Range Filters

(SuRF, Memento Filter, Diva)

Niv Dayan - CSC2525: Research Topics in Database Management

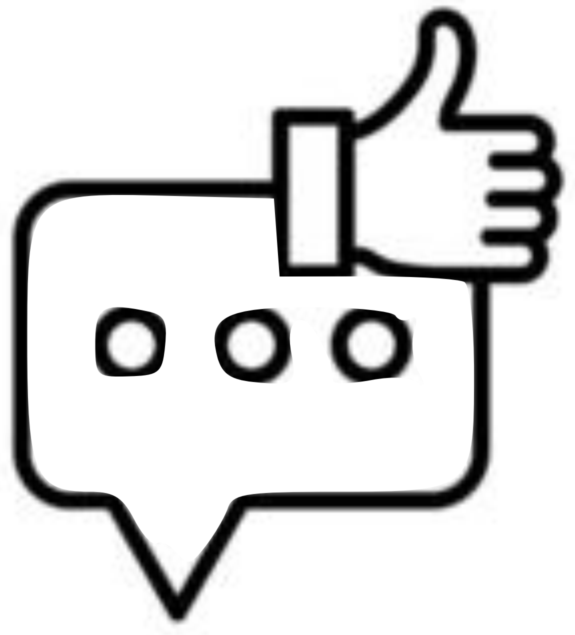
Last lecture

This was fun…

Logistics

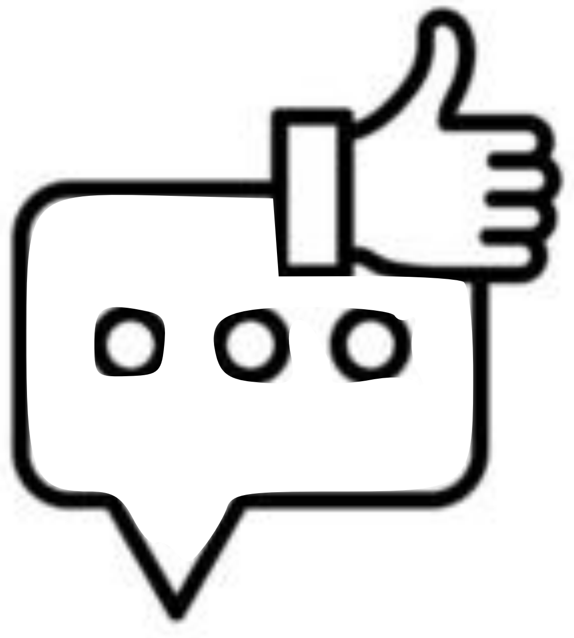
Projects due
on April 9

Oral Exams
April 8-10



Course
Evaluation

Course Evaluation



Helps us improve

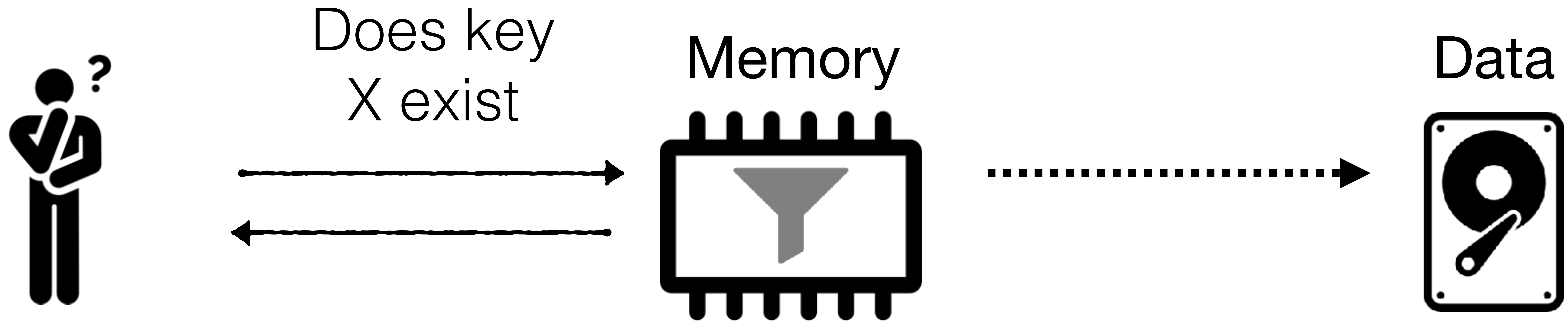
What is a Filter?

Does X exist? →

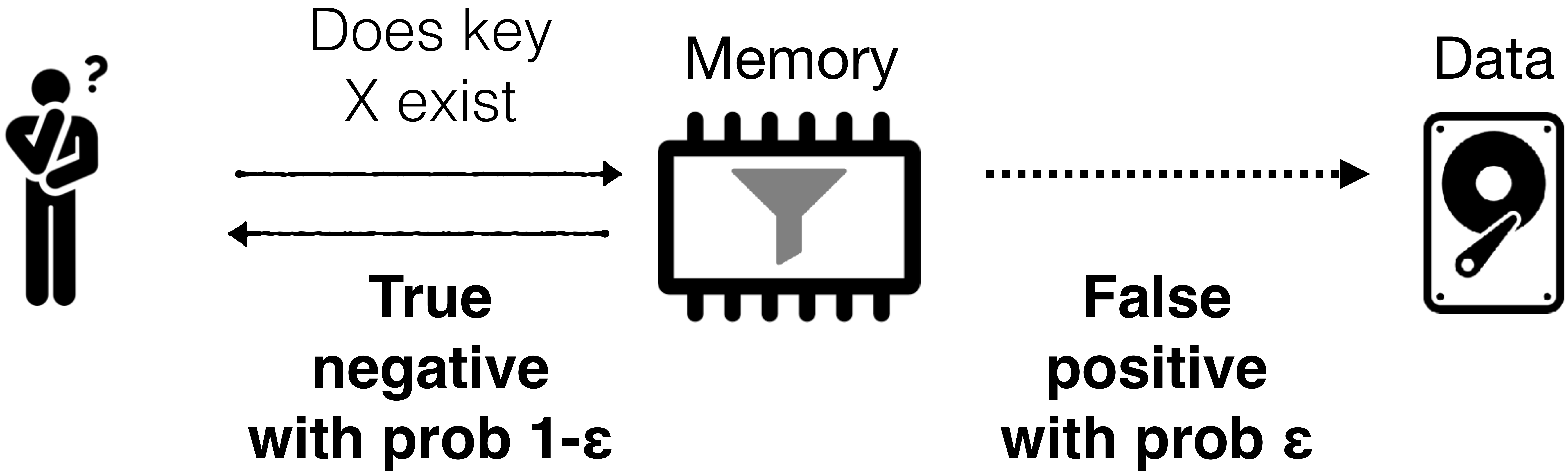
Set

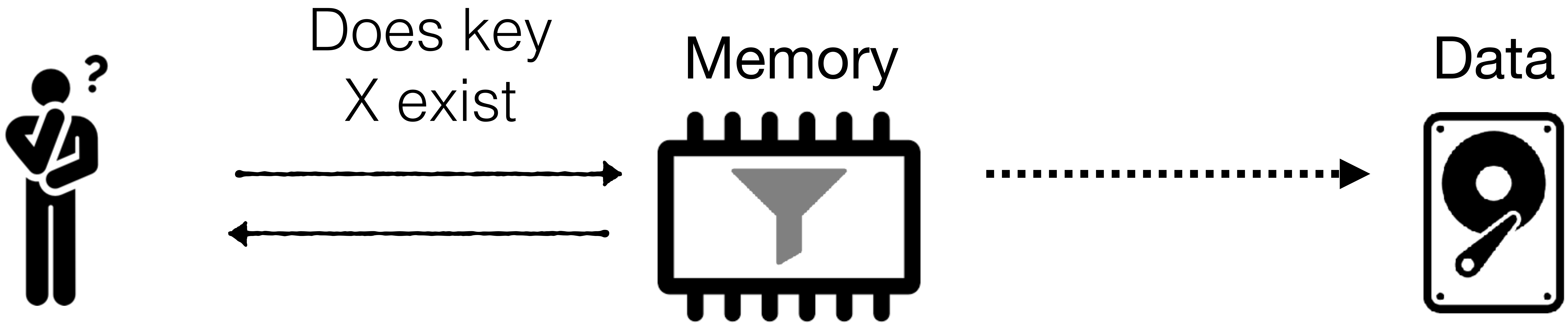
X Y Z

If key X does not exist



If key X does not exist





Saves storage accesses & network hops

only support point queries (to one key)

only support point queries (to one key)

How about a range?

How about a range?

Does anything in [A, B] exist? →

Set

X Y Z

How about a range?

Does anything in [3, 5] exist?

How about a range?

Does anything in
[3, 5] exist?

Set

1 4 9

True positive

Does anything in
[5, 8] exist?

→

Set

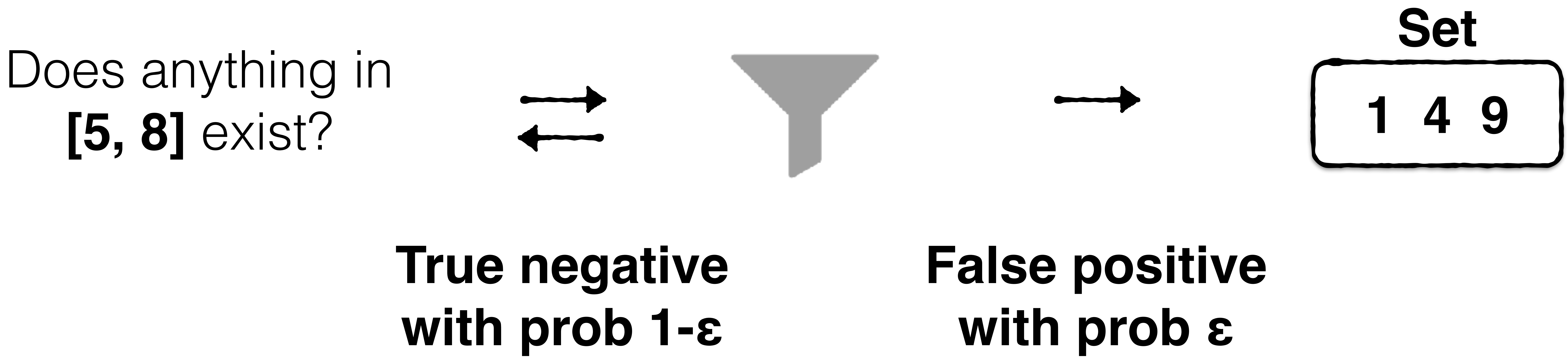
1 4 9

Does anything in
[5, 8] exist?

Set

1 4 9

**True negative
with prob 1-ε**


Does anything in
[5, 8] exist?
True negative
with prob 1-ε
False positive
with prob ε
Set
1 4 9

Applications?

Does anything in
[5, 8] exist?

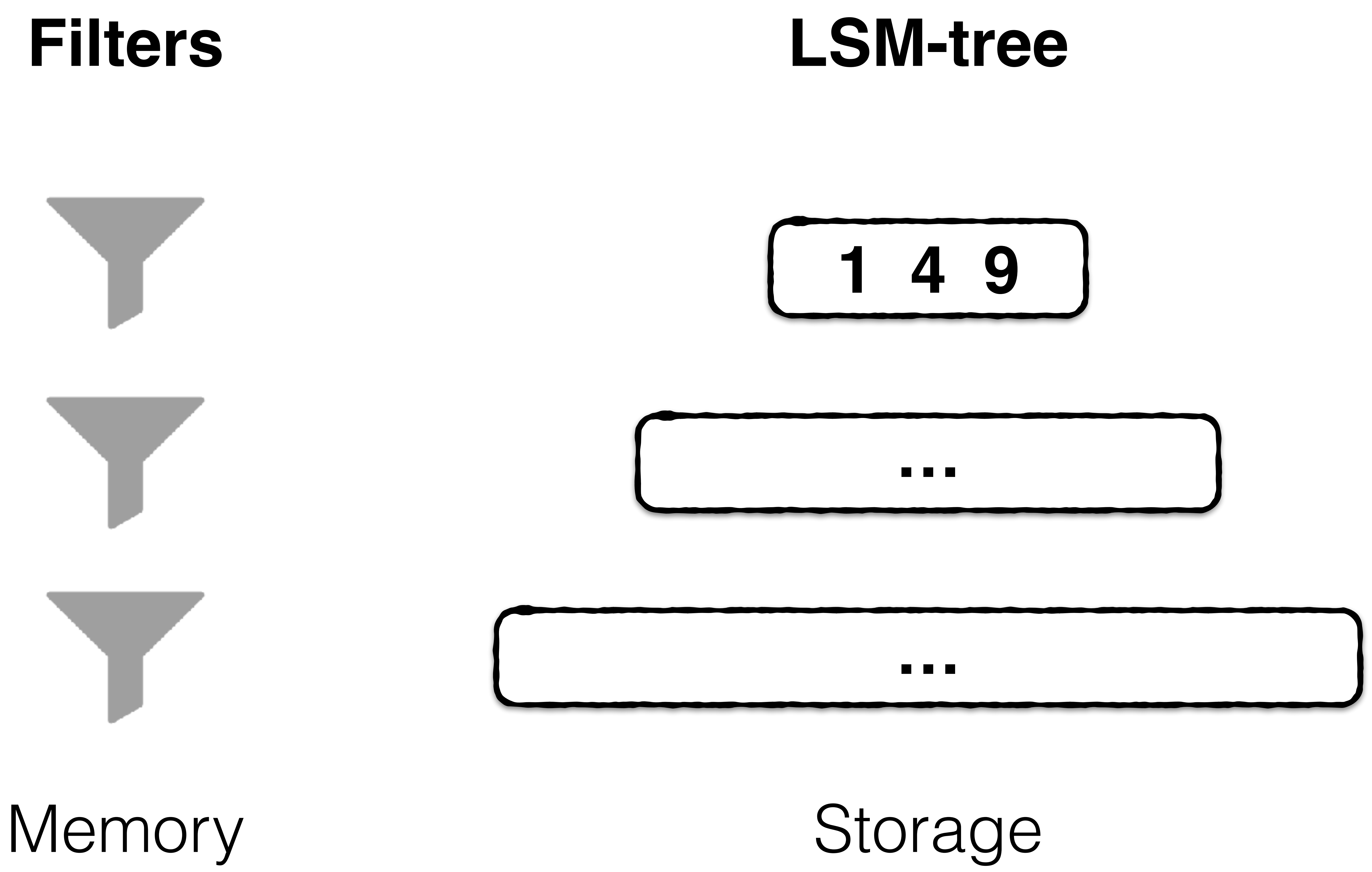
Set

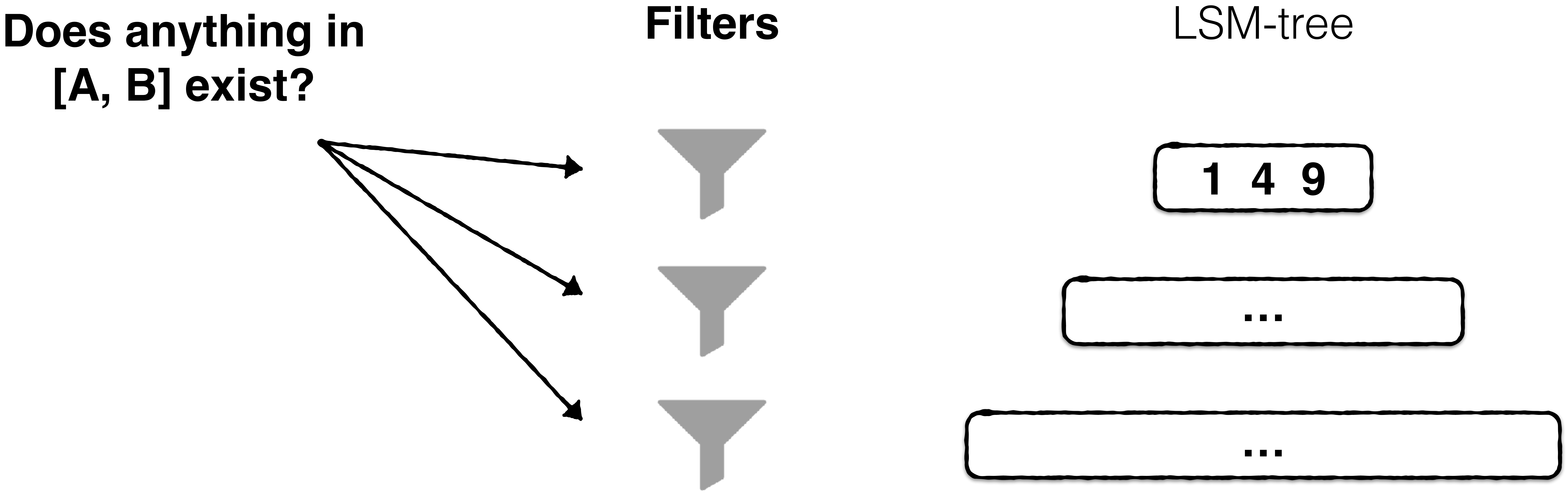
1 4 9

True negative
with prob 1-ε

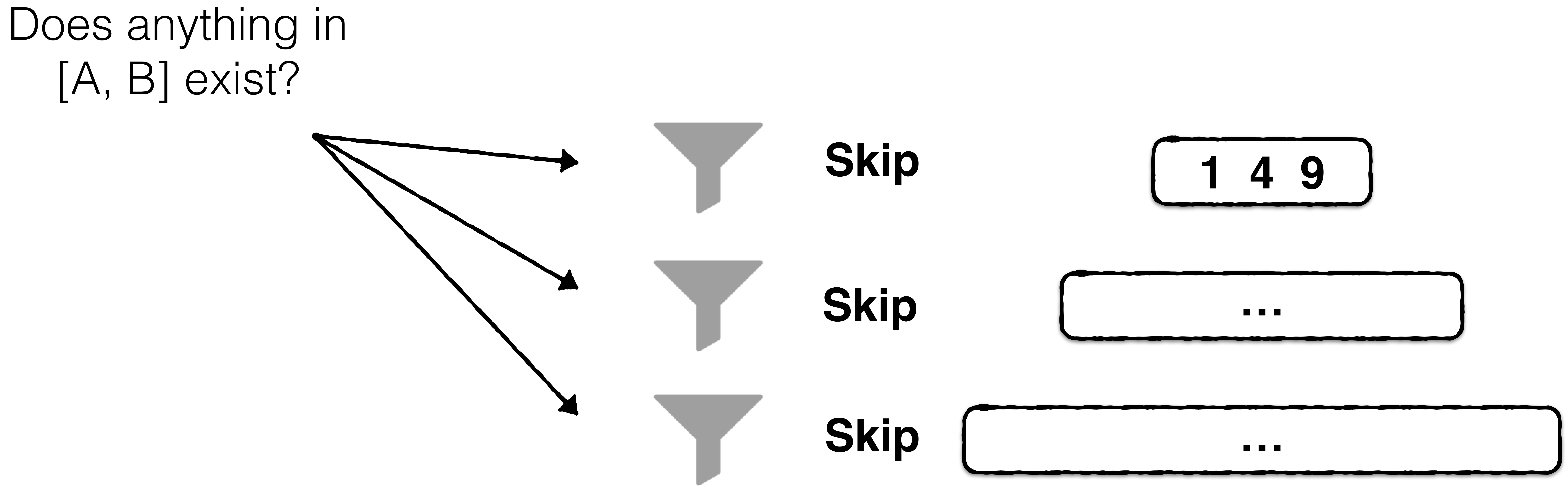
False positive
with prob ε

Does anything in
[5, 8] exist?



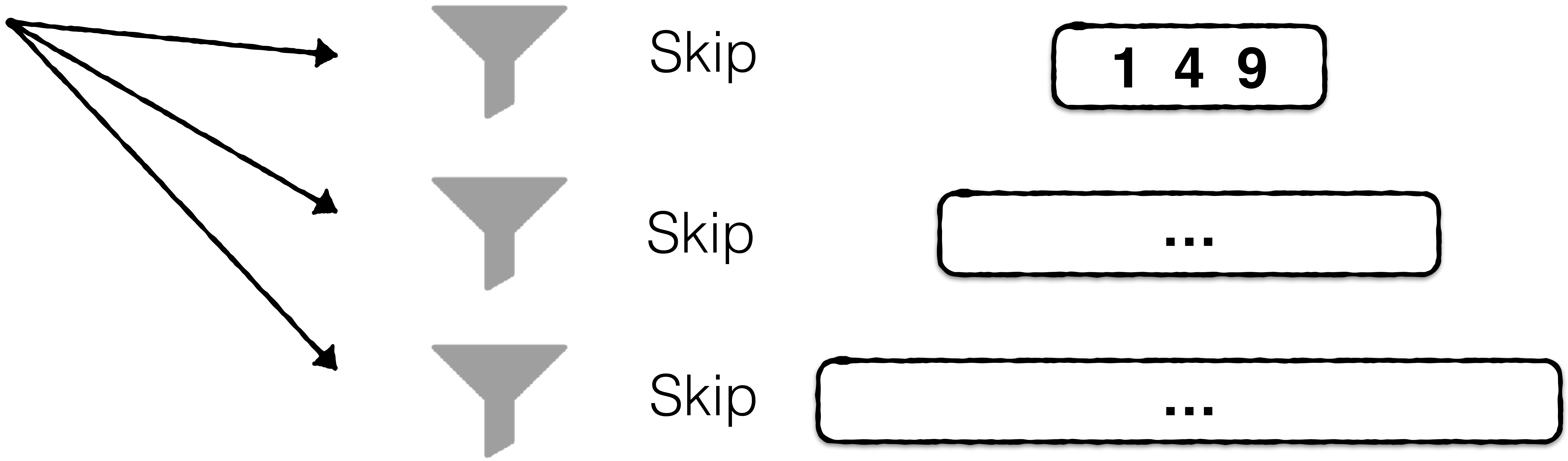

Does anything in
[A, B] exist?
Filters
LSM-tree
1 4 9
...
...

