# String indexing in the Word RAM model, part 2

Paweł Gawrychowski

University of Wrocław

We now know how to store the $\mathrm{lcp}$ array and the RMQ structure in $4n + o(n)$ bits. But we still need to store $SA$, so we need $n \log n$ bits (we might also need to store $SA^{-1}$, which is another $n \log n$ bits). Let's see how to decrease this bound!

# Compressed suffix arrays

A text of length $n$ over $\Sigma$ can be stored in $n \log |\Sigma|$ bits. Now if $\Sigma$ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

**Compressed suffix arrays**

Represent $SA$ in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\mathrm{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting $SA^{-1}$.)

**Grossi and Vitter 2000**

For any constant $\epsilon > 0$, $SA$ can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\mathrm{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

# Compressed suffix arrays

A text of length $n$ over $\Sigma$ can be stored in $n \log |\Sigma|$ bits. Now if $\Sigma$ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

## Compressed suffix arrays

Represent $SA$ in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\text{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting $SA^{-1}$.)

## Grossi and Vitter 2000

For any constant $\epsilon > 0$, $SA$ can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

# Compressed suffix arrays

A text of length $n$ over $\Sigma$ can be stored in $n \log |\Sigma|$ bits. Now if $\Sigma$ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

## Compressed suffix arrays

Represent $SA$ in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\mathrm{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting $SA^{-1}$.)

## Grossi and Vitter 2000

For any constant $\epsilon > 0$, $SA$ can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\mathrm{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

# Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

**Entropy (or zeroth order empirical entropy)**

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where $n_c$ is the number of occurrences of character $c$ in $T$.

***k*-th order empirical entropy**

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where $T_s$ is the concatenation of all characters in $T$ following an occurrence of $s$.

It is known that Lempel-Ziv compression methods approach the $k$-th order empirical entropy.

# Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

## Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where $n_c$ is the number of occurrences of character $c$ in $T$.

## $k$-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s\in\Sigma^k} |T_s| H_0(T_s)$$

where $T_s$ is the concatenation of all characters in $T$ following an occurrence of $s$.

It is known that Lempel-Ziv compression methods approach the $k$-th order empirical entropy.

# Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

## Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where $n_c$ is the number of occurrences of character $c$ in $T$.

## $k$-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where $T_s$ is the concatenation of all characters in $T$ following an occurrence of $s$.

It is known that Lempel-Ziv compression methods approach the $k$-th order empirical entropy.

# Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

## Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where $n_c$ is the number of occurrences of character $c$ in $T$.

## $k$-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where $T_s$ is the concatenation of all characters in $T$ following an occurrence of $s$.

It is known that Lempel-Ziv compression methods approach the $k$-th order empirical entropy.

# Can we do even better?

Now we would like to represent *SA* in space proportional to the *k*-th order empirical entropy of the text.

**Sadakane 2003**

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that $\mathrm{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \mathrm{polylog}(n)$.

**Grossi, Gupta, Vitter 2003**

*SA* can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits.

These bounds are painful to look at, so we will ignore them.

# Can we do even better?

Now we would like to represent *SA* in space proportional to the *k*-th order empirical entropy of the text.

### Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that $\mathsf{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \text{polylog}(n)$.

### Grossi, Gupta, Vitter 2003

*SA* can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits.

These bounds are painful to look at, so we will ignore them.

# Can we do even better?

Now we would like to represent *SA* in space proportional to the $k$-th order empirical entropy of the text.

### Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1+H_0(T))+3) + o(n)$ bits, so that $\mathsf{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \text{polylog}(n)$.

### Grossi, Gupta, Vitter 2003

*SA* can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits.

These bounds are painful to look at, so we will ignore them.

# Can we do even better?

Now we would like to represent *SA* in space proportional to the *k*-th order empirical entropy of the text.

### Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n^{\frac{1+\epsilon'}{\epsilon}} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that $\mathrm{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \mathrm{polylog}(n)$.

### Grossi, Gupta, Vitter 2003

*SA* can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits.

These bounds are painful to look at, so we will ignore them.

# Grossi and Vitter

We will assume $|\Sigma| = 2$.

*SA* can be represented in $\frac{1}{2} n \log\log n + 6n + \mathcal{O}(\frac{n}{\log\log n})$ bits, so that lookup($i$) takes $\mathcal{O}(\log\log n)$ time.

$SA_0$ is the suffix array for the original string $w = w_0$. We create a new string $w_1$ by chopping $w_0$ into blocks of two characters:

$$w[2]w[3], w[4]w[5], \ldots$$

and treating each such block as a single letter. In other words, we keep only suffixes starting at even positions. $SA_1$ is the suffix array constructed for $w_1$.

Is there any relation between $SA_0$ and $SA_1$?

In other words, assume that we can perform $\text{lookup}(i)$ on $SA_1$. Can we implement $\text{lookup}(i)$ on $SA_0$ if we add just a little bit of additional data?

$SA_0$ is the suffix array for the original string $w = w_0$. We create a new string $w_1$ by chopping $w_0$ into blocks of two characters:

$$w[2]w[3], w[4]w[5], \ldots$$

and treating each such block as a single letter. In other words, we keep only suffixes starting at even positions. $SA_1$ is the suffix array constructed for $w_1$.

Is there any relation between $SA_0$ and $SA_1$?

In other words, assume that we can perform $\mathrm{lookup}(i)$ on $SA_1$. Can we implement $\mathrm{lookup}(i)$ on $SA_0$ if we add just a little bit of additional data?

$SA_0$ is the suffix array for the original string $w = w_0$. We create a new string $w_1$ by chopping $w_0$ into blocks of two characters:

$$w[2]w[3], w[4]w[5], \ldots$$

and treating each such block as a single letter. In other words, we keep only suffixes starting at even positions. $SA_1$ is the suffix array constructed for $w_1$.

Is there any relation between $SA_0$ and $SA_1$?

In other words, assume that we can perform $\mathsf{lookup}(i)$ on $SA_1$. Can we implement $\mathsf{lookup}(i)$ on $SA_0$ if we add just a little bit of additional data?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$: | a | b | b | a | b | b | a | b | b | a | b | b | a | b | a | a | a | b | a | b | a | b | b | a | b | b | b | a | b | b | a | # |
| $SA_0$: | 15 | 16 | 31 | 13 | 17 | 19 | 28 | 10 | 7 | 4 | 1 | 21 | 24 | 32 | 14 | 30 | 12 | 18 | 27 | 9 | 6 | 3 | 20 | 23 | 29 | 11 | 26 | 8 | 5 | 2 | 22 | 25 |
| $B_0$: | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| $rank_0$: | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 14 | 14 | 15 | 16 | 16 |
| $\Psi_0$: | 2 | 2 | 14 | 15 | 18 | 23 | 7 | 8 | 28 | 10 | 30 | 31 | 13 | 14 | 15 | 16 | 17 | 18 | 7 | 8 | 21 | 10 | 23 | 13 | 16 | 17 | 27 | 28 | 21 | 30 | 31 | 27 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA_1$: | 8 | 14 | 5 | 2 | 12 | 16 | 7 | 15 | 6 | 9 | 3 | 10 | 13 | 4 | 1 | 11 |

