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 2026

version *2026−07−01 12:06*

Outline

6. Dictionaries for special keys
   - Lower bound
   - Interpolation Search
   - Tries
     - Standard Tries
     - Variations of Tries
     - Compressed Tries
     - Multiway Tries

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.

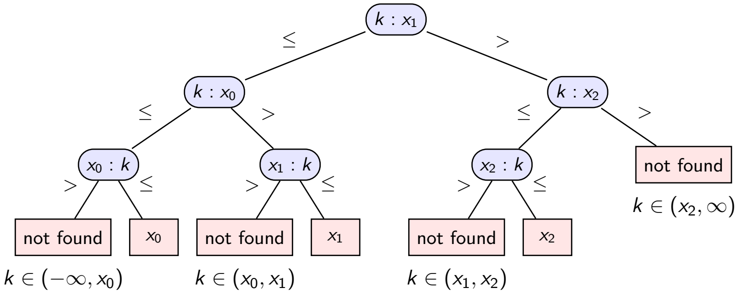
Question: Can one do better than $\Theta(\log n)$ time for *search*?
Answer: Yes and no! *It depends on what we allow*.

- 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



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

Outline

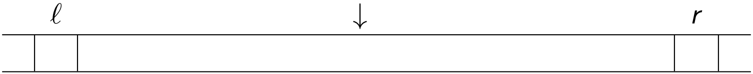
6. Dictionaries for special keys
   - Lower bound
   - Interpolation Search
   - Tries
     - Standard Tries
     - Variations of Tries
     - Compressed Tries
     - Multiway Tries

Recall: binary search

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

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

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

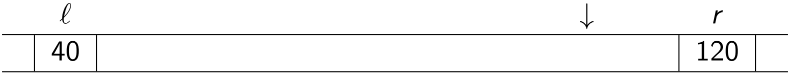


Recall: binary search

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

binary-search(A, n, k)
$\ell \leftarrow 0$, $r \leftarrow n-1$
while $(\ell \leq r)$
  $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
  if $(A[m]$ equals $k)$ **then return** "found at $A[m]$"
  else if $(A[m] < k)$ **then** $\ell \leftarrow m+1$
  else $r \leftarrow m-1$
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

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

interpolation-search$(A, n \leftarrow A.size, k)$
1. $\ell \leftarrow 0,\ r \leftarrow n-1$
2. **while** $(\ell \le r)$
3. **if** $(k < A[\ell]$ or $k > A[r])$ **return** "not found"
4. **if** $(k = A[r])$ **then return** "found at $A[r]$"
5. $m \leftarrow \ell + \lceil \frac{k - A[\ell]}{A[r]-A[\ell]} \cdot (r - \ell - 1) \rceil$
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** $r \leftarrow m-1$

- after steps 3-4, we know $A[\ell] \le k < A[r]$, so formula ends in range
- step 4 avoids division by zero when $A[\ell] = A[r]$

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

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
ℓ								$\uparrow$					r

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

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
ℓ				$\uparrow$			r						

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

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
					ℓ	$\uparrow$	r						

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

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
					ℓ	$\uparrow$	r						

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

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500
ℓ	$\uparrow$									r

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

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500
		ℓ	$\uparrow$							r

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

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500
				ℓ	$\uparrow$					r

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:

- very difficult: $\boxed{T^{\text{avg}}(n) \le T^{\text{avg}}(\lceil \sqrt{n} \rceil) + \Theta(1),\ n > 2}$
- average under $A[0] = 0$, $A[n-1] = 1$, $A[i]$'s chosen independently in $(0, 1)$ with uniform distribution
- resolves to $T^{\text{avg}}(n) \in O(\log \log n)$.

Outline

6. Dictionaries for special keys
   - Lower bound
   - Interpolation Search
   - Tries
     - Standard Tries
     - Variations of Tries
     - Compressed Tries
     - Multiway Tries

Flipped Lecture: Tries

The **Tries** section of this module will be delivered in a flipped lecture format. Instead of covering it in class, you will study the provided slides on your own before the next lecture.

