CS 240 – Data Structures and Data Management

Module 6: Dictionaries for special keys

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2024

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Intro
    - Standard Trie
    - Pruned Trie
    - Compressed Trie
    - Multiway Trie

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Intro
    - Standard Trie
    - Pruned Trie
    - Compressed Trie
    - Multiway Trie

Dictionary ADT: Implementations Thus Far

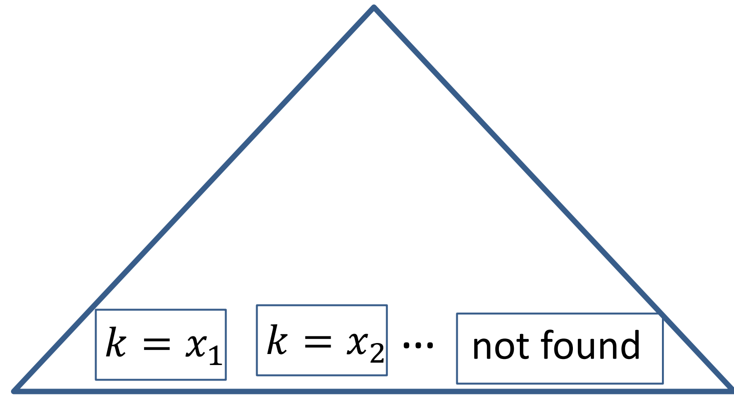
- Search is $\Theta(\log n)$ in fastest implementations of dictionary ADT
  - n is the number of items stored
- Search is $\Omega(\log n)$ in all realizations of ADT we know
- **Question**: Can we do better than $\Theta(\log n)$ search?
- **Answer**: *It depends on what we allow*
  - No: comparison-based searching lower bound is $\Omega(\log n)$
  - Yes: non-comparison based searching can achieve $o(\log n)$
    - keys have special properties
      1. Interpolation search: keys have special distribution
      2. Tries: keys are strings

Lower Bound For Search

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

Proof:

- Let algorithm A search for key for k among n items $x_1, x_2, \dots, x_n$
- There is a corresponding binary decision tree
- Chose a set of distinct keys $S = \{x_1, x_2, \dots, x_n\}$
- Consider $n + 1$ instances of search problem
    - search S for $k = x_1$
    - search S for $k = x_2$
    - …
    - search S for $k = x_n$
    - search S for k different from keys in S



decision tree

- Decision tree **must** have one leaf for each instance above
- Decision tree must have at least $(n + 1)$ leaves
- Binary tree of height h has at most 2^h leaves
- Thus $2^h \geq n + 1$
- Taking log of both sides, $h \geq \log(n + 1)$

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Intro
    - Standard Trie
    - Pruned Trie
    - Compressed Trie
    - Multiway Trie

