Tries and Huffman Coding
Tries
Information retrieval

- Given a collection of strings, perform “search” on the collection
- Pattern matching: is a given string X in the collection?
- Prefix matching: find all strings that have a given string X as a prefix
Tries

- Pronounced “try”, from retrieval (?)
- Tree-based data-structure for storing strings
  - Supports efficient pattern and prefix matching
Standard Tries

- A: an alphabet (set of valid characters)
- S: a set of strings from A
  - No string is the prefix of another
- Standard trie is an ordered tree
  - Nodes (except root) labeled with character from A
  - Tree as |S| leaves: concatenation of characters from root to leaf is the associated string
Standard Tries Example

- $S = \{\text{bear, bell, bid, bull, buy, sell, stock, stop}\}$
For standard trie $T$, collection $S$ of strings of total length $n$, alphabet of size $d$:

- Every internal node of $T$ has at most $d$ children
  - How many children does the root have?
- $T$ has $|S|$ leaves
- Height of $T$ is length of longest string in $S$
- Number of nodes of $T$ is $O(n)$
  - Worst case: no shared prefixes
Performance: $d$ is the size of the alphabet, $n$ the total size of $S$ in characters, $m$ the size of a given string

- Construct: $O(dn)$
- Insert: $O(dm)$
- Find: $O(dm)$

$d$ is usually constant (e.g., natural language)
Standard Tries Applications

- Pattern or prefix matching
  - Tree search

- Word matching
  - Construct trie from a text
  - Find is successful if search terminates in a leaf
  - Running time independent of text size
Standard Tries Implementation

- Use a tree structure, with labels (characters) as keys
- Construct, insert, and find all rely on tree search
  - Not binary
Standard Tries Find

- Given a trie T and a pattern s

  1. \( n \leftarrow \text{root} \)
  2. \( \text{for } c \text{ in } s \)
     \( \text{if } n.\text{is_leaf}() \)
     \( \text{return False} \)
  3. \( \text{found_char }\leftarrow \text{False} \)
  4. \( \text{for } i = 1...n.\text{children}.\text{length} \)
     \( \text{if } c = = n.\text{children}[i].\text{key} \)
     \( n \leftarrow n.\text{children}[i] \)
     \( \text{found_char }\leftarrow \text{True} \)
     \( \text{break} \)
  5. \( \text{if } !\text{found_char} \)
     \( \text{return False} \)
  6. \( \text{return True} \)

- Finds if s is a prefix to at least a string in s
Standard Tries Find

What if we want to find if there is an exact match to s?

1. \( n \leftarrow \text{root} \)
2. \( \text{for c in s} \)
3. \( \quad \text{if n.is_leaf()} \)
4. \( \quad \quad \text{return False} \)
5. \( \quad \text{found_char} \leftarrow \text{False} \)
6. \( \quad \text{for i = 1...n.children.length} \)
7. \( \quad \quad \text{if c == n.children[i].key} \)
8. \( \quad \quad \quad n \leftarrow n.children[i] \)
9. \( \quad \quad \text{found_char} \leftarrow \text{True} \)
10. \( \quad \quad \text{break} \)
11. \( \quad \text{if !found_char} \)
12. \( \quad \quad \text{return False} \)
13. \( \quad \text{if n.is_leaf()} \)
14. \( \quad \quad \text{return True} \)
15. \( \text{return False} \)
Standard Tries Insertion

- Given a trie \( T \) and a pattern \( s \)
- Assume \( \text{find} \) returns last node of longest matching prefix in \( n \) and corresponding index in the string in \( \text{index} \)

1. \( n \leftarrow T.\text{root}, \text{index} \leftarrow 0 \)
2. \( \text{find}(s, n, \text{index}) \)
3. for \( i = \text{index} \ldots s.\text{length} \)
   4. \( \text{new}_n \leftarrow n.\text{children}.\text{add}(\text{new} \text{ Node}(s[i])) \)
5. \( n \leftarrow n.\text{new}_n \)
Standard Tries Construction

- Start from a text $t$ and an empty tree $T$
  - Assume no repetitions and no prefixes in $t$
- Insert words in trie iteratively
  1. $T \leftarrow \text{Tree}()$
  2. for $s$ in $t$
  3. $T$.insert($s$)
Standard Tries Redundancy
Solution: Compressed Tries
Compressed Tries

- Every node has at least two children
- Nodes are labeled with strings
  - What is wrong with this?
  - Tree has $O(s)$ nodes, but still has $O(n)$ characters!
- Assume we have a lookup table with all strings
  - Label nodes with indices into the table and strings instead
  - Still $O(s)$ nodes, now size is also $O(s)$
  - Application: search engines
Compressed Tries with Indices

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bear</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bell</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>bid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>bull</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>buy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sell</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>stock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>stop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suffix Tries

- Encode all possible suffixes of a string of length \( m \)
  - There are \( O(m) \) such suffixes
  - Using standard tries: space \( O(m^2) \)
  - Using compressed tries: space \( O(m) \)

- Supports substring matching in time \( O(k) \), where \( k \) is the length of the substring
Suffix Tries Example

<table>
<thead>
<tr>
<th>m</th>
<th>i</th>
<th>n</th>
<th>i</th>
<th>m</th>
<th>i</th>
<th>z</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```
  e
 / \
 i  m
 / \
 mize nimize ze
 / \
 nimize ze
```

