Approximate Pattern Matching Using Suffix Tries

Hendrik Nigul

nigulh@math.ut.ee

University of Tartu

Overview

- Introduction, problem description
- Suffix tries
  - What is a suffix trie
  - How to create suffix tries
  - How to use suffix tries
- Algorithms with suffix tries
  - Exact string matching
  - Approximate string matching
  - Exact all-against-all matching
  - Approximate all-against-all matching
- Results
- Conclusions

Introduction

Problem statement:

Given text $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$, find all occurrences of P in T.

By an *occurrence* we mean a position i, such that

$$t_{i+1} = p_1, t_{i+2} = p_2, \dots, t_{i+m} = p_m$$

Introduction

Problem statement:

Given text $T = t_1 t_2 \ldots t_n$ and pattern $P = p_1 p_2 \ldots p_m$, find all occurrences of P in T.

By an *occurrence* we mean a position i, such that

$$t_{i+1} = p_1, t_{i+2} = p_2, \ldots, t_{i+m} = p_m$$

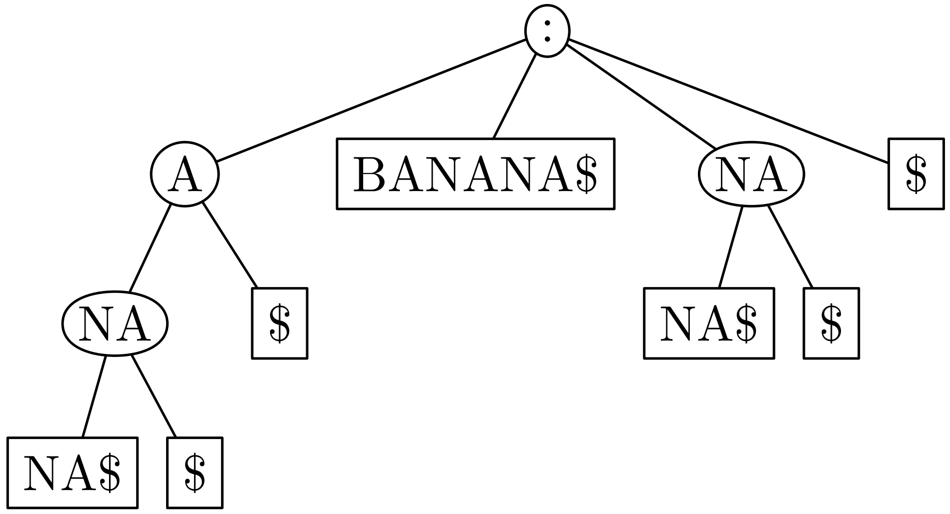
- Sometimes we have have several patterns:
  - Find occurrences of BANANA in text T
  - Find occurrences of ANANAS in text T
  - $\ldots$

Introduction

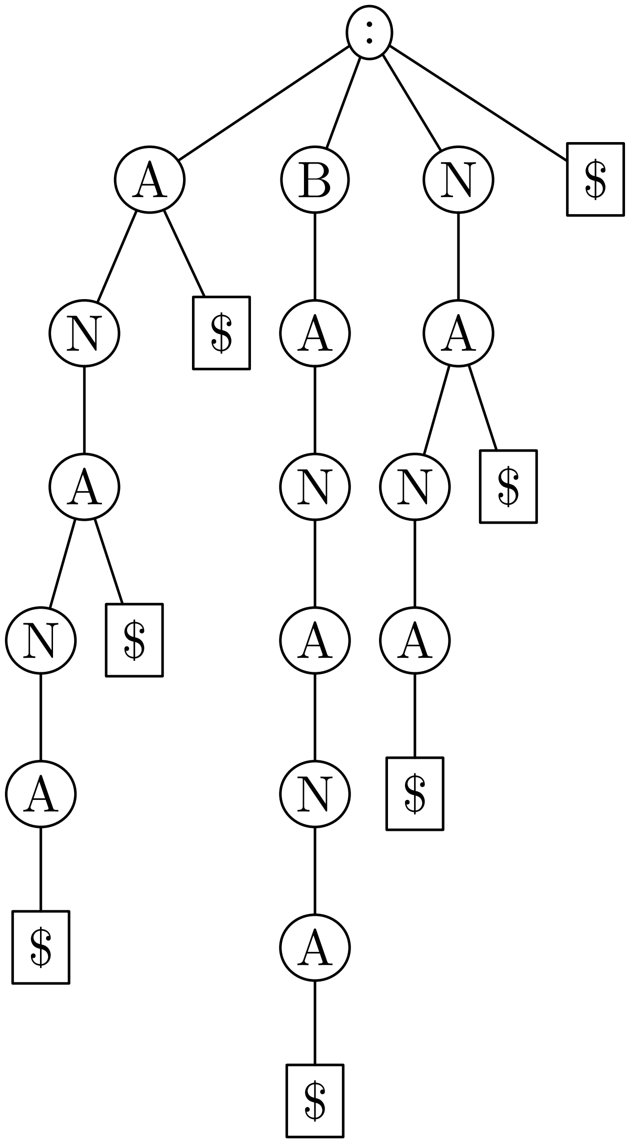
- Sometimes we accept approximate matches:
  - Find occurrences of `BANANA`, but also accept `MANANA`, `BANAANA`, `BAANA`, etc.
- If we make several queries, we should preprocess our text.
- We use suffix tries.

Suffix trie

Example: Create suffix trie for text BANANA



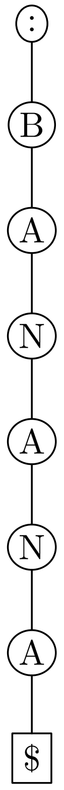
Suffix tree



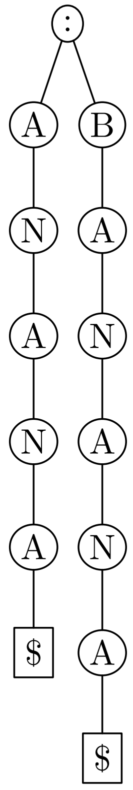
Suffix trie

Indexing

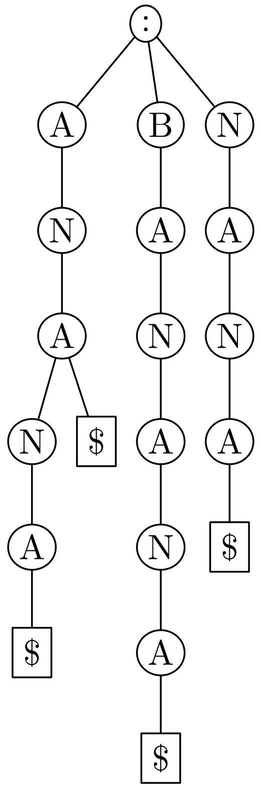
All suffixes are added to the trie one by one.



Inserting BANANA$



Inserting ANANA$



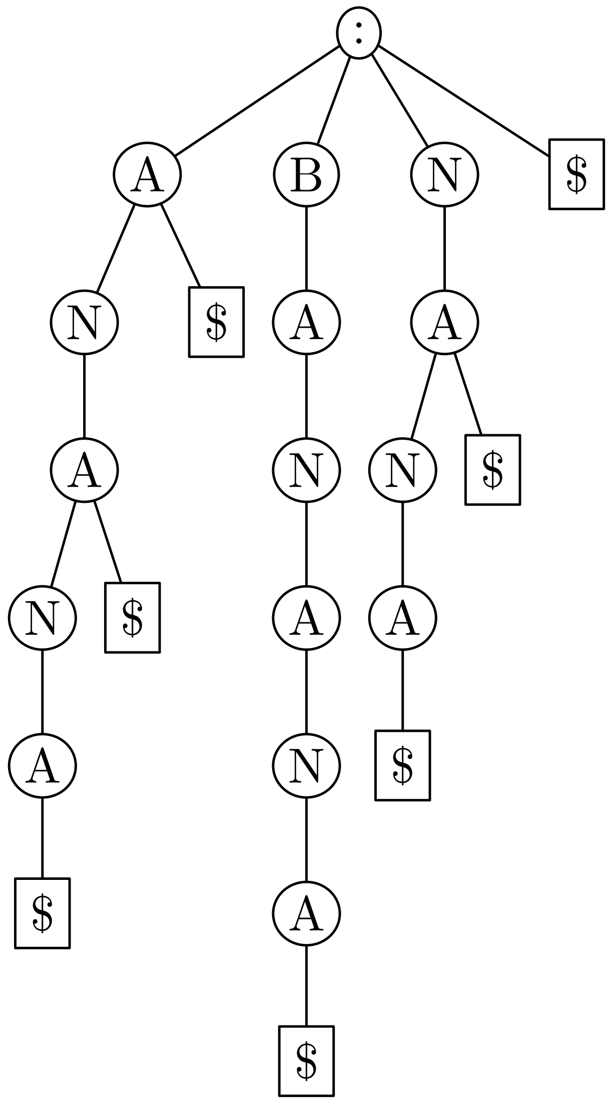
Inserting NANA$ and ANA$

Outputting index to a file

- We want to use the index many times.
- We want to write it into a file.
- Later we must be able to read that trie from file.
- We output the trie in *prefix* order, i.e. we output a node first, and then its children.
- We need to calculate the *size* of each node, that is *the number of bytes of the description of the subtree rooted with that node*

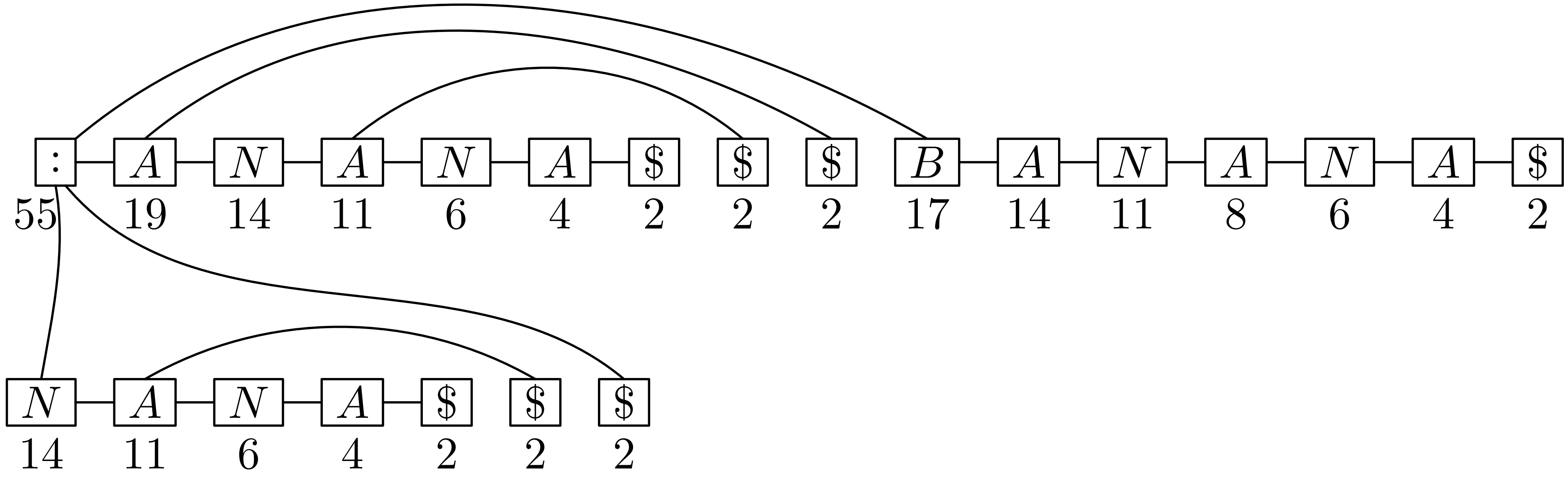
Outputting index to a file

- Suffix trie for BANANA contains suffixes
  - BANANA$
  - ANANA$
  - NANA$
  - ANA$
  - NA$
  - A$
  - $



Outputting index to a file

The suffix trie for BANANA

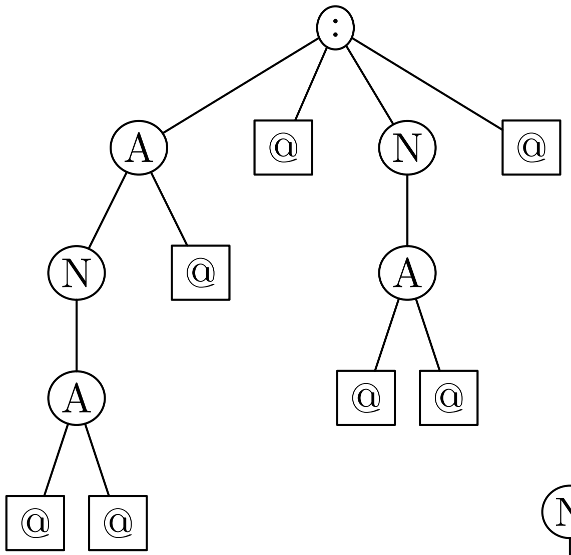


The index written to a file

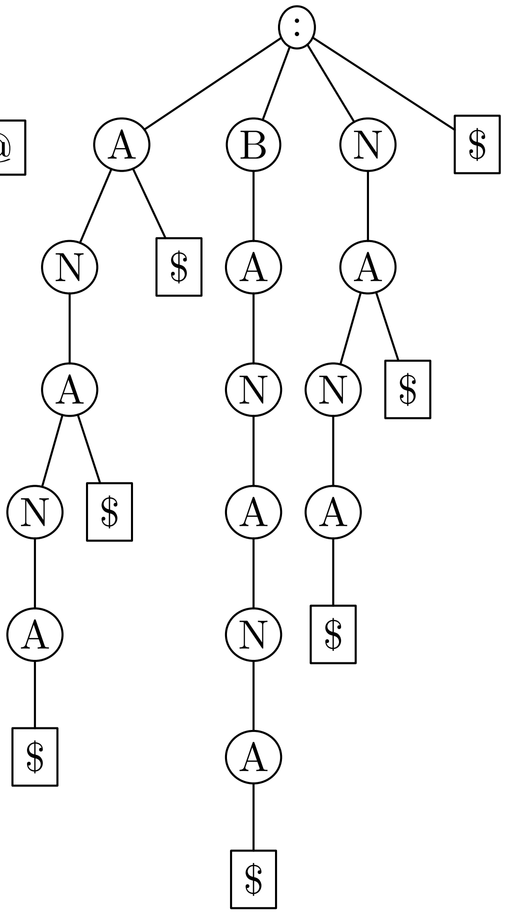
```
:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2
N14A11N6A4$2$2$2
```

Introducing pointers

- The size of a trie for string of length n is $O(n^2)$.
- Indexing of an $1MB$ textfile would be impractical.
- We will use the same idea as in suffix trees – group nodes with a single child. Here we only group nodes with a single *leaf* child.



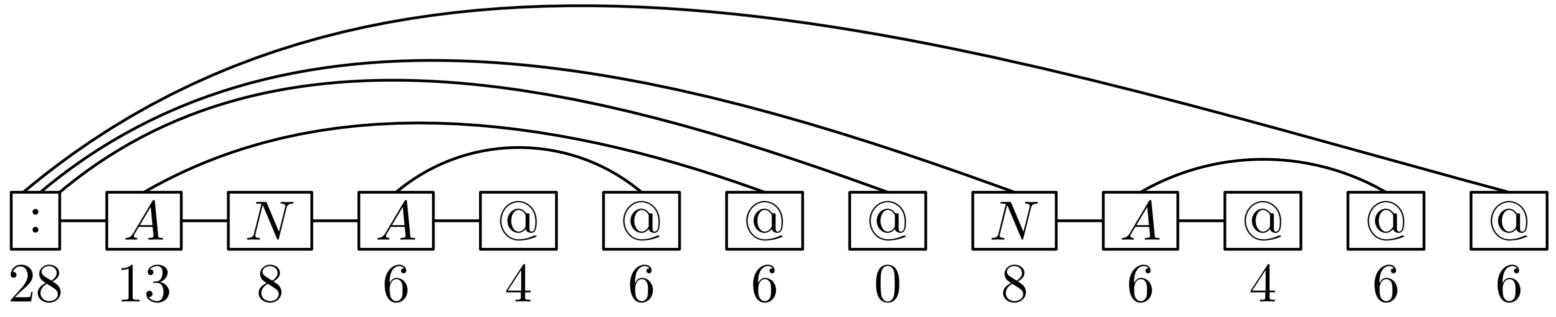
Trie with pointers



Trie before

Outputting index with pointers

- Input string `BANANA$`
  `0123456`
- Suffix trie with pointers



- Suffix trie in file
  `:28A13N8A6@4@6@6@0N8A6@4@6@6`
- In order to read suffix trie from file, we need the original input

Indexing

- Sometimes we have data consisting of several items.
- We can make suffix trie for many strings.
- Later we can use the index to search patterns from all the strings simultanously.

Size of index

Index size / text size ratio

No. of rows	Length of row		
	10	100	1000
1	12.9	163	2223
10	9.73	157	2214
100	6.94	152	
1000	4.93	146	
10000	3.54		

without pointers

No. of rows	Length of row			
	10	100	1000	10000
1	3.55	5.14	6.01	7.11
10	4.28	5.95	7.10	8.11
100	5.21	6.97	8.09	9.13
1000	5.92	7.93	9.11	
10000	6.65	8.92		

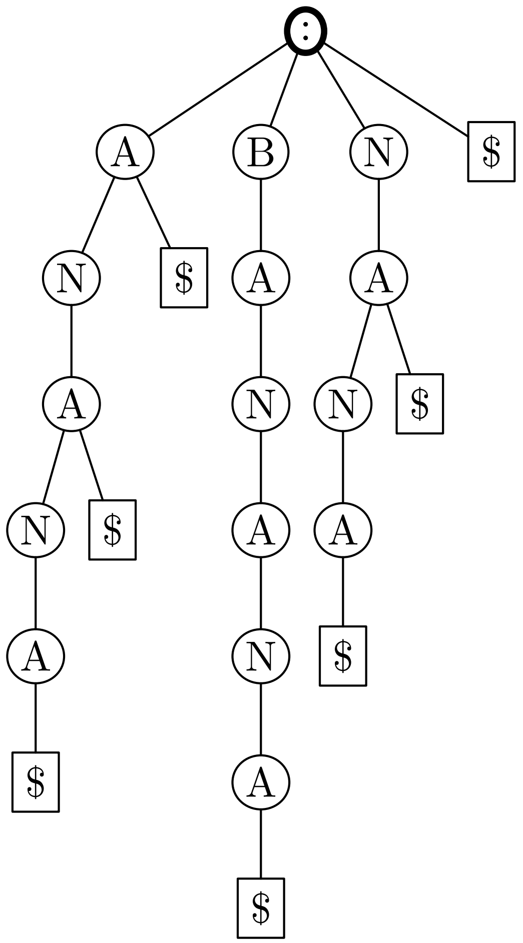
with pointers

- If a random string in 4-letter alphabet has length n, then the number of nodes is about $1.72n$.
- The description of each node is at most $1 + log_{10}n$ bytes.

Using the index

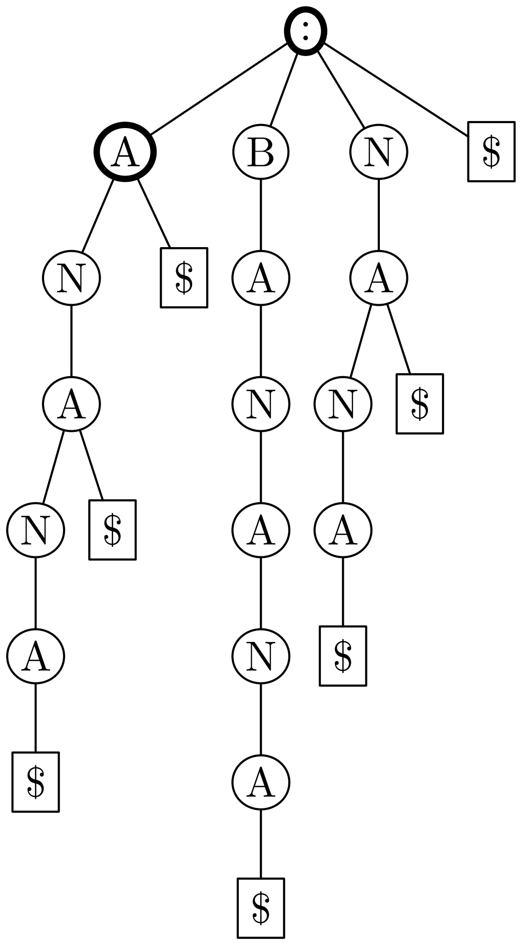
- Suppose we have an suffix trie S for text T written to a file.
- The two operations that can be performed for any node:
  - Get the next sibling of that node
  - Get the first child of that node
- `:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2 N14A11N6A4$2$2$2`
- How can we walk through the trie?

Walking through the trie



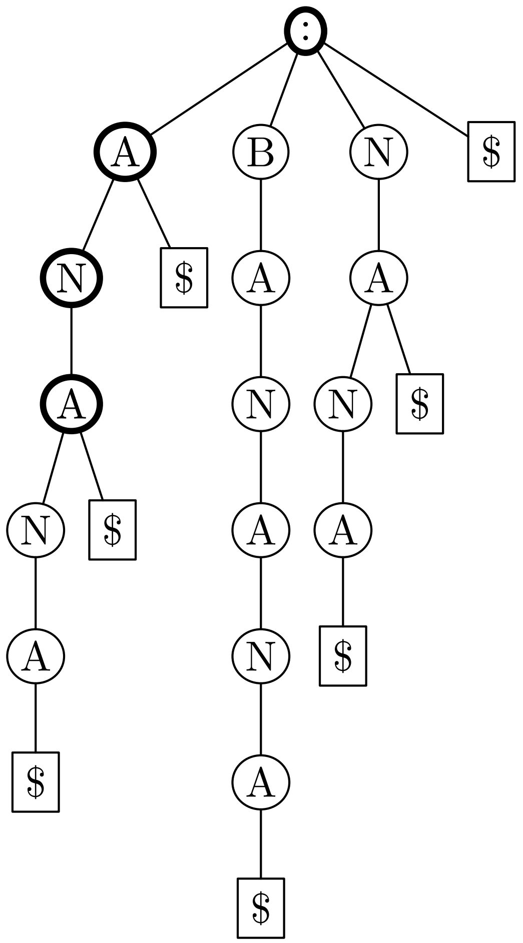
```
:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2
```

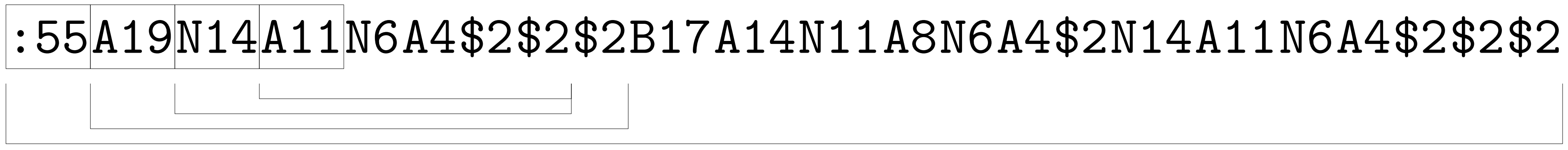
Walking through the trie



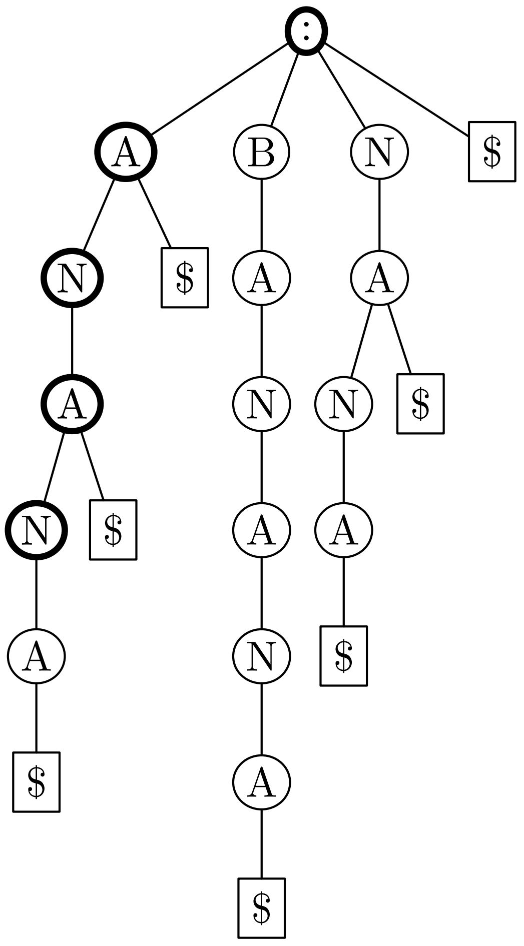
:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2

