Departamento de Computação

Programação Competitiva unesp

Strings: KMP e Palíndromos

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

Strings em Programação Competitiva

- Existem diversos problemas clássicos associados a Strings. Nesta aula trataremos sobre dois problemas específicos:
  - Busca em Strings / String *matching*
  - Substrings palindrômicas

Busca em strings

- O problema *substring search*/*pattern search*/*string matching* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

```
S = “Que a Força esteja com você”
P = “Força”
```

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “Que a Força esteja com você”

  P = “Força”

  Ocorrências: 6 (posição)

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “aabaacaadaabaaba”

  P = “aaba”

  Ocorrências: 0, 9 e 12

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “aabaacaadaabaaba”

  P = “aaba”

  Ocorrências: 0, 9 e 12

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “aabaacaadaabaaba”

  P = “aaba”

  Ocorrências: 0, 9 e 12

Busca em strings

- Algoritmo ingênuo

```
int search(string S, string P) {
    for(int i = 0; i <= S.size() - P.size(); i++) {
        for(int j = 0; j < P.size(); j++)
            if (S[i+j] != P[j])
                break;
        if (j == P.size())
            return i;
    }
    return -1;
}
```

Busca em strings

- Esse algoritmo, no pior caso, tem complexidade $O(m.n)$, fazendo $m.n$ comparações. Porém, em geral, ele não chega a realizar tantas comparações.
- Usar esse algoritmo é bastante razoável para vários casos, principalmente quando as strings não são muito grandes.
- Mas, existem algoritmos de busca de substrings mais eficientes, que podem ser necessários em algumas situações, como exemplo temos o KMP.

Alguns conceitos

- **Prefixo** de uma string S é a string obtida após a remoção de 0 ou mais caracteres do fim de S.
  - “a”, “adc”, “adcbaa” são prefixos de “adcbaa”
- **Sufixo** de uma string S é a string obtida após a remoção de 0 ou mais carateres do início de S.
  - “a”, “baa”, “adcbaa” são prefixos de “adcbaa”
- **Prefixo/sufixo próprio** de S é um prefixo/sufixo de S que é diferente de S.
- **Substring** de uma string S é uma string obtida após a remoção de 0 ou mais caracteres no início ou no fim de S.
  - “a”, “cba”, “adc”, “dcba”, “adcbaa” são prefixos de “adcbaa”

KMP

- Knuth Morrit Pratt
- Complexidade: $O(n + m)$ no pior caso

- No algoritmo ingênuo, sempre que detectamos caracteres diferentes, avançávamos um caracter na string principal (i++) e testamos toda a substring, desde o começo (começando sempre com j = 0).
- O KMP, porém, aproveita as comparações que foram feitas antes de encontrar dois caracteres diferentes, evitando comparar novamente caracteres que já sabemos que são compatíveis.

KMP

- A principal ideia deste algoritmo é pré-processar o padrão P, de modo a obter um vetor de inteiros lps, que conta o número de caracteres que podem ser “ignorados” em uma nova comparação.
- O nome lps refere-se à “*longest proper prefix and suffix*”, ou seja, o maior prefixo próprio (não pode ser a própria palavra) que também é sufixo.
    - Conhecido também como função de prefixo.

KMP

P = “ABABAC”
lps = {}

P = “ABABAC”
lps = {0}

P = “ABABAC”
lps = {0, 0}

P = “ABABAC”
lps = {0, 0, 1}

P = “ABABAC”
lps = {0, 0, 1, 2}

P = “ABABAC”
lps = {0, 0, 1, 2, 3}

P = “ABABAC”
lps = {0, 0, 1, 2, 3, 0}

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P =   ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

```
S = ABABABCABABABCABABABC
P = ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

```
S = ABABABCABABABCABABABC
P = ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

E agora?
mantemos o valor de i (ponteiro para posição de S)
j = lps[j - 1] = 3

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

```
S = ABABABCABABABCABABABC
P =   ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

KMP

```
int a[MAX], n, m;
char S[MAX], P[MAX];

void calculatePrefix(){
    int i = 0, j = -1;
    a[0] = -1;
    while(i < m){
        while(j >= 0 && P[i] != P[j])
            j = a[j];
        i++; j++;
        a[i] = j;
    }
}
```

KMP

```
vector<int> KMP2(){     //retorna todas as ocorrências da substring
    vector<int> resp;
    int i = 0, j = 0;
    calculatePrefix();
    while(i < n){
        while(j >= 0 && S[i] != P[j])
            j = a[j];
        i++; j++;
        if (j == m){
            resp.push_back(i - m);
            j = a[j];
        }
    }
    return resp;
}
```

KMP

- Sugestão para entender mais sobre o KMP e suas aplicações:
  - Algoritmo de KMP | Vídeo do Bruno Monteiro

Palíndromos

- Palíndromo é uma sequência de caracteres que ao ser invertida mantém-se idêntica.

A
EVE
RADAR
REVIVER
ROTATOR
LEPERS REPEL
MADAM I'M ADAM
STEP NOT ON PETS
DO GEESE SEE GOD
PULL UP IF I PULL UP
NO LEMONS, NO MELON
DENNIS AND EDNA SINNED
ABLE WAS I ERE I SAW ELBA
A MAN, A PLAN, A CANAL, PANAMA
A SANTA LIVED AS A DEVIL AT NASA
SUMS ARE NOT SET AS A TEST ON ERASMUS
ON A CLOVER, IF ALIVE, ERUPTS A VAST, PURE EVIL; A FIRE VOLCANO

Palíndromos

- Determinar se uma string é um palíndromo é um problema relativamente simples.
- Basta comparar as extremidades dos palíndromos, convergindo em direção ao centro.
- Complexidade $O(n)$

Substrings palindrômicas

- Porém, outro problema recorrente é o de procurar substrings em uma string S que são palíndromos.
- Problemas comuns:
  - Encontrar a maior substring palindrômica
  - Determinar quantas substrings são palíndromos

Substrings palindrômicas

- Solução ingênua
  - Determinar todas as possíveis substrings de S: $O(n^2)$
  - Verificar se cada substring é um palíndromo: $O(n)$
  - Complexidade total: $O(n^3)$

Algoritmo de Manacher

- É claro que, no pior caso, podemos ter $O(n^2)$ substrings palindrômicas, sugerindo que não há algoritmo linear para esse problema.
- Porém, o Algoritmo de Manacher consegue resolver esse problema em $O(n)$.

Algoritmo de Manacher

- A ideia básica é manter os palíndromos em uma forma mais comprimida: para cada posição $i = 0, 1, \dots, n-1$ encontrados os valores:
  - $d_1[i]$: número de palíndromos de tamanho ímpar com centro em i
  - $d_2[i]$: número de palíndromos de tamanho par com “centro” em i

$$a\ \overbrace{b\ a\ \underbrace{b}_{s_3}\ a\ b}^{d_1[3]=3}c$$

$$c\ \overbrace{b\ a\ \underbrace{a}_{s_3}\ b}^{d_2[3]=2}d$$

Algoritmo de Manacher

- Algoritmo trivial $O(n^2)$:

```
vector<int> d1(n),  d2(n);
for (int i = 0; i < n; i++) {
    d1[i] = 1;
    while (0 <= i - d1[i] && i + d1[i] < n && s[i - d1[i]] == s[i + d1[i]]) {
        d1[i]++;
    }