```
  8,8
 / \
 2,2
 / \
 5,8 3,8 7,8
 / \
 3,8 7,8
 / \
 3,8 7,8
```
Huffman Coding
Data Compression

- We want to encode characters as bit-strings
  - Store documents on disk
  - Transmit information over a network
- We want to minimize the overall length of the bit-strings
Fixed-Length Strings

- Simple approach: use the same number of bits for each character
- For $n$ characters, use $\lceil \log(n) \rceil$ bits per character
- Example: ASCII, UNICODE
- Can we do better?
Variable-Length Strings

- Characters occur with different frequencies
- Idea: use variable-length codes
  - Shorter codes for more frequent characters
Variable-Length Strings Example: Morse Code

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>01</td>
</tr>
</tbody>
</table>

- Originally conceived for telegraphic communication
- Assign short codes for common letters
- How do we decode 0101?
  - eta, aet, aa, etet
- Solution: additional “separation character”
  - 0-1-01: eta
  - Traditionally a pause: 0 pause 1 pause 01
Prefix Codes

- Morse code: some codes are *prefixes* of others
  - e (0) is a prefix for a (01)
- Prefix code: no encoding is a prefix for another
  - Given a set S of characters
  - \( c(x) \) converts x from S into a sequence of bits
  - \( c(x) \) is not a prefix of \( c(y) \) for any x, y in S
- To decode a string, read from left to right and add character x to the decoded string as soon as you recognize the code for x
## Prefix Codes Example

<table>
<thead>
<tr>
<th>x</th>
<th>c(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>b</td>
<td>01</td>
</tr>
<tr>
<td>c</td>
<td>001</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
<tr>
<td>e</td>
<td>000</td>
</tr>
</tbody>
</table>

The binary string `0010000011101` can be decoded as `c e c a b`.
Optimal Prefix Codes

- Each character has a frequency \( f(x) \) in a text

\[
f(x) = \frac{\text{Times } x \text{ appears}}{\text{Total number of characters}} = \frac{n_x}{n}
\]

- Total text length: \( \sum n f(x) |c(x)| \)

- Average character length (ACL): \( \sum f(x) |c(x)| \)

- Goal: find the code with the minimum ACL
Prefix Codes Representation

- Binary tree $T$, each leaf corresponds to one character
  - Follow path from root to leaf: each time the path goes to a left child is a 0, right child is a 1
  - Must be a prefix code: characters are leaves
  - The length of an encoding is the depth of the leaf

- The optimal $T$ ($T^*$) must be a full binary tree
  - Proof by contradiction: if a node has only one child, replace it with its child, reducing the length
Prefix Codes Representation Example

<table>
<thead>
<tr>
<th>x</th>
<th>c(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>b</td>
<td>01</td>
</tr>
<tr>
<td>c</td>
<td>001</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
<tr>
<td>e</td>
<td>000</td>
</tr>
</tbody>
</table>
Huffman Coding Key Idea

- Huffman coding recursively makes the lowest frequency characters the deepest in the tree
  - Given an unlabeled FBT, the optimal code assigns the lowest frequency characters to the deepest leaves

- The two lowest frequency characters $y^*$ and $z^*$ must be sibling leaves in the tree
  - $y^*$’s parent must have another child $x$.
  - This child must be the second lowest frequency character $z^*$ (else, a shorter code would swap $x$ with $z^*$)
Huffman Coding Algorithm

- Repeat:
  - Find two characters with lowest frequency $y^*$, $z^*$
  - Merge $y^*$, $z^*$ into a super-character (parent node) $w$ with $f_w = f_{y^*} + f_{z^*}$
  - Remove $y^*$, $z^*$ from alphabet and add $w$
Huffman Code Pseudocode

HuffmanCode(frequencies)
1.   n = new List<Node>
2.   for f in frequencies
3.       n.add(new Node(f))
4.   q = priorityQueue(n)
5.   while q.size() > 1
6.       y = q.removeMin(), z = q.removeMin()
7.       w = new Node(y.key + z.key)
8.       w.left = y, w.right = z, y.parent = w, z.parent = w
9.       q.insert(w)
10.  T = new BinaryTree()
11.  T.root = q.removeMin()
12.  return T
Huffman Coding Example

- Find the Huffman code for the following alphabet
- Solution on blackboard

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.32</td>
</tr>
<tr>
<td>b</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>0.20</td>
</tr>
<tr>
<td>d</td>
<td>0.18</td>
</tr>
<tr>
<td>e</td>
<td>0.05</td>
</tr>
</tbody>
</table>
Huffman Coding Optimality

- Proof by induction on the size of the alphabet
  - Base case: alphabet of size two produces 1 bit code, which is clearly optimal
  - Inductive step:
    - T' is the tree produced with alphabet of n-1: optimal by inductive hypothesis
    - T is produced by adding y and z as children of w
    - ACL(T) = ACL(T') + f_y + f_z
    - If T is suboptimal, then an optimal tree Z must have y and z as sibling leaves, and must be generated by adding y and z as children of w to a tree Z'
    - ACL(Z) = ACL(Z') + f_y + f_z
    - ACL(Z) < ACL(T) \Rightarrow ACL(Z') < ACL(T')... contradiction!
Huffman Coding Efficiency

- For an alphabet of size $n$, the algorithm enters the loop $O(n)$ times.
- Each loop involves $2 \text{q.removeMin()}$ and one $\text{q.insert()}$.
- Using a min heap:
  - Each operation takes $O(\log n)$.
  - Total running time: $O(n \log n)$. 