Tries & Digital Search Trees

Every structure since Lecture 7 has searched strings by comparing them whole. Today we stop comparing strings and start reading them — one character, or one bit, at a time.

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

~65 minutes
Session outcome: apply prefix searching and dictionary operations

Today, part by part

- **Refresher: the catch with storing strings in a BST** 00–03
  O(M log n) instead of O(log n) — comparisons themselves get expensive.
- **Tries: definition** 03–06
  One node per shared prefix; the alphabet becomes the branching factor.
- **Animation: building a trie, 8 words** 06–13
  bear, bell, bid, bull, buy, sell, stock, stop — every shared prefix reused.
- **Animation: searching — a hit and a fast-failing miss** 13–17
  Lookup cost depends only on the word's length, never on how many words are stored.
- **Animation: prefix search — the autocomplete operation** 17–21
  Find every word starting with "bu" without touching anything else.
- **Animation: deleting a word** 21–24
  Unmark, then prune upward — but only as far as it's safe to.
- **Space cost, and compressing the trie** 24–28
  Θ(N·|Σ|) is the price of one array per node — chains of single children collapse away.
- **Why Digital Search Trees? Binary branching on bits** 28–31
  Trade the alphabet-wide array for two pointers and a stored key, like a BST fused with a trie.
- **Digital Search Tree: definition** 31–34
  Branch on bit i at depth i; any key can be the root.
- **Animation: building a DST, 5 keys** 34–40
  0110, 0010, 1001, 1011, 0000 — every bit comparison traced.
- **Animation: searching a DST — a hit and a miss** 40–45
  Unlike a trie, every node visited needs its own key comparison.
- **Animation: deleting from a DST — the tricky case** 45–48
  Why you can't just promote any child, the way you would in a BST.
- **Complexity, and trie vs. DST head-to-head** 48–51
  One comparison vs. one-per-level; array space vs. two pointers.
- **Application: IP routing** 51–57
  Longest-prefix matching — a worked router, traced packet by packet.

Recap & what's next 57–62

Lecture 14: Suffix Trees and String Processing Applications.

Every ordered dictionary you've built assumed cheap comparisons

REFRESHER — LECTURES 7–10

AVL trees, Red-Black trees, Splay trees: all give O(log n) insert, delete, search, successor, predecessor, min/max — **as long as comparing two keys is O(1)**. That's true for integers. It is *not* true for strings.

Comparing two strings of lengths r and s costs O(min(r, s)) — you have to walk both strings to find where they first differ (or confirm they're equal).

THE CATCH

Put strings in a balanced BST and every one of those O(log n) comparisons secretly costs O(M), where M is the longest string in the tree. Total: **O(M log n)**, not O(log n). The tree structure is fine — the comparisons are the hidden cost.

Question: can we design a structure where the cost of an operation depends only on the length of the word we're searching for — *never* on how many words are stored, and never with a repeated re-comparison of characters we've already matched? That structure is the **trie**.

One node per shared prefix — the name comes from "retrieval"

FORMAL DEFINITION

Fix an alphabet Σ. A **trie** is a tree where every node stores:

- A bit marking whether the string spelled out from the root to this node is itself a stored word.
- An array of |Σ| pointers — one possible child per character of the alphabet.

Every node corresponds to exactly one string: the sequence of characters on the path from the root down to it.

WHY THIS IS FAST

Looking up a string w means following at most |w| pointers — each an O(1) array lookup. Total cost: **O(|w|)**. Notice what's missing from that bound: **n**, the number of strings stored. A trie with 10 words and a trie with 10 million words answer the same query in the same time, provided the query word itself is short. That is the entire promise of a trie — and it directly answers last slide's question.

Insert bear, bell, bid, bull, buy, sell, stock, stop

THE INSERTION RULE

```
insert(word):
  v = root
  for each character c in word:
      if v has no child for c:
          create that child       // new node
      v = v.child[c]              // descend
  mark v as "end of word"
```

HOW TO READ THE ANIMATION

Before you watch: each box below is one trie node, and its letter is the **edge** that reaches it from its parent — not the node's entire storage. Every node really holds a full array of 26 pointer slots (one per possible next letter); the diagram only draws whichever slot is actually filled and leaves the other ~25 empty ones out, purely for readability. Those hidden empty slots are exactly what L13·07's space cost — Θ(N·|Σ|) — is warning about: they're still allocated in real memory, one box just doesn't draw the rule to show you the exception.

Amber nodes mark the end of a stored word (a "●" confirms it); **teal** nodes are pure branching points, not words themselves. Watch which letters get **reused** from an earlier word and which are genuinely new — that reuse is the entire space-saving idea of a trie. "bear" and "bell" are walked **one character at a time** so you can see exactly where the pseudocode's create-vs-reuse decision happens; every word after that inserts in a single step, since by then the pattern is established.

WHAT NODE "B" REALLY LOOKS LIKE IN MEMORY, RIGHT AFTER "BEAR" IS INSERTED

The **highlighted** "e" slot is the only non-null pointer — it points to the child node built while inserting "bear". The other 25 slots are `null`, but they're still there, still allocated. The animation's single "e" box *is* that one filled slot; the 25 empty ones are simply never drawn.

INTERACTIVE — INSERT ONE WORD AT A TIME

STEP 1 OF 18

1. bear 2. bell 3. bid 4. bull 5. buy 6. sell 7. stock 8. stop

Start: empty trie — just an unlabeled root. Insert "bear" first. The row above tracks progress: done, current, and still to come.

STEP 2 OF 18

1. bear 2. bell 3. bid 4. bull 5. buy 6. sell 7. stock 8. stop

Insert "bear" — step 1/5: at root, no child for "b" yet → **create** it. Descend into "b".

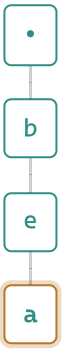
STEP 3 OF 18

1. bear 2. bell 3. bid 4. bull 5. buy 6. sell 7. stock 8. stop

Insert "bear" — step 2/5: at "b", no child for "e" yet → **create** it. Descend into "e".

STEP 4 OF 18

1. bear 2. bell 3. bid 4. bull 5. buy 6. sell 7. stock 8. stop



Insert "bear" — step 3/5: at "e", no child for "a" yet → **create** it. Descend into "a".

STEP 5 OF 18

1. bear | 2. bell | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop

Insert "bear" — step 4/5: at "a", no child for "r" yet → **create** it. Descend into "r" — the last character of "bear".

STEP 6 OF 18

1. bear | 2. bell | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop

Insert "bear" — step 5/5: end of the word → **mark** "r" as a word-end (●). "bear" is fully stored: b→e→a→r, four brand-new nodes.

STEP 7 OF 18

1. bear | **2. bell** | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop

Insert "bell" — step 1/5: at root, child "b" already exists → **reuse** it, don't recreate. Descend into "b".

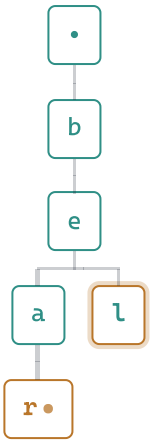
STEP 8 OF 18

1. bear | **2. bell** | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop

Insert "bell" — step 2/5: at "b", child "e" already exists → **reuse** it. Descend into "e".

STEP 9 OF 18

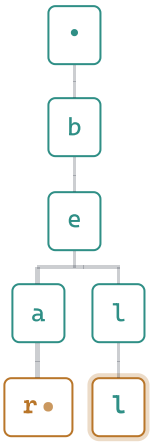
1. bear | **2. bell** | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop



Insert "bell" — step 3/5: at "e", no child for "l" yet → **create** it — "e" now branches two ways: "a" (from bear) and this new "l".

STEP 10 OF 18

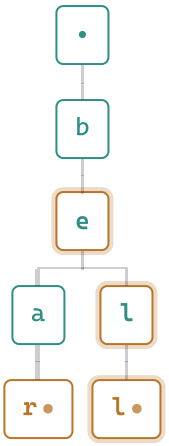
1. bear | **2. bell** | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop



Insert "bell" — step 4/5: at "l", no child for the second "l" yet → **create** it. Descend into it — the last character of "bell".

STEP 11 OF 18

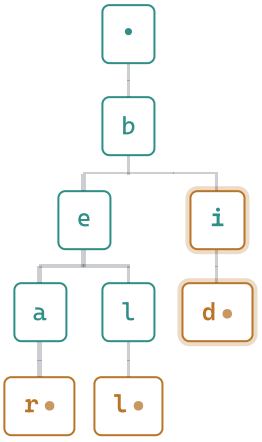
1. bear | **2. bell** | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop



Insert "bell" — step 5/5: end of the word → **mark** the second "l" as a word-end (●). "bell" is fully stored: b→e→l→l, sharing "b" and "e" with "bear", only 2 nodes new.

STEP 12 OF 18

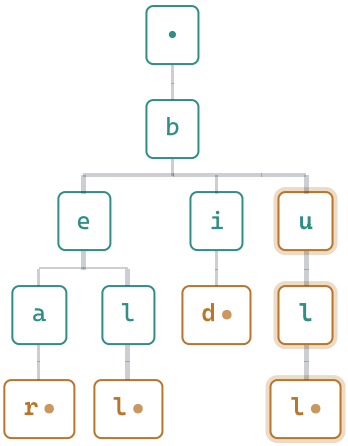
1. bear | 2. bell | **3. bid** | 4. bull | 5. buy | 6. sell | 7. stock | 8. stop



Insert "bid": "b" reused again. "i" and "d" are new, added as a fresh branch directly off "b", alongside the existing "e" branch.

STEP 13 OF 18

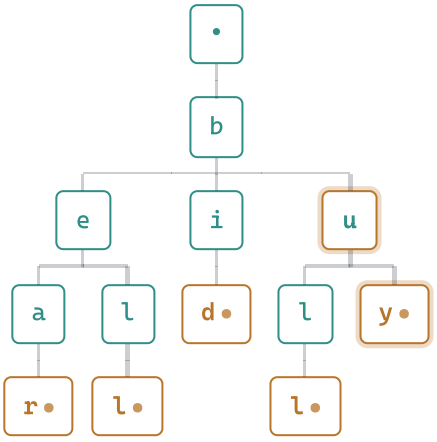
1. bear | 2. bell | 3. bid | **4. bull** | 5. buy | 6. sell | 7. stock | 8. stop



Insert "bull": "b" reused. "u", "l", "l" are new — a third branch off "b".

