<u>Indexing interlude</u>:

Bitvector Rank & Select: Primitives of succinct data structures

NOTE: This lecture is being recorded

Thinking theoretically about data structure size

Assume that storing some data, in an information-theoretically optimal manner, requires Z bits

Representation of this data is:

Implicit: Z + O(1) bits

Only a constant size larger than the theoretical minimum

Succinct: Z + o(Z) bits

Z bits, plus some term strictly smaller than Z bits

Compact: O(Z) bits

On the order of Z bits (grows linearly in Z)

https://en.wikipedia.org/wiki/Succinct_data_structure

Thinking theoretically about data structure size

The idea of succinct data structures was first introduced by Jacobson in his thesis “Succinct static data structures”*

In this thesis, among other things, he introduced succinct representations of trees and graphs that could be efficiently navigated.

As data sizes grow large, data structures that consume a lot of extra space become increasingly less feasible and so succinct data structures become increasingly important.

The *rank* and *select* operations become the basic building blocks of succinct data structures.

*Jacobson, G. J (1988). Succinct static data structures (Ph.D.). Pittsburgh, PA: Carnegie Mellon University.

Slides for the following taken from:
https://www.cs.helsinki.fi/u/puglisi/dct2015/slides10.pdf

credit to Simon J. Puglisi, University of Helsinki

Succinct Data Structures

- Succinct data structure
  = succinct representation of data + a succinct index

- (usually static)

- High-level goal: reduce space so the data structure might fit in RAM and therefore be faster to use

- Examples
  - Sets
  - Trees, graphs
  - Strings
  - Permutations, functions

credit to Simon J. Puglisi, University of Helsinki

Succinct Representation

- A representation of data whose size (roughly) matches the information-theoretic lower bound

- If the input is taken from L distinct possible inputs, then its information-theoretic lower bound is ceil(log L) bits
  - To be considered succinct a data structure must use:

    ceil(logL) + o(logL) bits

- Example: a lower bound for a set S, subset of {1,2,…,n}
  - $\log(2^n) = n$ bits
  - n = 3 we have 8 distinct sets… so d.s. will need at least 3 bits

    ∅

    {1} {2} {3}

    {1,2} {1,3} {2,3}

    {1,2,3}

credit to Simon J. Puglisi, University of Helsinki

Succinct Index

- Auxiliary data structure to support queries on the succinct representation
- Size: o(log*L*) bits
- The index should allow queries/operations on the succinct representation in (almost) the same time complexity as using a conventional data structure
  - This is the aim anyway
- Computational model is the word RAM
  - Assume word length w = loglogL
  - (this is the same pointer size as conventional data structures)
  - read/write w bits of memory in O(1) time
  - arithmetic/logical operations on w bit numbers take O(1) time
  - +,-,*,/,log,&,|,!,>>,<<

credit to Simon J. Puglisi, University of Helsinki

Binary rank and select

- The ability to answer *rank* and *select* queries over bit vectors (binary strings, bit arrays) is essential for implementing succinct data structures

- Given a binary string B[1..n]
  - $rank_B(i)$ returns the number of 1 bits in B[1..i]
  - $select_B(i)$ returns the position of the i^{th} 1 bit in B

credit to Simon J. Puglisi, University of Helsinki

Naïve rank

- To answer rank(i) scan B[1..i] and count 1-bits
- Simple but slow
  - O(i) time = O(n) time in the worst case
- How can we do better?
  - After all, what are we?

(Slightly) Less naïve rank

- Store an table A[1..n], containing the rank answers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0
A	1	1	1	2	2	3	4	5	5	6	6	6	7	7	8	8

- A[i] = rank(i)
  - Now rank(i) takes constant time – just an array lookup!

- Drawback:
  - A requires nlogn bits - logn times the size of B – not succinct!
  - We'd like a solution with O(1) queries and o(n) extra space...

credit to Simon J. Puglisi, University of Helsinki

We want O(1) queries with o(n) extra bits...

- General approach will be to precompute some tables

- Each table stores part of the answer to every query
  - For any given query, we can extract needed parts in O(1) time
  - The total size of the tables is o(n) bits

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

- Premise:
  - Can read O(logn) bits into an integer in range 1..n in O(1) time
  - However, to inspect each of those bits take O(logn) time

Tables : Superblocks

- Divide B into superblocks of size $s = \log^2 n/2 = 4*4/2 = 8$

- Build a small table R_s containing ranks for only some positions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0	1
R_s	0	5

- Store in $R_s[j] = \text{rank}_B(j*s)$, for all $0 \le j < n/s$

credit to Simon J. Puglisi, University of Helsinki

Tables : Blocks

- Divide each superblock into blocks of size b = logn/2 = 2

- Build a table R_b which contains the rank from the start of each block to the start of its superblock

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0	1
R_s	0	5

	0	1	2	3	0	1	2	3
R_b	0	1	2	3	0	1	1	2

- Store $R_b[k/b] = rank_B(k^*s) - rank_B(j^*s)$, for all $0 \leq k < n/b$

credit to Simon J. Puglisi, University of Helsinki

Intermission

- What we have so far (tables R_s and R_b) <u>almost</u> gets us the answer we're after

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0	1
R_s	0	5

	0	1	2	3	0	1	2	3
R_b	0	1	2	3	0	1	1	2

- $\text{rank}_B(i) \approx R_s[i/s] + R_b[i/b]$
  - Just need to answer in-block queries in O(1) time

credit to Simon J. Puglisi, University of Helsinki

Tables : Resolving in-block queries

- Solution? Use another table!

- Blocks have size $b = \log_2 n/2$
  - There are 2^b such blocks possible
  - In each block there are b possible rank queries
  - Each answer (relative to the block) is in the range 1..b

R_p

Type	0	1	rank(0)	rank(1)
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	2

credit to Simon J. Puglisi, University of Helsinki

Final Data Structure

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0	1
R_s	0	5

	0	1	2	3	0	1	2	3
R_b	0	1	2	3	0	1	1	2

R_p

Type	0	1	rank(0)	rank(1)
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	2

credit to Simon J. Puglisi, University of Helsinki

Size of table for within-block queries

- Blocks have size $b = \log_2 n/2$
  - There are 2^b such blocks possible
  - In each block there are b possible rank queries
  - Each answer (relative to the block) is in the range 1..b

R_p

Type	0	1	rank(0)	rank(1)
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	2

- Therefore size of R_p, the in-block data structure is
  - $2^b * b * \log b = n^{1/2} * \log n * \log\log n/2$ bits = o(n) bits

credit to Simon J. Puglisi, University of Helsinki

Summing up sizes…

- The size of R_s, the superblock data structure is
  - $2n/\log^2 n$ superblocks, each of size $\log n$ bits
  - $(n/\log^2 n) * \log n = 2n/\log n$ bits $= o(n)$ bits
- The size of R_b, the block data structure is
  - $2n/\log n$ blocks, each of size $\log\log n$ bits
  - $2n\log\log n/\log n$ bits $= o(n)$ bits
- $R_s + R_b + R_p = o(n)$ extra bits for $O(1)$ time rank queries
  - It is possible to construct this data structure in $O(n)$ time

credit to Simon J. Puglisi, University of Helsinki

Variations

- Just store R_s + use manual counting within superblocks
  - Saves space for R_b and R_p, takes time $O(\log^2 n)$ per query
- Store R_s and R_b + use manual counting within blocks
  - Saves only space for R_p, takes time $O(\log n)$ per query
- Use different superblock & block sizes
  - No more theoretical guarantees, but…
  - Perhaps faster in practice: blocks that are multiples of word sizes (32-bits) can be faster to handle

Summary of rank

- Rank index takes O(nloglogn/logn) = o(n) bits so we use n + o(n) overall and can answer queries in O(1) time

- While it is sublinear, we'd still like the o(n) term to be small
  – Best is by Patrastcu: O(n/log^{k}n) bits, O(k) time queries

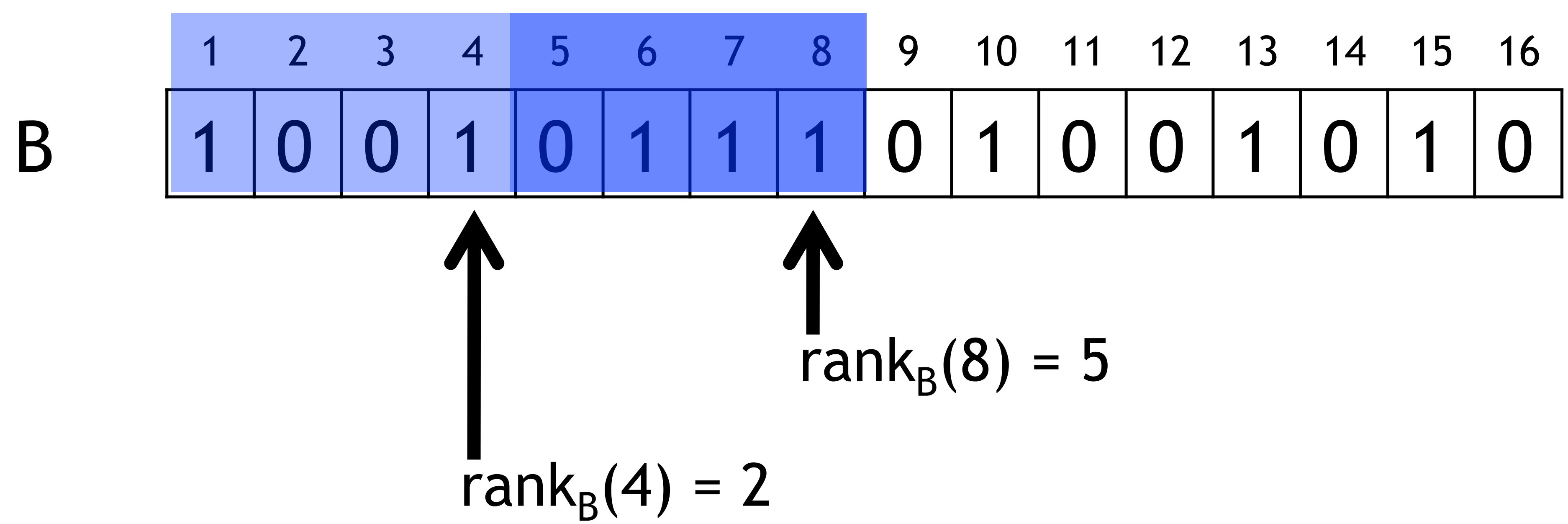
- Dynamic solutions exist
  – Queries no longer constant: O(logn/loglogn) time (Raman et al.)

Relationship to select(i)

- We can use our solution to rank to get a (fairly) efficient solution to select(i), with this observation:

- If rank(n/2) > i, then the i^{th} 1-bit is in B[1..n/2]
  - Otherwise it is in B[n/2+1..n]

$select_B(3)$



credit to Simon J. Puglisi, University of Helsinki

Relationship to select(i)

- Applying this idea recursively to arrive at select(i)
  - $O(\log_2 n)$ time, $o(n)$ space

- $O(1)$ time, $o(n)$ space solutions for select also exist
  - Slightly more complicated than $O(1)$ rank
  - (Munro and Clark)

- Similar variations as we discussed with rank (trading space for query time) are also possible

Information Systems 73 (2018) 25–34

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

ELSEVIER

Rank and select: Another lesson learned

Szymon Grabowski*, Marcin Raniszewski

Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, Łódź 90–924, Poland

credit to Simon J. Puglisi, University of Helsinki

These rank & select operations work over a binary alphabet — can be extended

High-Order Entropy-Compressed Text Indexes

Roberto Grossi* Ankur Gupta† Jeffrey Scott Vitter‡

Introduces the idea of the wavelet tree, a versatile index that can be extended to arbitrary alphabets. We'll discuss the simplest of variants according to the exposition of:

Wavelet Trees for All *

Gonzalo Navarro

Dept. of Computer Science, University of Chile. `gnavarro@dcc.uchile.cl`

Grossi, Roberto, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes." *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2003.