1. If $SA_0[i]$ is even, then we return $2 \cdot SA_1[i']$, where $i'$ is the number of even suffixes in $SA_0[1..i]$.
2. If $SA_0[i]$ is odd, then we return $2 \cdot SA_1[i'] - 1$, where $i'$ is the number of even suffixes in $SA_0[1..j]$, where $SA_0[i] = SA_0[j] - 1$.

$$\Psi_0(i) = \begin{cases} i & \text{if } SA_0[i] \text{ is even} \\ j & \text{if } SA_0[i] + 1 = SA_0[j] \text{ is odd} \end{cases}$$

In both cases, augmenting $B_0$ with a rank structure reduces the problem to storing $\Psi_0$ in small space.

| | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* | *17* | *18* | *19* | *20* | *21* | *22* | *23* | *24* | *25* | *26* | *27* | *28* | *29* | *30* | *31* | *32* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$: | a | b | b | a | b | b | a | b | b | a | b | b | a | b | a | a | a | b | a | b | a | b | b | a | b | b | b | a | b | b | a | # |
| $SA_0$: | 15 | 16 | 31 | 13 | 17 | 19 | 28 | 10 | 7 | 4 | 1 | 21 | 24 | 32 | 14 | 30 | 12 | 18 | 27 | 9 | 6 | 3 | 20 | 23 | 29 | 11 | 26 | 8 | 5 | 2 | 22 | 25 |
| $B_0$: | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| $rank_0$: | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 14 | 14 | 15 | 16 | 16 |
| $\Psi_0$: | 2 | 2 | 14 | 15 | 18 | 23 | 7 | 8 | 28 | 10 | 30 | 31 | 13 | 14 | 15 | 16 | 17 | 18 | 7 | 8 | 21 | 10 | 23 | 13 | 16 | 17 | 27 | 28 | 21 | 30 | 31 | 27 |

| | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA_1$: | 8 | 14 | 5 | 2 | 12 | 16 | 7 | 15 | 6 | 9 | 3 | 10 | 13 | 4 | 1 | 11 |

1. If $SA_0[i]$ is even, then we return $2 \cdot SA_1[i']$, where $i'$ is the number of even suffixes in $SA_0[1..i]$.
2. If $SA_0[i]$ is odd, then we return $2 \cdot SA_1[i'] - 1$, where $i'$ is the number of even suffixes in $SA_0[1..j]$, where $SA_0[i] = SA_0[j] - 1$.

$$\Psi_0(i) = \begin{cases} i & \text{if } SA_0[i] \text{ is even} \\ j & \text{if } SA_0[i] + 1 = SA_0[j] \text{ is odd} \end{cases}$$

In both cases, augmenting $B_0$ with a rank structure reduces the problem to storing $\Psi_0$ in small space.

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$: | a | b | b | a | b | b | a | b | b | a | b | b | a | b | a | a | a | b | a | b | a | b | b | a | b | b | b | a | b | b | a | # |
| $SA_0$: | 15 | 16 | 31 | 13 | 17 | 19 | 28 | 10 | 7 | 4 | 1 | 21 | 24 | 32 | 14 | 30 | 12 | 18 | 27 | 9 | 6 | 3 | 20 | 23 | 29 | 11 | 26 | 8 | 5 | 2 | 22 | 25 |
| $B_0$: | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| $rank_0$: | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 14 | 14 | 15 | 16 | 16 |
| $\Psi_0$: | 2 | 2 | 14 | 15 | 18 | 23 | 7 | 8 | 28 | 10 | 30 | 31 | 13 | 14 | 15 | 16 | 17 | 18 | 7 | 8 | 21 | 10 | 23 | 13 | 16 | 17 | 27 | 28 | 21 | 30 | 31 | 27 |

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA_1$: | 8 | 14 | 5 | 2 | 12 | 16 | 7 | 15 | 6 | 9 | 3 | 10 | 13 | 4 | 1 | 11 |

1. If $SA_0[i]$ is even, then we return $2 \cdot SA_1[i']$, where $i'$ is the number of even suffixes in $SA_0[1..i]$.
2. If $SA_0[i]$ is odd, then we return $2 \cdot SA_1[i'] - 1$, where $i'$ is the number of even suffixes in $SA_0[1..j]$, where $SA_0[i] = SA_0[j] - 1$.

$$\Psi_0(i) = \begin{cases} i & \text{if } SA_0[i] \text{ is even} \\ j & \text{if } SA_0[i] + 1 = SA_0[j] \text{ is odd} \end{cases}$$

In both cases, augmenting $B_0$ with a rank structure reduces the problem to storing $\Psi_0$ in small space.

| | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* | *17* | *18* | *19* | *20* | *21* | *22* | *23* | *24* | *25* | *26* | *27* | *28* | *29* | *30* | *31* | *32* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$: | a | b | b | a | b | b | a | b | b | a | b | b | a | b | a | a | a | b | a | b | a | b | b | a | b | b | b | a | b | b | a | # |
| $SA_0$: | 15 | 16 | 31 | 13 | 17 | 19 | 28 | 10 | 7 | 4 | 1 | 21 | 24 | 32 | 14 | 30 | 12 | 18 | 27 | 9 | 6 | 3 | 20 | 23 | 29 | 11 | 26 | 8 | 5 | 2 | 22 | 25 |
| $B_0$: | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| $rank_0$: | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 14 | 14 | 15 | 16 | 16 |
| $\Psi_0$: | 2 | 2 | 14 | 15 | 18 | 23 | 7 | 8 | 28 | 10 | 30 | 31 | 13 | 14 | 15 | 16 | 17 | 18 | 7 | 8 | 21 | 10 | 23 | 13 | 16 | 17 | 27 | 28 | 21 | 30 | 31 | 27 |

| | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA_1$: | 8 | 14 | 5 | 2 | 12 | 16 | 7 | 15 | 6 | 9 | 3 | 10 | 13 | 4 | 1 | 11 |

1. If $SA_0[i]$ is even, then we return $2 \cdot SA_1[i']$, where $i'$ is the number of even suffixes in $SA_0[1..i]$.
2. If $SA_0[i]$ is odd, then we return $2 \cdot SA_1[i'] - 1$, where $i'$ is the number of even suffixes in $SA_0[1..j]$, where $SA_0[i] = SA_0[j] - 1$.

$$\Psi_0(i) = \begin{cases} i & \text{if } SA_0[i] \text{ is even} \\ j & \text{if } SA_0[i] + 1 = SA_0[j] \text{ is odd} \end{cases}$$

