Compact Data Structures

A Theory-Oriented Survey

Gonzalo Navarro
www.dcc.uchile.cl/gnavarro
gnavarro@dcc.uchile.cl

Department of Computer Science
University of Chile

Compact Data Structures

They are data structures modified to use little space

- *Isn't that compression?*
- No: they must retain their functionality and direct access.
- *What for, if memory is so cheap?*
- Because they improve performance thanks to memory hierarchy.
  - Especially if we can operate in RAM a structure that otherwise would need the disk.
- Because they reduce the energy cost
  - In data centers, they require fewer servers running.
  - Larger memories consume more energy.
- Because they allow us handling larger collections.
  - On mobile and low-end devices, which have limited memory.

Compact Data Structures

A motivating example...

- The human genome, decoded around 2000.
- It contains 3 billion bases.
- Each base needs 2 bits (letters A, C, G, T).
- It holds comfortably in 1 GB RAM.
- *But bioinformatics need to perform complex searches on it!*
- Those operations are extremely slow in sequential form...
  - for example, finding the longest repeated string requires quadratic time without an appropriate index;
  - with the right index it is easily done in linear time.

Compact Data Structures

- The index solving all those problems is the suffix tree.
- But it requires 30 GB to 60 GB of memory!
- Even worse, it does not perform well on disk.
- In practice, it can only be used on toy sequences, which could indeed be handled sequentially.
- Using compact data structures, it fits in a RAM of 2 GB.
- It is much slower than the classic suffix tree on the same memory...
- but it is orders of magnitude faster running on RAM than the original one running on disk.

Worst-Case Entropy

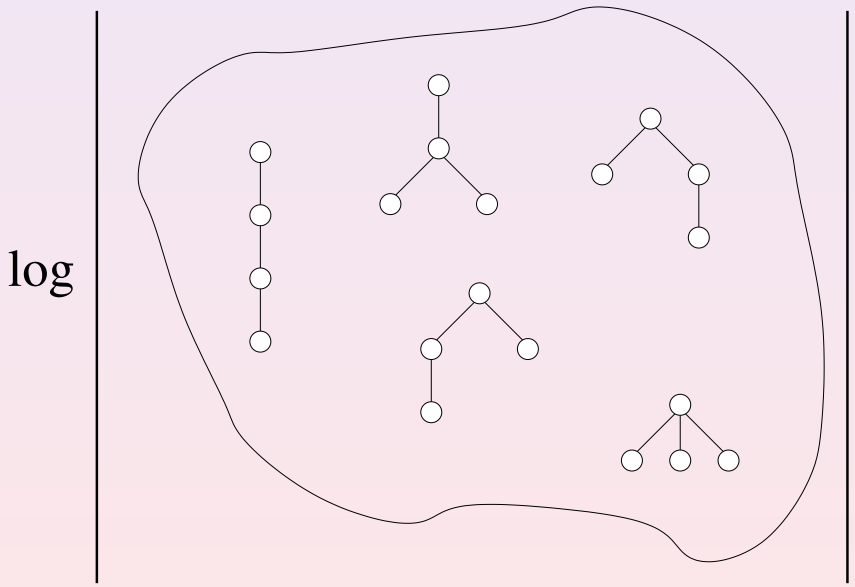
- Our gold standard will be the worst-case entropy.
- How many bits are needed to distinguish a combinatorial object from its family?
- It is always the (binary) logarithm of the family size.
- Example: There are 2^n bit sequences of length n, so the entropy is $\log 2^n = n$ bits.
- Example: There are σ^n sequences of length n over alphabet $[1, \sigma]$, so the entropy is $\log \sigma^n = n \log \sigma$ bits.

Worst-Case Entropy

- Example: There are

$$C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1} = 4^n/n^{3/2} \cdot O(1)$$

ordinal trees of n nodes, so the entropy is $2n - \Theta(\log n)$ bits.



- The entropy of trees shows that classical representations waste a lot of space, $O(n \log n)$ bits.

Empirical Entropy

- But bits and sequences seem to be optimally represented. How can they be compressed?
- When they belong to smaller, interesting subfamilies.
- Example: There are $\binom{n}{m}$ bit sequences of length n with m 1s, so the entropy is

$$\begin{aligned} \log \binom{n}{m} &= m \log \frac{n}{m} + (n-m) \log \frac{n}{n-m} - \Theta(\log n) \\ &= m \log \frac{n}{m} + O(m) \end{aligned}$$

  bits.
- This is related to the well known binary entropy function

$$H(p) = -p \log p - (1-p) \log(1-p).$$

- Our entropy is essentially $n \cdot H(m/n)$.

Empirical Entropy

- Example: There are $\binom{n}{n_1,\ldots,n_\sigma}$ sequences of length n over alphabet $[1,\sigma]$ with symbol frequencies $n_1,\ldots,n_\sigma$, so the entropy is

$$\log\binom{n}{n_1,\ldots,n_\sigma} = \sum_{c\in[1,\sigma]} n_c \log\frac{n}{n_c} - \Theta(\sigma\log n)$$

- This is also essentially $n\cdot H$, where H is the Shannon entropy of a zero-order (i.e., memoryless) source with symbol probabilities $p_c = n_c/n$.
- This is why the above entropies H are called the empirical entropy $H_0(S)$ of a sequence S with those symbol frequencies.
- This is a lower bound to zero-order compressors.

Bitvectors

- Consider a bit sequence $B[1, n]$.
- We are interested in the following operations on B:
  - $rank_b(B, i)$: how many times does b occur in $B[1, i]$?
  - $select_b(B, j)$: where is the j-th occurrence of b in B?
- Note $rank_0(B, i) = i - rank_1(B, i)$.
- By default let us assume $b = 1$.

Bitvectors

Result

- *rank* can be answered in constant time.
- This is easy by storing all the answers in $O(n \log n)$ bits, but one only needs

$$n + O\left(\frac{n \cdot \log\log n}{\log n}\right) \quad = \quad n + o(n)$$

  bits of space (the n bits for $B[1, n]$ plus a sublinear extra).
- This extra space on top of n is optimal to obtain constant time.

Rank

Rank in Constant Time

- We cut B into blocks of $b = (\log n)/2$ bits.
- We store and array $R[1, n/b]$ with the *rank* values at block beginnings.
- As we need $\log n$ bits to store a *rank* value, the total of bits used by R is

$$\frac{n}{b} \cdot \log n = \frac{n}{(\log n)/2} \cdot \log n = 2n$$

- Let us accept that space for now (it is more than promised).

Rank

- ¿How to compute $rank(B, i)$?
  - Decompose $i = q \cdot b + r$, $0 \leq r < b$.
  - The number of 1's up to qb is $R[q]$.
  - We must count the 1's in $B[qb + 1, qb + r]$.
- Assume we have a table $T[0, 2^b - 1][0, b - 1]$, such that

$$T[x, r] \quad = \quad \text{total of 1's in } x[1, r]$$

where we see x as a stream of b bits.

Rank

- Since each cell of T can take values in $[0, b-1]$, T needs

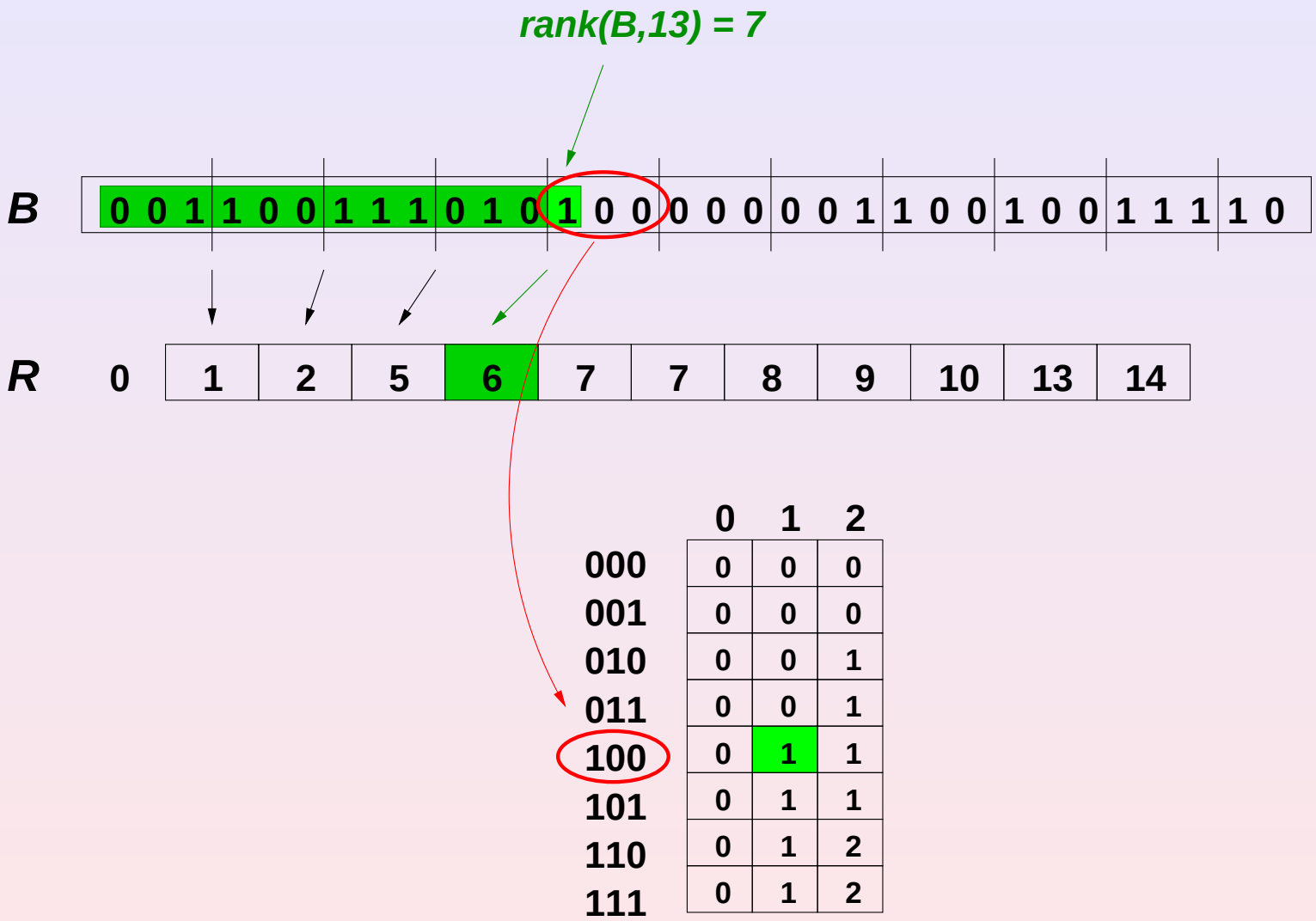
$$\begin{aligned} 2^b \cdot b \cdot \log b &= 2^{\frac{\log n}{2}} \cdot \frac{\log n}{2} \cdot (\log\log n - 1) \\ &\leq \frac{1}{2}\sqrt{n}\log n \log\log n = o(n) \text{ bits.} \end{aligned}$$

- Then, the final answer is obtained in constant time as:

$$R[q] + T[B[qb+1..qb+b], r]$$

- Good, but we spent $3n + o(n)$ bits...

Rank


rank(B,13) = 7
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0
R
0 1 2 5 6 7 7 8 9 10 13 14
0 1 2
000 0 0 0
001 0 0 0
010 0 0 1
011 0 0 1
100 0 1 1
101 0 1 1
110 0 1 2
111 0 1 2

Rank

Achieving $o(n)$ extra bits

- We also cut B into superblocks of

$$s = \frac{(\log n)^2}{2} = b \cdot \log n \text{ bits.}$$

- We store an array $S[1, n/s]$ with the *rank* at the beginning of superblocks.
- Now the values of R are stored adding just from the beginning of the corresponding superblock:

$$\begin{aligned} R[q] &= rank(B, qb) - S[\lfloor q/\log n \rfloor] \\ &= rank(B, qb) - rank(B, \lfloor q/\log n \rfloor \cdot \log n) \end{aligned}$$

Rank

- As we need $\log n$ bits to store a *rank* value in S, the total number of bits used by S is

$$\frac{n}{s} \cdot \log n = \frac{n}{(\log n)^2/2} \cdot \log n = \frac{2n}{\log n} = o(n)$$

- As the R values are stored relative to the superblock, they can be only as large as $s = O(\log n)^2$, and thus only need $2 \log \log n$ bits. Thus R now occupies

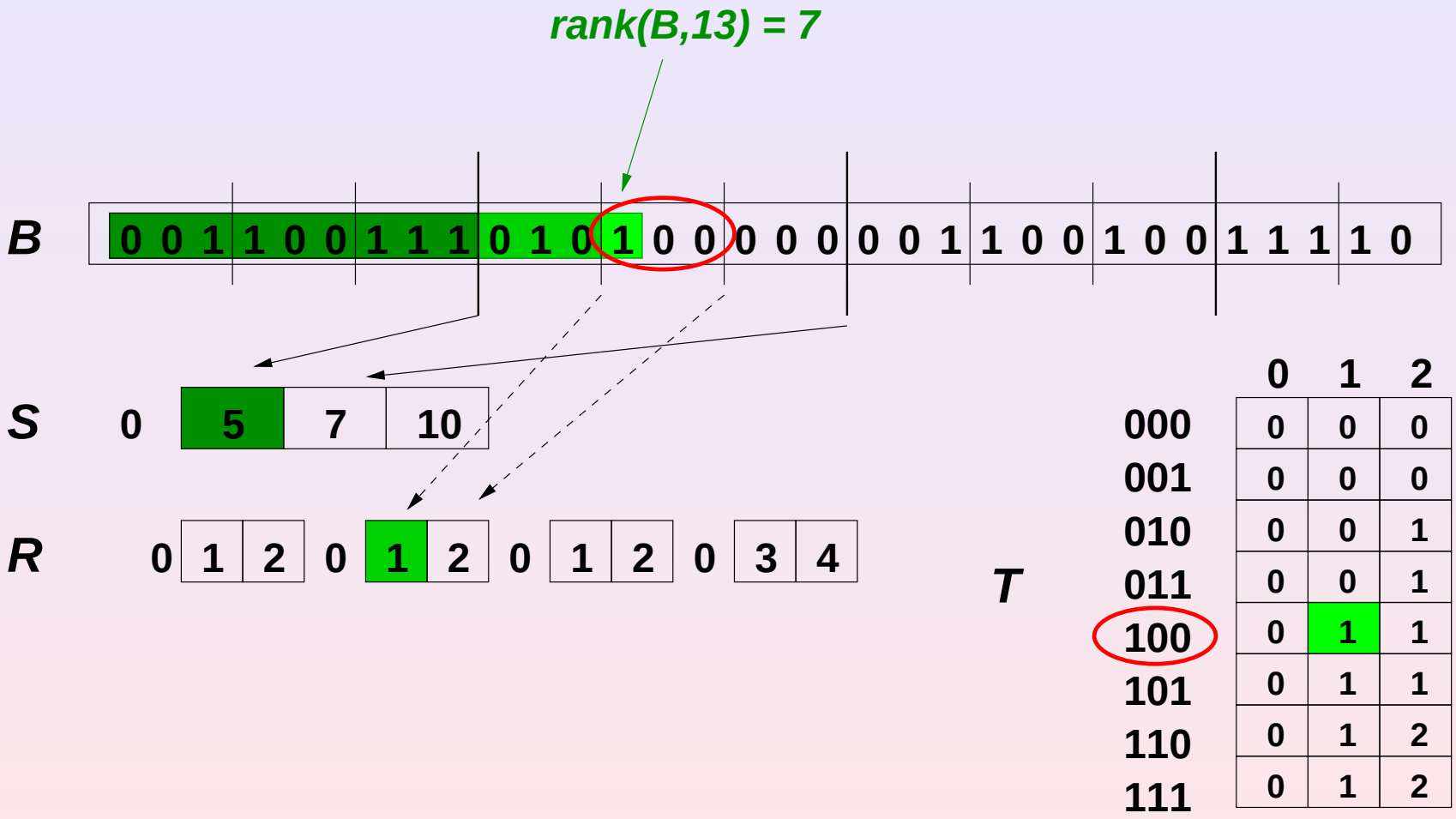
$$\frac{n}{b} \cdot 2 \log \log n = \frac{n}{(\log n)/2} \cdot 2 \log \log n$$
$$= \frac{4n \cdot \log \log n}{\log n} = o(n) \text{ bits.}$$

Rank

- ¿How we compute $rank(B, i)$?
  - Decompose $i = q \cdot b + r$, $0 \leq r < b$.
  - Decompose $i = q' \cdot s + r'$, $0 \leq r' < s$.
  - Add the contents of three tables:

$$rank(B, i) \quad = \quad S[q'] + R[q] + T[B[qb + 1..qb + b], r]$$

Rank


rank(B,13) = 7
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0
S
0
5
7
10
R
0 1 2 0 1 2 0 1 2 0 3 4
T
0 1 2
000 0 0 0
001 0 0 0
010 0 0 1
011 0 0 1
100 0 1 1
101 0 1 1
110 0 1 2
111 0 1 2

Rank on Compressed Sequences

- In many applications, the sequence $B[1, n]$ contains few or many 1's.
- Let us focus on the case where there are $m << n$ 1's.
- Binary entropy:

$$H_0(B) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$$

- We can have constant-time *rank* using just $nH_0(B) + o(n)$ bits.

