CS 240E – Data Structures and Data Management (Enriched)

Module 6: Dictionaries for special keys

Tom Iagovet
Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version *2026-06-09 22:07*

Outline

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

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.

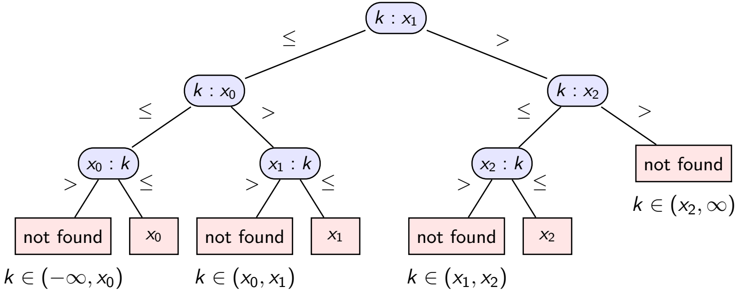
Question: Can one do better than $\Theta(\log n)$ time for *search*?
Answer: Yes and no! *It depends on what we allow*.

- 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



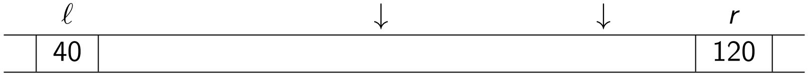
- How many possible outcomes are there?
- What does that tell us about the height of the decision tree?

Towards interpolation search

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

binary-search(A, n, k)
```
ℓ ← 0, r ← n − 1
while (ℓ ≤ r)
      m ← ⌊(ℓ+r)/2⌋
      if (A[m] equals k) then return "found at A[m]"
      else if (A[m] < k) then ℓ ← m + 1
      else r ← m − 1
return "not found, but would be between A[ℓ−1] and A[ℓ]"
```

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 ← A.size, k)
ℓ ← 0, r ← n − 1
while (ℓ ≤ r)
        if (k < A[ℓ] or k > A[r]) return "not found"
        if (k = A[r]) then return "found at A[r]"
        m ← ℓ + ⌈ (k−A[ℓ])) / (A[r]−A[ℓ]) · (r − ℓ − 1)⌉
        if (A[m] equals k) then return "found at A[m]"
        else if (A[m] < k) then ℓ ← m + 1
        else r ← 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):

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

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

Improving interpolation search

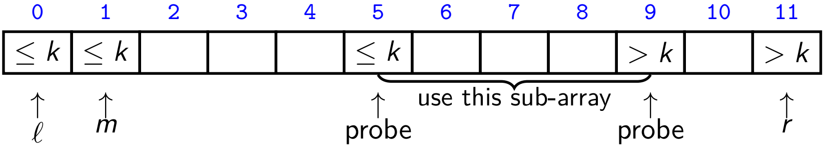
- Proving $T^{\text{avg}}(n) \le T^{\text{avg}}(\sqrt{n}) + \Theta(1)$ is very complicated.

  (
  - Switch to analyze run-time on randomly chosen input.
  - Study expected error, i.e., distance between index of k and where we probed.
  - Argue that expected error is in $O(\sqrt{n})$ in first round.
  - Argue that expected error is in $O(n^{(1/2)^i})$ after i rounds.
  - Study the martingale formed by the errors in the rounds.
  - Argue that its expected length is $O(\log \log n)$.

)

- Instead: Define a variant of *interpolation-search*
  - Better worst-case run-time.
  - Easier to analyze.
  - Avoid 3-way comparisons and extra checks.
- Idea: *Force* the sub-array to have size $\sqrt{n}$

Improving interpolation search



- First probe at m as before.
- If $A[m] \leq k$, probe rightward.
- Probes always go $\lceil \sqrt{N} \rceil$ indices rightward
  (where $N = r - \ell - 1 = \#$ unknown keys, here $N = 10$)
- Continue probing until $> k$ or out-of-bounds
- Recurse in the only sub-array where k can be.
  It has $\leq \lceil \sqrt{N} \rceil - 1$ unknown keys.
- Observe: $\#\{\text{probes in this round}\} \leq \sqrt{N}$

