CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

Olga Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025−02−10 10:50

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.

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.

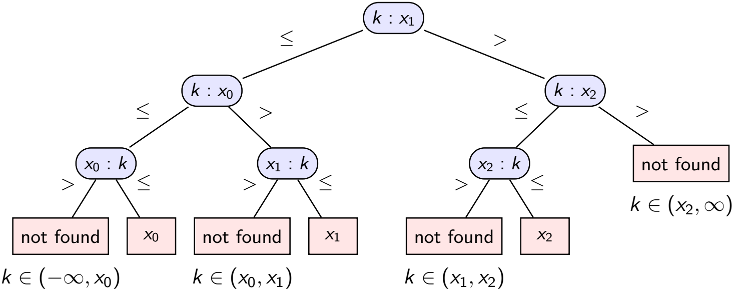
Question: Can one do better than $\Theta(\log n)$ time for *search*?
Answer: Yes and no! *It depends on what we allow*.

- 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.

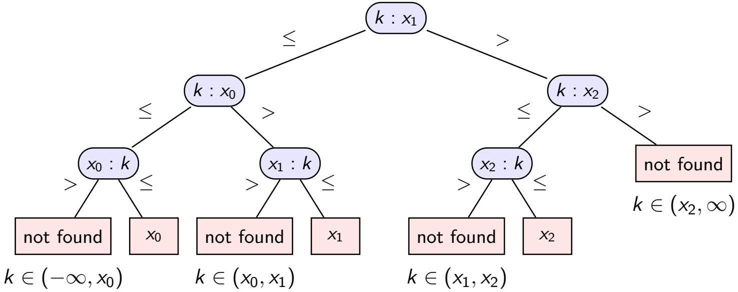
Proof: Via decision tree for items $x_0, \ldots, x_{n-1}$ and search for k



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

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

Interpolation Search Motivation

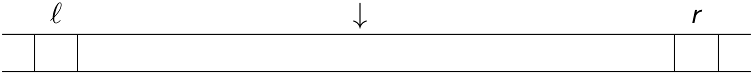
Scenario: Numbers in *sorted array*

0	1	2	3	4	5	6
40	50	70	90	100	110	120

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 $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lceil \frac{1}{2}(r-\ell-1) \rceil$



Interpolation Search Motivation

Scenario: Numbers in *sorted array*

0	1	2	3	4	5	6
40	50	70	90	100	110	120

```
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 $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lceil \frac{1}{2}(r - \ell - 1) \rceil$

ℓ		↓	r
40			120

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):

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10	11	12	12
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 Second Example

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 Second Example

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 Second Example

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 Second Example

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 Second Example

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 $T^{\text{avg}}(n) \in O(\log \log n)$.

Outline

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

Words (review)

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

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

$$\{\texttt{be}, \texttt{bear}, \texttt{beer}\}$$

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

Words (review)

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

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

$$\{\texttt{be}, \texttt{bear}, \texttt{beer}\}$$

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

Convention: Words have end-sentinel \$ (sometimes not shown)

- $w.size = |w| = \#$ non-sentinel characters: $|\texttt{be\$}|$=2.

Words (review)

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

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

$$\{\texttt{be}, \texttt{bear}, \texttt{beer}\}$$

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

Convention: Words have end-sentinel \$ (sometimes not shown)

- $w.size = |w| = \#$ non-sentinel characters: $|\texttt{be\$}|$=2.

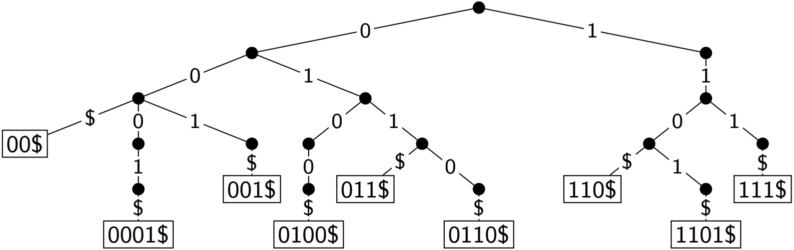
Should know:

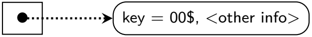
- prefix, suffix, substring
- Sort words **lexicographically**: $\texttt{be\$} <_{\text{lex}} \texttt{bear\$} <_{\text{lex}} \texttt{beer\$}$
  This is different from sorting numbers: $\texttt{001\$} <_{\text{lex}} \texttt{010\$} <_{\text{lex}} \texttt{1\$}$

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.



(Reminder: A more accurate picture would be )

Tries: Search

- Follow links that corresponds to current bits in w
- Repeat until no such link or w found at a leaf

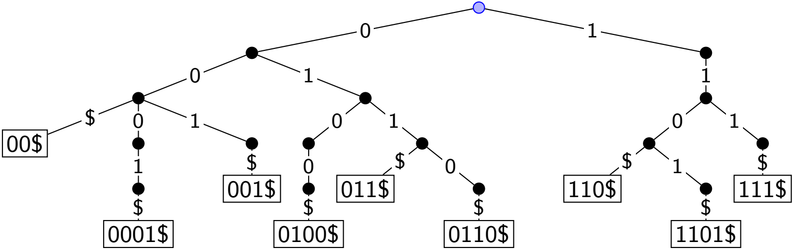
Similar as for skip lists, we find search-path separately first.

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
```

