SBVORIN: Organização e Recuperação da Informação

Aula 09: Tries

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Tries

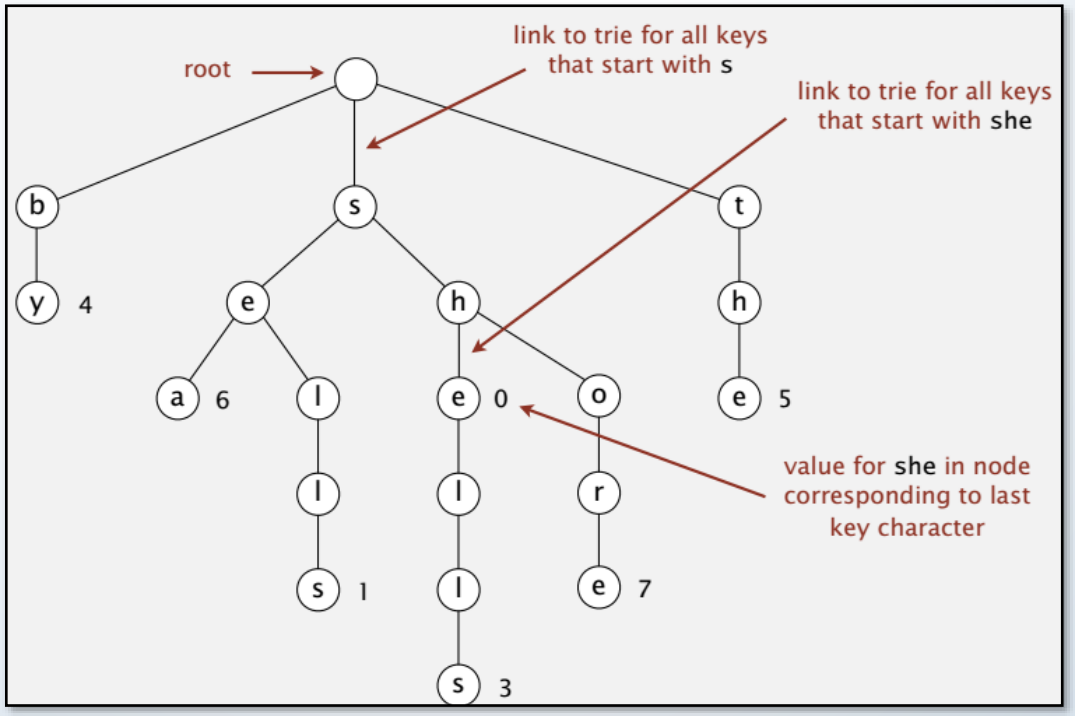
- Uma trie (lê-se try) é um tipo de árvore usado para implementar Tabelas de Símbolos de Strings;
- **Nome completo:** R-way trie (cada nó pode ter até R filhos);
- Tries também são conhecidas como árvores digitais e como árvores de prefixos.

public class StringST<Value>	
StringST()	*create an empty symbol table*
void put(String key, Value val)	*put key-value pair into the symbol table*
Value get(String key)	*return value paired with given key*
void delete(String key)	*delete key and corresponding value*
⋮	

Tries

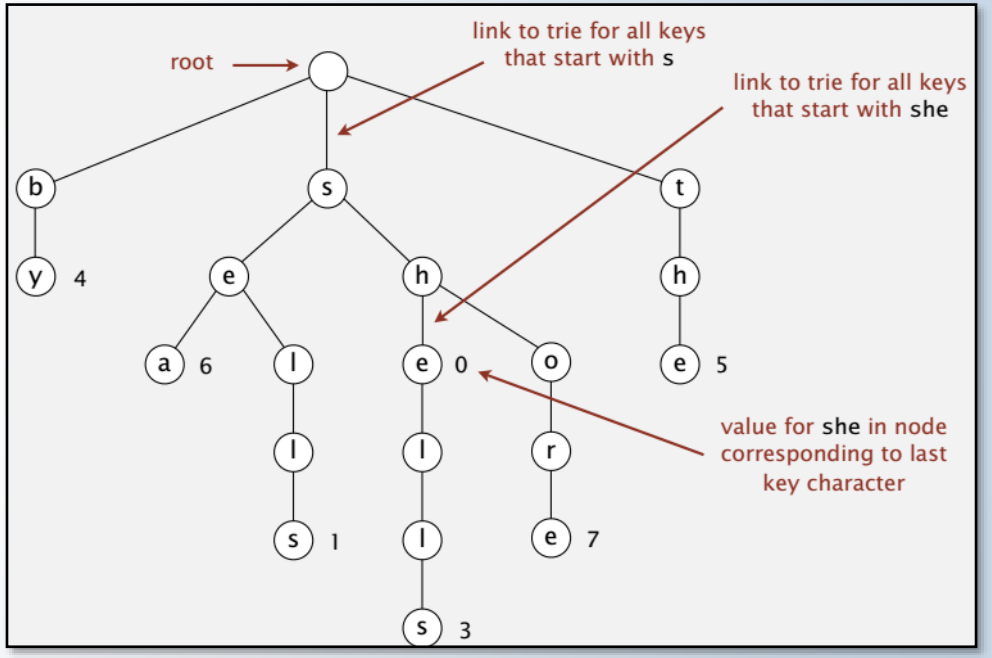
- Armazena os caracteres nas referências aos nós;
- Cada nó tem R filhos, um para cada caractere possível;
- Por enquanto, não nos preocuparemos com referências nulas.

