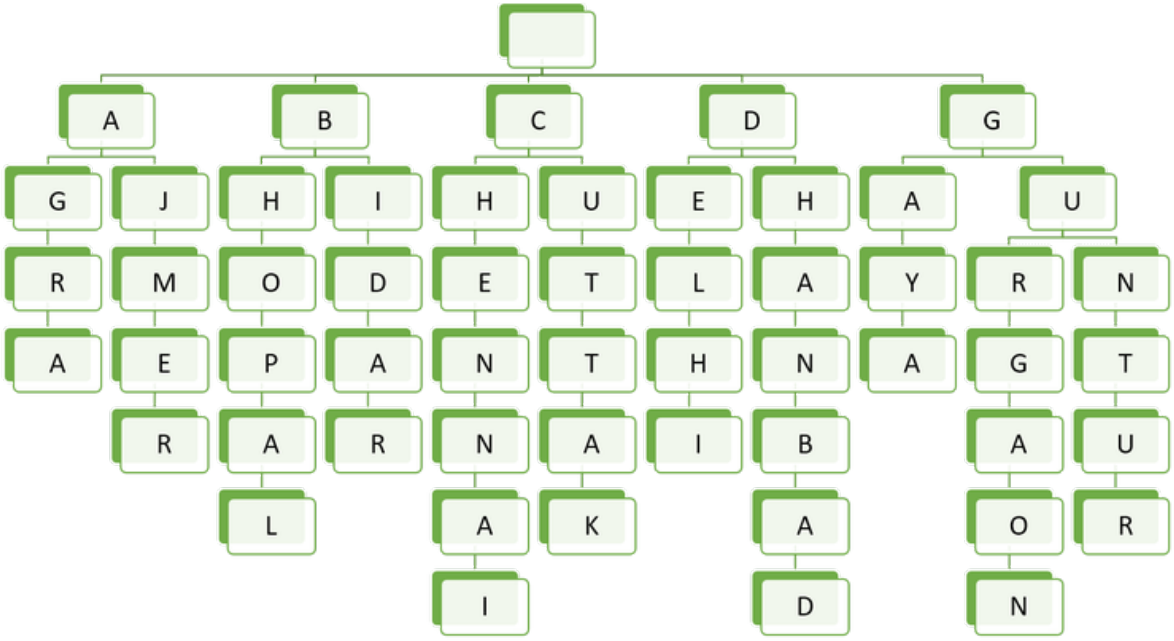
AULA 16

Tries (árvores digitais)



Fonte: `Building an autocomplete system using Trie`

Referências: Tries (árvores digitais) (PF); Tries (S&W); slides (S&W); Vídeo (S&W); TAOCP, vol 3, cap. 6.3

R-way tries

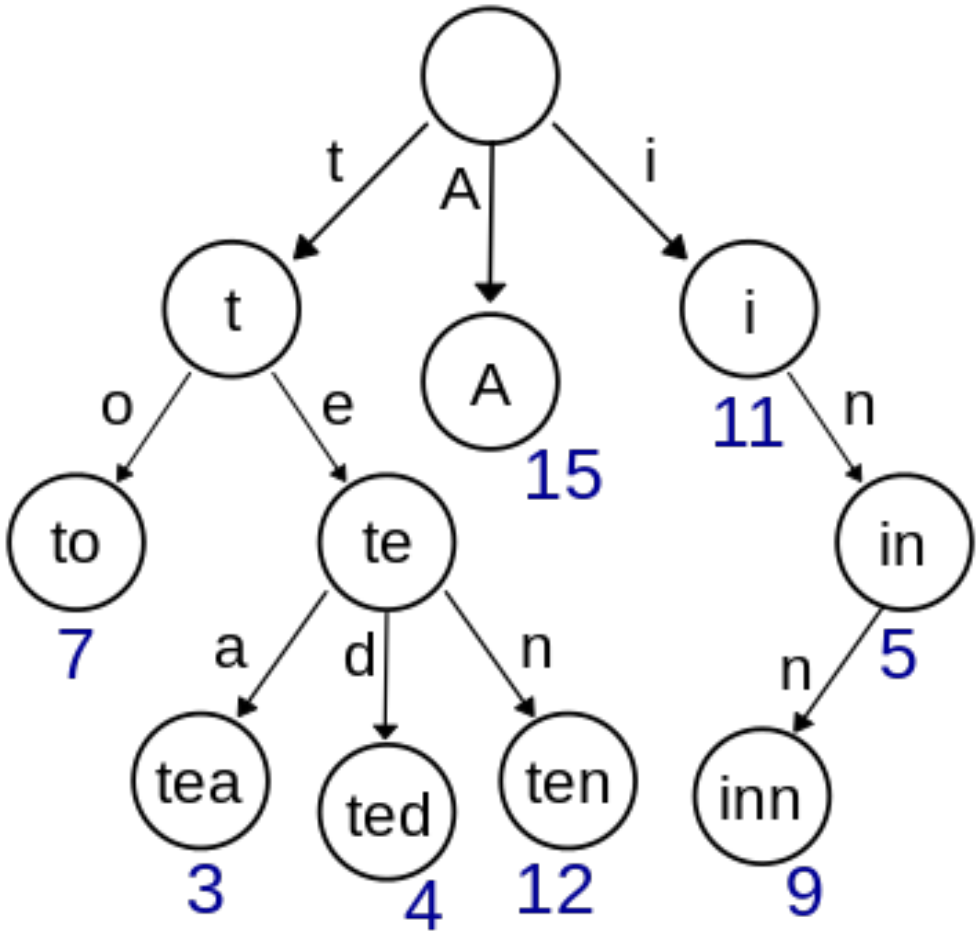
Uma **trie** (=*R-way trie*) é um tipo de árvore usado para implementar STs de strings sobre um alfabeto com R símbolos.

Tries também são conhecidas como árvores digitais e como árvores de prefixos.

Com Tries em vez de o método de busca ser baseado em comparações entre chaves, é utilizada a representação das chaves como caracteres de um alfabeto.

Considere, por exemplo, a busca de uma palavra no dicionário: a primeira letra indica as páginas que devemos olhar; a segunda letra restringe o espaço de busca; ....

Ilustração



Fonte: Wikipedia

Métodos específicos

A `API` de uma `trie` inclui, além do métodos usuais como `put()`, `get()`, `delete()`,..., 3 métodos específicos:

- `keysWithPrefix(String s)`: todas as chaves que têm prefixo `s`
- `keysThatMatch(String s)`: todas as chaves que *casam* com `s` quando '`.`' é usado como **curinga**
- `longestPrefixOf(String s)` a chave mais longa que é prefixo de `s`

Métodos específicos

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"

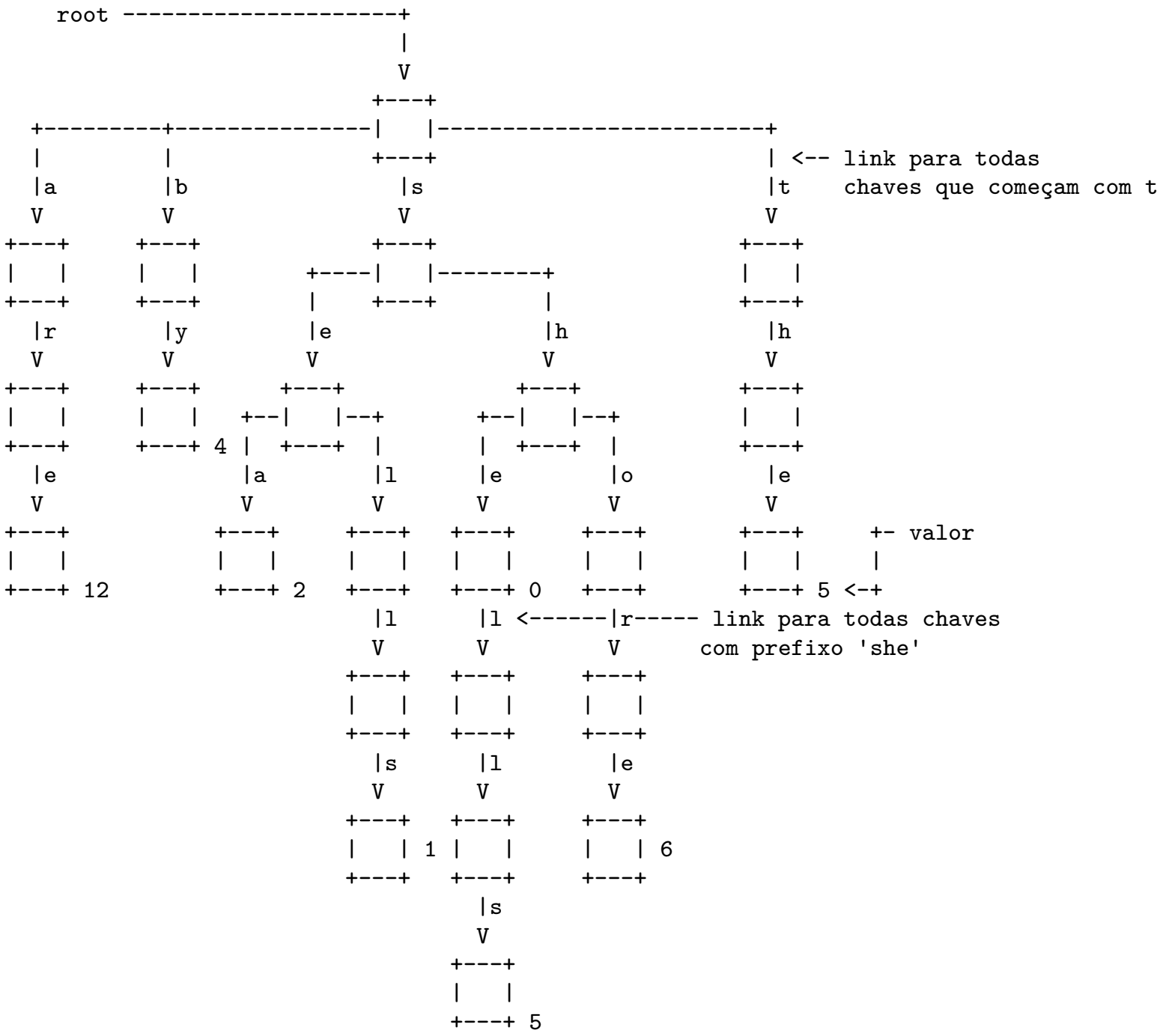
`longestPrefixOf("shellsort")` devolve "shells"

Outra ilustração

Para os pares `key`-`val`

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

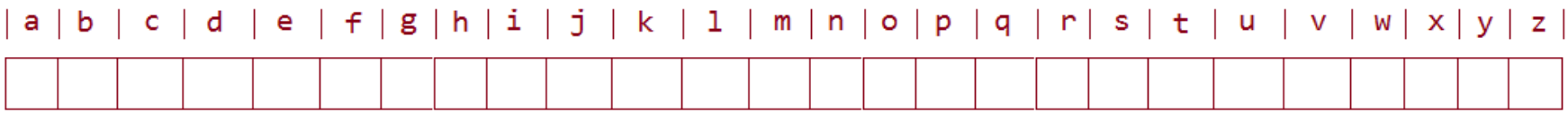
temos a `trie` a seguir


root
<-- link para todas
chaves que começam com t
4
12
2
0
5 <-+
+- valor
link para todas chaves
com prefixo 'she'
1
6
5

Estrutura do nó de uma `trie`

Os links correspondem a caracteres e não a chaves.
Tries são compostas por nós do tipo `Node`.