    d2[i] = 0;
    while (0 <= i - d2[i] - 1 && i + d2[i] < n && s[i - d2[i] - 1] == s[i + d2[i]]) {
        d2[i]++;
    }
}
```

Algoritmo de Manacher

- Agora, vamos buscar melhorias para essa solução trivial. Para isso, vamos nos concentrar apenas nos palíndromos de comprimento ímpar.
- Em primeiro lugar, manteremos as extremidades (l, r) da substring palindrômica encontrada mais à direita (com máximo r).
    - Inicialmente $l = 0$ e $r = -1$

Algoritmo de Manacher

- Para calcular $d_1[i]$, considerando os valores anteriores de $d_1[]$, faremos:
  - Se i estiver fora do sub-palíndromo atual, ou seja, $i > r$, iniciaremos o algoritmo trivial.
    - Incrementar $d_1[i]$ consecutivamente até encontrar a primeira divergência ou atingir os limites de S.
    - Atualizar (l, r).

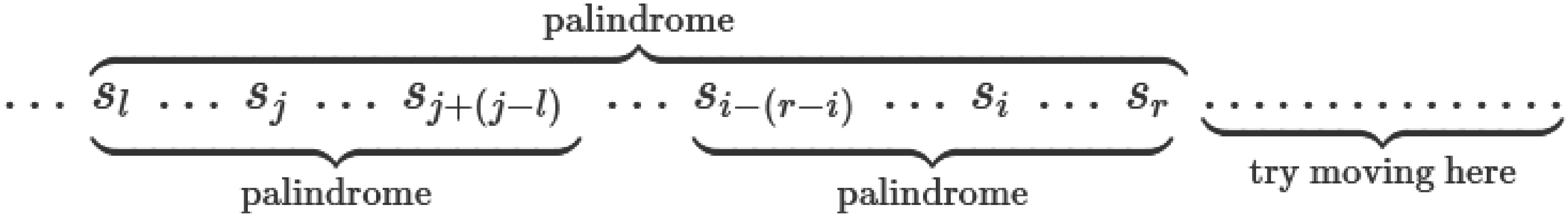
Algoritmo de Manacher

- Para calcular $d_1[i]$, considerando os valores anteriores de $d_1[]$, faremos:
  - Agora, se $i \leq r$, tentaremos extrair informações dos valores já calculados de $d_1[]$.
    - Inverteremos a posição i dentro do sub-palíndromo (l, r), ou seja, obteremos a posição $j = l + (r - i)$, e veremos o valor de $d_1[j]$. Como j é a posição simétrica a i, **quase sempre** podemos definir $d_1[i] = d_1[j]$

$$\dots \overbrace{s_l \dots \underbrace{s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1}}_{\text{palindrome}} \dots \underbrace{s_{i-d_1[j]+1} \dots s_i \dots s_{i+d_1[j]-1}}_{\text{palindrome}} \dots s_r}^{\text{palindrome}} \dots$$

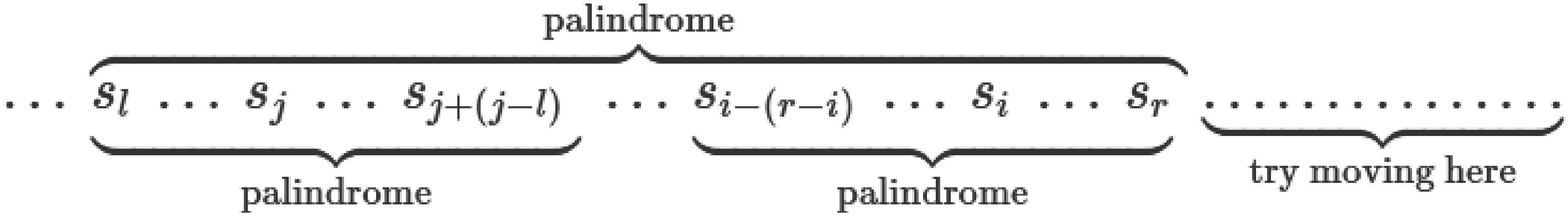
Algoritmo de Manacher

- Para calcular $d_1[i]$, considerando os valores anteriores de $d_1[]$, faremos:
  - Agora, se $i \leq r$, tentaremos extrair informações dos valores já calculados de $d_1[]$.
    - Mas há um caso mais complicado: quando o palíndromo "interno" atinge as bordas do "externo". Como a simetria fora do palíndromo "externo" não é garantida, apenas atribuir $d_1[i] = d_1[j]$ estará incorreto.



Algoritmo de Manacher

- Para calcular $d_1[i]$, considerando os valores anteriores de $d_1[]$, faremos:
  - Agora, se $i \le r$, tentaremos extrair informações dos valores já calculados de $d_1[]$.
    - Para resolver isto vamos "podar" o comprimento do nosso palíndromo, para que ele não ultrapasse os limites de $(l,\ r)$: $d_1[i] = \min(d_1[j],\ r\ - i + 1)$
    - Depois disso, tentaremos aumentar $d_1[i]$ pelo algoritmo trivial.
    - No final, atualizar $(l,\ r)$



Algoritmo de Manacher

- Algoritmo de Manacher – cálculo do $d_1[]$:

```
vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
        k++;
    }
    d1[i] = k--;
    if (i + k > r) {
        l = i - k;
        r = i + k;
    }
}
```

Algoritmo de Manacher

- Algoritmo de Manacher – cálculo do $d_2[]$:

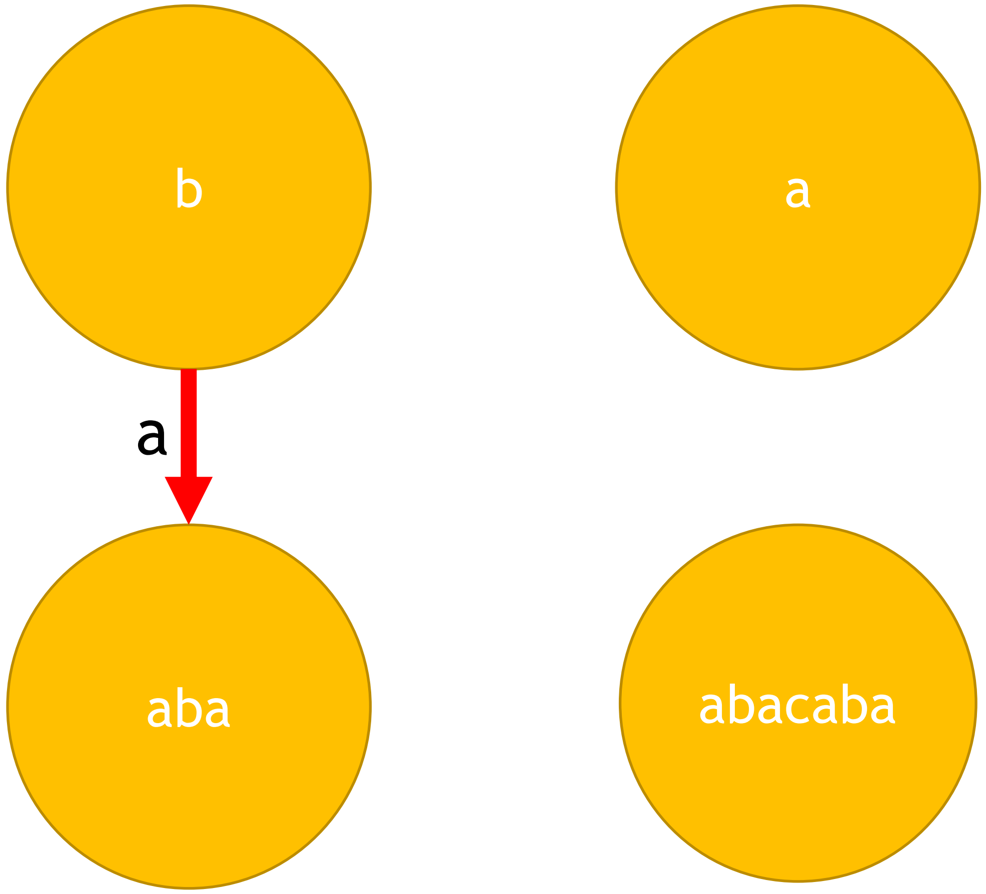
```
vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
        k++;
    }
    d2[i] = k--;
    if (i + k > r) {
        l = i - k - 1;
        r = i + k ;
    }
}
```

Palindromic Tree

- Outro método que nos auxilia a trabalhar com palíndromos é a utilização da estrutura Palindromic Tree.
- Esta solução é mais complicada que o Algoritmo de Manacher, e possui a mesma complexidade, porém é mais flexível.
- A ideia consiste em criar uma "árvore" em que os nós representam as substrings palindrômicas. Essa árvore possui dois tipos de arestas direcionadas:
  - Arestas anotadas com letras, indicando a adição de uma letra ao palíndromo
  - Arestas de sufixos

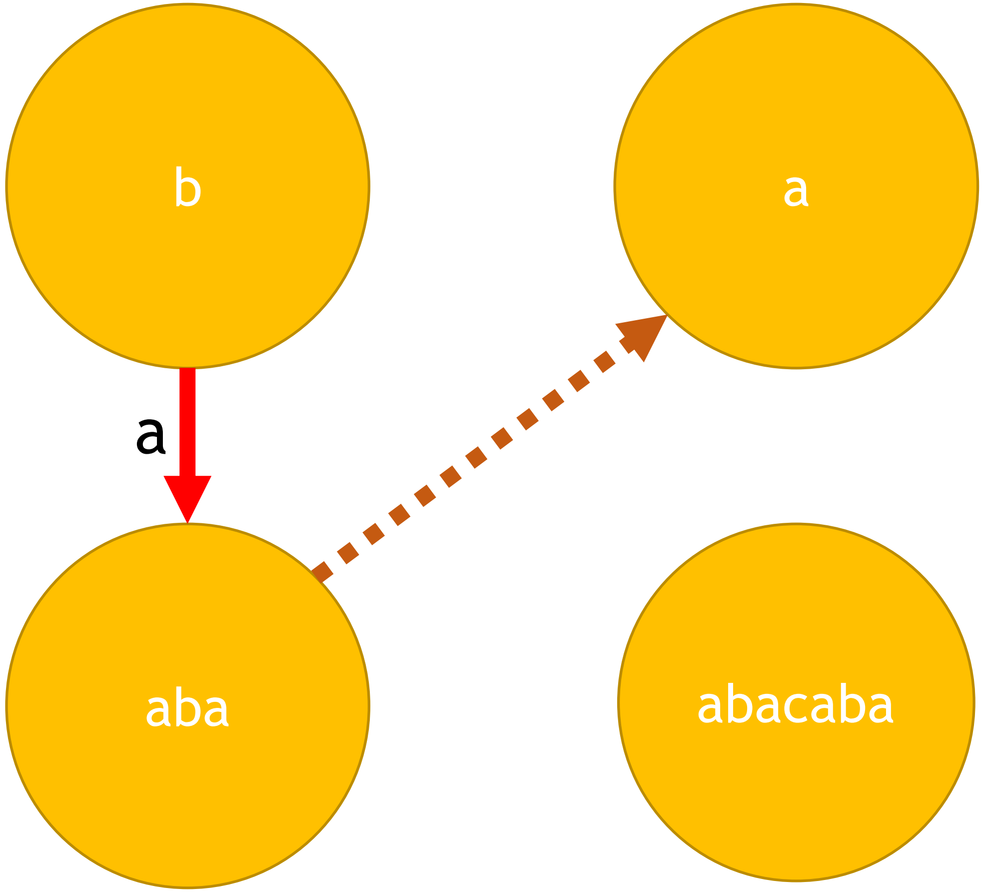
Palindromic Tree

- O primeiro tipo de aresta é uma aresta direcionada do nó u ao nó v, com um caractere associado.
