CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

T. Biedl   É. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

version 2021-02-13 21:48

Outline

1. Lower bound
2. Interpolation Search
3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries

Lower bound for search

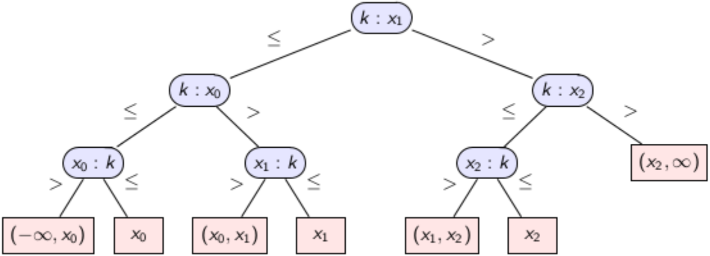
The fastest realizations of *ADT Dictionary* require $\Theta(\log n)$ time to search among n items. Is this the best possible?

Lower bound for search

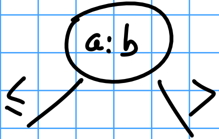
The fastest realizations of *ADT Dictionary* require $\Theta(\log n)$ time to search among n items. Is this the best possible?

Theorem: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

Proof: via decision tree



But can we beat the lower bound for special keys?



Remark: could also use

Claim: an algorithm (in the comparison model) to do search in a size-n dictionary $\Rightarrow$ a decision tree with at least $n+1$ leaves.

$x_0 < x_1$.

```
if   k ≤ x0
  if  x0 ≤ k
     return  "found", 0
  else
     return  "not found"
else
  if  k ≤ x1
    if  x1 ≤ k
      return  "found", 1
    else
       return  "not found"
  else
    return "not found"
```

$k \le x_0$

$k < x_0$

$k > x_0$

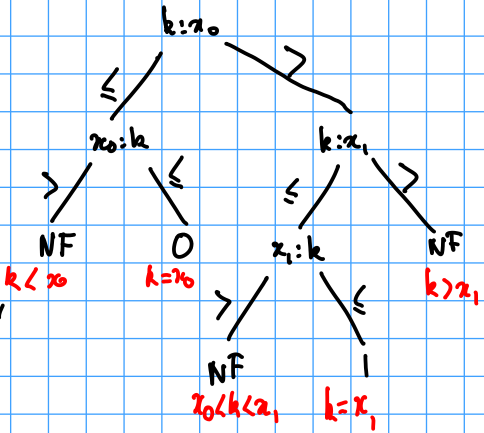
$x_0 < k \le x_1$

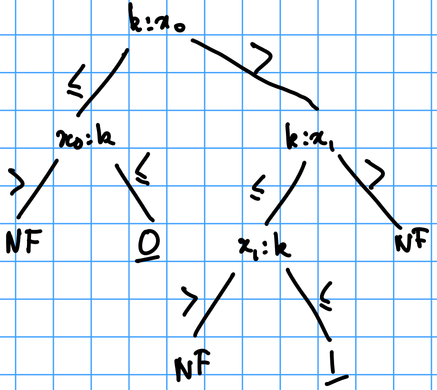
$x_0 < k < x_1$

$k > x_1$

$x_0 < x_1$.

```
if k ≤ x0
    if x0 ≤ k
        return "found", 0
    else
        return "not found"
else
    if k ≤ x1
        if x1 ≤ k
            return "found", 1
        else
            return "not found"
    else
        return "not found"
```





at least $3 = 2 + 1$ leaves

0, 1 NF

Let h be the worst-case # of comparisons that we do (for n keys)

→ any possible input k reaches a leaf after doing at most h comparisons

→ in the decision tree, there are at least $n+1$ leaves of depth $\leq h$.

In any binary tree, the number of leaves of depth $\leq h$ is at most 2^h. Proof: induction on h.

$n+1 \leq$ # of leaves of $\leq 2^h$

depth $\leq h$

$\rightarrow \quad n+1 \leq 2^h$

$\rightarrow \quad \log(n+1) \leq h.$

Binary Search

Recall the run-times in a *sorted array*:

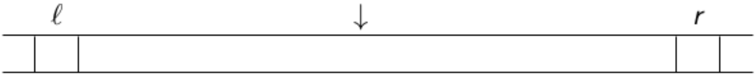
- *insert, delete*: $\Theta(n)$
- *search*: $\Theta(\log n)$

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

r ℓ=r+1

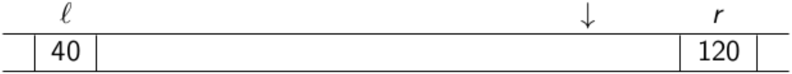
Interpolation Search: Motivation

binary-search$(A[\ell, r], k)$: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r-\ell) \rfloor$



