CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2023

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard
    - Variations of Tries
    - Compressed Tries
    - Multiway Tries

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard
    - Variations of Tries
    - Compressed Tries
    - Multiway Tries

Lower bound for search

- Search is $\Theta(\log n)$ in fastest implementations of dictionary ADT
  - n is the number of items stored
- Is this the best possible?

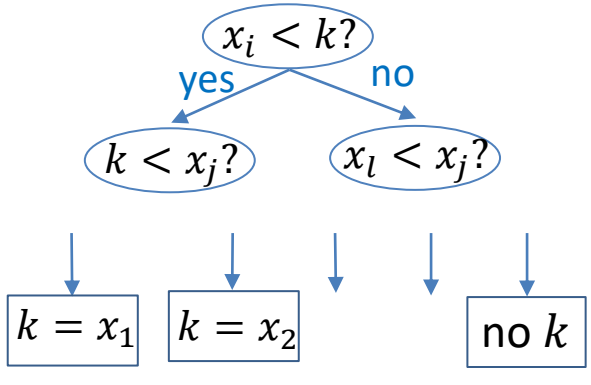
Theorem: $\Omega(\log n)$ comparisons required for search in comparison based model

Proof:

- consider binary decision tree
- leaves correspond to answers returned
- decision tree must have at least $(n + 1)$ leaves
  - $+1$ for "no key found"
- binary tree of height h has at most 2^h leaves
- thus

$$2^h \geq n + 1$$

$$h \geq \log(n + 1)$$



- Can we beat the lower bound if keys are special? Yes!
  1. Interpolation search: keys have special distribution
  2. Tries: keys are strings

Outline

- Lower bound for search
- Interpolation Search
- Tries
  - Standard
  - Variations of Tries
  - Compressed Tries
  - Multiway Tries

Binary Search on Ordered Array

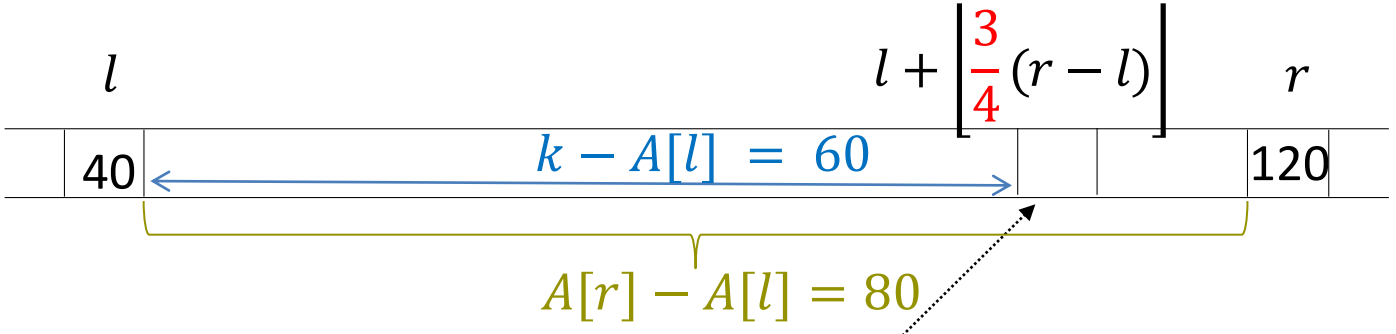
- insert and delete: $\Theta(n)$, search is $\Theta(\log n)$

Binary-search(A, n, k)
A: Array of size n, k: key

$l \leftarrow 0$
$r \leftarrow n - 1$
while $(l \leq r)$
$m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$
if $(k == A[m])$ **return** "found at $A[m]$"
else if $(A[m] < k)$ // key cannot be in the left part of A
$l \leftarrow m + 1$
else $r \leftarrow m - 1$ // key cannot be in the right part of A
else return m
return "not found but would be between $A[l-1]$ and $A[l]$ "

Interpolation Search: Motivation

- binary search looks at index $\left\lfloor \frac{l+r}{2} \right\rfloor = l + \left\lfloor \frac{1}{2}(r-l) \right\rfloor$ (middle)



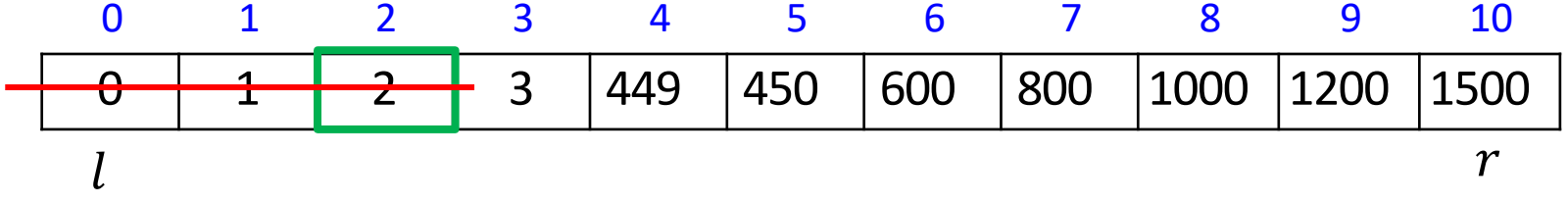
- If keys are close to uniformly distributed, where would key $k = 100$ be?
  - $k = 100$ is ¾ of the way between $A[l] = 40$ and $A[r] = 120$
  - so look at index which is ¾ of the way between l and r

- Interpolation search
  - look at index $l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]}(r-l) \right\rfloor$

fractional distance

Interpolation Search Example

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor$$



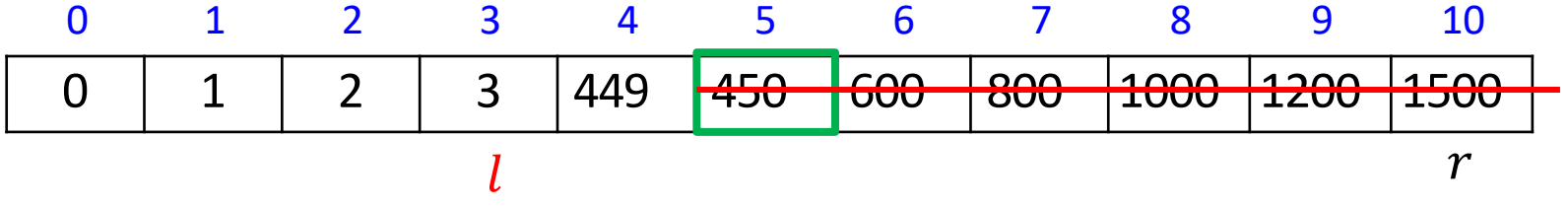
- *Search*(449), iteration 1

$l = 0,\ r = n - 1 = 10,$

$$m = 0 + \left\lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \right\rfloor = 2$$

Interpolation Search Example

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor$$



- *Search*(449), iteration 2

$l = 3,\ r = 10,$

$$m = 3 + \left\lfloor \frac{449 - 3}{1500 - 3} (10 - 3) \right\rfloor = 5$$

- Deleted 6 out of 8 elements, better than possible with binary search

Interpolation Search Example

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor$$

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

l r

key found

- *Search*(449), iteration 3

$l = 3,\ r = 4,$ $\quad m = 3 + \left\lfloor \frac{449 - 3}{499 - 3} (4 - 3) \right\rfloor = 4$

Interpolation Search

0	1	2	3	4	5	6	7	8	9	10
0	11	23	30	44	51	64	73	85	92	105

- Works well if keys are uniformly distributed
  - can show: the array in which we recurse into has expected size $\sqrt{n}$
  - recurrence relation is $T^{avg}(n) = T^{avg}(\sqrt{n}) + \Theta(1)$
  - this resolves to $T^{avg}(n) \in \Theta(\log \log n)$
- Worst case performance $\Theta(n)$
  - search(90)

0	1	2	3	4	5	6	7	8	9	10
0	90	91	92	93	94	95	96	97	98	99

- Clever trick
  - use interpolation search for $\log n$ steps
  - if key is still not found, switch to binary search
  - guarantees $O(\log n)$ worst case, but could be $\Theta(\log \log n)$

Interpolation Search

- Code similar to binary search, but compare at interpolated index
- Need extra test to avoid division by zero due to $A[l] = A[r]$

Interpolation-search(A, n, k)
A: Sorted array of size n, k: key
$l \leftarrow 0$
$r \leftarrow n - 1$
while $(l \leq r)$
if $(k < A[l]$ or $k > A[r])$ **return** "not found"
if $(k = A[r])$ **return** "found at $A[r]$"
$m \leftarrow l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor$
if $(A[m] == k)$ **return** "found at $A[m]$"
else if $(A[m] < k)$
$l \leftarrow m + 1$
elsif $r \leftarrow m - 1$
// we always return somewhere within while loop

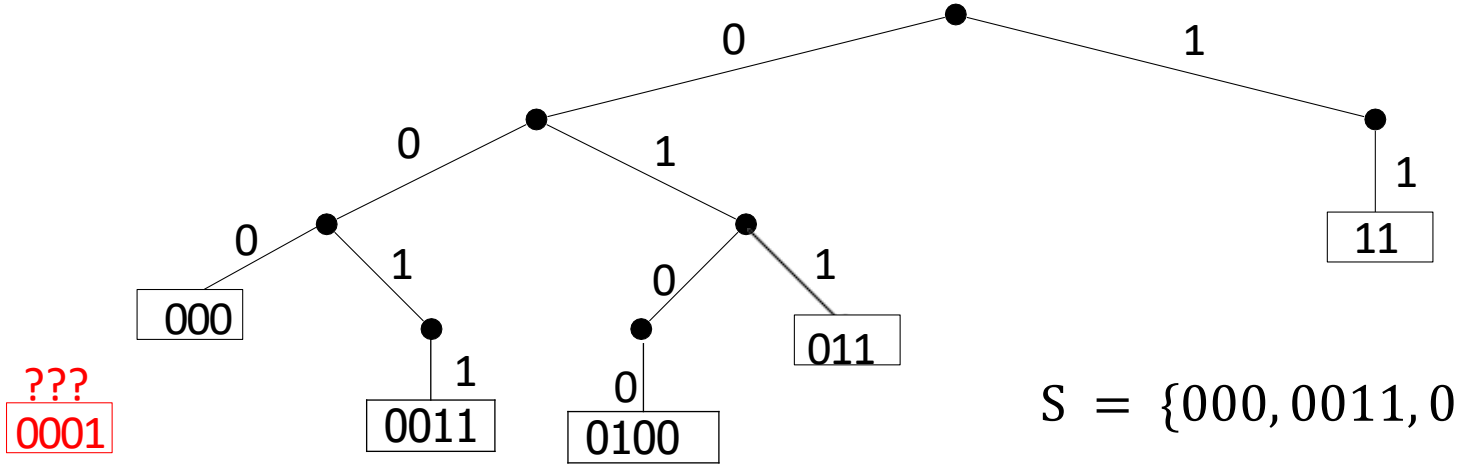
Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard
    - Variations of Tries
    - Compressed Tries
    - Multiway Tries

Tries: Introduction

- **Trie** (also known as **radix tree**): a dictionary for bit strings (words)
  - should know: string, word, alphabet, prefix, suffix, comparing words
- Comparison with AVL trees
  - let the number of strings in dictionary be n
  - let $|x|$ be the length of a string x
  - in tries, insert, find, delete strings is $O(|x|)$ time
    - independent of n
  - AVL tree requires $O(|x|\log(n))$ time
    - $O(\log(n))$ to search, $O(|x|)$ operations at each node
- Efficient for prefix search
  - find all words in the dictionary that start with “abl”
- Applications
  - auto-completion
    - smart phones
    - commands for operating systems
  - spell checking
  - DNA sequencing

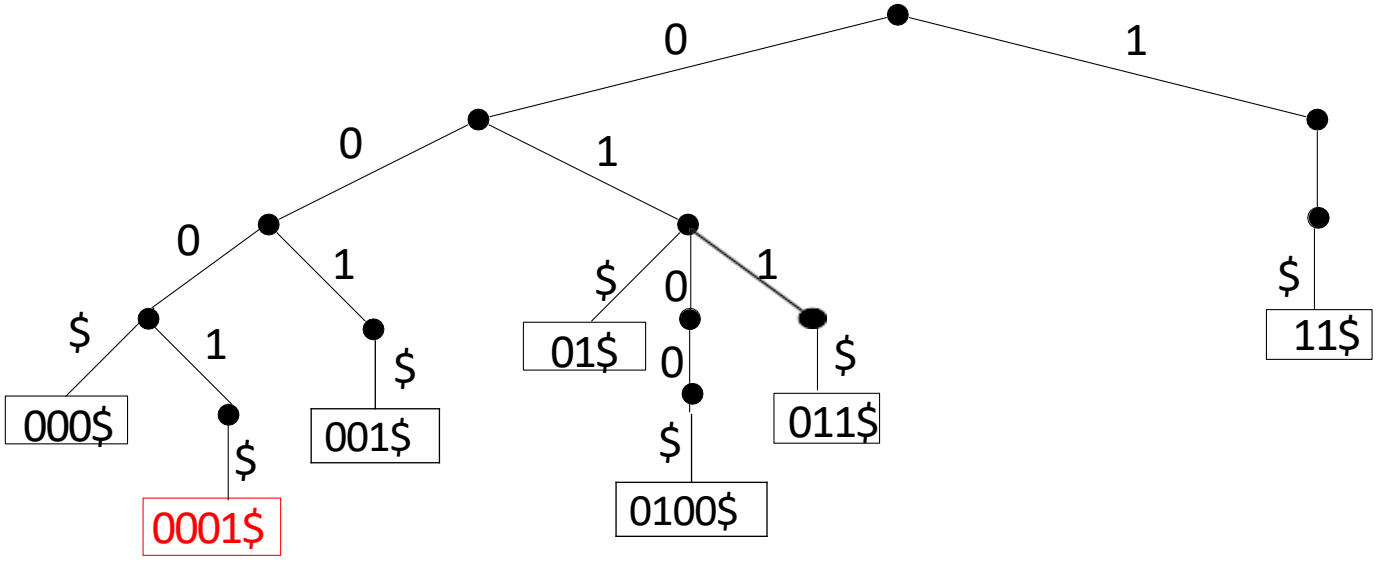
Tries: definition



$$S = \{000, 0011, 0100, 011, 11\}$$

- Trie (Radix Tree): comes from word retrieval, but pronounced "try"
  - tree based on bitwise comparisons: edges labeled with corresponding bit
  - keys are stored only at leaves
  - similar to radix sort: use individual bits, not the whole key
  - string stored at a leaf v is "read" from path from root to v
- So far, works only for prefix-free S
  - no pair of binary strings where one is prefix of another
    - prefix of a string $S[0 \ldots n-1]$ is $S[0 \ldots i]$ for some $0 \leq i < n-1$
  - always satisfied if S has strings of the same length

