Suffix tries, trees and arrays

CMSC420 0101

Spring 2019

String matching problem

- Suppose we have a (large) text T, with length (#characters) t.

String matching problem

- Suppose we have a (large) text T, with length (#characters) t.
- We also have a pattern (a smaller string) P, with length p.
    - It is assumed that t >> p.

String matching problem

- Suppose we have a (large) text T, with length (#characters) n.
- We also have a pattern (a smaller string) P, with length m.
  - It is assumed that t >> p.
- Then, the string matching problem consists of answering the question:

"Does P occur in T"?

String matching problem

- Suppose we have a (large) text T, with length (#characters) t.
- We also have a pattern (a smaller string) P, with length p.
  - It is assumed that t >> p.
- Then, the string matching problem consists of answering the question:

"Does P occur in T"?

- Or its common practical variant:

"Give me the positions (if any) where P occurs in T."

Naïve approach

```
function naiveMatcher(T: a string text, P: a string pattern){
      for(i = 0 : n – m – 1){ // Runs n – m = O(n) times
            if(T[i:m] == P){ // O(m) check
                  print "String matched at position " + i;
            }
      }
}
```

- This simple algorithm performs $(n - m) * m = n * m - m^2$ character checks (elementary operations), which is $\mathcal{O}(n \cdot m)$
- Goal: do better than $\mathcal{O}(n \cdot m)$:)

Types of string matching algorithms

1. Ones that pre-process the ***pattern***
   - Knuth-Morris Pratt (KMP)
   - Boyer-Moore
2. Ones that pre-process the ***text***
   - Rabin-Karp
   - Suffix tries / trees
   - Suffix arrays
   - Extended suffix arrays, LCP arrays…

Part 1: Pre-processing the ***pattern*** with KMP

KMP

- We will first discuss KMP as an example of an algorithm of type 1.

KMP

- We will first discuss KMP as an example of an algorithm of type 1.
- ***Key observation***: Suppose we try to match the pattern and the text across m characters. We encounter a mismatch at some character c.

KMP

- We will first discuss KMP as an example of an algorithm of type 1.
- ***Key observation***: Suppose we try to match the pattern and the text across m characters. We encounter a mismatch at some character c.

  Then, if a **suffix** of the ***matched portion*** of the pattern is ***also a prefix of the matched portion***, we should shift the pattern such that the next check happens at the character after that prefix! Otherwise, we shift the pattern by one position.

- Let's look at an example.

So, how do we do this?

- The algorithm uses an m-sized array F that is known as the *prefix* or *failure* function.
- Suppose we have an index i, 0 <= i < m. The part of the prefix from 0 to i (inclusive) will be notated P[0:i].
- Then, F[i] represents *the length of the longest suffix of* P[0:i] *that is also a prefix of it.*
- Example:

	0	1	2	3	4	5	6
P	e	t	e	t	e	s	e
F	0	0	1	2	3	0	1

Exercise!

- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	a	x	b	x	x	b	a	x	x	a	x	b
F														

Exercise!

- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	**x**	**x**	**a**	**x**	**b**	**x**	**x**	**b**	**a**	**x**	**x**	**a**	**x**	**b**
F	**0**	**1**	**0**	**1**	**0**	**1**	**2**	**0**	**0**	**1**	**2**	**3**	**4**	**5**

Exercise!

- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	a	x	b	x	x	b	a	x	x	a	x	b
F	0	1	0	1	0	1	2	0	0	1	2	3	4	5

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	x	x	x	x	x	x	x	x	x	x	x	a
F														

Exercise!

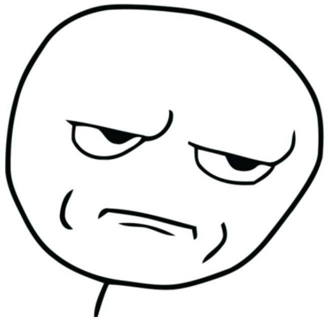
- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	a	x	b	x	x	b	a	x	x	a	x	b
F	0	1	0	1	0	1	2	0	0	1	2	3	4	5

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	x	x	x	x	x	x	x	x	x	x	x	a
F	0	1	2	3	4	5	6	7	8	9	10	11	12	0

Exercise!



- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	a	x	b	x	x	b	a	x	x	a	x	b
F	0	1	0	1	0	1	2	0	0	1	2	3	4	5

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	x	x	x	x	x	x	x	x	x	x	x	a
F	0	1	2	3	4	5	6	7	8	9	10	11	12	0

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	a	x	x	x	x	x	x	x	x	x	x	x	x	x
F														

Exercise!

- Compute the prefix function for the following pattern:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	a	x	b	x	x	b	a	x	x	a	x	b
F	0	1	0	1	0	1	2	0	0	1	2	3	4	5

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	x	x	x	x	x	x	x	x	x	x	x	x	x	a
F	0	1	2	3	4	5	6	7	8	9	10	11	12	0

- And this one!

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	a	x	x	x	x	x	x	x	x	x	x	x	x	x
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pseudocode for building F

```
Compute-Prefix-Function(p: String pattern){
     m = |p|;
     F = array[m];
     F[0] = 0;
     k = 0;
     for(q=1:m-1){
          while((k > 0) && (p[k] != p[q]) ){ // As long as you can’t match….
               k = F[k-1];                   // Keep backtracking to find other potential prefixes that are also suffixes.
          }
          if(p[k] == p[q]){
               k++;                          // Length of current matched prefix increased.
          }
          F[q] = k;                          // Set the current value to the length of the maximal prefix that is also a suffix.
     }
     return F;
}
```

Pseudocode for building F

```
Compute-Prefix-Function(p: String pattern){
    m = |p|;
    F = array[m];
    F[0] = 0;
    k = 0;
    for(q=1:m-1){
        while((k > 0) && (p[k] != p[q]) ){  // As long as you can’t match….
            k = F[k-1];                     // Keep backtracking to find other potential prefixes that are also suffixes.
        }
        if(p[k] == p[q]){
            k++;                            // Length of current matched prefix increased.
        }
        F[q] = k;                           // Set the current value to the length of the maximal prefix that is also a suffix.
    }
    return F;
}
```

$O(m)$!

Using the prefix function

- Now that we have the prefix function ready, we can use it to do *sweet string matching type stuff ™*
- Recall: We want to improve upon $O(n \cdot m)$, naïve matching's complexity.
- The KMP matching algorithm processes every character of the text exactly once, leading to an improvement in performance!
- Contrast this with the naïve string matcher, where every character of the text gets compared to as many as m characters (every character of the pattern)

The KMP matcher

```
KMP-Matcher(p: String pattern, T: String text){
     m = |p|;
     n = |T|;
     F = Compute-Prefix-Function(p);
     q = 0;
     for(i = 0: n-1){
          while((q > 0) && (p[q] != T[i]) ){      // As long as you can’t match….
               q = F[q - 1];                      // Keep backtracking to find other potential matches.
          }
          if(p[q] == T[i]){
               q++;                               // Update the progress of our match.
          }
          if(q == m) {
               print “Pattern occurs with shift: ” + (i – m + 1);
               q = F[q – 1];                      // Don’t forget that there might be more matches! We should report them all!
          }
     }
}
```

The KMP matcher

```
KMP-Matcher(p: String pattern, T: String text){
     m = |p|;
     n = |T|;
     F = Compute-Prefix-Function(p);
     q = 0;
     for(i = 0: n-1){
          while((q > 0) && (p[q] != T[i]) ){       // As long as you can't match….
               q = F[q - 1];                        // Keep backtracking to find other potential matches.
          }
          if(p[q] == T[i]){
               q++;                                 // Update the progress of our match.
          }
          if(q == m) {
               print "Pattern occurs with shift: " + (i – m + 1);
               q = F[q – 1];                        // Don't forget that there might be more matches! We should report them all!
          }
     }
}
```

$O(n)$!

Example

- Let’s trace the KMP matcher for matching the following pattern to the following text.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	**b**	**c**	**a**	**b**	**a**	**b**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**b**	**a**	**c**	**a**	**c**
P	**a**	**b**	**a**	**b**	**a**	**c**	**a**											
F	**?**	**?**	**?**	**?**	**?**	**?**	**?**											

Complexity of KMP matching

- The KMP prefix function is calculated with a loop that has m iterations.
- The KMP matcher runs with a loop that has n iterations
  - This is the part that leads to the increased efficiency.
  - Every character of the text is examined only once! No backtracking on the text!
- All in all, the algorithm finds **all** matchings of the pattern to the text in $\mathcal{O}(m + n)$ time.
  - Compare with $\mathcal{O}(m \cdot n)$ for the naïve string matcher.
  - If we want to match the same pattern with a different text, no reason to re-build the prefix function! It is unique to the pattern. So we pay $O(n)$, a smaller price, every time we **re-use** the pattern.

Part 2: Pre-processing the ***text*** with suffix tries and trees

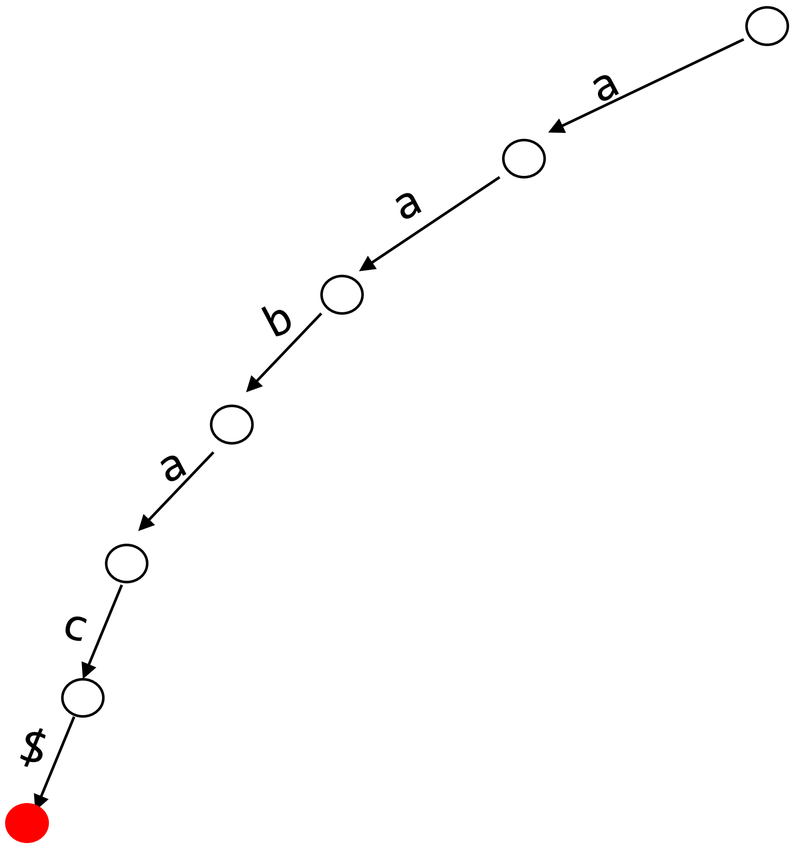
Suffix tries

- Key observation: If a pattern matches the text, it has to be the prefix of some suffix!
- Examples:
  - “key” in “keychain”
  - “chain” in “keychain”
  - “cha” in “keychain”
- We know that tries are excellent for finding prefixes!
- This motivates building a trie over the string’s suffixes!

Suffix tries

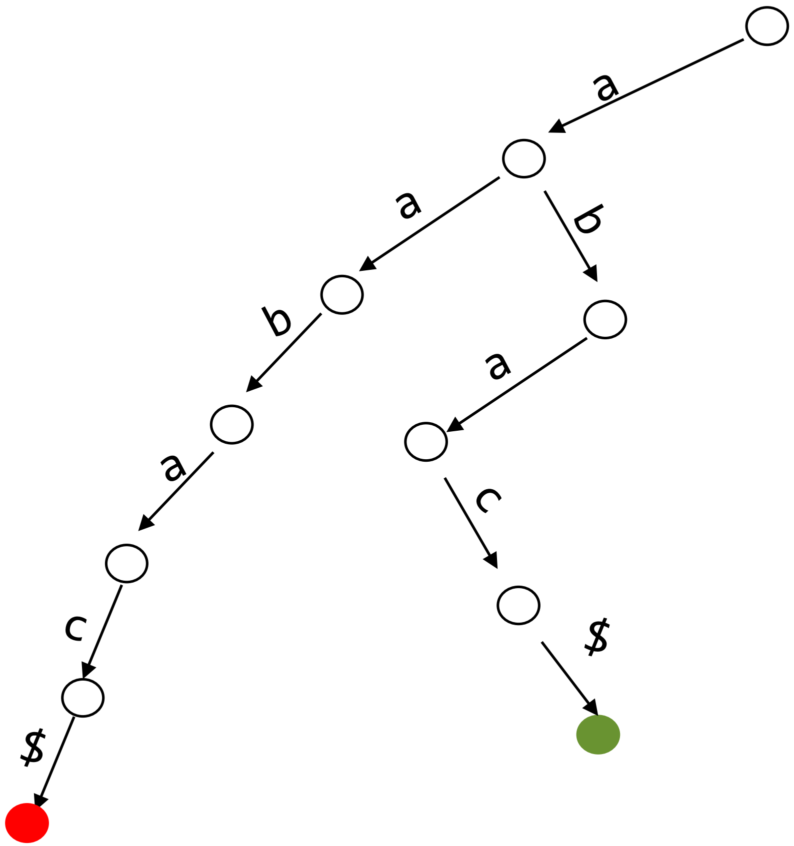
- Consider the string T=aabac.
- First thing we'll do is append a special character, '\$', to T, such that we now work with the "augmented" text `T$=aabac$`.
    - This character is used to signify which suffix trie nodes denote **actual** suffixes (will see what this means immediately).
- Then, we iterate through S, producing the substring `T[i:end]` for every value of $i \in \{0, 1, 2, \dots, m-1\}$, and inserting it into **an uncompressed trie**!
    - The result is known as a suffix trie.

