

Tries

Definição e Construção

Prof. Edson Alves - UnB/FGA

1. Definição
2. Construção *naive* $O(N^2)$ da *trie*
3. Construção *online* da *trie*

Definição

- Árvores de sufixos são estruturas de dados que representam o conjunto $B(s)$ de todas as substrings de uma string s dada
- A relação de pertinência ($r \in B(s)?$) é o mais básico problema associado a esta estrutura
- Uma “boa” árvore de sufixos tem três características fundamentais:
 1. pode ser construída com tamanho linear
 2. pode ser construída em tempo linear
 3. pode responder questão de pertinência em complexidade linear em relação ao tamanho de s

Conceitos elementares

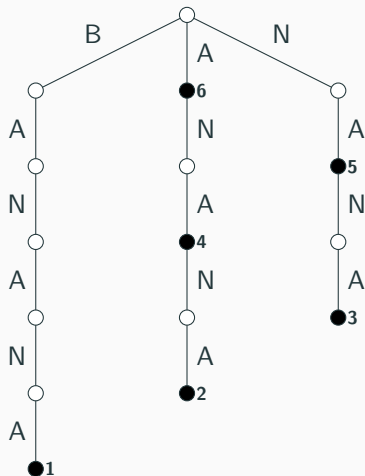
- Seja G um grafo acíclico direcionado, com raiz, cujas arestas recebem, como rótulos, caracteres ou palavras de um alfabeto A de tamanho constante
- Seja $label(e)$ o rótulo da aresta e
- O rótulo de um caminho p é a concatenação dos rótulos de todas as arestas do caminho
- Tal grafo representa um conjunto de strings, que são definidas pelos rótulos de todos os caminhos possíveis em G
- Seja

$$\mathcal{L}(G) = \{label(p) \mid p \text{ é caminho em } G \text{ com início na raiz}\}$$

- G representa todas as substrings de s se $\mathcal{L}(G) = B(s)$
- Um nó n cujo caminho da raiz até n tem como rótulo um sufixo de s é denominado nó essencial

- Uma *trie* de substrings de s , ou simplesmente *trie*, é o grafo G que representa todas as substrings de s , cujos rótulos consistem apenas de um único caractere
- O nome foi cunhado em 1961 por Edward Fredkin, a partir da sílaba central da palavra *retrieval*
- A pronúncia é idêntica a palavra *tree*, mas a grafia é diferente para diferenciar esta estrutura das árvores em geral
- A próxima figura ilustra a *trie* da palavra "BANANA"
- Os nós pretos são nós essenciais
- Os números ao lado dos nós essenciais são os índices do caractere inicial do sufixo

Visualização da *trie* da palavra 'BANANA'



Construção *naive* $O(N^2)$ **da** *trie*

Construção *naive* da *trie*

- Seja s uma string de tamanho N
- Cada nó da *trie* de s pode ter até $|A|$ filhos, onde A é o alfabeto
- Assim, cada nó pode ser implementado como um vetor de pares ou como um mapa, onde o par (c, n) , indicando que há uma aresta de rótulo c que aponta para o nó n
- A raiz da árvore será o nó identificado por $n = 0$
- Para cada caractere c do sufixo $s[i..N]$, e iniciando na raiz, verifica-se se existe a aresta (c, n)
- Em caso, afirmativo, segue-se esta aresta e se processa o caractere que sucede c
- Caso não exista, cria-se um novo nó m , adiciona-se ao nó atual a aresta (c, m) , e segue o processamento para m e para o próximo caractere
- Esta construção tem complexidade $O(N^2)$
- A memória necessária também é $O(N^2)$

Implementação *naive* da *trie*

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using Node = map<char, int>;
5 using Trie = vector<Node>;
6
7 Trie build_naive(const string& s)
8 {
9     int root = 0, next = 0;
10
11     Trie trie(1);          // Instancia o nó raiz vazio
12
13     for (int i = s.size() - 1; i >= 0; --i)
14     {
15         string suffix = s.substr(i);
16         int v = root;
17
18         for (auto c : suffix)
19         {
20             auto it = trie[v].find(c);
```

Implementação *naive* da *trie*