What you should know

- **The material is fully examinable.**
- It is part of the course just like every other topic.
- It may appear on assignments, tutorials, and the final exam.
- **Flipped is simply a different way of learning the material. It does *not* mean the topic is optional or less important.**

- Next week's tutorials will focus on selected problems involving tries to deepen your understanding.
- The next lecture will begin with **15 minutes of Q&A** to answer any questions after you have studied the material.

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 $i = 0, 1 \ldots$)

0	1	2	3	4
b	e	a	r	$

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

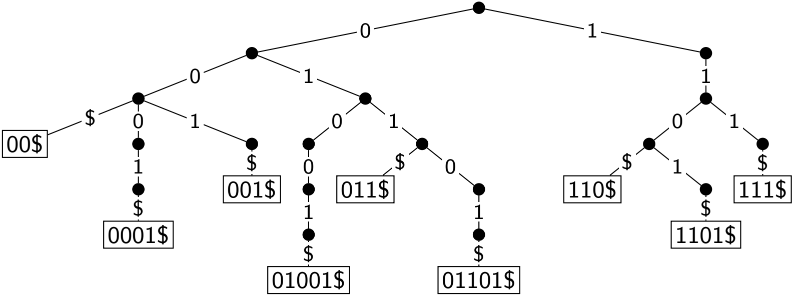
Should know:

- prefix, suffix, substring
- Sort words **lexicographically**: `be$` $<_{\text{lex}}$ `bear$` $<_{\text{lex}}$ `beer$`
  This is different from sorting numbers: `001$` $<_{\text{lex}}$ `010$` $<_{\text{lex}}$ `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 (we don't show values, as usual)



Tries: Search

- Follow links that corresponds 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

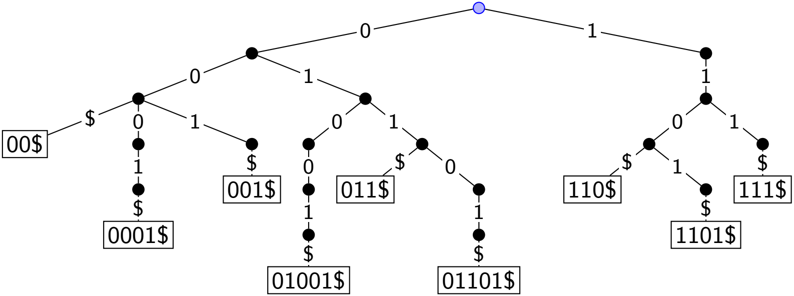
```
P ← empty stack; z ← root; d ← 0; P.push(z)
while d ≤ |w|
    if z has a child-link labelled with w[d]
        z ← child at this link; d++; P.push(z)
    else break
return P
```

Remarks:

- we assume that we can get $|w|$ in constant time
- we allow $d = |w|$, because $w[|w|]$ points to the end-sentinel

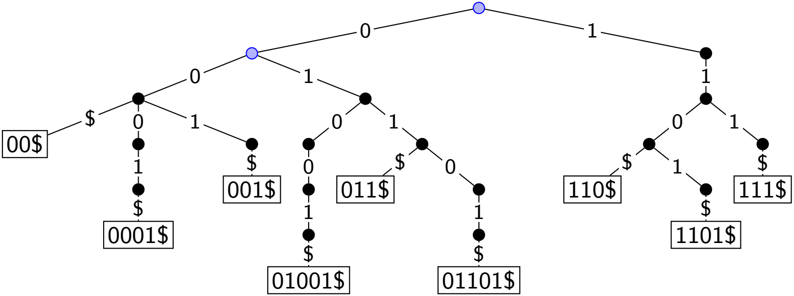
Tries search: Example

Example: Trie::search(011$)



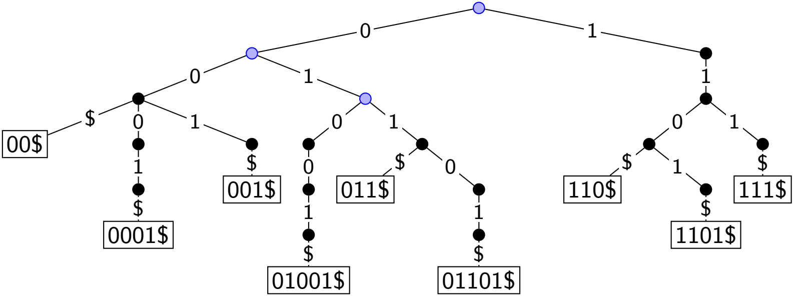
Tries search: Example

Example: Trie::search(011$)



