Patricia Tries

CMSC 420

Acronyms in Computer Science

- **PATRICIA** stands for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric

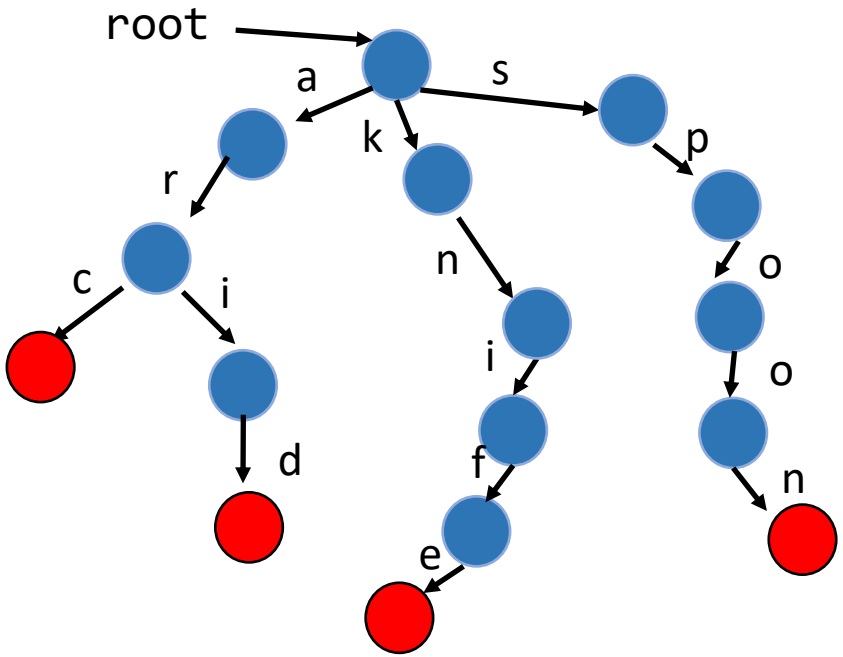
Problems with tries

- “Non-key” nodes with just one (non-null) child are about as useful as a Greek State Employee.

Source: Google Image Search for «δημόσιος υπάλληλος», which is Greek for “state employee”.

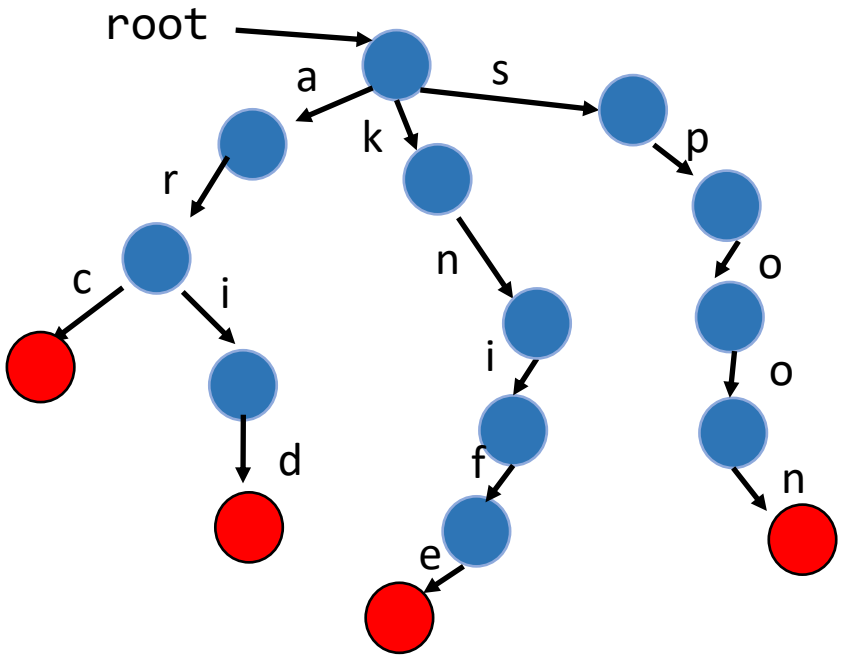
Problems with tries

- "Non-key" nodes with just one (non-null) child are about as useful as a Greek State Employee.
- Look at this example of a trie with two long keys, and two not so long ones, but which do share a prefix:



Problems with tries

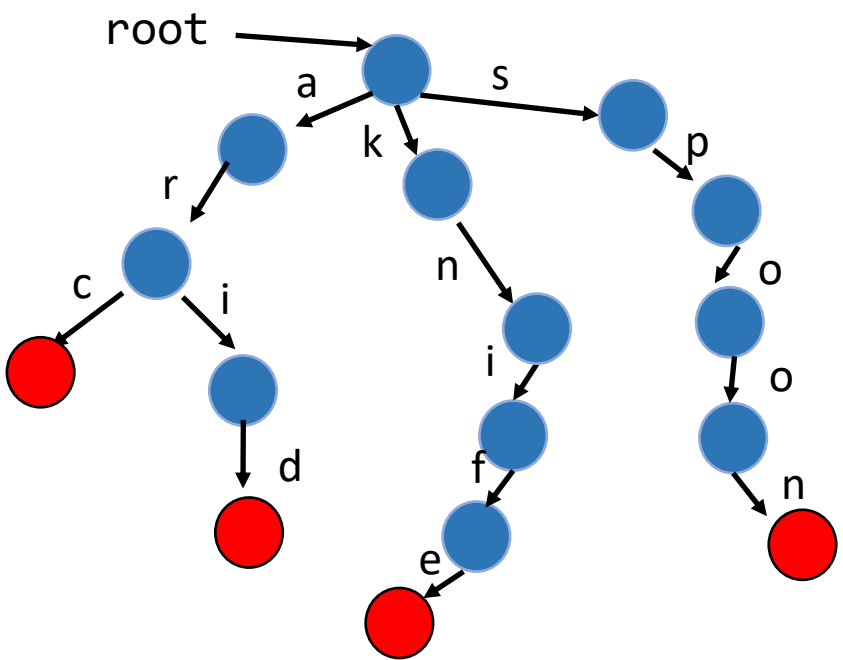
- "Non-key" nodes with just one (non-null) child are about as useful as a Greek State Employee.
- Look at this example of a trie with two long keys, and two not so long ones, but which do share a prefix:



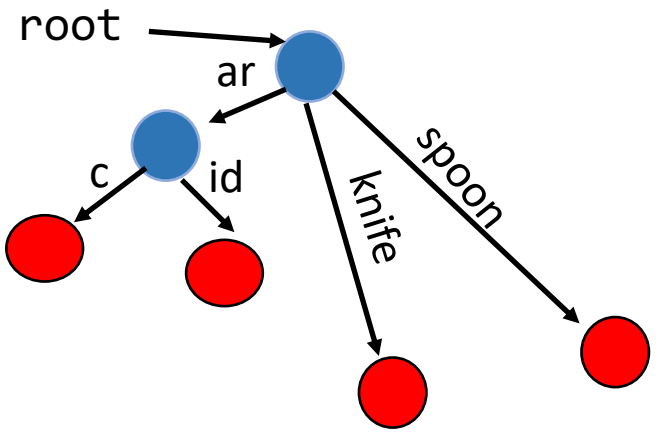
The problem stems from the fact that every link is associated with a character…

Problems with tries

- "Non-key" nodes with just one (non-null) child are about as useful as a Greek State Employee.
- Look at this example of a trie with two long keys, and two not so long ones, but which do share a prefix:

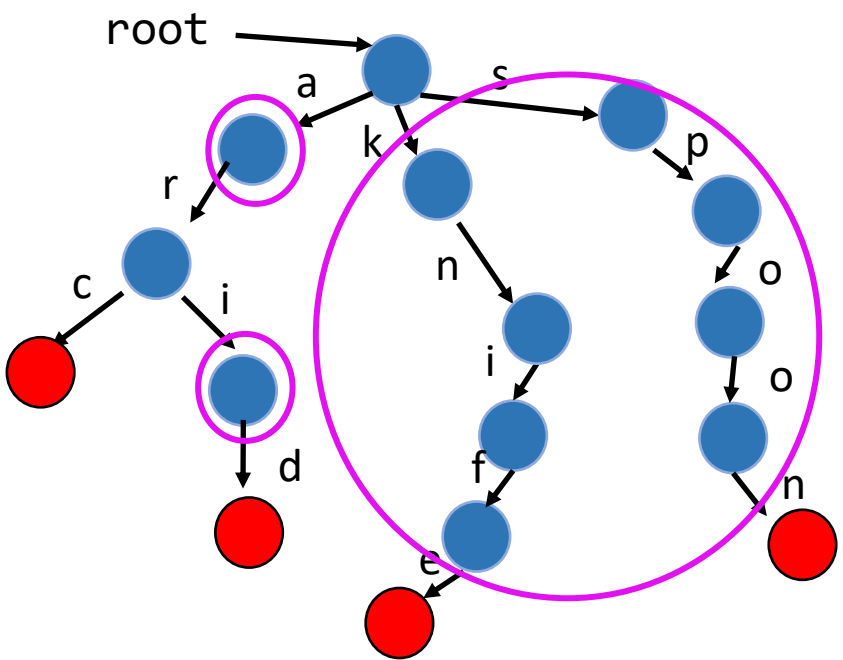


But if we were somehow able to associate entire paths with a string, compressing links…

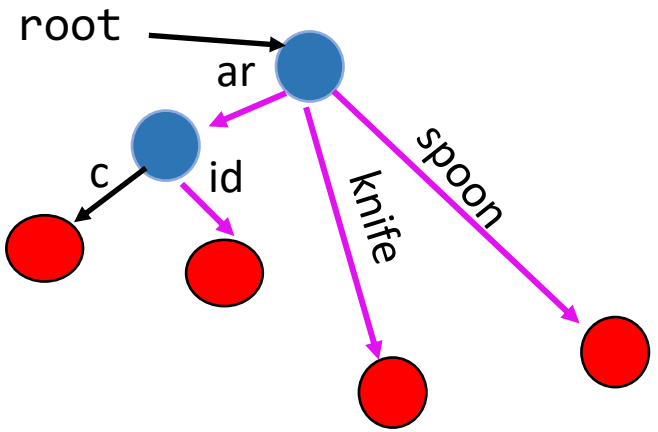


Problems with tries

- "Non-key" nodes with just one (non-null) child are about as useful as a Greek State Employee.
- Look at this example of a trie with two long keys, and two not so long ones, but which do share a prefix:

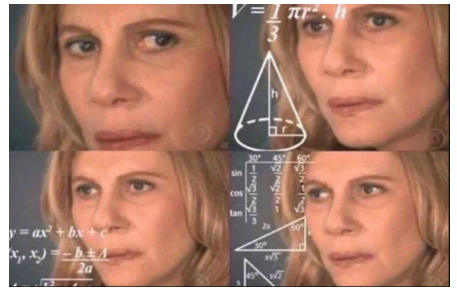


But if we were somehow able to associate entire paths with a string, compressing links…

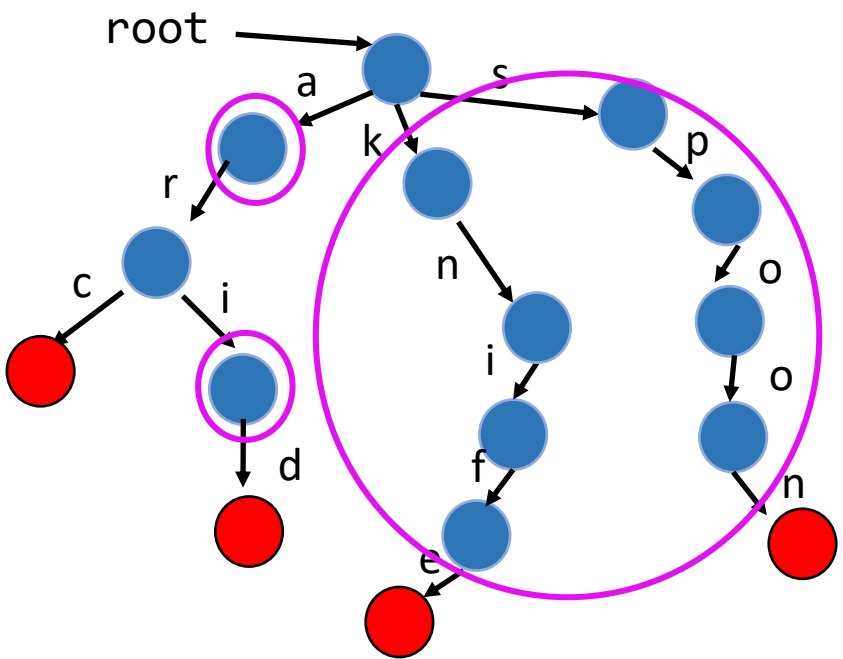


Note: All non-key-containing nodes with just one child node have now vanished, and the paths traversing them have been compressed to a single link!

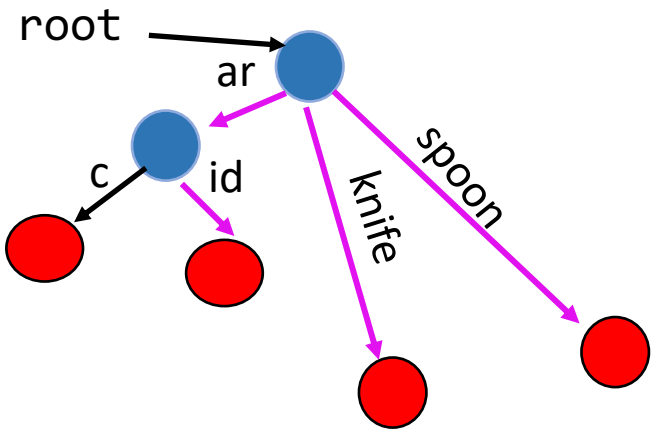
Problems with tries



- "Non-key" nodes with just one (non-null) child are about as useful as a Greek State Employee.
- Look at this example of a trie with two long keys, and two not so long ones, but which do share a prefix:



But if we were somehow able to associate entire paths with a string, compressing links…



Note: All non-key-containing nodes with just one child node have now vanished, and the paths traversing them have been compressed to a single link!

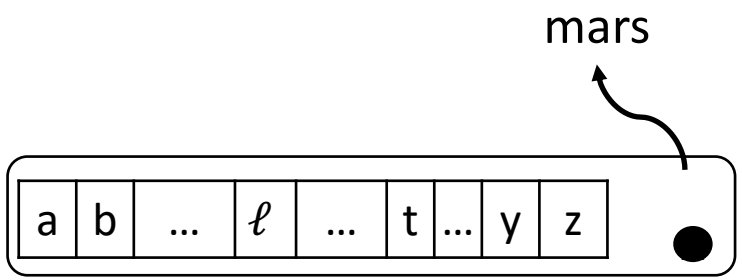
So how can we associate links with entire strings?

Patricia node structure

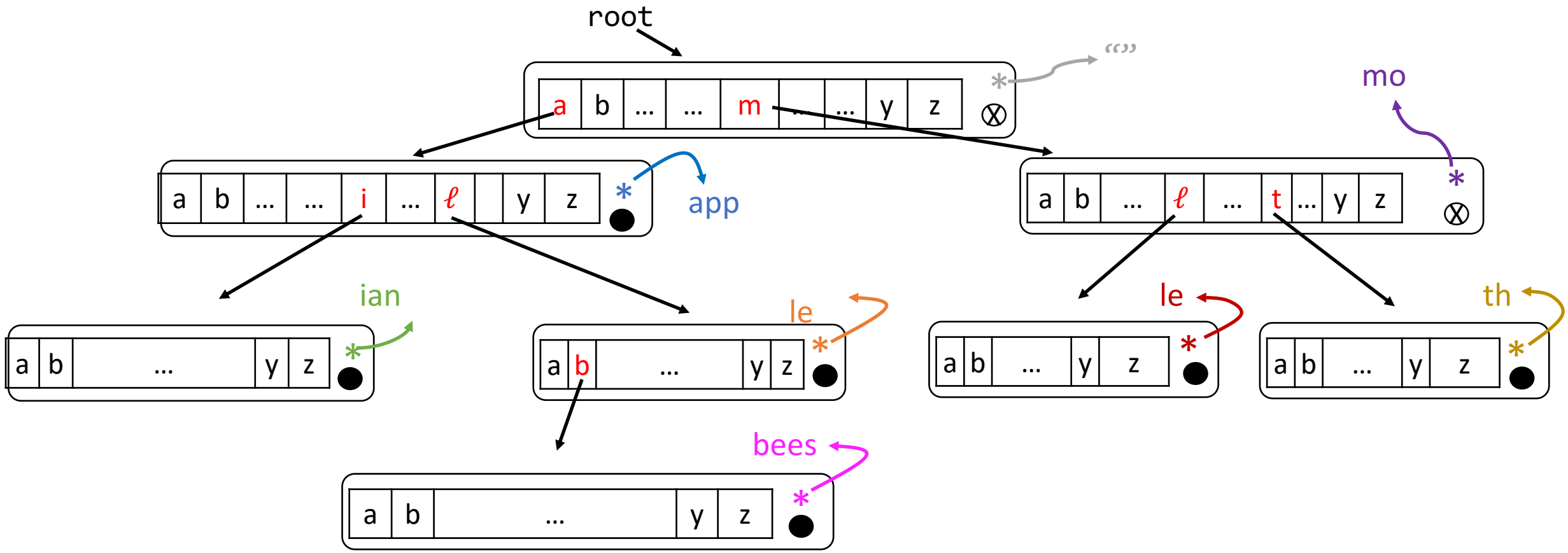
```
private class PatriciaNode{
        // Pointers to next elements
        Node[] arr;
        // Am I key or a “splitter” node?
        boolean isKey;
        // What key am I storing?
        String keyRef;
}
```

Patricia node structure

```
private class PatriciaNode{
        // Pointers to next elements
        Node[] arr;
        // Am I key or a “splitter” node?
        boolean isKey;
        // What key am I storing?
        String keyRef;
}
```

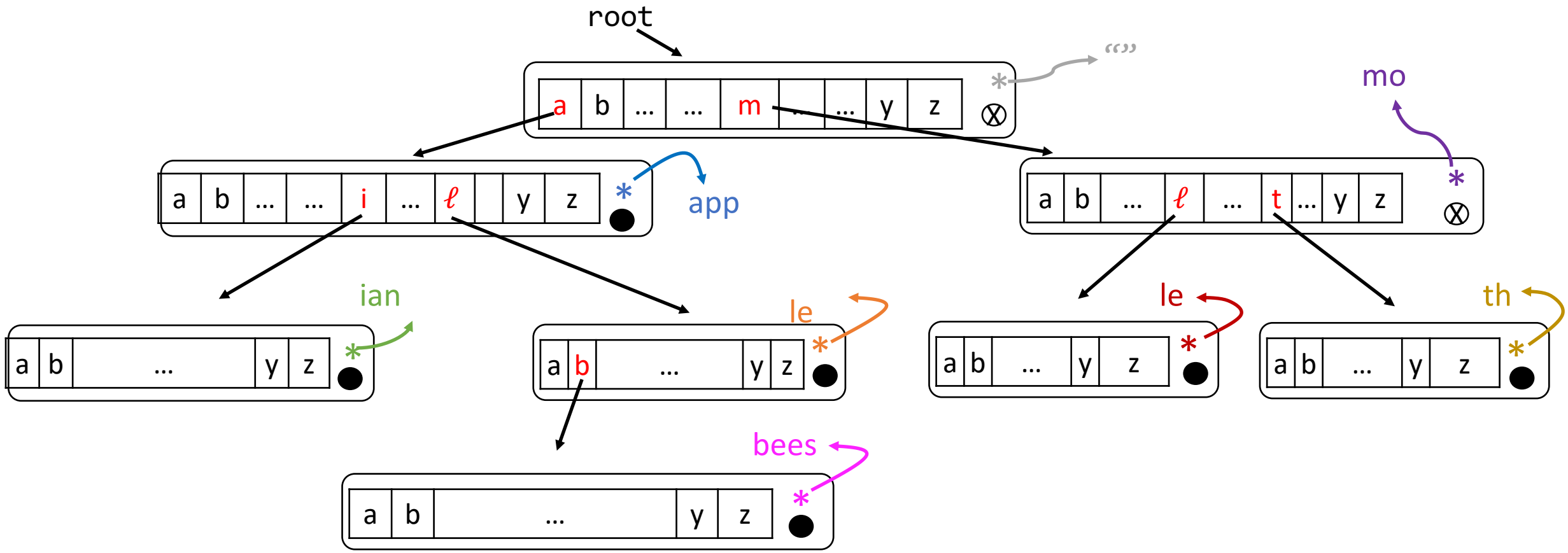


Patricia node structure



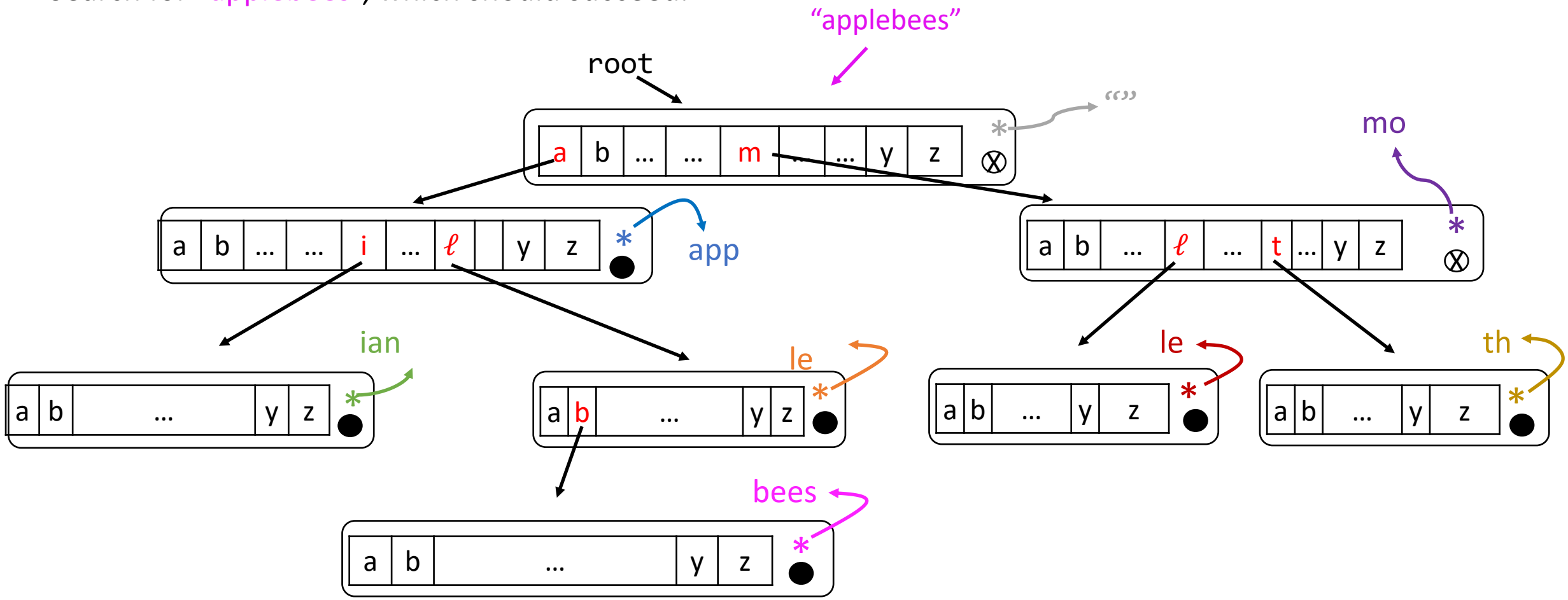
Search examples

- Search for “applebees”, which should succeed.



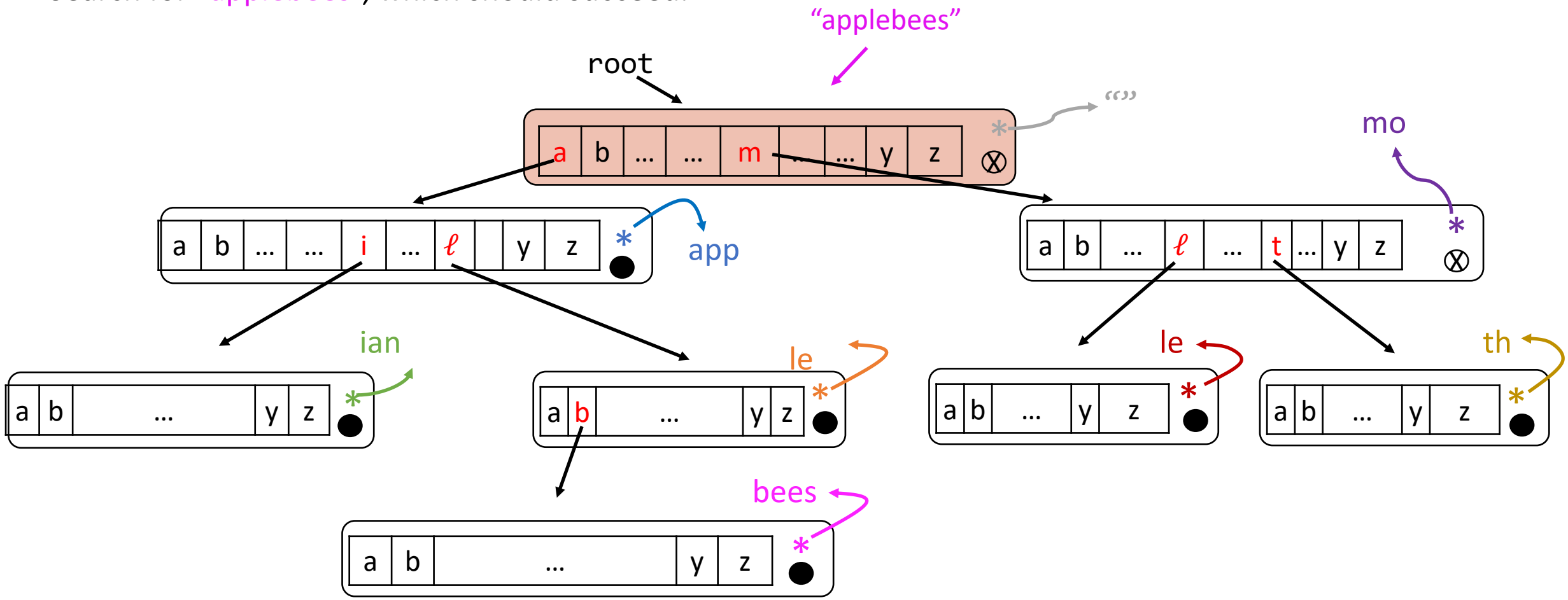
Search examples

- Search for “applebees”, which should succeed.



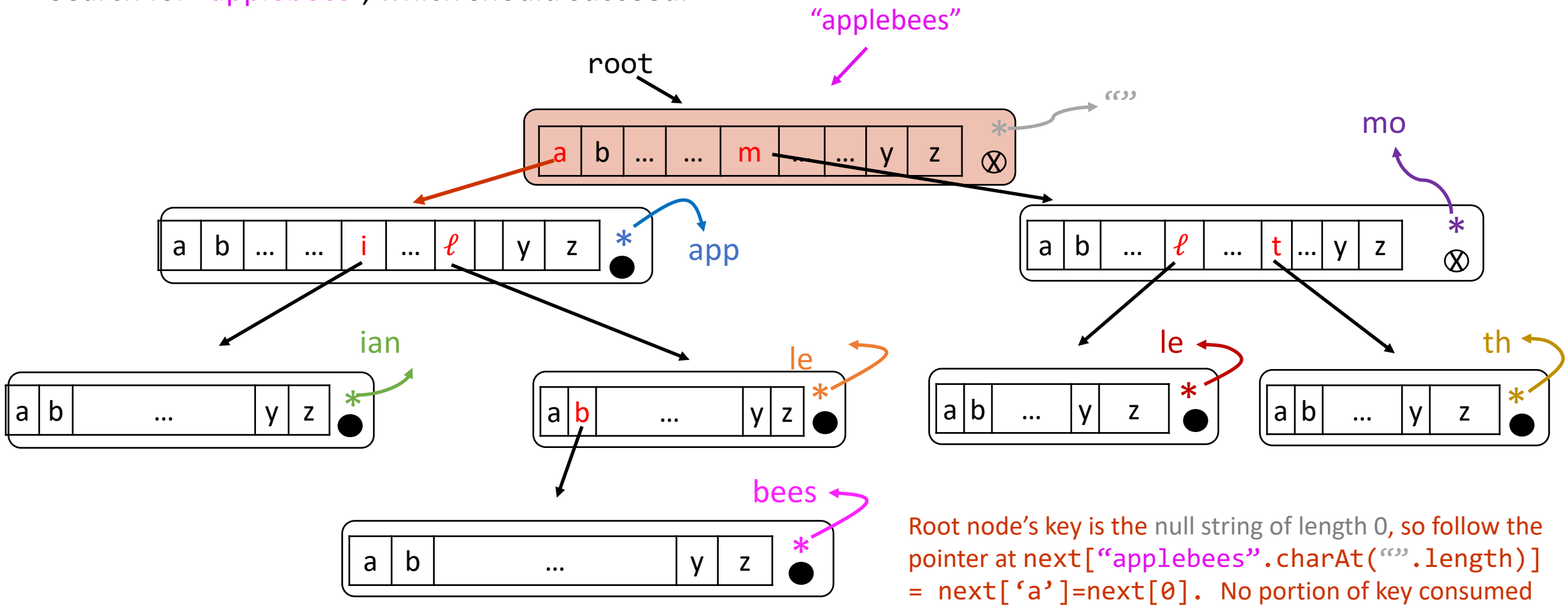
Search examples

- Search for “applebees”, which should succeed.



Search examples

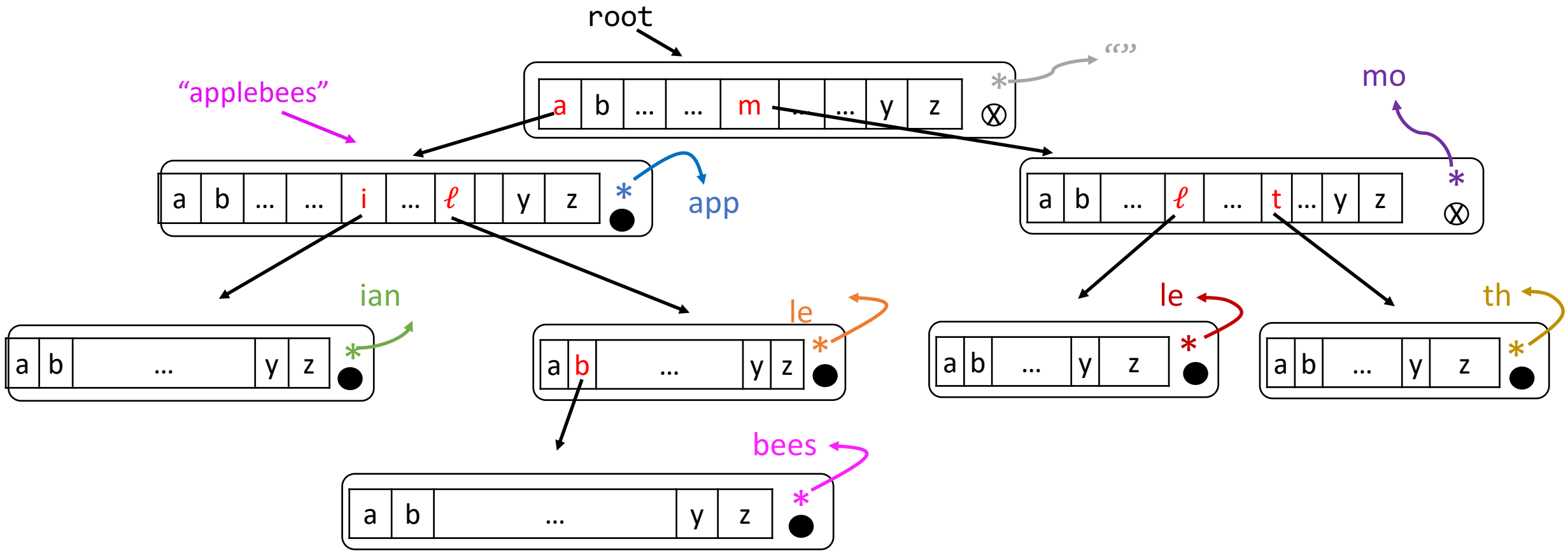
- Search for “applebees”, which should succeed.



Root node’s key is the null string of length 0, so follow the pointer at `next[“applebees”.charAt(“”.length)] = next[‘a’]=next[0].` No portion of key consumed since length of null string is 0!

