Succinct Data Structures: From Theory to Practice

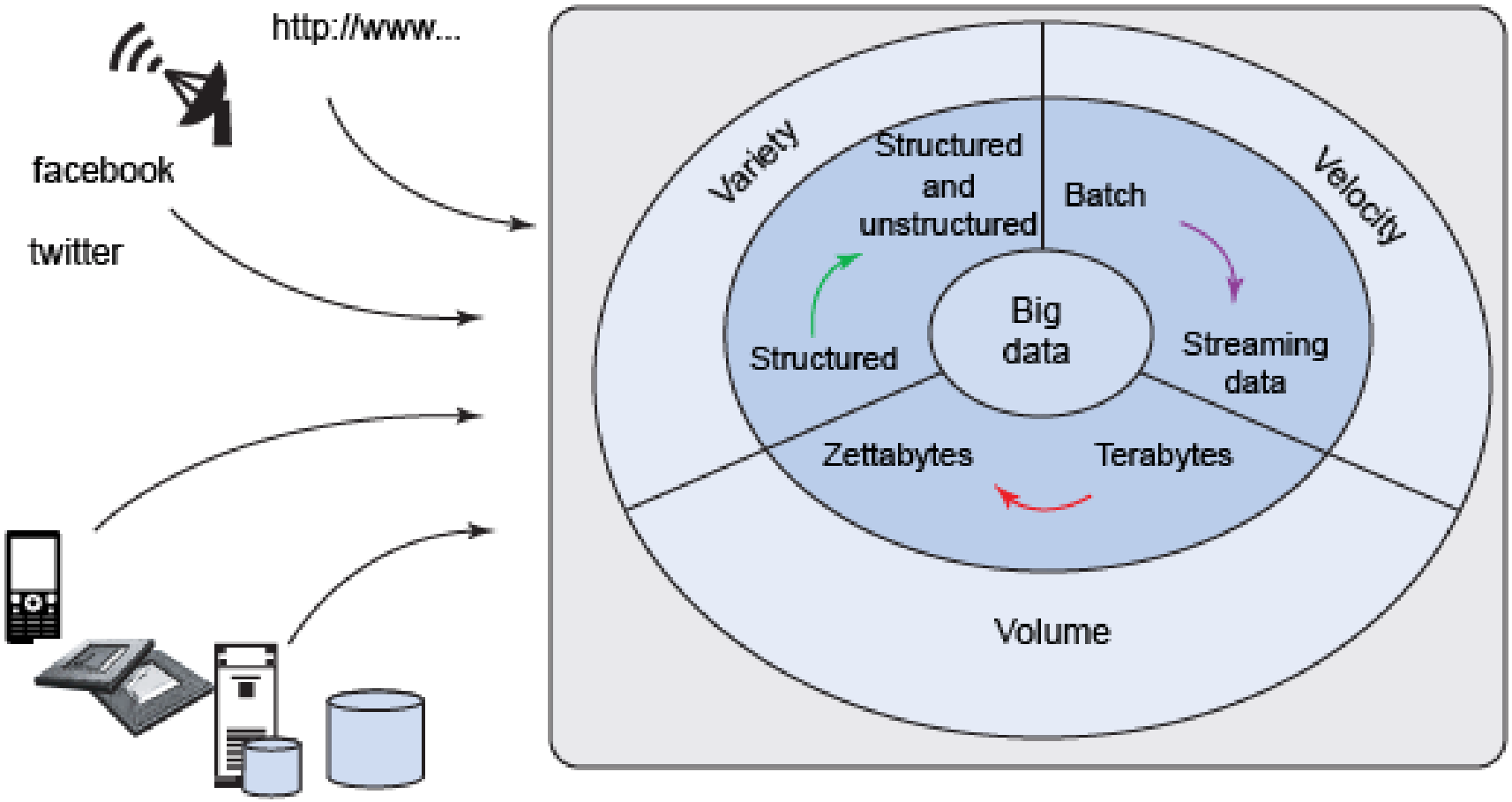
Simon Gog

Computing and Information Systems
The University of Melbourne

January 15th 2014

Big data: Volume, Velocity,...

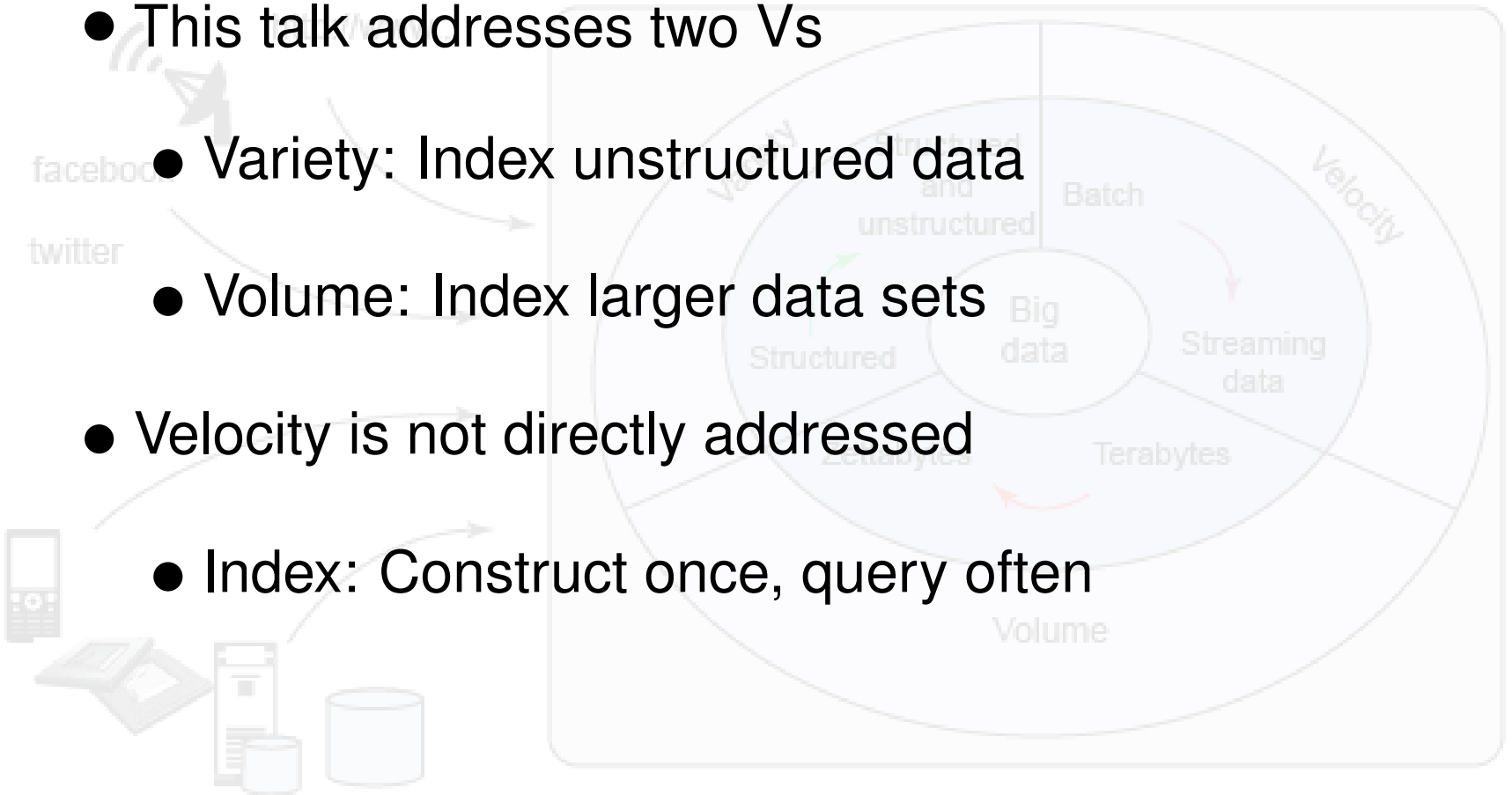
...and Variety



Picture source IBM

...and Variety

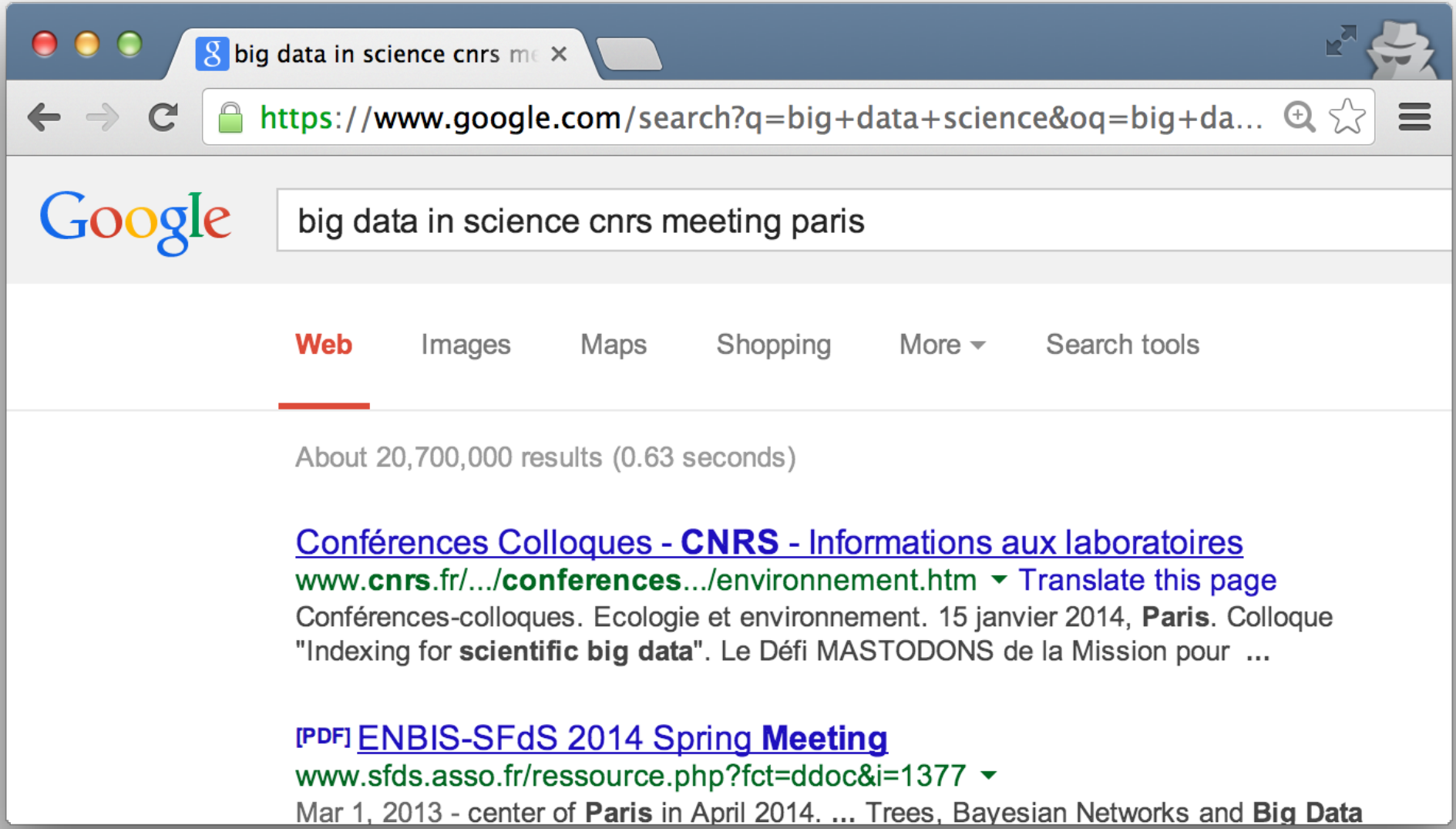
- This talk addresses two Vs
  - Variety: Index unstructured data
  - Volume: Index larger data sets
- Velocity is not directly addressed
  - Index: Construct once, query often



Picture source IBM

Outline

Index structures are the bases of information retrieval



Inverted Index

Inverted Indexes for a collection of documents C

- used for web indexing
- practical in domains with well-defined vocabulary
- space is usually $5 - 10\%$ of C (without positions)
- space is about 50% if positions are included
- can not be used to reconstruct original text (parsing, stemming, stopping,...)
- original text has to be stored separately to allow snippet extraction

