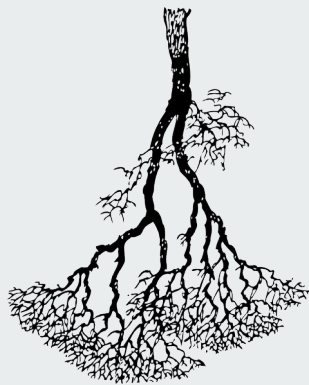


Árvores Específicas

Trie e Patricia



André Furlan
Hiago Matheus Brajato

Árvores Trie



- Propósitos
- Propriedades
- Análises
- Complexidades

Histórico

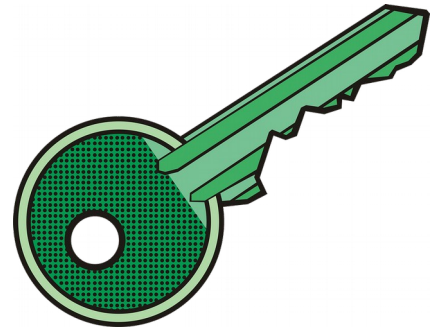


- 1960
- “ReTRIEval”
- Armazenamento de dados alfanuméricos
- Recuperação de dados

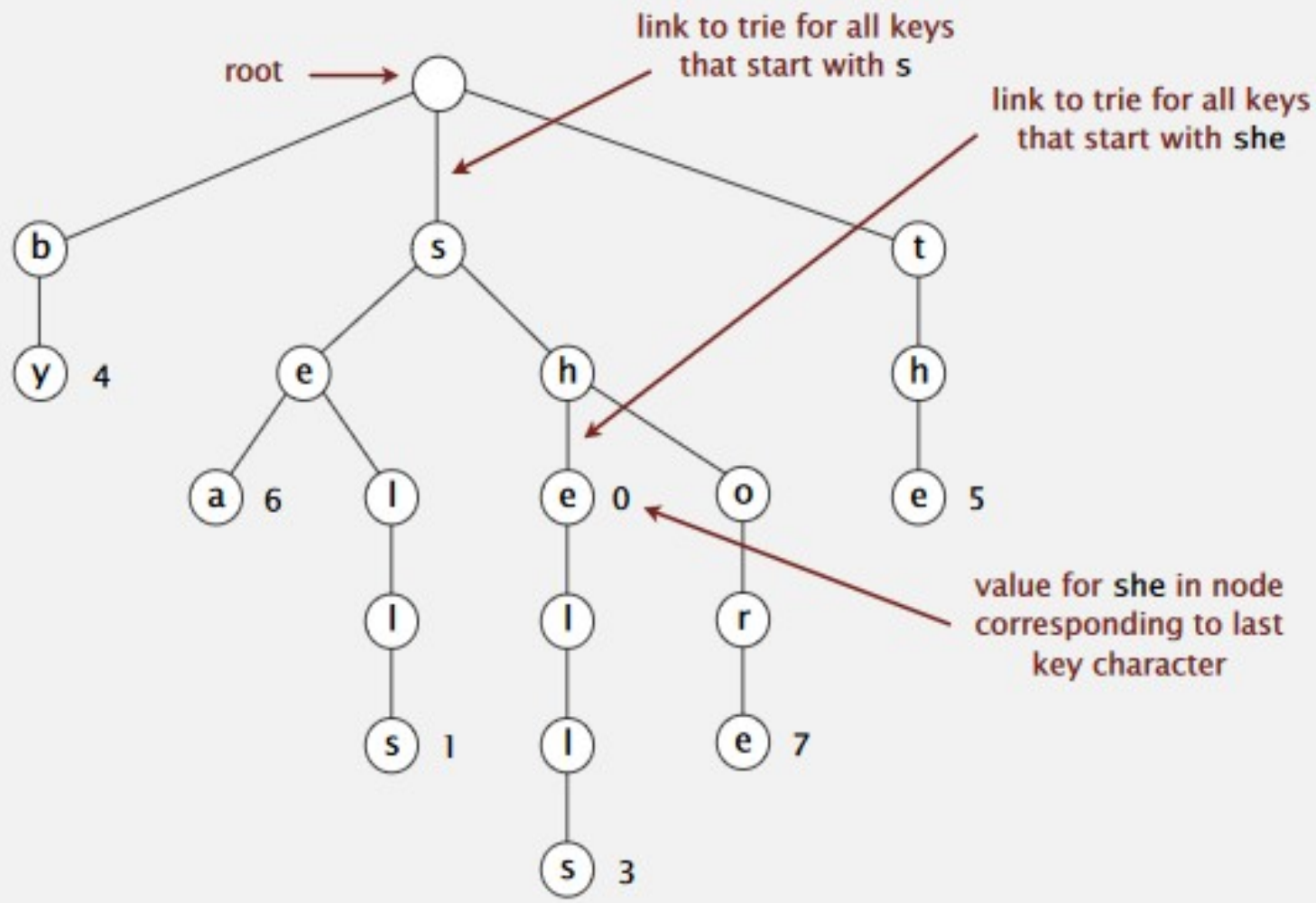
Estrutura de nós



- Chave é representada pela estrutura
- Chaves são “divisíveis”
- Caracteres armazenados



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



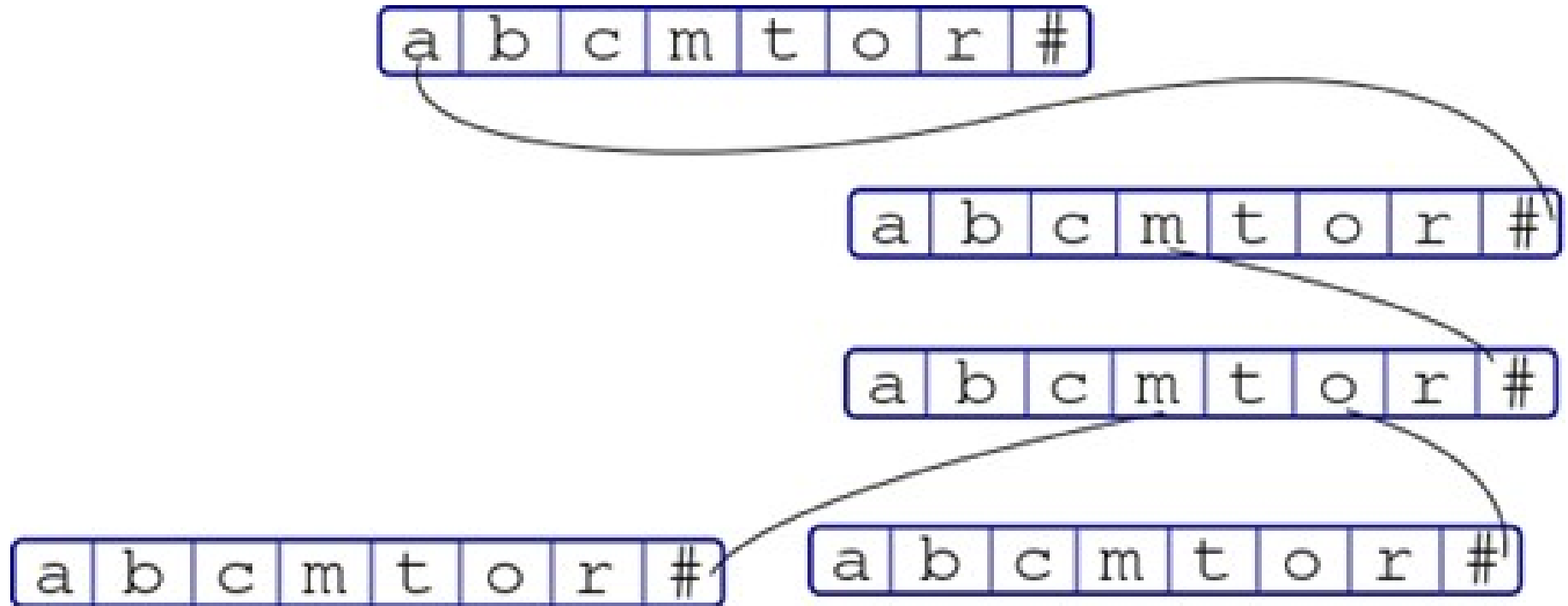
Implementação R-Way



- R: tamanho do alfabeto
- Way: caminho a seguir no próximo nó

```
public class TrieNode {  
    private TrieNode[] arrSubTries;  
    private boolean isLeaf = false;  
}
```

Estrutura de nós - Vetores Hierárquicos



Complexidade de Espaço



Espaço Necessário == N° de Links

N° de Links == $N * R * w$

- N: quantidade de chaves presentes
- R: nó associado a um valor
- w: tamanho médio da chave

