CSE3144 · Advanced Data Structures · Jul–Nov Semester 2026 · Lecture 14 of 36 · CO CSE3144.2

Suffix Trees & String Processing

Lecture 13 asked "is this word in my dictionary?" Today's question is different: given one text, answer almost anything about every substring it contains — in time that barely depends on the text's length at all.

Dr. Manu Shrivastava
Course Instructor · Consultation Fri 2–5 PM, LHC 308F

~70 minutes
Session outcome: analyze pattern matching and text indexing methods

Today, part by part

One text, every substring question, answered fast

A DIFFERENT KIND OF QUESTION

Lecture 13's tries answered "is this exact word stored?" against a *dictionary of many words*. Today's question is different: given **one long text** T, answer things like "does pattern P occur anywhere in T?", "what's the longest substring that repeats?", "what's the longest stretch shared by two texts?" — over and over, cheaply, without rescanning T each time.

Key trick: every substring of T is a *prefix of some suffix* of T. So if you store **every suffix** of T in a trie, answering "does P occur in T?" becomes exactly a prefix search — an operation you already know cold from Lecture 13.

THE PLAN

Take T = "banana", append a unique terminal character $ (smaller than every real character, and appearing nowhere else) to get "banana$" — this guarantees no suffix is ever a prefix of another, so every suffix gets its own distinct leaf. List its 7 suffixes:

`banana$, anana$, nana$, ana$, na$, a$, $`

Insert all 7 into a trie exactly as in Lecture 13. This structure is called a **suffix trie**. We'll build it in a few minutes — but first, a serious problem with it.

A naive suffix trie can cost Θ(m²) space — not Θ(m)

TWO STRINGS, TWO VERY DIFFERENT GROWTH RATES

String "a" repeated m times (aaaa…a): every suffix is a shorter run of a's — the trie is a single chain. Node count: `2m + 2` — **linear** in m.

String $a^n b^n$ (say n a's then n b's, so m = 2n): now there are n distinct "a-chains" (one per suffix starting inside the a-run), each needing its own trailing "b-chain" of length up to n. Node count: `n² + 4n + 2` — **quadratic** in n, i.e. quadratic in m.

WHY THIS HAPPENS, AND THE GENERAL BOUND

In the worst case: the trie's depth (top to bottom) is at most m + 1 — the length of the longest suffix. Its *width* (how many distinct substrings exist at any one length) can itself be as large as m. Depth × width gives an upper bound of **$O(m^2)$** nodes — and $a^n b^n$ shows this bound is actually achieved, not just a loose estimate.

For a genome-length text (m in the millions), an $O(m^2)$-space structure is unusable. We need the *same* lookup power in **guaranteed O(m)** space — that requirement is exactly what a **suffix tree** is built to satisfy.

Same length m = 4, two very different trees: 10 nodes vs. 14

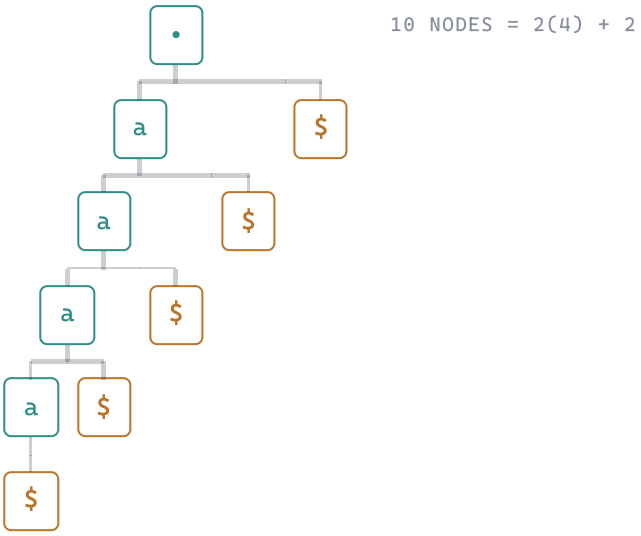
WHY THE LENGTH ALONE ISN'T THE WHOLE STORY

Both formulas on the last slide use **m** — but m by itself doesn't fix the tree's size. What actually decides it is how much the string's own suffixes overlap with each other. Fix m = 4 and compare two genuinely different strings of that exact length.

THE TWO CANDIDATES

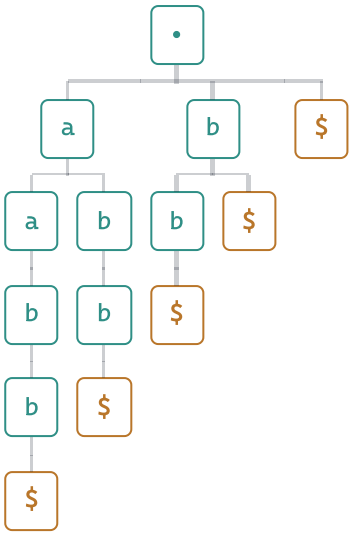
"aaaa$"** — repeated-character family, m = 4 — against **"aabb$" — the $a^n b^n$ family with n = 2, so m = 2n = 4. Same length. Watch what happens to the node count.

"AAAA$" — 5 SUFFIXES, MAXIMAL SHARING



aaaa$, aaa$, aa$, a$, $ — every suffix is a prefix of a longer one, so all 5 sit on **one spine**, each just peeling off its own "$" leaf. **2(4) + 2 = 10** nodes.

"AABB$" — 5 SUFFIXES, LIMITED SHARING



14 NODES = 2² + 4(2) + 2

`aabb$`, `abb$`, `bb$`, `b$`, `$` — the two suffixes starting inside the a-run share only their first "a", then diverge, and each needs its **own** unshared b-tail. **$2^2 + 4(2) + 2 = 14$** nodes.

THE TAKEAWAY

Same m, 40% more nodes — from a string that isn't even trying to be adversarial. Length alone never determines a suffix trie's size; internal repetition does. A highly repetitive string collapses onto one spine and stays linear; a string with only limited self-overlap, even one this short, already shows the quadratic behavior setting in. At genome scale you can't assume the text will be "nice" like `aaaa$` — which is exactly why the suffix **tree** is built to guarantee O(m) no matter which string you hand it.

Compress every non-branching chain — store index ranges, not text

THE COMPRESSION RULE

Take the suffix trie and find every maximal chain of nodes with exactly one child (nodes that carry no real branching decision). Collapse each such chain into a **single edge**. That edge is labeled with a **(start, end) index pair** into the original text T — never a copy of the substring itself.

WHAT THIS GUARANTEES

- Exactly **m leaves** for a text of length m (one per suffix, guaranteed distinct by the terminal $).
- **Fewer than m** internal branching nodes (excluding the root) — every internal node has ≥ 2 children.
- Total space: **O(m)** — always, not just on average. This is the structure called a **suffix tree**.

A Swiss-army knife for string problems — with a genuine catch

ADVANTAGES

- **Guaranteed O(m) space** — no worst-case blow-up, unlike the raw suffix trie.
- **O(p) exact pattern search** for any pattern of length p, after one O(m) preprocessing pass over T.
- One structure answers *many* different string questions — pattern matching, longest repeated substring, longest common substring of two texts, and more — all in time proportional to the answer, not to rescanning T.

LIMITATIONS

- **Construction is intricate.** The naive "insert every suffix, then compress" approach we're about to animate costs roughly $O(m^2)$ work. A genuinely linear O(m) build is possible, but the algorithm that achieves it is notoriously fiddly to implement without bugs — well beyond what this course derives by hand.
- **Real-world overhead.** Each node/edge carries pointers and index pairs — the constant factor per character is larger than a simpler alternative, the **suffix array** (a sorted array of suffix start-positions plus an LCP array), which is more cache-friendly and is often preferred in production systems despite similar asymptotic guarantees.