In both cases, augmenting $B_0$ with a rank structure reduces the problem to storing $\Psi_0$ in small space.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$: | a | b | b | a | b | b | a | b | b | a | b | b | a | b | a | a | a | b | a | b | a | b | b | a | b | b | b | a | b | b | a | # |
| $SA_0$: | 15 | 16 | 31 | 13 | 17 | 19 | 28 | 10 | 7 | 4 | 1 | 21 | 24 | 32 | 14 | 30 | 12 | 18 | 27 | 9 | 6 | 3 | 20 | 23 | 29 | 11 | 26 | 8 | 5 | 2 | 22 | 25 |
| $B_0$: | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| $rank_0$: | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 14 | 14 | 15 | 16 | 16 |
| $\Psi_0$: | 2 | 2 | 14 | 15 | 18 | 23 | 7 | 8 | 28 | 10 | 30 | 31 | 13 | 14 | 15 | 16 | 17 | 18 | 7 | 8 | 21 | 10 | 23 | 13 | 16 | 17 | 27 | 28 | 21 | 30 | 31 | 27 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA_1$: | 8 | 14 | 5 | 2 | 12 | 16 | 7 | 15 | 6 | 9 | 3 | 10 | 13 | 4 | 1 | 11 |

1. If $SA_0[i]$ is even, then we return $2 \cdot SA_1[i']$, where $i'$ is the number of even suffixes in $SA_0[1..i]$.
2. If $SA_0[i]$ is odd, then we return $2 \cdot SA_1[i'] - 1$, where $i'$ is the number of even suffixes in $SA_0[1..j]$, where $SA_0[i] = SA_0[j] - 1$.

$$\Psi_0(i) = \begin{cases} i & \text{if } SA_0[i] \text{ is even} \\ j & \text{if } SA_0[i] + 1 = SA_0[j] \text{ is odd} \end{cases}$$

In both cases, augmenting $B_0$ with a rank structure reduces the problem to storing $\Psi_0$ in small space.

# Storing $\Psi_0$

$\Psi_0[i]$ is the position of the even successor of $SA_0[i]$ in the suffix array.

We need to compress all $\Psi_0[i]$ corresponding to odd suffixes. But the values don't seem to have any special structure...

Or do they? Let's look at $\Psi_0[i]$ such that $B_0[i] = 0$ and $T[SA[i]] = \mathtt{a}$. The indices are:

$$1, 3, 4, 5, 6, 9, 11, 12$$

and the values are:

$$2, 14, 15, 18, 23, 28, 30, 31$$

So, all $\Psi_0[i]$ such that $B_0[i] = 0$ can be decomposed into two increasing lists. If the alphabet is larger, we just have more lists!

# Storing $\Psi_0$

$\Psi_0[i]$ is the position of the even successor of $SA_0[i]$ in the suffix array.

We need to compress all $\Psi_0[i]$ corresponding to odd suffixes. But the values don't seem to have any special structure...

Or do they? Let's look at $\Psi_0[i]$ such that $B_0[i] = 0$ and $T[SA[i]] = \mathtt{a}$. The indices are:

$$1, 3, 4, 5, 6, 9, 11, 12$$

and the values are:

$$2, 14, 15, 18, 23, 28, 30, 31$$

So, all $\Psi_0[i]$ such that $B_0[i] = 0$ can be decomposed into two increasing lists. If the alphabet is larger, we just have more lists!

# Storing $\Psi_0$

$\Psi_0[i]$ is the position of the even successor of $SA_0[i]$ in the suffix array.

We need to compress all $\Psi_0[i]$ corresponding to odd suffixes. But the values don't seem to have any special structure...

Or do they? Let's look at $\Psi_0[i]$ such that $B_0[i] = 0$ and $T[SA[i]] = \mathtt{a}$. The indices are:

$$1, 3, 4, 5, 6, 9, 11, 12$$

and the values are:

$$2, 14, 15, 18, 23, 28, 30, 31$$

So, all $\Psi_0[i]$ such that $B_0[i] = 0$ can be decomposed into two increasing lists. If the alphabet is larger, we just have more lists!

# Storing $\Psi_0$

We generate a list of pairs $(T[SA_0[i]], \Psi_0[i])$ for all $i$ such that $B_0[i] = 0$.

To store all $\Psi_0[i]$ in small space, it is enough to show how to store an increasing list of numbers. This sounds easier, as storing an increasing list is easier than storing an arbitrary list!

**Recursion**

We will recurse on $SA_0$, $SA_1$, $SA_2$, $SA_3$, .... In $SA_k$, our alphabet is of size $2^{2^k}$, because we are operating on blocks of $2^k$ characters from the original text. So storing $\Psi_k$ reduces to storing an increasing list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits, where $n_k = \frac{n}{2^k}$.

# Storing $\Psi_0$

We generate a list of pairs $(T[SA_0[i]], \Psi_0[i])$ for all $i$ such that $B_0[i] = 0$.

To store all $\Psi_0[i]$ in small space, it is enough to show how to store an increasing list of numbers. This sounds easier, as storing an increasing list is easier than storing an arbitrary list!

## Recursion

We will recurse on $SA_0$, $SA_1$, $SA_2$, $SA_3$, .... In $SA_k$, our alphabet is of size $2^{2^k}$, because we are operating on blocks of $2^k$ characters from the original text. So storing $\Psi_k$ reduces to storing an increasing list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits, where $n_k = \frac{n}{2^k}$.

## Lemma

A list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits can be stored in $\frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$ bits of space.

We split every number into a prefix of length $\log n_k$ and the rest:

1. The suffixes are stored naively, taking $2^k$ bits each, so $2^k\frac{n_k}{2} = \frac{n}{2}$ in total.
2. The prefixes are nondecreasing, so we store their differences. The differences are encoded in unary (as in the $\mathrm{lcp}$ representation), taking $n_k + \frac{1}{2}n_k = \frac{3}{2}n_k$ bits in total.

We augment the representation of the prefixes with a rank/select structure, so that we can extract any prefix in $\mathcal{O}(1)$ time. This adds $\mathcal{O}(\frac{n_k}{\log\log n_k})$ bits.

## Lemma

A list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits can be stored in $\frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$ bits of space.

We split every number into a prefix of length $\log n_k$ and the rest:

1. The suffixes are stored naively, taking $2^k$ bits each, so $2^k \frac{n_k}{2} = \frac{n}{2}$ in total.
2. The prefixes are nondecreasing, so we store their differences. The differences are encoded in unary (as in the lcp representation), taking $n_k + \frac{1}{2}n_k = \frac{3}{2}n_k$ bits in total.

We augment the representation of the prefixes with a rank/select structure, so that we can extract any prefix in $\mathcal{O}(1)$ time. This adds $\mathcal{O}(\frac{n_k}{\log\log n_k})$ bits.

## Lemma

A list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits can be stored in $\frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$ bits of space.

We split every number into a prefix of length $\log n_k$ and the rest:

1. The suffixes are stored naively, taking $2^k$ bits each, so $2^k \frac{n_k}{2} = \frac{n}{2}$ in total.
2. The prefixes are nondecreasing, so we store their differences. The differences are encoded in unary (as in the $\mathrm{lcp}$ representation), taking $n_k + \frac{1}{2}n_k = \frac{3}{2}n_k$ bits in total.

