String Matching: Tries

Madhavan Mukund

`https://www.cmi.ac.in/~madhavan`

Programming, Data Structures and Algorithms using Python

Week 10

Searching a fixed text

- String matching often involves searching a large fixed body of text
  - Collected works of Shakespeare
  - Comprehensive set of reference manuals
  - Genetic data

Searching a fixed text

- String matching often involves searching a large fixed body of text
  - Collected works of Shakespeare
  - Comprehensive set of reference manuals
  - Genetic data
- Make multiple queries on this text
  - Find the source of a famous quotation
  - Search for information on a part or a procedure
  - Search for a given gene sequence

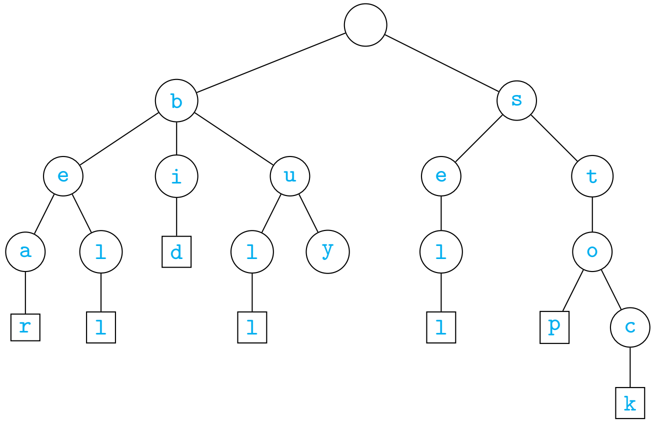
Searching a fixed text

- String matching often involves searching a large fixed body of text
  - Collected works of Shakespeare
  - Comprehensive set of reference manuals
  - Genetic data
- Make multiple queries on this text
  - Find the source of a famous quotation
  - Search for information on a part or a procedure
  - Search for a given gene sequence
- Preprocess the text to make the search efficient
  - Locate information about a pattern p of length m in time $O(m)$

Tries

- A trie is a special kind of tree
  - From "information retrieval"
  - Pronounced try, distinguish from tree

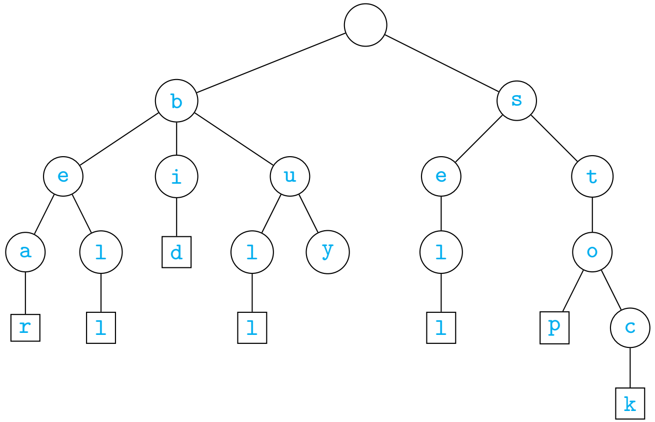
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- A trie is a special kind of tree
  - From "information retrieval"
  - Pronounced try, distinguish from tree
- Rooted tree
  - Other than root, each node labelled by a letter from Σ
  - Children of a node have distinct labels