Interpolation Search: Motivation

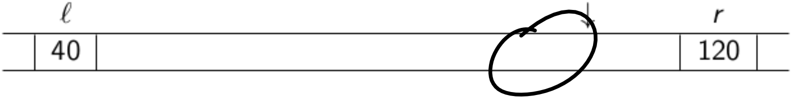
binary-search$(A[\ell, r], k)$: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r - \ell) \rfloor$



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

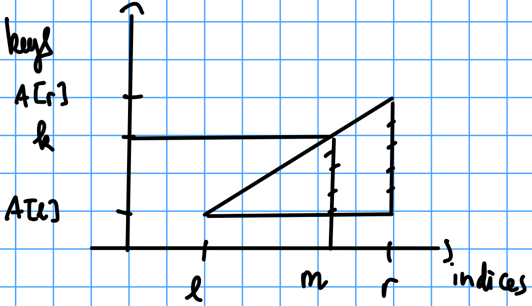
Interpolation Search: Motivation

binary-search$(A[\ell, r], k)$: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r-\ell) \rfloor$



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

interpolation-search$(A[\ell, r], k)$: Compare at index $\ell + \left\lfloor \frac{k - A[\ell]}{A[r] - A[\ell]}(r-\ell) \right\rfloor$



$$\frac{A[r]-A[l]}{r-l} = \frac{k-A[l]}{m-l}$$

$$\frac{m-l}{r-l} = \frac{k-A[l]}{A[r]-A[l]} \rightsquigarrow m = l + \frac{k-A[l]}{A[r]-A[l]}(r-l).$$

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

interpolation-search(A[0..10],449):

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500
ℓ		$\uparrow$								r

interpolation-search(A[0..10],449):

- Initially $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500
			ℓ		$\uparrow$					r

interpolation-search(A[0..10],449):

- Initially $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$
- $\ell = 3$, $r = 10$, $m = \ell + \lfloor \frac{449-3}{1500-3}(10-3) \rfloor = \ell + 2 = 5$

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500
			ℓ	$\uparrow, r$						

interpolation-search(A[0..10],449):

- Initially $\ell = 0,\ r = n-1 = 10,\ m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$
- $\ell = 3,\ r = 10,\ m = \ell + \lfloor \frac{449-3}{1500-3}(10-3) \rfloor = \ell + 2 = 5$
- $\ell = 3,\ r = 4,\ m = \ell + \lfloor \frac{449-3}{449-3}(4-3) \rfloor = \ell + 1 = 4$, found at $A[4]$

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

interpolation-search(A[0..10],449):

- Initially $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$
- $\ell = 3$, $r = 10$, $m = \ell + \lfloor \frac{449-3}{1500-3}(10-3) \rfloor = \ell + 2 = 5$
- $\ell = 3$, $r = 4$, $m = \ell + \lfloor \frac{449-3}{449-3}(4-3) \rfloor = \ell + 1 = 4$, found at $A[4]$

Works well if keys are *uniformly* distributed:

- Can show: Recurrence relation is $T^{(\text{avg})}(n) = T^{(\text{avg})}(\sqrt{n}) + \Theta(1)$.
- This resolves to $T^{(\text{avg})}(n) \in \Theta(\log \log n)$.

But: Worst case performance $\Theta(n)$

Interpolation Search Example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

interpolation-search(A[0..10],449):

- Initially $\ell = 0,\ r = n-1 = 10,\ m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$
- $\ell = 3,\ r = 10,\ m = \ell + \lfloor \frac{449-3}{1500-3}(10-3) \rfloor = \ell + 2 = 5$
- $\ell = 3,\ r = 4,\ m = \ell + \lfloor \frac{449-3}{449-3}(4-3) \rfloor = \ell + 1 = 4$, found at $A[4]$

Works well if keys are *uniformly* distributed:

- Can show: Recurrence relation is $T^{(\text{avg})}(n) = T^{(\text{avg})}(\sqrt{n}) + \Theta(1)$.
- This resolves to $T^{(\text{avg})}(n) \in \Theta(\log \log n)$.

But: Worst case performance $\Theta(n)$

$$T(n) = T(\sqrt{n}) + c$$

$$2 \to 2^2 \to 2^4 \to 2^8 \to 2^{16} \to \dots \to \underline{2^{2^i}} \to \dots$$

$$\to T(2^{2^i}) = T(2^{2^{i-1}}) + c$$

$$= T(2^{2^{i-2}}) + 2c$$

$$= T(2^{2^{i-3}}) + 3c$$

$$= \dots = T(2^{2^{i-i}}) + ic = T(2) + ic$$

$\to$ for $n = 2^{2^i}$, $T(n) = T(2) + c \cdot \log(\log(n))$.