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

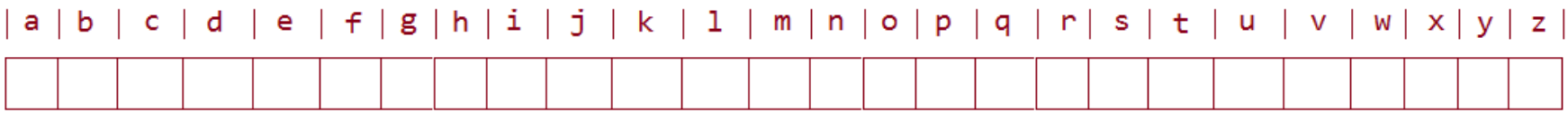


Fonte: JavaByPatel

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

Ilustração

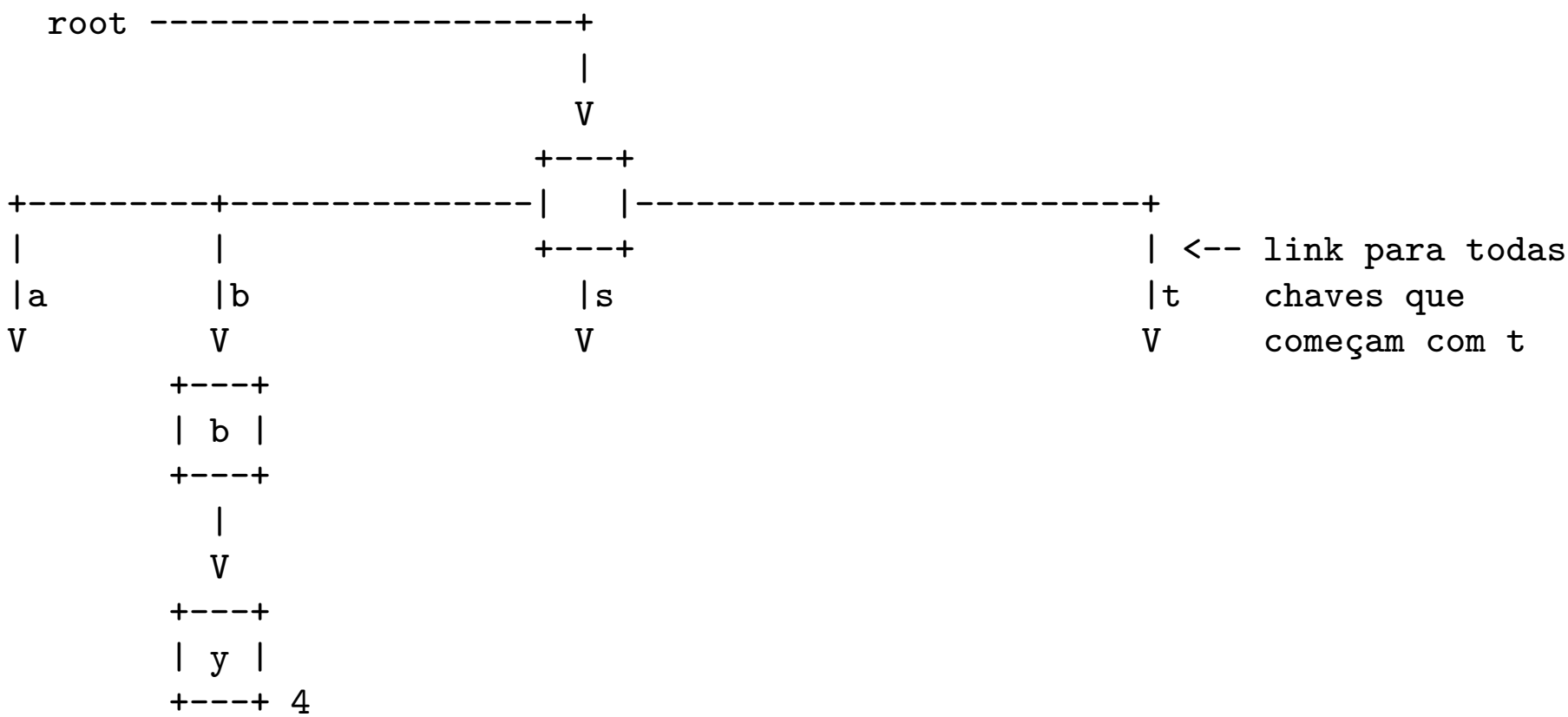
Se a trie é para o alfabeto `'a', 'b',...,'z'` temos