Preliminaries

$S[1,n] = s_1 s_2 \ldots s_n$ is a sequence of symbols where s_i in Σ

$\Sigma = [1 \ldots \sigma]$ is an alphabet of symbols

Representing S requires $n * \lceil \lg \sigma \rceil = n * \lg \sigma + O(n)$ bits.

Wavelet tree: balanced binary tree with σ nodes, where each subtree is also a wavelet tree (i.e. it is recursive)

Preliminaries

Structure. A wavelet tree [54] for sequence $S[1, n]$ over alphabet $[1..\sigma]$ can be described recursively, over a sub-alphabet range $[a..b] \subseteq [1..\sigma]$. A wavelet tree over alphabet $[a..b]$ is a binary balanced tree with $b - a + 1$ leaves. If $a = b$, the tree is just a leaf labeled a. Else it has an internal root node, v_{root}, that represents $S[1, n]$. This root stores a bitmap $B_{v_{root}}[1, n]$ defined as follows: if $S[i] \leq (a + b)/2$ then $B_{v_{root}}[i] = 0$, else $B_{v_{root}}[i] = 1$. We define $S_0[1, n_0]$ as the subsequence of $S[1, n]$ formed by the symbols $c \leq (a+b)/2$, and $S_1[1, n_1]$ as the subsequence of $S[1, n]$ formed by the symbols $c > (a+b)/2$. Then, the left child of v_{root} is a wavelet tree for $S_0[1, n_0]$ over alphabet $[a..\lfloor(a+b)/2\rfloor]$ and the right child of v_{root} is a wavelet tree for $S_1[1, n_1]$ over alphabet $[1 + \lfloor(a + b)/2\rfloor..b]$.

Preliminaries

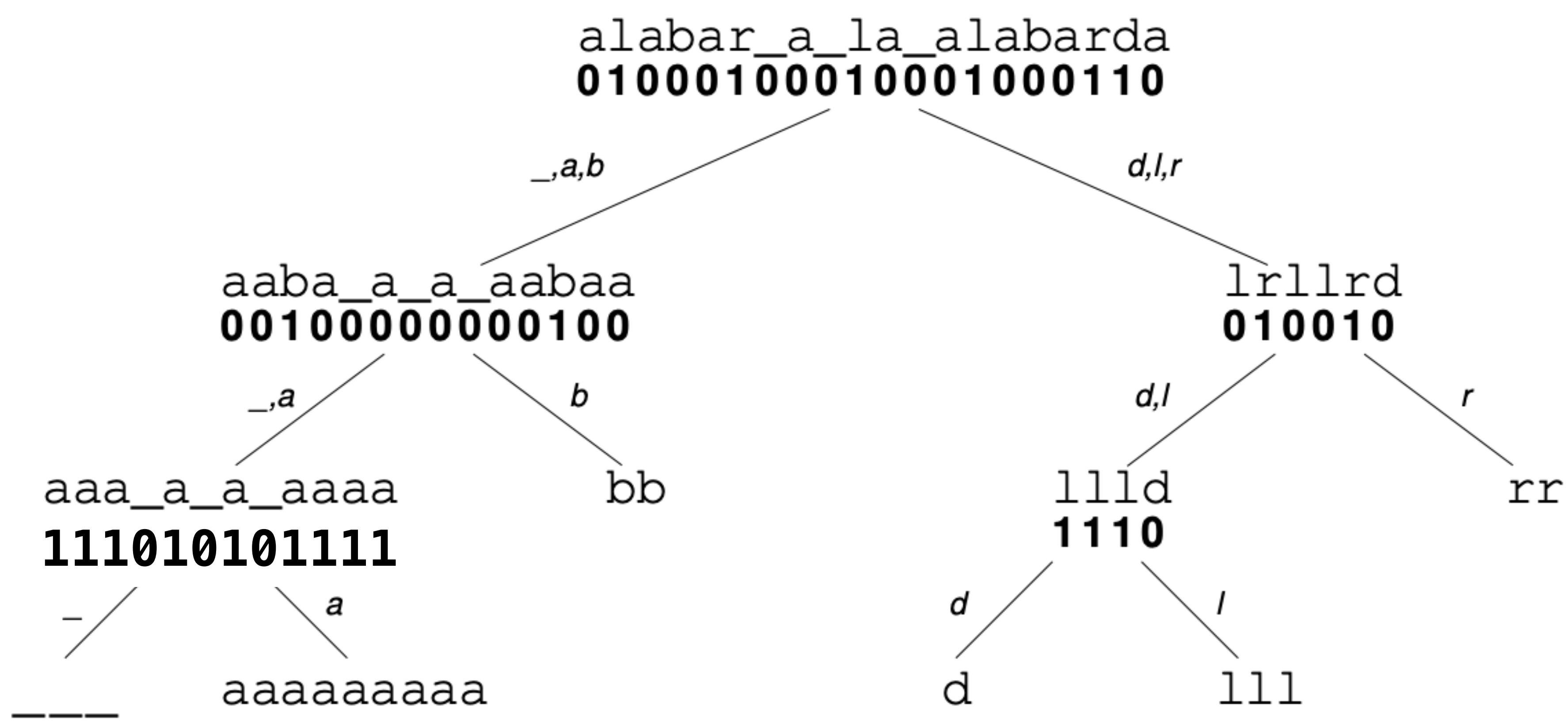
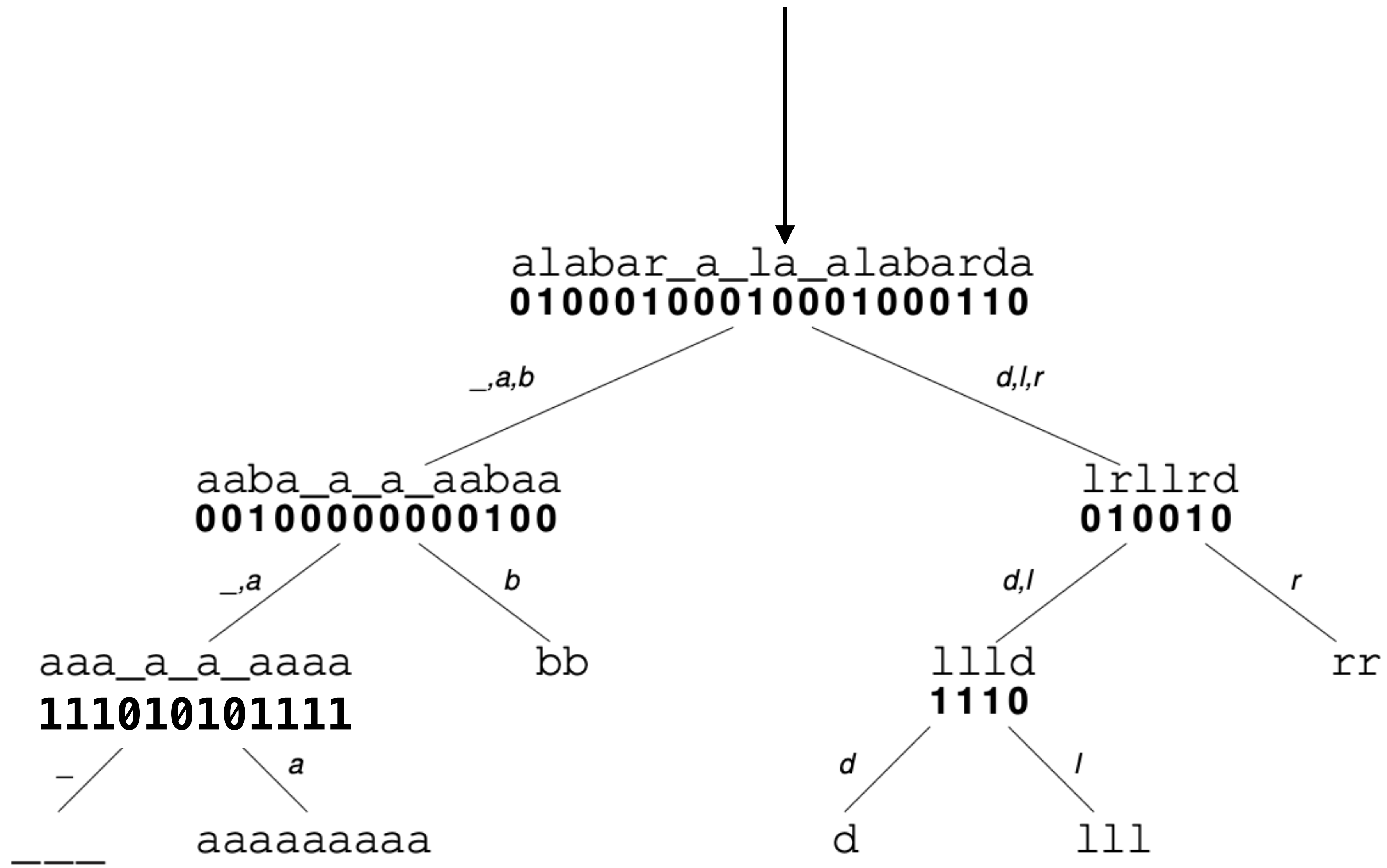


Fig. 1. A wavelet tree on string S = "**alabar a la alabarda**". We draw the spaces as underscores. The subsequences of S and the subsets of Σ labeling the edges are drawn for illustration purposes; the tree stores only the topology and the bitmaps.

Example: Rank

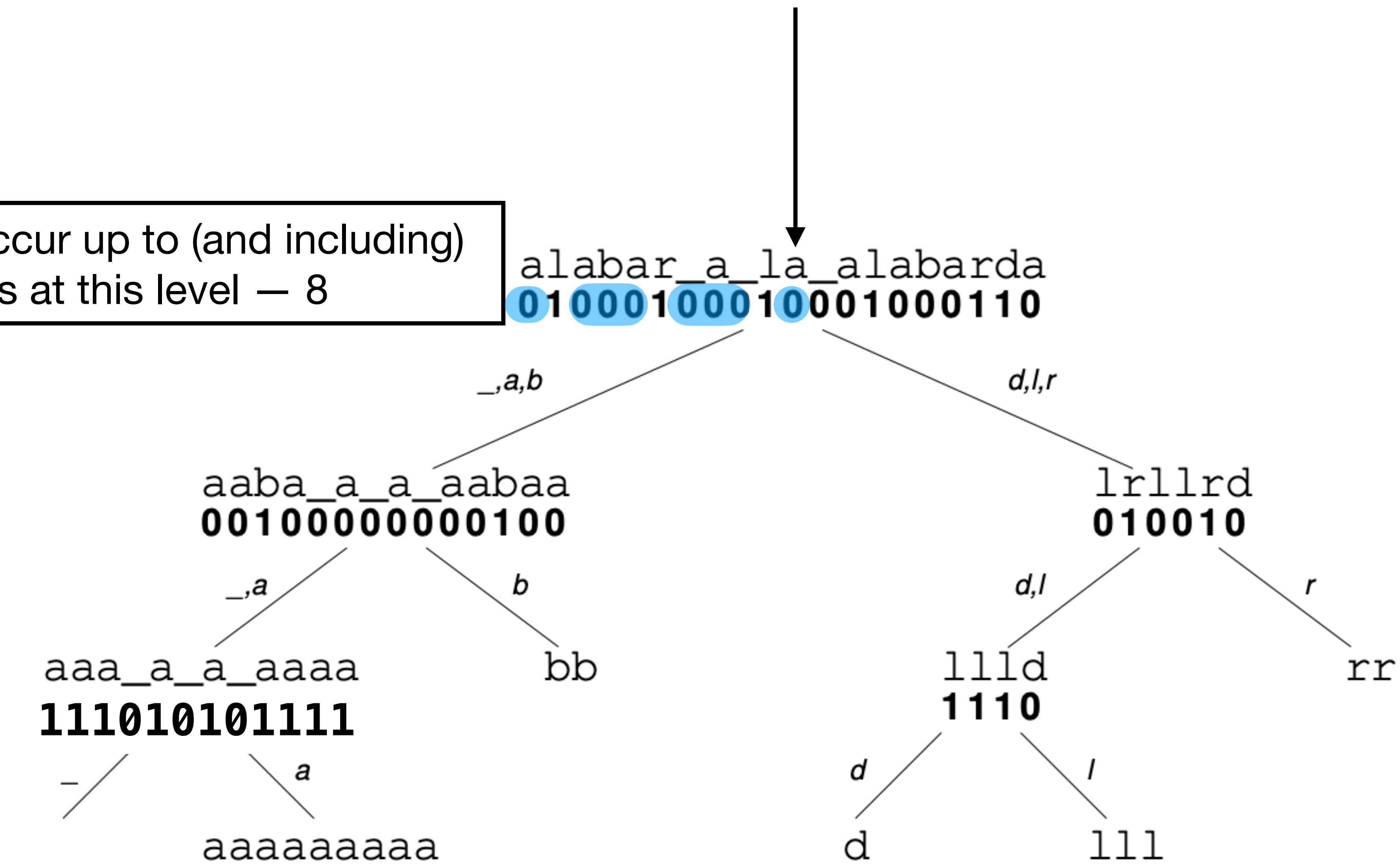
Consider asking for the rank of this “a”