```
22         if (it != trie[v].end())
23         {
24             v = it->second;
25         } else
26         {
27             trie.push_back({ });
28             trie[v][c] = ++next;
29             v = next;
30         }
31     }
32 }
33
34 return trie;
35 }
```

Busca de substring em um *trie*

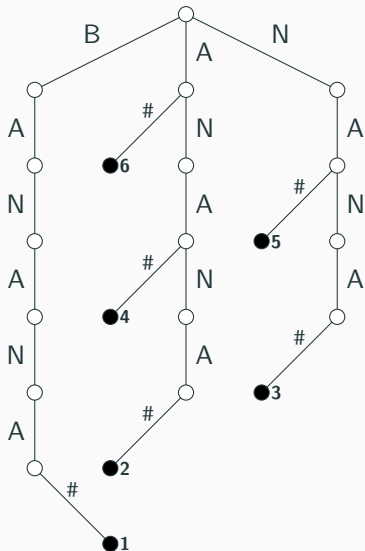
- A *trie* da string s pode ser utilizada para identificar se uma string p é ou não substring de s
- O algoritmo é semelhante à busca binária, e tem complexidade $O(m)$, onde $m = |p|$
- Por exemplo, se $s = \text{"BANANA"}$ e $p = \text{"NAN"}$, partindo da raiz, tem-se "N" na aresta à direita, "A" na única aresta e "N" na aresta seguinte: logo p é substring de s
- Como o nó de chegada é branco, p não é sufixo de s
- Para $p = \text{"NAS"}$, o último caractere ("S") não seria encontrado
- O mesmo vale para $p = \text{"MAS"}$, porém a falha acontece logo no primeiro caractere
- Para $p = \text{"NANAN"}$ a busca se encerraria ao atingir um nó nulo

Implementação da busca em $O(m)$ em uma *trie*

```
66 bool search(const Trie& trie, const string& s)
67 {
68     int v = 0;
69
70     for (auto c : s)
71     {
72         auto it = trie[v].find(c);
73
74         if (it == trie[v].end())
75             return false;
76
77         v = it->second;
78     }
79
80     return true;
81 }
```

- A construção proposta para a *trie* permite ao algoritmo de busca descrito apenas determinar se a substring p ocorre ou não em s
- Se for preciso determinar a posição (ou posições) desta ocorrência, é preciso modificar a construção da *trie*, de modo que seja possível discriminar os nós essenciais dos demais
- Uma maneira de fazê-lo é adicionar um caractere terminador (em geral, o caractere '#'), que não pertença a string original
- A este caractere estará associado o índice i da string tal que o sufixo terminado no marcador é igual a $S[i..N]$
- Importante notar que o segundo elemento do par terá dois significados distintos: ou será um ponteiro para o próximo nó, ou o índice do sufixo caso o rótulo da aresta seja o terminador
- É preciso atentar a esta diferença na implementação das rotinas de construção e busca

Visualização da *trie* da palavra 'BANANA' com terminador



Construção da *trie* com marcador

```
83 Trie build_naive_with_marker(const string& s)
84 {
85     int root = 0, next = 0;
86
87     Trie trie(1);      // Instancia o nó raiz vazio
88
89     for (int i = s.size() - 1; i >= 0; --i)
90     {
91         string suffix = s.substr(i) + '#';
92         int v = root;
93
94         for (auto c : suffix)
95         {
96             if (c == '#')
97             {
98                 trie[v][c] = i;
99                 break;
100             }
101
102             auto it = trie[v].find(c);
```


Construção da *trie* com marcador

```
104         if (it != trie[v].end())
105         {
106             v = it->second;
107         } else
108         {
109             trie.push_back({ });
110             trie[v][c] = ++next;
111             v = next;
112         }
113     }
114 }
115
116 return trie;
117 }
```

Identificação das ocorrências de uma substring em uma *trie* com marcadores

```
119 vector<int> find(const Trie& trie, const string& s)
120 {
121     vector<int> is;
122     int v = 0;
123
124     for (auto c : s)
125     {
126         auto it = trie[v].find(c);
127
128         if (it == trie[v].end())
129             return is;
130
131         v = it->second;
132     }
133
134     queue<int> q;
135     q.push(v);
136
```

Identificação das ocorrências de uma substring em uma *trie* com marcadores

