Ternary Tries

Referências

- Tries (S&W),
- slides (S&W)

Vídeo

Tries (S&W)

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` × `R` bytes. Veja alguns valores de `R` para de de alguns alfabeto na classe `Alphabet`. Essa tabela, entre outras coisas, foi copiada da página Alfabetos e a classe Alphabet (PF):

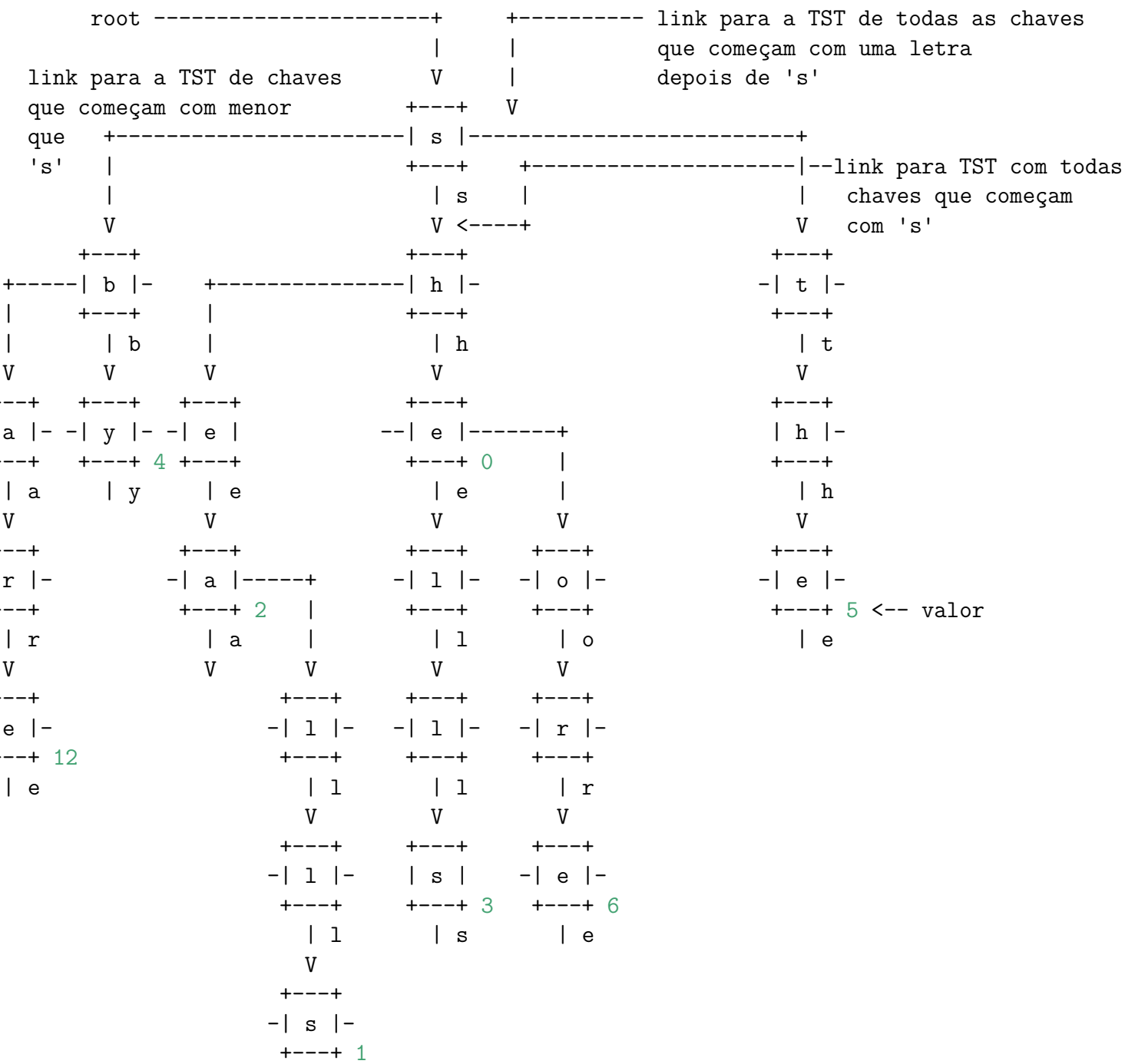
nome	R()	lg(R)	conjunto de caracteres
`BINARY`	2	1	`01`
`DNA`	4	2	`ACTG`
`OCTAL`	8	3	`01234567`
`DECIMAL`	10	4	`0123456789`
`HEXADECIMAL`	16	4	`0123456789ABCDEF`
`PROTEIN`	20	5	`ACDEFGHIKLMNPQRSTVWY`
`LOWERCASE`	26	5	`abcdefghijklmnopqrstuvwxyz`
`UPPERCASE`	26	5	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
`ASCII`	128	7	alfabeto ASCII
`EXTENDED_ASCII`	256	8	alfabeto ASCII estendido
`UNICODE16`	65536	16	alfabeto Unicode