Compressed Representation

- Divide the sequence into blocks of $b = (\log n)/2$ bits.
- Let c_i be the number of 1's in a block B_i.
- The number of distinct blocks of length b with c_i 1's is

$$\binom{b}{c_i}$$

and hence we will try to represent B_i using

$$\left\lceil \log \binom{b}{c_i} \right\rceil \text{ bits.}$$

Compressed Representation

- The representation of B_i will be (c_i, o_i), where
  - The class, c_i, needs $\lceil \log(b+1) \rceil$ bits.
  - The offset, o_i, needs $\lceil \log \binom{b}{c_i} \rceil$ bits.
- The c_i's are of fixed width, and take in total

$$\frac{n}{b} \cdot \lceil \log(b+1) \rceil \quad = \quad O\left(\frac{n \cdot \log \log n}{\log n} \right)$$

- The o_i's are of variable width. We will concatenate them all.
- Later we see how to decode them.

Compressed Representation

- The concatenation of the o_i's takes

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

Compressed Representation

- We have represented B using $nH_0(B) + o(n)$ bits!
- But how can we extract a block B_i in constant time?
- First problem: where is o_i?
  - We can store the positions in $P[1, n/b]$...
  - ... but this requires $(n/b) \log n = 2n$ extra bits.
  - We define superblocks of $s = b \cdot \log n$ bits...
  - ... and store $P[1, n/s]$ only for superblocks.
  - We will have $P'[1, n/b]$ with pointers relative to the superblock (like in *rank*).
  - Since $|o_i| = O(\log n)$, each $P'[i]$ needs $O(\log \log n)$ bits.
  - Total: $O(n \log \log n / \log n)$ bits.

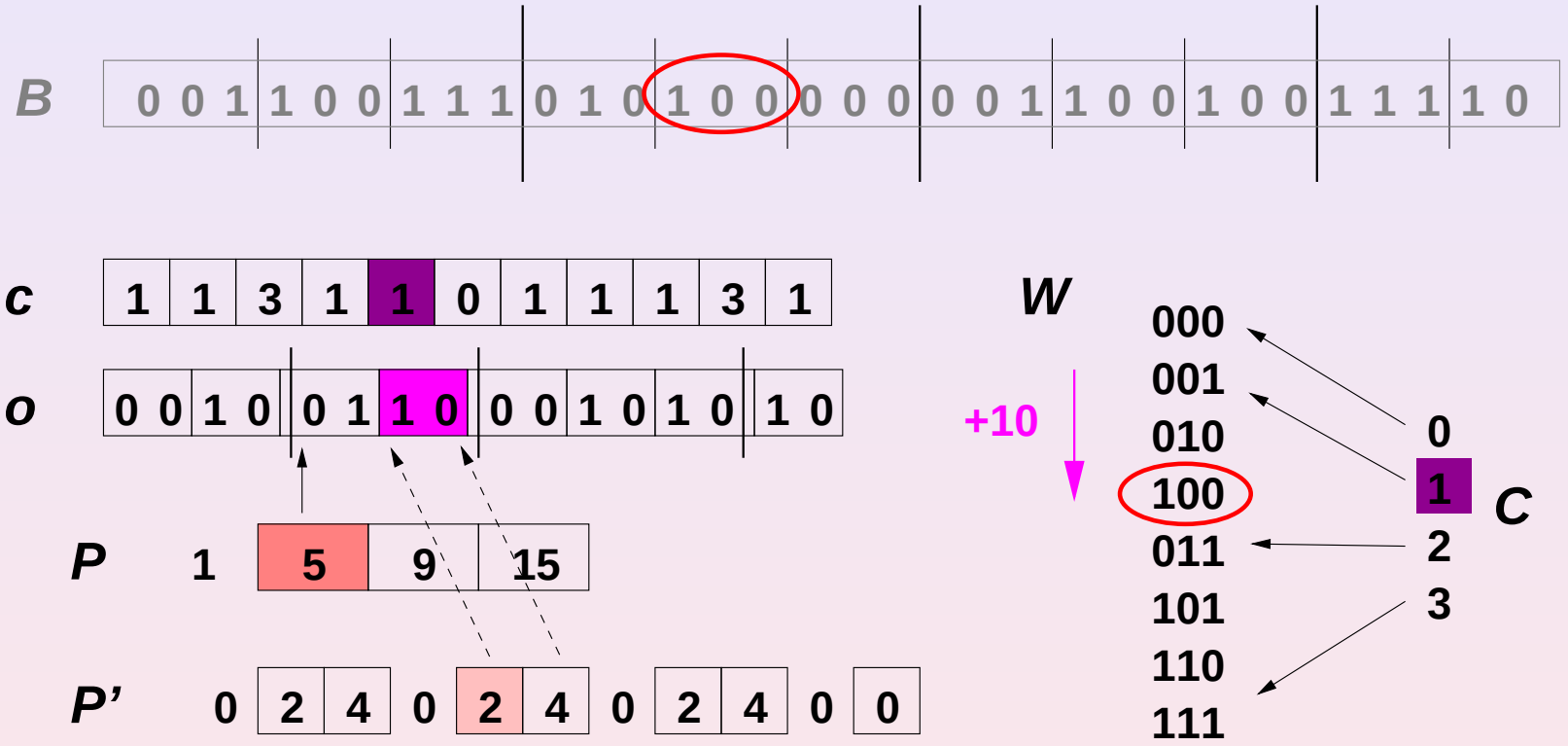
Compressed Representation

- Second problem: which block does (c_i, o_i) represent?
  - We will have a table $W[0, 2^b - 1]$ where all bitvectors of b bits are stored grouped by class.
  - The positions where each class begins in W will be precomputed in an array

$$C[c] = 1 + \sum_{i=0}^{c-1} \binom{b}{i}$$

  - W is sorted so that o_i is the corresponding index within the zone of class c_i.
  - Hence (c_i, o_i) represents $W[C[c_i] + o_i]$.
  - These tables occupy $O(\sqrt{n} \log n)$ bits.

Compressed Representation



Rank

- The structures for *rank* occupy $o(n)$ bits on top of B.
- They need constant-time access to blocks of B.
- This is what we have obtained with the compressed representation!
- Hence we have obtained *rank* in constant time over the compressed representation (to H_0).
- More precisely, we use $nH_0(B) + O(n \log \log n / \log n)$ bits.
- Again, the $O(n \log \log n / \log n)$ bits are necessary to obtain constant-time *rank*.

Select

- The solution for *select* is more complicated.
- First, we can always do in $O(\log n)$ time by binary search on *rank*.
- We show a solution that uses $n + O(n/\log\log n)$ bits and solves *select* in constant time.
- It is possible to use $n + O(n \log\log n/\log n)$ bits, which is optimal space, and have constant time, but the solution is too complicated.
- It is possible to combine with bitvector compression to obtain $nH_0(B) + o(n)$ bits in total.
- We will show the solution for $select_1$; the one for $select_0$ is analogous (and needs a copy of the extra structures).

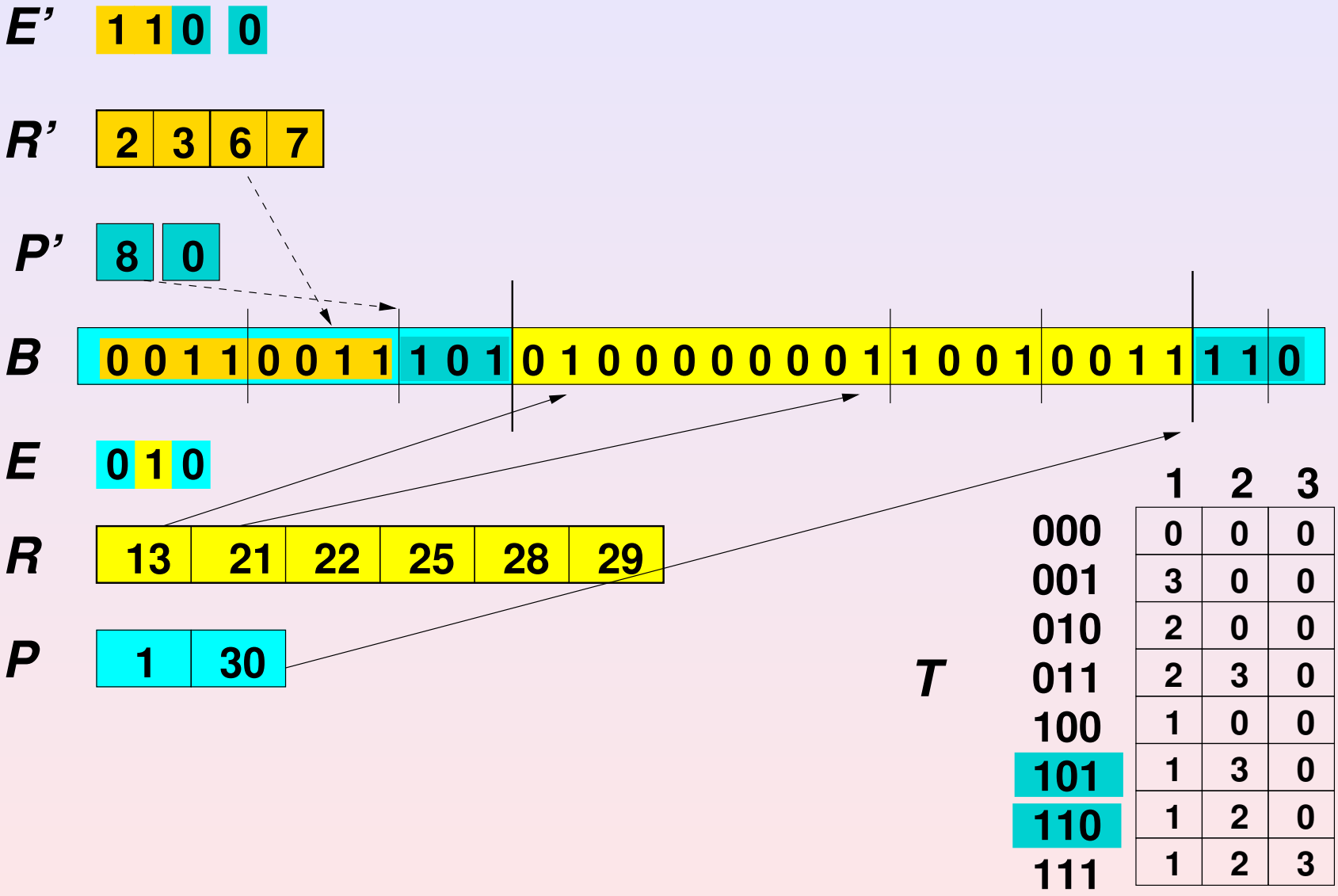
Select

- We regularly sample the arguments to *select*.
- Let $s = \log^2 n$; we store $S[1, n/s]$ with $S[i] = select(B, i \cdot s)$.
- Array S requires only $O(n/\log n)$ bits.
- Given $select(B, j)$, we find the superblock $i = j/s$ where the query lies.
- There are two cases:
  - If $S[i+1] - S[i] > \log^4 n$, then the superblock is long, and we have all its s answers precalculated. Each long block takes $s \log n = log^3 n$ bits, but there are at most $n/\log^4 n$ long superblocks, so in total we spend $O(n/\log n)$ bits.
  - Otherwise, the superblock is short. Since it is of length at most $\log^4 n$, we could binary search it in $O(\log \log n)$ time.

Select

- To reach constant time, we repeat the process inside short superblocks.
- Let $b = (\log \log n)^2$; we store the starting positions of blocks of b arguments in array S'.
- Array $S'x$ requires only $O(n/\log \log n)$ bits.
- Blocks are long if their length is at least $(\log \log n)^4$.
- Long blocks store all their answers in $O(n/\log \log n)$ bits.
- Short blocks are of length at most $(\log \log n)^4 = o(\log n)$, and thus can be scanned in constant time using universal tables like the table T used for *rank*.

Select


E'
1 1 0 0
R'
2 3 6 7
P'
8 0
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 0
E
0 1 0
R
13 21 22 25 28 29
P
1 30
T
1 2 3
000 0 0 0
001 3 0 0
010 2 0 0
011 2 3 0
100 1 0 0
101 1 3 0
110 1 2 0
111 1 2 3

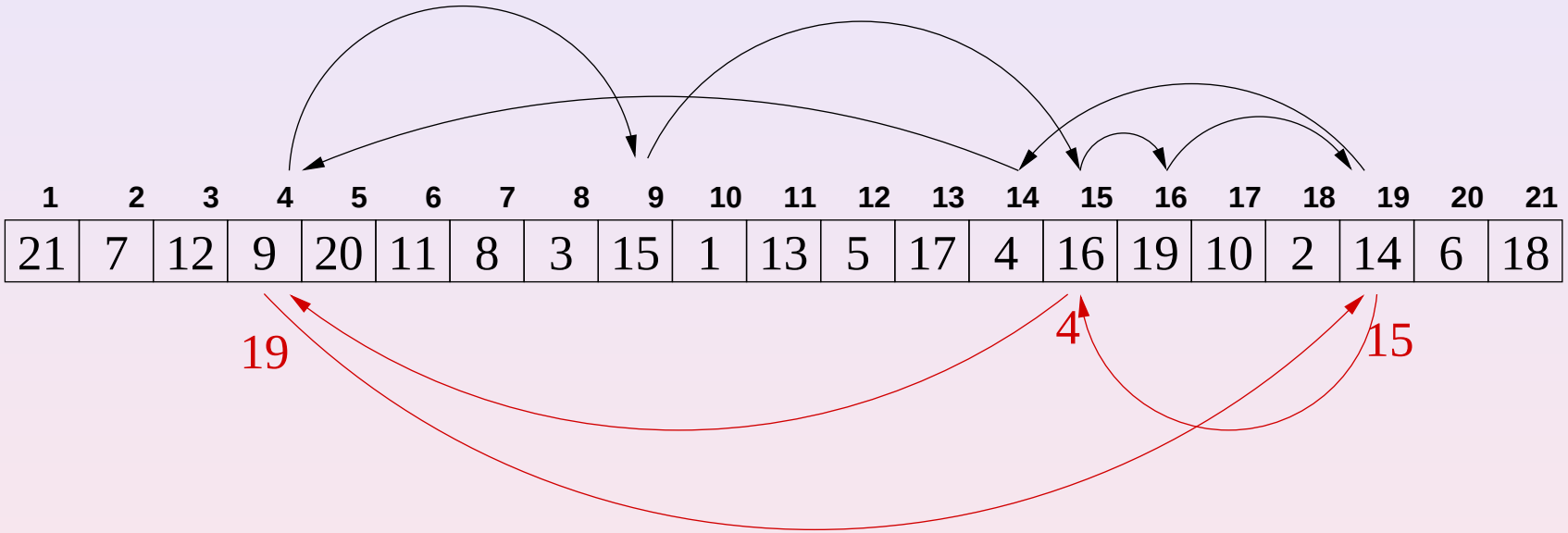
Permutations

- Let $\pi[1, n]$ be a permutation of $[1..n]$.
- There are $n!$ permutations, so we need $\log n! = n \log n - \Theta(n)$ bits to represent it.
- Thus an array representation is almost optimal.
- However, within this space we want to compute powers of the permutation: $\pi^k(i)$ for any positive or negative k.
- We will obtain $1/\epsilon$ time and $(1 + \epsilon)n \log n$ bits, for any ϵ.
- In particular, $\log n$ time with $n \log n + O(n)$ bits.
- This is close to the optimal, $O(\log n / \log \log n)$ time.

Inverse Permutation

- Assume we store π in an array.
- To find $j = \pi^{-1}(i)$, we compute $\pi(i)$, $\pi^2(i)$, $\pi^3(i)$, and so on, until $\pi^k(i) = i$. Thus $j = \pi^{k-1}(i)$.
- The time is $O(c)$, where c is the length of the cycle where i lies.
- To guarantee a certain time $O(t)$, we introduce shortcuts.
- These are backward arrows every t positions in the cycle.
- If we take the first shortcut we hit, the total time becomes $O(t)$.

Inverse Permutation



Inverse Permutation

- How to efficiently represent the shortcuts?
  - Mark in a bitvector $R[1, n]$ which positions have shortcuts.
  - Store their values compactly in $P[1, s]$, $s \leq n/t$.
  - If $R[i] = 1$, there is a shortcut in i, and its value is at $P[rank(R, i)]$.
- In total, one uses $n \log n + (n/t) \log n + n + o(n)$ bits.
- We solve $\pi(i)$ in $O(1)$ time and $\pi^{-1}(i)$ in $O(t)$.
- For example,
  - With $(1 + \epsilon)n \log n + O(n)$ bits, for constant ϵ, we solve π in time $O(1)$ and π^{-1} in time $O(1/\epsilon) = O(1)$.
  - With $n \log n + O(n \log \log n) = n \log n + o(n \log n)$ bits, we solve π in time $O(1)$ and π^{-1} in time $O(\log n / \log \log n)$.

Powers of Permutations

- And to solve π^k and π^{-k} ?
  - Imagine we write the cycles explicitly on a new permutation τ.
  - $\pi(i)$ follows i in S; π^{-1} precedes i.
  - We mark in a bitvector $C[1, n]$ where the cycles start.
  - Represent τ with the previous technique.
  - This is sufficient to compute any π^k and π^{-k} in time $O(t)$.

