Técnicas de Busca e Ordenação (TBO)

Tries e TST

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides dos Professores Eduardo Zambon e Giovanni Comarela)

Introdução

- É possível incorporar as ideias do *radix sort* no contexto de busca.
- Isso dá origem às estruturas chamadas *tries*, que analisam as chaves por partes.
- **Aula de hoje**: mostrar variações de *tries*.
- **Objetivos**: compreender o funcionamento e as diferentes implementações de *tries*.

Referências

Chapter 15 – Radix Search
R. Sedgewick

Implementações de tabelas de símbolos: sumário

implementation	typical case			ordered operations
	search	insert	delete	
red-black BST	$\log N$	$\log N$	$\log N$	✔
hash table	1 †	1 †	1 †	

† under uniform hashing assumption

Q: É possível fazer melhor?

A: Sim, se evitarmos de olhar a chave toda, como no *radix sort*.

Suposição: chaves das tabelas de símbolos são *strings*.

Implementações de tabelas de símbolos: sumário

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + (L/2) \lg N$	$(L/2) \lg N$	$(L/2) \lg N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6

Parameters

- N = number of strings
- L = length of string
- R = radix

file	size	words	distinct
moby.txt	1.2 MB	210 K	32 K
actors.txt	82 MB	11.4 M	900 K

- Tempo RBT: assume que cada string é comparada até a metade.
- Tempo *hash*: só precisa calcular o *hash* da *string*.
- Espaço RBT: cada nó tem até 4 ponteiros: `k`, `v`, `l`, `r`.
- Espaço *hash*: dois *arrays* com tamanho $2N$.

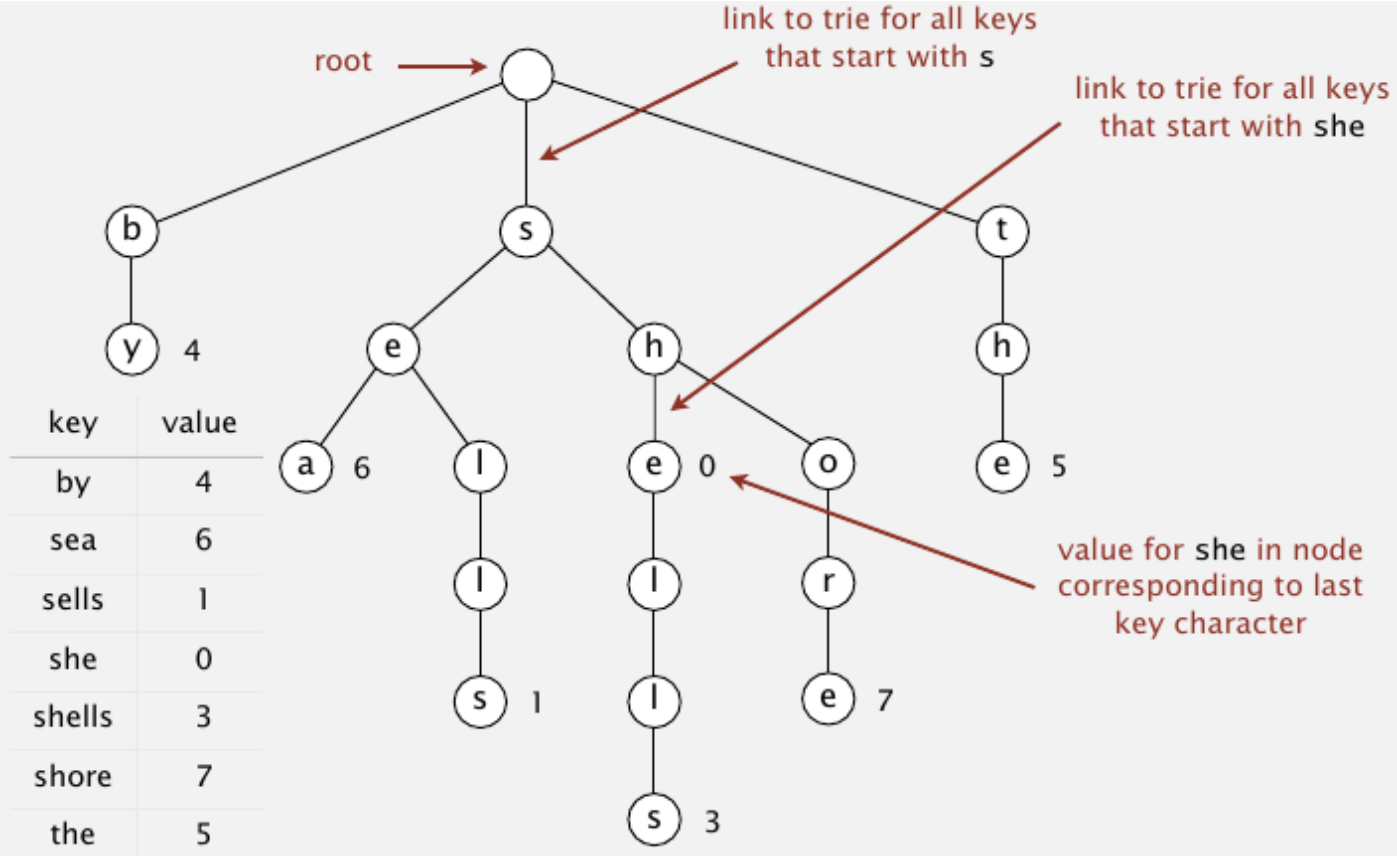
Part I

R-way tries

Tries

Tries: nome vem de *retrieval* mas se pronuncia "*try*".

- Cada nó armazena um caractere da chave.
- Um nó pode armazenar um valor.
- Cada nó tem R filhos, um para cada caractere possível.
- Suposição: ASCII estendido, radix $R = 256$.

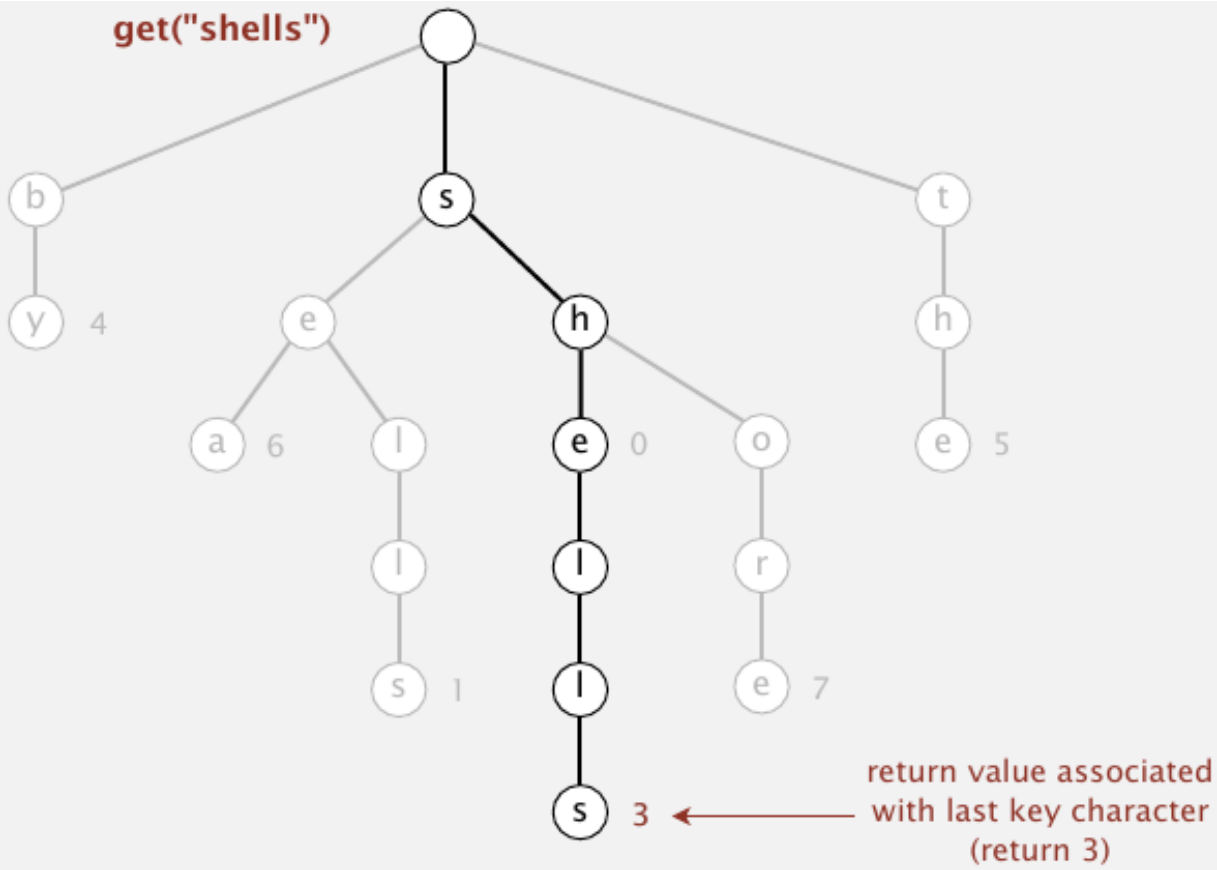


key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

Trie: busca

Percorra os *links* correspondentes a cada caractere da chave.

