Space-Efficient Data Structures

Francisco Claude
Gonzalo Navarro

Outline

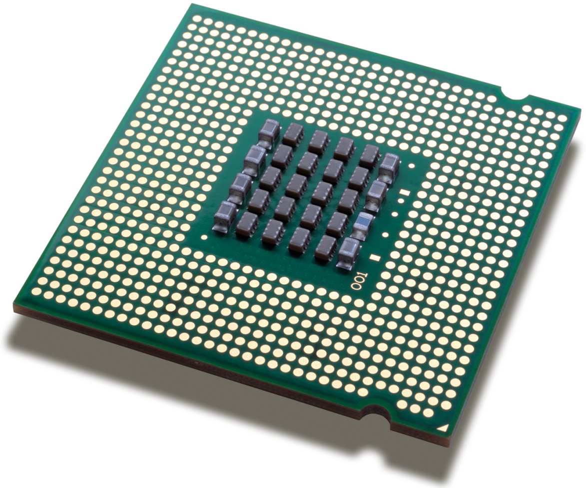
- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

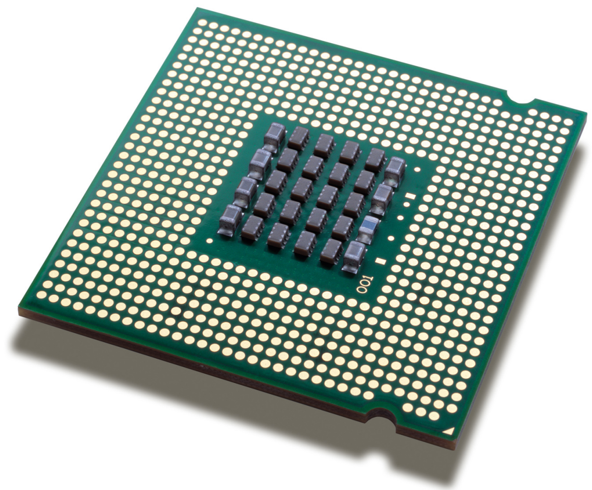
Motivation

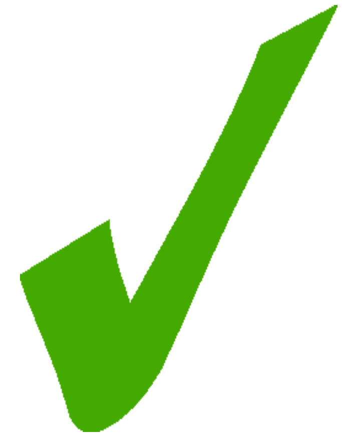
- Processor speed increasing



Motivation

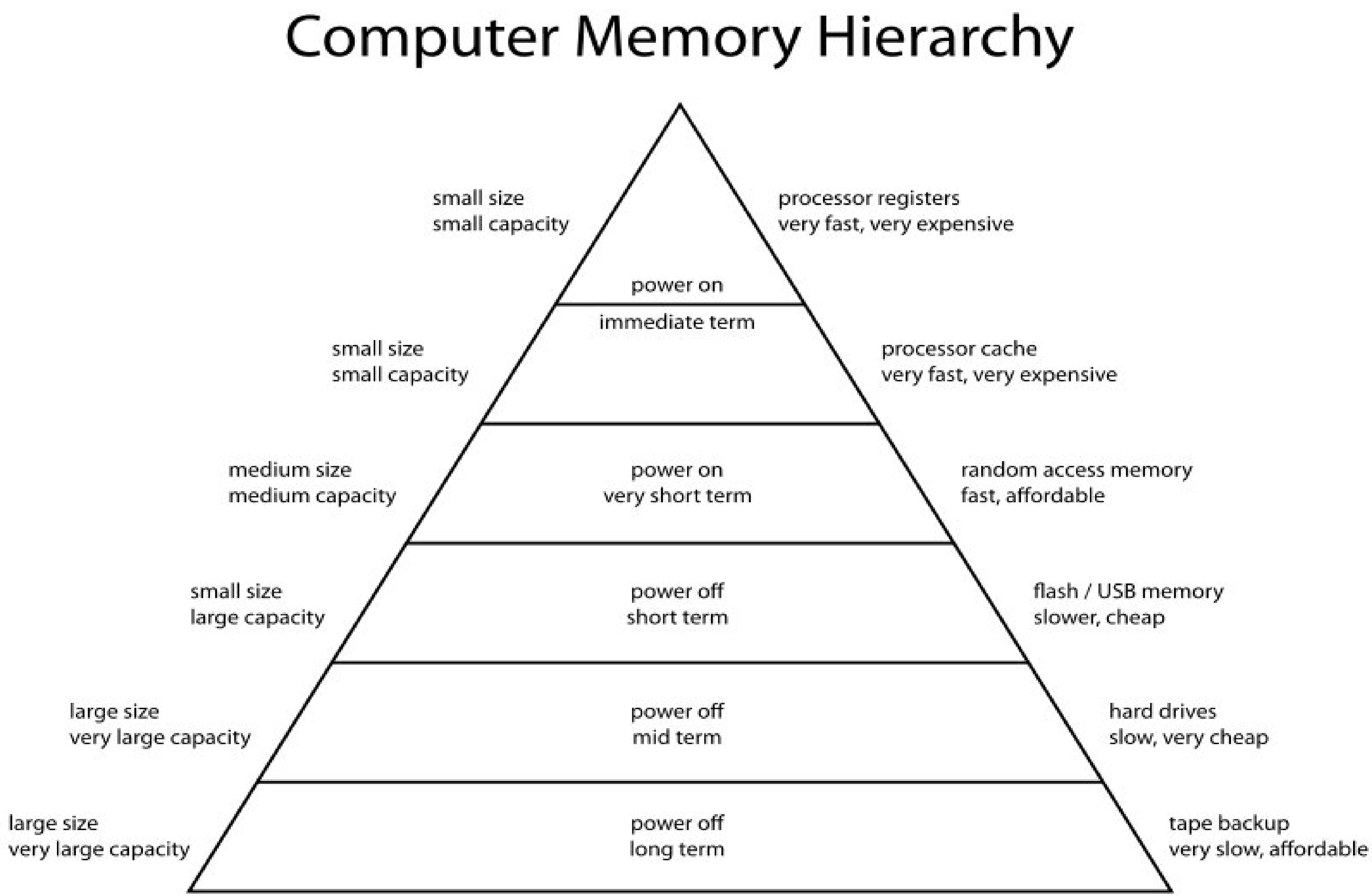
- Processor speed increasing





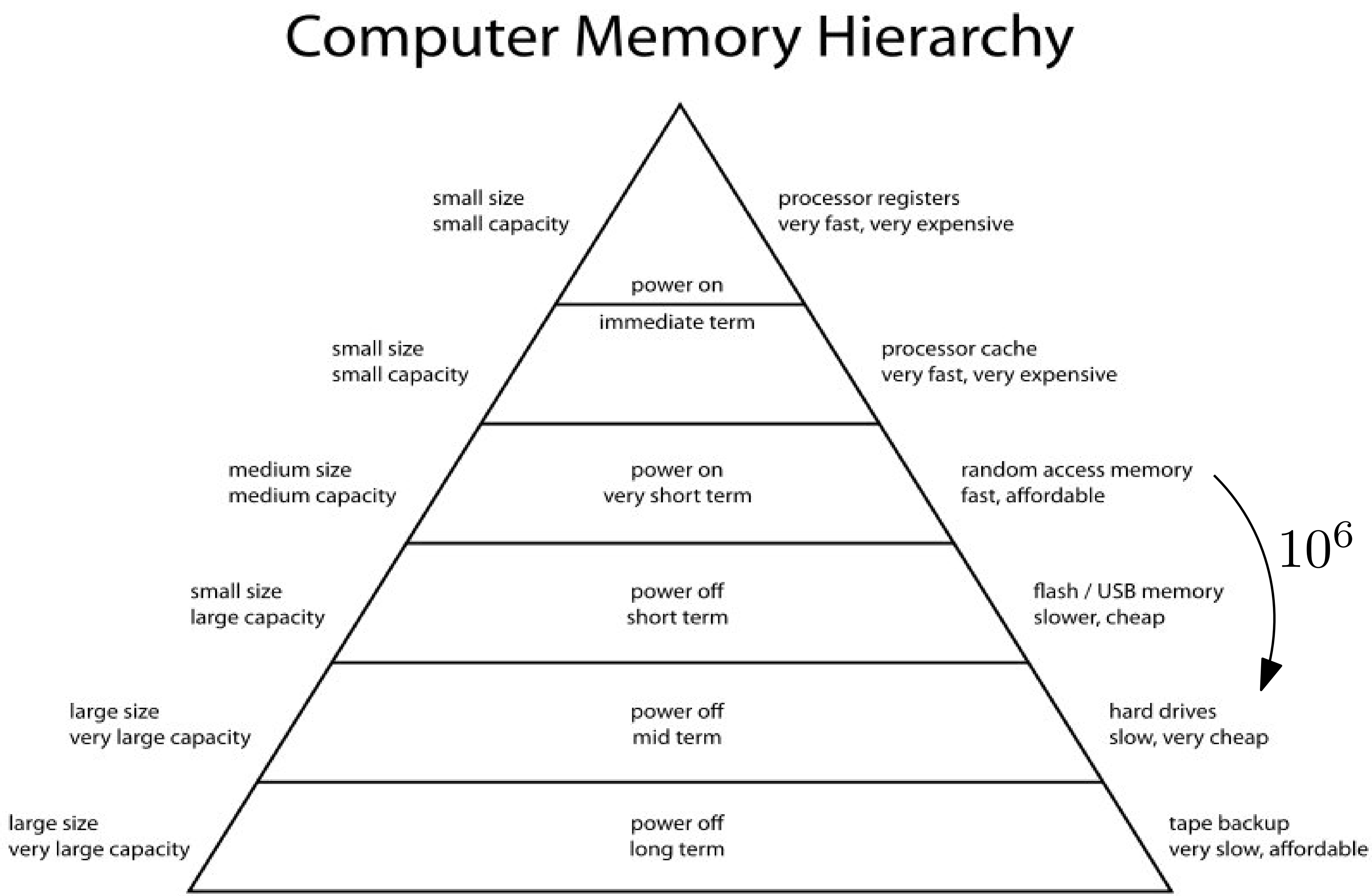
- Disks have not evolved at the same pace

Motivation



source: Wikipedia

Motivation



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!

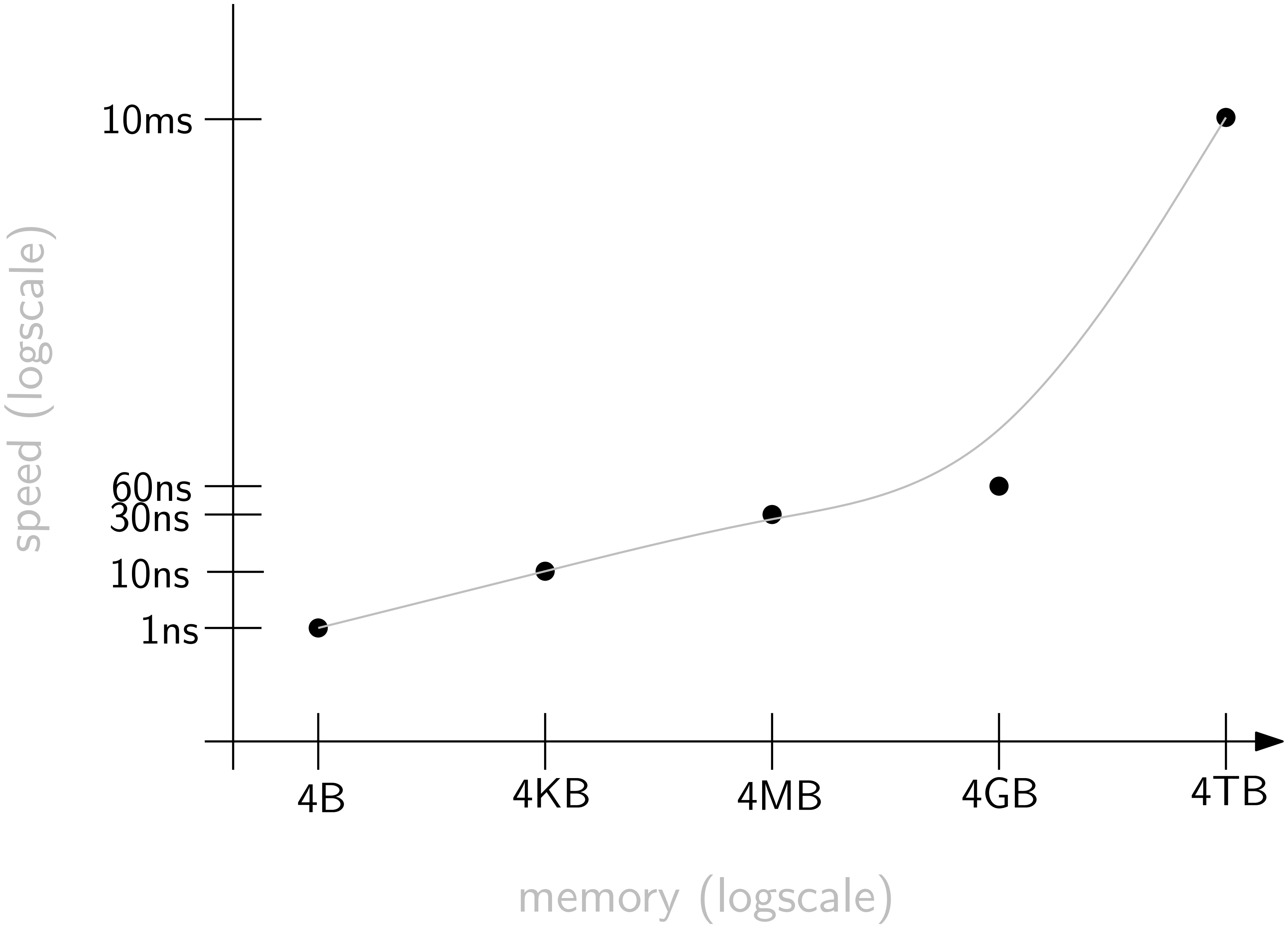
Motivation

Web Graph
uk-union-2006-06-2007-05
Nodes: $133,633,040$
Edges: $5,507,679,822$

A plain representation
requires 22GB!

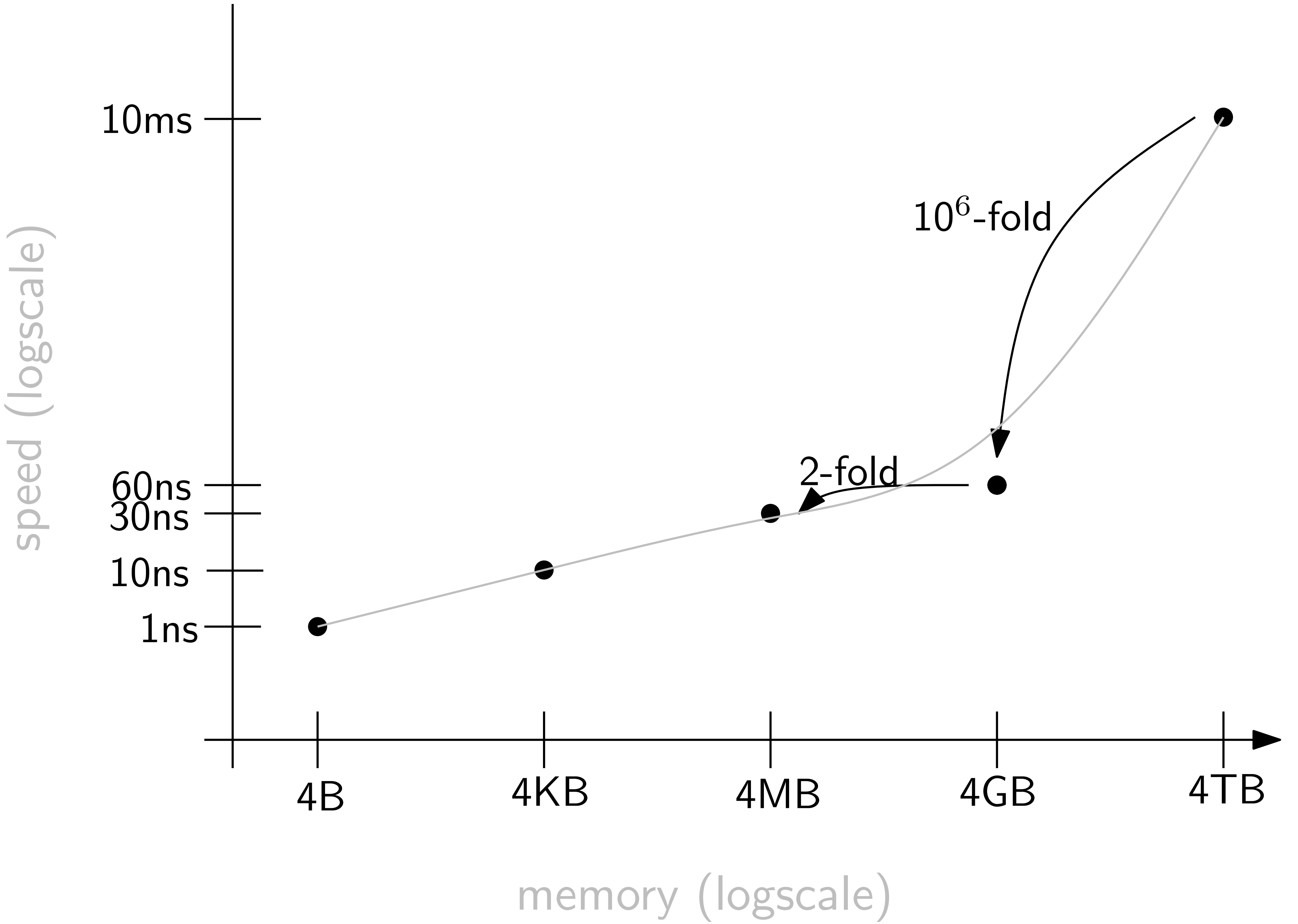
If we use a space-efficient representation: < 2GB

