CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

Armin Jamshidpey, Éric Schost
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2025

version 2025-06-16 17:40

Outline

Outline

Dictionary ADT: Implementations thus far

Realizations we have seen so far:

- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete (worst-case)
- **Skip lists**:
  $\Theta(\log n)$ search, insert, and delete (expected)
- Various other realizations sometimes faster on insert,
  but *search* always takes $\Omega(\log n)$ time.

Dictionary ADT: Implementations thus far

Realizations we have seen so far:

- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete (worst-case)
- **Skip lists**:
  $\Theta(\log n)$ search, insert, and delete (expected)
- Various other realizations sometimes faster on insert,
  but *search* always takes $\Omega(\log n)$ time.

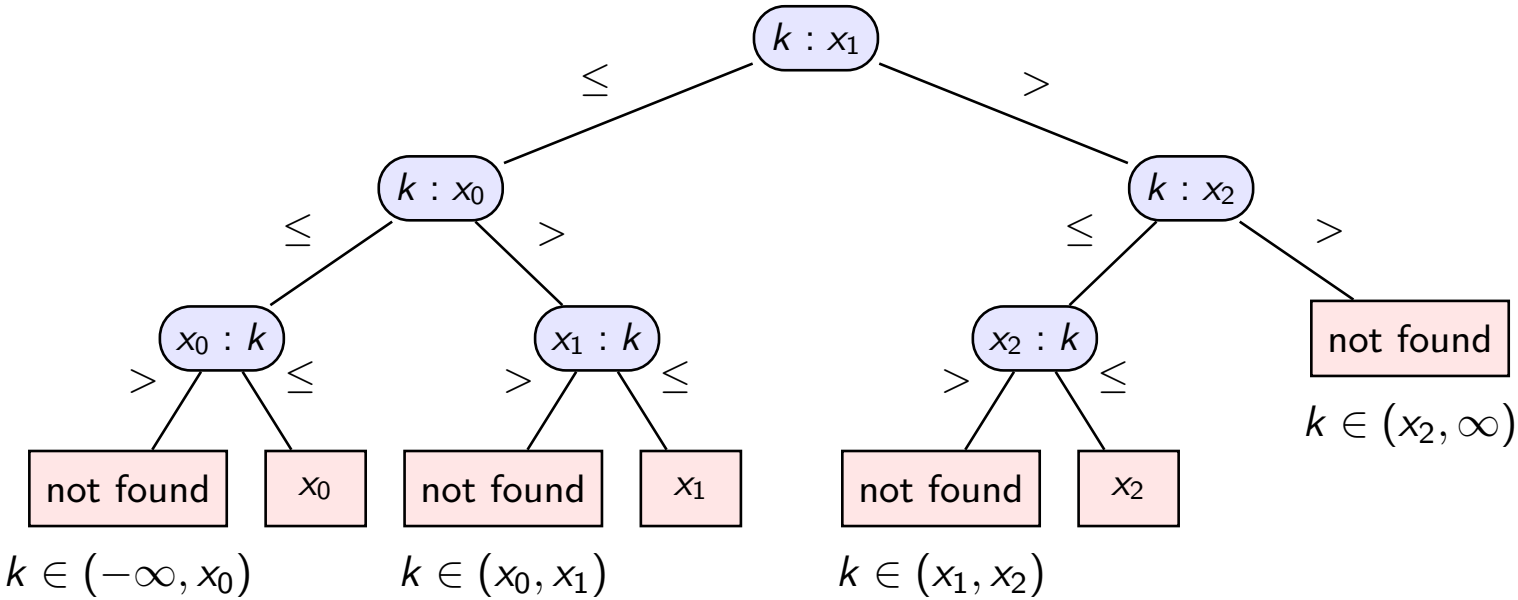
Question: Can one do better than $\Theta(\log n)$ time for *search*?
Answer: Yes and no! *It depends on what we allow*.

- No: Comparison-based searching lower bound is $\Omega(\log n)$.
- Yes: Non-comparison-based searching can achieve $o(\log n)$ (under restrictions!).

Lower bound for search

Theorem: Any *comparison-based* algorithm requires in the worst case $\Omega(\log n)$ comparisons to search among n distinct items.

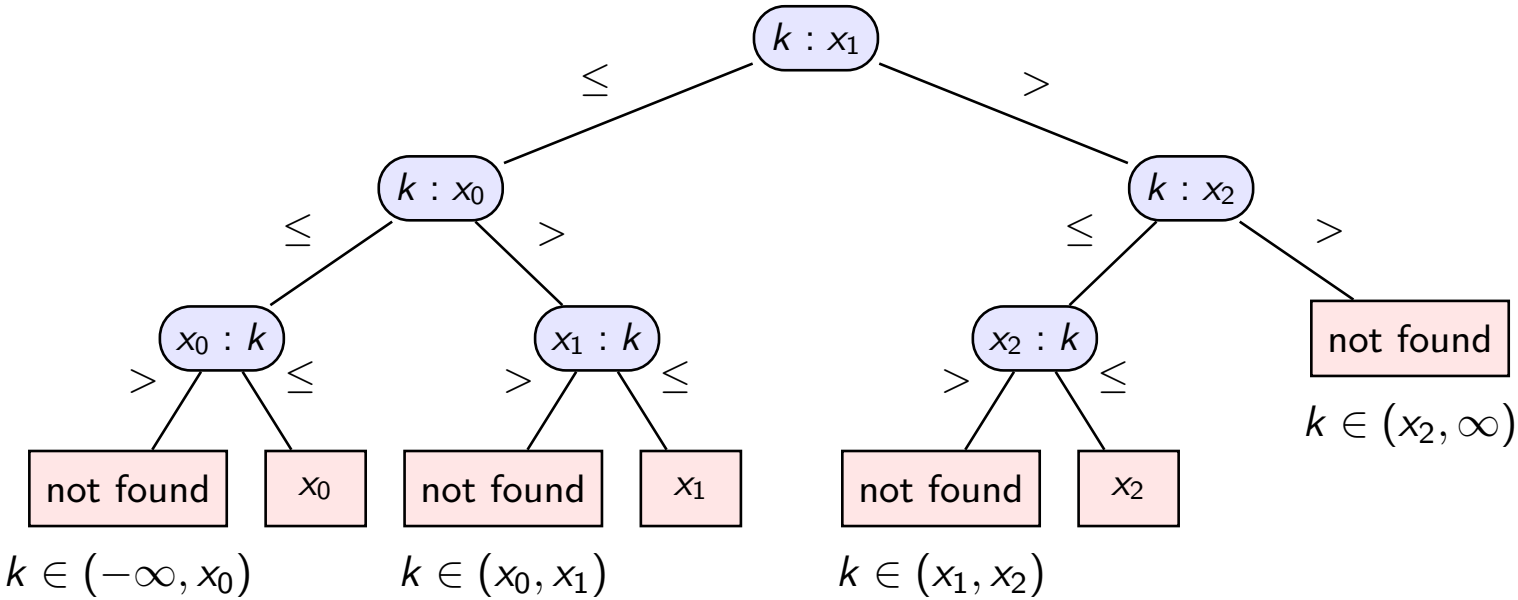
Proof: Via decision tree for items $x_0, \ldots, x_{n-1}$ and search for k



Lower bound for search

Theorem: Any *comparison-based* algorithm requires in the worst case $\Omega(\log n)$ comparisons to search among n distinct items.

Proof: Via decision tree for items $x_0, \ldots, x_{n-1}$ and search for k



- How many possible outcomes are there?
- What does that tell us about the height of the decision tree?

Outline

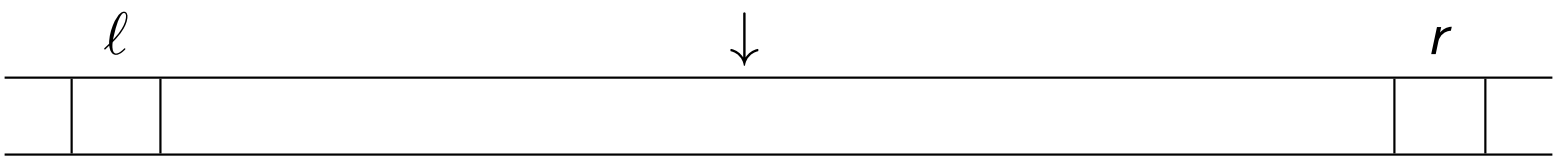
Towards interpolation search

We can match the lower bound asymptotically in a *sorted array*.

binary-search(A, n, k)

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m]$ equals $k)$ **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

binary-search: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lceil \frac{1}{2}(r-\ell-1) \rceil$



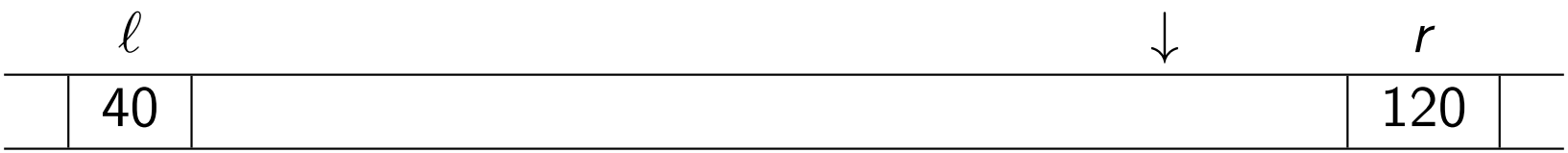
Towards interpolation search

We can match the lower bound asymptotically in a *sorted array*.

binary-search(A, n, k)

1. $\ell \leftarrow 0,\ r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m]$ equals $k)$ **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

binary-search: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lceil \frac{1}{2}(r-\ell-1) \rceil$



Question: If keys are *numbers*, where would you expect key $k = 100$?

Interpolation search

- Code very similar to binary search, but compare at index

$$\ell + \left\lceil \frac{\overbrace{k - A[\ell]}^{\text{distance from left key}}}{\underbrace{A[r] - A[\ell]}_{\text{distance between left and right keys}}} \underbrace{(r - \ell - 1)}_{\#\text{ unknown keys in range}} \right\rceil$$

- Need a few extra tests to avoid crash during computation of m.

interpolation-search$(A, n \leftarrow A.\textit{size}, k)$

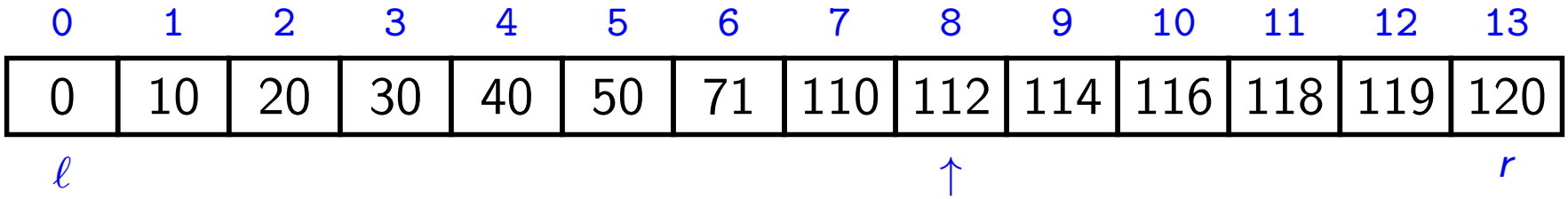
1. $\ell \leftarrow 0,\ r \leftarrow n - 1$
2. **while** $(\ell \leq r)$
3. **if** $(k < A[\ell]$ or $k > A[r])$ **return** "not found"
4. **if** $(k = A[r])$ **then return** "found at $A[r]$"
5. $m \leftarrow \ell + \left\lceil \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell - 1) \right\rceil$
6. **if** ($A[m]$ equals k) **then return** "found at $A[m]$"
7. **else if** $(A[m] < k)$ **then** $\ell \leftarrow m + 1$
8. **else** $r \leftarrow m - 1$

Interpolation search: Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	10	20	30	40	50	71	110	112	114	116	118	119	120

interpolation-search(A[0..13],14,71):

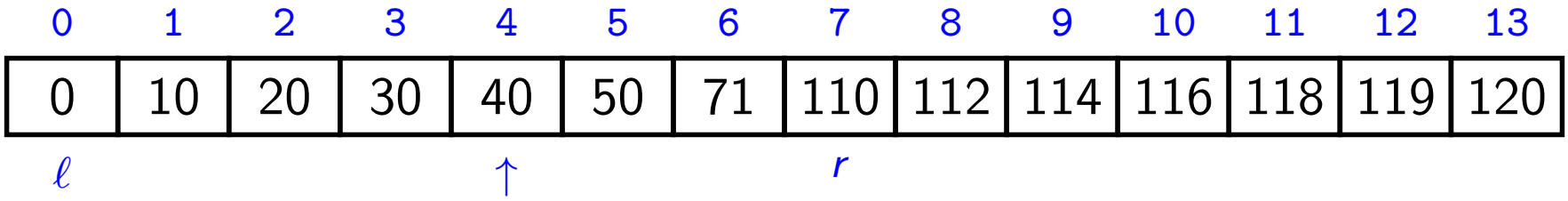
Interpolation search: Example



interpolation-search(A[0..13],14,71):

- $\ell = 0,\ r = n - 1 = 13,\ m = \ell + \lceil \frac{71-0}{120-0}(13-0-1) \rceil = \ell + 8 = 8$

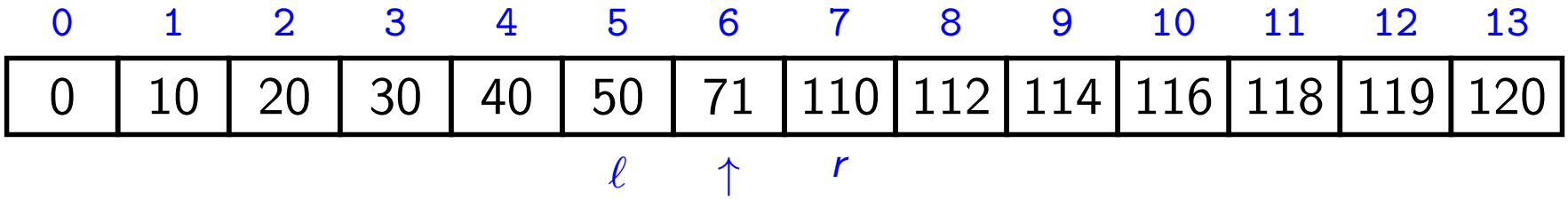
Interpolation search: Example



interpolation-search(A[0..13],14,71):

- $\ell = 0,\ r = n - 1 = 13,\ m = \ell + \lceil \frac{71-0}{120-0}(13-0-1) \rceil = \ell + 8 = 8$
- $\ell = 0,\ r = 7,\ m = \ell + \lceil \frac{71-0}{110-0}(7-0-1) \rceil = \ell + 4 = 4$

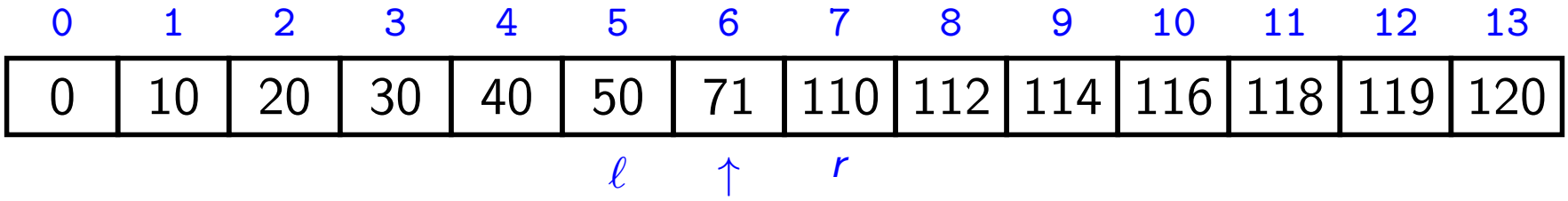
Interpolation search: Example



interpolation-search(A[0..13],14,71):