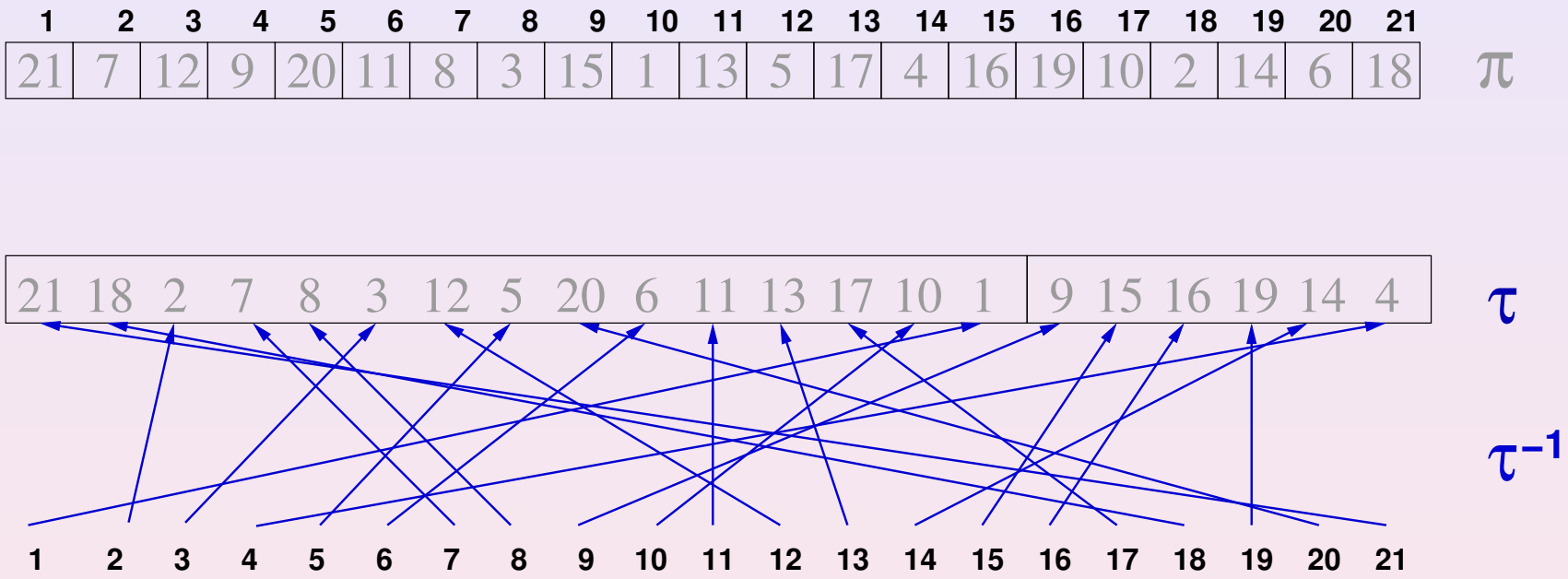
Powers of Permutations

- To compute $\pi^k(i)$ (k positive or negative):
  - With $i' = \tau^{-1}(i)$ we find i in τ.
  - Compute the limits of the cycle:
    - $l = select(C, rank(C, l))$
    - $r = select(C, rank(C, l) + 1)$
  - The position corresponding to $\pi^k(i)$ is

$$j' = l + (i' + k - l \mod (r - l))$$

  - Finally we return $\pi^k(i) = \tau(j')$
- It can be extended to general functions on $[1, n]$.

Powers of Permutations



Symbol Sequences

- Assume now we handle a sequence $S = s_1 s_2 \ldots s_n$ over an alphabet Σ of size σ.
- A plain representation of S needs $n \log \sigma$ bits.
- We wish to solve *rank* and *select* over this sequence:
  - $rank_c(S, i) =$ number of occurrences of c in $S[1, i]$.
  - $select_c(S, j) =$ position of the j-th occurrence of c in S.
- How can we extend our results on bits?

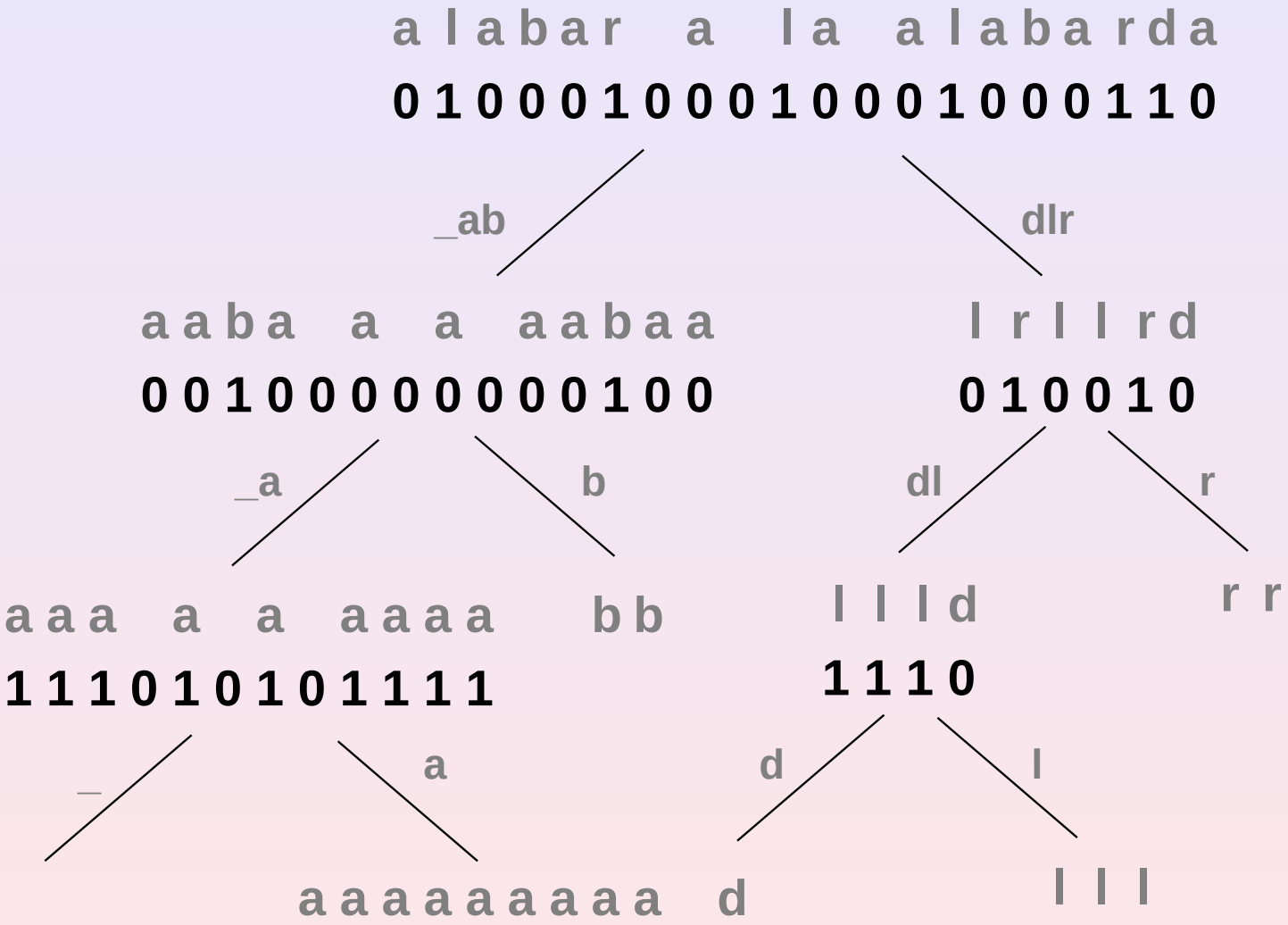
The Wavelet Tree

- Is an elegant solution that lets us store $S[1, n]$:
  - Using $n \log \sigma + o(n \log \sigma)$ bits.
  - Obtaining $S[i]$ in time $O(\log \sigma)$.
  - Solving *rank* in time $O(\log \sigma)$.
  - Solving *select* in time $O(\log \sigma)$.
- Has many other applications (geometry, binary relations, etc.)
- Trickier variants reduce the space to $n \log \sigma + o(n)$ bits and the time to $O(1 + \log \sigma / \log \log n)$, which is constant for polylog-sized alphabets.

The Wavelet Tree

- Assume we split the alphabet into two subsets of the same size (or almost).
- We create a bitvector indicating to which subset each symbol of S belongs to.
- We store that bitvector in the root of the wavelet tree.
- For the left/right subtree, we choose the symbols of S from each subset.
- We continue recursively until each subset has just one symbol.
- It is easy to see that all the bitvectors add up to $n \log \sigma$ bits.

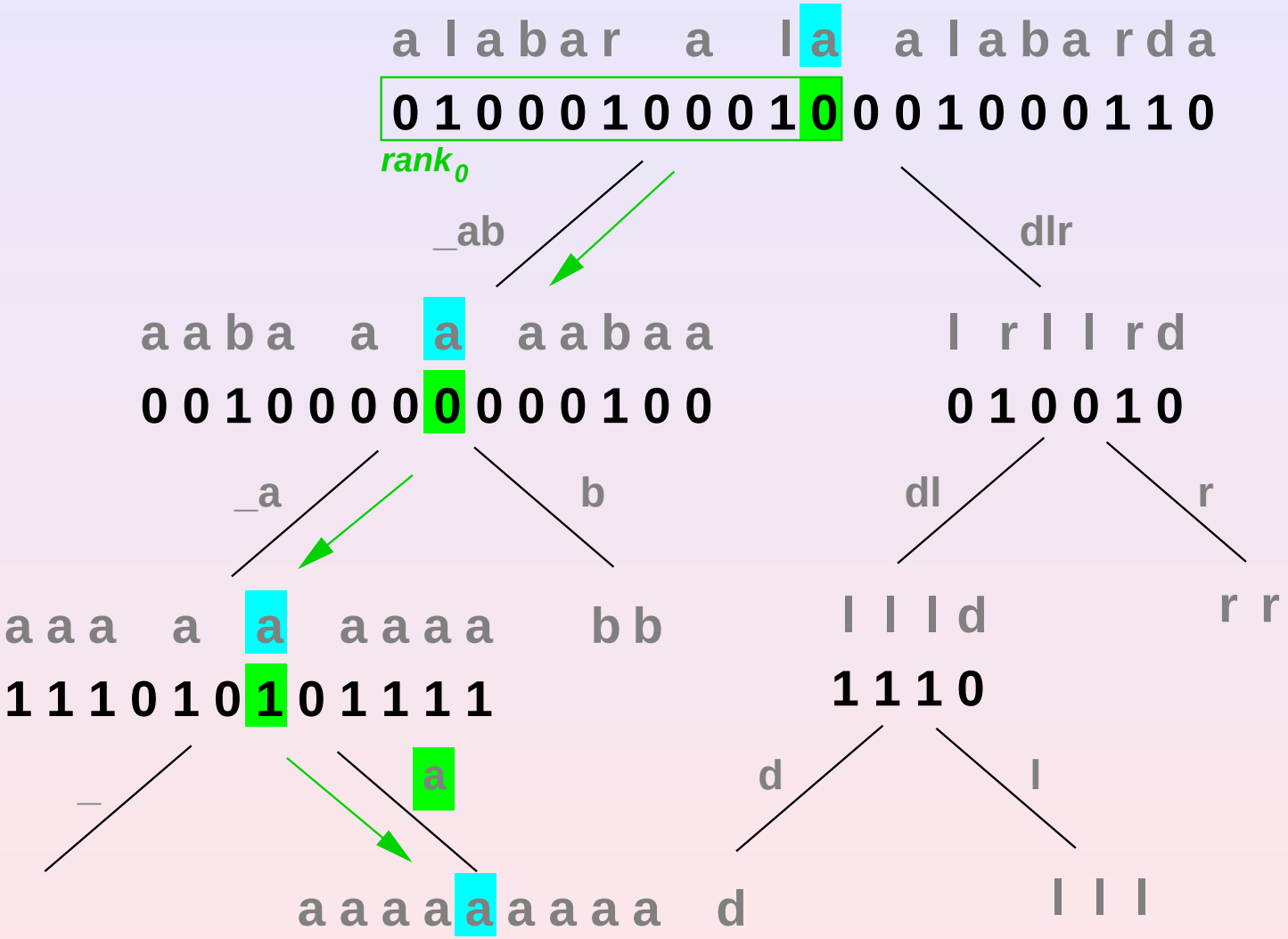
The Wavelet Tree



The Wavelet Tree

- How we retrieve symbol $S[i]$?
  - Look at $B[i]$ in the root bitvector.
  - If $B[i] = 0$,
    - Go to the left subtree.
    - The new position is $i' = rank_0(B, i)$.
  - If $B[i] = 1$,
    - Go to the right subtree.
    - The new position is $i' = rank_1(B, i)$.
  - When we arrive at a leaf, the corresponding symbol is $S[i]$.
- Time: $\log \sigma$ evaluations of *rank*.
- Need to preprocess the bitvectors for *rank* queries.

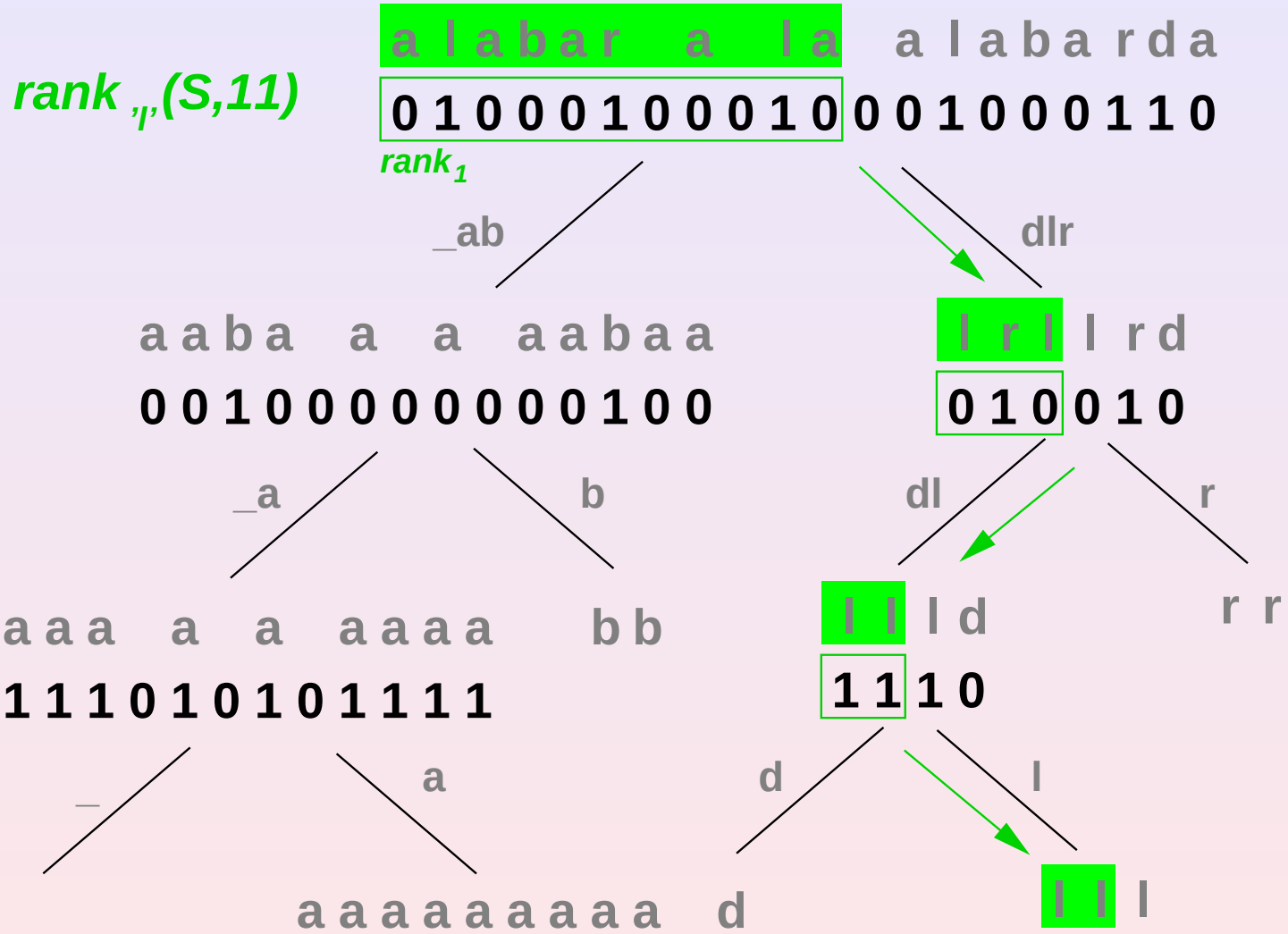
The Wavelet Tree



The Wavelet Tree

- How we compute $rank_c(S, i)$?
  - Let B be the root bitvector.
  - If c belongs to the left subtree,
    - Move to the left subtree.
    - The new position is $i' = rank_0(B, i)$.
  - If c belongs to the right subtree,
    - Move to the right subtree.
    - The new position is $i' = rank_1(B, i)$.
  - When we arrive at a leaf, the answer is i.
- Time: $\log \sigma$ evaluations of $rank$.