Operações Básicas sobre Tries



- Pesquisa
- Inserção
- Remoção

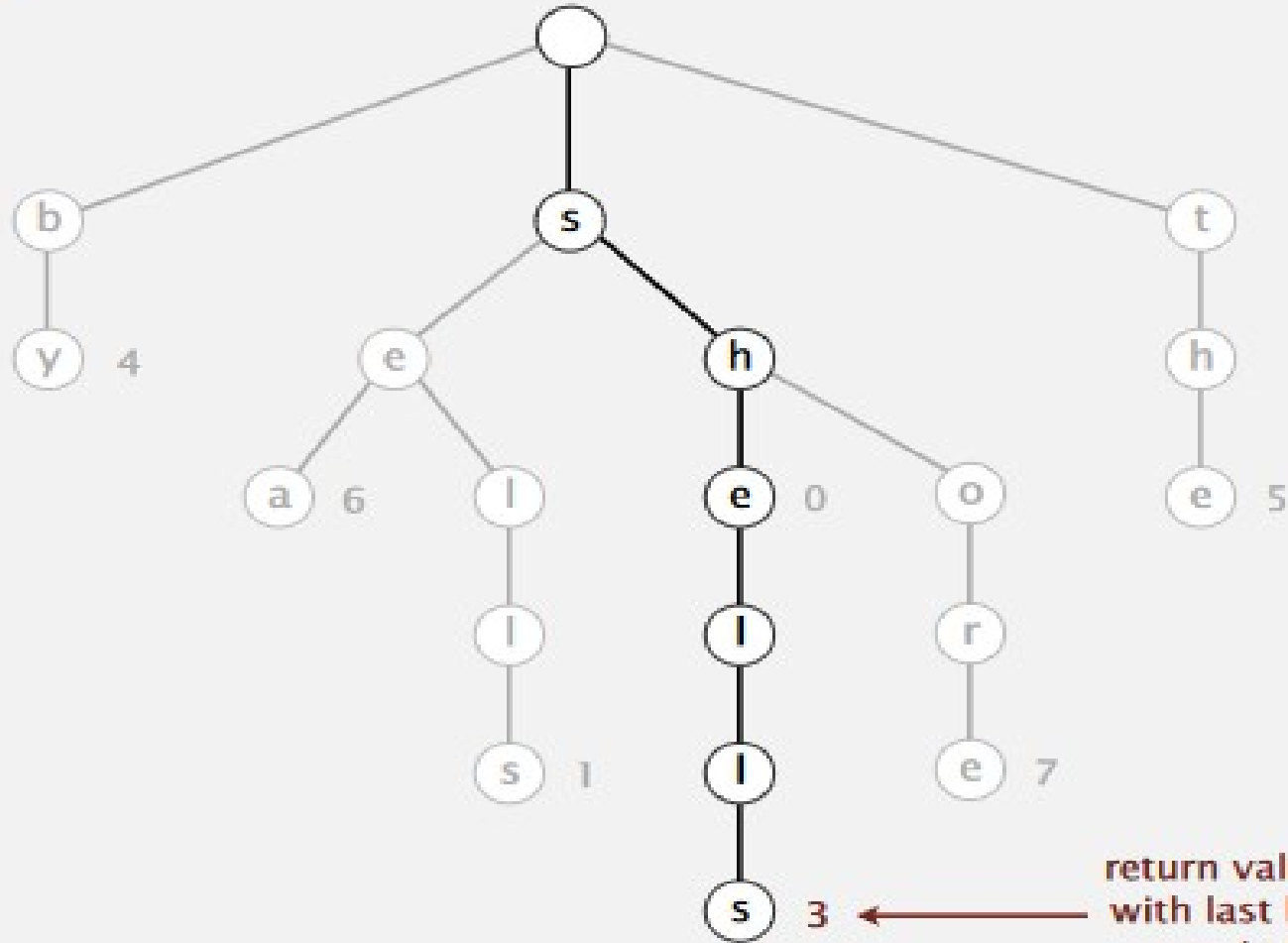
Pesquisa em Tries



Passos do Procedimento:

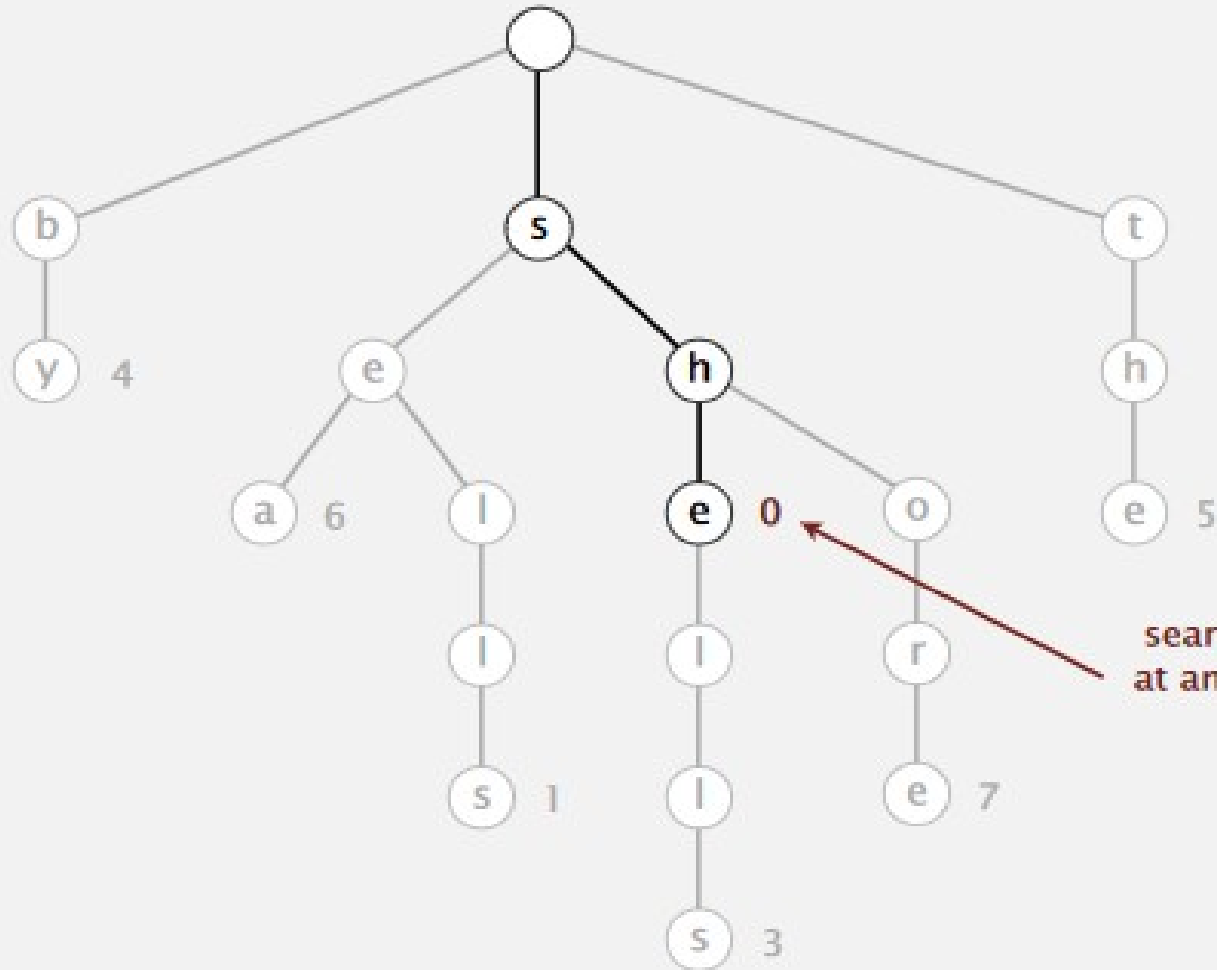
1. Primeiro caractere da palavra é encontrado;
2. Segue-se o link correspondente ao segundo caractere;
3. O passo 2 é repetido até que todos os caracteres de uma chave (ou link nulo) seja encontrado;
4. Verifica-se o valor do nó do último caractere (hit ou miss)

get("shells")



