

STRセミナー2024

Augmented Position Heapの 先頭追加オンライン構築

柴田 紘希（九州大学）

石畠 正和（NTTコミュニケーション科学基礎研究所）

稲永 俊介（九州大学）

本研究の背景・貢献

背景:

- **文字列検索**: テキスト T ($|T| = n$) 中のパターン P ($|P| = m$) の出現を列挙する問題
- **Position Heap (PH)**: 文字列検索が行える索引構造の一種
- **Augmented Position Heap (APH)**: PHの文字列検索を高速化した索引構造
- **先頭追加オンライン構築**: T の先頭に文字が追加されたときに、 T に対する索引構造を高速に更新する手法

本研究の貢献: APHに対する高速な先頭追加オンライン構築手法を提案

本研究の背景・貢献

背景:

- **文字列検索**: テキスト T ($|T| = n$) 中のパターン P ($|P| = m$) の出現を列挙する問題
- **Position Heap (PH)**: 文字列検索が行える索引構造の一種
- **Augmented Position Heap (APH)**: PHの文字列検索を高速化した索引構造
- **先頭追加オンライン構築**: T の先頭に文字が追加されたときに、 T に対する索引構造を高速に更新する手法

本研究の貢献: **APHに対する高速な先頭追加オンライン構築手法を提案**

Position Heap

Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

0

$T = \text{abbabaa}$

Position Heap

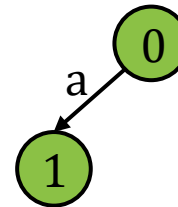
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

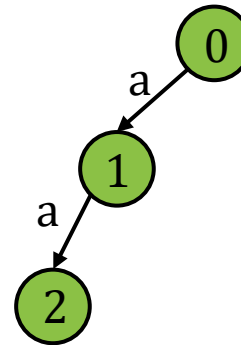
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

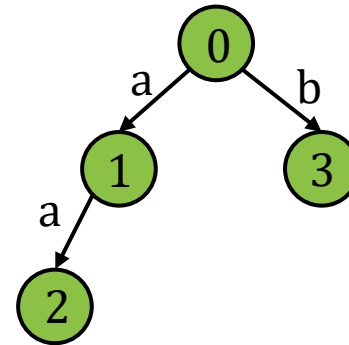
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

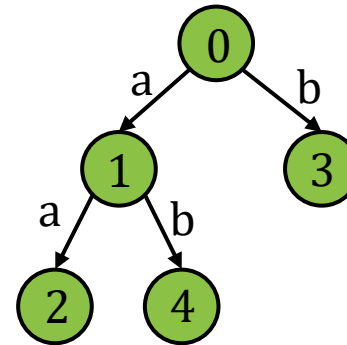
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

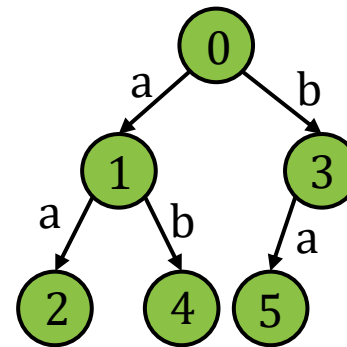
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

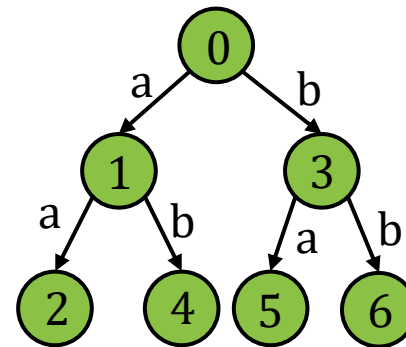
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