Example suffix trie



T=aabac$

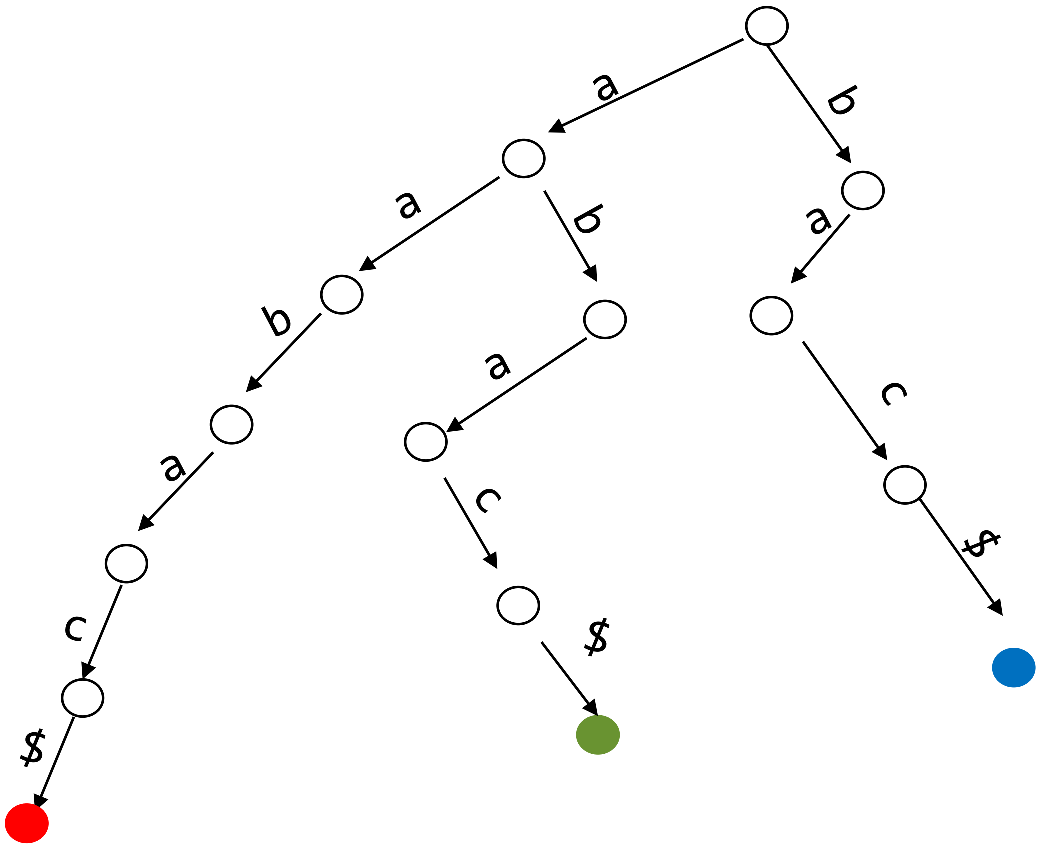
Example suffix trie



T=aabac$

abac$

Example suffix trie

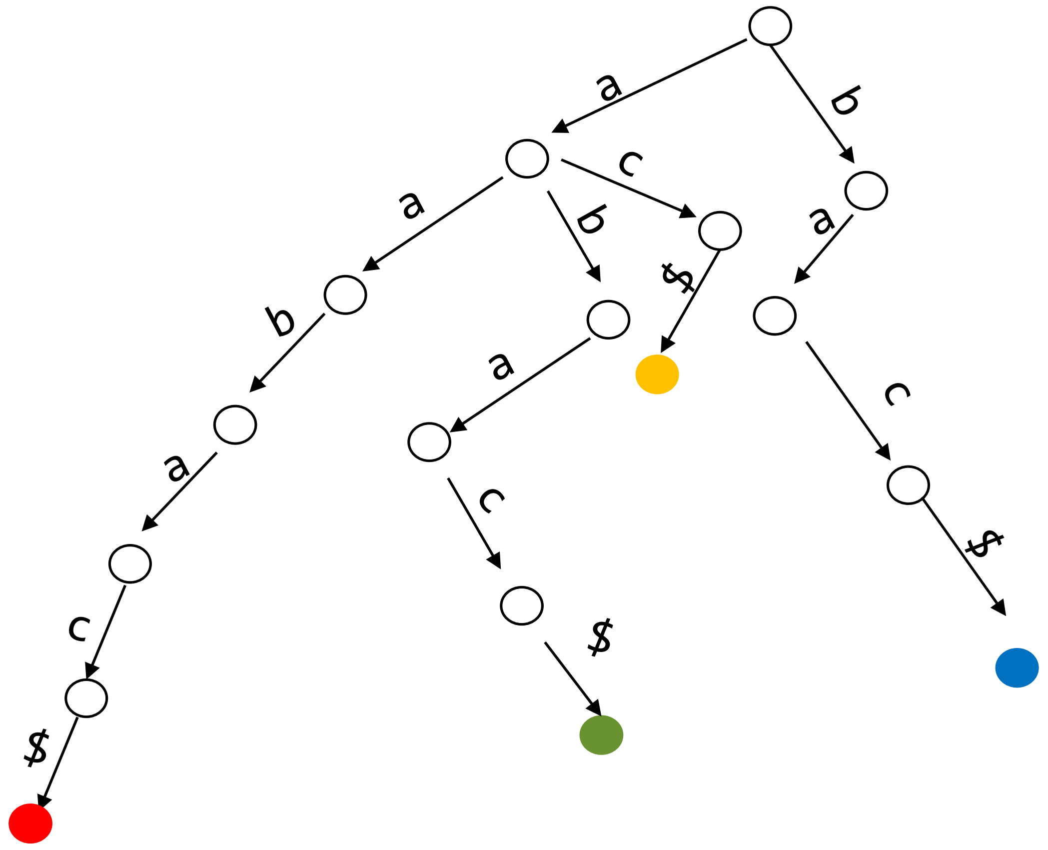


T=aabac$

abac$

bac$

Example suffix trie



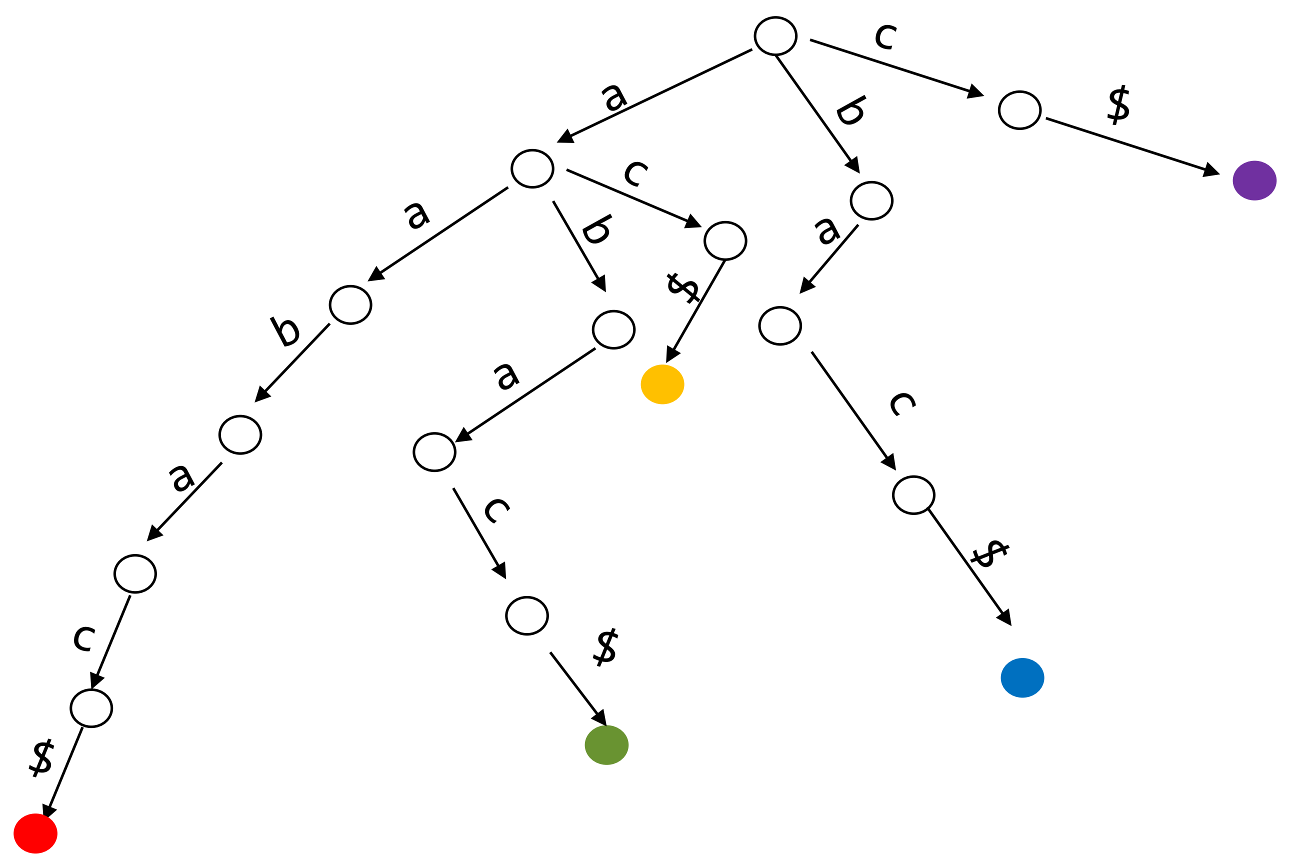
T=aabac$

abac$

bac$

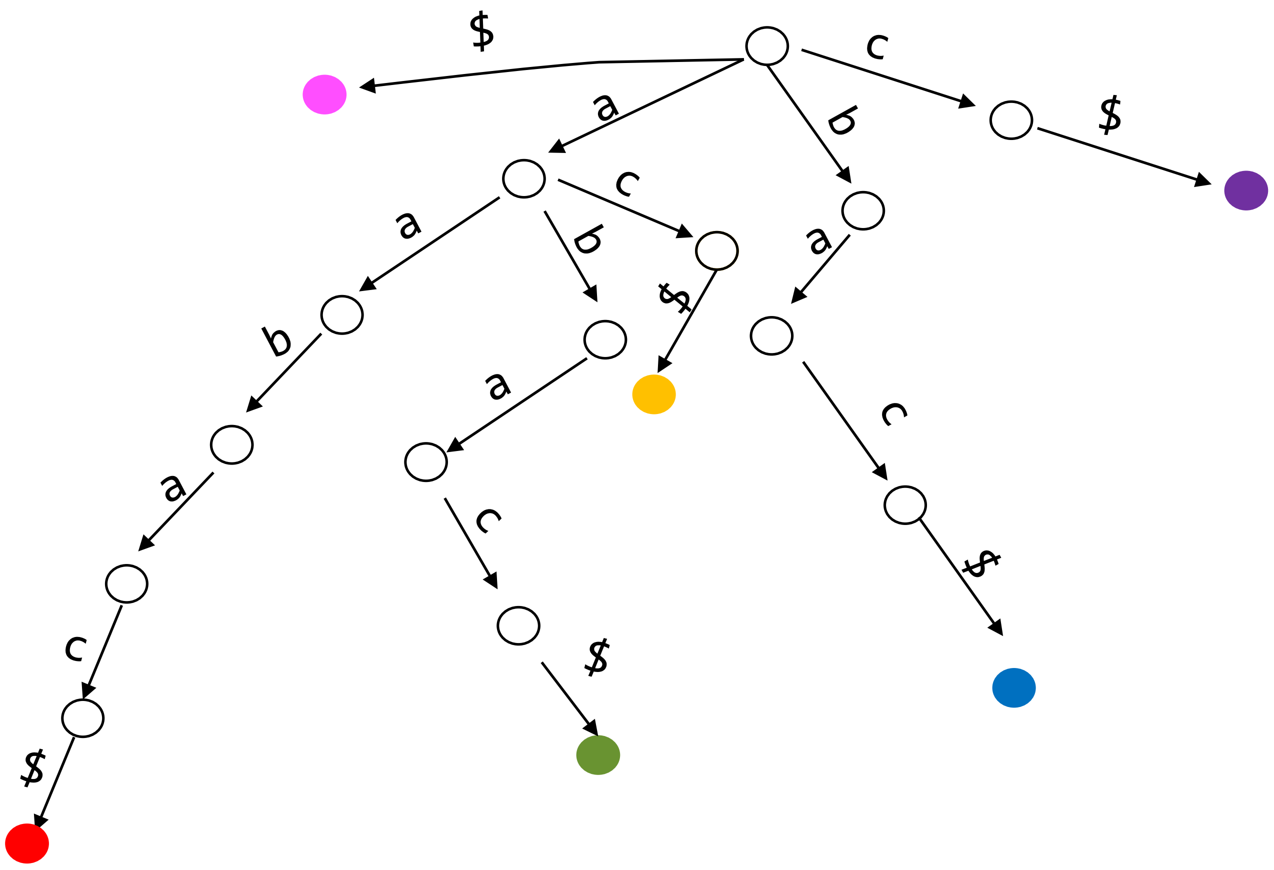
ac$

Example suffix trie



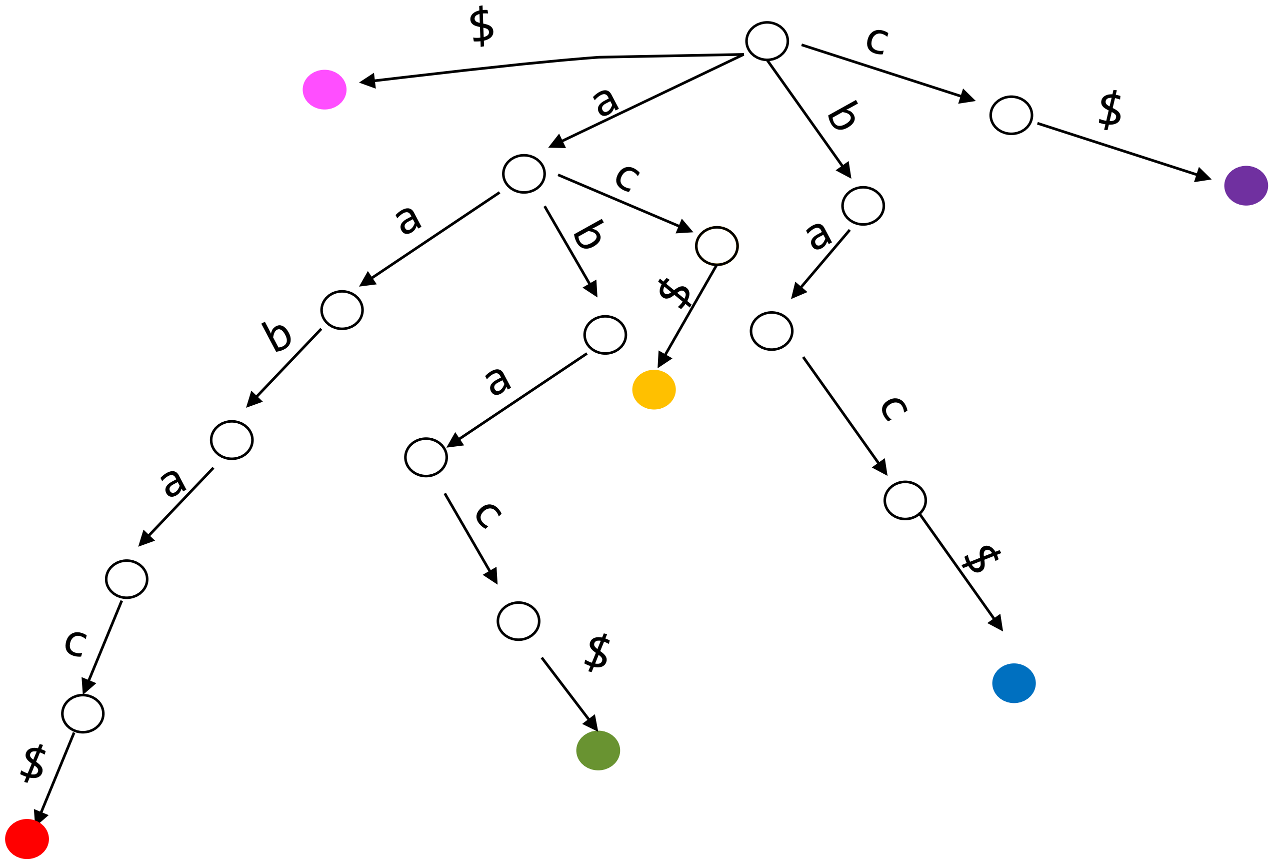
T=aabac$
abac$
bac$
ac$
c$

Example suffix trie



T=aabac$
abac$
bac$
ac$
c$
$

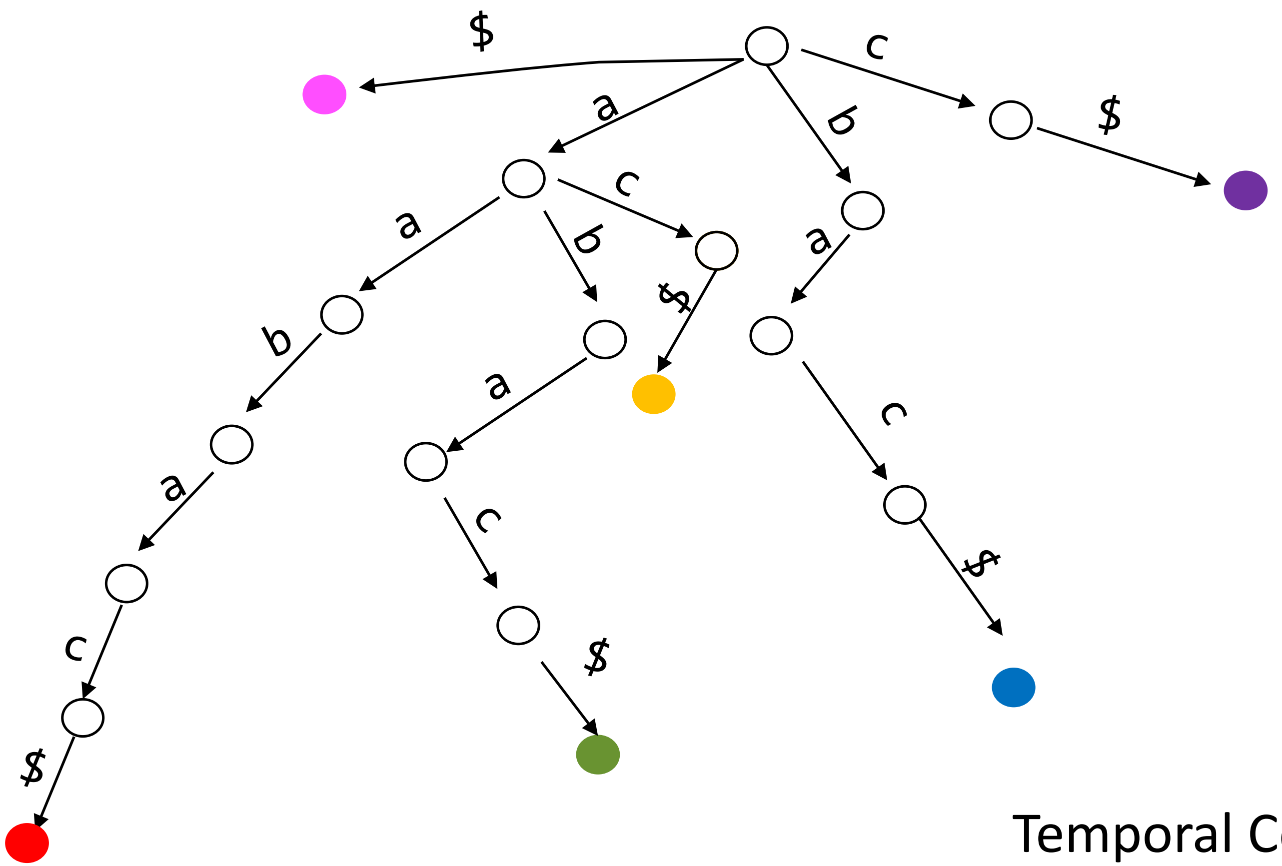
Example suffix trie



T=aabac$
abac$
bac$
ac$
c$
$

There are $\frac{n(n+1)}{2} = \mathcal{O}(n^2)$ characters total in all of those suffixes, + 1 for an “empty” character in the case of $

Example suffix trie



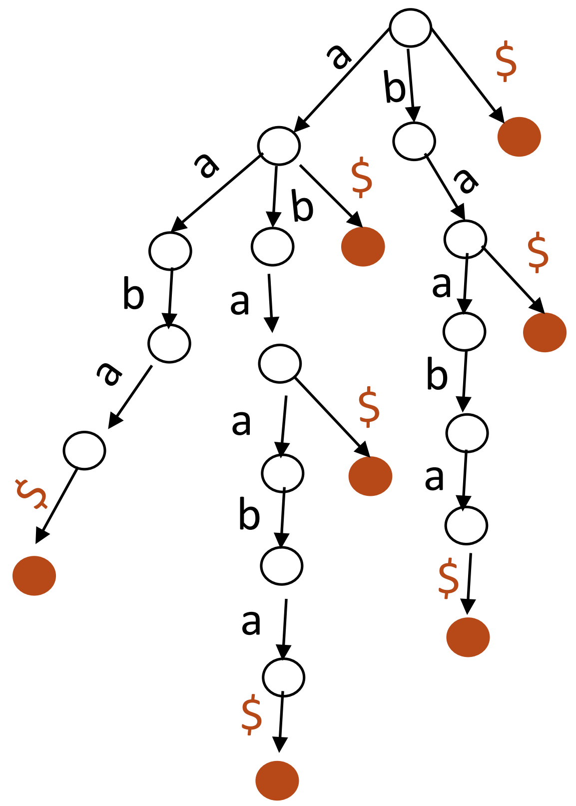
T=aabac$
abac$
bac$
ac$
c$
$

There are $\frac{n(n+1)}{2}$ characters total in all of those suffixes, + 1 for an "empty" character in the case of $

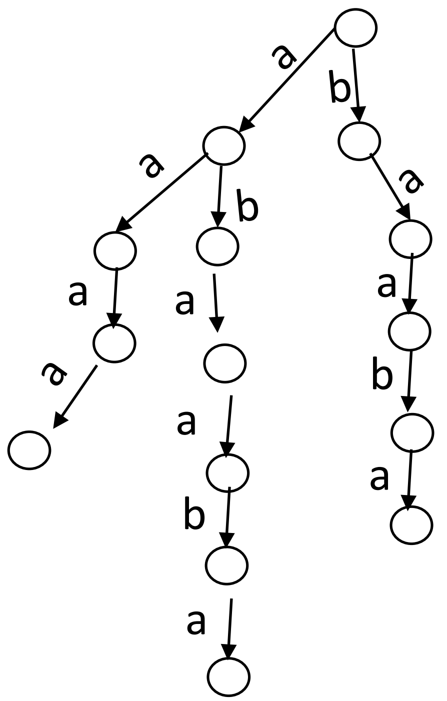
Temporal Cost to build: $\frac{1}{2}n^2 + \frac{1}{2}n + 1 = \mathcal{O}(n^2)$

We also have 19 nodes total, which is significantly bigger than 5.

Importance of $

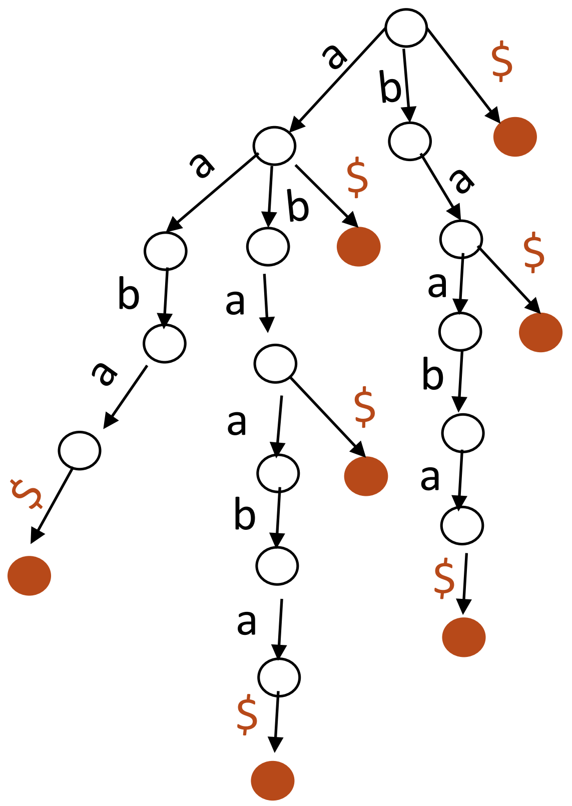


T$=abaaba$



T=abaaba

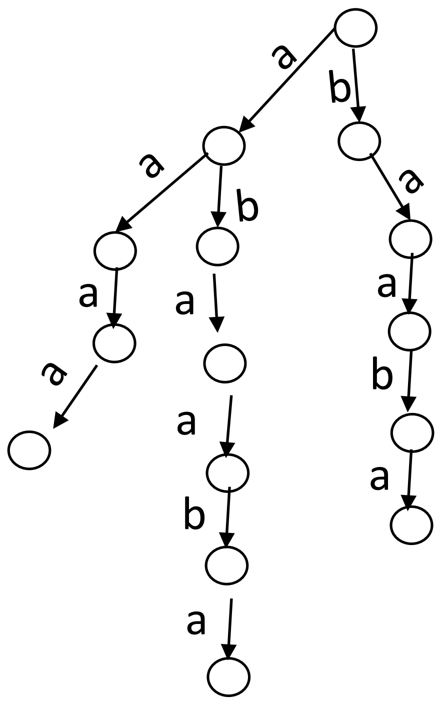
Importance of $



T$=abaaba$

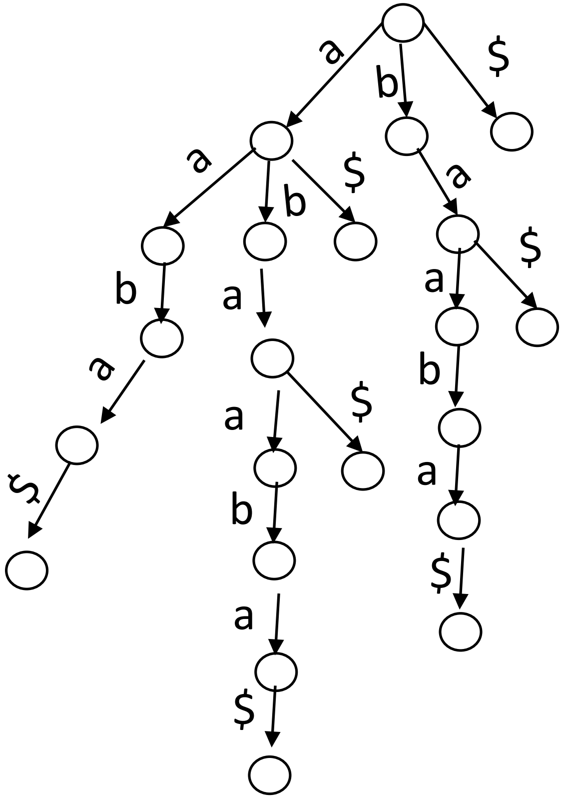
Getting rid of ‘$’ means that we now can no longer “read” the suffixes of the original text from our trie!

So, if somebody asks us if a certain string is a suffix of the original text (important query in DNA sequencing), **we cannot answer!**



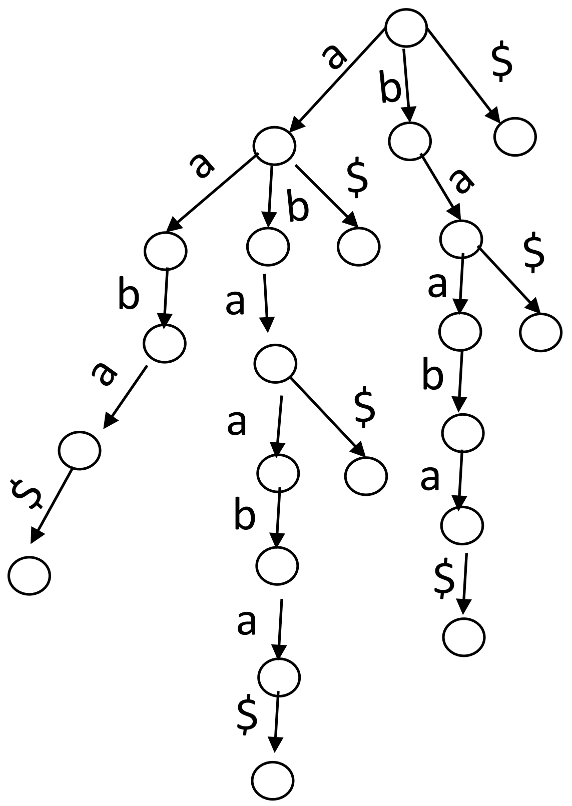
T=abaaba

Asks, you say?



How can we use a suffix trie to….

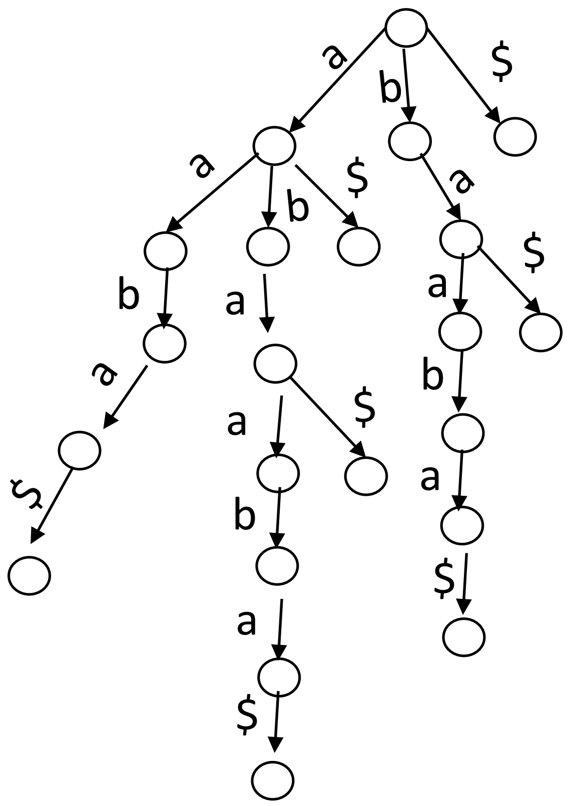
Asks, you say?



How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?

Asks, you say?

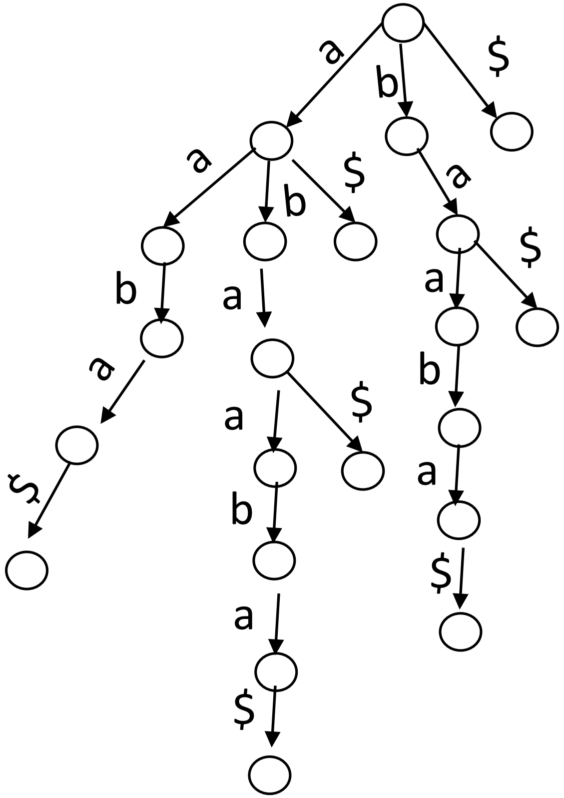


How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?

*Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*

Asks, you say?

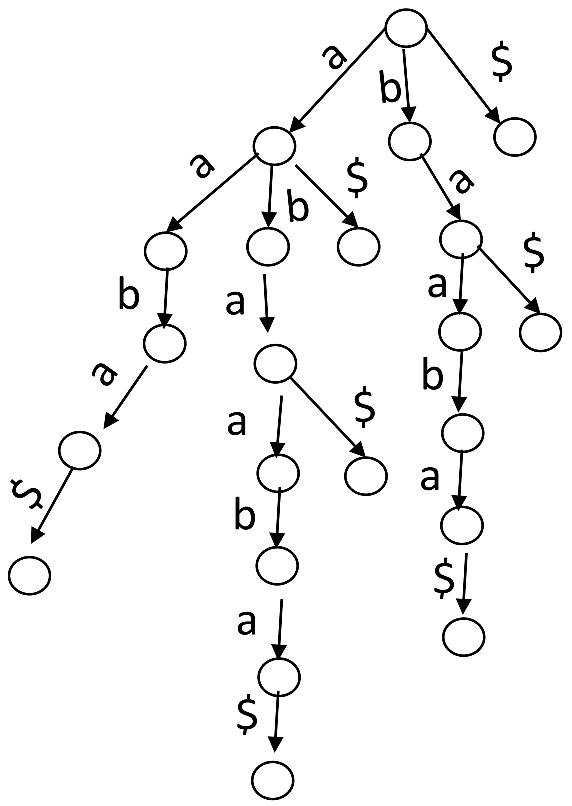


How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?
*Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*

b) Check if a string S is a ***suffix*** of the text T?