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



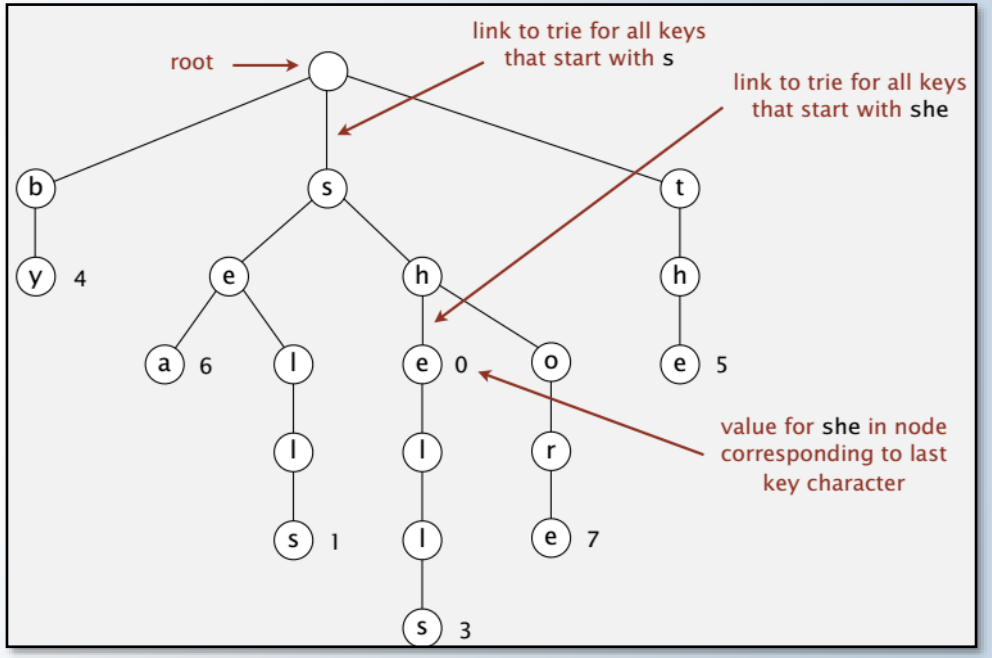
Tries

- Duas observações importantes sobre tries:
  - As chaves não são armazenadas explicitamente, pois elas ficam codificadas nos caminhos que começam na raiz;
  - Todos os prefixos de chaves estão representados na trie, ainda que alguns prefixos não sejam chaves;
- As referências da estrutura correspondem a caracteres e não a chaves. Nas figuras, o caractere escrito dentro de um nó é o caractere da referência que entra no nó.