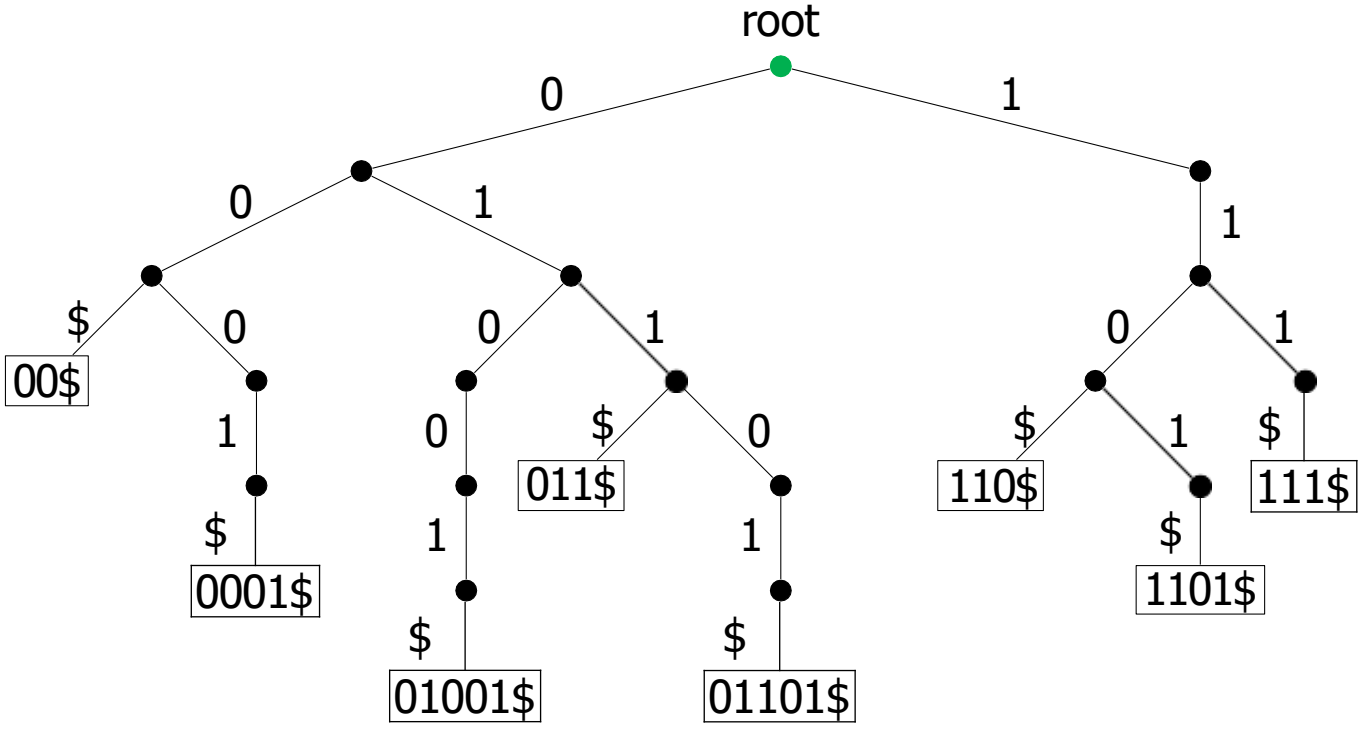
Tries: Relaxing Prefix-Free Requirement



- Add a special character '$' to signal string end
- Each node can have up to three children
- Trie structure is independent of the key insertion order
- Space requirements
  - for each word x, have $|x|$ nodes
  - total at most $\sum_{words\ x} |x|$
  - but usually need much less space as words share prefixes
    - shared prefix means shared trie node

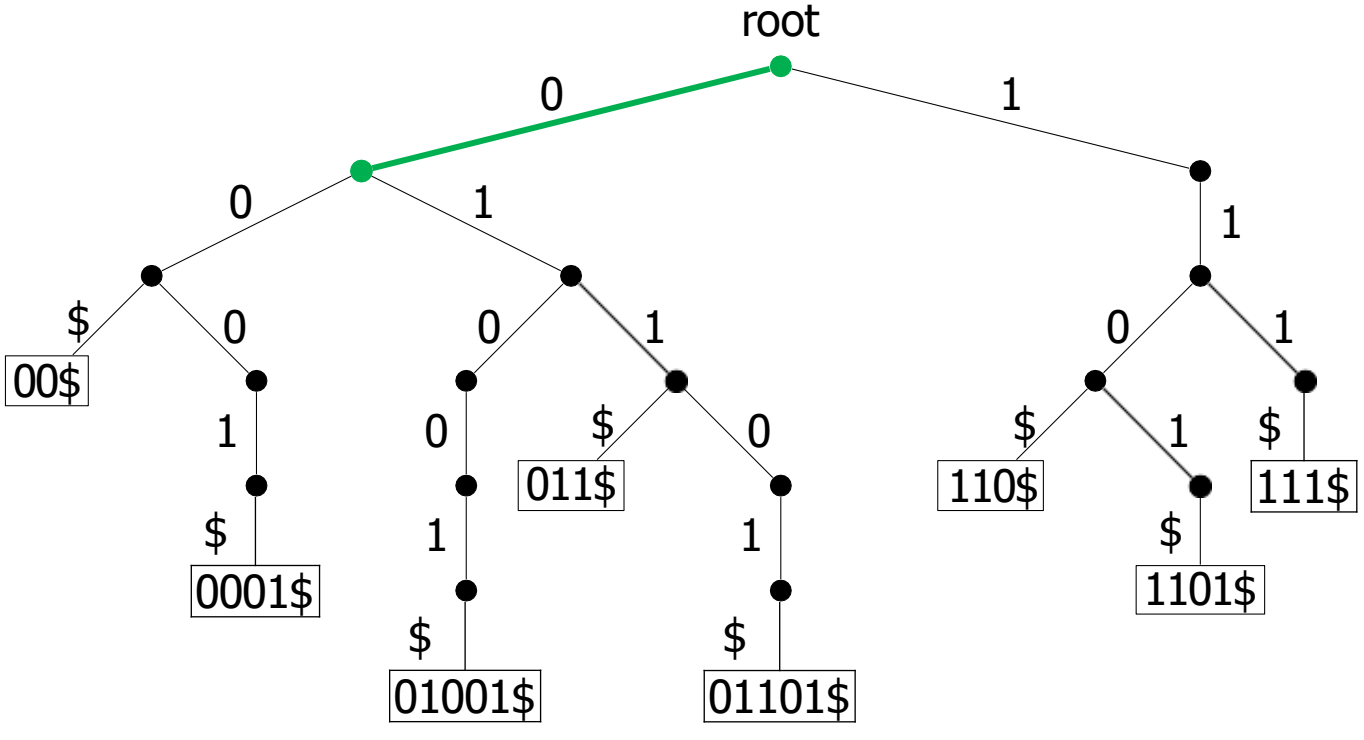
Tries: Search Example

Example: Search(011$)



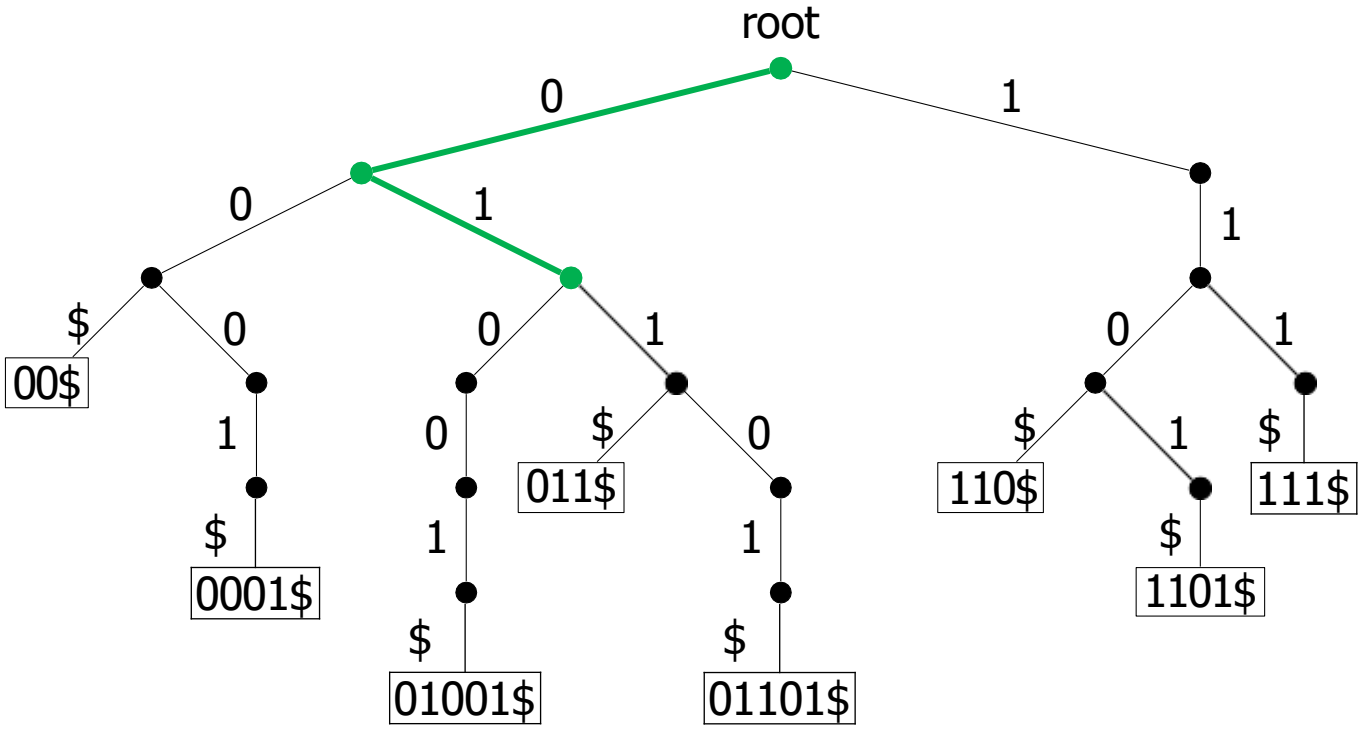
Tries: Search Example

Example: Search(011$)



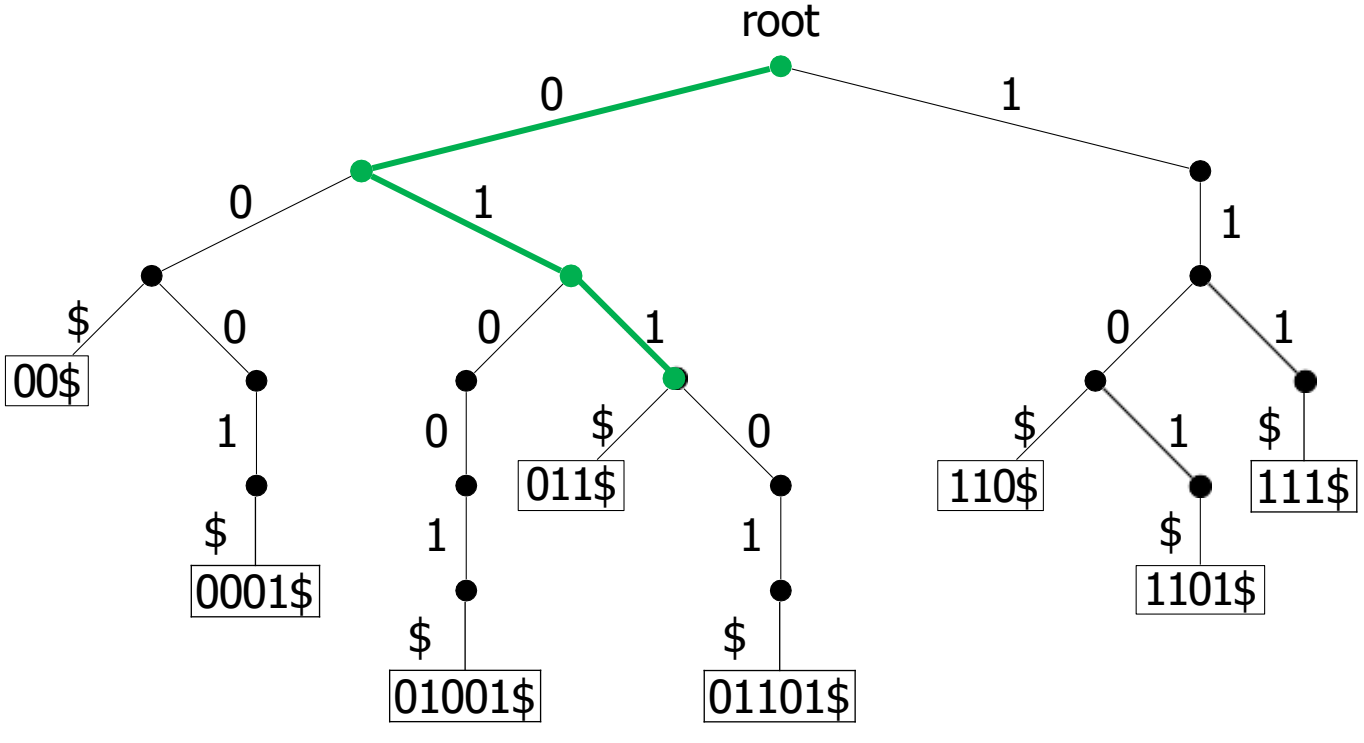
Tries: Search Example

Example: Search(011$)



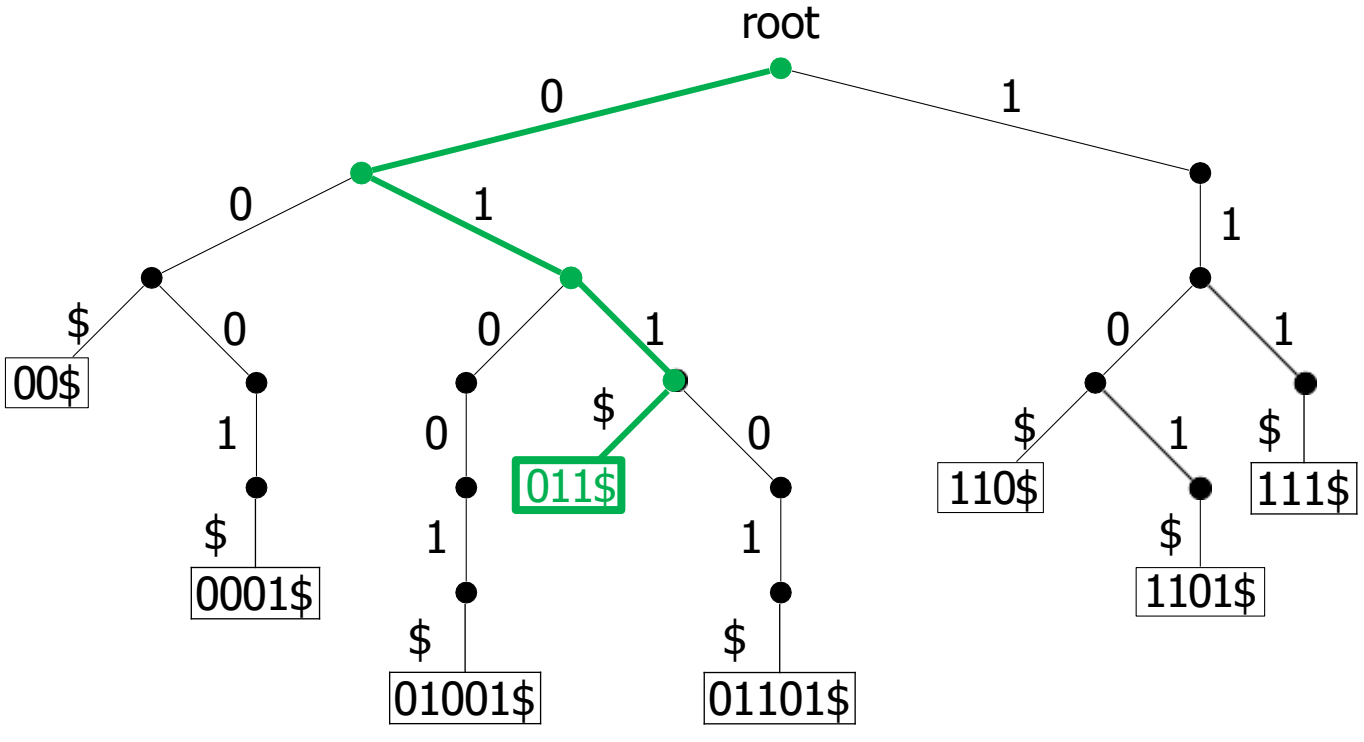
Tries: Search Example

Example: Search(011$)