Inverted Index

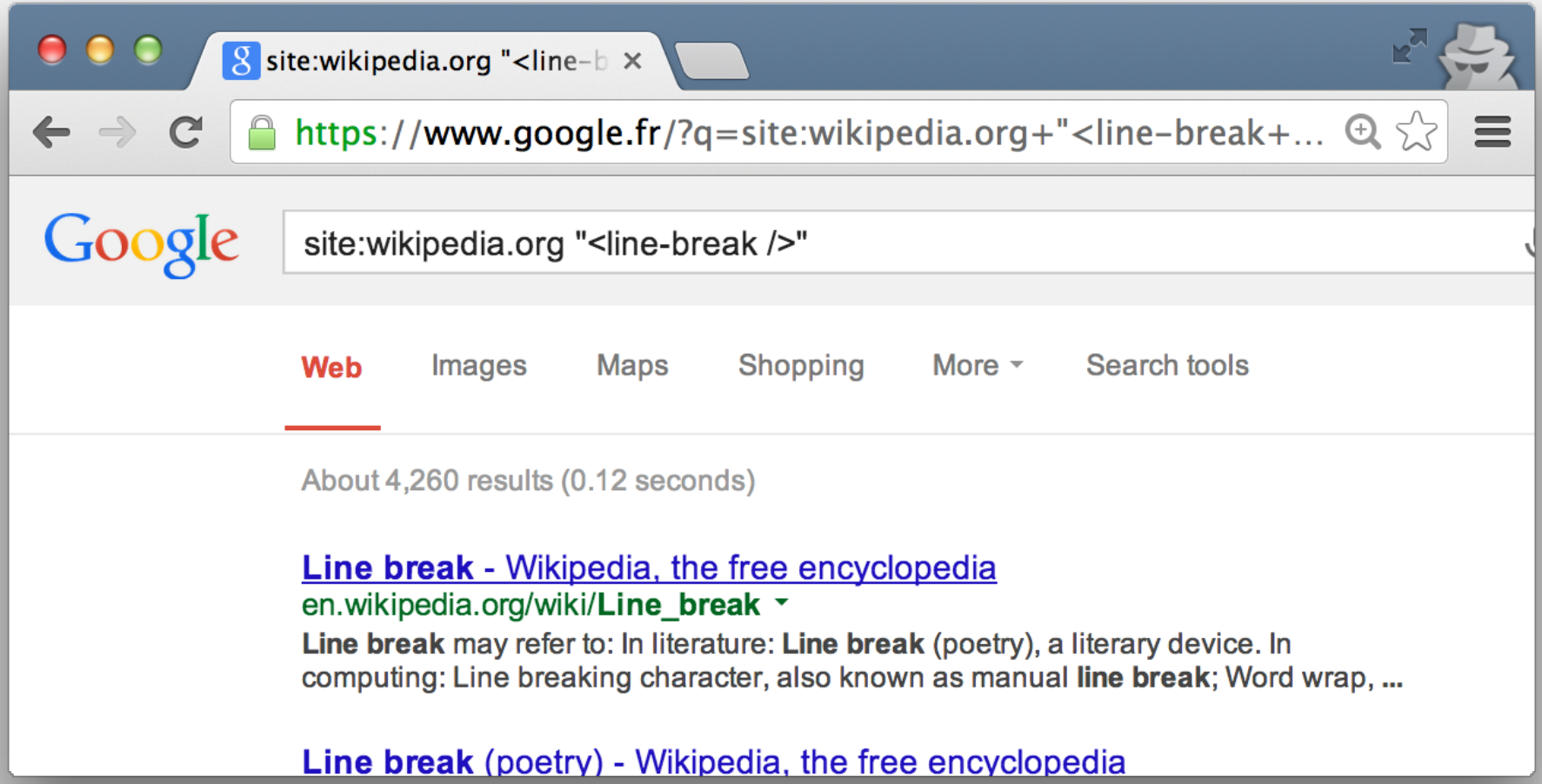
Documents (already normalized)

d_0 : is big data really big
d_1 : is it big in science
d_2 : big data is big

Inverted Lists

big	:	{(0,1),(0,4),(1,2),(2,0),(2,3)}
data	:	{(0,2),(2,1)}
in	:	{(1,3)}
is	:	{(0,0),(1,0),(2,2)}
really	:	{(0,3)}
science	:	{(1,4)}

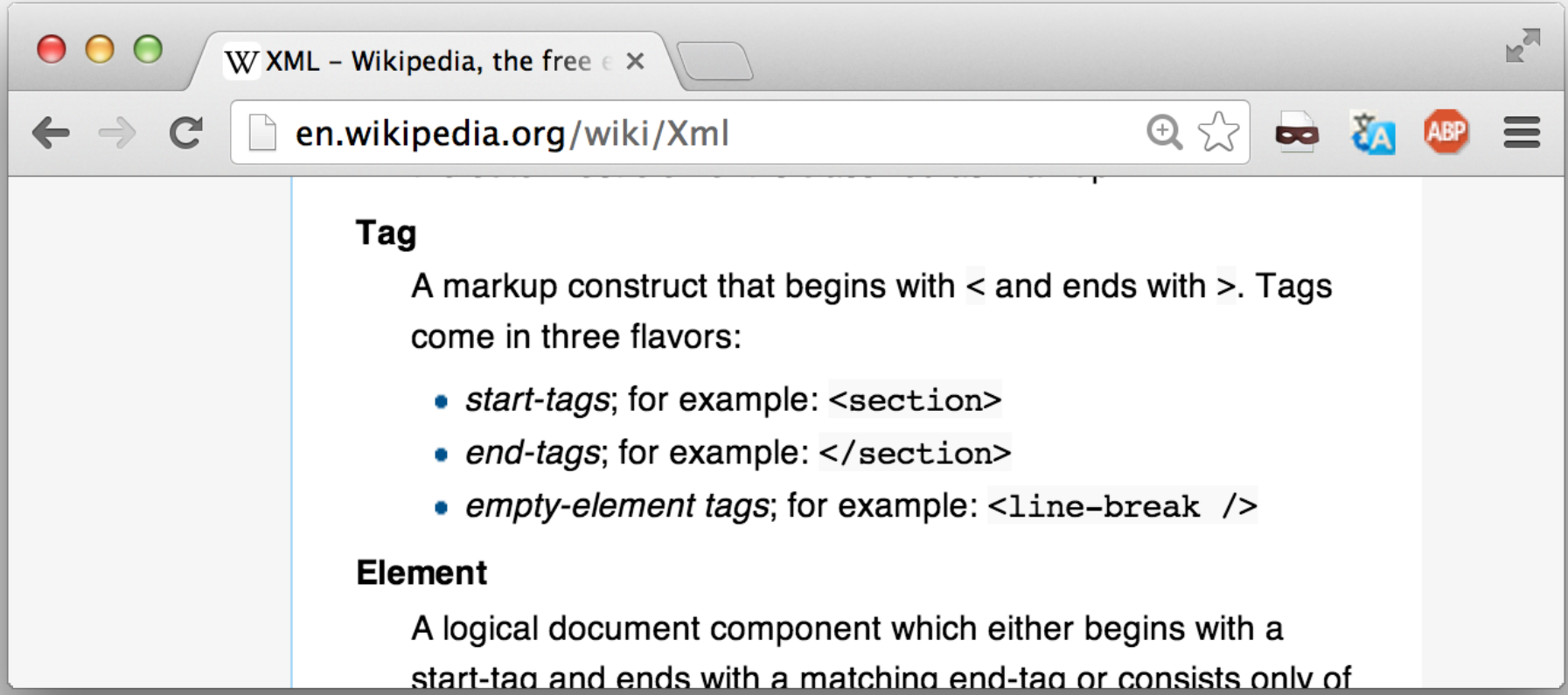
Exact search using Inverted Index based systems (1)



No document containing „<line=break />” is returned.

Exact search using Inverted Index based systems (2)

but it exists...



Summary: Inverted Indexes

Advantages:

- Sequential locality in many operations (fast!)
- Index can be stored on disk
- Good compression for natural language text

Disadvantages:

- Decision, what is searchable, made at parsing time
- No fast calculation of phrase queries
- No alphabet $\Rightarrow$ not applicable
- Adapt data to index structure

Search quality seems to be enough for casual web search.
Scientific tasks require high quality search.

Outline

Suffix sorted based indexes

Suffix sorted based indexes for a text T

- sort all n suffixes of T
- works also in domains with no well-defined vocabulary
- used in Bioinformatics
- space
  - uncompressed: $5\times$ to $50\times$ size of T
  - compressed: $n \times H_k(T) \leq$ bzip'd T

    index structure adapts to data
- can be used to reconstruct original text (Self-Index)

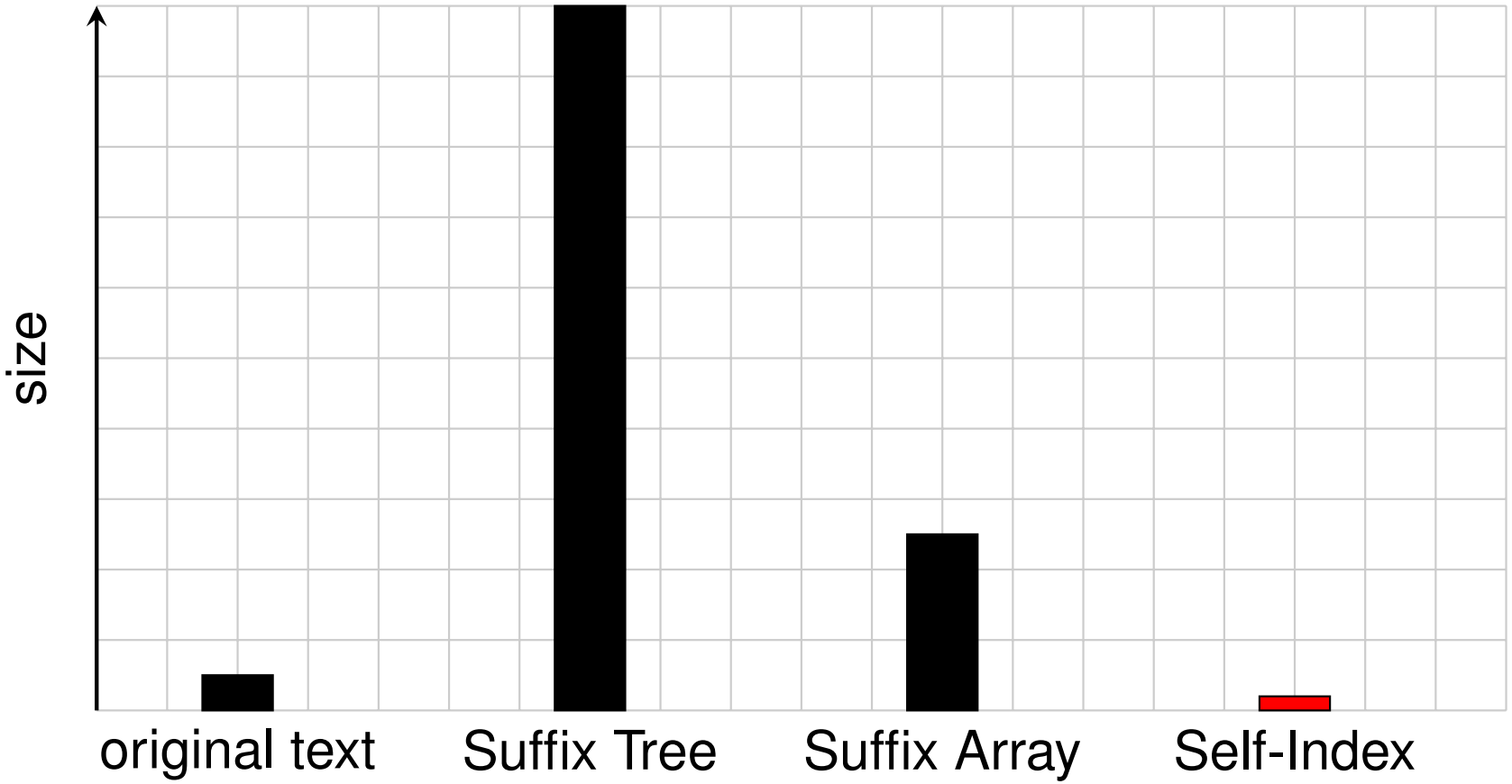
Not cover in this talk: LZ compressed indexes (works well for highly repetitive data)

H_k of *Pizza&Chili* corpus texts (200MB versions)

$H_k(T)$ in bits contexts/$|T|$ in percent

k	DBLP.XML		DNA		ENGLISH		PROTEINS	
0	5.26	0.0	1.97	0.0	4.53	0.0	4.20	0.0
1	3.48	0.0	1.93	0.0	3.62	0.0	4.18	0.0
2	2.17	0.0	1.92	0.0	2.95	0.0	4.16	0.0
3	1.43	0.1	1.92	0.0	2.42	0.0	4.07	0.0
4	1.05	0.4	1.91	0.0	2.06	0.3	3.83	0.1
5	0.82	1.3	1.90	0.0	1.84	1.0	3.16	1.7
6	0.70	2.7	1.88	0.0	1.67	2.7	1.50	17.4

Index size comparison (English text)



Self-Index Operations

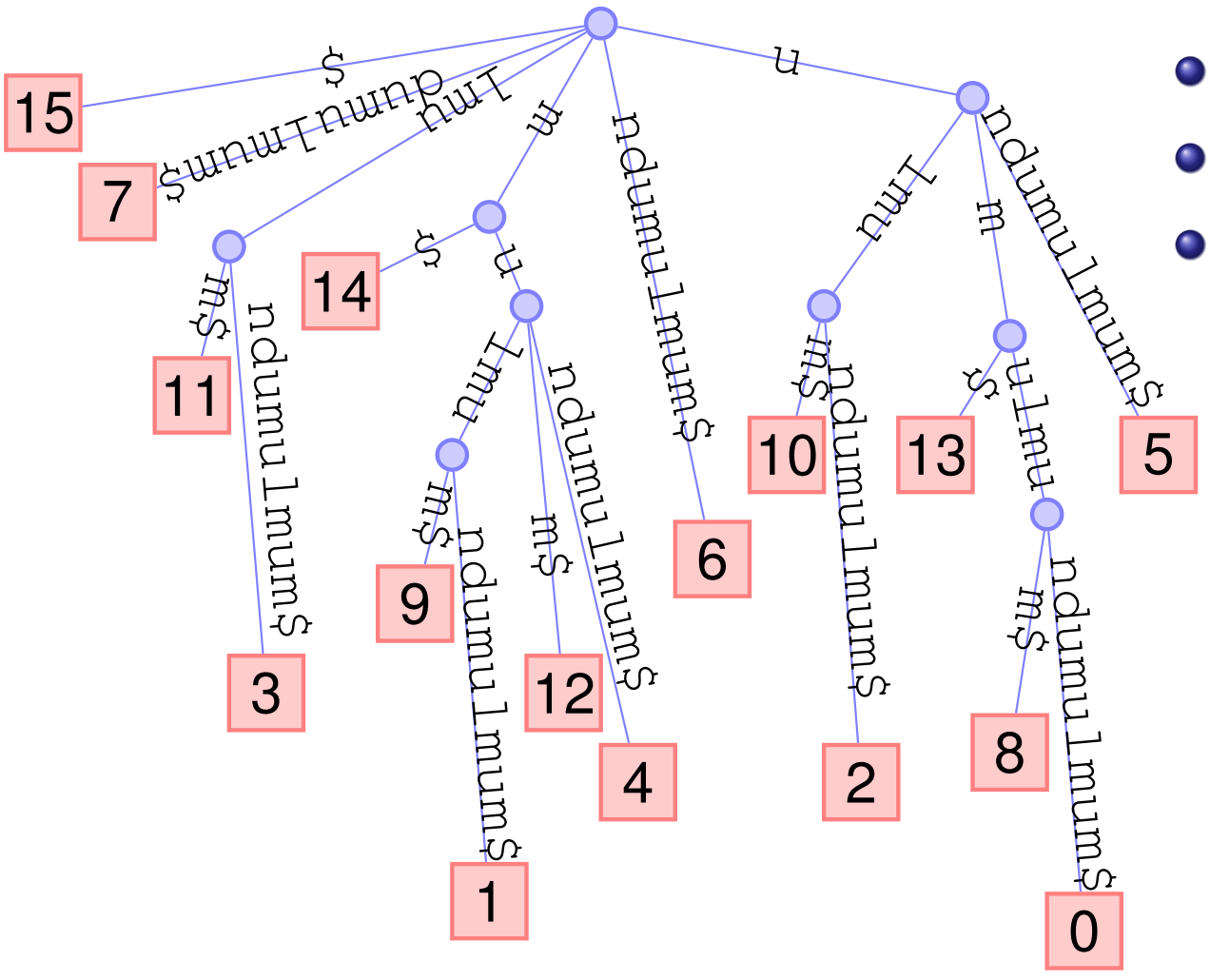
Let P be a character pattern of length $m = |P|$.