Improving interpolation search

Interpolation-search-better(A, n, k)
A: sorted array of size n, k: key

if $(k < A[0]$ or $k > A[n-1])$ **return** "not found"
if $(k = A[n-1])$ **return**"found at index $n{-}1$"
$\ell \leftarrow 0$, $r \leftarrow n-1$
while $(N \leftarrow (r-\ell-1) \geq 1)$ `// have` $A[\ell] \leq k < A[r]$
  $m \leftarrow \ell + \lceil \frac{k-A[\ell])}{A[r]-A[\ell]} \cdot (r-\ell-1) \rceil$
  if $(A[m] \leq k)$ `// probe rightward`
    for $h = 1, 2, \ldots$
      $\ell \leftarrow m + (h{-}1)\lceil\sqrt{N}\rceil$, $r' \leftarrow \min\{r, m + h\lceil\sqrt{N}\rceil\}$
      if $(r' = r$ or $A[r'] > k)$ **then** $r \leftarrow r'$ and **break**
  else $\ldots$ `// symmetrically probe leftward`
if $(k = A[\ell])$ **return** "found at index ℓ"
else return "not found"

interpolation-search-improved: Analysis

- Let $T(N)$ be the total number of probes if there were N uncompared keys.
- $T(N) \leq T(\lceil\sqrt{N}\rceil - 1) + \sqrt{N}$ since sub-array has $\leq \lceil\sqrt{N}\rceil - 1$ unknowns.
- This resolves to $O(\sqrt{N})$

  (see table, or prove $T(N) \leq 2\sqrt{N} + O(1)$ using induction.)

Result: The worst-case run-time of *interpolation-search-improved* is in $O(\sqrt{n})$.

Average-case run-time?

- Rephrase: If array-entries are chosen uniformly at random, what is the expected number of probes per found?

interpolation-search-improved: Analysis

Claim: The number of rounds is $\log\log n + O(1)$.

- Reverse analysis: The size at least squares with every round.

$$2 \to 4 \to 16 \to 256 \to \cdots \to 2^{2^i} \to \left(2^{2^i}\right)^2 = 2^{2^{i+1}}$$

- $2^{2^i} \geq n \Leftrightarrow i \geq \log\log n$

Claim: Expected number of probes per round is at most 2.5.

(Proof later, study consequences first)

- $$\begin{aligned}\#\text{probes} \leq & \quad \#(\text{rounds}) * \#(\text{probes per round}) \\ \leq & \quad 2.5\log\log n + O(1) \text{ on average}\end{aligned}$$

- **Result:** The average-case run-time of *interpolation-search-improved* is in **$O(\log\log n)$.**

Fewer probes than *binary-search-optimized*'s $\lceil \log n \rceil + 1$ even for small n.

Expected number of probes per round

- The "unknown" numbers $A[\ell{+}1..r{-}1]$ are randomly chosen

$\Rightarrow$ This defines random variable $idx(k)$ of location of k.

ℓ													$idx(k)$ $\downarrow$		r
$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$\le k$	$>k$	$>k$
					$\uparrow$ m			$\uparrow$ probe			$\uparrow$ probe			$\uparrow$ probe	

Show (no details):

- $E[idx(k)] = \ell + \frac{k - A[\ell]}{A[r]-A[\ell]}(r-\ell-1) \approx m$ (except for rounding)
- $P(\text{many probes}) = P(idx(k)$ is very far away from its expected value)
  - Use Chebyshev's inequality to bound this using the variance of $idx(k)$

  $\Rightarrow$ $P(\ge i \text{ probes}) \le \frac{1}{(i-2)^2}$ for $i \ge 3$.

$\Rightarrow$ $E[\#\text{probes}] = \sum_{i\ge 1} P(\ge i \text{ probes}) \le 2 + \sum_{i\ge 3} \frac{1}{(i-2)^2} = 2 + \frac{\pi^2}{6}$

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.