- Essa aresta indica a adição do caractere associado a ela aos polos da substring do nó u, gerando assim a substring do nó v.



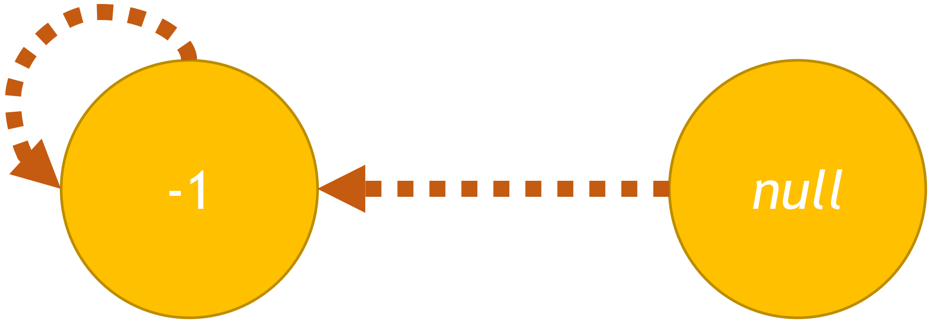
Palindromic Tree

- O segundo tipo de aresta, a aresta de sufixo, é uma aresta direcionada do nó v ao nó w.
- Esta aresta indica que w é o maior sufixo próprio (que também é palíndromo) de v.



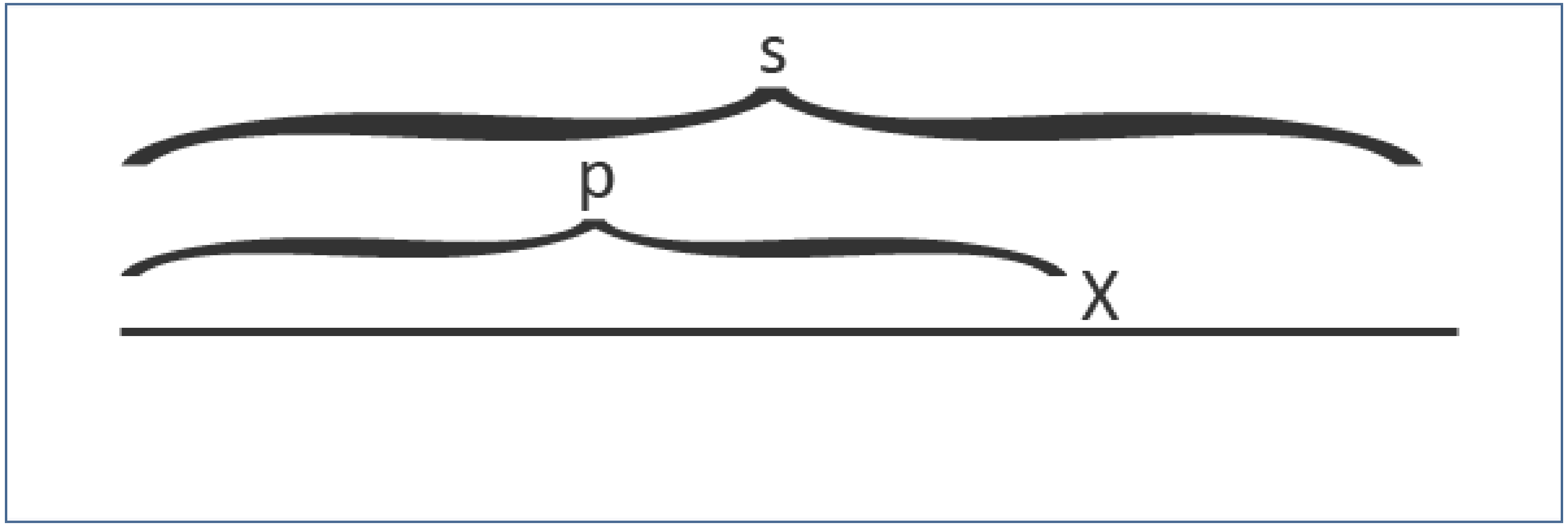
Palindromic Tree

- Esta estrutura se inicia com duas "raízes":
  - A raiz nula: representando a substring vazia ""
  - A raiz imaginária: representando uma substring imaginária de tamanho –1
    - Basicamente, esse é um artifício para a criação de palíndromos de tamanho ímpar.



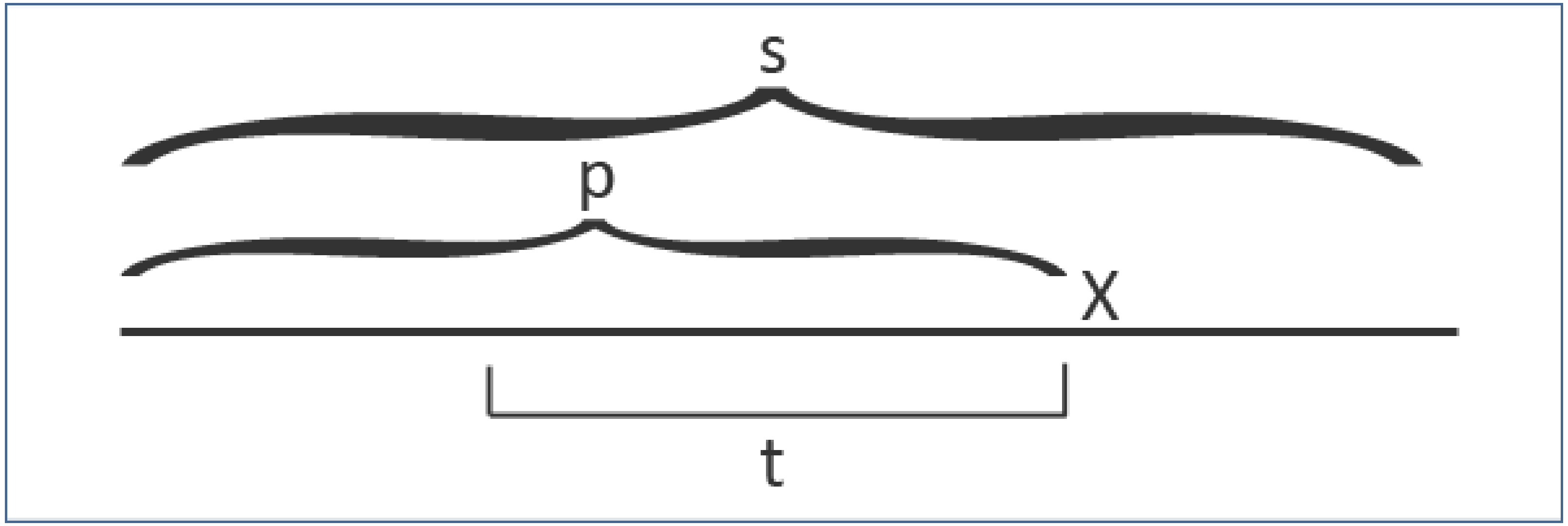
Palindromic Tree: Construção

- Para construir uma palindromic tree para um dada string s iremos processar caractere por caractere desta string.
- Supondo que já processamos um prefixo p de s e estamos processando um certo caractere x. Temos a seguinte situação:



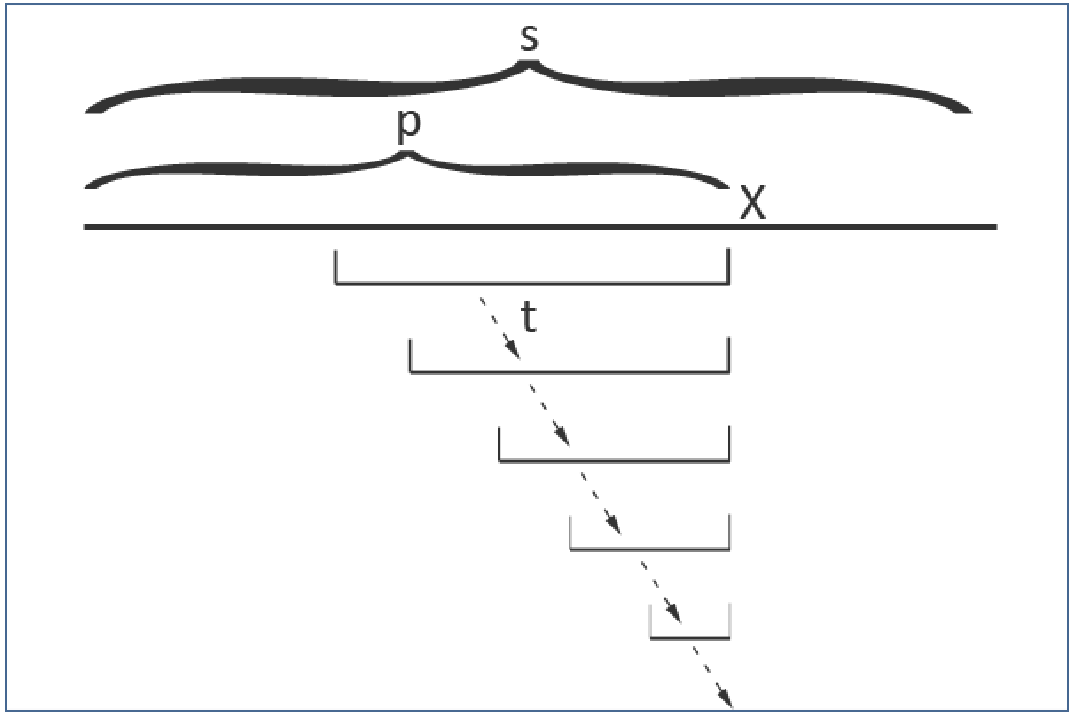
Palindromic Tree: Construção

- Nós mantemos também o maior sufixo palindrômico de p. Vamos chama-lo de t.



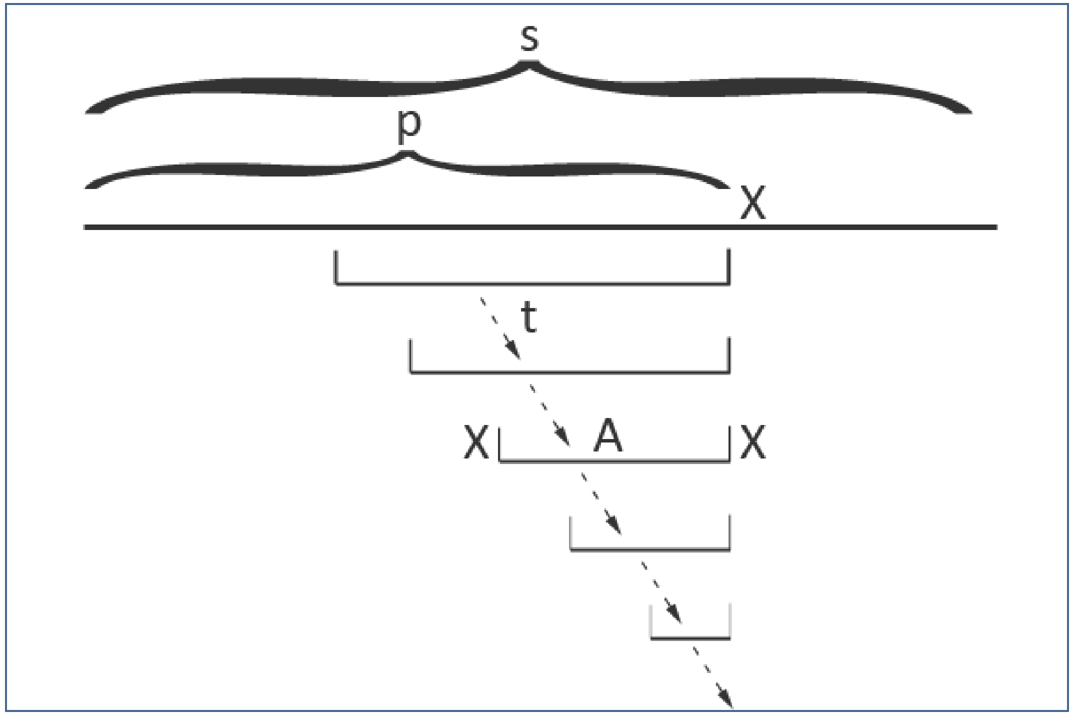
Palindromic Tree: Construção