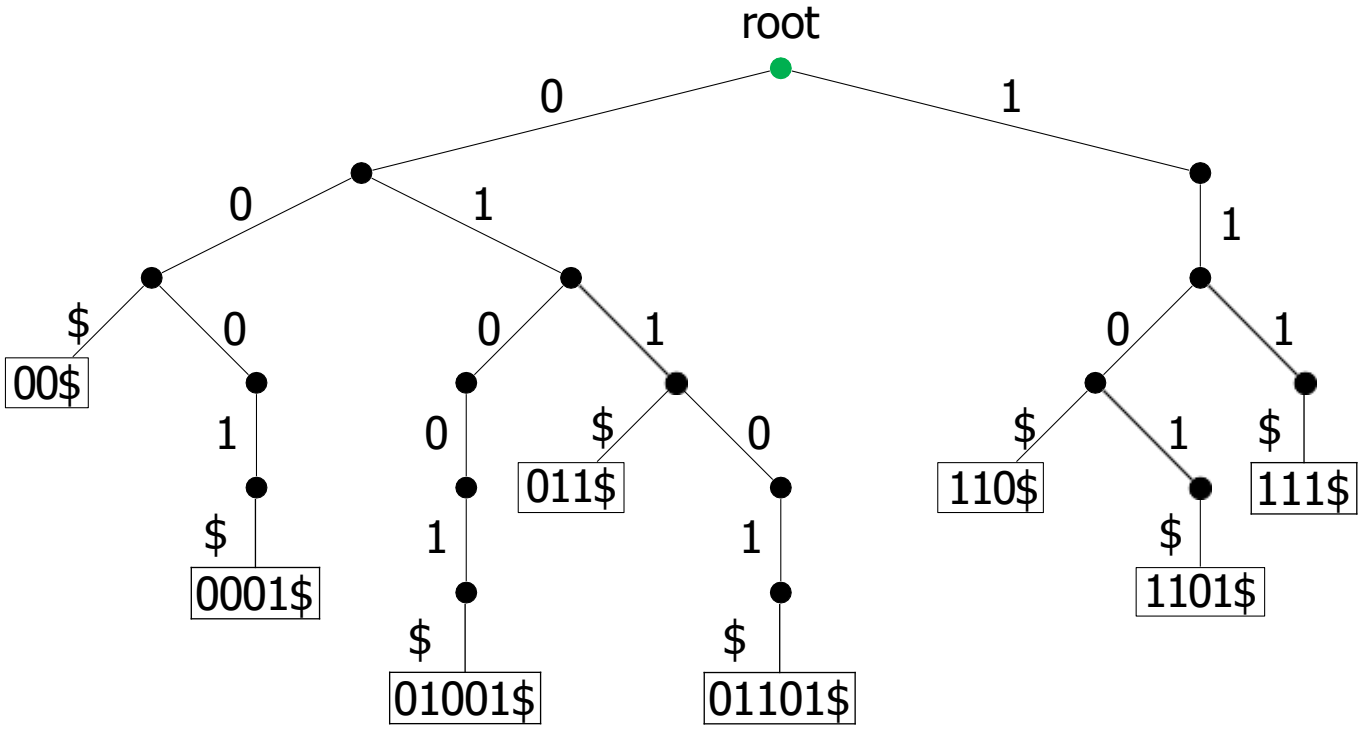
Tries: Search Example

Example: Search(011$) successful



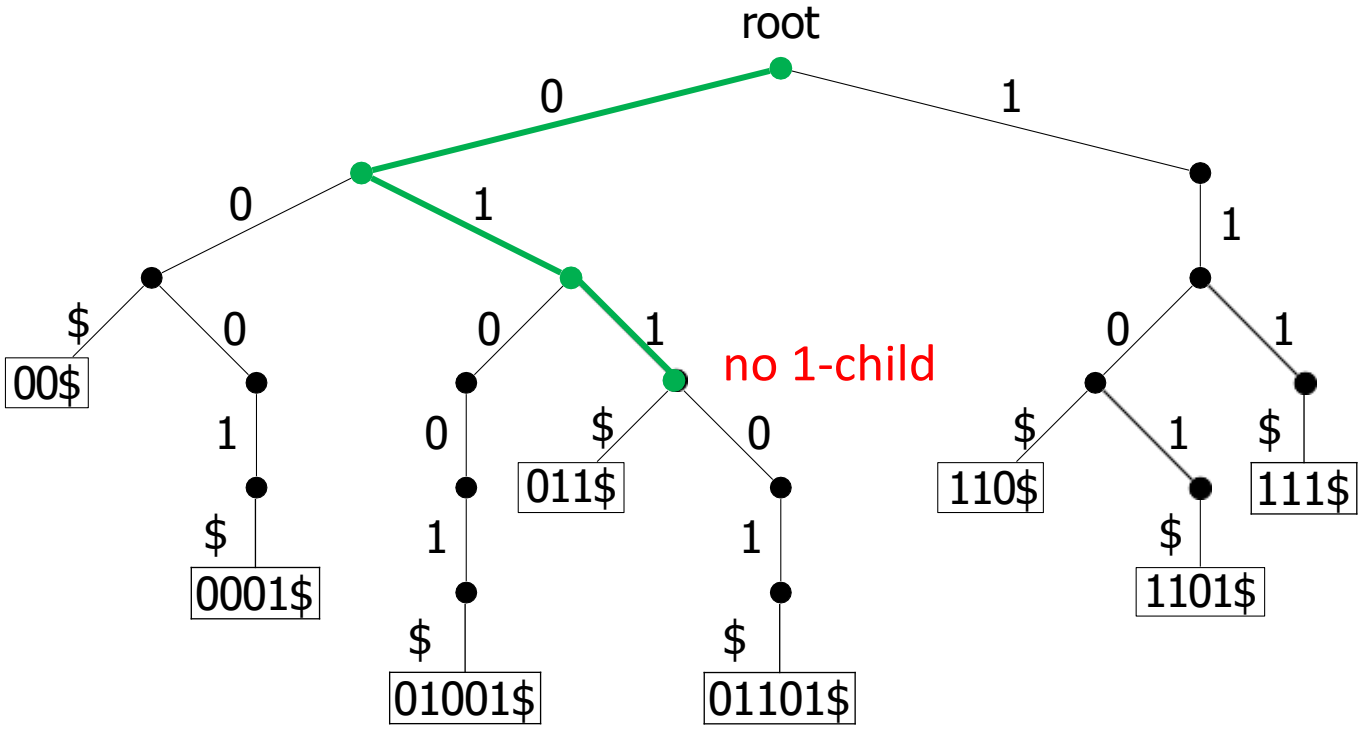
Tries: Search Example

Example: Search(0111$)



Tries: Search Example

Example: Search(0111$) unsuccessful



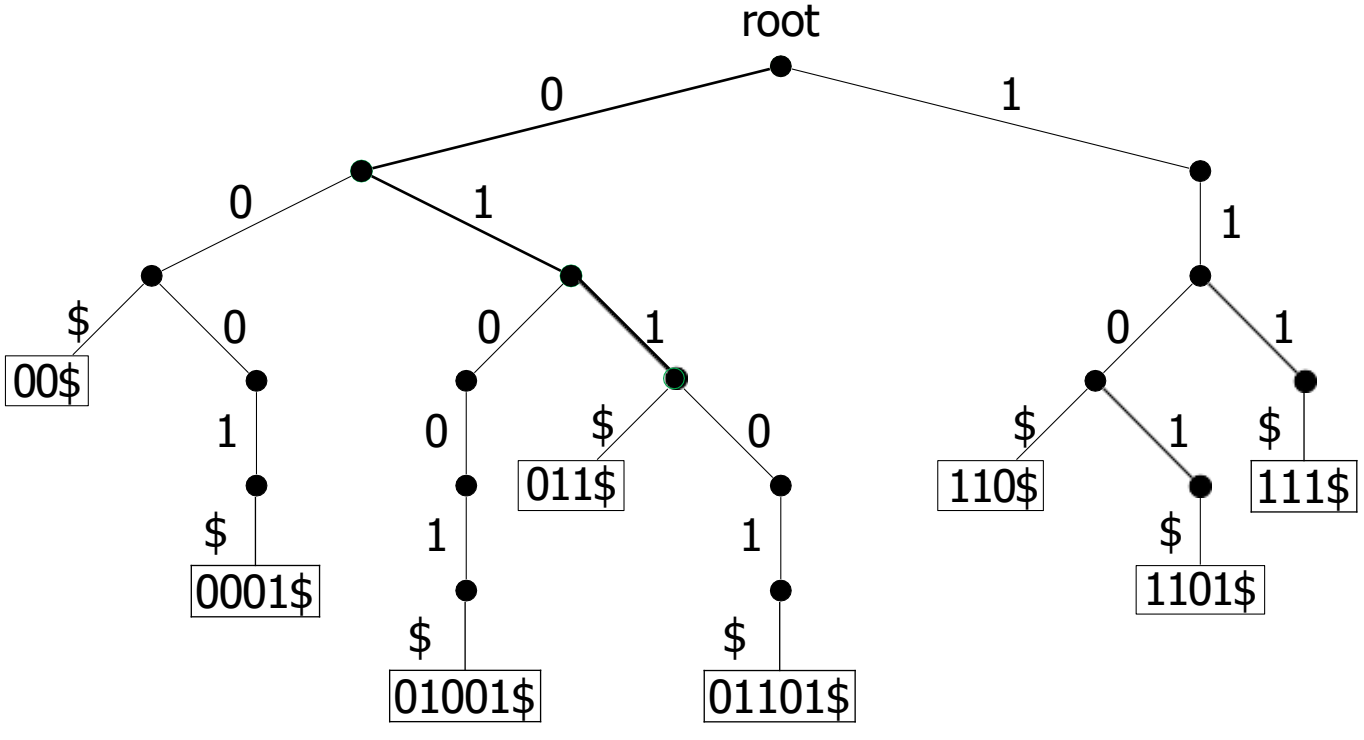
Tries: Search

- 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$(v \leftarrow root, d \leftarrow 0, x)$
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: Insert Example

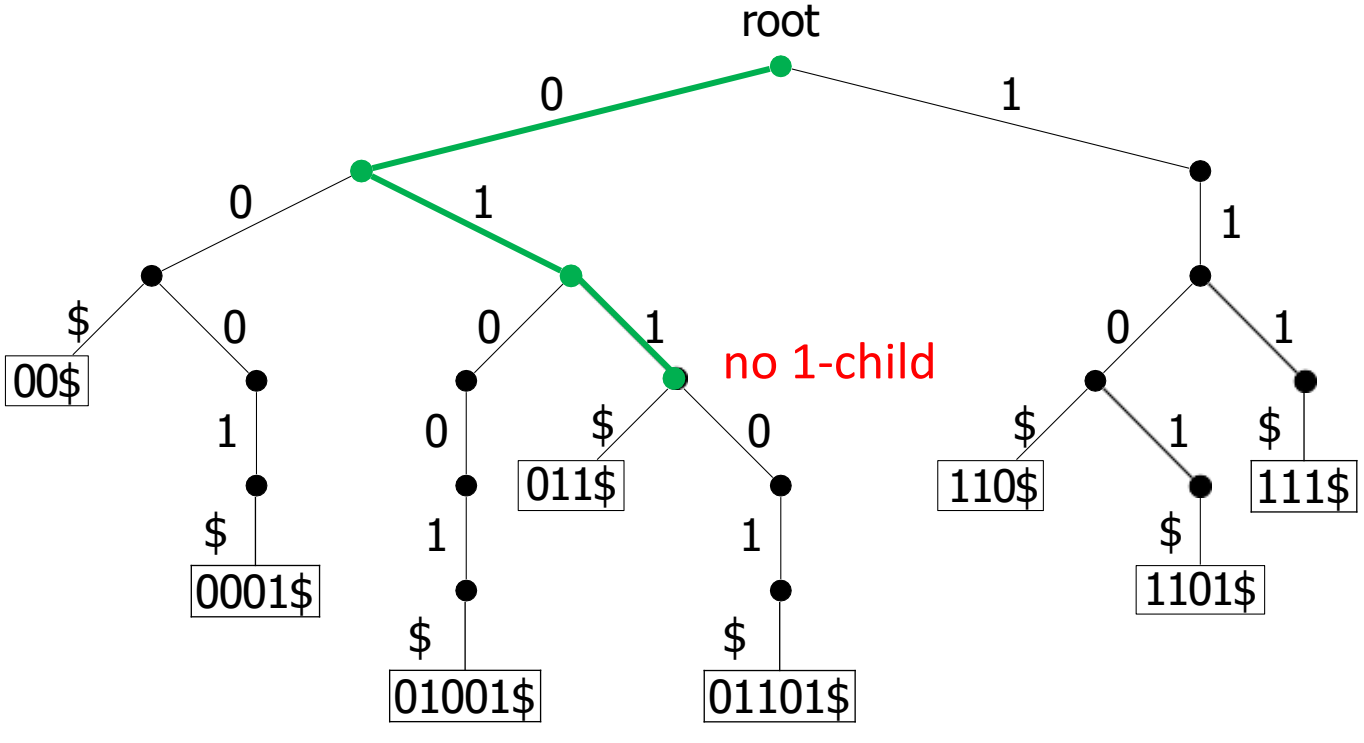
Example: Insert(0111$)



Tries: Insert Example

Example: Insert(0111$)

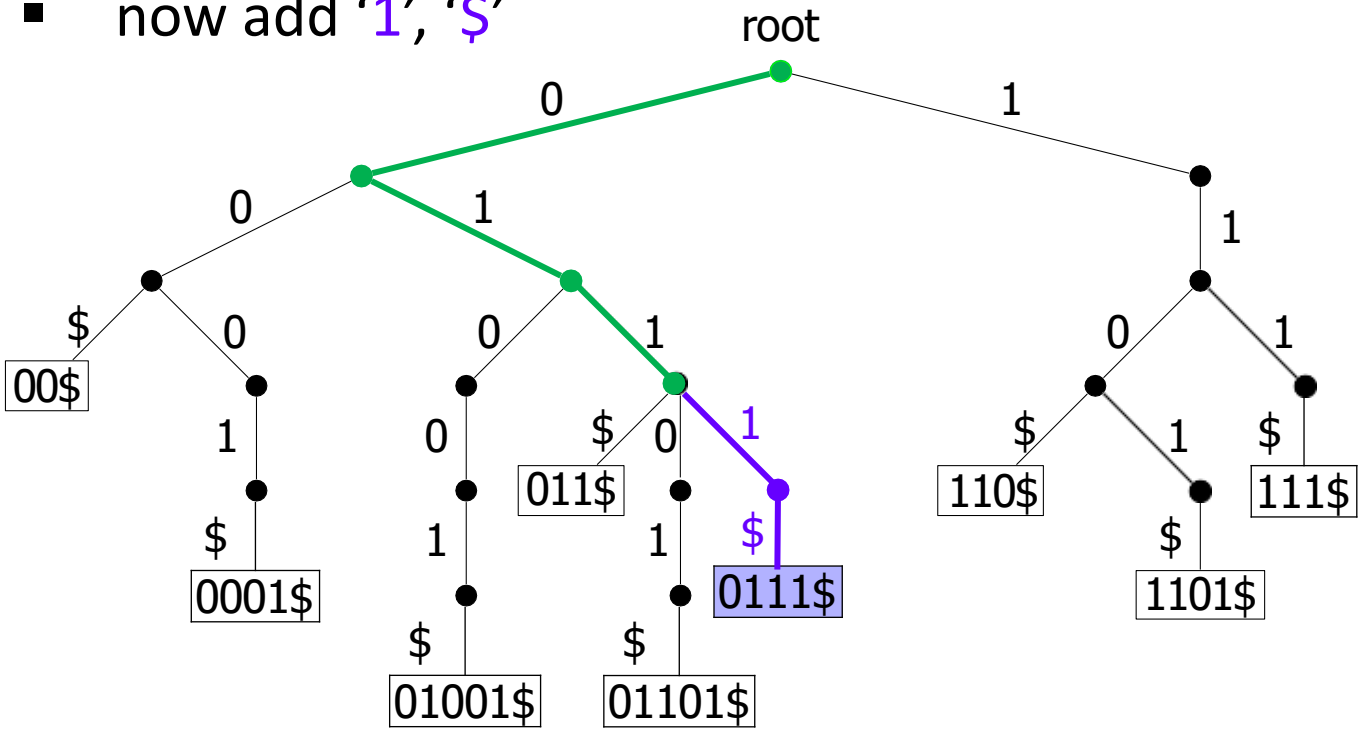
- first search(0111$)



