Chapter 11: Tries

Algorithms

Prefix tree | Node structure | Insertion | Search and prefix queries | Autocomplete | Deletion | Time and memory | Compressed and suffix tries

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_11_Tries.html

Learning Outcomes

Structure

Explain how a trie is built from a root, nodes, edges and end-of-word marks; draw the trie of a set of words.

Operations

Implement insert, search, prefix query, autocomplete and delete in Java and trace them step by step.

Analysis

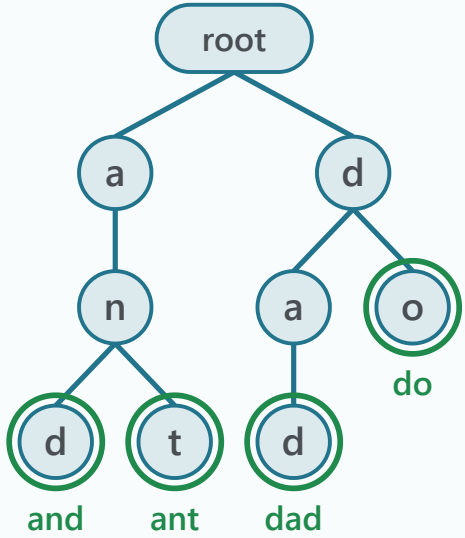
Justify the O(L) time and O(N · M) memory of the operations; compare tries with hash tables and BSTs.

Variants

Explain which problem compressed tries, suffix tries and Aho–Corasick each solve.

The Trie (Prefix Tree)

- A **tree-like** data structure that stores **strings** for fast lookup.
- **The root is empty**; each step down from the root represents one **character**.
- Words are stored as **paths from the root**; common **prefixes** are stored only once.
- Applications: **autocomplete**, **spell checking**, **IP routing**.



The name comes from re**trie**val (E. Fredkin, 1960). A double circle marks a node where a **word ends**.

Why a Trie?

We want **every word in a dictionary that starts with "an"**. How do different structures answer this?

Array / list

Compare against every word: O(n · L) for n words. Slows down as the dictionary grows.

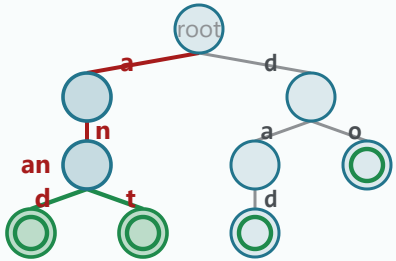
Hash table

Finds a whole word in O(L) on average, but **cannot answer prefix queries**: the hashes of "an" and "and" are unrelated.

Trie

Follow branch "a", then "n" (2 steps); **every word** below that node starts with "an".

Search time in a trie depends on the **length** (L) of the key, not on the **number** of words stored.



Basic Components

Root

An empty node; it represents no character and is where every search begins.

Nodes

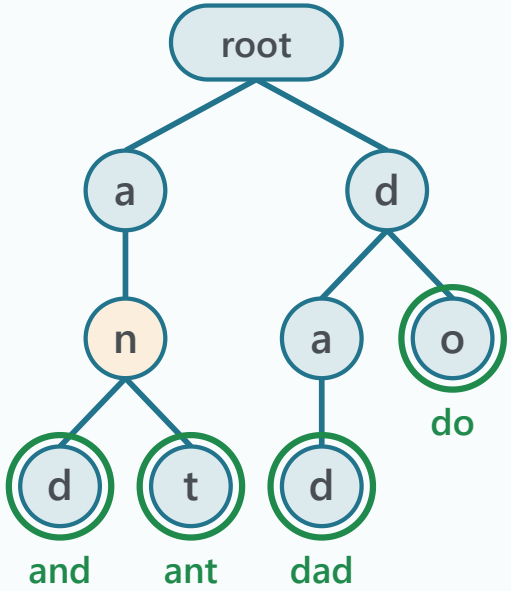
The path from the root to a node is a **prefix**; each node holds links to its children.

Edges

Links between nodes; each edge carries **one character**.

End-of-word mark

`isEnd` : a word ends at this node.

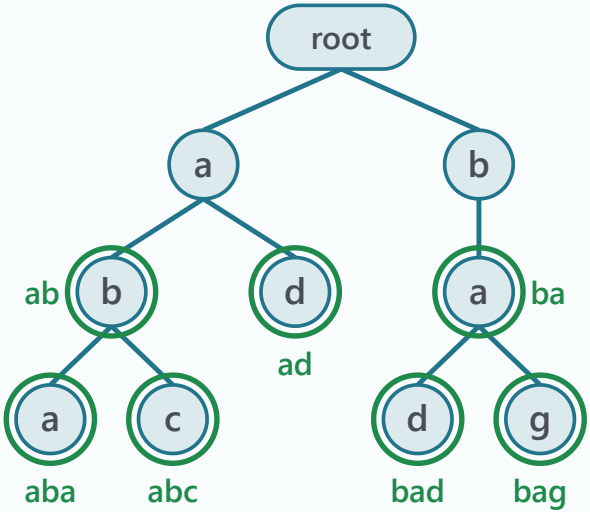


The colored node represents the prefix "an"; "an" is not a word (unmarked), but it is a prefix of "and" and "ant".

Example: A Seven-Word Trie

Words
ab · aba · abc
ad
ba · bad · bag

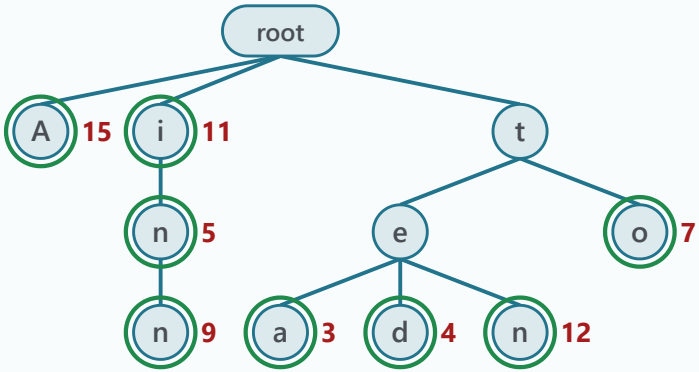
7 words, 18 characters in total → only **9 nodes** (+ root).



"ab" is both a **word** and a **prefix** of "aba" and "abc". The shape alone cannot tell us this, so the `isEnd` mark is essential.

Example: A Trie with Values

Key	Value	Key	Value
A	15	tea	3
i	11	ted	4
in	5	ten	12
inn	9	to	7



A trie can also serve as a **map**: the node stores a value instead of `isEnd`. "t" and "te" carry no value; they are only prefixes. The uppercase "A" needs a wider alphabet (e.g. a `HashMap`) than a 26-letter array.

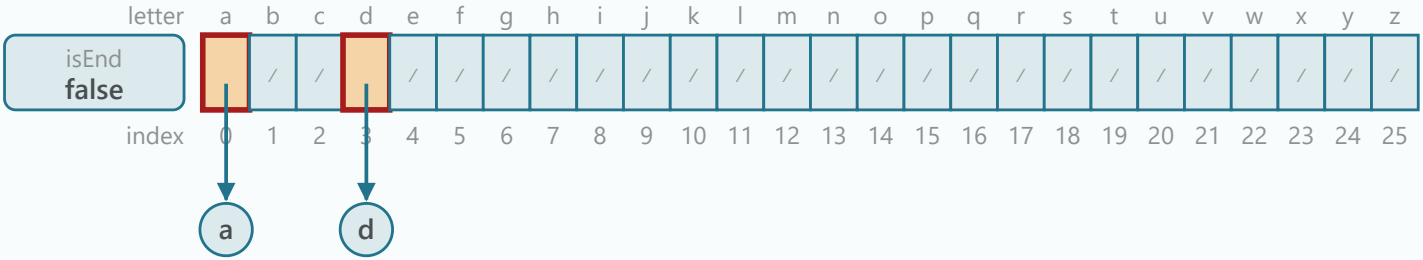
IMPLEMENTATION

The Trie Node Class

```
public class TrieNode {
  TrieNode[] children;  // 'a'..'z'
  boolean isEnd;  // word ends here?

  public TrieNode() {
    isEnd = false;
    children = new TrieNode[26];
  }
}
```

- Each node holds **26 child references**, all `null` initially.
- The character is **not stored** in the node: it is implied by **which index** of the parent the node sits at.
- `isEnd` : is the path from the root to here a word?

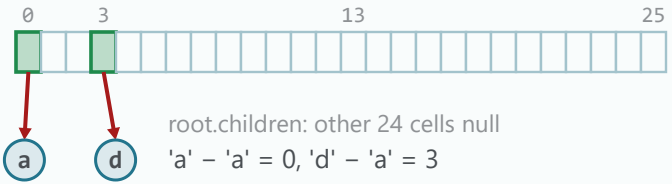


The root's `children` array after inserting and, ant, dad, do: only 2 of the 26 cells are used (/ = null).

IMPLEMENTATION

From Character to Index: `ch - 'a'`

ch	Unicode	ch - 'a'
'a'	97	0
'd'	100	3
'n'	110	13
'o'	111	14
't'	116	19
'z'	122	25



- In Java a `char` is a number; `ch - 'a'` maps a lowercase letter to 0..25.
- Accessing a child by array index is **O(1)**: `node.children[ch - 'a']`.
- For "and", `children[0]`, `children[13]`, `children[3]` are used in turn.

`'A' - 'a'` = −32, `'ç' - 'a'` = 134 → `ArrayIndexOutOfBoundsException`. For uppercase or Turkish letters, normalize the input or use a `HashMap`.

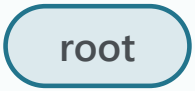
Insertion

```java
public void insert(String word) {
  TrieNode node = root;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (node.children[i] == null)
      node.children[i] = new TrieNode();
    node = node.children[i];
  }
  node.isEnd = true;
}
```

1. Start at the **root**.
2. For each character of the word, look at the matching child: `children[ch - 'a']`.
3. If the child is `null`, create a **new node**; otherwise (a shared prefix) **reuse the existing node**.
4. Move to that child.
5. When the characters run out, set `isEnd = true` on the last node.

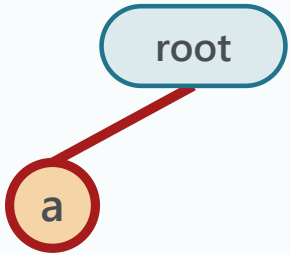
Time: **O(L)** (L = word length); at most L new nodes are created. Inserting the same word twice leaves the structure unchanged.

Insertion Step by Step: and, ant, dad, do



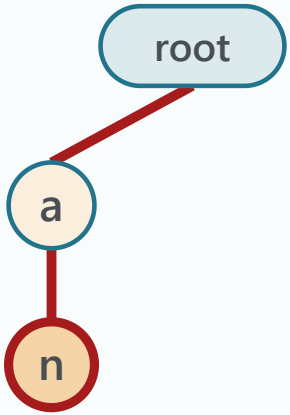
Initially the trie is just the **root**; all 26 of its children are `null` and `isEnd` = false.

Insertion Step by Step: and, ant, dad, do



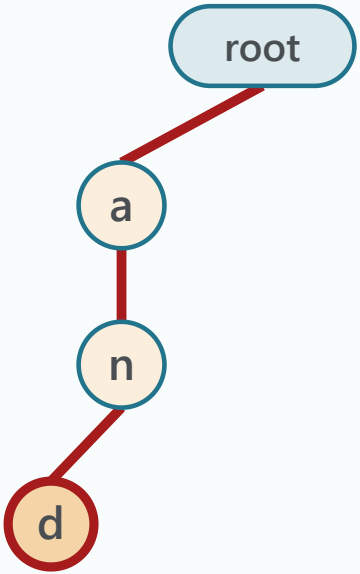
Inserting "and", character 'a': `children[0]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



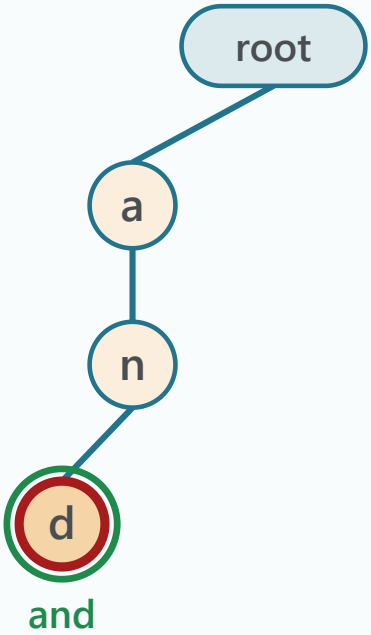
Inserting "and", character 'n': `children[13]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



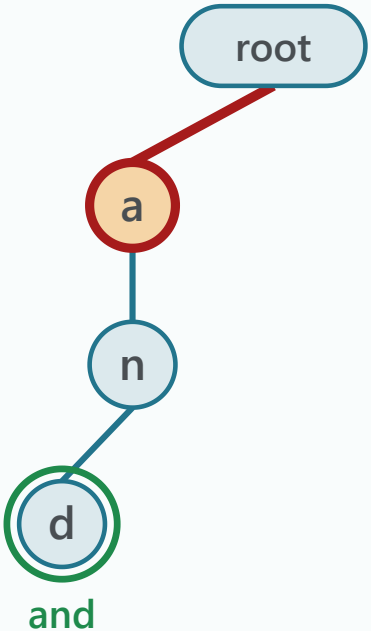
Inserting "and", character 'd': `children[3]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



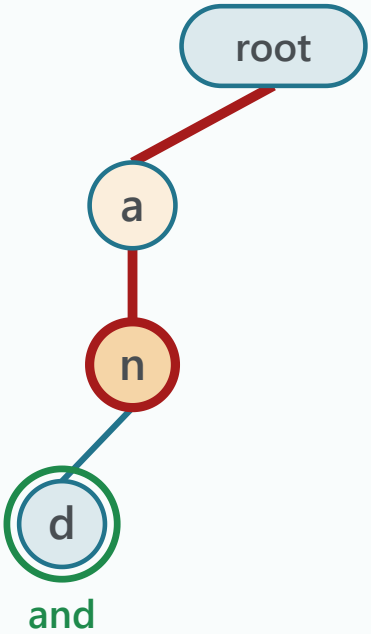
Characters exhausted: `isEnd = true` on the last node (double circle). "and" inserted. The path `children[0]` → `children[13]` → `children[3]` from the root is in place.