We augment the representation of the prefixes with a rank/select structure, so that we can extract any prefix in $\mathcal{O}(1)$ time. This adds $\mathcal{O}(\frac{n_k}{\log\log n_k})$ bits.

## Lemma

A list of $\frac{n_k}{2}$ numbers consisting of $2^k + \log n_k$ bits can be stored in $\frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$ bits of space.

We split every number into a prefix of length $\log n_k$ and the rest:

1. The suffixes are stored naively, taking $2^k$ bits each, so $2^k \frac{n_k}{2} = \frac{n}{2}$ in total.
2. The prefixes are nondecreasing, so we store their differences. The differences are encoded in unary (as in the $\mathrm{lcp}$ representation), taking $n_k + \frac{1}{2}n_k = \frac{3}{2}n_k$ bits in total.

We augment the representation of the prefixes with a rank/select structure, so that we can extract any prefix in $\mathcal{O}(1)$ time. This adds $\mathcal{O}(\frac{n_k}{\log\log n_k})$ bits.

# Final space bound

We use such encoding at every level. When $n_k \leq \frac{n}{\log n}$ we terminate and switch to the naive representation, so there are $\log\log n$ levels.

Then the total space (in bits) for storing all $\Psi_k$ is:

$$\frac{n}{\log n}\log n + \sum_{i=0}^{\log\log n} \frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$$

and the query time is $\mathcal{O}(\log\log n)$.

Together with all $B_k$, this gives us a bound of

$$\frac{1}{2}n\log\log n + 6n + \mathcal{O}(\frac{n}{\log\log n}).$$

# Final space bound

We use such encoding at every level. When $n_k \leq \frac{n}{\log n}$ we terminate and switch to the naive representation, so there are $\log \log n$ levels.

Then the total space (in bits) for storing all $\Psi_k$ is:

$$\frac{n}{\log n} \log n + \sum_{i=0}^{\log \log n} \frac{1}{2} n + \frac{3}{2} n_k + \mathcal{O}(\frac{n_k}{\log \log n_k})$$

and the query time is $\mathcal{O}(\log \log n)$.

Together with all $B_k$, this gives us a bound of

$$\frac{1}{2} n \log \log n + 6n + \mathcal{O}(\frac{n}{\log \log n}).$$

# Final space bound

We use such encoding at every level. When $n_k \leq \frac{n}{\log n}$ we terminate and switch to the naive representation, so there are $\log \log n$ levels.

Then the total space (in bits) for storing all $\Psi_k$ is:

$$\frac{n}{\log n} \log n + \sum_{i=0}^{\log \log n} \frac{1}{2} n + \frac{3}{2} n_k + \mathcal{O}(\frac{n_k}{\log \log n_k})$$

and the query time is $\mathcal{O}(\log \log n)$.

Together with all $B_k$, this gives us a bound of

$$\frac{1}{2} n \log \log n + 6n + \mathcal{O}(\frac{n}{\log \log n}).$$

# Final space bound

We use such encoding at every level. When $n_k \leq \frac{n}{\log n}$ we terminate and switch to the naive representation, so there are $\log\log n$ levels.

Then the total space (in bits) for storing all $\Psi_k$ is:

$$\frac{n}{\log n}\log n + \sum_{i=0}^{\log\log n} \frac{1}{2}n + \frac{3}{2}n_k + \mathcal{O}(\frac{n_k}{\log\log n_k})$$

and the query time is $\mathcal{O}(\log\log n)$.

Together with all $B_k$, this gives us a bound of

$$\frac{1}{2}n\log\log n + 6n + \mathcal{O}(\frac{n}{\log\log n}).$$

# Grossi and Vitter

We will again assume $|\Sigma| = 2$.

For any constant $\epsilon > 0$, *SA* can be represented using just $(1 + \frac{1}{\epsilon})n + o(n)$ bits, so that $\mathsf{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

We will build on the previous solution. Instead of storing $\log \log n$ levels, we will (for $\epsilon = 1/2$) store levels 0, $\ell' = 1/2 \log \log n$ and $\ell = \log \log n$.

Thus, we consider $SA_0$, $SA_{\ell'}$ and $SA_{\ell}$. We need a mechanism to determine if a given index in $SA_0$ corresponds to an index in $SA_{\ell'}$ (and similarly for $SA_{\ell'}$ and $SA_{\ell}$).

We will build on the previous solution. Instead of storing $\log\log n$ levels, we will (for $\epsilon = 1/2$) store levels 0, $\ell' = 1/2 \log\log n$ and $\ell = \log\log n$.

Thus, we consider $SA_0$, $SA_{\ell'}$ and $SA_{\ell}$. We need a mechanism to determine if a given index in $SA_0$ corresponds to an index in $SA_{\ell'}$ (and similarly for $SA_{\ell'}$ and $SA_{\ell}$).

# Static dictionary

Given a set $S \subseteq [U]$, we want to construct a structure for membership queries of the form “does $x \in S$?”. Ideally, the structure should also provide rank queries. We need constant query time!

**Pagh 2002**

Let $B = \log \binom{U}{n}$. Then, there is a static dictionary using

$$B + \mathcal{O}(\log\log|U|) + o(n)$$

bits of space with constant query time. For $U = n \operatorname{polylog} n$ the structure also provides rank queries (in constant time).

In fact, the dense case is enough here, we will see a simple implementation on the problemset.

# Static dictionary

Given a set $S \subseteq [U]$, we want to construct a structure for membership queries of the form "does $x \in S$?". Ideally, the structure should also provide rank queries. We need constant query time!

## Pagh 2002

Let $B = \log \binom{U}{n}$. Then, there is a static dictionary using

$$B + \mathcal{O}(\log\log|U|) + o(n)$$

bits of space with constant query time. For $U = n\,\mathrm{polylog}\,n$ the structure also provides rank queries (in constant time).

In fact, the dense case is enough here, we will see a simple implementation on the problemset.

# Static dictionary

Given a set $S \subseteq [U]$, we want to construct a structure for membership queries of the form "does $x \in S$?". Ideally, the structure should also provide rank queries. We need constant query time!

## Pagh 2002

Let $B = \log \binom{U}{n}$. Then, there is a static dictionary using

$$B + \mathcal{O}(\log\log|U|) + o(n)$$

bits of space with constant query time. For $U = n \operatorname{polylog} n$ the structure also provides rank queries (in constant time).

In fact, the dense case is enough here, we will see a simple implementation on the problemset.

We store indices of $SA_0$ correspond to an index in $SA_{\ell'}$ (and similarly for $SA_{\ell'}$ and $SA_{\ell}$) in static dictionaries with rank queries. We denote the respective structures by $D_0$ and $D_{\ell'}$.

We also store the function $\Phi_k$:

$$\Phi_k(i) = \begin{cases} j & \text{if } SA_k[i] \neq n_k \text{ and } SA_k[j] = SA_k[i] + 1 \\ 1 & \text{otherwise} \end{cases}$$

$\Psi_k$ was “half” of $\Phi_k$, the other “half” behaves similarly.

