

# New Algorithms for Position Heaps

Travis Gagie, Wing-Kai Hon, Tsung-Han Ku

---

CPM2013

# Outline

---

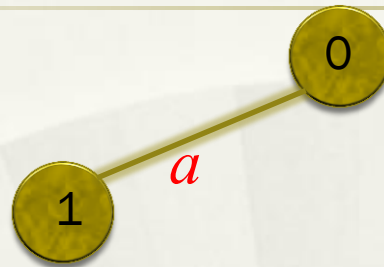
- \* Position Heap
- \* Limiting Length and Height
- \* Turing a Suffix Tree into a Position Heap
- \* Using a Position Heap as a Suffix Array
- \* Using a Compressed Suffix Array as a Position Heap

# Position Heap

---

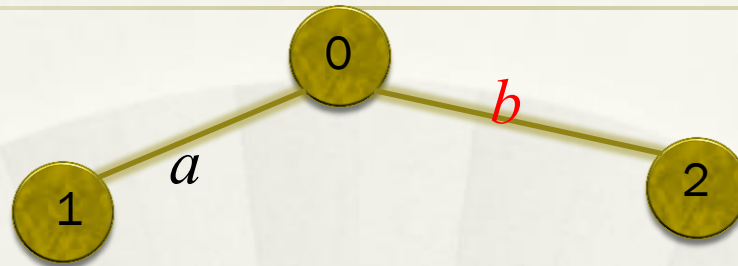
- \* The root is labeled 0 and the other nodes are labeled 1 to  $n$  such that parent's labels are smaller than their children's labels
- \* For  $1 \leq i \leq n$ , the path label of the node labeled  $i$  is a prefix of  $S[i..n]$ .
- \* For  $1 \leq i \leq n$ , the node labeled  $i$  stores a pointer (called maximal-reach pointer) to the deepest node whose path label is a prefix of  $S[i..n]$

# Position Heap



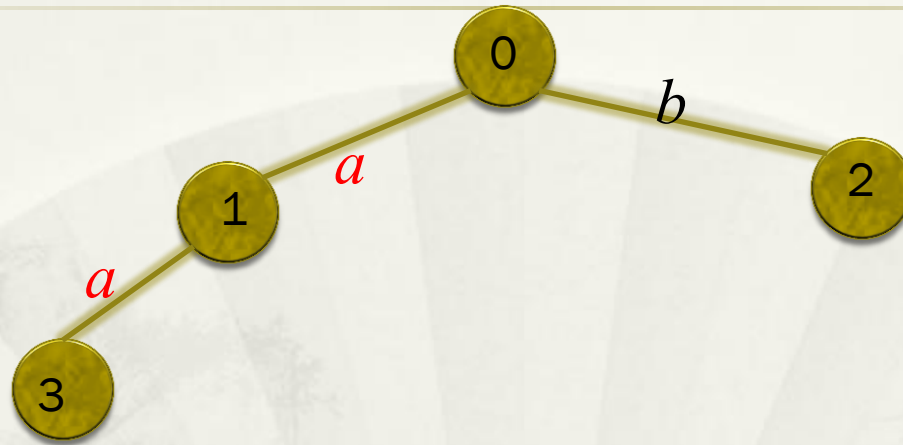
A position heap for  $S = \textit{abaababbabbab}\$$

# Position Heap



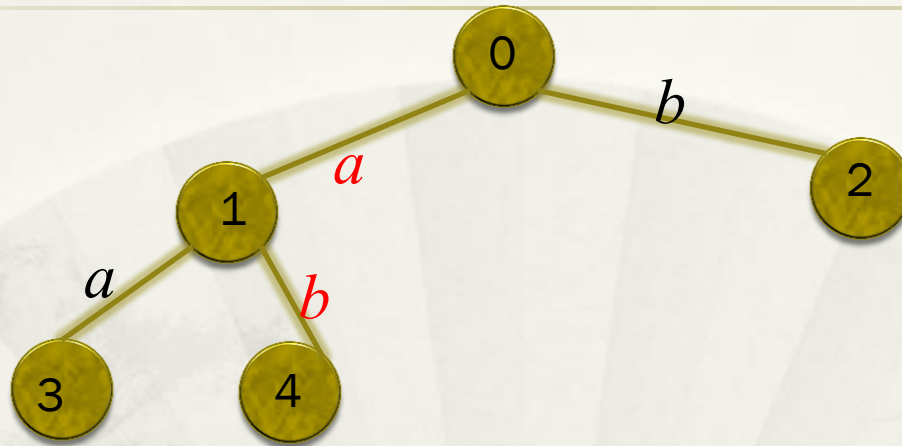
A position heap for  $S = a**b**aababbabbab\$$

# Position Heap



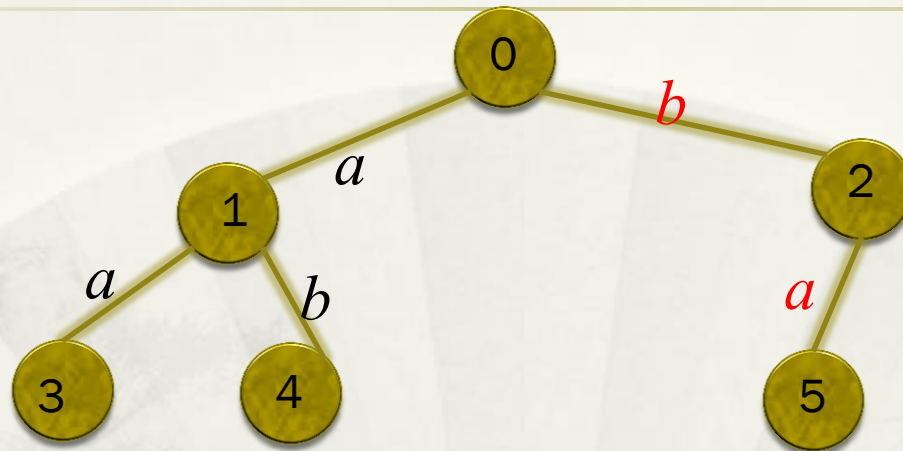
A position heap for  $S = ab\textcolor{red}{aa}babbabbab\$$

# Position Heap



A position heap for  $S = abaabbabbab\$$

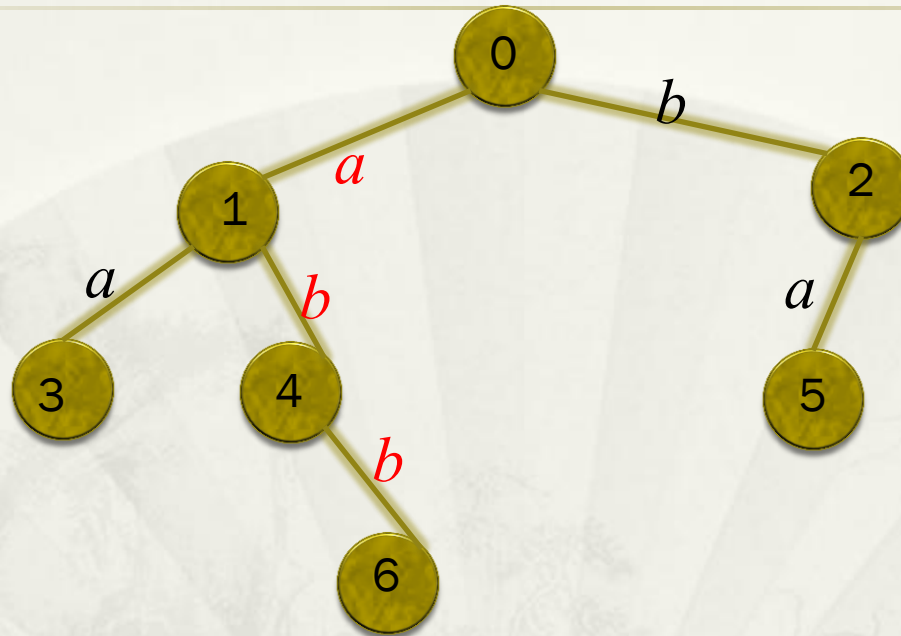
# Position Heap



A position heap for  $S = abaababbabbab\$$

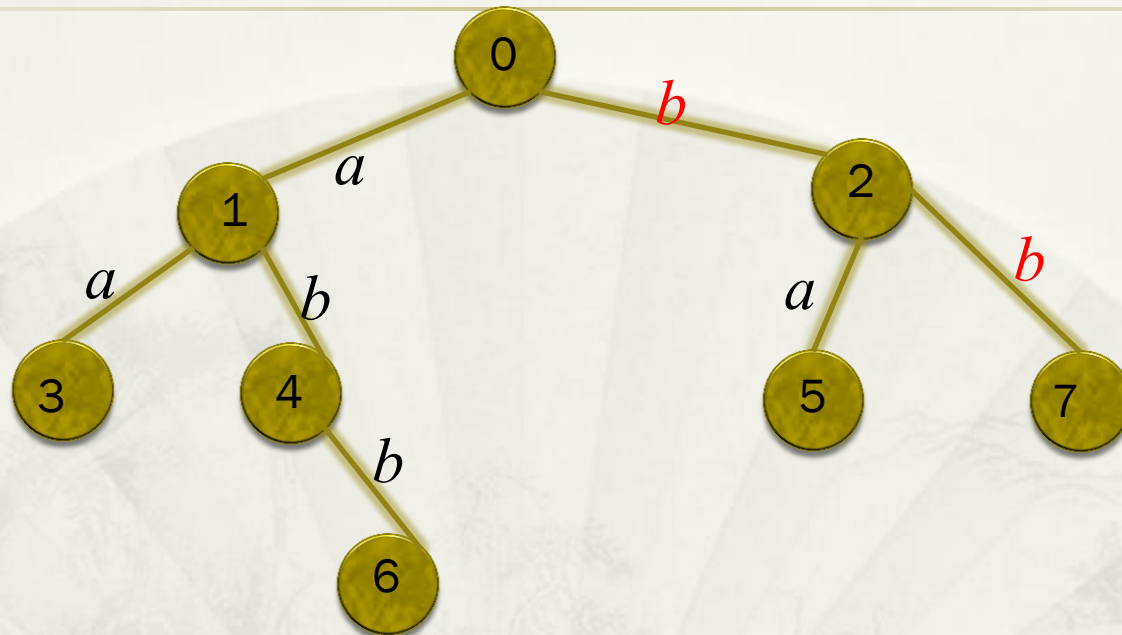


# Position Heap



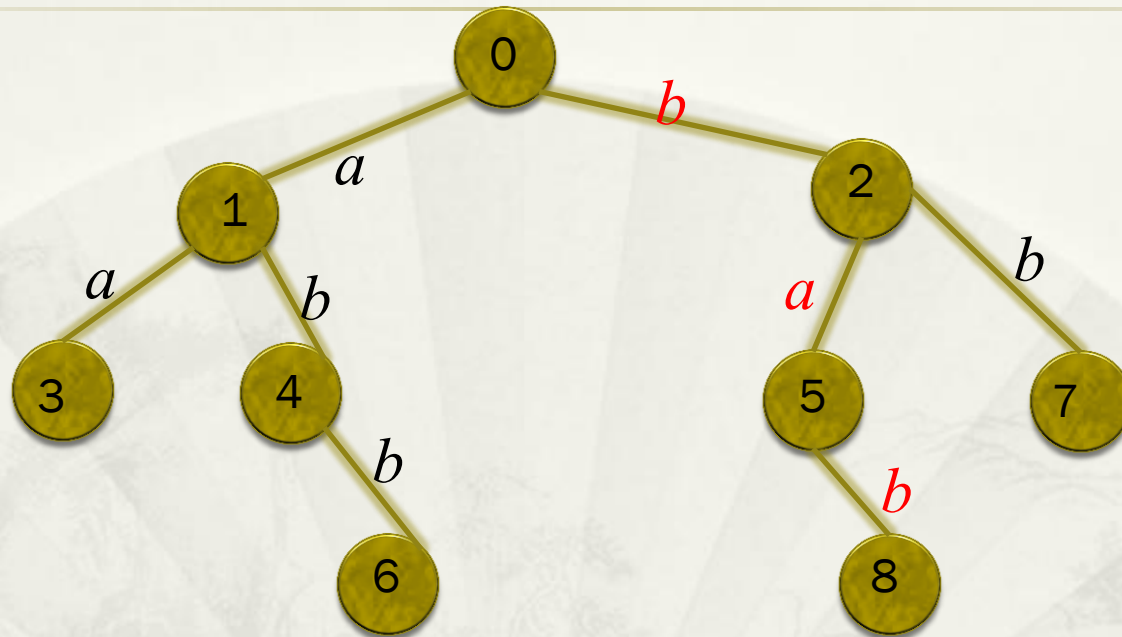
A position heap for  $S = abaababbabab\$$

# Position Heap



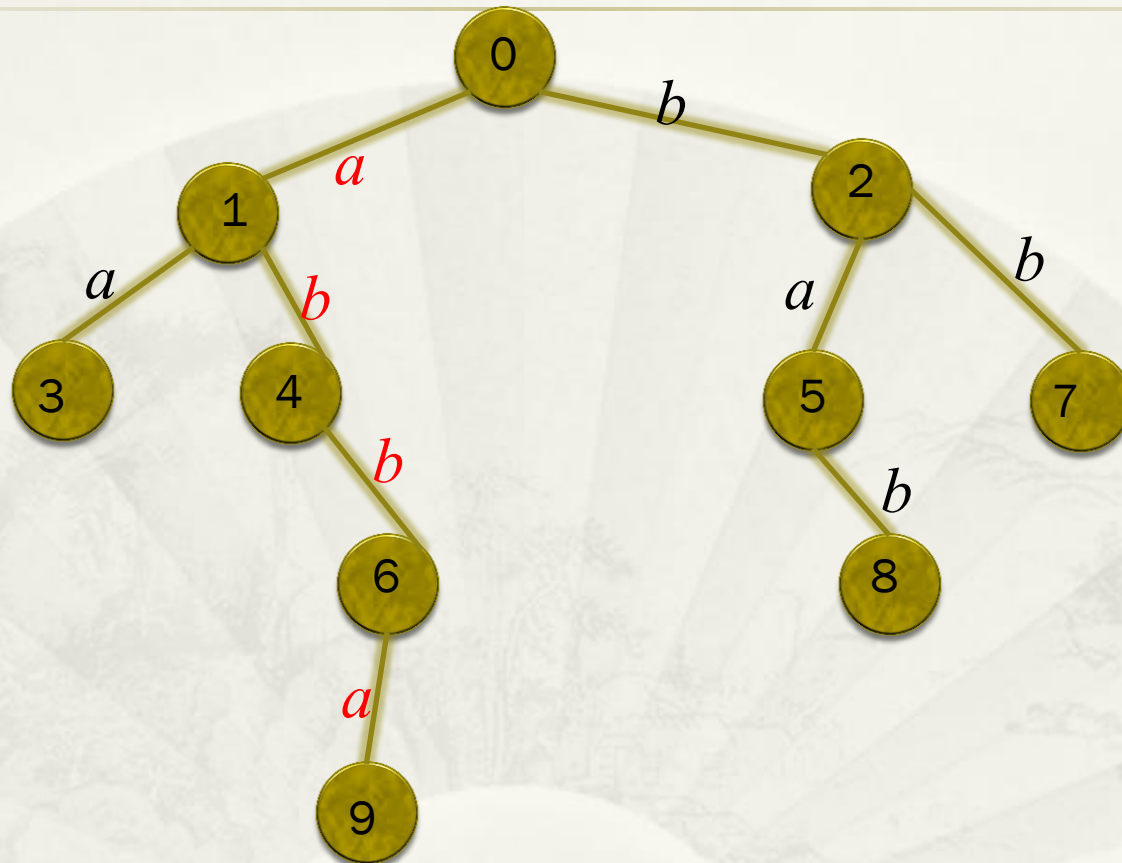
A position heap for  $S = abaabaabbab\$$

