

Tries

Referências

- [Tries \(árvores digitais\) \(PF\)](#),
- [Tries \(S&W\)](#),
- [slides \(S&W\)](#)

Vídeo

[Tries \(S&W\)](#)

Tries

Uma **trie** (pronuncie “trái”) é um tipo de árvore usado para implementar TSs de strings. Tries também são conhecidas como **árvores digitais** e como **árvores de prefixos**.

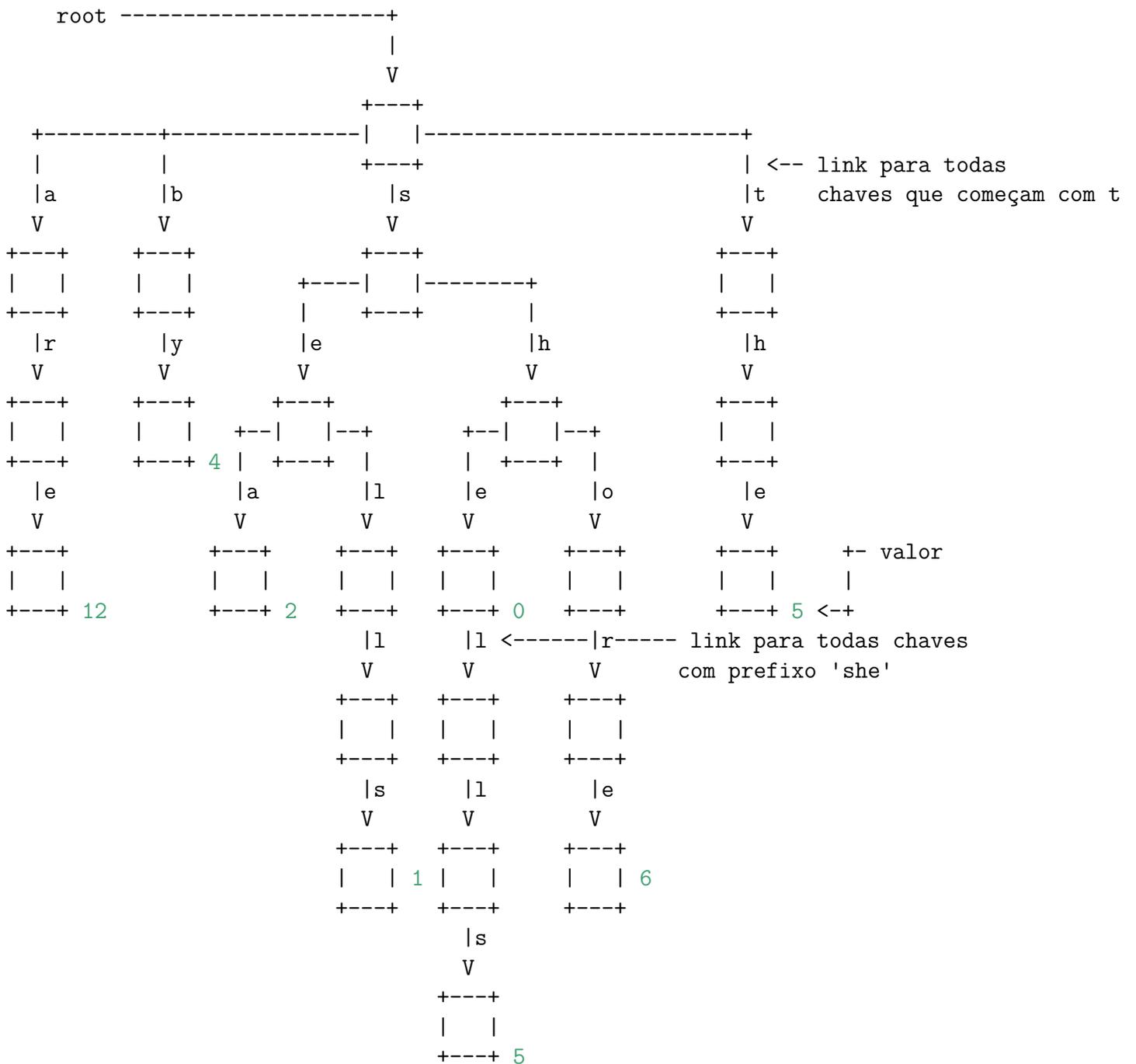
A API desta ST inclui, além dos métodos usuais como `put()`, `get()`, `delete()`, ..., possui 3 métodos específicos: `keysWithPrefix()`, `keysThatMatch()` e `longestPrefixOf()`.

```
public class StringST<Value>
-----
StringST()      cria uma TS de strings vazia
void           put(String key, Value val)   insere o par (key, val) nesta tabela
Value         get(String key)             valor associado à chave key
void          delete(String key)          remove key e seu valor desta tabela
boolean       isEmpty()                  esta tabela de símbolos está vazia?
int           size()                     número de pares (chave, valor) desta tabela
Iterable<String> keys()                   todas as chaves desta tabela
Iterable<String> keysWithPrefix(String s) todas as chaves que têm prefixo s
Iterable<String> keysThatMatch(String s)  todas as chaves que "casam" com s
quando '.' é usado como curinga
String        longestPrefixOf(String s)  a chave mais longa que é prefixo de s
```

Cada nó tem um pai, com exceção feita a raiz. Cada nó da árvore tem até R filhos, onde R é o tamanho do alfabeto.