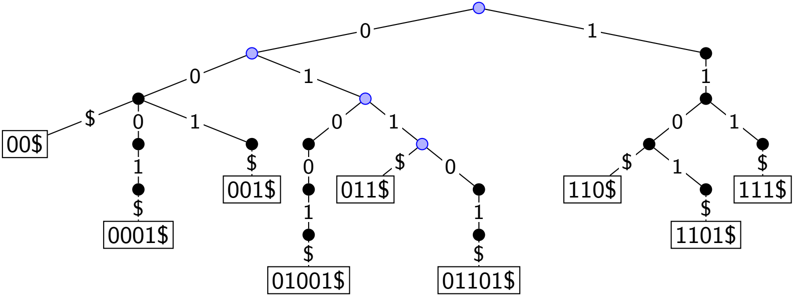
Tries search: Example

Example: Trie::search(011$)



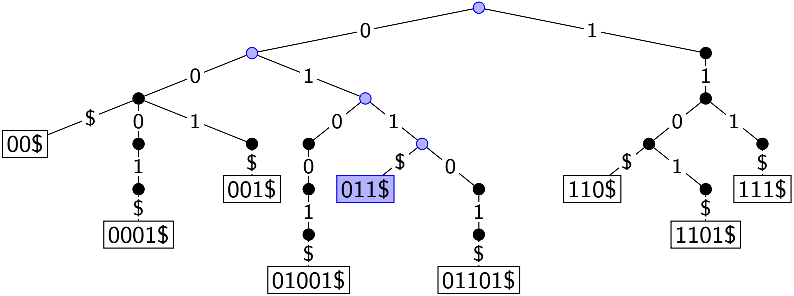
Tries search: Example

Example: Trie::search(011$)



Tries search: Example

Example: Trie::search(011$)

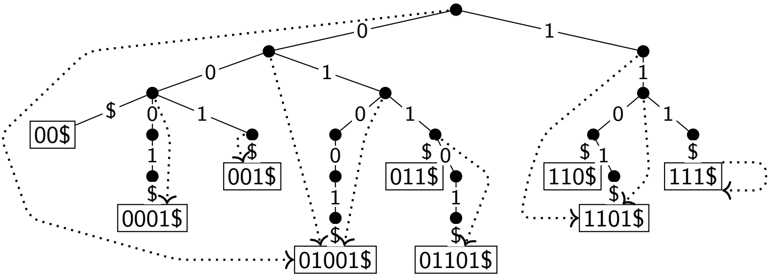


Trie::search(w)
1. $P \leftarrow$ *get-path-to*(w), $z \leftarrow P.$*top*
2. **if** (z is not a leaf) **then**
3. **return** "not found, would be in sub-trie of z"
4. **return** key-value pair at z

Tries: Prefix-search

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

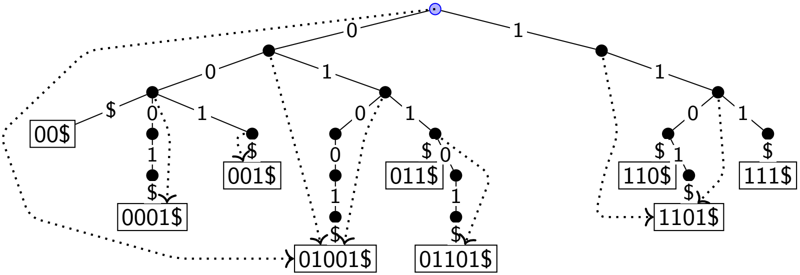
- *prefix-search*(w): Find **extension**: word for which w is a prefix.
- Testing whether extension exists is easy: there is an extension if search does $|w|$ steps
- To find extension quickly, we need **leaf-references**
  - Every node z stores reference $z.leaf$ to a leaf in subtree
  - **Convention:** store leaf with longest word



(not all leaf-references are shown)