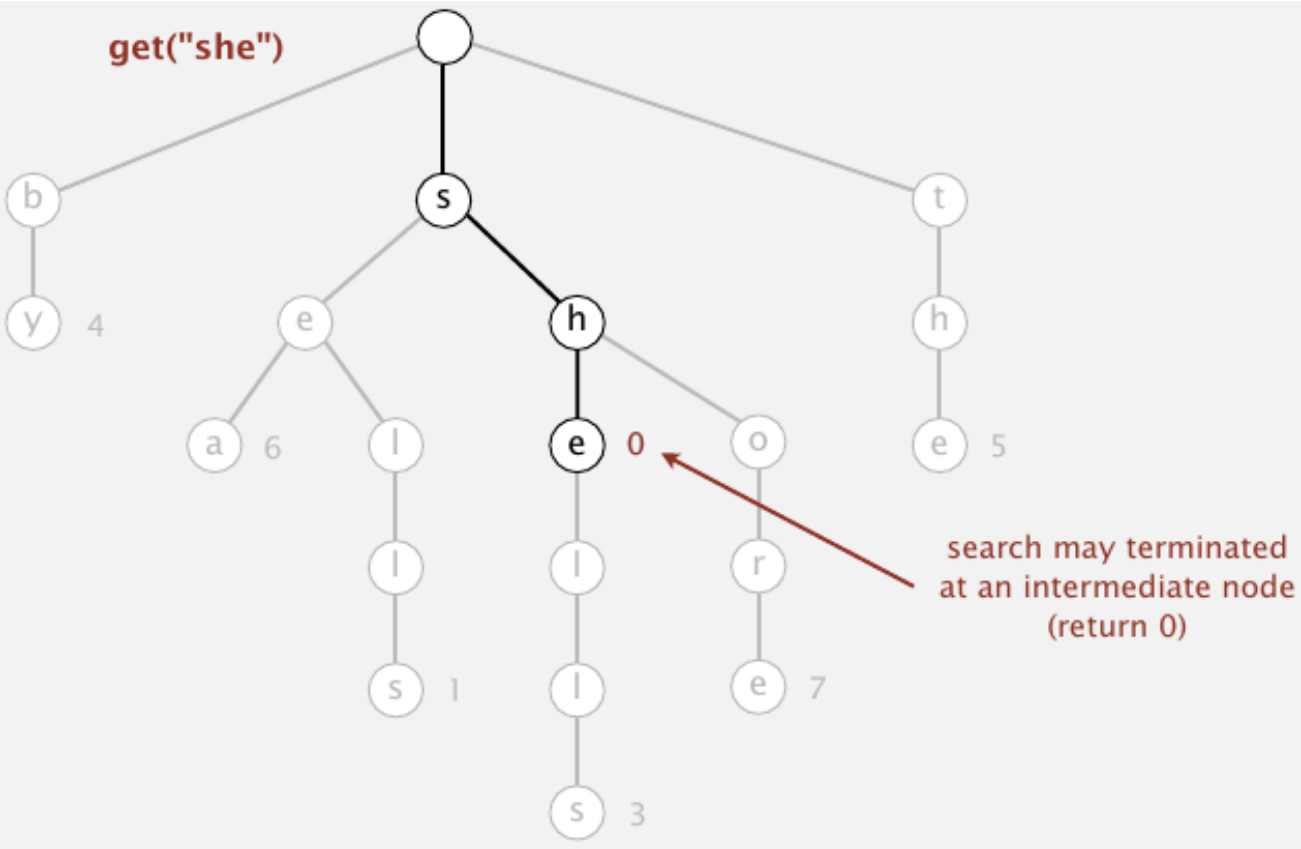
- *Search hit*: nó aonde a busca para tem um valor não-nulo.
- *Search miss*: *link* nulo ou nó tem um valor nulo.



Trie: busca

Percorra os *links* correspondentes a cada caractere da chave.

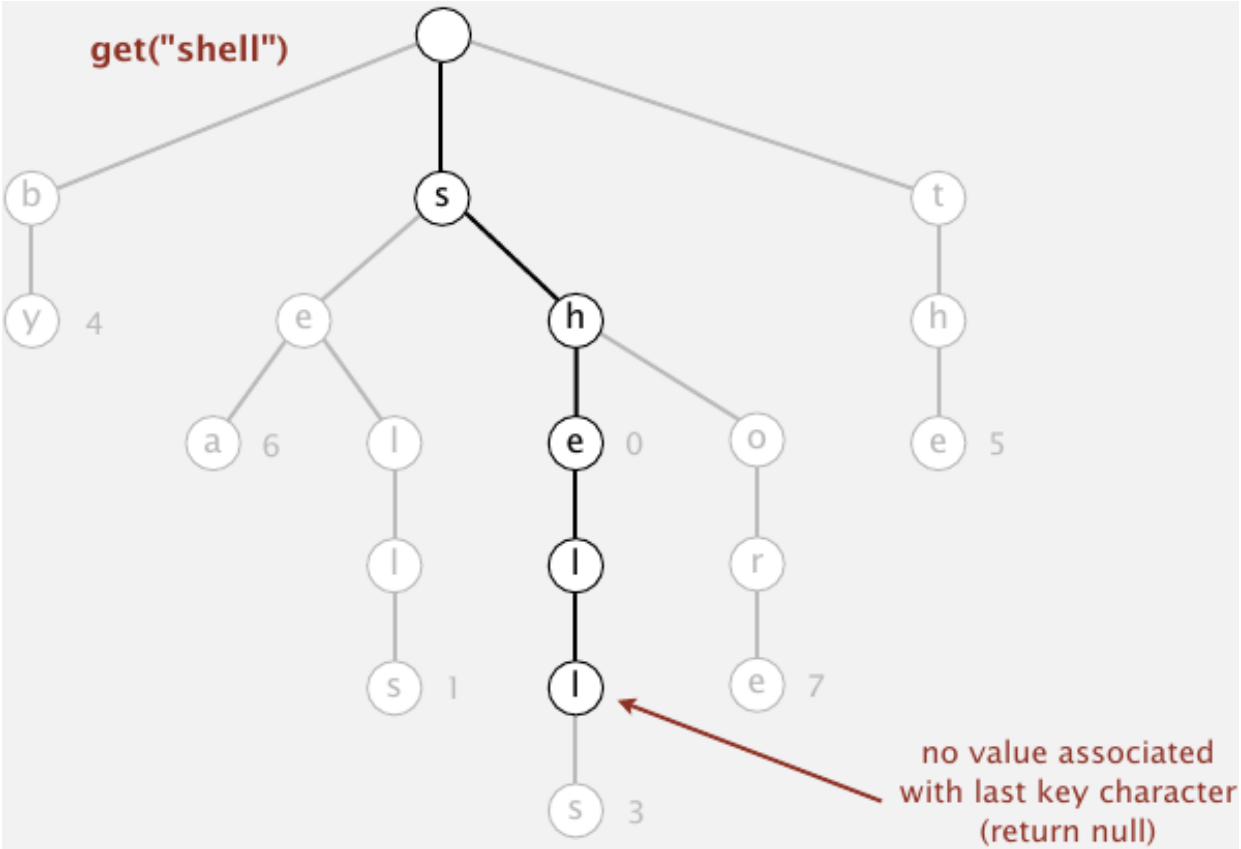
- *Search hit*: nó aonde a busca para tem um valor não-nulo.
- *Search miss*: *link* nulo ou nó tem um valor nulo.



Trie: busca

Percorra os *links* correspondentes a cada caractere da chave.

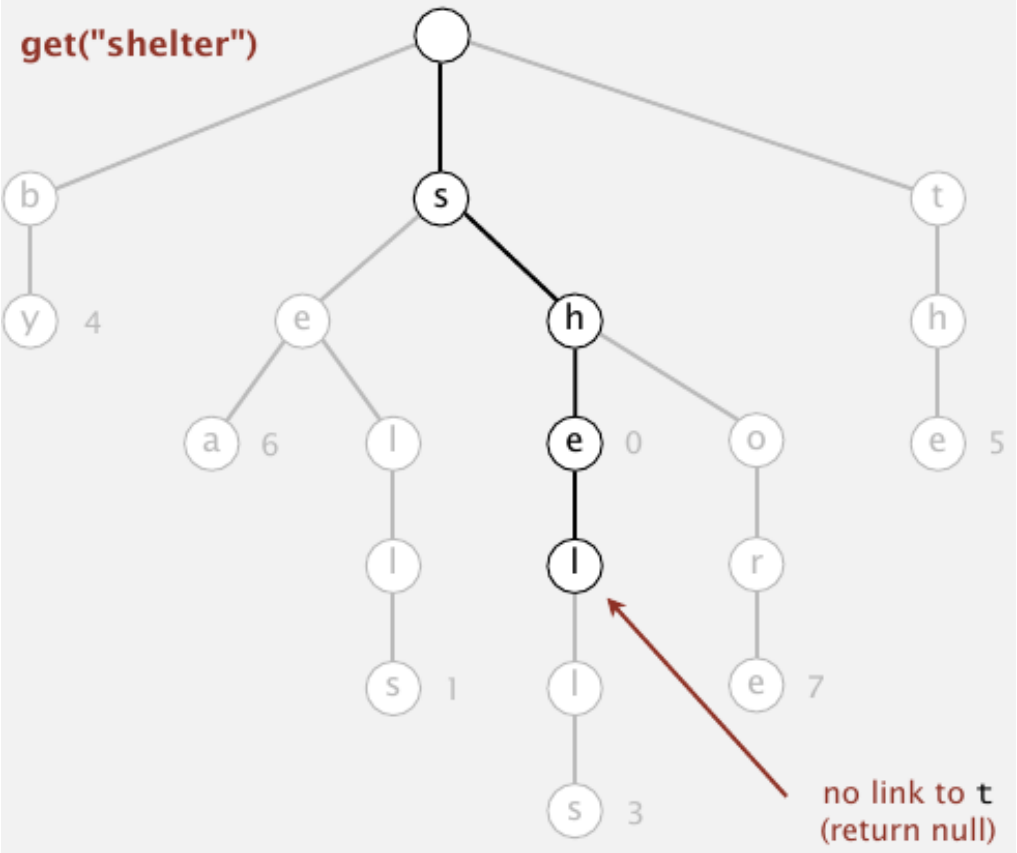
- *Search hit*: nó aonde a busca para tem um valor não-nulo.
- *Search miss*: *link* nulo ou nó tem um valor nulo.



