# Succinct Data Structures

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

## Contents

# 1 Why use compressed data directly?

A compressed file saves storage, but a query may require decompressing it first. A **succinct data structure** aims for both: space close to the information-theoretic minimum and direct support for useful operations.

This matters when topology is large. A pointer-based tree may spend more space on addresses than on its values, and scattered nodes make poor use of caches. A compact bit sequence can hold the same topology in a few bits per node while still supporting parent, child, depth, and subtree queries.

Succinctness is always relative to a stated object family and interface. Encoding only a trie's shape does not also encode edge characters, terminal markers, or values.

# 2 Learning goals

After studying this note, you should be able to:

- derive a space lower bound by counting possible objects;
- define rank and select with an unambiguous indexing convention;
- explain how a small auxiliary index supports constant-time bit-vector queries;
- encode and decode a tree with balanced parentheses and LOUDS;
- implement simple `findclose` and subtree-size operations;
- compare level-order, depth-first, and parenthesis encodings; and
- separate topology space from labels, payload, and query-index redundancy.

# 3 Prerequisites

- Bits, logarithms, and the rule that $b$ bits distinguish at most $2^b$ cases.
- Rooted ordered trees, binary trees, tries, breadth-first search, and depth-first search.
- Arrays and the word-RAM idea that one machine word contains $\Theta(\log n)$ bits.

# 4 Counting gives the target

For a first pass, follow one dependency chain: counting lower bounds, the word-RAM model, rank/select, and then balanced parentheses and LOUDS. DFUDS and the heap-like binary encoding are useful comparisons after that core chain is clear.

Suppose a family contains $N$ possible objects. Any lossless representation needs at least

$$L = \lceil \log_2 N \rceil$$

bits in the worst case, because fewer bits provide fewer than $N$ distinct codewords.

Terminology varies slightly across the literature, but a useful hierarchy is:

- **implicit:** essentially $L$ bits, with only constant extra space;
- **succinct:** $L + o(L)$ bits;
- **compact:** $O(L)$ bits.

The lower bound must count the right objects. There are

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

binary-tree shapes with $n$ internal nodes. These are Catalan numbers, and

$$\log_2 C_n = 2n - \Theta(\log n).$$

Thus roughly two bits per node are necessary just to distinguish arbitrary binary-tree shapes. A representation using $2n + o(n)$ bits is genuinely near the lower bound; two 64-bit child pointers per node are not.

# 5 Computational model and scope

The standard static results use a **word-RAM** with word size $w = \Theta(\log n)$. Arithmetic, bitwise Boolean operations, and shifts on one word take constant time. Small lookup tables shared by the whole structure are permitted.

Static and dynamic structures have different trade-offs. The $n + o(n)$-bit, constant-time rank/select results below concern a fixed bit vector. Supporting insertions and deletions into the middle of that vector requires more machinery and usually weaker bounds.

# 6 Bit vectors: rank and select

Let $B[0..n-1]$ be a bit vector. In this note, `rank` uses a zero-based **inclusive** position:

$$rank_1(i) = \sum_{j=0}^{i} B[j].$$

For an occurrence number $k \geq 1$,

$$select_1(k) = \min\{i : rank_1(i) = k\}.$$

Thus rank maps a position to an occurrence count, while select maps an occurrence count back to a position. Define $rank_0$ and $select_0$ analogously.

Many libraries instead define $rank_1(i)$ on the half-open prefix $B[0..i)$. Under that convention, this note's inclusive value at $i$ is the library's value at $i+1$. Check the boundary and occurrence-number conventions before comparing formulas or test results.

For

```
index:  0 1 2 3 4 5 6 7 8 9 10 11
B:      0 1 1 0 1 0 0 1 0 1  1  1
```

we have $rank_1(7) = 4$, $select_1(5) = 9$, and $rank_0(7) = 4$. Checking a small vector by hand is the best way to catch an off-by-one convention.