STEP 14 OF 18

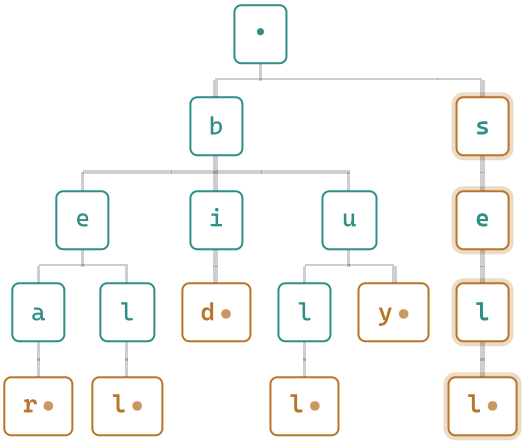
1. bear | 2. bell | 3. bid | 4. bull | **5. buy** | 6. sell | 7. stock | 8. stop



Insert "buy": "b" and "u" reused. Only "y" (word-end) is new, branching off "u" alongside the existing "l".

STEP 15 OF 18

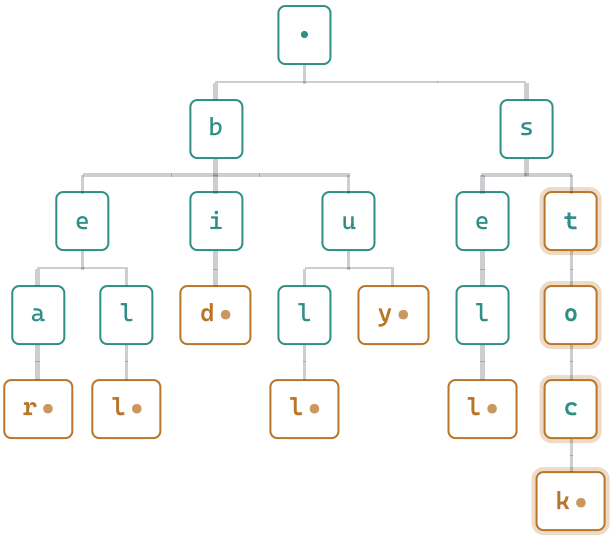
1. bear | 2. bell | 3. bid | 4. bull | 5. buy | **6. sell** | 7. stock | 8. stop



Insert "sell": the root gains its second-ever branch, "s" — nothing shared with any "b" word. "e", "l", "l" are all new.

STEP 16 OF 18

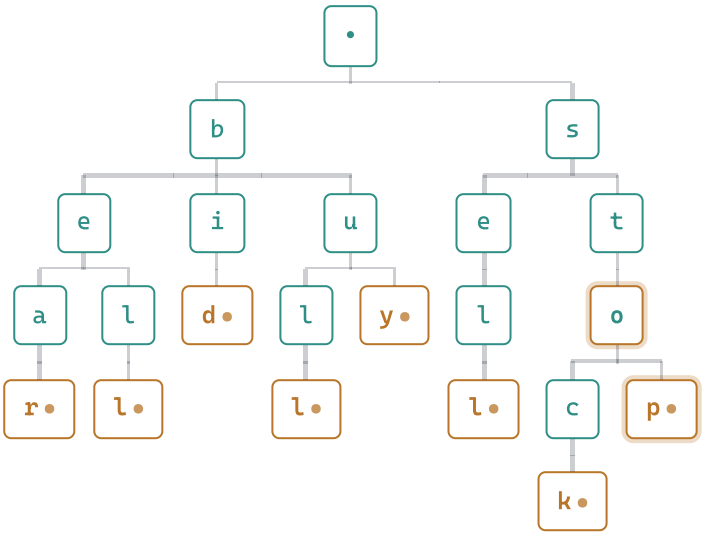
1. bear | 2. bell | 3. bid | 4. bull | 5. buy | 6. sell | **7. stock** | 8. stop



Insert "stock": "s" reused. "t", "o", "c", "k" are new — a second branch off "s", alongside "e".

STEP 17 OF 18

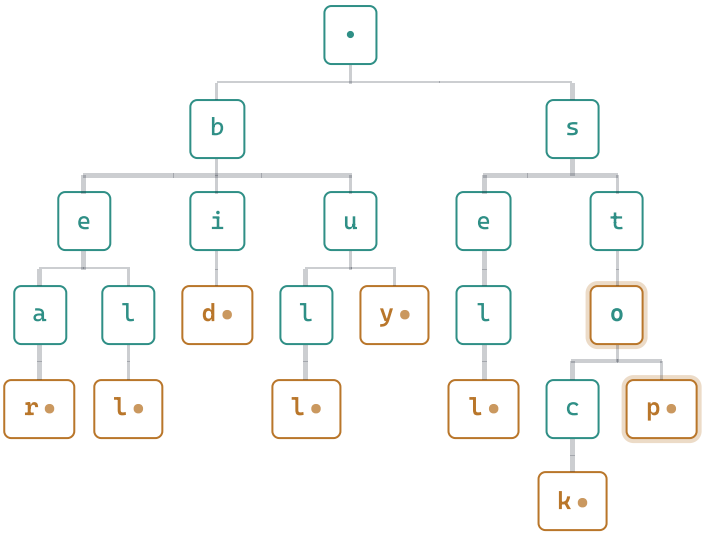
1. bear | 2. bell | 3. bid | 4. bull | 5. buy | 6. sell | 7. stock | **8. stop**



Insert "stop": "s", "t", "o" reused. Only "p" (word-end) is new, branching off "o" alongside the existing "c". All 8 words placed.

STEP 18 OF 18

1. bear 2. bell 3. bid 4. bull 5. buy 6. sell 7. stock 8. stop



ALL 8 WORDS PLACED — 8 WORD-END (•) NODES, 22 TOTAL NODES

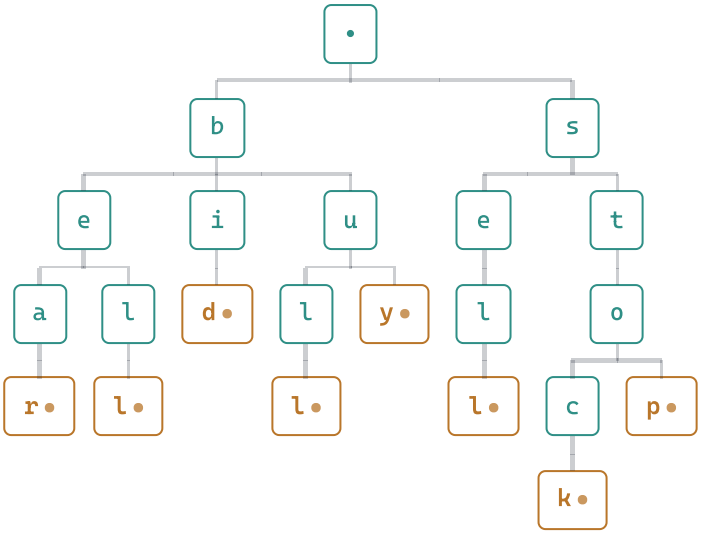
Done. 8 words, 22 nodes total (versus 8 × average length ≈ 3.9 ≈ 31 characters if nothing were shared). Every shared prefix — "b", "be", "bu", "s", "st" — was written into the trie exactly once.

A hit costs |word| steps. A miss can cost far fewer.

Search 1: look up **"bull"**, which is stored. Search 2: look up **"buzz"**, which is not — watch exactly where it falls off the tree.

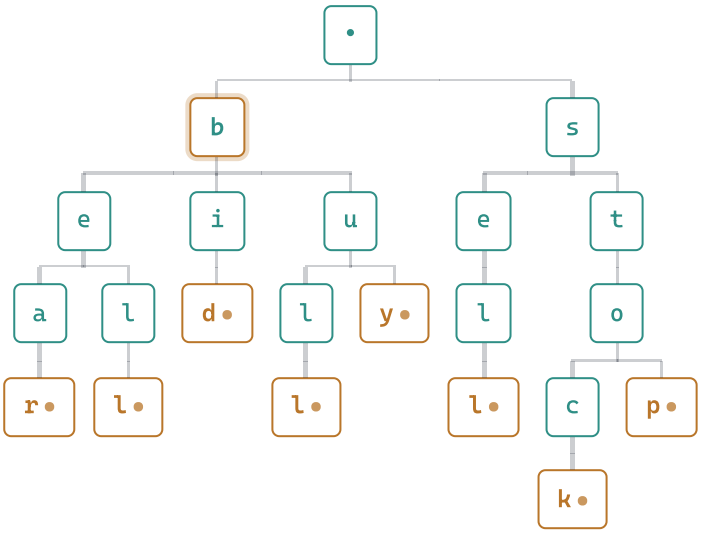
INTERACTIVE — TRACE BOTH SEARCHES

STEP 1 OF 9



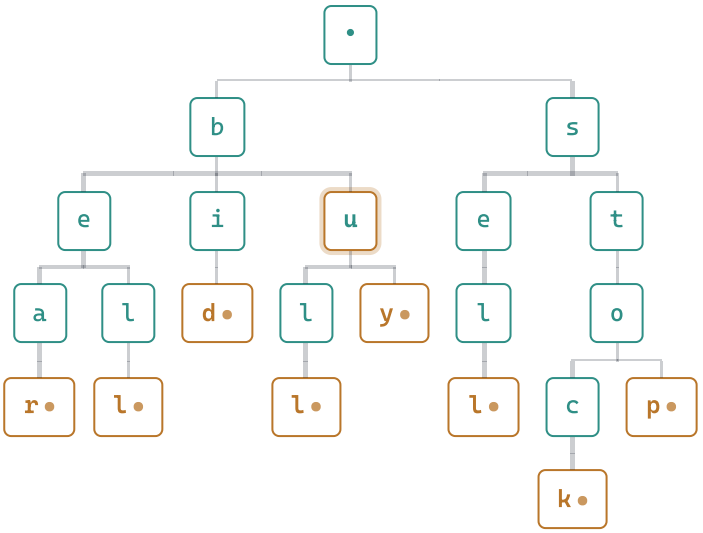
Search 1: look up "bull". Start at the root.

