Karlsruhe Institute of Technology

Advanced Data Structures

Lecture 07: Suffix Arrays and String B-Trees

Florian Kurpicz

PINGO

https://pingo.scc.kit.edu/172581

External Memory Model [AV88]

Definition: External Memory Model

- internal memory of M words
- instances of size $N \gg M$
- unlimited external memory
- transfer blocks of size B between memories

- measure number of blocks I/Os
- scanning N elements: $\Theta(N/B)$
- sorting N elements: $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$

External Memory Model [AV88]

Definition: External Memory Model

- internal memory of M words
- instances of size $N \gg M$
- unlimited external memory
- transfer blocks of size B between memories

- measure number of blocks I/Os
- scanning N elements: $\Theta(N/B)$
- sorting N elements: $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$

Set of Strings

- alphabet Σ of size σ
- k strings $\{s_1, \ldots, s_k\}$ over the alphabet Σ
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern P of length m

String Dictionary

Given a set $S \subseteq \Sigma^\star$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^\star$ in S
- add $x \notin S$ to S
- remove $x \in S$ from S
- predecessor and successor of $x \in \Sigma^\star$ in S

String Dictionary

Given a set $S \subseteq \Sigma^\star$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^\star$ in S
- add $x \notin S$ to S
- remove $x \in S$ from S
- predecessor and successor of $x \in \Sigma^\star$ in S

Definition: Trie

Given a set $S = \{S_1, \dots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. k leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is S_i
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique

String Dictionary

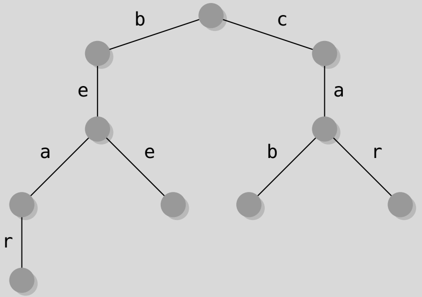
Given a set $S \subseteq \Sigma^\star$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^\star$ in S
- add $x \notin S$ to S
- remove $x \in S$ from S
- predecessor and successor of $x \in \Sigma^\star$ in S

Definition: Trie

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. k leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is S_i
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

Theoretical Comparison

Representation	Query Time (Contains)	Space in Words
arrays of variable size	$O(m \cdot \sigma)$	$O(N)$
arrays of fixed size	$O(m)$	$O(N \cdot \sigma)$
hash tables	$O(m)$ w.h.p.	$O(N)$
balanced search trees	$O(m \cdot \lg \sigma)$	$O(N)$
weight-balanced search trees	$O(m + \lg k)$	$O(N)$
two-levels with weight-balanced search trees	$O(m + \lg \sigma)$	$O(N)$

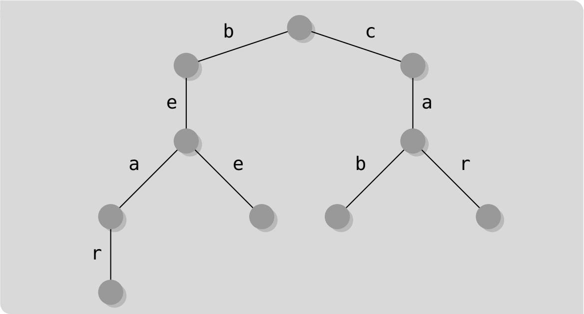
Theoretical Comparison

Representation	Query Time (Contains)	Space in Words
arrays of variable size	$O(m \cdot \sigma)$	$O(N)$
arrays of fixed size	$O(m)$	$O(N \cdot \sigma)$
hash tables	$O(m)$ w.h.p.	$O(N)$
balanced search trees	$O(m \cdot \lg \sigma)$	$O(N)$
weight-balanced search trees	$O(m + \lg k)$	$O(N)$
two-levels with weight-balanced search trees	$O(m + \lg \sigma)$	$O(N)$

- more details in lecture Text Indexing

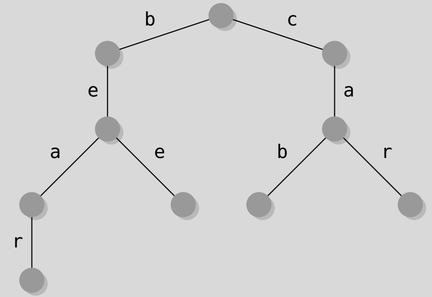
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters



Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters



Definition: Compact Trie

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges' labels.

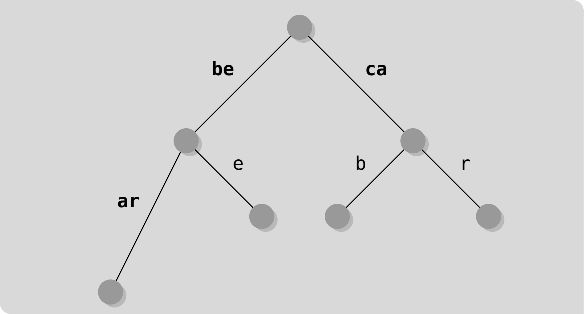
Compact Trie

Karlsruhe Institute of Technology

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

Definition: Compact Trie

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges' labels.