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

```
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"
```

Exemplo

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



```
                          | s
```

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

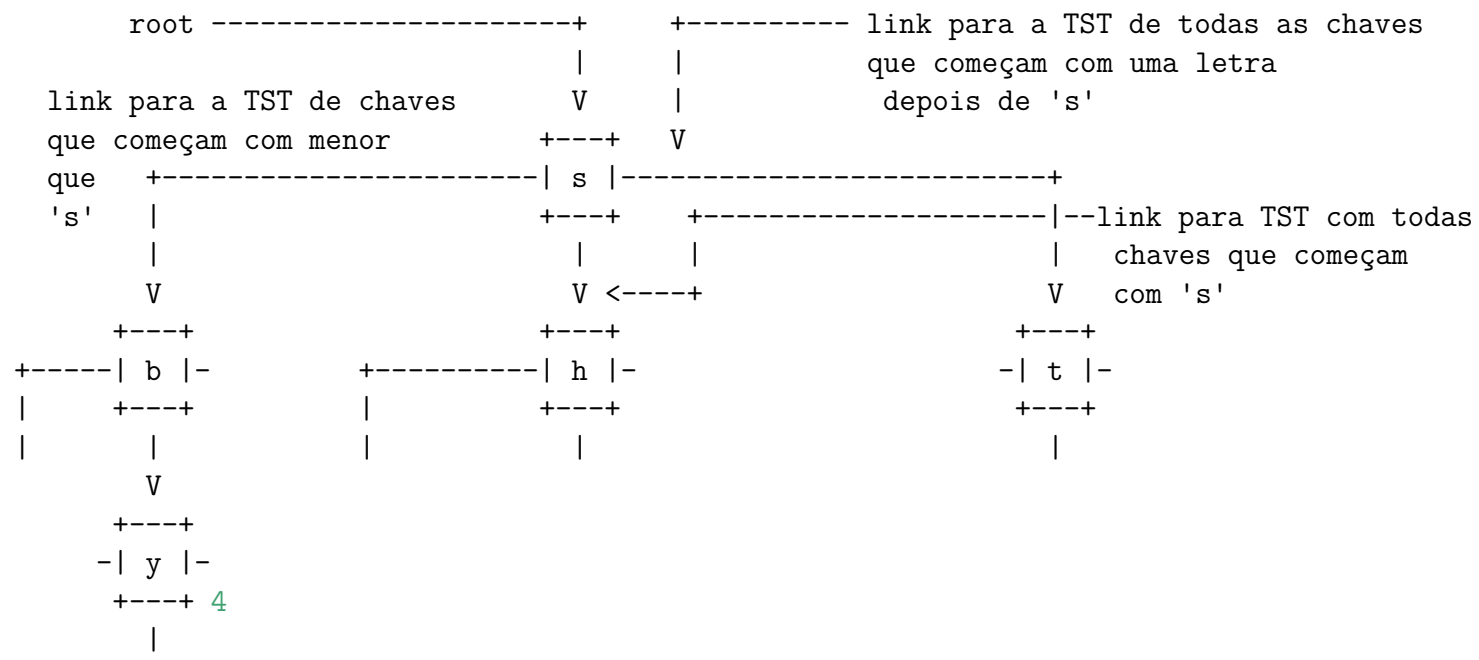
- chaves ficam codificadas nos caminhos que começam na raiz;
- prefixos de chaves, que nem sempre são chaves, estão representados na TST.