Insertion Step by Step: and, ant, dad, do



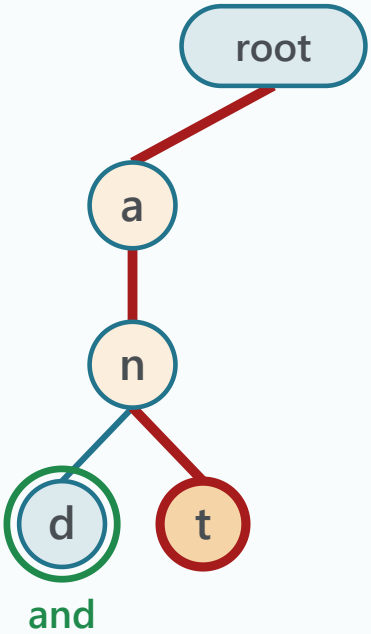
Inserting "ant", character 'a': `children[0]` is set → move to the **existing** "a" node (shared prefix).

Insertion Step by Step: and, ant, dad, do



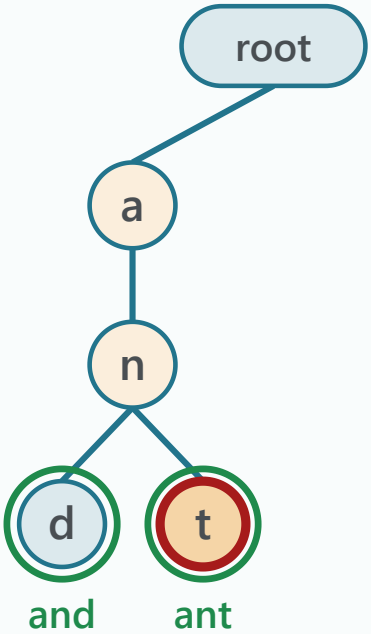
Inserting "ant", character 'n': `children[13]` is set → move to the **existing** "an" node (shared prefix).

Insertion Step by Step: and, ant, dad, do



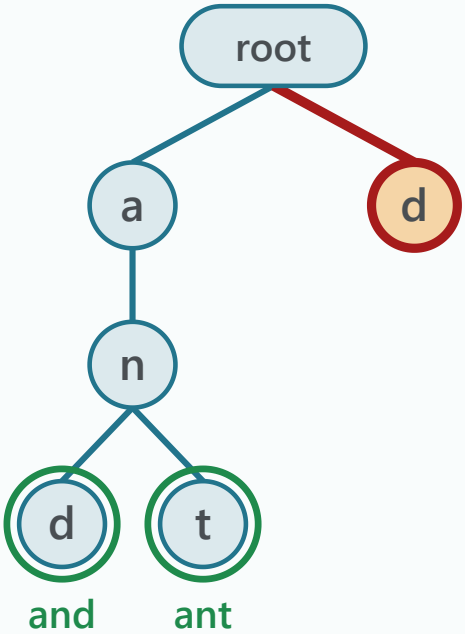
Inserting "ant", character 't': `children[19]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



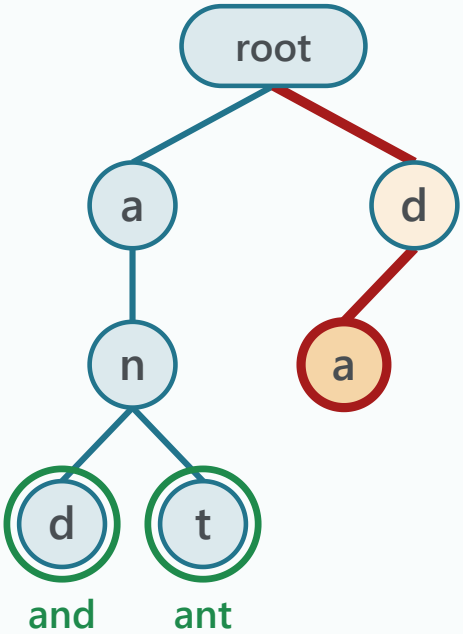
Characters exhausted: `isEnd = true` on the last node (double circle). "ant" inserted. Only 1 new node was needed: the prefix "an" is shared.

Insertion Step by Step: and, ant, dad, do



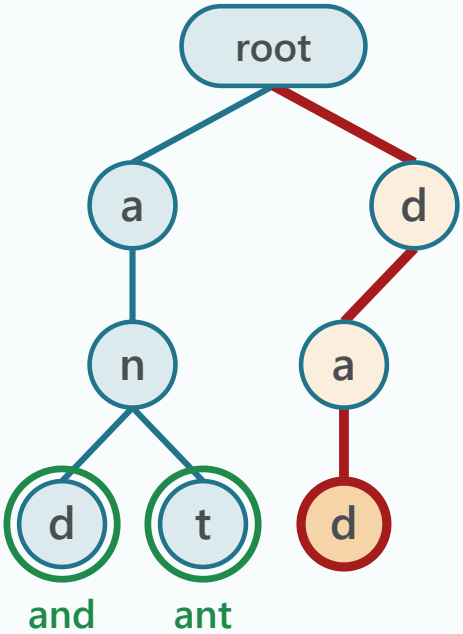
Inserting "dad", character 'd': `children[3]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



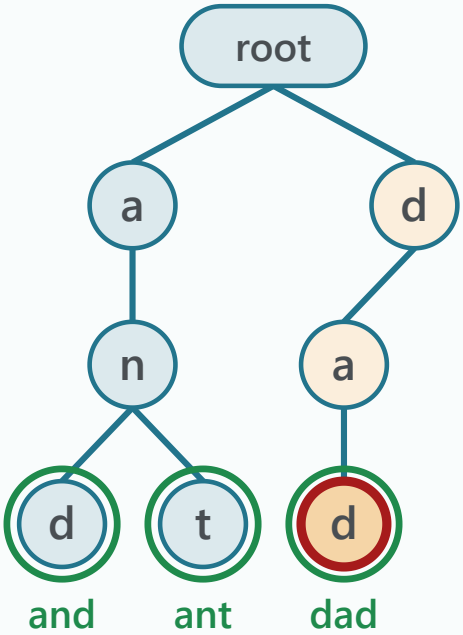
Inserting "dad", character 'a': `children[0]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



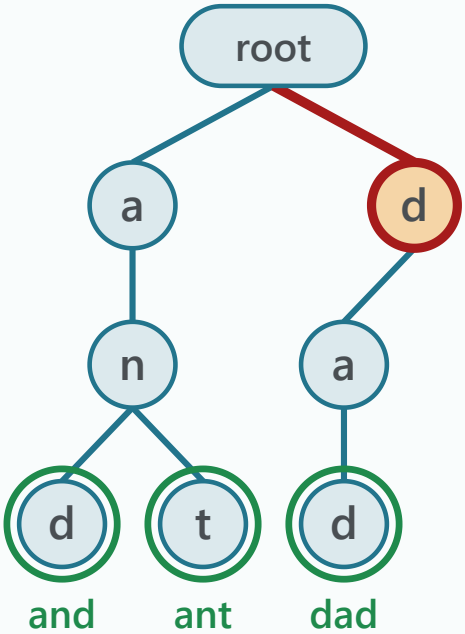
Inserting "dad", character 'd': `children[3]` null → create a **new node** and move to it.

Insertion Step by Step: and, ant, dad, do



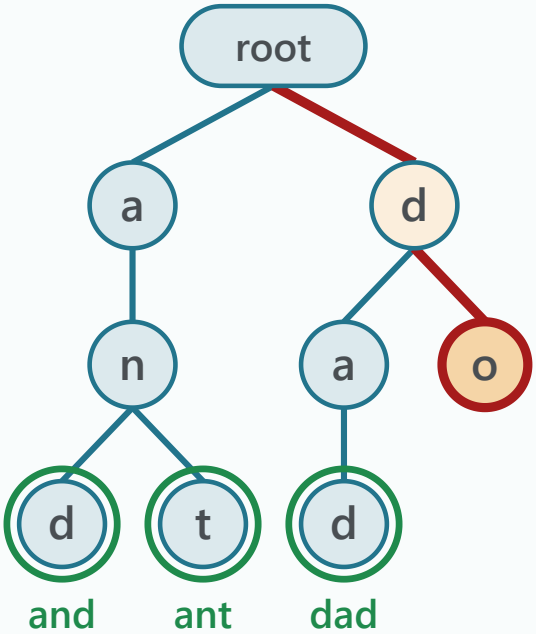
Characters exhausted: `isEnd = true` on the last node (double circle). "dad" inserted.

Insertion Step by Step: and, ant, dad, do



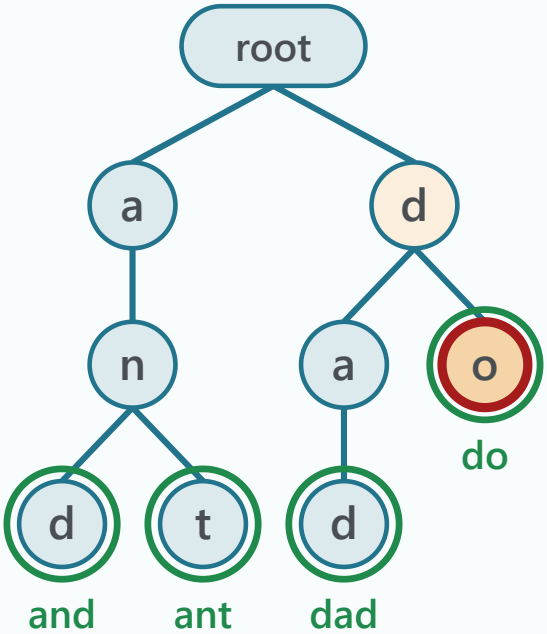
Inserting "do", character 'd': `children[3]` is set → move to the **existing** "d" node (shared prefix).

Insertion Step by Step: and, ant, dad, do



Inserting "do", character 'o': `children[14]` null → create a **new node** and move to it.

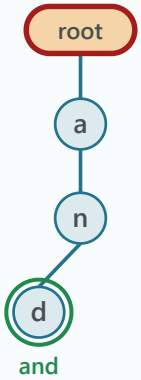
Insertion Step by Step: and, ant, dad, do



Characters exhausted: `isEnd = true` on the last node (double circle). "do" inserted. Final state: 4 words, 11 characters, **8 nodes** excluding the root.

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
▶    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



The trie holds only "and". `node` starts at the root.

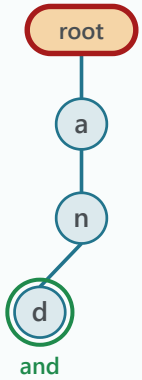
Step 1 / 12

VARIABLES

`word = "ant"` `node = root` `ch = –` `i = –`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



Character 'a' → `i = 'a' - 'a' = 0`.

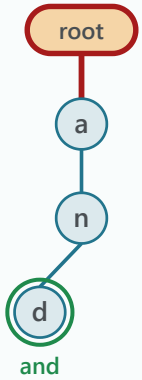
Step 2 / 12

VARIABLES

`word = "ant"` `node = root` `ch = 'a'` `i = 0`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



and

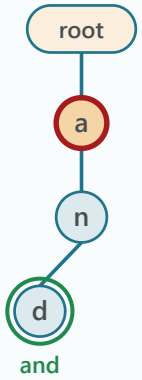
`node.children[0]` is not null → **no** new node needed.

VARIABLES

`word = "ant"` `node = root` `ch = 'a'` `i = 0`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
▶      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



`node` is now the "a" node.

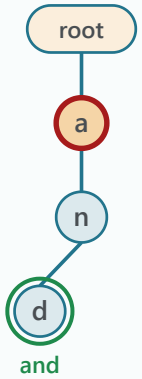
Step 4 / 12

VARIABLES

`word = "ant"` `node = "a"` `ch = 'a'` `i = 0`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



Character 'n' → `i = 'n' - 'a' = 13`.

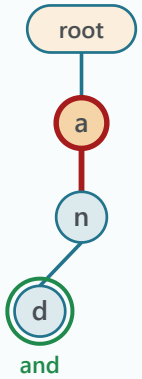
Step 5 / 12

VARIABLES

`word = "ant"` `node = "a"` `ch = 'n'` `i = 13`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



`node.children[13]` is not null → **no** new node needed.

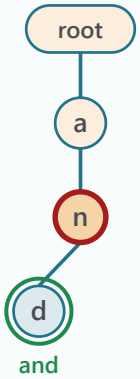
Step 6 / 12

VARIABLES

`word = "ant"` `node = "a"` `ch = 'n'` `i = 13`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
▶      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



`node` is now the "an" node.

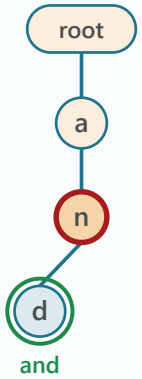
Step 7 / 12

VARIABLES

`word = "ant"` `node = "an"` `ch = 'n'` `i = 13`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



Character 't' → `i = 't' − 'a' = 19` .