Suffix Array and LCP-Array

Karlsruhe Institute of Technology

Definition: Suffix Array [GBS92; MM93]

Given a text T of length n, the **suffix array** (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	\$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
	\$	a\$	ababcabcabba\$	abba\$	abcabba\$	abcabcabba\$	ba\$	babcabcabba\$	bba\$	bcabba\$	bcabcabba\$	cabba\$	cabcabba\$

Suffix Array and LCP-Array

Definition: Suffix Array [GBS92; MM93]

Given a text T of length n, the **suffix array** (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

Definition: Longest Common Prefix Array

Given a text T of length n and its SA, the **LCP-array** is defined as

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell \colon T[SA[i]..SA[i]+\ell) = \\ \qquad T[SA[i-1]..SA[i-1]+\ell)\} & i \neq 1 \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
	$	a$	ababcabcabba$	abba$	abcabba$	abcabcabba$	ba$	babcabcabba$	bba$	bcabba$	bcabcabba$	cabba$	cabcabba$

Suffix Array and LCP-Array

Definition: Suffix Array [GBS92; MM93]

Given a text T of length n, the **suffix array** (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

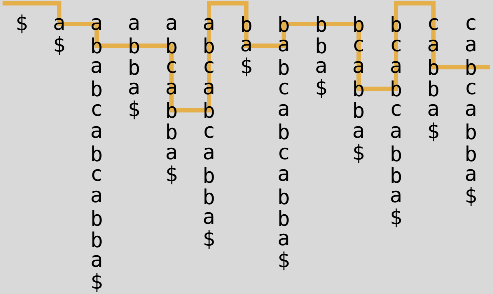
$$T[SA[i]..n] \leq T[SA[j]..n]$$

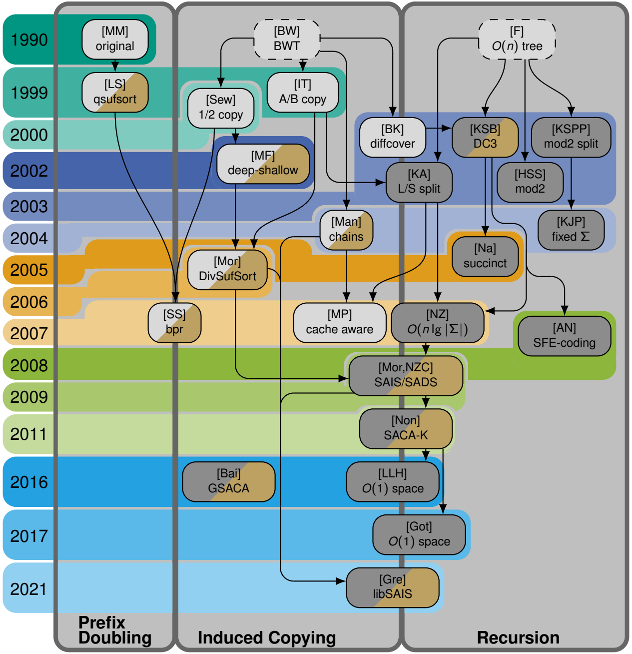
Definition: Longest Common Prefix Array

Given a text T of length n and its SA, the **LCP-array** is defined as

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell \colon T[SA[i]..SA[i]+\ell) = \\ \qquad T[SA[i-1]..SA[i-1]+\ell)\} & i \neq 1 \end{cases}$$

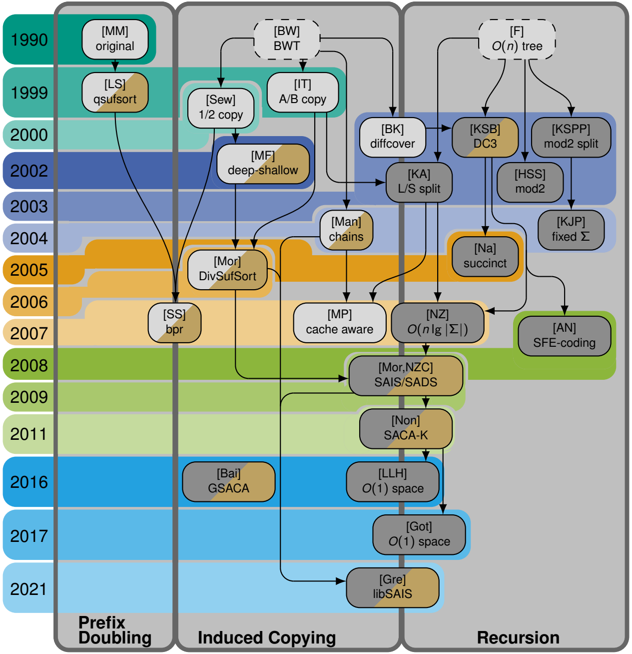
	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3





Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- **darker grey**: linear running time
- **brown**: available implementation

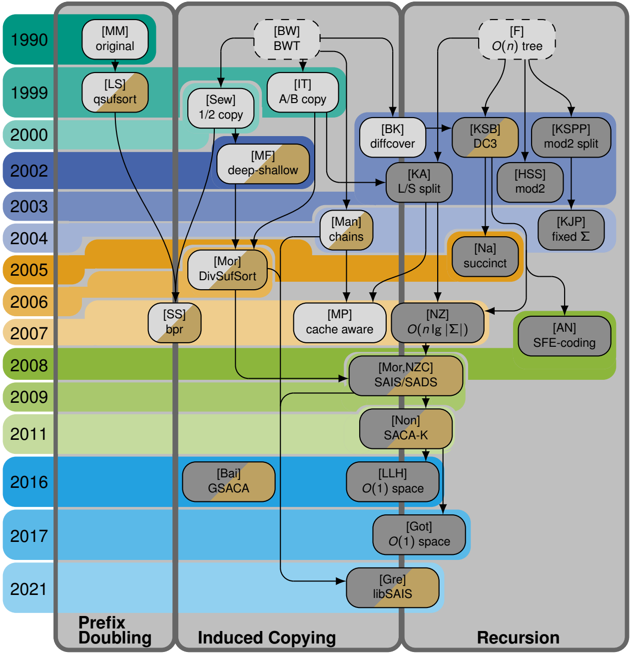


Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- **darker grey**: linear running time
- **brown**: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible

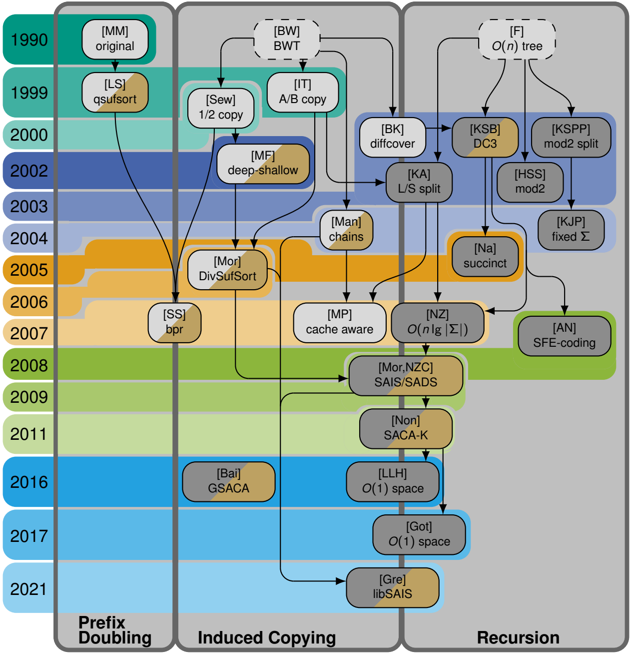


Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- **darker grey**: linear running time
- **brown**: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time



Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- **darker grey**: linear running time
- **brown**: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time
- since 2021: libSAIS fastest in practice with $O(n)$ running time

Suffix Sorting in External Memory

- best in practice: Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. “Engineering External Memory Induced Suffix Sorting”. In: *ALENEX*. SIAM, 2017, pages 98–108. DOI: 10.1137/1.9781611974768.8
- using induced copying
- $O(N/B)\log^2_{\frac{M}{B}}(N/B)$ I/Os

Pattern Matching with the Suffix Array (1/2)

Karlsruhe Institute of Technology

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

```
ℓ = 1, r = n + 1
while ℓ < r do
    i = ⌊(ℓ + r)/2⌋
    if P > T[SA[i]..SA[i] + m) then
        ℓ = i + 1
    else r = i
s = ℓ, ℓ = ℓ − 1, r = n
while ℓ < r do
    i = ⌈ℓ + r/2⌉
    if P = T[SA[i]..SA[i] + m) then ℓ = i
    else r = i − 1
return [s, r]
```

- pattern $P = \texttt{abc}$

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
	$	a$	ababcabcabba$	abba$	abcabba$	abcabcabba$	ba$	babcabcabba$	bba$	bcabba$	bcabcabba$	cabba$	cabcabba$

Pattern Matching with the Suffix Array (1/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do** ⓘ Find left border
  - $i = \lfloor (\ell + r)/2 \rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

- pattern $P =$ abc

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
	$	a$	ababcabcabba$	abba$	abcabba$	abcabcabba$	ba$	babcabcabba$	bba$	bcabba$	bcabcabba$	cabba$	cabcabba$

