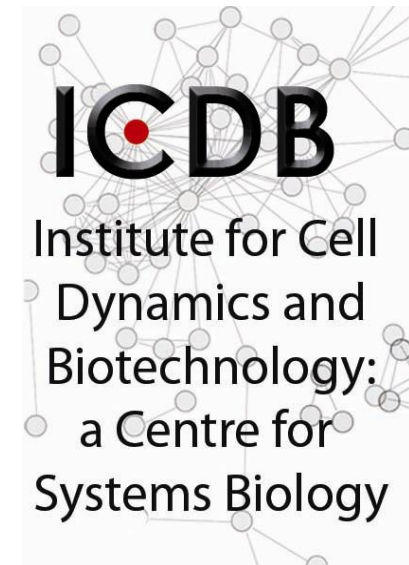


Space-Efficient Data Structures

Francisco Claude
Gonzalo Navarro



Outline

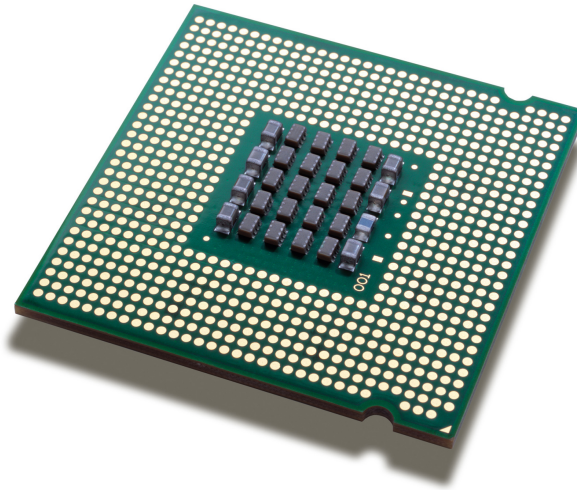
- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation ←
- Basics
- Bitmaps
- Sequences
- Applications

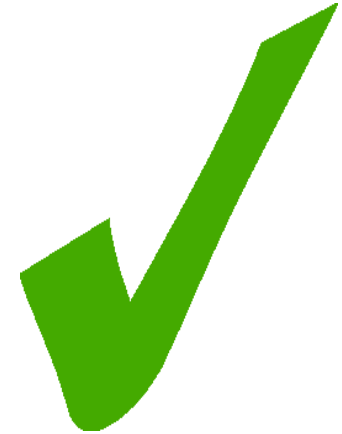
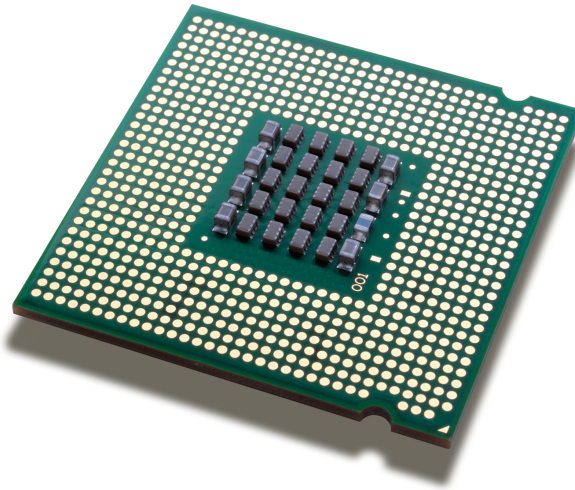
Motivation

- Processor speed increasing



Motivation

- Processor speed increasing

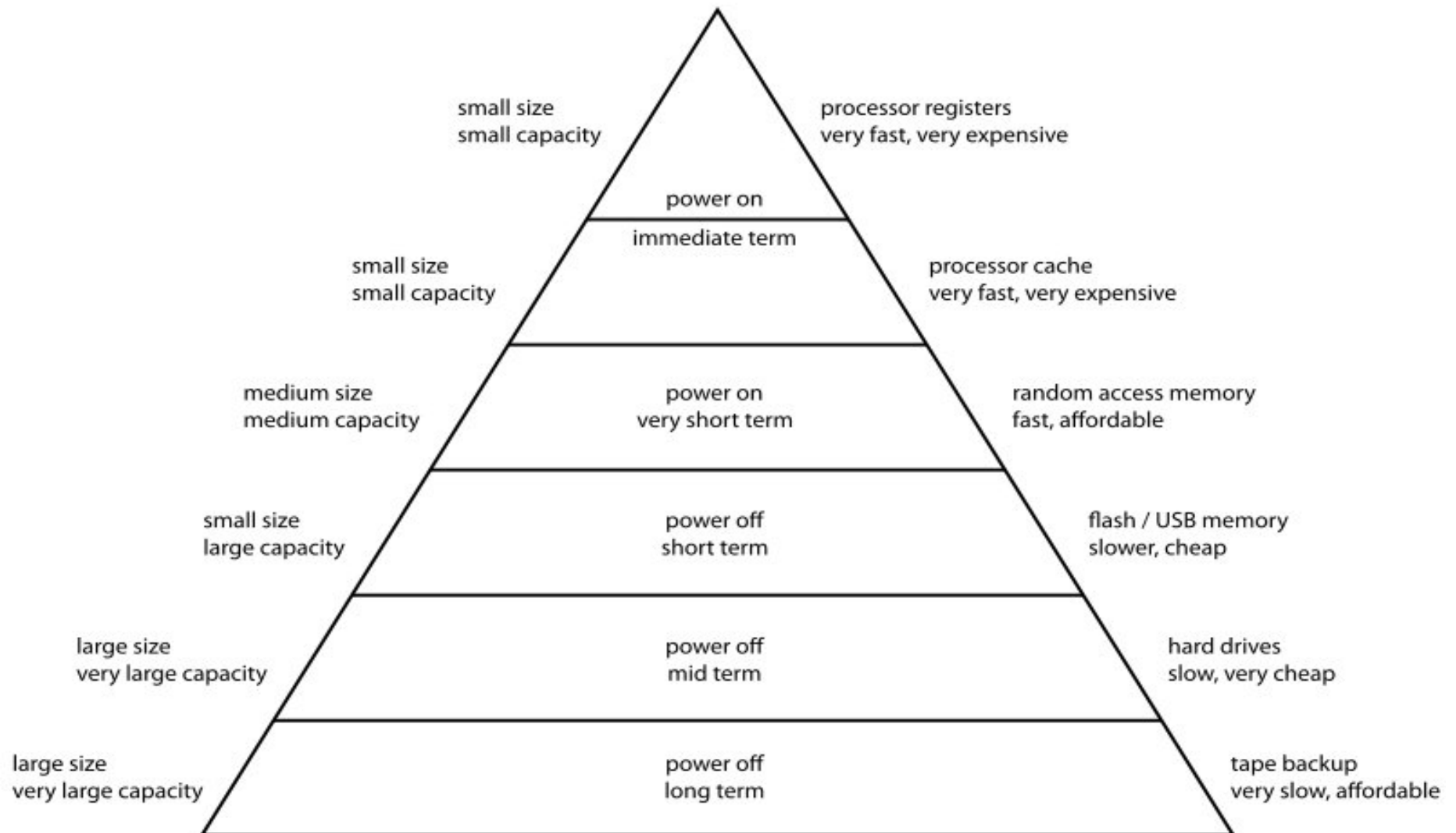


- Disks have not evolved at the same pace



Motivation

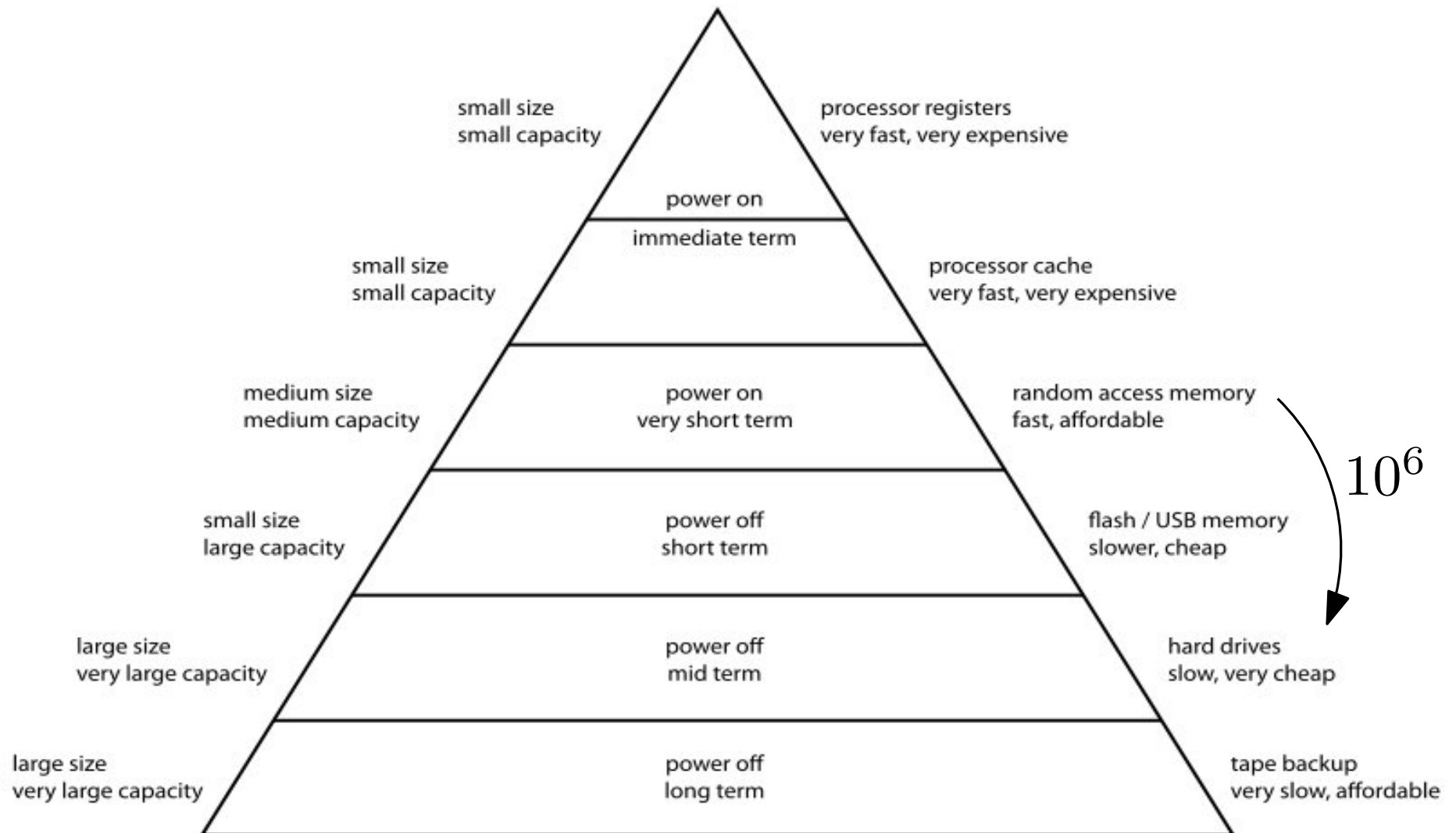
Computer Memory Hierarchy



source: Wikipedia

Motivation

Computer Memory Hierarchy



source: Wikipedia

Motivation

Web Graph

uk-union-2006-06-2007-05

Nodes: 133, 633, 040

Edges: 5, 507, 679, 822

Motivation

Web Graph

uk-union-2006-06-2007-05

Nodes: 133, 633, 040

Edges: 5, 507, 679, 822

A plain representation
requires 22GB!

Motivation

Web Graph

uk-union-2006-06-2007-05

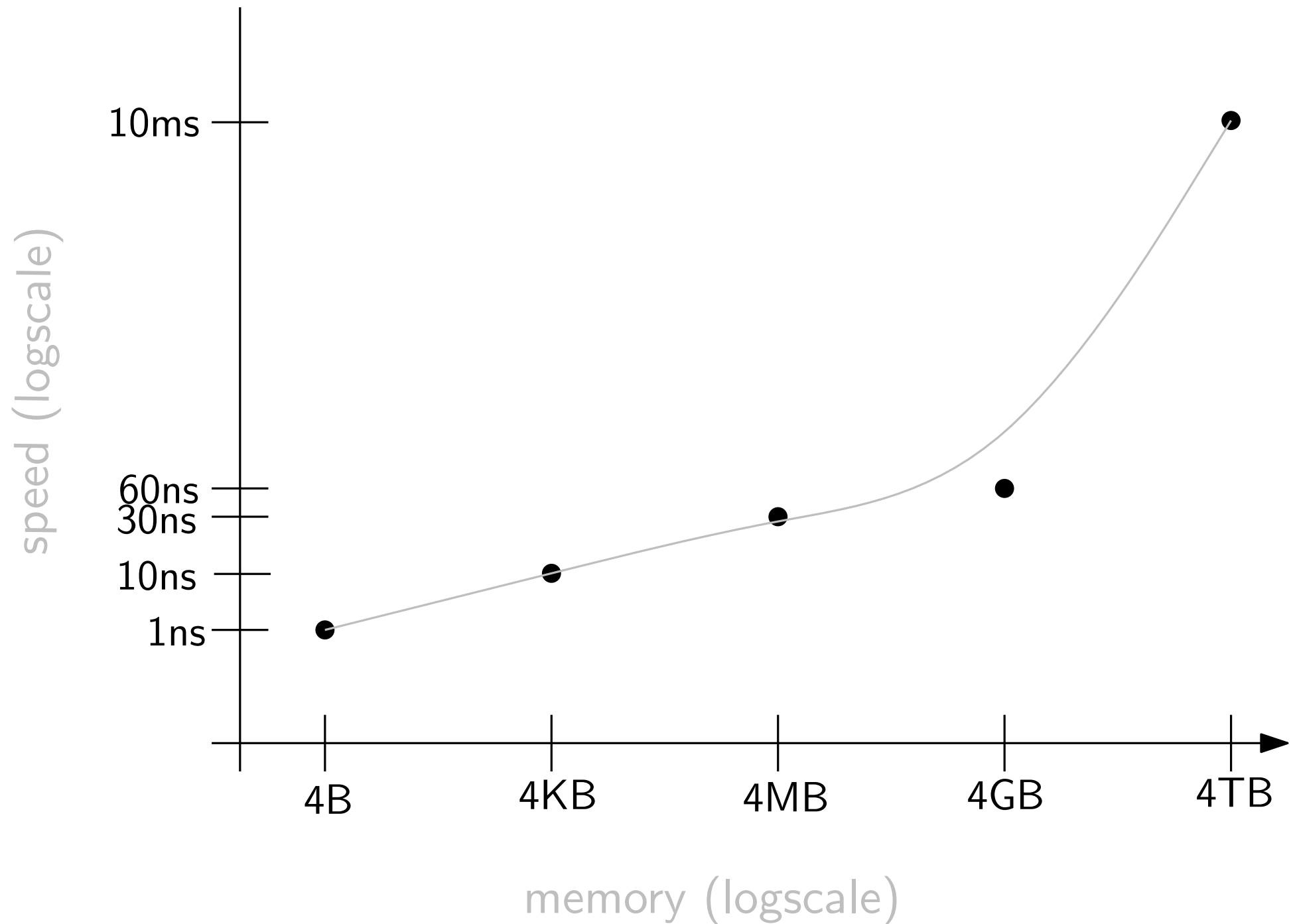
Nodes: 133, 633, 040

Edges: 5, 507, 679, 822

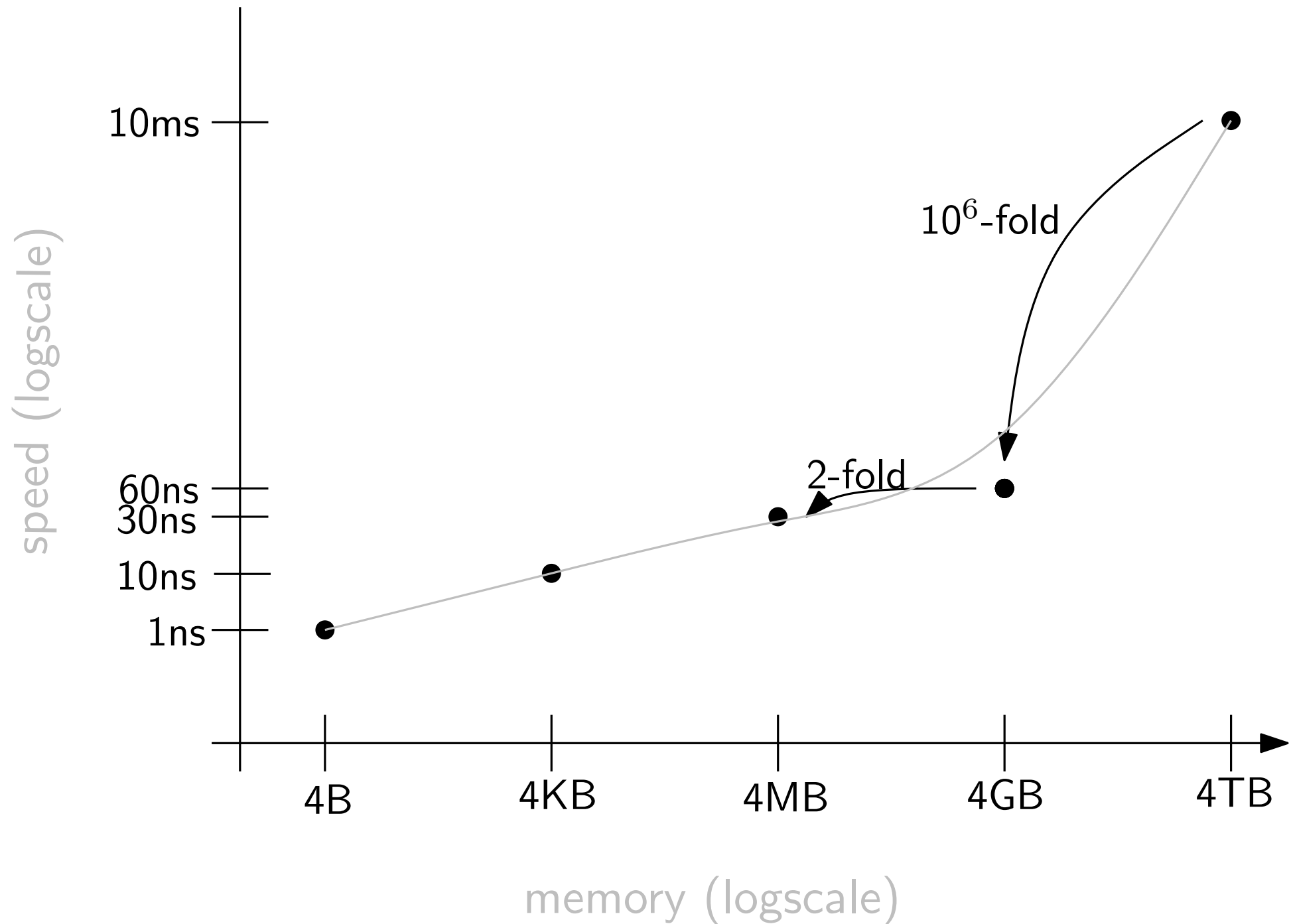
A plain representation
requires 22GB!

If we use a space-efficient representation: $< 2\text{GB}$

Motivation




Motivation



Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation
- Basics 
- Bitmaps
- Sequences
- Applications

Basics

- Plain representation of data
- Zero-order compression
- High-order compression

Plain Representation of Data

Array:

- length n
- maximum value m

$$n \lceil \log_2(m + 1) \rceil \text{ bits}$$

Plain Representation of Data

Array:

- length n
- maximum value m

$$n \lceil \log_2(m + 1) \rceil \text{ bits}$$

5	3	4	1	0	4	2	4	1	0
101	011	100	001	000	100	010	100	001	000

On a 32-bits machine this requires 1 word

Arrays in LIBCDS

```
size_t N;  
cout << "Enter array length: ";  
cin >> N;  
uint * A = new uint[N];  
for(size_t i=0;i<N;i++) {  
    cout << "Enter element at position " << i << ": ";  
    cin >> A[i];  
}
```

```
Array * a = new Array(A,N);  
delete [] A;
```

```
cout << "Size: " << a->getSize() << " bytes" << endl;  
for(uint i=0;i<N;i++)  
    cout << "A[" << i << "]= " << a->getField(i) << endl;  
  
delete a;
```

Arrays in LIBCDS

```
size_t N;
uint M;
cout << "Enter array length: ";
cin >> N;
cout << "Enter the maximum value to be stored: ";
cin >> M;
Array *a = new Array(N,M);
for(size_t i=0;i<N;i++) {
    uint tmp;
    cout << "Enter element at position " << i << ": ";
    cin >> tmp;
    a->setField(i,tmp);
}

cout << "Size: " << a->getSize() << " bytes" << endl;
for(uint i=0;i<N;i++)
    cout << "A[" << i << "]= " << a->getField(i) << endl;
delete a;
```

Zero-order Compression

Can we do better? It depends

$S = \text{aaabbcaaabbcaaad}$

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$$

$$H_0(S) = 1.5919$$

symbol	n_{symbol}
a	9
b	4
c	2
d	1

Zero-order Compression

Can we do better? It depends

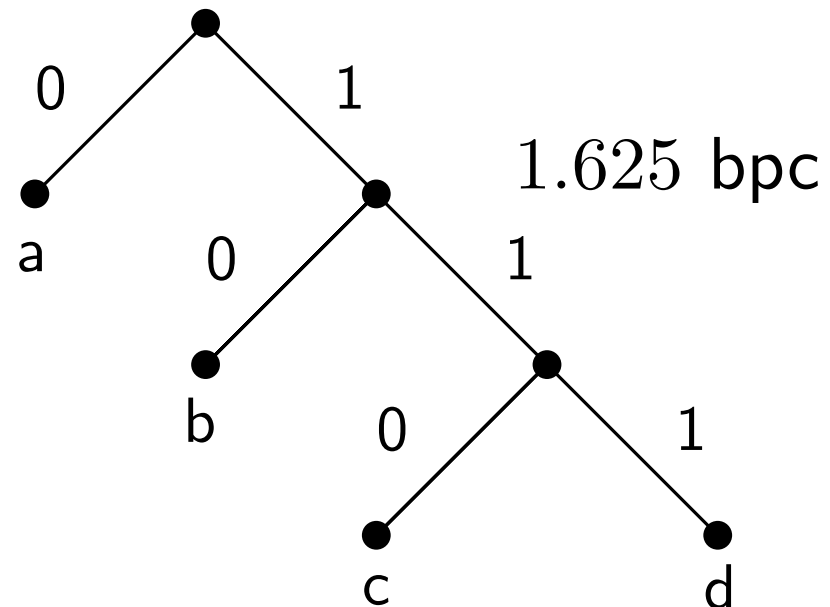
$S = \text{aaabbcaaabbcaaad}$

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c}$$

$$H_0(S) = 1.5919$$

symbol	n_{symbol}
a	9
b	4
c	2
d	1

Huffman



High-order Compression

Can we exploit other things?

$S = \text{aaabbcaaabbcaaad}$

High-order Compression

Can we exploit other things?