Tries: Insert Example

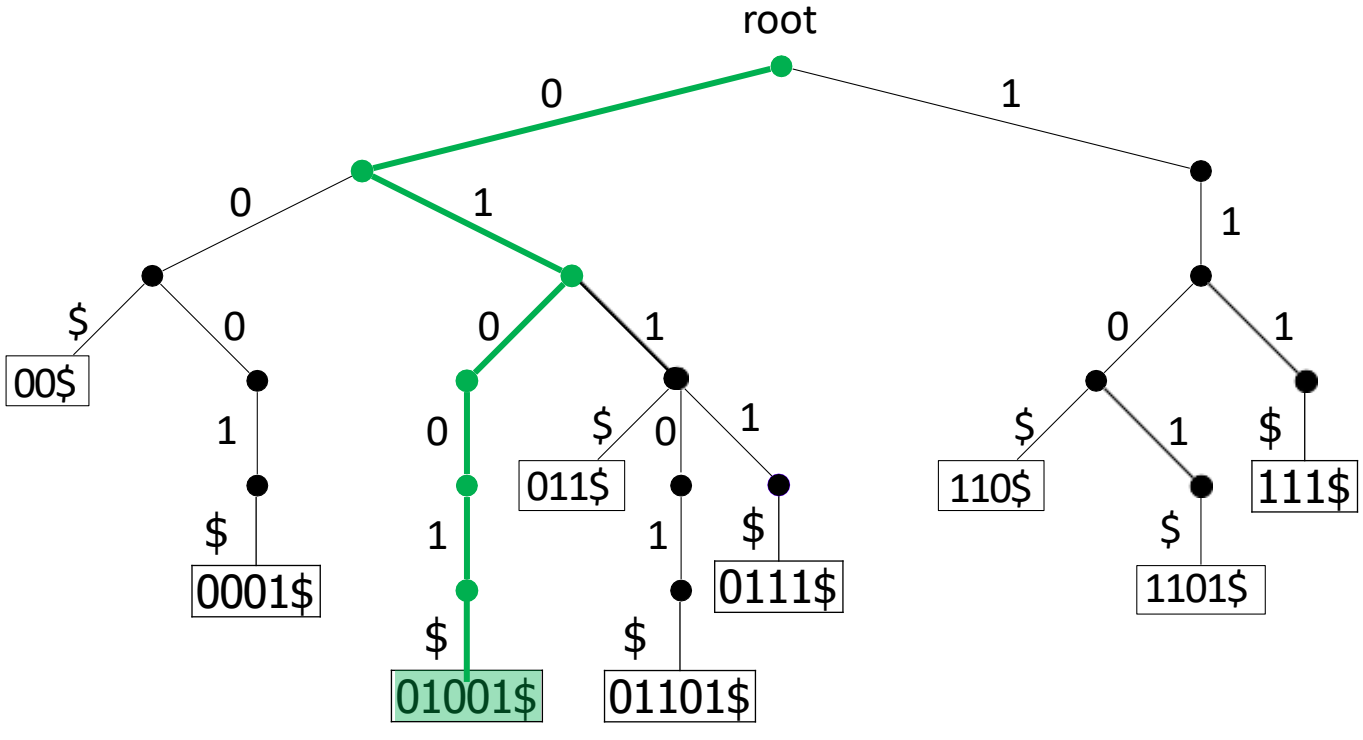
Example: Insert(0111$)

- first search(0111$)
- now add '1', '$'



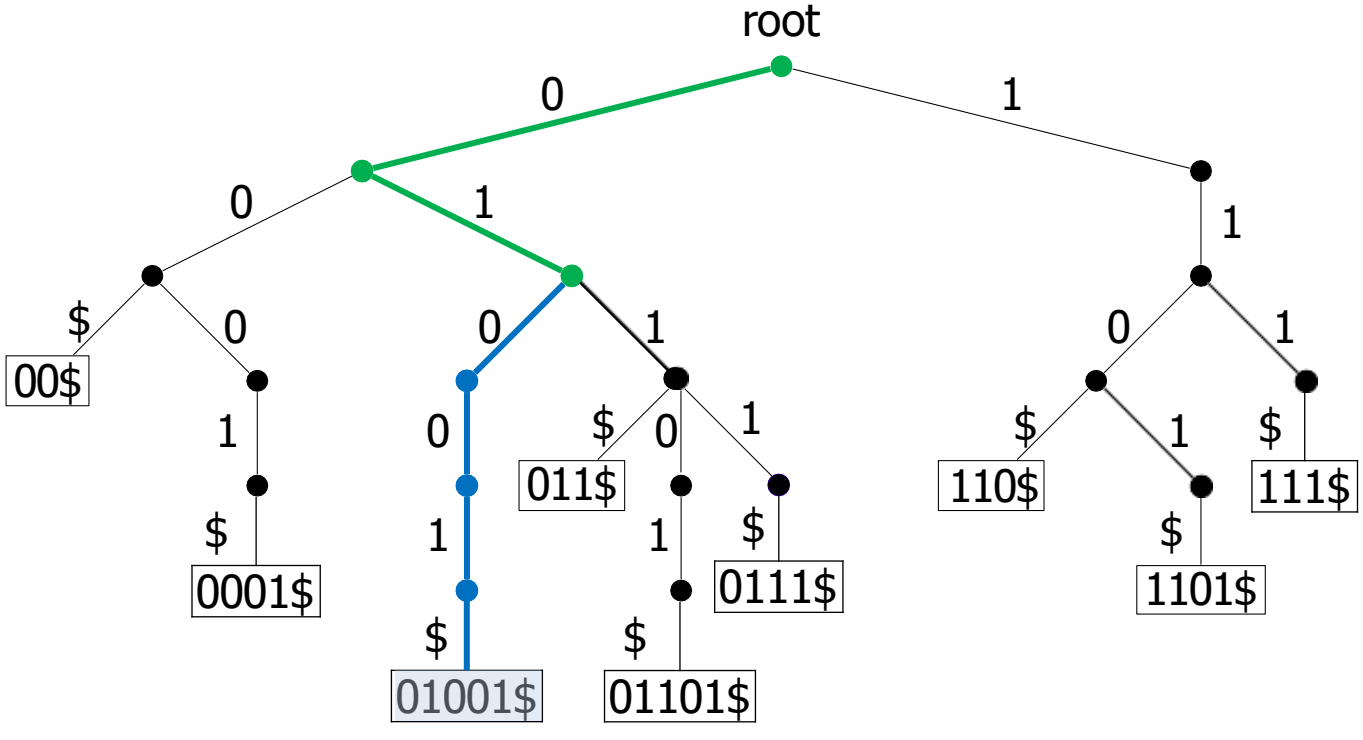
Tries: Delete Example

Example: Delete(01001$)



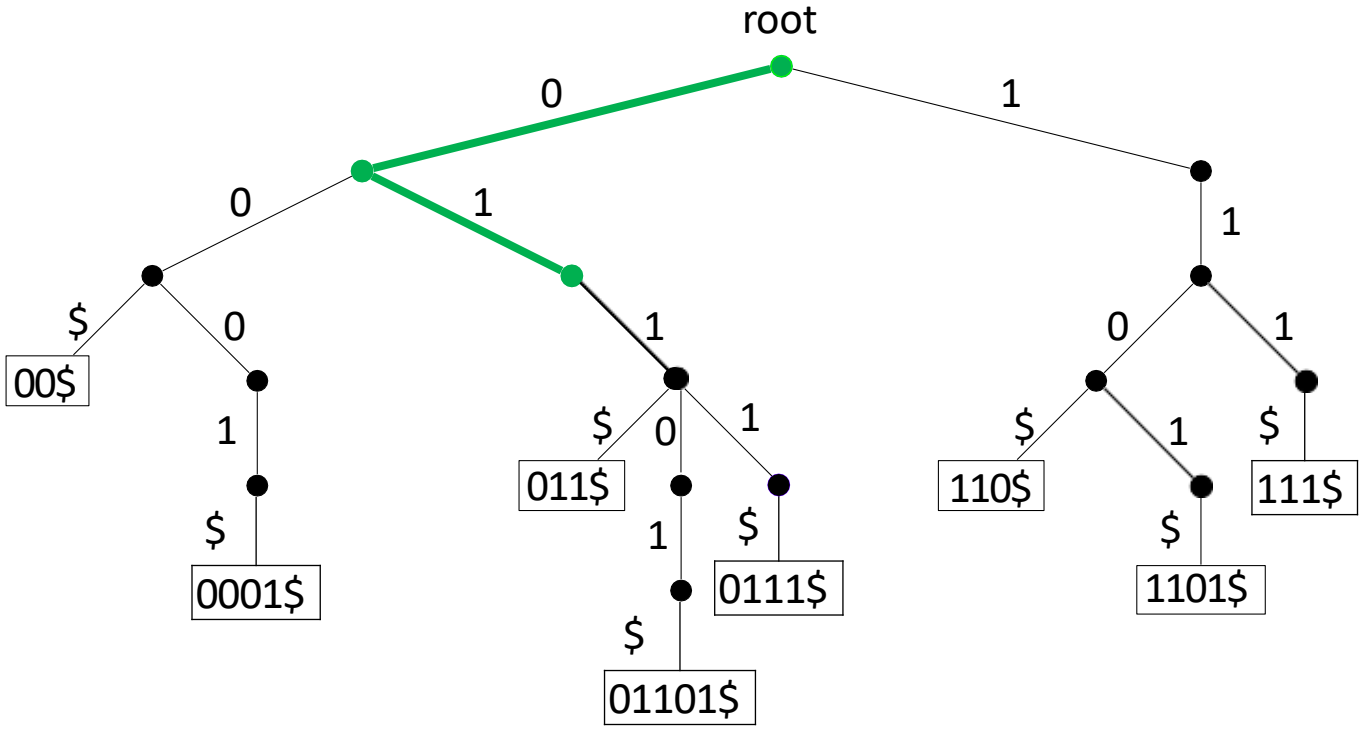
Tries: Delete Example

Example: Delete(01001$)



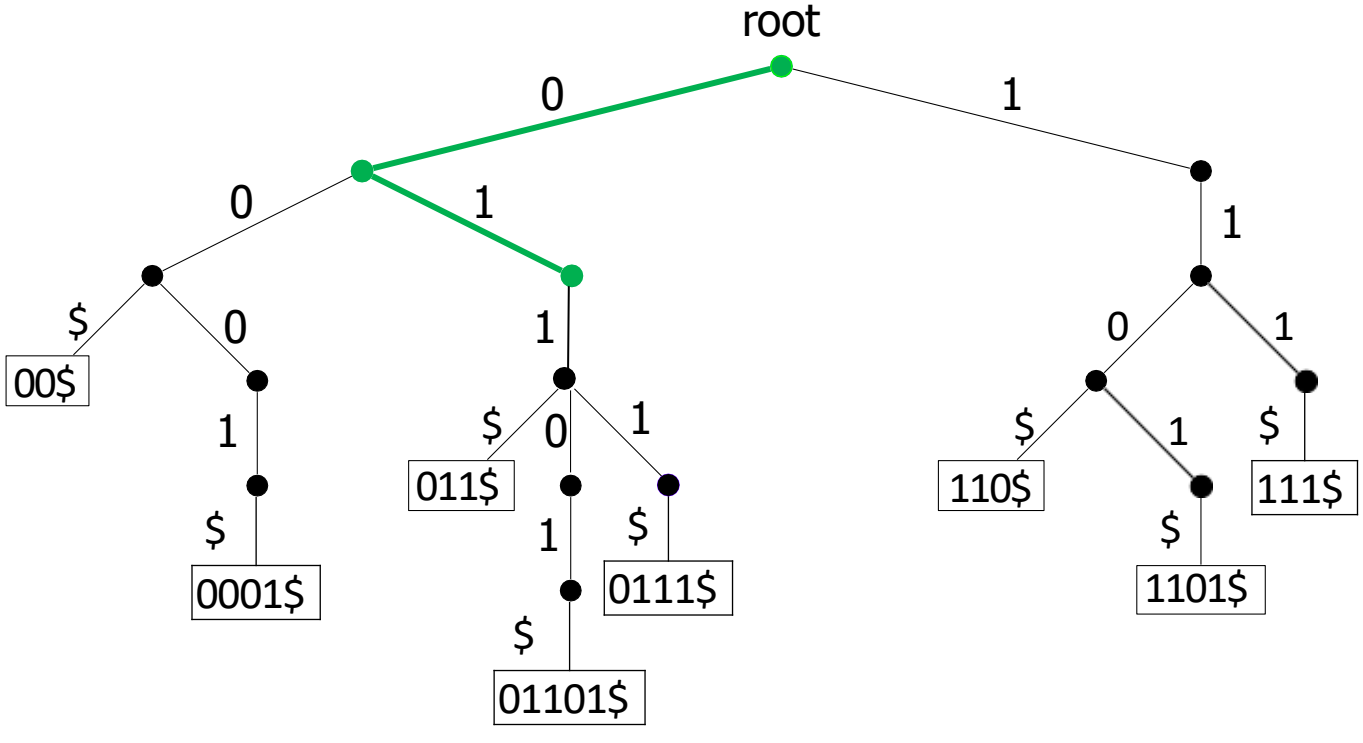
Tries: Delete Example

Example: Delete(01001$)



Tries: Delete Example

Example: Delete(01001$)



Tries: Insert & Delete

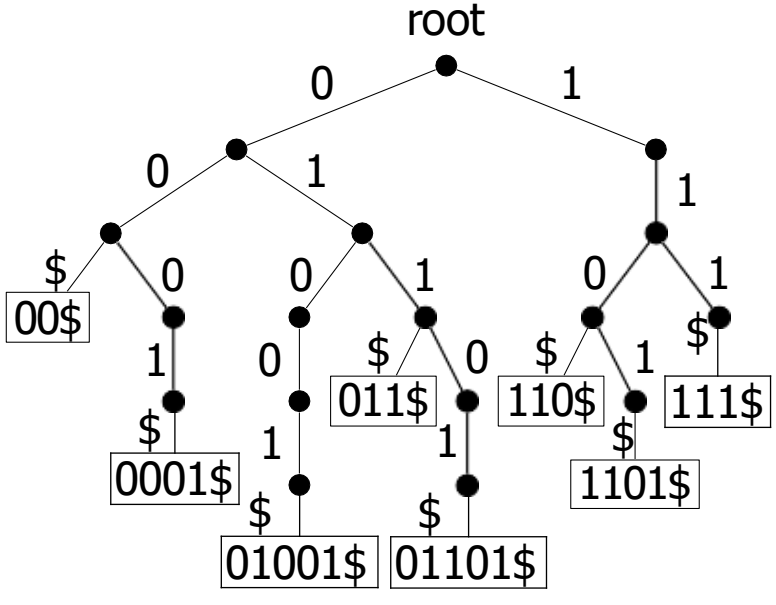
- Insert(x)
  - Search for x, this should be unsuccessful
  - Suppose finish search at node v that is missing a suitable child
    - x has extra bits left
  - Expand the trie from node v by adding necessary nodes corresponding to extra bits of x
- Delete(x)
  - search for x
  - let v be the leaf where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

