

CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

Mark Petrick

Based on lecture notes by many previous cs240 instructors

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

Fall 2020

References: Sedgewick 12.4, 15.2-15.4
Goodrich & Tamassia 23.5.1-23.5.2

Outline

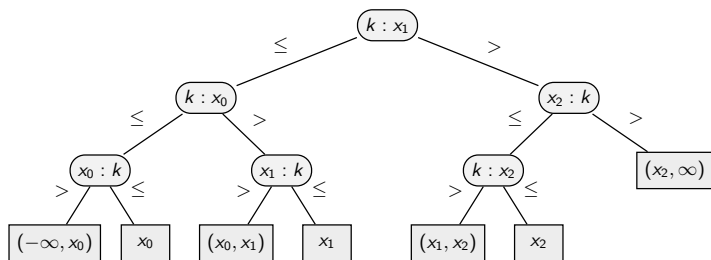
- ① Lower bound
- ② Interpolation Search
- ③ Tries
 - Standard Tries
 - Variations of Tries
 - Compressed Tries

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?

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?

Binary Search

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

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

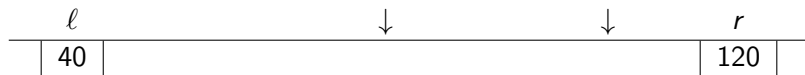
Binary-search(A, n, k)

A : Sorted array of size n , k : key

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

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 $\ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]}(r - \ell) \rfloor$

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: the array in which we recurse into has size \sqrt{n} on average.
- 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

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash due to $A[\ell] = A[r]$

interpolation-search(A, n, k)

A: Sorted array of size n , k : key

1. $\ell \leftarrow 0$
2. $r \leftarrow n - 1$
3. **while** ($\ell < r$) $\&\&$ ($A[r] \neq A[\ell]$) $\&\&$ ($k \geq A[\ell]$) $\&\&$ ($k \leq A[r]$)
4. $m \leftarrow \ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \rfloor$
5. **if** ($A[m] < k$) **then** $\ell = m + 1$
6. **else if** ($k < A[m]$) **then** $r = m - 1$
7. **else return** m
8. **if** ($k = A[\ell]$) **return** ℓ
9. **else return** “not found, but would be between $\ell - 1$ and ℓ ”

Outline

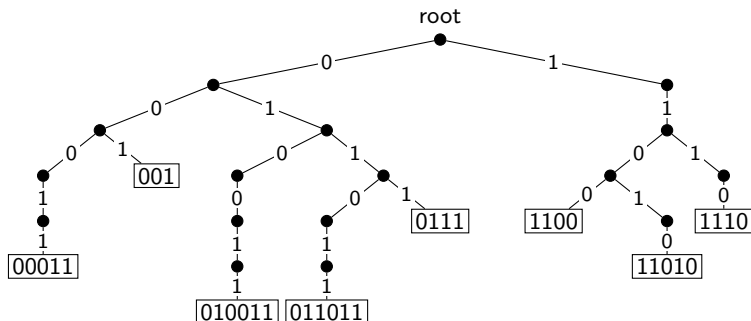
- ① Lower bound
- ② Interpolation Search
- ③ Tries
 - Standard Tries
 - Variations of Tries
 - Compressed Tries

