

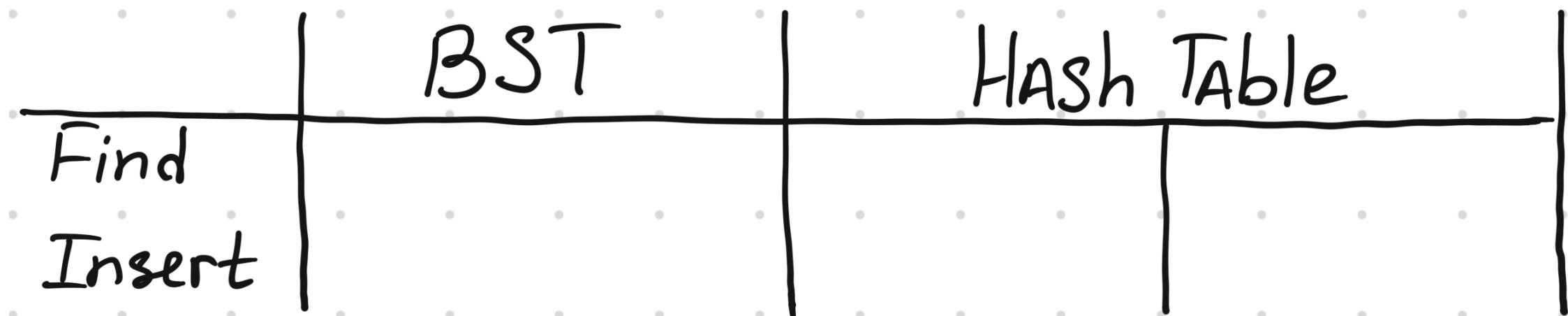
**Бор (префиксное дерево)**

**Trie (Prefix Tree)**

# Quiz



Чему равна сложность поиска и вставки строки  $S$  во множество размера  $n$ ?



# Quiz



Чему равна сложность поиска и вставки строки  $S$  во множество размера  $n$ ?

	BST	Hash Table	
Find	$O( S  \log n)$ w.c.	$O( S n)$ w.c.	$O( S )$ am.avg.
Insert	$O( S  \log n)$ w.c.	$O( S n)$ w.c.	$O( S )$ am.avg.

# Бор

Пусть  $\Sigma$  - конечный алфавит символов.

Бор (префиксное дерево) - структура данных в виде корневого дерева, обеспечивающая эффективный поиск и вставку (удаление) строк.

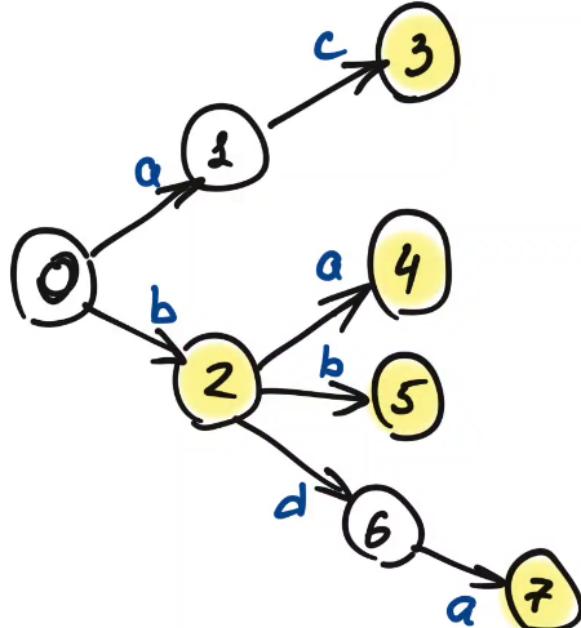
- Каждое ребро (переход - *next*) характеризуется буквой алфавита.
- Узел не может иметь двух ребер, начинающихся на одну букву.
- Узел терминальный, если на пути от корня до него можно прочитать строку хранимого множества.

```
struct Node {  
    dict<char -> node_id> next; // ребра  
    bool is_terminal;  
}
```

# Бор: пример

```
struct Node {  
    dict<char -> node_id> next;  
    bool is_terminal;  
}
```

{ b, ac, ba, bb, bda }



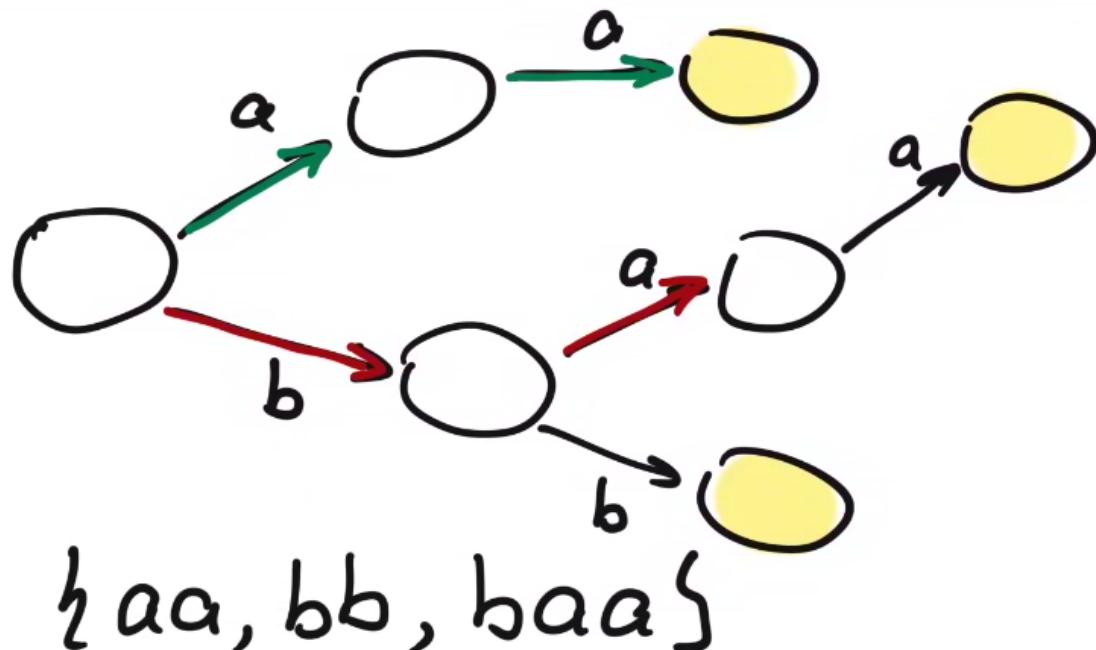
$\text{Node}_0 = \left\{ \begin{array}{l} \text{next: } \{a \rightarrow 1, b \rightarrow 2\} \\ \text{is\_terminal: false} \end{array} \right\}$

$\text{Node}_2 = \left\{ \begin{array}{l} \text{next: } \{a \rightarrow 4, b \rightarrow 5, d \rightarrow 6\} \\ \text{is\_terminal: true} \end{array} \right\}$

$\text{Node}_3 = \left\{ \begin{array}{l} \text{next: } \emptyset \\ \text{is\_terminal: true} \end{array} \right\}$

# Бор: поиск строки

0. Стартуем из корня
1. Читаем очередную букву строки. Если из текущей вершины нет требуемого перехода, то возвращаем *false*. Иначе переходим в следующую вершину.
2. Повторяем 1. пока не прочитаем строку целиком (или не вернем *false*).  
Ответ хранится в поле *is\_terminal* последней посещенной вершины.



Find (aa) → true  
Find (ba) → false  
Find (c) → false

# Бор: поиск строки

```
def Find(str):
    node = nodes[0] # root
    for symbol in str:
        next_id = node.next[symbol]
        if next_id is None:
            return False
        node = nodes[next_id]
    return node.is_terminal
```

Сложность зависит от реализации поля *next*:

