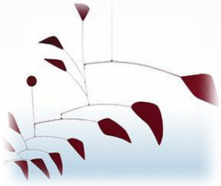
EECS 4101-5101
Advanced Data Structures



Shahin Kamali

Topic 7: String Data Structures

York University

Picture is from the cover of the textbook CLRS.

Objectives

- By the end of this module, you will be able to:
  - Explain dictionary abstract data types for maintaining a collection of strings.

Objectives

- By the end of this module, you will be able to:
  - Explain dictionary abstract data types for maintaining a collection of strings.
  - Describe basic data structures (e.g., tries, Patricia trees) for maintaining dictionaries of strings and explain how search, insert, delete operations are answered using them.

Objectives

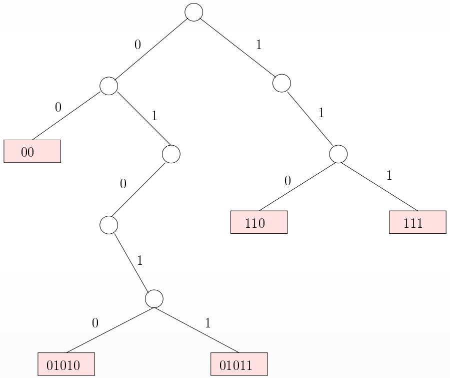
- By the end of this module, you will be able to:
  - Explain dictionary abstract data types for maintaining a collection of strings.
  - Describe basic data structures (e.g., tries, Patricia trees) for maintaining dictionaries of strings and explain how search, insert, delete operations are answered using them.
  - Explain Suffix trees and their application in answering pattern matching queries.

Tries

- **Trie**: A dictionary for binary strings
  - Items (keys) are stored **only** in the leaf nodes
  - A left child corresponds to a 0 bit
  - A right child corresponds to a 1 bit
- Keys can have different number of bits
- **prefix-free**: no key is a prefix of another key
- A **prefix** of a string $S[0..n-1]$:
  a substring $S[0..i]$ of S for some $0 \leq i \leq n-1$

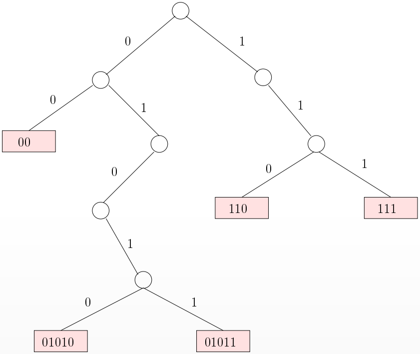
Tries

- Example: A trie for $S = \{00, 110, 111, 01010, 01011\}$



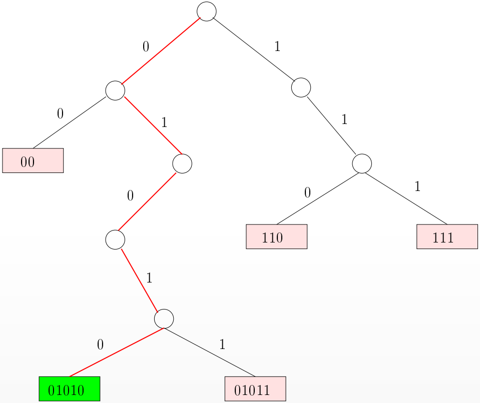
Tries: Search

- **Search**: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(01010)



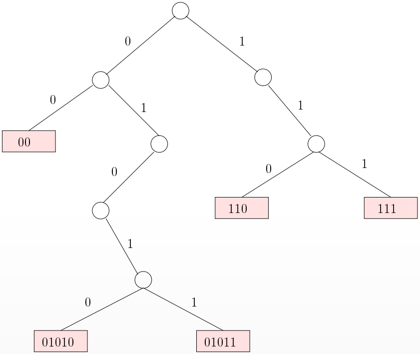
Tries: Search

- **Search**: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(01010) **successful**