Example: Rank

Consider asking for the rank of this “a”

how many _,a,b occur up to (and including) this one? Count 0’s at this level — 8

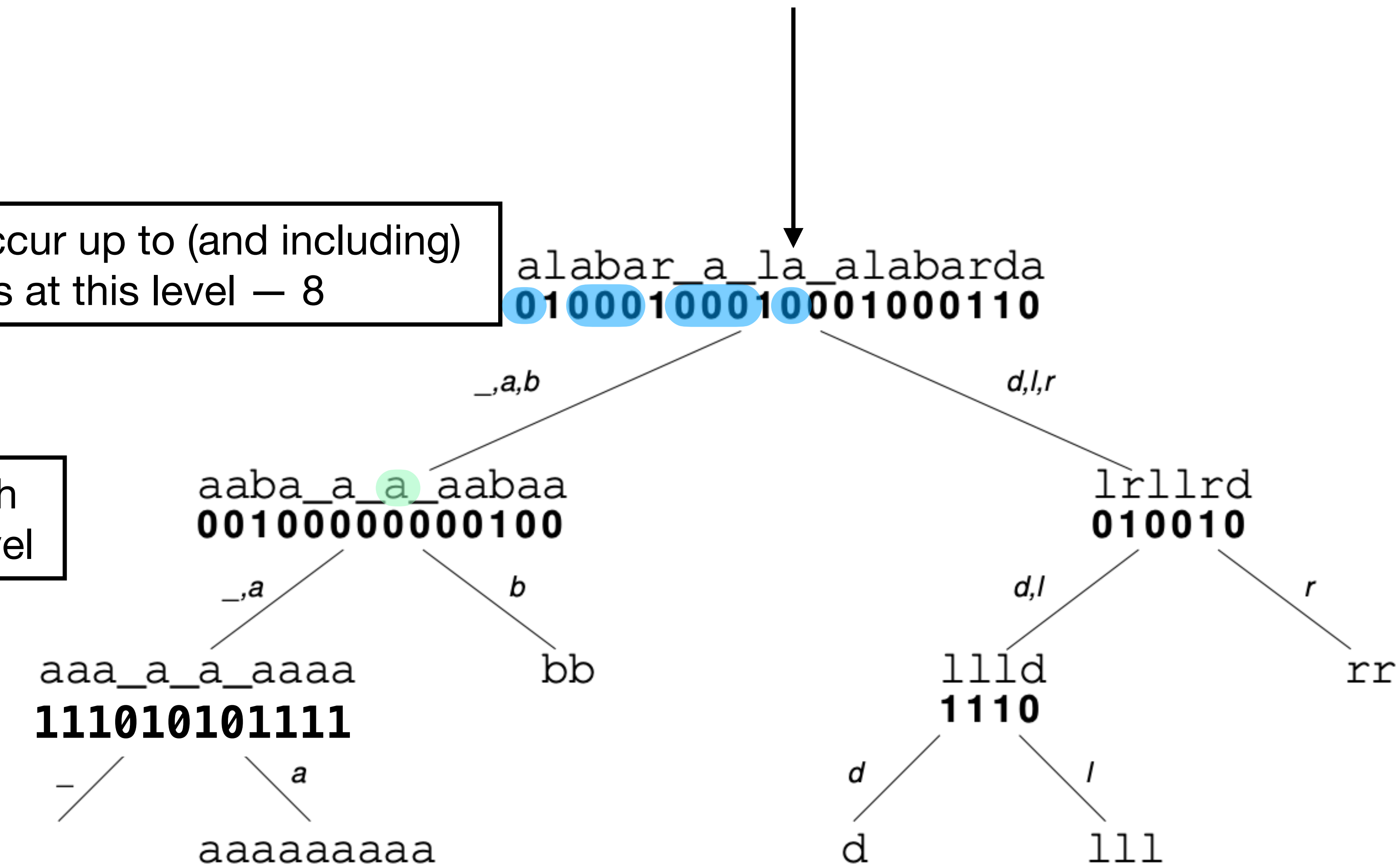


Example: Rank

Consider asking for the rank of this "a"

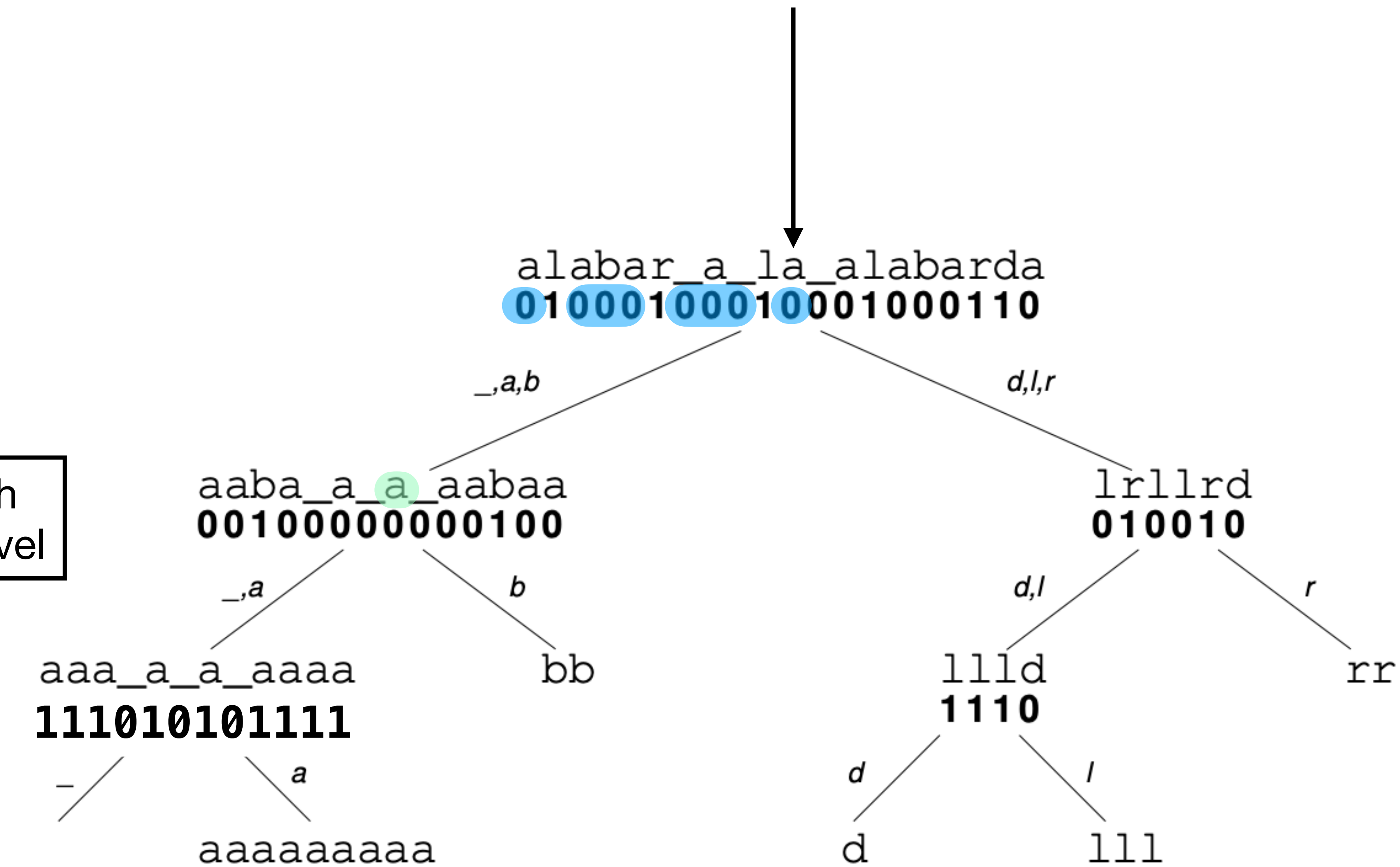
how many _,a,b occur up to (and including) this one? Count 0's at this level — 8

So that maps to 8th characterat this level



Example: Rank

Consider asking for the rank of this “a”