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 ← 0
        r ← n − 1
        while (l ≤ r)
            m ← ⌊(l+r)/2⌋
            if (k = A[m])  return "found at A[m]"
            else if (A[m] < k) // key cannot be in the left part of A
                l ← m + 1
            else r ← m − 1  // key cannot be in the right part of A
        return "not found but would be between A[l − 1] and A[l] "
```

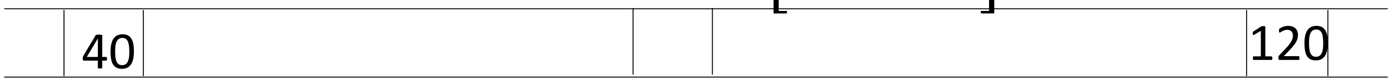
Interpolation Search: Motivation

- binary search looks at index

middle

$$\left\lfloor \frac{l + r}{2} \right\rfloor = l + \left\lfloor \frac{1}{2}(r - l) \right\rfloor$$

l r

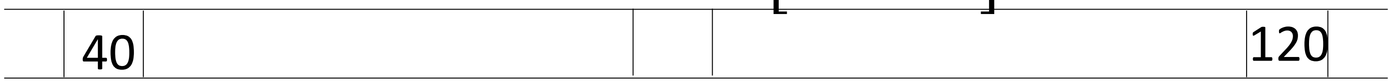


Interpolation Search: Motivation

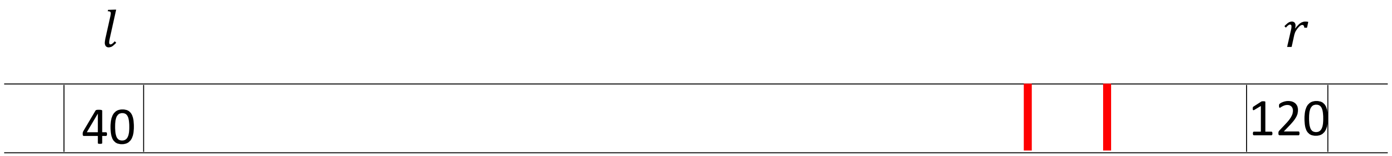
- binary search looks at index

middle

$$\left\lfloor \frac{l+r}{2} \right\rfloor = l + \left\lfloor \frac{1}{2}(r-l) \right\rfloor$$



- If keys are close to *evenly* distributed, where would key $k = 100$ be?



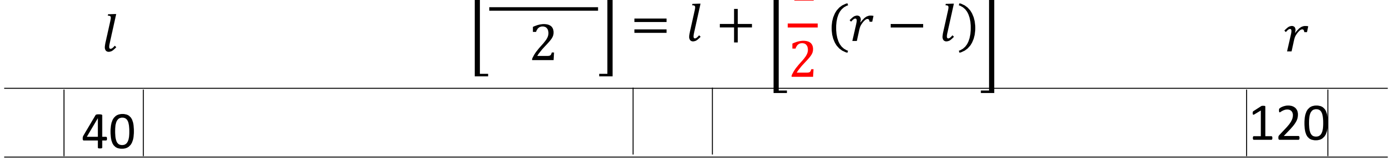
  - 100 should be much closer to $A[r] = 120$ than to $A[l] = 40$

Interpolation Search: Motivation

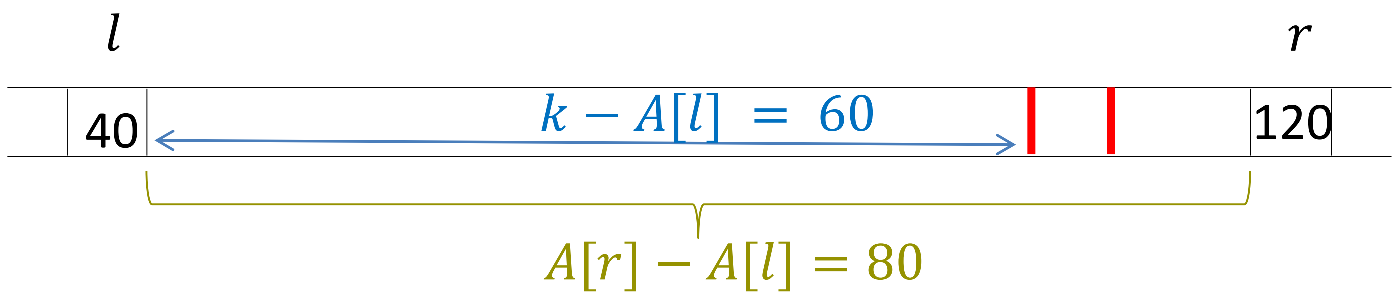
- binary search looks at index

middle

$$\left\lfloor \frac{l+r}{2} \right\rfloor = l + \left\lfloor \frac{1}{2}(r-l) \right\rfloor$$



- If keys are close to *evenly* distributed, where would key $k = 100$ be?



  - 100 should be much closer to $A[r] = 120$ than to $A[l] = 40$
  - fractional distance: $\frac{k-A[l]}{A[r]-A[l]} = 60/80 = \frac{3}{4}$ of the way between l and r

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

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

- *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

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

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

- *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

$l = 3,\ r = 4,$

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

Interpolation Search

- Works well if keys are close to *evenly* distributed
- But worst case performance on unevenly distributed keys is $\Theta(n)$
- Example: search(10)

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	1500

Interpolation Search

- Works well if keys are close to *evenly* distributed
- But worst case performance on unevenly distributed keys is $\Theta(n)$
- Example: search(10)

0	1	2	3	4	5	6	7	8	9	10
~~0~~	1	2	3	4	5	6	7	8	9	1500
l										r

- *Search*(10), iteration 1

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

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

Interpolation Search

- Works well if keys are close to *evenly* distributed
- But worst case performance on unevenly distributed keys is $\Theta(n)$
- Example: search(10)

0	1	2	3	4	5	6	7	8	9	10
0	~~1~~	2	3	4	5	6	7	8	9	1500
	l									r

- *Search*(10), iteration 2

$l = 1,\ r = 10,$

$$m = 1 + \left\lfloor \frac{10-1}{1500-1}(10-1) \right\rfloor = 1$$

Interpolation Search

- Works well if keys are close to *evenly* distributed
- But worst case performance on unevenly distributed keys is $\Theta(n)$
- Example: search(10)

0	1	2	3	4	5	6	7	8	9	10
0	1	~~2~~	3	4	5	6	7	8	9	1500
		l								r

- *Search*(10), iteration 3

$l = 2,\ r = 10,$

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

- Will continue in 'steps' of 1 at each iteration until reach the end of the array

Interpolation Search

- Works well on *average*
  - can show (difficult): $T^{avg}(n) \leq T^{avg}(\sqrt{n}) + \Theta(1)$
    - recurse into array of $\sqrt{n}$ size, on average
  - resolves to $T^{avg}(n) \in \Theta(\log \log n)$

- 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 $r \leftarrow m - 1$
// always return from inside the while loop

Outline

- Lower bound for search
- Interpolation Search
- Tries
  - Intro
  - Standard Trie
  - Pruned Trie
  - Compressed Trie
  - Multiway Trie

Tries: Introduction

- **Scenario**: Keys in dictionary are words
- Words (=strings): sequence of characters over alphabet Σ

  `{be, bear, beer}`

- Typical alphabets: $\{0,1\}$ (bitstrings), ASCII, etc.
- Stored in an array: $w[i]$ gets ith character (for $i = 0,1, \dots$)
- **Convention**: words have end-sentinel $ (sometimes not shown)
- $w.size = |w| =$ number of non-sentinel characters
    - |be$| = 2
    - $ is smaller than any other character and does not occur in Σ
- Should know
    - prefix, suffix, substring
    - sorting of words lexicographically

      be$ $<_{lex}$ bear$ $<_{lex}$ beer$

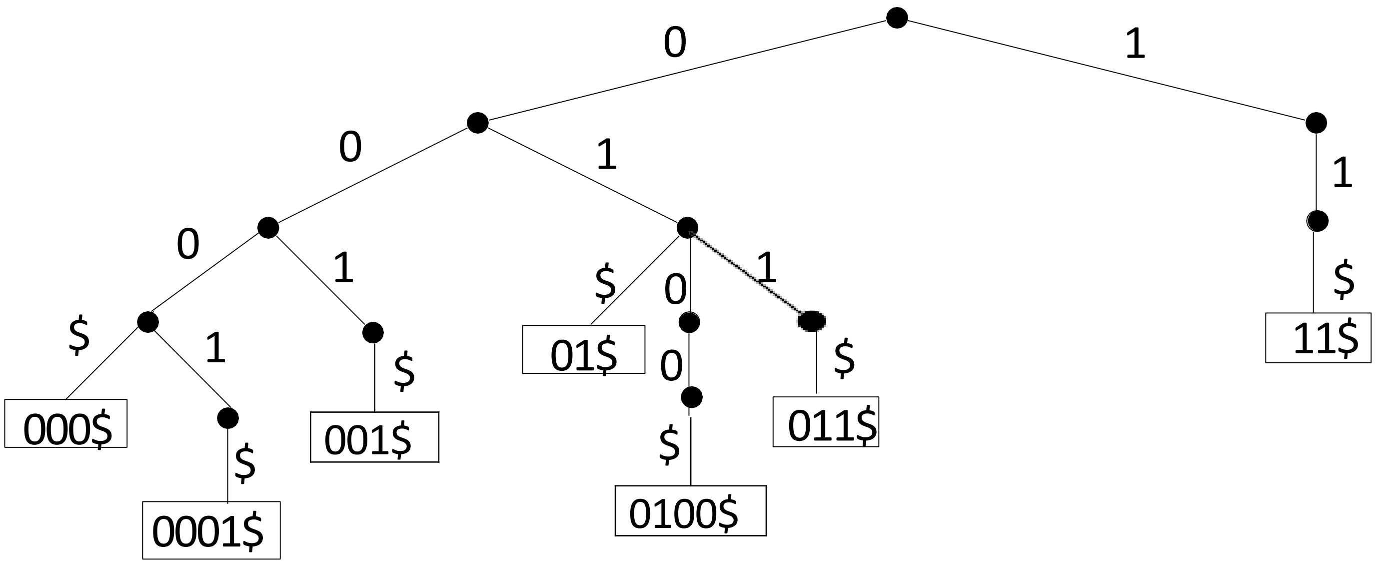
        - this is different from sorting numbers

          010$ $<_{lex}$ 1$

Tries: Introduction

- **Trie** (also known as **radix tree**): a dictionary for bit strings
    - comes from word retrieval, but pronounced "try"
- Trie vs. AVL tree
    - let the number of strings in dictionary be n
    - Trie: insert, find, delete is $O(|w|)$ time
        - independent of n
    - AVL tree: insert, find delete is $O(|w|\log(n))$ time
        - $O(\log(n))$ to search, $O(|w|)$ operations at each node
- Trie applications
    - auto-completion
        - smart phones, commands for operating systems
    - spell checking
    - DNA sequencing

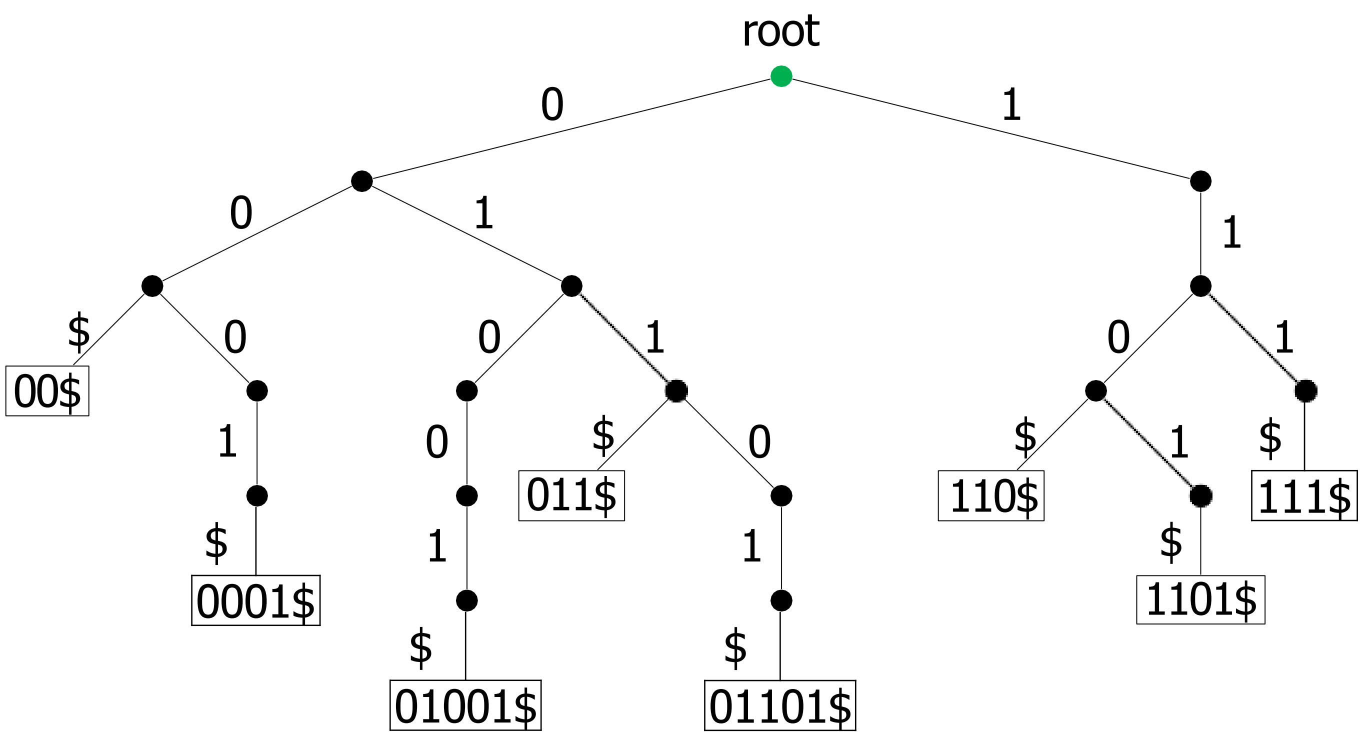
Tries: Introduction



- Trie (radix tree): dictionary for bitstrings
  - tree based on bitwise comparisons
  - edges 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

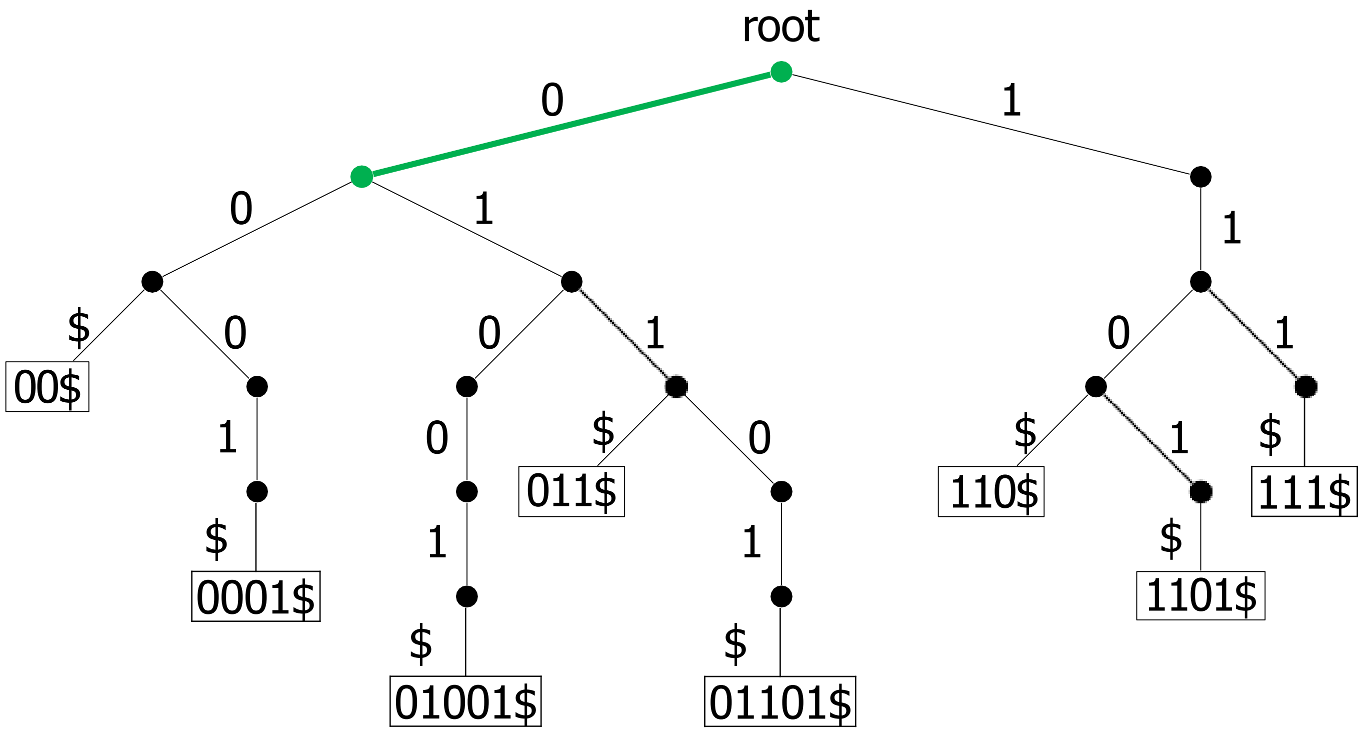
Tries: Search Example

Example: Search(011$)



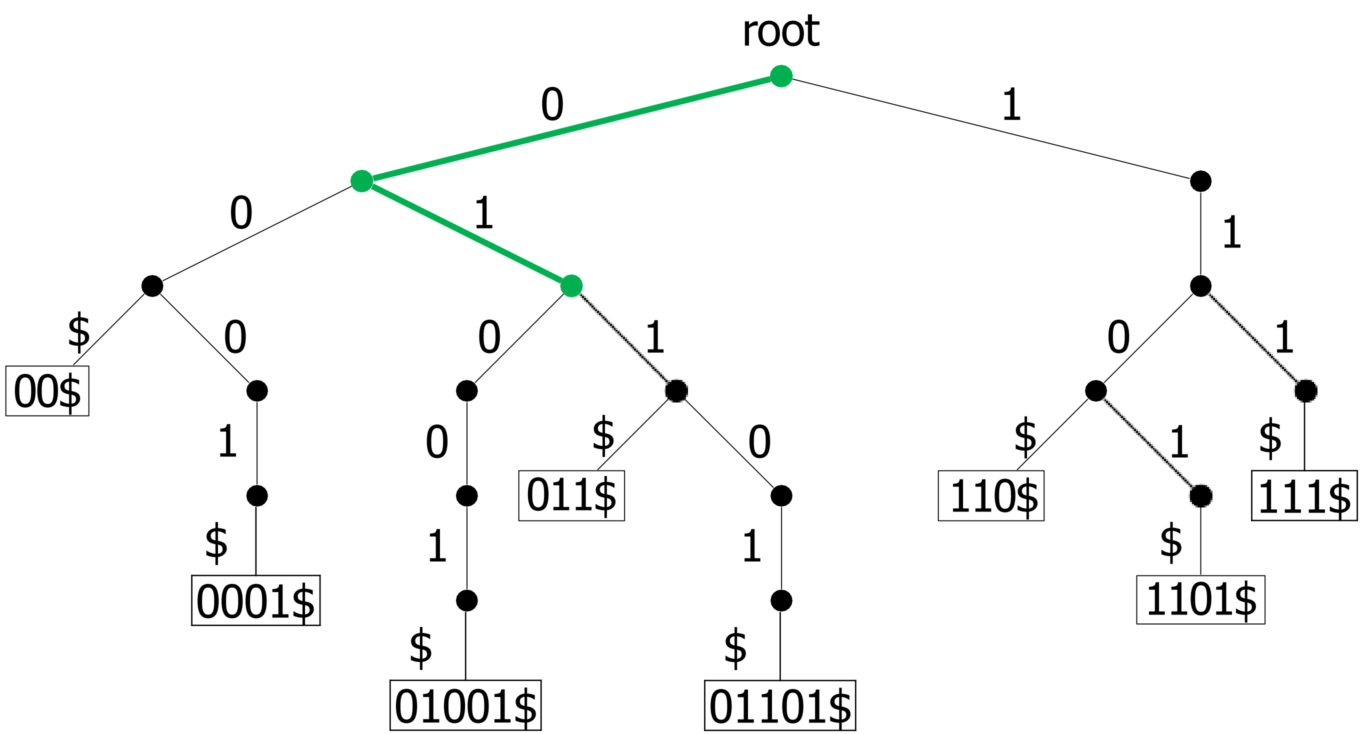
Tries: Search Example

Example: Search(011$)



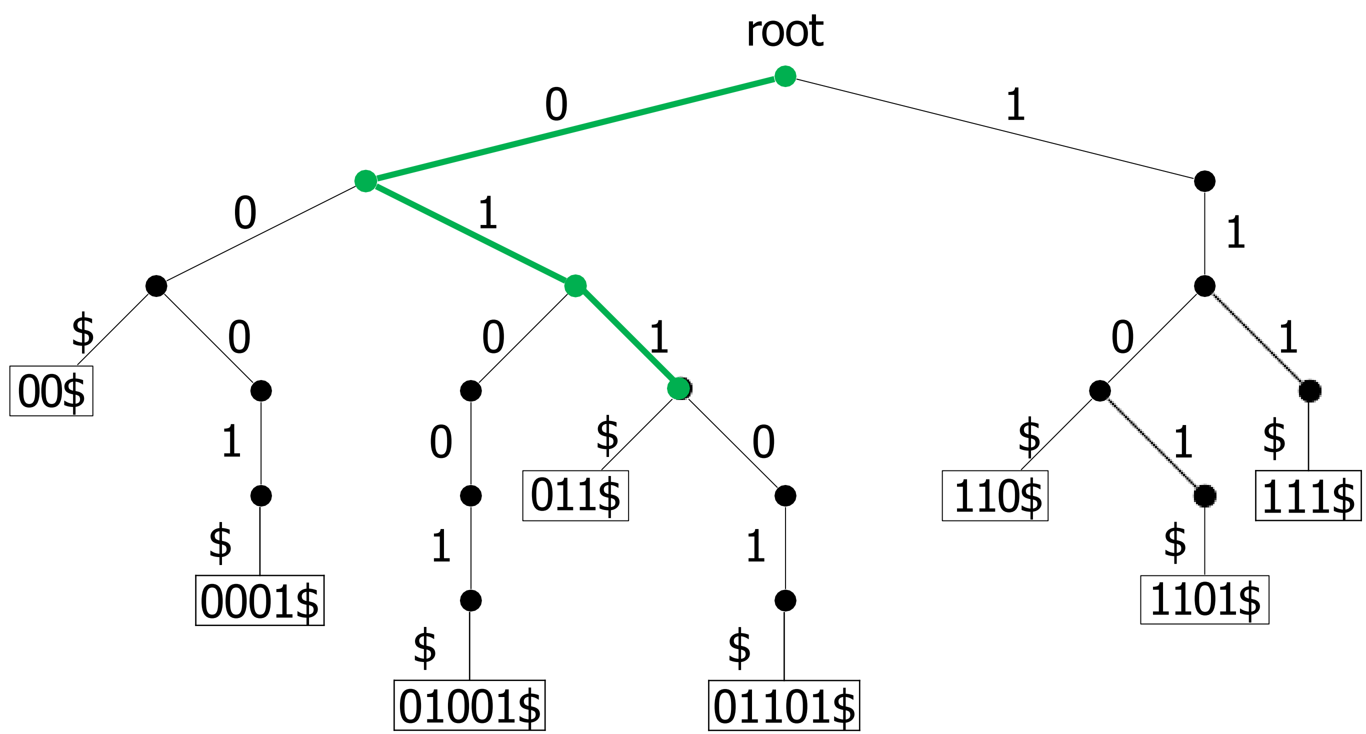
Tries: Search Example

Example: Search(011$)



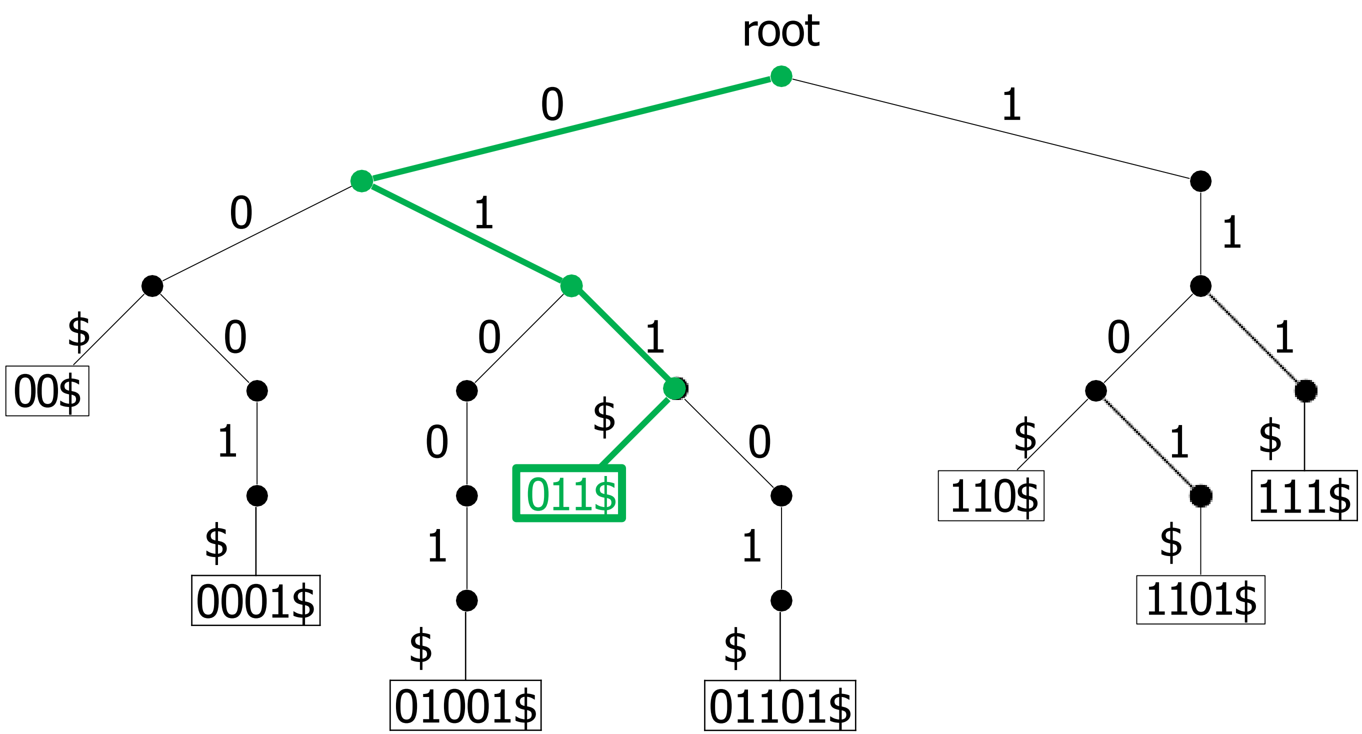
Tries: Search Example

Example: Search(011$)



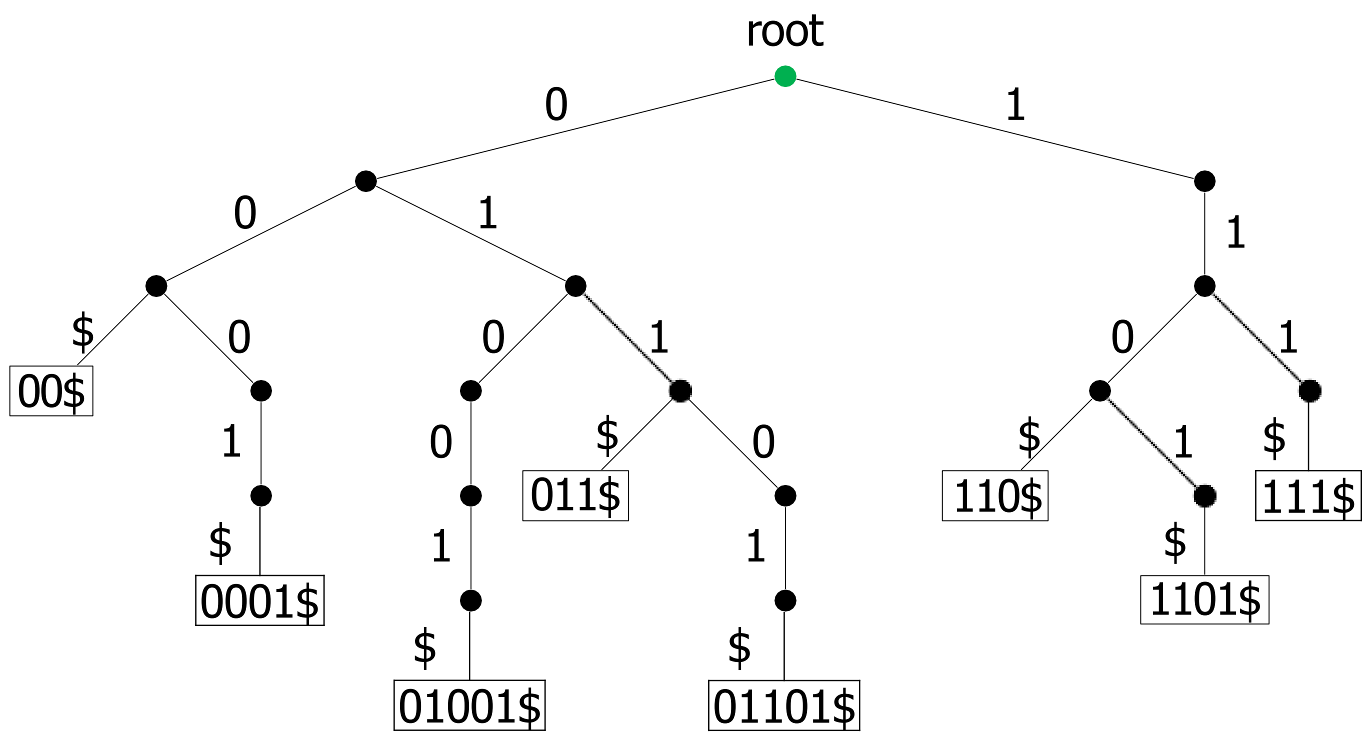
Tries: Search Example

Example: Search(011$) successful

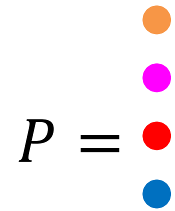


Tries: Search Example

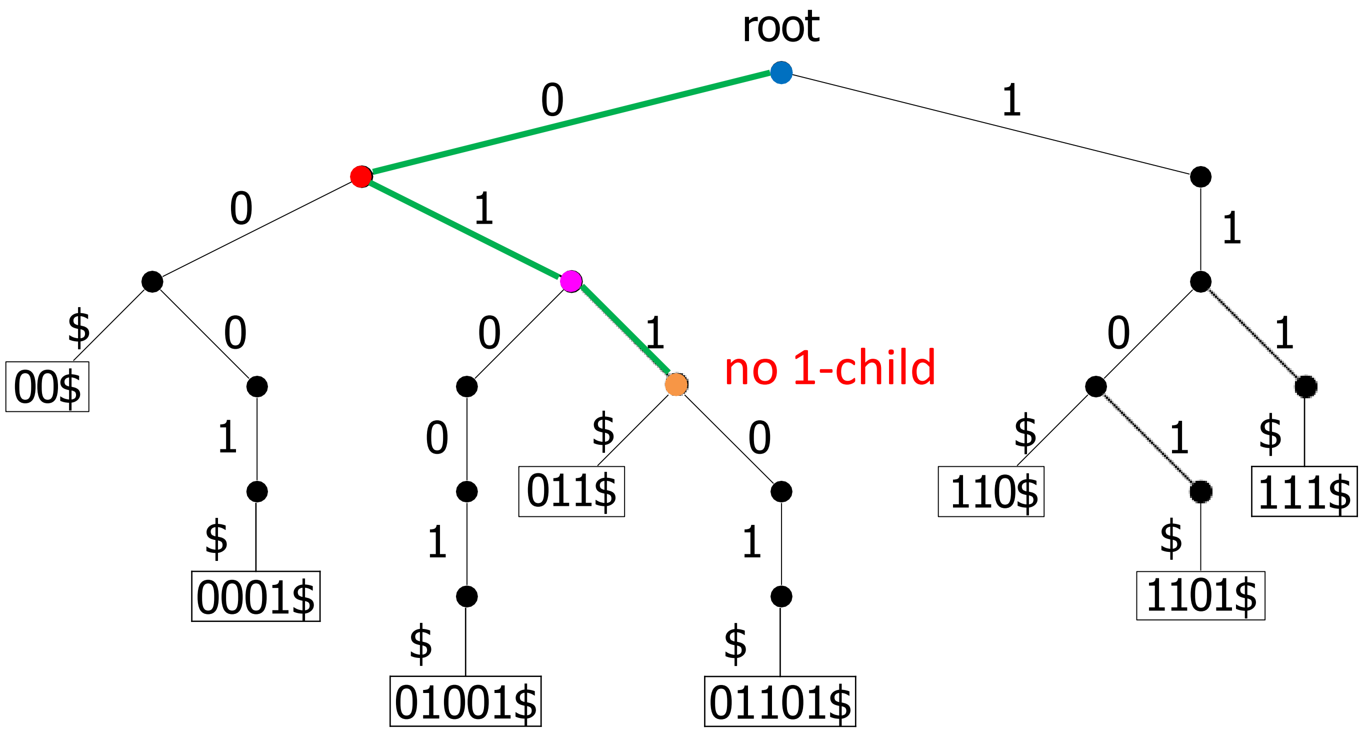
Example: Search(0111$)



Tries: Search Example



Example: Search(0111$) unsuccessful



Tries: Search

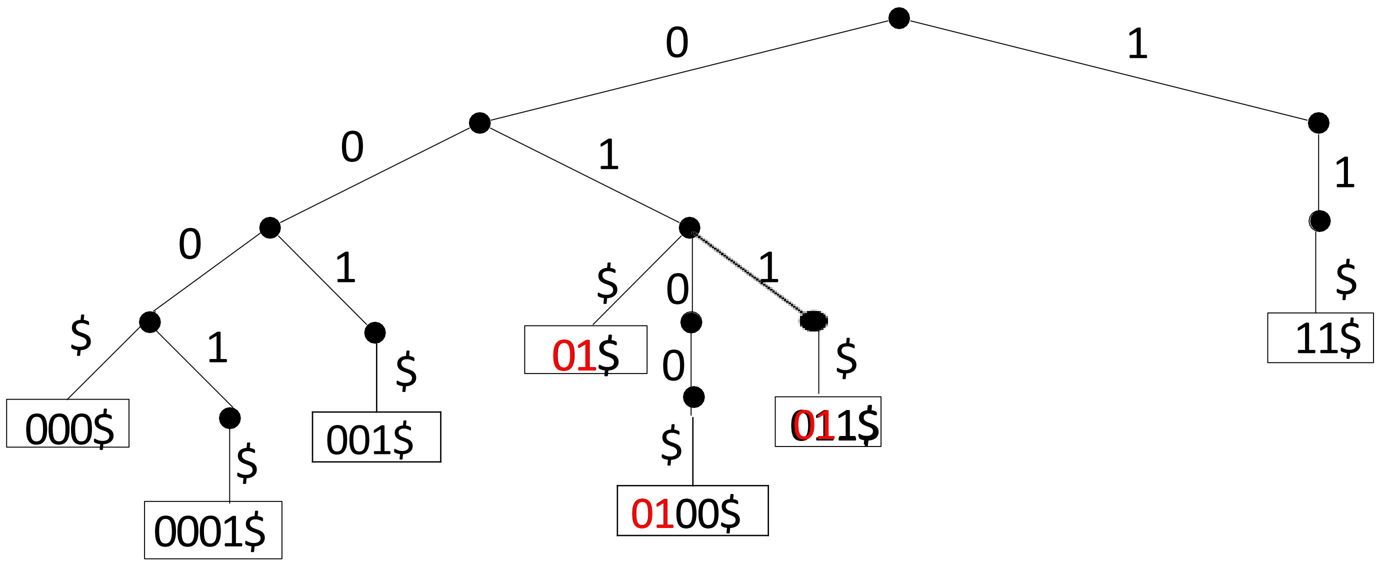
- Follow links that correspond to current bits in w
- Repeat until w is found or no such link