STEP 2 OF 9



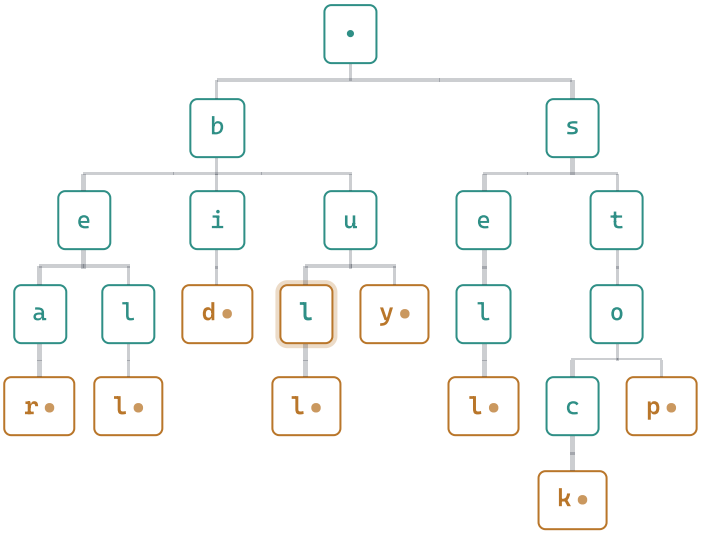
Root has a child "b" — follow it.

STEP 3 OF 9



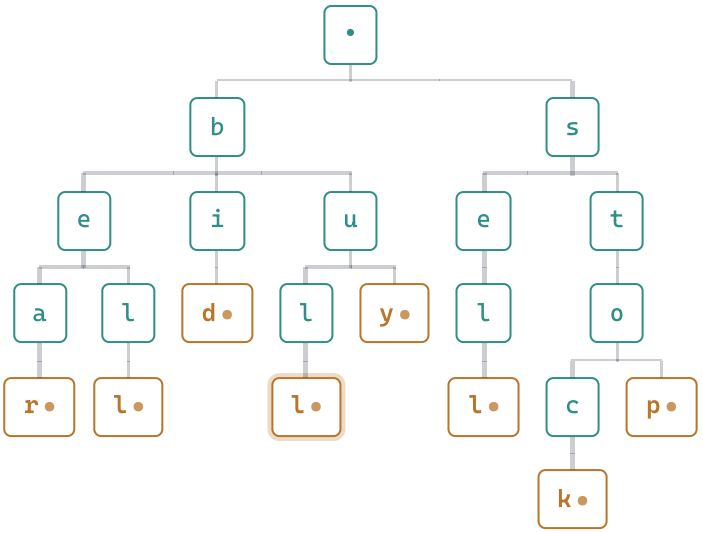
"b" has a child "u" — follow it.

STEP 4 OF 9



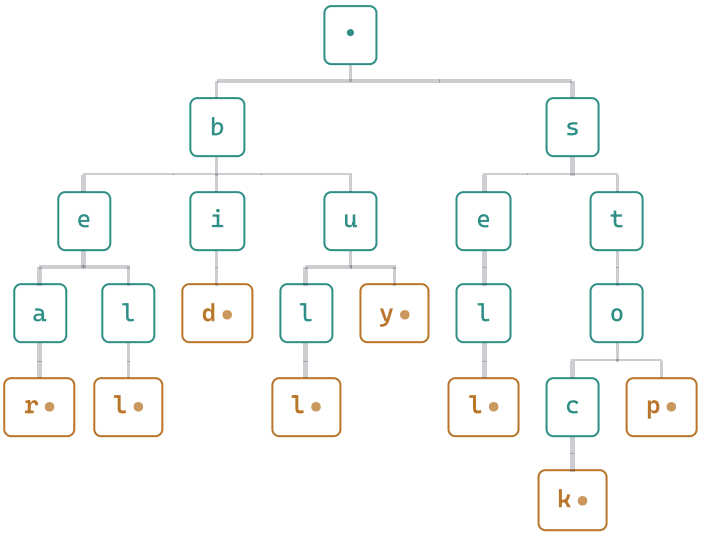
"u" has a child "l" — follow it.

STEP 5 OF 9



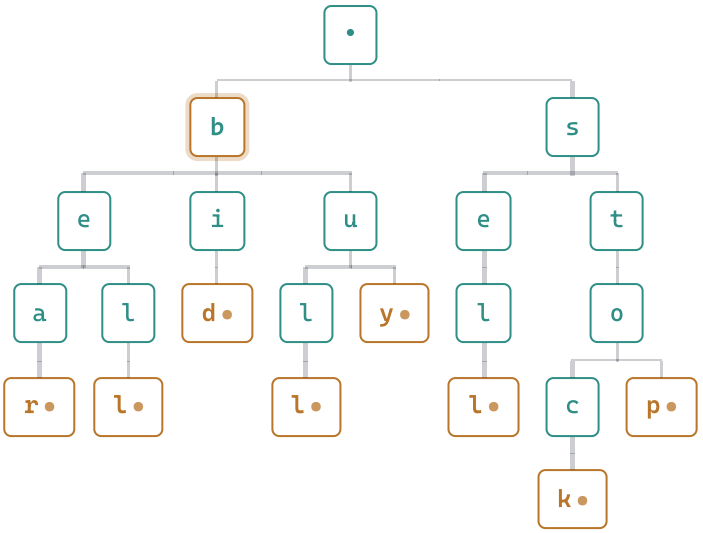
That "l" has a child "l" — follow it. String exhausted (4 letters, 4 steps) — and this node IS marked as a word-end. **"bull" found.** Total cost: 4 steps, regardless of the other 7 words stored.

STEP 6 OF 9



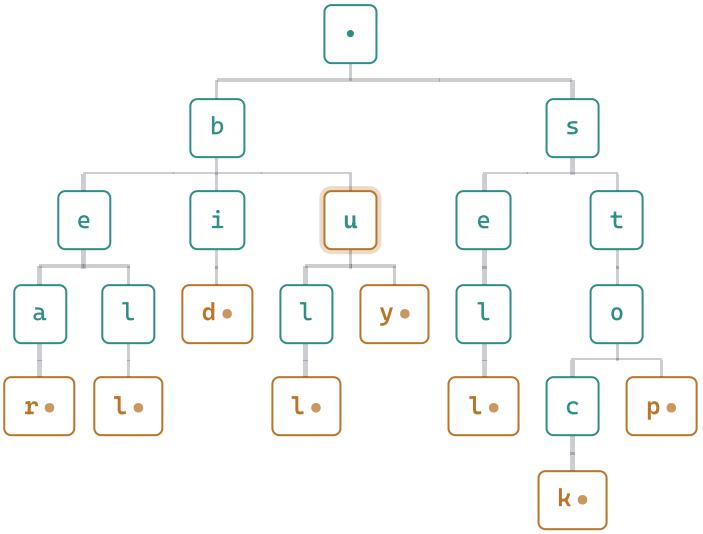
Search 2: look up "buzz". Start at the root again.

STEP 7 OF 9



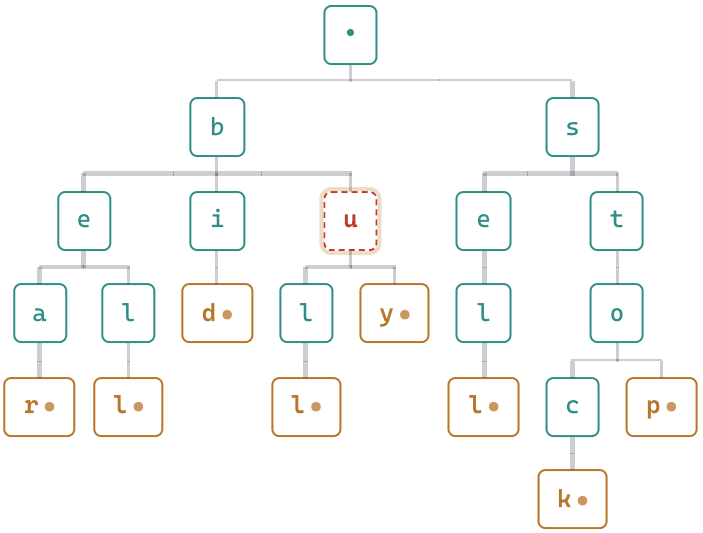
Root has a child "b" — follow it. (1 letter matched: "b")

STEP 8 OF 9



"b" has a child "u" — follow it. (2 letters matched: "bu")

STEP 9 OF 9



Need a child "z" from node "u" — but "u" only has children "l" and "y". **No child "z" exists — "buzz" is NOT in the trie.** The search failed after only 2 character comparisons, without ever looking at the 2 remaining letters of "buzz".

Autocomplete: every word that starts with "bu"

THE OPERATION

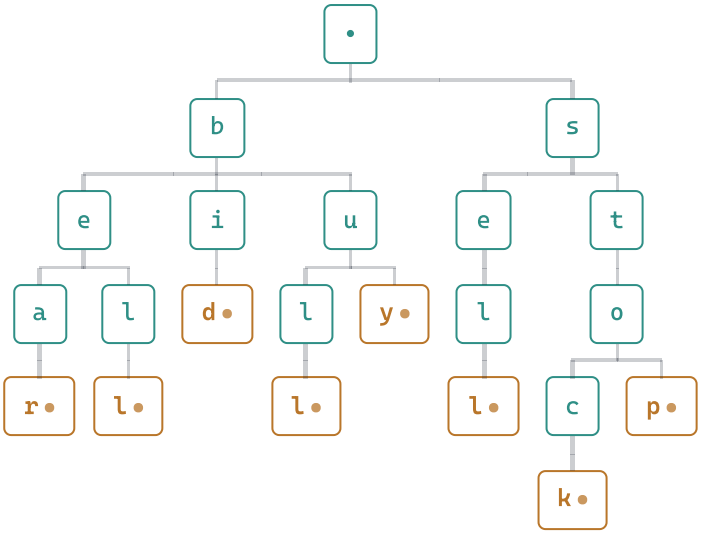
Descend along the prefix's own characters to find the node for "bu" — exactly like a search, but you don't need that node to be a word-end itself. Then explore **every** node in the subtree below it, collecting every word-end found. This is precisely what an autocomplete box, or a phone contact search, does on every keystroke.