Position Heap

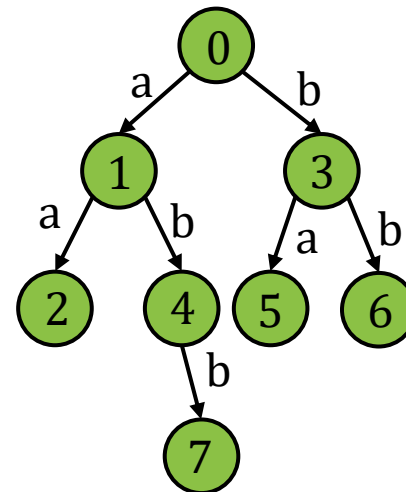
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



※ ε は空文字列

Position Heap

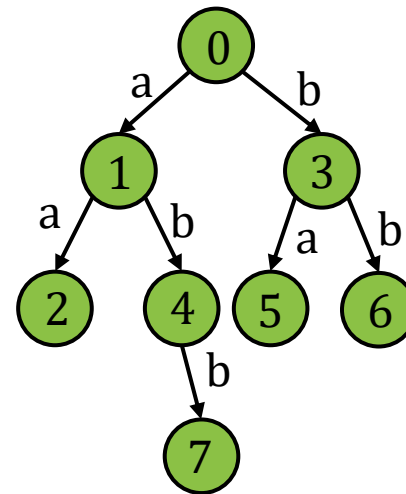
Trie構造の全文索引

Position Heap $PH(T)$: 以下のように再帰的に定義されるTrie

1. $PH(\varepsilon)$: 根頂点のみからなる 1 頂点のTrie
2. $PH(cS)$ ($c \in \Sigma, S \in \Sigma^*$): $PH(S)$ に以下の条件を満たす文字列 $cS[1..k]$ を挿入して得られる $|cS| + 1$ 頂点のTrie

※ $cS[1..k]$: $PH(S)$ に存在しないような cS の最短の接頭辞

$T = \text{abbabaa}$



※ ε は空文字列

Augmented Position Heap

追加の索引構造を保持することで検索を高速化したPosition Heap

Augmented Position Heap $APH(T)$: 以下のデータ構造の組

1. Position Heap $PH(T)$
2. $PH(T)$ 中の **maximal-reach pointer** (後述) と任意の頂点の子孫 関係を定数時間で判定できるデータ構造

嬉しさ: PHは文字列検索に $O(\min(m^2, mh(T)) + \text{occ})$ かかるが、APHであれば $O(m + \text{occ})$ 時間で文字列検索が行える

※定数アルファベットを仮定、 $h(T)$ はPHの高さ (最悪 $h(T) \in \Theta(n)$)、 occ は T 中の P の出現回数

Maximal-reach pointer

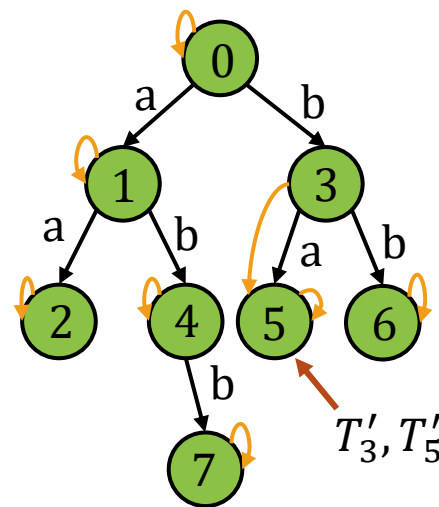
各接尾辞の「 $PH(T)$ に存在する中で最長の接頭辞」へのポインタ

$PH(T)$ の位置 i ($0 \leq i \leq n$) の **maximal-reach pointer**: 以下の条件を満たす文字列 T'_i へのポインタ