Pattern Matching with the Suffix Array (1/2)

Function SeachSA($T, SA[1..n], P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do** Find left border
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do** Find right border
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

- pattern $P =$ abc

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
	$	a$	ababcabcabba$	abba$	abcabba$	abcabcabba$	ba$	babcabcabba$	bba$	bcabba$	bcabcabba$	cabba$	cabcabba$

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$):

$\ell = 1, r = n + 1$

while $\ell < r$ **do**

  $i = \lfloor(\ell + r)/2\rfloor$

  if $P > T[SA[i]..SA[i] + m)$ **then**

   $\ell = i + 1$

  else $r = i$

$s = \ell, \ell = \ell - 1, r = n$

while $\ell < r$ **do**

  $i = \lceil \ell + r/2 \rceil$

  if $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$

  else $r = i - 1$

return $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then** $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the *SA* in $O(lgn)$ time
- each comparison requires $O(m)$ time

Pattern Matching with the Suffix Array (2/2)

Karlsruhe Institute of Technology

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time
- each comparison requires $O(m)$ time

Pattern Matching with the Suffix Array (2/2)

Karlsruhe Institute of Technology

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor (\ell + r)/2 \rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

$\ell = 1, r = n + 1$
while $\ell < r$ **do**
  $i = \lfloor(\ell + r)/2\rfloor$
  if $P > T[SA[i]..SA[i] + m)$ **then**
   $\ell = i + 1$
  else $r = i$
$s = \ell, \ell = \ell - 1, r = n$
while $\ell < r$ **do**
  $i = \lceil \ell + r/2 \rceil$
  if $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  else $r = i - 1$
return $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the *SA* in $O(lgn)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time
- reporting in $O(occ)$ additional time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

```
ℓ = 1, r = n + 1
while ℓ < r do
    i = ⌊(ℓ + r)/2⌋
    if P > T[SA[i]..SA[i] + m) then
        ℓ = i + 1
    else r = i
s = ℓ, ℓ = ℓ − 1, r = n
while ℓ < r do
    i = ⌈ℓ + r/2⌉
    if P = T[SA[i]..SA[i] + m) then ℓ = i
    else r = i − 1
return [s, r]
```

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the SA in $O(lgn)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time
- reporting in $O(occ)$ additional time

Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, $SA[1..n]$, $P[1..m]$)**:**

- $\ell = 1, r = n + 1$
- **while** $\ell < r$ **do**
  - $i = \lfloor(\ell + r)/2\rfloor$
  - **if** $P > T[SA[i]..SA[i] + m)$ **then**
    - $\ell = i + 1$
  - **else** $r = i$
- $s = \ell, \ell = \ell - 1, r = n$
- **while** $\ell < r$ **do**
  - $i = \lceil \ell + r/2 \rceil$
  - **if** $P = T[SA[i]..SA[i] + m)$ **then** $\ell = i$
  - **else** $r = i - 1$
- **return** $[s, r]$

Lemma: Running Time SeachSA

The SeachSA answers counting queries in $O(m \lg n)$ time and reporting queries in $O(m \lg n + occ)$ time

Proof (Sketch)

- two binary searches on the *SA* in $O(lgn)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time
- reporting in $O(occ)$ additional time

- how can this be improved? **PINGO**

Speeding Up Pattern Matching with the LCP-Array (1/4)

- remember how many characters of the pattern and suffix match
- identify how long the prefix of the old and next suffix is
- do so using the LCP-array and
- range minimum queries

Definition: Range Minimum Queries

Given an array $A[1..m)$, a **range minimum query** for a range $\ell \leq r \in [1, n)$ returns

$$RMQ_A(\ell, r) = \arg\min\{A[k] \colon k \in [\ell, r]\}$$

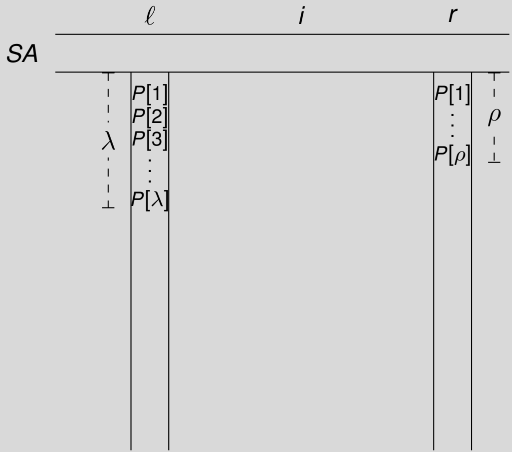
- $lcp(i, j) = \max\{k \colon T[i..i+k)$
- $lcp(i, j) = T[j..j+k)\} = LCP[RMQ_{LCP}(i+1, j)]$
- RMQs can be answered in $O(1)$ time and
- require $O(n)$ space

Speeding Up Pattern Matching with the LCP-Array (2/4)