WHY A BST CAN'T DO THIS AS CHEAPLY

A BST ordered by string comparison would need a *range query* ("everything between 'bu' and 'bv'") — correct, but still paying string-comparison costs at O(log n) nodes. A trie finds the prefix node in O(|prefix|) and then just walks a subtree — no comparisons against unrelated words at all.

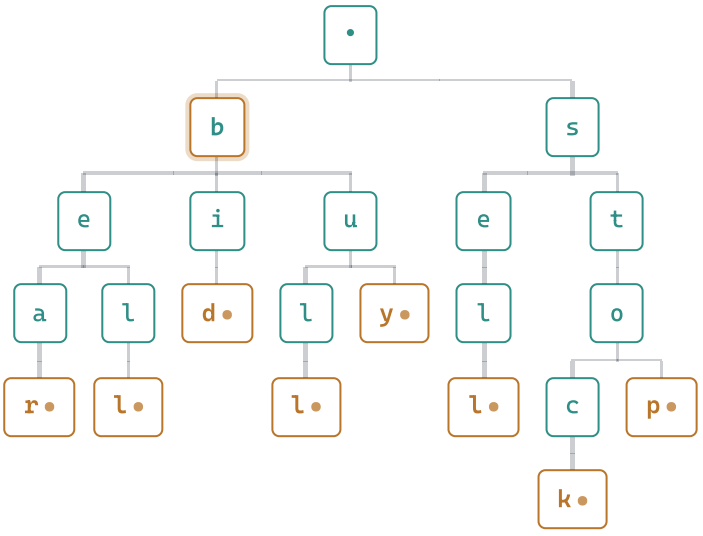
INTERACTIVE — FIND THE PREFIX NODE, THEN COLLECT EVERY WORD BELOW IT

STEP 1 OF 6



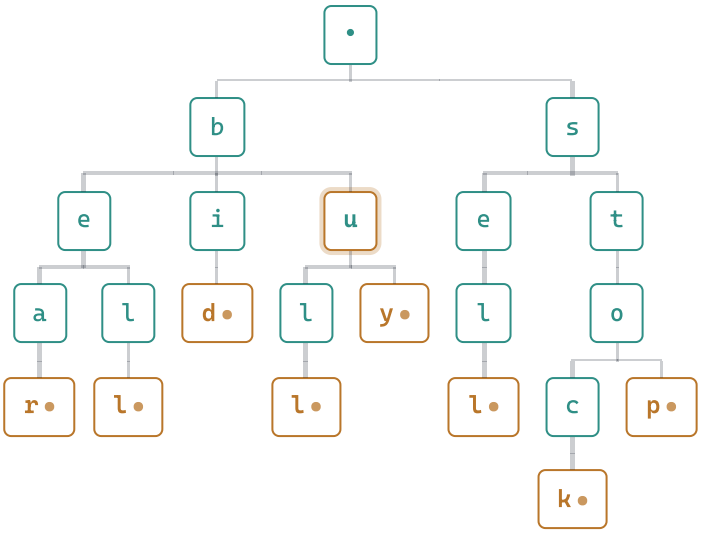
Prefix query: find every word starting with "bu". Start at the root.

STEP 2 OF 6



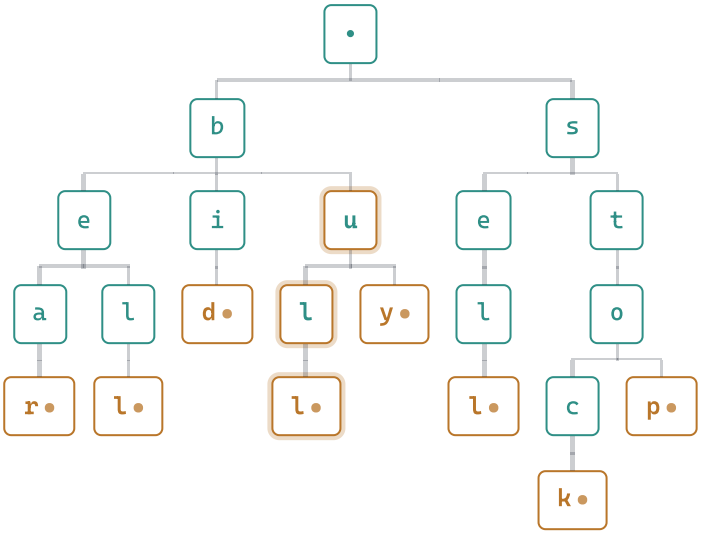
Root has a child "b" — follow it. (matched: "b")

STEP 3 OF 6



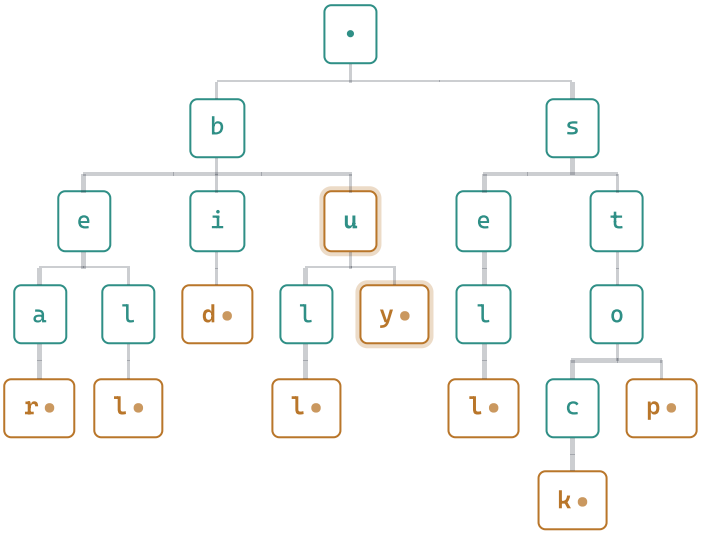
"b" has a child "u" — follow it. (matched: "bu"). This is the **prefix node** — note it is NOT itself a word-end.

STEP 4 OF 6



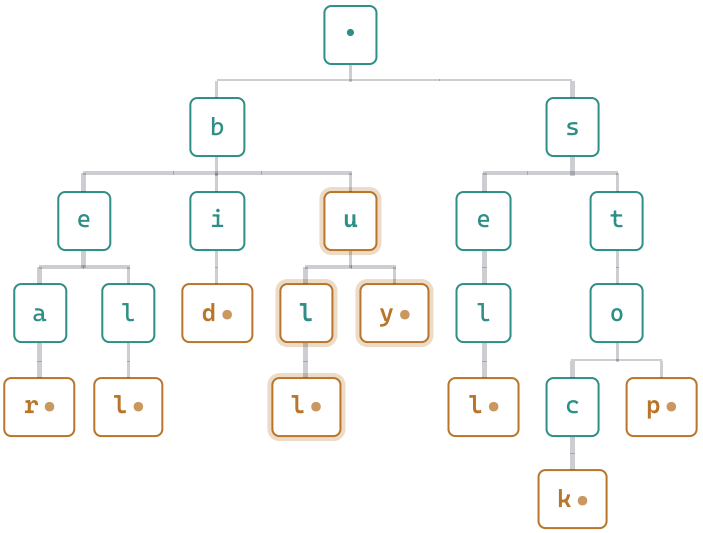
Explore every path below the prefix node. First branch: "l" → "l", and that second "l" is a word-end. **Found: "bull".**

STEP 5 OF 6



Second branch below "bu": "y", also a word-end. **Found: "buy".**

STEP 6 OF 6



RESULTS = { BULL, BUY }

Done. Prefix "bu" → exactly {"bull", "buy"} — found in 2 steps to locate the prefix, plus a small subtree walk, never touching "bear", "bell", "bid", "sell", "stock", or "stop" at all.

Unmark, then prune upward — but stop the moment it's unsafe

THE DELETION RULE

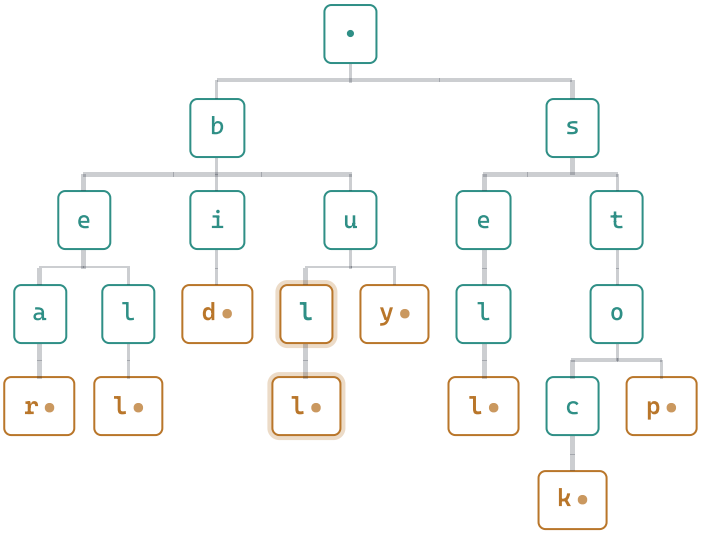
```
delete(word):
  v = the node found by searching for word
  unmark v as "end of word"
  while v has no children and v is not a word-end:
      parent = v's parent
      remove v from parent
      v = parent
```

WHAT TO WATCH FOR

Delete **"bull"**. Its path shares letters with "buy" — the pruning must stop *exactly* at the node still needed by "buy", and go no further. Removing one node too many would silently delete a different word.