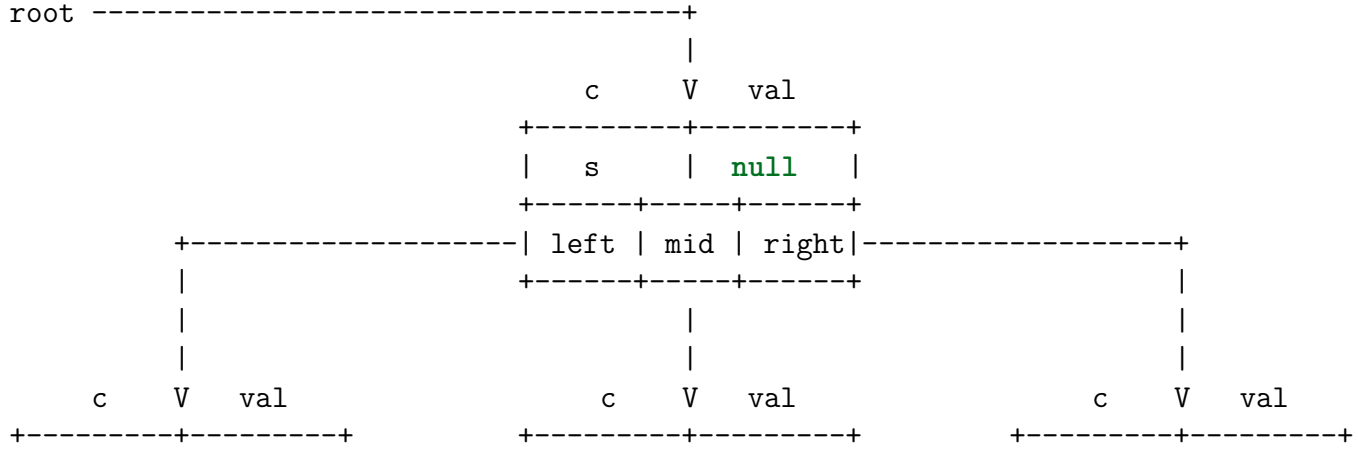
Estrutura de uma trie ternária

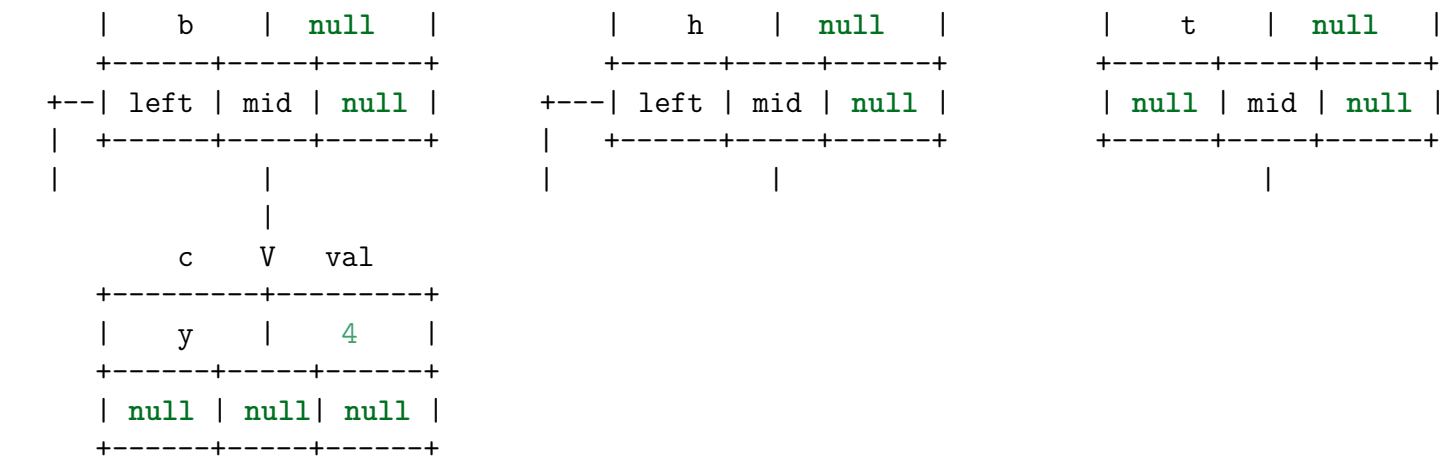
Os links da estrutura correspondem a caracteres e não a chaves. Nas figuras, o caractere escrito dentro de um nó é o caractere do link que sai pelo meio do nó.