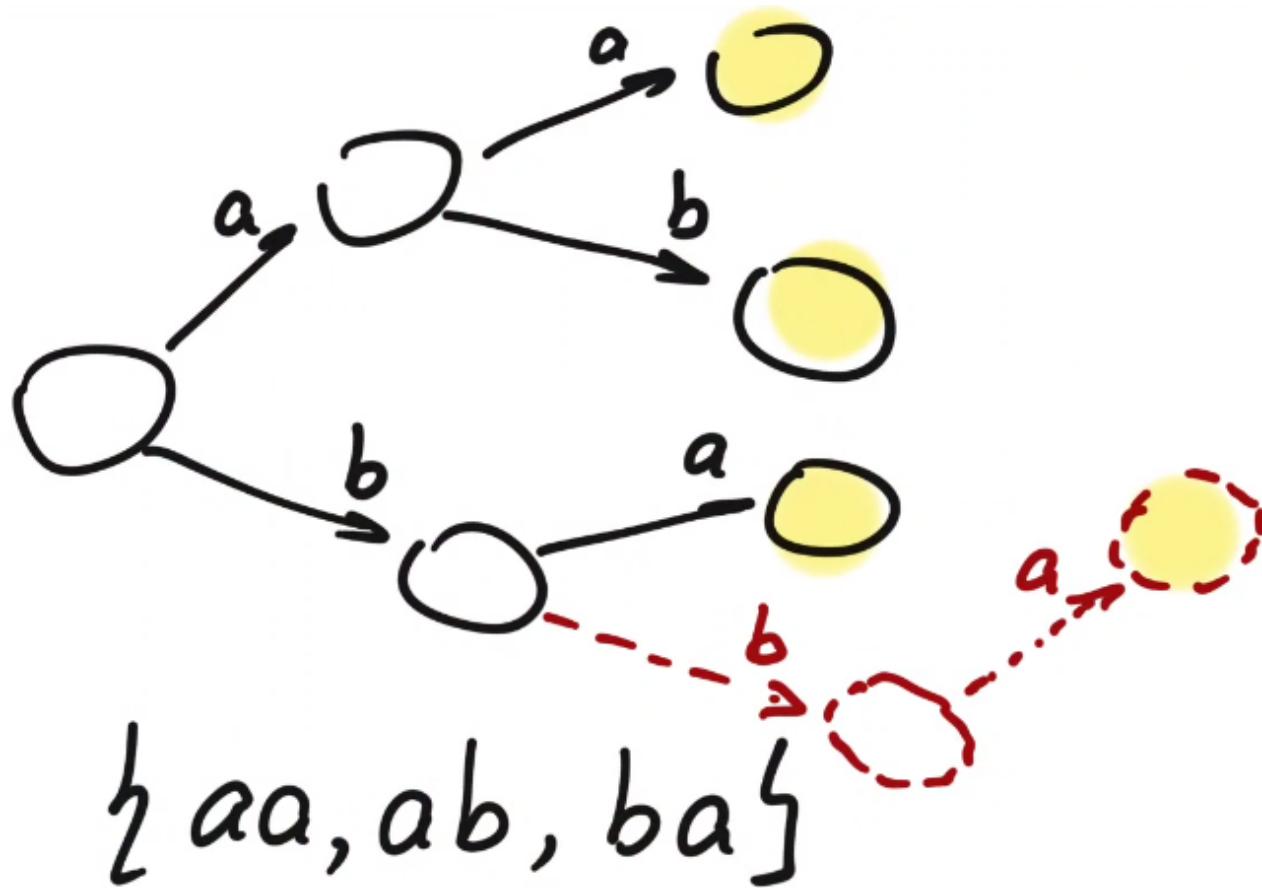
Если массив, то  $O(|S|)$ , но тогда размер каждой вершины  $O(|\Sigma|)$ .

Если хеш-таблица, то  $O(|S|)$  с оговорками.

Если дерево поиска, то  $O(|S| \log |\Sigma|)$ .

# Бор: вставка строки

Повторяем действия поиска, но в случае отсутствия перехода создаем его. В конце выставляем флаг терминальности в последнюю вершину.



Insert(b<sub>1</sub>a)

# Бор: вставка строки

```
def Insert(str):
    node = nodes[0] # root
    for symbol in str:
        next_id = node.next[symbol]
        if next_id is None: # отличие 1
            nodes.push_back(Node{})
            next_id = node.next[symbol] = nodes.size() - 1
        node = nodes[next_id]
        node.is_terminal = True # отличие 2
```

Сложность такая же как у поиска.

# Бор: удаление строки

Упражнение.

## Бор: итог

	BST	Hash Table	Trie
Find	$O( S  \log n)$ w.c.	$O( S n)$ w.c.	$O( S )$ am.avg
Insert	$O( S  \log n)$ w.c.	$O( S n)$ w.c.	$O( S )$ am.avg

# Поиск нескольких подстрок в строке

## Алгоритм Ахо-Корасик

# Задача

Дан фиксированный набор образцов (слов)  $P = \{P_1, P_2, \dots, P_k\}$ .

Поступают запросы-тексты  $S_1, S_2, \dots, S_n, \dots$ .

На запрос  $i$  нужно вернуть вхождения  $P$  в  $S_i$ .

Ясно, что задачу можно решить с помощью КМП за  $O(\sum |P_i| + k \sum |S_i| + |ans|)$ , но  $k \sum |S_i|$  выглядит страшно...

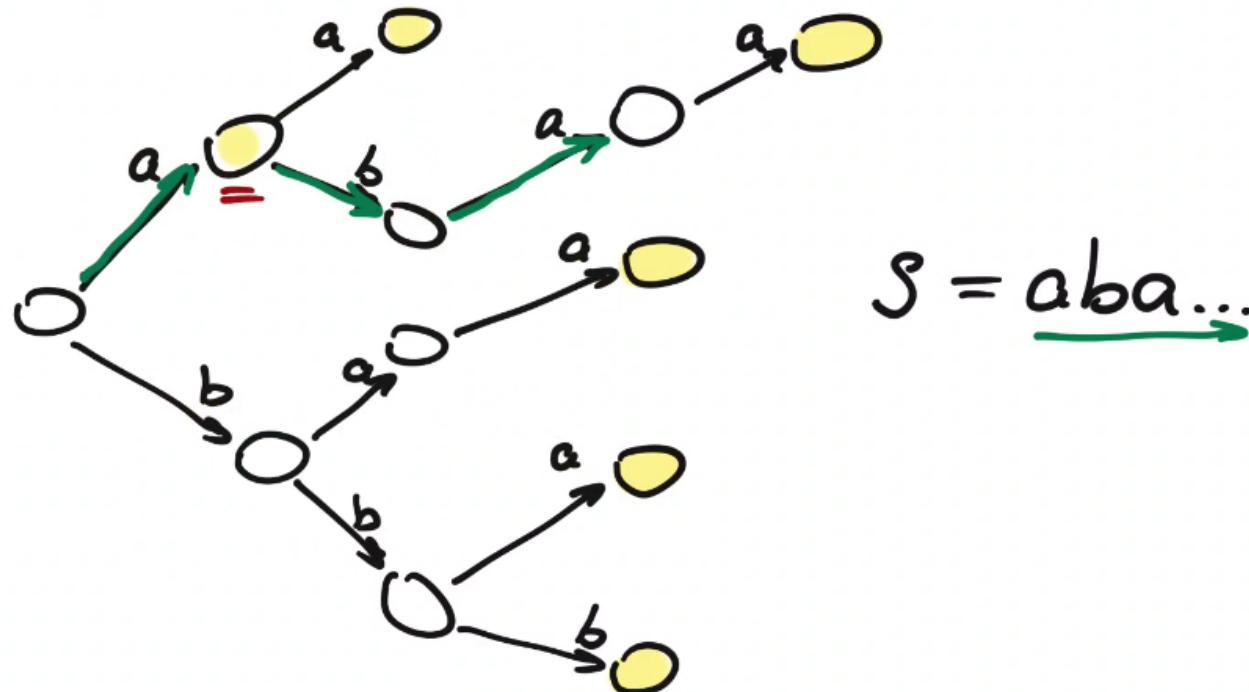


# Ахо-Корасик курильщика (step 1)

Шаг 1: построим бор по множеству  $P$ .

Идея: Будем идти по символам строки  $S$  и продвигаться по бору. Если встретили терминальную вершину, то нашли вхождение  $P$  в  $S$ .

$$P = \{a, aa, baa, bba, bbb, aba\}$$



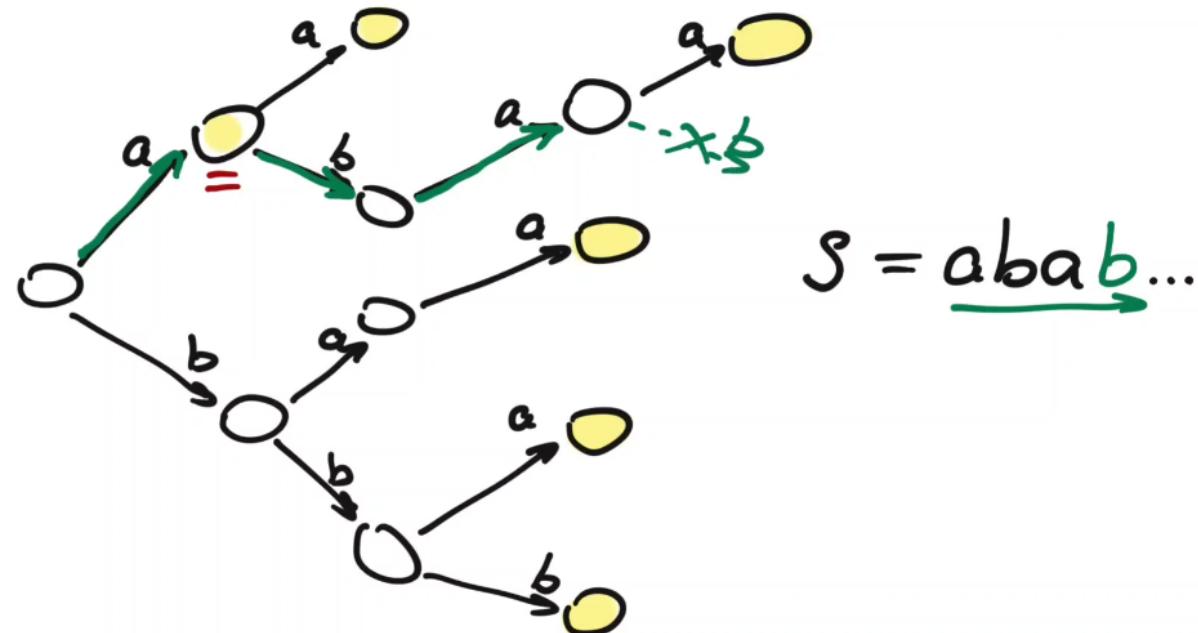
# Ахо-Корасик курильщика (step 1)

Шаг 1: построим бор по множеству  $P$ .