Fonte: `JavaByPatel`

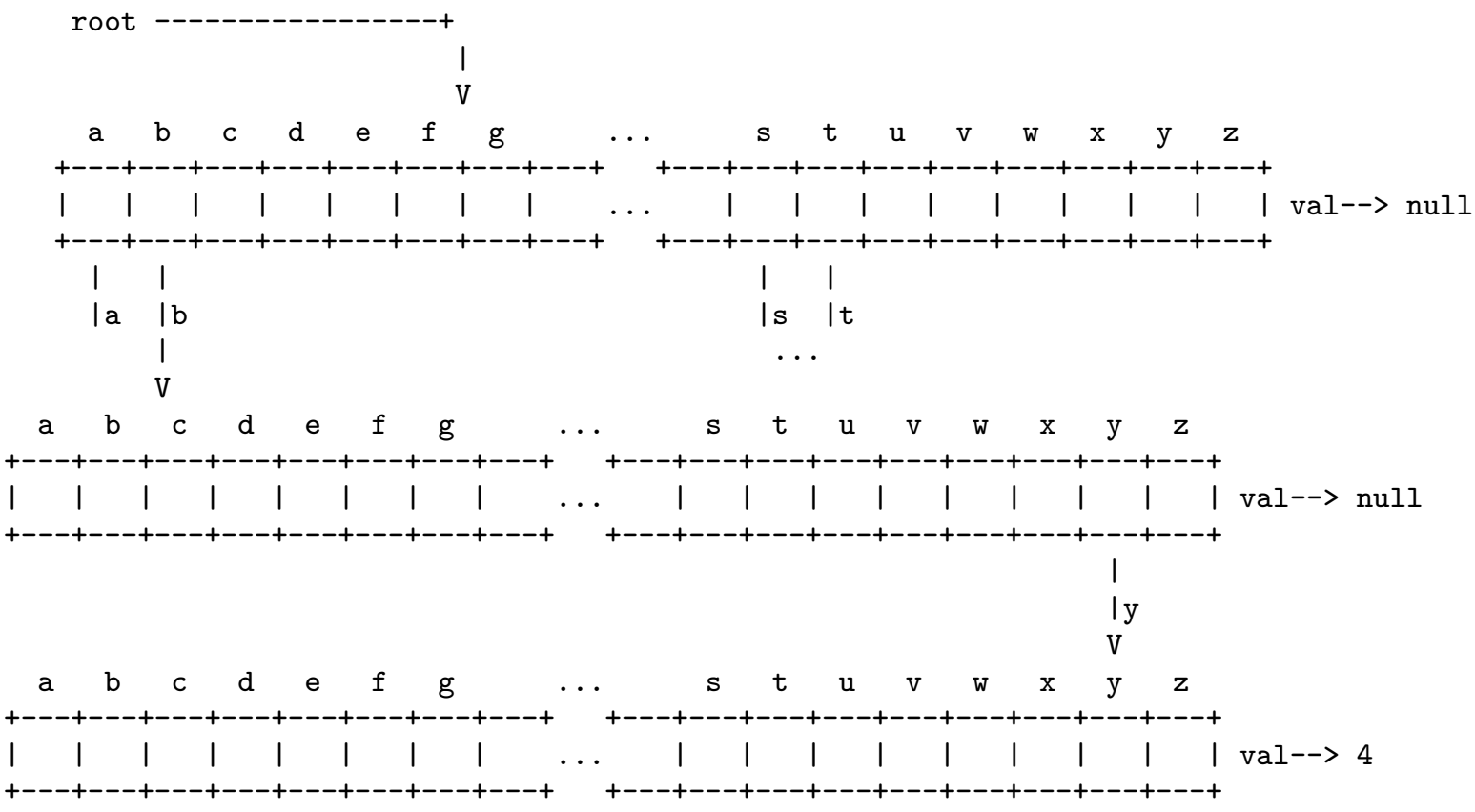
Ilustração

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



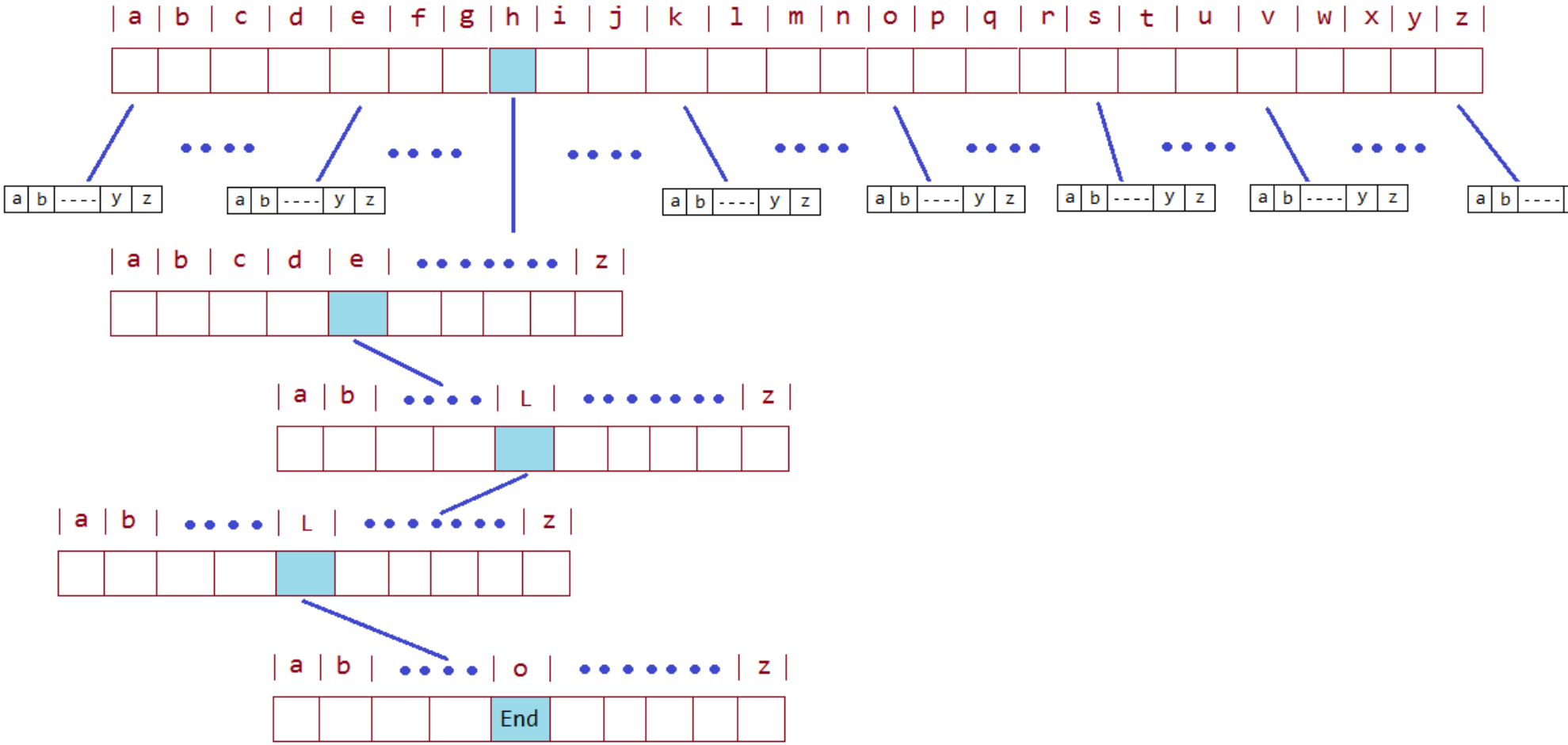
Outra ilustração

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



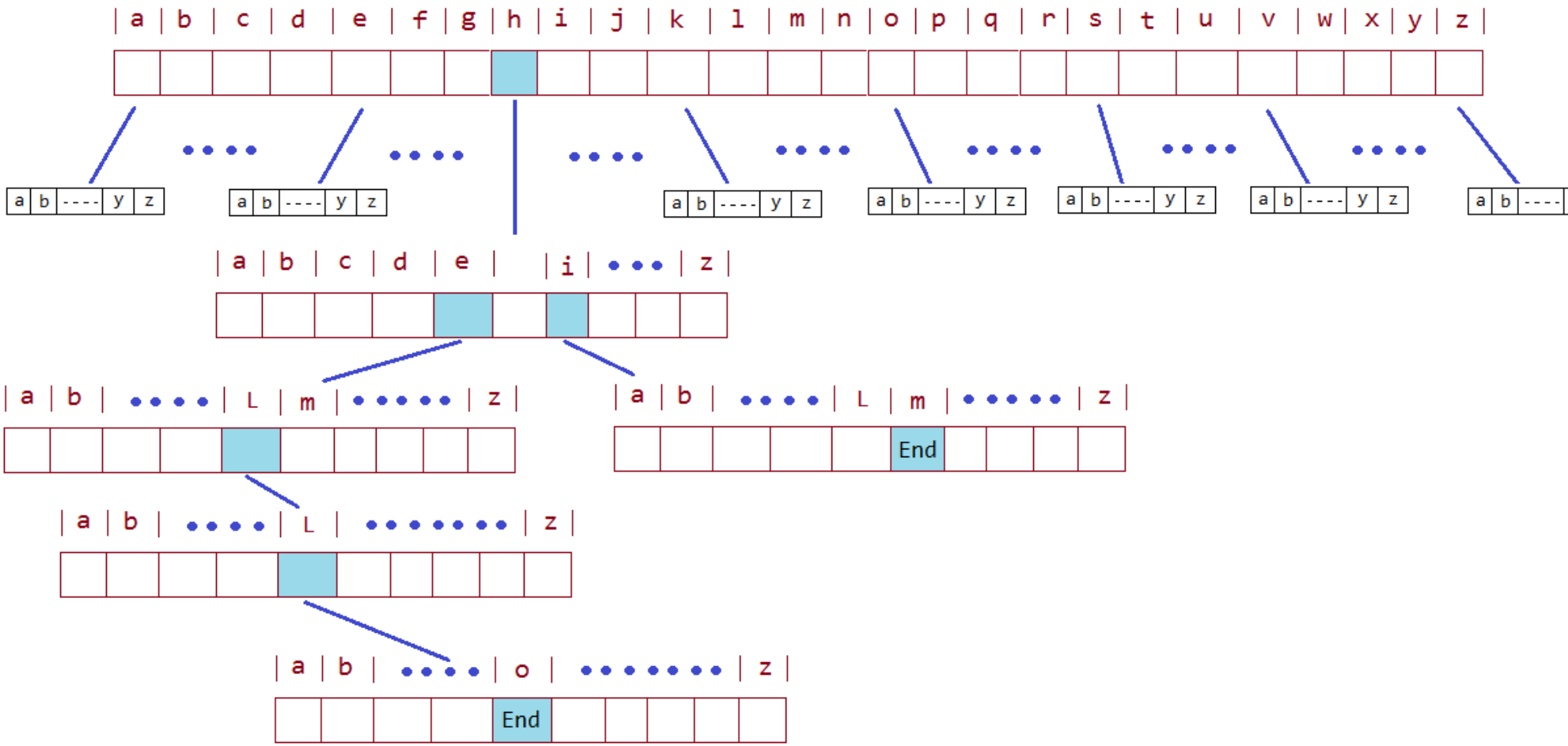
as posições vazias representam `null`.

Trie para "hello"



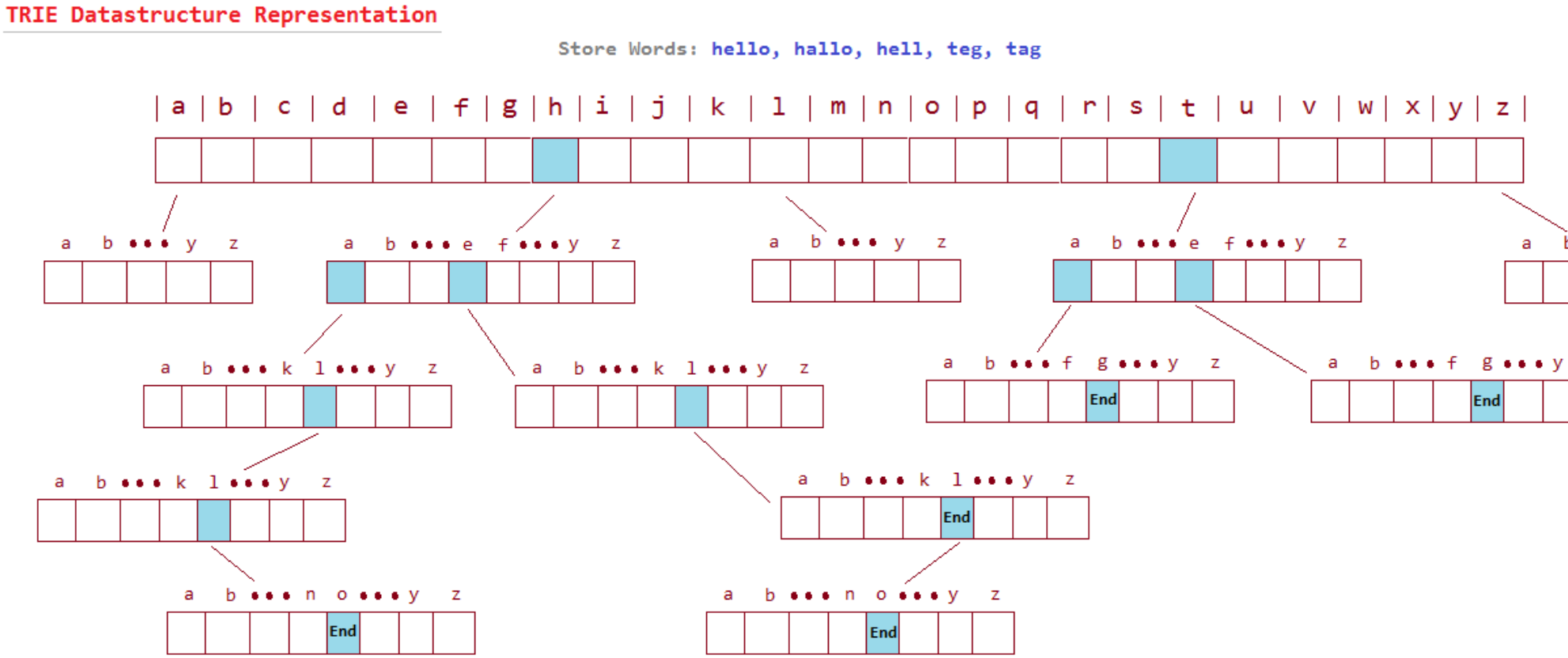
Fonte: JavaByPatel

Trie para "hello" e "him"