- $\ell = 0,\ r = n - 1 = 13,\ m = \ell + \lceil \frac{71-0}{120-0}(13-0-1) \rceil = \ell + 8 = 8$
- $\ell = 0,\ r = 7,\ m = \ell + \lceil \frac{71-0}{110-0}(7-0-1) \rceil = \ell + 4 = 4$
- $\ell = 5,\ r = 7,\ m = \ell + \lceil \frac{71-50}{110-50}(7-5-1) \rceil = \ell + 1 = 6$, found at $A[6]$

Interpolation search: Example



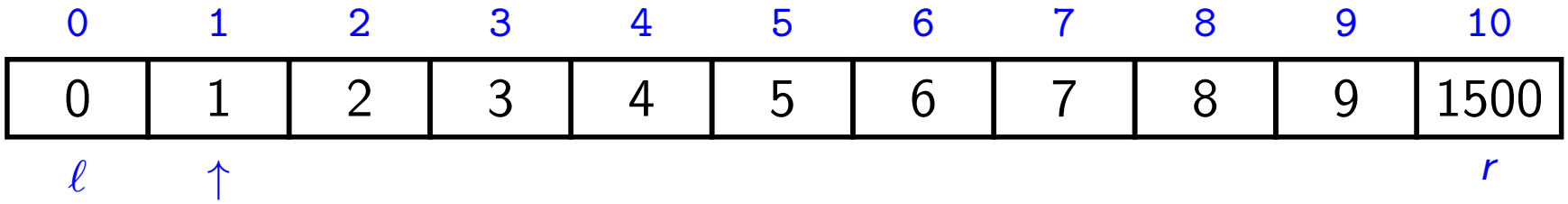
interpolation-search(A[0..13],14,71):

- $\ell = 0$, $r = n - 1 = 13$, $m = \ell + \lceil \frac{71-0}{120-0}(13-0-1) \rceil = \ell + 8 = 8$
- $\ell = 0$, $r = 7$, $m = \ell + \lceil \frac{71-0}{110-0}(7-0-1) \rceil = \ell + 4 = 4$
- $\ell = 5$, $r = 7$, $m = \ell + \lceil \frac{71-50}{110-50}(7-5-1) \rceil = \ell + 1 = 6$, found at $A[6]$

If instead we had $A[6] = 72$:

- $\ell = 5 = r$, exit at line 3 with "not found"

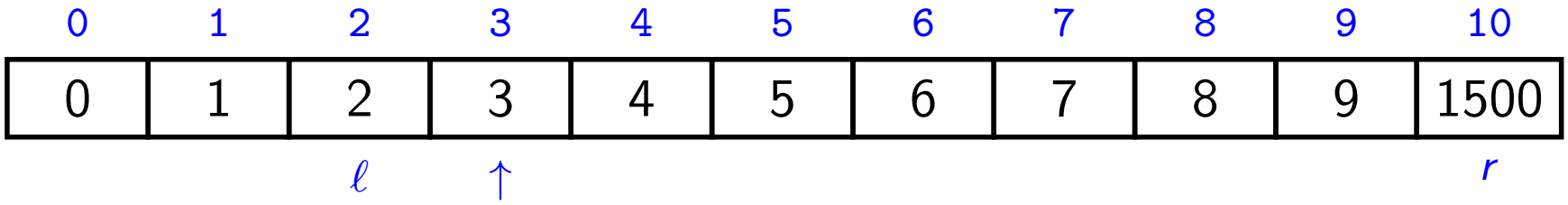
Interpolation search: Example 2



interpolation-search(A[0..10],10):

- $\ell = 0,\ r = n - 1 = 10,\ m = \ell + \lceil \frac{10-0}{1500-0}(10-0-1) \rceil = \ell + 1 = 1$

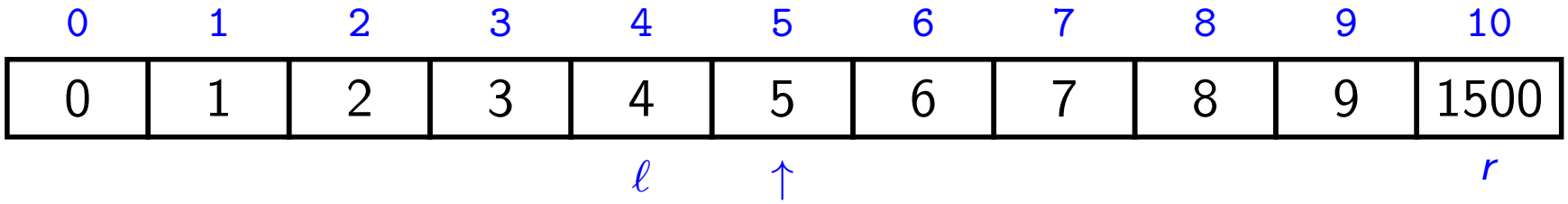
Interpolation search: Example 2



interpolation-search(A[0..10],10):

- $\ell = 0,\ r = n - 1 = 10,\ m = \ell + \lceil \frac{10-0}{1500-0}(10-0-1) \rceil = \ell + 1 = 1$
- $\ell = 2,\ r = 10,\ m = \ell + \lceil \frac{10-2}{1500-2}(10-2-1) \rceil = \ell + 1 = 3$

Interpolation search: Example 2



interpolation-search(A[0..10],10):

- $\ell = 0,\ r = n - 1 = 10,\ m = \ell + \lceil \frac{10-0}{1500-0}(10-0-1) \rceil = \ell + 1 = 1$
- $\ell = 2,\ r = 10,\ m = \ell + \lceil \frac{10-2}{1500-2}(10-2-1) \rceil = \ell + 1 = 3$
- $\ell = 4,\ r = 10,\ m = \ell + \lceil \frac{10-2}{1500-4}(10-4-1) \rceil = \ell + 1 = 5$

Interpolation search: Example 2

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500

interpolation-search(A[0..10],10):

- $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \lceil \frac{10-0}{1500-0}(10-0-1) \rceil = \ell + 1 = 1$
- $\ell = 2$, $r = 10$, $m = \ell + \lceil \frac{10-2}{1500-2}(10-2-1) \rceil = \ell + 1 = 3$
- $\ell = 4$, $r = 10$, $m = \ell + \lceil \frac{10-2}{1500-4}(10-4-1) \rceil = \ell + 1 = 5$
- ... in the worst case this can be very slow ($\Theta(n)$ time)

Interpolation search: Example 2

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500

interpolation-search(A[0..10],10):

- $\ell = 0,\ r = n - 1 = 10,\ m = \ell + \lceil \frac{10-0}{1500-0}(10-0-1) \rceil = \ell + 1 = 1$
- $\ell = 2,\ r = 10,\ m = \ell + \lceil \frac{10-2}{1500-2}(10-2-1) \rceil = \ell + 1 = 3$
- $\ell = 4,\ r = 10,\ m = \ell + \lceil \frac{10-2}{1500-4}(10-4-1) \rceil = \ell + 1 = 5$
- ... in the worst case this can be very slow ($\Theta(n)$ time)

But it works well on average:

- Can show (difficult): $\boxed{T^{\text{avg}}(n) \leq T^{\text{avg}}(\sqrt{n}) + \Theta(1).}$
- This resolves to $T^{\text{avg}}(n) \in O(\log \log n)$.

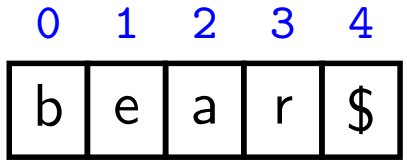
Outline

Review: Words

Scenario: Keys in dictionary are *words*. Need brief review.

Words (= strings): sequences of characters over alphabet Σ

- Typical alphabets: $\{0, 1\}$ ($\rightarrow$bitstrings), ASCII, $\{C, G, T, A\}$
- Stored in an array: $w[i]$ gets ith character (for $i = 0, 1, \ldots$)



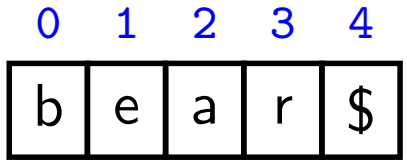
- Words have end-sentinel \$ (sometimes not shown)
- $w.size = |w| = \#$ non-sentinel characters: $|\texttt{bear\$}|=4$.

Review: Words

Scenario: Keys in dictionary are *words*. Need brief review.

Words (= strings): sequences of characters over alphabet Σ

- Typical alphabets: $\{0, 1\}$ ($\rightarrow$bitstrings), ASCII, $\{C, G, T, A\}$
- Stored in an array: $w[i]$ gets ith character (for $i = 0, 1, \ldots$)



- Words have end-sentinel \$ (sometimes not shown)
- $w.size = |w| = \#$ non-sentinel characters: $|\texttt{bear\$}|=4$.

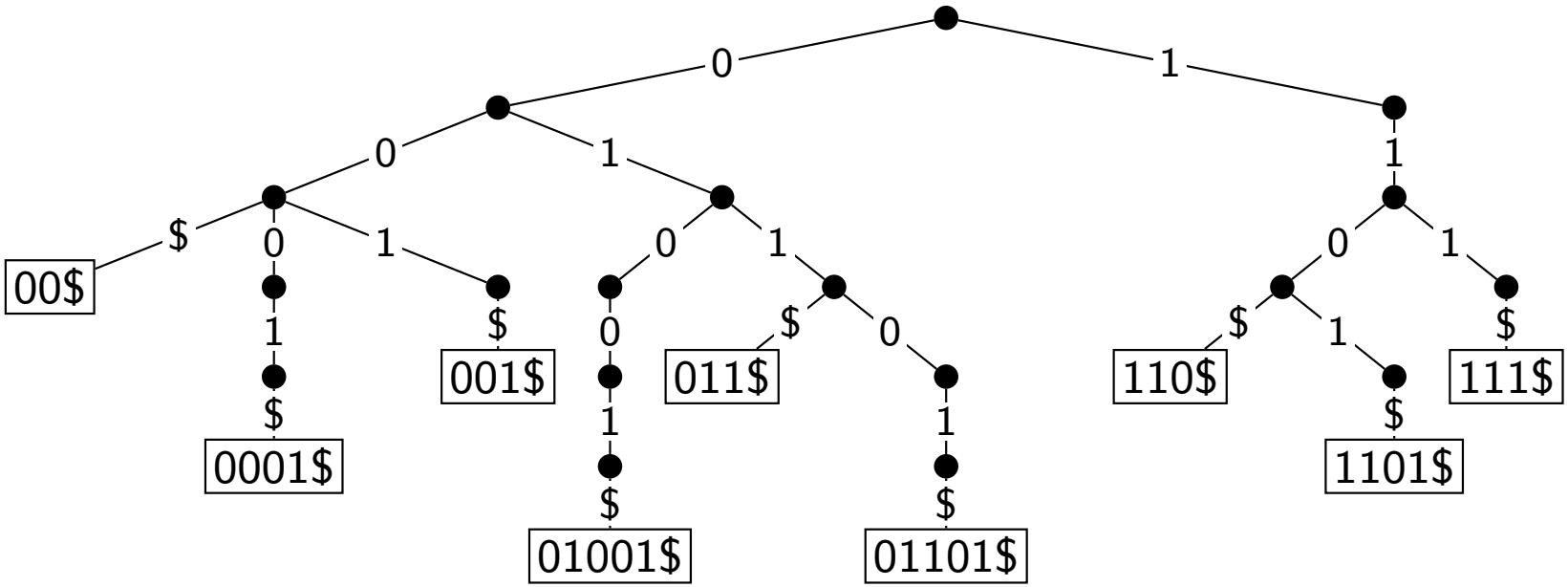
Should know:

- prefix, suffix, substring
- Sort words **lexicographically**: $\texttt{be\$} <_{\text{lex}} \texttt{bear\$} <_{\text{lex}} \texttt{beer\$}$
  This is different from sorting numbers: $\texttt{001\$} <_{\text{lex}} \texttt{010\$} <_{\text{lex}} \texttt{1\$}$

Tries: Introduction

Trie (also know as **radix tree**): A dictionary for bitstrings.

- Comes from retrieval, but pronounced "try"
- A tree based on *bitwise comparisons*: Edge labelled with corresponding bit
- Similar to *radix sort*: use individual bits, not the whole key
- Due to end-sentinels, all key-value pairs are at leaves.



Tries: Search

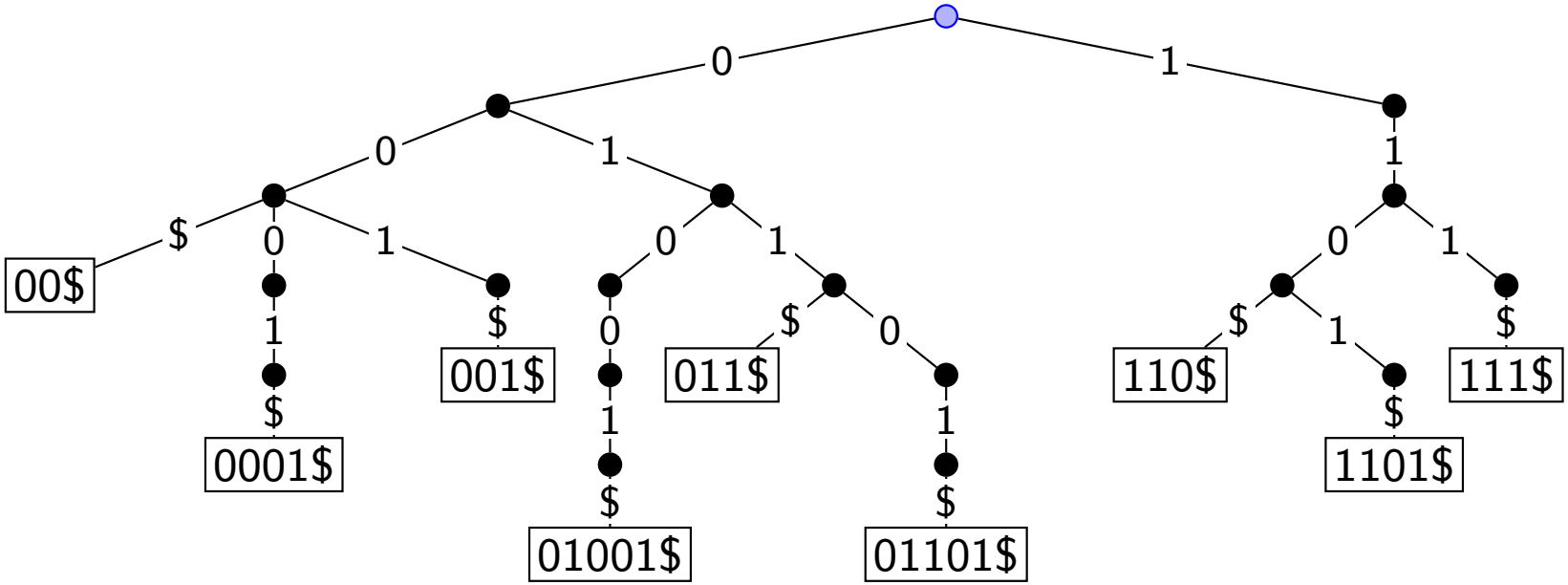
- Follow links that correspond to current bits in w, keeping track of current depth d
- Repeat until no such link or w found at a leaf

Similar as for skip lists, we find search-path P separately first.