Идея: Будем идти по символам строки  $S$  и продвигаться по бору. Если встретили терминальную вершину, то нашли вхождение  $P$  в  $S$ .

Проблема: что делать, если в боре идти больше некуда?

$$P = \{a, aa, baa, bba, bbb, aba\}$$

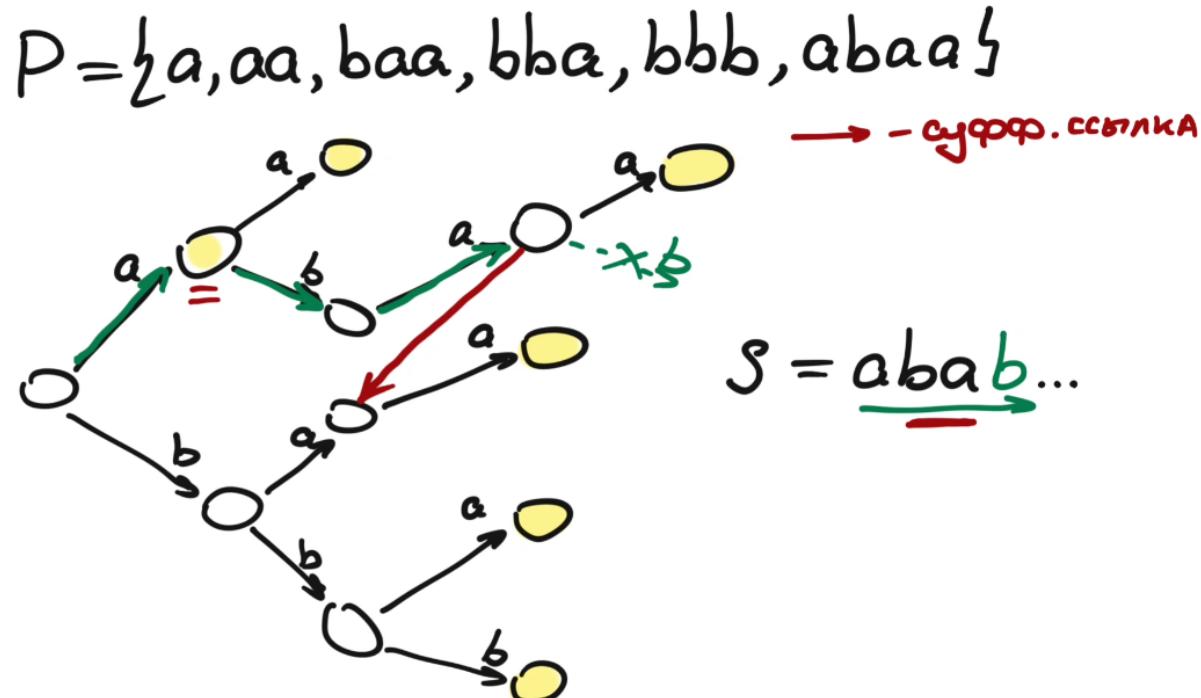


# Суффиксные ссылки

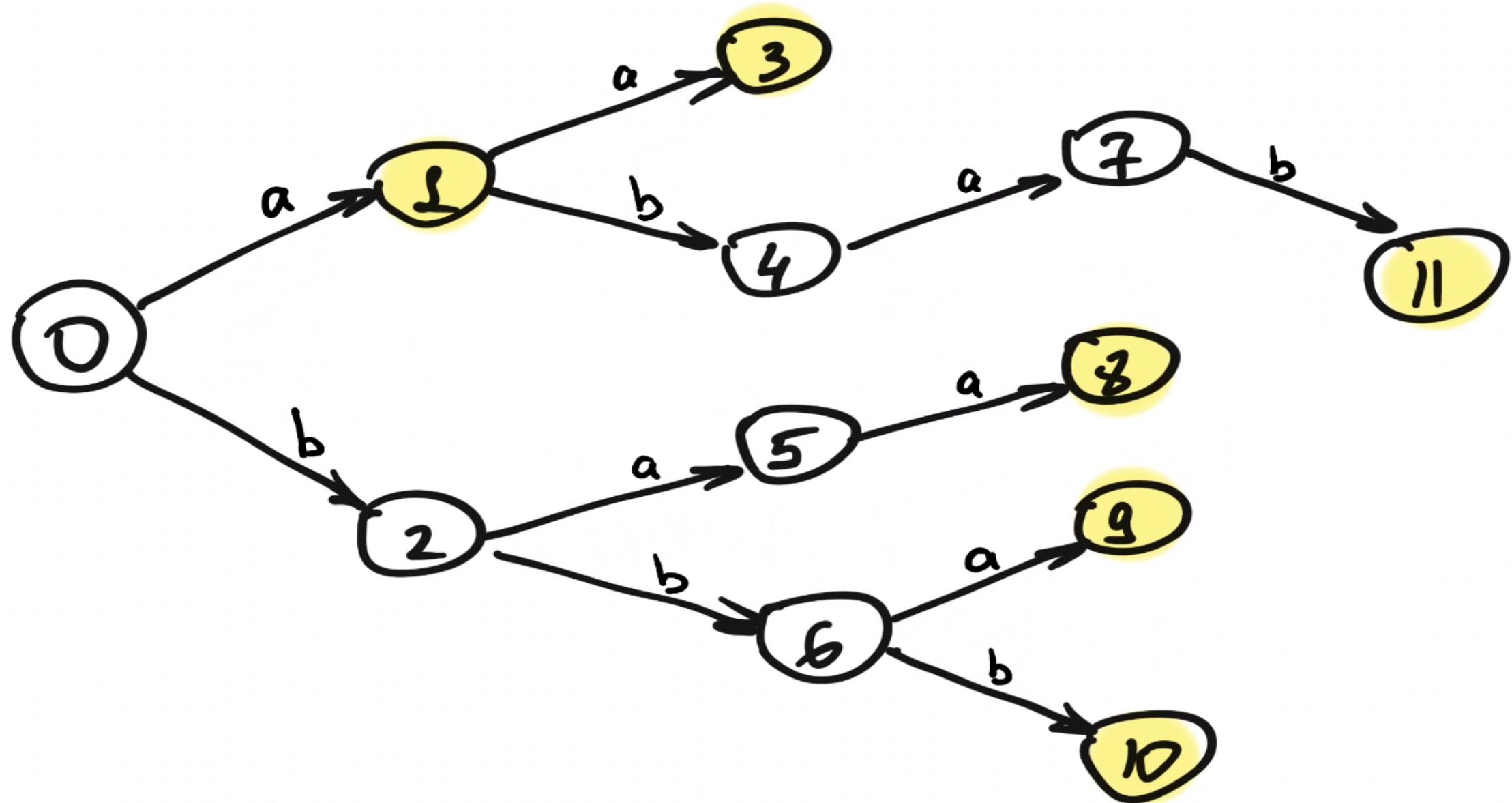
Проблема: что делать, если в боре идти больше некуда?

Хотелось бы перейти к наибольшему суффиксу текущей подстроки, который есть в боре, и попробовать продолжить его.

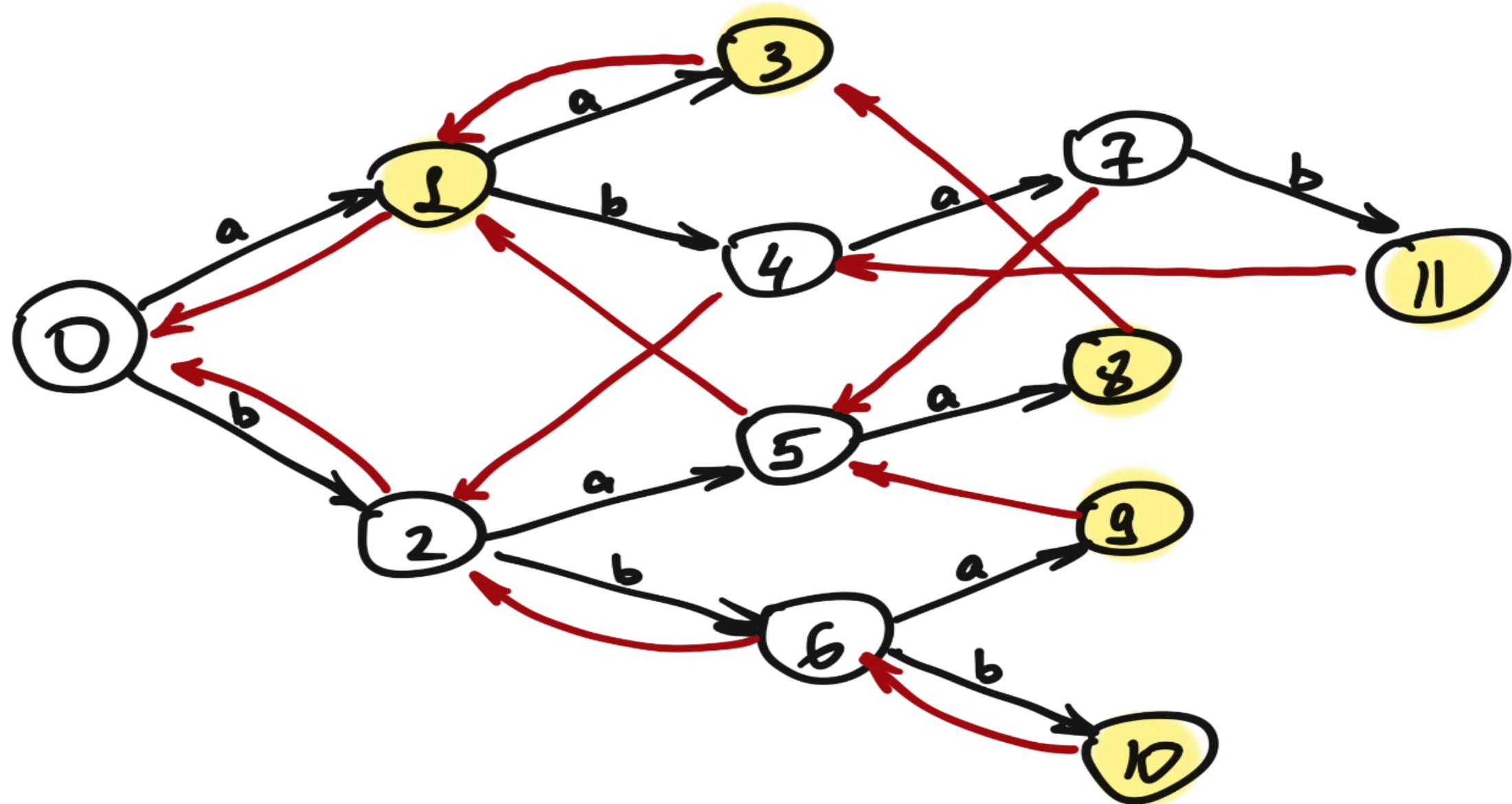
Данный переход будем называть суффиксной ссылкой.