# Position Heap



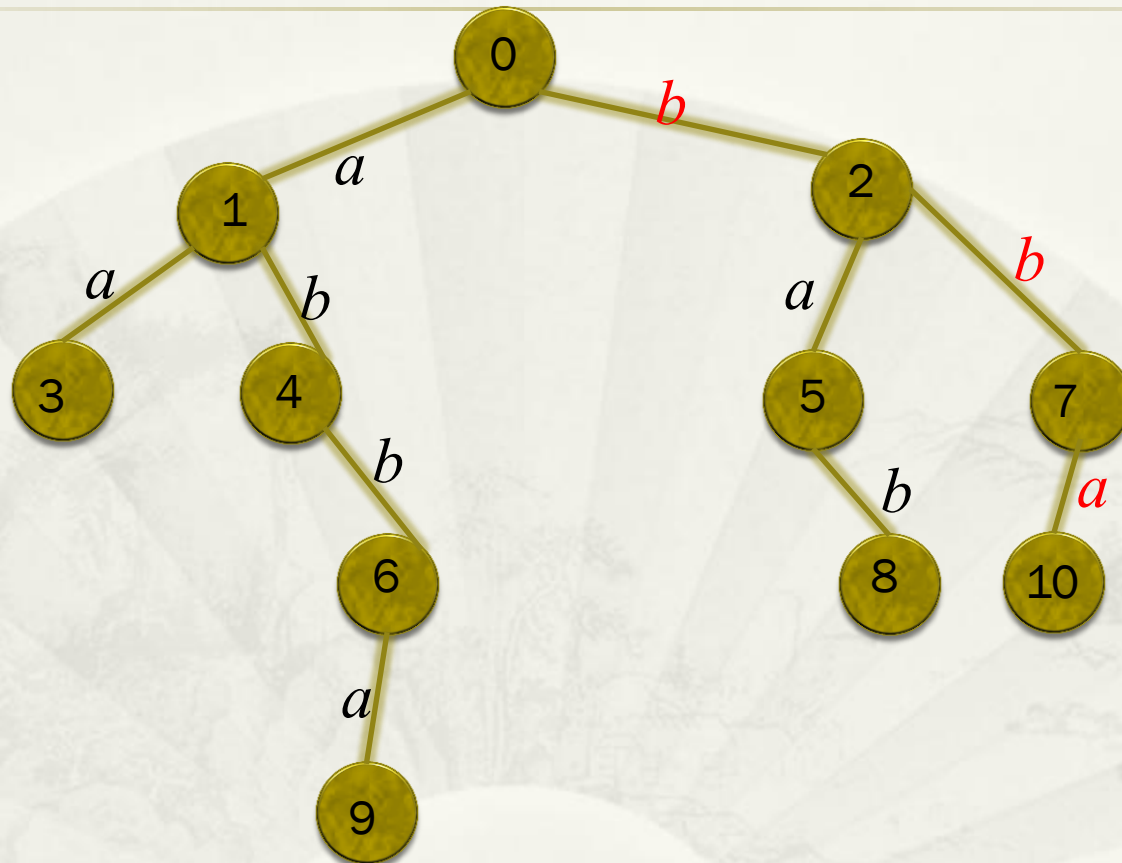
A position heap for  $S = abaababababab\$$

# Position Heap



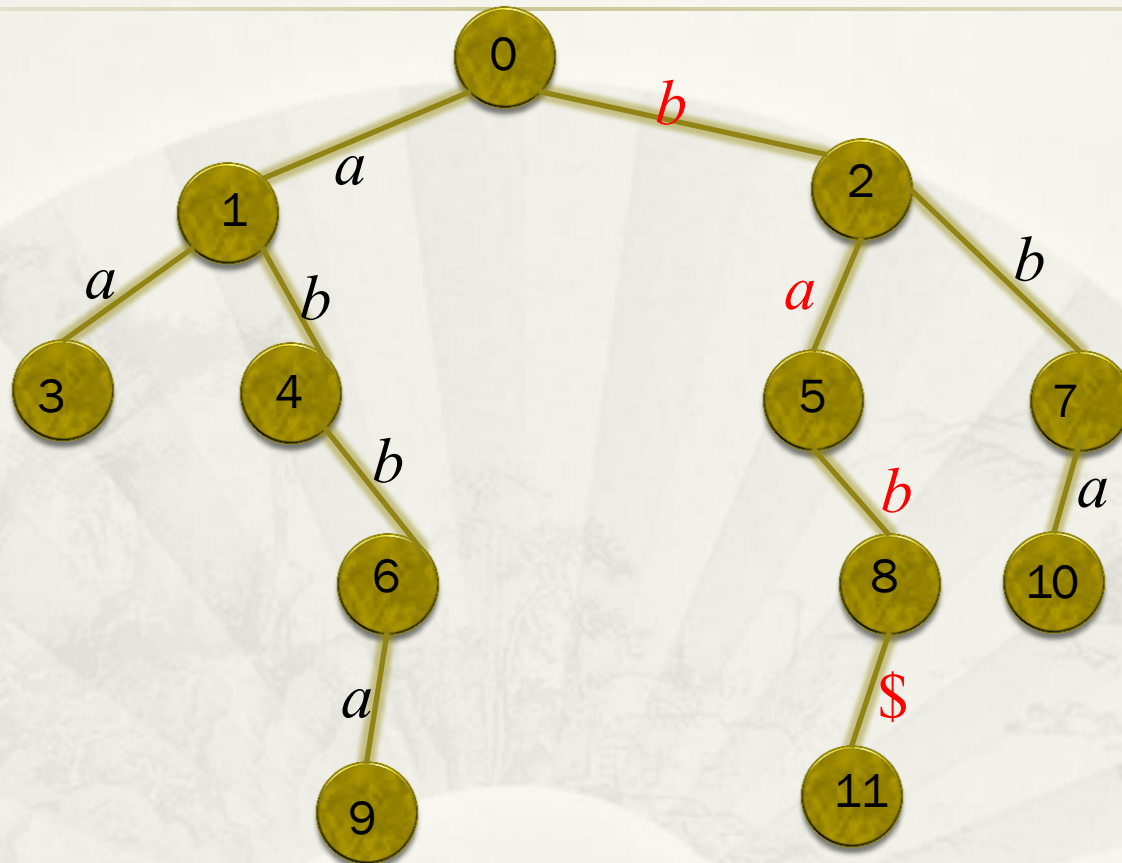
A position heap for  $S = abaababbabab\$$

# Position Heap



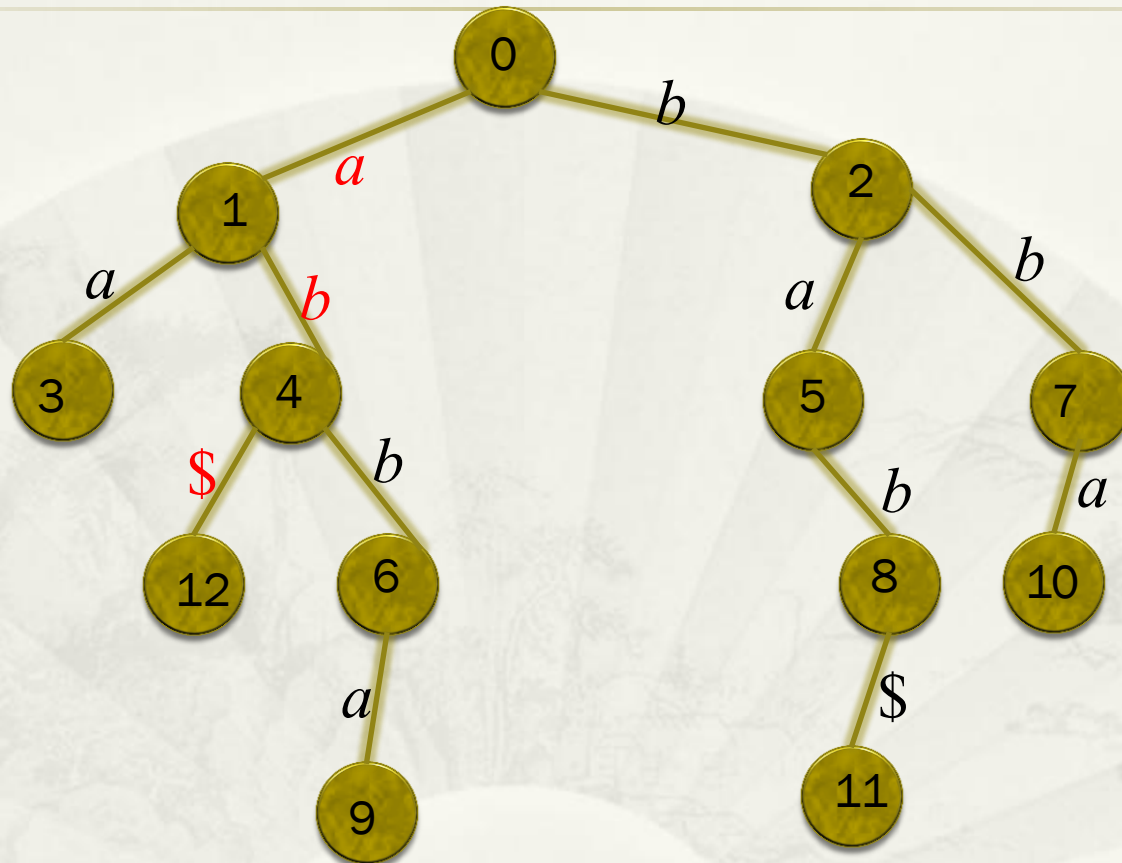
A position heap for  $S = abaababba**bb**ab\$$