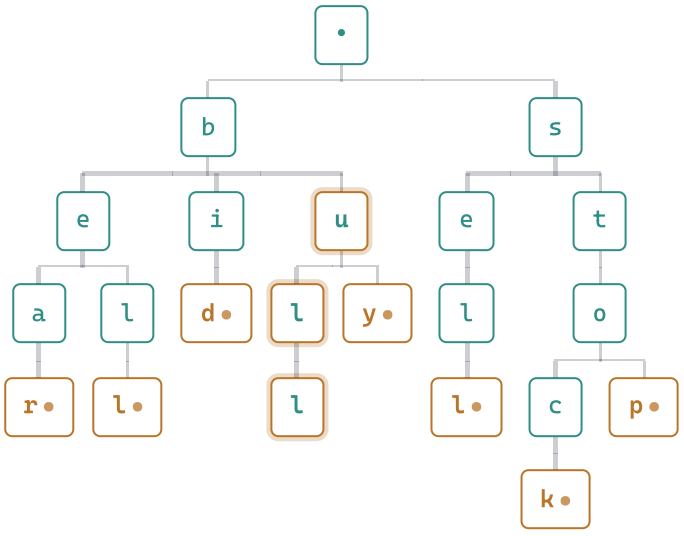
INTERACTIVE – DELETE "BULL"

STEP 1 OF 4



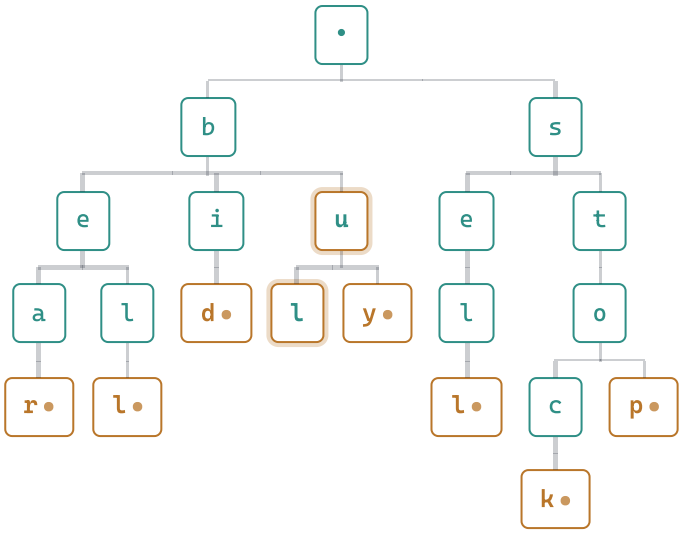
Delete "bull": path is b → u → l → l. "u" is shared with "buy" — watch the pruning stop before it touches "u".

STEP 2 OF 4



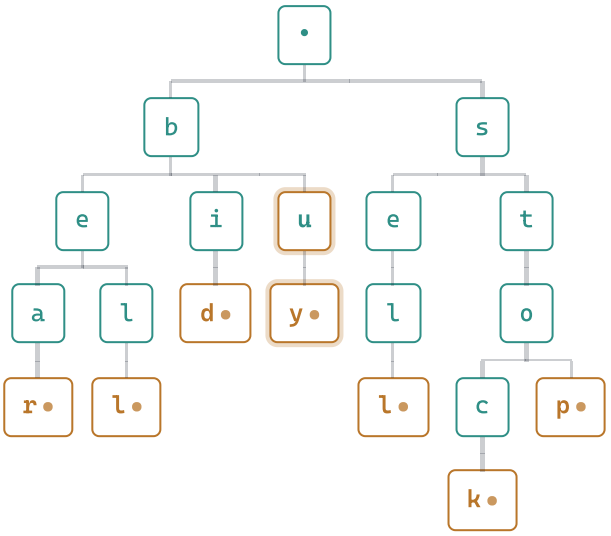
Step 1: unmark the last "l" as a word-end. It now has no children AND is not a word — **remove it.**

STEP 3 OF 4



Step 2: move up to the first "l". It now has no children (its only child was just removed) and it is not a word itself — **remove it too.**

STEP 4 OF 4



"BULL" FULLY REMOVED — "BUY" UNTOUCHED

Step 3: move up to "u". It still has a child — "y", needed by "buy" — so pruning **stops here**. "bull" is completely gone; "buy" is completely intact.

The price of speed: one array per node

THE SPACE PROBLEM

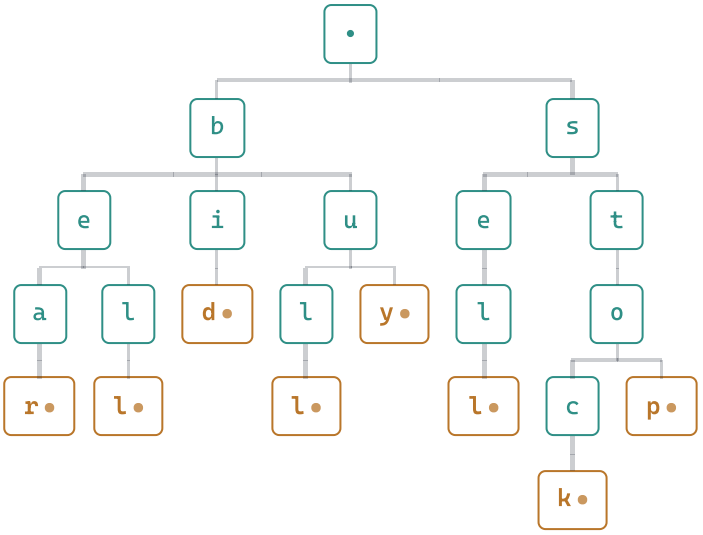
A trie with N nodes needs **Θ(N · |Σ|)** space — every node carries a full array of |Σ| pointers, even when only one or two are ever used. For English text (|Σ| = 26) that's wasteful; for a genome (|Σ| = 4) it's less severe, but the idea generalizes badly to large alphabets.

THE FIX: COMPRESS SINGLE-CHILD CHAINS

Whenever a node has exactly one child and is not itself a word-end, it's carrying no branching information — merge it into the edge above it. Repeat until every remaining node either branches or ends a word. This is a **compressed trie** (also called a PATRICIA-style trie), and it can be stored in O(s) space, where s is the number of words, using index ranges instead of full substrings at each node.

INTERACTIVE — BEFORE AND AFTER, SAME 8 WORDS

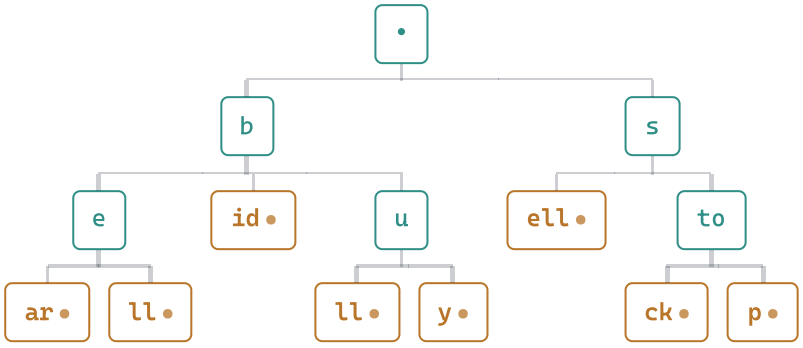
STEP 1 OF 2



STANDARD TRIE — 22 NODES FOR 8 WORDS

Before: the standard trie we built, one character per node. Chains like b→e→a→r or b→i→d carry single children the whole way down — pure overhead, no branching decision happening.

STEP 2 OF 2



COMPRESSED TRIE — 14 NODES, SAME 8 WORDS

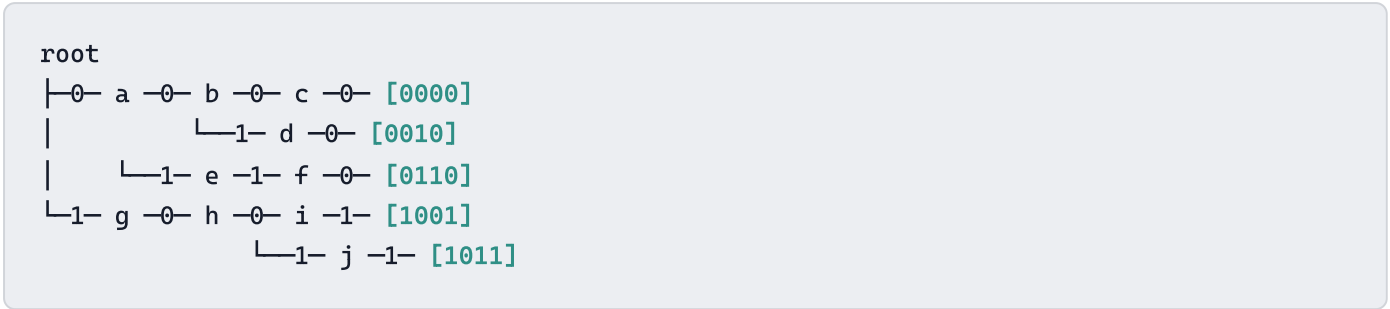
After: every chain of single-child, non-word nodes is merged into one edge holding the whole substring — "ar", "ll", "id", "ell", "to", "ck" each replace what used to be 2–3 separate nodes. Same 8 words, same lookup guarantee, fewer nodes — and with index-range storage (pointers into the original strings instead of copied substrings) this needs only O(s) space for s stored words.

Count the nodes: 16 to store 5 keys, or 5?

Suppose your keys aren't English words but fixed-length **bit strings** — IPv4 addresses, IDs, hash codes. Here |Σ| = 2, so a trie node's array is only 2 pointers, and the alphabet-size problem from L13·07 has gone away. So why change anything? Put the **same five 4-bit keys** into both structures and count.

BINARY TRIE — 16 NODES FOR 5 KEYS

Every key is 4 bits, so every key needs a 4-edge path, and only the **leaf** is the key. Everything above it exists purely to be walked through:



1 root + 2 + 3 + 5 + 5 leaves = **16 nodes**. Eleven of them (a…j and the root) store **nothing**. For n keys of b bits, that's up to **O(n·b)** nodes — the cost now scales with key *length*, not alphabet size.

