Space and Time-Efficient Data Structures for Massive Datasets

Giulio Ermanno Pibiri
giulio.pibiri@di.unipi.it

Supervisor
Rossano Venturini

Computer Science Department
University of Pisa

17/10/2016

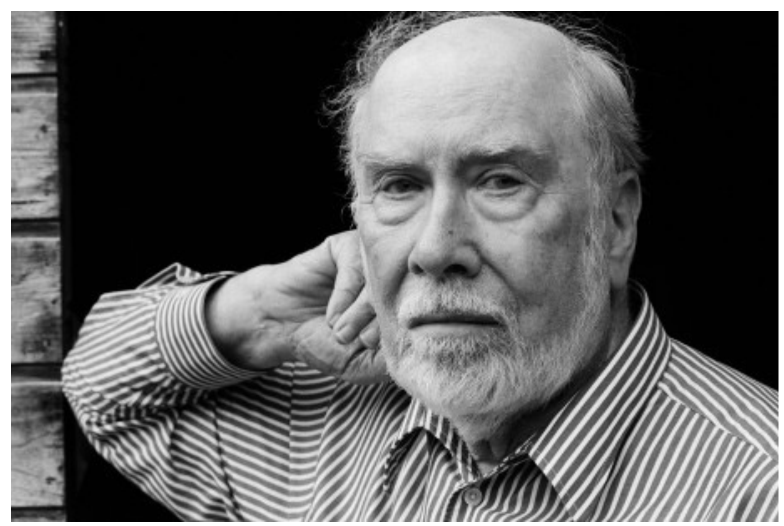

Statistics
Progress
Growth & Decline
Statistical
Analysis

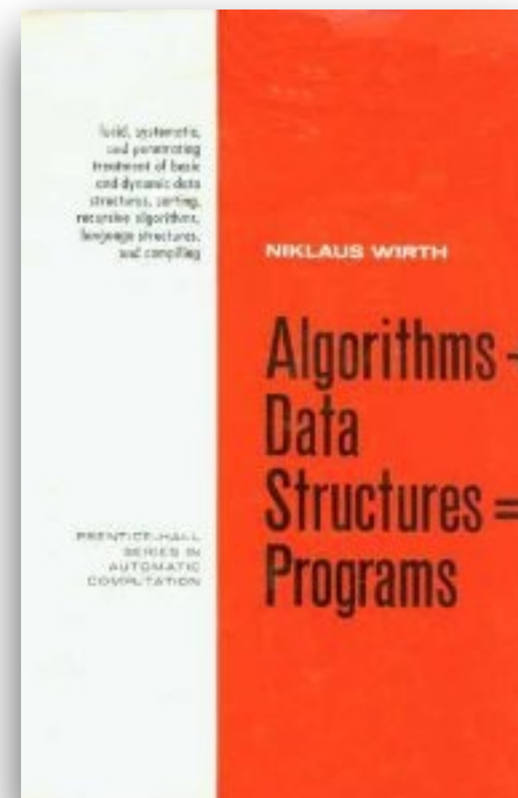

Evidence

The increase of information
does **not** scale with technology.

Evidence

The increase of information
does **not** scale with technology.





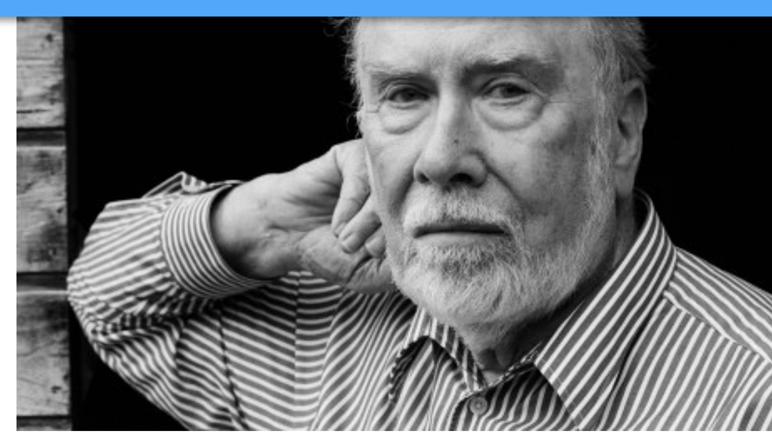
“Software is getting slower more rapidly than hardware becomes faster.”

Niklaus Wirth, A Plea for Lean Software

Evidence

The increase of information
does **not** scale with technology.

Even more relevant today!



“Software is getting slower more rapidly than hardware becomes faster.”

Niklaus Wirth, A Plea for Lean Software

Scenario

Data Structures

PERFORMANCE

"how quickly a program does its work" - **faster** work

Algorithms

EFFICIENCY

"how much work is required by a program" - **less** work

Scenario

Data Structures

PERFORMANCE

"how quickly a program does its work" - **faster** work

+ time
- space

Algorithms

EFFICIENCY

"how much work is required by a program" - **less** work

Scenario

Data Structures

PERFORMANCE

"how quickly a program does its work" - **faster** work

\+ time
\- space

Algorithms

EFFICIENCY

"how much work is required by a program" - **less** work

Data Compression

Scenario

Data Structures

PERFORMANCE

"how quickly a program does its work" - **faster** work

+ time
- space

Algorithms

EFFICIENCY

"how much work is required by a program" - **less** work

Data Compression

+ space
- time

Scenario

Data Structures

PERFORMANCE

"how quickly a program does its work" - **faster** work

\+ time
\- space

Algorithms

EFFICIENCY

"how much work is required by a program" - **less** work

?

Data Compression

\+ space
\- time

Dichotomy Problem

Small VS Fast?

Dichotomy Problem

Small VS Fast?

Choose one.

Dichotomy Problem

Small VS Fast?

Choose one.

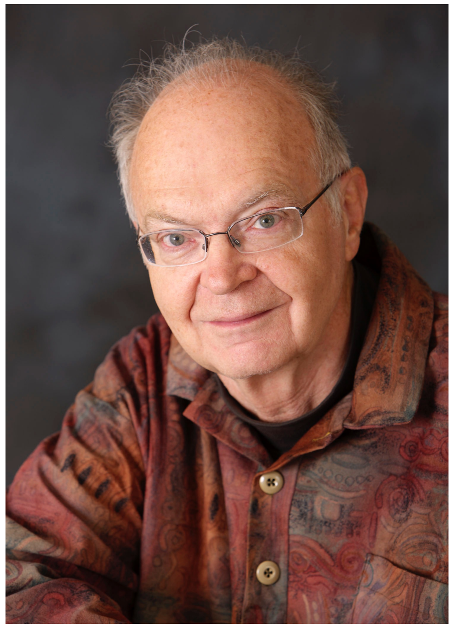
NO

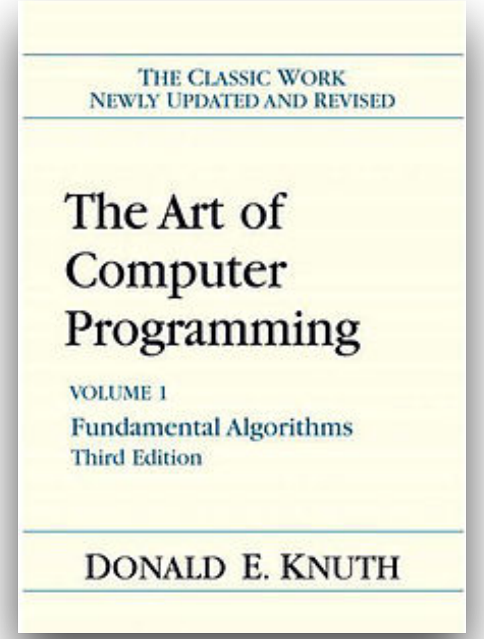
High Level Thesis

Data Structures + Data Compression → Faster Algorithms

High Level Thesis

Data Structures + Data Compression → Faster Algorithms

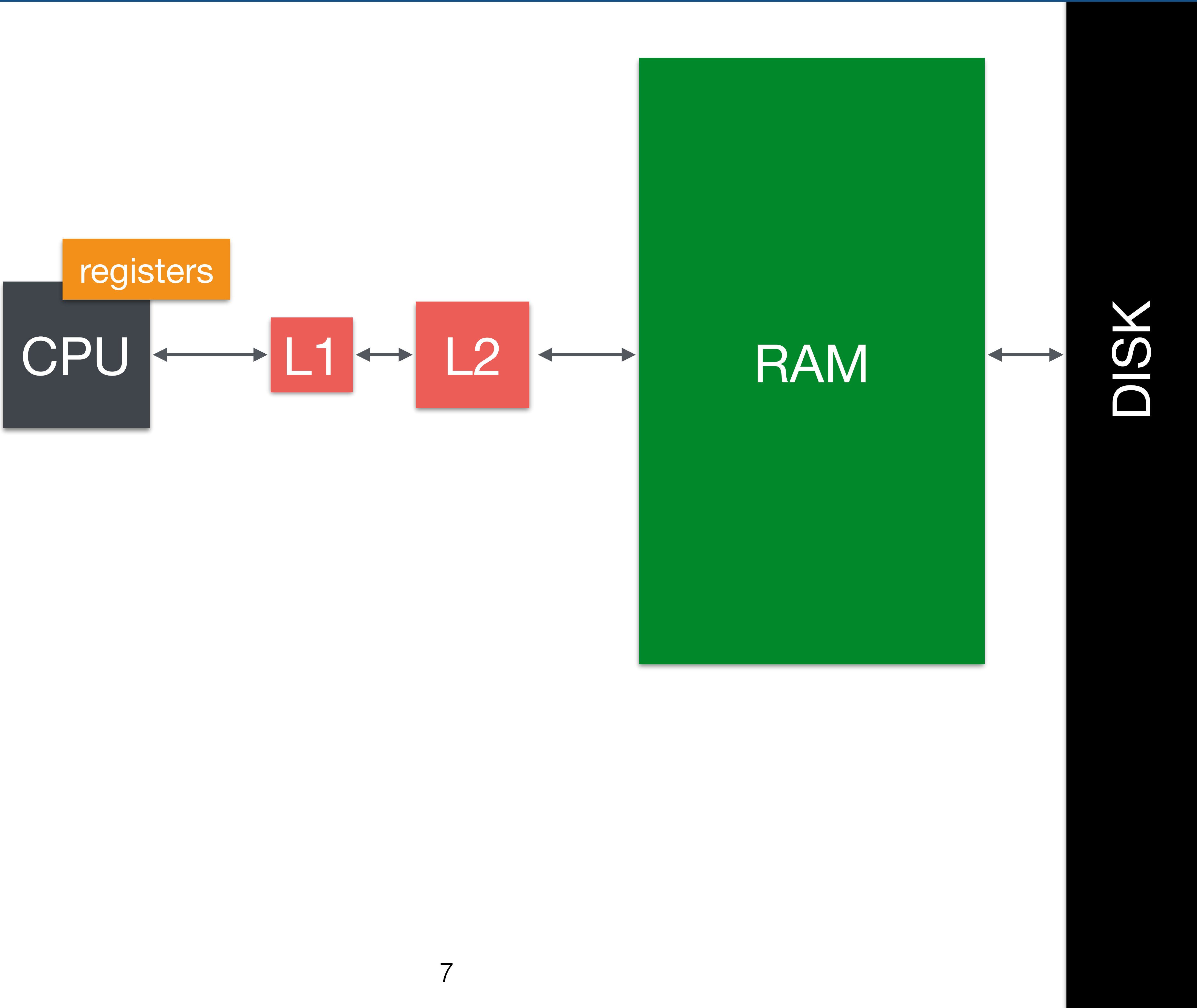




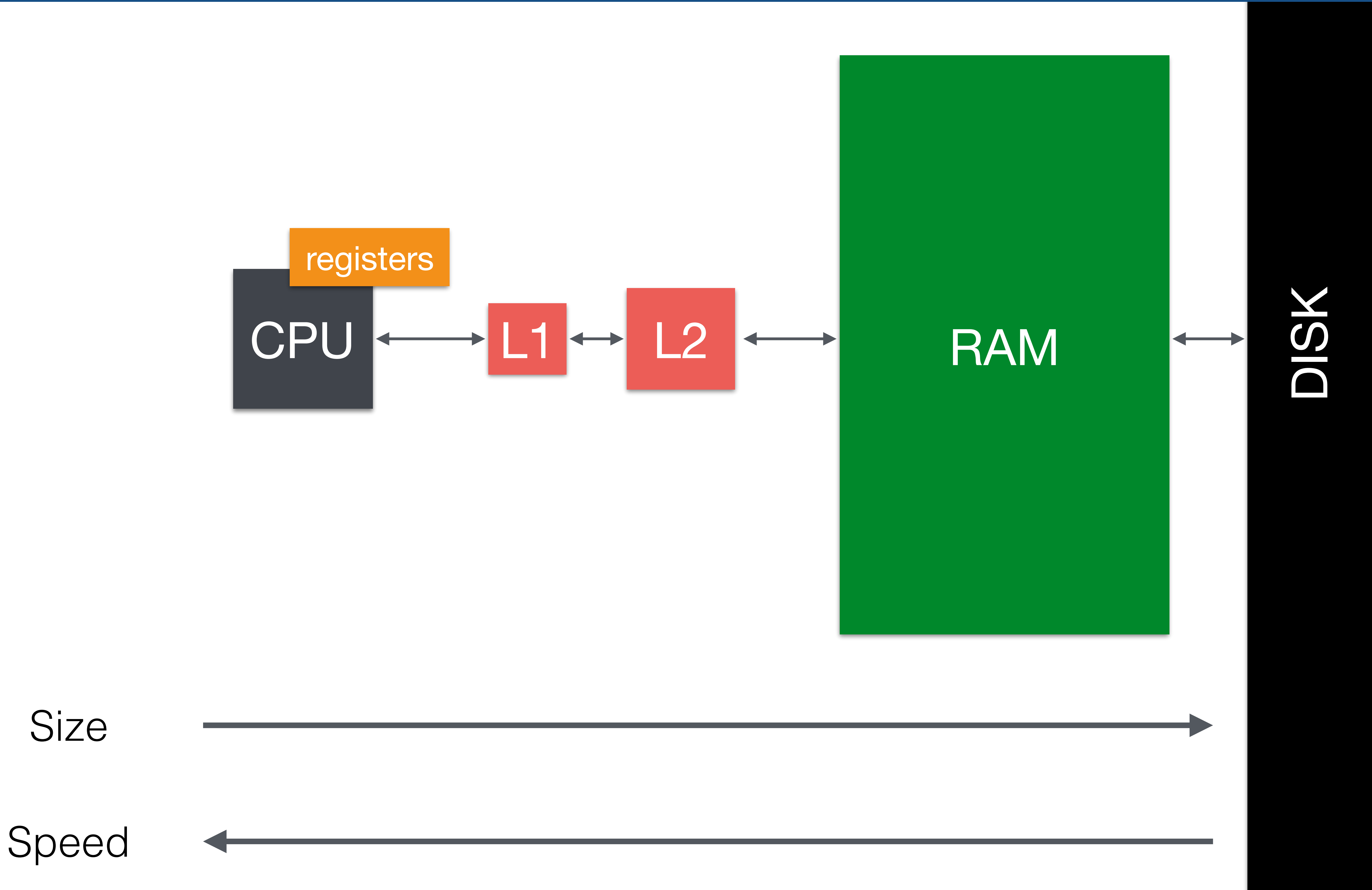
"Space optimization is closely related to time optimization in a disk memory."

Donald E. Knuth, The Art of Computer Programming

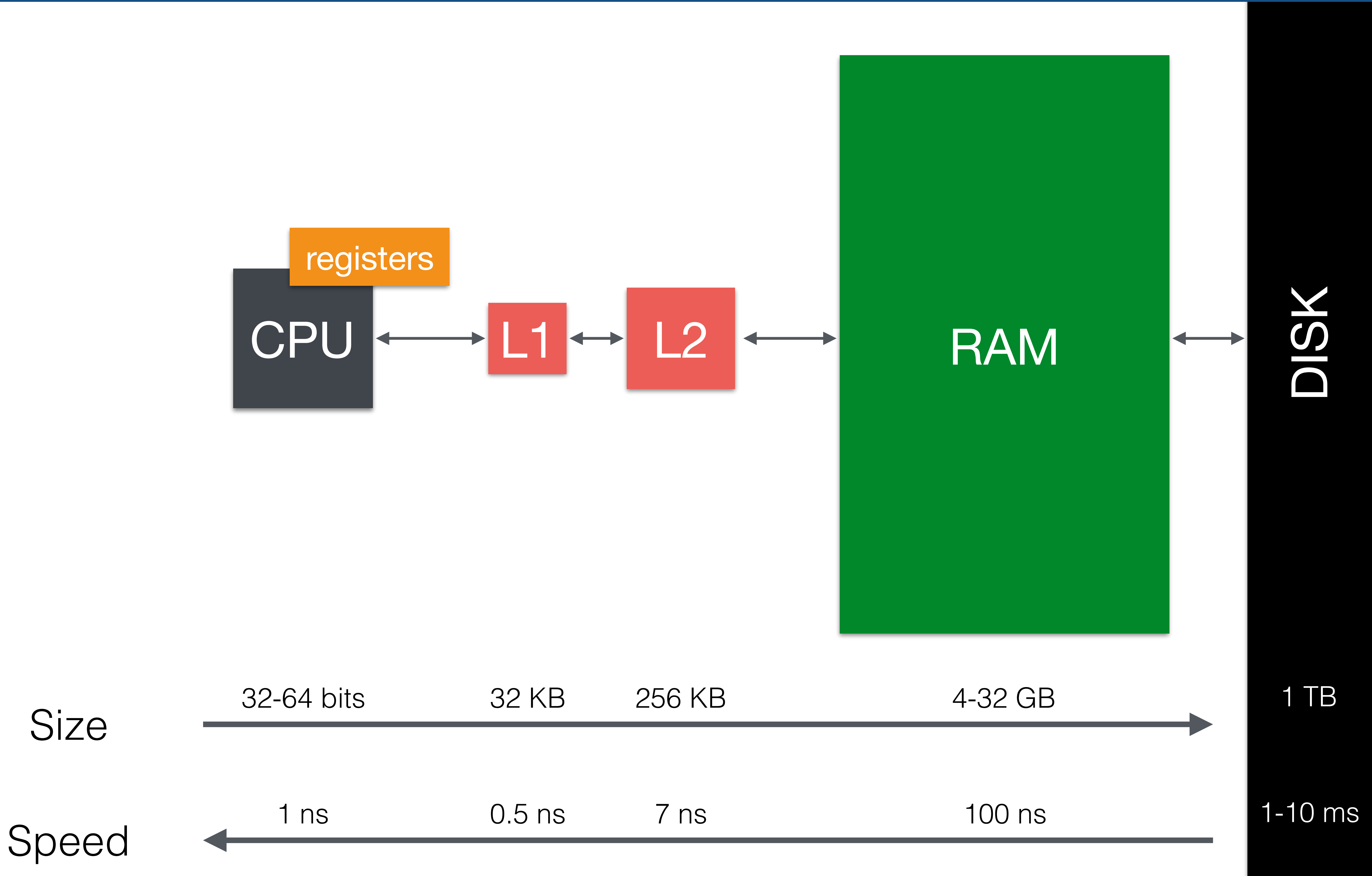
Hierarchical Memory Organisation



Hierarchical Memory Organisation

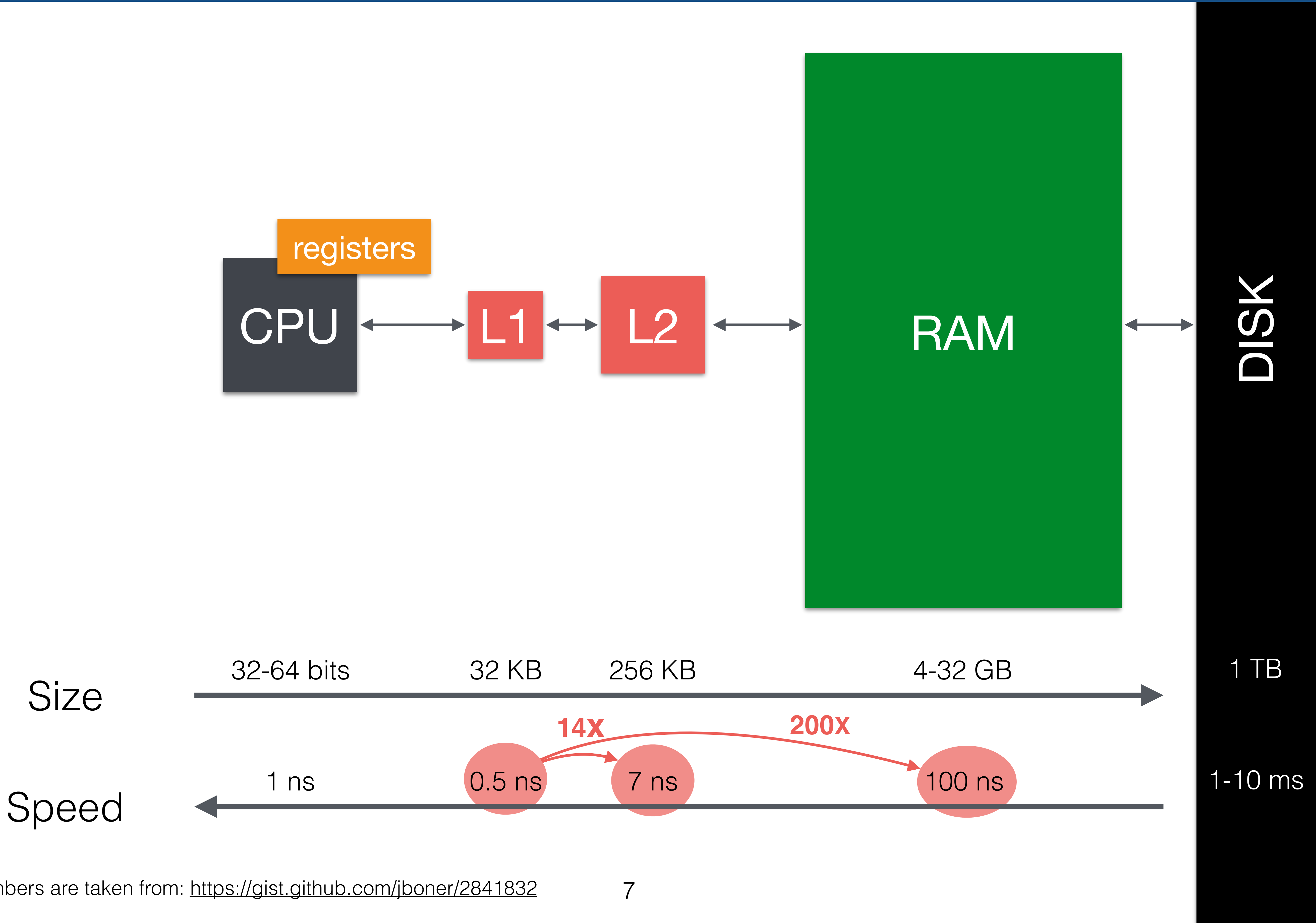


Hierarchical Memory Organisation



Numbers are taken from: https://gist.github.com/jboner/2841832

Hierarchical Memory Organisation



Numbers are taken from: https://gist.github.com/jboner/2841832

Proposal

Design **space-efficient** *ad-hoc* data structures, both from a theoretical *and* practical perspective, that support **fast data extraction**.

Proposal

Design **space-efficient** *ad-hoc* data structures,
both from a theoretical *and* practical perspective,
that support **fast data extraction**.

Data compression & Fast Retrieval
together.

Mature algorithmic solutions
now ready for technology transfer.

Proposal

Must exploit properties of the addressed problem.

Design **space-efficient** *ad-hoc* data structures,
both from a theoretical *and* practical perspective,
that support **fast data extraction**.

Data compression & Fast Retrieval
together.

Mature algorithmic solutions
now ready for technology transfer.

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

Google

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

facebook Google Dropbox

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Why? Who cares?

Because engineered data structures are the ones the *boost* the *availability* and *wealth* of information around us.

- Inverted Indexes
- N-grams
- B-trees
- …

Wait!

Can’t we use existing libraries?

Wait!

Can’t we use existing libraries?

C++ **S**tandard **T**emplate **L**ibrary (**STL**)?

Wait!

Can’t we use existing libraries?

C++ **S**tandard **T**emplate **L**ibrary (**STL**)?

std::list

std::stack

std::queue

std::map

std::unordered_map

Wait!

Can't we use existing libraries?

C++ **S**tandard **T**emplate **L**ibrary (**STL**)?

std::list

std::stack

std::queue

std::map

std::unordered_map

Wait!

Can’t we use existing libraries?

C++ **S**tandard **T**emplate **L**ibrary (**STL**)?

std::list

std::stack

std::queue

std::map

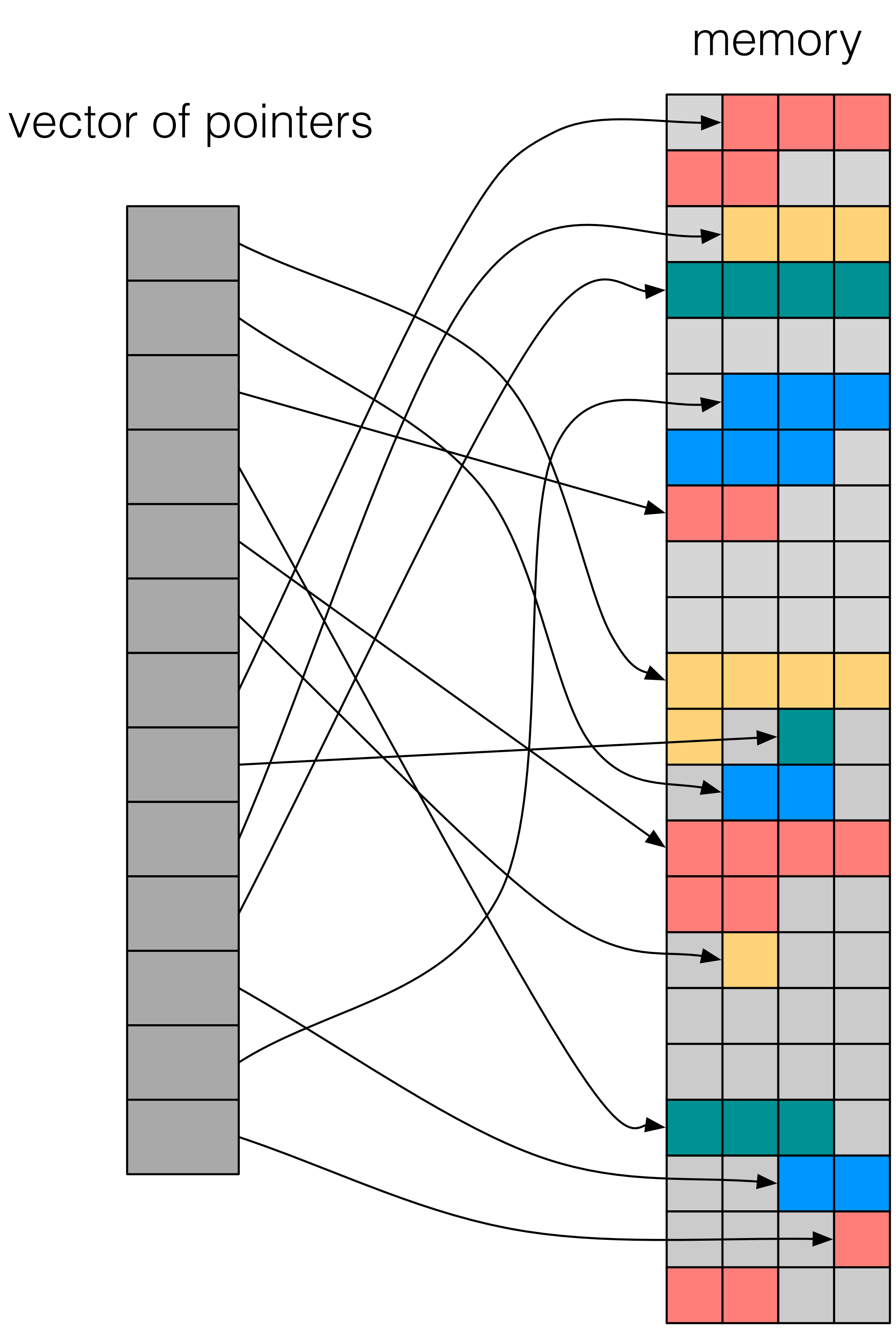
std::unordered_map

Prefer **cache-friendly** (non discontiguous) data structures.
Always.

Use std::vector.

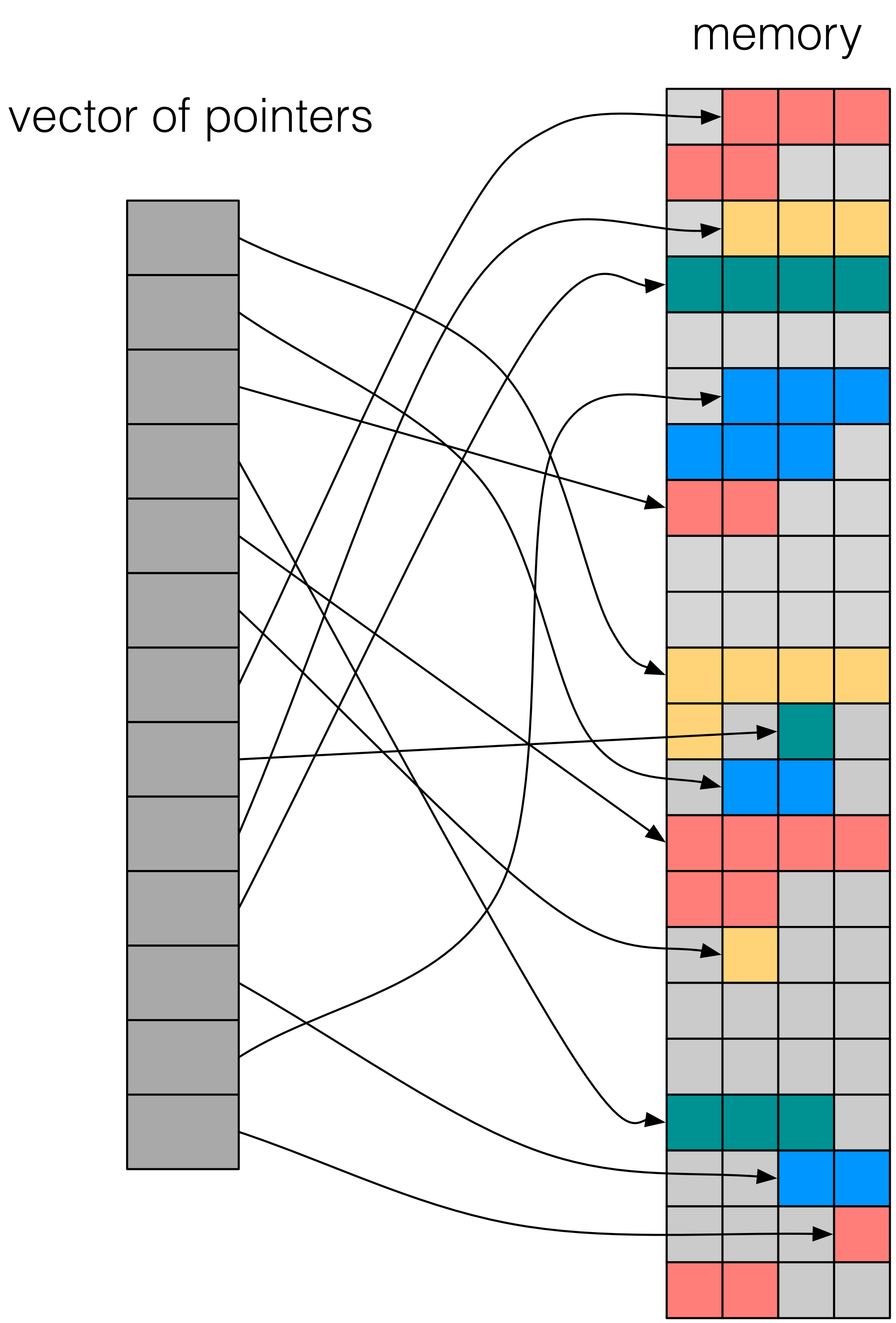
Example: vector of strings VS string pool

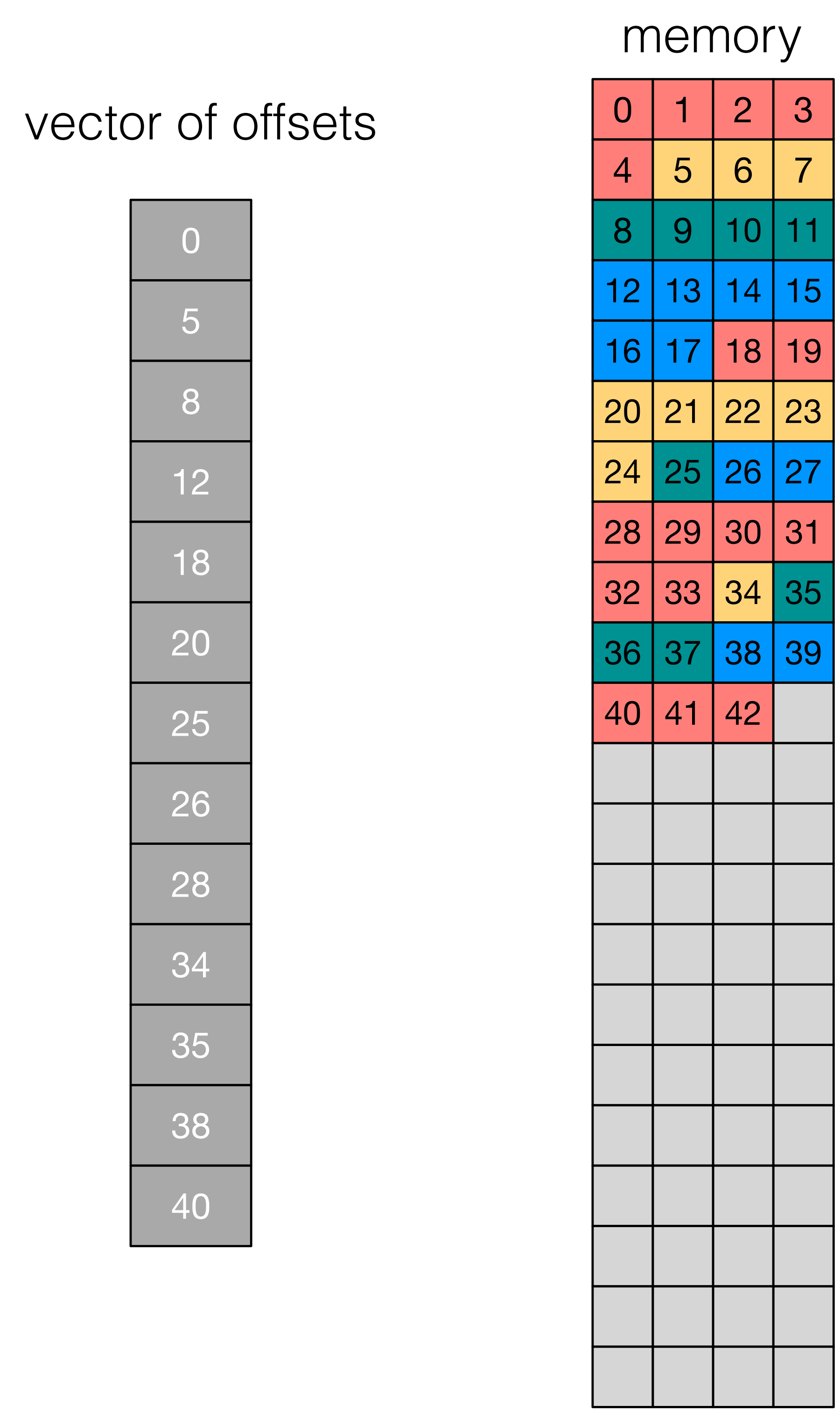
Prefer **cache-friendly** data structures.



Example: vector of strings VS string pool

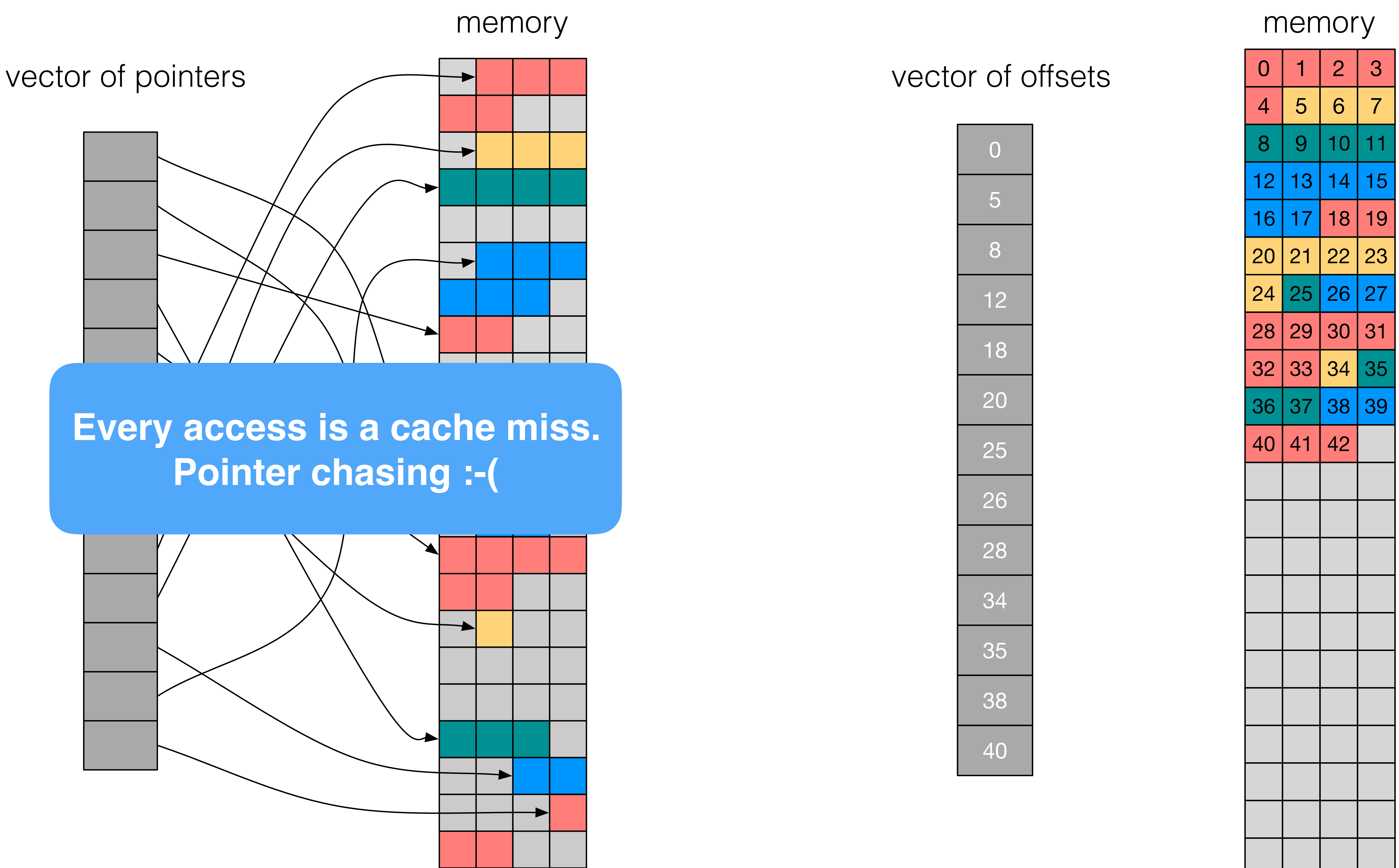
Prefer **cache-friendly** data structures.





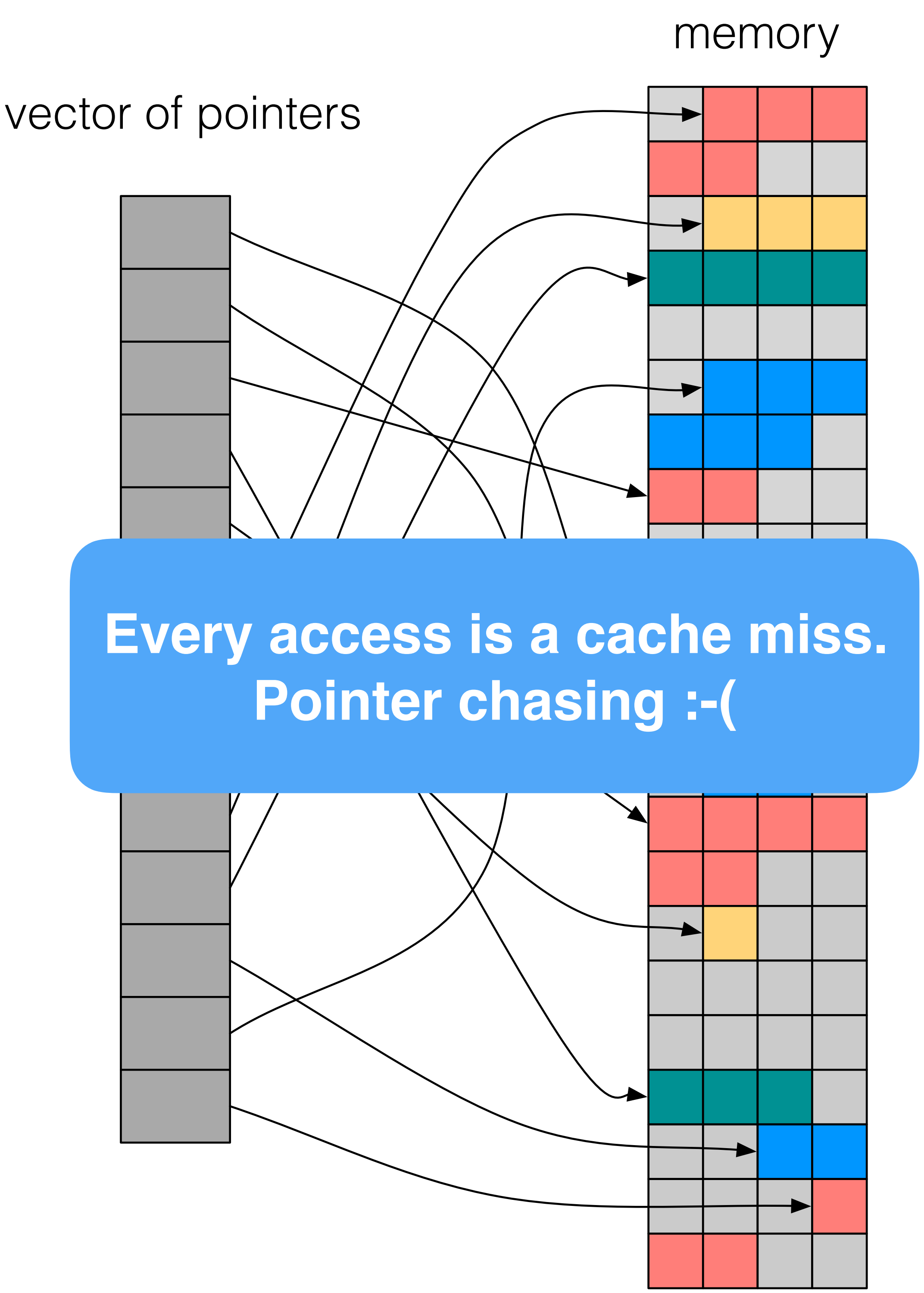
Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.