# Position Heap



A position heap for  $S = abaababbababab\$$

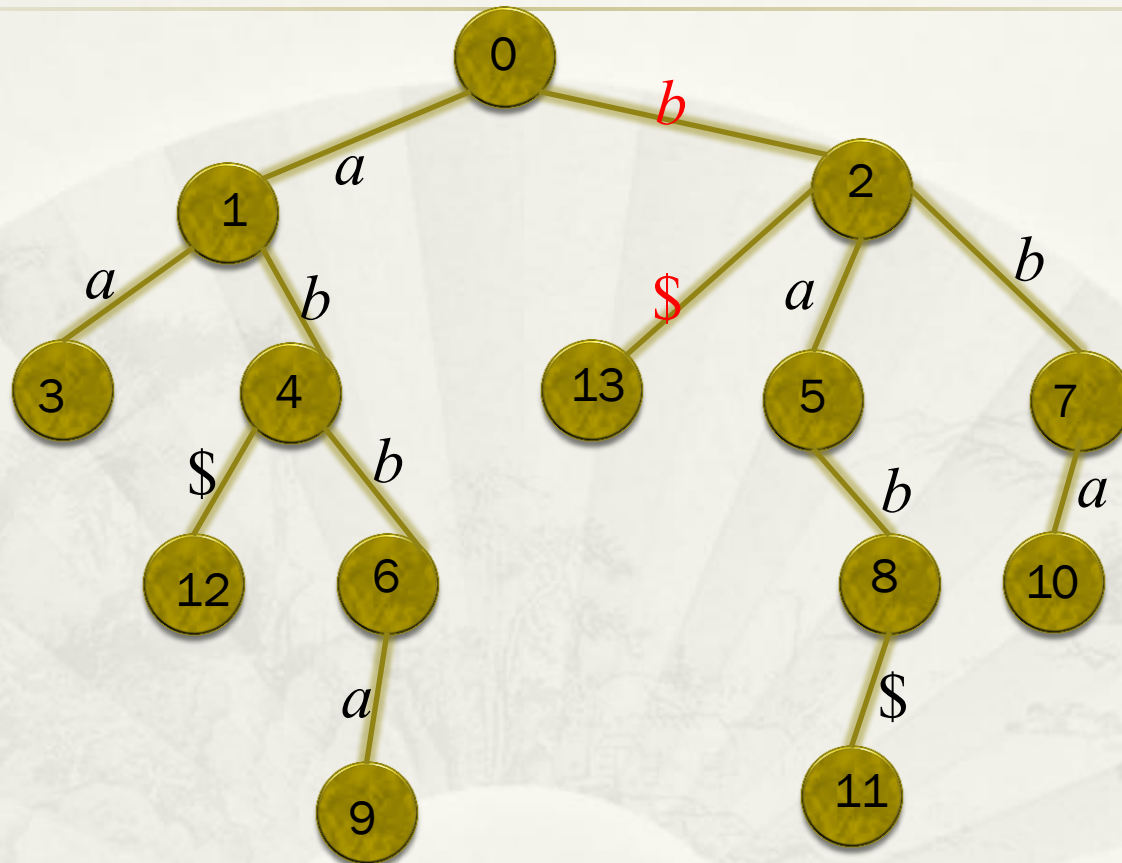
# Position Heap



A position heap for  $S = abaababbabbab\$$



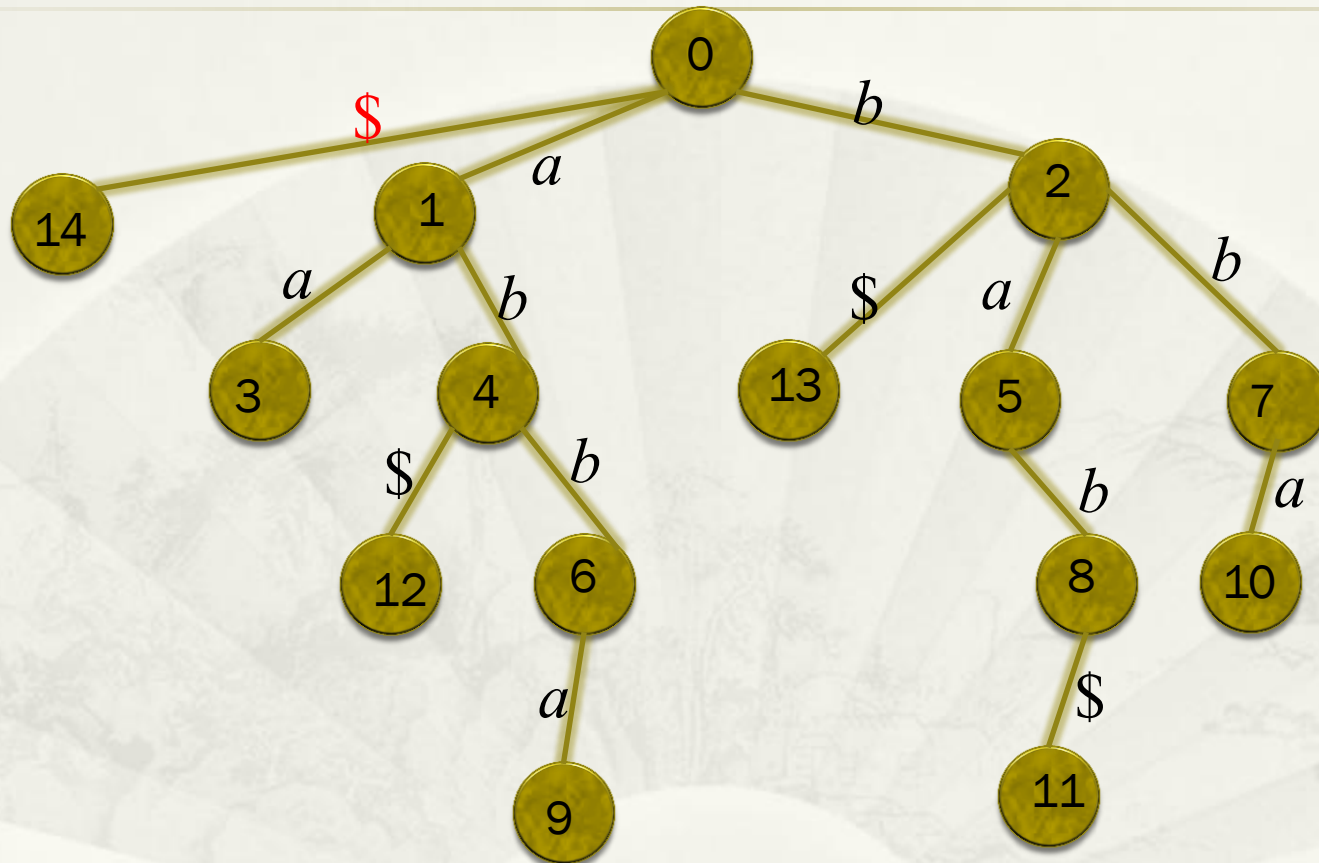
# Position Heap



A position heap for  $S = abaababbabba b\$$

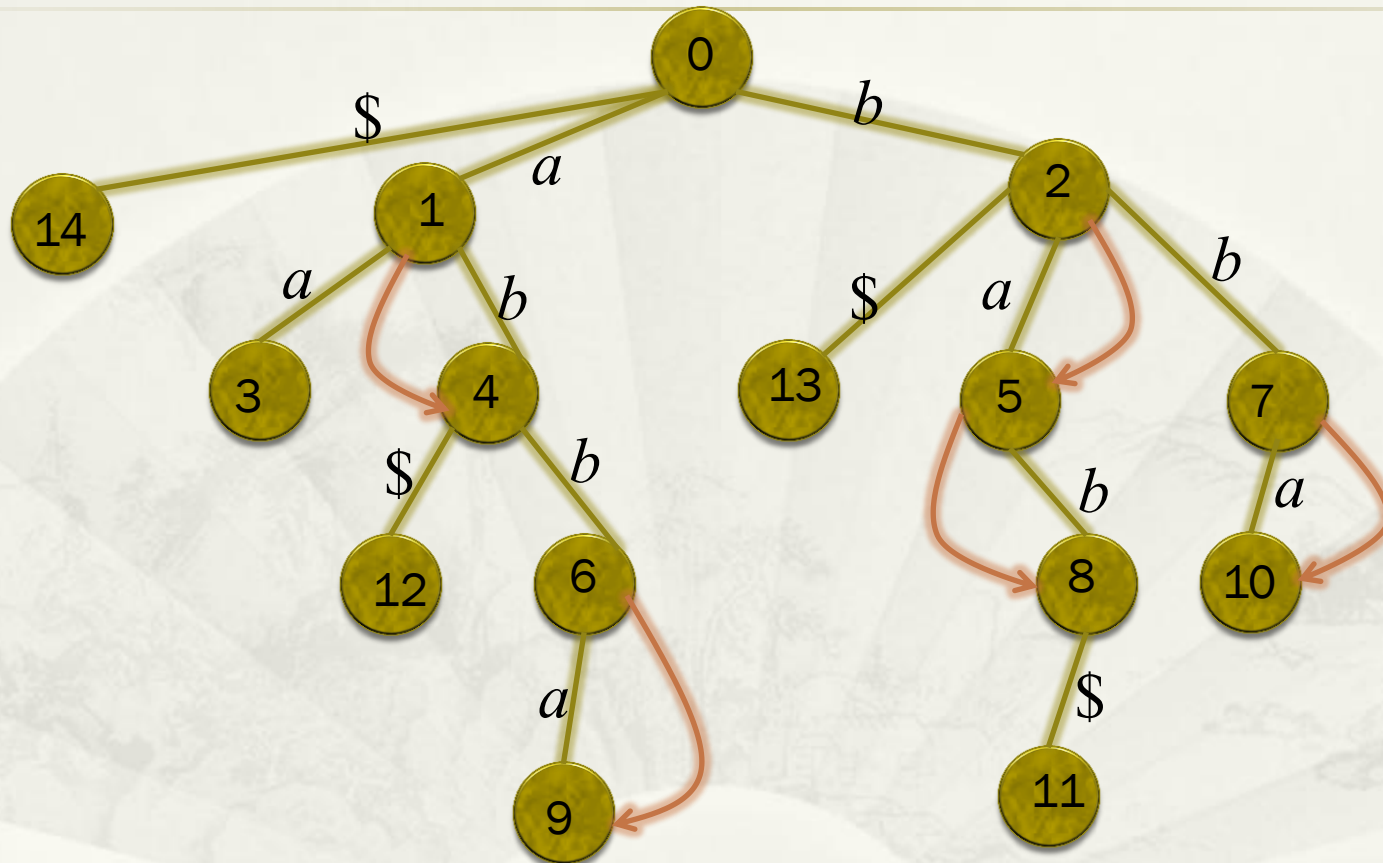


# Position Heap



A position heap for  $S = abaababbabbab\$$

# Position Heap

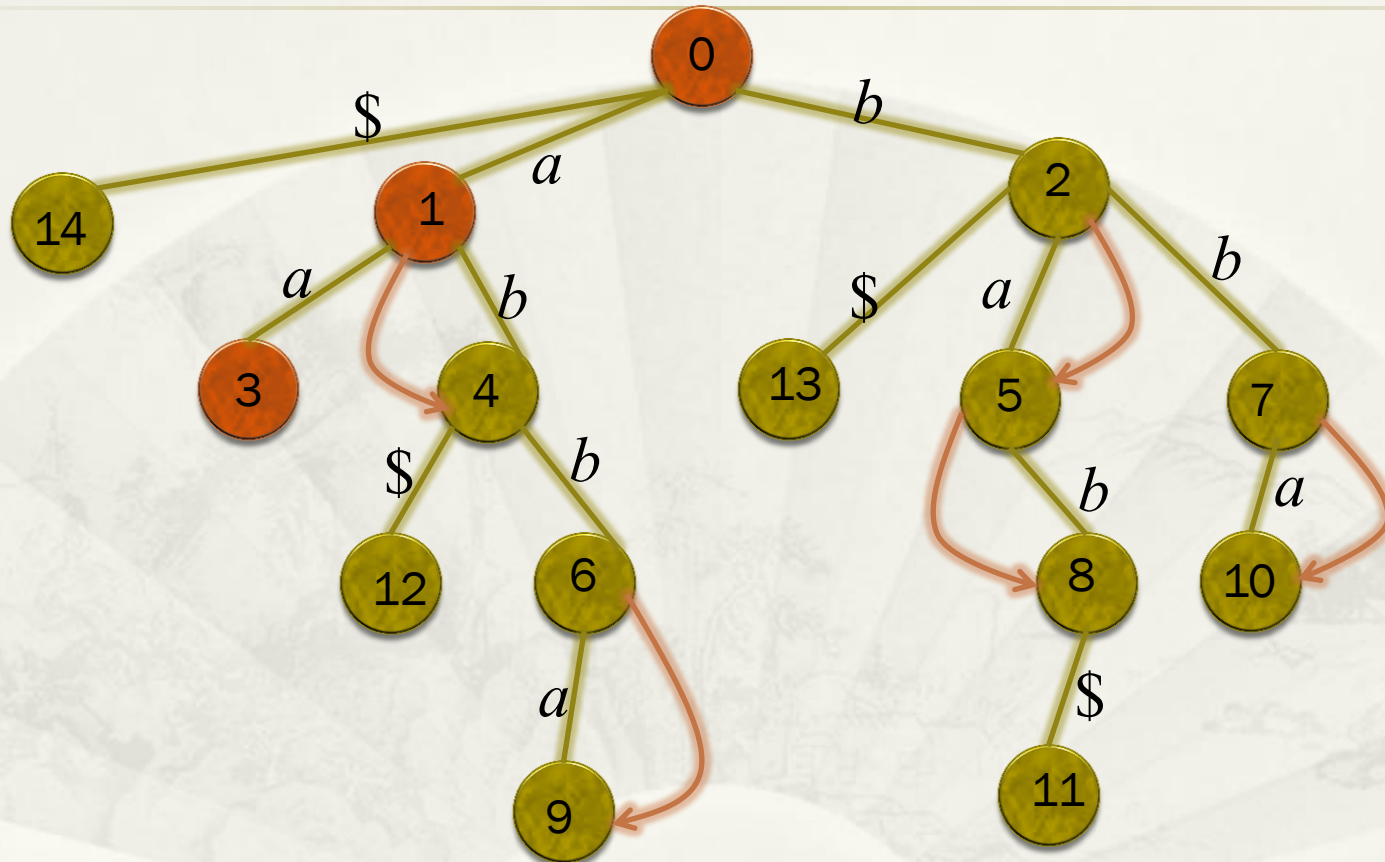


A position heap for  $S = abaababbabbab\$$

# Position Heap

- \* With the following auxiliary data structures, the pattern matching problem can be solved in optimal time (i.e.  $O(|P| + occ)$ ).
  - (1) An array  $A$  of pointers such that, given  $i$ , in  $O(1)$  time we can find the node labeled  $i$ .
  - (2) A data structure  $B$  such that, given  $i$  and  $j$ , in  $O(1)$  time we can determine whether the node labeled  $i$  is an ancestor of the node labeled  $j$ .

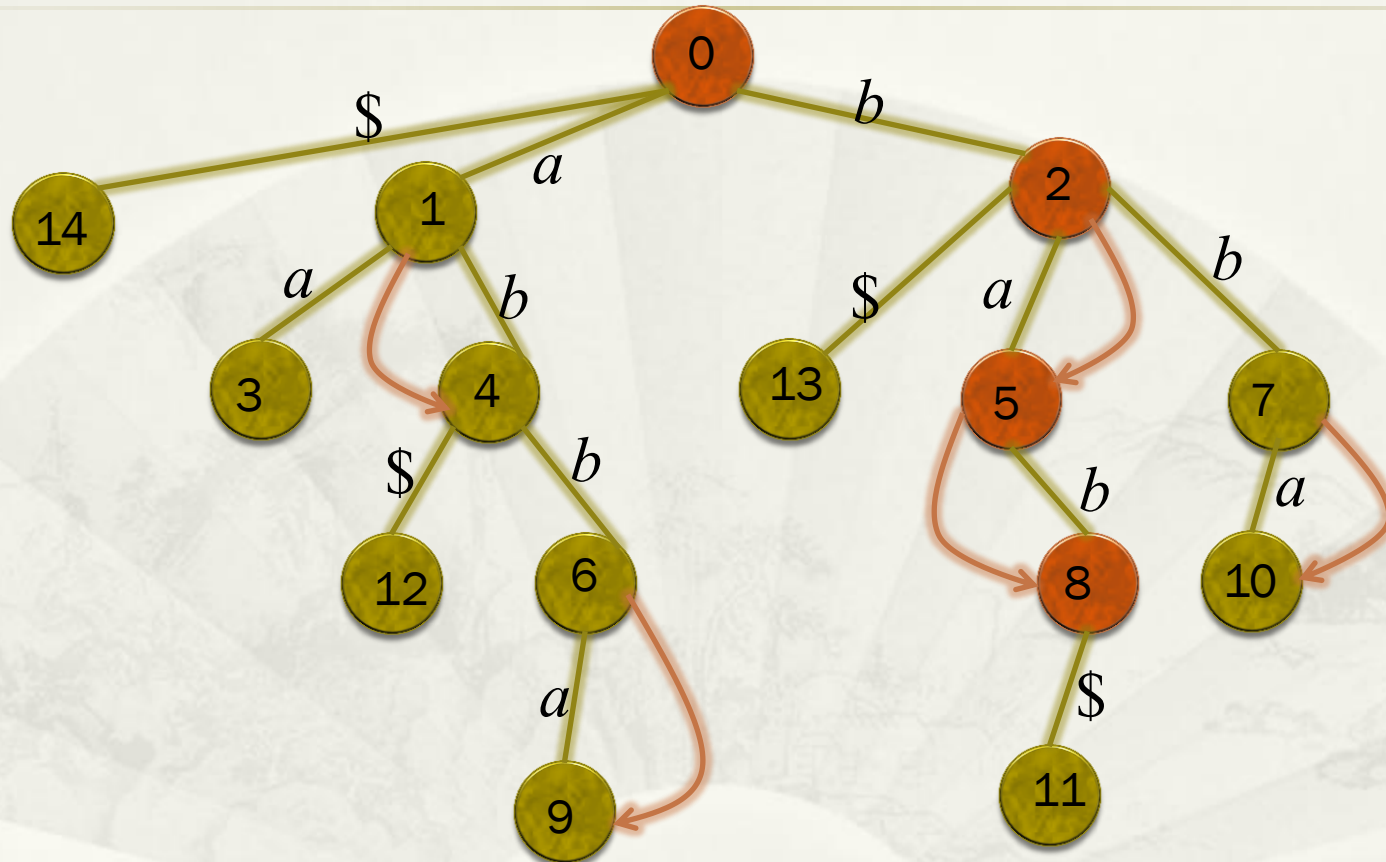
# Query on Position Heap



A position heap for  $S = ab\textcolor{red}{aab}abbabbab\$$

Query pattern  $P = \textcolor{red}{aa}bab$ , candidate are ~~1~~ and 3.