Insert all 7 suffixes of "banana$", one at a time

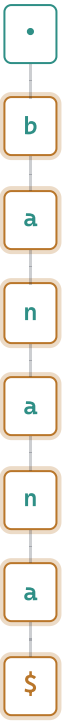
Exactly the Lecture-13 trie-insertion rule, applied to the 7 suffixes of "banana$" in left-to-right order. Watch how much of each new suffix is **brand new** versus **already there** — that ratio is the whole story of why compression will pay off.

INTERACTIVE — INSERT BANANA$, ANANA$, NANA$, ANA$, NA$, A$, $

STEP 1 OF 9

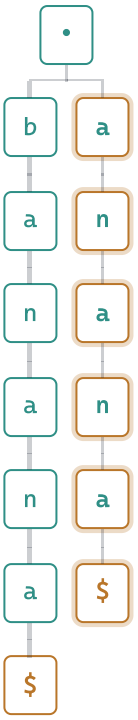
Start: empty trie. Insert the 7 suffixes of "banana$" left to right: banana$, anana$, nana$, ana$, na$, a$, $.

STEP 2 OF 9



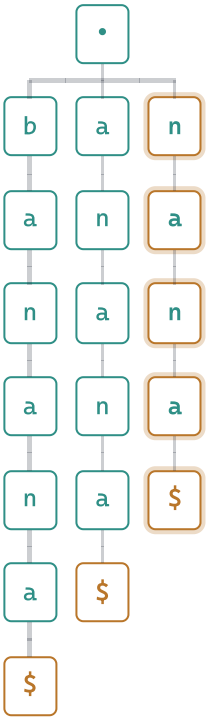
Insert "banana$": nothing exists yet — all 7 characters (b,a,n,a,n,a,$) create brand-new nodes.

STEP 3 OF 9



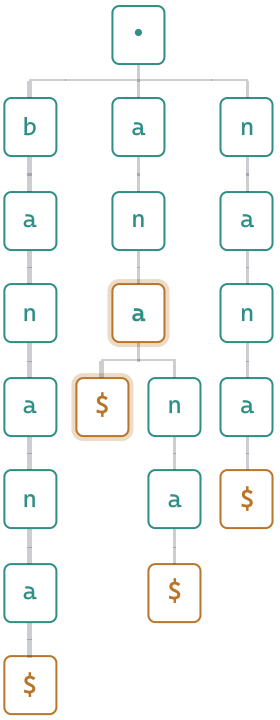
Insert "anana$": shares nothing with "banana$" (different first letter) — 6 brand-new nodes (a,n,a,n,a,$), a second branch off the root.

STEP 4 OF 9



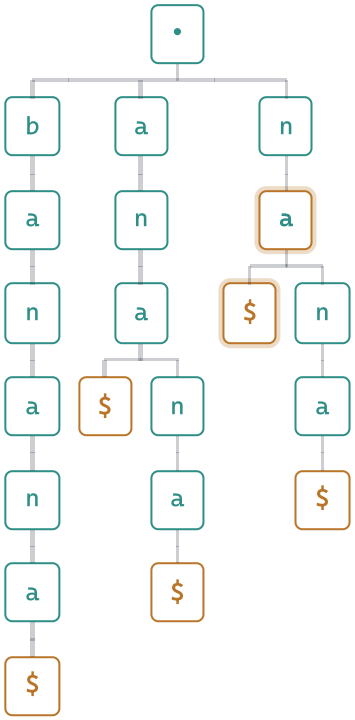
Insert "nana$": again nothing shared — 5 brand-new nodes (n,a,n,a,$), a third branch off the root.

STEP 5 OF 9



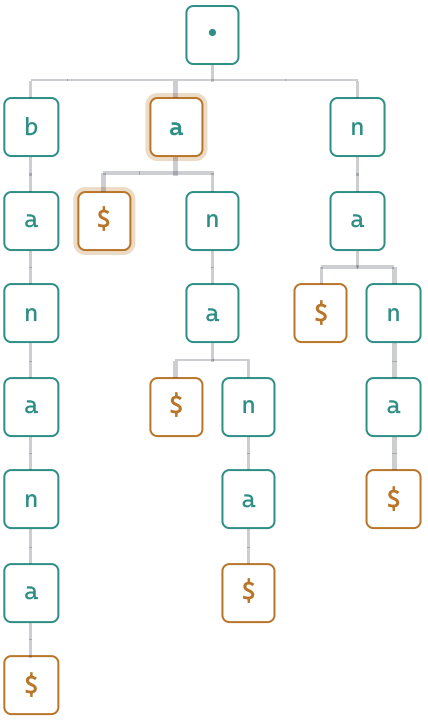
Insert "ana$": "a", "n", "a" already exist (reused) — but the existing path continues with 'n' (toward "anana$"), and this suffix needs '$' instead. **Branch:** only 1 new node, a "$" leaf.

STEP 6 OF 9



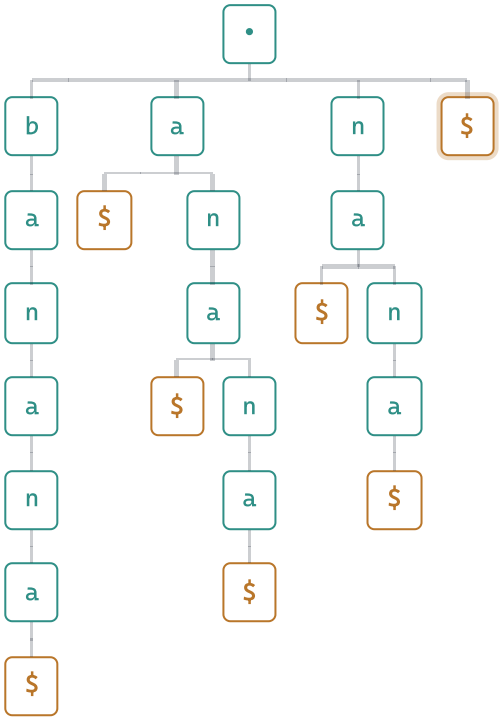
Insert "na$": "n", "a" already exist (reused) — the existing path continues with 'n' (toward "nana$"), this suffix needs '$'. **Branch:** only 1 new node.

STEP 7 OF 9



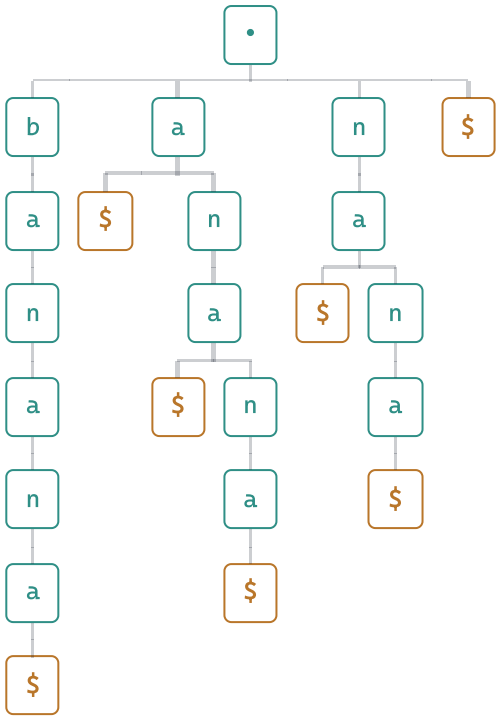
Insert "a$": root's "a" child already exists (reused) — it only had a 'n' child so far, and this suffix needs '$' directly. **Branch:** only 1 new node.

STEP 8 OF 9



Insert "$": the shortest possible suffix — root gains a direct "$" child. 1 new node. All 7 suffixes are now placed.

STEP 9 OF 9



23 TOTAL NODES — 7 LEAVES, 15 INTERNAL (EXCLUDING ROOT), ROOT

Done. New-node tally across the 7 insertions: 7, 6, 5, 1, 1, 1, 1 = 22, plus the root = **23 nodes total**. Notice how sharply reuse increased once the tree filled in — that reuse is exactly what compression is about to capture permanently.