Tries: Introduction

Trie (also known 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

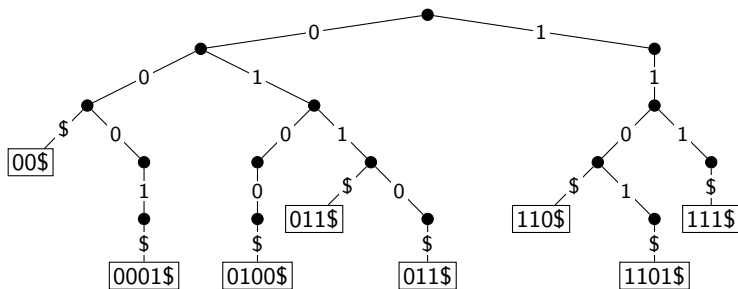


More on tries

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\$}



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

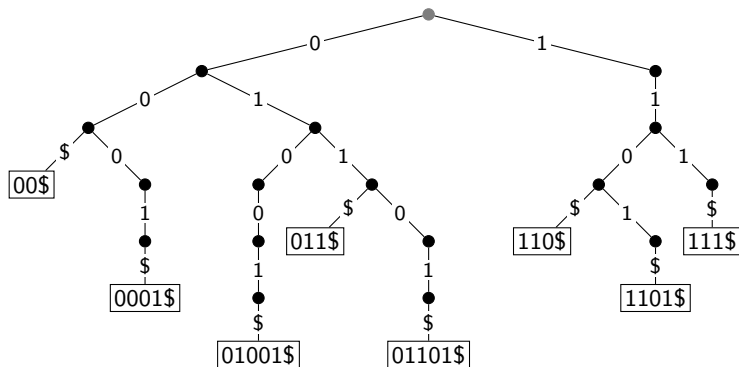
Trie::search($v \leftarrow \text{root}, d \leftarrow 0, x$)

v : node of trie; d : level of v , x : word stored as array of chars

1. **if** v is a leaf
2. **return** v
3. **else**
4. let v' be child of v labelled with $x[d]$
5. **if** there is no such child
6. **return** "not found"
7. **else** *Trie::search*($v', d + 1, x$)

Tries: Search Example

Example: Trie::search(011\$)

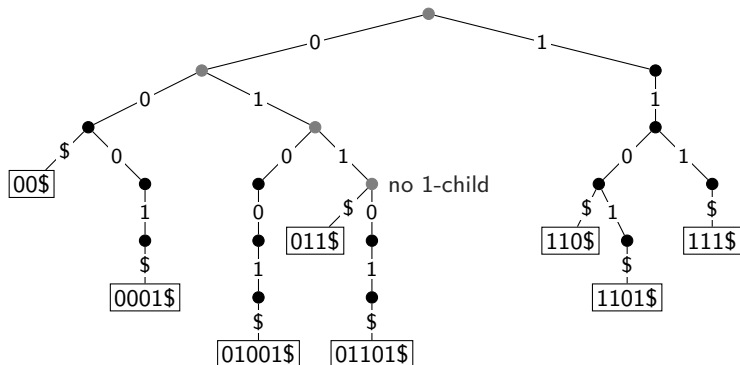


Tries: Insert & Delete

- *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\$)



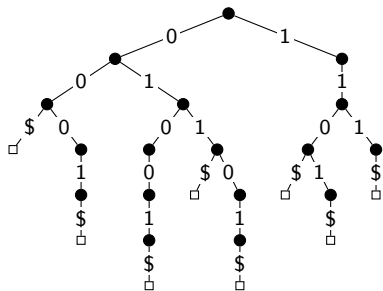
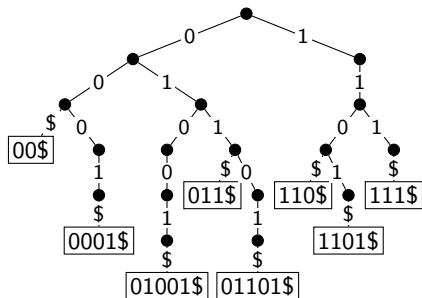
Outline

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

Variation 1 of Tries: No leaf labels

Do not store actual keys at the leaves.

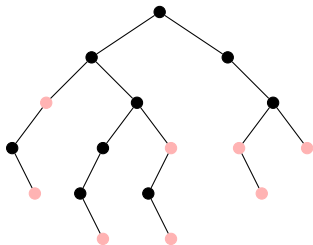
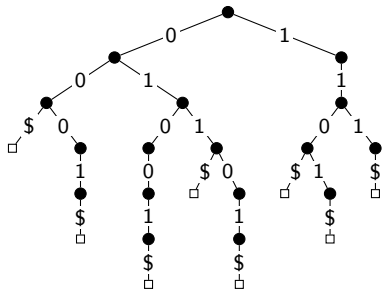
- The key is stored implicitly through the characters along the path to the leaf. It therefore need not be stored again.
- This halves the amount of space needed.



Variation 2 of Tries: Allow Proper Prefixes

Allow prefixes to be in dictionary.

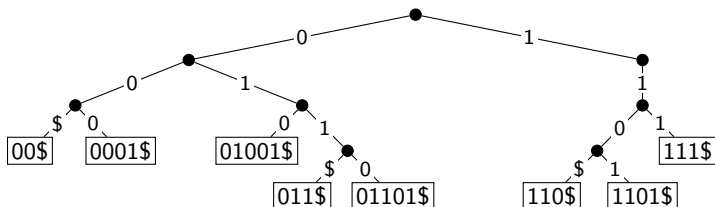
- Internal nodes may now also represent keys. Use a *flag* to indicate such nodes.
- No need for end-of-word character \$
- Now a trie of bitstrings is a binary tree. Can express 0-child and 1-child implicitly via left and right child.
- More space-efficient.