Motivation



10ms
60ns
30ns
10ns
1ns
speed (logscale)
4B
4KB
4MB
4GB
4TB
memory (logscale)

Motivation



10ms
60ns
30ns
10ns
1ns
speed (logscale)
10^6-fold
2-fold
4B
4KB
4MB
4GB
4TB
memory (logscale)

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 = 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

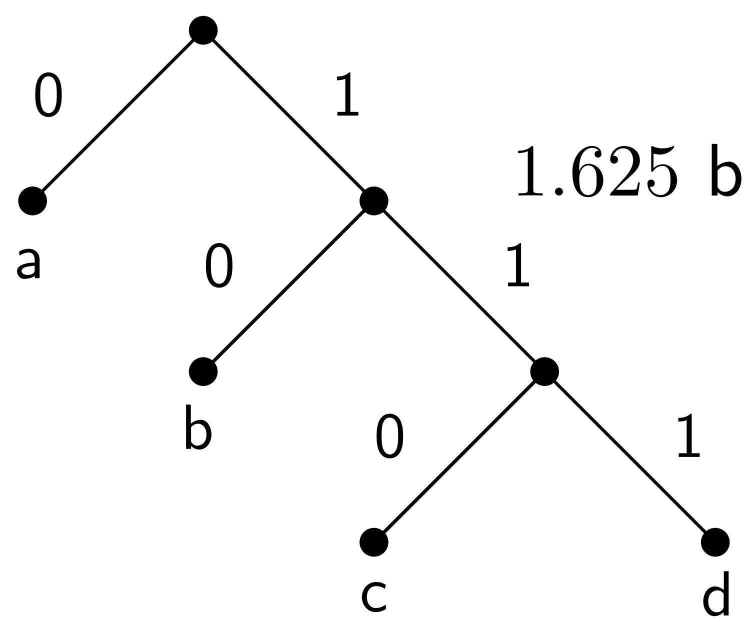
$$\mathsf{S} = \mathsf{aaabbcaaabbcaaad}$$

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

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

$$H_0(S) = 1.5919$$

Huffman



1.625 bpc

High-order Compression

Can we exploit other things?

$$\mathsf{S} = \mathsf{aaabbcaaabbcaaad}$$

High-order Compression

Can we exploit other things?

$$\mathsf{S} = \mathsf{aaabbcaaabbcaaad}$$

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

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

$$H_1(S) = 0.9387$$

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

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

$\Rightarrow 2n + o(n)$ bits

Outline

- Motivation
- Basics
- Bitmaps
- Sequences
- Applications

Outline

- Motivation
- Basics
- Bitmaps ←
- Sequences
- Applications

Bitmaps

$$B = 1001110000111100000$$

Bitmaps

$$B = \boxed{1001110000111}100000\boxed{0}$$

$rank(13) = 7$ $access(19) = 0$

$select(8) = 14$

Bitmaps

$$B = \boxed{1001110000111}100000\boxed{0}$$

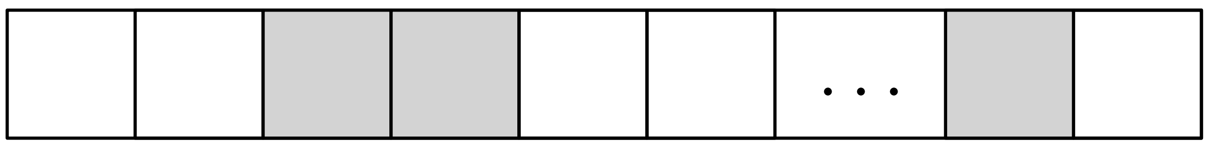
$rank(13) = 7$ $access(19) = 0$

$select(8) = 14$

Is this useful?

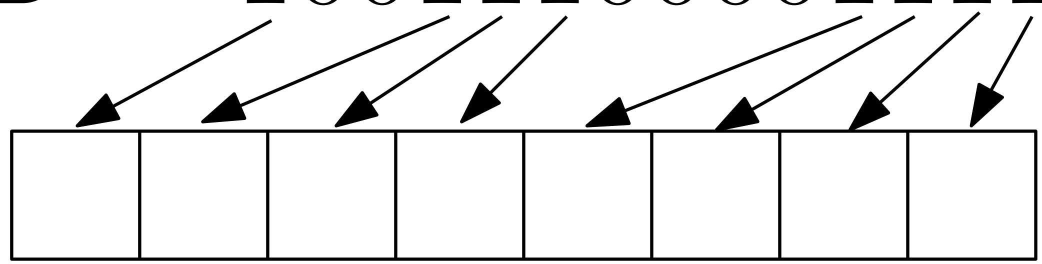
Hashing Example

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



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

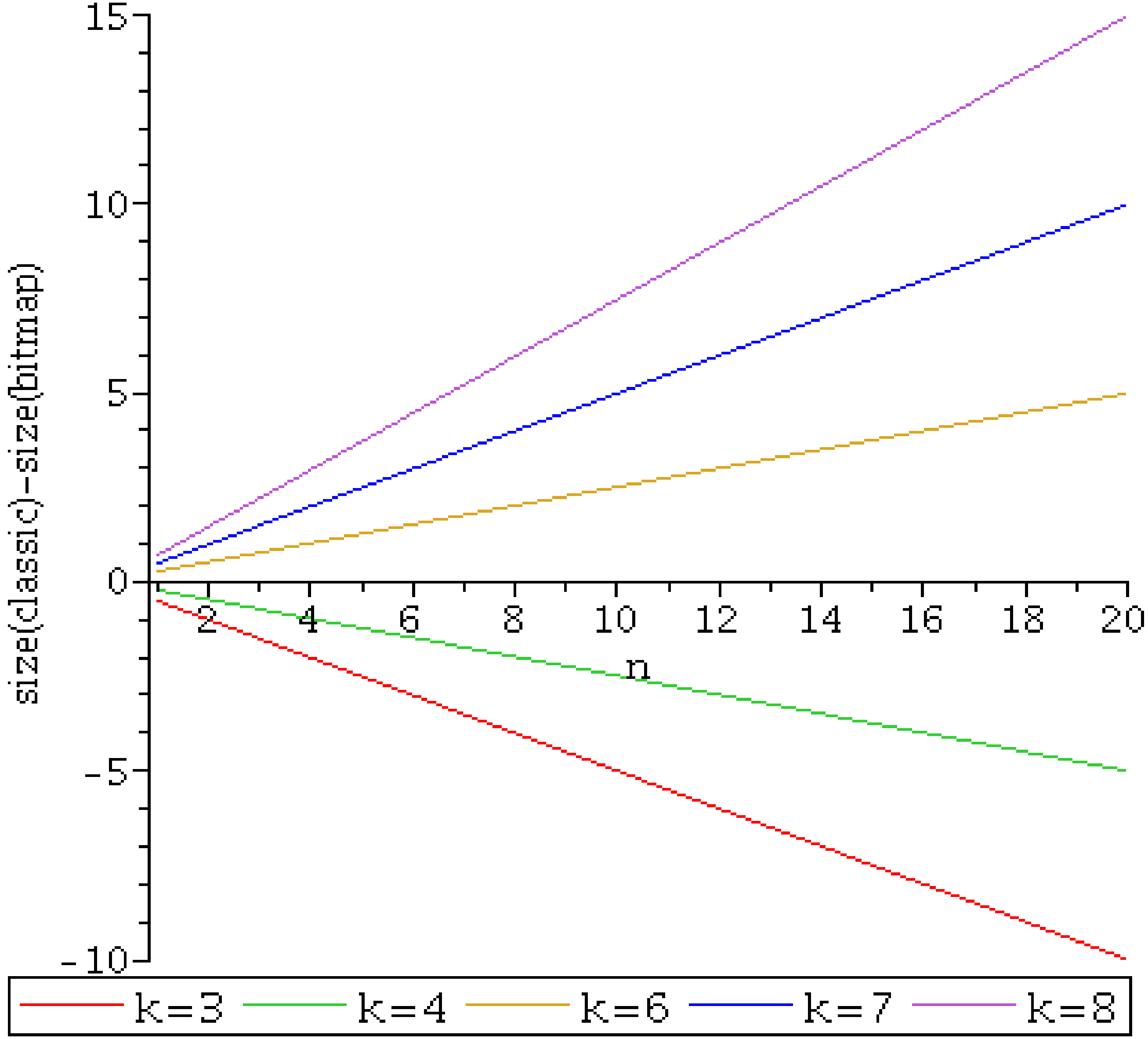
$$B = 1001110000111100000 \quad n/\alpha$$



nk

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

Hashing Example



Bitmaps: Example Applications

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

Bitmaps

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

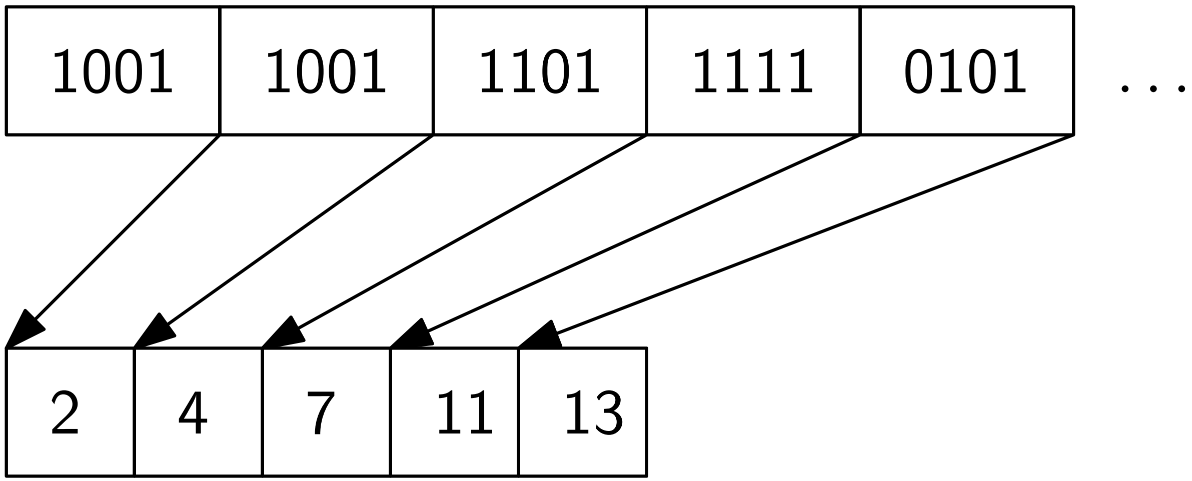
Rank

$$B = 1001010101110101011\ldots$$

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

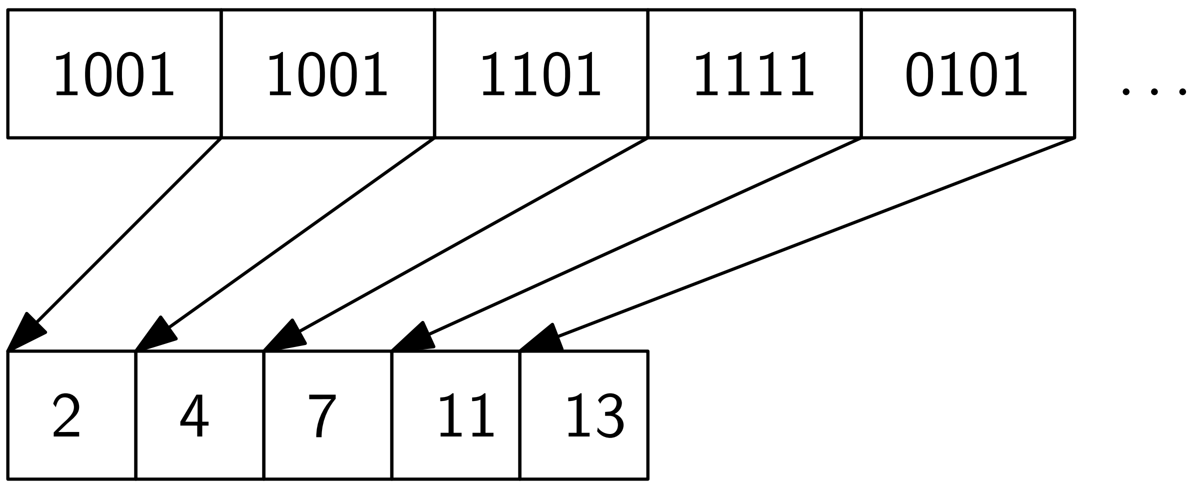
$O(1)$ rank
$n \lg n$ bits of space

Rank



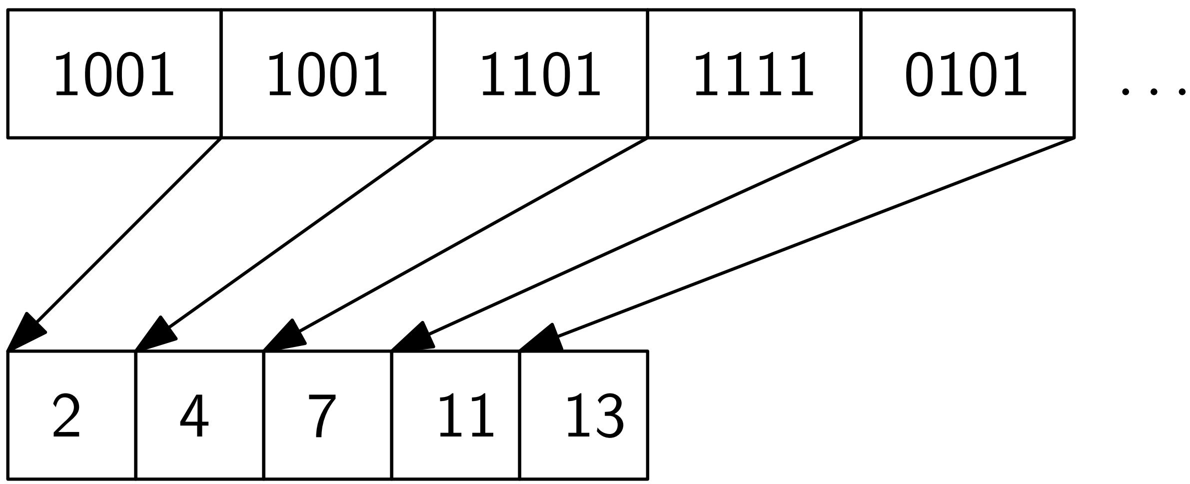
If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

Rank

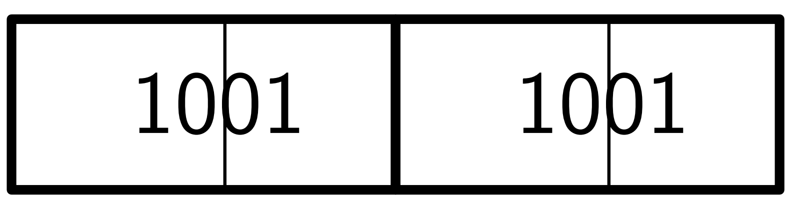


If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

Rank

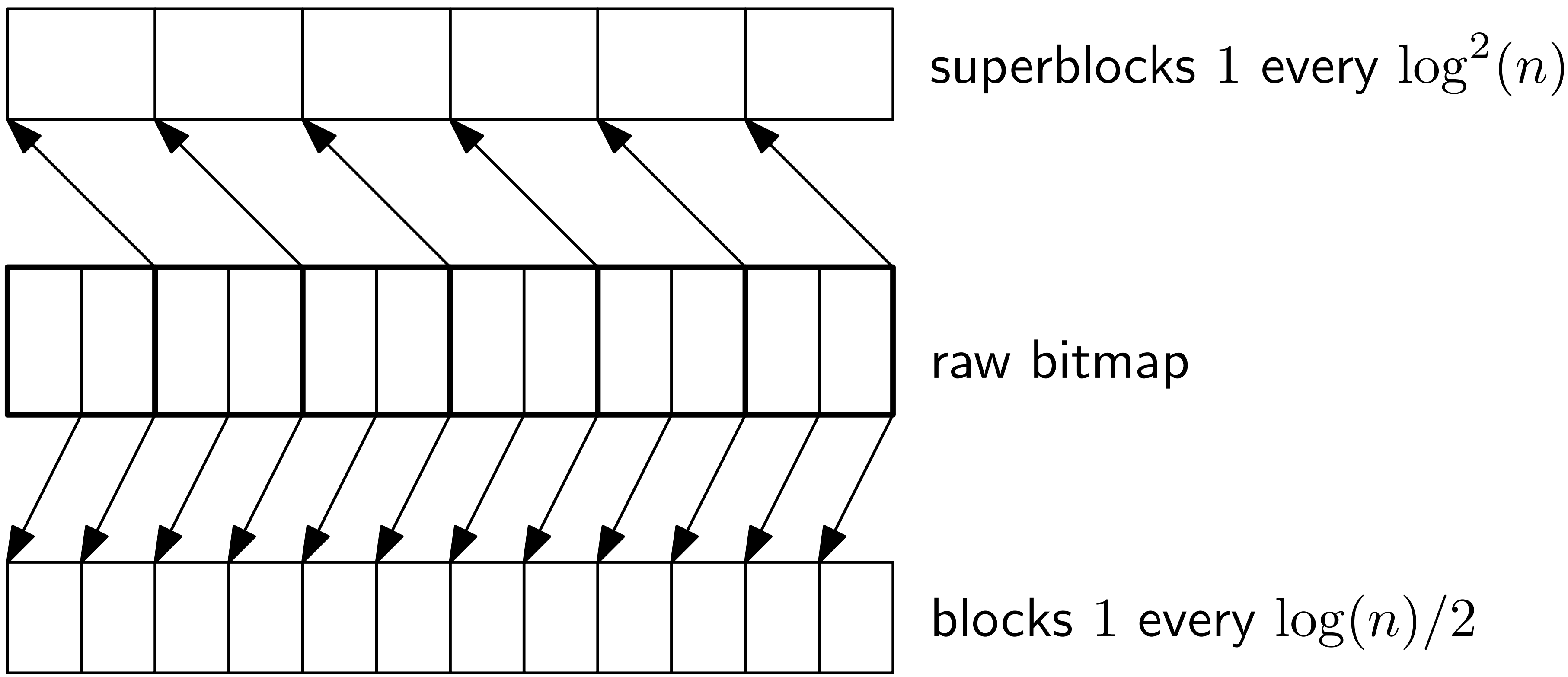


If we sample every s bits, we require $\frac{n \lg n}{s}$ bits. Rank takes $O(s)$ time.