T の長さ i の接尾辞

※ T'_i : $PH(T)$ に存在するような $T[n - i + 1..n]$ の最長の接頭辞

$T = \text{abbabaa}$



Maximal-reach pointerの嬉しさ

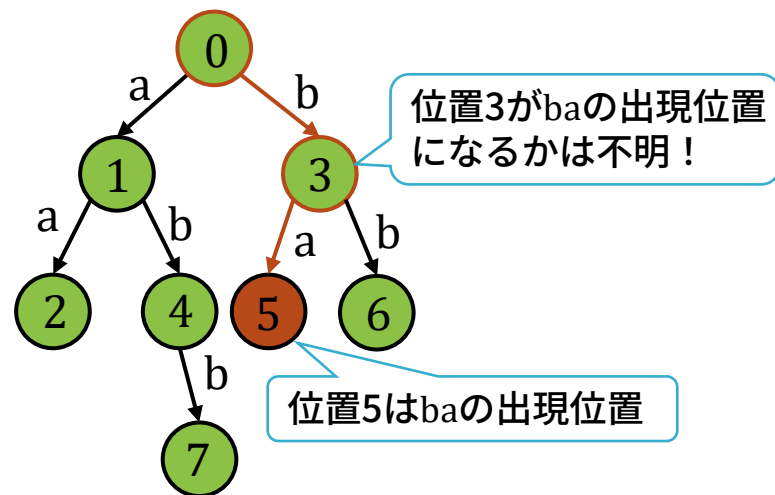
$PH(T)$ に現れる文字列

PHでは、パターン P をいくつかの文字列 $P_1, P_2, \dots$ に分割して検索する

P_j の出現の候補: P_j に対応する頂点の**部分木**か**祖先**の頂点に対応する位置のみ

- 部分木の場合: 必ず P_j の出現位置になる
- 祖先の場合: P_j の出現位置になるかは分からない

$T = \text{abbabaa}$
 $P_j = \text{ba}$



Maximal-reach pointerの嬉しさ

位置 i が P_j の出現かどうかを高速に知りたい (素のPHでは愚直に判定)

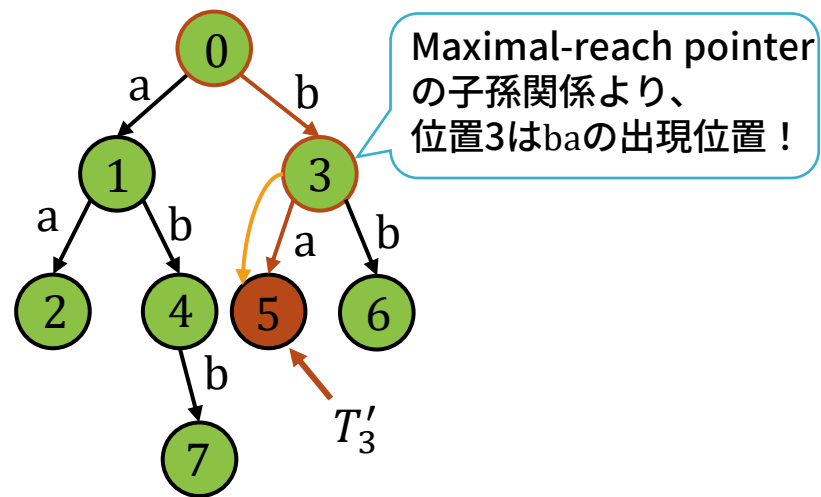
$PH(T)$ に存在するような $T[n - i + 1..n]$ の最長の接頭辞

→ T'_i が P_j を接頭辞に持つかを調べればよい

T'_i に対応する頂点へのポインタ

→ maximal-reach pointerと P_j を表す頂点の子孫関係で判定可能！

$T = \text{abbabaa}$
 $P_i = \text{ba}$



PH / APH のオンライン構築

- PHの先頭追加オンライン構築: 1文字あたりならし $O(1)$ 時間で可能
- APHの構築: どの既存手法もPHの構築後に $O(n)$ 時間かけて構築
難しい点: maximal-reach pointerのメンテナンスが困難

PH / APH のオンライン構築

- PHの先頭追加オンライン構築: ならし $O(n)$ 時間で可能
- APHの構築: どの既存手法もPHの構築後に $O(n)$ 時間かけて構築
難しい点: maximal-reach pointerのメンテナンスが困難