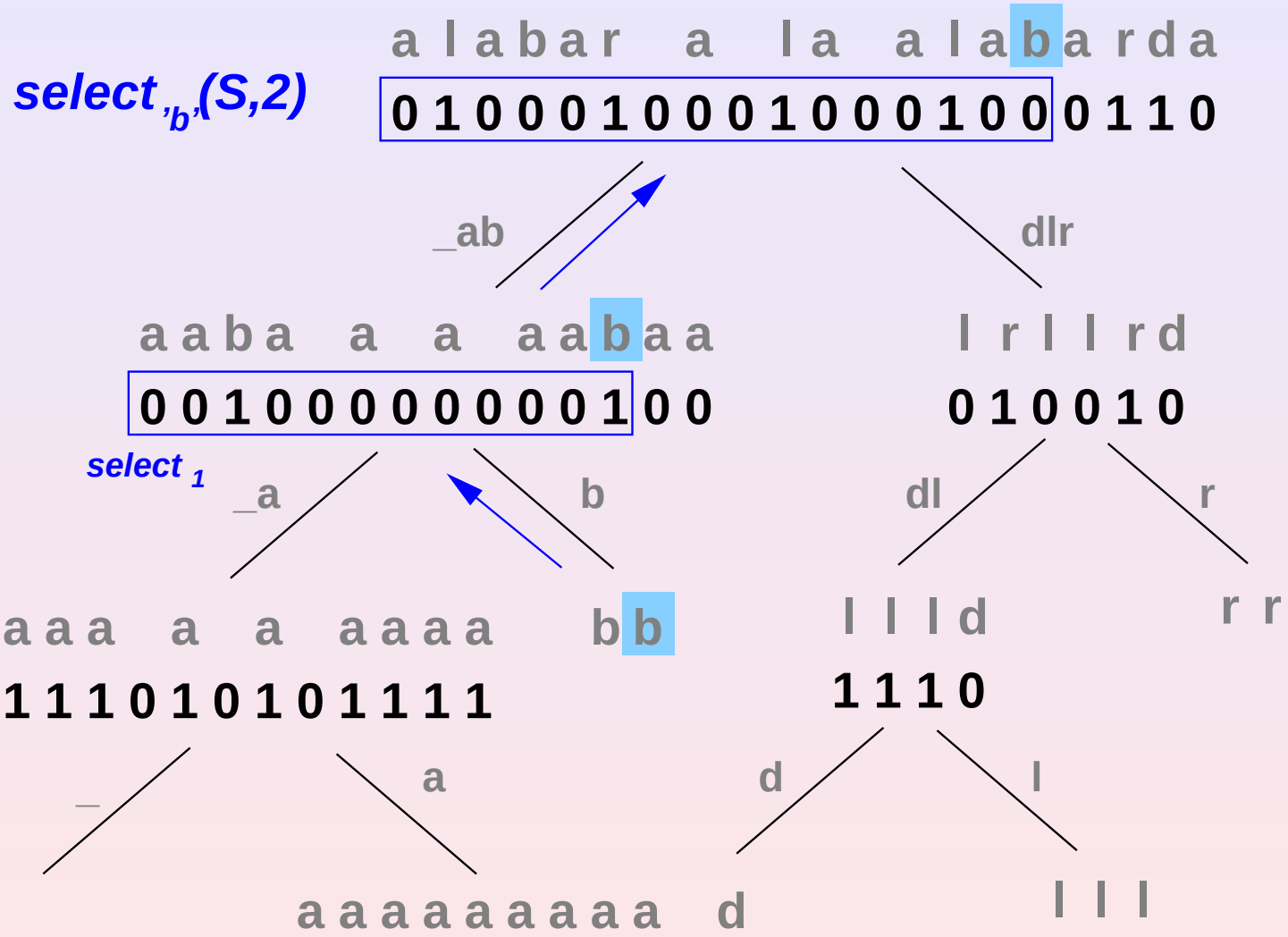
The Wavelet Tree



The Wavelet Tree

- How to compute $select_c(S, i)$?
  - We start at the leaf that corresponds to c.
  - Let B the bitvector of its parent.
  - If the leaf is a left child,
    - We move to the parent.
    - The new position is $i' = select_0(B, i)$.
  - If the leaf is a right child,
    - We move to the parent.
    - The new position is $i' = select_1(B, i)$.
  - When we reach the root, the answer is i.
- Time: $\log \sigma$ evaluations of *select*.
- We need to preprocess the bitvectors for *select* queries.

The Wavelet Tree



The Wavelet Tree

- If the bitvectors are uncompressed, the space is $n \log \sigma + o(n \log \sigma)$.
- We will see that, by compressing the bitvectors of the wavelet tree to zero-order, we compress the sequence to zero-order as well.
- Consider the successive levels of the wavelet tree.
- Let $B[1, n]$ be the highest bits of the symbols.
- Say B has n_0 0s and n_1 1s.
- It is compressed to

$$nH_0(B) = n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1}$$

bits.

The Wavelet Tree

- Consider now the subsequence $S_0[1, n_0]$ of the left child and B_0 the bitvector of its next highest bit.
- Let n_{00} and n_{01} be its number of 0s and 1s, respectively.
- B_0 is compressed to

$$n_0 H_0(B_0) = n_{00} \log \frac{n_0}{n_{00}} + n_{01} \log \frac{n_0}{n_{01}}$$

- Symmetrically with S_1. Adding up the space for the first two levels we get

$$n_{00} \log \frac{n}{n_{00}} + n_{01} \log \frac{n}{n_{01}} + n_{10} \log \frac{n}{n_{10}} + n_{11} \log \frac{n}{n_{11}}$$

- If continuing up to level $\log \sigma$, we get $nH_0(S)$.
- The sublinear parts add up to $o(n \log \sigma)$ further bits.

Based on Permutations

- An alternative sequence representation, based on permutations, offers better times on large alphabets.
- We will obtain time of the form $O(\log\log\sigma)$ for *rank* and *access*, and $O(1)$ for *select*.
- ... and $n\log\sigma + n \cdot o(\log\sigma)$ bits of space.
- The optimum is $O(\log \frac{\log\sigma}{\log\log n})$ time for *rank*, and any $\omega(1)$ for the product of *access* and *select*.
- Let us cut the sequence into blocks of σ symbols.
- We solve two separate subproblems.
  - Determine the answer up to the block granularity.
  - Refine the answer within the block.

Based on Permutations

Block granularity

- Consider the occurrences of a symbol c.
- We will set up a bitvector L_c that counts, in unary, the number of cs in each block:
  - Start with a 0.
  - Add a 1 each time c appears in the sequence.
  - Add a 0 when we move to the next block.
- Adding over all the L_c, there are n 1s and $\leq n$ 0s.

Based on Permutations

	a l a b a r	a	l a	a l a b a r d a
$B_{\_}$	0 0 0 0 0 0	1	0 1 0 0 1	0 0 0 0 0 0 0 0
B_{a}	1 0 1 0 1 0	0	1 0 0 1 0	1 0 1 0 1 0 0 1
B_{b}	0 0 0 1 0 0	0	0 0 0 0 0	0 0 0 1 0 0 0 0
B_{d}	0 0 0 0 0 0	0	0 0 0 0 0	0 0 0 0 0 0 1 0
B_{l}	0 1 0 0 0 0	0	0 0 1 0 0	0 1 0 0 0 0 0 0
B_{r}	0 0 0 0 0 1	0	0 0 0 0 0	0 0 0 0 0 1 0 0
$L_{\_}$	0 0 1 1 1 0 0			
L_{a}	0 1 1 1 0 1 1	0	1 1 1	0 1 0
L_{b}	0 1 0 0 1 0 0			
L_{d}	0 0 0 0 1 0			
L_{l}	0 1 0 1 0 1 0			
L_{r}	0 1 0 0 1 0 0			

Based on Permutations

- To compute $rank_c(S, i)$:
  - Compute $b = 1 + \lfloor i/\sigma \rfloor$.
  - Compute $r = rank_1(L_c, select_0(L_c, b))$.
  - b is the block where the complete the answer.
  - r is the number of occurrences of c before block b.
  - Within block b we must compute $rank_c(1 + (i \bmod \sigma))$.
- To compute $S[i]$:
  - Compute $b = 1 + \lfloor i/\sigma \rfloor$.
  - b is the block where to find the answer.
  - Within block b we must obtain the symbol at position $1 + (i \bmod \sigma)$.

Based on Permutations

- To compute $select(S, i) = select(B_c, i)$:
  - Compute $j = select_1(L_c, i)$.
  - Compute $b = rank_0(L_c, j)$.
  - b is the block where to complete the answer.
  - Within block b we must compute $select_c(j - select_0(L_c, b))$.
  - To that position we must add $b \cdot \sigma$.

Based on Permutations

Solving inside a block

- We store the positions where the first symbol occurs, in order, then those where the second symbol occurs, and so on.
- The result is a permutation π of $[1, \sigma]$.
- Store also a bitvector $P[1, 2\sigma]$ with a 1 per occurrence stored in π of each symbol, inserting a 0 before and every time the symbol changes.
- Then $select_c(i) = \pi(select_0(P, c) - c + i - 1)$.
- Similarly, $S[i] = rank_0(P, select_1(P, \pi^{-1}(i)))$.

Based on Permutations

	a l a b a r	a	l a	a l a b a r	d a
$B_{\_}$	0 0 0 0 0 0	1 0 1 0 0 1	0 0 0 0 0 0		0 0
B_a	1 0 1 0 1 0	0 1 0 0 1 0	1 0 1 0 1 0		0 1
B_b	0 0 0 1 0 0	0 0 0 0 0 0	0 0 0 1 0 0		0 0
B_d	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0		1 0
B_l	0 1 0 0 0 0	0 0 0 1 0 0	0 1 0 0 0 0		0 0
B_r	0 0 0 0 0 1	0 0 0 0 0 0	0 0 0 0 0 1		0 0

π 1 3 5 4 2 6

P 0 0 1 1 1 0 1 0 0 1 0 1 0

Based on Permutations

- The most difficult part is $rank_c(S, i)$:
  - The zone of the occurrences of c in π is $[l, r] = [select_0(P, c) - c + 1 \ldots select_0(P, c + 1) - (c + 1)]$.
  - With a binary search we obtain time $O(\log \sigma)$.
  - To obtain $O(\log \log \sigma)$:
    - Note the values $\pi[i \cdot \log \sigma]$ on a list.
    - This list contains $\frac{\sigma}{\log \sigma}$ numbers in $[1, \sigma]$.
    - Separate them into increasing sublists, one per c value.
    - We need to compute predecesor and succesor of i in one of those sublists.
    - Classical predecessor data structures obtain $O(\log \log \sigma)$ time using $O(\sigma)$ extra bits.
    - Complete with a binary search between samples, in time $O(\log \log \sigma)$.

Based on Permutations

- Bitvectors L_c use $2n + o(n)$ bits.
- Permutations π use $n \log \sigma$ bits.
- To obtain π^{-1} in time $O(\log \log \sigma)$ (for *access*) we need shortcuts, using $O(n \cdot \frac{\log \sigma}{\log \log \sigma}) = n \cdot o(\log \sigma)$.
- Bitvectors P use $2n + o(n)$ bits (the same L_c reordered).
- The predecessor and successor structures use $O(n)$ bits (may be reduced to $o(n)$).
- Total: $n \log \sigma + O(n \cdot \frac{\log \sigma}{\log \log \sigma})$ bits.
- $S[i]$ and $rank_c$ take time $O(\log \log \sigma)$.
- $select_c$ is computed in constant time.

Alphabet Partitioning

- Can we obtain these good times combined with zero-order compression?
- Yes, by combining wavelet trees and permutation-based representations.
- The space becomes $nH_0(S) + o(n(H_0(S) + 1))$.
- Recall that symbol c appears n_c times in S.
- Say that the class of c is $\lceil \log(n/n_c) \cdot \log n \rceil$.
- The representation is formed by:
  - A sequence $C[1, \sigma]$ where $C[c]$ is the class of symbol c.
  - A sequence $K[1, n]$ where $K[i]$ is the class of symbol $S[i]$.
  - A sequence S_k with the symbols of S of class k, for each k.
  - Instead of S_k, store L_k, where the symbols are mapped to $\{1, 2, \ldots\}$.

Alphabet Partitioning

A simplified example:

S `to be or not to be, that is the question`

K 012113210012102302302133130212

`abehinoqrstu`

C 321222133303

S_0	`ttttttt`	S_2	`bnbhihin`
L_0	1111111	L_2	14123234
S_1	`oeoooeeeo`	S_3	`rasqus`
L_1	212221112	L_3	314254

symbol	*freq*	*class*
t	7	0
o	5	1
e	4	
b	2	2
h	2	
i	2	
n	2	
s	2	3
a	1	
q	1	
r	1	
u	1	

Alphabet Partitioning

Then we have the following translation:

- For $access(S, i)$, we find the class $k = access(K, i)$, then its local position in L_k is $p = rank_k(K, i)$, the mapped symbol is $s = access(L_k, p)$, and finally the original symbol is $select_k(C, s)$.
- For $rank_c(S, i)$, we find the class $k = access(C, c)$, which appears $p = rank_k(K, i)$ times before position i, and the number of cs is then $rank_s(L_k, p)$, where $s = rank_k(C, c)$ is the mapped value of c in L_k.
- For $select_c(S, j)$, we find class $k = access(C, c)$ and the mapped symbol $s = rank_k(C, c)$. Then the symbol is at $p = select_s(L_k, j)$, corresponding to position $select_k(K, p)$ in S.

Alphabet Partitioning

Alphabet Partitioning

- We represent C and K with wavelet trees, and the L_k with the permutation-based representation.
- The alphabets for C and K are polylogarithmic.
- If we use the best possible wavelet tree for C and K, then the times are $O(1)$, and then each operation time is dominated by the same operation time on some L_k.
- The L_k are not compressed, but they store symbols of the same class, whose probability is about the same.
- Thus using $\log \sigma_k$ (σ_k is the alphabet size of L_k) does not really harm compression.
- The key is to show that σ_k is not too large.

Alphabet Partitioning

- Let a and b be any two symbols of the same class k.
- Then

$$\lceil \log(n/n_a) \log n \rceil = \lceil \log(n/n_b) \log n \rceil$$

$$\log(n/n_b) - \log(n/n_a) < 1/\log n$$

$$n_a < 2^{1/\log n} n_b$$

- Adding over the σ_k distinct symbols n_b, we have

$$\sum_{b,C[b]=k} n_a < \sum_{b,C[b]=k} 2^{1/\log n} n_b$$

$$\sigma_k n_a < 2^{1/\log n} |L_k|$$

$$\sigma_k < 2^{1/\log n} |L_k|/n_a$$

Alphabet Partitioning

- Now, adding up the sizes of all the L_k and also the compressed size of K, we have

$$\sum_k |L_k| \log \frac{n}{|L_k|} + \sum_k |L_k| \log \sigma_k$$

$$= \sum_k \sum_{a, C[a]=k} n_a \log \frac{n}{|L_k|} + \sum_k \sum_{a, C[a]=k} n_a \log \sigma_k$$

$$< \sum_k \sum_{a, C[a]=k} n_a \log \frac{n}{|L_k|} + \sum_k \sum_{a, C[a]=k} n_a \log(2^{1/\log n} |L_k|/n_a)$$

$$= \sum_k \sum_{a, C[a]=k} n_a \log \frac{n}{n_a} + O(n/\log n)$$

$$= nH_0 + O(n/\log n)$$

- The sublinear terms add up to $o(nH_0)$.

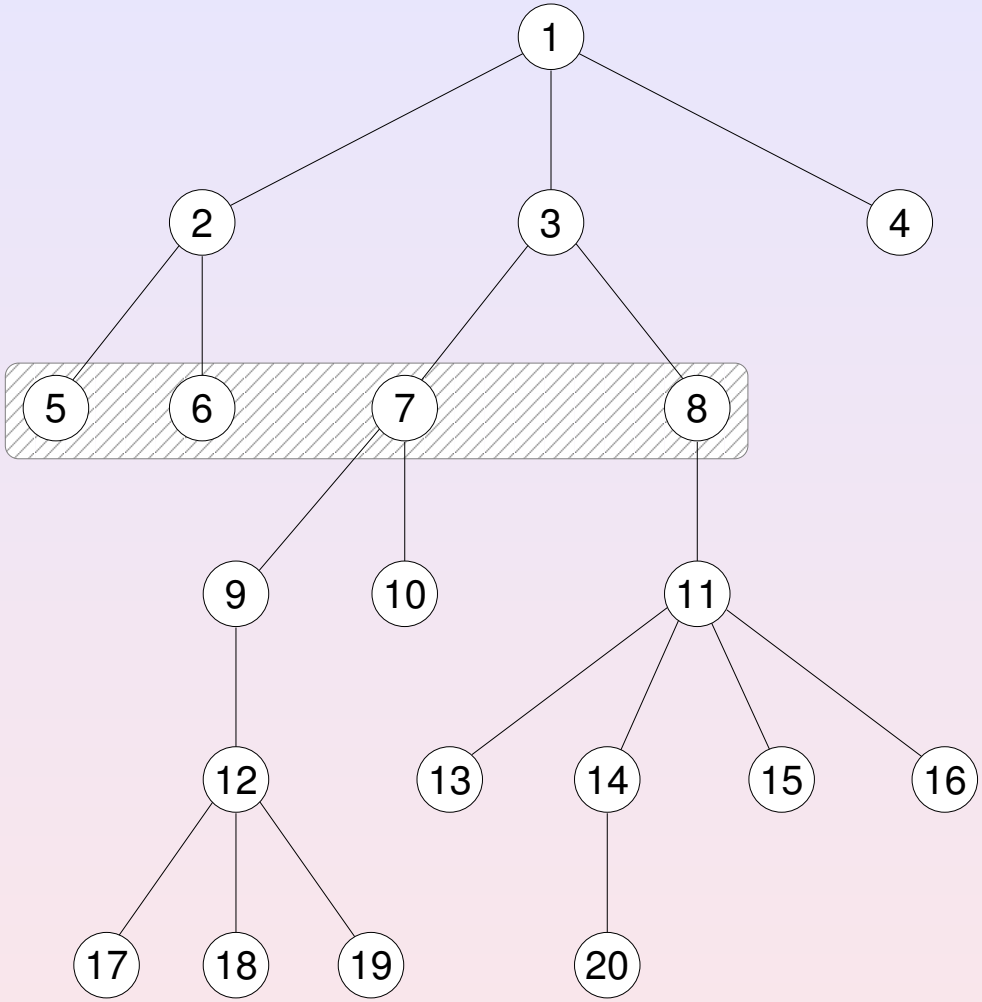
Ordinal Trees

- A classic pointer-based structure uses $O(n \log n)$ bits for a tree of n nodes.
- It allows us to navigate the tree in constant time.
- But the entropy of trees is just $2n - \Theta(\log n)$ bits.
- It is not hard to represent a tree within this space (compression)
- ... the chalenge is to navigate it without decompressing (compact structure).
- We will see how to do this in constant time per operation, using $2n + o(n)$ bits.