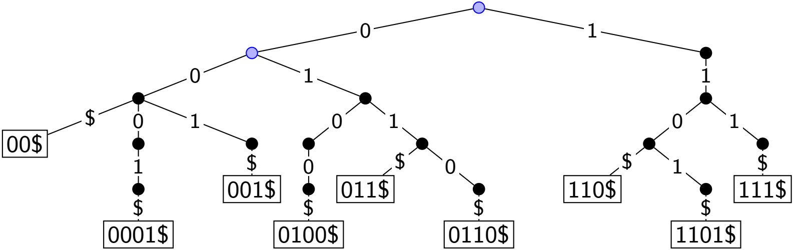
Tries: Search Example

Example: Trie::search(011$)



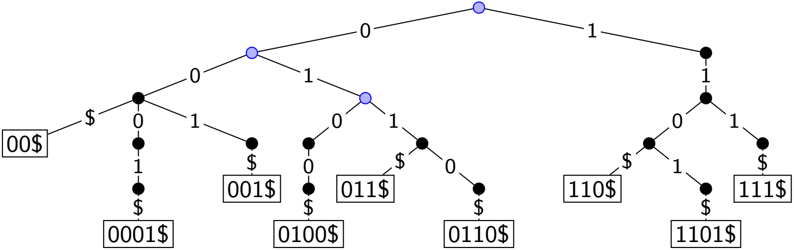
Tries: Search Example

Example: Trie::search(011$)



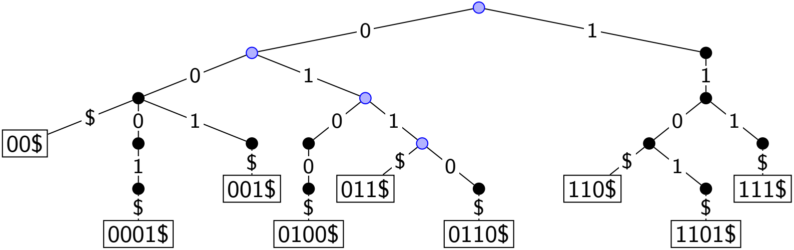
Tries: Search Example

Example: Trie::search(011$)



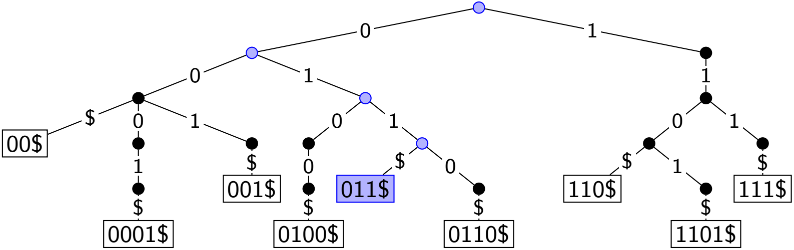
Tries: Search Example

Example: Trie::search(011$)



Tries: Search Example

Example: Trie::search(011$)