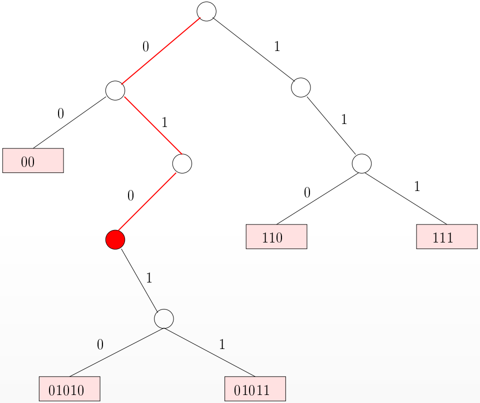
Tries: Search

- **Search**: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(0100)



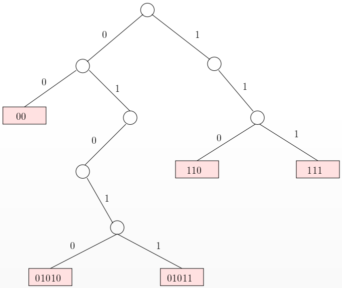
Tries: Search

- **Search**: start from the root, follow the relevant path using bitwise comparisons
- Example: Search(0100) **unsuccessful**



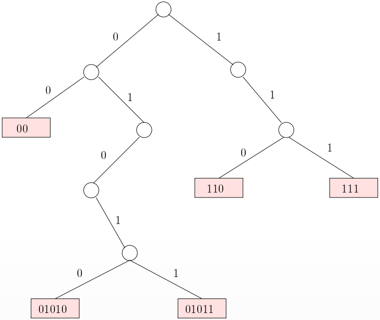
Tries: Insert

- **Insert(x): First search for x**
  - (a) If we finish at a leaf with key x, then x is already in trie: do nothing (e.g., when $x = 110$).
  - (b) If we finish at a leaf with a key $y \neq x$, then y is a prefix of x: not possible because our keys are **prefix-free** (e.g., $x = 1100$)
  - (c) If we finish at an internal node and there are no **extra bits**: not possible because our keys are **prefix-free** (e.g., when $x = 11$)
  - (d) If we finish at an internal node and there are extra bits: expand trie by adding necessary nodes that correspond to extra bits



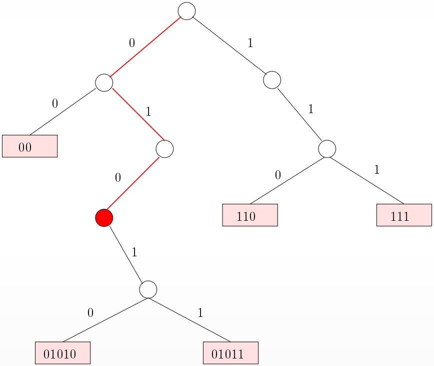
Tries: Insert

- **Insert(x): First search for x**
- Case (d) example: Insert(01000)



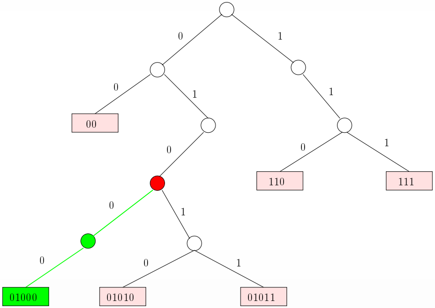
Tries: Insert

- **Insert(x): First search for x**
- Search(01000) **unsuccessful** Extra bits: 00



Tries: Insert

- **Insert(x): First search for x**

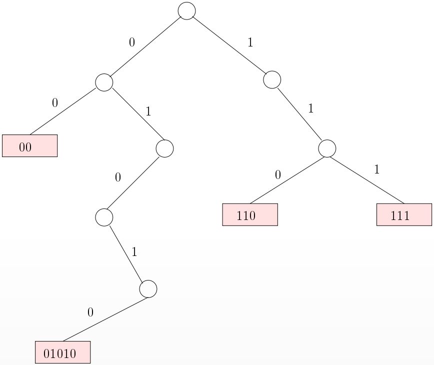


Tries: Delete

- **Delete(x)**
  - Search for x to find the leaf v_x
  - Delete v_x and all **ancestors** of v_x until we reach an ancestor that has two children