TSTs são compostas por nós do tipo `Node`.

```
private class Node {
    private char c;                 // character
    private Node left, mid, right; // left, middle, and right subtries
    private Value val;              // value associated with string
}
```







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`.

Implementação

```
public class TST<Value> {

   private Node root;

   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() {...}

}
```

Busca

Seguimos os ponteiros soletrando a string `key`.

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

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

Inserção

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

Exemplos: she sells sea shells by the sea shore

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

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

```java
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:

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

```
// TST
private Node delete(Node x, String k, 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], k, d+1);
   }
   if (x.val != null) return x;
   for (char c = 0; c < R; c++)
      if (x.next[c] != null) return x;
   return null;
}
```

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.

```
// TST
// all keys in subtrie rooted at x with given prefix
private void collect(Node x, String prefix, Queue<String> q) {
    if (x == null) return;
    collect(x.left,  prefix,       q);
    if (x.val != null) q.enqueue(prefix + x.c); // ordem lexicográfica
    collect(x.mid,   prefix + x.c, q);
    collect(x.right, prefix,       q);
}
```

keysWithPrefix()

```
// TST
public Iterable<String> keysWithPrefix(String prefix) {
    Queue<String> queue = new Queue<String>();
    Node<Value> x = get(root, prefix, 0);
    if (x == null) return queue;
    if (x.val != null) queue.enqueue(prefix);
    collect(x.mid, prefix, queue);
    return queue;
}
```

keys()

```
// TrieST
public Iterable<String> keys() {
```

```
    return keysWithPrefix("");
}


// TST
// all keys in symbol table
public Iterable<String> keys() {
    Queue<String> queue = new Queue<String>();
    collect(root, "", queue);
    return queue;
}
```

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);
}

/***************************************************************
* Find and return longest prefix of s in TST
***************************************************************/
public String longestPrefixOf(String s) {
    if (s == null || s.length() == 0) return null;
    int length = 0;
    Node x = root;
    int i = 0;
    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) length = i;
            x = x.mid;
        }
    }
    return s.substring(0, length);
}

// return all keys matching given wilcard pattern
public Iterable<String> wildcardMatch(String pat) {
```

```
    Queue<String> queue = new Queue<String>();
    collect(root, "", 0, pat, queue);
    return queue;
}

public void collect(Node x, String prefix, int i, String pat, Queue<String> q) {
    if (x == null) return;
    char c = pat.charAt(i);
    if (c == '.' || c < x.c) collect(x.left, prefix, i, pat, q);
    if (c == '.' || c == x.c) {
        if (i == pat.length() - 1 && x.val != null) q.enqueue(prefix + x.c);
        if (i < pat.length() - 1) collect(x.mid, prefix + x.c, i+1, pat, q);
    }
    if (c == '.' || c > x.c) collect(x.right, prefix, i, pat, q);
}
```

Análise

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 $\lg n$.