Not a new data structure — a trie node plus two integer arrays

You've just built the trie one character per edge. Before compressing it, look at what a node stores — because compression only works if a **single edge** can name a whole substring, and that is purely a question of what fits in a node.

Lecture 13's trie node — one array, indexed by character:

```
struct TrieNode {
    TrieNode* child[26];
    bool      isWord;
}
```

Slot c holds the subtree under letter c. The edge *is* that one letter.

Suffix tree node — same array, two integers added *per slot*:

```
struct STNode {
    STNode* child[26];
    int     start[26];
    int     end[26];
}
```

Same slot c, same pointer — plus "this edge spells T[start[c] .. end[c]]".

TWO NODES OF THE TRIE YOU JUST BUILT, LAID OUT IN MEMORY

This is still the **uncompressed** 23-node trie — every edge is one character, so every slot has `start[c] = end[c]`. The indices point into the **original text**, which is never copied — it just sits in memory once:

T	b	a	n	a	n	a	$
INDEX	1	2	3	4	5	6	7

ROOT (the parent — still the 23-node trie)

SLOT C	$	a	b	n	*c, d, e … z*
CHILD[C]	LEAF 7	**NODE A ↓**	node "b"	node "n"	*NULL*
START[C]	7	2	1	3	–
END[C]	7	2	1	3	–
READS AS	"$"	"a"	"b"	"n"	–

↓ child['a'] points here

NODE A (the node under slot "a" — same struct, nothing new)

SLOT C	$	n	*every other letter*
CHILD[C]	LEAF 6	node "an"	*NULL*
START[C]	7	3	–
END[C]	7	3	–
READS AS	"$"	"n"	–

So: not a different data structure. It is the same array-indexed-by-character node you built in Lecture 13 — the parent holds pointers, each pointer sits in the slot named by its edge's *first* character, and the child is just another node of the identical type. The only addition is `start[c]` and `end[c]`: two integers that say "this edge spells out T[start..end]" instead of the trie's implicit "this edge spells out exactly the one character c". Right now every slot is at `start[c] = end[c]` — one character, exactly the trie. But **nothing in the struct requires that**: two integers hold a 7-character edge just as cheaply as a 1-character one. Widening those `end` values is all compression does on the next slide, and it is why the node count drops from $O(m^2)$ to $O(m)$.

23 nodes become 11 — same guarantees, a fraction of the space

WHAT CHANGES

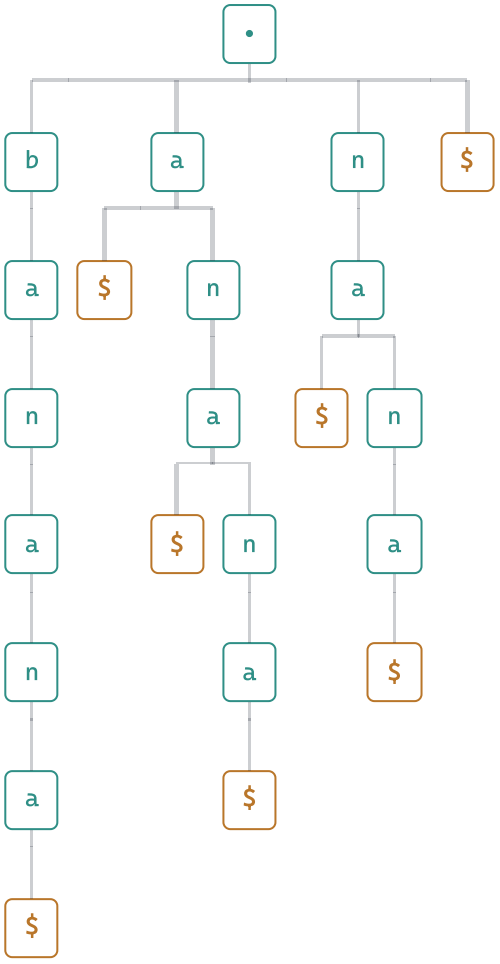
Every node with exactly one child and no branching decision gets merged into the edge above it. Leaves keep their identity (they're labeled here with the **starting position** of their suffix — standard suffix-tree convention); only the pure "pass-through" nodes disappear.

HOW TO READ THE COMPRESSED TREE

Every edge now carries a full substring label (shown as a small tag on the connecting line) instead of one character. Internal node boxes show the string spelled out from the root to that point — this is exactly the "string-depth" you'll need for the next two animations.

INTERACTIVE — BEFORE AND AFTER

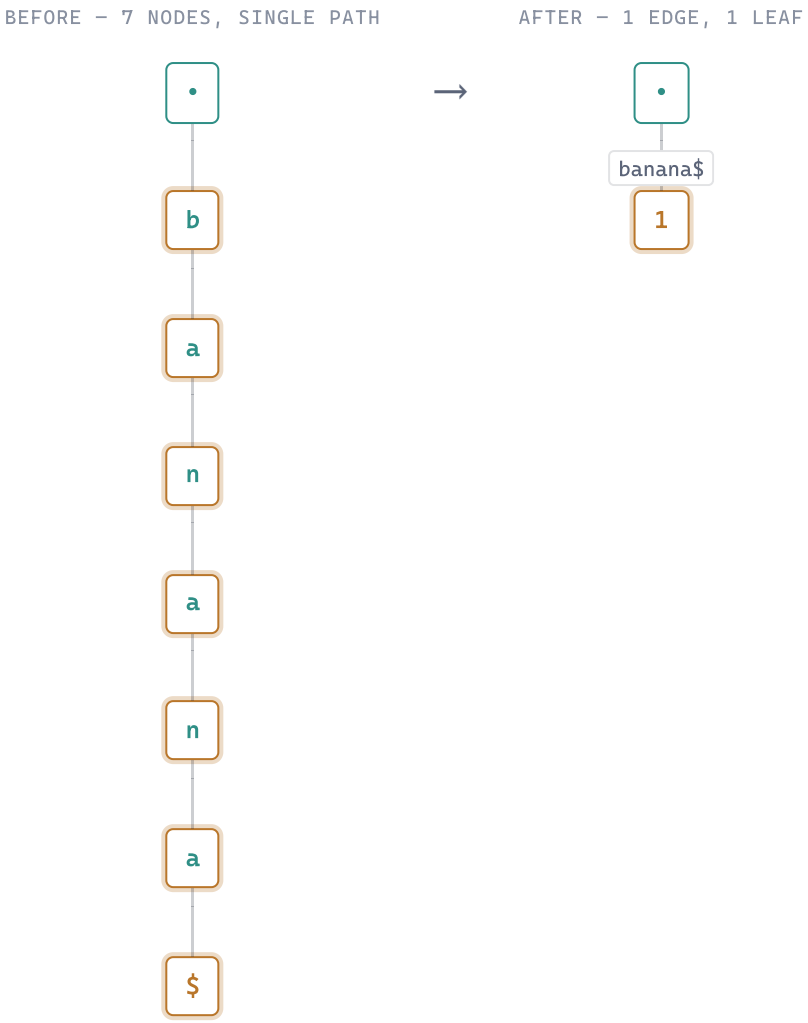
STEP 1 OF 5



UNCOMPRESSED SUFFIX TRIE — 23 NODES

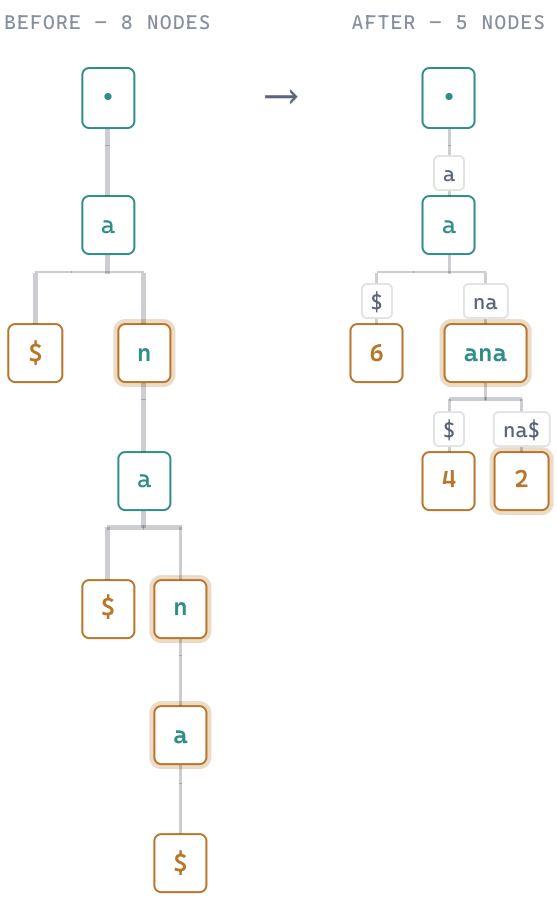
Before: the full suffix trie we just built. We'll collapse it one recognizable branch at a time, so you can see exactly which nodes disappear and why — not just the final answer.