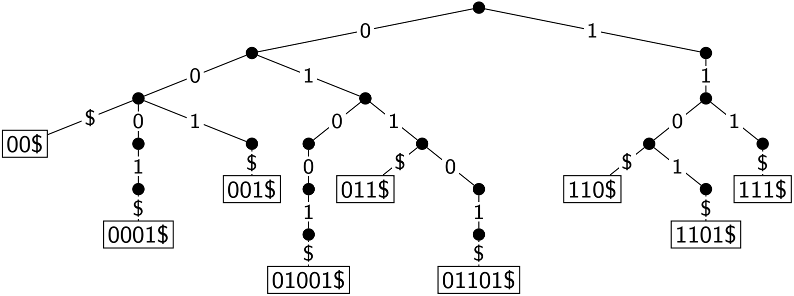
Should know:

- prefix, suffix, substring
- Sort words **lexicographically**: $\texttt{be\$} <_{\text{lex}} \texttt{bear\$} <_{\text{lex}} \texttt{beer\$}$
  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

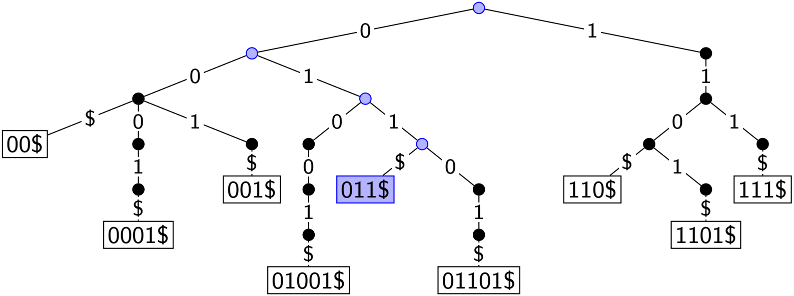
- Follow links that corresponds to current bits in w, keeping track of current depth d
- Repeat until no such link or w found at a leaf

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
$P \leftarrow$ empty stack; $z \leftarrow root$; $d \leftarrow 0$; $P.push(z)$
while $d \leq |w|$
  if z has a child-link labelled with $w[d]$
    $z \leftarrow$ child at this link; $d{+}{+}$; $P.push(z)$
  else break
return P

Tries search: Example

Example: Trie::search(011$)



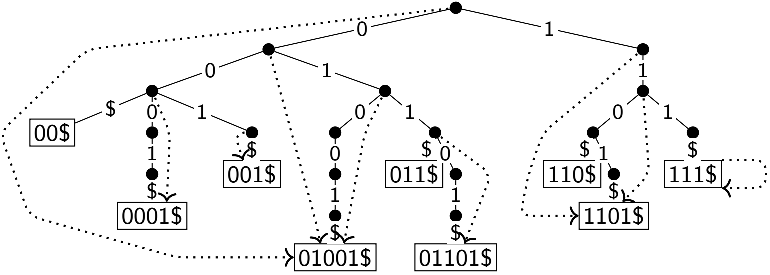
Trie::search(w)

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:

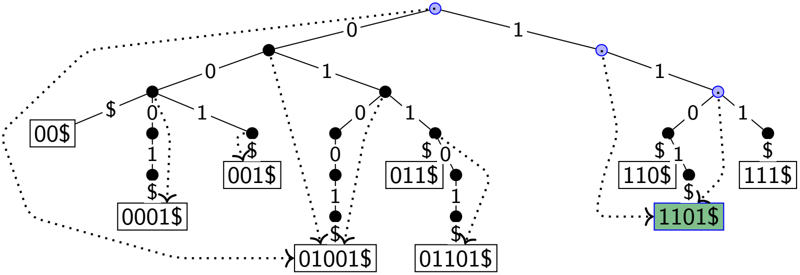
- *prefix-search*(w): Find **extension**: word for which w is a prefix.
- Testing whether extension exists is easy (how?)
- To find extension quickly, we need **leaf-references**
  - Every node z stores reference $z.$*leaf* to a leaf in subtree
  - **Convention:** store leaf with longest word



