Tries

- Trees of order $\geq 2$
- Variable length keys
- The decision on what path to follow is taken based on portion of the key
- Static environment, fast retrieval but large space overhead

Applications:
- dictionaries
- text searching (patricia tries)
- compression (Ziv-Lembel encoding)
Example

- MACCBOY
- MACAW
- MACARON
- MACARONI
- MACARONIC
- MACAQUE
- MACADAMIA
- MACADAM
- MACACACO
- MACABRE
  
  
  
  

E.G.M. Petrakis
1. Words in Tree

@: word terminating symbol
as many words as terminating symbols
Word Searching

- At most $m+1$ comparisons to find the word $x@ = a_1a_2...a_{m+1}$, with root: $i=1$ and $t(a_i)$: child of node $a_i$.

```java
i=1;  u = a_i;
while (u != @ && i <= m) {
    if (t(a_i): undefined)
        X is not in the trie;
    else {
        u = t(a_i);  i = i+1;
    }
    if (u == @ && word(t) == X) X is in the trie;
    else X is not in the trie;
}
```
2. Words at the Leafs
Example Implementation

26 symbols/node => large space overhead
Insertions

“bluejay” inserted

Figure 11.23 Section of trie of Figure 11.18 showing changes resulting from inserting bobwhite and bluejay
3. Compact Trie

compact nodes

words at the leafs
Patricia Trie

- Practical Algorithm To Retrieve Information Coded In Alphanumeric
  - especially good for text searching
  - replace sub-words by a reference position in text together with its length
    - e.g., “jay” in “the blue jay” is (10,3)
  - internal nodes keep only the length and the first letter of the phrase
  - external nodes keep the position of the phrase in the text
Suffix Trie

- Construct a trie with all suffixes
- “the blue-jay@” => 12 suffixes
- Merge branches with common prefix
- Leafs point to suffices
- Search: “e-jay”
  - go to position 3 or 7
  - increase by query length: 5
  - check suffices at positions 3+5 and 7+5
  - the second one matches

1. theblue-jay@
2. heblue-jay@
3. eblue-jay@
4. blue-jay@
5. lue-jay@
6. ue-jay@
7. e-jay@
8. -jay@
9. jay@
10. ay@
11. y@
12. @
Suffix Trie

The blue-jay

Search: “e-jay@”
Search Patricia

- Scan query from left to right
  - go to first matching symbol in trie
  - increase by the increment
  - if not in the trie => search failed
  - if all symbols match the query => found !!!
Patricia Pros & Cons

- Good for secondary storage applications
  - the trie is not high
- The space overhead is still a problem
- Mainly useful for static environments
Ziv-Lempel Encoding

- Mainly compression method
  - for any kind of data
  - based on tries and symbol alphabets
- Basic idea: compute code for repeating symbols or for sets of symbols
- Globally and locally optimal
- Compared with Huffman
  - no a-priori knowledge of symbol probabilities
  - Huffman is globally optimum but, not always
  - Locally optimal (for parts of the input)
Encoding

- Construct trie:
  - nodes: code of a symbol or of sequences of symbols
  - arc: alphabet symbol
  - initially a trie with all symbols
  - each symbol is assigned a unique code

- Scan input from left to right:
  - read next symbol
  - follow appropriate branch in trie
  - if symbol not in trie => insert symbol in trie, assign it a new code and output its code
initial trie

- root
- Codes: 1 for a, 2 for b, 3 for c, 26 for z
- Nodes: 1, 2, 3, 26
alphabet \{a, b, c\}

initial trie:

\[
\begin{array}{ccc}
\alpha & b & c \\
1 & 2 & 3 \\
\end{array}
\]

Input:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</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>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Decoding

- Follows the same sequence of steps in reverse order
  - start from the same initial trie
  - scan the code from left to right
  - read next code symbol
  - go to the node having the same code
  - if the code is not in the trie guess the appropriate position in the trie
Alphabet: \{a, b, c\}
Code: 1, 2, 4, 3, 5, 8, 1
read(1): must be coming from “a”

In order to go from “1” to “2”, we must have visited “b”
In order to go from “2” to “4” we must go through “ab” (passing from 1) “a” was a child of “2”
In order to go from “4” to “3” we must go through “c” and “c” was child of “4”
In order to go from “3” to “5” we must go through “ba” and “b” was child of “3”
The next symbol is “8”, but ... there is no “8” in the trie!!
This may happen only if after “5” we passed through the same path since “8” cannot be under “6” or “7” (there is no “6” or “7” in the code).
output “$\alpha$”
Generalize

output “dxd”

Special case: “x” can be null