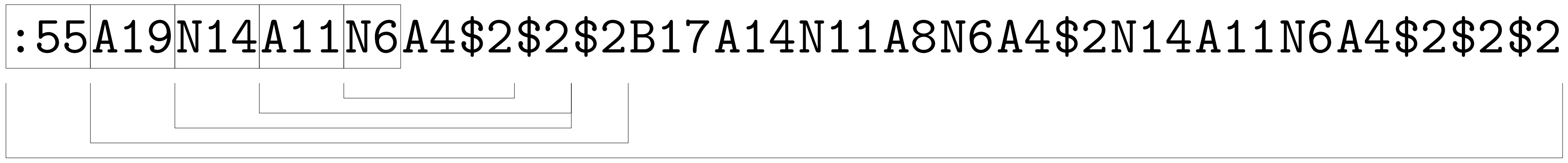
Walking through the trie



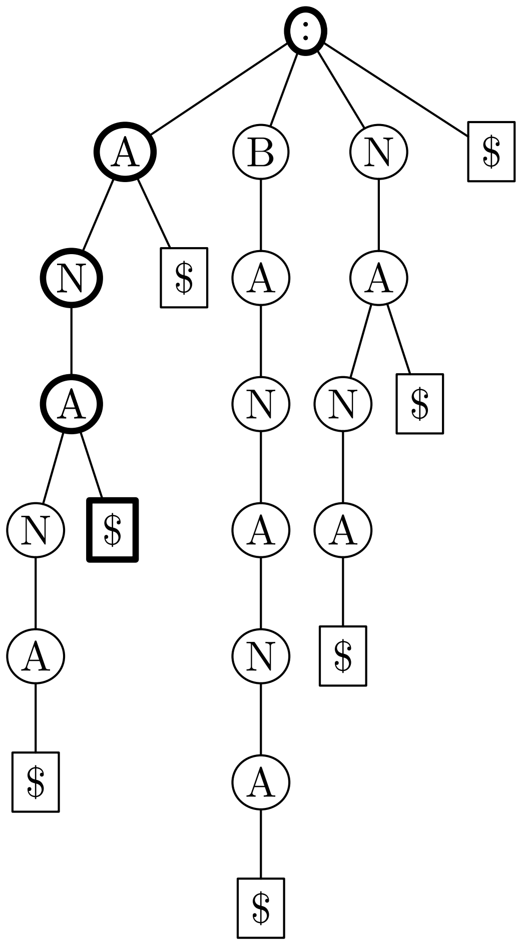


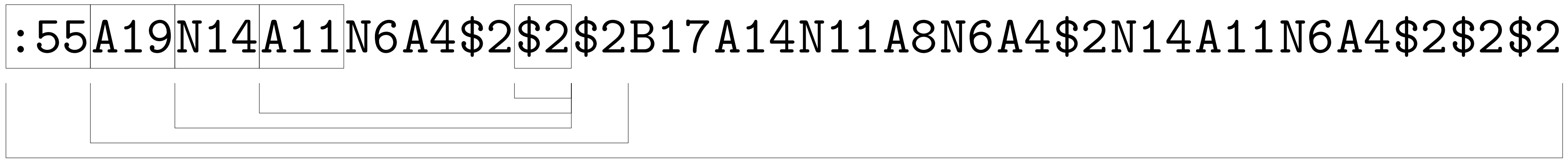
Walking through the trie



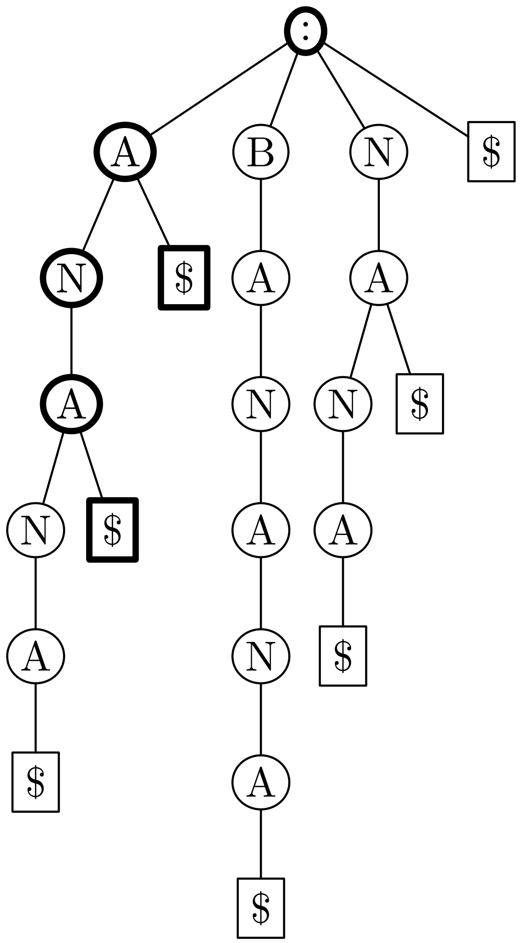


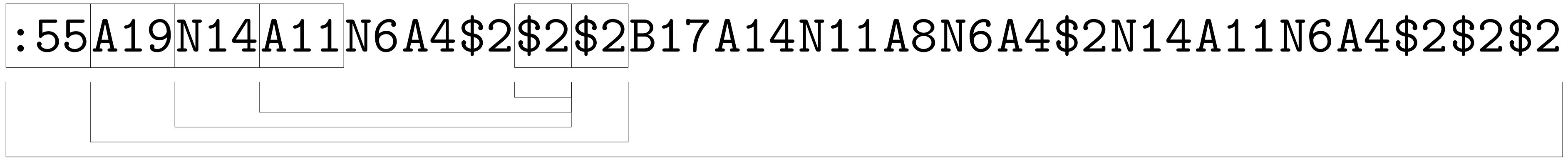
Walking through the trie





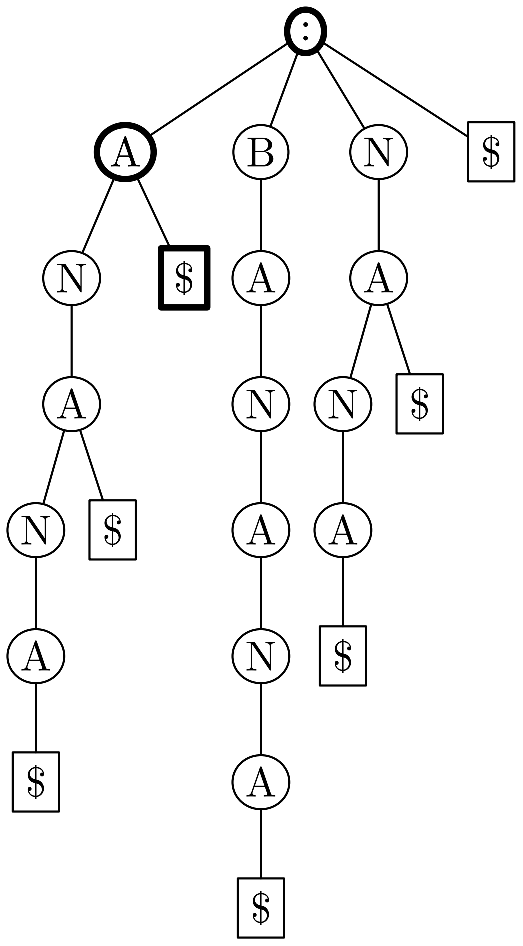
Walking through the trie

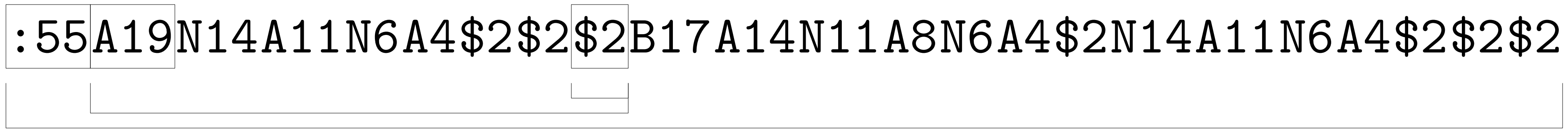




```
:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2
```

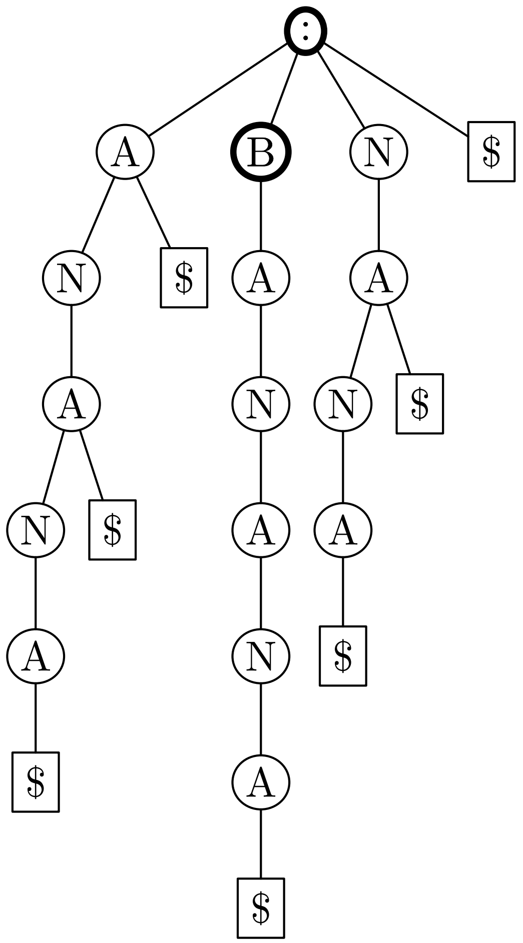
Walking through the trie





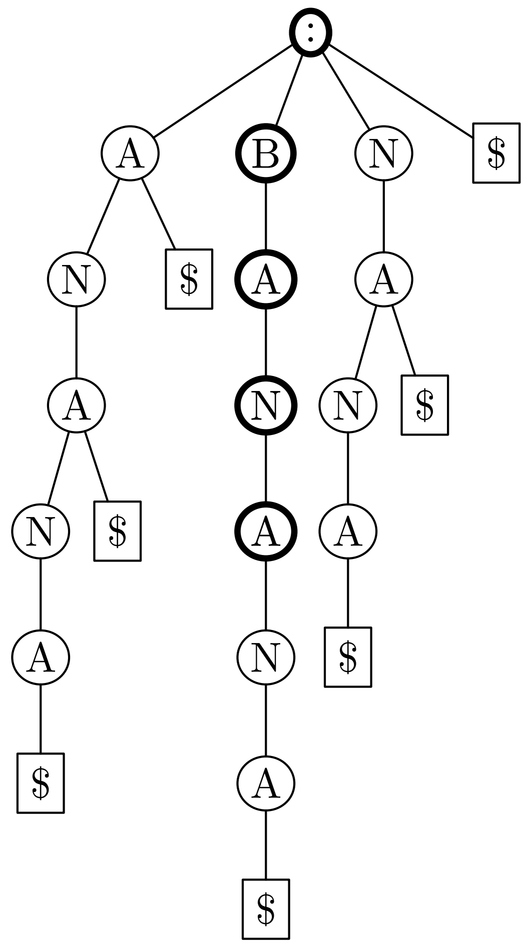
```
:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2
```

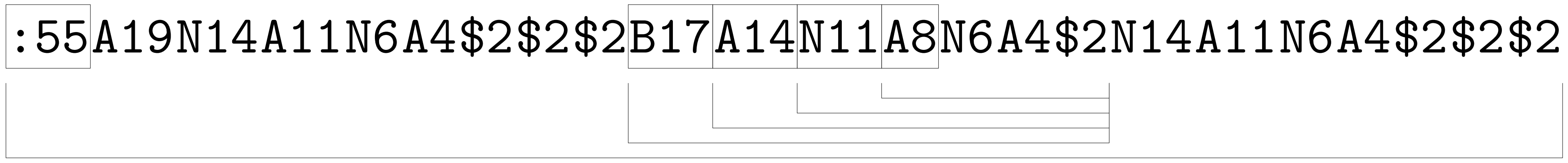
Walking through the trie



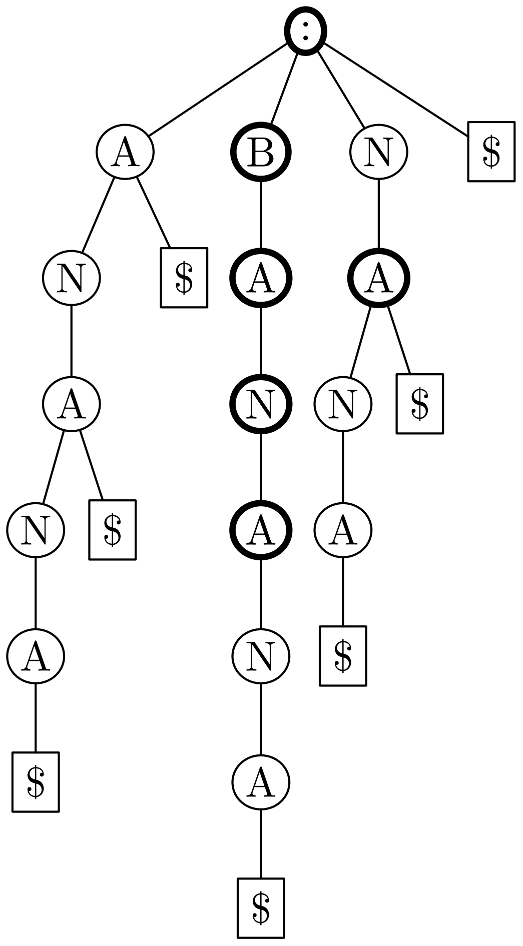
`:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2`

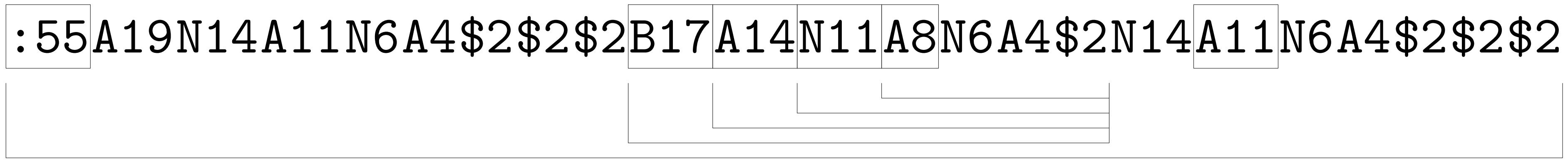
Walking through the trie



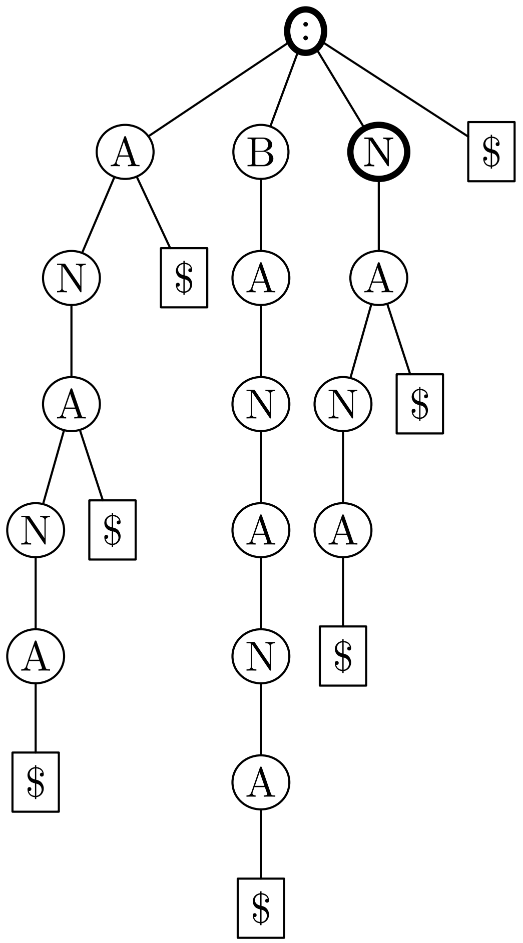


Walking through the trie



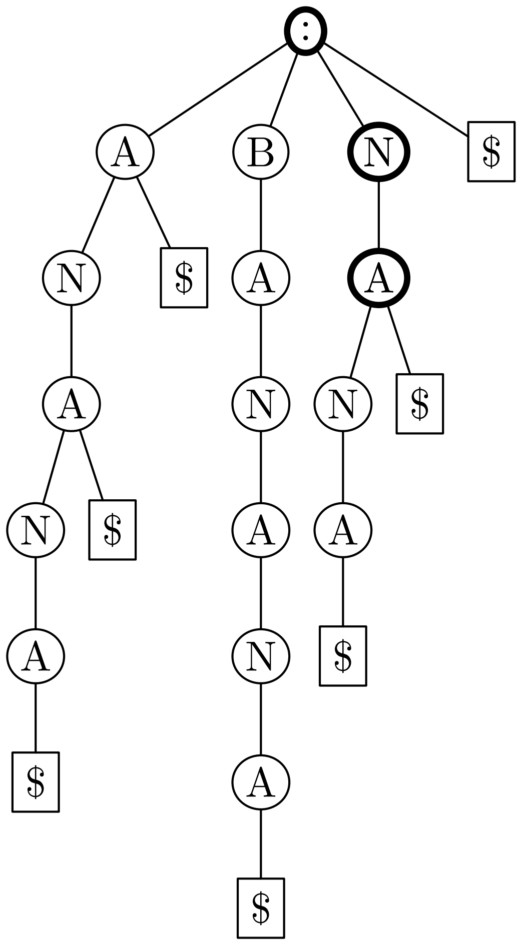


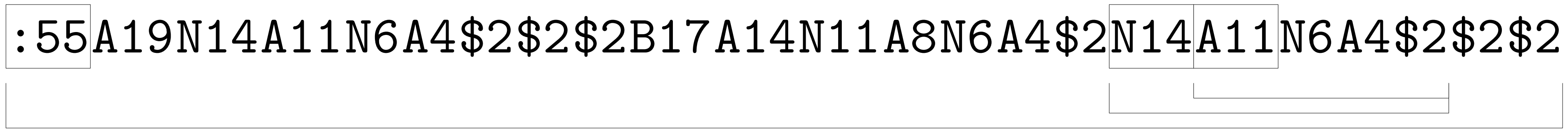
Walking through the trie



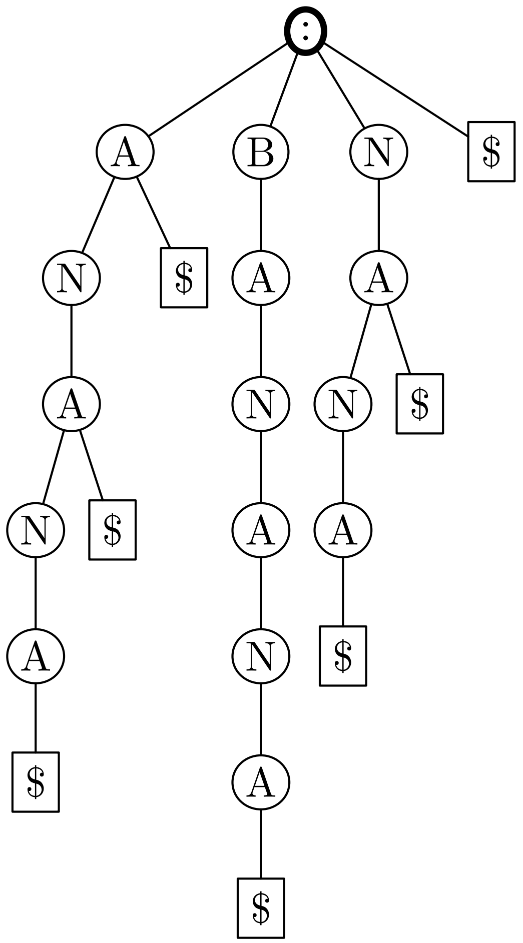
$\boxed{:55}$A19N14A11N6A4\$2\$2\$2B17A14N11A8N6A4\$2$\boxed{\text{N14}}$A11N6A4\$2\$2\$2

Walking through the trie





Walking through the trie



:55A19N14A11N6A4$2$2$2B17A14N11A8N6A4$2N14A11N6A4$2$2$2

Algorithms with tries

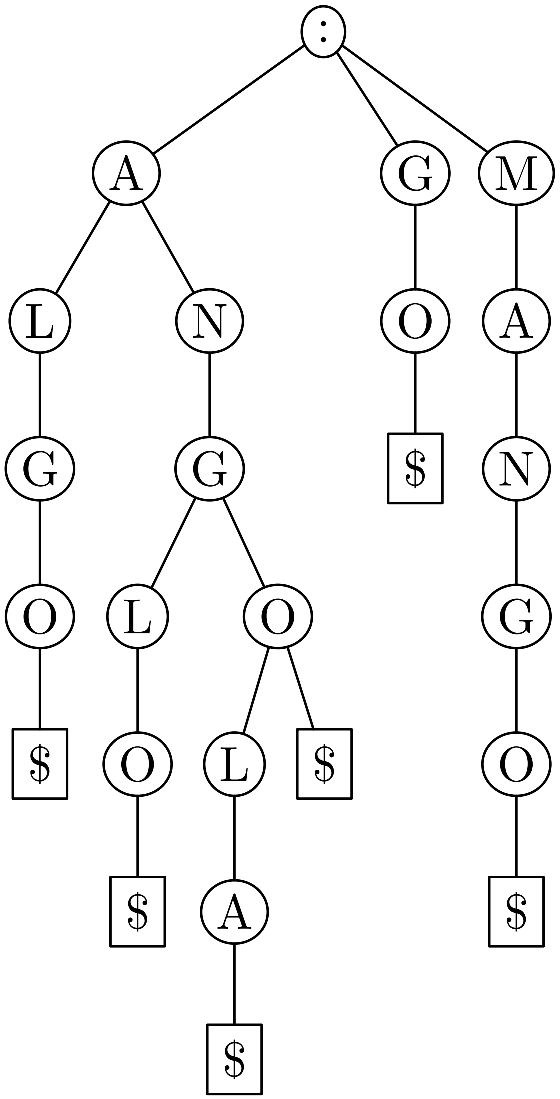
- We now show how to use tries.
- Suffix tries can be used in the same way.