Note that now we don’t need the bitvector $B_k$.

We store indices of $SA_0$ correspond to an index in $SA_{\ell'}$ (and similarly for $SA_{\ell'}$ and $SA_\ell$) in static dictionaries with rank queries. We denote the respective structures by $D_0$ and $D_{\ell'}$.

We also store the function $\Phi_k$:

$$\Phi_k(i) = \begin{cases} j & \text{if } SA_k[i] \neq n_k \text{ and } SA_k[j] = SA_k[i] + 1 \\ 1 & \text{otherwise} \end{cases}$$

$\Psi_k$ was “half” of $\Phi_k$, the other “half” behaves similarly.

Note that now we don’t need the bitvector $B_k$.

We store indices of $SA_0$ correspond to an index in $SA_{\ell'}$ (and similarly for $SA_{\ell'}$ and $SA_{\ell}$) in static dictionaries with rank queries. We denote the respective structures by $D_0$ and $D_{\ell'}$.

We also store the function $\Phi_k$:

$$\Phi_k(i) = \begin{cases} j & \text{if } SA_k[i] \neq n_k \text{ and } SA_k[j] = SA_k[i] + 1 \\ 1 & \text{otherwise} \end{cases}$$

$\Psi_k$ was “half” of $\Phi_k$, the other “half” behaves similarly.

Note that now we don’t need the bitvector $B_k$.

How to store $\Psi_k$? Similarly to the list $L_k$, we can define a list $L'_k$ for the other "half" of $\Psi_k$, and concatenate both lists.

**Lemma**

For $k = 0$, the concatenated lists can be stored in $n + \mathcal{O}(n/\log\log n)$ bits. For $k > 0$, they can be stored in $n + n/2^{k-1} + \mathcal{O}(n/2^k \log\log n)$.

For $k > 0$, this is the same as earlier. For $k = 0$, we store a single bitvector (treating $\#$ as a 0) with a select structure.

How to store $\Psi_k$? Similarly to the list $L_k$, we can define a list $L'_k$ for the other "half" of $\Psi_k$, and concatenate both lists.

### Lemma

For $k = 0$, the concatenated lists can be stored in $n + \mathcal{O}(n/\log\log n)$ bits. For $k > 0$, they can be stored in $n + n/2^{k-1} + \mathcal{O}(n/2^k \log\log n)$.

For $k > 0$, this is the same as earlier. For $k = 0$, we store a single bitvector (treating $\#$ as a 0) with a $\mathsf{select}$ structure.

Now assume that we can to access $SA[i] = SA_0[i]$. We use $\Psi_0$ to walk along indices $i', i'', \ldots$ until we reach an index stored in $D_0$. Let $s$ be the number of steps and $r$ the rank of the found index in $D_0$.

We switch to level $\ell'$ and proceed similarly. Let $s'$ be the number of steps and $r'$ the rank of the found index in $D_{\ell'}$.

We return $SA_l[r'] + s' \cdot 2^{\ell'} + s \cdot 2^0$. The total number of steps is $2\sqrt{\log n}$.

Now assume that we can to access $SA[i] = SA_0[i]$. We use $\Psi_0$ to walk along indices $i', i'', \ldots$ until we reach an index stored in $D_0$. Let $s$ be the number of steps and $r$ the rank of the found index in $D_0$.

We switch to level $\ell'$ and proceed similarly. Let $s'$ be the number of steps and $r'$ the rank of the found index in $D_{\ell'}$.

We return $SA_l[r'] + s' \cdot 2^{\ell'} + s \cdot 2^0$. The total number of steps is $2\sqrt{\log n}$.

Now assume that we can to access $SA[i] = SA_0[i]$. We use $\Psi_0$ to walk along indices $i', i'', \ldots$ until we reach an index stored in $D_0$. Let $s$ be the number of steps and $r$ the rank of the found index in $D_0$.

We switch to level $\ell'$ and proceed similarly. Let $s'$ be the number of steps and $r'$ the rank of the found index in $D_{\ell'}$.

We return $SA_l[r'] + s' \cdot 2^{\ell'} + s \cdot 2^0$. The total number of steps is $2\sqrt{\log n}$.

This easily generalises to $\epsilon^{-1} + 1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned} &\tfrac{n \log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1 + \tfrac{1}{2^{k-1}} + \mathcal{O}(\tfrac{1}{2^k \log\log n}) \\ &= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n) \\ &= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) \end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell) + o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}\tfrac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1+\tfrac{1}{2^{k-1}} + \mathcal{O}(\tfrac{1}{2^{k}\log\log n})\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n)\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell) + o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}\frac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1+\frac{1}{2^{k-1}}+\mathcal{O}(\frac{1}{2^{k}\log\log n})\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n)\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell)+o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}\frac{n\log n}{2^{\ell}}+n+\mathcal{O}(n/\log\log n)&+\sum\nolimits_{k=i\epsilon\ell,i\geq 1} n(1+\tfrac{1}{2^{k-1}}+\mathcal{O}(\tfrac{1}{2^{k}\log\log n}))\\ &=(1+\epsilon^{-1})n+\mathcal{O}(n/\log\log n)+\mathcal{O}(n/\log^{\epsilon} n)\\ &=(1+\epsilon^{-1})n+\mathcal{O}(n/\log\log n)\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell)+o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\frac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \sum_{k=i\epsilon\ell, i\geq 1} n(1+\frac{1}{2^{k-1}} + \mathcal{O}(\frac{1}{2^{k}\log\log n})$$
$$= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n)$$
$$= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell)+o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}
&\tfrac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1+\tfrac{1}{2^{k-1}}+\mathcal{O}(\tfrac{1}{2^{k}\log\log n})) \\
&= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n) \\
&= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)
\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell)+o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}
&\tfrac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1+\tfrac{1}{2^{k-1}} + \mathcal{O}(\tfrac{1}{2^{k}\log\log n})\\
&= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n)\\
&= (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)
\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell)+o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

This easily generalises to $\epsilon^{-1}+1$ levels instead of 2, with the total number of steps becoming $\mathcal{O}(\log^{\epsilon} n)$.

Now we analyse the total space in bits:

$$\begin{aligned}\frac{n\log n}{2^{\ell}} + n + \mathcal{O}(n/\log\log n) + \textstyle\sum_{k=i\epsilon\ell, i\geq 1} n(1+\frac{1}{2^{k-1}} + \mathcal{O}(\frac{1}{2^{k}\log\log n})\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n) + \mathcal{O}(n/\log^{\epsilon} n)\\ = (1+\epsilon^{-1})n + \mathcal{O}(n/\log\log n)\end{aligned}$$

The largest dictionary takes only $\mathcal{O}(n_{\epsilon\ell}\epsilon\ell) + o(n)$ bits of space, so this is subsumed by $\mathcal{O}(n/\log\log n)$.