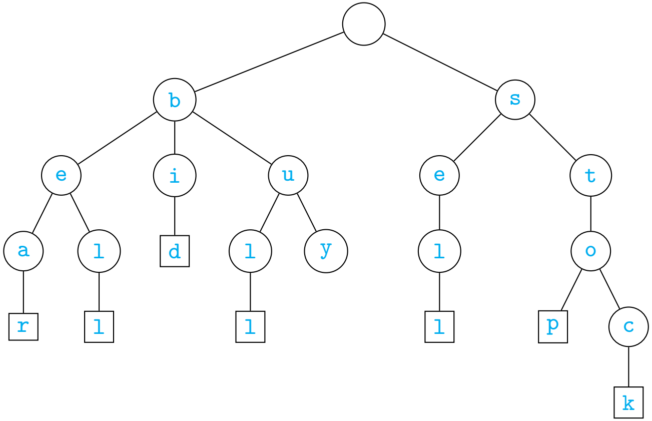
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- A trie is a special kind of tree
    - From "information retrieval"
    - Pronounced try, distinguish from tree
- Rooted tree
    - Other than root, each node labelled by a letter from Σ
    - Children of a node have distinct labels
- Each maximal path is a word
    - One word should not be a prefix of another
    - Add special end of word symbol \$

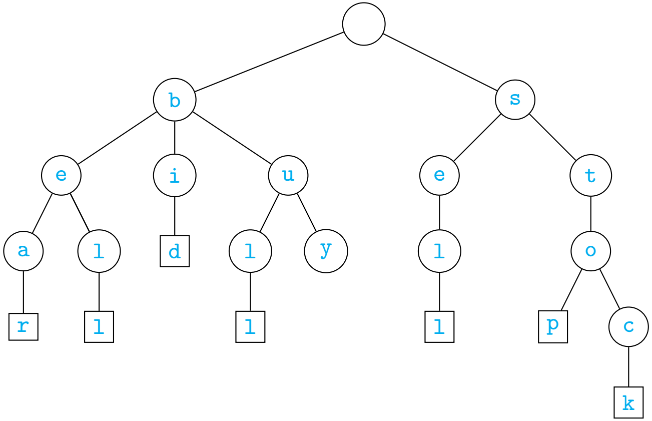
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols

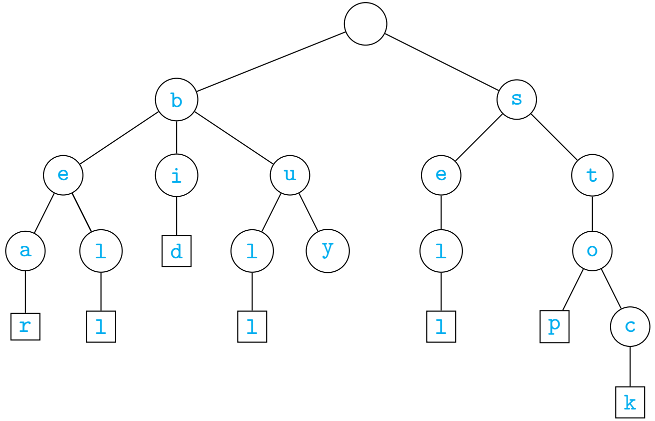
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path

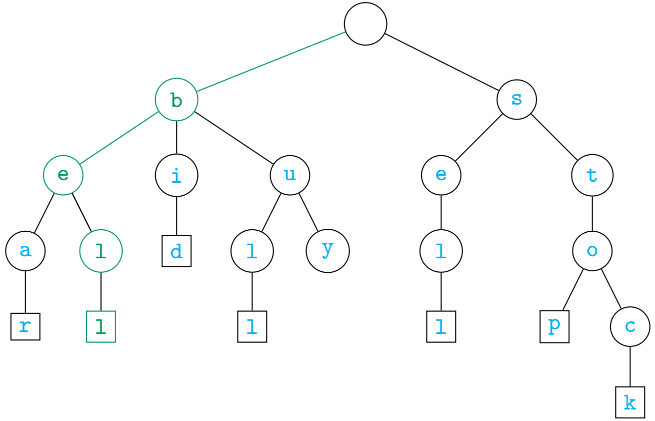
Search for `bell`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path
  - If the node we reach has \$ as a successor, $w \in S$