Ideally skip runs that don't contain relevant key range



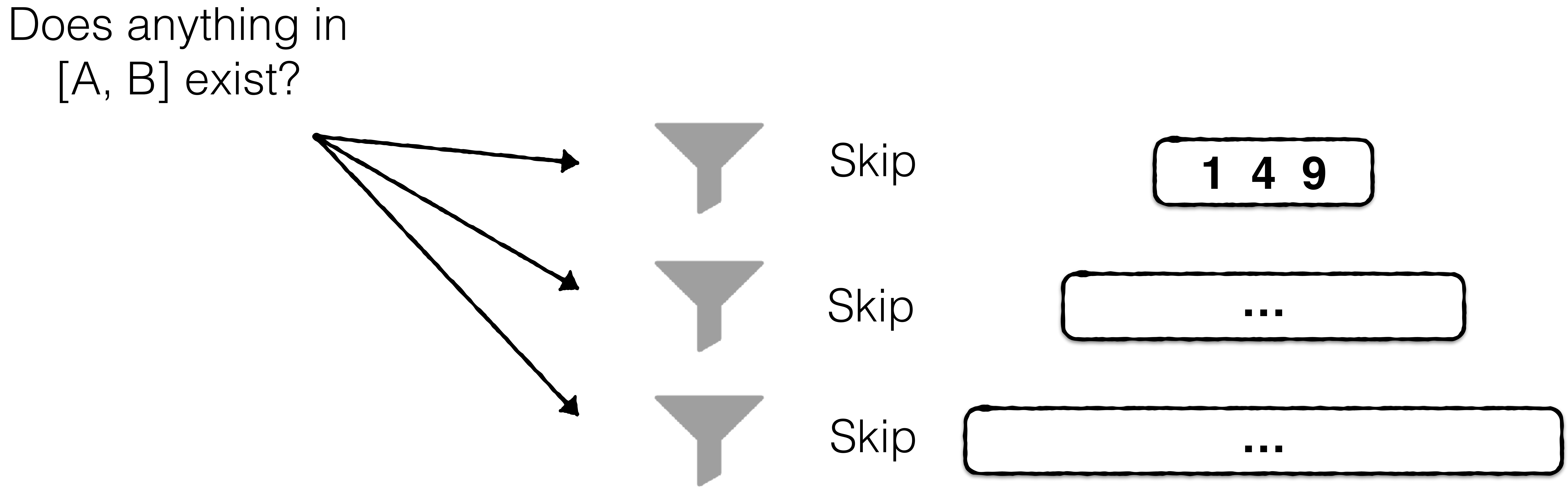
How to implement a range filter?

Does anything in
[A, B] exist?



How to implement a range filter?

Can we use a point filter (Bloom, Quotient) to answer range queries?



Does anything in
[5, 8] exist?

→

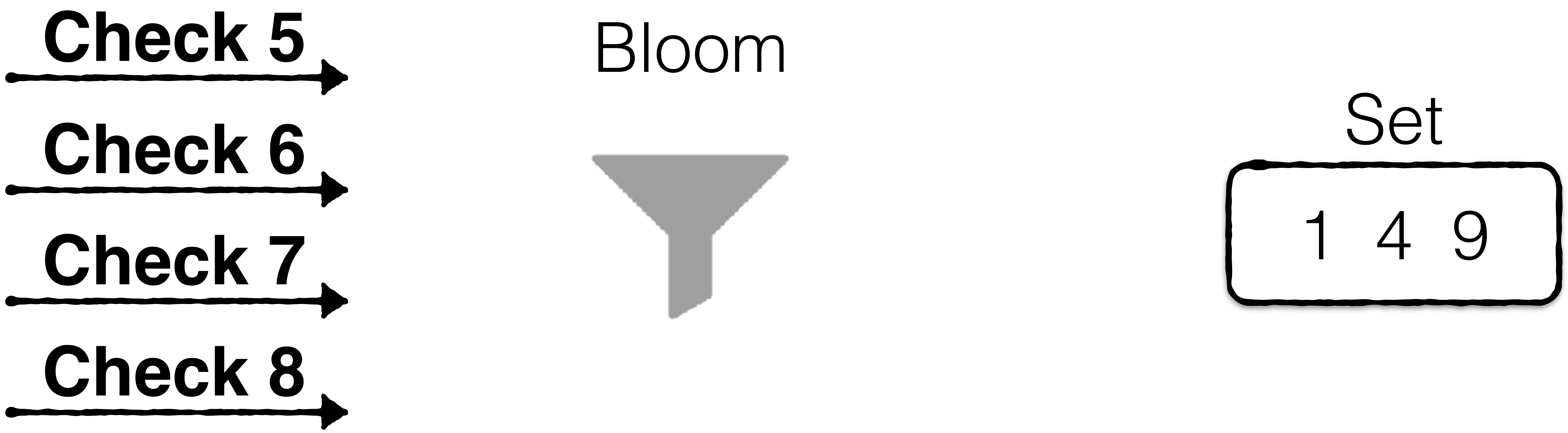
Bloom

Set

1 4 9

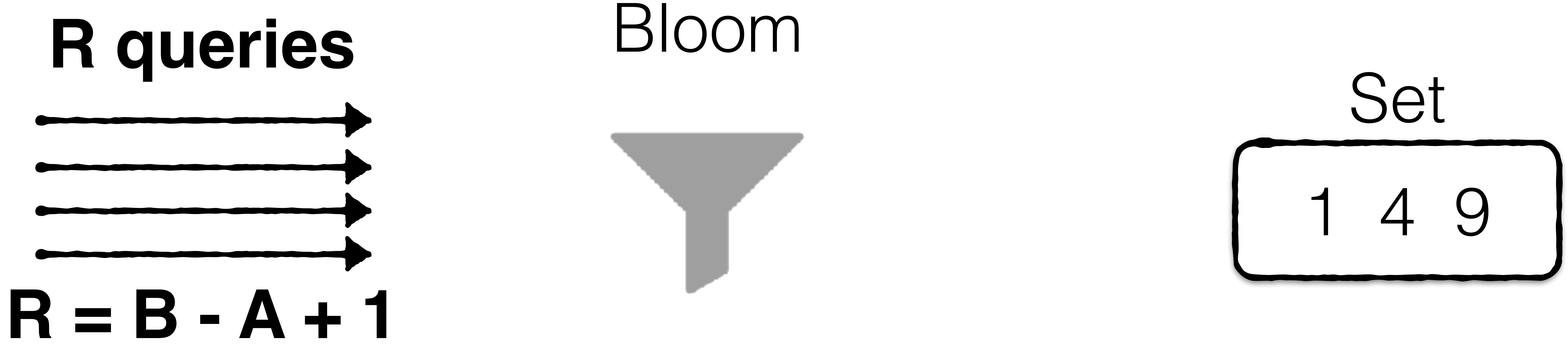
Query every key in range

Does anything in
[5, 8] exist?



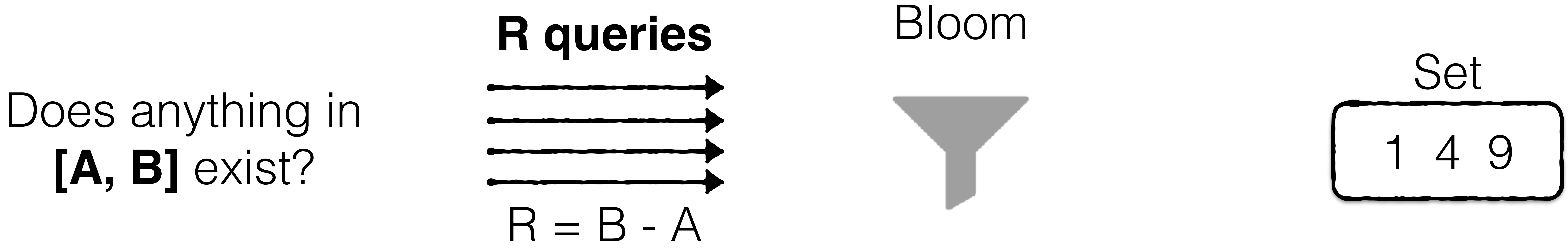
Query every key in range

Does anything in **[A, B]** exist?



If all queries return negative, we know key does not exist

Query every key in range



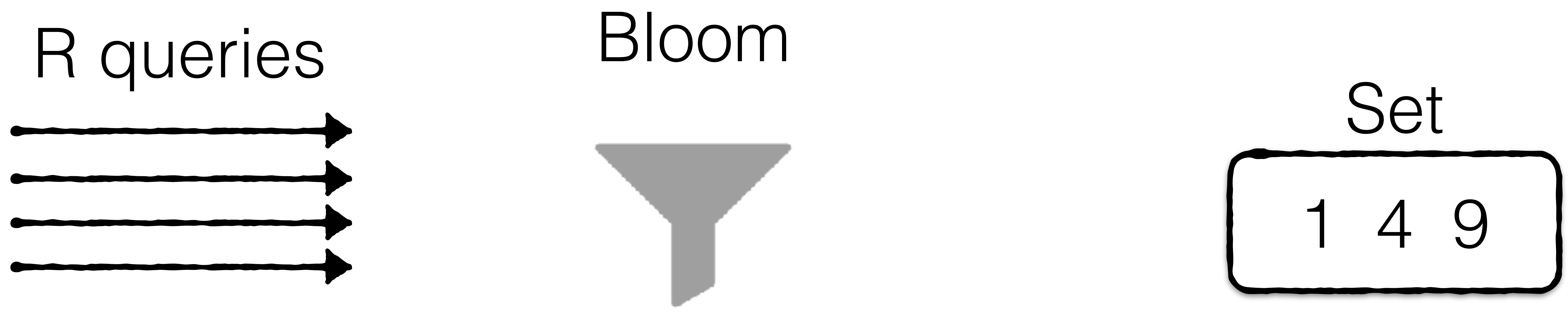
If all queries return negative, we know key does not exist

If at least one returns a positive, the overall outcome is a positive

Query every key in range

Problems?

Does anything in
[A, B] exist?

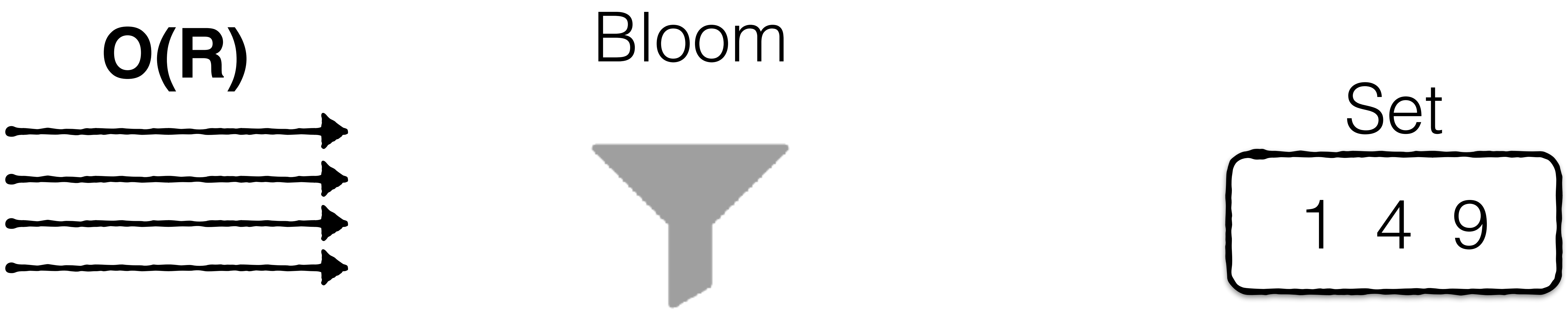


If all queries return negative, we know key does not exist

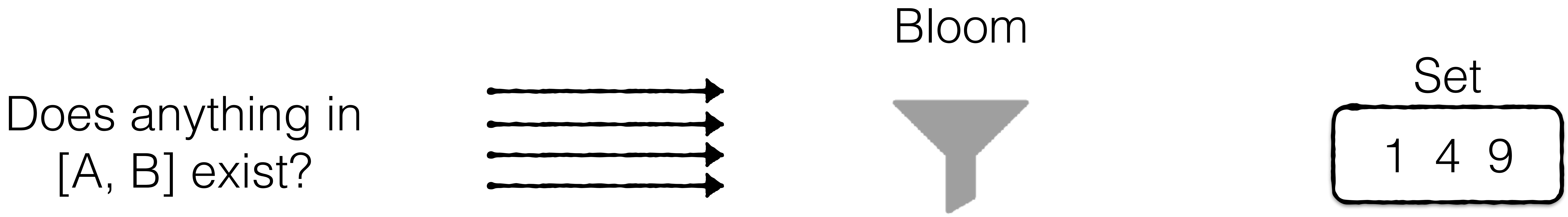
If at least one returns a positive, the overall outcome is a positive

Problem 1: Query Cost

Does anything in
[A, B] exist?

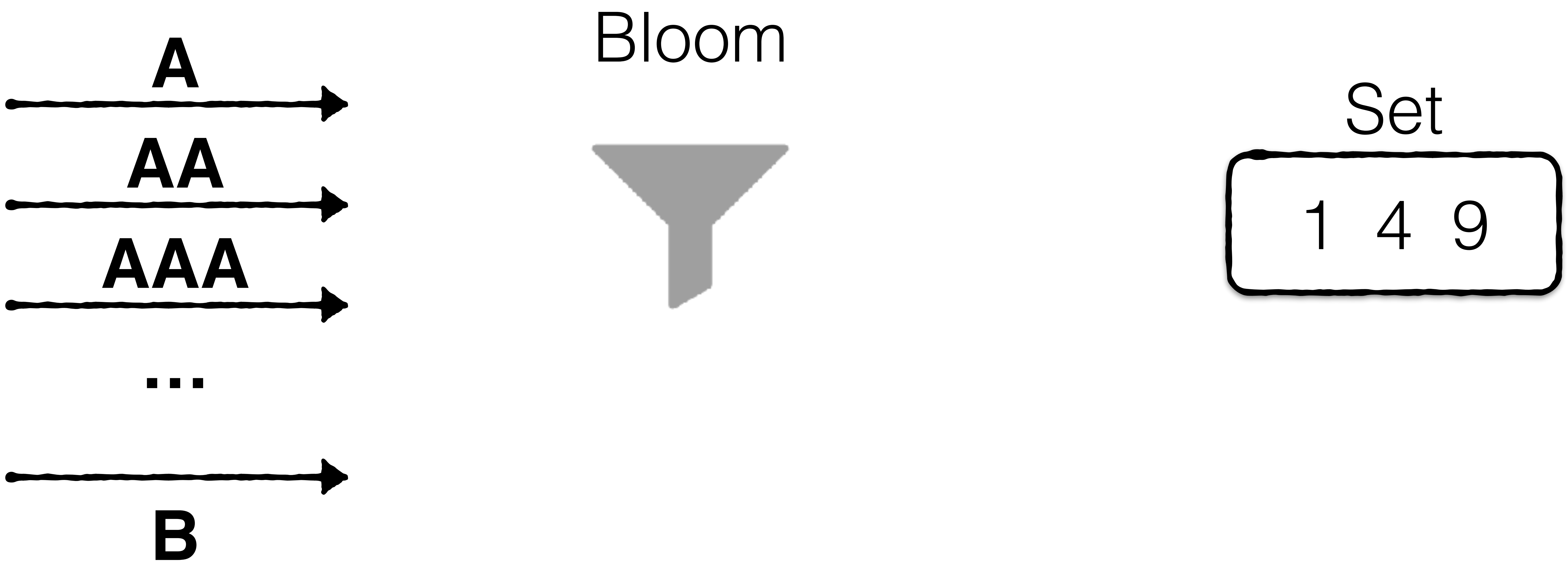


Problem 2: Does not work for infinite universe (e.g., strings)

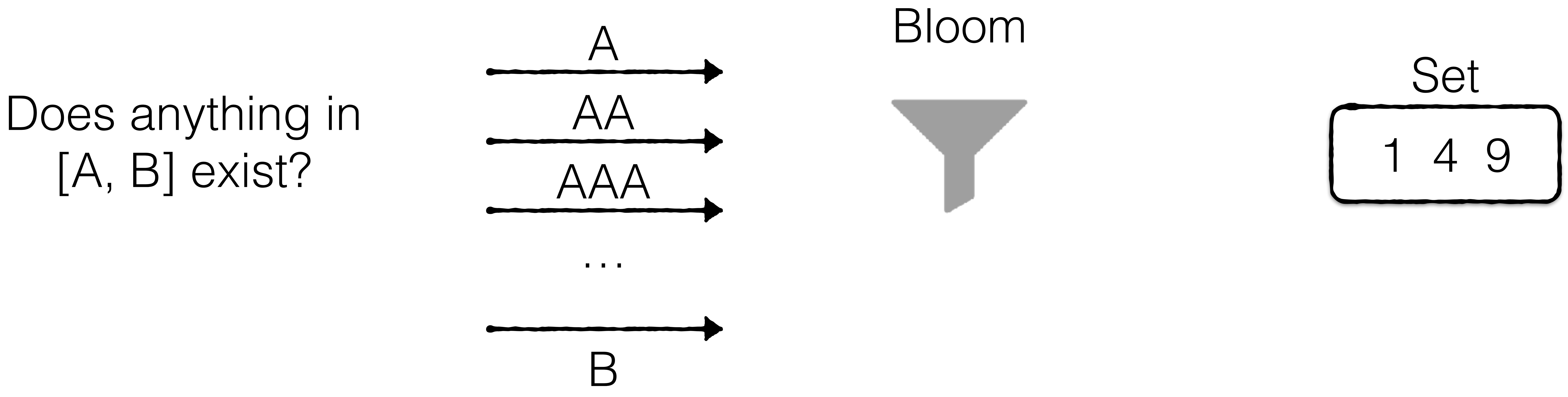


Problem 2: Does not work for infinite universe (e.g., strings)

Does anything in [A, B] exist?



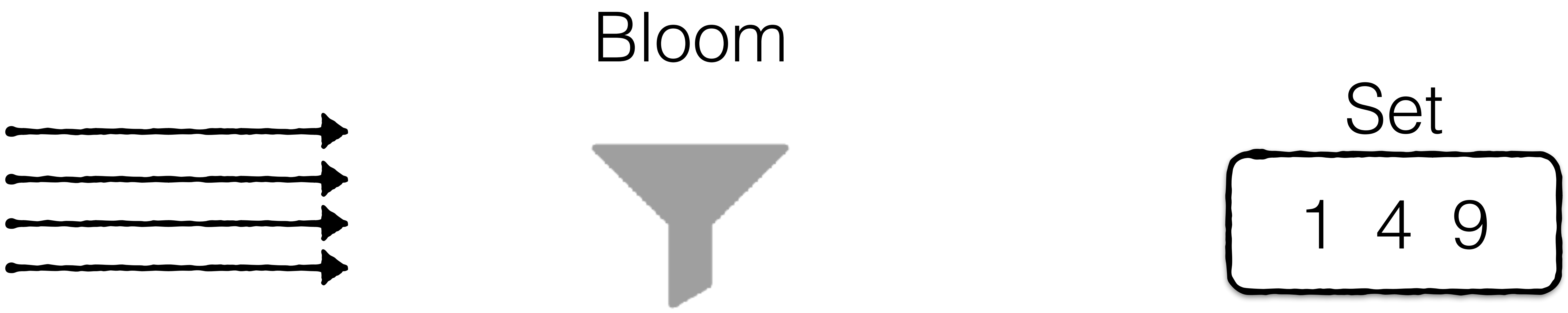
Problem 2: Does not work for infinite universe (e.g., strings)



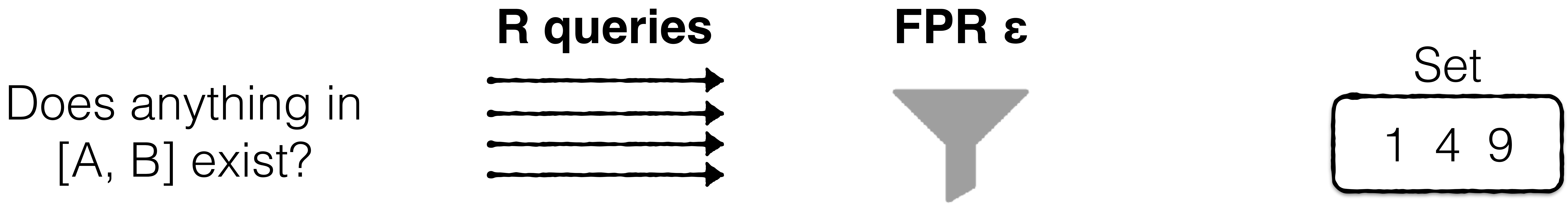
There are infinitely many possible keys in a range :)

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?



Problem 3: Higher False Positive Rate



Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

P[false positive]?