```
137 while (not q.empty())
138 {
139     auto u = q.front();
140     q.pop();
141
142     for (auto [c, v] : trie[u])
143     {
144         if (c == '#')
145             is.push_back(v);
146         else
147             q.push(v);
148     }
149 }
150
151 return is;
152 }
```

Número de substrings distintas

- Outra informação que pode ser obtida a partir da *trie* é o número de substring distintas de s
- Se s tem n caracteres, ela terá $n(n+1)/2$ substrings não vazias, não necessariamente distintas
- Estas substrings correspondem a todos os pares de índices (i, j) , com $i \leq j$, onde $i, j = 1, 2, \dots, n$
- Em uma *trie*, qualquer nó, exceto a raiz, representa uma substring distinta, formada pela concatenação dos rótulos do caminho da raiz até o nó em questão

Contagem de substrings distintas em uma *trie*

```
154 size_t unique_substrings(const Trie& trie)
155 {
156     size_t count = 0;
157     queue<int> q;
158     q.push(0);
159
160     while (not q.empty()) {           // BFS para contabilizar o número de nós
161         auto u = q.front();
162         q.pop();
163
164         for (auto [c, v] : trie[u]) {
165             if (c != '#') {
166                 ++count;
167                 q.push(v);
168             }
169         }
170     }
171
172     return count;
173 }
```

Considerações sobre a construção *naive* da *trie*

- Embora as buscas apresentadas satisfaçam o terceiro critério para uma boa árvore de sufixo, os outros dois critérios não são satisfeitos
- Se a string s inicial tem N caracteres, a construção e o espaço em memória são $O(N^2)$.
- É possível melhorar a complexidade da construção da *trie*, por meio de um algoritmo *online*
- O espaço em memória, contudo, permanecerá $O(N^2)$, por conta da representação de cada caractere por meio de uma aresta
- Assim, a redução de memória só é possível por meio de uma mudança na representação dos caracteres e dos sufixos, o que leva a uma outra estrutura, a *suffix trie*

Construção *online* da *trie*