# Query on Position Heap



Query pattern  $P = aa**bab**$ , candidate is 3.

Check if  $3+2=5$  is on the path or in the subtree of 8.

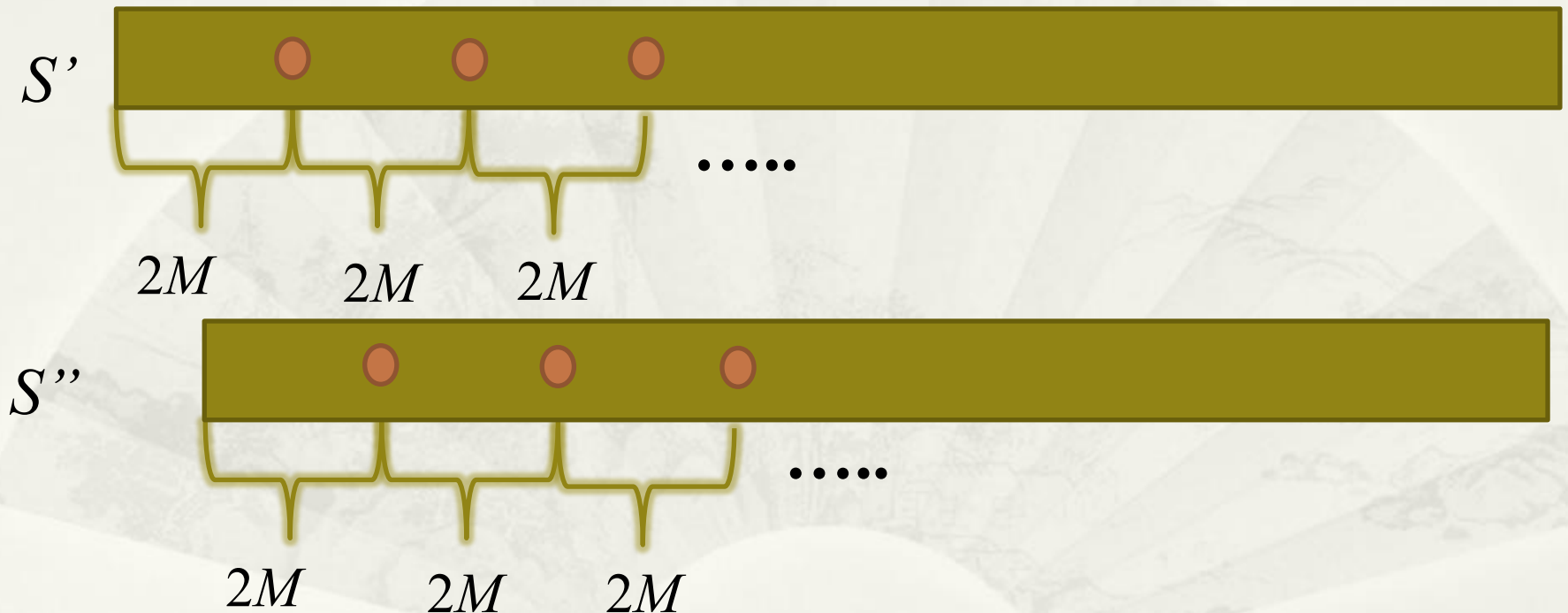
# Limiting Length and Height

---

- \* Suppose the length of the query pattern  $P$  is always less than or equal to a variable  $M$ .
- \* Then we can build a position heap whose height would be bounded in  $O(M)$ .

# Limiting Length and Height

- \* Suppose we want to build the position heap of a string



Instead of building the position heap on  $S$ , we build the position heap of  $S'!S''$ .



# Limiting Length and Height

According to the nature of the position heap, the height of the position heap of  $S'!S''$  would be  $O(M)$ .

By adopting Ehrenfeucht et al.'s approaches with an AVL tree, we have the following theorem.

**Theorem 1:** If we will never search for a pattern of length greater than  $M$  in a dynamic string  $S$ , then we can maintain a position heap that works as an index for  $S$  such that

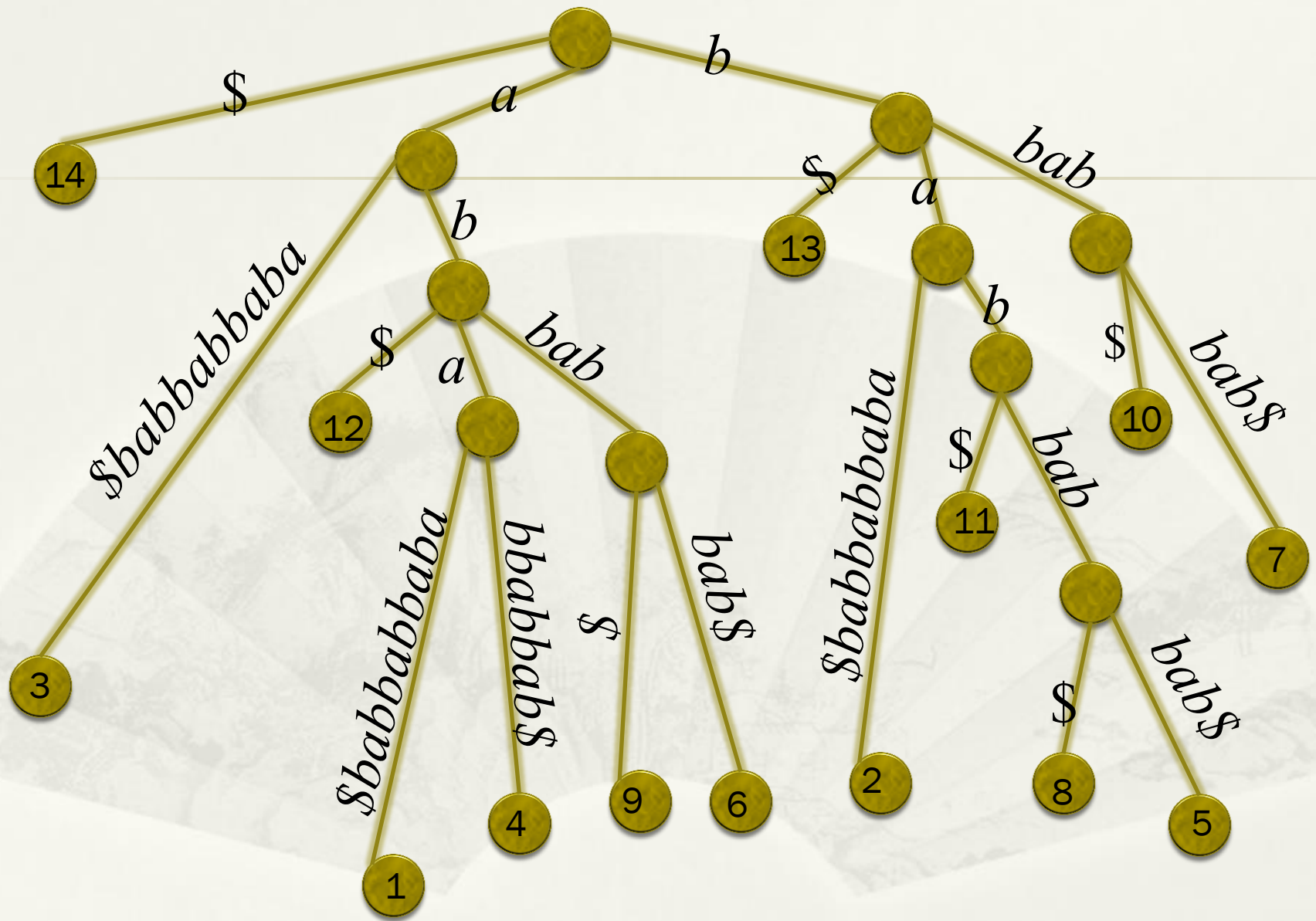
- Searching for a pattern of length  $m \leq M$  takes  $O(m \log |S| + occ)$  time,
- Inserting a substring of length  $l$  takes  $O((M+l)M \log(|S|+l))$  time,
- Deleting a substring of length  $l$  takes  $O((M+l)M \log |S|)$  time.



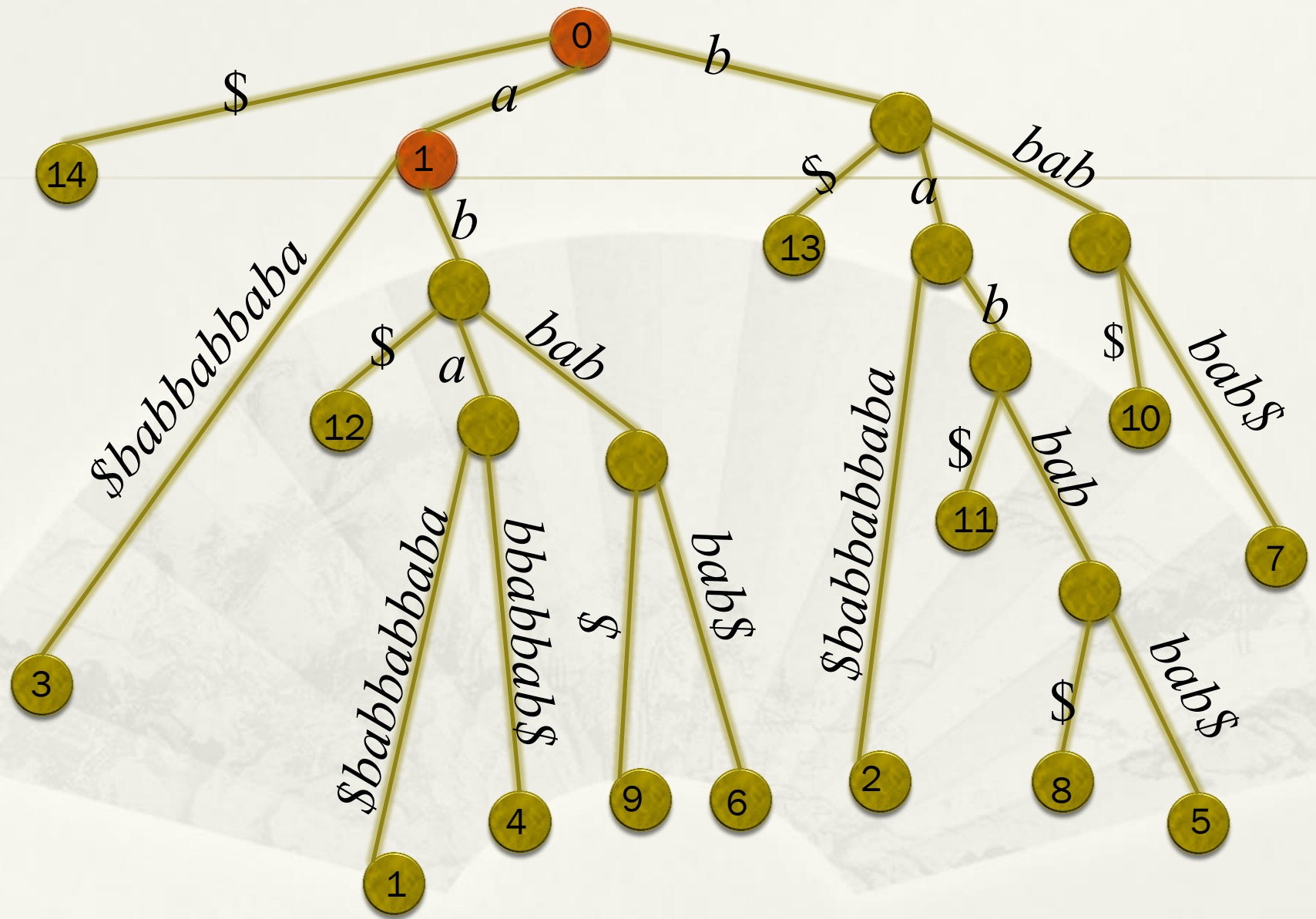
# Suffix Tree Into Position Heap

---

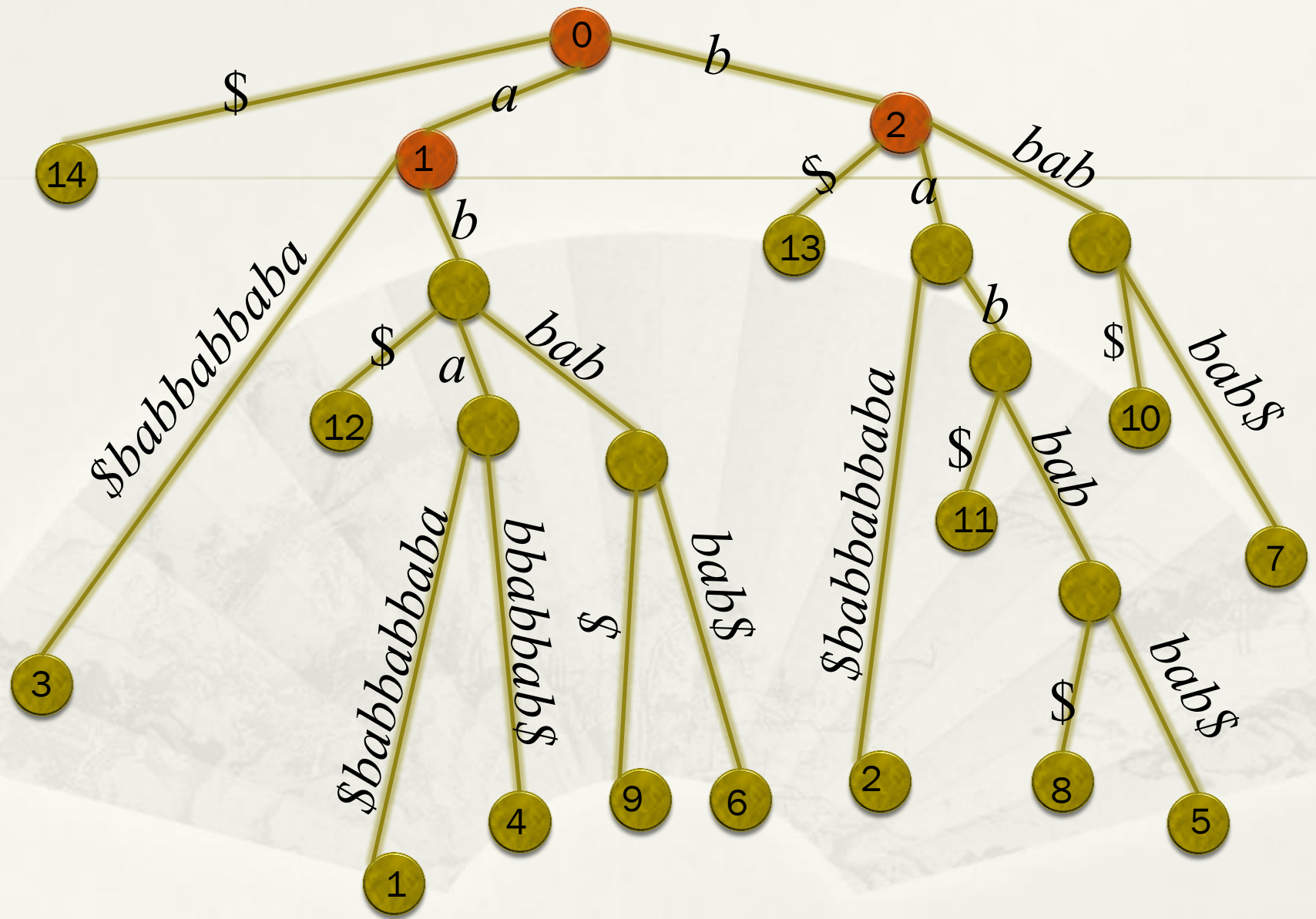
- \* A position heap of a string  $S$  is a subtree of the suffix trie of  $S$ .
- \* Suffix tree is a compact suffix trie.
- \* Here, we show how to turn a suffix tree into a position heap.