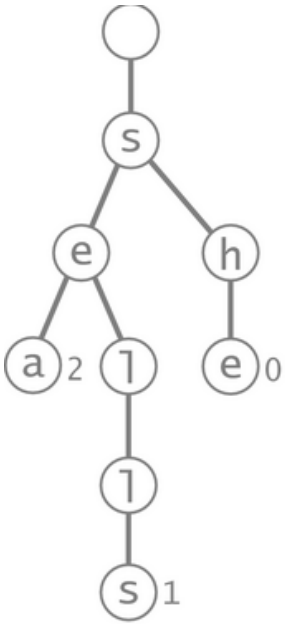
Fonte: JavaByPatel

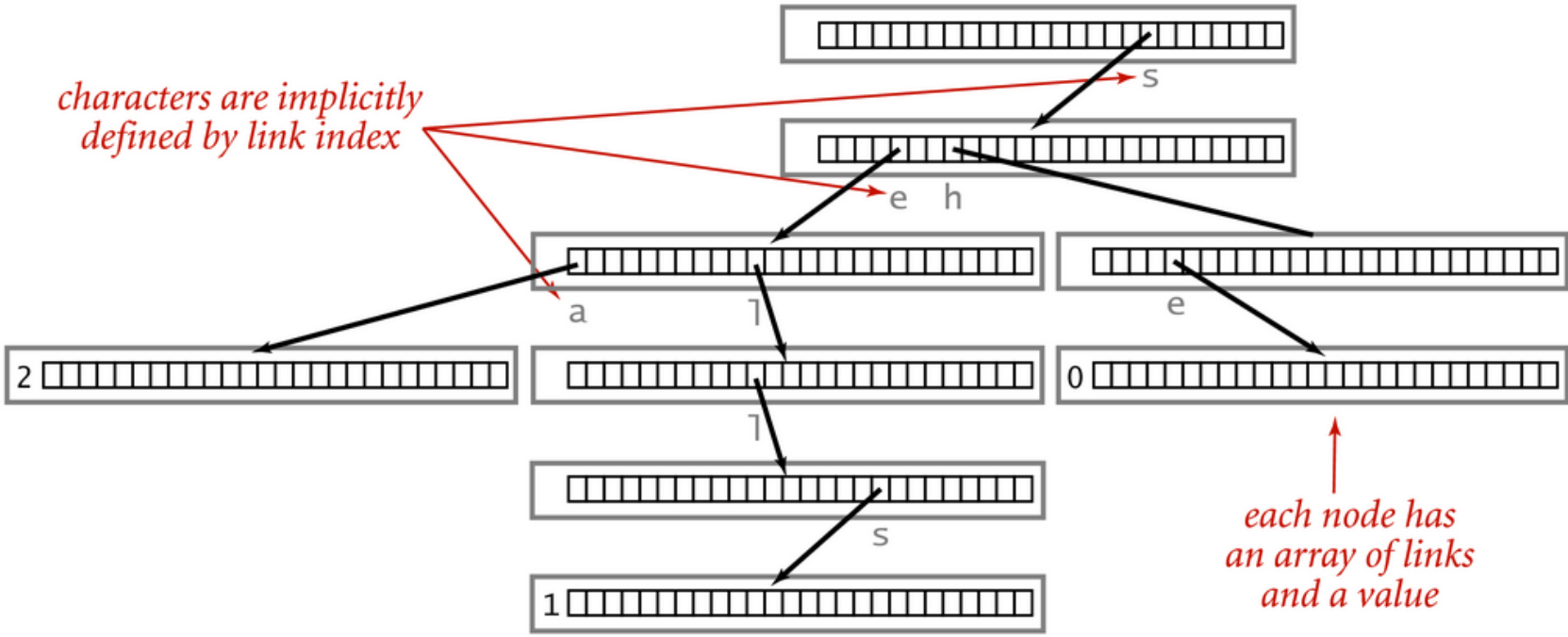
Trie para ...



Fonte: JavaByPatel

Trie para "sea", "sells" e "he"

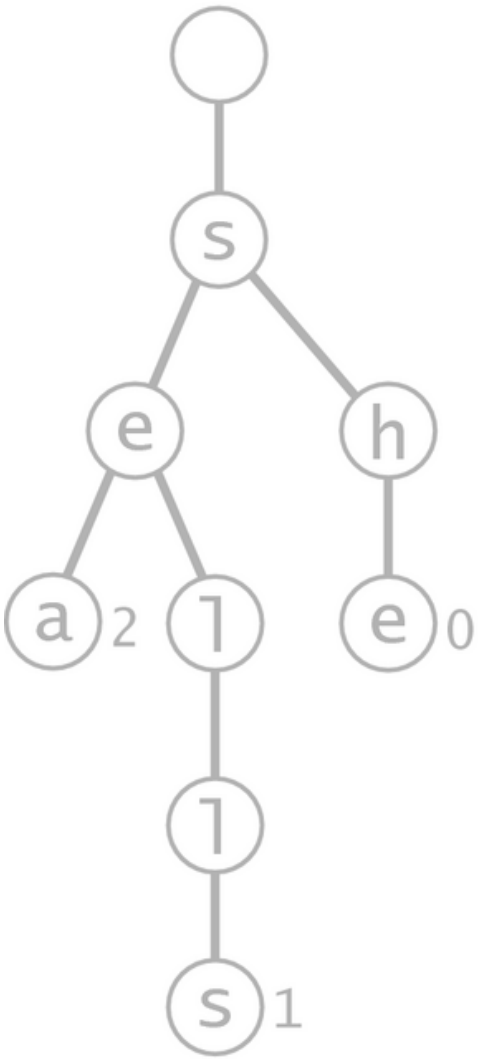


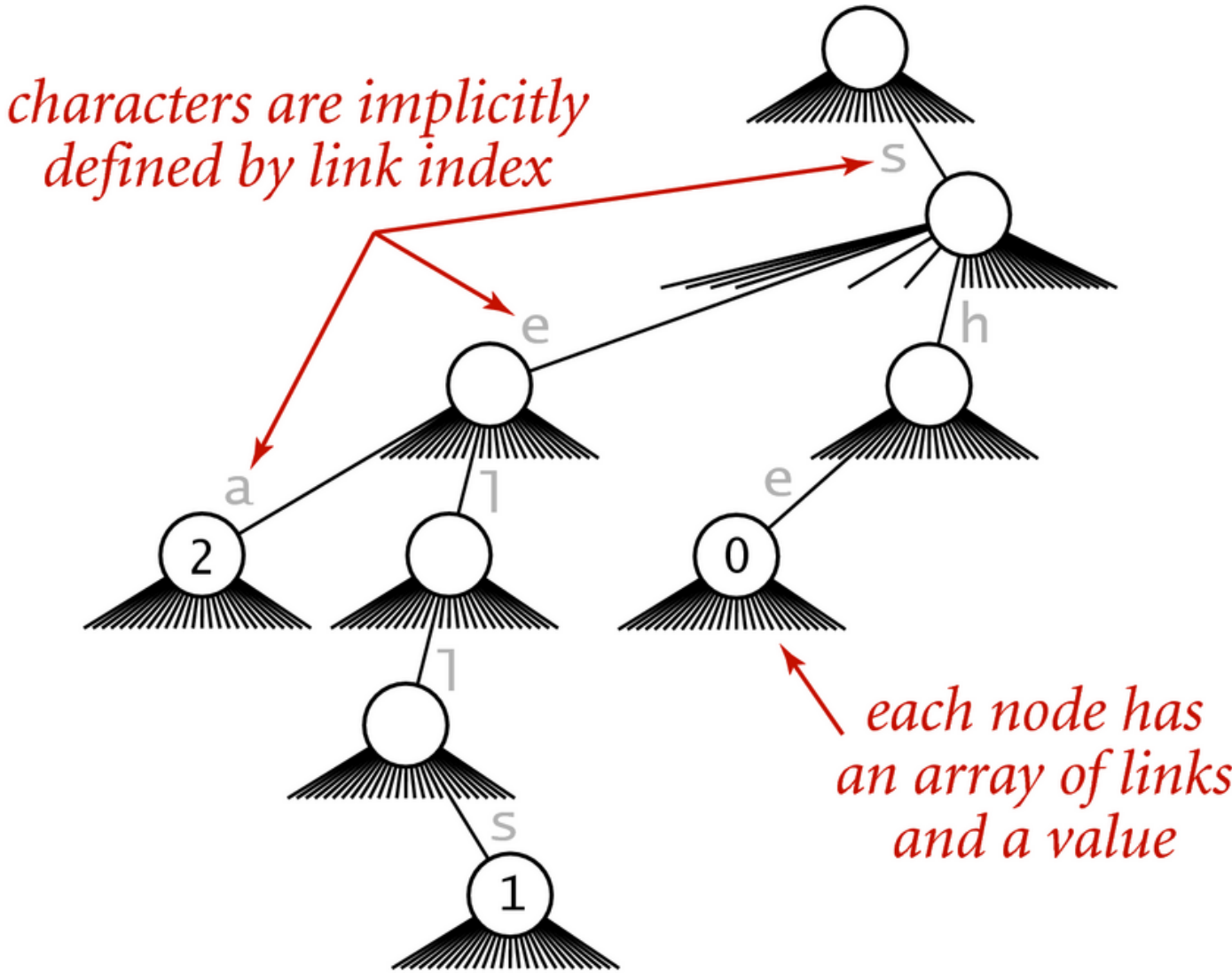


Trie representation ($R = 26$)