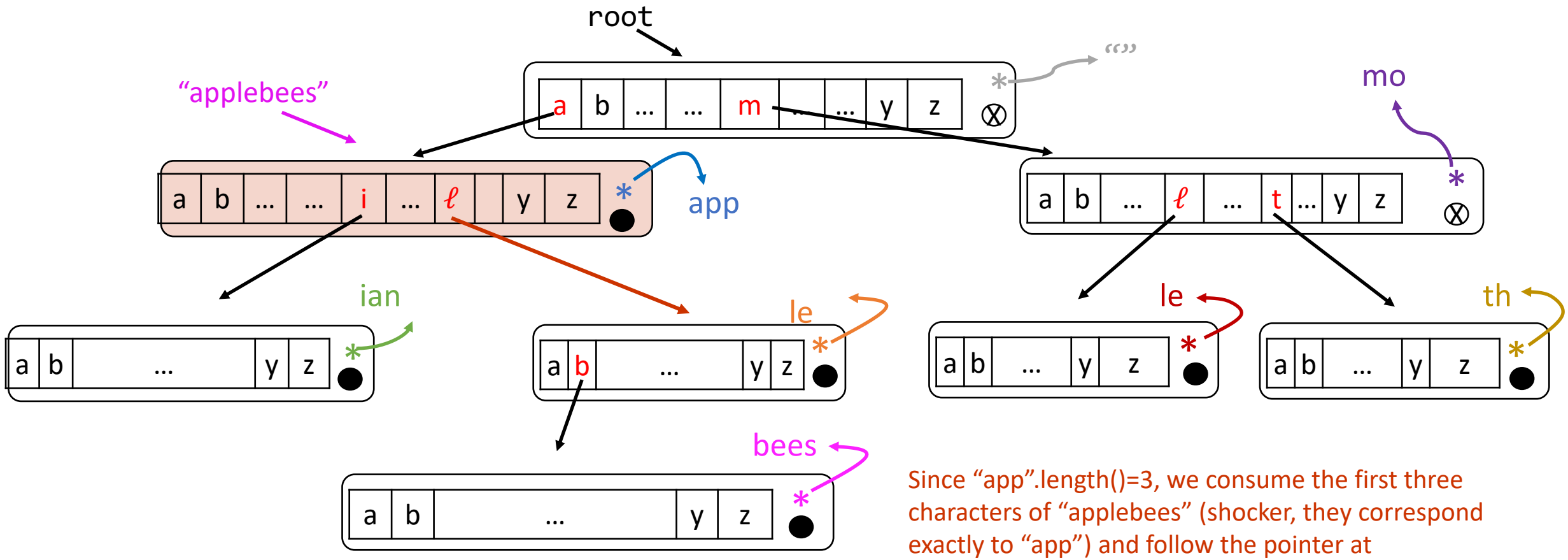
Search examples

- Search for “applebees”, which should succeed.



Search examples

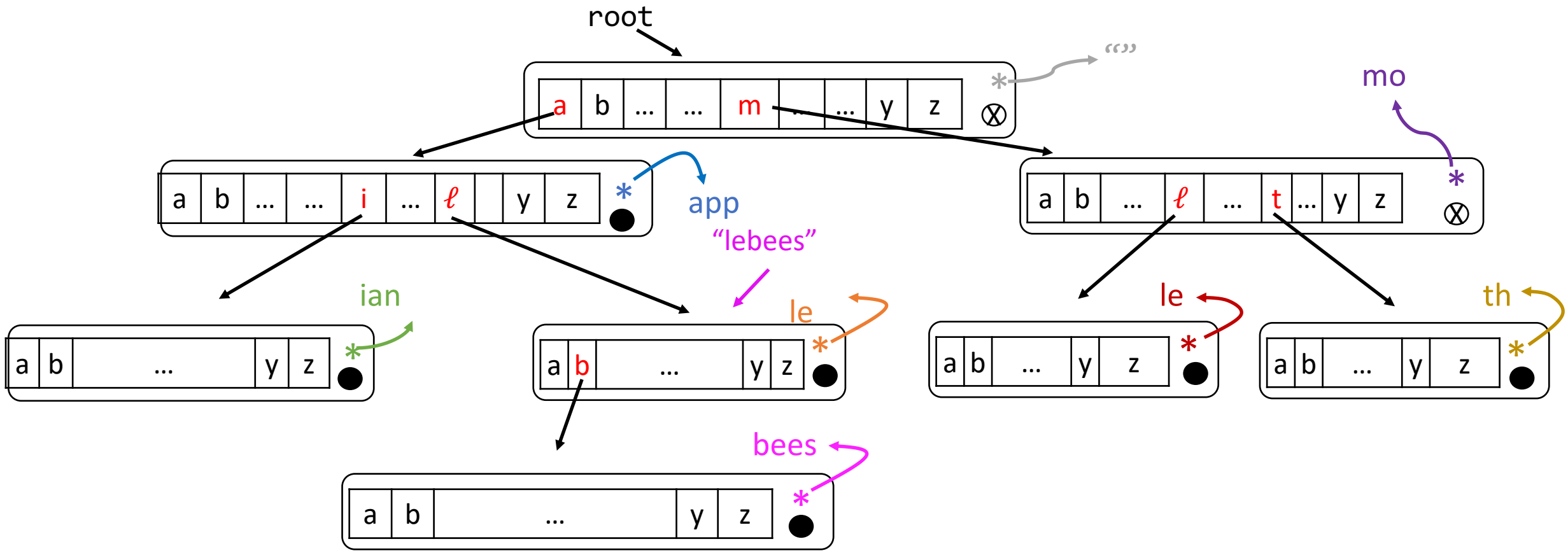
- Search for “applebees”, which should succeed.



Since “app”.length()=3, we consume the first three characters of “applebees” (shocker, they correspond exactly to “app”) and follow the pointer at `next[“applebees”.charAt(“app”.length()) = next[′ℓ′]=next[11]`

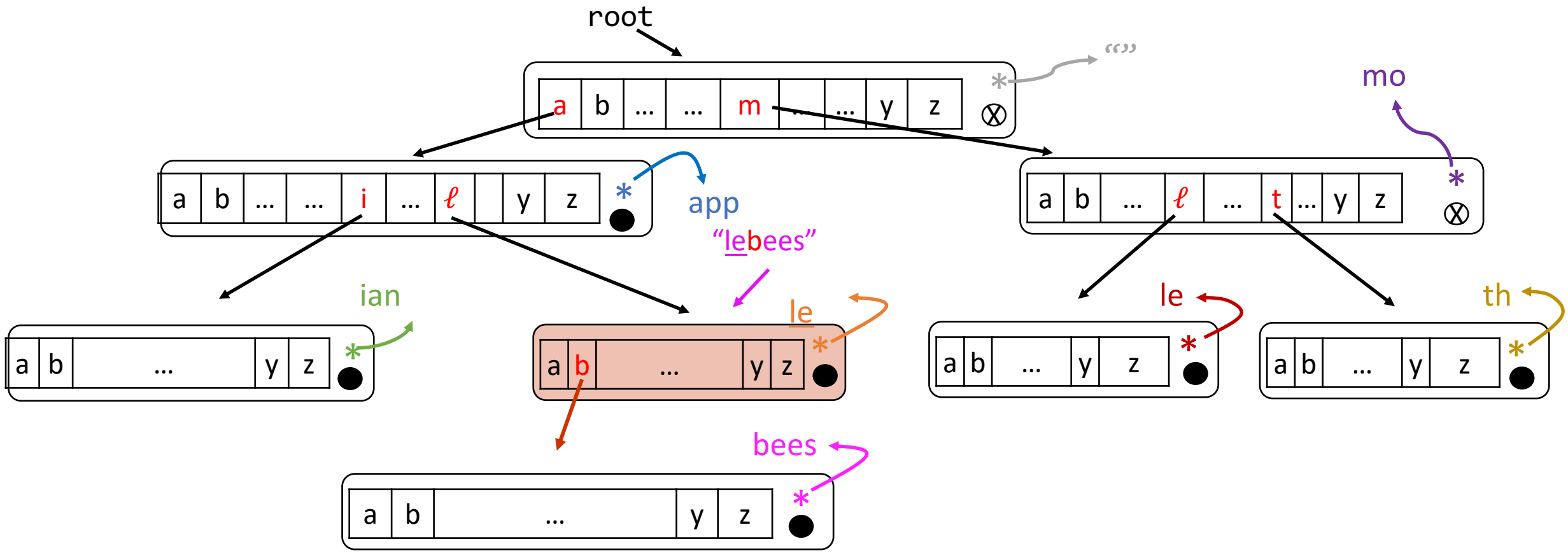
Search examples

- Search for “applebees”, which should succeed.



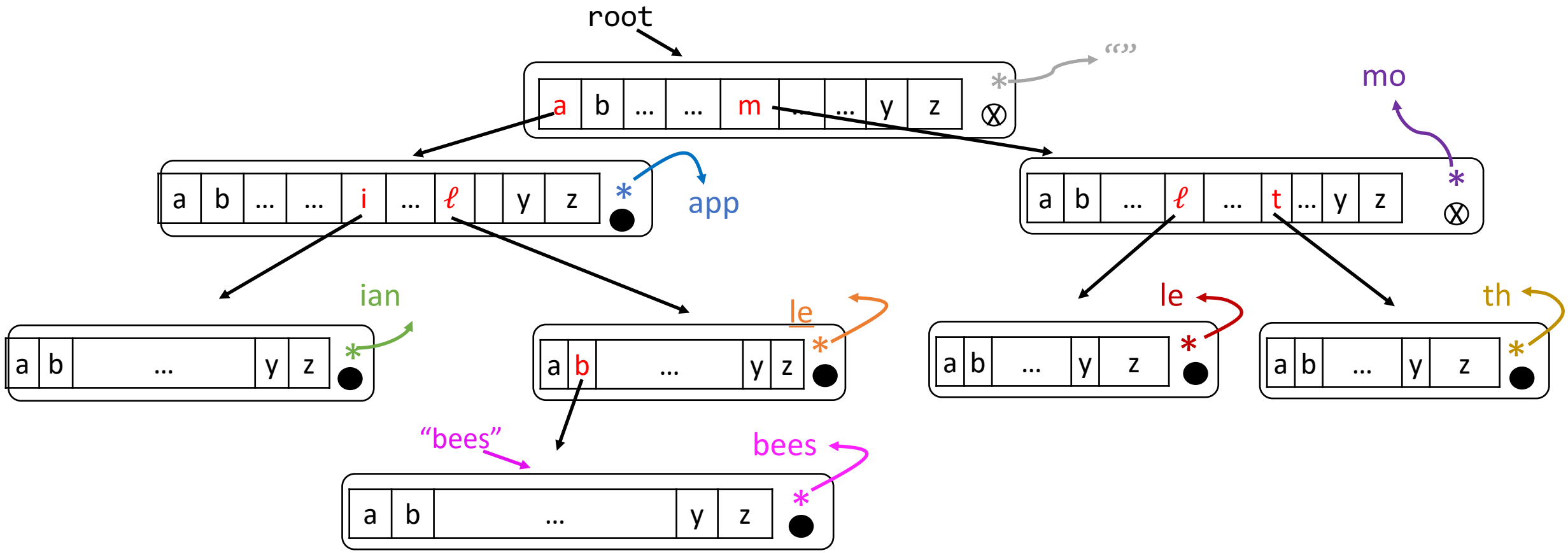
Search examples

- Search for “applebees”, which should succeed.



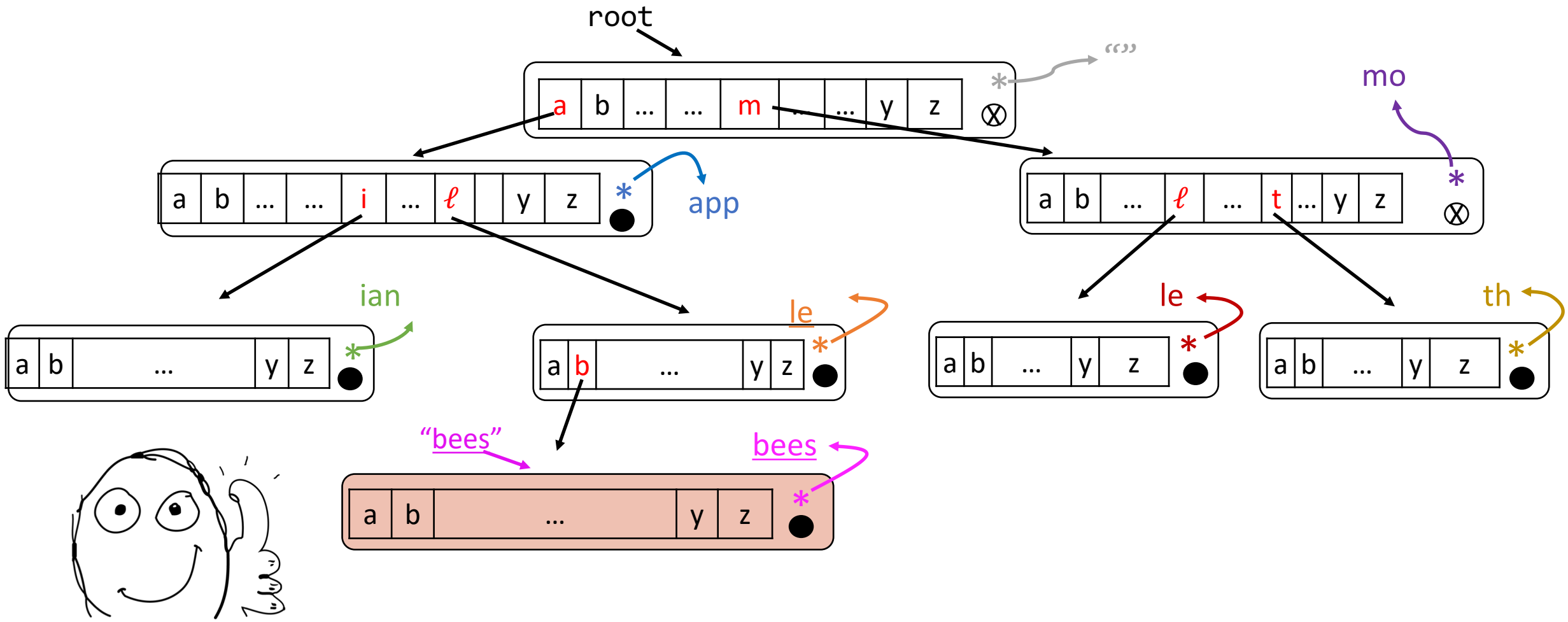
Search examples

- Search for “applebees”, which should succeed.



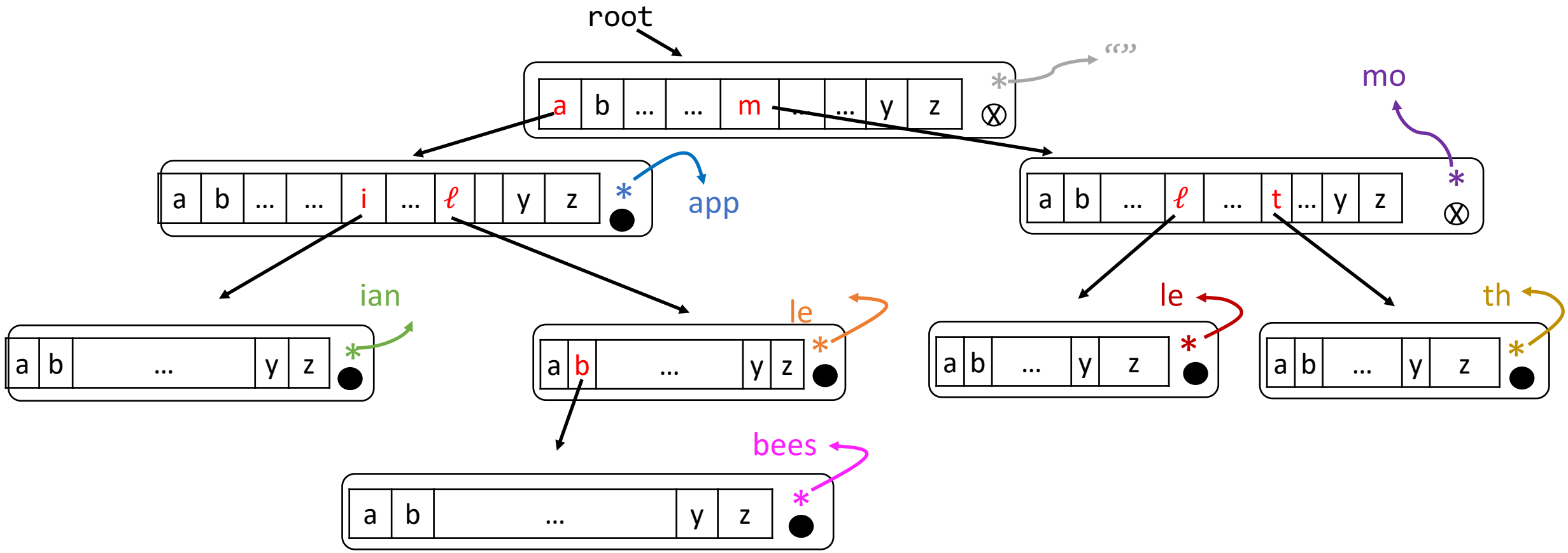
Search examples

- Search for “applebees”, which should succeed.



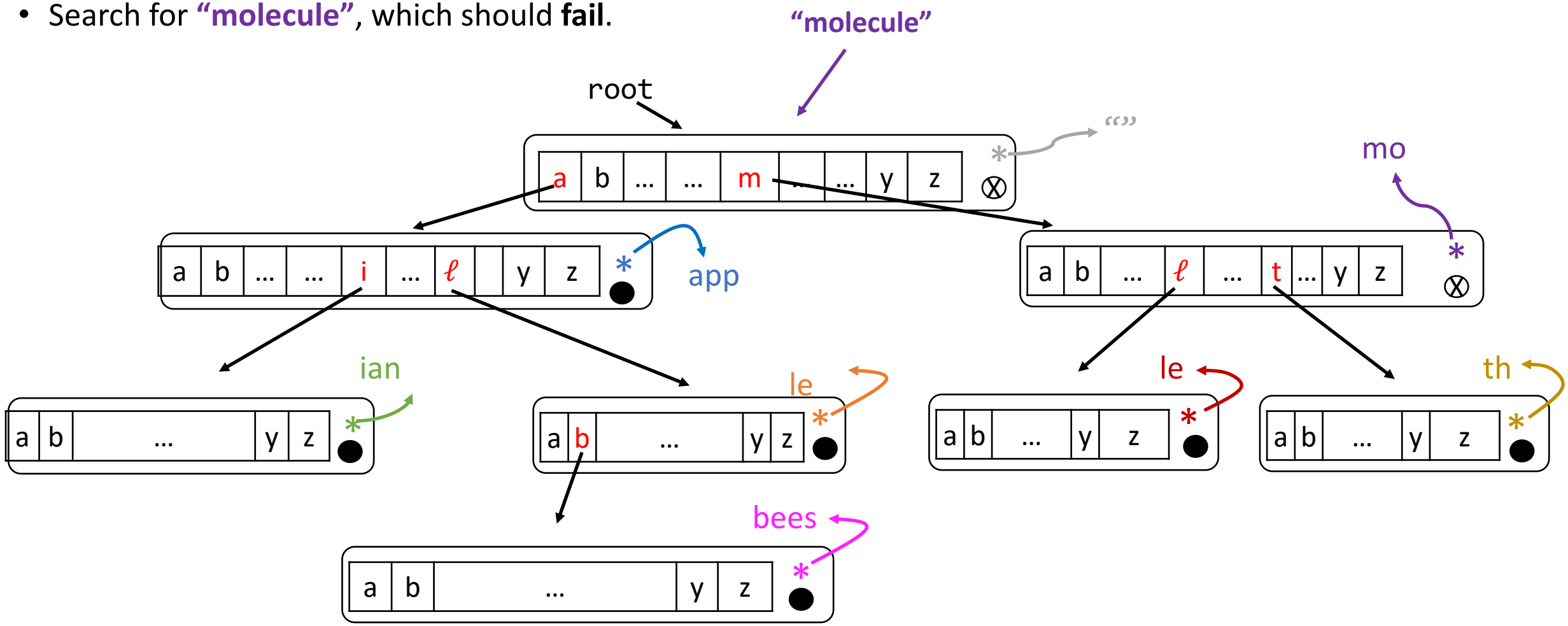
Search examples

- Search for **“molecule”**, which should **fail**.



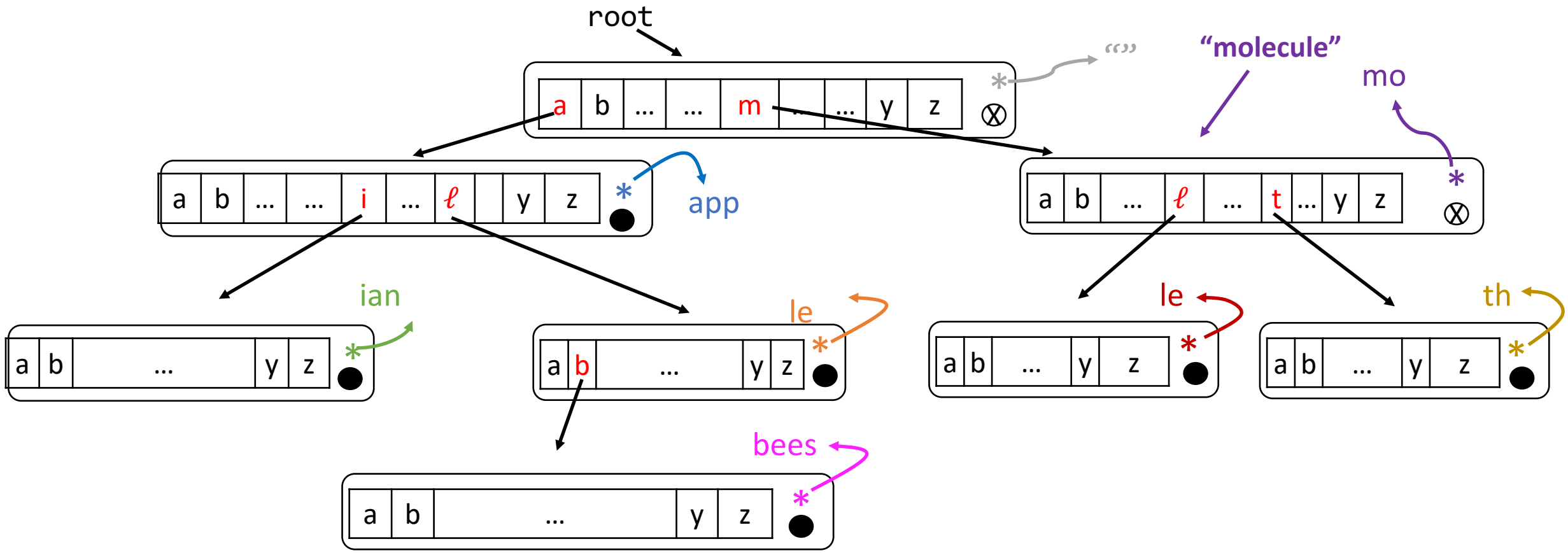
Search examples

- Search for **“molecule”**, which should **fail**.



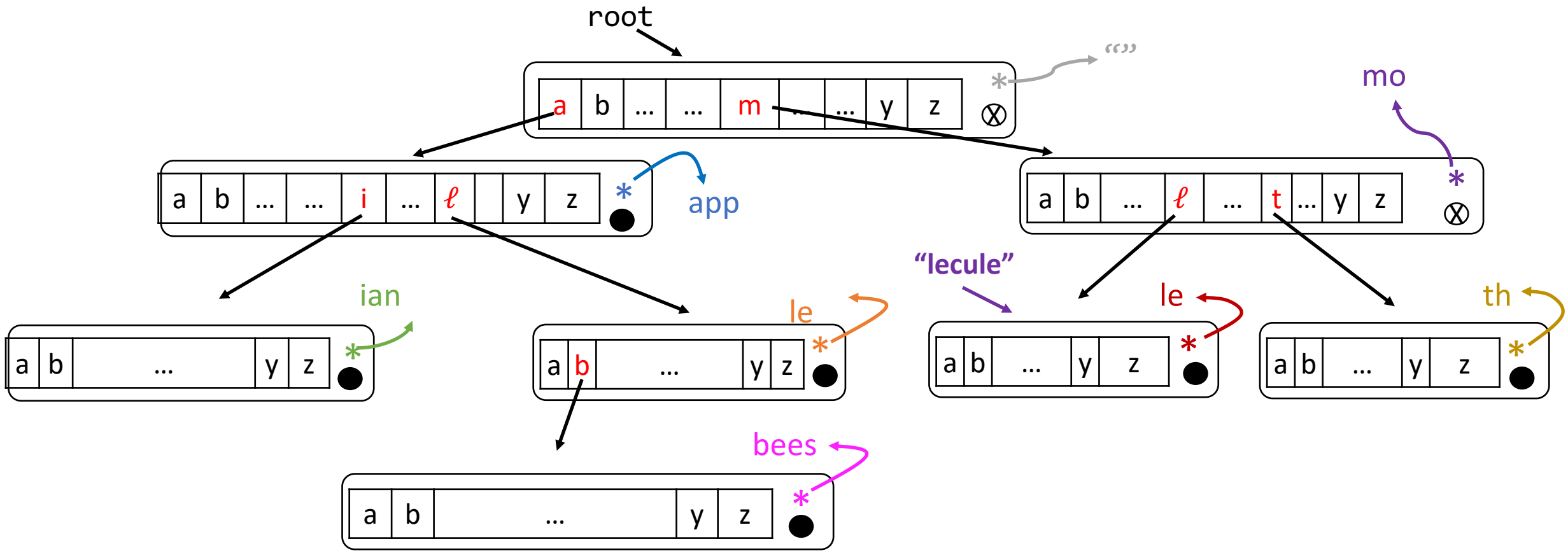
Search examples

- Search for **“molecule”**, which should **fail**.



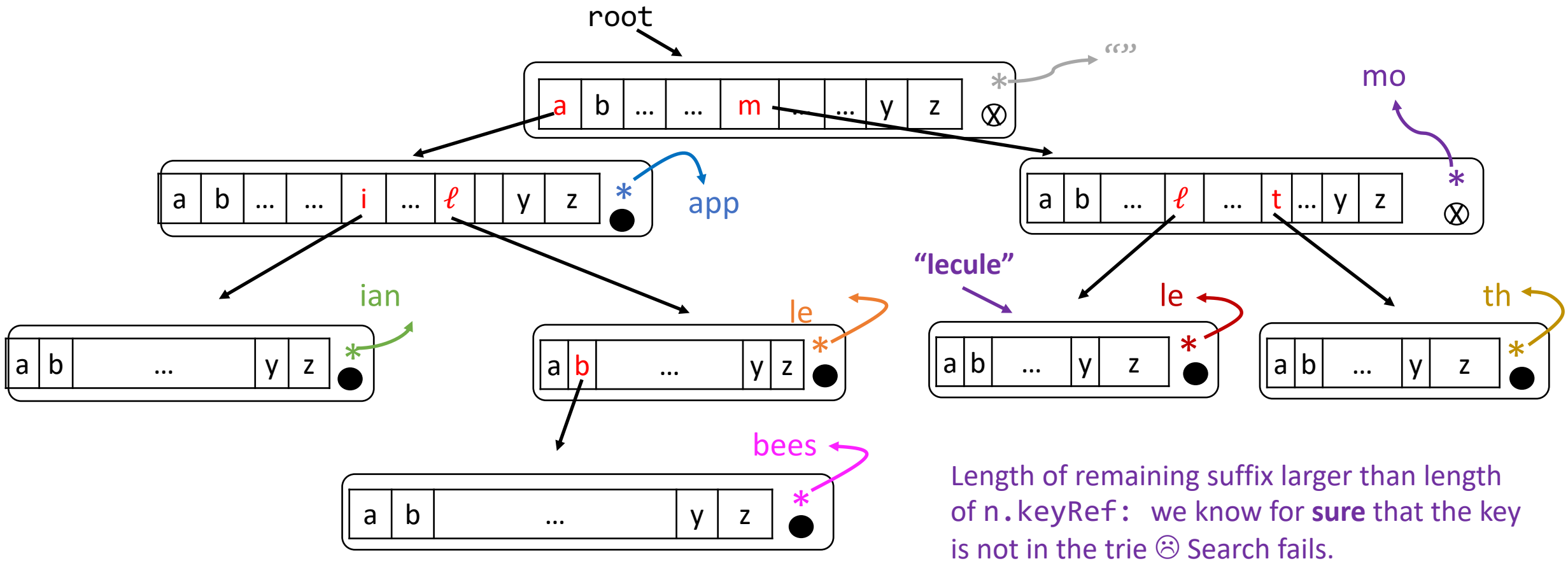
Search examples

- Search for **“molecule”**, which should **fail**.



Search examples

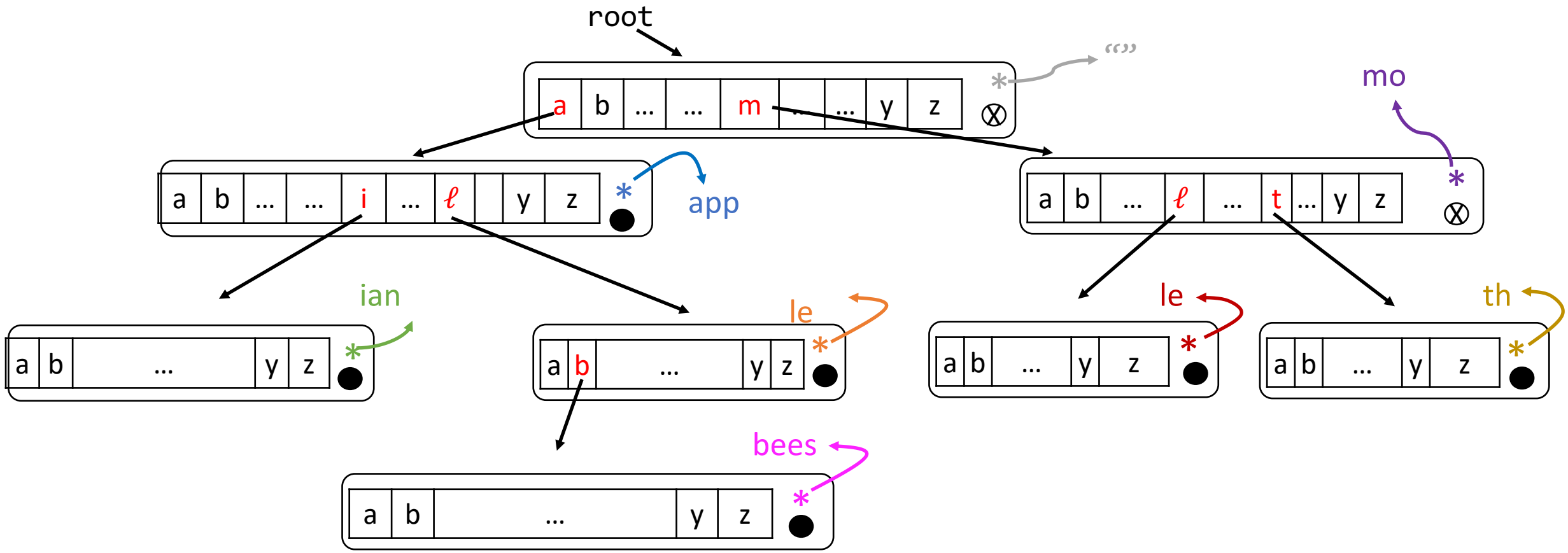
- Search for **“molecule”**, which should **fail**.



Length of remaining suffix larger than length of `n.keyRef:` we know for **sure** that the key is not in the trie ☹ Search fails.

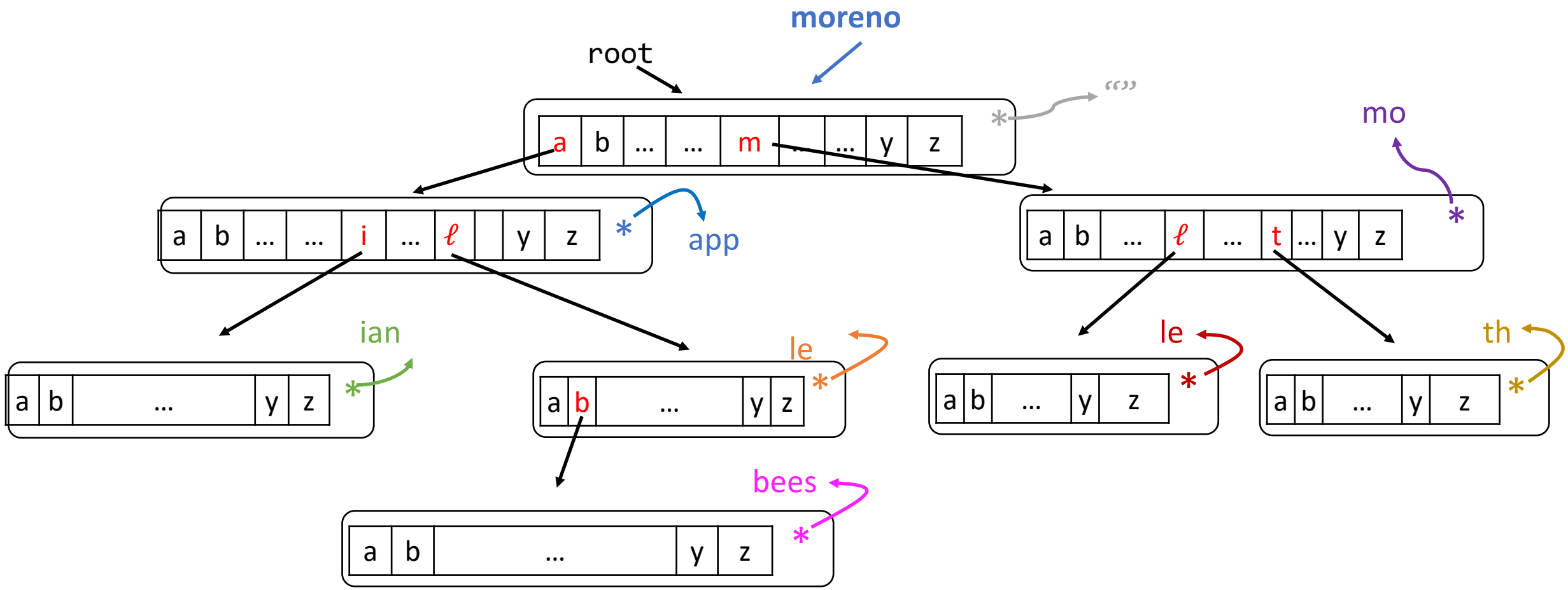
Search examples

- Search for **“moreno”**, which should **fail**.