STEP 2 OF 5



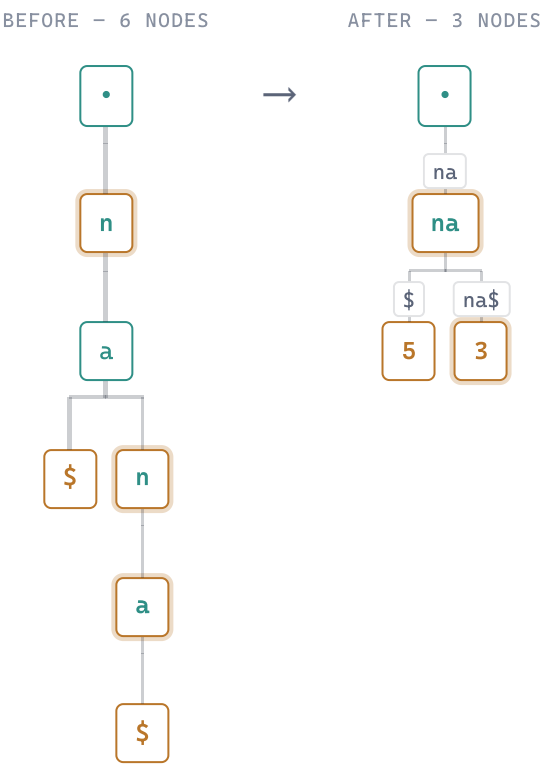
The "b" branch: root→b→a→n→a→n→a→$. Every single node here has exactly **one** child — nowhere along this path is there ever a decision to make. So the whole run collapses into one edge, labeled with the whole string it spelled out: "banana$", straight to the leaf for suffix 1. 7 nodes become 1 edge.

STEP 3 OF 5



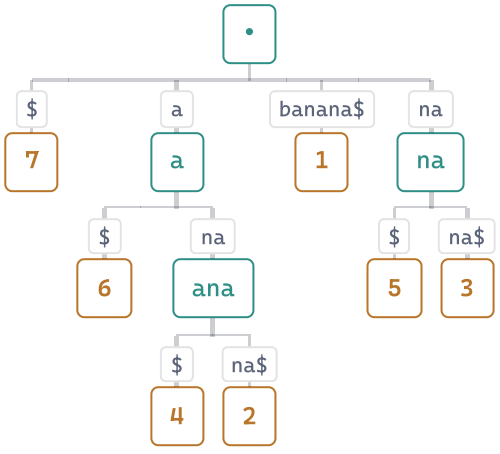
The "a" branch: root→a branches immediately (into "$" and "n"), so that node survives as-is. But past it, "n" alone has only one child ("a") — collapse it into edge **"na"**. Then that node branches again (into "$" and another "n") and survives — but the "n,a" run after *that* has no branching either, so it collapses into edge **"na$"**. Two separate merges, same branch: 8 nodes become 5.

STEP 4 OF 5



The "n" branch: the exact same pattern, mirrored. root→n (one node, one child) collapses into edge "na"; the following "n,a" run collapses into edge "na$". 6 nodes become 3 — no new logic here, just the same rule applied again.

STEP 5 OF 5



11 TOTAL NODES — 1 ROOT, 3 INTERNAL, 7 LEAVES, 10 EDGES

All three branches, put back together. 1 (root) + 1 (the collapsed "b" branch) + 5 (the "a" branch) + 3 (the "n" branch) + 1 (leaf 7, "$" alone, untouched — it was never a chain) = **11 nodes**, down from 23. Leaves are labeled with their suffix's starting position (1–7) — the standard convention. Same 7 suffixes, same guarantees, less than half the nodes.

The same root array, before and after the merge

You saw the picture collapse. Here is what actually changed **in memory** — the root's own array, the one from L14·05a, in both states. The text T is untouched throughout; only the node's three arrays are written.

BEFORE — THE UNCOMPRESSED TRIE: EVERY EDGE IS ONE CHARACTER

ROOT (23-node trie)

SLOT C	$	a	b	n	c,d,e … z
CHILD[C]	LEAF 7	node "a"	node "b"	node "n"	NULL
START[C]	7	2	1	3	–
END[C]	7	2	1	3	–
READS AS	"$"	"a"	"b"	"n"	–

AFTER — THE 11-NODE SUFFIX TREE: AMBER CELLS ARE THE ONLY WRITES

ROOT (11-node suffix tree)

SLOT C	$	a	b	n	c,d,e … z
CHILD[C]	LEAF 7	NODE A	LEAF 1	NODE C	NULL
START[C]	7	2	1	3	–
END[C]	7	2	7	4	–
READS AS	"$"	"a"	"banana$"	"na"	–

THREE THINGS TO NOTICE

- **start[c] never moved.** Not one slot. An edge always begins where it always began — merging only makes it run further.
- **Only child[c] and end[c] were written.** Take the child's **end**, repoint past the child, free it. Two writes per merge, no matter how long the chain was.
- **"reads as" is not stored.** It's computed as T[start..end] whenever the characters are actually needed. There is no fourth array.

WHY TWO SLOTS DIDN'T CHANGE AT ALL

Slot $ already pointed straight at a leaf — no chain beneath it to collapse. Slot a pointed at a node that **branches immediately** (into "$" and "n"), so it carries a real decision and must survive. Only b and n sat above non-branching runs, so only those two slots were touched — exactly the two merges you watched in the animation.

Compare the two **end** rows: slot **b** went from 1 to 7, swallowing six nodes into a seven-character edge — and the cost of that label stayed exactly what it was before, **two integers**. That is the whole reason 23 nodes became 11 with no loss of information.

Three searches on the same tree — three different outcomes

THE RULE

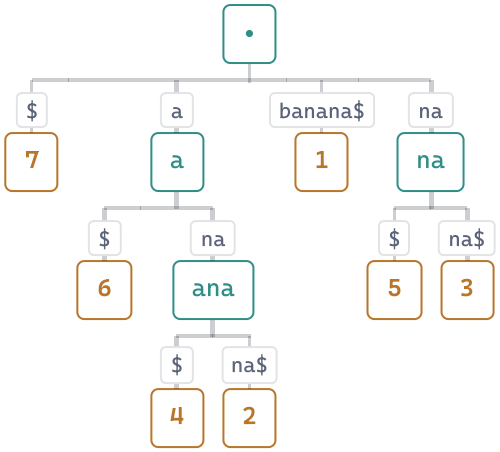
Walk from the root, matching the pattern character-by-character against edge labels. Three things can happen: you land exactly **on an internal node** (pattern occurs — once per leaf below that node); you stop **partway along an edge** having matched the whole pattern (pattern occurs exactly once — wherever that edge's leaf says); or you hit a **mismatch** (pattern does not occur at all).