Operations:

`count(`P`)`	How many times does P occur in T?
`locate(`P`)`	List all count(P) positions where P occurs.
`extract(`i`,`j`)`	Extract text $T[i..j]$ from the index.

Operations are efficient (like $\mathcal{O}(m \log \sigma)$, $\mathcal{O}(m \log n)$)

Suffix Tree (ST) of T=umulmundumulmum$



- $n = 16$
- $\Sigma = \{\$, \mathtt{d}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{u}\}$
- $\sigma = 6$

$\mathcal{O}(n)$ construction (Weiner,1973)

Problem:
Structure size:
$\geq 20 \times |T|$

Suffix Array (SA) of T=umulmundumulmum$

i		T
0	0	umulmundumulmum$
1	1	mulmundumulmum$
2	2	ulmundumulmum$
3	3	lmundumulmum$
4	4	mundumulmum$
5	5	undumulmum$
6	6	ndumulmum$
7	7	dumulmum$
8	8	umulmum$
9	9	mulmum$
10	10	ulmum$
11	11	lmum$
12	12	mum$
13	13	um$
14	14	m$
15	15	$

Suffix Array (SA) of T=umulmundumulmum$

i	SA	T
0	15	$
1	7	dumulmum$
2	11	lmum$
3	3	lmundumulmum$
4	14	m$
5	9	mulmum$
6	1	mulmundumulmum$
7	12	mum$
8	4	mundumulmum$
9	6	ndumulmum$
10	10	ulmum$
11	2	ulmundumulmum$
12	13	um$
13	8	umulmum$
14	0	umulmundumulmum$
15	5	undumulmum$

- Size: $\geq 5 \times |T|$
- Binary search to answer `count`$(|P|)$
- Match pattern from left to right: e.g. `mum`

Burrows-Wheeler Transform (BWT) of T

i	SA	T^{BWT}	T
0	15	m	$
1	7	n	dumulmum$
2	11	u	lmum$
3	3	u	lmundumulmum$
4	14	u	m$
5	9	u	mulmum$
6	1	u	mulmundumulmum$
7	12	l	mum$
8	4	l	mundumulmum$
9	6	u	ndumulmum$
10	10	m	ulmum$
11	2	m	ulmundumulmum$
12	13	m	um$
13	8	d	umulmum$
14	0	$	umulmundumulmum$
15	5	m	undumulmum$

- Well-compressible
- New search algorithm (Ferragina & Manzini, 2000): Backward search
- Does not require SA.

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

Array C contains start of each character interval:

$	a	b	c	d	r	r+1
0	1	9	13	14	15	19

- $rank(i, c) =$ # occurences of c in $T^{BWT}[0, i)$.
- Search backwards for `bar`.

Backward Search

i	T^{BWT}	T
0	a	$abracadabrabarbara
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = C[r] + rank(sp_0, r)$
  $ep_1 = C[r] + rank(ep_0 + 1, r) - 1$

Backward Search

i	T^{BWT}	T
0	a	$abracadabrabarbara
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + rank(0, r)$
  $ep_1 = 15 + rank(19, r)$

Backward Search

i	T^{BWT}	T
0	a	$abracadabrabarbara
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval:
  $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + 0$
  $ep_1 = 15 + rank(19, r) - 1$

Backward Search

i	T^{BWT}	T
0	a	$abracadabrabarbara
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15+0 = 15$
  $ep_1 = 15+4-1 = 18$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = C[a] + rank(sp_1 - 1, a)$
  $ep_2 = C[a] + rank(ep_1, a) - 1$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a)$
  $ep_2 = 1 + rank(ep_1, a)$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a)$
  $ep_2 = 1 + rank(ep_1, a)$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + 6$
  $ep_2 = 1 + rank(19, a) - 1$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1+6 = 7$
  $ep_2 = 1+8-1 = 8$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = C[b] + rank(sp_2, b)$
  $ep_3 = C[b] + rank(ep_2 + 1, b) - 1$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b)$
  $ep_3 = 9 + rank(ep_1, b)$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b)$
  $ep_3 = 9 + rank(ep_1, b)$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + 0$
  $ep_3 = 9 + rank(9, b) - 1$

Backward Search

i	T^{BWT}	T
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Now search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9+0 = 9$
  $ep_3 = 9+2-1 = 10$

Conclusion Backward Search

String matching on T only requires

- a structure which can answer rank queries on T^{BWT}
- the C array

Text T itself is not needed.

We search a data structure

- represents T^{BWT} compressed
- answers rank quickly

The Wavelet Tree (WT)

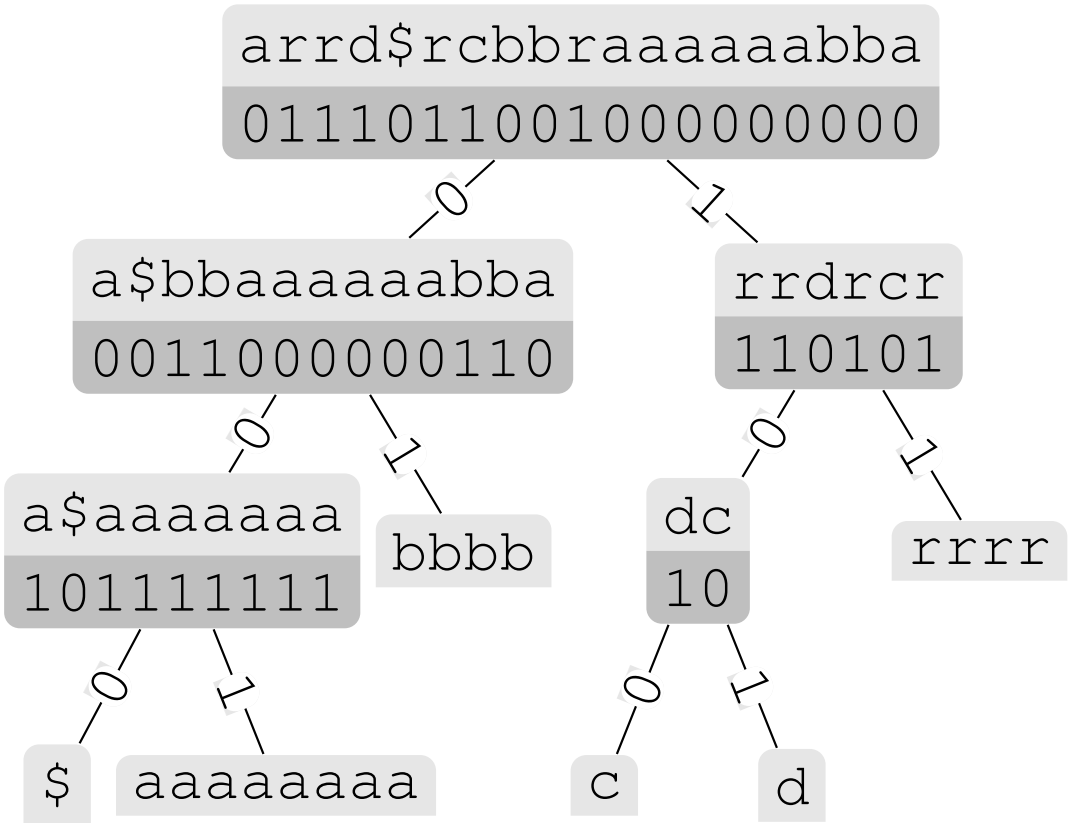
Let X be a sequence of length n over alphabet Σ of size σ.

Efficient Wavelet Tree operations:

`access(`i`)`	Recover $X[i]$.
`rank(`i`,`c`)`	How many times does c occur in $X[0, i)$?
`select(`i`,`c`)`	Location of the i-th count(P) positions where P occurs.

Space: $\approx n \log \sigma + o(n \log \sigma) + \sigma \log n$

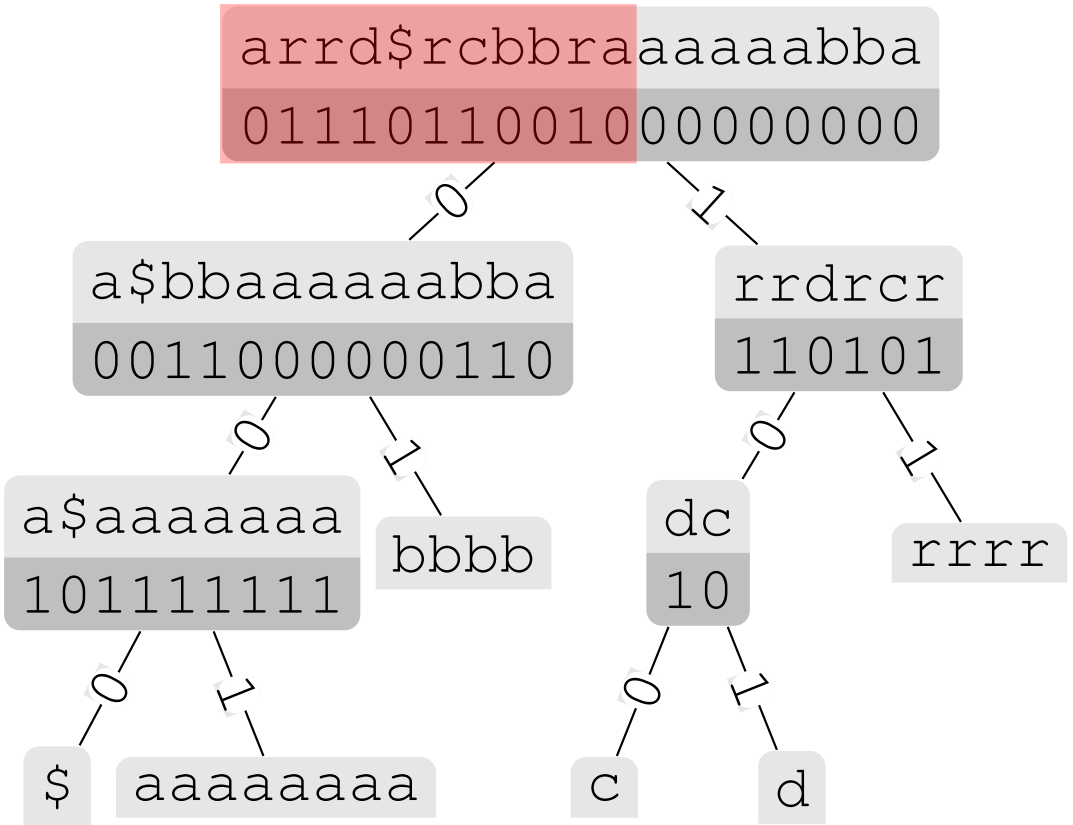
WT operation `rank`



- Only the bitvectors (dark gray) is stored
- $code(\texttt{a}) = \texttt{001}$
- Reduce WT rank on bitvector rank

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

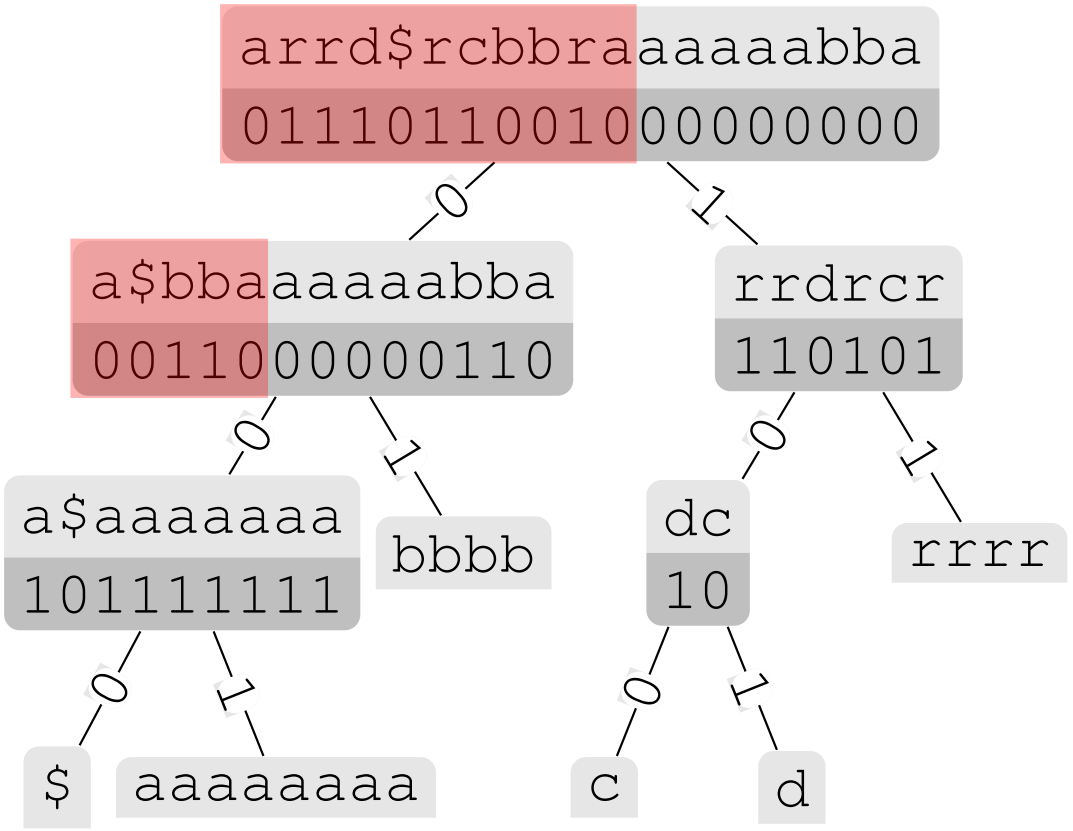
WT operation `rank`



- Only the bitvectors (dark gray) is stored
- $code(\texttt{a}) = \texttt{001}$
- Reduce WT rank on bitvector rank