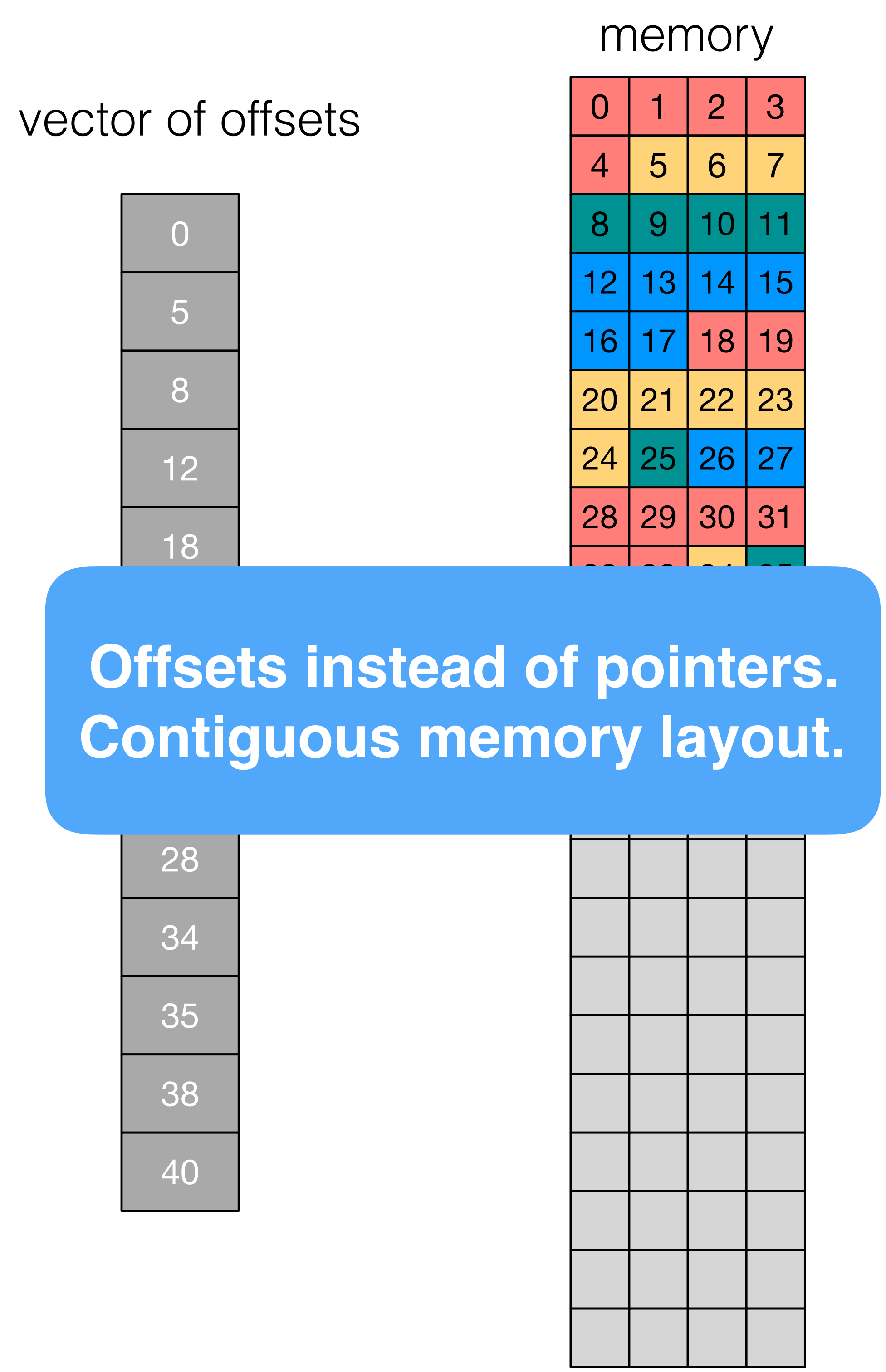


Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.



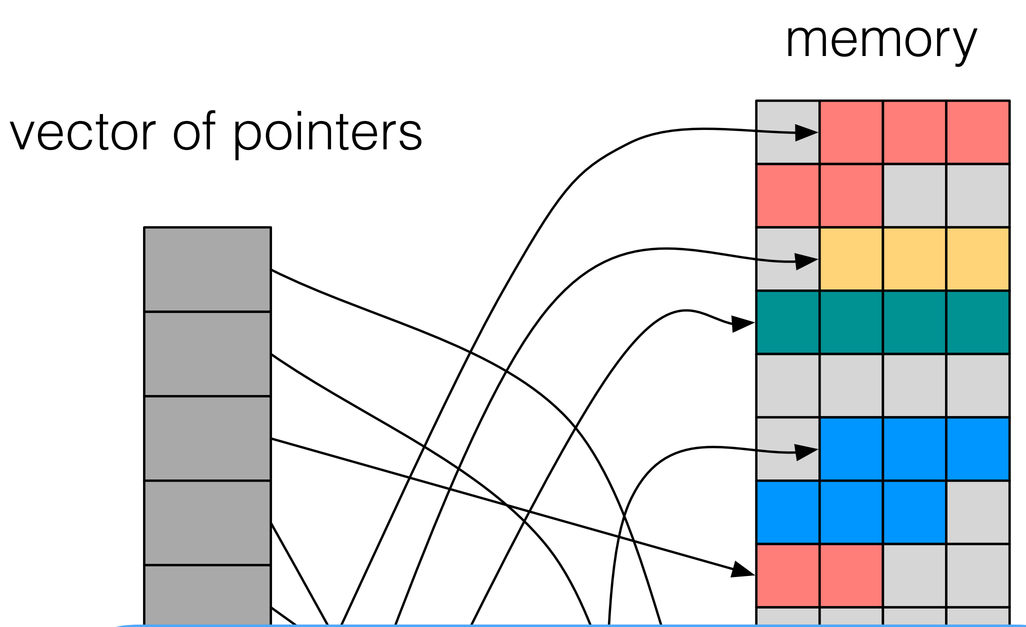
Every access is a cache miss.
Pointer chasing :-(

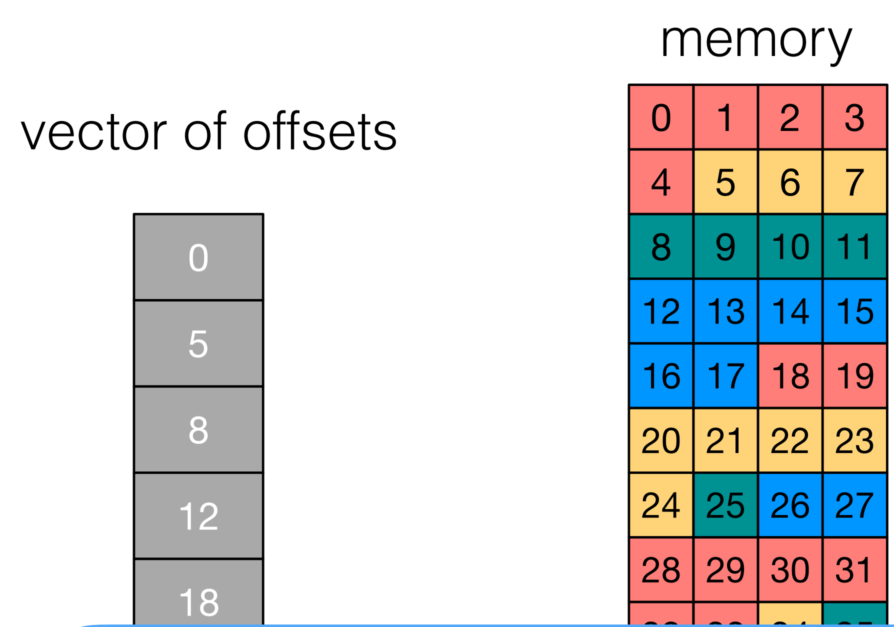


Offsets instead of pointers.
Contiguous memory layout.

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.





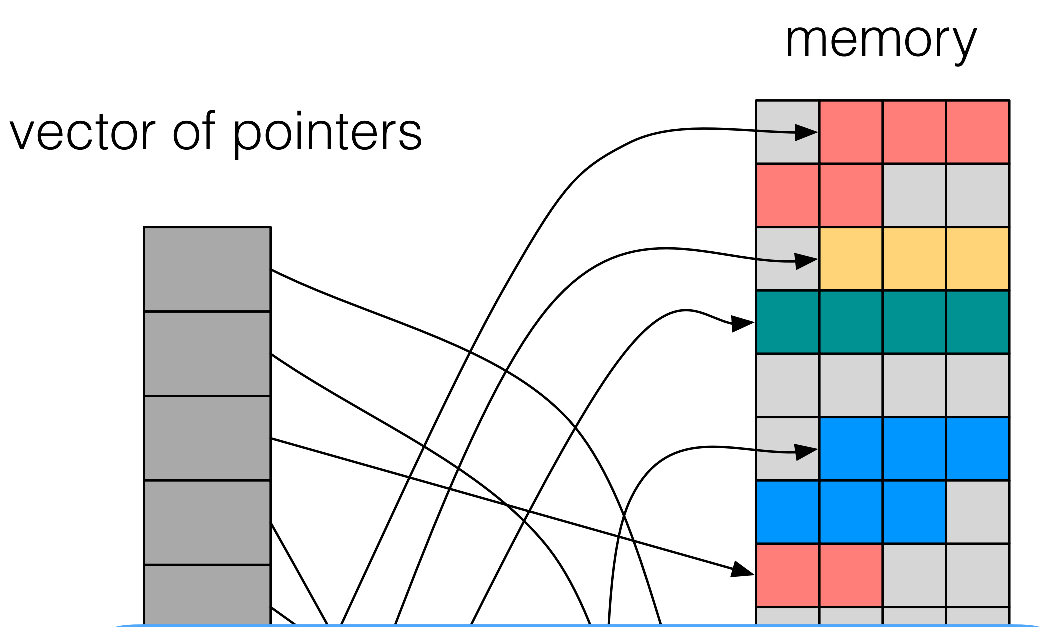
Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

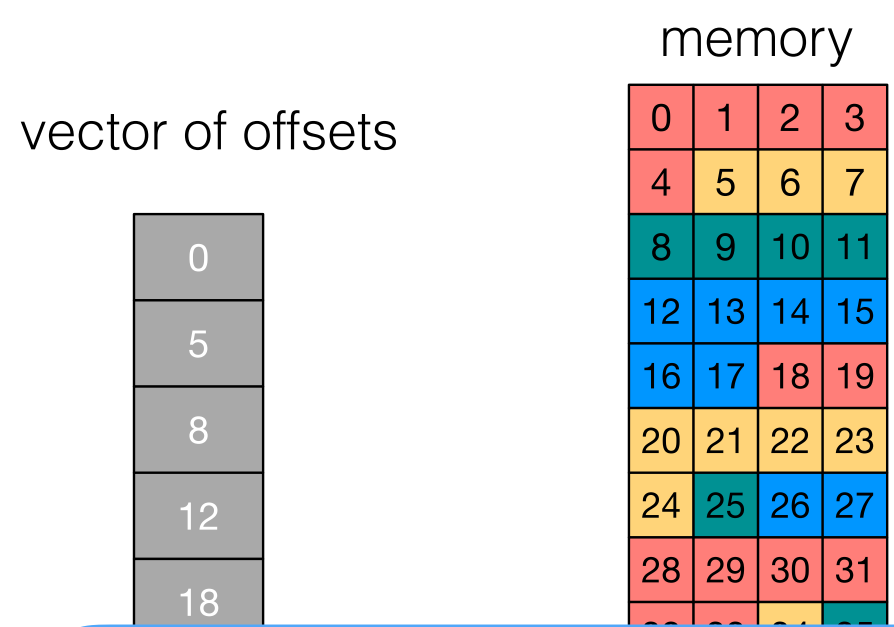
```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.



Every access is a cache miss.
Pointer chasing :-(

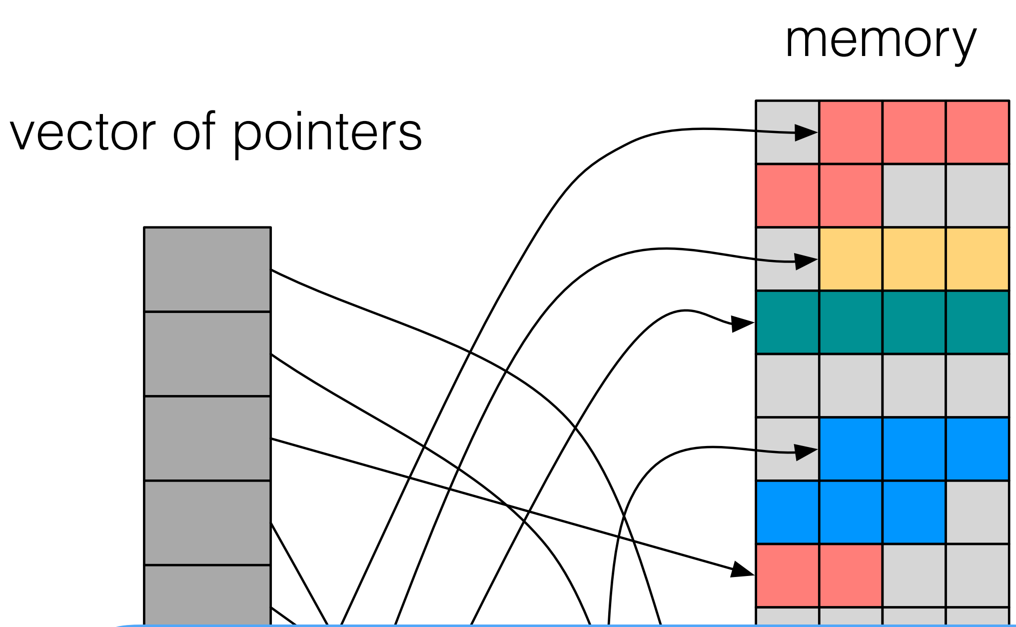


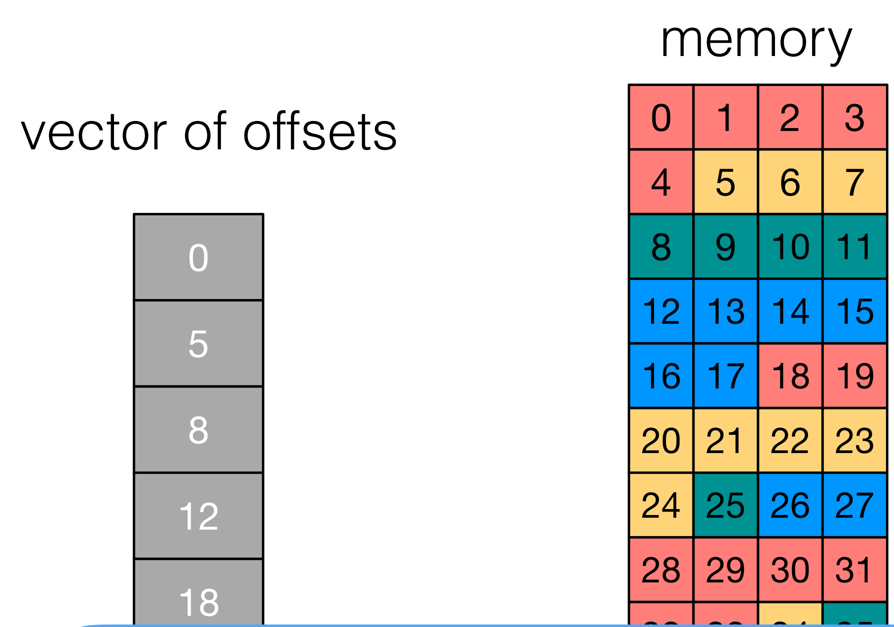
Offsets instead of pointers.
Contiguous memory layout.

```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.





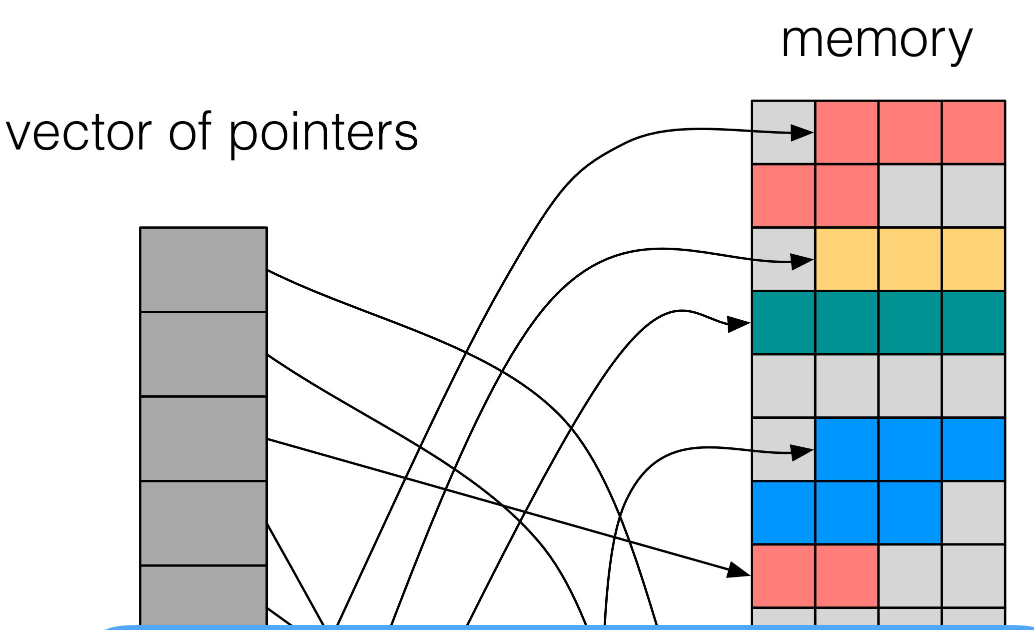
**Every access is a cache miss.
Pointer chasing :-(**

**Offsets instead of pointers.
Contiguous memory layout.**

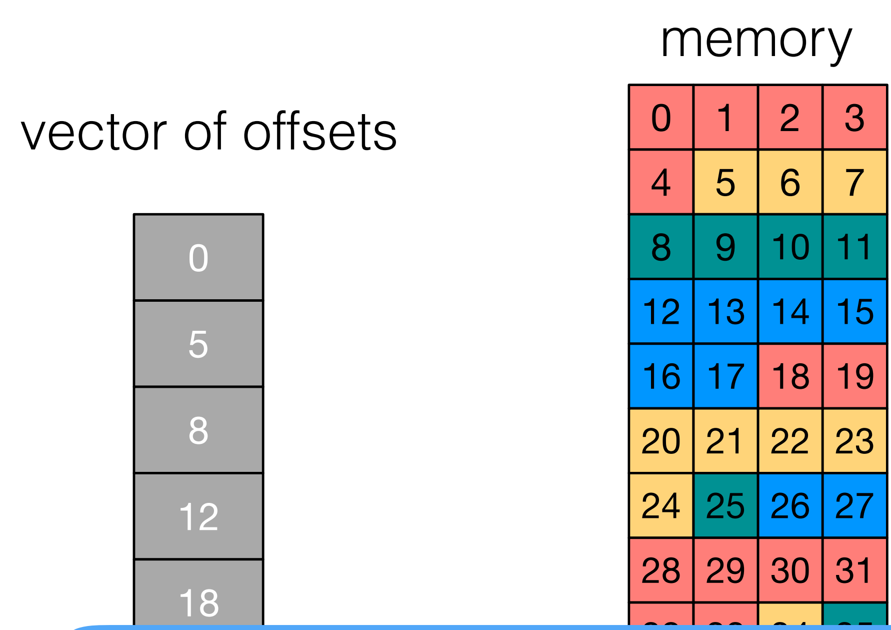
```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.



Every access is a cache miss.
Pointer chasing :-(



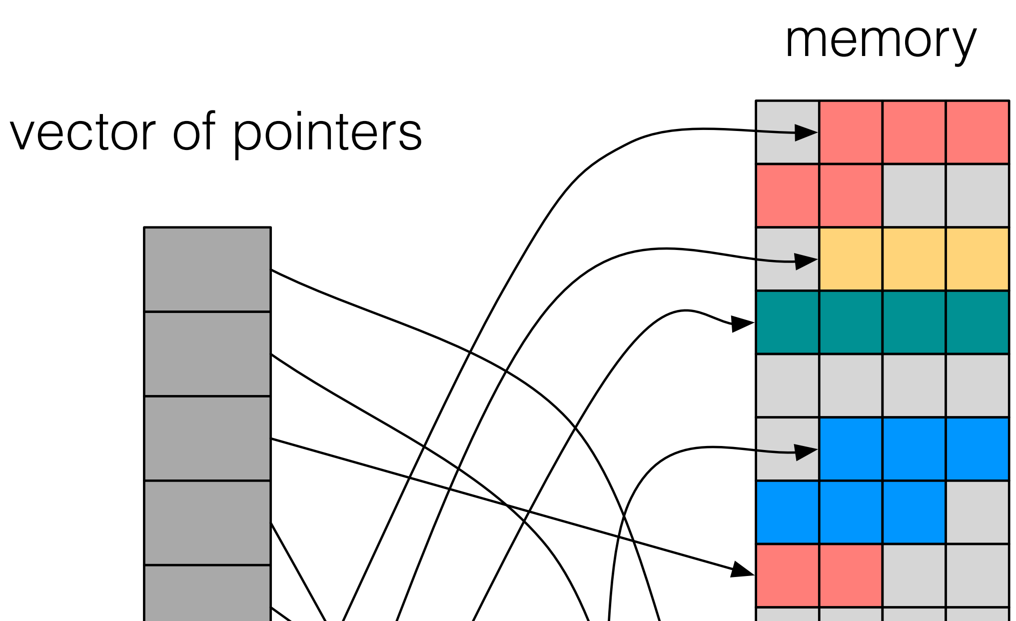
Offsets instead of pointers.
Contiguous memory layout.

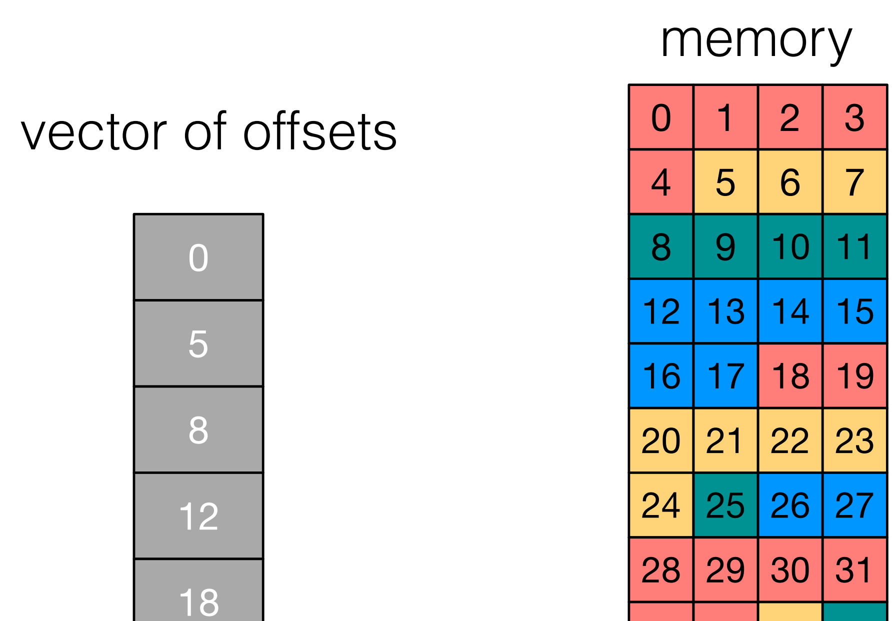
```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

~5.2GBs

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.





Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

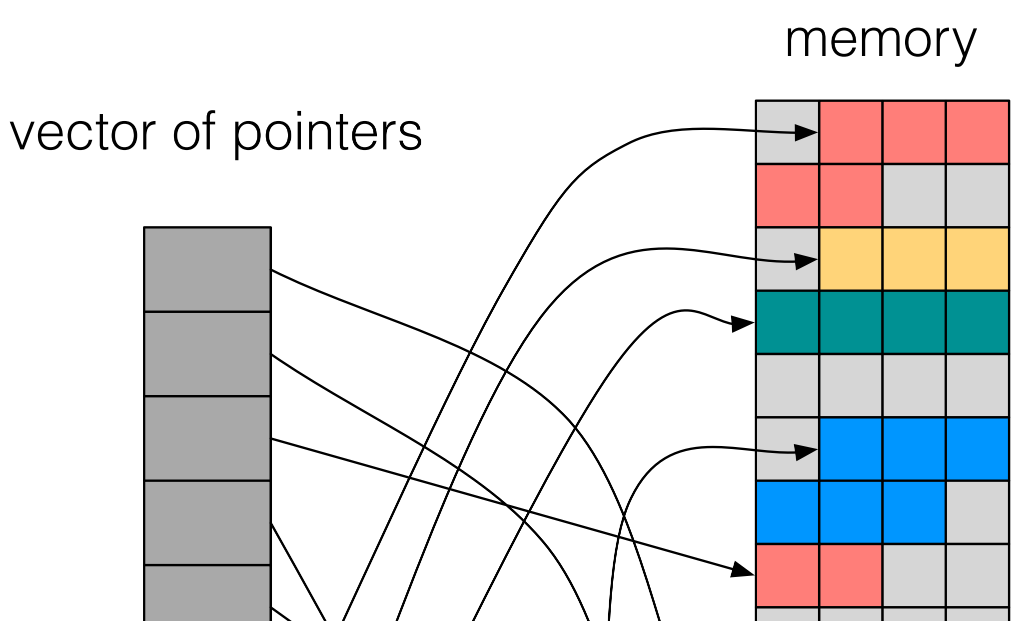
```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

~5.2GBs

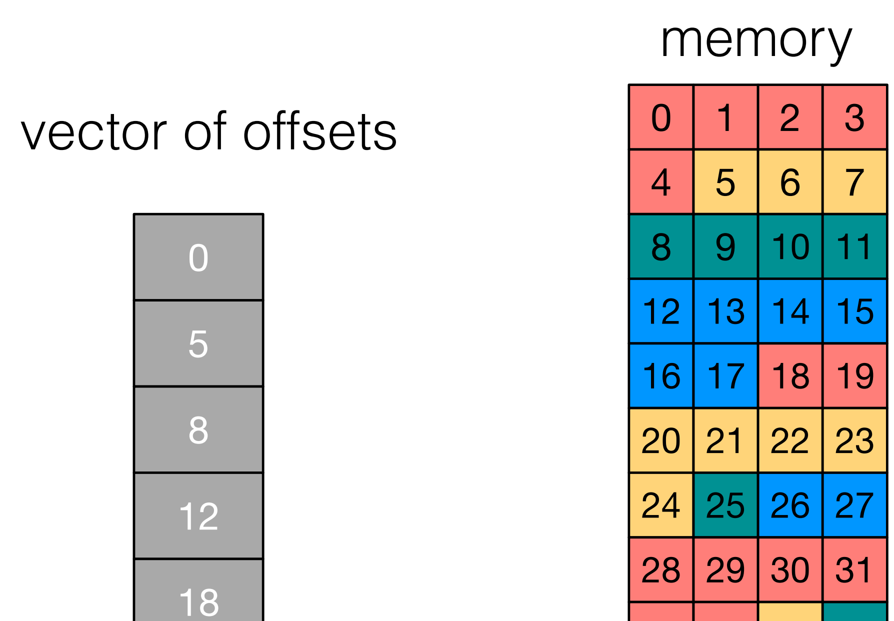
```
giulio@xor:~/sxlm/build$ ./string_pool_benchmark ~/random_strings.50M.128
2016-10-12 09:44:18: Loading strings
2016-10-12 09:44:26: read 3224822962 bytes
2016-10-12 09:44:26: Scanning strings
3224822962
2016-10-12 09:44:27: elapsed 1.341409 [sec]
```

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.



Every access is a cache miss.
Pointer chasing :-(



Offsets instead of pointers.
Contiguous memory layout.

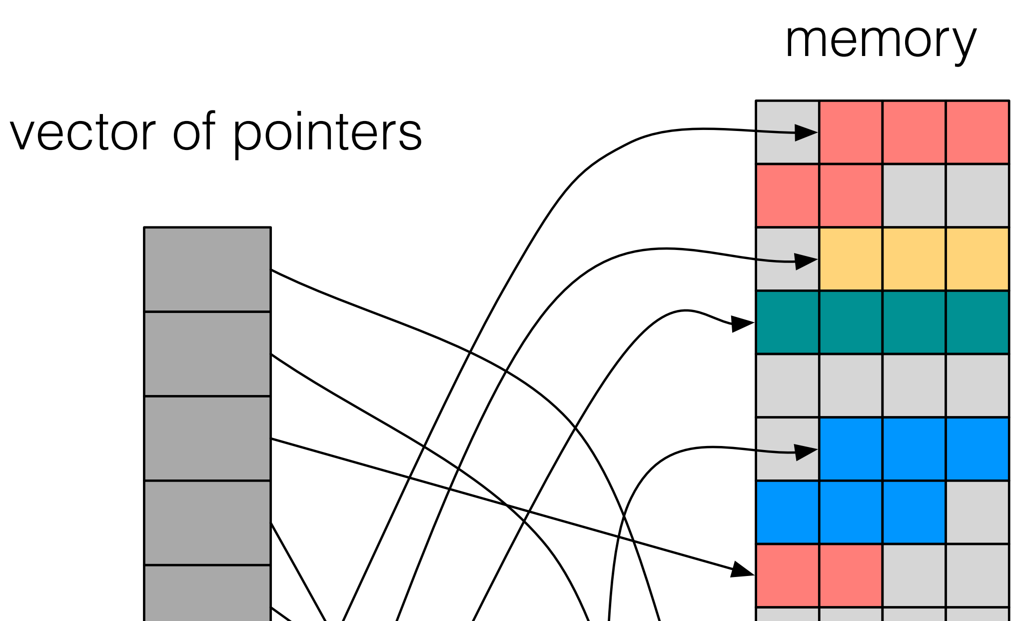
```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

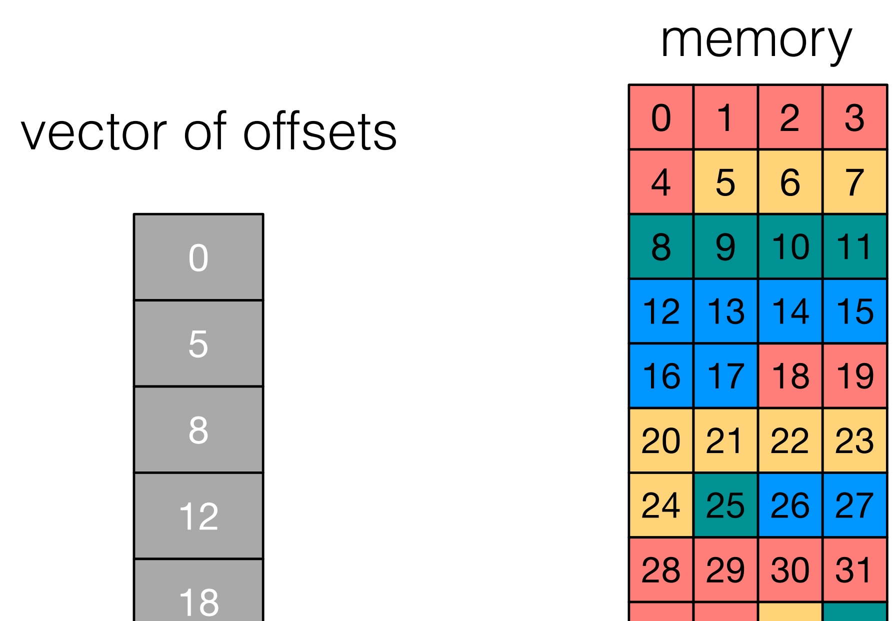
~5.2GBs

```
giulio@xor:~/sxlm/build$ ./string_pool_benchmark ~/random_strings.50M.128
2016-10-12 09:44:18: Loading strings
2016-10-12 09:44:26: read 3224822962 bytes
2016-10-12 09:44:26: Scanning strings
3224822962
2016-10-12 09:44:27: elapsed 1.341409 [sec]
```

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.





Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
```

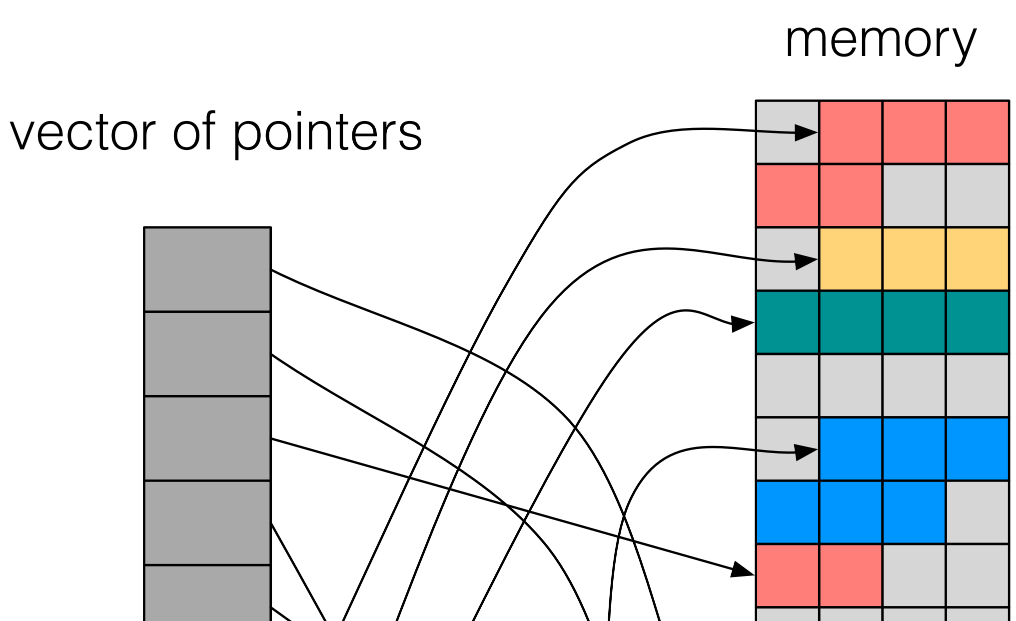
~5.2GBs

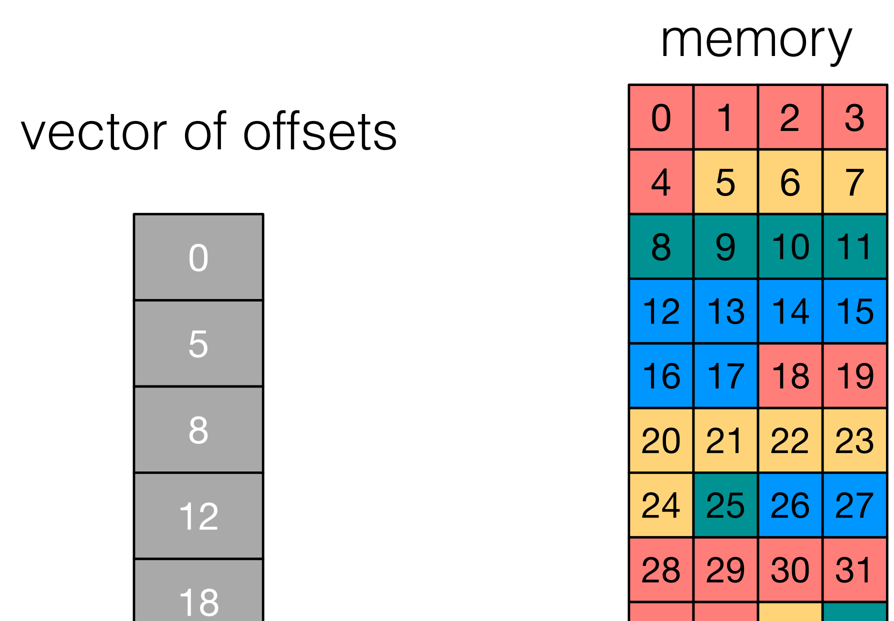
```
giulio@xor:~/sxlm/build$ ./string_pool_benchmark ~/random_strings.50M.128
2016-10-12 09:44:18: Loading strings
2016-10-12 09:44:26: read 3224822962 bytes
2016-10-12 09:44:26: Scanning strings
3224822962
2016-10-12 09:44:27: elapsed 1.341409 [sec]
```

~3GBs

Example: vector of strings VS string pool

Prefer **cache-friendly** data structures.





Every access is a cache miss.
Pointer chasing :-(

Offsets instead of pointers.
Contiguous memory layout.