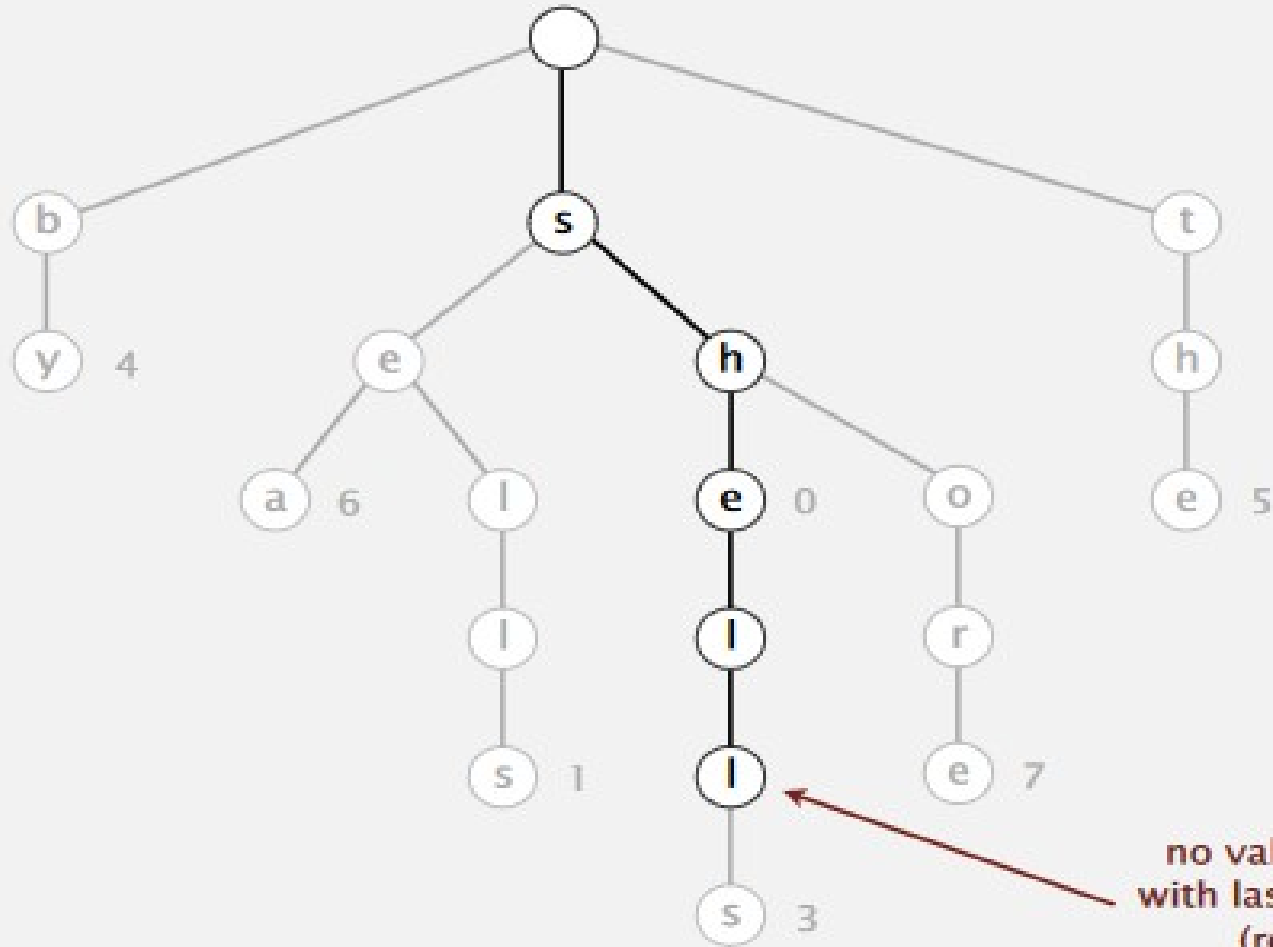
return value associated
with last key character
(return 3)

get("she")



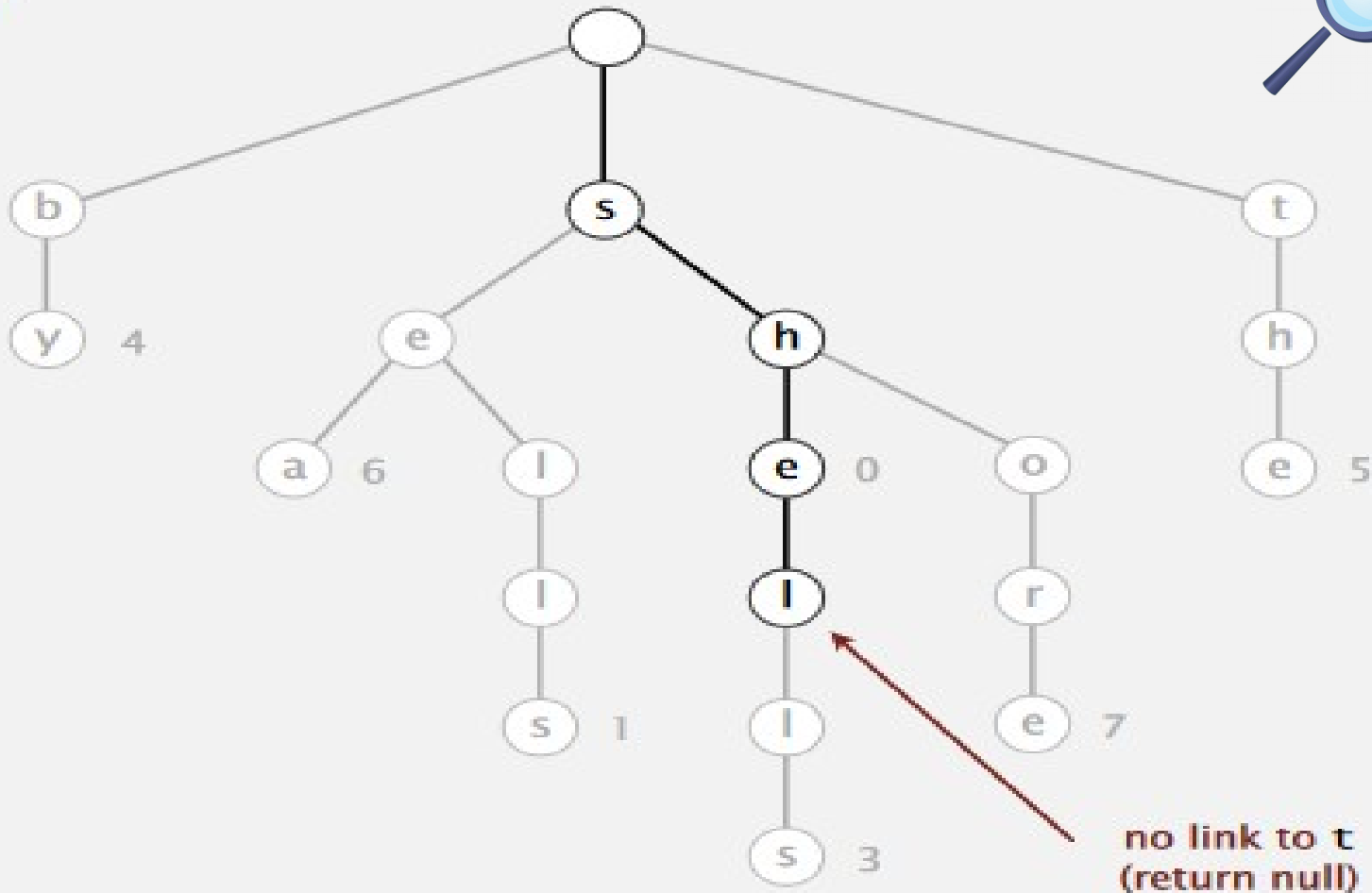
search may terminated
at an intermediate node
(return 0)

get("shell")



no value associated
with last key character
(return null)

get("shelter")



Implementação com “Flags”

Nós são marcados como folhas os não (isLeaf):

- Nós folhas: fim de uma chave
- Nós não-folhas: não é o fim de uma chave

```
public class TrieNode {  
    private TrieNode[] arrSubTries;  
    private boolean isLeaf = false;  
}
```

Corretude: Pesquisa



A corretude do método de Pesquisa em Tries é feito a partir da indução nas subárvores:

- se (chave PERTENCE à estrutura)
 - retorne a chave;
- se (chave NÃO PERTENCE à estrutura)
 - retorne null;

Análise de Complexidade: Pesquisa



Pior Caso: $O(n)$: corresponde ao tamanho da chave

Caso Médio: $O(n)$: prefixo da String pesquisada \longrightarrow P.A.
 $(n+1)/2$

Melhor Caso: $O(1)$: nenhum prefixo da String pesquisada
(somente um nó visitado)

Inserção em Tries

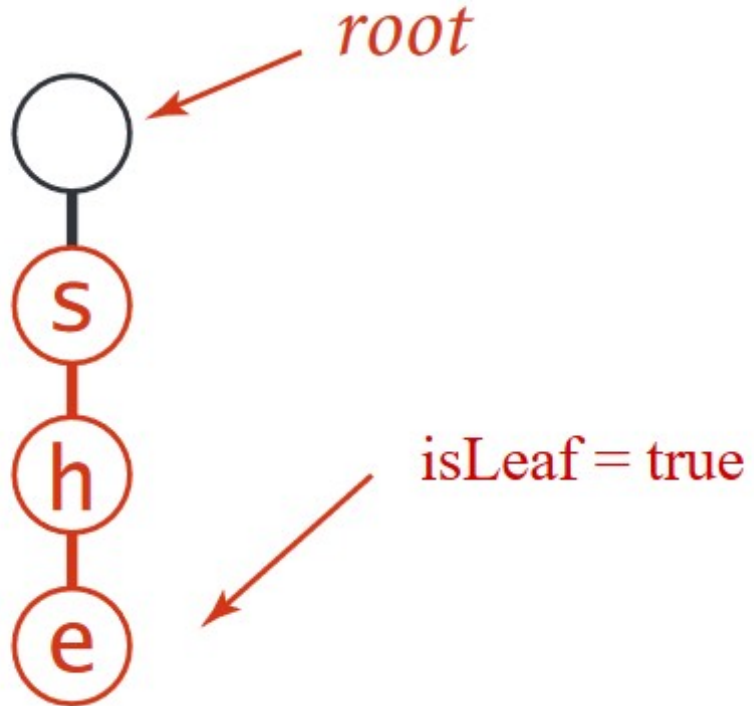


Passos do Procedimento:

1. É realizada uma pesquisa na estrutura:
 - a. Chave já se encontra dentro da árvore (nada é feito)
 - b. Chave não existe na árvore (caracteres serão inseridos)
2. Os caracteres são inseridos a partir do maior prefixo já existente na Trie.
3. O nó referente ao último caractere recebe **isLeaf=true**;

