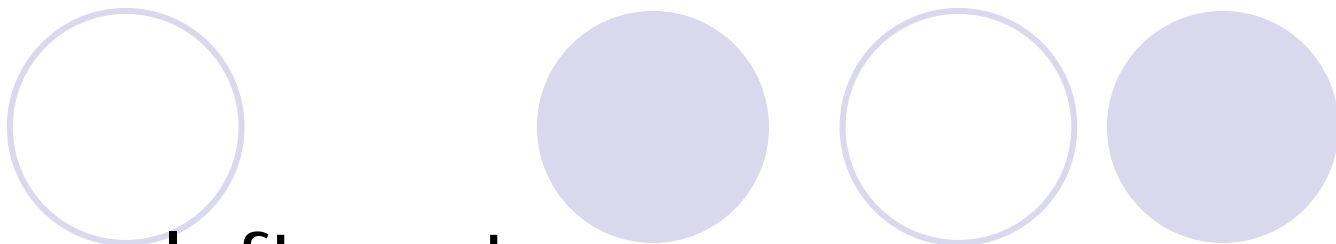
(0(0(0(0)00)(00)00(00))

rightmost



これらのslを求める

slの例

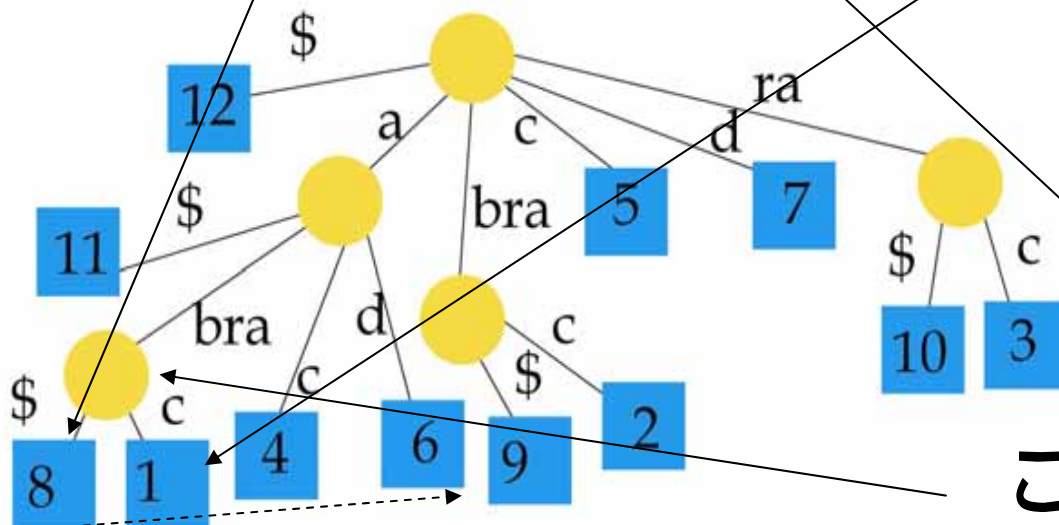


leftmost



$((0(0(0(0)00)(00)00(00)))$

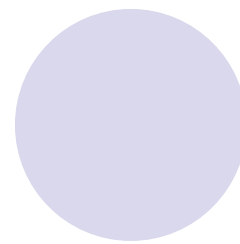
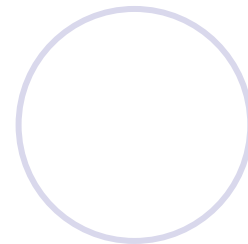
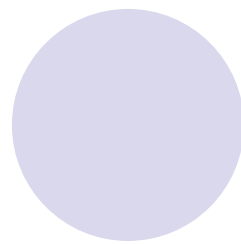
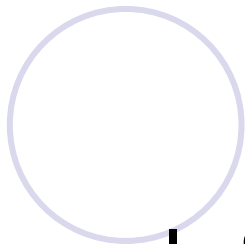
rightmost



これらのslを求める

(3)