$S = \text{aaabbcaaabbcaaad}$

Yes, for example $P(a|b) = 0$

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

$$H_k(S) = 0.9387$$

Things to Remember

- $\lceil \log_2(m+1) \rceil$ bits to represent a number $\leq m$
- Compression: H_0 and H_k

$$H_k \leq H_0 \leq \log_2 m$$

Things to Remember

- $\lceil \log_2(m+1) \rceil$ bits to represent a number $\leq m$
- Compression: H_0 and H_k

$$H_k \leq H_0 \leq \log_2 m$$

One step forward: ordinal trees

$$C_n = \frac{1}{n+1} \binom{2n}{n} \approx \frac{4^n}{n^{3/2} \sqrt{\pi}}$$

$$\Rightarrow 2n + o(n) \text{ bits}$$

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

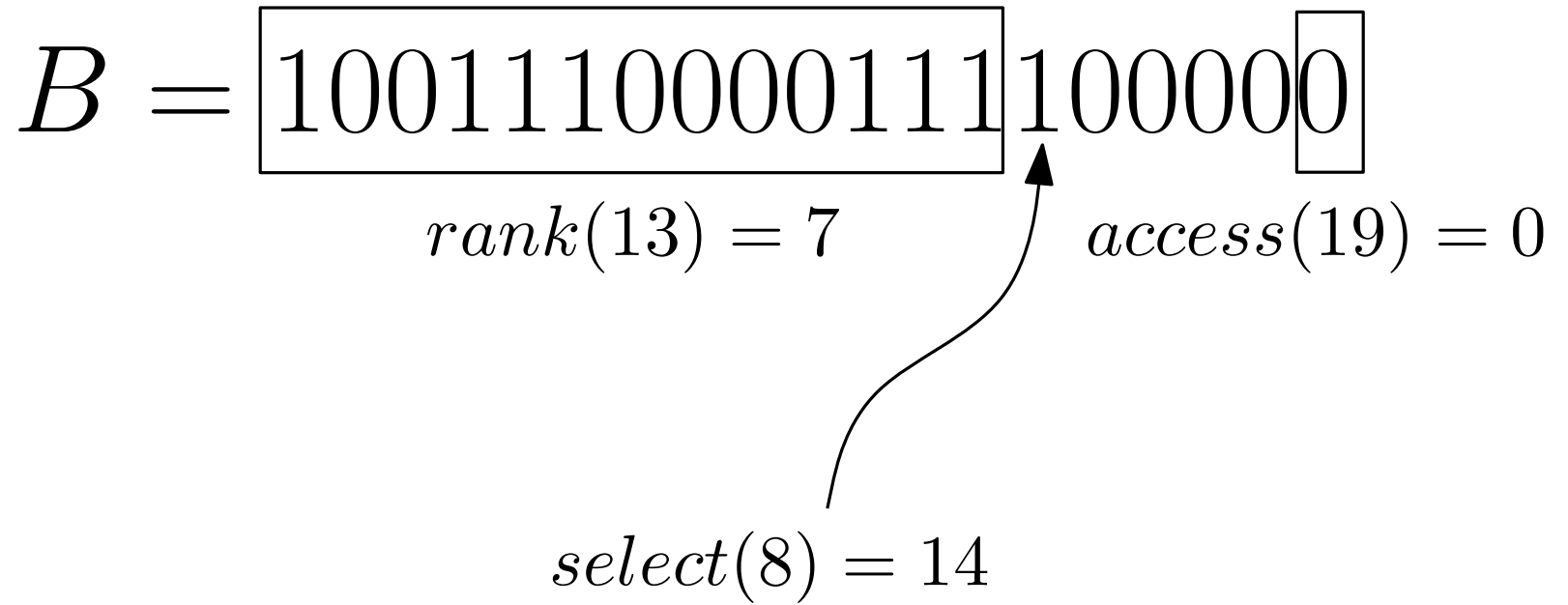
Outline

- Motivation
- Basics
- Bitmaps ←
- Sequences
- Applications

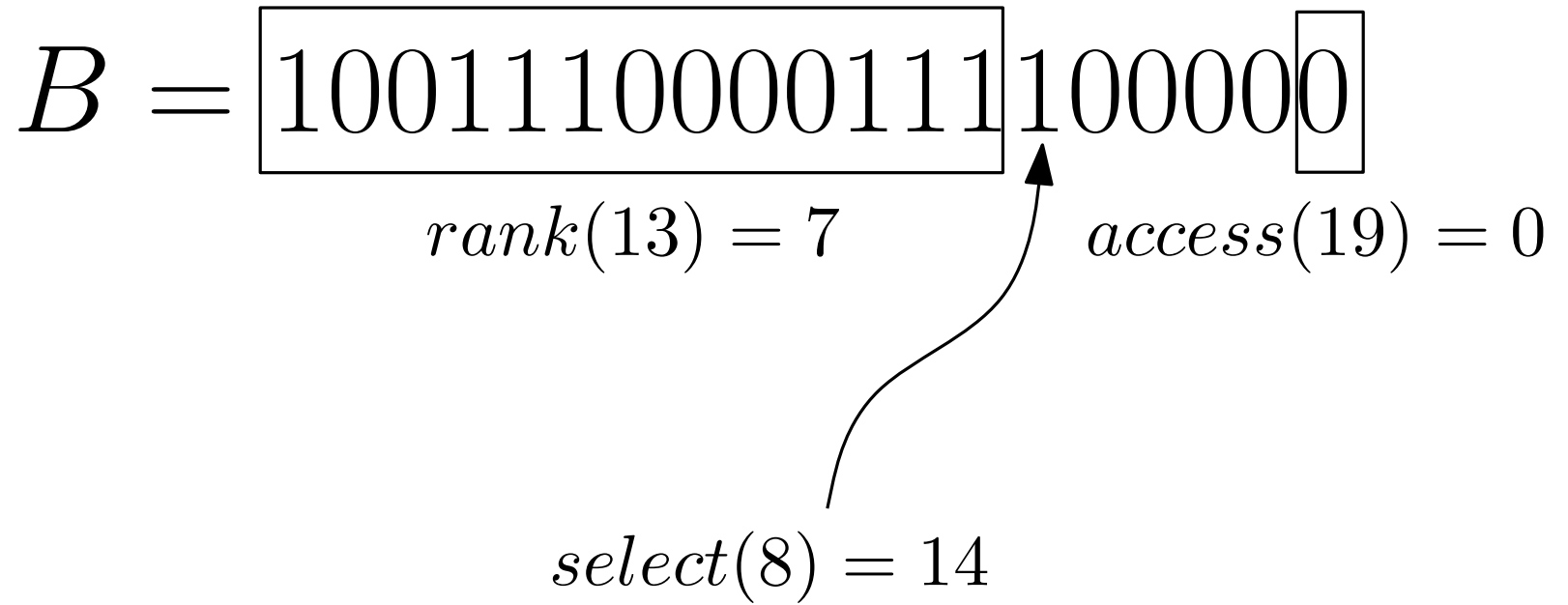
Bitmaps

$$B = 1001110000111100000$$

Bitmaps



Bitmaps



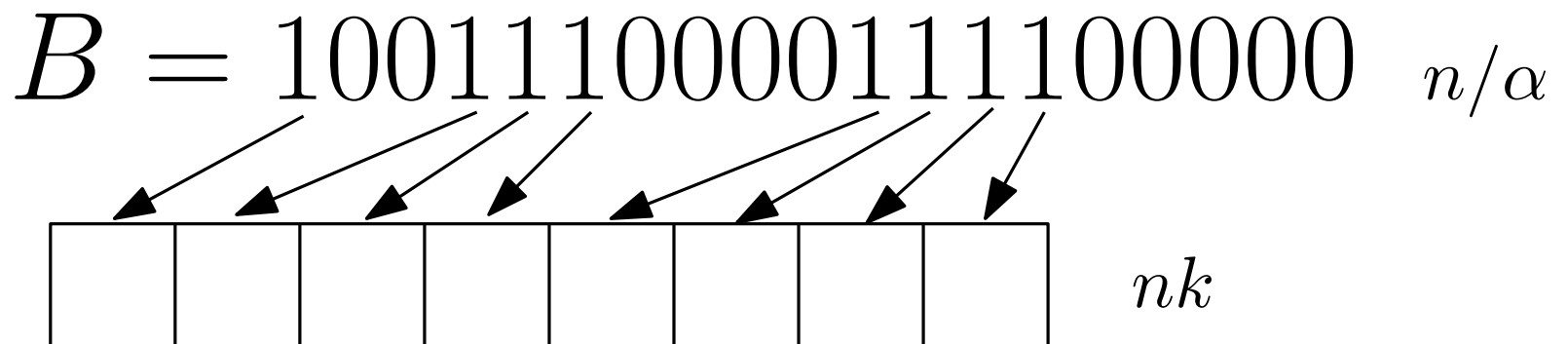
Is this useful?

Hashing Example

- Static hashtable with n elements implemented with linear probing
- Expected successful search cost = 3 ($\alpha = 0.8$)
- Each key requires k bits

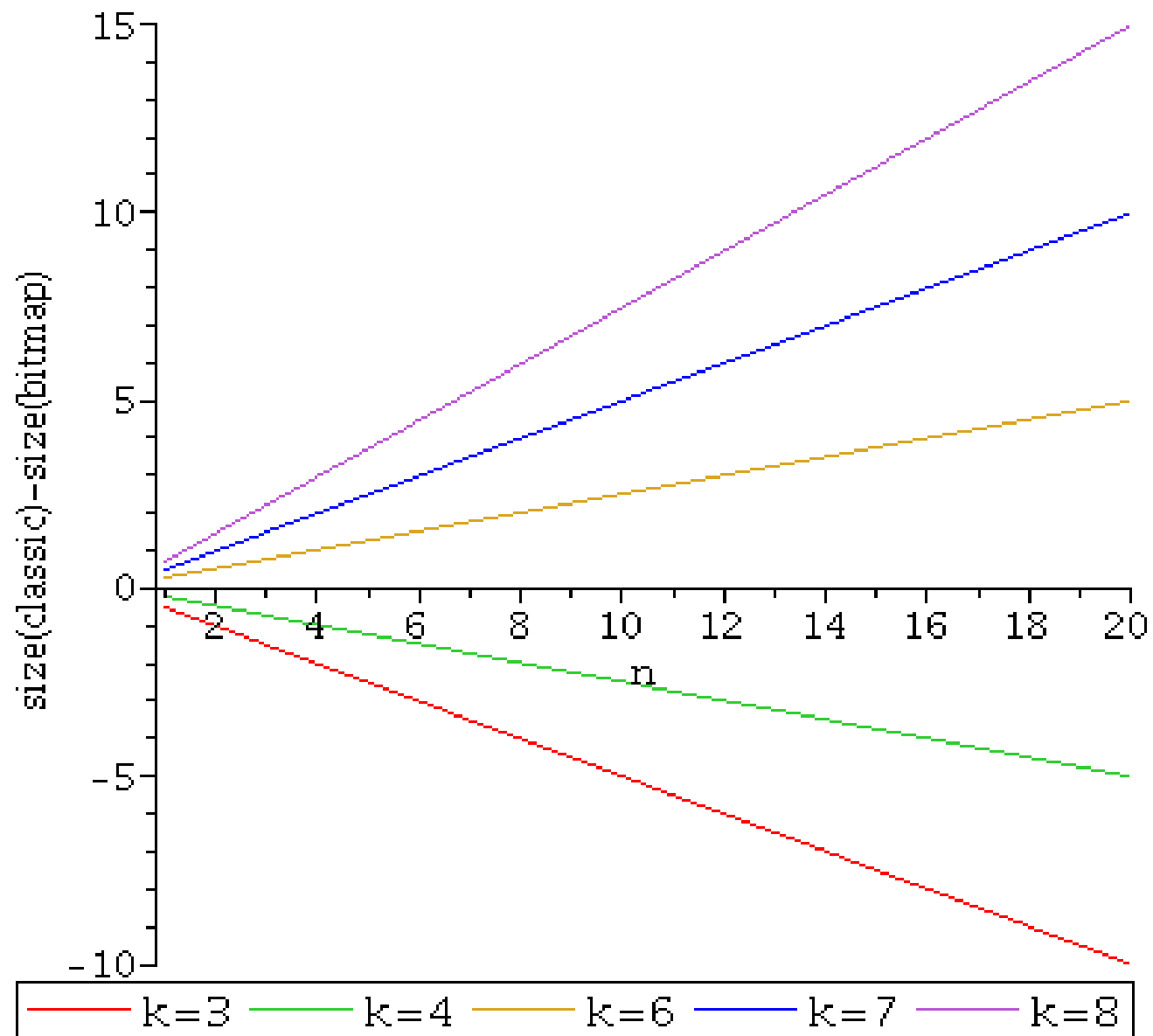


Standard solution requires $k \frac{n}{\alpha}$ bits



Requires $\frac{n}{\alpha} + nk$ bits

Hashing Example



Bitmaps: Example Applications

- Linear-probing hashing
- Perfect hashing
- Bitmaps are basic building blocks
- Partial Sums

Bitmaps

- Plain [Jacobson, Clark and Munro]
- Compressed [Raman et al.]
- Very Sparse [Okanohara and Sadakane]

Rank

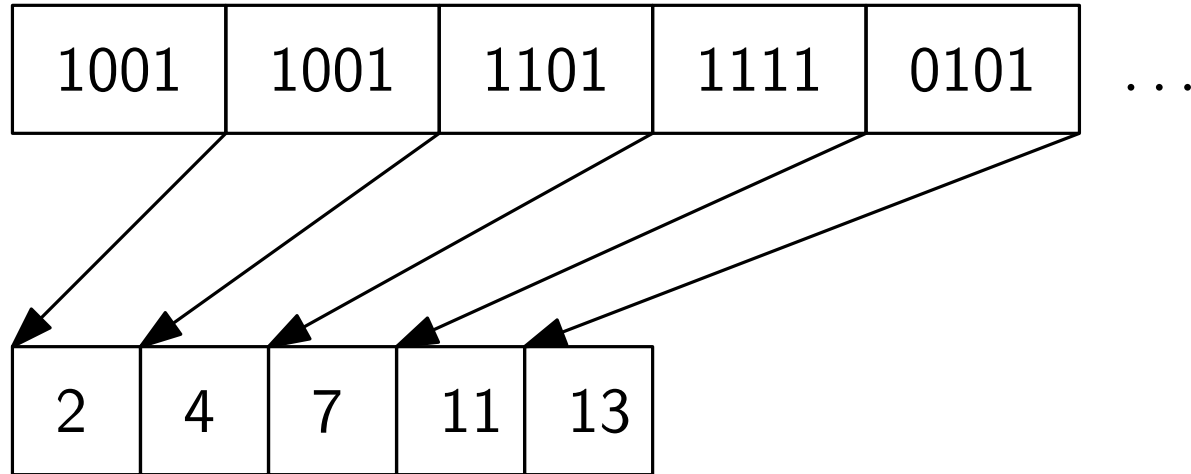
$$B = 1001010101110101011\dots$$

1	1	1	2	2	3	3	4	4	5	6	7	7	...
---	---	---	---	---	---	---	---	---	---	---	---	---	-----

$O(1)$ rank

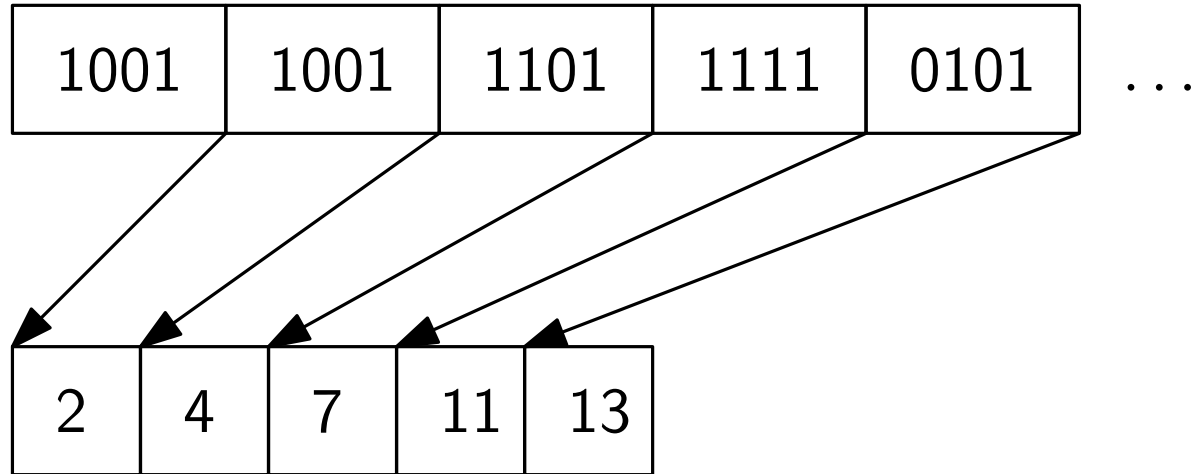
$n \lg n$ bits of space

Rank



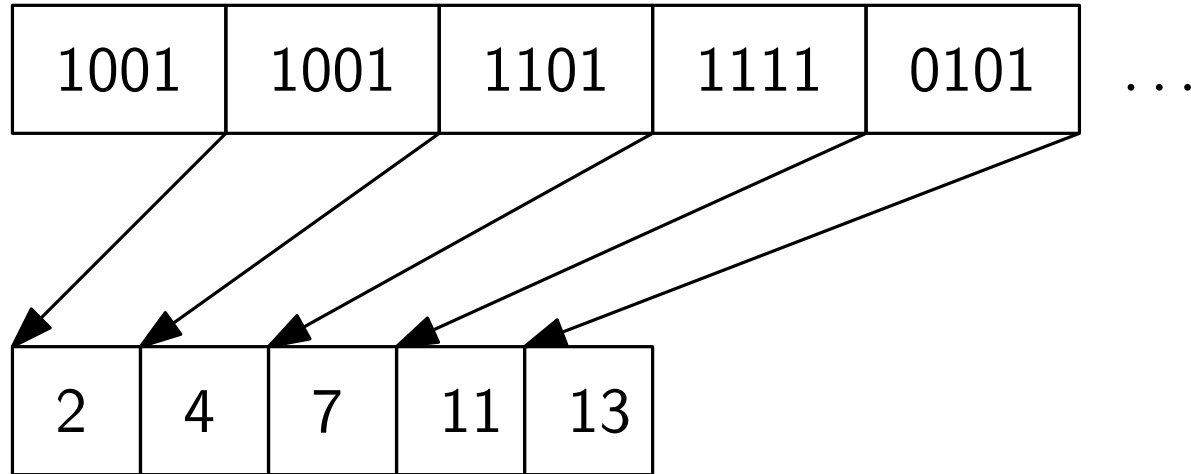
If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

Rank

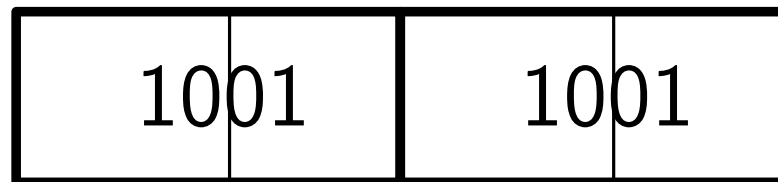


If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

Rank

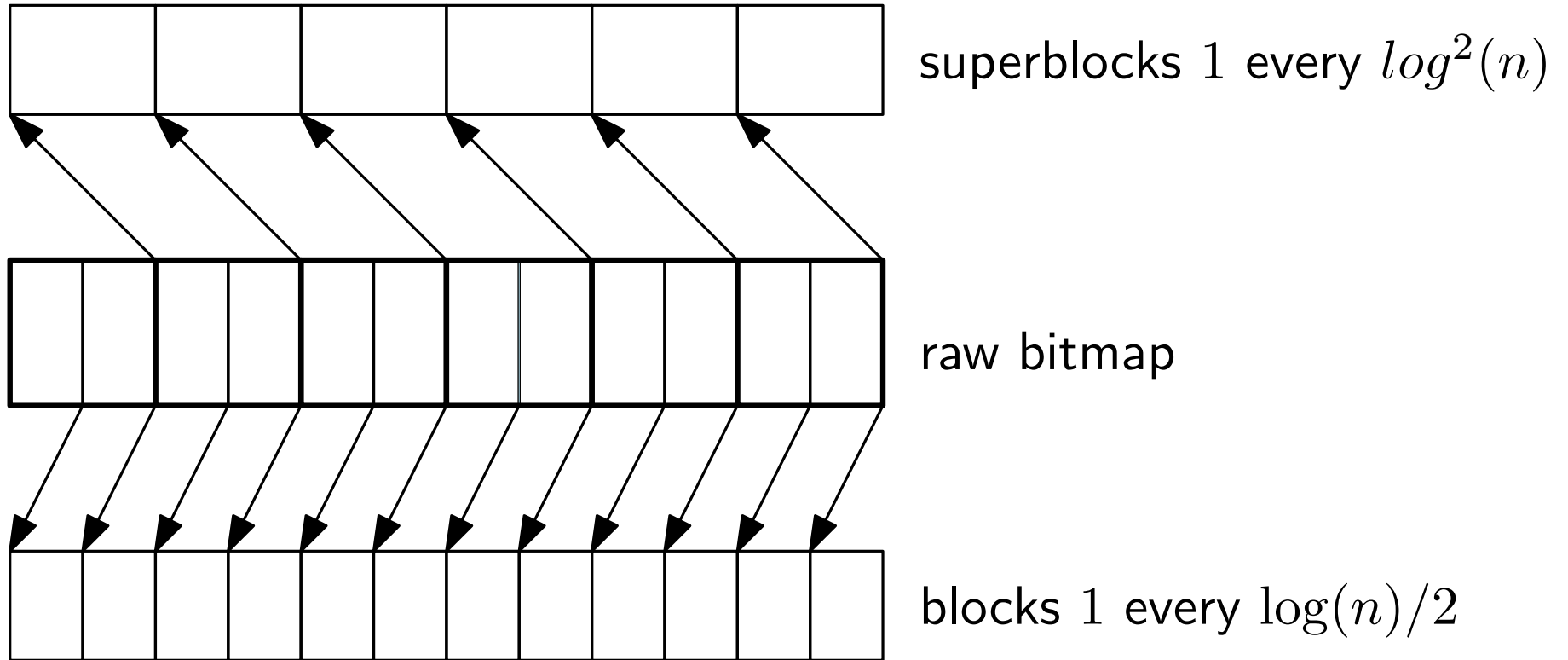


If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

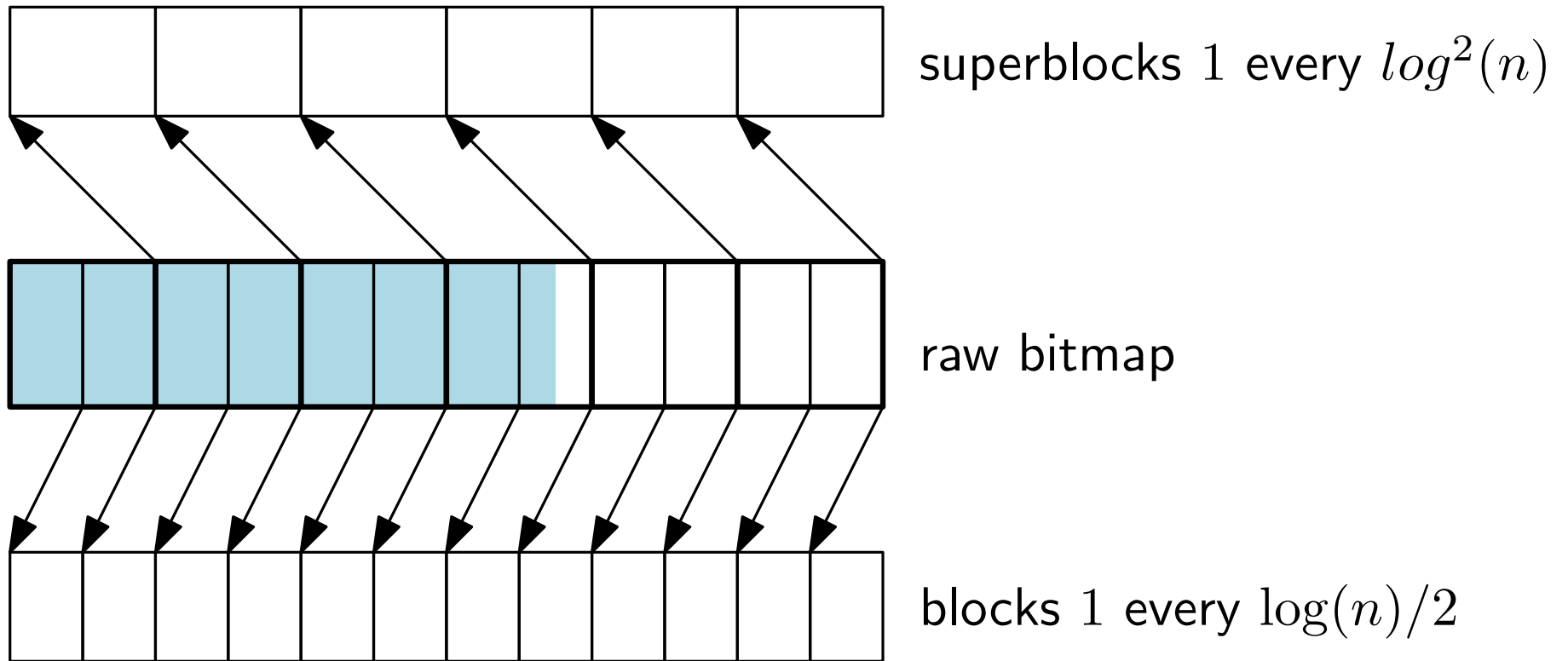


Recurse inside blocks sampling every b . This requires $\frac{n \lg s}{b}$ bits and now we can answer in $O(b)$ time.