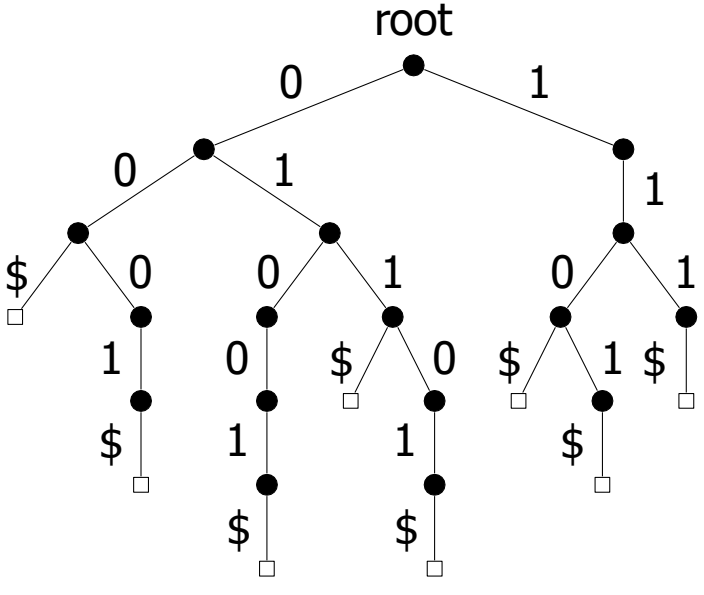
Outline

- Lower bound for search
- Interpolation Search
- Tries
  - Standard
  - Variations of Tries
  - Compressed Tries
  - Multiway Tries

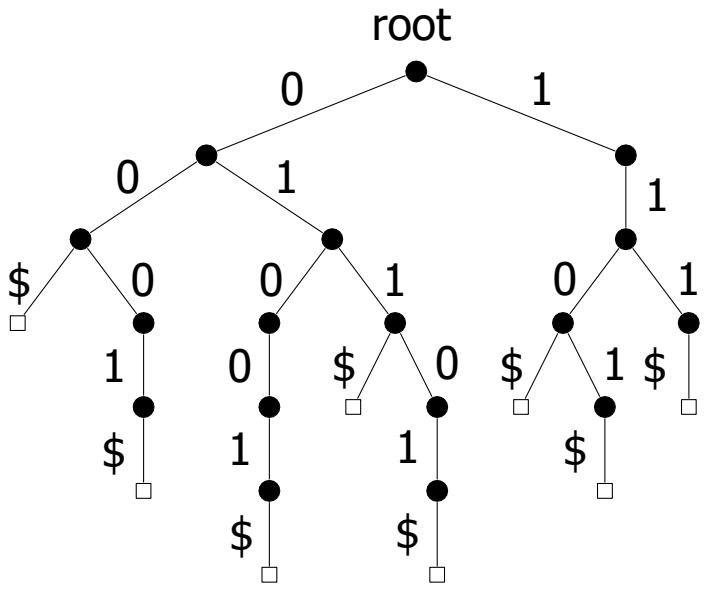
Variation 1 of Tries: No leaf labels

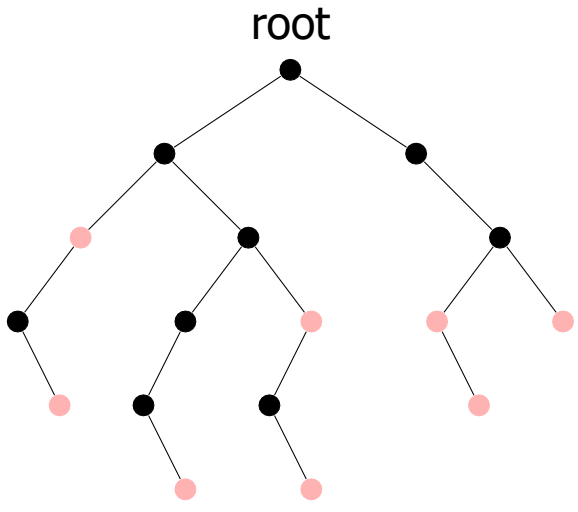
- Do not store actual keys at the leaves
- The key is stored implicitly through characters along the path to the leaf
- Halves the amount of space





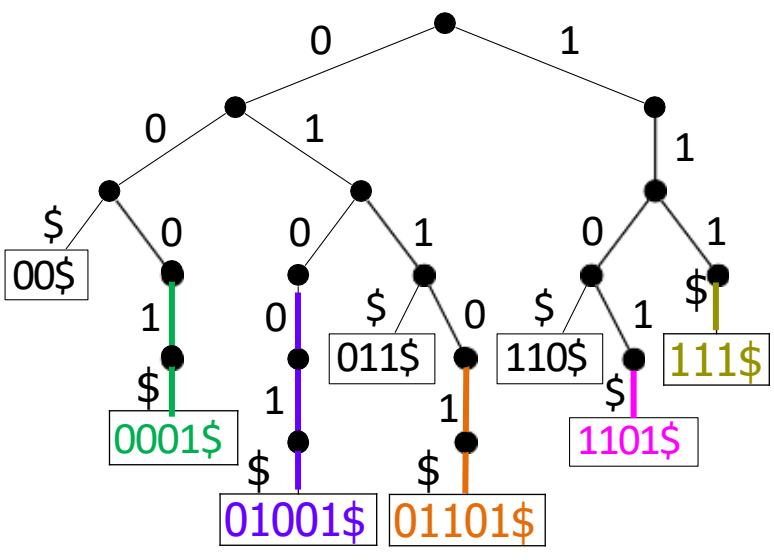
Variation 2 of Tries: Allow Proper Prefixes

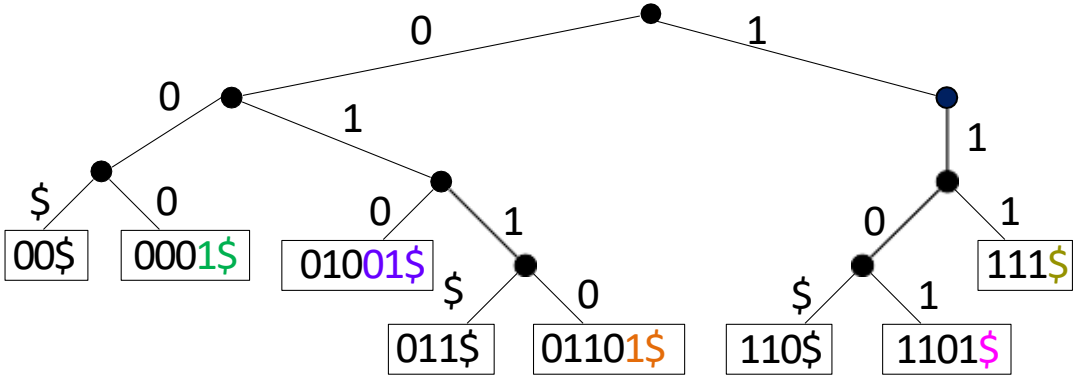




- Allow prefixes to be in dictionary
  - internal nodes may now also represent keys
    - use a *flag* to indicate such nodes
  - remove $-children, replace by flags
  - now trie is a binary tree
    - expresses 0-child and 1-child implicitly via left and right child
  - more space-efficient

Variations 3 of Tries: Remove Chains to Leafs (Labels)



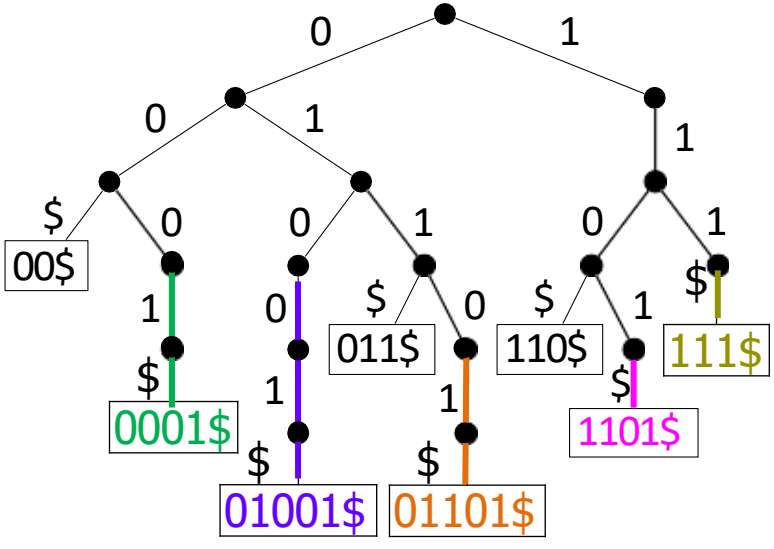


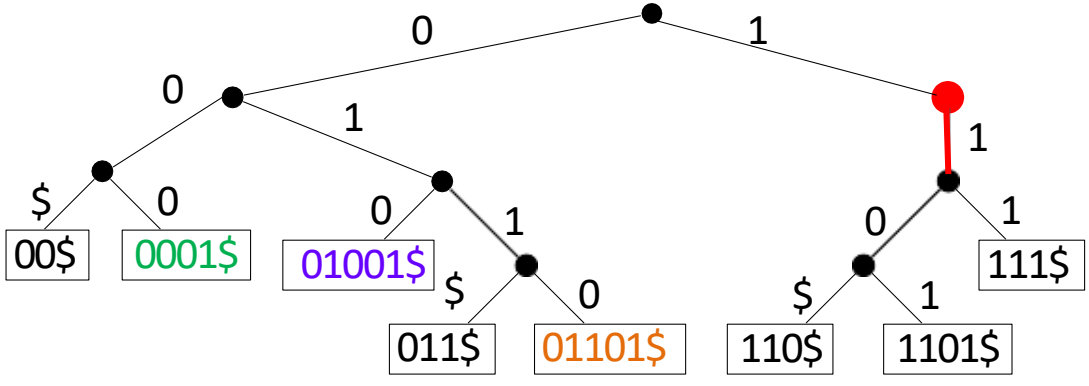
- **Pruned** trie: stop adding nodes to trie as soon as the key is unique
  - node has a child only if it has at least two descendants
  - saves space if there are only few bitstrings that are long
  - can even store really long bitstrings more efficiently (real numbers)
  - this variation *cannot* be combined with the previous one
    - why?
  - more efficient version of tries, but operations get complicated

Outline

- Lower bound for search
- Interpolation Search
- Tries
  - Standard Tries
  - Variations of Tries
  - Compressed Tries
  - Multiway Tries

Variation 4: Compressed Tries (Patricia Tries)

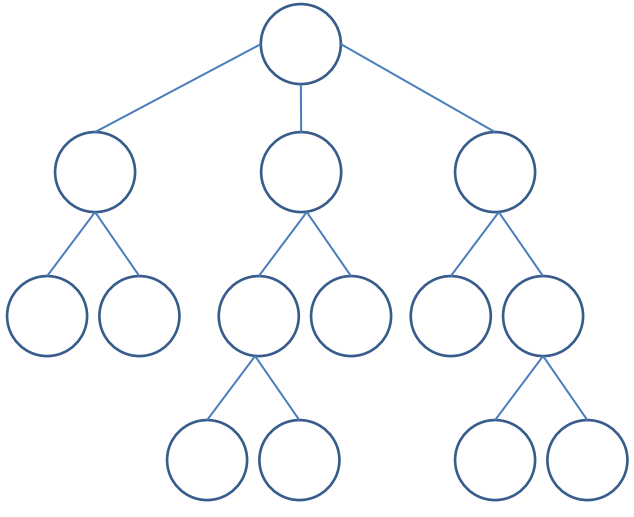




- Removing chains to labels helps, but can still have internal nodes with one child
- Such ‘chains’ in a trie waste space and reduce search/insert/delete efficiency
- If we insure each internal node has at least 2 children, no space wasted
  - n leaf nodes = n keys stored
  - at most $n - 1$ internal nodes
  - at most $2n - 1$ total nodes

Compressed Tries

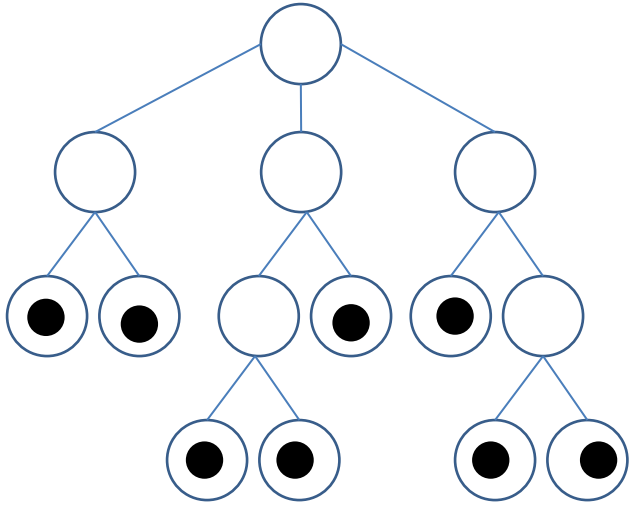
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof
  - put a stone on each leaf
  - there are m stones

Compressed Tries

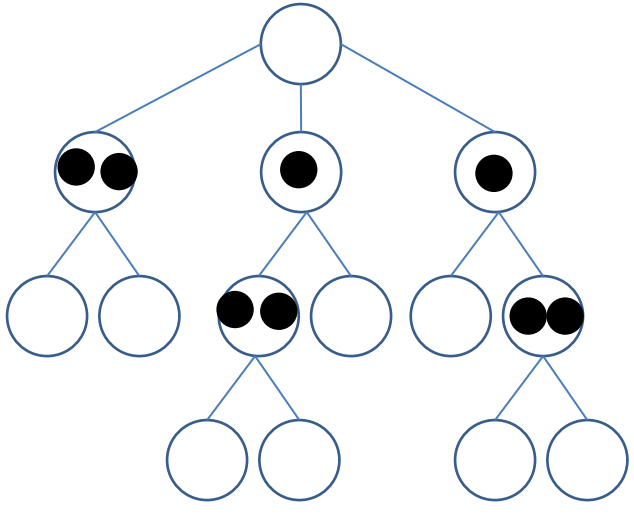
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof:
  - put a stone on each leaf
  - there are m stones
  - all leaves pass a stone to the parent

Compressed Tries

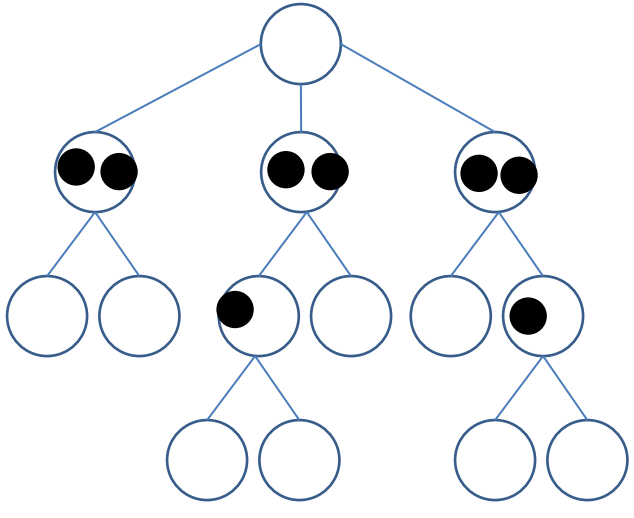
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof:
  - put a stone on each leaf
  - there are m stones
  - all leaves pass a stone to the parent
  - all internal nodes at level $h - 1$ have at least 2 stones, they leave one stone and pass one stone to the parent