- during binary search matched
- λ characters with left border ℓ and
- ρ characters with right border r
- w.l.o.g. let $\lambda \geq \rho$

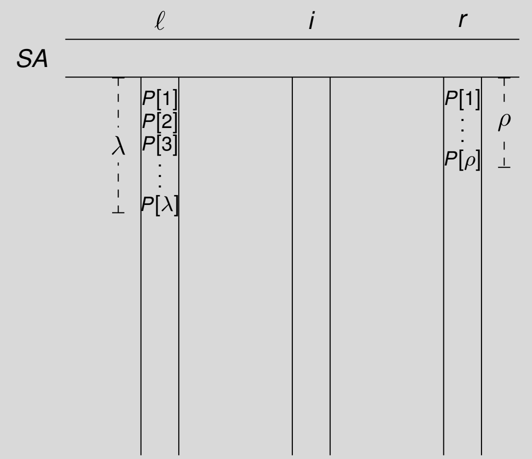
- middle position i
- decide if continue in $[\ell, i]$ or $[i, r]$

- let $\xi = lcp(SA[\ell], SA[i])$ ℹ $O(1)$ time with RMQs



Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

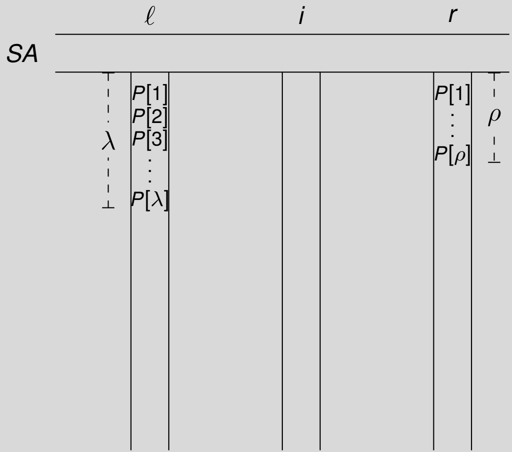


Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

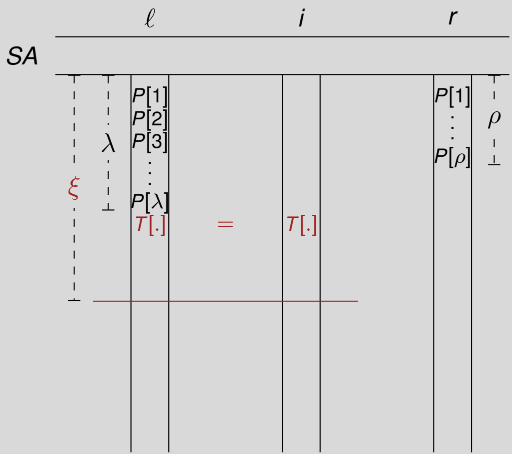


Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

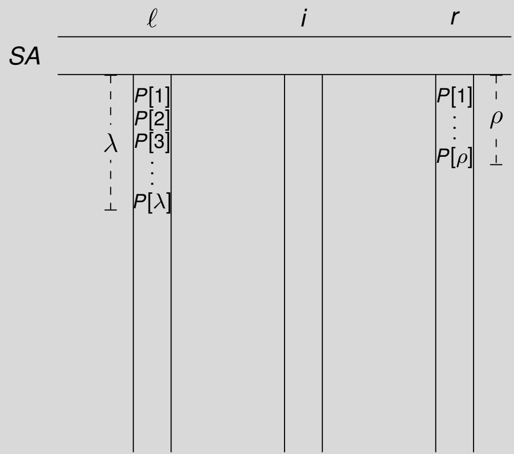


Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison



Speeding Up Pattern Matching with the LCP-Array (3/4)

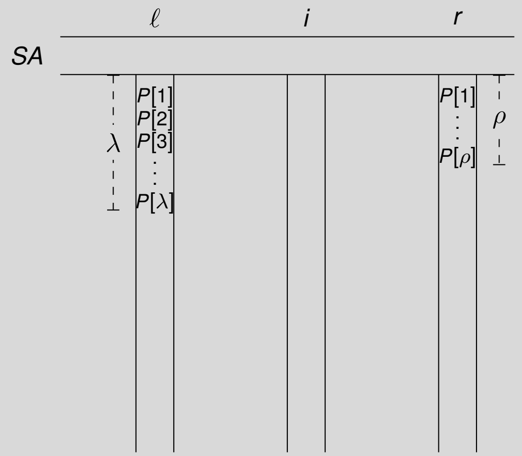
- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before



Speeding Up Pattern Matching with the LCP-Array (3/4)