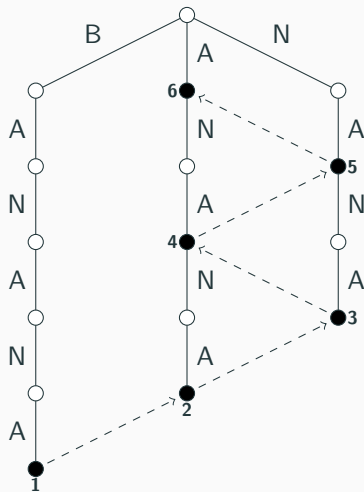
Algoritmo *online* para construção de uma *trie*

- A construção pode ser melhorada por meio de um algoritmo *online*
- A ideia principal é, ao invés de construir toda a *trie* de s de uma só vez, construí-la a partir da *trie* de $s[1..(N-1)]$
- Seja T_j a *trie* do prefixo $s[1..j]$ de s
- A principal observação a ser feita é que T_j pode ser construída a partir da inserção do caractere $s[j]$ em T_{j-1} , nas arestas dos novos nós a serem adicionados aos nós essenciais de T_{j-1} , quando for o caso
- Isto acontecerá quando o nó essencial não tem um filho ligado a ele por meio de uma aresta cujo rótulo é $s[j]$
- O ponto principal, portanto, se torna determinar a sequência dos nós essenciais $v_k, v_{k-1}, \dots, v_2, v_1, v_0$, onde v_i corresponde ao prefixo $s[1..i]$ de T_k
- Esta tarefa pode ser feita por meio do uso de *links* de sufixos

- Seja u um nó de T_k
- Defina o *link* de sufixo $suf(u) = v$, onde v é um nó cujo caminho $p(v)$ da raiz até v é igual ao caminho de $[2..p(u)]$, isto é, o caminho $p(u)$ sem o seu primeiro caractere
- Por definição, se a raiz de T_k corresponde ao vértice v_0 , então $suf(v_0) = v_0$
- Contudo, interpretar $suf(v_0)$ como **nullptr** pode ser mais interessante na implementação do algoritmo
- Esta definição leva a igualdade

$$(v_k, v_{k-1}, \dots, v_0) = (v_k, suf(v_k), suf^2(v_k), \dots, suf^{k-1}(v_k))$$

Visualização dos *links* de sufixos da *trie* da palavra ‘BANANA’



A construção *online* de T_k a partir de T_{k-1} pode ser feita por meio dos seguintes passos:

1. identifique os nós essenciais $v_{k-1}, v_{k-2}, \dots, v_1, v_0$ de T_{k-1} , em ordem decrescente em relação ao tamanho do sufixo relacionado
2. escolha os nós v_i consecutivos até que se atinja um nó v_t tal que exista um filho de v_t unido por uma aresta cujo rótulo é $s[k]$
3. para cada um dos nós escolhidos, crie novos nós filhos ligados por arestas cujos rótulos sejam $s[k]$