Time for a pattern matching query is $\mathcal{O}(m \log^{\epsilon} n + \log n)$, disappointing.

This can be improved to e.g. $\mathcal{O}(m / \log n + \log^{\epsilon} n)$ in $(\mathcal{O}(1) + \epsilon^{-1})n$ bits of space using a hierarchy of compacted tries.

Time for a pattern matching query is $\mathcal{O}(m \log^{\epsilon} n + \log n)$, disappointing.

This can be improved to e.g. $\mathcal{O}(m / \log n + \log^{\epsilon} n)$ in $(\mathcal{O}(1) + \epsilon^{-1})n$ bits of space using a hierarchy of compacted tries.

So, we have seen suffix arrays (and compressed suffix arrays). The annoying thing about suffix arrays is that we pay some additional penalty of $\log n$ (or even more) for every query. Is this necessary?

NO!

We can use suffix trees.

So, we have seen suffix arrays (and compressed suffix arrays). The annoying thing about suffix arrays is that we pay some additional penalty of $\log n$ (or even more) for every query. Is this necessary?

NO!

We can use suffix trees.

So, we have seen suffix arrays (and compressed suffix arrays). The annoying thing about suffix arrays is that we pay some additional penalty of $\log n$ (or even more) for every query. Is this necessary?

NO!

We can use suffix trees.

## Suffix tree $ST(w[1..n])$

We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]\$$ in a compacted trie.

Take a *banana*. The suffixes are \$, *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

## Suffix tree $ST(w[1..n])$

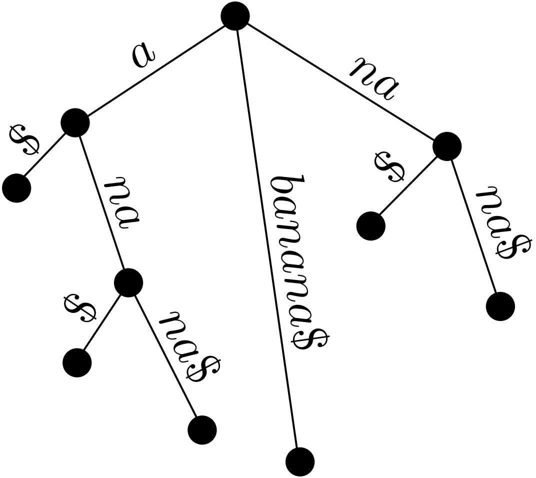
We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]\$$ in a compacted trie.

Take a *banana*. The suffixes are \$, *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

# Suffix tree $ST(w[1..n])$

We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]\$$ in a compacted trie.

Take a *banana*. The suffixes are \$, *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

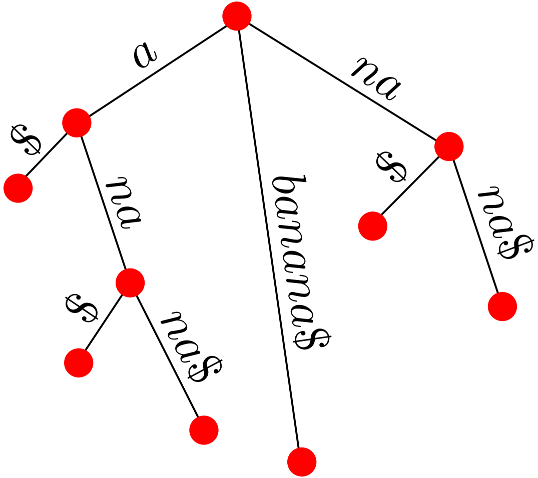


## Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.

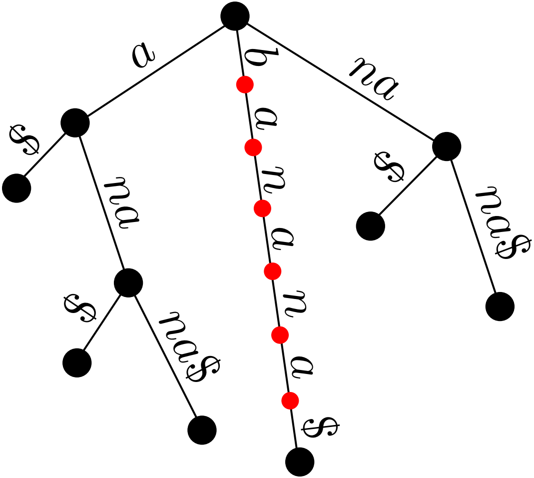
## Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.



## Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.



So, a suffix tree allows us to index the input word.

**Text indexing**

Given a word $w[1..n]$, construct a small structure allowing to answer queries of the form “where does $p[1..m]$ occur in $w[1..n]$?”.

We keep only the explicit nodes, there are $n$ of them. The labels of the edges are not kept explicitly, we just remember where do they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and a query can be answered in $\mathcal{O}(m + occ)$ time.

So, a suffix tree allows us to index the input word.

**Text indexing**

Given a word $w[1..n]$, construct a small structure allowing to answer queries of the form “where does $p[1..m]$ occur in $w[1..n]$?”.

We keep only the explicit nodes, there are $n$ of them. The labels of the edges are not kept explicitly, we just remember where do they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and a query can be answered in $\mathcal{O}(m + occ)$ time.

So, a suffix tree allows us to index the input word.

## Text indexing

Given a word $w[1..n]$, construct a small structure allowing to answer queries of the form “where does $p[1..m]$ occur in $w[1..n]$?”.

We keep only the explicit nodes, there are $n$ of them. The labels of the edges are not kept explicitly, we just remember where do they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and a query can be answered in $\mathcal{O}(m + occ)$ time.

So, a suffix tree allows us to index the input word.

**Text indexing**

Given a word $w[1..n]$, construct a small structure allowing to answer queries of the form “where does $p[1..m]$ occur in $w[1..n]$?”.

We keep only the explicit nodes, there are $n$ of them. The labels of the edges are not kept explicitly, we just remember where do they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and a query can be answered in $\mathcal{O}(m + occ)$ time.

We consider a fundamental data structure question: how to represent a tree?

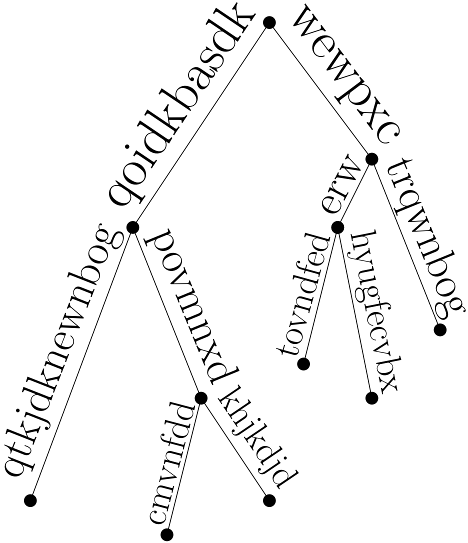
### (Compacted) Trie

A trie is simply a tree with edges labeled by single characters. A compacted trie is created by replacing maximal chains of unary vertices with single edges labeled by (possibly long) words.