```
giulio@xor:~/sxlm/build$ ./string_vector_benchmark 50000000 ~/random_strings.50M.128
2016-10-12 09:43:31: Loading strings
2016-10-12 09:43:39: Scanning strings
3224822962
2016-10-12 09:43:46: elapsed 6.705554 [sec]
giulio@xor:~/sxlm/build$ ./string_pool_benchmark ~/random_strings.50M.128
2016-10-12 09:44:18: Loading strings
2016-10-12 09:44:26: read 3224822962 bytes
2016-10-12 09:44:26: Scanning strings
3224822962
2016-10-12 09:44:27: elapsed 1.341409 [sec]
```

~5.2GBs

X5

~3GBs

Inverted Indexes

Given a *textual collection* D, each document can be seen as a (multi-)set of terms. The set of terms occurring in D is the *lexicon* T.

For each term t in T we store in a list L_t the identifiers of the documents in which t appears.

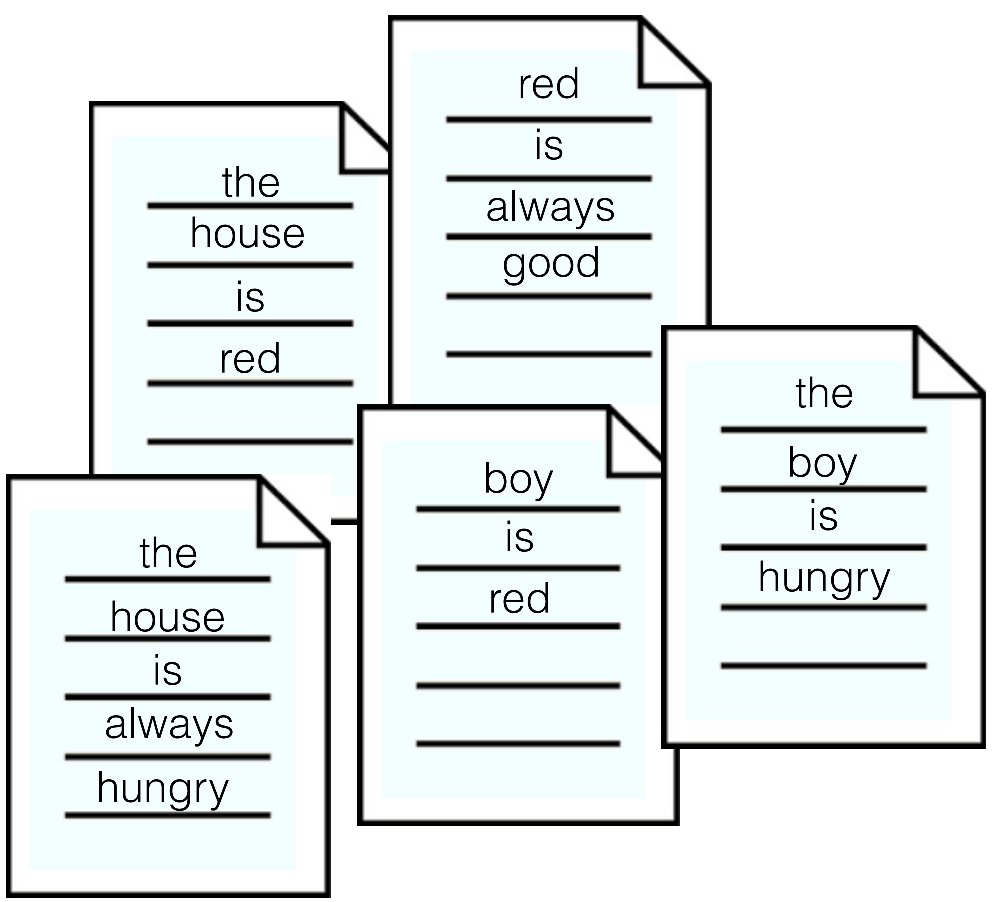
The collection of all inverted lists $\{L_{t_1},\ldots,L_{t_T}\}$ is the inverted index.

Inverted Indexes

Given a *textual collection* D, each document can be seen as a (multi-)set of terms. The set of terms occurring in D is the *lexicon* T.

For each term t in T we store in a list L_t the identifiers of the documents in which t appears.

The collection of all inverted lists $\{L_{t_1},\ldots,L_{t_T}\}$ is the inverted index.

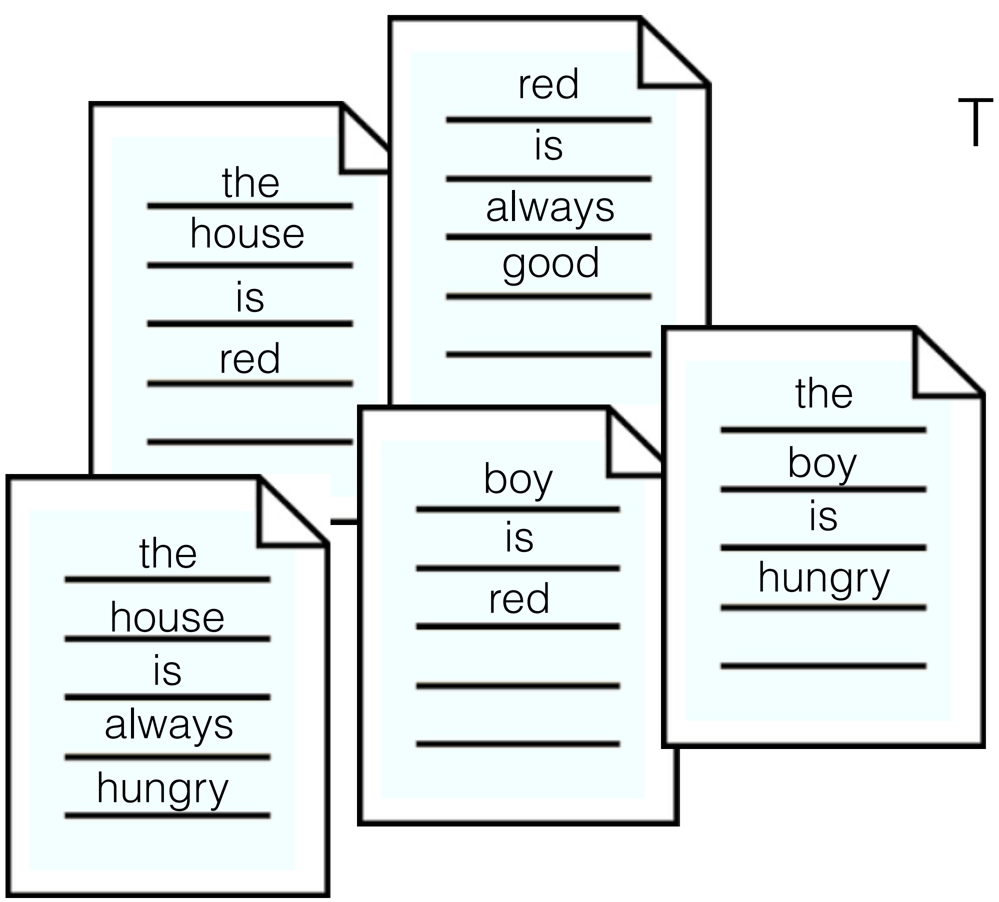


Inverted Indexes

Given a *textual collection* D, each document can be seen as a (multi-)set of terms. The set of terms occurring in D is the *lexicon* T.

For each term t in T we store in a list L_t the identifiers of the documents in which t appears.

The collection of all inverted lists $\{L_{t_1},\ldots,L_{t_T}\}$ is the inverted index.



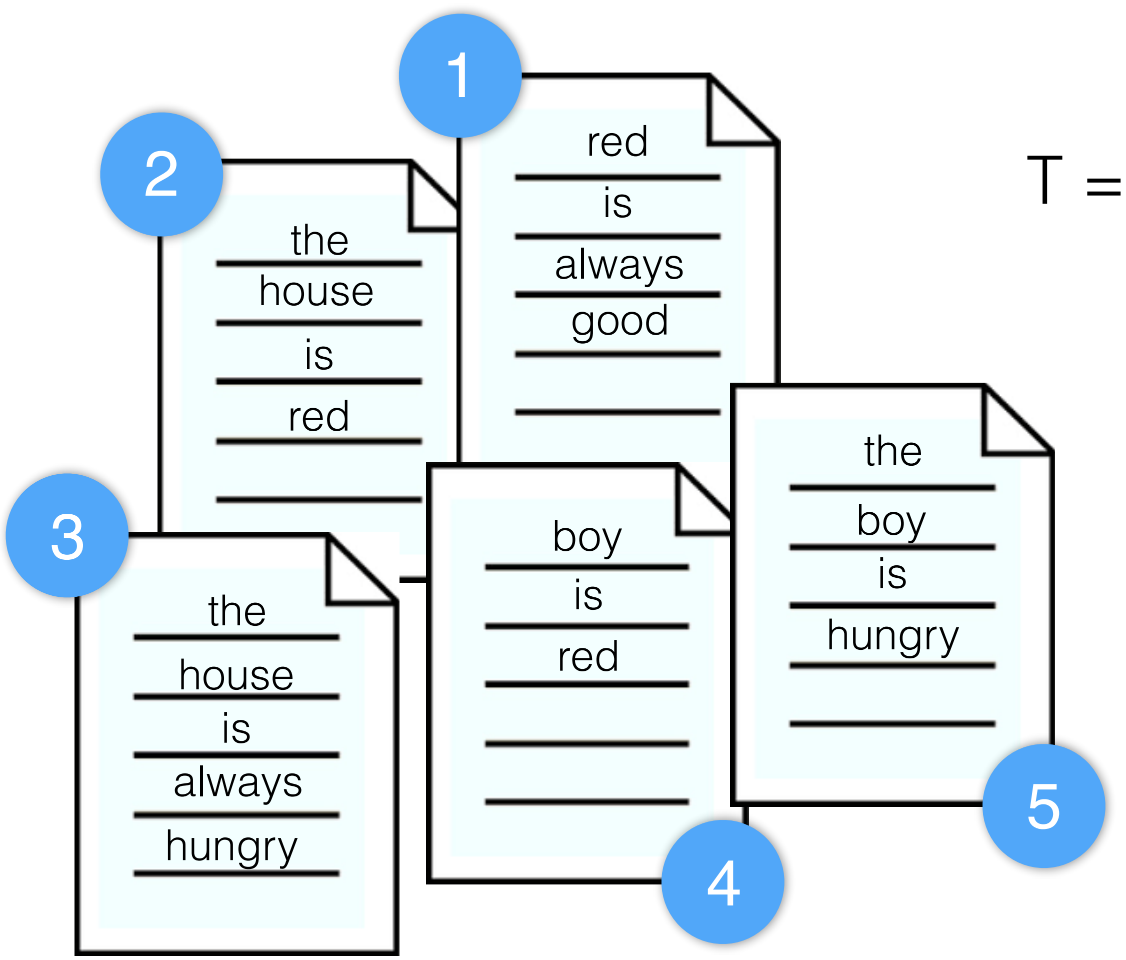
$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

Inverted Indexes

Given a *textual collection* D, each document can be seen as a (multi-)set of terms. The set of terms occurring in D is the *lexicon* T.

For each term t in T we store in a list L_t the identifiers of the documents in which t appears.

The collection of all inverted lists $\{L_{t_1}, \ldots, L_{t_T}\}$ is the inverted index.



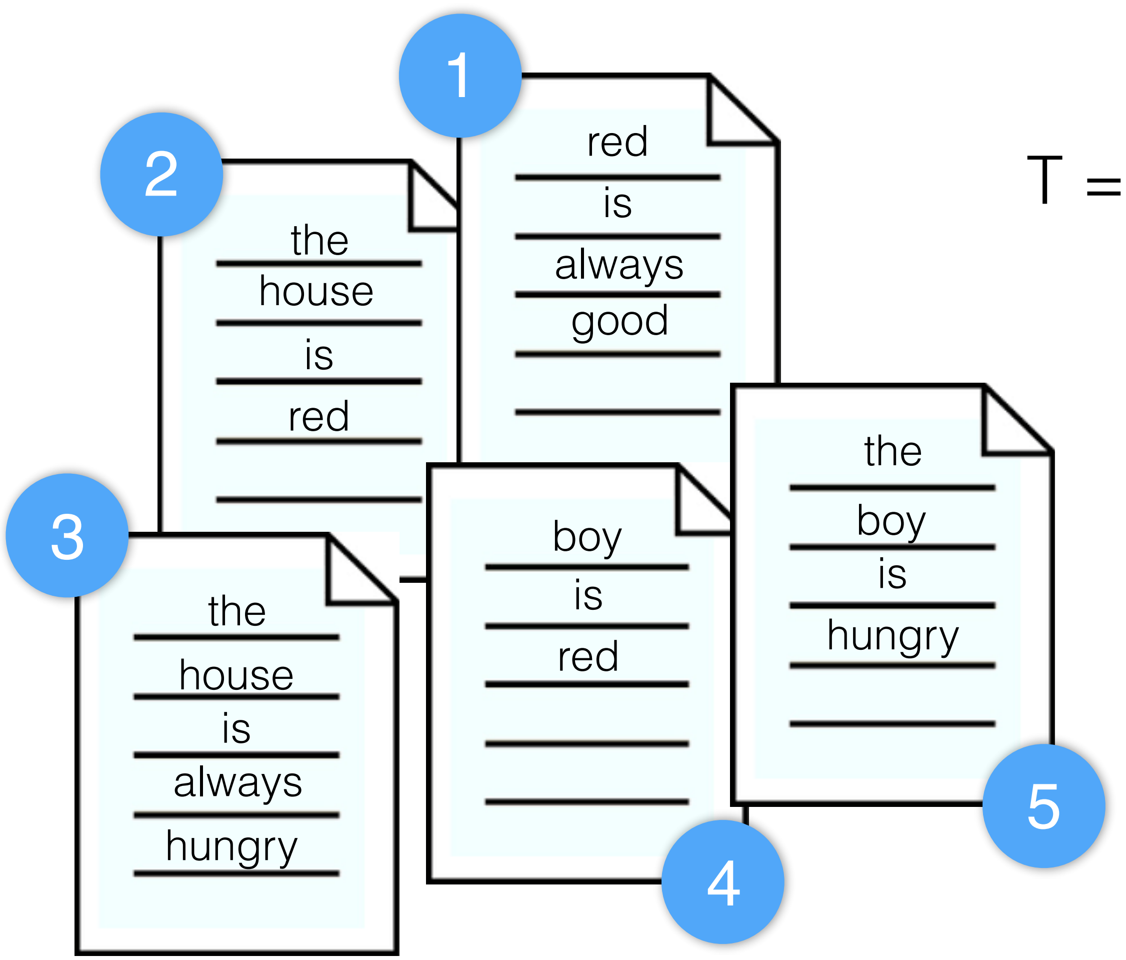
$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

Inverted Indexes

Given a *textual collection* D, each document can be seen as a (multi-)set of terms. The set of terms occurring in D is the *lexicon* T.

For each term t in T we store in a list L_t the identifiers of the documents in which t appears.

The collection of all inverted lists $\{L_{t_1},\ldots,L_{t_T}\}$ is the inverted index.



$$T = \{\underset{t_1}{\text{always}}, \underset{t_2}{\text{boy}}, \underset{t_3}{\text{good}}, \underset{t_4}{\text{house}}, \underset{t_5}{\text{hungry}}, \underset{t_6}{\text{is}}, \underset{t_7}{\text{red}}, \underset{t_8}{\text{the}}\}$$

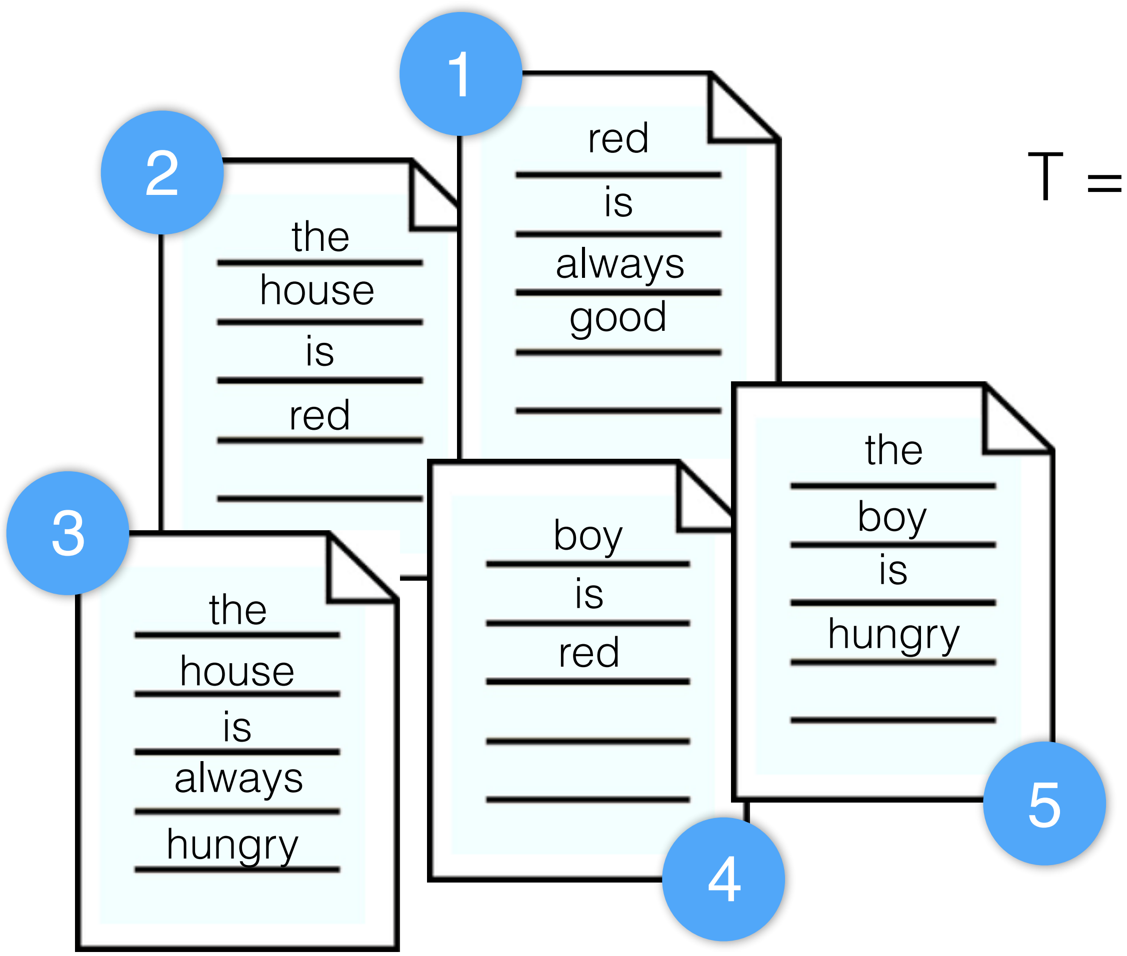
$L_{t_1}=[1, 3]$
$L_{t_2}=[4, 5]$
$L_{t_3}=[1]$
$L_{t_4}=[2, 3]$
$L_{t_5}=[3, 5]$
$L_{t_6}=[1, 2, 3, 4, 5]$
$L_{t_7}=[1, 2, 4]$
$L_{t_8}=[2, 3, 5]$

Inverted Indexes

Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms $\{t_1,\ldots,t_k\}$ occur”.

Inverted Indexes

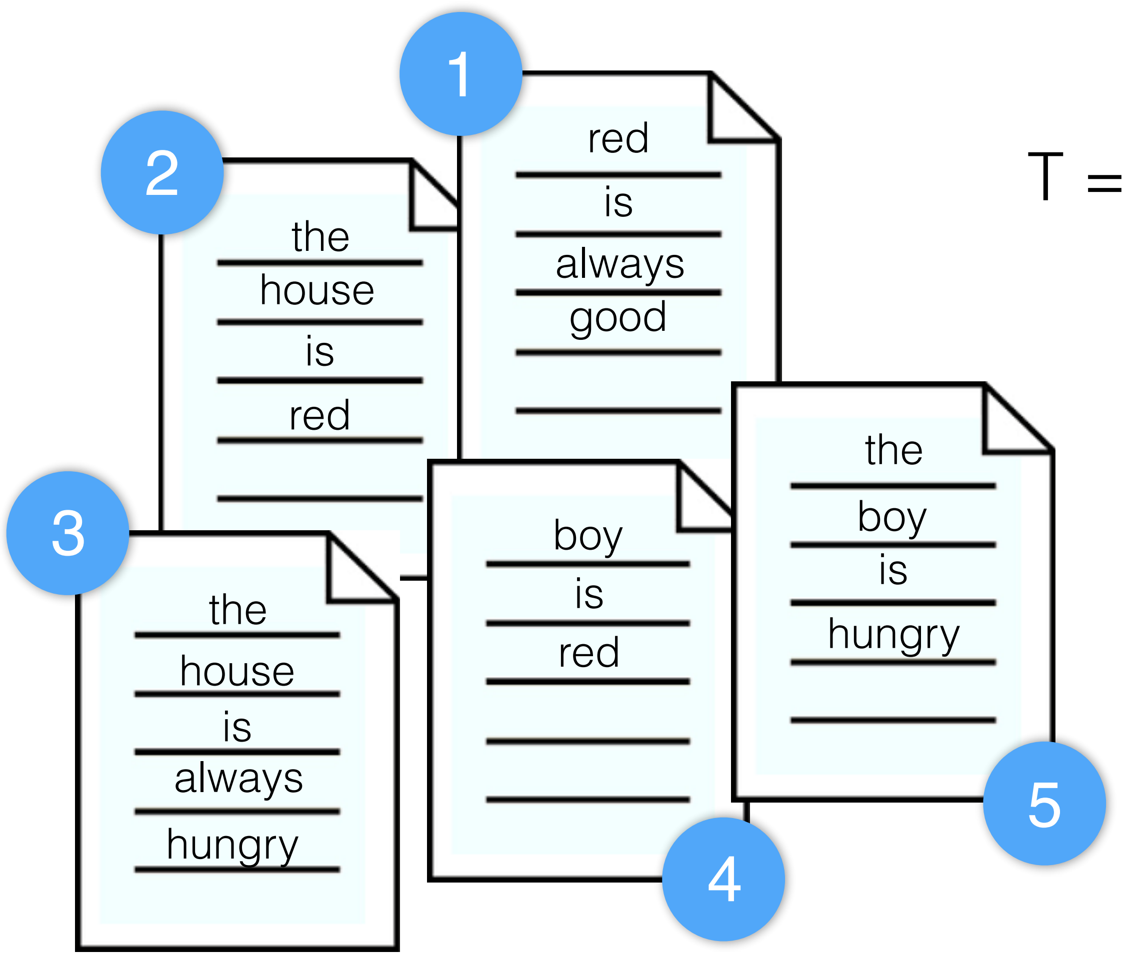
Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms $\{t_1,\ldots,t_k\}$ occur”.



$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

$L_{t_1}=[1, 3]$
$L_{t_2}=[4, 5]$
$L_{t_3}=[1]$
$L_{t_4}=[2, 3]$
$L_{t_5}=[3, 5]$
$L_{t_6}=[1, 2, 3, 4, 5]$
$L_{t_7}=[1, 2, 4]$
$L_{t_8}=[2, 3, 5]$

Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms $\{t_1,\dots,t_k\}$ occur”.



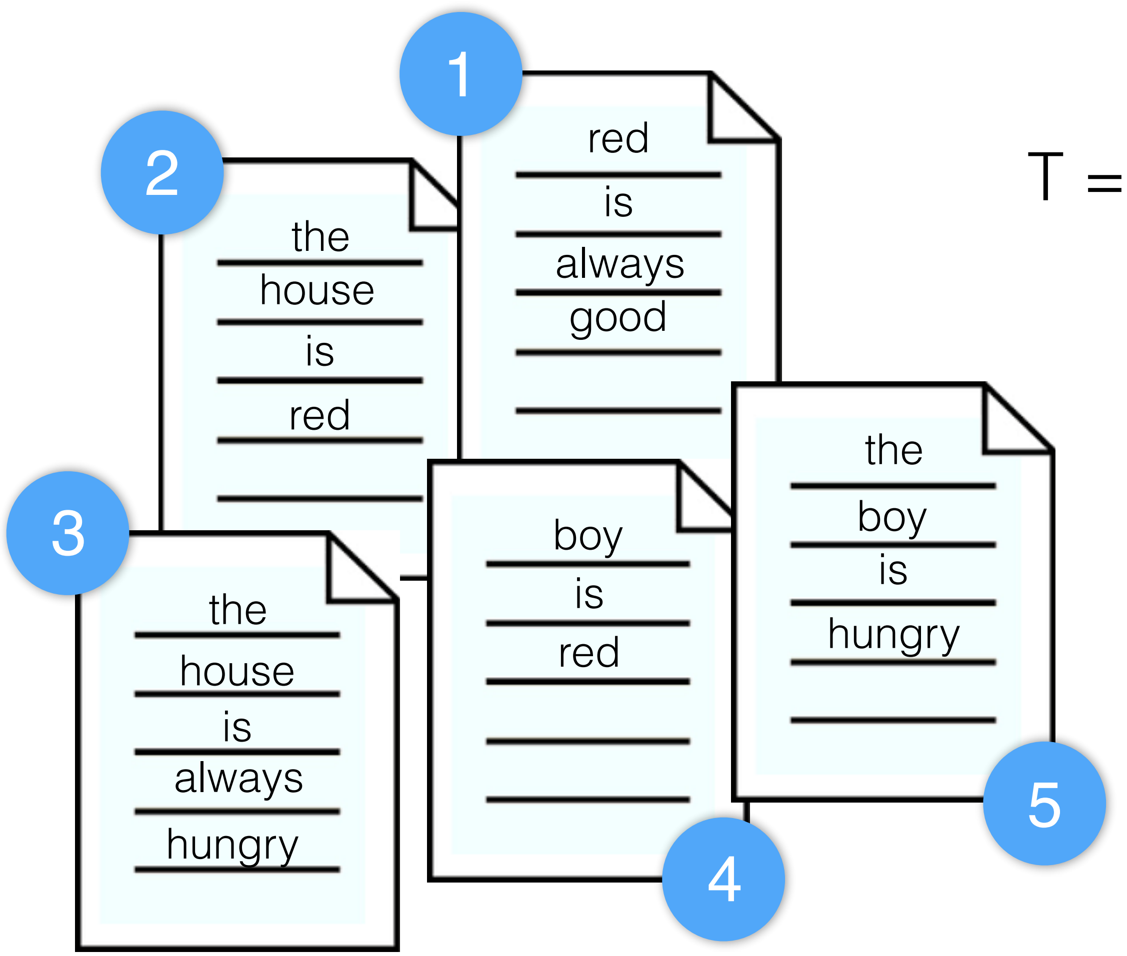
$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

$L_{t_1}=[1, 3]$
$L_{t_2}=[4, 5]$
$L_{t_3}=[1]$
$L_{t_4}=[2, 3]$
$L_{t_5}=[3, 5]$
$L_{t_6}=[1, 2, 3, 4, 5]$
$L_{t_7}=[1, 2, 4]$
$L_{t_8}=[2, 3, 5]$

$q = \{\text{boy}, \text{is}, \text{the}\}$

Inverted Indexes 2

Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms $\{t_1,\ldots,t_k\}$ occur”.



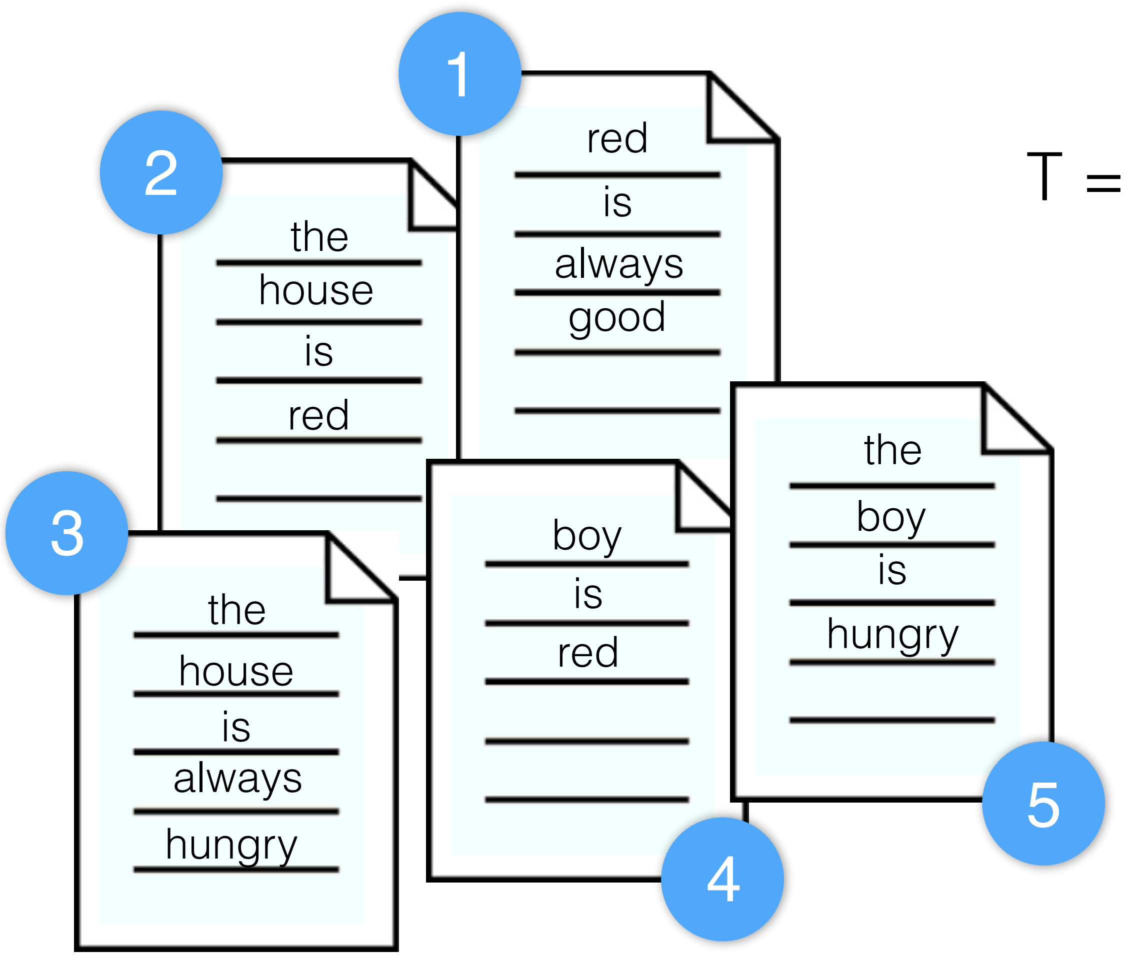
$$\begin{array}{cccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 \\ \end{array}$$

$$T = \{\text{always, boy, good, house, hungry, is, red, the}\}$$

L_{t_1}=[1, 3]
L_{t_2}=[4, 5]
L_{t_3}=[1]
L_{t_4}=[2, 3]
L_{t_5}=[3, 5]
L_{t_6}=[1, 2, 3, 4, 5]
L_{t_7}=[1, 2, 4]
L_{t_8}=[2, 3, 5]

q = {boy, is, the}

Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms $\{t_1,\ldots,t_k\}$ occur”.



$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$

$L_{t_1}=[1, 3]$
$L_{t_2}=[4, 5]$
$L_{t_3}=[1]$
$L_{t_4}=[2, 3]$
$L_{t_5}=[3, 5]$
$L_{t_6}=[1, 2, 3, 4, 5]$
$L_{t_7}=[1, 2, 4]$
$L_{t_8}=[2, 3, 5]$

$q = \{\text{boy, is, the}\}$

posting lists intersection

General Problem

Consider a sequence S[0,n) of n *positive* and *monotonically increasing integers*, i.e., $S[i] \leq S[i+1]$ for $0 \leq i < n-1$, possibly repeated.

How to represent it as a *bit vector* in which each original integer is *self-delimited*, using as few as possible bits?

General Problem

Consider a sequence S[0,n) of n *positive* and *monotonically increasing integers*, i.e., S[i] ≤ S[i+1] for 0 ≤ i < n-1, possibly repeated.

How to represent it as a *bit vector* in which each original integer is *self-delimited*, using as few as possible bits?

Huge research corpora describing different space/time trade-offs.

- Elias gamma/delta [Elias, TIT 1975]
- Variable Byte [Salomon, Springer 2007]
- Binary Interpolative Coding [Moffat and Stuiver, IRJ 2000]
- Simple-9 [Anh and Moffat, IRJ 2005]
- PForDelta [Zukowski *et al*., ICDE 2006]
- OptPFD [Yan *et al*., WWW 2009]
- Simple-16 [Anh and Moffat, SPE 2010]
- Varint-G8IU [Stepanov *et al*., CIKM 2011]
- Elias-Fano [Vigna, WSDM 2013]
- Partitioned Elias-Fano [Ottaviano and Venturini, SIGIR 2014]

General Problem

Consider a sequence S[0,n) of n *positive* and *monotonically increasing integers*, i.e., S[i] ≤ S[i+1] for 0 ≤ i < n-1, possibly repeated.

How to represent it as a *bit vector* in which each original integer is *self-delimited*, using as few as possible bits?

Huge research corpora describing different space/time trade-offs.

- Elias gamma/delta [Elias, TIT 1975]
- Variable Byte [Salomon, Springer 2007]
- Binary Interpolative Coding [Moffat and Stuiver, IRJ 2000]
- Simple-9 [Anh and Moffat, IRJ 2005]
- PForDelta [Zukowski *et al.*, ICDE 2006]
- OptPFD [Yan *et al.*, WWW 2009]
- Simple-16 [Anh and Moffat, SPE 2010]
- Varint-G8IU [Stepanov *et al.*, CIKM 2011]
- Elias-Fano [Vigna, WSDM 2013]
- Partitioned Elias-Fano [Ottaviano and Venturini, SIGIR 2014]

Dropbox Tech Blog

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/

Improving the performance of full-text search

Adam Faulkner | September 7, 2016

0 13 2

For Firefly, Dropbox full text-search engine, speed has always been a priority.

They *were unable to scale* because of the dimension of their (distributed) inverted index.
Consequence? Query time latencies deteriorate **from 250ms to 1s**.

Dropbox Tech Blog

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/

Improving the performance of full-text search

Adam Faulkner September 7, 2016

0 13 2

For Firefly, Dropbox full text-search engine, speed has always been a priority.

They *were unable to scale* because of the dimension of their (distributed) inverted index.
Consequence? Query time latencies deteriorate **from 250ms to 1s**.

Dropbox Tech Blog

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/

Improving the performance of full-text search

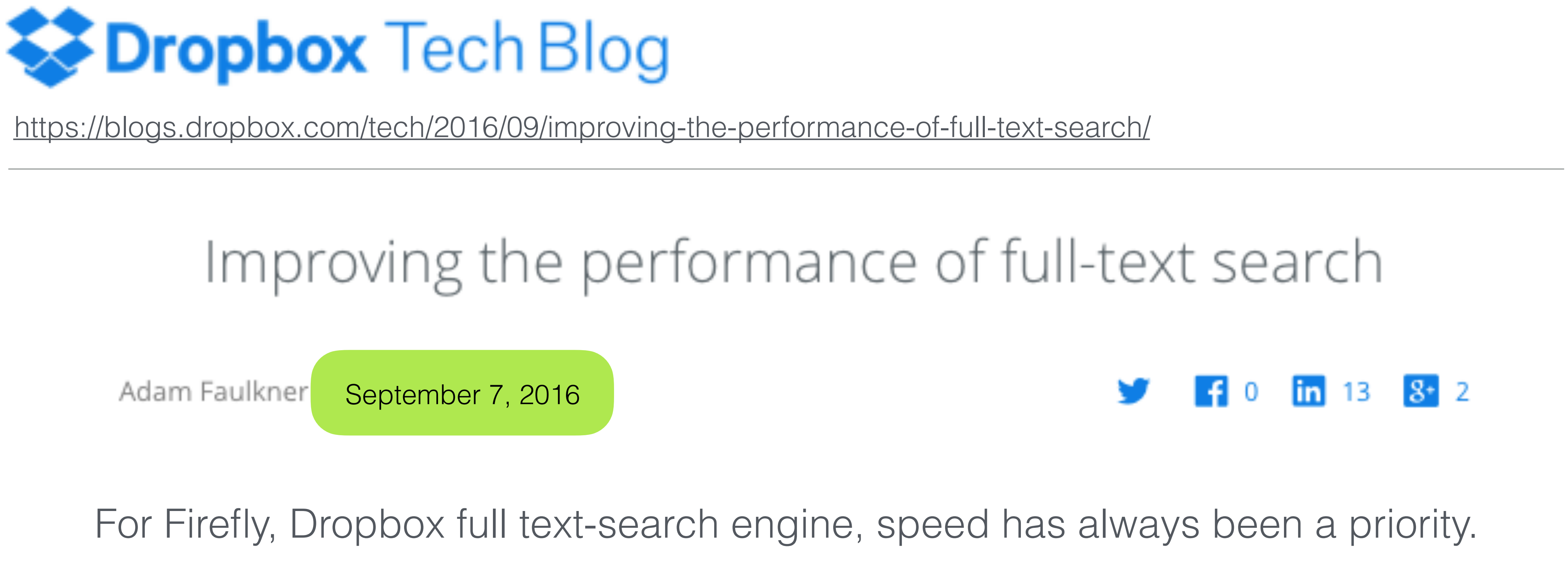
Adam Faulkner September 7, 2016

0 13 2

For Firefly, Dropbox full text-search engine, speed has always been a priority.