Tries prefix-search: Example

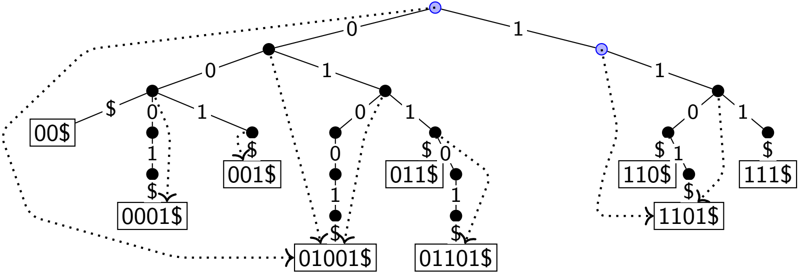
Example: Trie::prefix-search(11$)



we do $|w|$ steps iff we visit $|w|+1$ nodes

Tries prefix-search: Example

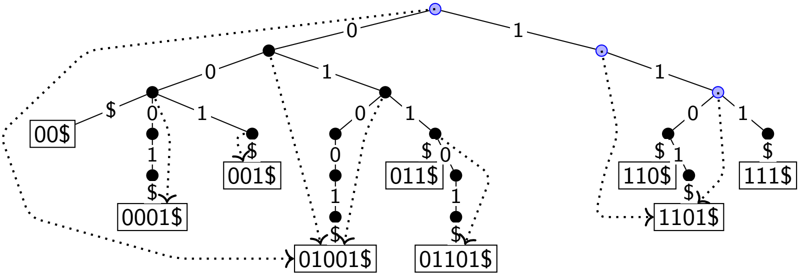
Example: Trie::prefix-search(11\$)



we do $|w|$ steps iff we visit $|w| + 1$ nodes

Tries prefix-search: Example

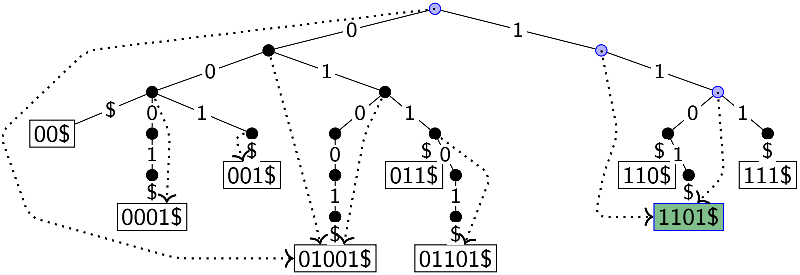
Example: Trie::prefix-search(11$)



we do $|w|$ steps iff we visit $|w|+1$ nodes

Tries prefix-search: Example

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

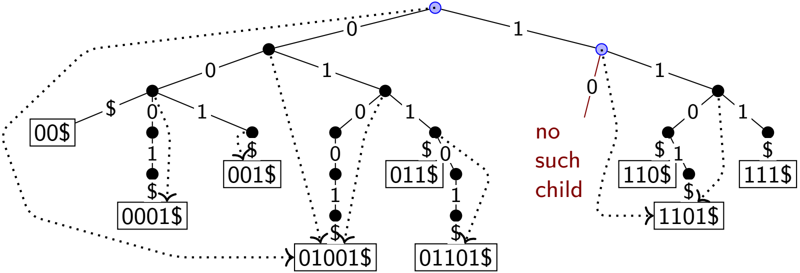


we do $|w|$ steps iff we visit $|w|+1$ nodes

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 not $|w|+1$
3. **return** "no extension of w found"
4. **return** P.*top*().*leaf*

Tries prefix-search: Example

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



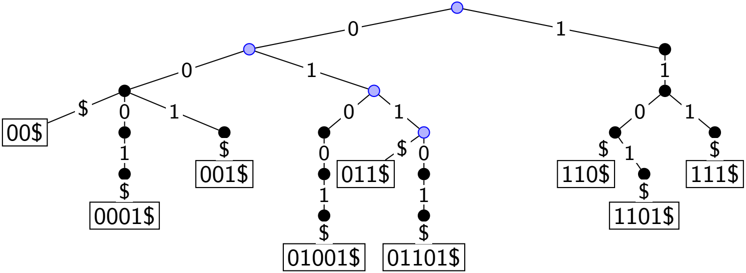
- word $w = 10\$$ has size $|w| = 2$.
- *get-path-to*(10) returns stack with two nodes.