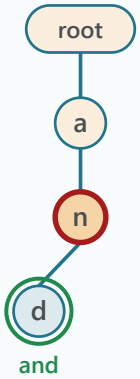
Step 8 / 12

VARIABLES

`word = "ant"` `node = "an"` `ch = 't'` `i = 19`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



`node.children[19]` == null → condition true.

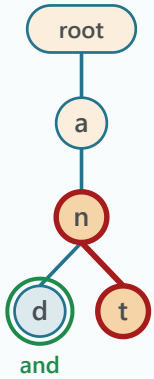
Step 9 / 12

VARIABLES

`word = "ant"` `node = "an"` `ch = 't'` `i = 19`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
▶        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



and

A new `TrieNode` is created and linked at `children[19]` (`isEnd` = false).

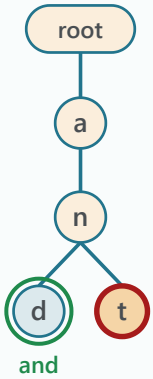
Step 10 / 12

VARIABLES

`word = "ant"` `node = "an"` `ch = 't'` `i = 19`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
▶      node = node.children[i];
8    }
9    node.isEnd = true;
10 }
```



`node` is now the "ant" node.

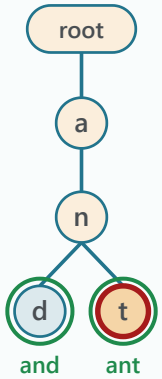
Step 11 / 12

VARIABLES

`word = "ant"` `node = "ant"` `ch = 't'` `i = 19`

insert("ant"): Step Through the Code

```
1  public void insert(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        node.children[i] = new TrieNode();
7      node = node.children[i];
8    }
▶    node.isEnd = true;
10 }
```



Loop done: `node.isEnd = true`. "ant" inserted; only 1 node was created.

Step 12 / 12

VARIABLES

`word = "ant"` `node = "ant"` `ch = 't'` `i = 19`

SEARCH

Search

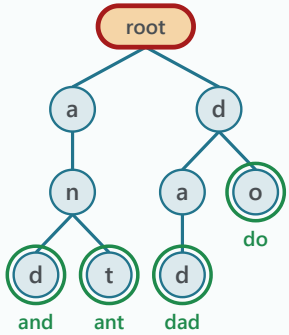
```java
public boolean search(String word) {
  TrieNode node = root;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (node.children[i] == null)
      return false;
    node = node.children[i];
  }
  return node.isEnd;
}
```

- Start at the **root**.
- For each character, follow the matching **branch** (child).
  - If the branch is missing (`null`), the word is **not** in the trie.
- At the end of the word, check `isEnd` :
  - `true` → the word is **found**;
  - `false` → not a word; it is only a **prefix** of a stored word.

Time: **O(L)**. Even with a million words in the trie, "dad" is found in at most 3 steps.

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
▶    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



Trie: and, ant, dad, do. `node` starts at the root.

Step 1 / 11

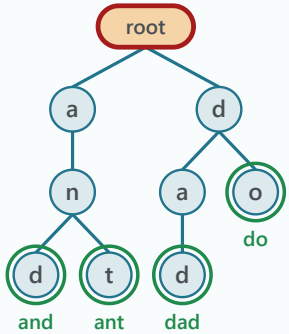
VARIABLES

`word = "dad"` `node = root` `ch = –` `i = –`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



Character 'd' → `i = 3`.

Step 2 / 11

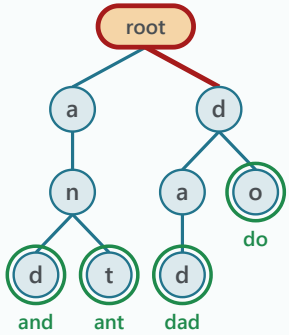
VARIABLES

`word = "dad"` `node = root` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`children[3]` is not null → branch exists, continue.

Step 3 / 11

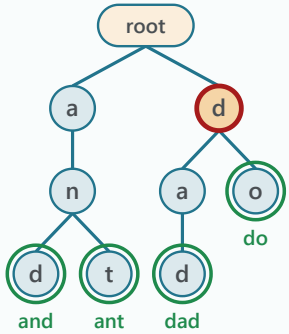
VARIABLES

`word = "dad"` `node = root` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`node` = "d".

Step 4 / 11

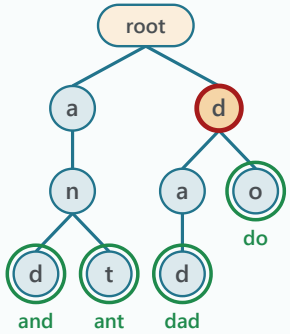
VARIABLES

`word = "dad"` `node = "d"` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



Character 'a' → `i = 0`.

Step 5 / 11

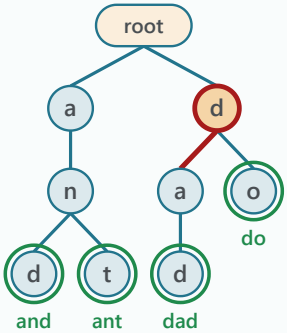
VARIABLES

`word = "dad"` `node = "d"` `ch = 'a'` `i = 0`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`children[0]` is not null → branch exists, continue.

Step 6 / 11

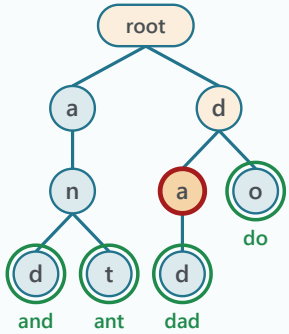
VARIABLES

`word = "dad"` `node = "d"` `ch = 'a'` `i = 0`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`node` = "da".

Step 7 / 11

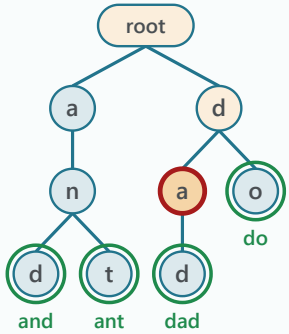
VARIABLES

`word = "dad"` `node = "da"` `ch = 'a'` `i = 0`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
▶    for (char ch : word.toCharArray()) {
▶      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



Character 'd' → `i = 3`.

Step 8 / 11

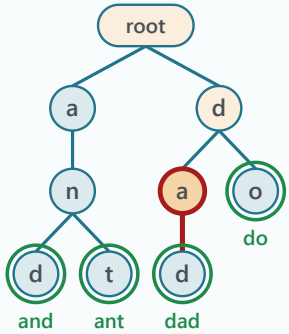
VARIABLES

`word = "dad"` `node = "da"` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`children[3]` is not null → branch exists, continue.

Step 9 / 11

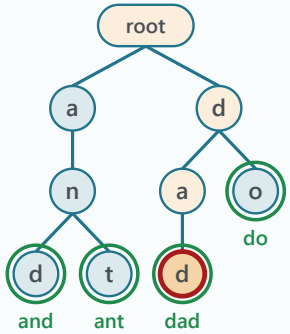
VARIABLES

`word = "dad"` `node = "da"` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return node.isEnd;
10 }
```



`node` = "dad".

Step 10 / 11

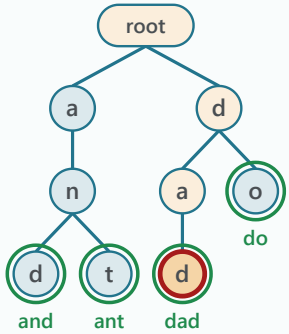
VARIABLES

`word = "dad"` `node = "dad"` `ch = 'd'` `i = 3`

OUTPUT

search("dad"): Step Through the Code

```
1  public boolean search(String word) {
2    TrieNode node = root;
3    for (char ch : word.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
▶    return node.isEnd;
10 }
```



End of word; `node.isEnd` = true → return **true** (word found).

Step 11 / 11

VARIABLES

`word = "dad"` `node = "dad"` `ch = 'd'` `i = 3`

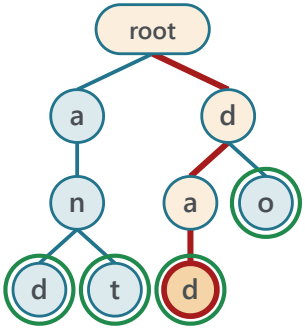
OUTPUT

```
search("dad") → true
```

Three Possible Outcomes of a Search

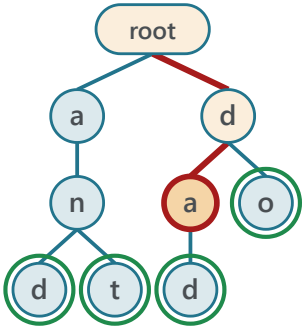
search("dad") → true

All characters found and the last node is marked.



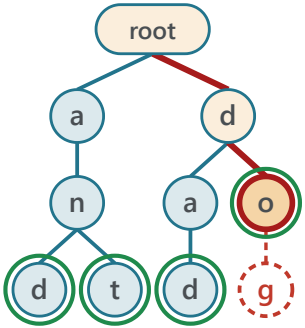
search("da") → false

The path exists, but node "da" has `isEnd` = false: "da" is only a **prefix**.



search("dog") → false

At node "do", `children[6]` ('g') is null: **no branch**, early exit.



Dashed red circle: a node that was searched for but is not in the trie.

SEARCH

Prefix Query: `startsWith`

```
public boolean startsWith(String prefix) {
  TrieNode node = root;
  for (char ch : prefix.toCharArray()) {
    int i = ch - 'a';
    if (node.children[i] == null)
      return false;
    node = node.children[i];
  }
  return true;  // search: node.isEnd
}
```

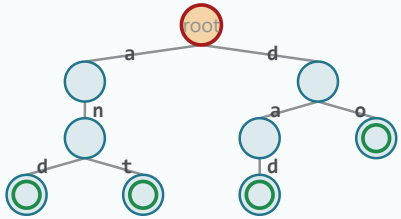
Query	search	startsWith
"dad"	true	true
"da"	false	true
"an"	false	true
"dog"	false	false

The only difference is **the last line**: a prefix query does not check the end-of-word mark; the path existing is enough. Trie: and, ant, dad, do. Time: O(P), P = prefix length.

SEARCH · STEP THROUGH THE CODE

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
▶    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



The trie holds and, ant, dad, do (ringed node = word end).
First startsWith("da"): node is set to the root.

Step 1 / 8

VARIABLES

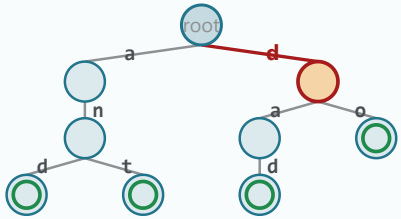
prefix = "da" ch = –

i = – node = root

OUTPUT

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
▶      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



ch = 'd', i = 3: children[3] is set → node moves down to that child ("d").

Step 2 / 8

VARIABLES

`prefix = "da"`

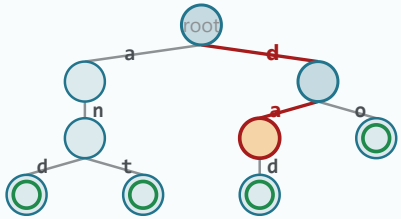
`ch = 'd'` `i = 3`

`node = "d"`

OUTPUT

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
▶      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



ch = 'a', i = 0: children[0] is set → node moves down to that child ("da").

Step 3 / 8

VARIABLES

prefix = "da"

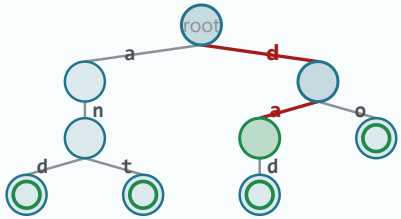
ch = 'a' i = 0

node = "da"

OUTPUT

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
▶    return true;  // search: node.isEnd
10 }
```



The prefix is consumed and the path never broke → **true**. Node "da" is unmarked, so `search("da")` would return false; a prefix query ignores the end mark.

Step 4 / 8

VARIABLES

`prefix = "da"`

`ch = 'a'` `i = 0`

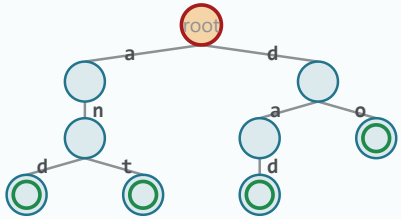
`node = "da"`

OUTPUT

```
startsWith("da") = true
```

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
▶    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
4      int i = ch - 'a';
5      if (node.children[i] == null)
6        return false;
7      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



Second call startsWith("dog"): node is reset to the root.

Step 5 / 8

VARIABLES

`prefix = "dog"`

`ch = –` `i = –`

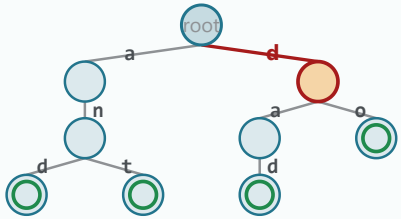
`node = root`

OUTPUT

```
startsWith("da") = true
```

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
▶      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



ch = 'd', i = 3: children[3] is set → node moves down to that child ("d").

Step 6 / 8

VARIABLES

`prefix = "dog"`

`ch = 'd'` `i = 3`

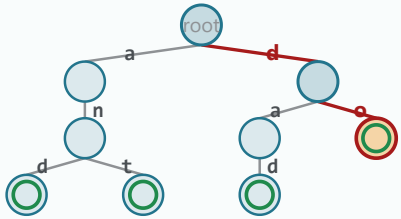
`node = "d"`

OUTPUT