# Суффиксные ссылки: пример



# Суффиксные ссылки: пример

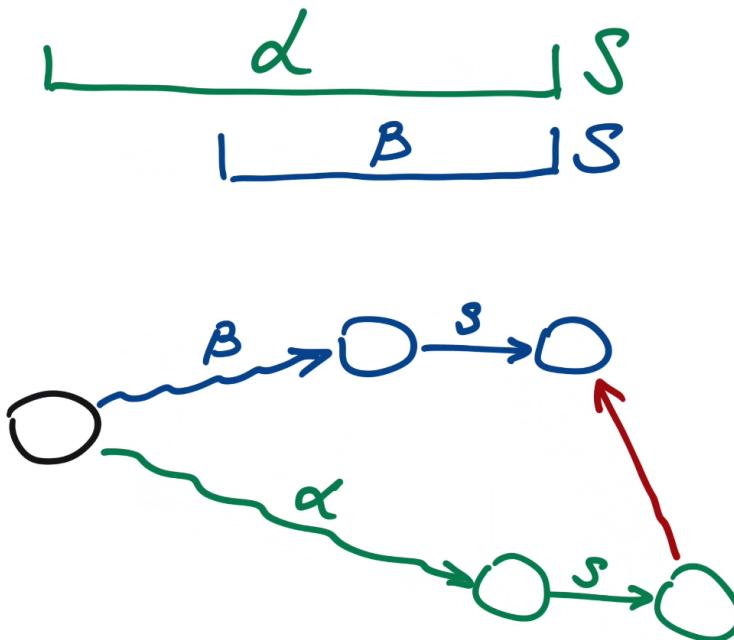


# Суффиксные ссылки: вычисление

Суффиксная ссылка для корня не определена.

Суффиксные ссылки детей корня ведут в корень.

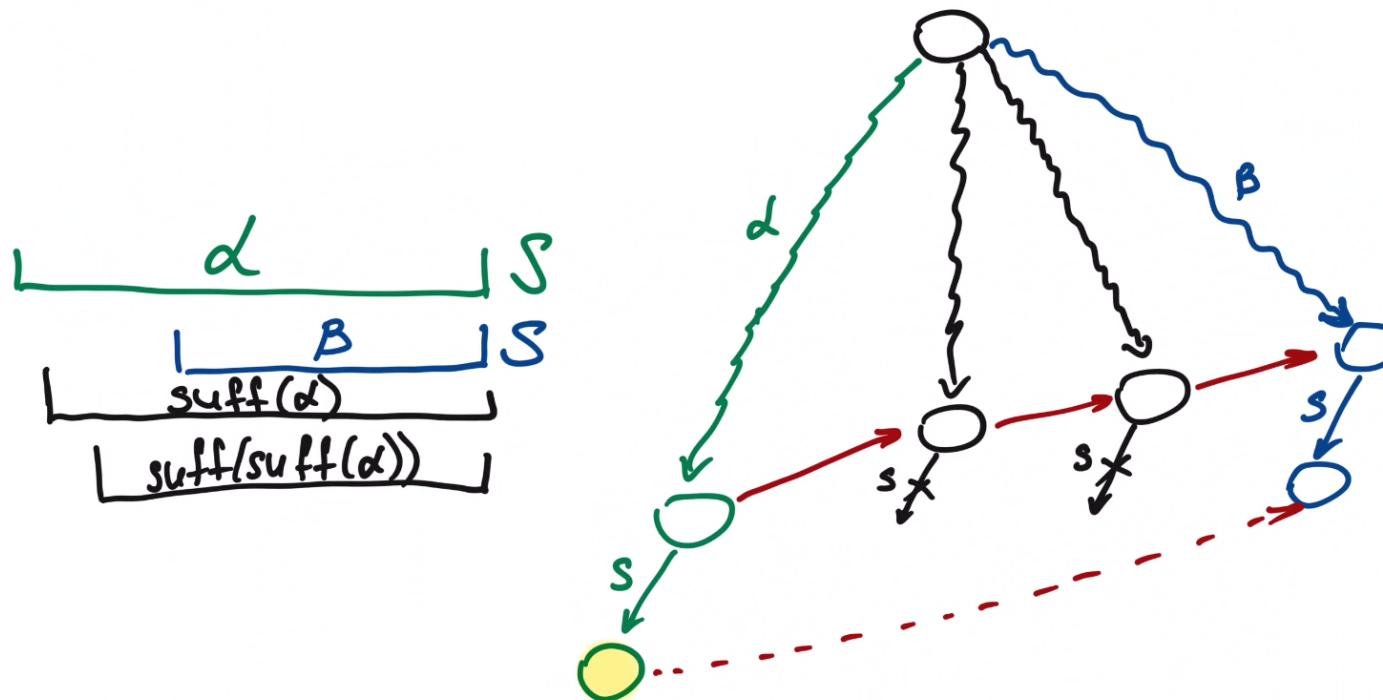
Хотим найти суффиксную ссылку для вершины  $\alpha s$  ( $\alpha$  - префикс,  $s$  - последний символ). Она либо имеет вид  $\beta s$ , где  $\beta$  - суффикс  $\alpha$ , либо является ссылкой на корень (в боре нет непустого суффикса строки  $\alpha s$ ).