They *were unable to scale* because of the dimension of their (distributed) inverted index.
Consequence? Query time latencies deteriorate **from 250ms to 1s**.

Solution?



They *were unable to scale* because of the dimension of their (distributed) inverted index.
Consequence? Query time latencies deteriorate **from 250ms to 1s**.

Solution?

Compress the index to reduce I/O pressure.

Dropbox Tech Blog

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/

Improving the performance of full-text search

Adam Faulkner September 7, 2016

0 13 2

For Firefly, Dropbox full text-search engine, speed has always been a priority.

They *were unable to scale* because of the dimension of their (distributed) inverted index.
Consequence? Query time latencies deteriorate **from 250ms to 1s**.

Solution?

Compress the index to reduce I/O pressure.

Data Structures + Data Compression ➡ Faster Algorithms

Elias-Fano - Genesis

Peter Elias
[1923 - 2001]

Robert Fano
[1917 - 2016]

Robert Fano. *On the number of bits required to implement an associative memory*. Memorandum 61, Computer Structures Group, MIT (1971).

Peter Elias. *Efficient Storage and Retrieval by Content and Address of Static Files*. Journal of the ACM (JACM) 21, 2, 246–260 (1974).

Elias-Fano - Genesis

Peter Elias
[1923 - 2001]

Robert Fano
[1917 - 2016]

Robert Fano. *On the number of bits required to implement an associative memory*. Memorandum 61, Computer Structures Group, MIT (1971).

Peter Elias. *Efficient Storage and Retrieval by Content and Address of Static Files*. Journal of the ACM (JACM) 21, 2, 246–260 (1974).

40 years later!

Sebastiano Vigna. *Quasi-Succinct Indices*.

In Proceedings of the 6-th ACM International Conference on Web Search and Data Mining (WSDM), 83-92 (2013).

Elias-Fano - Encoding example

3	1
4	2
7	3
13	4
14	5
15	6
21	7
43	8

Elias-Fano - Encoding example

	3	1
	4	2
	7	3
	13	4
	14	5
	15	6
	21	7
u =	43	8

Elias-Fano - Encoding example

2

0 0 0 0 1 1	3	1
0 0 0 1 0 0	4	2
0 0 0 1 1 1	7	3
0 0 1 1 0 1	13	4
0 0 1 1 1 0	14	5
0 0 1 1 1 1	15	6
0 1 0 1 0 1	21	7
1 0 1 0 1 1	u = 43	8

Elias-Fano - Encoding example

high	low		
lg n	lg(u/n)		
0 0 0	0 1 1	3	1
0 0 0	1 0 0	4	2
0 0 0	1 1 1	7	3
0 0 1	1 0 1	13	4
0 0 1	1 1 0	14	5
0 0 1	1 1 1	15	6
0 1 0	1 0 1	21	7
1 0 1	0 1 1	u = 43	8

Elias-Fano - Encoding example

high ($\lg n$)	low ($\lg(u/n)$)		
0 0 0	0 1 1	3	1
0 0 0	1 0 0	4	2
0 0 0	1 1 1	7	3
0 0 1	1 0 1	13	4
0 0 1	1 1 0	14	5
0 0 1	1 1 1	15	6
0 1 0	1 0 1	21	7
1 0 1	0 1 1	$u =$ 43	8

Elias-Fano - Encoding example

high lg n	low lg(u/n)		
0 0 0	0 1 1	3	1
0 0 0	1 0 0	4	2
0 0 0	1 1 1	7	3
0 0 1	1 0 1	13	4
0 0 1	1 1 0	14	5
0 0 1	1 1 1	15	6
0 1 0	1 0 1	21	7
1 0 1	0 1 1	u = 43	8

L = 011100111101110111101011

Elias-Fano - Encoding example

	high	low		
	lg n	lg(u/n)		
	0 0 0	0 1 1	3	1
3	0 0 0	1 0 0	4	2
	0 0 0	1 1 1	7	3
	0 0 1	1 0 1	13	4
	0 0 1	1 1 0	14	5
	0 0 1	1 1 1	15	6
	0 1 0	1 0 1	21	7
	1 0 1	0 1 1	u = 43	8

L = 011100111101110111101011

Elias-Fano - Encoding example

	high (lg n)	low (lg(u/n))		
	0 0 0	0 1 1	3	1
3	0 0 0	1 0 0	4	2
	0 0 0	1 1 1	7	3
	0 0 1	1 0 1	13	4
3	0 0 1	1 1 0	14	5
	0 0 1	1 1 1	15	6
	0 1 0	1 0 1	21	7
	1 0 1	0 1 1	u = 43	8

L = 011100111101110111101011

Elias-Fano - Encoding example

	high lg n	low lg(u/n)		
	0 0 0	0 1 1	3	1
3	0 0 0	1 0 0	4	2
	0 0 0	1 1 1	7	3
	0 0 1	1 0 1	13	4
3	0 0 1	1 1 0	14	5
	0 0 1	1 1 1	15	6
1	0 1 0	1 0 1	21	7
	1 0 1	0 1 1	u = 43	8

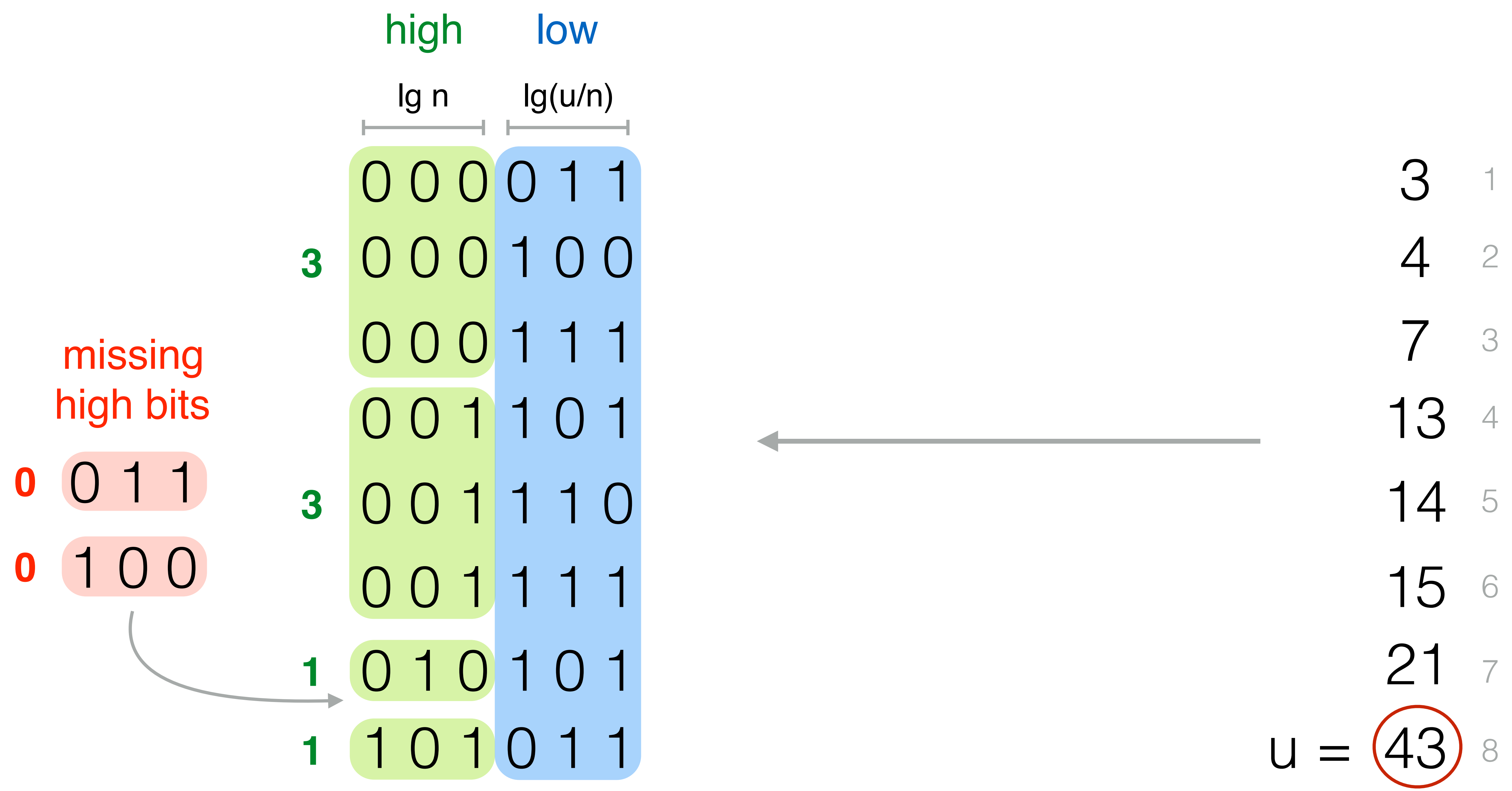
L = 011100111101110111101011

Elias-Fano - Encoding example

	high (lg n)	low (lg(u/n))		
	0 0 0	0 1 1	3	1
3	0 0 0	1 0 0	4	2
	0 0 0	1 1 1	7	3
	0 0 1	1 0 1	13	4
3	0 0 1	1 1 0	14	5
	0 0 1	1 1 1	15	6
1	0 1 0	1 0 1	21	7
1	1 0 1	0 1 1	u = 43	8

L = 011100111101110111101011

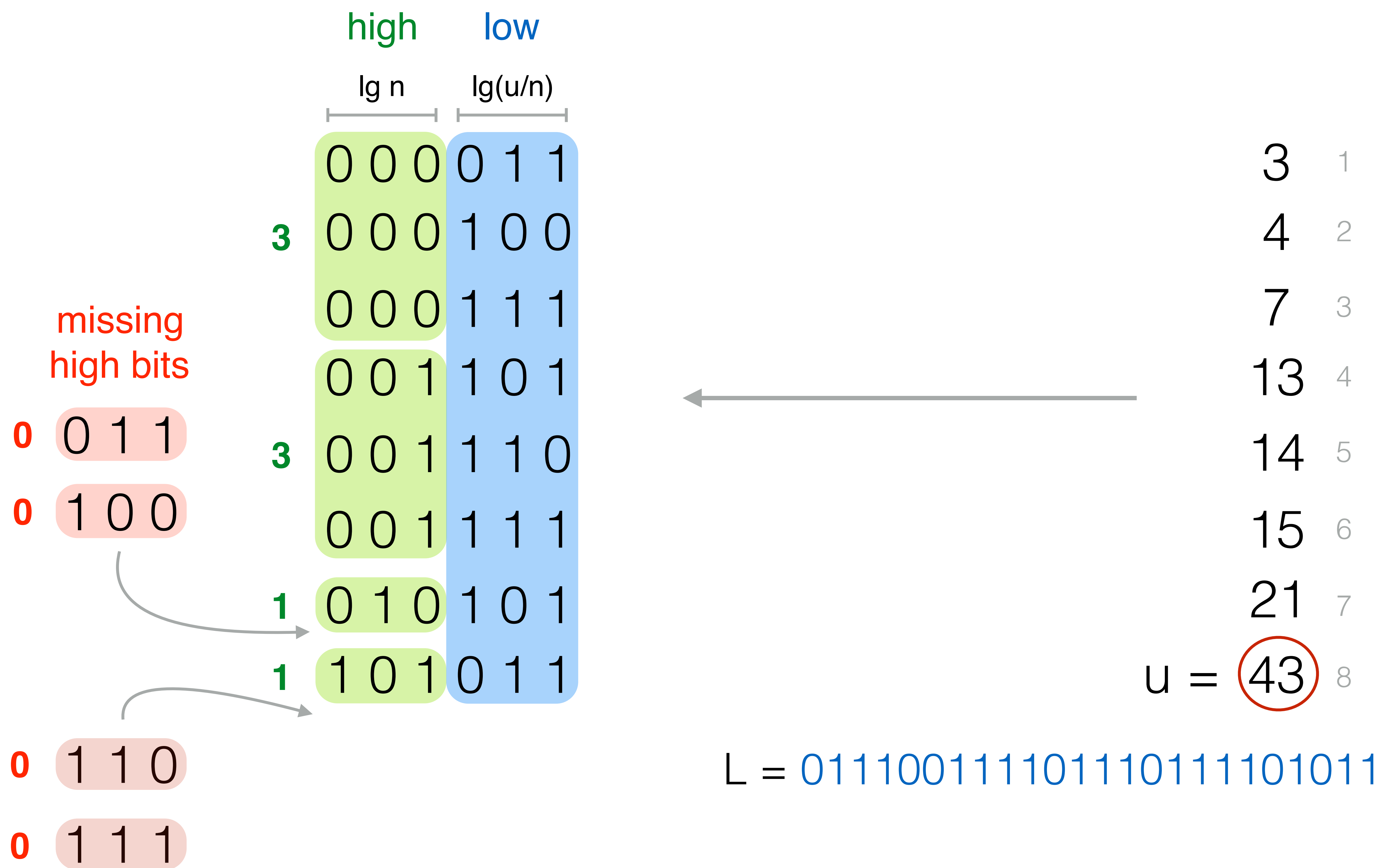
Elias-Fano - Encoding example



L = 011100111101110111101011

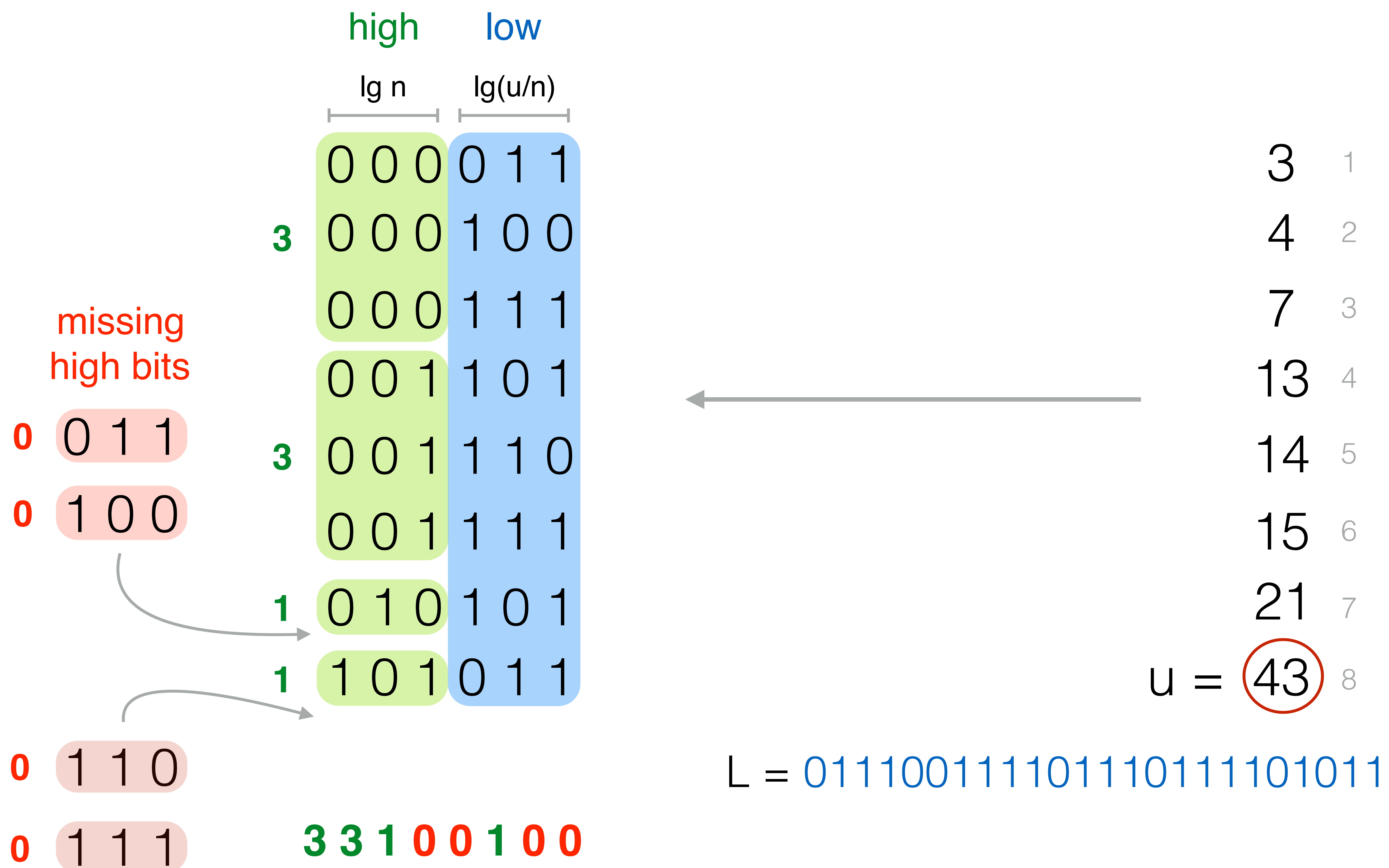
Elias-Fano - Encoding example

2

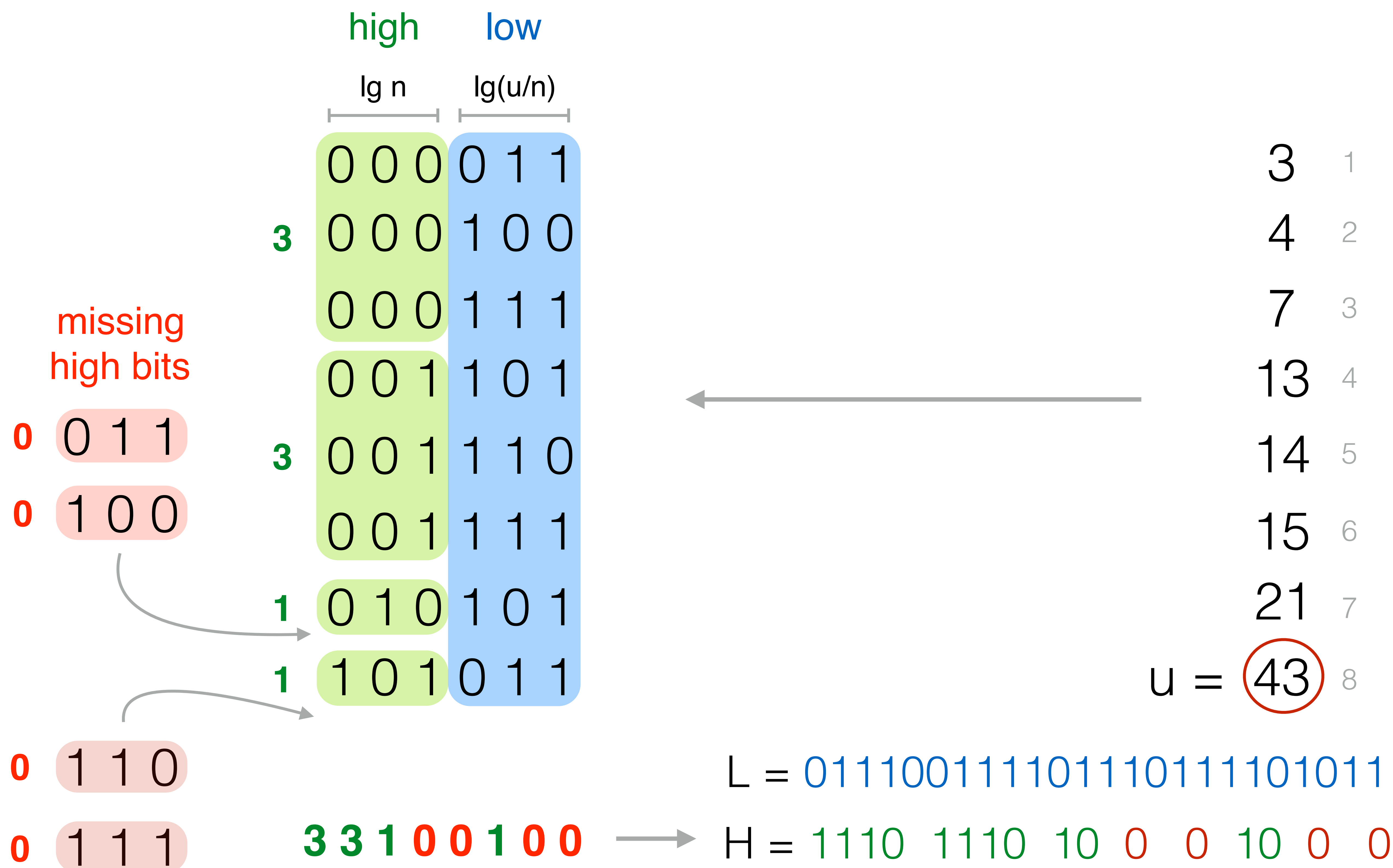


L = 011100111101110111101011

Elias-Fano - Encoding example



Elias-Fano - Encoding example



L = 011100111101110111101011

3 3 1 0 0 1 0 0 → H = 1110 1110 10 0 0 10 0 0

Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \ \text{ bits}$$

Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \ \text{ bits}$$

$\mathcal{X}$ is the set of all monotone sequence
of length n drawn from a universe u.

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

$\mathcal{X}$ is the set of all monotone sequence
of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \quad \text{bits}$$

$\mathcal{X}$ is the set of all monotone sequence of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lceil \lg \binom{u+n}{n} \right\rceil \approx n \lg \frac{u+n}{n}$$

$$EF(S[0,n)) = \boxed{n\left\lceil \lg \frac{u}{n} \right\rceil + 2n} \text{ bits}$$

$\mathcal{X}$ is the set of all monotone sequence of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lceil \lg \binom{u+n}{n} \right\rceil \approx \boxed{n \lg \frac{u+n}{n}}$$

Elias-Fano - Properties

$$EF(S[0,n)) = \boxed{n \left\lceil \lg \frac{u}{n} \right\rceil + 2n} \text{ bits}$$

(less than half a bit away [Elias, JACM 1974])

$\mathcal{X}$ is the set of all monotone sequence of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lceil \lg \binom{u+n}{n} \right\rceil \approx \boxed{n \lg \frac{u+n}{n}}$$

$$\mathrm{EF}(S[0,n)) = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

(less than half a bit away [Elias, JACM 1974])

$\mathcal{X}$ is the set of all monotone sequence of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lceil \lg \binom{u+n}{n} \right\rceil \approx n \lg \frac{u+n}{n}$$

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

$\mathcal{X}$ is the set of all monotone sequence
of length n drawn from a universe u.

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lceil \lg \binom{u+n}{n} \right\rceil \approx n \lg \frac{u+n}{n}$$

Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

Elias-Fano - Properties

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

$$\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

Elias-Fano - Properties

$$EF(S[0,n)) = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

$$\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

Elias-Fano - Properties

$$\mathrm{EF}(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

$$\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…

$$EF(S[0,n)) = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

$$\mathsf{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\mathsf{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…

need o(n) bits more to support *fast*
rank/select primitives on bitvector H

$$\mathrm{EF}(S[0,n)) = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

access to each S[i] in O(1) worst-case

$$\mathsf{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\mathsf{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…

need o(n) bits more to support *fast*
rank/select primitives on bitvector H

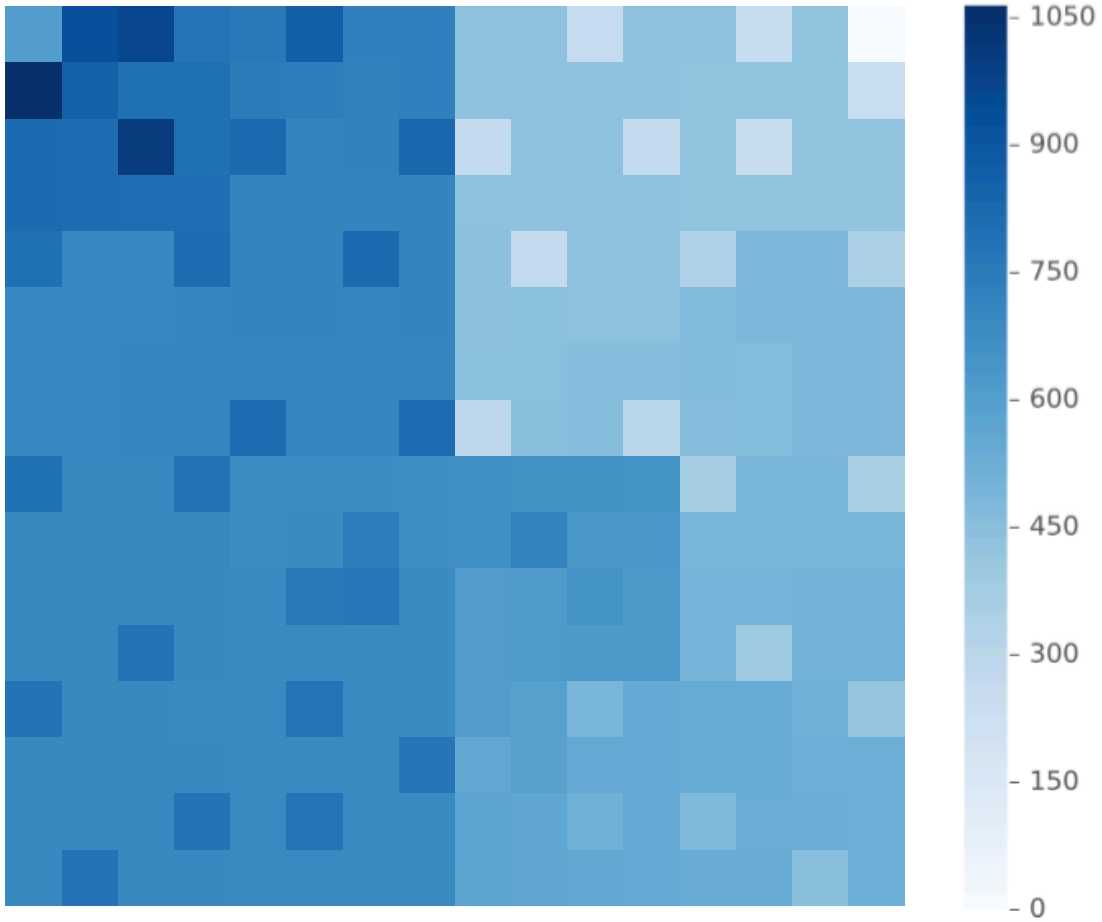
Clustered Elias-Fano Indexes

Every encoder represents each sequence *individually*.

No exploitation of redundancy.

Every encoder represents each sequence *individually.*

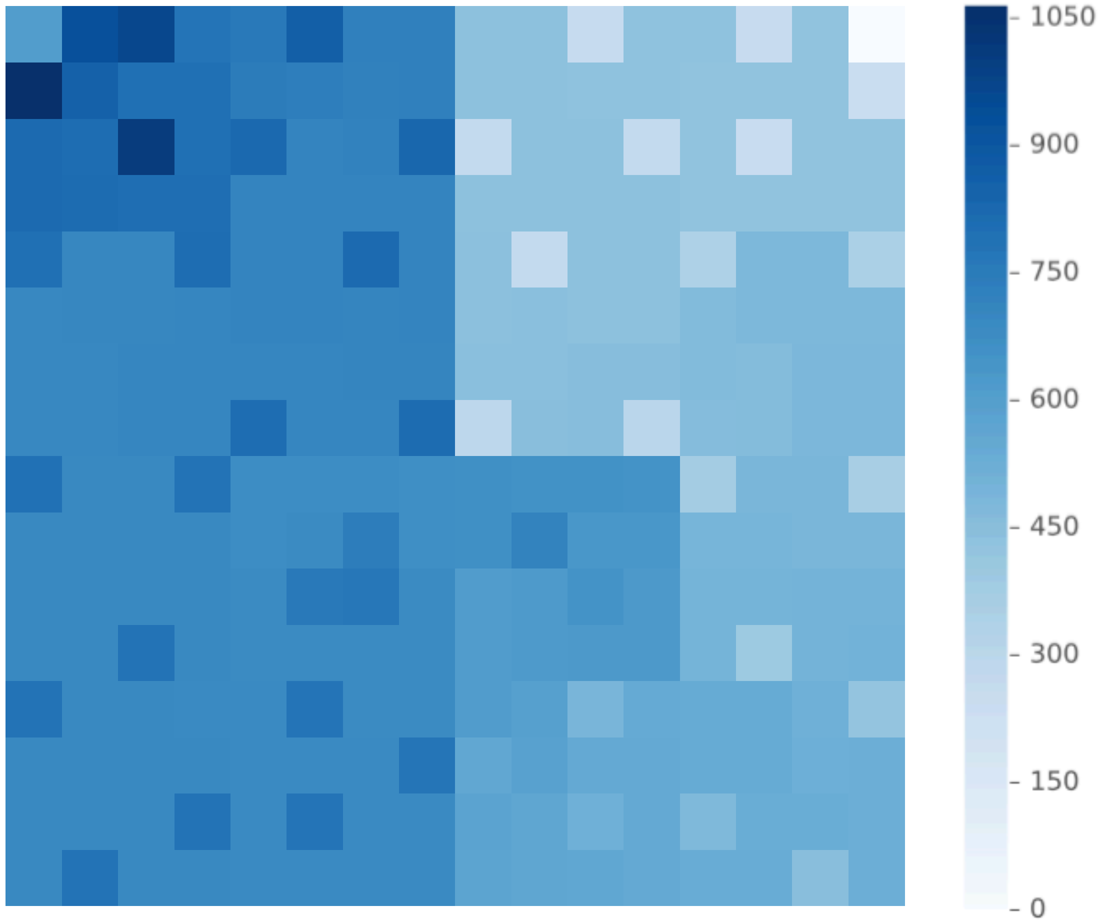
No exploitation of redundancy.



Clustered Elias-Fano Indexes

Every encoder represents each sequence *individually.*

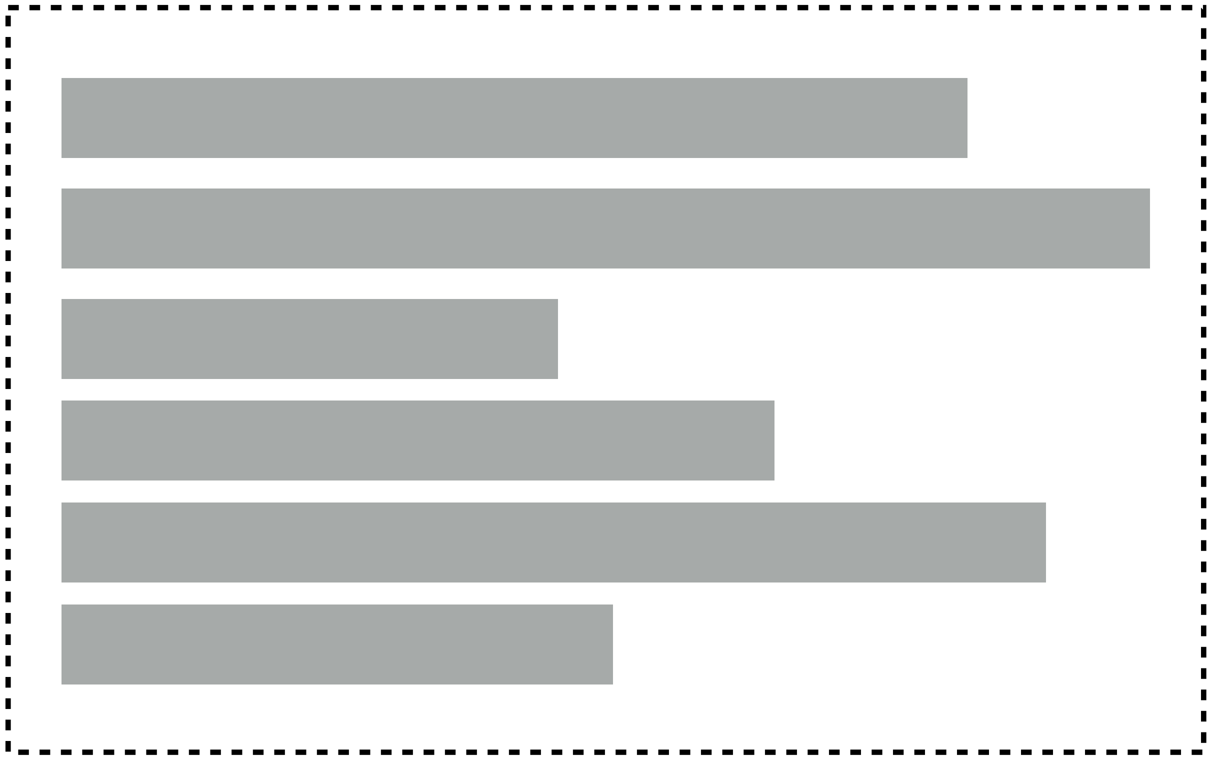
No exploitation of redundancy.



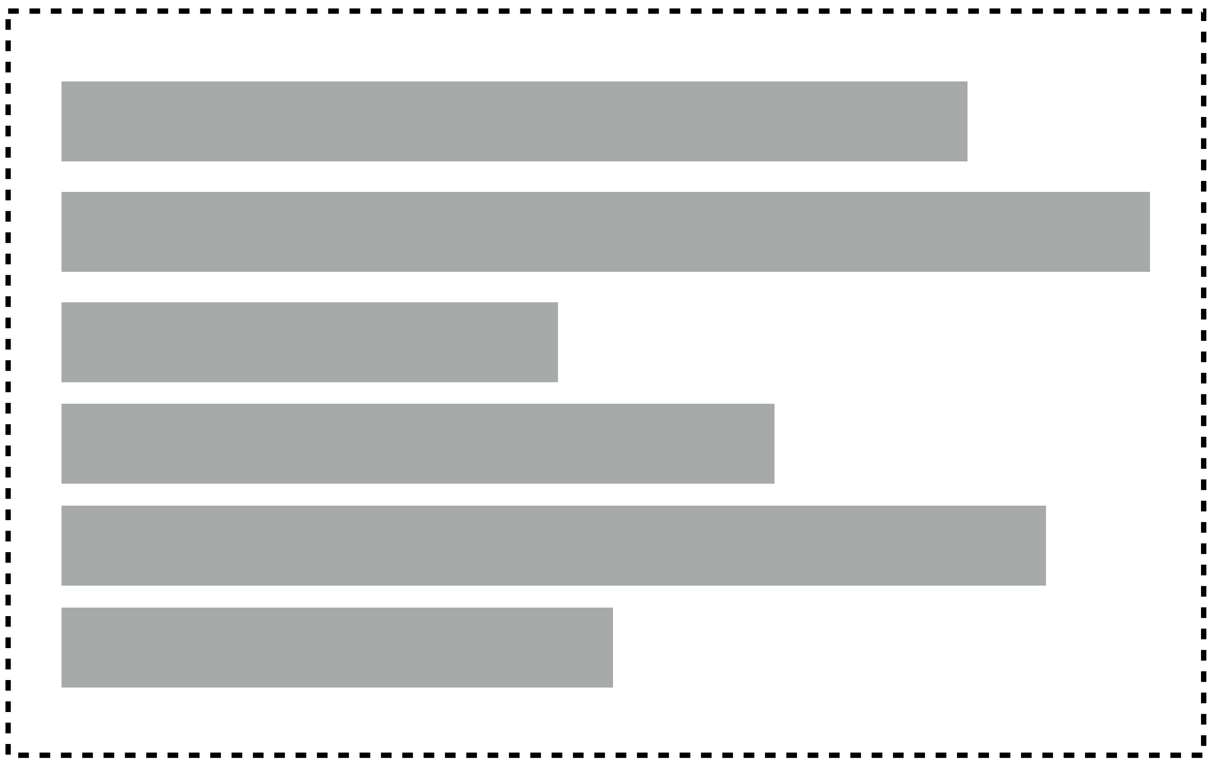
Idea: encode **clusters** of posting lists.

Clustered Elias-Fano Indexes

cluster of posting lists