- Como t já foi processado, então ele corresponde a um nó da palindromic tree, e possui uma aresta de sufixo ligando outro nó existente, e assim por diante.



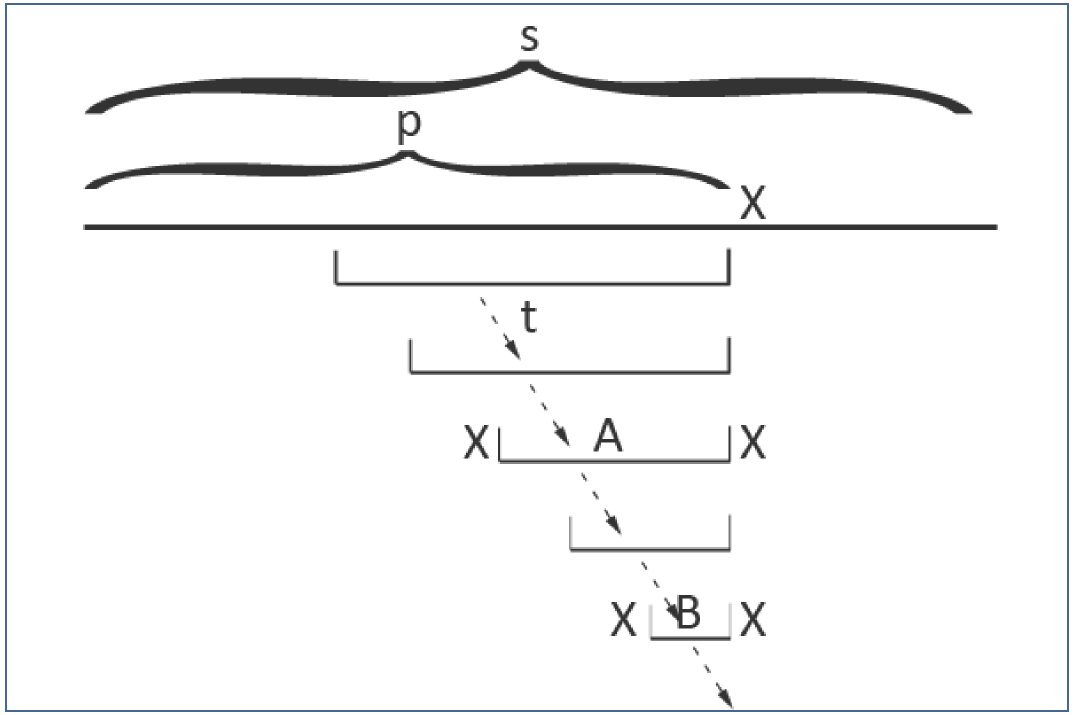
Palindromic Tree: Construção

- Para descobrirmos o maior palíndromo que termina em x (da forma xAx) basta percorrermos o ramo de sufixos em busca do primeiro que a letra anterior é x.



Palindromic Tree: Construção

- Se não existia nó para xAx, então precisamos encontrar seu maior sufixo palindrômico. Para isso, basta continuarmos explorando o ramo de sufixos de p, em busca em um xBx.



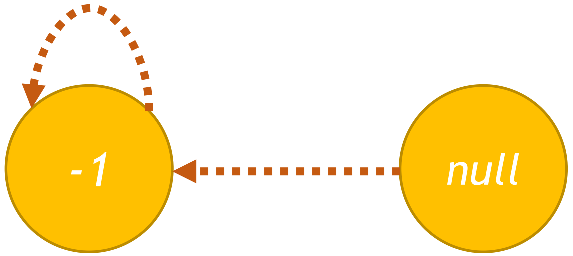
Palindromic Tree: Construção

- A complexidade linear deste problema deriva do fato de que aproveitamos as informações dos palíndromos que já foram processados (representados por nós) e de que uma certa string não possui mais de n palíndromos distintos.

Palindromic Tree: Construção

aabcba

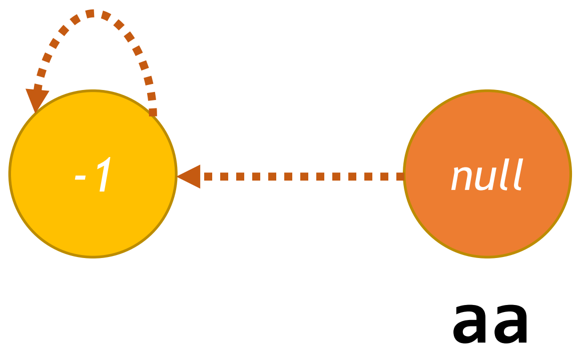
nó		len	num
1	-1	-1	0
2	null	0	0
3			
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aabcba

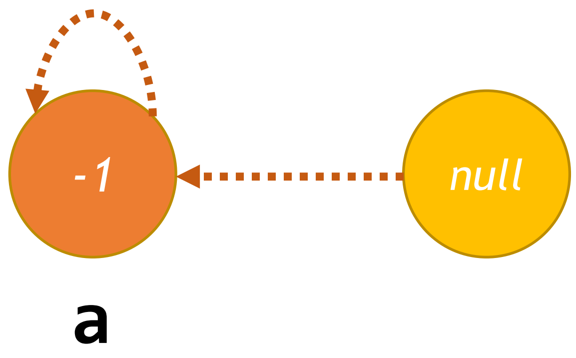
nó		len	num
1	-1	-1	0
2	null	0	0