```
startsWith("da") = true
```

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
▶      int i = ch - 'a';
▶      if (node.children[i] == null)
6        return false;
▶      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



ch = 'o', i = 14: children[14] is set → node moves down to that child ("do").

Step 7 / 8

VARIABLES

`prefix = "dog"`

`ch = 'o'` `i = 14`

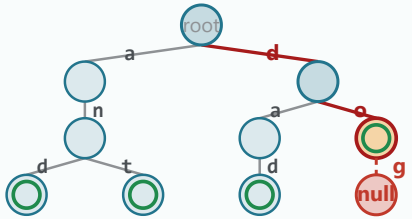
`node = "do"`

OUTPUT

```
startsWith("da") = true
```

startsWith("da") and startsWith("dog")

```
1  public boolean startsWith(String prefix)
   {
2    TrieNode node = root;
3    for (char ch : prefix.toCharArray()) {
▶      int i = ch - 'a';
▶      if (node.children[i] == null)
▶        return false;
7      node = node.children[i];
8    }
9    return true;  // search: node.isEnd
10 }
```



ch = 'g', i = 6: children[6] of node "do" is null → **false**. "do" is a word, but no word starts with "dog".

Step 8 / 8

VARIABLES

`prefix = "dog"`

`ch = 'g'` `i = 6`

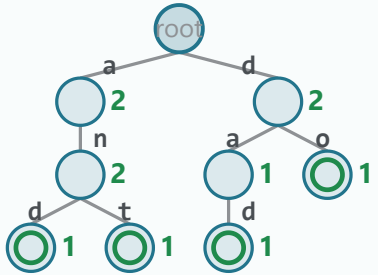
`node = "do"`

OUTPUT

```
startsWith("da") = true
startsWith("dog") = false
```

Prefix Counter: `countPrefix` in O(P)

```
void insert(String word) { if (search(word)) return;
  TrieNode d = root;
  d.count++;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (d.children[i] == null) d.children[i] = new TrieNode();
    d = d.children[i];
    d.count++;          // one more word passed
  }
  d.isEnd = true;
}
int countPrefix(String prefix) {
  TrieNode d = root;
  for (char ch : prefix.toCharArray()) {
    d = d.children[ch - 'a'];
    if (d == null) return 0;
  }
  return d.count;
}
```



Green numbers: count (4 at the root).

Query	result
countPrefix("an")	2
countPrefix("da")	1
countPrefix("x")	0

Cost: one `int count` per node; without it the query walks the subtree, O(P + S · M).

SEARCH

Wildcard Search: `"d.d"`

```
boolean wildcard(TrieNode d, String s, int j) {
  if (d == null) return false;
  if (j == s.length()) return d.isEnd;
  char ch = s.charAt(j);
  if (ch != '.')
    return wildcard(d.children[ch - 'a'], s, j + 1);
  for (int i = 0; i < 26; i++)
    if (wildcard(d.children[i], s, j + 1)) return true;
  return false;
}
```

Pattern	result	match
"d.d"	true	dad
".n."	true	and
".o"	true	do
"b.."	false	no b branch
"...."	false	no 4-letter word

- Trie: and, ant, dad, do. Call `wildcard(root, pattern, 0)`.
- A normal letter narrows the search to one path; '.' tries every non-null branch (backtracking). The first match propagates `true` upward.
- Worst case 26^k branches for k wildcards, but empty branches return at once: the real cost is bounded by the paths that exist in the trie. A hash table can answer this only by scanning every word.

SEARCH

Spell Checking: Edit Distance on a Trie

- Question: which dictionary words are within edit distance (insert / delete / replace) **at most 1** of "dod"?
- Instead of computing the distance to each word separately, traverse the trie depth-first; each node derives **one DP row** from its parent's row.
- A common prefix's row is computed once: the "d" row serves both dad and do.
- **Pruning:** if the row's minimum exceeds k, no word in the subtree can be within k; the minimum of row "an" is 2 > 1, so and and ant are never visited.

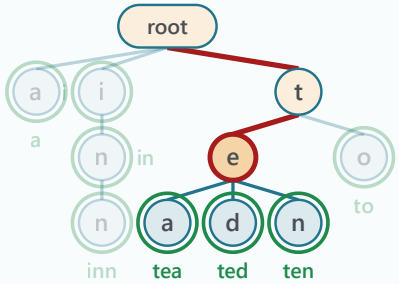
node	"" d o d	last	action
a	1 1 2 3	—	continue
an	2 2 2 3	—	prune (min 2)
d	1 0 1 2	—	continue
da	2 1 1 2	—	continue
dad	3 2 2 1	1	suggest
do	2 1 0 1	1	suggest

The last cell of a row is the distance between the node's word and "dod".

Suggestions: **dad** (o → a) and **do** (delete the final d). O(|query|) work per row; thanks to pruning, most of the dictionary is never visited. The same traversal works with a weighted distance that makes adjacent keyboard letters cheap; as k grows, pruning weakens and the number of visited nodes rises quickly.

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
▶    List<String> result = new ArrayList<>();
▶    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



First `findNode("te")` walks t → e from the root to node "te" (like a prefix query).

Step 1 / 16

VARIABLES

`prefix = "te"`

`path = –` `i = –`

CALL STACK

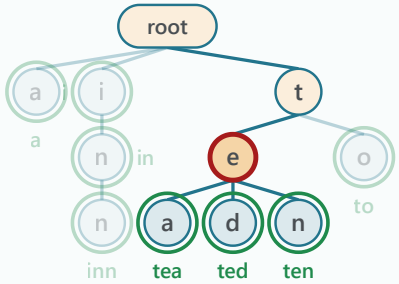
`complete("te")`

OUTPUT

```
result = []
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
▶      collect(d, new StringBuilder(prefix),
result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



Starting at node "te", the subtree is traversed **depth-first**.

Step 2 / 16

VARIABLES

`prefix = "te"`

`path = "te"` `i = –`

CALL STACK

`collect("te")`

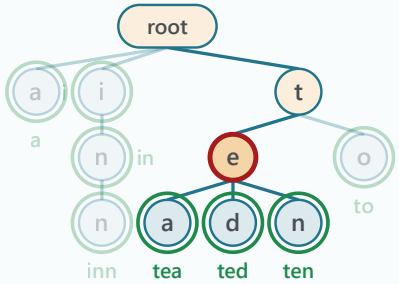
`complete("te")`

OUTPUT

```
result = []
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
▶    if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



"te" is not a word (`isEnd` = false) → not added to the list.

Step 3 / 16

VARIABLES

`prefix = "te"`

`path = "te"` `i = –`

CALL STACK

`collect("te")`

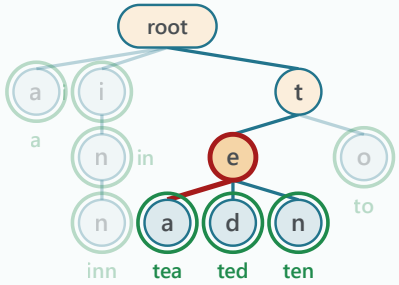
`complete("te")`

OUTPUT

```
result = []
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.children[i] != null) {
▶        path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



`i = 0` : `children[0]` ('a') is set → `path` = "tea".

Step 4 / 16

VARIABLES

`prefix = "te"`

`path = "tea"`

`i = 0`

CALL STACK

`collect("te")`

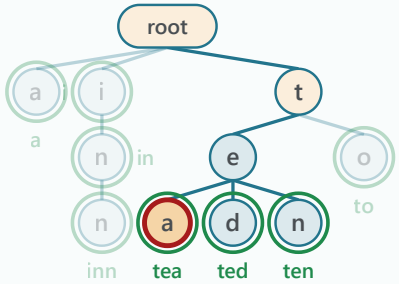
`complete("te")`

OUTPUT

```
result = []
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
▶        collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



Recursive call: `collect("tea")`.

Step 5 / 16

VARIABLES

- `prefix = "te"`
- `path = "tea"`
- `i = –`

CALL STACK

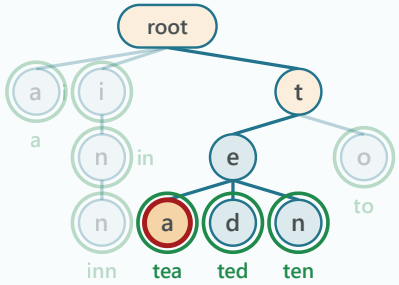
- `collect("tea")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = []
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
▶    if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



"tea" is marked → added to `result`. No children; the loop passes over 26 empty indices.

Step 6 / 16

VARIABLES

- `prefix = "te"`
- `path = "tea"`
- `i = –`

CALL STACK

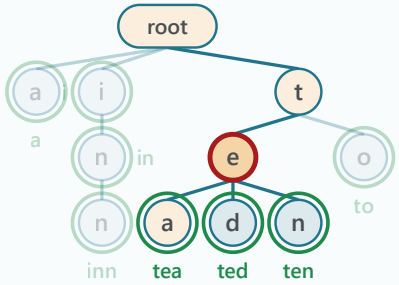
- `collect("tea")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
▶        path.setLength(path.length() - 1);
16     }
17 }
```



The call returns; drop the last letter (`path` = "te") → backtracking.

Step 7 / 16

VARIABLES

`prefix = "te"`

`path = "te"` `i = 0`

CALL STACK

`collect("te")`

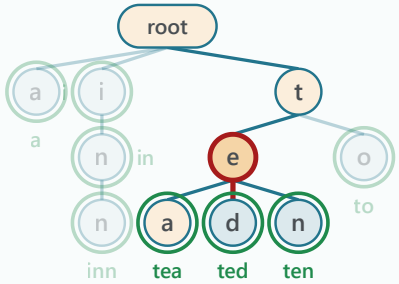
`complete("te")`

OUTPUT

```
result = [tea]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.children[i] != null) {
▶        path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



`i = 3` : `children[3]` ('d') is set → `path` = "ted".

Step 8 / 16

VARIABLES

- `prefix = "te"`
- `path = "ted"`
- `i = 3`

CALL STACK

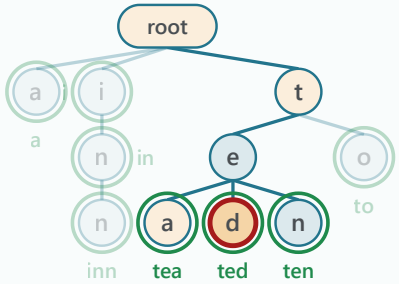
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
▶        collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



Recursive call: `collect("ted")`.

Step 9 / 16

VARIABLES

- `prefix = "te"`
- `path = "ted"`
- `i = –`

CALL STACK

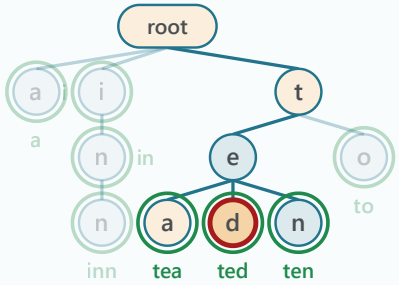
- `collect("ted")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
▶    if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



"ted" is marked → added to `result`. No children; the loop passes over 26 empty indices.

Step 10 / 16

VARIABLES

- `prefix = "te"`
- `path = "ted"`
- `i = –`

CALL STACK

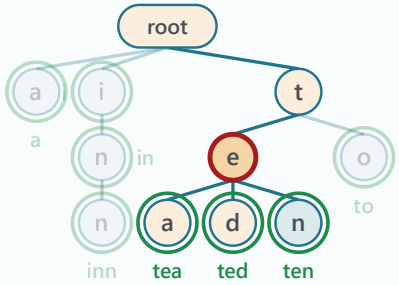
- `collect("ted")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea, ted]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
▶        path.setLength(path.length() - 1);
16     }
17 }
```



The call returns; drop the last letter (`path` = "te") → backtracking.

Step 11 / 16

VARIABLES

`prefix = "te"`

`path = "te"` `i = 3`

CALL STACK

`collect("te")`

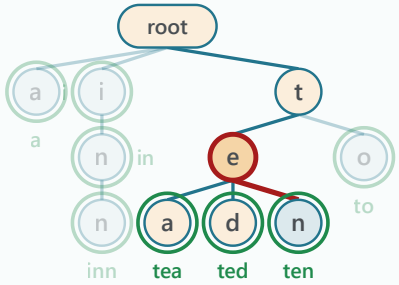
`complete("te")`

OUTPUT

```
result = [tea, ted]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.children[i] != null) {
▶        path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



`i = 13` : `children[13]` ('n') is set → `path` = "ten".

Step 12 / 16

VARIABLES

`prefix = "te"`

`path = "ten"`

`i = 13`

CALL STACK

`collect("te")`

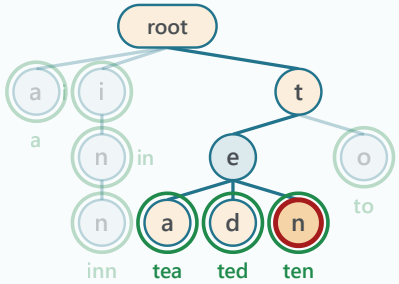
`complete("te")`

OUTPUT

```
result = [tea, ted]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9               List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
▶        collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



Recursive call: `collect("ten")`.

Step 13 / 16

VARIABLES

- `prefix = "te"`
- `path = "ten"`
- `i = –`

CALL STACK

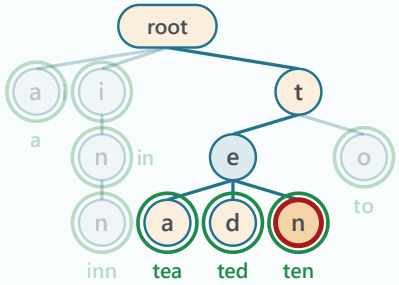
- `collect("ten")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea, ted]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
▶    if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



"ten" is marked → added to `result`. No children; the loop passes over 26 empty indices.