Fonte: algs4

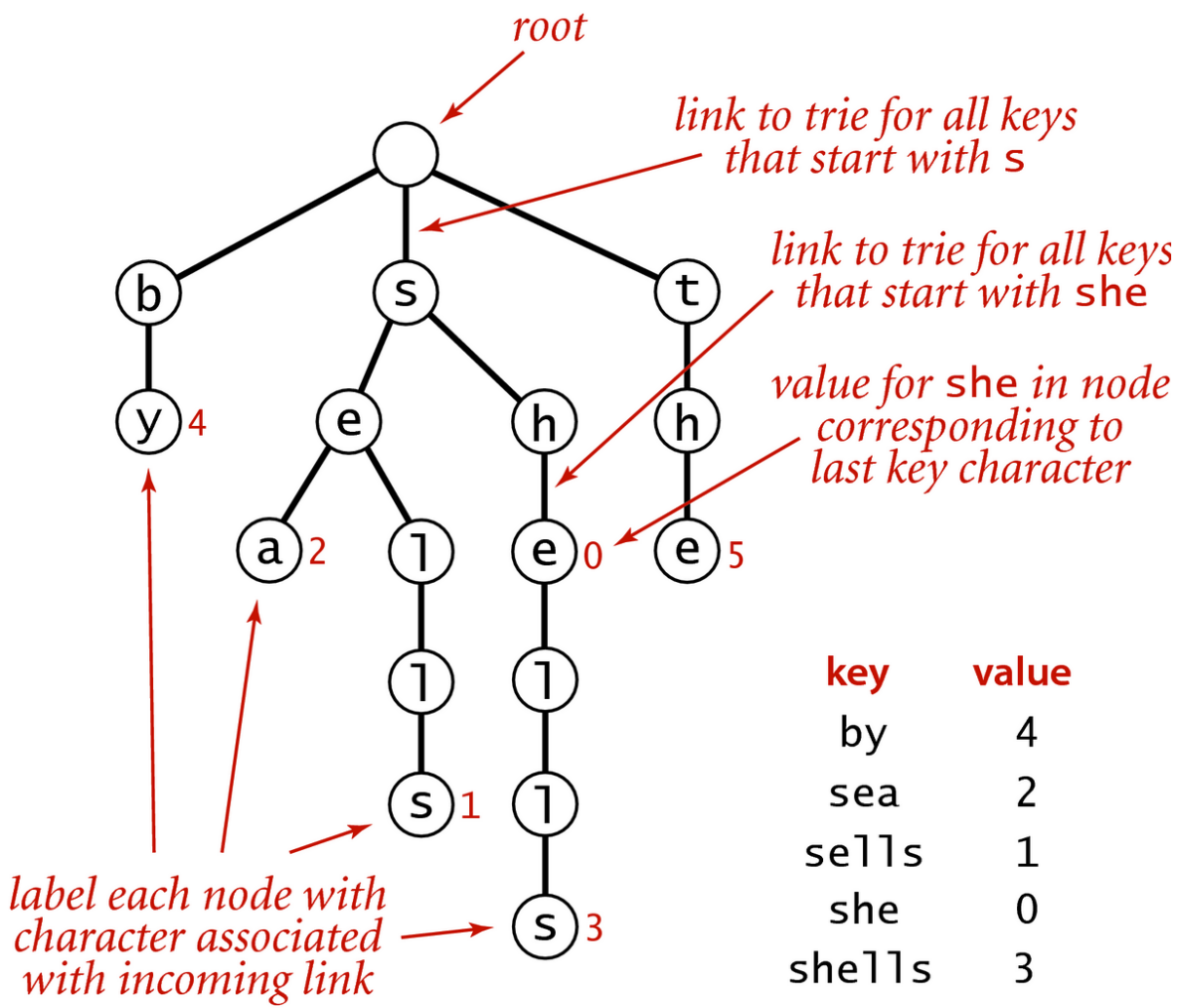
Outra representação da mesma trie





Trie representation

Anatomia de uma Trie



key	value
by	4
sea	2
sells	1
she	0
shells	3
the	5

Anatomy of a trie

Fonte: algs4

Observações

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.

Ao descer da raiz até um nó `x`, **soletramos** uma string, digamos `s`. Dizemos que `s` leva ao nó `x`.

Dizemos também que o nó `x` é localizado pela string `s`.

A string que leva a um nó `x` é uma chave se e somente se `x.val != null`.

Alfabeto

A implementação utiliza a classe `Alphabet`

Um **alfabeto** é um conjunto de caracteres ou símbolos.

Em Java, cada caractere é um número entre `0` e `65535`.

A classe `Alphabet` permite definir um alfabeto personalizado com `R` caracteres numerados de 0 a `R`-1.

`R` é a base do alfabeto.

Veja a página Alfabetos e a classe Alphabet (PF)

Alphabet API

public class	Alphabet	
	Alphabet(String s)	cria uma alfabeto com caracteres em s
int	toIndex(char c)	converte c em um índice em 0..R-1
char	toChar(int i)	converte i para o caractere correspondente
boolean	contains(char c)	c está no alfabeto?
int	R()	base do alfabeto
int	lgR()	número de bits de um índice

Alfabeto do DNA

```
Alphabet("ACTG")
```

c	A	C	T	G
toIndex(c)	0	1	2	3

i	0	1	2	3
toChar(i)	A	C	T	G

Alfabeto ASCII

Para o alfabeto ASCII temos simplesmente

- `toIndex(c)` == `c` para todo `c` e
- `toChar(i)` == `i` para todo `i`.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Fonte: Wikipedia

TrieST: esqueleto

```
public classTrieST<Value> {
   // tamanho do alfabeto
   private static int R = 256;
   // número de pares chave-valor
   private int n;
   private Node r; // raiz da trie
   private static class Node{...}
   public Value get(String key) {...}
   private Node get(Node x, String key, int d)
   {...}
   public void put(String key, Value val) {...}
   private Node put(Node x, String key,
            Value val, int d) {...}
   public void delete(String k) {...}
   public Iterable<String> keys() {...}
}
```

get(key): método clássico

Seguimos os ponteiros soletrando a string key.

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

get(key): método clássico

Seguimos os ponteiros soletrando a string key.

```
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);
}
```

put(key, val): método clássico

É feita uma busca.
Se a key é encontrada o valor val é substituido.
Caso contrário chegamos a um null e devemos continuar a inserção ou chegamos no último caractere de key.

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

put(key, val): método clássico

```
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;
}
```

`delete(key)`: método clássico

`delete(key)` remove a chave `key` do conjunto de chaves da trie.
Em princípio, a implementação de `delete()` é fácil: basta encontrar o nó `x` localizado por `key` e fazer

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

Infelizmente, a trie resultante dessa operação pode não ser *limpa*.

Para manter a `trie` limpa, é preciso fazer algo mais complexo.

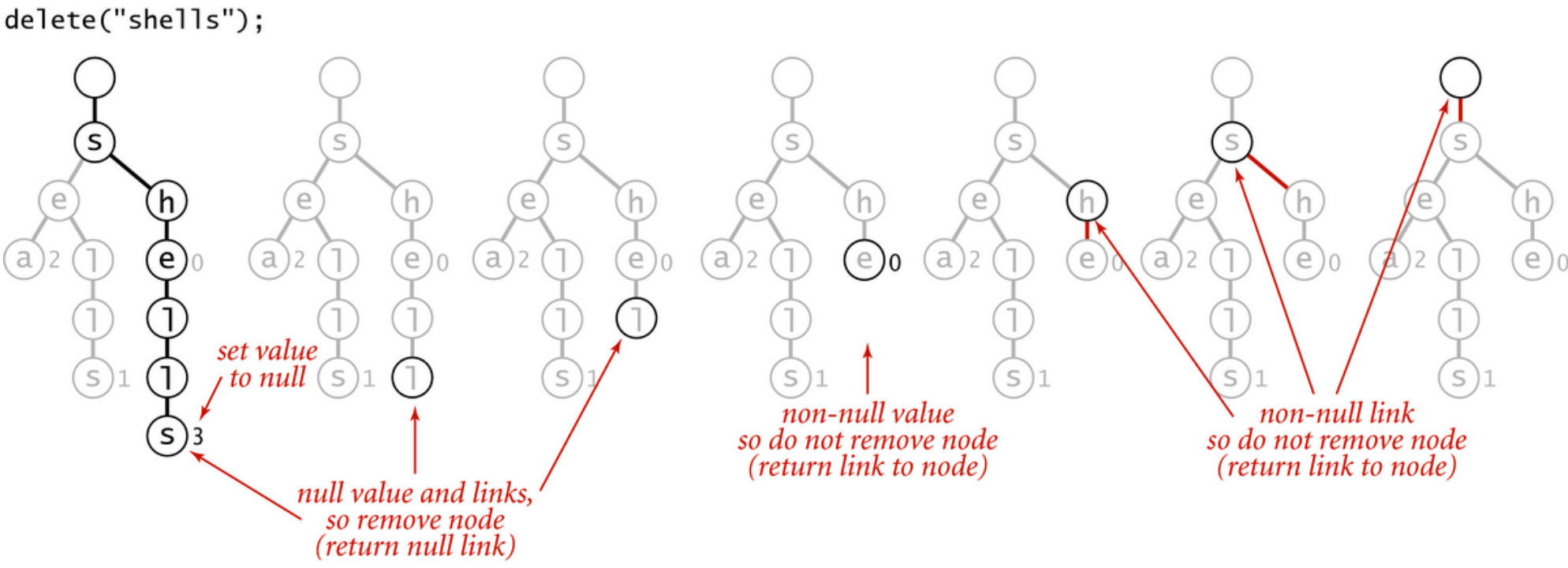
delete(key): método clássico

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

delete(key): método clássico

```
private Node delete(Node x, String key,
                    int d) {
   if (x == null) return null;
   if (d == key.length()) x.val = null;
   else {
       char c = key.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;
}
```

delete(key): ilustração



Deleting a key (and its associated value) from a trie

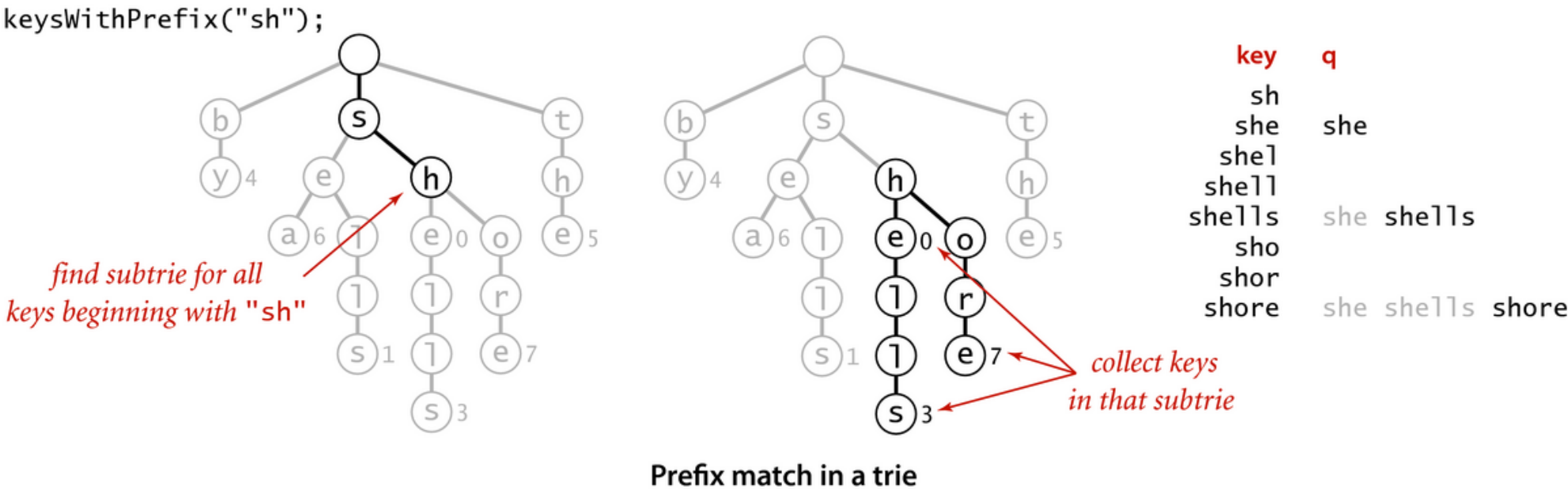
Fonte: algs4

keysWithPrefix(pre): método especial

Devolve todas as chaves na ST que têm prefixo pre.

```
public Iterable<String>
            keysWithPrefix(String pre) {
   Queue<String> q = new Queue<String>();
   Node x = get(r, pre, 0);
   collect(x, pre, q);
   return q;
}
```