Exemplo

key	val
are	12
by	4
sea	2
sells	1
she	0
shells	3
the	5
shore	6



Exemplos

Exemplos para o conjunto de chaves she sells sea shells by the sea shore :

```
keysWithPrefix("she") devolve "she" e "shells"
keysWithPrefix("se") devolve "sells" e "sea"
keysThatMatch(".he") devolve "she" e "the"
keysThatMatch("s..") devolve "she" e "sea"
longestPrefixOf("shell") devolve "she"
longestPrefixOf("shellsort") devolve "shells"
```

Duas observações importantes sobre tries:

- chaves ficam codificadas nos caminhos que começam na raiz;
- prefixos de chaves, que nem sempre são chaves, estão representados na trie.

Estrutura de uma trie

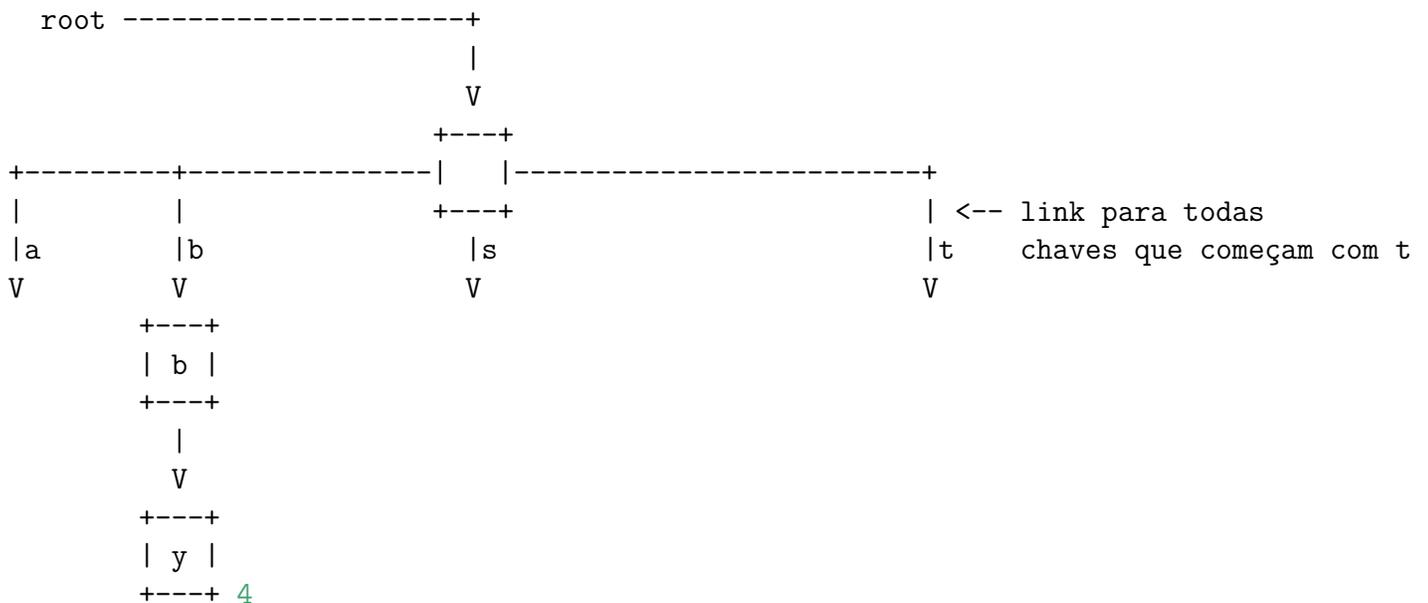
Os links da estrutura correspondem a caracteres e não a chaves. Nas figuras, o caractere escrito dentro de um nó é o caractere do link que entra no nó.

Tries são compostas por nós do tipo `Node`. O campo `val` é um `Object` porque pode eventualmente ser um vetor e Java não permite vetores genéricos:

```
private static class Node {
    private Object val;
    private Node[] next = new Node[R];
}
```

Muitos dos `R` ponteiros podem ser `null`.