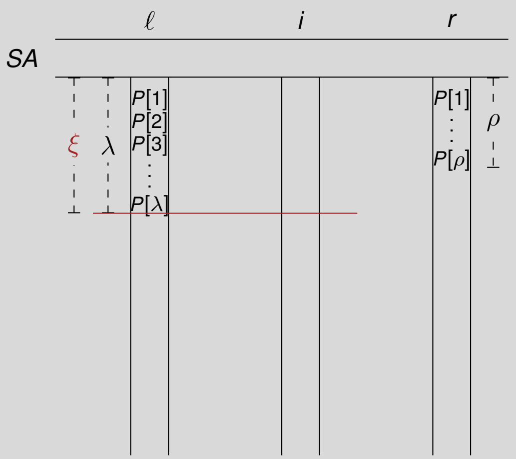
- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before



Speeding Up Pattern Matching with the LCP-Array (3/4)

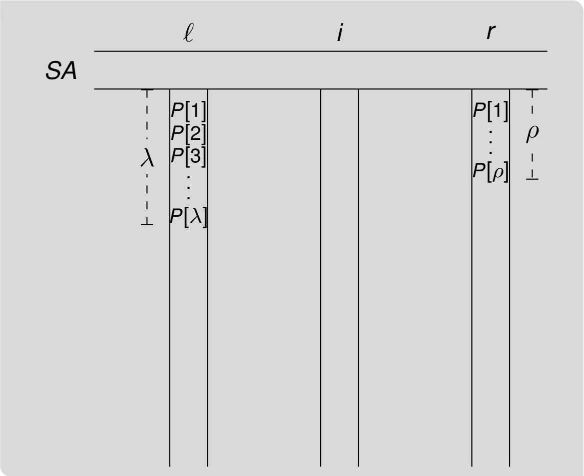
- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before



Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

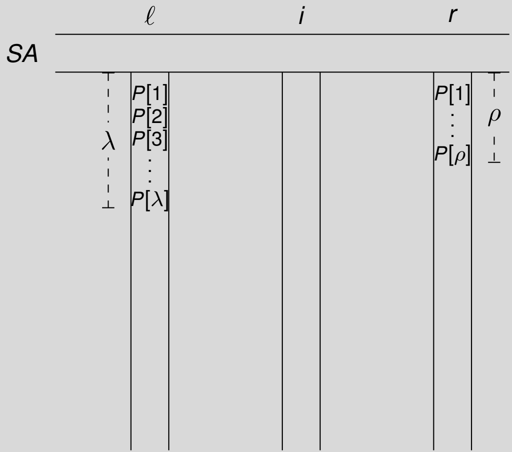
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before

$\xi < \lambda$

- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison



Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

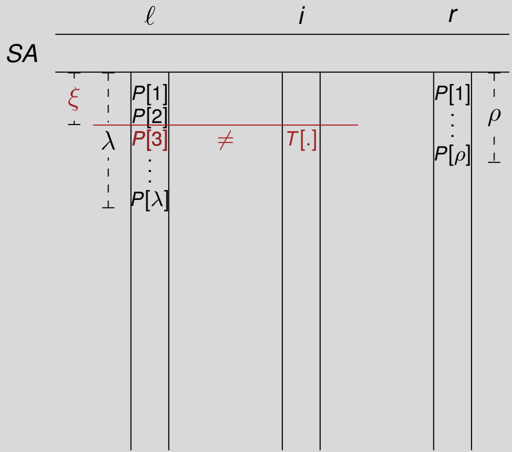
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before

$\xi < \lambda$

- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison



Speeding Up Pattern Matching with the LCP-Array (4/4)

Lemma:

Using RMQs, `SeachSA` answers counting queries in $O(m + \lg n)$ time and reporting queries in $O(m + \lg n + occ)$ time

Speeding Up Pattern Matching with the LCP-Array (4/4)

Lemma:

Using RMQs, `SeachSA` answers counting queries in $O(m + \lg n)$ time and reporting queries in $O(m + \lg n + occ)$ time

Proof (Sketch)

- either halve the range in the suffix array ($\xi \neq \lambda$) or
- compare characters of the pattern (at most m)

(Recap) B-Trees

- search tree with out-degree in $[b, 2b)$
- works well in external memory
- uses separators to find subtree
- can be dynamic
- who knows B-trees **PINGO**

- example on the board

From Atomic Values to Strings

- strings take more time to compare
- load as few strings from disk as possible

String B-Tree [FG99]

- strings are stored in EM
- strings are identified by starting positions

- B-tree layout for sorted suffixes identified by position
- at least $b = \Theta(B)$ children
- tree height $O(\log_B N)$

- given node v
- $L(v)$ is lexicographically smallest string at v
- $R(v)$ is lexicographically largest string at v

- given node v with children $v_0, \ldots, v_k$ with $k \in [b, 2b)$
- inner: store separators $L(v_0), R(v_0), \ldots, L(v_k), R(v_k)$
- leaf: store strings and link leaves

Search in String B-Tree

- task: find all occurrences of pattern P
- two traversals of String B-Tree
- identify leftmost/rightmost occurrence
- output all strings in $O(occ/B)$