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

P[false positive] = P[at least one query returns positive]

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

P[false positive] = P[at least one query returns positive]

= 1 - P[all queries return negative]

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

P[false positive] = P[at least one query returns positive]

= 1 - P[all queries return negative]

= 1 - P[one query return negative]^R

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

$$
\begin{aligned}
P[\text{false positive}] \quad &= \quad P[\text{at least one query returns positive}] \\
&= \quad 1 - P[\text{all queries return negative}] \\
&= \quad 1 - P[\text{one query return negative}]^R \\
&\mathbf{=} \quad \mathbf{1 - (1-\varepsilon)^R}
\end{aligned}
$$

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

$$
\begin{aligned}
P[\text{false positive}] &= P[\text{at least one query returns positive}] \\
&= 1 - P[\text{all queries return negative}] \\
&= 1 - P[\text{one query return negative}]^R \\
&= 1 - (1-\varepsilon)^R \\
&\approx \mathbf{1 - e^{\varepsilon \cdot R}}
\end{aligned}
$$

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

$$
\begin{aligned}
P[\text{false positive}] &= P[\text{at least one query returns positive}] \\
&= 1 - P[\text{all queries return negative}] \\
&= 1 - P[\text{one query return negative}]^{R} \\
&= 1 - (1-\varepsilon)^{R} \\
&\approx 1 - e^{\varepsilon \cdot R} \\
&\boldsymbol{< \varepsilon \cdot R} \qquad \textbf{By union bound}
\end{aligned}
$$

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

P[false positive] < ε · R

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

P[false positive] $<$ $\varepsilon \cdot R$

Useless for large R

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

R queries

FPR ε

Set

P[false positive] $< \boldsymbol{\varepsilon} \cdot R$

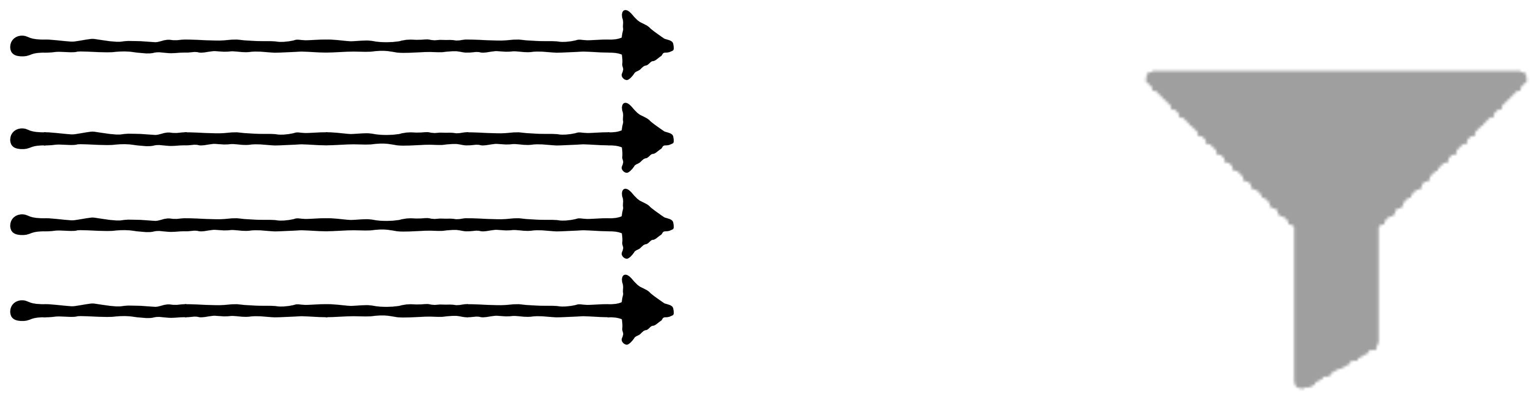
Recall that: $\boldsymbol{\varepsilon = 2^{-F}}$ **where F = bits / entry**

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

R queries

FPR ε



Set

1 4 9

P[false positive] $<$ $\mathbf{2^{-F}} \cdot R$

How many extra bits per entry do we need to make up for R?

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

$$P[\text{false positive}] \quad < \quad \mathbf{2^{-F}} \cdot R$$

How many extra bits per entry do we need to make up for R?

$\mathbf{\log_2(R)}$

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

$$P[\text{false positive}] \quad < \quad \mathbf{2^{-F\ -\ \log2(R)}} \cdot R$$

How many extra bits per entry do we need to make up for R?

Problem 3: Higher False Positive Rate

Does anything in
[A, B] exist?

$$P[\text{false positive}] \quad < \quad 2^{-F - \cancel{\mathbf{log2(R)}}} \cdot \cancel{\mathbf{R}}$$

How many extra bits per entry do we need to make up for R?

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

$$P[\text{false positive}] \quad < \quad 2^{-F - \cancel{\log2(R)}} \cdot \cancel{R}$$

FPR is higher by factor of R, or we need $\log_2(R)$ extra bits / entry to keep it stable

Problem 3: Higher False Positive Rate

Does anything in [A, B] exist?

$$P[\text{false positive}] \quad < \quad 2^{-F - \cancel{\log 2(R)}} \cdot \cancel{R}$$

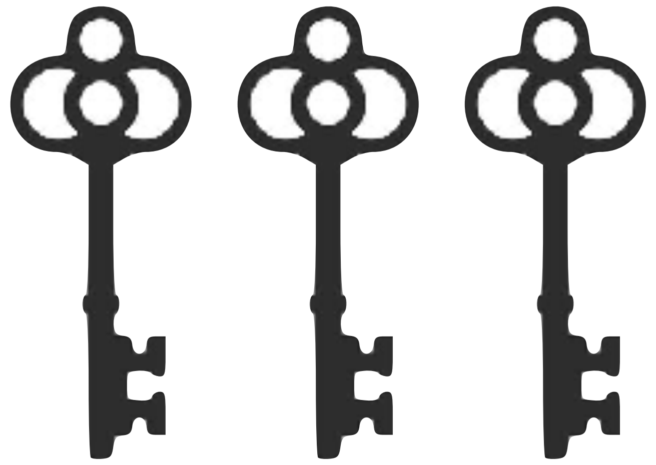
FPR is higher by factor of R, or we need $\log_2(R)$ extra bits / entry to keep it stable

But this also requires us to know max range query length in advance

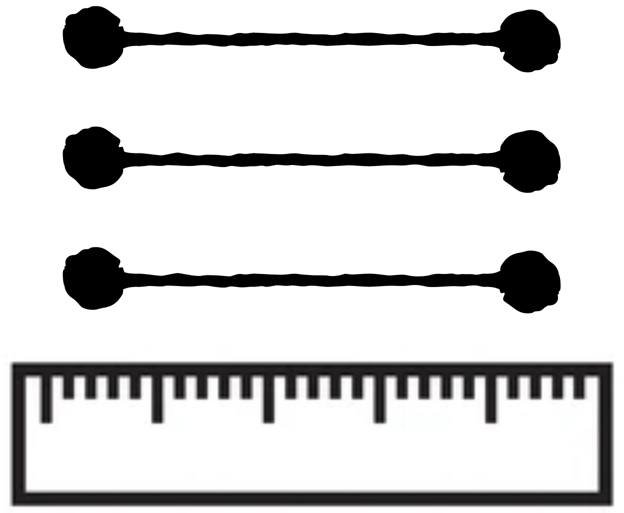
Problems

query cost
$O(R)$

FPR
$O(\varepsilon \cdot R)$



Fixed-length
keys



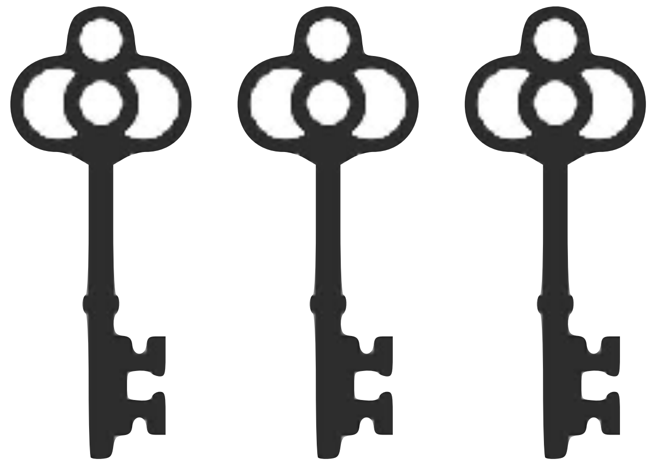
Bounded
queries

Problems

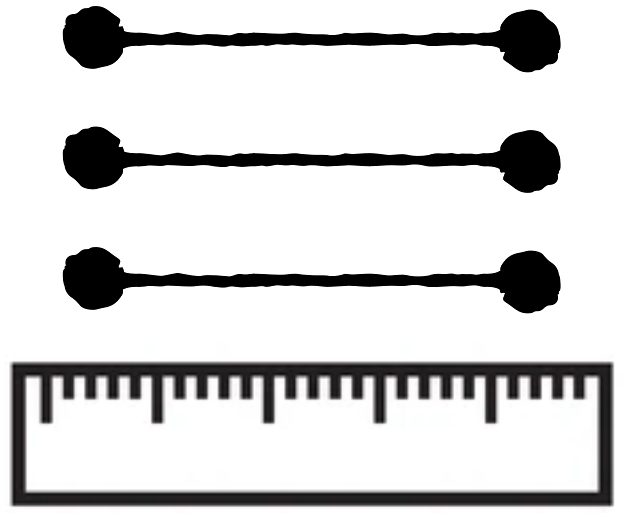
How much can we improve these?

query cost
O(R)

FPR
O(ε · R)



Fixed-length
keys



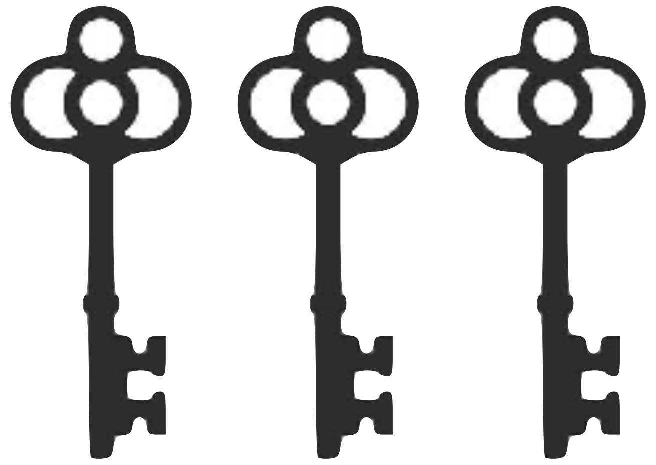
Bounded
queries

How much can we improve these?

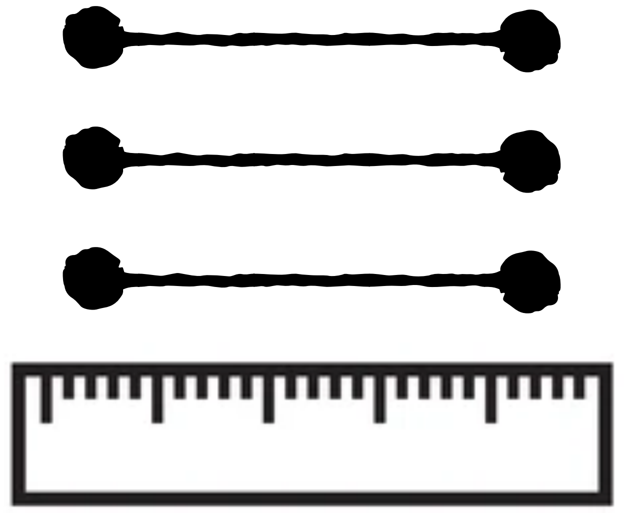
query cost
O(1)

?

FPR
O(ε · R)



Fixed-length
keys



Bounded
queries

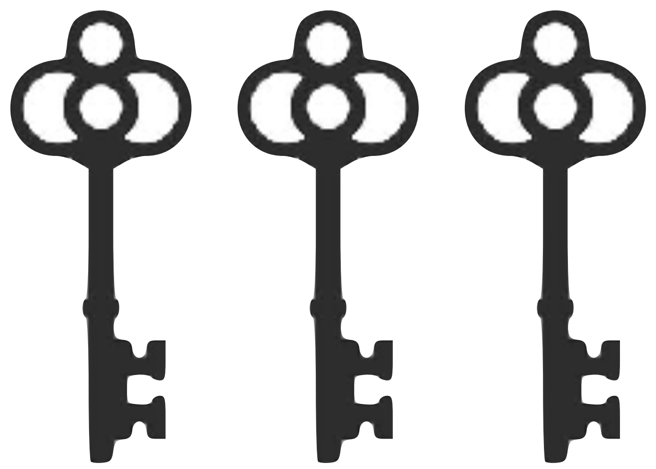
How much can we improve these?

query cost
O(1)

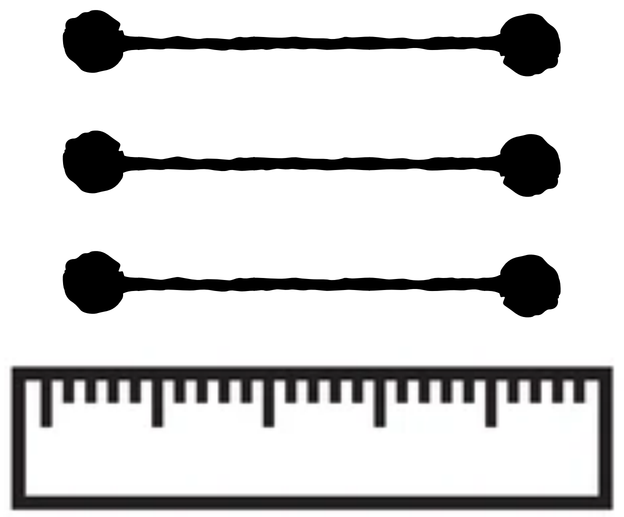
FPR
$\mathbf{O(\varepsilon \cdot \cancel{R})}$

↑

?



Fixed-length
keys

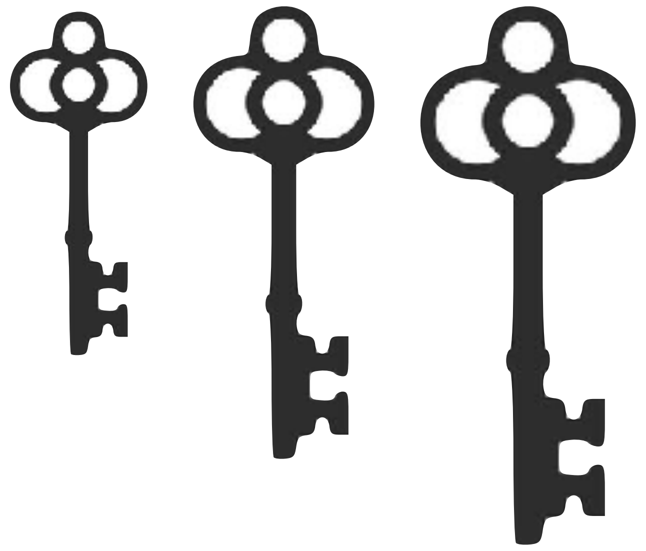


Bounded
queries

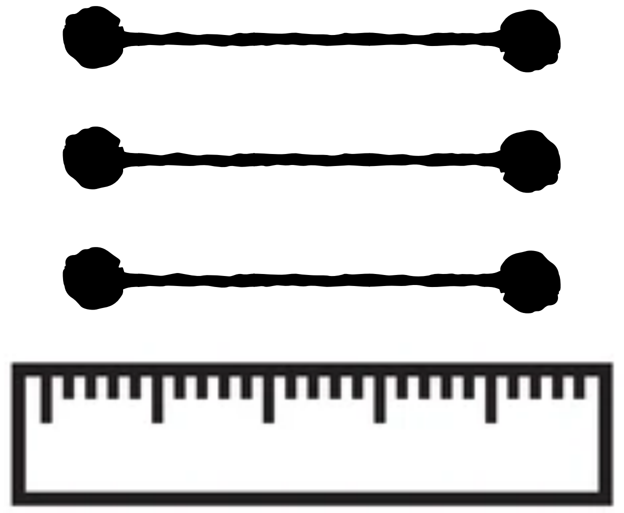
How much can we improve these?

query cost
O(1)

FPR
O(ε)



Var-length keys



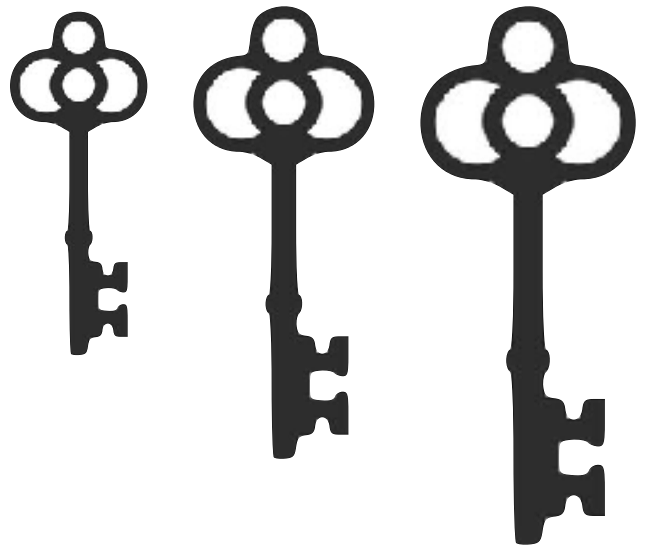
Bounded queries

?

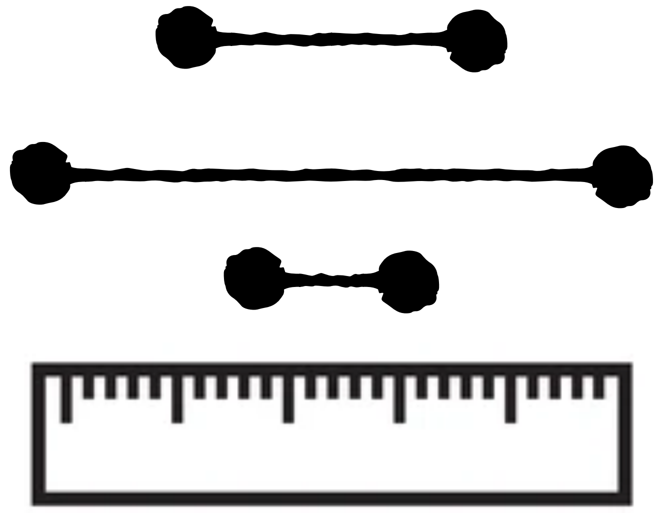
How much can we improve these?

query cost
O(1)

FPR
O(ε)



Var-length
keys

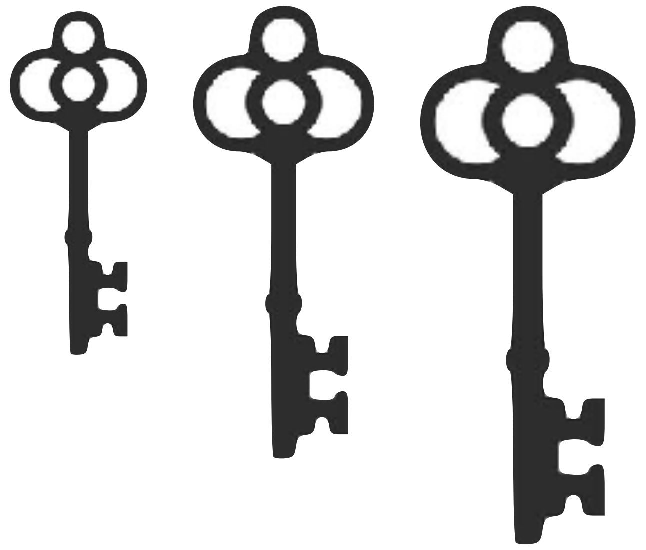


Var-length
queries

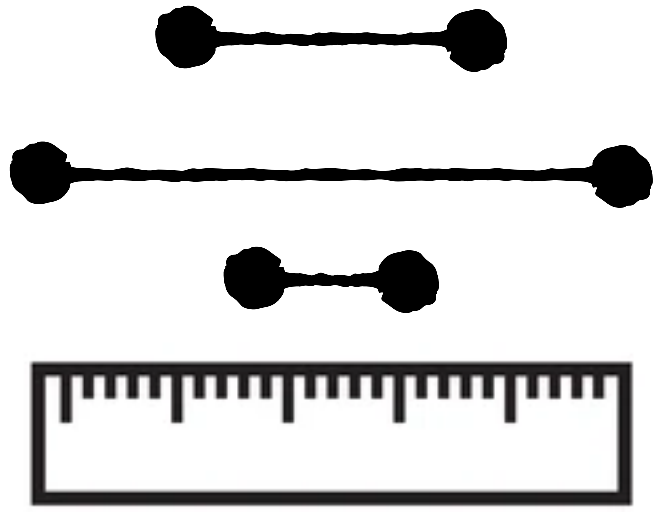
?

query cost
O(1)

FPR
O(ε)



Var-length
keys



Var-length
queries

+ Dynamic operations

(Inserts, deletes, expansions, contractions)

Hot research topic in past decade

GGLR SODA14	**SuRF** SIGMOD18	**Rosetta** SIGMOD2020	**SNARF** SIGMOD22	**Proteus** SIGMOD22

REncoder ICDE23	**BloomRF** EDBT23	**Grafite** SIGMOD24	**Oasis** VLDB24	**Memento** SIGMOD24	**Diva** …25

SuRF
SIGMOD18

Memento
SIGMOD24

Diva
…25

	SuRF	Memento	Diva
			
Var-length keys & queries			
FPR guarantee			
Query speed			
Dynamic			

	SuRF	Memento	Diva
			
Var-length keys & queries	Yes		
FPR guarantee	None		
Query speed	O(L) (L = key length)		
Dynamic	No		