```
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
```

```
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-Refernces

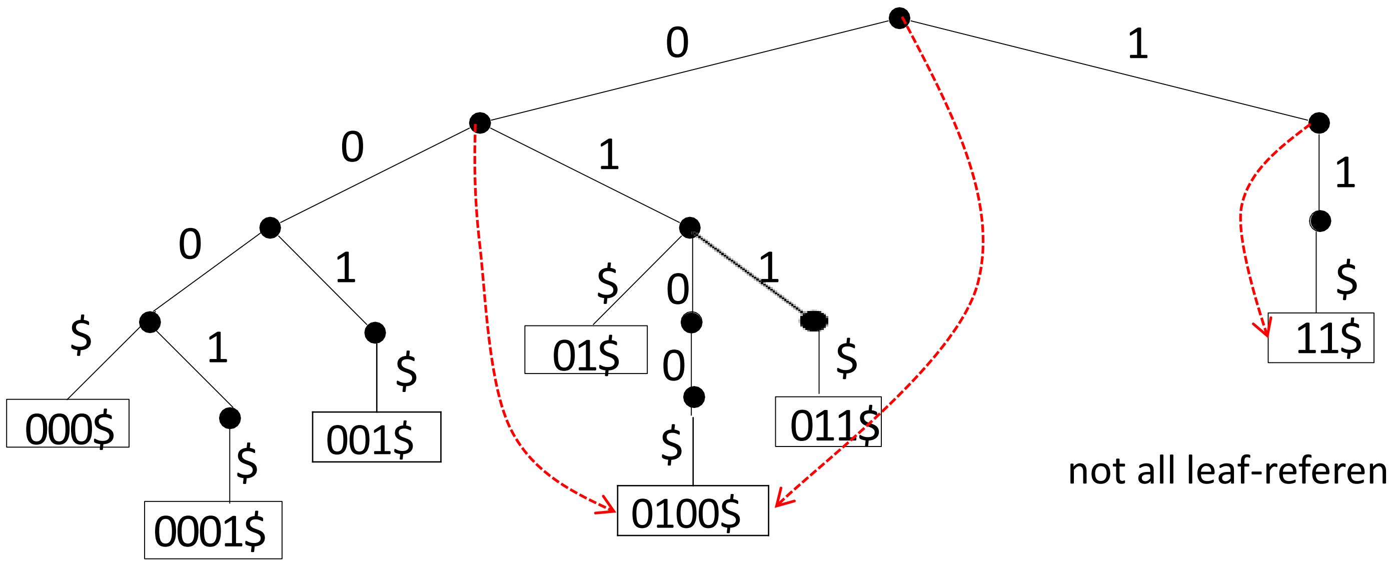
- For later applications of tries, want prefix-search(w)
  - find word v in a trie for which w is a prefix



prefix-search$(01\$)$ can return: $01\$$ or $0100\$$ or $011\$$

Tries: Leaf-Refernces

- For later applications of tries, want prefix-search(w)
  - find word v in a trie for which w is a prefix

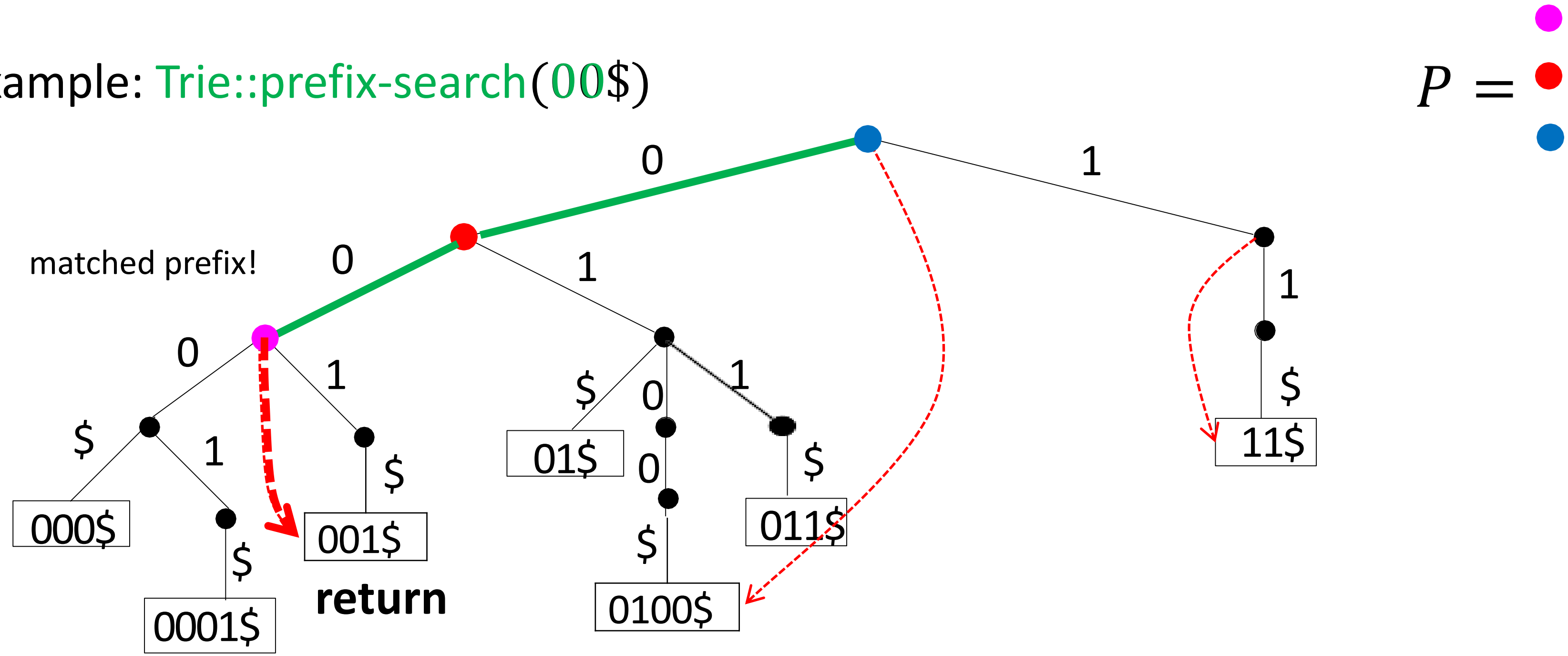


not all leaf-references are shown

- To find v quickly, need leaf-references
- Convention: reference to leaf with longest word in the subtree
  - ties broken arbitrarily

Tries: Leaf-Refernces

- Example: Trie::prefix-search($00\$$)

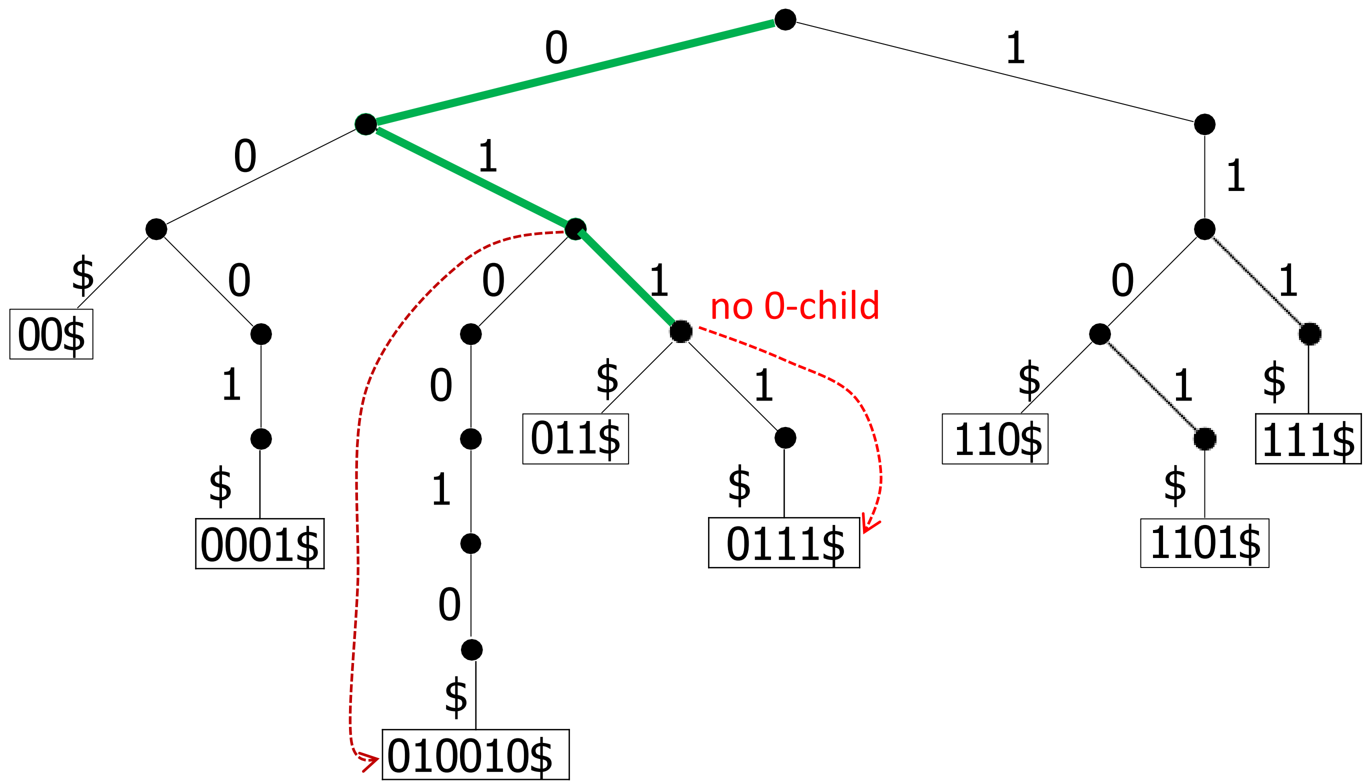


- If match, stack size is larger by exactly 1 than size of prefix w
  - 1 node for the root
  - 1 node for each character of w

Trie::prefix-search(w)
$P \longleftarrow$ *get-path-to*(w); $p \longleftarrow P$.*top*()
if number of nodes on P is $w.size$ or less **then**
    return "not string with prefix w found"
return $p.leaf$

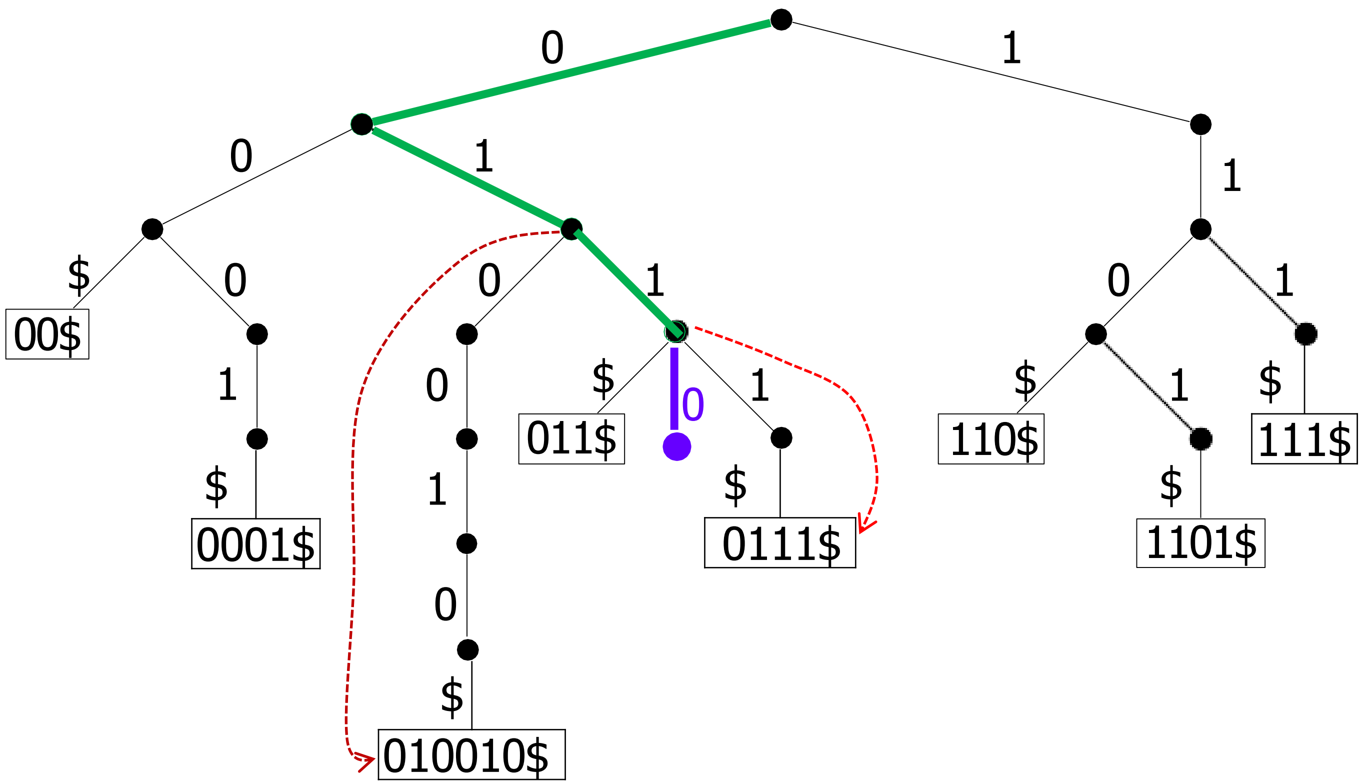
Tries: Insert

- $P \longleftarrow$ *get-path-to*(w) gives ancestors that exist already
- Expand trie from $p \longleftarrow P$.*top*() by adding nodes for the extra bits of w
- Update leaf-references (also at P if w is longer than previous leaves)
- Example: Insert(01101$)



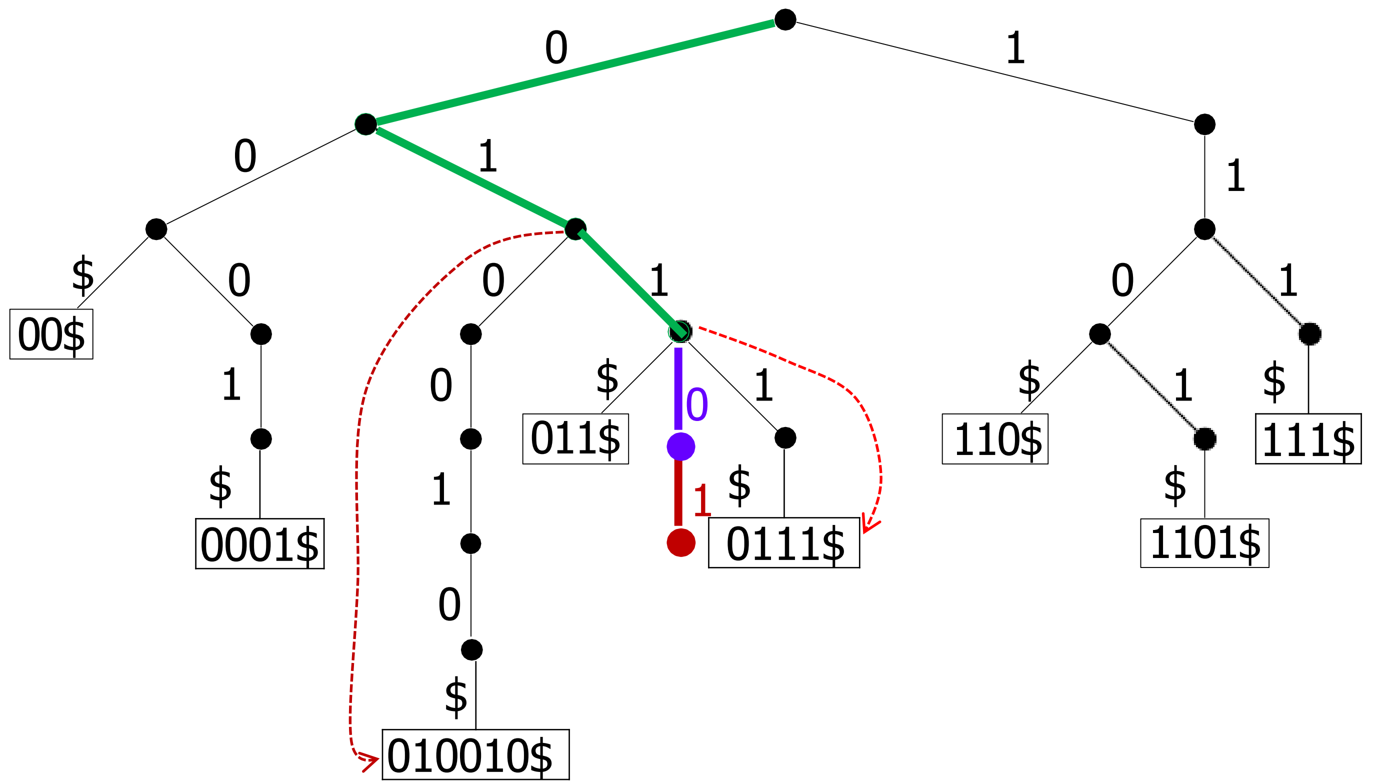
Tries: Insert

- $P \longleftarrow$ *get-path-to*(w) gives ancestors that exist already
- Expand trie from $p \longleftarrow P$.*top*() by adding nodes for the extra bits of w
- Update leaf-references (also at P if w is longer than previous leaves)
- Example: Insert(01101$)