Trie: busca

Percorra os *links* correspondentes a cada caractere da chave.

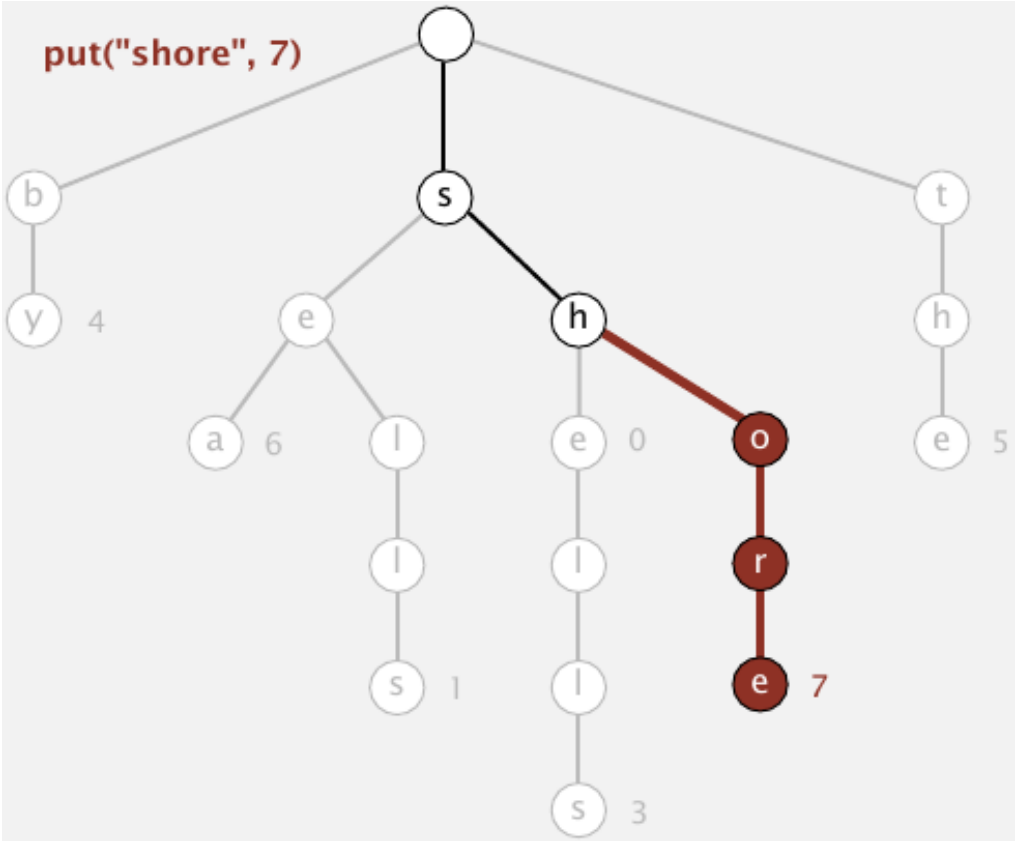
- *Search hit*: nó aonde a busca para tem um valor não-nulo.
- *Search miss*: *link* nulo ou nó tem um valor nulo.



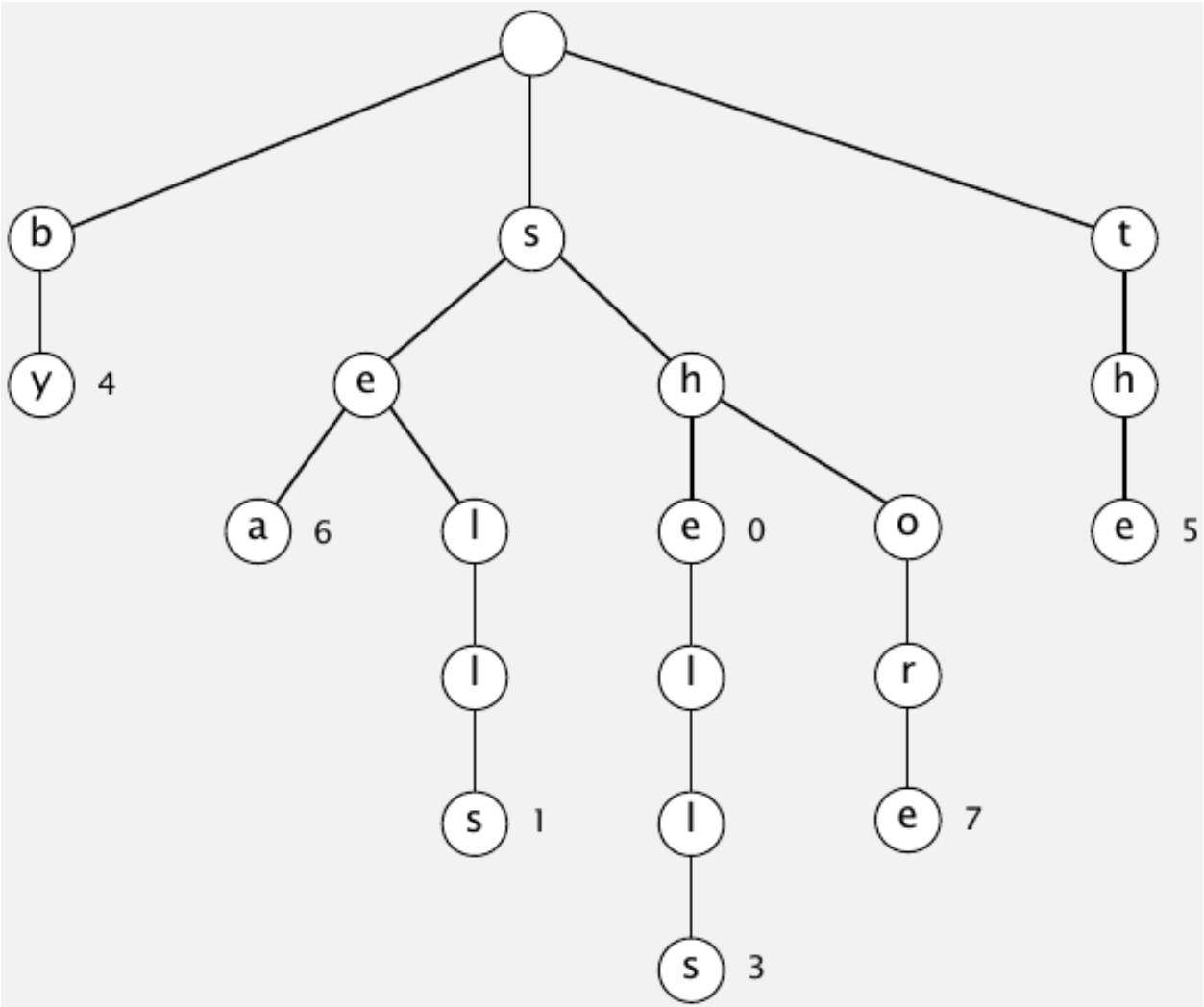
Trie: inserção

Percorra os *links* correspondentes a cada caractere da chave.

- Encontrou um *link* nulo: cria um novo nó.
- Encontrou último caractere da chave: modifica valor no nó.



Trie: demo



Veja o arquivo `52DemoTrie.mov`

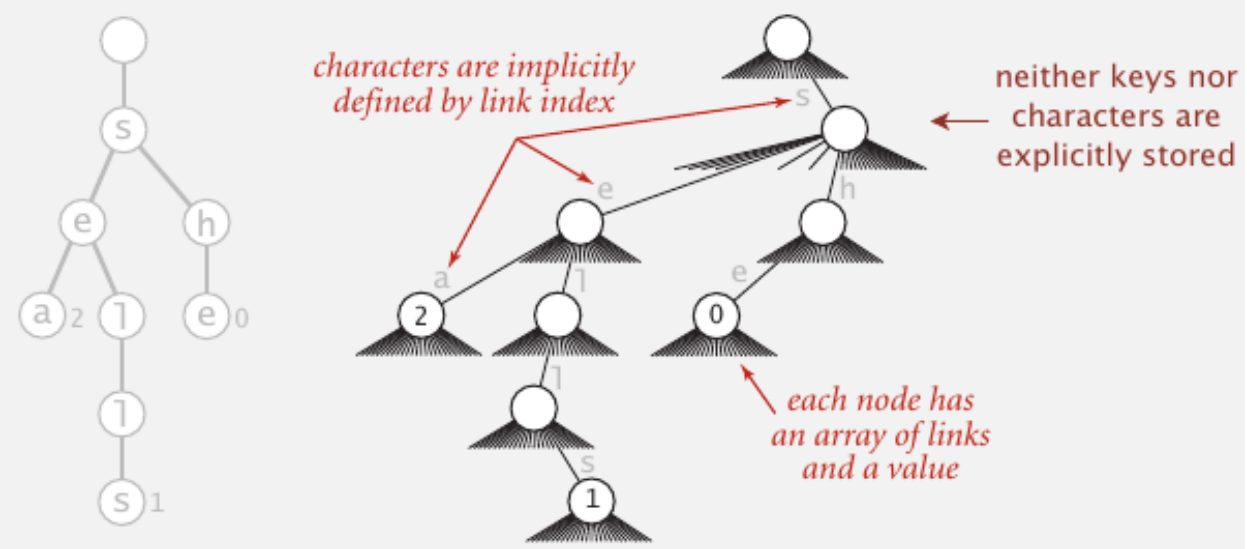
Representação de *trie*: implementação em C