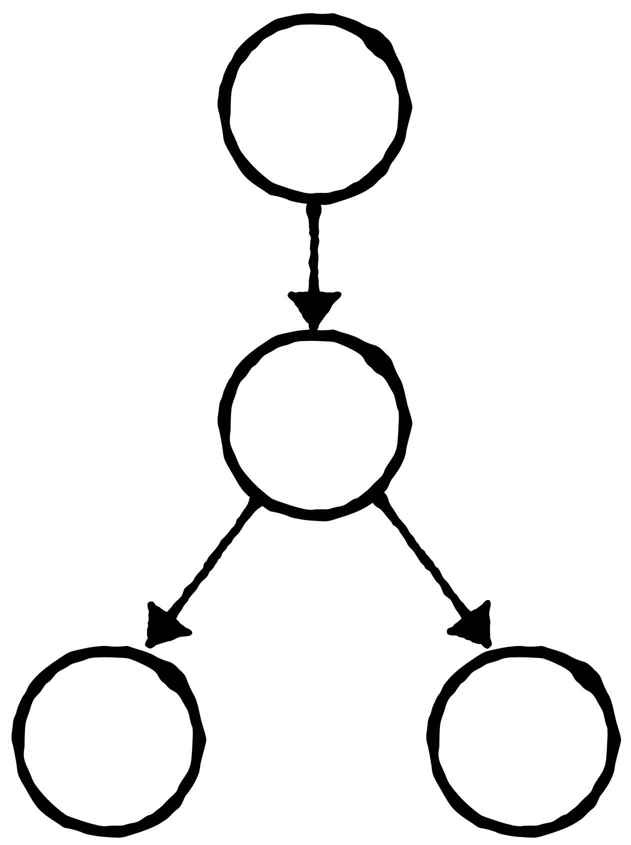
	SuRF	Memento	Diva
Var-length keys & queries	Yes	**No**	
FPR guarantee	None	**Robust**	
Query speed	O(L) (L = key length)	**O(1)**	
Dynamic	No	**Yes**	

	SuRF	Memento	Diva
			
Var-length keys & queries	Yes	No	**Yes**
FPR guarantee	None	Robust	**Semi-Robust**
Query speed	O(L) (L = key length)	O(1)	**O(log L)**
Dynamic	No	Yes	**Yes**

SuRF: Practical Range Query Filtering with Fast Succinct Tries

Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen,
Michael Kaminsky, Kimberly Keeton, Andrew Pavlo

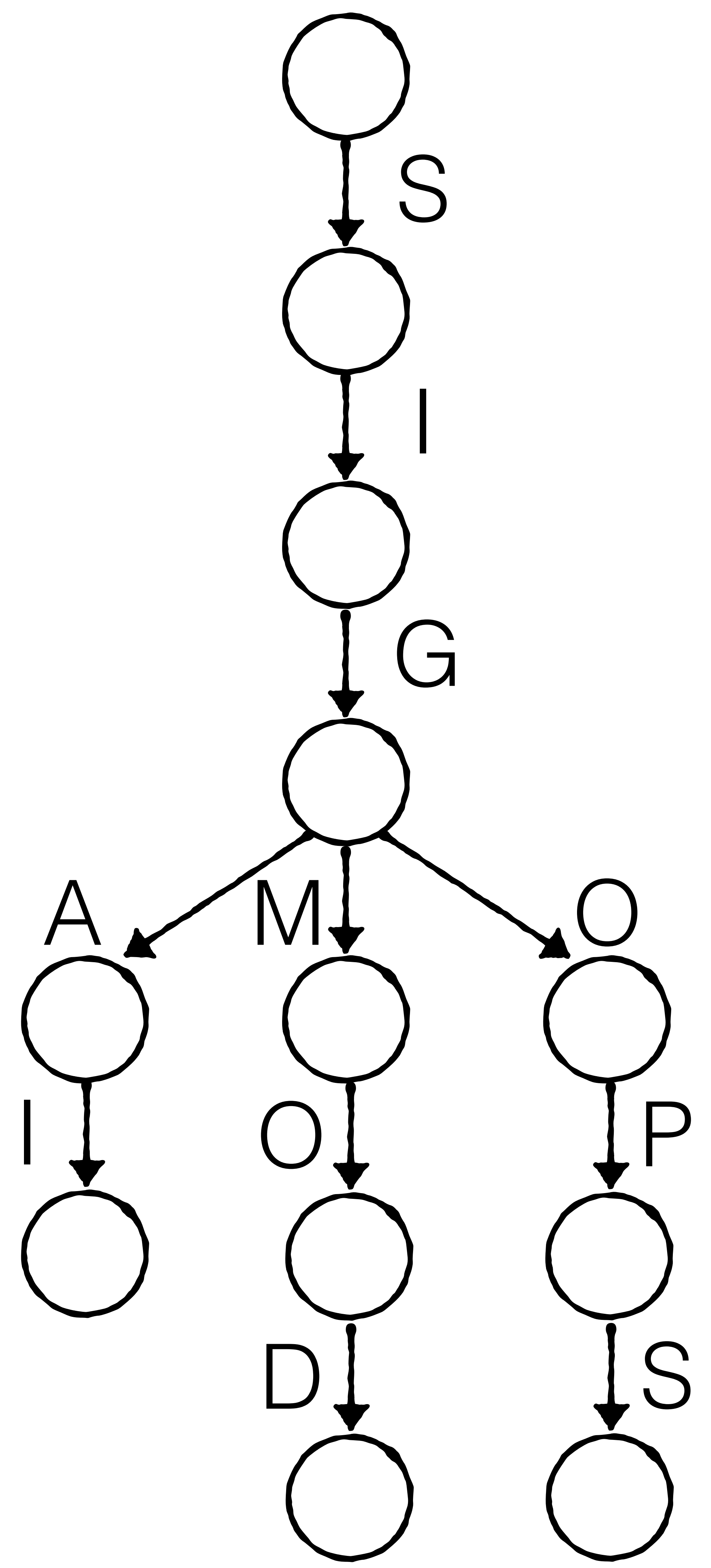
SIGMOD18



Starting Point

Keys

SIGAI
SIGMOD
SIGOPS



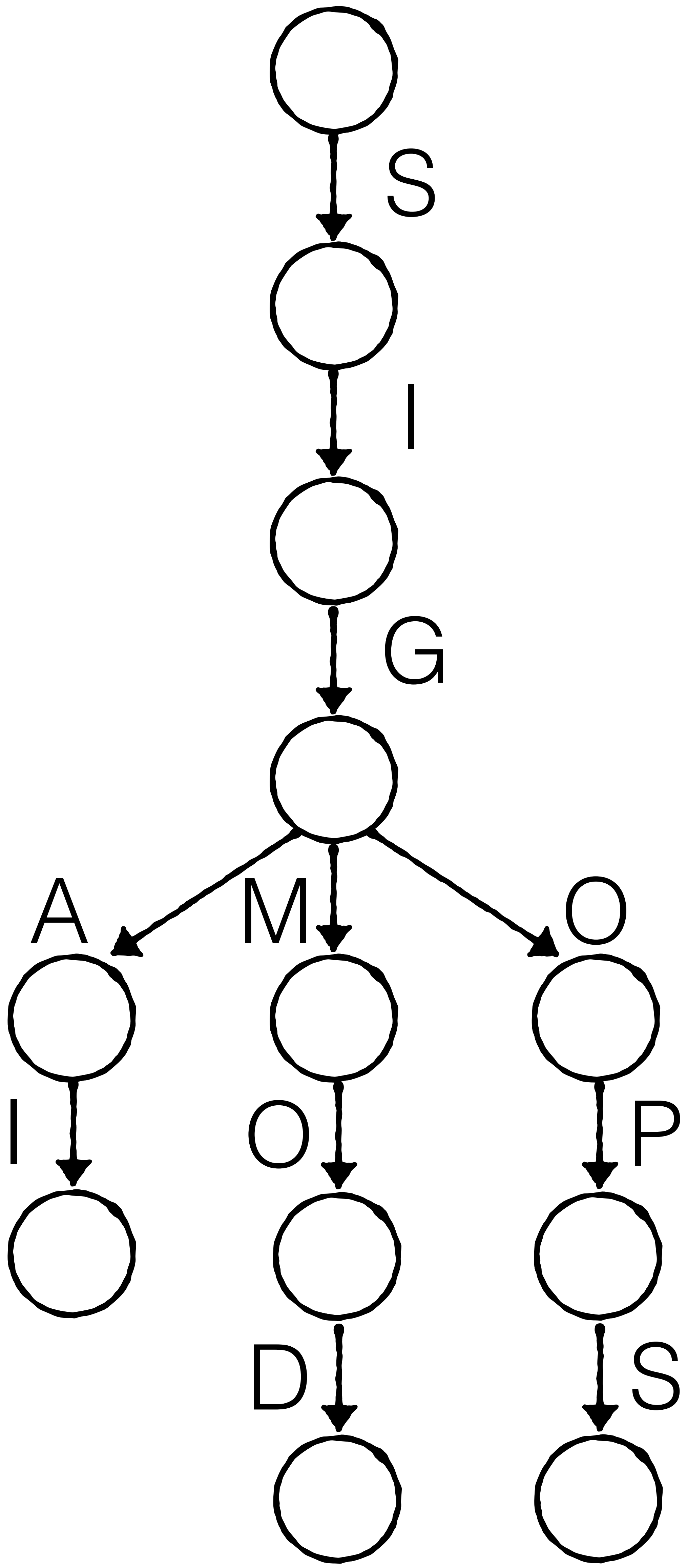
Starting Point

Keys

SIGAI

SIGMOD

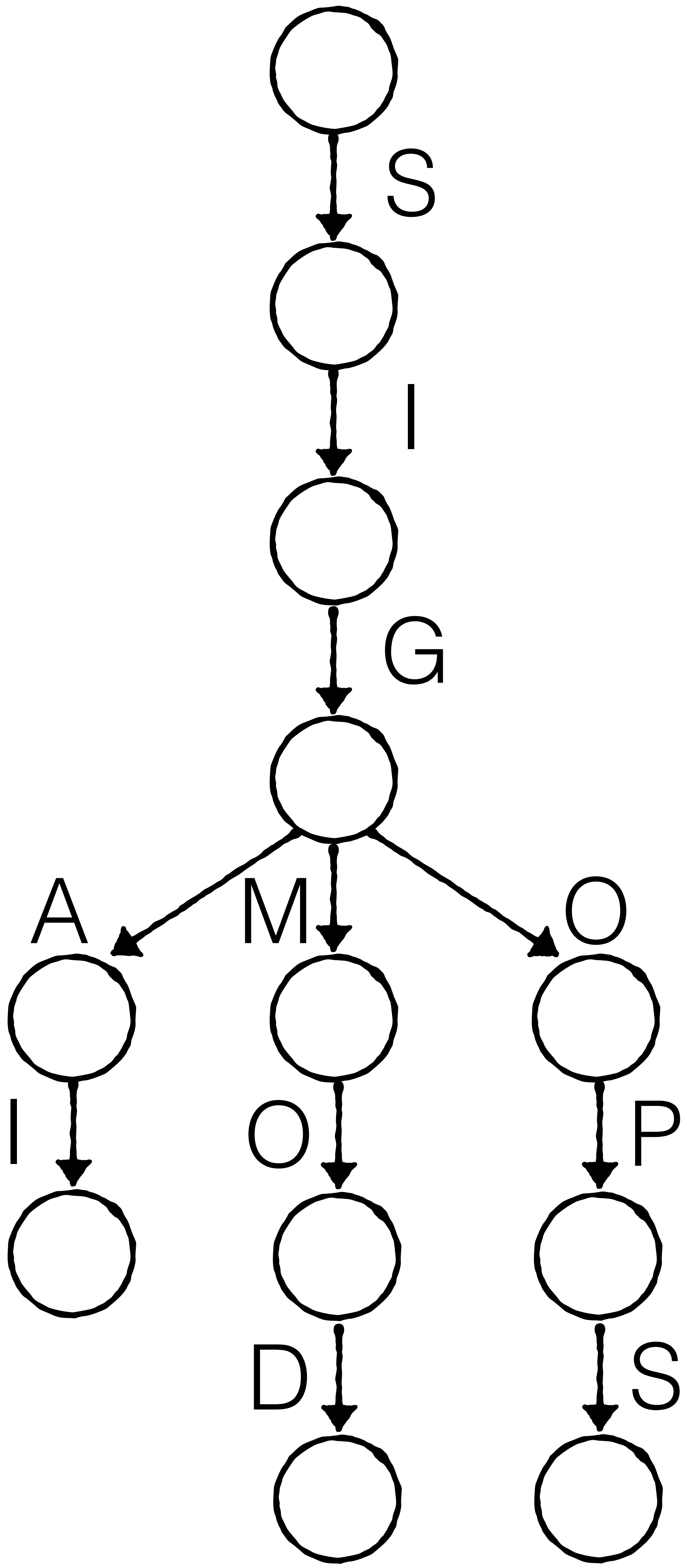
SIGOPS



Fanout - 256

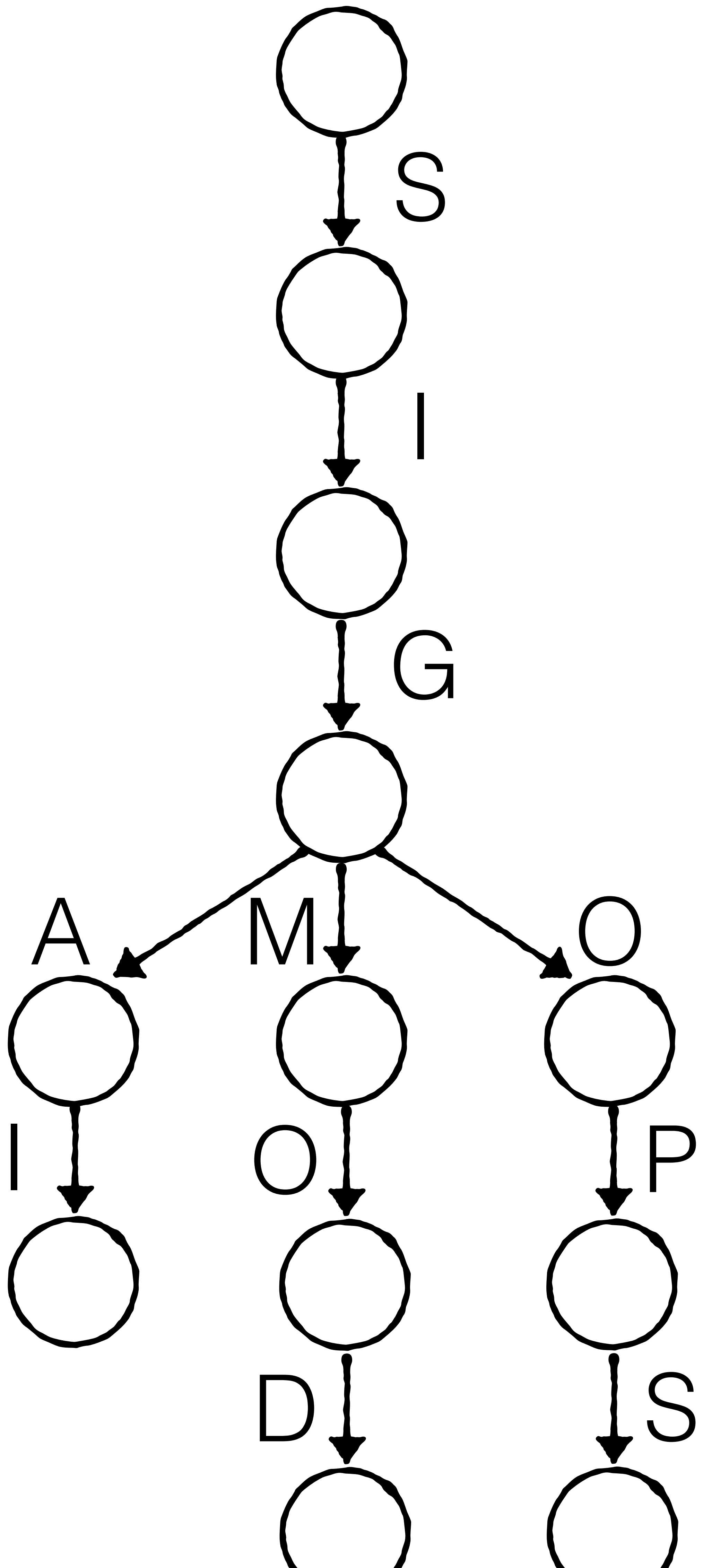
(1 byte)

Problems?



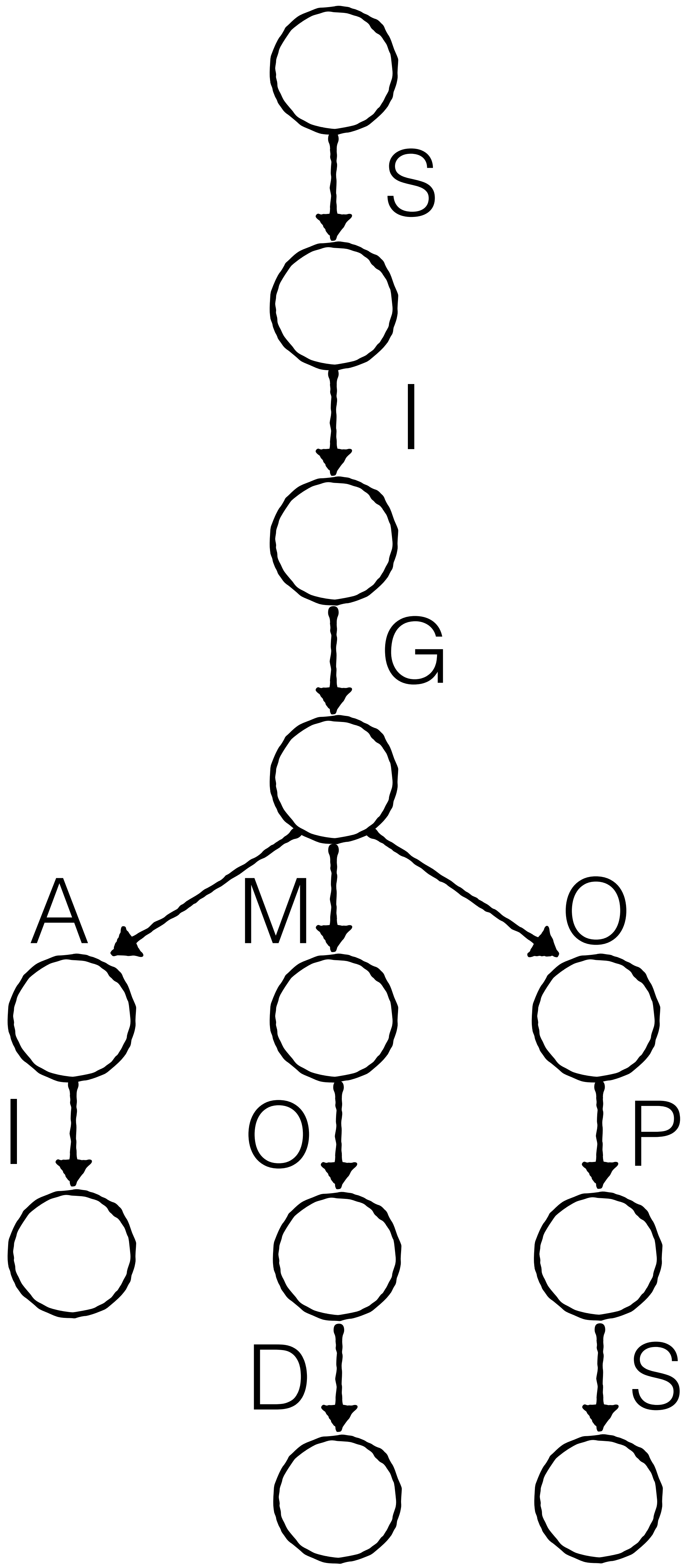
Problems?

1. **Stores full keys**



Problems?