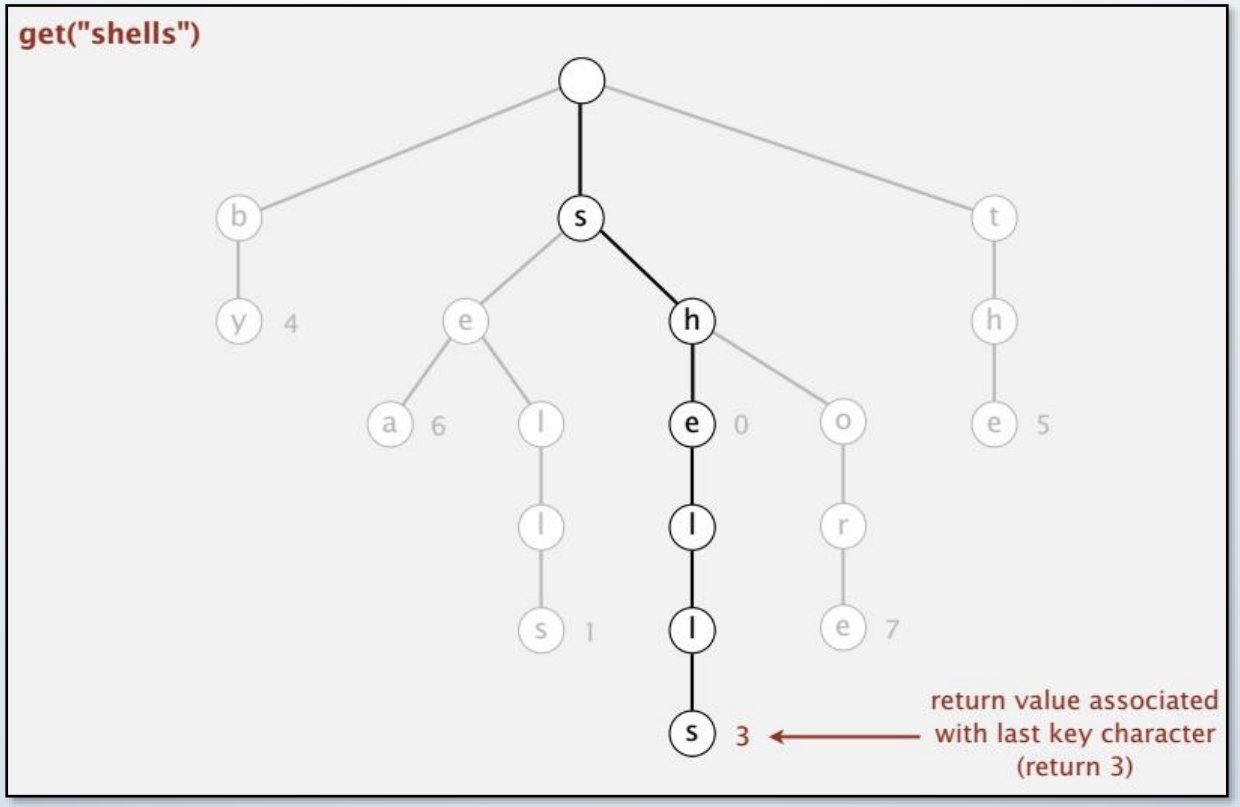
Tries

- Ao descer da raiz até um nó x, soletramos uma String s:
  - Diz que s leva ao nó x;
  - Diz também que o nó x é localizado pela String s;
  - O nó localizado pela String vazia é a raiz;
- A String que leva a um nó x é uma chave se e somente se `x.val != null`;
- Subtries: Cada nó x da trie é a raiz de uma subtrie X
  - Representa o conjunto de todas as chaves da trie que têm como prefixo a string que leva da raiz da trie até x.



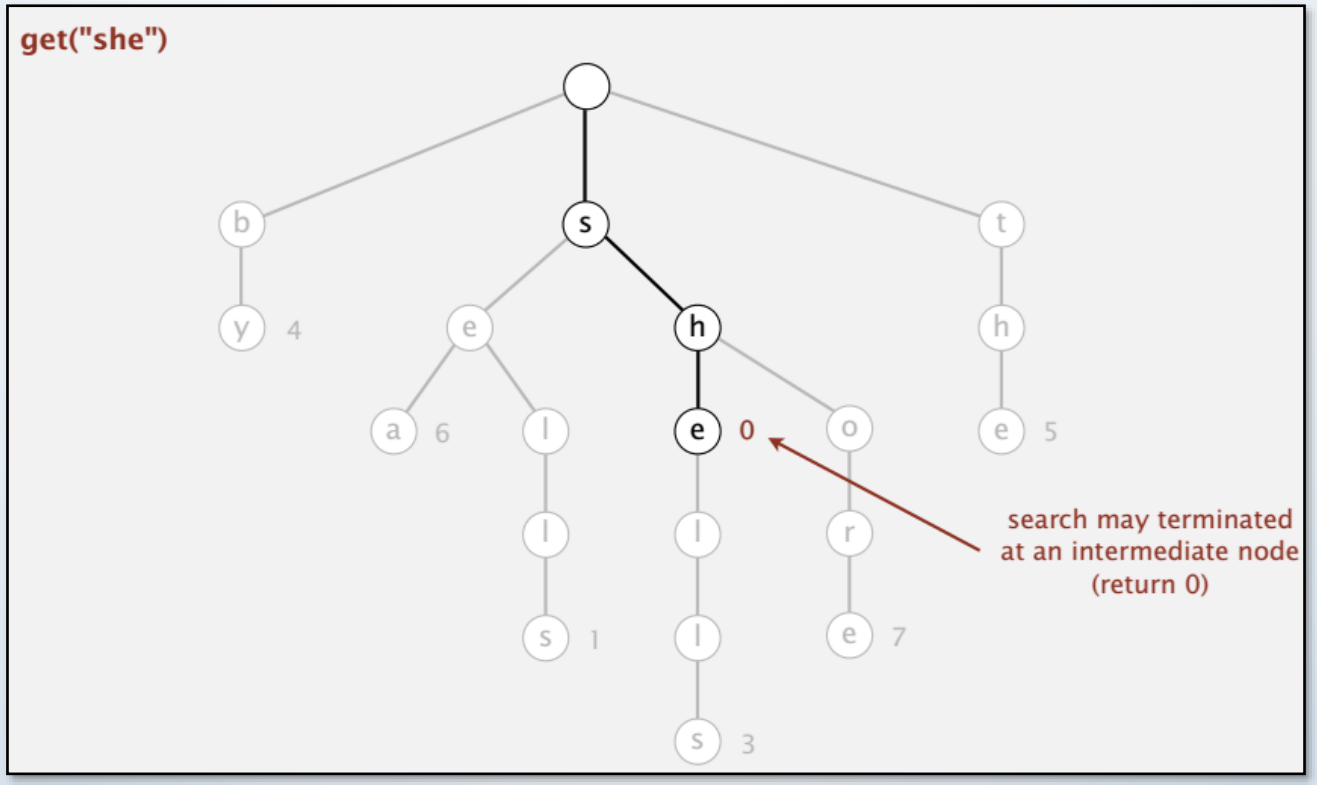
Tries
Operação de Busca

- Seguir os links correspondentes a cada caractere da chave:
  - Resultado com sucesso (*search hit*): nó onde a pesquisa termina tem um valor não nulo;
  - Erro na pesquisa (*search miss*): alcança uma referência nula ou nó onde a pesquisa termina em um valor nulo.