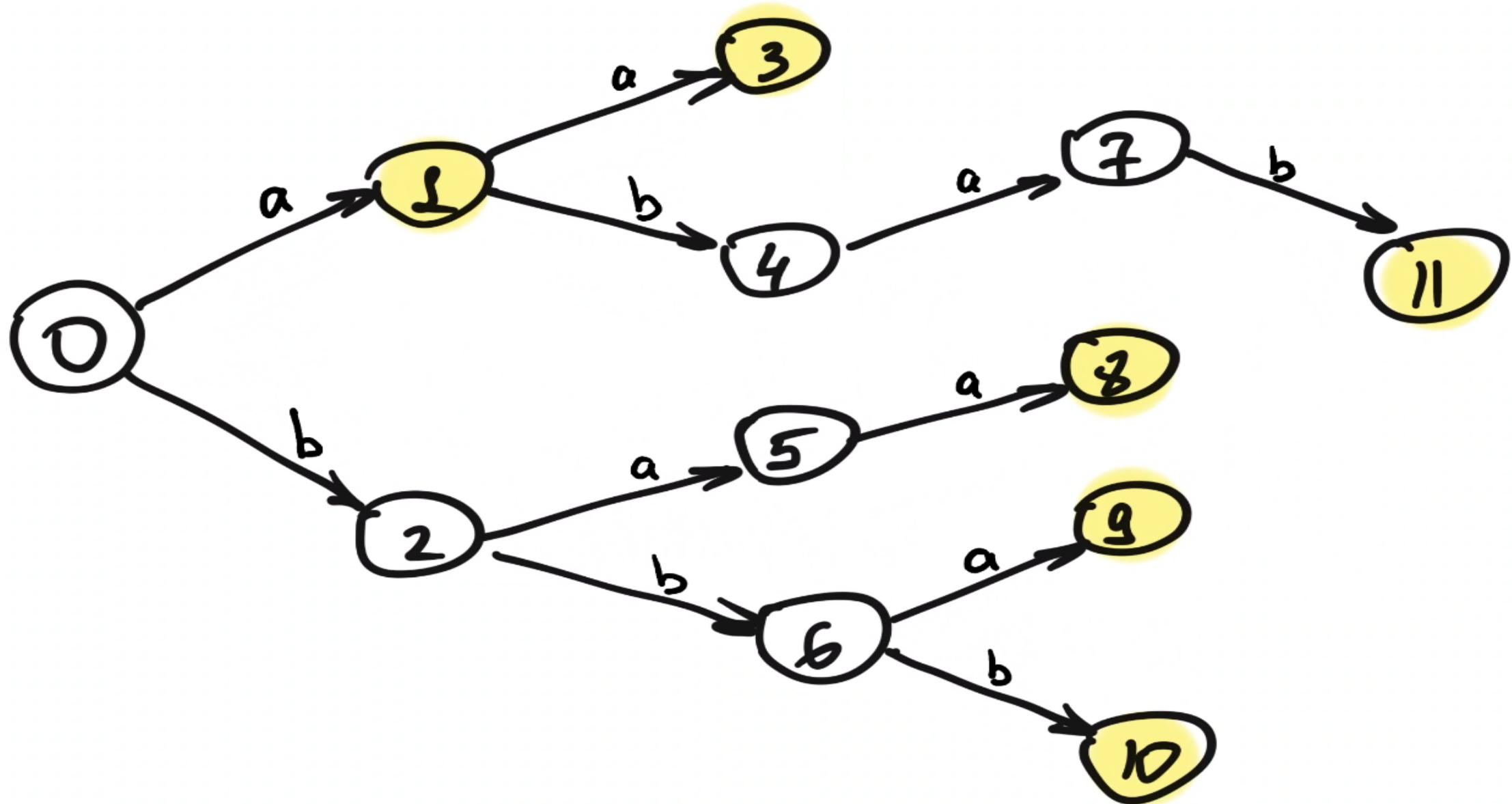
# Суффиксные ссылки: вычисление

Хотим найти суффиксную ссылку для вершины  $\alpha s$  ( $\alpha$  - префикс,  $s$  - последний символ). Она либо имеет вид  $\beta s$ , где  $\beta$  - суффикс  $\alpha$ , либо является ссылкой на корень (в боре нет непустого суффикса строки  $\alpha s$ ).

Будем перебирать суффиксы строки  $\alpha$  в порядке убывания пока не найдем переход по  $s$ . Как это сделать? - Переходить по суффиксным ссылкам.



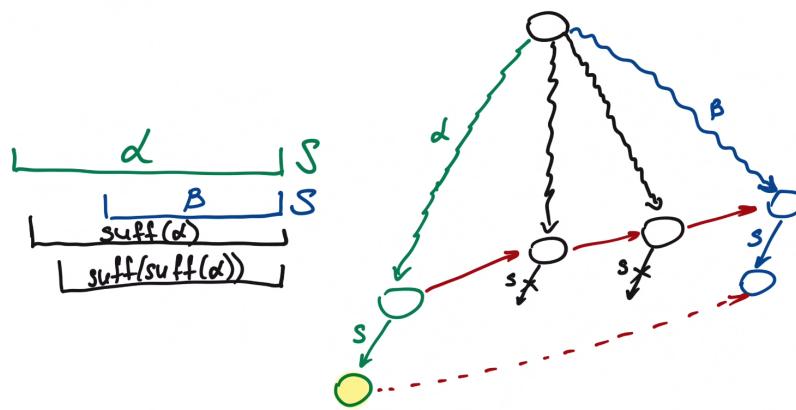
# Суффиксные ссылки: вычисление



# Суффиксные ссылки: алгоритм

Будем строить суфф. ссылки по слоям - с помощью BFS.

```
def Trie.BuildSuffixReferences():
    root.suff_ref = None
    queue = {(symbol, root, node) for (symbol, node) in root.next}
    while queue is not empty:
        s, parent, node = queue.pop()
        suff = parent.suff_ref
        while (suff is not None) and (s not in suff.next):
            suff = suff.suff_ref
        node.suff_ref = (suff is None ? root : suff.next[s])
        for (symbol, neighbor) in node.next:
            queue.push((symbol, node, neighbor))
```



# Суффиксные ссылки: алгоритм

Будем строить суфф. ссылки по слоям - с помощью BFS.

```
def Trie.BuildSuffixReferences():
    root.suff_ref = None
    queue = {(symbol, root, node) for (symbol, node) in root.next}
    while queue is not empty:
        s, parent, node = queue.pop()
        suff = parent.suff_ref
        while (suff is not None) and (s not in suff.next):
            suff = suff.suff_ref
        node.suff_ref = (suff is None ? root : suff.next[s])
        for (symbol, neighbor) in node.next:
            queue.push((symbol, node, neighbor))
```

Рассмотрим путь от корня до листа ( $P_i$ ). Для каждой вершины на этом пути мы несколько раз поднимаемся по суфф.ссылкам и не более одного раза спускаемся на одну вершину вдоль  $s$ . Суммарно подъемов  $\leq$  спусков (коих  $\leq |P_i|$ ). Таким образом, общее время построения  $O(\sum |P_i|)$

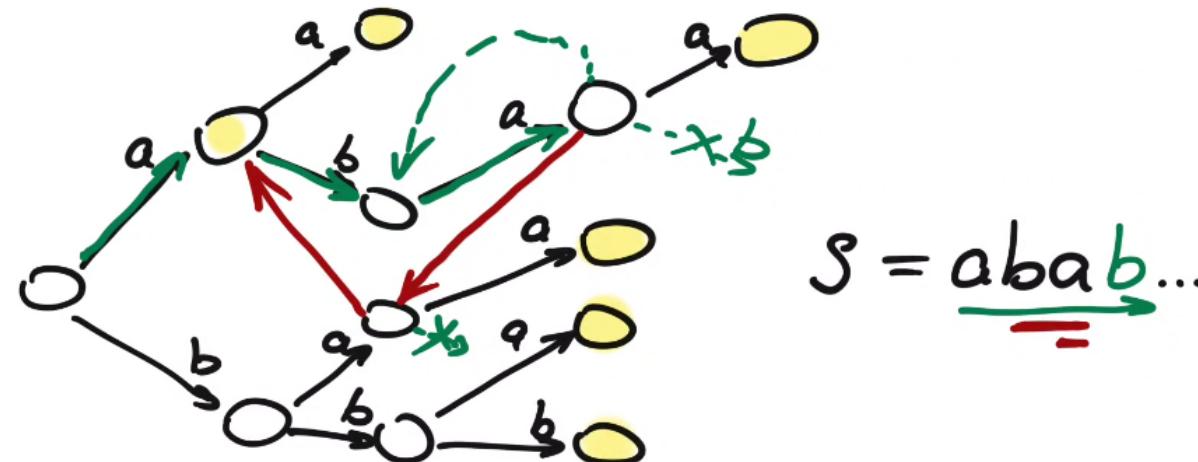
# Ахо-Корасик курильщика (step 2)

Шаг 1: построим бор по множеству  $P$ .

Шаг 2: вычислим суффиксные ссылки.

**Идея:** Будем идти по символам строки  $S$  и продвигаться по бору. Если встретили терминальную вершину, то нашли вхождение  $P$  в  $S$ . Если идти некуда, то переходим по суффиксной ссылке и пытаемся продвинуться там. Если находимся в корне и нет перехода, то рассматриваем следующий символ.

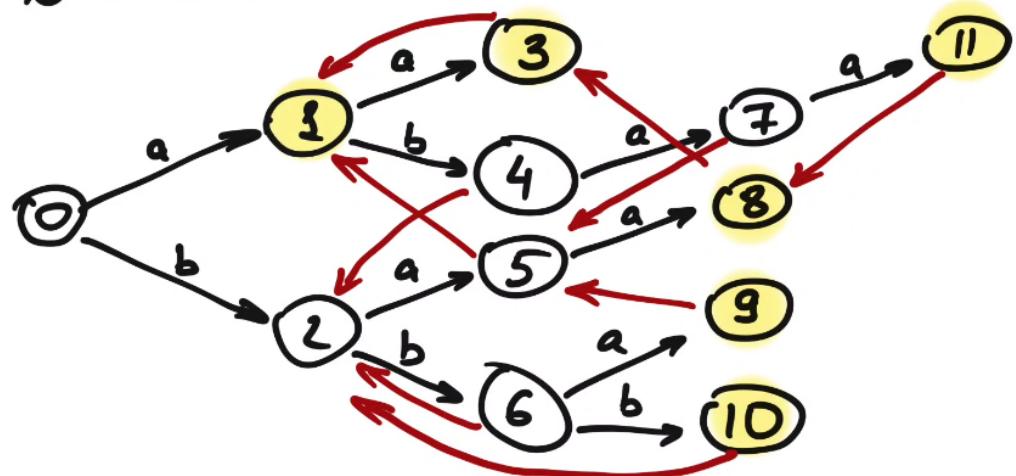
$$P = \{a, aa, baa, bba, bbb, aba\}$$



## Ахо-Корасик курильщика (step 2)

Идея: Будем идти по символам строки  $S$  и продвигаться по бору. Если встретили терминальную вершину, то нашли вхождение  $P$  в  $S$ . Если идти некуда, то переходим по суффиксной ссылке и пытаемся продвинуться там. Если находимся в корне и нет перехода, то рассматриваем следующий символ.