Compressed Tries

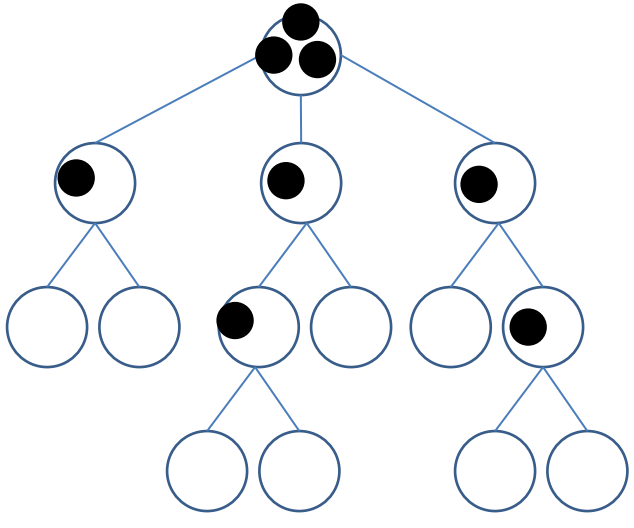
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof:
  - put a stone on each leaf
  - there are m stones
  - all leaves pass a stone to the parent
  - all internal nodes at level $h - 1$ have at least 2 stones, they leave one stone and pass one stone to the parent
  - all internal nodes at level $h - 2$ have at least 2 stones, they leave one stone and pass one stone to the parent

Compressed Tries

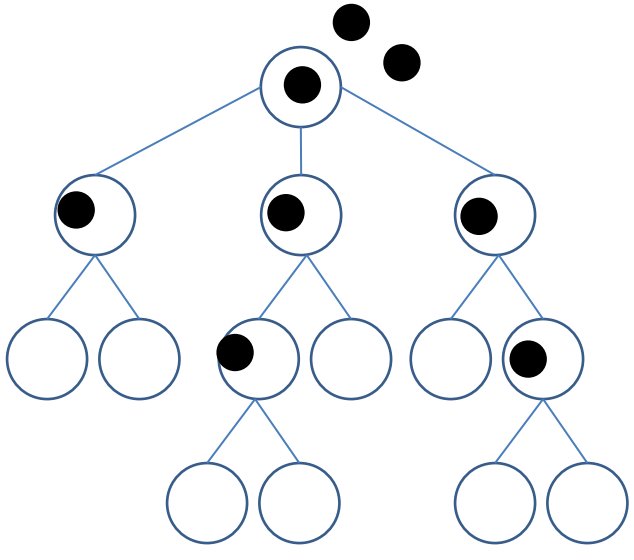
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof:
  - continue until reach the root
  - now each internal node has 1 stone and root has 2 or more stones

Compressed Tries

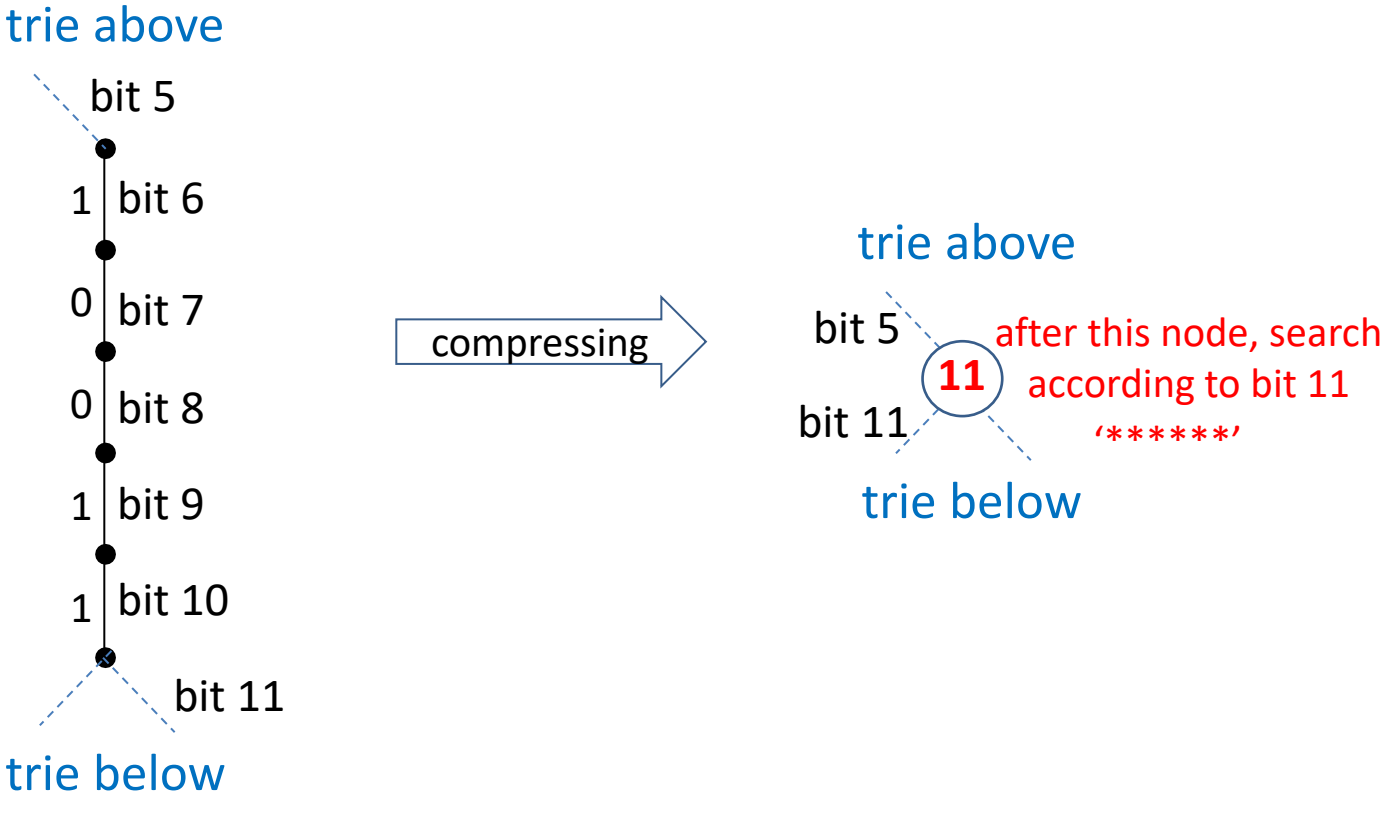
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- Visual proof:
  - continue until reach the root
  - now each internal node has 1 stone and root has 2 or more stones
  - root leaves 1 stone and throws the rest outside the tree
  - now each internal node has 1 stone, and there is one or more stones outside the tree
  - since number of stones is equal to the number of leaves, the number of internal nodes is strictly less than the number of leaves

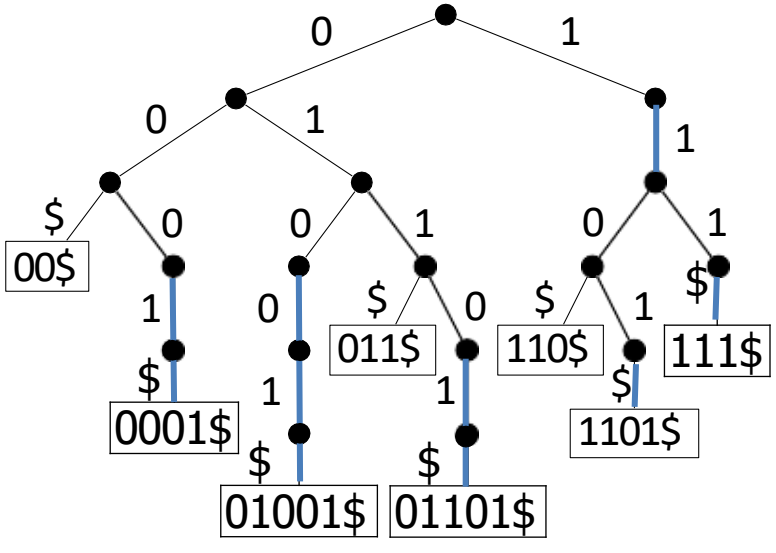
Compressed Tries (Patricia Tries)

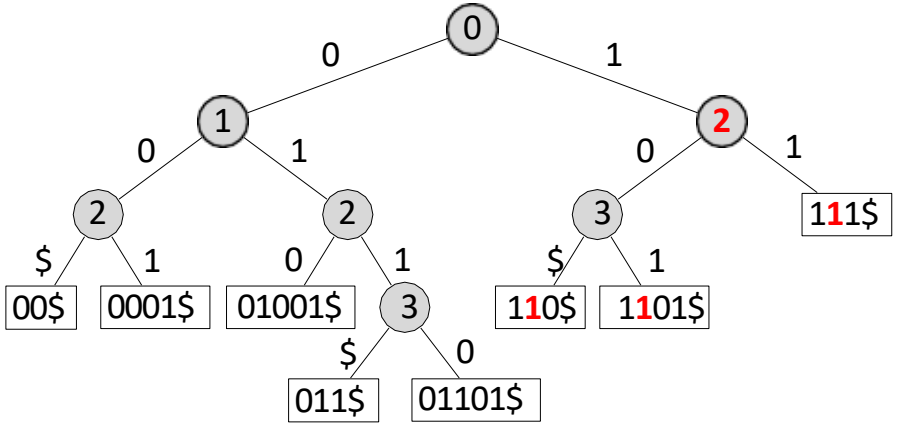
- How to compress



- But now we lost part of the binary string '10011'
- Check the final answer (leaf) if it stores exact match to the search key

Compressed Tries (Patricia Tries)

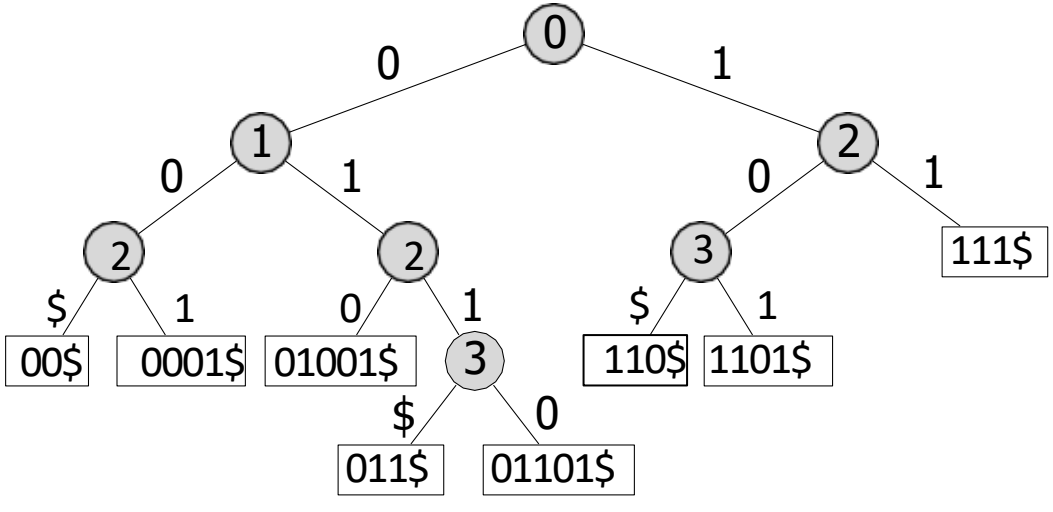




- Morrison (1968): *Patricia-Tries*
- *Practical Algorithm to Retrieve Information Coded in Alphanumeric*
- **Idea:** compress paths of nodes with only one child
- Each node stores an *index* : next bit to be tested during a search
- Compressed trie with n keys has at most $n - 1$ internal (non-leaf) nodes

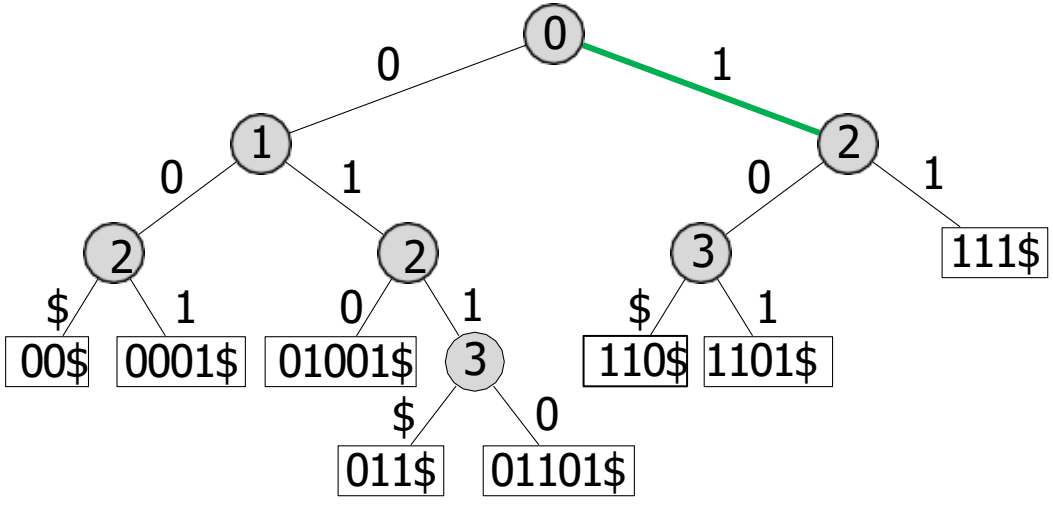
Compressed Tries: Search Example

Example: Search(10$)



Compressed Tries: Search Example

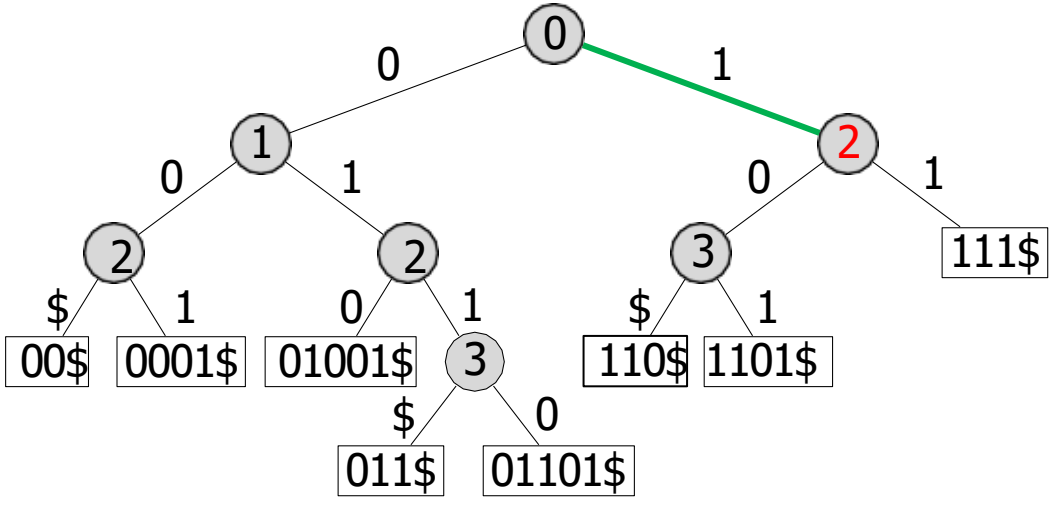
Example: Search(10$)