本研究の解決策:

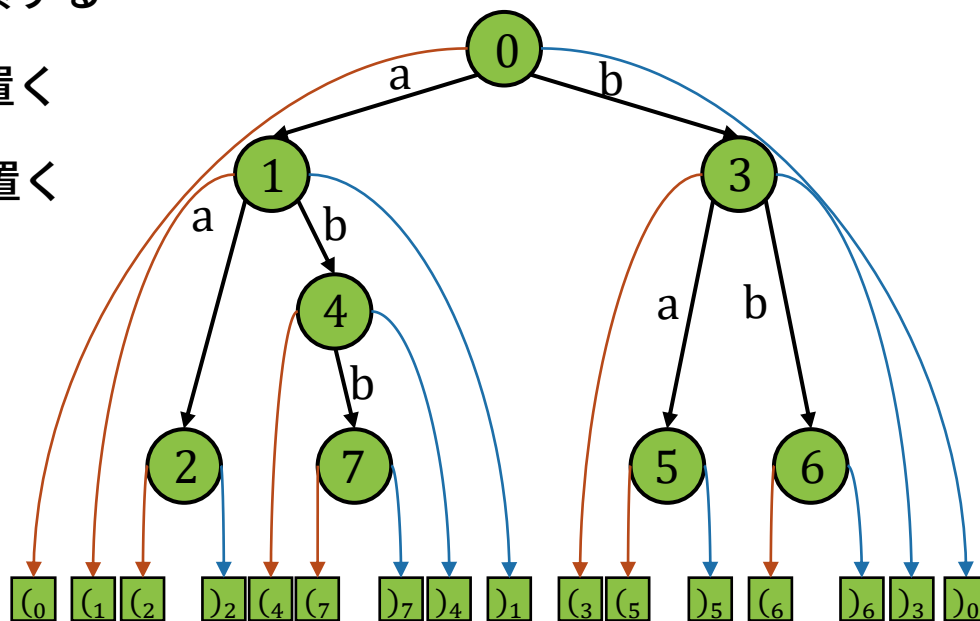
- 木の括弧列表現・文字列間の辞書順・平衡二分木を組み合わせて maximal-reach pointerを仮想的に表現
- Order-maintenance problemのデータ構造を用いて検索の時間計算量を維持

木の括弧列表現

■ Trieの構造は括弧列を用いて表現できる

辺ラベルの辞書順に深さ優先探索する

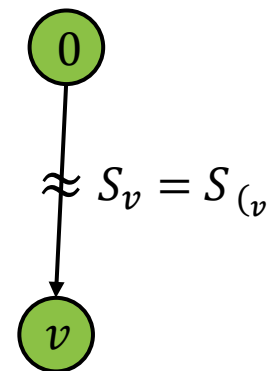
- 頂点 v の探索開始時に $(_v$ を置く
- 頂点 v の探索終了時に $)_v$ を置く



木の括弧列表現と辞書順の関係

■ Trieの括弧列表現は各頂点の辞書順で表現できる

- Trieの頂点 v が表す文字列を S_v とする
- 各頂点 v に対応する開き括弧 $(_v$ と閉じ括弧 $)_v$ に対して以下のように文字列 $S_{(_v}, S_{)_v}$ を定義し、開き・閉じ括弧の集合を C とする
 - ▲ $S_{(_v} = S_v$
 - ▲ $S_{)_v} = S_v \#$