THREE SEARCHES

Search **"ana"** — lands exactly on an internal node. Search **"ban"** — matches fully, but stops partway along one edge. Search **"bant"** — fails partway along that same edge.

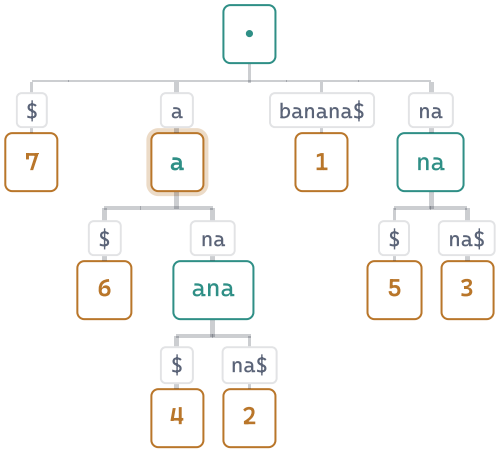
INTERACTIVE — TRACE ALL THREE SEARCHES

STEP 1 OF 9



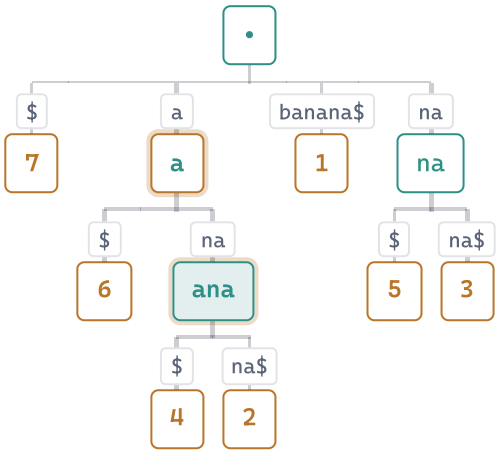
Search 1 — "ana": start at the root.

STEP 2 OF 9



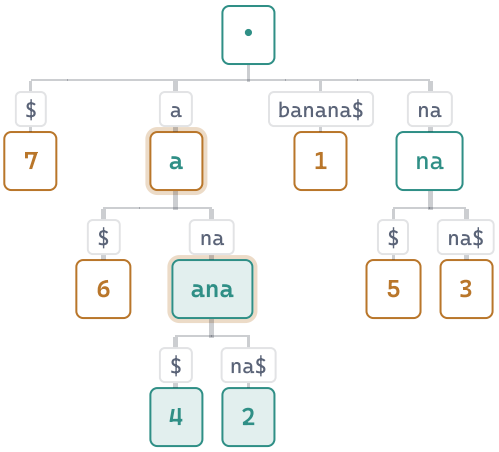
Root has an edge starting with 'a' — labeled "a", leading to the node spelling "a". Follow it (1 of 3 characters matched).

STEP 3 OF 9



From there, the remaining pattern is "na" — there's an edge labeled exactly "na" leading to the node spelling "ana". Follow it fully (3 of 3 characters matched) — we land **exactly on an internal node**.

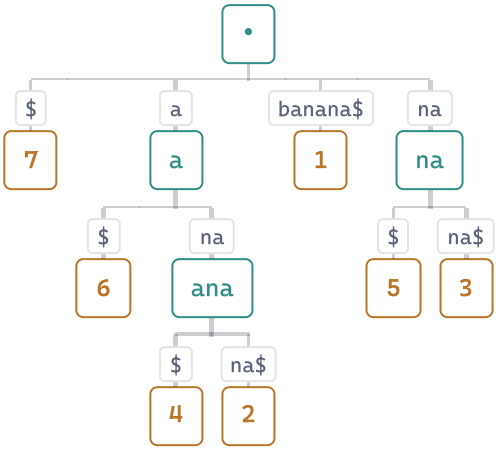
STEP 4 OF 9



OCCURRENCES AT POSITIONS 2 AND 4

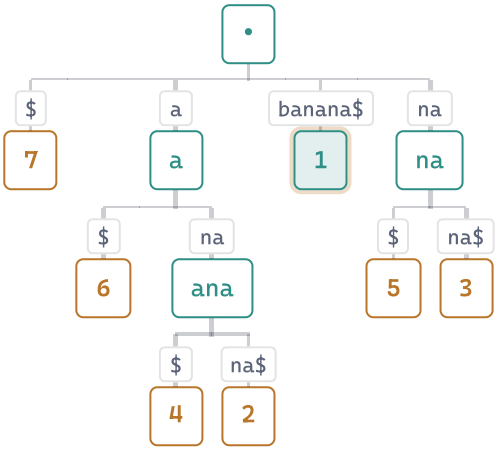
Landing on an internal node means: every leaf below it is one occurrence. Two leaves here — suffix 4 ("ana$") and suffix 2 ("anana$"). **"ana" occurs at positions 2 and 4** of "banana" — matching what we already know is the longest repeated substring.

STEP 5 OF 9



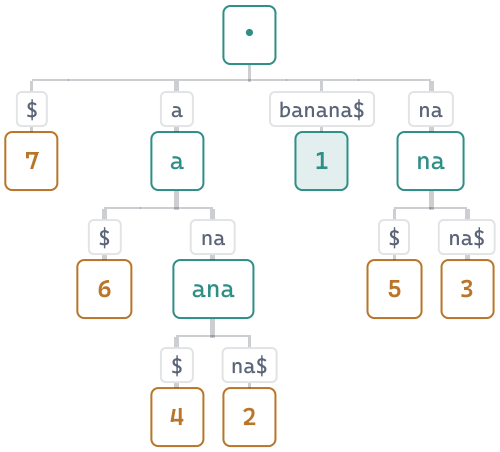
Search 2 — "ban": start at the root again.

STEP 6 OF 9



Root's only 'b'-edge is labeled "banana$" (7 characters, leading straight to leaf 1). Compare "ban" against the first 3 characters of that label — "b","a","n" all match. But the label has 4 more characters ("ana$") — we stop **partway along the edge**, having matched the whole pattern.

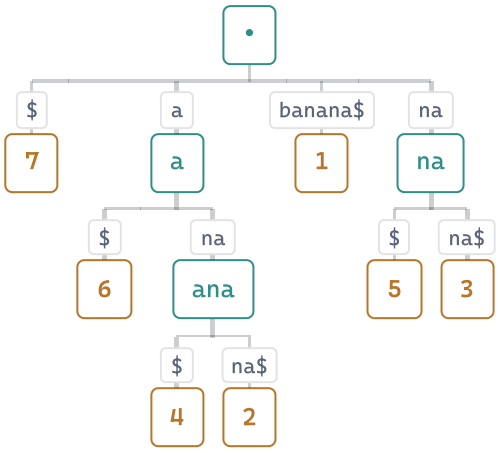
STEP 7 OF 9



UNIQUE OCCURRENCE AT POSITION 1

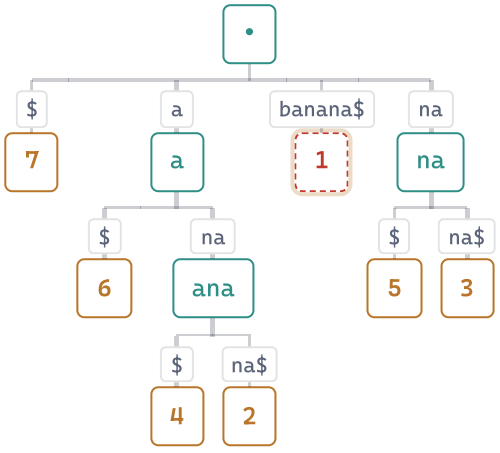
Stopping mid-edge (not at a node) means this edge leads to exactly **one** leaf below it — leaf 1. **"ban" occurs exactly once**, at position 1.

STEP 8 OF 9



Search 3 — "bant": start at the root once more.