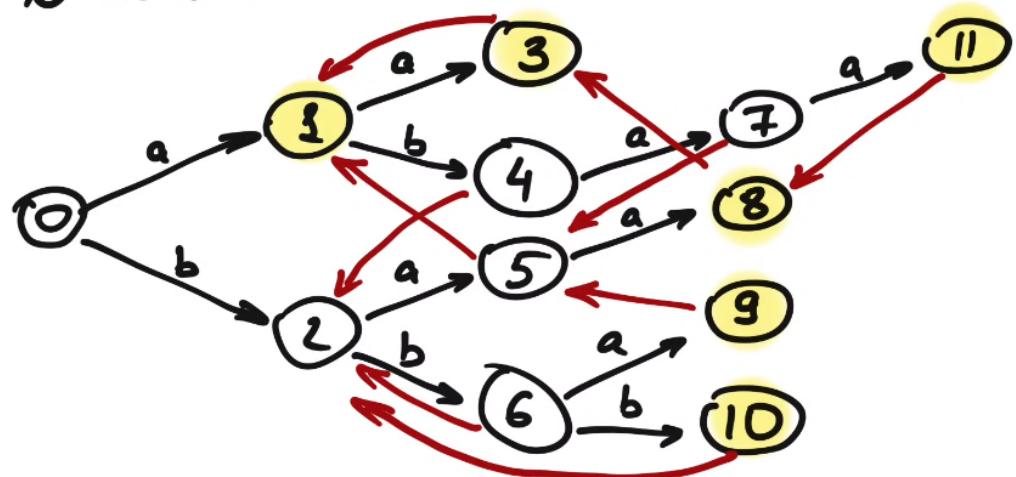
$$P = \{a, aa, baa, bba, bbb, aba\}$$
$$S = ababaaa$$



# Ахо-Корасик курильщика (step 2)

Идея: Будем идти по символам строки  $S$  и продвигаться по бору. Если встретили терминальную вершину, то нашли вхождение  $P$  в  $S$ . Если идти некуда, то переходим по суффиксной ссылке. Если находимся в корне и нет перехода, то рассматриваем следующий символ.

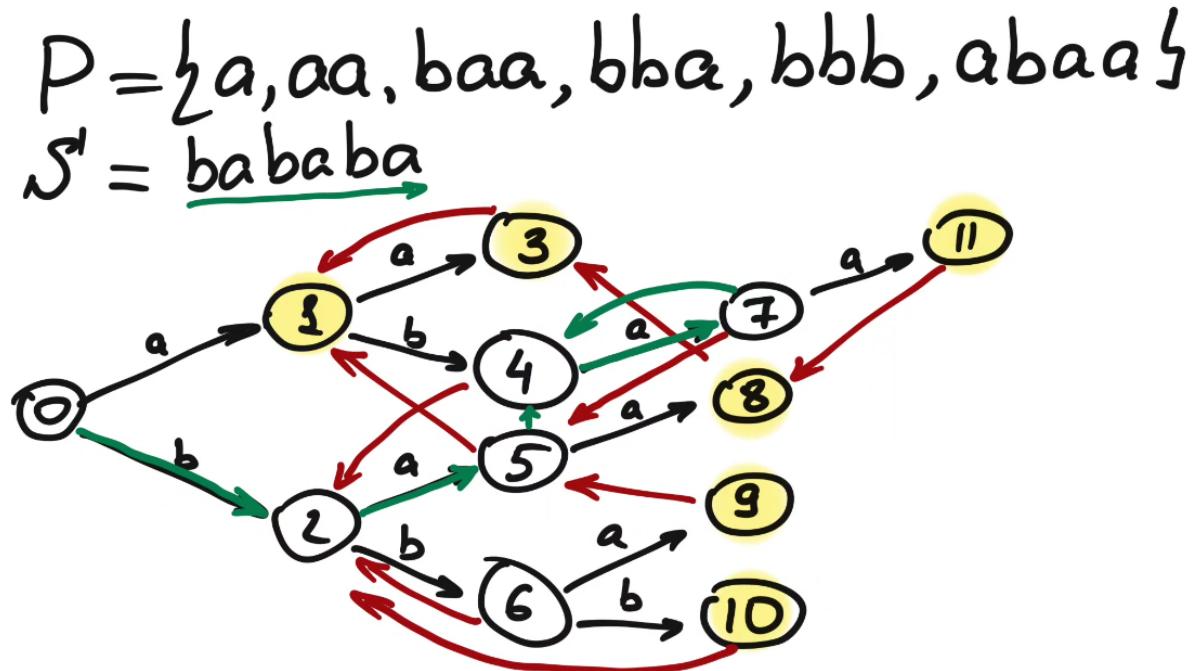
$$P = \{a, aa, baa, bba, bbb, aba\}$$
$$S = ababaaa$$



Найденные вхождения: "a": 0, "abaa": 2, "aa": 5.

## Ахо-Корасик курильщика (step 2)

Проблема: мы пропускаем много вхождений подстрок. Может даже все:



Ничего не нашли (ни разу не побывали в терминальной вершине).

# Сжатые суффиксные ссылки

Проблема: мы пропускаем много вхождений подстрок.

Беда в том, что мы жадно пытаемся найти наибольшую подстроку, пропуская мелкие.



# Сжатые суффиксные ссылки

**Проблема:** мы пропускаем много вхождений подстрок.

Беда в том, что мы жадно пытаемся найти наибольшую подстроку, пропуская мелкие.

**Решение:** перебрать все суффиксы (по суфф. ссылкам) и найти терминальные.



# Сжатые суффиксные ссылки

**Проблема:** мы пропускаем много вхождений подстрок.

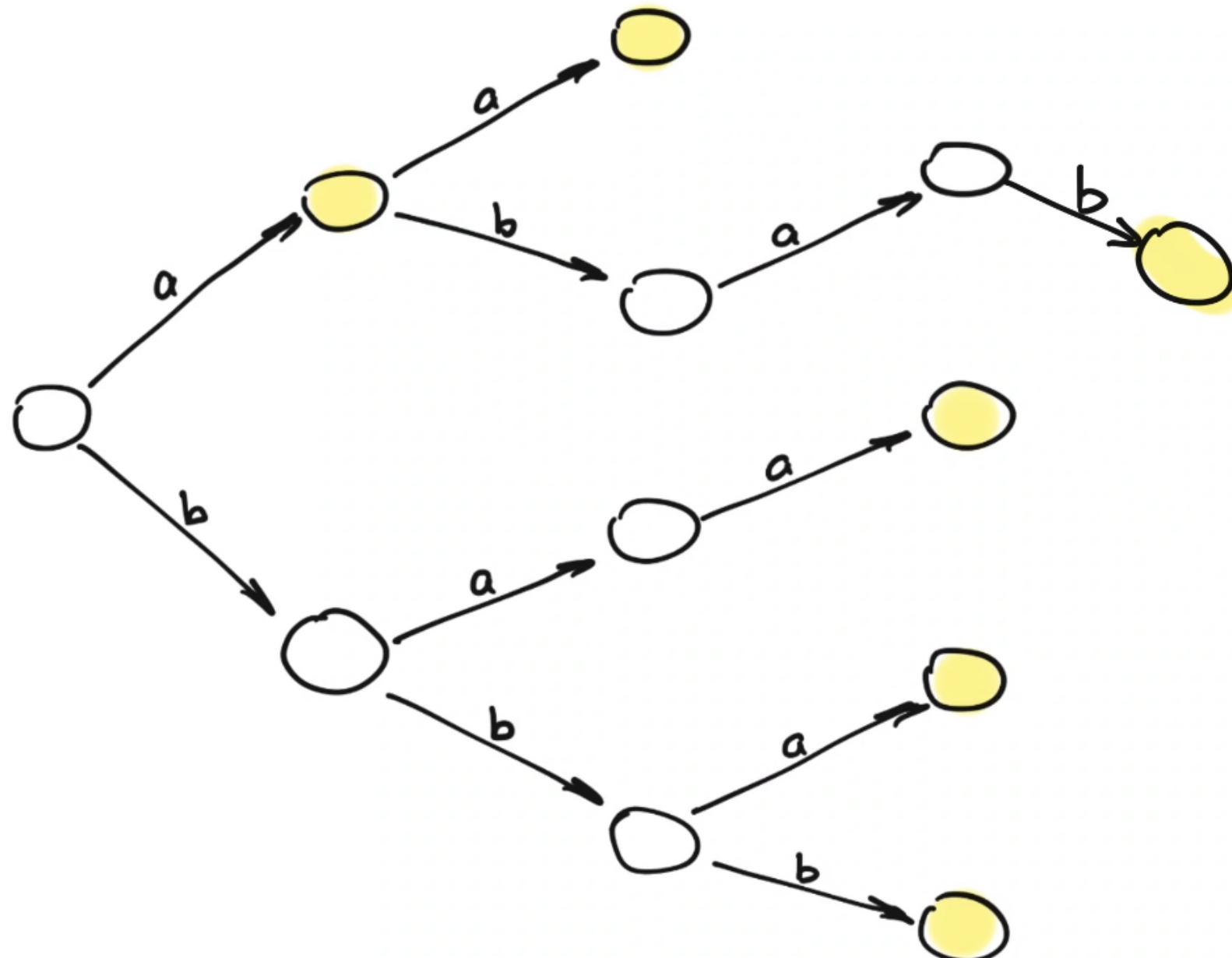
Беда в том, что мы жадно пытаемся найти наибольшую подстроку, пропуская мелкие.

**Решение:** перебрать все суффиксы (по суфф. ссылкам) и найти терминальные. перебрать все терминальные суффиксы (по сжатым суфф. ссылкам).