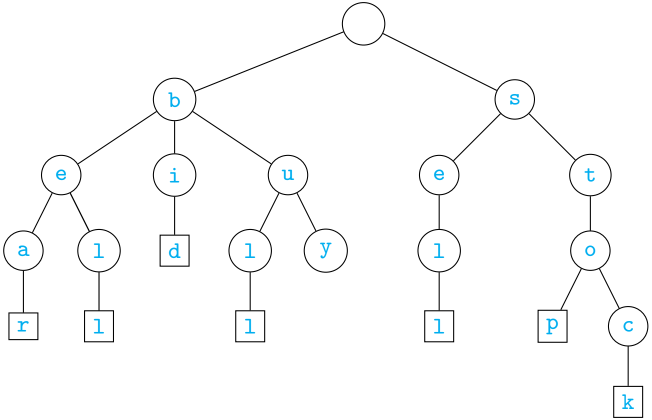
Search for `bell`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path
  - If the node we reach has $\$$ as a successor, $w \in S$
  - $w \notin S$ — path cannot be completed, or w is a prefix of some $w' \in S$

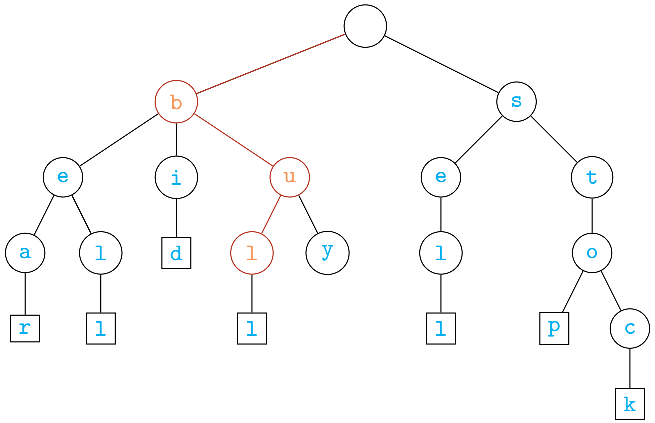
Search for `bulk`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path
  - If the node we reach has \$ as a successor, $w \in S$
  - $w \notin S$ — path cannot be completed, or w is a prefix of some $w' \in S$

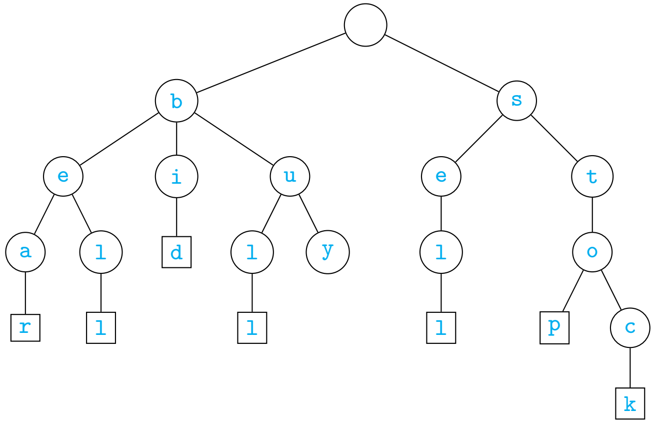
Search for `bulk`



Tries

- Build a trie T from a set of words S with s words and n total symbols

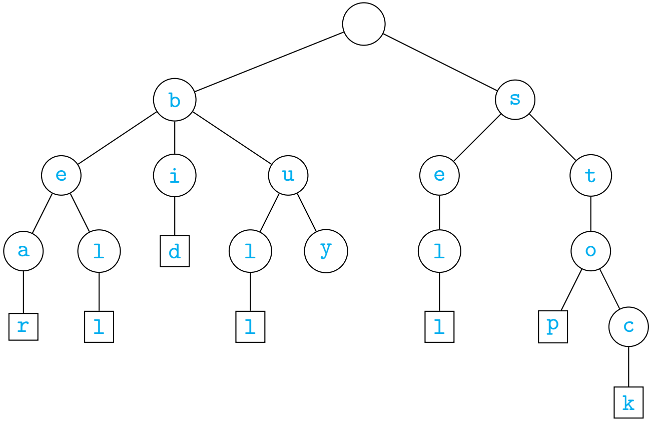
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols
- Basic properties for T built from S
  - Height of T is $\max_{w \in S} len(w)$
  - A node has at most $|\Sigma|$ children
  - The number of leaves in T is s
  - The number of nodes in T is $n + 1$, plus s nodes labelled $\$$