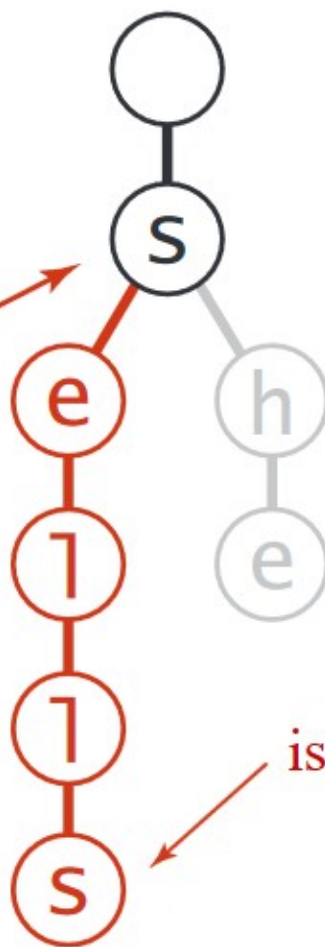


key: she



key: sells

inserção a partir
do maior prefixo



isLeaf = true

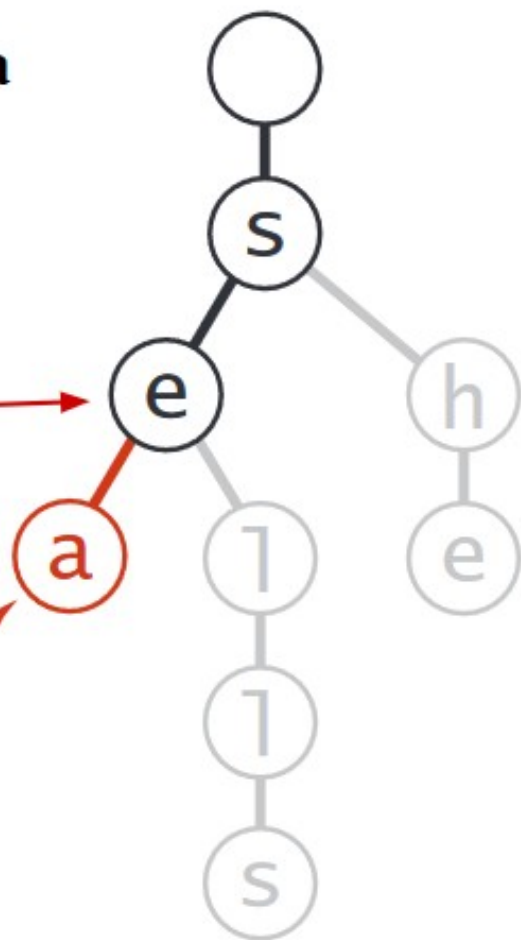


key: sea

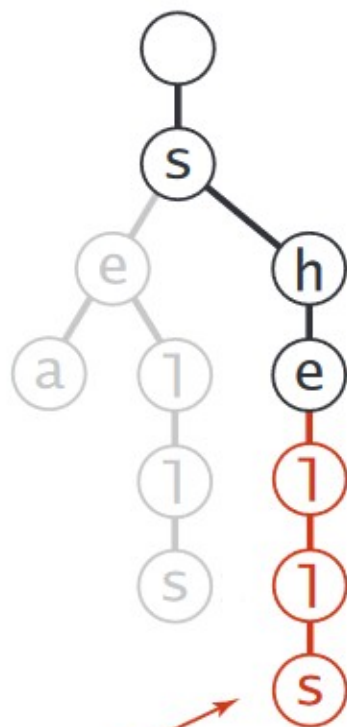


inserção a partir
do maior prefixo

isLeaf = true



key: shells

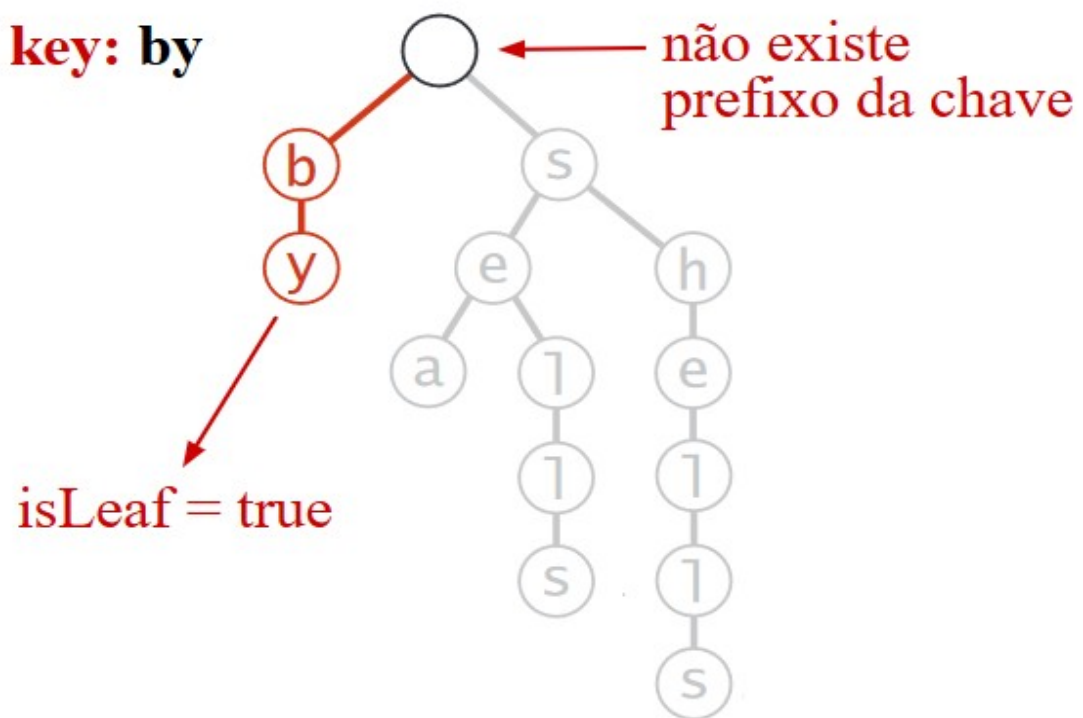


inserção a partir
do maior prefixo

isLeaf = true



key: by



Corretude: Inserção



A corretude no método de Inserção pode ser provada a partir de dois princípios:

- **Chave já está presente na Árvore:** Corretude Trivial demonstrada pela Pesquisa feita antes da Inserção
- **Chave não está presente na Árvore:** Corretude demonstrada a partir do método de Inserção que retorna o nó referente ao último caractere (Indução)

Análise de Complexidade: Inserção



Pior Caso: $O(n)$: corresponde ao tamanho da chave

Caso Médio: $O(n)$: prefixo da chave já presente \longrightarrow P.A. $(n+1)/2$

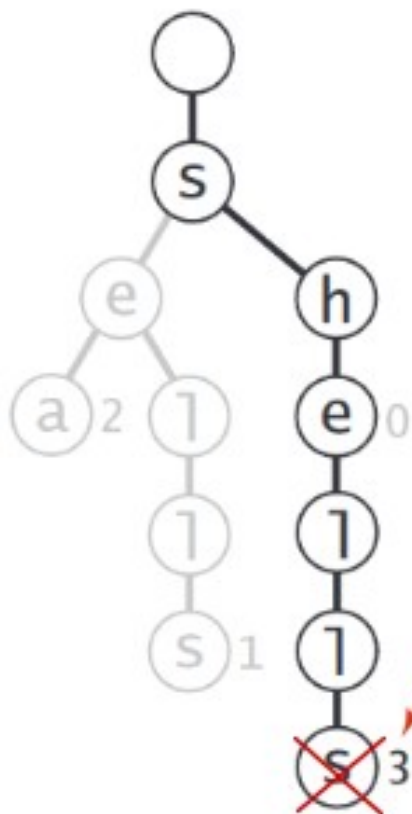
Melhor Caso: $O(1)$: chave já está na Árvore (**isLeaf=true**)

Remoção em Tries



1. Realiza-se uma pesquisa na estrutura:
 - a. Chave não existe na Árvore (nada é feito)
 - b. Chave existe na Árvore (nó do último caractere é retornado)
2. Nó do último caractere é marcado com valor nulo ou `isLeaf=false`
3. Realiza-se uma sub-pesquisa nos nós anteriores ao último:
 - a. `if (TrieNode.isLeaf==FALSE)` —→ nó é removido
 - b. `if (TrieNode.isLeaf==TRUE)` —→ nó é mantido

