# CS 240 – Data Structures and Data Management

## Module 6: Dictionaries for special keys

A. Jamshidpey G. Kamath É. Schost
Based on lecture notes by many previous cs240 instructors

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

Spring 2020

References: Sedgewick 12.4, 15.2-15.4
Goodrich & Tamassia 9.2.1-9.2.2

*version 2020−06−23 00:15*

# Outline

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

# Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ 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:

But can we beat the lower bound for special keys?

# Interpolation Search: Motivation

Ordered array

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

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

| | $\ell$ | | $\downarrow$ | $\downarrow$ | $r$ | |
|---|---|---|---|---|---|---|
| | 40 | | | | 120 | |

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

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

# 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 |

*Search*(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 expected size $\sqrt{n}$.
- 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*$(A, n, k)$
$A$: Array of size $n$, $k$: key

$\ell \leftarrow 0$
$r \leftarrow n - 1$
**while** $((A[r] != A[\ell]) \&\& (k \geq A[\ell]) \&\& (k \leq A[r]))$
    $m \leftarrow \ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \rfloor$
    **if** $(A[m] < k)$ $\ell = m + 1$
    **elsif** $(k < A[m])$ $r = m - 1$
    **else** return $m$
**if** $(k = A[\ell])$ return $\ell$
**else** return "not found"

# Outline

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

# Tries: Introduction

- **Trie (Radix Tree)**: A dictionary for binary strings
  - Comes from retrieval, but pronounced "try"
  - A tree based on **bitwise comparisons**
  - Similar to **radix sort**: use individual bits, not the whole key
- Keys can have different number of bits

**Prefix** of a string $S[0..n-1]$: a substring $S[0..i]$ of $S$ for some $0 \leq i \leq n-1$.

**Prefix-free**: there is no pair of binary strings in the dictionary where one is the prefix of the other.

Assumption: Dictionary is prefix-free:

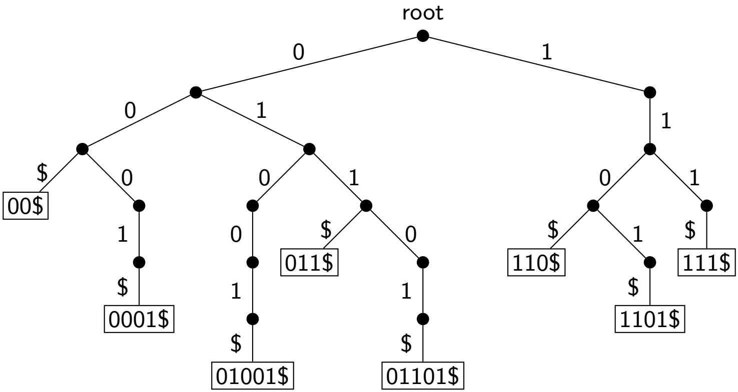
- This is always satisfied if all strings have the same length.
- This is always satisfied if all strings end with a special 'end-of-word' character \$.

# Tries: structure

Structure of trie:

- Items (keys) are stored **only** in the leaf nodes
- Edge to child is labelled with corresponding bit or \$

Example: A trie for
$S = \{00\$, 0001\$, 01001\$, 011\$, 01101\$, 110\$, 1101\$, 111\$\}$