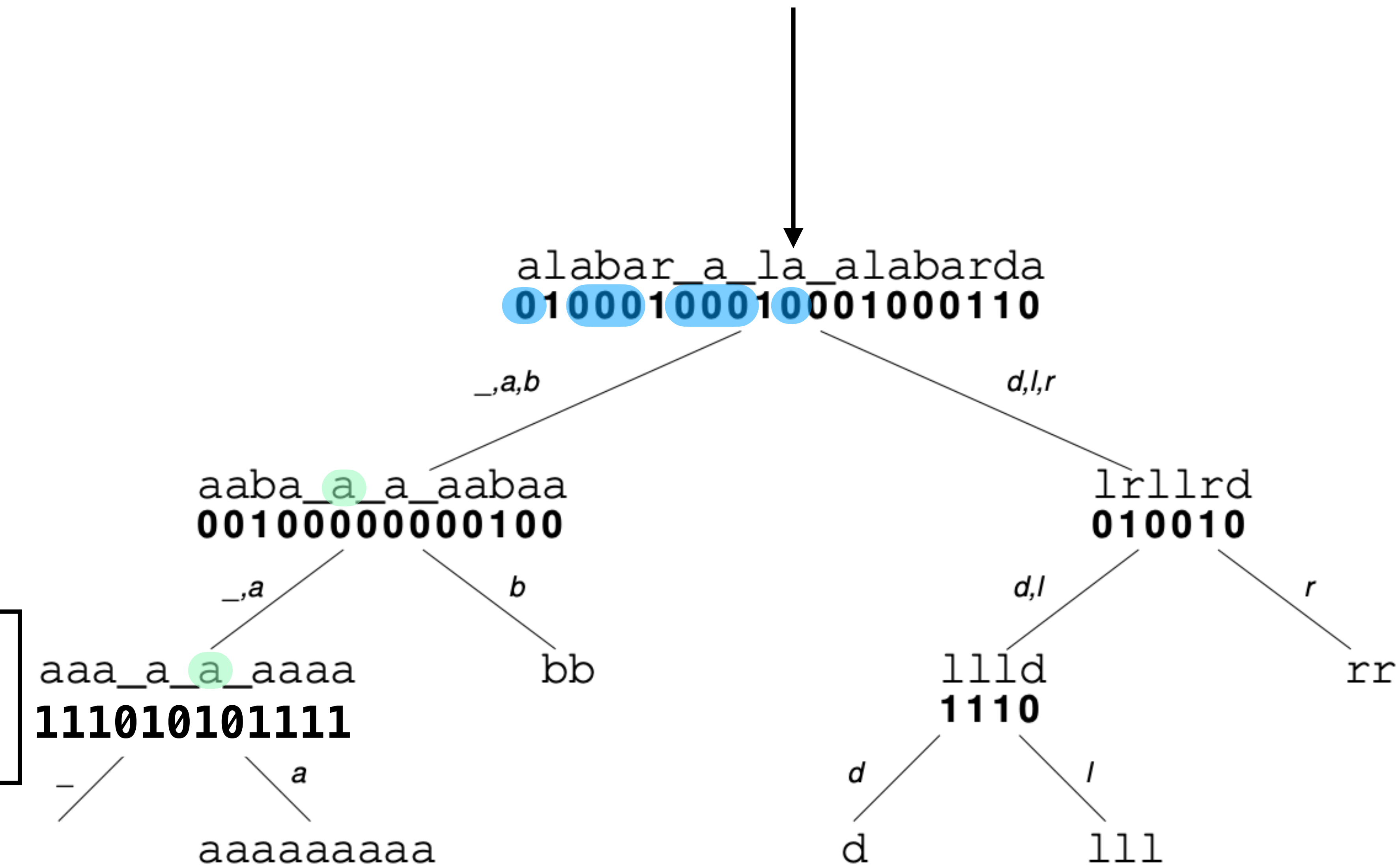


So that maps to 8th character at this level

Count 0’s at this level — 7

Example: Rank

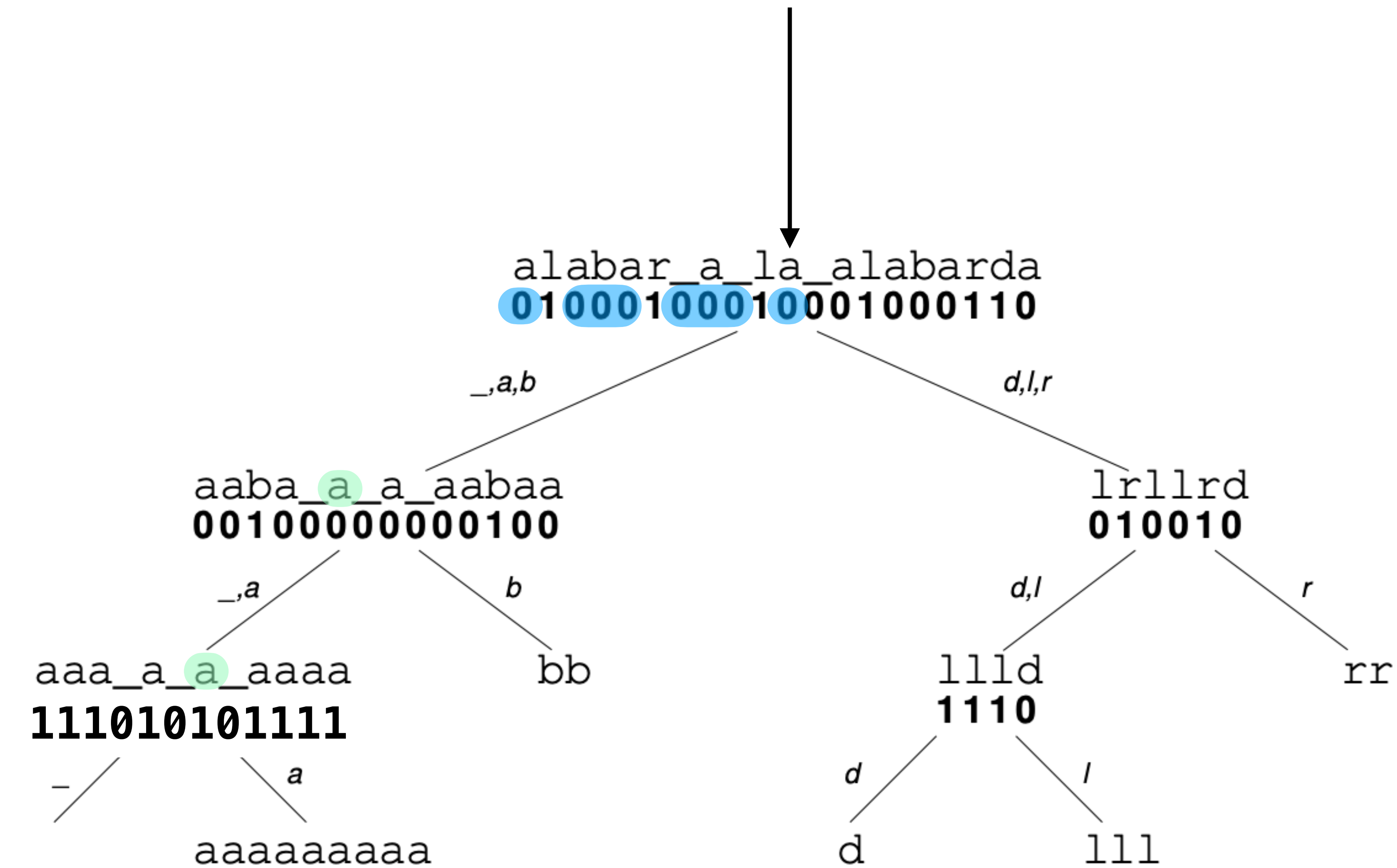
Consider asking for the rank of this "a"



So that maps to 7th character at this level

Example: Rank

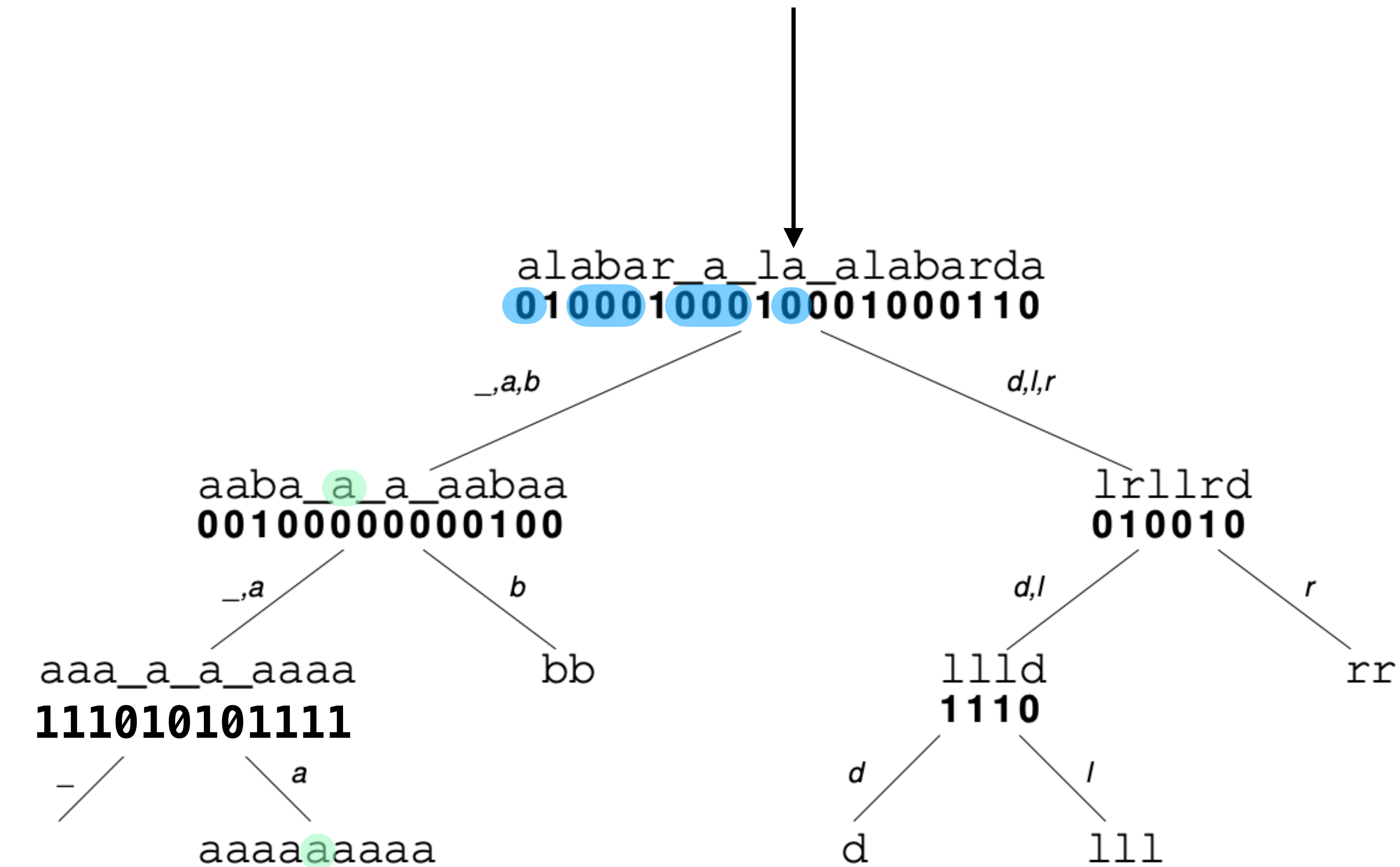
Consider asking for the rank of this “a”



Count 1’s at this level — 5

Example: Rank

Consider asking for the rank of this “a”