`keysWithPrefix(pre)`: ilustração



Prefix match in a trie

Fonte: `algs4`

collect(): método auxiliar

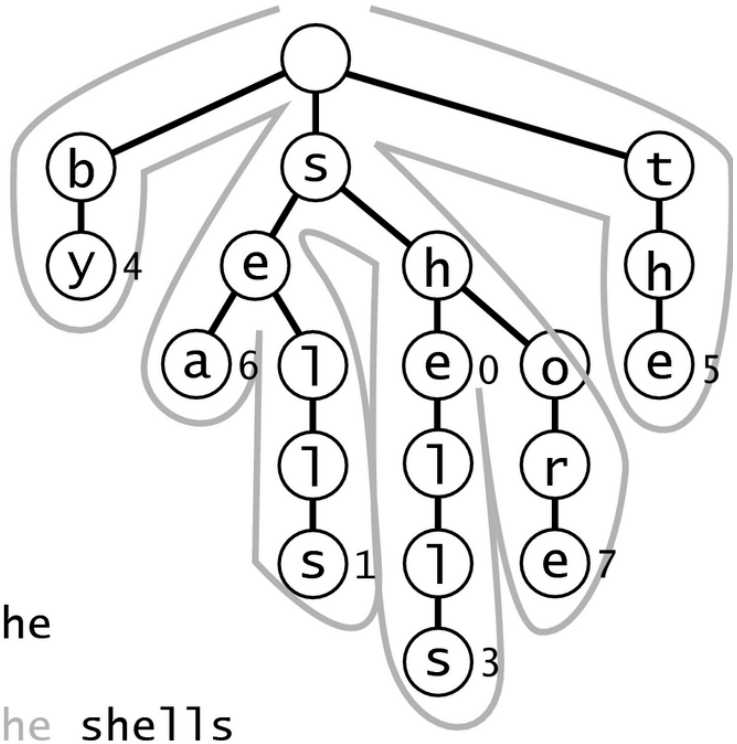
O método coloca na fila q todas as chaves da subtrie cuja raiz é x depois de acrescentar o prefixo pre a todas essas chaves.

```
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);
}
```

collect(): ilustração

```
keysWithPrefix("");
```

key	q
b	
by	by
s	
se	
sea	by sea
sel	
sell	
sells	by sea sells
sh	
she	by sea sells she
shell	
shells	by sea sells she shells
sho	
shor	
shore	by sea sells she shells shore
t	
th	
the	by sea sells she shells shore the



Collecting the keys in a trie (trace)

Fonte: algs4

keys(): método clássico

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

longestPrefixOf(s): método especial

Devolve a maior chave que é prefixo de s.

```
public String longestPrefixOf(String s) {
   int max = -1;
   Node x = r;
   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(): método especial

Devolve todas as chaves que casam com o padrão pat.
Todos os caracteres '.' em pat são curingas.

```
public Iterable<String>
keysThatMatch(String pat) {
   Queue<String> q = new Queue<String>();
   collect(r, "", pat, q);
   return q;
}
```

Mais um collect()

Coloca em q todas as chaves da trie que têm prefixo pre e casam com o padrão pat.

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

Consumo de espaço e tempo

A estrutura de uma `trie` não depende da ordem em ques as chaves são inseridas ou removidas.

O consumo de tempo das operações sobre uma `trie` não depende do número `n` de itens.

O número de nós visitados por `get(key)` é no máximo $1 + w$, onde $w =$ `key.length()`.

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

Consumo de espaço e tempo

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 $\log_{\texttt{R}} \texttt{n}$.

Tries ternárias

Fonte: The Little Prince, Antoine de Saint-Exupéry

Referências: Tries (árvores digitais) (PF); Tries (S&W); slides (S&W); Vídeo (S&W); TAOCP, vol 3, cap. 6.3

Tries ternárias

O maior problema das `tries` é possivelmente o espaço, já que cada nó contém `R` referências.

Assim, cada nó utiliza pelo menos $8 \times$ `R` bytes.

Veja alguns valores de `R` para de alguns alfabeto na classe `Alphabet`‘ na página Alfabetos e a classe Alphabet (PF).

Para evitar o custo excessivo de espaço associamos de uma `R-trie`, consideramos uma representação como uma **ternary search trie** (`TST`).

Alphabet

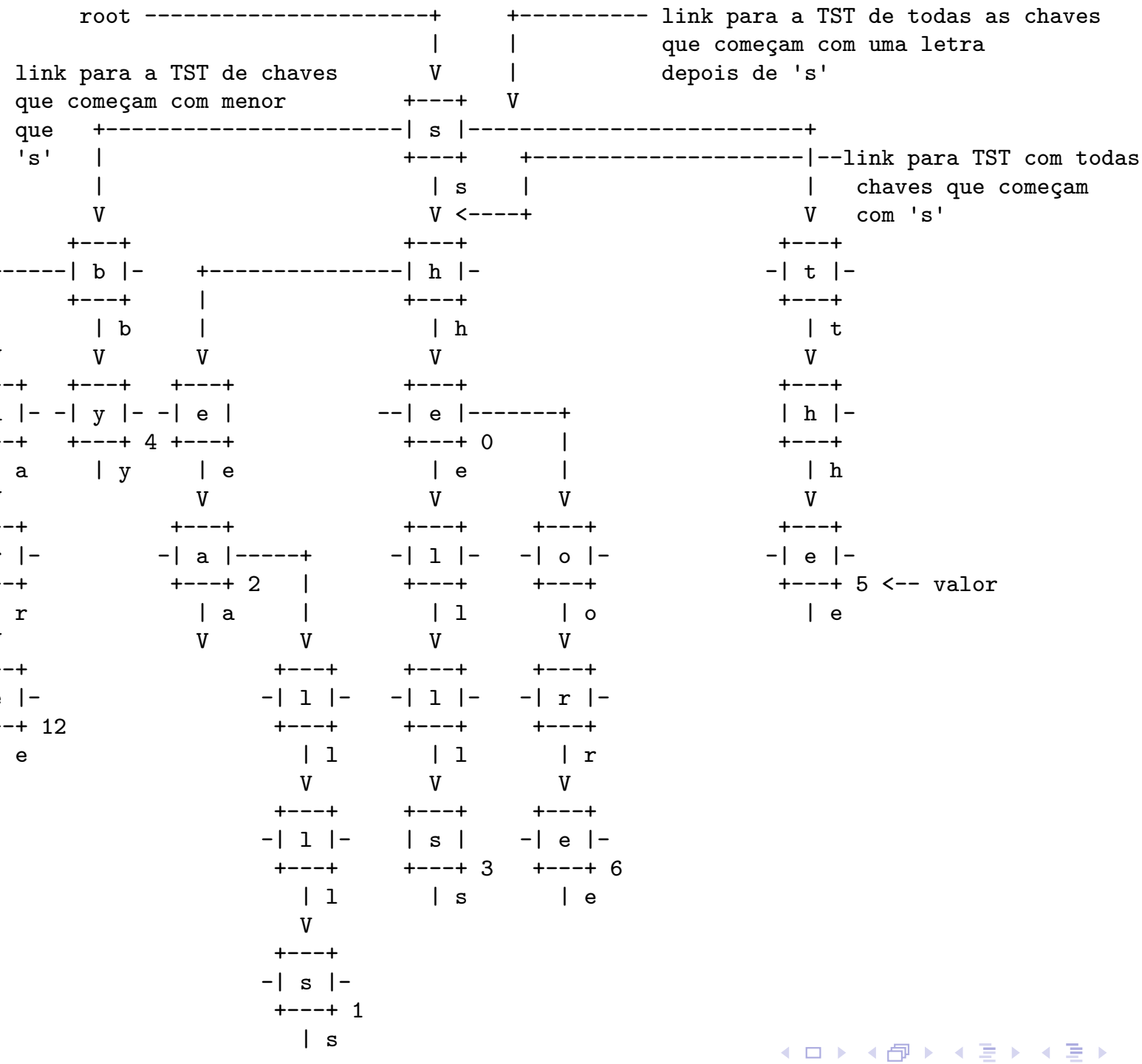
nome	R()	lg(R)	conjunto de caracteres
BINARY	2	1	'01'
DNA	4	2	'ACTG'
OCTAL	8	3	'01234567'
DECIMAL	10	4	'0123456789'
HEXADECIMAL	16	4	'0123456789ABCDEF'
PROTEIN	20	5	'ACDEFGHIKLMNPQRSTVW
LOWERCASE	26	5	'abcd...wxyz'
UPPERCASE	26	5	'ABCD...WXYZ'
ASCII	128	7	alfabeto ASCII
EXTENDED_ASCII	256	8	alfabeto ASCII estendido
UNICODE16	65536	16	alfabeto Unicode

Outra ilustração

Para os pares `key`-`val`

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

temos a `TST` a seguir


root
link para a TST de chaves que começam com menor que 's'
link para a TST de todas as chaves que começam com uma letra depois de 's'
link para TST com todas chaves que começam com 's'
b
a
y
4
e
a
2
l
l
s
1
s
h
e
0
l
l
s
3
s
t
h
e
5 <-- valor
o
r
e
6
e
r
e
12
e

TSTs

De maneira semelhante ao que ocorre com `tries`, nas `tries` ternárias:

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

Estrutura de uma trie ternária

Os links da estrutura correspondem a caracteres.
Nas figuras, o caractere escrito dentro de um nó é o caractere do link que sai pelo meio do nó.
TSTs são compostas por nós do tipo Node.