LOUDS Representation

- This represents an n-node tree using exactly $2n+1$ bits, and navigates it using *rank* and *select* on the bitvector $B[1, 2n+1]$.
- We start writing 10 on B, to avoid some border cases.
- Then we traverse the tree levelwise. For each visited node v having k children, we append 1^k0 to B.
- This is called the description of v, and we identify v with the first position of its description.
- So we append one 0 per node (n in total) and one 1 per child ($n-1$ in total).
- The levelwise index of v is then $rank_0(B, v-1)$, and the node with levelwise index i is $select_0(i)+1$.
- The key property: nodes are enumerated levelwise twice: at their 0-terminated description, and with a 1 within the description of their parent.

LOUDS Representation



$$B = 10\,11\,10\,11\,10\,11\,00\,00\,11\,01\,01\,00\,11\,11\,01\,11\,10\,01\,00\,00\,00\,00$$

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 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

1 2 3 4

LOUDS Representation

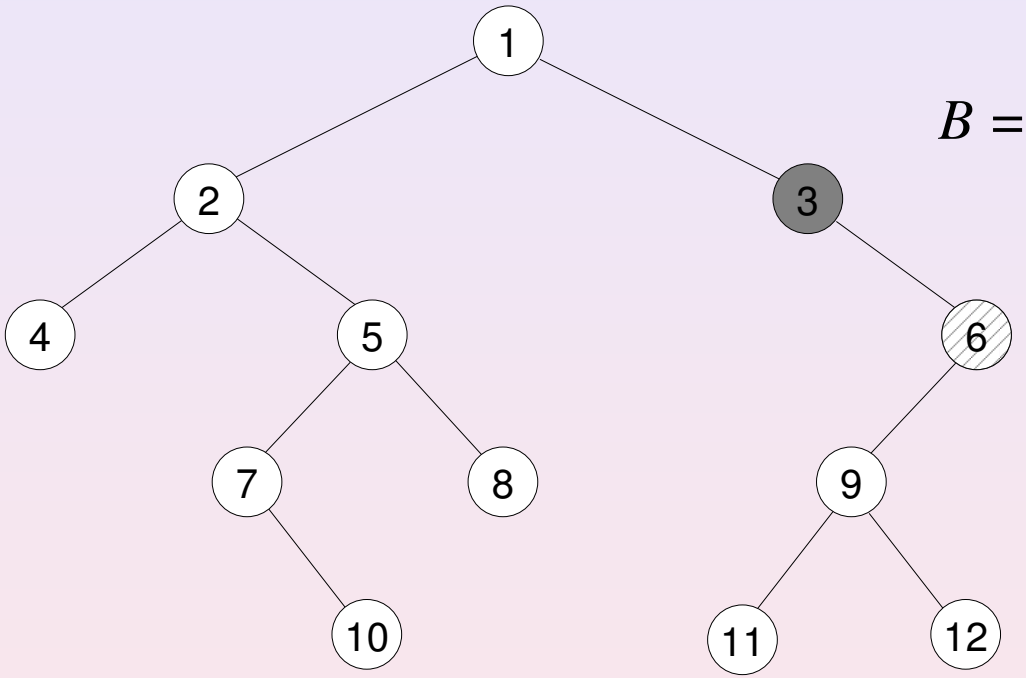
This supports simple navigation operations

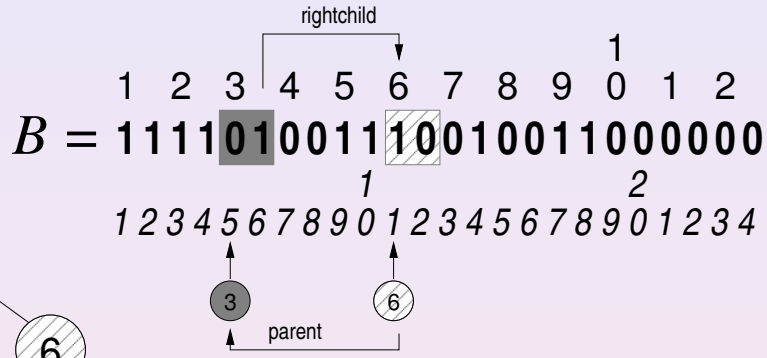
- The root is $v = 3$.
- v is a leaf iff $B[v] = 0$.
- The number of children of v is $select_0(v, rank_0(v-1)+1) - v$.
- The t-th child of v is $select_0(B, rank_1(B, v-1+t)) + 1$.
- The parent of v is $select_0(rank_0(B, j)) + 1$, where $j = select_1(B, rank_0(B, v-1))$.
- and v is the t-th child of its parent, for $t = j - select_0(rank_0(B, j))$.

LOUDS Representation

- LOUDS can also support cardinal (k-ary) trees, by attaching the k-bit description of the nodes in levelwise order.
- The operations are slightly simpler.
- For example, for binary trees:
  - The root is 1.
  - The parent of v is $\lceil select_1(B, v-1)/2 \rceil$.
  - Node v is a leaf iff $B[2v-1, 2v] = 00$.
  - The left child of v is $rank_1(B, 2v-1) + 1$.
  - The right child of v is $rank_1(B, 2v) + 1$.
  - Node v is a left (1) or right (2) child of its parent with: $((select_1(B, v-1) - 1) \bmod 2) + 1$.

LOUDS Representation





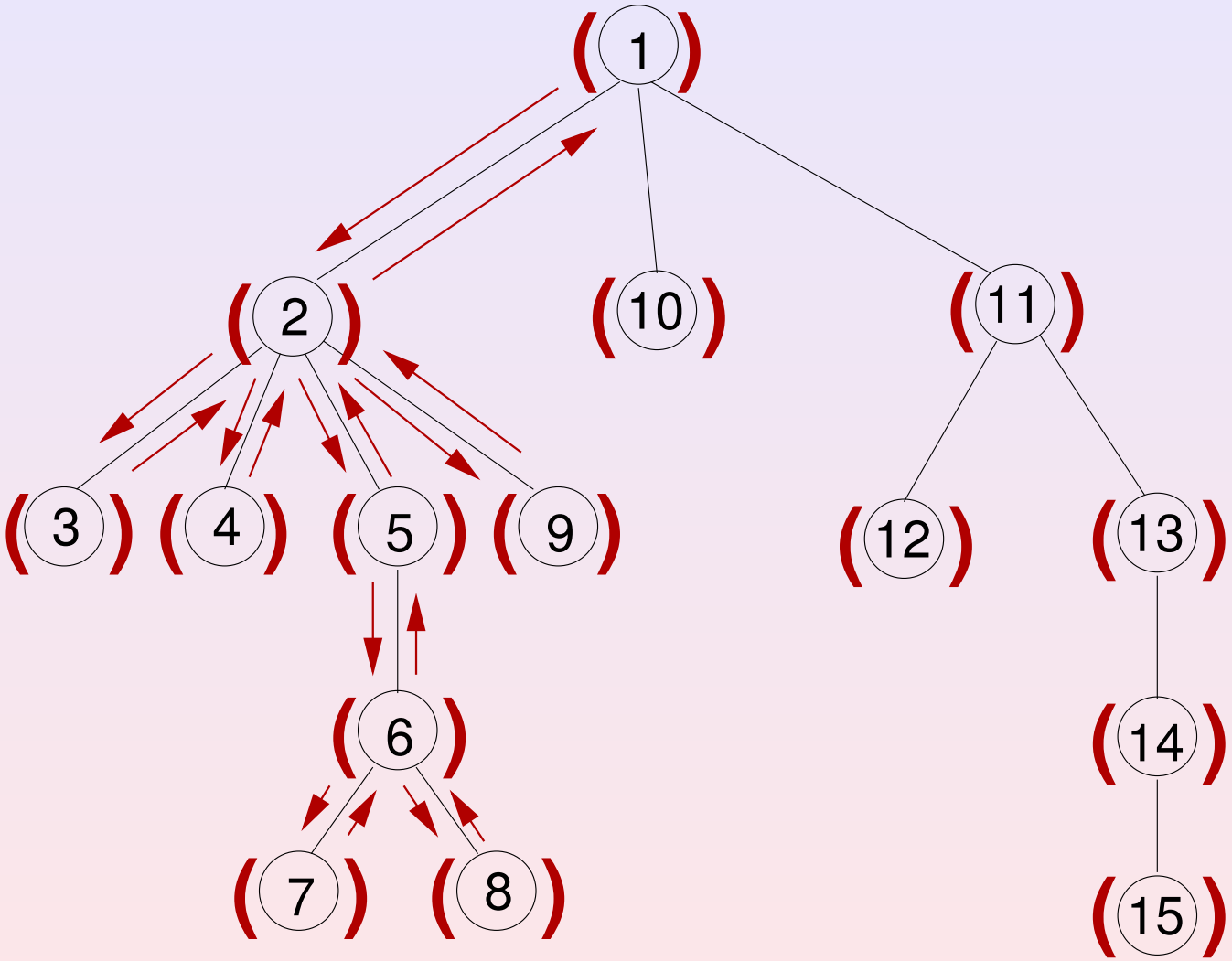
Representation with Parentheses

- A more powerful representation of ordinal trees uses balanced parentheses.
- We traverse the tree in preorder.
- That is, upon arriving at a node:
  - We write a '('.
  - We traverse its children in order.
  - We write a ')'.
- Or, said another way, the representation $R(v)$ of the subtree rooted at node v, which has children $v_1, v_2 \ldots v_k$ is

$$R(v) \quad = \quad (\; R(v_1) \, R(v_2) \; \ldots \; R(v_k) \;)$$

- The result is a balanced sequence of $2n$ parentheses.

Representation with Parentheses



Representation with Parentheses

- A balanced sequence of parentheses satisfies:
  - It is a sequence over symbols `'('` and `')'`.
  - There are as many `'('` as `')'`.
  - At any point i in the sequence, the number of `'('` in $S[1, i]$ is $\geq$ the number of `')'` in $S[1, i]$.
- This last value is called the excess in i:

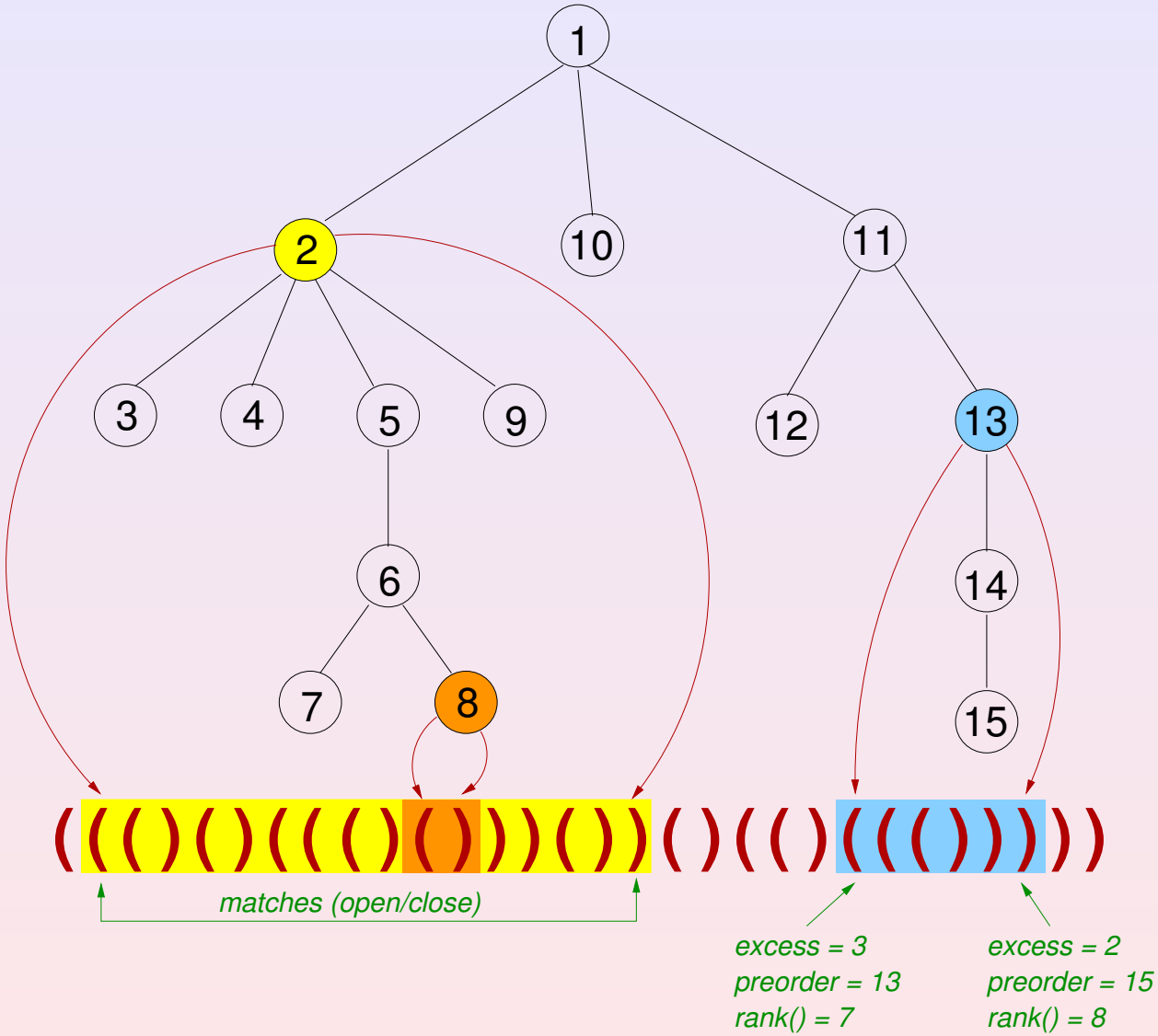
$$excess(S, i) \quad = \quad rank_{'('}(S, i) - rank_{')'}(S, i)$$

and the property then says that $excess(S, i) \geq 0$ for all i, and that $excess(S, 2n) = 0$.

Representation with Parentheses

- We call the match of a `'('` the `')'` that closes it, and the match of a `')'` the `'('` that opens it.
- If $S[v] =' ('$, we call $close(v)$ the position of its match.
- If $S[v] =')'$, we call $open(v)$ the position of its match.
- Note that:
  - $close(v)$ is the smallest $v' > v$ s.t. $excess(v') = excess(v) - 1$.
  - $open(v')$ is the largest $v < v'$ s.t. $excess(v) = excess(v') + 1$.

Representation with Parentheses



Representation with Parentheses

Various interesting properties

- We identify node v with its corresponding `'('`.
- A leaf looks like `'()'`.
- u is an ancestor of v iff $[u, close(u)]$ strictly contains $[v, close(v)]$.
- The depth of v in the tree is $excess(v)$.
- The preorder position of v is $rank_{'('}(S, v)$.
- The subtree size of v is $(close(v) - v + 1)/2$.

Representation with Parentheses

Navigation

- The next sibling of v is $close(v) + 1$
  - But if it is a `')'`, then v has no next sibling.
- The previous sibling of v is $open(v - 1)$
  - But if $S[v - 1] =$ `'('`, then v has no previous sibling.
- The first child of v is $v + 1$
  - But if it is a `')'`, then v has no children.
- The parent of v is not that easy to compute.
  - We create operation $enclose(v)$.
  - It is the `'('` of the deepest node that contains v.