## 6.1 Why constant-time rank is possible

Storing every prefix count would cost $n \log n$ bits. Instead, use two sampling levels:

1. Divide the vector into superblocks of about $(\log n)^2$ bits and store the absolute rank at each boundary.
2. Divide each superblock into subblocks of about $\frac{1}{2} \log n$ bits and store each subblock's rank relative to its superblock.
3. Answer the remaining short fragment by a shared lookup table or a machine population-count operation.

Superblock counters use about

$$\frac{n}{(\log n)^2} \log n = O\Big(\frac{n}{\log n}\Big)$$

bits. Relative subblock counters and the shared table also use $o(n)$ bits with suitable constants. The original vector plus index therefore occupies $n + o(n)$ bits and answers rank in $O(1)$ time. Select uses a related sampling scheme; the construction is more involved but reaches the same asymptotic bounds.

The index is part of the representation's space. Calling a raw bit string "succinct" while ignoring a large query index is incomplete accounting.

# 7 Tree topology and payload

Rank and select are the navigation primitives, not the final application. A tree encoding turns structural questions into a small number of rank, select, and excess queries on its topology bit vector.

An ordered rooted tree distinguishes the order of a node's children. Tree encodings below describe that topology. Applications usually need more:

- labels on nodes or edges;
- terminal bits in a trie;
- associated values;
- offsets into external data; and
- auxiliary indexes supporting the chosen queries.

These components should be reported separately. A trie can share many prefixes, so its number of nodes is at most, not necessarily equal to, the total number of characters in its keys.

# 8 Balanced-parentheses representation

Perform a depth-first traversal of an ordered rooted tree. Write `(` when entering a node and `)` when leaving it. Equivalently, write 1 for an opening parenthesis and 0 for a closing parenthesis. An $n$-node tree produces exactly $2n$ bits.

The running **excess**

$$excess(i) = rank_1(i) - rank_0(i)$$

is the number of currently open nodes after position $i$. For an opening parenthesis at position $i$, the node depth is $excess(i) - 1$ when the root has depth zero.

If `findclose(i)` returns the matching closing parenthesis, the complete encoding of that node's subtree is the contiguous interval from $i$ through `findclose(i)`. Therefore

$$subtree\_size(i) = \frac{findclose(i) - i + 1}{2}.$$

The parent is represented by the nearest opening parenthesis that encloses the node's pair. A first child begins immediately after the node's opening parenthesis if that next symbol is open; a next sibling begins immediately after its matching close if that next symbol is also open.

A simple linear scan is enough to validate the representation:

```
findclose(B, i):
    require B[i] == '('
    balance = 1
    j = i + 1
    while balance > 0:
        if B[j] == '(': balance += 1
        else:           balance -= 1
        j += 1
    return j - 1
```

This scan can take $O(n)$. A succinct excess-search index adds $o(n)$ bits and supports matching, enclosing, depth, parent, child, and many other navigation operations in constant time.

# 9 LOUDS: level-order unary degrees

**LOUDS** stands for level-order unary degree sequence. Visit nodes in breadth-first order. For a node of degree $d$, write $d$ one-bits followed by a zero. The degrees sum to $n-1$, so a nonempty $n$-node tree uses $(n-1)+n = 2n-1$ bits.

Suppose a root has children $a$ and $b$, and $a$ has one child $c$. Breadth-first degrees are $2, 1, 0, 0$, hence

```
110 10 0 0  -> 1101000
```

Zeros delimit nodes; one-bits introduce their children in breadth-first order.

For the following formulas only, use one-based bit positions and one-based node numbers. Let

$$z_i = select_0(i), \qquad z_0 = 0.$$

Here $rank_b(p)$ counts $b$-bits in positions 1 through $p$, with $rank_b(0) = 0$ for $b \in \{0, 1\}$.

Therefore $degree(i) = z_i - z_{i-1} - 1$.