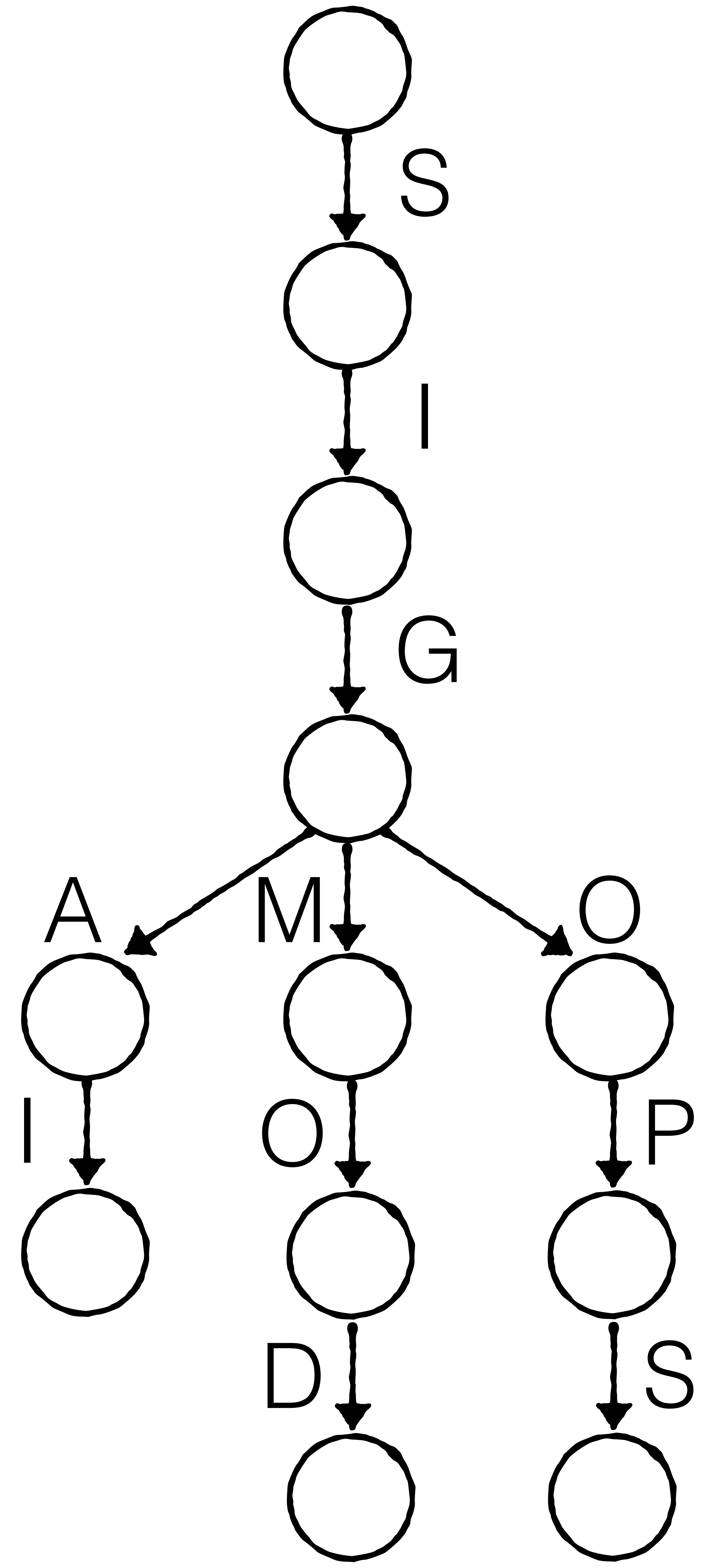
1. Stores full keys



2. Pointers take space

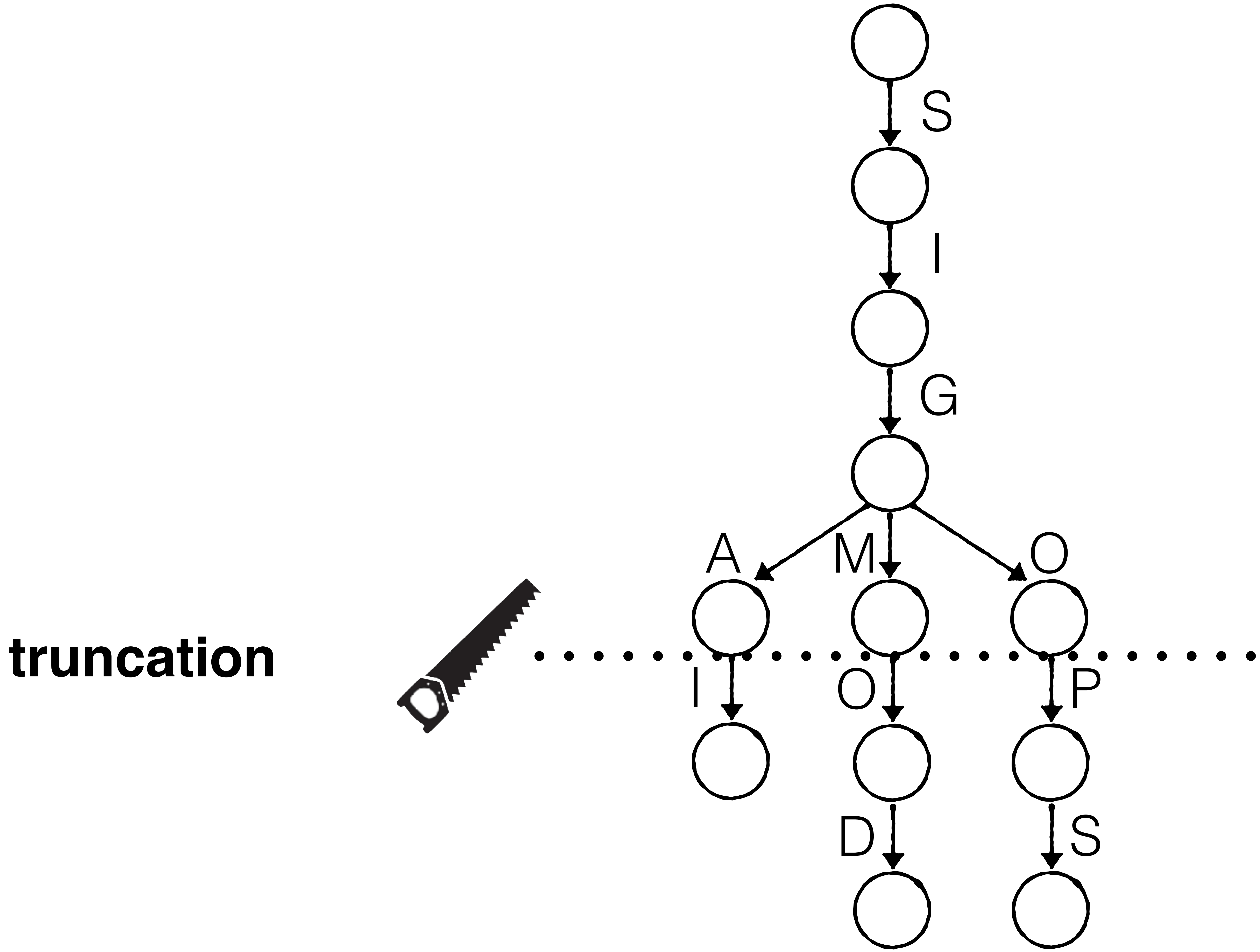
1. Stores full keys

truncation

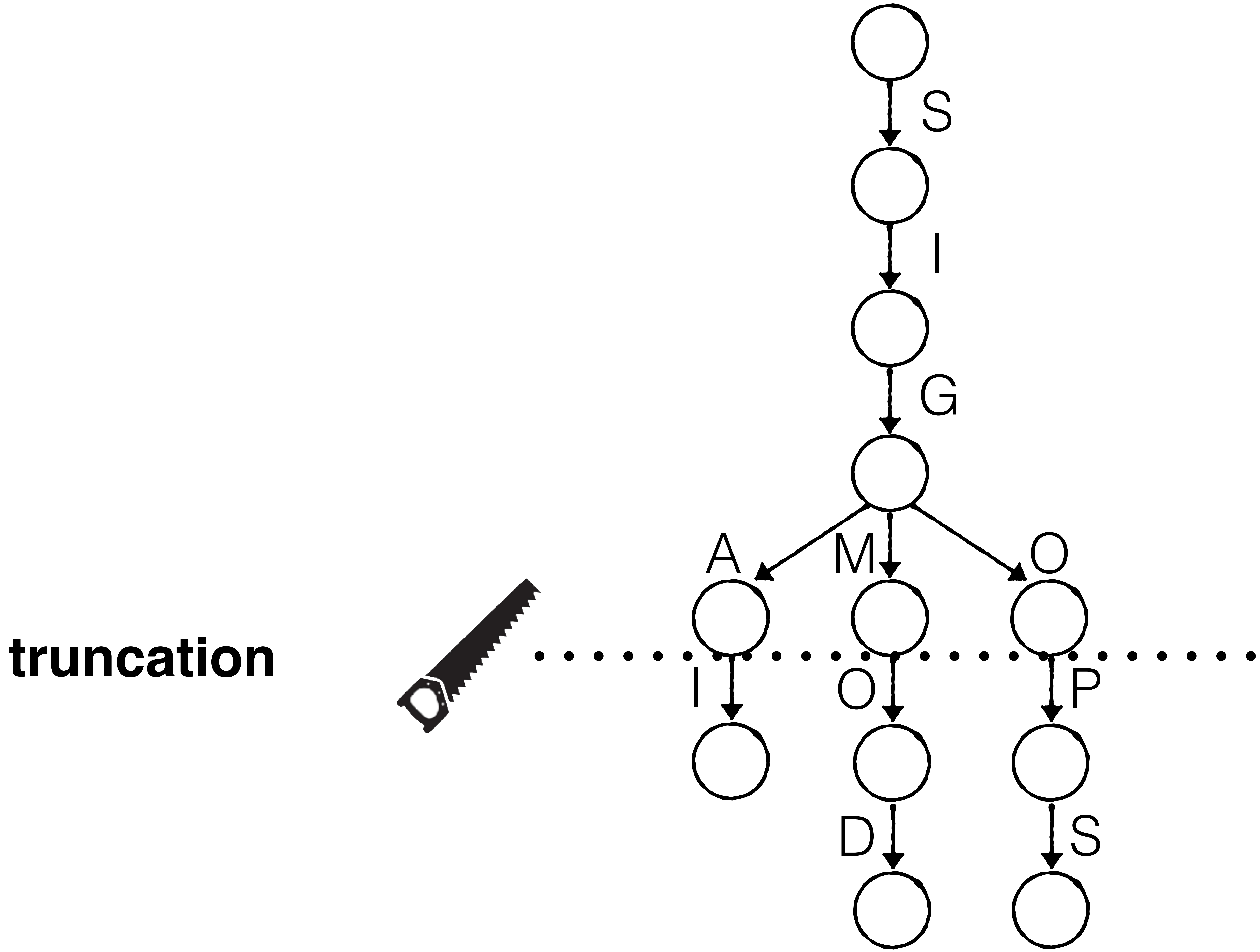


2. Pointers take space

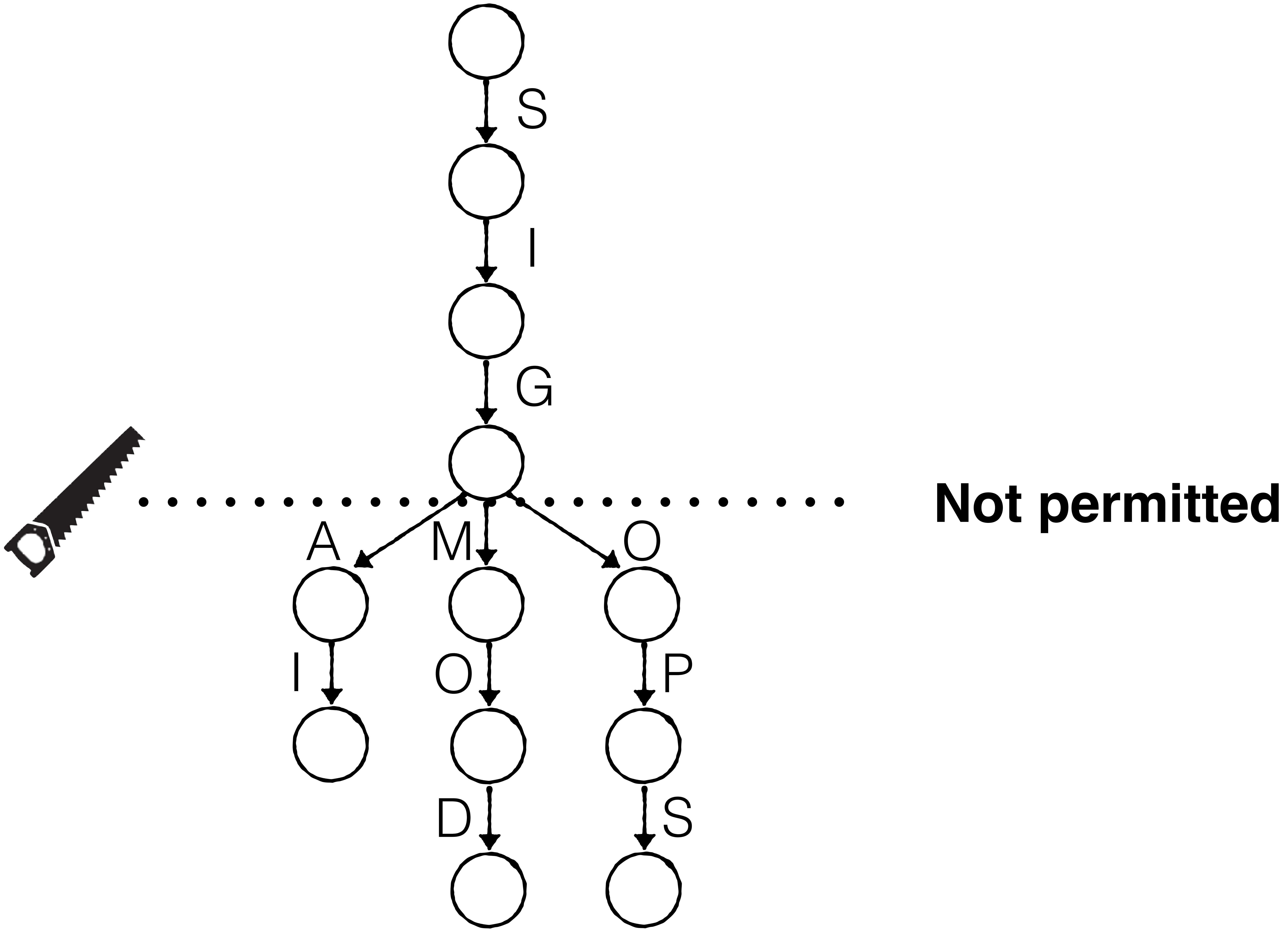
Succinct encoding


truncation
S
I
G
A
M
O
I
O
P
D
S

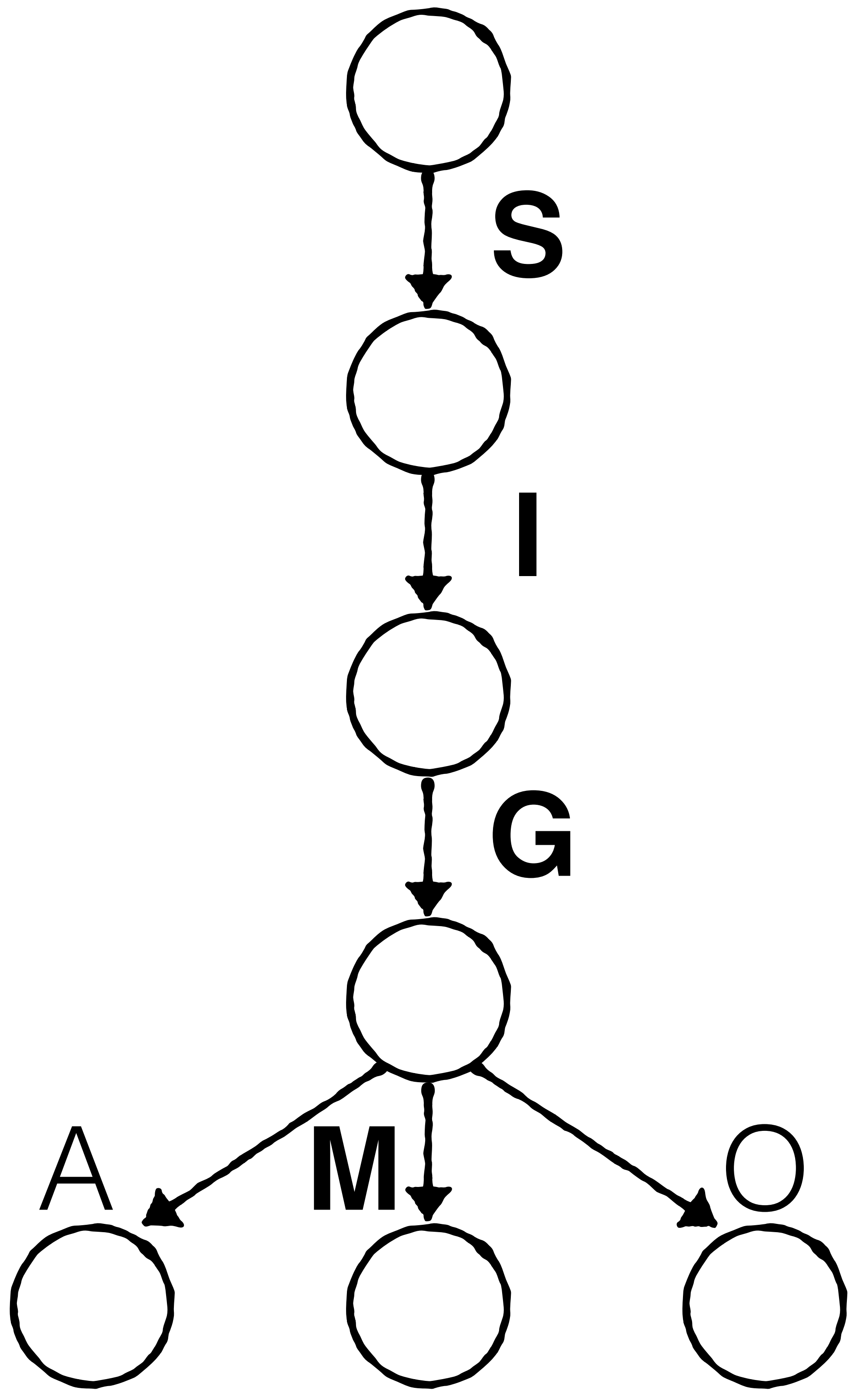
Keep at least one unique byte for each key



Keep at least one unique byte for each key

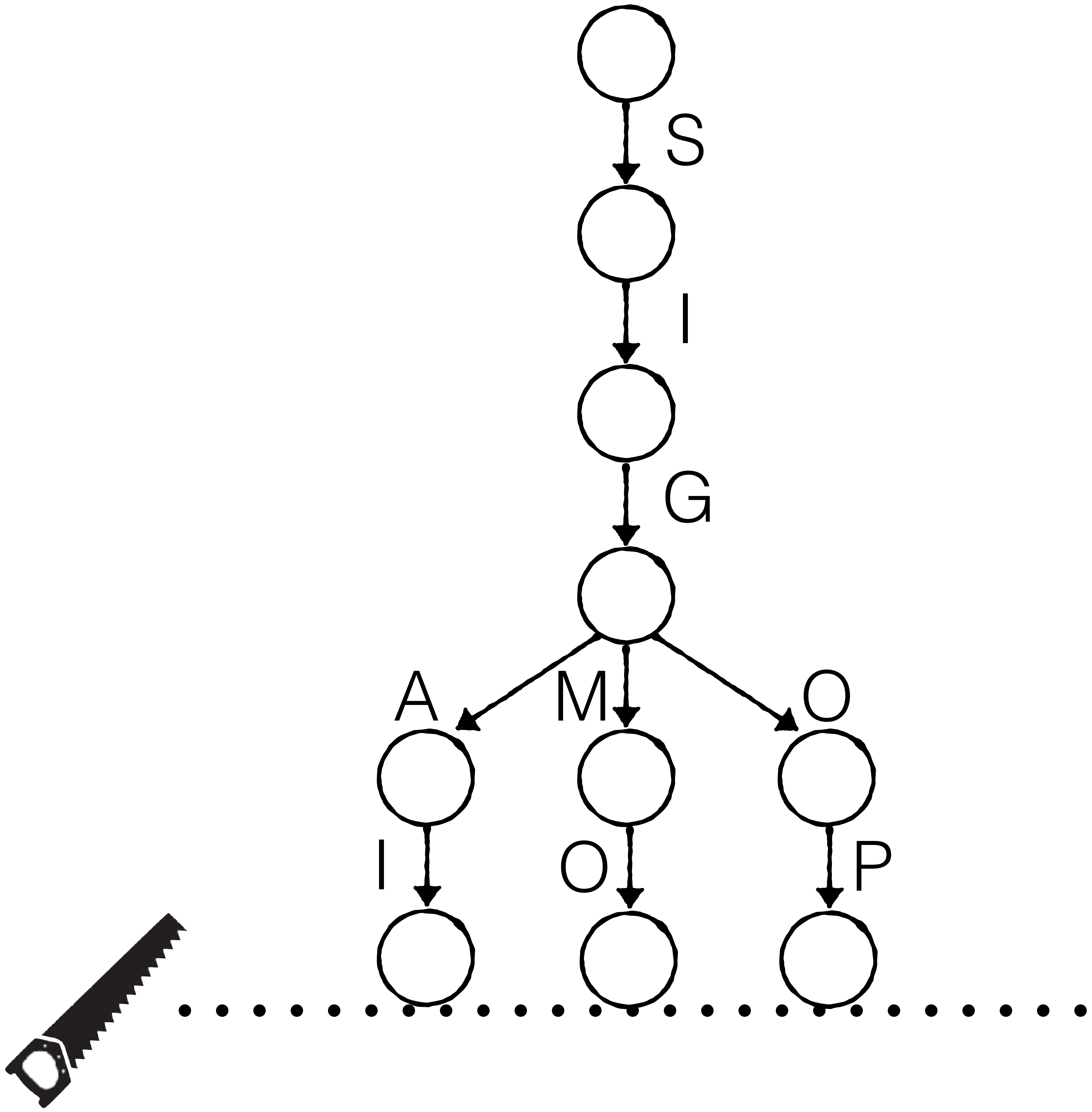


We now get false positives (e.g., query SIGMETRICS)