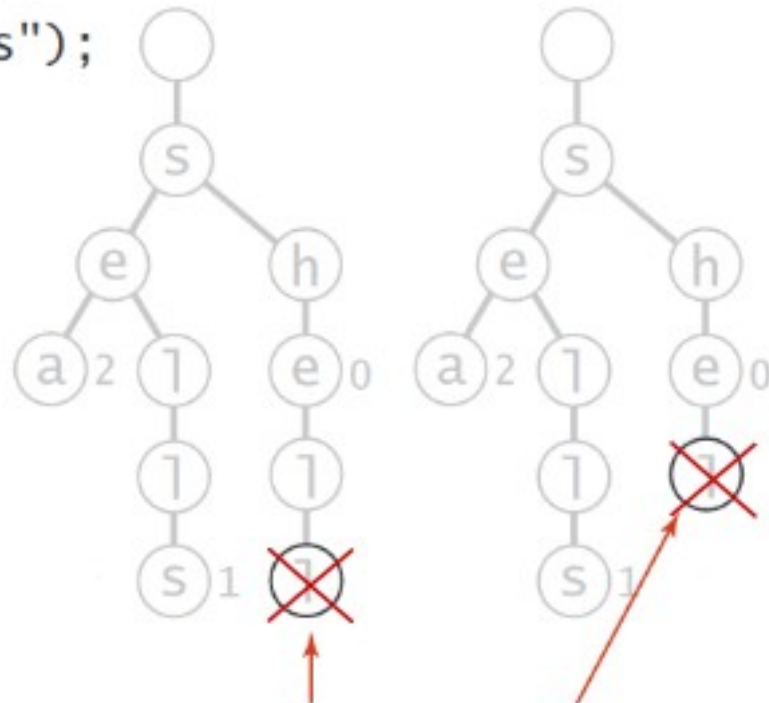
```
delete("shells");
```



if (TrieNode.isLeaf == false)
nó é removido;

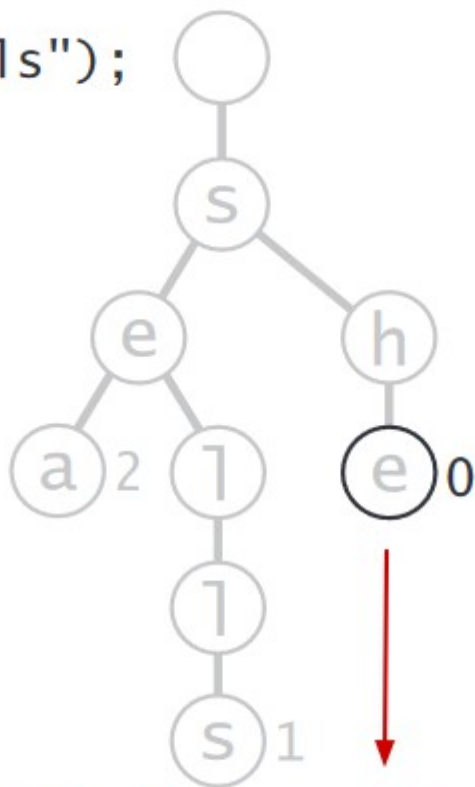
set value = null
ou
isLeaf = false

`delete("shells");`



`if (TrieNode.isLeaf == false)`
nó é removido;

delete("shells");



if (TrieNode.isLeaf == true OR value != null)
 nó é mantido;

Corretude: Remoção



A corretude no método de Remoção pode ser provada a partir de dois princípios:

- **Chave \neq Árvore:** Link nulo é retornado (Trivial)
- **Chave \exists Árvore:** O último nó não-folha removido é retornado (Indução nas subárvores). Veracidade pode ser visualizada a partir de nova pesquisa.

Análise de Complexidade:

Remoção



Pior Caso: $O(n)$: corresponde ao tamanho da chave

Caso Médio: $O(n)$: existem nós folhas no caminho da chave
P.A. $(n+1)/2$

Melhor Caso: $O(0)$: chave não existe na Árvore

Implementação: Tries

Demonstração de Operações sobre Trie

Implementação: JAVA

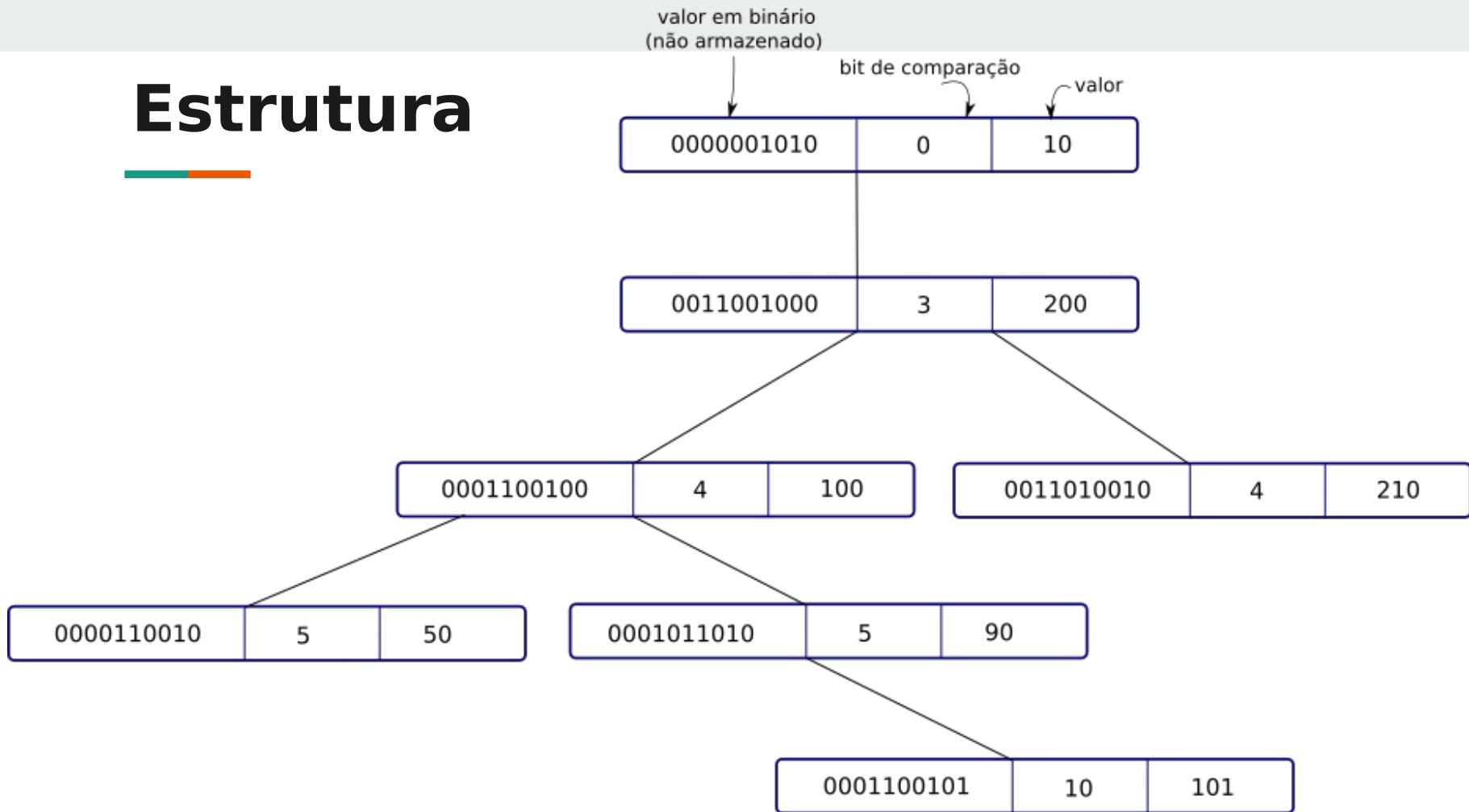




Árvores Patricia

- Practical Algorithm To Retrieve Information Coded In Alphanumeric
- Donald Morrison em 1968

Estrutura





Árvores Patricia

- Trie Compactada Binária
- (Árvore binária chique)



Trie Compactada Binária

- Na Trie cada nó ocupa o espaço do alfabeto, vários nós são necessários para armazenar uma chave.
- Na Patricia 1 nó = 1 chave.