The one-bits between positions $z_{i-1} + 1$ and $z_i - 1$ represent the children of node $i$. A one-bit of rank $r$ introduces node $r + 1$. For a non-root node $j$, let $p = select_1(j-1)$; then

$$parent(j) = rank_0(p) + 1.$$

If $degree(i) > 0$, its children have consecutive node numbers beginning at

$$first\_child(i) = rank_1(z_{i-1}) + 2.$$

LOUDS supports parent, degree, child, and sibling navigation naturally in breadth-first numbering. A subtree is not generally one contiguous LOUDS interval, so subtree-size queries are less direct.

# 10 Depth-first unary degrees

**DFUDS** writes the same unary degree code, $1^d0$, but visits nodes in depth-first preorder, with a conventional extra opening bit to align its navigation formulas. Subtrees are contiguous because depth-first traversal finishes one subtree before entering the next.

With rank/select and balanced-parentheses primitives, DFUDS can support parent, degree, child, sibling, and subtree-size operations in constant time using $2n + o(n)$ bits. It combines unary-degree information with depth-first locality.

# 11 Enrichment: heap-like binary-tree encoding

For a binary tree, label internal nodes 1 and external null nodes 0. Emit the root label, then, for each internal node in breadth-first order, emit the labels of its left and right children; null nodes emit no children. There are $n$ internal and $n + 1$ external nodes, so the sequence uses $2n + 1$ bits.

Let $T[1..2n+1]$ be this sequence. If an internal node occurs at bit position $x$, its breadth-first internal-node number is $i = rank_1(x)$, and its two child labels occur at bit positions $2i$ and $2i + 1$. Conversely, a non-root internal node at bit position $x$ has parent number $\lfloor x/2 \rfloor$, whose bit position is

$$select_1(\lfloor x/2 \rfloor).$$

The multiplication applies to the compact internal-node number $i$, not directly to the node's bit position $x$. Rank and select provide exactly that translation.

## 12 Comparing tree encodings

| Encoding | Order | Topology bits | Natural strengths | Main caution |
|---|---|---|---|---|
| Balanced parentheses | depth first | $2n$ | matching, depth, ancestors, subtree interval | needs excess-search index |
| LOUDS | breadth first | $2n-1$ | degree, parent, child range, level order | subtree not contiguous |
| DFUDS | depth first | about $2n$ | degree navigation plus subtree locality | formulas are convention-sensitive |
| Heap-like binary | level order | $2n+1$ | binary child/parent mapping | includes explicit external nodes |

No encoding is universally best. Choose the operation interface first, then the representation and auxiliary index that support it.

## 13 Worked trace: a small trie

Consider keys `a`, `ac`, and `b`. The topology has four nodes: a root with children `a` and `b`, and node `a` has child `c`. The labels `a,b,c` and terminal markers for all three keys are payload; the topology alone does not contain them.

Depth-first balanced parentheses are

```
((())())
01234567
```

Positions 0, 1, 2, and 5 open the root, `a`, `c`, and `b`. For the prefix node `a` at position 1, `findclose(1)=4`. Its subtree size is

$$\frac{4-1+1}{2} = 2,$$

corresponding to nodes `a` and `c`. Its next symbol is another opening parenthesis, so it has a child. A fast implementation replaces the scan with a matching/excess index but returns the same structural answer.

LOUDS for the same topology uses degrees $2, 1, 0, 0$ and is `1101000`. The two encodings describe the same tree in different traversal orders.

## 14 Complexity and trade-offs

| Task | Simple representation | Succinct target |
|---|---|---|
| Store ordered-tree topology | pointer words per node | $2n + o(n)$ bits |
| Bit-vector rank/select | scan: $O(n)$ time | $O(1)$ time, $n + o(n)$ bits total |
| Parenthesis matching | scan: $O(n)$ time | $O(1)$ with $o(n)$-bit index |
| Construction | often straightforward | may need $O(n)$ temporary workspace |
| Dynamic updates | local pointer change | substantially more difficult |

Compact storage can improve cache behaviour and reduce I/O, but extra bit operations and construction complexity can dominate on small instances. Measure the complete structure, including alignment, allocator overhead, labels, and temporary workspace.