```
private static class Node {
   private char c; // caractere
   private Value val;
   private Node left; // < c
   private Node mid; // == c
   private Node right; // > c
}
```

Ilustração

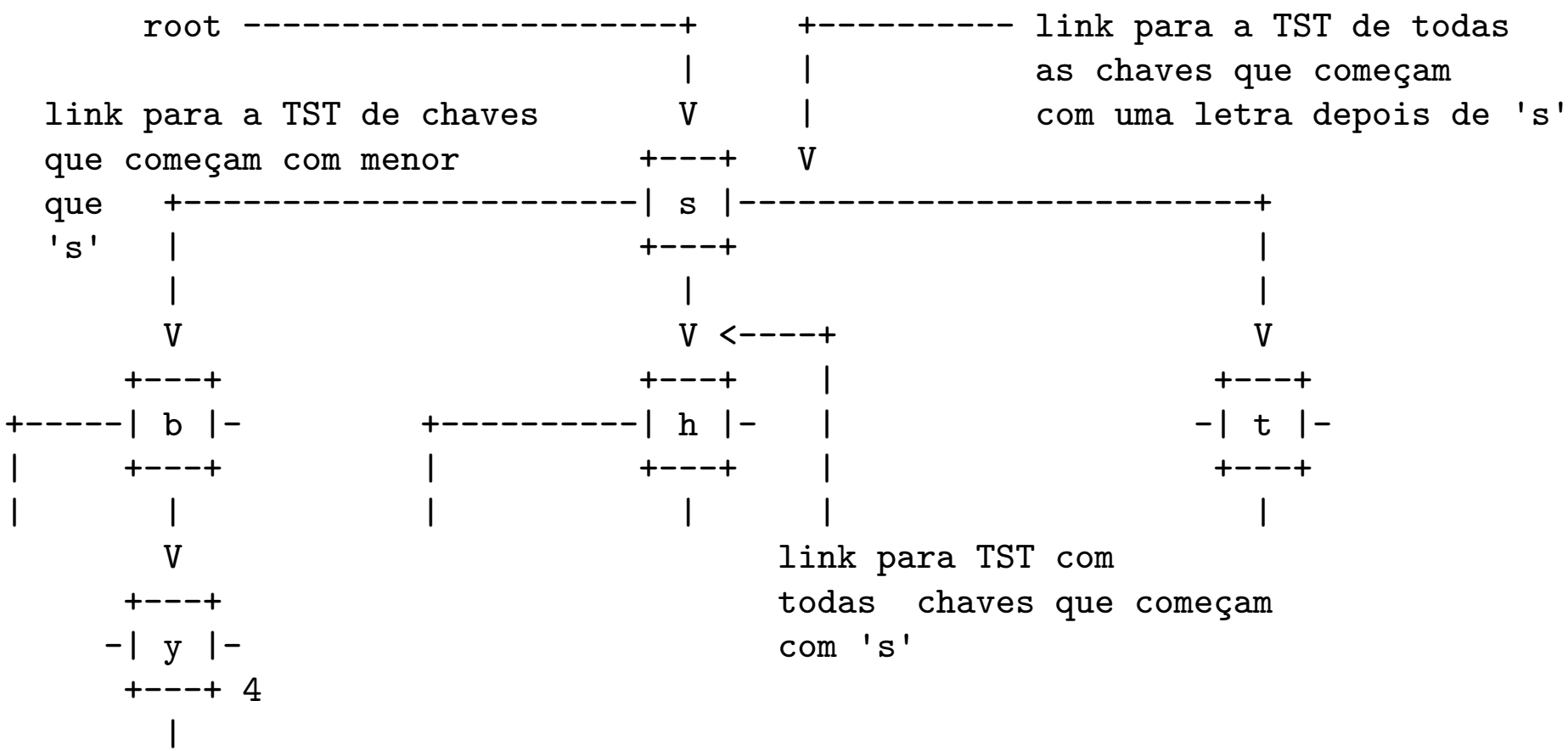
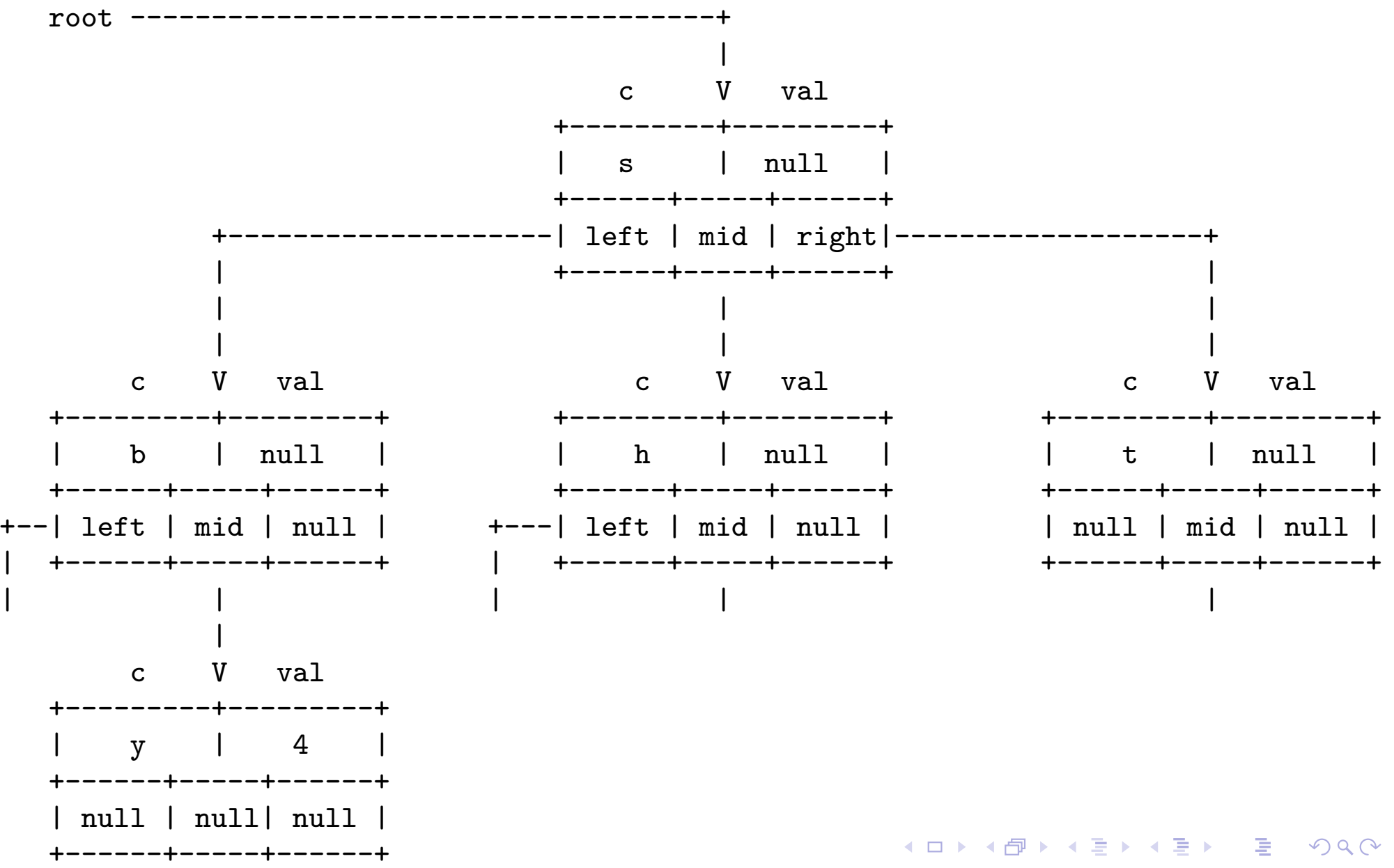
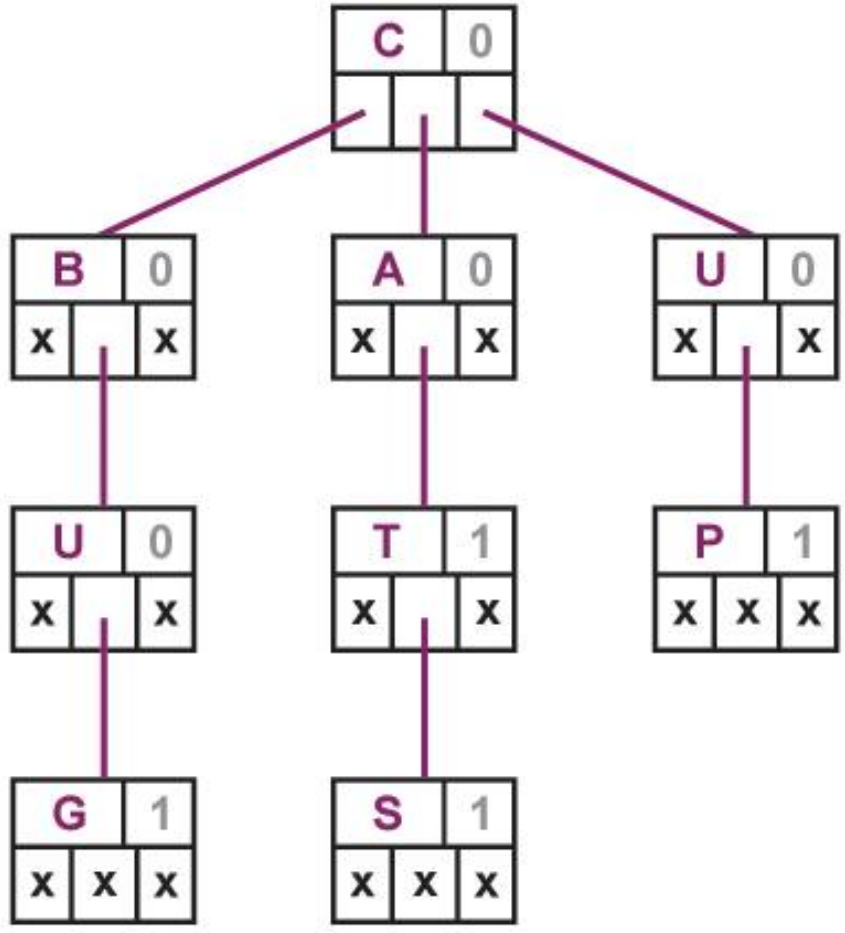

root
link para a TST de chaves que começam com menor que 's'
link para a TST de todas as chaves que começam com uma letra depois de 's'
s
b
h
t
y
4
link para TST com todas chaves que começam com 's'

Ilustração com zoom



Outra ilustração



Ternary Search Tree for CAT, BUG, CATS, UP

Fonte: Ternary search trees for autocompletion and spell checking

TST: esqueleto