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.

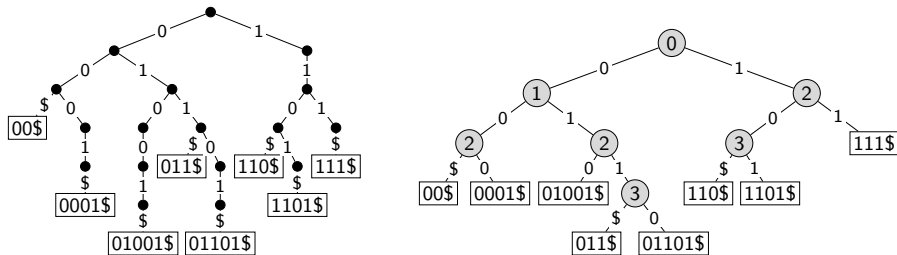
Outline

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

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



Also known as **Patricia-Tries:**

Practical Algorithm to Retrieve Information Coded in Alphanumeric

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, explicitly check whether word stored at leaf is x
- else recurse on the new node and the next bit of x

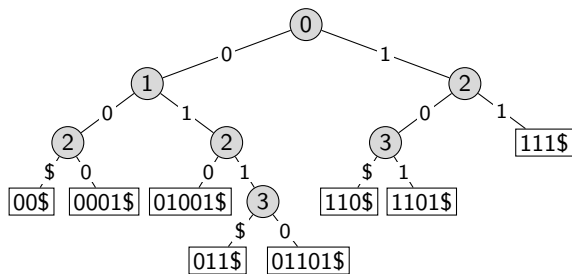
```
CompressedTrie::search( $v \leftarrow \text{root}, x$ )
```

v : node of trie; x : word

1. **if** v is a leaf
2. **return** *strcmp*($x, v.\text{key}$)
3. $d \leftarrow$ index stored at v
4. **if** x has at most d bits
5. **return** "not found"
6. $v' \leftarrow$ child of v labelled with $x[d]$
7. **if** there is no such child
8. **return** "not found"
9. *CompressedTrie::search*(v', x)

Compressed Tries: Search Example

Example: `CompressedTrie::search(10$)` unsuccessful



Compressed Tries: Insert & Delete

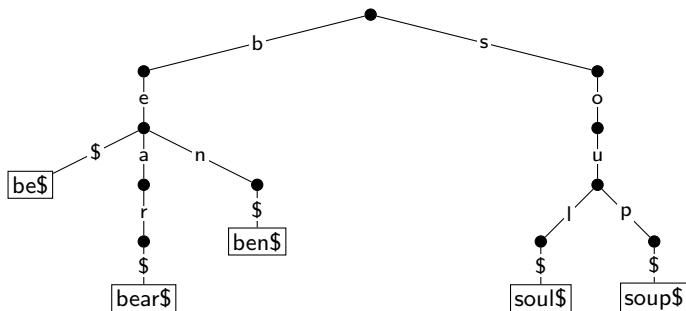
- *CompressedTrie::delete*(x):
 - ▶ Perform *search*(x)
 - ▶ Remove the node v that stored x
 - ▶ Compress along path to v whenever possible.
- *CompressedTrie::insert*(x):
 - ▶ Perform *search*(x)
 - ▶ Let v be the node where the search ended.
 - ▶ Conceptually simplest approach:
 - ★ Uncompress path 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.

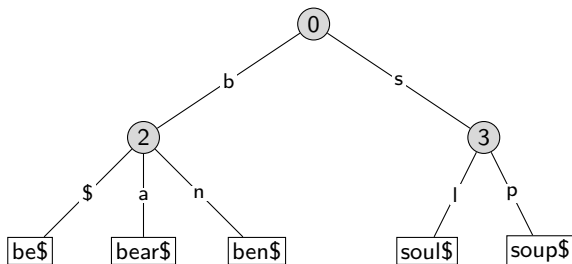
Multiway Tries: Larger Alphabet

- 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 $O(|x| \cdot (\text{time to find the appropriate child}))$

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, $O(|\Sigma|n)$ space.

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).