DIGITAL SEARCH TREE — 5 NODES FOR 5 KEYS

Store the **actual key** at every node — like a BST — but decide left-or-right using **one bit** of the key at each depth instead of comparing whole keys. Depth 1 uses bit 1, depth 2 uses bit 2, and so on.

Every node is now a real, useful key; nothing is a pure router. Five keys, **exactly five nodes** — you'll build this very tree in the next animation, and it comes out at height 3. That is a **Digital Search Tree (DST)**.

AND WHY NOT JUST USE A BST?

A plain BST over these keys can degenerate into a chain depending on insertion order; AVL and Red-Black fix that, but only by carrying balance factors or colour bits and doing **rotations** on the way back up. A DST branches on *bits of the key* rather than on comparisons against what's already stored — so it stays short by construction. Watch the next animation finish with **zero rotations** and no balance metadata anywhere.

WHAT IT COSTS — THE HONEST TRADE

Nothing is free. In a trie only the **final** node needs a real check; every step before it is a bare array lookup. In a DST every node holds a genuine key, so you pay a **full key comparison at every level** (L13·11 traces exactly this). That trade only pays off when a comparison is cheap — which is precisely the case for fixed-length keys that fit in a machine word: a 32-bit address, an ID, a hash code.

Branch on bit i at depth i — any key may be the root

FORMAL DEFINITION (FIXED-LENGTH KEYS)

- An empty DST is empty. A non-empty DST's root holds **any one** key-value pair from the set — there's no ordering requirement on which key goes where.
- All remaining pairs whose key's **first** bit is 0 go into the **left** subtree; all whose first bit is 1 go into the **right** subtree.
- The left and right subtrees are themselves Digital Search Trees, built on the **remaining** bits (bit 2 decides the split one level down, bit 3 the level after that, and so on).

INSERTION, PRECISELY

The first key inserted becomes the root. Every later key descends from the root using its **own** bits, read most-significant-bit first, choosing left (bit = 0) or right (bit = 1) at each level — until it reaches an empty pointer, where it is inserted as a brand-new node. Its final depth is however many bits it took to find that empty slot.

Insert 0110, 0010, 1001, 1011, 0000 — 4-bit keys, MSB first

Every level is its own step, and the panel above the tree shows all three things happening at that level: the **key in hand** with the bit currently being read highlighted, the **node we are standing at** and which bit its depth makes it branch on, and the **comparison** that follows — first "is this a duplicate?", then "which way does that bit send us?". Empty slots are drawn as `NIL`, so you can see the gap before the key drops into it.

INTERACTIVE – INSERT FIVE 4-BIT KEYS

STEP 1 OF 13

EMPTY TREE

Start: empty Digital Search Tree. Bits are read most-significant-bit first. At every node we do two things: **compare** the incoming key against the key stored there, and — if they differ — **branch on one bit**. Insert 0110, 0010, 1001, 1011, 0000 in that order.

STEP 2 OF 13

INSERTING	0 1 1 0
THEN	Tree is empty → 0110 becomes the **root**, depth 1.

0110

Insert 0110. Nothing to compare against and no bit to read — the first key simply becomes the root. Its own value plays no special role; any key could have been first.

STEP 3 OF 13

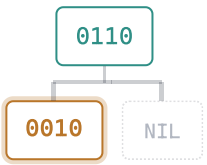
INSERTING	0 0 1 0 reading bit 1
AT NODE	0110 depth 1 → branches on bit 1
THEN	Is 0010 = 0110? **No.** Bit 1 of 0010 is **0** → go **LEFT**.

0110

Insert 0010 — at the root. First the comparison: 0010 is not 0110, so this is not a duplicate. The root sits at depth 1, so it branches on **bit 1** — which is 0, sending us left.

STEP 4 OF 13

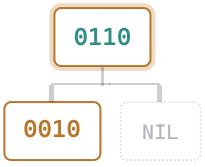
INSERTING	0 0 1 0 reading bit 1
THEN	The root's left slot is **empty** → place 0010 here, depth 2.



Insert 0010 — placed. The slot we were sent to was empty (it showed as NIL a moment ago), so the key lands there. One comparison, one bit read.

STEP 5 OF 13

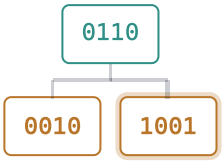
INSERTING	1 0 0 1	reading bit 1
AT NODE	0110	depth 1 → branches on bit 1
THEN	Is 1001 = 0110? **No.** Bit 1 of 1001 is **1** → go **RIGHT**.	



Insert 1001 — at the root. Not a duplicate. Depth 1 → read bit 1 of 1001, which is 1, so we go right this time. Note the right slot is drawn as NIL — that is the empty slot we are about to fill.

STEP 6 OF 13

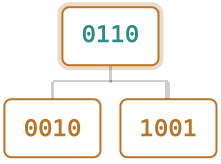
INSERTING	1 0 0 1	reading bit 1
THEN	The root's right slot is **empty** → place 1001 here, depth 2.	



Insert 1001 — placed. Again a single hop: one comparison at the root, one bit read, one empty slot found.

STEP 7 OF 13

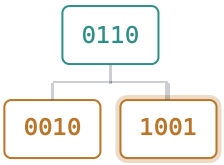
INSERTING	1 0 1 1	reading bit 1
AT NODE	0110	depth 1 → branches on bit 1
THEN	Is 1011 = 0110? **No.** Bit 1 of 1011 is **1** → go **RIGHT**.	



Insert 1011 — at the root. Not a duplicate. Bit 1 is 1, so go right — but this time the slot is *not* empty: 1001 already lives there. So we descend into it rather than stopping.

STEP 8 OF 13

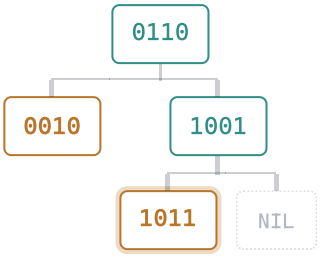
INSERTING	1 **0** 1 1	reading bit 2
AT NODE	1001	depth 2 → branches on bit 2
THEN	Is 1011 = 1001? **No.** Bit 2 of 1011 is **0** → go **LEFT**.	



Insert 1011 — at 1001. Compare again: 1011 is not 1001. This node sits at depth 2, so it branches on **bit 2** — not bit 1 again. Bit 2 of 1011 is 0, so we go left.

STEP 9 OF 13

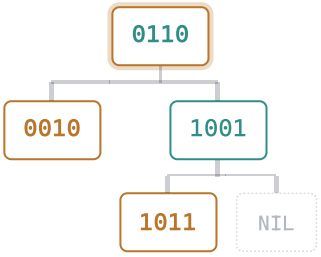
INSERTING	1 **0** 1 1	reading bit 2
THEN	1001's left slot is **empty** → place 1011 here, depth 3.	



Insert 1011 — placed. Two comparisons and two bits read, because the key had to pass through one occupied node on the way down.

STEP 10 OF 13

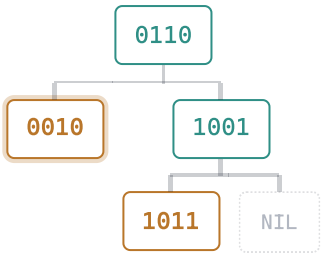
INSERTING	**0** 0 0 0	reading bit 1
AT NODE	0110	depth 1 → branches on bit 1
THEN	Is 0000 = 0110? **No.** Bit 1 of 0000 is **0** → go **LEFT**.	



Insert 0000 — at the root. Not a duplicate. Bit 1 is 0 → go left, where 0010 is already sitting, so we descend into it.

STEP 11 OF 13

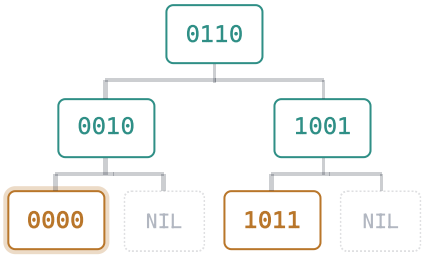
INSERTING	0 **0** 0 0	reading bit 2
AT NODE	0010	depth 2 → branches on bit 2
THEN	Is 0000 = 0010? **No.** Bit 2 of 0000 is **0** → go **LEFT**.	



Insert 0000 — at 0010. Compare: not equal. Depth 2, so branch on **bit 2**. Bit 2 of 0000 is 0 → go left. Notice the two keys agree on bit 1 *and* bit 2; it is bit 3 that finally separates them — but we never get that far, because the left slot here is already free.

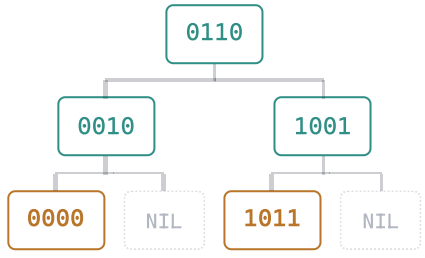
STEP 12 OF 13

INSERTING	0 **0** 0 0	reading bit 2
THEN	0010's left slot is **empty** → place 0000 here, depth 3.	



Insert 0000 — placed. A key stops at the *first* empty slot it reaches — it does not descend all the way to where its bits would eventually differ. That is exactly why a DST needs only n nodes.

STEP 13 OF 13



FINAL DST — 5 KEYS, 5 NODES, HEIGHT 3, ZERO ROTATIONS

Done. Five 4-bit keys placed using 6 comparisons and 6 bit-reads in total across the four later insertions. Every node holds a real key — compare that with the 16-node binary trie from L13·08. And no rotation happened, because no rebalancing step exists in a DST at all: the shape is dictated entirely by the bits of the keys themselves.

Every node visited needs its own key comparison

THE CRUCIAL DIFFERENCE FROM A TRIE

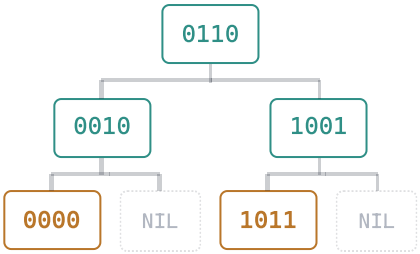
In a trie, only the *last* node reached needs its isWord bit checked — every earlier step is just an array lookup. In a DST, every node holds a real key, so at **every** level you must compare the target against the key stored there before deciding whether to continue.