Step 14 / 16

VARIABLES

- `prefix = "te"`
- `path = "ten"`
- `i = –`

CALL STACK

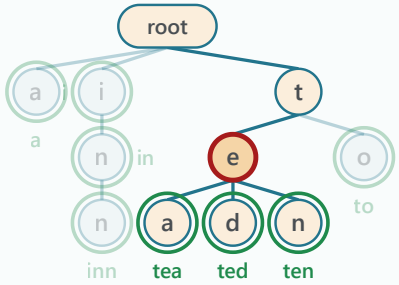
- `collect("ten")`
- `collect("te")`
- `complete("te")`

OUTPUT

```
result = [tea, ted, ten]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
6    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
▶        path.setLength(path.length() - 1);
16     }
17 }
```



The call returns; drop the last letter (`path` = "te") → backtracking.

Step 15 / 16

VARIABLES

`prefix = "te"`

`path = "te"`

`i = 13`

CALL STACK

`collect("te")`

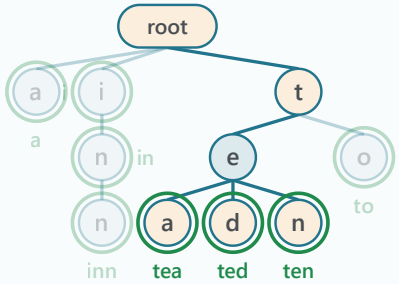
`complete("te")`

OUTPUT

```
result = [tea, ted, ten]
```

Autocomplete: complete("te")

```
1  List<String> complete(String prefix) {
2    List<String> result = new ArrayList<>();
3    TrieNode d = findNode(prefix);
4    if (d != null)
5      collect(d, new StringBuilder(prefix),
   result);
▶    return result;
7  }
8  void collect(TrieNode d, StringBuilder path,
9              List<String> result) {
10   if (d.isEnd) result.add(path.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.children[i] != null) {
13       path.append((char) ('a' + i));
14       collect(d.children[i], path, result);
15       path.setLength(path.length() - 1);
16     }
17 }
```



Loop done; `complete("te")` → [tea, ted, ten]. Thanks to index order (a < d < n), the results come out in **alphabetical** order.

Step 16 / 16

VARIABLES

`prefix = "te"`

`path = –` `i = –`

CALL STACK

`complete("te")`

OUTPUT

```
result = [tea, ted, ten]
```

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
▶    StringBuilder sb = new StringBuilder();
▶    TrieNode d = root;
4    while (!d.isEnd) {
5      int only = -1, count = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.children[i] != null) { count++; only =
   i; }
8      if (count != 1) break;  // branch or leaf
9      sb.append((char) ('a' + only));
10     d = d.children[only];
11   }
12   return sb.toString();
13 }
```



Trie: kar, kart, karton. Walk down from the root through nodes that have **one child** and are **unmarked**.

Step 1 / 6

VARIABLES

`sb = ""` `d = root` `count = –` `only = –`

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
2    StringBuilder sb = new StringBuilder();
3    TrieNode d = root;
▶    while (!d.isEnd) {
▶      int only = -1, count = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.children[i] != null) { count++; only = i; }
▶      if (count != 1) break;  // branch or leaf
9      sb.append((char) ('a' + only));
10     d = d.children[only];
11   }
12   return sb.toString();
13 }
```



The root is unmarked; count its children: only 'k' is set → count = 1, only = 10.

Step 2 / 6

VARIABLES

`sb = ""` `d = root` `count = 1` `only = 10`

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
2    StringBuilder sb = new StringBuilder();
3    TrieNode d = root;
▶    while (!d.isEnd) {
▶      int only = -1, count = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.children[i] != null) { count++; only =
   i; }
▶      if (count != 1) break;  // branch or leaf
▶      sb.append((char) ('a' + only));
▶      d = d.children[only];
11   }
12   return sb.toString();
13 }
```



Append 'k', d = "k". This node also has a single child ('a', only = 0).

Step 3 / 6

VARIABLES

`sb = "k"` `d = "k"` `count = 1` `only = 0`

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
2    StringBuilder sb = new StringBuilder();
3    TrieNode d = root;
▶    while (!d.isEnd) {
▶      int only = -1, count = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.children[i] != null) { count++; only =
   i; }
▶      if (count != 1) break;  // branch or leaf
▶      sb.append((char) ('a' + only));
▶      d = d.children[only];
11   }
12   return sb.toString();
13 }
```



Append 'a', d = "ka"; its single child is 'r' (only = 17).

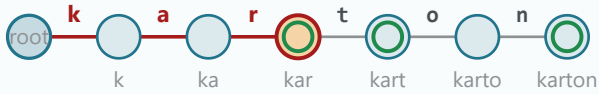
Step 4 / 6

VARIABLES

`sb = "ka"` `d = "ka"` `count = 1` `only = 17`

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
2    StringBuilder sb = new StringBuilder();
3    TrieNode d = root;
▶    while (!d.isEnd) {
5      int only = -1, count = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.children[i] != null) { count++; only =
   i; }
8      if (count != 1) break;  // branch or leaf
▶      sb.append((char) ('a' + only));
▶      d = d.children[only];
11   }
12   return sb.toString();
13 }
```



Append 'r', d = "kar". "kar" is a **word** → the loop ends: even with a single child, the word "kar" does not share a longer prefix.

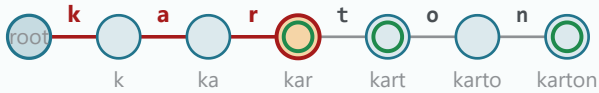
Step 5 / 6

VARIABLES

`sb = "kar"` `d = "kar"` `count = 1` `only = 17`

longestCommonPrefix(): Code Trace

```
1  String longestCommonPrefix() {
2    StringBuilder sb = new StringBuilder();
3    TrieNode d = root;
4    while (!d.isEnd) {
5      int only = -1, count = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.children[i] != null) { count++; only =
   i; }
8      if (count != 1) break;  // branch or leaf
9      sb.append((char) ('a' + only));
10     d = d.children[only];
11   }
▶    return sb.toString();
13 }
```



Returns "kar". With only kart and karton in the trie, the walk would continue to "kart" and stop at its end mark: "kart".

Step 6 / 6

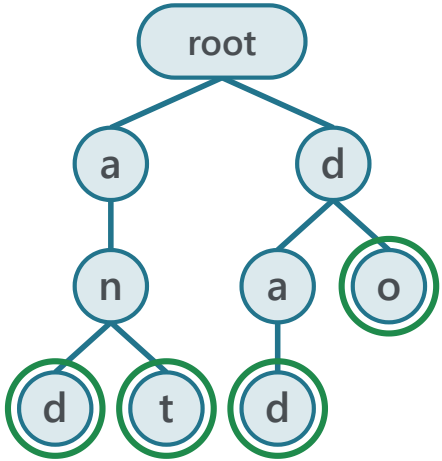
VARIABLES

`sb = "kar"`

SEARCH

Try It: Trie Operations

Word dot insert search prefix / complete delete sample reset



Sample trie: and, ant, dad, do (9 nodes, including the root). Type a word and pick an operation.

Lowercase a–z only (at most 10 letters, 14 words). Double circle: word end · orange: last visited node · dashed red: missing branch. Enter = insert.

Deletion: Three Cases

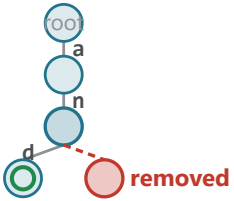
1. The word is a prefix of another word

The last node has children → just set `isEnd = false`; no node is removed.



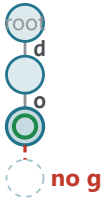
2. The word has a tail of its own

Working back from the end, remove nodes that are **childless and unmarked**; stop at the shared prefix.



3. The word is not in the trie

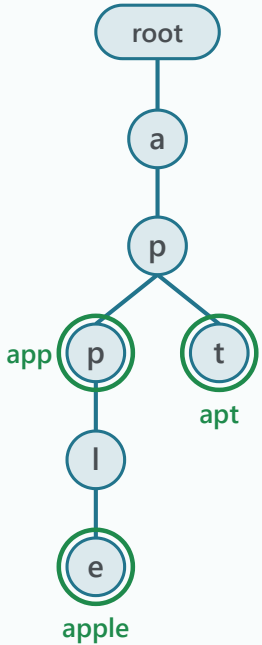
A branch is missing or the last node is unmarked → nothing changes.



Question: If the trie holds "apple" and "app", how does the structure change when "app" is deleted?

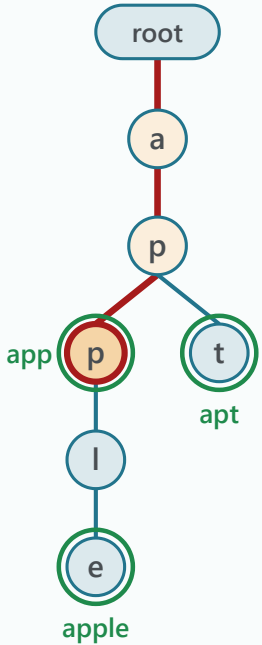
▶ **Show answer**

Deletion Step by Step: app, apple, apt



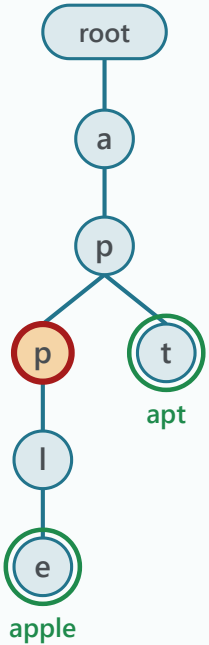
Start: app, apple, apt (6 nodes excluding the root). Both scenarios below start from this trie.

Deletion Step by Step: app, apple, apt



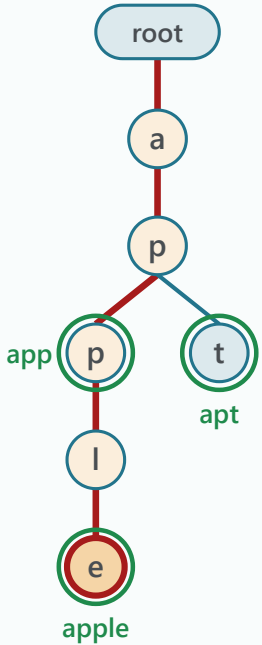
delete("app"): follow a → p → p; node "app" is marked, so the word exists.

Deletion Step by Step: app, apple, apt



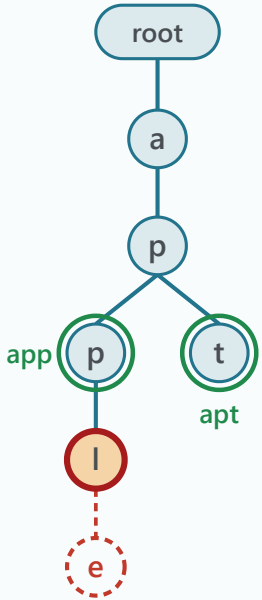
`isEnd = false`. The node has a child ('l'), so it is **not removed** → case 1. "apple" is still found.

Deletion Step by Step: app, apple, apt



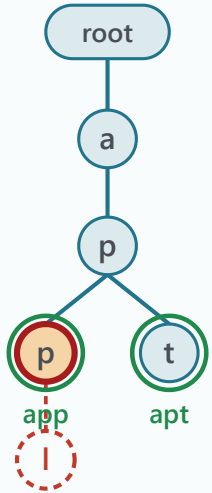
delete("apple") (again from the original trie): follow a-p-p-l-e; "apple" is found.

Deletion Step by Step: app, apple, apt



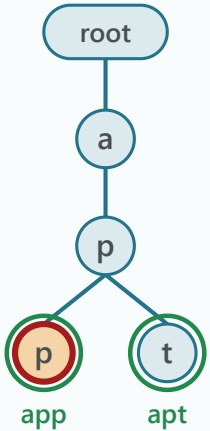
Node 'e' loses its mark; it has no children → its parent removes it (`children[4] = null`).

Deletion Step by Step: app, apple, apt



Node 'l' is now childless and unmarked → it is removed too.

Deletion Step by Step: app, apple, apt



Node "app" is **marked** (another word) → deletion stops here. Result: app, apt; 4 nodes excluding the root (case 2).

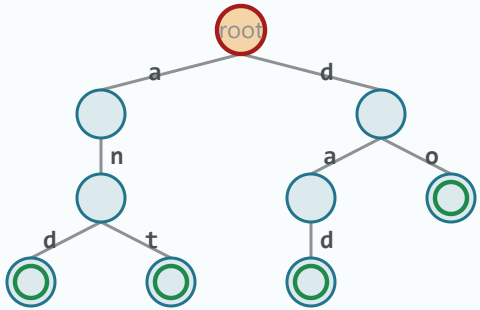
Deletion: Recursive Code