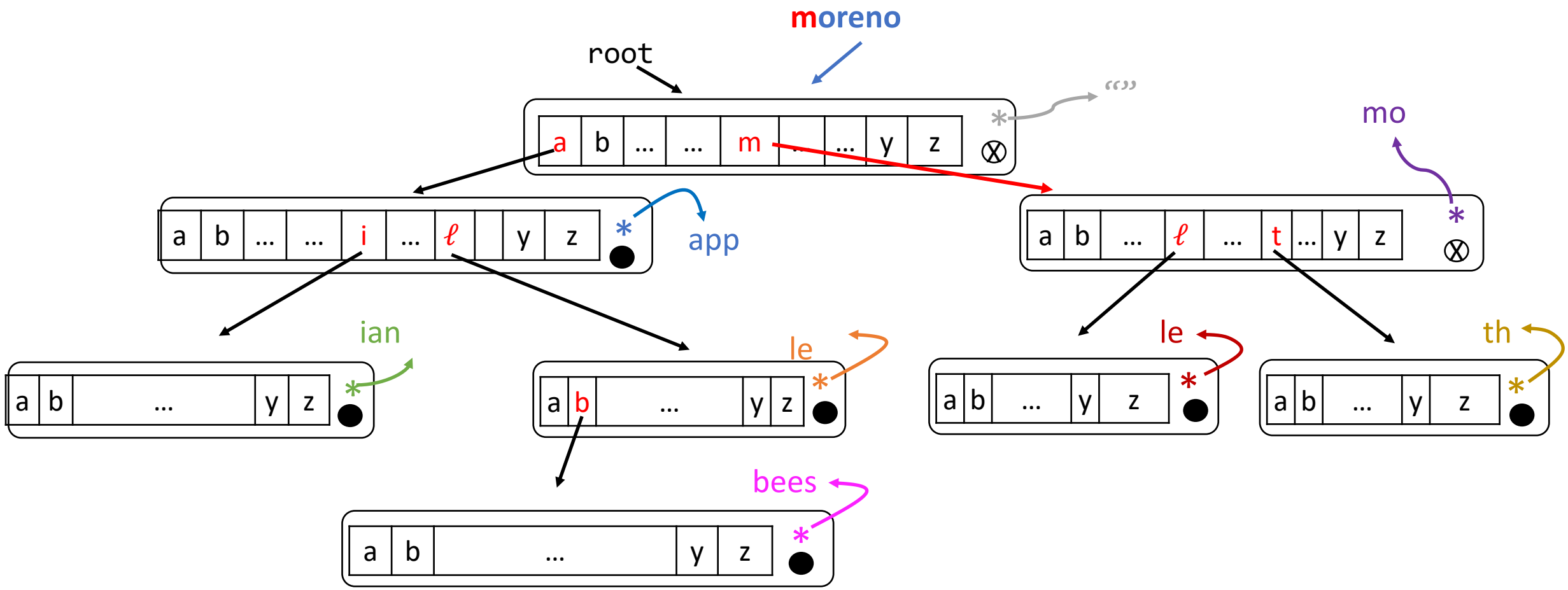
Search examples

- Search for **“moreno”**, which should **fail**.



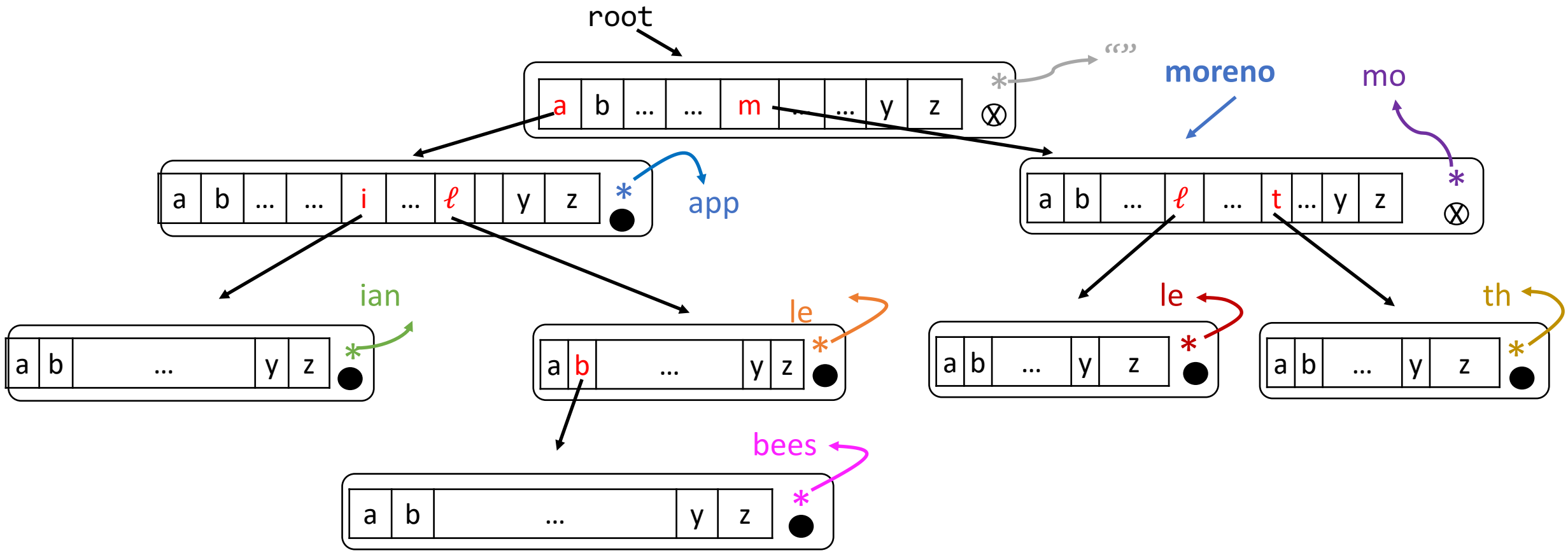
Search examples

- Search for **“moreno”**, which should **fail**.



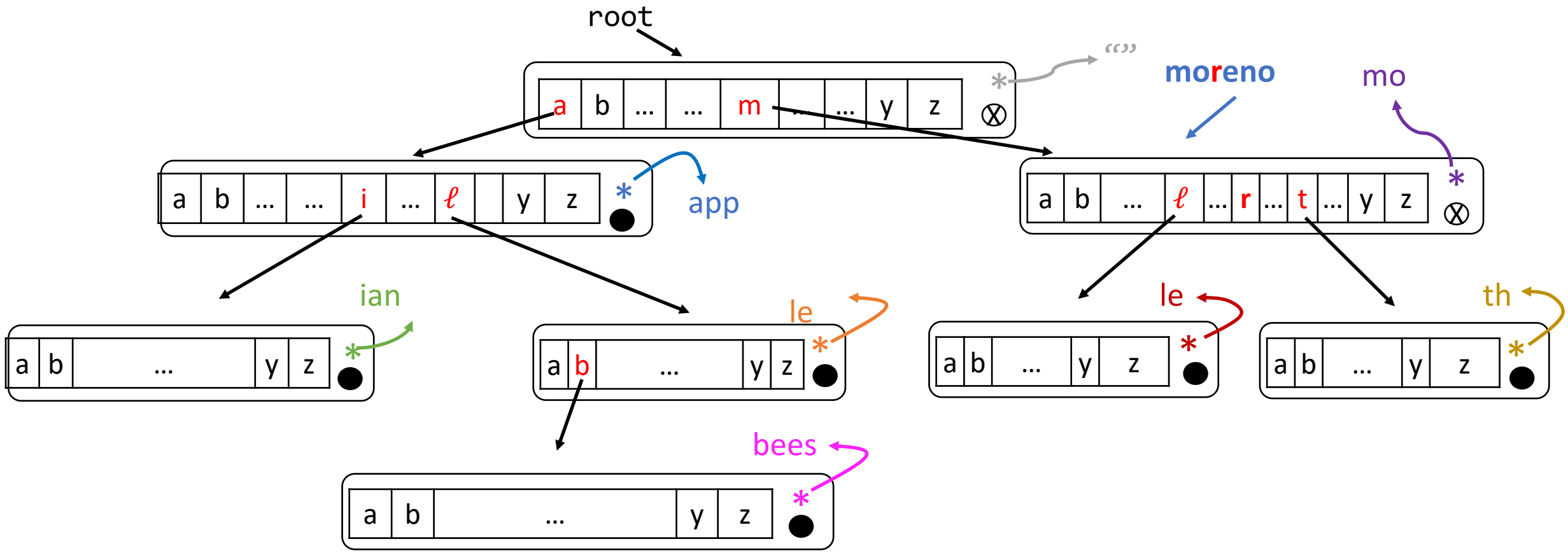
Search examples

- Search for **“moreno”**, which should **fail**.



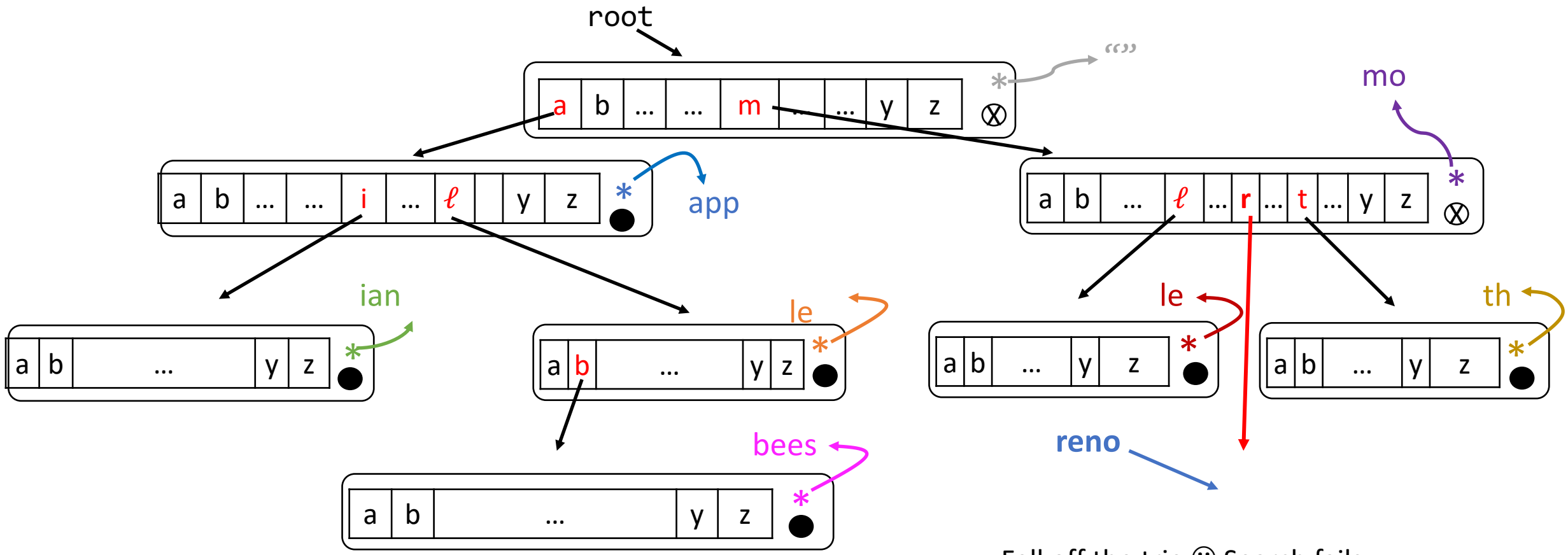
Search examples

- Search for **“moreno”**, which should **fail**.



Search examples

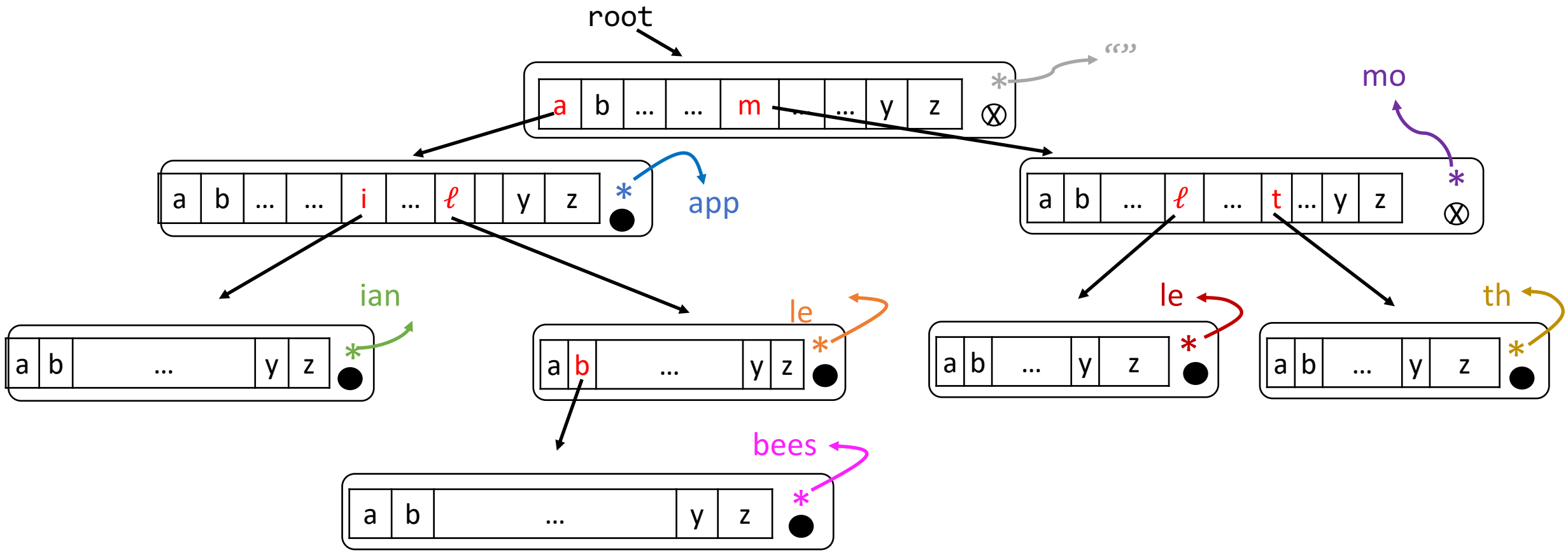
- Search for **“moreno”**, which should **fail**.



Fell off the trie ☹ Search fails.

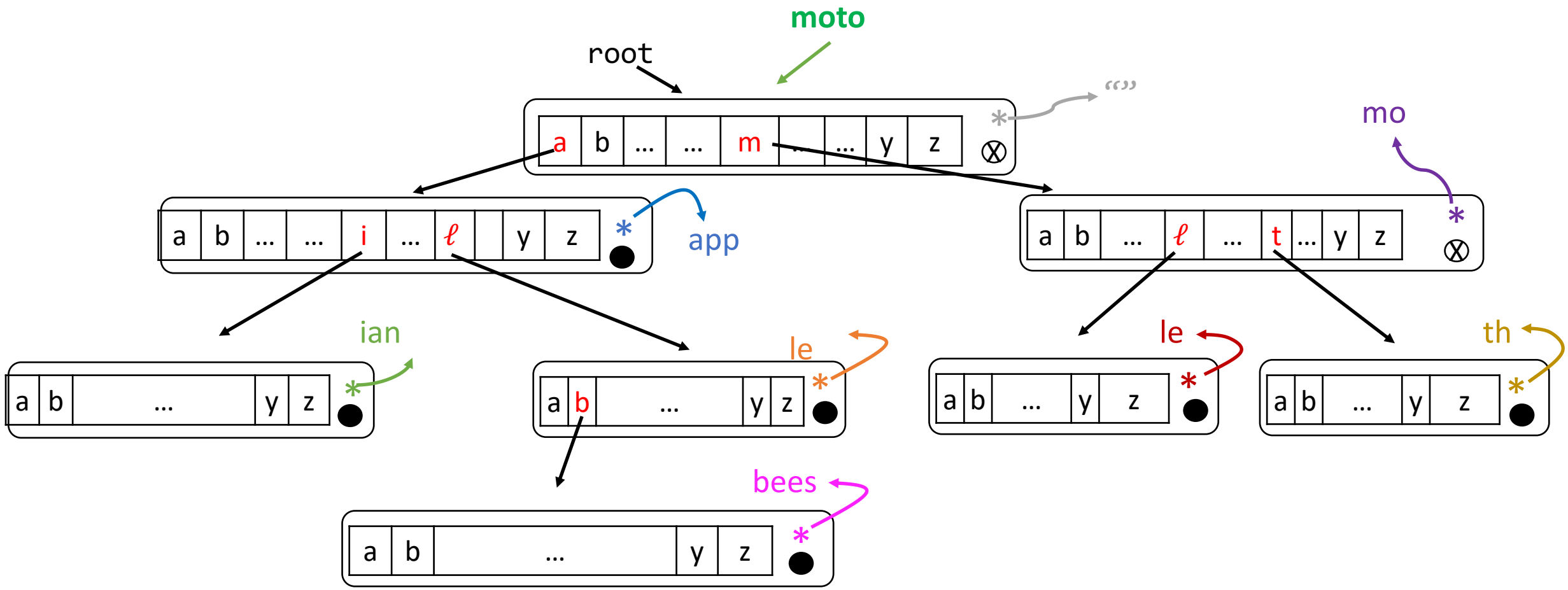
Search examples

- Search for **“moto”**, which should **fail**.



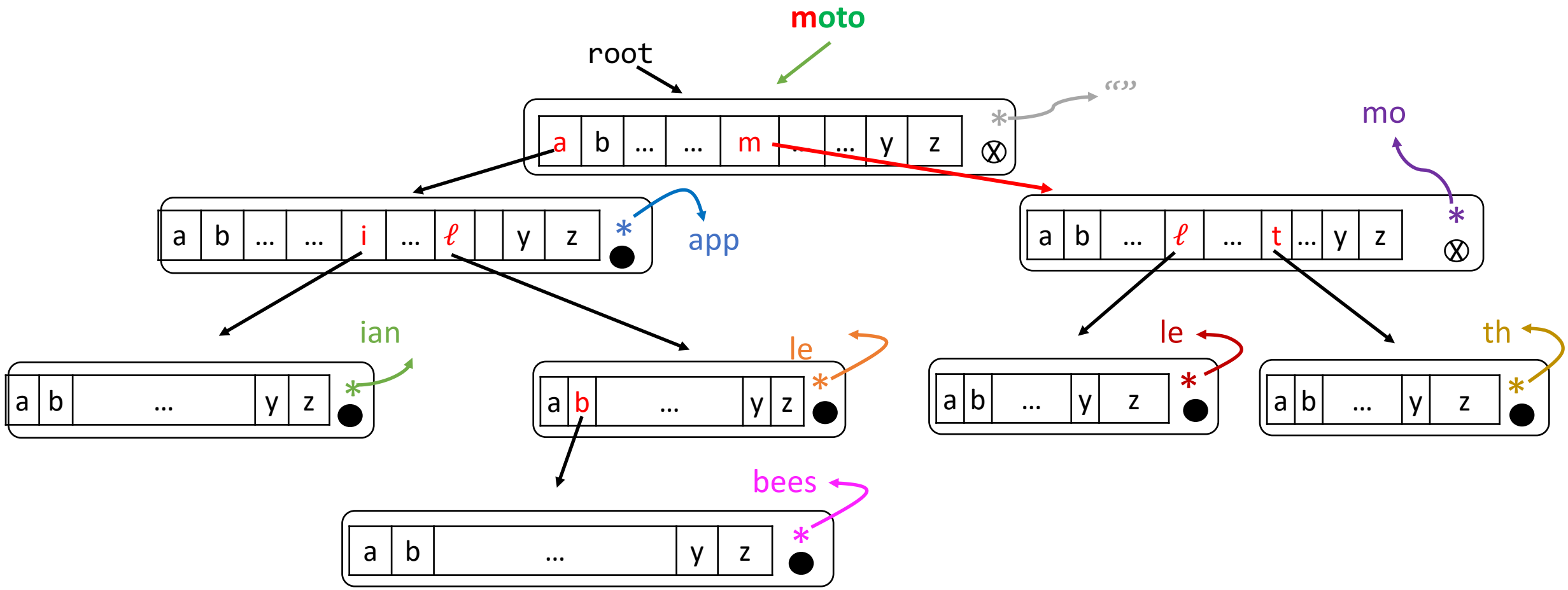
Search examples

- Search for **“moto”**, which should **fail**.



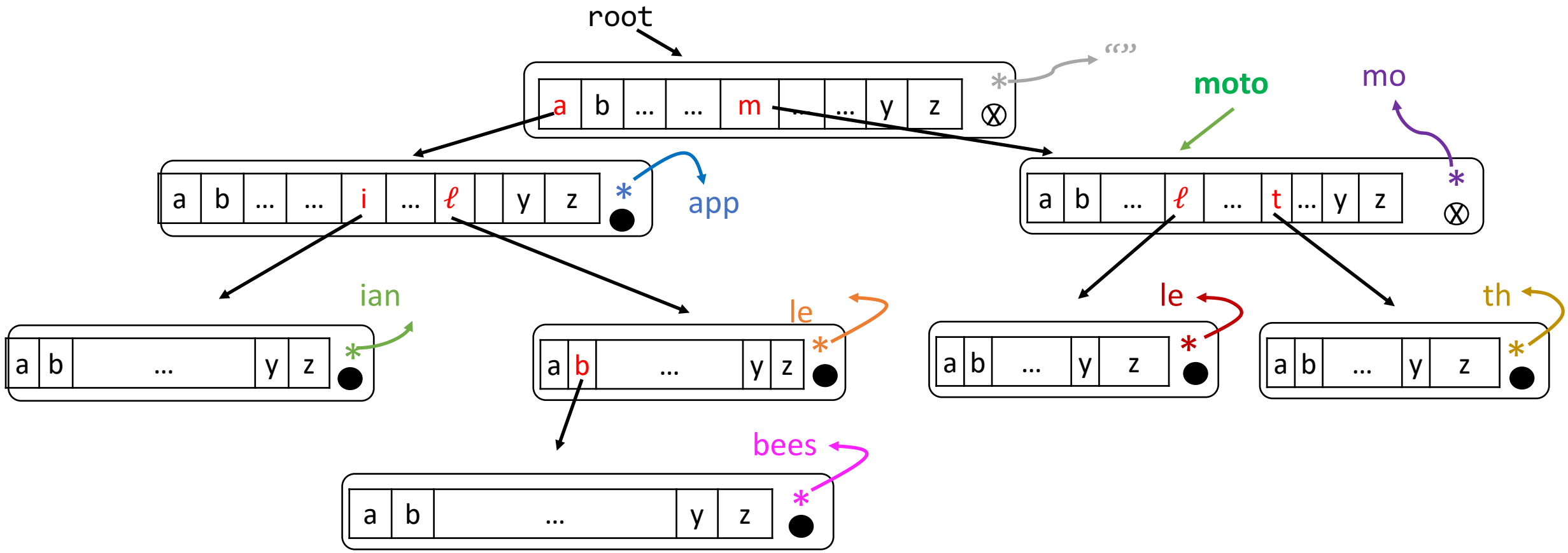
Search examples

- Search for **“moto”**, which should **fail**.



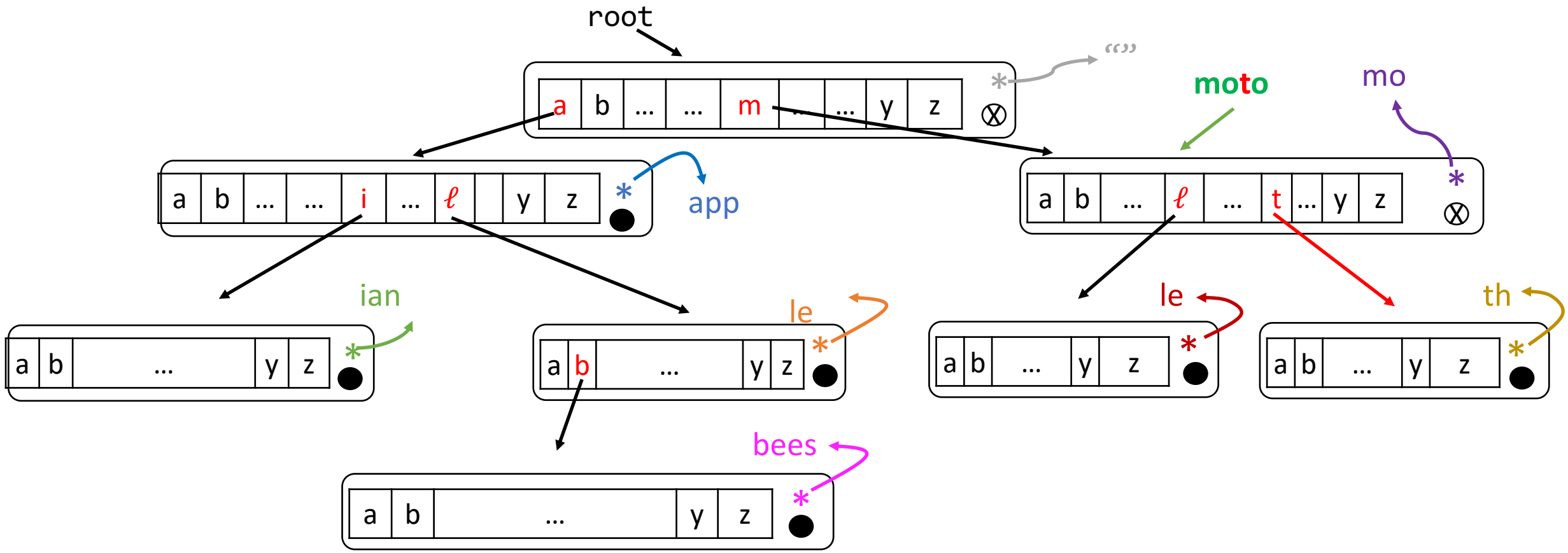
Search examples

- Search for **“moto”**, which should **fail**.



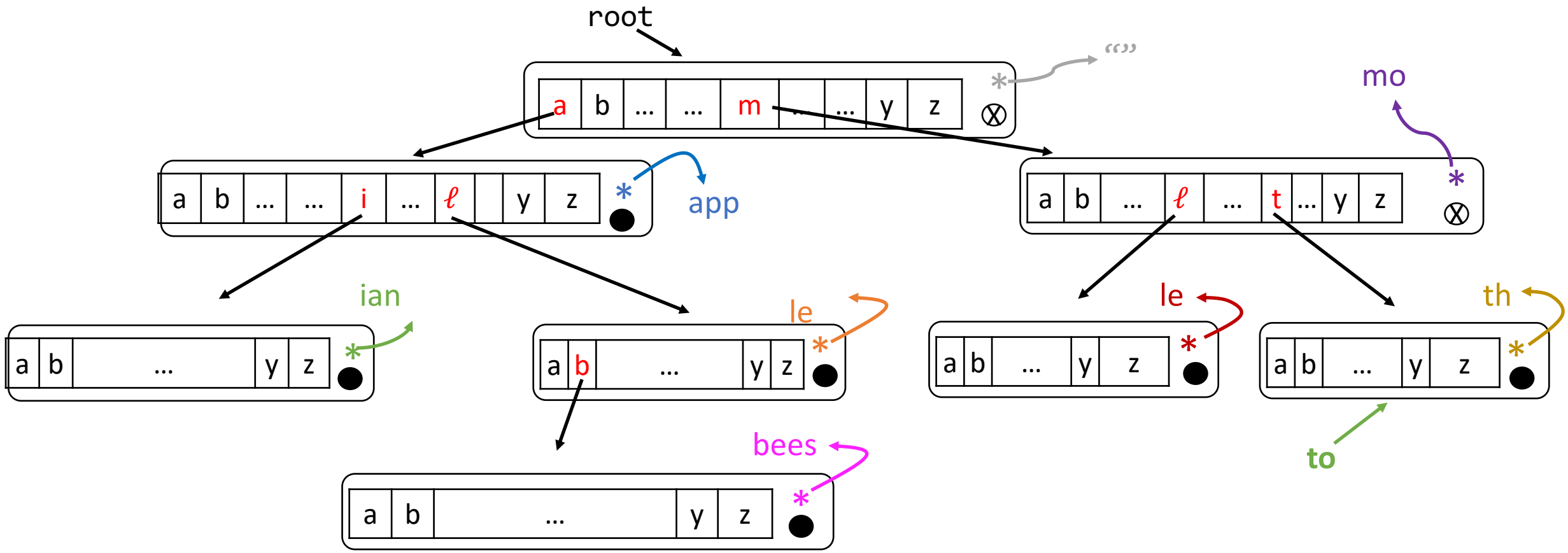
Search examples

- Search for **“moto”**, which should **fail**.



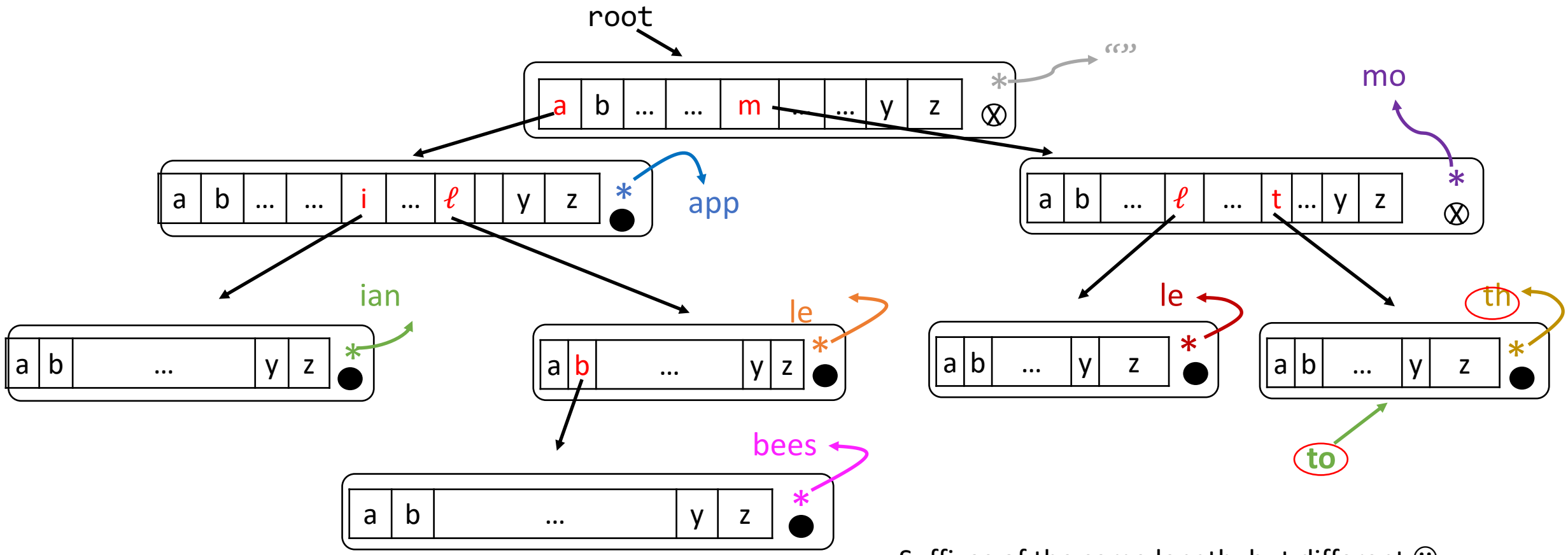
Search examples

- Search for **“moto”**, which should **fail**.



Search examples

- Search for **“moto”**, which should **fail**.

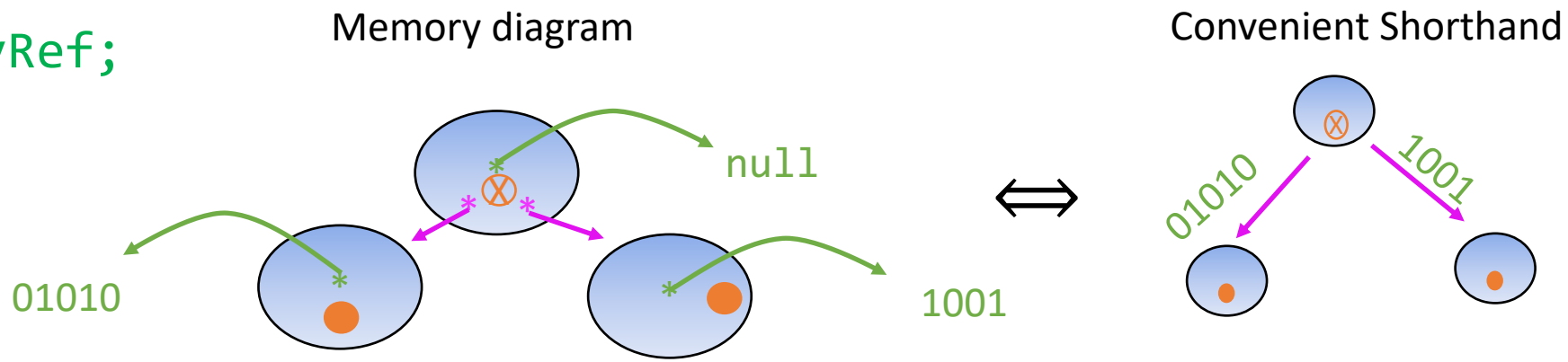


Suffixes of the same length, but different ☹
Search fails.