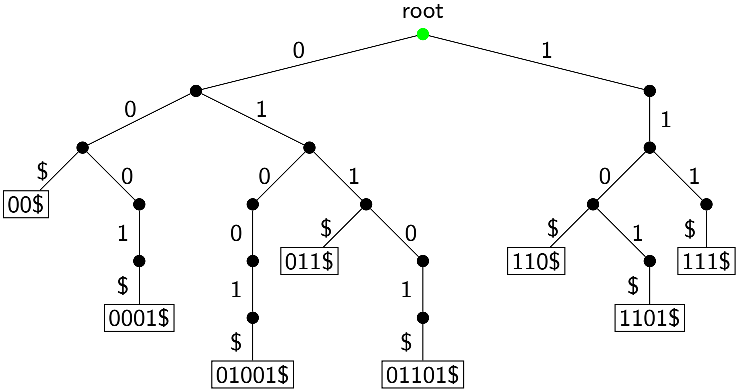
# Tries: Search

- 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
    if v is a leaf
        return v
    else
        let c be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Trie-search(c, d + 1, x)
```

# Tries: Search Example

Example: Search(011$)

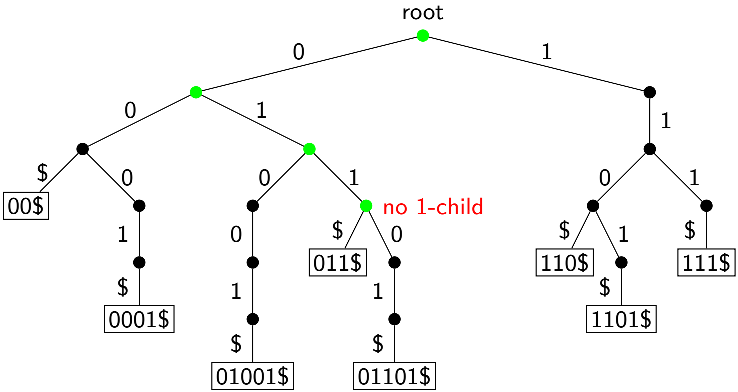


# Tries: Insert & Delete

- **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 $x$.
- **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 $x$

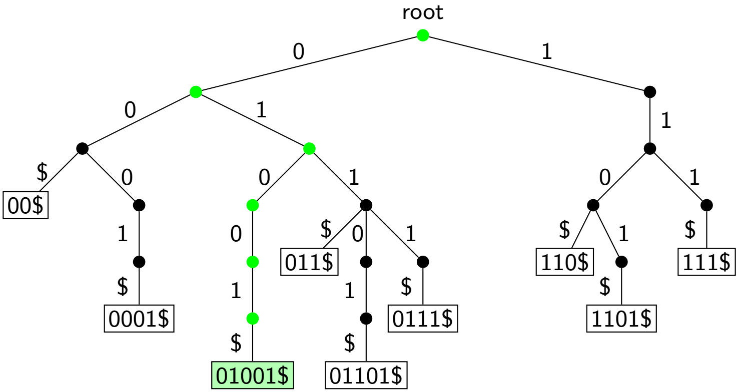
# Tries: Insert Example

Example: Insert(0111$)



# Tries: Delete Example

Example: Delete(01001$)



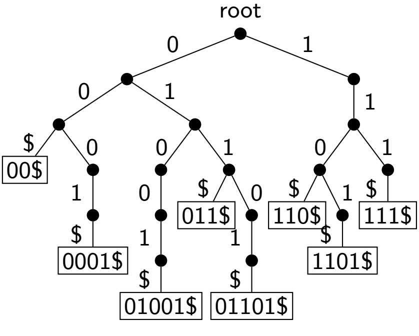
# Outline

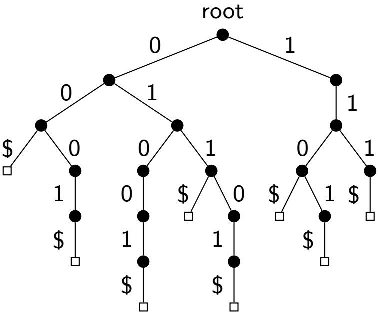
1. Lower bound
2. Interpolation Search
3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries

# 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.

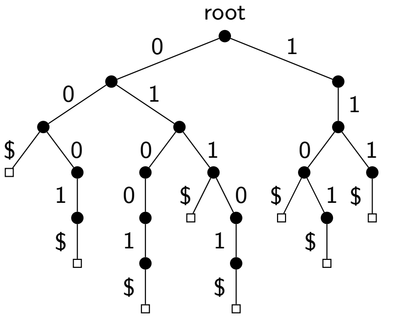


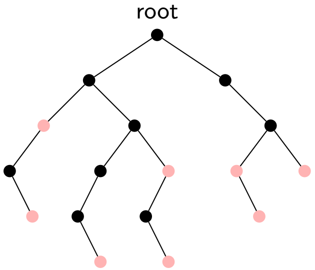



# Variation 2 of Tries: Allow Proper Prefixes

Allow prefixes to be in dictionary.

- Then internal nodes may also represent keys. We then use a *flag* at each node to indicate whether this represents a key of the dictionary.
- This replaces the reference to the \$-child with a flag (one bit) and saves space.
- For bitstrings, we then do not need \$, and so every node has at most two children. Can express 0-child and 1-child implicitly via left and right child.



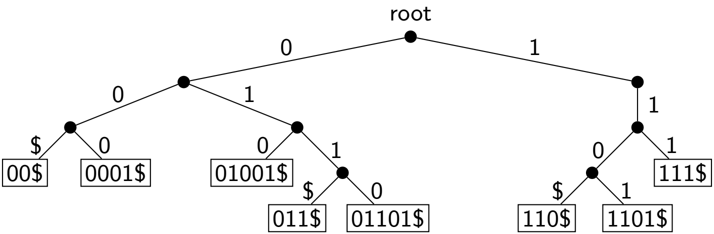



# Variations 3 of Tries: Remove Chains to Labels

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.
- Note that this variation *cannot* be combined with the the first variation (why not?)

This variation is the one presented in Sedgewick.



# Outline

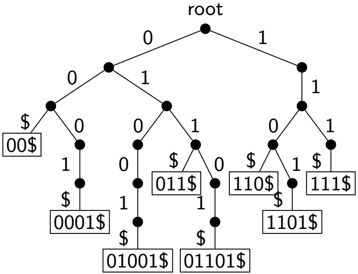
1. Lower bound
2. Interpolation Search
3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries

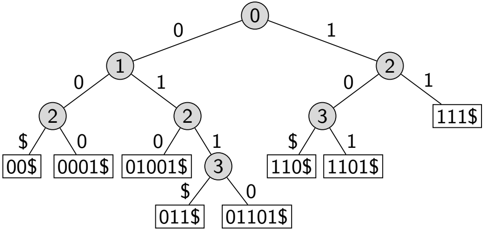
# Compressed Tries (Patricia Tries)

- Morrison (1968):
  *Patricia: Practical Algorithm to Retrieve Information Coded in Alphanumeric*
- Compress paths of nodes with only one child
- Each node stores an *index* indicating the next bit to be tested during a search (index $= 0$ for the first bit, index $= 1$ for the second bit, etc.)
- A compressed trie storing $n$ keys always has at most $n - 1$ internal (non-leaf) nodes

# Compressed Tries Example

Example: A trie and the equivalent compressed trie.



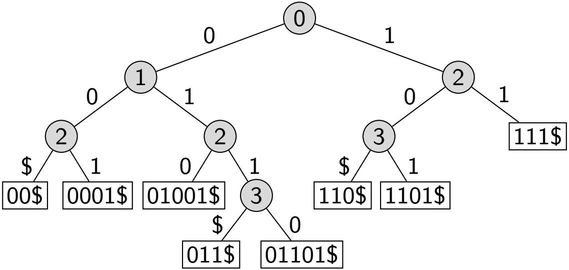
# Compressed Tries: Search

- 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 $x$