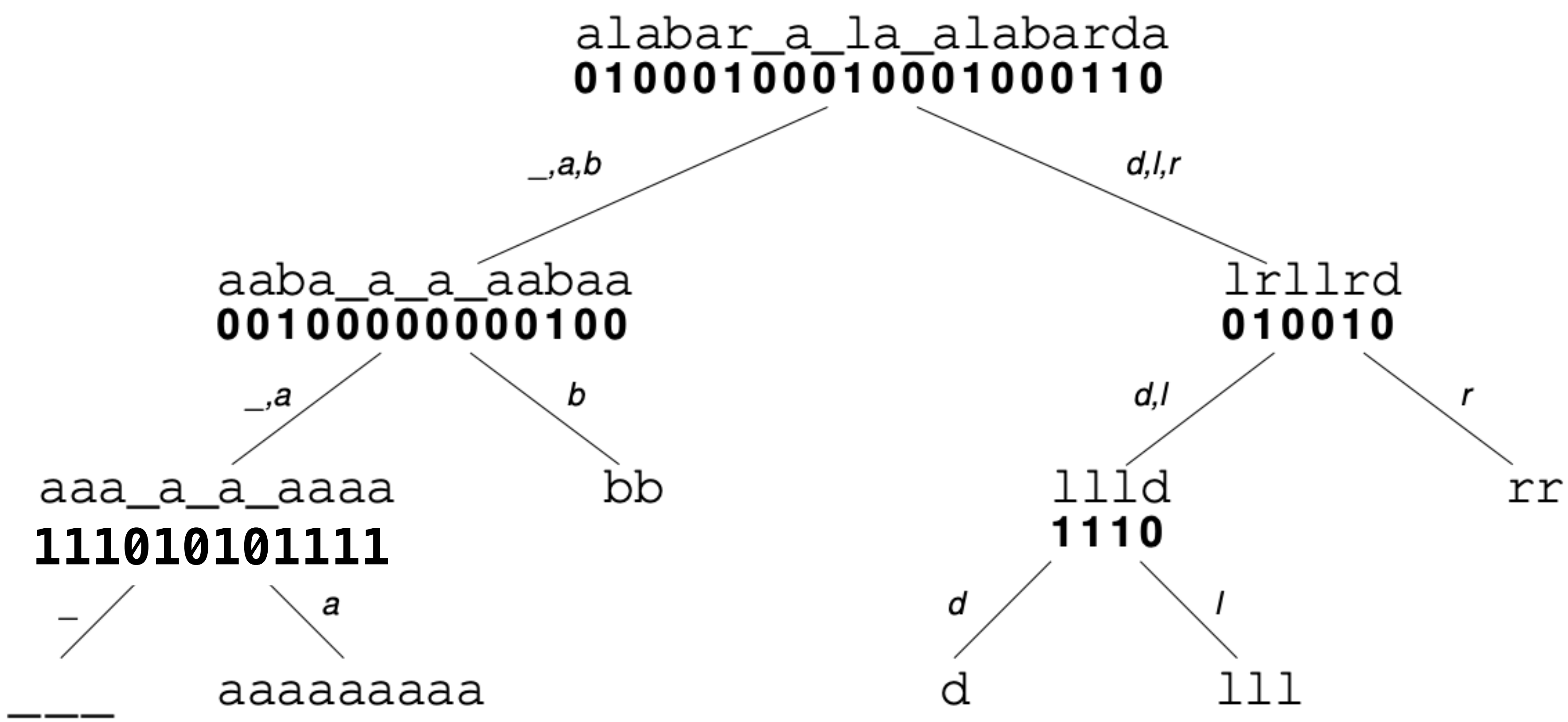


This is the 5th ‘a’
— $\text{Rank}_a(S, 11) = 5$

If we are 1-indexing

Example: Rank

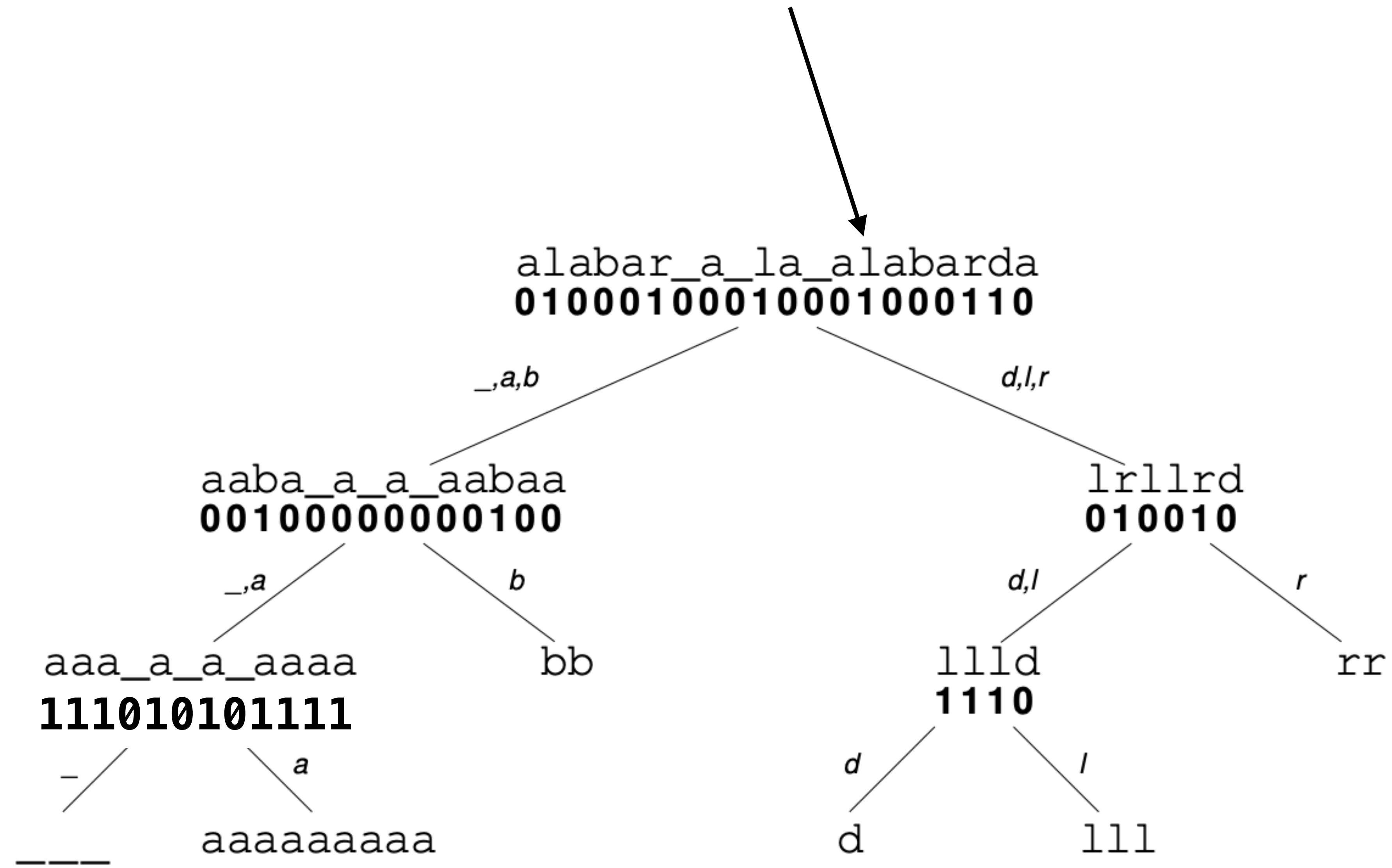
This procedure turns rank for any character in the alphabet into lg σ rank calculations over bitvectors.



We can answer rank queries for an arbitrary character in lg σ * O(1) = O(lg σ) time. For small, constant alphabets, through the magic of Big-O, this is *constant time*. :)

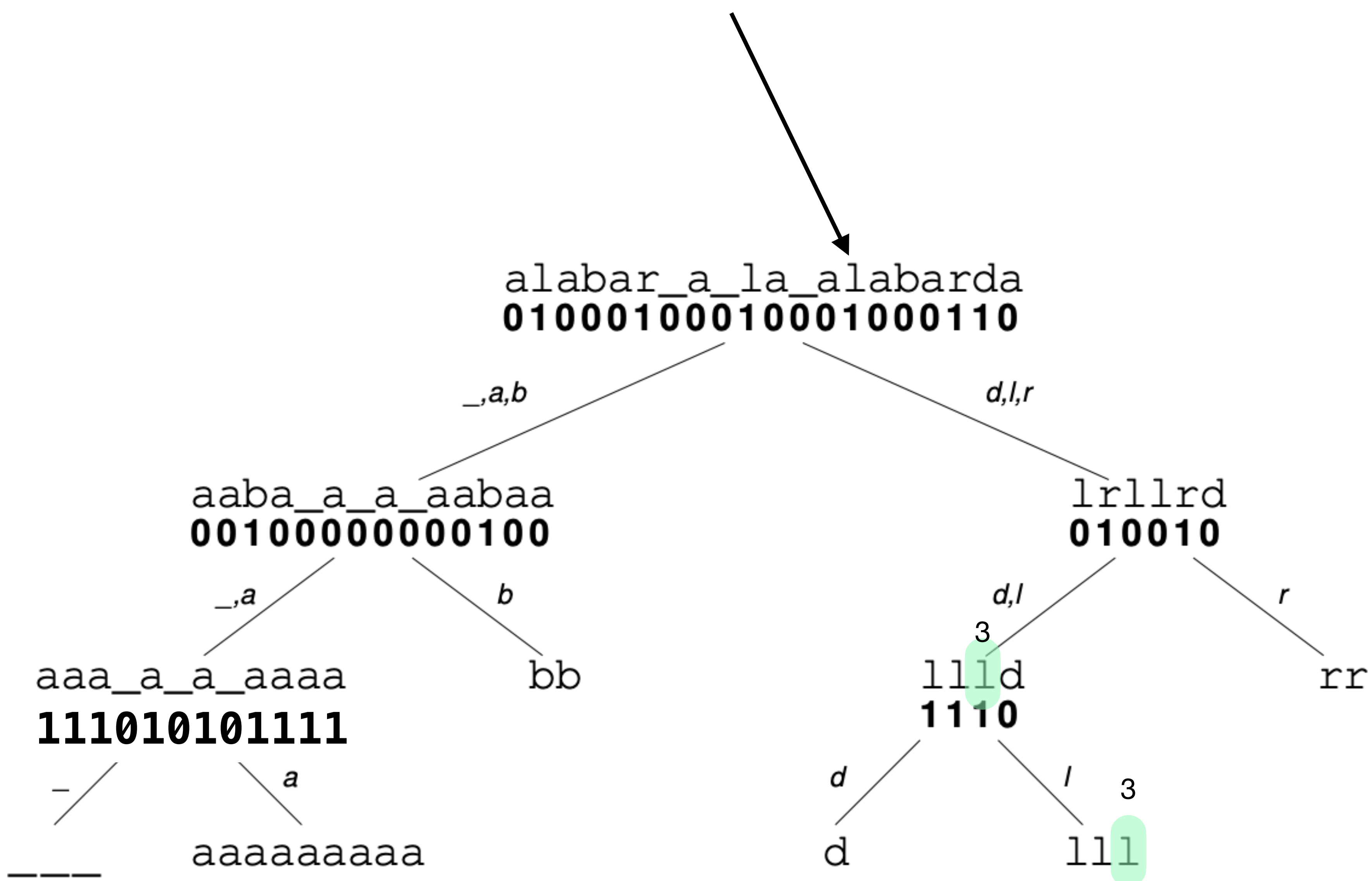
Example: Select

Select the 3rd “l” (at what index does it occur?)



Example: Select

Select the 3rd “l” (at what index does it occur?)