A simplified Node structure for BPTs

- Since your project involves **B**inary **P**atricia **T**ries, which use the minimal alphabet {0,1}, we can simplify our node structure as follows:

```
private class BPTNode{
        BPTNode left, right; // Could also have an array of size 2
        boolean isKey;
        String keyRef;
}
```

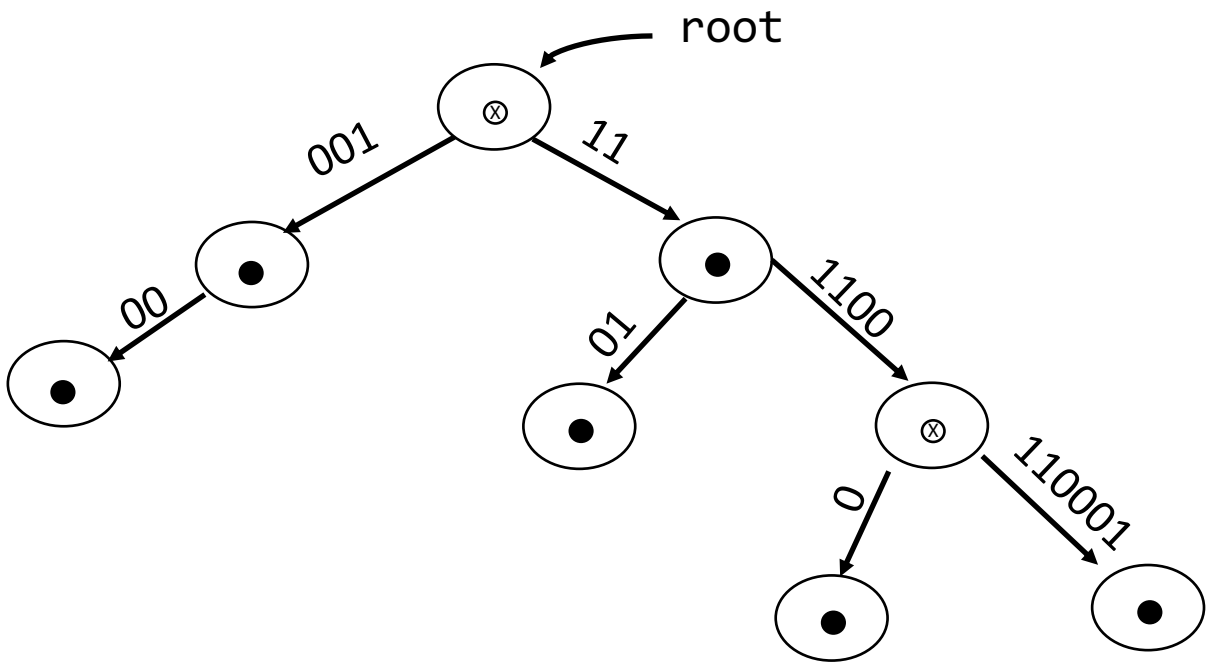


Insertion

- Cases of insertion:
  1. If node examined is **null**, allocate new node with input key and set bit.
  2. If node examined contains exactly the input string, simply set bit to 1 (it might already be 1, that is fine – duplicate insertions are trivially handled this way).
  3. If the node's stored string is a (strict) prefix of the given key, consume the prefix and recurse appropriately.
  4. If our key is a (strict) prefix of the node's stored string, make a new parent node containing our key, and change the current node's key to contain the part of the key after the matched prefix. (2^{nd} most complex case)
  5. If the node's stored suffix is exactly as long as our input key but not equal to it, or if the stored suffix is longer than our input key but the key is **not** its prefix, we will split the node into three (most complex case):
     a) A parent node that contains the longest prefix of (input key, stored suffix), with unset bit.
     b) A node that branches from the parent at the input key's character that is indicated **exactly** by the length of that longest prefix, and which contains the new key (set bit).
     c) Another node with the same property, but which contains the old key.
- Some examples with Binary Patricia Tries follow.

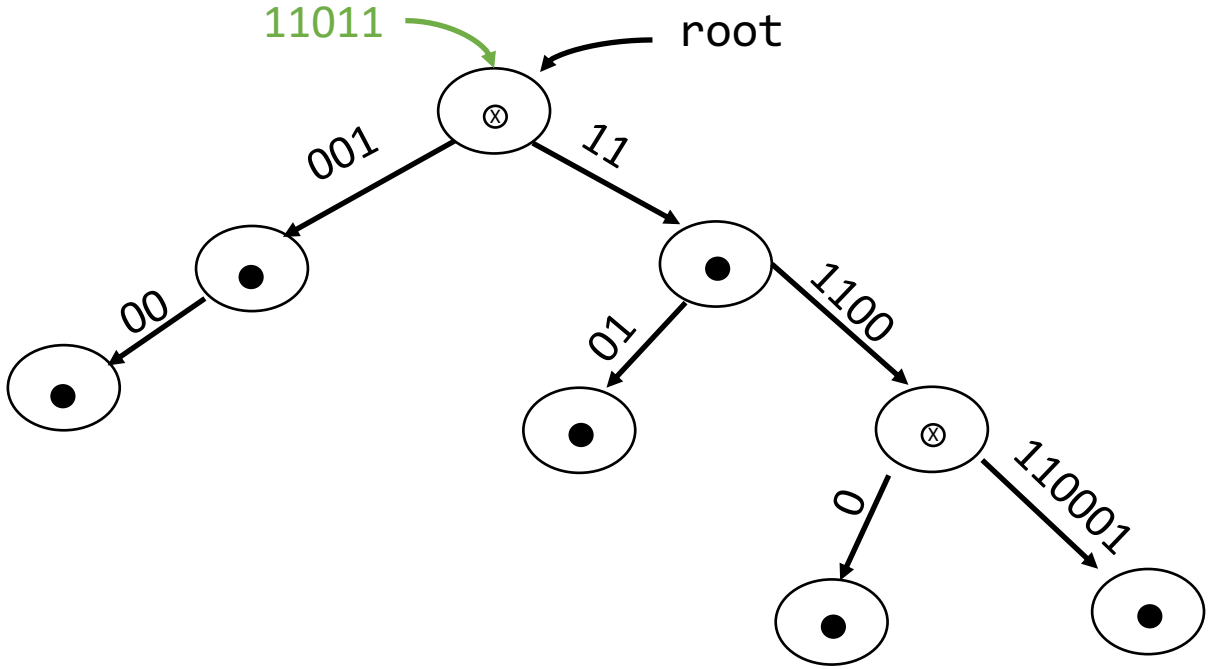
Insertion case #1: Allocate new node

Insert 11011



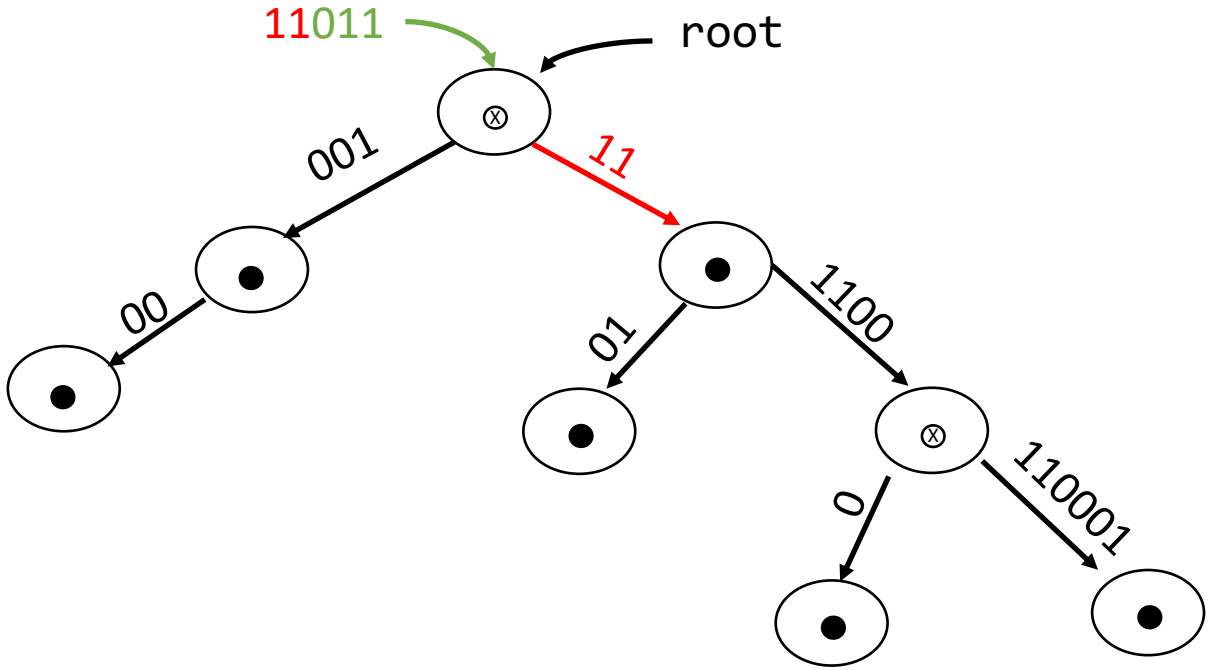
Insertion case #1: Allocate new node

Insert 11011



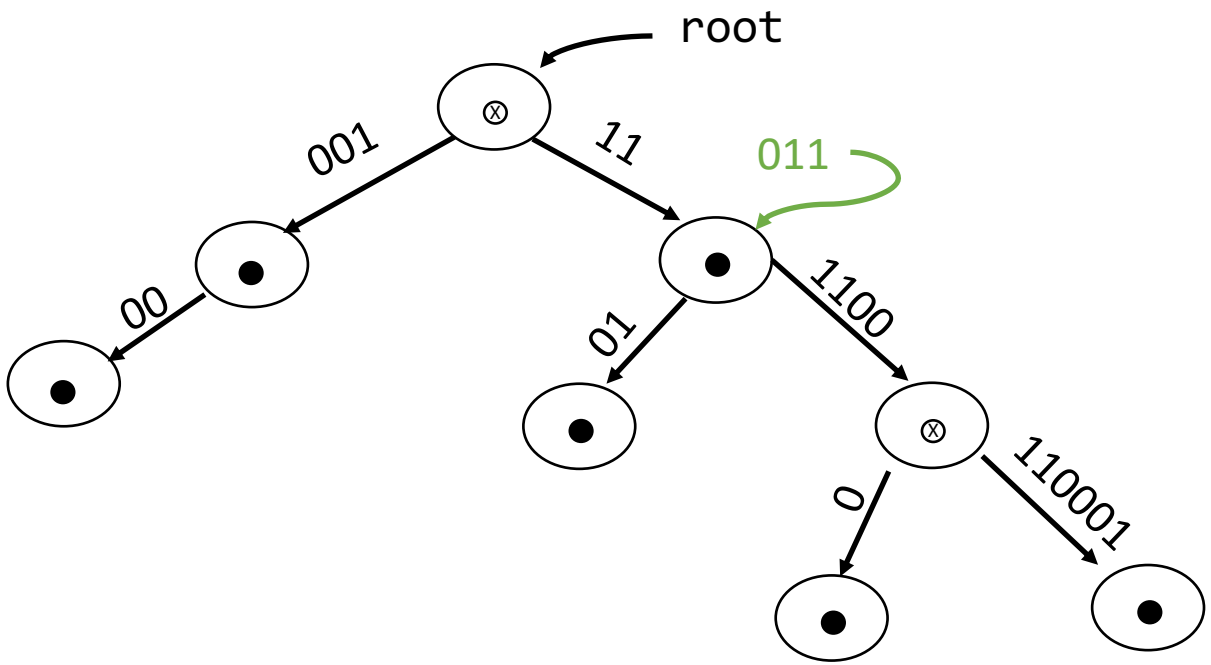
Insertion case #1: Allocate new node

Insert 11011



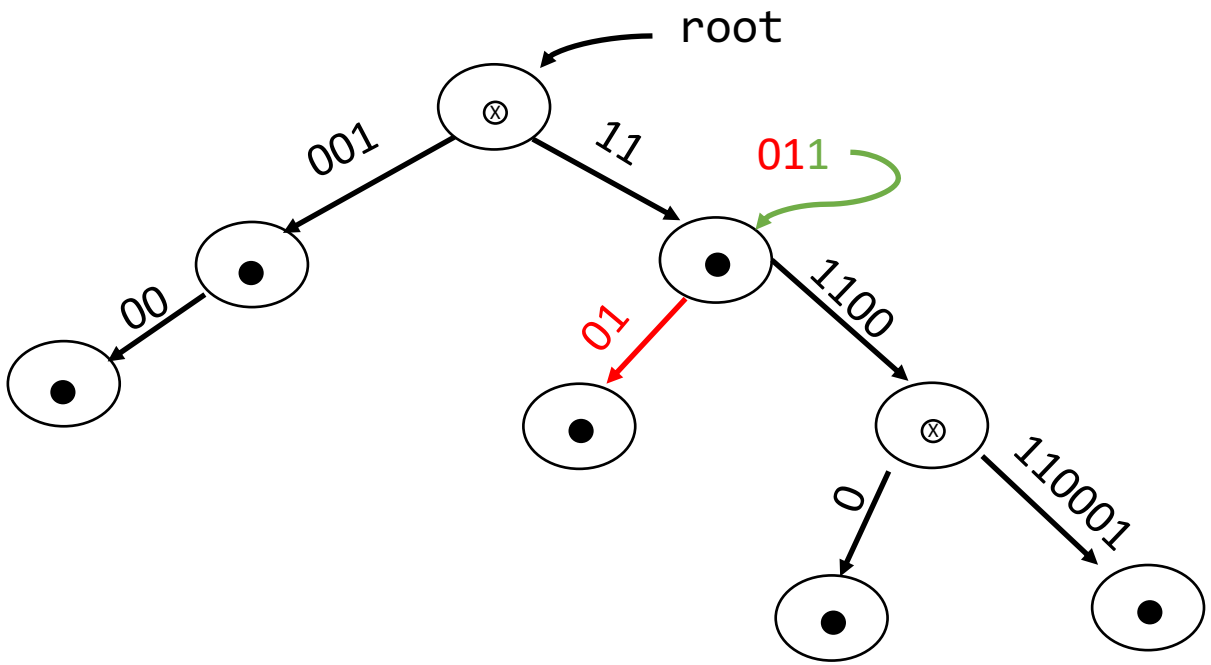
Insertion case #1: Allocate new node

Insert 11011



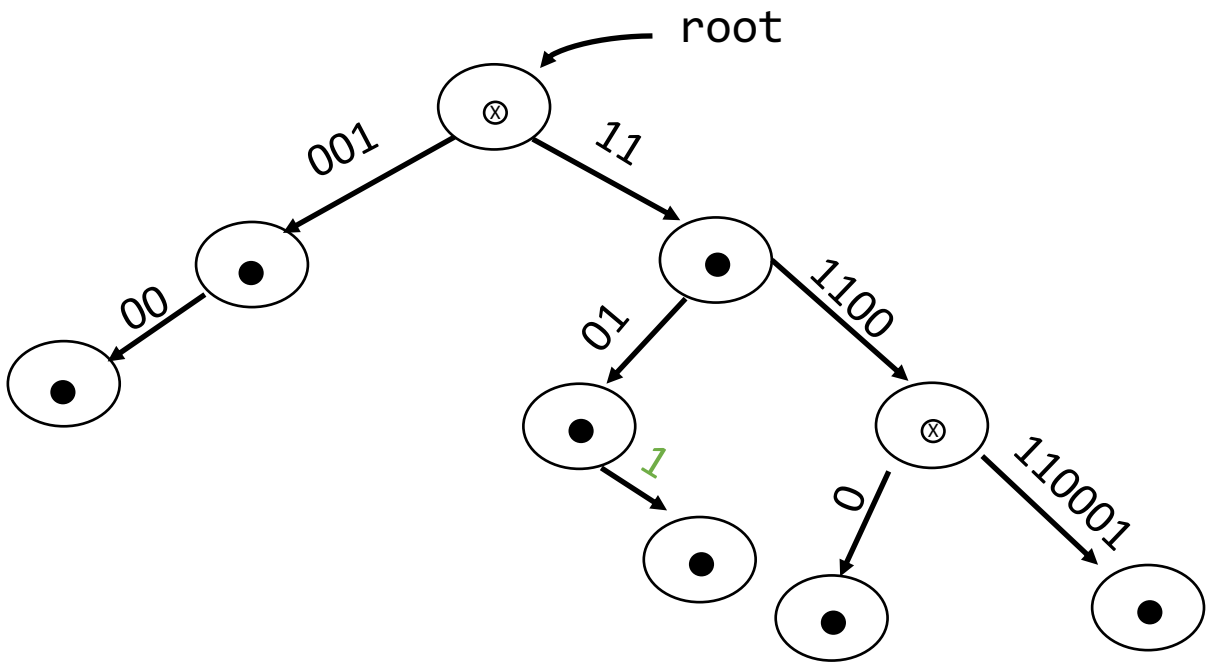
Insertion case #1: Allocate new node

Insert 11011



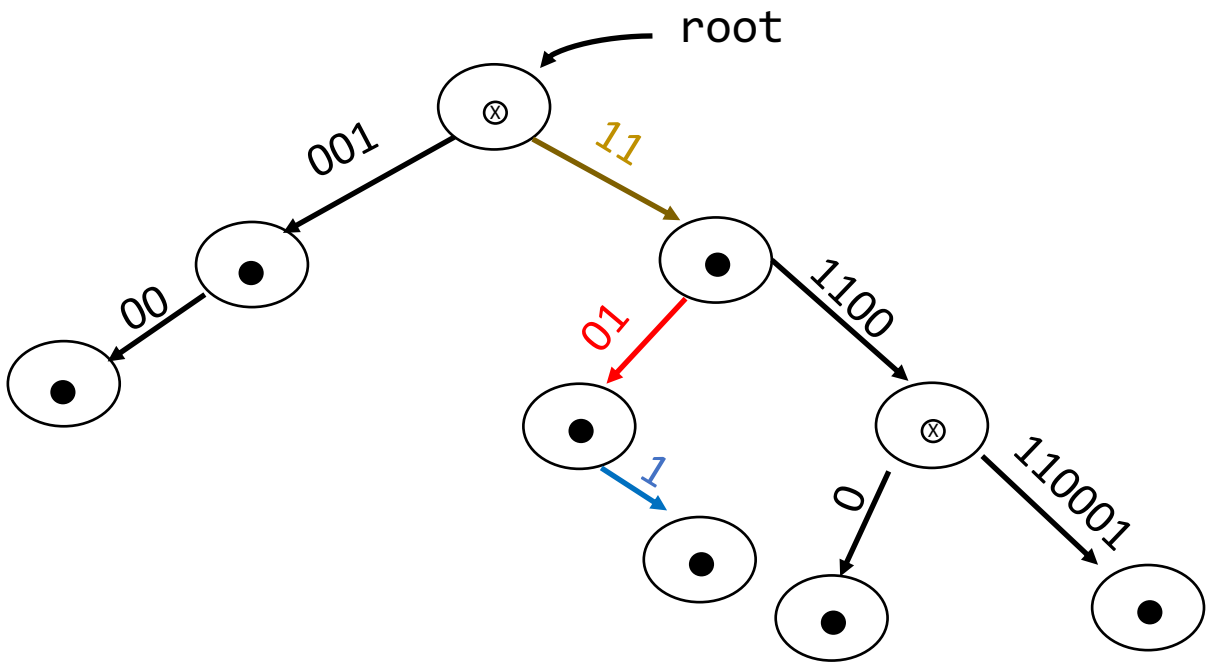
Insertion case #1: Allocate new node

Insert 11011



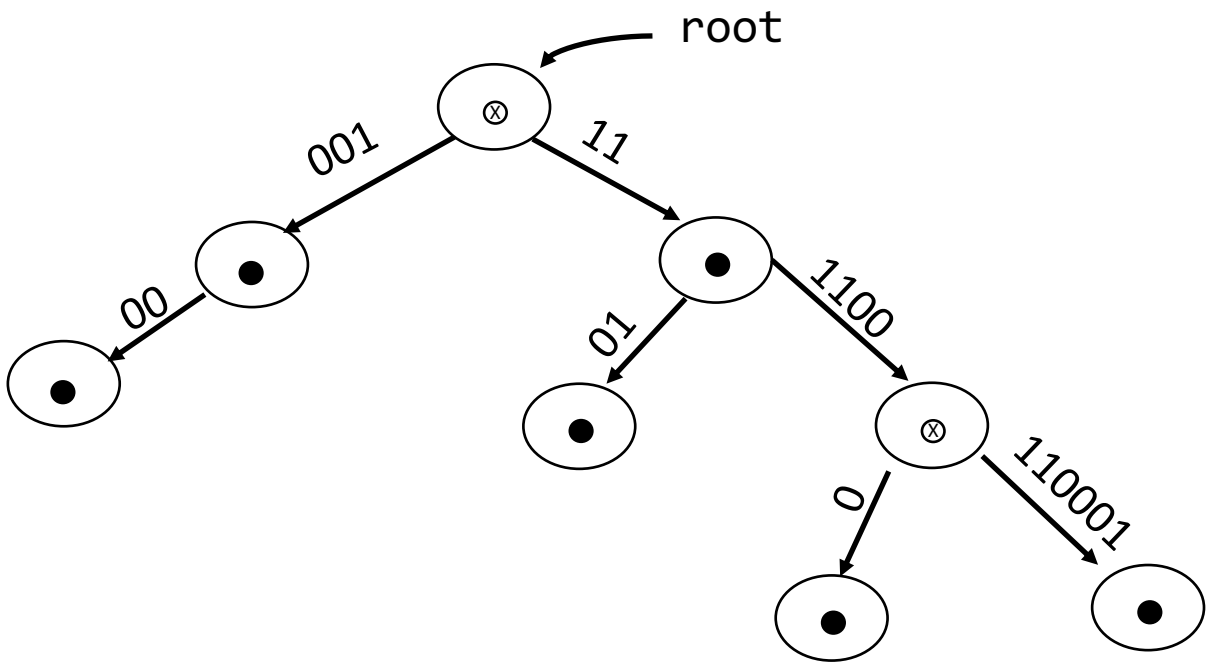
Insertion case #1: Allocate new node

Insert 11011



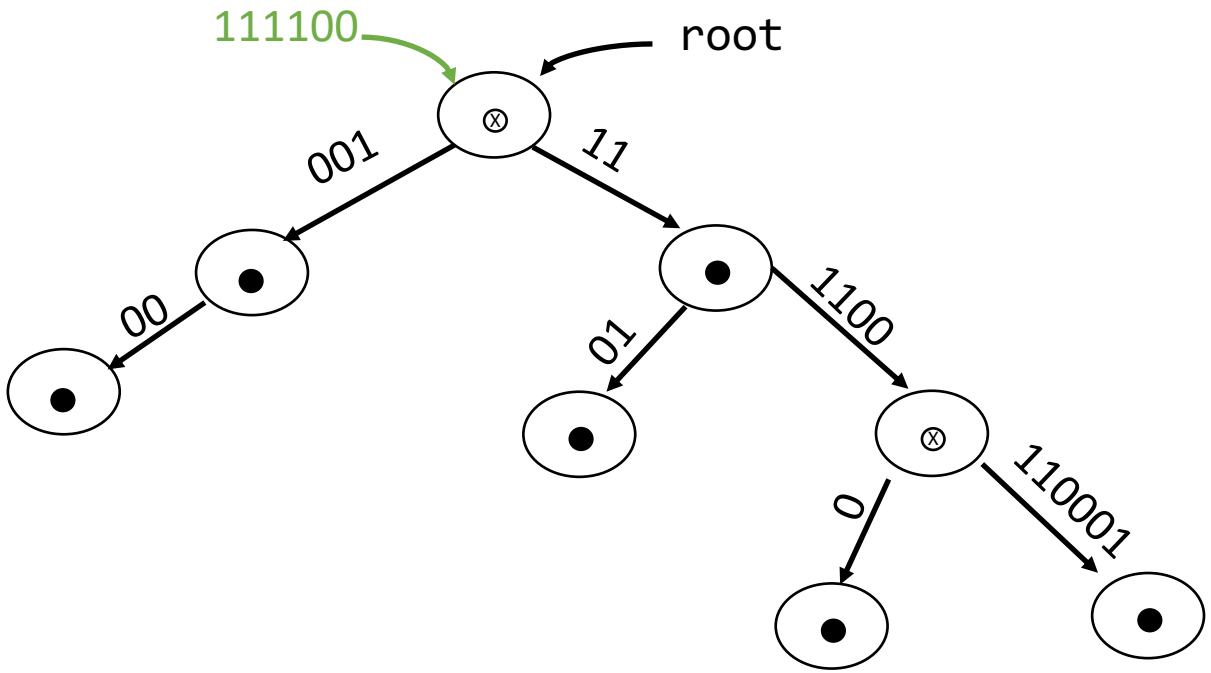
Insertion case #2: Simply set the bit

Insert 111100



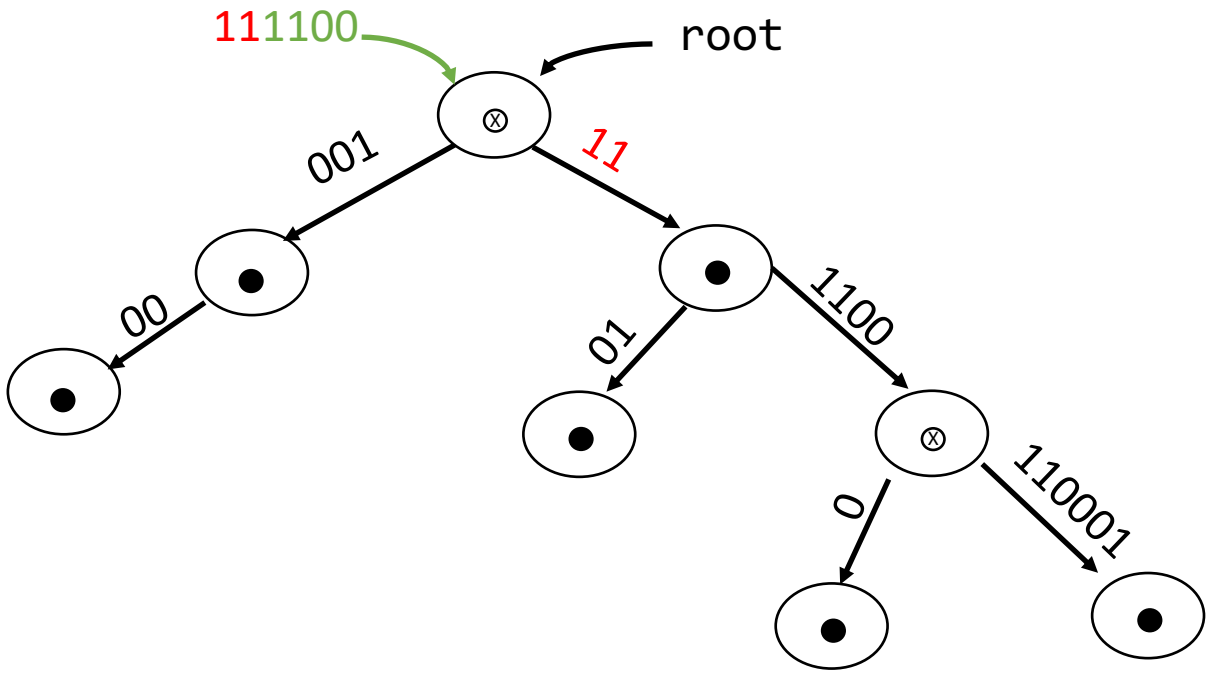
Insertion case #2: Simply set the bit

Insert 111100



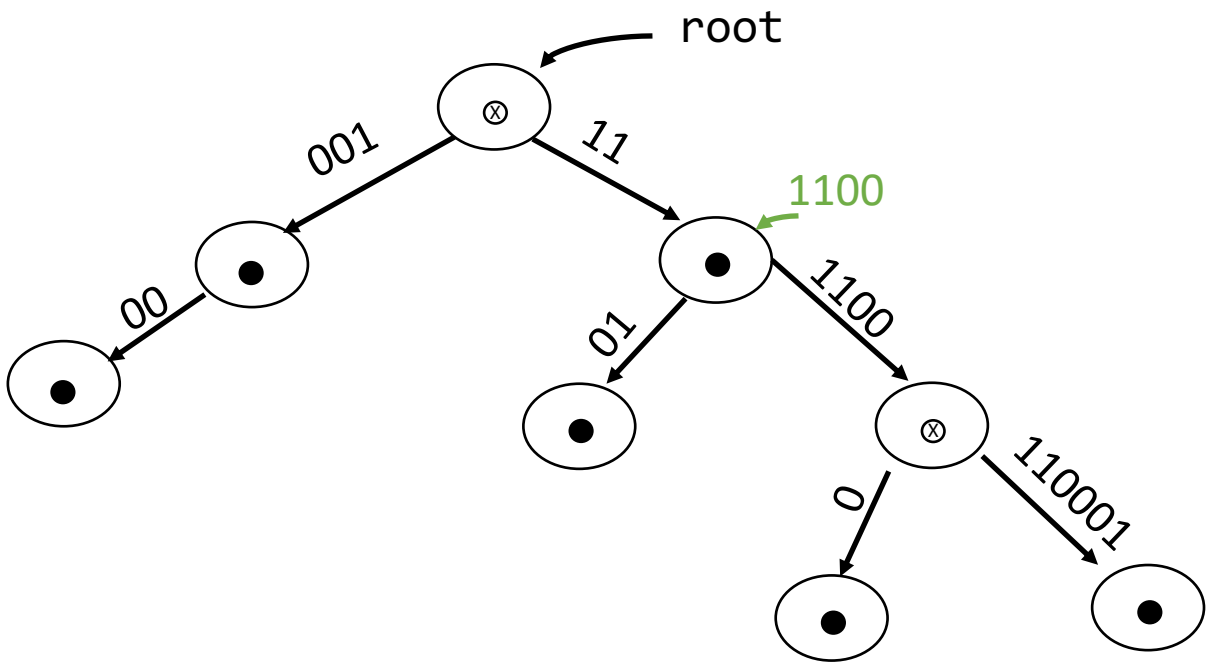
Insertion case #2: Simply set the bit

Insert 111100



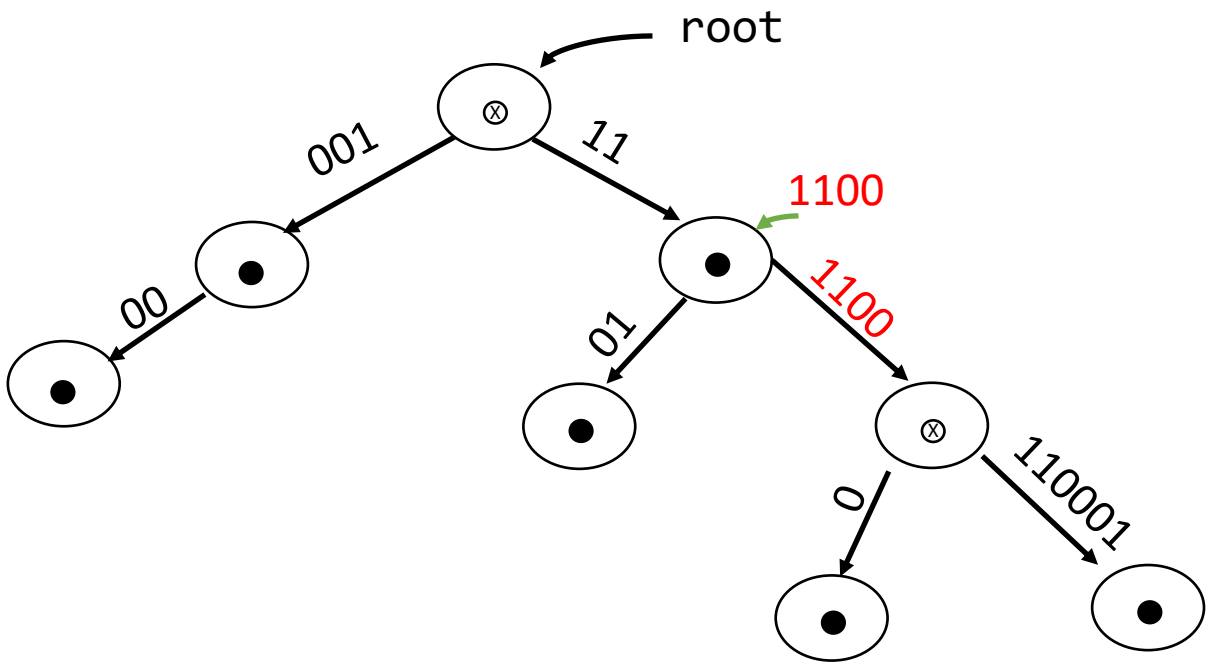
Insertion case #2: Simply set the bit

Insert 111100



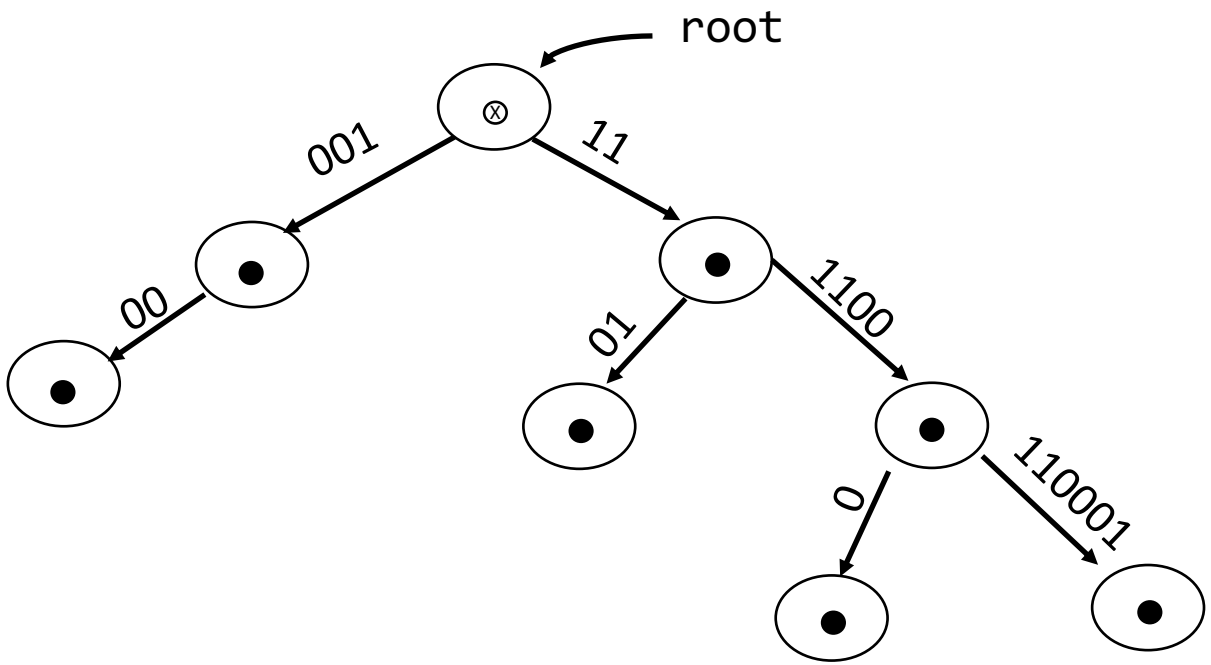
Insertion case #2: Simply set the bit

Insert 111100



Insertion case #2: Simply set the bit

Insert 111100

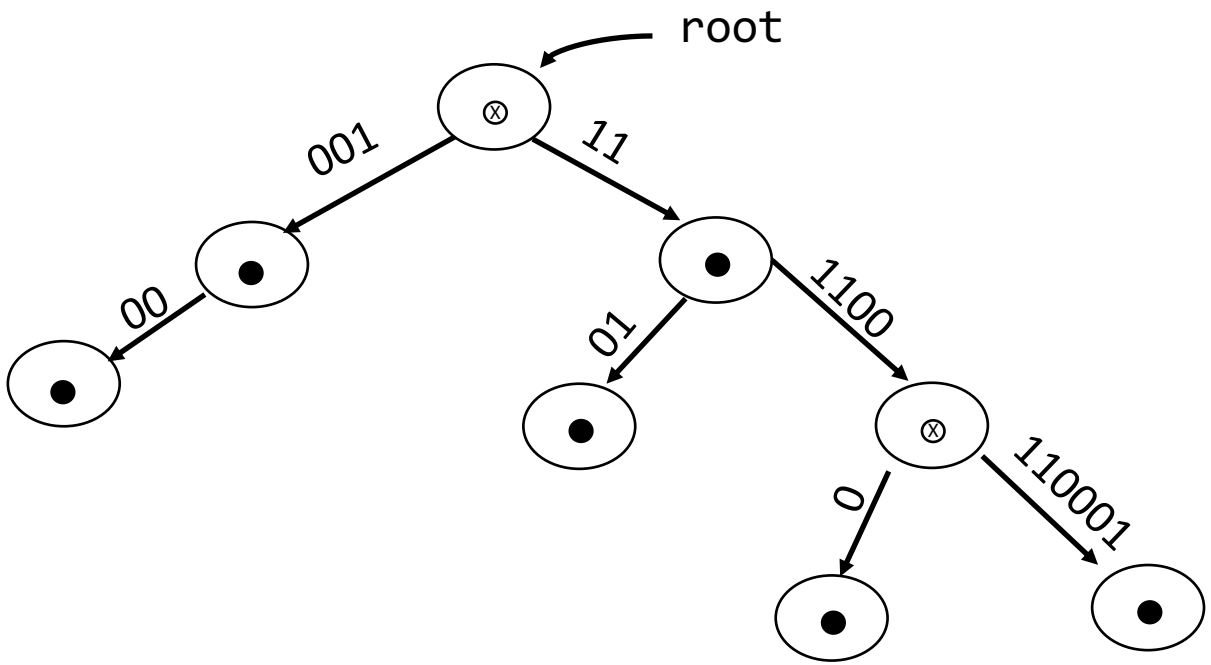


Insertion case #3: Consume and Recurse

- This case is implicit in all others: essentially, whenever I can consume some of the key, I should do so and recurse to the appropriate child based on the next character.