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

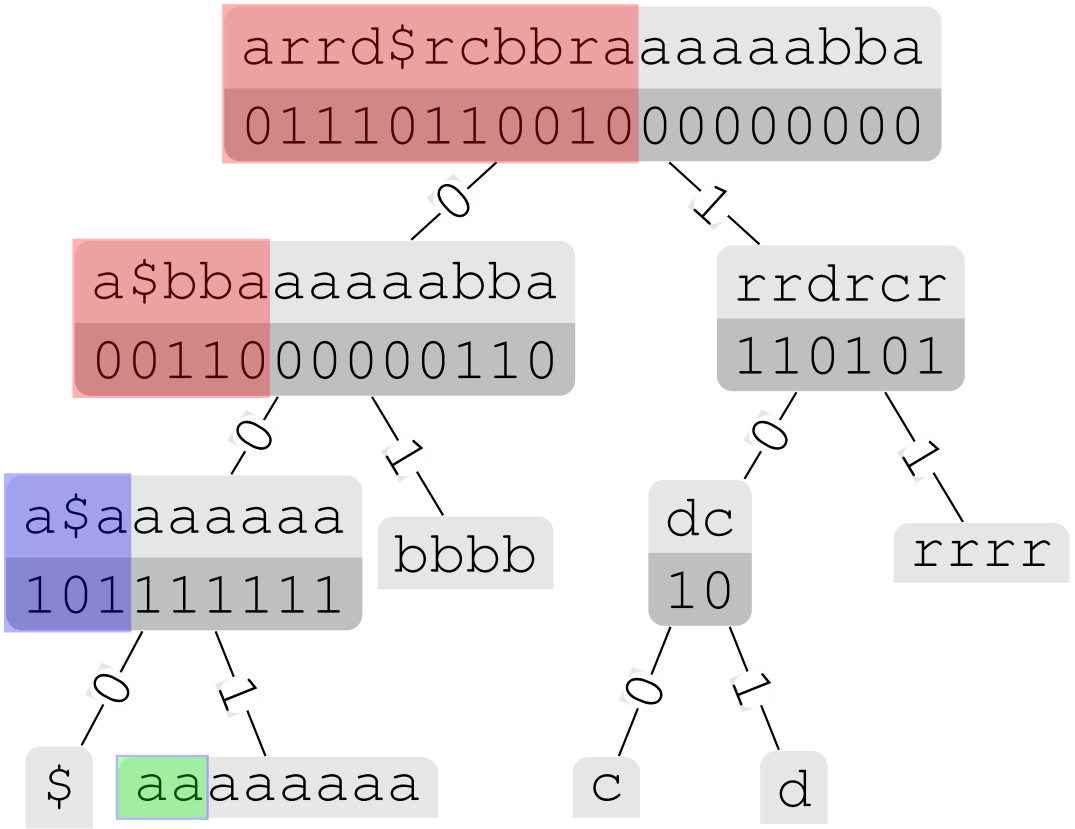
WT operation `rank`



- Only the bitvectors (dark gray) is stored
- $code(\texttt{a}) = 001$
- Reduce WT rank on bitvector rank

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

WT operation `rank`



- Only the bitvectors (dark gray) is stored
- $code(\texttt{a}) = 001$
- Reduce WT rank on bitvector rank

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

Rank on bitvectors

$n + o(n)$ bit space and constant time solution

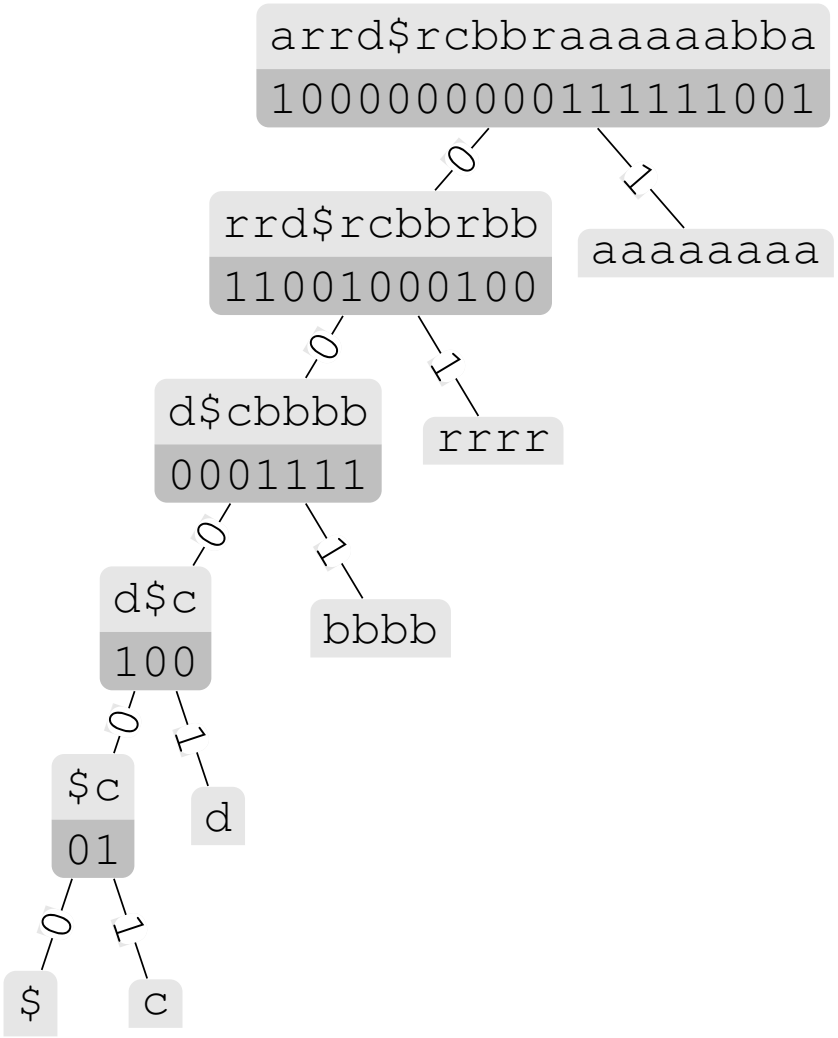
Jacobson: *Space-efficient Static Trees and Graphs*, FOCS 1989.
Two level schema:

- Divide bitvector in superblocks of size $\log^2 n$ and blocks of $\log n/2$.
- Store absolute prefix sums for superblocks.
- Relative perfix sums for blocks.
- Four Russian-Trick (Lookup-Table counting set bits) for blocks.

Practice today

Since 2008 CPUs support `popcount` operation $\Rightarrow$ no need for Lookup-Tables.

Improving WT space: Huffman shape



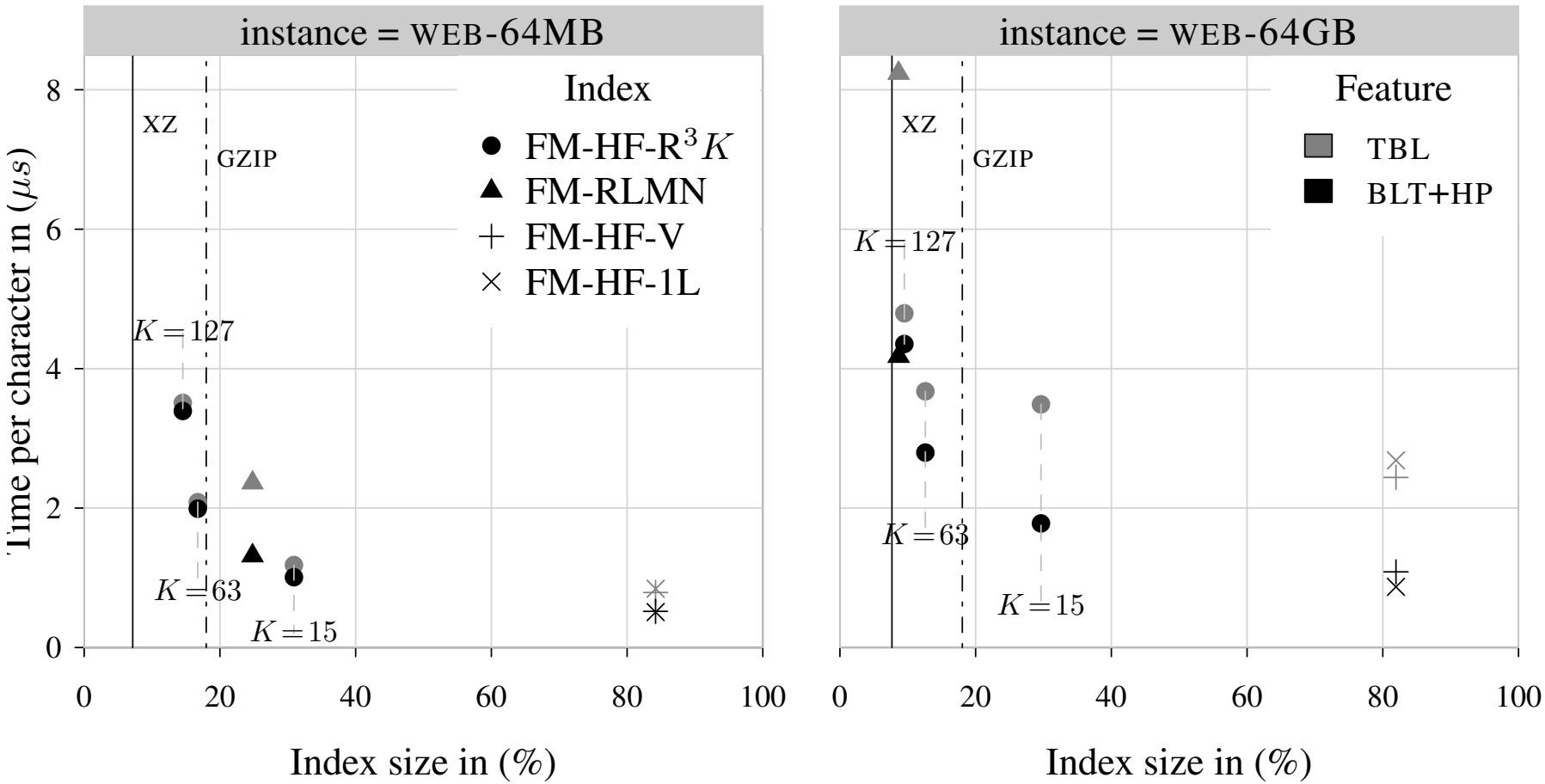
Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

Avg. depth: $H_0(BWT)$. Total space: $\approx nH_0 + 2\sigma \log n$

Other WT parametrizations

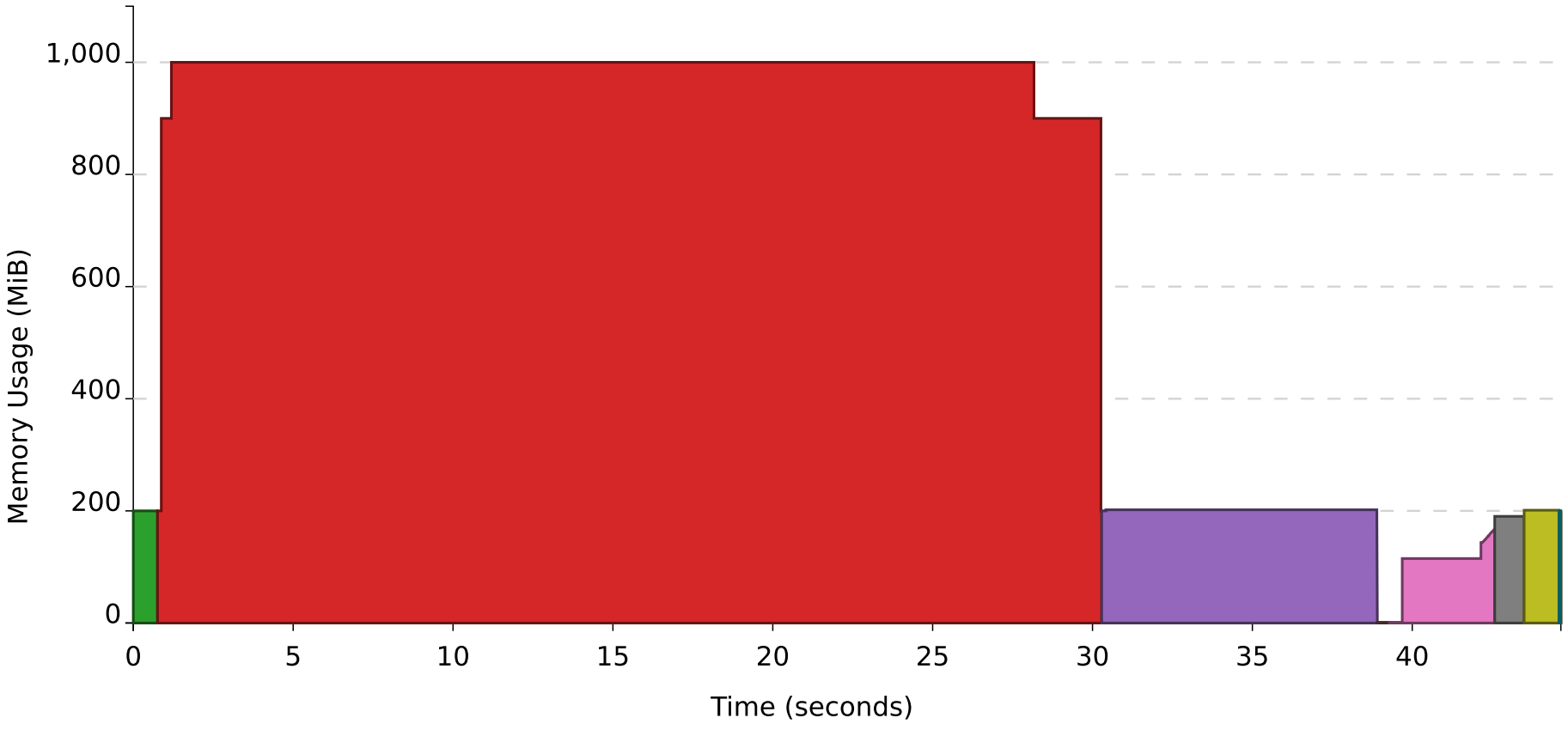
- Shape: balanced, Huffman-shape, Hu-Tucker-shape, Wavelet Matrix
- for large alphabets (decreases $\mathcal{O}(\sigma \log n)$ to $\mathcal{O}(\log \sigma \log n)$)
- run-length compressed versions
- parameterized bitvector