Compressed Tries: Search Example

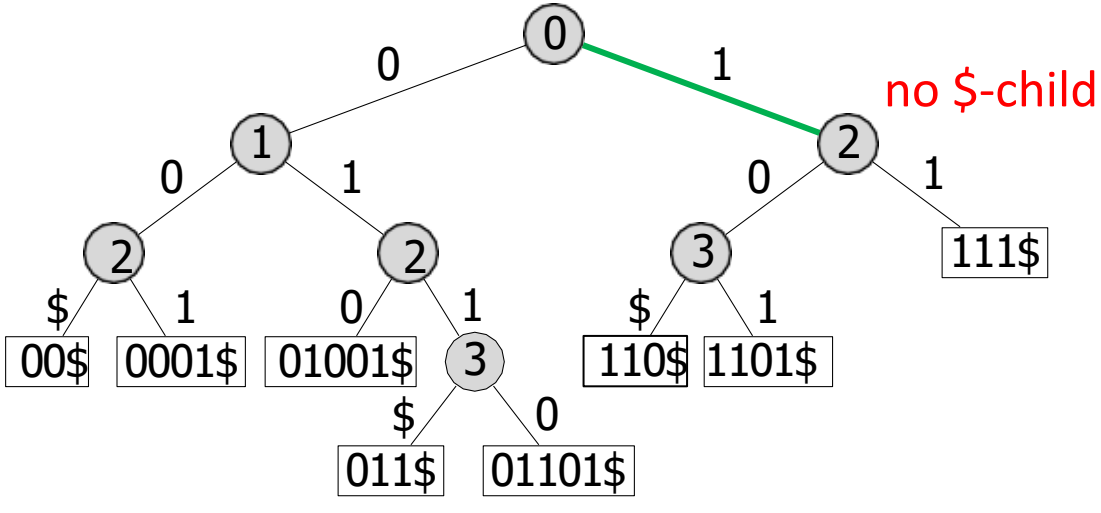
Example: Search(10$)

skip



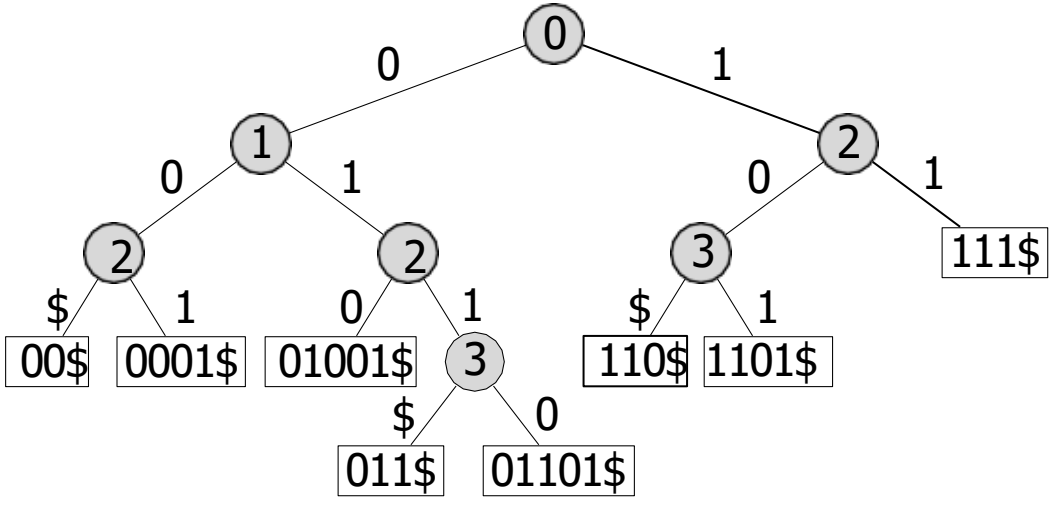
Compressed Tries: Search Example

Example: Search(10$) unsuccessful



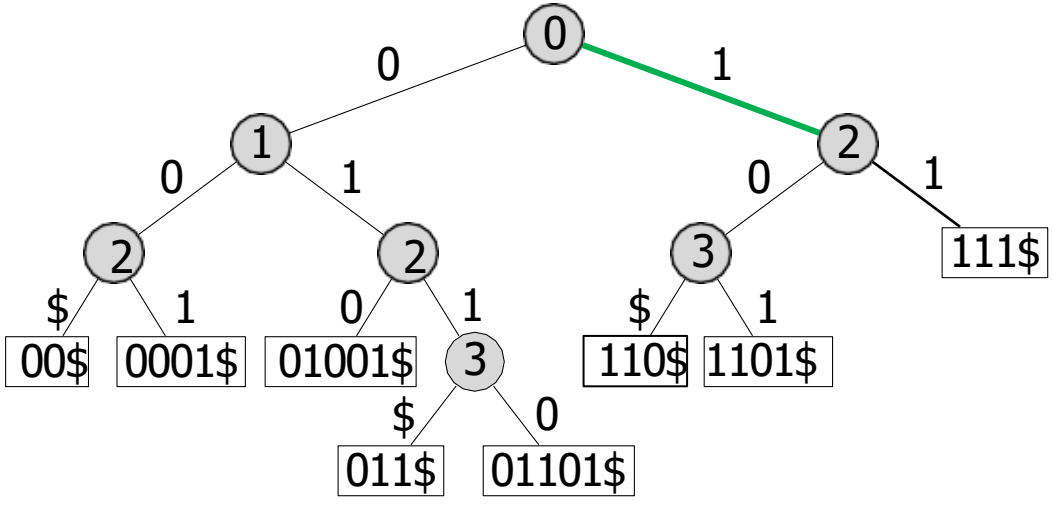
Compressed Tries: Search Example

Example: Search(101$)



Compressed Tries: Search Example

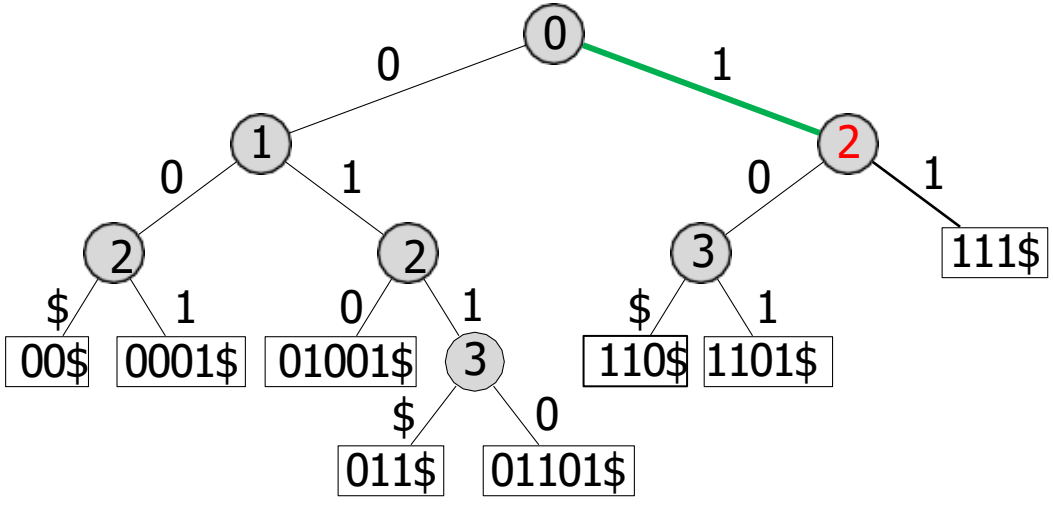
Example: Search(101$)



Compressed Tries: Search Example

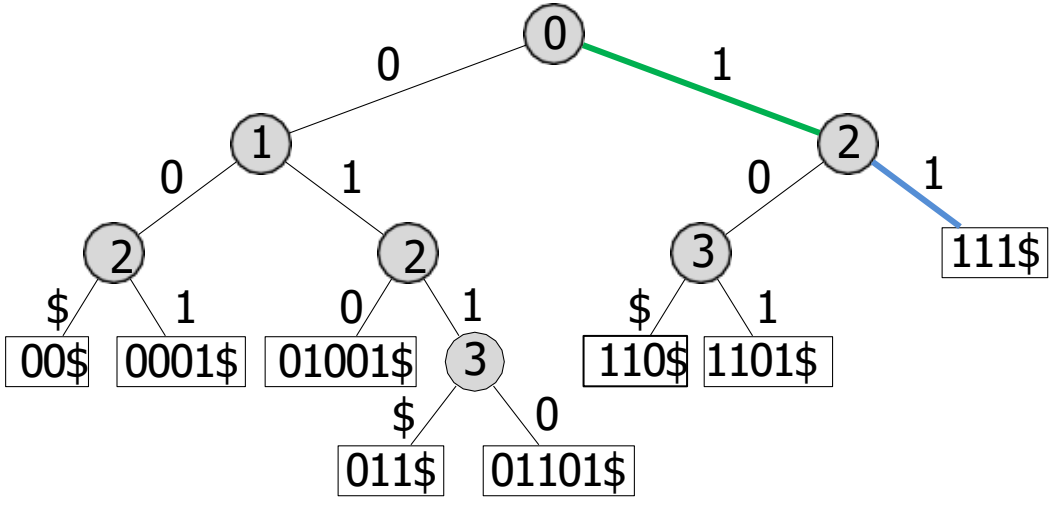
Example: Search(101$)

skip



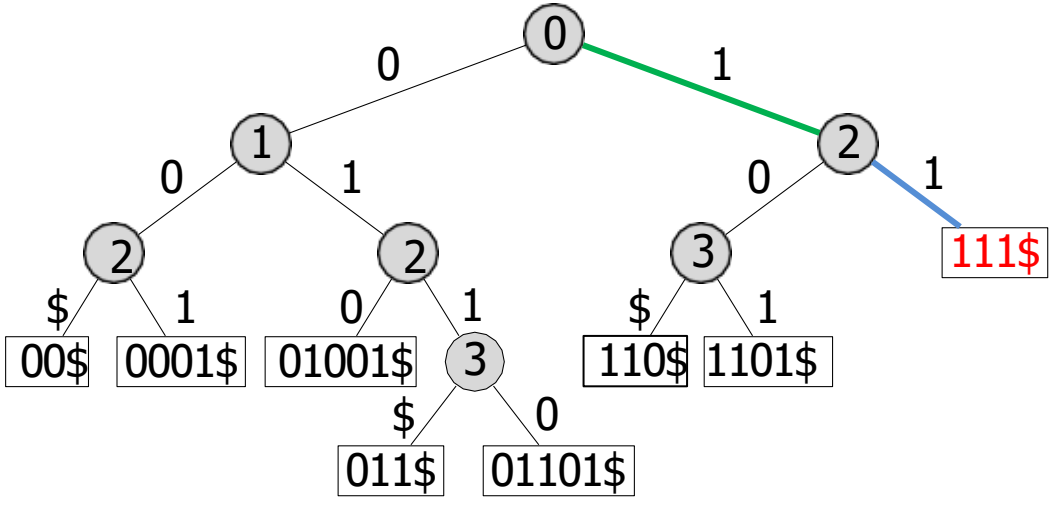
Compressed Tries: Search Example

Example: Search(101$)



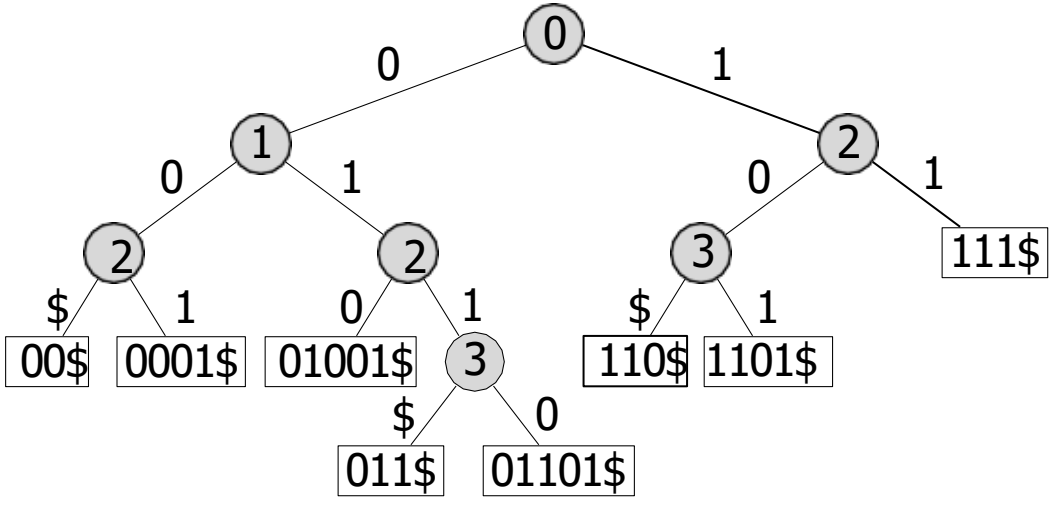
Compressed Tries: Search Example

Example: Search(101$) Unsuccessful (101$ is not equal 111$)



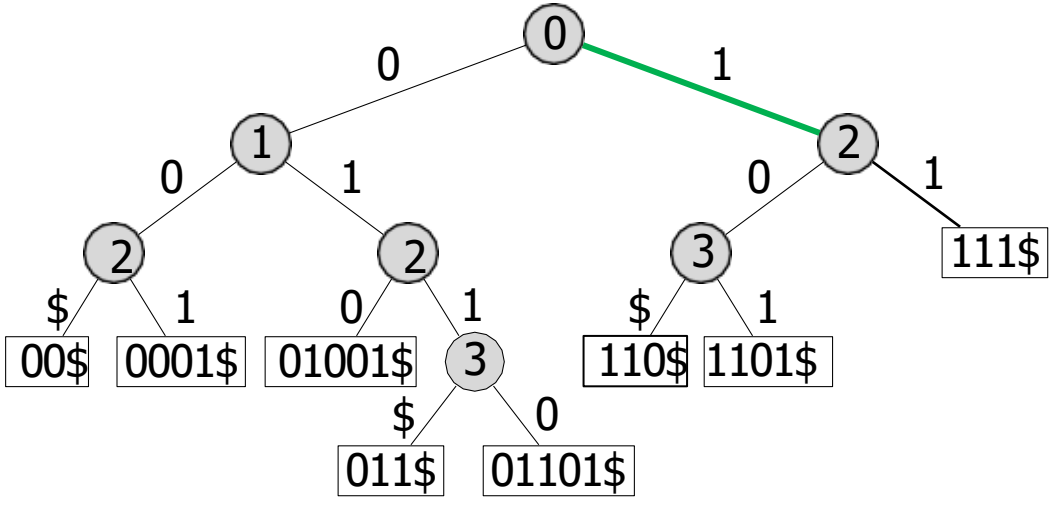
Compressed Tries: Search Example

Example: Search(111$)



Compressed Tries: Search Example

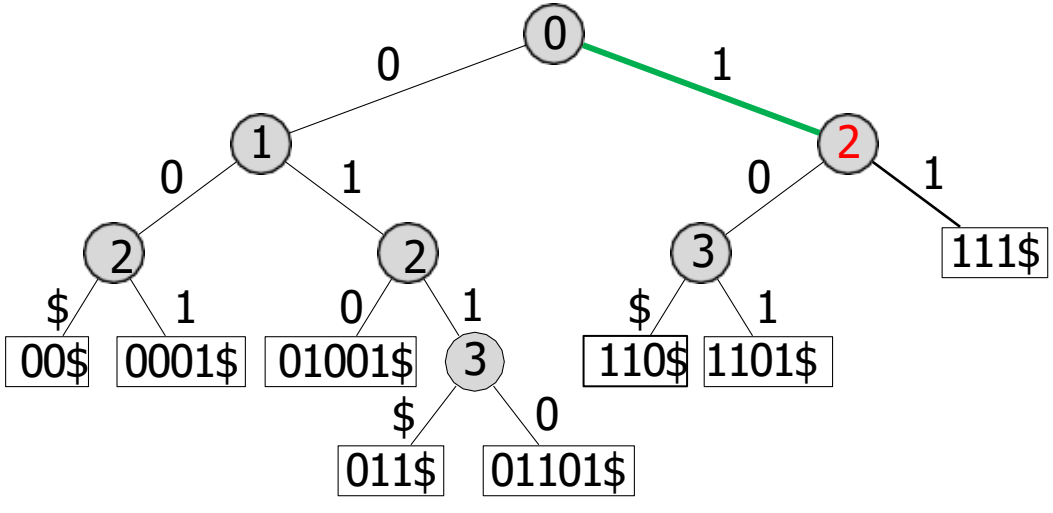
Example: Search(111$)