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\$)

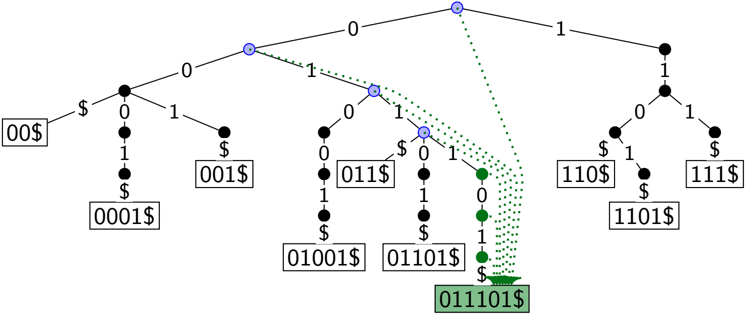


Tries: Insert

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\$)



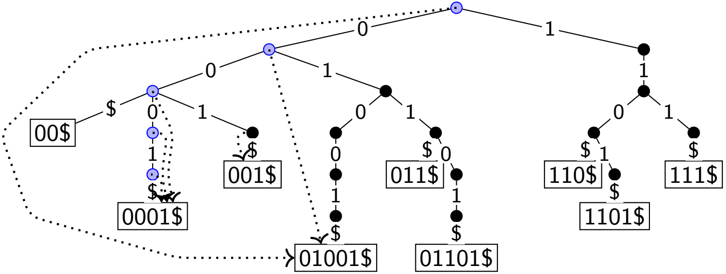
(only updated leaf-references are shown)

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\$)



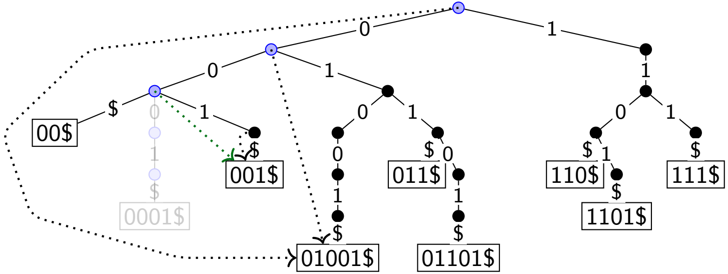
(only some leaf-references are shown)

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 $\Theta(|w|)$.

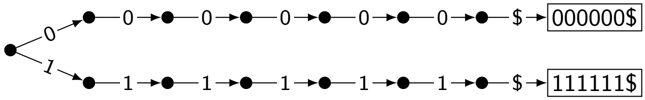
- Search-time is *independent* of number n of words stored in the trie!
- Search-time is small for short words.

The trie for a given set of words is unique

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

Disadvantages:

- Tries can be wasteful with respect to space.

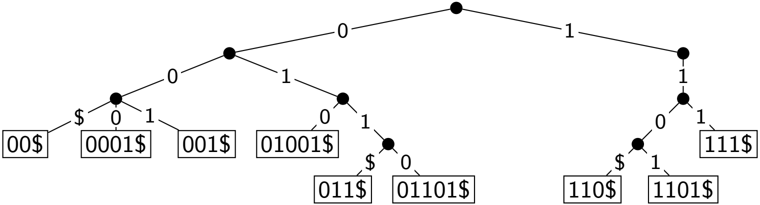


- Worst-case space is $\Theta(n \cdot (\text{maximum length of a word}))$
- What can we do to save space?

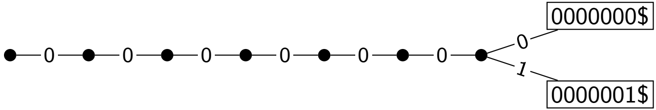
Variations of tries: Pruned tries

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.



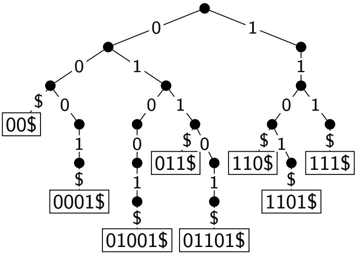
But the space can still be bad.