Experimental results: operation `count`



(G & Petri, 2014)

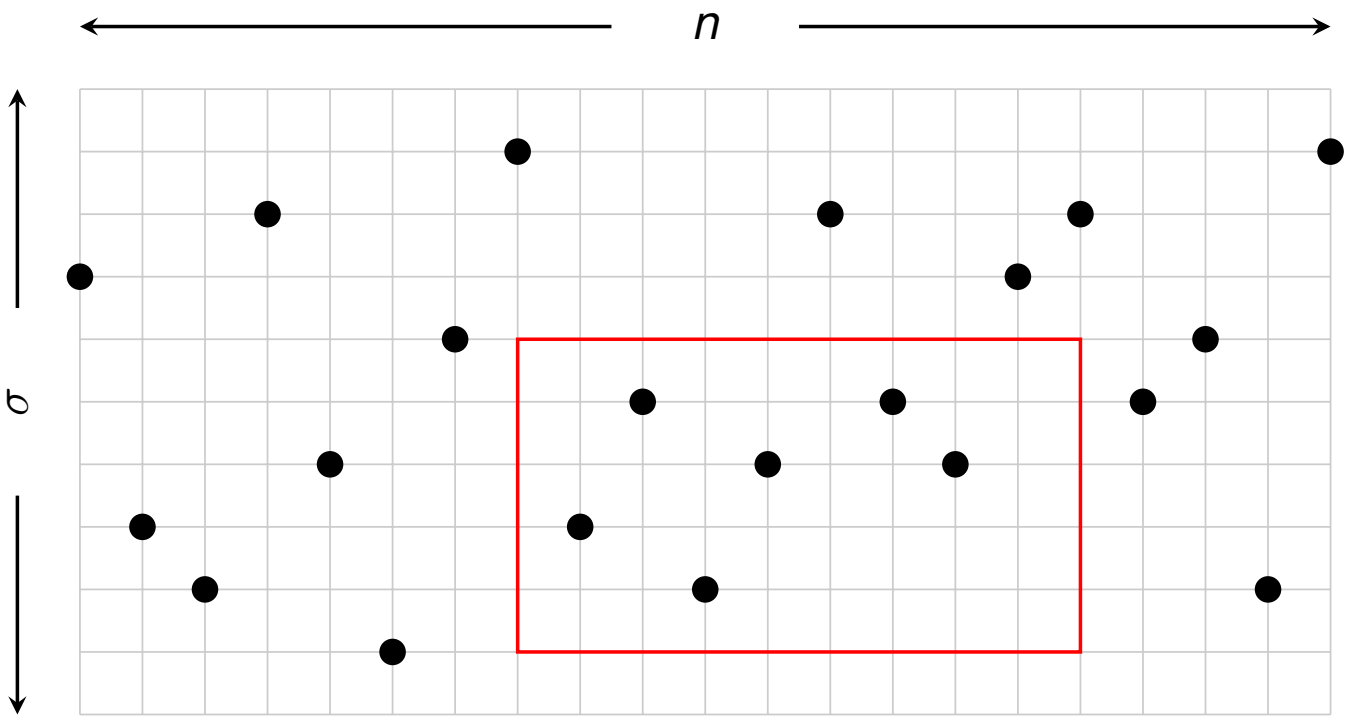
Experimental results: construction



Resource diagram for the construction of a Self-Index for 200MB of English text.

Outline

More WT operations: Query point grids



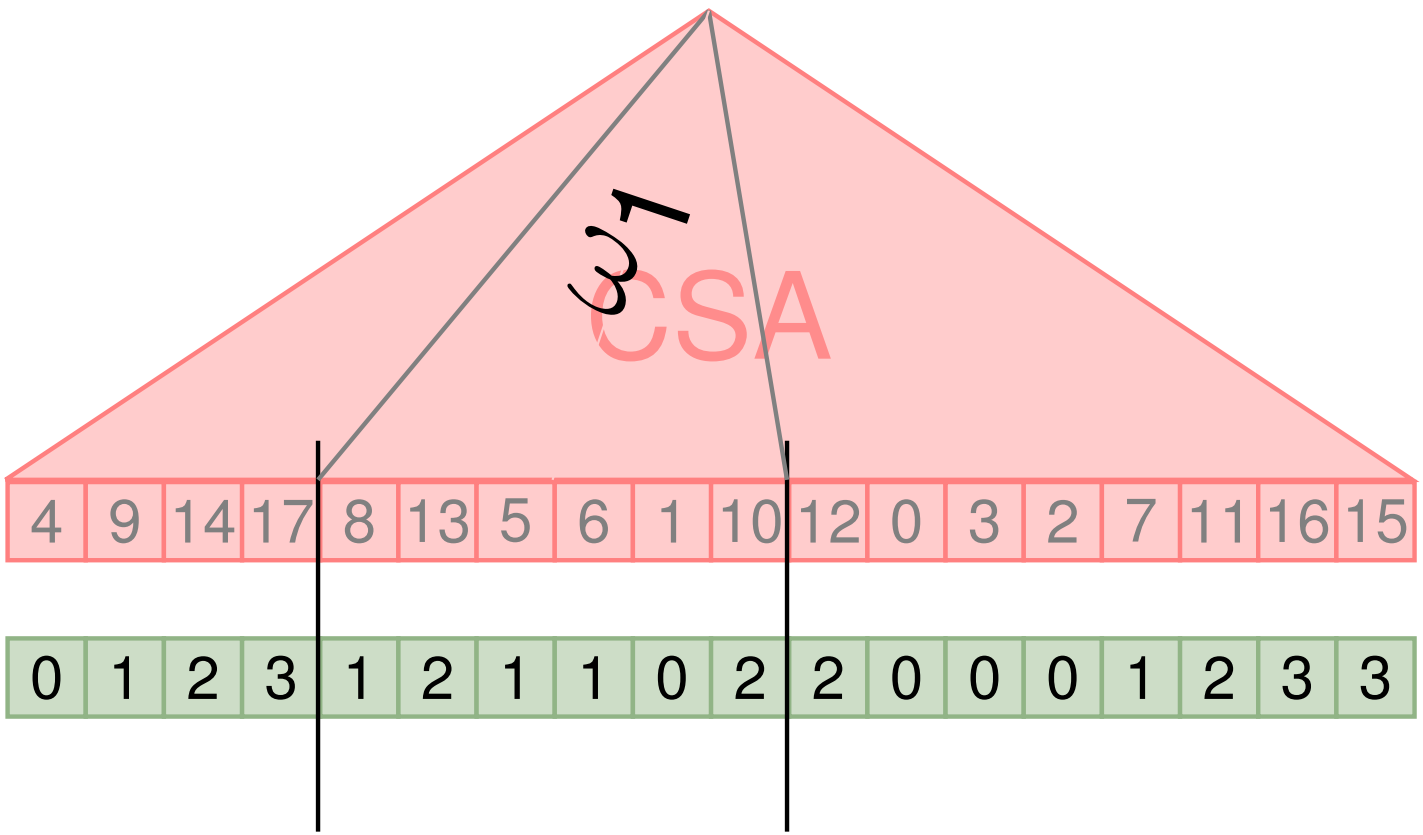
Operations:

`count`(x_0, x_1, y_0, y_1) # points in rectangle $[x_0, x_1][y_0, y_1]$.

`report`(x_0, x_1, y_0, y_1) Report points in rectangle $[x_0, x_1][y_0, y_1]$.

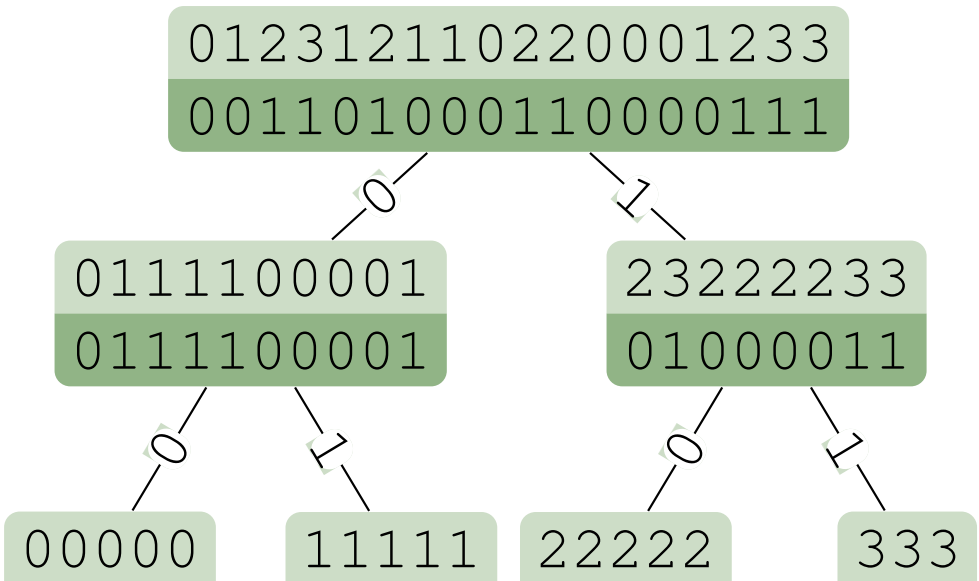
WT application: Top-k document retrieval

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\mathcal{T} =$	ω_2	ω_1	ω_3	ω_3	#	ω_1	ω_1	ω_4	ω_1	#	ω_1	ω_4	ω_3	ω_1	#	ω_5	ω_5	#
$b =$	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1



WT application: Top-k document retrieval

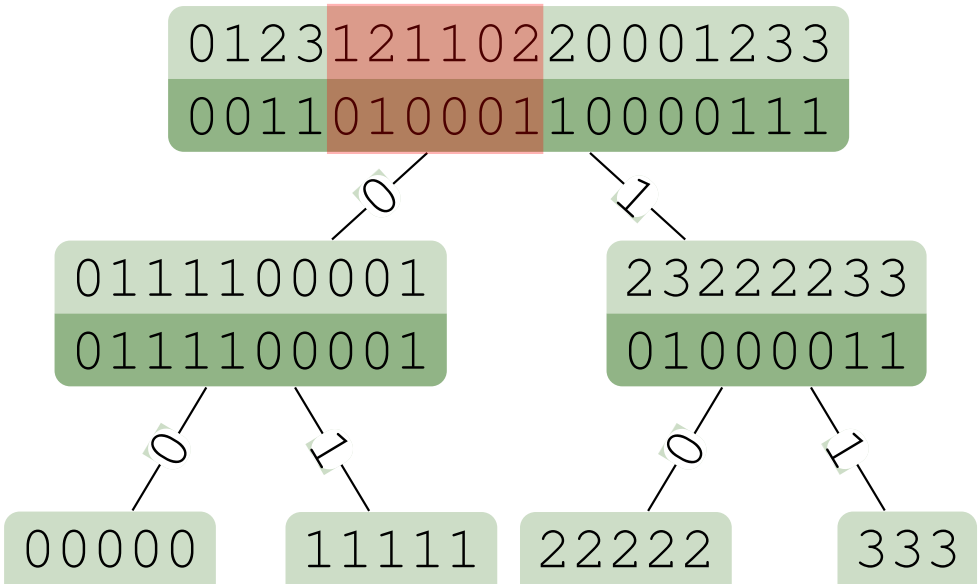
- Represent document array $\mathcal{D}$ as wavelet tree
- First explore nodes, which contain the maximal interval
- Example: Search top-2 documents (frequency based ranking)



Top documents containing ω_1:

WT application: Top-k document retrieval

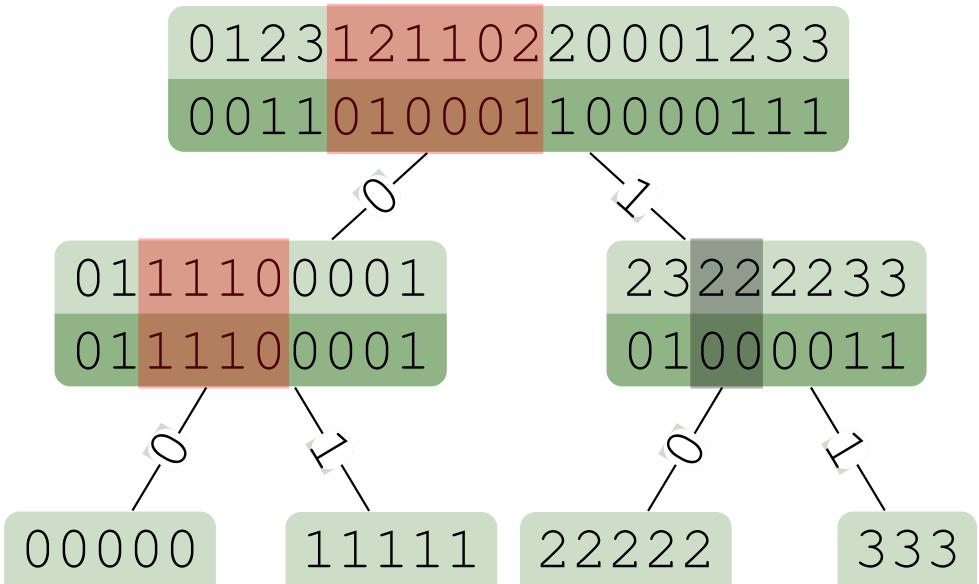
- Represent document array $\mathcal{D}$ as wavelet tree
- First explore nodes, which contain the maximal interval
- Example: Search top-2 documents (frequency based ranking)



