Outline and Reading
- Standard tries (§9.2.1)
- Compressed tries (§9.2.2)
- Suffix tries (§9.2.3)
- Huffman encoding tries (§9.3.1)

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
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A trie supports pattern matching queries in time proportional to the pattern size

Standard Trie (1)
- 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 \( S = \{ \text{bear}, \text{bell}, \text{bid}, \text{bull}, \text{buy}, \text{sell}, \text{stock}, \text{stop} \} \)

Standard Trie (2)
- 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 string parameter of the operation
  - \( d \): size of the alphabet

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
Compressed Trie

- A compressed trie has internal nodes of degree at least two.
- It is obtained from a 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.
  - Uses $O(s)$ space, where $s$ is the number of strings in the array.
  - Serves as an auxiliary index structure.

Suffix Trie (1)

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

Suffix Trie (2)

- 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.

Encoding Trie (1)

- 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 (2)

- 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 long code-words.
  - Rare characters should have short code-words.
- Example:
  - $X = \text{abracadabra}$
  - $T_1$ encodes $X$ into 29 bits.
  - $T_2$ encodes $X$ into 24 bits.
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 $\text{HuffmanEncoding}(X)$

Input: string $X$ of size $n$

Output: optimal encoding trie for $X$

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

Example

$X = \text{abracadabra}$

Frequencies

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

1. $\Rightarrow$

2. $\Rightarrow$

3. $\Rightarrow$