Nó: um (possível) valor, mais *R* ponteiros para os filhos.

```
typedef struct node Trie;

#define R 256

struct node {
    Value val;
    Trie* next[R];
};
```



R-way trie: implementação em C

```
Trie* rec_insert(Trie* t, String* key, Value val, int d) {
    if (t == NULL)      { t = create_node(); }
    if (d == key->len) { t->val = val; return t; }
    unsigned char c = key->c[d];
    t->next[c] = rec_insert(t->next[c], key, val, d+1);
    return t;
}
Trie* Trie_insert(Trie* t, String* key, Value val) {
    return rec_insert(t, key, val, 0);
}

Trie* rec_search(Trie* t, String* key, int d) {
    if (t == NULL)      { return NULL; }
    if (d == key->len) { return t; }
    unsigned char c = key->c[d];
    return rec_search(t->next[c], key, d+1);
}
Value Trie_search(Trie* t, String* key) {
    t = rec_search(t, key, 0);
    if (t == NULL) { return NULL_Value; }
    else           { return t->val;     }
}
```

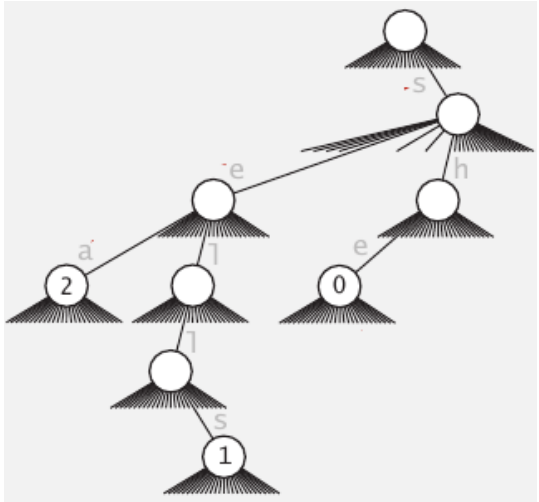
Trie: desempenho

Search hit: Precisa examinar todos os *L* caracteres para determinar igualdade.

Search miss.

- Pode diferir já no primeiro caractere.
- Caso típico: só precisa examinar alguns poucos caracteres (sublinear em *L*).

Espaço: *R* ponteiros nulos em cada folha.

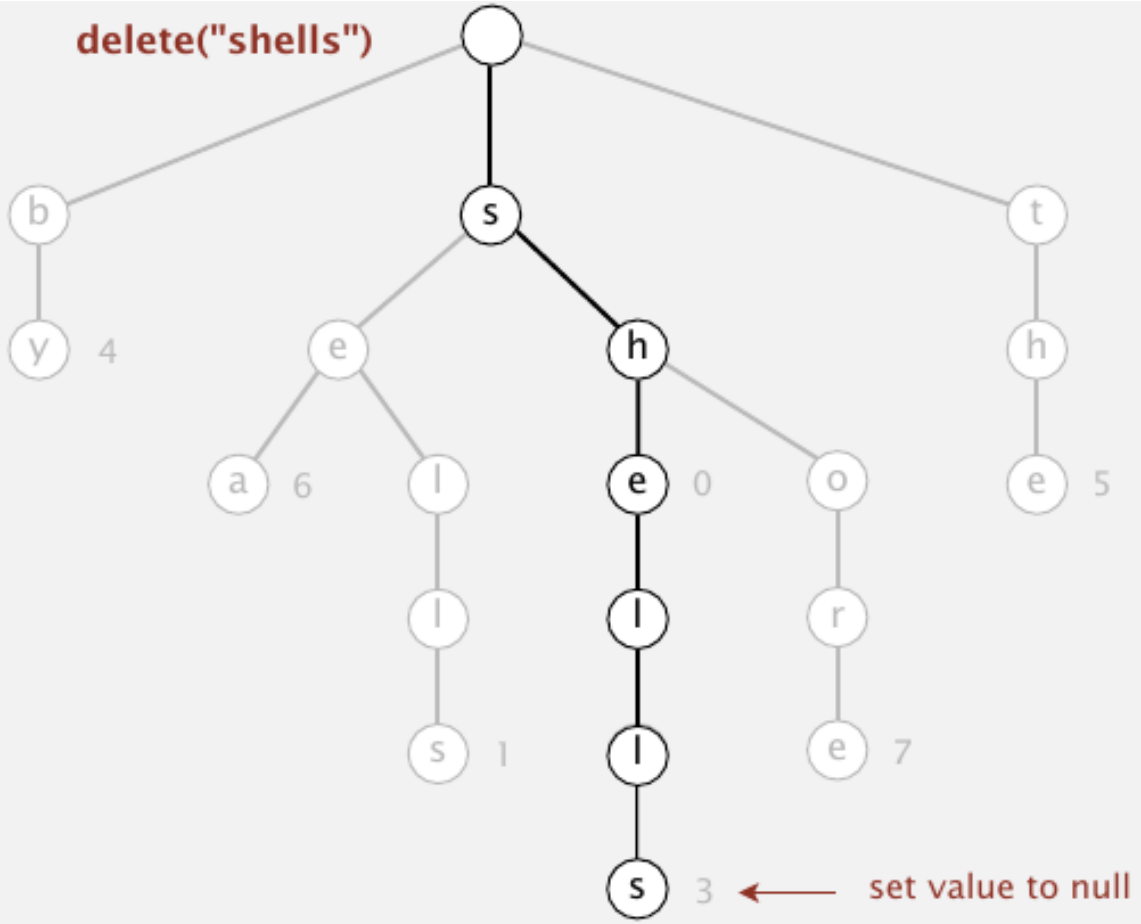


Conclusão: busca muito eficiente mas desperdiça espaço.

R-way trie: remoção

Para remover um par chave–valor:

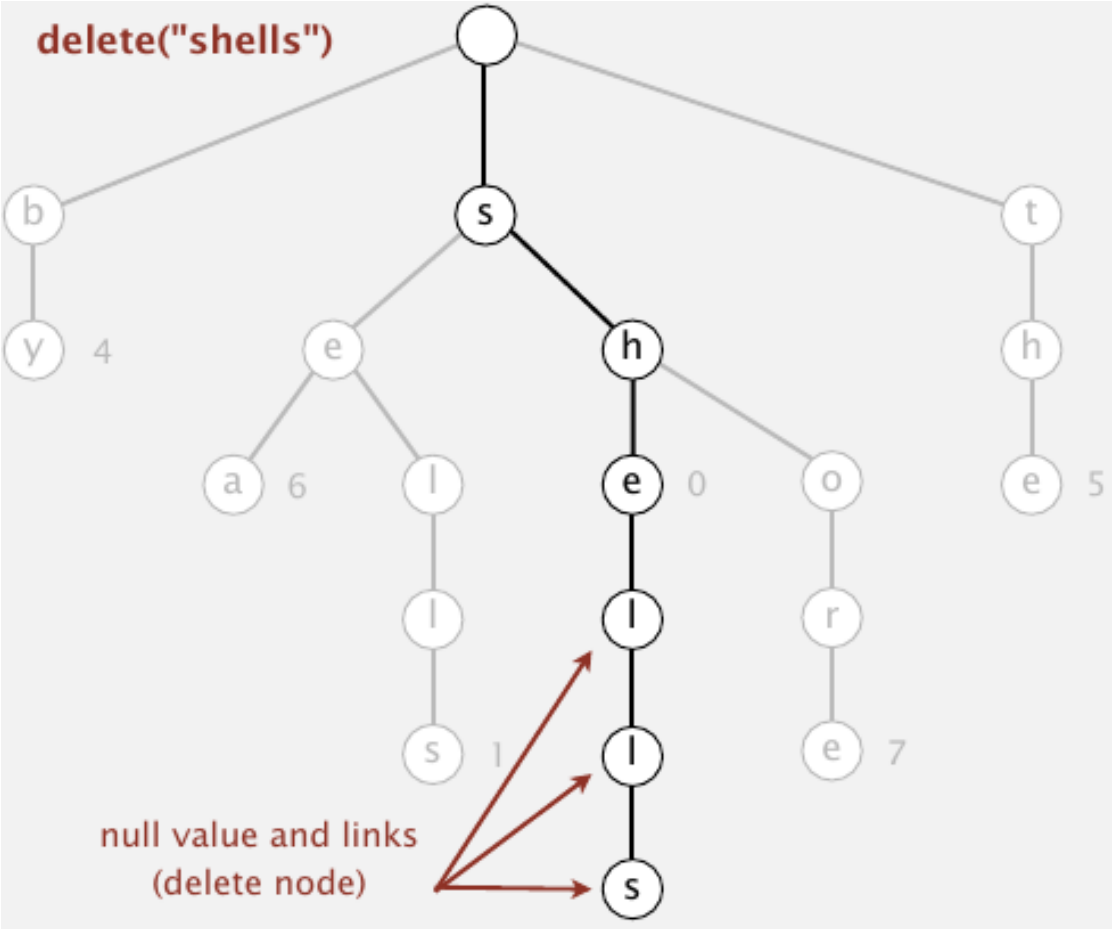
- Encontre nó que contém o valor e coloque `NULL_Value`.
- Se nó tem *link* e valor nulos: remova o nó e repita com pai.