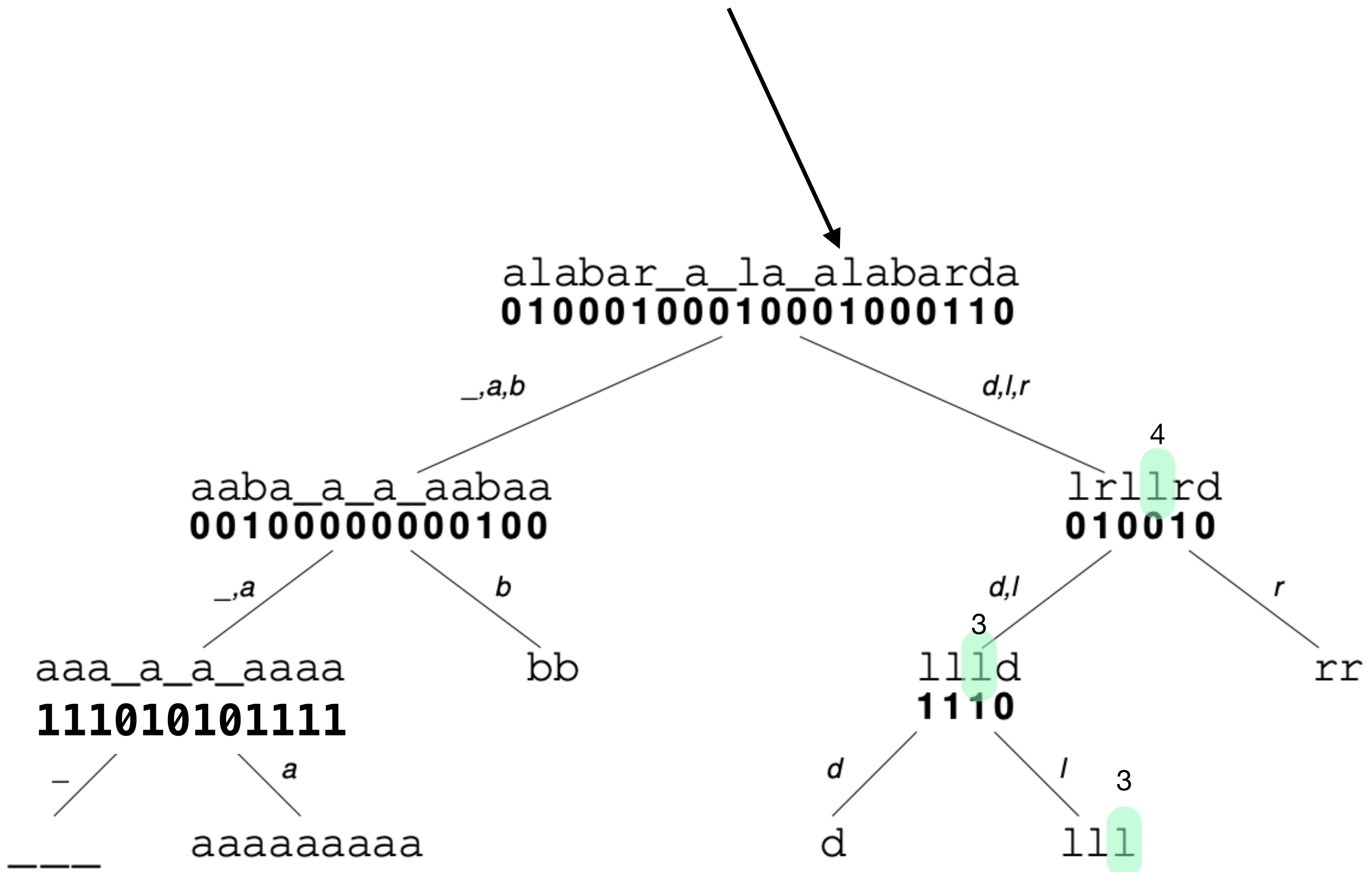


Where does this”l” go at this level? It’s the 3rd 1

Here, we start at the bottom of the tree and work up.

Example: Select

Select the 3rd “1” (at what index does it occur?)

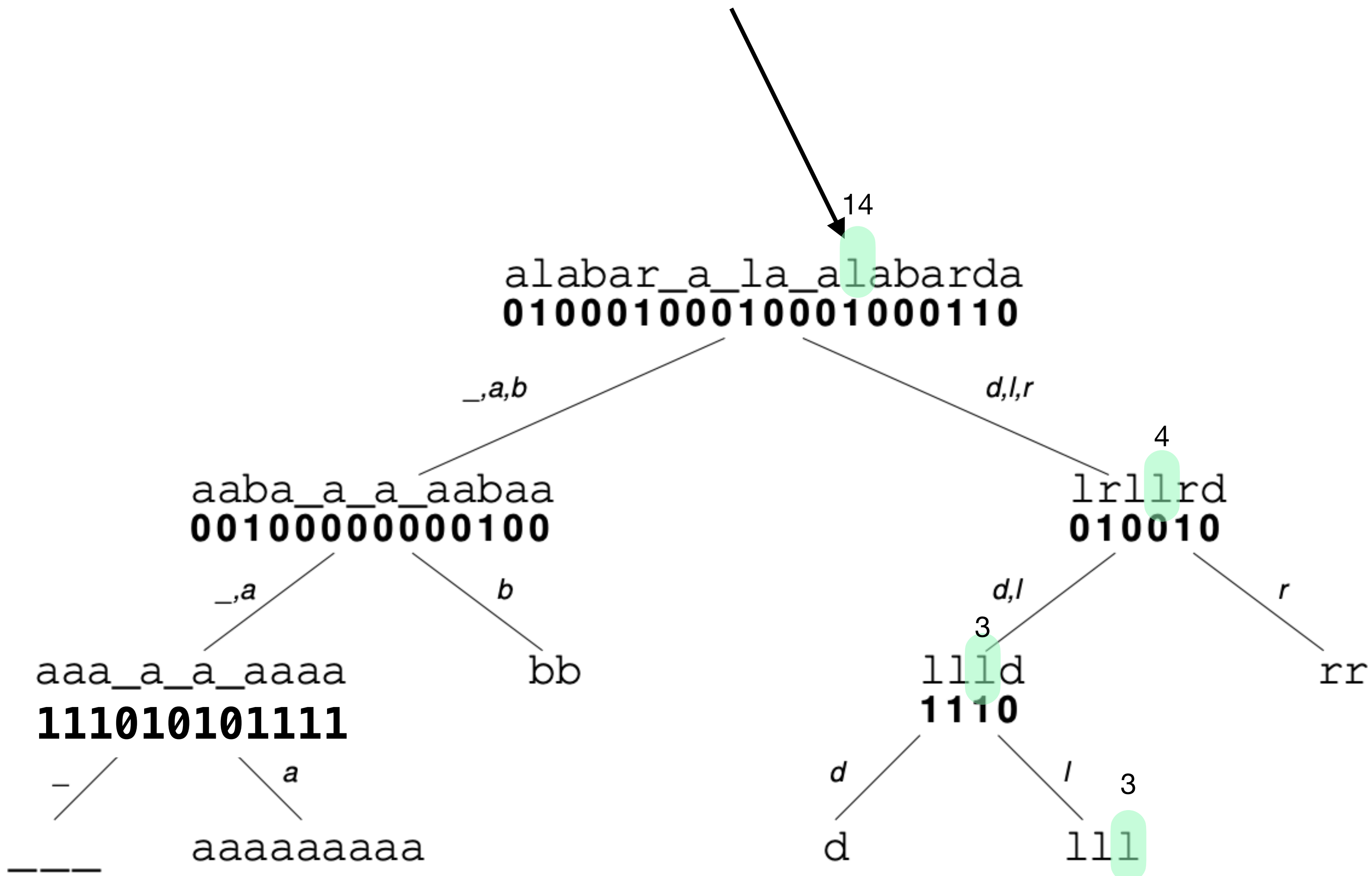


Where does this”l” go at this level? It’s the 3rd 0

Here, we start at the bottom of the tree and work up.

Example: Select

Select the 3rd “1” (at what index does it occur?)

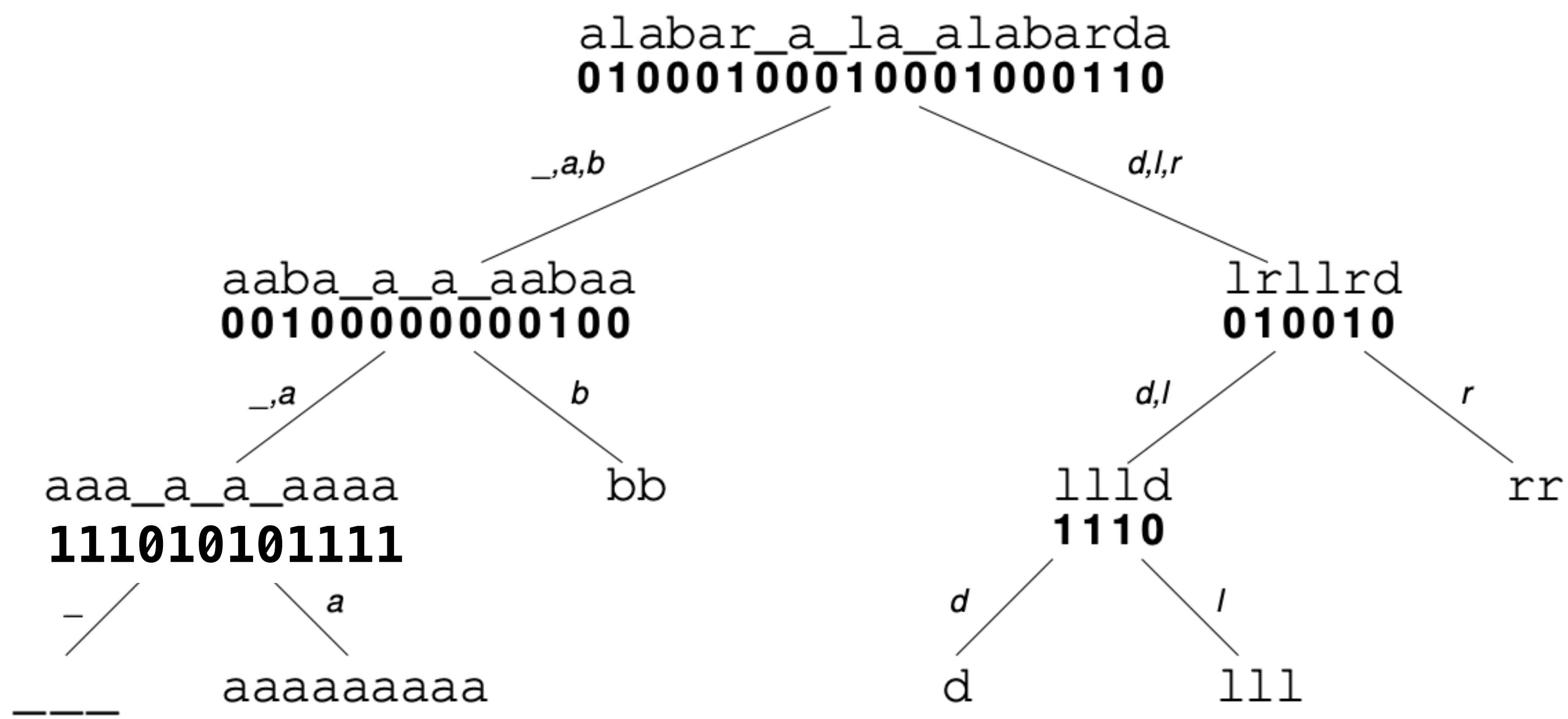


Where does this”l” go at this level? It’s the 4th 1

Here, we start at the bottom of the tree and work up.

Example: Select

This procedure turns select for any character in the alphabet into lg σ select calculations over bitvectors.



We can answer select queries for an arbitrary character in lg σ * O(1) = O(lg σ) time. For small, constant alphabets, through the magic of Big-O, this is *constant time*. :)

Succinct Data Structures

We have only scratched the surface on what is possible with rank & select and succinct data structures in general.

However, we'll assume familiarity with rank and select moving forward as we talk about data structures in Comp Bio that use them.

Gonzalo Navarro alone publishes 14-24 papers / year in this field :):

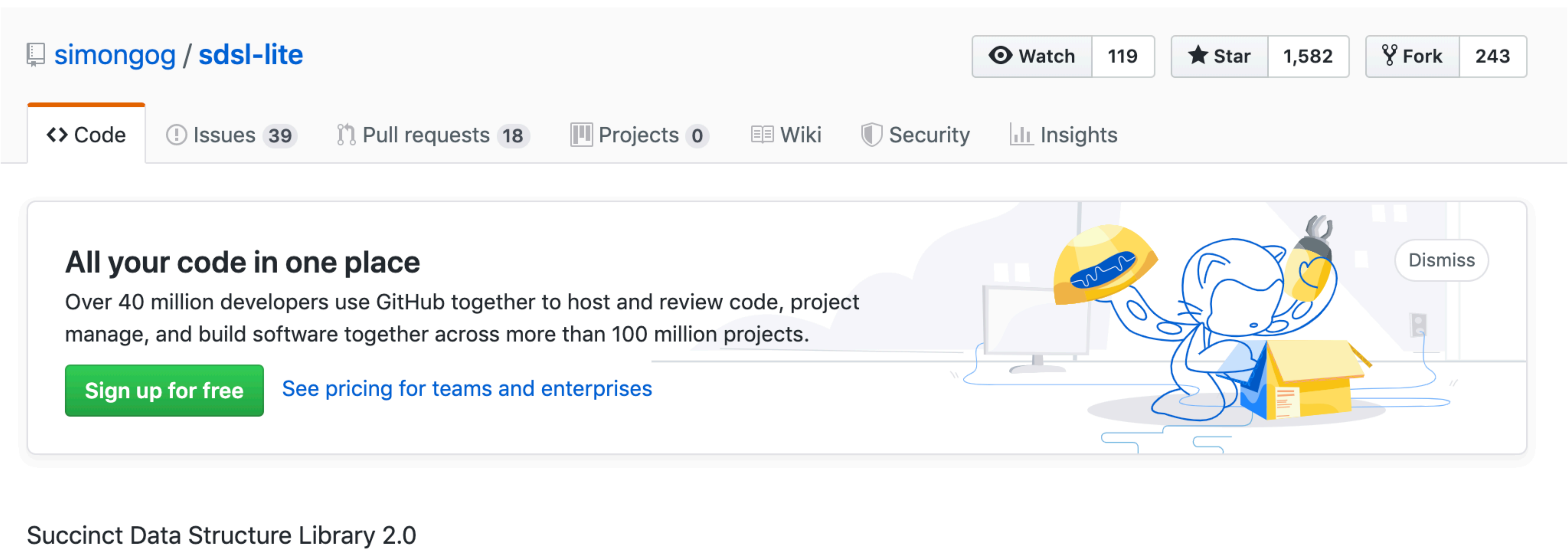
2019 (21)	**2018 (14)**
2017 (21)	**2016 (19)**

A google search on Gonzalo, and succinct data structures will send you down a wonderful rabbit-hole; I recommend you try it!