```
// true: the parent may remove d
boolean delete(TrieNode d, String k, int j) {
  if (j == k.length()) {         // end of word
    if (!d.isEnd) return false;
    d.isEnd = false;
    return isEmpty(d);           // childless?
  }
  int i = k.charAt(j) - 'a';
  TrieNode c = d.children[i];
  if (c == null) return false;   // no such word
  if (delete(c, k, j + 1)) {
    d.children[i] = null;        // remove the child
    return !d.isEnd && isEmpty(d);
  }
  return false;
}
```

- Call: `delete(root, word, 0)`; the root's return value is ignored.
- First **descend** to the end of the word; removal decisions are made **on the way back** (bottom-up).
- A node is removed only if it has **no children** and is **unmarked**.
- `isEmpty(d)`: are all 26 children null? (O(M)).
- Time: O(L · M) with an array; O(L) if a child count is kept.
- In the counting version, if the word exists (check with `search` first), do `d.count--` at each node on the way down.

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieNode c = d.children[i];
▶    if (c == null) return false;
▶    if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



delete(root, "dad", 0): the first character is 'd' → i = 3, c = the root's 'd' child. Removal decisions come on the way back; first we descend.

Step 1 / 10

VARIABLES

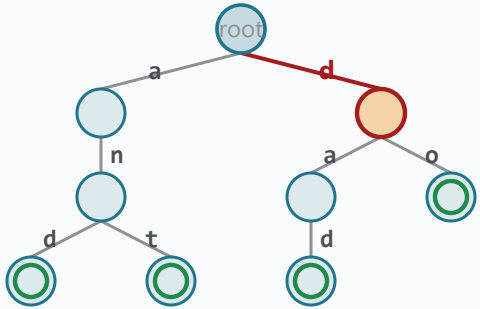
`d = root` `j = 0` `i = 3` `c = "d"`

CALL STACK

`delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieNode c = d.children[i];
▶    if (c == null) return false;
▶    if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



delete(d, "dad", 1): 'a' → i = 0, c = node "da". The recursive call goes one level deeper.

Step 2 / 10

VARIABLES

`d = d` `j = 1` `i = 0` `c = "da"`

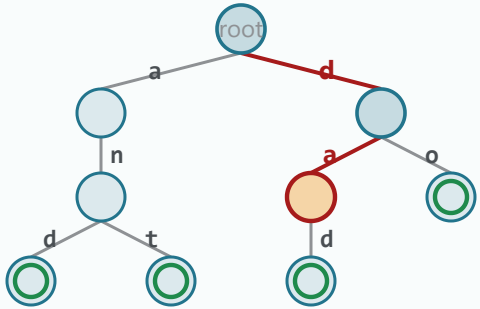
CALL STACK

`delete(d, 1)`

`delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieNode c = d.children[i];
▶    if (c == null) return false;
▶    if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



delete(da, "dad", 2): 'd' → i = 3, c = node "dad".

Step 3 / 10

VARIABLES

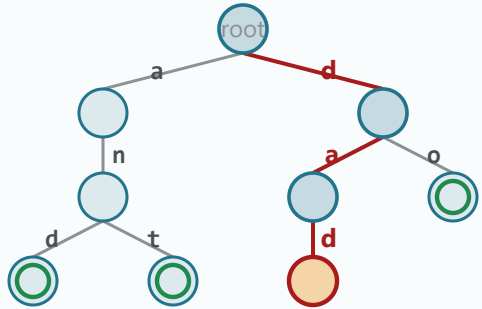
d = da j = 2 i = 3 c = "dad"

CALL STACK

- delete(da, 2)
- delete(d, 1)
- delete(root, 0)

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



j = 3 = k.length(): end of word. "dad" is marked, so set `isEnd = false`.

Step 4 / 10

VARIABLES

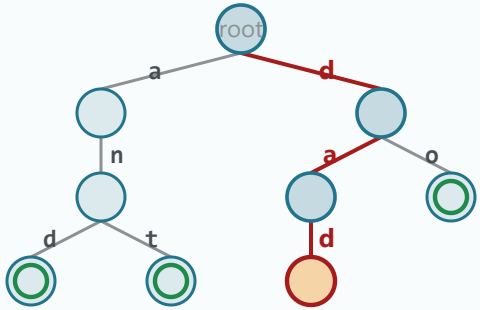
d = dad j = 3 i = – c = –

CALL STACK

- delete(dad, 3)
- delete(da, 2)
- delete(d, 1)
- delete(root, 0)

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
▶      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



isEmpty(dad) = true: the node has no children → return **true**; the parent may remove it.

Step 5 / 10

VARIABLES

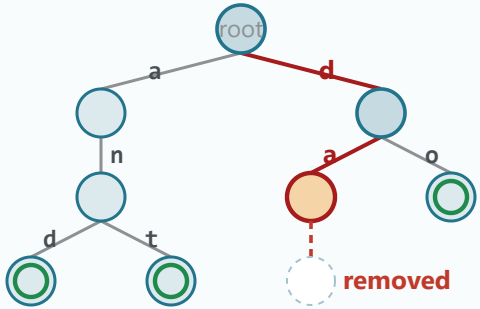
`d = dad` `j = 3` `i = –` `c = –`

CALL STACK

- `delete(dad, 3)`
- `delete(da, 2)`
- `delete(d, 1)`
- `delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
▶      d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



delete(da) returned true → `d.children[3] = null` : node "dad" is cut off.

Step 6 / 10

VARIABLES

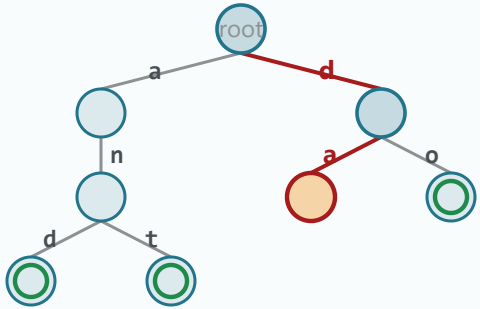
`d = da` `j = 2` `i = 3` `c = "dad"`

CALL STACK

- `delete(da, 2)`
- `delete(d, 1)`
- `delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
▶      return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



"da" is unmarked and now childless → **true**: this node can be removed too.

Step 7 / 10

VARIABLES

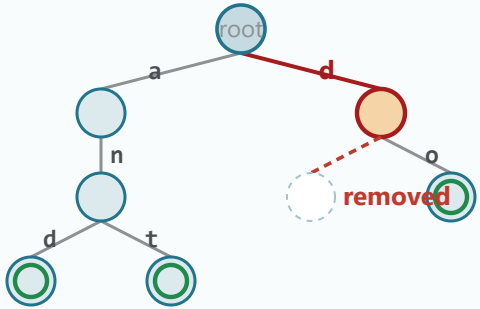
`d = da` `j = 2` `i = 3` `c = "dad"`

CALL STACK

- `delete(da, 2)`
- `delete(d, 1)`
- `delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
▶      d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



delete(d) returned true → `d.children[0] = null` : node "da" is cut off.

Step 8 / 10

VARIABLES

`d = d` `j = 1` `i = 0` `c = "da"`

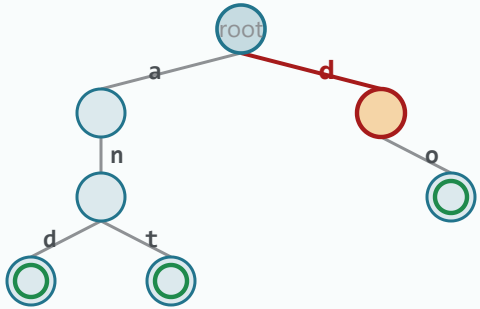
CALL STACK

`delete(d, 1)`

`delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
▶      return !d.isEnd && isEmpty(d);
15   }
16   return false;
17 }
```



"d" is unmarked but has the child 'o' for "do" → isEmpty is false → **false**. Deletion stops here.

Step 9 / 10

VARIABLES

`d = d` `j = 1` `i = 0` `c = "da"`

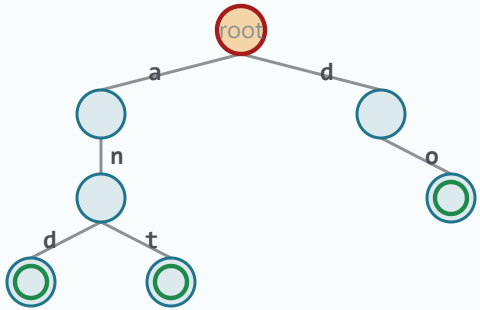
CALL STACK

`delete(d, 1)`

`delete(root, 0)`

delete(root, "dad", 0)

```
1  // true: the parent may remove d
2  boolean delete(TrieNode d, String k,
3                 int j) {
4    if (j == k.length()) {  // done
5      if (!d.isEnd) return false;
6      d.isEnd = false;
7      return isEmpty(d);   // childless?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieNode c = d.children[i];
11   if (c == null) return false;
12   if (delete(c, k, j + 1)) {
13     d.children[i] = null;  // remove child
14     return !d.isEnd && isEmpty(d);
15   }
▶  16   return false;
17 }
```



At the root, delete(d) returned false → the root removes nothing and returns false (the value is ignored). Remaining words: and, ant, do.

Step 10 / 10

VARIABLES

`d = root` `j = 0` `i = 3` `c = "d"`

CALL STACK

`delete(root, 0)`

ANALYSIS

Complexity

Operation	Time	Notes
insert(word)	O(L)	One step per character; at most L new nodes
search(word)	O(L)	Ends earlier if a branch is missing
startsWith(prefix)	O(P)	P: prefix length
complete(prefix)	O(P + S · M)	S: number of nodes in the subtree of the prefix node
delete(word)	O(L · M)	`isEmpty` scans 26 cells; O(L) with a child count

Memory: O(N · M)

N: number of nodes, M: alphabet size (26). N ≤ (total length of all words) + 1.

Independent of n

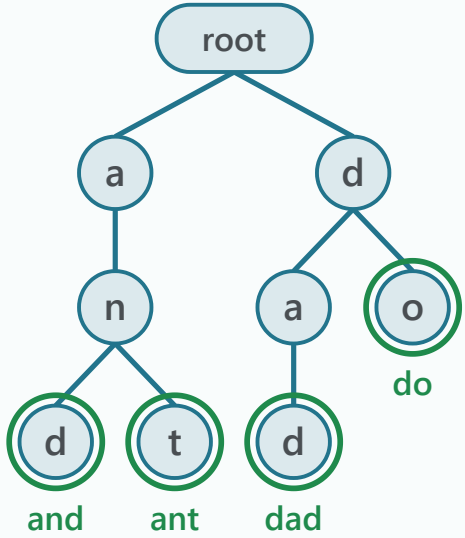
Running times do not depend on the **number** of stored words; there is no log n factor as in a BST.

ANALYSIS

Memory: A Worked Example

Trie: **and, ant, dad, do**

Total characters	3 + 3 + 3 + 2 = 11
Number of nodes N	1 (root) + 8 = 9
Child references	9 × 26 = 234
Used references	8 (number of edges)
Empty (null) references	226 (≈ 97%)



Shared prefixes shrink 11 characters to 8 nodes, but 26-element arrays fill most of the memory with **null**. The bigger the alphabet (Unicode: 65,536+), the worse it gets.

ANALYSIS

Pros and Cons

Pros

+ Fast **prefix**-based search: autocomplete.
+ Word lookup in **O(L)** time (L: word length).
+ Overlapping prefixes are stored once: **shared memory**.
+ Traversal yields words in **lexicographic order**; no hash collisions.

Cons

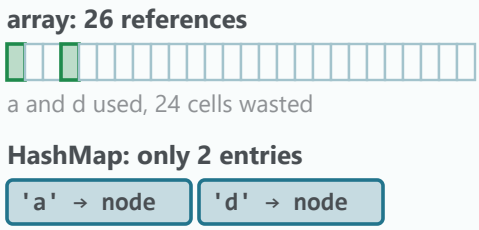
– **High memory** use, especially with large alphabets.
– Space complexity **O(N · M)** (N: number of nodes, M: alphabet size).
– No gain for sets with few common prefixes (e.g. random keys).
– Many small objects: not cache-friendly.

ANALYSIS

Ways to Store the Children

Representation	Child access	Memory per node	Notes
`TrieNode[26]` array	O(1)	M references	Simple, fast; wasteful for sparse nodes
`HashMap<Character, TrieNode>`	O(1) avg.	one per child	Works with any alphabet (Turkish, Unicode); no ordering
`TreeMap` / sorted list	O(log M)	one per child	Keeps alphabetical traversal
Ternary search trie (TST)	O(log M) avg.	3 references	Less / equal / greater link at each node

The trie's **logic** stays the same; only `node.children[i]` becomes a call such as `node.children.get(ch)`. Choose by alphabet size, memory and the need for ordered output.



ANALYSIS

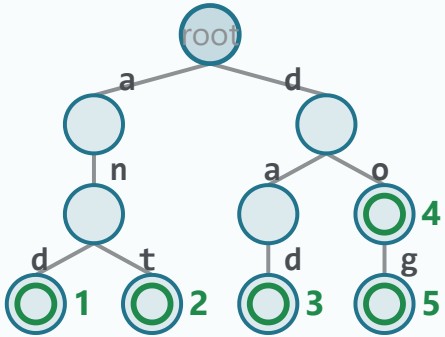
Trie, Hash Table or BST?

Operation (n words, length L)	Trie	Hash table	Balanced BST
Exact word lookup	O(L)	O(L) average	O(L · log n)
Prefix query	O(P)	not supported	O(P · log n)
k words with a prefix	O(P + S · M)	scan the whole table	O(P · log n + k · L)
Sorted listing	DFS, O(N · M)	needs sorting	inorder, O(n)
Worst case	O(L)	O(n · L) (collisions)	O(L · log n)
Memory	high	low	low

A hash table also reads all L characters to compute the key's hash, so for exact lookup the two are of the same order. The trie's edge is in **prefix** and **ordered** queries. S: number of nodes in the subtree of the prefix node.