事実

C の各要素 $x \in C$ を S_x の辞書順に並べ替えることでTrieの括弧列表現を得られる

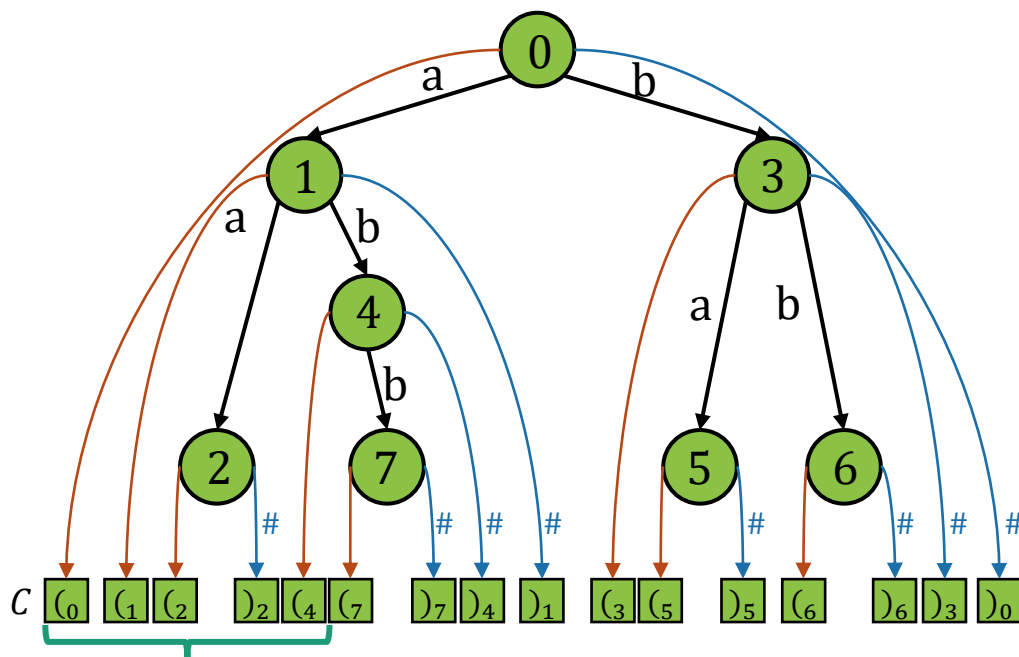
木の括弧列表現と辞書順の関係

事実

C の各要素 $x \in C$ を S_x の辞書順に並べ替えることでTrieの括弧列表現を得られる

C の各要素に対応する文字列:

- $S_{(0)} = \varepsilon$
- $S_{(1)} = a$
- $S_{(2)} = aa$
- $S_{(2)} = aa\#$
- $S_{(4)} = ab$



※ $\# \in \Sigma$ は T, P に現れない文字で、 $\forall c \in \Sigma \setminus \{\#\}, c < \#$ を満たすものとする