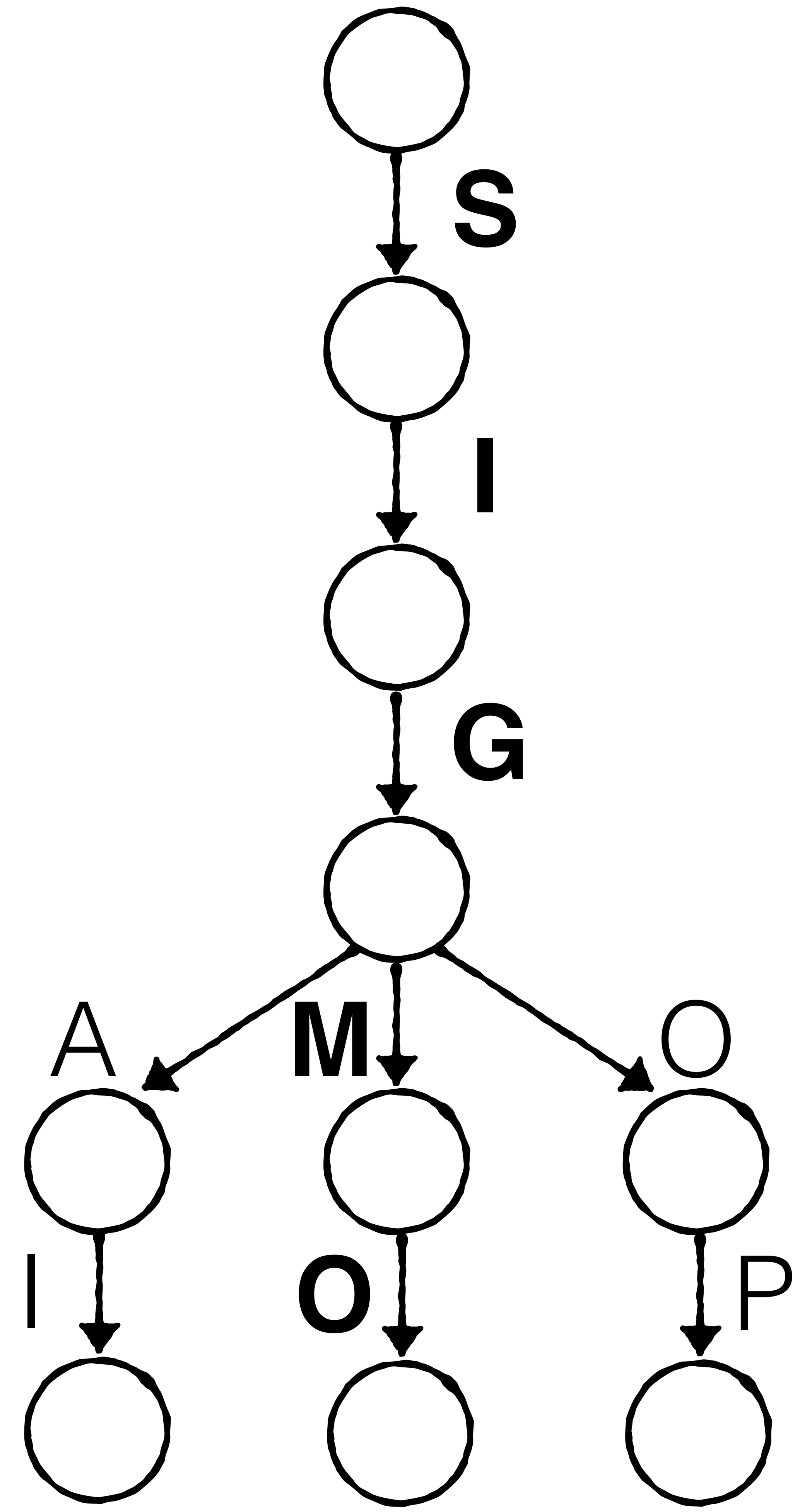
Match

We now get false positives (e.g., query SIGMETRICS)



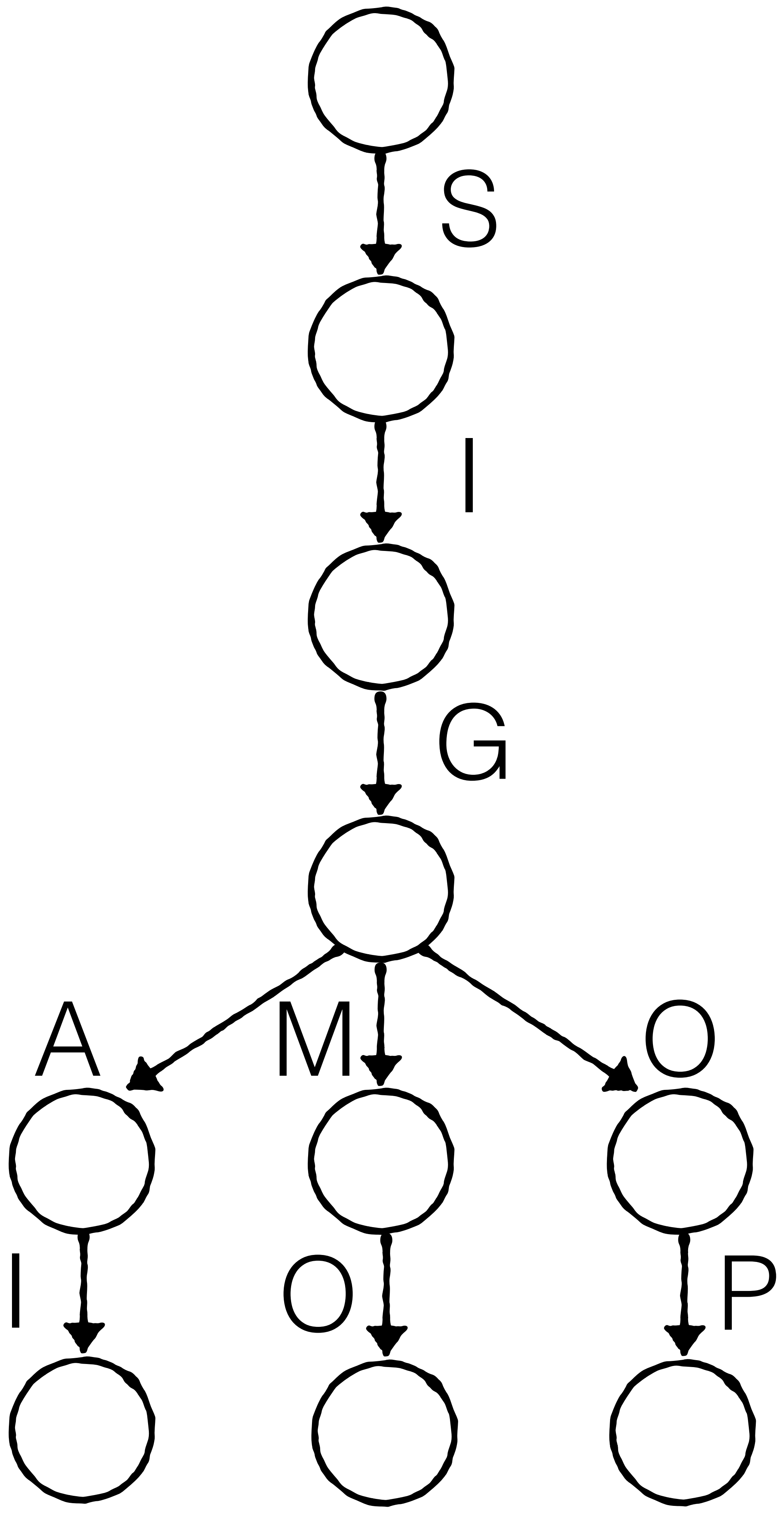
Can truncate later - better FPR, worse space

We now get false positives (e.g., query **SIGMETRICS**)

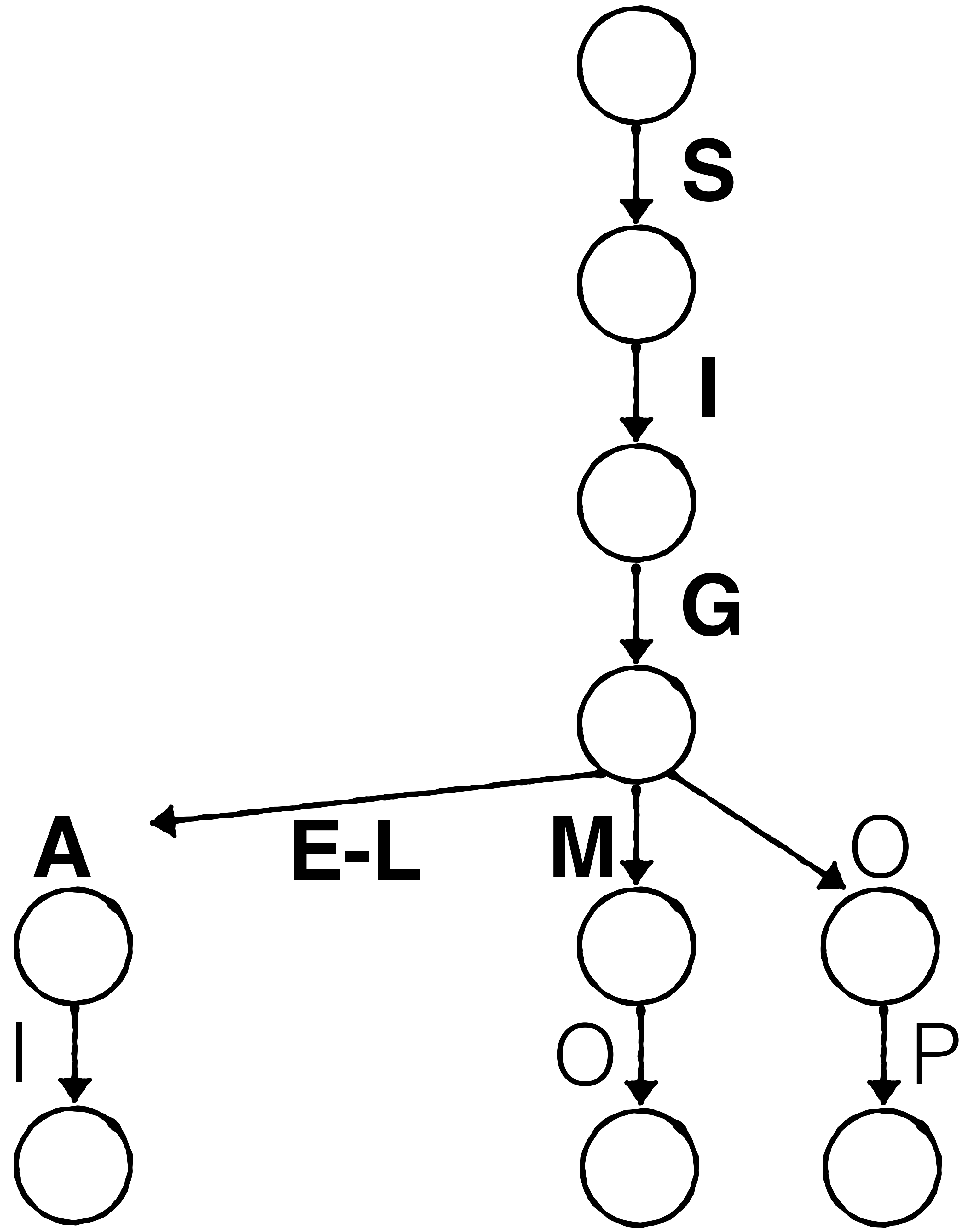


Negative :)

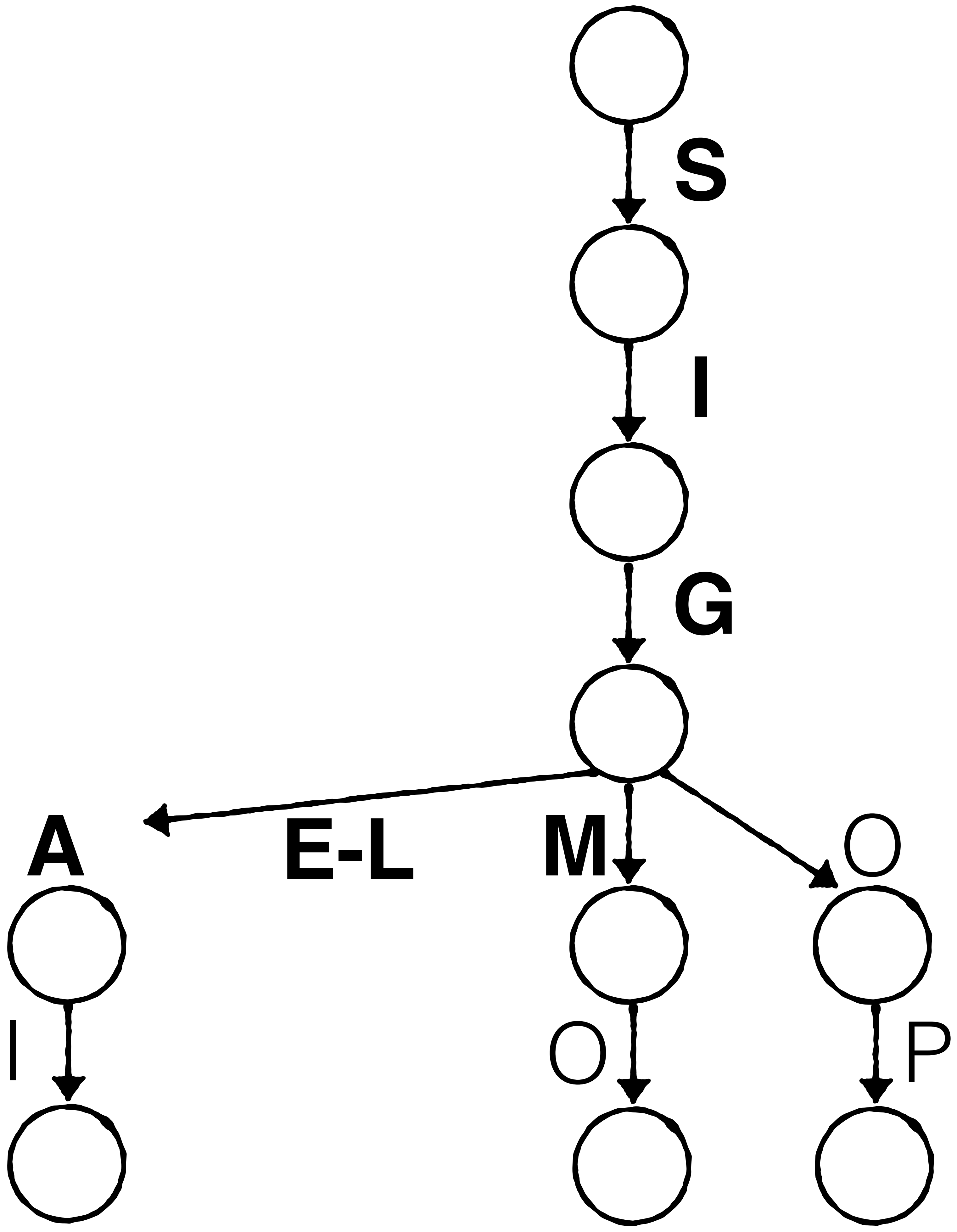
Range query: SIGE - SIGL



Range query: SIG**E - **SIG**L**

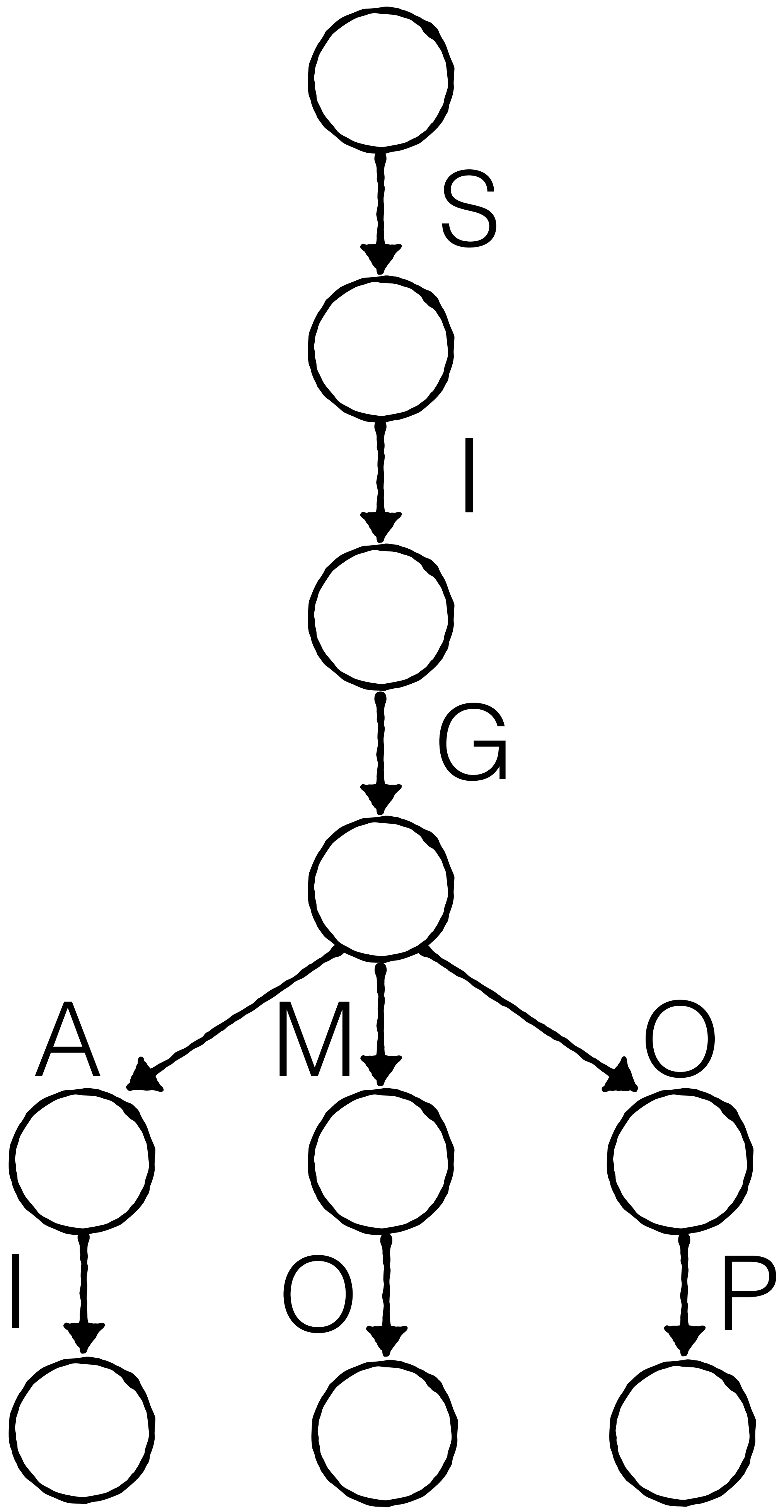


Range query: SIG**E - **SIG**L**

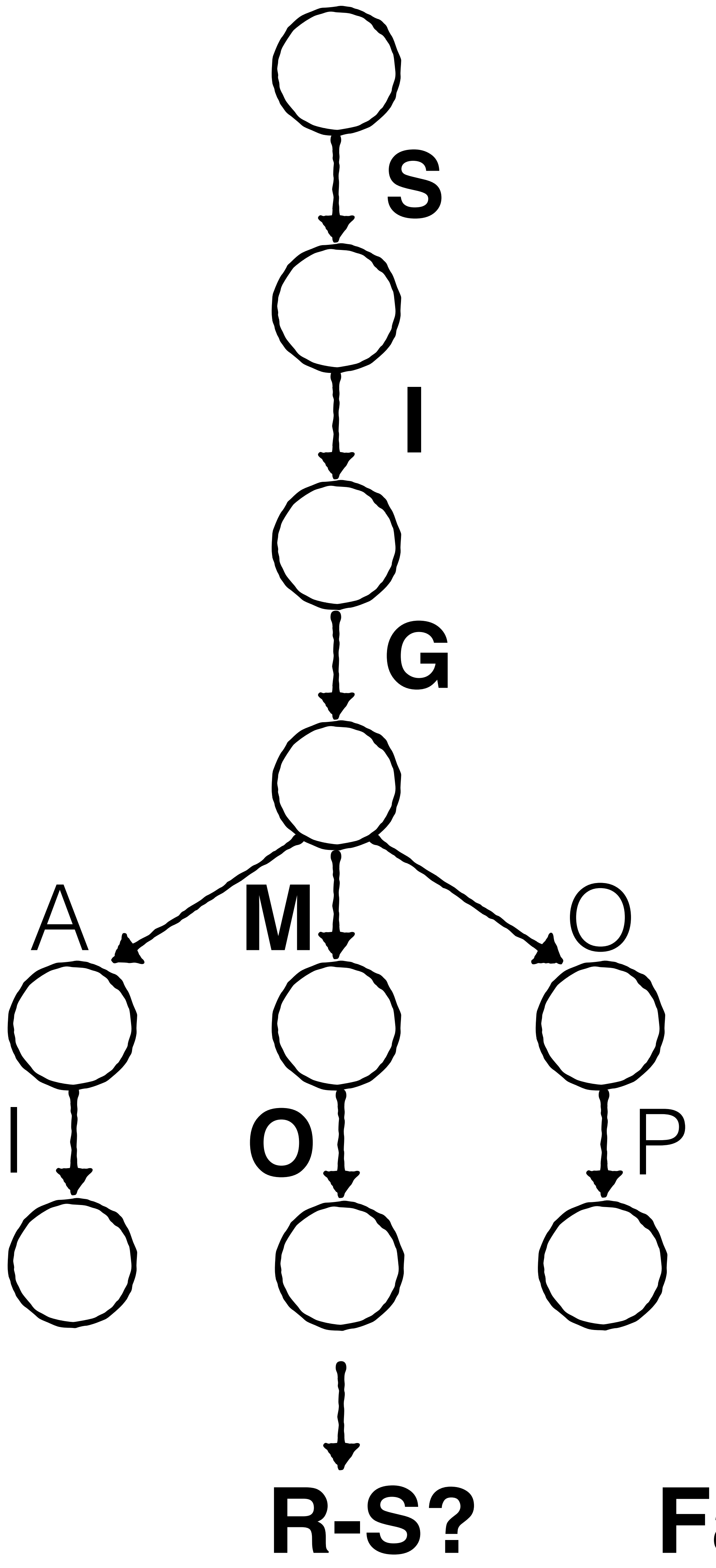


Negative :)