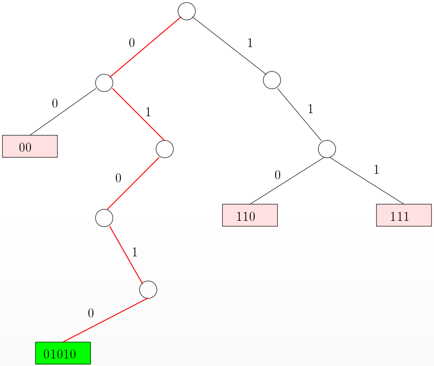
Tries: Delete

- **Delete(x)**
- Example: Delete(01010)



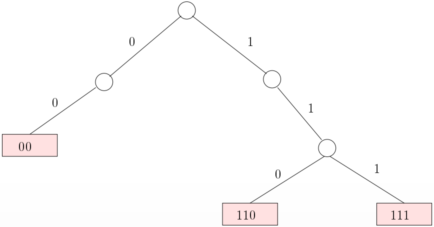
Tries: Delete

- **Delete(x)**
- Example: Delete(01010)
- Search(01010) **successful**



Tries: Delete

- **Delete(x)**
- Example: Delete(01010)



Tries: Operations

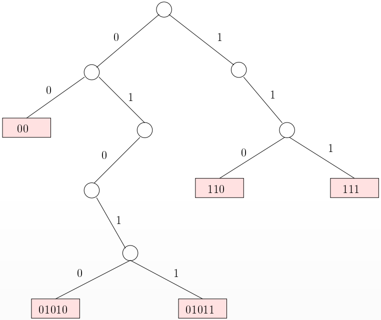
- Time Complexity of all operations (search, insert, delete) is $\Theta(|x|)$
  $|x|$: length of binary string x, i.e., the number of bits in x

Compressed Tries (Patricia Tries)

- **Patricia**: Practical Algorithm To Retrieve Information Coded in Alphanumeric (Introduced by Morrison (1968))
- Reduces **storage requirement**: eliminate nodes with only one child
- Every path of one-child nodes is compressed to a single edge
- Each node stores an index indicating the next bit to be tested during a search
- A compressed trie storing n keys always has $n-1$ internal (non-leaf) nodes

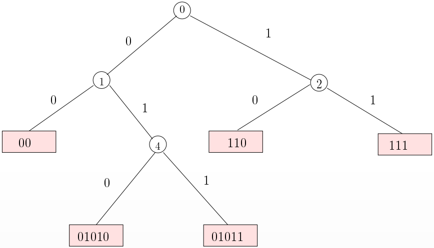
Compressed Tries (Patricia Tries)

- Each node stores an index indicating the next bit to be tested during a search
- Example: A trie



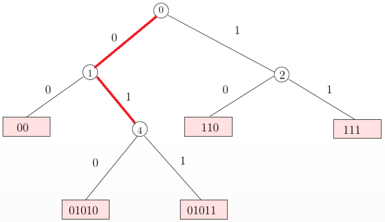
Compressed Tries (Patricia Tries)

- Each node stores an index indicating the next bit to be tested during a search
- Equivalent compressed trie



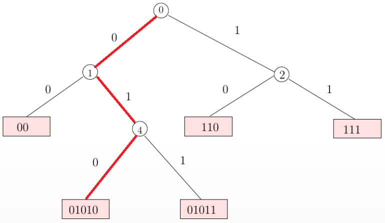
Compressed Tries: Operations

- Search(x):
  - Follow the proper path from the root down in the tree to a leaf
  - If search ends in an internal node, it is unsuccessful
    - E.g., search for **011**: we search index 0, go to the left, index 1, go to the right, and then there is no index 4; terminate!