Trie::get-path-to(w)
Output: Stack with all ancestors of where w would be stored
1. $P \leftarrow$ empty stack; $z \leftarrow root$; $d \leftarrow 0$; P.*push*(z)
2. **while** $d \leq |w|$
3. **if** z has a child-link labelled with $w[d]$
4. $z \leftarrow$ child at this link; d++; P.*push*(z)
5. **else break**
6. **return** P

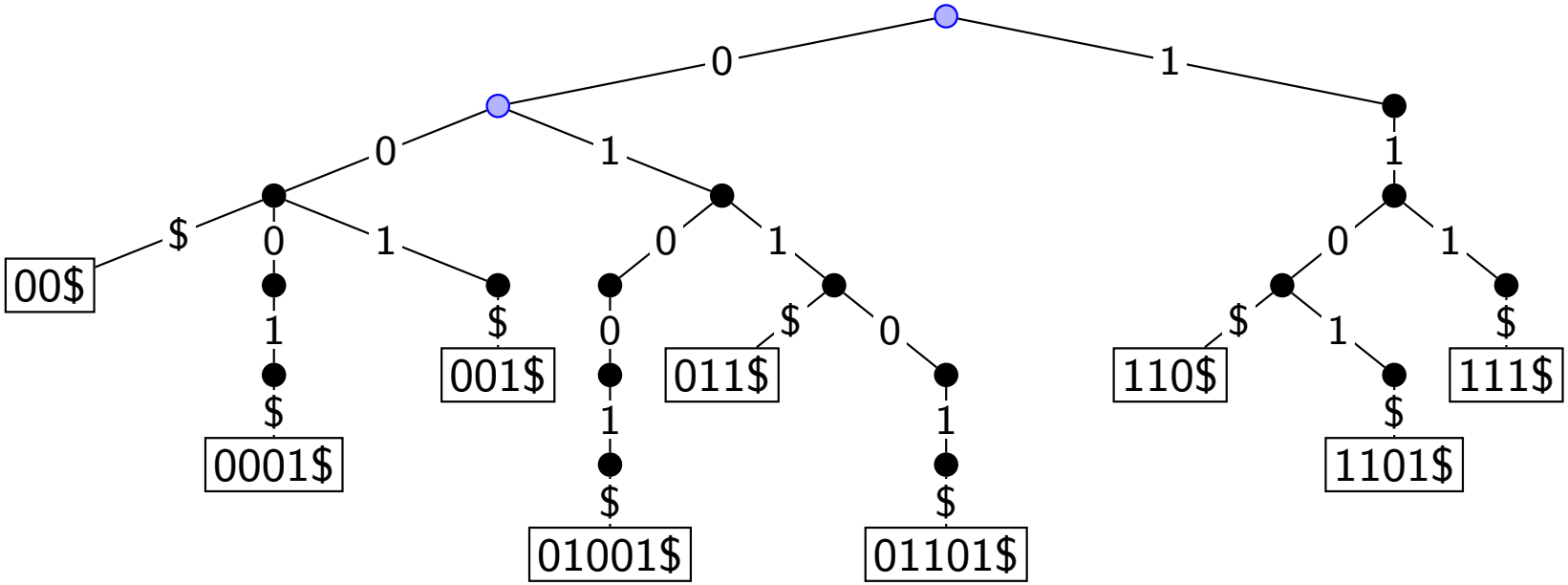
Tries search: Example

Example: Trie::search(011$)



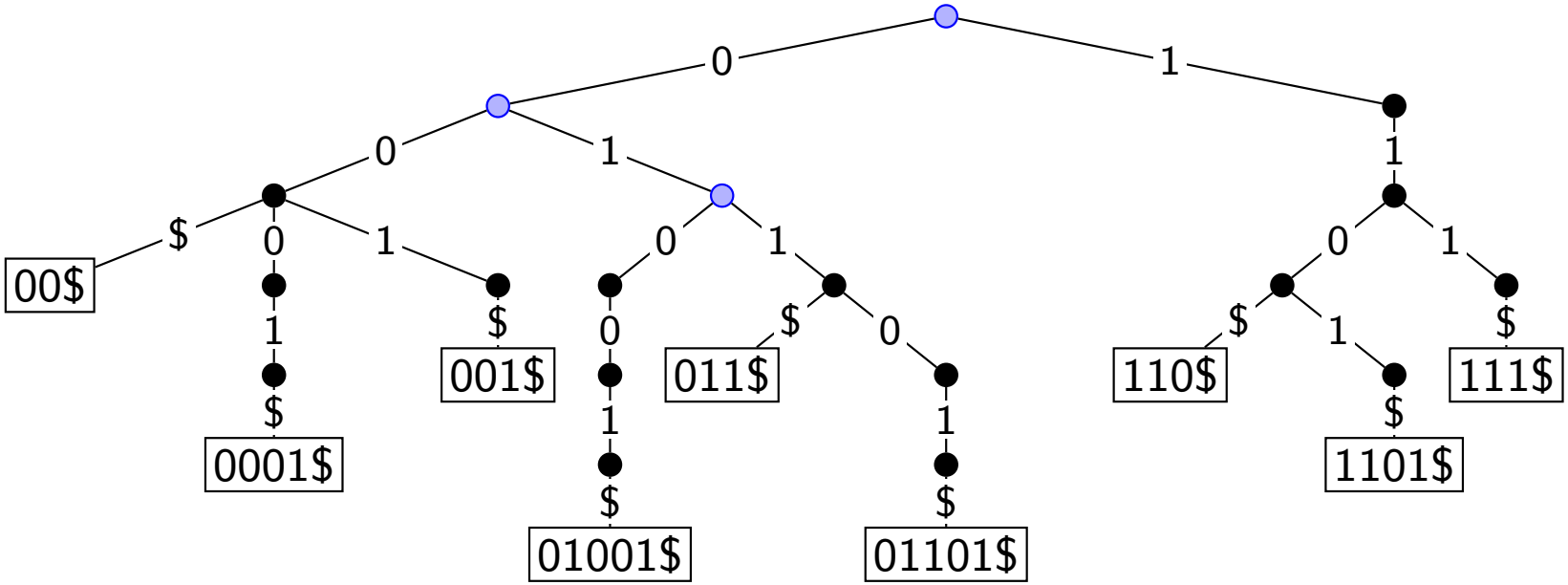
Tries search: Example

Example: Trie::search(011$)



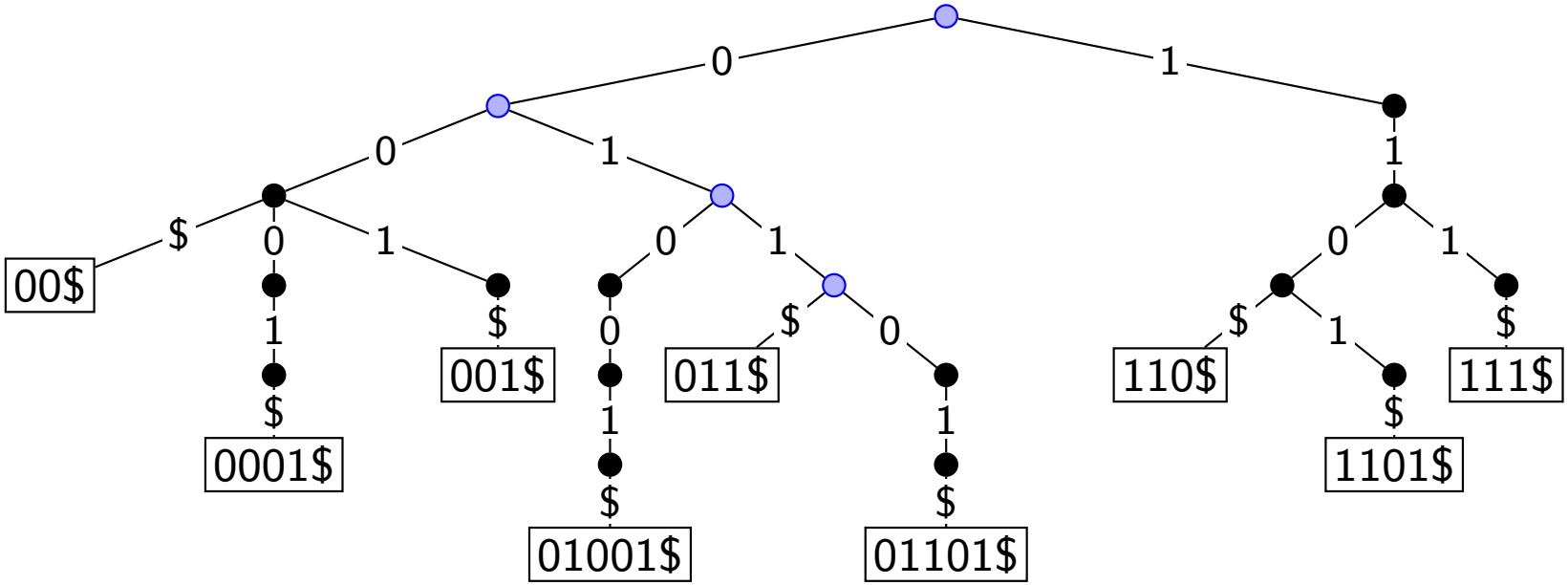
Tries search: Example

Example: Trie::search(011$)



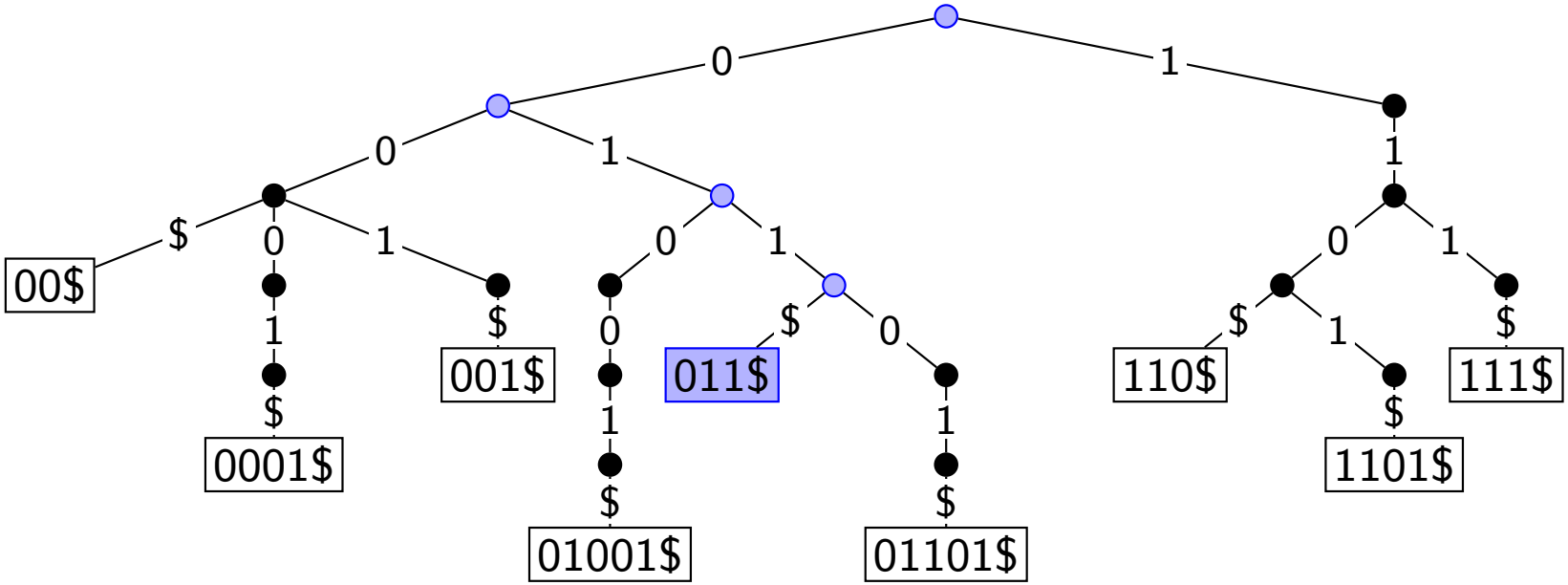
Tries search: Example

Example: Trie::search(011$)



Tries search: Example

Example: Trie::search(011$)



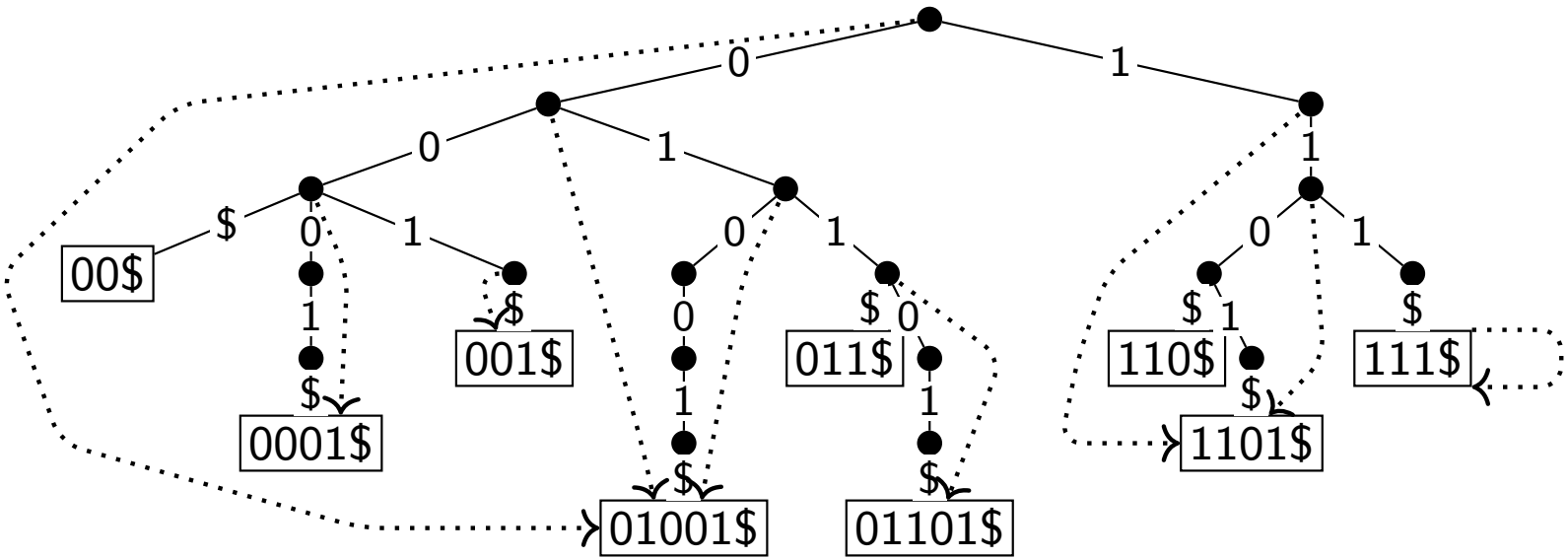
Trie::search(w)

1. $P \leftarrow$ *get-path-to*(w), $z \leftarrow P$.*top*
2. **if** (z is not a leaf) **then**
3. **return** "not found, would be in sub-trie of z"
4. **return** key-value pair at z

Tries: Prefix-search

For later applications of tries, we want another search-operation:

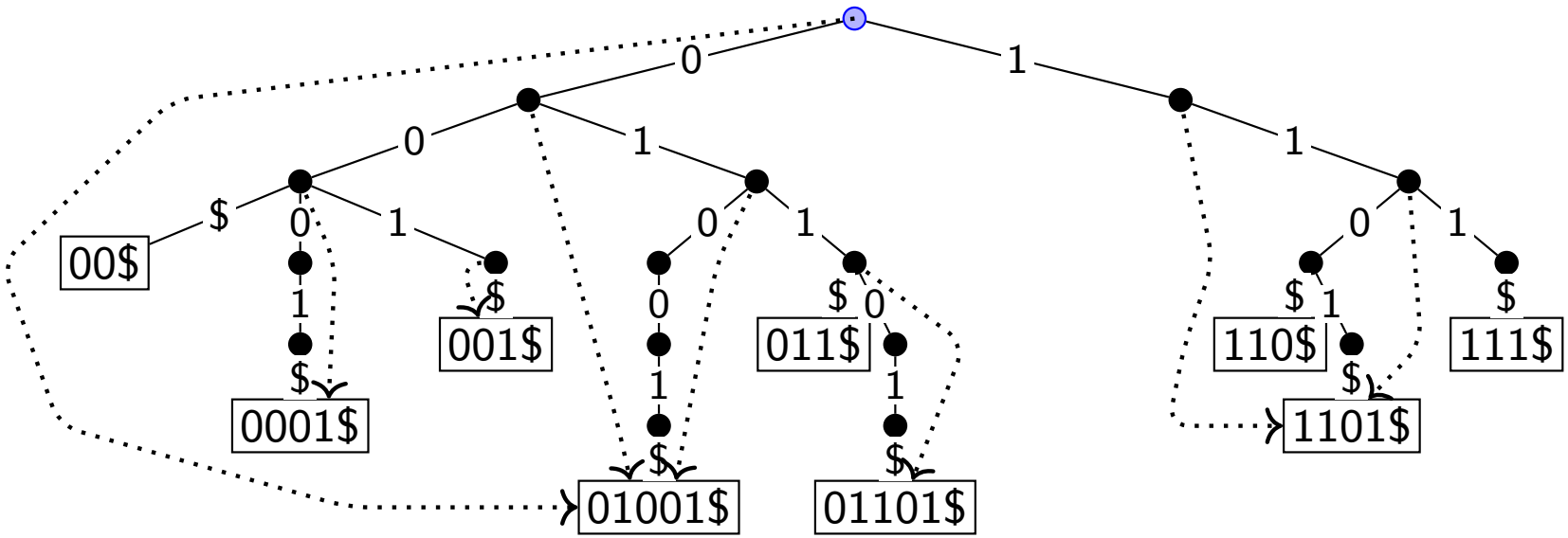
- *prefix-search*(w): Find **extension**(word for which w is a prefix).
- Testing whether extension exists is easy (how?)
- To find extension quickly, we need **leaf-references**
  - Every node z stores reference $z.\mathit{leaf}$ to a leaf in subtree
  - **Convention:** store leaf with longest word



(not all leaf-references are shown)

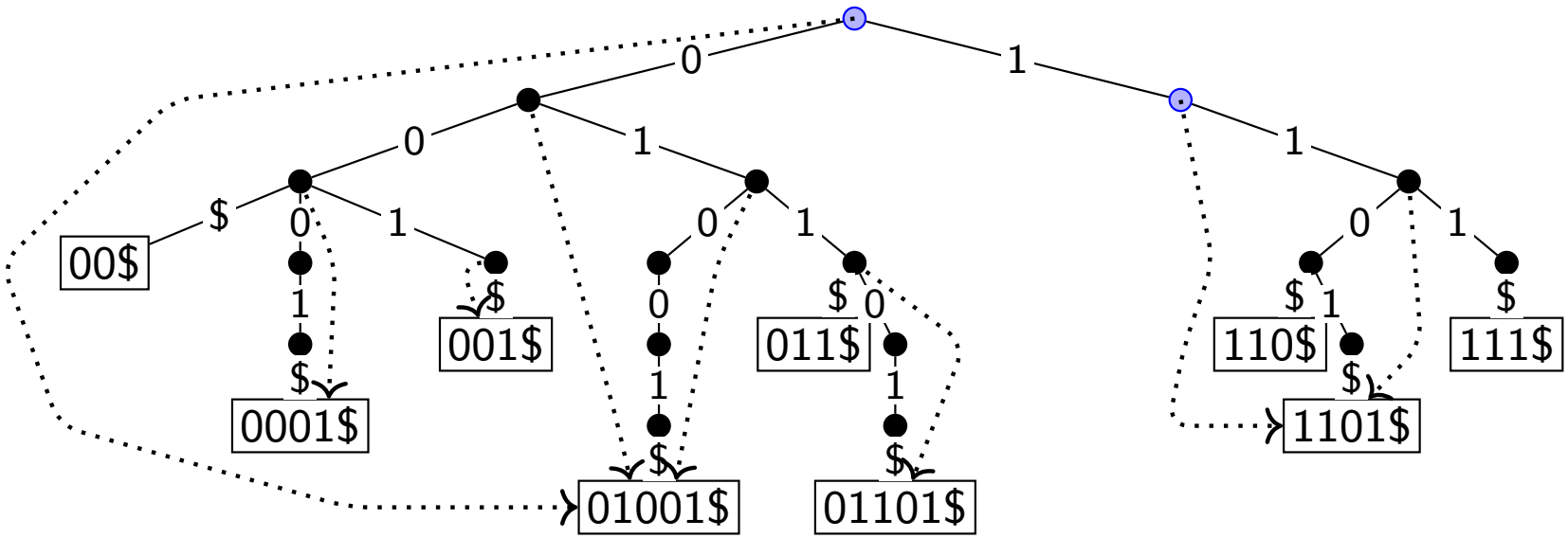
Tries prefix-search: Example

Example: Trie::prefix-search(11$)



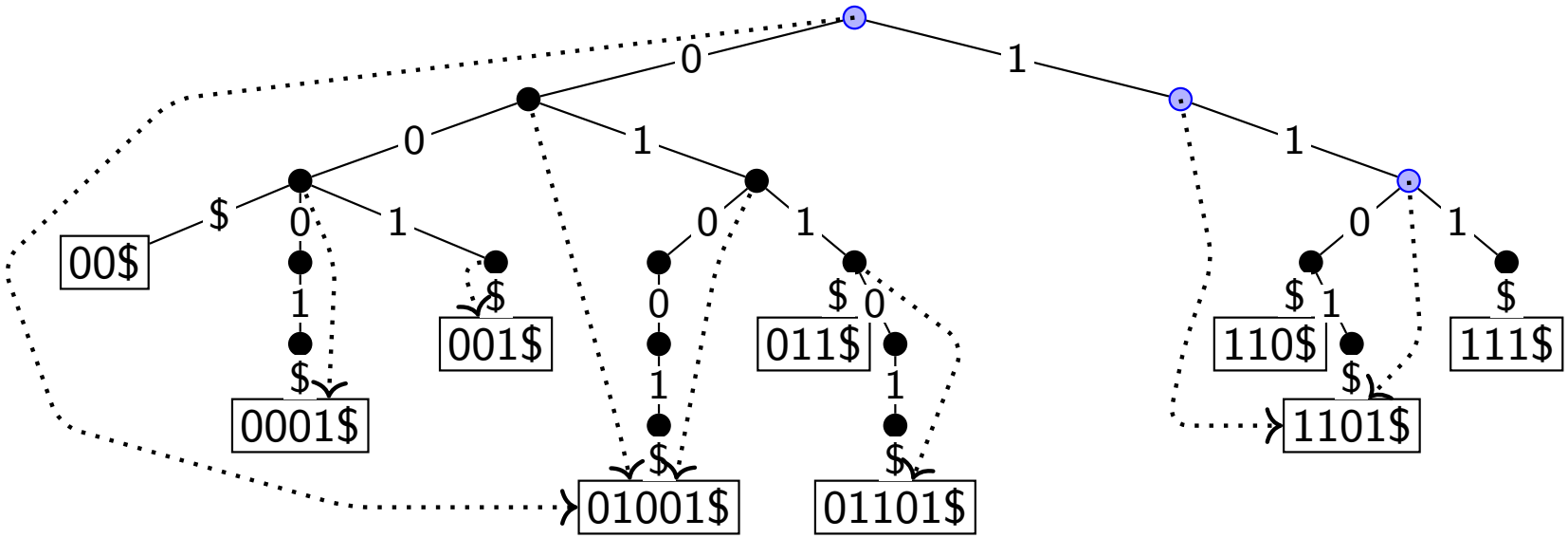
Tries prefix-search: Example

Example: Trie::prefix-search(11$)



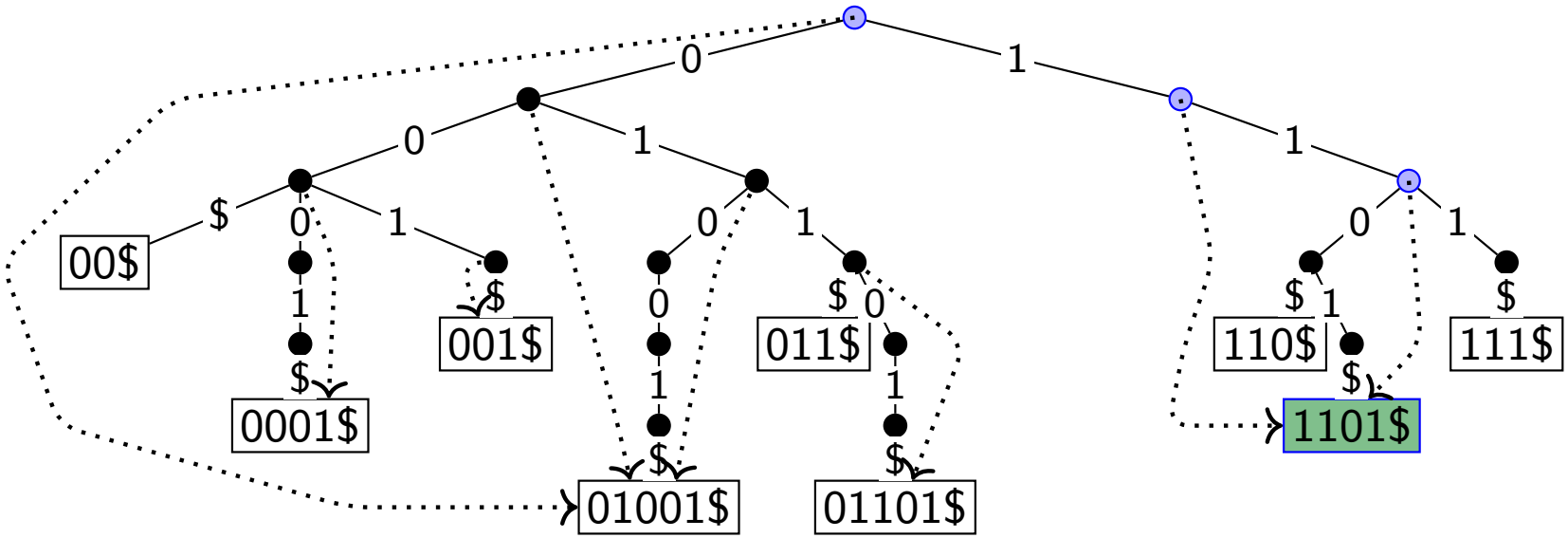
Tries prefix-search: Example

Example: Trie::prefix-search(11$)



Tries prefix-search: Example

Example: Trie::prefix-search(11$)

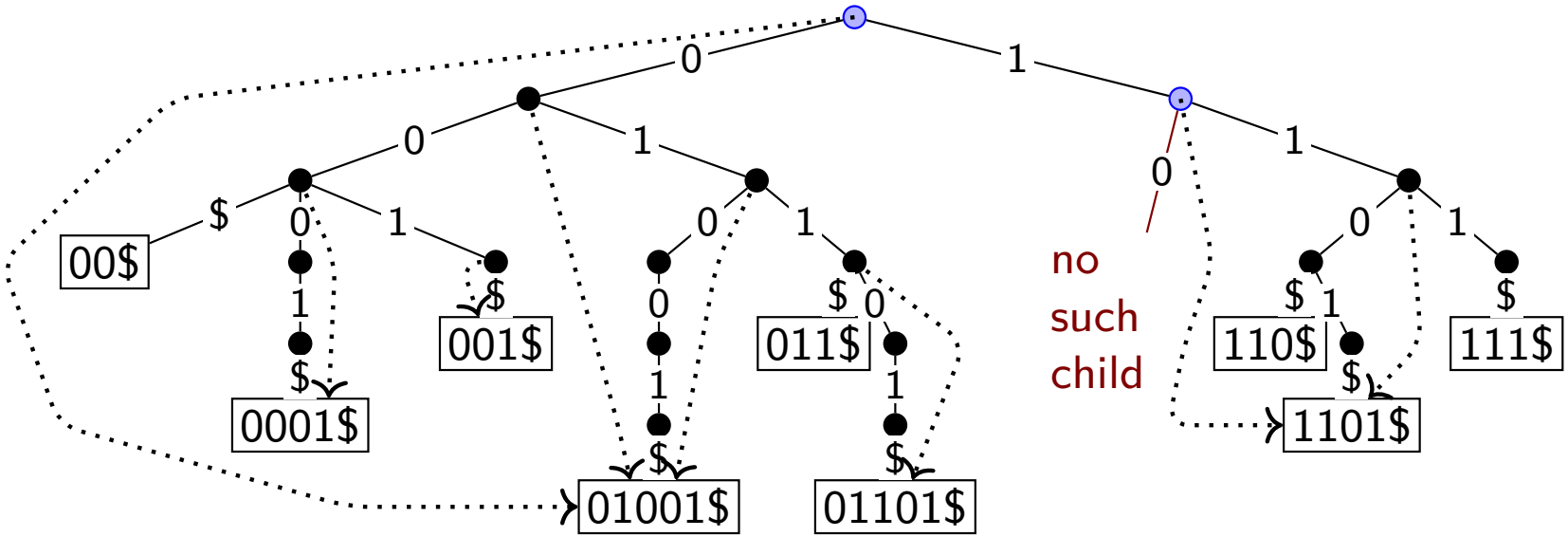


Trie::prefix-search(w)

1. $P \leftarrow$ *get-path-to*$(w[0..|w|-1])$ `// ignore end-sentinel`
2. **if** number of nodes on P is at most $|w|$
3. **return** "no extension of w found"
4. **return** P.*top*().*leaf*

Tries prefix-search: Example

Example: Trie::prefix-search(10$)



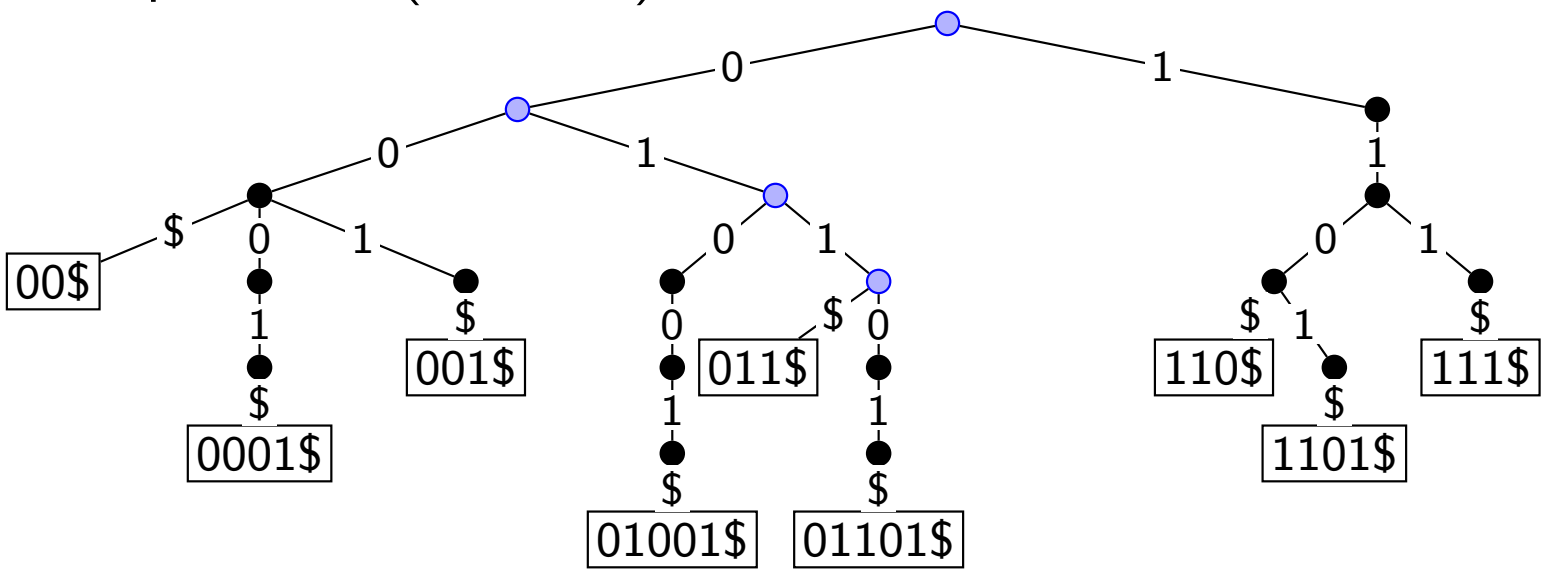
- Word 10$ has size 2.
- *get-path-to*(10) returns stack with two nodes.
- We need more than $|w|$ nodes on P to have an extension.