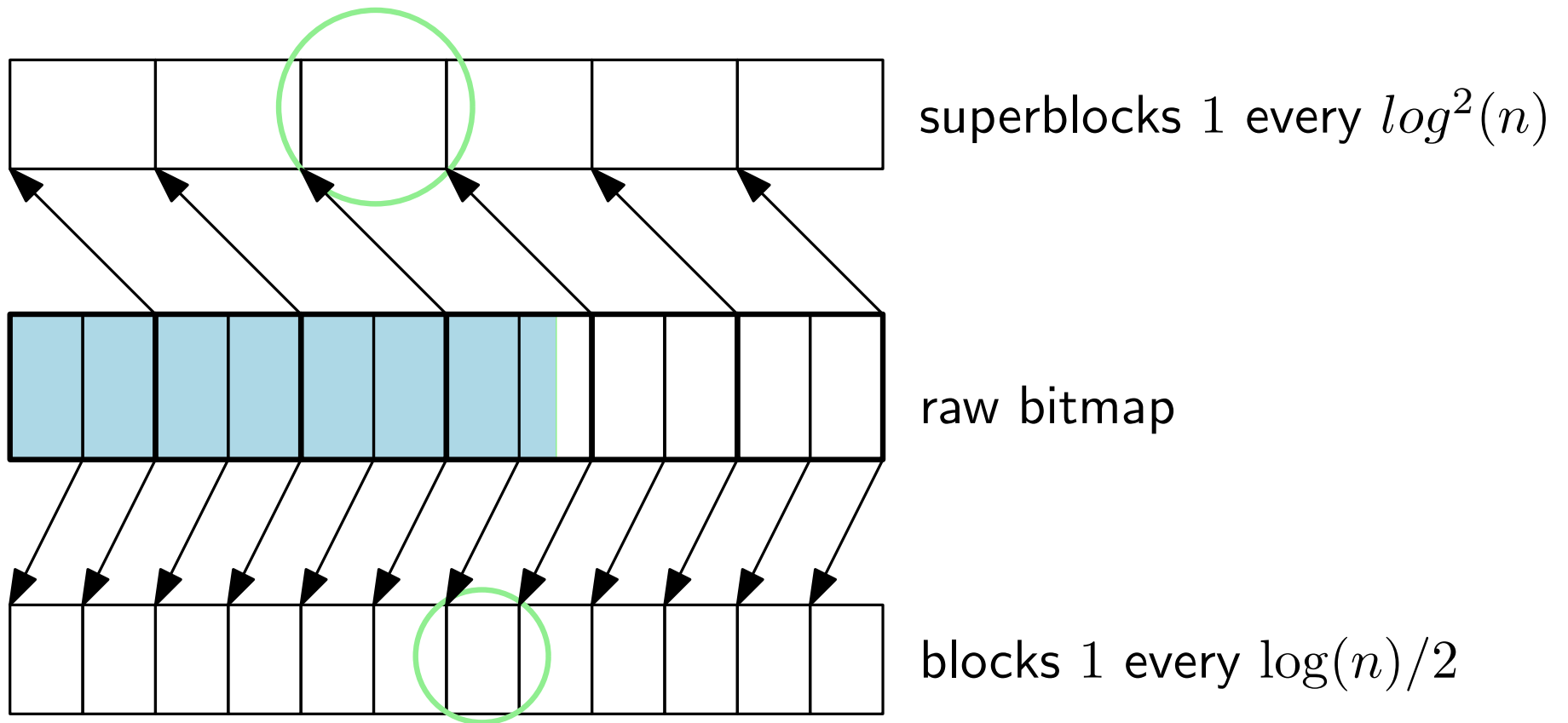
Rank



Rank



Rank



overall space: $n + n / \log n + n \log \log n / \log n$ bits

Rank

So far we have $n + o(n)$ bits and we can answer in $O(\log(n))$ time. The nice thing is that we can handle blocks of size $\log(n)/2$ in constant time.

Idea:

List all blocks of size $\log(n)/2$ and store the rank up to any position in a table for lookup.

Rank

So far we have $n + o(n)$ bits and we can answer in $O(\log(n))$ time. The nice thing is that we can handle blocks of size $\log(n)/2$ in constant time.

Idea:

List all blocks of size $\log(n)/2$ and store the rank up to any position in a table for lookup.

Space required:

$2^{\log(n)/2} \log(n) \log \log(n)/2 \approx \sqrt{n} \log(n) \log \log(n)$ bits.

This is $o(n)$ bits.

Universal Tables

Example with size 8.

00000000	0	0	0	0	0	0	0	0
00000001	0	0	0	0	0	0	0	1
00000010	0	0	0	0	0	0	1	1
...			.	.	.			
01100101	0	1	2	2	2	3	3	4
...			.	.	.			
11111111	1	2	3	4	5	6	7	8

Size in practice:

$\log(n)/2$	size
8	2KB
16	1MB

Universal Tables

Example with size 8.

00000000	0	0	0	0	0	0	0	0
00000001	0	0	0	0	0	0	0	1
00000010	0	0	0	0	0	0	1	1
...			.	.	.			
01100101	0	1	2	2	2	3	3	4
...			.	.	.			
11111111	1	2	3	4	5	6	7	8

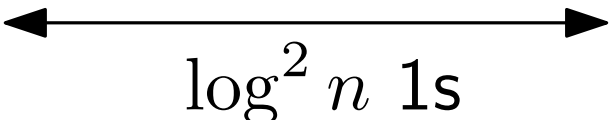
Size in practice:

$\log(n)/2$	size
8	2KB
16	1MB

In practice we use popcnt

Select Operation

Partition according to the number of 1s

$$B = \boxed{10100101} \boxed{001001011} \boxed{0010101001}$$


$\log^2 n$ 1s

Select Operation

Partition according to the number of 1s

$$B = \boxed{10100101} \boxed{001001011} \boxed{0010101001}$$

\longleftrightarrow
 $\log^2 n$ 1s

Sparse superblock: length $\geq \log^3 n \cdot \log \log n$

We store the answer for all sparse superblocks in plain form

Space required: $O(n / \log \log n) + n / \log^2 n + o(n)$ bits

Select Operation

Partition according to the number of 1s

$$B = \boxed{10100101} \boxed{001001011} \boxed{0010101001}$$

$\longleftarrow \log^2 n \text{ 1s} \longrightarrow$

Store the positions where each dense block begins

Divide every superblock into blocks of $(\log \log n)^2 1s$

We consider a block sparse if the size is $\geq 4(\log \log n)^4$
 $\Rightarrow O(n / \log n)$ bits for storing the answers

We recurse again! The next level is small enough.

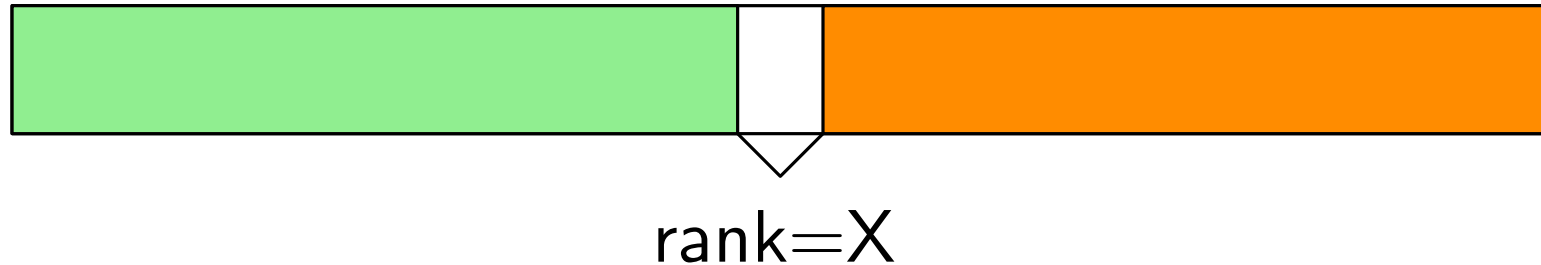
Select Operation

The solution is quite complicated and does not work well in practice

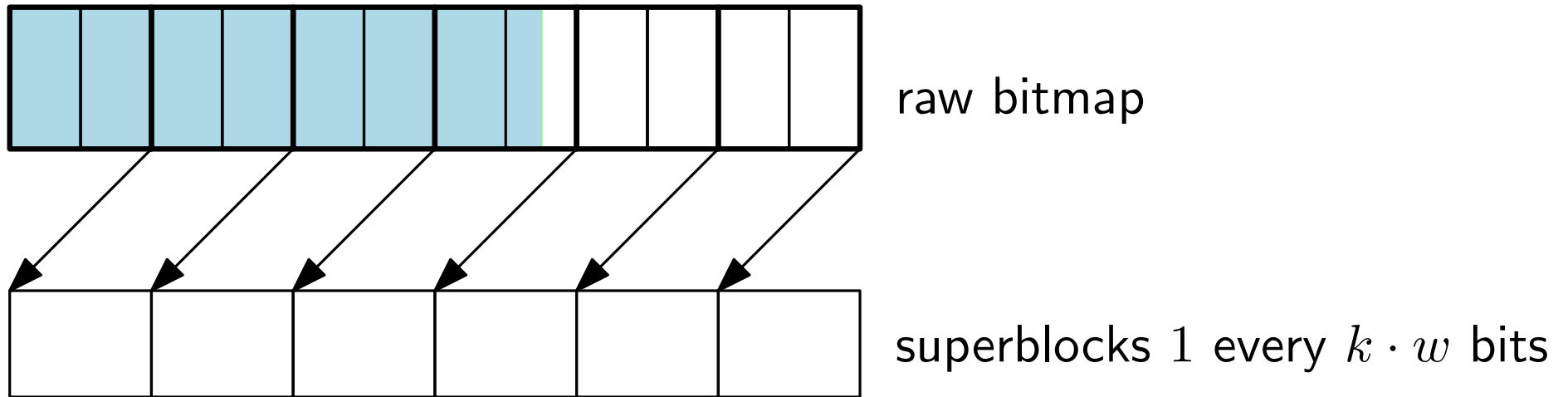
Select Operation

The solution is quite complicated and does not work well in practice

Practical solution: binary search using rank



Implementation in LIBCDS



k popcounts for rank

select requires a binary search over the superblocks + sequential search

Usage in LIBCDS

```
size_t N;
cout << "Length of the bitmap: ";
cin >> N;
uint * bs = new uint[uint_len(N,1)];
for(uint i=0;i<N;i++) {
    uint b;
    cout << "bit at position " << i << ": ";
    cin >> b;
    if(b==0) bitclean(bs,i);
    else bitset(bs,i);
}
BitSequenceRG * bsrg = new BitSequenceRG(bs,N,20);
cout << "rank(" << N/2 << ")=" << bsrg->rank1(N/2) << endl;
cout << "select(1) = " << bsrg->select1(1) << endl;
cout << "size = " << bsrg->getSize() << endl;
delete bsrg;
```

Compressed Bitmaps

B =

101	001	010	001	001	011	001	010	101	000
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Class	Bitmap	Offset
0	000	0
1	001	0
	010	1
	100	2
2	011	0
	101	1
	110	2
3	111	0

Compressed Bitmaps

B = 10100101000100101100010101000

Class	Bitmap	Offset
0	000	0
1	001	0
	010	1
	100	2
2	011	0
	101	1
	110	2
3	111	0

C	2	1	1	1	1	2	1	2	0
O	1	0	1	1	1	2	1	1	0

Compressed Bitmaps

Blocks of size $b = \log(n)/2$

C requires $2n \log \log n / \log n = o(n)$ bits

O?

Compressed Bitmaps

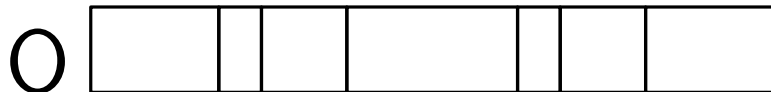
Blocks of size $b = \log(n)/2$

C requires $2n \log \log n / \log n = o(n)$ bits

O?

We can represent an element of class c_i with $\lceil \log \binom{b}{c_i} \rceil$ bits

Total space: $\sum_{i=0}^{n/b} \lceil \log \binom{b}{c_i} \rceil \leq nH_0(B) + O(n/\log n)$ bits



Compressed Bitmaps

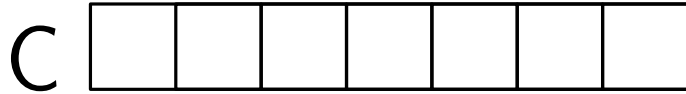
$$\begin{aligned}\sum_{i=1}^{n/b} \left\lceil \log \binom{b}{c_i} \right\rceil &\leq \sum_{i=1}^{n/b} \log \binom{b}{c_i} + n/b \\&= \log \prod_{i=1}^{n/b} \binom{b}{c_i} + O(n/\log n) \\&\leq \log \binom{(n/b)b}{\sum_{i=1}^{n/b} c_i} + O(n/\log n) \\&= \log \binom{n}{m} + O(n/\log n) \\&\leq nH_0(B) + O(n/\log n)\end{aligned}$$

Compressed Bitmaps



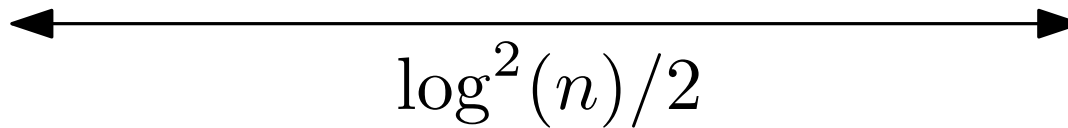
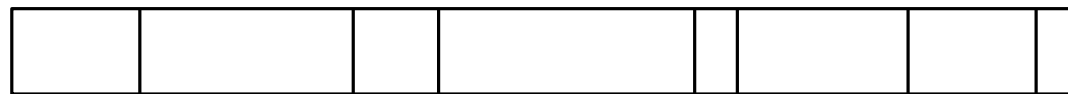
How can we support constant time access?

Compressed Bitmaps



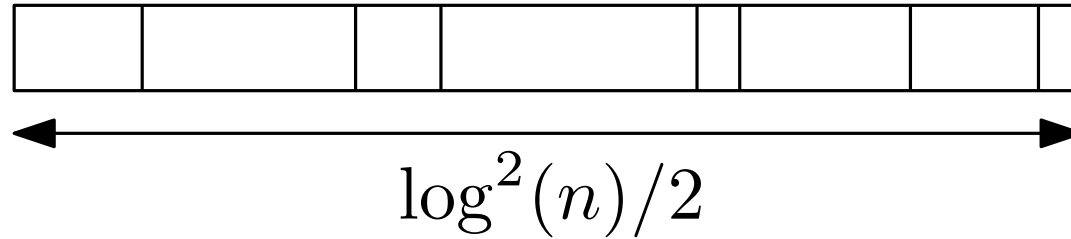
How can we support constant time access?

Store a pointer to O every $\log^2(n)/2$ blocks



$O(n/\log n)$ extra bits

Compressed Bitmaps

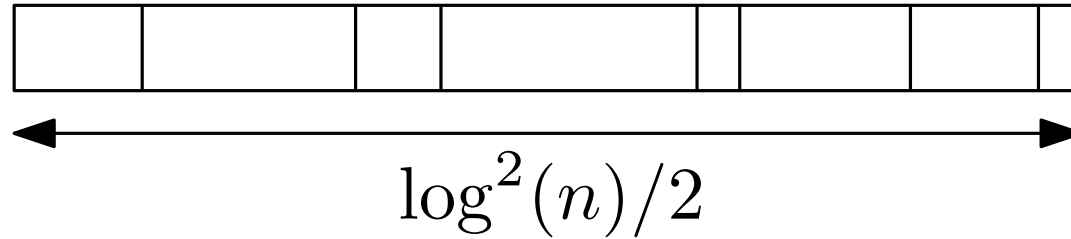


Store offset for each element

One element is at most $O(\log \log n)$ bits

Sum offsets: $O(n \log \log n / \log n)$ bits

Compressed Bitmaps



Store offset for each element

One element is at most $O(\log \log n)$ bits

Sum offsets: $O(n \log \log n / \log n)$ bits

Same idea behind rank

Compressed Bitmaps

Class	Bitmap	Offset
0	000	0
1	001	0
	010	1
	100	2
2	011	0
	101	1
	110	2
3	111	0

Space required for this table?

Compressed Bitmaps

Class	Bitmap	Offset
0	000	0
1	001	0
	010	1
	100	2
2	011	0
	101	1
	110	2
3	111	0

Space required for this table?

$$b2^b + b^2 = O(\sqrt{n} \log n) \text{ bits}$$

Compressed Bitmaps

We can access the bitmap in constant time

What about rank and select?

Compressed Bitmaps

We can access the bitmap in constant time

What about rank and select?

We can access $b = \log(n)/2$ bits at the time

This replaces the plain representation for the solutions already shown

Constant time rank, select and access within $nH_0(B) + o(n)$ bits

Compressed Bitmaps

Practical Implementation

The table and C are easy

O and its sampling?

Compressed Bitmaps

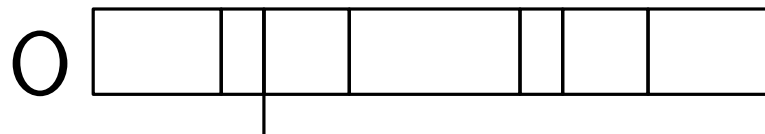
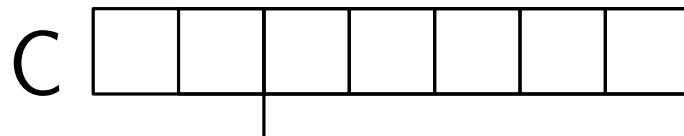
Practical Implementation

The table and C are easy

O and its sampling?

We only keep superblocks (parameter)

Superblocks are traversed linearly



Compressed Bitmaps

```
size_t N;  
cout << "Length of the bitmap: ";  
cin >> N;  
uint * bs = new uint[uint_len(N,1)];  
for(uint i=0;i<N;i++) {  
    uint b;  
    cout << "bit at position " << i << ": ";  
    cin >> b;  
    if(b==0) bitclean(bs,i);  
    else bitset(bs,i);  
}
```

```
BitSequenceRRR * bsrrr = new BitSequenceRRR(bs,N,16);  
cout << "rank(" << N/2 << ")=" << bsrrr->rank1(N/2) << endl;  
cout << "select(1) = " << bsrrr->select1(1) << endl;  
cout << "size = " << bsrrr->getSize() << endl;  
delete bsrrr;
```

Very Sparse Bitmaps

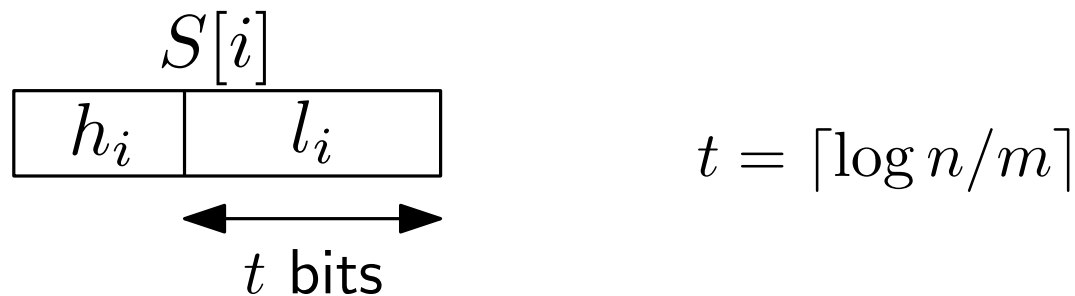
The previous solution does not work well for very sparse bitmaps

store $S[i] = \text{select}(B, i)$ and solve rank with binary search

Very Sparse Bitmaps

The previous solution does not work well for very sparse bitmaps

store $S[i] = \text{select}(B, i)$ and solve rank with binary search

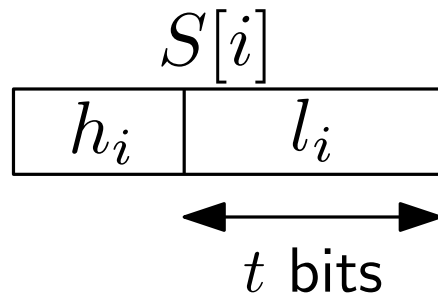


Two arrays: H and L

H is stored in a bitmap of length $2m$

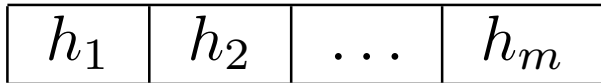
L is stored in $m \log n/m + O(m)$ bits

Very Sparse Bitmaps



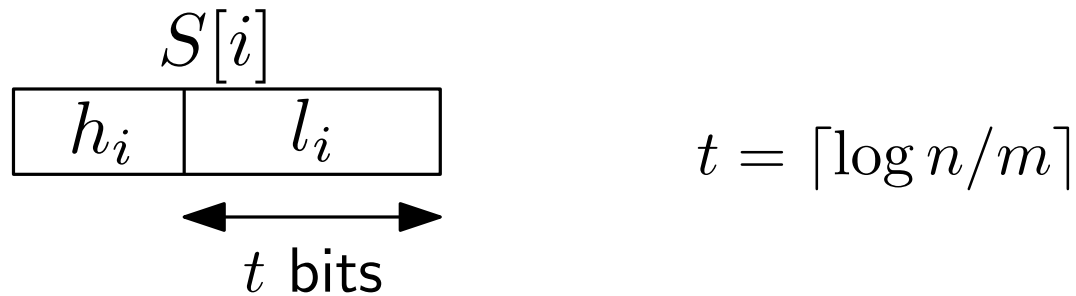
$$t = \lceil \log n/m \rceil$$

Two arrays: H and L

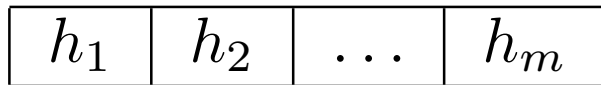


Positions $h_i + i$ are ones

Very Sparse Bitmaps



Two arrays: H and L

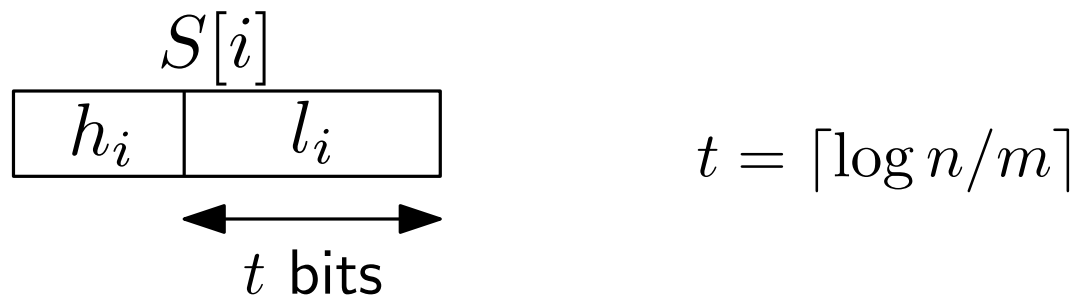


Positions $h_i + i$ are ones

$$h_i = \text{select}(H, i) - i$$

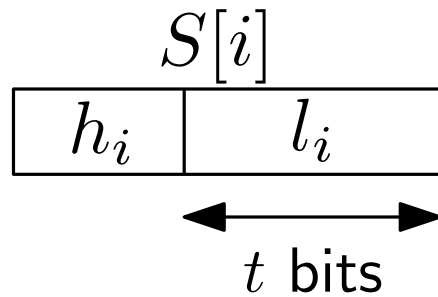
$$h_m + m \leq n/2^t + m \leq 2m$$

Very Sparse Bitmaps



Two arrays: H and L

Very Sparse Bitmaps

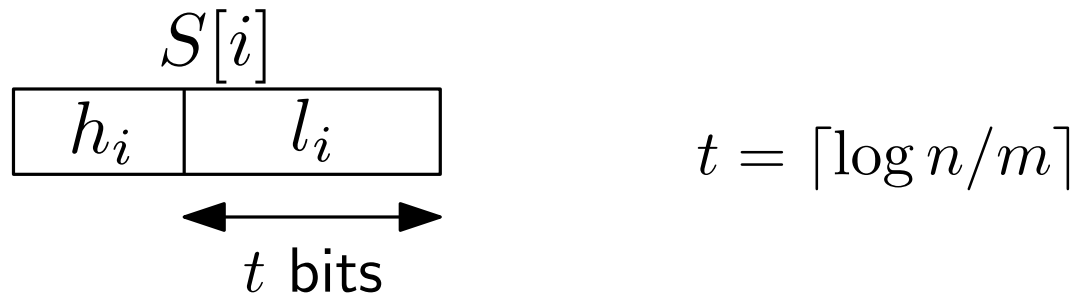


$$t = \lceil \log n / m \rceil$$

Two arrays: H and L

$$\text{select}(B, i) = (\text{select}(H, i) - i) \cdot 2^t + L[i]$$

Very Sparse Bitmaps



Two arrays: H and L

$$\text{select}(B, i) = (\text{select}(H, i) - i) \cdot 2^t + L[i]$$

rank:

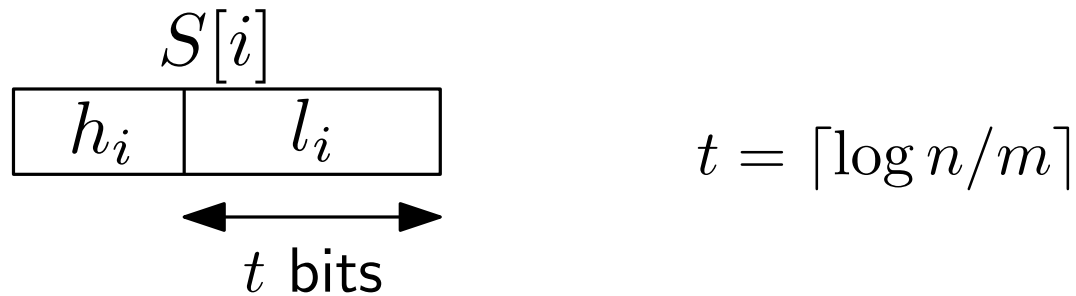
$$i = h \cdot 2^t + l$$

$$x = 1 + \text{rank}(H, \text{select}_0(H, h))$$

$$y = \text{rank}(H, \text{select}_0(H, h + 1))$$

binary search $L[x, y]$

Very Sparse Bitmaps



Two arrays: H and L

Space: $O(m)$ for H and $m \log n/m$ for L

select: $O(1)$

rank: $O(\log n/m)$

access: $O(\log n/m)$

Very Sparse Bitmaps

```
size_t N;
cout << "Length of the bitmap: ";
cin >> N;
uint * bs = new uint[uint_len(N,1)];
for(uint i=0;i<N;i++) {
    uint b;
    cout << "bit at position " << i << ": ";
    cin >> b;
    if(b==0) bitclean(bs,i);
    else bitset(bs,i);
}
BitSequenceSDArray * bss = new BitSequenceSDArray(bs,N);
cout << "rank(" << N/2 << ")=" << bss->rank1(N/2) << endl;
cout << "select(1) = " << bss->select1(1) << endl;
cout << "size = " << bss->getSize() << endl;
delete bss;
```

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation
- Basics
- Bitmaps
- Sequences ←
- Applications

Sequences

$S = \text{EHDHACEEGBCBGCF}$

Sequences

$S = \boxed{\text{EHDHACE}}\text{EGBCB}\boxed{\text{B}}\text{GCF}$

$\text{access}_S(12) = B$

$\text{rank}_S(H, 7) = 2$

$\text{select}_S(E, 3) = 8$

$$n = |S|$$

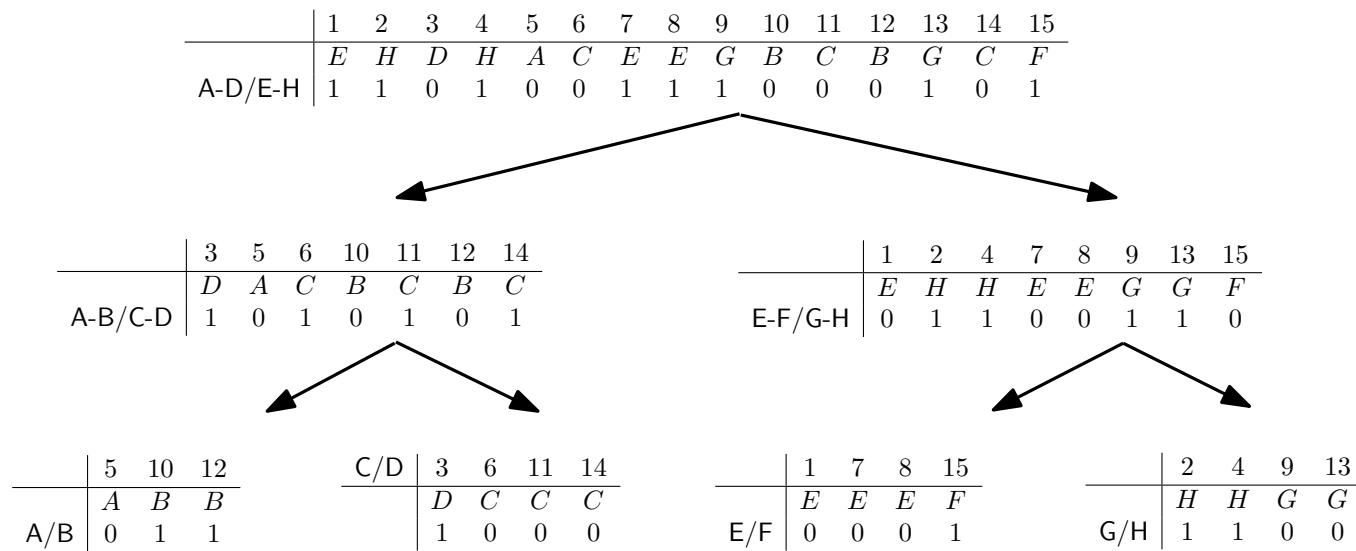
$$\sigma = |\Sigma|$$

Space: $n \lceil \log \sigma \rceil$ bits

Sequences

- Wavelet Trees [Grossi et al.]
- GMR [Golynski et al.]
- Alphabet Partitioning [Barbay et al.]