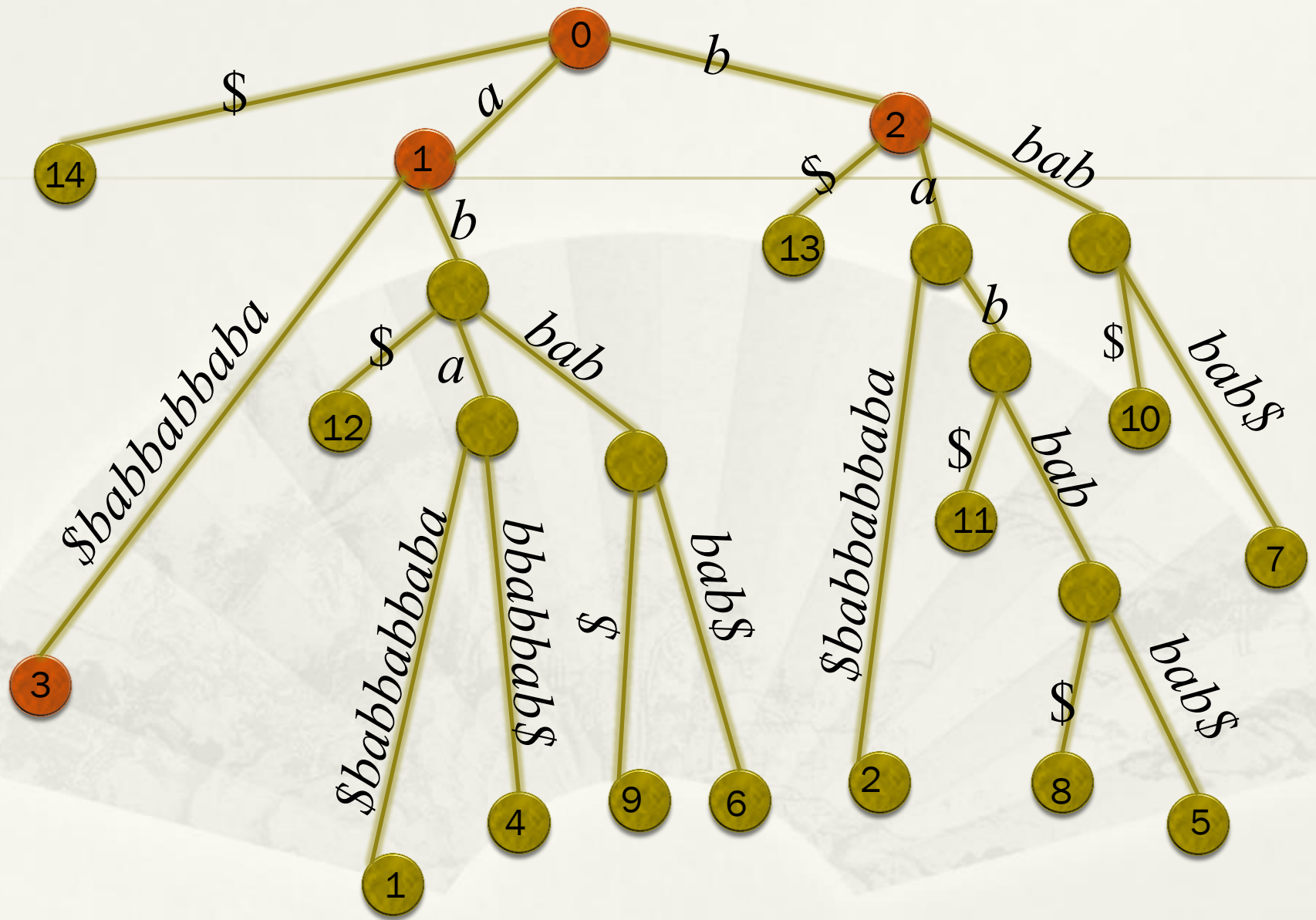
The suffix tree of  $S = abaababbabbab\$$ .



The suffix tree of  $S = abaababbabbab\$$ .

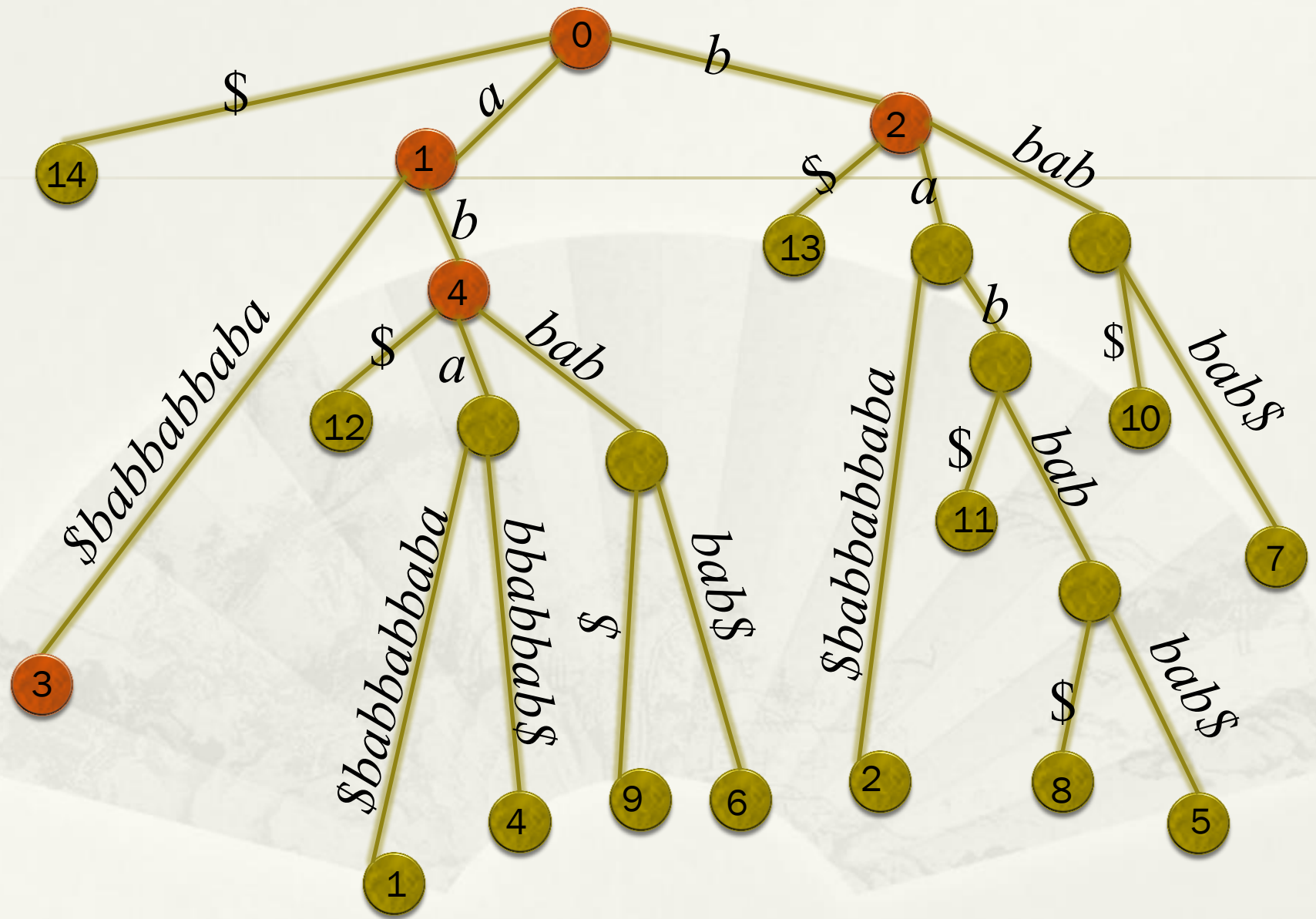


The suffix tree of  $S = abaababbabbab\$$ .

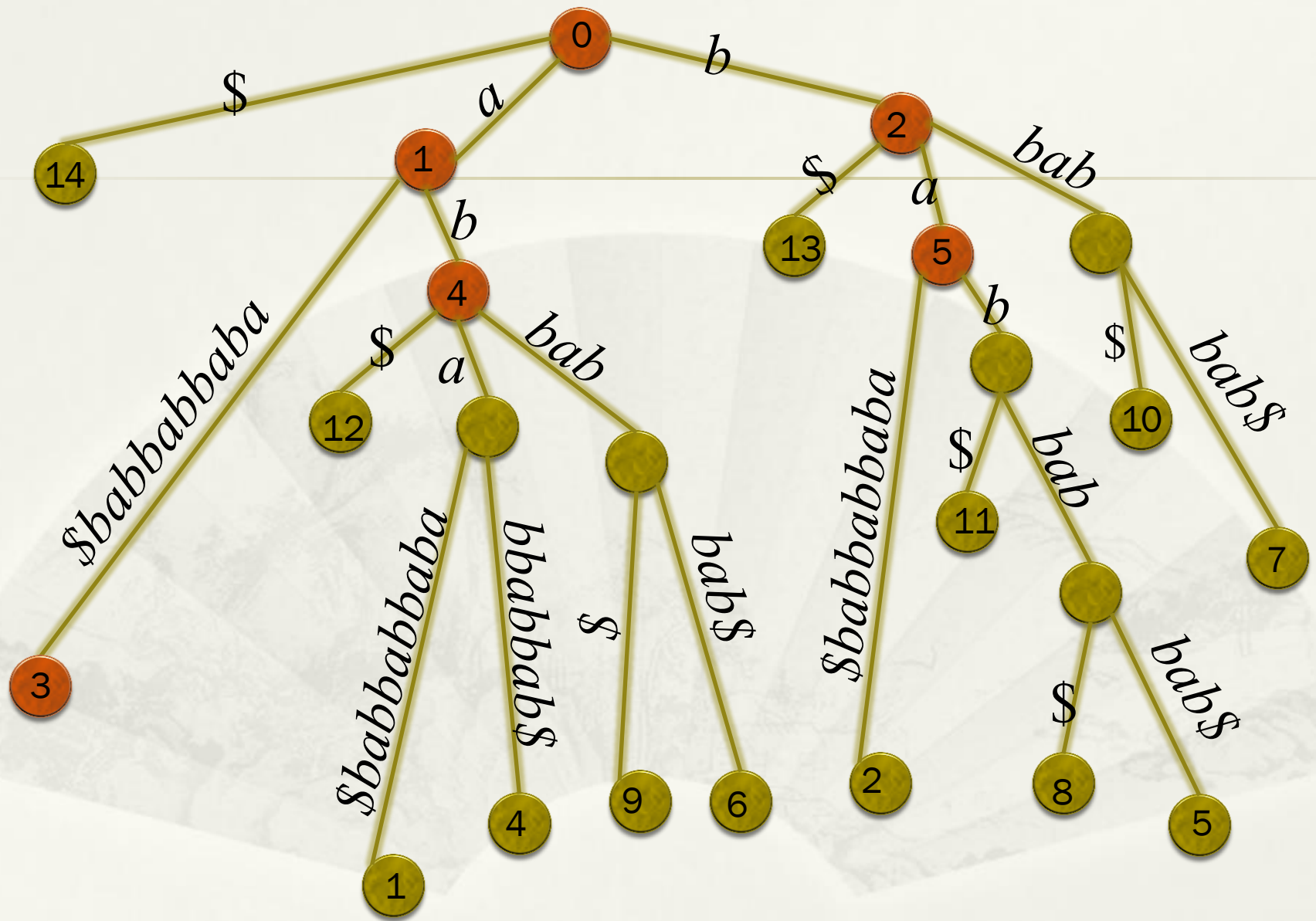


The suffix tree of  $S = abaababbabbab\$$ .