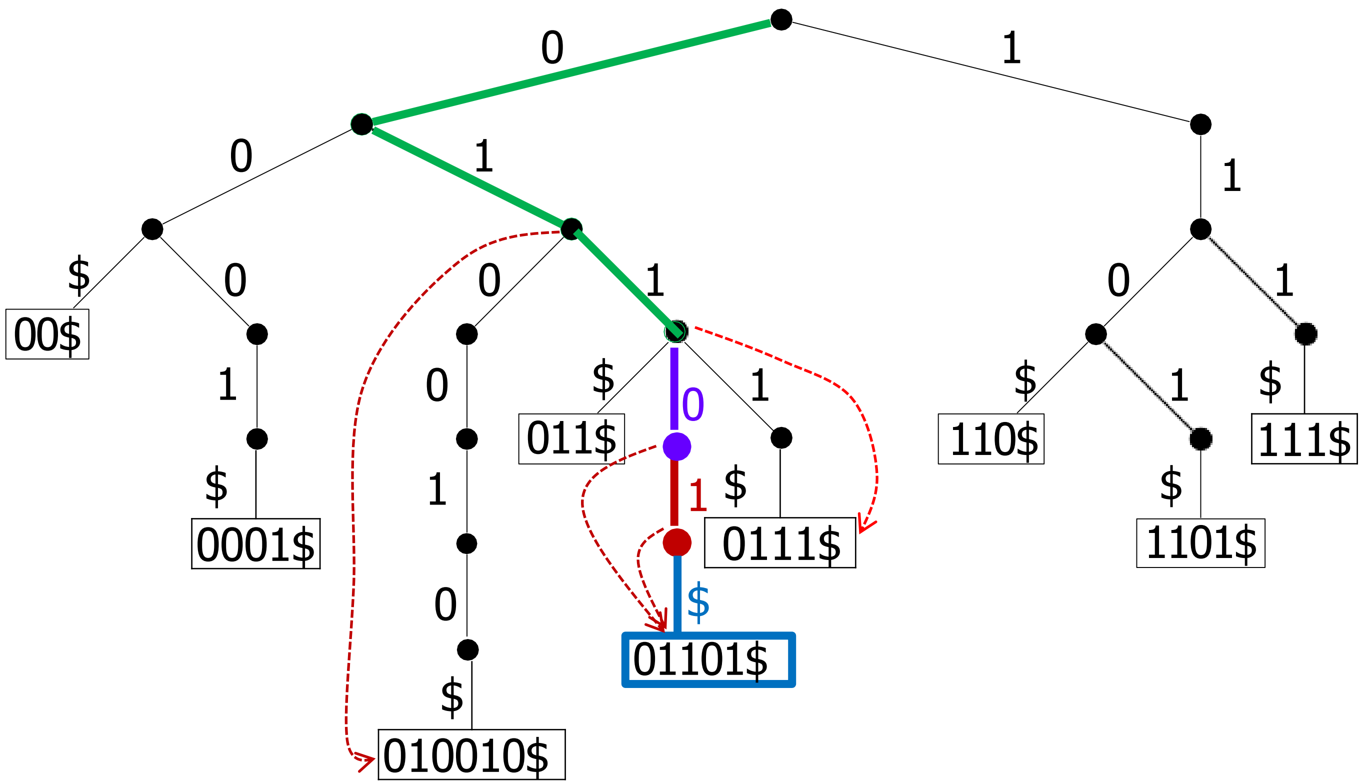
Tries: Insert

- $P \longleftarrow$ *get-path-to*(w) gives ancestors that exist already
- Expand trie from $p \longleftarrow P.top()$ by adding nodes for the extra bits of w
- Update leaf-references (also at P if w is longer than previous leaves)
- Example: Insert(01101$)



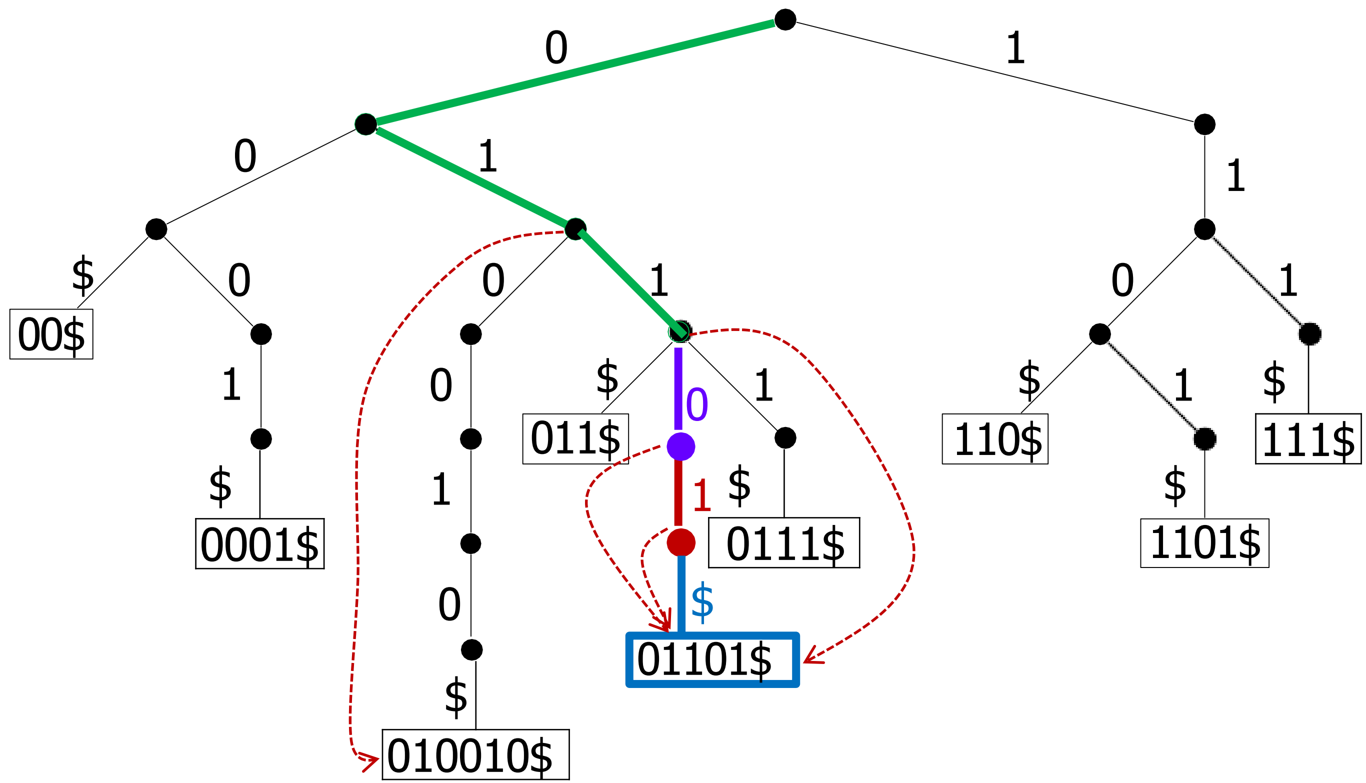
Tries: Insert

- $P \longleftarrow$ *get-path-to*(w) gives ancestors that exist already
- Expand trie from $p \longleftarrow P$.*top*() by adding nodes for the extra bits of w
- Update leaf-references (also at P if w is longer than previous leaves)
- Example: Insert(01101$)



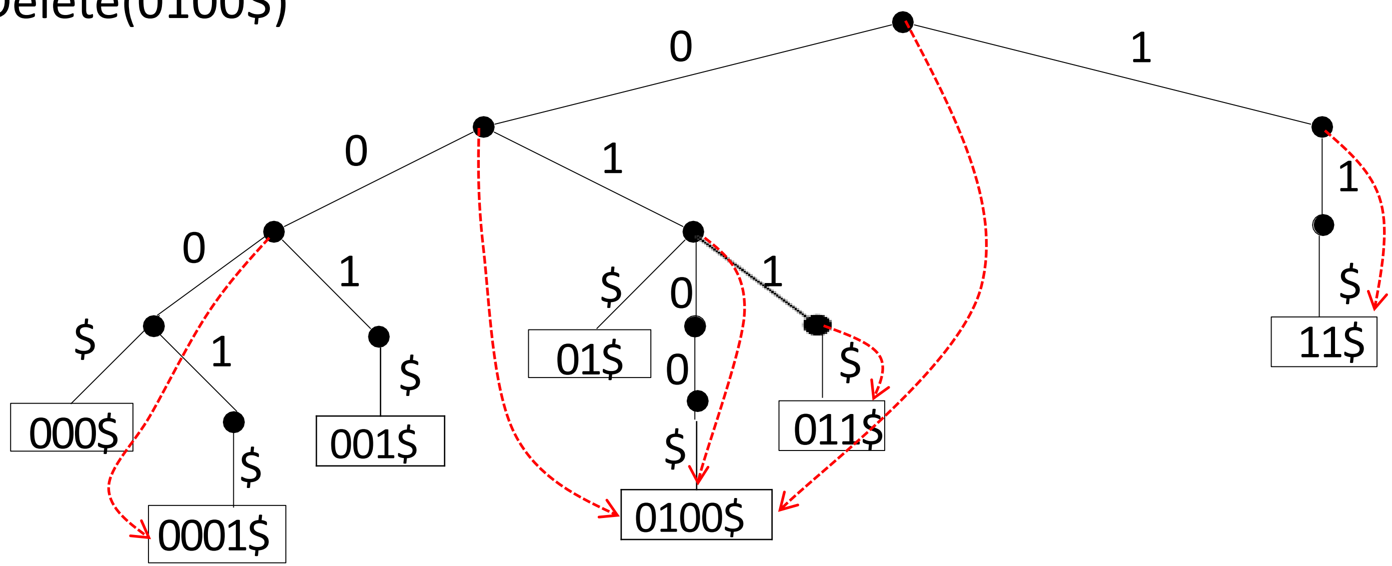
Tries: Insert

- $P \longleftarrow$ *get-path-to*(w) gives ancestors that exist already
- Expand trie from $p \longleftarrow P.top()$ by adding nodes for the extra bits of w
- Update leaf-references (also at P if w is longer than previous leaves)
- Example: Insert(01101$)



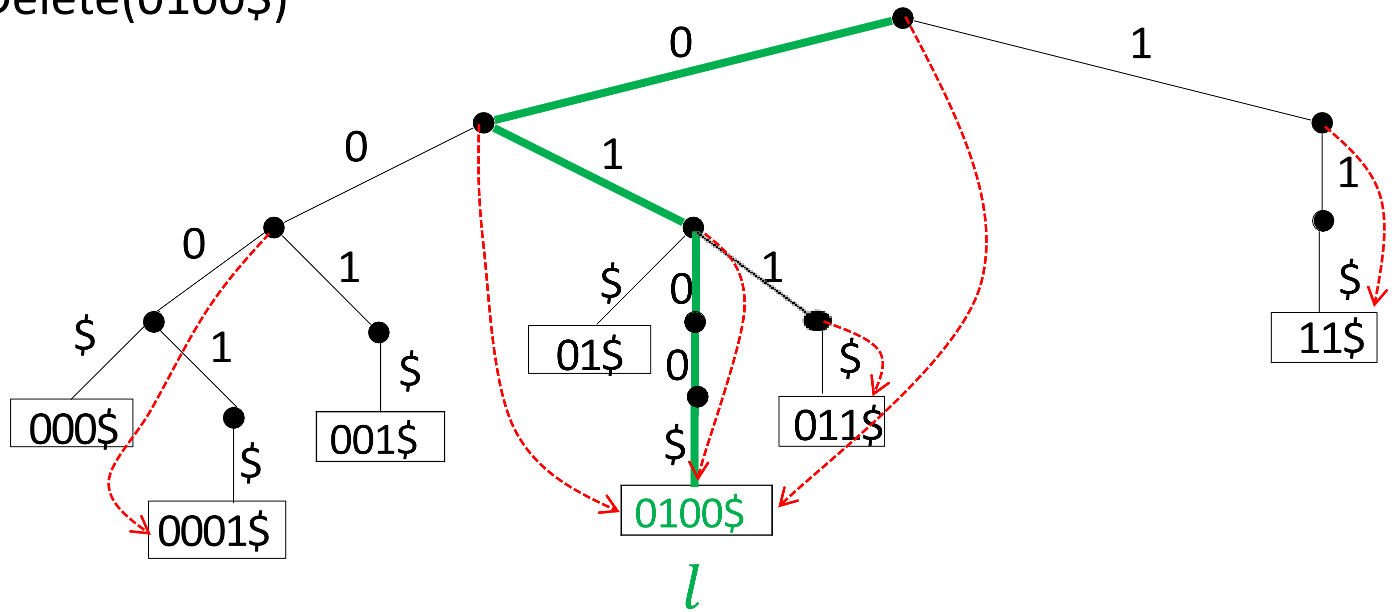
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
  - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



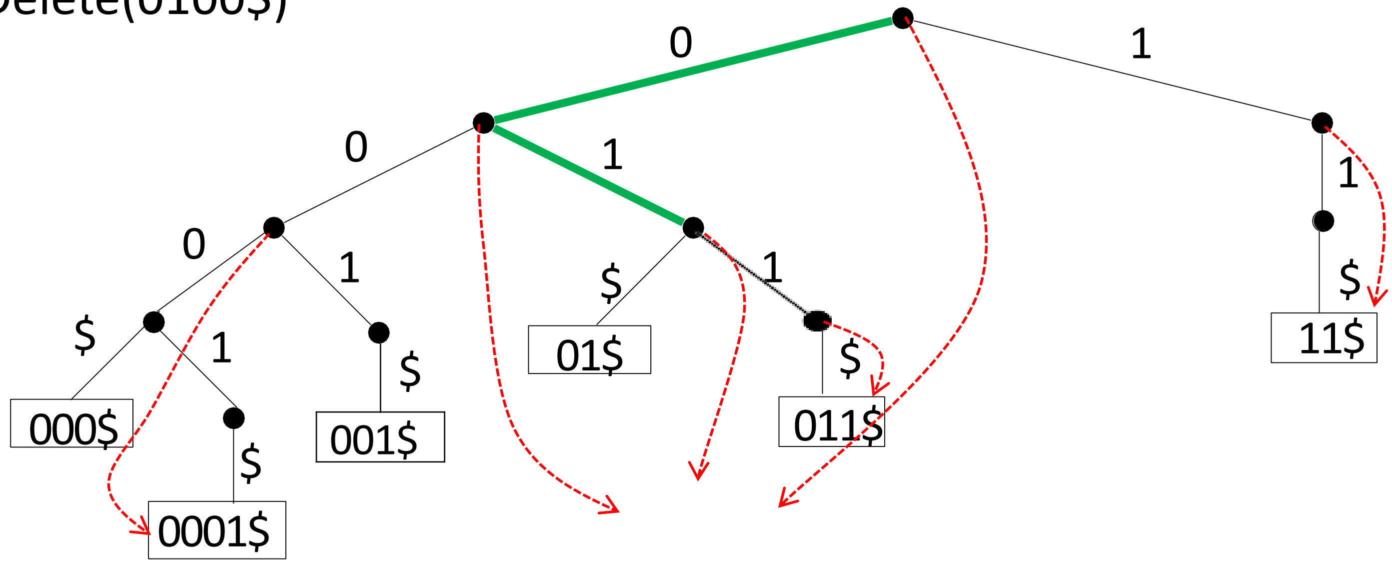
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
  - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



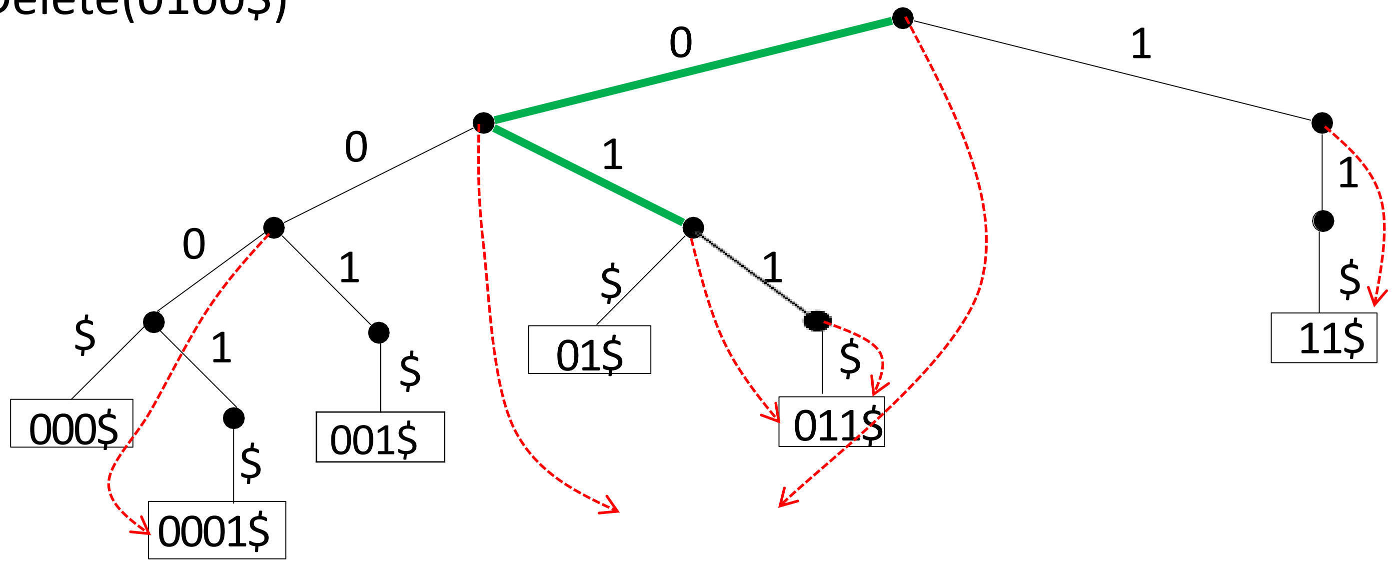
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
    - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