Interpolation Search

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash during computation of m.

interpolation-search(A, n, k)
A: Sorted array of size n, k: key

$\ell \leftarrow 0$, $r \leftarrow n-1$
while $(\ell \leq r)$
  if $(k < A[\ell]$ or $k > A[r])$ **return** "not found"
  if $(A[\ell] = A[r])$ **then return** "found at $A[\ell]$"
  $m \leftarrow \ell + \lfloor \frac{k - A[\ell])}{A[r]-A[\ell]} \cdot (r - \ell) \rfloor$
  if $(A[m] < k)$ **then** $\ell = m + 1$
  else if $(k < A[m])$ **then** $r = m - 1$
  else return "found at $A[m]$"
// We always return from somewhere within while-loop

Outline

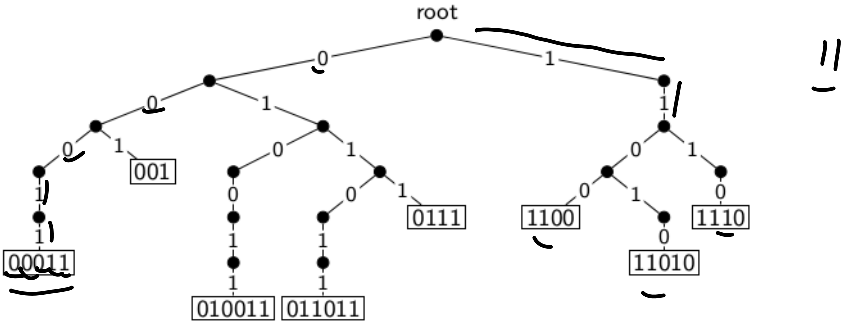
1. Lower bound
2. Interpolation Search
3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries

Tries: Introduction

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

(Should know: string, word, $|w|$, alphabet, prefix, suffix, comparing words,....)

- 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

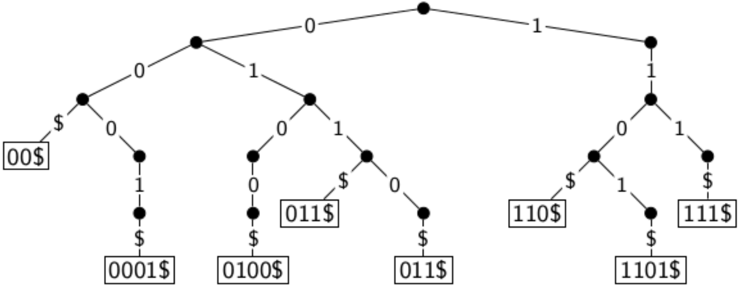


More on tries

Assumption: Dictionary is **prefix-free**: no string is a prefix of another

- Assumption satisfied if all strings have the same length.
- Assumption satisfied if all strings end with 'end-of-word' character \$.

Example: A trie for $\{00\$, 0001\$, 0100\$, 011\$, 0110\$, 110\$, 1101\$, 111\$\}$

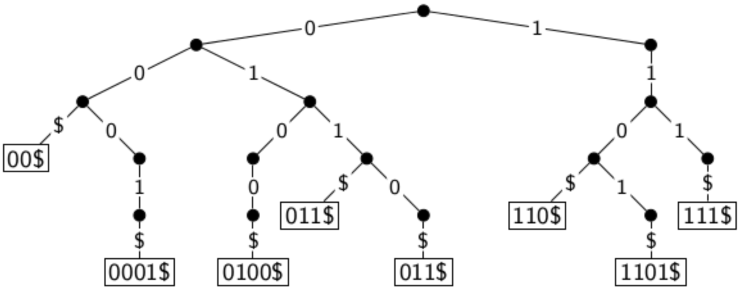


More on tries

Assumption: Dictionary is **prefix-free**: no string is a prefix of another

- Assumption satisfied if all strings have the same length.
- Assumption satisfied if all strings end with 'end-of-word' character $.

Example: A trie for {00$, 0001$, 0100$, 011$, 0110$, 110$, 1101$, 111$}



Then items (keys) are stored *only* in the leaf nodes

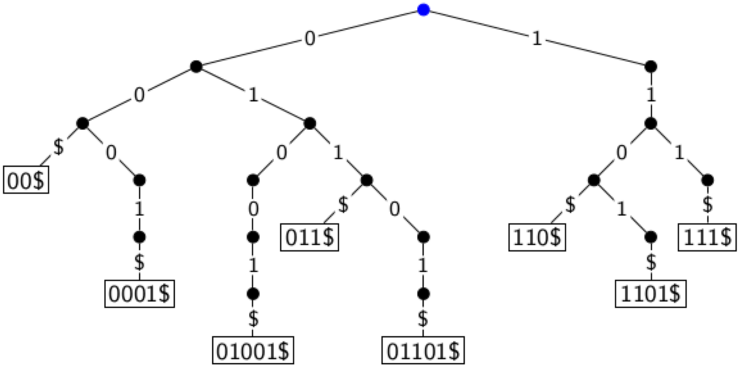
Tries: Search

- start from the root and the most significant bit of x
- follow the link that corresponds to the current bit in x; return failure if the link is missing
- return success if we reach a leaf (it must store x)
- else recurse on the new node and the next bit of x

```
Trie::search(v ← root, d ← 0, x)
v: node of trie; d: level of v, x: word stored as array of chars
    if v is a leaf
        return v
    else
        let v′ be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Trie::search(v′, d + 1, x)
```