3			
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aabcba

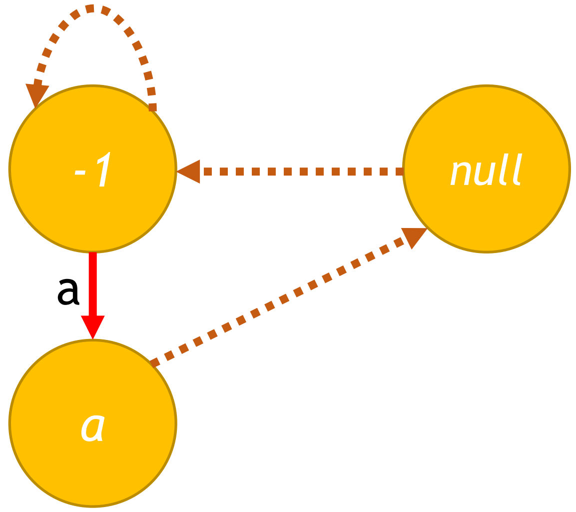
nó		len	num
1	-1	-1	0
2	null	0	0
3			
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aabcba

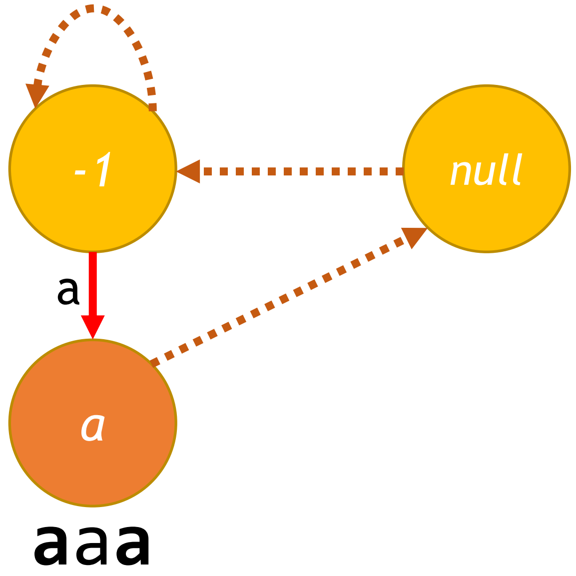
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aabcba

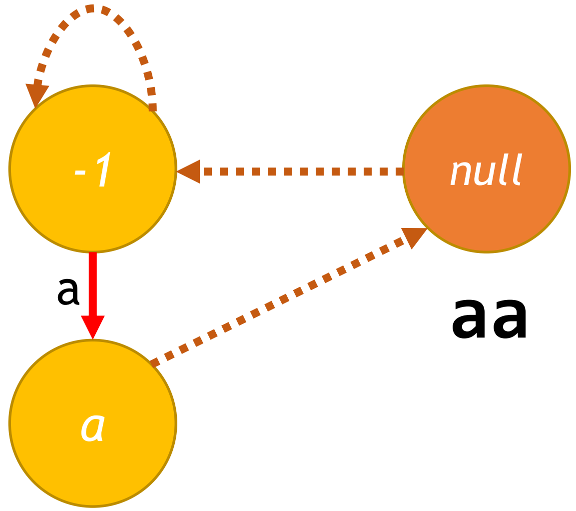
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

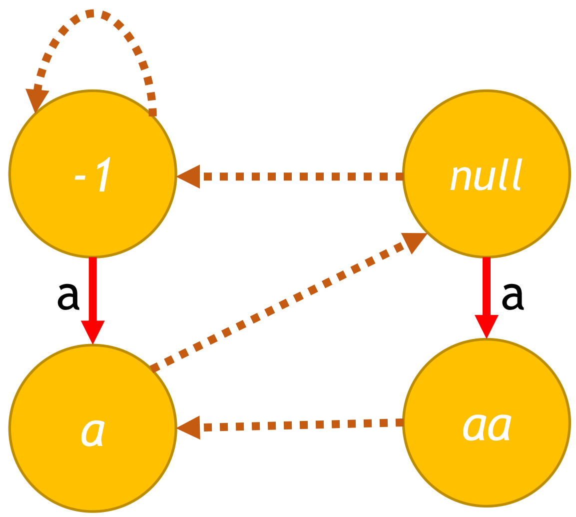
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4			
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

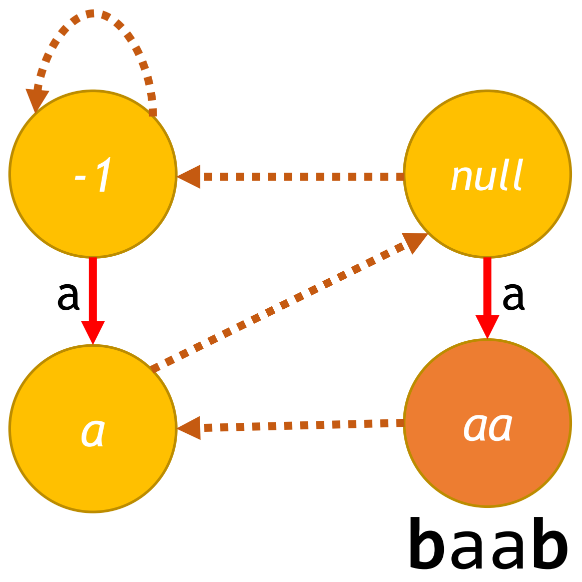
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

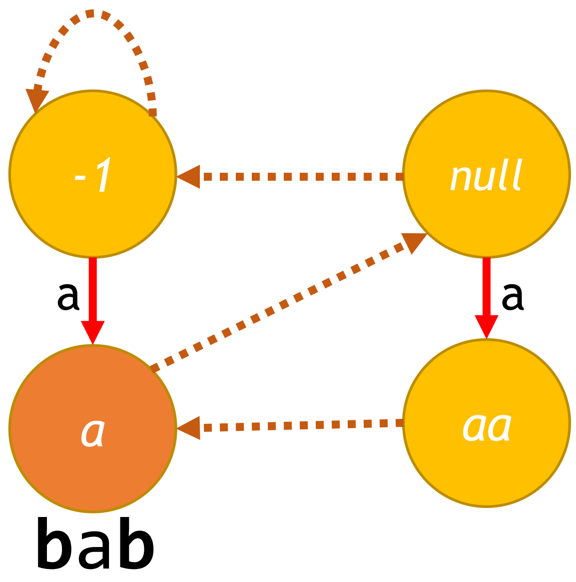
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

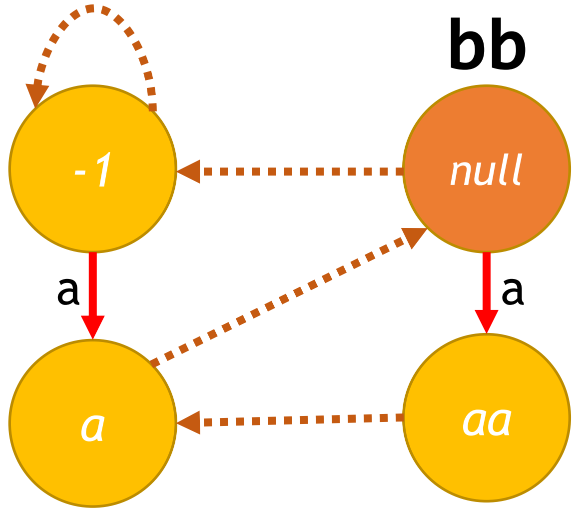
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

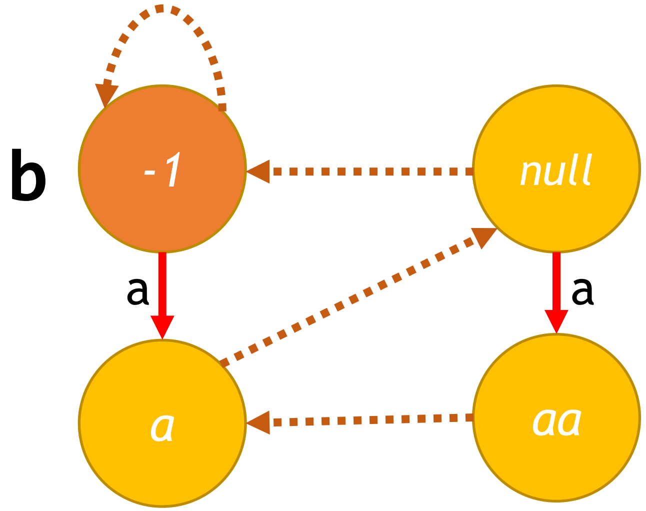
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

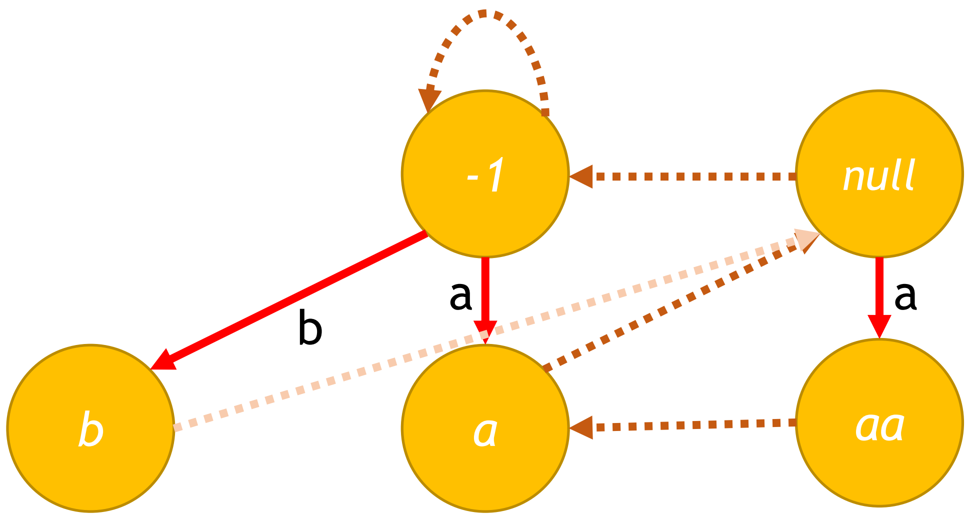
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5			
6			
7			
8			



Palindromic Tree: Construção

aa**b**cba

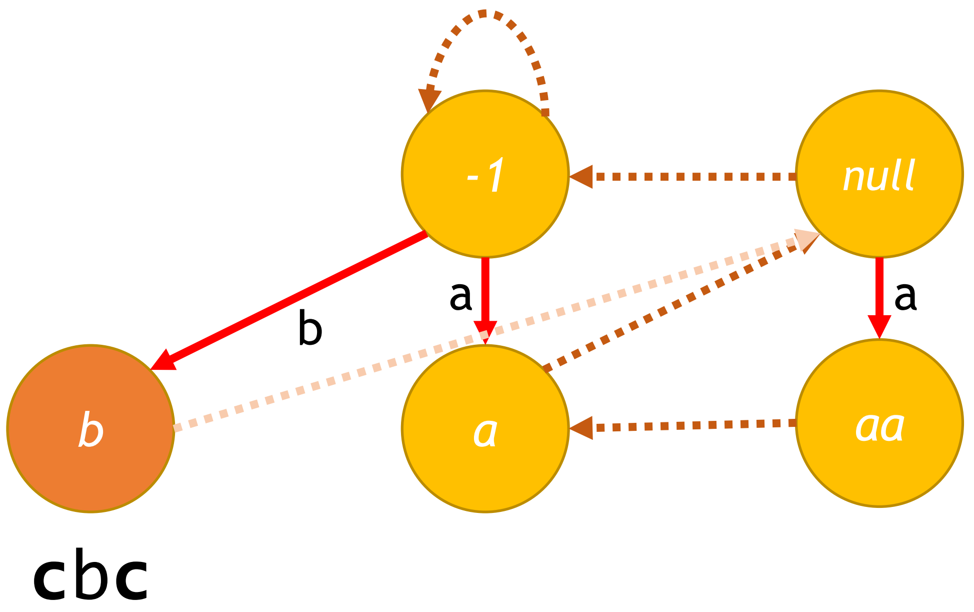