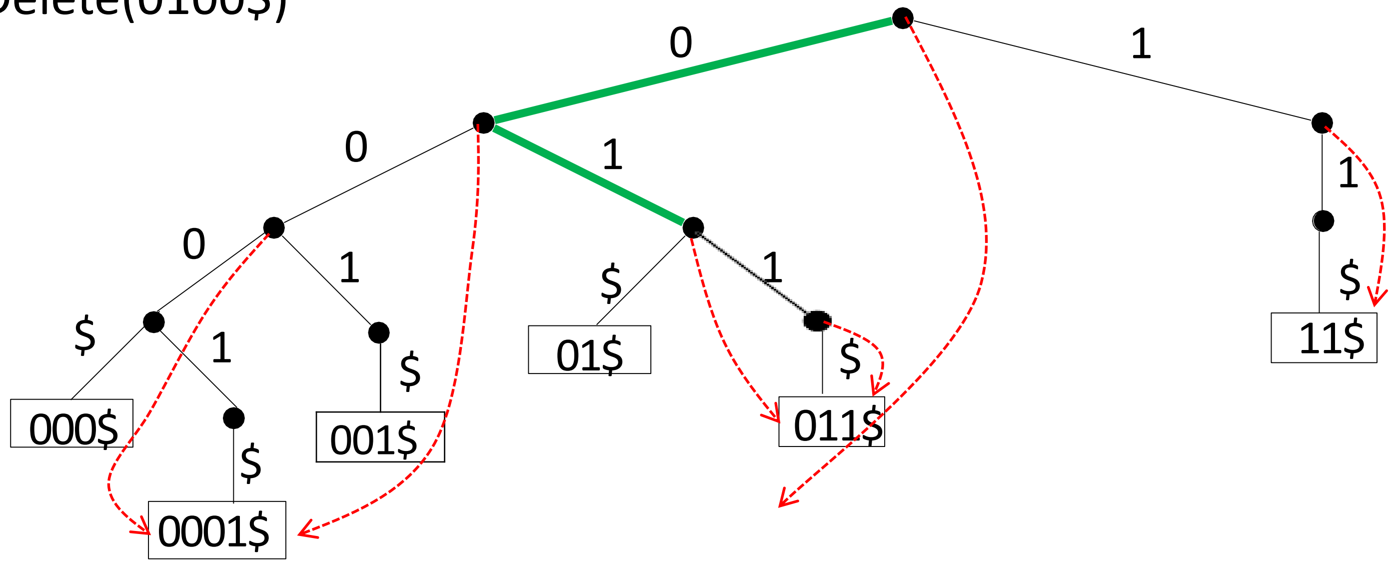
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
  - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



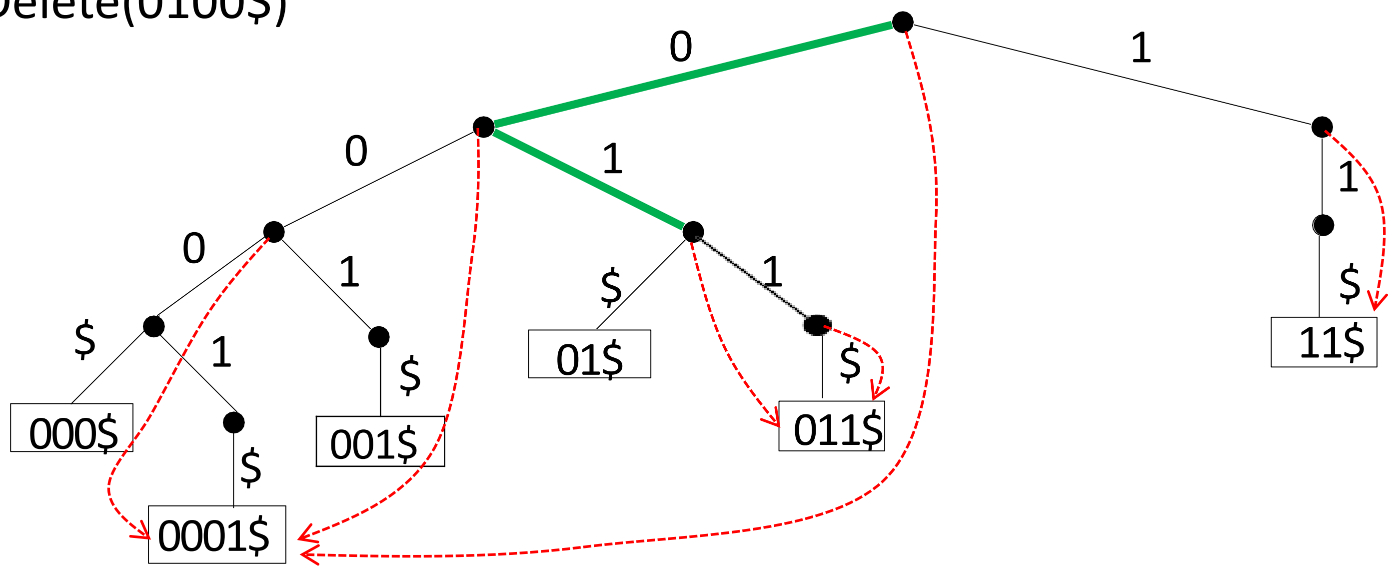
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
  - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



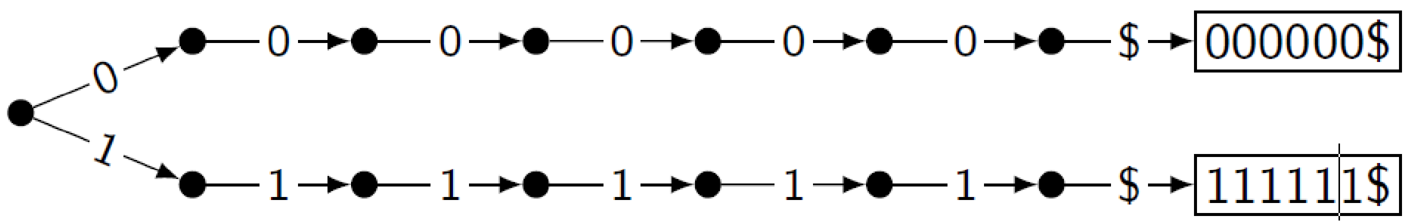
Tries: Delete

- $P \longleftarrow$ *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 the rest of P
  - if $z \epsilon P$ referred to l, find new $z.leaf$ from current children of z
- Delete(0100$)



Standard Trie Summary

- search(w), prefix-search(w), insert(w), delete(w) all take $\Theta(|w|)$ time
  - time is independent of n, the number of words stored in the trie
  - time is small for short words
- Trie for a given set of words is unique
  - except for order of children and ties among leaf-references
- Disadvantages
  - can be wasteful with respect to space
    - the problem is 'chains'



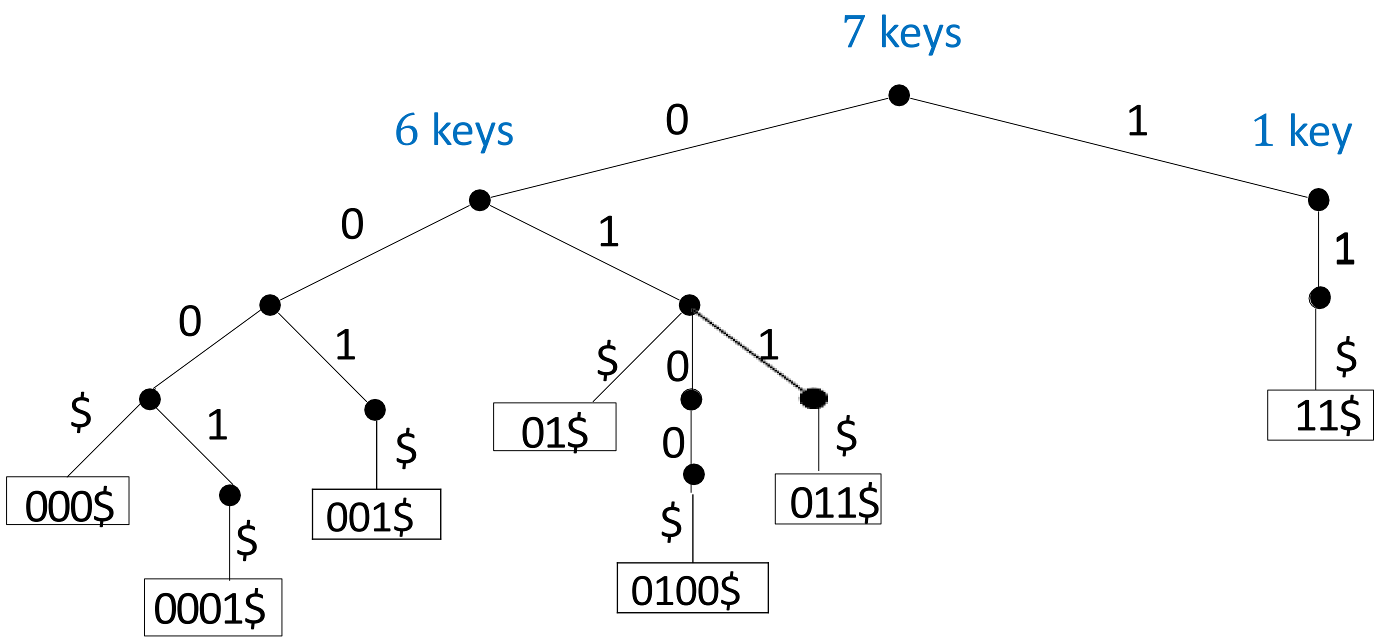
- Worst case space is $\Theta(n \cdot \text{maximum word length})$
- How to save space?

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard Trie
    - Pruned Trie
    - Compressed Trie
    - Multiway Trie

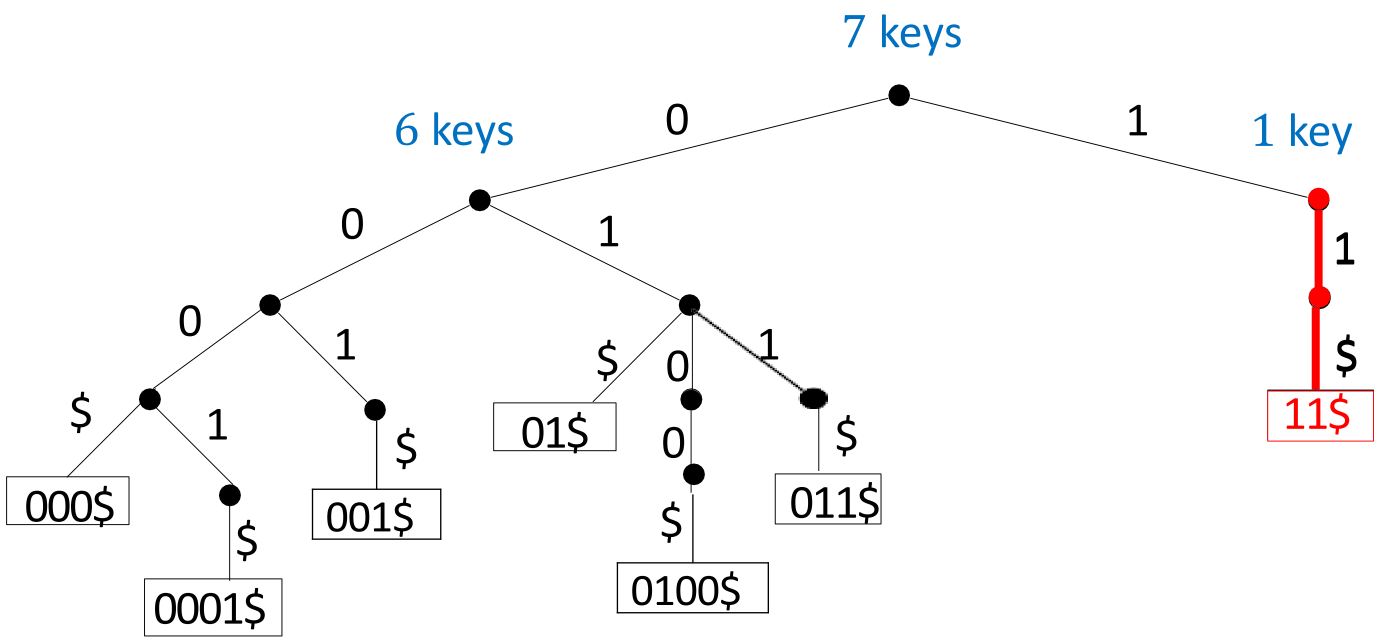
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



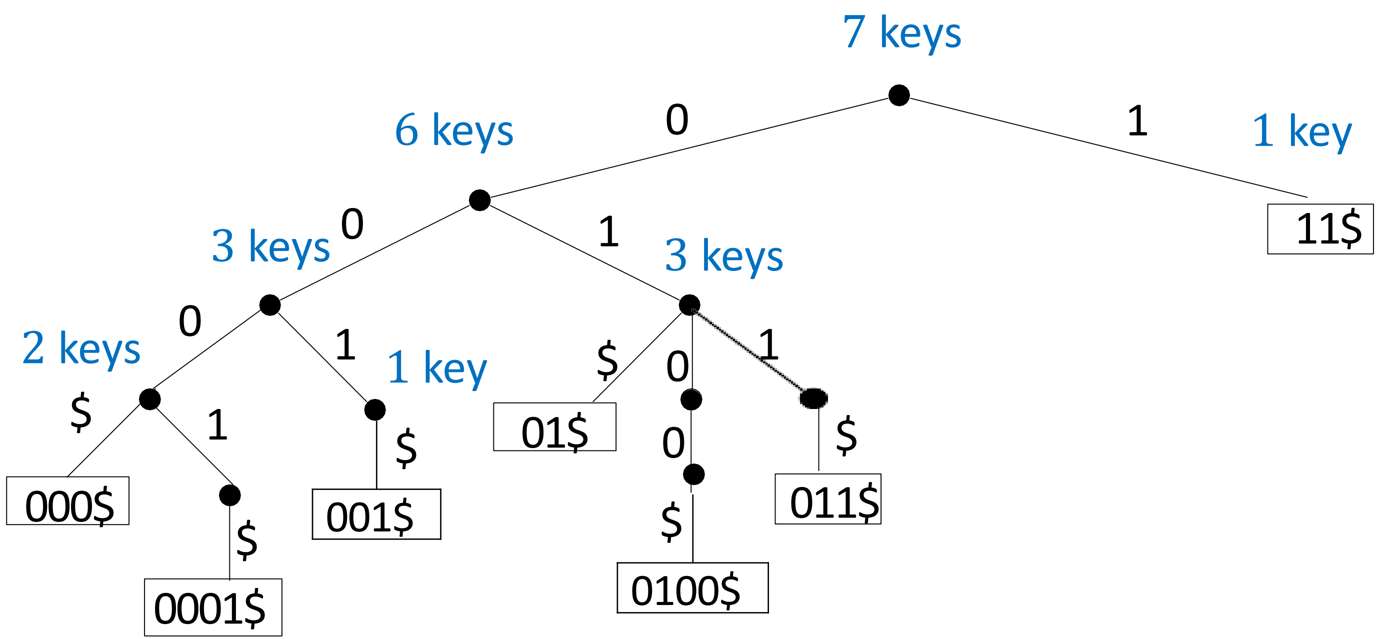
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



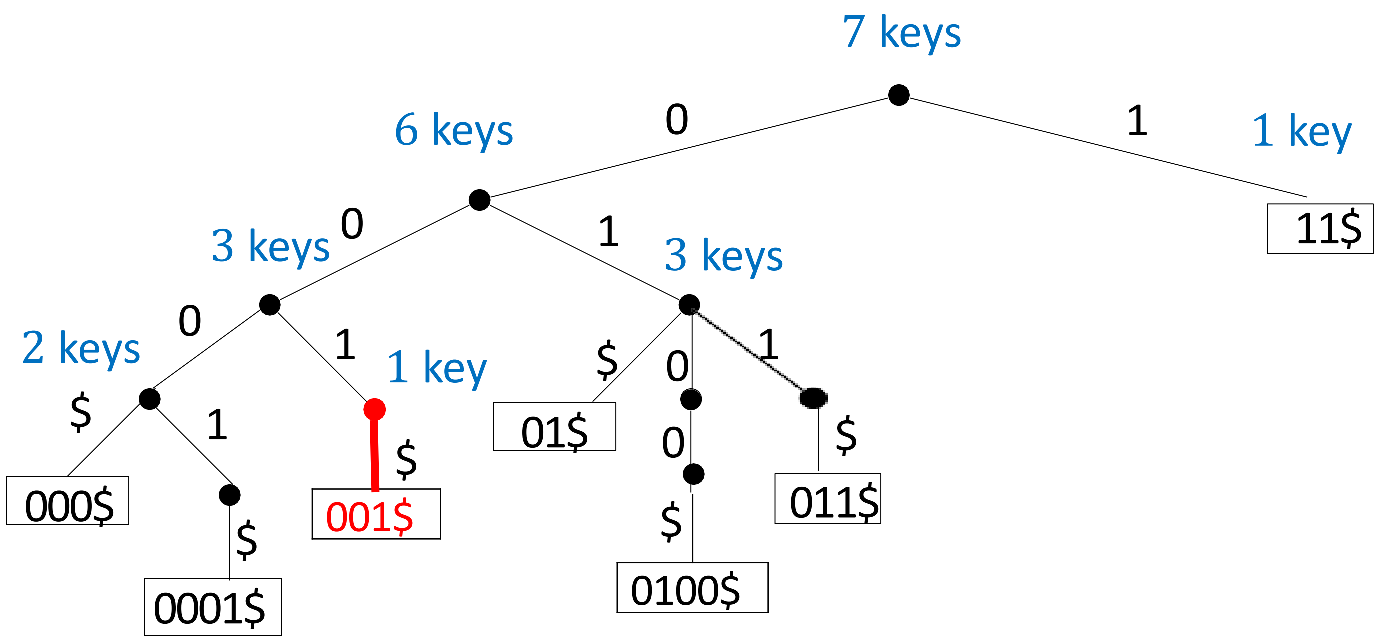
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