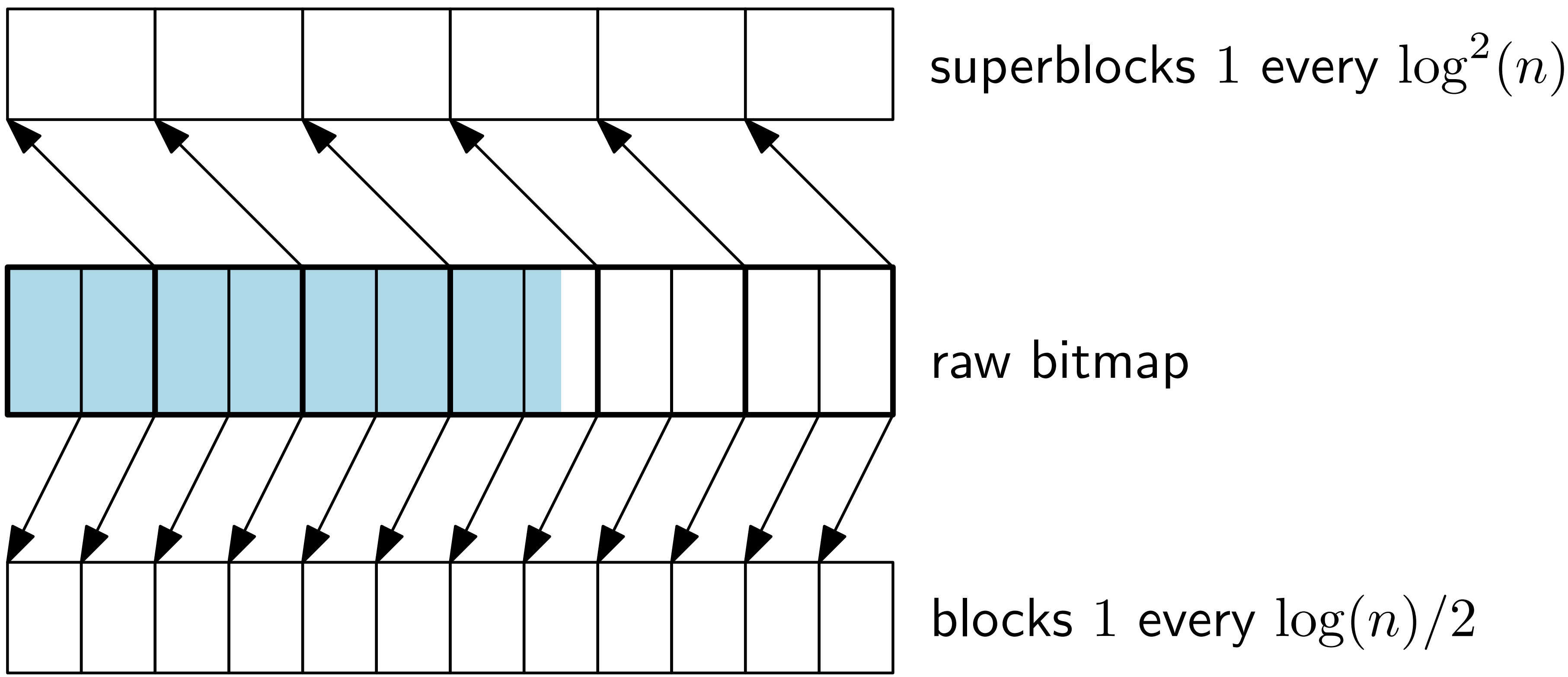


Recurse inside blocks sampling every b. This requires $\frac{n \lg s}{b}$ bits and now we can answer in $O(b)$ time.

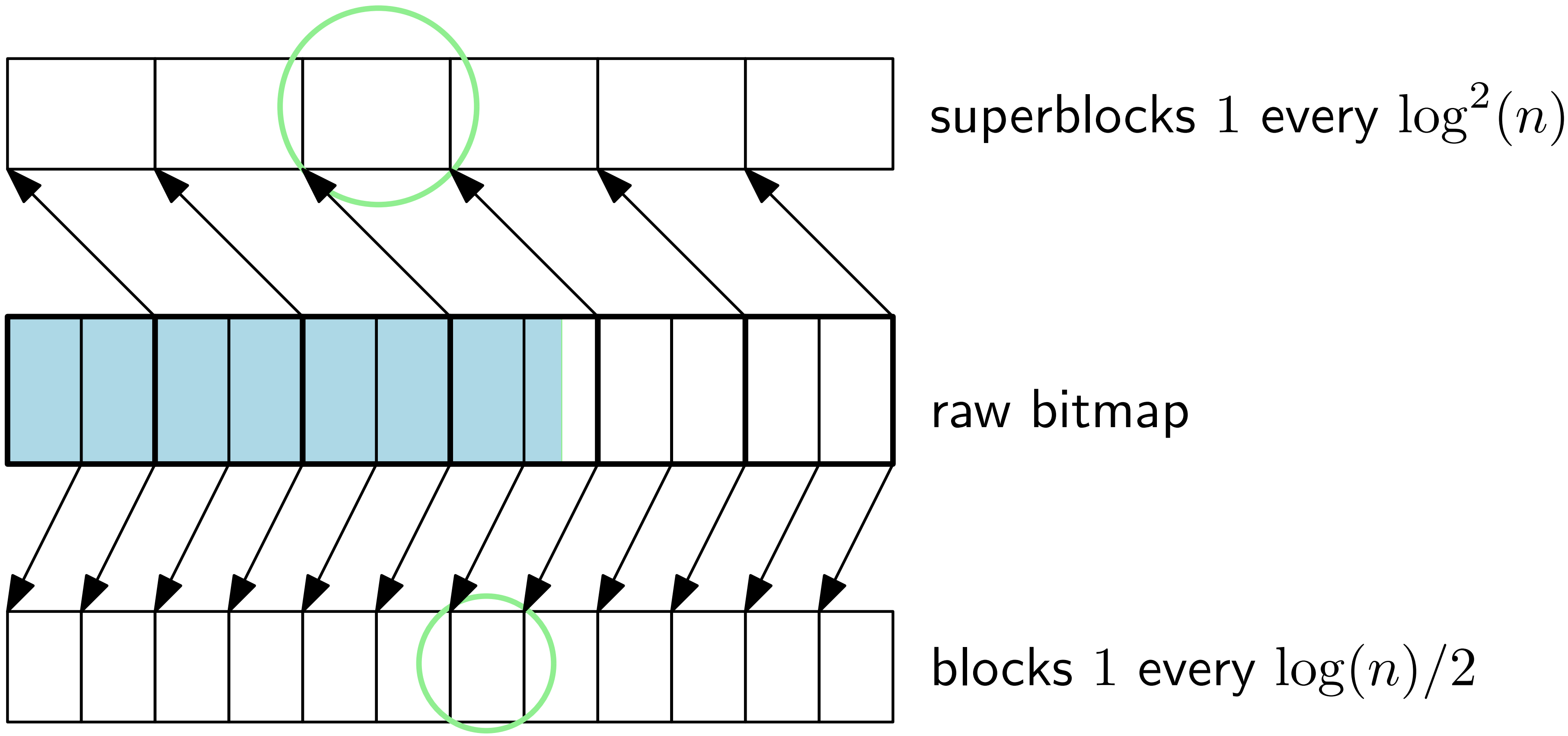
Rank



Rank



Rank



overall space: $n + n/\log n + \frac{2n}{\log n} \log \log n$ 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

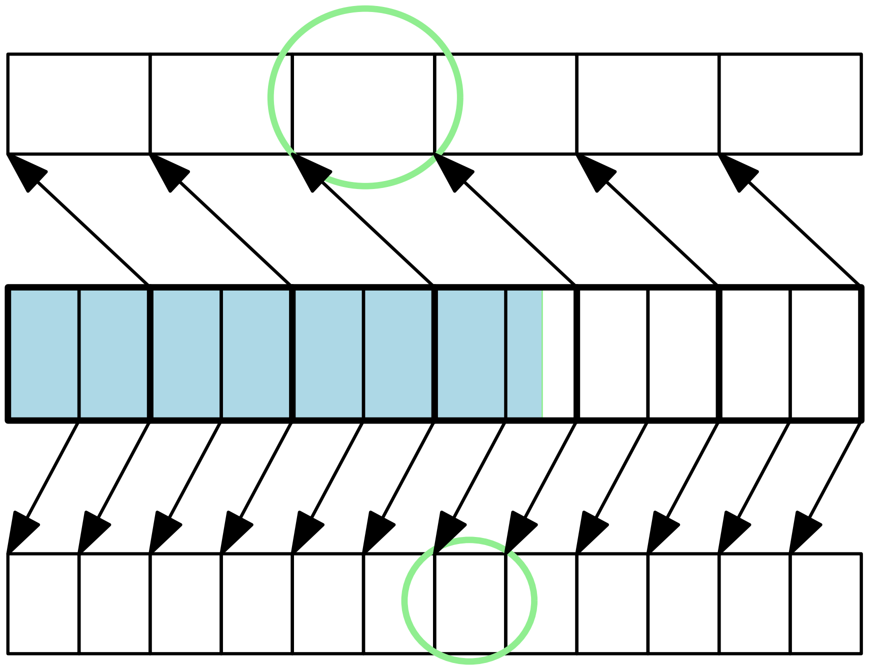
Rank

Superblocks: $O(n/\log n)$ bits

Blocks: $O(n \log\log n/\log n)$ bits

Table: $O(\sqrt{n}\log n \log\log n)$ bits

3 steps answer rank



Select Operation

Partition according to the number of 1s

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

$\log^2 n$ 1s

Select Operation

Partition according to the number of 1s

$$\mathsf{B} = \boxed{10100101}\boxed{001001011}\underbrace{\boxed{0010101001}}_{\log^2 n \text{ 1s}}$$

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

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

$\log^2 n$ 1s

Store the positions where each dense superblock 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.

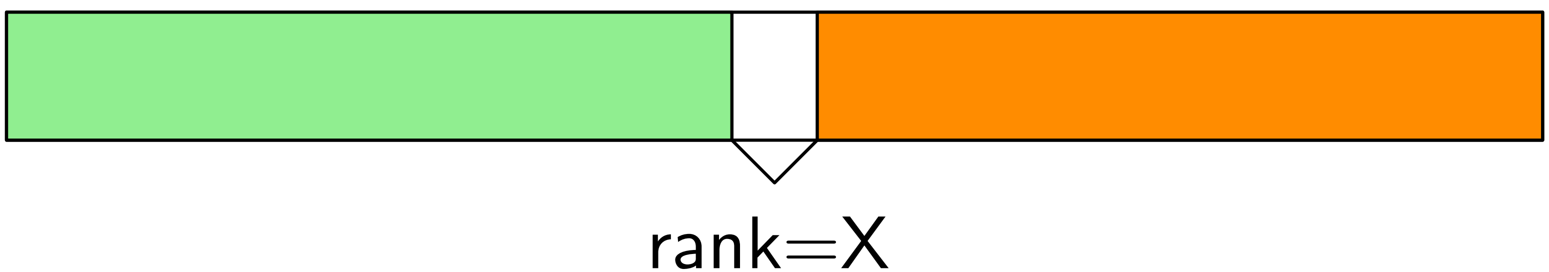
Select Operation

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

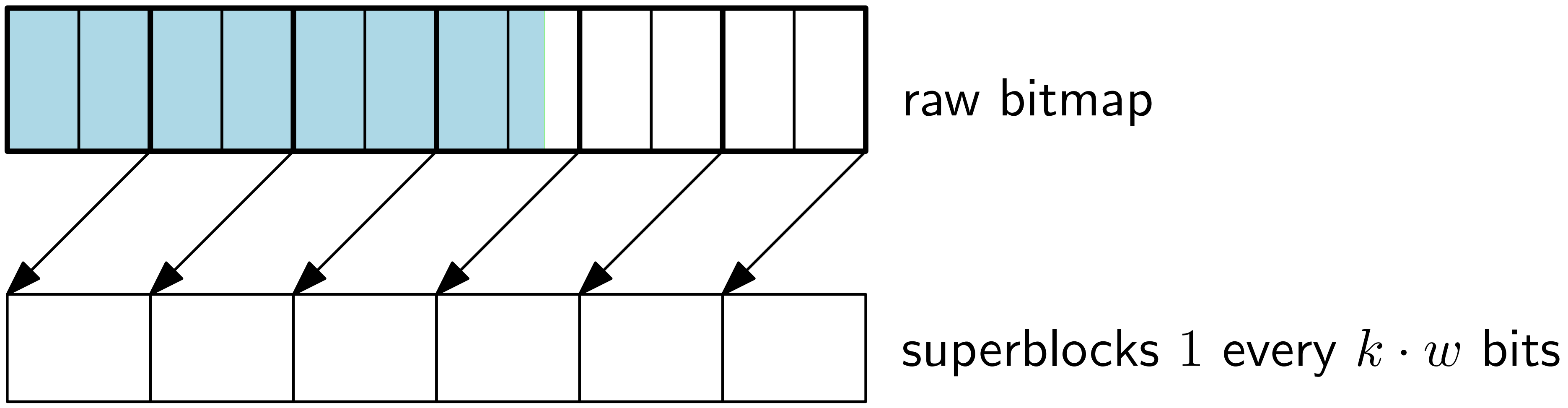
Select Operation

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

$$B = \boxed{101}\boxed{001}\boxed{010}\boxed{010}\boxed{010}\boxed{110}\boxed{010}\boxed{101}\boxed{000}$$

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

$$B = \boxed{101}\boxed{001}\boxed{010}\boxed{010}\boxed{010}\boxed{110}\boxed{010}\boxed{101}\boxed{000}$$

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

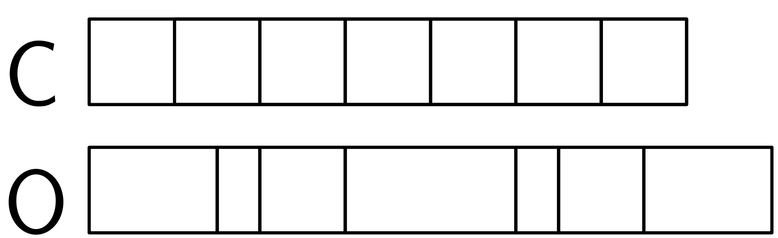
Blocks of size $b = \log(n)/2$

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

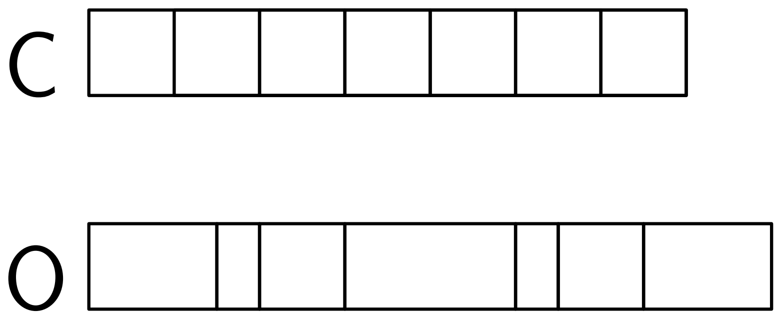
Total space: $\sum_{i=0}^{n/b} \lceil \log \binom{b}{c_i} \rceil \leq nH_0(B) + O(n/\log n)$ bits



Compressed Bitmaps

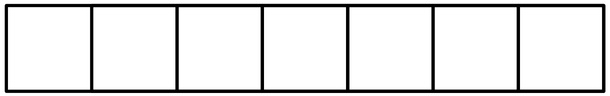
$$
\begin{aligned}
\sum_{i=1}^{n/b} \left\lceil \log \binom{b}{c_i} \right\rceil &\leq \sum_{i=1}^{n/b} \log \binom{b}{c_i} + n/b \\
&= \log \Pi_{i=1}^{n/b} \binom{b}{c_i} + O(n/\log n) \\
&\leq \log \binom{(n/b)b}{\sum_{i=1}^{n/b} c_i} + O(n/\log n) \\
&= \log \binom{n}{m} + O(n/\log n) \\
&\leq nH_0(B)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

Compressed Bitmaps



How can we support constant time access?

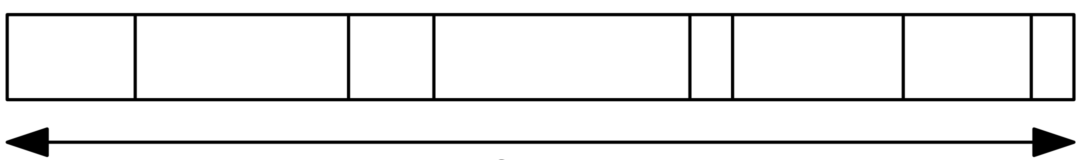
Compressed Bitmaps

C 

O 

How can we support constant time access?