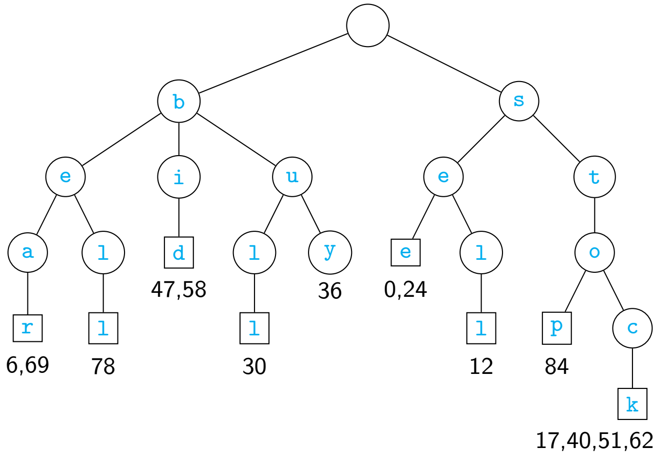
{bear,bell,bid,bull,buy,sell,stop,stock}



Auxiliary information

- Can maintain auxiliary information for each word
    - e.g., list of positions where the word occurs

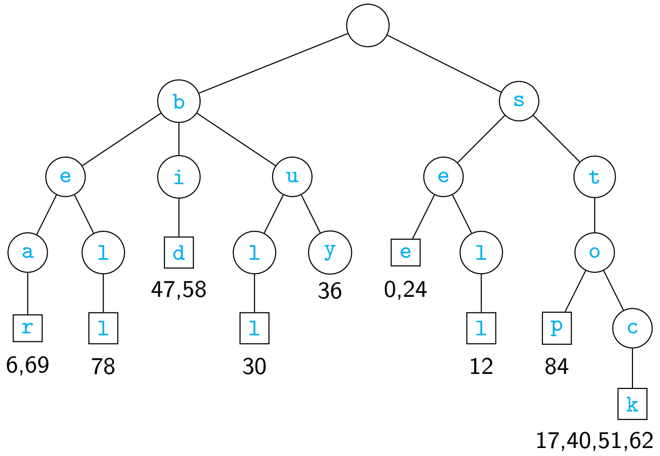
"see a bear? sell stock! see a bull? buy stock! bid stock! bid stock! bear the bell? stop!"



Auxiliary information

- Can maintain auxiliary information for each word
  - e.g., list of positions where the word occurs
- Trie as a key-value map
  - Keys are words in S
  - Values are relevant information about the word

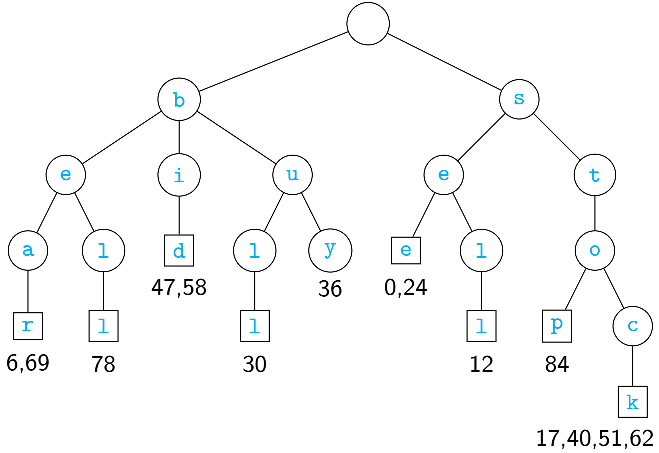
```
"see a bear?  sell stock!  see a bull?  buy stock!
bid stock!  bid stock!  bear the bell?  stop!"
```