Árvore binária chique

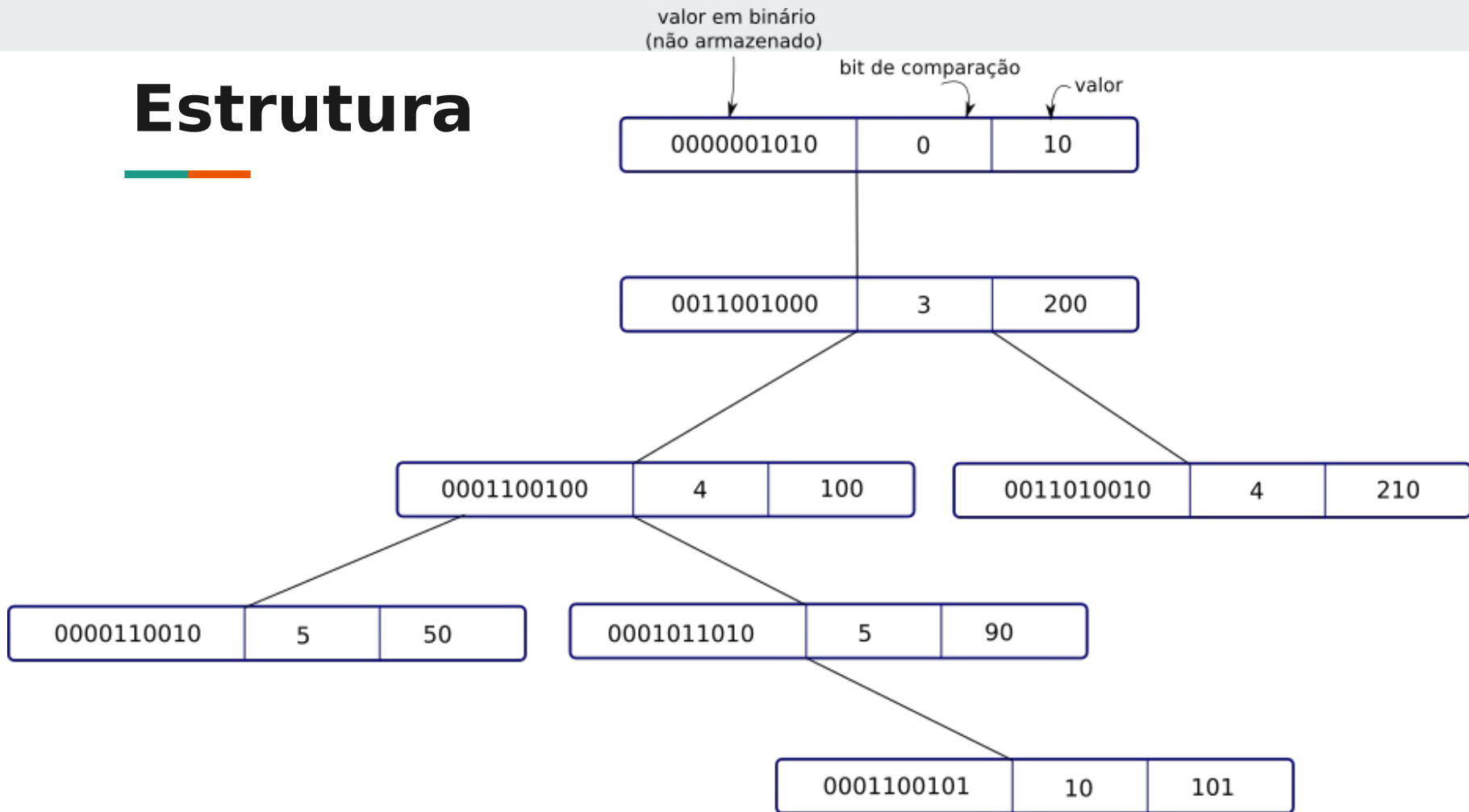
- Dois filhos por nó
- Segue os mesmos princípios de ordenação
- Armazena qualquer coisa



Árvore binária chique

- Comparação em cadeias binárias
- Bit diferenciador
- O primeiro nó NÃO tem dois filhos

Estrutura





O bit diferenciador e as cadeias binárias



Converte os valores

- Inserir 496
 - $496 = 111110000$
- Primeiro nó 502
 - $502 = 111110110$



Complete com zeros a esquerda

- Se necessário

Compare os bits



● 496 = 1 1 1 1 1 0 0 0 0

● 502 = 1 1 1 1 1 0 1 1 0



Bit de comparação = 0



Compare os bits



● 496 = 1 1 1 1 1 0 0 0 0

● 502 = 1 1 1 1 1 0 1 1 0



Bit de comparação = 1

Compare os bits



● 496 = 1 1 1 1 1 0 0 0 0

● 502 = 1 1 1 1 1 0 1 1 0



Bit de comparação = 2

Compare os bits



● 496 = 1 1 1 1 1 0 0 0 0

● 502 = 1 1 1 1 1 0 1 1 0



Bit de comparação = 3

Compare os bits



- 496 = 1 1 1 1 1 0 0 0 0

- 502 = 1 1 1 1 1 0 1 1 0



Bit de comparação = 4



Compare os bits

• 496 = 1 1 1 1 1 0 0 0 0

• 502 = 1 1 1 1 1 0 1 1 0

Bit de comparação = 5

Compare os bits

Está diferente essa desgraça!!!



● 496 = 1 1 1 1 1 0 0 0 0

● 502 = 1 1 1 1 1 0 1 1 0



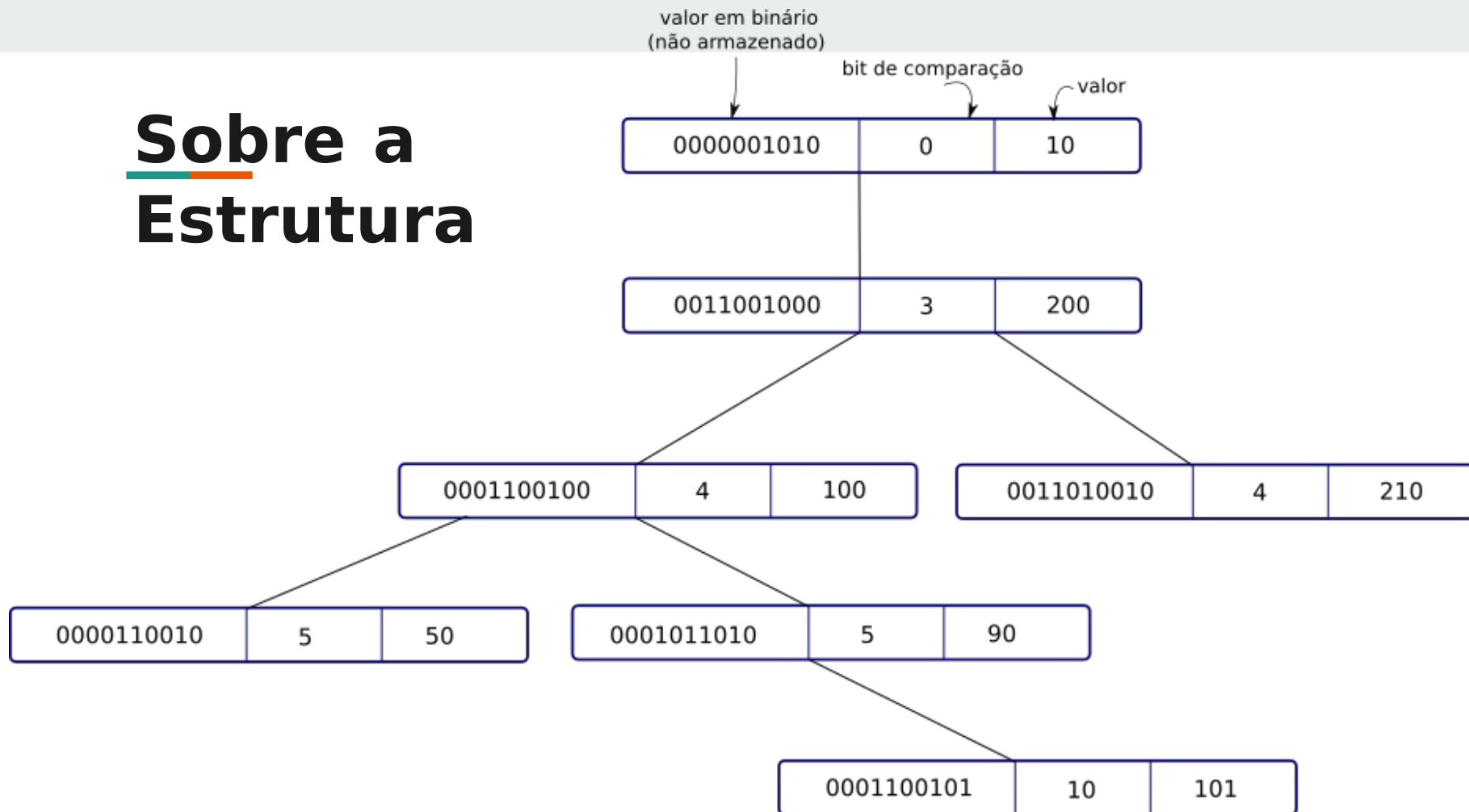
Bit de comparação = 6



Armazenamento

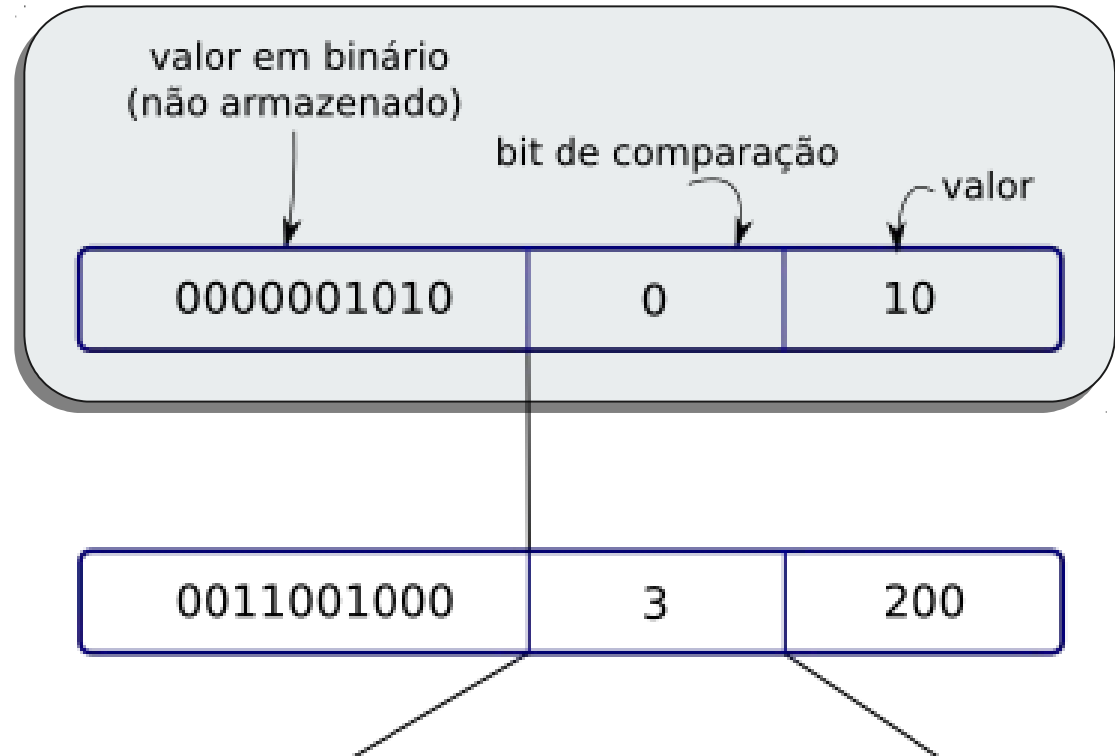
- Valor no formato original
 - Leitura mais fácil
- Valor em binário
 - Pesquisa mais veloz