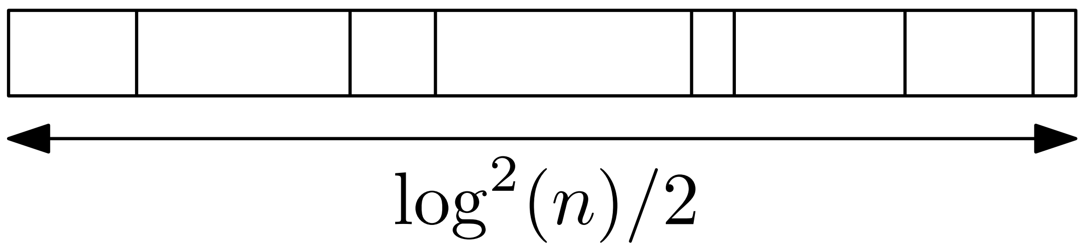
Store a pointer to O every $\log(n)$ blocks



$\log^2(n)/2$

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

Compressed Bitmaps

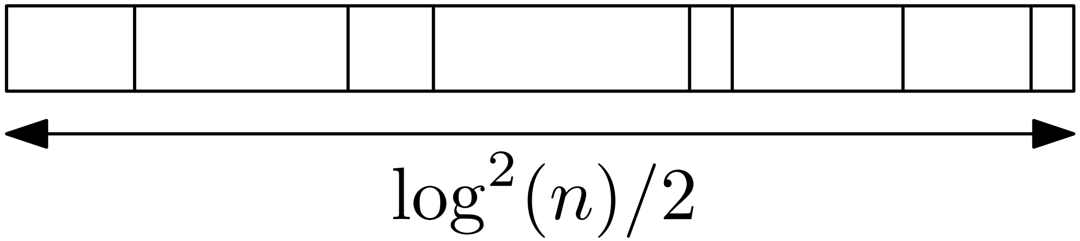


Store offset for each element

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

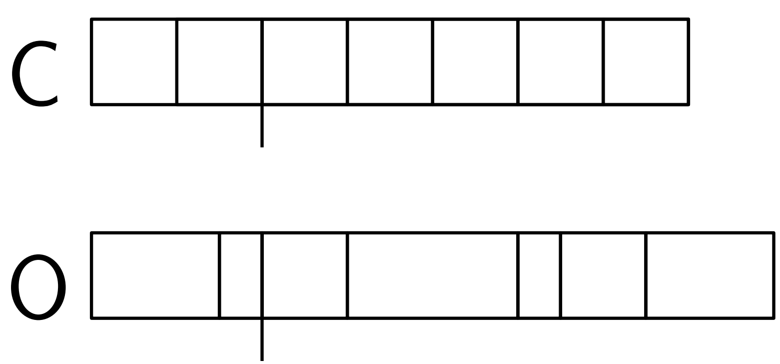
Practical Implementation

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

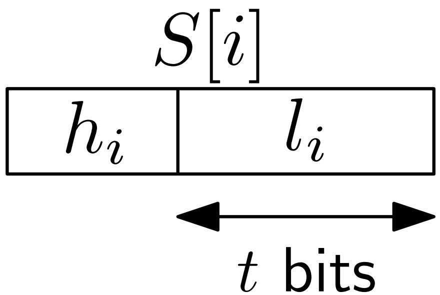
The previous solution does not work well for very sparse bitmaps

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

Very Sparse Bitmaps

The previous solution does not work well for very sparse bitmaps

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



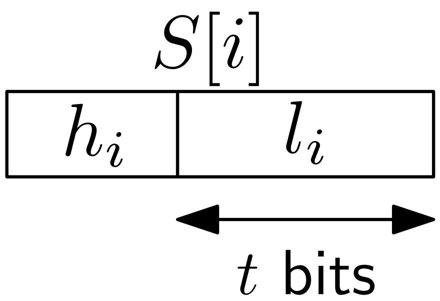
$t = \lceil \log n/m \rceil$

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



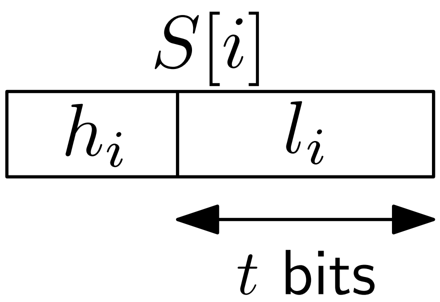
$$t = \lceil \log n/m \rceil$$

Two arrays: H and L

h_1	h_2	$\ldots$	h_m

Positions $h_i + i$ are ones

Very Sparse Bitmaps



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

Two arrays: H and L

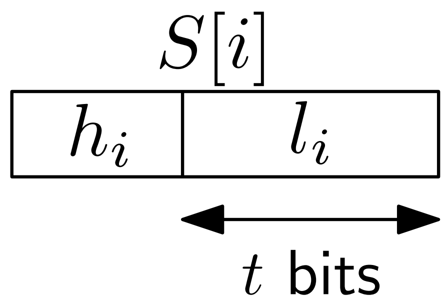
h_1	h_2	$\dots$	h_m

Positions $h_i + i$ are ones

$$h_i = select(H, i) - i$$

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

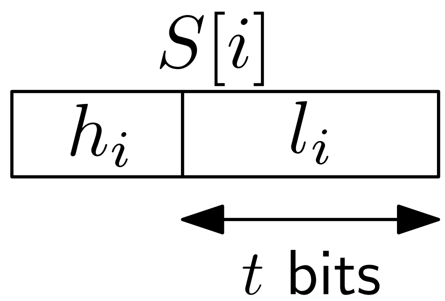
Very Sparse Bitmaps



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

Two arrays: H and L

Very Sparse Bitmaps

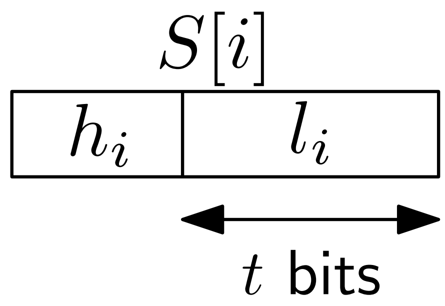


$$t = \lceil \log n/m \rceil$$

Two arrays: H and L

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

Very Sparse Bitmaps



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

Two arrays: H and L

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

rank:

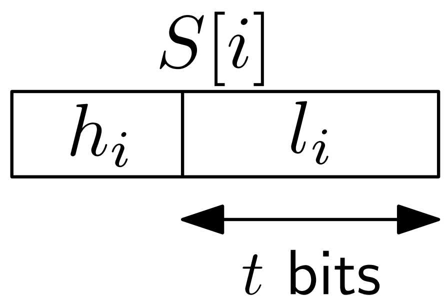
$i = h \cdot 2^t + l$

$x = 1 + rank(H, select_0(H, h))$

$y = rank(H, select_0(H, h + 1))$

binary search $L[x, y]$

Very Sparse Bitmaps



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

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

$$S = \mathsf{EHDHACEEGBCBGCF}$$

Sequences

$$S = \boxed{EHDHACE}EGBC\boxed{B}GCF$$

$access_S(12) = B$

$rank_S(H, 7) = 2$

$select_S(E, 3) = 8$

$n = |S|$

$\sigma = |\Sigma|$

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

Sequences

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

Wavelet Trees

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

$n + o(n)$ bits

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$n + o(n)$ bits

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

$n + o(n)$ bits

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

Wavelet Trees

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

$n + o(n)$ bits

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$n + o(n)$ bits

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

$n + o(n)$ bits

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

Wavelet Trees

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

$n + o(n)$ bits

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$n + o(n)$ bits

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

$n + o(n)$ bits

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

Wavelet Trees

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

$n + o(n)$ bits

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$n + o(n)$ bits

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

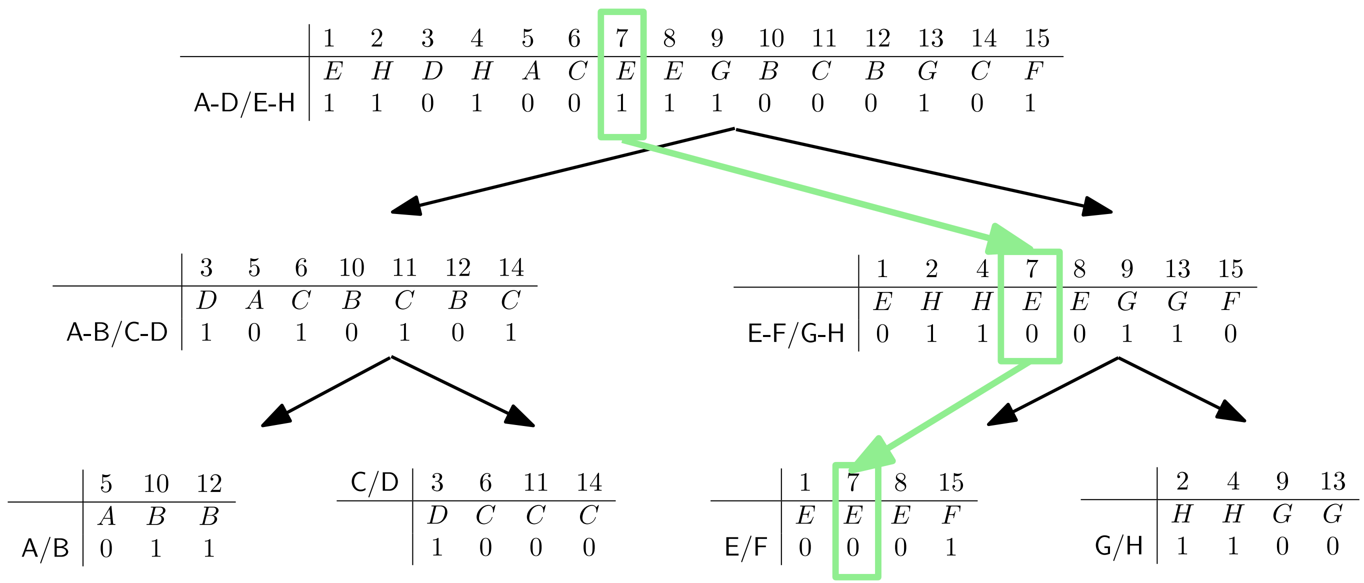
	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

$n + o(n)$ 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

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$rank(E, 9)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$rank(E, 9)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$rank(E, 9)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$rank(E, 9)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$rank(E, 9)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$select(B, 2)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$select(B, 2)$$

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

Wavelet Trees

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

$$select(B, 2)$$

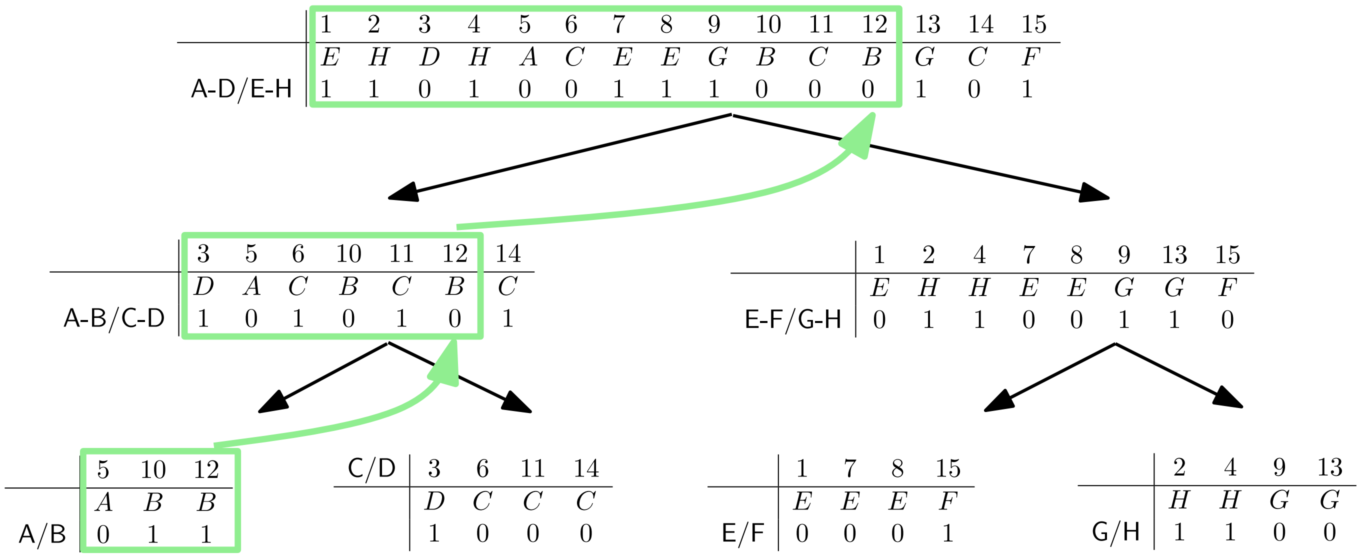
	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

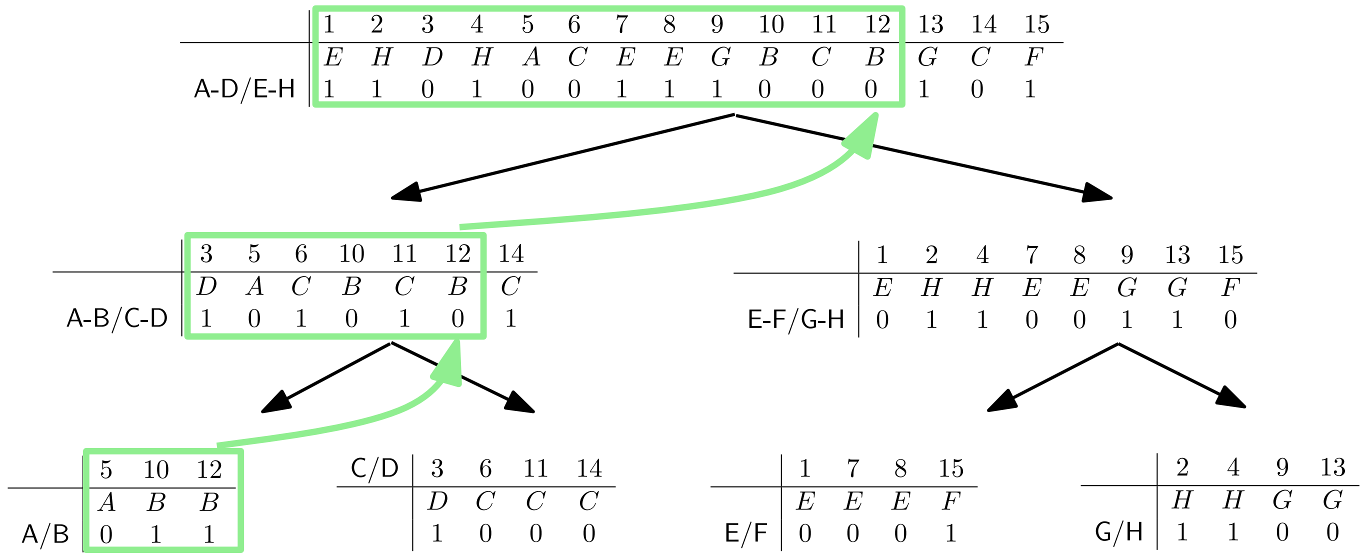
	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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
	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
A-D/E-H	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

	3	5	6	10	11	12	14
	D	A	C	B	C	B	C
A-B/C-D	1	0	1	0	1	0	1

	1	2	4	7	8	9	13	15
	E	H	H	E	E	G	G	F
E-F/G-H	0	1	1	0	0	1	1	0

	5	10	12
	A	B	B
A/B	0	1	1

C/D	3	6	11	14
	D	C	C	C
	1	0	0	0

	1	7	8	15
	E	E	E	F
E/F	0	0	0	1

	2	4	9	13
	H	H	G	G
G/H	1	1	0	0

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

With a higher fan-out we can get $O(\log \sigma / \log \log n)$ time per query.