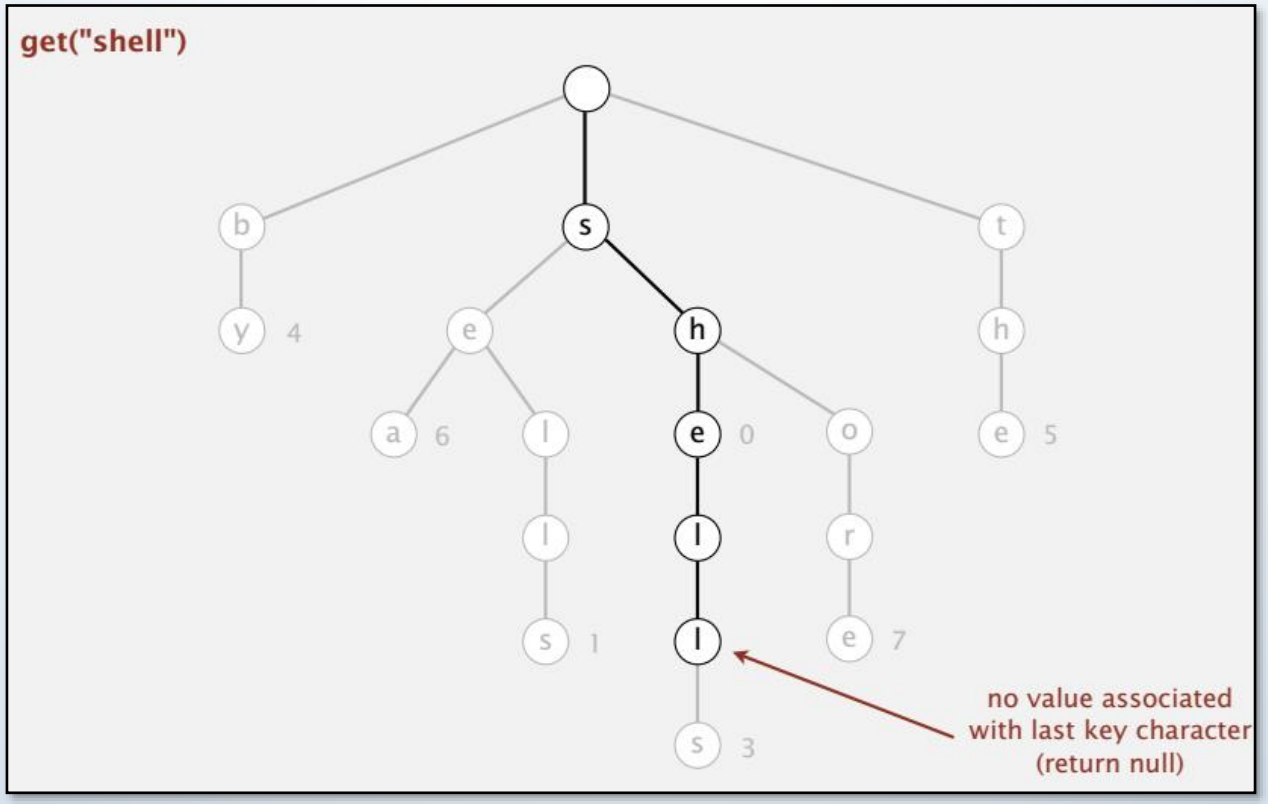
Tries
Operação de Busca

- Seguir os links correspondentes a cada caractere da chave:
  - Resultado com sucesso (*search hit*): nó onde a pesquisa termina tem um valor não nulo;
  - Erro na pesquisa (*search miss*): alcança uma referência nula ou nó onde a pesquisa termina em um valor nulo.