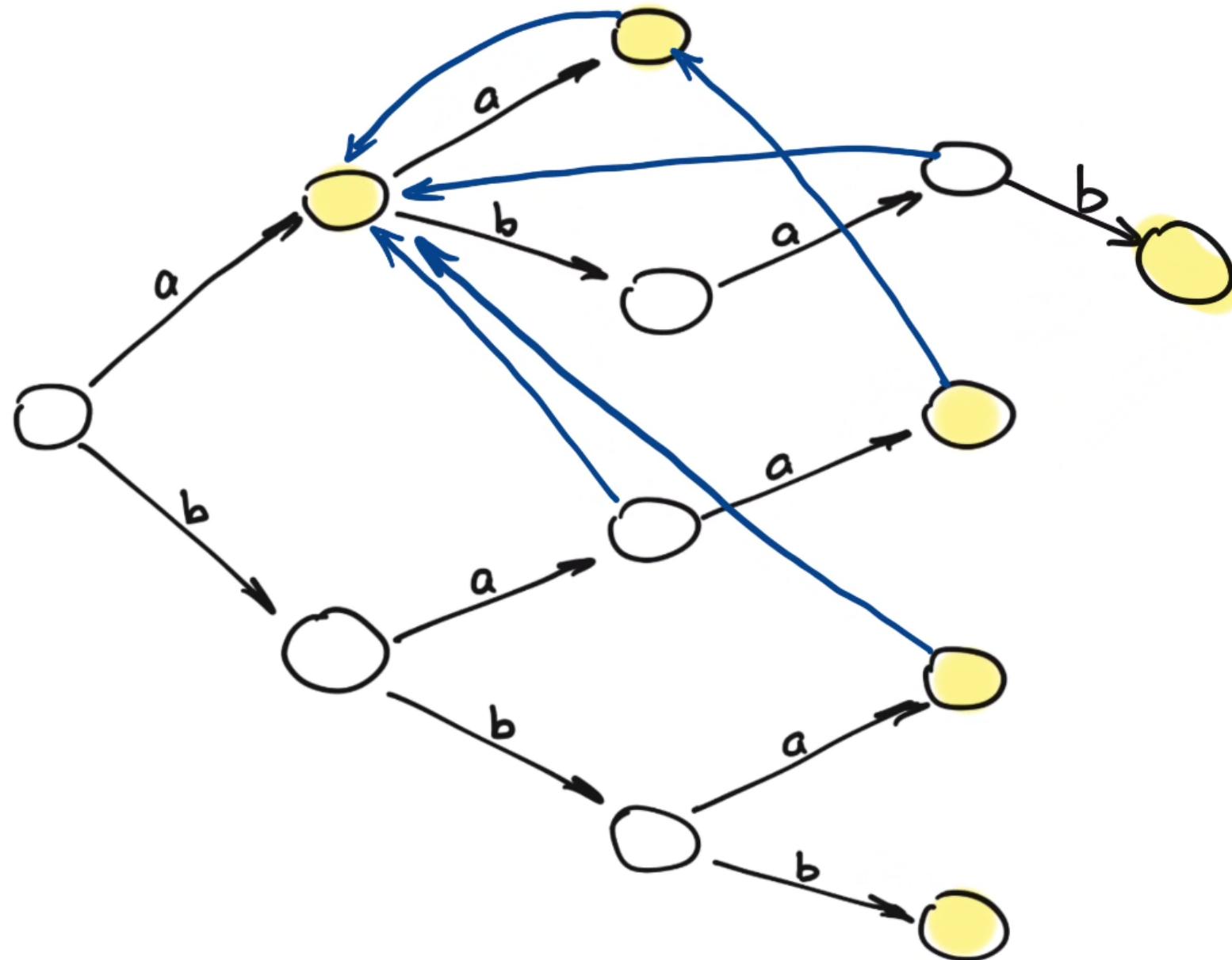
Сжатая суффиксная ссылка - ссылка в наибольший терминальный суффикс.



# Сжатые суффиксные ссылки: пример



# Сжатые суффиксные ссылки: пример

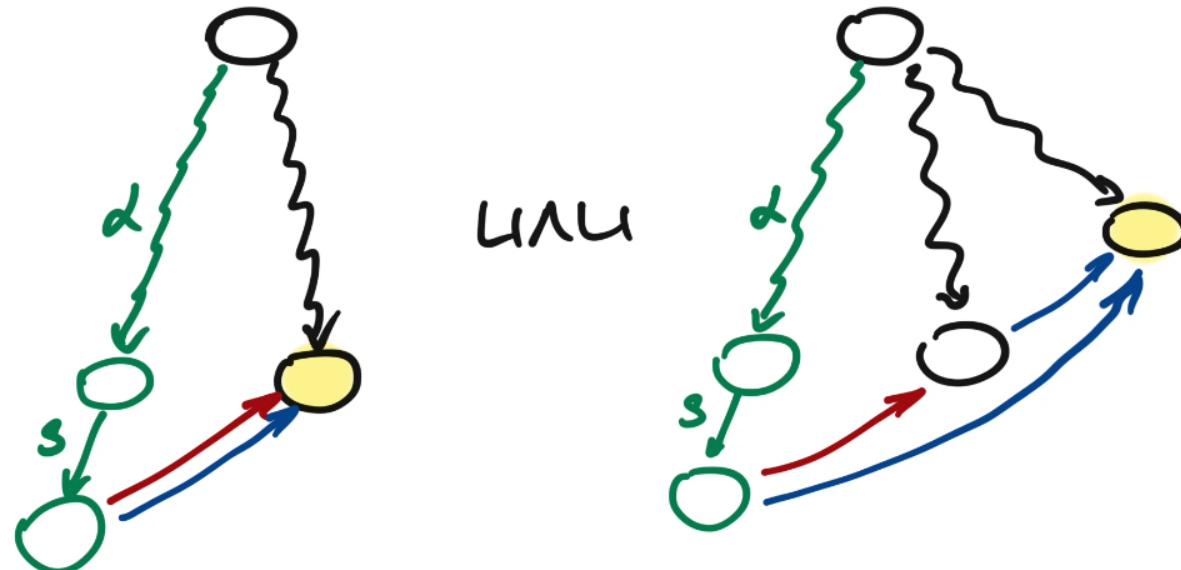


# Сжатые суффиксные ссылки: вычисление

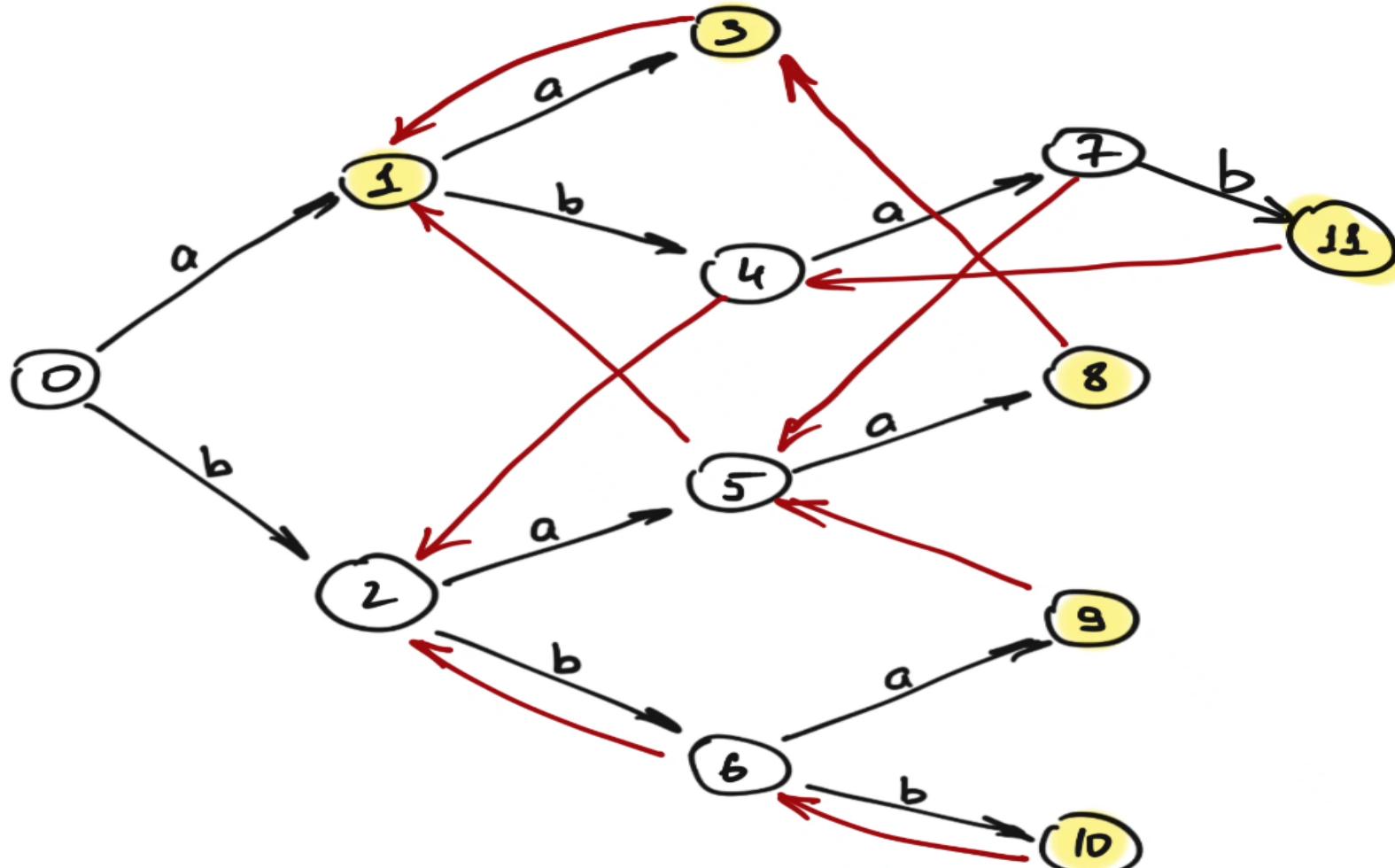
Сжатых суффиксных ссылок для корня и его детей нет.