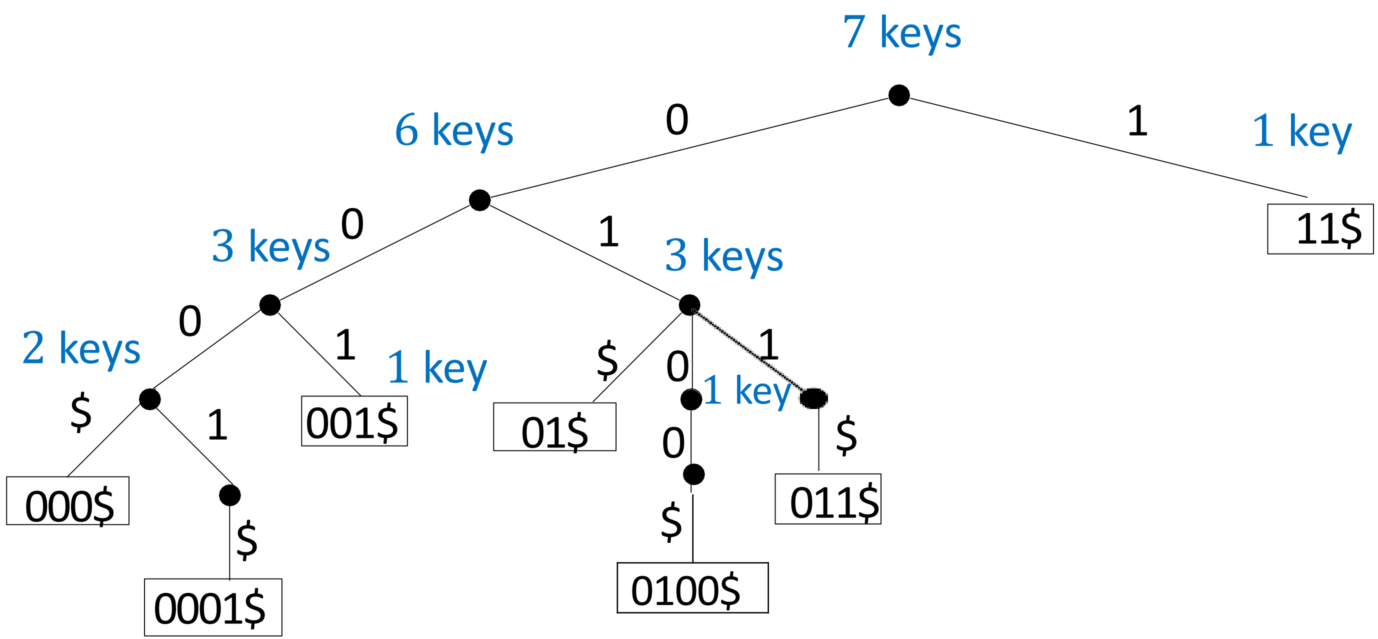
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



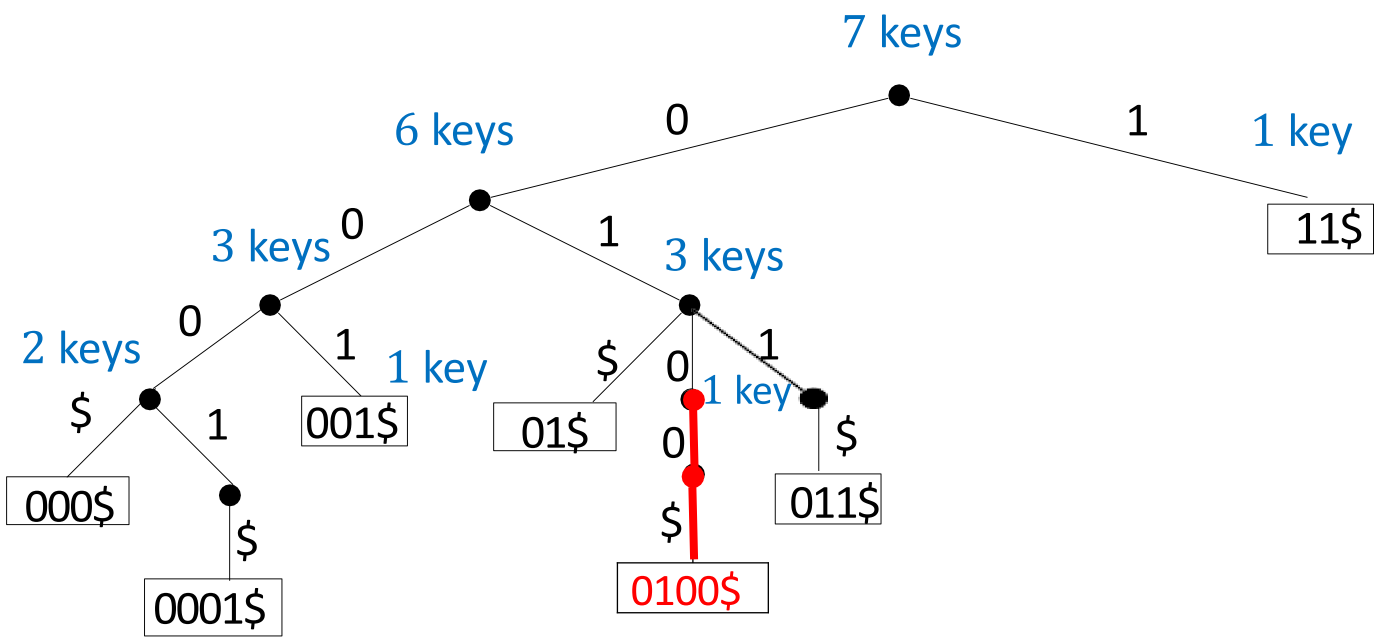
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



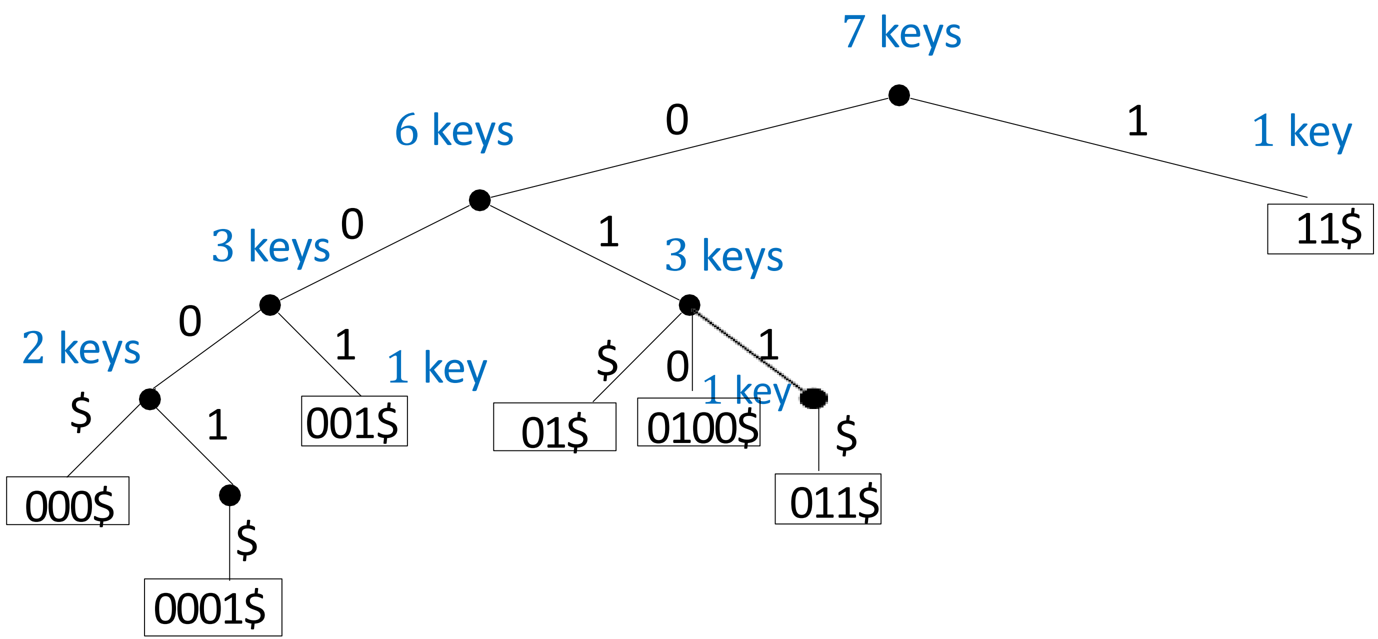
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



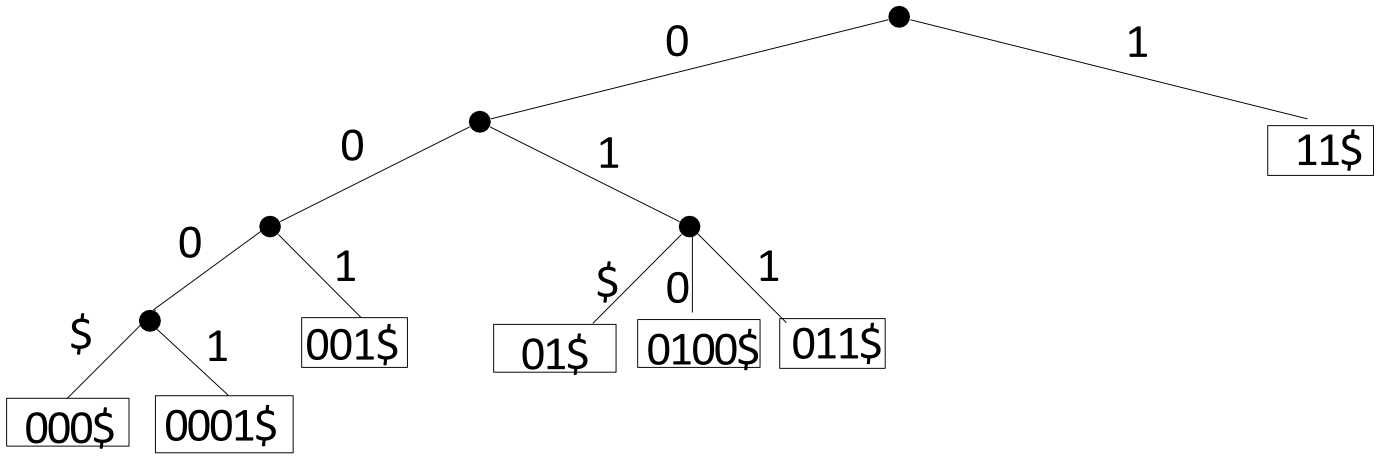
Pruned Trie

- Sub-trie with one key has only one node
- Convert standard trie into pruned trie



Pruned Trie

- Sub-trie with one key has only one node
- Final pruned trie

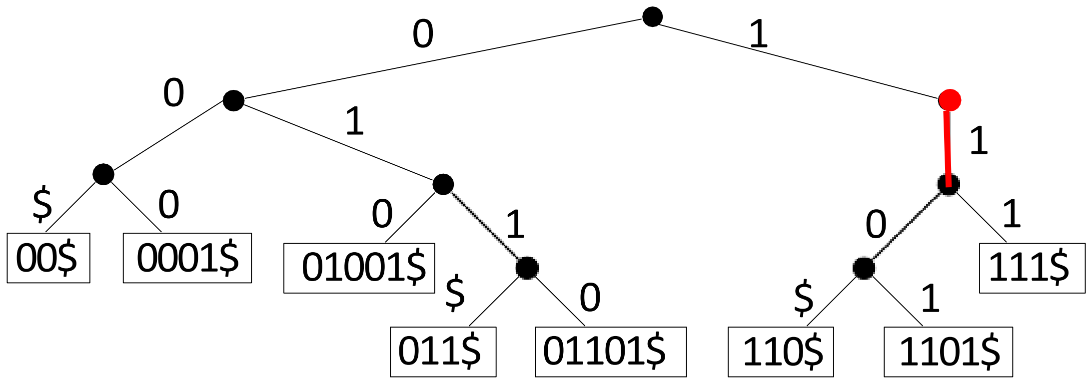


- node has a child only if it has at least two descendants
- saves space if there are only few bitstrings that are long
- can even store really long bitstrings more efficiently (real numbers)
- more efficient version of tries, but operations get a bit more complicated

Outline

- Lower bound for search
- Interpolation Search
- Tries
    - Standard Trie
    - Pruned Trie
    - Compressed Trie
    - Multiway Trie

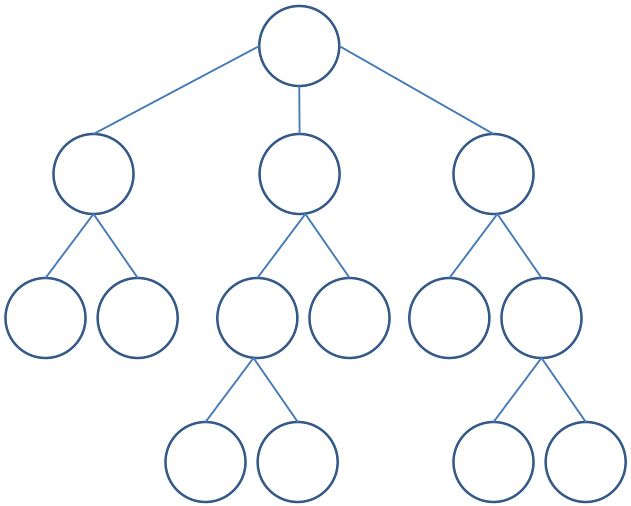
Pruned Trie: Internal Nodes with One Child



- Pruned trie can have internal nodes with one child
- Such ‘chains’ in a trie waste space and reduce search/insert/delete efficiency
- If we compress chains into one node, each internal node will have at least 2 children
- Let n be the number of leaf nodes (also equal to the number of stored keys)
- Can show if each internal node has at least 2 children, then there are at most $n-1$ internal nodes
- Therefore at most $2n-1$ total nodes
  - no wasted space, i.e. space is $O(n)$

Tree with no ‘chains’ Theorem

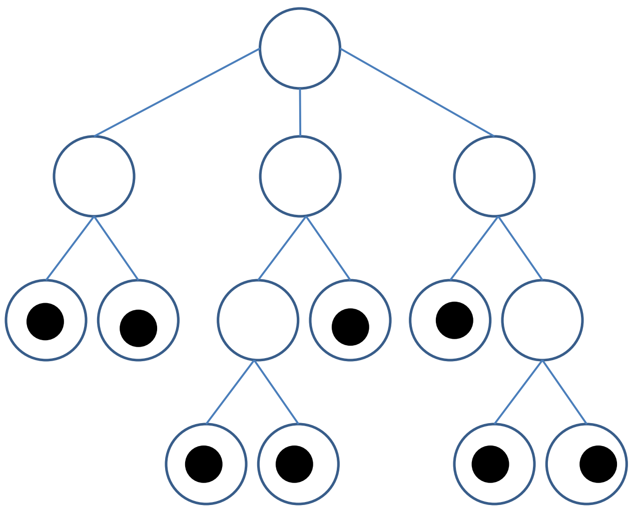
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



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

Tree with no 'chains' Theorem

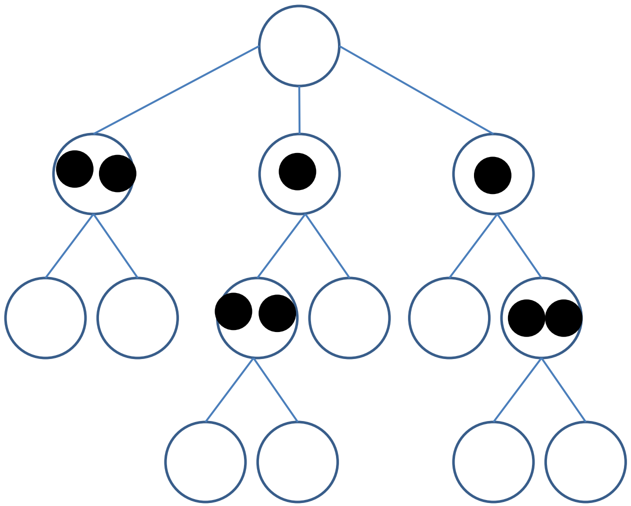
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



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

Tree with no ‘chains’ Theorem

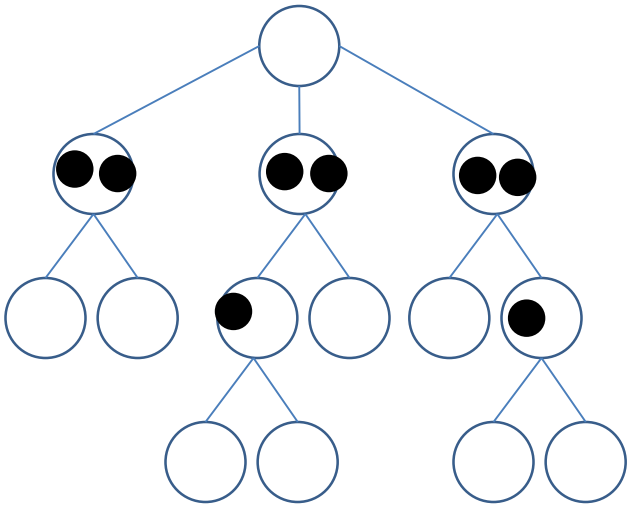
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- 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

Tree with no 'chains' Theorem

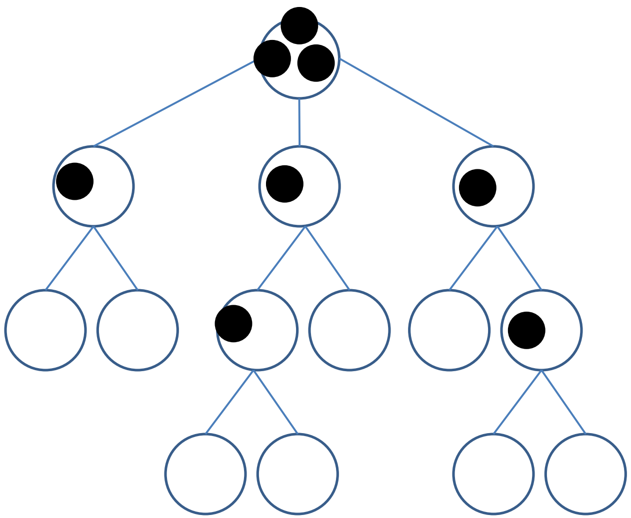
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



- 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

Tree with no ‘chains’ Theorem

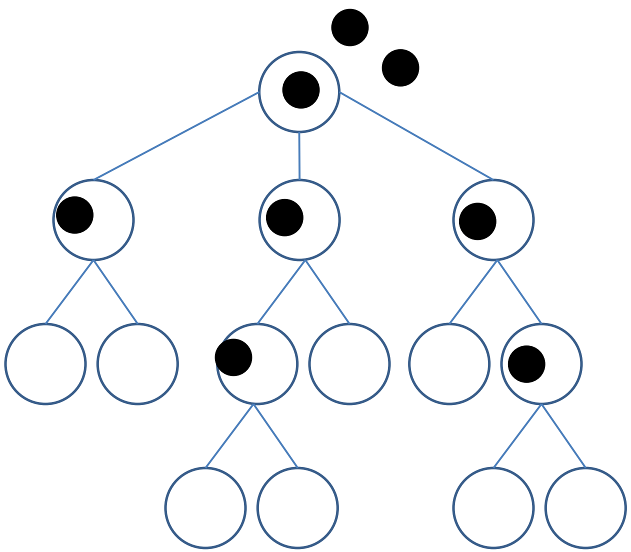
- Let T be a tree with m leafs. If every non-leaf (internal) node has at least 2 children, then the tree has at most $m - 1$ internal nodes