4. atualize os *links* de sufixos para os novos nós recém-criados

Visualização da construção *online* da *trie*

$$k = 0$$



i	v_i	$suf[v_i]$
0	0	-1

Visualização da construção *online* da *trie*

$$k = 1$$

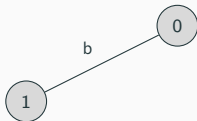
$$c = \text{'b'}$$



i	v_i	$\text{suf}[v_i]$
0	0	-1

Visualização da construção *online* da *trie*

$k = 1$
 $c = \text{'b'}$

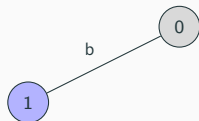


i	v_i	$\text{suf}[v_i]$
0	0	-1
1	1	0

Visualização da construção *online* da *trie*

$$k = 2$$

$$c = \text{'a'}$$



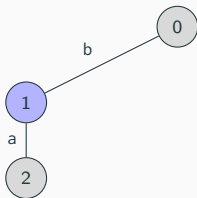
i	v_i	$\text{suf}[v_i]$
0	0	-1
1	1	0

Visualização da construção *online* da *trie*

$$k = 2$$

$$c = 'a'$$

i	v_i	$suf[v_i]$
0	0	-1
1	1	0
2	2	0

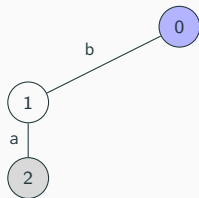


Visualização da construção *online* da *trie*

$$k = 2$$

$$c = 'a'$$

i	v_i	$suf[v_i]$
0	0	-1
1	1	0
2	2	0

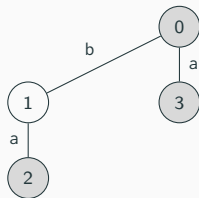


Visualização da construção *online* da *trie*

$$k = 2$$

$$c = 'a'$$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	2	3

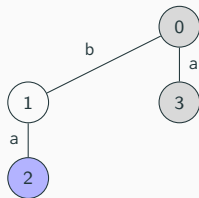


Visualização da construção *online* da *trie*

$$k = 3$$

$$c = \text{'n'}$$

i	v_i	$\text{suf}[v_i]$
0	0	-1
1	3	0
2	2	3

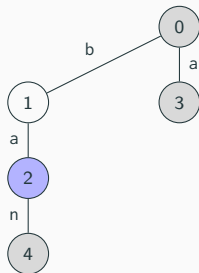


Visualização da construção *online* da *trie*

$$k = 3$$

$c = \text{'n'}$

i	v_i	$\text{suf}[v_i]$
0	0	-1
1	3	0
2	2	3
3	4	0

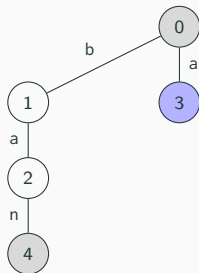


Visualização da construção *online* da *trie*

$$k = 3$$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	2	3
3	4	0

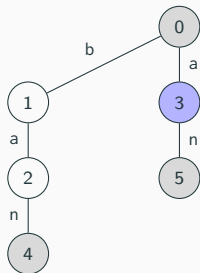


Visualização da construção *online* da *trie*

$$k = 3$$

$c = \text{'n'}$

i	v_i	$\text{suf}[v_i]$
0	0	-1
1	3	0
2	5	3
3	4	5

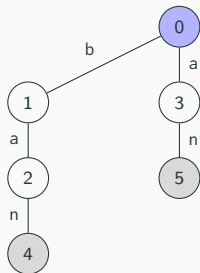


Visualização da construção *online* da *trie*

$$k = 3$$

$c = \text{'n'}$

i	v_i	$\text{suf}[v_i]$
0	0	-1
1	3	0
2	5	3
3	4	5

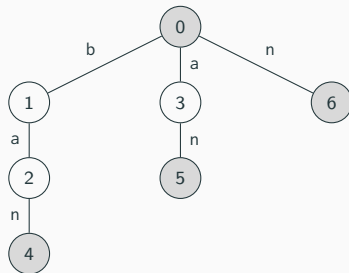


Visualização da construção *online* da *trie*