Wavelet Trees



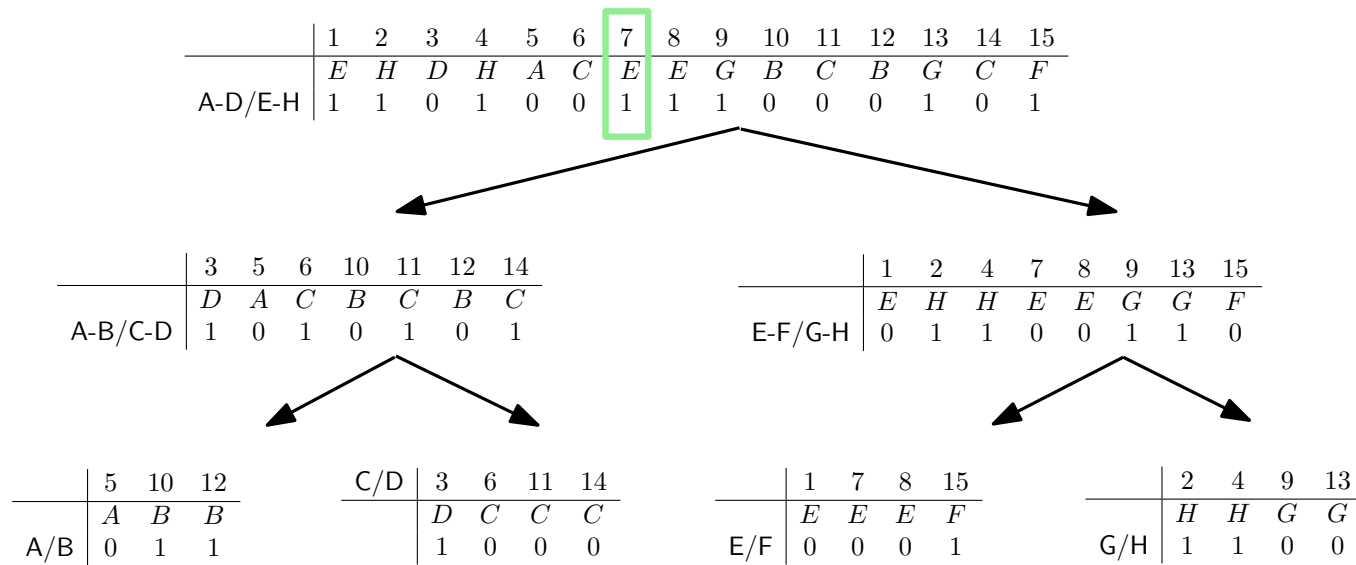
$n + o(n)$ bits

$n + o(n)$ bits

$n + o(n)$ bits

$n \lceil \log \sigma \rceil (1 + o(1))$

Wavelet Trees



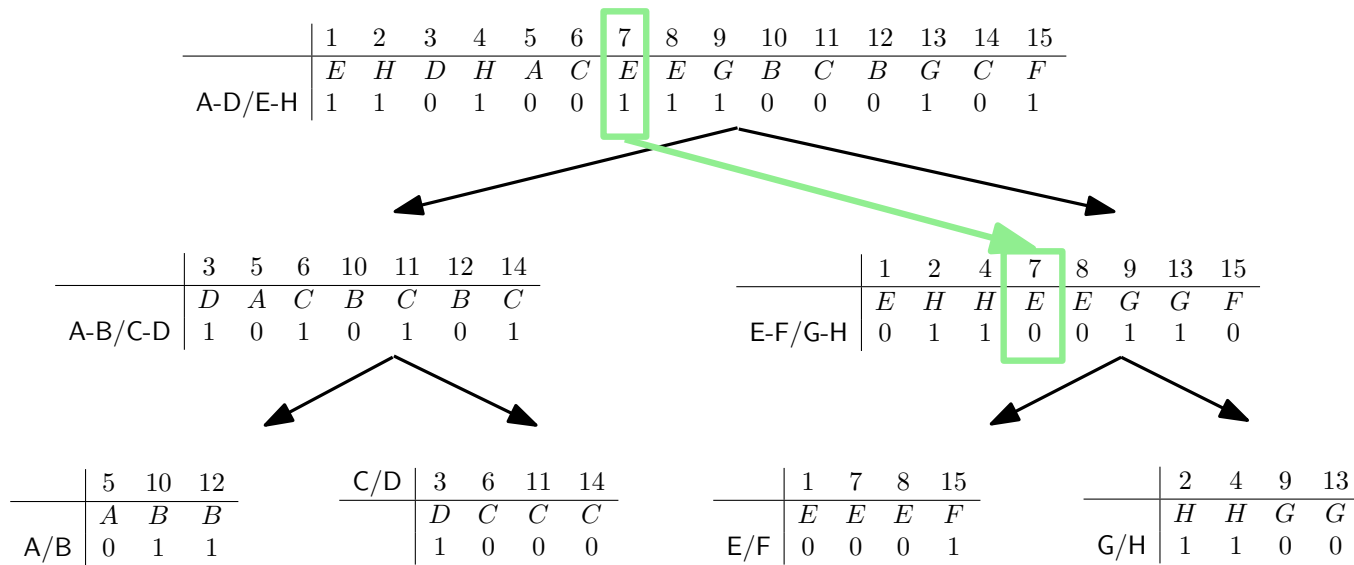
$n + o(n)$ bits

$n + o(n)$ bits

$n + o(n)$ bits

$n \lceil \log \sigma \rceil (1 + o(1))$

Wavelet Trees



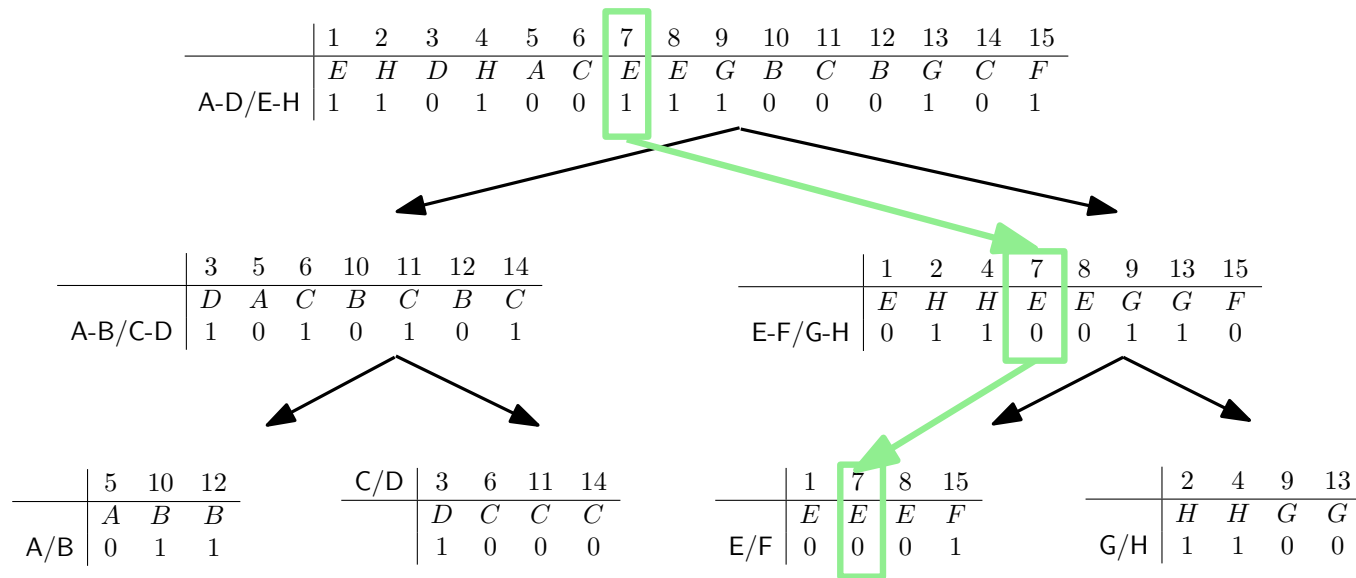
$n + o(n)$ bits

$n + o(n)$ bits

$n + o(n)$ bits

$$n \lceil \log \sigma \rceil (1 + o(1))$$

Wavelet Trees



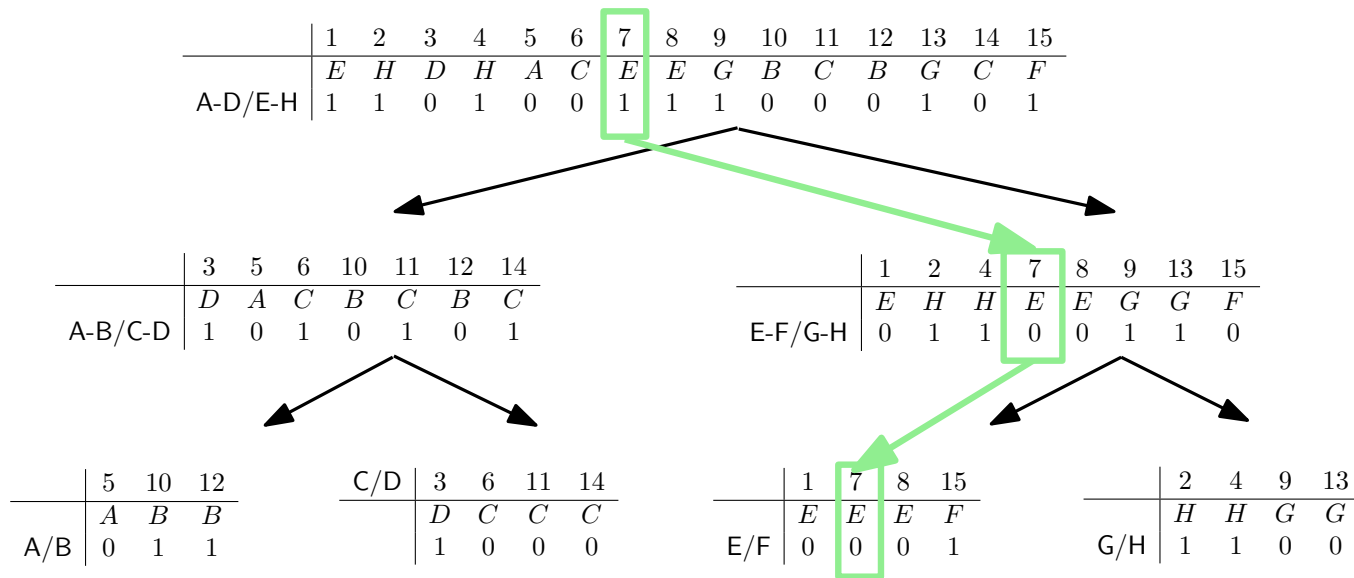
$n + o(n)$ bits

$n + o(n)$ bits

$n + o(n)$ bits

$n \lceil \log \sigma \rceil (1 + o(1))$

Wavelet Trees



$n + o(n)$ bits

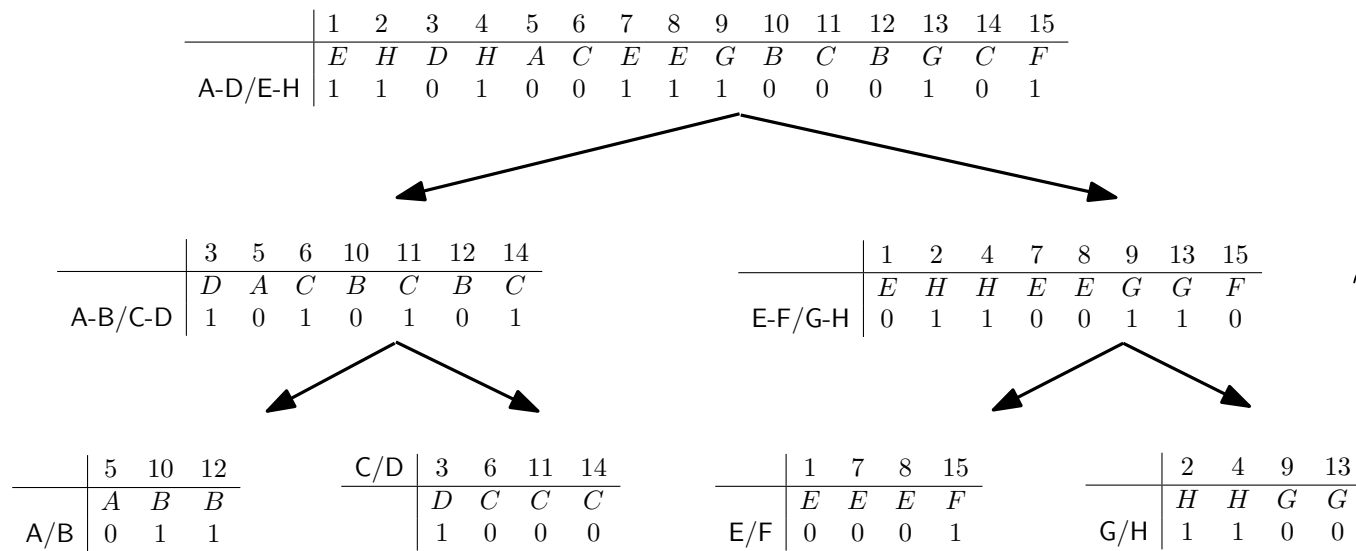
$n + o(n)$ bits

$n + o(n)$ bits

$$n \lceil \log \sigma \rceil (1 + o(1))$$

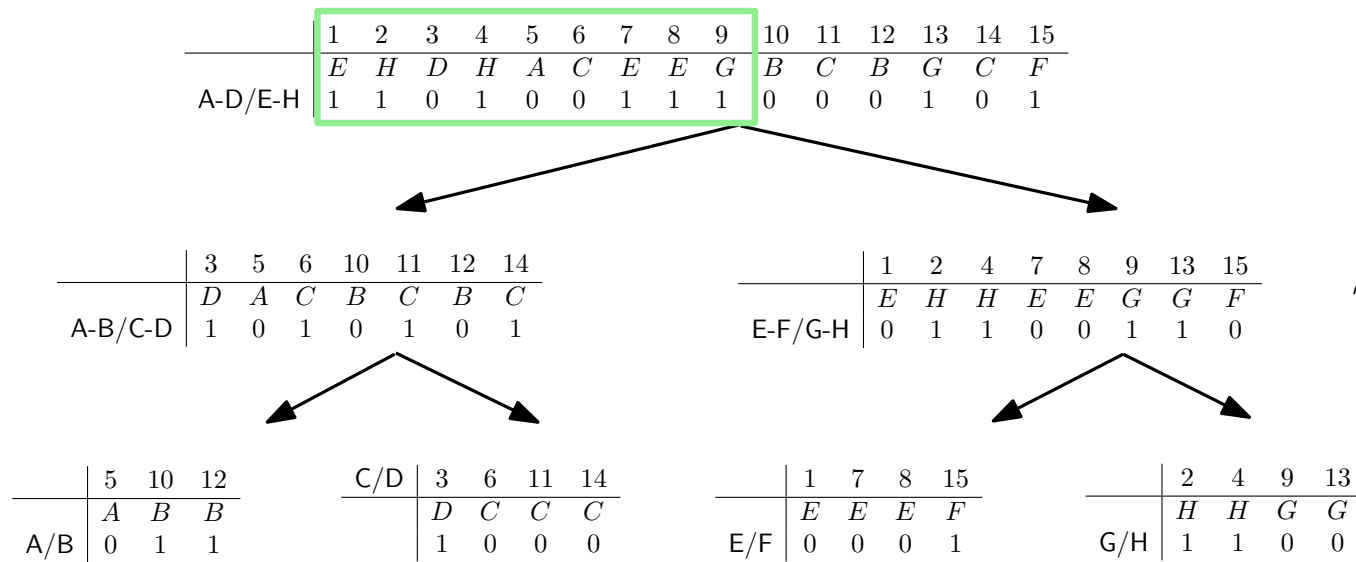
access takes $O(\log \sigma)$ time

Wavelet Trees

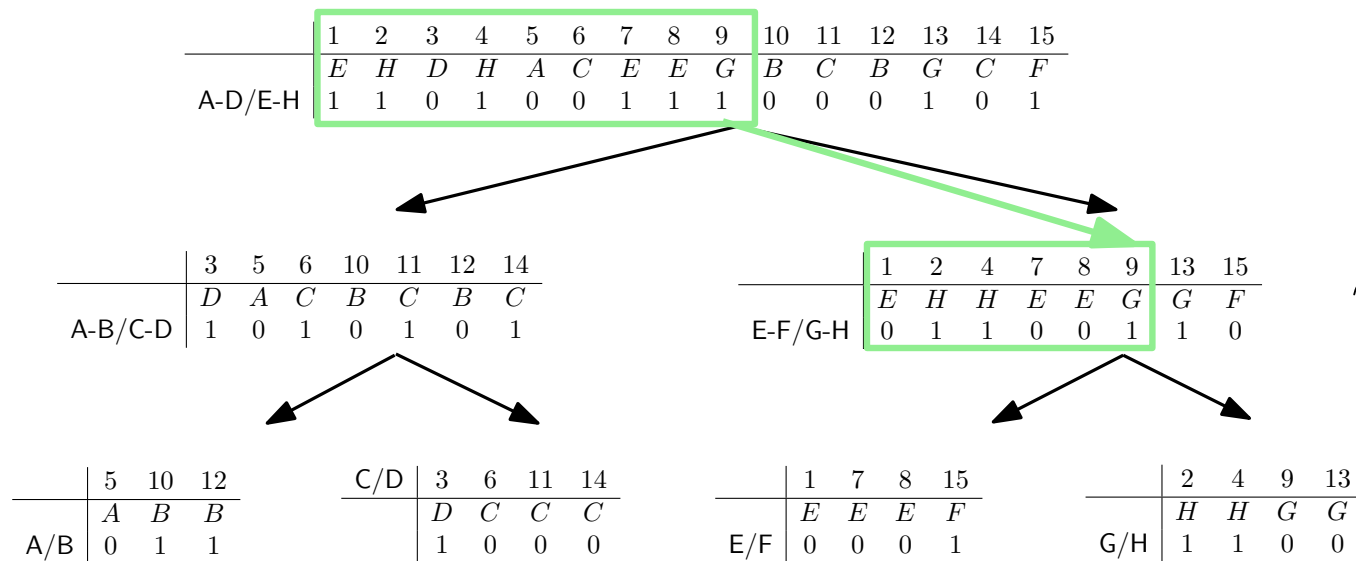


$rank(E, 9)$

Wavelet Trees

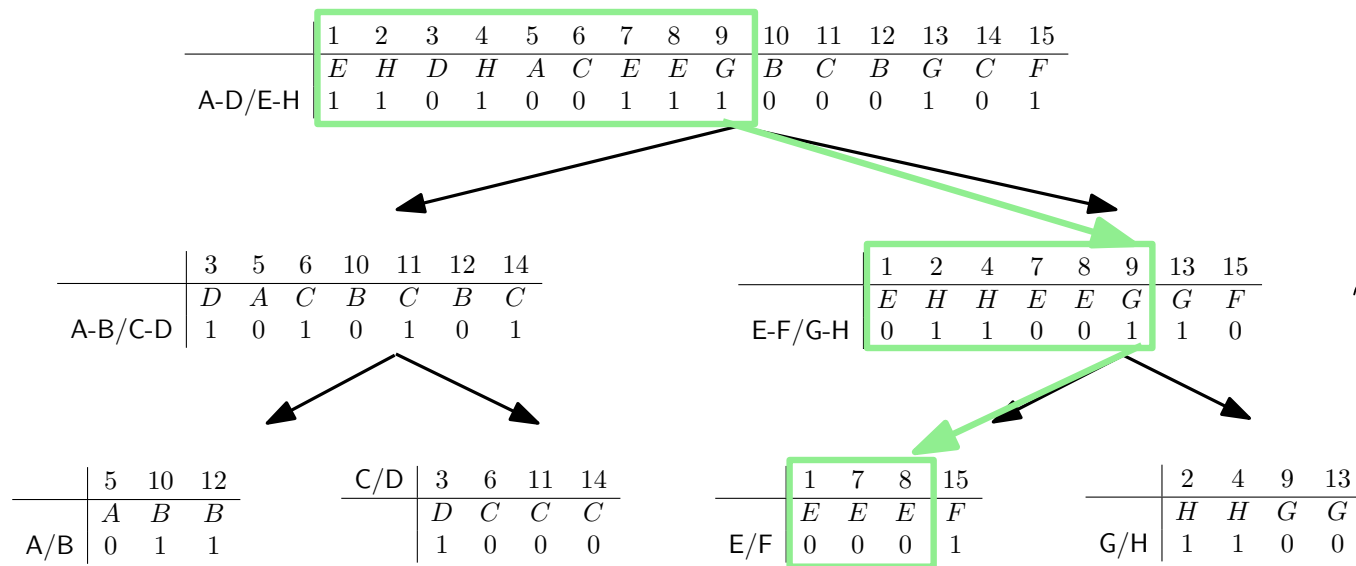


Wavelet Trees



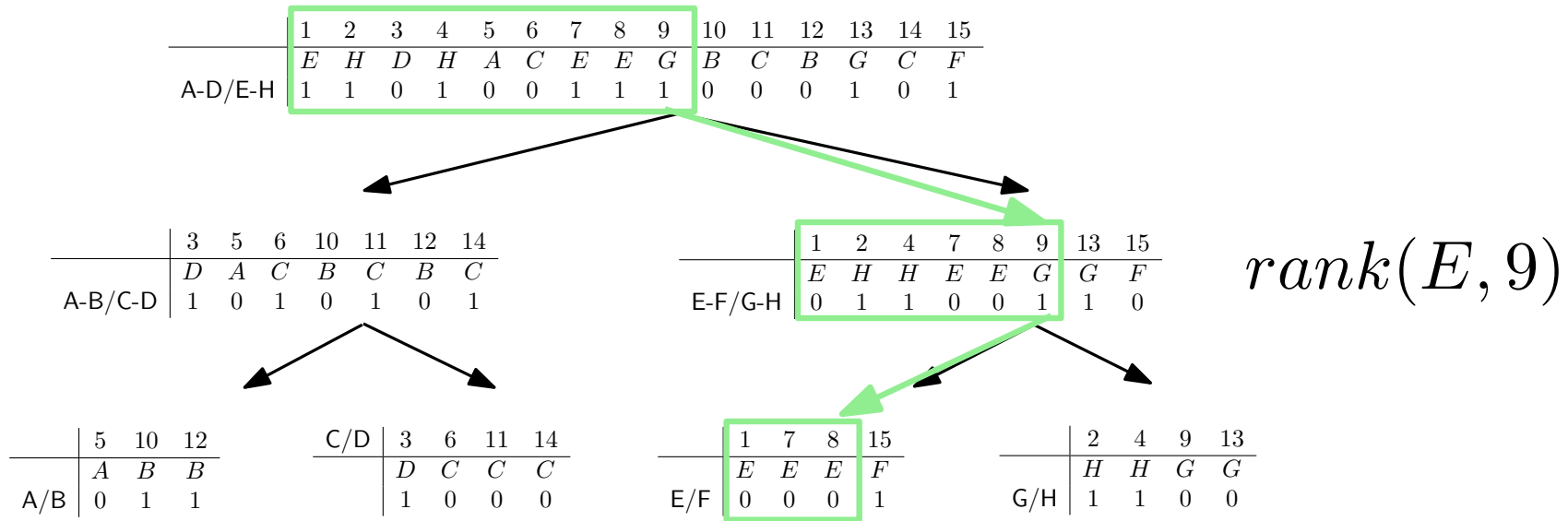
$rank(E, 9)$

Wavelet Trees



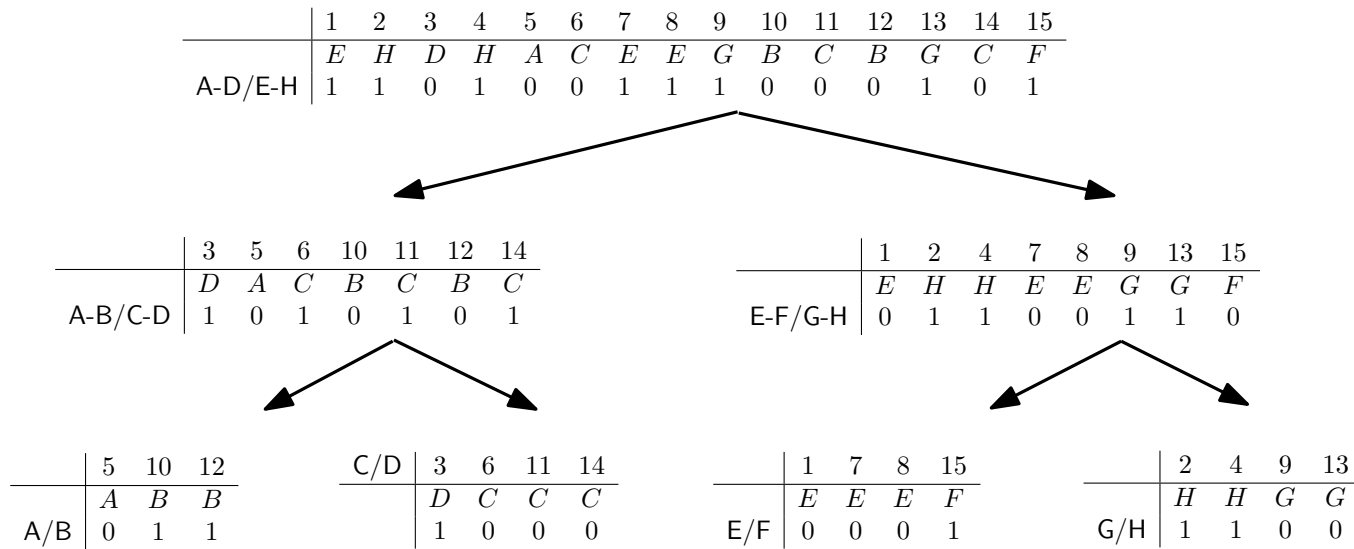
$\text{rank}(E, 9)$

Wavelet Trees



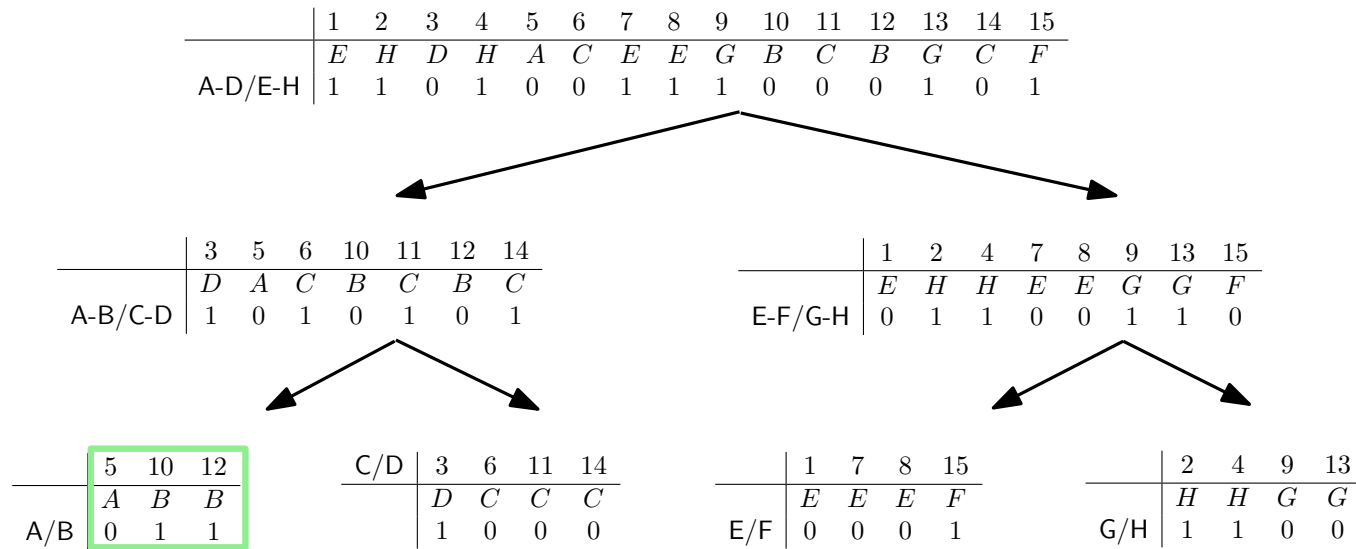
rank also takes $O(\log \sigma)$ time

Wavelet Trees



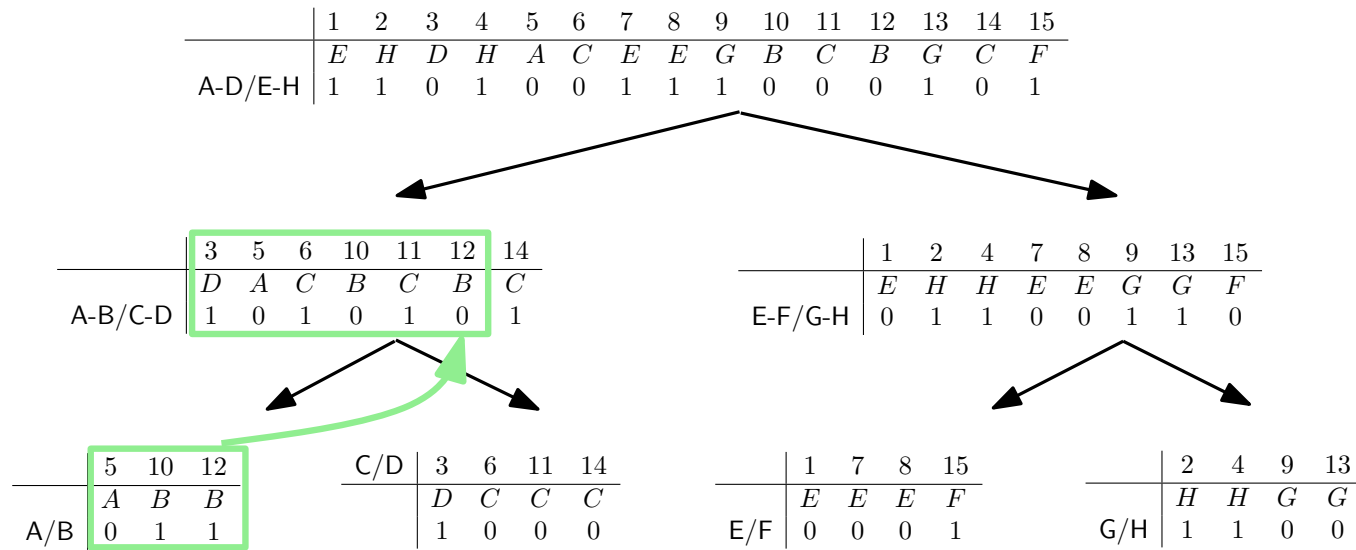
select(B, 2)

Wavelet Trees



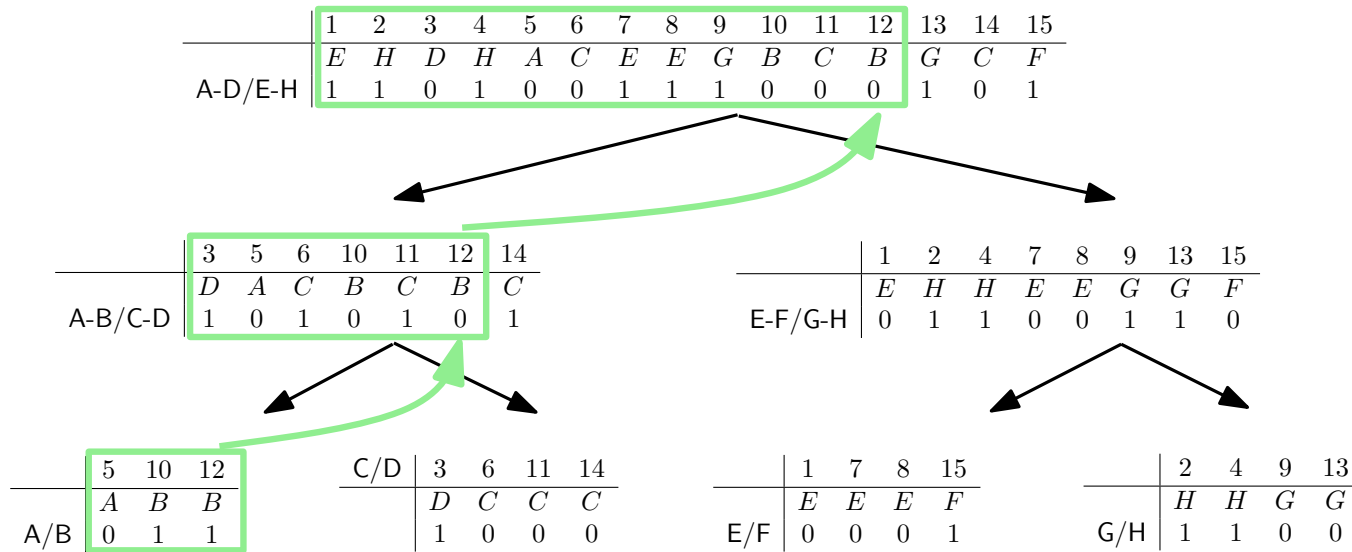
select(*B*, 2)

Wavelet Trees



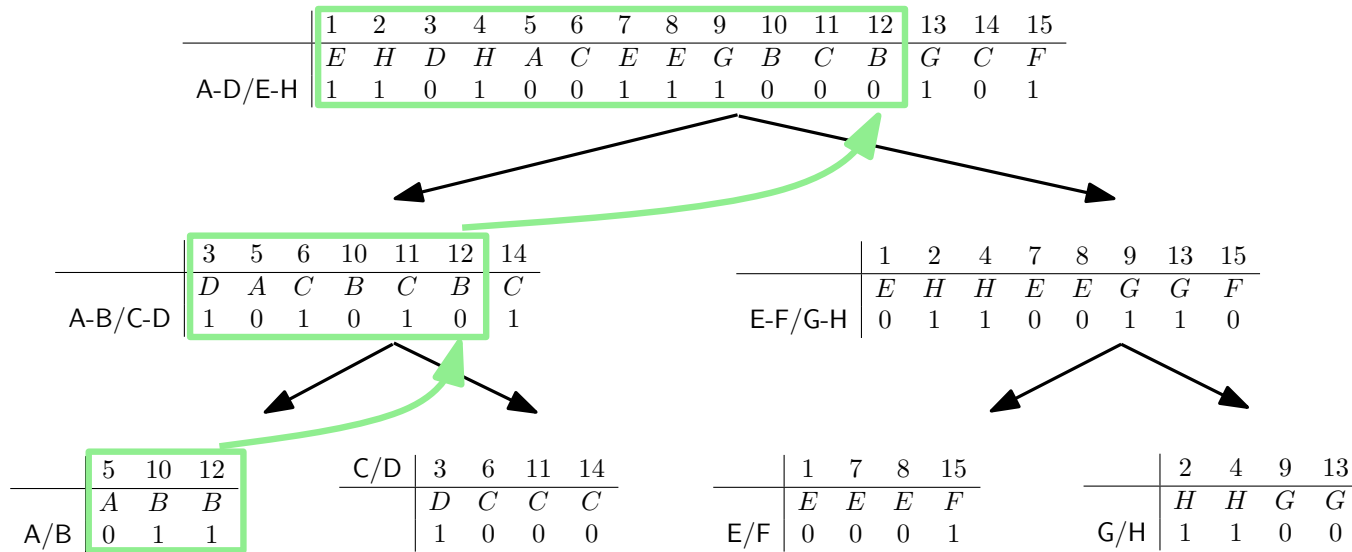
select(B, 2)

Wavelet Trees



select(B, 2)

Wavelet Trees



select(B, 2)

select takes $O(\log \sigma)$ time

Wavelet Trees

We can save pointers

Other codifications work too

Huffman Shape

Space: $nH_0(S) + o(n \log \sigma)$ bits

Query time: $O(H_0(S))$ expected

Wavelet Trees

We can save pointers ←

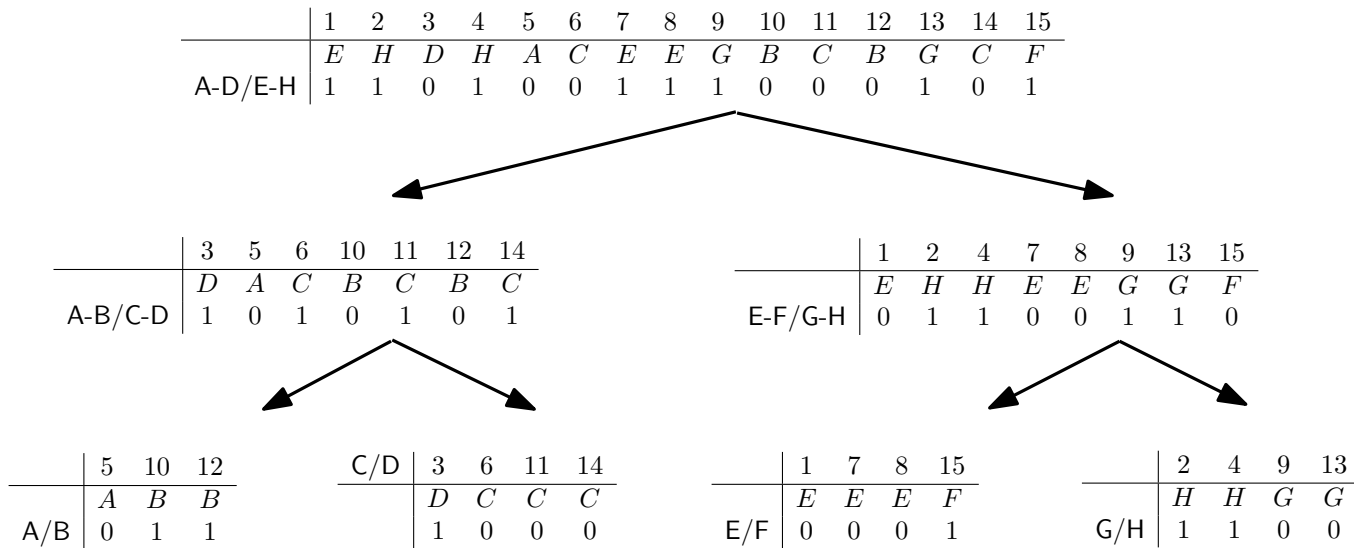
Other codifications work too

Huffman Shape

Space: $nH_0(S) + o(n \log \sigma)$ bits

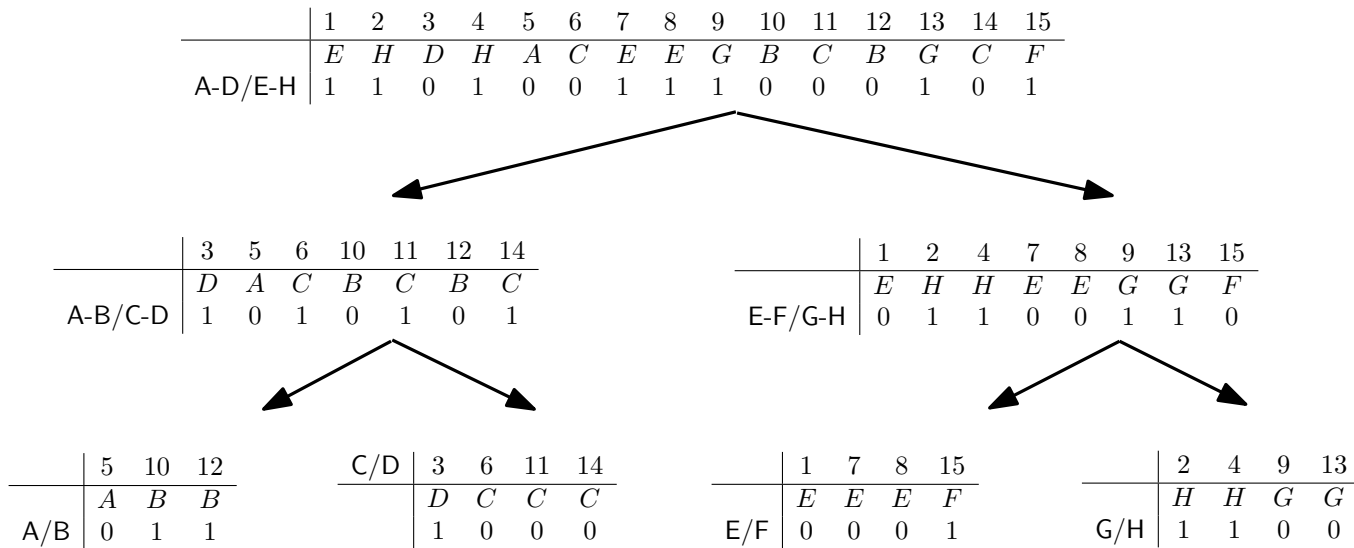
Query time: $O(H_0(S))$ expected

Wavelet Trees



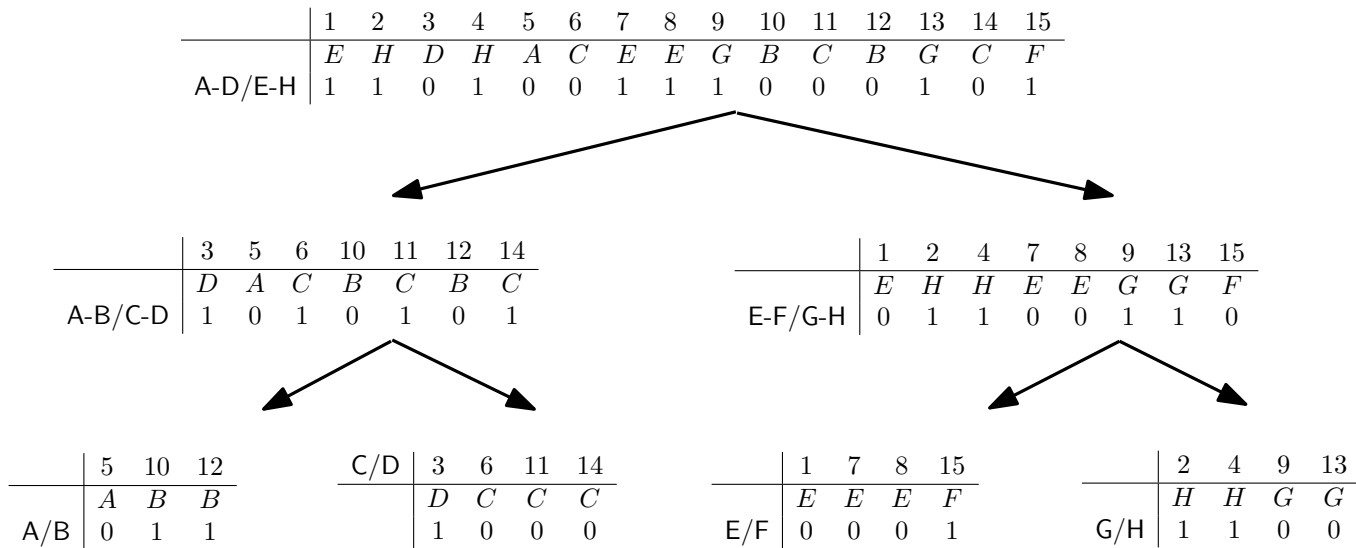
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees



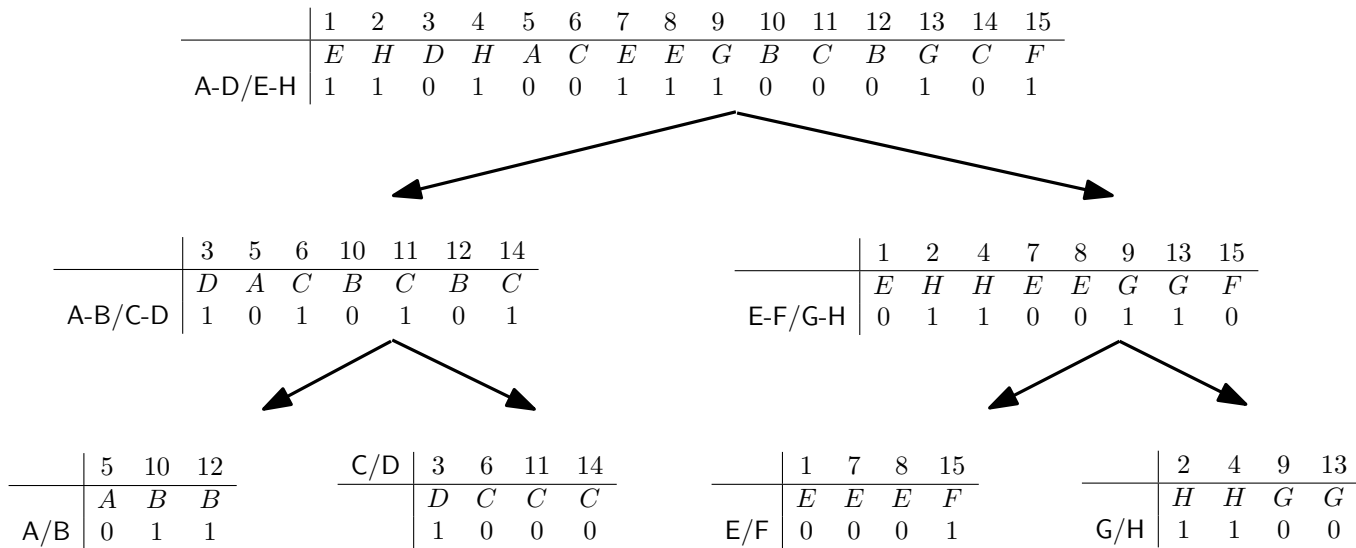
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees



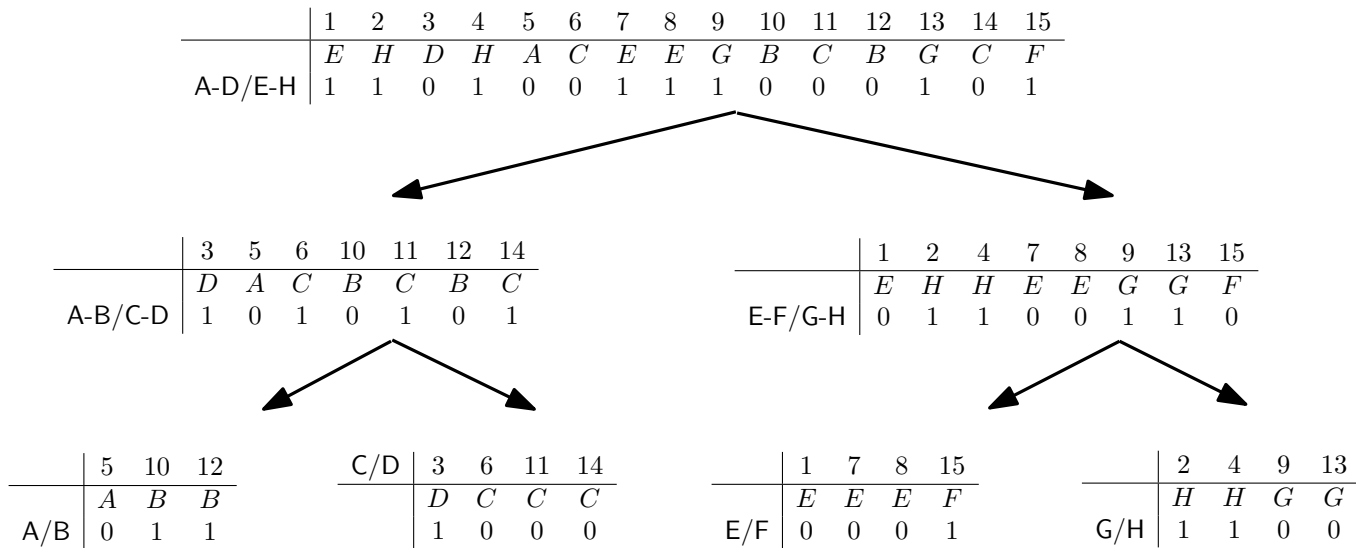
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

Wavelet Trees

Saving pointers is useful for large alphabets: $O(\sigma \log n)$ bits

We showed a solution with $O(\log \sigma)$ pointers

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>E</i>	<i>H</i>	<i>D</i>	<i>H</i>	<i>A</i>	<i>C</i>	<i>E</i>	<i>E</i>	<i>G</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>G</i>	<i>C</i>	<i>F</i>
1	1	0	1	0	0	1	1	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	0	0	0	1	1	1	0	0

We can reduce it further to 1 pointer

Wavelet Trees

Classes Implemented

WaveletTree

WaveletTreeNoptrs

WaveletTree

Sequence

Length

Coder

Bitmap Builder

Mapper

WaveletTreeNoptrs

Sequence

Length

Bitmap Builder

Mapper

Wavelet Trees

Classes Implemented

WaveletTree

WaveletTreeNoptrs

WaveletTree

Sequence

Length



Coder

Bitmap Builder

Mapper

WaveletTreeNoptrs

Sequence

Length

Bitmap Builder

Mapper

Wavelet Trees

Classes Implemented

WaveletTree

WaveletTreeNoptrs

WaveletTree

Sequence

Length



Coder



Bitmap Builder
Mapper

WaveletTreeNoptrs

Sequence

Length



Bitmap Builder
Mapper

Wavelet Trees

Classes Implemented

WaveletTree

WaveletTreeNoptrs

WaveletTree

Sequence

Length



Coder



Bitmap Builder



Mapper

WaveletTreeNoptrs

Sequence

Length



Bitmap Builder



Mapper

Wavelet Trees

```
size_t N;
uint s;
cout << "Length: ";
cin >> N;
uint * seq = new uint[N];
for(size_t i=0;i<N;i++) {
    uint v;
    cout << "Element at position " << i << ": ";
    cin >> seq[i];
}

WaveletTree * wt1 = new WaveletTree(seq, N,
    new wt_coder_huff(seq, N,
        new MapperNone()),
    new BitSequenceBuilderRG(20),
    new MapperNone());
cout << "size = " << wt1->getSize() << " bytes" << endl;
```

GMR

$S = \text{abracadabraa}$

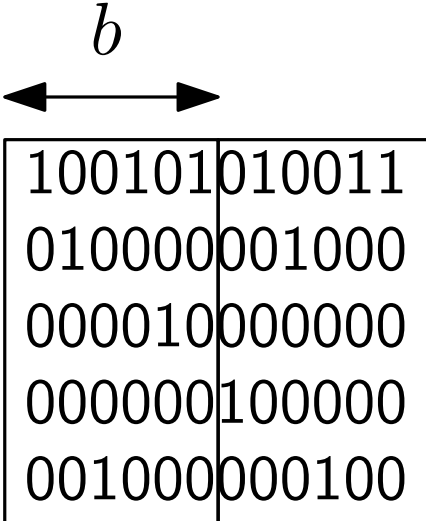
b
↔


a	100101	010011
b	010000	001000
c	000010	000000
d	000000	100000
r	001000	000100

$B = 10001000 \ 1010 \ 101 \ 110 \ 1010$

GMR

$S = \text{abracadabraa}$



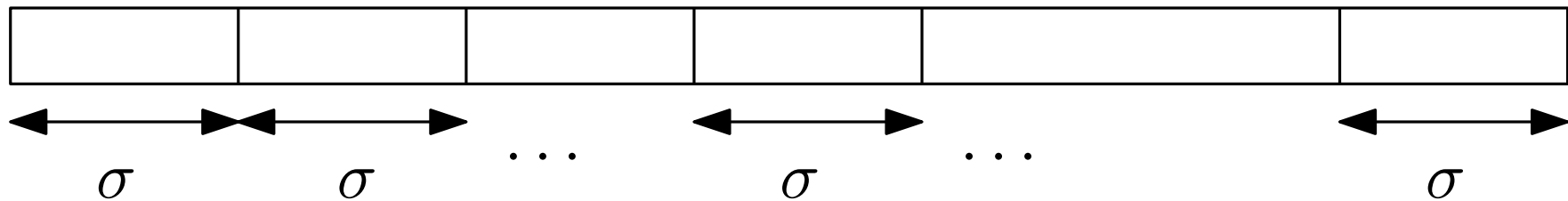
	b	
		
a	100101	010011
b	010000	001000
c	000010	000000
d	000000	100000
r	001000	000100

$B = 10001000 \ 1010 \ 101 \ 110 \ 1010$

Space: $n\sigma/b + n$ bits

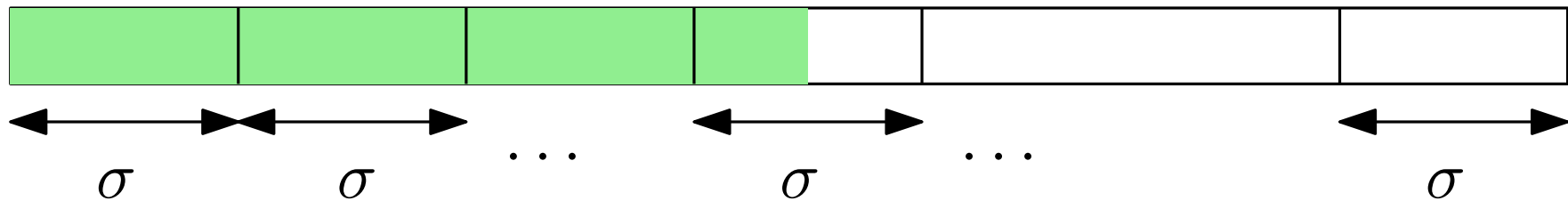
GMR

We can solve rank and select for multiples of σ



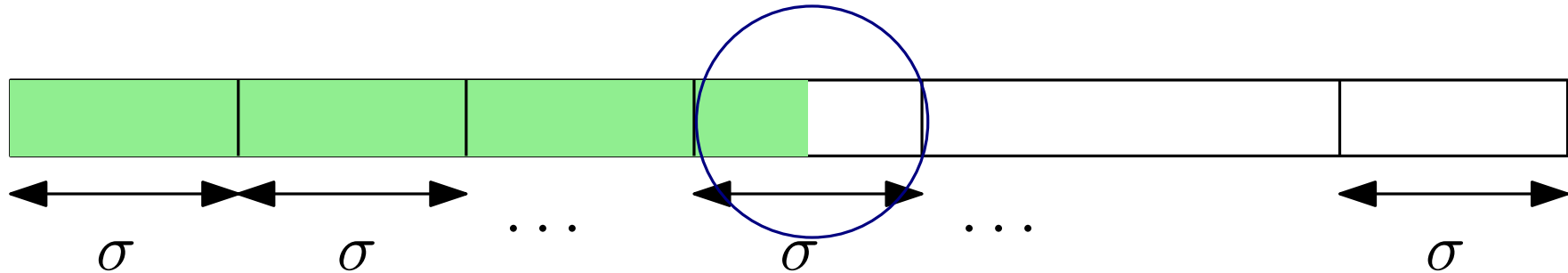
GMR

We can solve rank and select for multiples of σ



GMR

We can solve rank and select for multiples of σ



We need to solve rank, select and access for sequences of length σ

These are called *chunks*

GMR

$S = \text{abcaaccbbcaa}$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

GMR

$S = \text{abcaaccbbcaa}$

access(3)

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

GMR

$S = \text{abcaaccbbcaa}$

$\text{access}(3)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$


$$\pi^{-1}(3) = 9$$

GMR

$S = \text{abcaaccbbcaa}$

$\text{access}(3)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

$$\pi^{-1}(3) = 9$$


GMR

$S = \text{abcaaccbbcaa}$

$X = 100000100010000$

$\text{access}(3)$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

$$\pi^{-1}(3) = 9$$

GMR

$S = \text{abcaaccbbcaa}$

$\text{select}(b, 2)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

GMR

$S = \text{abcaaccbbcaa}$

$\text{select}(b, 2)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

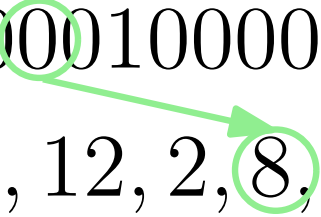
GMR

$S = \text{abcaaccbbcaa}$

$\text{select}(b, 2)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

A green circle highlights the 7th character '0' in the binary string X. A green arrow points from this circle to the 7th element '8' in the permutation array pi, which is also circled in green.

GMR

$S = \text{abcaaccbbcaa}$

$\text{rank}(c, 8)$

$X = 100000100010000$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

GMR

$S = \text{abcaaccbbcaa}$

$\text{rank}(c, 8)$

$X = 10000010001\boxed{0000}$

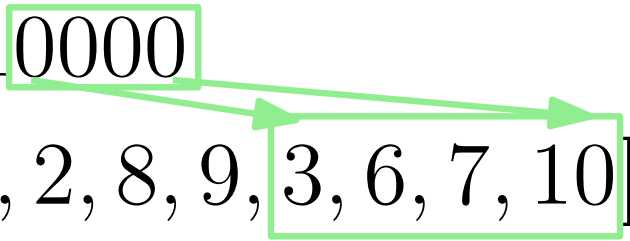
$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

GMR

$S = \text{abcaaccbbcaa}$

$X = 1000001000100000$
 $\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

$\text{rank}(c, 8)$

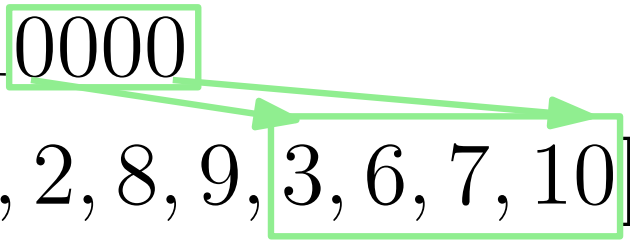


GMR

$S = \text{abcaaccbbcaa}$

$X = 1000001000100000$ $\text{rank}(c, 8)$

$\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$



With a Y-Fast trie, the search takes $O(\log \log \sigma)$

GMR

$S = \text{abcaaccbbcaa}$

$X = 1000001000100000$
 $\pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10]$

$\text{rank}(c, 8)$

The diagram illustrates the rank of character 'c' at position 8 in the string S. The string S is 'abcaaccbbcaa'. The character 'c' is at index 8 (0-indexed). The rank of 'c' is the number of characters less than 'c' to its left, which is 3 (characters 'a', 'b', 'c'). The diagram shows the binary representation X = 1000001000100000, where the 8th bit is 1. The permutation pi = [1, 4, 5, 11, 12, 2, 8, 9, 3, 6, 7, 10] maps the 8th position to the 9th position in the permutation array. The 9th element of pi is 3, which is the rank of 'c'.

With a Y-Fast trie, the search takes $O(\log \log \sigma)$

In practice we use binary search

GMR

We can represent a permutation over σ values in $\sigma \log \sigma (1 + o(1))$ bits

$\pi(i)$ takes $O(1)$ time and $\pi^{-1}(i)$ takes $O(\log \log \sigma)$
[Munro et al.]

GMR supports rank and access in $O(\log \log \sigma)$ time and select in constant time, within space $n \log \sigma + n \cdot o(\log \sigma)$ bits of space.

GMR

In LIBCDS

SequenceGMR { chunk_length
PermutationBuilderMRRR
SequenceBuilderGMRChunk

GMR

```
size_t N;
uint s;
cout << "Length: ";
cin >> N;
uint * seq = new uint[N];
for(size_t i=0;i<N;i++) {
    uint v;
    cout << "Element at position " << i << ": ";
    cin >> seq[i];
}
SequenceGMR * gmr = new SequenceGMR(seq, N, 5u,
    new BitSequenceBuilderRG(20),
    new SequenceBuilderGMRChunk(
        new BitSequenceBuilderRG(20),
        new PermutationBuilderMRRR(
            20,
            new BitSequenceBuilderRG(20))));
cout << "size = " << gmr->getSize() << " bytes" << endl;
```


Alphabet Partitioning

$S = EHDHACEEGBCBGC F$

Symbol	Freq
A	1
B	2
C	3
D	1
E	3
F	1
G	2
H	2

Alphabet Partitioning

$S = EHDHACEEGBCBGC F$

Symbol	Freq		Symbol	Freq
A	1		C	3
B	2		E	3
C	3		B	2
D	1		G	2
E	3		H	2
F	1		A	1
G	2		D	1
H	2		F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGC F$

Symbol	Freq		Symbol	Freq	
A	1		C	3	Class 1
B	2		E	3	
C	3		B	2	Class 2
D	1	→	G	2	
E	3		H	2	Class 3
F	1		A	1	
G	2		D	1	
H	2		F	1	Class 4

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

Symbol	Freq		
C	3	Class 1	
E	3		C:
B	2	Class 2	
G	2		O:
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = E H D H A C E E G B C B G C F$

Symbol	Freq		
C	3	Class 1	
E	3		C:
B	2	Class 2	
G	2		O:
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = E H D H A C E E G B C B G C F$

Symbol	Freq		
C	3	Class 1	
E	3		C: 2
B	2	Class 2	
G	2		O: 0
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = E H D H A C E E G B C B G C F$

Symbol	Freq		
C	3	Class 1	
E	3		C: 2
B	2	Class 2	
G	2		O: 0
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = EH D H A C E E G B C B G C F$

Symbol	Freq		
C	3	Class 1	
E	3		C: 2 3
B	2	Class 2	
G	2		O: 0 1
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = EH D H A C E E G B C B G C F$

Symbol	Freq		
C	3	Class 1	
E	3		C: 2 3 ...
B	2	Class 2	
G	2		O: 0 1 ...
H	2	Class 3	
A	1		
D	1		
F	1	Class 4	

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2 3 ...

O: 0 1 ...

Represent C using a Wavelet Tree

Alphabet Partitioning

$S = EHDHACEEGBCBGC F$

C: 2 3 ...

O: 0 1 ...

Represent C using a Wavelet Tree

How many classes do we have?

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2 3 ...

O: 0 1 ...

Represent C using a Wavelet Tree

How many classes do we have?

Space for C is $O(n \log \log \sigma)$, queries take constant time.

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

$$\left. \begin{array}{l} O_1 \\ O_2: 0, 0, 0, 1, 1 \\ O_3: 1, 3, 1, 2, 0, 0 \\ O_4: 0 \end{array} \right\} \approx nH_0(B)$$

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

access(7)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

access(7)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

access(7)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

rank(H,7)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

rank(H,7)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4


O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

rank(H,7)



Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

$C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4$

O_1

$O_2: 0, 0, 0, 1, 1$

$O_3: 1, 3, 1, 2, 0, 0$

$O_4: 0$

$\text{rank}(H, 7)$

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1



Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

$C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4$

O_1

$O_2: 0, 0, 0, 1, 1$

$O_3: 1, 3, 1, 2, 0, 0$

$O_4: 0$

$\text{rank}(H, 7)$

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

select(B,1)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4



Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

select(B,1)

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4



Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

select(B,1)

Alphabet Partitioning

$S = EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2 : 0, 0, 0, 1, 1

O_3 : 1, 3, 1, 2, 0, 0

O_4 : 0

select(B,1)

Symbol	Freq
C	3
E	3
B	2
G	2
H	2
A	1
D	1
F	1

Alphabet Partitioning

...

```
SequenceBuilder * sb1 = new SequenceBuilderWaveletTree(  
    new BitSequenceBuilderRG(20),  
    new MapperNone());  
SequenceBuilder * sb2 = new SequenceBuilderGMRChunk(  
    new BitSequenceBuilderRG(20),  
    new PermutationBuilderMRRR(  
        20,  
        new BitSequenceBuilderRG(20)));  
SequenceBuilder * sb3 = new SequenceBuilderGMR(  
    new BitSequenceBuilderRG(20), sb2);  
  
SequenceAlphPart * ap = new SequenceAlphPart(seq, N, Ou,  
    sb1, sb3);  
cout << "size = " << ap->getSize() << " bytes" << endl;
```

...