Auxiliary information

- Can maintain auxiliary information for each word
  - e.g., list of positions where the word occurs
- Trie as a key-value map
  - Keys are words in S
  - Values are relevant information about the word
- Trie vs hash functions
  - Time to look up is proportional to length of key
  - No collisions in tries
  - Tries take up more space

```
"see a bear? sell stock! see a bull? buy stock!
bid stock! bid stock! bear the bell? stop!"
```



Trie: Implementation

- A Python `class` implementing tries

```
class Trie:

  def __init__(self,S=[]):
    self.root = {}
    for s in S:
      self.add(s)

  def add(self,s):
    curr = self.root
    s = s + "$"
    for c in s:
      if c not in curr.keys():
        curr[c] = {}
      curr = curr[c]
```

Trie: Implementation

- A Python `class` implementing tries
- `add` inserts a new word into the trie

```
class Trie:

  def __init__(self,S=[]):
    self.root = {}
    for s in S:
      self.add(s)

  def add(self,s):
    curr = self.root
    s = s + "$"
    for c in s:
      if c not in curr.keys():
        curr[c] = {}
      curr = curr[c]
```

Trie: Implementation

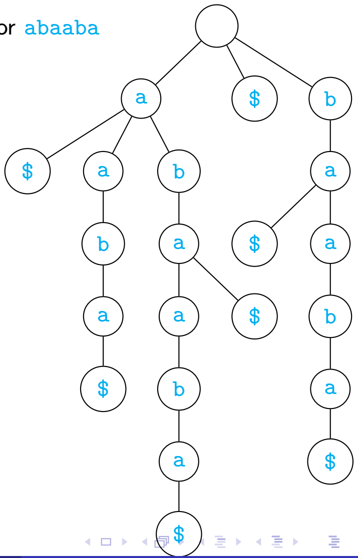
- A Python `class` implementing tries
- `add` inserts a new word into the trie
- `query` checks for a complete word
  - `True` — `s` is a complete word in `T`
  - `False` — `s` is not found in `T`
  - `None` — `s` is a prefix of some word in `T`

```
class Trie:

  def __init__(self,S=[]):
    self.root = {}
    for s in S:
      self.add(s)

  def add(self,s):
    ...

  def query(self,s):
    curr = self.root
    for c in s:
      if c not in curr.keys():
        return(False)
      curr = curr[c]
    if "$" in curr.keys():
      return(True)
    else:
      return(None)
```

Suffix tries

- Expand S to include all suffixes
  - For simplicity, assume $S = \{s\}$
  - $\mathit{suffix}(S) = \{w \mid \exists v, vw = s\}$

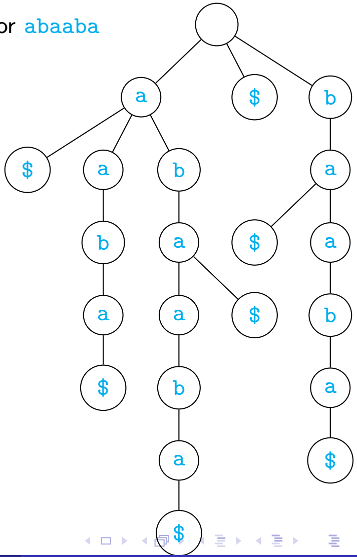
Suffix trie for abaaba



Suffix tries

- Expand S to include all suffixes
  - For simplicity, assume $S = \{s\}$
  - $suffix(S) = \{w \mid \exists v, vw = s\}$
- Build a trie for $suffix(S)$
  - Use $ to mark end of word
  - Suffix trie for S