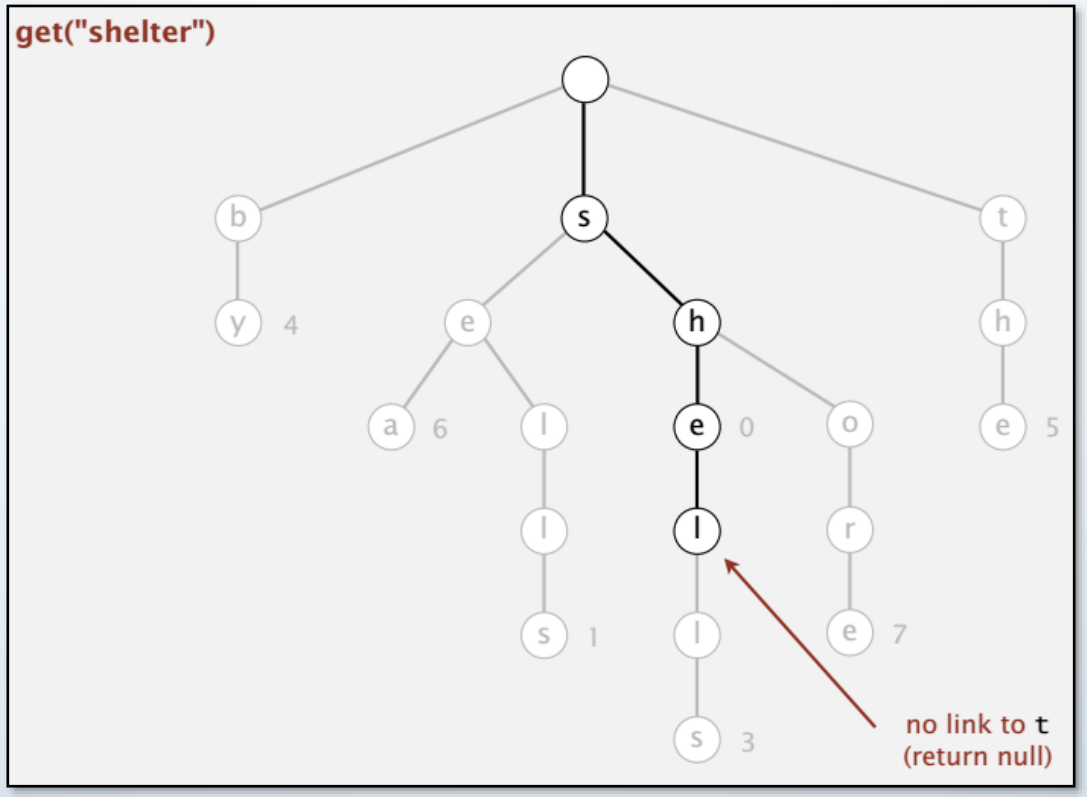
Tries
Operação de Busca

- Seguir os links correspondentes a cada caractere da chave:
  - Resultado com sucesso (*search hit*): nó onde a pesquisa termina tem um valor não nulo;
  - Erro na pesquisa (*search miss*): alcança uma referência nula ou nó onde a pesquisa termina em um valor nulo.