Tries: Insert

Trie::insert(w)

- $P \leftarrow$ *get-path-to*(w) gives ancestors that exist already,
- Expand the trie from P.*top*() by adding necessary nodes that correspond to extra bits of w.
- Update leaf-references (also at P if w is longer than previous leaves)

Example: *insert*(011101$)

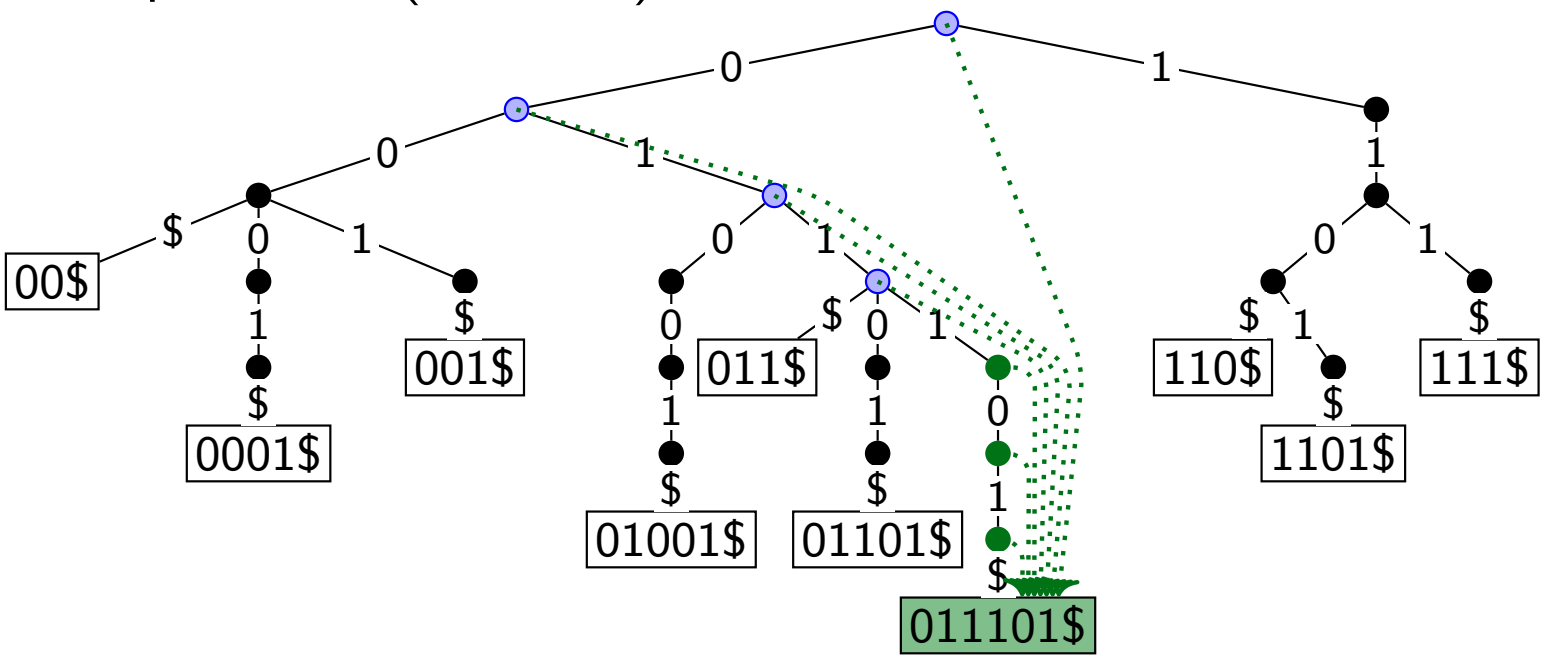


Tries: Insert

Trie::insert(w)

- $P \leftarrow$ *get-path-to*(w) gives ancestors that exist already,
- Expand the trie from P.*top*() by adding necessary nodes that correspond to extra bits of w.
- Update leaf-references (also at P if w is longer than previous leaves)

Example: *insert*(011101$)



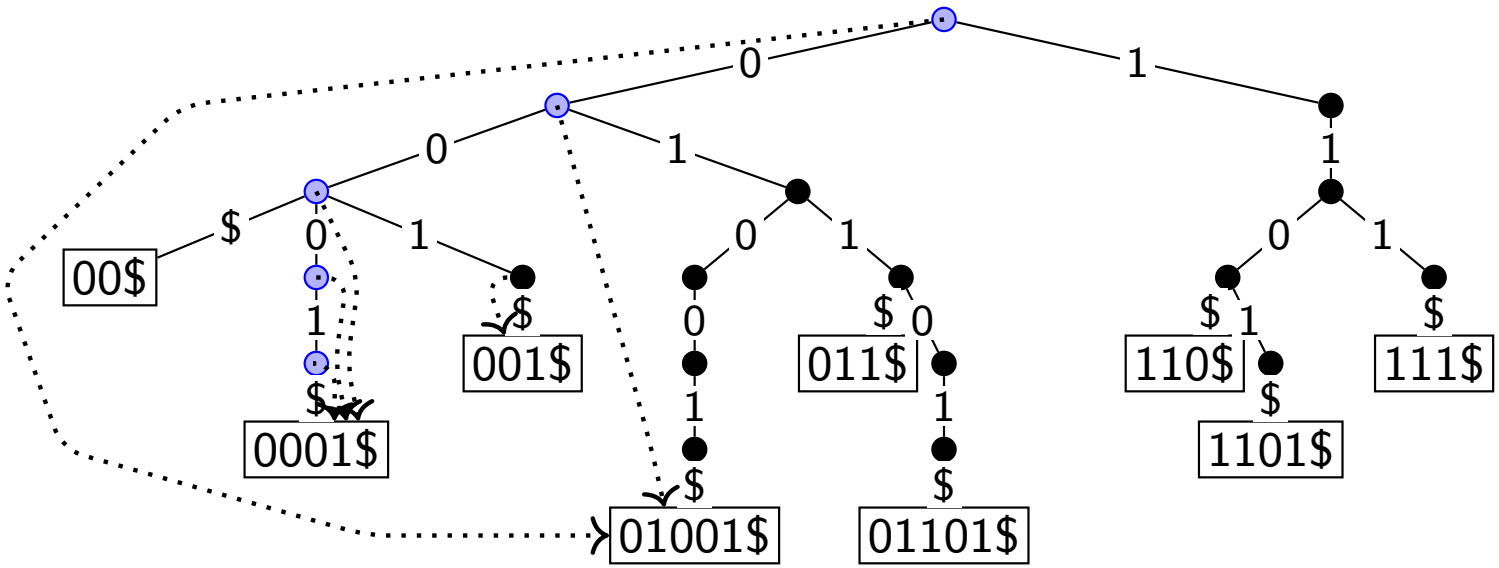
(only updated leaf-references are shown)

Tries: Delete

Trie::delete(w)

- $P \leftarrow$ *get-path-to*(w) gives all ancestors.
- Let ℓ be the leaf where w is stored
- Delete ℓ and nodes on P until ancestor that had two or more children.
- Update leaf-references on rest of P.
  (If $z \in P$ referred to ℓ, find new z.*leaf* from other children.)

Example: *trie::delete*(0001$)



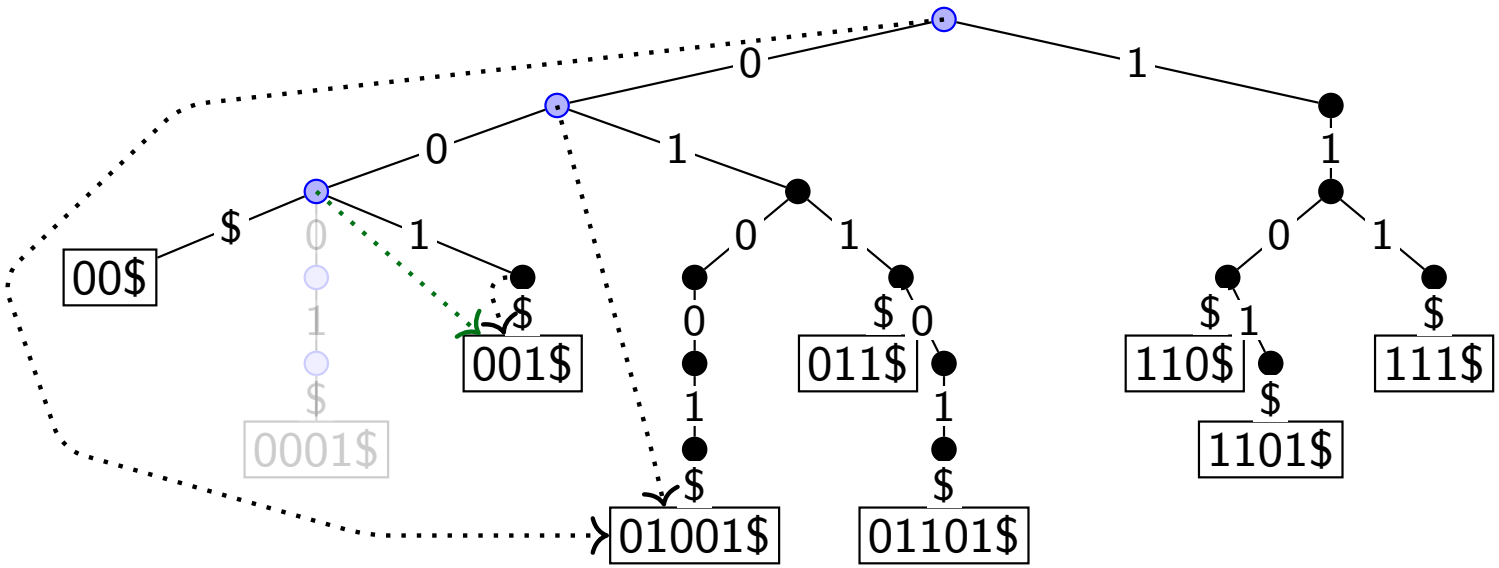
(only some leaf-references are shown)

Tries: Delete

Trie::delete(w)

- $P \leftarrow$ *get-path-to*(w) gives all ancestors.
- Let ℓ be the leaf where w is stored
- Delete ℓ and nodes on P until ancestor that had two or more children.
- Update leaf-references on rest of P.
  (If $z \in P$ referred to ℓ, find new z.*leaf* from other children.)

Example: *trie::delete*(0001$)



(only some leaf-references are shown)

Binary tries: Summary

search(w), *prefix-search*(w), *insert*(w), *delete*(w) all take time $\Theta(|w|)$.

- Search-time is *independent* of number n of words stored in the trie!
- Search-time is small for short words.

The trie for a given set of words is unique

(except for order of children and ties among leaf-references)

Binary tries: Summary

search(w), *prefix-search*(w), *insert*(w), *delete*(w) all take time $\Theta(|w|)$.

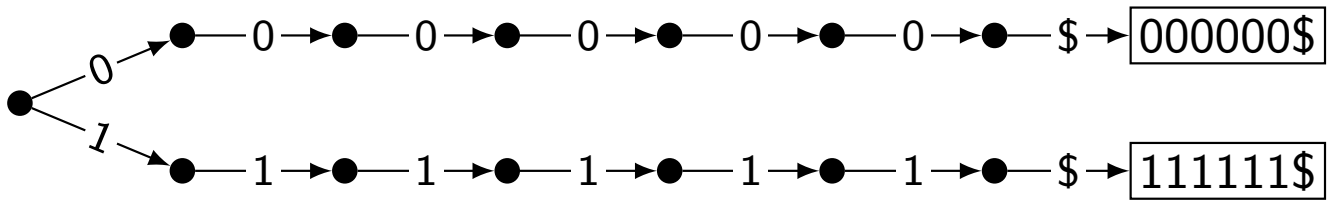
- Search-time is *independent* of number n of words stored in the trie!
- Search-time is small for short words.

The trie for a given set of words is unique

(except for order of children and ties among leaf-references)

Disadvantages:

- Tries can be wasteful with respect to space.

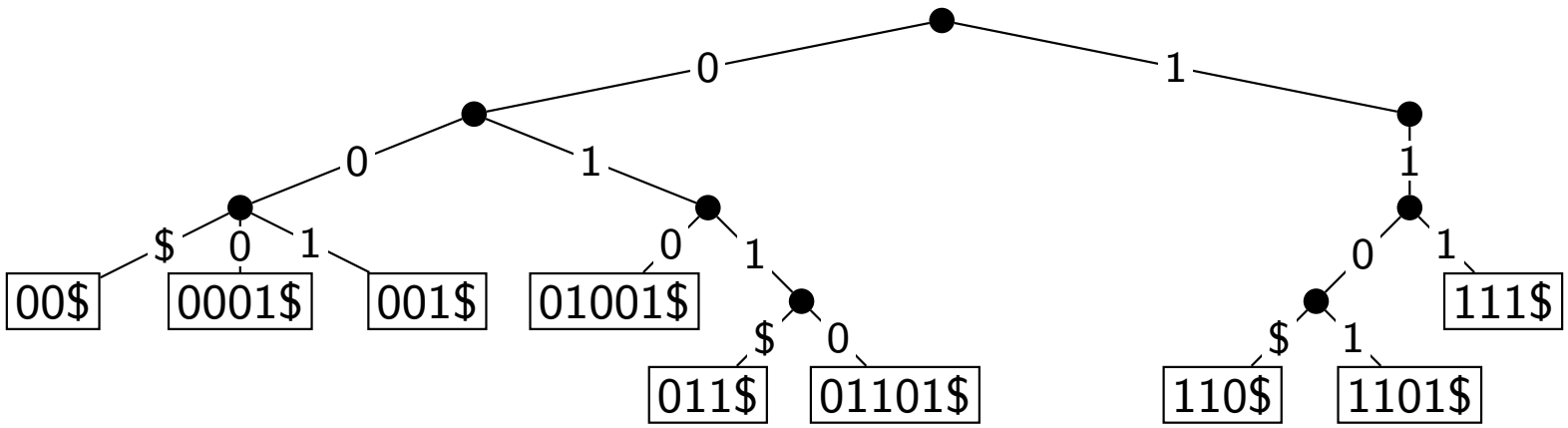


- Worst-case space is $\Theta(n \cdot (\text{maximum length of a word}))$
- What can we do to save space?