cluster of posting lists



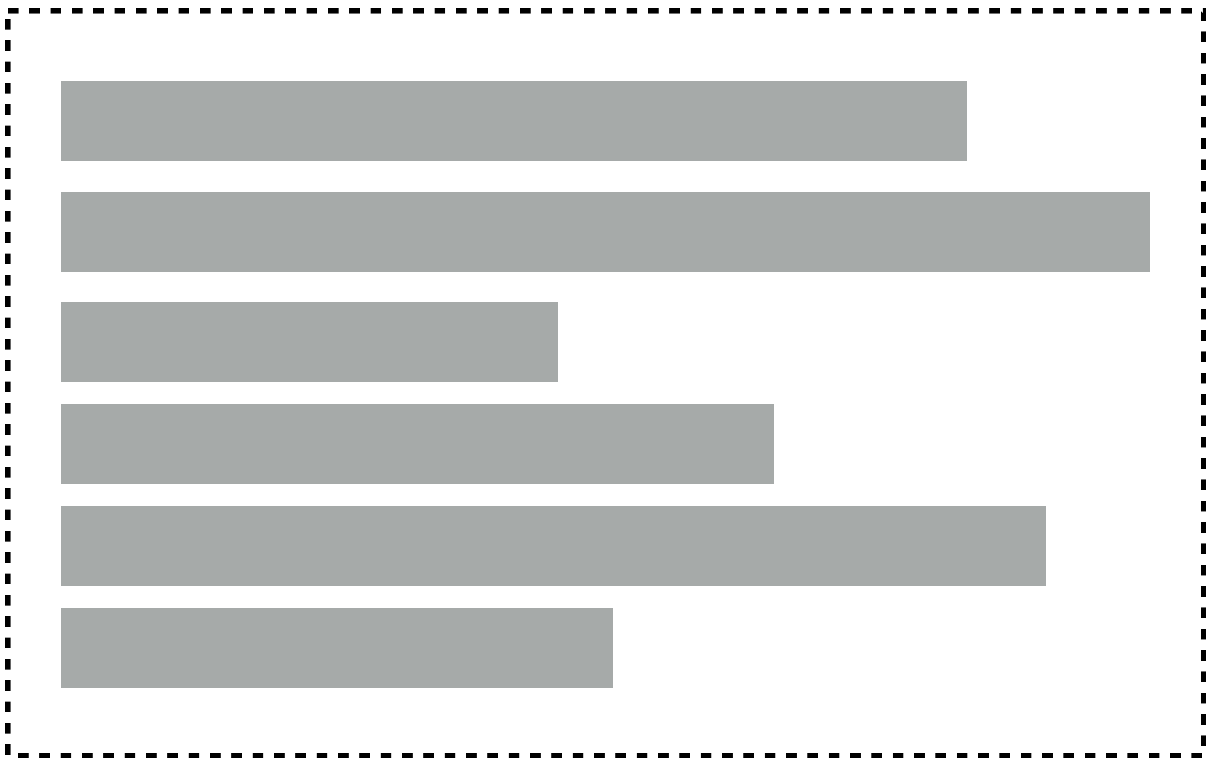
unbounded universe

lg u bits

Clustered Elias-Fano Indexes

Clustered Elias-Fano Indexes 2

cluster of posting lists



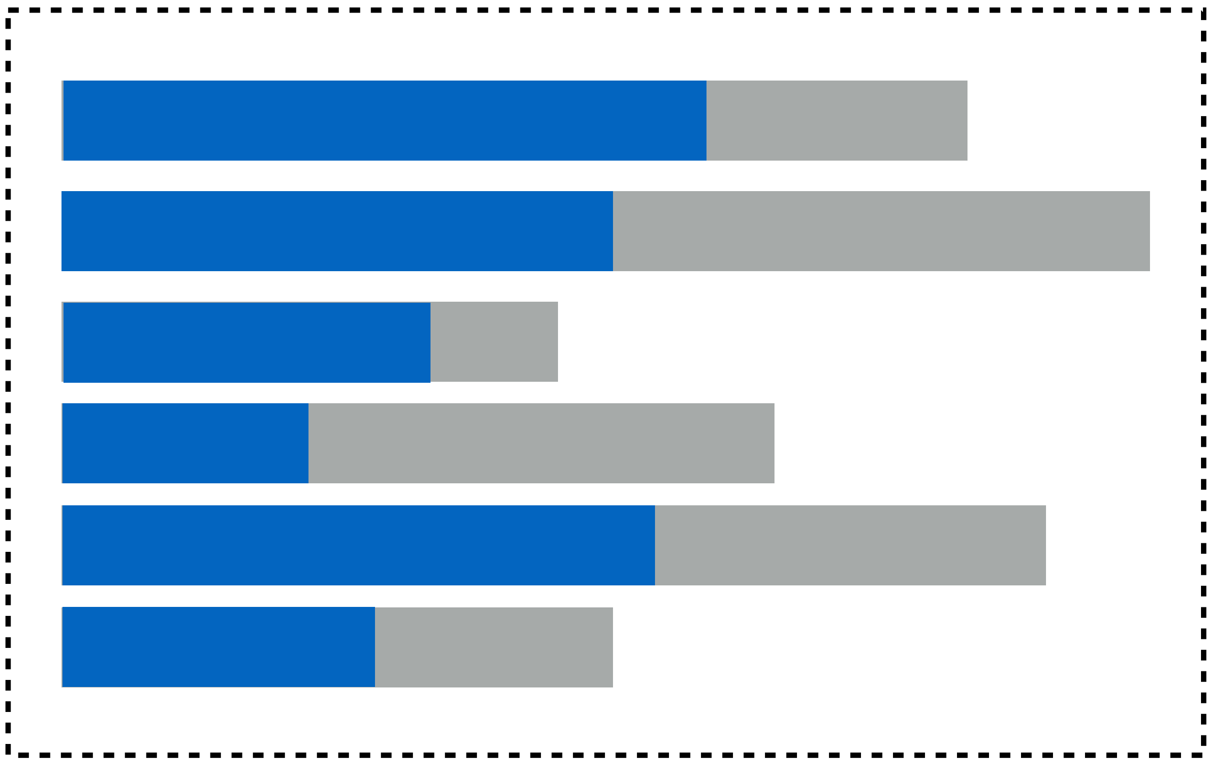
unbounded universe

lg u bits

reference list

R

cluster of posting lists



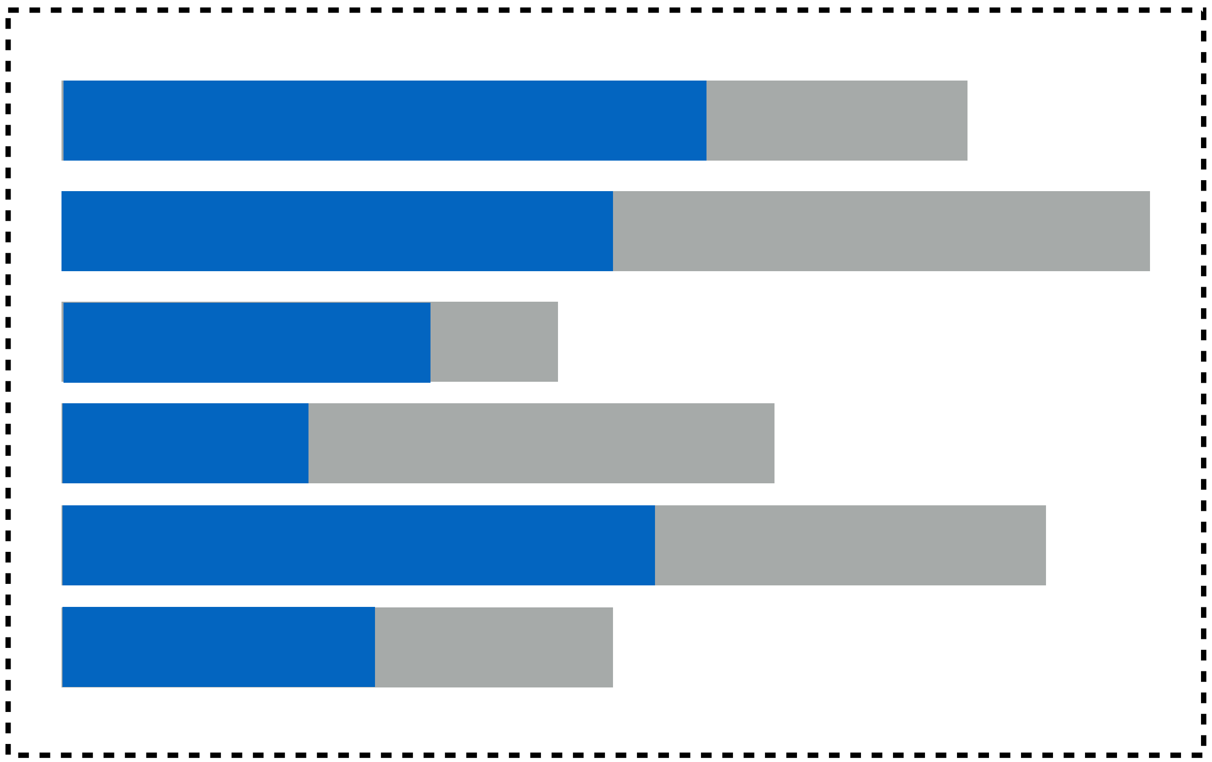
unbounded universe

lg u bits

reference list

R

cluster of posting lists



unbounded universe

lg u bits

VS

reference list

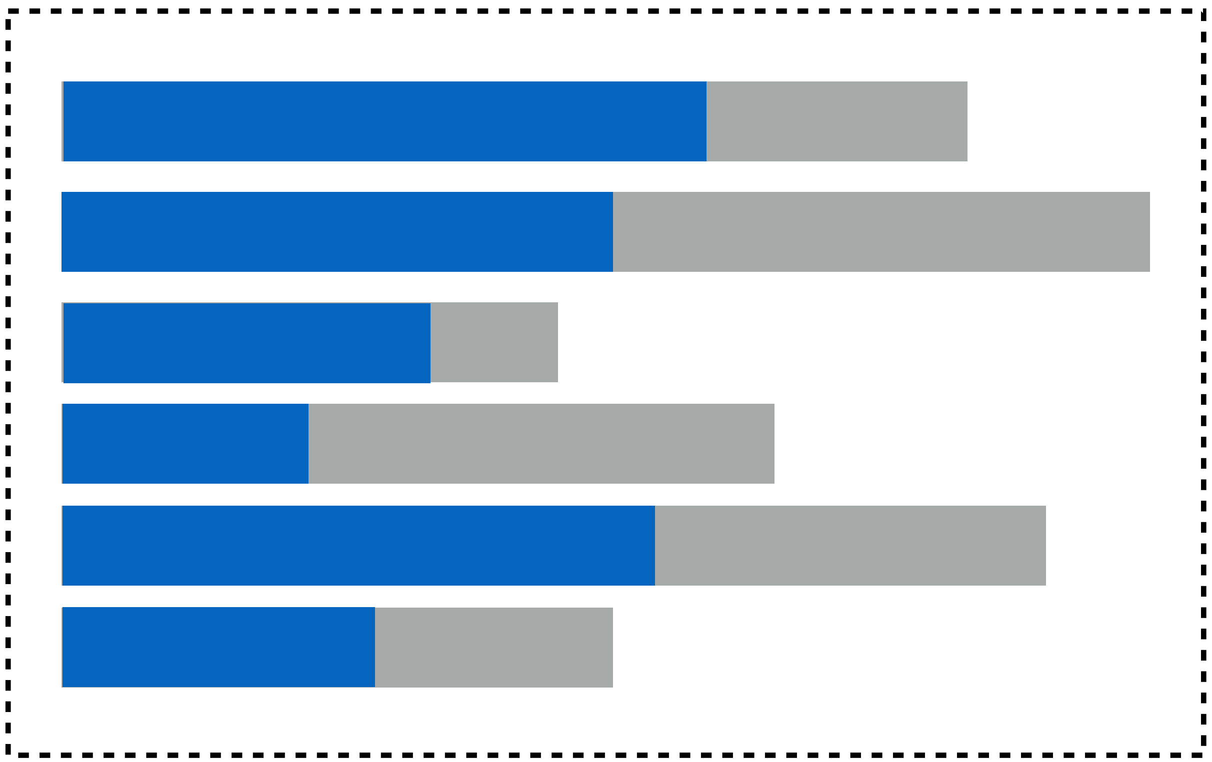
R

R << u

lg R bits

Clustered Elias-Fano Indexes

cluster of posting lists



reference list

R

unbounded universe

R << u

lg u bits

VS

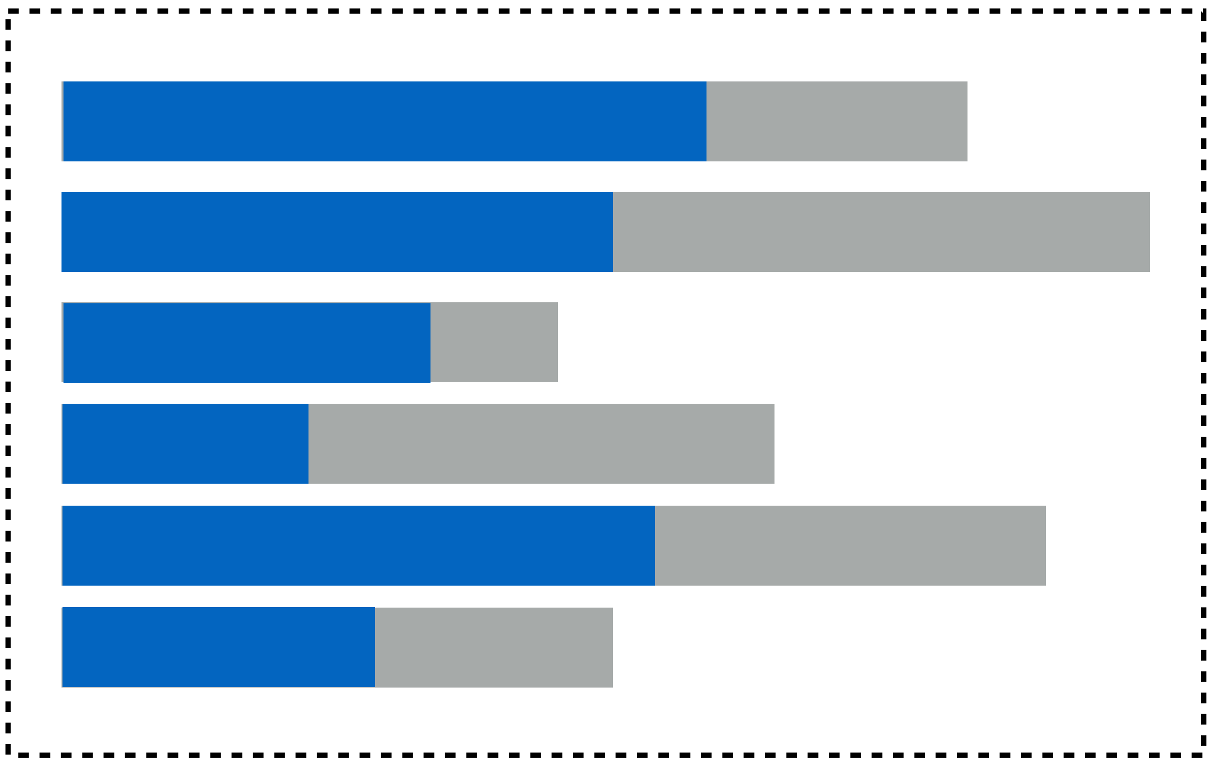
lg R bits

Problems

1. how to build clusters
2. how to synthesise the reference list

Clustered Elias-Fano Indexes 2

cluster of posting lists



unbounded universe

lg u bits

VS

reference list

R << u

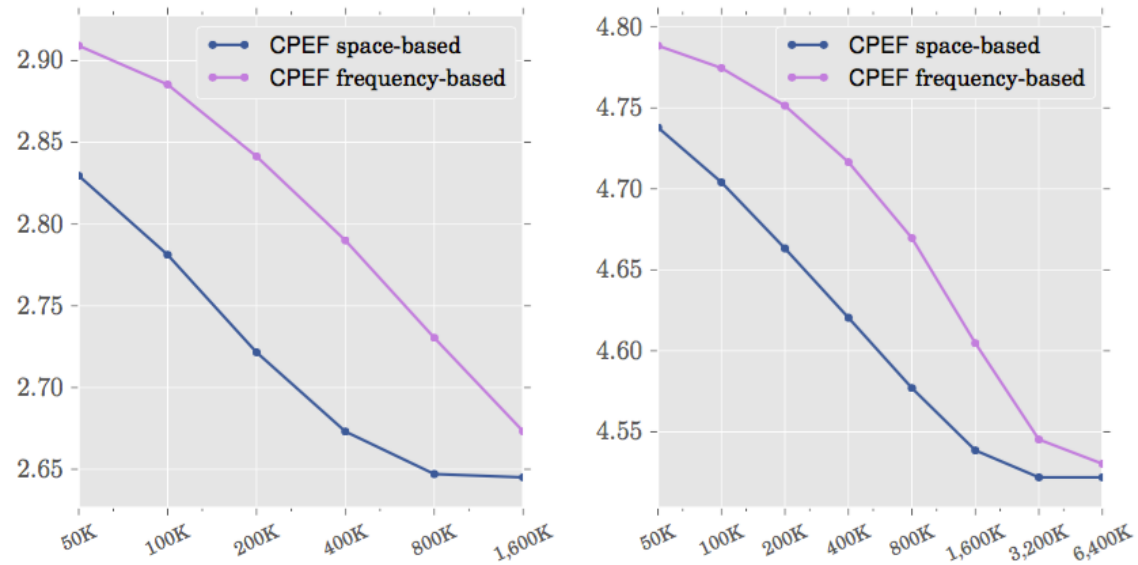
lg R bits

Problems

1. how to build clusters
2. how to synthesise the reference list

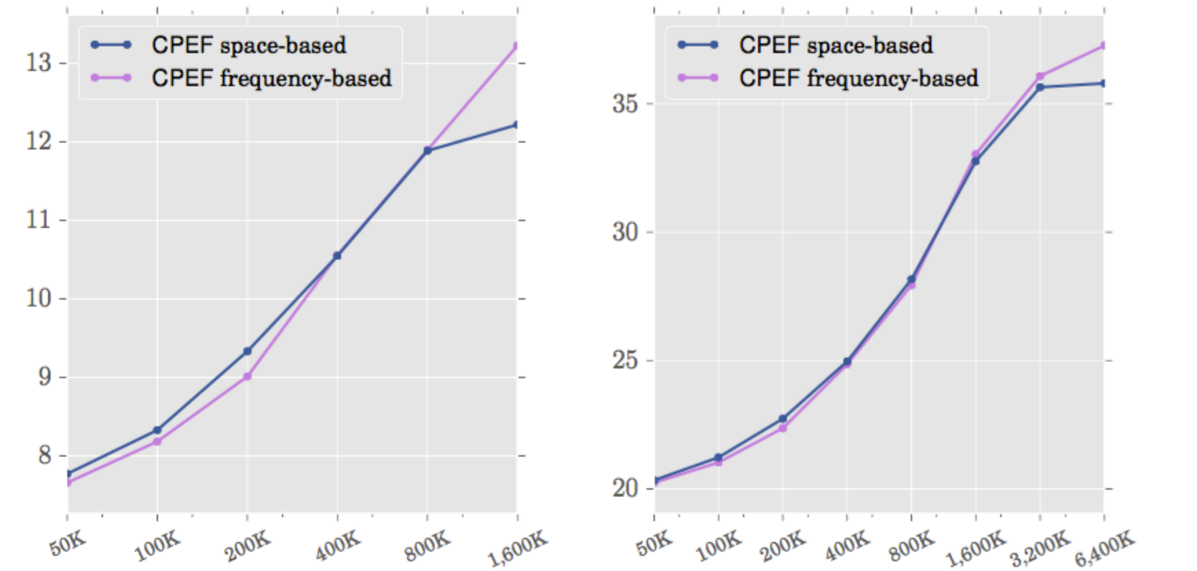
NP-hard problem
already for a simplified formulation.

Time VS Space tradeoffs by varying reference size



(a) Gov2 (b) ClueWeb09

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.



(a) Gov2 (b) ClueWeb09

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

	MIN	MID	MAX
PEF	2.94 (+5.60%)	2.94 (+7.91%)	2.94 (+10.95%)
CPEF	2.78	2.72	2.65
BIC	2.80 (+0.53%)	2.80 (+2.74%)	2.80 (+5.63%)

(a) Gov2

	MIN	MID	MAX
PEF	4.80 (+2.13%)	4.80 (+3.98%)	4.80 (+6.25%)
CPEF	4.70	4.62	4.52
BIC	4.27 (−9.22%)	4.27 (−7.58%)	4.27 (−5.56%)

(b) ClueWeb09

Table 2: Bits per posting in selected trade-off points.

		MIN	MID	MAX
TREC 05	PEF	14.6 (−17.5%)	14.6 (−29.0%)	14.6 (−49.7%)
	CPEF	17.7	20.6	29.1
	BIC	41.1 (+131.9%)	41.1 (+99.5%)	41.1 (+41.3%)
TREC 06	PEF	17.7 (−16.6%)	17.7 (−29.1%)	17.7 (−50.3%)
	CPEF	21.2	25.0	35.6
	BIC	55.1 (+159.7%)	55.1 (+120.8%)	55.1 (+54.7%)

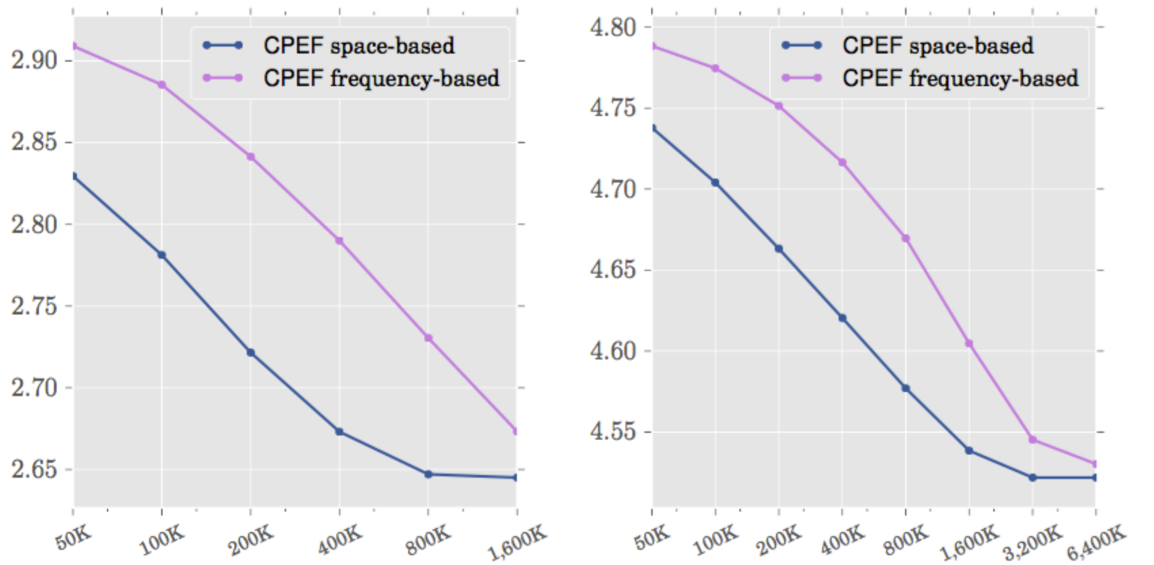
(a) ClueWeb09

		MIN	MID	MAX
TREC 05	PEF	3.7 (−30.4%)	3.7 (−37.5%)	3.7 (−52.1%)
	CPEF	5.3	5.9	7.8
	BIC	10.5 (+96.2%)	10.5 (+76.2%)	10.5 (+35.0%)
TREC 06	PEF	6.1 (−27.4%)	6.1 (−35.2%)	6.1 (−49.1%)
	CPEF	8.3	9.3	11.9
	BIC	18.5 (+122.6%)	18.5 (+98.6%)	18.5 (+56.0%)

(b) Gov2

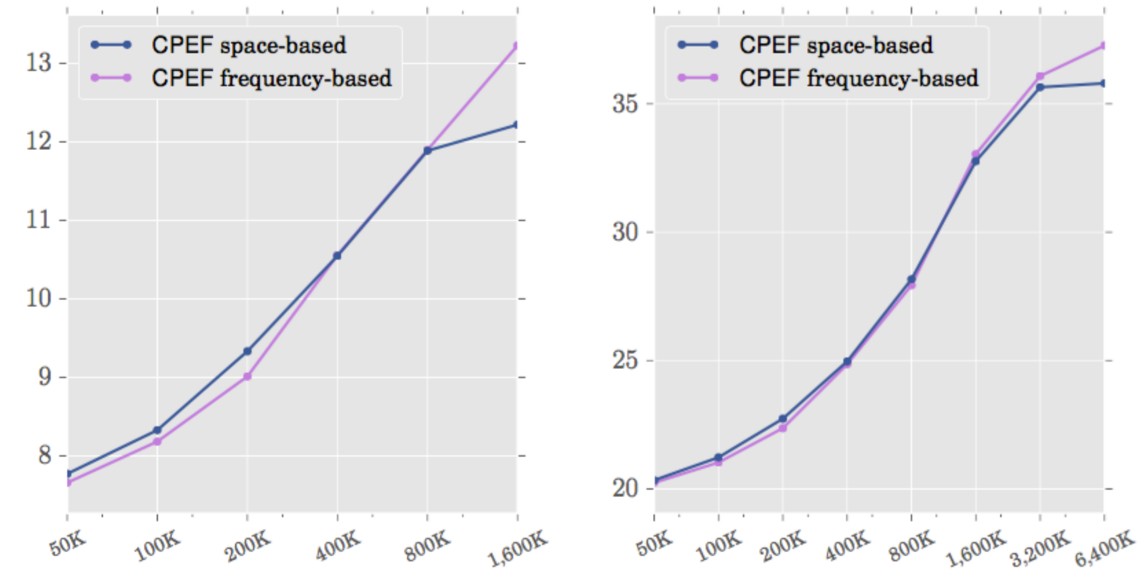
Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

Time VS Space tradeoffs by varying reference size



(a) Gov2 (b) ClueWeb09

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.



(a) Gov2 (b) ClueWeb09

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

	MIN	MID	MAX
PEF	2.94 (+5.60%)	2.94 (+7.91%)	2.94 (+10.95%)
CPEF	2.78	2.72	2.65
BIC	2.80 (+0.53%)	2.80 (+2.74%)	2.80 (+5.63%)

(a) Gov2

	MIN	MID	MAX
PEF	4.80 (+2.13%)	4.80 (+3.98%)	4.80 (+6.25%)
CPEF	4.70	4.62	4.52
BIC	4.27 (−9.22%)	4.27 (−7.58%)	4.27 (−5.56%)

(b) ClueWeb09

Table 2: Bits per posting in selected trade-off points.

		MIN	MID	MAX
TREC 05	PEF	14.6 (−17.5%)	14.6 (−29.0%)	14.6 (−49.7%)
	CPEF	17.7	20.6	29.1
	BIC	41.1 (+131.9%)	41.1 (+99.5%)	41.1 (+41.3%)
TREC 06	PEF	17.7 (−16.6%)	17.7 (−29.1%)	17.7 (−50.3%)
	CPEF	21.2	25.0	35.6
	BIC	55.1 (+159.7%)	55.1 (+120.8%)	55.1 (+54.7%)

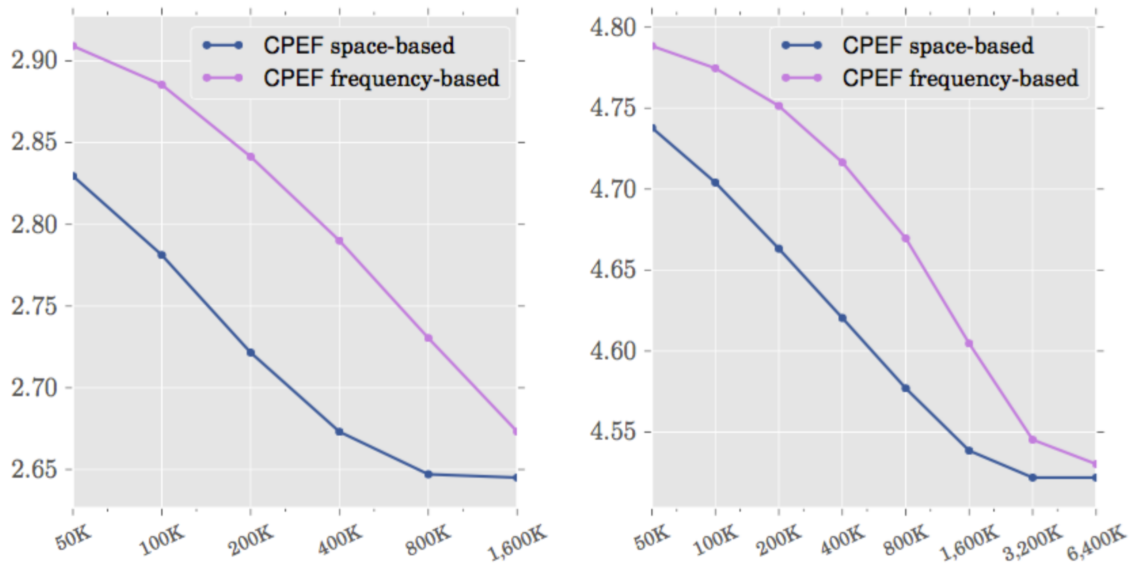
(a) ClueWeb09

		MIN	MID	MAX
TREC 05	PEF	3.7 (−30.4%)	3.7 (−37.5%)	3.7 (−52.1%)
	CPEF	5.3	5.9	7.8
	BIC	10.5 (+96.2%)	10.5 (+76.2%)	10.5 (+35.0%)
TREC 06	PEF	6.1 (−27.4%)	6.1 (−35.2%)	6.1 (−49.1%)
	CPEF	8.3	9.3	11.9
	BIC	18.5 (+122.6%)	18.5 (+98.6%)	18.5 (+56.0%)

(b) Gov2

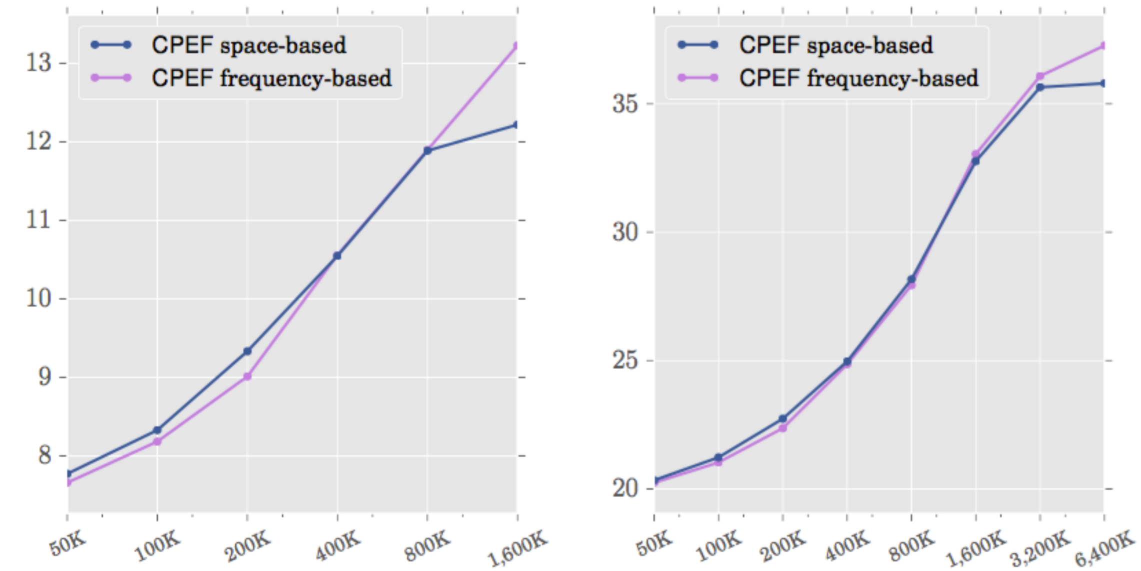
Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

Time VS Space tradeoffs by varying reference size



(a) Gov2 (b) ClueWeb09

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.



(a) Gov2 (b) ClueWeb09

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

	MIN	MID	MAX
PEF	2.94 (+5.60%)	2.94 (+7.91%)	2.94 (+10.95%)
CPEF	2.78	2.72	2.65
BIC	2.80 (+0.53%)	2.80 (+2.74%)	2.80 (+5.63%)

(a) Gov2

	MIN	MID	MAX
PEF	4.80 (+2.13%)	4.80 (+3.98%)	4.80 (+6.25%)
CPEF	4.70	4.62	4.52
BIC	4.27 (−9.22%)	4.27 (−7.58%)	4.27 (−5.56%)

(b) ClueWeb09

Table 2: Bits per posting in selected trade-off points.

Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

		MIN	MID	MAX
TREC 05	PEF	14.6 (−17.5%)	14.6 (−29.0%)	14.6 (−49.7%)
	CPEF	17.7	20.6	29.1
	BIC	41.1 (+131.9%)	41.1 (+99.5%)	41.1 (+41.3%)
TREC 06	PEF	17.7 (−16.6%)	17.7 (−29.1%)	17.7 (−50.3%)
	CPEF	21.2	25.0	35.6
	BIC	55.1 (+159.7%)	55.1 (+120.8%)	55.1 (+54.7%)

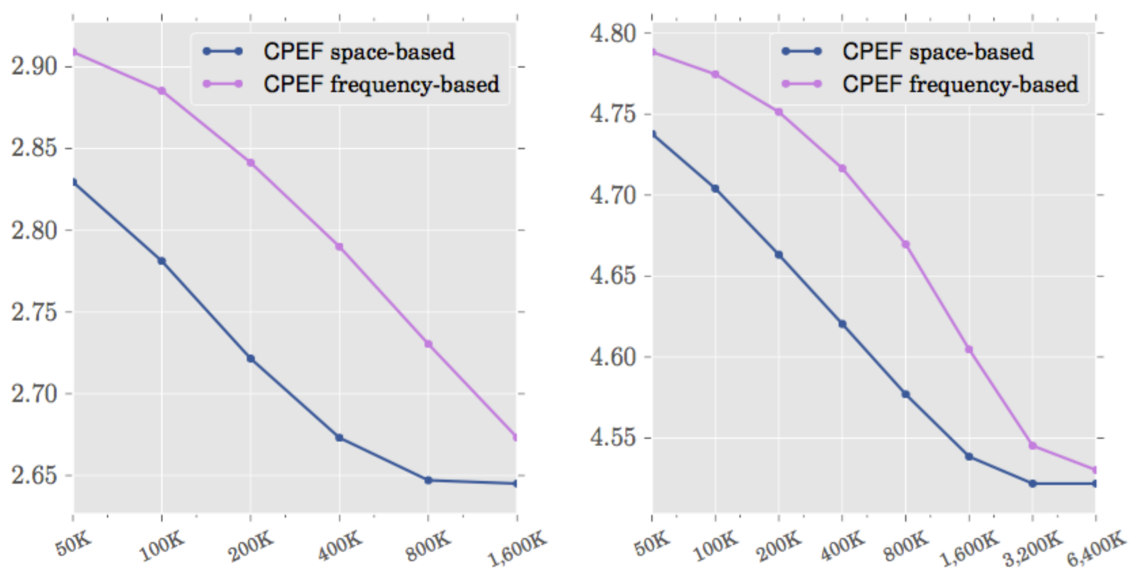
(a) ClueWeb09

		MIN	MID	MAX
TREC 05	PEF	3.7 (−30.4%)	3.7 (−37.5%)	3.7 (−52.1%)
	CPEF	5.3	5.9	7.8
	BIC	10.5 (+96.2%)	10.5 (+76.2%)	10.5 (+35.0%)
TREC 06	PEF	6.1 (−27.4%)	6.1 (−35.2%)	6.1 (−49.1%)
	CPEF	8.3	9.3	11.9
	BIC	18.5 (+122.6%)	18.5 (+98.6%)	18.5 (+56.0%)

(b) Gov2

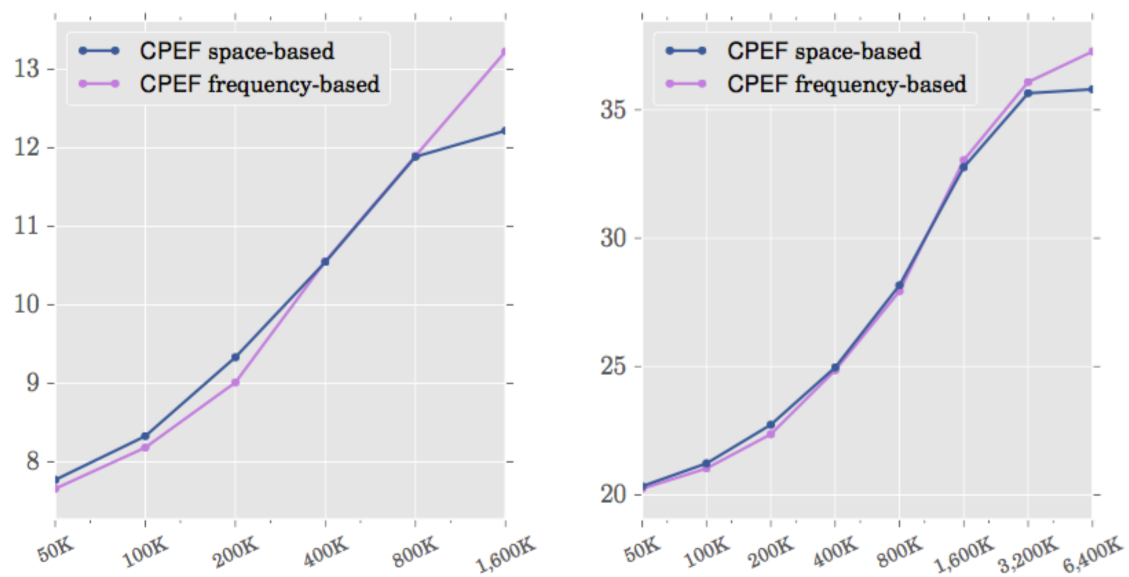
Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

Time VS Space tradeoffs by varying reference size



(a) Gov2 (b) ClueWeb09

Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.



(a) Gov2 (b) ClueWeb09

Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

	MIN	MID	MAX
PEF	2.94 (+5.60%)	2.94 (+7.91%)	2.94 (+10.95%)
CPEF	2.78	2.72	2.65
BIC	2.80 (+0.53%)	2.80 (+2.74%)	2.80 (+5.63%)

(a) Gov2

	MIN	MID	MAX
PEF	4.80 (+2.13%)	4.80 (+3.98%)	4.80 (+6.25%)
CPEF	4.70	4.62	4.52
BIC	4.27 (−9.22%)	4.27 (−7.58%)	4.27 (−5.56%)

(b) ClueWeb09

Table 2: Bits per posting in selected trade-off points.

Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

		MIN	MID	MAX
TREC 05	PEF	14.6 (−17.5%)	14.6 (−29.0%)	14.6 (−49.7%)
	CPEF	17.7	20.6	29.1
	BIC	41.1 (+131.9%)	41.1 (+99.5%)	41.1 (+41.3%)
TREC 06	PEF	17.7 (−16.6%)	17.7 (−29.1%)	17.7 (−50.3%)
	CPEF	21.2	25.0	35.6
	BIC	55.1 (+159.7%)	55.1 (+120.8%)	55.1 (+54.7%)

(a) ClueWeb09

		MIN	MID	MAX
TREC 05	PEF	3.7 (−30.4%)	3.7 (−37.5%)	3.7 (−52.1%)
	CPEF	5.3	5.9	7.8
	BIC	10.5 (+96.2%)	10.5 (+76.2%)	10.5 (+35.0%)
TREC 06	PEF	6.1 (−27.4%)	6.1 (−35.2%)	6.1 (−49.1%)
	CPEF	8.3	9.3	11.9
	BIC	18.5 (+122.6%)	18.5 (+98.6%)	18.5 (+56.0%)

(b) Gov2

Table 3: Timings in milliseconds for AND queries on ClueWeb09 and Gov2, using query sets TREC 05 and TREC 05. In parentheses we show the relative percentage against CPEF.

Time VS Space tradeoffs by varying reference size

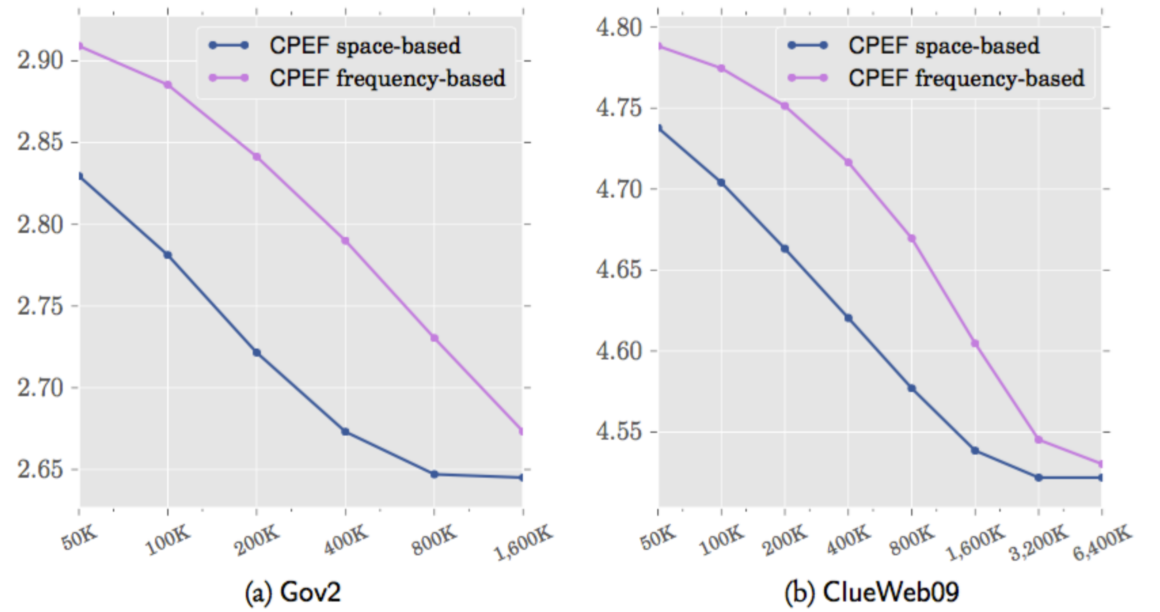


Figure 2: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

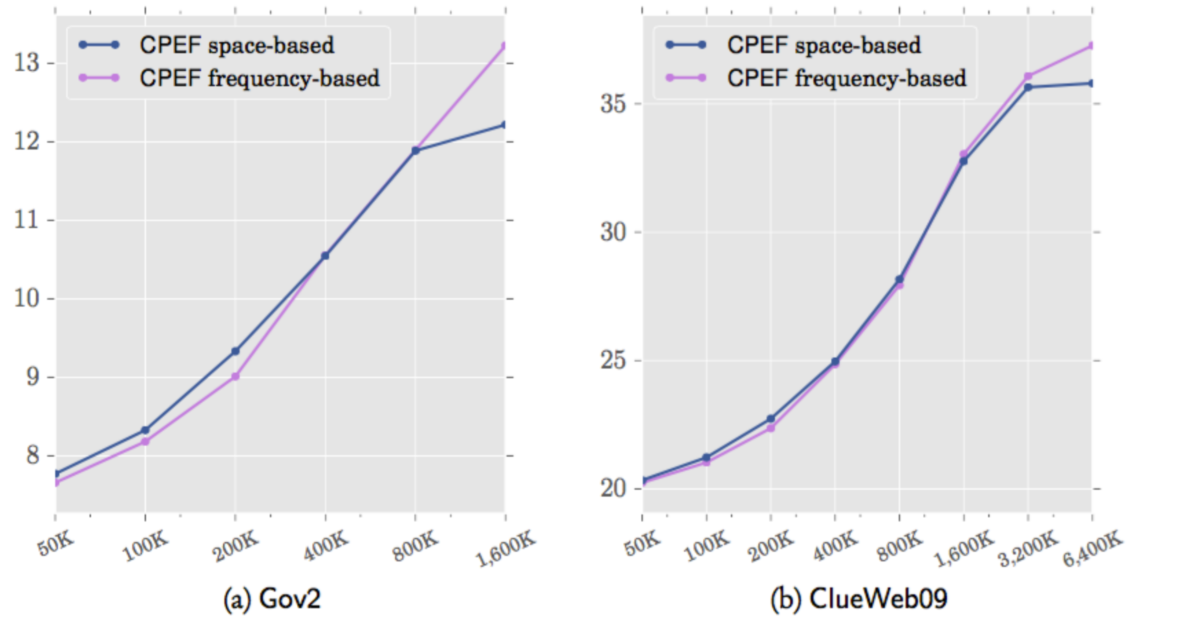


Figure 3: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

(a) Gov2

	MIN		MID		MAX	
PEF	2.94	(+5.60%)	2.94	(+7.91%)	2.94	(+10.95%)
CPEF	2.78		2.72		2.65	
BIC	2.80	(+0.53%)	2.80	(+2.74%)	2.80	(+5.63%)

(b) ClueWeb09

	MIN		MID		MAX	
PEF	4.80	(+2.13%)	4.80	(+3.98%)	4.80	(+6.25%)
CPEF	4.70		4.62		4.52	
BIC	4.27	(−9.22%)	4.27	(−7.58%)	4.27	(−5.56%)

Table 2: Bits per posting in selected trade-off points.

Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

(a) ClueWeb09

		MIN		MID		MAX	
TREC 05	PEF	14.6	(−17.5%)	14.6	(−29.0%)	14.6	(−49.7%)
	CPEF	17.7		20.6		29.1	
	BIC	41.1	(+131.9%)	41.1	(+99.5%)	41.1	(+41.3%)
TREC 06	PEF	17.7	(−16.6%)	17.7	(−29.1%)	17.7	(−50.3%)
	CPEF	21.2		25.0		35.6	
	BIC	55.1	(+159.7%)	55.1	(+120.8%)	55.1	(+54.7%)

(b) Gov2

		MIN		MID		MAX	
TREC 05	PEF	3.7	(−30.4%)	3.7	(−37.5%)	3.7	(−52.1%)
	CPEF	5.3		5.9		7.8	
	BIC	10.5	(+96.2%)	10.5	(+76.2%)	10.5	(+35.0%)
TREC 06	PEF	6.1	(−27.4%)	6.1	(−35.2%)	6.1	(−49.1%)
	CPEF	8.3		9.3		11.9	
	BIC	18.5	(+122.6%)	18.5	(+98.6%)	18.5	(+56.0%)

Much faster than BIC (103% on average)
Slightly slower than PEF (20% on average)

Elias-Fano matches the
information theoretic minimum.

n lg(u/n) + 2n + o(n) bits