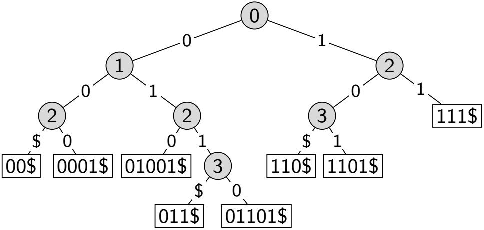


Compressed tries

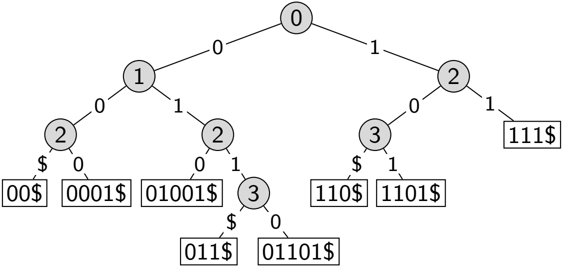
Another (important!) variation:

- remove nodes with only one child
- each remaining node stores an *index*, corresponding to its level in the uncompressed trie (on level d, we searched for link with $w[d]$)





Compressed tries



- **Invariant:** At any node z where $d = z.\textit{index}$,
  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 $\#\text{nodes} \leq 2n - 1$.

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

Compressed tries: Search

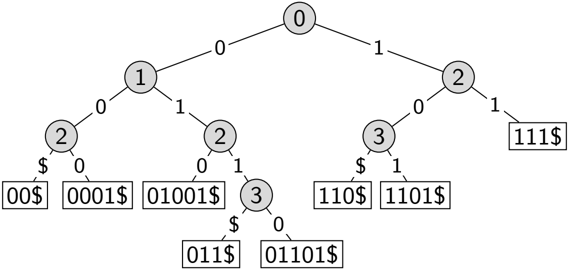
- As for tries, follow links that corresponds to current bits in w
- Main difference: stored indices say which bits to compare.
- Also: must compare w to word found at the leaf

```
CompressedTrie::get-path-to(w)
    P ← empty stack; z ← root; P.push(z)
    while z is not a leaf and (d ← z.index ≤ |w|) do
        if (z has a child-link labelled with w[d]) then
            z ← child at this link; P.push(z)
        else break
    return P
```

```
CompressedTrie::search(w)
    P ← get-path-to(w), z ← P.top
    if (z is not a leaf or word stored at z is not w) then
        return "not found"
    return key-value pair at z
```

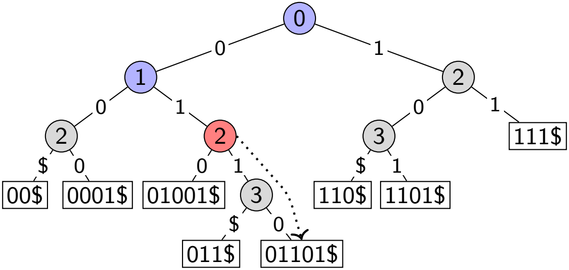
Compressed tries search: Example

Example 1: CompressedTrie::search($\begin{array}{ccc} \scriptstyle 0 & \scriptstyle 1 & \scriptstyle 2 \\ \boxed{0} & \boxed{1} & \boxed{\$} \end{array}$)



Compressed tries search: Example

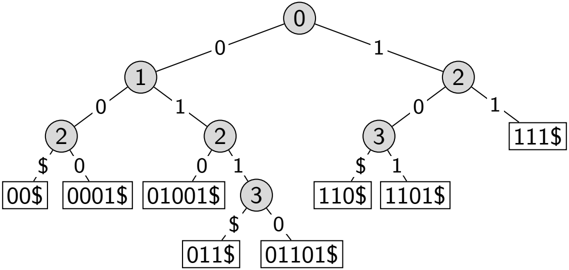
Example 1: CompressedTrie::search($\begin{array}{ccc} 0 & 1 & 2 \\ \hline \boxed{0} & \boxed{1} & \boxed{\$} \end{array}$) unsuccessful (no \$-child)