Top documents containing ω_1:

WT application: Top-k document retrieval

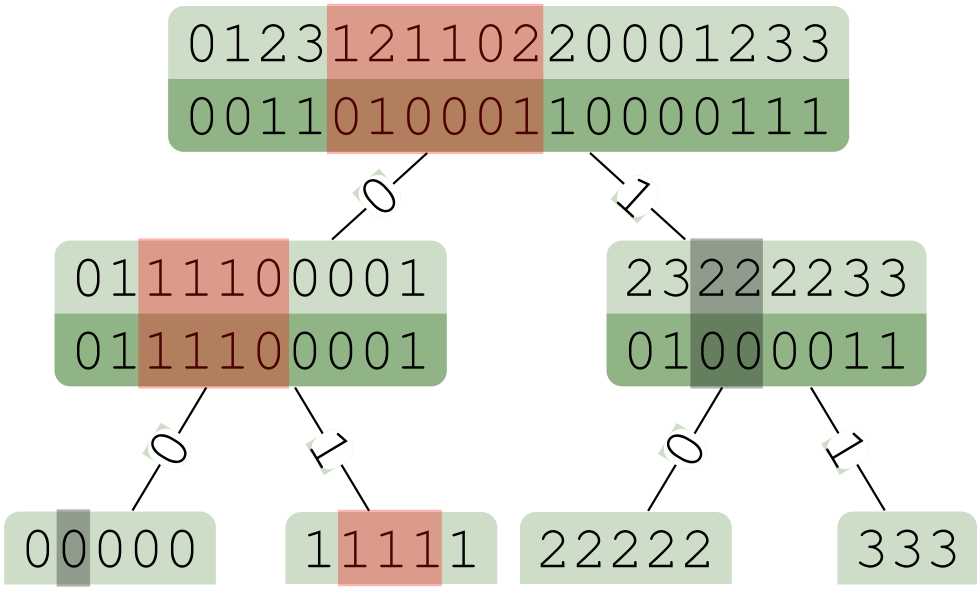
- Represent document array $\mathcal{D}$ as wavelet tree
- First explore nodes, which contain the maximal interval
- Example: Search top-2 documents (frequency based ranking)



Top documents containing ω_1:

WT application: Top-k document retrieval

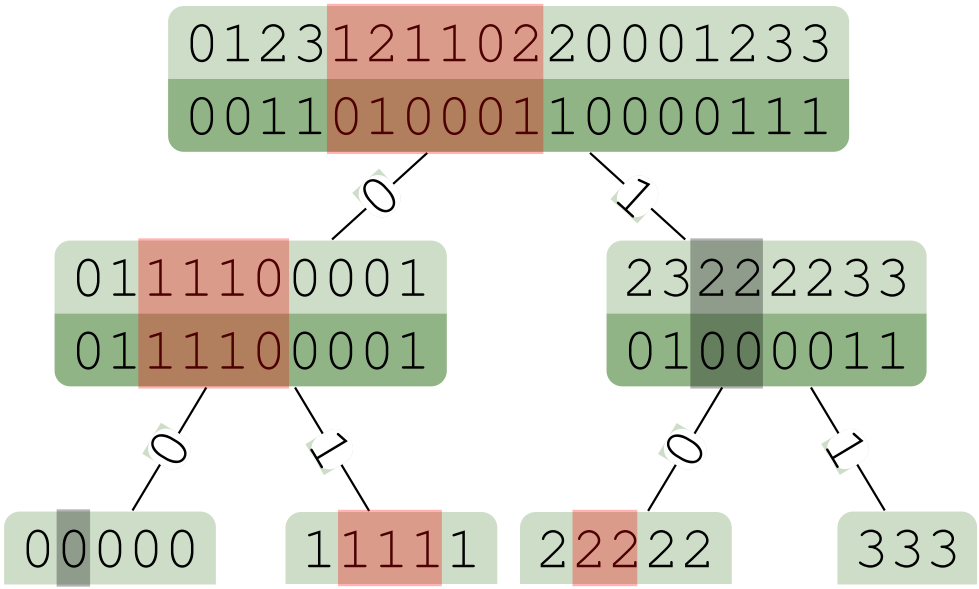
- Represent document array $\mathcal{D}$ as wavelet tree
- First explore nodes, which contain the maximal interval
- Example: Search top-2 documents (frequency based ranking)



Top documents containing ω_1: 1 (3 times)

WT application: Top-k document retrieval

- Represent document array $\mathcal{D}$ as wavelet tree
- First explore nodes, which contain the maximal interval
- Example: Search top-2 documents (frequency based ranking)



Top documents containing ω_1: 1 (3 times), 2 (2 times)

WT application: Top-k document retrieval

Other results:

- $2n + o(n)$ structure answers document frequency in constant time (Sadakane 2007)
- Use RMQs for document listing (Sadakane 2007)
- Prestore selected intervals (Hon et al. 2009). Use skeleton Suffix Tree for mapping.

Outline

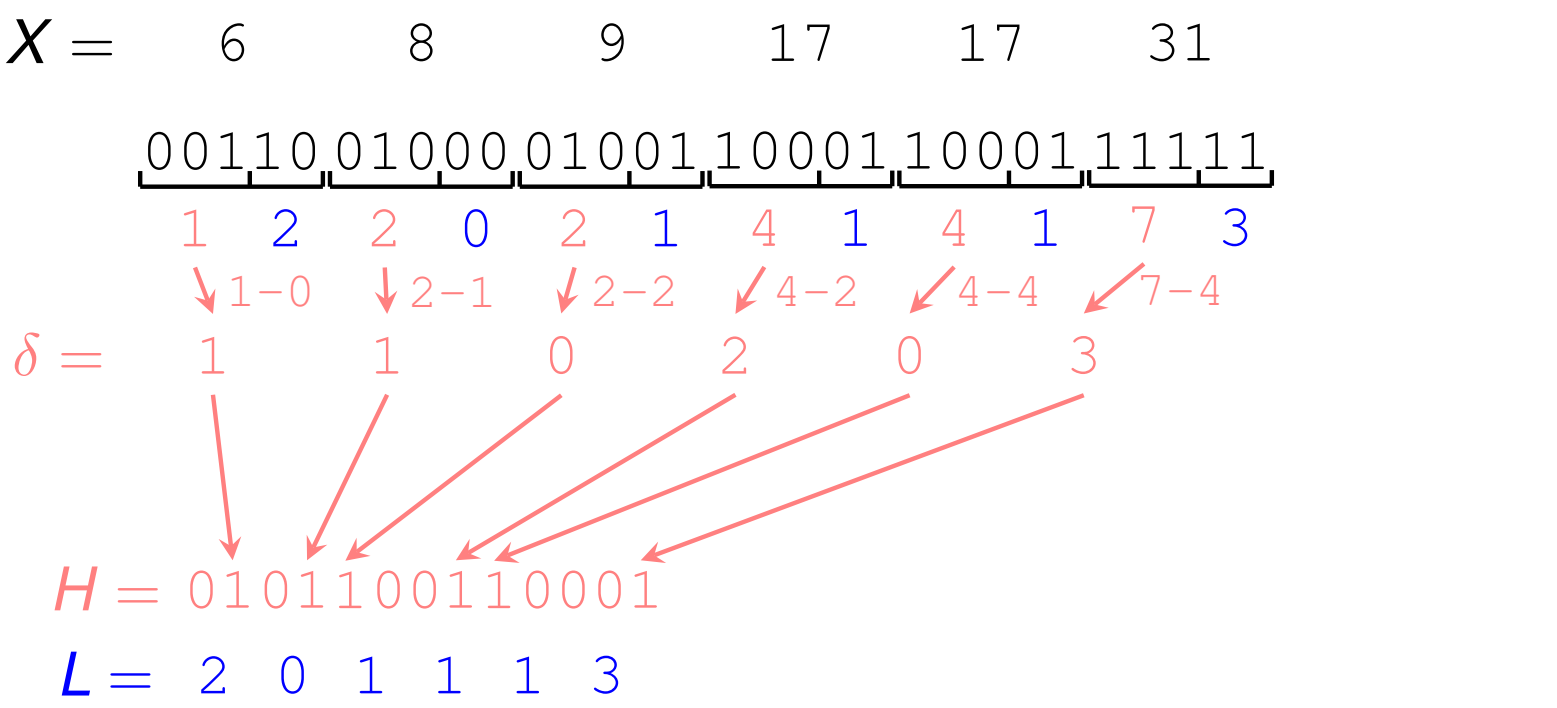
Compressed Bitvectors: SD-array (Elias-Fano coding)

Applications

- Graphs (deBruijn subgraph, web graph,...)
- Lists (Inverted Lists,...)

- monotone sequence X with $0 \leq x_0 \leq x_1 \leq \ldots \leq x_{m-1} \leq n$.
- Lower $\ell = \lfloor \log \frac{n}{m} \rfloor$ bits stored explicitly.
- Upper bits as sequence H of unary encoded gaps
- Uses $2 + \lfloor \log \frac{n}{m} \rfloor$ bits per element
- Constant time access X by addding $o(m)$ extra bits to H.

Compressed Bitvectors: SD-array (Elias-Fano coding)



$$\text{X[3]}=rank_0(H, select_1(H, 4)) \cdot 2^2 + L[3] = 4 \cdot 4 + 1 = 17$$

Compressed Bitvectors: SD-array (Elias-Fano coding)

- Interpret X as positions of set bits in a bitvector b.
- Constant time `acccess`, `rank`, `select` on b.
- Well suited for sparse bitvectors.

$$b = \begin{array}{c} \scriptstyle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1 \\ 0\,0\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

$$H = 0\,1\,0\,1\,1\,0\,0\,1\,1\,0\,0\,0\,1$$

$$L = \begin{array}{ccccccc} 2 & 0 & 1 & 1 & 1 & 3 \end{array}$$

$$select(b, 4) = rank_0(H, select_1(H, 4)) \cdot 2^2 + L[3] = 4 \cdot 4 + 1 = 17$$

$rank(b, i)$ and $access(b, i)$: use $select_0$ on H to select right upper part + scan left for entry in L

Balanced Parentheses Sequences (BPS)

Represent BPS as bitvector.

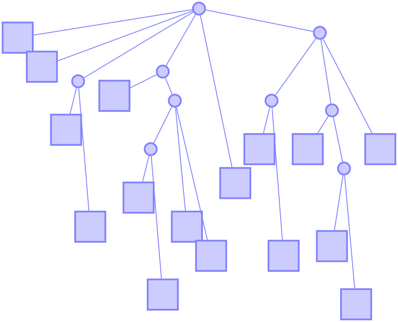
(()()(()())(()((()())()()))()((()())(()(()())))())

`find_open(`i`)`	Find matching closing paren of $BPS[i]$.
`find_close(`P`)`	Find matching opening paren of $BPS[i]$.
`enclose(`i`)`	Find tightest paren pair which encloses pair (i,`find_close(`i`)`)

and more (`double_enclose(`i,j`)`, `rank(`i`)`, `select(`i`)`,...) can be answered in constant time by just adding $o(n)$ space. (Munro 1996, Geary et al. 2004 & 2006)

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



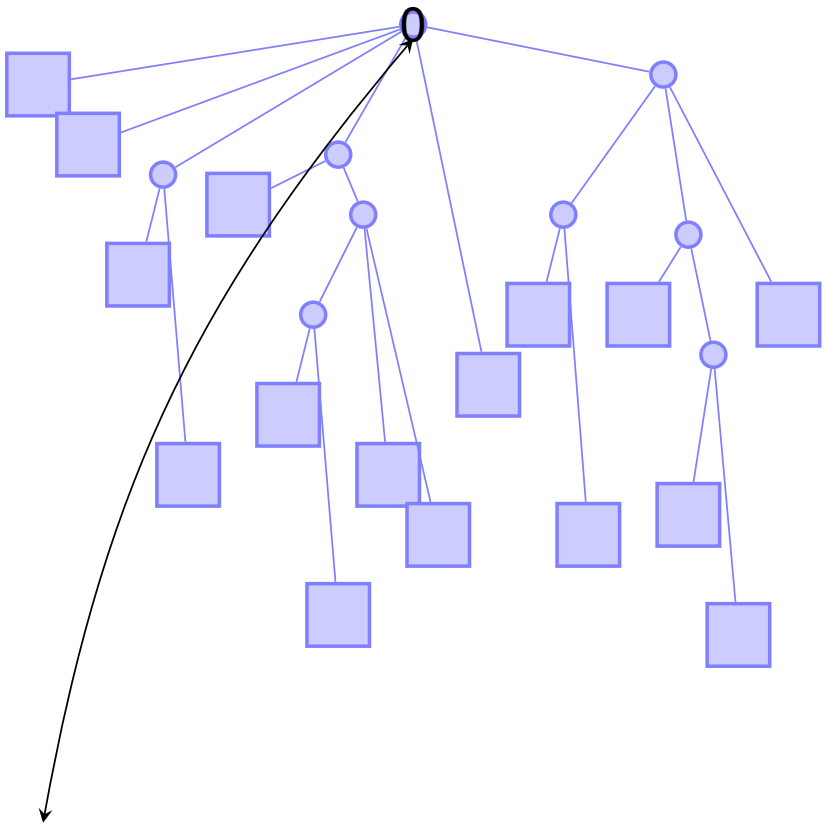
tree uncompressed
$\mathcal{O}(n \log n)$ bits

BPS_{dfs}= (()()(()())(()((()())()()))()((()())(()(()()))()))