R-way trie: remoção

Para remover um par chave–valor:

- Encontre nó que contém o valor e coloque `NULL_Value`.
- Se nó tem *link* e valor nulos: remova o nó e repita com pai.



Implementações de tabelas de símbolos: sumário

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + (L/2) \lg N$	$(L/2) \lg N$	$(L/2) \lg N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\ N$	1.12	*out of memory*

R-way trie.

- Melhor método para R pequeno: examina o mínimo possível de caracteres.
- Muito desperdício de memória para R grande.

Desafio: Usar menos memória.

Part II

Ternary search tries

Ternary search tries

Ternary search tries (TSTs) [Bentley & Sedgewick, 1997]:

- Cada nó armazena: um caractere da chave, e talvez um valor.
- Cada nó tem 3 filhos: menores (esquerda), iguais (meio), e maiores (direita).

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley*   Robert Sedgewick#

Abstract

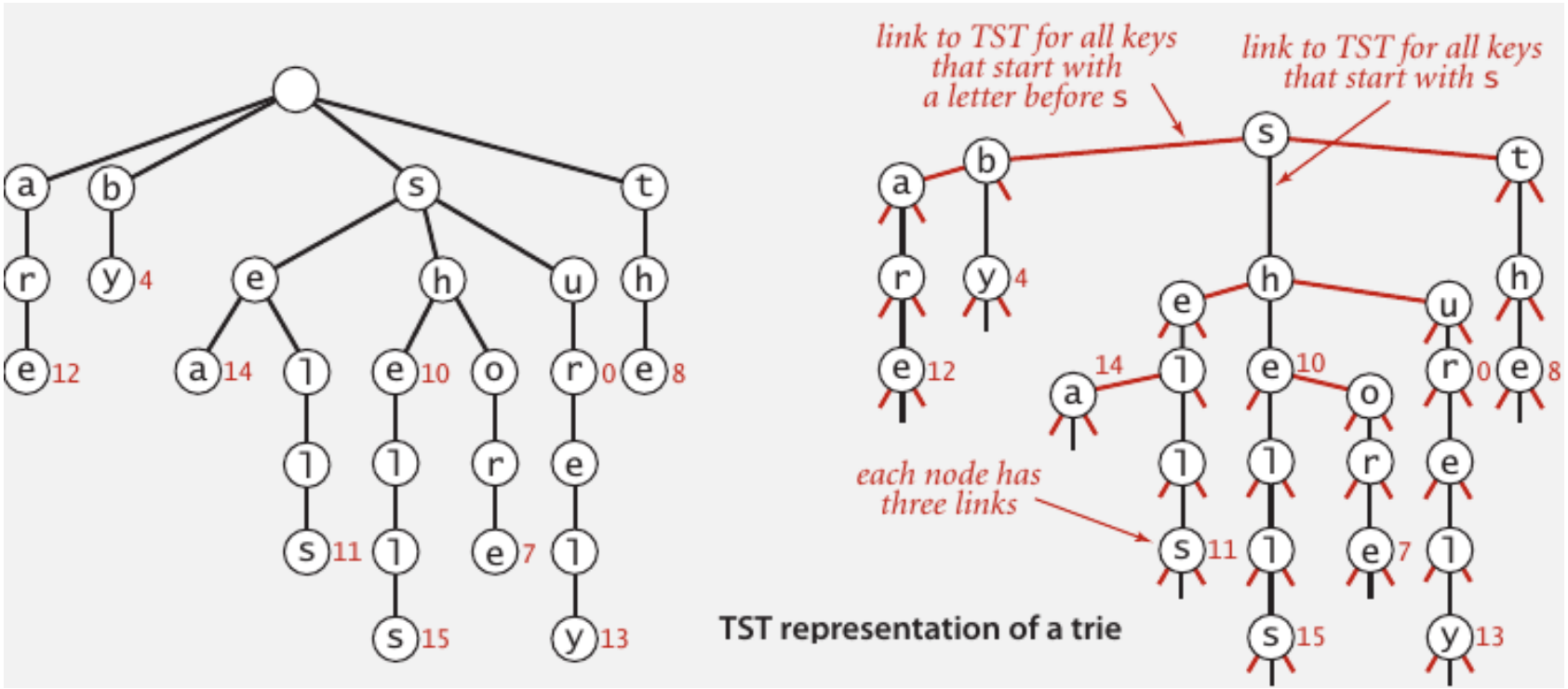
We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary search trees; it is faster than hashing and other commonly used search methods. The basic ideas behind the algo- that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,

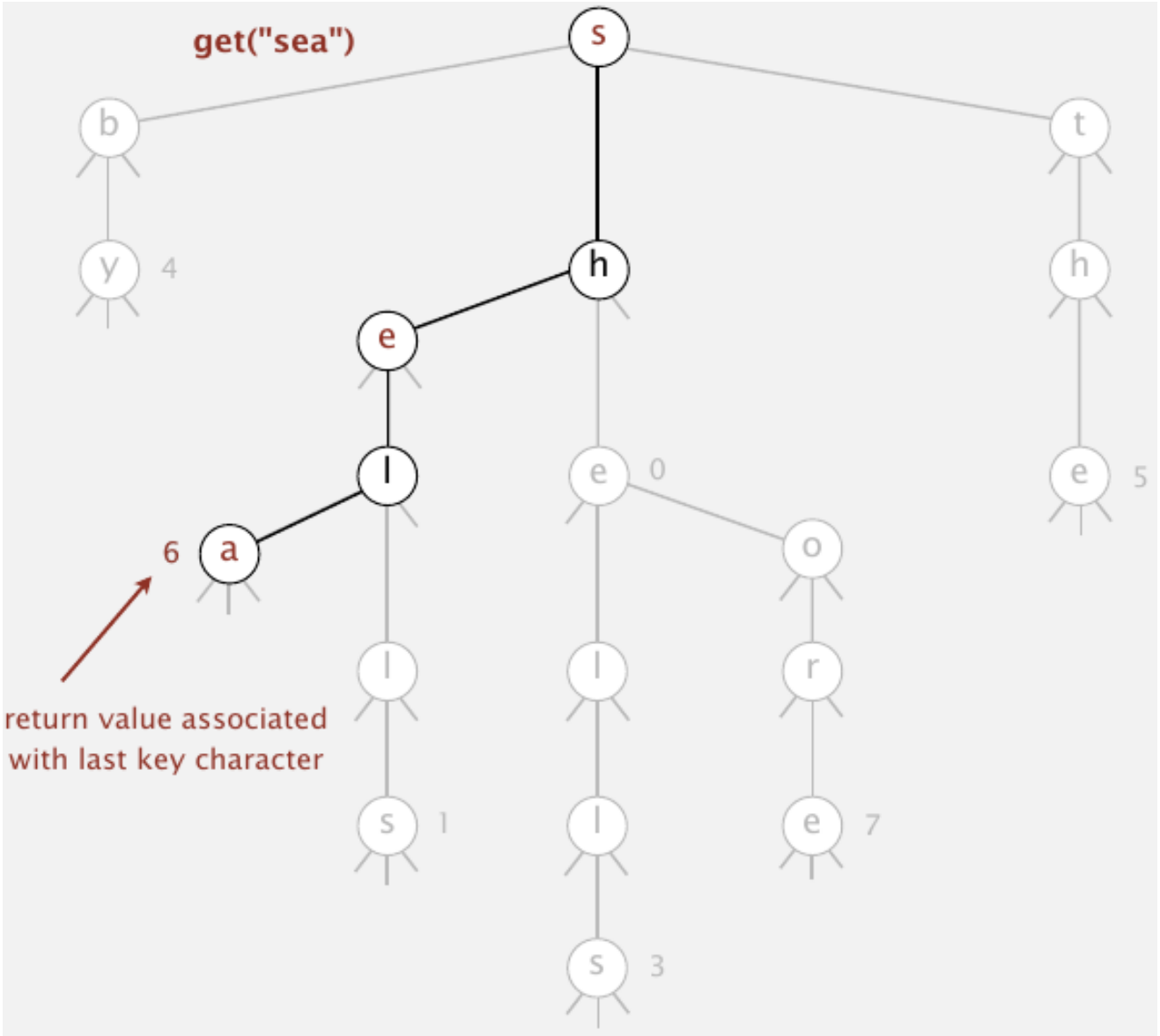
Ternary search tries

Ternary search tries (TSTs) [Bentley & Sedgewick, 1997]:

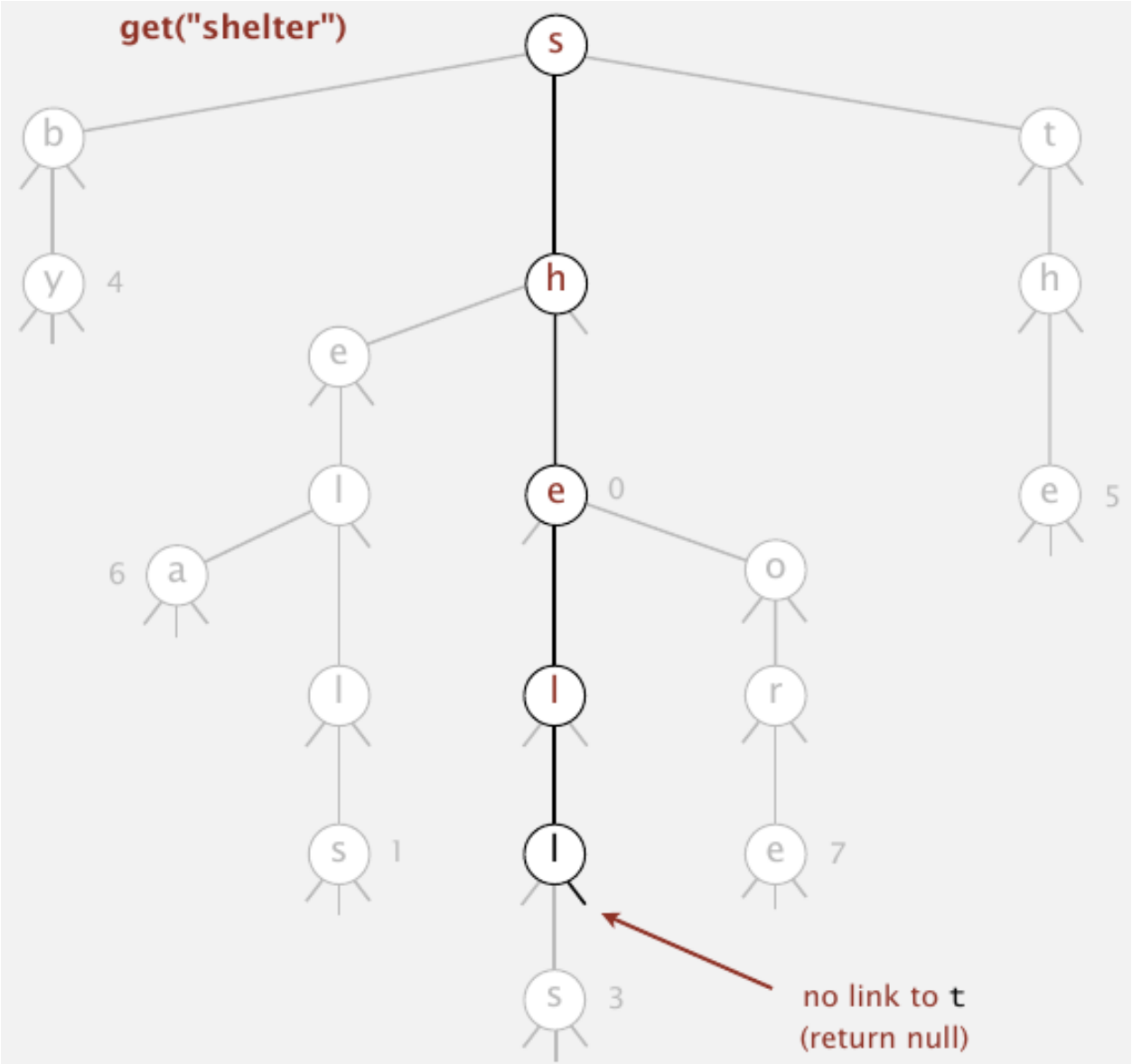
- Cada nó armazena: um caractere da chave, e talvez um valor.
- Cada nó tem 3 filhos: menores (esquerda), iguais (meio), e maiores (direita).



