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 Tries

- 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 $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$

Analysis of Standard Tries

- 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

- insert the words of the text into trie
- Each leaf is associated with one particular word
- Leaf stores indices where associated word begins

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>l</td>
<td></td>
<td></td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>l</td>
<td>?</td>
<td>s</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>s</td>
<td>t</td>
<td>o</td>
<td>c</td>
<td>k</td>
<td>k</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>
```

```
| 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| b | i | d | s | t | o | c | k | l | b | i | d | s | t | o | c | k | l |
```

Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of "redundant" nodes
- Ex. the "i" and "d" in "bid" are "redundant" because they signify the same word
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

```
S[0] = [0, 1, 2, 3, 4]  S[4] = [0, 1, 2, 3]  S[7] = [0, 1, 2, 3]
S[1] = [b, e, a, r]    S[5] = [b, u, l, l]    S[8] = [b, e, l, l]
S[2] = [s, e, l, l]    S[6] = [b, l, d]       S[9] = [s, t, o, p]
S[3] = [s, l, o, c, k] S[7] = [s, l, o, c, k]
```

Suffix Trie

- The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$
Analysis of Suffix Tries

- 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
  - Can be constructed in $O(n)$ time

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 short code-words.
  - Rare characters should have long code-words.

Example
- $X = \text{abracadabra}$
- $T_1$ encodes $X$ into 29 bits.
- $T_2$ encodes $X$ into 24 bits.

$T_1$ and $T_2$ diagrams are shown.