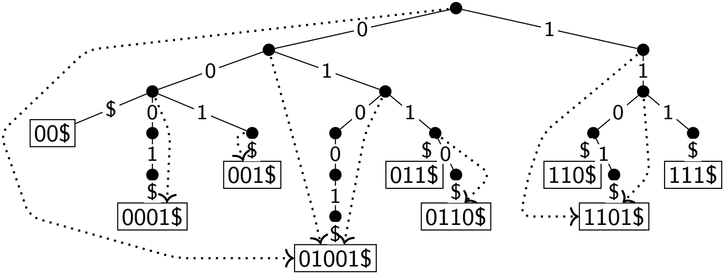



```
Trie::search(w)
    P ← get-path-to(w), z ← P.top
    if (z is not a leaf) then
        return "not found, would be in sub-trie of z"
    return key-value pair at z
```

Tries: Leaf-references

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

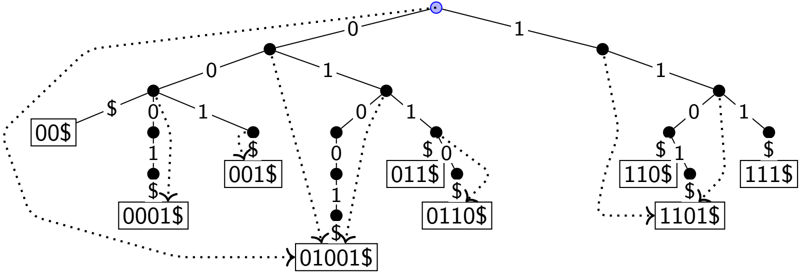
- *prefix-search*(w): Find word w' in trie for which w is a prefix.
- Testing whether w' exists is easy (how?)
- To find w' quickly, we need **leaf-references**
  - Every node z stores reference $z.$*leaf* to a leaf in subtree
  - **Convention:** store leaf with longest word



(not all leaf-references are shown)

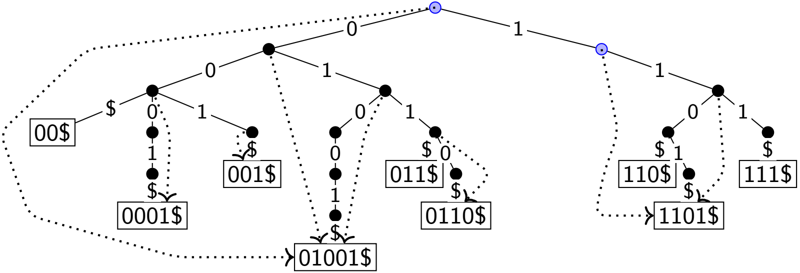
Tries: Prefix-Search Example

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



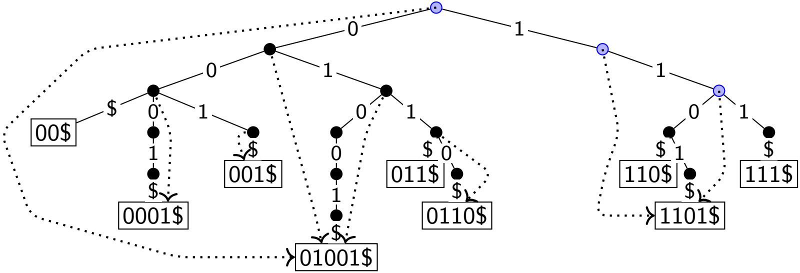
Tries: Prefix-Search Example

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



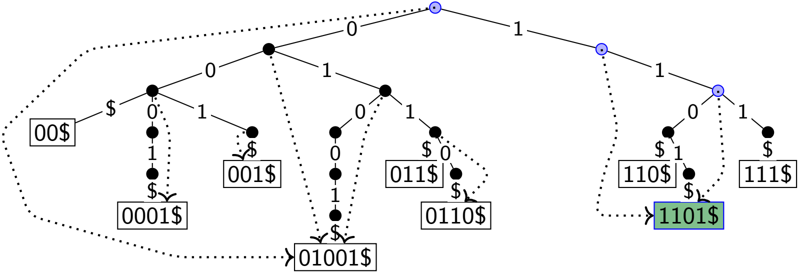
Tries: Prefix-Search Example

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



Tries: Prefix-Search Example

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



Trie::prefix-search(w)