TWO SEARCHES

Search 1: look up **1011**, which is stored — found after 2 comparisons and a match on the third node. Search 2: look up **1010**, which is not stored — watch it fail only after reaching a node with no further child to follow.

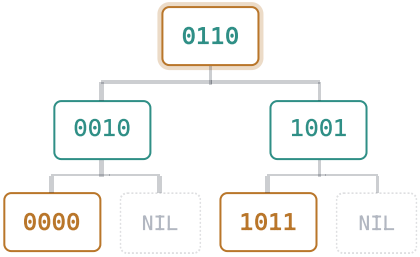
INTERACTIVE — TRACE BOTH SEARCHES

STEP 1 OF 8



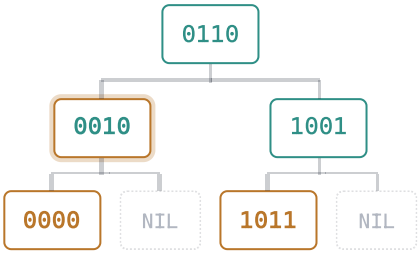
Search 1: look up 1011. Start at the root, 0110.

STEP 2 OF 8



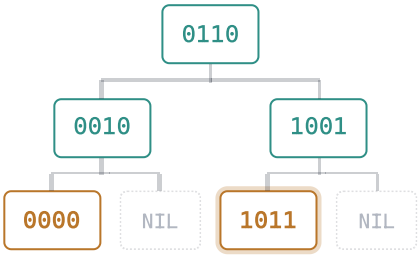
Compare: is 0110 = 1011? No. Bit 1 of 1011 is 1 → go right, to 1001.

STEP 3 OF 8



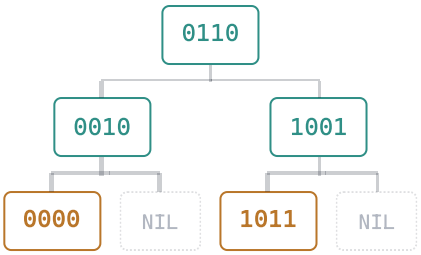
Compare: is 1001 = 1011? No. Bit 2 of 1011 is 0 → go left, to 1011.

STEP 4 OF 8



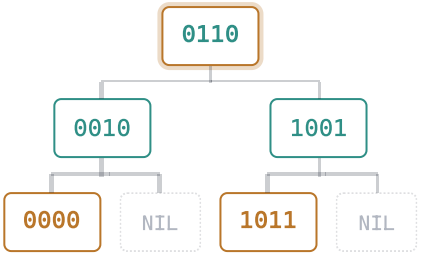
Compare: is 1011 = 1011? **Yes — found.** 3 key comparisons, one per level, matching the tree's height.

STEP 5 OF 8



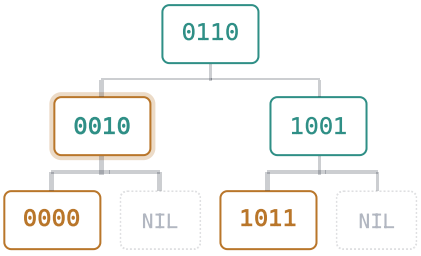
Search 2: look up 1010. Start at the root, 0110, again.

STEP 6 OF 8



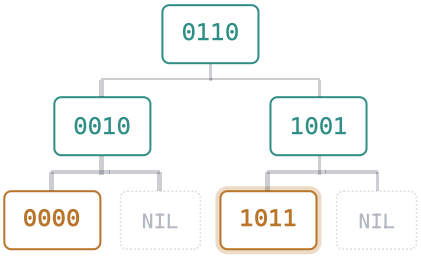
Compare: 0110 = 1010? No. Bit 1 of 1010 is 1 → go right, to 1001.

STEP 7 OF 8



Compare: 1001 = 1010? No. Bit 2 of 1010 is 0 → go left, to 1011.

STEP 8 OF 8



Compare: 1011 = 1010? No. Bit 3 of 1010 is 1 → go right of 1011 — but 1011 has **no right child**. **Not found**, after 3 comparisons.

One child or two — promotion breaks either way

THE RULE THIS ALL COMES DOWN TO

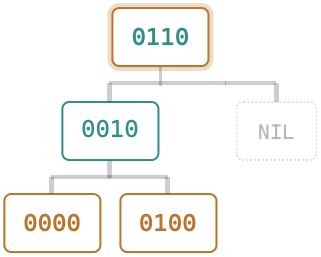
In a DST, whatever sits at depth **d** is reached because bit **d** of its key was read at that level — depth 1 reads bit 1, depth 2 reads bit 2, and so on, for every node, always. A BST splits deletion into three cases by child count (0, 1, or 2) and has a safe move for each. A DST doesn't get that luxury: **promoting any subtree to a shallower depth** can break this rule for everything beneath it — and that danger shows up whether the node you're deleting has one child or two.

WHAT THE ANIMATION COVERS

Two small examples, one after another: first, deleting a node with exactly **one** child — the obvious "just promote it" move — and where it silently loses a key. Then a **two**-child deletion, using BST's actual rule (copy the in-order successor's key up, remove the successor from below) — showing that it only survives by *luck*, not by any real guarantee.

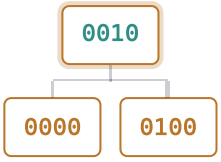
INTERACTIVE — ONE-CHILD PROMOTION, THEN TWO-CHILD PROMOTION, BOTH EXAMINED

STEP 1 OF 9



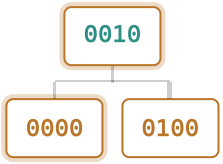
Delete 0110. It has exactly one child here — the 0010-subtree. A BST would promote 0010 straight into the root. Let's try exactly that.

STEP 2 OF 9



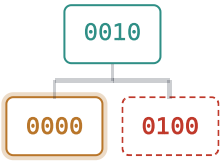
Promote 0010 into the root. Its whole subtree comes with it — 0000 and 0100 both shift from depth 3 up to depth 2. Their values didn't change. Their depth did.

STEP 3 OF 9



Search for **0100**. At the new root, 0010 (depth 1): not equal. Depth 1 → read bit 1 of the target. Bit 1 of 0100 is 0 → go left.

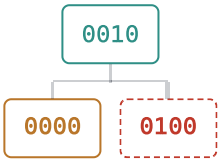
STEP 4 OF 9



DEAD END — 0000 HAS NO RIGHT CHILD

At 0000 (depth 2): not equal. Depth 2 → read bit 2 of the target. Bit 2 of 0100 is 1 → go right. But 0000 has no right child.

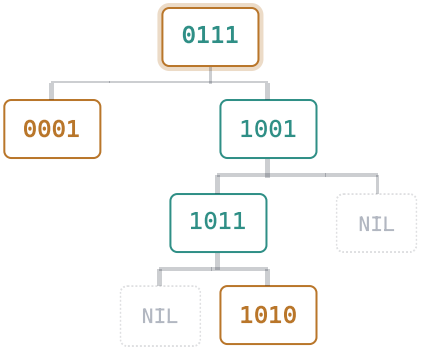
STEP 5 OF 9



NOT FOUND — 0100 IS STILL PHYSICALLY PRESENT

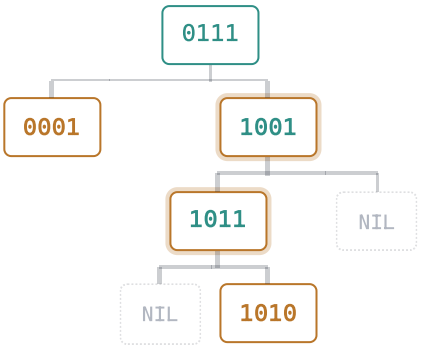
NOT FOUND — but 0100 is sitting right there, still 0010's real right child. 0010 is now depth 1, so it always reads bit 1 — and bit 1 is 0 for *both* its children, since both were originally forced under the root's left side. Bit 1 can no longer tell them apart. The right child is gone for good, silently.

STEP 6 OF 9



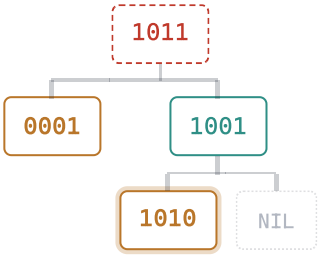
A different tree, where the deleted node genuinely has **two** children: 0001 (a leaf) and the 1001-subtree. BST's real two-child rule was never "promote a child" — it's "copy the in-order successor's key up, then remove the successor from below."

STEP 7 OF 9



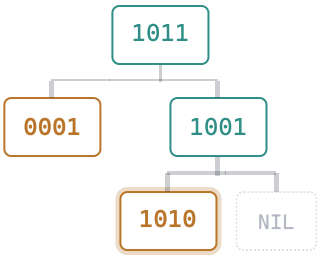
Successor = leftmost node in the right subtree. Descend into 1001, then keep going left: 1011 has no left child of its own — so 1011 is it. (It has a right child, 1010 — successors are only ever allowed a right child.)

STEP 8 OF 9



Copy 1011's key into the root. Now remove 1011 from its old spot — but it has one child, 1010, on its right. Same move as before: that child gets promoted into 1011's old slot, shifting it from depth 4 to depth 3.

STEP 9 OF 9



SAFE HERE — BUT ONLY BECAUSE 1010 HAS NO CHILDREN OF ITS OWN

This time 1010 happens to be a leaf, so this particular promotion is harmless. But BST's rule only guaranteed the successor has **at most one child** — never that the child is a leaf. If 1010 had a child of its own, the exact same bug from the one-child case would resurface right here, one level deeper. The only fix that works at every depth: never promote anything with descendants — always copy up an actual leaf instead. That's exactly what the animation below does.