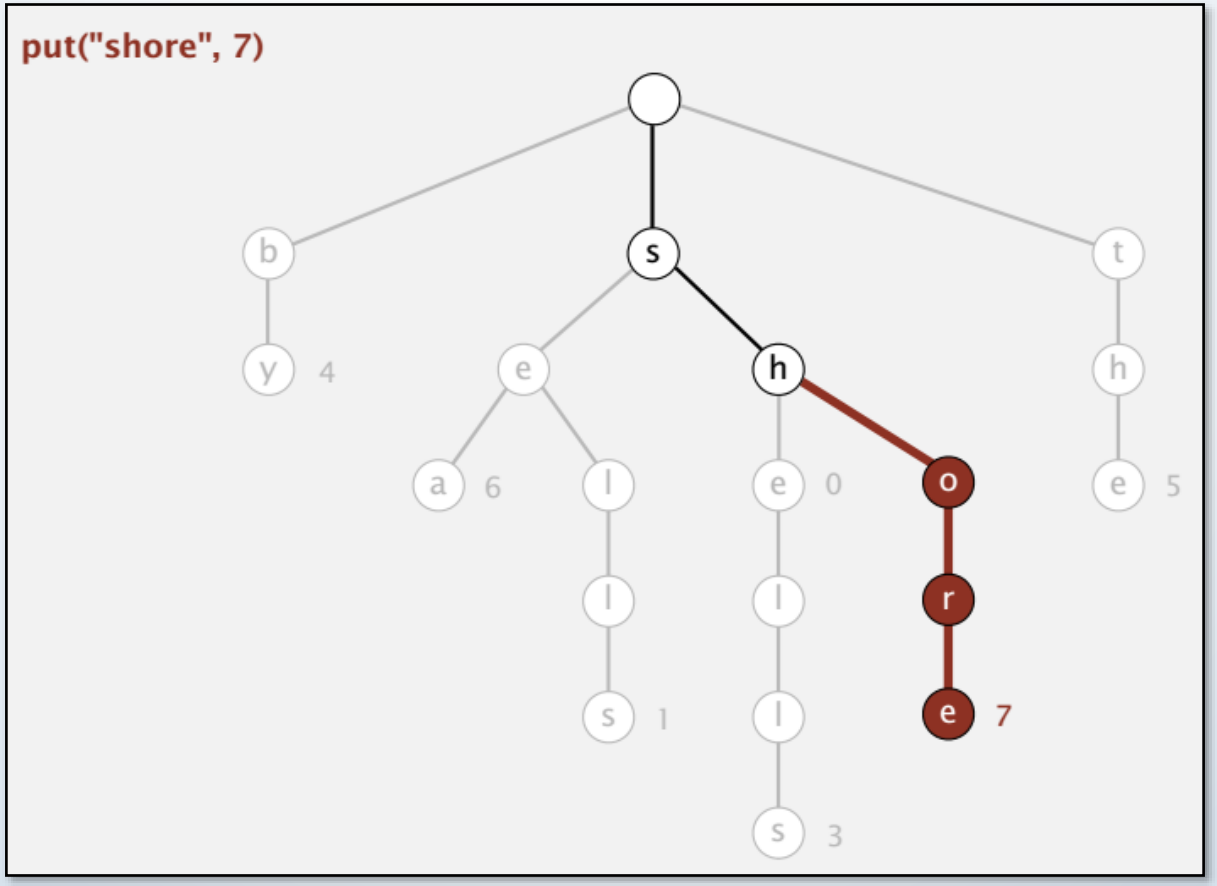
Tries
Operação de Busca

- Seguir os links correspondentes a cada caractere da chave:
  - Resultado com sucesso (*search hit*): nó onde a pesquisa termina tem um valor não nulo;
  - Erro na pesquisa (*search miss*): alcança uma referência nula ou nó onde a pesquisa termina em um valor nulo.

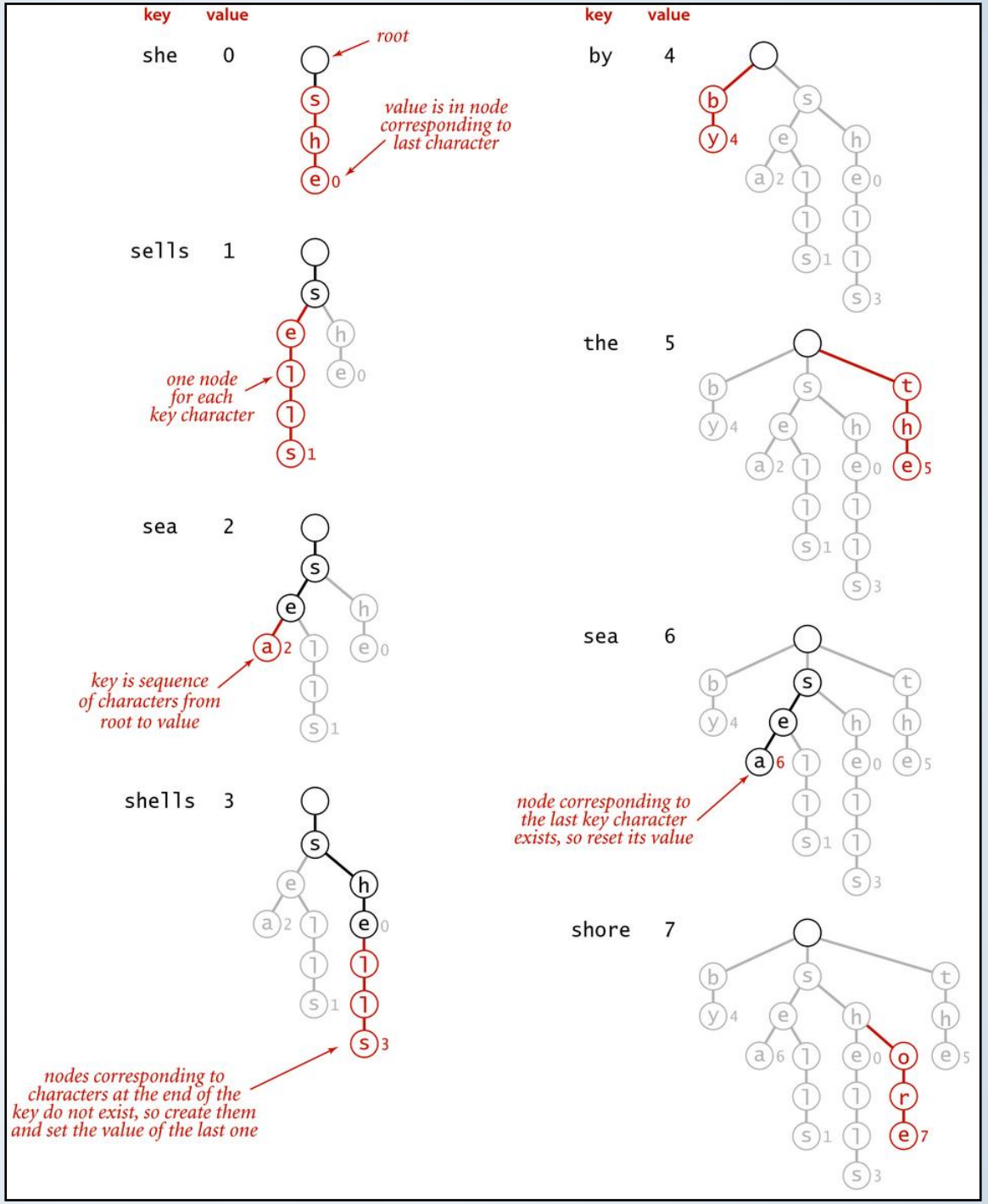


Tries
Operação de Inserção

- Seguir as referências correspondentes à cada caractere da chave:
  - Encontre uma referência nula: criar um novo nó;
  - Encontrar o último caractere da chave: definir valor naquele nó.