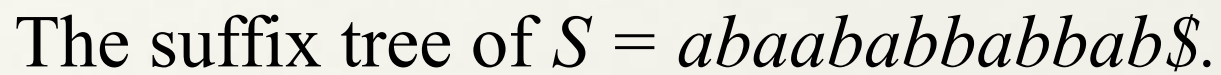




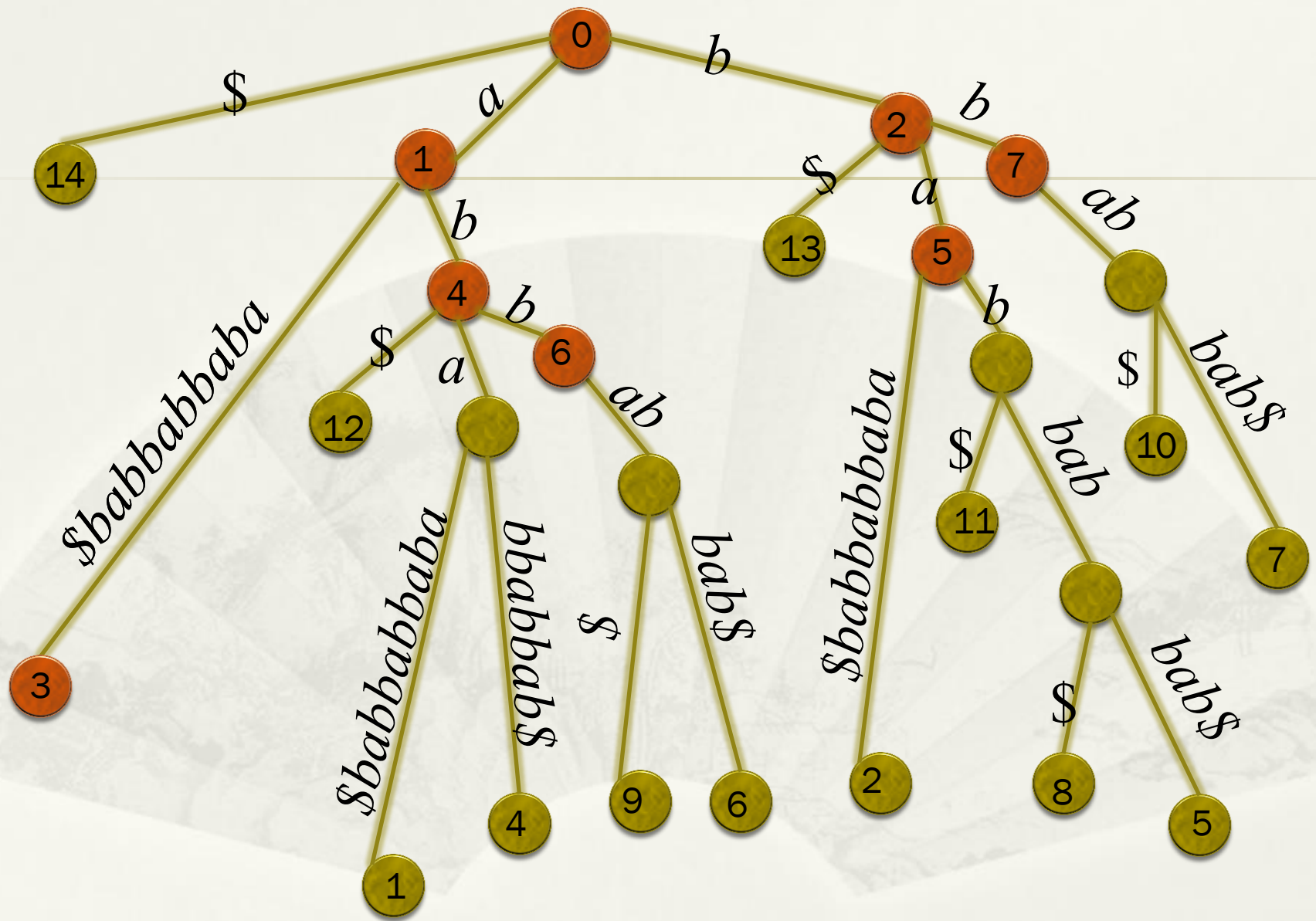
The suffix tree of  $S = abaababbabbab\$$ .



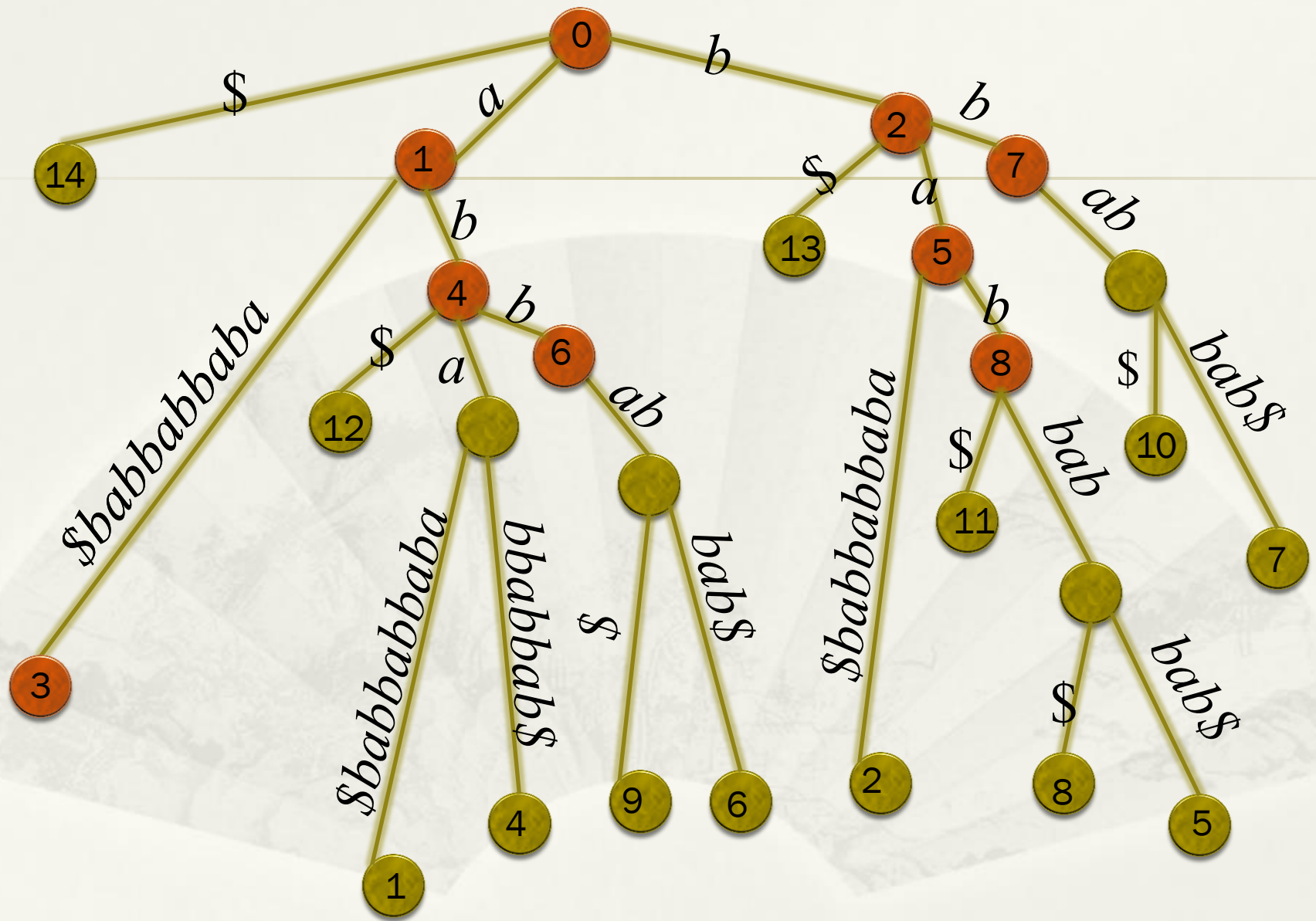
The suffix tree of  $S = abaababbabbab\$$ .



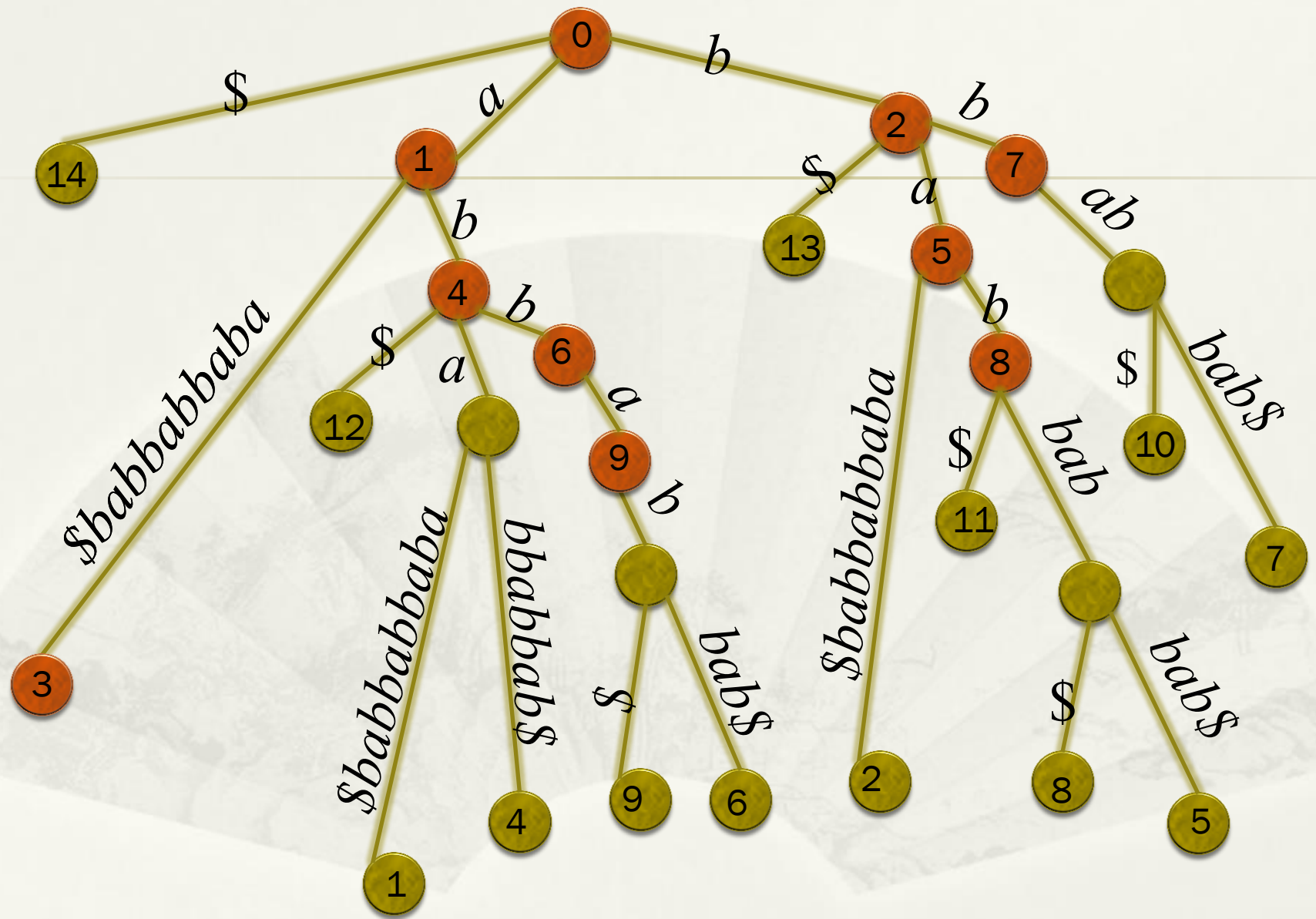




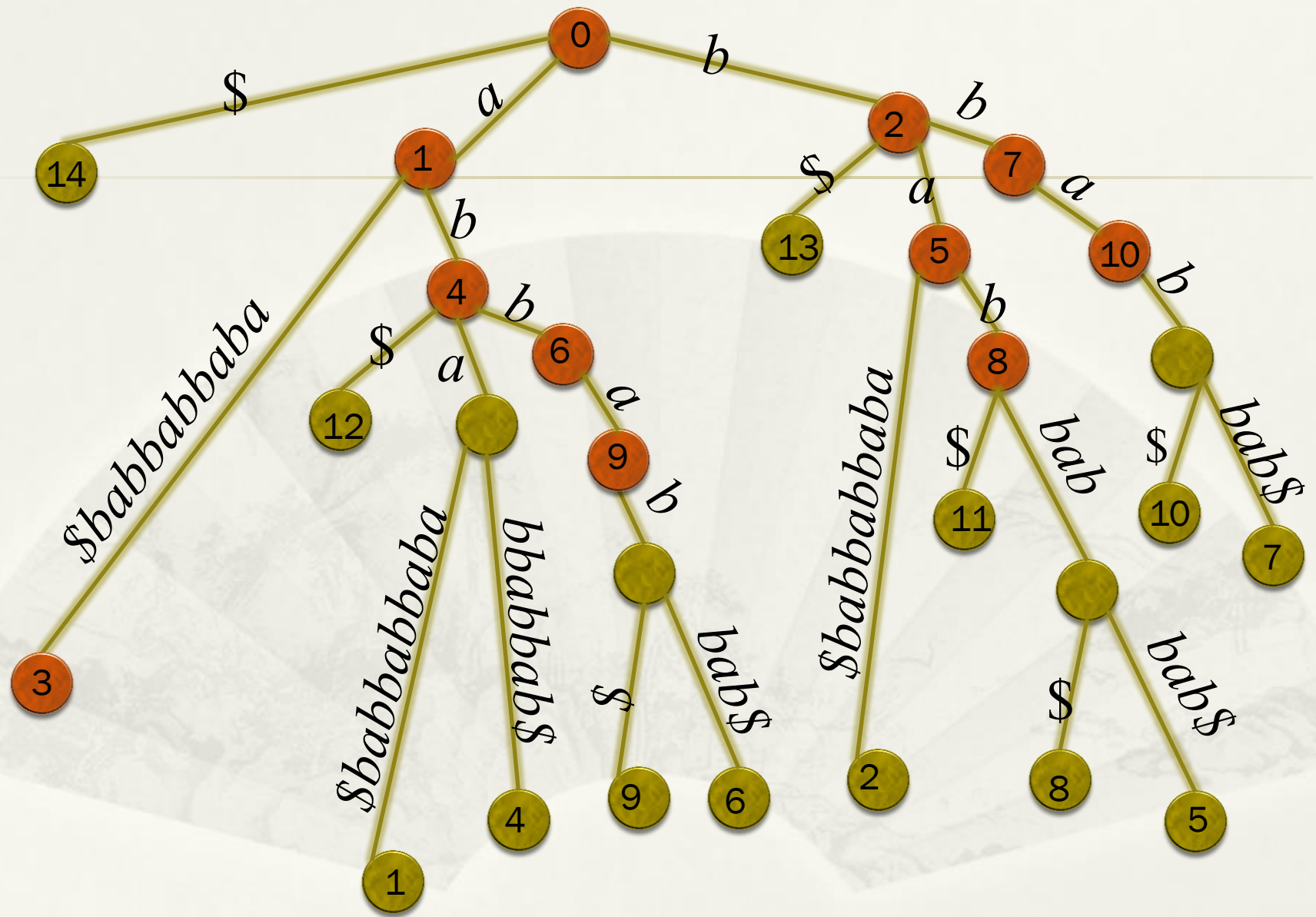
The suffix tree of  $S = abaababbabbab\$$ .



The suffix tree of  $S = abaababbabbab\$$ .