Exact string matching

- **Example.** We have an index containing strings:
  - `ALGO`
  - `ANGLO`
  - `ANGOLA`
  - `ANGO`
  - `GO`
  - `MANGO`
- We want to search for occurrences of string `ANGO`

Exact string matching

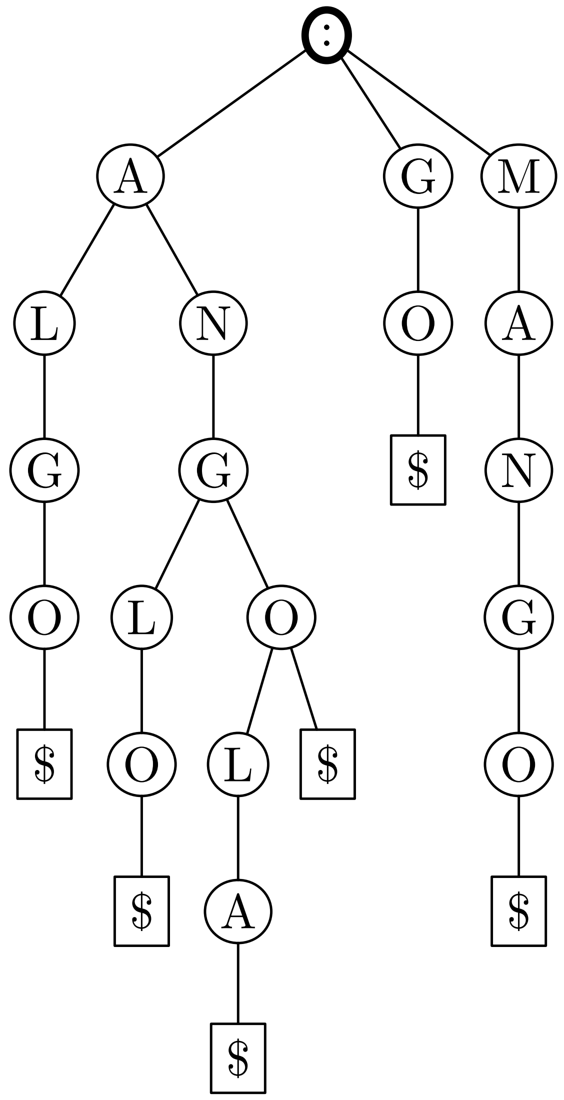
- Trie containig strings:
  - `ALGO`
  - `ANGLO`
  - `ANGOLA`
  - `ANGO`
  - `GO`
  - `MANGO`



Exact string matching

- Searching for string `ANGO`
- Search table

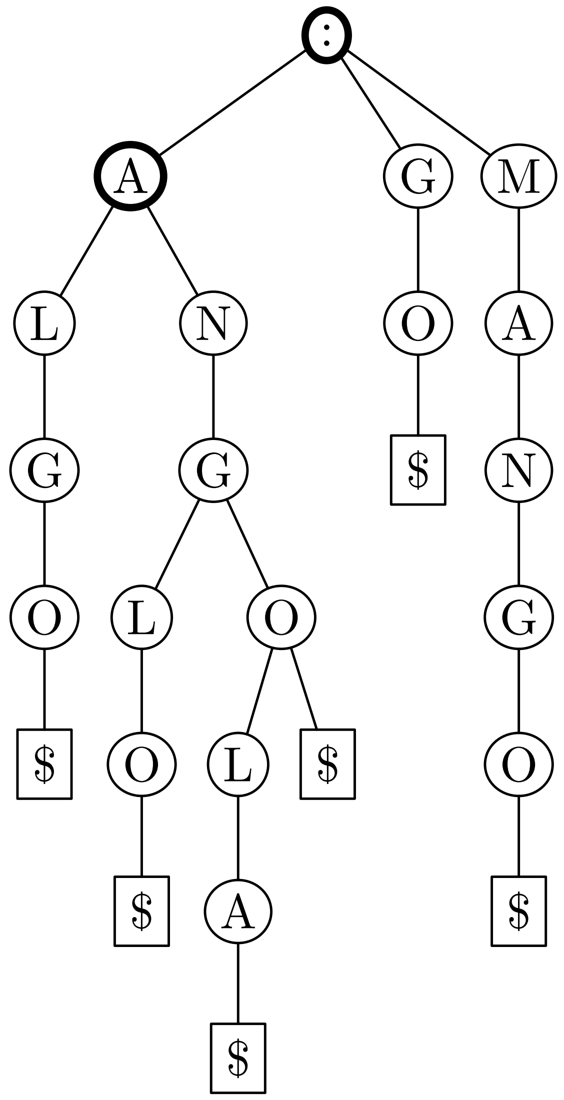
char	OK
	+
A	
N	
G	
O	



Exact string matching

- Searching for string `ANGO`
- Search table

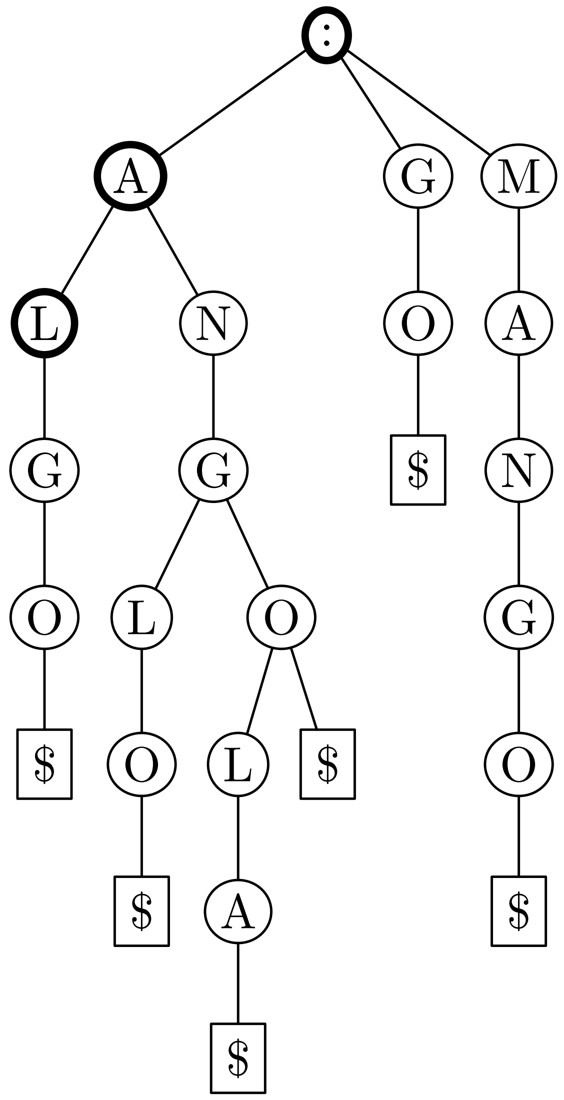
char	OK
	+
A	+
N	
G	
O	



Exact string matching

- Searching for string `ANGO`
- Search table

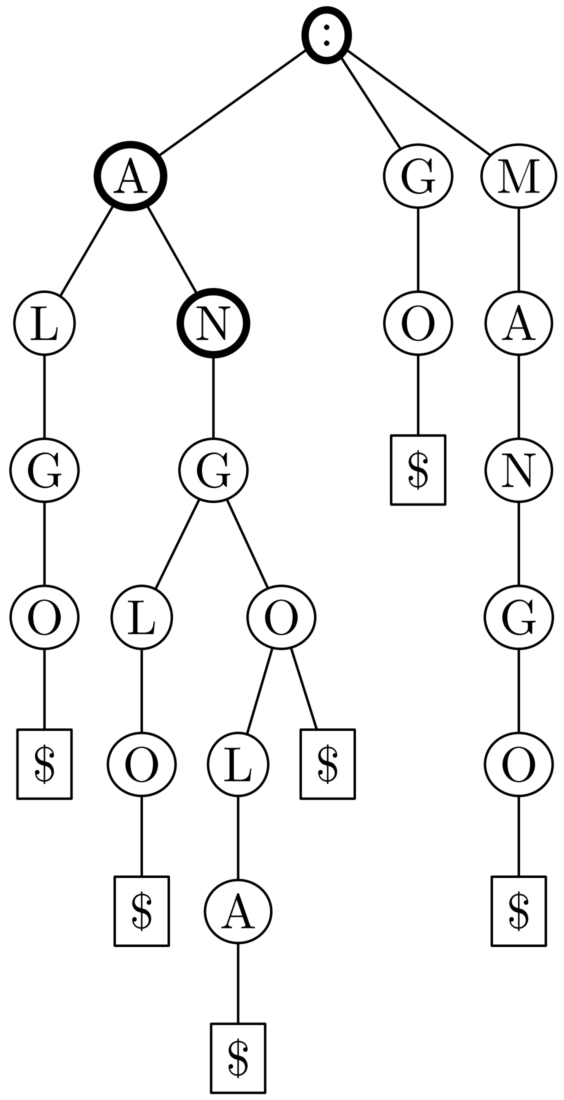
char	OK
	+
A	+
N	-
G	
O	



Exact string matching

- Searching for string `ANGO`
- Search table

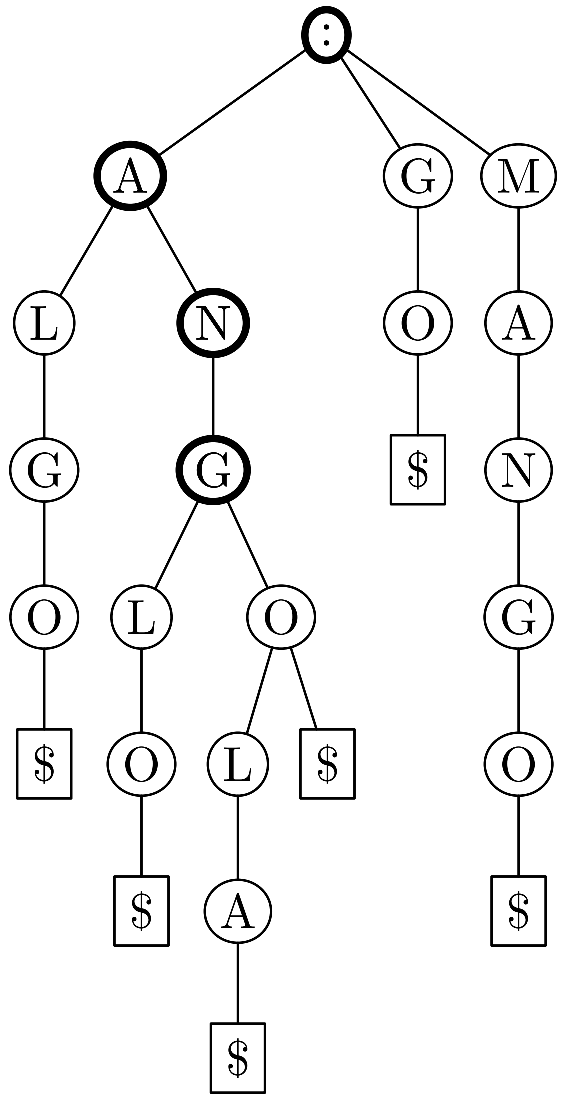
char	OK
	+
A	+
N	+
G	
O	



Exact string matching

- Searching for string `ANGO`
- Search table

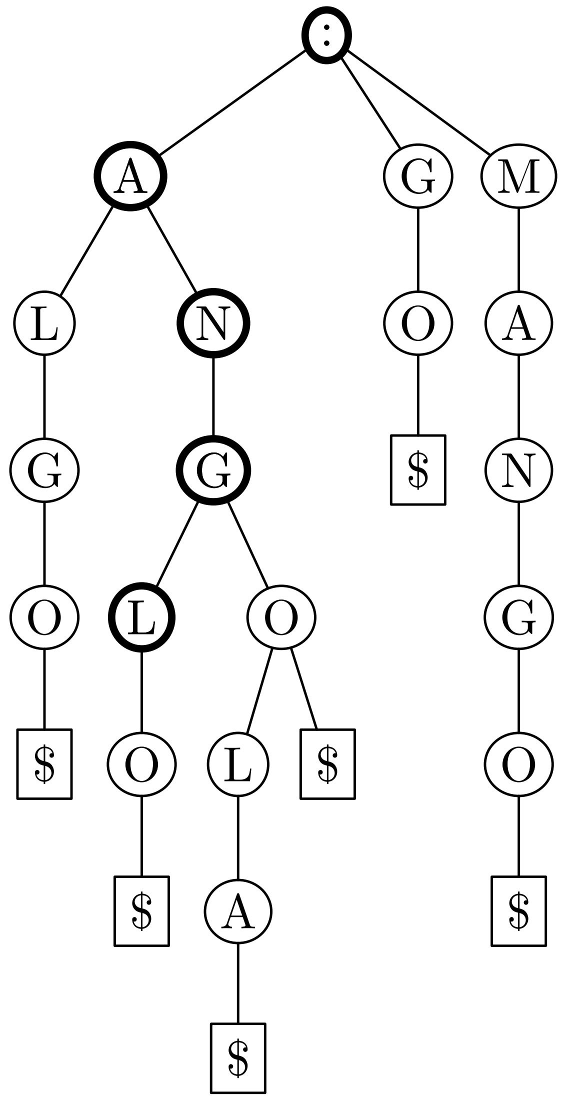
char	OK
	+
A	+
N	+
G	+
O	



Exact string matching

- Searching for string `ANGO`
- Search table

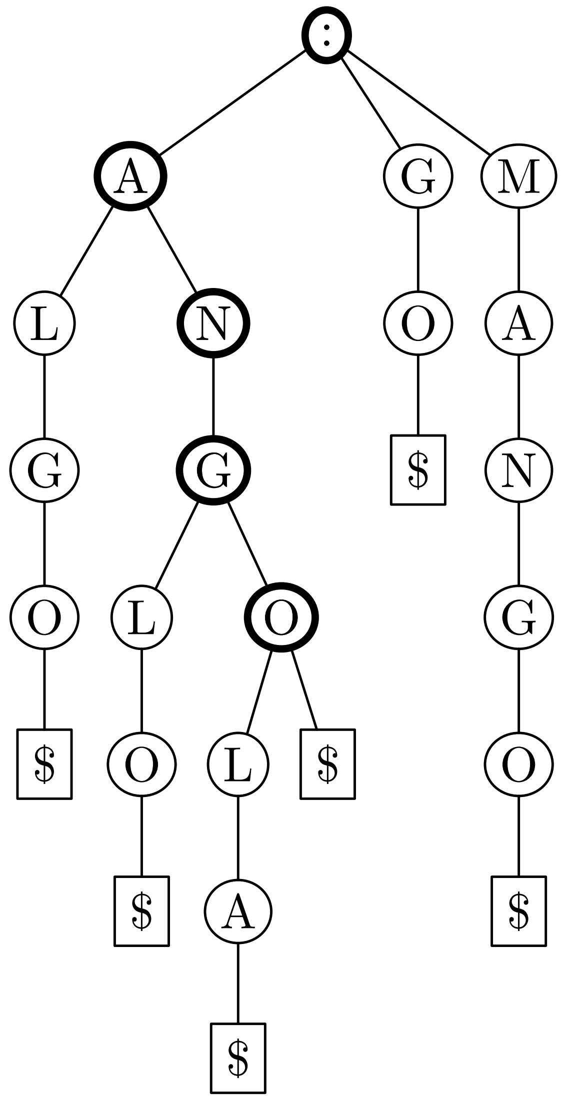
char	OK
	+
A	+
N	+
G	+
O	-



Exact string matching

- Searching for string `ANGO`
- Search table

char	OK
	+
A	+
N	+
G	+
O	+

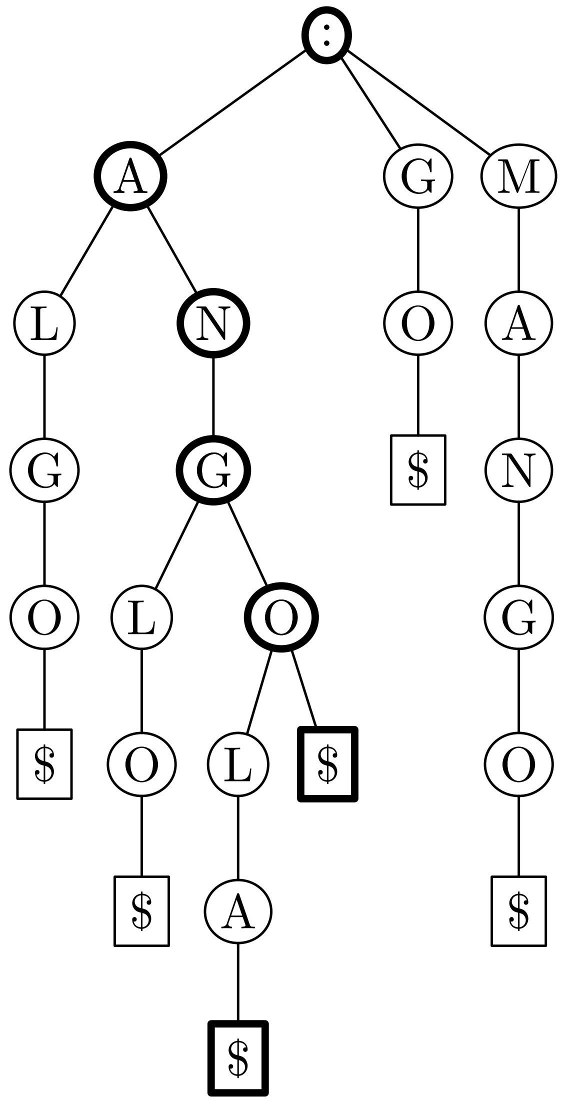


Exact string matching

- Searching for string `ANGO`
- Search table

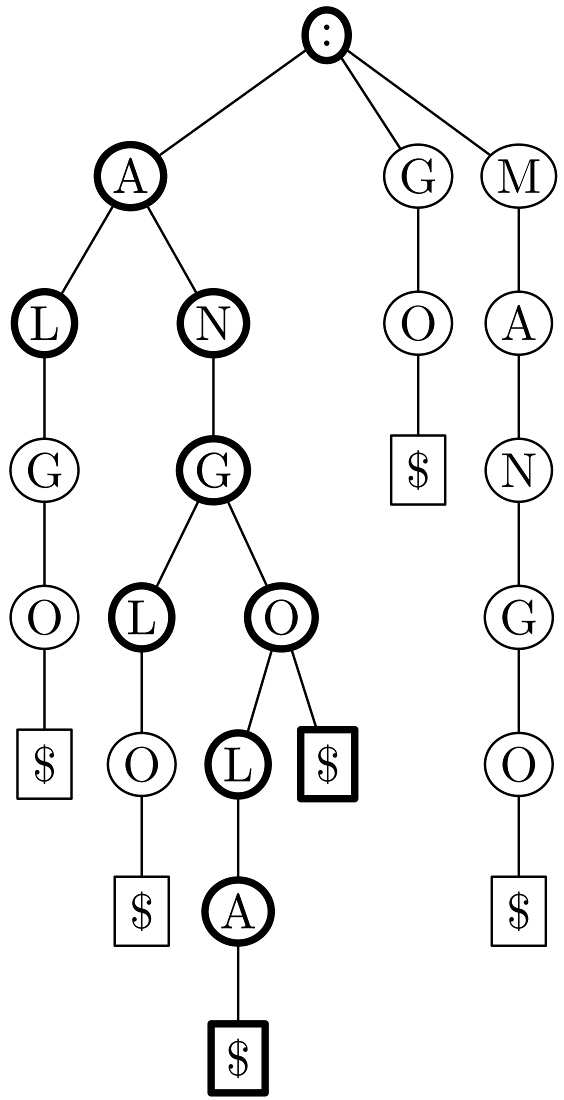
char	OK
	+
A	+
N	+
G	+
O	+

- Print the results



Exact string matching

- Searching for string `ANGO`
- This is how many nodes we needed to examine in the trie.
- Searching for a string of length m can be done in $O(ms + M)$ time, where s is the size of the alphabet and M is the number of matches.



Approximate string matching

- We would like to find all substrings of text T, which have the *edit distance* from pattern P at most D.
- Computation of edit distance of strings x and y:
  - $M_{0,0} \leftarrow 0$
  - $M_{i,j} \leftarrow \min(M_{i-1,j-1} + \delta(x_i, y_i),\ M_{i-1,j} + 1,\ M_{i,j-1} + 1)$
  - Return $M_{|x|,|y|}$
- Here $\delta(x_i, y_i) =$
  - 0, if $x_i = y_i$
  - 1, if $x_i \neq y_i$
- The edit distance of `ANGEL` and `MANGO` is 3:

		M	A	N	G	O
	0	1	2	3	4	5
A	1	1	1	2	3	4
N	2	2	2	1	2	3
G	3	3	3	2	1	2
E	4	4	4	3	2	2
L	5	5	5	4	3	3

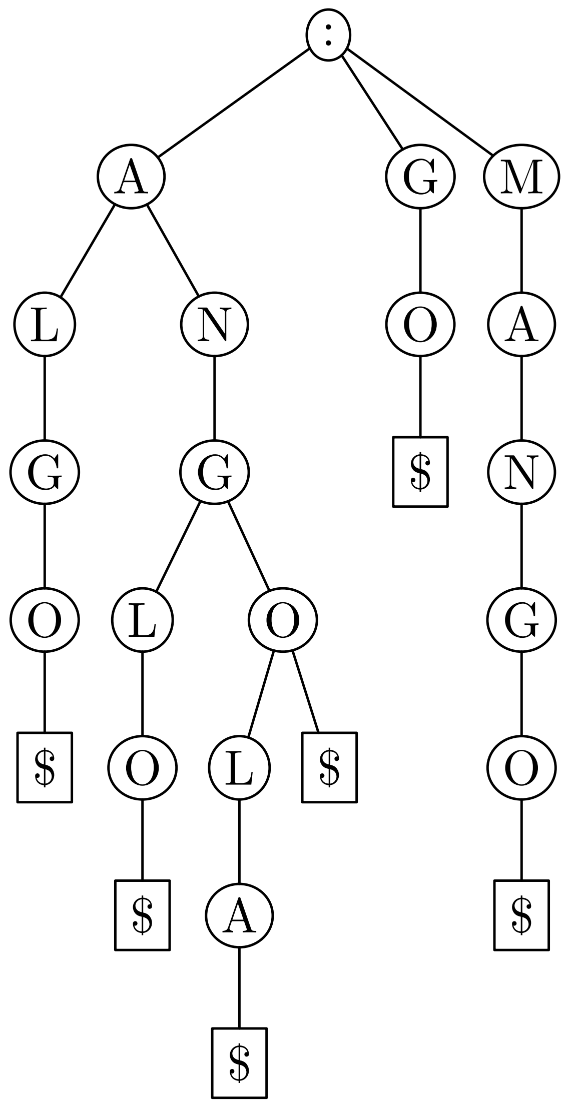
Calculation of edit distance

- We do not need to calculate the whole table.
- If $D = 2$, we only need some values close to the main diagonal.

		M	A	N	G	O
	0	1	2			
A	1	1	1	2		
N	2	2	2	1	2	
G				2	1	2
E					2	2
L						

Approximate string matching

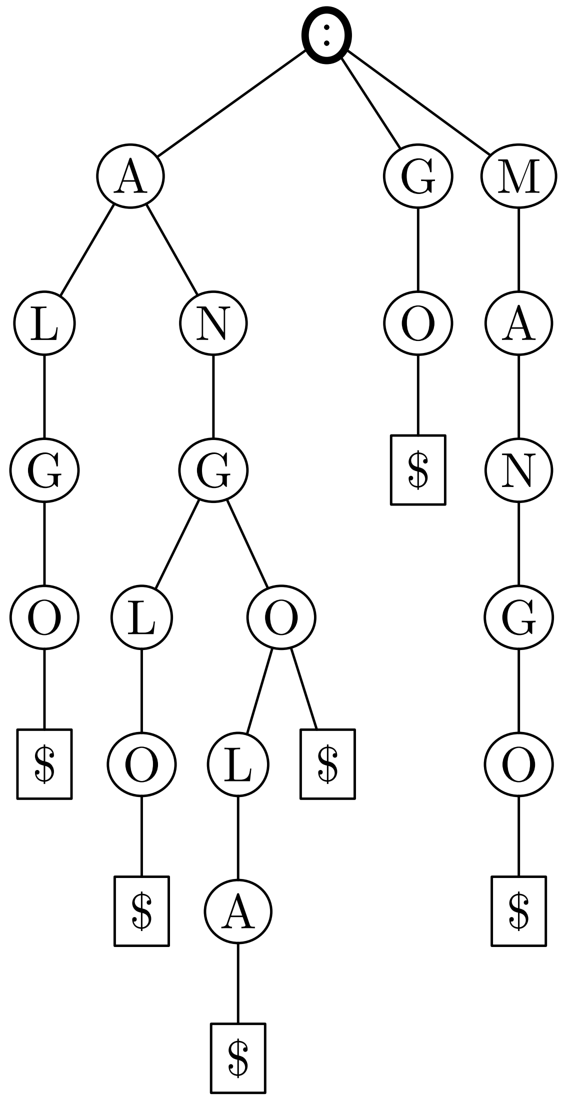
- **Example.** We have a trie containing strings:
  - `ALGO`
  - `ANGLO`
  - `ANGOLA`
  - `ANGO`
  - `GO`
  - `MANGO`
- We want to search for occurrences of string `ANGEL` with edit distance at most 1.