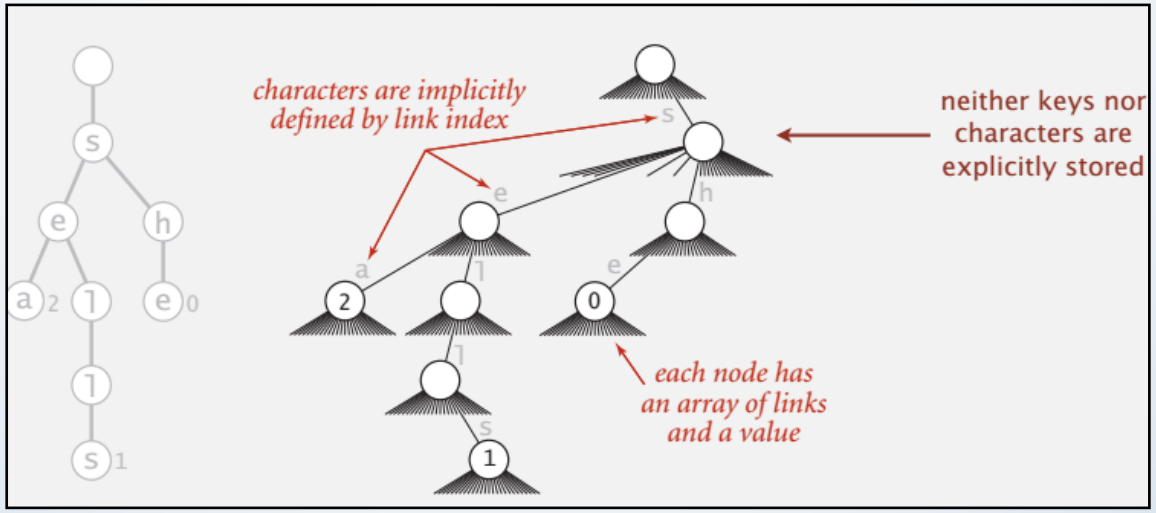
Tries
Exemplo de Construção



Tries
Implementação em Java

Um valor e uma referência para R nós

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



Tries
Implementação em Java

- O que faz o segundo método `put()`?
  Ao receber um inteiro `d <= key.length()`, o método acrescenta a chave `key.substring(d)` ao conjunto de chaves representadas na subtrie cuja raiz é x:
  - Expande a subtrie cuja raiz é x de modo a acomodar a string `key.substring(d)` e associa o valor val a essa String;
  - Devolve a raiz da subtrie resultante da inserção; essa raiz é igual a x, a menos que o método tenha sido invocado com primeiro argumento igual a `null`.

```
public class TrieST<Value>
{
   private static final int R = 256;      ← extended ASCII
   private Node root = new Node();

   private static class Node
   {  /* see previous slide */  }

   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()) { x.val = val; return x; }
      char c = key.charAt(d);
      x.next[c] = put(x.next[c], key, val, d+1);
      return x;
   }

   ⋮
```

- Observações sobre o segundo método `put()`:
  - Pode envolver a criação de zero, um, dois, ou mais novos nós. Se a String `key.substring(d)` já estava representada na subtrie, nenhum novo nó é criado;
  - Nunca é invocado com argumentos arbitrários:temos sempre `d <= key.length()` e a String `key.substring(0,d)` leva da raiz da trie até o nó x;
  - Se `d == key.length()`, o método dá a resposta correta, pois `key.substring(d)` é vazio e x é o nó localizado pela String vazia.