Representation with Parentheses

- Thus, it suffices with implementing *open*, *close* and *enclose*...
- ... to have a good number of navigation operations on a tree represented with parentheses.
- We can achieve constant time for these operations using $2n + o(n)$ bits.
- But there are several other important operations:
  - d-th ancestor
  - lowest common ancestor
  - t-th child
  - height, deepest leaf
  - next and previous node of the same level

Navigating with Parentheses

A large number of the operations is solved with these few primitives:

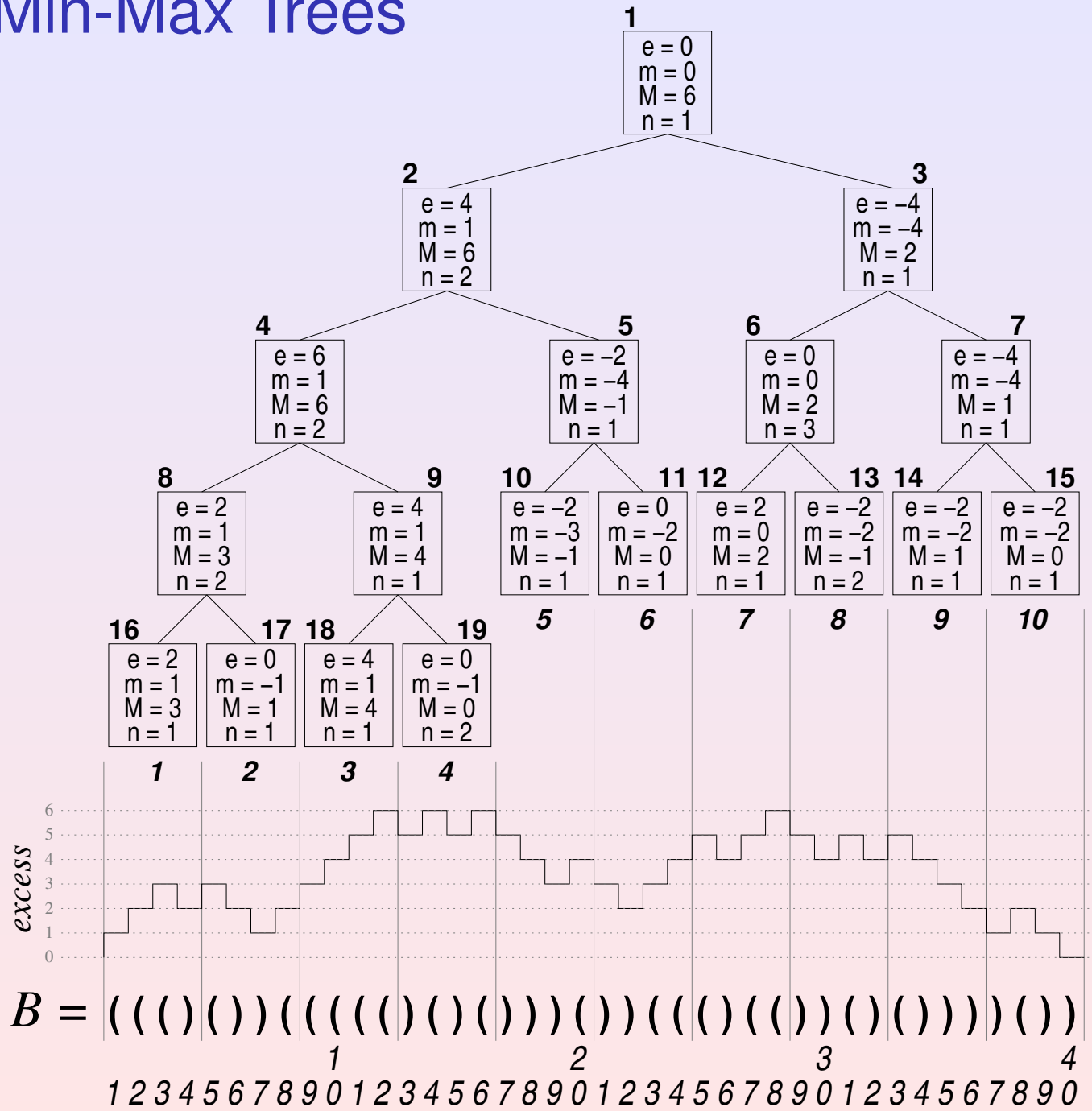
- $fwd\_search(i, d) = \min\{j > i, excess(j) = excess(i) + d\}$
- $bwd\_search(i, d) = \max\{j < i, excess(j) = excess(i) - d\}$
- $rmq(i, j) = \text{argmin}\{excess(k), i \leq k \leq j\}$
- $RMQ(i, j) = \text{argmax}\{excess(k), i \leq k \leq j\}$

Navigating with Parentheses

For example

- $close(i) = fwd\_search(i, 0)$
- $open(i) = bwd\_search(i, 0)$
- $enclose(i) = bwd\_search(i, 2)$
- $level{-}ancestor(i, \ell) = bwd\_search(i, \ell + 1)$
- $lca(i, j) = parent(rmq(i, j) + 1)$
- $deepest{-}node(i) = RMQ(i, close(i))$
- $height(i) = depth(deepest{-}node(i)) - depth(i)$
- $level{-}next(i) = fwd\_search(close(i), 0)$
- $level{-}prev(i) = bwd\_search(i, 0)$

Range Min-Max Trees



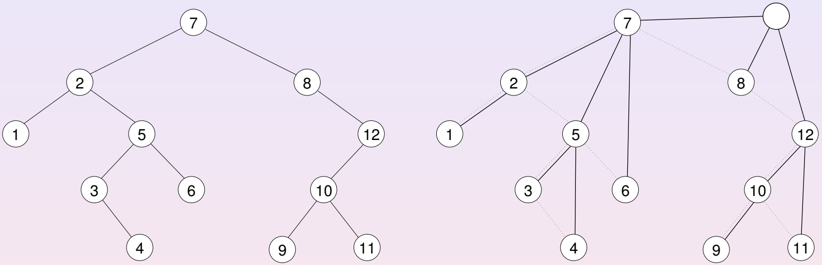
Range Min-Max Trees

- The four operations are solved climbing up and then down.
- Others, like the d-th child, are done by finding the dth occurrence of the minimum excess in $[v + 1, close(v) - 1]$.
- Thus the range min-max tree also counts how many times the minimum occurs in its subtree.
- This also allows us know the number of children of a node, and whose child of its parent a node is.

Complexities

- The time is logarithmic in n.
- Making the rmM-tree of arity $\sqrt{\log n}$, time becomes constant for polylogarithmic segments of parentheses.
- For inter-segment operations we use other structures (there are less space problems).
- This involves some novel structures, such as left-to-right minima trees and weighted level ancestors.
- Finally, all the operations are done in constant time.
- Space is $2n + O(n/\log^c n)$ bits, for any c (which is optimal).

Isomorphism with Binary Trees




```
                       1 1     1
  7 2 1   5 3  4   6   8  2 0 9    1
B = ( ( ( ( ) ) ( ( ) ( ) ) ( ) ) ( ) ( ( ( ) ) ( ) ) )
                  1           2
  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
```

Isomorphism with Binary Trees

- The root node is 2.
- v is a leaf iff $B[v, v+2] = ())$; we can also count leaves.
- The left child of v is $v+1$ (if it is not a).
- The right child of v is $close(v)+1$ (if it is not a).
- The parent of v is $v-1$ if this is a ((v is a left child),
- otherwise it is $open(v-1)$.
- The subtree size of v is $(fwdsearch(v, -2) - v)/2$.
- u is an ancestor of v iff $u \leq v \leq fwdsearch(u, -2)$.
- preorder is preorder, inorder is postorder.
- The LCA of $u < v$, if u is not an ancestor of v, is $open(rmq(u, v))$.

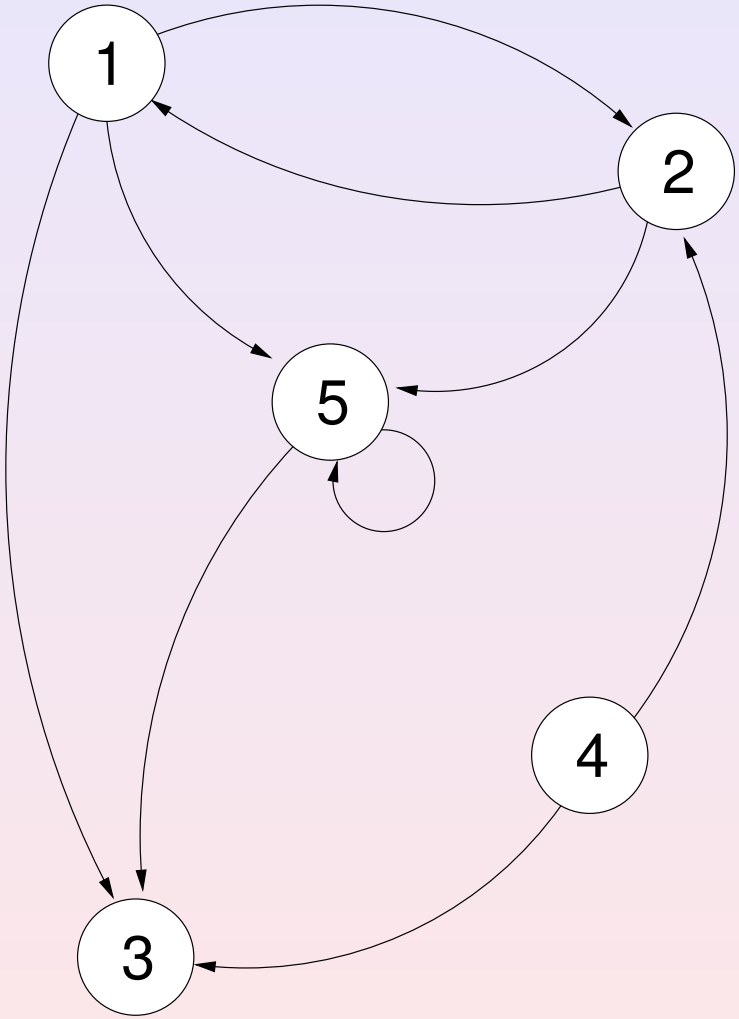
Graphs

- Let $G(V, E)$ be a directed graph with n nodes and e edges.
- There are $\binom{n^2}{e}$ possible graphs.
- Therefore, the entropy is $e \log \frac{n^2}{e} + O(e)$.
- An adjacent list representation uses $e \log n + n \log e$, not so far.
- There has been work to compress families of graphs, such as planar graphs in $O(n)$ bits.
- We instead focus on general graphs, providing forward and backward navigation within the same space.
- Also, we do not need to double the space for undirected graphs.

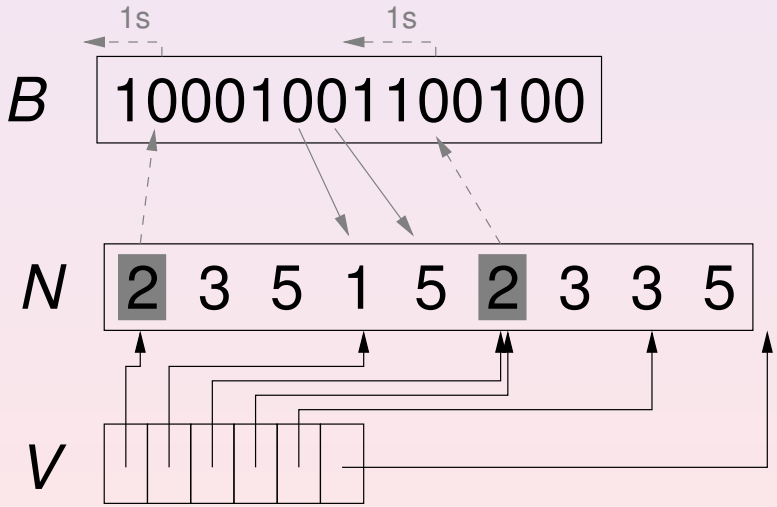
Graphs

- Let us represent the concatenation of adjacency lists as a sequence $N[1, e]$ and a bitvector $B[1, n + e]$ that counts in unary the list lengths.
- Then the i-th neighbor of node v is $N[select_1(B, v) - v + i]$.
- And node v has $select_1(B, v + 1) - select_1(B, v) - 1$ neighbors.
- But we can also list reverse neighbors using *select* on N!
- The i-th reverse neighbor of v is $select_0(B, p) - p$, where $p = select_v(N, i)$.
- And the number of reverse neighbors of v is $rank_v(N, e)$.
- On undirected graphs, we represent one of the two edges and consider both direct and reverse neighbors.

Graphs



N(1) = 2,3,5
N(2) = 1,5
N(3) =
N(4) = 2,3
N(5) = 3,5



Graphs

- Thus we find any neighbor in $O(\log \log n)$ time or less.
- We find any reverse neighbor in constant time.
- We can represent the transpose of G to exchange those times.
- On undirected graphs, we represent one of the two edges and consider both direct and reverse neighbors.
- We can use alphabet partitioning to reach the zero-order entropy of node indegrees.

Two-Dimensional Grids

- Say we have an $[1, c] \times [1, r]$ grid with n points, all at distinct positions to simplify.
- We convert it into an $[1, n] \times [1, r]$ grid with n points, with exactly one point per x-coordinate.
- This is done with a bitvector $B[1, c + n]$ that tells in unary the number of points at each actual coordinate.
- With *rank* and *select* on B we easily map from one grid to the other.
- The new grid can be represented as a sequence $S[1, n] = y_1 y_2 \ldots y_n$ of the n y-coordinates, over alphabet $[1, r]$.
- We will see that a wavelet tree representation of S solves some relevant geometric queries.

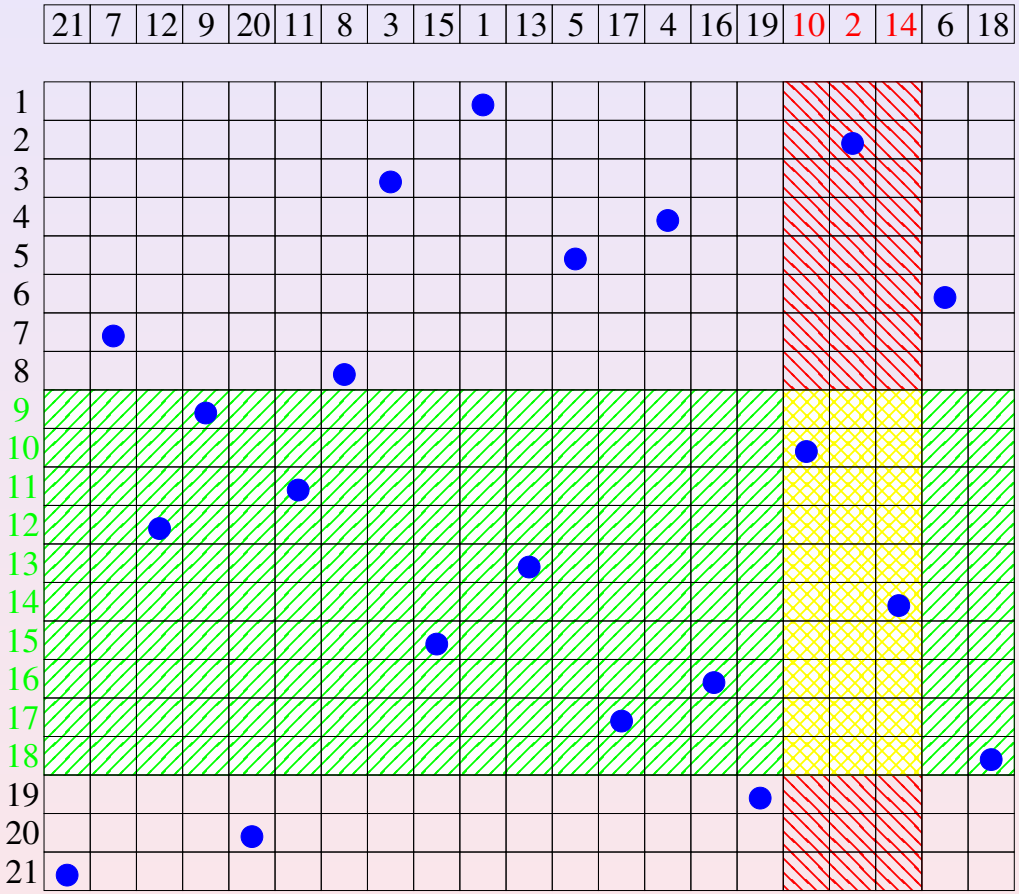
Two-Dimensional Grids

- We wish to answer range queries:
  - ¿How many points are there in $[x_1, x_2] \times [y_1, y_2]$?
  - ¿Which points are there in $[x_1, x_2] \times [y_1, y_2]$?
- With a binary wavelet tree we answer those queries:
  - Using $(c + n \log r)(1 + o(1))$ bits.
  - Counting the number of points in time $O(\log r)$.
  - Reporting each point in time $O(\log r)$.
- Multi-ary wavelet trees reduce these times to $O(\log r / \log \log r)$, which is optimal for counting.
- If $r > n$ we can also compact the y-coordinates with another bitvector.