(not all leaf-references are shown)

Tries prefix-search: Example

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

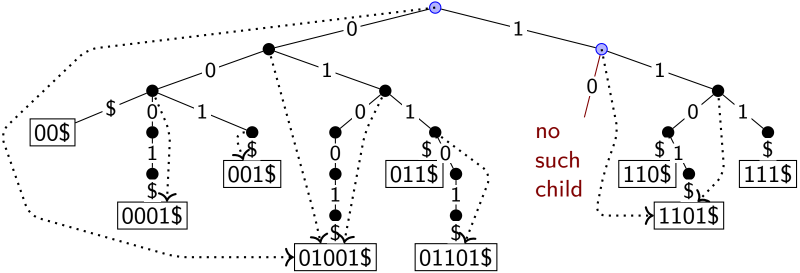


Trie::prefix-search(w)

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$)



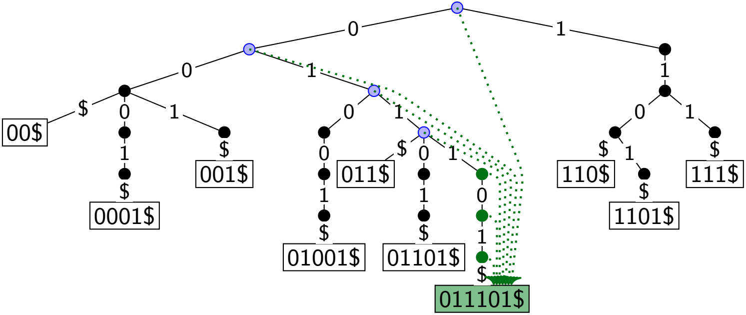
- Word 10$ has size 2.
- *get-path-to*(10) returns stack with two nodes.
- We need more than $|w|$ nodes on P to have an extension.

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\$)



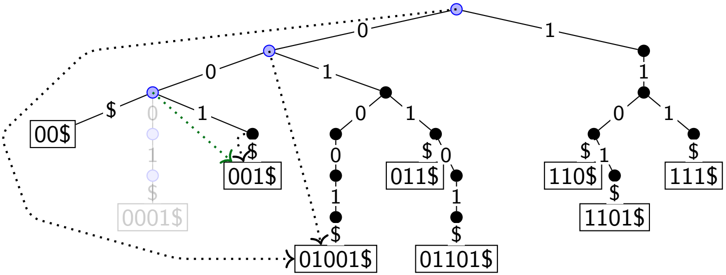
(only updated leaf-references are shown)

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)

Binary tries: Summary

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

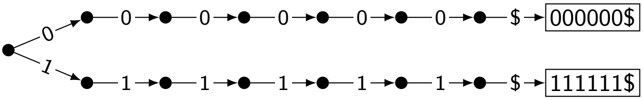
- Search-time is *independent* of number n of words stored in the trie!
- Search-time is small for short words.

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.

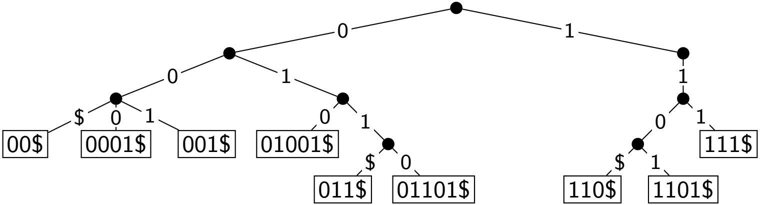


- Worst-case space is $\Theta(n \cdot (\text{maximum length of a word}))$
- What can we do to save space?

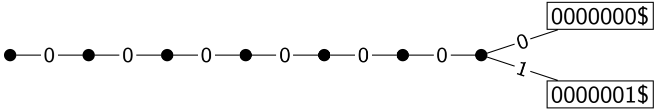
Variations of tries: Pruned tries

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.