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6			
7			
8			



Palindromic Tree: Construção

aabcba

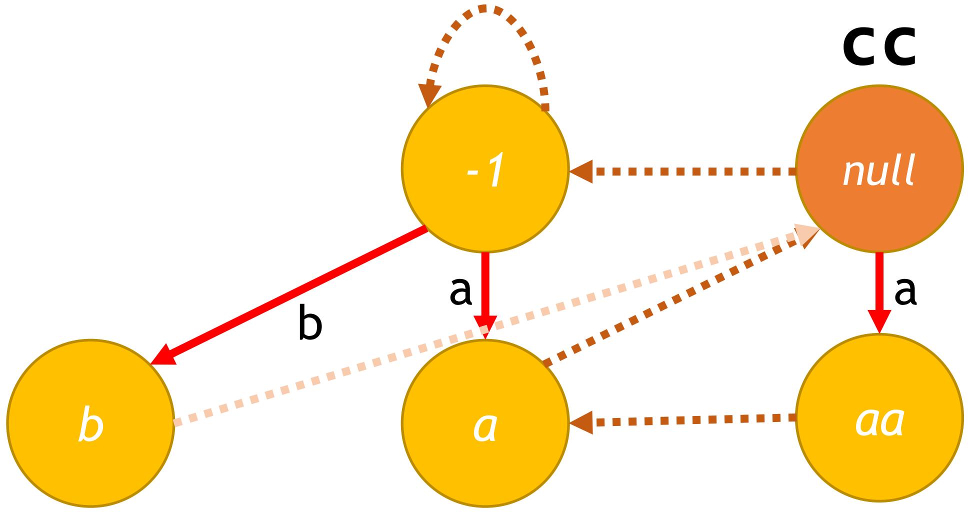
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6			
7			
8			



Palindromic Tree: Construção

aab**c**ba

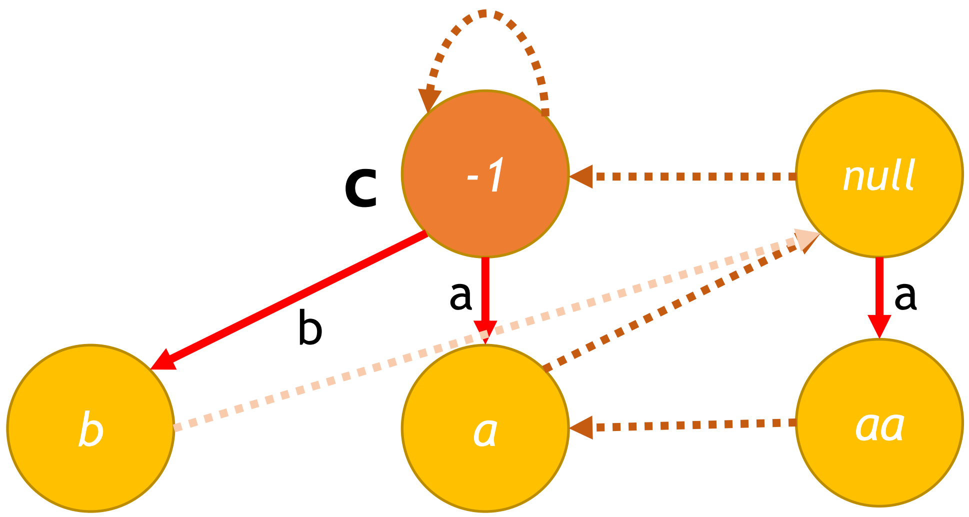
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6			
7			
8			



Palindromic Tree: Construção

aabc**c**ba

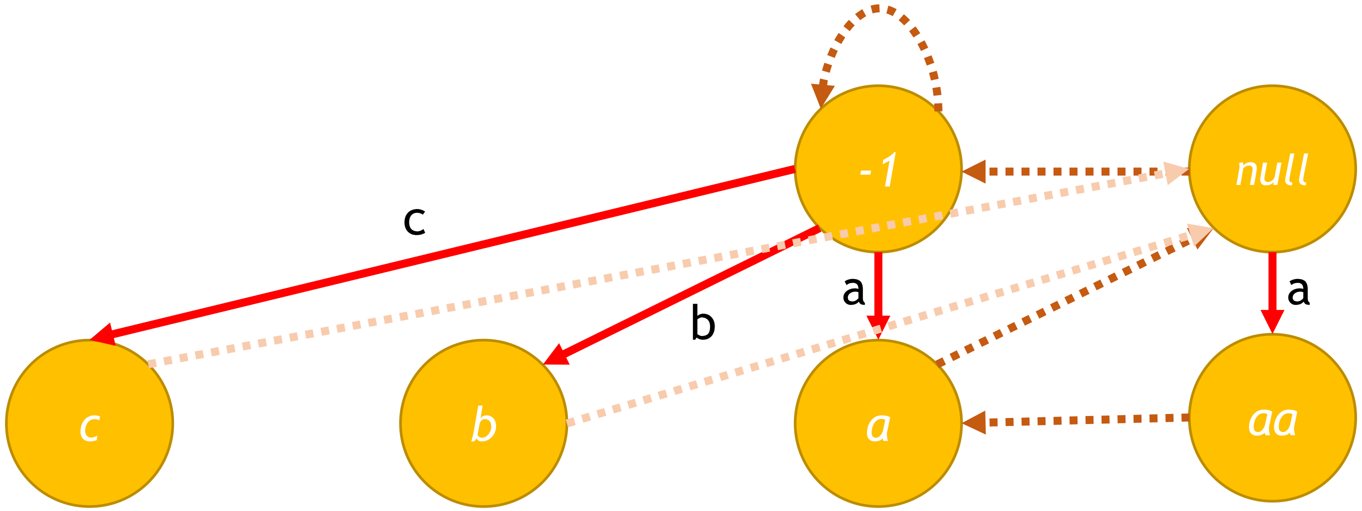
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6			
7			
8			



Palindromic Tree: Construção

aab**c**ba

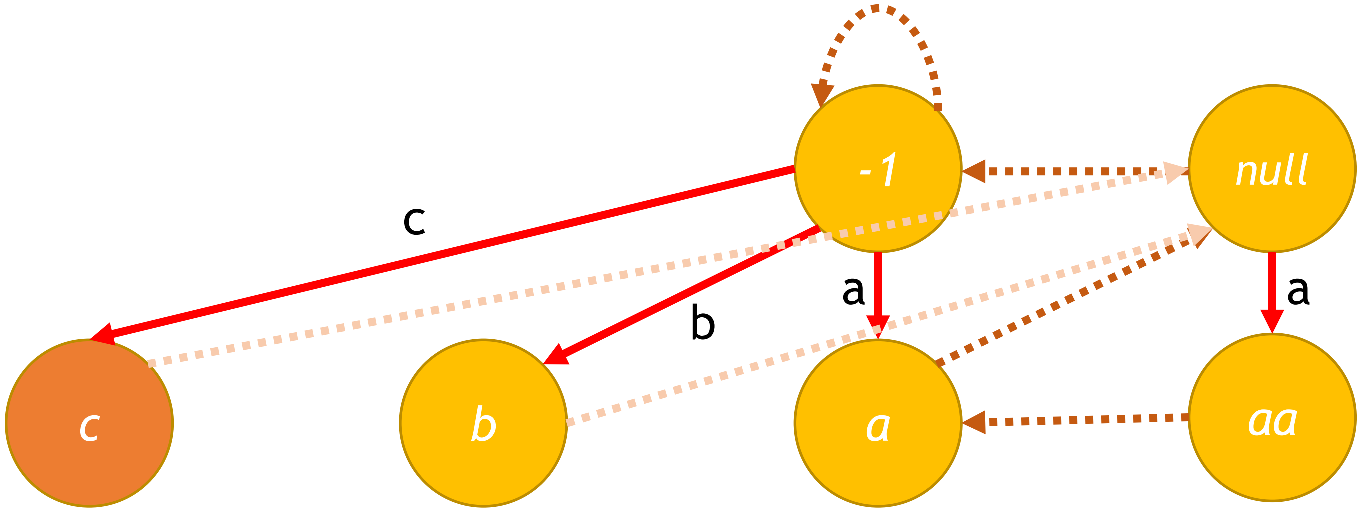
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7			
8			



Palindromic Tree: Construção

aab**cba**

nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7			
8			

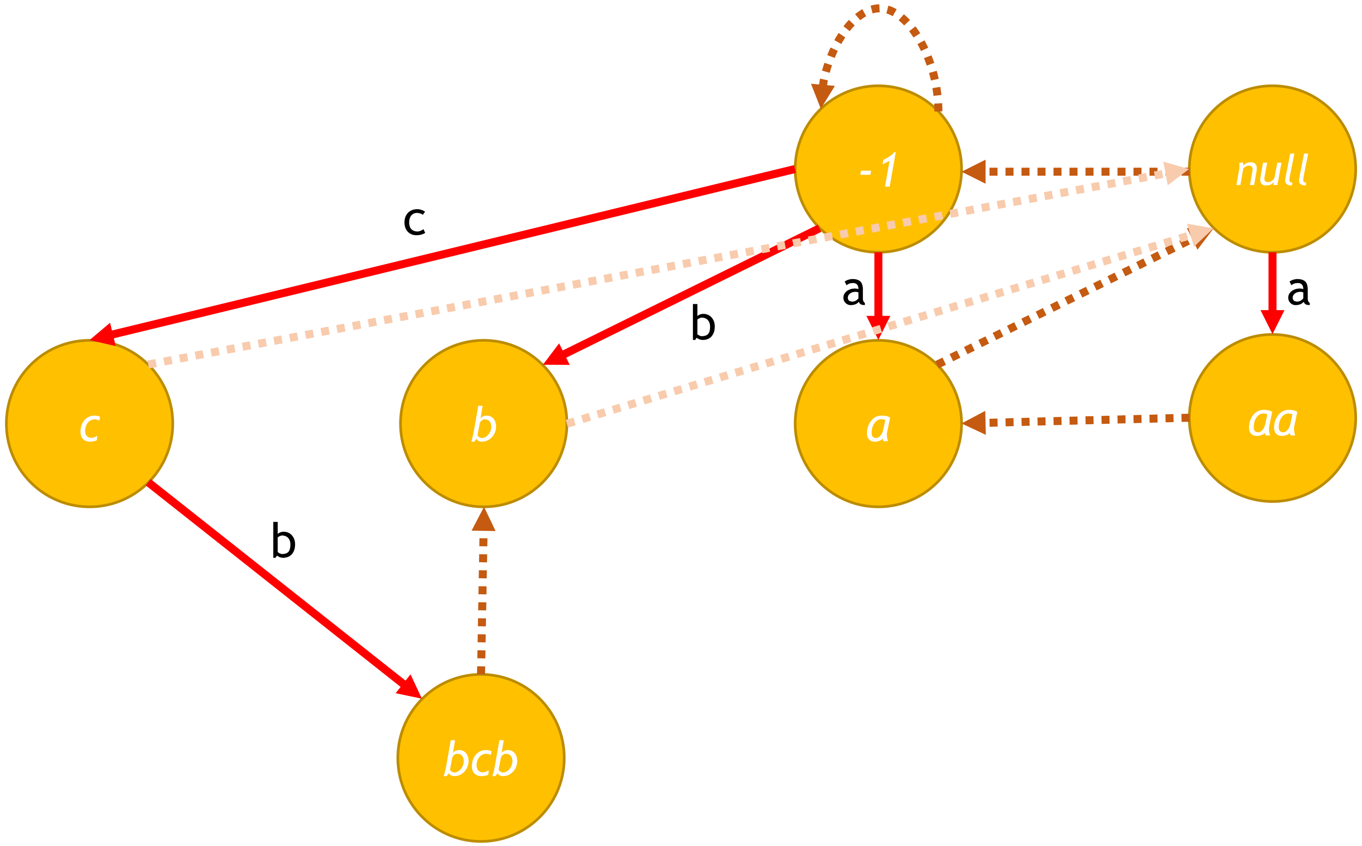


bcb

Palindromic Tree: Construção

aabcba

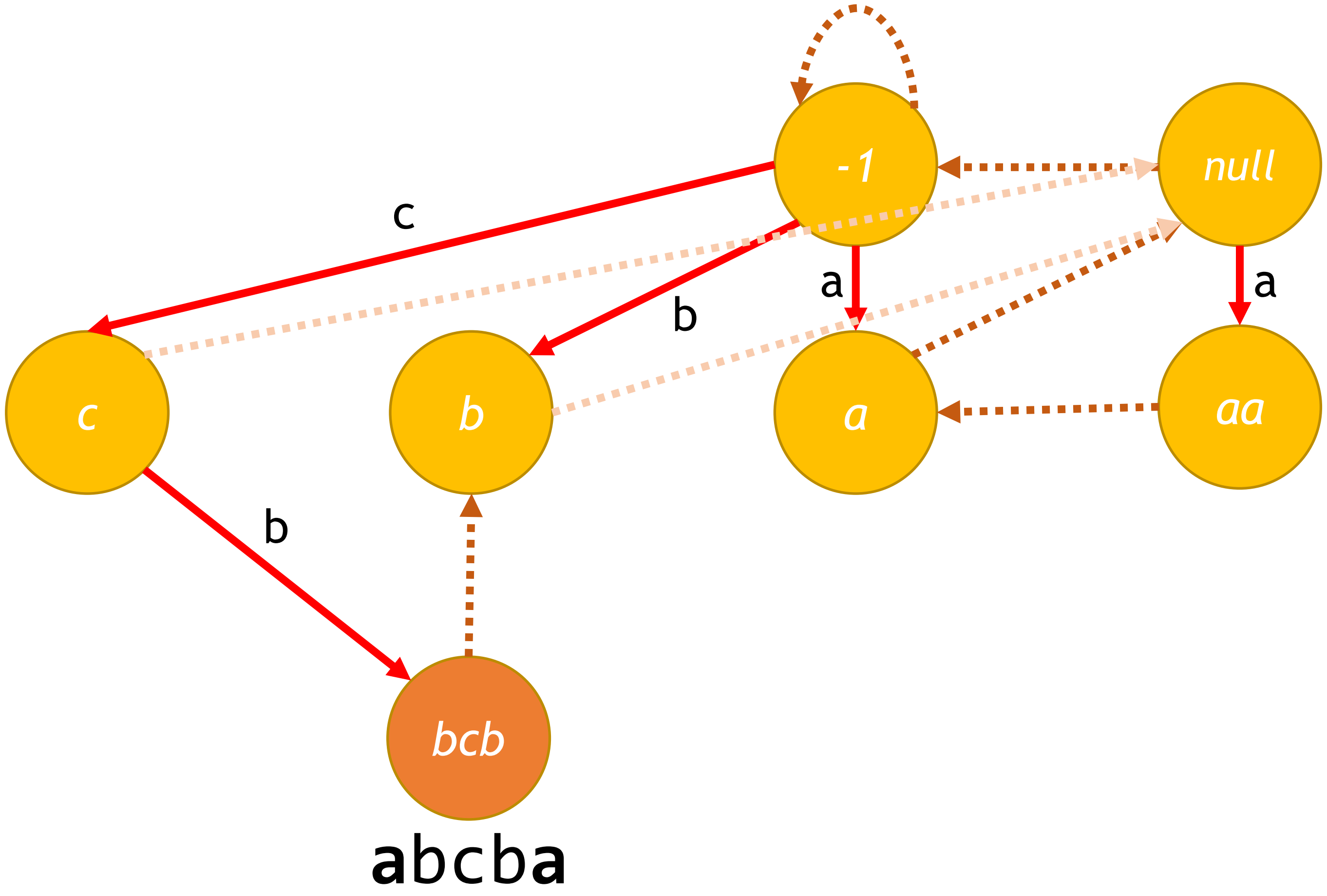
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7	bcb	3	2
8			



Palindromic Tree: Construção

aabcb**a**

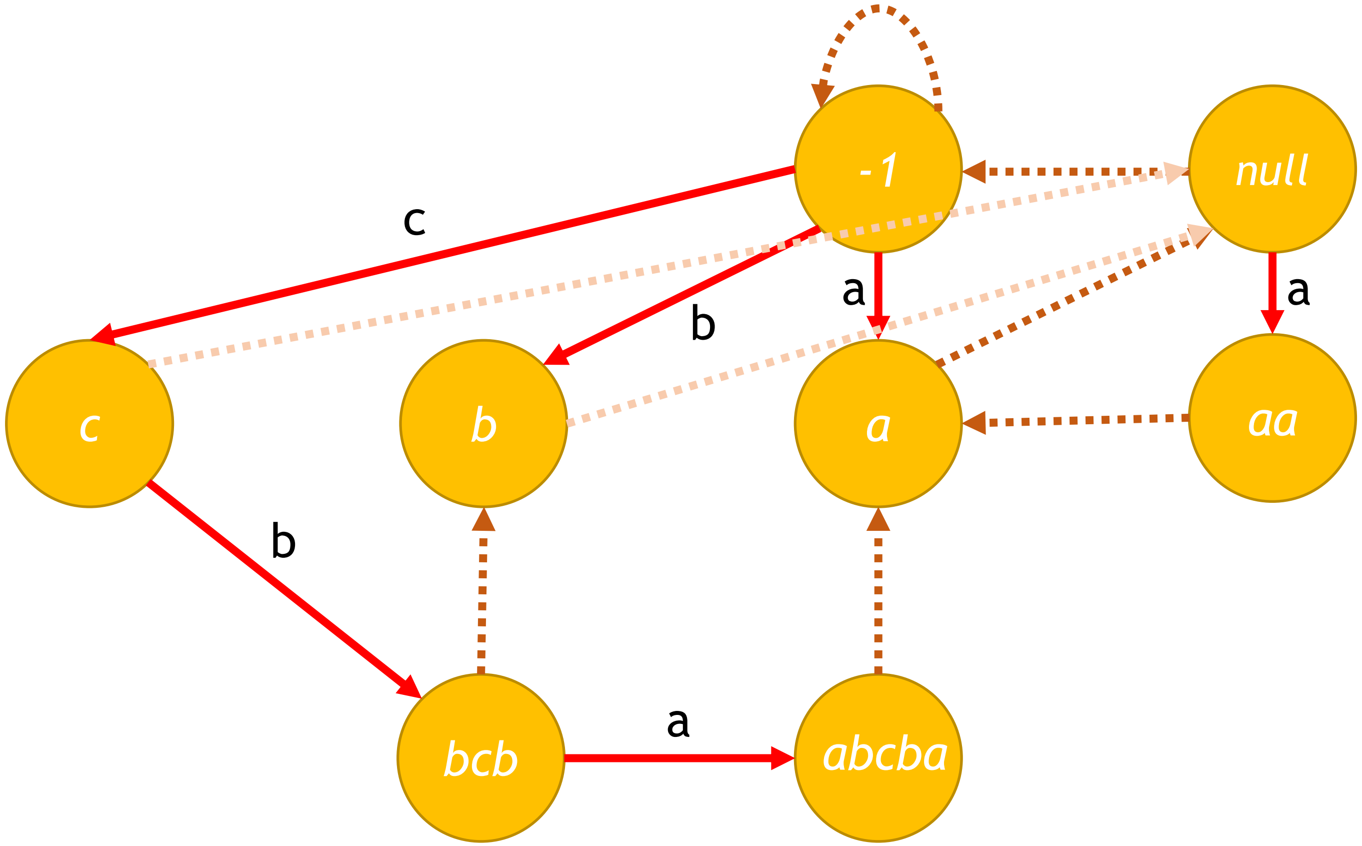
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7	bcb	3	2
8			



Palindromic Tree: Construção