Sobre a Estrutura



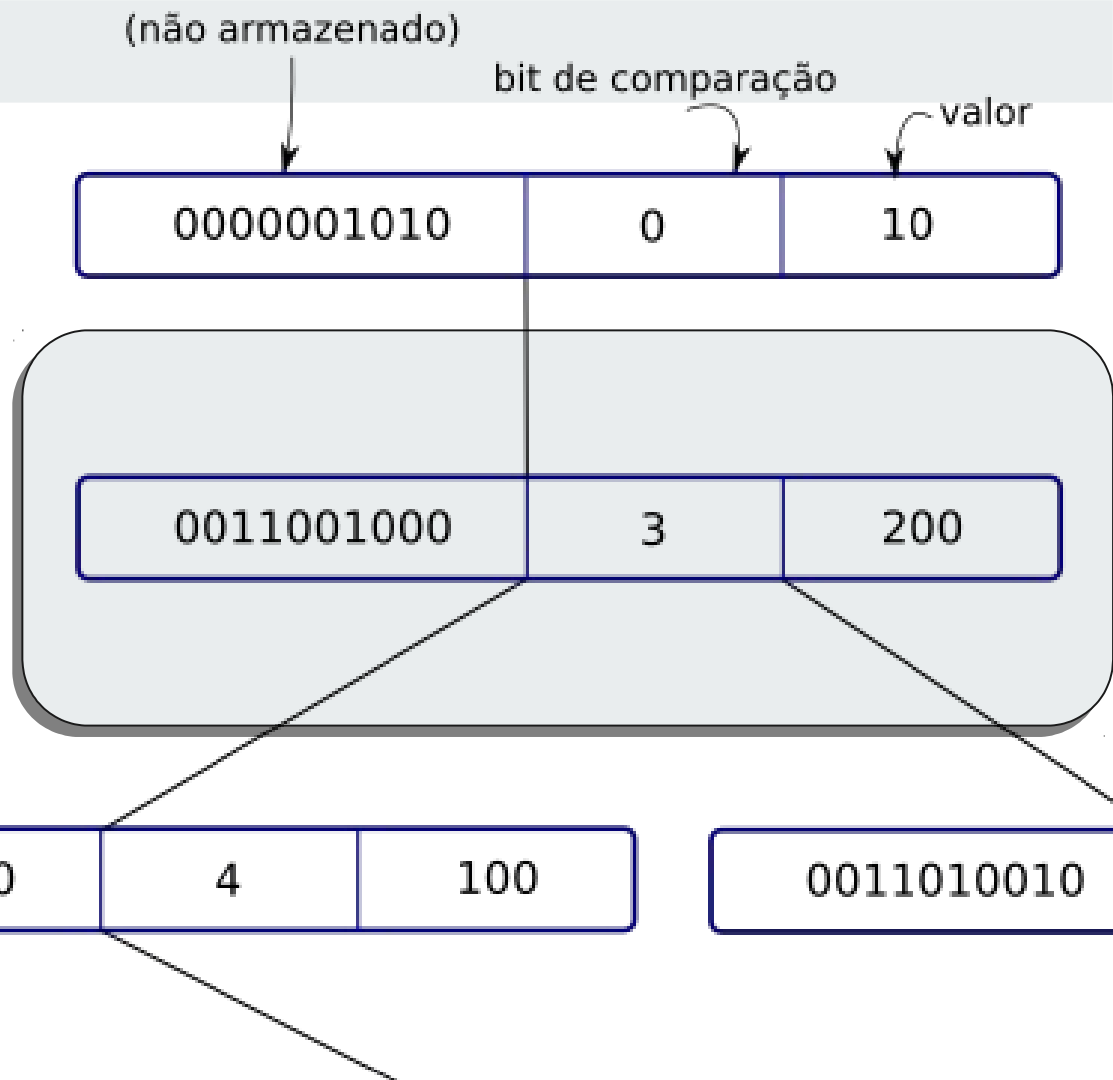
Primeiro nó

- Bit de comparação
 - Zero
- Filhos
 - Só 1



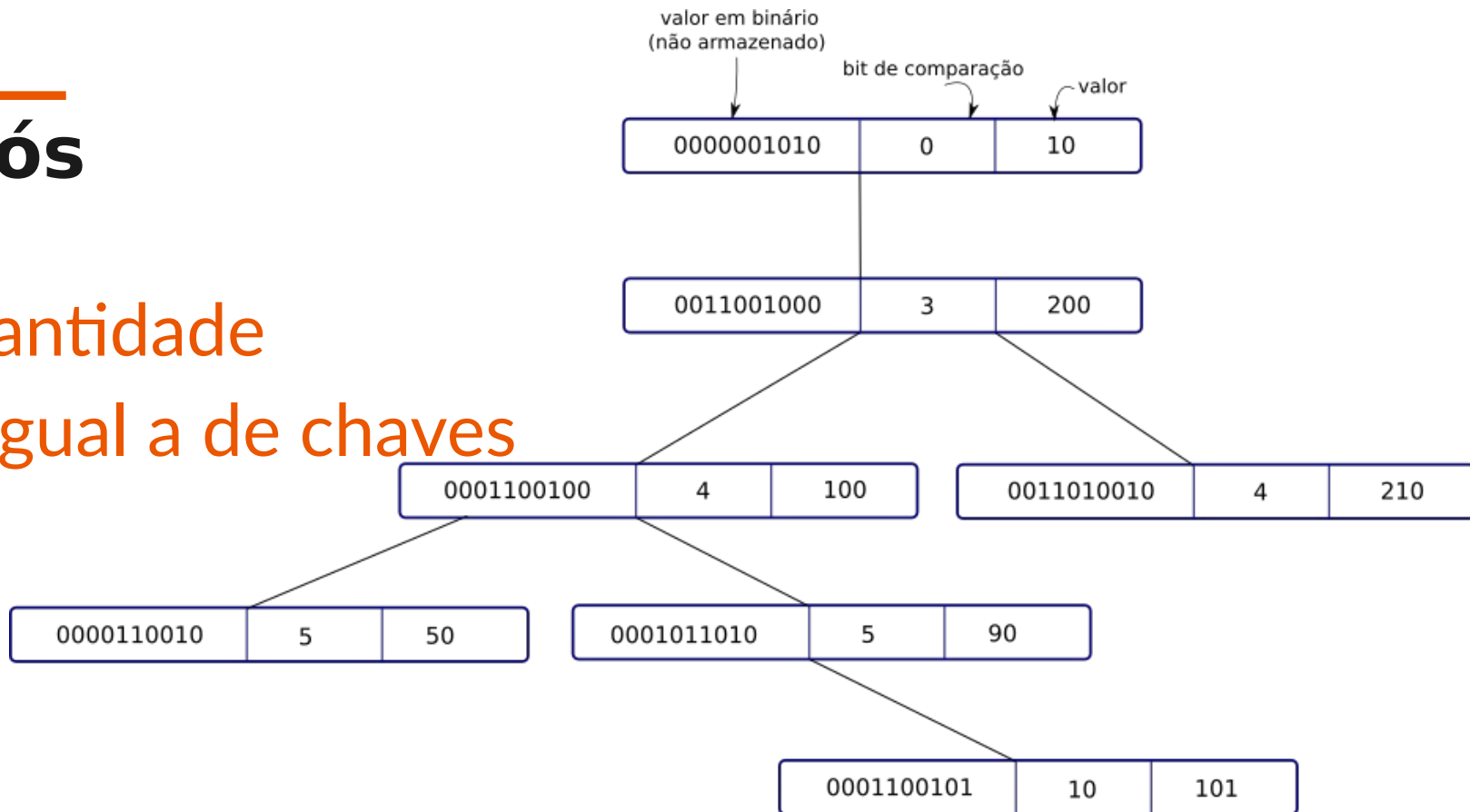
Outros nós

- Bit de comparação
 - \neq zero
 - Maior que do pai
- Filhos
 - Até 2



Nós

- Quantidade
- Igual a de chaves





Implementação e algoritmos




O nó

```
class PatriciaNode {  
    int bitNumber;  
    int data;  
    PatriciaNode leftChild, rightChild;  
}
```


Pesquisa

```
private PatriciaNode search(PatriciaNode node, int value) {  
  
    PatriciaNode currentNode, nextNode;  
  
    if (node == null) return null;  
  
    currentNode = node;  
    nextNode = node.leftChild;  
  
    String binaryValue = toBinary(value);  
    while (nextNode.bitNumber > currentNode.bitNumber) {  
        currentNode = nextNode;  
  
        if (isBit1At(binaryValue, nextNode.bitNumber)) {  
            nextNode = nextNode.rightChild;  
        } else {  
            nextNode = nextNode.leftChild;  
        }  
    }  
  
    return nextNode;  
}
```





valor em binário
(não armazenado)

bit de comparação

valor

Procurando por 90:

90 em binário é 0001011010

0000001010	0	10
------------	---	----

Posição 3 de 0001011010 é 0:

Vá para a esquerda

0011001000	3	200
------------	---	-----

Posição 4 de 0001011010 é 1:

Vá para a direita

0001100100	4	100
------------	---	-----

0011010010	4	210
------------	---	-----

0000110010	5	50
------------	---	----

0001011010	5	90
------------	---	----

Valor encontrado!!