Asks, you say?



How can we use a suffix trie to….

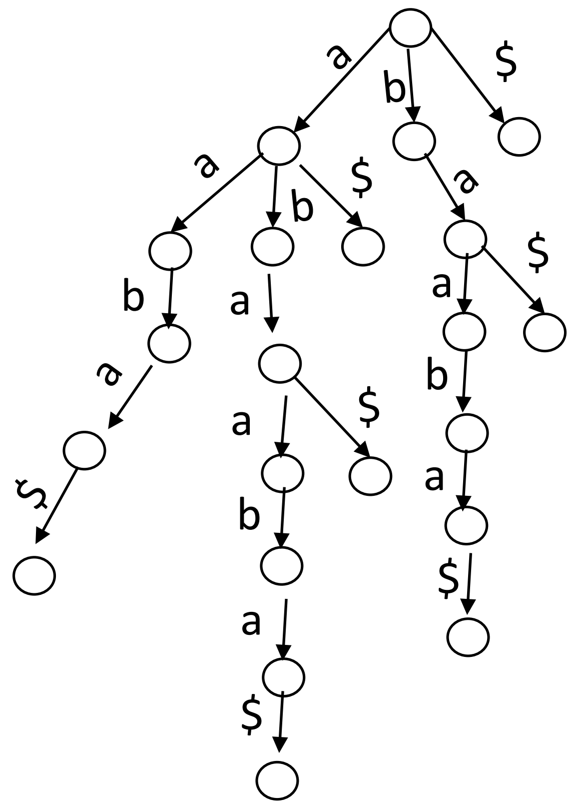
a) Check if a string S is a ***substring*** of the text T?

*Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*

b) Check if a string S is a ***suffix*** of the text T?

Similar as (a), except for the fact that the node you need to arrive at to answer "Yes" has to have an outgoing edge reached by '$'.

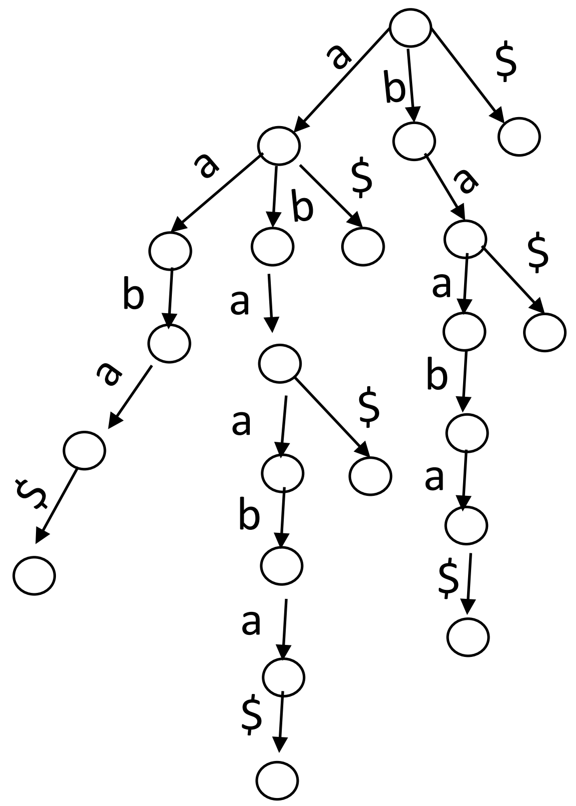
Asks, you say?



How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?
   *Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*
b) Check if a string S is a ***suffix*** of the text T?
   Similar as (a), except for the fact that the node you need to arrive at to answer "Yes" has to have an outgoing edge reached by '$'.
c) Count the ***number of times*** a string S occurs as a substring T?

Asks, you say?



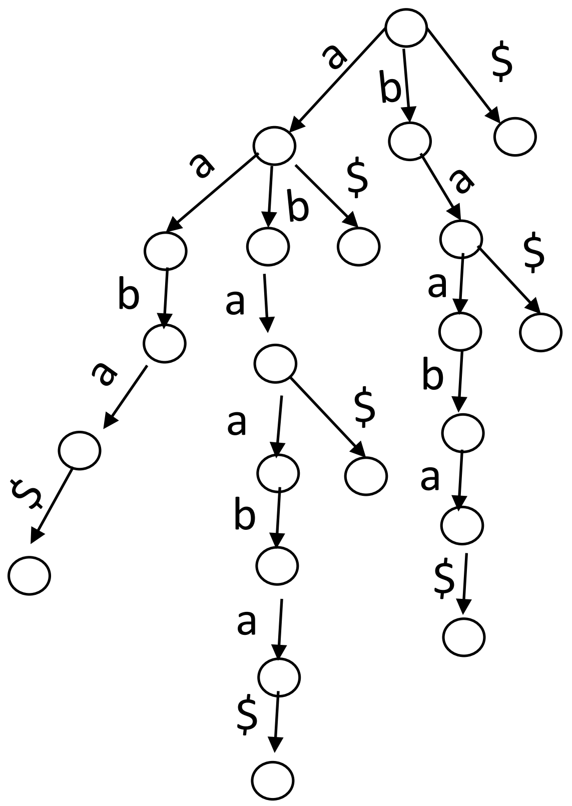
How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?
*Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*

b) Check if a string S is a ***suffix*** of the text T?
Similar as (a), except for the fact that the node you need to arrive at to answer "Yes" has to have an outgoing edge reached by '$'.

c) Count the ***number of times*** a string S occurs as a substring T?
*Traverse the trie as in (a), hopefully reaching a node (otherwise anwer is 0), and then count the **number of leaf nodes of the subtrie rooted at that node** (DFS).*

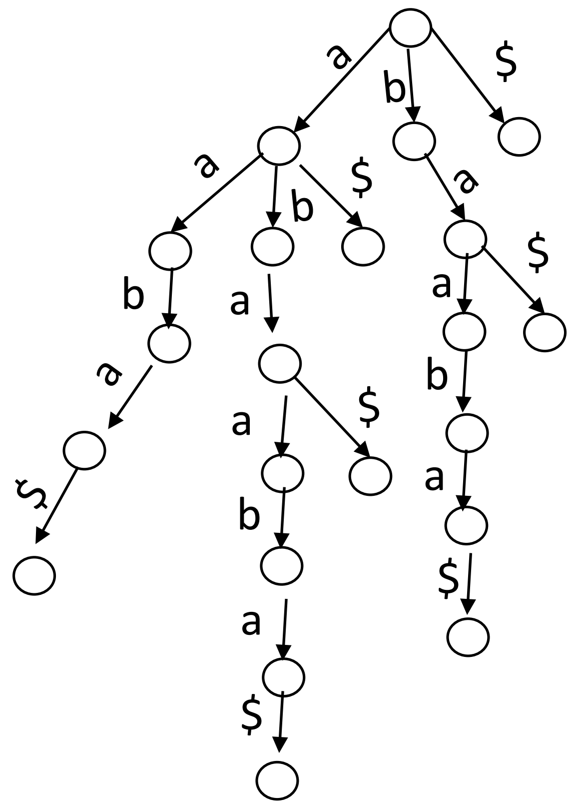
Asks, you say?



How can we use a suffix trie to….

a) Check if a string S is a ***substring*** of the text T?
   *Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*
b) Check if a string S is a ***suffix*** of the text T?
   Similar as (a), except for the fact that the node you need to arrive at to answer "Yes" has to have an outgoing edge reached by '$'.
c) Count the ***number of times*** a string S occurs as a substring T?
   *Traverse the trie as in (a), hopefully reaching a node (otherwise anwer is 0), and then count the **number of leaf nodes of the subtrie rooted at that node** (DFS).*
d) Find the ***longest repeated substring*** of T?

Asks, you say?



How can we use a suffix trie to….

a) Check if a string S is a ***<u>substring</u>*** of the text T?
*Keep processing the characters of S until you either arrive at some node(S **is** a substring) or fall off the trie (S is **not** a substring).*

b) Check if a string S is a ***<u>suffix</u>*** of the text T?
Similar as (a), except for the fact that the node you need to arrive at to answer "Yes" has to have an outgoing edge reached by '$'.

c) Count the ***<u>number of times</u>*** a string S occurs as a substring T?
*Traverse the trie as in (a), hopefully reaching a node (otherwise anwer is 0), and then count the **<u>number of leaf nodes of the subtrie rooted at that node</u>** (DFS).*

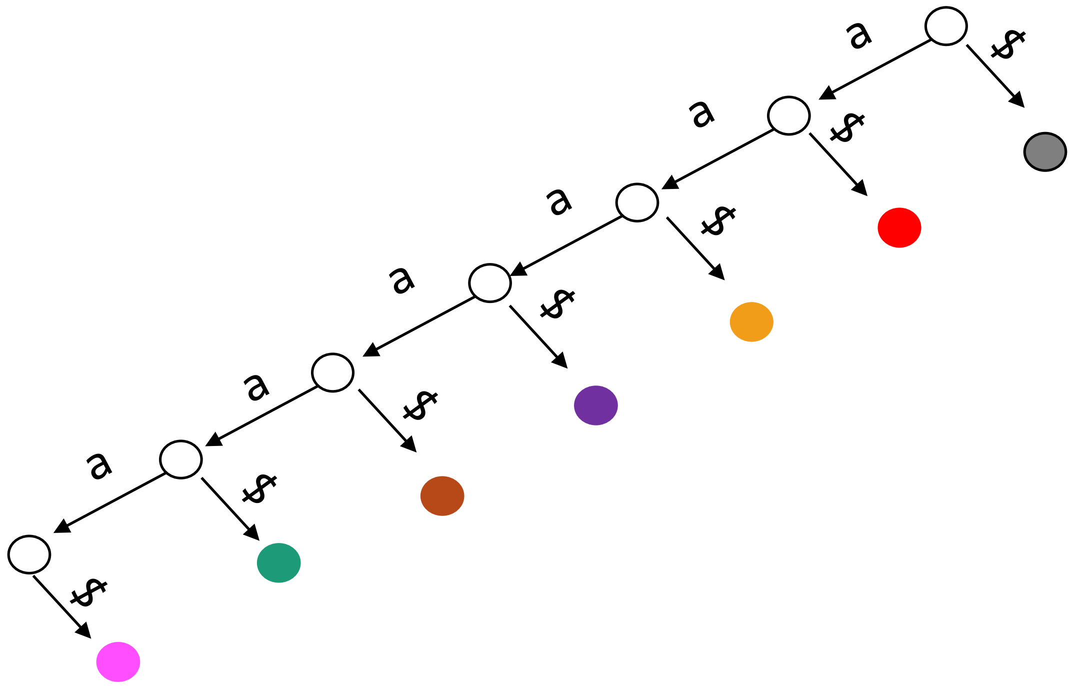
d) Find the ***<u>longest repeated substring</u>*** of T?
Deepest node with at least 2 children.

Give me the trie!

- Please give us the suffix trie for the augmented text T$=aaaaaa$!

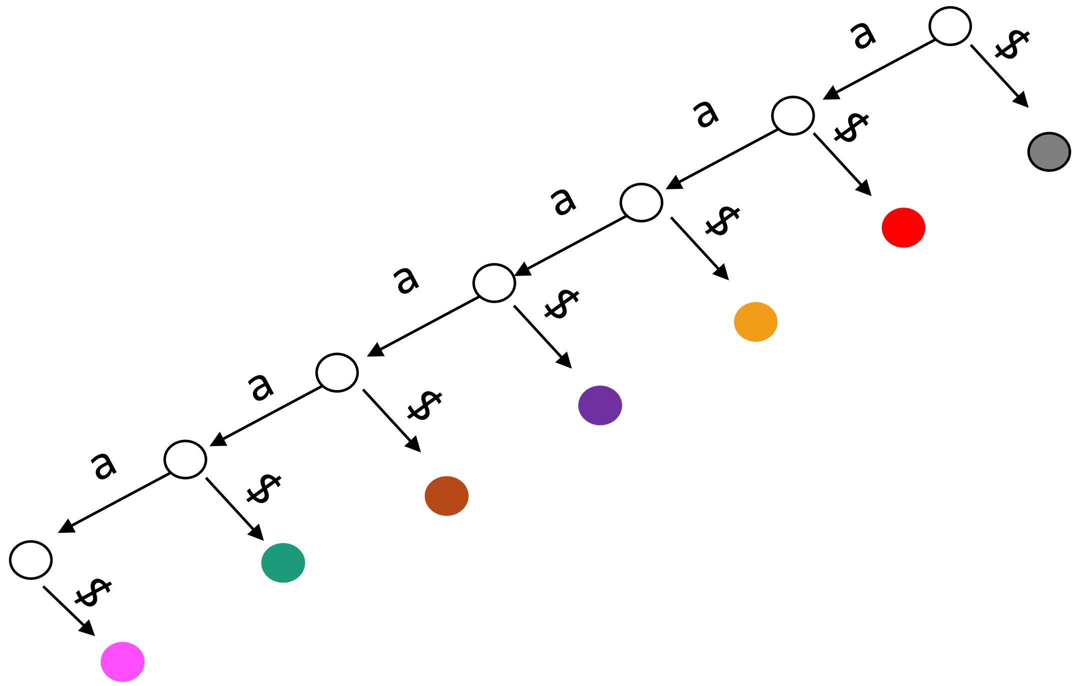
Give me the trie!

T=aaaaaa$
aaaaa$
aaaa$
aaa$
aa$
a$
$



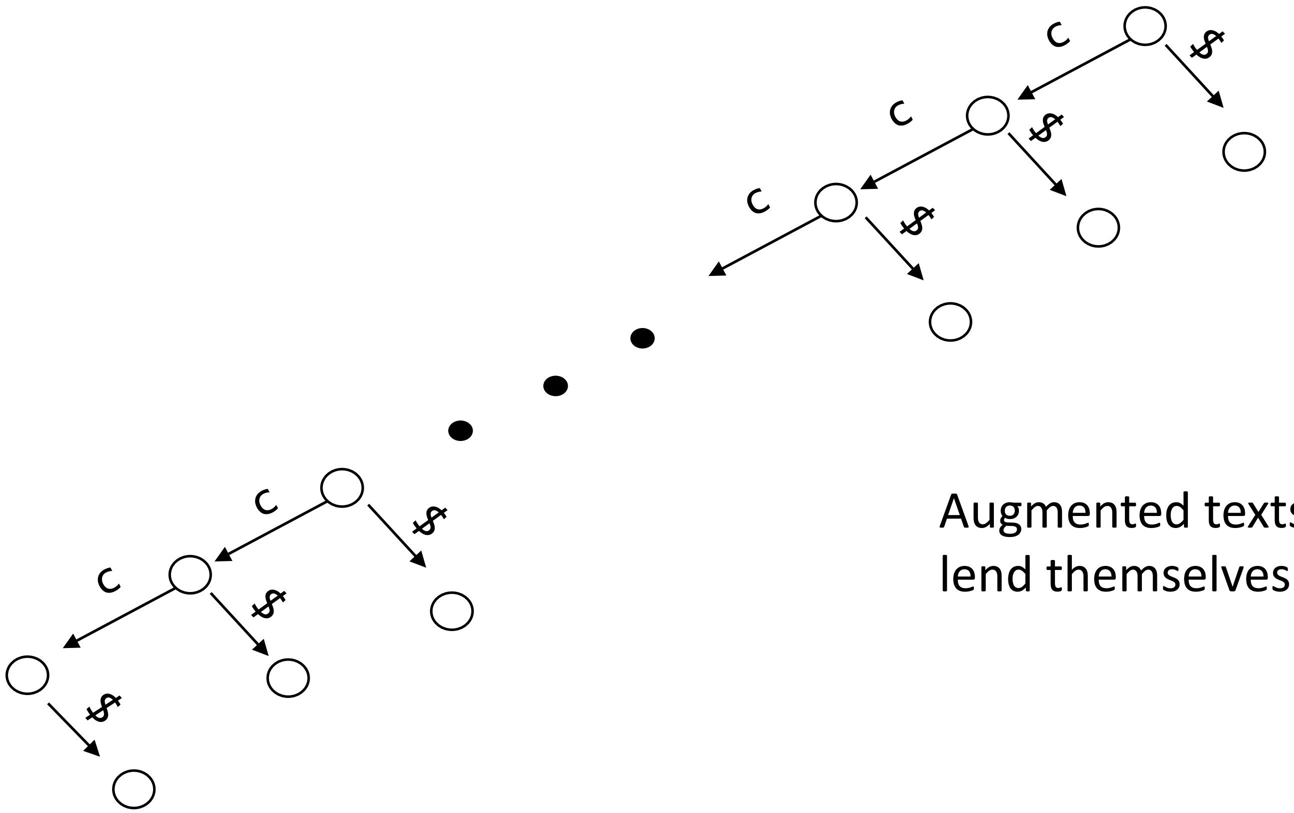
Give me the trie!

T=aaaaaa$
aaaaa$
aaaa$
aaa$
aa$
a$
$



m=6
#nodes=14=2*6 + 2

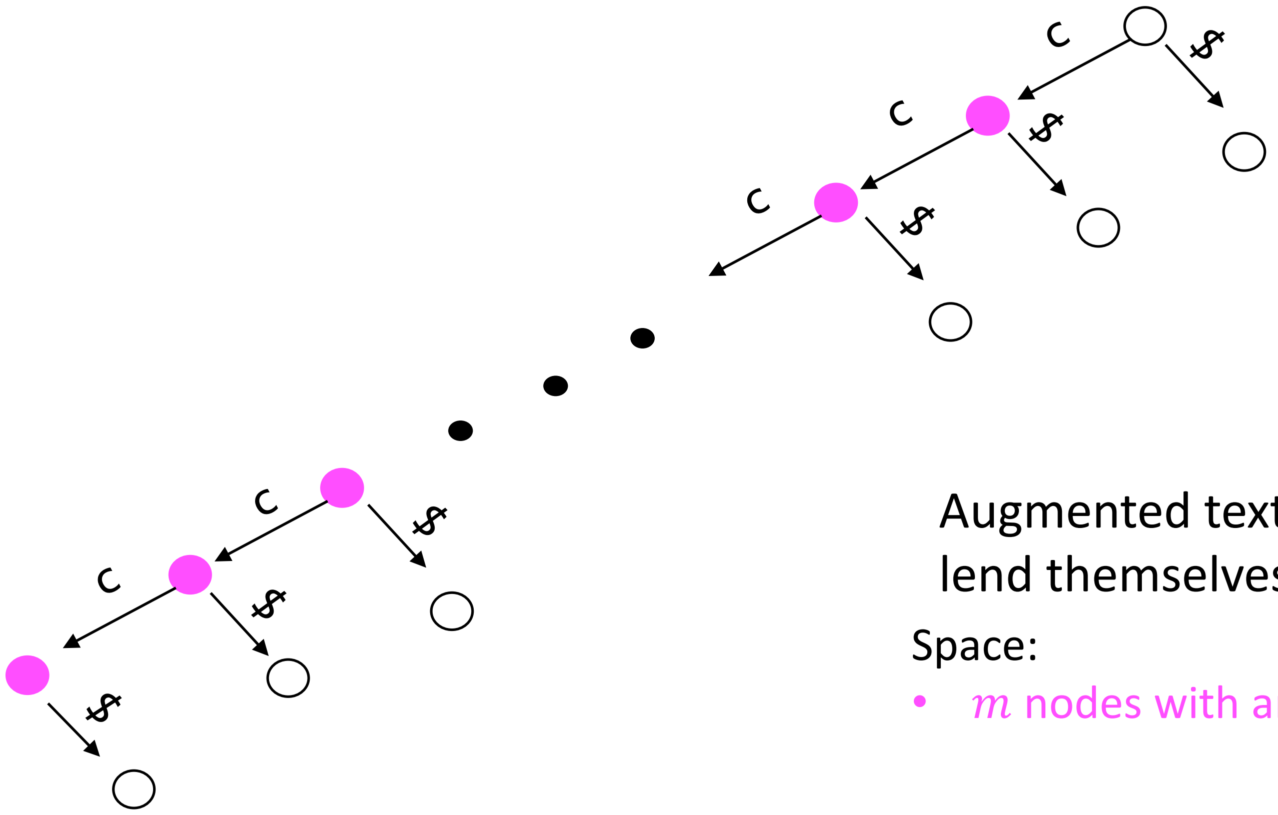
A good family of strings



m=6
#nodes=14

Augmented texts of form $c^m\$$ for a given character c lend themselves to **linear-space suffix tries**!

A good family of strings



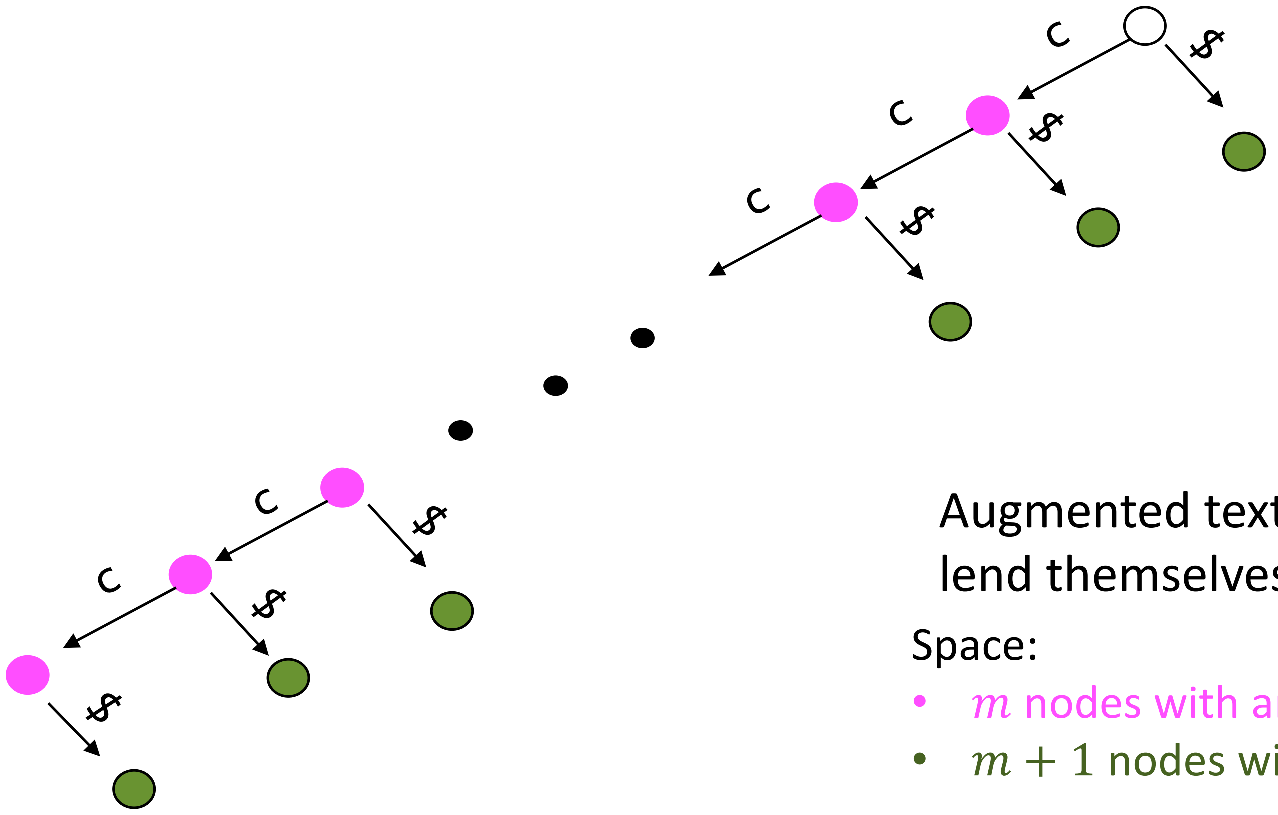
m=6
#nodes=14

Augmented texts of form $c^m\$$ for a given character c lend themselves to **linear-space suffix tries**!