Se a trie é para o alfabeto 'a', 'b', ..., 'z' temos




```

public void delete(String k) {...}

public Iterable<String> keys() {...}
}

```

Busca

Seguimos os ponteiros soletrando a string key.

```

public Value get(String key) {
    Node x = get(root, key, 0);
    if (x == null) return null;
    return (Value) x.val;
}

private Node get(Node x, String key, int d) {
    if (x == null) return null;
    if (d == key.length()) return x;
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
}

```

Inserção

É feita uma busca. Se a chave é encontrada o valor val é substituído. Caso contrário chegamos a um null é evemos continuar a inserção ou chegamos no último caractere da chave.

Exemplos: she sells sea shells by the sea shore

```

public void put(String key, Value val) {
    root = put(root, key, val, 0);
}

private Node put(Node x, String key, Value val, int d) {
    if (x == null) x = new Node();
    if (d == key.length()) {
        if (x.val == null) n++;
        x.val = val;
        return x;
    }
    char c = key.charAt(d);
    x.next[c] = put(x.next[c], key, val, d+1);
    return x;
}

```

Remoção

A operação `delete()` remove uma dada chave `k` do conjunto de chaves da trie. Em princípio, a implementação da operação `delete()` é fácil: basta encontrar o nó `x` localizado pela string `k` e fazer

```
x.val = null;
```

Infelizmente, a trie resultante dessa operação pode não ser limpa, mesmo que a trie original seja limpa. Para manter a trie limpa, é preciso fazer algo mais complexo:

```
public void delete(String key) {
    root = delete(root, key, 0);
}

private Node delete(Node x, String key, int d) {
    if (x == null) return null;
    if (d == k.length())
        x.val = null;
    else {
        char c = k.charAt(d);
        x.next[c] = delete(x.next[c], key, d+1);
    }
    if (x.val != null) return x;
    for (char c = 0; c < R; c++)
        if (x.next[c] != null) return x;
    return null;
}
```

keys()

```
public Iterable<String> keys() {
    return keysWithPrefix("");
}
```

keysWithPrefix()

```
/**
 * Devolve todas as chaves na TS que têm prefixo prefix.
 */
public Iterable<String> keysWithPrefix(String prefix) {
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}
```

collect()

O método coloca na fila q todas as chaves da subtrie cuja raiz é x depois de acrescentar o prefixo pre a todas essas chaves.

```
private void collect(Node x, String pre, Queue<String> q) {
    if (x == null) return;
    if (x.val != null) q.enqueue(pre);
    for (char c = 0; c < R; c++)
        collect(x.next[c], pre + c, q);
}
```

longestPrefixOf()

```
public String longestPrefixOf(String s) {
    int max = -1;
    Node x = root;
    for (int d = 0; x != null; d++) {
        if (x.val != null) max = d;
        if (d == s.length()) break;
        x = x.next[s.charAt(d)];
    }
    if (max == -1) return null;
    return s.substring(0, max);
}
```

keysThatMatch()

```
/**
 * Devolve todas as chaves que casam com o padrão pat
 * (em particular, todos os caracteres . em pat são curingas).
 */
public Iterable<String> keysThatMatch(String pat) {
    Queue<String> q = new Queue<String>();
    collect(root, "", pat, q);
    return q;
}

/**
 * Acrescenta à fila q todas as chaves da trie
 * que têm prefixo prefix e casam com o padrão pat.
 * (Supõe que prefix.length() <= pat.length().)
 */
public void collect(Node x, String prefix, String pat, Queue<String> q) {
    if (x == null) return;
    if (prefix.length() == pat.length() && x.val != null)
        q.enqueue(prefix);
    if (prefix.length() == pat.length())
```

```

    return;
char next = pat.charAt(prefix.length());
for (int c = 0; c < R; c++)
    if (next == '.' || next == c)
        collect(x.next[c], prefix + c, pat, q);
}

```

Análise

A aparência de uma BST (árvore binária de busca) depende muito da ordem em que as chaves são inseridas e removidas. Não é o que acontece com tries (e portanto a expressão “trie balanceada” não faz sentido).

Proposição F: A estrutura de uma trie não depende da ordem em que as chaves são inseridas e removidas.

O consumo de tempo das operações sobre uma trie não depende do número, n , de chaves:

Proposição G: O número de nós visitados para buscar ou inserir uma chave de comprimento w em uma trie é no máximo $1 + w$.

O consumo de tempo médio em uma busca malsucedida é bem menor que $1 + w$, pois a busca termina tão logo encontramos um link `null`.

Proposição H: O número esperado de nós visitados durante uma busca malsucedida em uma trie com n chaves aleatórias sobre um alfabeto de tamanho R é aproximadamente $\log_R n$.

Consequência pouco intuitiva: se as chaves são aleatórias, o consumo de tempo médio não depende do comprimento das chaves.

Proposição I: O número de links em uma trie é entre Rn e Rnw onde w é o comprimento médio de uma chave.