- at every node with children $v_0, \ldots, v_k$
- binary search for P in $L(v_0), \ldots, R(v_k)$
  - if $R(v_i) < P \leq L(v_{i-1})$: found
  - if $L(v_i) < P \leq R(v_i)$: continue in v_i

Lemma: String B-Tree

Using a String B-tree, a pattern P can be found in a set of strings with total length N in $O(|P|/B \log N)$ I/Os

Proof (Sketch)

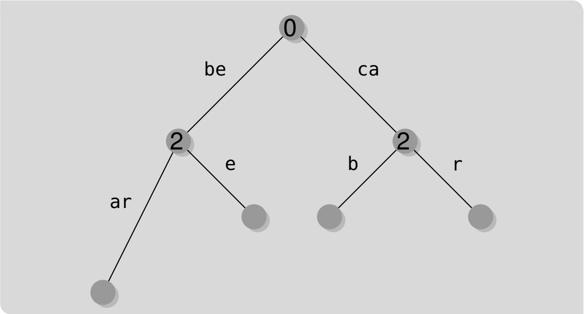
- String B-Tree has height $\log_B N$
- load separators of node: $O(1)$ I/O
- load strings for binary search: $O(|P|/B)$ I/Os
- total:
  $O(\log_B N \cdot \log B \cdot |P|/B) = O(|P|/B \log N)$ I/Os

Improving String B-Tree with Patricia Tries (1/2)

Karlsruhe Institute of Technology

Patricia Trie

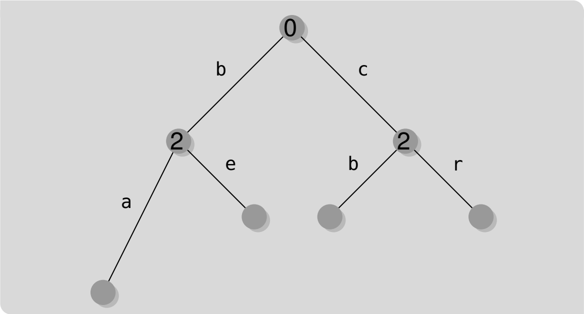
- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for k strings



Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for k strings



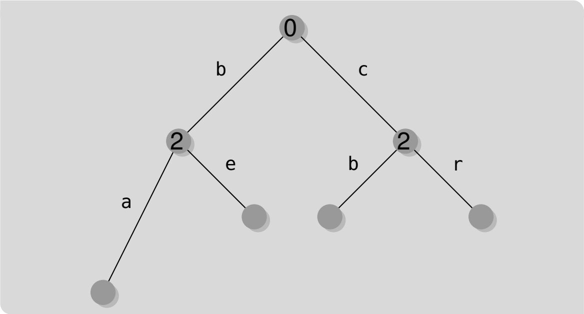
Improving String B-Tree with Patricia Tries (1/2)

Karlsruhe Institute of Technology

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for k strings

- search requires two steps
- first blind search using only trie
- blind search can result in false matches
- second a comparison with resulting string
- use any leaf after matching pattern

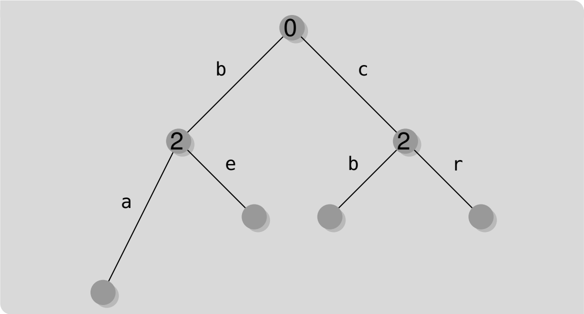


Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for k strings

- search requires two steps
- first blind search using only trie
- blind search can result in false matches
- second a comparison with resulting string
- use any leaf after matching pattern



- How do Patricia tries help? **PINGO**

Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf w
- compute $L = lcp(P, w)$
- let u be first node on root-to-w path with $d \geq L$

Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf w
- compute $L = lcp(P, w)$
- let u be first node on root-to-w path with $d \geq L$

$d = L$

- find matching children v_i and v_{i+1} of w with
- branching characters $c_i < P[L+1] < c_{i+1}$
- example on the board

Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf w
- compute $L = lcp(P, w)$
- let u be first node on root-to-w path with $d \geq L$

$d = L$

- find matching children v_i and v_{i+1} of w with
- branching characters $c_i < P[L+1] < c_{i+1}$
- example on the board

$d > L$

- consider next branching character c on path
- if $P[L+1] < c$ continue in leftmost leaf
- if $P[L+1] > c$ continue in rightmost leaf

Searching in Improved String B-Tree

Karlsruhe Institute of Technology

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for w ℹ result of blind search
- load one string and compare with P
- identify child and continue