```
Patricia-Trie-search(v ← root, x)
v: node of trie; x: word
    if v is a leaf
        return strcmp(x, key(v))
    else
        let d be the bit stored at v
        let c be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Patricia-Trie-search(c, x)
```

# Compressed Tries: Search Example

Example: Search(10$) **unsuccessful**



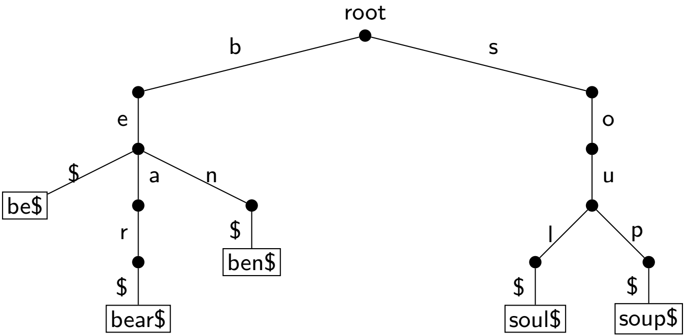
# Compressed Tries: Insert & Delete

- **Delete($x$)**:
  - Perform Search($x$)
  - Remove the node $v$ that stored $x$
  - Compress along path to $v$ whenever possible.
- **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.

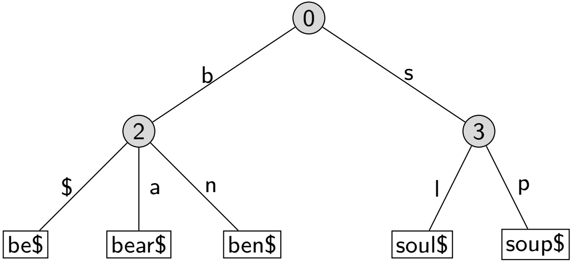
# Multiway Tries: Larger Alphabet

- To represent **Strings** over any **fixed alphabet** $\Sigma$
- 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

- **Compressed** multi-way tries
- 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 $|\Sigma| + 1$ references to children. How should they be stored?
  - Could store array of size $|\Sigma| + 1$ at each node. $O(1)$ time to find the appropriate child, but then for $n$ nodes the total space is $O(|\Sigma| n)$.
  - Could store list of children at each node. Then total space is $O(n)$, but time to find child increases to $O(|\Sigma|)$. Use MTF!
  - Could use some good dictionary implementation (AVL-tree? Skip list?) at each node. Then the total space is $O(n)$ and the time to find a child is $O(\log |\Sigma|)$ or better. Best in theory, but in practice not worth it unless the alphabet is huge.