$$k = 3$$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	4	5

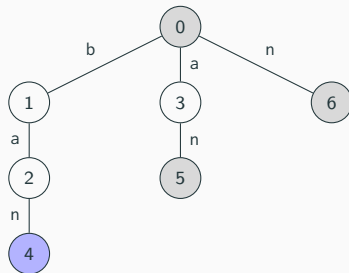


Visualização da construção *online* da *trie*

$$k = 4$$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	4	5

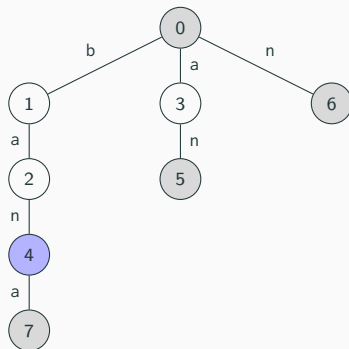


Visualização da construção *online* da *trie*

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	4	5
4	7	0

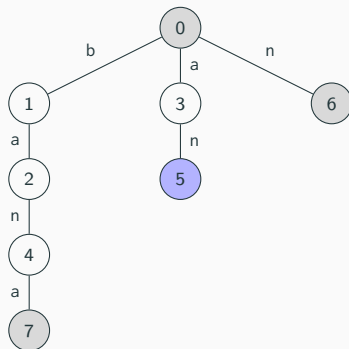


Visualização da construção *online* da *trie*

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	4	5
4	7	0

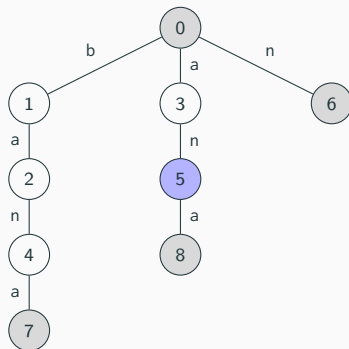


Visualização da construção *online* da trie

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	8	5
4	7	8

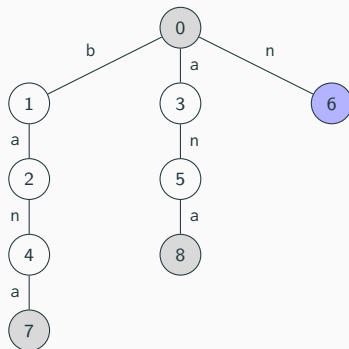


Visualização da construção *online* da trie

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	8	5
4	7	8

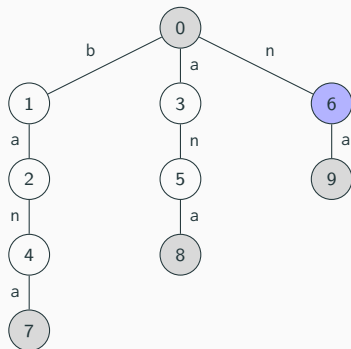


Visualização da construção *online* da *trie*

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	9	6
3	8	9
4	7	8

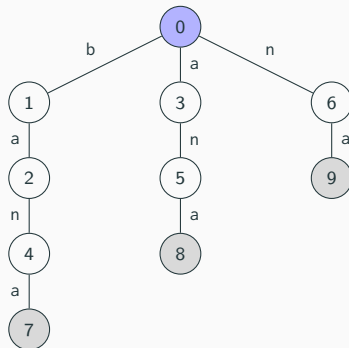


Visualização da construção *online* da *trie*

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	9	6
3	8	9
4	7	8

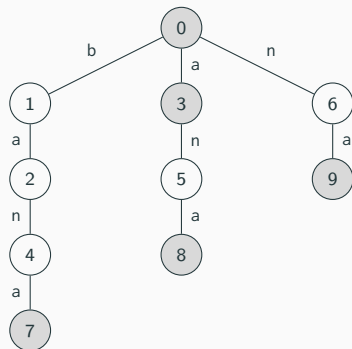


Visualização da construção *online* da *trie*

$k = 4$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	7	8

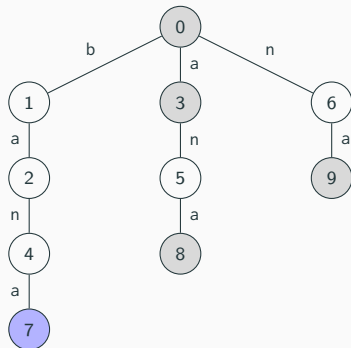


Visualização da construção *online* da *trie*

$k = 5$

$c = \text{'n'}$

i	v_i	$\text{suf}[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	7	8

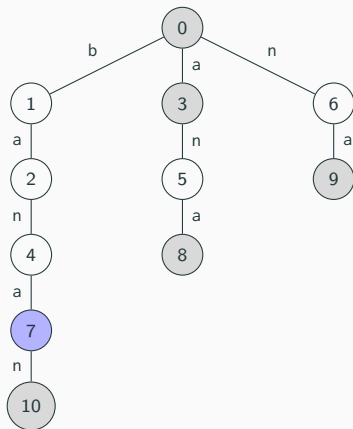


Visualização da construção *online* da *trie*

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	7	8
5	10	0

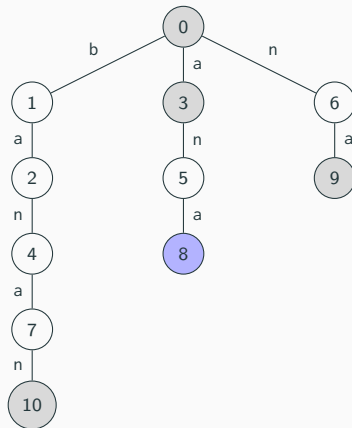


Visualização da construção *online* da trie

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	7	8
5	10	0

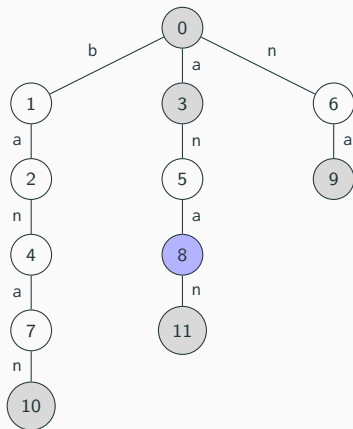


Visualização da construção *online* da trie

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	11	8
5	10	11

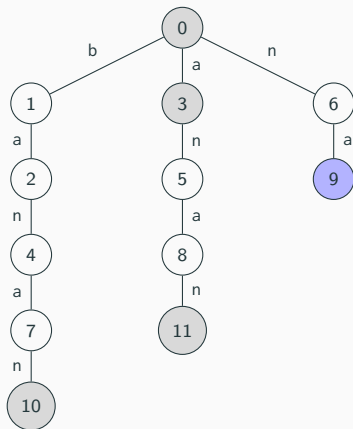