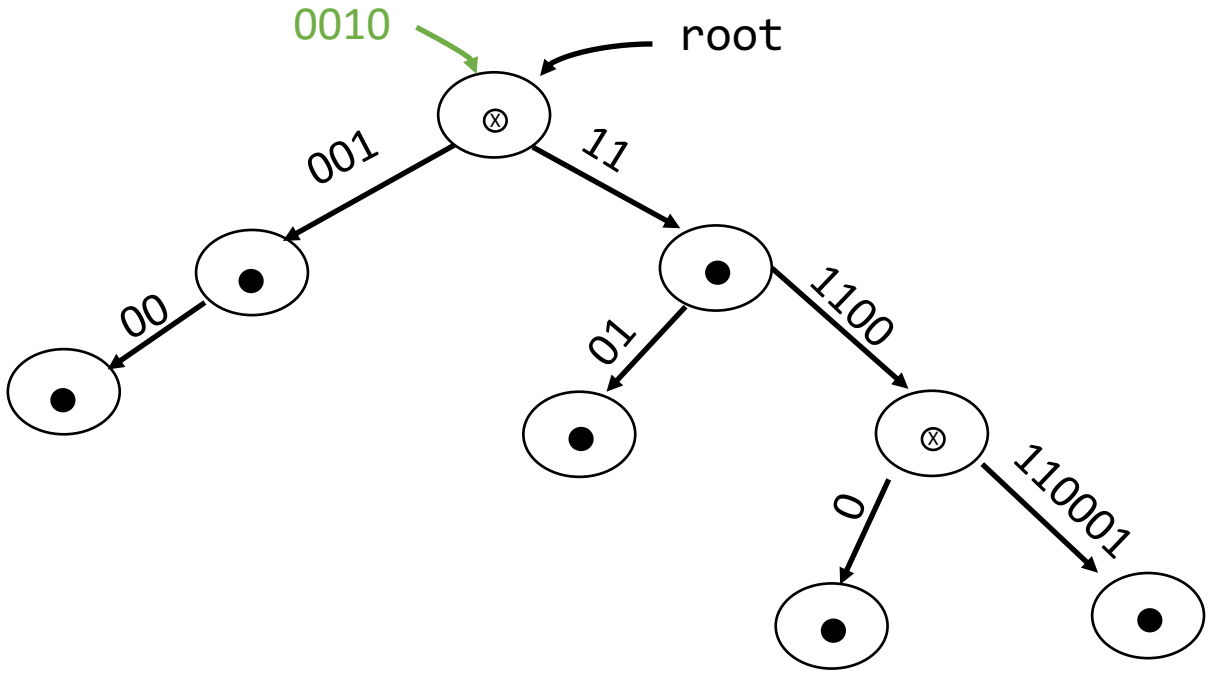
Insertion case #4: Make a new parent

Insert 0010



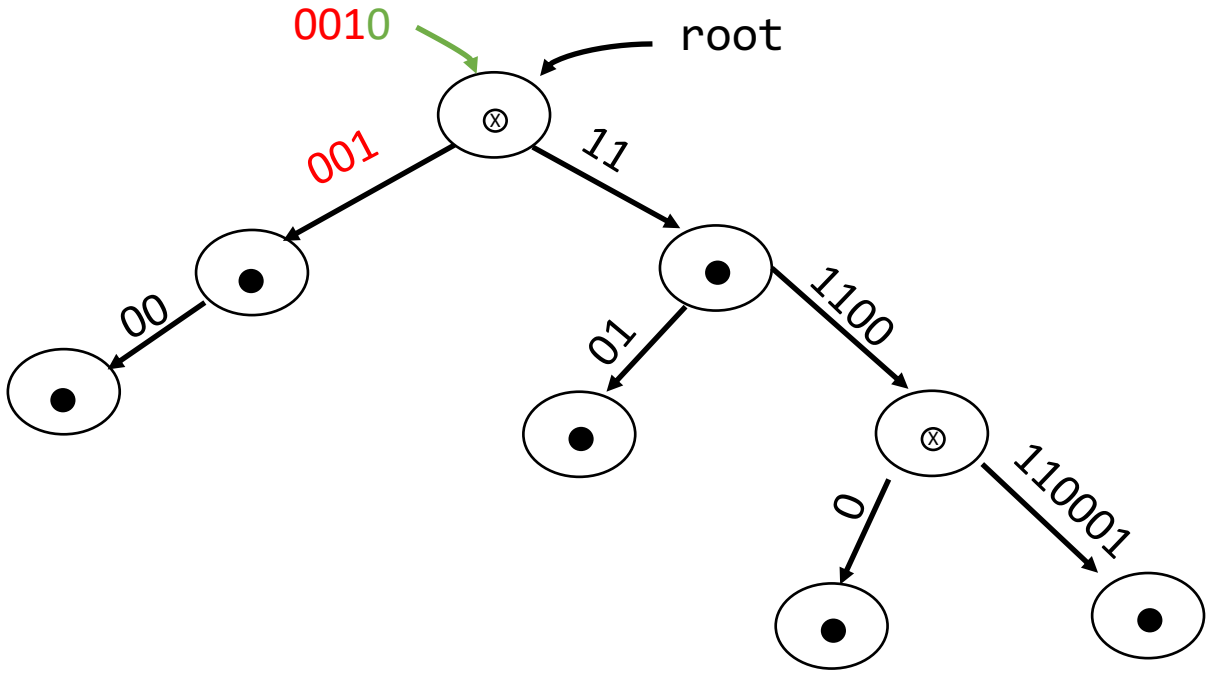
Insertion case #4: Make a new parent

Insert 0010



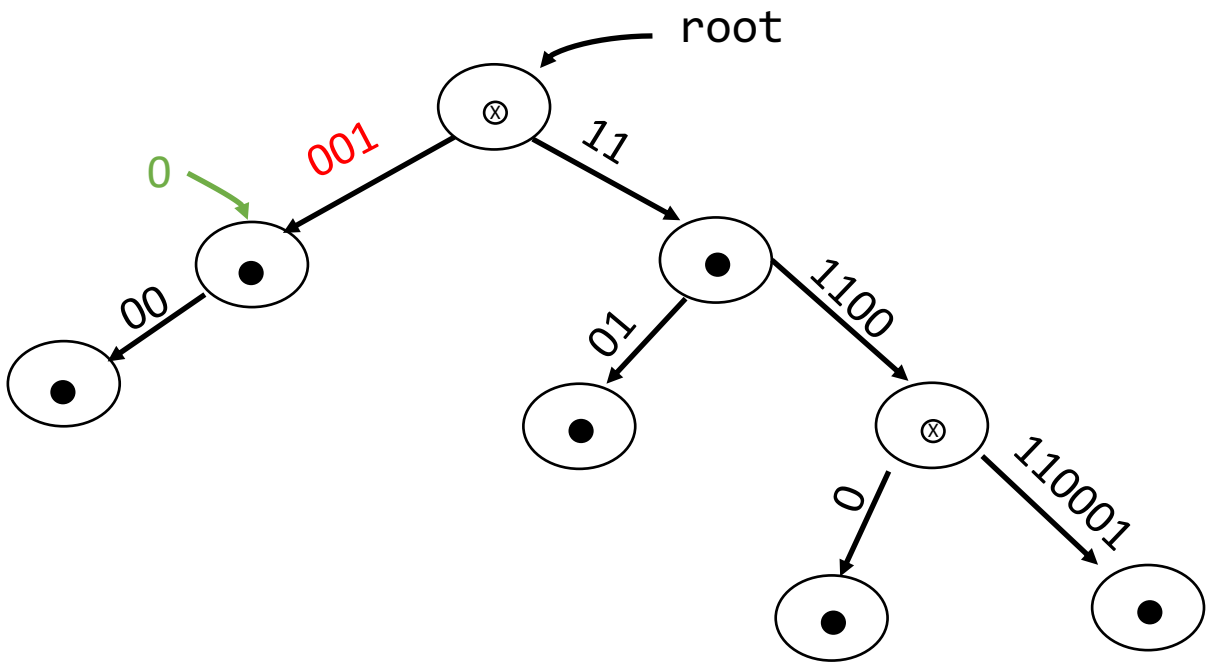
Insertion case #4: Make a new parent

Insert 0010



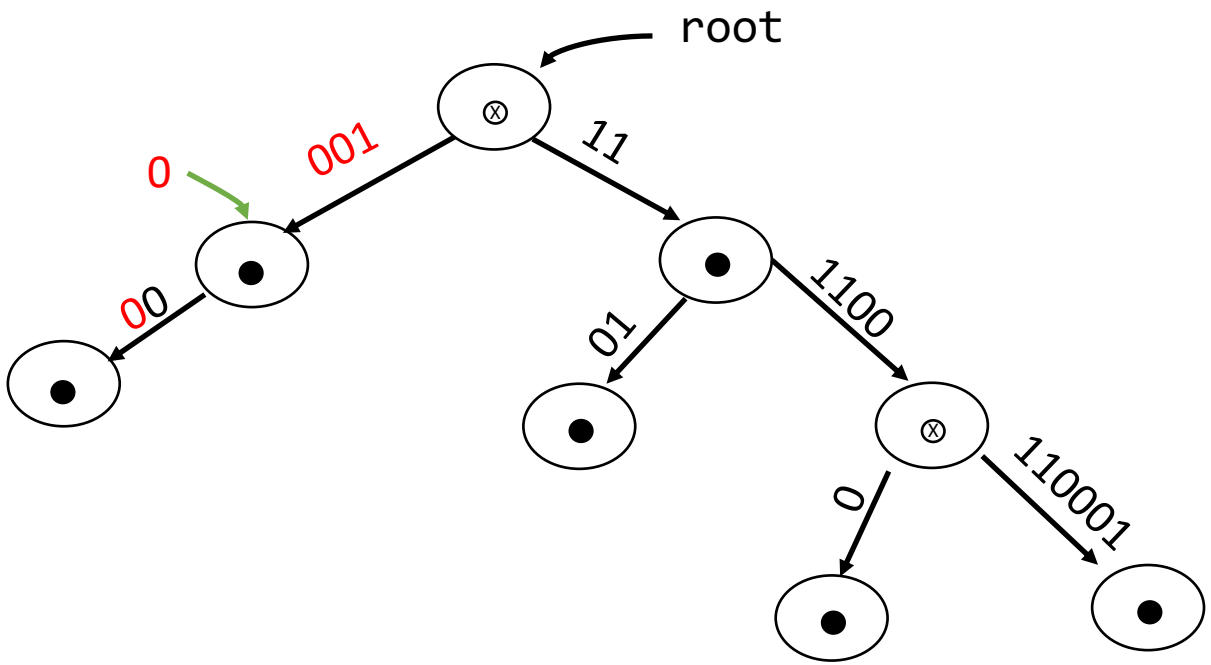
Insertion case #4: Make a new parent

Insert 0010



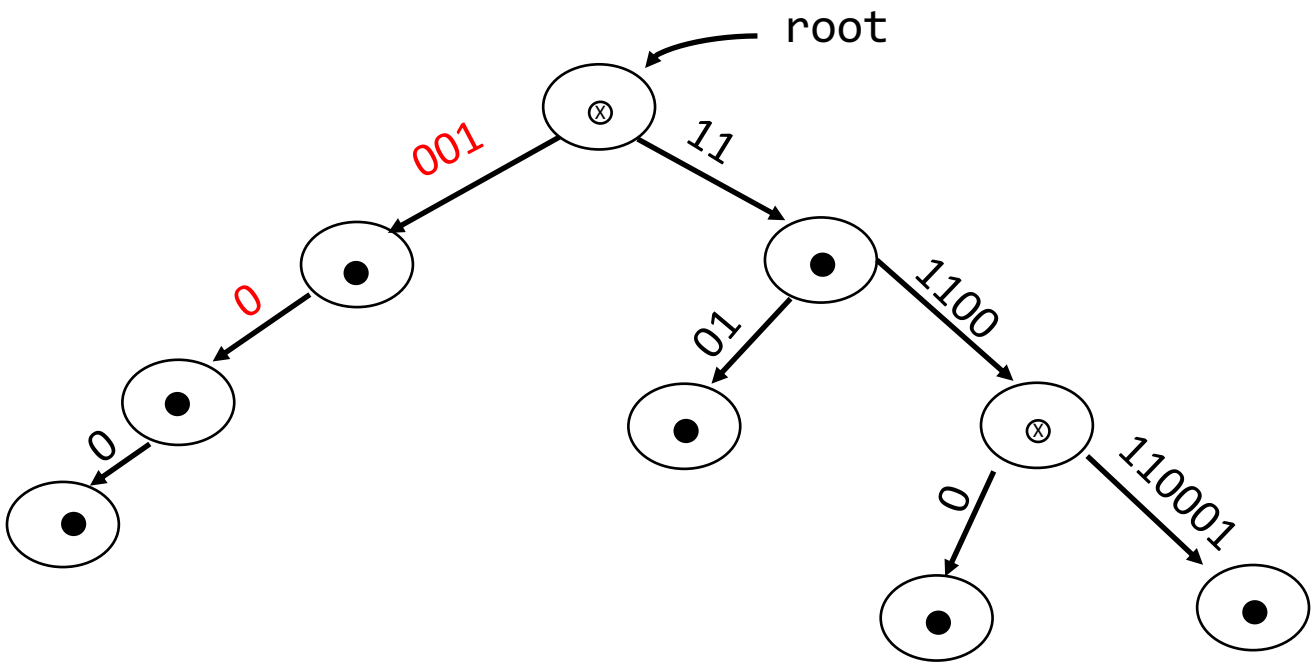
Insertion case #4: Make a new parent

Insert 0010



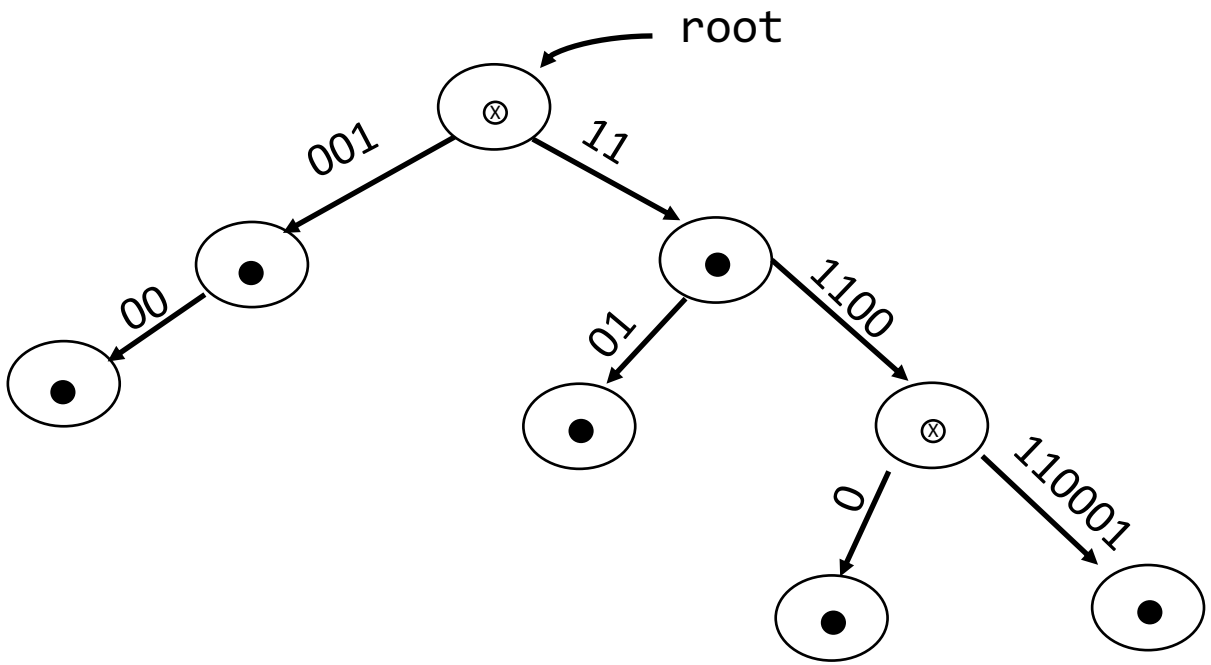
Insertion case #4: Make a new parent

Insert 0010



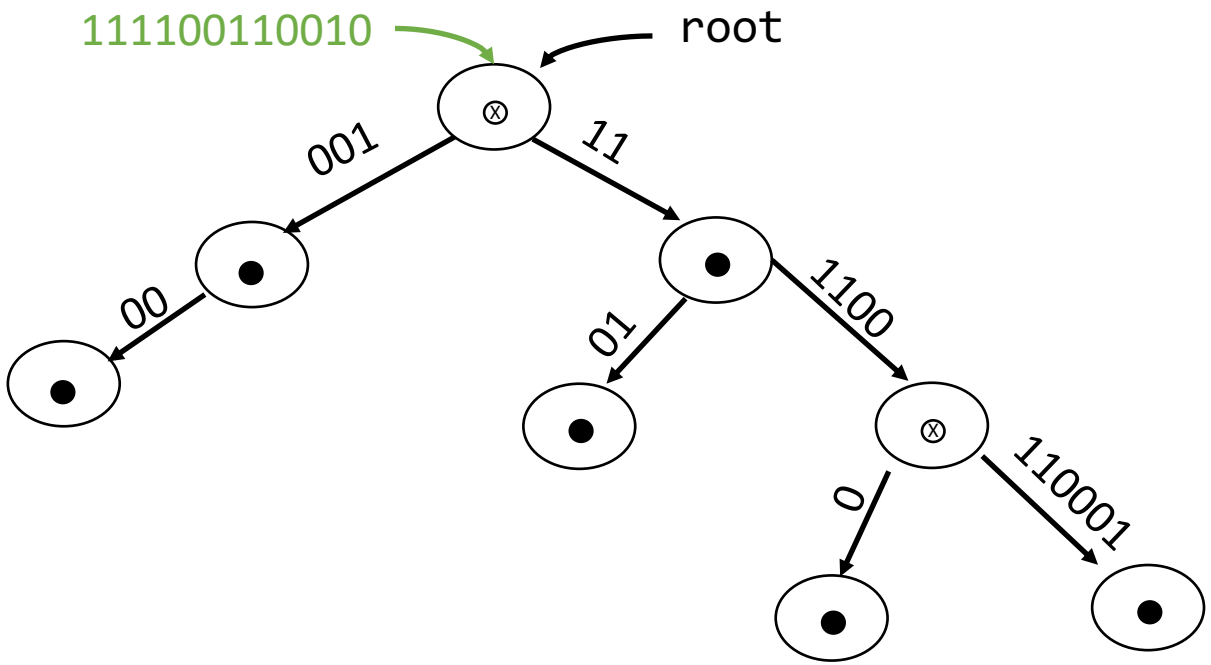
Insertion case #5: Split node

Insert 111100110010



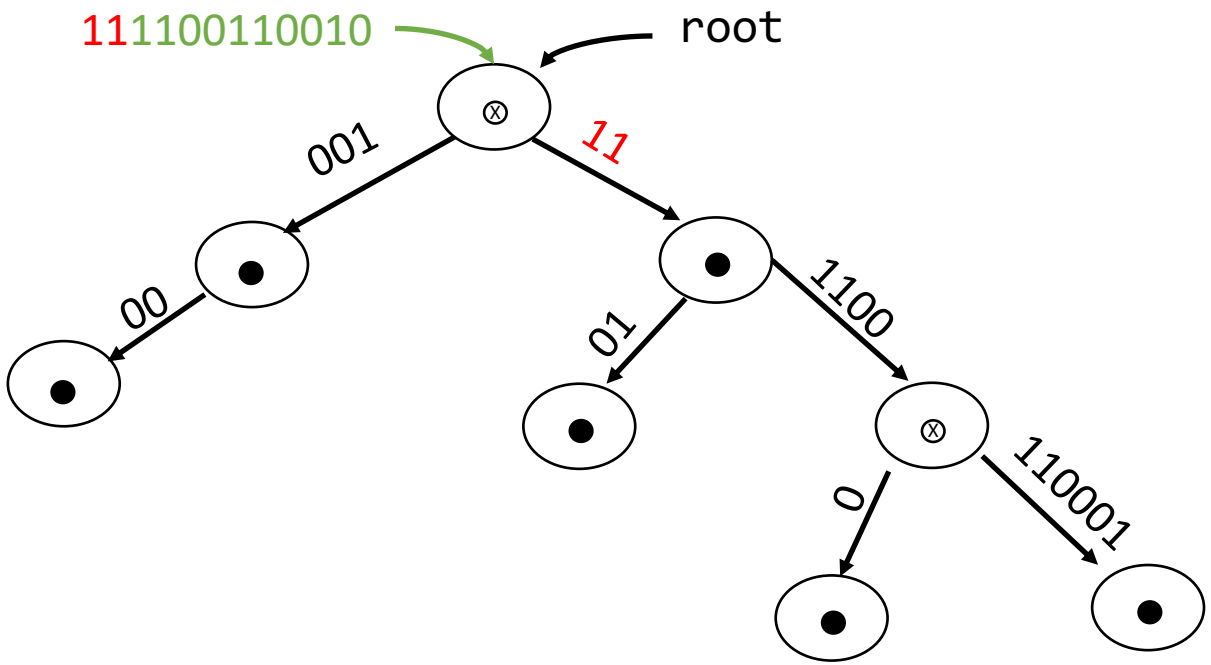
Insertion case #5: Split node

Insert 111100110010



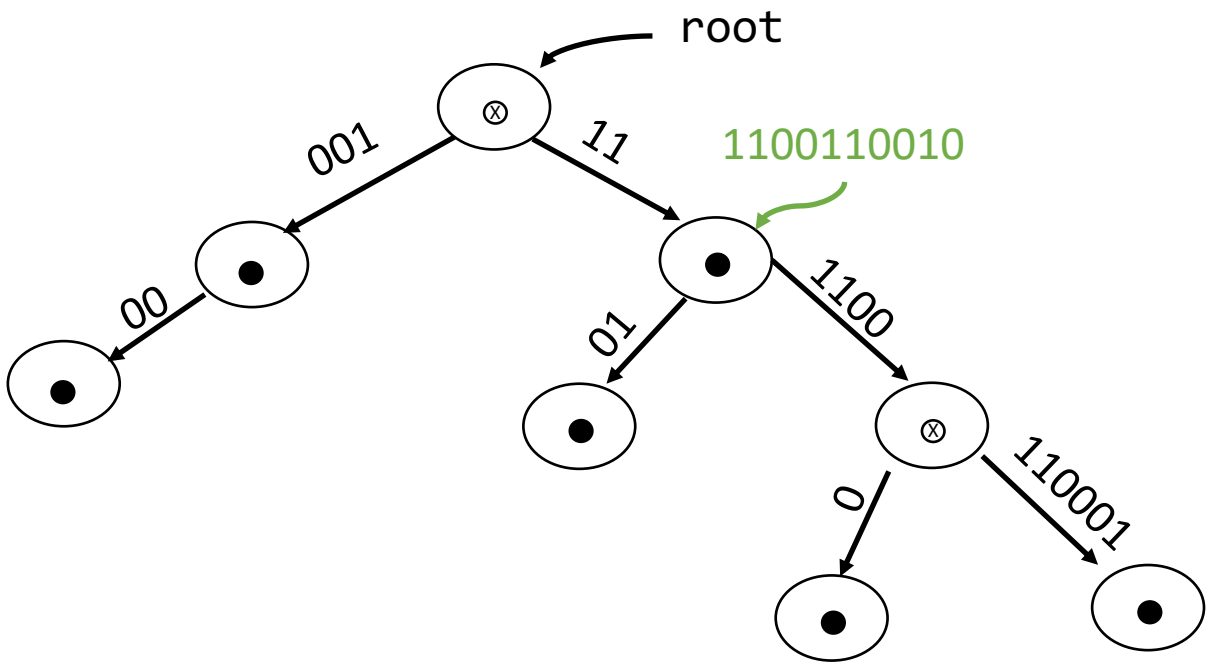
Insertion case #5: Split node

Insert 111100110010



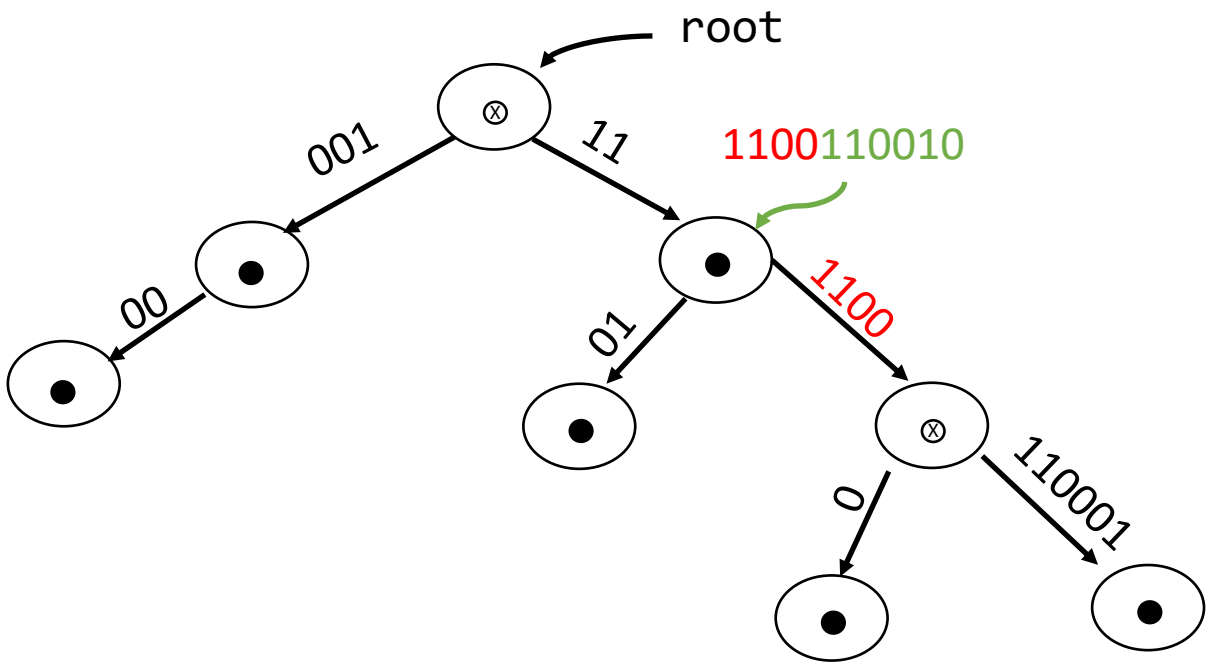
Insertion case #5: Split node

Insert 111100110010



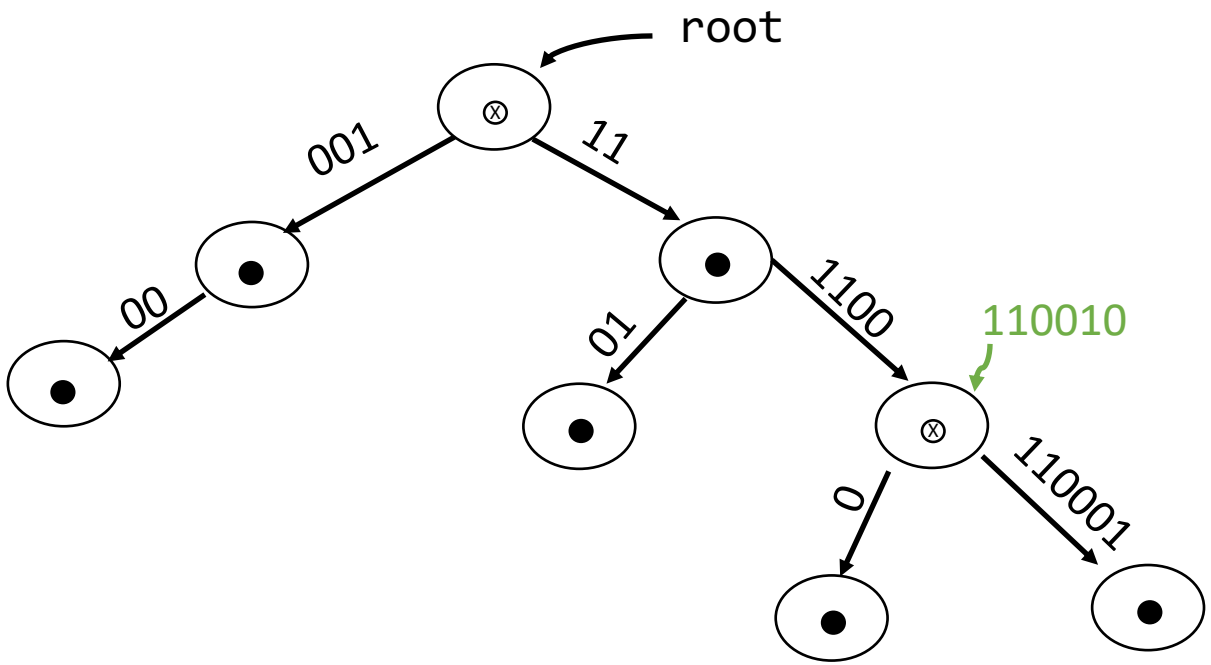
Insertion case #5: Split node

Insert 111100110010



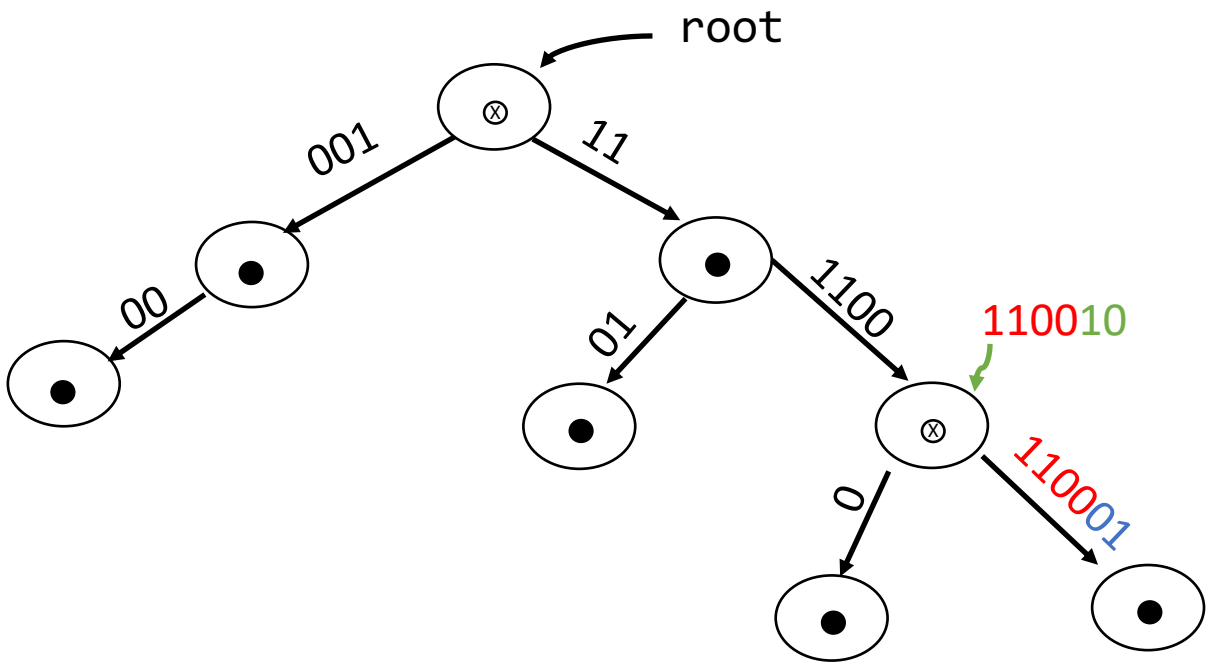
Insertion case #5: Split node

Insert 111100110010



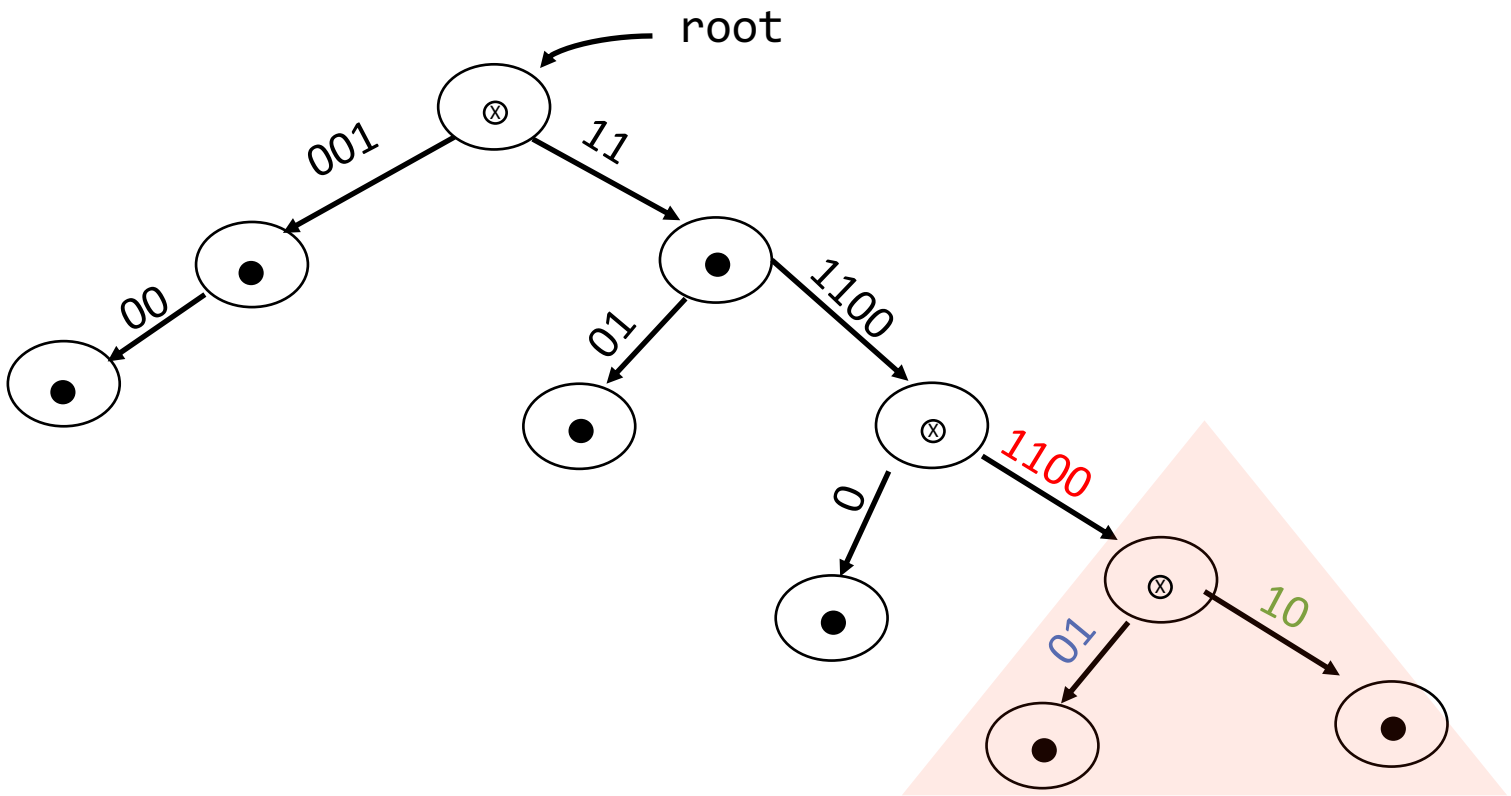
Insertion case #5: Split node

Insert 111100110010



Insertion case #5: Split node

Insert 111100110010

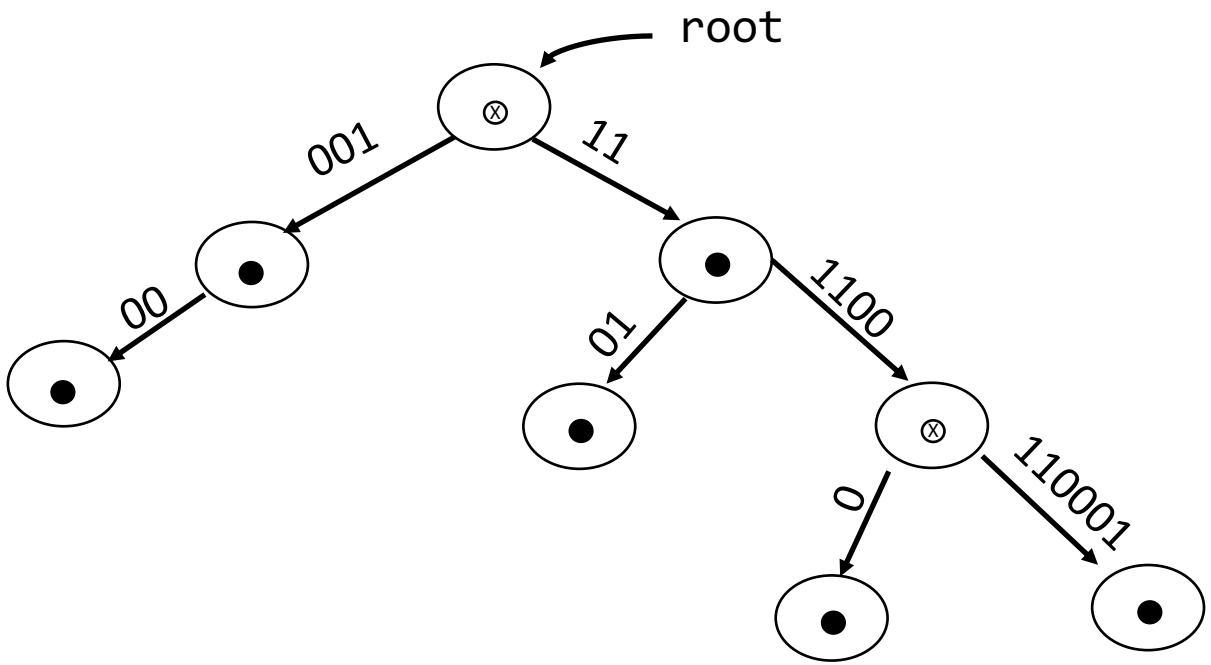


Deletion

- Cases of deletion:
  1. The current node is `null`: key not in trie, nothing to delete.
  2. The (remaining) key is a (strict) prefix of the node's string: (original) key not in trie, nothing to delete.
  3. The (remaining) key is exactly as long as the node's string, but it's not equal to it: nothing to delete.
  4. The node's string is a (strict) prefix of the (remaining) key: consume and recurse
  5. The (remaining) key is equal to the node's string:
     a. If the node has at least two non-null children (for a BPT, that's all of them), just set the bit to false.
     b. If the node has exactly one non-null child, merge the two into a new node whose key is node_key.append(child_key).
     c. If the node has no non-null children, throw him away.
- Some examples with Binary Patricia Tries follow.

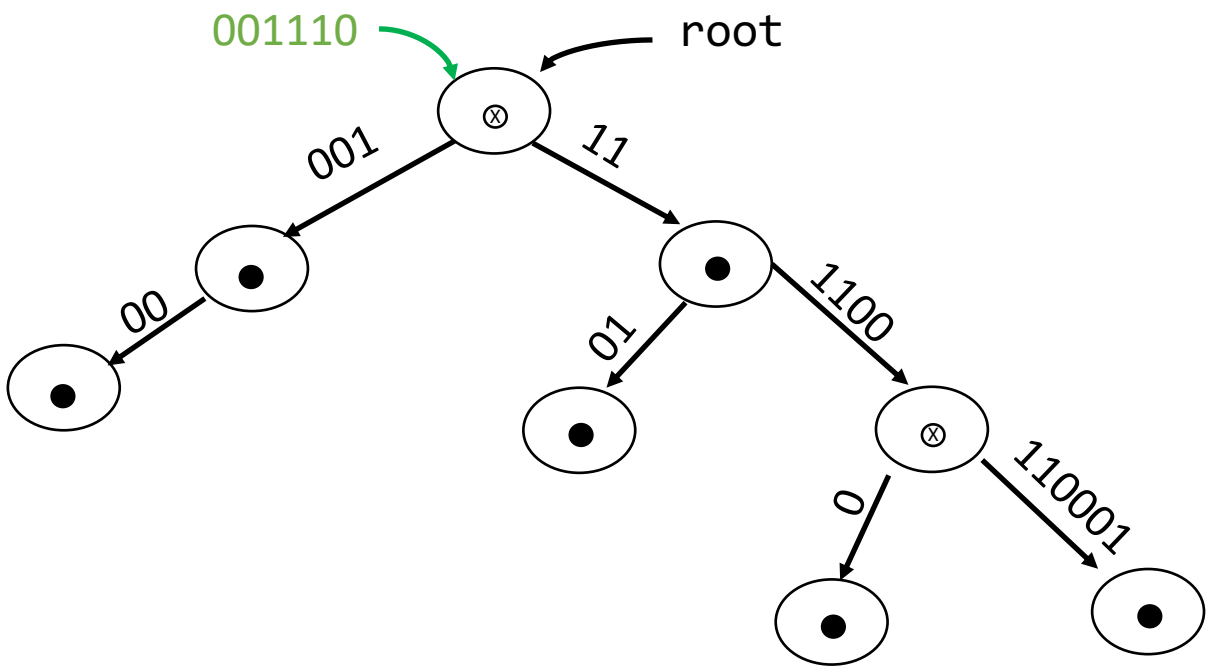
Deletion Case #1 : Fell off the trie

Delete 001110



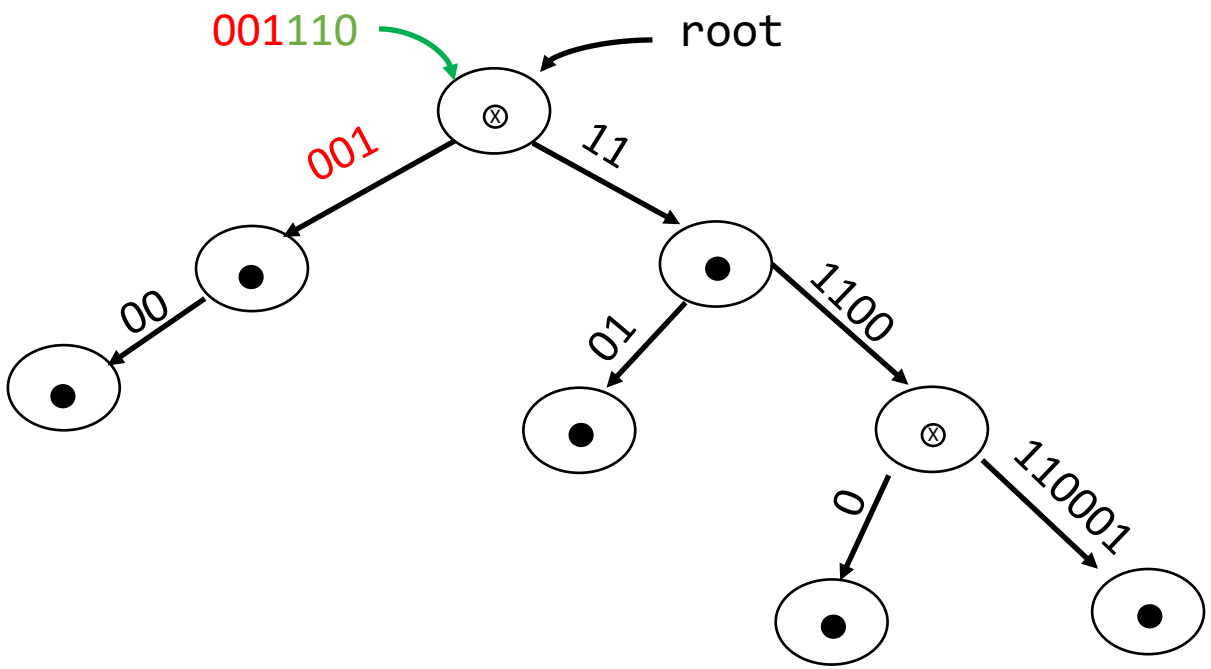
Deletion Case #1 : Fell off the trie

Delete 001110



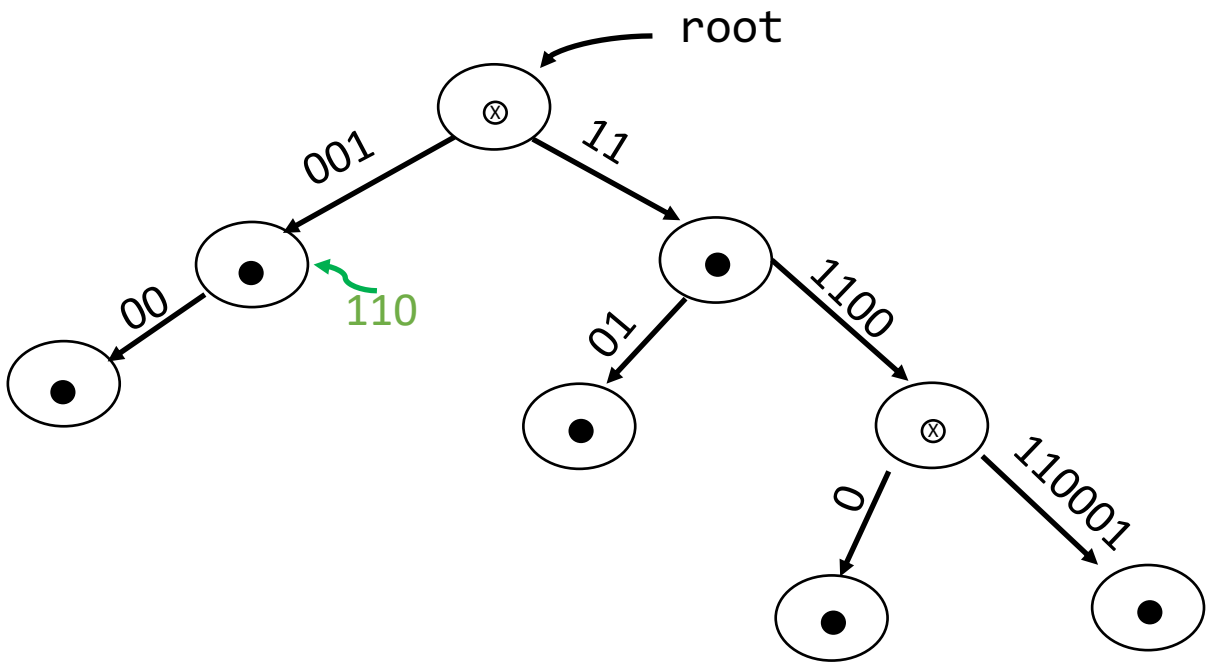
Deletion Case #1 : Fell off the trie

Delete 001110



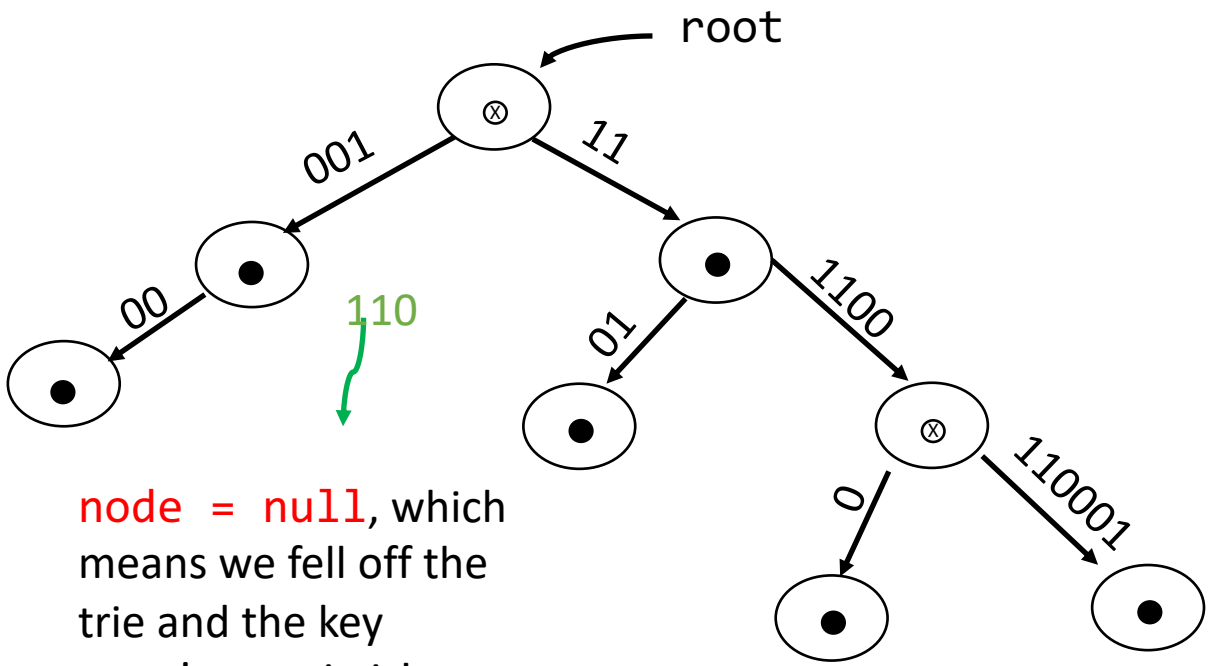
Deletion Case #1 : Fell off the trie

Delete 001110



Deletion Case #1 : Fell off the trie

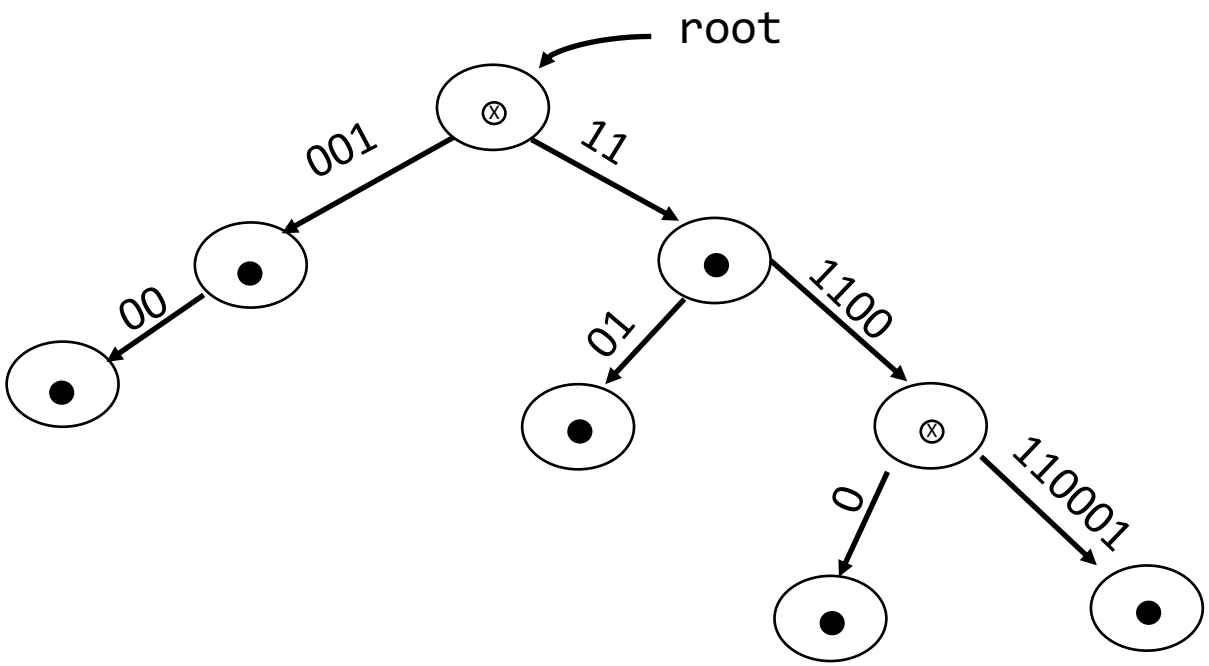
Delete 001110



`node = null`, which means we fell off the trie and the key wasn't even in it!