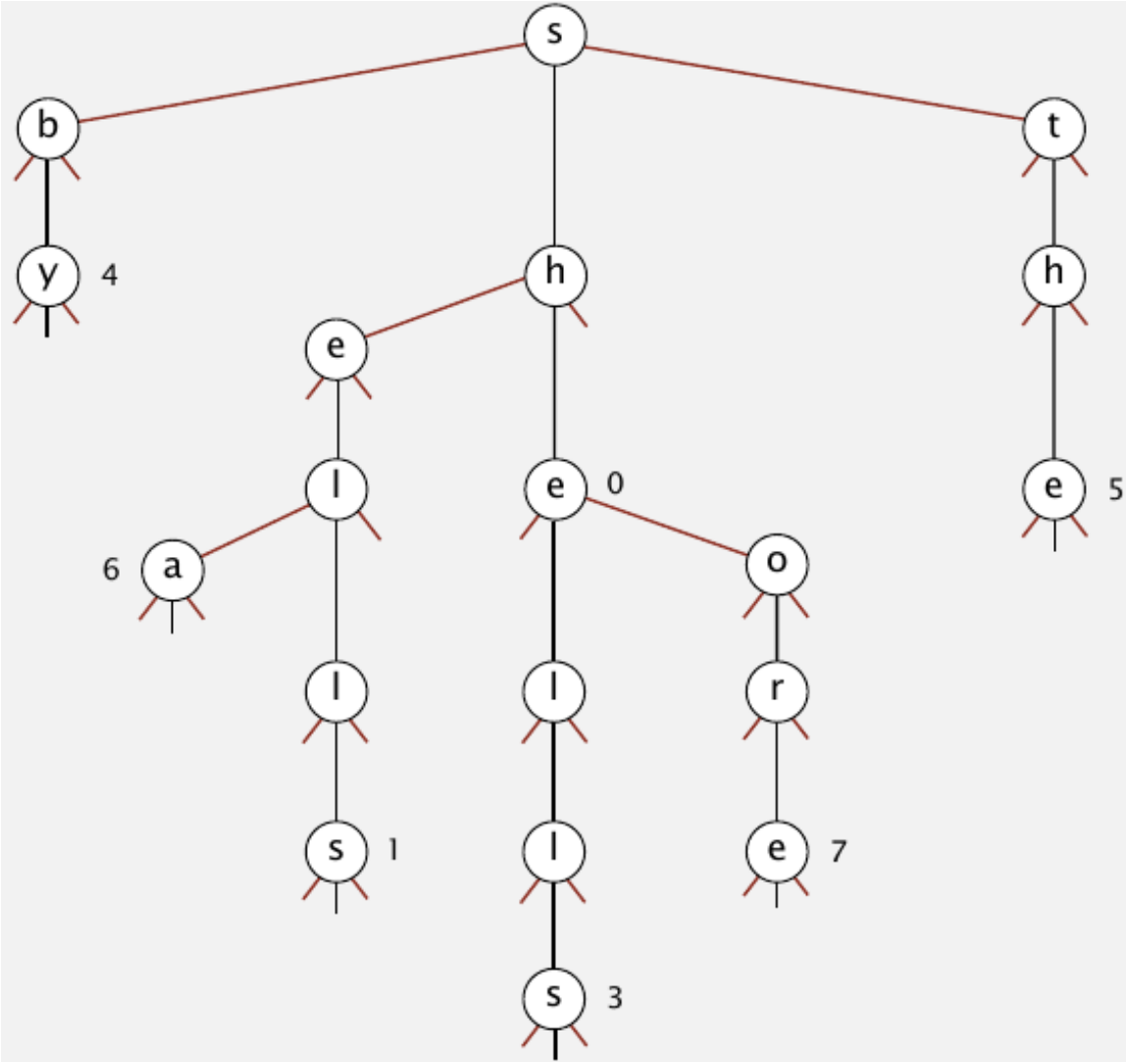
TST: *search hit*



TST: *search miss*



TST: demo

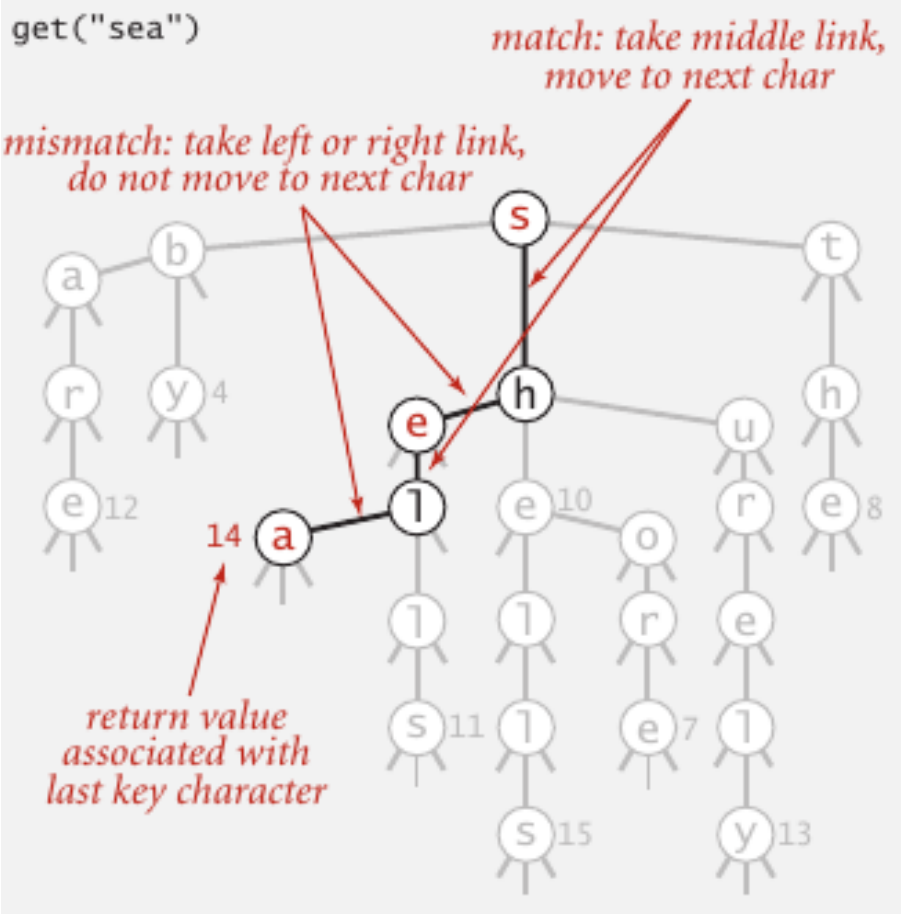


Veja o arquivo `52DemoTST.mov`

TST: busca

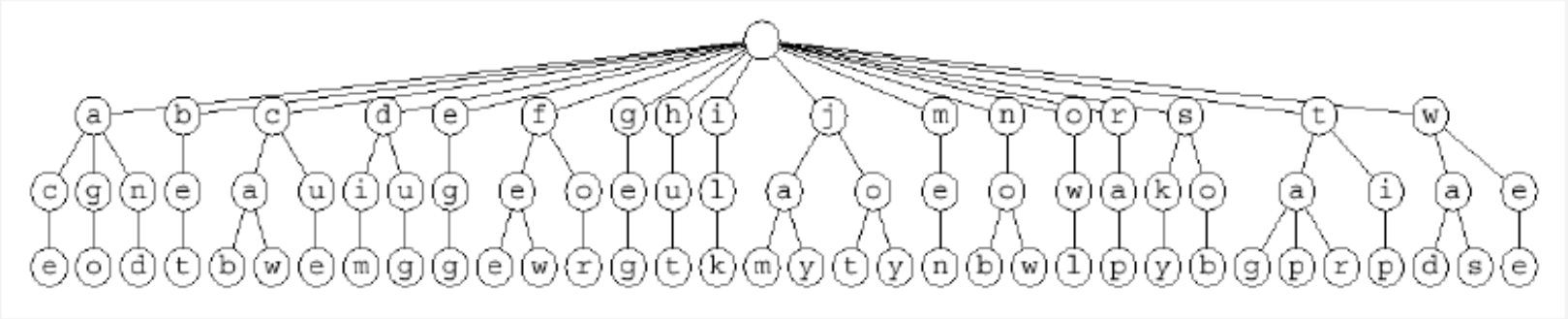
Percorra os *links* correspondentes a cada caractere da chave.

- Se menor: vá para esquerda; se maior: direita.
- Se igual, tome o meio e avance o caractere.
- *Search hit* e *miss*: igual ao *R-way trie*.



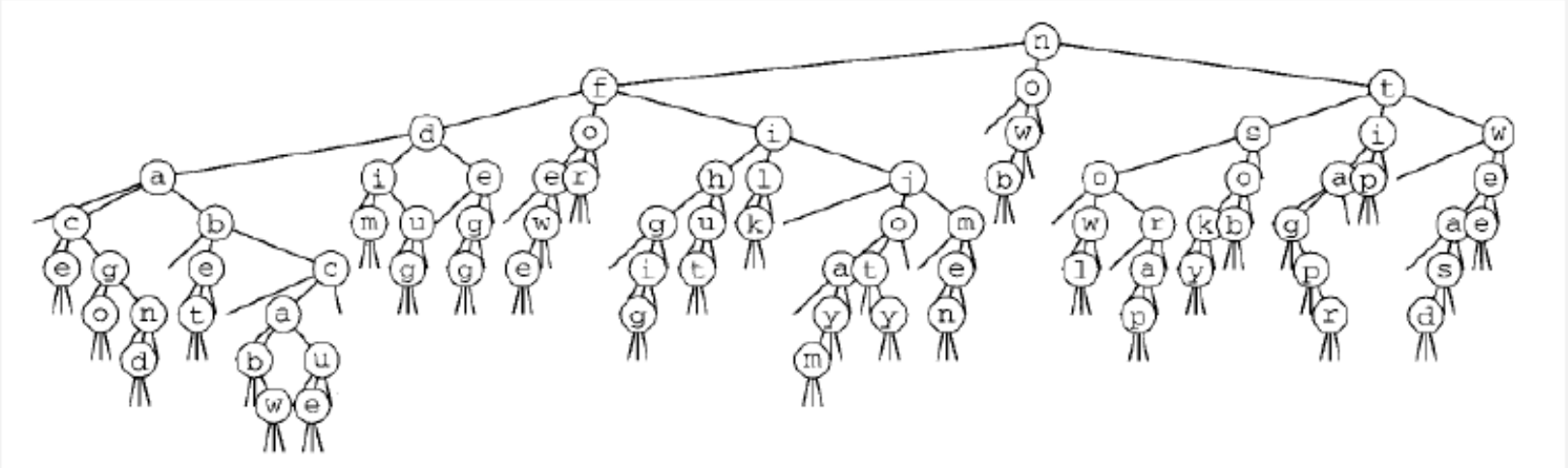
26-way trie vs. TST

26-way trie. 26 null links in each leaf.



26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

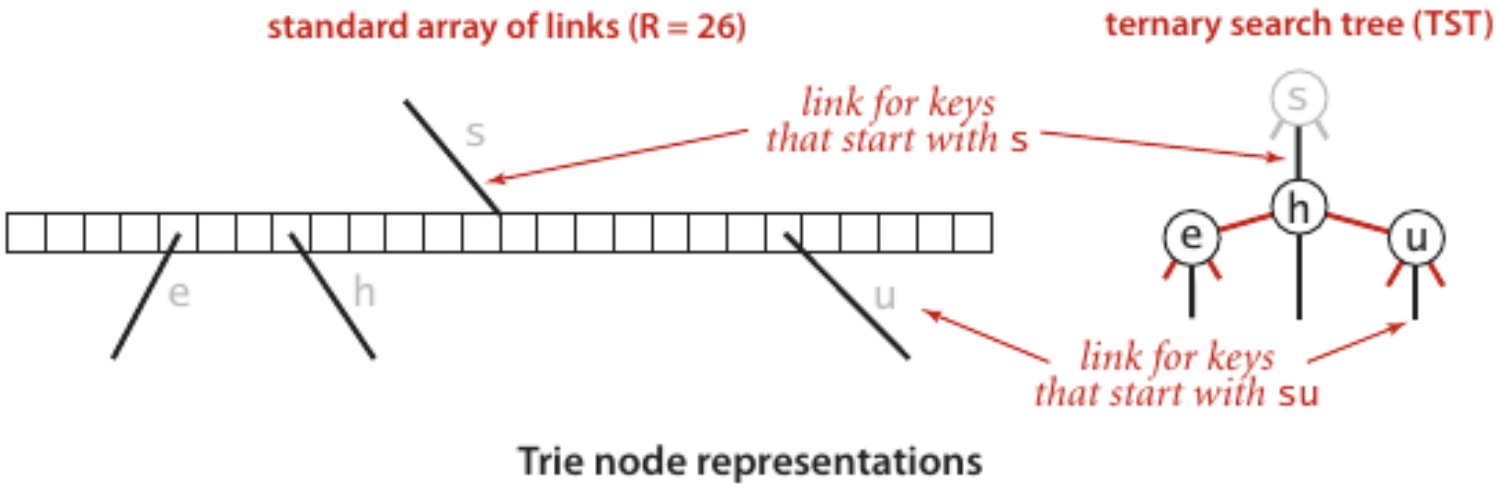
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

Representação de *TST*: implementação em C

Um nó de um TST possui cinco campos:

- Um valor.
- Um caractere da chave.
- Um ponteiro para o TST da esquerda, meio e direita.

```
typedef struct node TST;
struct node {
    Value val;
    unsigned char c;
    TST *l, *m, *r;
};
```



Trie node representations

TST: implementação em C

Inserção em TST:

```
TST* rec_insert(TST* t, String* key, Value val, int d) {
    unsigned char c = key->c[d];
    if (t == NULL) { t = create_node(); t->c = c;}
    if      (c < t->c) { t->l = rec_insert(t->l, key, val, d); }
    else if (c > t->c) { t->r = rec_insert(t->r, key, val, d); }
    else if (d < key->len - 1) {
        t->m = rec_insert(t->m, key, val, d+1);
    } else { t->val = val; }
    return t;
}

TST* TST_insert(TST* t, String* key , Value val) {
    return rec_insert(t, key, val, 0);
}
```

TST: implementação em C

Busca em TST:

```
TST* rec_search(TST* t, String* key, int d) {
    if (t == NULL) { return NULL; }
    unsigned char c = key->c[d];
    if      (c < t->c) { return rec_search(t->l, key, d); }
    else if (c > t->c) { return rec_search(t->r, key, d); }
    else if (d < key->len - 1) {
        return rec_search(t->m, key, d+1);
    } else { return t; }
}

Value TST_search(TST* t, String* key) {
    t = rec_search(t, key, 0);
    if (t == NULL) { return NULL_Value; }
    else           { return t->val;     }
}
```