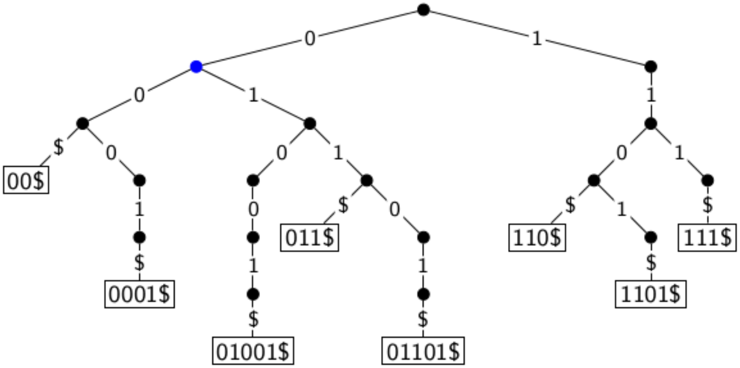
Tries: Search Example

Example: Trie::search(011$)



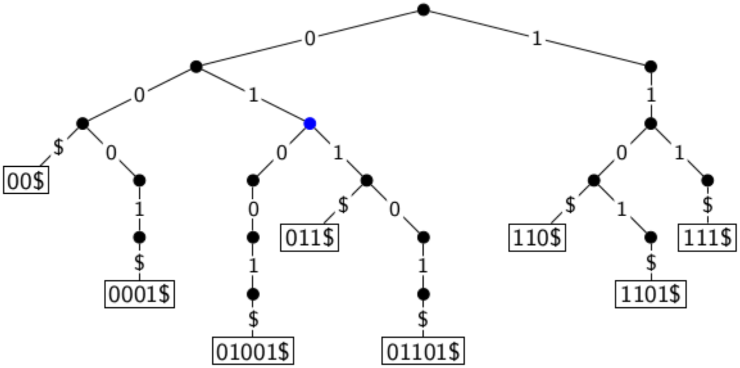
Tries: Search Example

Example: Trie::search(011$)



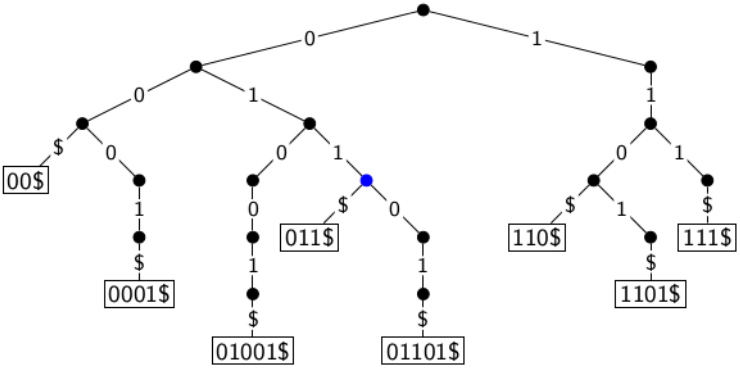
Tries: Search Example

Example: Trie::search(011$)



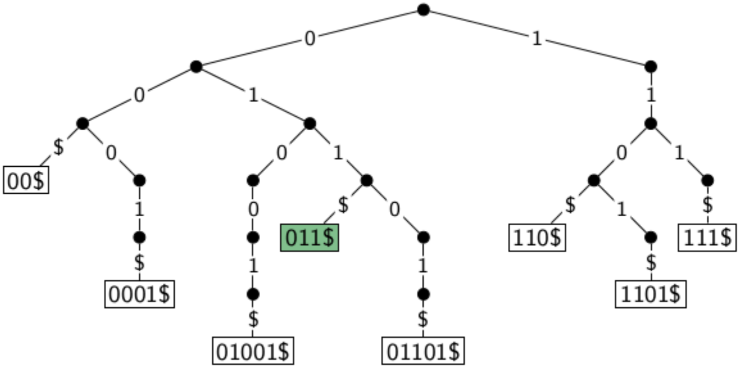
Tries: Search Example

Example: Trie::search(011$)