Хотим найти сжатую суффиксную ссылку для вершины  $\alpha s$  ( $\alpha$  - префикс,  $s$  - последний символ).

- Если суффиксная ссылка ведет в терминальную, то она же является и сжатой (по определению сжатой суффиксной ссылки).
- Если нет, то ответ - сжатая суффиксная ссылка суффиксной ссылки.



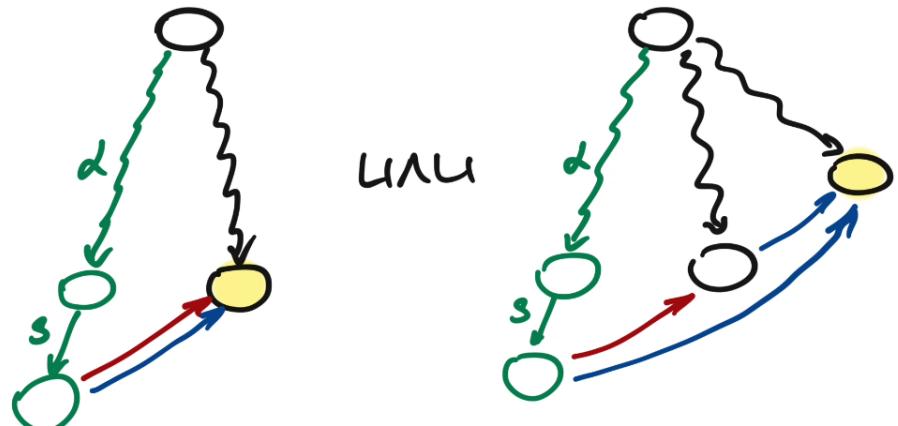
# Сжатые суффиксные ссылки: вычисление



# Сжатые суффиксные ссылки: алгоритм

Будем строить сжатые суфф. ссылки по слоям - с помощью BFS.

```
def Trie.BuildExitLinks():
    root.exit_link = None
    queue = {node for _, node in root.next}
    while queue is not empty:
        node = queue.pop()
        suff = node.suff_ref
        if suff != root:
            node.exit_link = (suff.is_terminal ? suff : suff.exit_link)
        for _, neighbor in node.next:
            queue.push(neighbor)
```



# Сжатые суффиксные ссылки: алгоритм

Будем строить сжатые суфф. ссылки по слоям - с помощью BFS.

```
def Trie.BuildExitLinks():
    root.exit_link = None
    queue = {node for (_, node) in root.next}
    while queue is not empty:
        node = queue.pop()
        suff = node.suff_ref
        if suff != root:
            node.exit_link = (suff.is_terminal ? suff : suff.exit_link)
        for (_, neighbor) in node.next:
            queue.push(neighbor)
```

Каждая сжатая суффиксная ссылка вычисляется за  $O(1)$ . Поэтому общая сложность  $O(nodes.size()) \subset O(\sum |P_i|)$

# Алгоритм Ахо-Корасик здорового человека

**Шаг 1:** построим бор по множеству  $P$ .

**Шаг 2:** вычислим суффиксные ссылки.

**Шаг 3:** вычислим сжатые суффиксные ссылки.

**Алгоритм:** считываем  $S$  поэлементно, продвигаясь по бору. В каждой вершине проверяем терминальность, а также проходим по сжатым суфф. ссылкам для проверки вхождения подстрок в строку  $S$ .

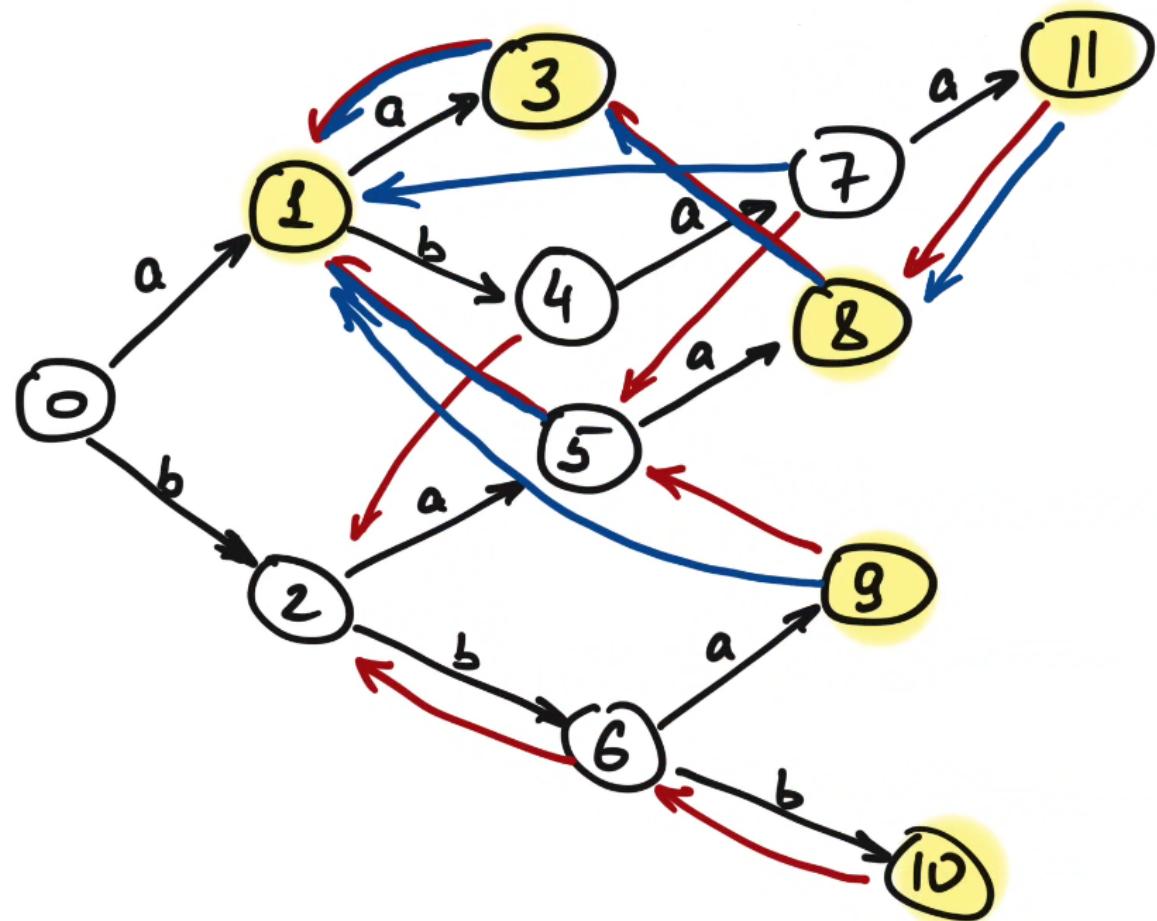
Если нет нужного ребра, то идем по суфф. ссылке и пытаемся пройти по ребру там, если снова провал, то идем по суфф. ссылке и так далее.

Если находимся в корне и нет перехода, то переходим к следующей букве.

# Алгоритм Ахо-Корасик

$P = \{a, aa, baa, bba, bbb, abaab\}$

$S = abaabcbba$



# Алгоритм Ахо-Корасик

```
def NextState(node, symbol):
    while node is not None and symbol not in node.next:
        node = node.suff_ref
    return node is None ? root : node.next[symbol]

def PrintEntries(index, node):
    if not node.is_terminal:
        node = node.exit_link
    while node is not None:
        print(index, node)
        node = node.exit_link

def AhoCorasick(str, patterns):
    trie = BuildTrie(patterns)
    trie.BuildSuffixReferences()
    trie.BuildExitLinks()
    node = trie.root
    for i in range(|str|):
        node = NextState(str[i], node)
        PrintEntries(i, node)
```

# Алгоритм Ахо-Корасик

```
def Ahocorasick(str, patterns):
    trie = BuildTrie(patterns)
    trie.BuildSuffixReferences()
    trie.BuildExitLinks()
    node = trie.root
    for i in range(|str|):
        node = NextState(str[i], node)
        PrintEntries(i, node)
```

Построение бора -  $O(\sum |P_i|)$

Построение суффиксных и сжатых ссылок -  $O(\sum |P_i|)$

При считывании очередного символа строки  $S_j$  несколько раз поднимаемся по суфф. ссылкам и один раз спускаемся по *next*. Число подъемов  $\leq$  число спусков  $\leq |S_j|$ . Значит сложность обработки одной строки  $O(|S_j|)$ .

*PrintEntries* суммарно работает за  $O(|ans|)$ .

Сложность алгоритма:  $O(\sum |P_i| + \sum |S_j| + |ans|)$

Понять все: бесценно