Implementações de tabelas de símbolos: sumário

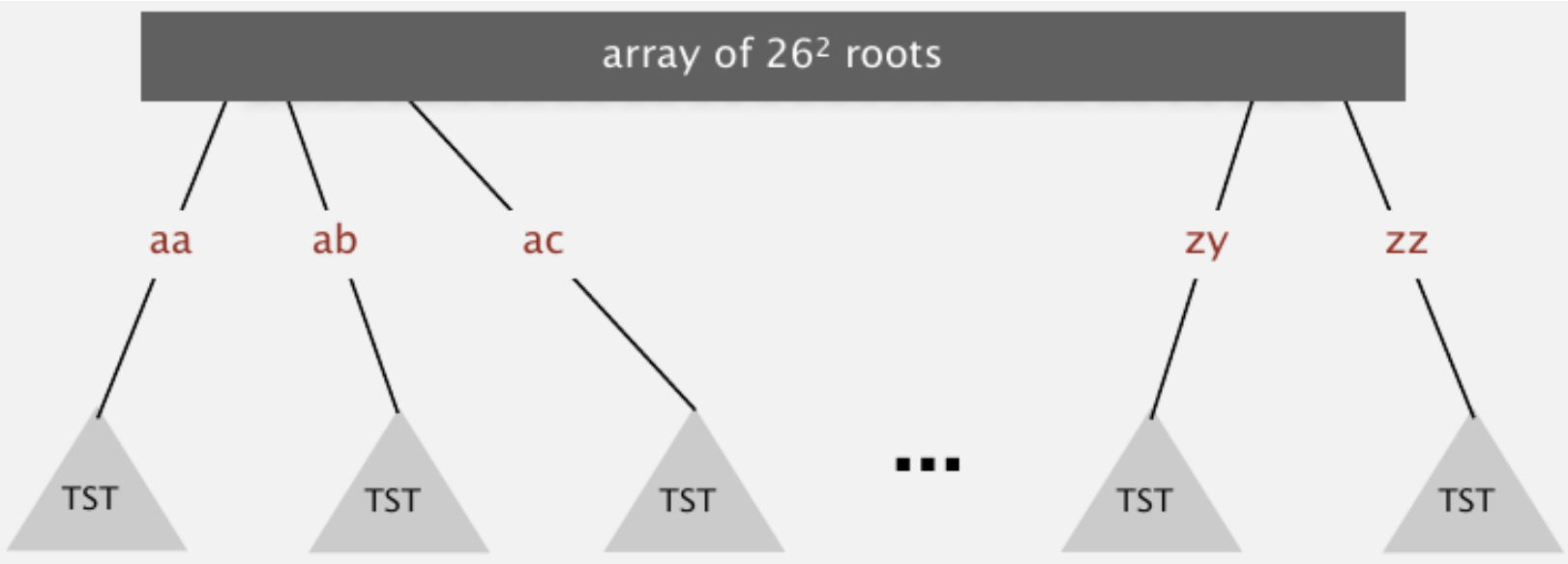
	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + (L/2) \lg N$	$(L/2) \lg N$	$(L/2) \lg N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\ N$	1.12	*out of memory*
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4\ N$	0.72	38.7

Conclusão: TST tem o mesmo desempenho que *hashing* (para *strings* como chaves), eficiente com memória.

TST com R^2 *branching* na raiz

Híbrido de um *R-way trie* e um TST.

- Faz R^2 *branching* na raiz.
- Cada um dos elementos da raiz aponta para um TST.



Q: O que fazer com palavras de um ou dois caracteres?

Implementações de tabelas de símbolos: sumário

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + (L/2) \lg N$	$(L/2) \lg N$	$(L/2) \lg N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\ N$	1.12	*out of memory*
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4\ N$	0.72	38.7
TST with R^2	$L + \ln N$	$\ln N$	$L + \ln N$	$4\ N + R^2$	0.51	32.7

Conclusão: TST com R^2 foi mais rápido que *hashing* para os casos de teste.

TST vs. *hashing*

Hashing.

- Precisa examinar a chave toda.
- *Search hits* e *misses* têm praticamente o mesmo custo.
- Desempenho depende da função *hash*.
- Não suporta operações ordenadas sobre as chaves.

TSTs.

- Só funciona para *strings* ou chaves digitais.
- Examina o número mínimo de caracteres.
- *Search miss* pode ser ainda mais eficiente.
- Suporta operações ordenadas e mais.

Conclusão: quando são possíveis de serem usados, TSTs são mais eficientes e flexíveis que *hashing*.

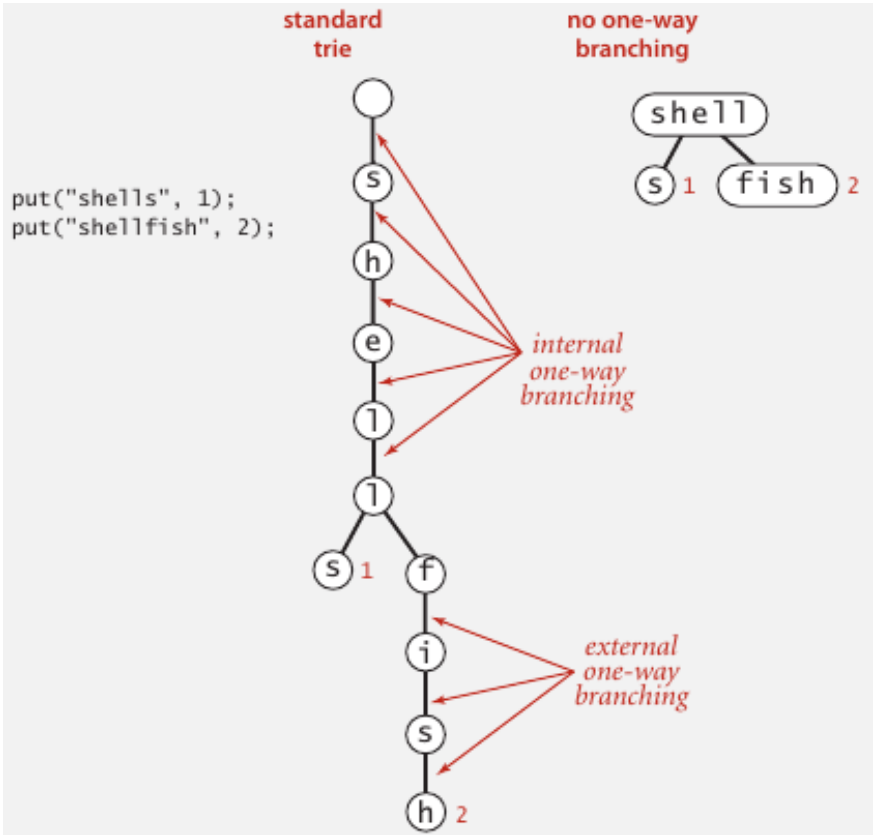
Part III

Sumário

Patricia *trie*

*P*ractical *A*lgorithm *T*o *R*etrieve *I*nfo. *C*oded *I*n *A*lphanumeric:

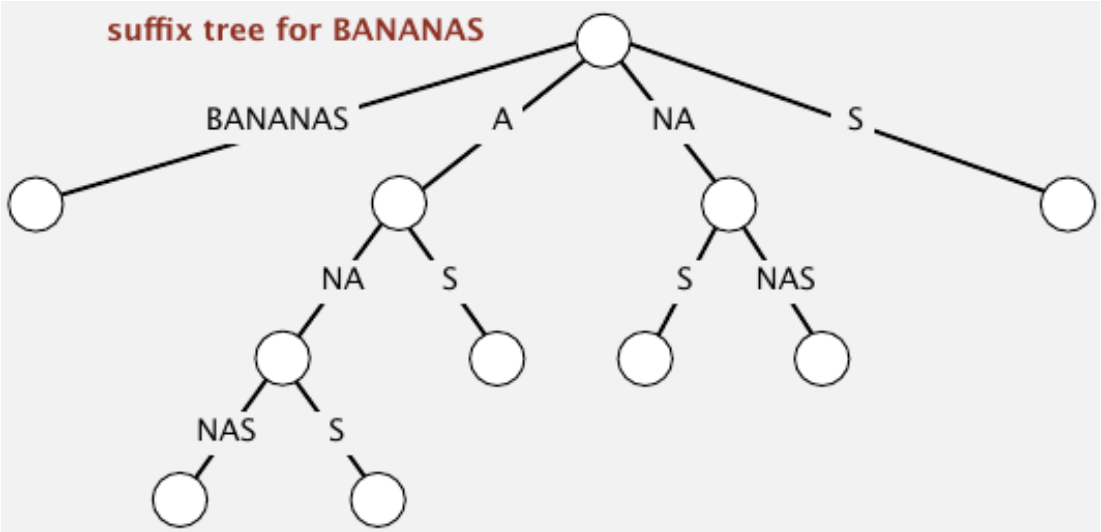
- Elimina sequências lineares de ponteiros.
- Cada nó armazena uma sequência de caracteres.
- Muito usada em bancos de dados e redes distribuídas.
- AKA: *crit-bit tree*, *radix tree*.



Árvore de sufixos

Árvore de sufixos:

- Patricia dos sufixos de uma *string*.
- Pode ser construída em tempo linear. (Difícil!)
- Aplicações: bases de dados de biologia computacional.



Tabelas de símbolos para *strings*: sumário

Uma história de sucesso em projeto e análise de algoritmos.

BSTs balanceadas:

- Garantia de desempenho: $\log N$ comparações de chaves.
- Suporta operações ordenadas.

Tabelas *hash*:

- Garantia de desempenho: número constante de *probes*.
- Requer uma boa função *hash* para o tipo da chave.

Tries:

- Garantia de desempenho: $\log N$ acessos a `char`s.
- Suporta operações ordenadas e sobre caracteres.

Mensagem final: todas as EDs acima têm a sua utilidade, o importante é saber quando usar cada uma!