Lexicographic Order with a Trie: Sorting Strings

- Input order: dog, ant, do, and, dad. Insert them all into a trie.
- Traverse the trie in **preorder** from the root, visiting children in 'a' → 'z' order and emitting a word at each marked node.
- Output: and, ant, dad, do, dog. A node is emitted before its children, so a **prefix comes before its extensions** (do < dog).
- With N total characters and a fixed alphabet, insertion is O(N) and traversal O(N·M) (M = 26 child checks): no need for the O(n log n) comparisons of a comparison sort, each costing O(L).
- The same idea underlies burst sort and string versions of MSD radix sort.



Green: output order in the preorder traversal.

Applications

Autocomplete

Search boxes, IDEs: descend to the typed prefix's node and suggest the words in its subtree (most frequent first).

Spell checking

The dictionary lives in a trie; for an unknown word, search the trie for nearby words via small edits (insert / delete / replace).

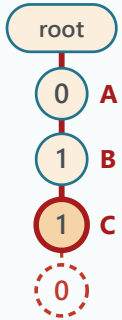
IP routing

Longest prefix match with a binary trie (alphabet {0, 1}): address 0110… goes to the longest of 0* / 01* / 011*, namely 011*.

Word games and T9

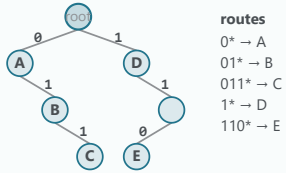
Boggle / word puzzles: abandon a path as soon as it is not a prefix (pruning). Suggesting words from a key sequence.

Longest prefix match: routes 0* → A, 01* → B, 011* → C. For address 0110…, descend 0 → 1 → 1 from the root; there is no '0' branch. The **last** route seen on the way is C → the packet goes to C.



longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
▶    BitNode d = root;
▶    String best = null;
4    for (int k = 0; k < addr.length(); k++) {
5      if (d.route != null) best = d.route;  // longer prefix
6      d = d.children[addr.charAt(k) - '0'];
7      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



Binary trie (alphabet {0, 1}); some nodes carry a route. addr = "0110": d = root, best = null.

Step 1 / 9

VARIABLES

addr = **"0110"**

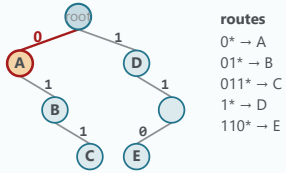
k = **–** d = **root**

best = **null**

OUTPUT

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 0: no route at the root. Bit '0' → d = "0".

Step 2 / 9

VARIABLES

addr = "0110"

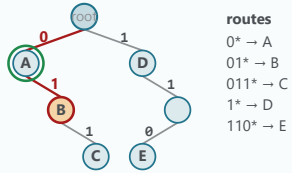
k = 0 d = "0"

best = null

OUTPUT

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 1: node "0" has A → best = A. Bit '1' → d = "01".

Step 3 / 9

VARIABLES

addr = "0110"

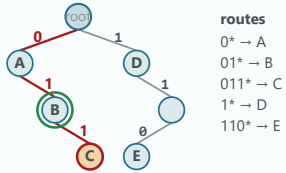
k = 1 d = "01"

best = A

OUTPUT

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 2: "01" has B → best = B (the longer prefix wins). d = "011".

Step 4 / 9

VARIABLES

addr = "0110"

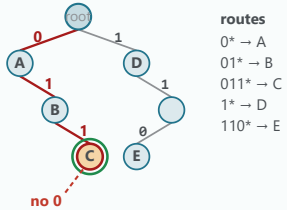
k = 2 d = "011"

best = B

OUTPUT

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 3: "011" has C → best = C. No '0' branch → d = null, return **C**.

Step 5 / 9

VARIABLES

addr = "0110"

k = 3 d = null

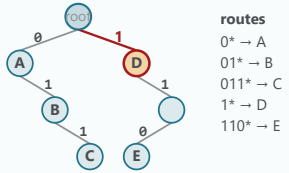
best = C

OUTPUT

```
C
```

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
▶    BitNode d = root;
▶    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



New address "1001": k = 0, no route at the root; d = "1".

Step 6 / 9

VARIABLES

addr = "1001"

k = 0 d = "1"

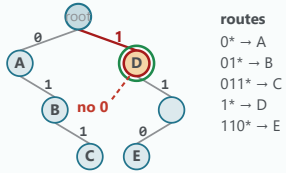
best = null

OUTPUT

```
C
```

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 1: "1" has D → best = D. No '0' branch → return **D**. Without a more specific route, the general one applies.

Step 7 / 9

VARIABLES

addr = "1001"

k = 1 d = null

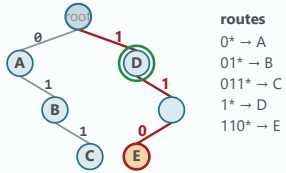
best = D

OUTPUT

```
C
D
```

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
▶    BitNode d = root;
▶    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



"1101": descend 1 → 11 → 110. "11" is an internal node (no route), so best stays D.

Step 8 / 9

VARIABLES

addr = "1101"

k = 2 d = "110"

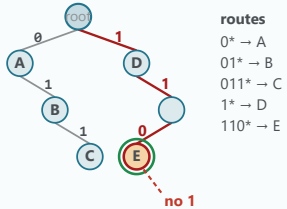
best = D

OUTPUT

```
C
D
```

longestMatch(): Routing with a Binary Trie

```
1  String longestMatch(String addr) {
2    BitNode d = root;
3    String best = null;
▶    for (int k = 0; k < addr.length(); k++) {
▶      if (d.route != null) best = d.route;  // longer prefix
▶      d = d.children[addr.charAt(k) - '0'];
▶      if (d == null) return best;
8    }
9    return d.route != null ? d.route : best;
10 }
```



k = 3: "110" has E → best = E; no '1' branch → **E**. The number of steps is bounded by the address length.

Step 9 / 9

VARIABLES

`addr = "1101"`

`k = 3` `d = null`

`best = E`

OUTPUT

```
C
D
E
```

The Maximum XOR Pair with a Binary Trie

- Question: which pair in [3, 10, 5, 25, 2, 8] maximizes a ⊕ b? Brute force tries n^2 pairs.
- Insert the numbers into a 5-bit binary trie, starting from the **most significant bit**.
- For each x, descend from the root and at every bit take the **opposite** branch if it exists: that bit of the result becomes 1. A 1 in a higher bit outweighs all lower bits combined, so the greedy choice is correct.
- For x = 5 = 00101 we want 11010: 1 ✓ (25), 1 ✓, 0 ✓, no 1 → 0, no 0 → 1: we reach 11001 = 25, 5 ⊕ 25 = **28**.

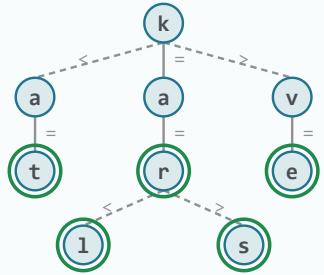
n	binary (5 bits)
3	`00011`
10	`01010`
5	`00101`
25	`11001`
2	`00010`
8	`01000`
5 ⊕ 25	`11100 = 28`

Method	time	n = 10^5, B = 30
All pairs	$O(n^2)$	≈ $5 \cdot 10^9$ operations
Binary trie	$O(n \cdot B)$	≈ $6 \cdot 10^6$ steps

Ternary Search Tries

- Each node holds **one character** and three links: `left` (smaller), `mid` (equal, go to the next character), `right` (larger).
- A node's left/right chain is a small binary search tree over the "sibling" children of a trie.
- Word ends are marked in the node (ring). Insertion order shapes the tree: the first word inserted becomes the "spine".

kar, kal, at, ve, kas	nodes	links
Trie with 26-element arrays	10	10 × 26 = 260
TST	9	9 × 3 = 27

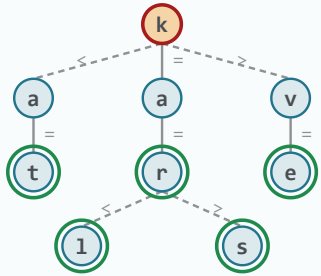


Solid line: mid (=); dashed: left (<) and right (>).

A search for L characters takes L mid steps plus a binary search among siblings at each level: O(L + log n) on average. With a large alphabet (Unicode), a TST keeps memory and speed in balance where arrays cannot (Bentley and Sedgewick, 1997).

TST: search("kas") and search("al")

```
1  boolean search(String s) {
▶    TstNode d = root;
▶    int i = 0;
4    while (d != null) {
5      char ch = s.charAt(i);
6      if (ch < d.c) d = d.left;
7      else if (ch > d.c) d = d.right;
8      else if (i == s.length() - 1) return
   d.end;
9      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



TST: kar, kal, at, ve, kas. search("kas"): d = root ('k'), i = 0.

Step 1 / 8

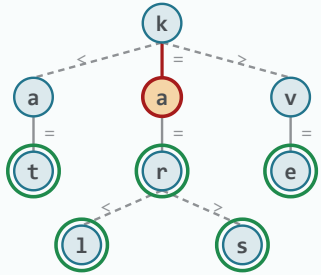
VARIABLES

s = "kas" | i = 0 | ch = – | d = 'k'

OUTPUT

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
▶      else if (ch > d.c) d = d.right;
▶      else if (i == s.length() - 1) return
   d.end;
▶      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



ch = 'k' = d.c and not the last character → mid link, i = 1. d = 'a'.

Step 2 / 8

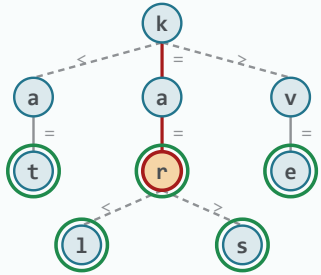
VARIABLES

s = "kas" | i = 1 | ch = 'k' | d = 'a'

OUTPUT

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
▶      else if (ch > d.c) d = d.right;
▶      else if (i == s.length() - 1) return
   d.end;
▶      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



ch = 'a' equal → mid, i = 2. d = 'r'.

Step 3 / 8

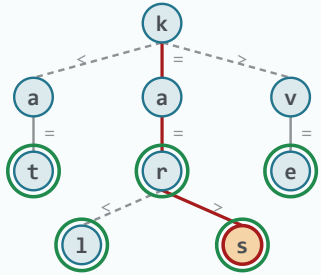
VARIABLES

s = "kas" | i = 2 | ch = 'a' | d = 'r'

OUTPUT

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
▶      else if (ch > d.c) d = d.right;
8      else if (i == s.length() - 1) return
   d.end;
9      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



ch = 's' > 'r' → **right** link; i is unchanged (same character, searching the siblings). d = 's'.

Step 4 / 8

VARIABLES

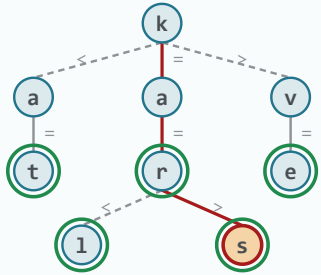
`s = "kas"` `i = 2`

`ch = 's'` `d = 's'`

OUTPUT

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
▶      else if (ch > d.c) d = d.right;
▶      else if (i == s.length() - 1) return
   d.end;
9      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



ch = 's' equal and i = 2 is the last index → return d.end: **true**. 4 nodes visited for 3 characters.

Step 5 / 8

VARIABLES

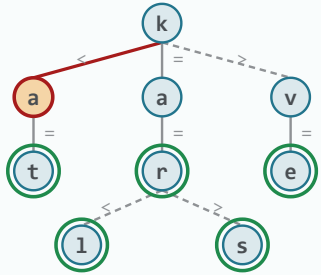
`s = "kas"` `i = 2` `ch = 's'` `d = 's'`

OUTPUT

```
true
```

TST: search("kas") and search("al")

```
1  boolean search(String s) {
▶    TstNode d = root;
▶    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
7      else if (ch > d.c) d = d.right;
8      else if (i == s.length() - 1) return
   d.end;
9      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



search("al"): ch = 'a' < 'k' → **left**, d = 'a' (the branch of the word at).