Compressed Tries: Search Example

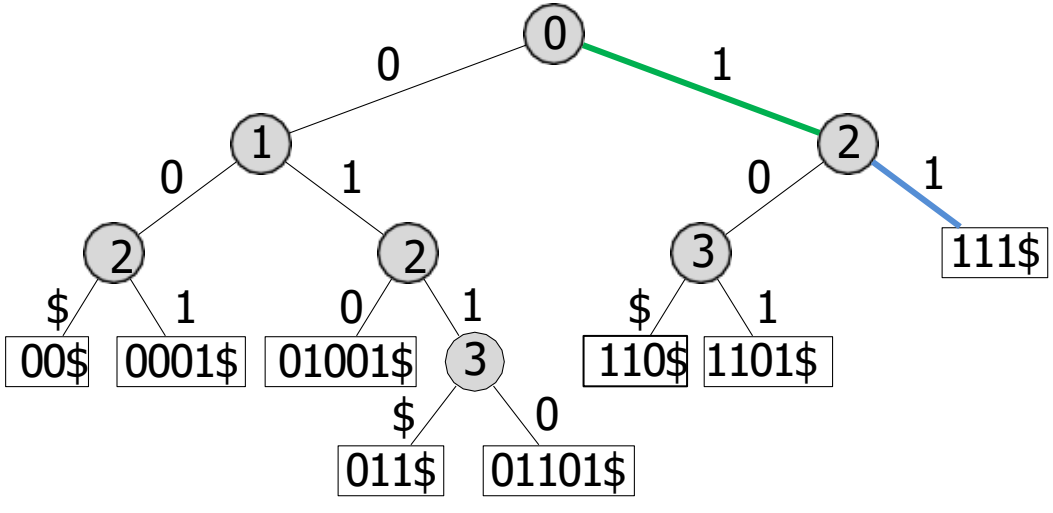
Example: Search(111$)

skip



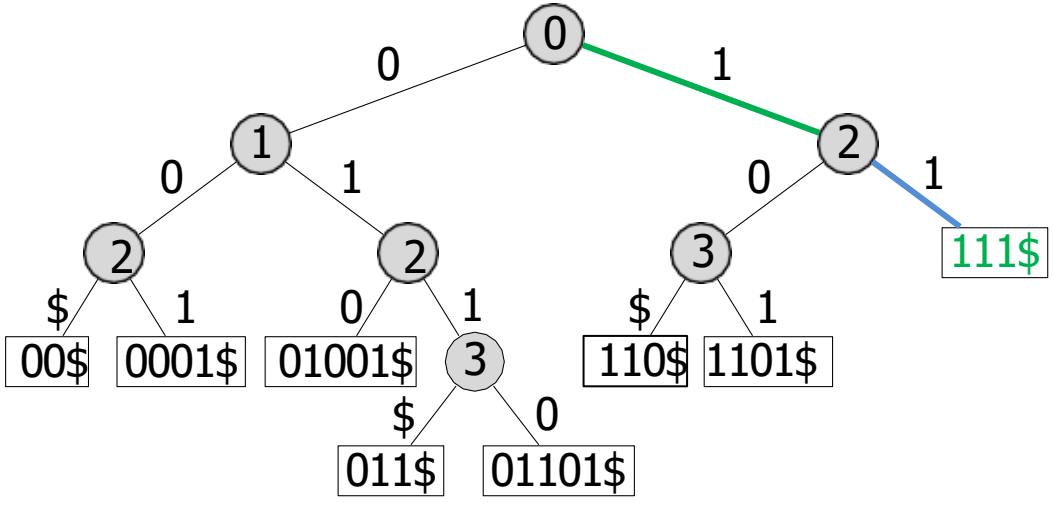
Compressed Tries: Search Example

Example: Search(111$)



Compressed Tries: Search Example

Example: Search(111$) successful (111$=111$)



Compressed Tries: Search

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

- 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 reach a leaf, expicitly check whether word stored at leaf is x
- Else recurse on the new node and the next bit of x

Compressed Tries: Insert & Delete

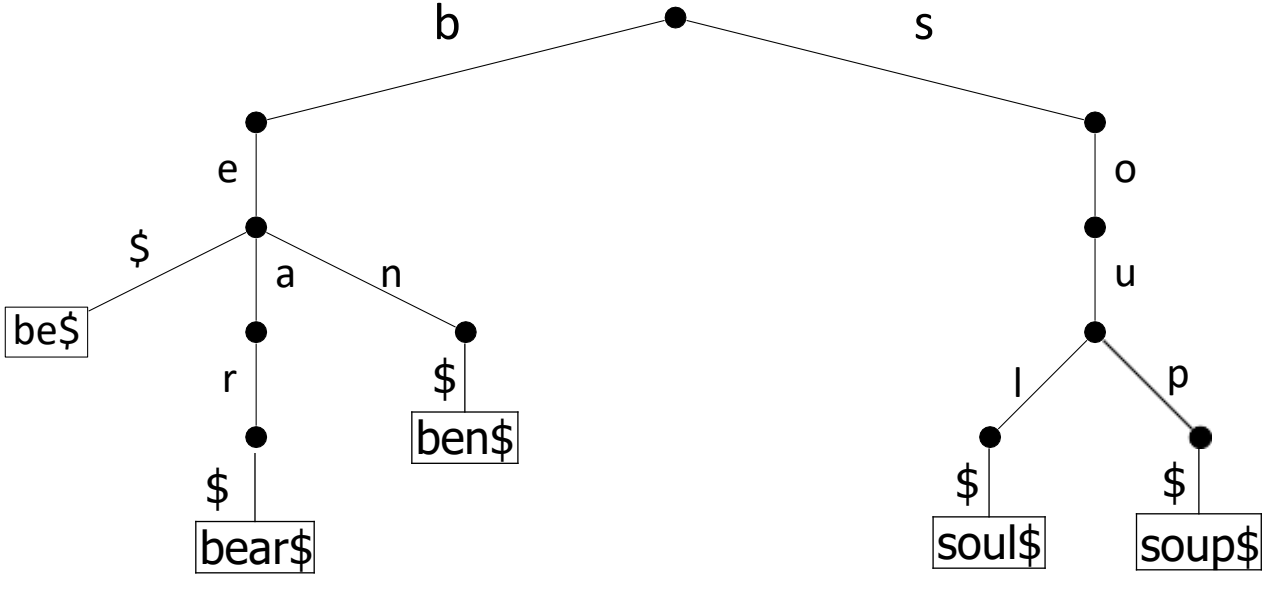
- Delete(x)
  - perform search(x)
  - remove the node v that stores x
  - compress along path to v whenever possible
- Insert(x)
  - perform search(x)
  - let v be node where search ends
  - conceptually simplest approach
    - uncompress path from root to v
    - insert x as in an uncompressed trie
    - compress paths from root to v and from root to x
  - can also be done by only adding those nodes that are needed
    - see the textbook for details
- All operations take $O(|x|)$ time
- Compressed tries are much more complicated, but space savings are worth it if words are unevenly distributed

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard Tries
    - Variations of Tries
    - Compressed Tries
    - Multiway Tries

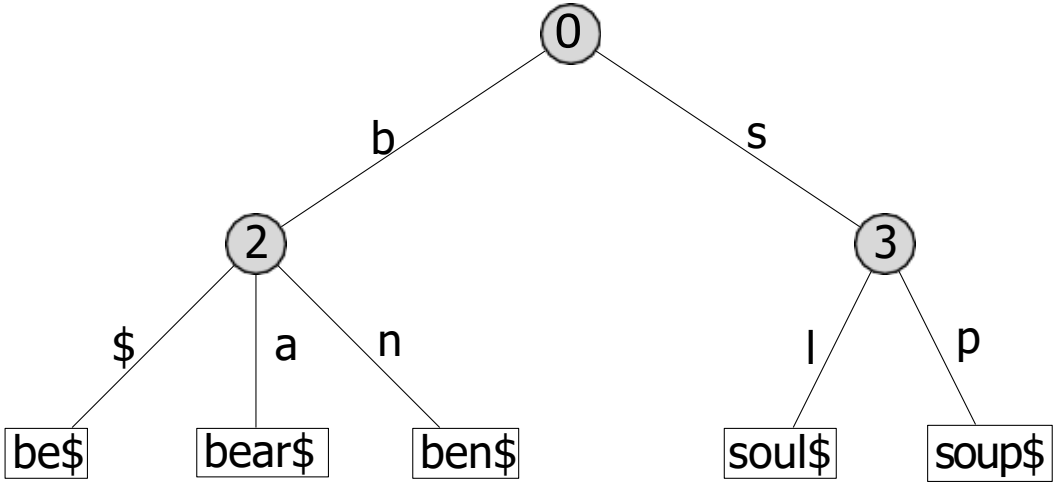
Multiway Tries: Larger Alphabet

- Represents Strings over any fixed alphabet Σ
- Any node has at most $|\Sigma| + 1$ children
  - one child for the end-of-word character $
- Example: A trie holding strings {bear$, ben$, be$, soul$, soup$}



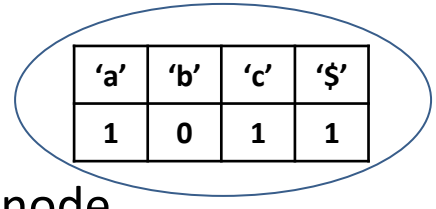
Compressed Multiway Tries

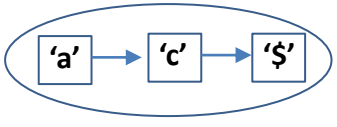
- Compressed multi-way tries
- Example: A compressed trie holding strings {bear$, ben$, be$, soul$, soup$}



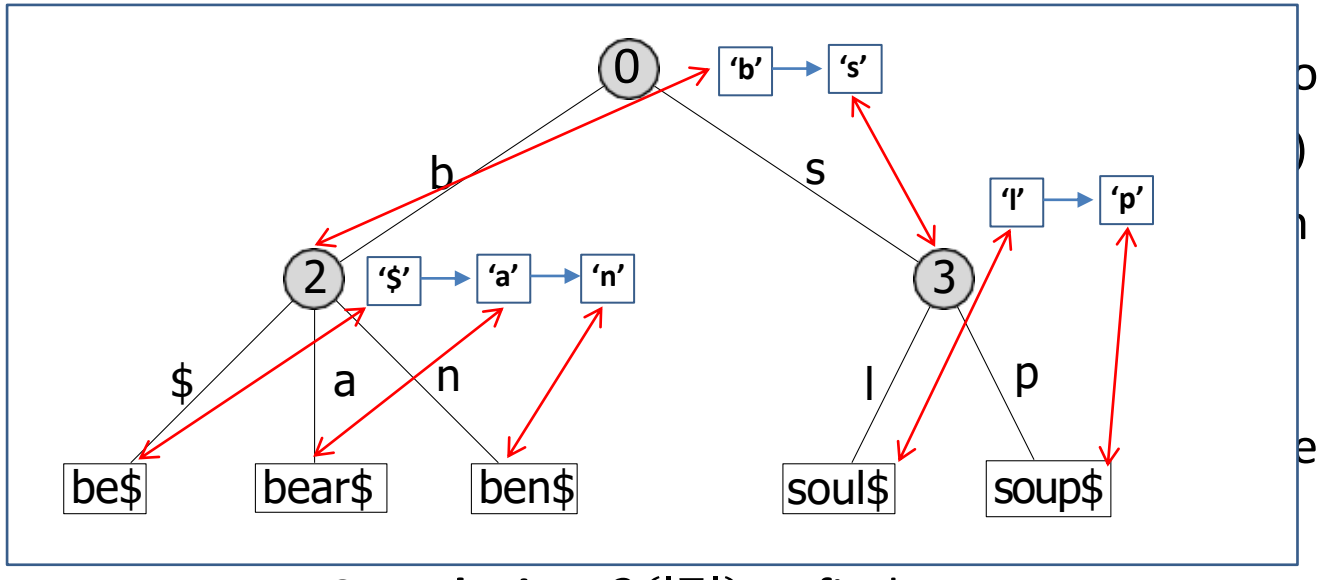
Multiway Tries: Summary

- Operations search(x), insert(x) and delete(x) are as for bitstring tries
- 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? Assume compressed trie
  - **Solution 1:** Array of size $|\Sigma| + 1$ for each node
    - **Complexity:** $O(1)$ to find child, $O(|\Sigma|)$ space per node
  - **Solution 2:** List of children for each node
    - **Complexity:** $O(|\Sigma|)$ to find child, $(\#children)$ space per node,
      - $O(n)$ total space assuming compressed trie
      - one-one correspondence between each trie node (except the root) and nodes of all linked lists

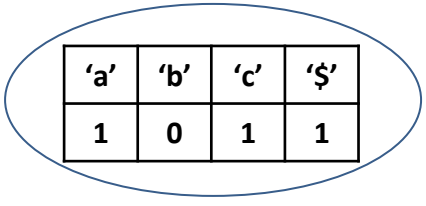


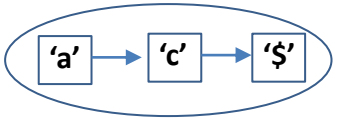


Multiway Tries: Summary



or bitstring tries

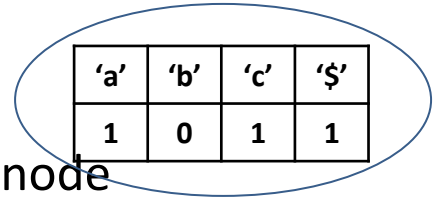




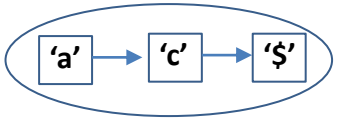
- **Complexity:** $O(|\Sigma|)$ to find child, $(\#children)$ space per node,
  - $O(n)$ total space assuming compressed trie
  - one-one correspondence between each trie node (except the root) and nodes of all linked lists

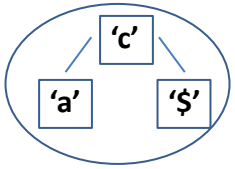
Multiway Tries: Summary

- Operations search(x), insert(x) and delete(x) are as for bitstring tries
- 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? Assume compressed trie
  - **Solution 1:** Array of size $|\Sigma| + 1$ for each node
    - **Complexity:** $O(1)$ to find child, $O(|\Sigma|)$ space per node



  - **Solution 2:** List of children for each node
    - **Complexity:** $O(|\Sigma|)$ to find child, $(\#children)$ space per node
      - $O(n)$ total space assuming compressed trie
      - one-one correspondence between each trie node (except the root) and nodes of all linked lists





  - **Solution 3:** AVL-tree of children for each node
    - **Complexity:** $O(\log(|\Sigma|))$ time to find a child, $O(n)$ space
    - best in theory, not worth it in practice unless $|\Sigma|$ is huge
  - **Solution 4:** *hashing,* often used in practice
    - keys are in typically small range Σ