Some practical advice

Succinct data structure papers tend to be quite theoretical (go figure!).

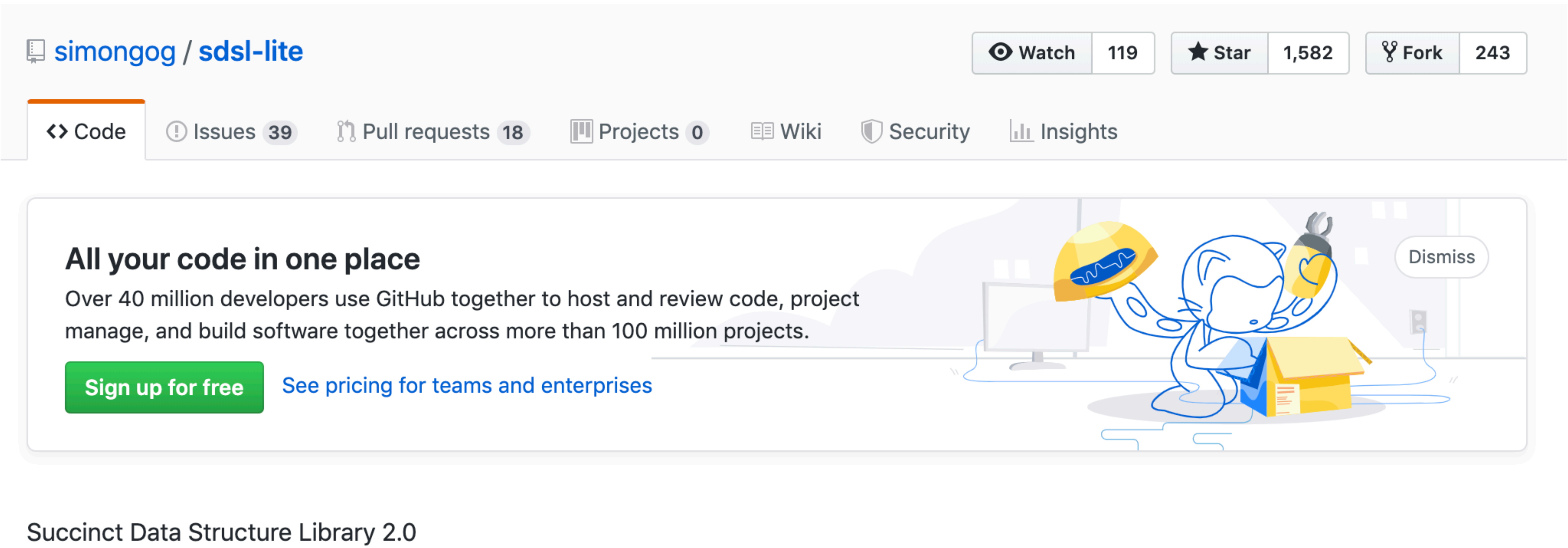
Luckily, there is a *go-to* library for implementation of these ideas.



Some practical advice

Succinct data structure papers tend to be quite theoretical (go figure!).

Luckily, there is a *go-to* library for implementation of these ideas.



Provides a modern, modular C++ implementation of many different succinct data structures.

Some practical advice

Create an FM-index over some text?

```
#include <sdsl/suffix_arrays.hpp>
#include <string>
#include <iostream>

string index_file   = string(argv[1])+index_suffix;
csa_wt<wt_huff<rrr_vector<127> >, 512, 1024> fm_index;

if (!load_from_file(fm_index, index_file)) {
    ifstream in(argv[1]);
    if (!in) {
        cout << "ERROR: File " << argv[1] << " does not exist. Exit." << endl;
        return 1;
    }
    cout << "No index "<<index_file<< " located. Building index now." << endl;
    construct(fm_index, argv[1], 1); // generate index
    store_to_file(fm_index, index_file); // save it
}
```

Some practical advice

Perform rank queries over a bit vector?

```
#include <iostream>
#include <sdsl/bit_vectors.hpp>

using namespace std;
using namespace sdsl;

int main()
{
    bit_vector b = bit_vector(8000, 0);
    for (size_t i=0; i < b.size(); i+=100)
        b[i] = 1;
    rank_support_v<1> b_rank(&b);
    for (size_t i=0; i<=b.size(); i+= b.size()/4)
        cout << "(" << i << ", " << b_rank(i) << ") ";
    cout << endl;
}
```

Some practical advice

How about $select_0$

```
#include <iostream>
#include <sdsl/bit_vectors.hpp>

using namespace std;
using namespace sdsl;

int main()
{
    bit_vector b = {0,1,0,1,1,1,0,0,0,1,1};
    size_t zeros = rank_support_v<0>(&b)(b.size());
    bit_vector::select_0_type b_sel(&b);

    for (size_t i=1; i <= zeros; ++i) {
        cout << b_sel(i) << " ";
    }
    cout << endl;
}
```

Some practical advice

You get the idea! An incredibly powerful library, at your fingertips.

sdsl Cheat Sheet

Data structures

The library code is in the `sdsl` namespace. Either import the namespace in your program (`using namespace sdsl;`) or qualify all identifieres by a `sdsl::`-prefix.

Each section corresponds to a header file. The file is hyperlinked as part of the section heading.

We have two types of data structures in sdsl. *Self-contained* and *support* structures. A support object `s` can extend a self-contained object `o` (e.g. add functionality), but requires access to `o`. Support structures contain the substring `support` in their class names.

Integer Vectors (IV)

The core of the library is the class `int_vector<w>`. Parameter w corresponds to the fixed length of each element in bits. For $w = 8, 16, 32, 64, 1$ the length is fixed during compile time and the vectors correspond to `std::vector<uintw_t>` resp. `std::vector<bool>`. If $w = 0$ (default) the length can be set during runtime. *Constructor:* `int_vector<>(n,x,ℓ)`, with n equals size, x default integer value, ℓ width of integer (has no effect for $w > 0$).

Public methods: `operator[i]`, `size()`, `width()`, `data()`.

Manipulating `int_vector<w> v`

Method	*Description*
`v[i]=x`	Set entry `v[i]` to x.
`v.width(ℓ)`	Set width to ℓ, if $w = 0$.
`v.resize(n)`	Resize `v` to n elements.
Useful methods in namespace `sdsl::util`:	
`set_to_value(v,k)`	Set `v[i]=k` for each i.
`set_to_id(v)`	Set `v[i]=i` for each i.
`set_random_bits(v)`	Set elements to random bits.
`mod(v,m)`	Set `v[i]=v[i]` mod m for each i.
`bit_compress(v)`	Gets $x = \max_i$ `v[i]` and $\ell = \lceil \log(x-1) \rceil + 1$ and packs the entries in ℓ-bit integers.
`expand_width(v,ℓ)`	Expands the width of each integer to ℓ bits, if $\ell \geq$ `v.width()`.

Compressed Integer Vectors (CIV)

For a vector `v`, `enc_vector` stores the self-delimiting coded deltas (`v[i+1]−v[i]`). Fast random access is achieved by sampling values of `v` at rate `t_dens`. Available coder are `coder::elias_delta`, `coder::elias_gamma`, and `coder::fibonacci`.

Class `vlc_vector` stores each `v[i]` as self-delimiting codeword. Samples at rate `t_dens` are inserted for fast random access.

Class `dac_vector` stores for each value x the least (`t_b` -1) significant bits plus a bit which is set if $x \geq 2^{b-1}$. In the latter case, the process is repeated with $x' = x/2^{b-1}$.

Bitvectors (BV)

Representations for a bitvector of length n with m set bits.

Class	*Description*	*Space*
`bit_vector`	plain bitvector	$64\lceil n/64+1 \rceil$
`bit_vector_il`	interleaved bitvector	$\approx n(1+64/K)$
`rrr_vector`	H_0-compressed bitvector	$\approx \lceil \log \binom{n}{m} \rceil$
`sd_vector`	sparse bitvector	$\approx m \cdot (2+\log \frac{n}{m})$
`hyb_vector`	hybrid bitvector	

`bit_vector` equals `int_vector<1>` and is therefore dynamic.
Public Methods: `operator[i]`, `size()`, `begin()`, `end()`
Public Types: `rank_1_type`, `select_1_type`, `select_0_type`[1].
Each bitvector can be constructed out of a `bit_vector` object.

Rank Supports (RS)

RSs add rank functionality to BV. Methods `rank(i)` and `operator(i)` return the number of set bits[2] in the prefix $[0..i)$ of the supported BV for $i \in [0, n]$.

Class	*Compatible BV*	*+Bits*	*Time*
`rank_support_v`	`bit_vector`	$0.25n$	$\mathcal{O}(1)$
`rank_support_v5`	`bit_vector`	$0.0625n$	$\mathcal{O}(1)$
`rank_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`rank_support_il`	`bit_vector_il`	128	$\mathcal{O}(1)$
`rank_support_rrr`	`rrr_vector`	80	$\mathcal{O}(k)$
`rank_support_sd`	`sd_vector`	64	$\mathcal{O}(\log \frac{n}{m})$
`rank_support_hyb`	`hyb_vector`	64	-

Call `util::init_support(rs,bv)` to initialize rank structure `rs` to bitvector `bv`. Call `rs(i)` to get `rank(i)` $= \sum_{k=0}^{k<i}$ `bv[k]`

Select Supports (SLS)

SLSs add select functionality to BV. Let m be the number of set bits in BV. Methods `select(i)` and `operator(i)` return the position of the i-th set bit[3] in BV for $i \in [1..m]$.

Class	*Compatible BV*	*+Bits*	*Time*
`select_support_mcl`	`bit_vector`	$\leq 0.2n$	$\mathcal{O}(1)$
`select_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`select_support_il`	`bit_vector_il`	64	$\mathcal{O}(\log n)$
`select_support_rrr`	`rrr_vector`	64	$\mathcal{O}(\log n)$
`select_support_sd`	`sd_vector`	64	$\mathcal{O}(1)$

Call `util::init_support(sls,bv)` to initialize `sls` to bitvector `bv`. Call `sls(i)` to get `select(i)` $= \min\{j \mid$ `rank(j+1)` $= i\}$.

Wavelet Trees (WT=BV+RS+SLS)

Wavelet trees represent sequences over byte or integer alphabets of size σ and consist of a tree of BVs. Rank and select on the sequences is reduced to rank and select on BVs, and the runtime is multiplied by a factor in $[H_0, \log \sigma]$.