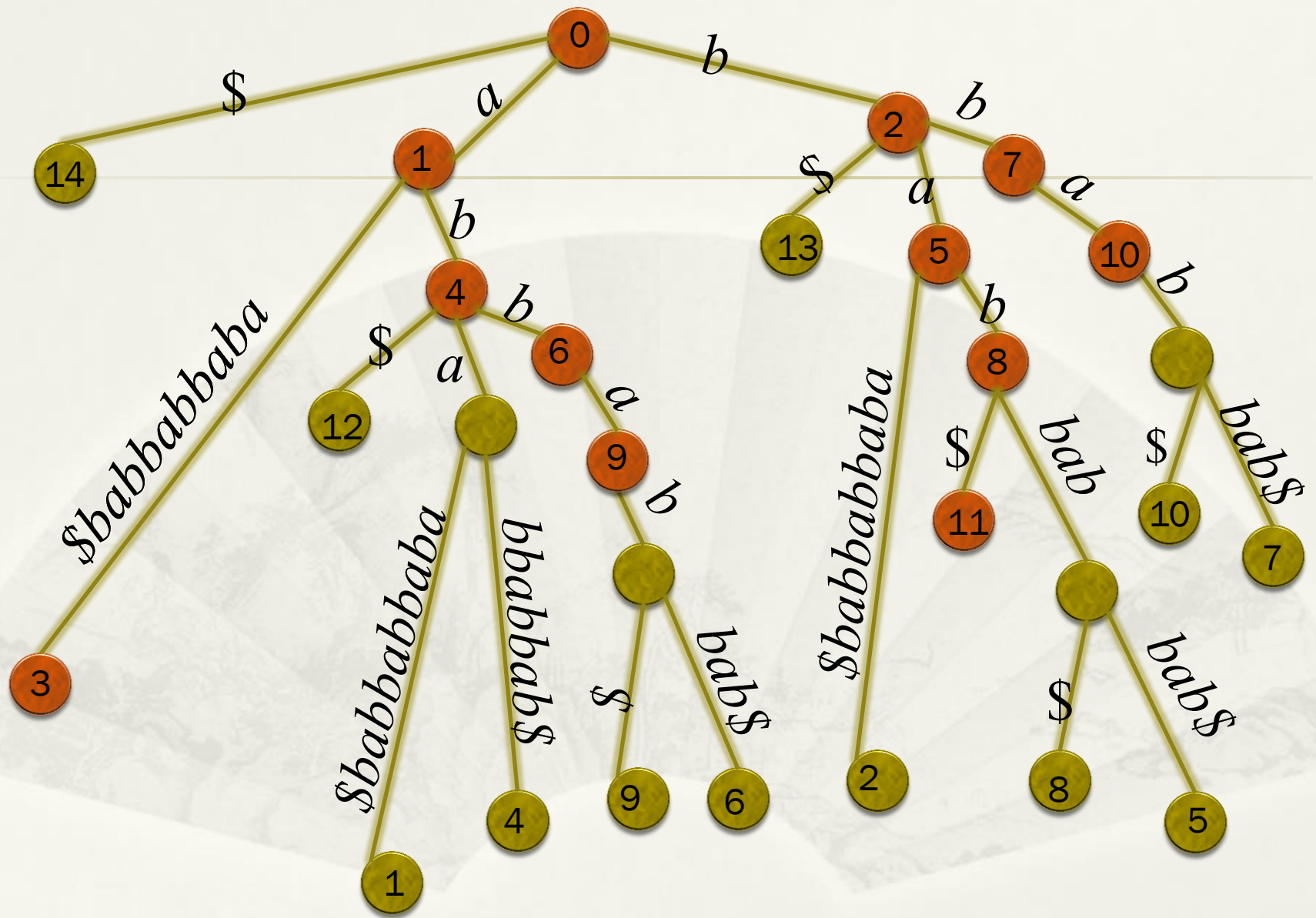


The suffix tree of  $S = abaababbabbab\$$ .

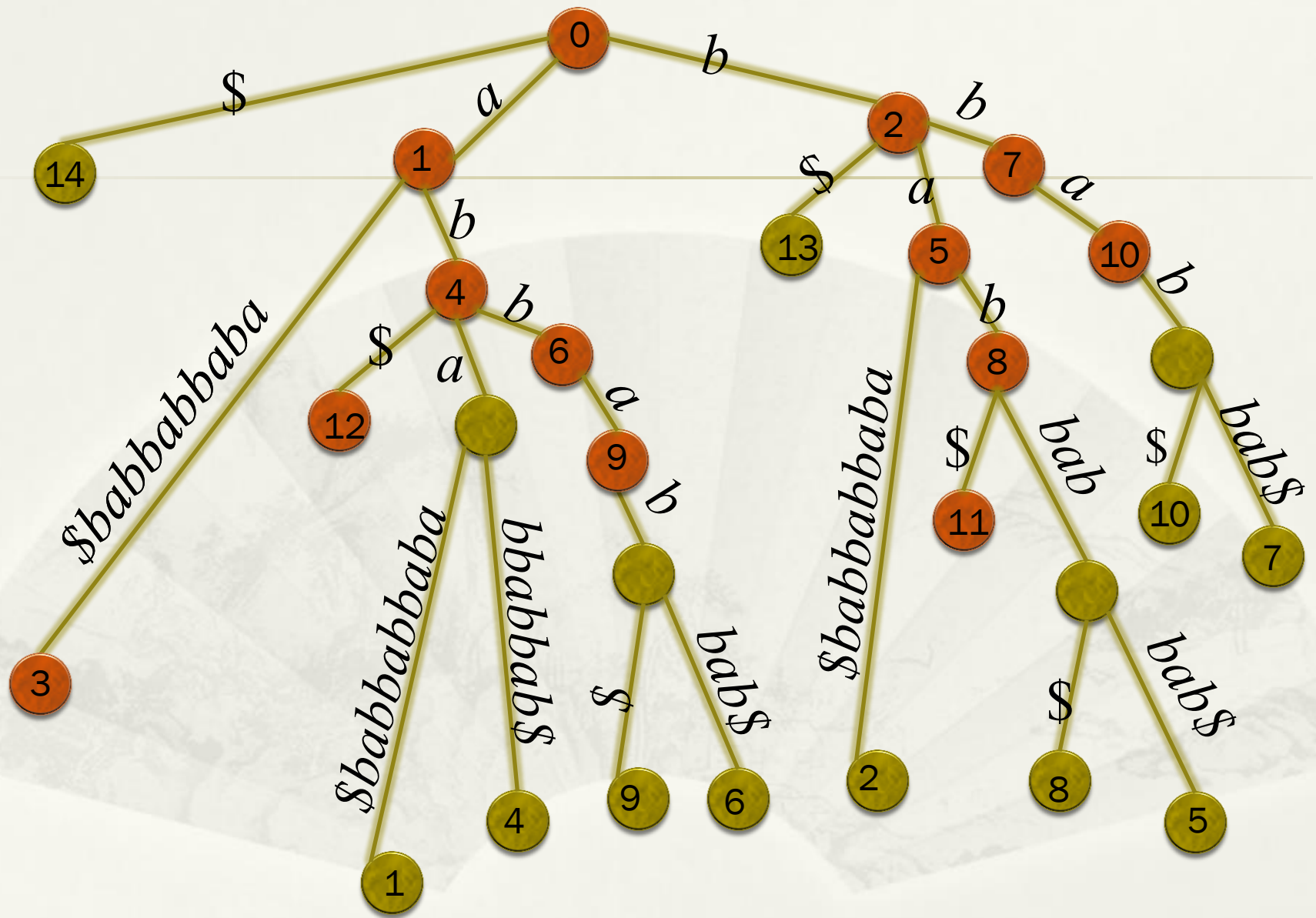


The suffix tree of  $S = abaababbabbab\$$ .

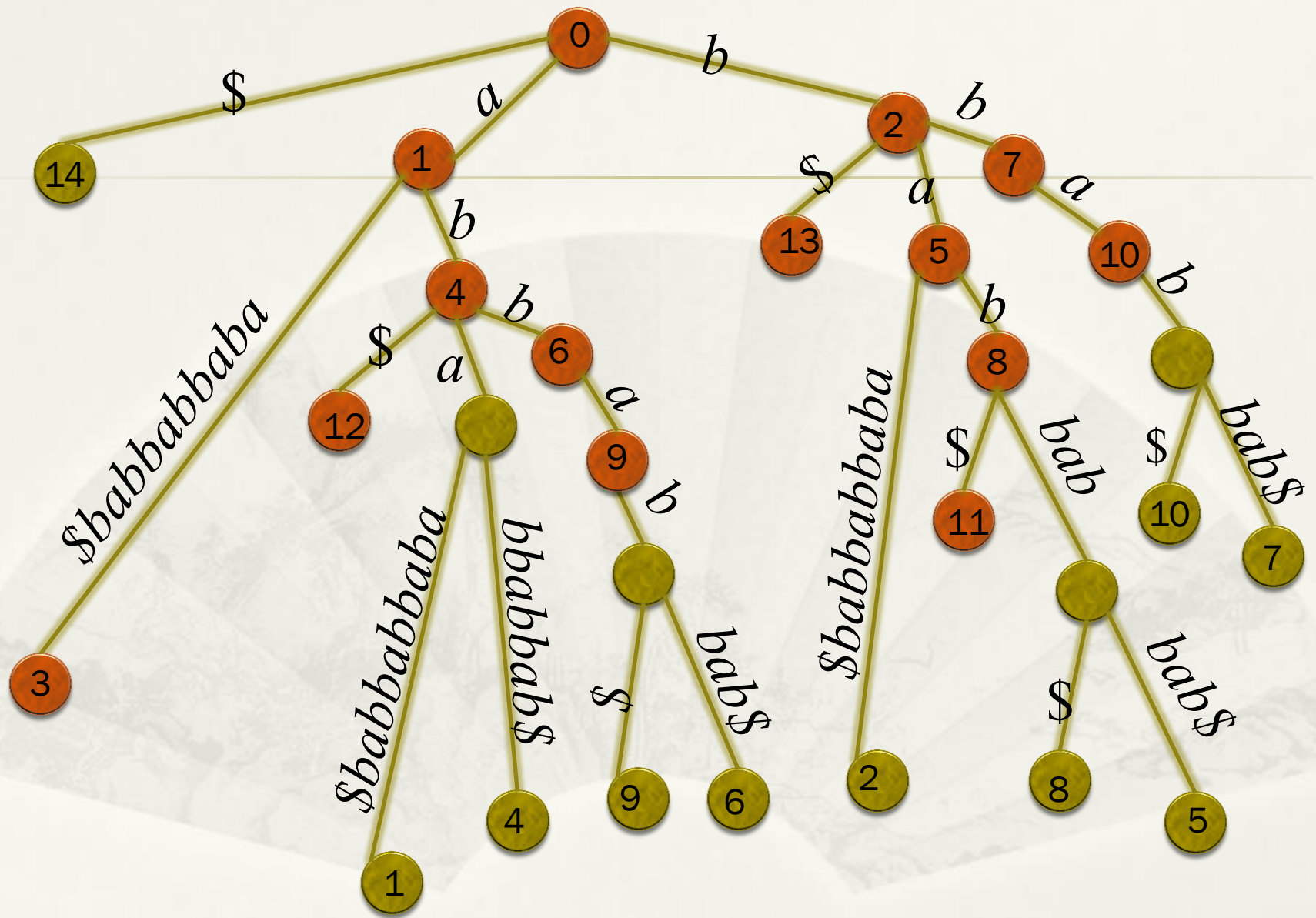




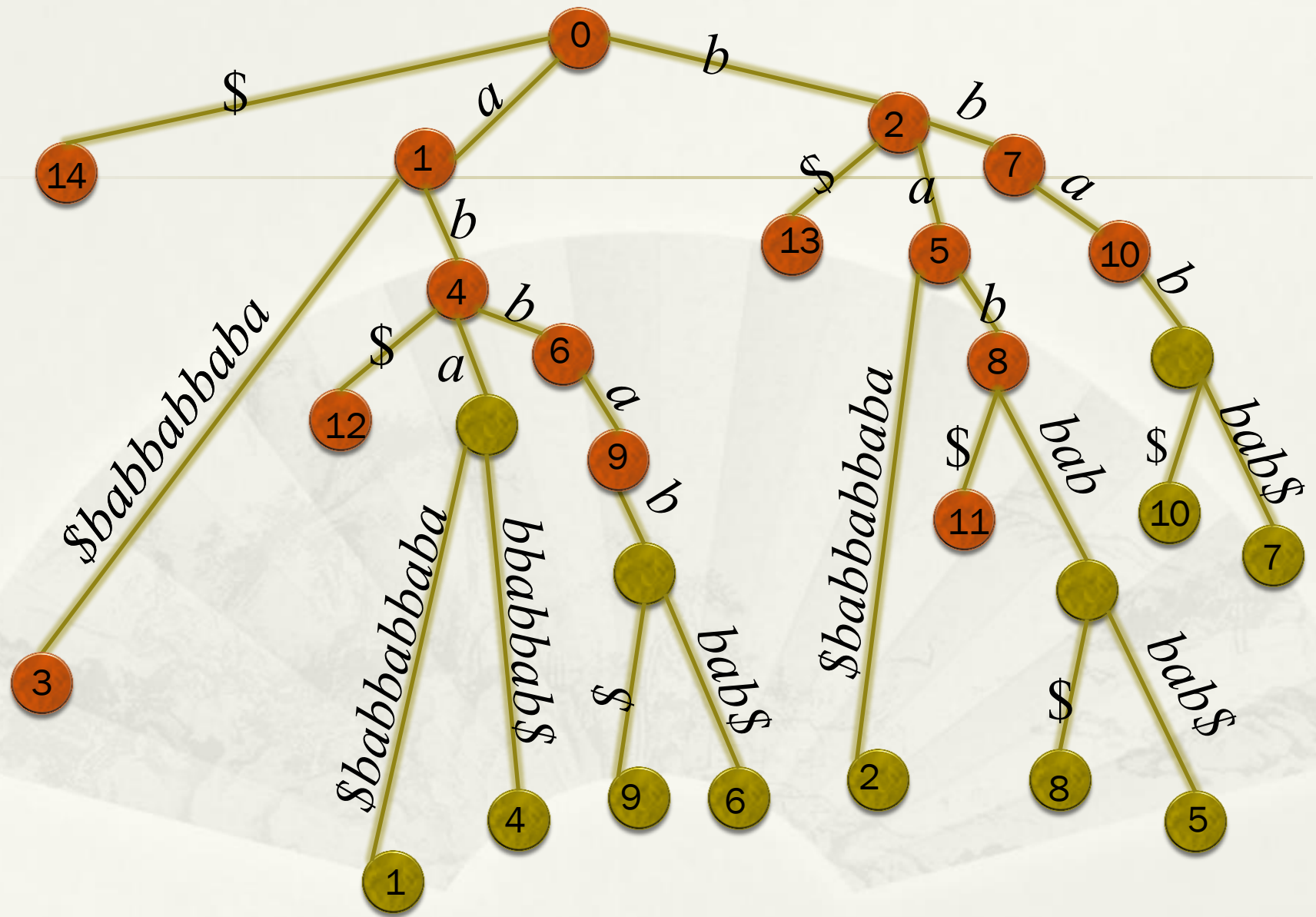
The suffix tree of  $S = abaababbabbab\$$ .