Elias-Fano matches the
information theoretic minimum.

n lg(u/n) + 2n + o(n) bits

- **O(1)** random access
- **O(lg(u/n))** predecessor/successor

Integer Data Structures

Elias-Fano matches the
information theoretic minimum.

$n \lg(u/n) + 2n + o(n)$ bits

- **O(1)** random access
- **O(lg(u/n))** predecessor/successor

Static succinct data structure.

NO dynamic updates.

Elias-Fano matches the
information theoretic minimum.

n lg(u/n) + 2n + o(n) bits

- **O(1)** random access
- **O(lg(u/n))** predecessor/successor

Static succinct data structure.

NO dynamic updates.

- **vEB Trees** [van Emde Boas, FOCS 1975]
- **x/y-Fast Tries** [Willard, IPL 1983]
- **Fusion Trees** [Fredman and Willard, JCSS 1993]
- **Exponential Search Trees** [Andersson and Thorup, JACM 2007]
- …

- Dynamic
- Most of them take optimal time

Elias-Fano matches the
information theoretic minimum.

$n \lg(u/n) + 2n + o(n)$ bits

- **O(1)** random access
- **O(lg(u/n))** predecessor/successor

Static succinct data structure.

NO dynamic updates.

- **vEB Trees** [van Emde Boas, FOCS 1975]
- **x/y-Fast Tries** [Willard, IPL 1983]
- **Fusion Trees** [Fredman and Willard, JCSS 1993]
- **Exponential Search Trees** [Andersson and Thorup, JACM 2007]
- …

- Dynamic
- Most of them take optimal time

O(n lg u) bits

(or even worse)

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns $\max\{y \in S : y < x\}$
- successor(x) returns $\min\{y \in S : y \geq x\}$

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns $\max\{y \in S : y < x\}$
- successor(x) returns $\min\{y \in S : y \geq x\}$

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns $\max\{y \in S : y < x\}$
- successor(x) returns $\min\{y \in S : y \geq x\}$

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list S of n sorted integer, support the following operations

- access(i) return the i-th smallest element of S
- insert(x) inserts x in S
- delete(x) deletes x from S

under the assumption that $w \leq \lg^{\gamma} n$ for some γ.

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns max{y ∊ S : y < x}
- successor(x) returns min{y ∊ S : y ≥ x}

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list S of n sorted integer, support the following operations

- access(i) return the i-th smallest element of S
- insert(x) inserts x in S
- delete(x) deletes x from S

under the assumption that $w \leq \lg^{\gamma} n$ for some γ.

Ω(lg n/lg lg n) amortized time per operation, in the cell-probe computational model.

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns max{y ∈ S : y < x}
- successor(x) returns min{y ∈ S : y ≥ x}

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list S of n sorted integer, support the following operations

- access(i) return the i-th smallest element of S
- insert(x) inserts x in S
- delete(x) deletes x from S

under the assumption that $w \leq \lg^{\gamma} n$ for some γ.

Ω(lg n/lg lg n) amortized time per operation, in the cell-probe computational model.

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns max{y ∊ S : y < x}
- successor(x) returns min{y ∊ S : y ≥ x}

[Patrascu and Thorup, STC 2007]

Optimal space/time trade-off for a *static* data structure taking m = $n2^a w$ bits, where *a* is the number of bits necessary to represent the mean number of bits per integer, i.e., a = lg(m/n) – lg w

$$\Theta\left(\min\left\{\log_w n, \lg\frac{w-\lg n}{a}, \frac{\lg\frac{w}{a}}{\lg\left(\frac{a}{\lg n}\lg\frac{w}{a}\right)}, \frac{\lg\frac{w}{a}}{\lg\left(\lg\frac{w}{a}/\lg\frac{\lg n}{a}\right)}\right\}\right)$$

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list S of n sorted integer, support the following operations

- access(i) return the i-th smallest element of S
- insert(x) inserts x in S
- delete(x) deletes x from S

under the assumption that w ≤ $\lg^\gamma n$ for some γ.

Ω(lg n/lg lg n) amortized time per operation, in the cell-probe computational model.

The (general) Dictionary problem

The dynamic dictionary problem consists in representing a set S of n objects so that the following operations are supported.

- insert(x) inserts x in S
- delete(x) deletes x from S
- search(x) checks whether x belongs to S
- minimum() returns the minimum element of S
- maximum() returns the maximum element of S
- predecessor(x) returns max{y ∈ S : y < x}
- successor(x) returns min{y ∈ S : y ≥ x}

[Patrascu and Thorup, STC 2007]

Optimal space/time trade-off for a *static* data structure taking m = $n2^a w$ bits, where *a* is the number of bits necessary to represent the mean number of bits per integer, i.e., a = lg(m/n) – lg w

$$\Theta\left(\min\left\{\log_w n, \lg\frac{w-\lg n}{a}, \frac{\lg\frac{w}{a}}{\lg\left(\frac{a}{\lg n}\lg\frac{w}{a}\right)}, \frac{\lg\frac{w}{a}}{\lg\left(\lg\frac{w}{a}/\lg\frac{\lg n}{a}\right)}\right\}\right)$$

The Dynamic List Representation problem

[Fredman and Saks, STC 1989]

Given a list S of n sorted integer, support the following operations

- access(i) return the i-th smallest element of S
- insert(x) inserts x in S
- delete(x) deletes x from S

under the assumption that $w \leq \lg^{\gamma} n$ for some γ.

$\Omega(\lg n/\lg \lg n)$ amortized time per operation, in the cell-probe computational model.

Goals

Goals

$n \lg(u/n) + 2n + o(n)$ bits

Goals

n lg(u/n) + 2n + o(n) bits + **o(n)** bits

Goals

negligible redundancy!

n lg(u/n) + 2n + o(n) bits + **o(n)** bits

Goals

negligible redundancy!

n lg(u/n) + 2n + o(n) bits + **o(n)** bits

1. Extend the *static* Elias-Fano representation to support predecessor and successor queries in optimal worst-case O(lg lg n) time.

2. Maintain S in a *fully dynamic fashion*, supporting in optimal worst-case time all the operations defined in the Dynamic Dictionary and Dynamic List Representation problems.

- **optimal time/space trade-off for successor search** [Patrascu and Thorup, STC 2007]
- **y-fast tries** [Willard, IPL 1983]

▶ **Theorem 1.** *There exists a data structure representing an ordered set* $\mathcal{S}(n, u)$ *of* n *integers drawn from a polynomial universe of size* $u = n^\gamma$*, for any* $\gamma = \Theta(1)$*, that takes* $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ *bits of space and supports* Access *in* $\mathcal{O}(1)$ *worst-case and* Predecessor/Successor *queries in optimal* $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ *worst-case time.*

- **optimal time/space trade-off for successor search** [Patrascu and Thorup, STC 2007]
- **y-fast tries** [Willard, IPL 1983]

▶ **Theorem 1.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes* $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ *bits of space and supports* Access *in* $\mathcal{O}(1)$ *worst-case and* Predecessor/Successor *queries in optimal* $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ *worst-case time.*

Idea: divide the sequence into **blocks** and use a **y-fast trie** to index the blocks.

Results - Dynamic Elias-Fano

- **optimal time/space trade-off for successor search** [Patrascu and Thorup, STC 2007]
- **y-fast tries** [Willard, IPL 1983]
- **dynamic prefix-sum data structure** [Bille *et al.*, arXiv preprint 2015]

▶ **Lemma 4.** *The total order of the blocks of* $\mathcal{C}$ *can be maintained by using a data structure that takes* $\mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ *bits of space and supports the following operations in* $\mathcal{O}(\log\log n)$ *worst-case time:* $\mathsf{Search}(x)$ *which returns a pointer to the block containing the integer* x*;* $\mathsf{Access}(i)$ *which returns the* i*-th integer of the total order;* $\mathsf{Insert}/\mathsf{Delete}$ *of a block.*