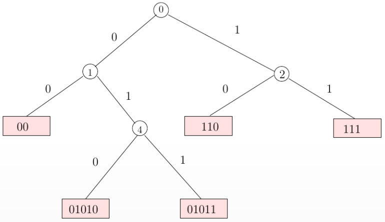
Compressed Tries: Operations

- Search(x):
  - Follow the proper path from the root down in the tree to a leaf
  - If search ends in an internal node, it is unsuccessful
    - E.g., search for **011**: we search index 0, go to the left, index 1, go to the right, and then there is no index 4; terminate!
  - In search ends in a leaf, we need to check again if the key stored at the leaf is indeed x.
    - e.g., search for **01110**; we end up in a leaf but the search is not successful!



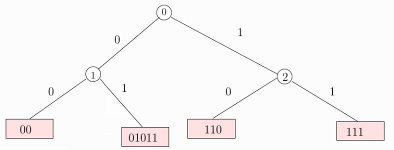
Compressed Tries: Operations

- Delete(x):
  - Perform Search(x) to find x in a leaf
  - If the search was successful, delete the leaf and its parent
    - E.g., delete **01010** from the trie



Compressed Tries: Operations

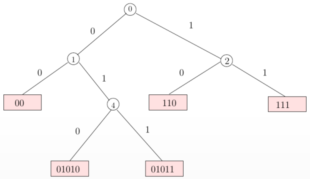
- Delete(x):
  - Perform Search(x) to find x in a leaf
  - If the search was successful, delete the leaf and its parent
    - E.g., delete **01010** from the trie



Compressed Tries: Operations

- Insert(x):

  .
  - Perform Search(x); If the search ends at a leaf L with key y, compare x and y to find the first index i where they disagree.
    - Then create a new node N with index i.
    - Insert N along the path from the root to L so that the parent of N has index $< i$ and one child of N is either L or an existing node on the path from the root to L that has index $> i$.
    - The other child of N will be a new leaf node containing x.
    - E.g., insert **01111**: create N with index 2, insert it between nodes with indices 1 and 4 on the path to the leaf

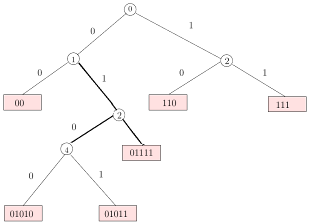


Compressed Tries: Operations

- Insert(x):

  .

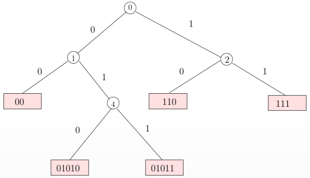
  - Perform Search(x); If the search ends at a leaf L with key y, compare x and y to find the first index i where they disagree.
    - Then create a new node N with index i.
    - Insert N along the path from the root to L so that the parent of N has index $< i$ and one child of N is either L or an existing node on the path from the root to L that has index $> i$.
    - The other child of N will be a new leaf node containing x.
    - E.g., insert **01111**: create N with index 2, insert it between nodes with indices 1 and 4 on the path to the leaf



Compressed Tries: Operations

- Insert(x):

  .
  - If the search ends at an internal node, we find the key corresponding to that internal node and proceed in a similar way to the previous case.
    - E.g., insert **0100**: create N with index 3, insert it between nodes with indices 1 and 4 on the path to the leaf

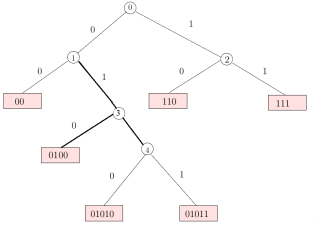


Compressed Tries: Operations

- Insert(x):

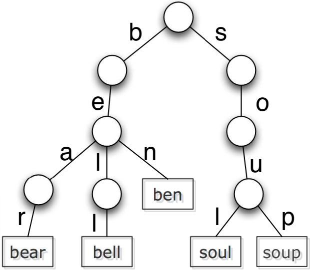
  .