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

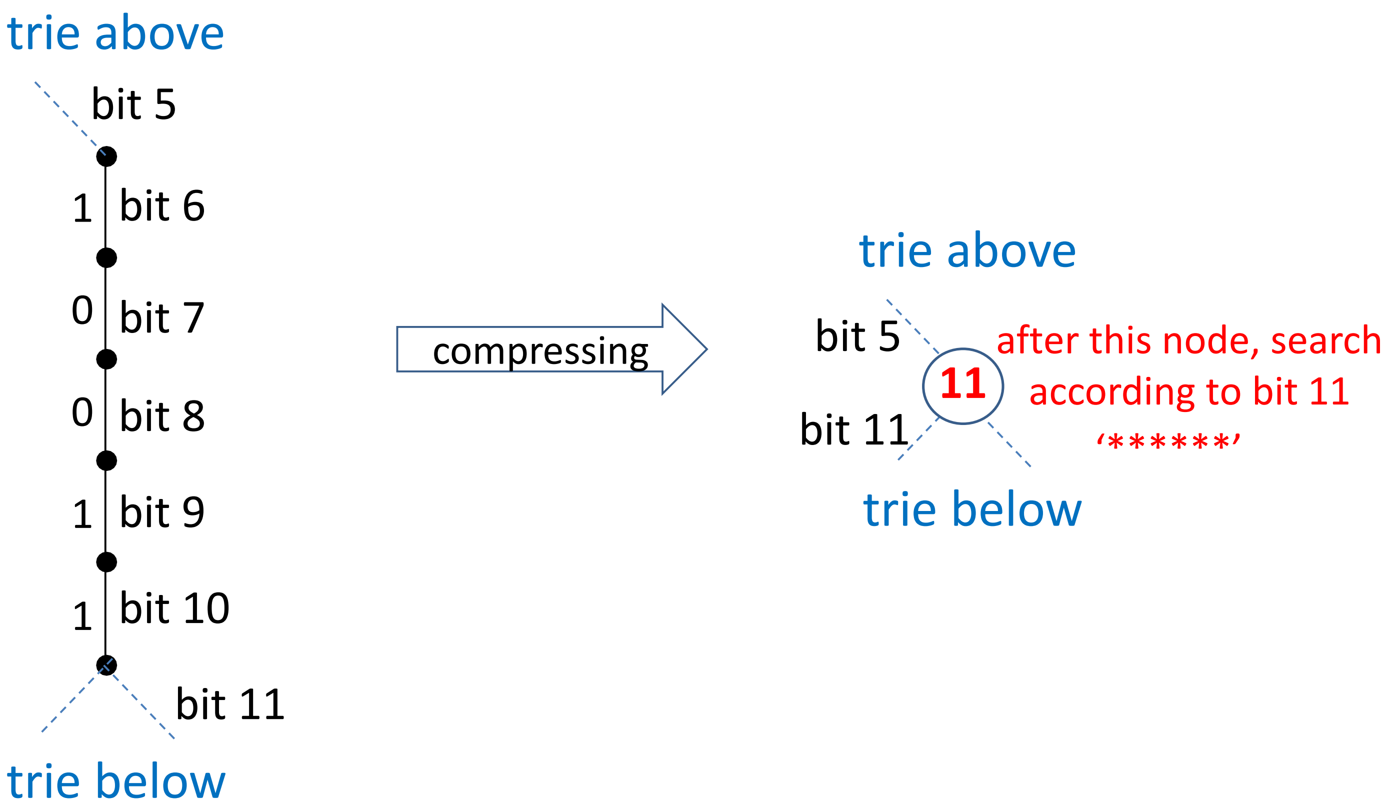
Tree with no ‘chains’ Theorem

- 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



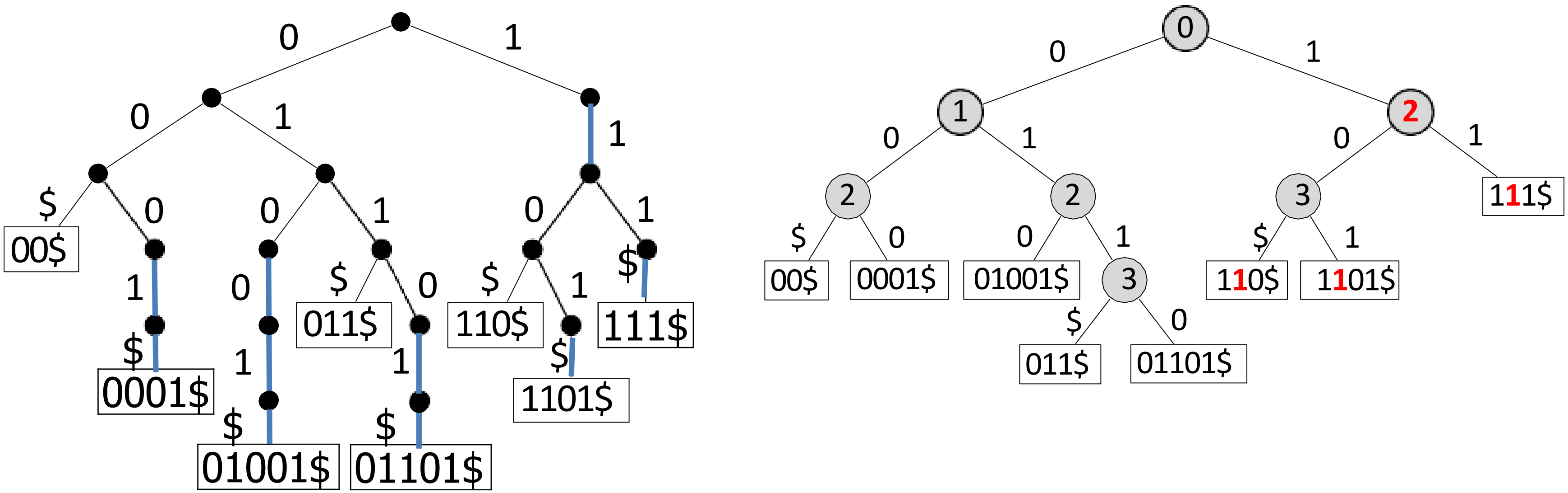
- Visual proof:
  - continue until reach the root
  - now each internal node has 1 stone and root has 2 or more stones
  - root leaves 1 stone and throws the rest outside the tree
  - now each internal node has 1 stone, and there is one or more stones outside the tree
  - since number of stones is m,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

Compressing Chains



- But now we lost part of the binary string '10011'
- Check if the leaf we reach stores the search key

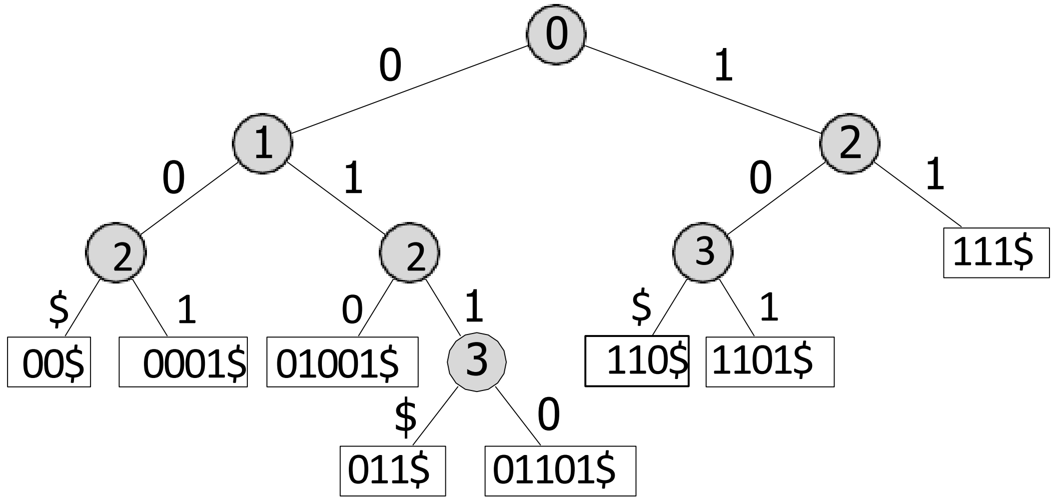
Compressed Tries (Patricia Tries)



- 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

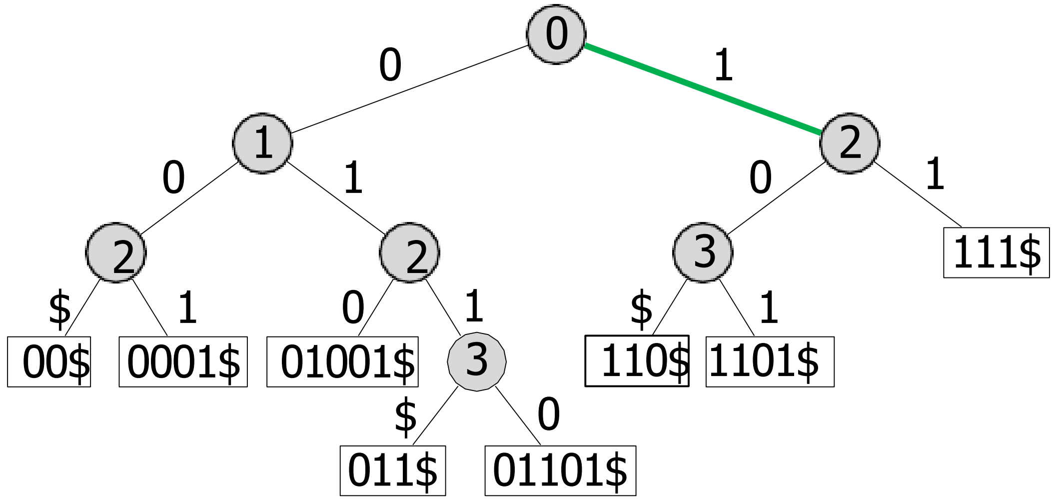
Compressed Tries: Search Example

Example: Search(10$)



Compressed Tries: Search Example

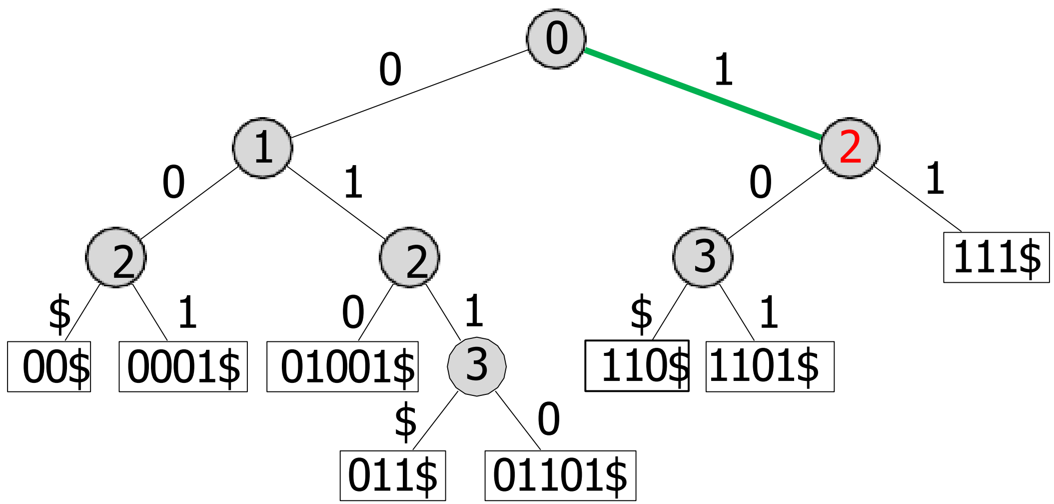
Example: Search(10$)



Compressed Tries: Search Example

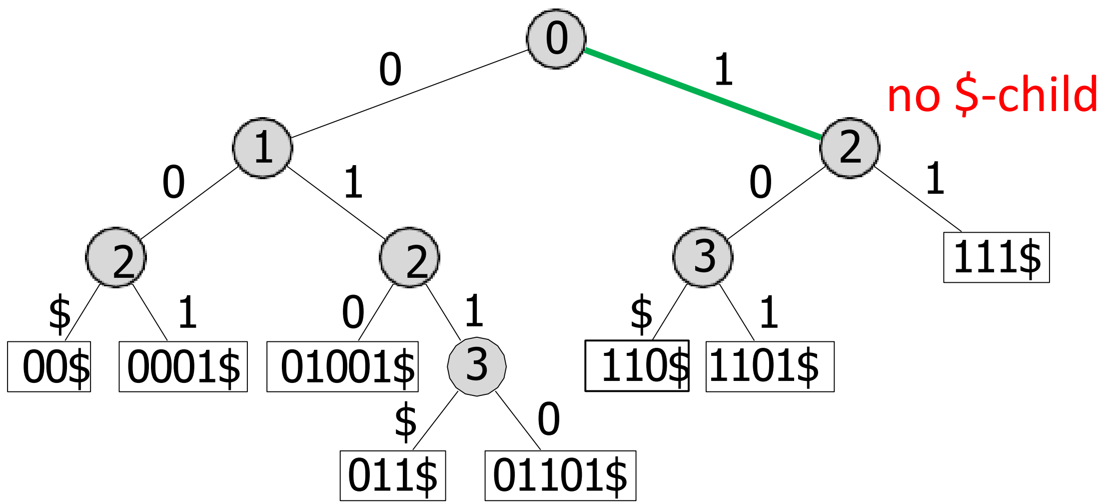
Example: Search(10$)

skip



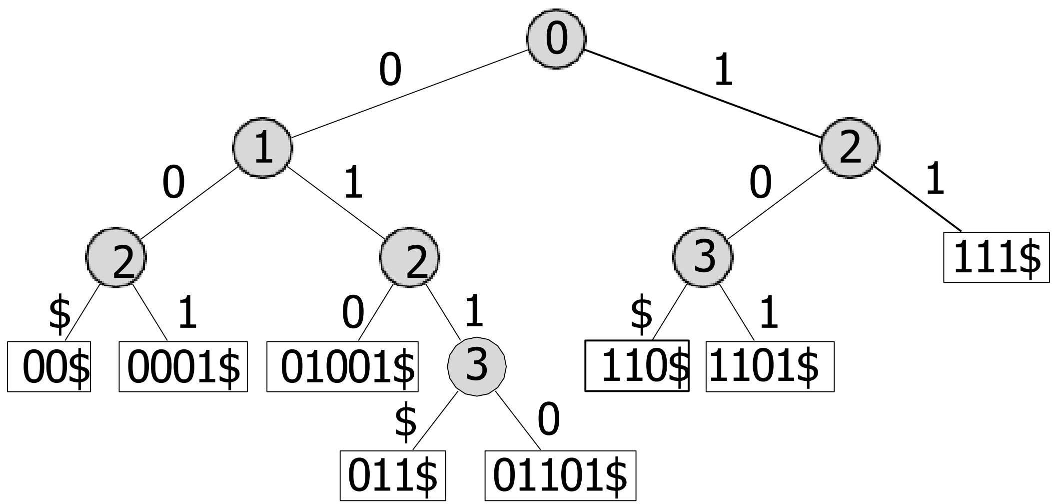
Compressed Tries: Search Example

Example: Search(10$) unsuccessful



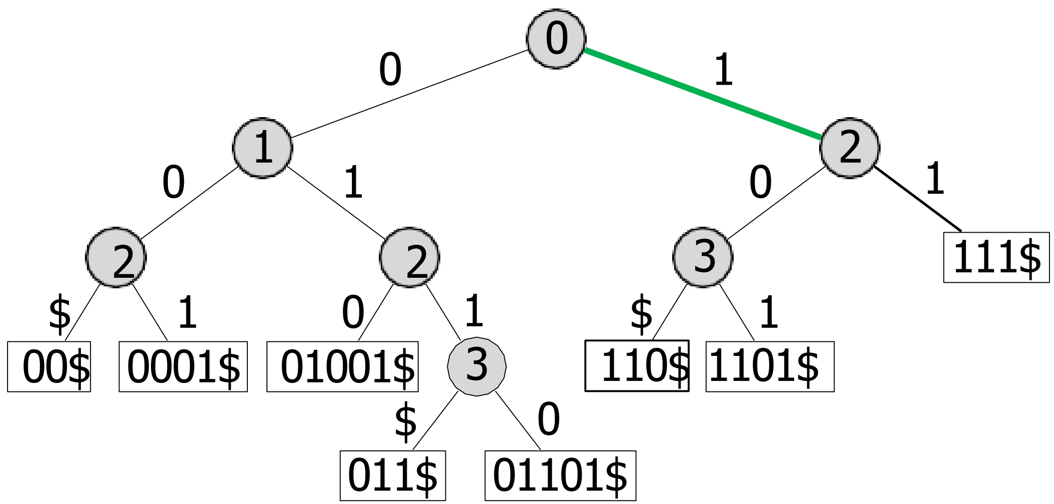
Compressed Tries: Search Example

Example: Search(101$)



Compressed Tries: Search Example

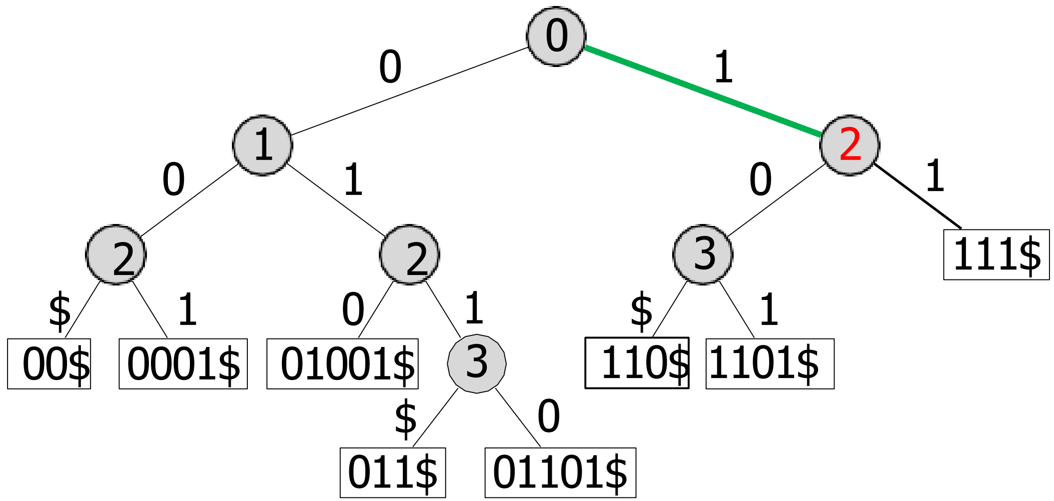
Example: Search(101$)



