Tries
Outline

- Standard tries
- Compressed tries
- Suffix tries
- Huffman encoding tries
Where does “trie” come from?

- From the word retrieval
- Introduced in the 1960’s
Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP’s algorithm performs pattern matching in time proportional to the text size
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
  - Thus do better than $O(n+m)$ for text of size $n$ and pattern of size $m$!

- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A tries supports pattern matching queries in time proportional to the pattern size ($\sim O(m)$)!
Standard Trie

The standard trie for a set of strings $S$ is an ordered tree such that:

- Each node but the root is labeled with a character
- The children of a node are alphabetically ordered
- The paths from the external nodes to the root yield the strings of $S$

Example: standard trie for the set of strings $S = \{\text{bear, bell, bid, bull, buy, sell, stock, stop}\}$
Standard Trie

- What space does the trie use?
- What is the maximum height of the tree?
Standard Trie

A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:

- $n$ total size of the strings in $S$
- $m$ size of the (maximum) string parameter of the operation
- $d$ size of the alphabet
Standard Trie

- When is \( n \), and thus space, maximum?
  - When \( S \) consists of mutually unique words with no letters in common

- What type of word(s) produces the largest search time?
  - Short word? Long word?
  - Answer: long words, especially those whose prefix is very common
  - \( \rightarrow \) will do improvements later…
Word Matching with a Trie

We insert the words of the text into a trie.

Each leaf stores the occurrences of the associated word in the text.

![Trie Diagram]
Standard Trie Construction

How do you build it?
Standard Trie Construction

Assuming the input strings are words in the English language, how many children does the root node have?
Standard Trie Construction

The number of children of the root node equals to the maximum number of distinct first letters all the words in the input string

- 2 in this example, maximum of 26 in English
Standard Trie Construction

What does the tree for “bid” look like?
Standard Trie Construction

What does the tree for “bid” and “sell” look like?
Standard Trie Construction

What is the tree after adding "buy" and "stop"?
Standard Trie Construction

What is the tree after adding “buy” and “stop”?
Standard Trie Construction

What is the tree after adding “stock”? 

```
  b
 / 
u i
 / 
 y d
```
```
  s
 / 
 e t
 / 
 l o
 / 
 l p
```
Standard Trie Construction

What is the tree after adding “stock”?

![Trie Diagram]

- The tree after adding “stock” would look like the diagram above, where "stock" is added as a new path from the root node to the leaf node."
Standard Trie Construction

After “bear, bell, bid, bull, buy, sell, stock, stop”...
Improvements

What comes to mind for tries?
- e.g., is this entire tree really necessary?
Improvements

Two types of compression

- Compress internal single-children node sequences
  - Also called “PATRICIA Tries” – Why?
  - = Practical AlgoRIThm to Retrieve Information Coded In Alphanumeric (also called a “radix tree”)

- Compress external single-children leaf-node sequences
**Compressed Trie**

- A compressed trie has internal nodes of degree at least two.
- It is obtained from standard trie by compressing chains of “redundant” nodes.
Compact Representation

Compact representation of a compressed trie for an array of strings:

- Stores at the nodes ranges of indices instead of substrings
- Serves as an auxiliary index structure

```
S[0] = see
S[1] = bear
S[2] = sell
S[3] = stock
S[4] = bull
S[5] = buy
S[7] = hear
S[8] = bell
S[9] = stop
```
More Improvements

- We can build a standard trie and then compress it.
- But, can we build some sort of compressed trie directly?
- Ideas?
**Suffix Trie**

The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$.

```
minimize
0 1 2 3 4 5 6 7
```

```
   e
  / \
 i   \\
 /    \
 mize
```

```
   mi
  /  \
 nimize
```

```
   ze
  / \
 nimize
```

```
   ze
```

4/18/2016 6:48 AM
Suffix Trie

- Compact representation of the suffix trie for a string $X$ of size $n$ from an alphabet of size $d$
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in $X$ in $O(dm)$ time, where $m$ is the size of the pattern
  - Repetitive words not stored repetitively

![Diagram of a suffix trie]

$m i n i m i z e$

0 1 2 3 4 5 6 7
More Improvements

- There is still some repetition in this tree
  - e.g., “mize” appears several times
- How can we further compress the trie, thus reducing space and improving query time?
- Ideas?
A code is a mapping of each character of an alphabet to a binary code-word.

A prefix code is a binary code such that no code-word is the prefix of another code-word.

An encoding trie represents a prefix code:
- Each leaf stores a character.
- The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child).
Encoding Trie

- Given a text string $X$, we want to find a prefix code for the characters of $X$ that yields a small encoding for $X$
  - Frequent characters should have short code-words
  - Rare characters should have long code-words
  - Why?

Example
- $X = \text{abracadabra}$
- $T_1$ encodes $X$ into 29 bits
  - $29 = 3+2+3+3+2+3+2+3+3+3$
- $T_2$ encodes $X$ into how many bits?
  - $24 = 2+2+2+2+3+2+3+2+2+2+2+2$

How can we build a good encoding trie?
Huffman’s Algorithm

- Given a string $X$, Huffman’s algorithm constructs a prefix code that minimizes the size of the encoding of $X$.
- It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$.
- A heap-based priority queue is used as an auxiliary structure.

**Algorithm** *HuffmanEncoding*(X)

**Input** string $X$ of size $n$

**Output** optimal encoding trie for $X$

1. $C \leftarrow \text{distinctCharacters}(X)$
2. $\text{computeFrequencies}(C, X)$
3. $Q \leftarrow$ new empty heap
4. for all $c \in C$
   - $T \leftarrow$ new single-node tree storing $c$
   - $Q.\text{insert}(\text{getFrequency}(c), T)$
5. while $Q.\text{size}() > 1$
   - $f_1 \leftarrow Q.\text{minKey}()$
   - $T_1 \leftarrow Q.\text{removeMin}()$
   - $f_2 \leftarrow Q.\text{minKey}()$
   - $T_2 \leftarrow Q.\text{removeMin}()$
   - $T \leftarrow \text{join}(T_1, T_2)$
   - $Q.\text{insert}(f_1 + f_2, T)$
6. return $Q.\text{removeMin}()$
Example

\[ X = \text{abracadabra} \]

Frequencies

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

4/18/2016 6:48 AM
# Summary of Pattern Matching

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Search Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>$O(nm)$</td>
<td>simple, no preprocessing, slow (good for small inputs)</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>$O(nm+s)$</td>
<td>$O(m)$ preprocessing, significantly faster than previous in practice</td>
</tr>
<tr>
<td>KMP</td>
<td>$O(n+m)$</td>
<td>$O(m+s)$ preprocessing, more complex, but ideal very fast</td>
</tr>
<tr>
<td>Standard Trie</td>
<td>$O(dm)$</td>
<td>$O(n)$ preprocessing, $d = \text{size of alphabet}$, fast</td>
</tr>
<tr>
<td>Suffix Trie</td>
<td>$O(dm)$</td>
<td>$O(n)$ preprocessing, faster in practice because “compressed”</td>
</tr>
<tr>
<td>Huffman-Encoding Trie</td>
<td>$O(dm)$</td>
<td>$O(n+d\log d)$ preprocessing, fastest and smallest in practice, leads to lossless compression: ZIP</td>
</tr>
</tbody>
</table>