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

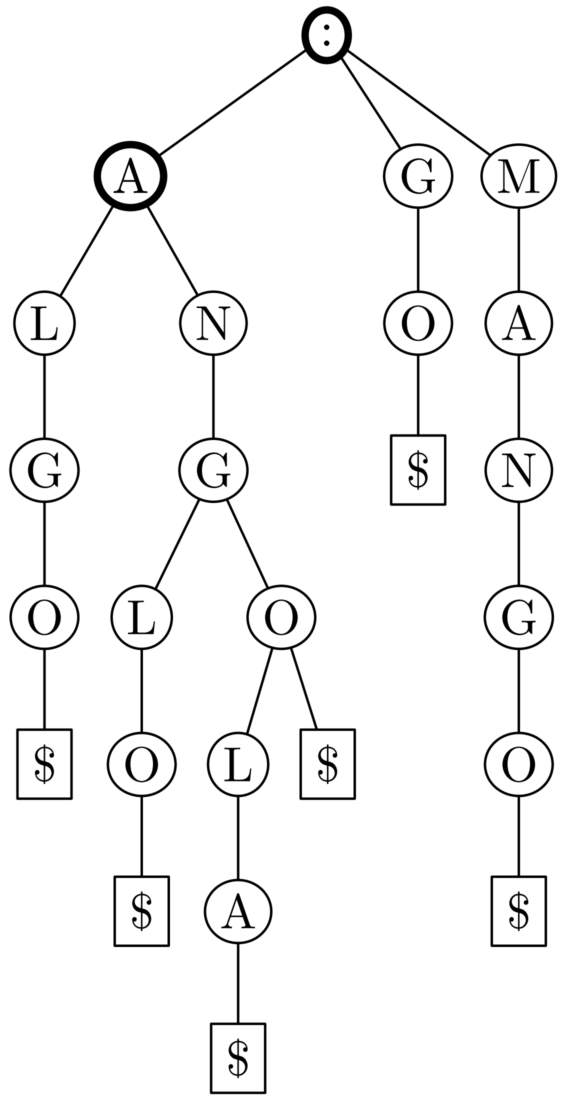
	:
	0
A	1
N	
G	
E	
L	



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

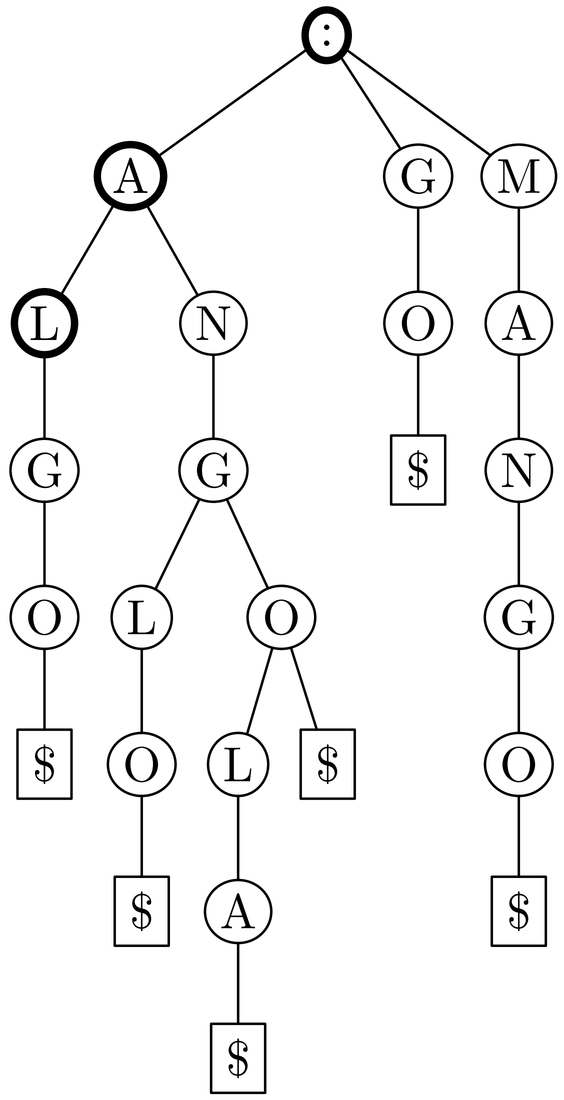
	:	A
	0	1
A	1	0
N		1
G		
E		
L		



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

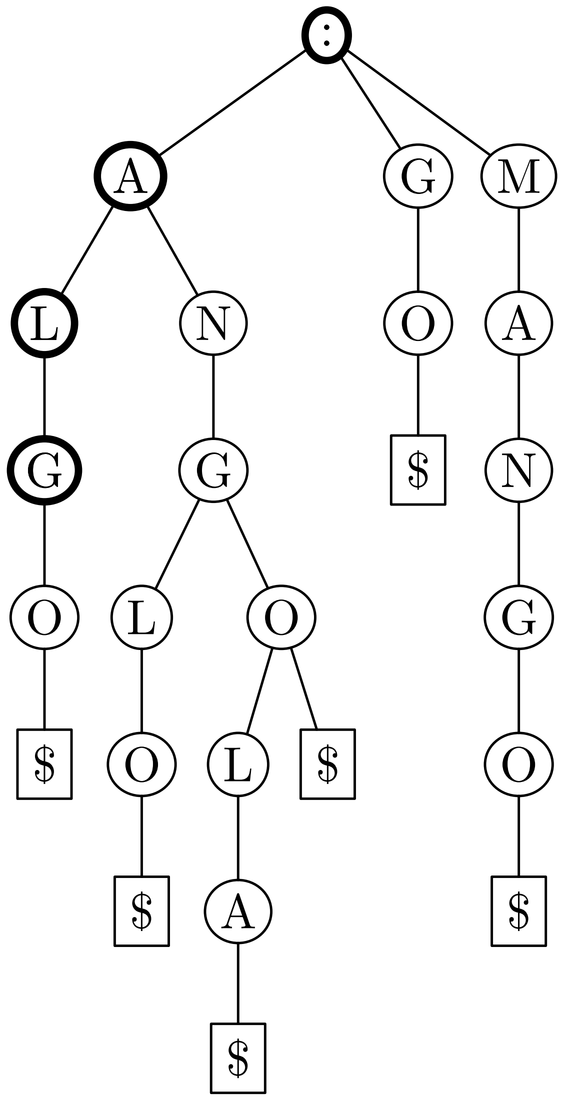
	:	A	L
	0	1	
A	1	0	1
N		1	1
G			
E			
L			



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

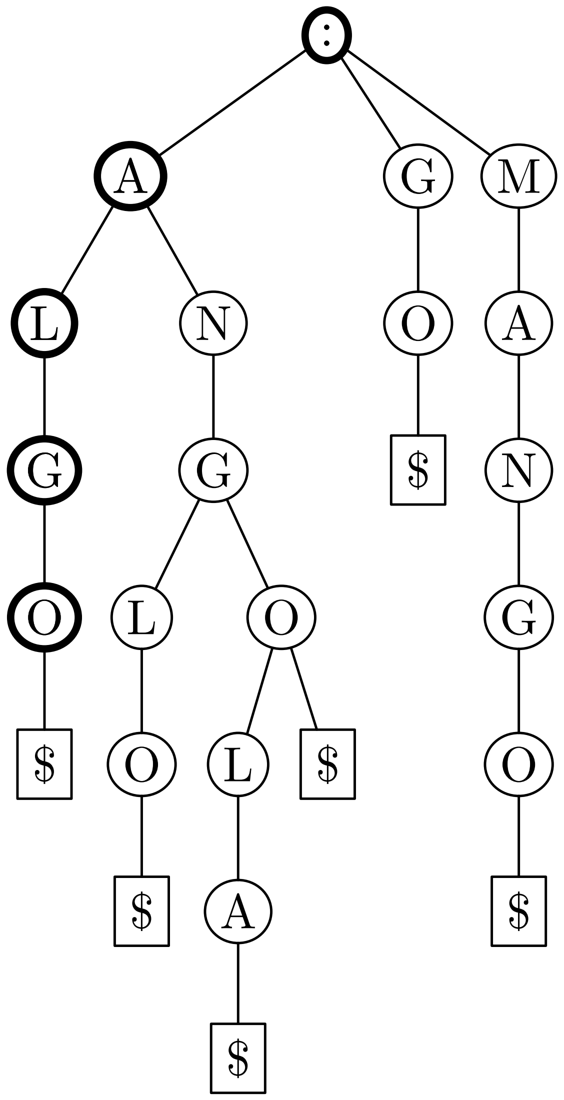
	:	A	L	G
	0	1		
A	1	0	1	
N		1	1	
G				1
E				
L				



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

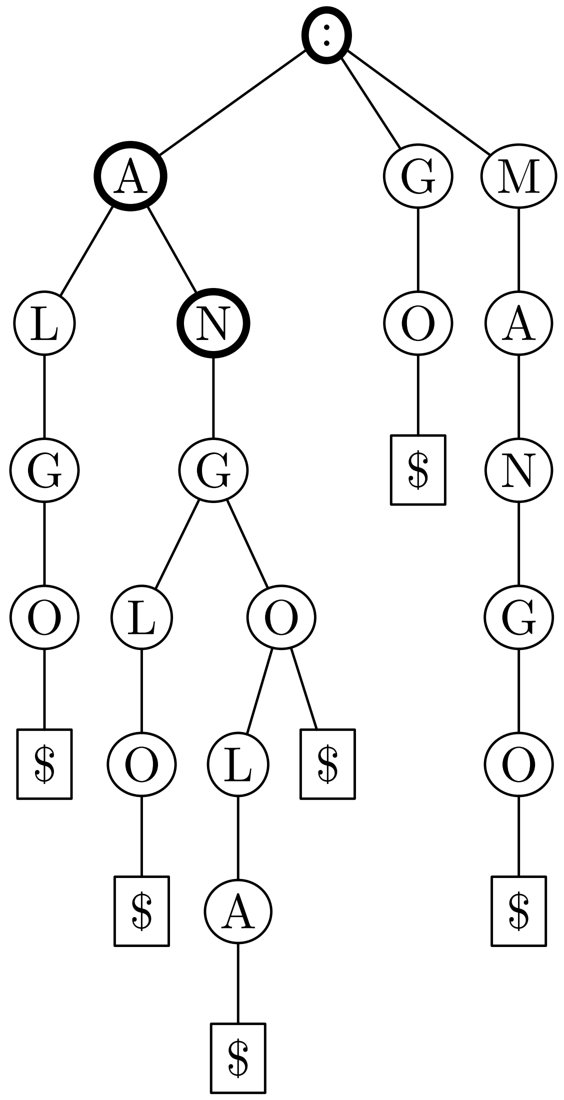
	:	A	L	G	O
	0	1			
A	1	0	1		
N		1	1		
G				1	
E					
L					



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

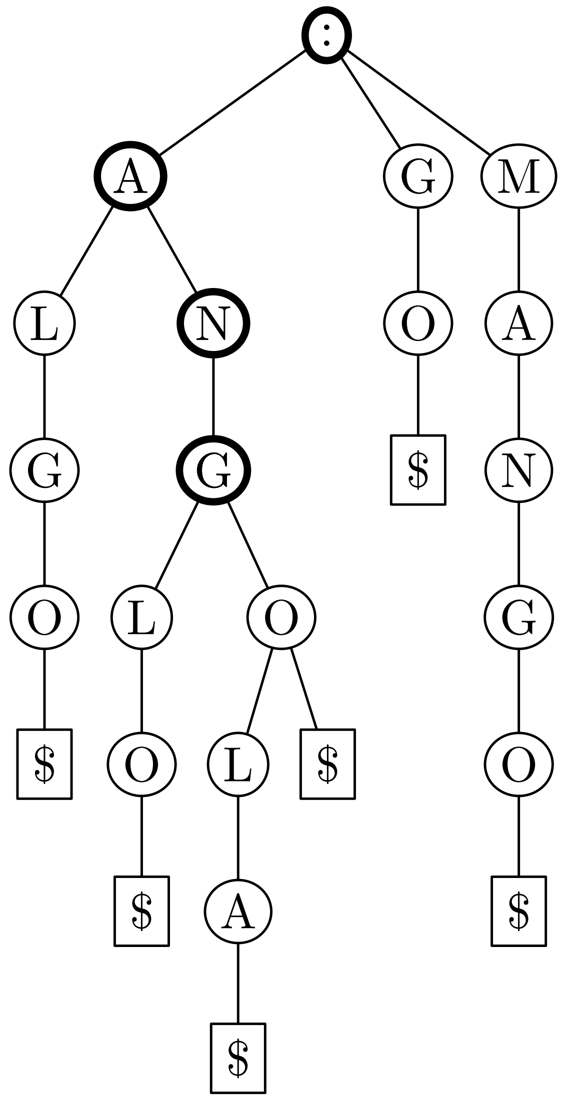
	:	A	N
	0	1	
A	1	0	1
N		1	0
G			1
E			
L			



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

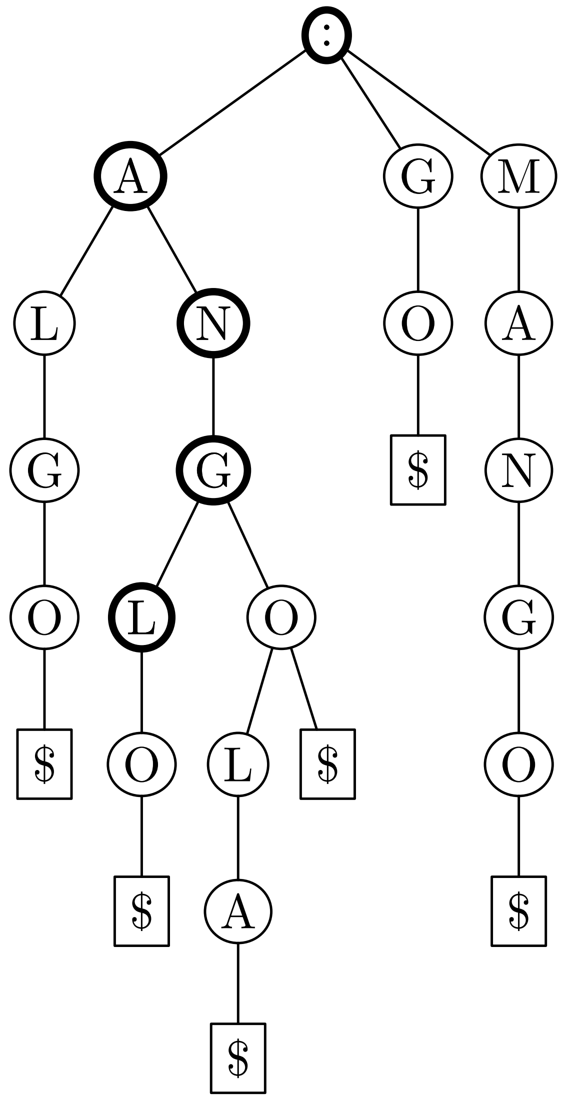
	:	A	N	G
	0	1		
A	1	0	1	
N		1	0	1
G			1	0
E				1
L				



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

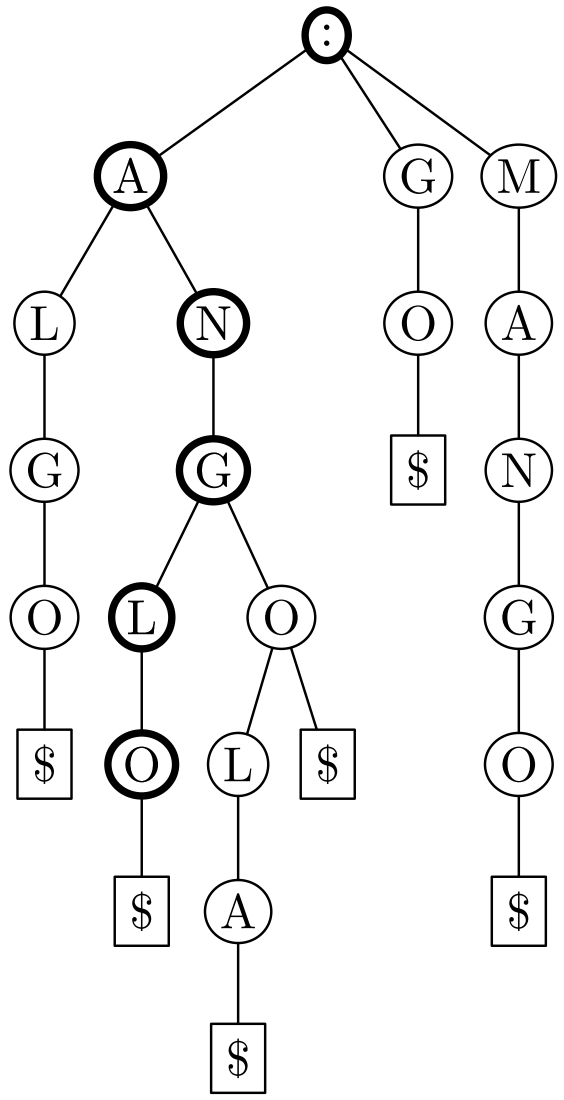
	:	A	N	G	L
	0	1			
A	1	0	1		
N		1	0	1	
G			1	0	1
E				1	1
L					



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

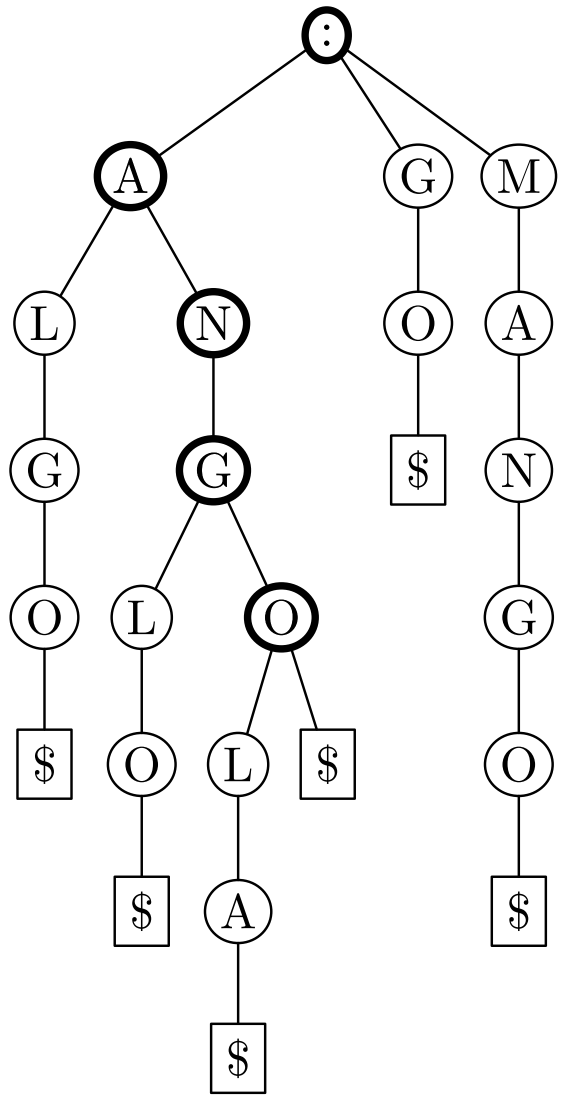
	:	A	N	G	L	O
	0	1				
A	1	0	1			
N		1	0	1		
G			1	0	1	
E				1	1	
L						



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

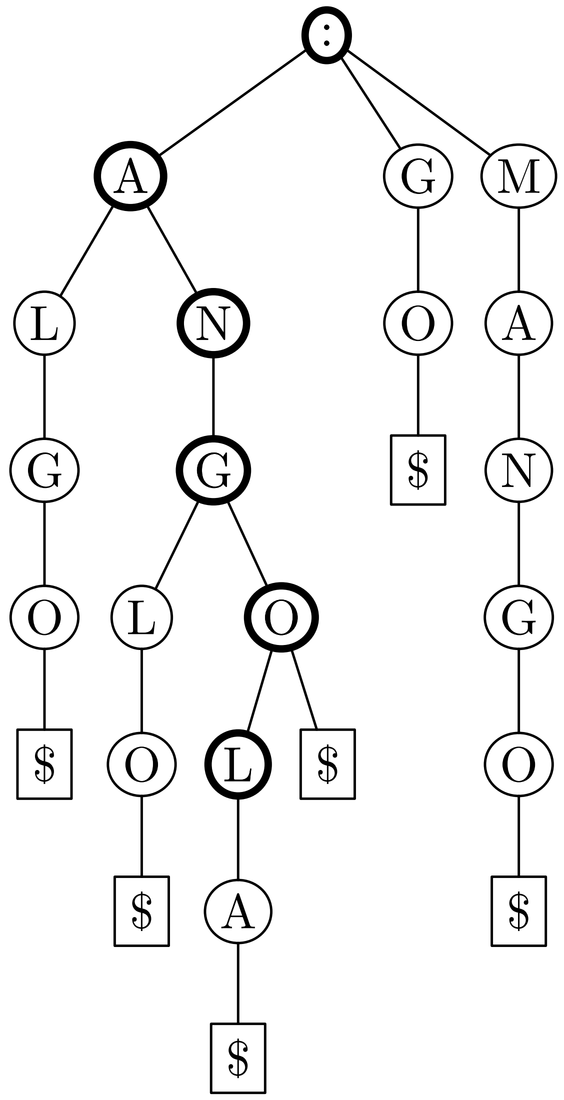
	:	A	N	G	O
	0	1			
A	1	0	1		
N		1	0	1	
G			1	0	1
E				1	1
L					



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

	:	A	N	G	O	L
	0	1				
A	1	0	1			
N		1	0	1		
G			1	0	1	
E				1	1	
L						1

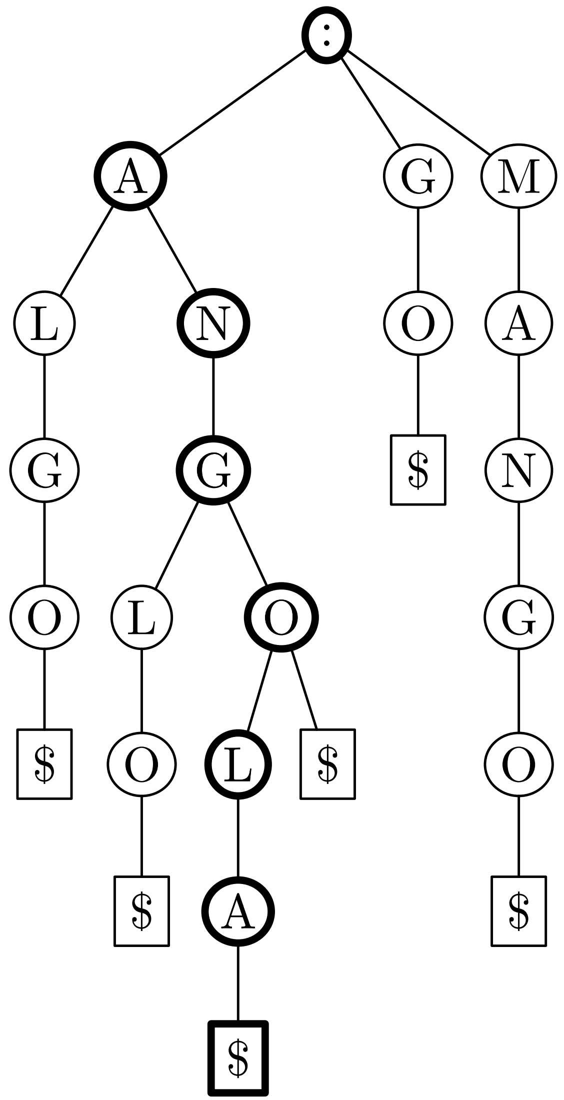


Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

	:	A	N	G	O	L
	0	1				
A	1	0	1			
N		1	0	1		
G			1	0	1	
E				1	1	
L						1

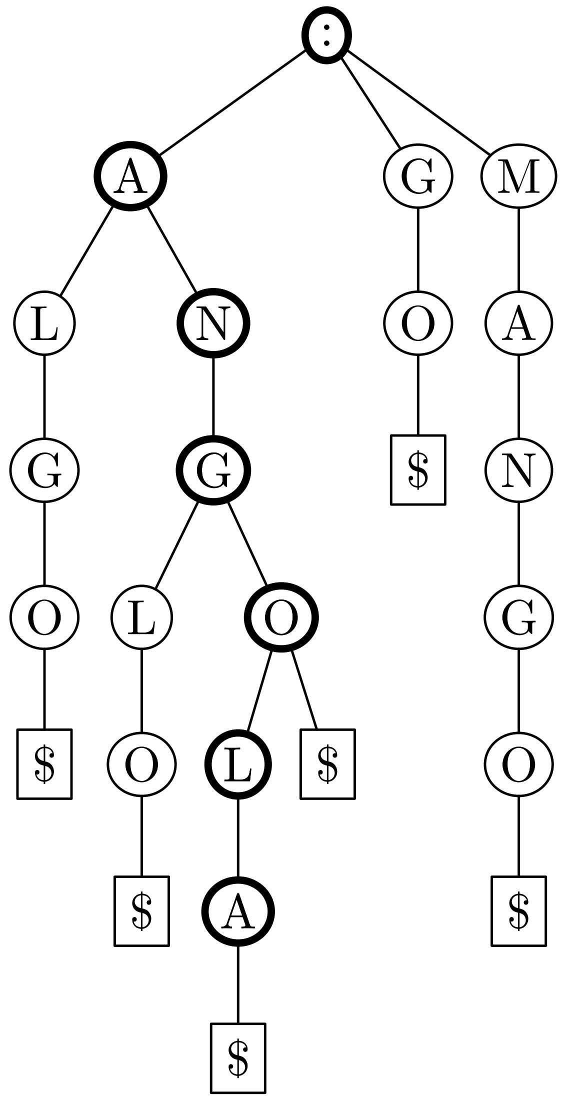
- Print occurrence



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

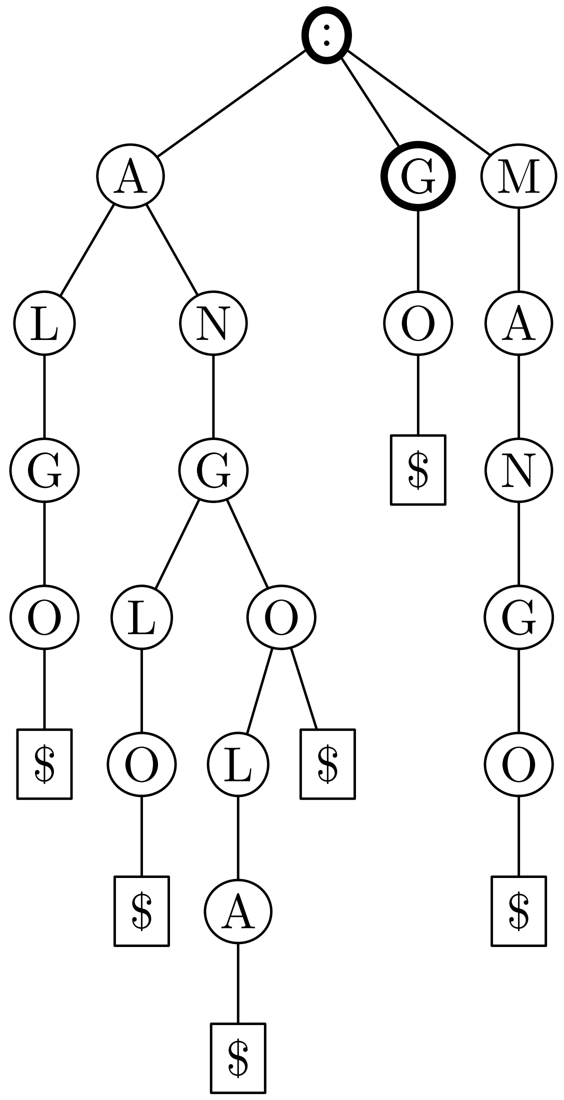
	:	A	N	G	O	L	A
	0	1					
A	1	0	1				
N		1	0	1			
G			1	0	1		
E				1	1		
L						1	