各接尾辞の「PH内に存在する最長の接頭辞」へのポインタ

Maximal-reach pointerと括弧列表現

Maximal-reach pointerを括弧列と一緒に順序付けする

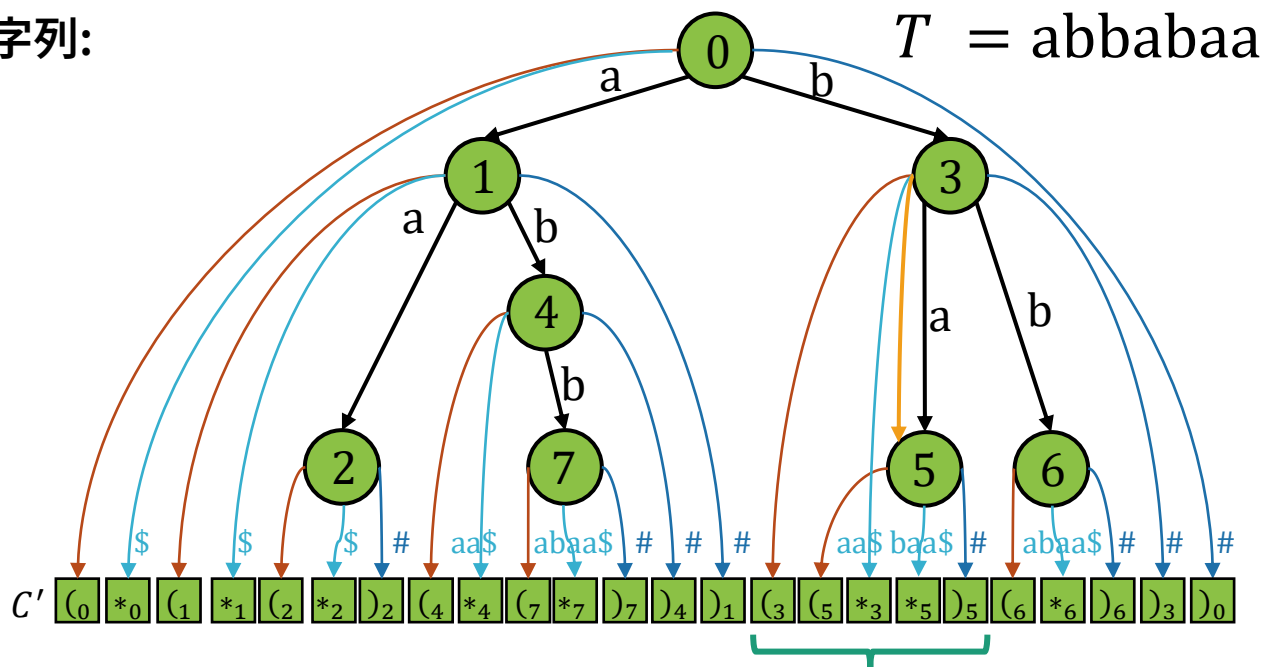
位置 i の maximal-reach pointer に対応する記号 $*_i$ ($0 \leq i \leq n$) を導入し、

$S_{*_i} = T[n - i + 1..n]\$$ とする

開き・閉じ括弧と $*_i$ の集合を C' とし、各要素 $x \in C'$ を S_x の辞書順に並べ替える

C' の各要素に対応する文字列:

- $S_{(3)} = b$
- $S_{(5)} = ba$
- $S_{*_3} = baa\$$
- $S_{*_5} = ba baa\$$
- $S_{)_5} = ba\#$



※ $\$ \in \Sigma$ は T, P に現れない文字で、 $\forall c \in \Sigma \setminus \{\$, \# \}, \$ < c$ を満たすものとする

各接尾辞の「PH内に存在する最長の接頭辞」へのポインタ

Maximal-reach pointerと括弧列表現

性質

開き・閉じ括弧と $*_i$ の集合 C' の各要素 $x \in C'$ を S_x の辞書順に並べ替えた表現から、maximal-reach pointerと各頂点の子孫関係を復元できる

再掲: C' の各要素に対応する文字列は以下のように定義

- $S_{(v)} = S_v$
- $S_{)v} = S_v \#$
- $S_{*_i} = T[i..n] \$$

$S_v < T[n+i-1..n] \$ < S_v \#$ と同値
 $\rightarrow T[n+i-1..n]$ が S_v を接頭辞に持つ

位置 i の maximal-reach pointer と頂点 v の子孫関係は $(v < *_i <)_v$ で判定できる

各接尾辞の「PH内に存在する最長の接頭辞」へのポインタ

Maximal-reach pointerと括弧列表現

位置 i の maximal-reach pointer と頂点 v の子孫関係は $(v < *i <)_v$ で判定できる

$S_v < T[n + i - 1..n]\$ < S_v\#$ と同値
 $\rightarrow T[n + i - 1..n]$ が S_v を接頭辞に持つ