slの例

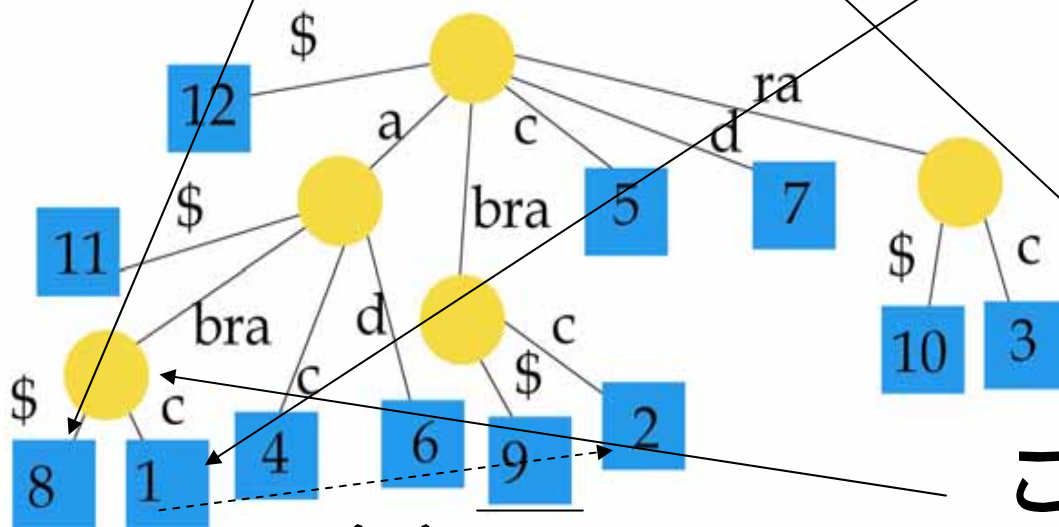


leftmost



(0(0(**0**0)00)(00)00(00))

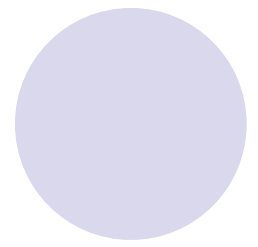
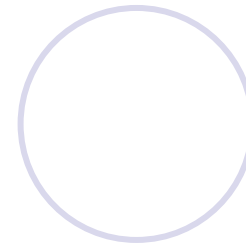
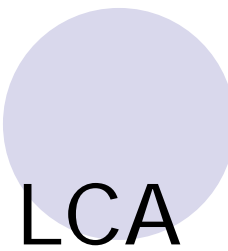
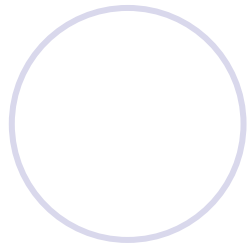
rightmost



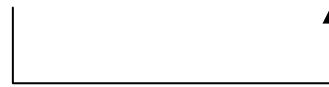
(4)

これらのslを求める

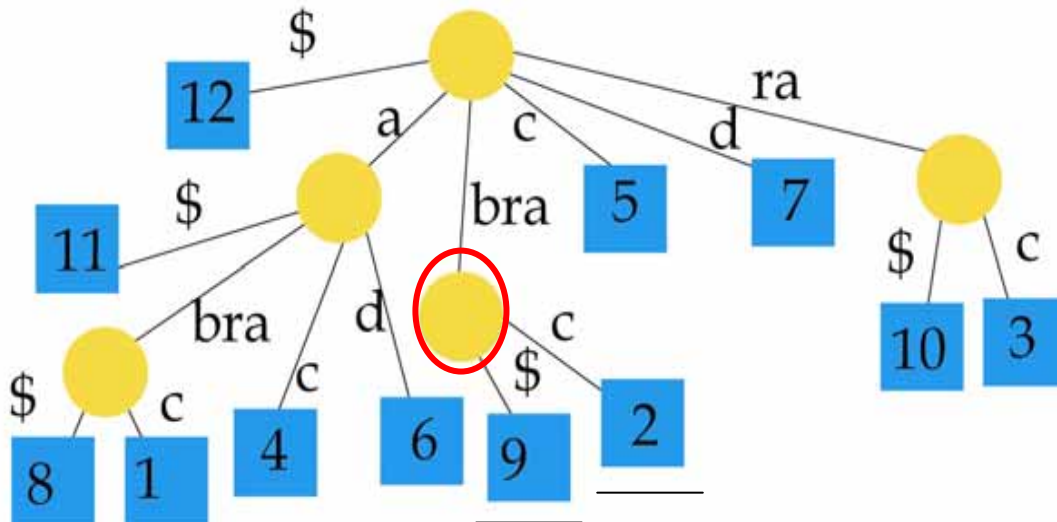
slの例



((0(0(**(0**(0)00)**(0**(0)00(00))



suffix link



CSTのまとめ

- 各操作は、定数時間または $O(\lg n)$ (`child, depth, edge, sl`) で動作。
- 領域量は $|CSA| + 6n + o(n)$ bit
6nの内訳は、4nは括弧木の表現、
2nはHgtの表現で必要

CSTの今後の課題

- $6n$ を $o(n)$ に減らすことができるか？ (by論文)
- 各操作のアクセス領域を全体から一部分に限定し、キャッシュ上に載るようにできるか
- CSTをSuffix Treesからではなく、元データから直接構築できるか？
(CSTのIncremental Algorithmなど)
- Compressed Suffix Treesの操作を拡張
(近似マッチングなど)