Approximate string matching

- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

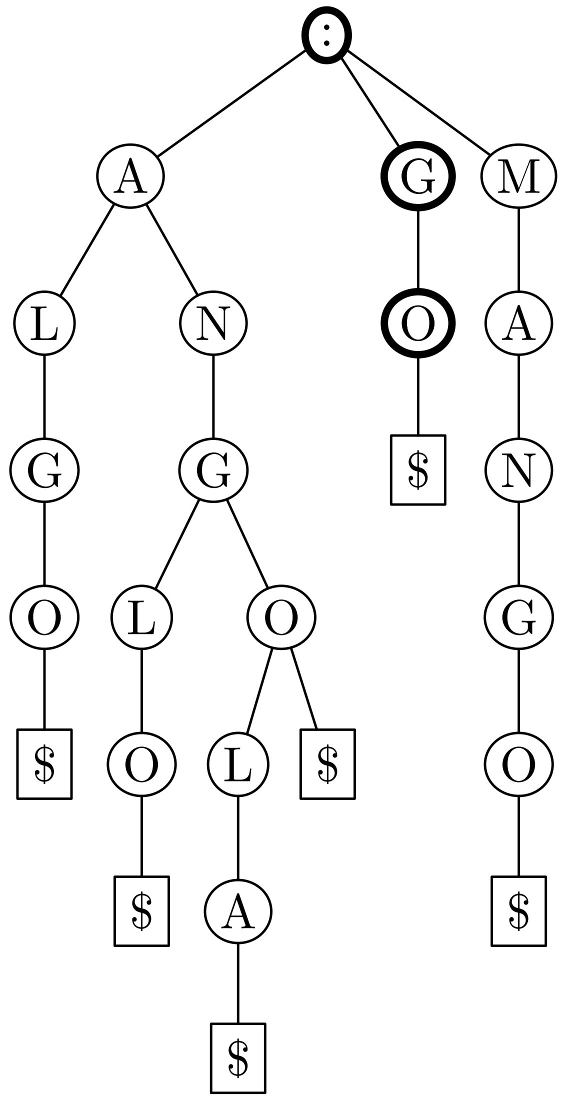
	:	G
	0	1
A	1	1
N		
G		
E		
L		



Approximate string matching

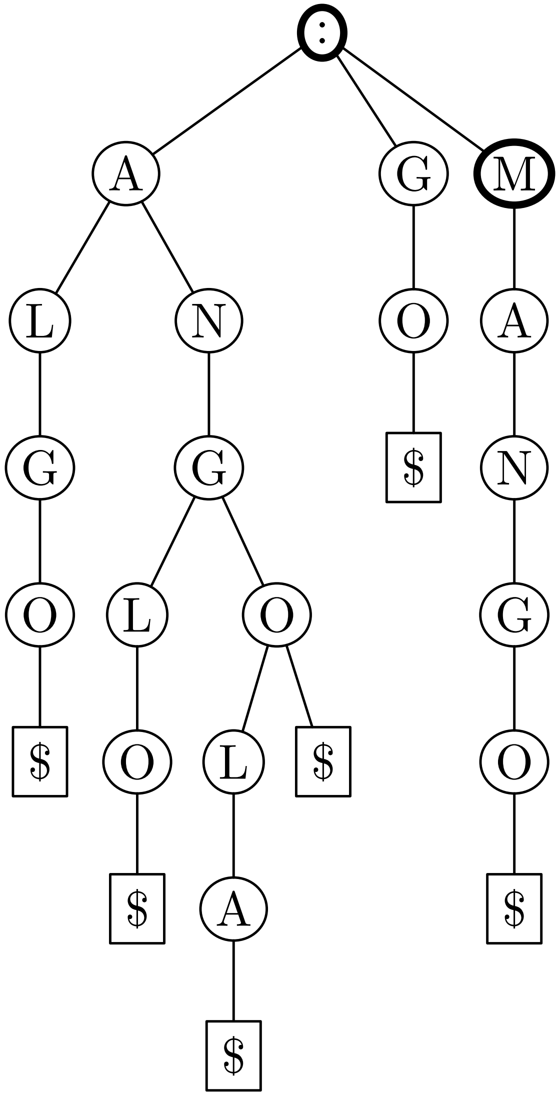
- Searching for string `ANGEL` with edit distance at most 1.
- Edit distance table

	:	G	O
	0	1	
A	1	1	
N			
G			
E			
L			



Approximate string matching

- And so on...



Approximate string matching

- Searching for a string of length m with edit distance D can be done in $O((ms)^{D+1} + M)$ time, where s is the size of the alphabet and M is the number of matches.

Exact all-against-all matching

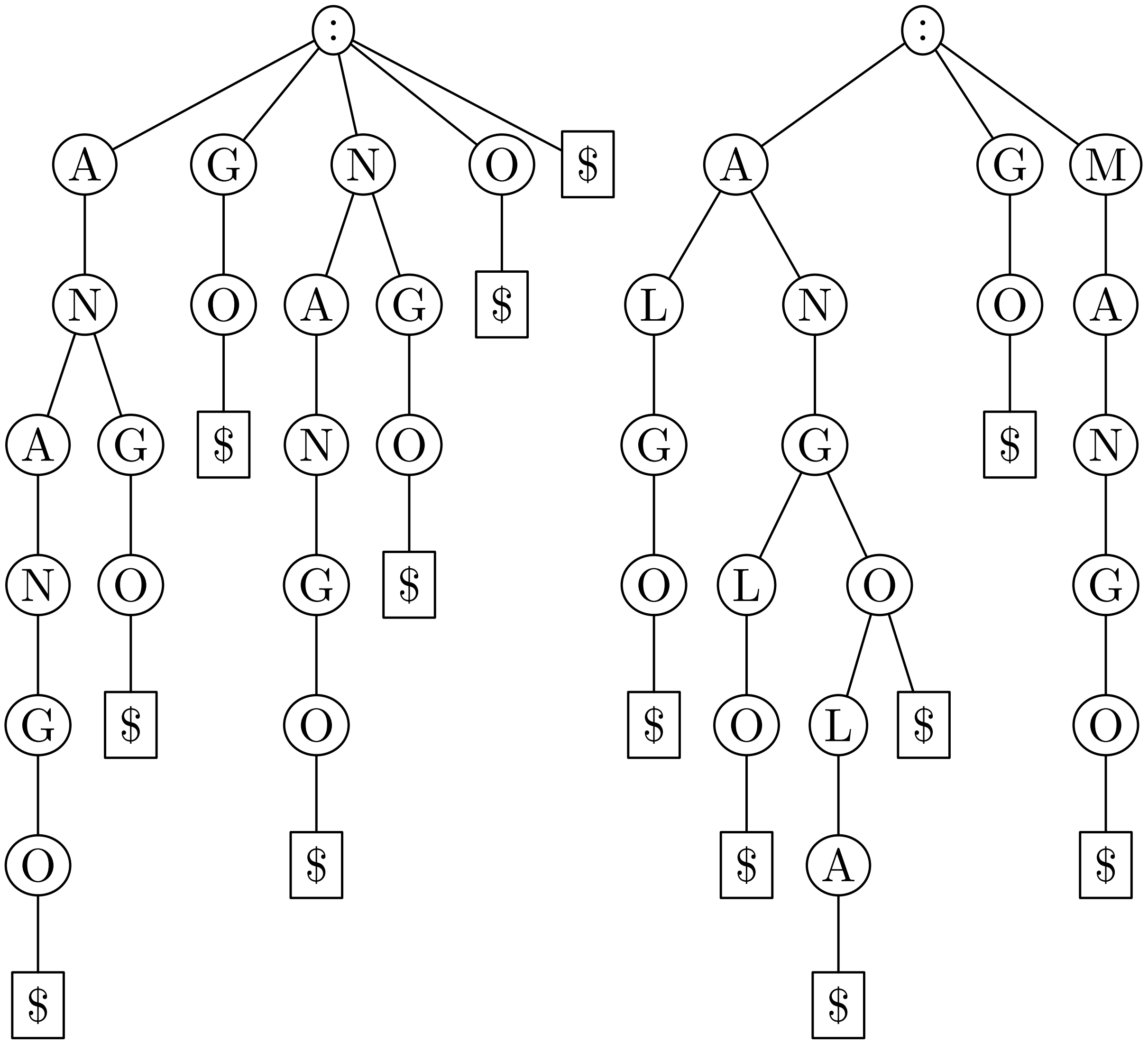
- Suppose we would like to find all substrings of pattern `ANANGO` in a trie.
- That is, we are interested in finding all prefixes of the following strings:
  - `ANANGO`
  - `NANGO`
  - `ANGO`
  - `NGO`
  - `GO`
  - `O`
- What should we do?

Exact all-against-all matching

- We should index the pattern string first.
- We now have two tries. We would like to find the common nodes of the tries.

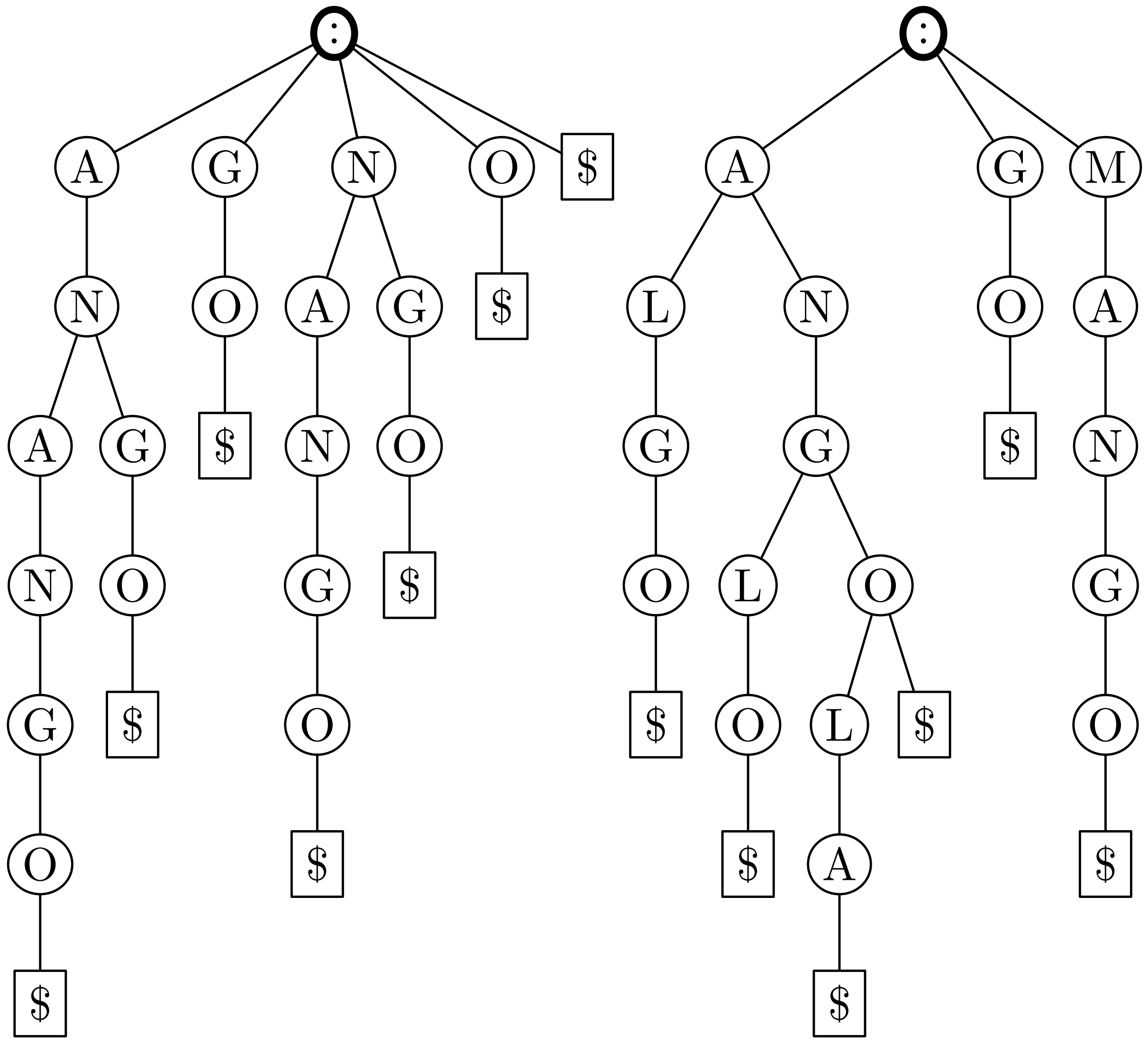
Exact all-against-all matching

- Common nodes:



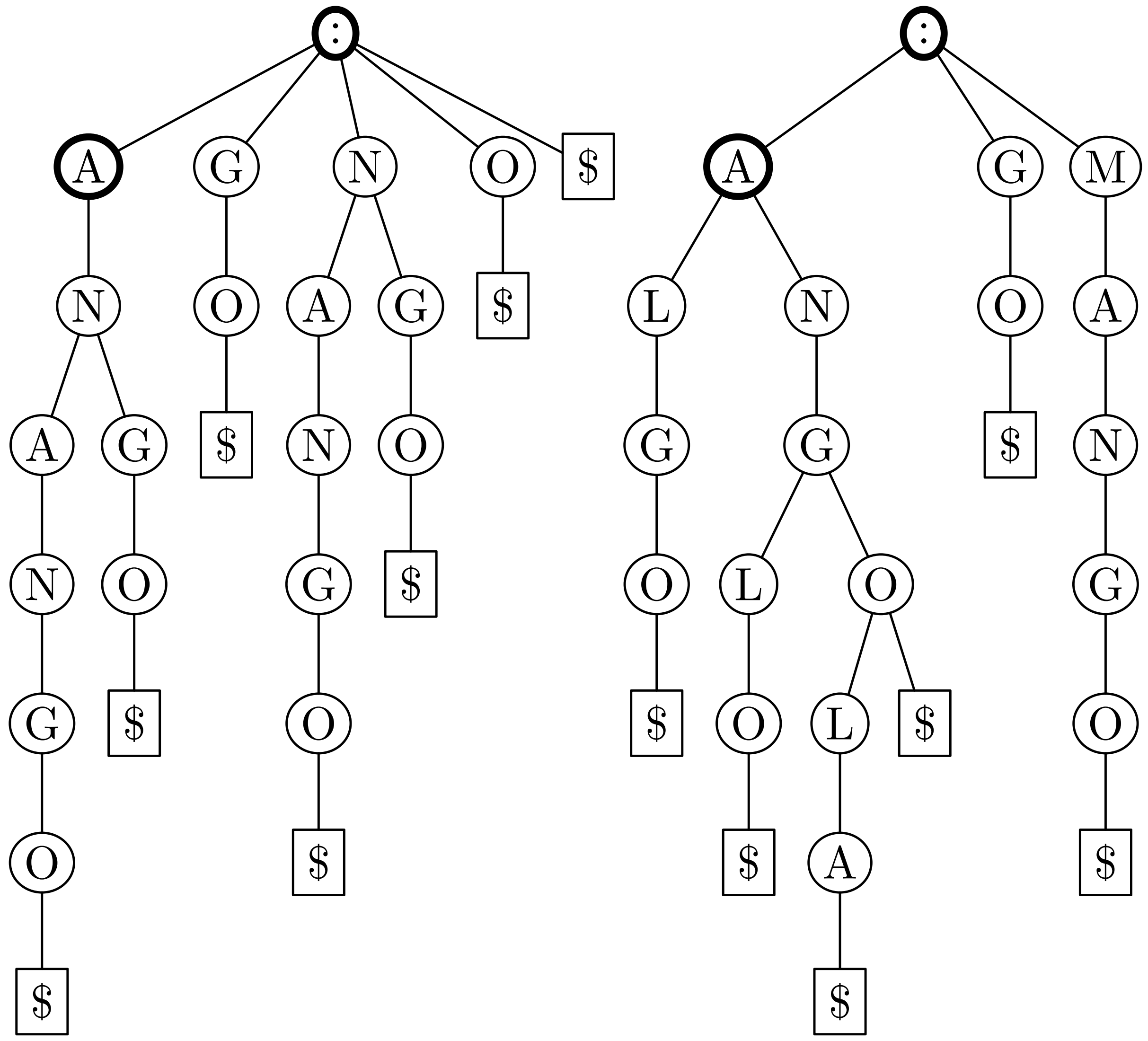
Exact all-against-all matching

- Common nodes:



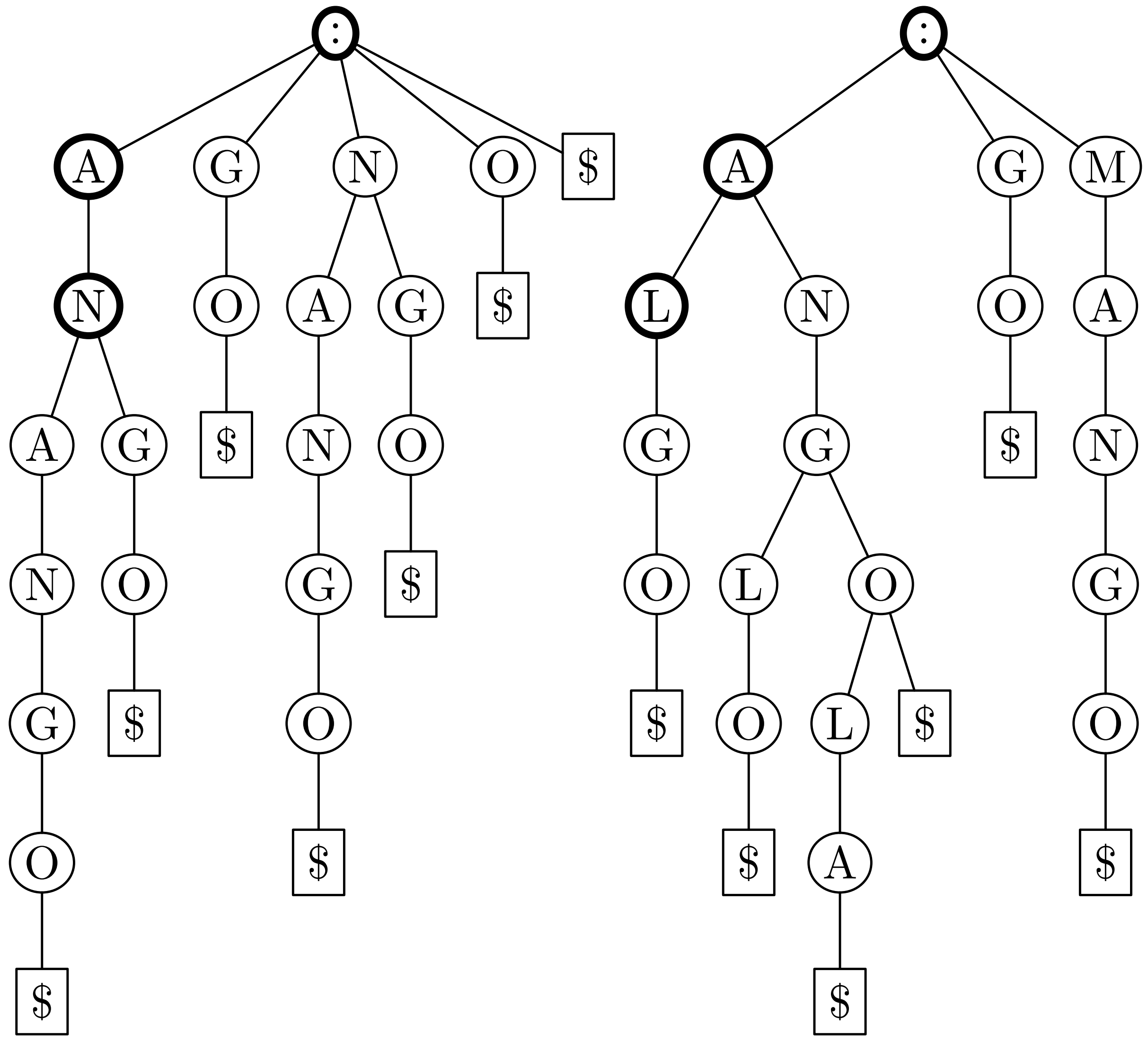
Exact all-against-all matching

- Common nodes:
  - A



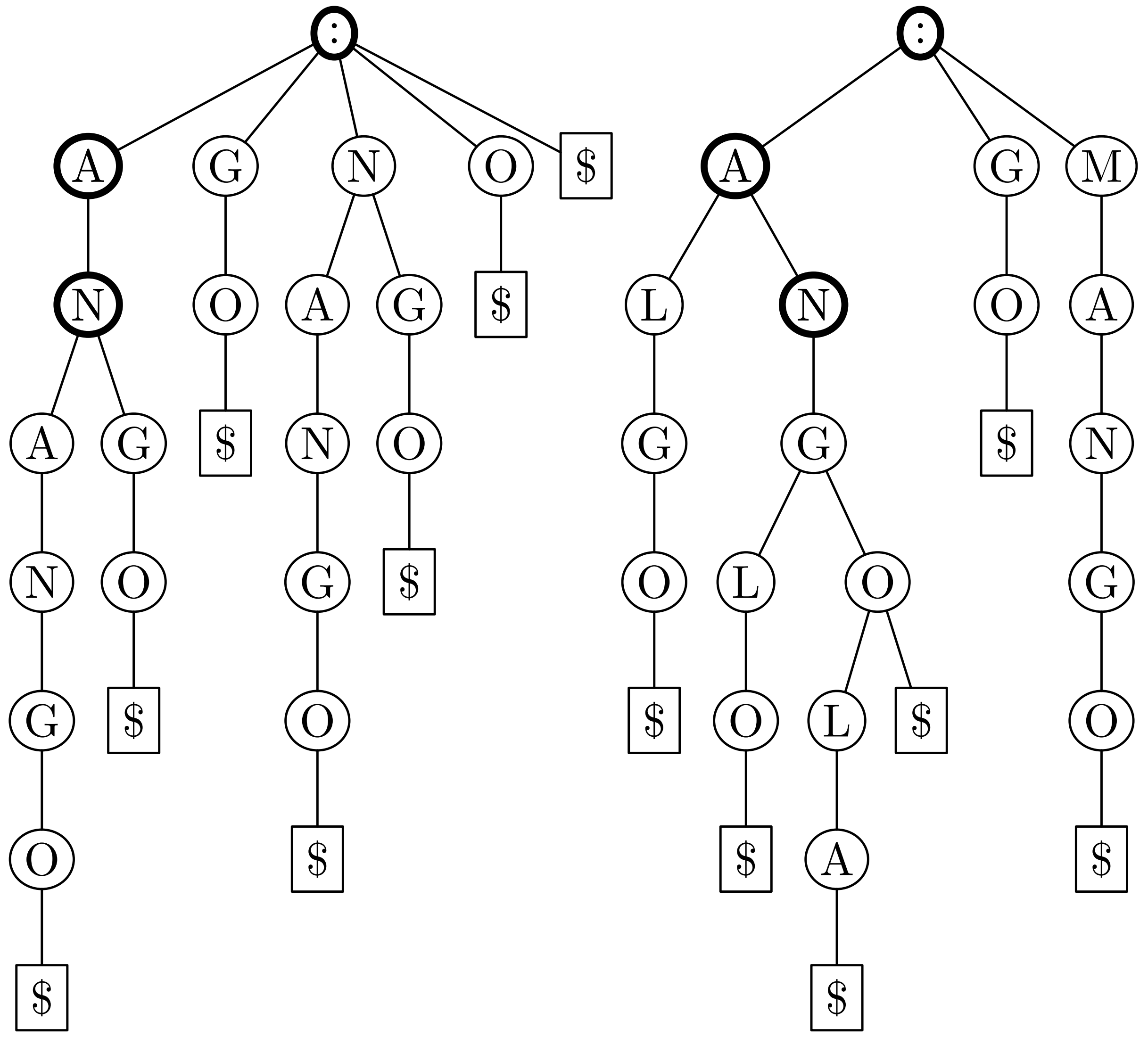
Exact all-against-all matching

- Common nodes:
  - A



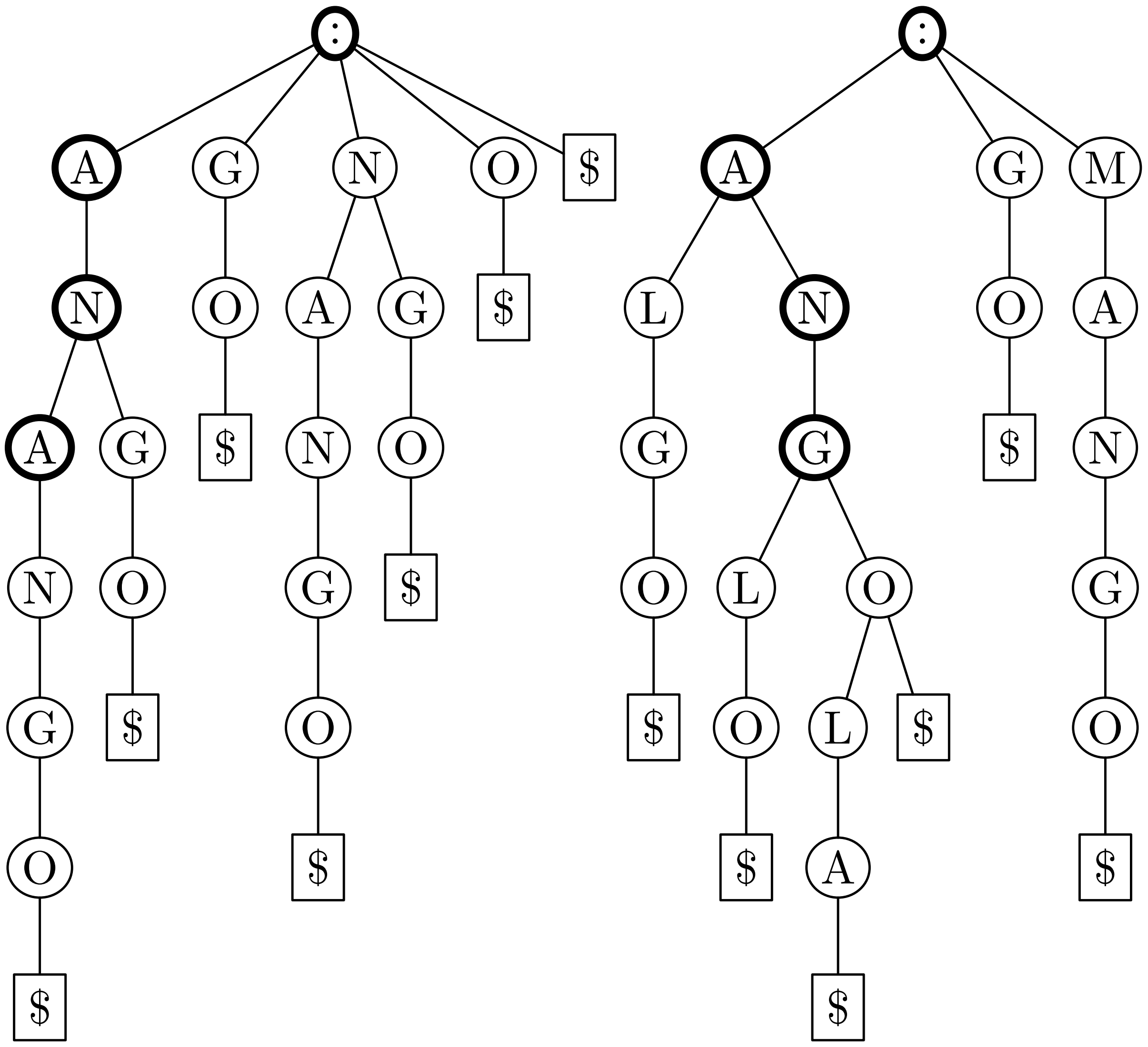
Exact all-against-all matching

- Common nodes:
  - A
  - AN



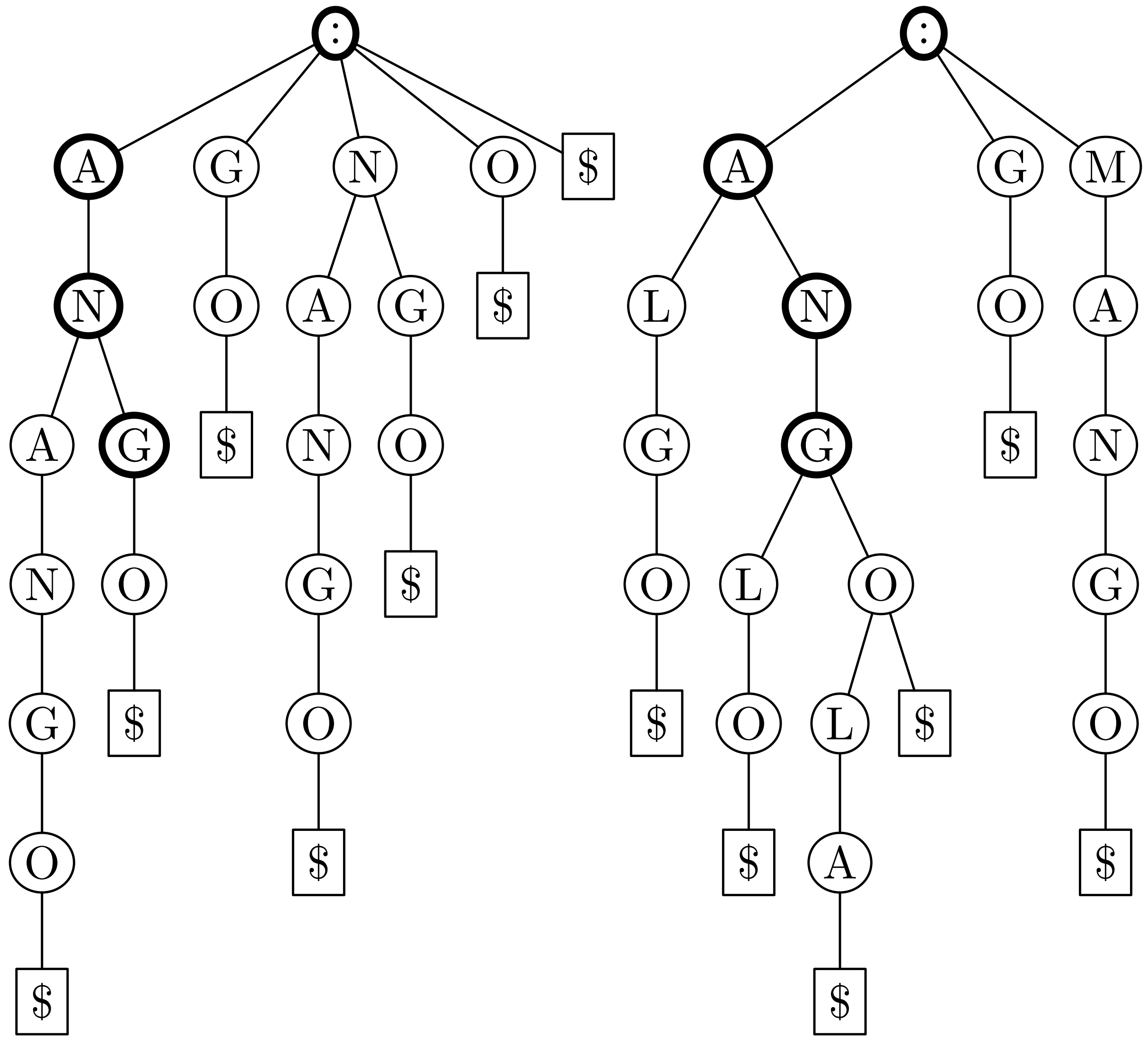
Exact all-against-all matching

- Common nodes:
  - A
  - AN



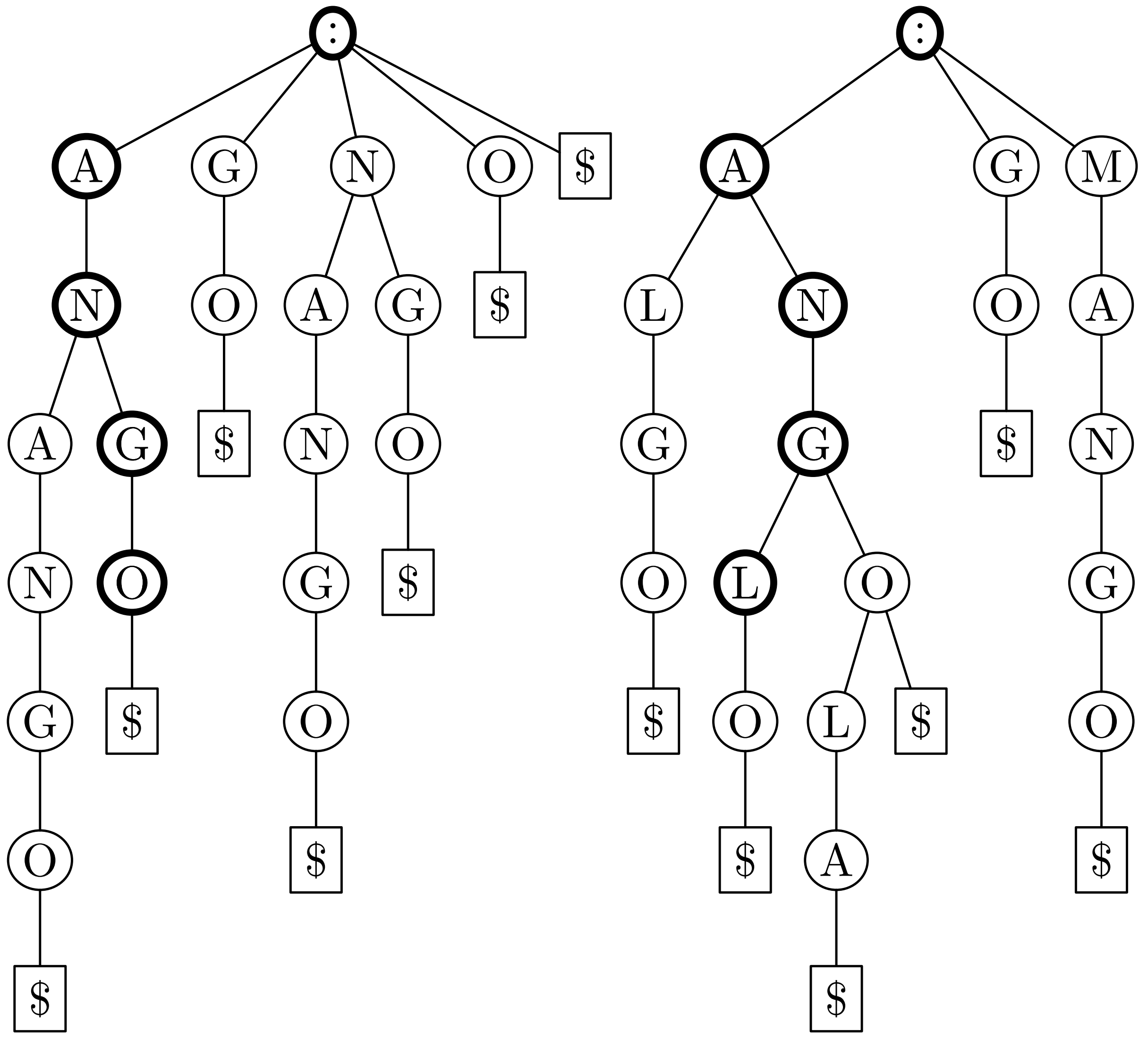
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG



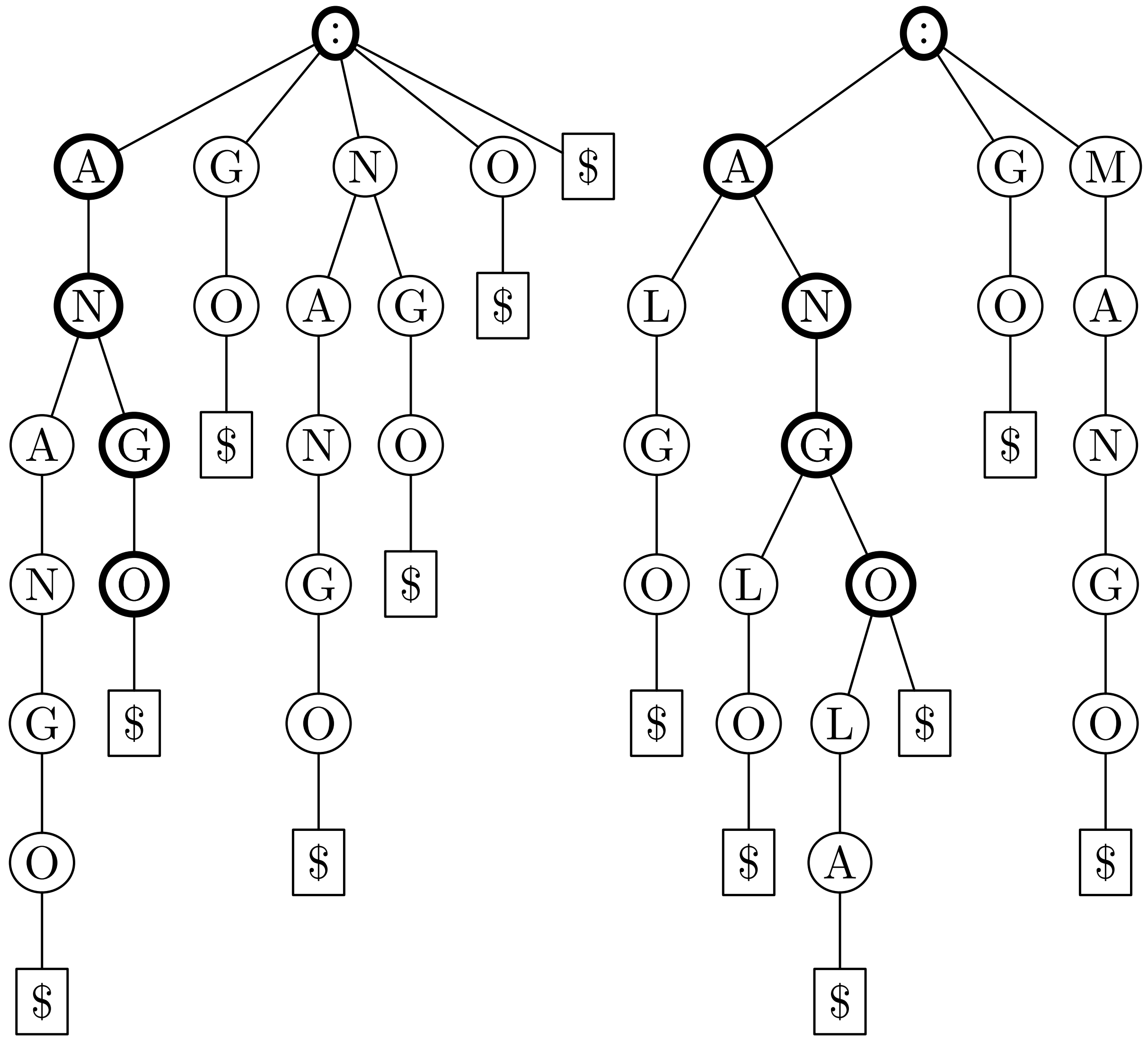
Exact all-against-all matching

- Common nodes:
    - A
    - AN
    - ANG



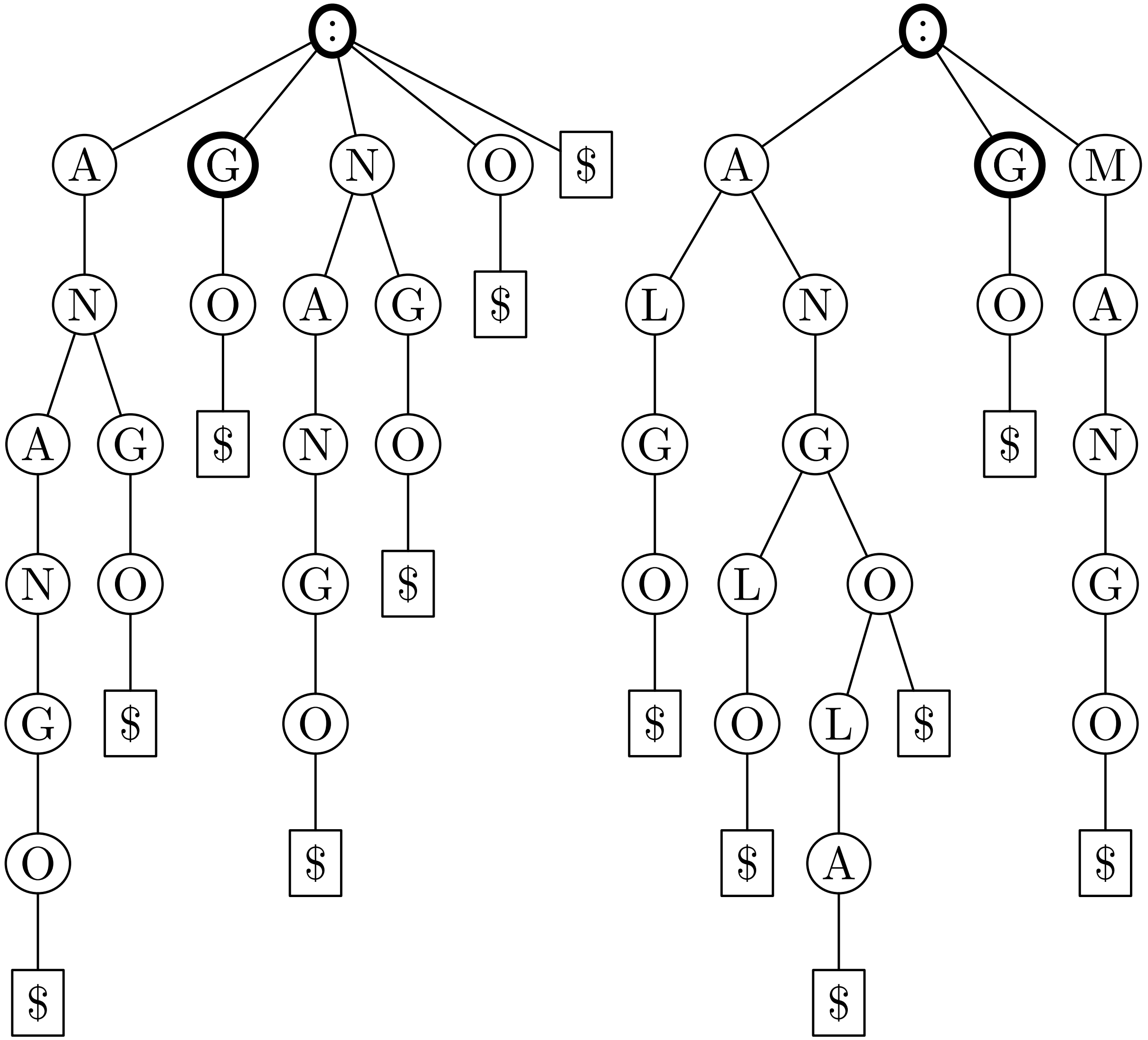
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG
  - ANGO



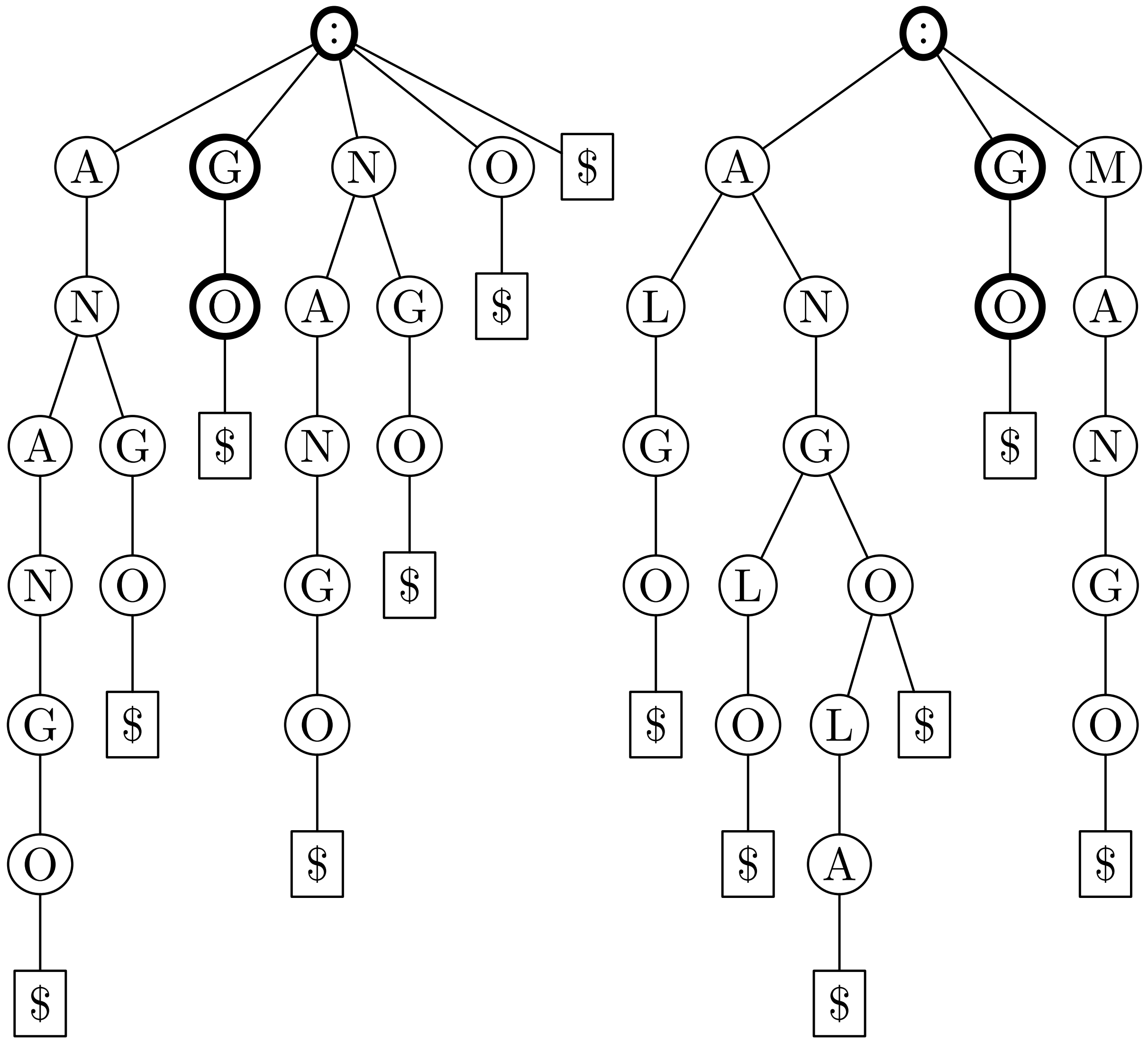
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG
  - ANGO
  - G