$P \leftarrow$ *get-path-to*(w)

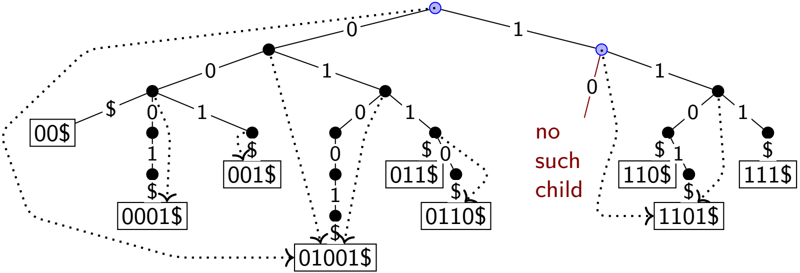
if number of nodes on P is w.*size* or less

  return "no extension of w found"

return P.*top*().*leaf*

Tries: Prefix-Search Example

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



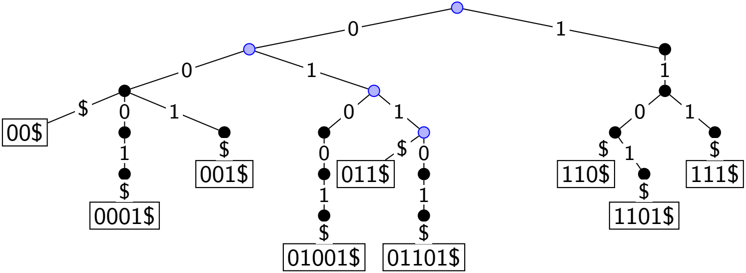
- Word 10$ has size 2.
- *get-path-to*(w) returns stack with two nodes.
- We need more than $w.size$ nodes on P to have an extension.

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$)

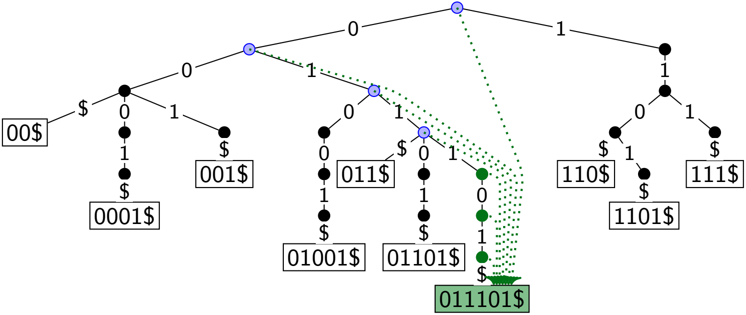


Tries: Insert

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$)



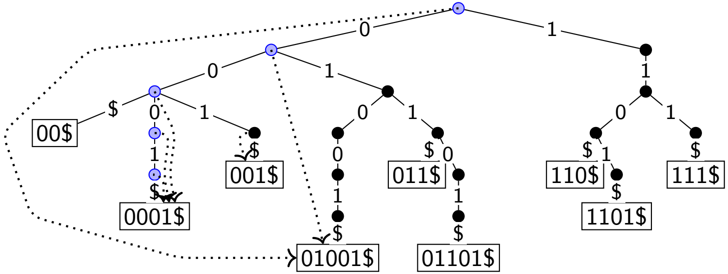
(only updated leaf-references are shown)

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 has 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$)



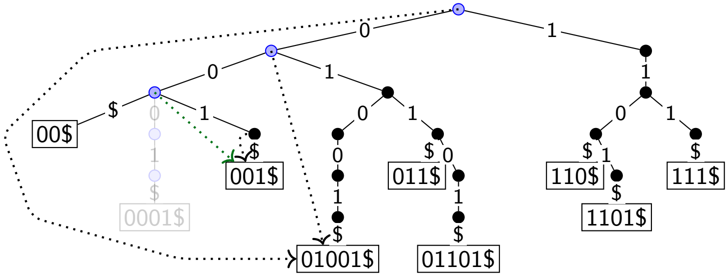
(only some leaf-references are shown)

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 has 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 $\Theta(|w|)$.