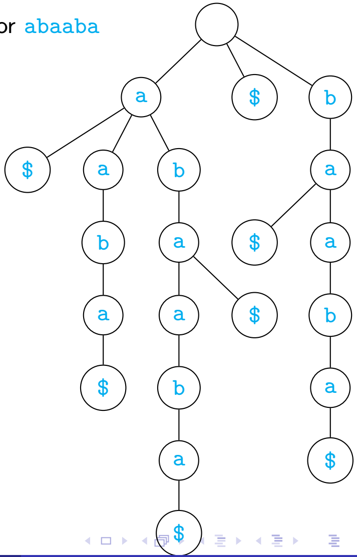
Suffix trie for abaaba



Suffix tries

- Expand S to include all suffixes
  - For simplicity, assume $S = \{s\}$
  - $suffix(S) = \{w \mid \exists v, vw = s\}$
- Build a trie for $suffix(S)$
  - Use \$ to mark end of word
  - Suffix trie for S
- Using a suffix trie we can answer the following
  - Is w a substring of s
  - How many times does w occur as a substring in s
  - What is the longest repeated substring in s

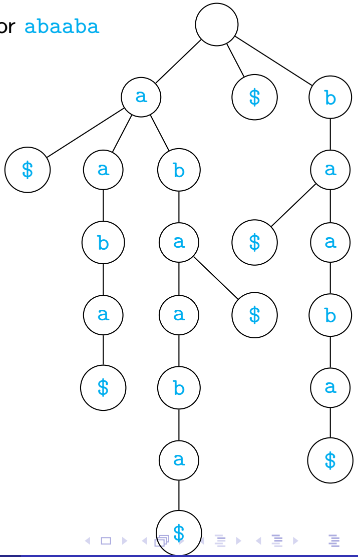
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
    - abaaba — yes, baabb — no

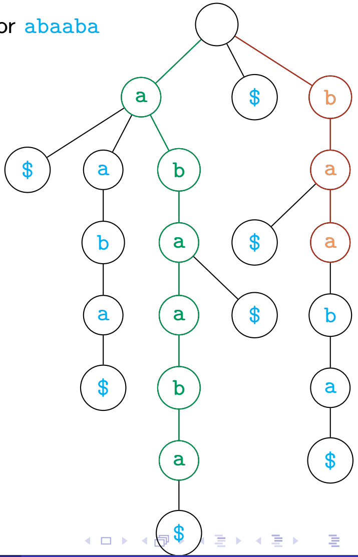
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
  - abaaba — yes, baabb — no

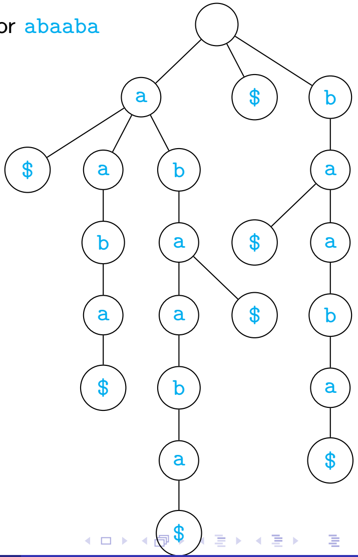
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s?
  - abaaba — yes, baabb — no
- Is w a suffix of s?
  - baa — no, aba — yes

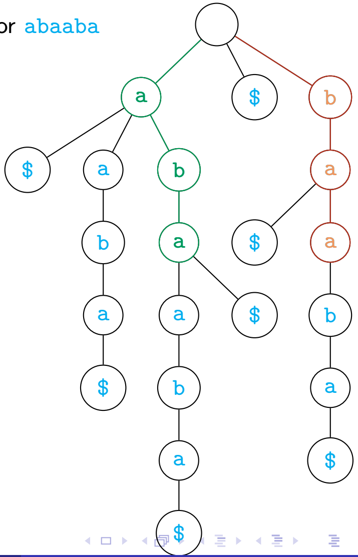
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
  - abaaba — yes, baabb — no
- Is *w* a suffix of *s*?
  - baa — no, aba — yes