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



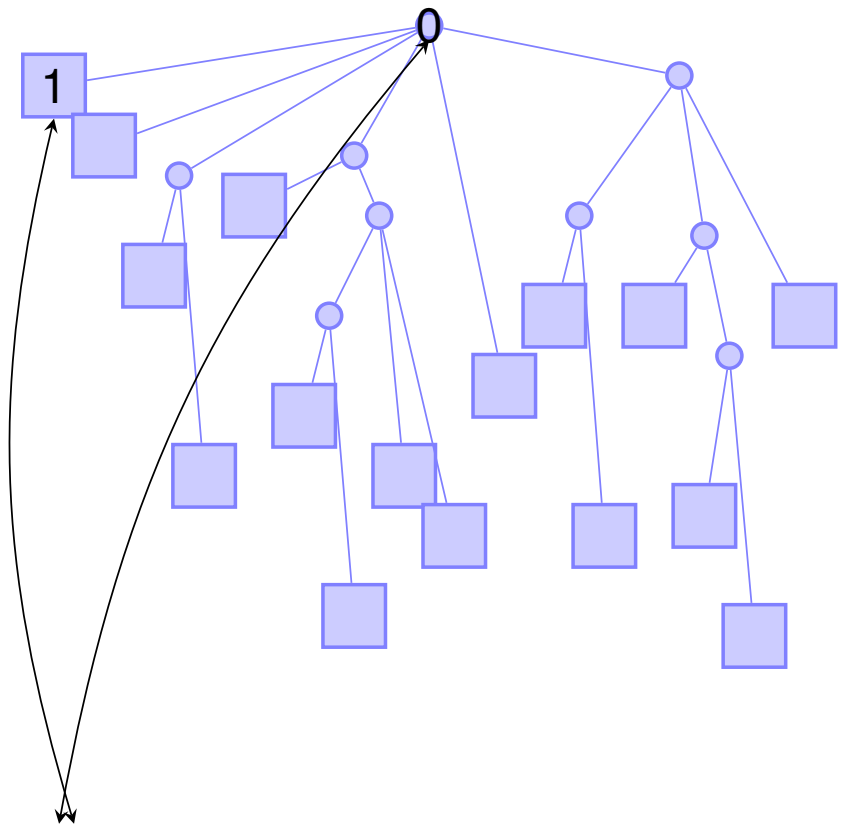
BPS_{dfs}= ((()()())(()(()())()())()(()())(()(()())))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



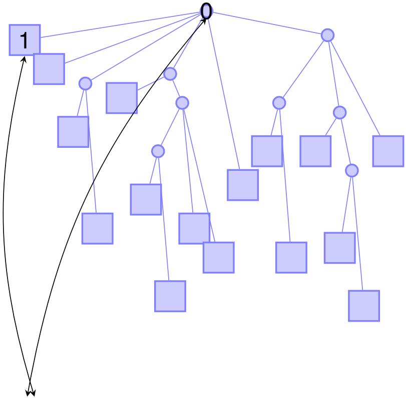
BPS_{dfs}= ((()()(()())(()(()())()())()(()()))(()(()())()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



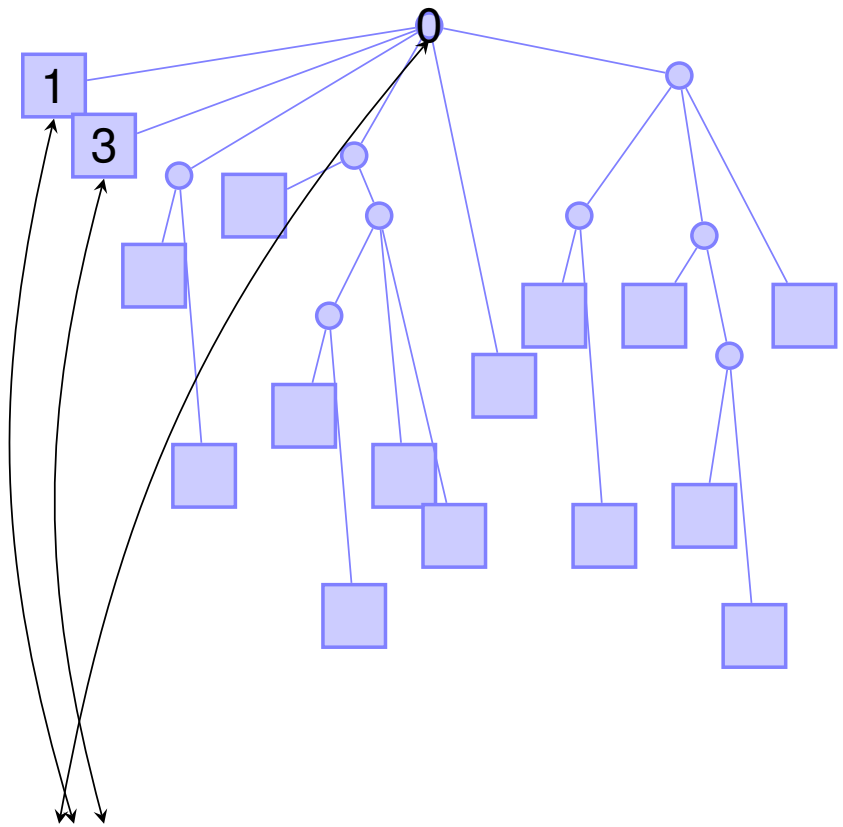
BPS_{dfs}= (()()(()())(()((()())(()()))()(((()())(()(()()))())))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



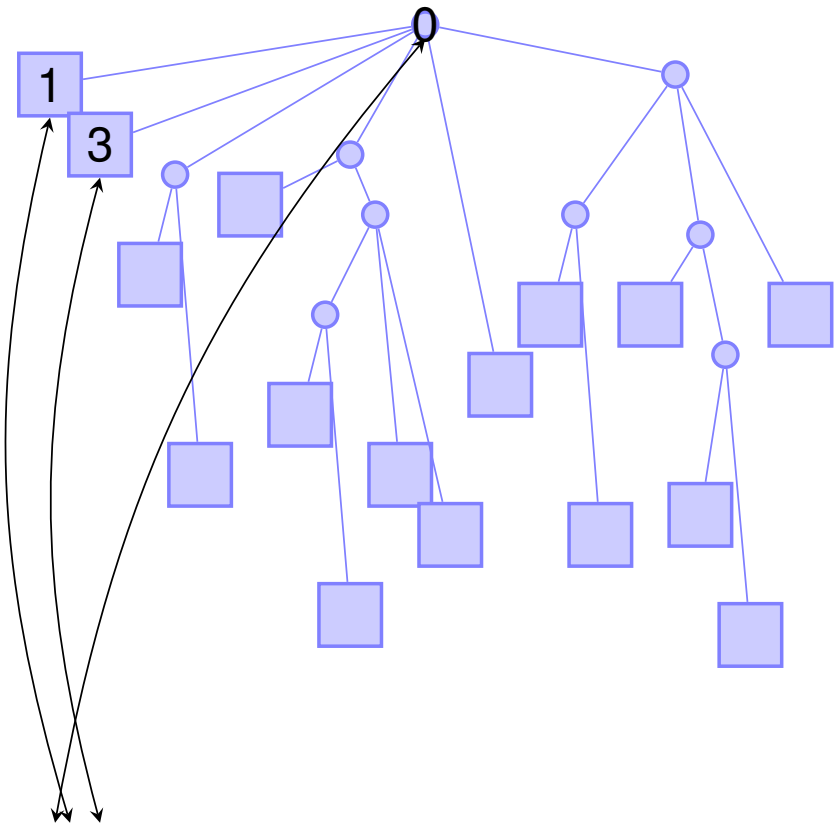
BPS_{dfs}= (()(()())(()(()())()())()(()())(()(()))))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



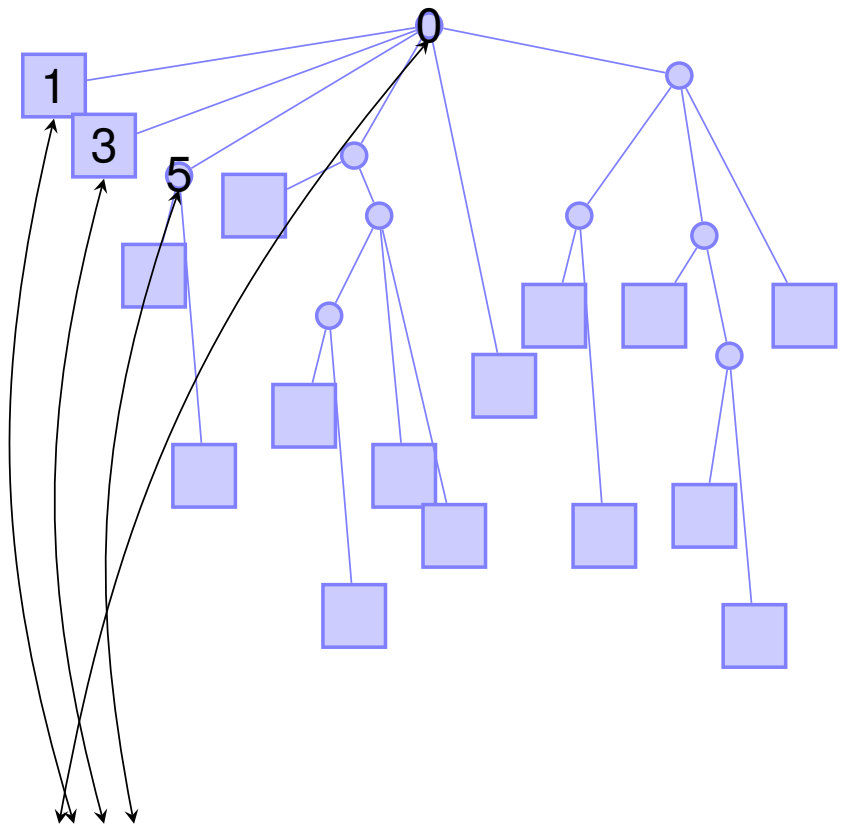
BPS_{dfs}= (()()(()())(()(()())()))()(()())(()(()()))())

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



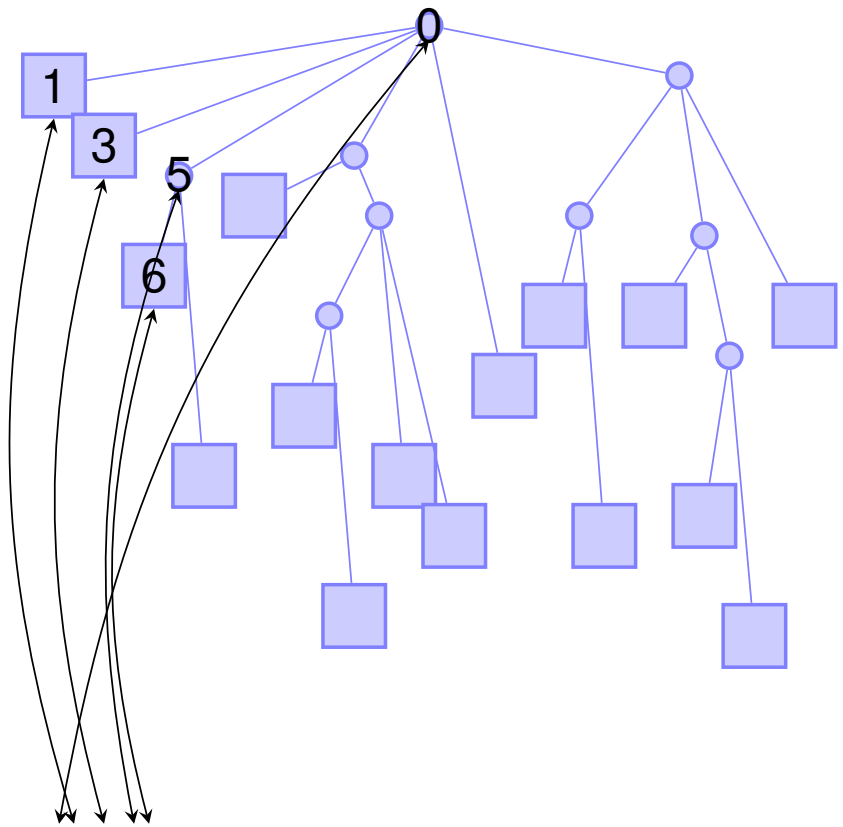
BPS_{dfs}= (()()(()())(()(()())()))(()())(()(()()))())

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



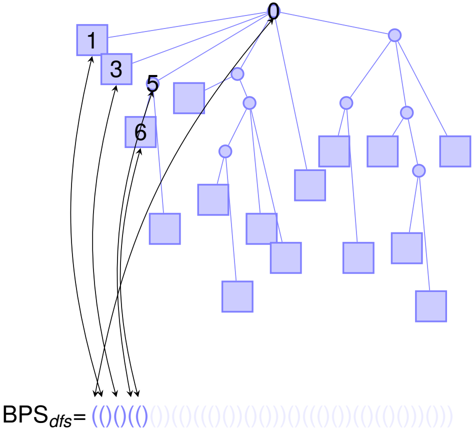
BPS_{dfs}= ((()(((()))(((()))))()))(((()))(((()))))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees

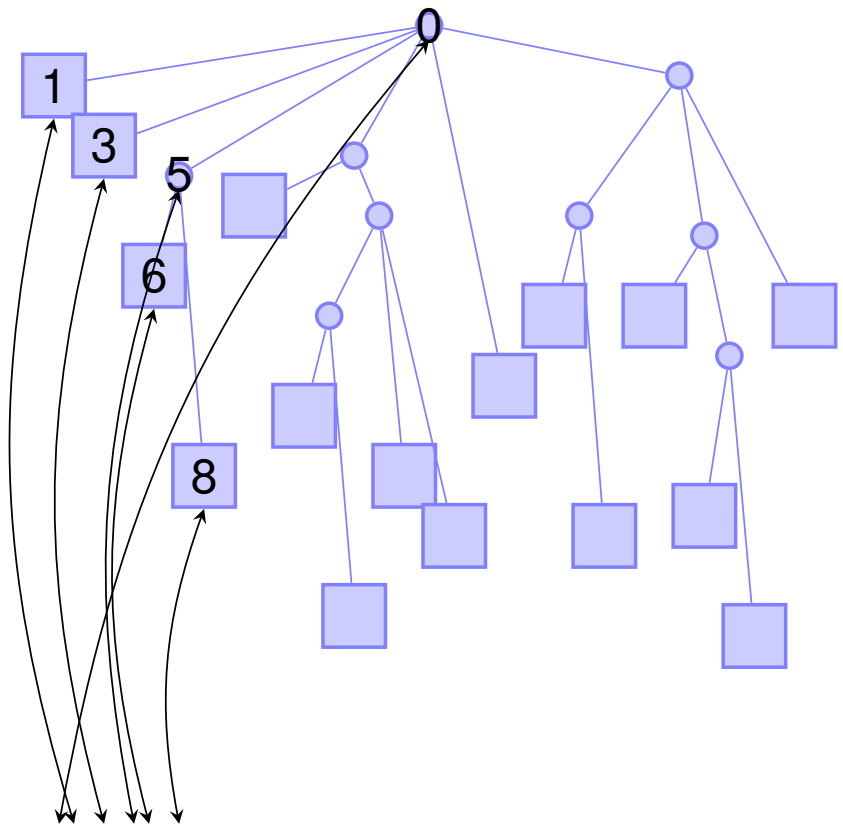


tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



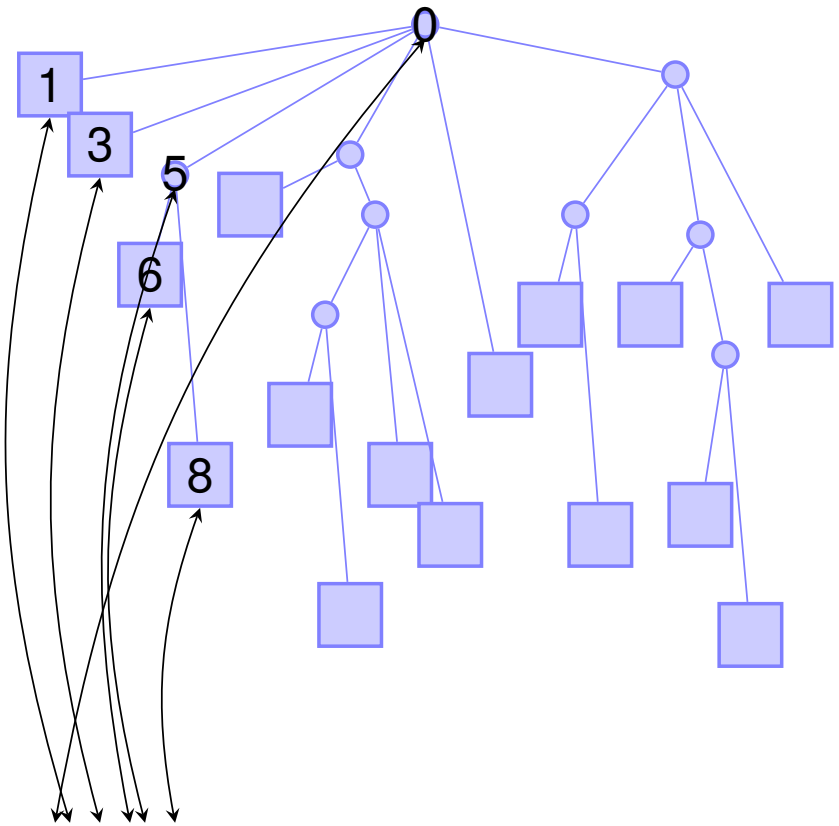
BPS_{dfs}= (()()(()()(()()))()(()()))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



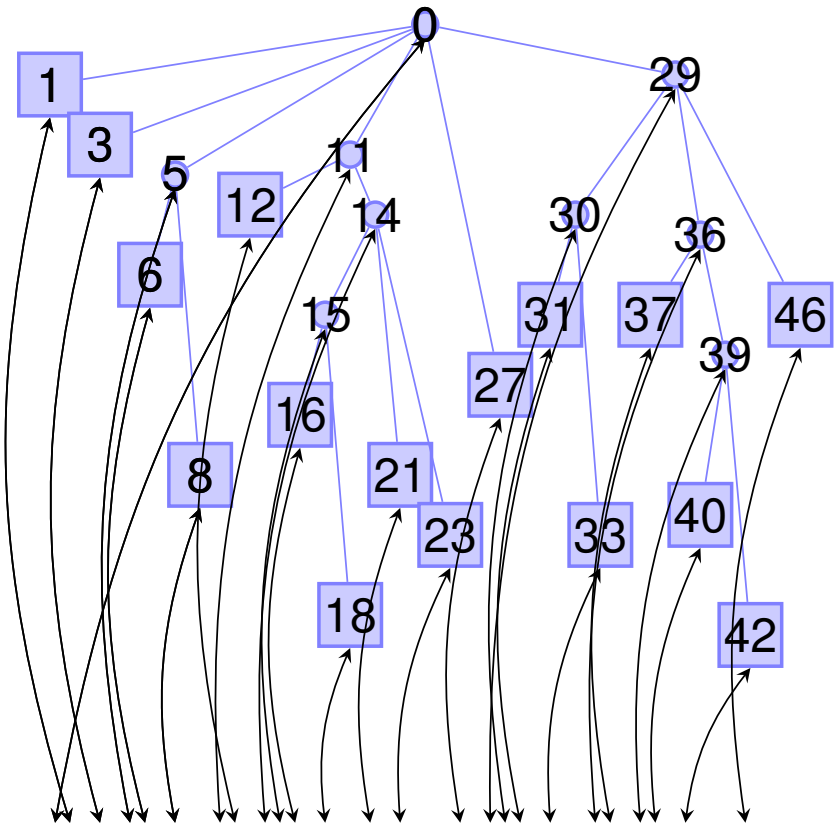
BPS_{dfs}= (()()(()()((((()())()())()((()())(()(()())))))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

Application: Succinct representation of trees



BPS_{dfs}= ((()(()())(()(()())()())()(()())(()(()())())))

tree uncompressed
$\mathcal{O}(n \log n)$ bits

compressed
$4n$ bits

Balanced Parentheses Sequences (BPS)

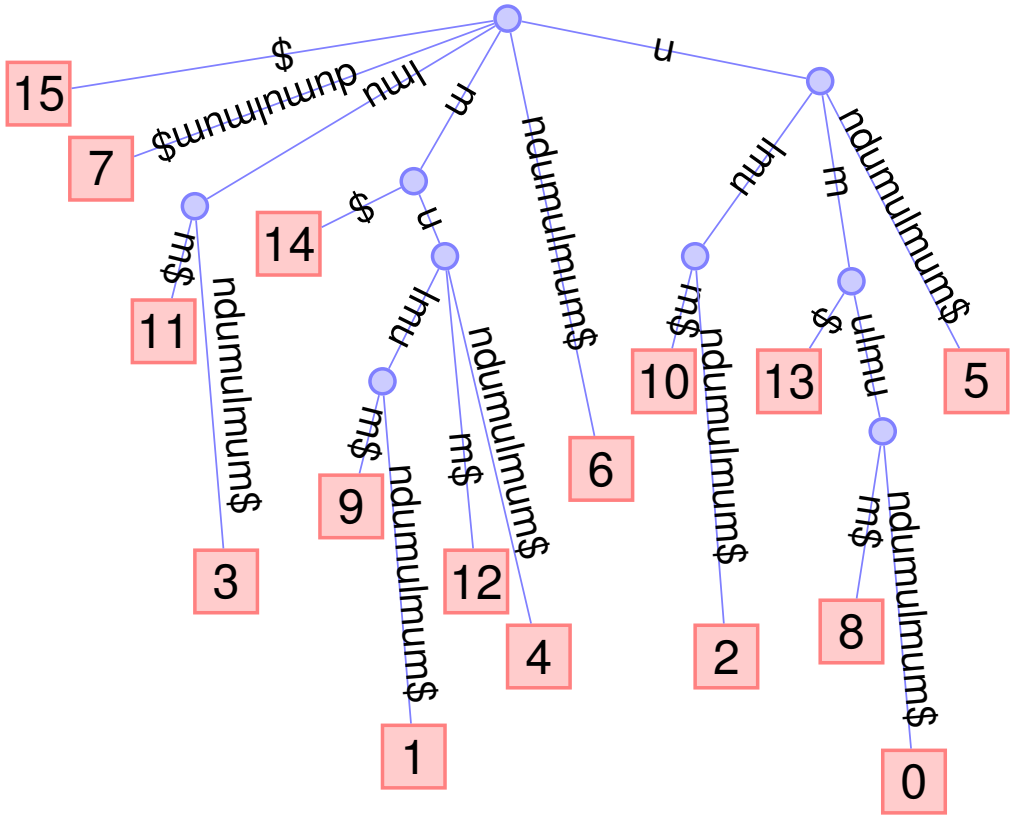
Application: Range Minimum Queries

- Array X of n elements from a well-ordered set.
- `rmq(i,j)` returns position of minimum element in $X[i,j]$.
- $2n + o(n)$ bits and $\mathcal{O}(1)$ time solution (Fischer & Heun 2011)

$X =$	1	4	6	2	99	10	5	13	2	74	32
	0	1	2	3	4	5	6	7	8	9	10

rmq(2, 6)=3

Compressed Suffix Trees (CST)



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5
0	0	0	3	0	1	5	2	2	0	0	4	1	2	6	1

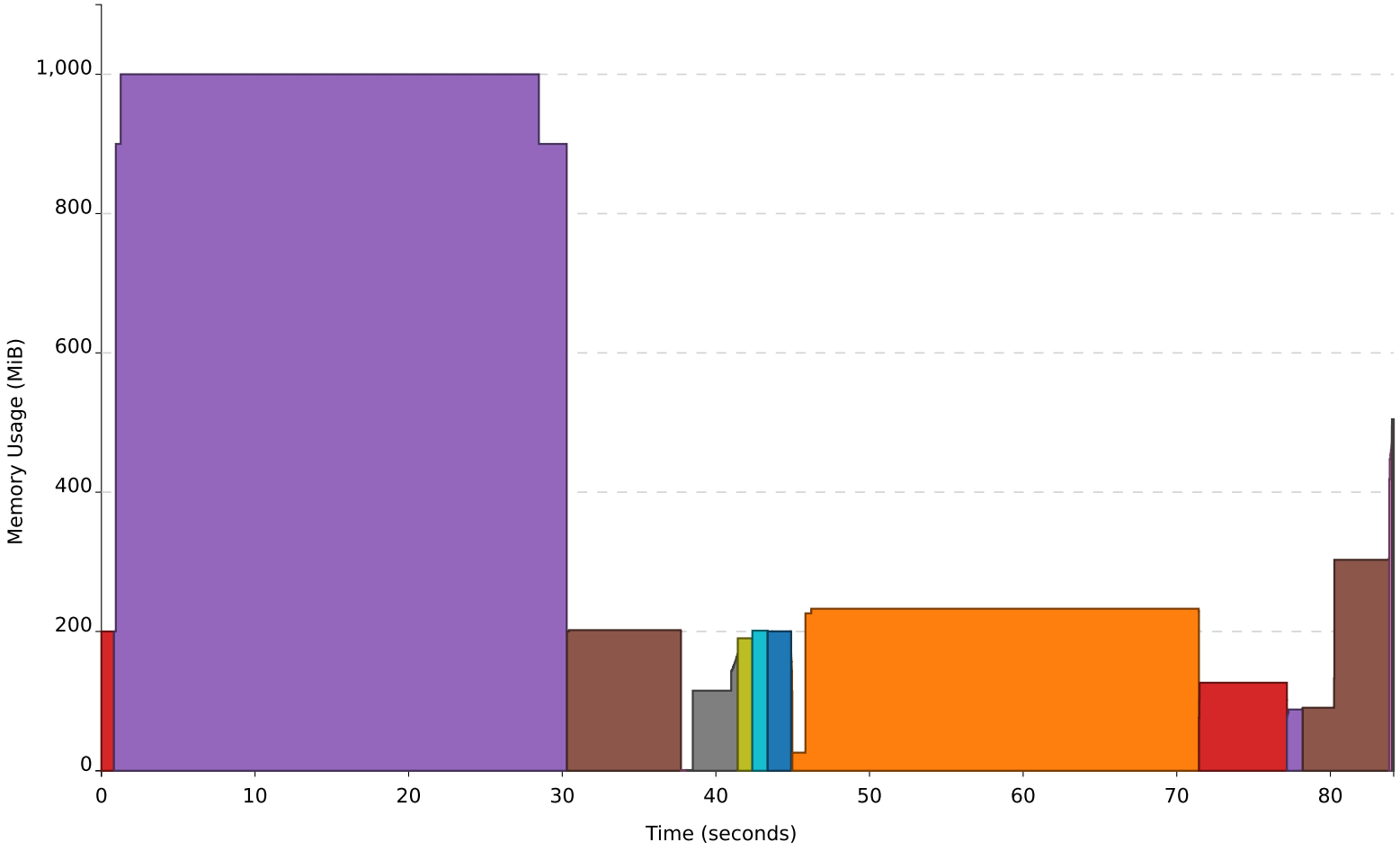
(()()(()())(()((()())()()))()(((())(()(()())))()))

Full functionality...

- $root()$
- $is\_leaf(v)$
- $parent(v)$
- $degree(v)$
- $child(v, c)$
- $select\_child(v, i)$
- $sibling(v)$
- $depth(v)$
- $lca(v, w)$
- $sl(v)$, $wl(v, c)$
- $id(v)$, $inv\_id(i)$

Size: $|CSA| + |LCP| + |BPS|$, i.e. about original text size

Compressed Suffix Trees (CST) – Construction



Resource diagram for the construction of a CST for 200MB of English text.

Outline

The Succinct Data Structure Library *SDSL*

https://github.com/simongog/sdsl-lite

- Input from about 40 research papers
- Fast and space-efficient construction
- Index structures available for byte and integer alphabets
- Well-optimized (e.g. hardware POPCOUNT, hugepages,...)
- Easy configuration of myriads of CSAs, CSTs, WTs
- Fast prototyping
- Tests, benchmarks, tutorial, cheat sheet

Outline

Conclusion

Limitations/Challenges of Succinct Structures:

- Often only work for a static setting
- Memory access pattern often non-local
- Implementation from scratch difficult
- Construction is often the bottleneck
- Algorithms have to be adapted to work fast with succinct structures

Conclusion

Advantages of Succinct Structures:

- Allow indexation of larger data sets (about 10-100 larger compared to classical indexes).
- Adapt to the data: E.g. capture repetitions.
- Often provide more functionality.
- More functionality often produces conceptional easier solutions.
- Not limited to text indexing.

Conclusion

Many applications heavily depend on Indexes. Self-Indexes made it possible to process Next Generation Sequencing in Genomics and Life Sciences.

Why?

Only the speed of self indexes matches the amazing throughput of sequencing technologies.

Thank you for your attention.
Thank MASTODONS