The suffix tree of  $S = abaababbabbab\$$ .

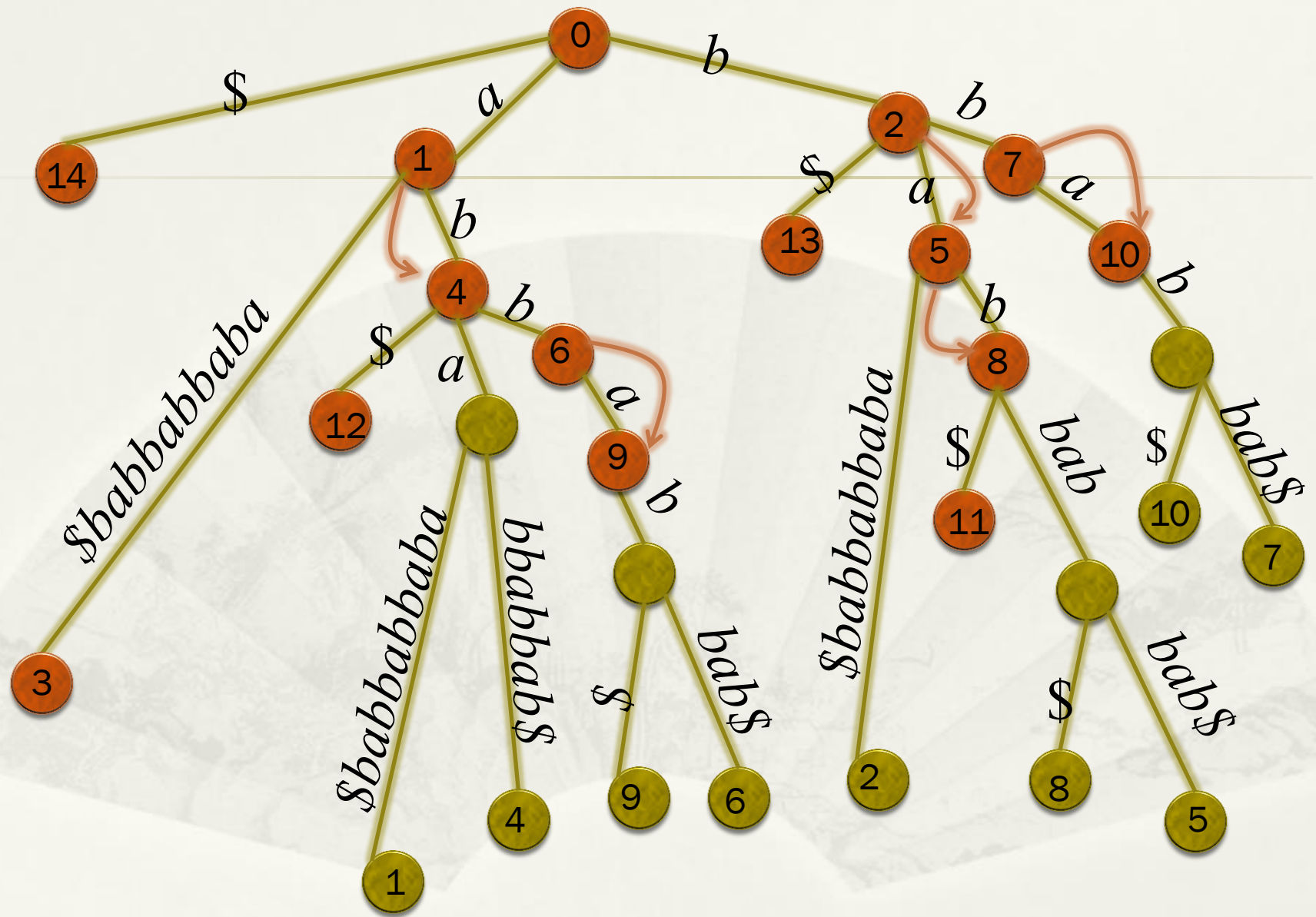


The suffix tree of  $S = abaababbabbab\$$ .



The suffix tree of  $S = abaababbabbab\$$ .





The suffix tree of  $S = abaababbabbab\$$ .

# Suffix Tree Into Position Heap

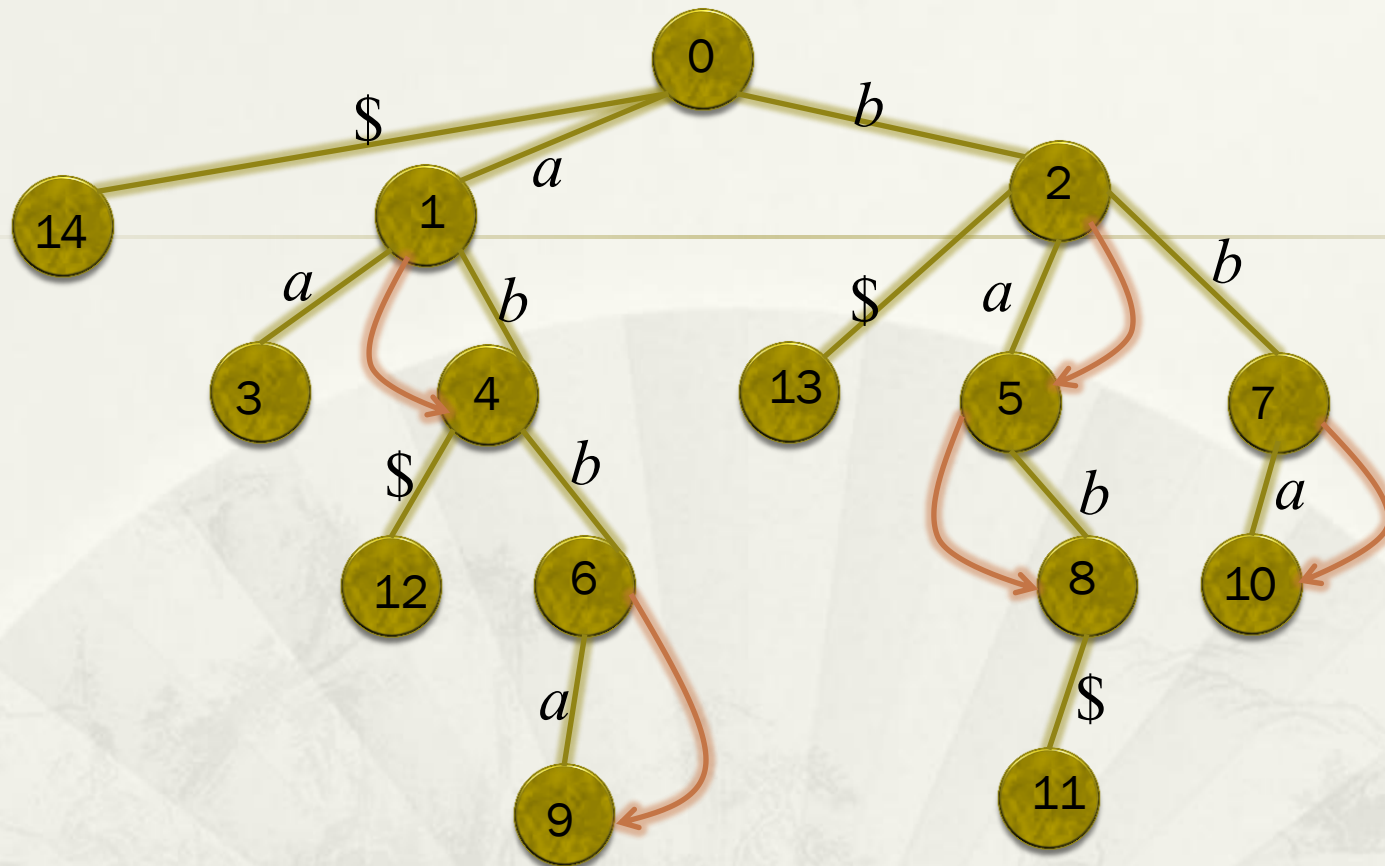
---

- \* Given a suffix tree, we can build the position heap in linear time by using Westbrook's approaches, and Berkman et al.'s approaches.
- \* By using Farach's linear time suffix tree construction algorithm independent of the alphabet size, we can build a position heap of a string  $S$  in linear time independent of the alphabet size.

# Using Position Heap as Suffix Array

---

- \* Define  $D[i]$  = the depth of the node labeled  $SA[i]$
- \* Define  $E[i]$  = the depth of the node with rank  $i$  in preorder



+

$D = [1, 2, 3, 1, 2, 4, 3, 2, 1, 4, 3, 2, 3, 2]$

$E = [1, 1, 2, 2, 3, 3, 4, 1, 2, 2, 3, 4, 2, 3]$

=

$SA = [14, 3, 12, 1, 4, 9, 6, 13, 2, 11, 8, 5, 10, 7]$

# Using Position Heap as Suffix Array

---

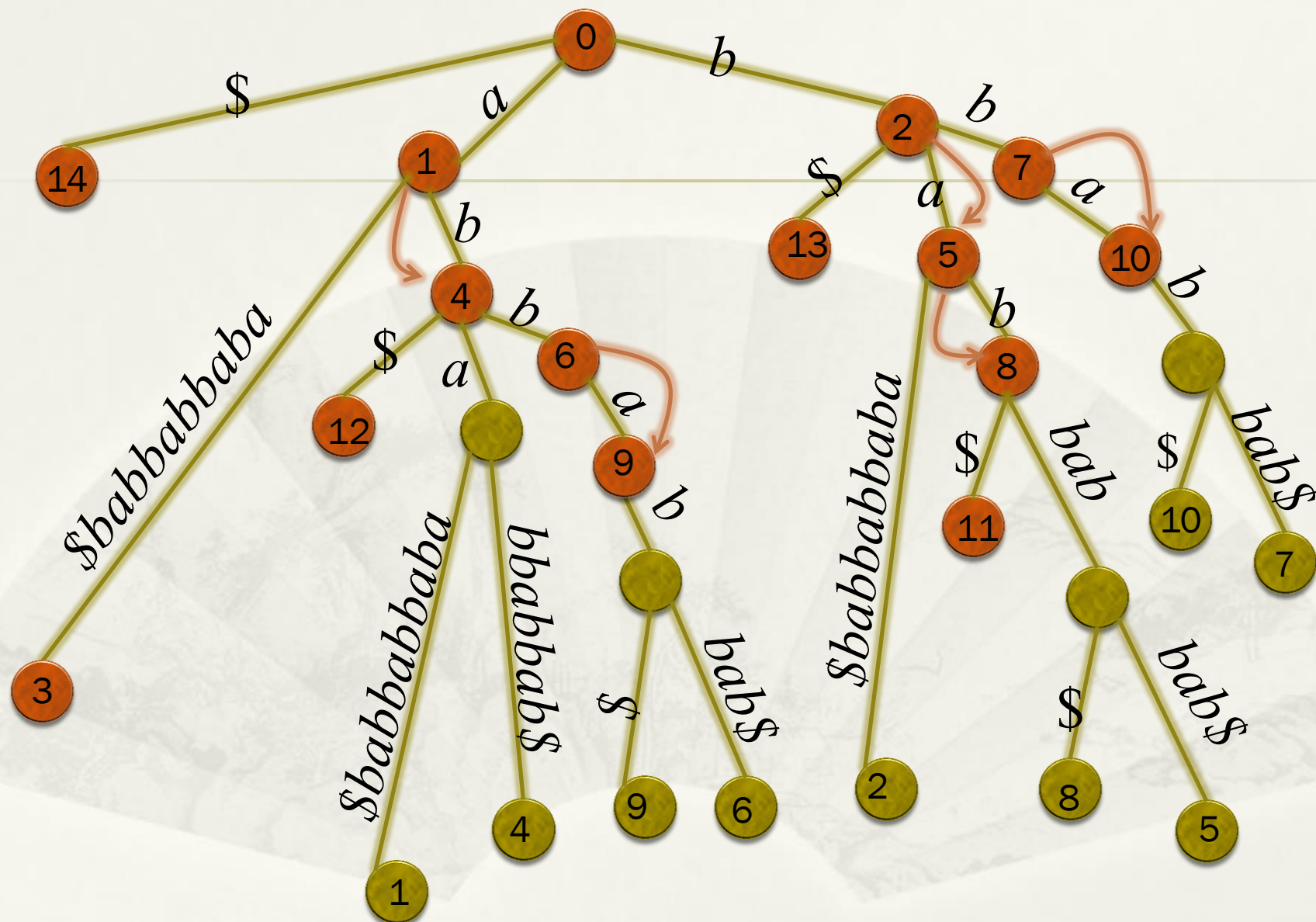
- \* By using succinct partial rank and select query structures which support constant time query on arrays  $D$  and  $E$ , we can compute the  $SA[i]$  value in constant time.
- \* Theorem: We can add  $O(n \log h)$  bits to a position heap, where  $h$  is the height of the heap, such that it supports access to the corresponding suffix array and inverse suffix array in  $O(1)$  time.

# Using CSA as Position Heap

To represent a position heap, we need

- \* Its structure as a tree; ➔ balanced parentheses representation
- \* The nodes' labels; ➔ by using  $E$ ,  $D$ , and  $SA$
- \* The edges' labels; ➔ by using  $SA^{-1}$  and a bit-vector
- \* The maximal-reach pointers; ➔ next slide, we will show
- \* The array  $A$  of pointers; ➔ similar to previous one described in the previous slides
- \* The data structure  $B$ ; ➔ similar to previous one described in the previous slides






$((*)((*)((*)**((**))))((*)(*(*)**))((**))))$



# Using CSA as Position Heap

$((*)((*)((*)^{**}(\textcolor{violet}{*}\textcolor{red}{*})))((*)(*(\textcolor{violet}{*})^{**}))((^{**})))$



This pair represents the node labeled 9.

Suppose we want to compute the maximal-reach pointer of the node labeled 6.

First we compute  $SA^{-1}[6] = 7$ . ; it is the 7-th star we visited

Then we compute which parentheses pair encloses the 7-th star.

---



Thank you