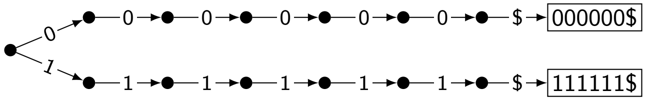
- Search-time is *independent* of number n of words stored in the trie!
- Search-time is small for short words.

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.

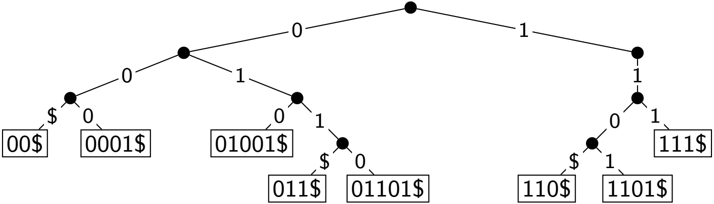


- Worst-case space is $\Theta(n \cdot (\text{maximum length of a word}))$
- What can we do to save space?

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.
- Could even store infinite bitstrings (e.g. real numbers)

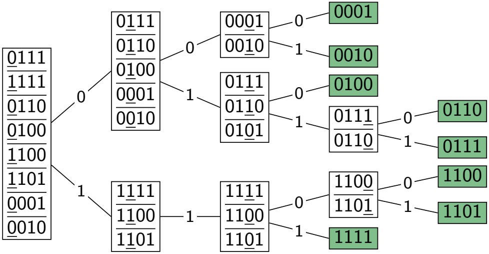


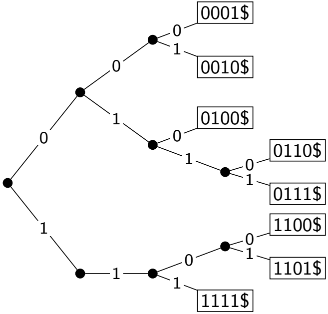
A more efficient version of tries, but the operations get a bit more complicated.

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.

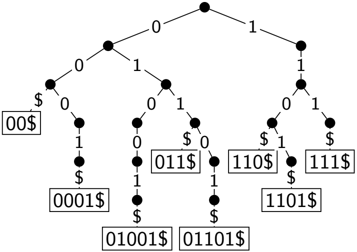


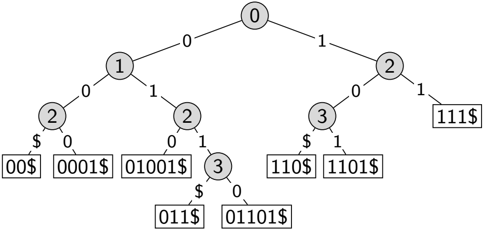


Compressed Tries

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 $w[d]$.)





Also known as **Patricia-Tries**:

Practical Algorithm to Retrieve Information Coded in Alphanumeric

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)

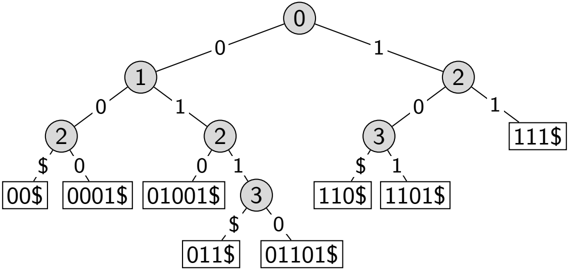
$P \leftarrow$ empty stack; $z \leftarrow$ *root*; P.*push*(z)
while z is not a leaf and ($d \leftarrow z.index \leq w.size$) **do**
    if (z has a child-link labelled with $w[d]$) **then**
        $z \leftarrow$ child at this link; P.*push*(z)
    else break
return P

CompressedTrie::search(w)

$P \leftarrow$ *get-path-to*(w), $z \leftarrow P$.*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