Space:

- m nodes with an incoming edge labeled with c.

A good family of strings



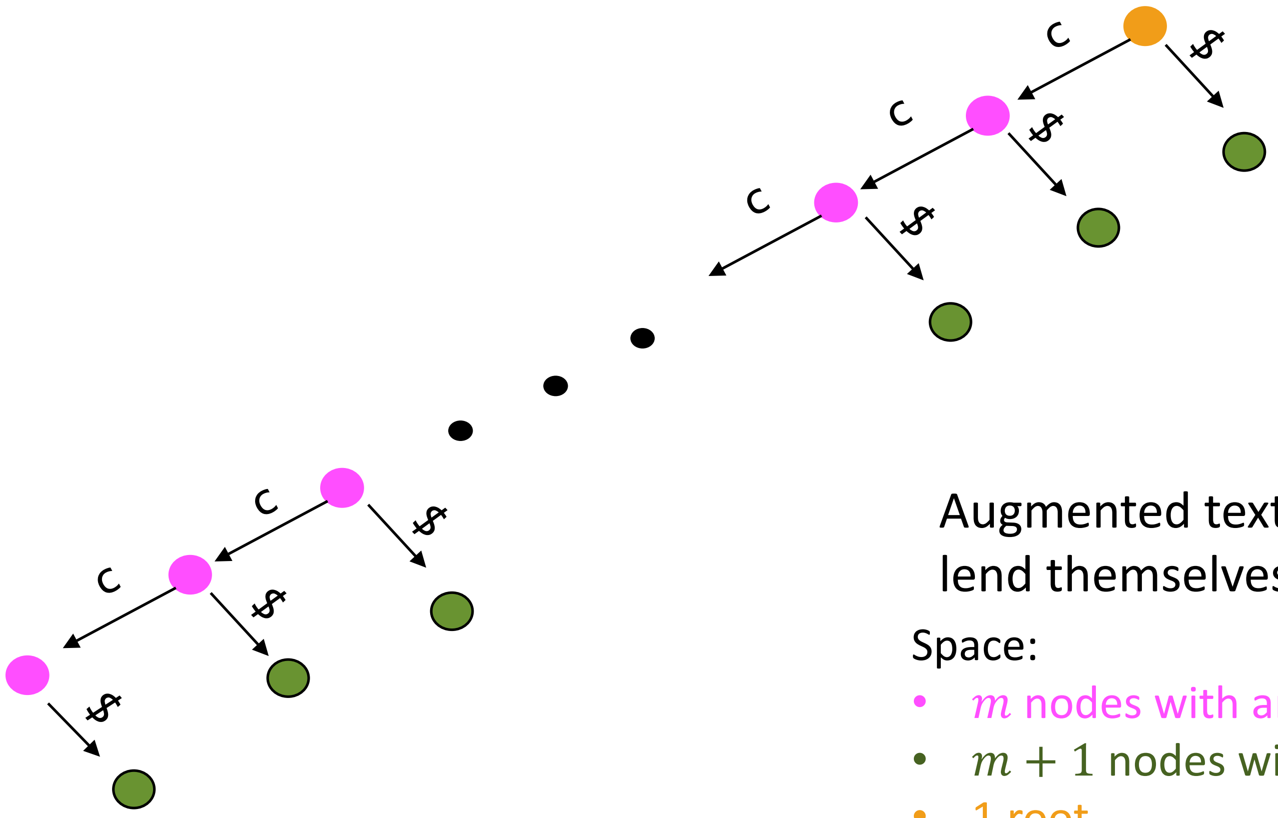
m=6
#nodes=14

Augmented texts of form $c^m\$$ for a given character c lend themselves to **linear-space suffix tries**!

Space:

- m nodes with an incoming edge labeled with c.
- $m+1$ nodes with an incoming edge labeled with $\$$.

A good family of strings



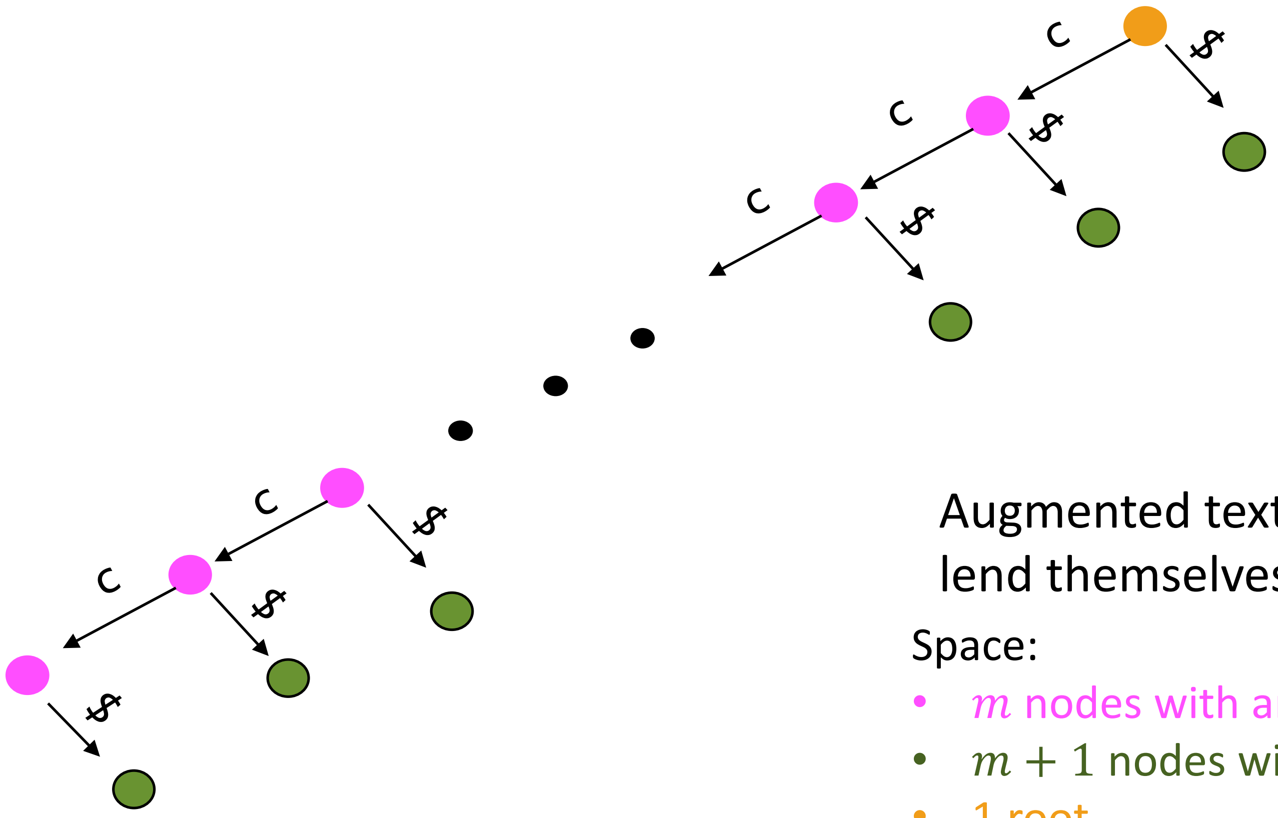
m=6
#nodes=14

Augmented texts of form $c^m\$$ for a given character c lend themselves to **linear-space suffix tries**!

Space:

- m nodes with an incoming edge labeled with c.
- $m + 1$ nodes with an incoming edge labeled with $\$$.
- 1 root

A good family of strings



m=6
#nodes=14

Augmented texts of form $c^m\$$ for a given character c lend themselves to **linear-space suffix tries**!

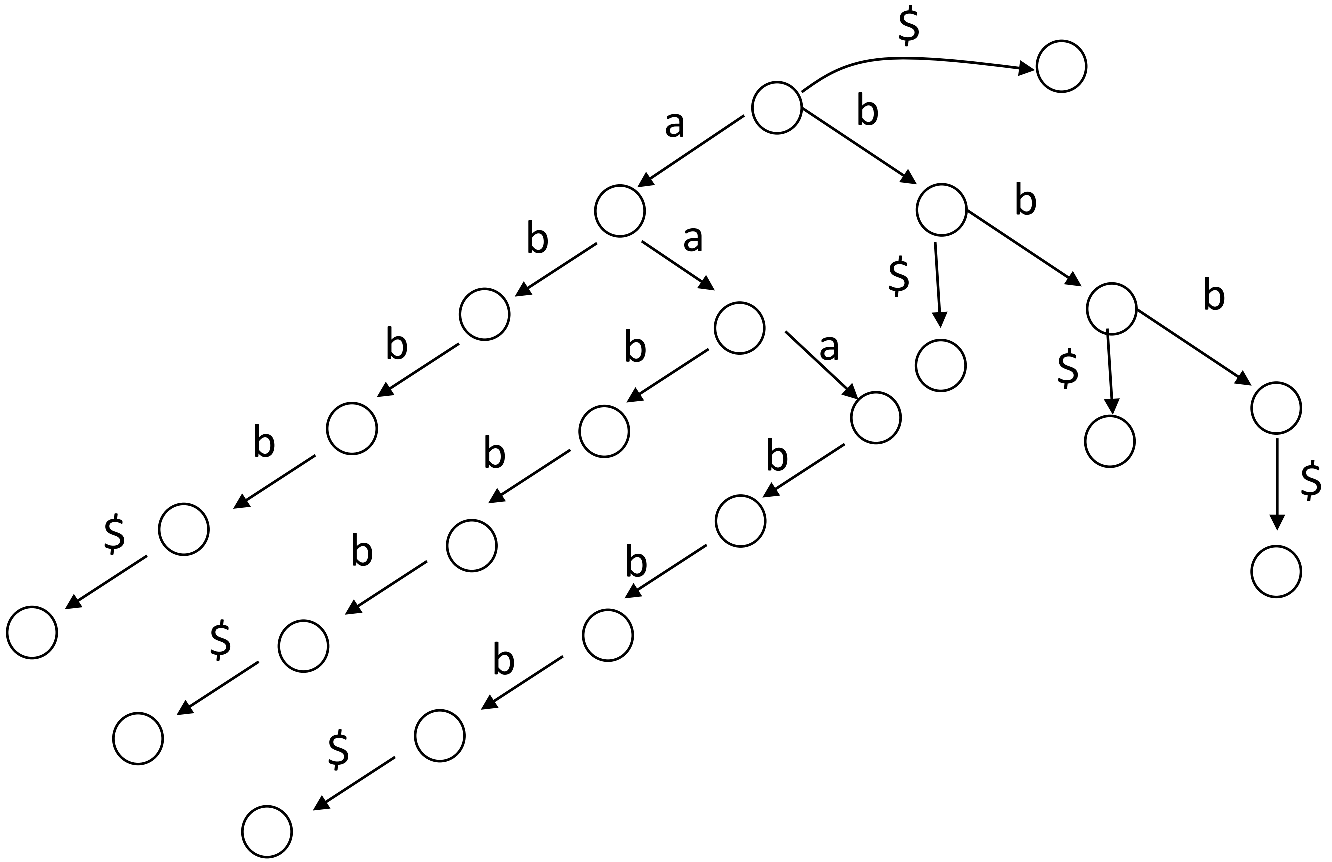
Space:

- m nodes with an incoming edge labeled with c.
- $m + 1$ nodes with an incoming edge labeled with $\$$.
- 1 root
- In total: $2m + 2 = O(m)$ space!

Give me the trie! Part 2

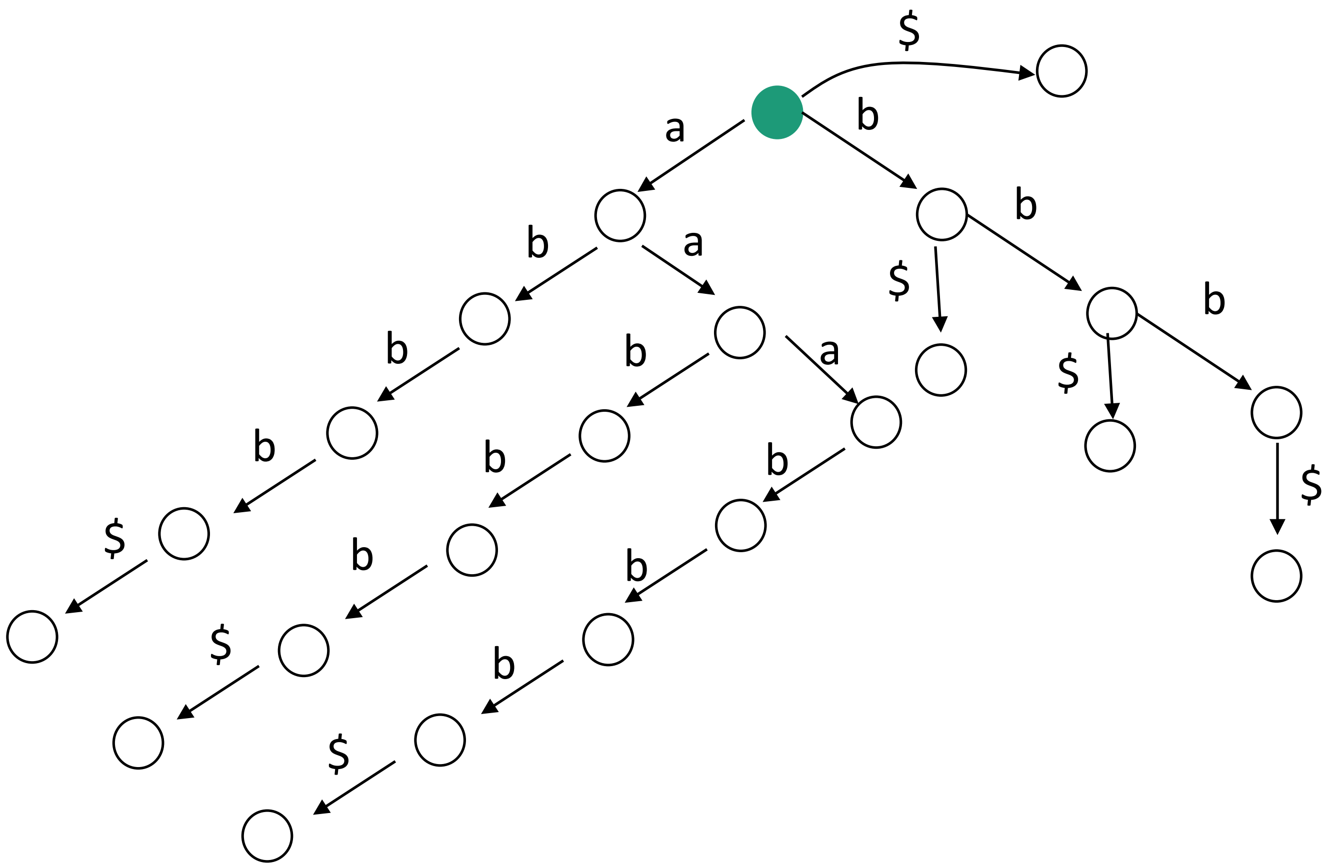
- Please give us the suffix trie for the augmented text T$=aaabbb$!

