Tries

A estrutura de dados *trie*

Uma *trie* é uma árvore cujos nós têm filhos que correspondem a símbolos do alfabeto das chaves

Uma chave está contida numa *trie* se o percurso que ela induz na *trie*, a partir da sua raiz, termina num nó que marca o fim de uma chave

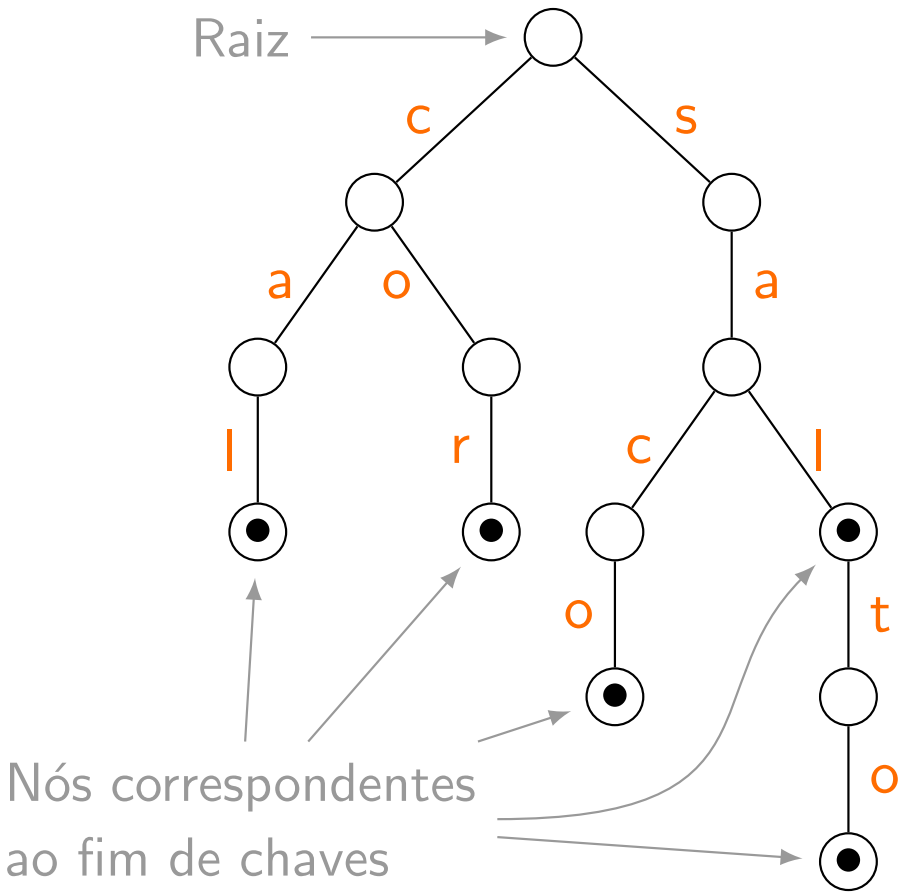
As *tries* apresentam algumas características que as distinguem de outras estruturas de dados

1. A complexidade das operações não depende do número de elementos que ela contém
2. As chaves não têm de estar explicitamente contidas na *trie*
3. As operações não se baseiam em comparações entre chaves

Uma *trie*

Exemplo

Representação de uma *trie* com as chaves (palavras) cal, cor, saco, sal e salto



Tries

d – dimensão do alfabeto (nº de símbolos distintos)

Chaves

k[1..m] – chave

|k| = m

Conteúdo dos nós (implementação com vector de filhos)

c[1..d] – filhos

p – pai (opcional)

word – TRUE sse a chave que termina no nó está na *trie*

ou

element – elemento associado à chave que termina(ria) no nó

TRIE-SEARCH(T, k)

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 return x != NIL and x.word
```

Argumentos

T – *trie*

k – chave (palavra)

TRIE-SEARCH(T, k) — Complexidade

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 return x != NIL and x.word
```

Análise da complexidade para o pior caso

- Linhas 1, 2, 4, 5 e 6, e testes da linha 3: custo constante

$$O(1) + O(1) + (m+1)O(1) + m\,O(1) + m\,O(1) + O(1) \;=$$
$$4\,O(1) + 3m\,O(1) \;=\; 3\,O(m) \;=$$
$$O(m)$$

TRIE-INSERT(T, k)

```
1 if T.root = NIL then
2     T.root <- ALLOCATE-NODE()
3     T.root.p <- NIL
4 x <- T.root
5 i <- 1
6 while i <= |k| and x.c[k[i]] != NIL do
7     x <- x.c[k[i]]
8     i <- i + 1
9 TRIE-INSERT-REMAINING(x, k, i)
```

ALLOCATE-NODE() devolve um novo nó da *trie* com

c[1..d] = NIL
p = NIL
word = FALSE

TRIE-INSERT-REMAINING(x, k, i)

```
1 y <- x
2 for j <- i to |k| do
3     y.c[k[j]] <- ALLOCATE-NODE()
4     y.c[k[j]].p <- y
5     y <- y.c[k[j]]
6 y.word <- TRUE
```

Função que acrescenta, a partir do nó x, os nós necessários para incluir na *trie* o sufixo da chave k que ainda não está na *trie* e que começa no i-ésimo símbolo da chave

Se $i > |k|$, só afecta a marca de fim de palavra no nó x

TRIE-DELETE(T, k) (1)

```
1 x <- T.root
2 i <- 1
3 while x != NIL and i <= |k| do
4     x <- x.c[k[i]]
5     i <- i + 1
6 if x != NIL and x.word then
7     x.word <- FALSE          // k deixa de estar na trie
8 ...
```

Falta remover os nós da *trie* que deixam de ter um papel útil, por não corresponderem ao fim de uma palavra nem terem filhos

TRIE-DELETE(T, k) (2)

```
 5 ...
 6 if x != NIL and x.word then
 7     x.word <- FALSE          // k deixa de estar na trie
 8     repeat
 9         i <- i - 1
10         childless <- TRUE              // x tem filhos?
11         j <- 1
12         while j <= d and childless do
13             if x.c[j] != NIL then
14                 childless <- FALSE
15             j <- j + 1
16         if childless then      // se não tem, é apagado
17             y <- x.p
18             if y = NIL then
19                 T.root <- NIL     // a trie ficou vazia
20             else
21                 y.c[k[i]] <- NIL
22             FREE-NODE(x)
23           x <- y
24     until x = NIL or not childless or x.word
```

Complexidade temporal das operações sobre uma *trie*

Implementação com vector de filhos — Resumo

Pesquisa da palavra k

$$O(m)$$

Inserção da palavra k

$$O(m)$$

Remoção da palavra k

$$O(m\,d)$$

Complexidade espacial

$$O(n\,w\,d)$$

Onde

- $m = |\mathrm{k}|$
- d é o número de símbolos do alfabeto
- n é o número de palavras na *trie*
- w é comprimento médio das palavras na *trie*