Searching in Improved String B-Tree

- at every node with children $v_0, \dots, v_k$
- load Patricia trie for $L(v_0), \dots, R(v_k)$
- search Patricia trie for w ⓘ result of blind search
- load one string and compare with P
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern P can be found in a set of strings with total length N with $O(|P|/B \log_B N)$ I/Os

Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for w ⓘ result of blind search
- load one string and compare with P
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern P can be found in a set of strings with total length N with $O(|P|/B \log_B N)$ I/Os

Proof (Sketch)

- loading PT: $O(1)$ I/Os
- blind search: no I/Os
- loading one string: $O(|P|/B)$ I/Os
- identify child: no I/Os
- total $O(|P|/B \log_B N)$ I/Os

Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for w ℹ result of blind search
- load one string and compare with P
- identify child and continue

- How can this be improved even further?
  PINGO

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern P can be found in a set of strings with total length N with $O(|P|/B \log_B N)$ I/Os

Proof (Sketch)

- loading PT: $O(1)$ I/Os
- blind search: no I/Os
- loading one string: $O(|P|/B)$ I/Os
- identify child: no I/Os
- total $O(|P|/B \log_B N)$ I/Os

Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree $p_0, p_1, p_2, \ldots$
- in Patricia tries PT_{p_i} compute $L = lcp(P, w)$
- all strings in p_i have prefix $P[0..L)$
- do not compare previously matched characters
- load only $|P| - L$ characters at next node
- pass L down the String B-tree

Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree $p_0, p_1, p_2, \ldots$
- in Patricia tries PT_{p_i} compute $L = lcp(P, w)$
- all strings in p_i have prefix $P[0..L)$
- do not compare previously matched characters
- load only $|P| - L$ characters at next node
- pass L down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern P can be found in a set of strings with total length N in $O(|P|/B + \log_B N)$ I/Os

Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree $p_0, p_1, p_2, \ldots$
- in Patricia tries PT_{p_i} compute $L = lcp(P, w)$
- all strings in p_i have prefix $P[0..L)$
- do not compare previously matched characters
- load only $|P| - L$ characters at next node
- pass L down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern P can be found in a set of strings with total length N in $O(|P|/B + \log_B N)$ I/Os

Proof (Sketch)

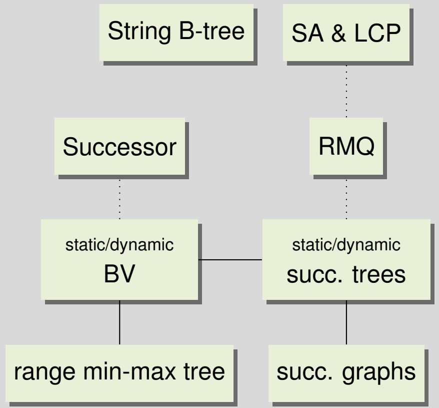
- passing down LCP-value: no I/Os
- telescoping sum $\sum_{i \leq h} \frac{L_i - L_{i-1}}{B}$
- $h = \log_B N$ height of String B-tree
- L_i is LCP-value on Level i
- $L_0 = 0$ and $L_h \leq |P|$
- total: $O(|P|/B + \log_B N)$ I/Os

Conclusion and Outlook

This Lecture

- suffix array and LCP array
- String B-tree

Advanced Data Structures



Bibliography I

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pages 1116–1127. DOI: 10.1145/48529.48535.

[Bah+19] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. "SACABench: Benchmarking Suffix Array Construction". In: *SPIRE*. Volume 11811. Lecture Notes in Computer Science. Springer, 2019, pages 407–416. DOI: 10.1007/978-3-030-32686-9_29.

[Bin18] Timo Bingmann. "Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools". PhD thesis. Karlsruhe Institute of Technology, Germany, 2018. DOI: 10.5445/IR/1000085031.

[FG99] Paolo Ferragina and Roberto Grossi. "The String B-tree: A New Data Structure for String Search in External Memory and Its Applications". In: *J. ACM* 46.2 (1999), pages 236–280. DOI: 10.1145/301970.301973.

Bibliography II

[GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. "New Indices for Text: Pat Trees and Pat Arrays". In: *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992, pages 66–82.

[Kär+17] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. "Engineering External Memory Induced Suffix Sorting". In: *ALENEX*. SIAM, 2017, pages 98–108. DOI: 10.1137/1.9781611974768.8.

[Kur20] Florian Kurpicz. "Parallel Text Index Construction". PhD thesis. Technical University of Dortmund, Germany, 2020. DOI: 10.17877/DE290R-21114.

[MM93] Udi Manber and Eugene W. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM J. Comput.* 22.5 (1993), pages 935–948. DOI: 10.1137/0222058.

[PST07] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. "A Taxonomy of Suffix Array Construction Algorithms". In: *ACM Comput. Surv.* 39.2 (2007), page 4. DOI: 10.1145/1242471.1242472.