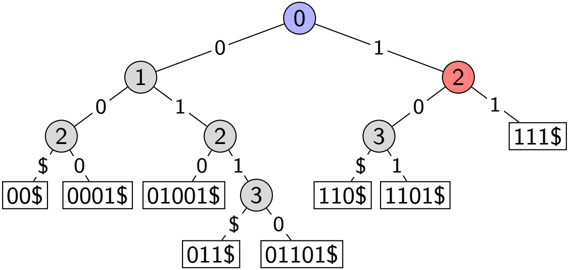
Compressed Tries: Search Example

Example 1: CompressedTrie::search(1 $) (array indices: 0, 1)



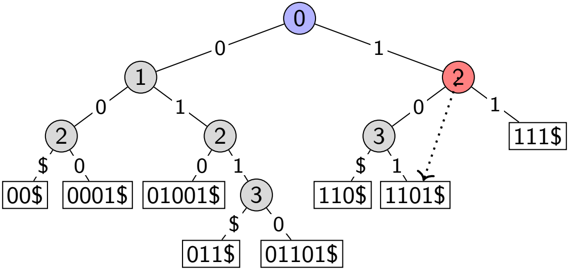
Compressed Tries: Search Example

Example 1: CompressedTrie::search(1 $) unsuccessful (d too big)



Compressed Tries: Search Example

Example 1: CompressedTrie::search(1 $) unsuccessful (d too big)



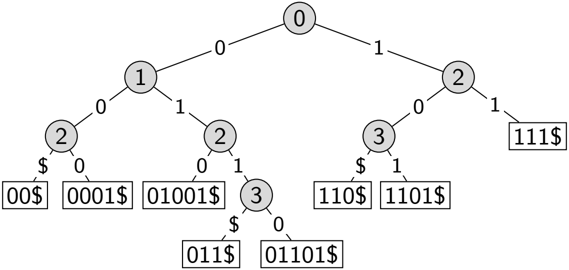
prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed Tries: Search Example

Example 2: CompressedTrie::search(

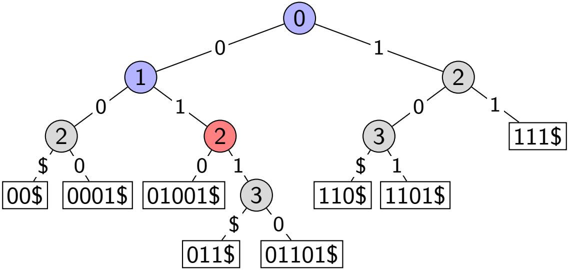
0	1	2
0	1	$

)



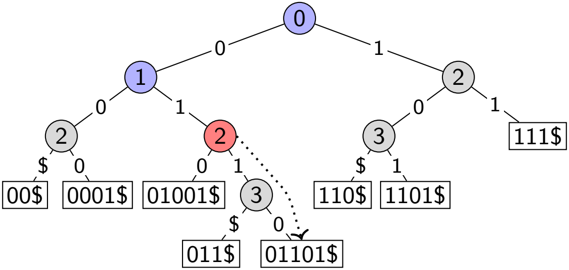
Compressed Tries: Search Example

Example 2: CompressedTrie::search(| 0 | 1 | $ |) unsuccessful (no $-child)



Compressed Tries: Search Example

Example 2: CompressedTrie::search(0 1 $) unsuccessful (no $-child)

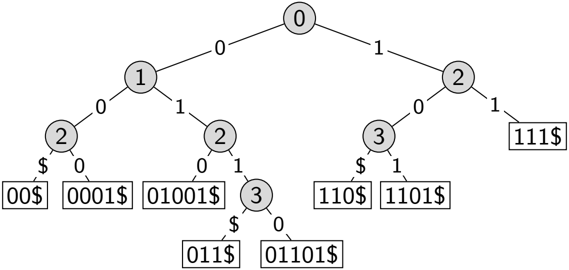


prefix-search(w): Compare w to z.*leaf* at last visited node z.