Range query: **SIGMOR - SIGMOS**

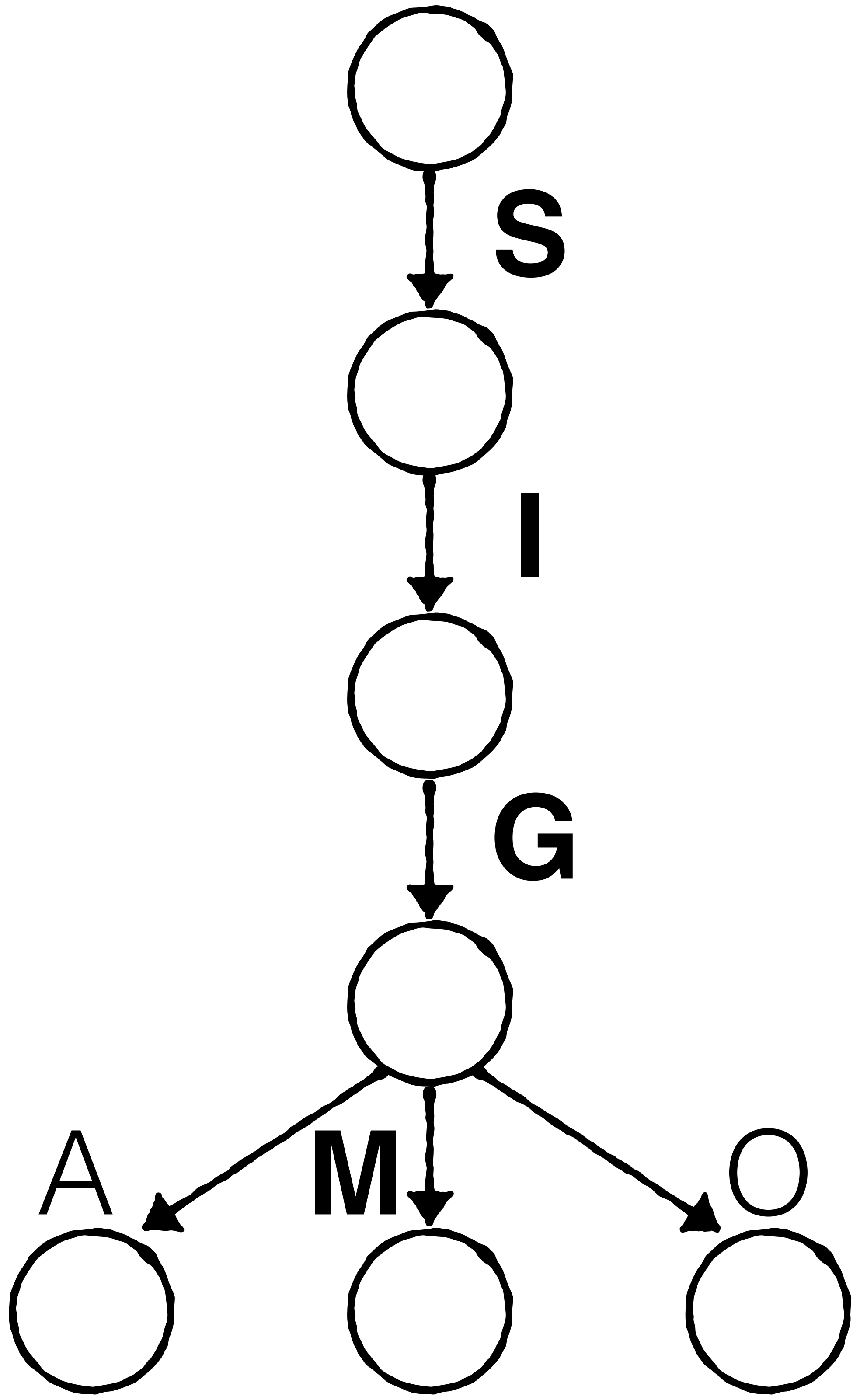


Range query: **SIGMOR - SIGMOS**



False positive

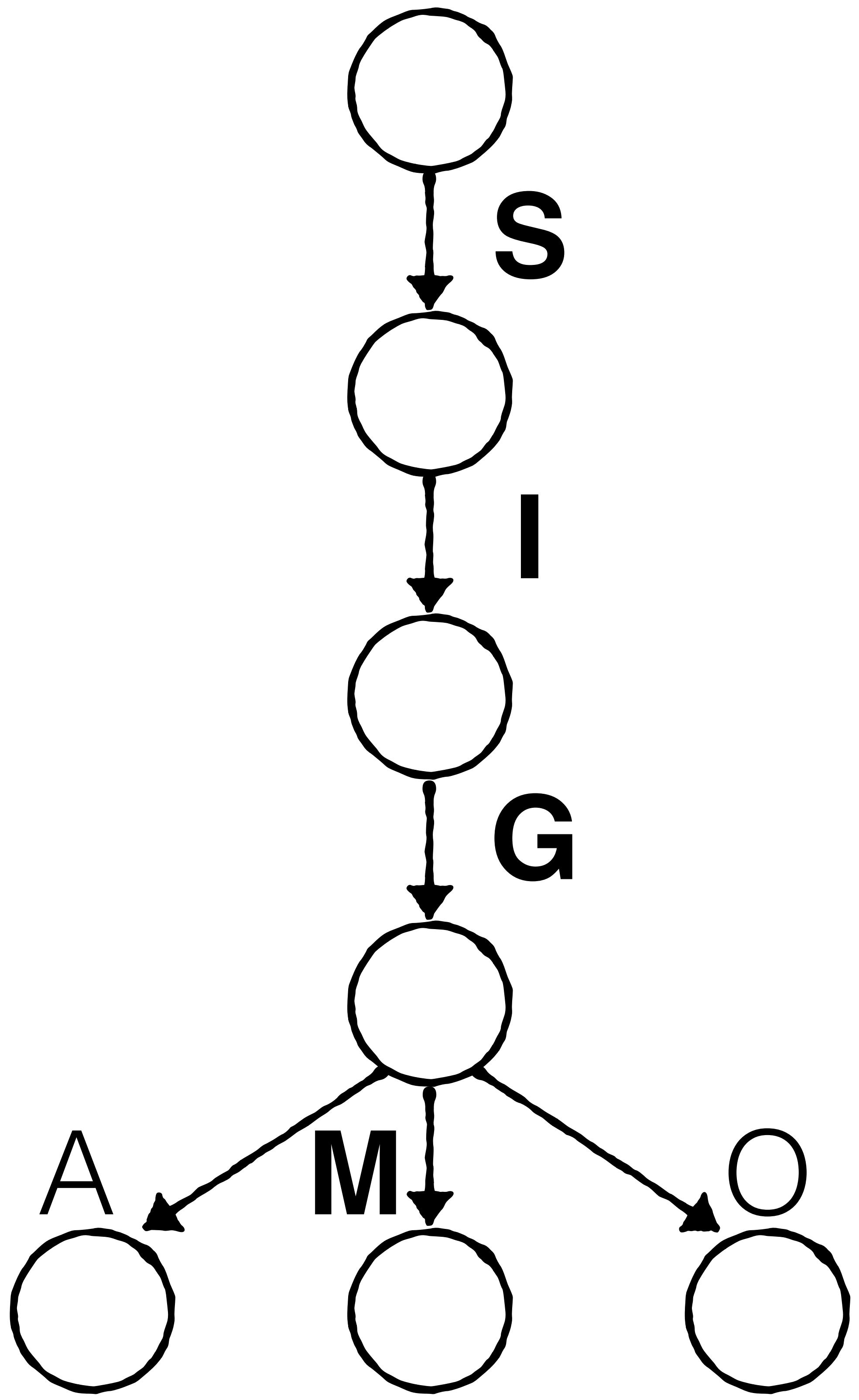
If many queries match existing prefixes, the FPR increases



e.g., **SIGMA**

SIGMOID

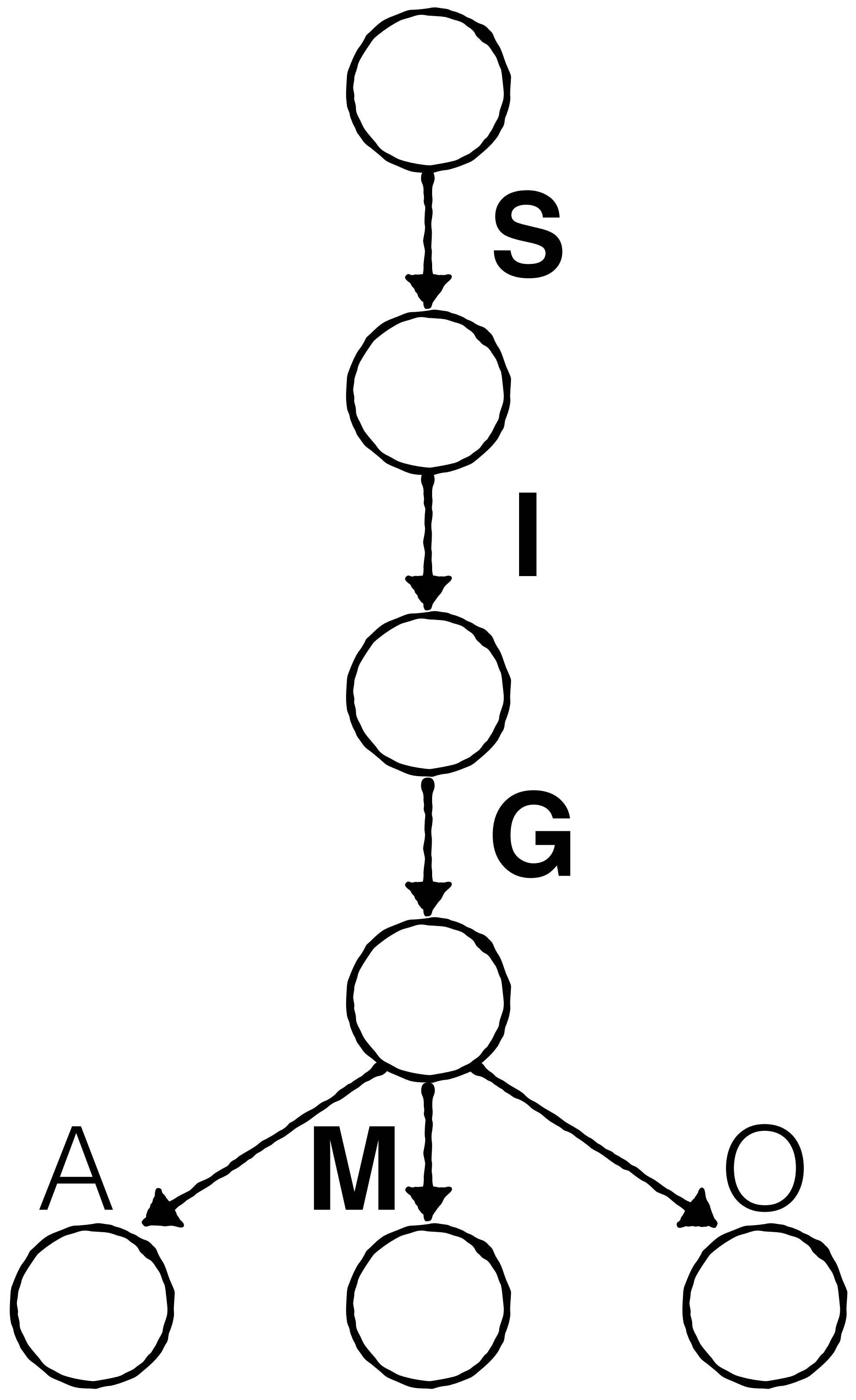
If many queries match existing prefixes, the FPR increases



e.g., SIGMA
SIGMOID

Correlated Workload

If many queries match existing prefixes, the FPR increases



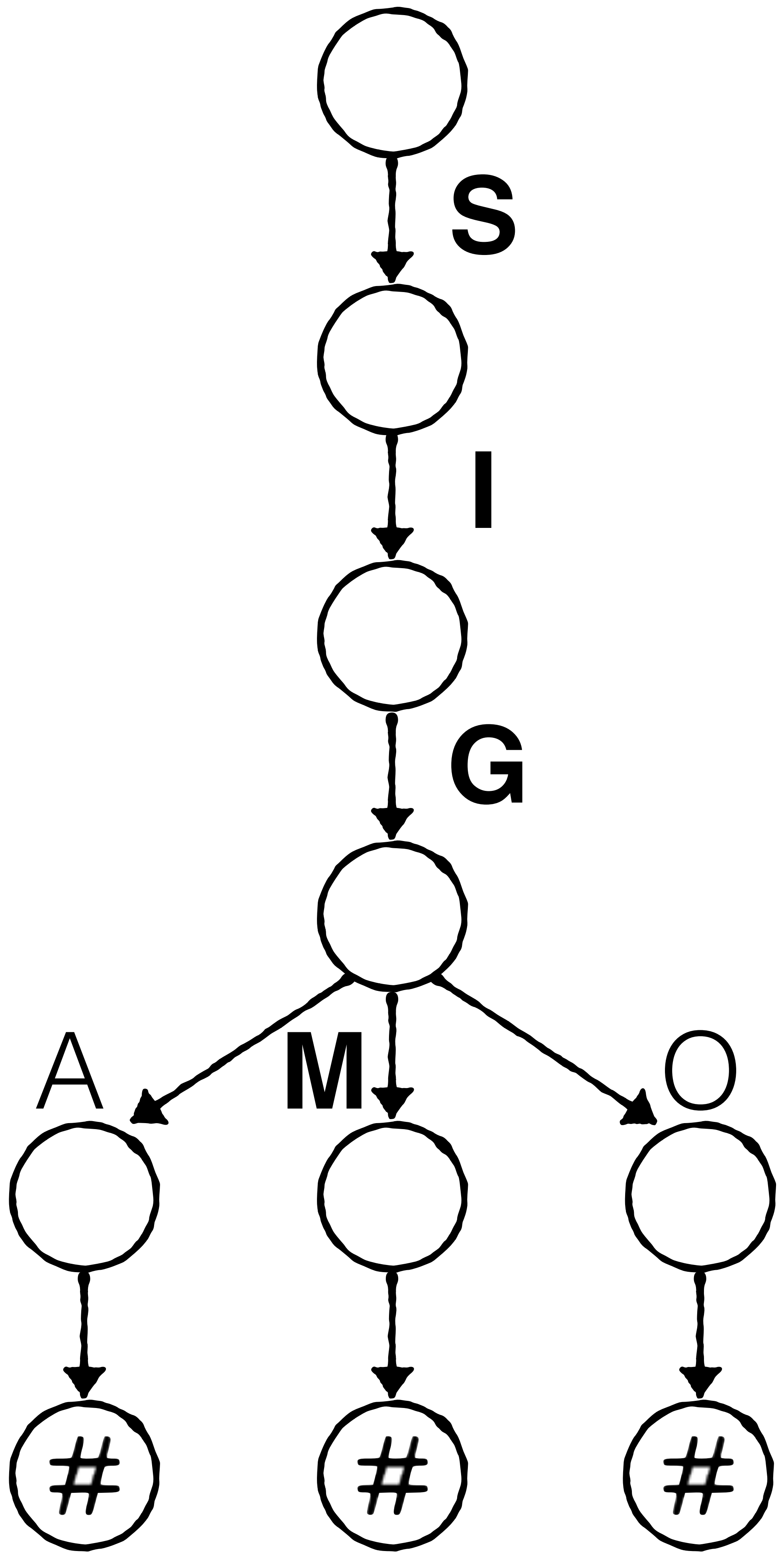
e.g., SIGMA

SIGMOID

Correlated Workload

Can we alleviate this problem for point queries?