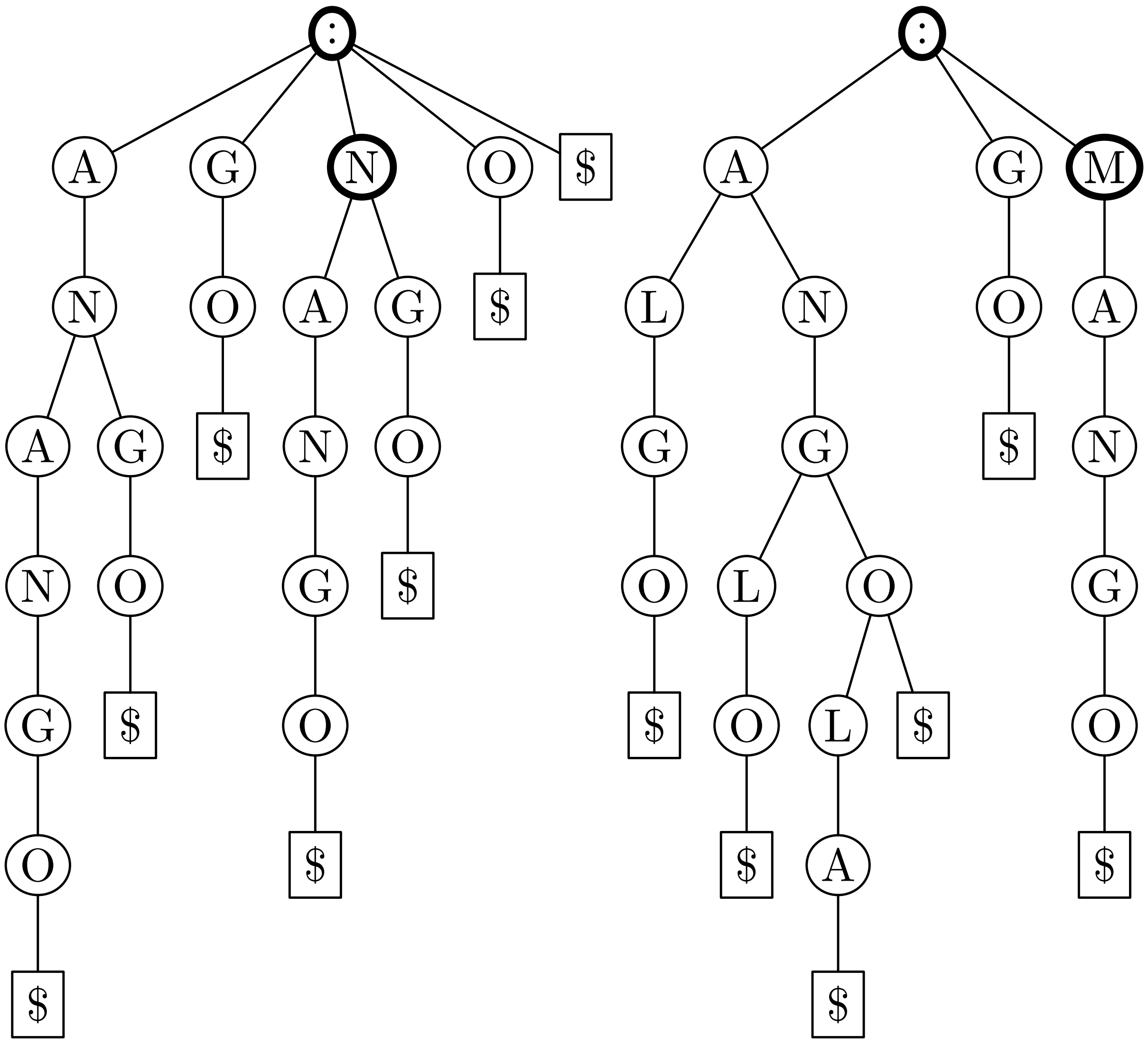
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG
  - ANGO
  - G
  - GO



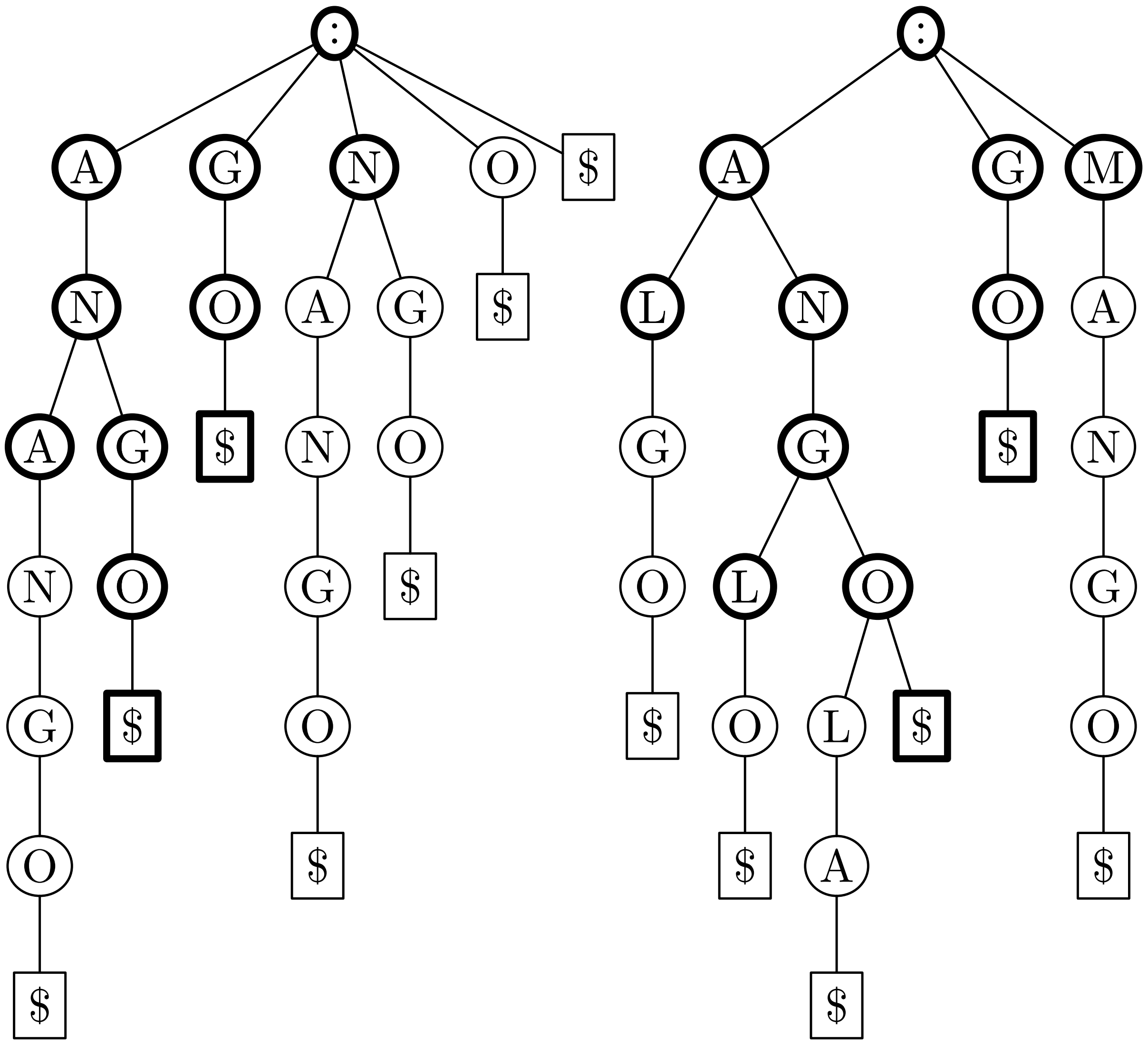
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG
  - ANGO
  - G
  - GO



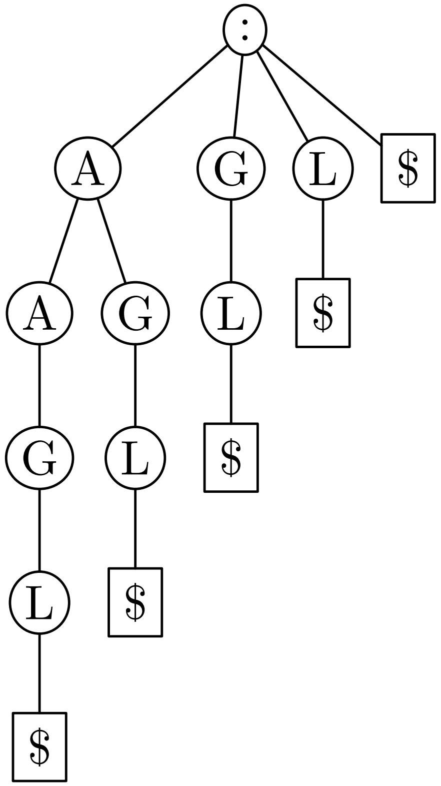
Exact all-against-all matching

- Common nodes:
  - A
  - AN
  - ANG
  - ANGO
  - G
  - GO

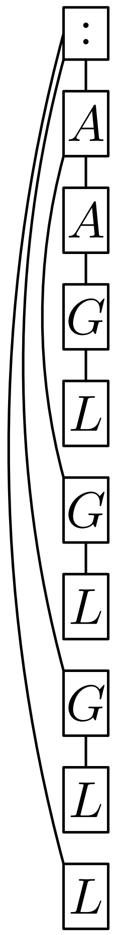


Approximate all-against-all matching

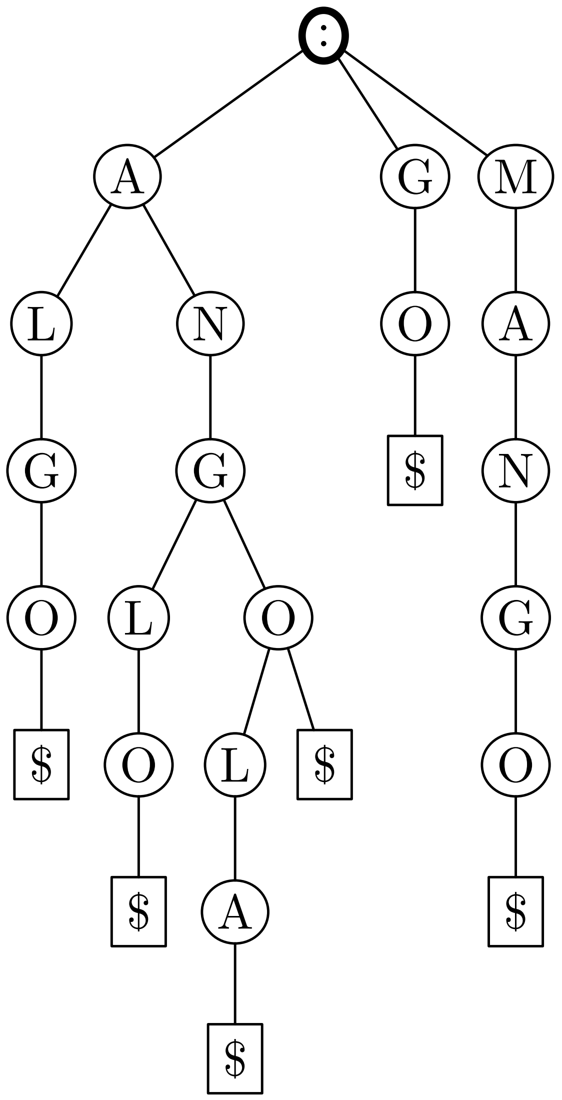
- Find all approximate occurrences of any substring of `AAGL`.
- Maximum edit distance 1.
- Suffix trie:



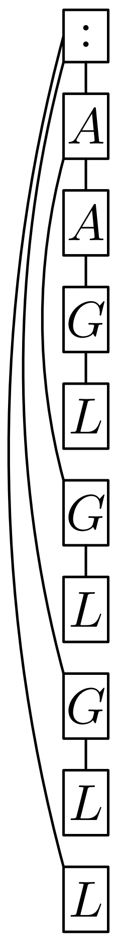
Approximate all-against-all matching



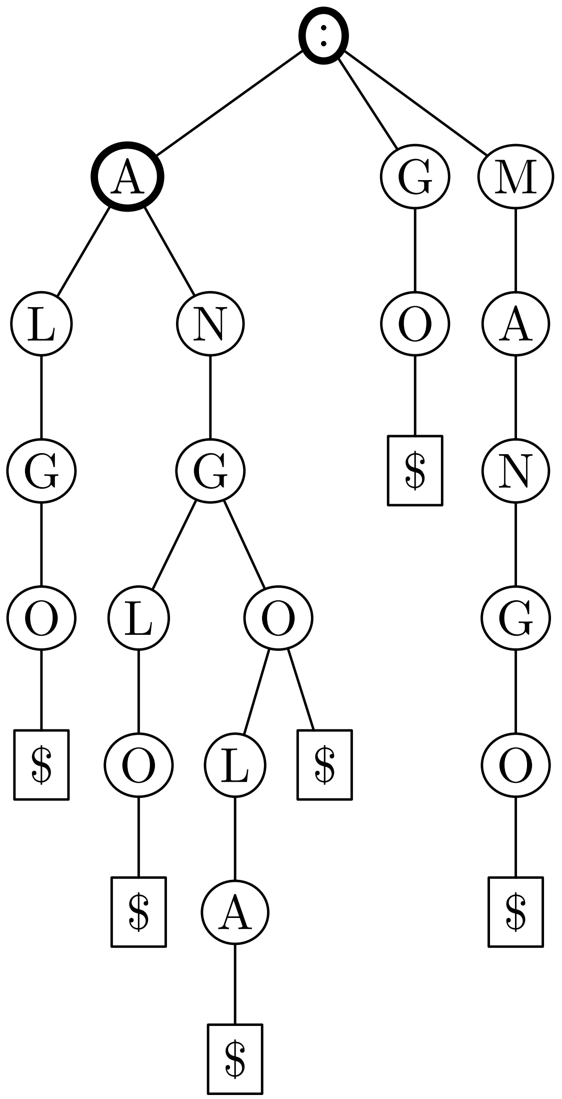
	:
	0
A	1
AA	
AAG	
AAGL	
AG	
AGL	
G	1
GL	
L	1



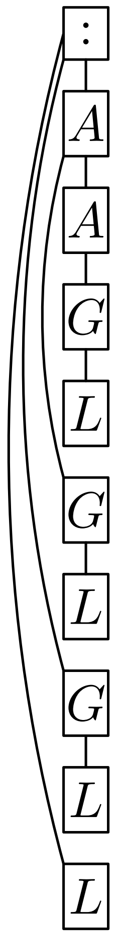
Approximate all-against-all matching



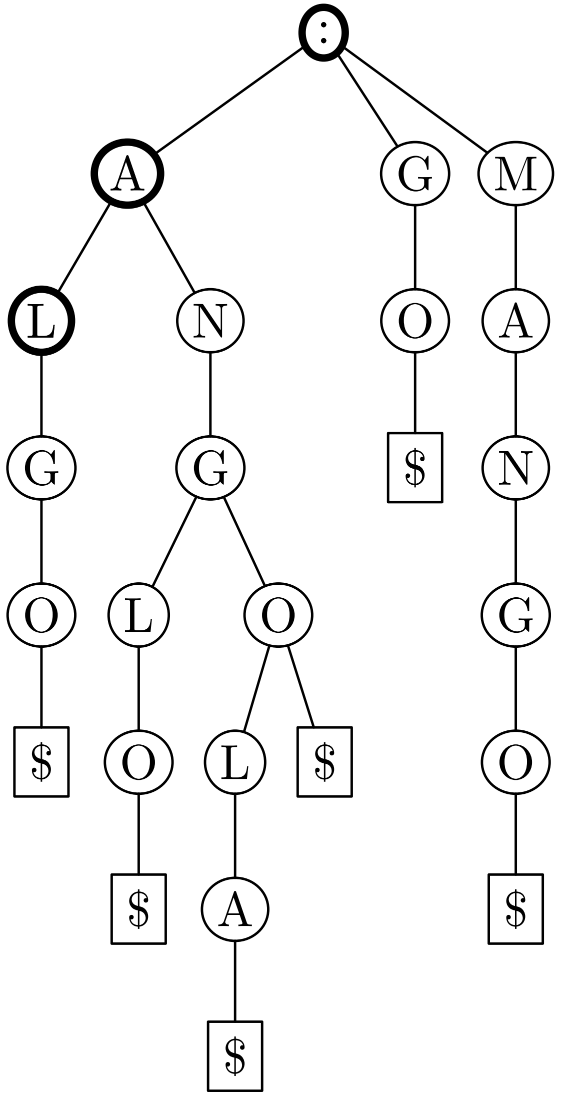
	:	A
	0	1
A	1	0
AA		1
AAG		
AAGL		
AG		1
AGL		
G	1	1
GL		
L	1	1



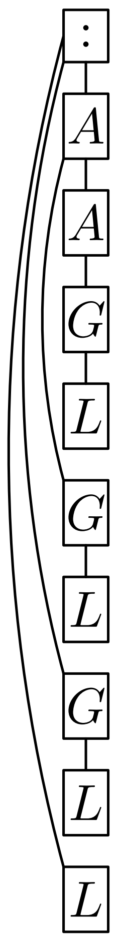
Approximate all-against-all matching



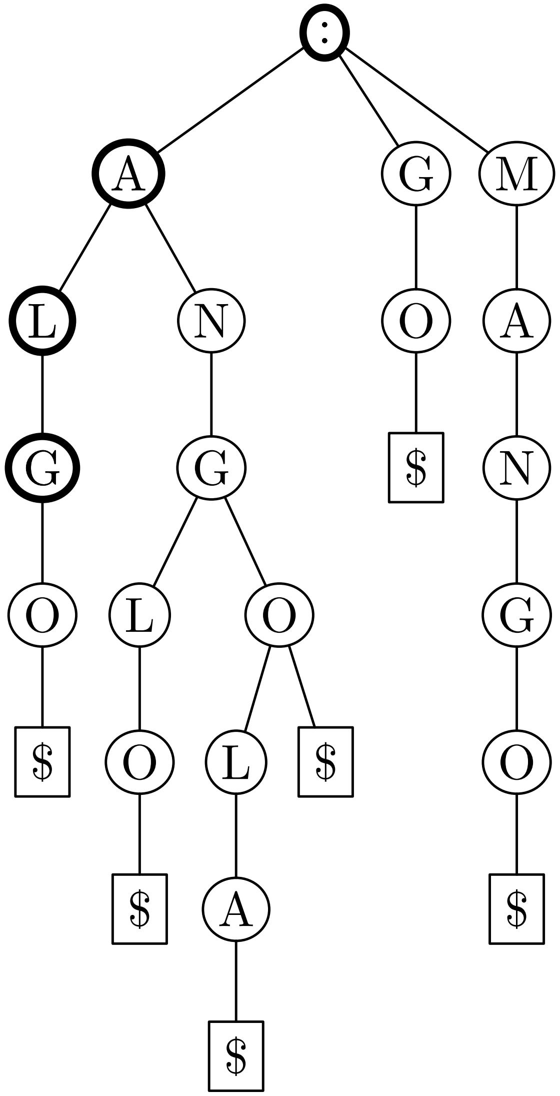
	:	A	L
	0	1	
A	1	0	1
aA		1	1
aaG			
aagL			
aG		1	1
agL			1
G	1	1	
gL			1
L	1	1	1



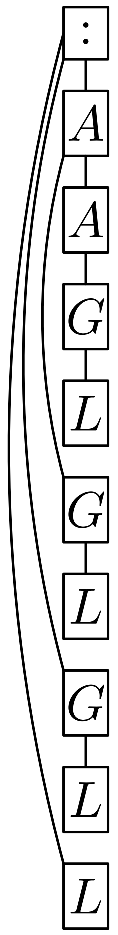
Approximate all-against-all matching



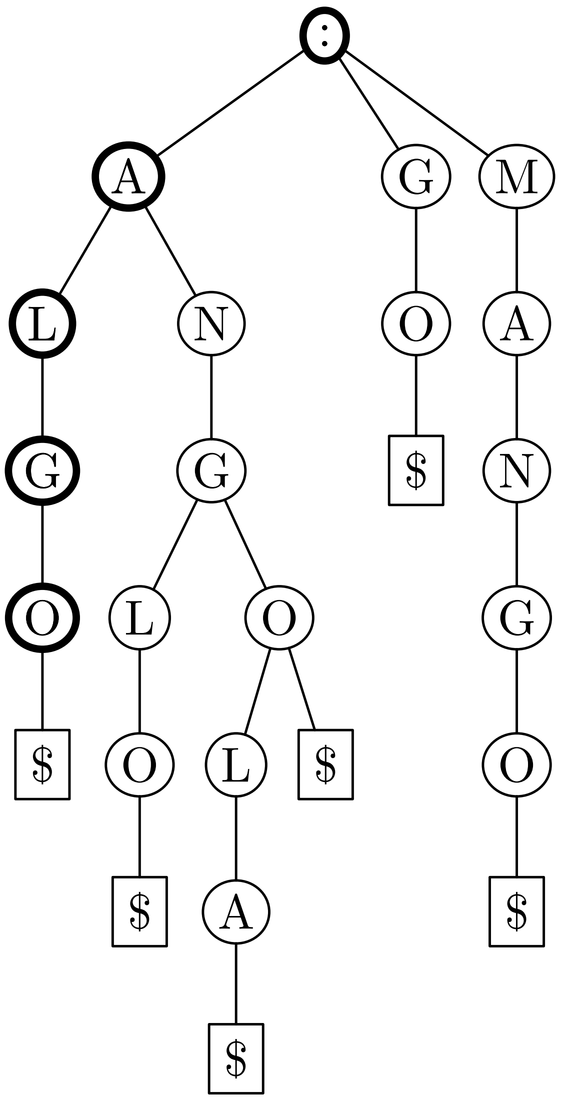
	:	A	L	G
	0	1		
A	1	0	1	
AA		1	1	
AAG				1
AAGL				
AG		1	1	1
AGL			1	
G	1	1		
GL			1	
L	1	1	1	



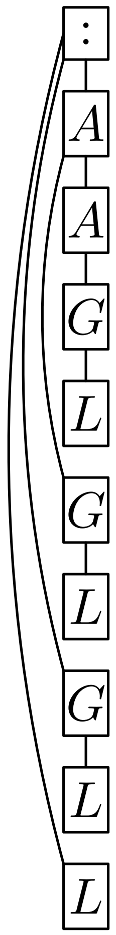
Approximate all-against-all matching



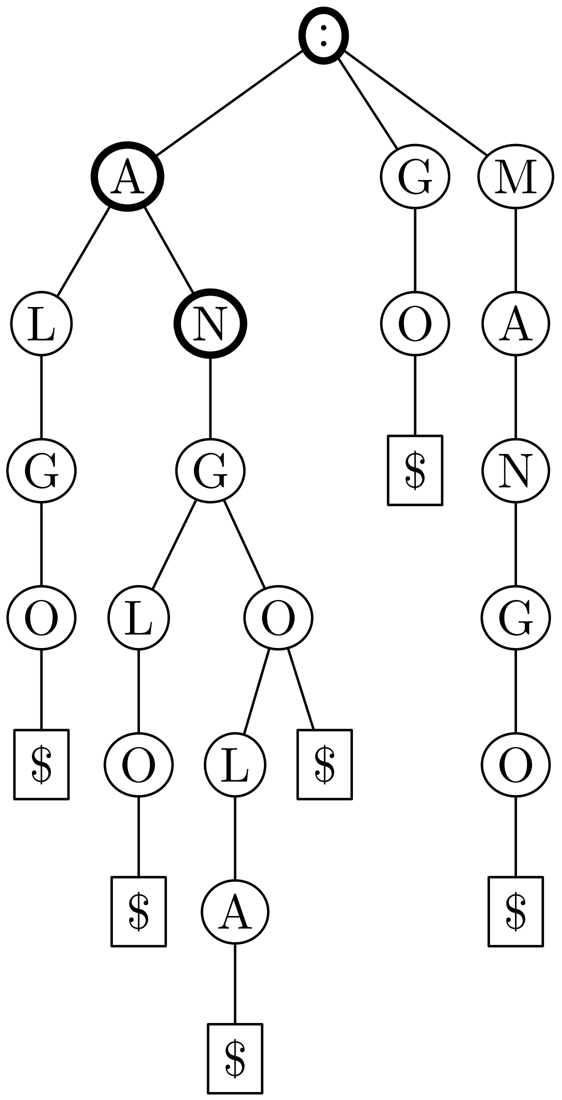
	:	A	L	G	O
	0	1			
A	1	0	1		
AA		1	1		
AAG				1	
AAGL					
AG		1	1	1	
AGL			1		
G	1	1			
GL			1		
L	1	1	1		