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

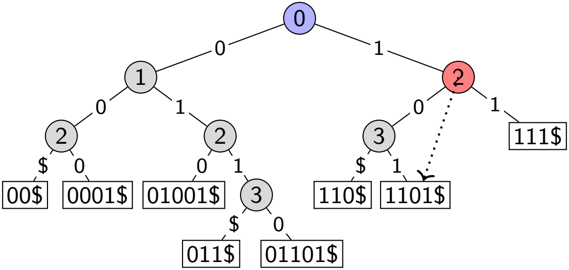
Compressed tries search: Example

Example 2: CompressedTrie::search($\begin{array}{cc} \scriptstyle 0 & \scriptstyle 1 \\ \boxed{1} & \boxed{\$} \end{array}$)



Compressed tries search: Example

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

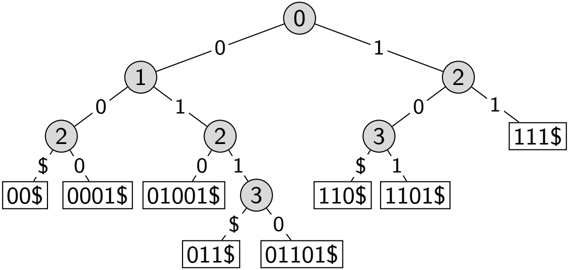


prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed tries search: Example

Example 3: CompressedTrie::search(1 0 1 $)

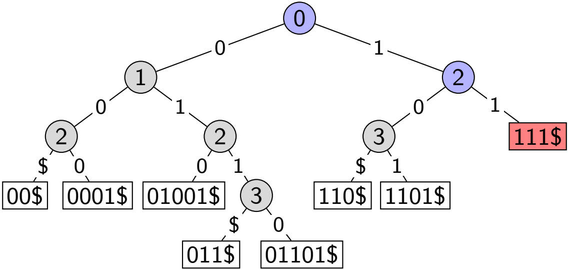
0	1	2	3
1	0	1	$



Compressed tries search: Example

Example 3: CompressedTrie::search($\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \boxed{1} & \boxed{0} & \boxed{1} & \boxed{\$} \end{array}$) unsuccessful

(wrong word at leaf)



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

Compressed tries: Summary

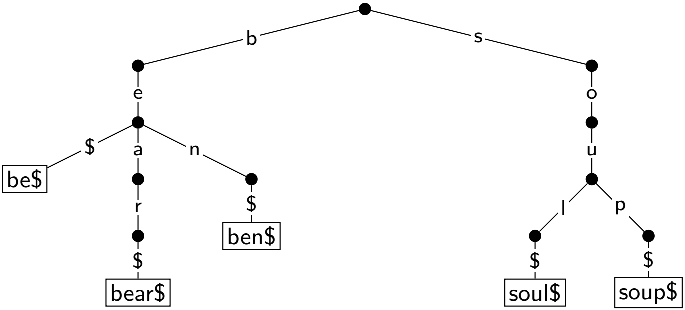
- *search*(w) and *prefix-search*(w) are fairly easy.
- *insert*(w) and *delete*(w) are conceptually simple by *uncompressing*:
  - Search for path P to word w (say we reach node z)
  - Uncompress this path (using characters of $z.$*leaf*)
  - Insert/Delete w as in an uncompressed trie.
  - Compress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

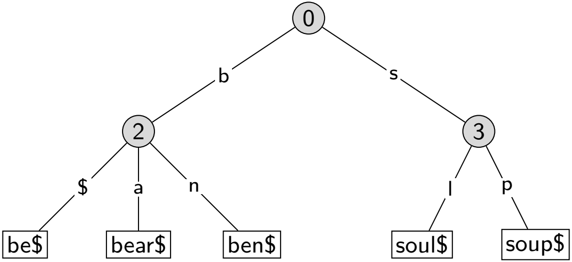
Multiway tries for larger alphabets

- 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\$}



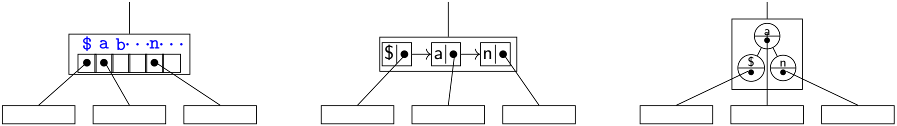
Compressed multiway tries

- **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 $O(|w| \cdot \text{(time to find the appropriate child)})$
- 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: array using *direct addressing* or *hashing* ($\rightarrow$ module 07).