Compressed Tries: Search Example

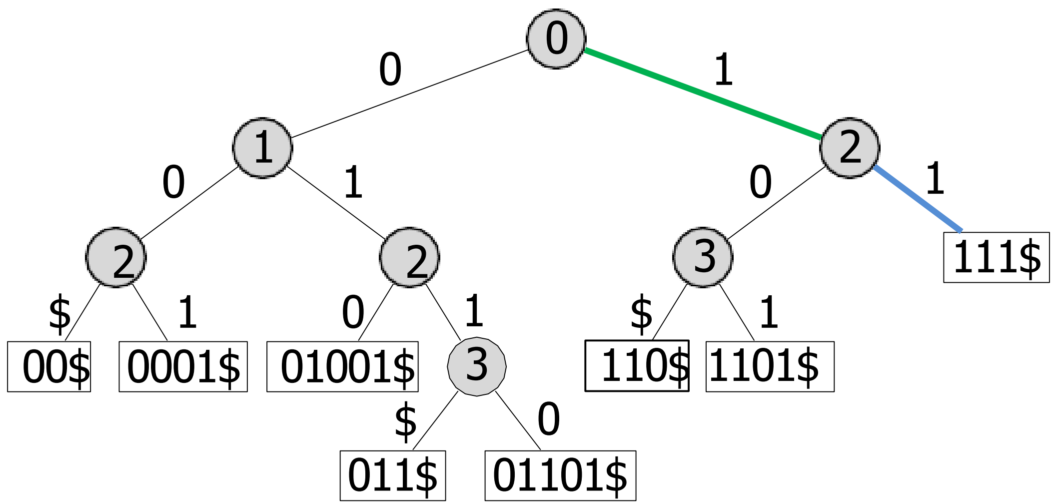
Example: Search(101$)

skip



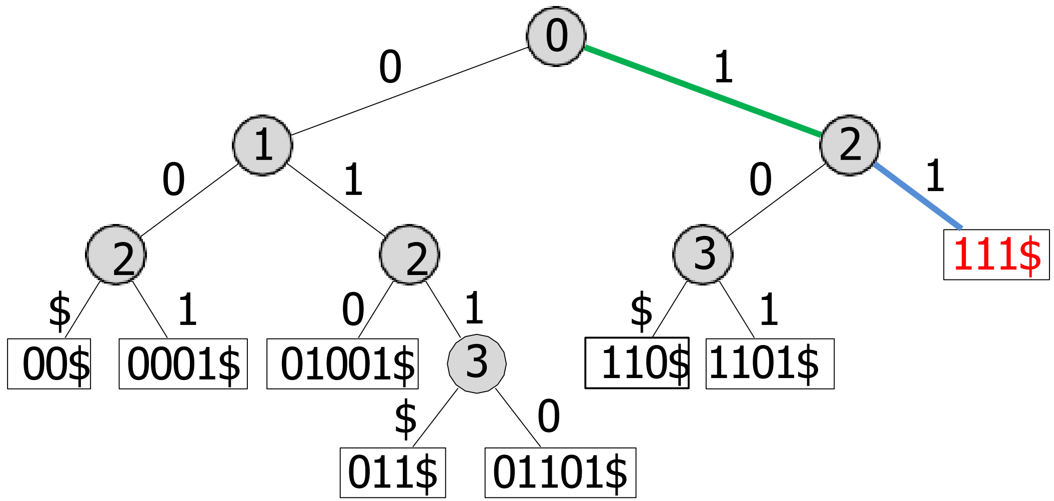
Compressed Tries: Search Example

Example: Search(101$)



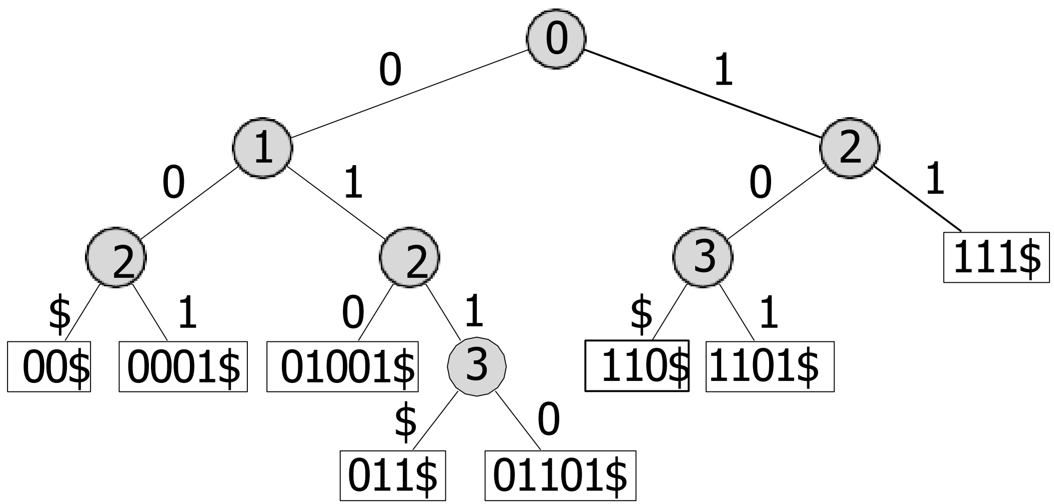
Compressed Tries: Search Example

Example: Search(101$) Unsuccessful (wrong word at the leaf)



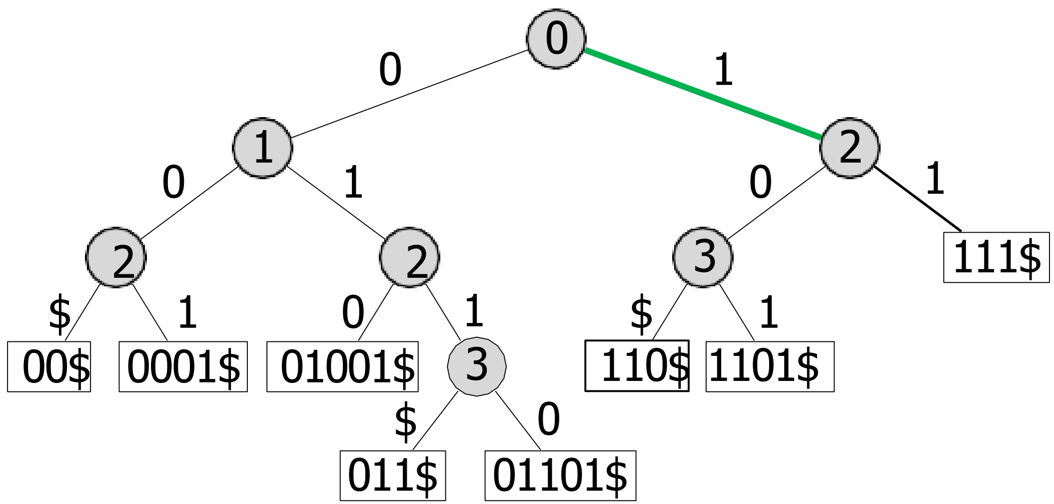
Compressed Tries: Search Example

Example: Search(111$)



Compressed Tries: Search Example

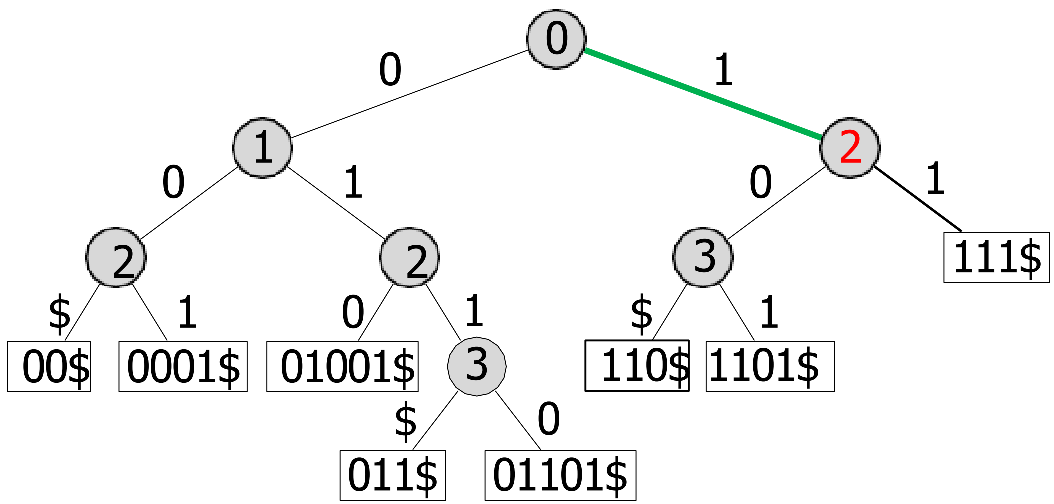
Example: Search(111$)



Compressed Tries: Search Example

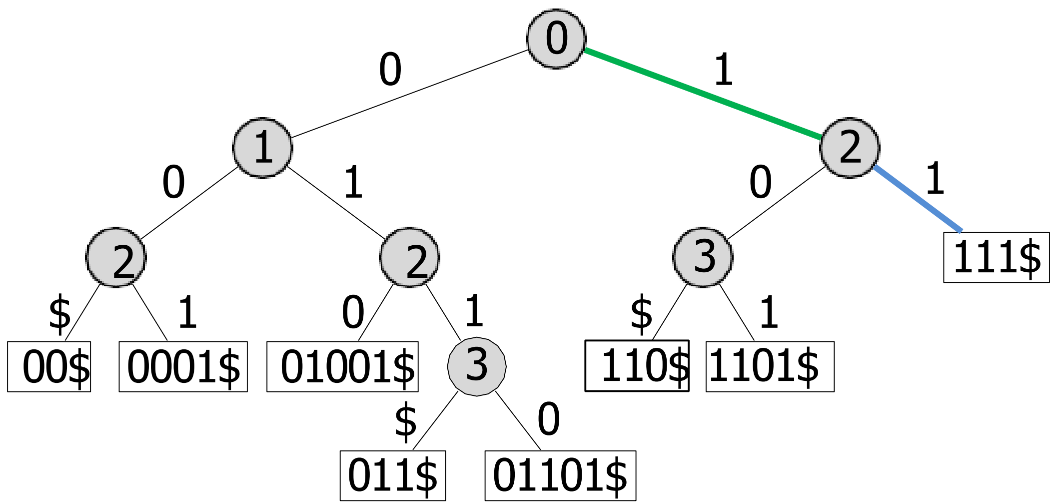
Example: Search(111$)

skip



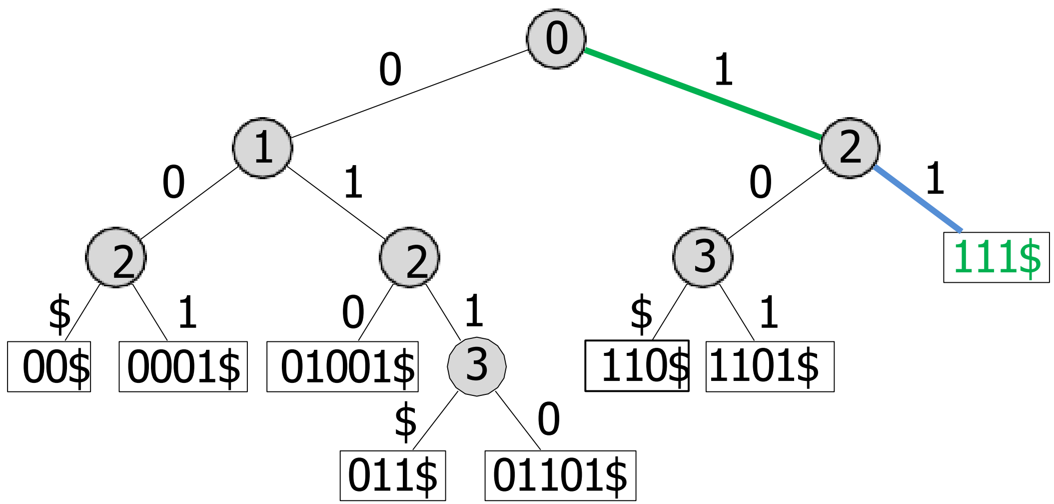
Compressed Tries: Search Example

Example: Search(111$)



Compressed Tries: Search Example

Example: Search(1<u>1</u>1$) successful (111$=111$)



Compressed Tries: Search

CompressedTrie::get-path-to(w)
$P \longleftarrow$ empty stack; $z \longleftarrow$ 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 $w[d]$
                $z \longleftarrow$ child at this link; P.*push*(z)
    else break
return P

CompressedTrie::search(w)
$P \longleftarrow$ *get-path-to*(w); $z \longleftarrow$ 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

- As in standard tries, follow links that correspond to current bits in w
- Main difference
    - stored indices say which bits to compare
    - also must compare w to the word found at the leaf

Compressed Tries: Summary

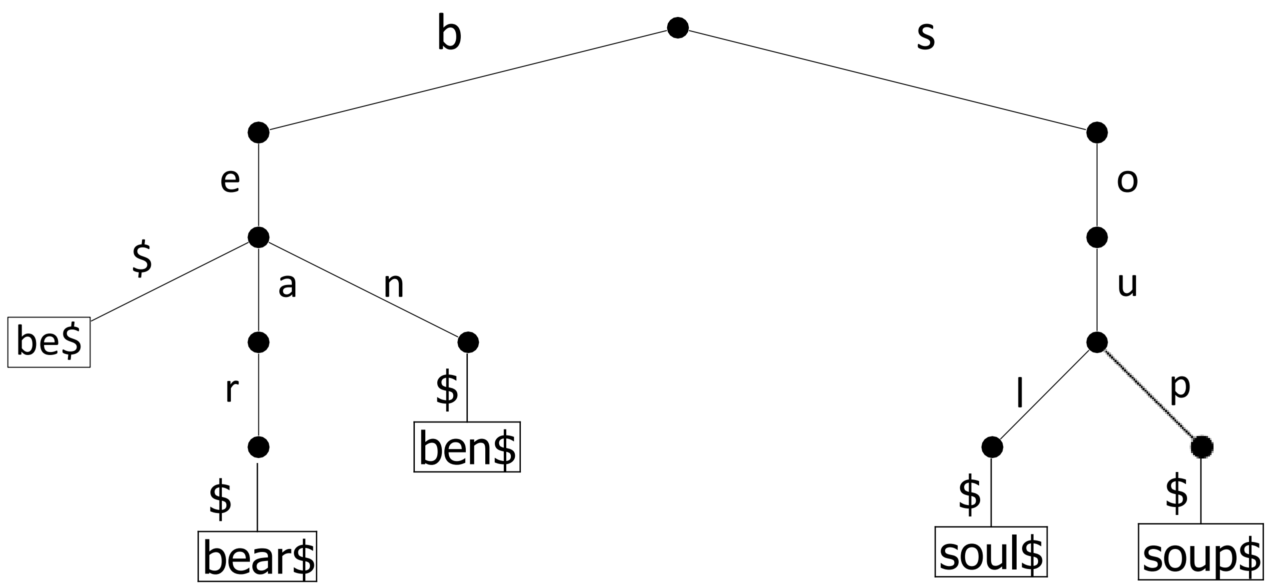
- *search*(w) and *prefix-search*(w) are easy
- *insert*(w) and *delete*(w) are conceptually simple
  - search for path P to word w (say we reach node z)
  - uncompress this path (using characters of $z.leaf$)
  - insert/delete w as in uncompressed trie
  - compress path from root to where changed happened
- All operations take $O(|w|)$ time for word w
- Compressed trie use $O(n)$ space
- Compressed tries are more complicated than standard tries, but space savings are worth it if words are unevenly distributed

Outline

- Lower bound for search
- Interpolation Search
- Tries
  - Standard Trie
  - Pruned Tries
  - Compressed Trie
  - Multiway Trie

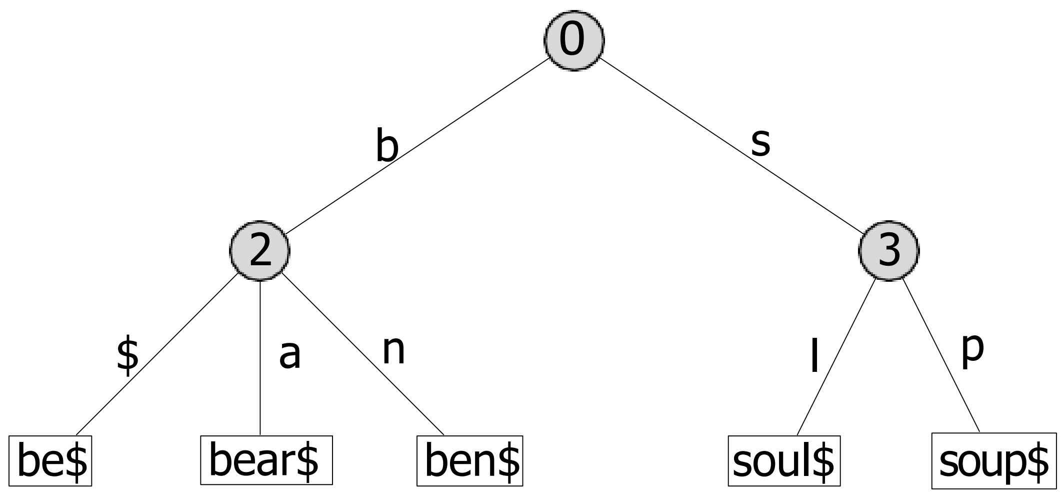
Multiway Tries: Larger Alphabet

- 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 $\{\text{bear\$}, \text{ben\$}, \text{be\$}, \text{soul\$}, \text{soup\$}\}$



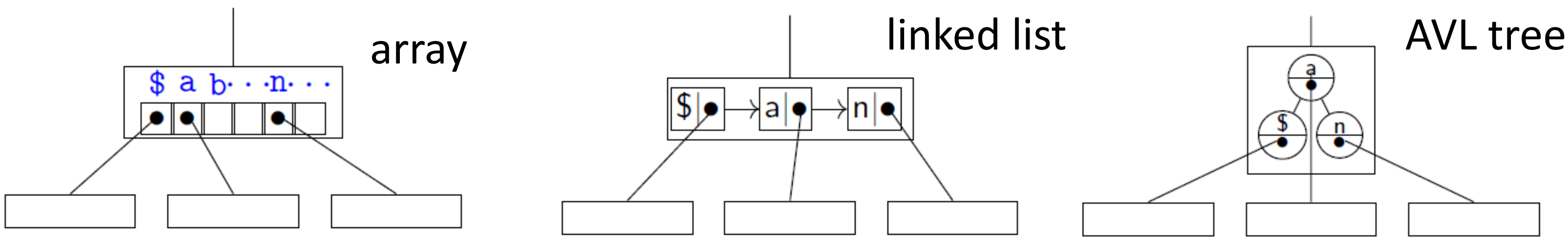
Compressed Multiway Tries

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



Multiway Tries: Summary

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



- Time/Space tradeoff: arrays are fast, lists are space efficient
- AVL tree is best in theory, but not worth it in practice unless $|\Sigma|$ is huge
- In practice, use hashing (next module)