Variations of tries: Pruned tries

Pruned Trie: Stop adding nodes to trie as soon as the key is unique.

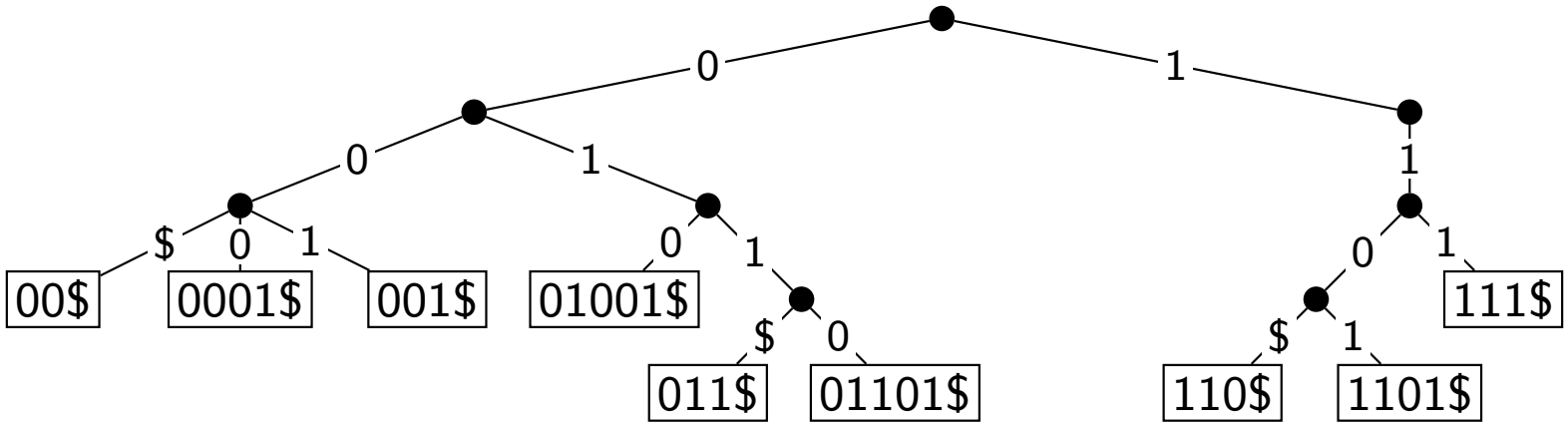
- A node has a child only if it has at least two descendants.
- Saves space if there are only few bitstrings that are long.



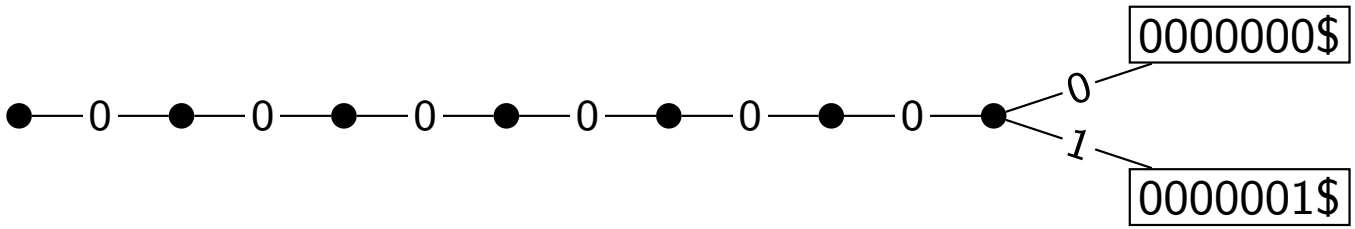
Variations of tries: Pruned tries

Pruned Trie: Stop adding nodes to trie as soon as the key is unique.

- A node has a child only if it has at least two descendants.
- Saves space if there are only few bitstrings that are long.



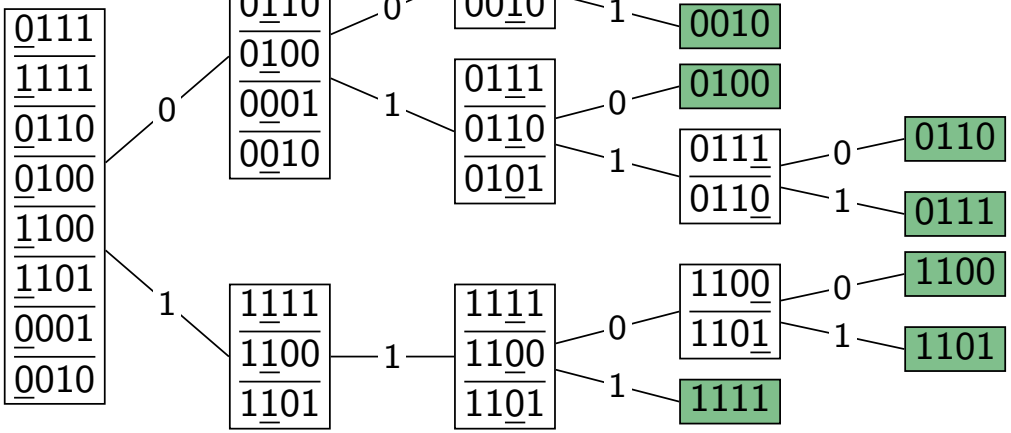
But the space can still be bad.

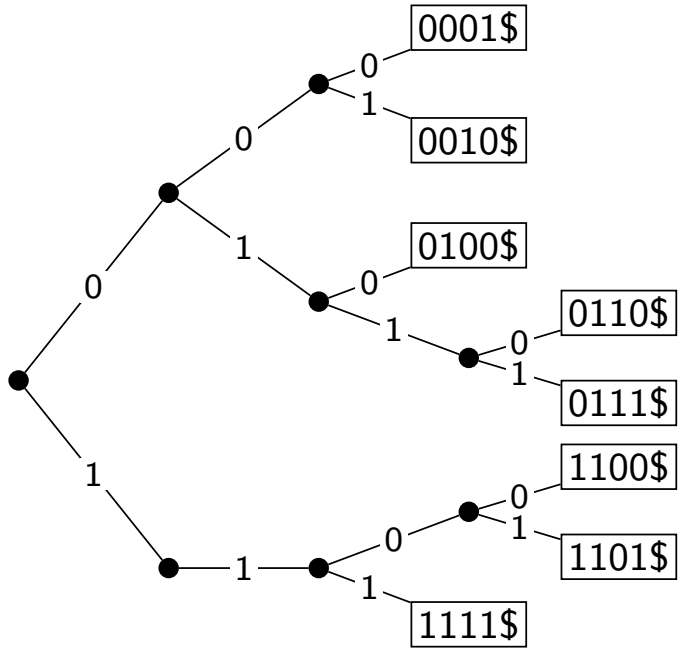


Pruned tries and MSD-radix sort

We have (implicitly) seen pruned tries before:

- For equal-length bitstrings:
  Pruned trie equals recursion tree of MSD radix-sort.

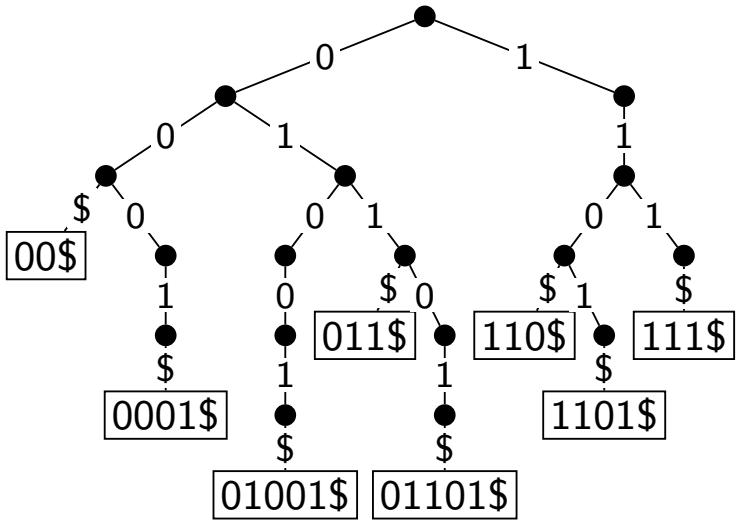


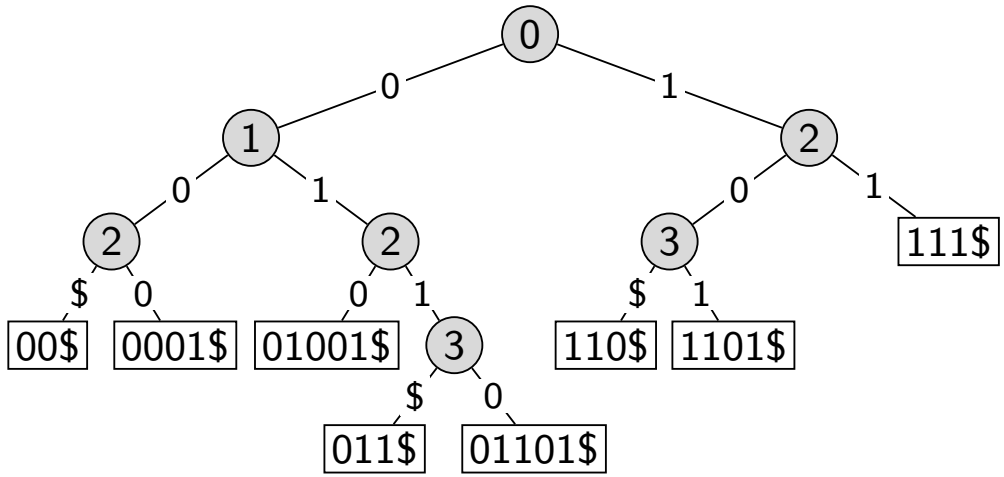


Compressed tries

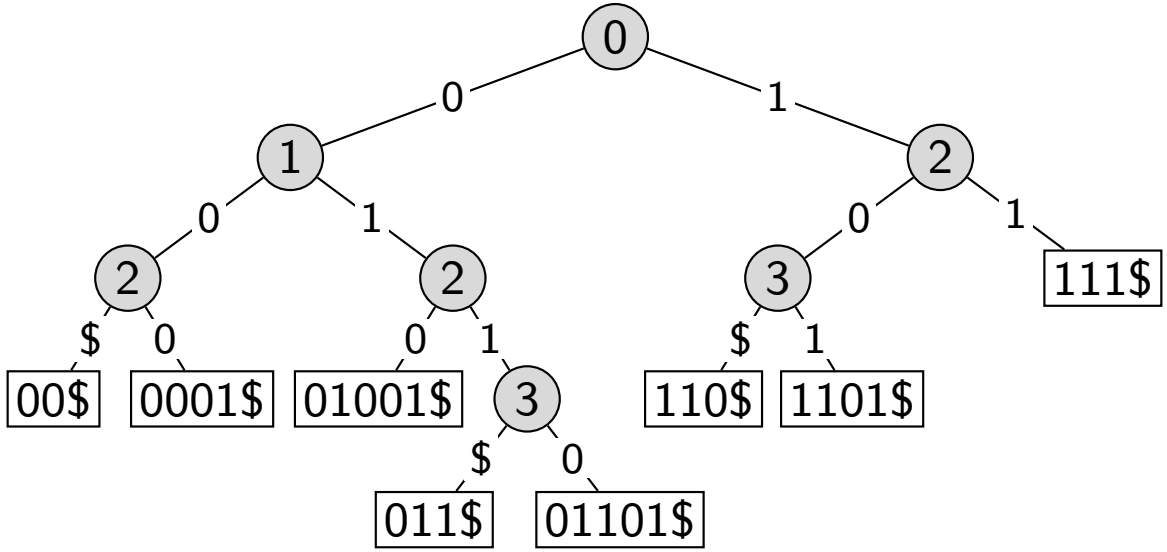
Another (important!) variation:

- Compress paths of nodes with only one child.
- Each node stores an *index*, corresponding to the level of the node in the uncompressed trie. (On level d, we searched for link with $w[d]$.)



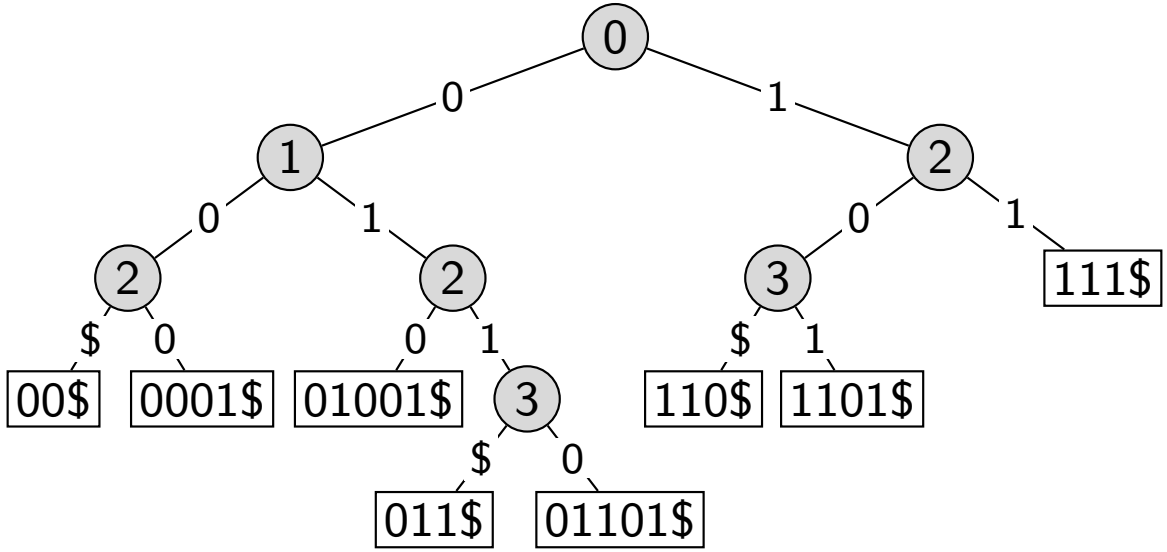


Compressed tries



- **Invariant:** At any node z where $d = z.\mathit{index}$,
  all words in the subtree rooted at z have the same initial d bits.

Compressed tries



- **Invariant:** At any node z where $d = z.\mathit{index}$,
  all words in the subtree rooted at z have the same initial d bits.
- **Observe:** Any compressed trie with n words has $O(n)$ nodes.
  - #nodes = # leaves + # internal node.
  - Every internal node has two or more children.
  - **Can show:** Therefore more leaves than internal nodes.
  - So #nodes $\leq 2n - 1$.

  In particular we use $O(n)$ auxiliary space.

Compressed tries: Search

- As for tries, follow links that corresponds to current bits in w
- Main difference: stored indices say which bits to compare.
- Also: must compare w to word found at the leaf (why?)