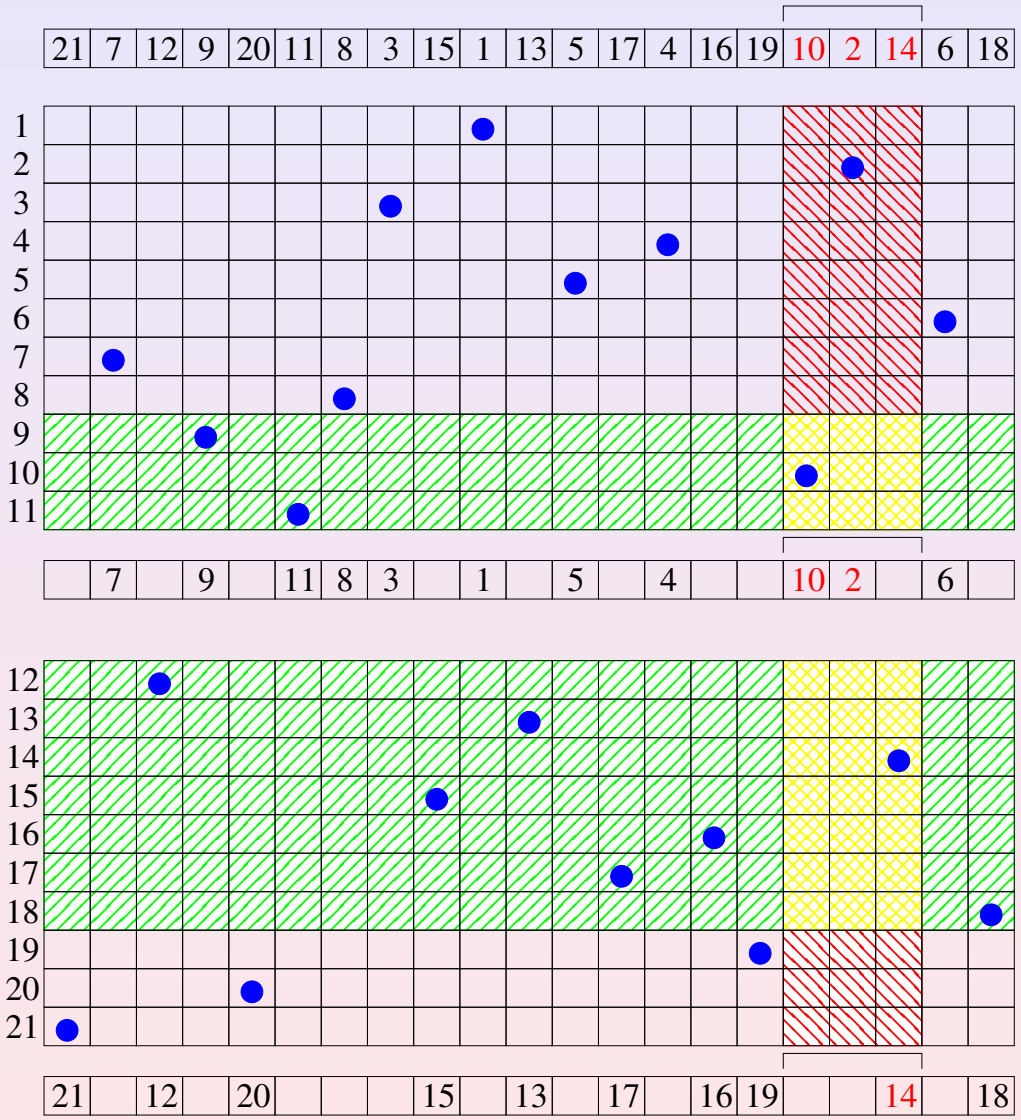
Two-Dimensional Grids

- Let us consider the wavelet tree on S.
- The segments of S correspond to ranges in the x-axis.
- The partitions in halves made by the wavelet tree correspond to the y-axis.
- Given a point $p = (x, y)$, we have $S[x] = y$.
- If we track p down the tree, using *rank*, we reach the y-th leaf.
- If we track a y value upwards from a leaf, using *select*, we reach the corresponding x-coordinate position at the root.

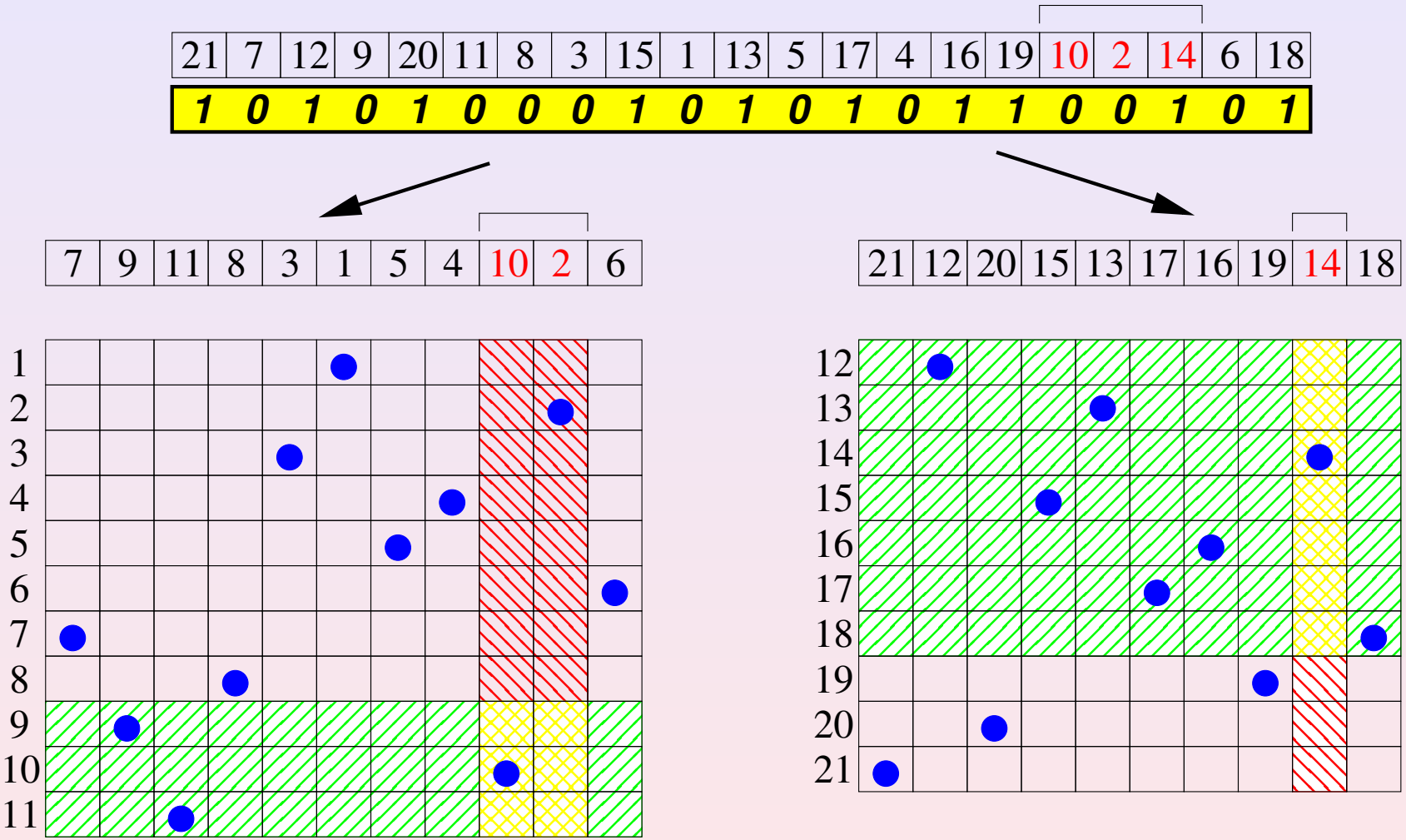
Two-Dimensional Grids



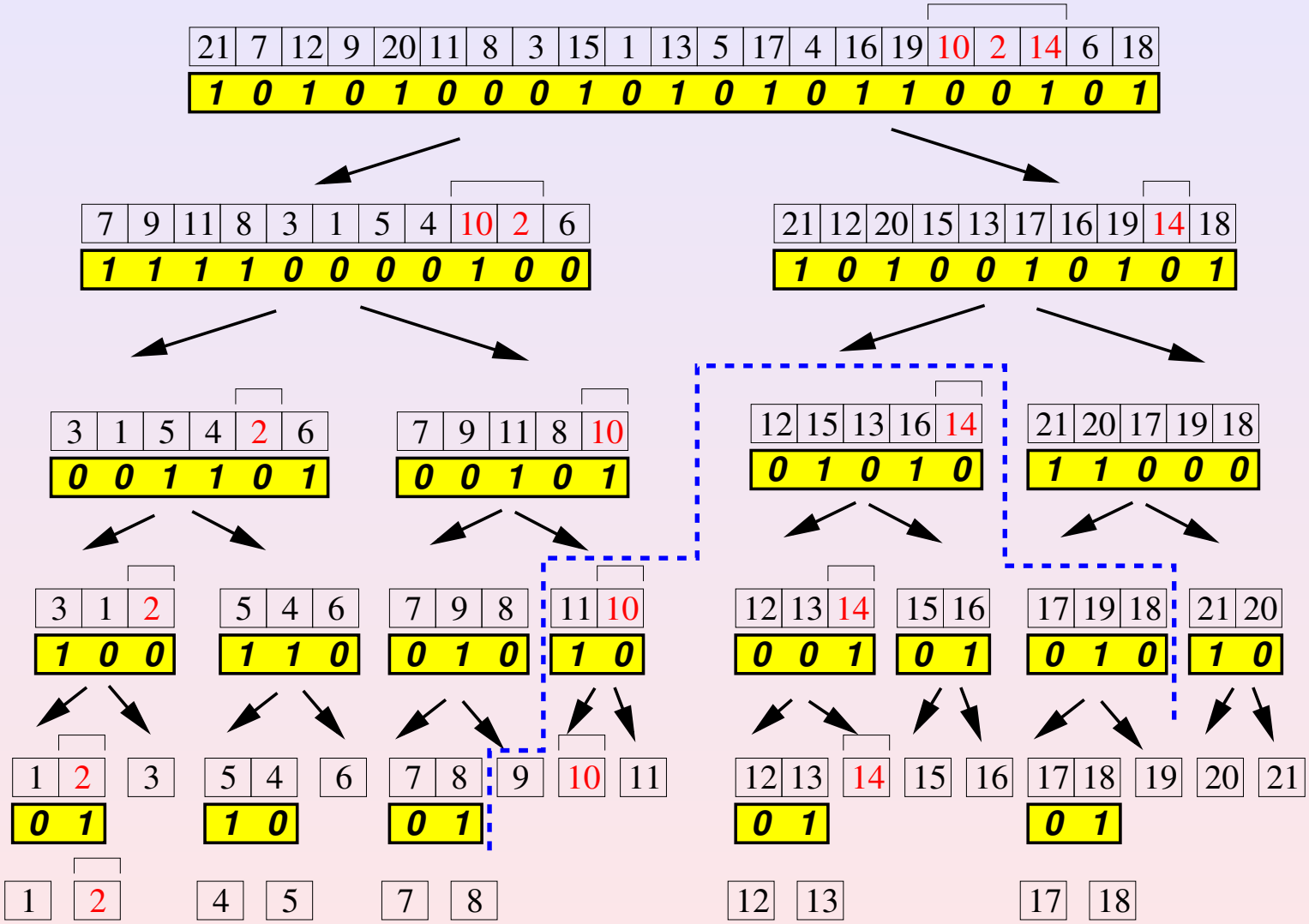
Two-Dimensional Grids



Two-Dimensional Grids



Two-Dimensional Grids



Two-Dimensional Grids

- More in general, we need to project a range of x-values:
  - If in a node v, corresponding to range $[y, y']$,
  - we have a range $B[x, x']$ on the bitvector of v,
  - then $[1 + rank_0(B, x-1), rank_0(B, x')]$ is the range of those points falling on the first half of $[y, y']$,
  - and $[1 + rank_1(B, x-1), rank_1(B, x')]$ is the range of those points falling on the second half of $[y, y']$.

Two-Dimensional Grids

- To count how many points are there in $[x_1, x_2] \times [y_1, y_2]$:
  - We start from the root with the range $[x, x'] = [x_1, x_2]$.
  - We project the range on both subtrees.
  - We continue recursively, stopping when:
    - The interval $[x, x']$ is empty (there are no points in the original range $[x_1, x_2]$ that fall in the range $[y, y']$ of this node).
    - The interval $[y, y']$ is disjoint with the original ($[y_1, y_2]$).
    - The interval $[y, y']$ is contained in the original ($[y_1, y_2]$): Add $x' - x + 1$ to the total
  - Every subinterval $[y_1, y_2]$ is covered with $O(\log r)$ nodes $[y, y']$ of the wavelet tree.
  - The algorithm performs $O(\log r)$ operations.

Two-Dimensional Grids

- To find those points:
  - Start from each node where we have found results.
  - Track each point of $[x, x']$ downward to find its y-coordinate.
  - And/or track each point of $[x, x']$ upwards to find its x-coordinate.
- This costs $O(\log r)$ time per point.
- We must concatenate the bitvectors at each level to avoid having too many pointers in the tree.

Texts

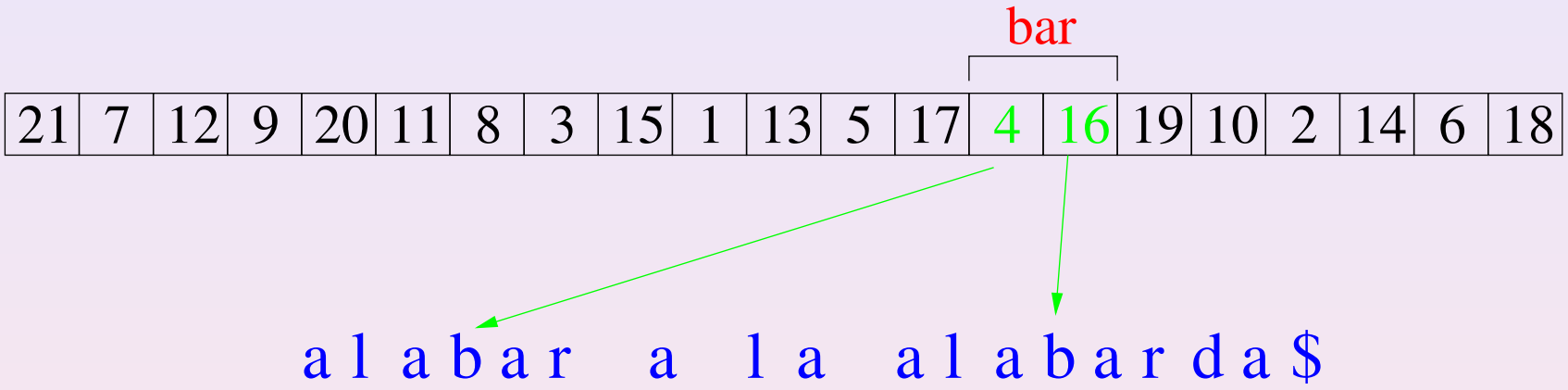
- Given an alphabet Σ of size σ...
- ... a text $T[1, n]$ is a sequence over Σ.
- This is just a sequence, but we are interested in other operations.
- Given a pattern $P[1, m]$:
  - Count the number of occurrences of P in T (occ).
  - Locate those occ occurrences in T.

Suffix Arrays

- Unlike inverted indexes, make no assumption on the text.
- Can retrieve any text substring.
- General model: consider the n suffixes $T[i, n]$ of T.
- Every substring of T is the prefix of a suffix of T.
- This structures indexes the set of suffixes of T and allows one to find all those sharing a given prefix.

Suffix Arrays

- It is an array with the n text suffixes in lexicographic order.
- Finding the substrings equal to P is the same as finding the suffixes that start with P.
- These form a lexicographical interval in the suffix array.
- This interval can be binary searched in $O(m \log n)$ time.
- However, it uses $O(n \log n)$ bits, asymptotically more than the text.


bar
21 7 12 9 20 11 8 3 15 1 13 5 17 4 16 19 10 2 14 6 18
a l a b a r a l a a l a b a r d a $

The Burrows-Wheeler Transform (BWT)

- Is a reversible permutation of T.
- It is used as a previous step to compression algorithms like bzip2.
- Puts together symbols having the same context.
- It is enough to compress those symbols "put together" to zero-order to achieve high-order entropy.
- Entropy of order k: if S_A is the sequence of the symbols that precede the occurrences of A in S, then

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

- Lower bound to encodings that consider the k symbols preceding the one to encode (e.g. PPM).