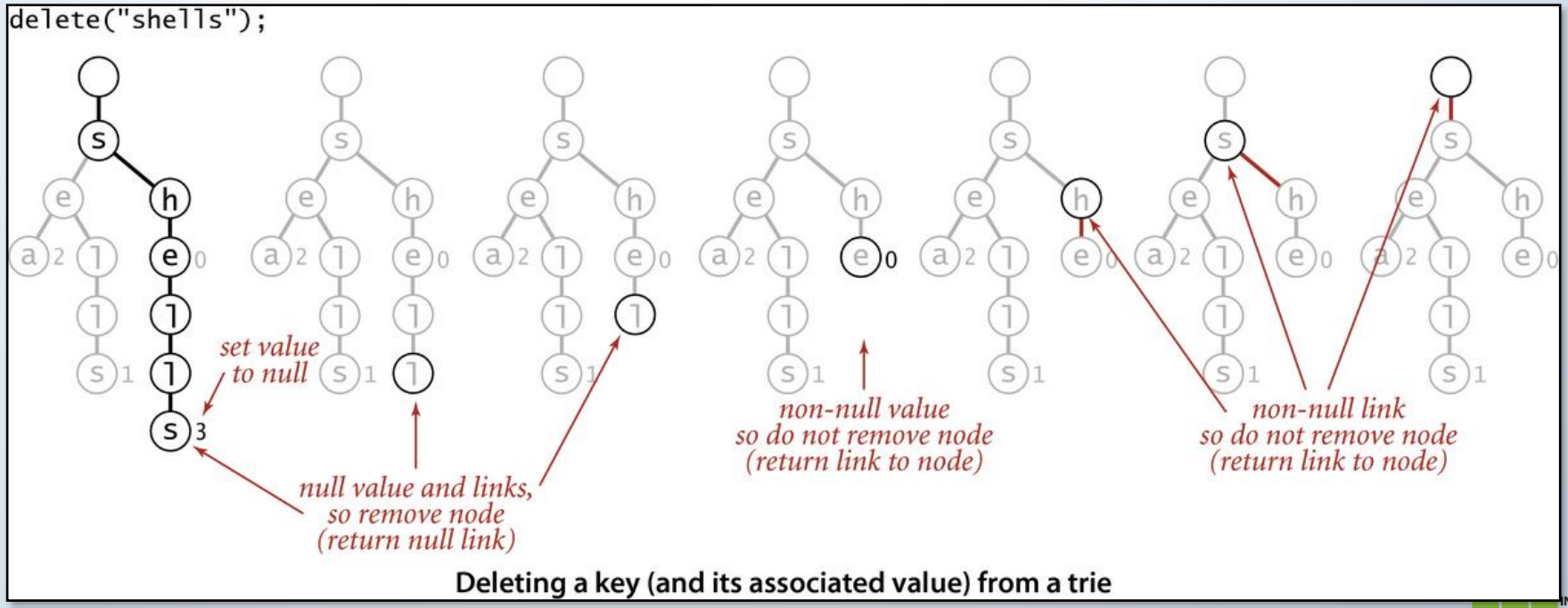
Tries
Implementação em Java

- O que faz o segundo método `get()`?
  Ao receber um inteiro `d <= key.length()`, o método devolve o nó localizado pela String `key.substring(d)` na subtrie cuja raiz é x. Se tal nó não existe, retorna `null`;
- Observações sobre o segundo método `get()`:
  - O método nunca é invocado com argumentos arbitrários:
    - `key.substring(0,d)` sempre leva da raiz da trie até o nó x;
  - Se `d == key.length()`, o método dá a resposta correta, pois na subtrie cuja raiz é x o nó localizado pela String vazia é x;
  - O método ignora a distinção entre chaves e as Strings que não são chaves, pois nunca consulta o campo `val` dos nós.

```
   ⋮
   public boolean contains(String key)
   {  return get(key) != null;  }

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

   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);
   }

}
```

Tries
Operação de Exclusão

- Para excluir uma chave do conjunto de chaves da trie:
  - Encontrar o nó correspondente à chave e definir o valor como nulo;
  - Se o nó tiver valor nulo e todos as referências nulas, remover o nó em questão recursivamente.



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

Tries
Operação de Exclusão

- O que faz o segundo método `delete()`?
  - O método recebe um nó x de uma trie, uma String k, que pode não ser uma chave, e um inteiro `d <= k.length()`
  - O método supõe que `k.substring(0, d-1)` leva da raiz da trie até x. Diga-se que X é a subtrie com raiz x e K' é o conjunto de todas as chaves da trie que têm prefixo `k.substring(0, d-1)`
  - O segundo método `delete()` remove k do conjunto K', e portanto também do conjunto de chaves da trie, e devolve a raiz de uma subtrie, parte da subtrie X, que representa o conjunto de chaves $K' - \{k\}$:
    - Se $K' - \{k\}$ for vazio, retorna `null`;
    - Senão, devolve x;
  - Se k não é chave, o método efetivamente não faz nada.

```
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 == 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;
}
```

Tries

Análise de Performance

- A aparência de uma ABB depende muito da ordem em que as chaves são inseridas e removidas. Não é o que acontece com Tries. A expressão "Trie balanceada" não faz sentido;
- 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 de chaves:
  - O número de nós visitados para buscar ou inserir uma chave de comprimento W 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 uma referência nula;
- 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 $\log_R N$
  - Em termos mais técnicos: se o número de nós examinados é E então $\frac{\mathrm{E}}{\log_R N}$ tende a 1 quando N tende a infinito;
  - A prova envolve apenas teoria das probabilidades elementar. O ponto de partida é a hipótese de que cada caractere de uma chave aleatória tem a mesma probabilidade de ser qualquer um dos R caracteres do alfabeto;
  - Consequência pouco intuitiva: se as chaves são aleatórias, o consumo de tempo médio não depende do comprimento das chaves.

Bibliografia

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

GOODRICHM M. T.; TAMASSIA, R. **Estruturas de Dados & Algoritmos em Java**. Porto Alegre: Bookman, 2013. 700 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.