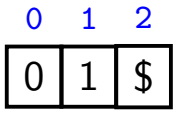
CompressedTrie::get-path-to(w)

1. $P \leftarrow$ empty stack; $z \leftarrow$ *root*; P.*push*(z)
2. **while** z is not a leaf and ($d \leftarrow z$.*index* $\leq |w|$) **do**
3. **if** (z has a child-link labelled with $w[d]$) **then**
4. $z \leftarrow$ child at this link; P.*push*(z)
5. **else break**
6. **return** P

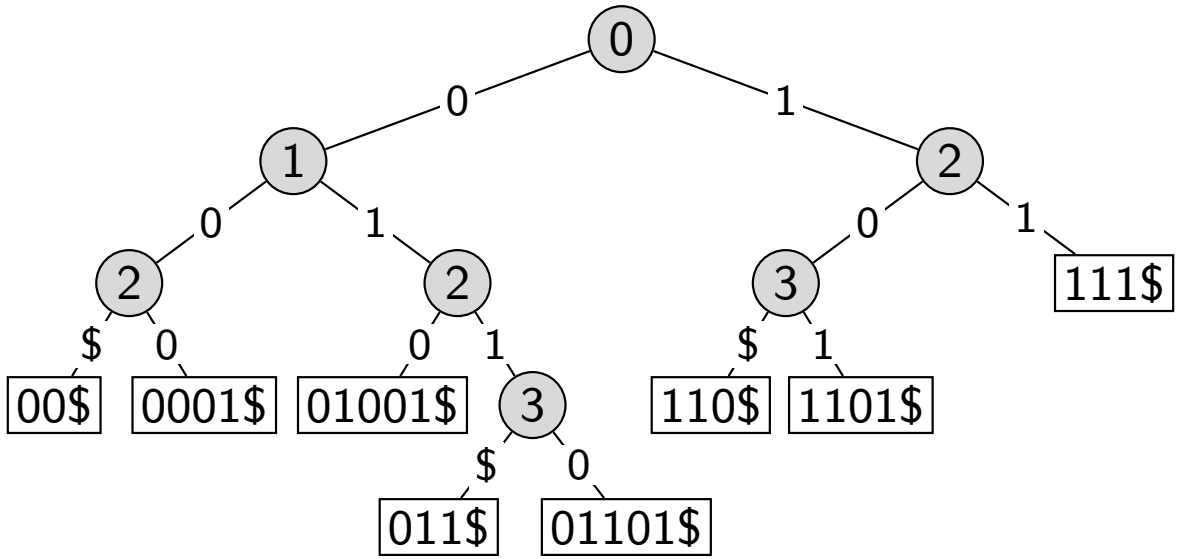
CompressedTrie::search(w)

1. $P \leftarrow$ *get-path-to*(w), $z \leftarrow P$.*top*
2. **if** (z is not a leaf or word stored at z is not w) **then**
3. **return** "not found"
4. **return** key-value pair at z

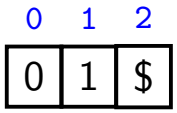
Compressed tries search: Example

Example 1: CompressedTrie::search(

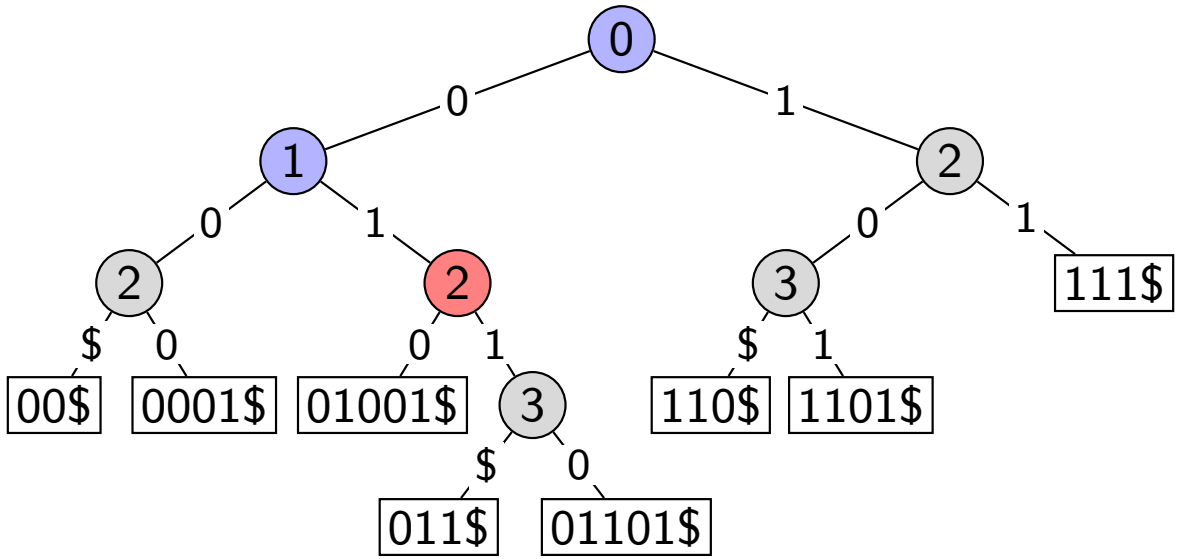
)



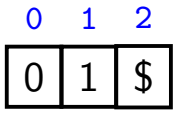
Compressed tries search: Example

Example 1: CompressedTrie::search(

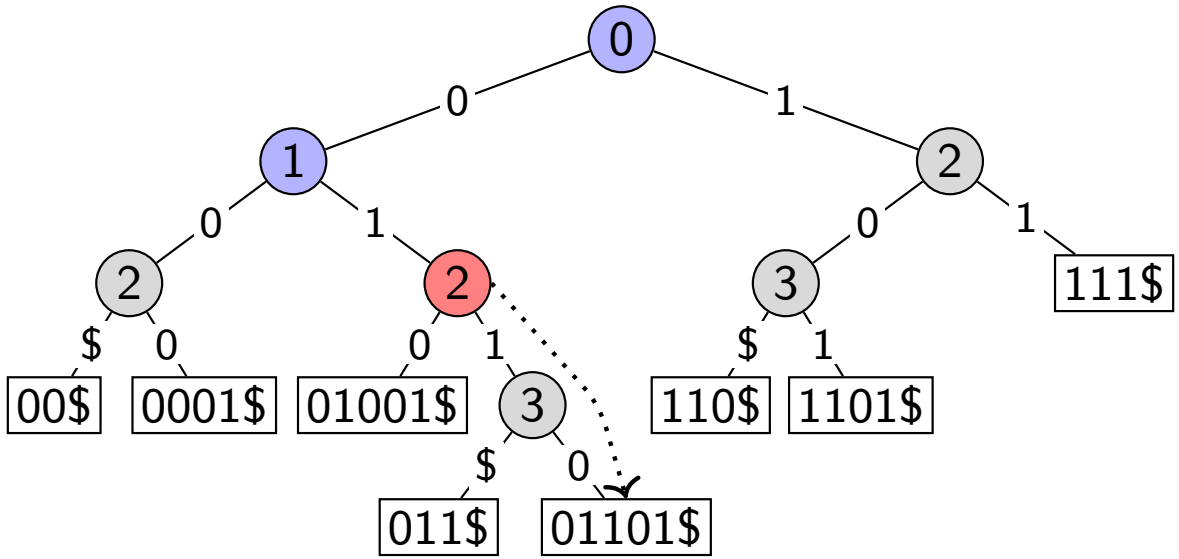
) unsuccessful (no $-child)



Compressed tries search: Example

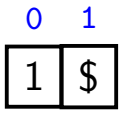
Example 1: CompressedTrie::search(

) unsuccessful (no $-child)

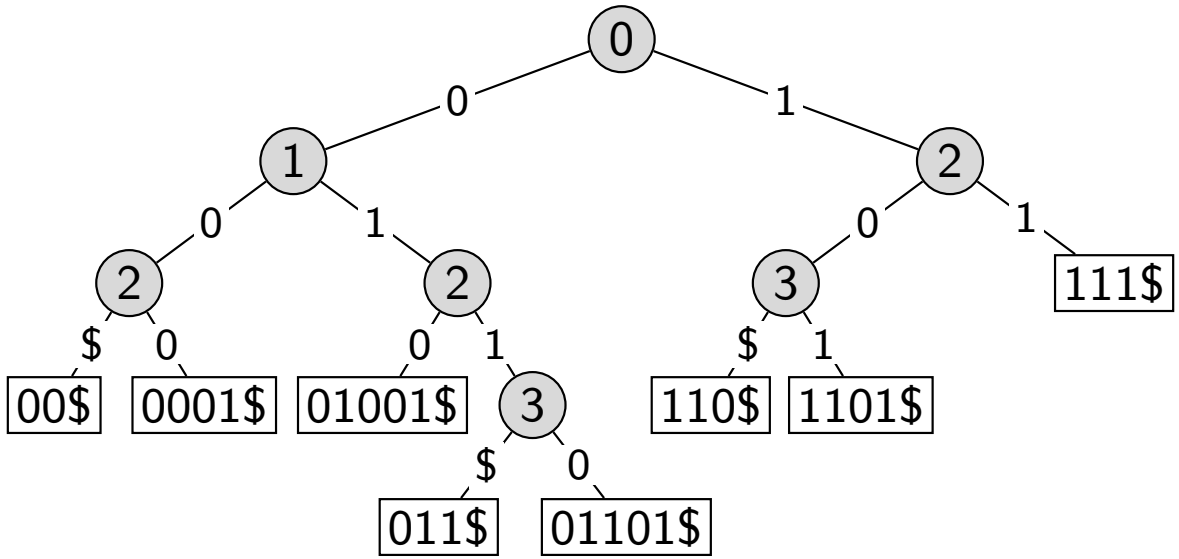


prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed tries search: Example

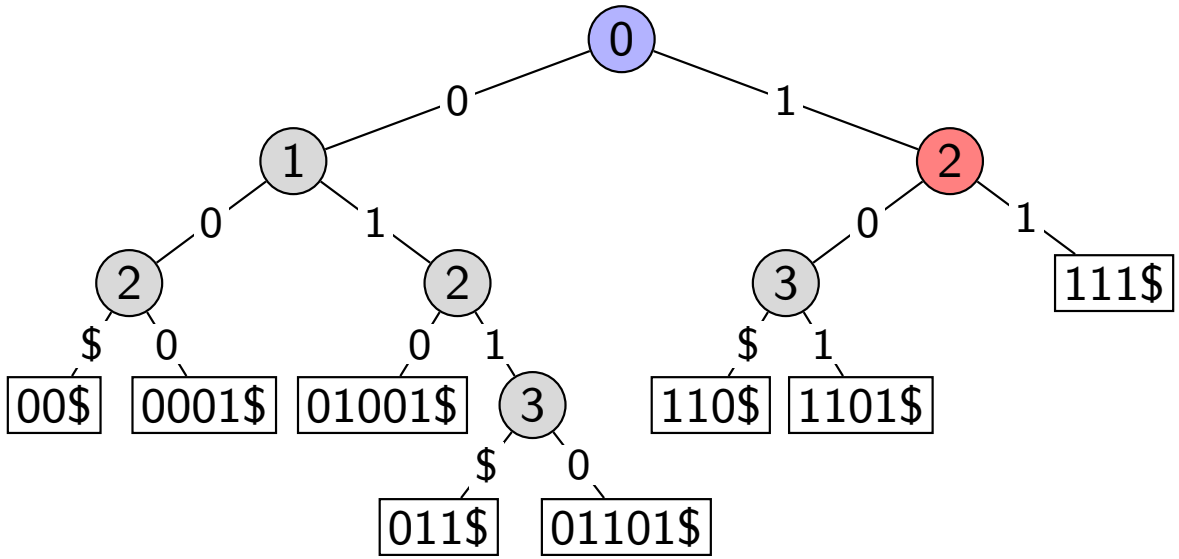
Example 2: CompressedTrie::search(

)

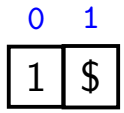


Compressed tries search: Example

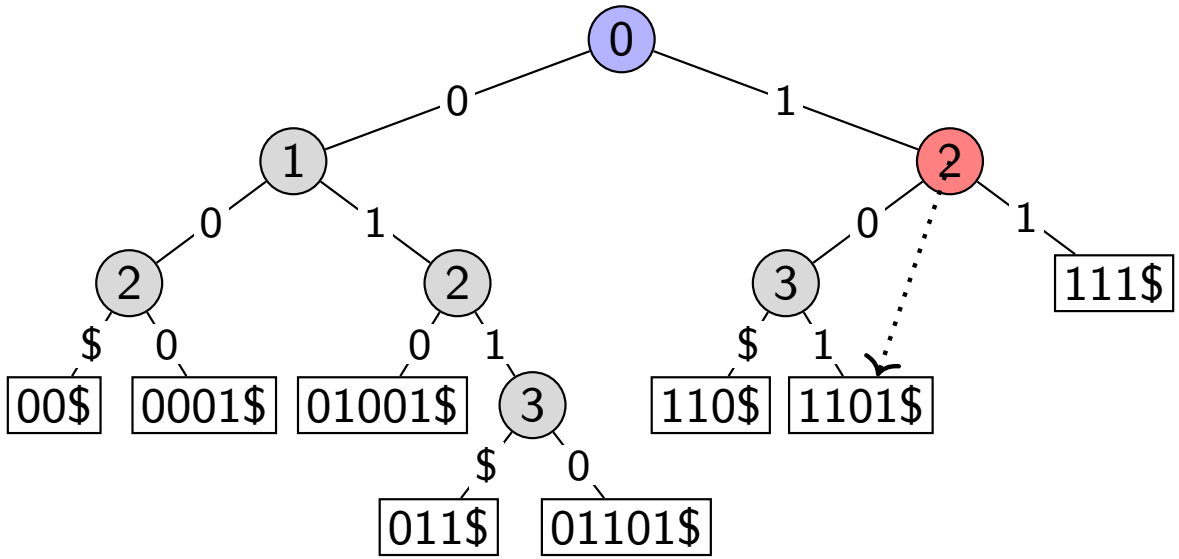
Example 2: CompressedTrie::search($\begin{array}{cc} 0 & 1 \\ \boxed{1} & \boxed{\$} \end{array}$) unsuccessful (d too big)



Compressed tries search: Example

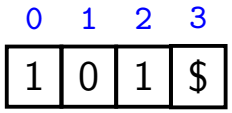
Example 2: CompressedTrie::search(

) unsuccessful (d too big)