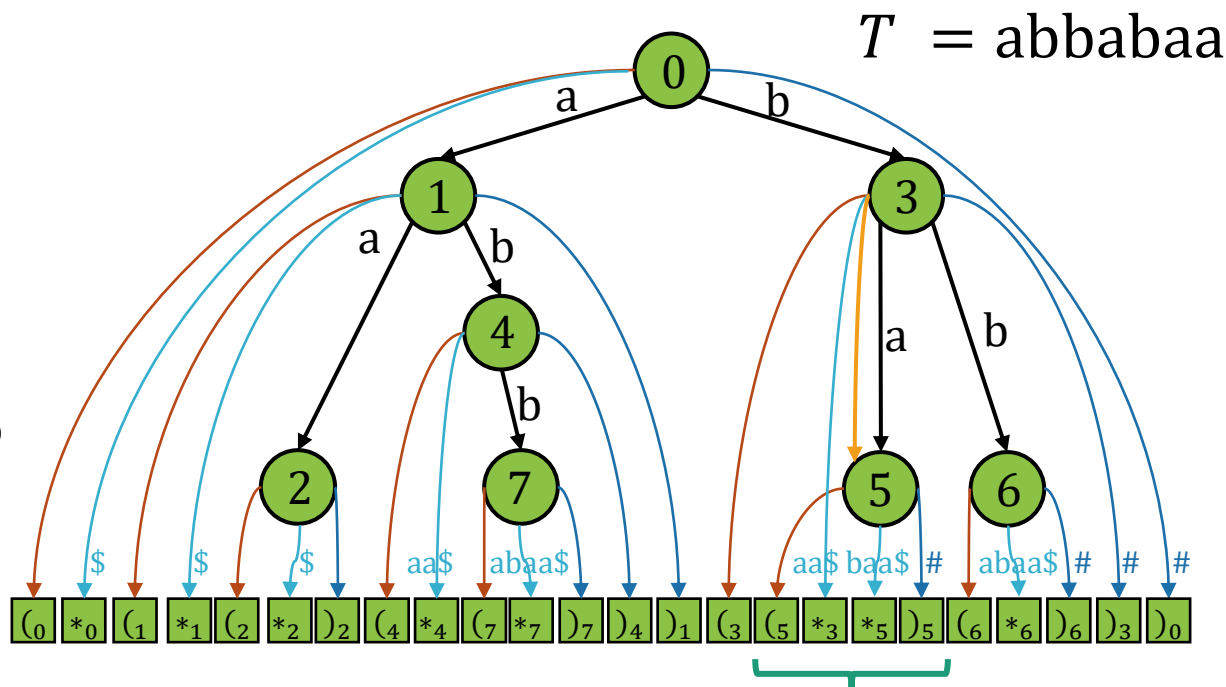
■ $S_{(5)} = ba$

■ $S_{*3} = baa\$$

■ $S_{)5} = ba\#$

より $(5 < *3 <)_5$ が成り立つ

$\rightarrow$ 位置 3 の maximal-reach pointer は頂点 5 の子孫



C' を管理するデータ構造

提案手法: $PH(T)$ と C' の組で augmented position heap を表現

APHの定義（再掲）

$APH(T)$ は以下の2つのデータ構造の組である:

1. $PH(T)$
2. $PH(T)$ 中の maximal-reach pointer と任意の頂点の 子孫関係を定数時間で判定できるデータ構造

C' は以下の2つの要件を満たす必要がある

1. 動的更新が可能（オンライン構築のため）
2. Maximal-reach pointerの子孫関係が定数時間で判定可能

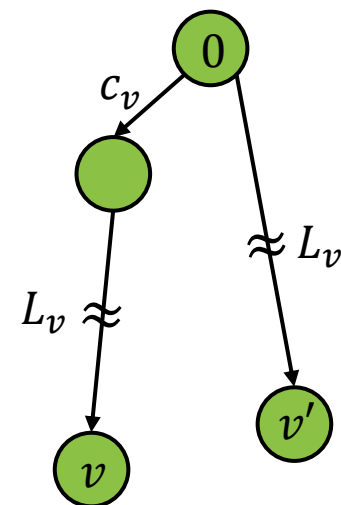
→ C' の要素を S_x の辞書順に並べた列を **平衡二分木** で管理する