The Burrows-Wheeler Transform (BWT)

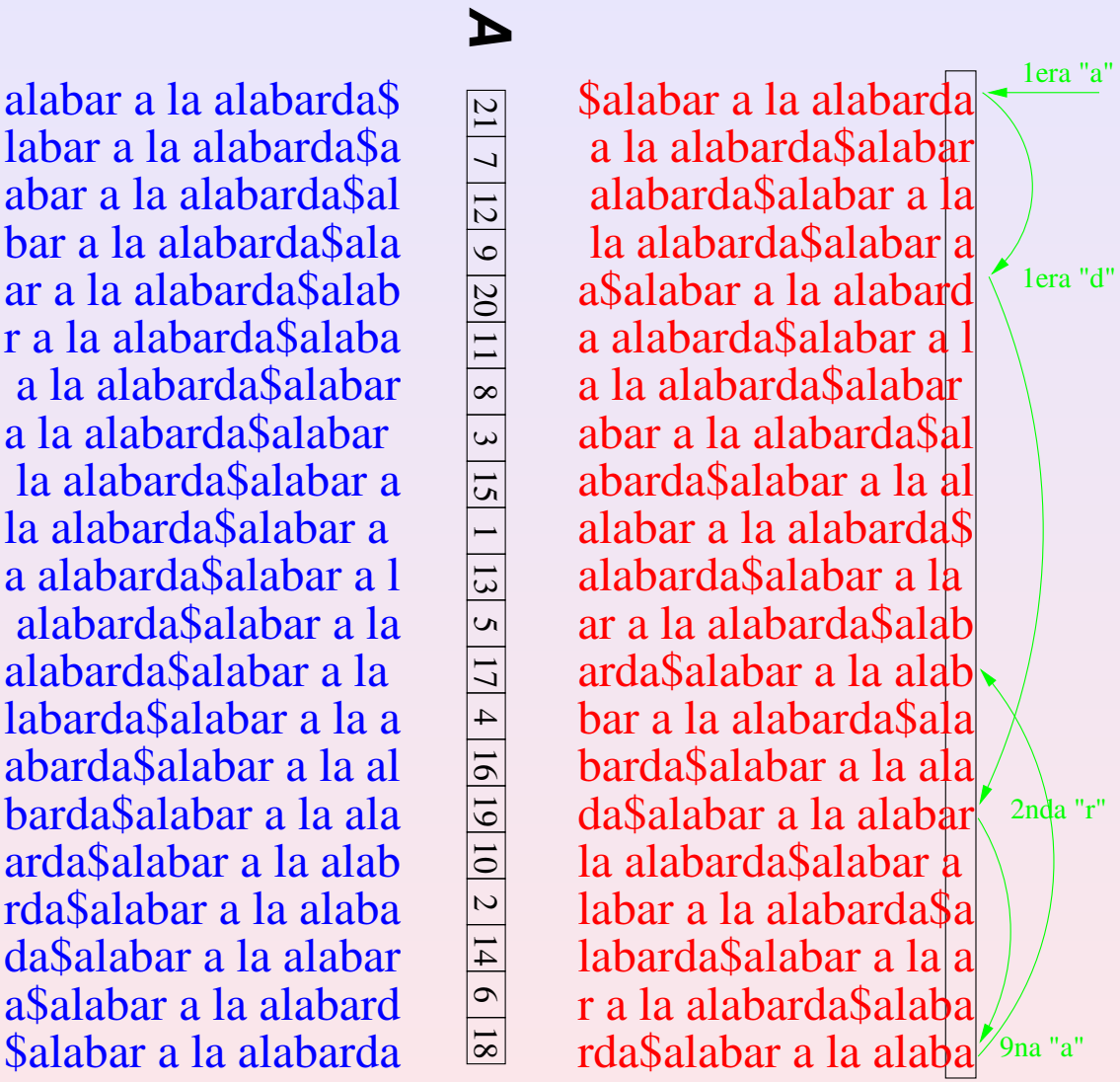
- Take all cyclic shifts of T.
- That is, $t_i t_{i+1} \dots t_{n-1} \$ t_1 t_2 \dots t_{i-1}$.
- The result is a matrix M of $n \times n$ symbols.
- Sort the rows lexicographically.
- M is essentially the list of suffixes of T in this order:

$$M[i] = T[A[i] \dots n] \cdot T[1 \dots A[i] - 1]$$

- The first column, F, holds the first characters of the suffixes: $F[i] = S[rank(D, i)]$.
- The last column, L, is the BWT of T, $T^{bwt} = L$.
- It is the sequence of symbols that precede the suffixes $T[A[i]..]$.

$$L[i] \;=\; T^{bwt}[i] \;=\; T[A[i] - 1]$$

(except if $A[i] = 1$, where $T^{bwt} = T[n] = \$$).



T alabar a la alabarda$

T^{bwt} araadl ll$ bbaar aaaa

The Burrows-Wheeler Transform (BWT)

- How to revert the BWT?
- For all i, $T = \ldots L[i]F[i]\ldots$.
- We know $L[1] = T[n-1]$, since $F[1] = \$ = T[n]$.
- Where is $c = L[1]$ in F?
- All occurrences of c appear in the same order in F and L:
  - Is the order given by the suffix that follows c in T.
- That c is at $F[C[c] + rank_c(L, i)]$, where

$$C[c] \quad = \quad \text{number of occurrences of symbols } < c \text{ in } T$$

- This is called LF-mapping, since it maps from L to F:

$$LF(i) = C(L[i]) + rank_{L[i]}(L, i)$$

- Then $T[n-2] = L[LF(1)]$, $T[n-3] = L[LF(LF(1))]$, etc.

The FM-Index

- Uses the connection between the suffix array and the BWT.
- Replaces the binary search by the more efficient backward search.
- To search for $P[1, m]$ we start with p_m.
- The segment of A corresponding to it is $A[C(p_m) + 1, C(p_m + 1)]$.
- In general, we will know that the occurrences of $P[i + 1, m]$ start at $A[sp_{i+1}, ep_{i+1}]$...
- ... and will use something similar to the LF-mapping to obtain the zone $A[sp_i, ep_i]$ corresponding to $P[i, m]$.
- We finish with sp_1 and ep_1.

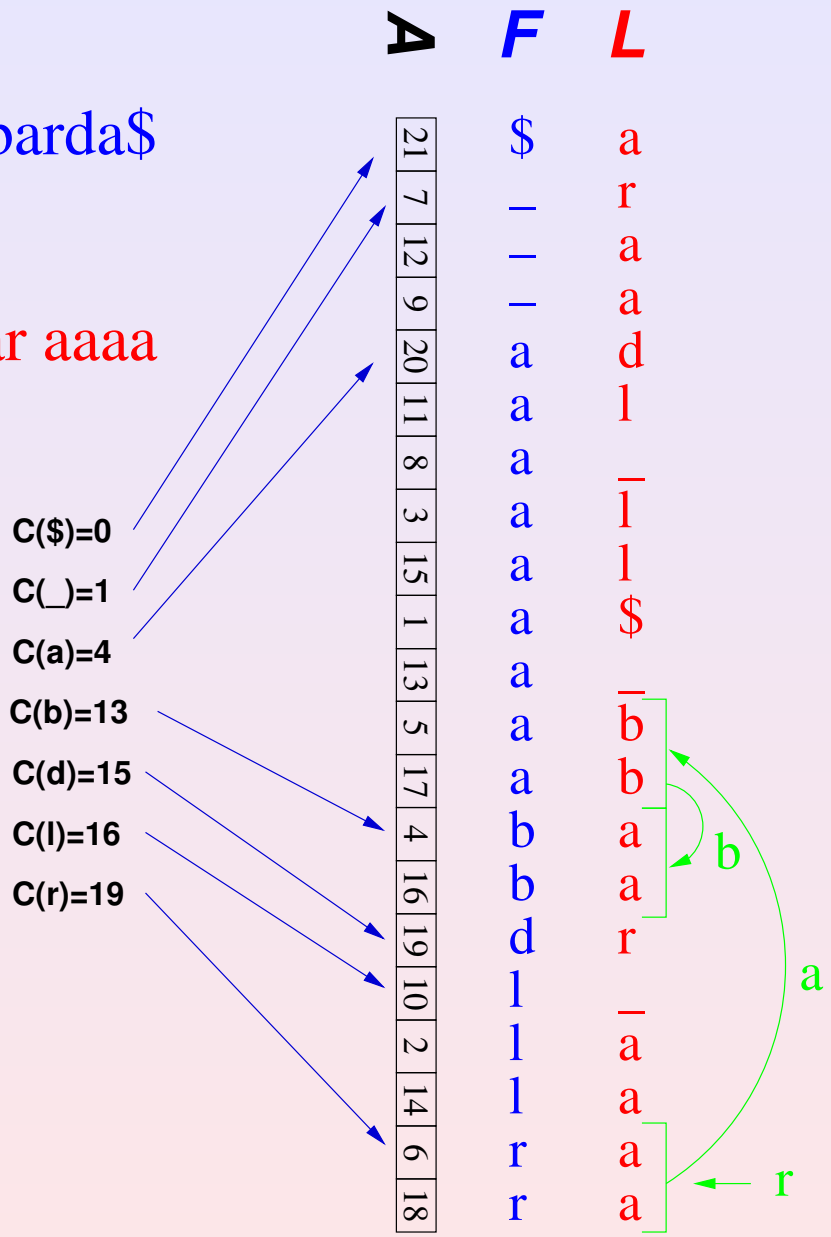
The FM-Index

- Assume we start with the segment $A[sp_{i+1}, ep_{i+1}]$ of the occurrences of $P[i+1, m]$.
- How we update it to the occurrences of $P[i, m]$?
  - For a $sp_{i+1} \leq j \leq ep_{i+1}$, $M[j][1..m-i] = P[i+1, m]$.
  - The occurrences of $P[i, m]$ in T appear as $L[j] = p_i$ in this area, since $T = \ldots L[j] \cdot M[j][1..m-i] \ldots$
  - Hence the answer is the range of the $LF(j)$ where $L[j] = p_i$.
- This is computed simply as

$$\begin{aligned} sp_i &= C(p_i) + rank_{p_i}(L, sp_{i+1} - 1) + 1 \\ ep_i &= C(p_i) + rank_{p_i}(L, ep_{i+1}) \end{aligned}$$

T alabar a la alabarda$

T^{bwt} araadl ll$ bbaar aaaa



The FM-Index

- After m iterations, we get the interval for $P[1, m]$.
- The cost is $2m$ calls to $rank_c$.
- Using a wavelet tree on $T^{bwt} = L$,
  - We get space $nH_0(T) + o(n \log \sigma)$.
  - We count in time $O(m \log \sigma)$.
- If the alphabet is large, we can use alphabet partitioning:.
  - We get space $nH_0(T) + o(n(H_0(T) + 1))$.
  - We count in time $O(m \log \log \sigma)$.
- It is possible to reduce counting time to $O(m)$.

The FM-Index

- But, how to obtain $A[i]$ and $T[l, r]$?
- We will sample and store some cells of A.
- We take those $A[i]$ pointing to positions of T which are multiples of some b.
- We store them in increasing order of i in contiguous form, in an array $A'[1, n/b]$.
- ... and we have a bitvector $B[1, n]$ marking the sampled is.
- The positions and the bitvector will occupy $O(\frac{n}{b} \log n)$ bits.

The FM-Index

- Assume we want to find out $A[i]$ (but have not A).
  - If $B[i] = 1$, then $A[i] = A'[rank(B, i)]$.
  - Else, if $B[LF(i)] = 1$, then $A[LF(i)] = A'[rank(B, LF(i))]$, $A[i] = A[LF(i)] + 1$.
  - Else, ...
  - If $B[LF^t(i)] = 1$, then $A[i] = A'[rank(B, LF^t(i))] + t$.
- This ends in at most b steps.
- Hence, we can know $A[i]$ in time $O(b \log \sigma)$.
- For example, in time $O(\log^{1+\epsilon} n)$, spending $o(n \log \sigma)$ extra bits.
- Now we have replaced A!

The FM-Index

- Now we store the same sampling in another way.
- We store the i values by increasing position in T, in $E[1, n/b]$.
- Say we want to extract $T[l, r]$ (but have not T).
  - The first sampled position after it is $r' = 1 + \lfloor r/b \rfloor \cdot b$.
  - And is pointed from $A[i] = A[E[1 + \lfloor r/b \rfloor]]$.
  - We know $T[r' - 1] = L[i]$.
  - We know $T[r' - 2] = L[LF(i)]$.
  - ...
  - We know $T[l] = L[LF^{r'-l-1}(i))]$.
- The total cost is $O((b + r - l) \log \sigma)$.
- For example, in time $O((r - l) \log \sigma + \log^{1+\epsilon} n)$, spending $o(n \log \sigma)$ extra bits.
- Now we have replaced the text!

The FM-Index

- Can we compress to k-th order entropy?
- Consider the $t = \sigma^k$ partitions of the BWT by context of length k

$$T^{bwt} = S = S^1 S^2 \dots S^t$$

- By the H_k formula,

$$nH_k(T) = \sum_{i=1}^{t} |S^i| H_0(S^i)$$

- Thus, we can partition the BWT and store each part in zero-order space, to obtain $nH_k(T)$ global space!

The FM-Index

- If we use wavelet trees with compressed bitvectors, this is not even necessary.
- Recall that the zero-order compression we achieved for the wavelet tree was the sum of the zero-order entropies of small blocks:

$$nH_0(S) \leq bH_0(S_1) + bH_0(S_2) + \ldots + bH_0(S_{n/b})$$

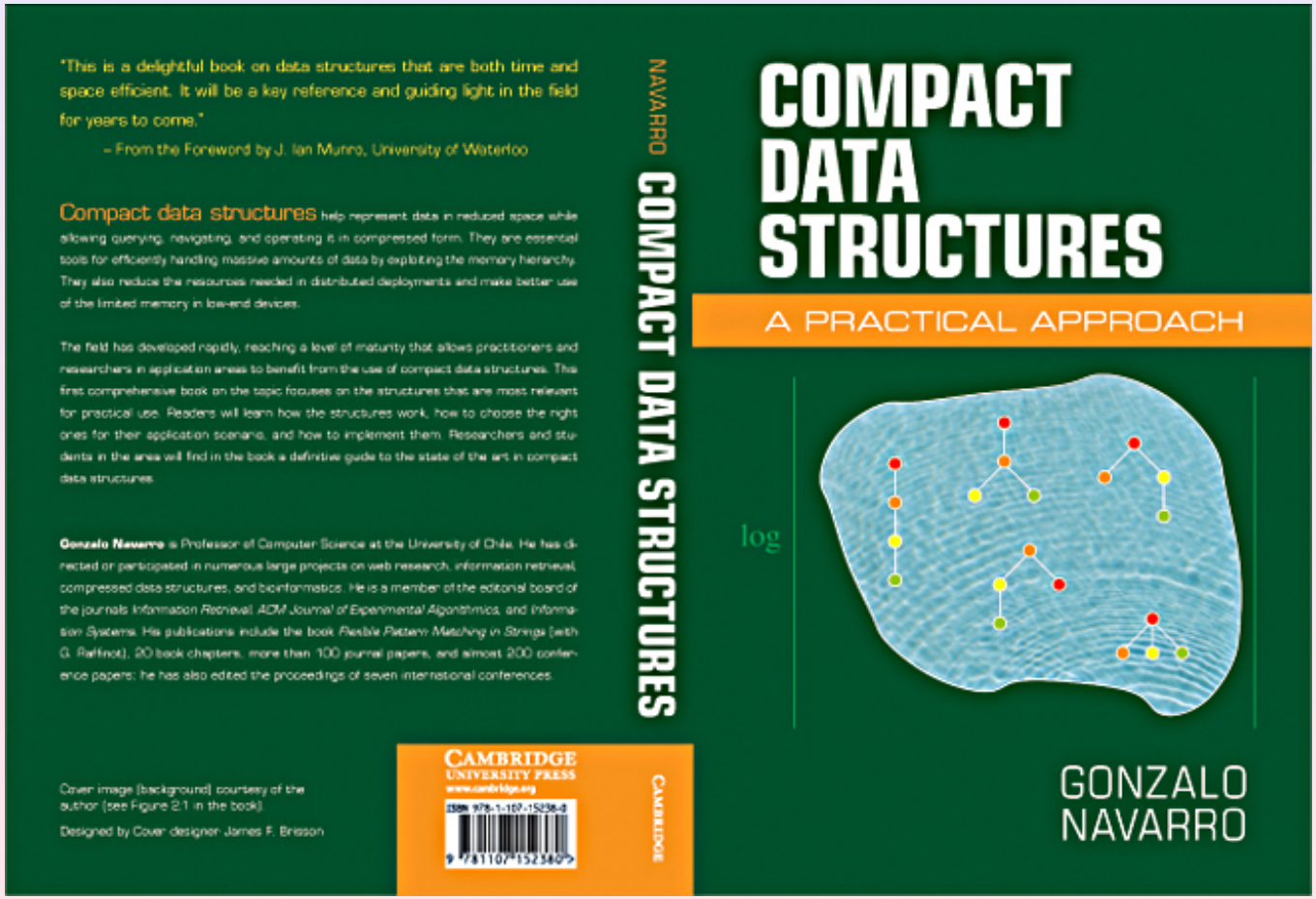
where the S_j's are the blocks of length b.

The FM-Index

- Thus the wavelet tree compresses $S^1 S^2 \ldots S^t$ to $H_0(S^1) + H_0(S^2) + \ldots + H_0(S^t)$...
- ... plus some worst-case measure for the limits among strings.
- The total size of the compressed wavelet tree is $nH_k(T) + O(\sigma^k \log n)$.
- The second term is $o(n \log \sigma)$ for moderate $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$.

Epilogue

There is an upcoming book on Compact Data Structures, focusing also on practice:



Epilogue

- There are strong and professional libraries implementing most practical succinct data structures, like SDSL:

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

- There is an active community of researchers and students working on this topic. There are papers on compact data structures appearing every year in the best conferences and journals.

Epilogue

- There are several outstanding problems, some partially unsolved, and some not solved at all.
  - How to build the structures within little space?
  - How to support updates?
  - How to improve locality of reference?
  - How to handle secondary storage?
  - How to compress highly repetitive data?
  - Which other data structures can be compacted?
  - What are the limits of encoding data structures?

Epilogue

- A brave new world of algorithmic challenges is ahead:
  - For those who love classical algorithmics, it is plenty of opportunities to redesign classical data structures.
  - For those who like compression and information theory, this gives the new challenge of designing schemes that not only can recover the data but also retain functionality and direct access to it.
- More young and enthusiastic talented students — future researchers — are needed!