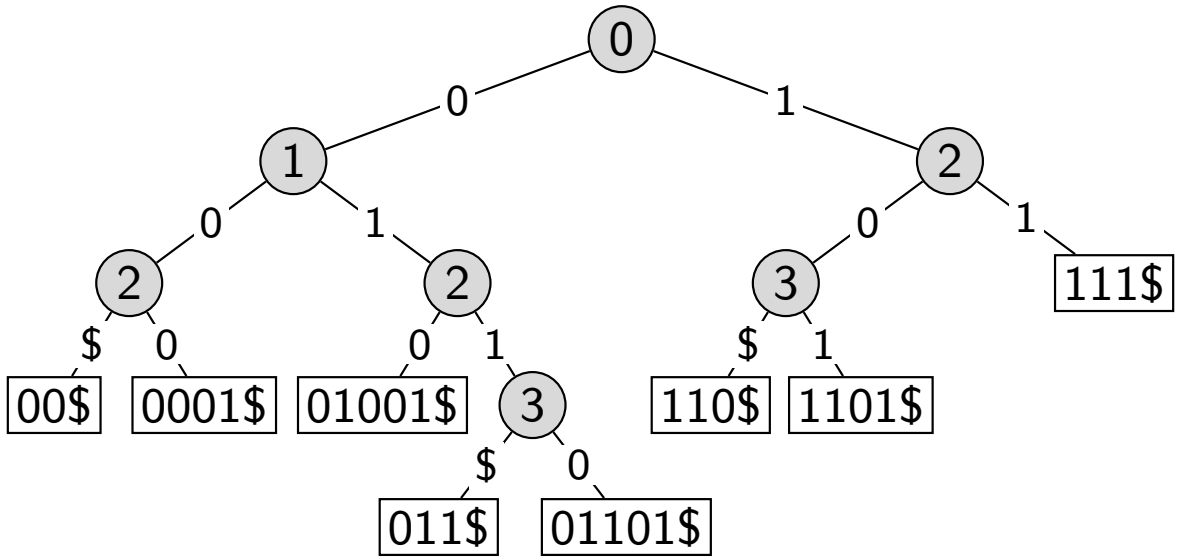


prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed tries search: Example

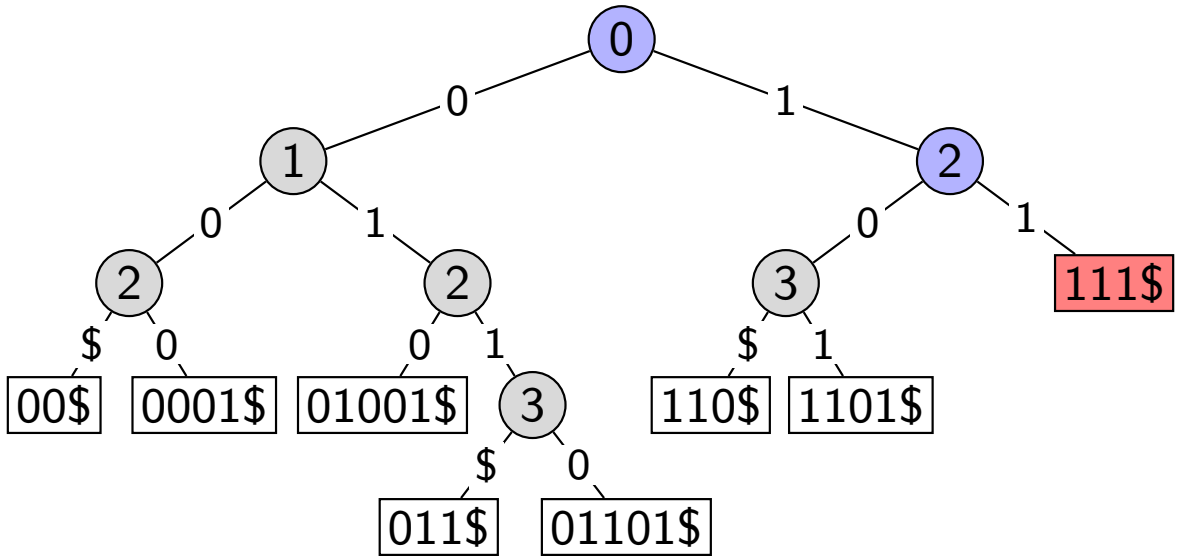
Example 3: CompressedTrie::search(

)



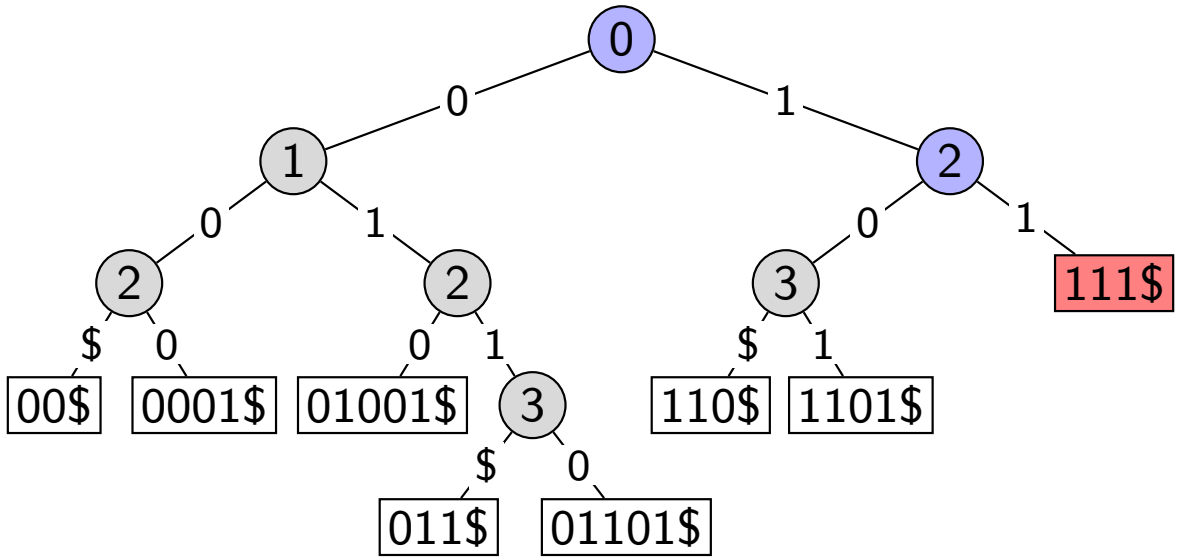
Compressed tries search: Example

Example 3: CompressedTrie::search(| 1 | 0 | 1 | $ |) unsuccessful (wrong word at leaf)



Compressed tries search: Example

Example 3: CompressedTrie::search(| 1 | 0 | 1 | $ |) unsuccessful (wrong word at leaf)



prefix-search(w): Compare w to word at reached leaf.

Compressed tries: Summary

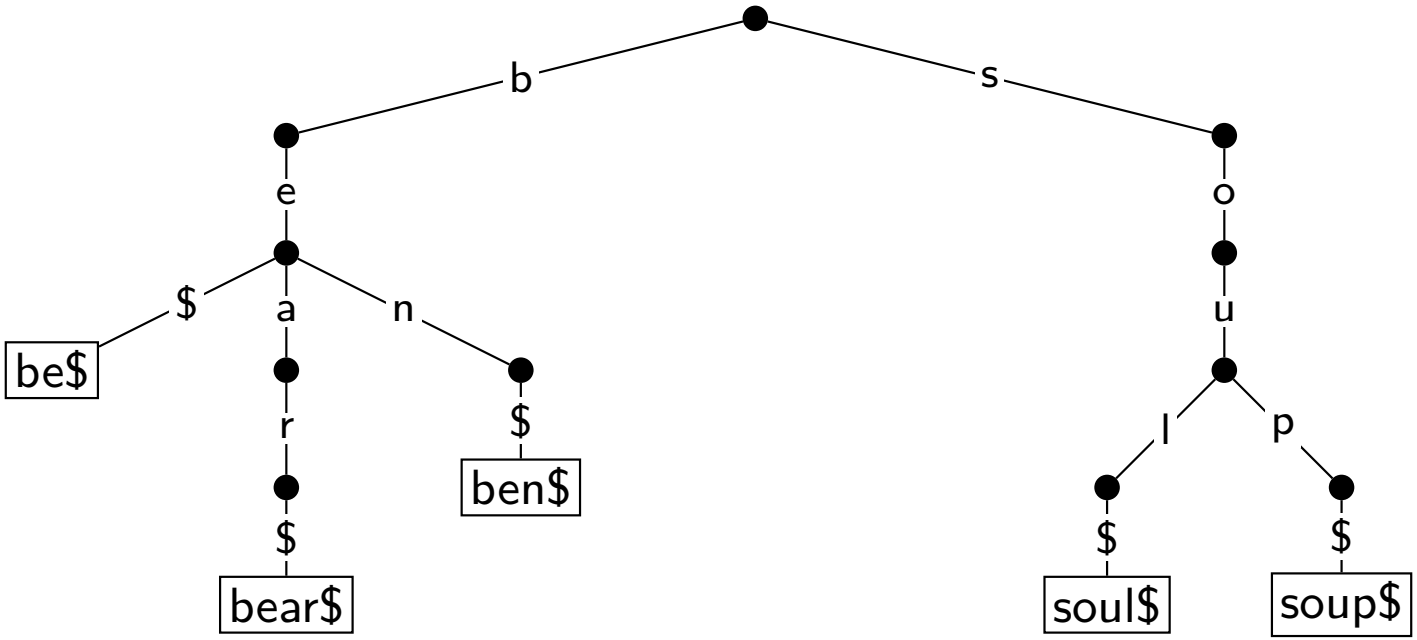
- *search*(w) and *prefix-search*(w) are fairly easy.
- *insert*(w) and *delete*(w) are conceptually simple by *uncompressing*:
  - Search for path P to word w (say we reach node z)
  - Uncompress this path (using characters of $z.leaf$)
  - Insert/Delete w as in an uncompressed trie.
  - Compress path from root to where change happened

  (Pseudocode gets more complicated and is omitted.)
- All operations take $O(|w|)$ time for a word w.
- Compressed tries use $O(n)$ space (better than other trie-variants)

Overall, code is more complicated, but space-savings are worth it if words are unevenly distributed.

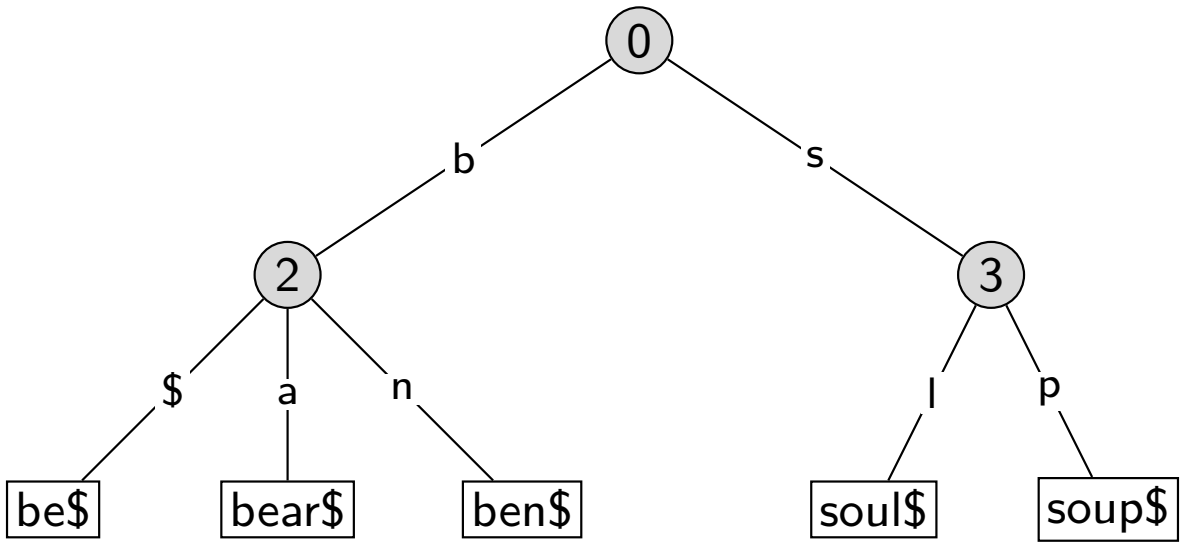
Multiway tries for larger alphabets

- To represent *strings* over any *fixed alphabet* Σ
- Any node will have at most $|\Sigma| + 1$ children (one child for the end-of-word character \$)
- Example: A trie holding strings {bear\$, ben\$, be\$, soul\$, soup\$}



Compressed multiway tries

- **Variation**: Compressed multi-way tries: compress paths as before
- Example: A compressed trie holding strings {bear$, ben$, be$, soul$, soup$}



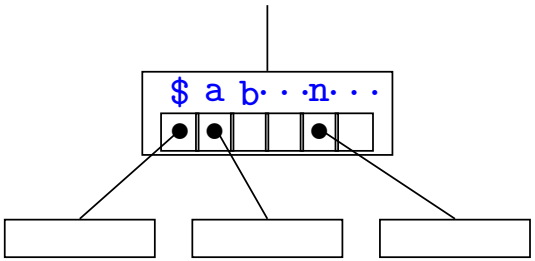
Multiway tries: Summary

- Operations *search*(w), *prefix-search*(w), *insert*(w) and *delete*(w) are exactly as for tries for bitstrings.
- Run-time $O(|w| \cdot (\text{time to find the appropriate child}))$

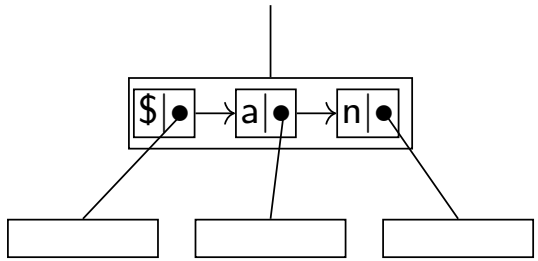
Multiway tries: Summary

- Operations *search*(w), *prefix-search*(w), *insert*(w) and *delete*(w) are exactly as for tries for bitstrings.
- Run-time $O(|w| \cdot (\text{time to find the appropriate child}))$

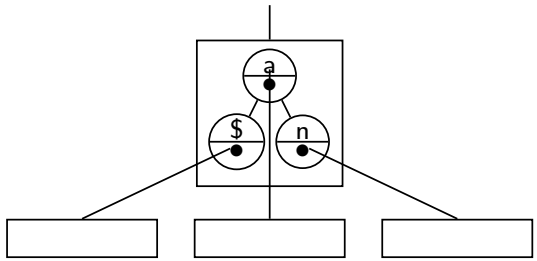
- Each node now has up to $|\Sigma| + 1$ children. How should they be stored?



Array?



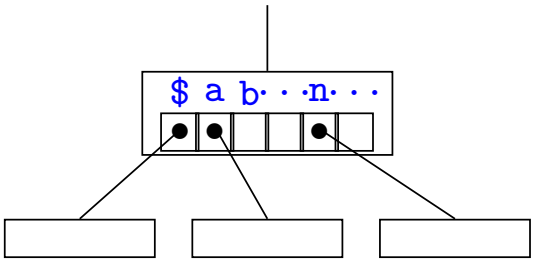
List?



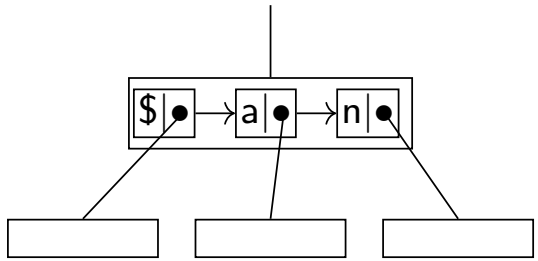
Dictionary?

Multiway tries: Summary

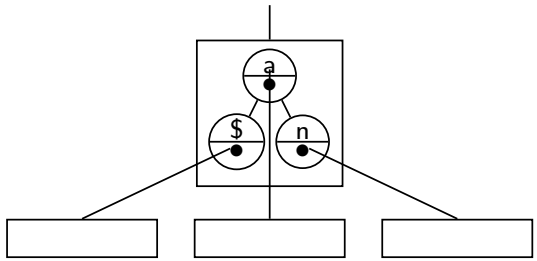
- Operations *search*(w), *prefix-search*(w), *insert*(w) and *delete*(w) are exactly as for tries for bitstrings.
- Run-time $O(|w| \cdot (\text{time to find the appropriate child}))$
- Each node now has up to $|\Sigma| + 1$ children. How should they be stored?



Array?



List?



Dictionary?

- Time/space tradeoff: arrays are fast, lists are space-efficient.
- Dictionary best in theory, not worth it in practice unless $|\Sigma|$ is huge.
- In practice, use *direct addressing* or *hashing* ($\rightarrow$ module 07).