S_x の再帰的な表現

$PH(T)$ の根以外の各頂点 v について、以下を定義

- $c_v \in \Sigma, L_v \in \Sigma^* : S_v = c_v L_v$ を満たす文字と文字列
- $v' : S_{v'} = L_v$ を満たす頂点

※このような頂点 v' は必ず存在 (証明略)



S_x の再帰的な表現を用いた順序比較

平衡二分木の各ノードに対応する文字列 $S_{(v)}, S_{)v}, S_{*i}$ の再帰的な表現:

- $S_{(v)} = S_v = c_v L_v = c S_{(v)'}$
- $S_{)v} = S_v \# = c_v L_v \# = c S_{)v}'$
- $S_{*i} = T[n + i - 1..|T|] = c_i T[n + i - 2..|T|] = c_i S_{*i+1}$
 ※ただし、 $c_i = T[n + i - 1]$

この表現を用いることで、平衡二分木のノード順序の比較が「**1文字比較** + **平衡二分木の既存ノードの順序比較**」によって実現できる

これを用いて新しいノードの挿入位置を二分探索することで、 $O(\log n)$ 回の比較でノードの挿入位置が特定できる

平衡二分木のノードの比較

現在の問題点: 平衡二分木のノード間順序の比較は**定数時間**で行えない

→ Order-maintenance problem用のデータ構造を用いて解決

Order-maintenance problem: 列に対する以下のクエリを処理する問題

1. 列の要素 u の直後に v を挿入
2. 列の要素 u, v について、 u が列上で v より前に存在するかを判定

定理 [Bender et al., 2002]

order-maintenance problemの各処理を定数時間で行えるデータ構造が存在する

平衡二分木のノードの比較

現在の問題点: 平衡二分木のノード間順序の比較は**定数時間**で行えない

→ Order-maintenance problem用のデータ構造を用いて解決

Order-maintenance problem: 列に対する以下のクエリを処理する問題

1. 列の要素 u の直後に v を挿入
2. 列の要素 u, v について、 u が列上で v より前に存在するかを判定

定理 [Bender et al., 2002]

order-maintenance problemの各処理を定数時間で行えるデータ構造が存在する

平衡二分木のノードの比較

Order-maintenance problem: 列に対する以下のクエリを処理する問題

1. 列の要素 u の直後に v を挿入
2. 列の要素 u, v について、 u が列上で v より前に存在するかを判定

定理 [Bender et al., 2002]

order-maintenance problemの各処理を定数時間で行えるデータ構造が存在する

平衡二分木の各ノードにorder-maintenance problemのデータ構造中の対応するノードへのポインタを保持することで、比較の計算量が $O(1)$ になる

データ構造のまとめと計算量

■ $APH(T)$ のデータ構造:

1. Position Heap $PH(T)$
2. C' を適切な順序で管理する平衡二分木
3. order-maintenance problemのデータ構造

■ 各処理の時間計算量

- Maximal-reach pointerと頂点の子孫関係の取得: $O(1)$

※平衡二分木中のノード比較に帰着できる

- T への1文字追加時の $APH(T)$ の更新: **ならし $O(\log n)$**

▲ $PH(T)$ の更新がならし定数時間

▲ 2., 3. のデータ構造の更新が $O(\log n)$ 時間

※挿入位置の特定と木の平衡化にそれぞれ $O(\log n)$ かかる

まとめ・展望

Augmented Position Heapの先頭追加オンライン構築手法を提案

- 1文字追加の計算量はならし $O(\log n)$ 時間
- 検索の計算量は元のAPHと変わらず $O(|P| + \text{occ})$ 時間

発表資料作成中に気づきました😊

さらなる改善

- 更新計算量 $O(\log n)$ から $O(\log h)$ への改善
 - 連続する $*_i$ の個数が $O(h)$ 個であることからうまく示せる
- 接尾辞木を使ったシミュレート
 - PHは接尾辞木の部分構造
 - 接尾辞木をオンライン構築しながらPHをシミュレートできる