Summary

BitSequence:

- BitSequenceRG
- BitSequenceRRR
- BitSequenceSDArray

Sequence:

- WaveletTree
- WaveletTreeNoptrs
- SequenceGMR
- SequenceGMRChunk
- AlphPart

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications ←

Applications

- Permutation
- Graph
- Binary Relation
- Text – FM-index [Ferragina and Manzini]

Permutations

$$\pi = [1, 3, 7, 4, 2, 5, 6]$$

Permutations

$$\pi = [1, 3, 7, 4, 2, 5, 6]$$

$$\pi(i) = \textit{access}(i)$$

Permutations

$$\pi = [1, 3, 7, 4, 2, 5, 6]$$

$$\pi(i) = \textit{access}(i)$$

$$\pi^{-1}(i) = \textit{select}(i)$$

Permutations

$$\pi = [1, 3, 7, 4, 2, 5, 6]$$

$$\pi(i) = \textit{access}(i)$$

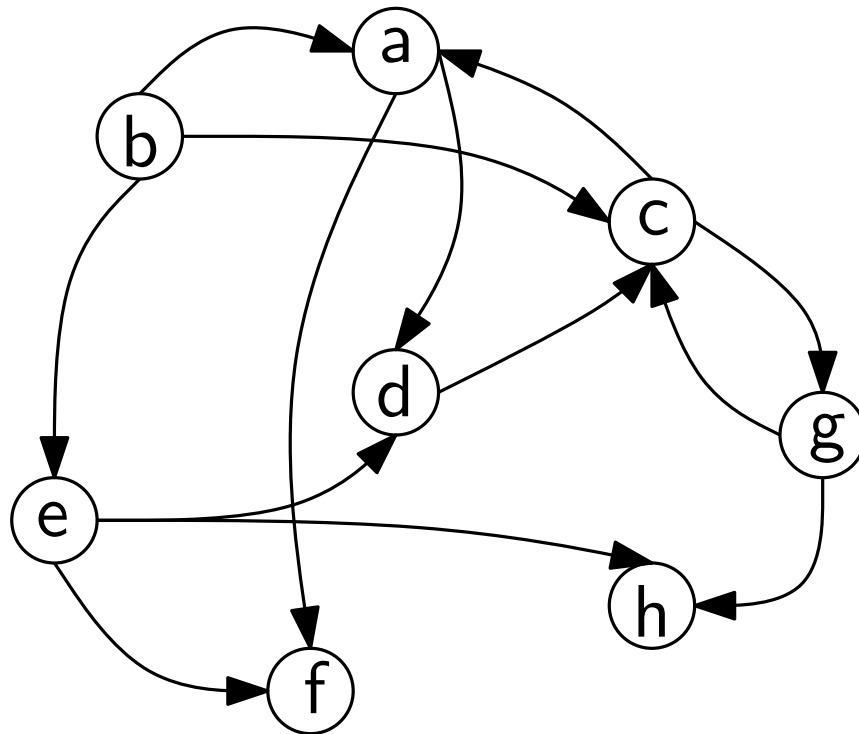
$$\pi^{-1}(i) = \textit{select}(i)$$

MRRR achieves $(1, \log \log n)$

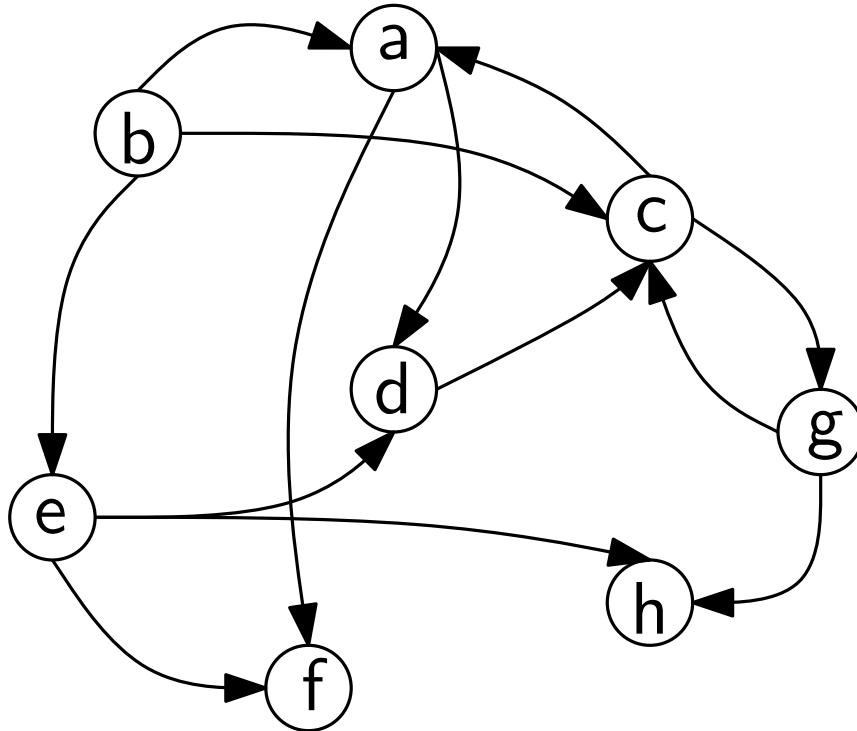
Wavelet Trees achieve $(\log n, \log n)$

GMR achieves $(\log \log n, 1)$

Graphs

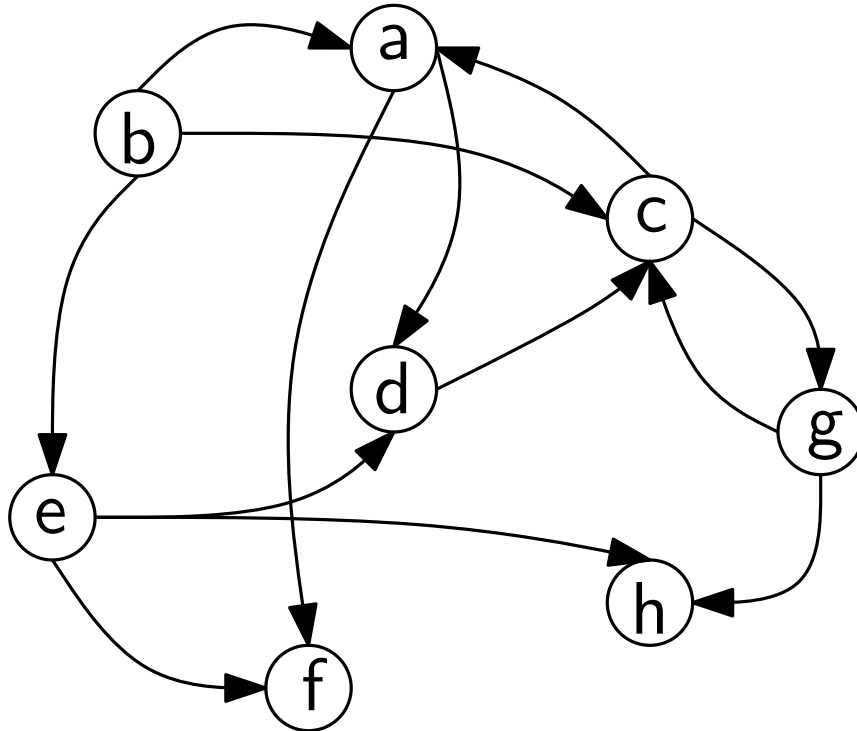


Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

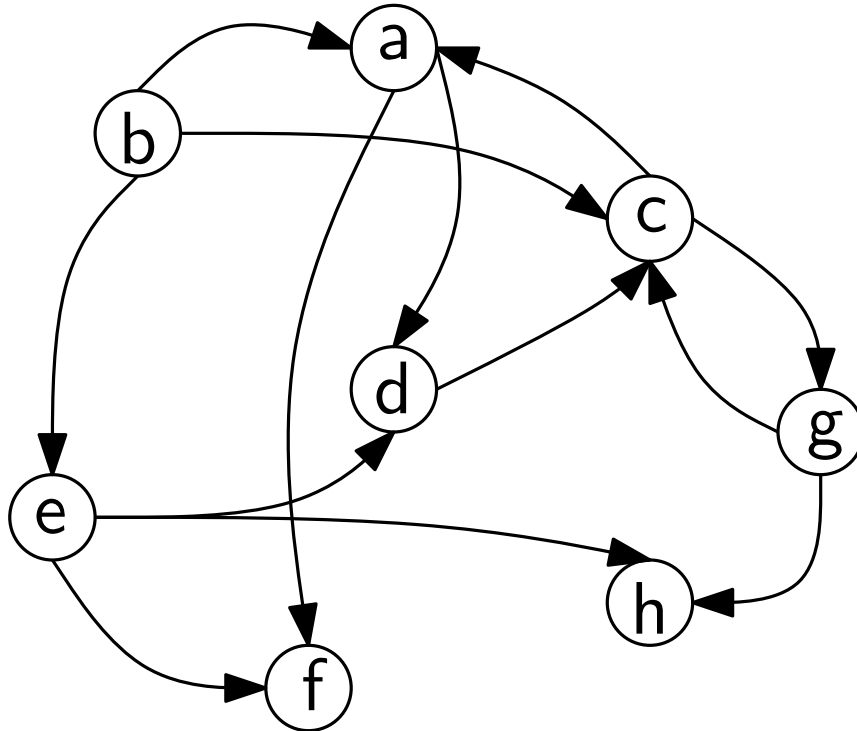
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	1
S	=		d	f		a	c	e		a	g		c		d	f	h			h	

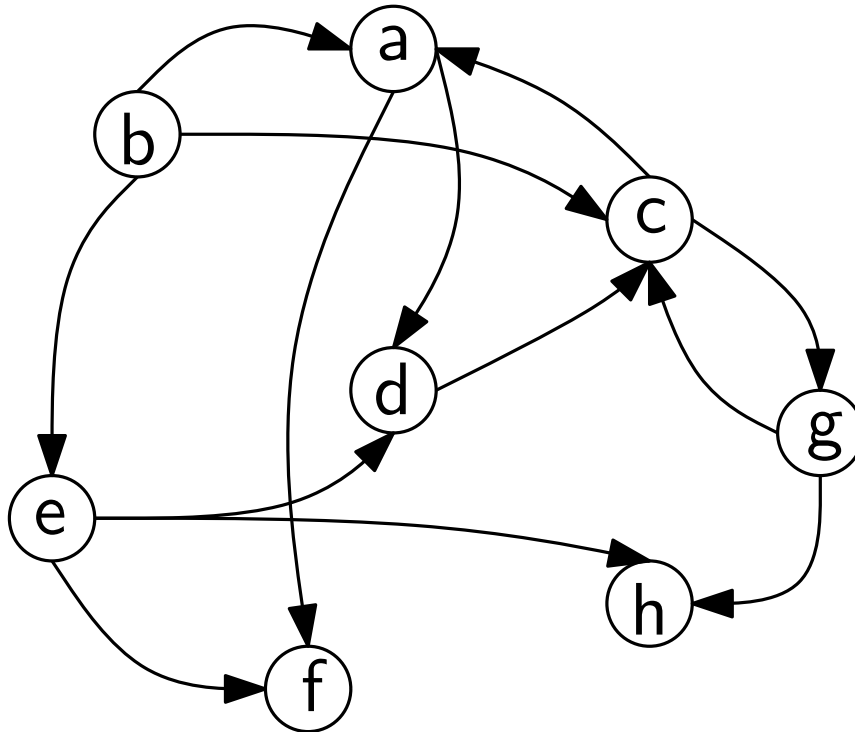
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	1	0	0	0	1	1	0	1
S	=		d	f		a	c	e		a	g		c		d	f	h	

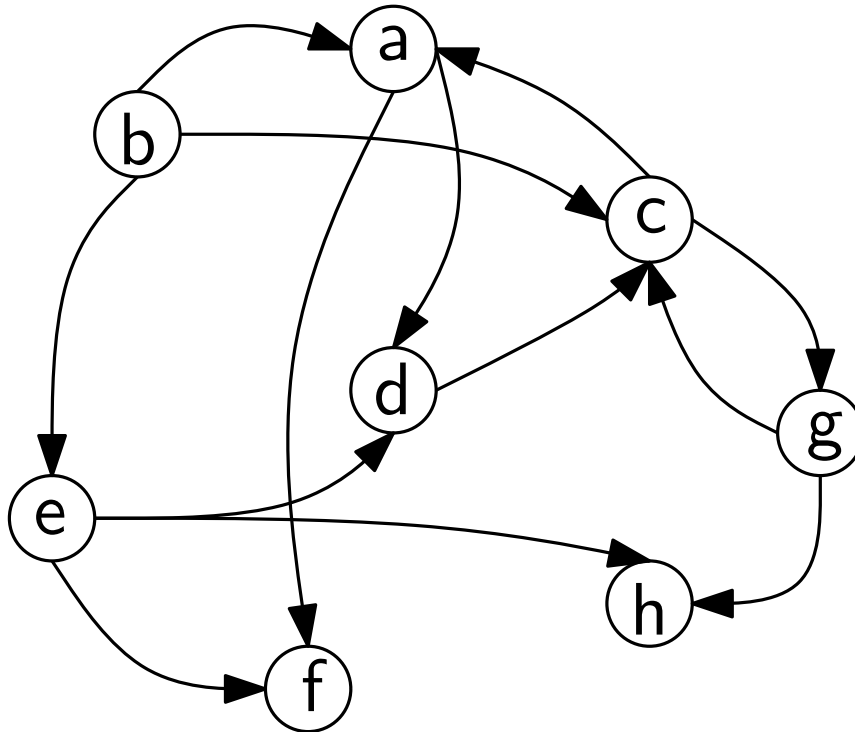
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	1
S	=		d	f		a	c	e		a	g		c		d	f	h			h	

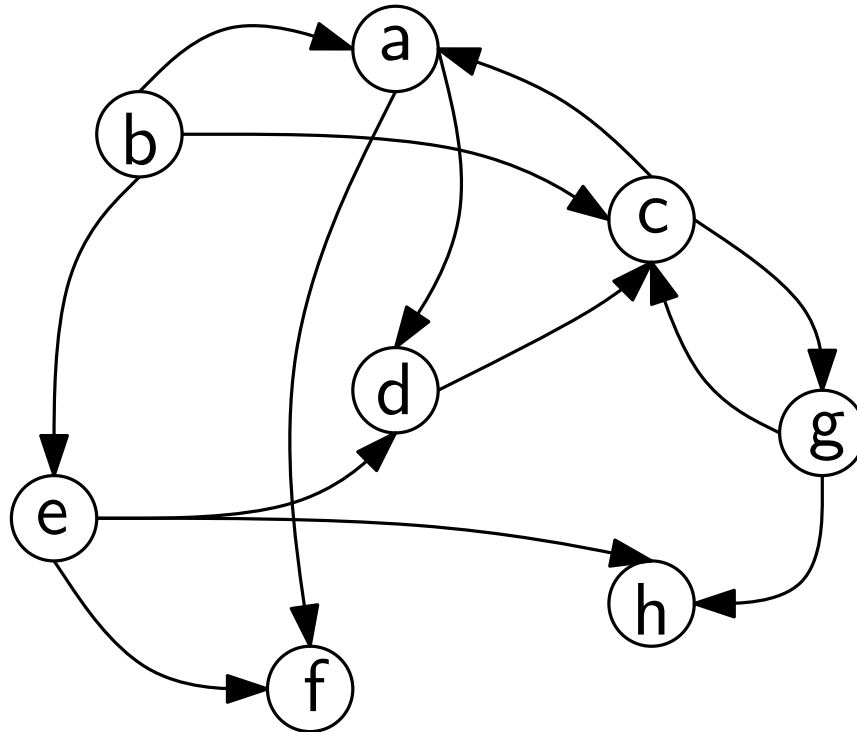
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	1
S	=		d	f		a	c	e		a	g		c		d	f	h			h	

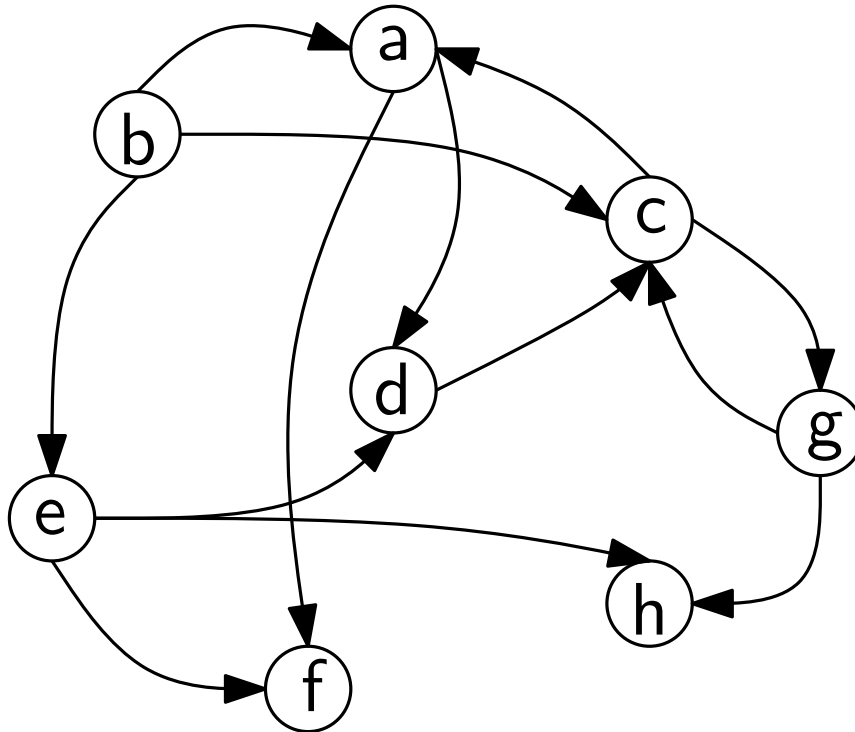
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B = 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 1 0 1
 S = d f a c e a g c d f h h

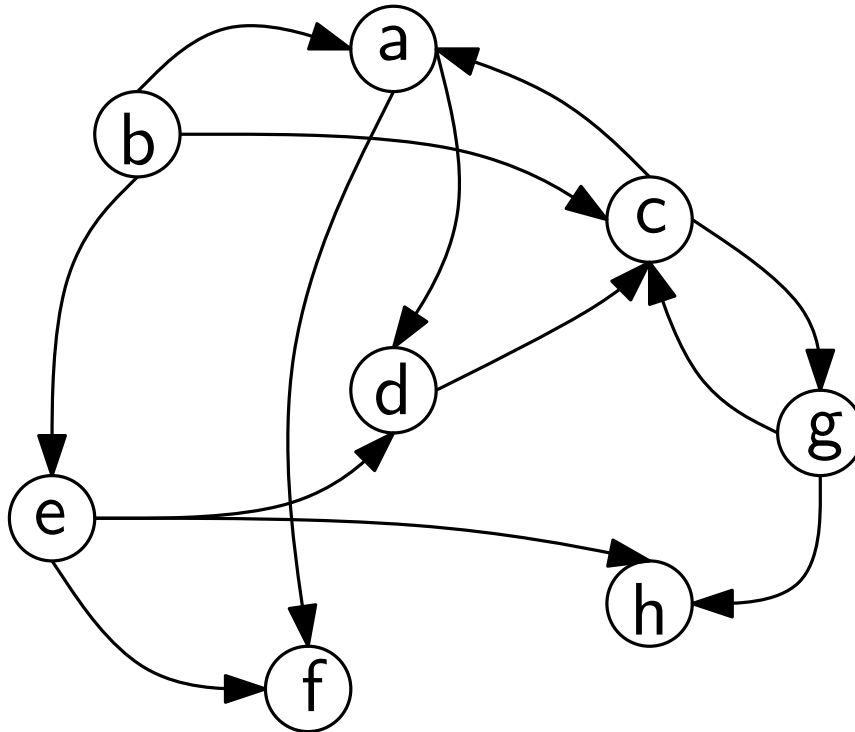
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	1
S	=	d	f		a	c	e		a	g	c		d	f	h					h	

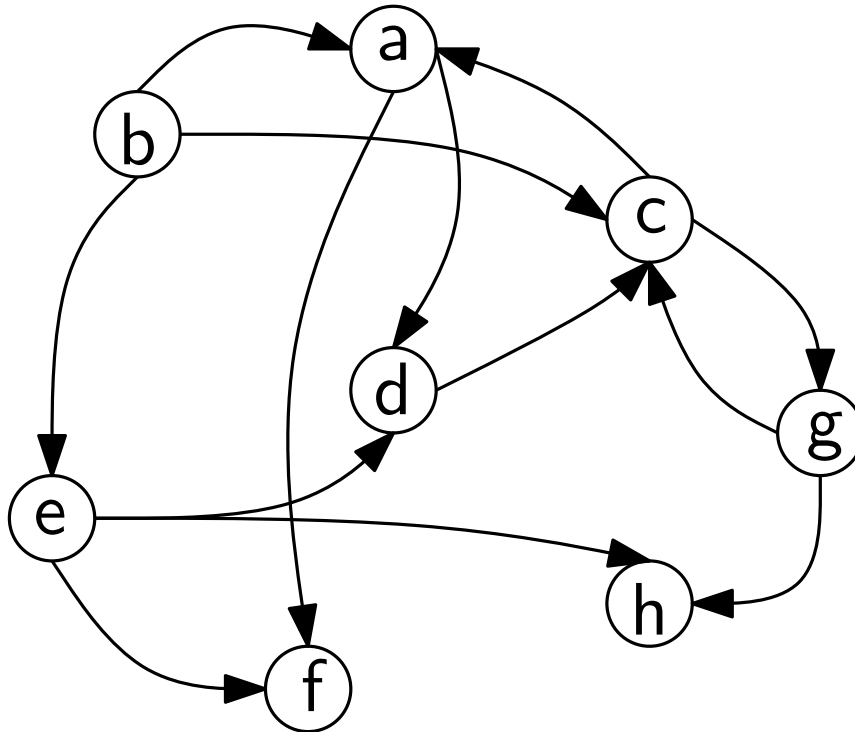
Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B = 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 1 0 1
 S = d f a c e a g c d f h h

Graphs



a	d,f
b	a,c,e
c	a,g
d	c
e	d,f,h
f	
g	h
h	

B	=	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	1
S	=	d	f		a	c	e		a	g		c	d	f	h					h	

Graphs

- Space: $m \log n(1 + o(1))$
- Retrieve Neighbors: $O(\log \log n)$
- Retrieve Reverse Neighbors: $O(1)$
- Check Connection: $O(\log \log n)$

Graphs

- Space: $m \log n(1 + o(1))$
- Retrieve Neighbors: $O(\log \log n)$
- Retrieve Reverse Neighbors: $O(1)$
- Check Connection: $O(\log \log n)$

Adjacency list requires $n \log m + m \log n$

- Neighbors $O(1)$
- Reverse Neighbors?
- Check Connection?

Graphs

- Space: $m \log n(1 + o(1))$
- Retrieve Neighbors: $O(\log \log n)$
- Retrieve Reverse Neighbors: $O(1)$
- Check Connection: $O(\log \log n)$










Adjacency list requires $n \log m + m \log n$

- Neighbors $O(1)$
- Reverse Neighbors?
- Check Connection?