  - If the search ends at an internal node, we find the key corresponding to that internal node and proceed in a similar way to the previous case.
    - E.g., insert **0100**: create N with index 3, insert it between nodes with indices 1 and 4 on the path to the leaf



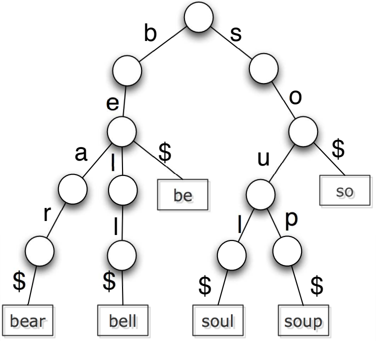
Multiway Tries

- To represent Strings over any fixed alphabet Σ
- Any node will have at most $|\Sigma|$ children
- Example: A trie holding strings {bear, bell, ben, soul, soup}



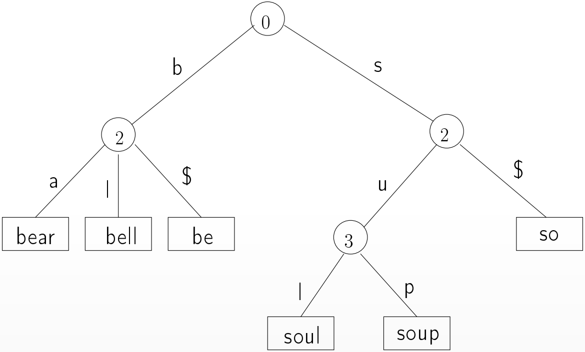
Multiway Tries

- Allow strings that are prefixes of other strings:
  Append a special end-of-word character, say \$, to all keys
- Example: A trie holding strings {bear, bell, be, so, soul, soup}



Multiway Tries

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



Pattern Matching

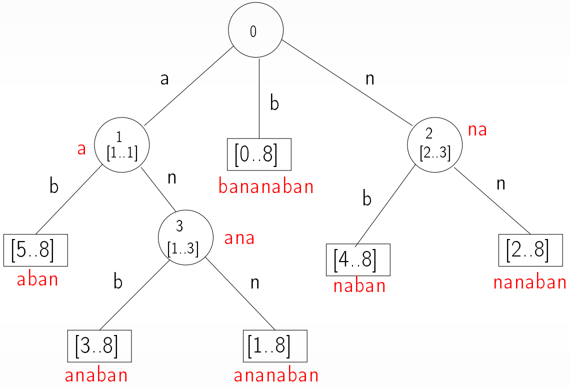
- Search for a the first occurrence of a **pattern** P in a large body of **text** T.
  - Example:
    - $T =$ "Where is he?"
    - $P_1 =$ "he"
    - $P_2 =$ "who"
- If P does not occur in T, return `FAIL`
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining

Suffix Trees

- A suffix of T:
  a substring $T[i..n-1]$ of T for some $0 \leq i \leq n-1$
  - Build a compressed trie that stores all suffixes of text T
  - Insert suffixes in decreasing order of length
  - If a suffix is a prefix of another suffix, we do not insert it
  - Store two indexes l, r on each node v (both internal nodes and leaves) where node v corresponds to substring $T[l..r]$

Suffix Trees: Example

T =bananaban

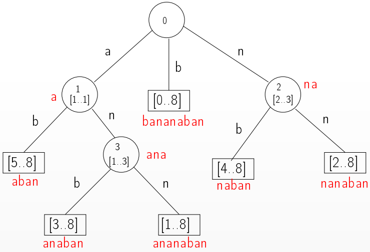


0	1	2	3	4	5	6	7	8
b	a	n	a	n	a	b	a	n

Suffix Trees: Pattern Matching

To search for pattern P of length m:

- Similar to Search in compressed trie with the difference that we are looking for a prefix match rather than a complete match
    - If we reach a leaf with a corresponding string length less than m, then search is unsuccessful (e.g., search for "abana")
    - Otherwise, we reach a node v (leaf or internal) with a corresponding string length of at least m. Then it suffices to check the first m characters of that string to see if there indeed is a match (e.g., search for "anab")



$T = bananaban$

String Data Structures Summary

- If you need to store a dictionary of multiple strings, use a compressed Patricia tree for better search, insert, delete time.

String Data Structures Summary

- If you need to store a dictionary of multiple strings, use a compressed Patricia tree for better search, insert, delete time.
- If you need to store a text T to support pattern-matching queries, maintain a Suffix tree of T.