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	WaveletTreeNoptrs
Sequence	Sequence
Length	Length
Coder	Bitmap Builder
Bitmap Builder	Mapper
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

$$\mathsf{S} = \mathsf{abracadabraa}$$

$$\begin{array}{c|c|c|}
 & \overleftrightarrow{\quad b \quad} & \\
\hline
\mathsf{a} & 100101 & 010011 \\
\mathsf{b} & 010000 & 001000 \\
\mathsf{c} & 000010 & 000000 \\
\mathsf{d} & 000000 & 100000 \\
\mathsf{r} & 001000 & 000100 \\
\hline
\end{array}$$

$$\mathsf{B} = 10001000\ 1010\ 101\ 110\ 1010$$

GMR

S = abracadabraa

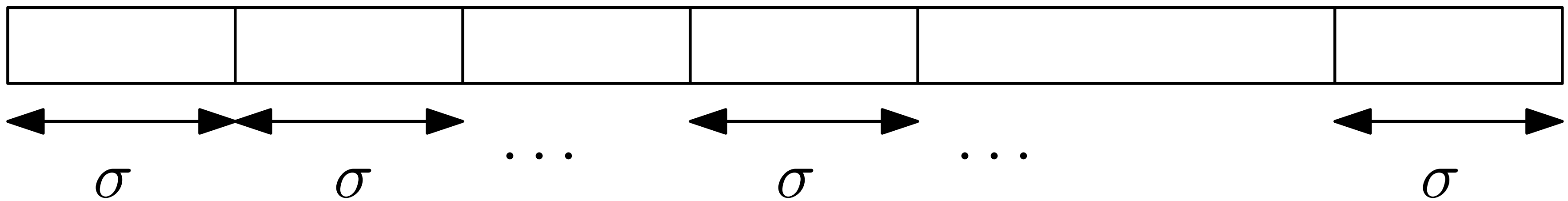
	b	
a	100101	010011
b	010000	001000
c	000010	000000
d	000000	100000
r	001000	000100

B = 10001000 1010 101 110 1010

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

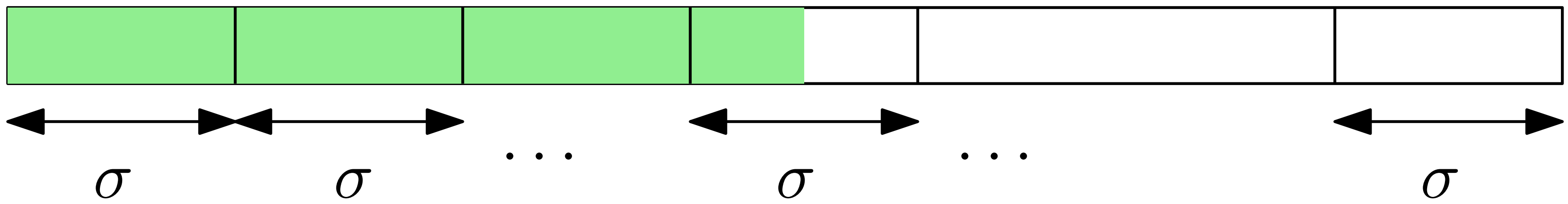
GMR

We can solve rank and select for multiples of σ



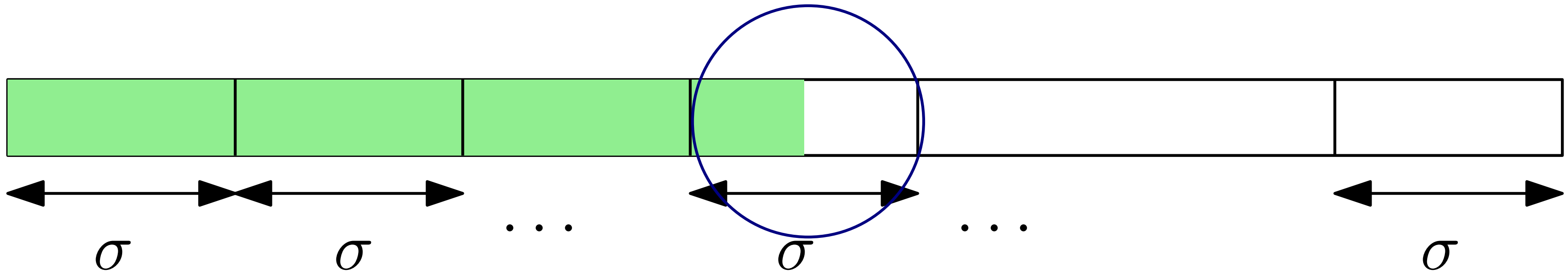
GMR

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=abcaaccbbcaa

$$X = 100000100010000$$

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

GMR

S=abcaaccbbcaa

$access(3)$

$$X = 100000100010000$$

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

GMR

S=abcaaccbbcaa

$access(3)$

$X = 100000100010000$

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

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

GMR

S=abcaaccbbcaa

$access(3)$

$X = 1000001000010000$

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

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

GMR

S=abcaaccbbcaa

$$access(3)$$

$$X = 10000010001 0000$$

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

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

GMR

S=abcaaccbbcaa

$select(b, 2)$

$X = 100000100010000$

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

GMR

S=abcaaccbbcaa

$select(b, 2)$

$$X = 1000001000010000$$

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

GMR

S=abcaaccbbcaa

$select(b, 2)$

$X = 100000100010000$

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

GMR

S=abcaaccbbcaa

$rank(c, 8)$

$X = 100000100010000$

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

GMR

S=abcaaccbbcaa

$rank(c, 8)$

$$X = 10000010001\boxed{0000}$$

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

GMR

S=abcaaccbbcaa

$rank(c, 8)$

$$X = 10000010001\boxed{0000}$$

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

GMR

S=abcaaccbbcaa

$$rank(c, 8)$$

$$X = 10000010001\boxed{0000}$$

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

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

GMR

S=abcaaccbbcaa

$rank(c, 8)$

$X = 1000001000\boxed{0000}$

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

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

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

$$\longrightarrow$$

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

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

→

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

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

C:

O:

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

Class 1

Class 2

Class 3

Class 4

C:

O:

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

C: 2

O: 0

Alphabet Partitioning

$$S = EHDHACEEGBCBGCF$$

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

C: 2

O: 0

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

C: 2 3

O: 0 1

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

Class 1

Class 2

Class 3

Class 4

C: 2 3 . . .

O: 0 1 . . .

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

C: 2 3 ...

O: 0 1 ...

Represent C using a Wavelet Tree

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

C: 2 3 ...

O: 0 1 ...

Represent C using a Wavelet Tree

How many classes do we have?

Alphabet Partitioning

$$\mathsf{S} = EHDHACEEGBCBGCF$$

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

O_1

O_2: 0, 0, 0, 1, 1

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

O_4: 0

$$\left.\begin{array}{l} O_1 \\ O_2 \\ O_3 \\ O_4 \end{array}\right\} \approx nH_0(B)$$

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: $\boxed{2, 3, 3, 3, 3, 1, 2,}$ 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

access(7)

Alphabet Partitioning

S $= EHDHACEEGBCBGCF$

C: $\boxed{2, 3, 3, 3, 3, 1, 2,}$ 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

access(7)

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

rank(H,7)

Alphabet Partitioning

S $= \boxed{EHDHACE}EGBCBGCF$

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

rank(H,7)

Alphabet Partitioning

S $= \boxed{EHDHACE}EGBCBGCF$

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

rank(H,7)

Alphabet Partitioning

S $= \boxed{EHDHACE}EGBCBGCF$

C: $\boxed{2,\ 3,\ 3,\ 3,\ 3,\ 1,\ 2,}$ 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

rank(H,7)

Alphabet Partitioning

S $= \boxed{EHDHACE}EGBCBGCF$

C: $\boxed{2,\ 3,\ 3,\ 3,\ 3,\ 1,\ 2,}$ 2, 3, 2, 1, 2, 3, 1, 4

O_1

O_2: 0, 0, 0, 1, 1

O_3: $\boxed{1,\ 3,\ 1,\ 2,}$ 0, 0

O_4: 0

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

rank(H,7)

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

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

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

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

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

select(B,1)

Alphabet Partitioning

S $= EHDHACEEGBCBGCF$

C: 2, 3, 3, 3, 3, 1, 2, 2, 3, $\boxed{2}$, 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