## 15 Connections

- Tree terminology and traversals supply the objects and visit orders encoded here.
- Rank/select will reappear in the Burrows-Wheeler transform and FM-index.
- Trie topology connects this note to exact matching and full-text indexing.
- Information-theoretic counting is the same style of lower-bound reasoning used for comparison sorting.
- LOUDS is a bridge between breadth-first graph traversal and compact tree navigation.

## 16 Common mistakes

- Calling any compressed representation succinct without comparing it with a lower bound.
- Mixing inclusive and exclusive rank or zero-based and one-based select conventions.
- Counting two topology bits per node while omitting labels, terminal bits, and indexes.
- Claiming constant-time rank/select without stating the static word-RAM model.
- Assuming a LOUDS subtree is contiguous because a parenthesis-encoded subtree is.
- Treating a byte-per-parenthesis prototype as a two-bit-per-node implementation.

## 17 Self-check

1. Compute $rank_1(10)$ and $select_0(4)$ for the example bit vector.
2. Why is a two-level rank directory $o(n)$ rather than $O(n \log n)$ bits?
3. Encode a root with three leaf children using balanced parentheses and LOUDS.
4. Derive the LOUDS degree formula from the positions of consecutive zeros.
5. How does `findclose` reveal subtree size?
6. Which additional data are required to turn a succinct trie topology into a dictionary?
7. Why can a smaller representation also run faster?

## 18 Revision summary

- Counting possible objects gives the correct space target.
- Succinct means lower-bound space plus lower-order redundancy, together with a stated query interface.
- Rank and select translate between positions and occurrence numbers.
- Balanced parentheses make subtrees contiguous; LOUDS makes breadth-first child ranges direct.
- DFUDS combines unary degrees with depth-first locality.
- Topology, labels, payload, and query indexes must all be counted.
- Static succinctness does not automatically imply easy dynamic updates.

## 19 Implementations and hands-on exploration

- Python's maintained `bitarray` package provides compact bit storage and fast C-level bit operations. It is a useful prototype substrate, but it does not by itself add the $o(n)$ rank/select directory required for constant-time succinct queries.
- SDSL v3 is the current header-only C++ Succinct Data Structure Library, with bit vectors, rank/select supports, balanced-parentheses structures, wavelet trees, and compressed text indexes. It targets modern C++, and its concrete API and conventions should be checked rather than inferred from older SDSL v2 examples.
- Rust's `sux` provides composable bit vectors, several rank/select trade-offs, Elias-Fano dictionaries, and basic balanced-parentheses support. Performance-oriented unchecked operations and nested support structures make boundary conventions and total memory accounting especially important.
- Rust's Vers implements succinct bit vectors, rank/select, and balanced-parentheses trees. Its fastest paths depend on CPU features such as `popcnt` and BMI2, so portable and native builds can have materially different timings.

- Java's Sux4J implements rank/select structures, Elias-Fano lists, and succinct static functions. Most structures are static, and the memory of indexes and Java objects must be included when testing succinctness.

### 19.1 Small experiments

1. Use Python `bitarray` to implement naive inclusive `rank1` and one-based `select1`. Exhaustively test all bit vectors up to length 12, then adapt the calls to one library above and verify the half-open/inclusive translation at every boundary.
2. Build rank/select supports for bit vectors of several lengths and one-bit densities. Measure construction time, serialized or reported total bytes, and random-query time separately; compare a fast higher-redundancy index with a smaller index instead of reporting query time alone.

## 20 Sources and further study

- Succinct data structures lecture deck, Jaak Vilo.
- SDSL v3 repository and documentation, SDSL contributors.
- Gonzalo Navarro, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016.