STEP 9 OF 9



"BANT" IS NOT A SUBSTRING OF "BANANA"

Same edge, "banana$". First 3 characters match "ban" — but the pattern's 4th character is 't', while the edge's 4th character is 'a'. **Mismatch — search fails.** "bant" does not occur anywhere in "banana".

The deepest branching node — nothing more to compute

FIRST — WHAT THE QUESTION ACTUALLY ASKS

Given one text T, find the **longest string that appears in it at least twice**. Not two different texts — one text, comparing it against itself.

For `banana`, work through the candidates: `"a"` appears 3× (positions 2, 4, 6). `"an"` and `"na"` each appear 2×. `"ana"` appears 2× — at positions 2 and 4. But `"anan"` appears only once, and so does `"nana"`. So the answer is **`"ana"`**, length 3.

ONE RULE THAT SURPRISES PEOPLE — OVERLAPS COUNT

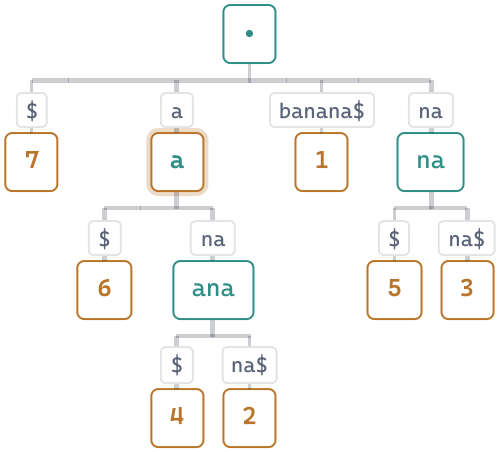
Those two copies of `"ana"` **overlap**: the first covers positions 2–4, the second covers 4–6, and they share position 4. That still counts as repeating twice.

Why it matters in practice: repeated blocks are what compression algorithms hunt for, what plagiarism detectors flag, and what biologists look for as repeated motifs in a genome.

Now the tree does all the work. Every internal node spells out a substring that occurs more than once (that's exactly why it branches — two or more different suffixes continue differently from there). So the **longest repeated substring** is simply whichever internal node is **deepest** in terms of characters spelled, not tree-edges. No scanning of T, no comparing pairs — just find the deepest branching node.

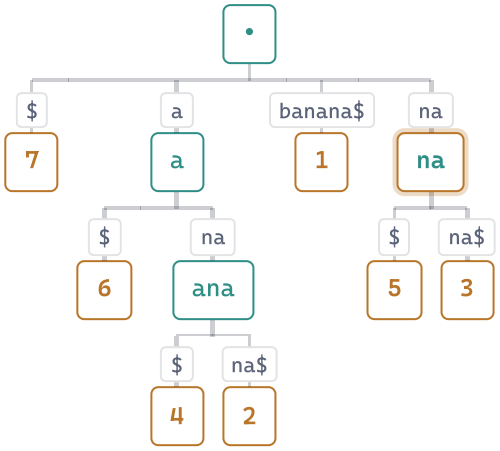
INTERACTIVE — COMPARE THE THREE INTERNAL NODES

STEP 1 OF 4



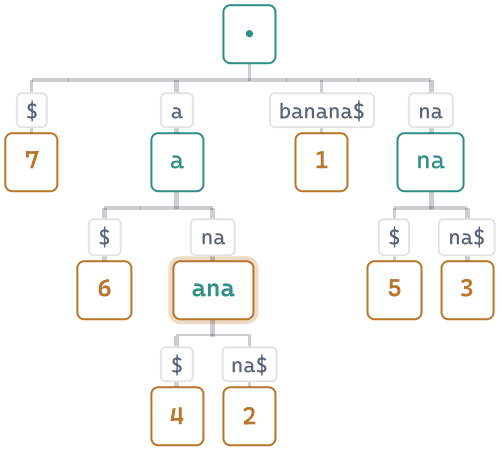
Candidate 1: the node reached by root --"a"--> — string-depth **1**, spells "a". It branches (children "$" and "na"), so "a" does repeat — but is it the *longest* repeat?

STEP 2 OF 4



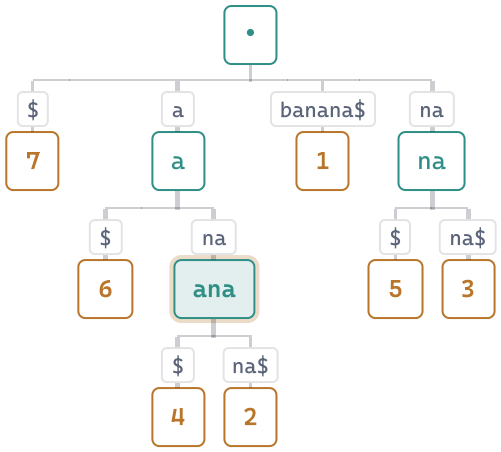
Candidate 2: the node reached by root --"na"--> — string-depth **2**, spells "na". Also branches. Deeper than candidate 1.

STEP 3 OF 4



Candidate 3: the node reached by root --"a"--"na"--> — string-depth **3**, spells "ana". Also branches. Deeper still.

STEP 4 OF 4



LONGEST REPEATED SUBSTRING = "ANA" (DEPTH 3)

Compare all three: depths 1, 2, 3 — the node spelling "ana" is the deepest. There are no other internal nodes to check (only 3 exist besides the root). **Longest repeated substring = "ana"**, occurring at positions 2 and 4 — exactly the pattern-match result from the last animation.

One tree over two texts: the generalized suffix tree

THE IDEA

Append a **different** unique terminator to each text (say # to S1, $ to S2), then build one suffix tree over the concatenation of both. Label every leaf with **which text** its suffix came from.

The **longest common substring** of S1 and S2 is then the deepest internal node whose subtree contains **at least one leaf from each** text — exactly the same "deepest internal node" idea as longest repeated substring, just with an extra bookkeeping condition.

A SMALL VERIFIED EXAMPLE

S1 = `abczz`, S2 = `xyabc`. Checking every possible common substring by hand: single characters a, b, c are common; two-character "ab" and "bc" are common; the three-character **"abc"** is common (S1 positions 1–3, S2 positions 3–5); no four-character substring matches (S1's 4-grams are "abcz","bczz"; S2's are "xyab","yabc" — no overlap).

Longest common substring = "abc", length 3 — found at the deepest node with leaves tagged from both S1 and S2 beneath it.

THE 12 SUFFIXES THAT GO INTO ONE TREE

Two different terminators keep the texts distinguishable — every leaf is tagged with the text it came from and its starting position:

S1 = abczz# — 6 suffixes

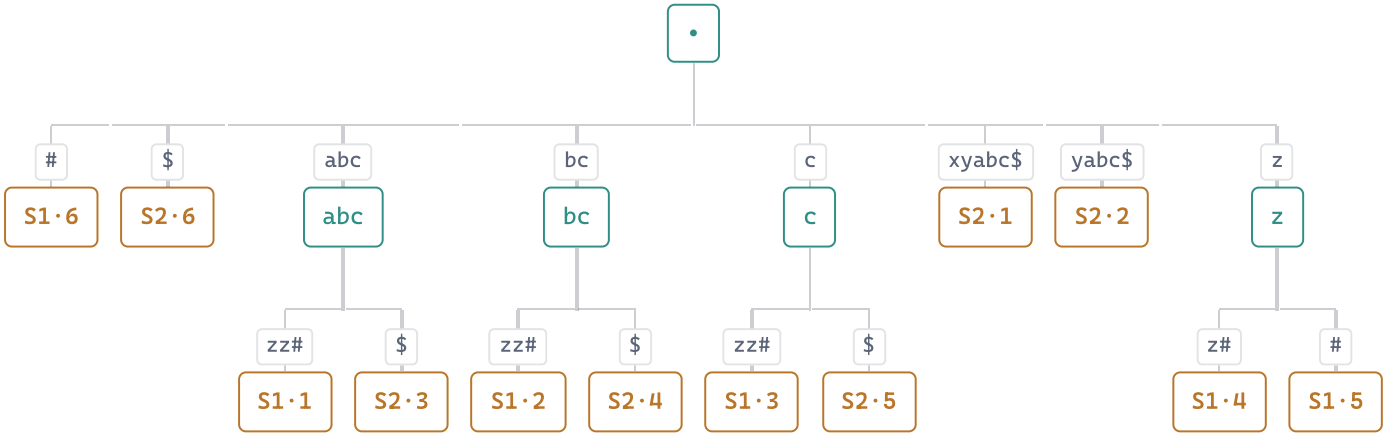
`abczz#  bczz#  czz#  zz#  z#  #`

S2 = xyabc$ — 6 suffixes

`xyabc$  yabc$  abc$  bc$  c$  $`

INTERACTIVE — CHECK EVERY INTERNAL NODE FOR LEAVES FROM BOTH TEXTS

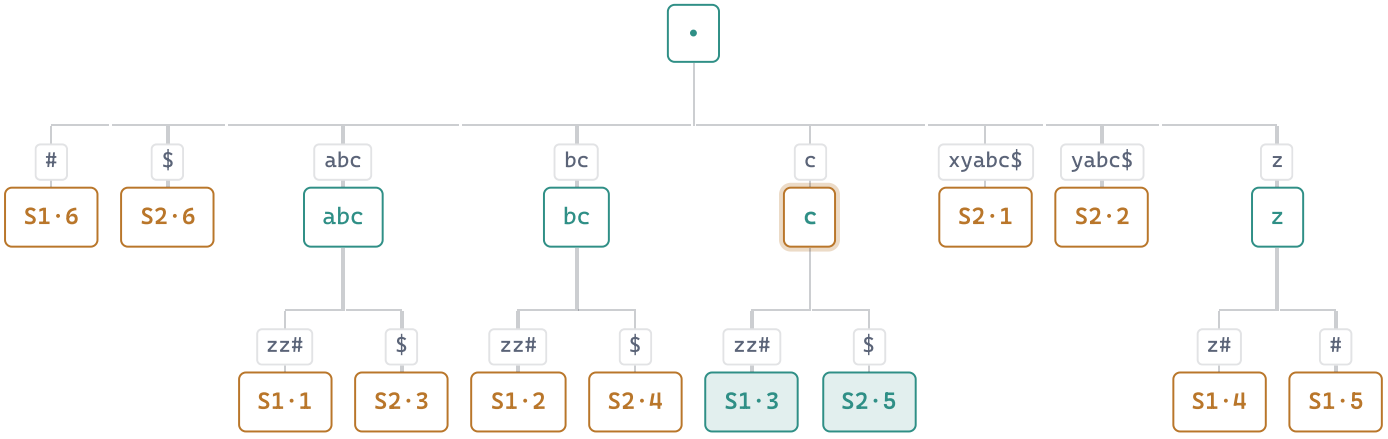
STEP 1 OF 6



ONE TREE, ALL 12 SUFFIXES — 4 INTERNAL NODES TO CHECK: C, Z, BC, ABC

One tree over both texts. Every leaf is tagged with the text it came from. The rule: find the **deepest** internal node whose subtree holds at least one S1 leaf **and** at least one S2 leaf. Four internal nodes exist — check each in turn.

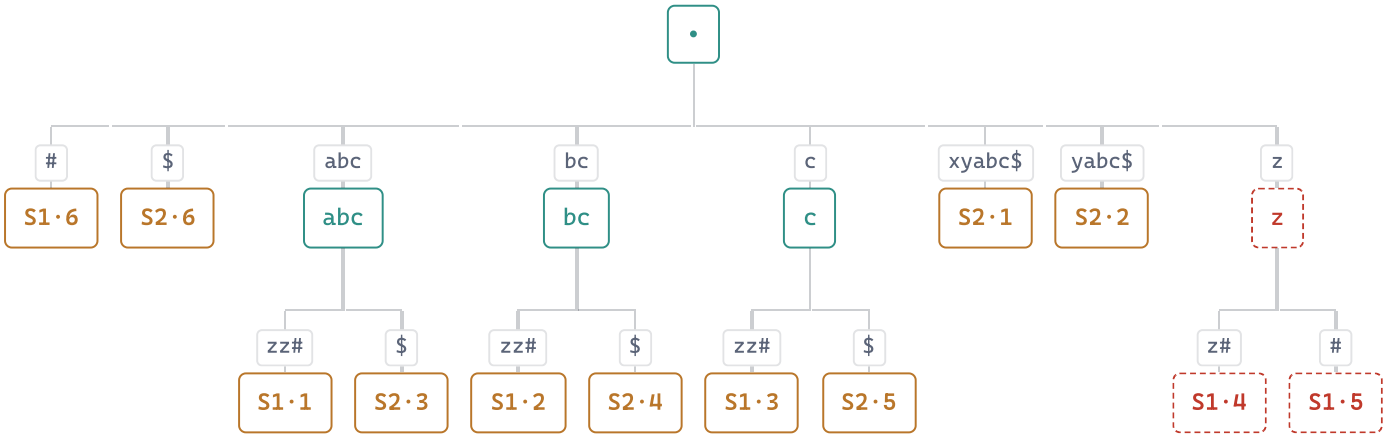
STEP 2 OF 6



NODE "C" — S1·3 AND S2·5 BELOW IT

Node "c" — string-depth **1**. Leaves below: **S1·3** and **S2·5** — one from each text. So "c" **is** a common substring, but only 1 character long. Keep looking for something deeper.

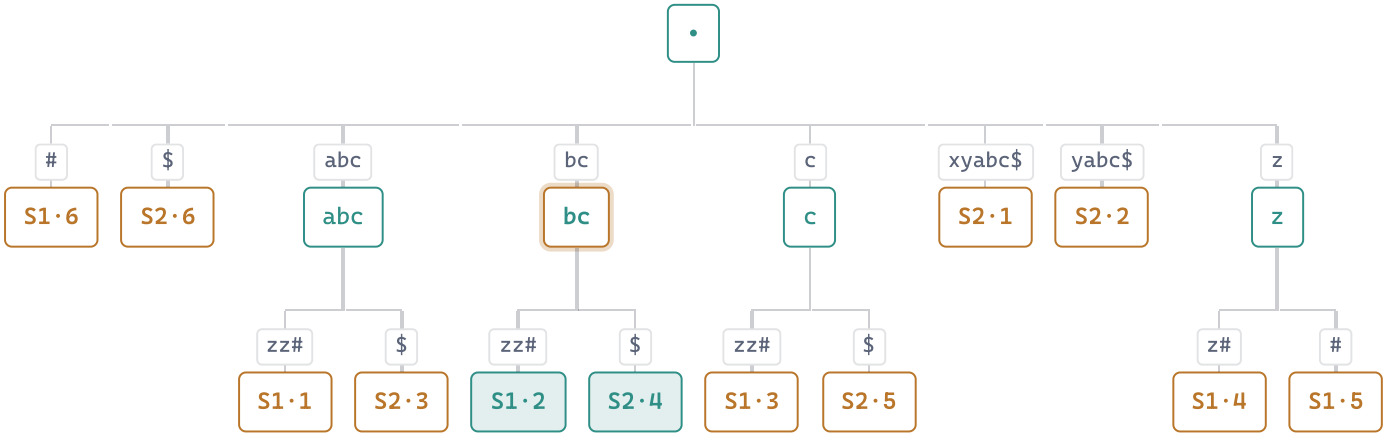
STEP 3 OF 6



NODE "Z" — S1·4 AND S1·5: BOTH FROM S1

Node "z" — string-depth 1. It branches, so "z" *does* repeat — but look at the tags: **S1·4** and **S1·5**, **both from S1**. "z" never appears in S2 at all, so it is **rejected**. This is the extra condition that longest-repeated-substring never had to check.