...

```
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, 0u,
                           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
- Grid
- Text – FM-index [Ferragina and Manzini]

Permutations

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

Permutations

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

$\pi(i) = access(i)$

Permutations

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

$\pi(i) = access(i)$

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

Permutations

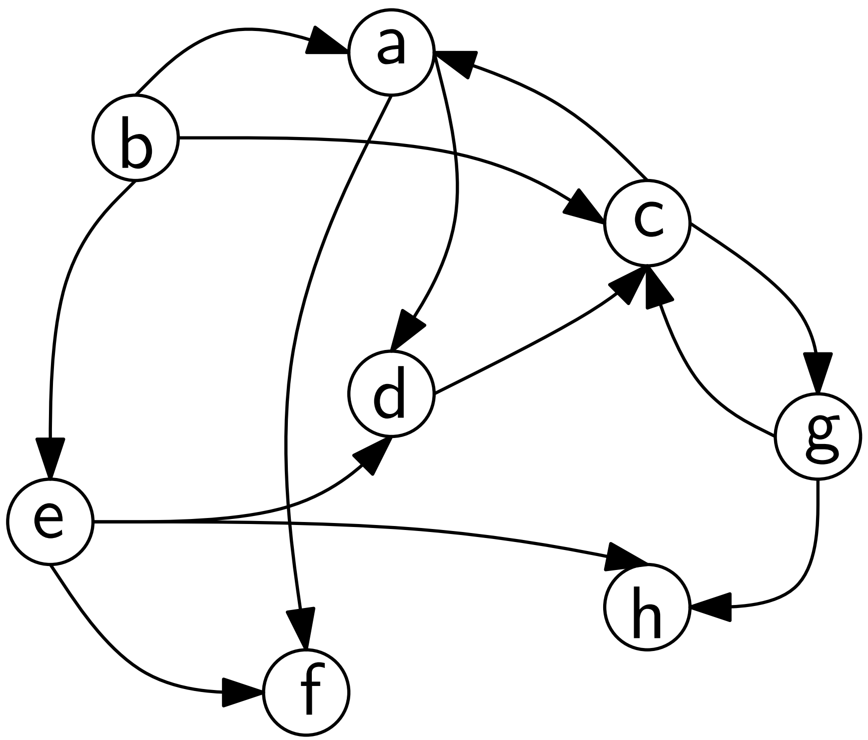
$$\pi = [1, 3, 7, 4, 2, 5, 6]$$

$\pi(i) = access(i)$

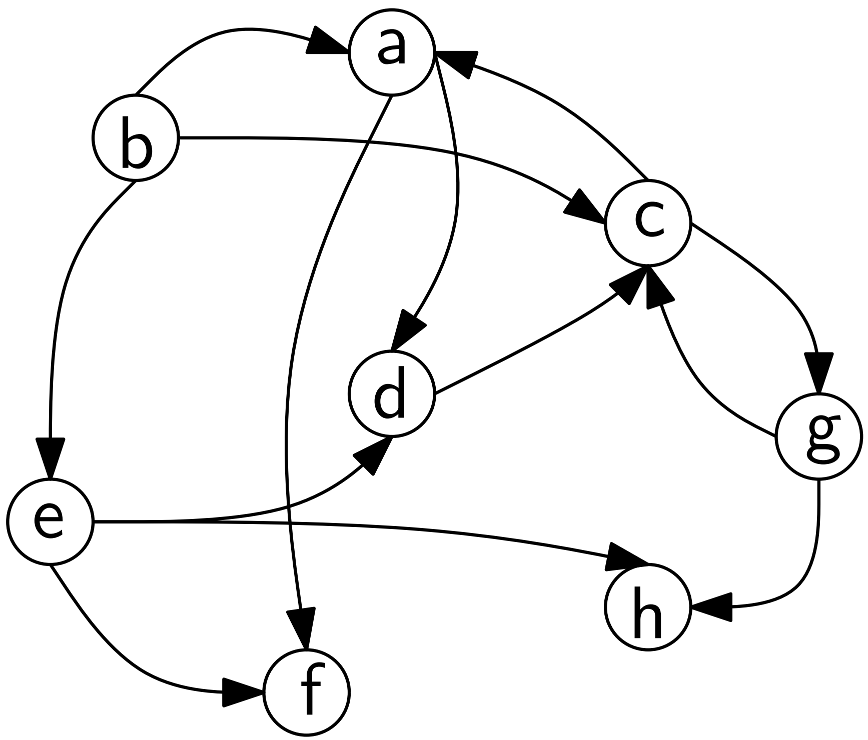
$\pi^{-1}(i) = select(i)$

MRRR achieves $(1, \log \log n)$
Wavelet Trees achieve $(\log n, \log n)$
GMR achieves $(\log \log n, 1)$

Graphs

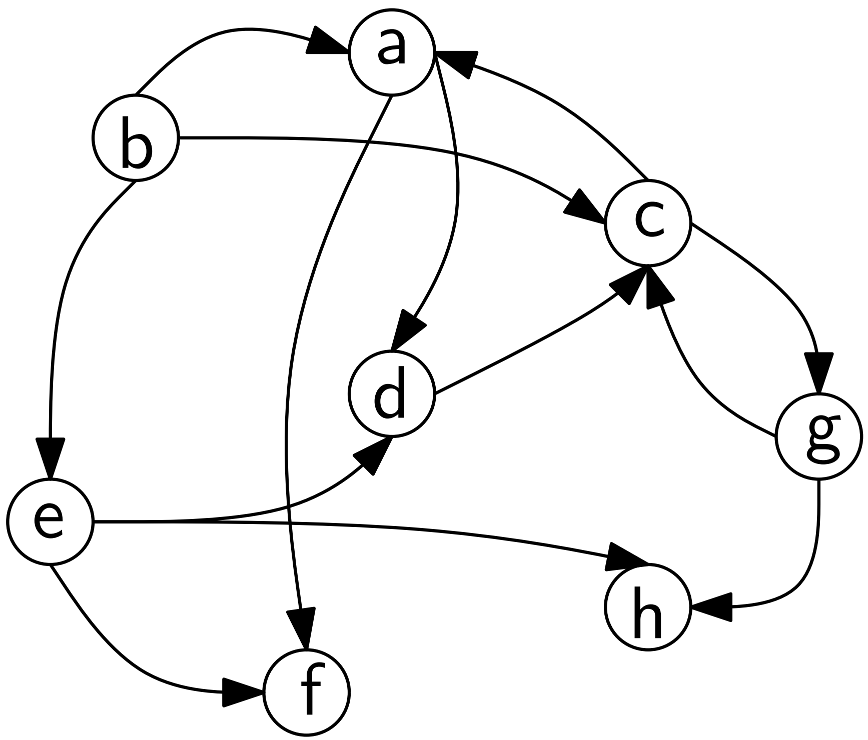


Graphs



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

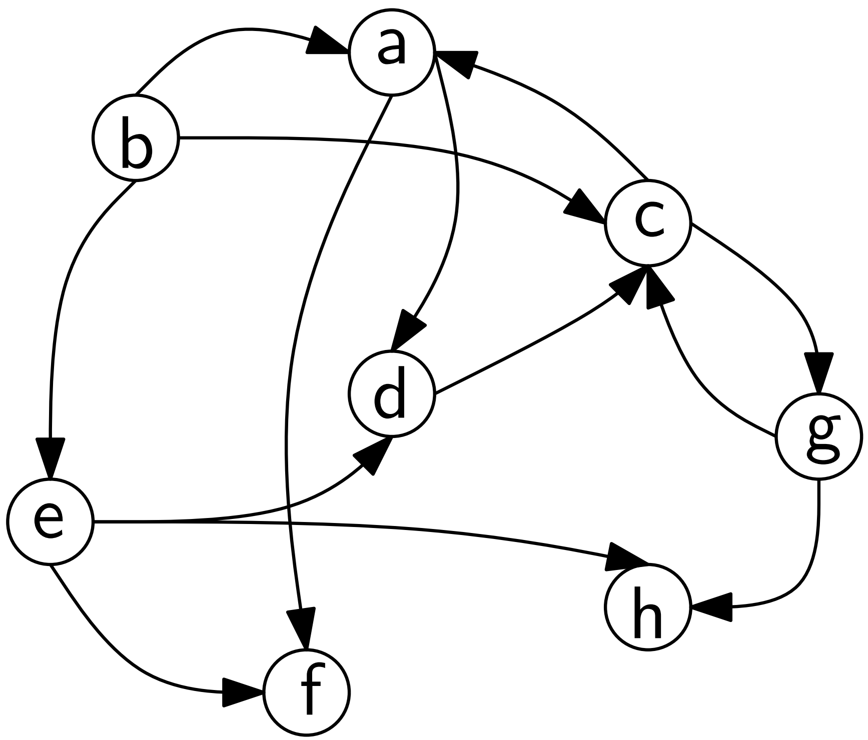
Graphs



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	

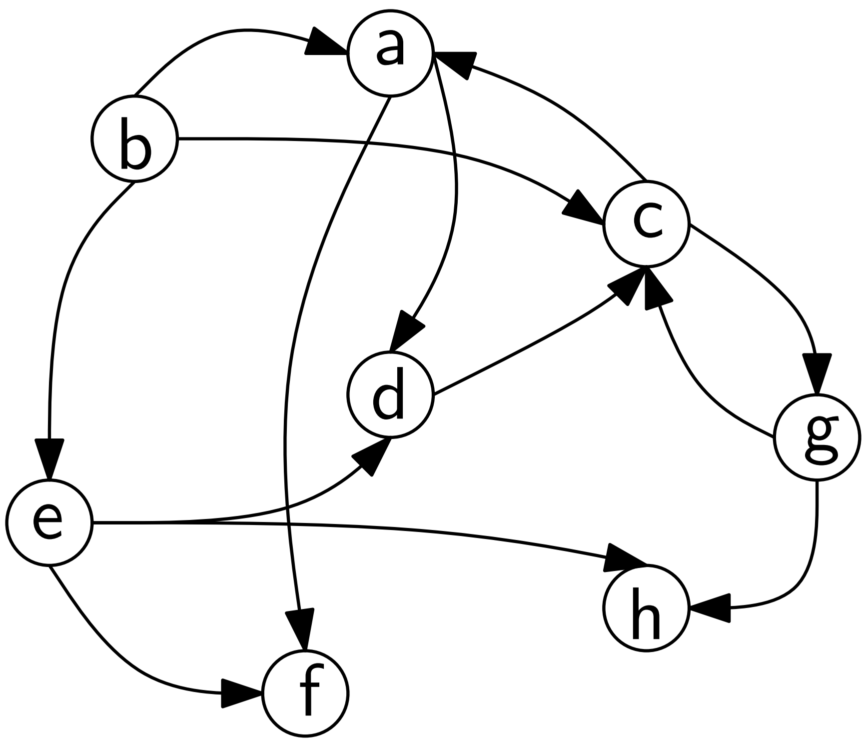
Graphs



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

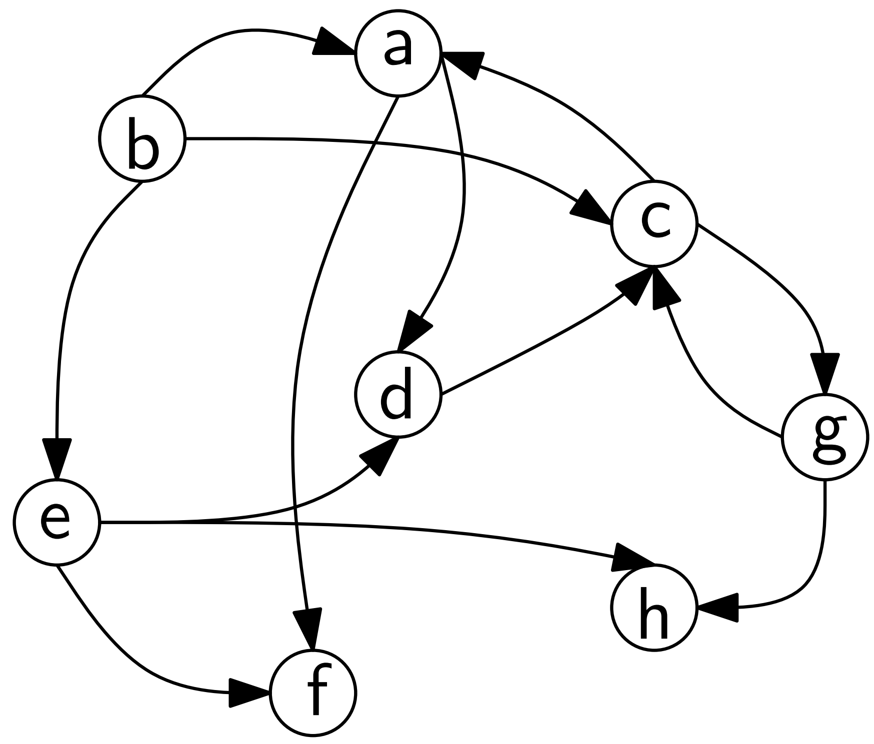
Graphs



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	

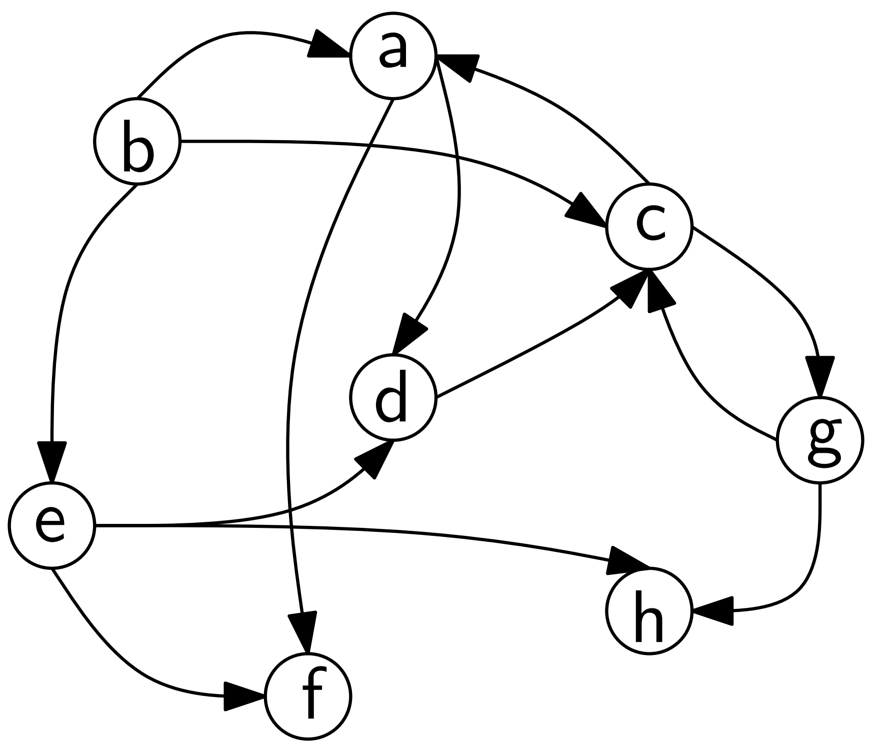
Graphs



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	

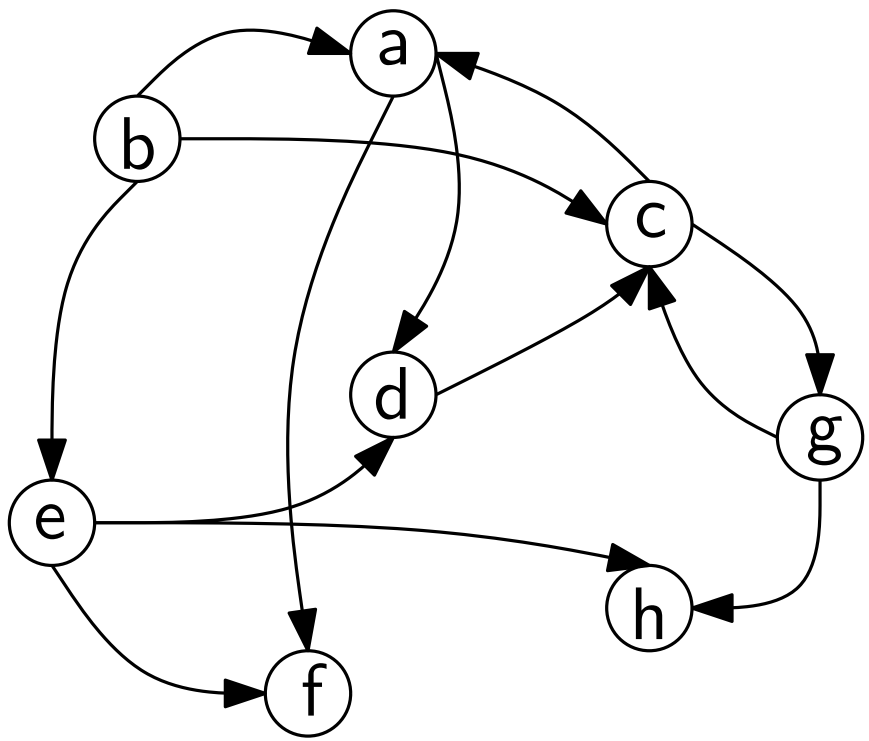
Graphs



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	

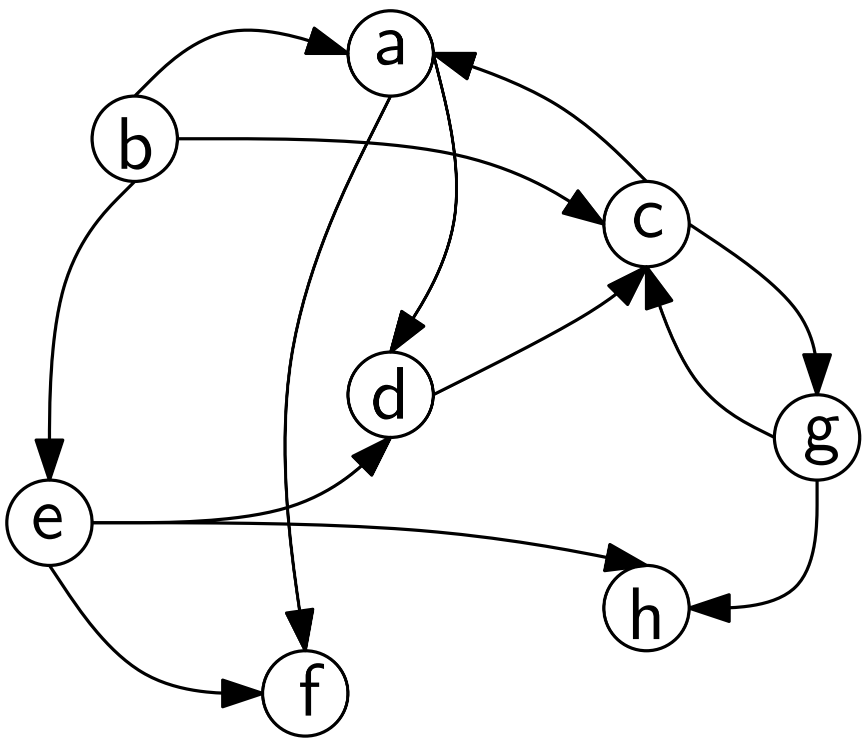
Graphs



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	

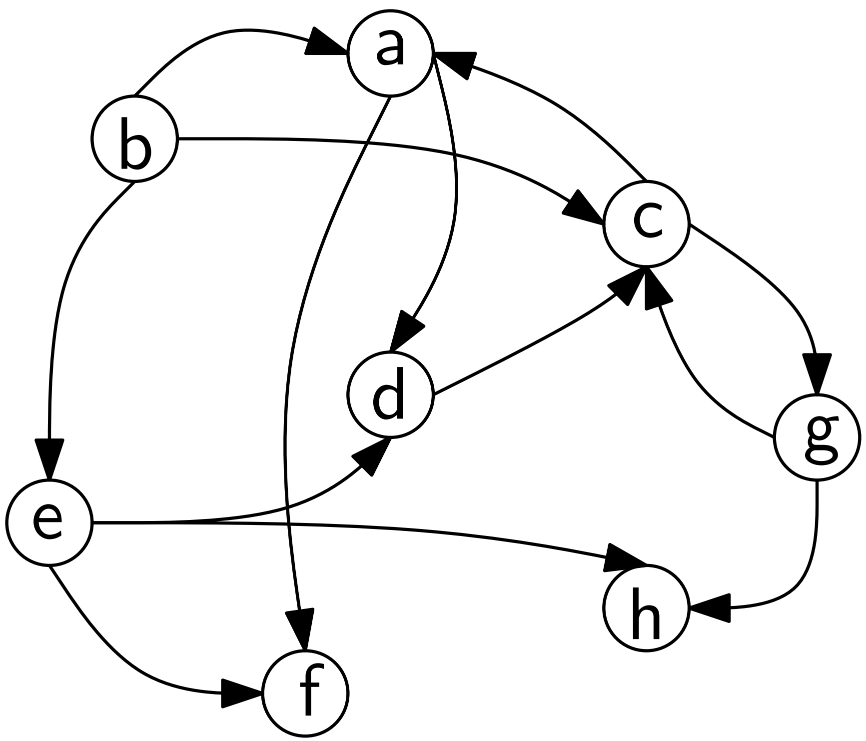
Graphs



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



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: $O(\log \log n)$

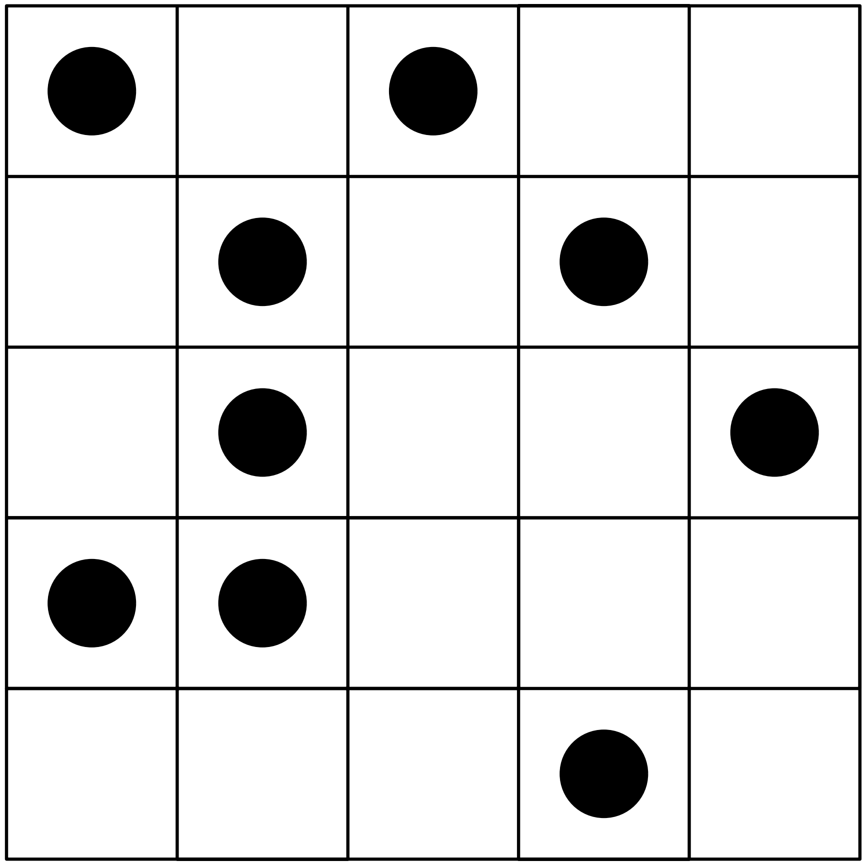
Adjacency list requires $n \log m + m \log n$

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

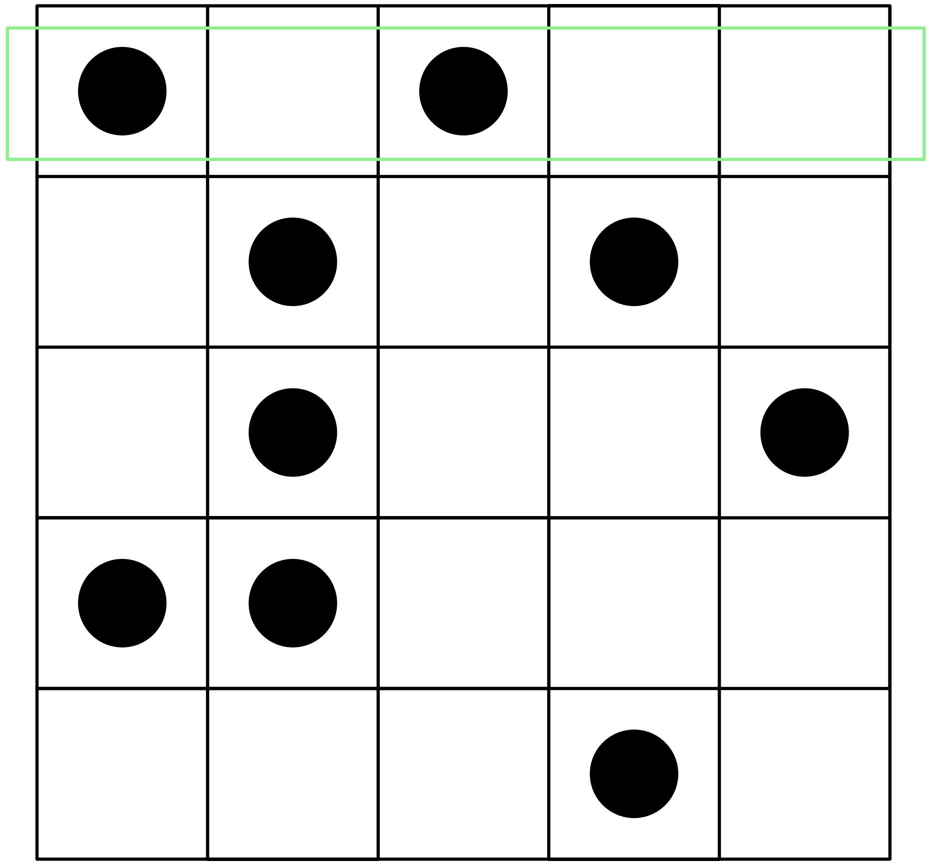
Adjacency matrix requires n^2

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

Grids

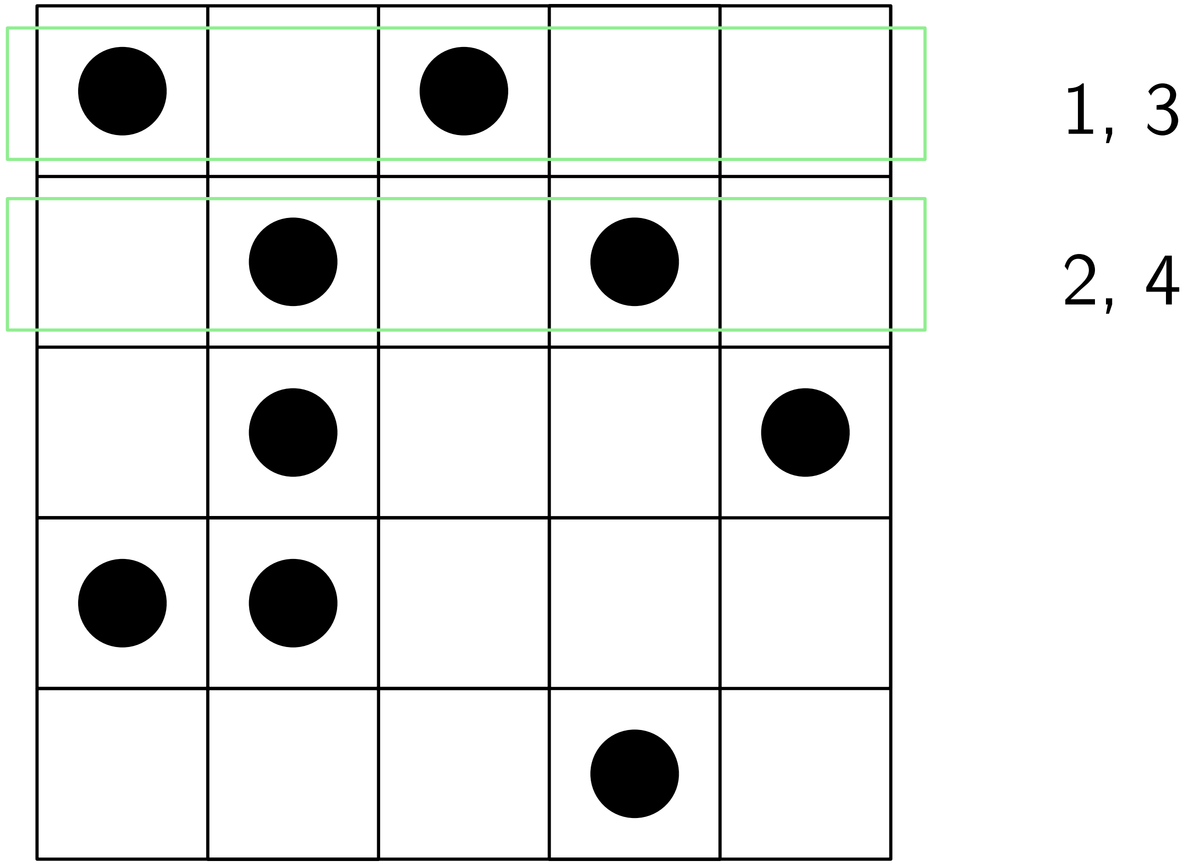


Grids



$1, 3$

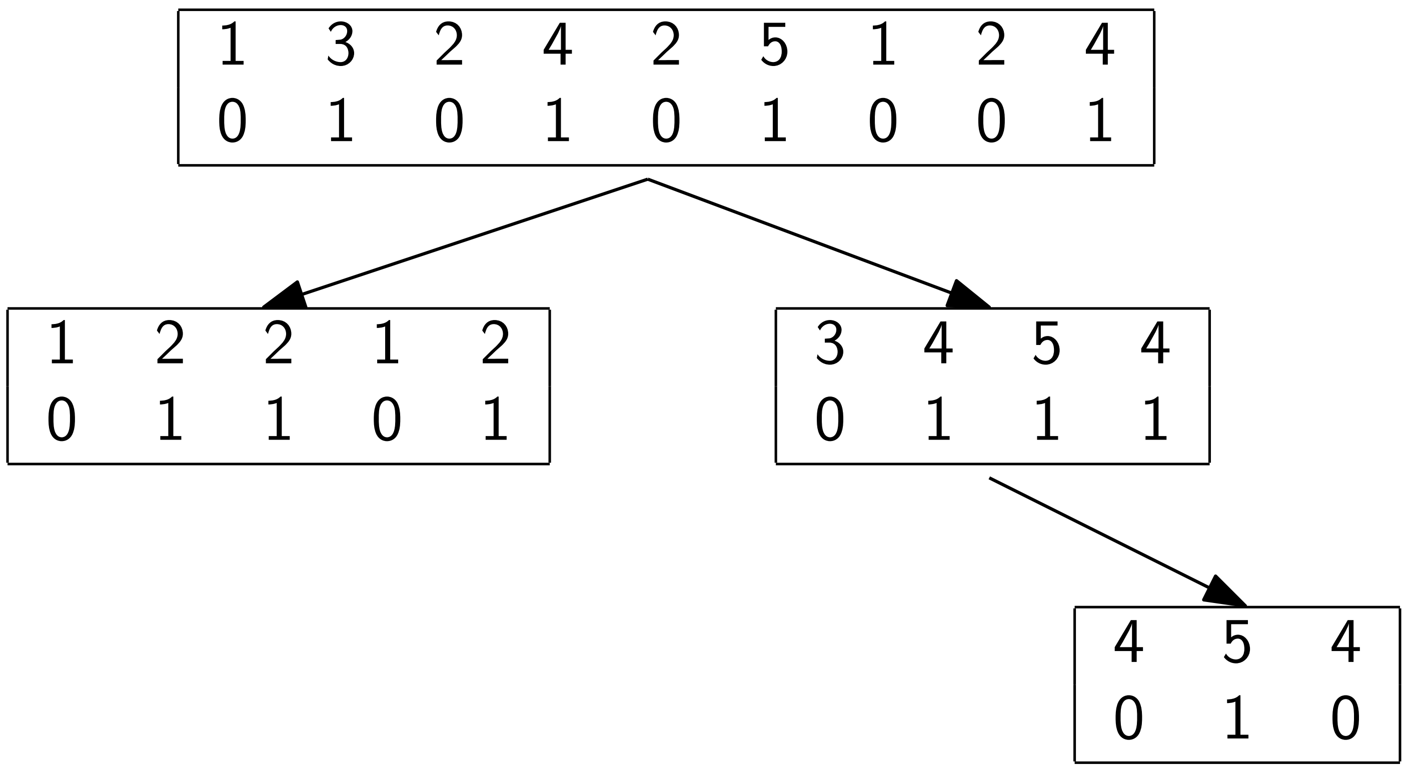
Grids



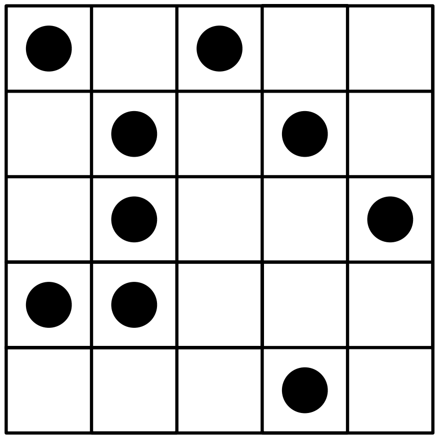