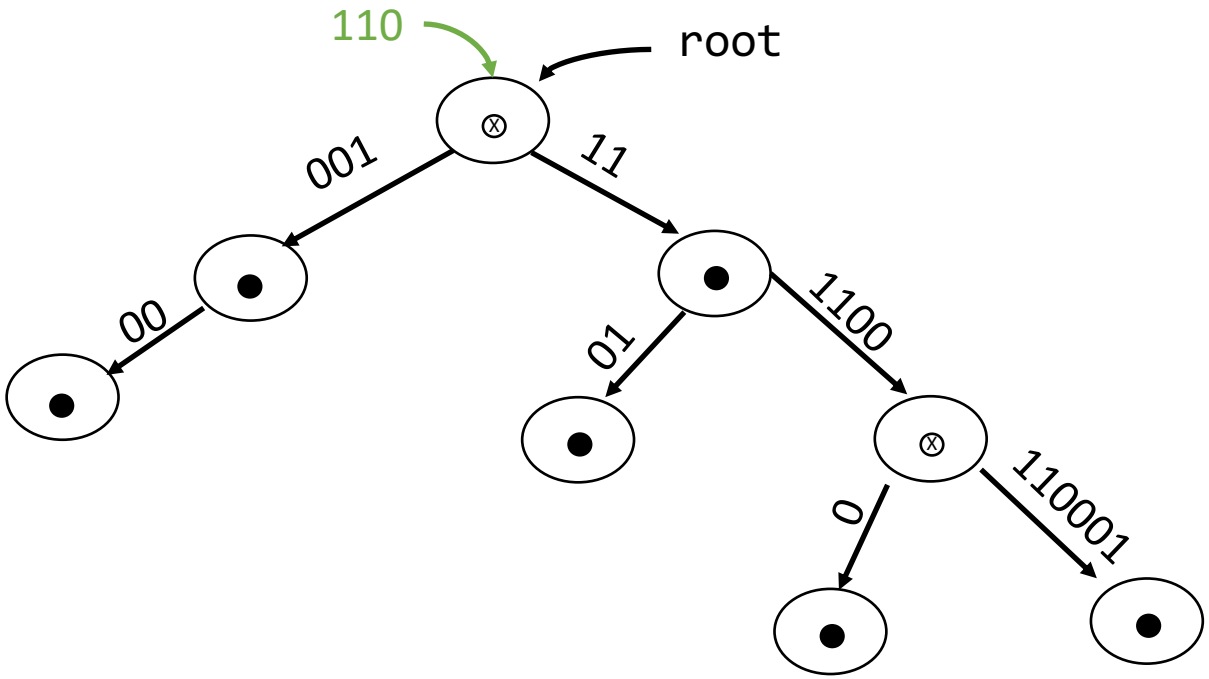
Deletion Case #2: key shorter

Delete 110



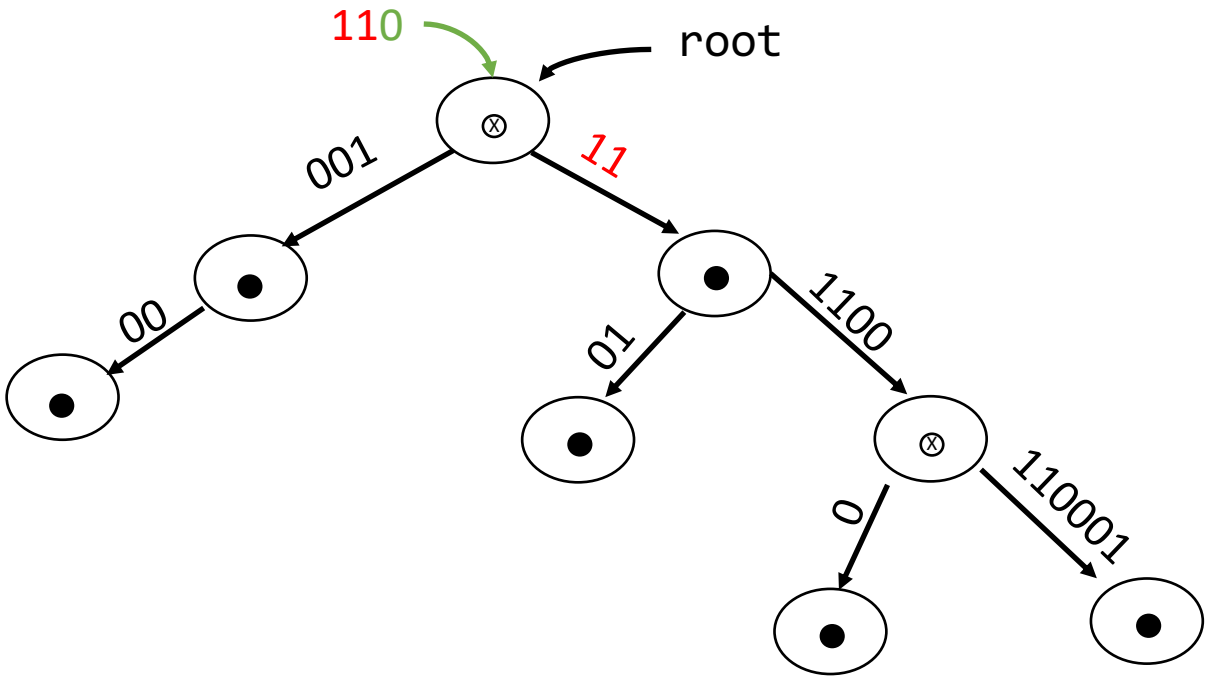
Deletion Case #2: key shorter

Delete 110



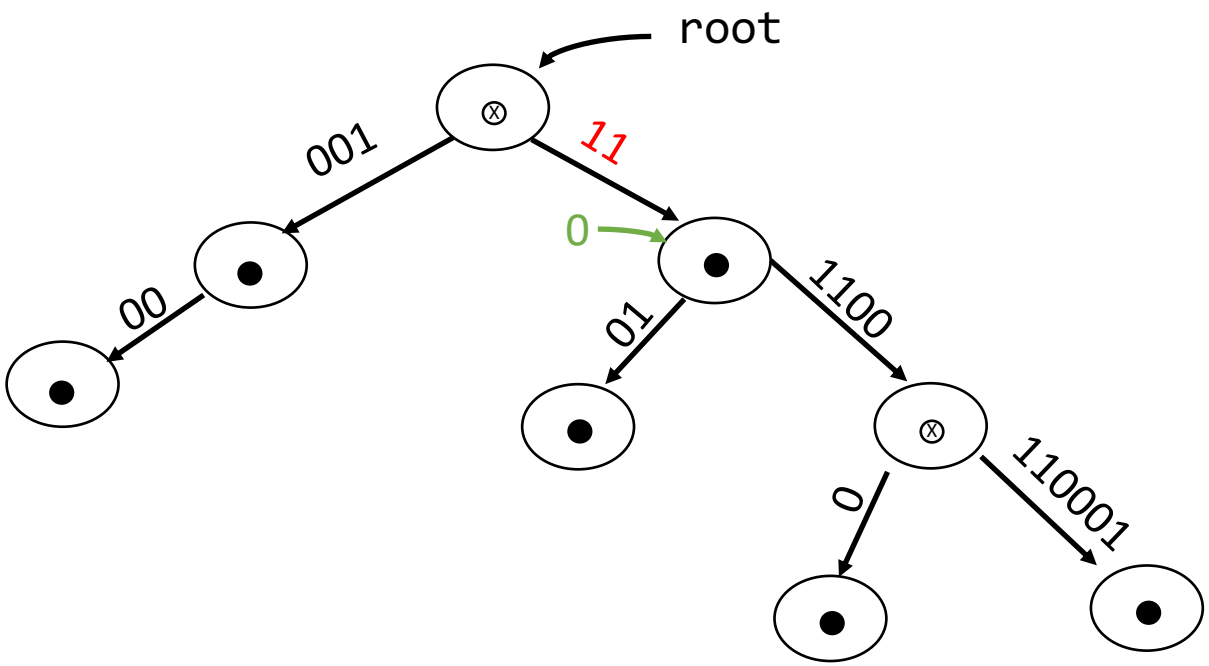
Deletion Case #2: key shorter

Delete 110



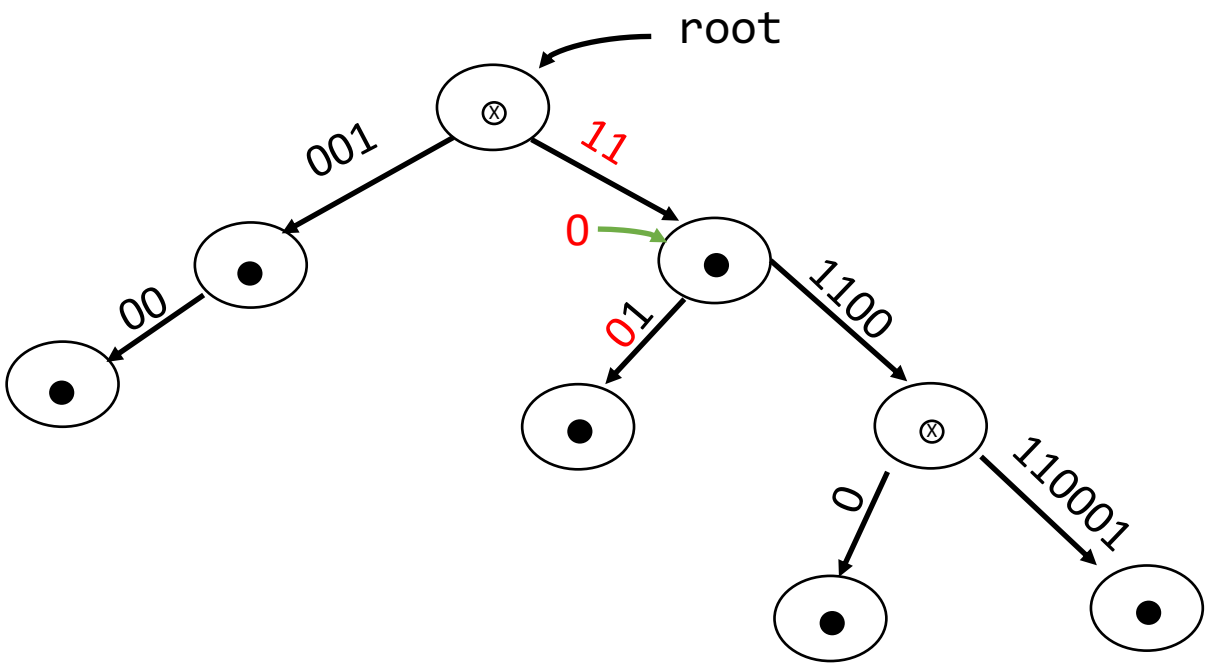
Deletion Case #2: key shorter

Delete 110



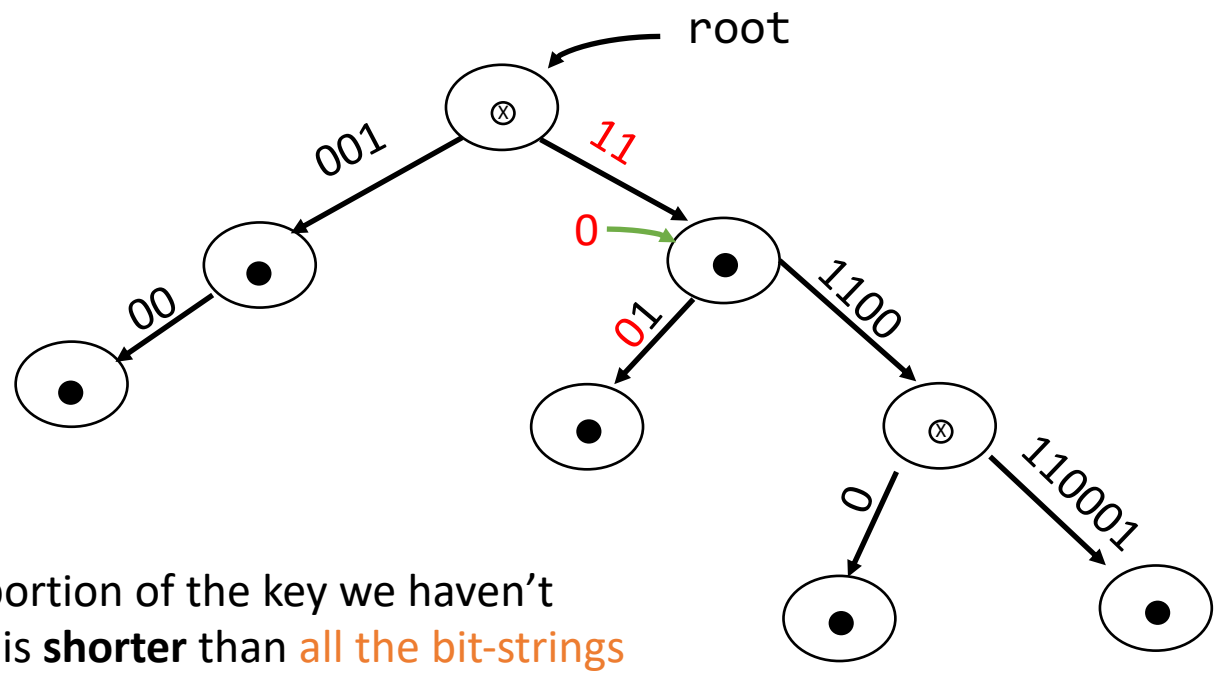
Deletion Case #2: key shorter

Delete 110



Deletion Case #2: key shorter

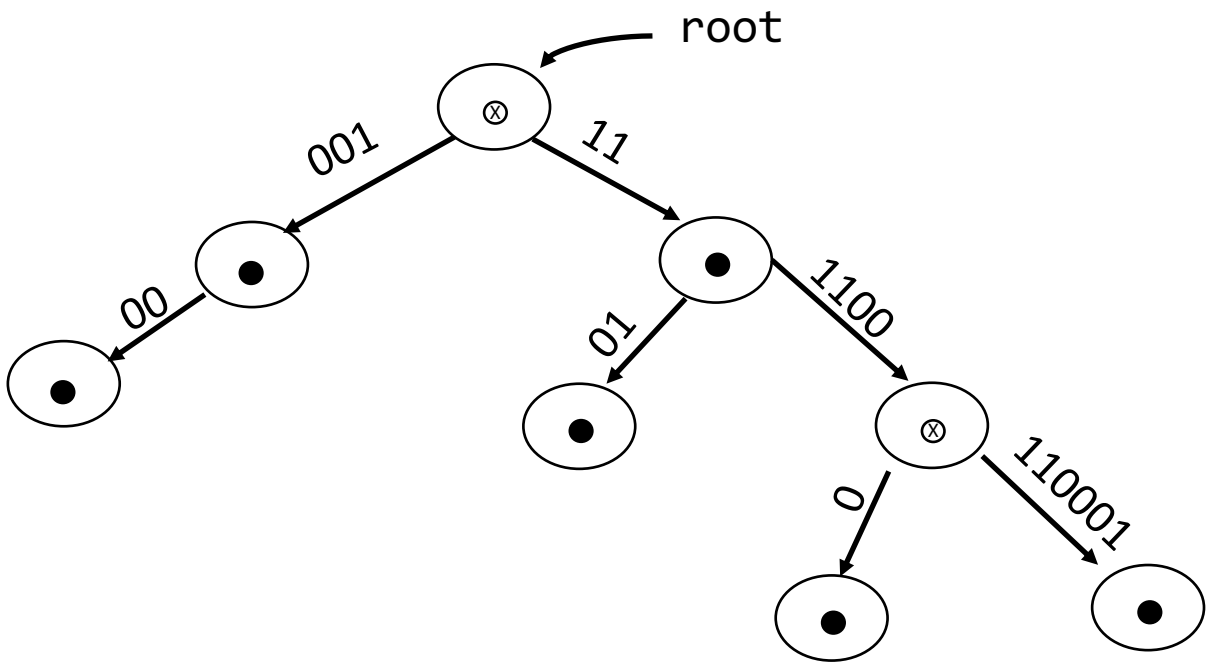
Delete 110



Since the portion of the key we haven't consumed is **shorter** than all the bit-strings that have been routed through this node (1, in this case), we know that there's no way the key can be in the trie. Deletion fails.

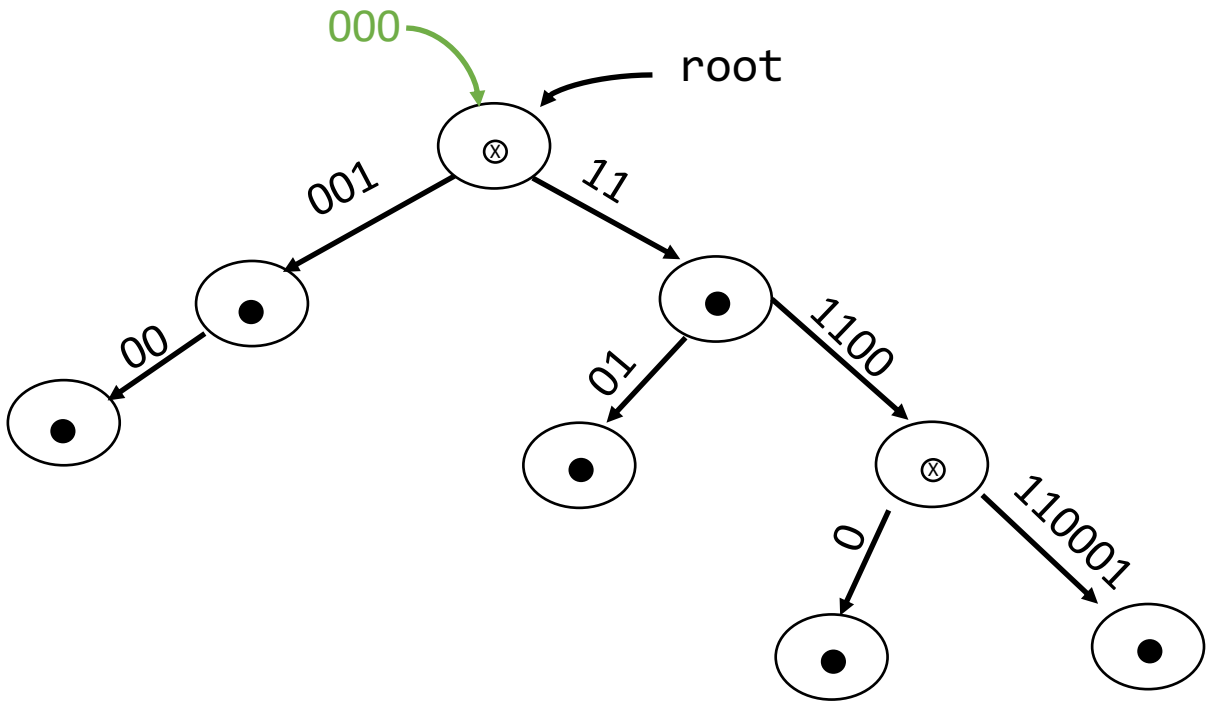
Deletion Case #3: key same length bot not equal

Delete 000



Deletion Case #3: key same length bot not equal

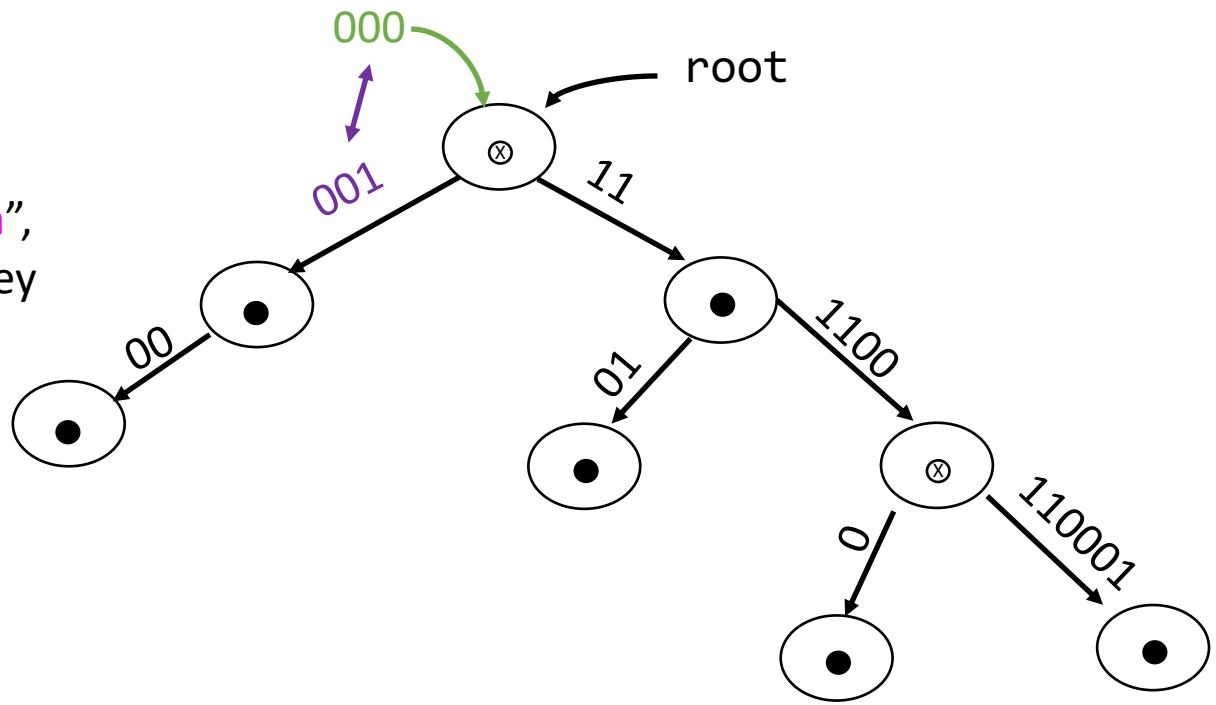
Delete 000



Deletion Case #3: key same length bot not equal

Delete 000

Same length “consumption”, but of a different kind ☹ Key not in trie, deletion fails.

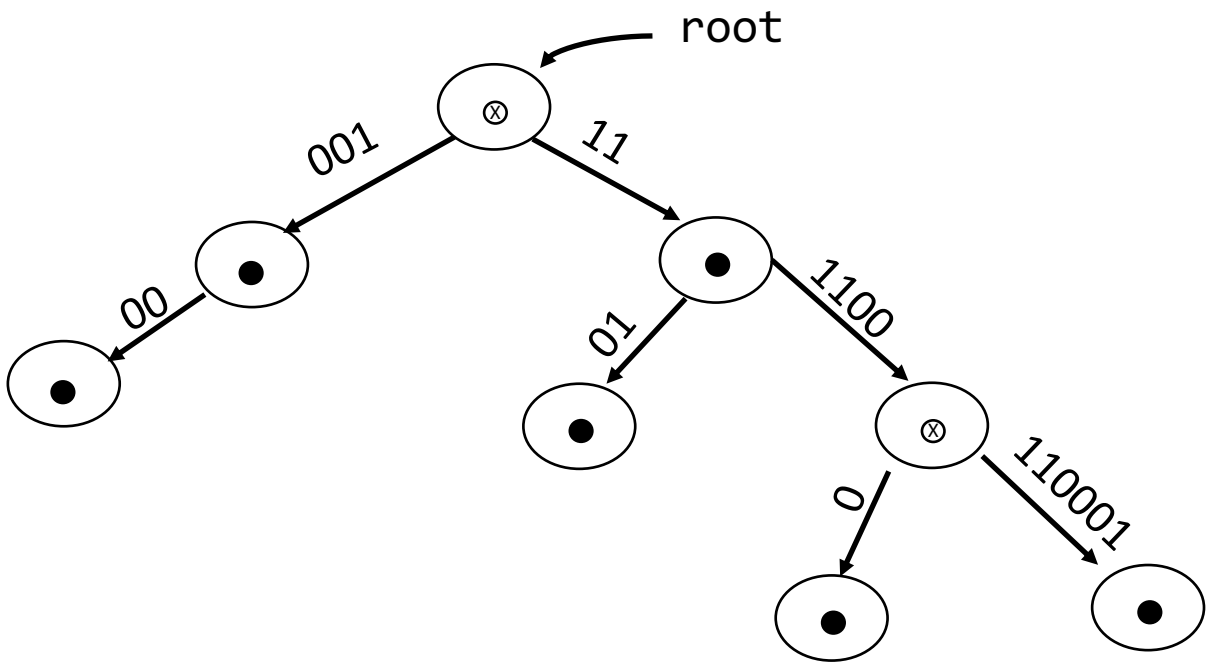


Deletion Case #4: string prefix of key: consume and recurse

- Case implicit in all others: whenever I can consume some of the (remaining) key, I do so and recurse to the appropriate child based on the subsequent character of the key.