Class	*Shape*	`lex_ordered`	*Default alphabet*	Traversable
`wt_rlmn`	underlying WT dependent			×
`wt_gmr`	none	×	integer	×
`wt_ap`	none	×	integer	×
`wt_huff`	Huffman	×	byte	✓
`wm_int`	Balanced	×	integer	✓
`wt_blcd`	Balanced	✓	byte	✓
`wt_hutu`	Hu-Tucker	✓	byte	✓
`wt_int`	Balanced	✓	integer	✓

Public types: `value_type`, `size_type`, and `node_type` (if WT is traversable). In the following let c be a symbol, i,j,k, and q integers, v a node, and r a range.
Public methods: `size()`, `operator[i]`, `rank(i,c)`, `select(i,c)`, `inverse_select(i)`, `begin()`, `end()`.
Traversable WTs provide also: `root()`, `is_leaf(v)`, `empty(v)`, `size(v)`, `sym(v)`, `expand(v)`, `expand(v,r)`, `expand(v,std::vector<r>)`, `bit_vec(v)`, `seq(v)`.
`lex_ordered` WTs provide also: `lex_count(i,j,c)` and `lex_smaller_count(i,c)`. `wt_int` provides: `range_search_2d`.
wt_algorithm.hpp contains the following generic WT method (let *wt* be a WT object): `intersect(wt, vector<r>)`, `quantile_freq(wt,i,j,q)`, `interval_symbols(wt,i,j,k,...)`, `symbol_lte(wt,c)`, `symbol_gte(wt,c)`, `restricted_unique_range_values(wt,x_i,x_j,y_i,y_j)`.

Suffix Arrays (CSA=IV+WT)

Compressed suffix arrays use CIVs or WTs to represent the suffix arrays (SA), its inverse (ISA), BWT, Ψ, and LF. CSAs can be built over byte and integer alphabets.

Class	*Description*
`csa_bitcompressed`	Based on SA and ISA stored in a IV.
`csa_sada`	Based on Ψ stored in a CIV.
`csa_wt`	Based on the BWT stored in a WT.

Public methods: `operator[i]`, `size()`, `begin()`, `end()`.
Public members: `isa`, `bwt`, `lf`, `psi`, `text`, `L`, `F`, `C`, `char2comp`, `comp2char`, `sigma`.
Policy classes: alphabet strategy (e.g. `byte_alphabet`, `succinct_byte_alphabet`, `int_alphabet`) and SA sampling strategy (e.g. `sa_order_sa_sampling`, `text_order_sa_sampling`)

Longest Common Prefix (LCP) Arrays

Class	*Description*
`lcp_bitcompressed`	Values in a `int_vector<>`.
`lcp_dac`	Direct accessible codes used.
`lcp_byte`	Small values in a byte; 2 words per large.
`lcp_wt`	Small values in a WT; 1 word per large.
`lcp_vlc`	Values in a `vlc_vector`.
`lcp_support_sada`	Values stored permuted. CSA needed.
`lcp_support_tree`	Only depths of CST inner nodes stored.
`lcp_support_tree2`	+ large values are sampled using LF.

Public methods: `operator[i]`, `size()`, `begin()`, `end()`

Balanced Parentheses Supports (BPS)

We represent a sequence of parentheses as a `bit_vector`. An opening/closing parenthesis corresponds to `1`/`0`.

Class	*Description*
`bp_support_g`	Two-level pioneer structure.
`bp_support_gg`	Multi-level pioneer structure.
`bp_support_sada`	Min-max-tree over excess sequence.

Public methods: `find_open(i)`, `find_close(i)`, `enclose(i)`, `double_enclose(i,j)`, `excess(i)`, `rr_enclose(i,j)`, `rank(i)`[4], `select(i)`.
Call `util::init_support(bps,bv)` to initialize a BPS `bps` to `bit_vector bv`.

Some practical advice

You get the idea! An incredibly powerful library, at your fingertips.

Suffix Trees (CST=CSA+LCP+BPS)

A CST can be parametrized by any combination of CSA ,LCP, and BPS. The operation of each part can still be accessed through member varaibles. The additional operations are listed below. CSTs can be built for byte or integer alphabets.

Class	*Description*
`cst_sada`	Represents a node as position in BPS. Navigational operations are fast (they are directly translated in BPS operations on the DFS-BPS). Space: $4n+o(n)+\|CSA\|+\|LCP\|$ bits.
`cst_sct3`	Represents nodes as intervals. Fast construction, but slower navigational operations. Space: $3n+o(n)+\|CSA\|+\|LCP\|$

Public types: `node_type`. In the following let v and w be nodes and i, d, lb, rb integers.
Public methods: `size()`, `nodes()`, `root()`, `begin()`, `end()`, `begin_bottom_up()`, `end_bottom_up`, `size(`v`)`, `is_leaf(`v`)`, `degree(`v`)`, `depth(`v`)`, `node_depth(`v`)`, `edge(`v, d`)`, `lb(`v`)`, `rb(`v`)`, `id(`v`)`, `inv_id(`i`)`, `sn(`v`)`, `select_leaf(`i`)`, `node(`lb, rb`)`, `parent(`v`)`, `sibling(`v`)`, `lca(`v, w`)`, `select_child(`v, i`)`, `child(`v, c`)`, `children(`v`)`, `sl(`v`)`, `wl(`v, c`)`, `leftmost_leaf(`v`)`, `rightmost_leaf(`v`)`
Public members: `csa`, `lcp`.
The traversal example shows how to use the DFS-iterator.

Range Min/Max Query (RMQ)

A RMQ `rmq` can be used to determine the position of the minimum value[5] in an arbitrary subrange $[i, j]$ of an preprocessed vector `v`. Operator `operator(`i,j`)` returns $x = \min\{r \mid r \in [i,j] \wedge v[r] \leq v[k]\ \forall k \in [i,j]\}$

Class	*Space*	*Time*
`rmq_support_sparse_table`	$n \log^2 n$	$\mathcal{O}(1)$
`rmq_succint_sada`	$4n + o(n)$	$\mathcal{O}(1)$
`rmq_succint_sct`	$2n + o(n)$	$\mathcal{O}(1)$

Constructing data structures

Let `o` be a WT-, CSA-, or CST-object. Object `o` is built with `construct(o,file,num_bytes=0)` from a sequence stored in `file`. File is interpreted dependent on the value of `num_bytes`:

Value	*File interpreted as*
`num_bytes=0`	serialized `int_vector<>`.
`num_bytes=1`	byte sequence of length `util::file_size(file)`.
`num_bytes=2`	16-bit word sequence.
`num_bytes=4`	32-bit word sequence.
`num_bytes=8`	64-bit word sequence.
`num_bytes=d`	Parse decimal numbers.

Note: `construct` writes/reads data to/from disk during construction. Accessing disk for small instances is a considerable overhead. `construct_im(o,data,num_bytes=0)` will build `o` using only main memory. Have a look at this handy tool for an example.

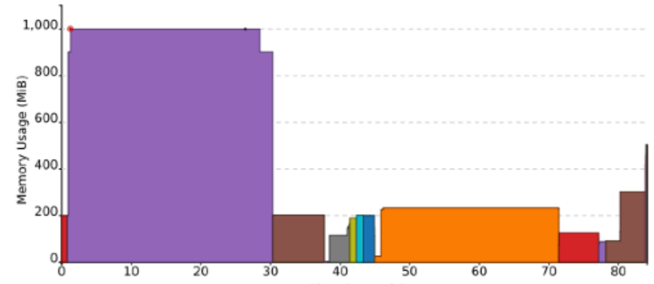
Configuring construction

The locations and names of the intermediate files can be configured by a `cache_config` object. It is constructed by `cache_config(del,tmp_dir,id, map)` where `del` is a boolean variable which specifies if the intermediate files should be deleted after construction, `tmp_dir` is a path to the directory where the intermediate files should be stored, `id` is used as part of the file names, and `map` contains a mapping of keys (e.g. `conf::KEY_BWT`, `conf::KEY_SA`,...) to file paths.
The `cache_config` parameter extends the construction method to: `construct(o,file,config,num_bytes)`.
The following methods (`key` is a key string, `config` represenet a `cache_config` object, and `o` a sdsl object) should be handy in customized construction processes:

```
cache_file_name(key,config)
cache_file_exists(key,config)
register_cache_file(key,config)
load_from_cache(o,key,config)
store_to_cache(o,key,config)
```

Resource requirements

Memory: The memory peak of CSA and CST construction occurs during the SA construction, which is 5 times the texts size for byte-alphabets and inputs < 2 GiB (see the Figure below for a 200 MB text) and 9 times for larger inputs. For integer alphabets the construction takes about twice the space of the resulting output.
Time: A CST construction processes at about 2 MB/s. The Figure below shows the resource consumption during the construction of a `cst_sct3<>` CST for 200 MB English text. For a detailed description of the phases click on the figure.



This diagram was generated using the sample program memory-visualization.cpp.

Reading and writing data

Importing data into sdsl structures

`load_vector_from_file(v, file, num_bytes)`
Load `file` into an `int_vector v`. Interpretation of `file` depends on `num_bytes`; see method `construct`.

Store sdsl structures

Use `store_to_file(o, file)` to store an sdsl object `o` to `file`. Object `o` can also be serialized into a `std::ostream`-object `out` by the call `o.serialize(out)`.

Load sdsl structures

Use `load_from_file(o, file)` to load an sdsl object `o`, which is stored in `file`. Call `o.load(in)` reads `o` from `std::istream`-object `in`.

Utility methods

More useful methods in the `sdsl::util` namespace:

Method	*Description*
`pid()`	Id of current process.
`id()`	Get unique id inside the process.
`basename(p)`	Get filename part of a path `p`.
`dirname(p)`	Get directory part of a path `p`.
`demangle(o)`	Demangles output of `typeid(o).name()`.
`demangle2(o)`	Simplifies output of `demangle`. E.g. removes `sdsl::`-prefixes, ...
`to_string(o)`	Transform object `o` to a string.
`assign(o1,o2)`	Assign `o1` to `o2`, or swap `o1` and `o2` if the objects are of the same type.
`clear(o)`	Set `o` to the empty object.

Measuring and Visualizing Space

`size_in_bytes(o)` returns the space used by an sdsl object `o`. Call `write_structure<JSON_FORMAT>(o,out)` to get a detailed space breakdown written in JSON format to stream `out`. `<HTML_FORMAT>` will write a HTML page (like this), which includes an interactive SVG-figure.

Methods on words

Class `bits` contains various fast methods on a 64-bit word x. Here the most important ones.

Method	*Description*
`bits::cnt(`x`)`	Number of set bits in x.
`bits::sel(`x,i`)`	Position of i-th set bit, $i \in [0, \texttt{cnt}(x)-1)$.
`bits::lo(`x`)`	Position of least significant set bit.
`bits::hi(`x`)`	Position of most significant set bit.

Note: Positions in x start at 0. `lo` and `hi` return 0 for $x = 0$.

Tests

A `make test` call in the test directory, downloads test inputs, compiles tests, and executes them.

Benchmarks

Directory benchmark contains configurable benchmarks for various data structure, like WTs, CSAs/FM-indexes (measuring time and space for operations count, locate, and extract).

Debugging

You get the gdb command `pv <int_vector> <idx1> <idx2>`, which displays the elements of an `int_vector` in the range `[idx1, idx2]` by appending the file sdsl.gdb to your `.gdbinit`.

Cheatsheet template provided by Winston Chang
http://www.stdout.org/~winston/latex/

Notes

1 `select_0_type` not defined for `sd_vector`.
2 It is also possible to rank `0` or the patterns `10` and `01`.
3 It is also possible to select `0` or the patterns `10` and `01`.
4 For PBS the bits are counted in the prefix $[0..i]$.
5 Or maximum value; can be set by a template parameter.