B	1	0	0	1	0	0	1	0	0	1	0	0	1	0
T		1	3		2	4		2	5		1	2		4

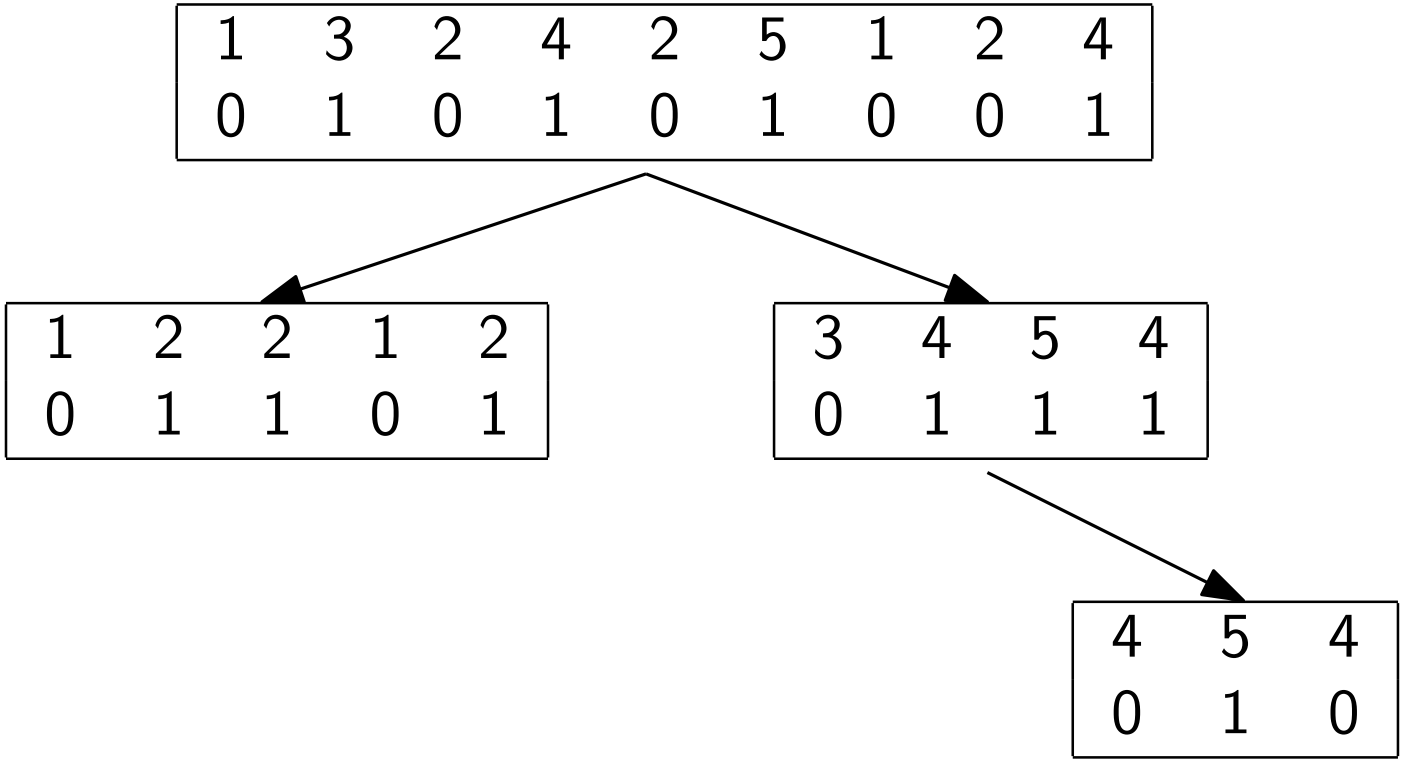
Grids

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

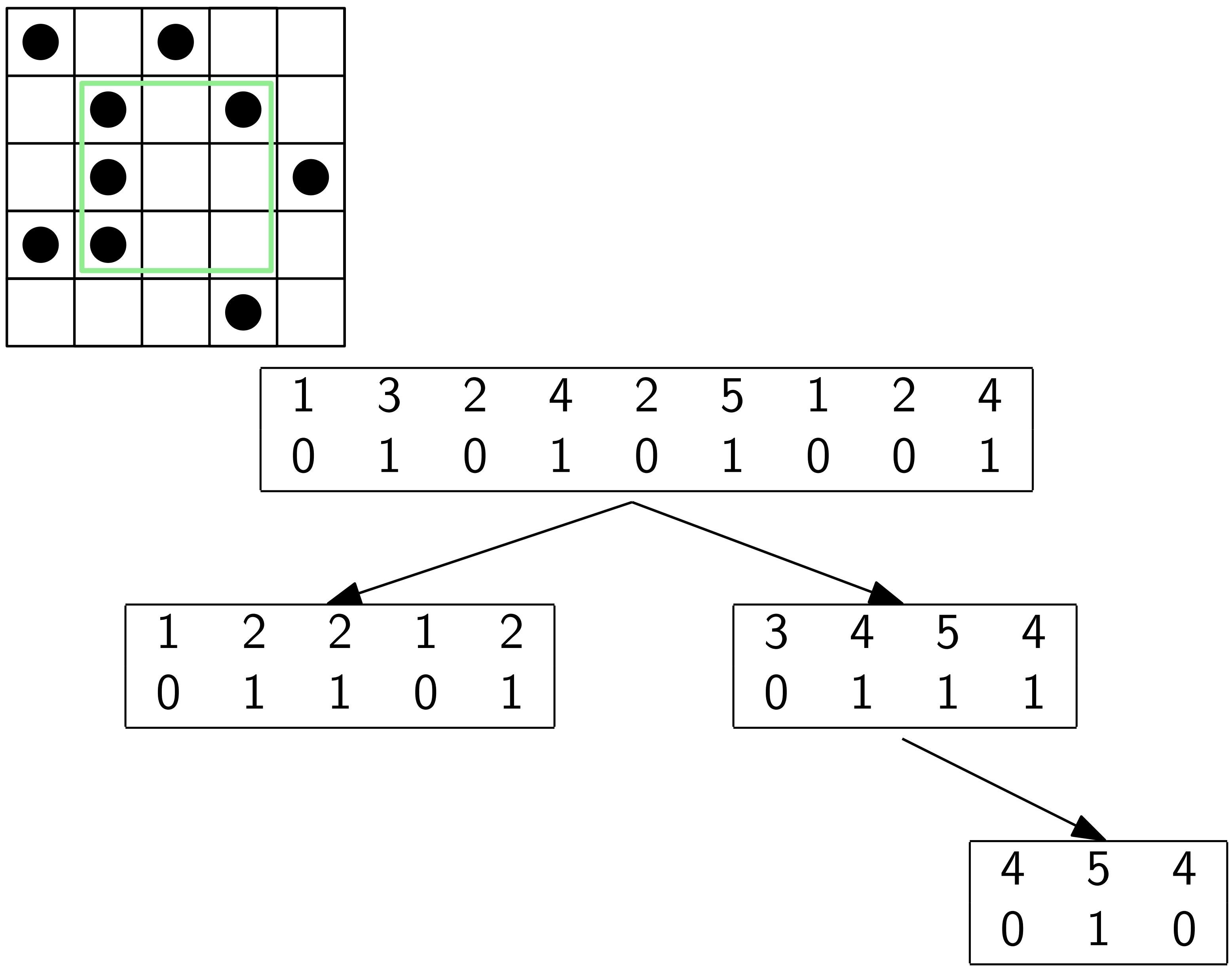


Grids

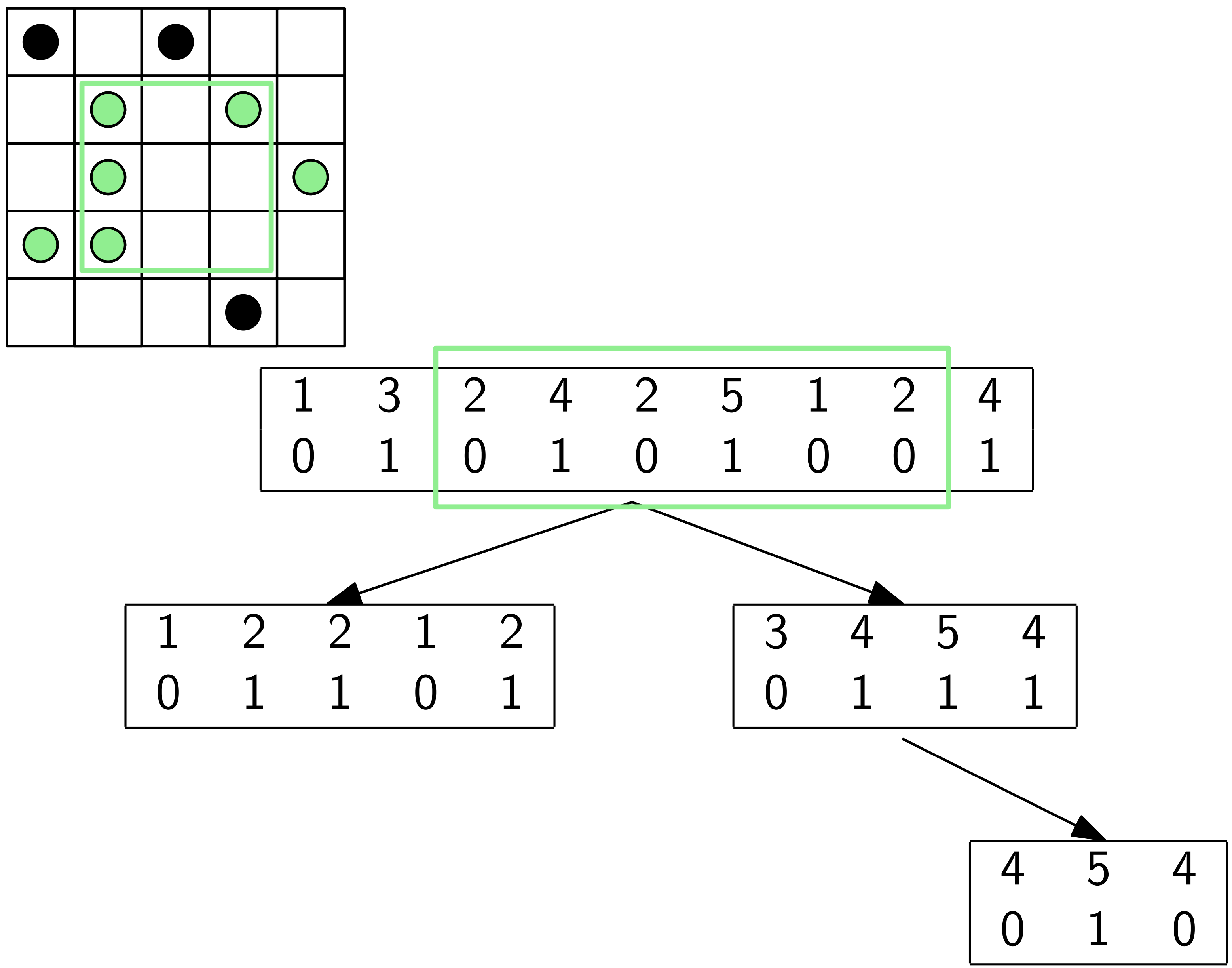




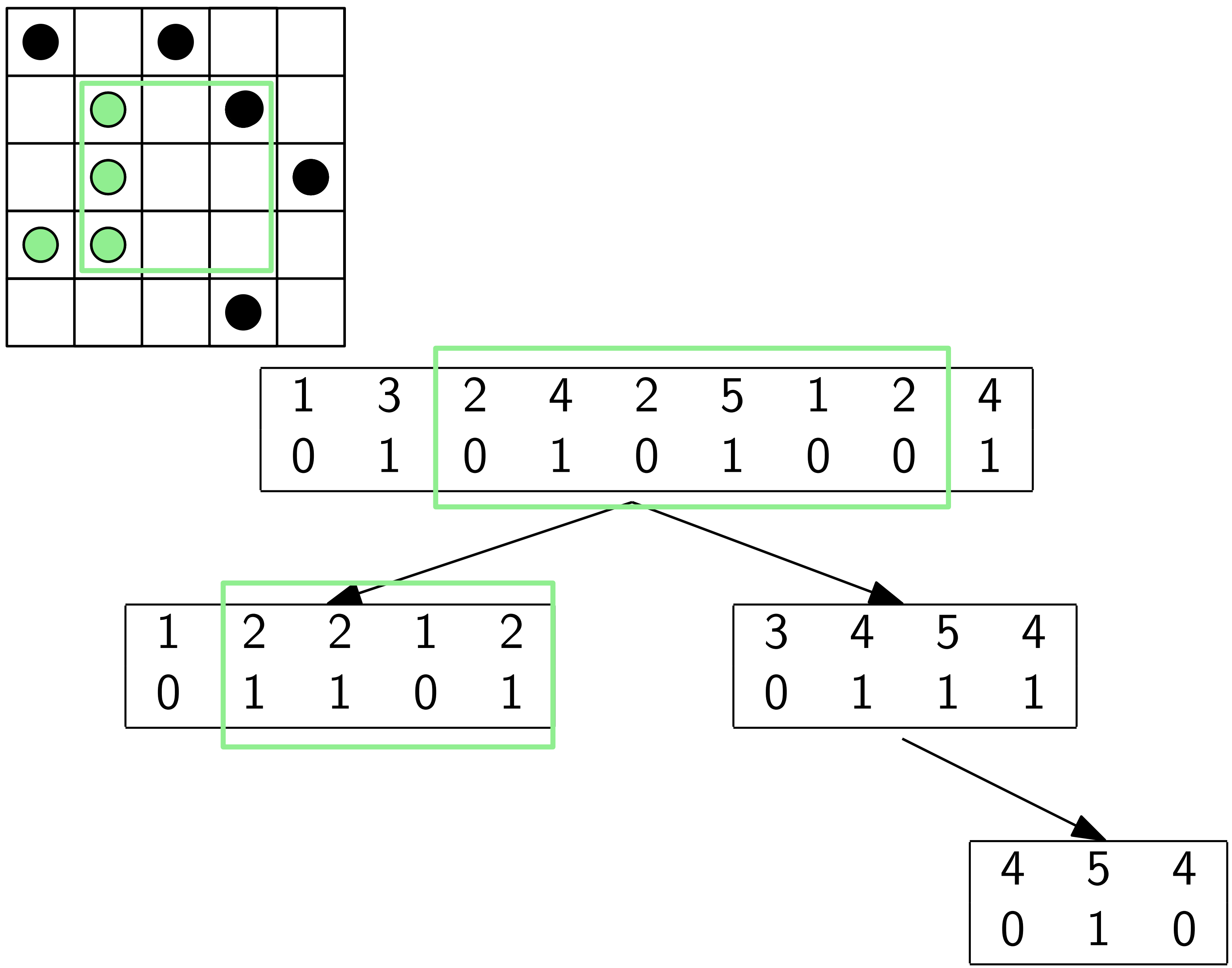
Grids



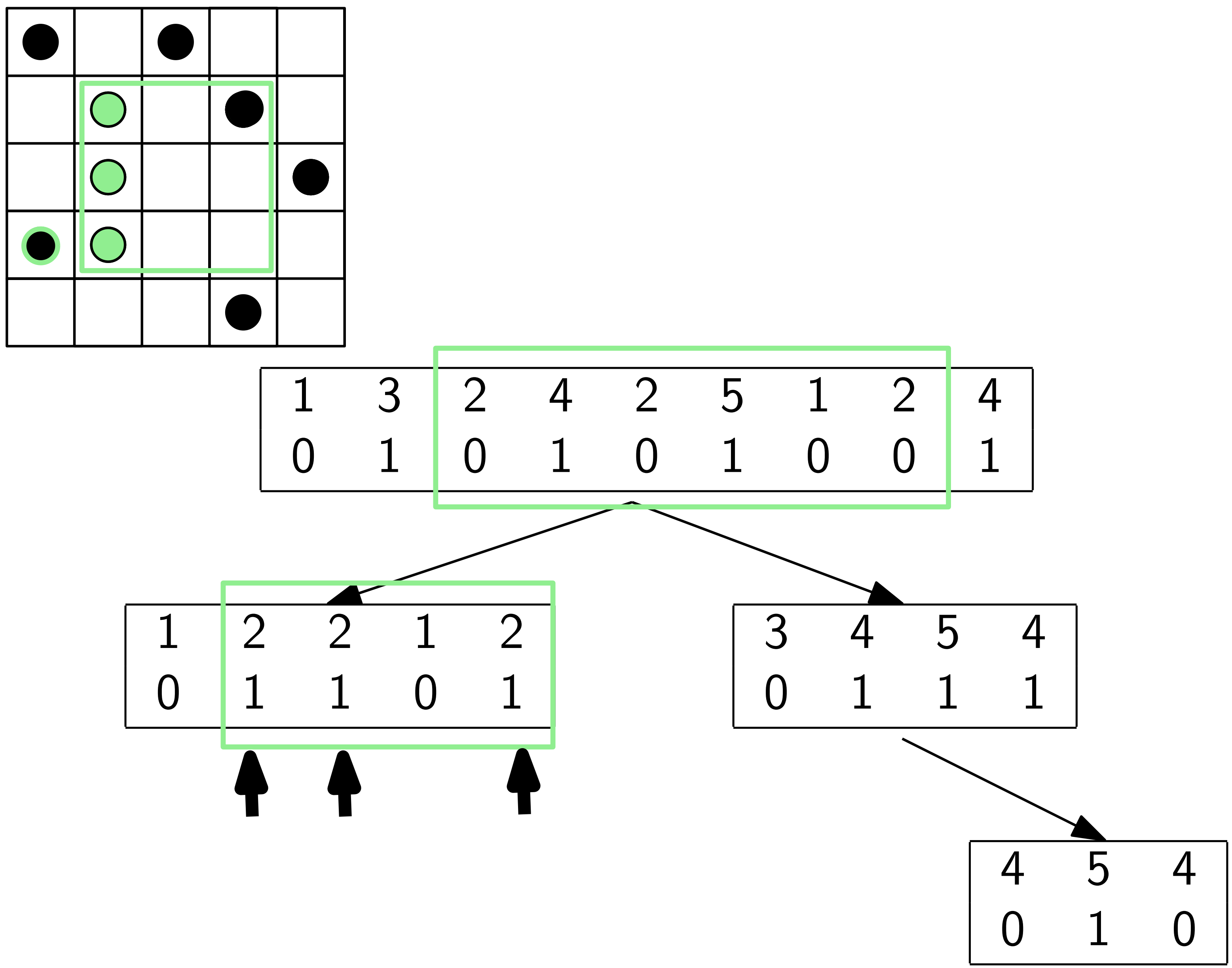
Grids



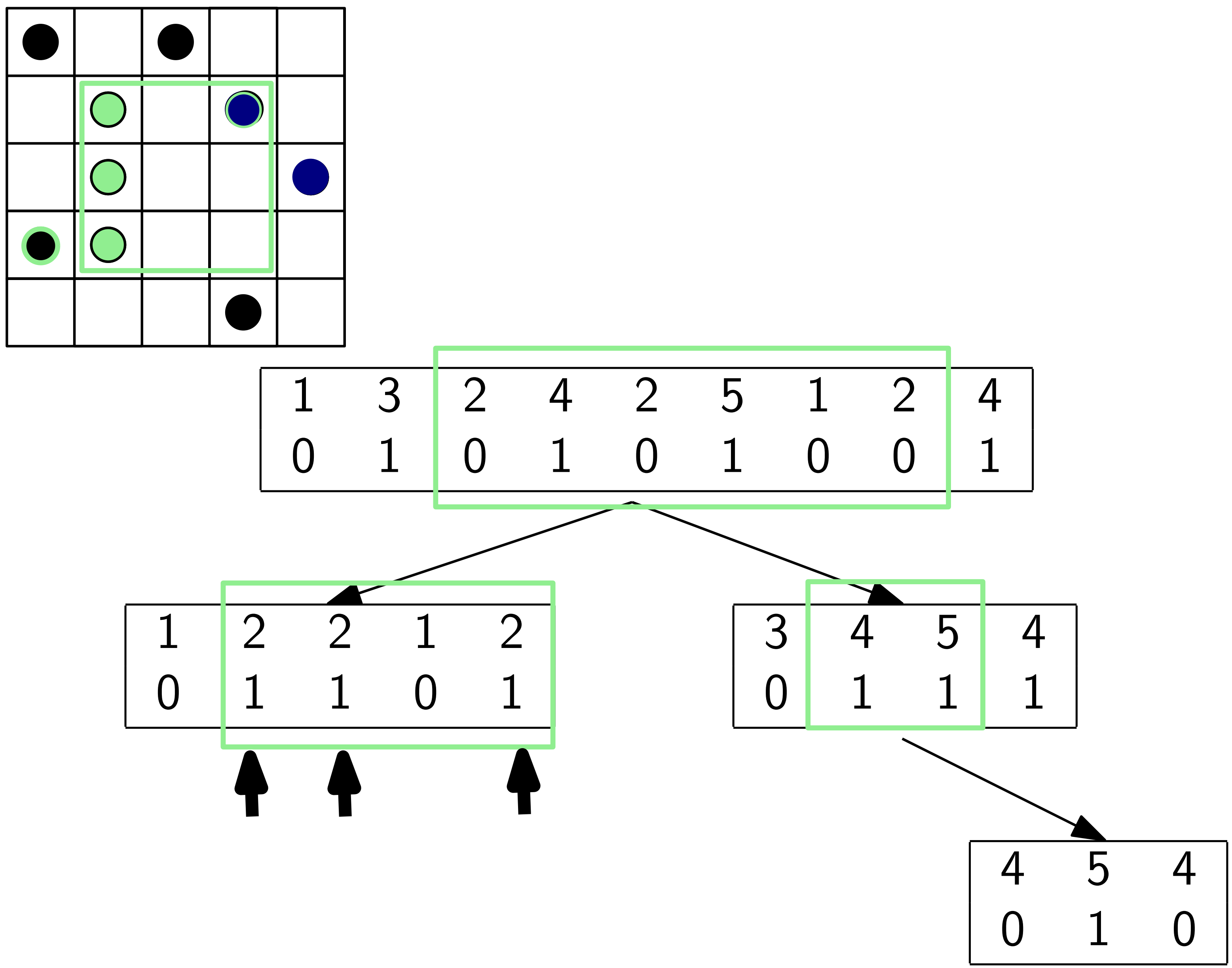
Grids



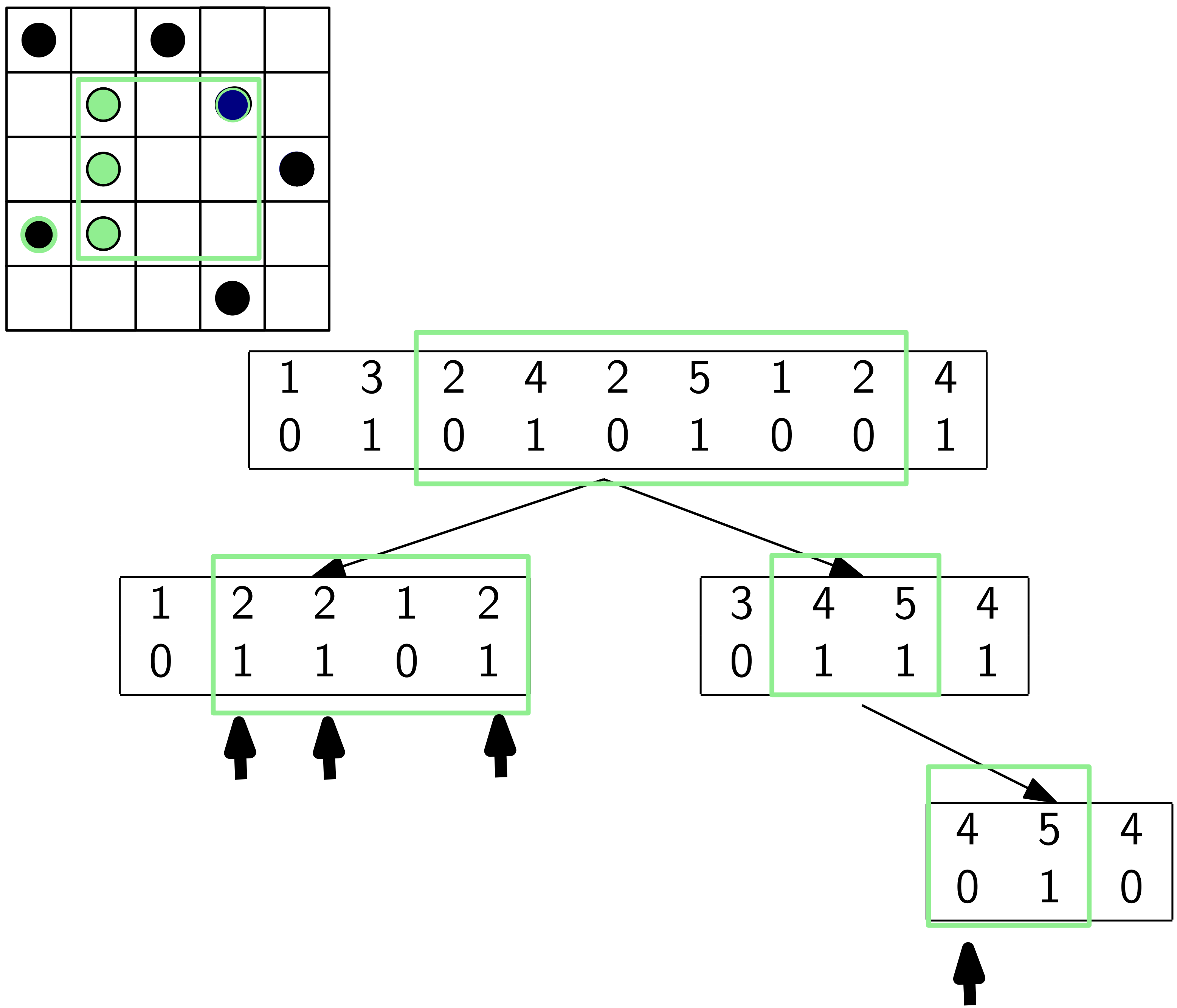
Grids



Grids



Grids

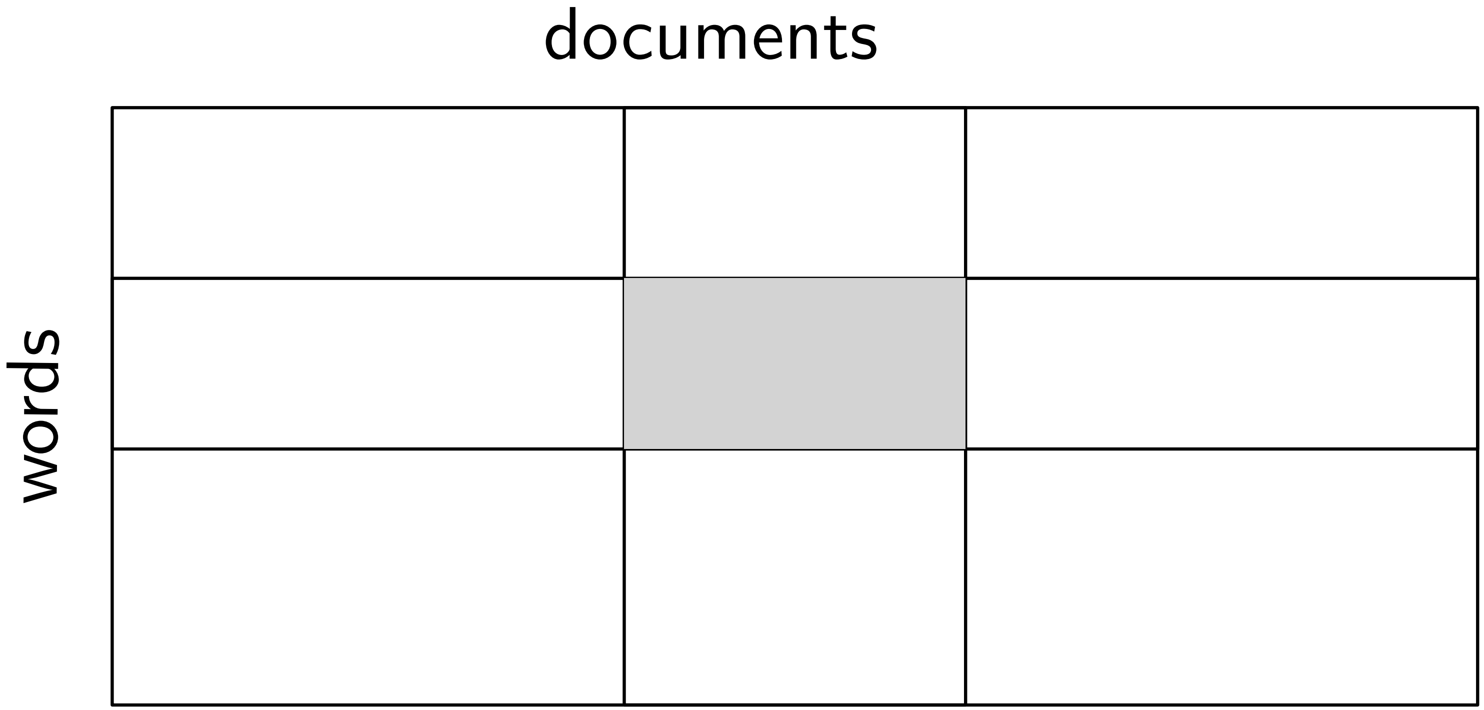


Grids

Binary Relations
Graphs
Inverted Indexes

Grids

Binary Relations
Graphs
Inverted Indexes



Text

FM-Index

$$\text{self-index} \begin{cases} \text{locate} \\ \text{count} \\ \text{extract} \end{cases}$$

compressed

Text

FM-Index

self-index $\begin{cases} \text{locate} \\ \text{count} \\ \text{extract} \end{cases}$

compressed

Burrows-Wheeler Transform

Text

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

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	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

$$LF(i) = rank(L[i], i) + C[L[i]]$$

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

$$LF(i) = rank(L[i], i) + C[L[i]]$$

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

Text Indexing

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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

F																L
$	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	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	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**
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**
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	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**
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**

Search for *lala*

Text Indexing

F																L
$	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	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	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**
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**
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	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**
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**

Search for *lala*

Text Indexing

F																L
$	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	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	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**
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**
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	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**
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**

Search for *lala*

Text Indexing

F																L
$	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	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	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**
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**
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	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**
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**

Search for *lala*

Text Indexing

F																L
$	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	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	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**
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**
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	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**
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**

Search for *lala* 1 occ

Text Indexing

Representing L with a wavelet tree

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

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

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

Text Indexing

Representing L with a wavelet tree

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

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

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

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

Other technical problems

The road ahead...

Improving what we already have

Construction of these structures

Other technical problems

LIBCDS2!

Links

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

Thanks for your attention

The End