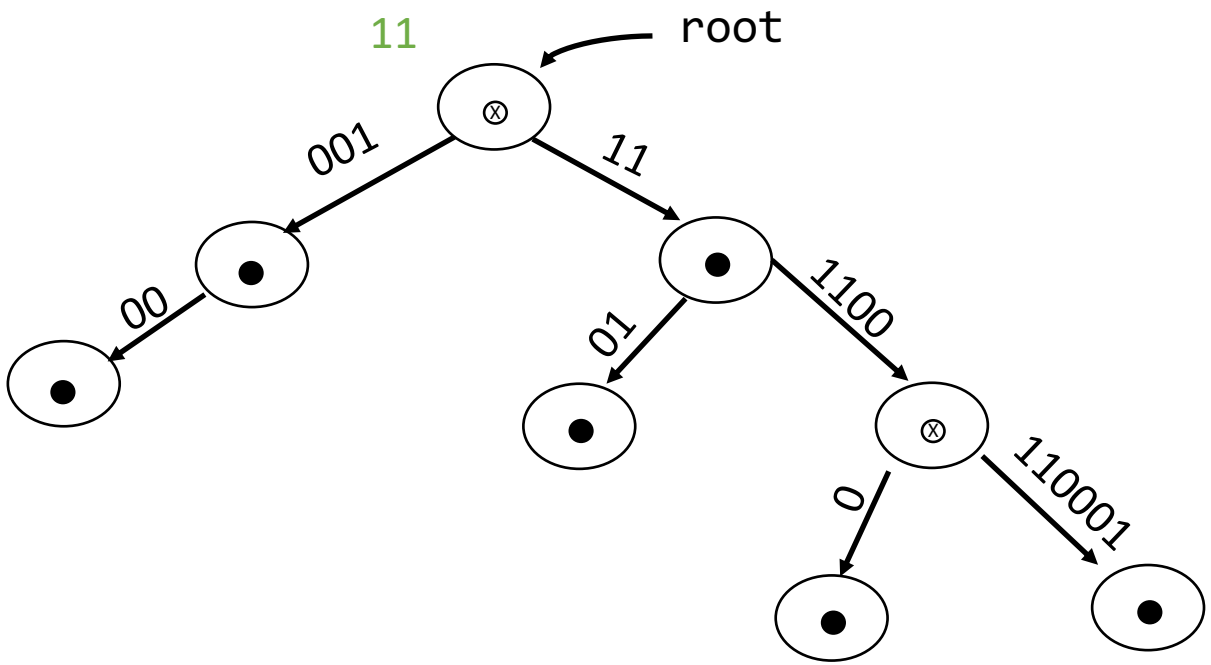
Deletion Case #5(a): Simply unset bit

Delete 11



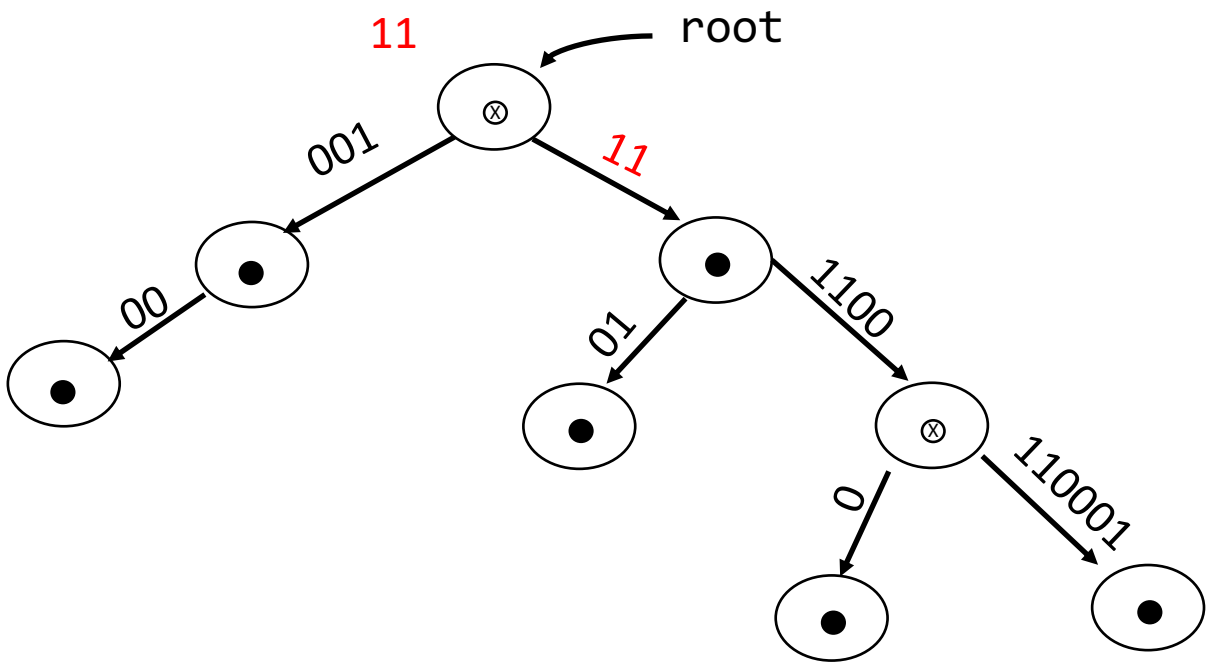
Deletion Case #5(a): Simply unset bit

Delete 11



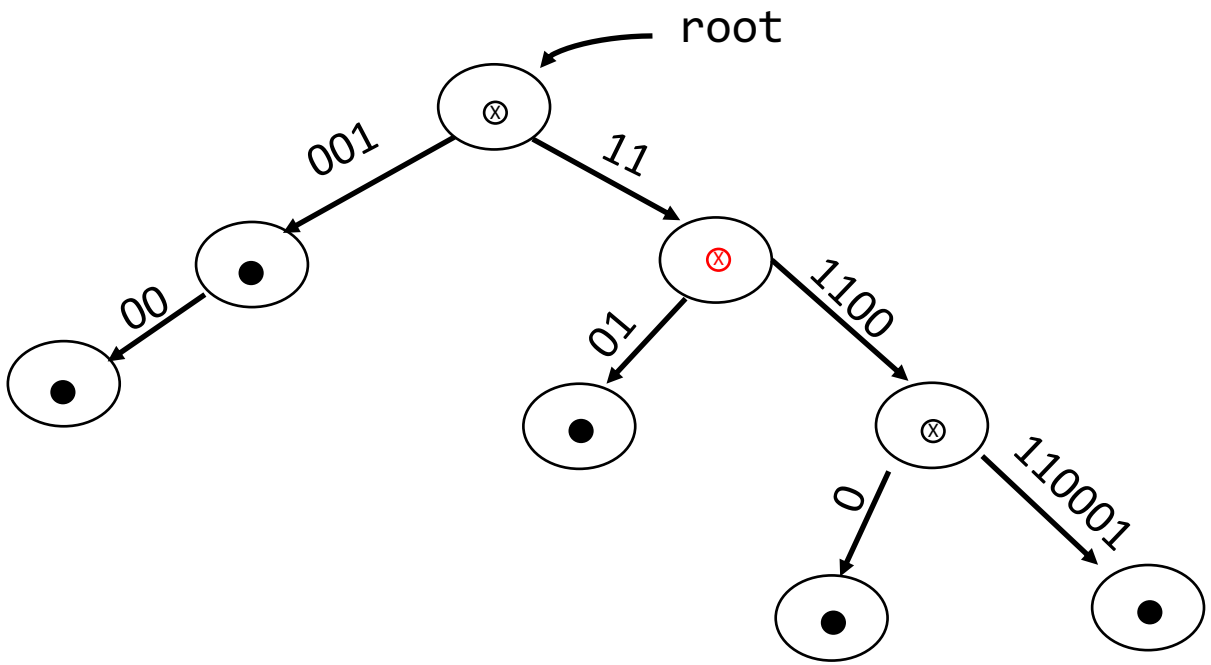
Deletion Case #5(a): Simply unset bit

Delete 11



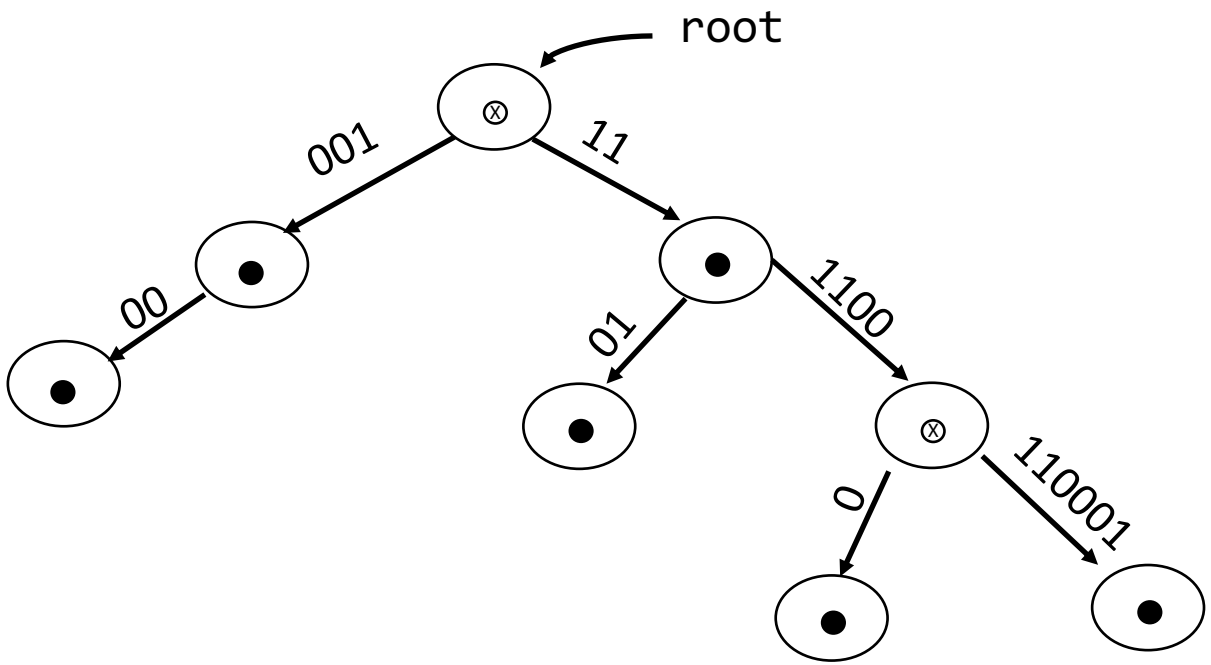
Deletion Case #5(a): Simply unset bit

Delete 11



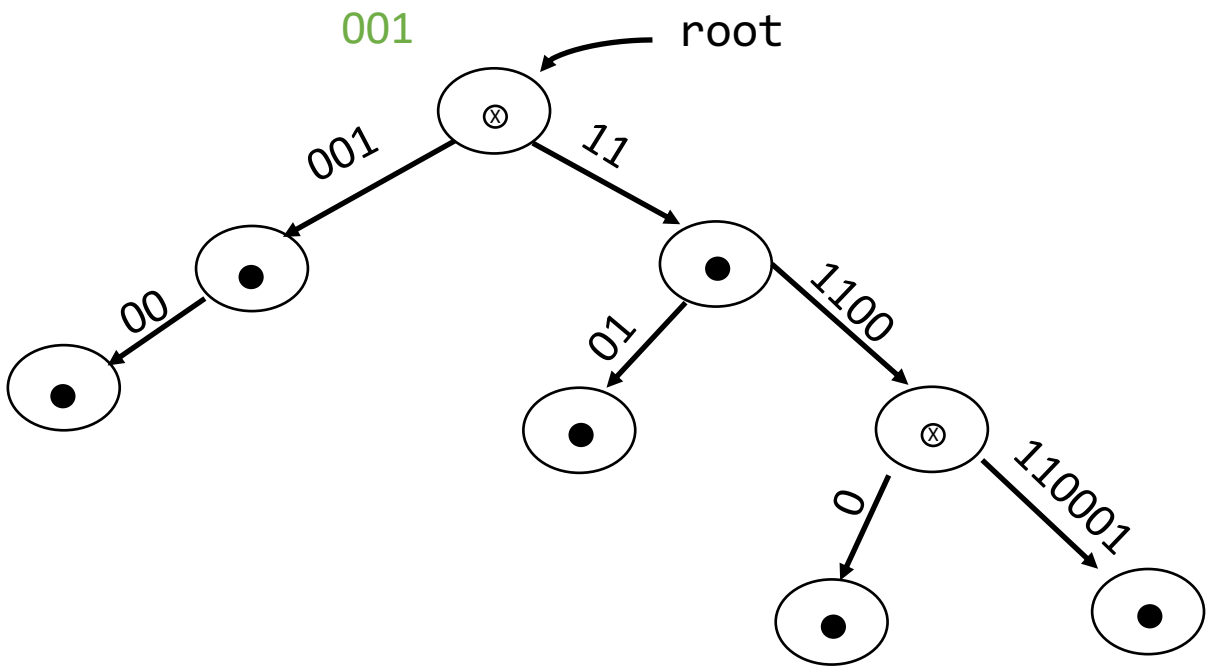
Deletion Case #5(b): Merge with child

Delete 001



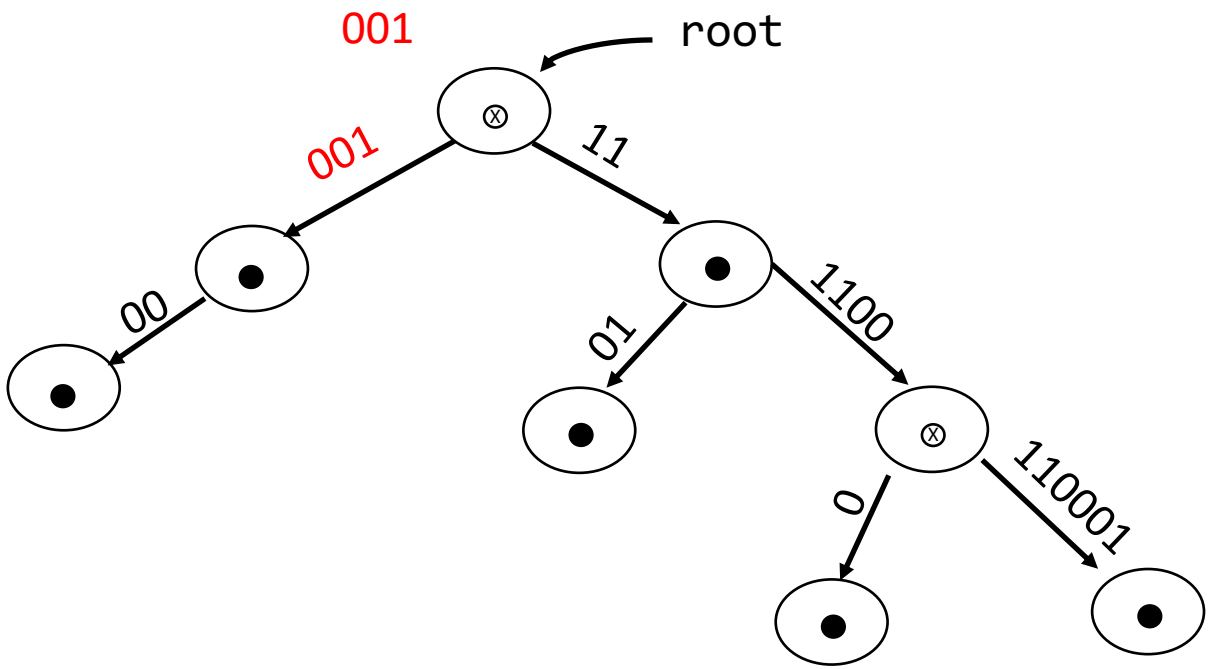
Deletion Case #5(b): Merge with child

Delete 001



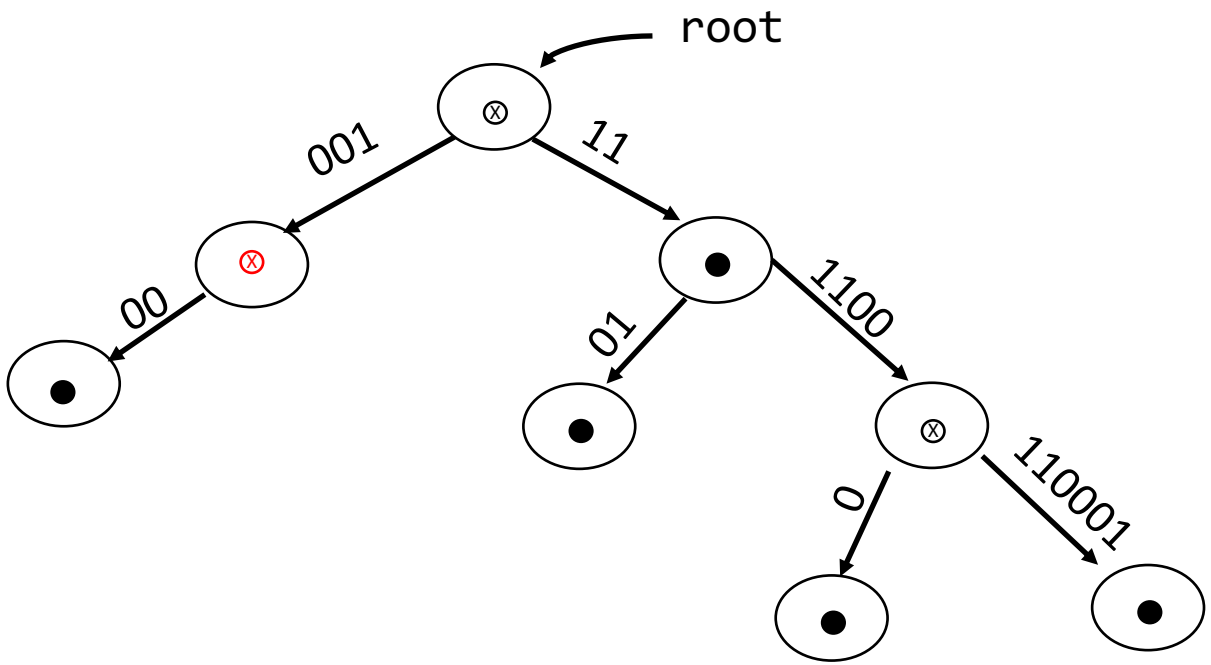
Deletion Case #5(b): Merge with child

Delete 001



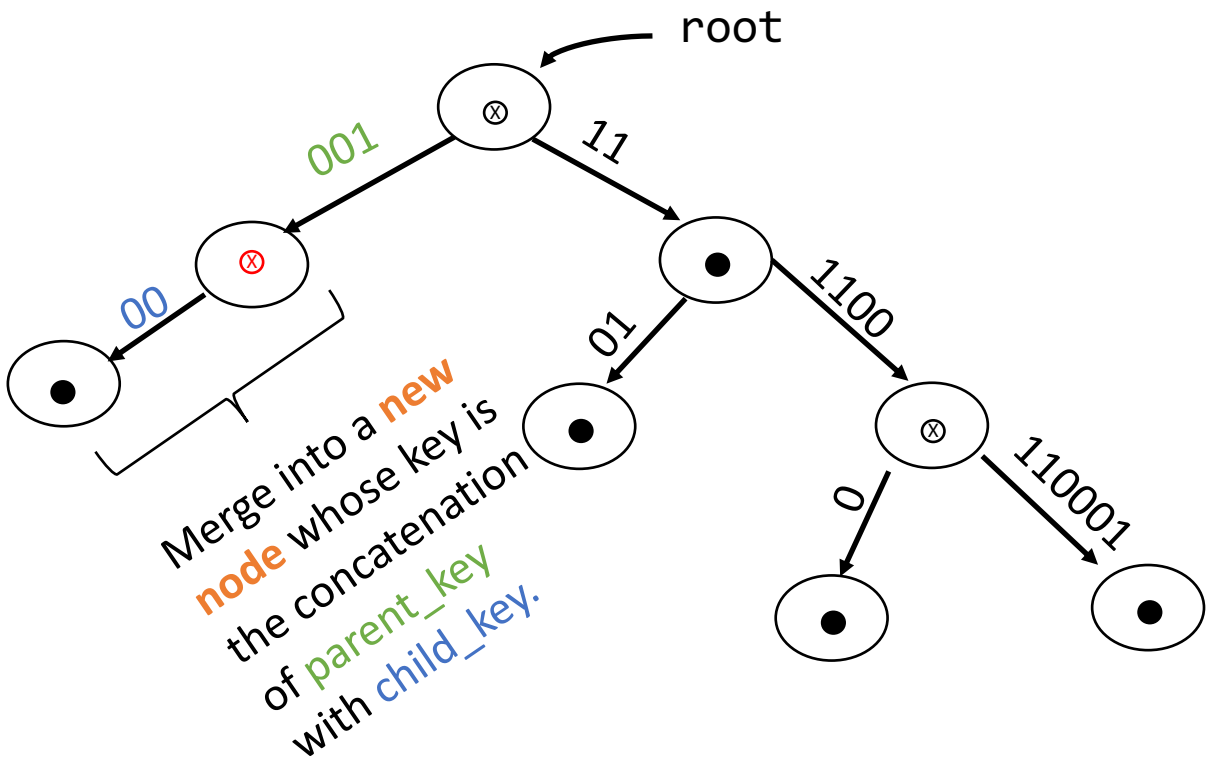
Deletion Case #5(b): Merge with child

Delete 001



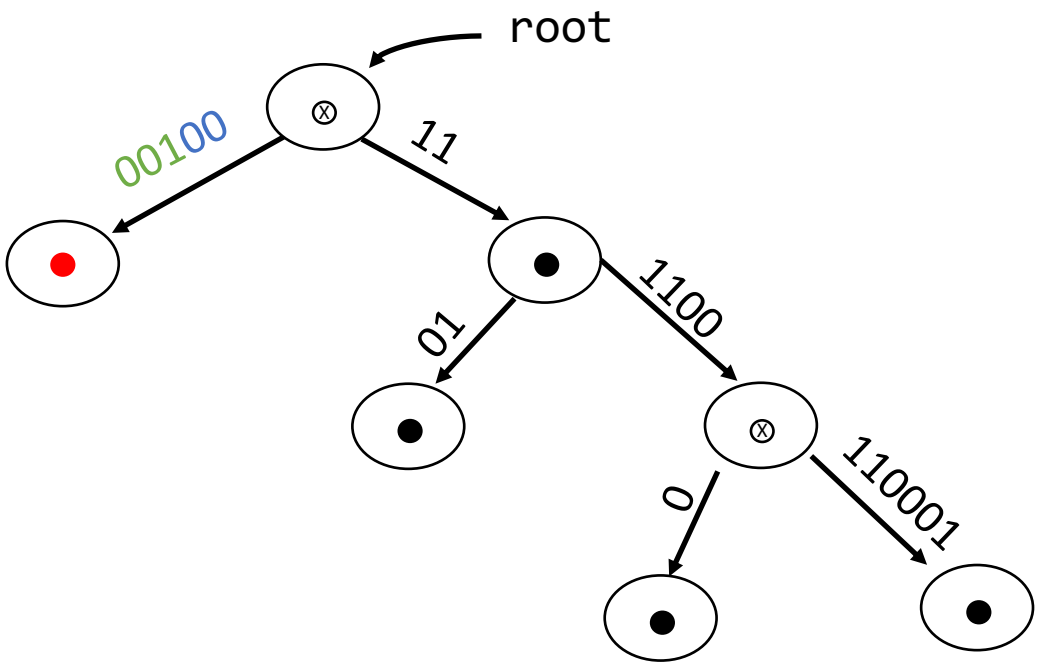
Deletion Case #5(b): Merge with child

Delete 001



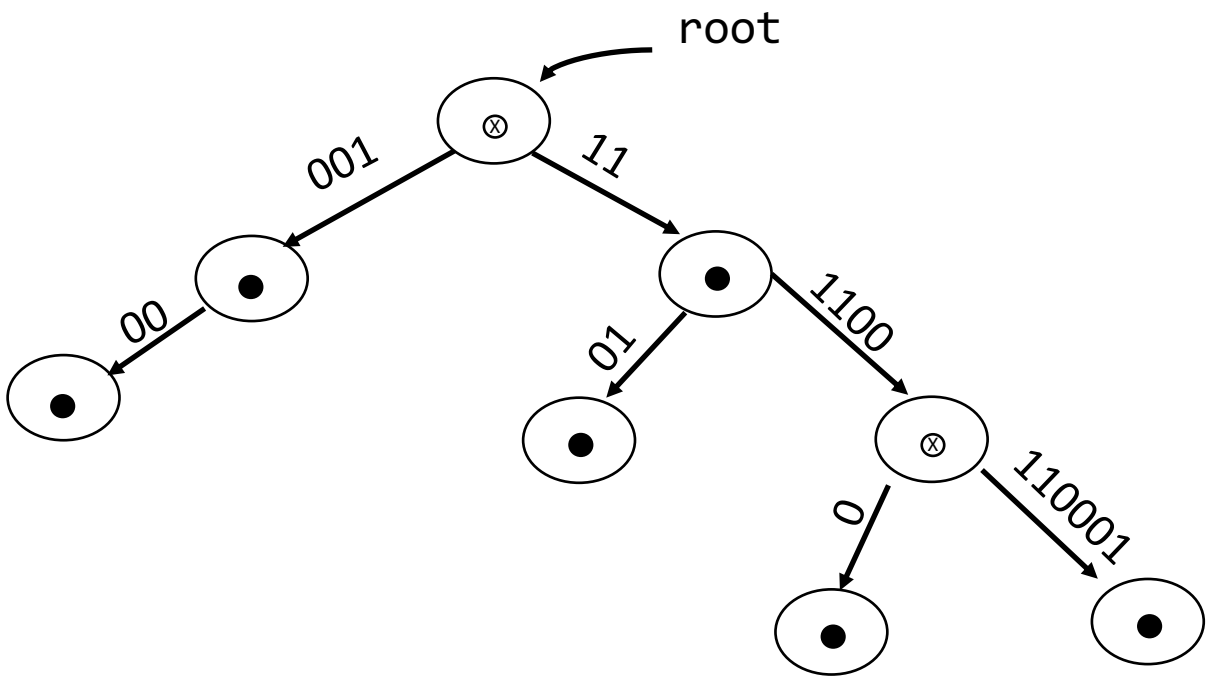
Deletion Case #5(b): Merge with child

Delete 001



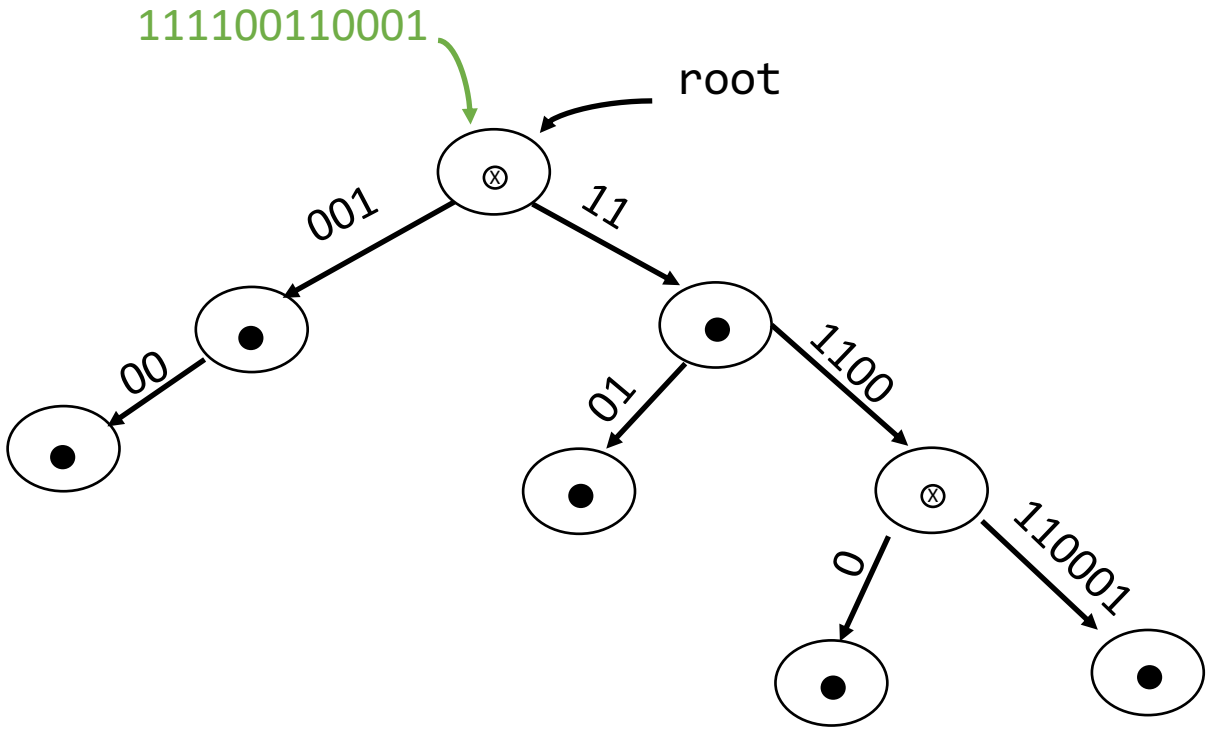
Deletion Case #5(c): Throw away node

Delete 111100110001



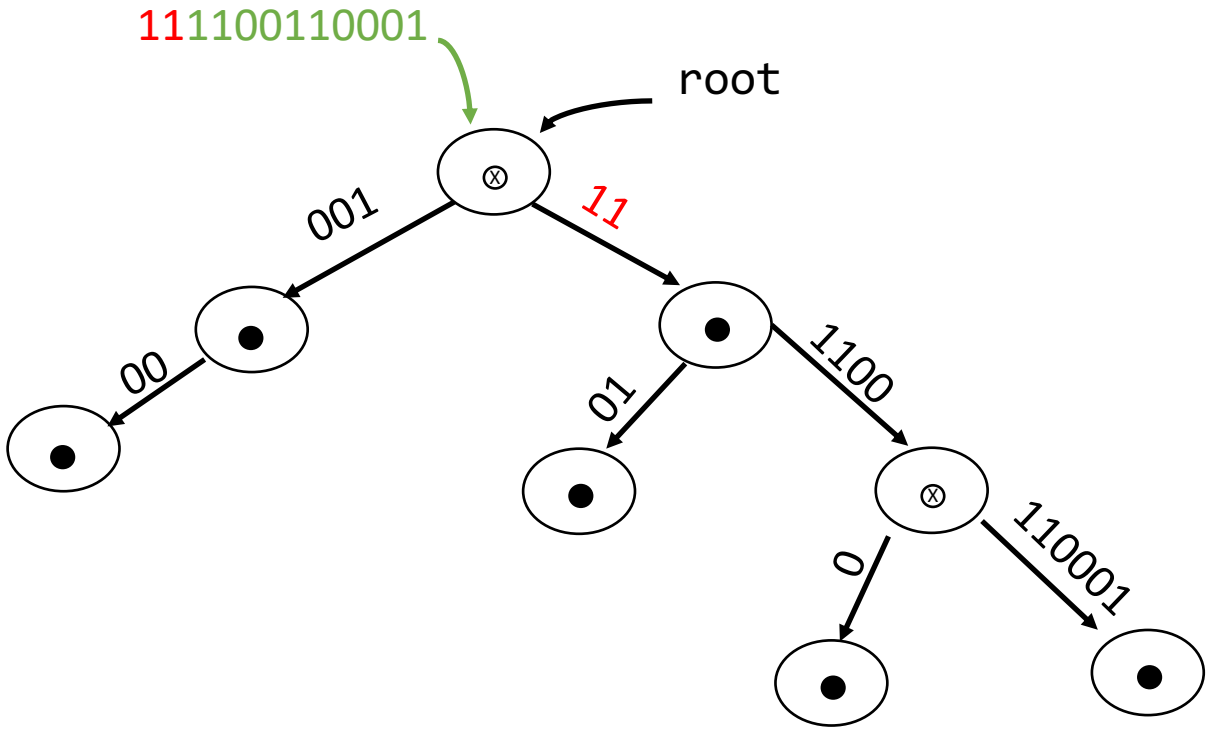
Deletion Case #5(c): Throw away node

Delete 111100110001