$T\$ = aaabbb\$ = a^3b^3\$$ Give me the trie! Part 2



$T\$ = aaabbb\$ = a^{3}b^{3}\$$

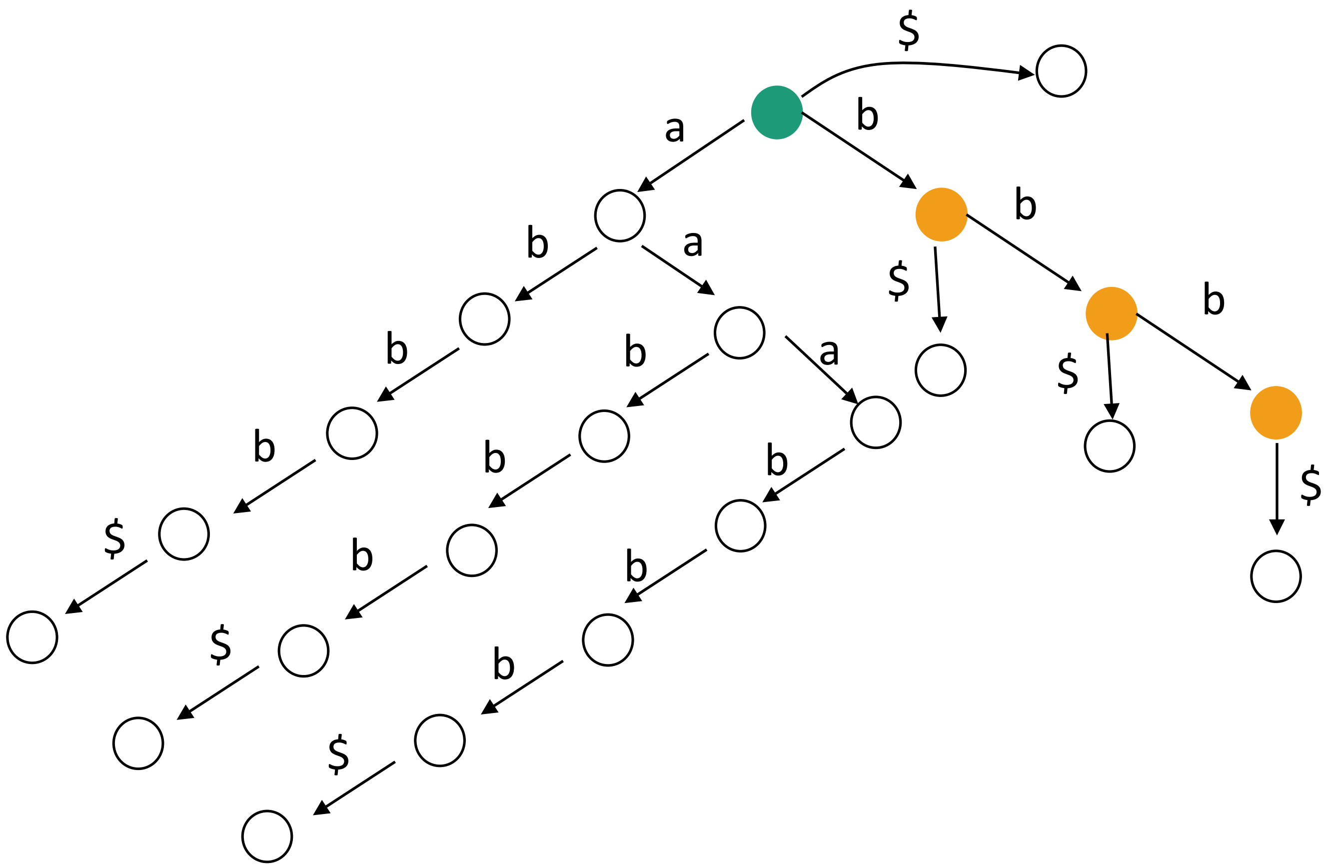
Give me the trie! Part 2



1 root +

$T\$ = aaabbb\$ = a^{3}b^{3}\$$

Give me the trie! Part 2

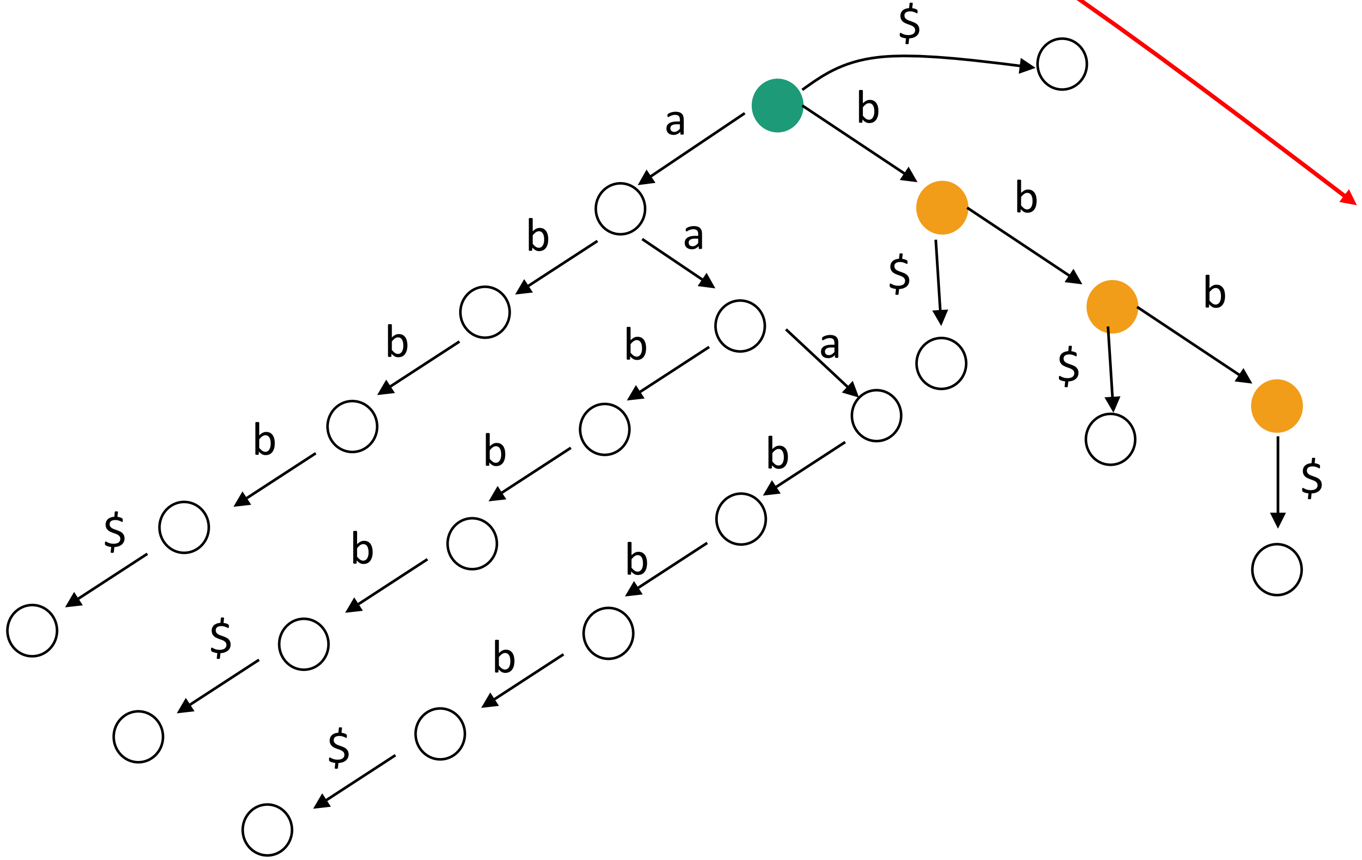


1 root +

3 nodes along ‘b’ chain +

$T\$ = aaabbb\$ = a^3 b^3 \$$

Give me the trie! Part 2

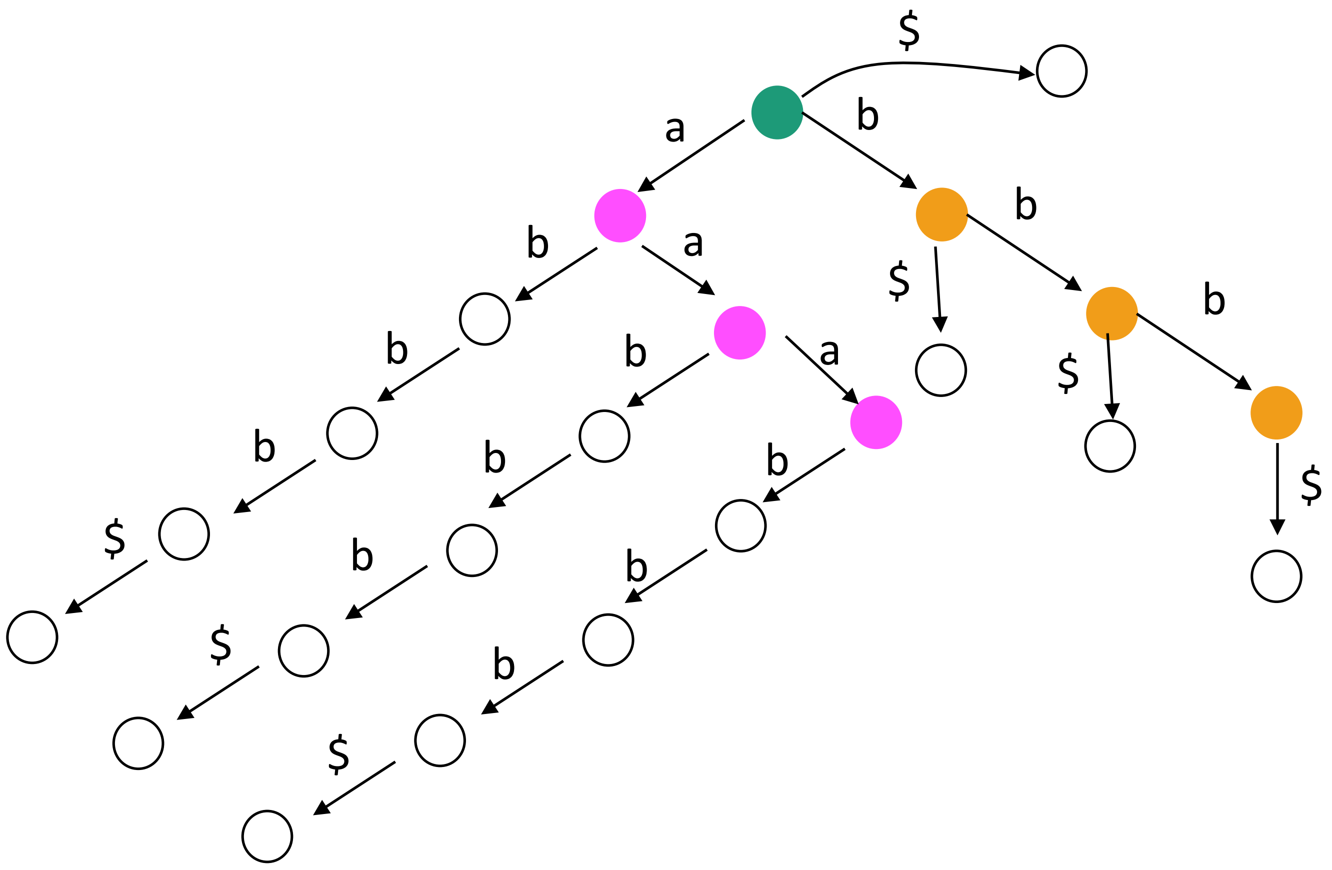


1 root +

3 nodes along ‘b’ chain +

$T\$ = aaabbb\$ = a^{3}b^{3}\$$

Give me the trie! Part 2



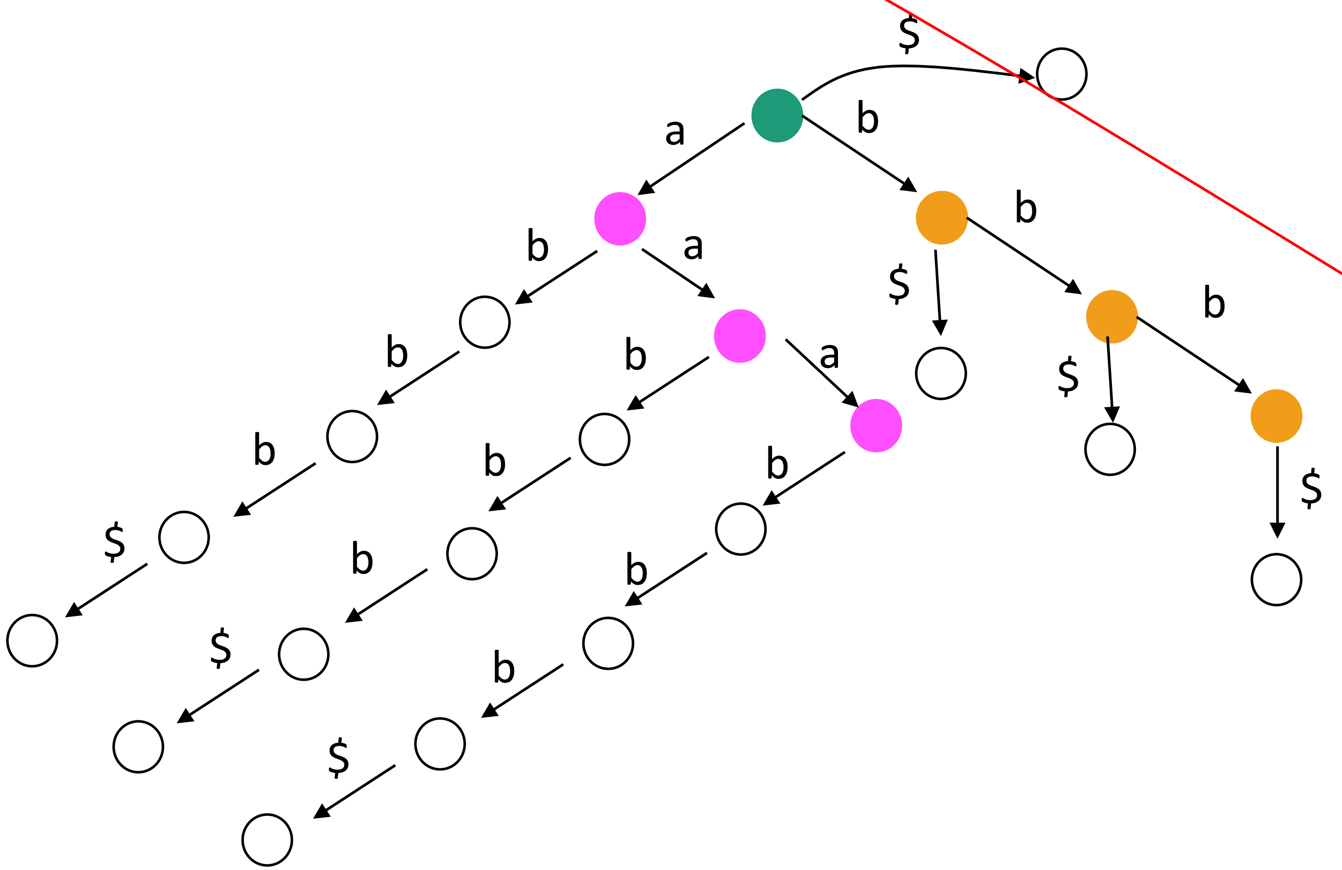
1 root +

3 nodes along ‘b’ chain +

3 nodes along ‘a’ chain +

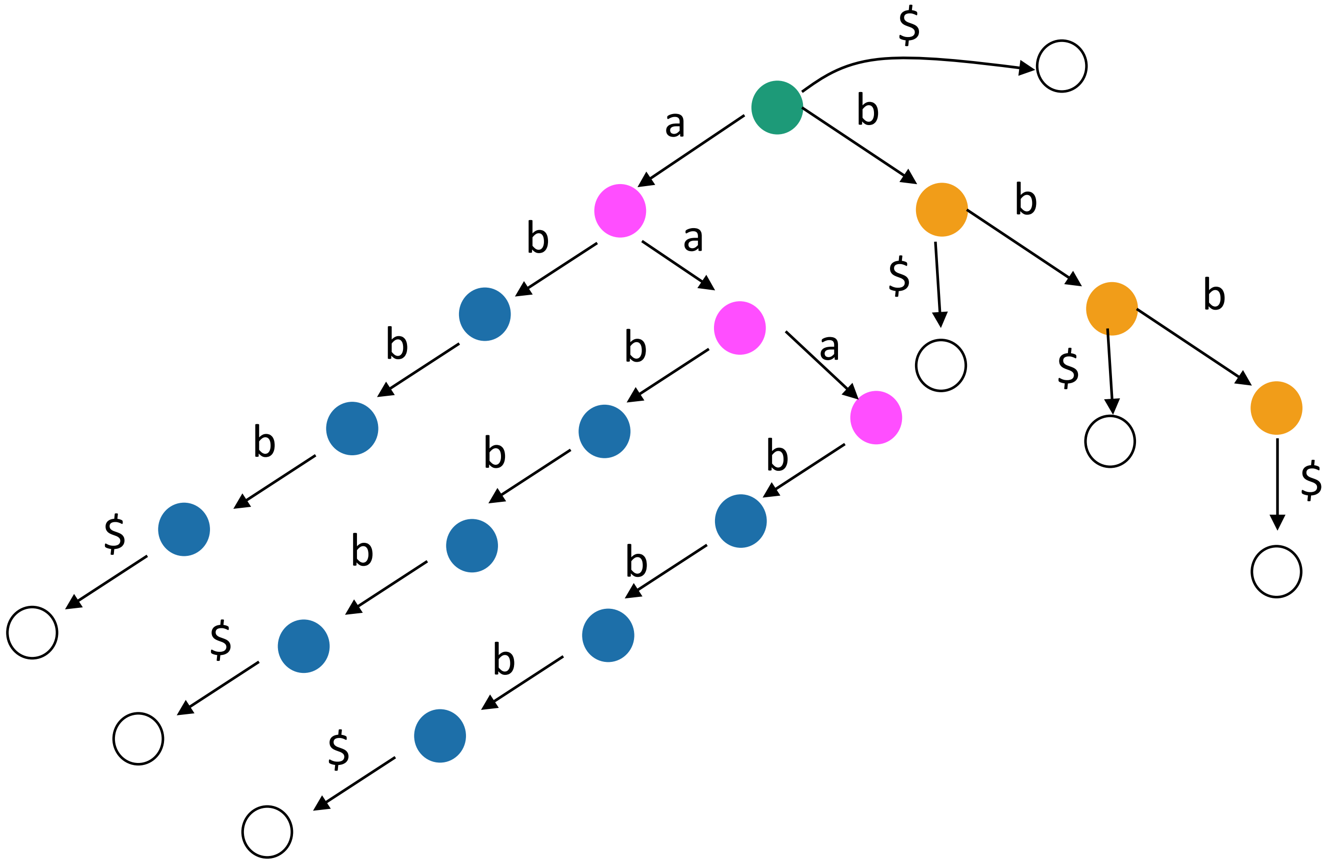
$T\$ = aaabbb\$ = a^{3}b^{3}\$$

Give me the trie! Part 2



1 root +
3 nodes along ‘b’ chain +
3 nodes along ‘a’ chain +

$T\$ = aaabbb\$ = a^{3}b^{3}\$$ Give me the trie! Part 2



1 root +

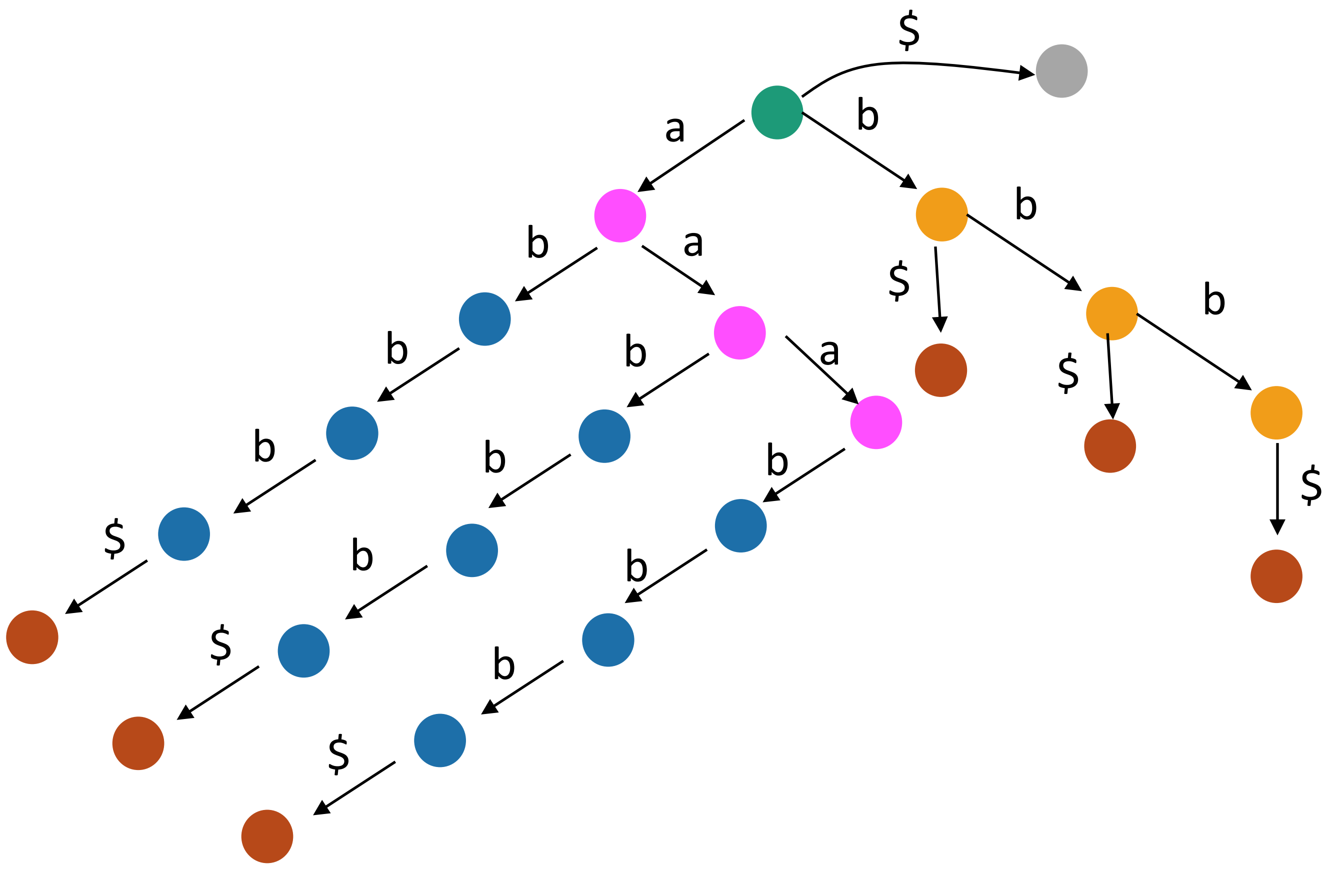
3 nodes along ‘b’ chain +

3 nodes along ‘a’ chain +

3 chains of 3 ‘b’ nodes each +

$T\$ = aaabbb\$ = a^3b^3\$$

Give me the trie! Part 2



1 root +

3 nodes along ‘b’ chain +

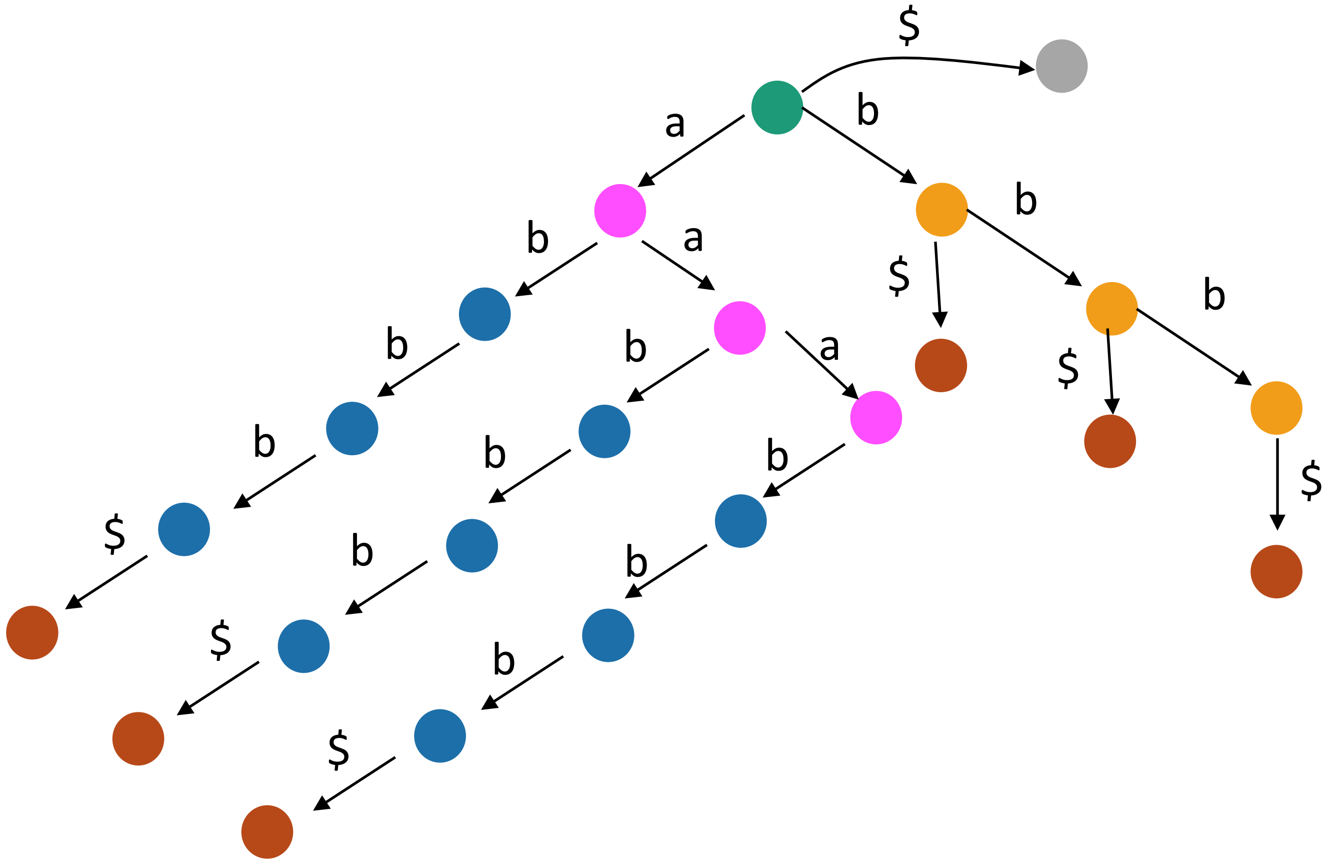
3 nodes along ‘a’ chain +

3 chains of 3 ‘b’ nodes each +

$\underbrace{2 * 3 + 1}_{\text{suffix } \$}$ leaves

$T\$ = aaabbb\$ = a^3 b^3 \$$

Give me the trie! Part 2



1 root +

3 nodes along ‘b’ chain +

3 nodes along ‘a’ chain +

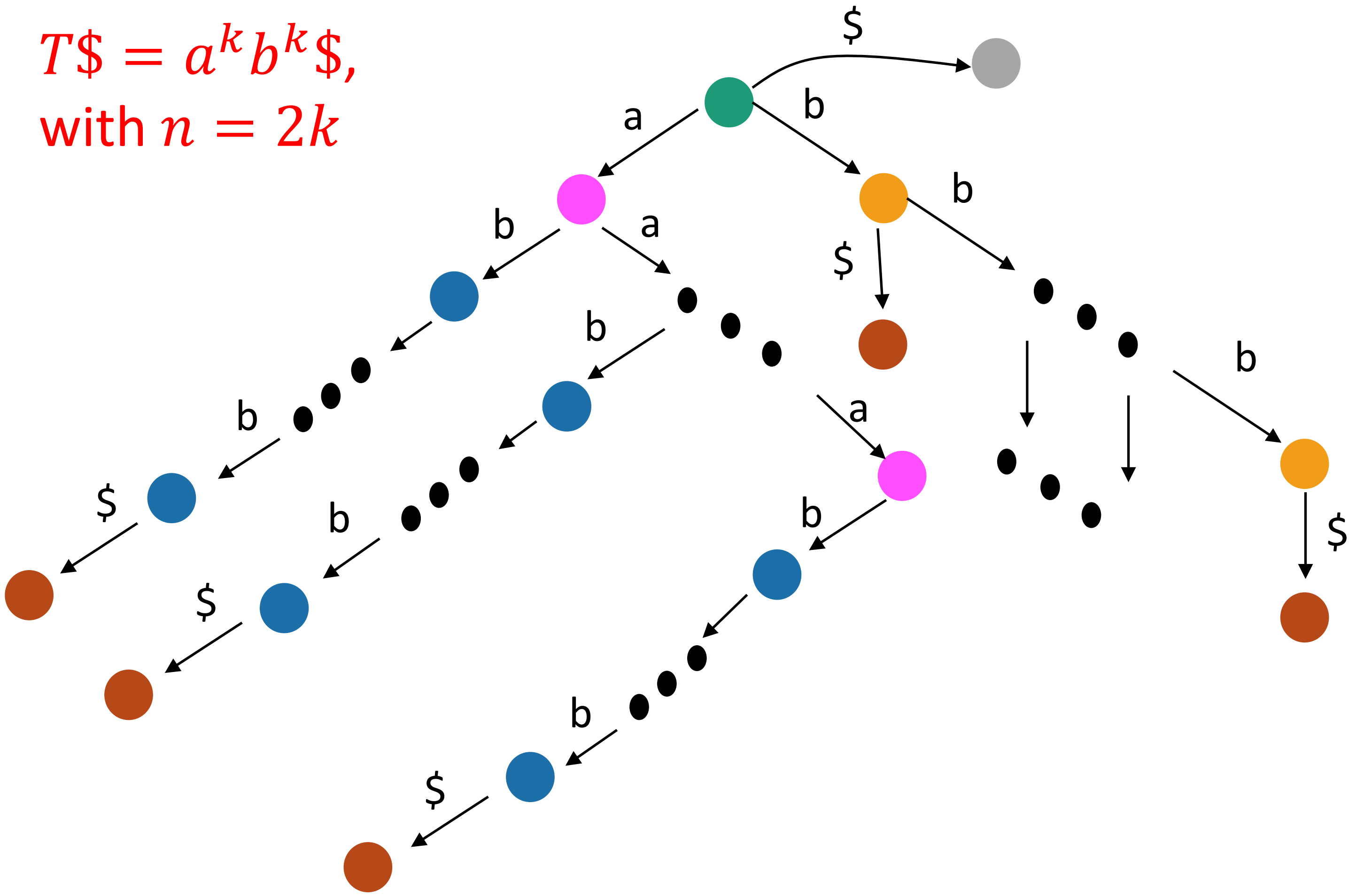
3 chains of 3 ‘b’ nodes each +

$2 * 3 + \underbrace{1}_{\text{suffix \$}}$ leaves

= 23 nodes, alarmingly close to $\frac{1}{2}m^2 + \frac{1}{2}m = 21$

A bad family of strings

$T\$ = a^k b^k \$,$
with $n = 2k$

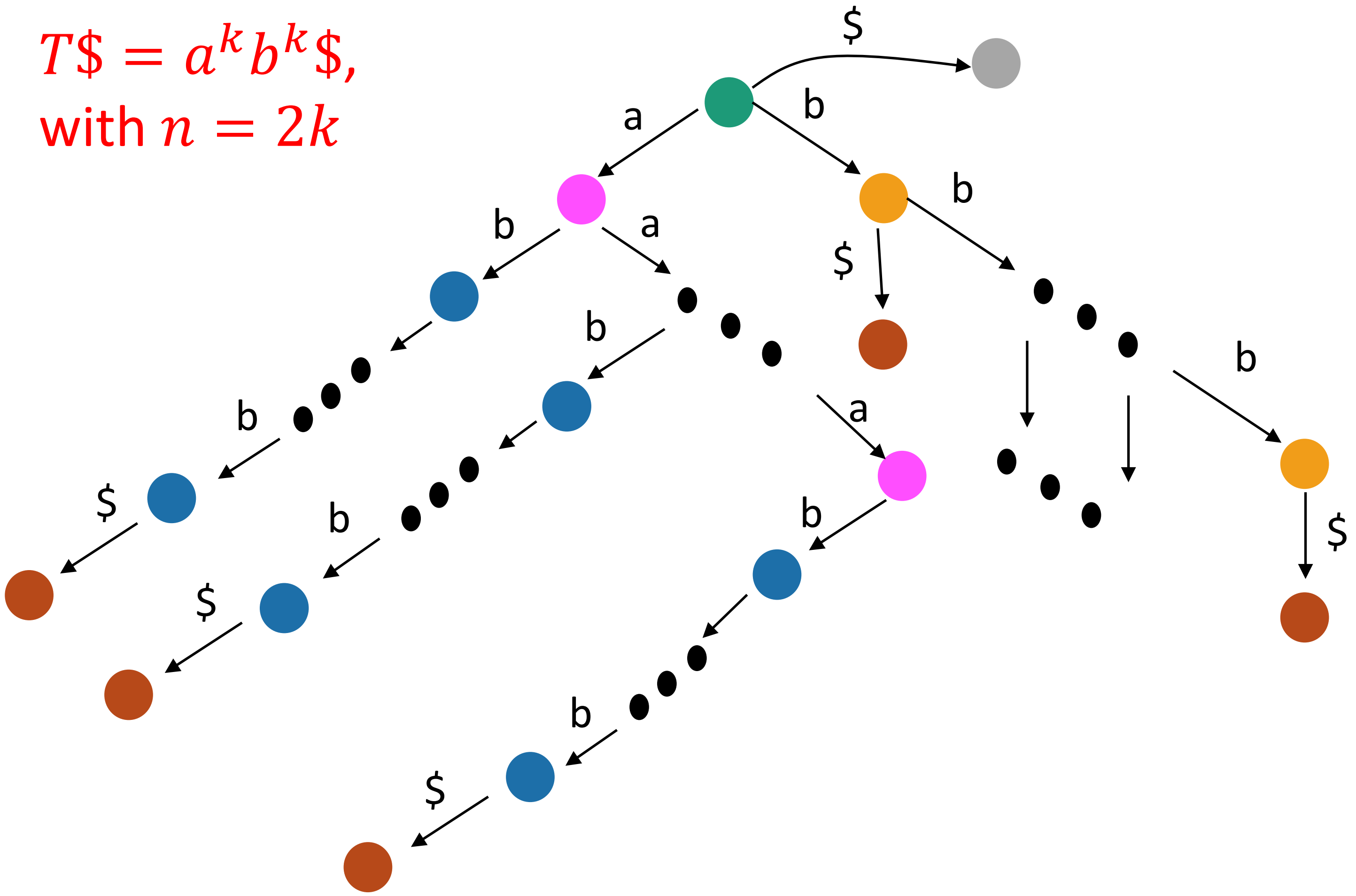


1 root +
k nodes along ‘b’ chain +
k nodes along ‘a’ chain +
k chains of k ‘b’ nodes each +
$2 * k + \underbrace{1}_{\text{suffix \$}}$ leaves

$$= k^2 + 4k + 2$$
$$= \left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{2}\right) + 2$$
$$= \frac{1}{4}n^2 + 2n + 2$$

A bad family of strings

$T\$ = a^k b^k \$,$
with $n = 2k$

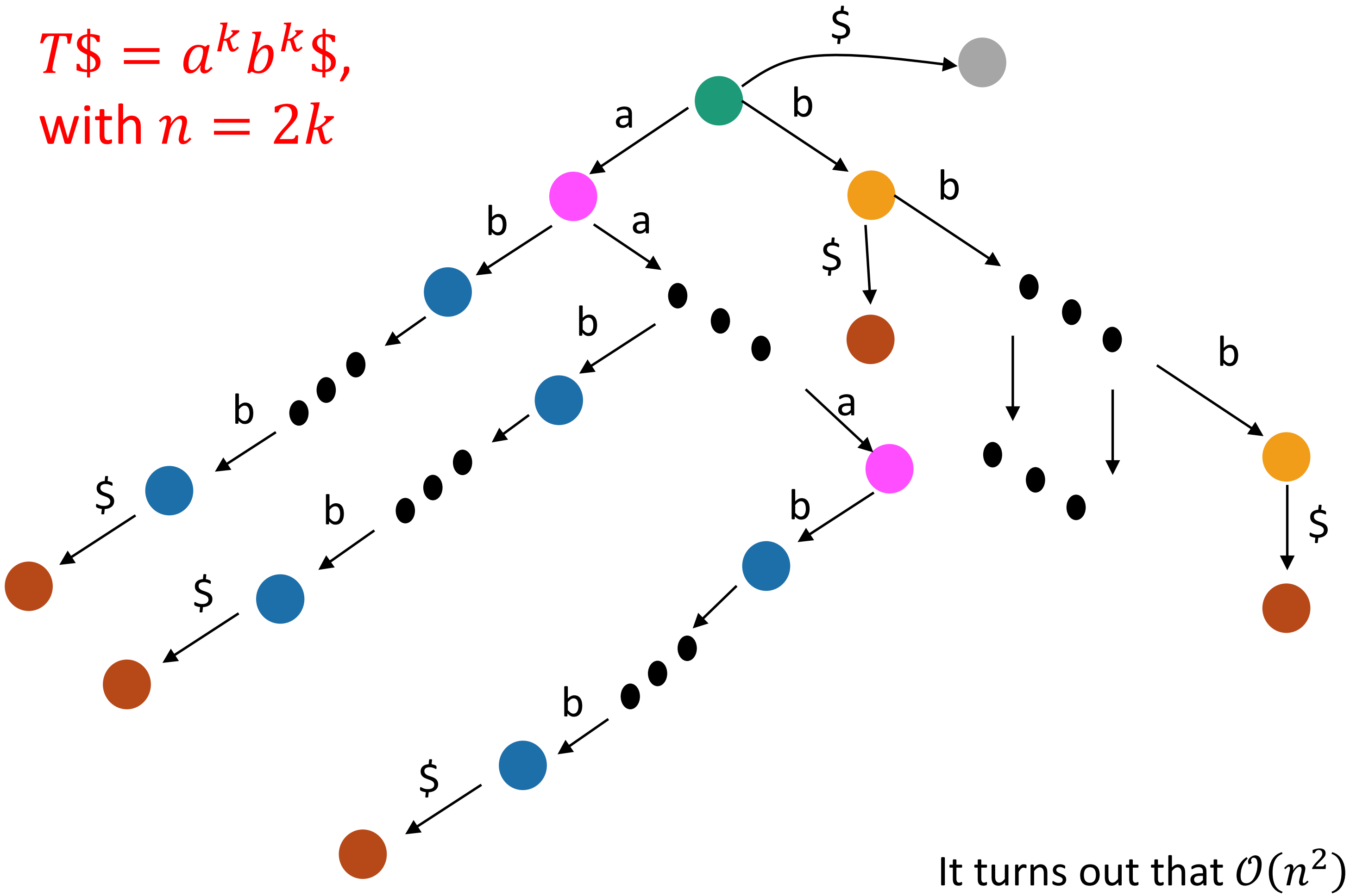


1 root +
k nodes along 'b' chain +
k nodes along 'a' chain +
k chains of k 'b' nodes each +
$2 * k + \underbrace{1}_{\text{suffix } \$}$ leaves

$$= k^2 + 4k + 2$$
$$= \left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{2}\right) + 2$$
$$= \frac{1}{4}n^2 + 2n + 2$$
$$= \mathcal{O}(n^2)$$

A bad family of strings

$T\$ = a^k b^k \$,$