Visualização da construção *online* da *trie*

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	11	8
5	10	11

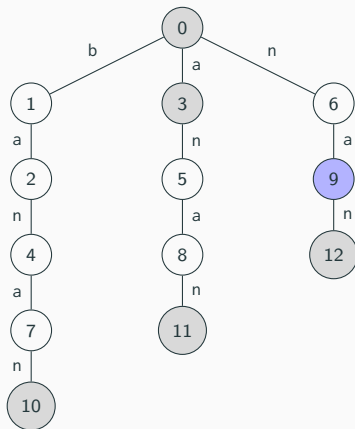


Visualização da construção *online* da *trie*

$$k = 5$$

$$c = 'n'$$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	12	9
4	11	12
5	10	11

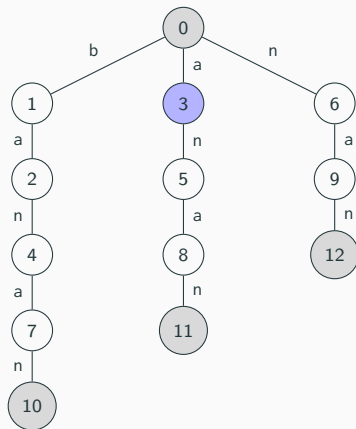


Visualização da construção *online* da *trie*

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	12	9
4	11	12
5	10	11

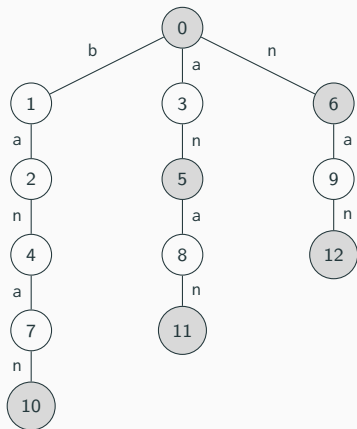


Visualização da construção *online* da *trie*

$k = 5$

$c = 'n'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	10	11

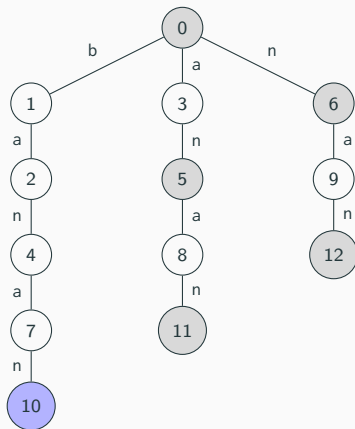


Visualização da construção *online* da *trie*

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	10	11

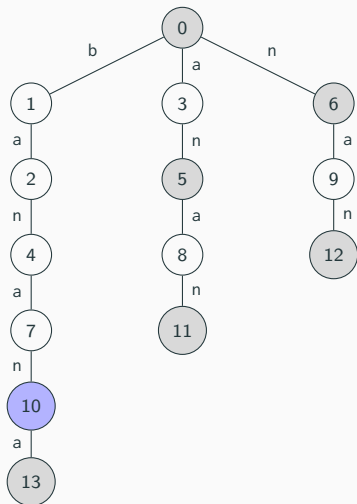


Visualização da construção *online* da trie

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	10	11
6	13	0

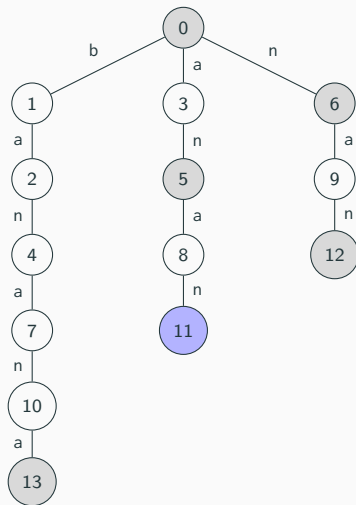


Visualização da construção *online* da trie

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	10	11
6	13	0

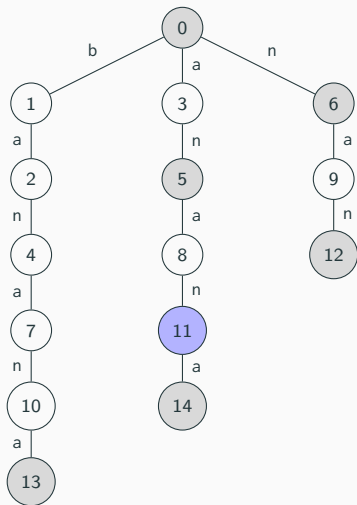


Visualização da construção *online* da trie

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	14	11
6	13	14

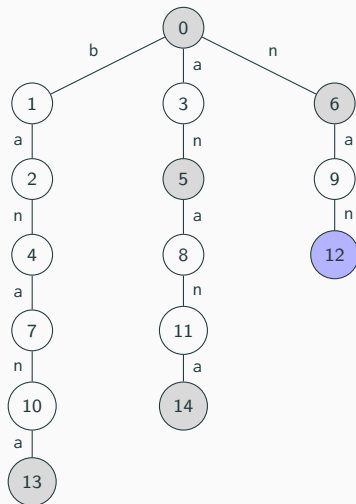


Visualização da construção *online* da trie

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	11	12
5	14	11
6	13	14

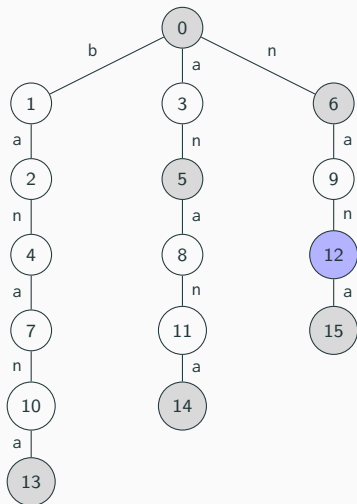


Visualização da construção *online* da trie

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	15	12
5	14	15
6	13	14

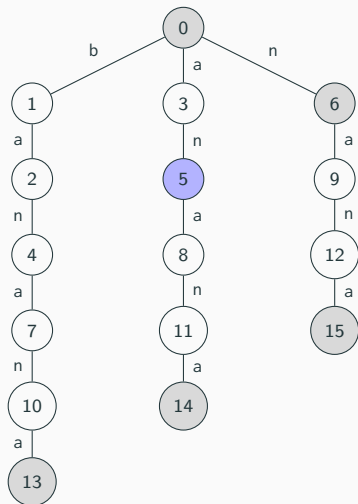


Visualização da construção *online* da *trie*

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	6	0
2	5	6
3	12	5
4	15	12
5	14	15
6	13	14

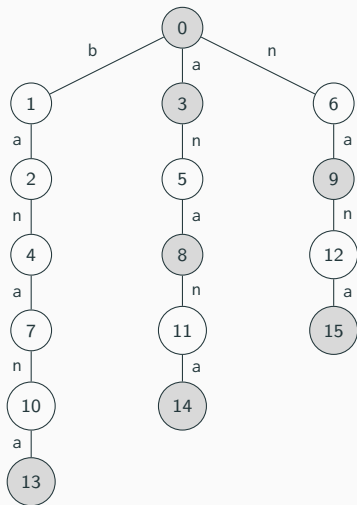


Visualização da construção *online* da *trie*

$k = 6$

$c = 'a'$

i	v_i	$suf[v_i]$
0	0	-1
1	3	0
2	9	3
3	8	9
4	15	8
5	14	15
6	13	14



Implementação da construção *online* da trie