▶ **Theorem 3.** *There exists a data structure representing an ordered set* $\mathcal{S}(n, u)$ *of* n *integers drawn from a polynomial universe of size* $u = n^{\gamma}$*, for any* $\gamma = \Theta(1)$*, that takes* $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ *bits of space and supports:* Access *in* $\mathcal{O}(\log n/\log\log n)$ *worst-case;* $\mathsf{Insert}/\mathsf{Delete}$ *in* $\mathcal{O}(\log n/\log\log n)$ *amortized;* $\mathsf{Minimum}/\mathsf{Maximum}$ *in* $\mathcal{O}(1)$ *and* $\mathsf{Predecessor}/\mathsf{Successor}$ *queries in* $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ *worst-case time. These time bounds are optimal.*

Results - Dynamic Elias-Fano

- **optimal time/space trade-off for successor search** [Patrascu and Thorup, STC 2007]
- **y-fast tries** [Willard, IPL 1983]
- **dynamic prefix-sum data structure** [Bille *et al.*, arXiv preprint 2015]

▶ **Lemma 4.** *The total order of the blocks of $\mathcal{C}$ can be maintained by using a data structure that takes $\mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ bits of space and supports the following operations in $\mathcal{O}(\log\log n)$ worst-case time:* Search(x) *which returns a pointer to the block containing the integer x;* Access(i) *which returns the i-th integer of the total order;* Insert/Delete *of a block.*

▶ **Theorem 3.** *There exists a data structure representing an ordered set $\mathcal{S}(n, u)$ of n integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes* EF$(\mathcal{S}(n, u)) + o(n)$ *bits of space and supports:* Access *in $\mathcal{O}(\log n / \log\log n)$ worst-case;* Insert/Delete *in $\mathcal{O}(\log n / \log\log n)$ amortized;* Minimum/Maximum *in $\mathcal{O}(1)$ and* Predecessor/Successor *queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time. These time bounds are optimal.*

Idea: use a 2-level indexing data structure.

- First level indexes **blocks** using a **y-fast trie** and the **dynamic prefix-sum** data structure by Bille *et al.*
- Second level indexes **mini blocks** using the data structure of the Lemma.

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

devised

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

devised

⋮

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

devised

⋮

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

devised

⋮

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1

different

algorithms

devised

⋮

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	
algorithms	
devised	
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	
algorithms	
devised	
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	different algorithms
algorithms	
devised	
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	different algorithms
algorithms	
devised	
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	different algorithms
algorithms	algorithms devised
devised	
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	different algorithms
algorithms	algorithms devised
devised	
⋮	

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

27

N = 1	**N = 2**
different	different algorithms
algorithms	algorithms devised
devised	devised to
⋮	

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

N = 1	**N = 2**
different	different algorithms
algorithms	algorithms devised
devised	devised to
⋮	⋮

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

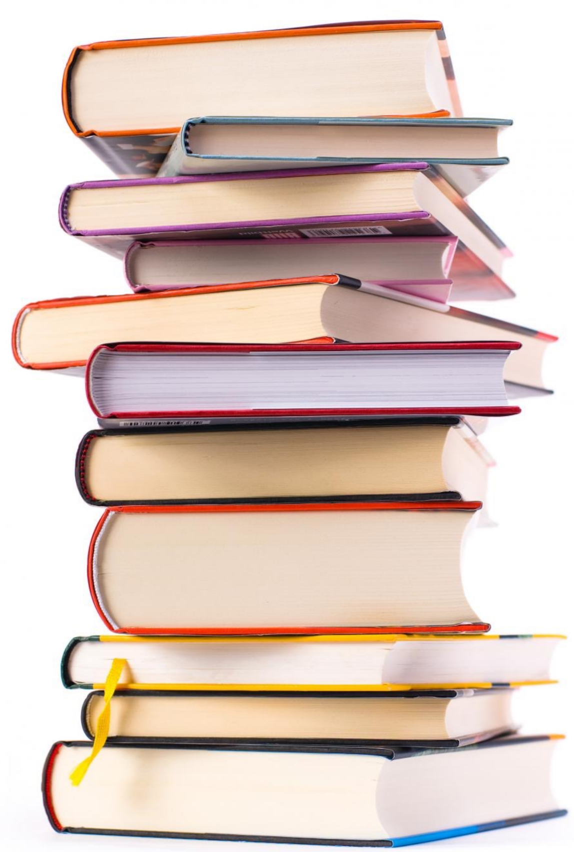
N = 1	**N = 2**
different	different algorithms
algorithms	algorithms devised
devised	devised to
⋮	⋮

N	number of grams
1	8761
2	38900
3	61516
4	70186
5	73187

N-grams

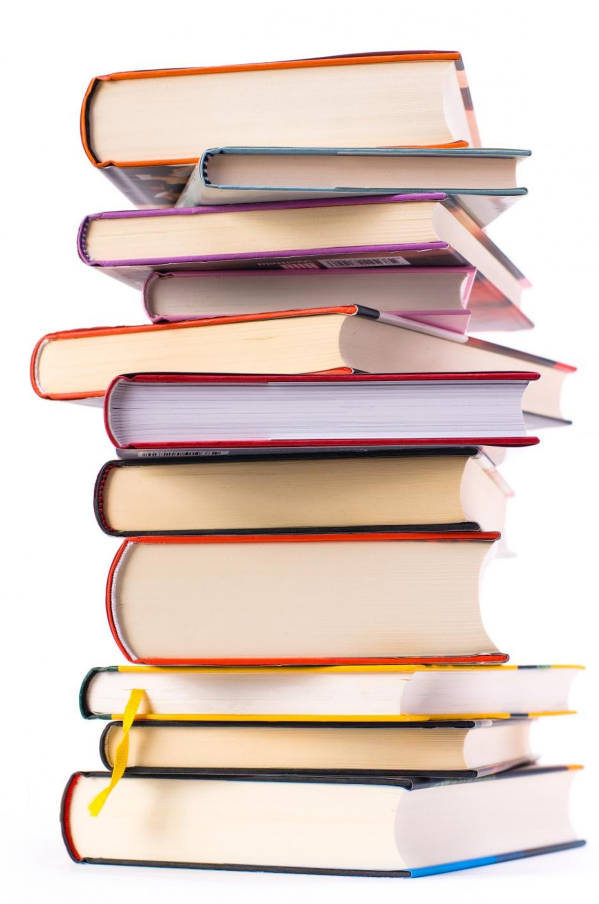
Strings of at most N words.

N typically ranges from 1 to 5.



Strings of at most N words.

N typically ranges from 1 to 5.



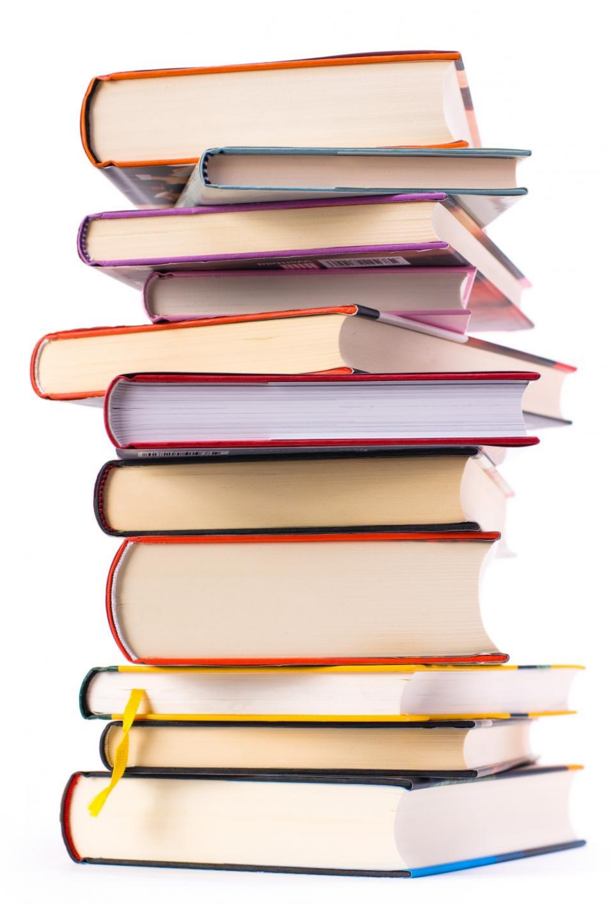
Google Books

~6% of the books ever published

N-grams

Strings of at most N words.

N typically ranges from 1 to 5.



Google Books

~6% of the books ever published

N	number of grams
1	24,359,473
2	667,284,771
3	7,397,041,901
4	1,644,807,896
5	1,415,355,596

More than 11 billion grams.

N-grams - Why? 2

Word prediction.

N-grams - Why?

Word prediction.

space and time-efficient $\boxed{\quad ? \quad}$

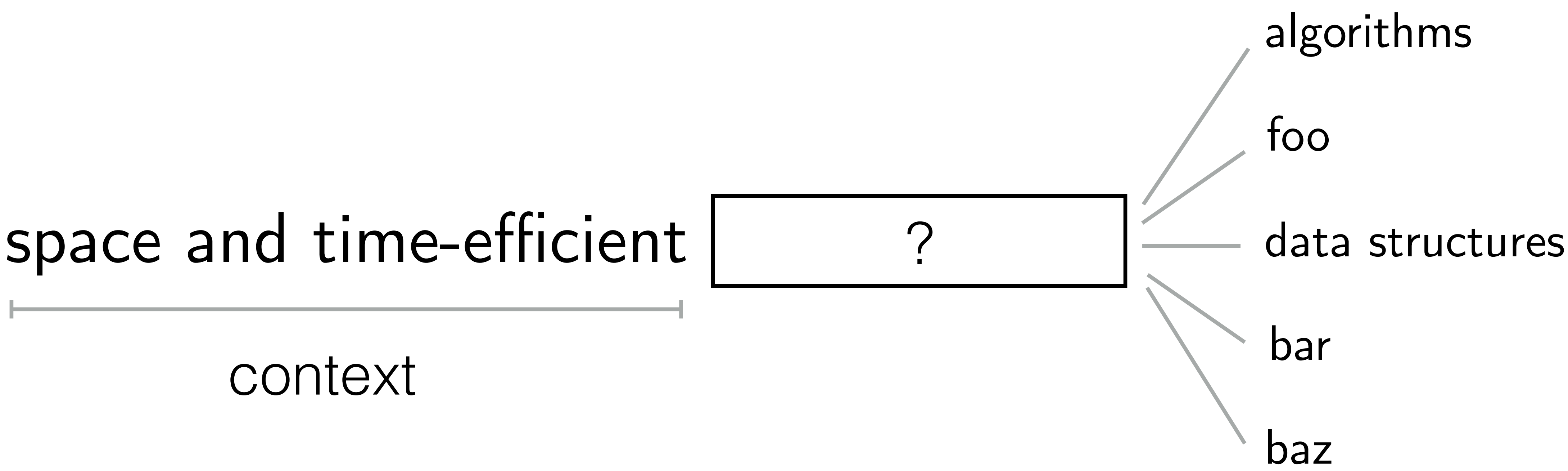
N-grams - Why?

Word prediction.

space and time-efficient [?]

context

Word prediction.



N-grams - Why? 2

Word prediction.

space and time-efficient ?

context

	frequency count
algorithms	**1214**
foo	**2**
data structures	**3647**
bar	**3**
baz	**1**

N-grams - Why?

Word prediction.

space and time-efficient ?

context

	frequency count
algorithms	**1214**
foo	**2**
data structures	**3647**
bar	**3**
baz	**1**

$$\mathcal{P}\left(\text{``data structures''} \mid \text{``space and time-efficient''}\right) \approx \frac{f\left(\text{``space and time-efficient data structures''}\right)}{f\left(\text{``space and time-efficient''}\right)}$$

N-grams - Who cares?

Google Research Blog

The latest news from Research at Google

All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of *one trillion words* from public Web pages.

Google Research Blog

The latest news from Research at Google

All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of *one trillion words* from public Web pages.

N-grams - Who cares?

Google Research Blog

The latest news from Research at Google

Google Translate

All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of *one trillion words* from public Web pages.

What can I help you with?

What can I
help you with?

Siri

N-grams - Challenge

Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.

N-grams - Challenge

Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.

Efficient map.

N-grams - Challenge

Store massive N-grams datasets such that given a pattern, we can return its frequency count at light speed.

Efficient map.

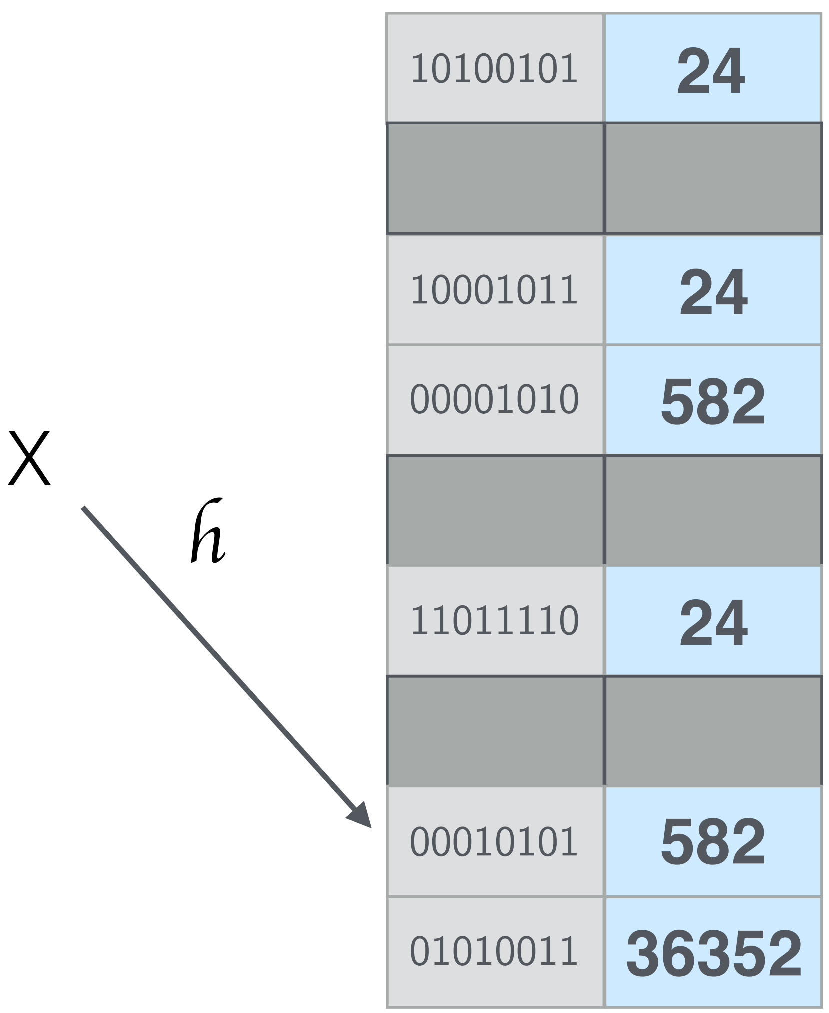
Data Structures **+** Data Compression ➡ Faster Algorithms

Open-addressing VS **Tries**

Open-addressing VS **Tries**

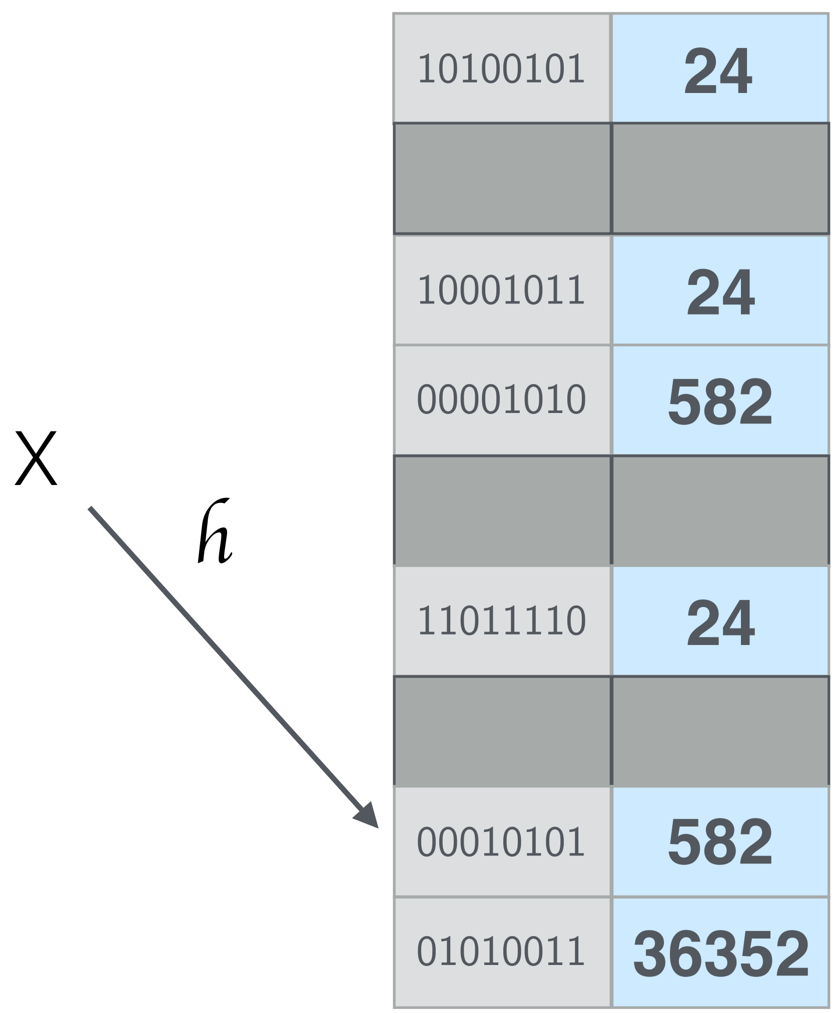
10100101	24
10001011	24
00001010	582
11011110	24
00010101	582
01010011	36352

Open-addressing VS **Tries**



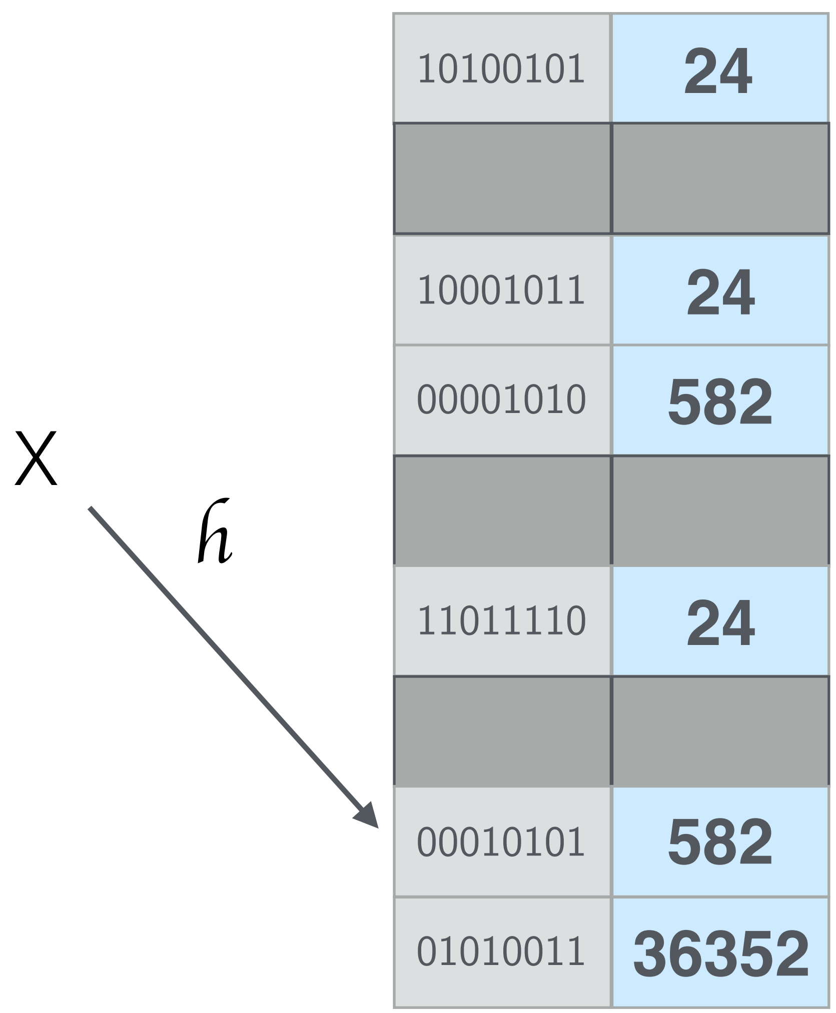
Open-addressing VS Tries

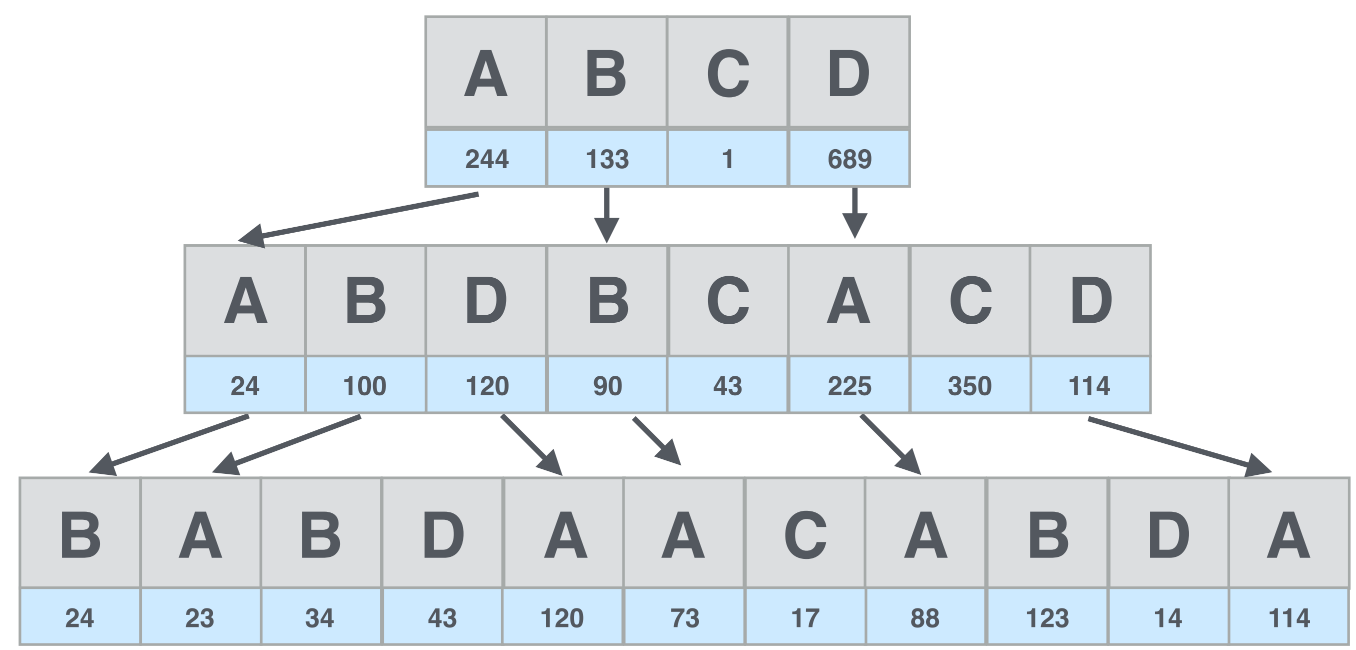
+ time
- space



Open-addressing VS Tries

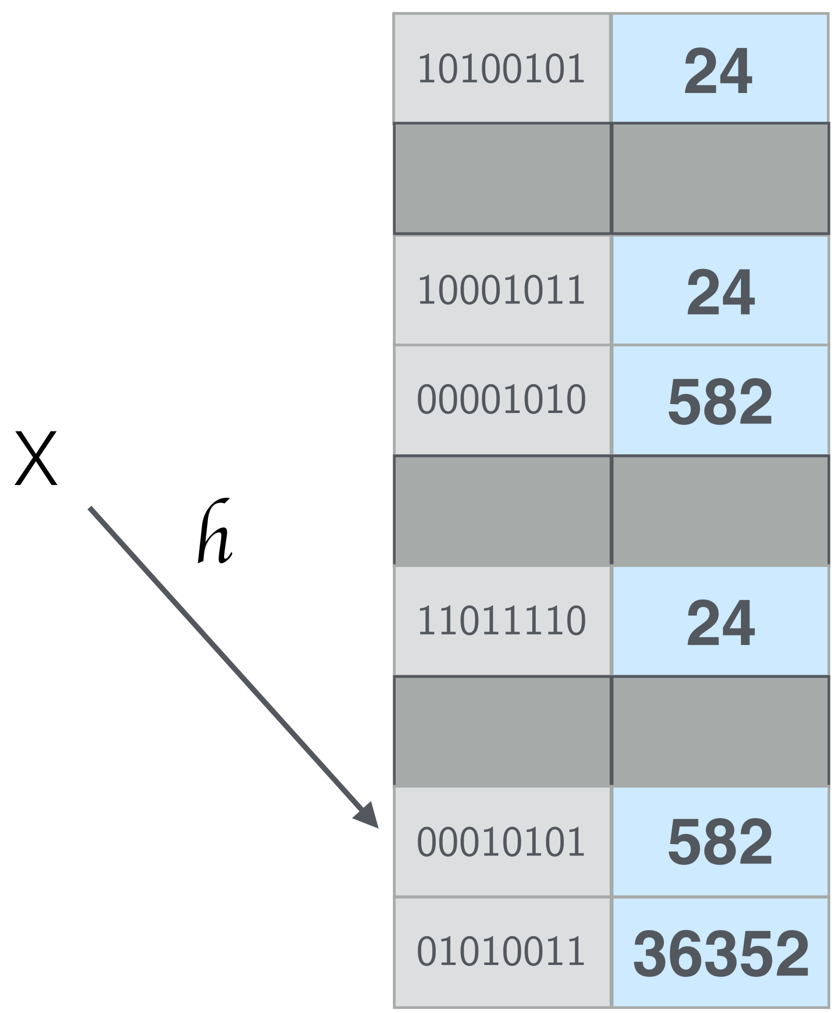
+ time
- space

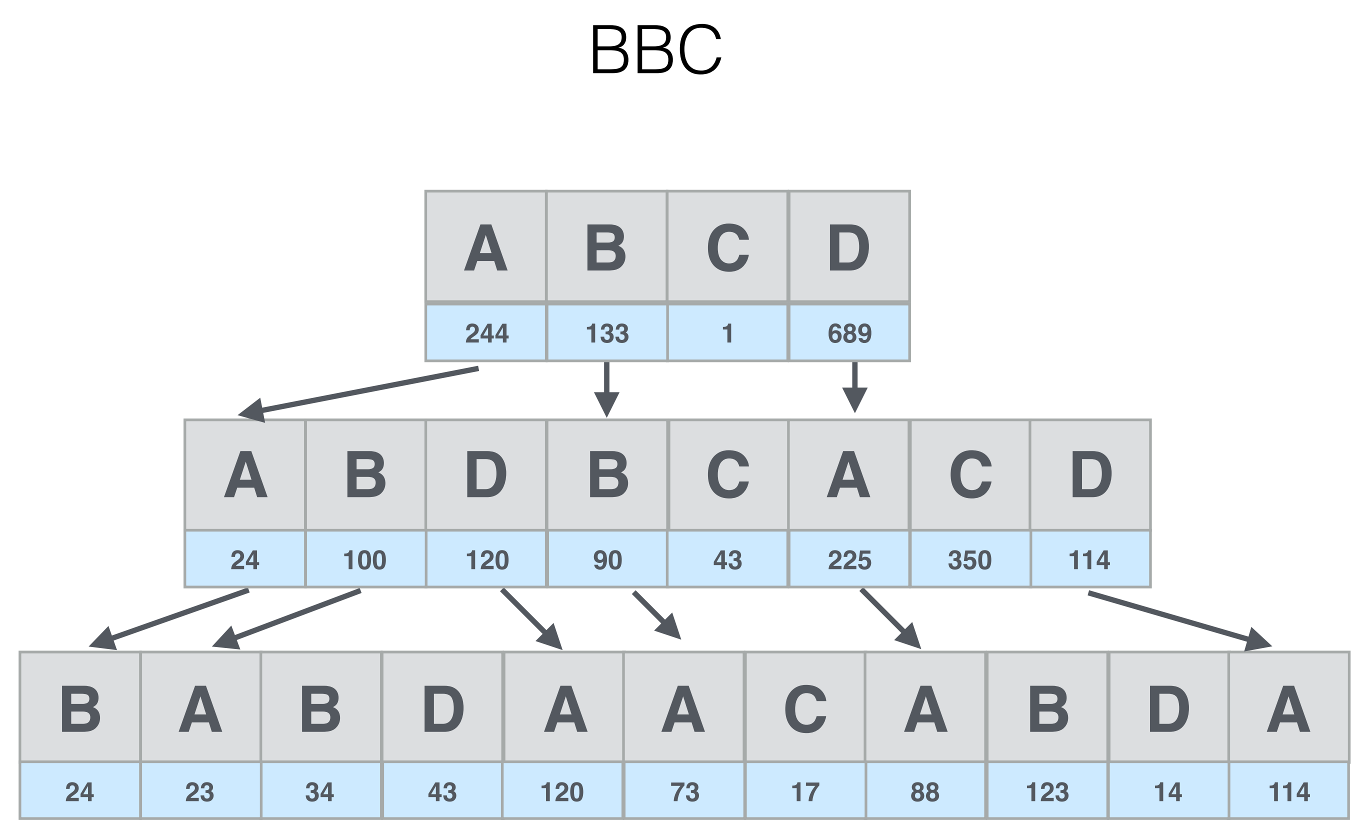




Open-addressing VS Tries

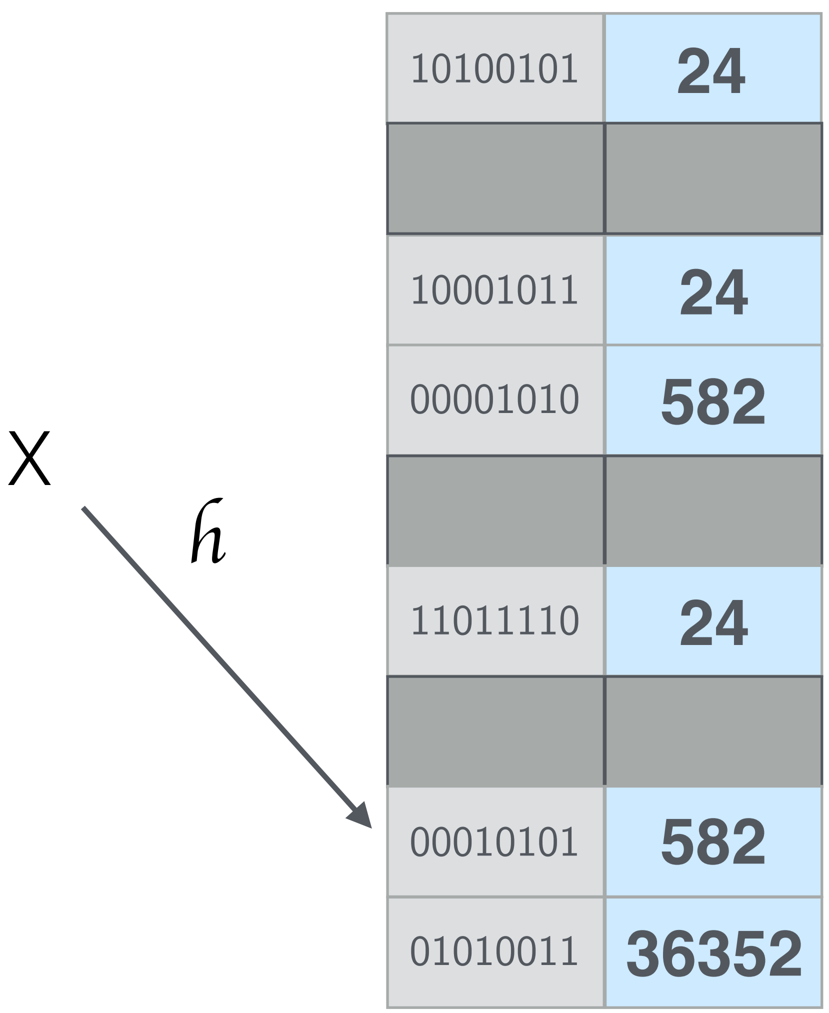
+ time
- space

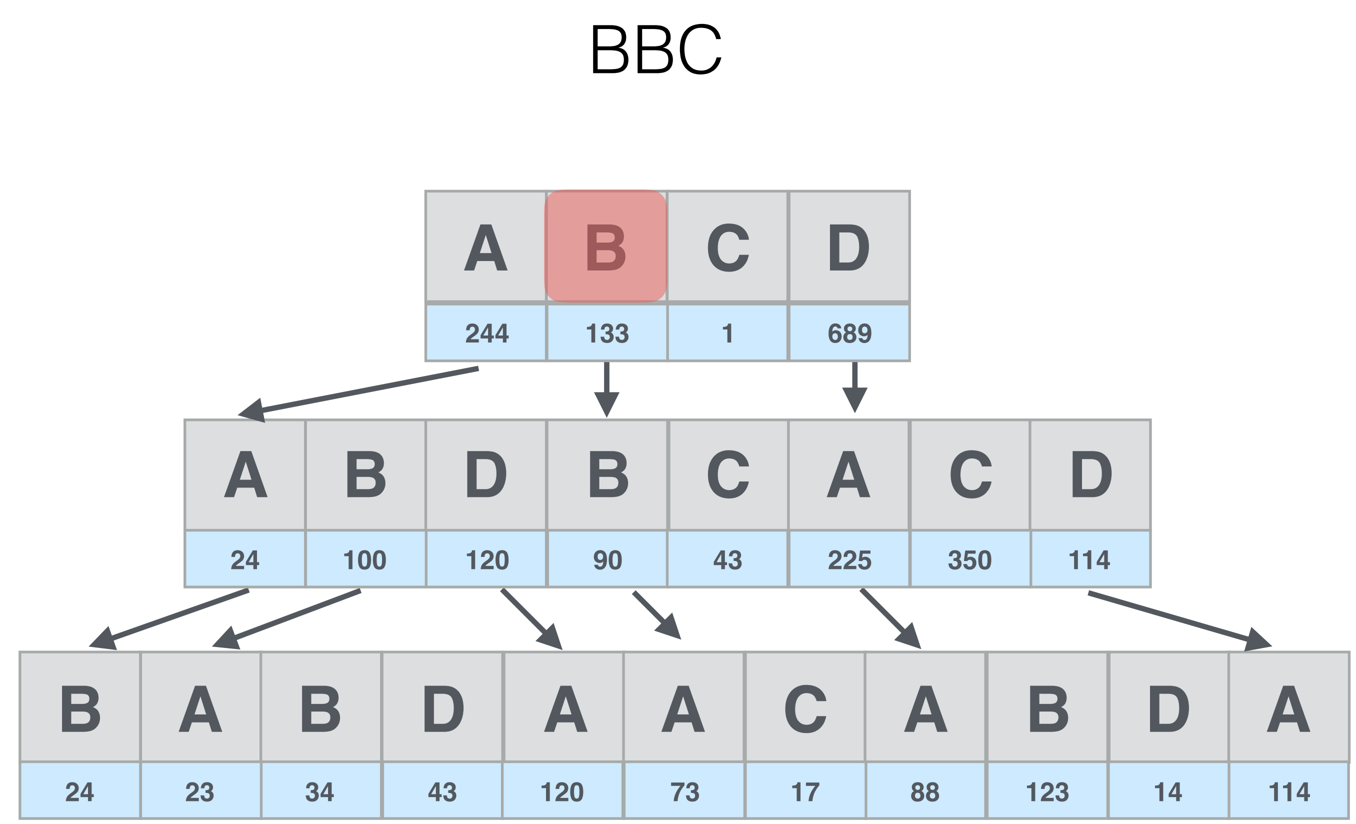




Open-addressing VS Tries

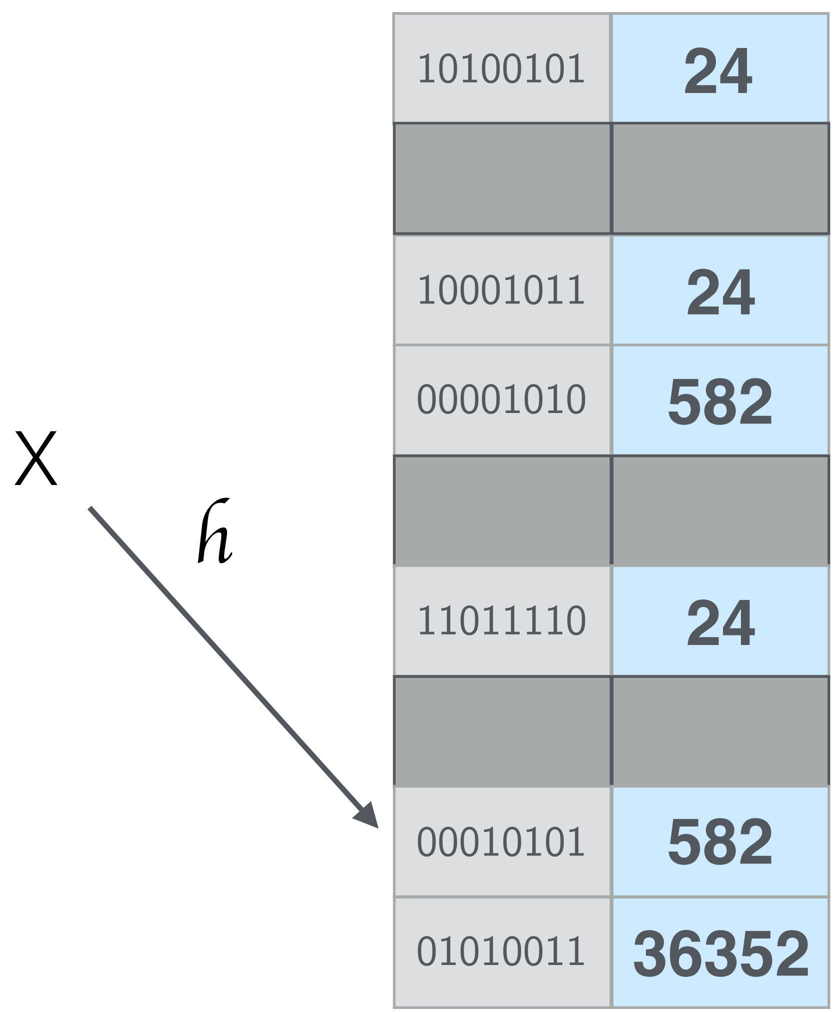
+ time
- space

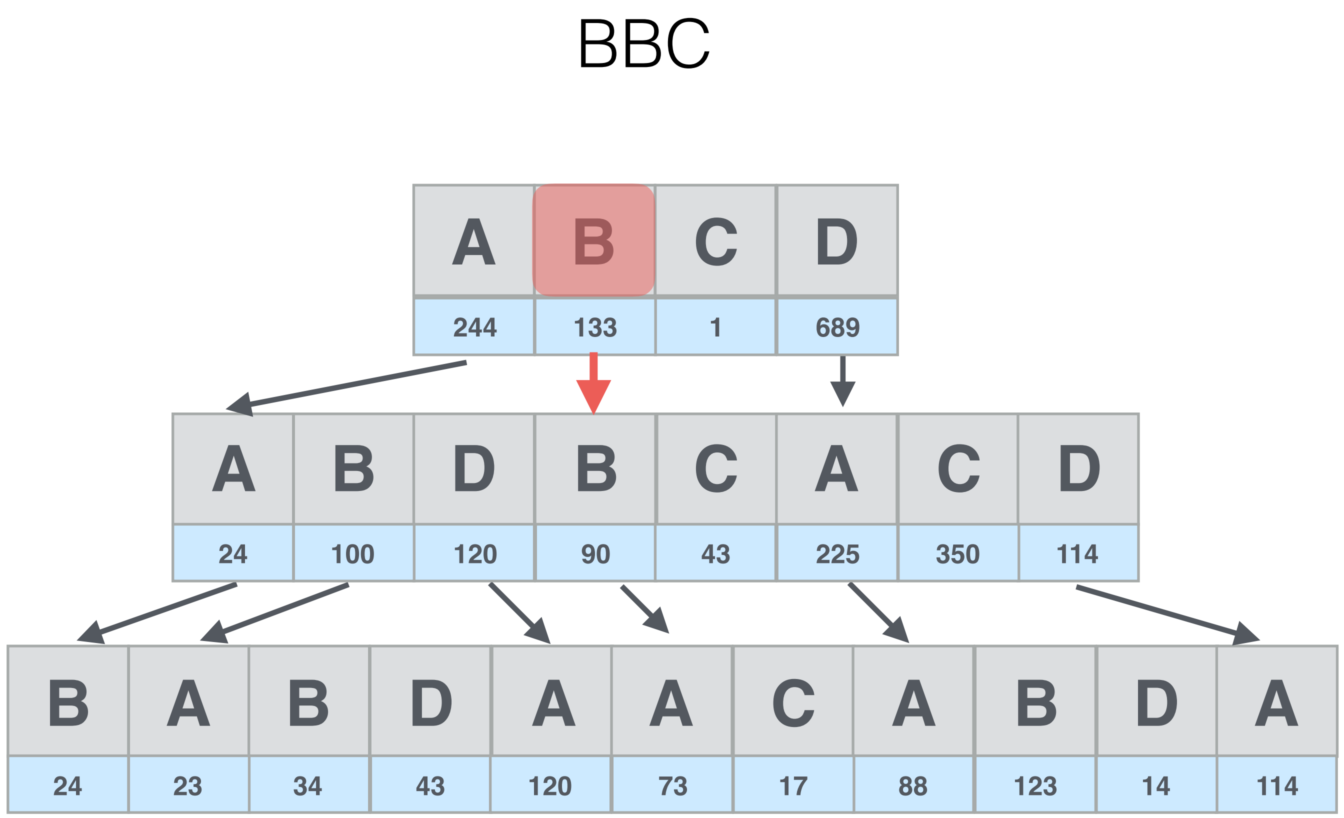




Open-addressing VS **Tries**

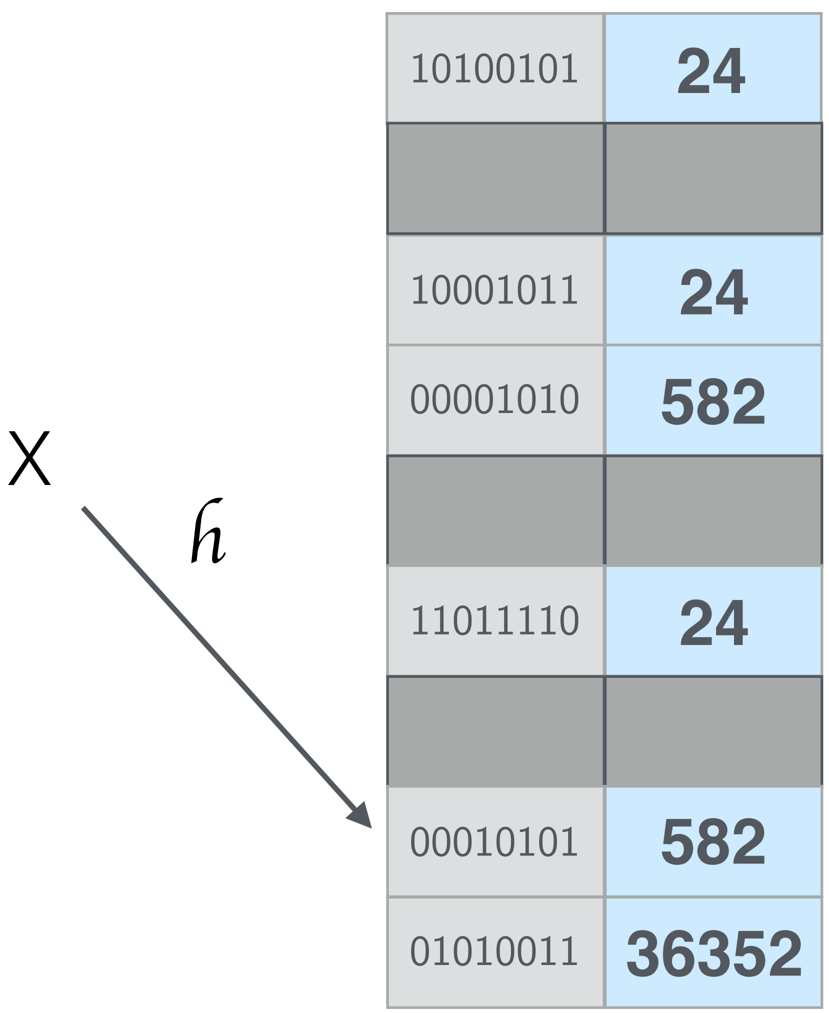
+ time
- space

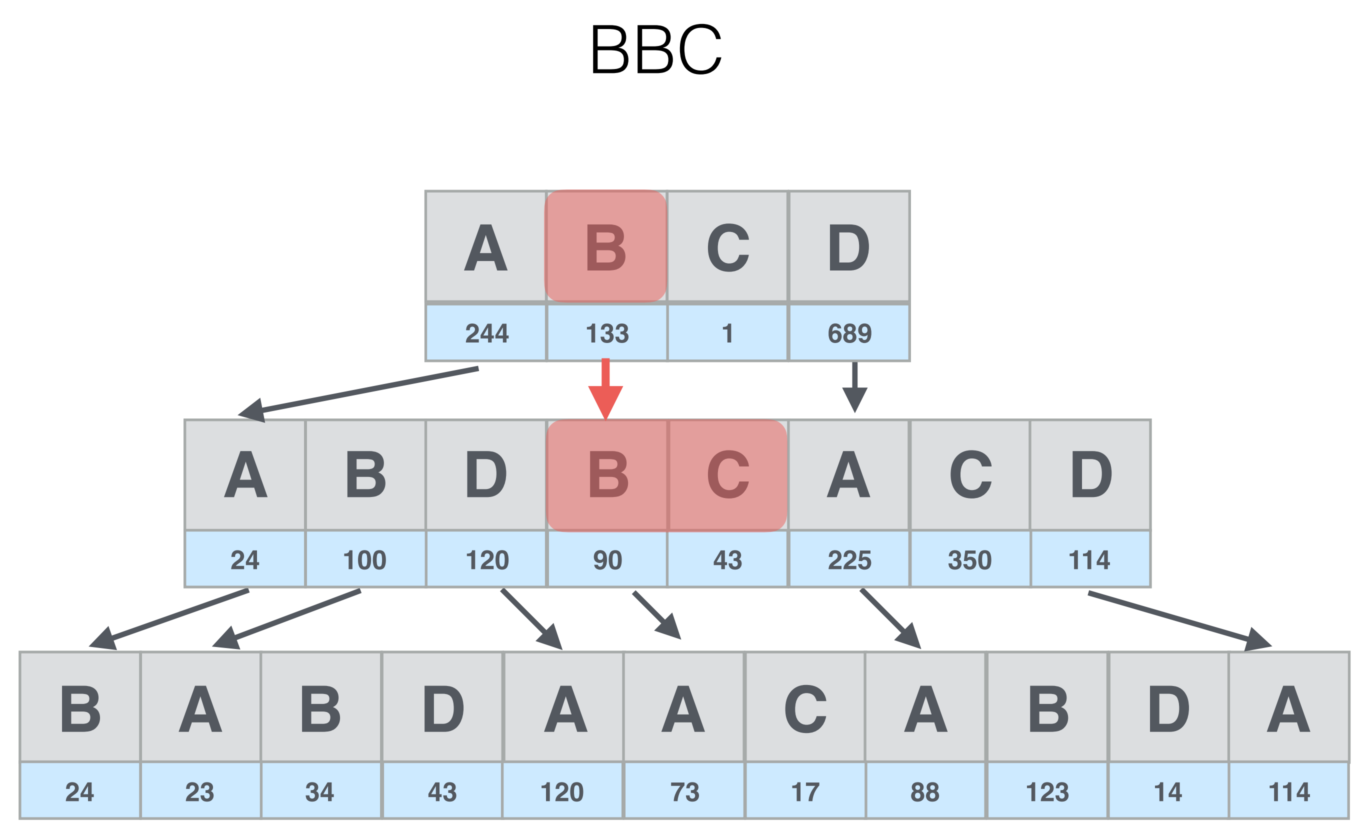




Open-addressing VS **Tries**

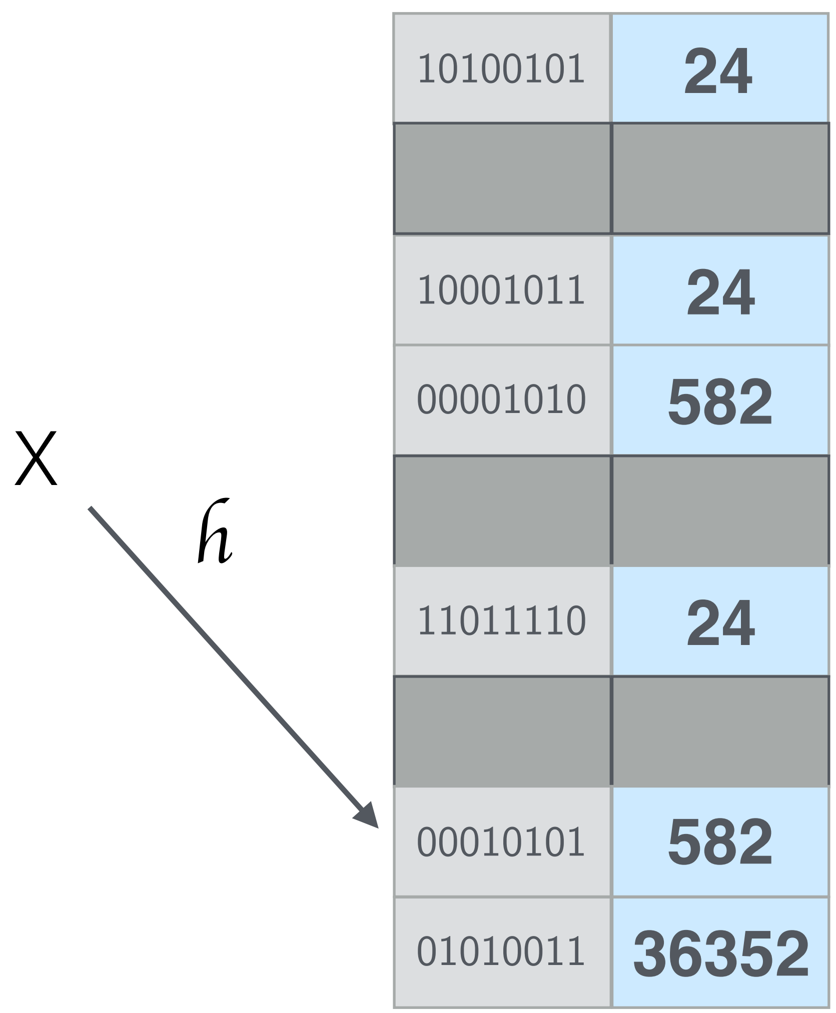
+ time
- space

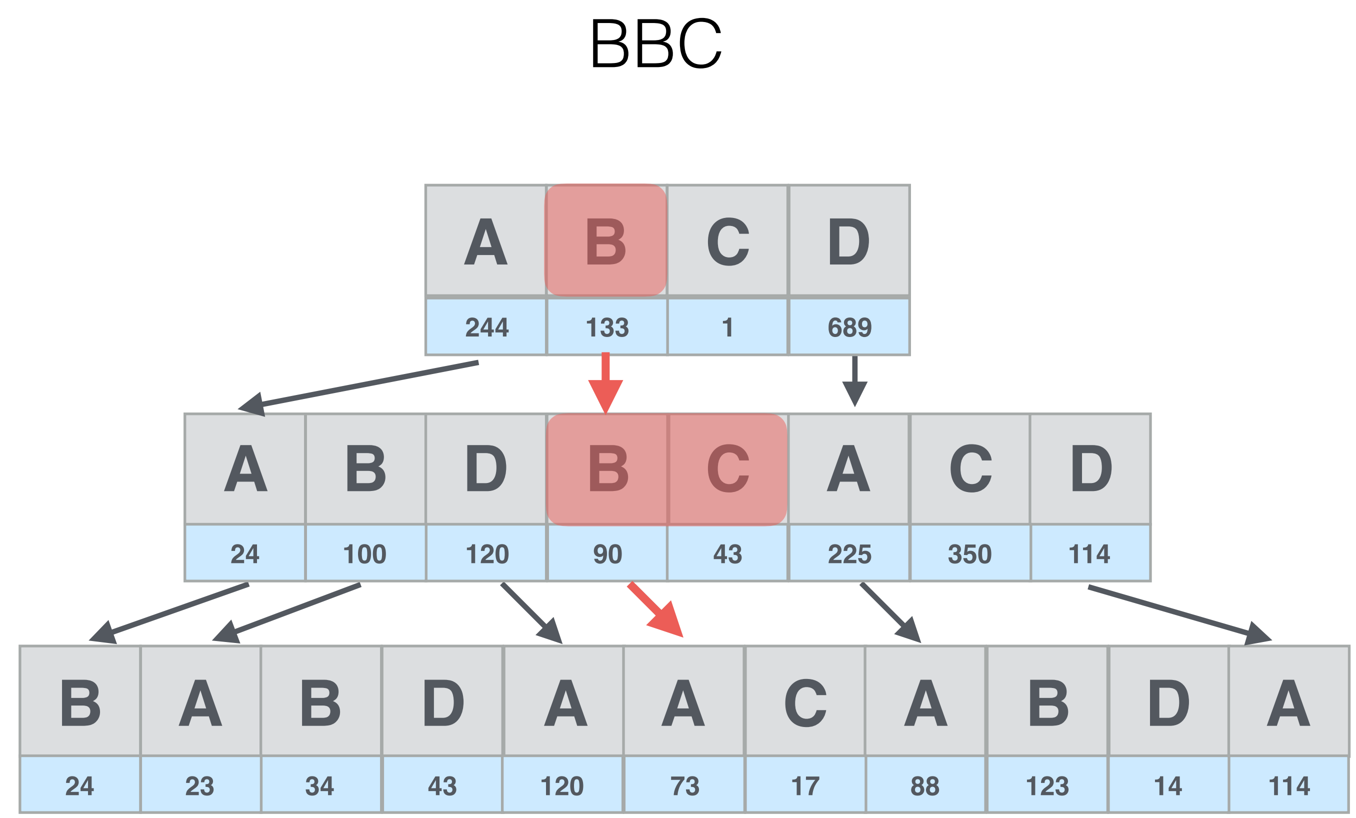




Open-addressing VS **Tries**

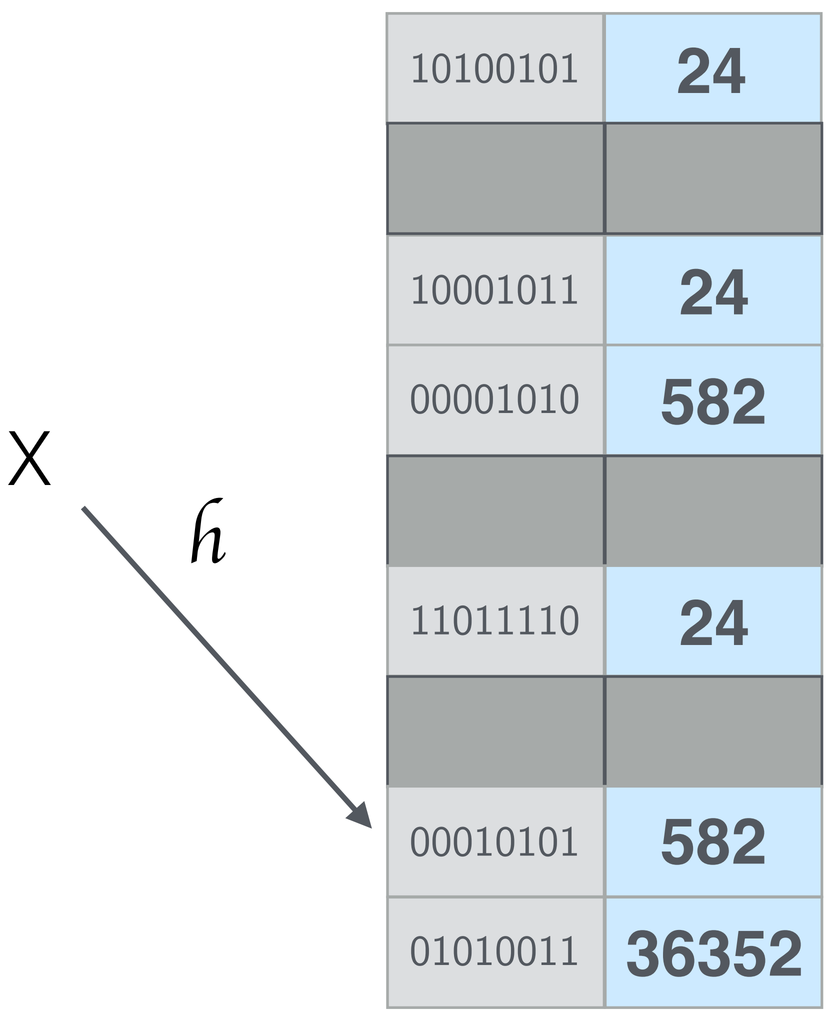
+ time
- space



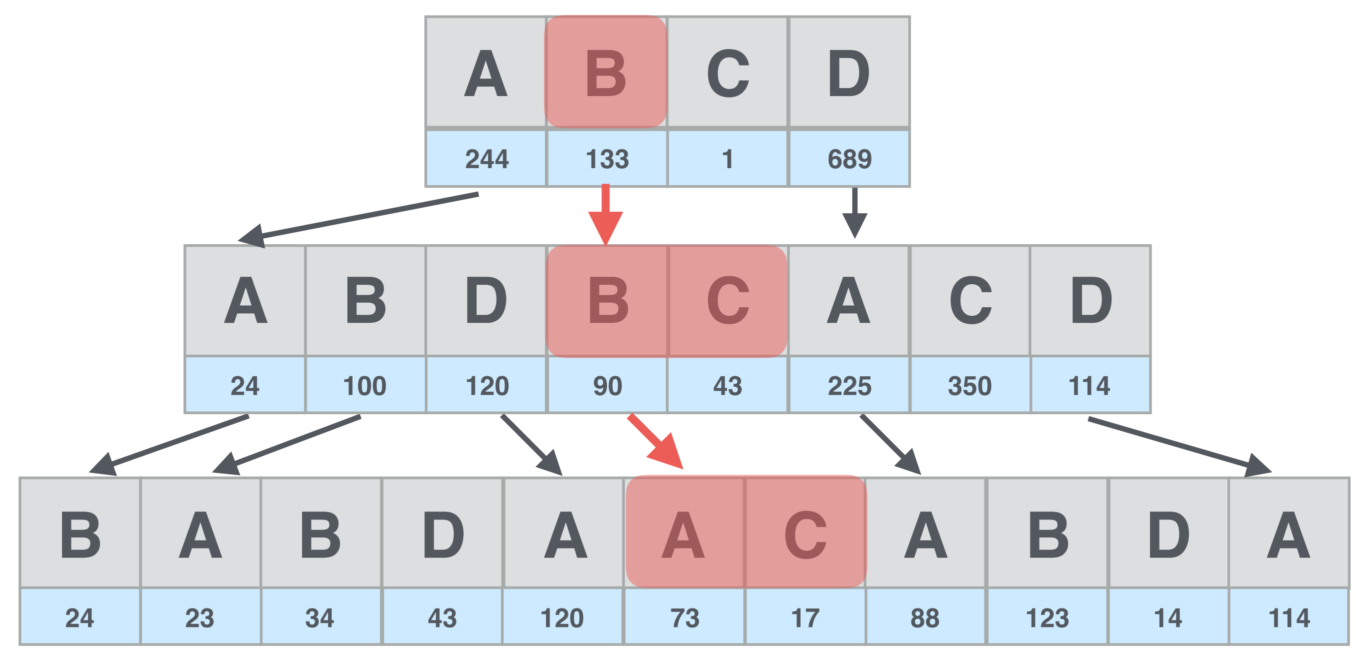


Open-addressing VS Tries

+ time
- space



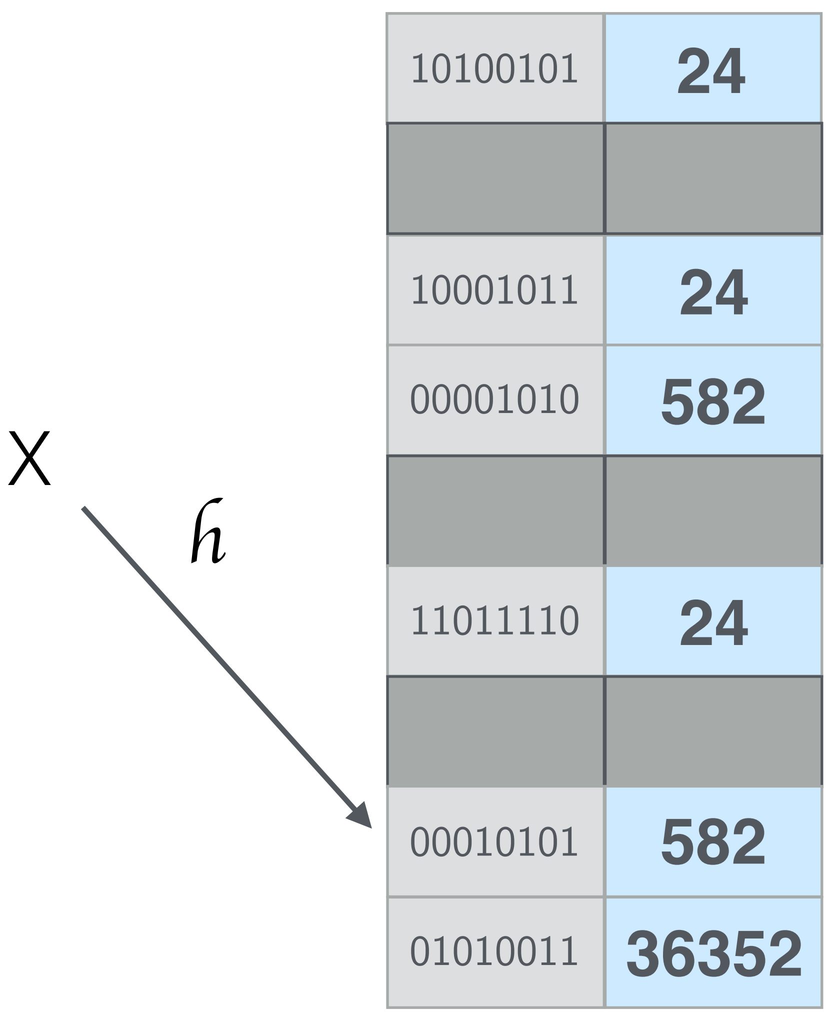
BBC

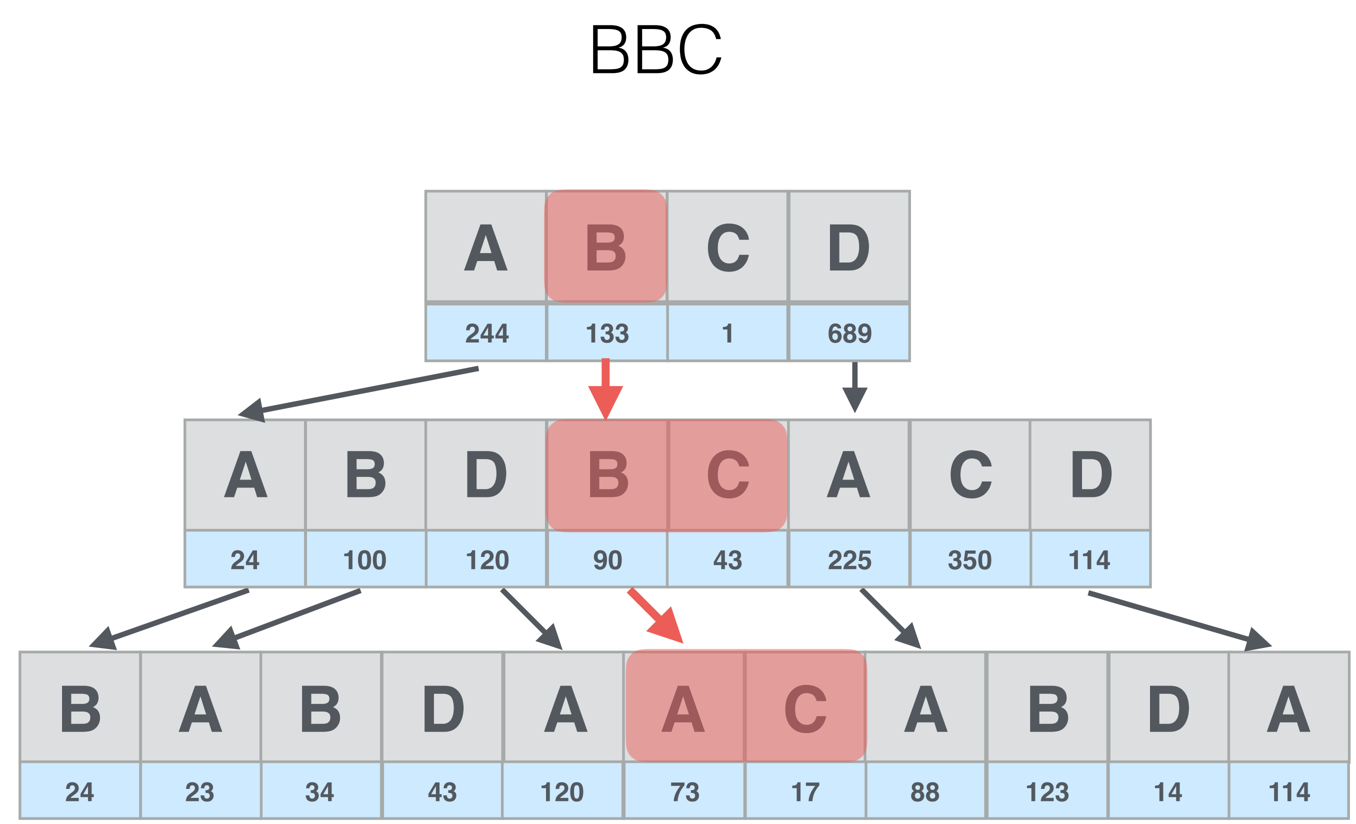


N-grams - Data structures

Open-addressing VS **Tries**

Open-addressing	Tries
+ time - space	+ space - time

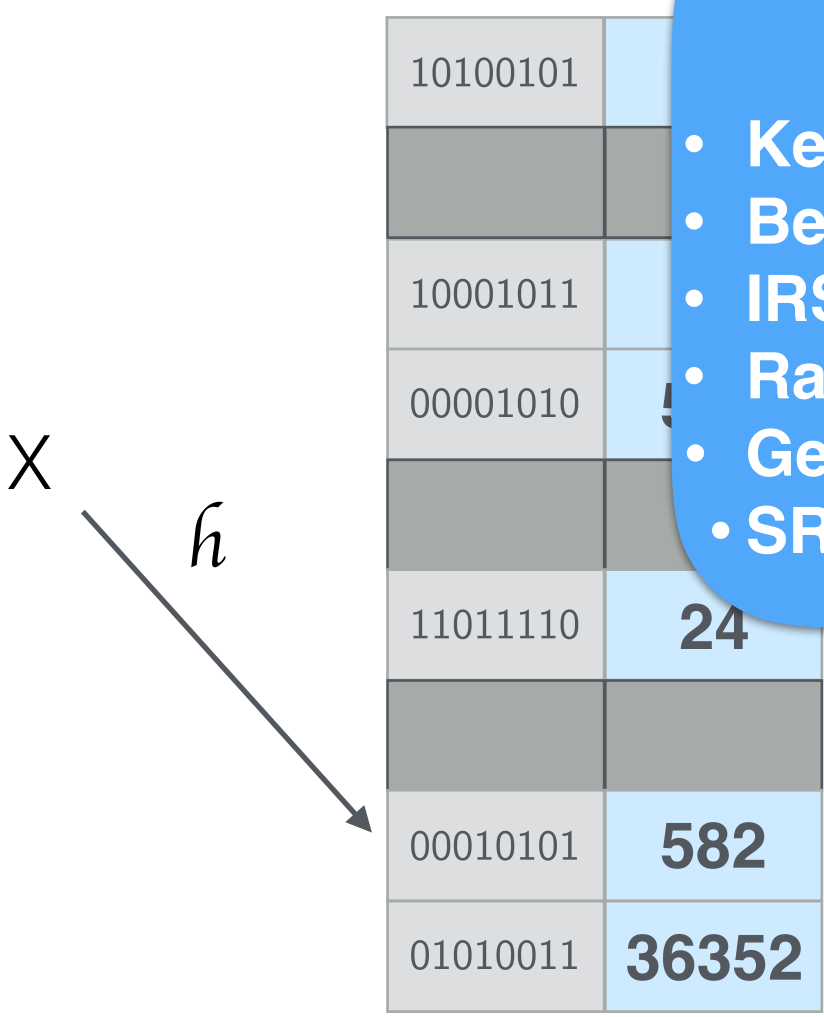


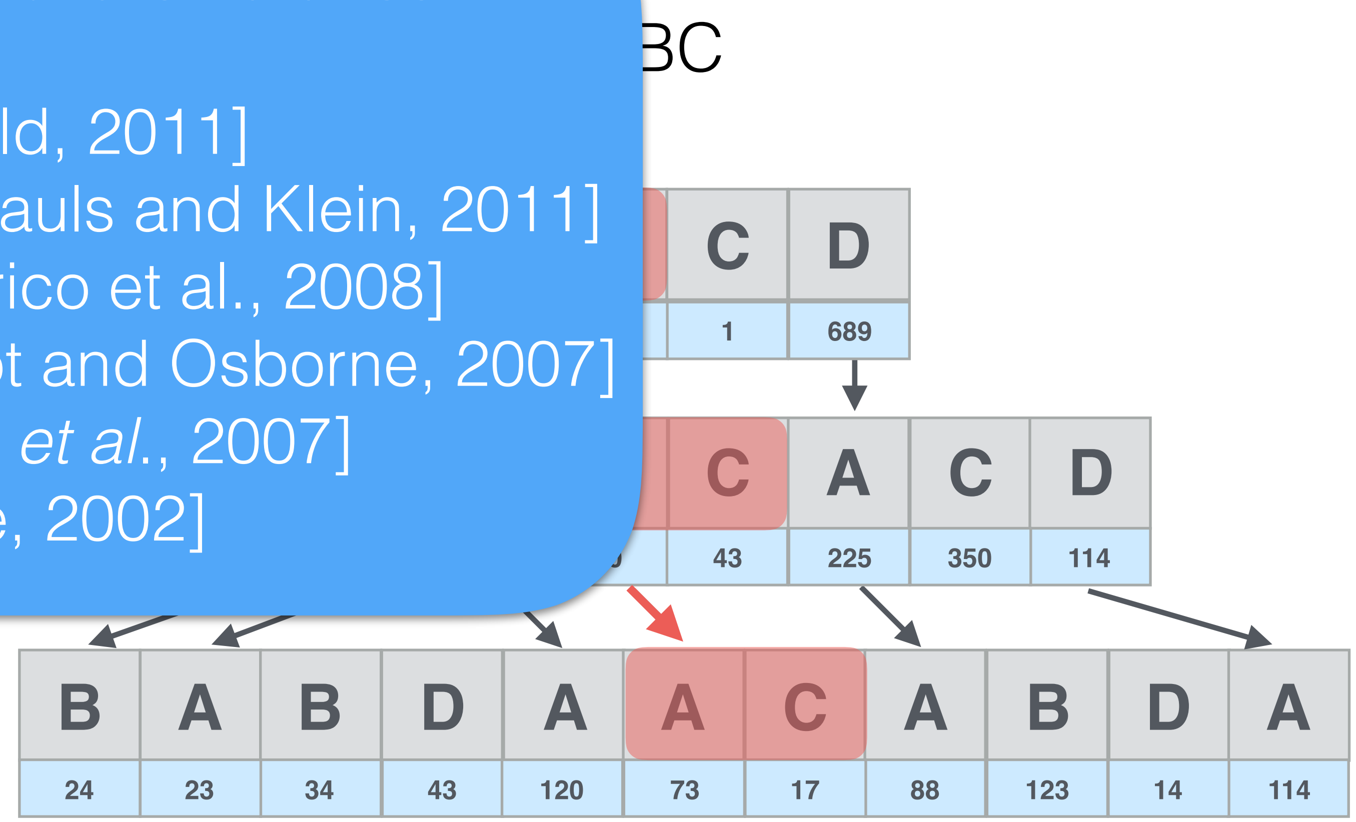


Open-addressing VS Tries

+ time
- space

+ space