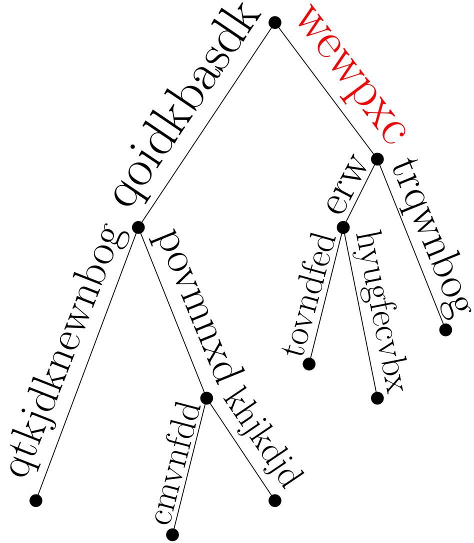
### Navigation queries

Given a pattern *p*, we want to traverse the edges of a compacted trie to find the node corresponding to *p*. If there is no such node, we would like to compute its longest prefix for which the corresponding node does exist.

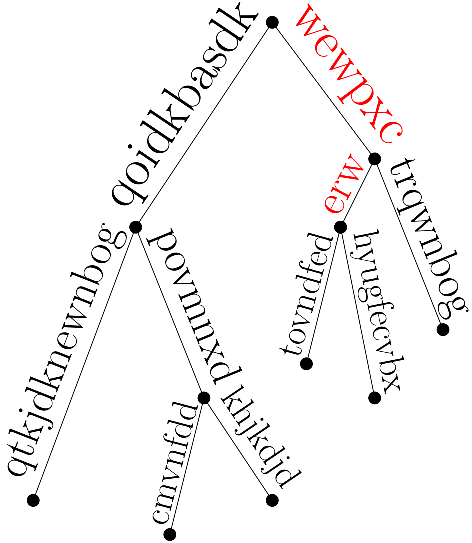
Consider $p = wewpxcwrehyzrt$ and the following compacted trie.



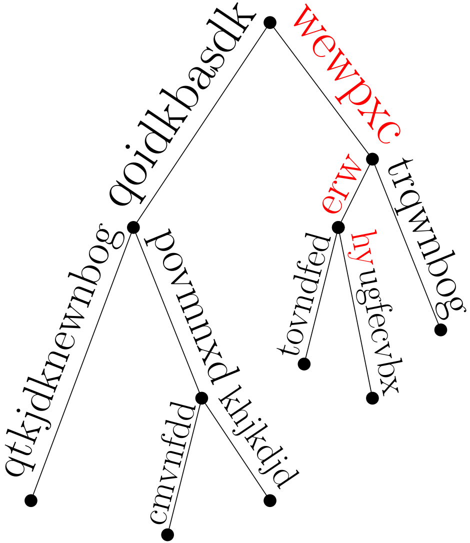
Consider $p = \textcolor{red}{wewpxc}wrehyzrt$ and the following compacted trie.



Consider $p = \textcolor{red}{wewpxcwre}hyzrt$ and the following compacted trie.



Consider $p = \textcolor{red}{wewpxcwrehy}zrt$ and the following compacted trie.



## Static case

Given a compacted trie, can we **quickly** construct a **small** structure which allows us to execute navigation queries **efficiently**?

There are clearly three parameters: the number of nodes in the compacted trie $n$, the size of the alphabet $\sigma$, and the length of the pattern $m$. We aim to achieve good bounds in terms of those $n, \sigma, m$.

**Static case**

Given a compacted trie, can we **quickly** construct a **small** structure which allows us to execute navigation queries **efficiently**?

There are clearly three parameters: the number of nodes in the compacted trie $n$, the size of the alphabet $\sigma$, and the length of the pattern $m$. We aim to achieve good bounds in terms of those $n, \sigma, m$.

So, what would be your first idea?

### Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

### Table

Or, for each node store a table of size $\sigma$ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

### BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

To make life interesting, the rules of the game are as follows:

1. the solution must be deterministic,
2. the space usage must be linear in $n$, irrespectively of $\sigma$,

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of $\sigma$, for instance $f(\sigma) = \log\sigma$, or something better if we use a more sophisticated predecessor structure. (Maybe) Surprisingly, this is not true.

**Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06**

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\sigma)$ time, which can be constructed in linear time.

To make life interesting, the rules of the game are as follows:

1. the solution must be deterministic,
2. the space usage must be linear in $n$, irrespectively of $\sigma$,

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of $\sigma$, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. (Maybe) Surprisingly, this is not true.

**Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06**

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be constructed in linear time.

To make life interesting, the rules of the game are as follows:

1. the solution must be deterministic,
2. the space usage must be linear in $n$, irrespectively of $\sigma$,

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of $\sigma$, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. (Maybe) Surprisingly, this is not true.

**Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06**

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be constructed in linear time.

To make life interesting, the rules of the game are as follows:

1. the solution must be deterministic,
2. the space usage must be linear in $n$, irrespectively of $\sigma$,

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of $\sigma$, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. (Maybe) Surprisingly, this is not true.

**Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06**

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be constructed in linear time.

To make life interesting, the rules of the game are as follows:

1. the solution must be deterministic,
2. the space usage must be linear in $n$, irrespectively of $\sigma$,

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of $\sigma$, for instance $f(\sigma) = \log\sigma$, or something better if we use a more sophisticated predecessor structure. (Maybe) Surprisingly, this is not true.

### Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\sigma)$ time, which can be constructed in linear time.

The natural question is if the $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds are the best possible. The answer is... no, they are not.

### Andersson and Thorup (even in the dynamic setting)

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time.

### Are these bounds are the best possible?

Under some assumptions, yes. More specifically, they are the best possible if $\sigma$ is unbounded in terms of $n$, and we are interested in stronger version of the navigation queries, which actually gives us the predecessor of the string we are searching for.

The natural question is if the $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds are the best possible. The answer is... no, they are not.

### Andersson and Thorup (even in the dynamic setting)

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time.

### Are these bounds are the best possible?

Under some assumptions, yes. More specifically, they are the best possible if $\sigma$ is unbounded in terms of $n$, and we are interested in stronger version of the navigation queries, which actually gives us the predecessor of the string we are searching for.

The natural question is if the $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds are the best possible. The answer is... no, they are not.

### Andersson and Thorup (even in the dynamic setting)

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time.

### Are these bounds are the best possible?

Under some assumptions, yes. More specifically, they are the best possible if $\sigma$ is unbounded in terms of $n$, and we are interested in stronger version of the navigation queries, which actually gives us the predecessor of the string we are searching for.

But it seems reasonable to consider the scenario where $\sigma$ is non-constant, yet (significantly) smaller than $n$. Hence we get the following question: what are the best possible time bounds in terms of $\sigma$?

**Gawrychowski and Fischer (very simple)**

There exists a static deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\log\sigma)$ time, which can be constructed in linear time.