```
9 Trie build_online(const string& s)
10 {
11     int next = 0, deepest = 0; // deepest = v_(k-1)
12     string S = s + '#';        // adiciona o terminador
13     vector<int> suf { -1 };     // suf[root] = NULL
14     Trie trie(1);              // Instancia o nó raiz vazio
15
16     for (size_t i = 0; i < S.size(); ++i)
17     {
18         // Calculo de Tk, com k = i + 1
19         char c = S[i];
20         int u = deepest;
21
22         while (u >= 0)
23         {
24             // Procura por c no nó u
25             auto it = trie[u].find(c);
26
```

Implementação da construção *online* da *trie*

```
27      // Caso #1: link não encontrado
28      if (it == trie[u].end())
29      {
30          // Adiciona um novo nó, com aresta c
31          trie.push_back({ });
32          trie[u][c] = ++next;
33
34          // valor sentinela: será corrigido na próxima iteração
35          suf.push_back(0);
36
37          if (u != deepest)
38              suf[next - 1] = next;    // correção atrasada
39          else
40              deepest = next;          // v_k é o nó recém-criado
41      } else
42      {
43          // Caso #2: link encontrado: suf[v_t] aponta para ele
44          suf[next] = it->second;
45          break;
46      }
```

Implementação da construção *online* da *trie*

```
48         u = suf[u];    // v_(r-1) = suf[v_r]
49     }
50 }
51
52 return trie;
53 }
```

Considerações sobre a construção *online* da *trie*

- Observe que, na implementação proposta, os valores v_k são usados implicitamente
- Para uma string s de tamanho n , a construção *online* da *trie* tem complexidade $O(|T_n|)$
- Embora ainda não seja a complexidade desejada ($O(n)$), esta estratégia será pode ser utilizada, com alguns ajustes, para atingir tal complexidade
- Para reduzir o tamanho em memória da *trie* uma estratégia possível é compactar as cadeias, onde uma cadeia é o maior caminho possível composto por nós não-essenciais com grau de saída um
- Esta compactação resulta em uma nova estrutura, denominada *suffix tree*

1. CP Algorithms. [Suffix Tree](#). [Ukkonen's Algorithm](#), acesso em 02/10/2019.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **ROY, TUSHAR**. [Trie](#), acesso em 02/10/2019.
4. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
5. Wikipédia. [Trie](#), acesso em 02/10/2019.