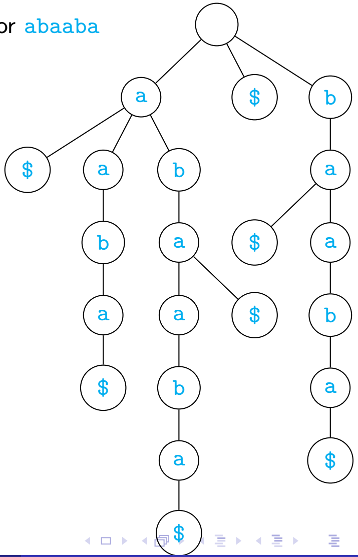
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
  - abaaba — yes, baabb — no
- Is *w* a suffix of *s*?
  - baa — no, aba — yes
- Number of times *w* occurs as a substring of *s*
  - aba — 2 occurrences
  - Number of leaves below the node

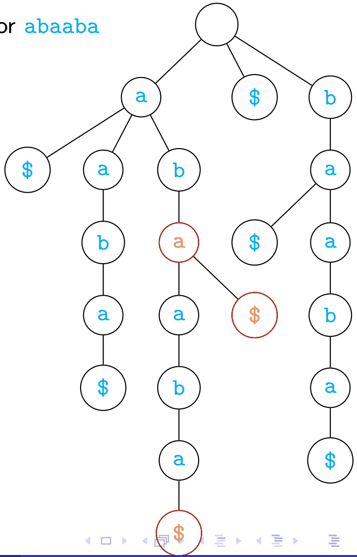
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
    - abaaba — yes, baabb — no
- Is *w* a suffix of *s*?
    - baa — no, aba — yes
- Number of times *w* occurs as a substring of *s*
    - aba — 2 occurrences
    - Number of leaves below the node

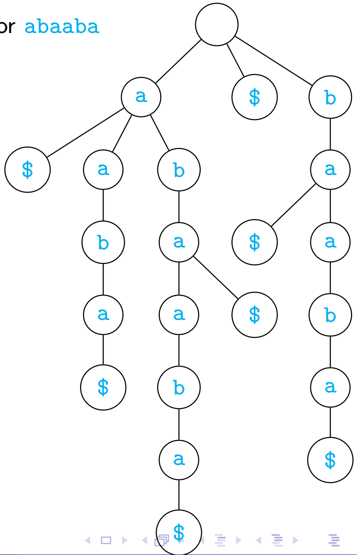
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
  - abaaba — yes, baabb — no
- Is *w* a suffix of *s*?
  - baa — no, aba — yes
- Number of times *w* occurs as a substring of *s*
  - aba — 2 occurrences
  - Number of leaves below the node
- Longest repeated substring of *s*
  - aba — 2 occurrences
  - Deepest node with more than one child

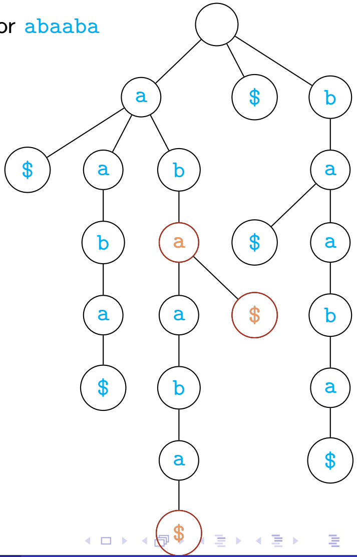
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
  - abaaba — yes, baabb — no
- Is *w* a suffix of *s*?
  - baa — no, aba — yes
- Number of times *w* occurs as a substring of *s*
  - aba — 2 occurrences
  - Number of leaves below the node
- Longest repeated substring of *s*
  - aba — 2 occurrences
  - Deepest node with more than one child