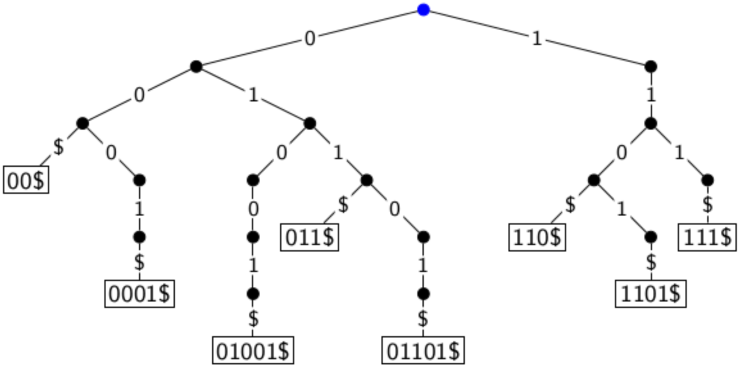
Tries: Search Example

Example: Trie::search(011$) successful



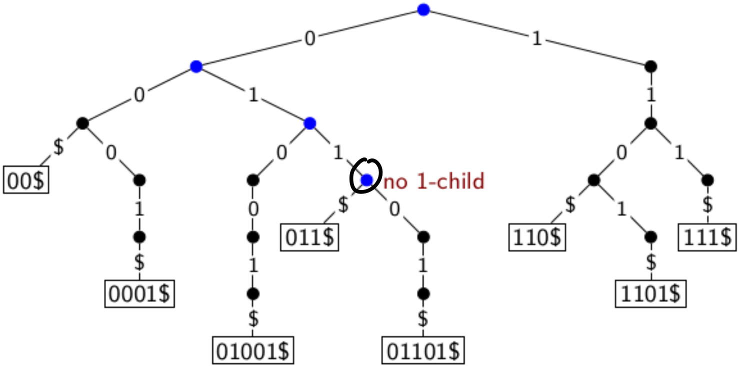
Tries: Search Example

Example: Trie::search(0111$)



Tries: Search Example

Example: Trie::search(0111$) unsuccessful

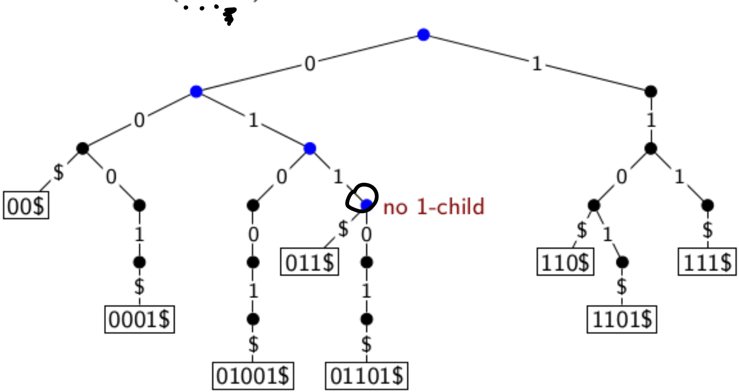


Tries: Insert & Delete

- *Trie::insert(x)*
  - Search for x, this should be unsuccessful
  - Suppose we finish at a node v that is missing a suitable child. Note: x has extra bits left.
  - Expand the trie from the node v by adding necessary nodes that correspond to extra bits of x.
- *Trie::delete*(x)
  - Search for x
  - let v be the leaf where x is found
  - delete v and all ancestors of v until we reach an ancestor that has two children.
- **Time Complexity** of all operations: $\Theta(|x|)$
  $|x|$: length of binary string x, i.e., the number of bits in x

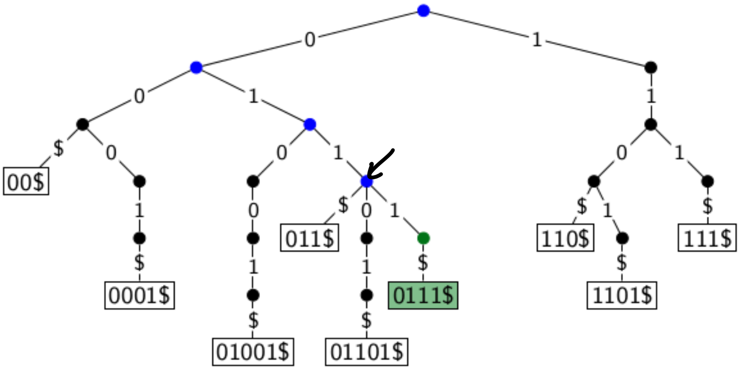
Tries: Insert Example

Example: *Trie::insert*(0111$)



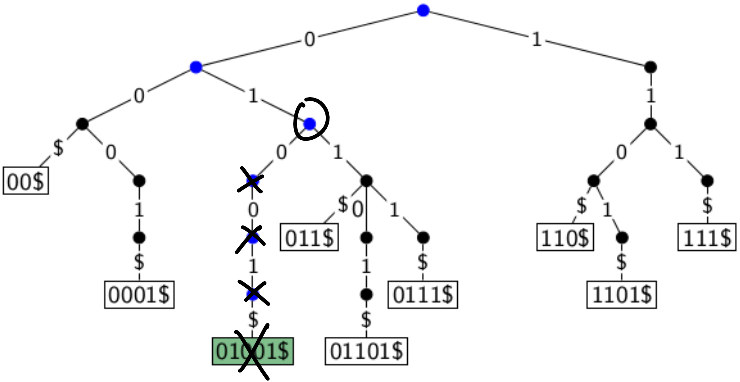
Tries: Insert Example

Example: *Trie::insert*(0111\$)