aabcb**a**

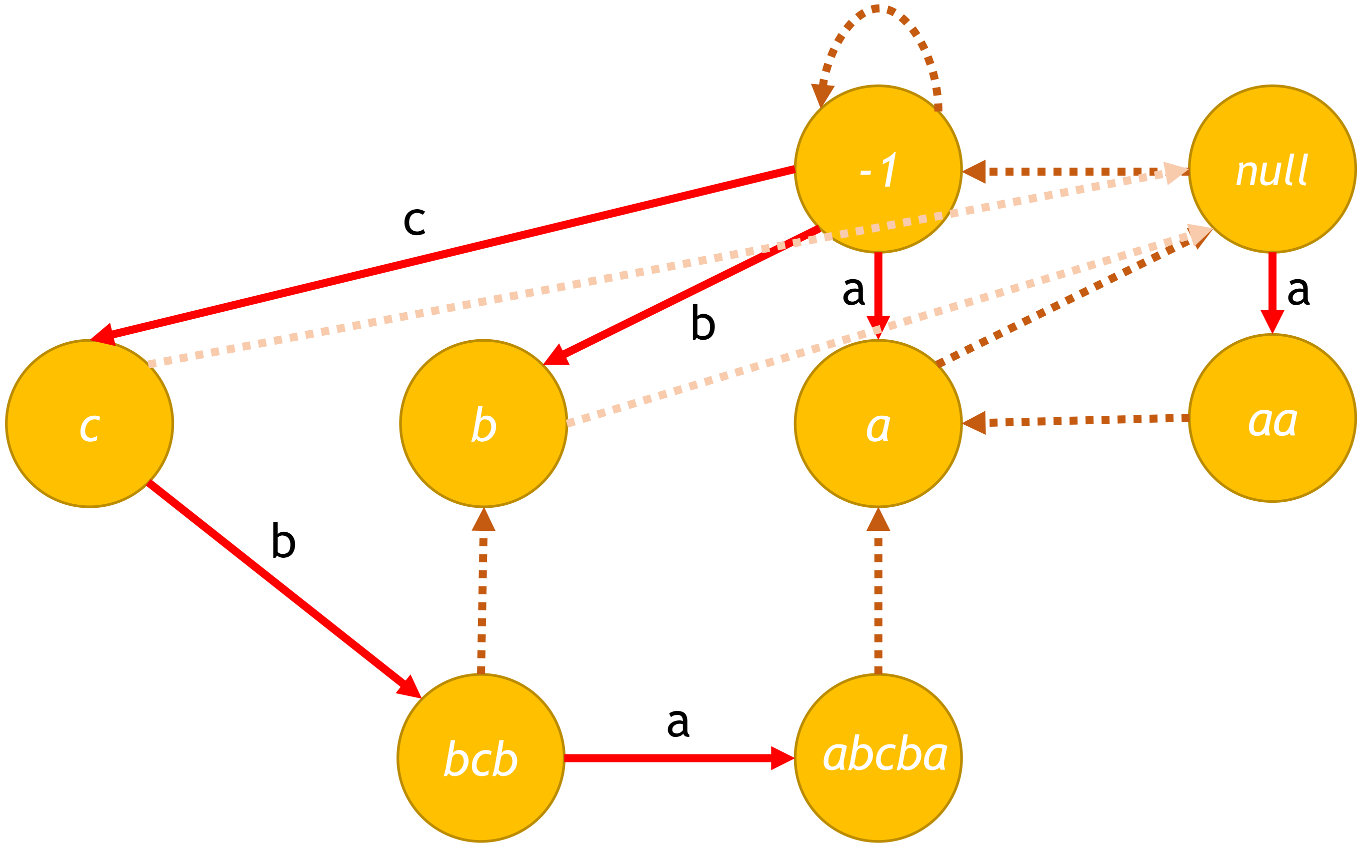
nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7	bcb	3	2
8	abcba	5	2



Palindromic Tree: Construção

aabcba

nó		len	num
1	-1	-1	0
2	null	0	0
3	a	1	1
4	aa	2	2
5	b	1	1
6	c	1	1
7	bcb	3	2
8	abcba	5	2



Palindromic Tree: Considerações finais

- Implementação: https://github.com/ADJA/algos/blob/master/Strings/PalindromeTree.cpp

- Material complementar:
  - Seminário: Substrings Palindrômicas | LPC II 2020
  - Palindromic tree
  - Palindromic tree | GeeksforGeeks

Referências

S. Halim e F. Halim. Competitive Programming 2.

Fábio L. Usberti. Processamento de Cadeias de Caracteres. Summer School 2019

Denis Henrique, Mateus Rijo, Thomas Santos e Vinícius Coutinho. Seminário: Substrings palindrômicas. https://www.youtube.com/watch?v=SWrZzvlJX_0

https://www.youtube.com/watch?v=RXISWaGmYW8

https://cp-algorithms-brasil.com/strings/prefixo.html

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/kmp.html

https://cp-algorithms-brasil.com/strings/manacher.html

http://adilet.org/blog/palindromic-tree/

https://www.geeksforgeeks.org/palindromic-tree-introduction-implementation/