Step 6 / 8

VARIABLES

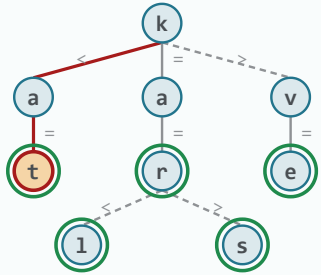
`s = "al"` `i = 0` `ch = 'a'` `d = 'a'`

OUTPUT

```
true
```

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
▶      else if (ch > d.c) d = d.right;
▶      else if (i == s.length() - 1) return
   d.end;
▶      else { d = d.mid; i++; }
10   }
11   return false;
12 }
```



'a' equal, not the last character → mid, i = 1, d = 't'.

Step 7 / 8

VARIABLES

`s = "al"` `i = 1`

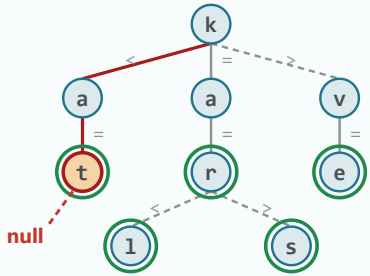
`ch = 'a'` `d = 't'`

OUTPUT

```
true
```

TST: search("kas") and search("al")

```
1  boolean search(String s) {
2    TstNode d = root;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.left;
7      else if (ch > d.c) d = d.right;
8      else if (i == s.length() - 1) return
   d.end;
9      else { d = d.mid; i++; }
▶    }
▶    return false;
12 }
```



'l' < 't' → left: null, **false**.

Step 8 / 8

VARIABLES

`s = "al"` `i = 1`

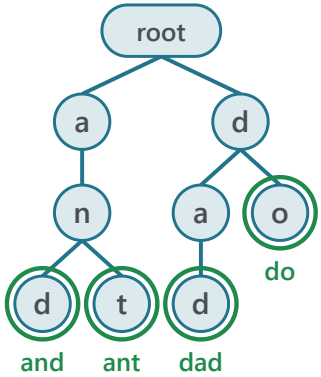
`ch = 'l'` `d = null`

OUTPUT

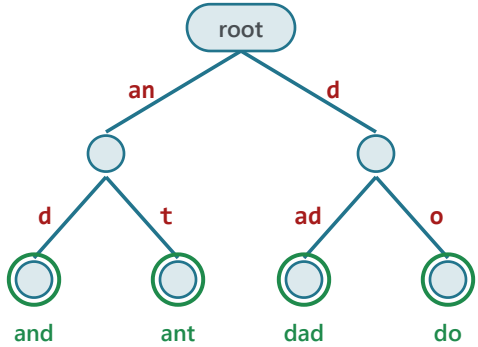
```
true
false
```

Compressed Tries (Radix / Patricia)

Standard trie: 8 nodes



Compressed trie: 6 nodes



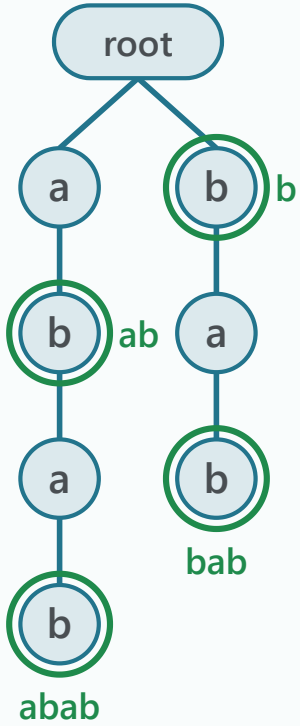
- Chains of **single-child, unmarked** nodes are merged into one edge, which now carries a **substring** ("an", "ad").
- Every internal node has at least 2 children (or an end-of-word mark), so the number of nodes is **O(n)** (n: number of words).
- Patricia (Morrison, 1968) is the binary-alphabet version of this idea; it is used in IP routing tables.

Suffix Tries

Insert **every suffix** of the text T = **"abab"** into a trie:

`abab` · `bab` · `ab` · `b`

- P is a **substring** of T ⇔ P is a prefix of some suffix ⇔ P is a path from the root.
- Query: `startsWith(P)` → **O(m)** (m = |P|), independent of the text length. E.g. "ba" is present, "bb" is not.
- The price: up to **$O(n^2)$** nodes for a text of length n.
- **Suffix tree** = compressed suffix trie: O(n) nodes, built in O(n) time by Ukkonen's algorithm.



suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
9        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```



t = "abab": its 4 suffixes (abab, bab, ab, b) will be inserted in turn. The trie is just the root.

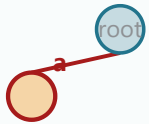
Step 1 / 13

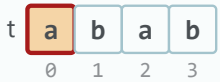
VARIABLES

s = – j = – i = –

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
▶          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 0: start inserting suffix "abab"; d = root. s = 0, j = 0: 'a' branch missing → new node, d moves there ("a").

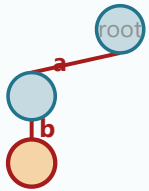
Step 2 / 13

VARIABLES

s = 0 j = 0 i = 0

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
▶          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 0, j = 1: 'b' branch missing → new node, d moves there ("ab").

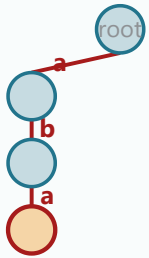
Step 3 / 13

VARIABLES

s = 0 j = 1 i = 1

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
▶          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 0, j = 2: 'a' branch missing → new node, d moves there ("aba").

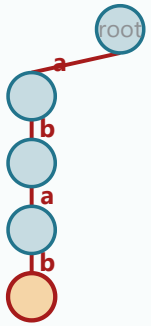
Step 4 / 13

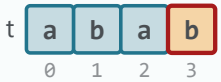
VARIABLES

s = 0 j = 2 i = 0

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
         int i = t.charAt(j) - 'a';
         if (d.children[i] == null)
           d.children[i] = new TrieNode();
         d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 0, j = 3: 'b' branch missing → new node, d moves there ("abab").

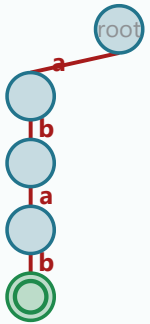
Step 5 / 13

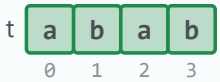
VARIABLES

s = 0 j = 3 i = 1

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
9        d = d.children[i];
10     }
▶      d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





Suffix "abab" done → mark the last node (ring).

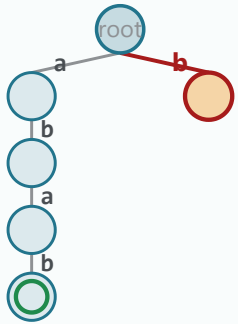
Step 6 / 13

VARIABLES

`s = 0` `j = 4` `i = 1`

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
         int i = t.charAt(j) - 'a';
         if (d.children[i] == null)
           d.children[i] = new TrieNode();
         d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 1: start inserting suffix "bab"; d = root. s = 1, j = 1: 'b' branch missing → new node, d moves there ("b").

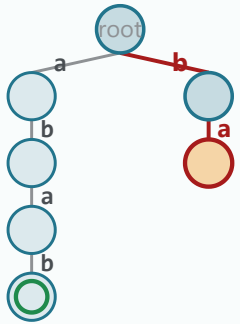
Step 7 / 13

VARIABLES

s = 1 j = 1 i = 1

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
▶          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 1, j = 2: 'a' branch missing → new node, d moves there ("ba").

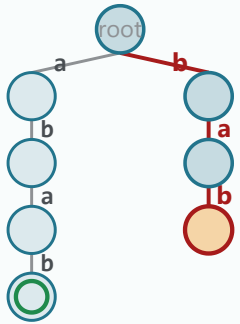
Step 8 / 13

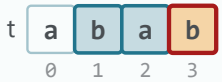
VARIABLES

s = 1 j = 2 i = 0

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
▶          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 1, j = 3: 'b' branch missing → new node, d moves there ("bab").

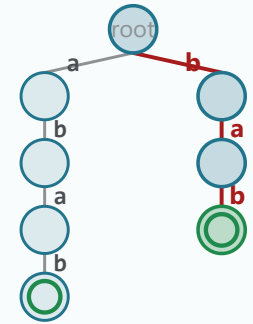
Step 9 / 13

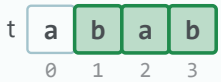
VARIABLES

s = 1 j = 3 i = 1

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
9        d = d.children[i];
10     }
▶      d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





Suffix "bab" done → mark the last node (ring).

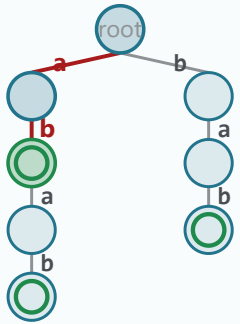
Step 10 / 13

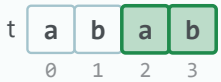
VARIABLES

`s = 1` `j = 4` `i = 1`

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
▶      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
▶      d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 2: the path "ab" already exists from inserting abab; descend without creating nodes and mark "ab".

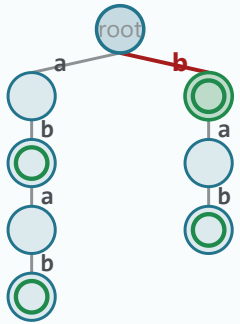
Step 11 / 13

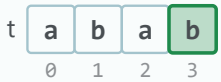
VARIABLES

`s = 2` `j = 4` `i = 1`

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
▶      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
▶        d = d.children[i];
10     }
▶      d.isEnd = true;  // suffix done
12   }
13   return root;
14 }
```





s = 3: the path "b" exists too; only the mark is added. Shared prefixes are stored once: 8 nodes including the root.

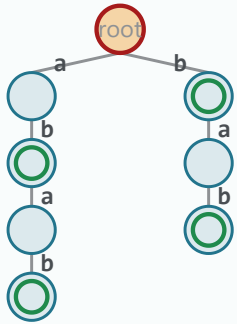
Step 12 / 13

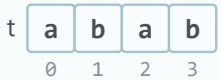
VARIABLES

`s = 3` `j = 4` `i = 1`

suffixTrie("abab")

```
1  TrieNode suffixTrie(String t) {
2    TrieNode root = new TrieNode();
3    for (int s = 0; s < t.length(); s++) {
4      TrieNode d = root;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.children[i] == null)
8          d.children[i] = new TrieNode();
9        d = d.children[i];
10     }
11     d.isEnd = true;  // suffix done
12   }
▶    return root;
14 }
```





The suffix trie is ready. "ba" is a path from the root → a substring; there is no path "bb" → not a substring. Query O(m).

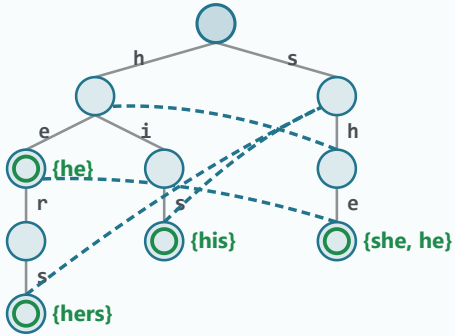
Step 13 / 13

VARIABLES

s = 4 | j = 4 | i = 1

Aho–Corasick: Many Patterns in One Pass

- The patterns (he, she, his, hers) go into a trie; the text is read **once**, left to right.
- **Failure link** (dashed): the longest proper suffix of the node's string that is in the trie. "she" → "he", "sh" → "h", "hers" → "s".
- When a branch is missing, fall back along the failure link; this is the multi-pattern counterpart of KMP's prefix table.
- Output sets merge along the failure chain: the "she" node also reports he.



Text "ushers"	state	found
u, s, h	root, s, sh	—
e (index 3)	she	she, he
r	she → he → her	—
s (index 5)	hers	hers

Time O(n + m + z): n text length, m total pattern length, z number of matches (Aho and Corasick, 1975). Used in virus signature scanning and banned-word filters.

Which Trie for Which Question?

Structure	Stores	Good at	Query
Standard trie	a set of words	exact word, prefix, autocomplete	O(L)
Trie with prefix counts	+ words through each node	“how many words have this prefix?”	O(P)
Compressed trie	single-child chains merged	long shared prefixes, memory	O(L)
Ternary search trie	3 links per node	large alphabets, little memory	O(L + log n) avg.
Binary trie	bit strings (IP, integers)	longest prefix match, maximum XOR	O(B)
Suffix trie / array	all suffixes (array: O(n) memory)	“is P a substring of T?”	O(m) / O(m log n)
Aho–Corasick	pattern trie + failure links	many patterns in one pass	O(n + z)

Same idea throughout: read the key a character (or bit) at a time and share prefixes; variants differ in what a node stores and how children are reached. Choose by question type first (exact, prefix, substring, many patterns), then by alphabet size and memory.

Summary

Topic	Key points
Structure	empty root; each edge one character; a word = a path from the root; common prefixes are shared
Node	`TrieNode[26] children` + `boolean isEnd`; index = ch − 'a'
Insert	create a node if missing, else move on; mark the end — O(L)
Search	false if a branch is missing; return `isEnd` at the end — O(L)
Prefix	same path, no mark check — O(P); complete = DFS from the prefix node
Delete	clear the mark; remove childless, unmarked nodes bottom-up
Memory	O(N · M); HashMap / TST / compressed trie for sparse nodes
Variants	radix/Patricia trie, suffix trie → suffix tree, suffix array

SUMMARY

Common Mistakes

Forgetting `isEnd`

Checking only that the path exists counts every prefix as a word: "da" appears to be found.

Confusing search with startsWith

A prefix query ignores the last node's mark; an exact search **must** check it.

Characters outside the alphabet

`'A' - 'a'` = −32, `'ş' - 'a'` = 254: out of array bounds. Lowercase the input or use a HashMap.

Deleting a shared node

Deleting the whole path also destroys the other words that share the prefix. Remove only childless, unmarked nodes.

Stating complexity in terms of n

Trie operations are not O(log n) or O(n) but **O(L)**: independent of the number of words.

Underestimating memory

Each node holds 26 references; millions of nodes take hundreds of MB. A sparse trie needs a HashMap or compression.

SUMMARY

Sample Problems

Insert the words in order: **car, cart, care, cat, dog** (trie with 26-element arrays).

1. Draw the trie. How many nodes, including the root?
2. What do `search("ca")` and `startsWith("ca")` return?
3. Write the result of `complete("car")` in order.
4. How many nodes do `delete("car")` and (from the original trie) `delete("cart")` remove?
5. How many child references are `null` in total?
6. How many nodes, including the root, does the **compressed** trie of the same five words have?

▶ **Show answers**

Exercises

1. `int countWords(TrieNode d)` : count the trie's words.
2. `int countPrefix(String prefix)` : return the number of words with this prefix in O(P).
3. `String longestCommonPrefix()` : find the longest common prefix of all words in the trie.
4. Write a trie with `HashMap<Character, TrieNode>` children for Turkish letters.
5. How many nodes, with root, does the trie of A, to, tea, ted, ten, i, in, inn have?
6. Search "abab"'s suffix trie for "bab" and "bb"; how many comparisons?

▶ **Answers / hints**

References

- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (Section 5.2: Tries).
- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- E. Fredkin, "Trie Memory", *Communications of the ACM* 3(9), 1960.
- D. R. Morrison, "PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM* 15(4), 1968.
- U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM Journal on Computing* 22(5), 1993.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

The End

Chapter 11: Tries

Previous: Chapter 10: Advanced String Algorithms · Next course: Advanced Algorithms