```
public classTST<Value> {
   // tamanho do alfabeto
   private static int R = 256;
   private int n;
   private Node r; // raiz da tst
   private static class Node{...}
   public Value get(String key) {...}
   private Node get(Node x, String key, int d)
   {...}
   public void put(String key, Value val) {...}
   private Node put(Node x, String key,
            Value val, int d) {...}
   public Iterable<String> keys() {...}
}
```

get(key): método clássico

A string que leva a um nó x é uma chave se e somente se x.c é o último caractere da chave e x.val != null.

```
// TrieST e TST
public Value get(String key) {
   Node x = get(r, key, 0);
   if (x == null) return null;
   return x.val;
}
```

get(key): método clássico

```
private Node get(Node x, String key,
                 int d) {
   char c = key.charAt(d);
   if (x == null) return null;
   if (c < x.c)
       return get(x.left, key, d);
   if (c > x.c)
       return get(x.right, key, d);
   if (d < key.length()-1)
       return get(x.mid, key, d+1);
   return x;
}
```

put(key, val): método clássico

É feita uma busca.
Se a key é encontrada o valor val é substituido.
Caso contrário chegamos a um null e devemos continuar a inserção ou chegamos no último caractere de key.

```
// TrieST e TST
public void put(String key, Value val) {
   r = put(r, key, val, 0);
}
```

put(key, val): método clássico

```
private Node put(Node x, String key,
                 Value val, int d) {
   char c = key.charAt(d);
   if (x == null)
       { x = new Node(); x.c = c; }
   if (c < x.c)
       x.left = put(x.left, key, val, d);
   else if (c > x.c)
       x.right = put(x.right,key, val, d);
   else if (d < key.length()-1)
       x.mid = put(x.mid, key, val, d+1);
   else x.val = val;
   return x;
}
```

collect(): método auxiliar

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;
   collect(x.left, pre, q);
   // ordem lexicográfica
   if (x.val != null) q.enqueue(pre+x.c);
   collect(x.mid, pre+x.c, q);
   collect(x.right, pre, q);
}
```

keys(): método clássico

```
public Iterable<String> keys() {
   Queue<String> q = new Queue<String>();
   collect(r, "", q);
   return q;
}
```

keysWithPrefix(): método especial

Devolve todas as chaves na ST que têm prefixo pre.

```
public Iterable<String>
            keysWithPrefix(String pre) {
   Queue<String> q = new Queue<String>();
   Node x = get(r, pre, 0);
   if (x == null) return q;
   if (x.val != null) q.enqueue(pre);
   collect(x.mid, pre, q);
   return q;
}
```

`longestPrefixOf()`: método especial

Devolve a maior chave que é prefixo de `s`.

```
public String longestPrefixOf(String s) {
   if (s == null || s.length()==0)
       return null;
   int max = 0;
   Node x = r;
   int i = 0;
```

longestPrefixOf(): método especial

```
    while (x != null && i < s.length()) {
        char c = s.charAt(i);
        if (c < x.c) x = x.left;
        else if (c > x.c) x = x.right;
        else {
            i++;
            if (x.val!= null) max = i;
            x = x.mid;
        }
    }
    return s.substring(0, max);
}
```

keysThatMatch(): método especial

Devolve todas as chaves que casam com o padrão pat. Todos os caracteres '.' em pat são curingas.

```
public Iterable<String>
keysThatMatch(String pat) {
   Queue<String> q = new Queue<String>();
   collect(r, "", 0, pat, q);
   return q;
}
```

Mais um collect()

Coloca em q todas as chaves da trie que têm prefixo pre e casam com o padrão pat.

```
private void collect(Node x, String pre,
      int i,String pat,Queue<String> q){
   if (x == null) return;
   char c = pat.charAt(i);
   if (c == '.'  || c < x.c)
      collect(x.left, pre, i, pat, q);
```

Mais um collect()

```
    if (c == '.'  || c == x.c) {
        if (i == pat.length() - 1
            && x.val != null)
            q.enqueue(pre+x.c);
        else if (i < pat.length() - 1)
            collect(x.mid,pre+x.c,i+1,pat,q);
    }
    if (c == '.'  || c > x.c)
        collect(x.right, pre, i, pat, q);
}
```

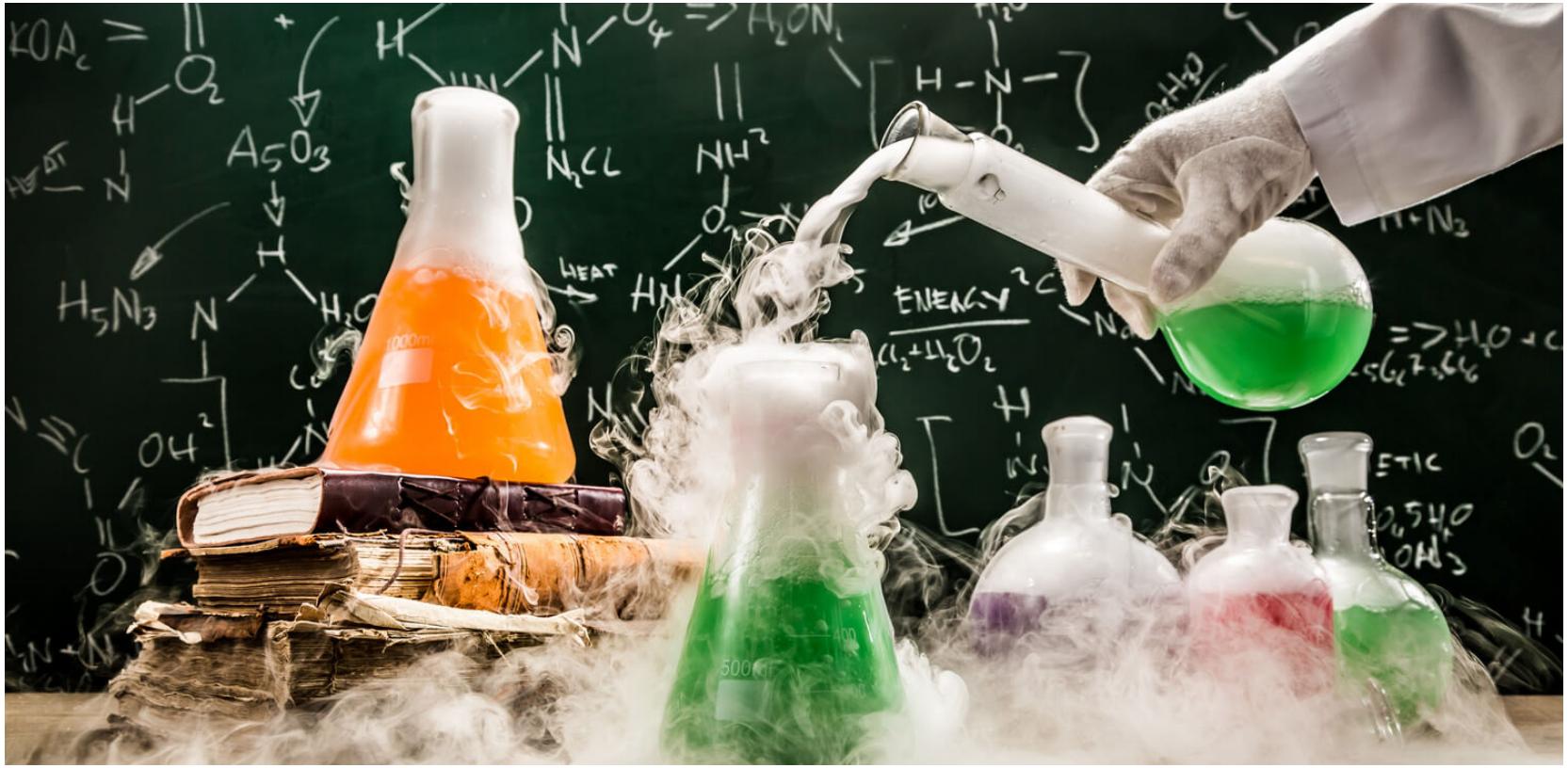
Consumo de espaço e tempo

Espaço. A propriedade mais importante de uma TST é que ela tem apenas três links por nó.

Proposição J: O número de links em uma TST com n chaves de comprimento médio w é entre 3n e 3nw.

Proposição K: O número esperado de nós visitados durante uma busca malsucedida em uma TST com n chaves aleatórias é aproximadamente $\lg n$.

Alguns experimentos



Fonte: https://singularityhub.com/

Experimentos: les_miserables.txt

ST com 26764 itens

```
% wc les-miserables.txt
68116 568531 3322649 les-miserables.txt
```

ST	criada (s)
BST	0.626
RedBlackBST	0.577
SeparateChainST	0.495 (4096, 18, $\alpha = 6$)
LinearProbingST	0.411 (65536, 67,$\alpha = 0.40$)
TrieST	0.574
TST	0.497

Experimentos: actors.list

ST com 1482495 itens

```
% wc actors.list
16612200 124796815 932688622 actors.list
```

ST	criada (s)
BST	141.968
RedBlackBST	168.723
SeparateChainST	110.035 (262144, 19, $\alpha = 5$)
LinearProbingST	73.123 (4194304, 4787 $\alpha = 0.35$)
TrieST	OutOfMemoryError
TST	107.223