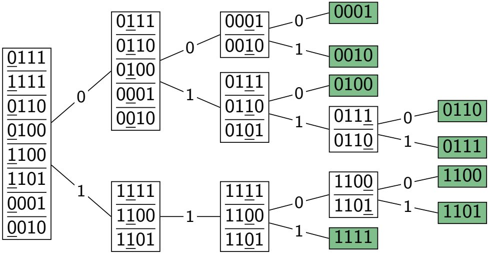
But the space can still be bad.

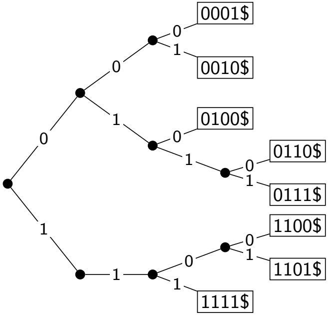


Pruned tries and MSD-radix sort

We have (implicitly) seen pruned tries before:

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

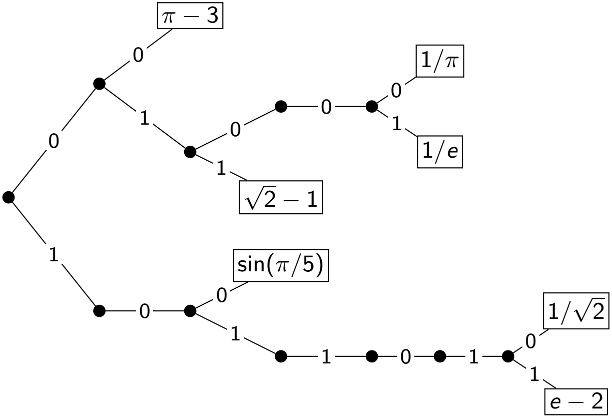




Storing infinite bitstrings

We can even store infinite-length bitstrings

(if given via a look-up function for ith bit).

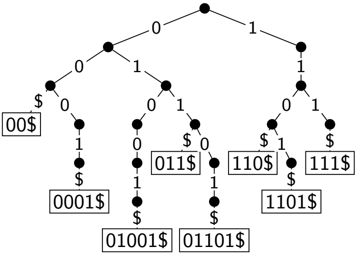


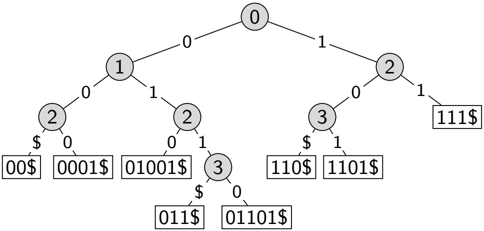
Can show: For n infinite bitstrings with randomly chosen bits, the expected search-time is $O(\log n)$.

Compressed tries

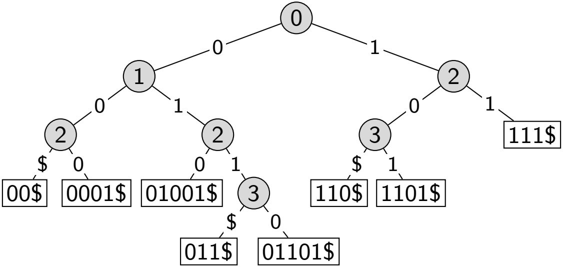
Another (important!) variation:

- 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 $d = z.index$,
  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

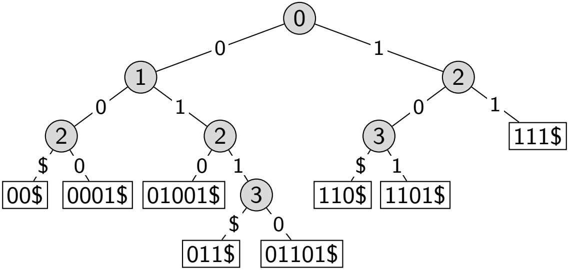
- As for tries, follow links that corresponds to current bits in w
- Main difference: stored indices say which bits to compare.
- Also: must compare w to word found at the leaf (why?)