Why you can't delete a DST node the way you delete a BST node

THE TRAP

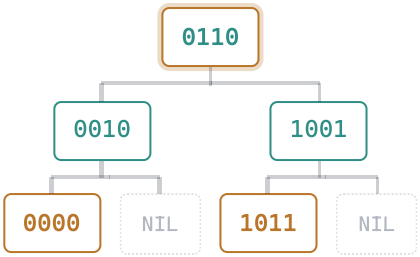
In a BST, deleting an internal node promotes its in-order predecessor or successor — any descendant will do. In a DST that's **wrong**: every node's depth encodes which bit it's supposed to branch on. Promoting an internal descendant to a shallower depth would silently misalign its own children's bit rule.

THE FIX

Delete key **0110**, the root. Instead of promoting any descendant, find a **leaf** anywhere in its subtree (a leaf has no descendants of its own to misalign), copy that leaf's key into the deleted spot, and remove the leaf from its original position — a trivial removal, since leaves have no children.

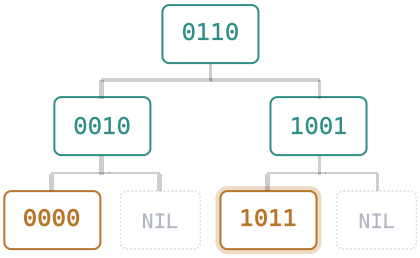
INTERACTIVE — DELETE THE ROOT, 0110

STEP 1 OF 4



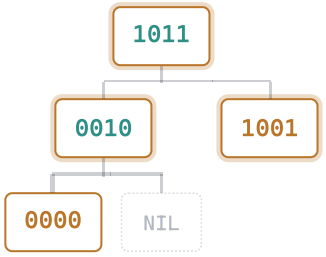
Delete 0110 (the root). It has two children, so we cannot just remove it — and we must NOT simply promote a child like a BST would, since that could put an internal node at the wrong depth for its own descendants.

STEP 2 OF 4



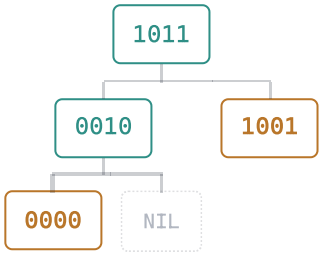
Find any **leaf** within the root's subtree — candidates are 0000 (bottom-left) and 1011 (bottom-right). Pick 1011: it has no children, so removing it later is trivial.

STEP 3 OF 4



Copy 1011's value into the root position, then delete the original leaf 1011 from its old spot. The right subtree (formerly rooted at 1001 with child 1011) now just holds 1001 alone.

STEP 4 OF 4



VALID DST — EVERY SUBTREE STILL BRANCHES ON THE CORRECT BIT FOR ITS OWN DEPTH

Done. New tree: root 1011, left subtree {0010, 0000} unchanged, right subtree {1001} alone. Check it: every node's children still correctly reflect the bit at THEIR depth — the root's own value never constrained anything, exactly as the definition allows ("root contains one pair — any pair").

Same family of ideas, two different trade-offs

OPERATION	TRIE	DIGITAL SEARCH TREE
Search / insert / delete	O(|w|) — length of the word	O(b) worst case, O(log n) expected — b = key length in bits
Comparisons needed per operation	One (at the final node only)	One per level visited

ASPECT	TRIE	DIGITAL SEARCH TREE
Branching factor	|Σ| (alphabet size)	2 (one bit at a time)
What's stored at a node	An array of pointers + an isWord bit — no key value	An actual key, plus two child pointers
Space per node	Θ(|Σ|) — can be large	O(1) — always exactly 2 pointers
Best suited to	Text, natural-language dictionaries, variable-length keys	Fixed-length bit keys — IP addresses, IDs, hash values
Typical real use	Autocomplete, spell-checkers, search-engine indices	IP routing / packet classification, firewalls

One packet in, one cable out — longest-prefix matching

A router is a box with a few cables leaving it. A packet arrives carrying a **destination address**, and the router must pick exactly one cable to send it out on. That is the entire job — and a trie is how it gets done, several hundred million times a second.

WHY PREFIXES, NOT ADDRESSES

IPv4 has **4 billion** addresses. No router can hold one rule per address, so it holds rules for **blocks** instead — and a block is written as a **prefix**: `10*` means "every address whose first two bits are 1, 0".

Sorting post, same idea. Two rules on the wall: PIN starting `30…` → Rajasthan bag; PIN starting `3020…` → Jaipur bag. A letter for `302017` matches *both*. It goes in the Jaipur bag, because `3020…` is the **more specific** rule. That is longest-prefix matching, and it is the whole problem.

TWO DIFFERENT THINGS – THIS IS THE PART THAT TRIPS PEOPLE UP

- **What goes INTO the trie:** the routing table's **prefixes** — the rules on the wall. Each prefix is a path down from the root, and the node where that path ends is **marked** with its cable.
- **What you SEARCH with:** one arriving packet's **complete destination address**. Not a prefix — a full address.
- **The answer:** the **deepest marked node** you passed on the way down. Deepest = longest = most specific.

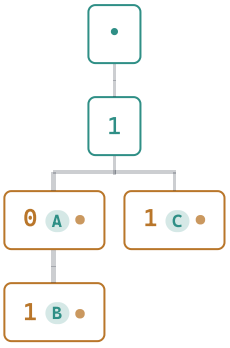
A WORKED ROUTER — 4-BIT ADDRESSES, THREE CABLES (REAL ONES ARE 32-BIT; THE IDEA IS IDENTICAL)

The routing table. Only this left column goes into the trie:

PREFIX	COVERS	SEND OUT
`10*`	`1000, 1001, 1010, 1011`	**A**
`101*`	`1010, 1011`	**B**
`11*`	`1100, 1101, 1110, 1111`	**C**

Notice `101*` sits **inside** `10*`. That is deliberate — it is how you say "send this whole block to A, *except* this smaller sub-block, which goes to B."

The trie built from it. Three marked nodes, nothing else stored:



THREE MARKED NODES — THE WHOLE ROUTING TABLE

Each node's label is the bit you read to reach it, exactly as in every trie this lecture has built. **Amber** and the "•" mean the same thing they always have — **a stored entry ends here** — and the badge names the cable it stores. The teal `"1"` node carries neither: no rule is just `1*`, so nothing is stored there and the walk simply passes through.

A PACKET FOR 1011 ARRIVES — WALK IT DOWN

READ BIT	NOW AT	MARKED?	BEST SO FAR
`1`	`"1"`	no	—
`0`	`"10"`	✦ A	A (length 2)
`1`	`"101"`	✦ B	**B** (length 3)
`1`	nothing there	—	stop

Out it goes on cable **B**. The rule is simply: **keep walking, remember the last mark you passed, stop when you fall off the tree.**

THREE RULES ANSWERING ALL SIXTEEN ADDRESSES

ADDRESS	CABLE	WHY
`1000, 1001`	**A**	matches `10*` only
`1010, 1011`	**B**	matches `10*` *and* `101*` — longer wins
`1100 … 1111`	**C**	matches `11*`
`0000 … 0111`	—	no rule matches → default route

Scale it up: roughly a million prefix rules covering 4 billion addresses, every packet resolved in at most **32** steps. That compression is why the structure exists.

THIS IS NOT THE PREFIX SEARCH FROM L13·05 — IT RUNS THE OTHER WAY

Prefix search (L13·05): you are *given a prefix* and want every key underneath it — `"st"` → stock, stop. Walk down, then collect everything below.

Longest-prefix match (here): you are *given a full key* and want the deepest stored prefix sitting *above* it on the path. Walk down, remember the last thing you passed. Same tree, opposite directions.

WHY NOT A HASH TABLE — AND WHY NOT A DST

A **hash table** needs an exact key. To use one here you would have to look up all 32 possible prefix lengths of the address separately and keep the longest hit. The trie answers it in a single walk, because the tree's *shape* already encodes "more specific = further down".

A **DST** is the wrong tool for this one: it stores each whole key at the first free node it finds, in a position that has nothing to do with what that path spells. Longest-prefix matching needs exactly the opposite — a prefix must sit at the node its own path spells out. Real routers use bit-tries and their compressed PATRICIA form.

Stop comparing whole keys — read them instead

- **A trie spells strings out along paths**, one character per edge, storing real keys only implicitly (as paths) — lookup cost O(|w|), completely independent of how many words are stored. Verified today across build, search, prefix search, and deletion on the same 8-word set.
- **Compression collapses single-child chains**, cutting a trie's node count from O(N·|Σ|) space down toward O(s) — the same underlying idea PATRICIA tries and space-efficient routing tables use.
- **A Digital Search Tree trades the alphabet-wide array for two pointers**, storing a real key at every node and branching on one bit per level — ideal exactly where tries are wasteful: fixed-length bit keys like IP addresses. Verified today across a full 5-key build, search, and the one delete case that genuinely needs care.
- Both structures are variations on one theme carried since Lecture 7: **choose what you compare against very carefully**, and the cost of every dictionary operation collapses.

LECTURE 14 — NEXT

Suffix Trees and String Processing Applications

Every suffix of a text, in one structure — pattern matching, longest repeated substring, and more, all in linear preprocessing time.

HOMEWORK — BRING TO LECTURE 14

- Build a trie for the words: cat, car, cart, dog, do. Mark which nodes are word-ends and which are pure branching points.
- On that trie, trace a prefix search for "ca" and list every word found.
- Insert the 4-bit keys 1100, 0101, 0111, 1110, 1000 (MSB first) into an empty Digital Search Tree, one at a time. Draw the resulting tree and state its height.
- Using the DST built in this lecture, trace search(0010) and search(0011); how many key comparisons does each need?
- In 3–4 sentences: why would a DST be a poor choice for storing English dictionary words, and why would a trie be a poor choice for storing 128-bit IPv6 addresses?
- Reading: Horowitz, Sahni & Anderson-Freed, *Fundamentals of Data Structures*, tries/digital search trees chapter; CLRS §12 (BSTs, for contrast).

Questions?

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

> *Next: Lecture 14 — Suffix Trees and String Processing Applications.*