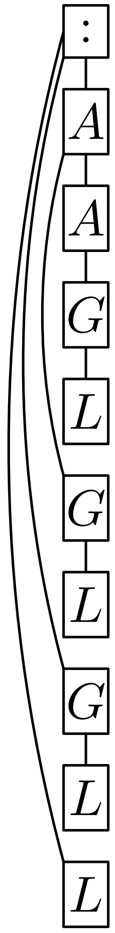
Approximate all-against-all matching



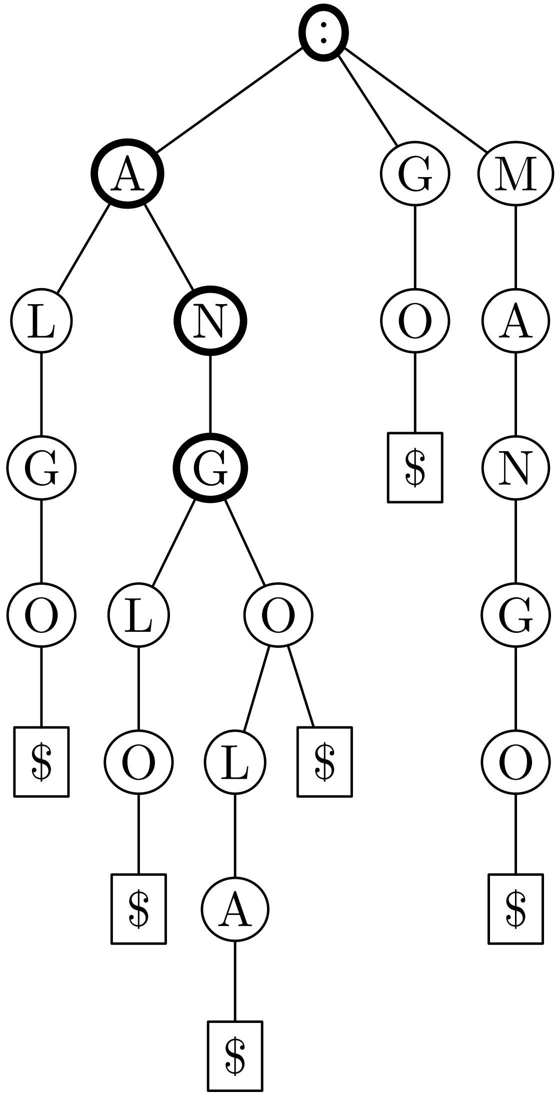
	:	A	N
	0	1	
A	1	0	1
AA		1	1
AAG			
AAGL			
AG		1	1
AGL			
G	1	1	
GL			
L	1	1	



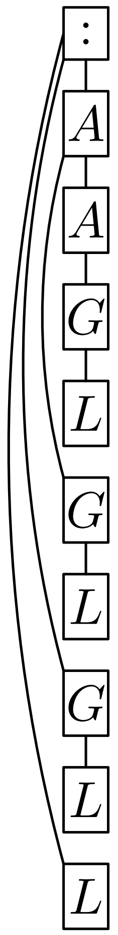
Approximate all-against-all matching



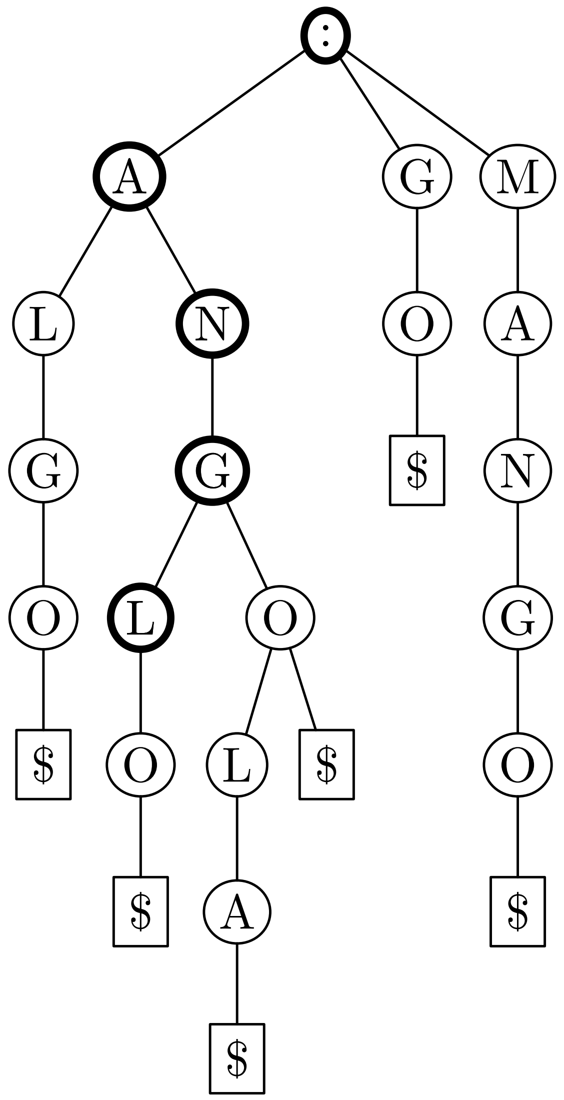
	:	A	N	G
	0	1		
A	1	0	1	
AA		1	1	
AAG				1
AAGL				
AG		1	1	1
AGL				
G	1	1		
GL				
L	1	1		



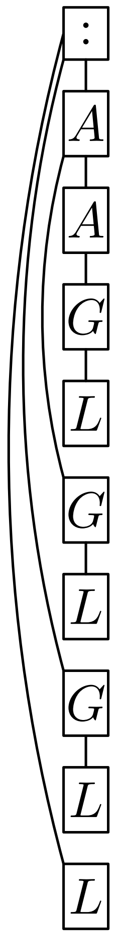
Approximate all-against-all matching



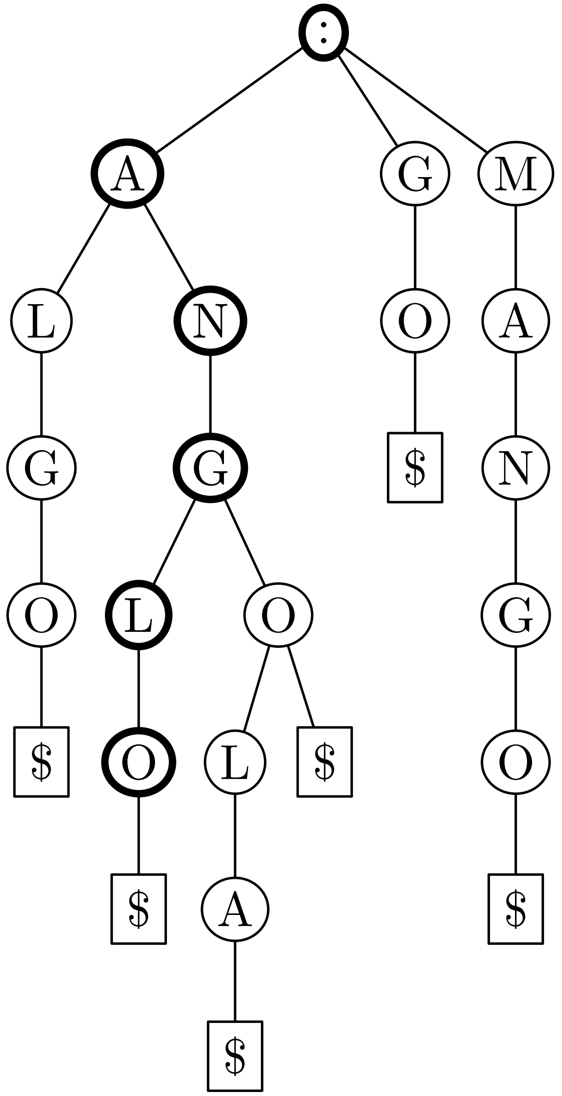
	:	A	N	G	L
	0	1			
A	1	0	1		
AA		1	1		
AAG				1	
AAGL					1
AG		1	1	1	
AGL					1
G	1	1			
GL					
L	1	1			



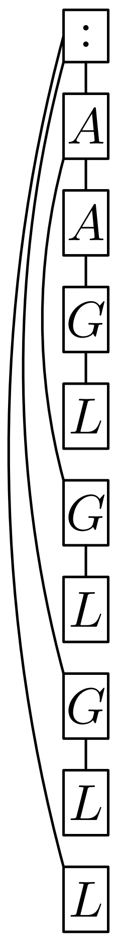
Approximate all-against-all matching



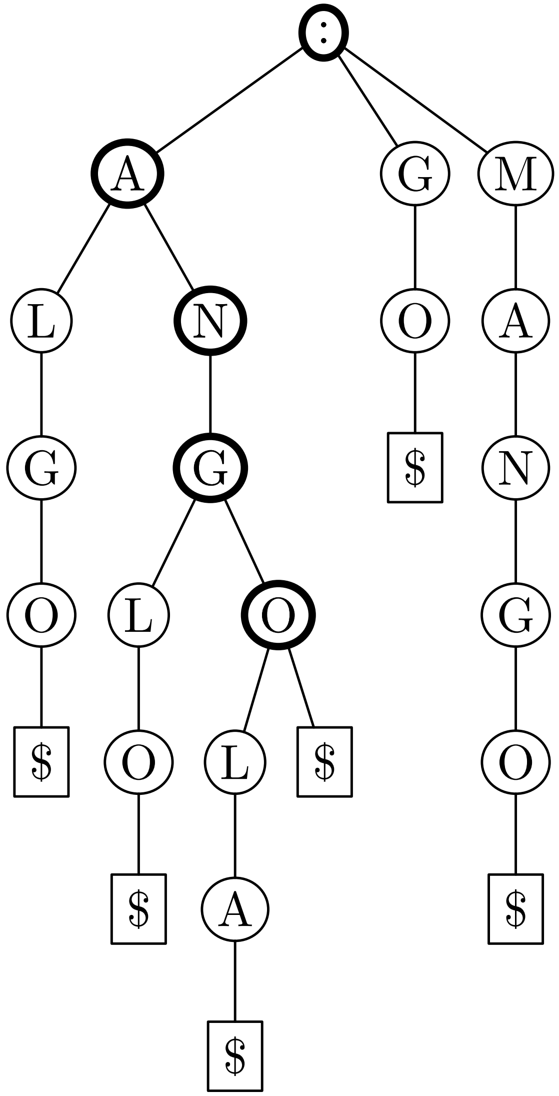
	:	A	N	G	L	O
	0	1				
A	1	0	1			
AA		1	1			
AAG				1		
AAGL					1	
AG		1	1	1		
AGL					1	
G	1	1				
GL						
L	1	1				



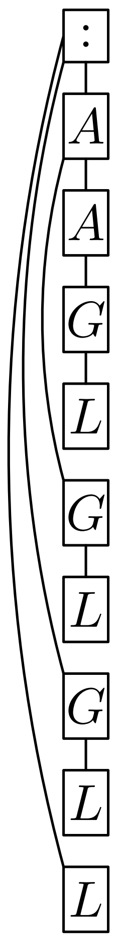
Approximate all-against-all matching



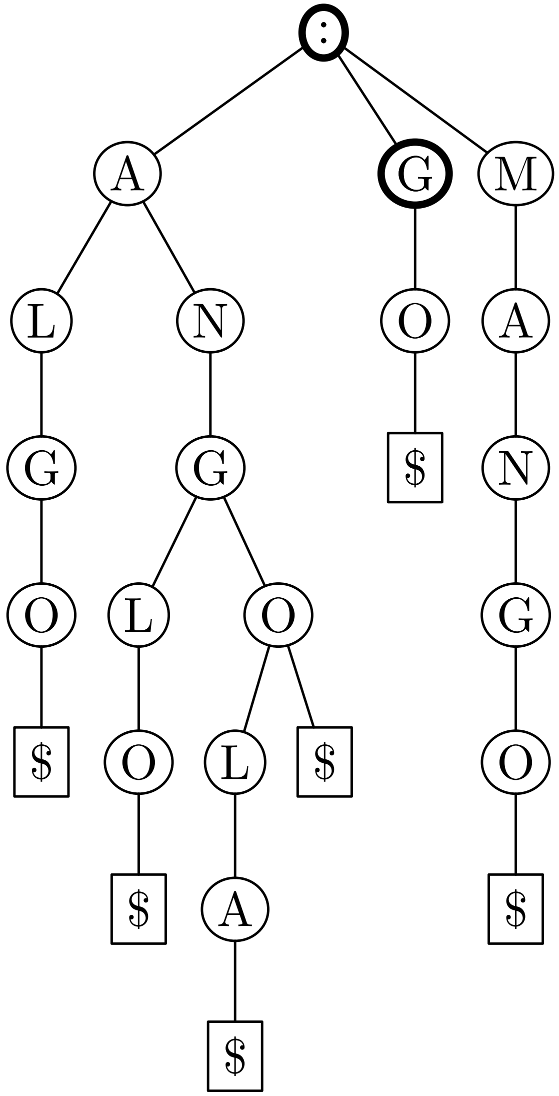
	:	A	N	G	O
	0	1			
A	1	0	1		
AA		1	1		
AAG				1	
AAGL					
AG		1	1	1	
AGL					
G	1	1			
GL					
L	1	1			



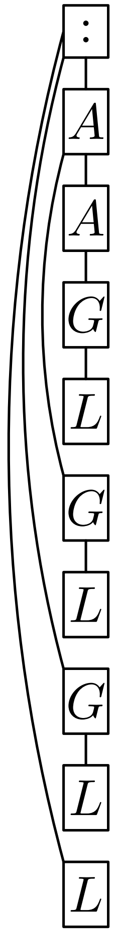
Approximate all-against-all matching



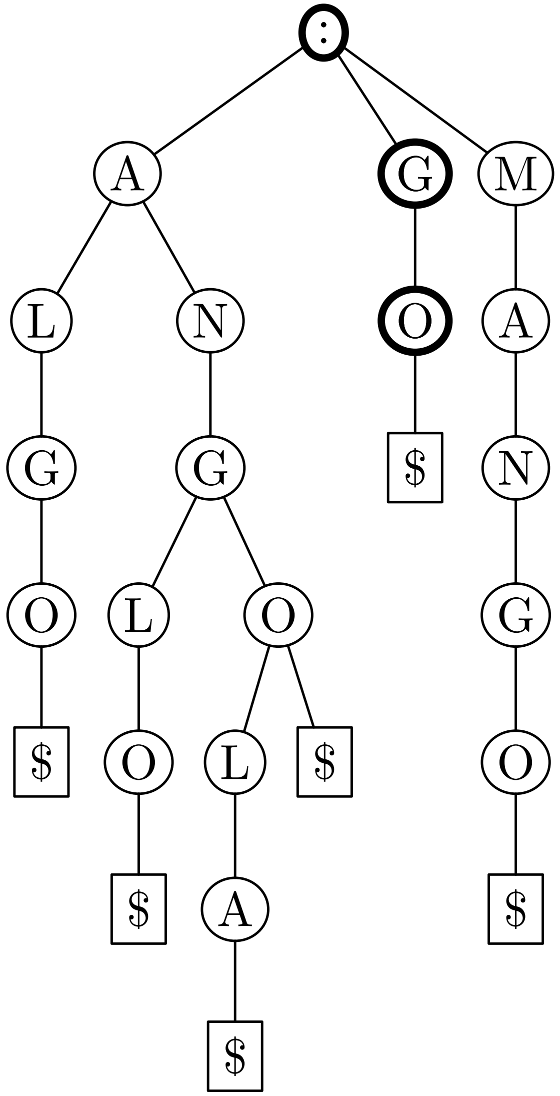
	:	G
	0	1
A	1	1
AA		
AAG		
AAGL		
AG		1
AGL		
G	1	0
GL		1
L	1	1



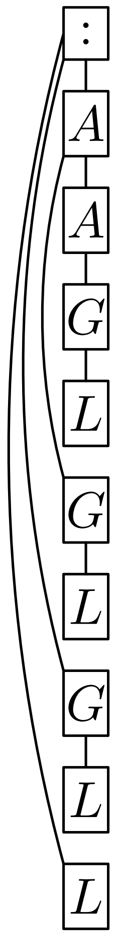
Approximate all-against-all matching



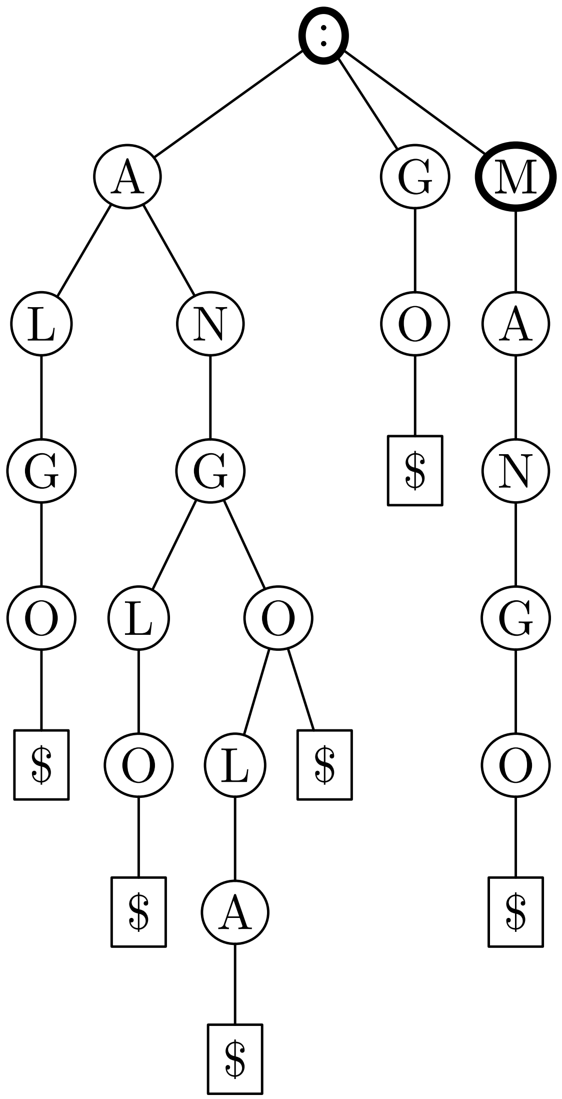
	:	G	O
	0	1	
A	1	1	
AA			
AAG			
AAGL			
AG		1	
AGL			
G	1	0	1
GL		1	1
L	1	1	



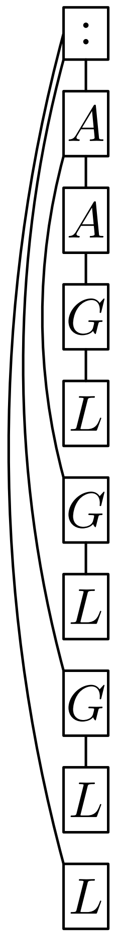
Approximate all-against-all matching



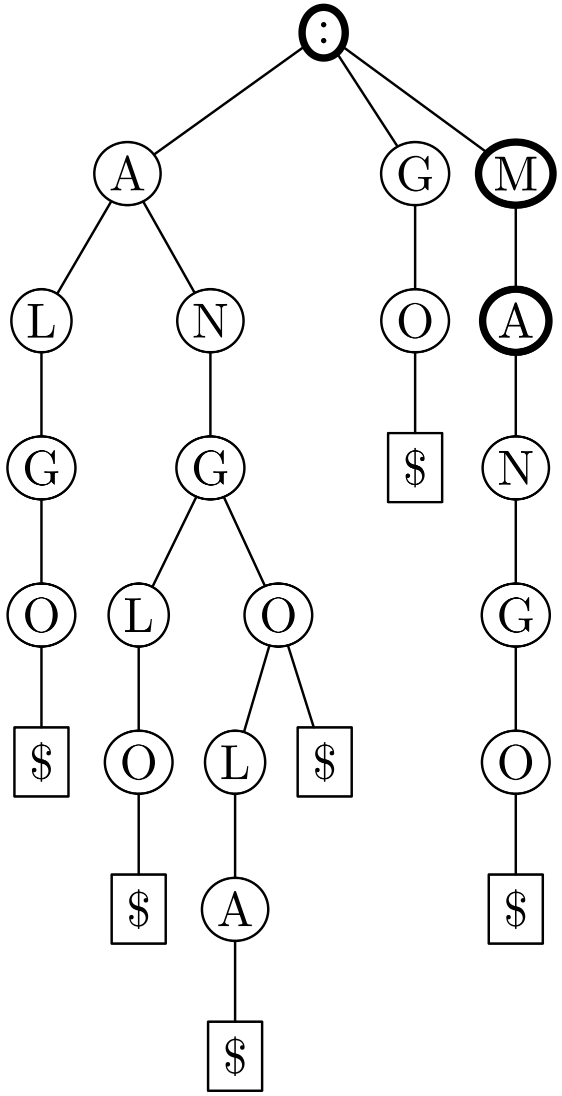
	:	M
	0	1
A	1	1
AA		
AAG		
AAGL		
AG		
AGL		
G	1	1
GL		
L	1	1



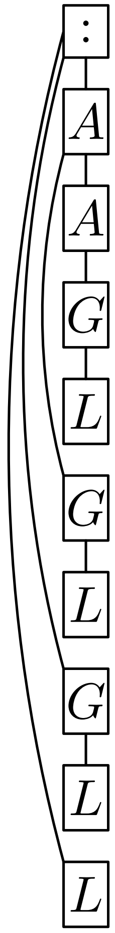
Approximate all-against-all matching



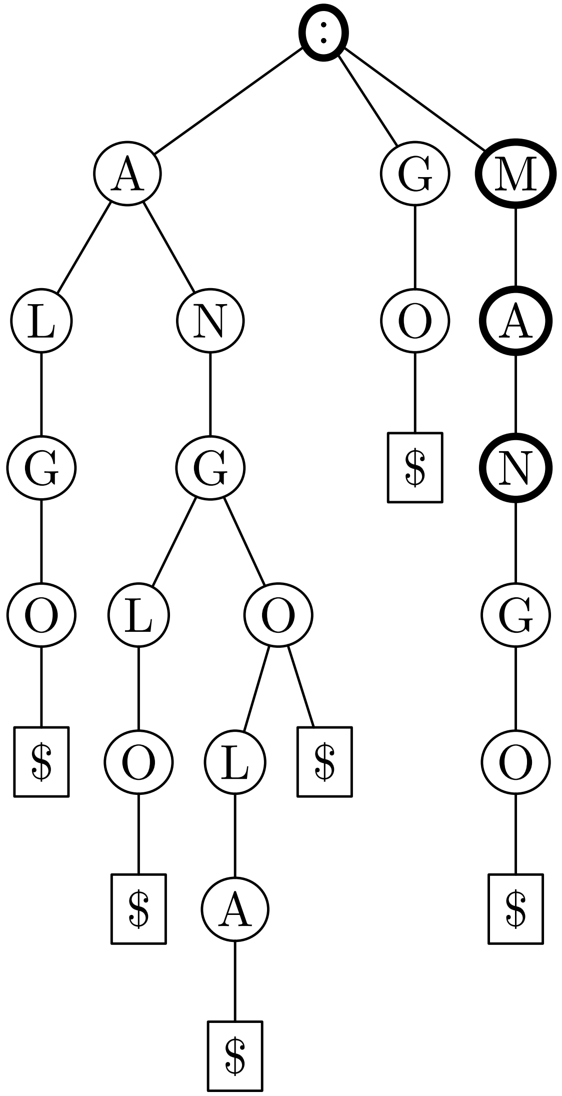
	:	M	A
	0	1	
A	1	1	1
AA			1
AAG			
AAGL			
AG			
AGL			
G	1	1	
GL			
L	1	1	



Approximate all-against-all matching



	:	M	A	N
	0	1		
A	1	1	1	
AA			1	
AAG				
AAGL				
AG				
AGL				
G	1	1		
GL				
L	1	1		



Implementing approximate all-against-all

- We cannot use the same DP table as in approximate string matching algorithm
- This is because the elements we want to calculate are not always close to the main diagonal.

Implementing approximate all-against-all

- We cannot use the same DP table as in approximate string matching algorithm
- This is because the elements we want to calculate are not always close to the main diagonal.
- Instead of the error column, we use a list for each node in the text trie.
- Each element of a list is pair $(pos, error)$, where pos is the position in pattern trie and $error$ is the corresponding error table value.

Implementing approximate all-against-all

- $M_{i,j} \leftarrow \min(M_{i-1,j-1} + \delta(x_i, y_i),\ M_{i-1,j} + 1,\ M_{i,j-1} + 1)$ which worked in approximate string matching, will not work here.
- Each element in list for column $j - 1$ gives new elements to list for column j.
- Duplicates are removed from new list.

Experiment results

- All tests are performed in alphabet $\Sigma = \{A, C, G, T\}$.
- All texts and patterns are random.
- The computer was $2.8GHz$ Pentium 4.
- The text consisted of 100000 lines, each line containing 100 symbols and additional newline `'\n'`, total size $9.63MB$.

- The creation of text took 2.6 seconds (size $9.63MB$).
- The creation of index took 34.0 seconds (size $95.2MB$).
- All searching times are in milliseconds and do not contain the time for outputting matches.

tagrep vs. agrep

Error	Program	Length of pattern				
		5	10	15	20	25
1	tagrep	649	82	82	81	72
	agrep	286	185	241	275	366
2	tagrep	6908	121	97	105	94
	agrep	247	301	312	319	525
3	tagrep	29904	542	218	191	193
	agrep	235	395	447	458	1374
4	tagrep		5314	715	633	697
	agrep	226	242	456	480	2438
5	tagrep		18602	2435	2121	2439
	agrep		261	496	581	3422

Experiment results

- Finding all exact substrings of length 10 or more of a pattern of 10000 symbols from the $10MB$ text took 0.18 seconds.
- Finding all approximate substrings with error 1 (other parameters are same) took 13.7 seconds.
- Finding approximate substrings with error 1 and length 20 took 4.1 seconds.
- Finding approximate substrings with error 2 and length 20 took 43.8 seconds.

Conclusion

- Suffix tries are useful, when we need to make several queries from the same text.
- Tagrep beats agrep!

Questions?