with $n = 2k$



1 root +
k nodes along ‘b’ chain +
k nodes along ‘a’ chain +
k chains of k ‘b’ nodes each +
$2 * k + \underbrace{1}_{\text{suffix }\$}$ leaves

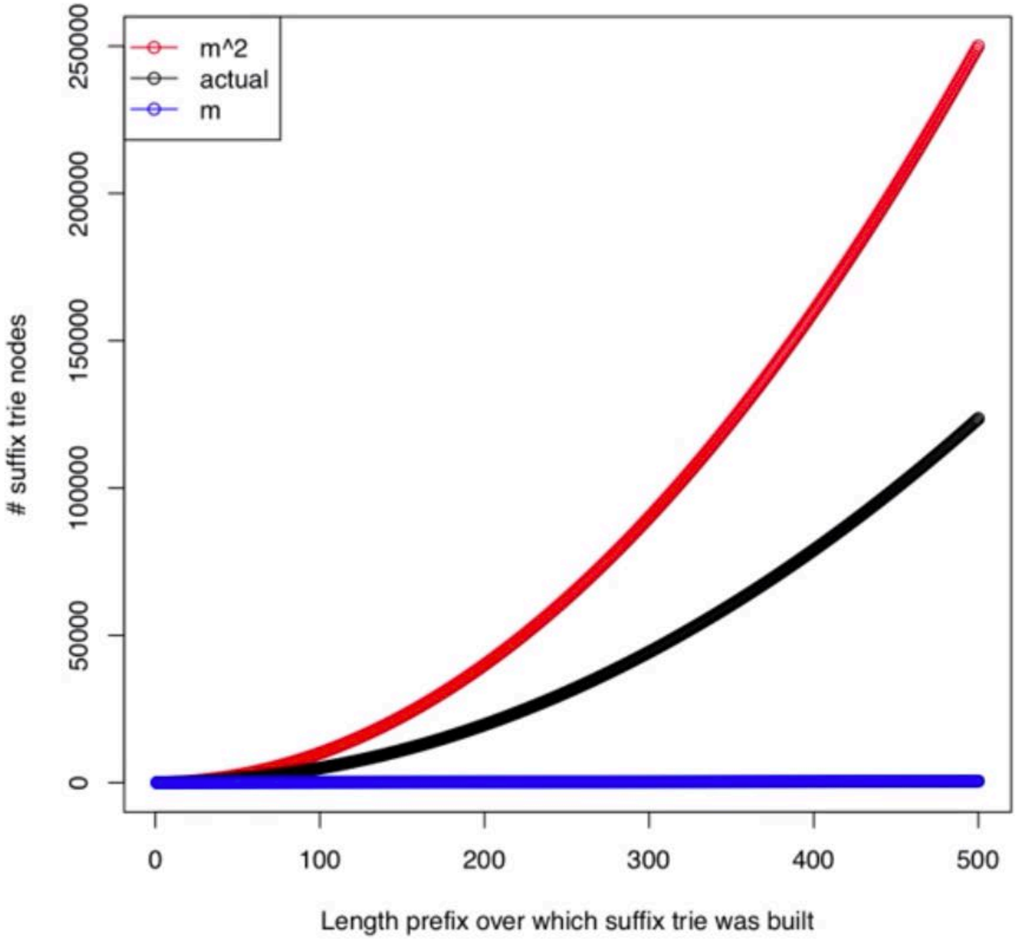
$$= k^2 + 4k + 2$$
$$= \left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{2}\right) + 2$$
$$= \frac{1}{4}n^2 + 2n + 2$$
$$= \mathcal{O}(n^2)$$

It turns out that $\mathcal{O}(n^2)$ is also **the worst** that we can do in terms of spatial complexity of a given suffix trie.

Spatial cost in practice

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length



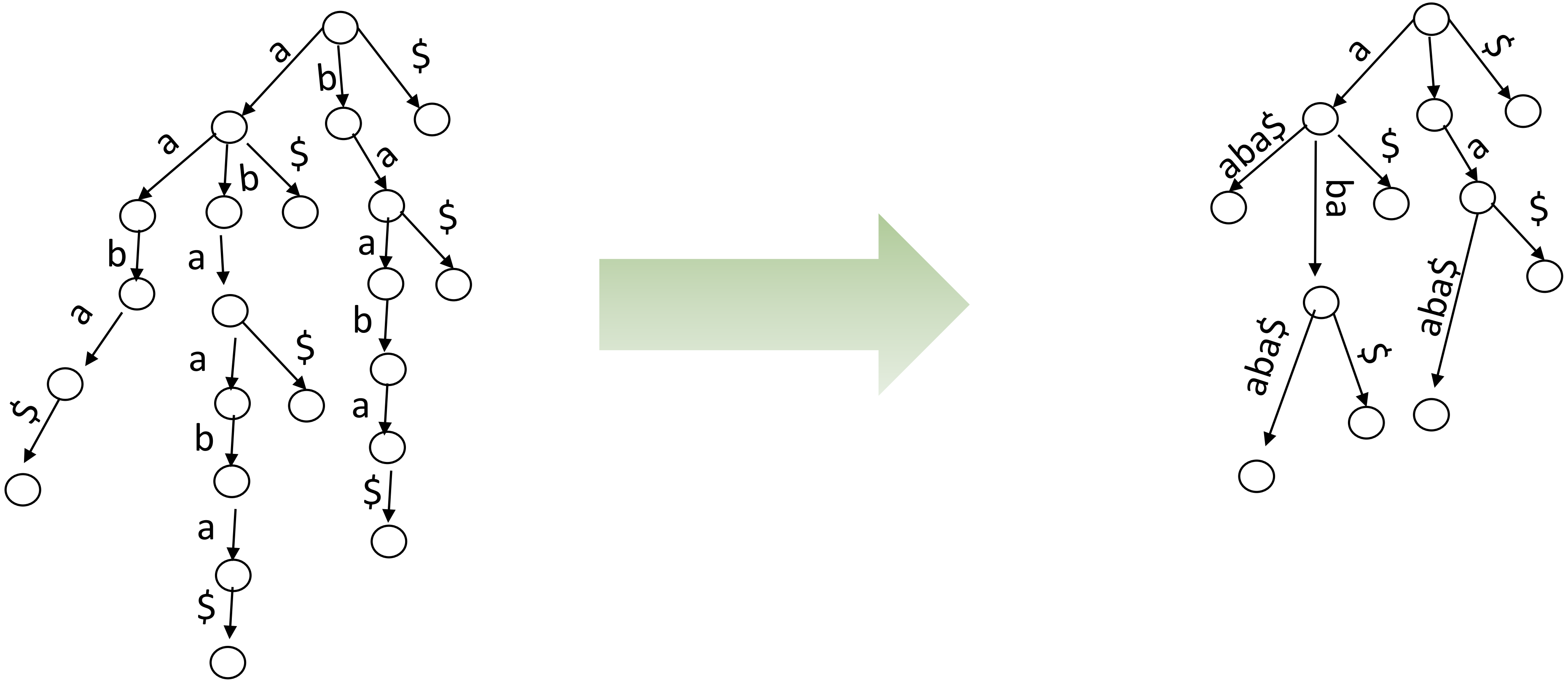
Graph Credit:
[Ben Langmead, JHU]

Suffix Tries – not good enough!

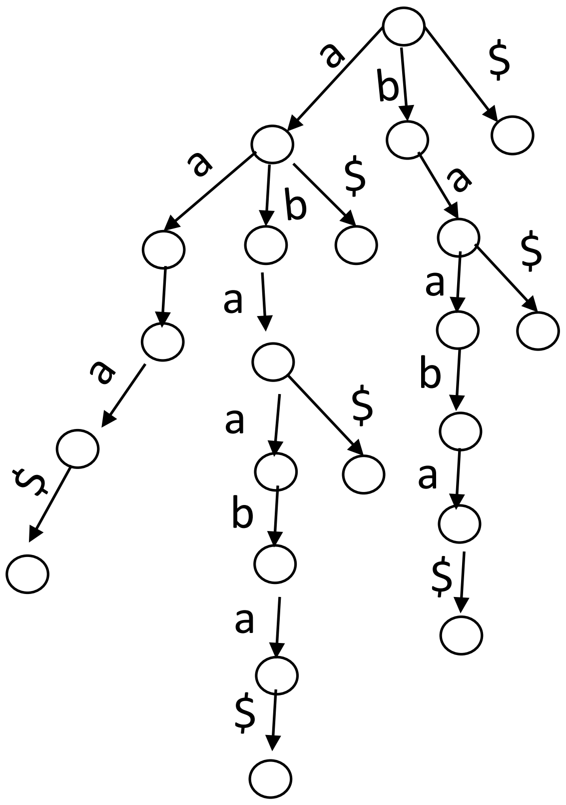
- $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space is not good enough.
  - W.r.t. space, recall that every node in the trie also needs to hold a $|\Sigma|$ −sized array of pointers!
- So suffix tries are not really practical, but offer interesting theoretical insights.
  - Goal: Maintaining the same functionality *(allowing the same queries)*, beat $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space! ☺

Suffix Trees

- A suffix tree is a ***compressed*** (***Patricia***) suffix trie with some additional bells and whistles.

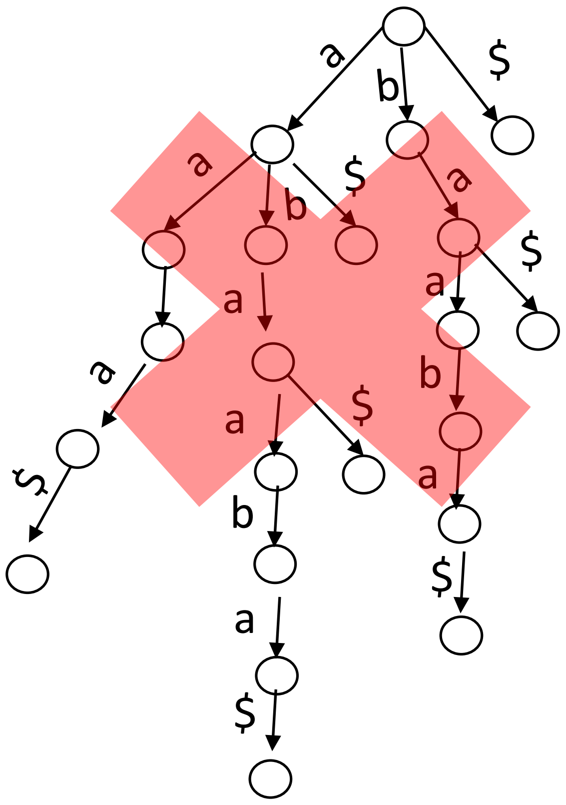


nodes in a suffix tree



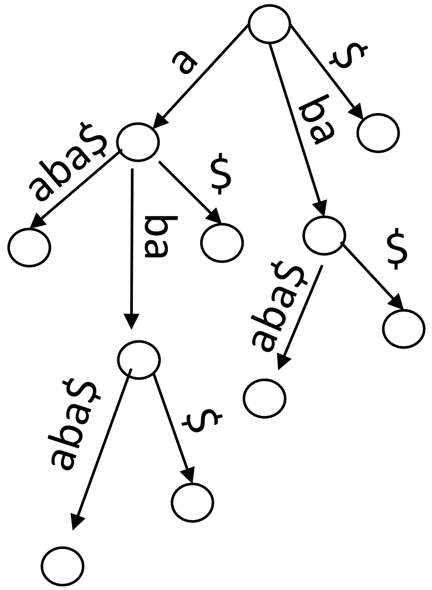
- We saw that suffix tries have $O(n^2)$ nodes…

nodes in a suffix tree



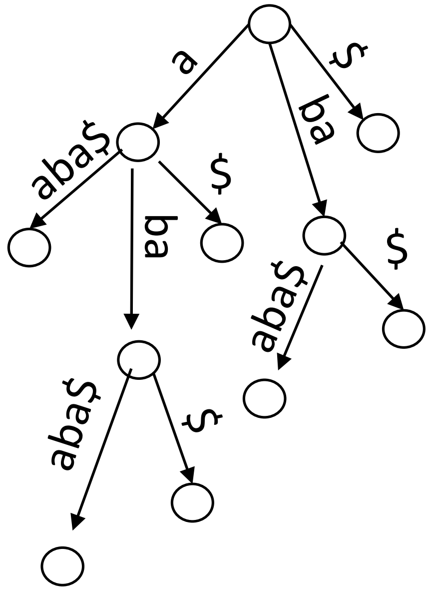
- We saw that suffix tries have $O(n^2)$ nodes… which is **not acceptable** in practice.

nodes in a suffix tree



- We saw that suffix tries have $O(n^2)$ nodes... which is not acceptable in practice.
- Question: In terms of "big-Oh" what is the **spatial complexity** of a suffix **tree?**

nodes in a suffix tree



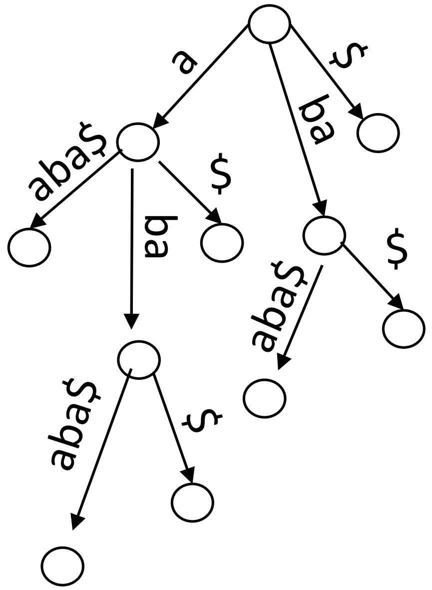
- We saw that suffix tries have $\mathcal{O}(n^2)$ nodes… which is not acceptable in practice.
- Question: In terms of "big-Oh" what is the **spatial complexity** of a suffix **tree?**

$\mathcal{O}(n^2)$

Better

Worse

nodes in a suffix tree



- We saw that suffix tries have $\mathcal{O}(n^2)$ nodes… which is not acceptable in practice.
- Question: In terms of "big-Oh" what is the **spatial complexity** of a suffix **tree?**

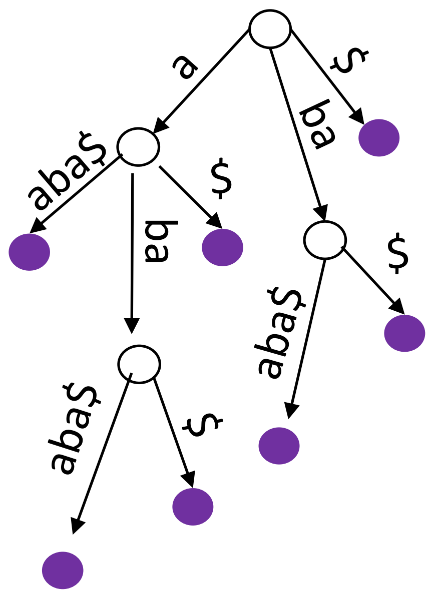
$\mathcal{O}(n)$!