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

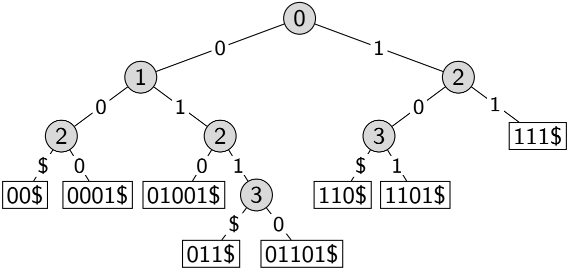
Example 1: CompressedTrie::search($\begin{array}{ccc} 0 & 1 & 2 \\ \boxed{0} & \boxed{1} & \boxed{\$} \end{array}$) unsuccessful (no \$-child)



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

Compressed tries search: Example

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

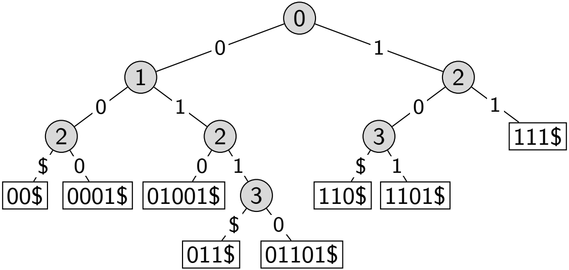


prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed tries search: Example

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

0	1	2	3
1	0	1	$



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

Compressed tries: Summary

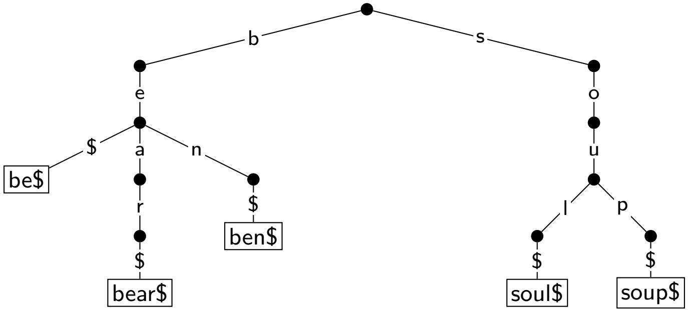
- *search*(w) and *prefix-search*(w) are fairly easy.
- *insert*(w) and *delete*(w) are conceptually simple by *uncompressing*:
  - Search for path P to word w (say we reach node z)
  - Uncompress this path (using characters of $z.$*leaf*)
  - Insert/Delete w as in an uncompressed trie.
  - Compress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

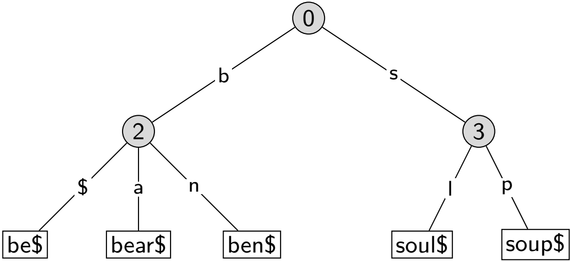
Multiway tries for larger alphabets

- 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

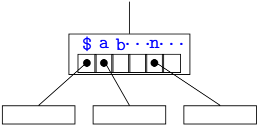
- **Variation**: Compressed multi-way tries: compress paths as before
- Example: A compressed trie holding strings {bear$, ben$, be$, soul$, soup$}



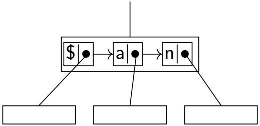
Multiway tries: Summary

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

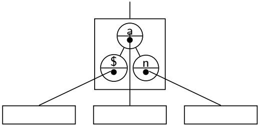
- Each node now has up to $|\Sigma| + 1$ children. How should they be stored?



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, use *direct addressing* or *hashing* ($\rightarrow$ module 07).