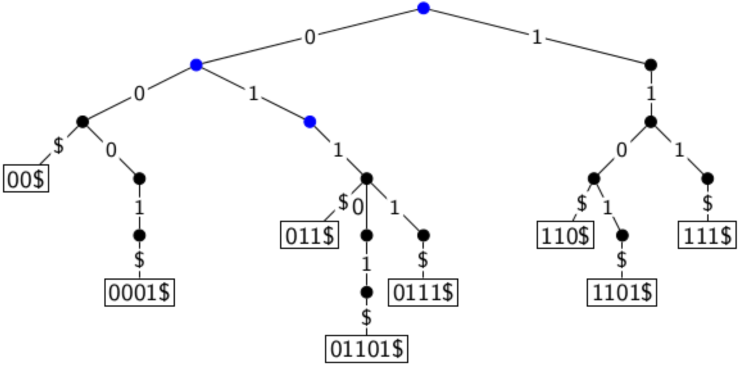
Tries: Delete Example

Example: *Trie::delete*(01001$)



Tries: Delete Example

Example: *Trie::delete*(01001$)

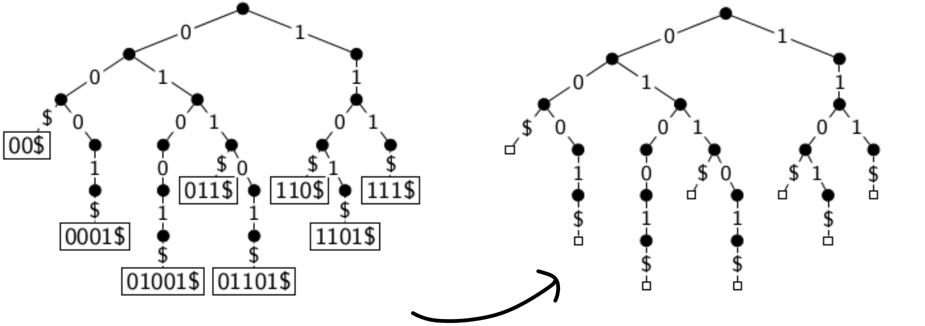


Outline

Variation 1 of Tries: No leaf labels

Do not store actual keys at the leaves.

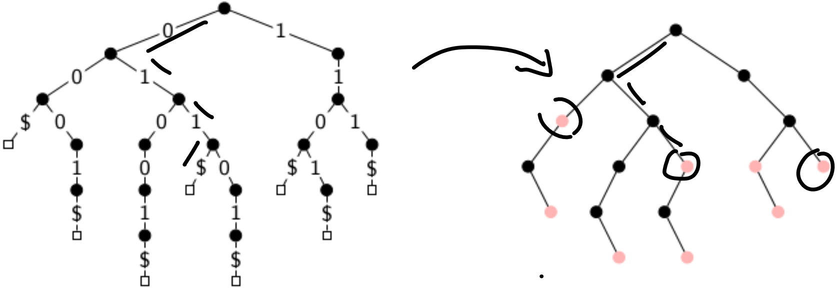
- The key is stored implicitly through the characters along the path to the leaf. It therefore need not be stored again.

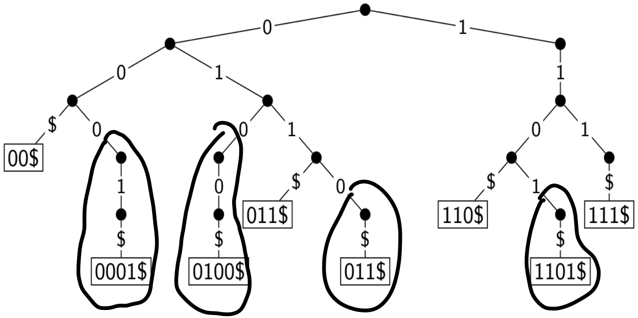


Variation 2 of Tries: Allow Proper Prefixes

Allow prefixes to be in dictionary.

- Internal nodes may now also represent keys.
  Use a *flag* to indicate such nodes.
- No need for end-of-word character $
- Now a trie of bitstrings is a binary tree. Can express 0-child and 1-child implicitly via left and right child.
- More space-efficient.



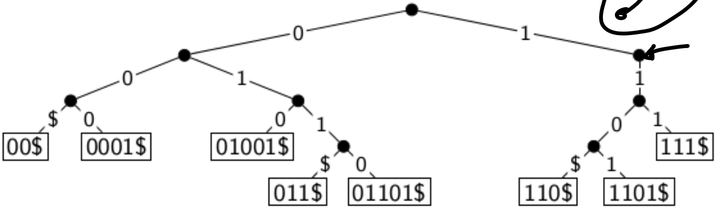


0
1
0
1
1
$
0
0
1
0
1
00$
1
0
$
0
$
1
$
011$
110$
111$
$
$
$
$
0001$
0100$
011$
1101$

Variations 3 of 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.
- Note that now we *must* store the full keys (why?)
- Saves space if there are only few bitstrings that are long.
- Could even store infinite bitstrings (e.g. real numbers)



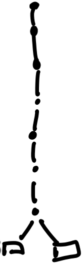
This is in practice the most efficient version of tries, but the operations get a bit more complicated.