$\mathcal{O}(n^2)$

Better

Worse

nodes in a suffix tree



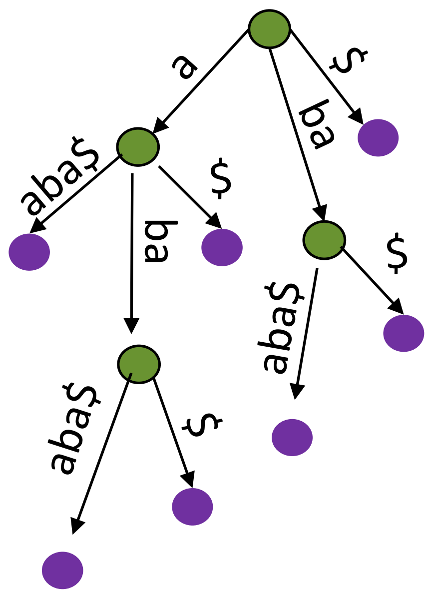
- We saw that suffix tries have $\mathcal{O}(n^2)$ nodes… which is not acceptable in practice.
- Question: In terms of “big-Oh” what is the **spatial complexity** of a suffix **tree?**

$\mathcal{O}(n)$!

$\mathcal{O}(n^2)$ | Better | Worse

- For text of length n, there are $n + 1$ leaf nodes

nodes in a suffix tree



- We saw that suffix tries have $\mathcal{O}(n^2)$ nodes… which is not acceptable in practice.
- Question: In terms of "big-Oh" what is the **spatial complexity** of a suffix **tree?**

$\mathcal{O}(n)$!

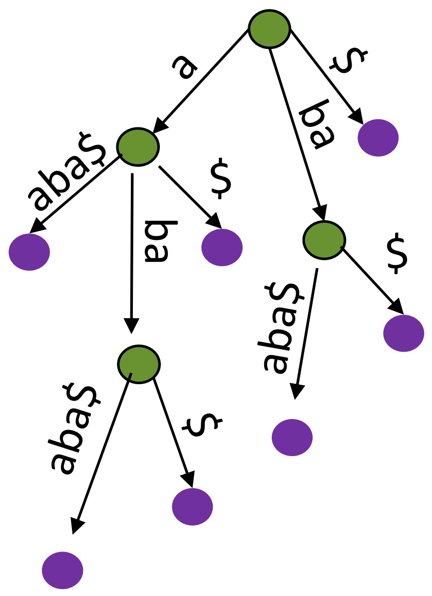
$\mathcal{O}(n^2)$

Better

Worse

- For text of length n, there are $n + 1$ leaf nodes
- There are also k inner nodes, for some $k \in \mathbb{N}$

nodes in a suffix tree



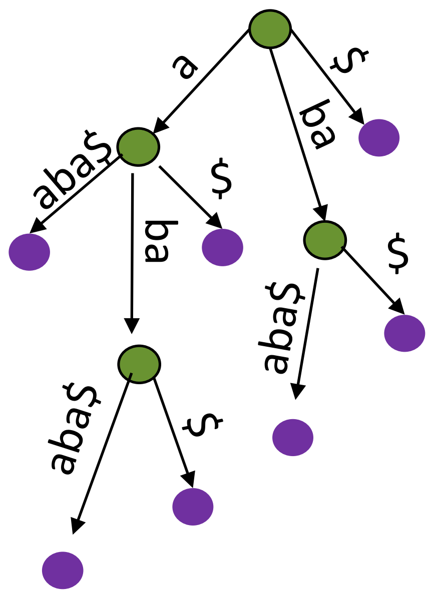
- We saw that suffix tries have $\mathcal{O}(n^2)$ nodes… which is not acceptable in practice.
- Question: In terms of “big-Oh” what is the **spatial complexity** of a suffix **tree?**

$\mathcal{O}(n)$!

$\mathcal{O}(n^2)$ | Better | Worse

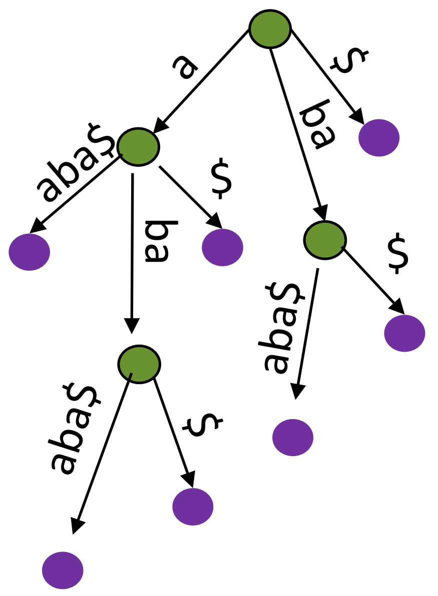
- For text of length n, there are $n + 1$ leaf nodes
- There are also k inner nodes, for some $k \in \mathbb{N}$
- However, $k = \mathcal{O}(n)$!

Why $k = \mathcal{O}(n)$



- Since suffix trees are Patricia tries , every inner node has at least 2 children.
- So our suffix tries will be "at least" **full binary trees (0 or 2 children).**
- But in a full binary tree, **#inner nodes = #leaves – 1**.
  - Increasing the arity of the tree can only serve to make the leaves more than the inner nodes!
- Therefore, #inner nodes $\leq$ #leaf nodes - 1$\Leftrightarrow k \leq (n+1) - 1 \Leftrightarrow k \leq n \Rightarrow k = \mathcal{O}(n)$
- Add to this the $n+1$ leaves and we have an upper bound of $2n+1 = \mathcal{O}(n)$ for the number of total nodes in the tree (inner ones and leaves)

Quiz time!



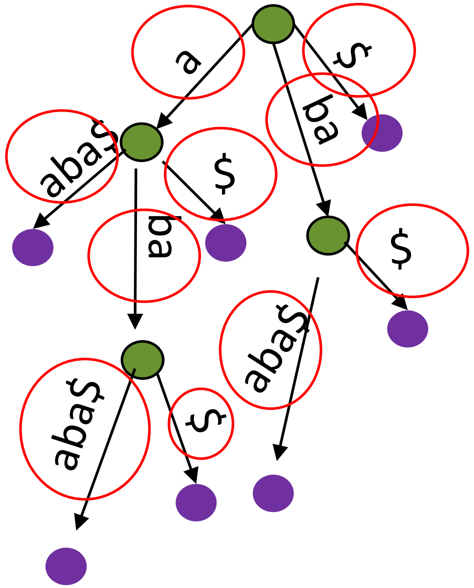
- A suffix tree, as shown on the left, consumes $\mathcal{O}(n)$ **space** (for a text of length n).

True

False

Quiz time!

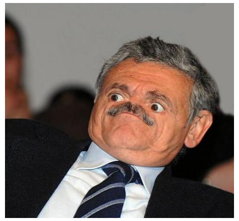
Inserting quadratically many substrings ☹



- A suffix tree, as shown on the left, consumes $\boldsymbol{\mathcal{O}(n)}$ **space** (for a text of length n).

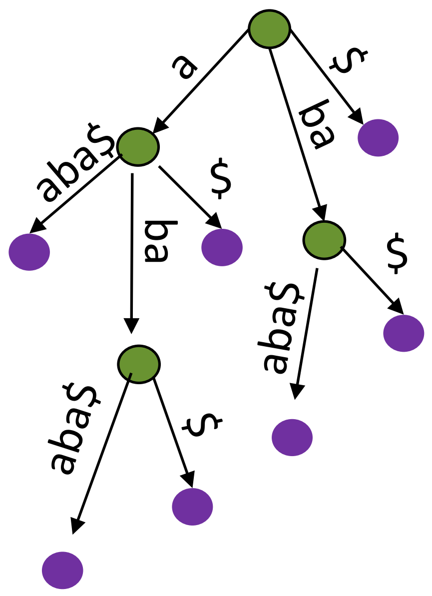
True

False



Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

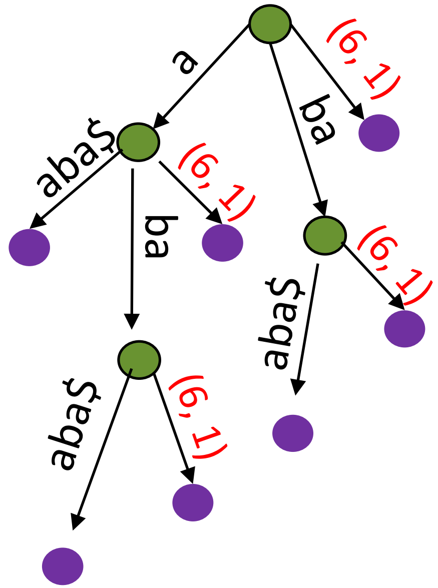
True

False

- <u>**Solution**</u>: Instead of actual characters (like in a Patricia Trie), store **two integers**, offset and length, in every node.
  - Those will be the offset from 0 and length of the substring *the way it appears in T$*.
- Consistent with what the structure would be used for (string matching).

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes $\boldsymbol{\mathcal{O}(n)}$ **space** (for a text of length n).

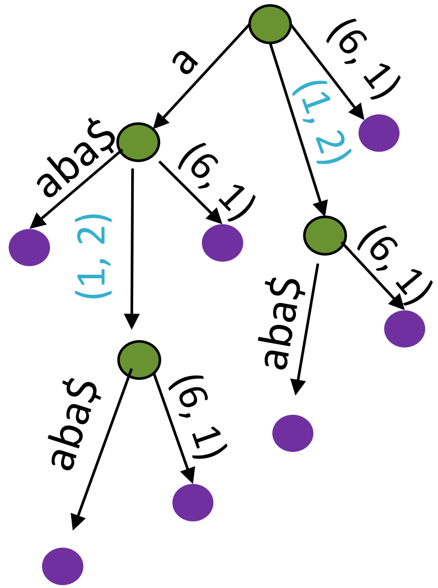
True

False

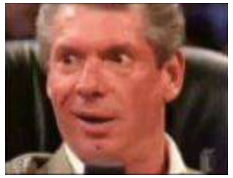
- **Solution**: Instead of actual characters (like in a Patricia Trie), store **two integers**, offset and length, in every node.
  - Those will be the offset from 0 and length of the substring *the way it appears in T$*.
- Consistent with what the structure would be used for (string matching).

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).



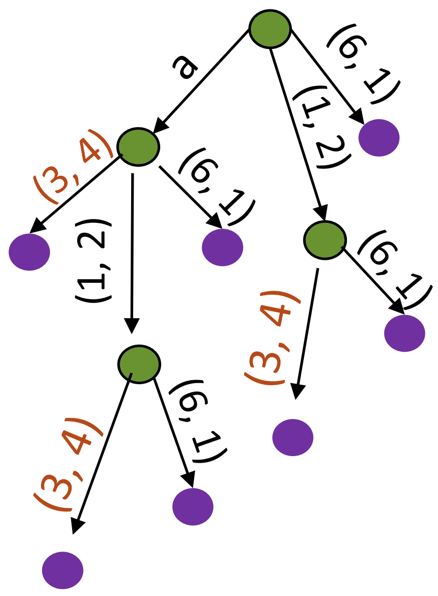
True

False

- Solution: Instead of actual characters (like in a Patricia Trie), store **two integers**, offset and length, in every node.
  - Those will be the offset from 0 and length of the substring *the way it appears in T$*.
- Consistent with what the structure would be used for (string matching).

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

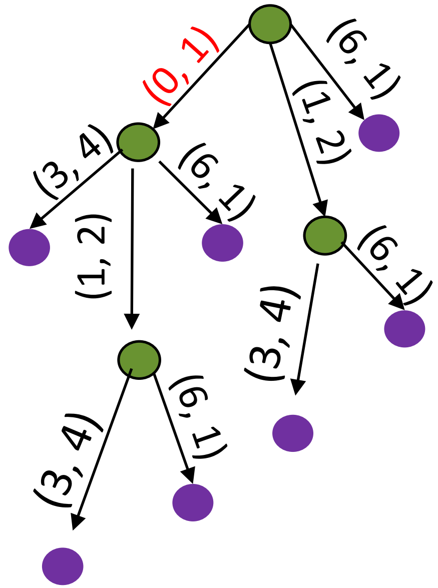
True

False

- Solution: Instead of actual characters (like in a Patricia Trie), store **two integers**, offset and length, in every node.
  - Those will be the offset from 0 and length of the substring *the way it appears in T$*.
- Consistent with what the structure would be used for (string matching).

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes $\mathcal{O}(n)$ **space** (for a text of length n).

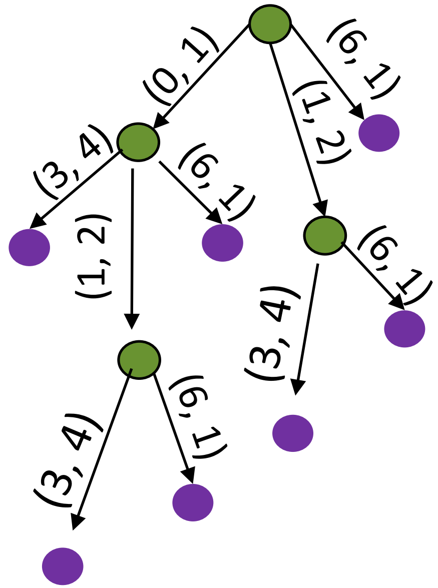
True

False

- Solution: Instead of actual characters (like in a Patricia Trie), store **two integers**, offset and length, in every node.
    - Those will be the offset from 0 and length of the substring *the way it appears in T\$*.
- Consistent with what the structure would be used for (string matching).

Quiz time!

$T\$ = abaaba\$$



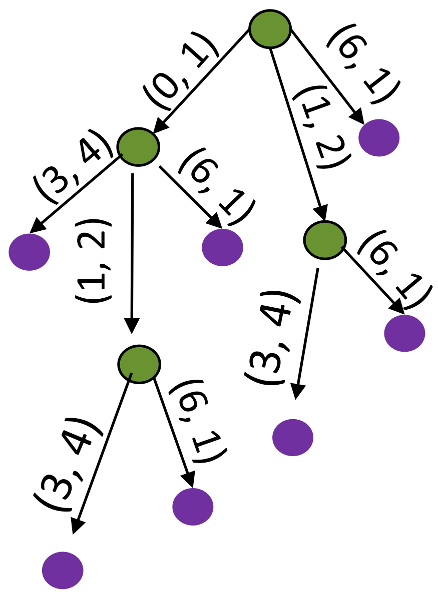
- A suffix tree, ***as shown on the left,*** consumes **$\mathcal{O}(n)$ space** (for a text of length n).

True

False

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

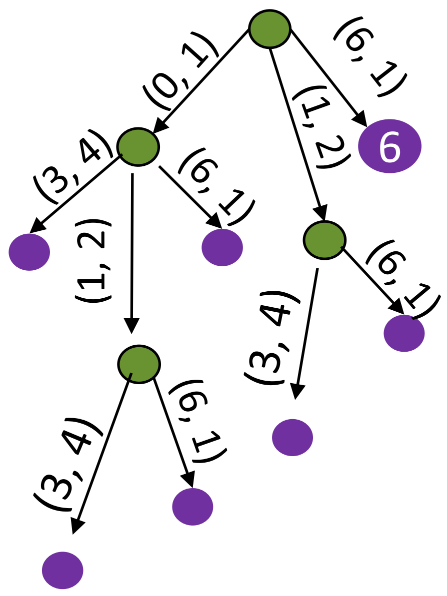
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

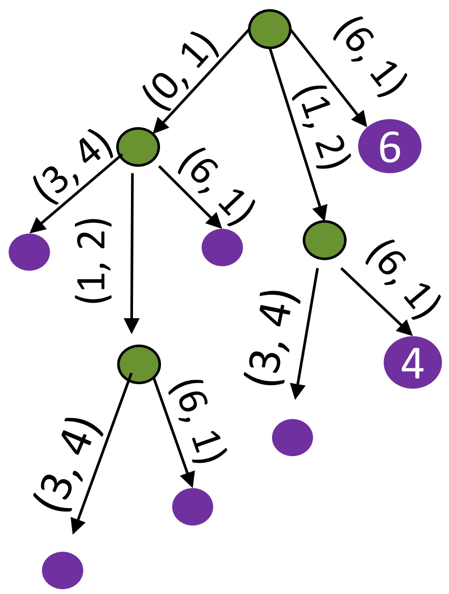
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = aba\textcolor{purple}{\boldsymbol{aba\$}}$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

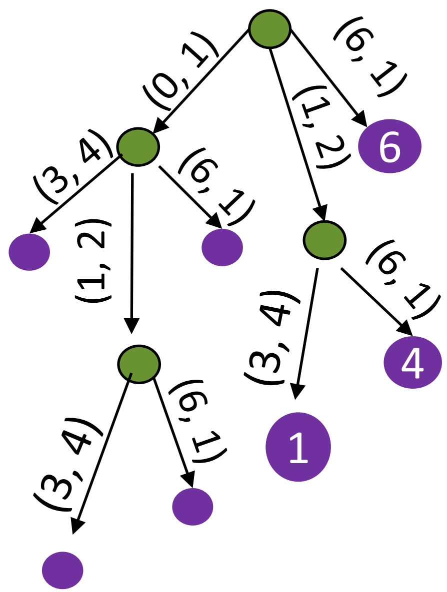
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = abaaba\$$



- A suffix tree, as shown on the left, consumes **$\mathcal{O}(n)$ space** (for a text of length n).

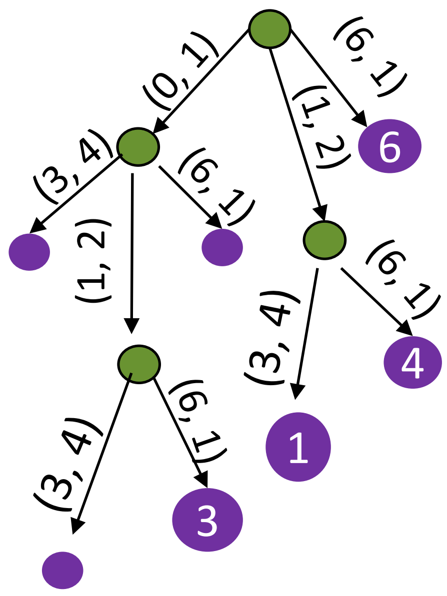
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = aba\mathbf{aba}\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

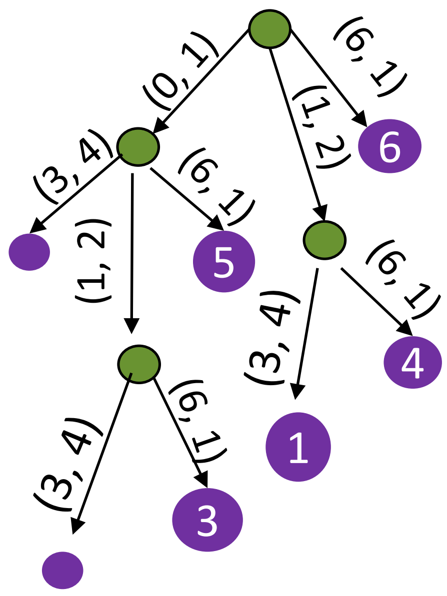
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = ababa\textcolor{purple}{a}\$$



- A suffix tree, as shown on the left, consumes <u>$\boldsymbol{\mathcal{O}(n)}$ **space**</u> (for a text of length n).

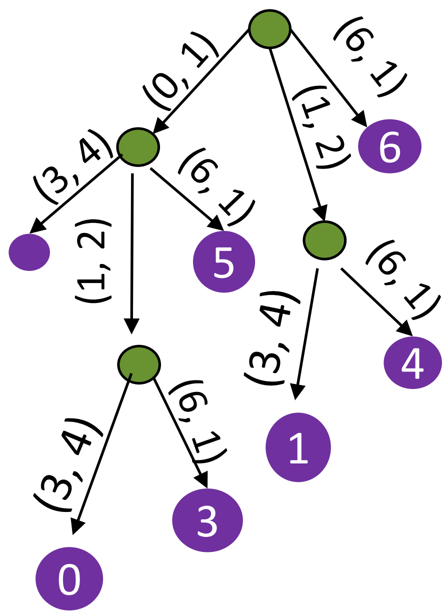
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = \boldsymbol{abaaba\$}$



- A suffix tree, as shown on the left, consumes $\boldsymbol{\mathcal{O}(n)}$ **space** (for a text of length n).

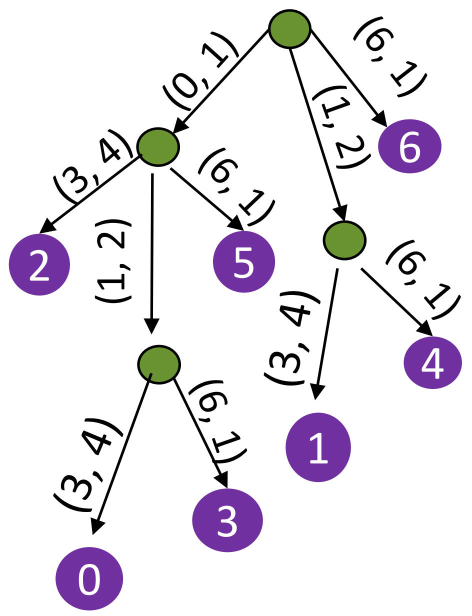
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

Quiz time!

$T\$ = ab\textbf{aaba}\$$



- A suffix tree, as shown on the left, consumes $\boldsymbol{\mathcal{O}(n)}$ **space** (for a text of length n).

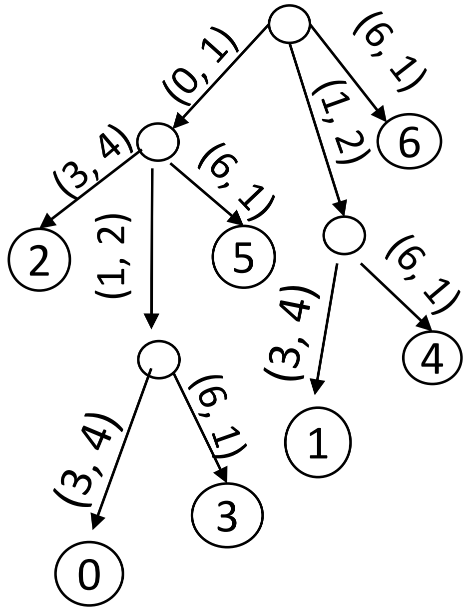
True

False

- Final tweak: The **leaf nodes** will be associated with the (zero-indexed) offsets of the ***entire suffixes to which they correspond.***
  - Recall that the per-inner-node offsets we have stored concern the offsets of **substrings** of these suffixes.