0001100101	10	101
------------	----	-----



Inserção

- Pesquisa antes se a chave existe
- Se não existir a pesquisa retorna um candidato a pai/mãe
- Código mais extenso



Inserindo por 220:

220 em binário é 0011011100

valor em binário
(não armazenado)

bit de comparação

valor

0000001010	0	10
------------	---	----

Posição 3 de 0011011100 é 1:
Vá para a direita

0011001000	3	200
------------	---	-----

0001100100	4	100
------------	---	-----

0011010010	4	210
------------	---	-----

Posição 4 de 0011011100 é 1:
Insira na direita!

0000110010	5	50
------------	---	----

0001011010	5	90
------------	---	----

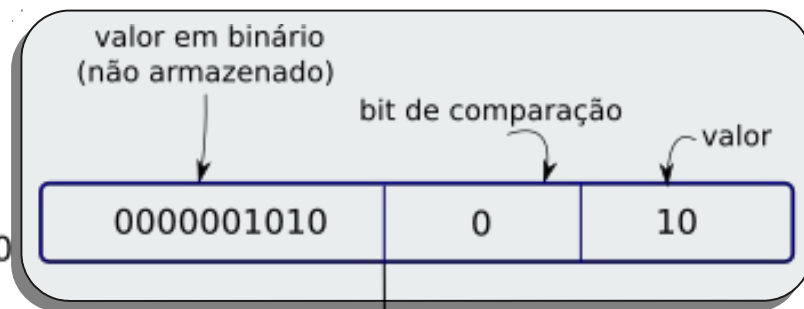
0011011100	5	220
------------	---	-----

0001100101	10	101
------------	----	-----



Inserindo por 220:

220 em binário é 0011011100



Posição 3 de 0011011100 é 1:
Vá para a direita



Posição
Insira n



valor em binário
(não armazenado)

bit de comparação

valor

Inserindo por 220:

220 em binário é 0011011100

0000001010	0	10
------------	---	----

Posição 3 de 0011011100 é 1:
Vá para a direita

0011001000	3	200
------------	---	-----

0001100100	4	100
------------	---	-----

0011010010	4	210
------------	---	-----

Posição
Insira n

5	50
---	----

0001011010	5	90
------------	---	----

0011011100	5
------------	---



é 1:

0011001000	3	200
------------	---	-----

00	4	100
----	---	-----

0011010010	4	210
------------	---	-----

Posição 4 de 0011011100 é 1:
Insira na direita!

0001011010	5	90
------------	---	----

0011011100	5	220
------------	---	-----

0001100101	10	101
------------	----	-----



00

0011010010	4	210
------------	---	-----

Posição 4 de 0011011100 é 1:
Insira na direita!

5	90
---	----

0011011100	5	220
------------	---	-----

001100101	10	101
-----------	----	-----



Exclusão

- Pesquisa antes se a chave existe
- Localiza o sucessor
- Substitui
- Código mais extenso ainda



valor em binário
(não armazenado)

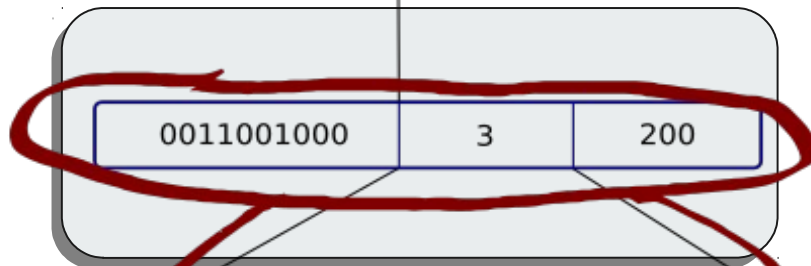
bit de comparação

valor

Removendo 200

200 em binário é 0011001000

0000001010	0	10
------------	---	----



0001100100	4	100
------------	---	-----

0011010010	4	210
------------	---	-----

Profundidade 1 - Descartado!!!

0000110010	5	50
------------	---	----

0001011010	5	90
------------	---	----

0001100101	10	101
------------	----	-----

Profundidade 3 - Selecionado!!!

Sucessor



valor em binário
(não armazenado)

bit de comparação

valor

Removendo 200

200 em binário é 0011001000

Substituição

0000001010	0	10
------------	---	----

0001100101	3	101
------------	---	-----

0001100100	4	100
------------	---	-----

0011010010	4	210
------------	---	-----

0000110010	5	50
------------	---	----

0001011010	5	90
------------	---	----

Análise de Complexidade



- Pior caso para pesquisa: $O(\log n)$: Onde “n” corresponde a quantidade de elementos na estrutura;
- Melhor Caso: $O(1)$: Ocorre quando o primeiro nó é o nó que se deseja encontrar / excluir.



Para mais informações

Para acessar os códigos fontes com as implementações correspondentes acesse: <https://github.com/ensismoebius/algoritmos>

Os códigos fontes estão sob a licença GPL versão 3.

Este documento está licenciado sob a licença Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0).

Referências Bibliográficas



- Árvores Trie e Patricia. Disponível em <
<https://profschreiner.files.wordpress.com/2015/01/arvoredigital.pdf>>. Acesso em 10 de Abril de 2019.
- Busca Digital (Trie e Árvore Patrícia). Disponível em: <
http://www.ufjf.br/jairo_souza/files/2009/12/6-Strings-Pesquisa-Digital.pdf>. Acesso em 28 de Abril de 2019.
- Compressed tries (Patricia tries). Disponível em:<
<http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie02.html>>. Acesso em 5 de maio de 2019.
- Compressing Radix Trees Without (Too Many) Tears. Disponível em: <
<https://medium.com/basecs/compressing-radix-trees-without-too-many-tears-a2e658adb9a0>>. Acesso em 13 de maio de 2019.
- Da Silva, Osmar .Q., **Estrutura de Dados e Algoritmos Usando C - Fundamentos e Aplicações**. Rio de Janeiro, CIÊNCIA MODERNA, 2007.
- Dinesh P. Mehta, Sartaj Sahni. **Handbook of data structures and applications**: CHAPMAN & HALL/CRC, 2005.

Referências Bibliográficas



- Drozdek Adam, **Data Structures and algorithms in C++**, Boston, Cengage Learning, 2005.
- Fundamental Data Structures. Disponível em: <<http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf>>. Acesso em 29 abril 2019.
- Goodrich Michael and Tamassia Roberto, Trad. Copstein Bernardo, Pompermeier B. Leandro. **Estrutura de dados e algoritmos em Java**: Porto Alegre, BOOKMAN, 2007.
- Java Algorithms and Clients. Disponível em: <<https://algs4.cs.princeton.edu/code/>>. Acesso em 13 de maio de 2019.
- llvm-project - Revision 360608. Disponível em: <<https://llvm.org/svn/llvm-project/test-suite/trunk/MultiSource/Benchmarks/MiBench/network-patricia/>>. Acesso em 1 maio 2019.
- Pesquisa digital. Disponível em <<https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsearch.pdf>>. Acesso em 11 de abril de 2019.

Referências Bibliográficas



- Radix Tree Reference. Disponível em: <<http://www.voidcn.com/article/p-zrnpwmrn-tn.html>>. Acesso em 12 maio 2019.
- Sedgewick Robert, **Algorithms in C Third Edition**, Boston, Princeton University, 1946.
- Sedgewick Robert, Wayne Kevin, **Algorithms fourth edition**: Boston, Princeton University, 2011.
- Tenenbaum Aaron , Langsam Yedidyah, Augenstein Moshe J., Trad. Souza Tereza C. P. **Estruturas de Dados Usando C**. São Paulo, MAKRON Books, 1995.
- Tries (árvores digitais). Disponível em: <<https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html>>. Acesso em 29 abril 2019
- Tries. Disponível em:<<https://algs4.cs.princeton.edu/lectures/52Tries.pdf>>. Acesso em 10 de maio de 2019.
- Ziviani Nivio, **Projetos de Algoritmos Com Implementações em Pascal e C 4 ed**. São Paulo, Thomson Pioneira, 2007.
- What is the difference between radix trees and Patricia tries?. Disponível em:<
<https://cs.stackexchange.com/questions/63048/what-is-the-difference-between-radix-trees-and-patricia-tries>
> Acesso em 3 de maio de 2019