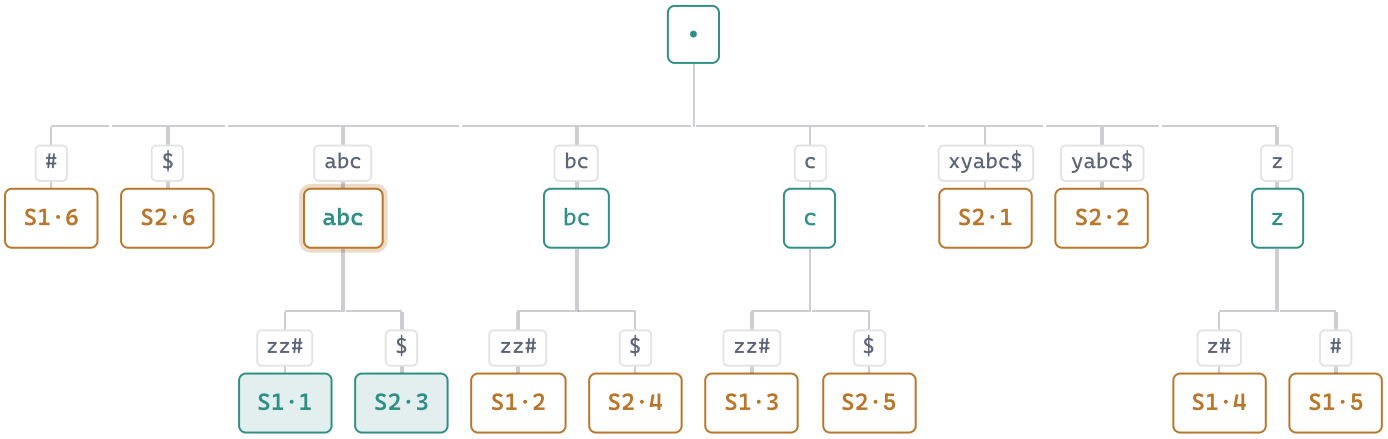
STEP 4 OF 6



NODE "BC" — S1·2 AND S2·4 BELOW IT

Node "bc" — string-depth **2**. Leaves below: **S1·2** and **S2·4** — one from each. Common, and deeper than "c". Best so far.

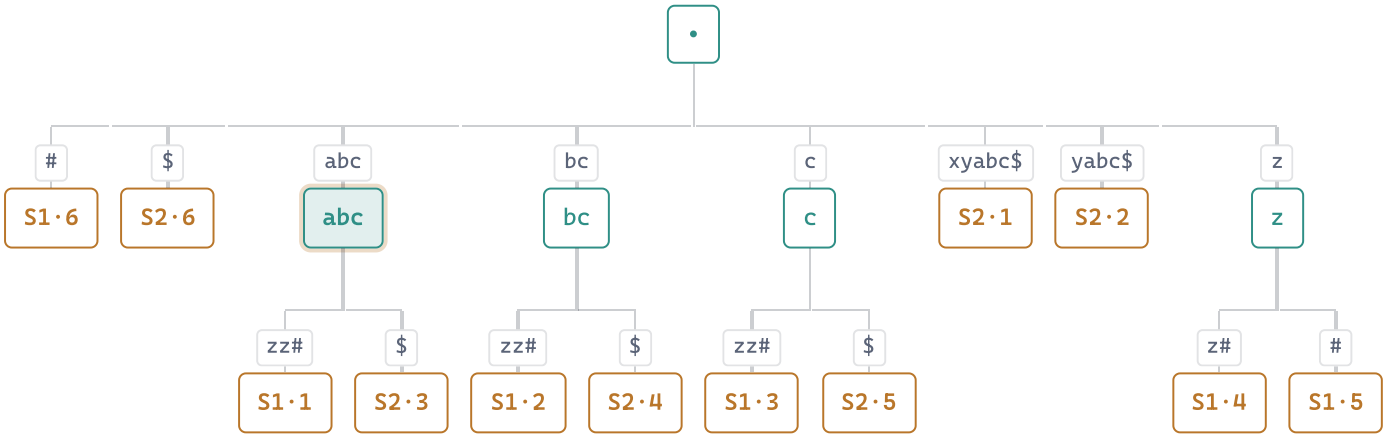
STEP 5 OF 6



NODE "ABC" — S1·1 AND S2·3 BELOW IT

Node "abc" — string-depth **3**. Leaves below: **S1·1** and **S2·3** — one from each. Common, and deeper still. All four internal nodes have now been checked.

STEP 6 OF 6



LONGEST COMMON SUBSTRING = "ABC" (DEPTH 3)

Compare the qualifying nodes: "c" at depth 1, "bc" at depth 2, "abc" at depth 3 — and "z" disqualified for being S1-only. The deepest qualifier wins: **longest common substring = "abc"**, at S1 position 1 and S2 position 3 — matching the by-hand check above. One pass over the tree, no pairwise comparison of substrings anywhere.

Same underlying idea, three different structures

OPERATION	COST
Naive build (insert + compress)	≈ $O(m^2)$ — what we just animated
Linear-time build (not covered here)	O(m) — the standard production approach
Exact pattern search (pattern length p)	O(p)
Longest repeated substring	O(m) — one pass over the already-built tree
Longest common substring (generalized tree)	$O(m_1 + m_2)$ — build once, then one pass

ASPECT	SUFFIX TRIE	SUFFIX TREE	SUFFIX ARRAY
Space	$O(m^2)$ worst case	O(m) guaranteed	O(m), smallest constant
Structure	One char per edge	Compressed, index-range edges	Sorted array of suffix start-positions + LCP array
Build cost	$O(m^2)$ naive	O(m) (linear-time construction)	O(m) (modern linear-time array construction)
Ease of implementation	Simple	Hard — genuinely intricate construction	Moderate; very cache-friendly in practice
Typical real use	Teaching / tiny alphabets only	Bioinformatics, when tree structure itself is needed	Search engines, genome indices (e.g. BWA, bwt-based tools)

Every substring, one tree, no rescanning

- **Every substring is a prefix of some suffix** — so storing every suffix of a text T turns substring questions into prefix searches, exactly like Lecture 13's tries.
- **A naive suffix trie can cost Θ(m^2) space** in the worst case (verified via the $a^n b^n$ growth pattern) — **compressing** every non-branching chain into a single index-range edge fixes this, guaranteeing O(m) space always. Verified today: 23 uncompressed nodes → 11 compressed nodes for "banana$".
- **Pattern matching, longest repeated substring, and longest common substring** are all answered by walking or comparing depths in the same tree — no re-scanning of the original text required, ever.
- The price is a genuinely intricate O(m) construction algorithm — which is exactly why, in practice, many production systems reach for the simpler, more cache-friendly **suffix array** instead, accepting the same asymptotic guarantees with a smaller constant.

LECTURE 15 — NEXT

Comparative Analysis of Advanced Tree Structures

Every balanced and indexing structure from Lectures 7–14, laid side by side — when to reach for which one, and why.

HOMEWORK — BRING TO LECTURE 15

- List the 5 suffixes of "abcab$" (including the terminal). Build the naive suffix trie by hand, then compress it — how many nodes remain?
- On your compressed tree, trace a pattern search for "ab" and for "abz". What does each search return, and why?
- Find the longest repeated substring of "abcab$" using the deepest-internal-node rule — verify it by directly scanning the string for repeats.
- In 3–4 sentences: why does appending a unique terminal character like $ matter for guaranteeing exactly m leaves? What could go wrong without it?
- Reading: Gusfield, *Algorithms on Strings, Trees, and Sequences*, Ch. 5–7 (suffix trees, linear-time construction, generalized suffix trees).

Questions?

Dr. Manu Shrivastava — LHC 308F — Friday 2:00–5:00 PM

> *Next: Lecture 15 — Comparative Analysis of Advanced Tree Structures.*