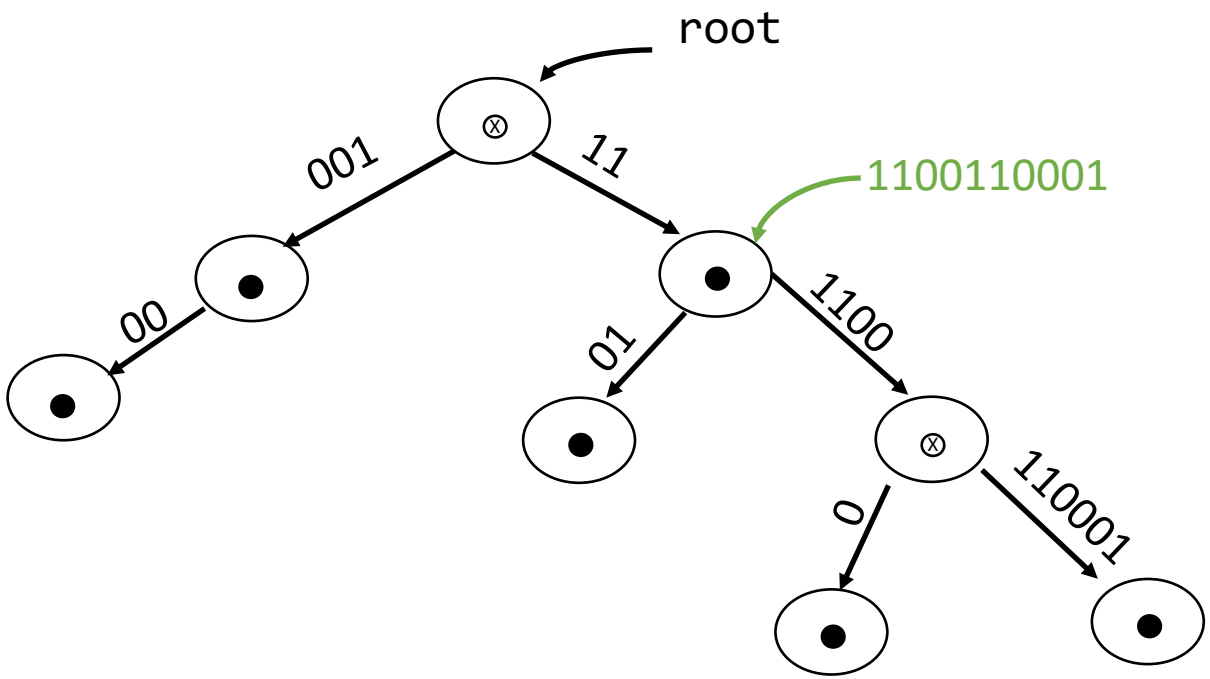
Deletion Case #5(c): Throw away node

Delete 111100110001



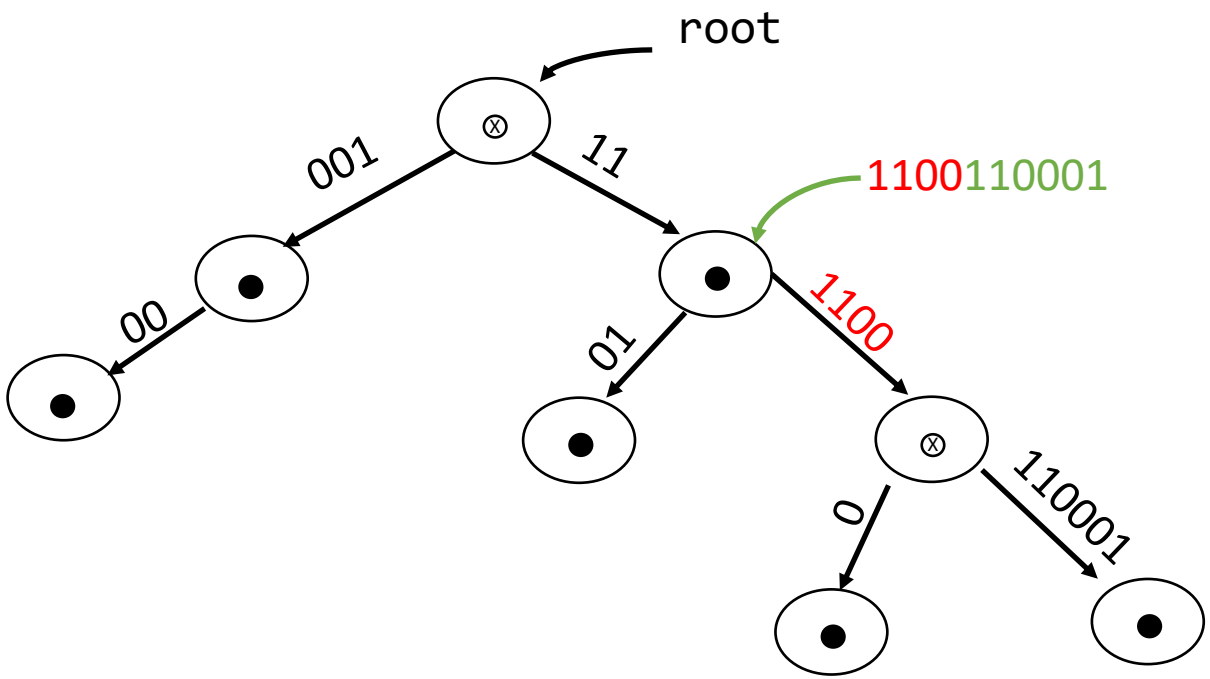
Deletion Case #5(c): Throw away node

Delete 111100110001



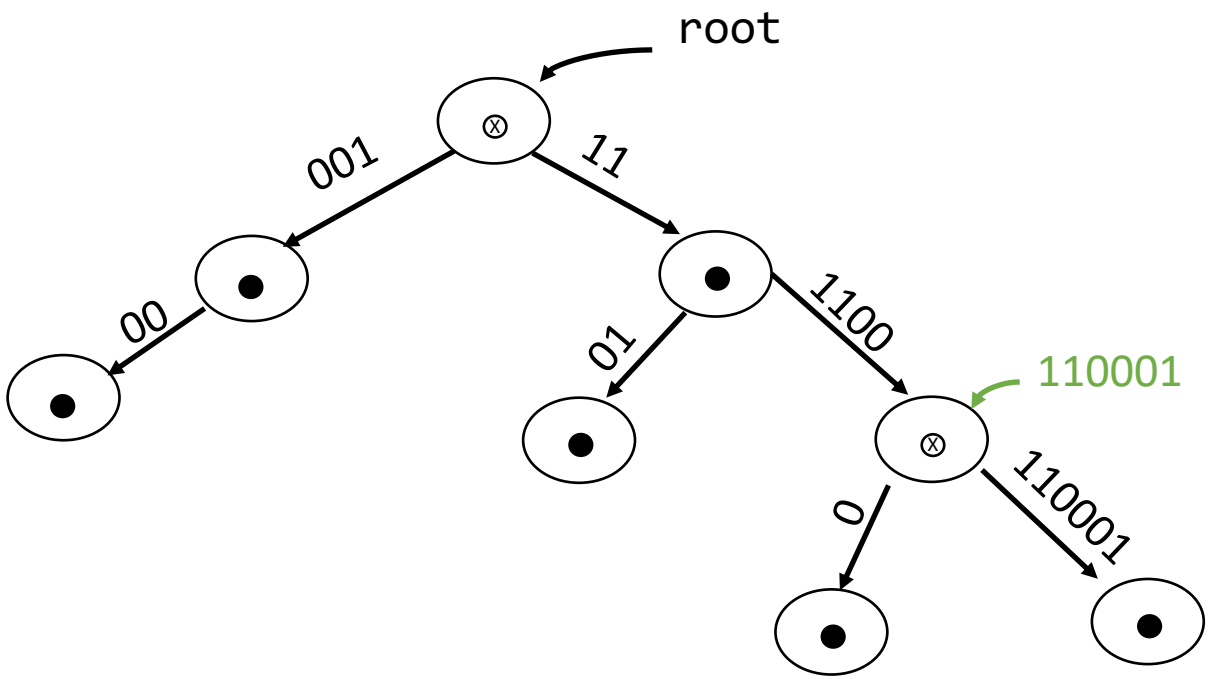
Deletion Case #5(c): Throw away node

Delete 111100110001



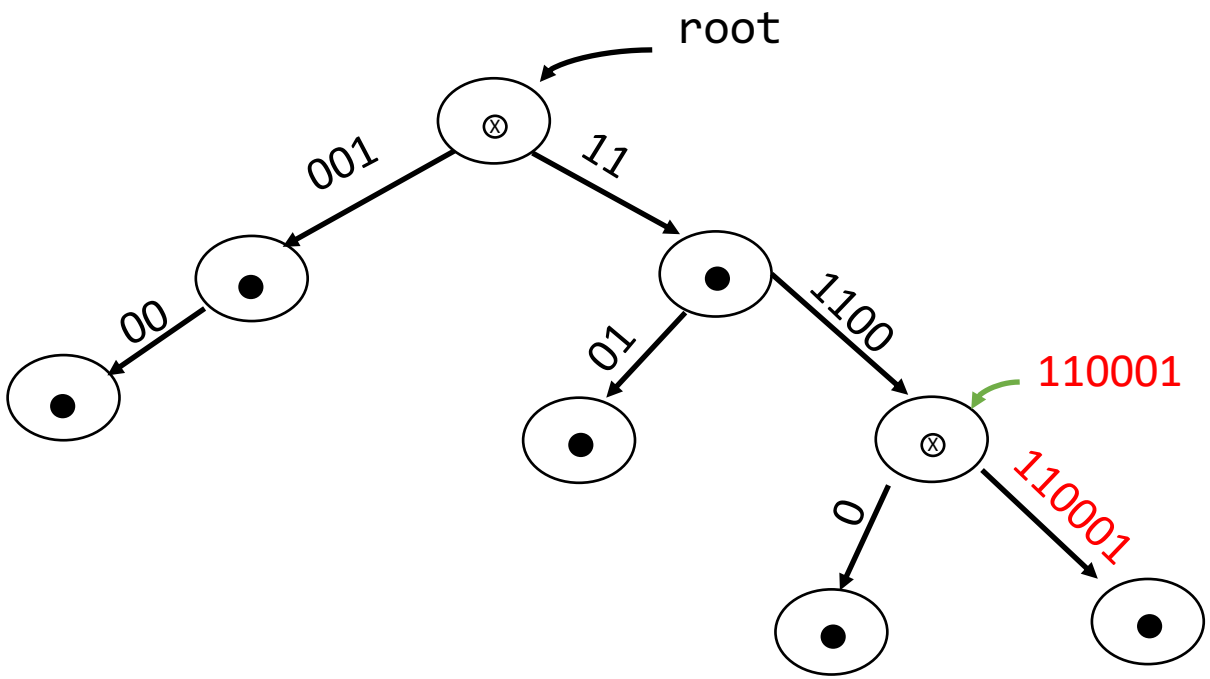
Deletion Case #5(c): Throw away node

Delete 111100110001



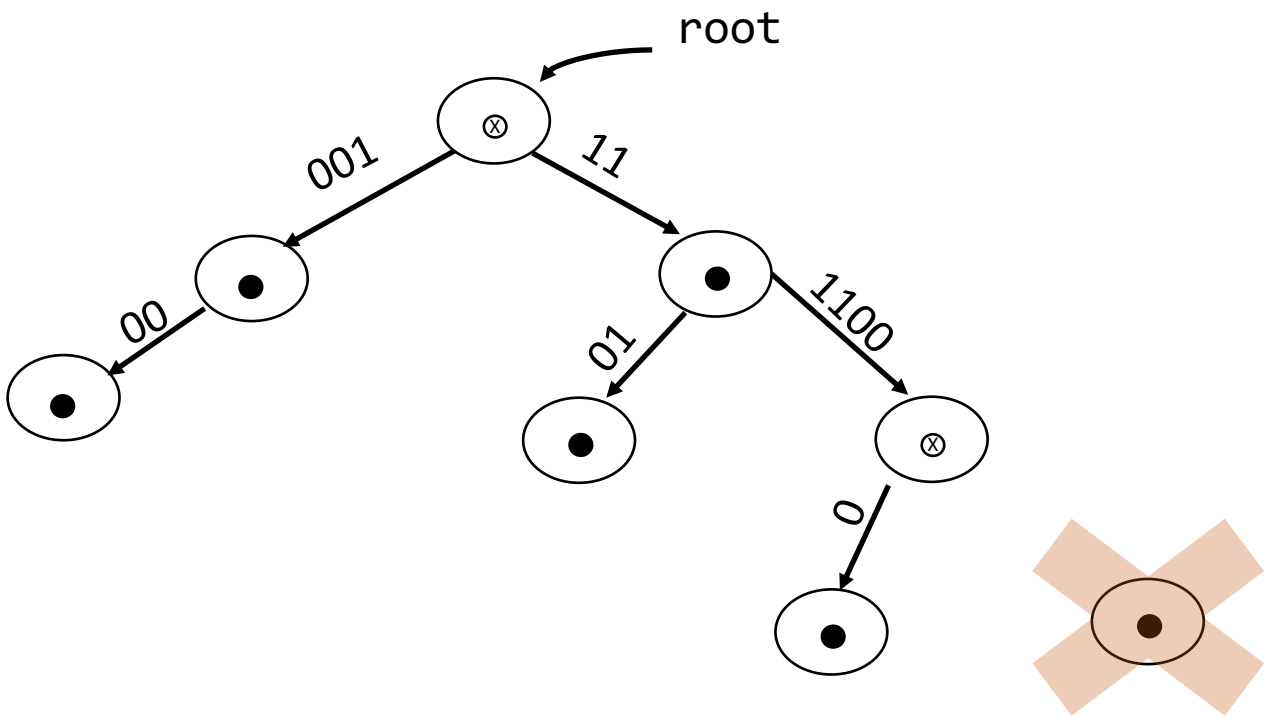
Deletion Case #5(c): Throw away node

Delete 111100110001



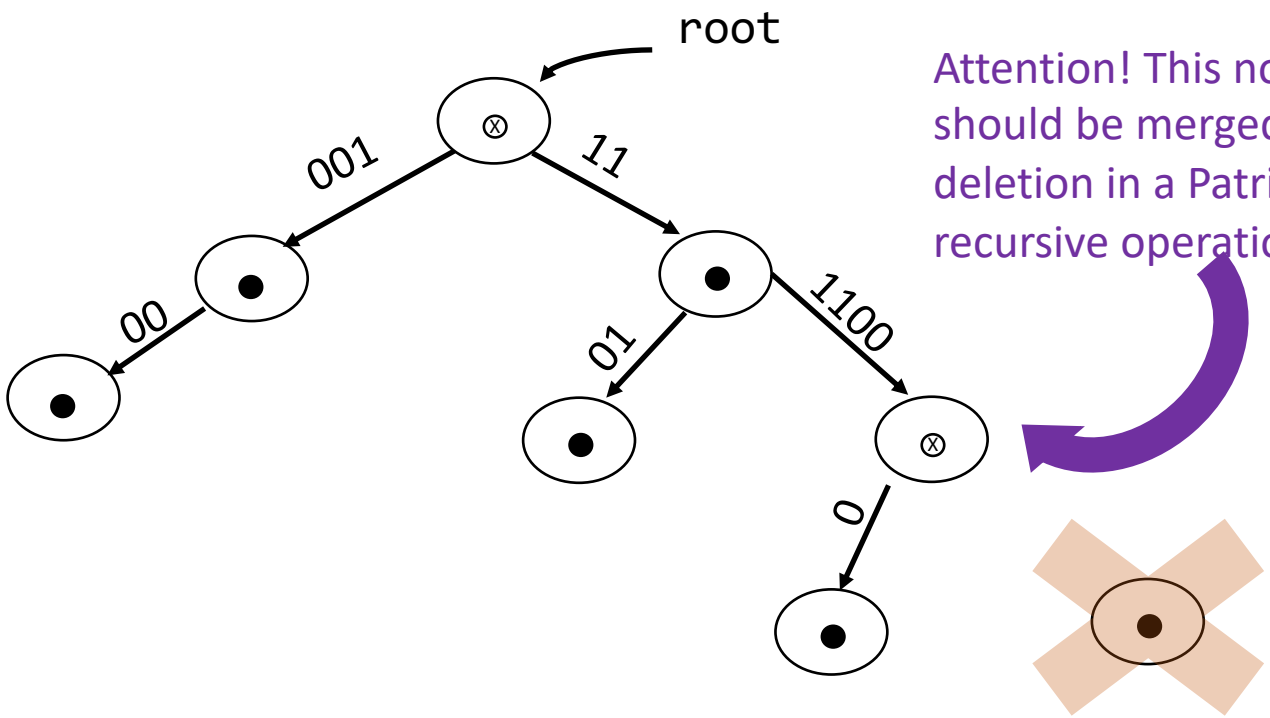
Deletion Case #5(c): Throw away node

Delete 111100110001



Deletion Case #5(c): Throw away node

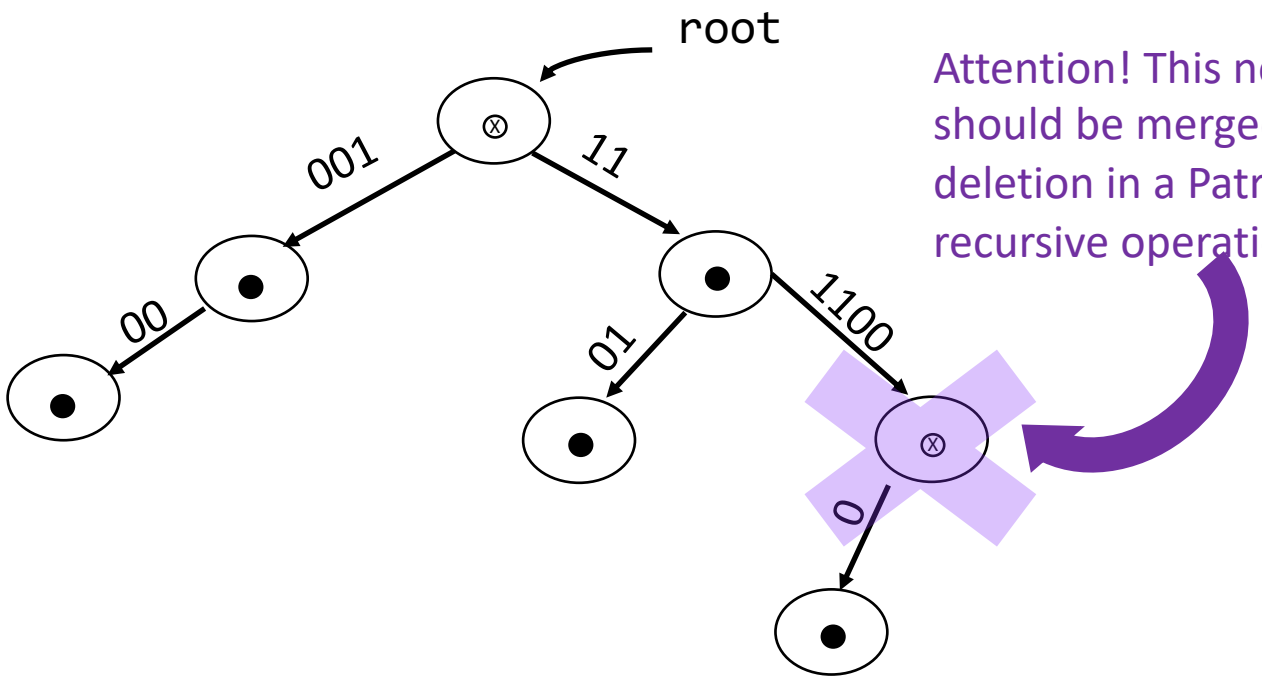
Delete 111100110001



Attention! This node is now useless, and should be merged with the child! (i.e deletion in a Patricia trie is ***not*** a tail-recursive operation!)

Deletion Case #5(c): Throw away node

Delete 111100110001



Attention! This node is now useless, and should be merged with the child! (i.e deletion in a Patricia trie is ***not*** a tail-recursive operation!)

Deletion Case #5(c): Throw away node

Delete 111100110001