Suffix trie for abaaba



Suffix trie: Implementation

- Constructor builds a trie with every suffix of s

```
class SuffixTrie:

  def __init__(self,s):
    self.root = {}
    s = s + "$"
    for i in range(len(s)):
      curr = self.root
      for c in s[i:]:
        if c not in curr.keys():
          curr[c] = {}
        curr = curr[c]

  def followPath(self,s):
    curr = self.root
    for c in s:
      if c not in curr.keys():
        return(None)
      curr = curr[c]
    return(curr)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`
- `followPath` follows the path dictated by `s`
    - Return `None` if path fails
    - Return last node in the path if it succeeds

```
class SuffixTrie:

  def __init__(self,s):
    self.root = {}
    s = s + "$"
    for i in range(len(s)):
      curr = self.root
      for c in s[i:]:
        if c not in curr.keys():
          curr[c] = {}
        curr = curr[c]

  def followPath(self,s):
    curr = self.root
    for c in s:
      if c not in curr.keys():
        return(None)
      curr = curr[c]
    return(curr)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`
- `followPath` follows the path dictated by `s`
  - Return `None` if path fails
  - Return last node in the path if it succeeds
- If `followPath` finds a path, `s` is a valid substring

```
class SuffixTrie:

  def __init__(self,s):
    self.root = {}
    s = s + "$"
    for i in range(len(s)):
      curr = self.root
      for c in s[i:]:
        if c not in curr.keys():
          curr[c] = {}
        curr = curr[c]

  def followPath(self,s):
    ...

  def hasSubstring(self,s):
    return(self.followPath(s) is not None)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`
- `followPath` follows the path dictated by `s`
  - Return `None` if path fails
  - Return last node in the path if it succeeds
- If `followPath` finds a path, `s` is a valid substring
- If `followPath` ends in `$`, `s` is a suffix

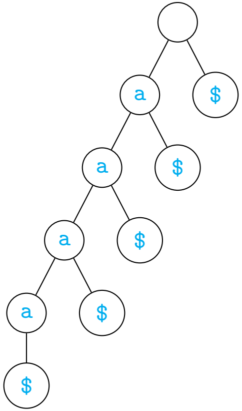
```
class SuffixTrie:

  def __init__(self,s):
    self.root = {}
    s = s + "$"
    for i in range(len(s)):
      curr = self.root
      for c in s[i:]:
        if c not in curr.keys():
          curr[c] = {}
        curr = curr[c]

  def followPath(self,s):
    ...

  def hasSuffix(self,s):
    node = self.followPath(s)
    return(node is not None and
                 "$" in node.keys())
```

Suffix trie: size

- How big can a suffix trie be for s of length n?

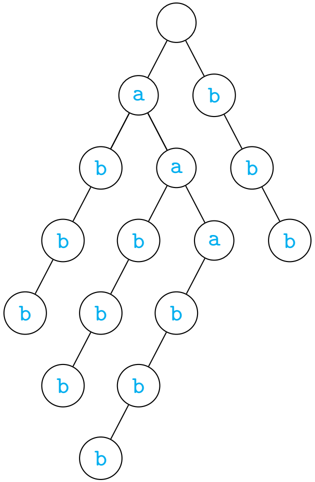
Suffix trie: size

- How big can a suffix trie be for s of length n?
- Number of nodes proportional to n?
  - Yes, a^n



Suffix trie: size

- How big can a suffix trie be for s of length n?
- Number of nodes proportional to n?
  - Yes, a^n
- Number of nodes proportional to n^2?
  - Yes, $a^n b^n$
  - $ nodes not shown



Summary

- Tries are useful to preprocess fixed text for multiple searches
- Searching for `p` is proportional to length of `p`
- Suffix tries allow us to make more expressive searches
- Main drawback of a trie is size