- time

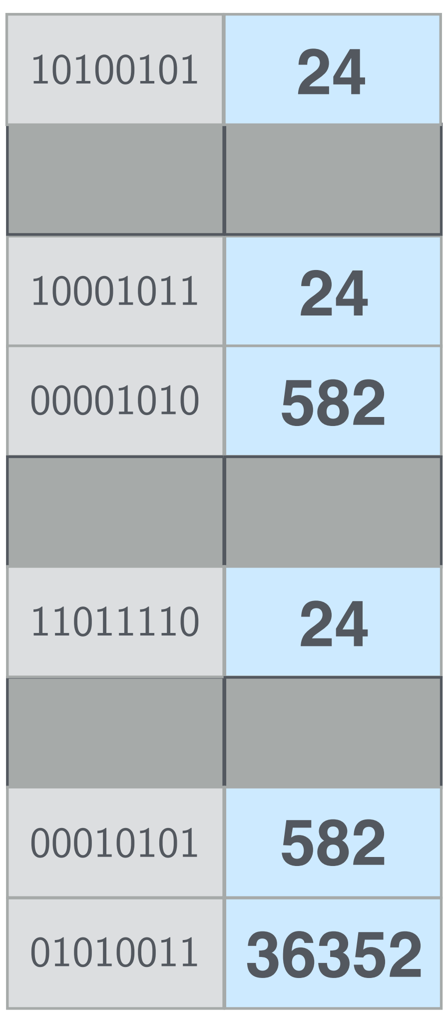


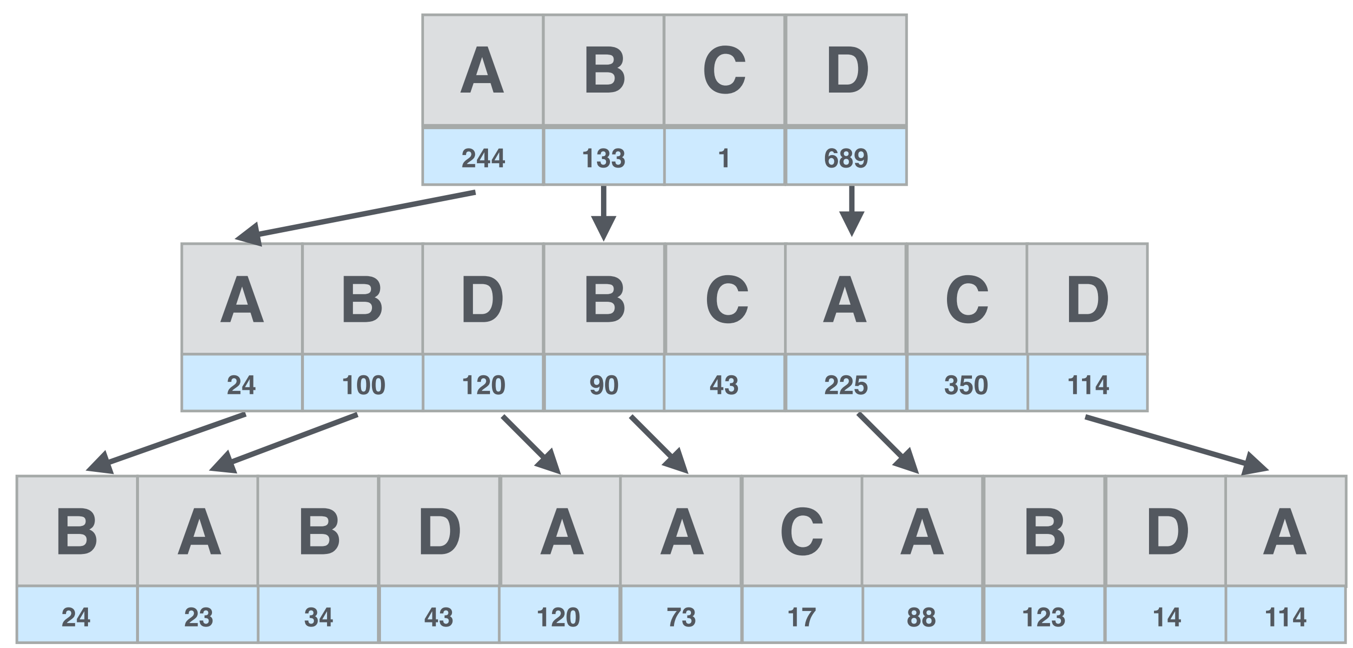


Several software libraries

- **KenLM** [Heafield, 2011]
- **BerkeleyLM** [Pauls and Klein, 2011]
- **IRSTLM** [Federico et al., 2008]
- **RandLM** [Talbot and Osborne, 2007]
- **Get1T** [Hawker *et al.*, 2007]
- **SRILM** [Stolcke, 2002]

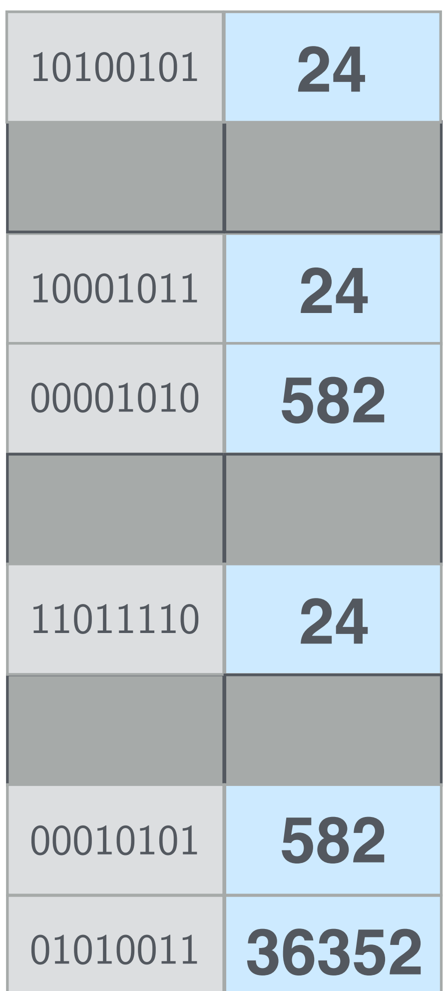
Hash-based VS **Trie-based**

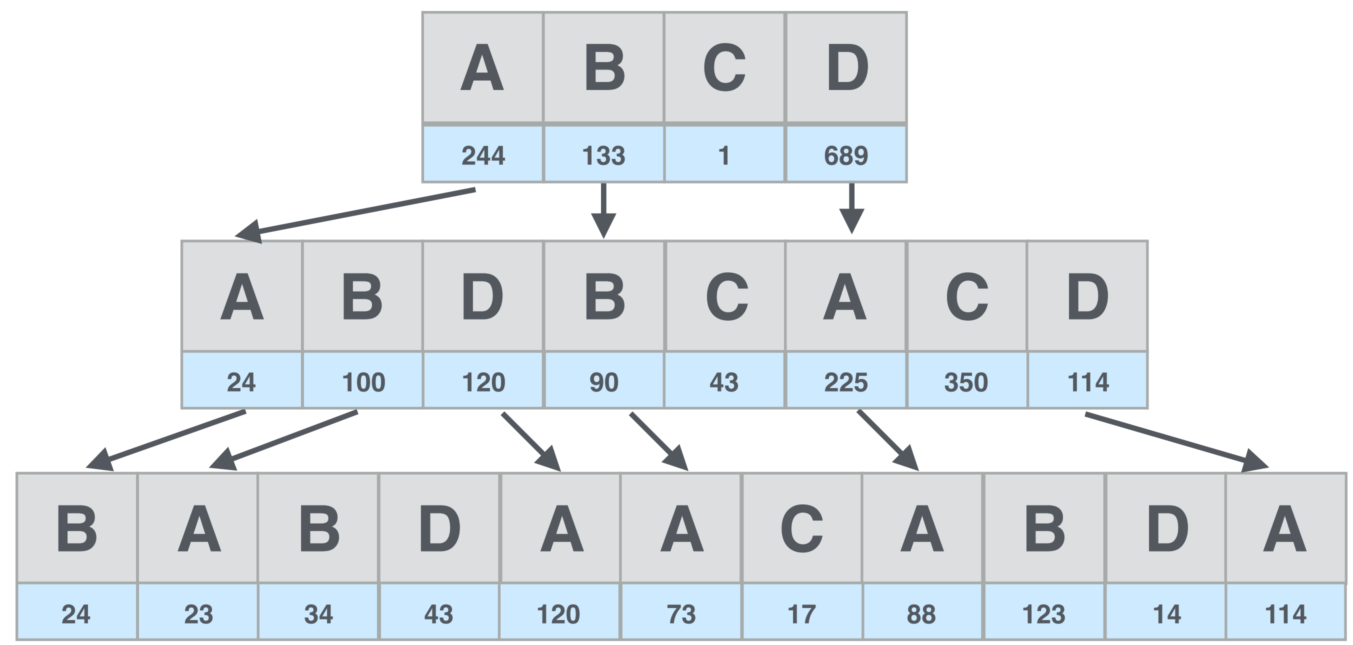




Hash-based VS **Trie-based**

Open addressing?





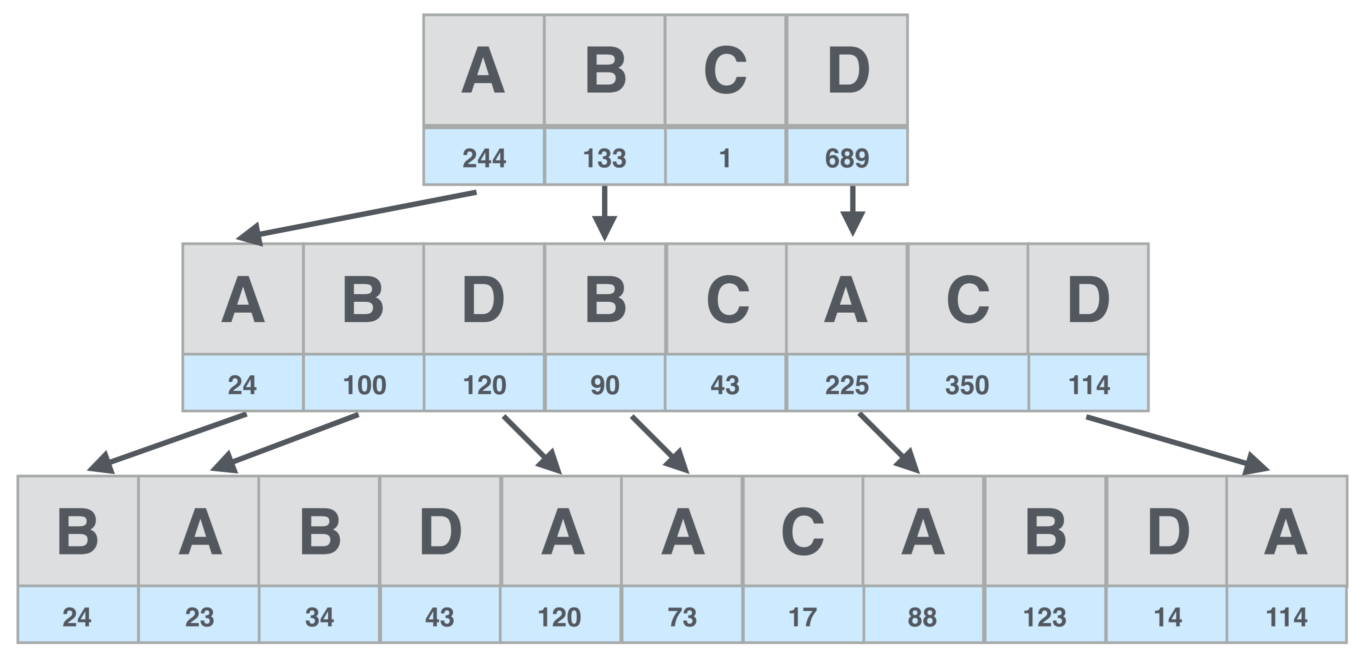
Hash-based VS Trie-based

Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.

10100101	24
10001011	24
00001010	582
11011110	24
00010101	582
01010011	36352



`tongrams` - Tons of *N*-Grams

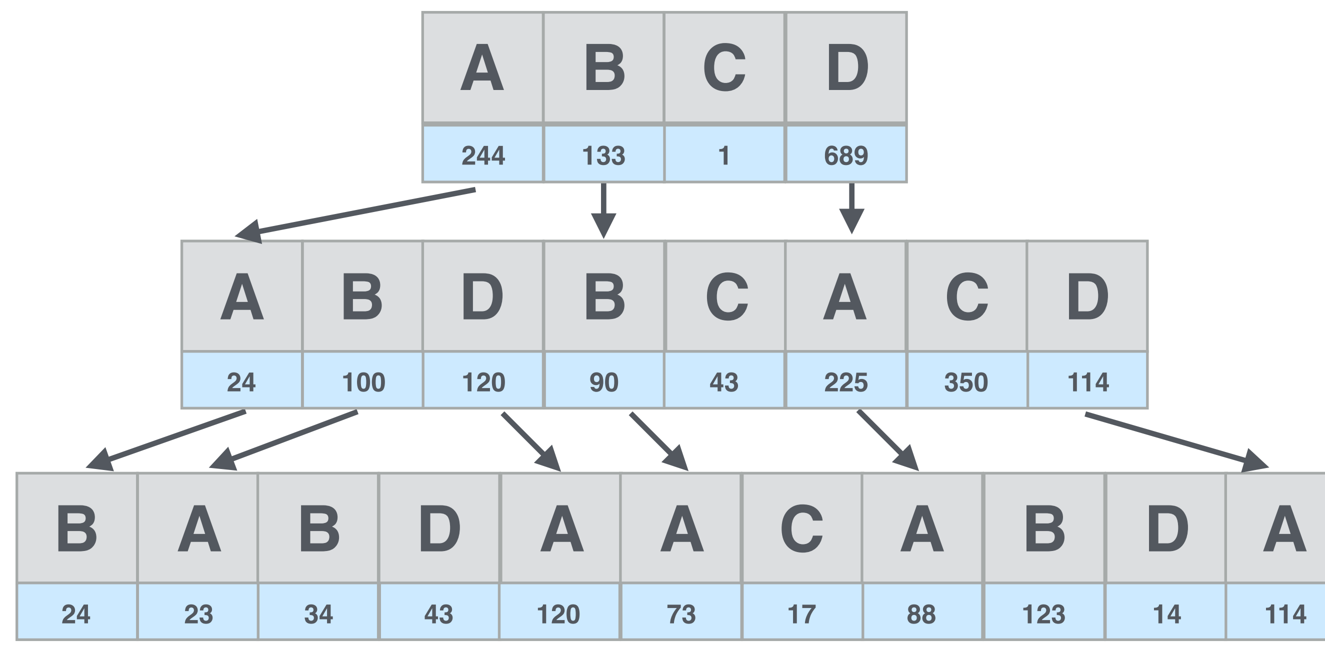
Hash-based VS **Trie-based**

Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.

10100101	24
10001011	24
00001010	582
11011110	24
00010101	582
01010011	36352



Hash-based VS **Trie-based**

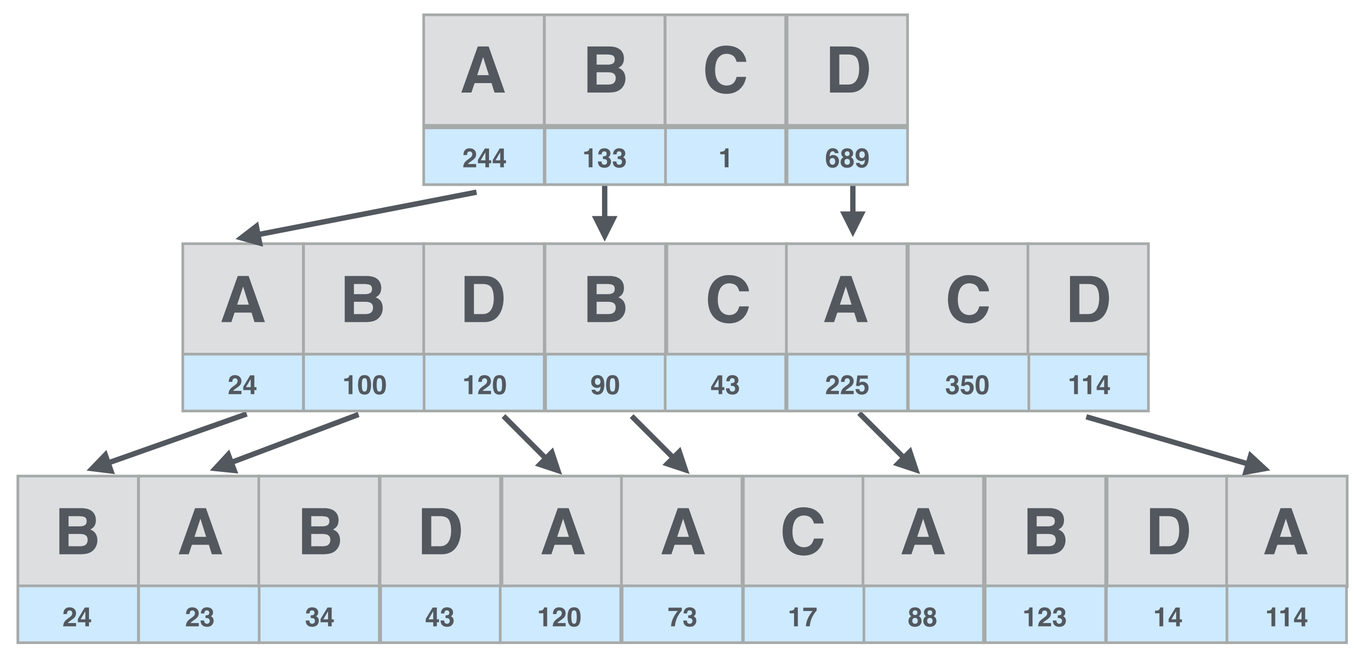
Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.

10100101	24
10001011	24
00001010	582
11011110	24

00010101	582
01010011	36352



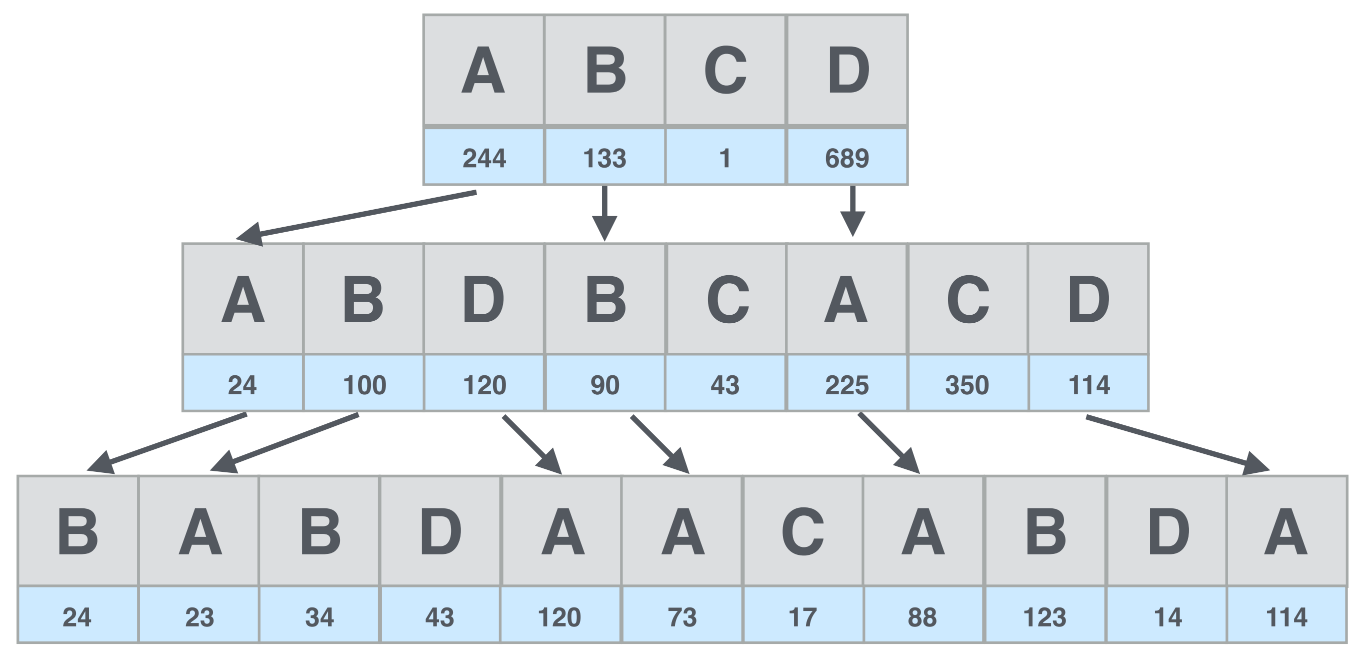
Hash-based VS **Trie-based**

Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.

10100101	24
10001011	24
00001010	582
11011110	24
00010101	582
01010011	36352



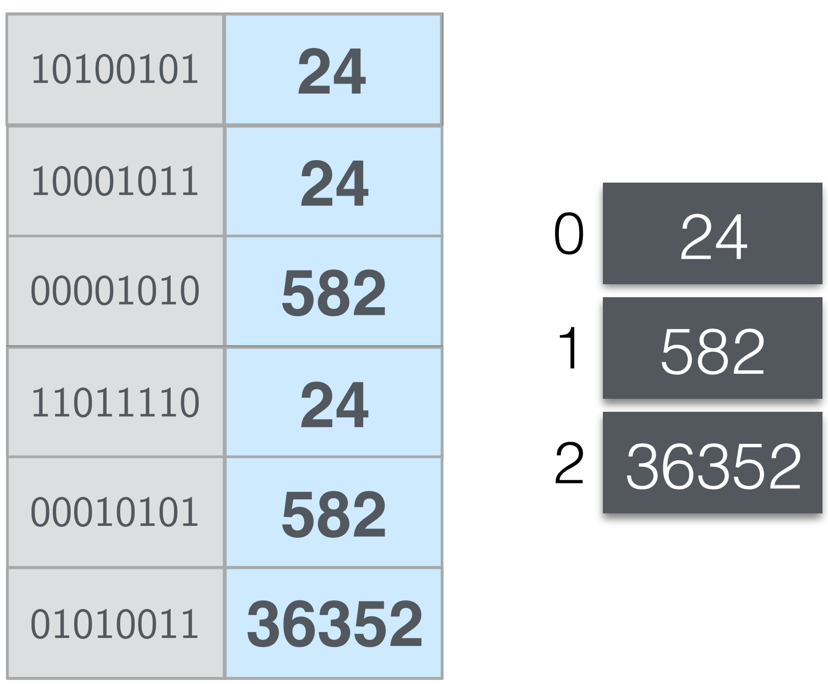
tongrams - Tons of *N*-Grams 1

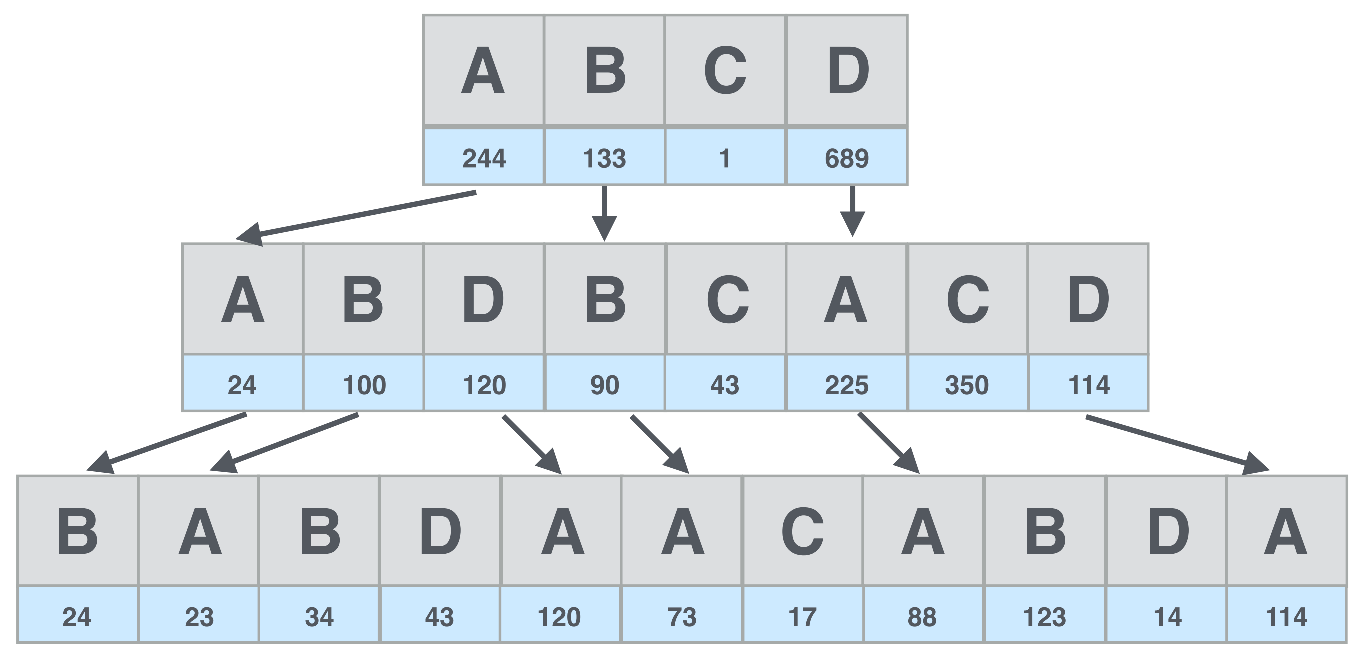
Hash-based VS **Trie-based**

Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.



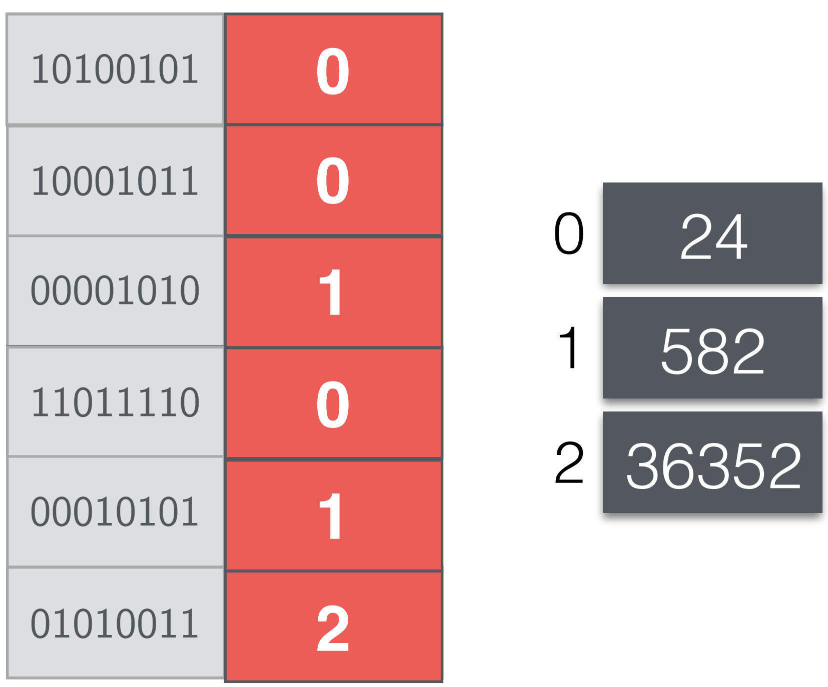


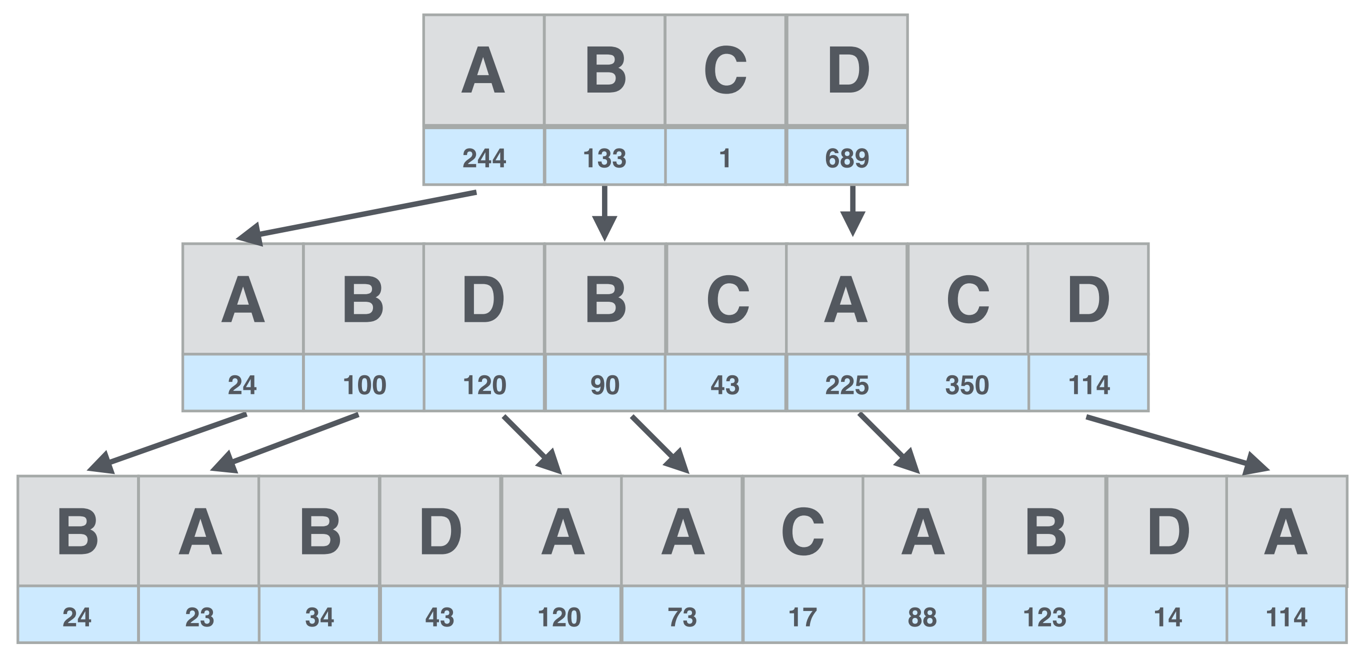
Hash-based VS **Trie-based**

Open addressing?

Data structure is *static*.

Minimal Perfect Hashing.



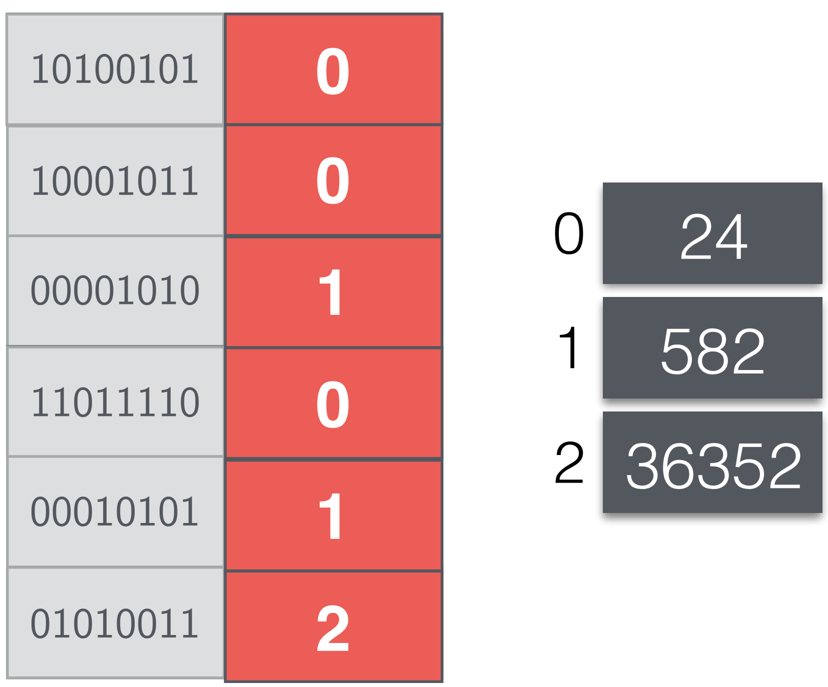


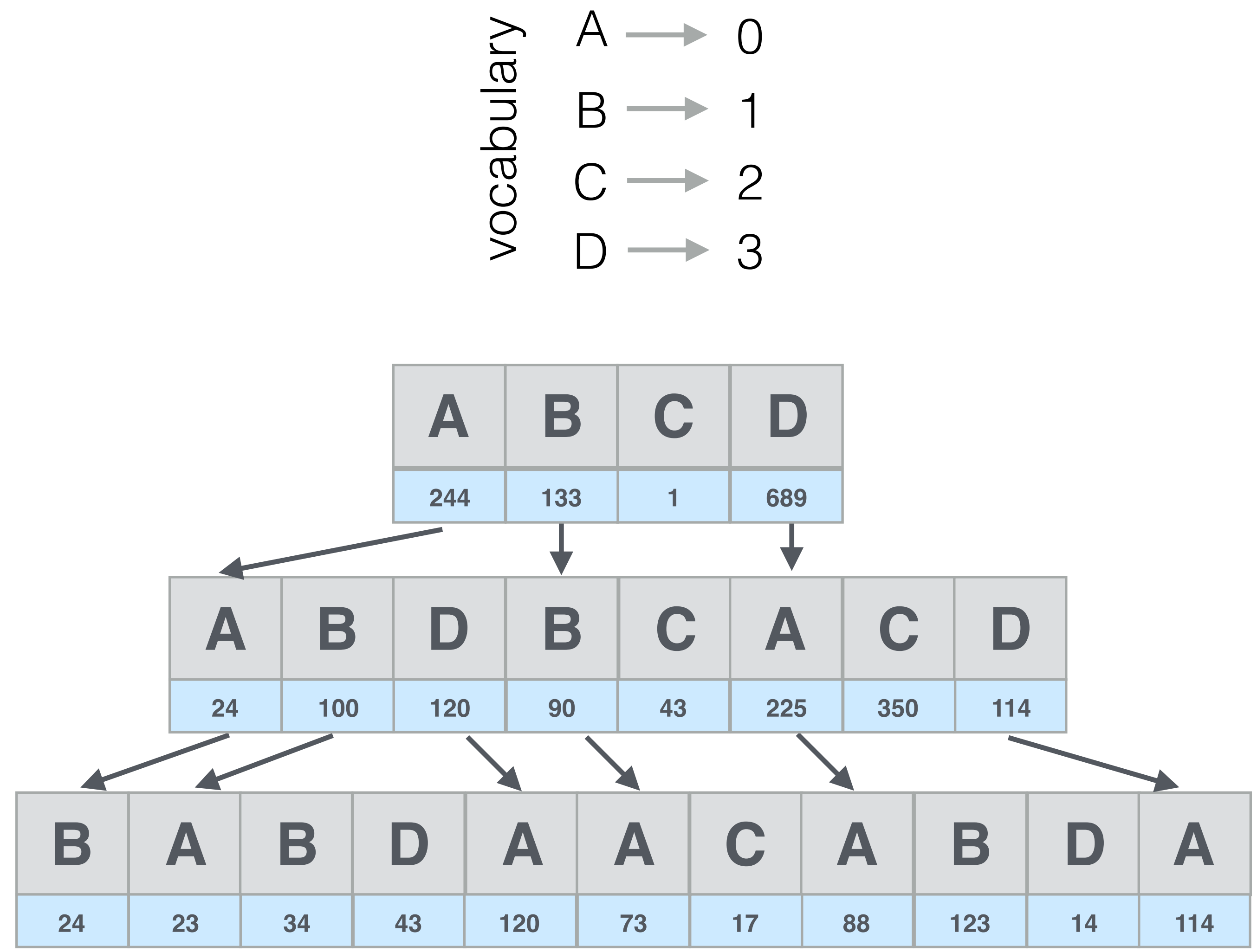
Hash-based VS Trie-based

Open addressing?

Data structure is *static*.

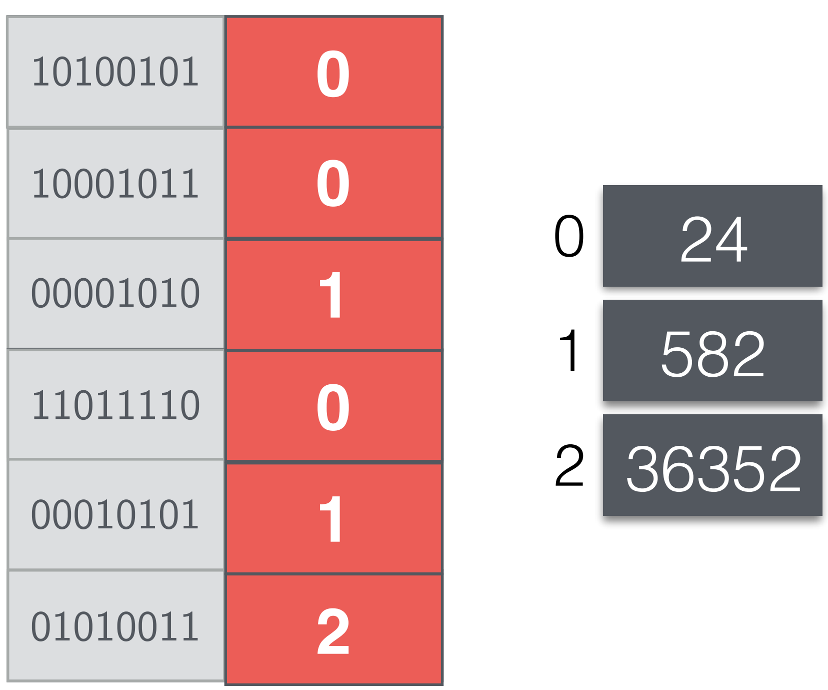
Minimal Perfect Hashing.

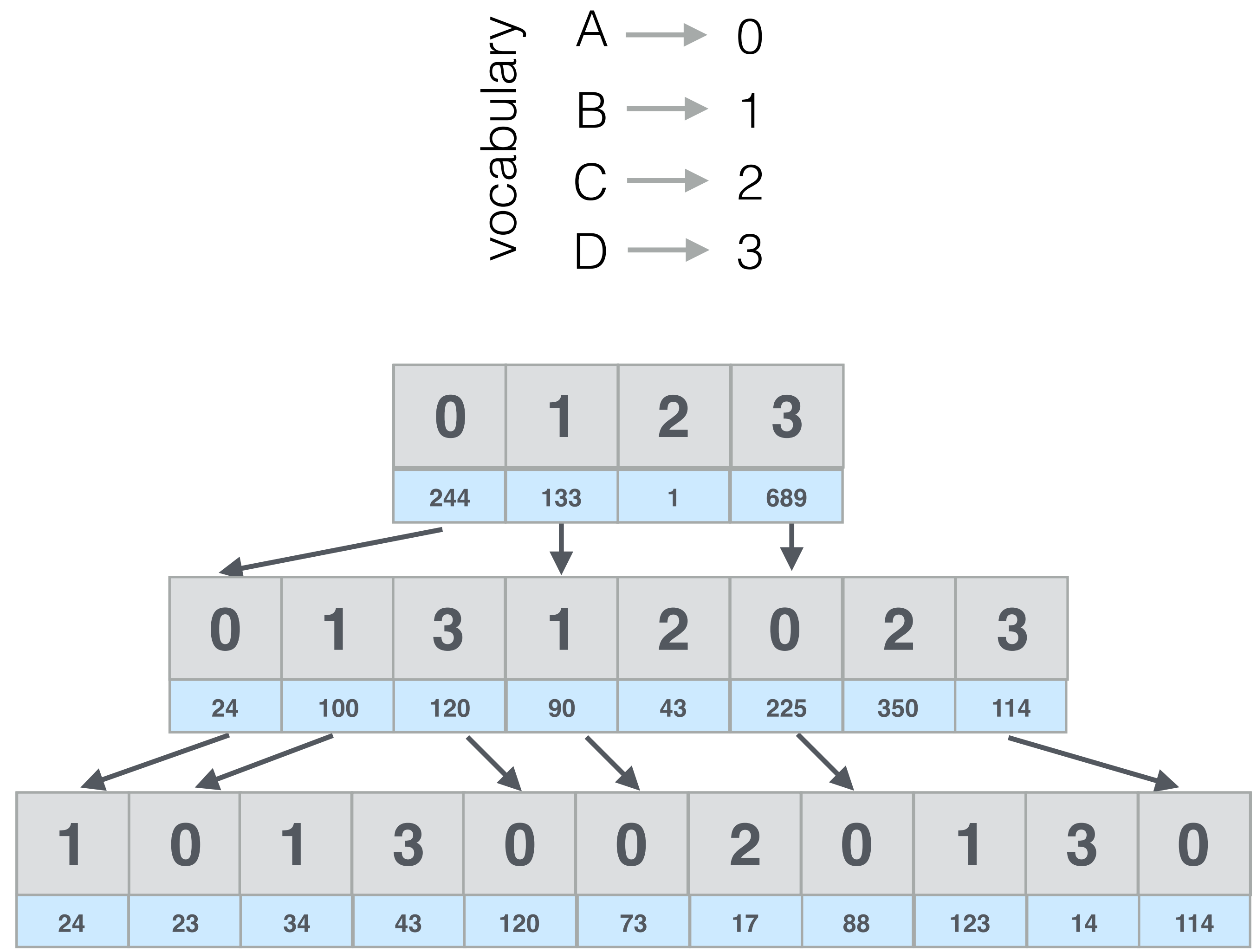




Hash-based VS **Trie-based**

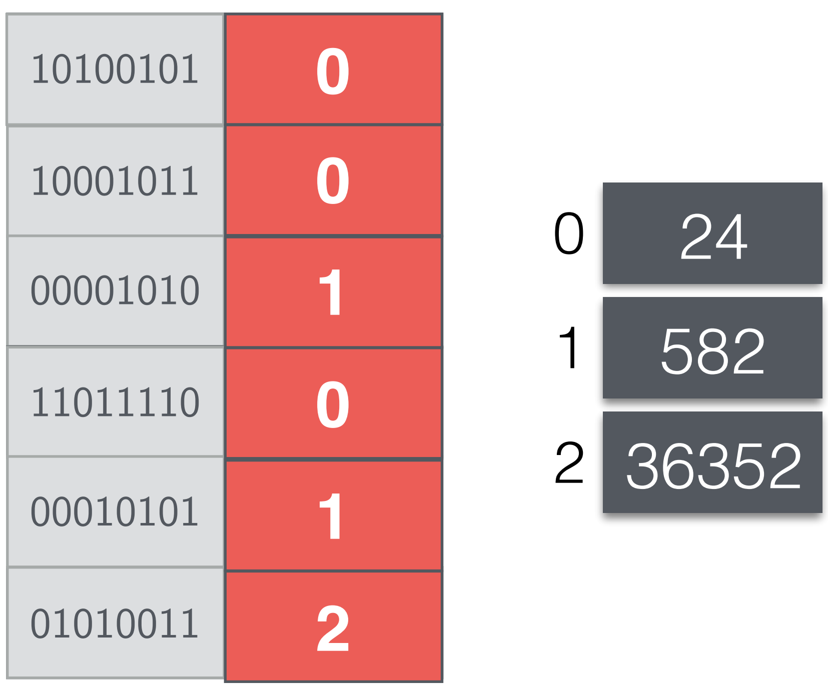
Open addressing?
Data structure is *static*.
Minimal Perfect Hashing.



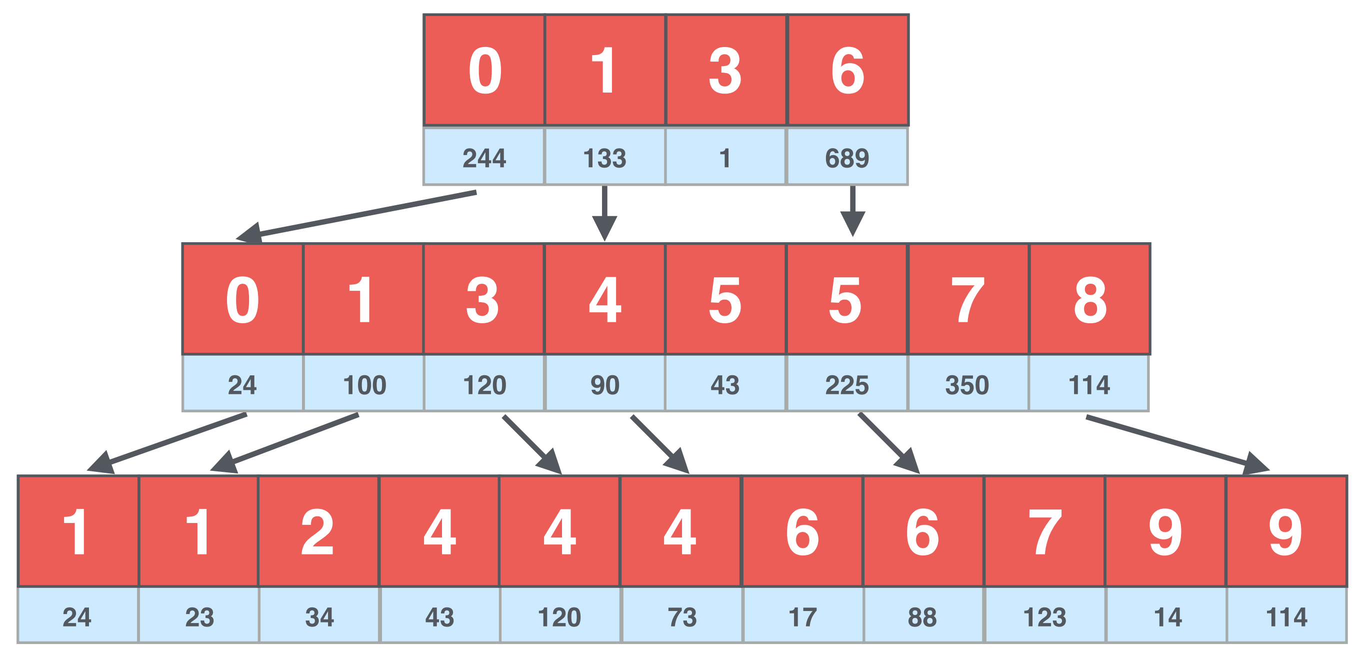


Hash-based VS **Trie-based**

Open addressing?
Data structure is *static*.
Minimal Perfect Hashing.



vocabulary
A → 0
B → 1
C → 2
D → 3



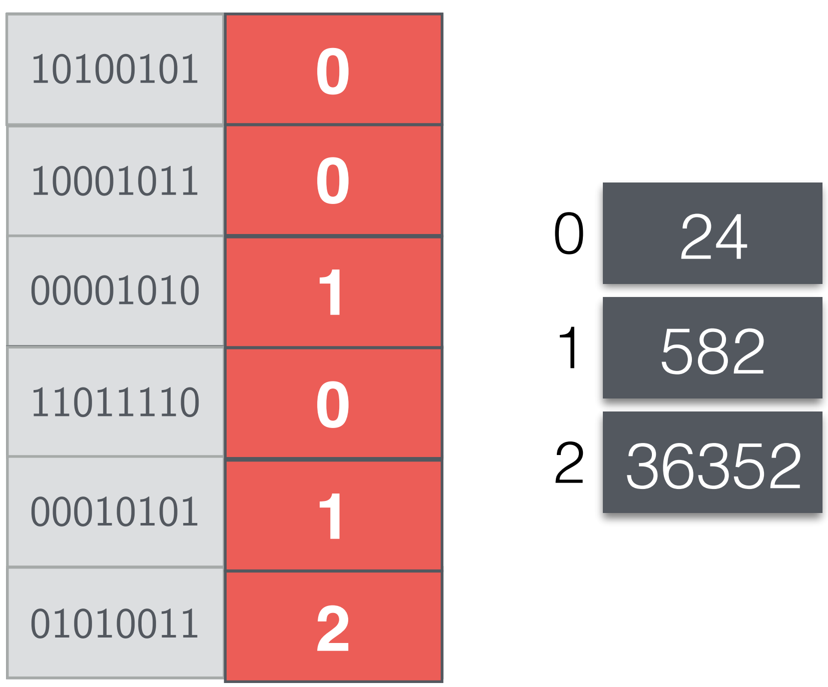
Hash-based

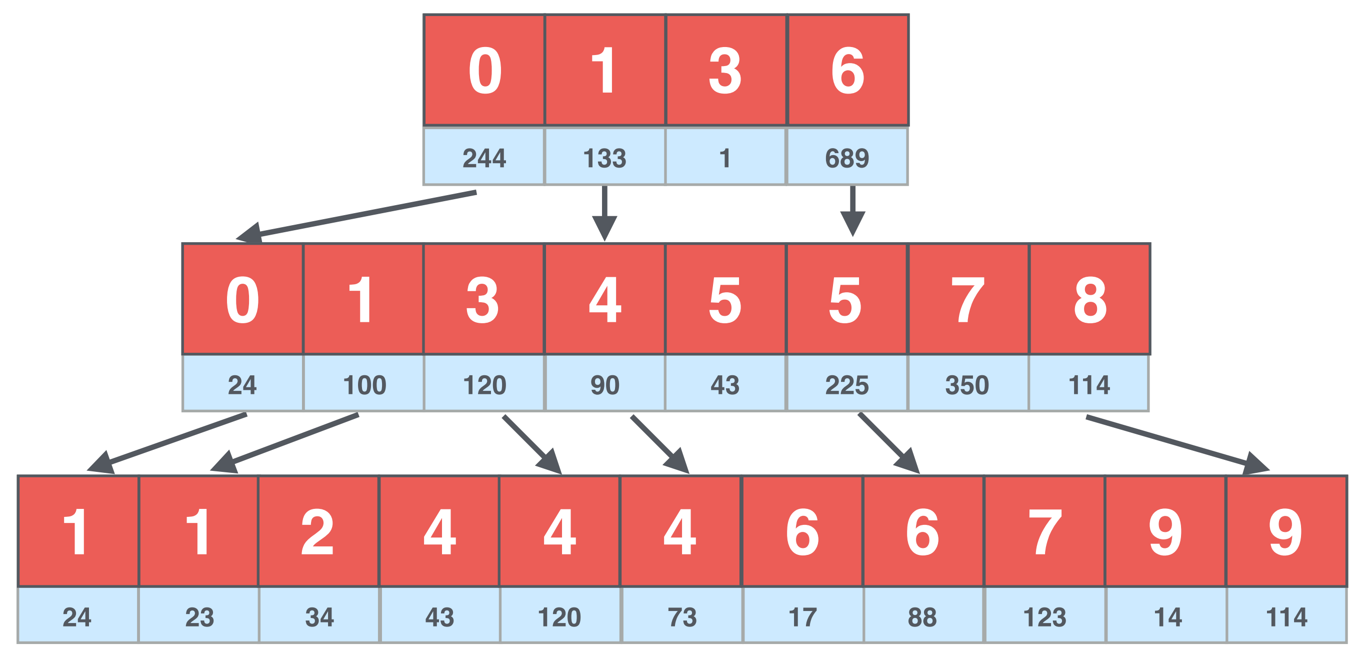
VS

Trie-based

Open addressing?
Data structure is *static*.
Minimal Perfect Hashing.

vocabulary
A → 0
B → 1
C → 2
D → 3





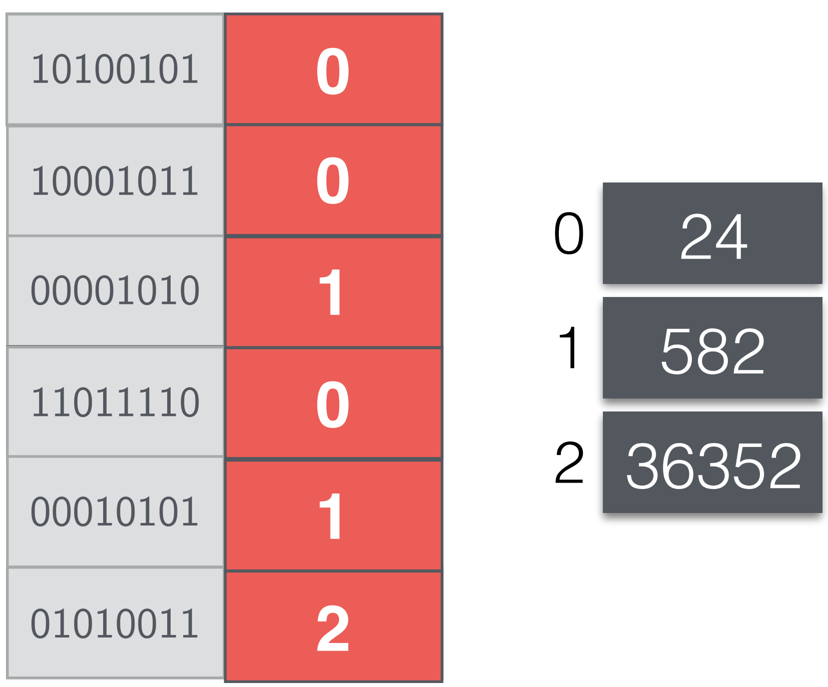
Encode each level
with **Elias-Fano**.

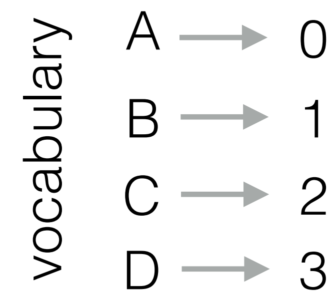
Hash-based

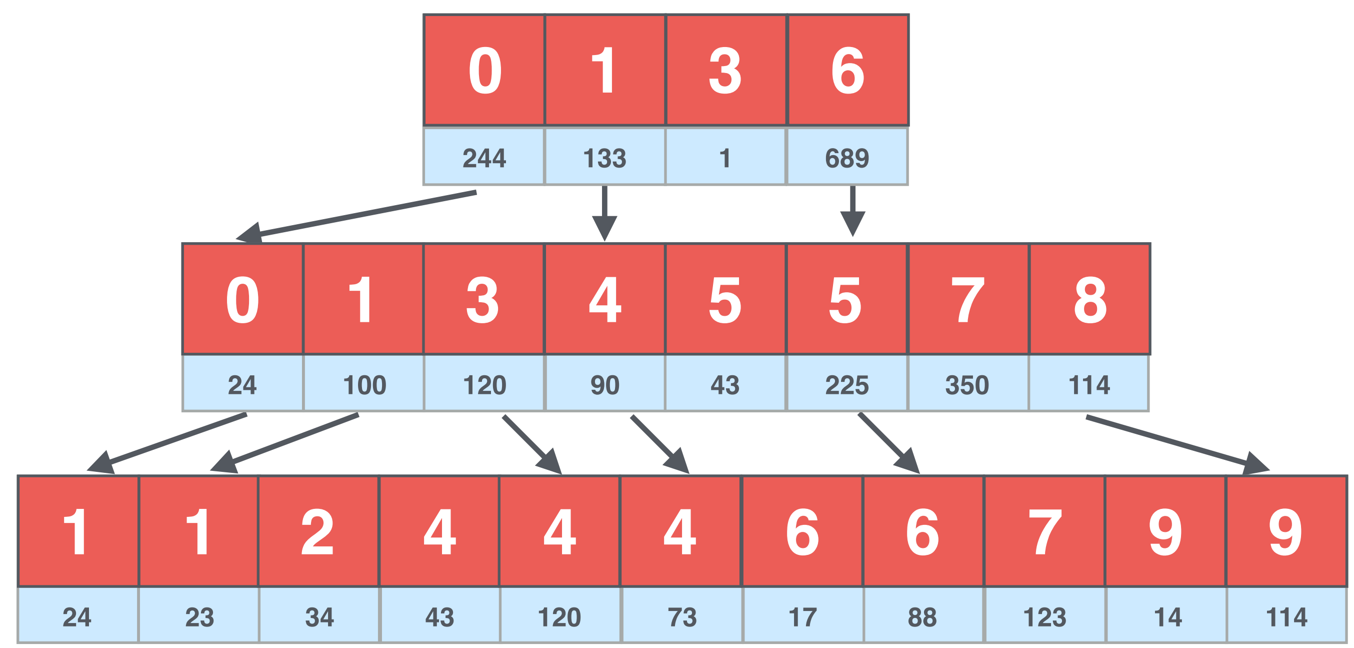
VS

Trie-based

Open addressing?
Data structure is *static*.
Minimal Perfect Hashing.







Encode each level
with **Elias-Fano**.

Random access.

tongrams - Preliminary results

n	Number of n-grams	Maximum frequency count	Unique frequency counts	$\lceil \lg \rceil$ of unique frequency counts
1	24,359,472	468,491,999,592	246,588	18
2	5,089,239	155,178,163	44,822	16
3	52,635,338	102,329,901	71,690	17
4	11,149,161	6,401,274	21,127	15
5	8,261,975	958,556	12,171	14
Total	101,495,185	468,491,999,592	266,760	19

Table 4: Basic statistics for the GoogleWeb1T subset.

		Total space in GBs	Bytes per gram		Lookup time [μs]	
HASH	KenLM	2.570	27.19		0.248	
	sxlm	**1.012**	**10.43**	(−61.64%)	**0.242**	(−2.42%)
TRIE	KenLM	1.829	21.5		1.272	
	sxlm	**0.541**	**5.7**	(−73.34%)	**1.229**	(−3.38%)

Table 5: Bytes per grams and average lookup time in μs for the GoogleWeb1T subset.

tongrams - Preliminary results 2

n	Number of n-grams	Maximum frequency count	Unique frequency counts	$\lceil \lg \rceil$ of unique frequency counts
1	24,359,472	468,491,999,592	246,588	18
2	5,089,239	155,178,163	44,822	16
3	52,635,338	102,329,901	71,690	17
4	11,149,161	6,401,274	21,127	15
5	8,261,975	958,556	12,171	14
Total	101,495,185	468,491,999,592	266,760	19

Table 4: Basic statistics for the GoogleWeb1T subset.

		Total space in GBs	Bytes per gram		Lookup time [μs]	
HASH	KenLM	2.570	27.19		0.248	
	sxlm	**1.012**	**10.43**	(−61.64%)	**0.242**	(−2.42%)
TRIE	KenLM	1.829	21.5		1.272	
	sxlm	**0.541**	**5.7**	(−73.34%)	**1.229**	(−3.38%)

Table 5: Bytes per grams and average lookup time in μs for the GoogleWeb1T subset.

tongrams - Preliminary results

n	Number of n-grams	Maximum frequency count	Unique frequency counts	$\lceil \lg \rceil$ of unique frequency counts
1	24,359,472	468,491,999,592	246,588	18
2	5,089,239	155,178,163	44,822	16
3	52,635,338	102,329,901	71,690	17
4	11,149,161	6,401,274	21,127	15
5	8,261,975	958,556	12,171	14
Total	101,495,185	468,491,999,592	266,760	19

Table 4: Basic statistics for the GoogleWeb1T subset.

		Total space in GBs	Bytes per gram		Lookup time [μs]	
HASH	KenLM	2.570	27.19	X2.6	0.248	
	sxlm	**1.012**	**10.43**	(−61.64%)	**0.242**	(−2.42%)
TRIE	KenLM	1.829	21.5	X3.8	1.272	
	sxlm	**0.541**	**5.7**	(−73.34%)	**1.229**	(−3.38%)

Table 5: Bytes per grams and average lookup time in μs for the GoogleWeb1T subset.

(Some) Future Research Problems

Dynamic Inverted Indexes.

Dynamic Inverted Indexes.

Classic solution: use two indexes.
One is big and **cold**; the other is small and **hot**.
Merge them periodically.

(Some) Future Research Problems 1

Dynamic Inverted Indexes.

Classic solution: use two indexes.
One is big and **cold**; the other is small and **hot**.
Merge them periodically.

(Some) Future Research Problems 2

Compressed B-trees.

Compressed B-trees.

Problem: maintain a dictionary on disk.

Motivations: databases and file-systems.

Compressed B-trees.

Problem: maintain a dictionary on disk.

Motivations: databases and file-systems.

"Fancy indexing structures may be a luxury now,
but they will be essential by the decade's end."

Compressed B-trees.

Problem: maintain a dictionary on disk.

Motivations: databases and file-systems.

"Fancy indexing structures may be a luxury now, but they will be essential by the decade's end."

Michael Bender

Stony Brook University

Martin Farach-Colton

Rutgers University

Bradley Kuszmaul

MIT Laboratory for Computer Science

Tokutek®

(Some) Future Research Problems

Fast Successor for IP-lookup.

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

0 0 0 0 1 1
0 0 0 1 0 0
0 0 0 1 1 1
0 0 1 1 0 1
0 0 1 1 1 0
0 0 1 1 1 1
0 1 0 1 0 1
1 0 1 0 1 1

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

1110111010001000

Build an index on zeros.

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

1110111010001000

Build an index on zeros.

$$\mathbf{p = select_0(h_x) - h_x}$$

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

11101111010001000

Build an index on zeros.

$p = \text{select}_0(h_x) - h_x$

x = 001100 (12)

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

1110111010001000

Build an index on zeros.

x = 001100 (12)

$$p = \mathrm{select}_0(h_x) - h_x$$

Fast Successor for IP-lookup.

Successor search is what routers do for every incoming packet.

Hence, the most run algorithm in the world.

Time *and* space efficiency is crucial.

We can directly jump to the position of the first address having the same *lg n* bits as the searched pattern in O(1) using the powerful search capabilities of Elias-Fano.

0 0 0	0 1 1
0 0 0	1 0 0
0 0 0	1 1 1
0 0 1	1 0 1
0 0 1	1 1 0
0 0 1	1 1 1
0 1 0	1 0 1
1 0 1	0 1 1

1110111010001000

Build an index on zeros.

x = 001100 (12)

$$\mathbf{p = select_0(h_x) - h_x}$$

Thanks for your attention, time, patience!

Any questions?