Compressed Tries: Search Example

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

0	1	2	3
1	0	1	$

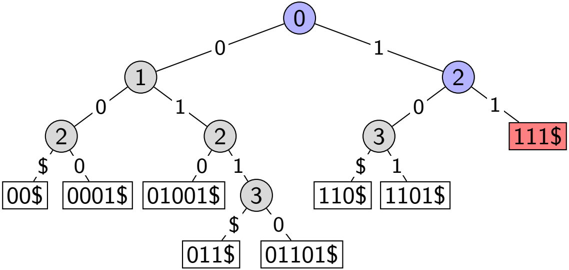


Compressed Tries: Search Example

Example 3: CompressedTrie::search(

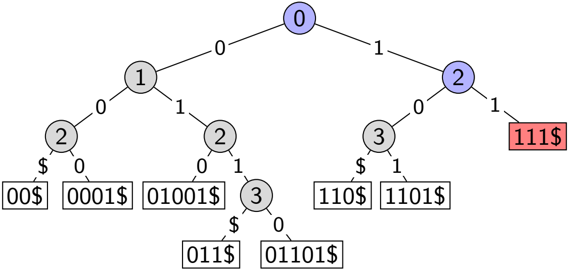
0	1	2	3
1	0	1	$

) 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 easy.
- *insert*(w) and *delete*(w) are conceptually simple:
  - 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

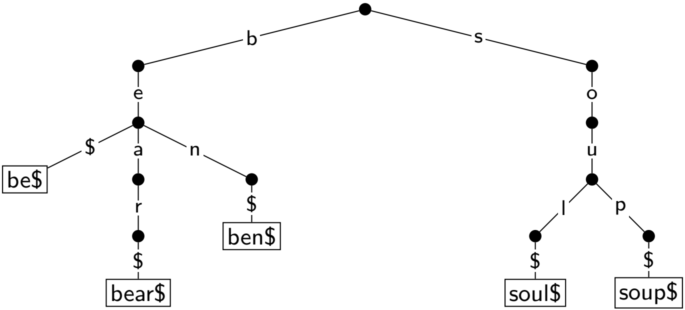
(Pseudocode gets more complicated and is omitted.)

- All operations take $O(|w|)$ time for a word w.
- Compressed tries use $O(n)$ *space*
  - We have n leaves.
  - Every internal node has two or more children.
  - **Can show:** Therefore more leaves than internal nodes.

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

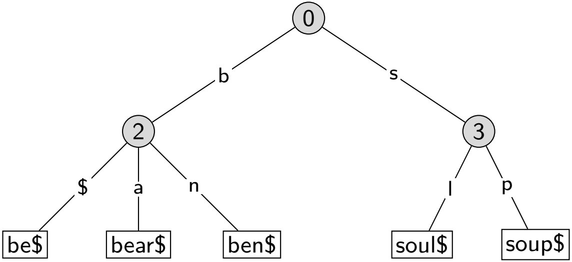
Multiway Tries: Larger Alphabet

- 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$}

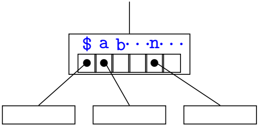


Multiway Tries: Summary

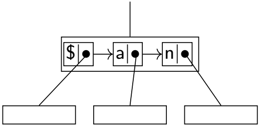
- Operations *search*(w), *prefix-search*(w), *insert*(w) and *delete*(w) are exactly as for tries for bitstrings.
- Run-time $O(|w| \cdot \text{(time to find the appropriate child)})$

Multiway Tries: Summary

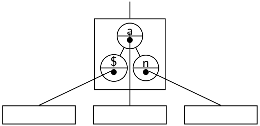
- Operations *search*(w), *prefix-search*(w), *insert*(w) and *delete*(w) are exactly as for tries for bitstrings.
- Run-time $O(|w| \cdot \text{(time to find the appropriate child)})$
- Each node now has up to $|\Sigma| + 1$ children. How should they be stored?



Array?



List?

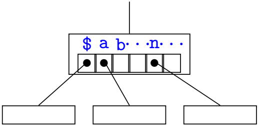


Dictionary?

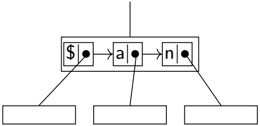
Multiway Tries: Summary

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

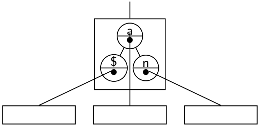
- Each node now has up to $|\Sigma| + 1$ children. How should they be stored?



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 *hashing* ($\rightarrow$ module 07).