Outline

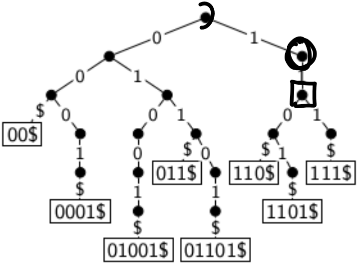
Variation 4 of Tries

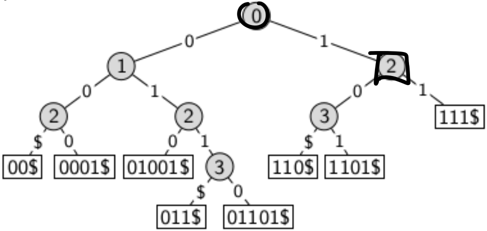
Compressed Trie: compress paths of nodes with only one child

- Each node stores an *index*, corresponding to the depth in the uncompressed trie.
  - This gives the next bit to be tested during a search
- A compressed trie with n keys has at most $n-1$ internal nodes



→ at most 2n−1 nodes





Also known as **Patricia-Tries**:

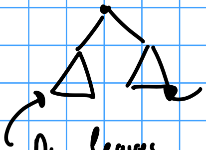
Practical Algorithm to Retrieve Information Coded in Alphanumeric

Proof: by induction on the height.

1) if $h = 0$

$n = 1$, no internal nodes $\to$ OK.

2) Suppose true for $0 \dots h-1$; prove it for a trie of height h.



n_2 leaves
at most $n_2 - 1$ internal nodes.

n_1 leaves
at most $n_1 - 1$ internal nodes

in the whole tree:

- # leaves $= n_1 + n_2$
- # internal nodes = # internal nodes on the left *

  + # internal nodes on the right **

  +1

  $\leq n_1 - 1 + n_2 - 1 + 1 = \underbrace{n_1 + n_2}_{\text{\#leaves}} - 1$

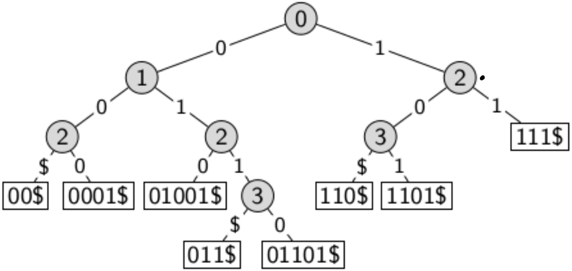
Compressed Tries: Search

- start from the root and the bit indicated at that node
- follow the link that corresponds to the current bit in x; return failure if the link is missing
- if we reach a leaf, expicitly check whether word stored at leaf is x
- else recurse on the new node and the next bit of x

```
CompressedTrie::search(v ← root, x)
v: node of trie; x: word
    if v is a leaf
        return strcmp(x, v.key)
    d ← index stored at v
    if x has at most d bits
        return "not found"
    v′ ← child of v labelled with x[d]
    if there is no such child
        return "not found"
    CompressedTrie::search(v′, x)
```

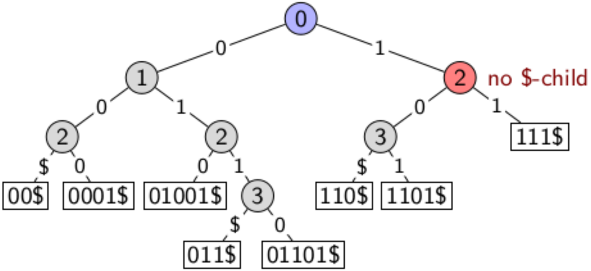
Compressed Tries: Search Example

Example: CompressedTrie::search(10$)



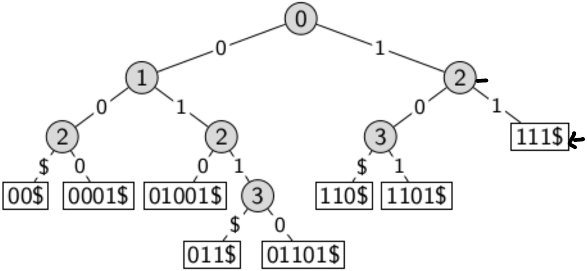
Compressed Tries: Search Example

Example: CompressedTrie::search(10$) unsuccessful



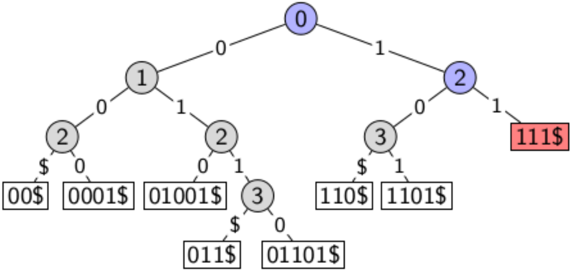
Compressed Tries: Search Example

Example: CompressedTrie::search(101$)