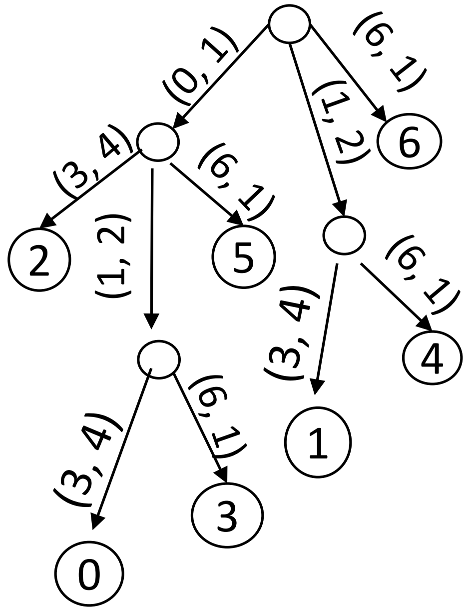
Using a suffix tr*ee*

How can we use a suffix tree to….



$T\$ = abaaba\$$

Using a suffix tr***ee***

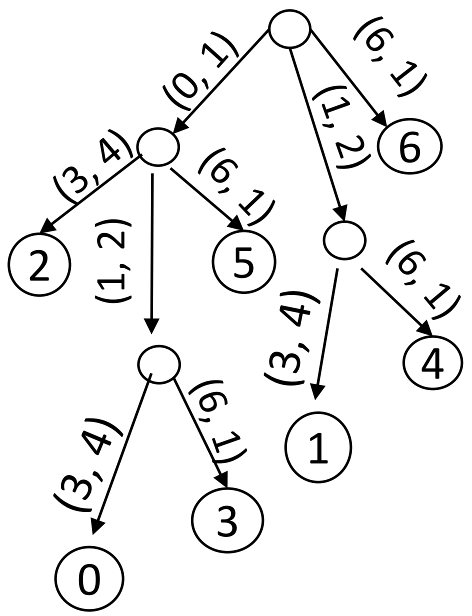


$T\$ = abaaba\$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

Using a suffix tr*ee*



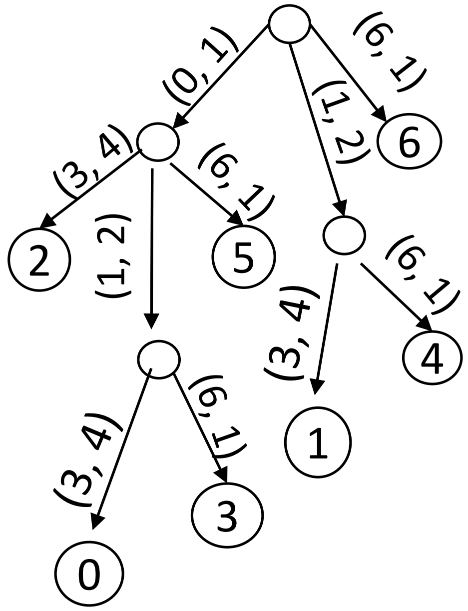
$T\$ = abaaba\$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

Using a suffix tr*ee*



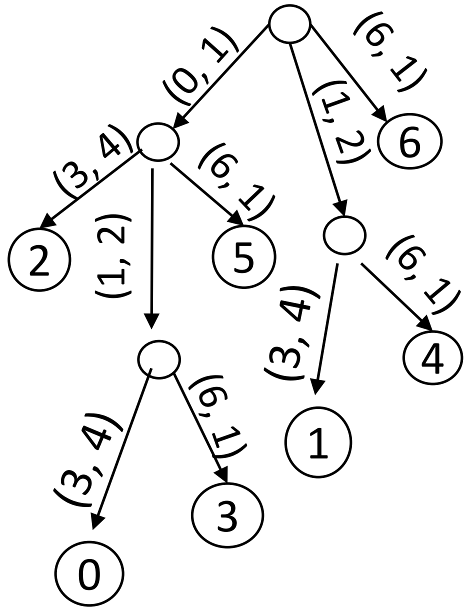
$$T\$ = abaaba\$$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?
Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

Using a suffix tr*ee*



$$T\$ = abaaba\$$$

How can we use a suffix tree to….

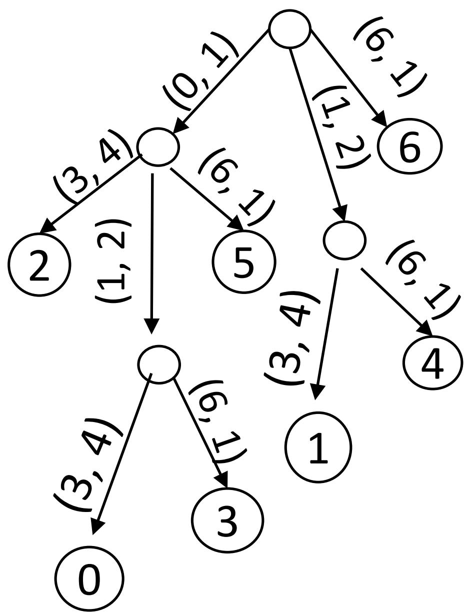
a) Check if a string S is a ***substring*** of the text T?

Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

Using a suffix tr*ee*



$$T\$ = abaaba\$$$

How can we use a suffix tree to….

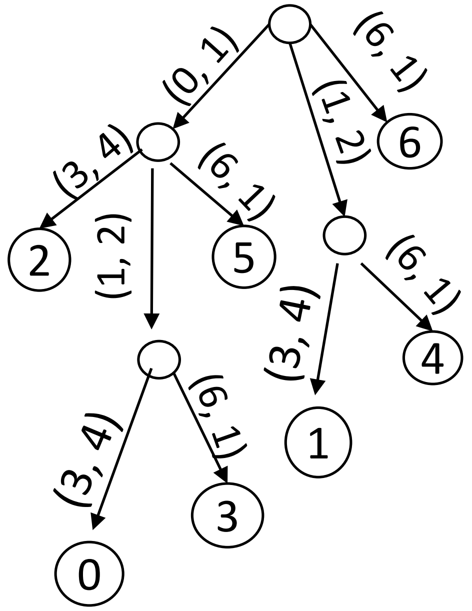
a) Check if a string S is a ***substring*** of the text T?

Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

Using a suffix tr***ee***



$T\$ = abaaba\$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

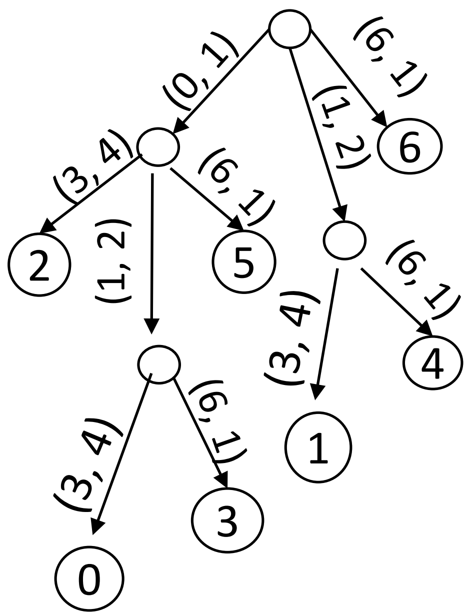
Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

c) Count the ***number of times*** a string S occurs as a substring of T?

Using a suffix tr*ee*



$T\$ = abaaba\$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

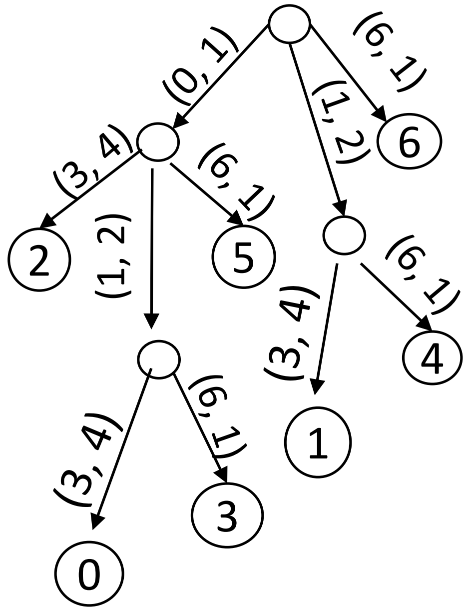
Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

c) Count the ***number of times*** a string S occurs as a substring of T?

Using a suffix tr***ee***



$$T\$ = abaaba\$$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

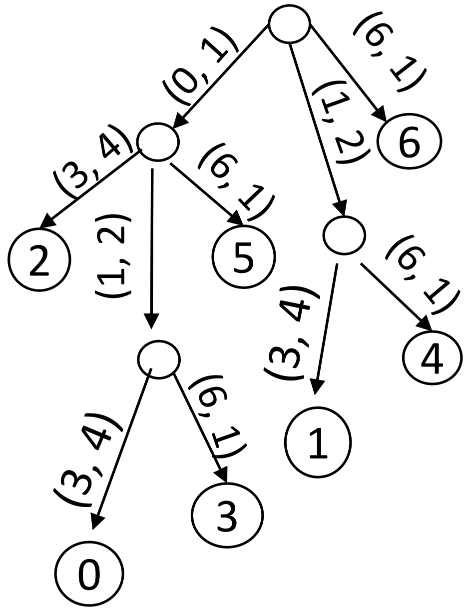
b) Check if a string S is a ***suffix*** of the text T?

The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

c) Count the ***number of times*** a string S occurs as a substring of T?

By using the strictly positive "length" parameter of any given node, at some point we will exhaust S, either "at a node" or "at an edge" (length parameter longer than residual string). In either case, the solution is the #leaf nodes underneath the relevant node's subtree.

Using a suffix tr*ee*



$$T\$ = abaaba\$$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?

Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?

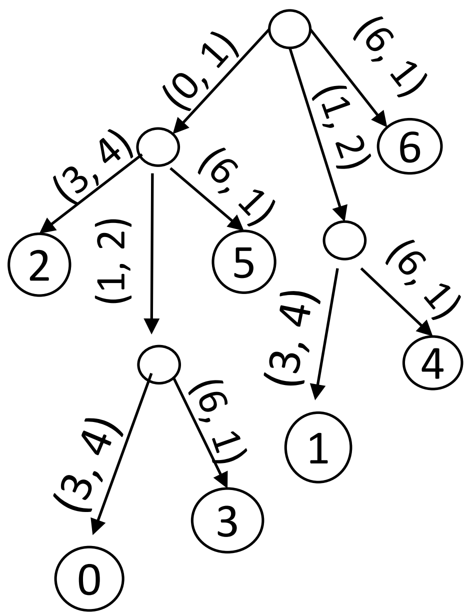
The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

c) Count the ***number of times*** a string S occurs as a substring of T?

By using the strictly positive "length" parameter of any given node, at some point we will exhaust S, either "at a node" or "at an edge" (length parameter longer than residual string). In either case, the solution is the #leaf nodes underneath the relevant node's subtree.

d) Find the ***longest repeated substring*** of T?

Using a suffix tr*ee*



$T\$ = abaaba\$$

How can we use a suffix tree to….

a) Check if a string S is a ***substring*** of the text T?
Consume the input string by chopping away "length"-many characters at every node, and following the link corresponding to the next character. If we don't fall of the tree through any of these links, we answer "yes". If we consume the entire string before traversing an outgoing link from a node, we also answer "yes". Otherwise, "no".

b) Check if a string S is a ***suffix*** of the text T?
The entire string needs to have been consumed by the various nodes' length parameters, and there needs to be an outgoing edge labeled '$'.

c) Count the ***number of times*** a string S occurs as a substring of T?
By using the strictly positive "length" parameter of any given node, at some point we will exhaust S, either "at a node" or "at an edge" (length parameter longer than residual string). In either case, the solution is the #leaf nodes underneath the relevant node's subtree.

d) Find the ***longest repeated substring*** of T?
Similar to suffix tries, we need to find the "deepest" node with at least two children.

Suffix trees – you try them out!

- Give me the suffix tree for the text **T=xxyzxxzy.**

Analysis of suffix trees

- When compared to suffix tries, they offer $\mathcal{O}(n)$ space (instead of $\mathcal{O}(n^2)$) .
  - Construction can be done in $\mathcal{O}(n)$ time as well by using **Ukkonen's algorithm** (check uploaded CS423 slides).

Analysis of suffix trees

- When compared to suffix tries, they offer $\mathcal{O}(n)$ space (instead of $\mathcal{O}(n^2)$) .
  - Construction can be done in $\mathcal{O}(n)$ time as well by using **Ukkonen's algorithm** (check uploaded CS423 slides).
- One issue: Space might be $\mathcal{O}(n)$, but it has an unfavorable constant up front.
  - Recall: every node needs a $|\Sigma|$ - sized array of links, two integers, and possible suffix links (those are required by Ukkonen's algorithm)

Analysis of suffix trees

- When compared to suffix tries, they offer $\mathcal{O}(n)$ space (instead of $\mathcal{O}(n^2)$) .
  - Construction can be done in $\mathcal{O}(n)$ time as well by using **Ukkonen's algorithm** (check uploaded CS423 slides).
- One issue: Space might be $\mathcal{O}(n)$, but it has an unfavorable constant up front.
  - Recall: every node needs a $|\Sigma|$ - sized array of links, two integers, and possible suffix links (those are required by Ukkonen's algorithm)
- ***Goal***: reproduce functionality while lowering space cost ☺

Part 3: Pre-processing the ***text*** with suffix arrays

Suffix arrays

- Given a text T, a *suffix array* is a single-dimensional array of **lexicographically sorted suffixes** of T$.

Suffix arrays

- Given a text T, a *suffix array* is a single-dimensional array of **lexicographically sorted suffixes** of T$.
- Example: consider the string T=jason.

Suffix arrays

- Given a text T, a *suffix array* is a single-dimensional array of **lexicographically sorted suffixes** of T$.
- Example: consider the string T=jason.

Suffixes	Suffix array
$	$
n$	ason$
on$	jason$
son$	n$
ason$	on$
jason$	son$

Suffix arrays

- Given a text T, a *suffix array* is a single-dimensional array of **lexicographically sorted suffixes** of T$.
- Example: consider the string T$=jason$.

Suffixes	**Suffix array**
$	**5** ~~$~~
n$	**1** ~~ason$~~
on$	**0** ~~jason$~~
son$	**4** ~~n$~~
ason$	**3** ~~on$~~
jason$	**2** ~~son$~~

To avoid storing quadratically many characters, replace suffixes with offsets into augmented string!

Suffix arrays

- Given a text T, a *suffix array* is a single-dimensional array of **lexicographically sorted suffixes** of T$.
- Example: consider the string T$=jason$.

Suffixes	Suffix array
$	**5**
n$	**1**
on$	**0**
son$	**4**
ason$	**3**
jason$	**2**

To avoid storing quadratically many characters, replace suffixes with offsets into augmented string!

Space cost: $\mathcal{O}(n)$ *with a far better constant up front (4, for size of int)*

Building a suffix array

1) Build the suffix tree and do an inorder traversal. Because this traversal is sorted traversal in Patricia tries, every time you encounter a string you insert it in `SuffArr[i++]`.
   - Makes sense if you have a suffix tree that you want to ditch.
   - Recall: Suffix tree built in $O(n)$ time through Ukkonen's algorithm and consumes $O(n)$ space, but with a bad constant up front.
2) Insert all the strings into an array which you then sort in-place.
   - $O(n)$ space
   - $O(n \log_2 n)$ time

Building a suffix array

1) Build the suffix tree and do an inorder traversal. Because this traversal is sorted traversal in Patricia tries, every time you encounter a string you insert it in `SuffArr[i++]`.
    - Makes sense if you have a suffix tree that you want to ditch.
    - Recall: Suffix tree built in $O(n)$ time through Ukkonen's algorithm and consumes $O(n)$ space, but with a bad constant up front.
2) Insert all the strings into an array which you then sort in-place.
    - $O(n)$ space
    - $O(n \log_2 n)$ time

True or False: For approach #2, I would need a stable sorting algorithm

Building a suffix array

1) Build the suffix tree and do an inorder traversal. Because this traversal is sorted traversal in Patricia tries, every time you encounter a string you insert it in `SuffArr[i++]`.
   - Makes sense if you have a suffix tree that you want to ditch.
   - Recall: Suffix tree built in $O(n)$ time through Ukkonen's algorithm and consumes $O(n)$ space, but with a bad constant up front.
2) Insert all the strings into an array which you then sort in-place.
   - $O(n)$ space
   - $O(n \log_2 n)$ time

True or False: For approach #2, I would need a stable sorting algorithm

True

False

Who cares?

Building a suffix array

1) Build the suffix tree and do an inorder traversal. Because this traversal is sorted traversal in Patricia tries, every time you encounter a string you insert it in `SuffArr[i++]`.
   - Makes sense if you have a suffix tree that you want to ditch.
   - Recall: Suffix tree built in $O(n)$ time through Ukkonen's algorithm and consumes $O(n)$ space, but with a bad constant up front.
2) Insert all the strings into an array which you then sort in-place.
   - $O(n)$ space
   - $O(n \log_2 n)$ time

True or False: For approach #2, I would need a stable sorting algorithm

True

False

Who cares?

Since you store different-length suffixes, there will never be a tie… Either mergesort or quicksort will do!

Using a suffix array

- We will use suffix arrays exclusively for the problem of ***string matching***.
  - I.e we will not reproduce the full functionality of suffix trees, sticking with the most fundamental functionality.
  - Extensions of suffix arrays, like the LCP array, allow for the full functionality of suffix trees with similar gains in space.

Using a suffix array

- We will use suffix arrays exclusively for the problem of ***string matching***.
  - I.e we will not reproduce the full functionality of suffix trees, sticking with the most fundamental functionality.
  - Extensions of suffix arrays, like the LCP array, allow for the full functionality of suffix trees with similar gains in space.
- Key **requirement**: we need **all matches** and their original text offsets!

Using a suffix array

- We will use suffix arrays exclusively for the problem of ***string matching***.
  - I.e we will not reproduce the full functionality of suffix trees, sticking with the most fundamental functionality.
  - Extensions of suffix arrays, like the LCP array, allow for the full functionality of suffix trees with similar gains in space.
- Key **requirement**: we need **all matches** and their original text offsets!
- Key **observation**: **if** P occurs in T, then **all** its occurrences in the suffix array are **consecutive**!

Using a suffix array

- We will use suffix arrays exclusively for the problem of ***string matching***.
  - I.e we will not reproduce the full functionality of suffix trees, sticking with the most fundamental functionality.
  - Extensions of suffix arrays, like the LCP array, allow for the full functionality of suffix trees with similar gains in space.
- Key **requirement**: we need **all matches** and their original text offsets!
- Key **observation**: **if** P occurs in T, then **all** its occurrences in the suffix array are **consecutive**!
  - We can find the end-points of this interval with **two binary searches** in the suffix array!

Pseudocode

```
function printMatches(T:text, P:pattern){
        n = |T|; m = |P|;
        A = createSuffixArray(T);                    // Pre-processing the text!
        left = 0; right = n;
        while(left < right) {
                mid = (left + right) / 2;
                if(P > T[A[mid]])                    // Pattern lexicographically *smaller* than text at A[mid] (closer to 'Z')
                        left = mid + 1;              // Right subarray
                else
                        right = mid;                 // Left subarray;
        }
        /* Done with first search; Variable left now has the left end of the interval.
         * We execute another binary search to find the right end-point. */
        start = left; right = n;
        while(left < right){
                mid = (left + right)/ 2:
                if(P < T[A[mid]])                    // Pattern lexicographically *greater* than text at A[mid] (closer to `A')
                        right = mid;
                else
                        left = mid + 1;
        }
        end = right + 1;
        printFunc(A, start, end);
}
```

```
function printFunc(A: suffix array, start: int, end:int) {
        assert 0 <= start <= end < A.length          // Requirements for this to work
        for(int i = start; i <= end; i ++){
                print "Pattern occurs in text in index: " + A[i]
        }
}
```

Example: T=quixoticelixir (band name)

Current array indices	Suffix array indices for T$=*quixoticelixir$*	Actual suffixes of T$=quixoticelixir$ (would not be stored in practice, denoted just for clarity)
0	14	$
1	7	celixir$
2	8	elixir$
3	6	icelixir$
4	12	ir$
5	10	ixir$
6	2	ixoticelixir$
7	9	lixir$
8	4	oticelixir$
9	0	quixoticelixir$
10	13	r$
11	5	ticelixir$
12	1	uixoticelixir$
13	11	xir$
14	3	xoticelixir$

Search for "ix"

Example: T=quixoticelixir (band name)

Current array indices	Suffix array indices for T$=*quixoticelixir$*	Actual suffixes of T$=quixoticelixir$ (would not be stored in practice, denoted just for clarity)
0	14	$
1	7	celixir$
2	8	elixir$
3	6	icelixir$
4	12	ir$
5	10	ixir$
6	2	ixoticelixir$
7	9	lixir$
8	4	oticelixir$
9	0	quixoticelixir$
10	13	r$
11	5	ticelixir$
12	1	uixoticelixir$
13	11	xir$
14	3	xoticelixir$

Example: T=quixoticelixir

(band name)

Search for "ix"

Note that indices are consecutive in array…

Current array indices	**Suffix array indices for** ***T$=quixoticelixir$***	**Actual suffixes of T$=quixoticelixir$ (would not be stored in practice, denoted just for clarity)**
0	14	$
1	7	celixir$
2	8	elixir$
3	6	icelixir$
4	12	ir$
5	10	ixir$
6	2	ixoticelixir$
7	9	lixir$
8	4	oticelixir$
9	0	quixoticelixir$
10	13	r$
11	5	ticelixir$
12	1	uixoticelixir$
13	11	xir$
14	3	xoticelixir$

Your turn

- Build the suffix array for "mississippi".
- Then, simulate the search for "is".

Analysis

- Suppose $|T| = n$ and $|P| = m$.
- Then, the computational complexity of finding ***all*** occurrences of P in M is…

Analysis

- Suppose $|T| = n$ and $|P| = m$.
- Then, the computational complexity of finding ***all*** occurrences of P in M is…

$\mathcal{O}(\log_2 n)$

$\mathcal{O}(m \cdot \log_2 n)$

$\mathcal{O}(n \cdot \log_2 m)$

Something else (what?)

Analysis

- Suppose $|T| = n$ and $|P| = m$.
- Then, the computational complexity of finding ***all*** occurrences of P in M is…

$\mathcal{O}(\log_2 n)$ | $\mathcal{O}(m \cdot \log_2 n)$ | $\mathcal{O}(n \cdot \log_2 m)$ | Something else (what?)

- Constant up front: 2, since we execute 2 binary searches.
  - The m comes to be because of the fact that we do an m- length comparison (`strncmp`) every time we compute a mid-point index in the binary searches.
- The 2nd binary search will almost always be cheaper than $\log_2 n$, since we start it from the left end-point of the matched interval.

Recap

- For string matching, we can either pre-process the pattern or the text.
  - For the first approach, we discussed KMP.
  - For the second, we discussed several approaches.

1. Suffix tries: Impractical in practice because of $\mathcal{O}(n^2)$ space and construction time, yet give us a good starting point.
   - Allow for some interesting queries beyond string matching.
2. Suffix trees: An improvement over suffix tries that allows for $\mathcal{O}(n)$ space and construction time (Ukkonen's algorithm)!
   - Allow for the same queries as a suffix trie.
   - One issue: constant in front of the $\mathcal{O}(n)$ space is pretty big.
3. Suffix arrays: Improvement over suffix tries in terms of space (smaller constant involved in $\mathcal{O}(n)$)
   - Vanilla suffix array cannot answer all queries answerable by a suffix tree.
   - Extensions that do: Extended Suffix Arrays, LCP arrays, etc.