Let us first see the folklore solution with $\mathcal{O}(m + \log n)$ query time that uses weight-balanced BSTs.

**Weight-balanced BST**

Given an ordered collection of $n$ items, the $i$-th item having weight $w_i$ and $\sum_i w_i = W$, we can arrange them in a BST such that the depth of the $i$-th item is $\mathcal{O}(1 + \log(W/w_i))$.

See the problemset.

Now the solution is to simply store the outgoing edges (at each node) in weight-balanced BSTs, with weights being the sizes of the substrees.

Do you see why this gives $\mathcal{O}(m + \log n)$ query time?

Let us first see the folklore solution with $\mathcal{O}(m + \log n)$ query time that uses weight-balanced BSTs.

## Weight-balanced BST

Given an ordered collection of $n$ items, the $i$-th item having weight $w_i$ and $\sum_i w_i = W$, we can arrange them in a BST such that the depth of the $i$-th item is $\mathcal{O}(1 + \log(W/w_i))$.

See the problemset.

Now the solution is to simply store the outgoing edges (at each node) in weight-balanced BSTs, with weights being the sizes of the substrees.

Do you see why this gives $\mathcal{O}(m + \log n)$ query time?

Let us first see the folklore solution with $\mathcal{O}(m + \log n)$ query time that uses weight-balanced BSTs.

## Weight-balanced BST

Given an ordered collection of $n$ items, the $i$-th item having weight $w_i$ and $\sum_i w_i = W$, we can arrange them in a BST such that the depth of the $i$-th item is $\mathcal{O}(1 + \log(W/w_i))$.

See the problemset.

Now the solution is to simply store the outgoing edges (at each node) in weight-balanced BSTs, with weights being the sizes of the substrees.

Do you see why this gives $\mathcal{O}(m + \log n)$ query time?

Let us first see the folklore solution with $\mathcal{O}(m + \log n)$ query time that uses weight-balanced BSTs.

### Weight-balanced BST

Given an ordered collection of $n$ items, the $i$-th item having weight $w_i$ and $\sum_i w_i = W$, we can arrange them in a BST such that the depth of the $i$-th item is $\mathcal{O}(1 + \log(W/w_i))$.

See the problemset.

Now the solution is to simply store the outgoing edges (at each node) in weight-balanced BSTs, with weights being the sizes of the substrees.

Do you see why this gives $\mathcal{O}(m + \log n)$ query time?

Let us first see the folklore solution with $\mathcal{O}(m + \log n)$ query time that uses weight-balanced BSTs.

## Weight-balanced BST

Given an ordered collection of $n$ items, the $i$-th item having weight $w_i$ and $\sum_i w_i = W$, we can arrange them in a BST such that the depth of the $i$-th item is $\mathcal{O}(1 + \log(W/w_i))$.

See the problemset.

Now the solution is to simply store the outgoing edges (at each node) in weight-balanced BSTs, with weights being the sizes of the substrees.

Do you see why this gives $\mathcal{O}(m + \log n)$ query time?

To construct a static deterministic linear-size structure, we could simply to try to find a perfect hashing function storing pairs $(node, character)$. It is well-known that such functions can be found in polynomial time, but we need linear time.

**Ružić ICALP'08**

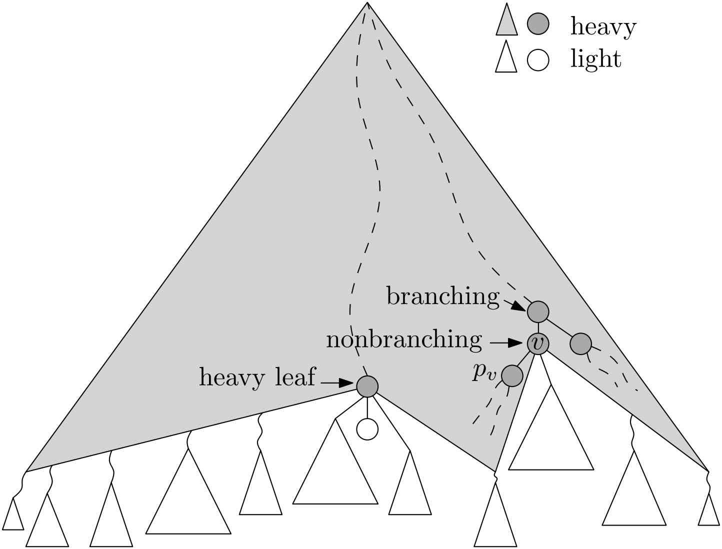
A static linear-size constant-access dictionary on a set of $k$ keys can be deterministically constructed in time $\mathcal{O}(k \log^2 \log k)$.

Hence we immediately get a static deterministic structure which can be constructed in close-to-linear time. Can we do better?

We store the edges outgoing from $v$ in a few different ways depending on the size of the subtree rooted at $v$.

### Heavy nodes

A node is heavy if its subtree contains at least $s = \Theta(\log^2 \log \sigma)$ leaves, and otherwise light. Furthermore, a heavy node is branching if it has more than one heavy child.


heavy
light
branching
nonbranching
$v$
$p_v$
heavy leaf

We classify edges outgoing from heavy nodes into three types, and deal with each type separately:

1. from (any) heavy node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node,

We classify edges outgoing from heavy nodes into three types, and deal with each type separately:

1. from (any) heavy node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node,

At most one such edge per node, can be stored separately.

We classify edges outgoing from heavy nodes into three types, and deal with each type separately:

1. from (any) heavy node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node,

The total number of such edges is just $\frac{n}{s}$, hence we can afford the super-linear construction time. More precisely, we compute perfect hashing functions for each such node separately in

$$\mathcal{O}(k \log^2 \log k) = \mathcal{O}(k \log^2 \log \sigma) = \mathcal{O}(ks)$$

time, which takes $\mathcal{O}(\frac{n}{s} s) = \mathcal{O}(n)$ time in total.

We classify edges outgoing from heavy nodes into three types, and deal with each type separately:

1. from (any) heavy node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node,

We store all such edges in a predecessor structure. By combining the perfect hashing result and the classical *x*-fast trees by Willard, there exists a linear-size predecessor structure with $\mathcal{O}(\log\log\sigma)$ query time, which can be constructed in linear time.

Observe that any navigation query traverses an edge of type (1) at most once, hence we pay $\mathcal{O}(\log\log\sigma)$ just once (so far). But what happens when we reach a light node?

Each light node contains at most $s$ leaves. We can execute a binary search over those leaves using the suffix array trick, namely in each step we achieve at least one of the following:

1. halve the current interval,
2. consume one character from the pattern.

Hence in $\mathcal{O}(m + \log s)$ time we can locate the predecessor of the pattern among all leaves, and the search actually computes the longest prefix of the pattern which is a prefix of a string corresponding to some leaf.

The total time complexity for a query is

$$\mathcal{O}(m + \log\log\sigma + \log s) = \mathcal{O}(m + \log\log\sigma)$$

and the total construction time is linear.

Questions?