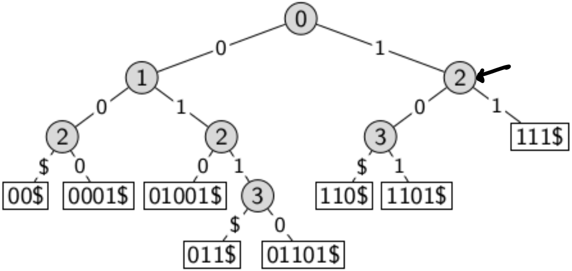
Compressed Tries: Search Example

Example: CompressedTrie::search(101$) unsuccessful



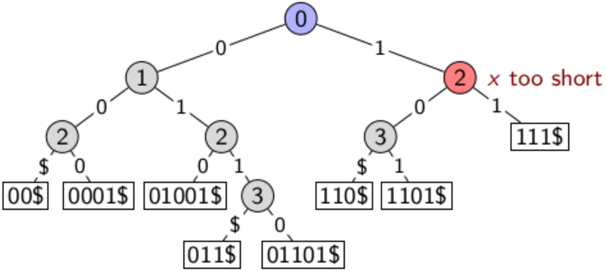
Compressed Tries: Search Example

Example: CompressedTrie::search(1$)



Compressed Tries: Search Example

Example: CompressedTrie::search(1\$) unsuccessful



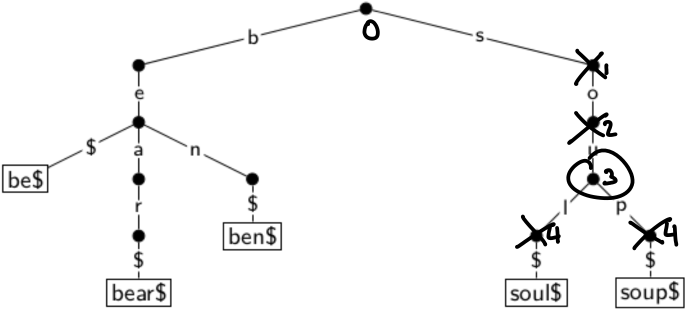
Compressed Tries: Insert & Delete

- *CompressedTrie::delete*(x):
  - Perform *search*(x)
  - Remove the node v that stored x
  - Compress along path to v whenever possible.
- *CompressedTrie::insert*(x):
  - Perform *search*(x)
  - Let v be the node where the search ended.
  - Conceptually simplest approach:
    - Uncompress path from root to v.
    - Insert x as in an uncompressed trie.
    - Compress paths from root to v and from root to x.

    But it can also be done by only adding those nodes that are needed, see the textbook for details.
- All operations take $O(|x|)$ time.

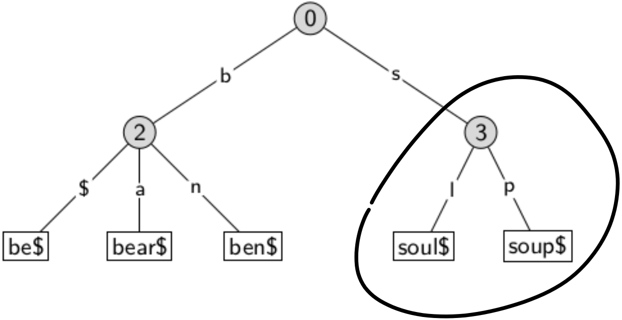
Multiway Tries: Larger Alphabet

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

Multiway Tries: Summary

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

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

Multiway Tries: Summary

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

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

Solution 1: Array of size $|\Sigma| + 1$ for each node.
Complexity: $O(1)$ time to find child, $O(|\Sigma|)$ space per node.

Solution 2: List of children for each node.
Complexity: $O(|\Sigma|)$ time to find child, $O(\#\text{children})$ space.

Solution 3: Dictionary (AVL-tree?) of children for each node.
Complexity: $O(\log(\#\text{children}))$ time, $O(\#\text{children})$ space.

Best in theory, but not worth it in practice unless $|\Sigma|$ is huge.

In practice, use *hashing* (keys are in (typically small) range Σ).

Multiway Tries: Summary

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

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

Solution 1: Array of size $|\Sigma| + 1$ for each node.
Complexity: $O(1)$ time to find child, $O(|\Sigma|)$ space per node.

Solution 2: List of children for each node.
Complexity: $O(|\Sigma|)$ time to find child, $O(\#\text{children})$ space.

Solution 3: Dictionary (AVL-tree?) of children for each node.
Complexity: $O(\log(\#\text{children}))$ time, $O(\#\text{children})$ space.

Best in theory, but not worth it in practice unless $|\Sigma|$ is huge.

In practice, use *hashing* (keys are in (typically small) range Σ).