Adjacency matrix requires n^2

- Neighbors: $O(n)$
- Reverse Neighbors: $O(n)$
- Check Connection: $O(1)$

Binary Relations

Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

1, 3

Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

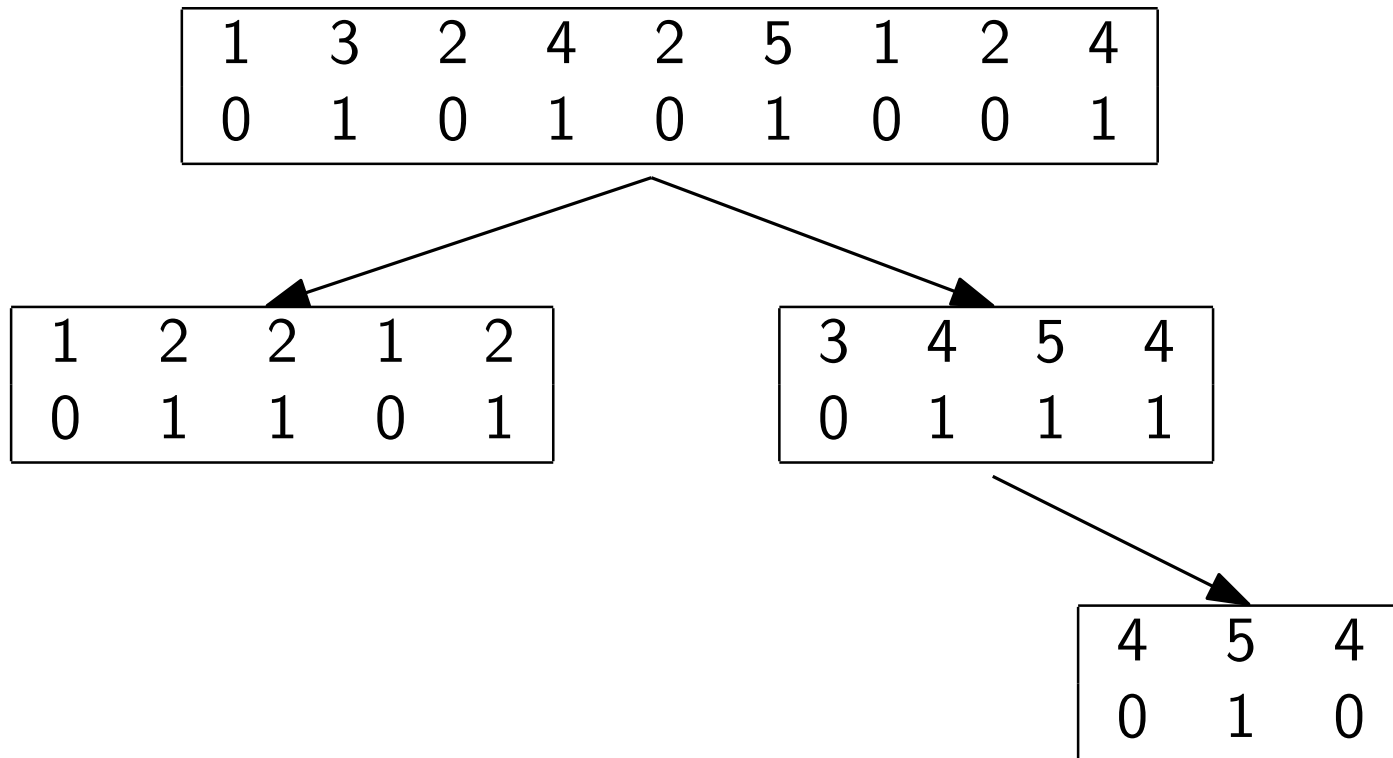
1, 3

2, 4

B	1	0	0	1	0	0	1	0	0	1	0	0	1	0
T		1	3		2	4		2	5		1	2		4

Binary Relations

B	1	0	0	1	0	0	1	0	0	1	0	0	1	0
T		1	3		2	4		2	5		1	2		4



Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

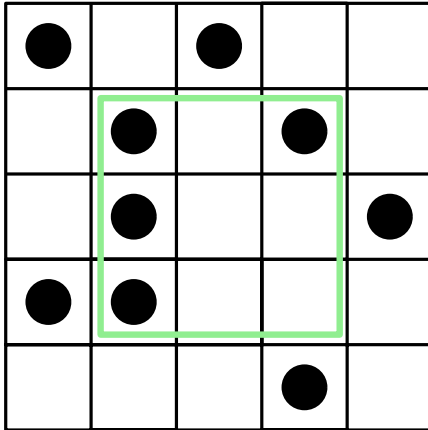
1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0

Binary Relations



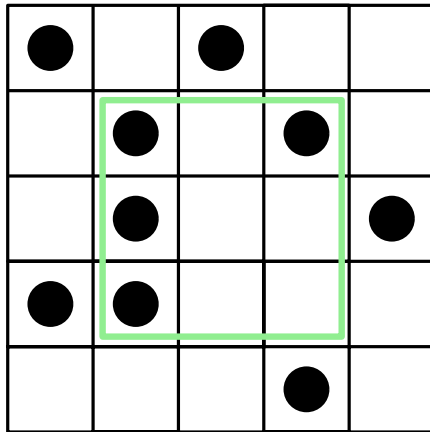
1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0

Binary Relations



1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0

Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0



Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0



Binary Relations

●		●		
	●		●	
	●			●
●	●			
			●	

1	3	2	4	2	5	1	2	4
0	1	0	1	0	1	0	0	1

1	2	2	1	2
0	1	1	0	1

3	4	5	4
0	1	1	1

4	5	4
0	1	0



Text

FM-Index

self-index { locate
count
extract
compressed

Text

FM-Index

self-index { locate
count
extract
compressed

Burrows-Wheeler Transform

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	r	d	a	\$	a	l	a	b	a	r	a	l	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	b	a	r	a	\$
a	l	a	b	a	r	a	a	d	a	l	\$	a	b	r	a	l
a	l	a	l	a	b	a	r	d	a	\$	a	b	a	l	a	r
a	r	a	l	a	l	a	b	a	r	d	a	a	\$	l	a	b
a	r	d	a	\$	a	l	a	b	a	r	a	d	a	l	a	b
b	a	r	a	l	a	l	a	b	a	b	a	r	\$	a	l	a
b	a	r	d	a	\$	a	a	l	a	a	r	a	l	a	l	a
d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r
l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a
l	a	l	a	r	d	a	\$	a	l	a	a	b	r	a	l	a
r	a	l	a	b	a	b	a	r	d	a	\$	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	r	d	a	\$	a	l	a	b	a	r	a	l	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	b	a	r	a	\$
a	l	a	b	a	r	a	a	d	a	\$	a	a	b	r	a	l
a	l	a	l	a	b	a	a	b	a	\$	d	a	l	a	a	r
a	r	a	l	a	l	a	l	a	r	d	r	a	\$	l	a	b
a	r	d	a	\$	a	l	a	b	a	b	a	d	a	a	l	b
b	a	r	a	l	\$	a	a	l	a	b	r	a	\$	a	l	a
b	a	r	d	a	\$	a	a	l	a	b	a	l	l	a	l	a
d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r
l	a	b	a	r	a	l	a	l	a	b	a	d	a	\$	l	a
l	a	l	a	r	d	a	\$	a	l	a	a	r	a	a	r	a
r	a	l	a	l	a	b	a	d	\$	a	l	a	a	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

<i>F</i>																<i>L</i>
\$ a a a a a a a a b b d l l l r r	a \$ b b l l l r r a a a a a a a d	l a a a a a a d r r \$ b b l l a	a l r r b l l a d a a a a a a \$	b a d a a a \$ l a l r b l a	a b l a r b l a \$ a \$ a d a a l	r a a \$ d a a l l a b l a r b a	a r l a l a b a l a \$ d a b	l a l a \$ d a b b a r l a r a	a l b l a a r a b a l \$ d r	l a b l \$ d r a l b a a l	a l r a b a a d r a b l l	b a d r a b l \$ l a a l r a a	a b a r a a \$ l a b a a d r b l l	r a \$ l d r b l l a b a a a	d r a a a a a l l a \$ l r b b	a d l l \$ l r b b a a r a a a a a

Text

<i>F</i>																<i>L</i>
\$ a a a a a a a a b b d l l l r r	a \$ b b l l l r r a a a a a a a d	l a a a a a a d r r \$ b b l l a	a l r r b l l a d a a a a a a \$	b a d a a a a \$ l a l r b a	a b l a r b l a \$ a \$ a d a l	r a a \$ d a a l l a b l a r b a	a r l a l a b a l a \$ d a b	l a l a \$ d a b b a r l a r a	a l b a l a r a b a l \$ d r	l a b a l \$ d r a l b a a a	a l r a b a a d r a l a a	b a d r a b l \$ l a a l r a a	a b a r a a \$ l a d r b l l	r a \$ l d r b l a a b a a a	d r a a a a a l l a \$ l r b b	a d l l \$ l r b b a a r a a a a a a

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	r	a	r	a	\$	l	a	b	a	a	r	d	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	a	b	r	a	\$
a	l	a	b	a	r	a	a	d	a	\$	a	a	b	a	a	l
a	l	a	l	a	b	a	l	a	r	d	a	a	\$	l	a	r
a	r	a	l	a	l	a	a	b	a	r	a	a	l	a	a	b
a	r	d	a	\$	a	l	l	b	a	r	d	a	\$	a	l	b
b	a	r	a	l	a	a	a	a	b	a	a	a	a	a	a	a
b	a	r	d	a	\$	a	a	l	a	l	a	l	a	a	a	a
d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	\$	a
l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	l	a
l	a	b	a	r	d	a	\$	a	l	a	a	b	a	a	a	a
l	a	l	a	b	a	r	d	a	\$	a	a	a	a	a	b	a
r	a	l	a	l	a	b	a	r	d	a	l	a	a	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	b	a	r	a	\$	l	a	b	a	a	r	d	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	a	a	r	a	\$
a	l	a	l	a	b	a	a	d	a	\$	d	a	a	b	a	l
a	r	a	l	a	l	a	b	a	r	a	d	a	\$	l	a	r
a	r	d	a	\$	a	l	l	a	a	b	r	a	a	a	l	b
b	a	r	a	l	a	l	a	b	a	a	a	d	\$	a	l	a
b	a	r	d	a	\$	a	b	a	b	l	a	r	a	l	a	a
d	a	\$	a	l	a	b	l	a	r	a	a	l	a	b	\$	r
l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	l	a
l	a	a	a	r	d	a	\$	a	l	a	a	a	b	a	\$	a
l	a	l	a	b	a	r	d	a	\$	a	a	a	a	a	r	a
r	a	l	a	l	a	b	a	r	d	a	l	a	a	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	r	a	r	a	\$	l	a	b	a	a	r	d	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	a	a	r	a	\$
a	l	a	b	a	r	a	a	d	a	\$	a	a	b	l	a	l
a	r	a	l	a	l	a	b	a	r	d	a	a	\$	l	a	r
a	r	d	a	\$	a	l	a	b	a	r	a	d	a	l	a	b
b	a	r	a	l	a	l	a	b	a	b	a	r	a	\$	l	a
b	a	r	d	a	\$	a	b	a	l	a	r	a	l	a	b	a
d	a	\$	a	l	a	b	l	a	r	a	a	l	a	b	a	r
l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a
l	a	a	a	r	d	a	\$	a	l	a	a	b	a	a	l	a
r	a	l	a	b	a	b	d	a	\$	a	a	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$	a	l
a	b	a	r	a	r	a	\$	l	a	b	a	r	a	l	a	l
a	l	a	b	a	r	d	a	\$	a	l	a	a	b	r	a	\$
a	l	a	l	a	b	a	a	d	a	\$	d	a	l	b	a	l
a	r	a	l	a	l	a	b	a	r	a	d	a	\$	l	a	r
a	r	d	a	\$	a	l	a	b	a	b	r	a	\$	a	l	b
b	a	r	a	l	a	l	a	l	a	a	r	d	a	a	l	a
b	a	r	d	a	\$	a	b	a	b	a	l	a	l	a	a	a
d	a	\$	a	l	a	a	l	a	r	a	a	l	a	b	\$	r
l	a	b	a	r	a	d	a	\$	l	a	b	r	a	a	l	a
l	a	a	a	b	a	r	a	d	a	a	l	a	b	a	r	a
l	a	l	a	l	a	b	a	\$	a	a	l	a	a	a	b	a
r	a	a	a	a	a	b	a	d	a	a	\$	a	a	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text

F																	L
\$ a a a a a a a b b d l l l r r	a \$ b b l l l r r a a a a a d	l a a a a a a d r r \$ b b l l a	a l r r b b l l a d a a a a \$	b a a d a a a \$ l a l r r b l a	a b l a r b l a a \$ a d a a l	r a a \$ a d a l l a b l a r b a	a r l a l a r b a l a a \$ d a b	l a a l a \$ d a b b a r l a a r a	a l b a l a r a b a a l \$ d r	l a a b a l \$ d r r a l b a a a	a l r a b a a d r a a b l \$ l	b a d r a b l \$ l a a l r a a a	a b a a r a a \$ l a d r b l l	r a \$ l d r b l l a a b a a a	d r a a a a a l l a \$ l r b b	a d l l \$ l r b b a a r a a a a	

Text

[illegible]

Text Indexing

F	\$	a	a	a	a	a	a	a	a	b	b	d	l	l	l	r	r
L	a	d	l	l	\$	l	r	b	b	a	a	r	a	a	a	a	a

T a l a b a r a l a l a b a r d a \$

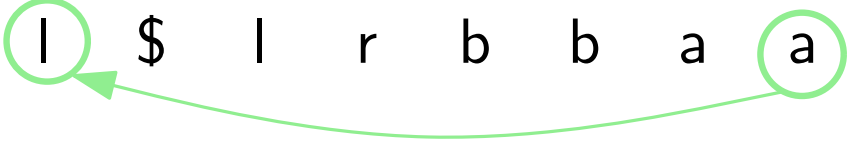
Text Indexing

F	\$	a	a	a	a	a	a	a	a	b	b	d	l	l	l	r	r
L	a	d	l	l	\$	l	r	b	b	a	a	r	a	a	a	a	a

T a l a b a r a l a l a b a r d a \$

Text Indexing

F \$ a a a a a a a b b d l l l r r
L a d l l \$ l r b b a a r a a a a a



A diagram illustrating a transition in a text indexing structure. Two green circles highlight the character 'l' in the second row (index 5) and the character 'a' in the second row (index 11). A green curved arrow points from the 'a' circle to the 'l' circle, indicating a transition from the state 'a' to the state 'l'.

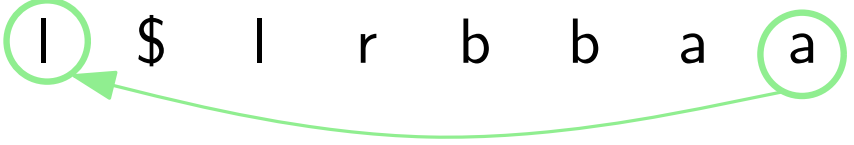
T a l a b a r a l a l a b a r d a \$



A diagram illustrating a transition in a text indexing structure. Two green circles highlight the character 'l' at index 10 and the character 'a' at index 11. A green curved arrow points from the 'l' circle to the 'a' circle, indicating a transition from the state 'l' to the state 'a'.

Text Indexing

F	\$	a	a	a	a	a	a	a	a	b	b	d	l	l	l	r	r
L	a	d	l	l	\$	l	r	b	b	a	a	r	a	a	a	a	a



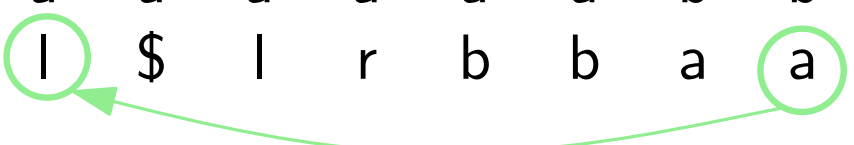
$$LF(i) = rank(T^{BWT}[i], i) + occ[T^{BWT}[i]]$$

T a l a b a r a l a l a b a r d a \$



Text Indexing

F	\$	a	a	a	a	a	a	a	a	b	b	d	l	l	l	r	r
L	a	d	l	l	\$	l	r	b	b	a	a	r	a	a	a	a	a



$$LF(i) = rank(T^{BWT}[i], i) + occ[T^{BWT}[i]]$$

T	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----



Text Indexing

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	\$	a	b	a	b	r	a	\$	a	l
a	l	a	b	a	r	a	l	a	\$	a	l	a	b	l	a	\$
a	l	a	l	a	r	b	a	\$	d	a	\$	a	l	r	a	l
a	r	a	l	a	l	a	a	b	a	r	d	a	\$	l	a	r
a	r	d	a	\$	a	l	a	b	a	a	r	a	l	a	a	b
b	a	r	a	l	\$	l	a	b	a	b	a	d	\$	a	l	a
b	a	r	d	a	\$	a	l	a	r	a	r	a	l	a	l	a
d	a	\$	a	l	a	b	a	l	a	l	a	l	a	b	a	r
l	a	b	a	r	a	l	a	a	\$	a	b	r	d	a	\$	a
l	a	l	a	b	d	a	\$	a	a	a	a	a	a	a	l	a
r	a	l	a	l	a	b	a	r	d	a	\$	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Text Indexing

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	r	a	l	a	l	a	b	a	r	d
a	b	a	r	a	l	a	\$	a	l	a	b	r	a	\$	a	l
a	b	a	r	a	r	a	\$	a	l	a	b	a	r	l	a	\$
a	l	a	b	a	r	a	d	a	\$	a	l	a	b	r	a	l
a	l	a	l	a	b	a	r	b	a	d	a	\$	a	l	a	l
a	r	a	l	a	l	a	l	a	b	r	a	l	a	l	a	l
a	r	d	a	\$	a	l	a	l	a	b	a	l	a	\$	a	l
b	a	r	a	l	\$	a	\$	a	b	r	a	a	l	a	b	a
b	a	r	d	a	\$	a	a	b	a	l	a	l	a	a	a	a
d	a	\$	a	l	a	b	l	a	r	l	a	r	a	b	\$	a
l	a	b	a	r	a	a	a	l	a	b	a	a	d	a	l	a
l	a	b	a	b	d	a	\$	a	a	a	a	a	r	a	r	a
l	a	l	a	l	a	b	r	a	l	a	l	a	a	a	b	a
r	a	l	a	l	a	a	a	a	d	a	\$	a	l	a	a	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Search for *lala*

Text Indexing

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	r	a	l	a	l	b	a	r	r	d
a	b	a	r	a	l	a	\$	a	l	a	b	r	a	\$	a	l
a	b	a	r	a	r	a	a	l	a	b	a	d	a	r	a	l
a	l	a	b	a	r	d	a	\$	d	a	l	a	b	r	a	\$
a	l	a	l	a	b	a	r	b	a	r	d	a	l	a	a	l
a	r	a	l	a	l	a	l	a	b	r	a	\$	a	l	a	r
a	r	d	a	\$	a	l	a	l	a	a	d	a	\$	a	l	a
b	a	r	a	l	\$	a	\$	a	b	a	r	a	l	a	b	a
b	a	r	d	a	\$	a	a	b	a	l	a	a	l	a	a	a
d	a	\$	a	l	a	b	a	r	l	b	a	l	a	b	\$	a
l	a	b	a	r	a	l	a	\$	a	a	b	r	d	a	l	a
l	a	l	a	b	d	a	r	d	a	a	l	a	b	a	r	a
r	a	l	a	l	a	b	a	a	r	a	\$	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Search for *lala*

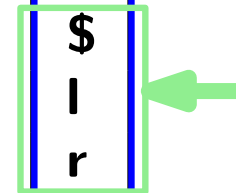
Text Indexing

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	a	l	a	l	a	b	a	r	a	d
a	b	a	r	a	l	a	\$	a	l	a	b	r	a	\$	a	l
a	l	a	b	a	r	a	l	a	\$	a	l	a	b	r	a	l
a	l	a	l	a	r	a	b	a	r	d	a	\$	a	l	a	\$
a	r	a	l	a	l	a	l	a	b	r	a	l	a	l	a	r
a	r	d	a	\$	a	l	a	b	a	r	a	d	a	\$	a	a
b	a	r	a	l	\$	a	l	a	b	a	r	a	l	a	l	a
b	a	r	d	a	\$	a	l	a	b	a	l	a	l	a	l	a
d	a	\$	a	l	a	b	a	r	l	a	b	a	l	a	\$	a
l	a	b	a	r	d	a	\$	a	l	a	a	r	d	a	l	a
l	a	l	a	b	a	r	b	a	r	a	l	a	b	a	r	a
r	a	l	a	l	a	a	a	a	d	a	\$	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a

Search for *lala*

Text Indexing

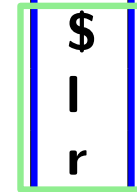
<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	a	l	a	l	a	b	a	r	a	d
a	b	a	r	a	l	a	\$	a	l	a	b	r	a	\$	a	l
a	l	a	b	a	r	a	l	a	\$	a	l	a	b	r	a	l
a	l	a	l	a	r	b	a	r	d	a	\$	a	l	a	a	\$
a	r	a	l	a	l	a	l	a	b	r	d	a	\$	a	a	l
a	r	d	a	\$	a	l	a	l	a	b	a	l	a	\$	a	l
b	a	r	a	l	\$	a	l	a	b	a	b	a	\$	a	l	a
b	a	r	d	a	\$	a	l	a	b	a	r	a	l	a	b	a
d	a	\$	a	l	a	b	a	l	a	l	a	l	a	b	a	\$
l	a	b	a	r	a	l	a	\$	a	b	a	r	d	a	\$	l
l	a	l	a	b	a	r	a	d	a	a	l	a	b	a	r	a
r	a	l	a	l	a	b	a	r	a	l	a	a	l	a	b	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a



Search for *lala*

Text Indexing

<i>F</i>																<i>L</i>
\$	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a
a	\$	a	l	a	b	r	a	l	a	l	a	b	a	r	a	d
a	b	a	r	a	l	a	\$	a	l	a	b	r	a	\$	a	l
a	l	a	b	a	r	a	d	a	\$	a	l	a	b	r	a	\$
a	l	a	l	a	r	b	a	r	d	a	\$	a	l	a	a	l
a	r	a	l	a	l	a	l	a	b	a	r	d	a	l	a	l
a	r	d	a	\$	a	l	a	l	a	b	a	l	a	\$	a	l
b	a	r	a	l	\$	a	l	a	b	a	r	a	\$	a	l	a
b	a	r	d	a	\$	a	l	a	b	a	l	a	l	a	b	a
d	a	\$	a	l	a	b	a	l	a	l	a	l	a	b	a	\$
l	a	b	a	r	a	l	a	\$	a	b	a	r	d	a	l	r
l	a	l	a	b	d	a	r	d	a	a	l	a	b	a	a	a
l	a	l	a	r	a	b	a	a	\$	a	l	a	a	a	b	a
r	a	l	a	l	a	b	a	r	d	a	\$	a	l	a	a	a
r	d	a	\$	a	l	a	b	a	r	a	l	a	l	a	b	a



Search for *lala* 1 occ

Text Indexing

Representing L with a wavelet tree

count: $O(m \log \sigma)$

extract: $O((s_t + \ell) \log \sigma)$

locate: $O((m + s_a occ) \log \sigma)$

Text Indexing

Representing L with a wavelet tree

count: $O(m \log \sigma)$

extract: $O((s_t + \ell) \log \sigma)$

locate: $O((m + s_a occ) \log \sigma)$

space: $nH_k(T) + o(n \log \sigma)$

The road ahead...

Improving what we already have

The road ahead...

Improving what we already have

Construction of these structures

The road ahead...

Improving what we already have

Construction of these structures

More technical problems

The road ahead...

Improving what we already have

Construction of these structures

More technical problems

LIBCDS2!

Links

- <http://libcds.recoded.cl/>
- <https://github.com/fclaude/libcds>
- <https://github.com/fclaude/libcds2>

Thanks for your attention

The End