If many queries match existing prefixes, the FPR increases

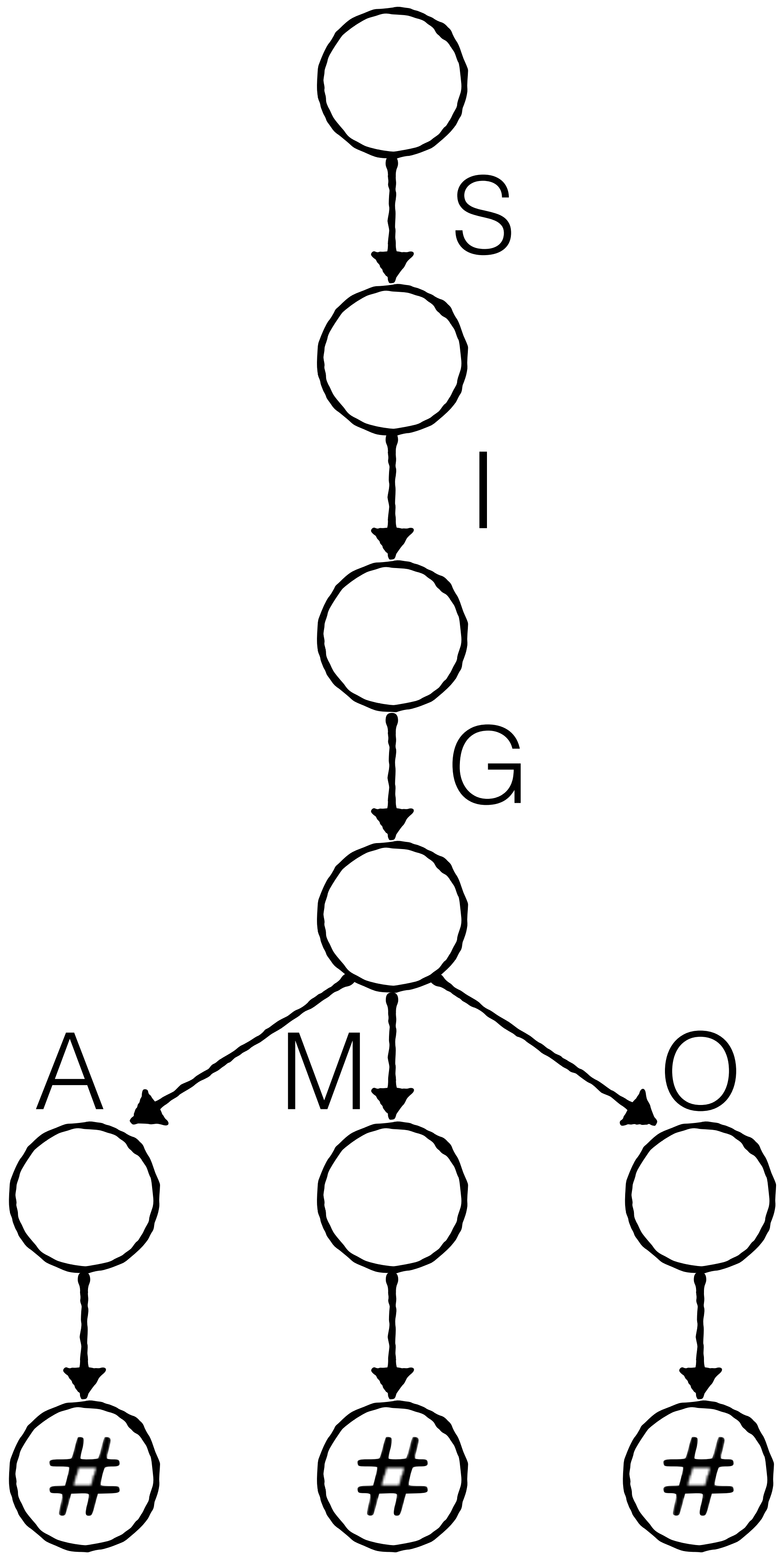


e.g., SIGMA

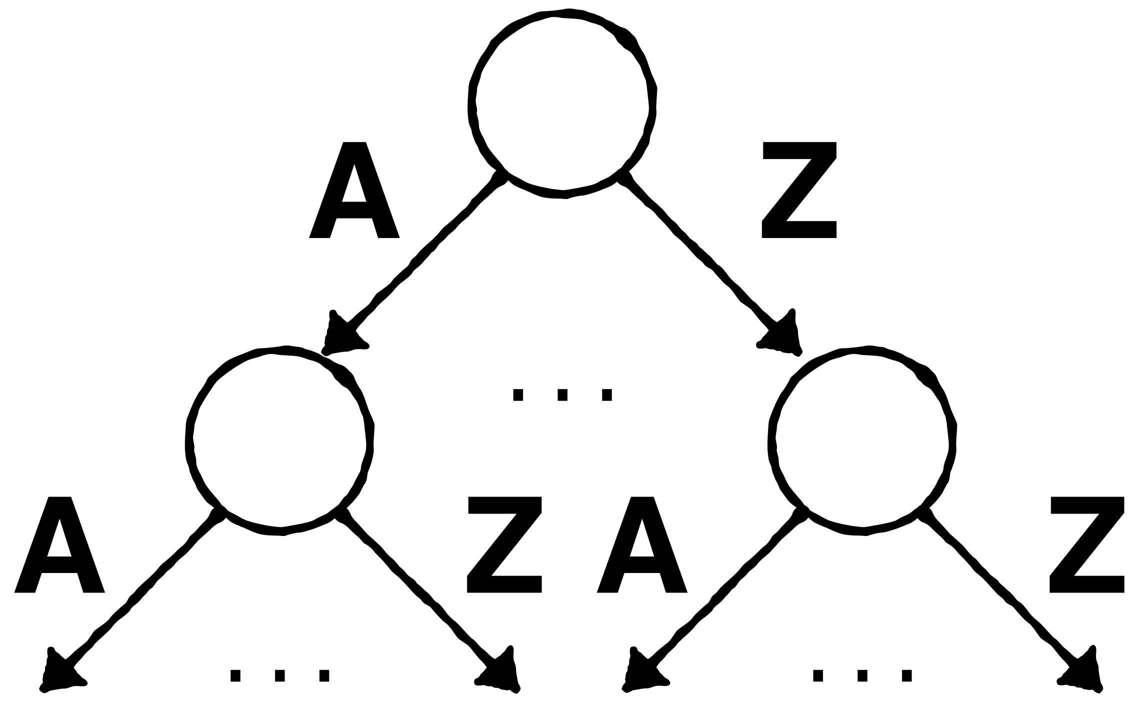
SIGMOID

Can we alleviate this problem for point queries? **Add 1 byte hash**

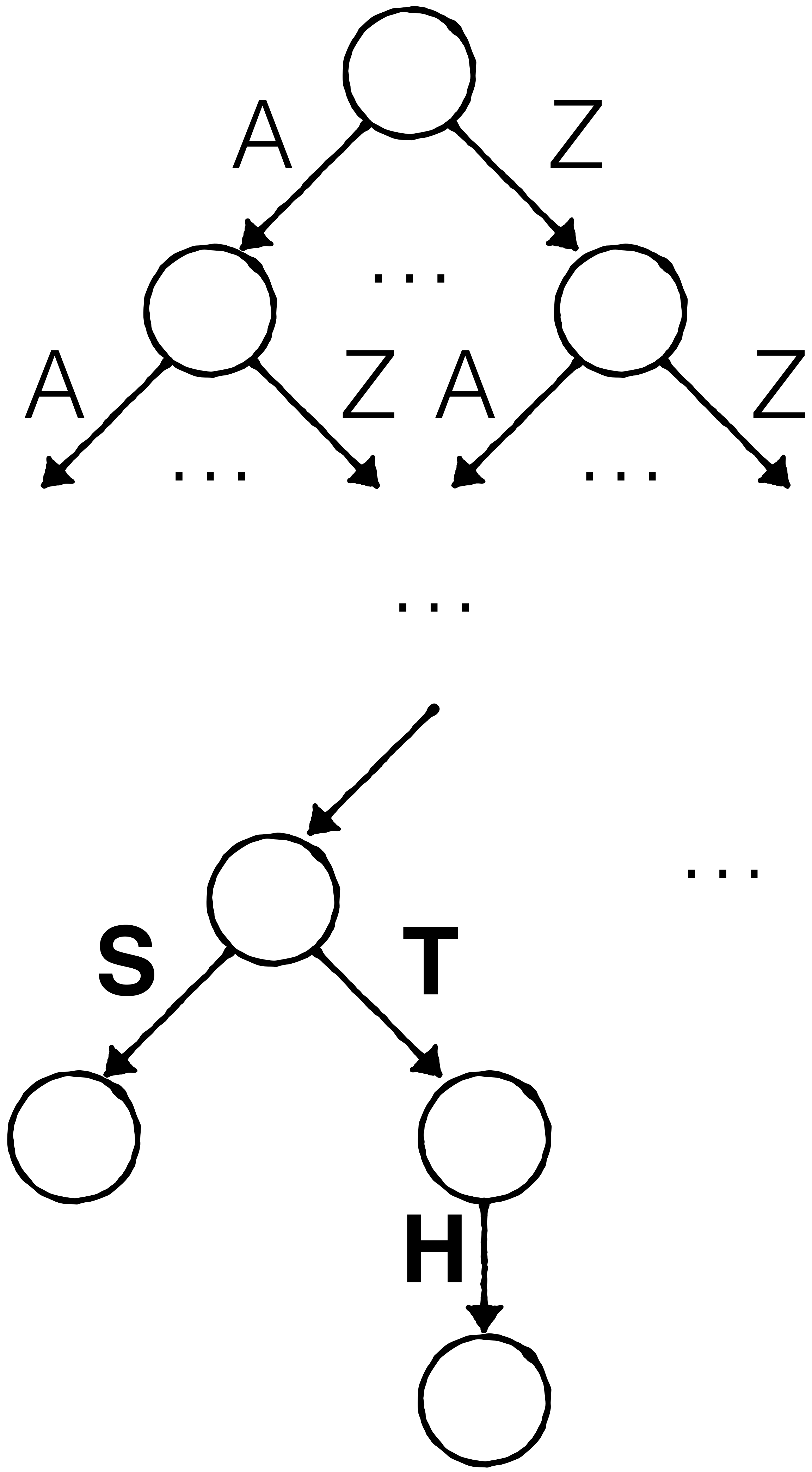
How to encode the trie succinctly?



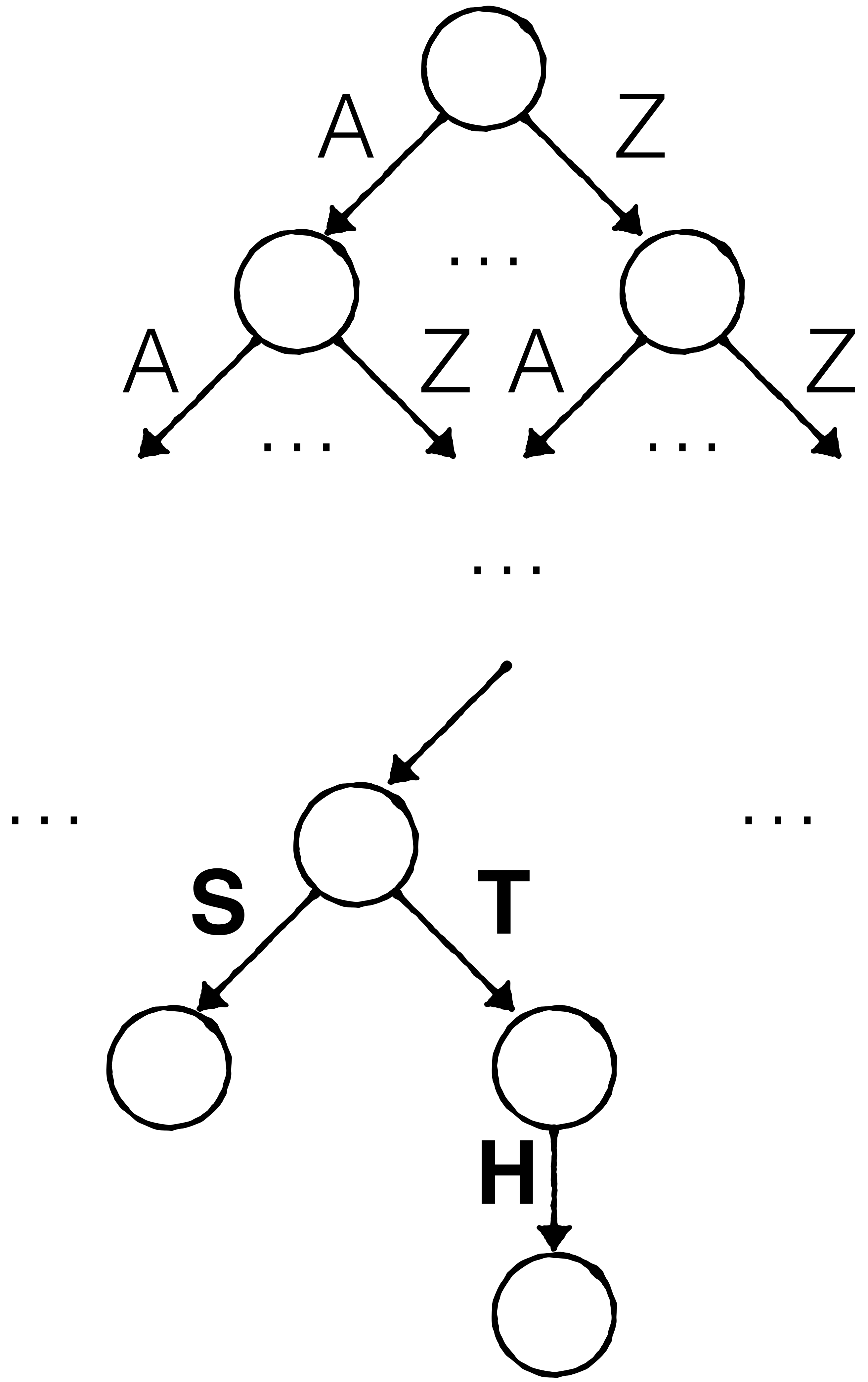
Typically, the upper levels of the trie are more full



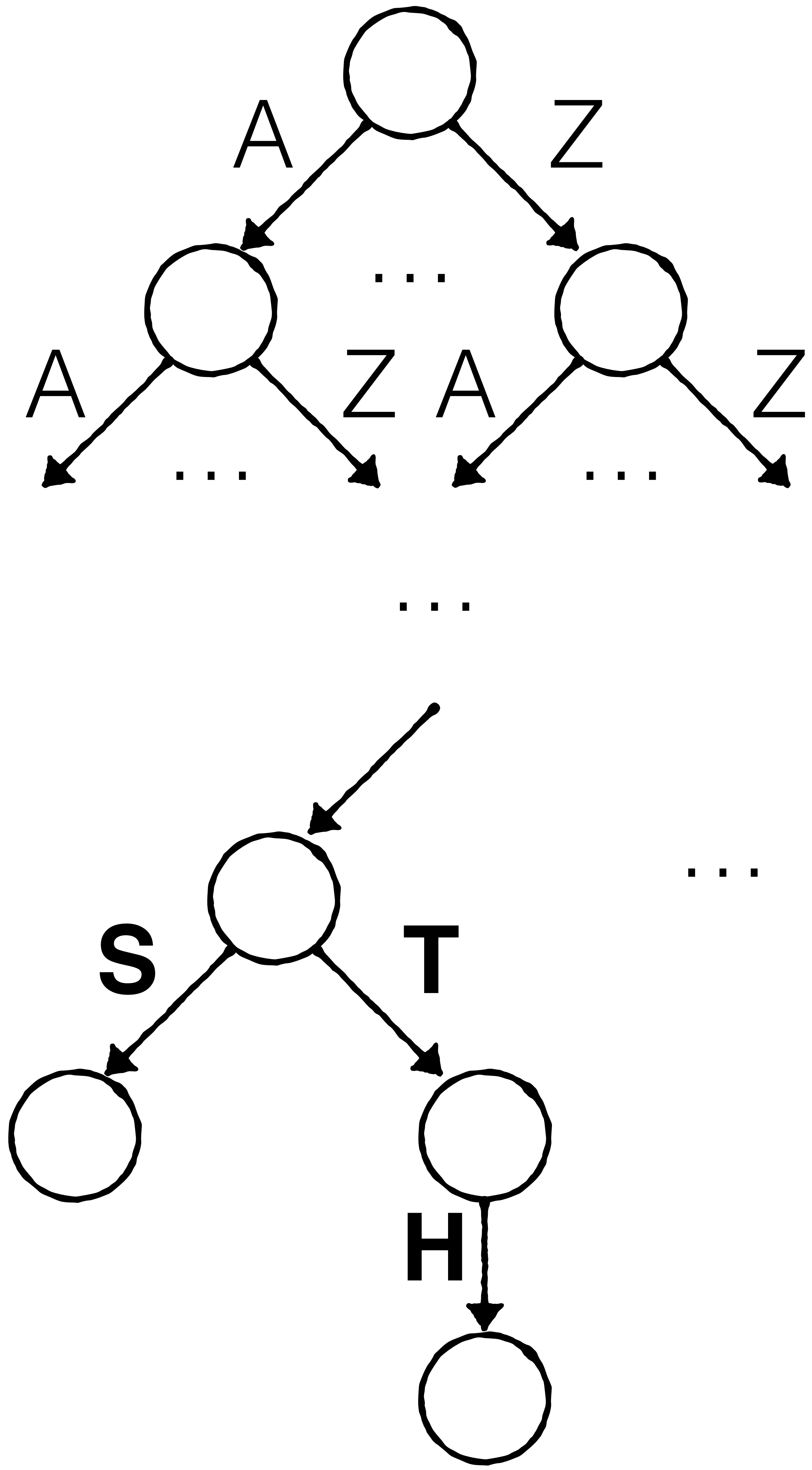
Typically, the upper levels of the trie are more full



More queries go through base layers



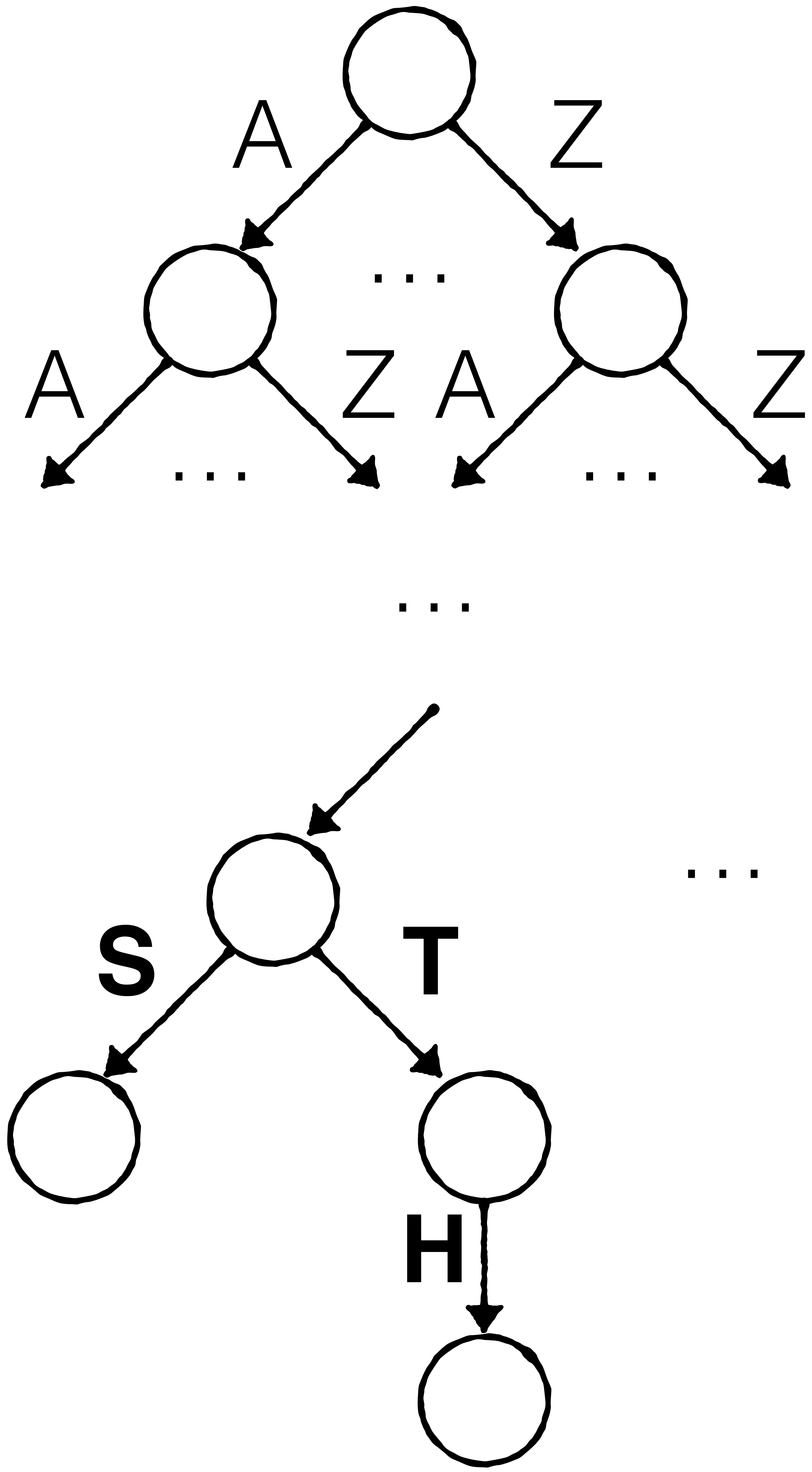
More queries go
through base layers



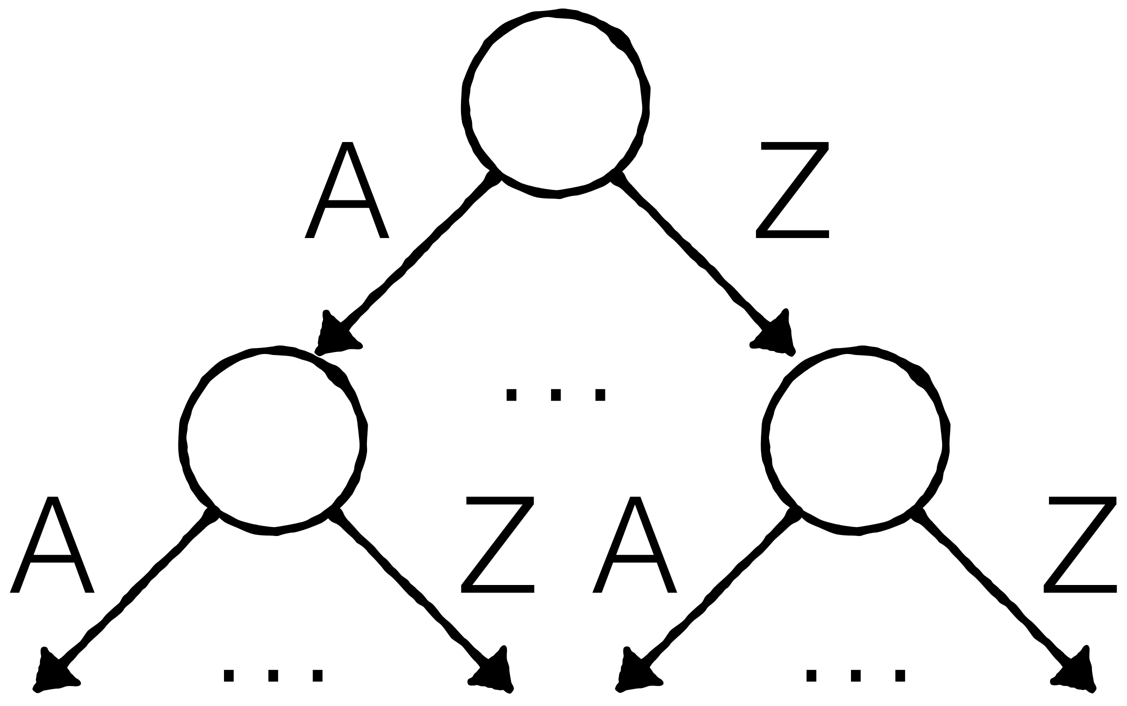
**A leaf is reached on
avg. by fewer queries**

More queries go through base layers

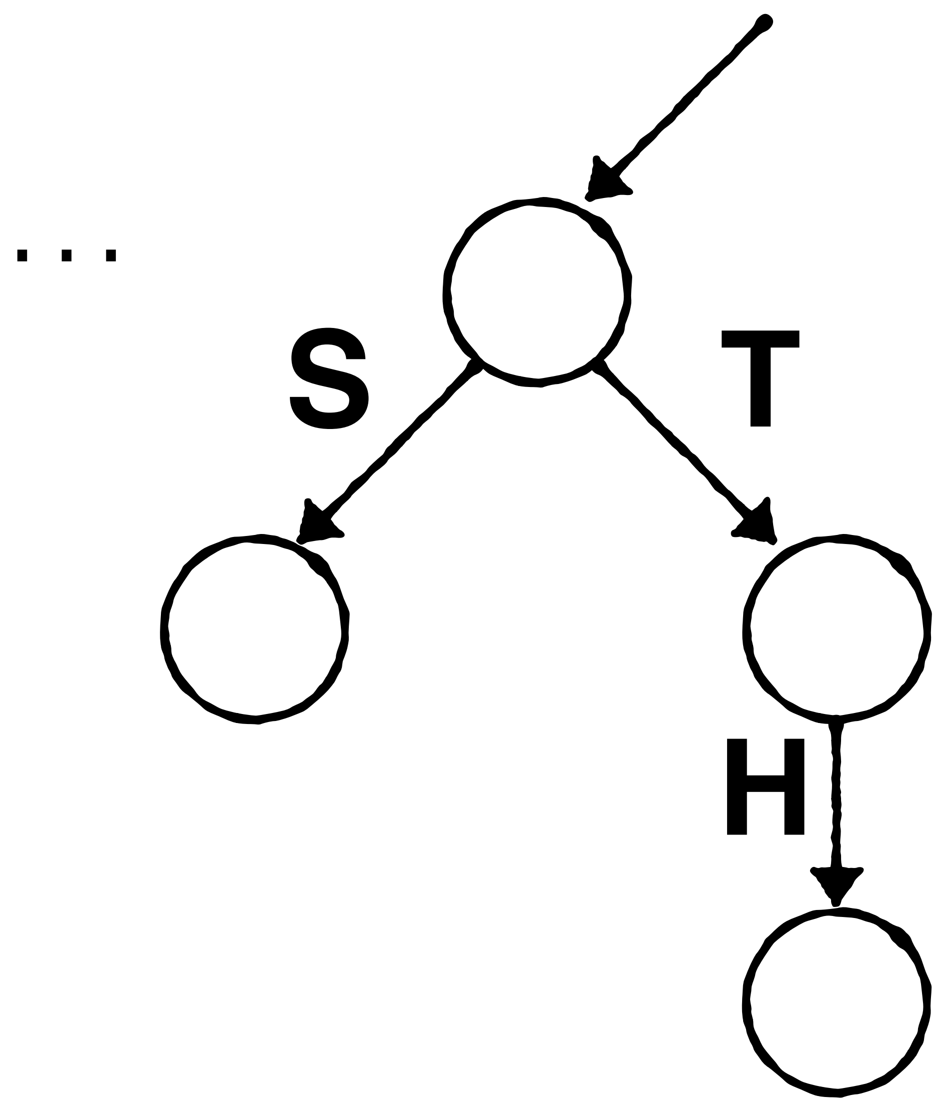
A leaf is reached on avg. by fewer queries



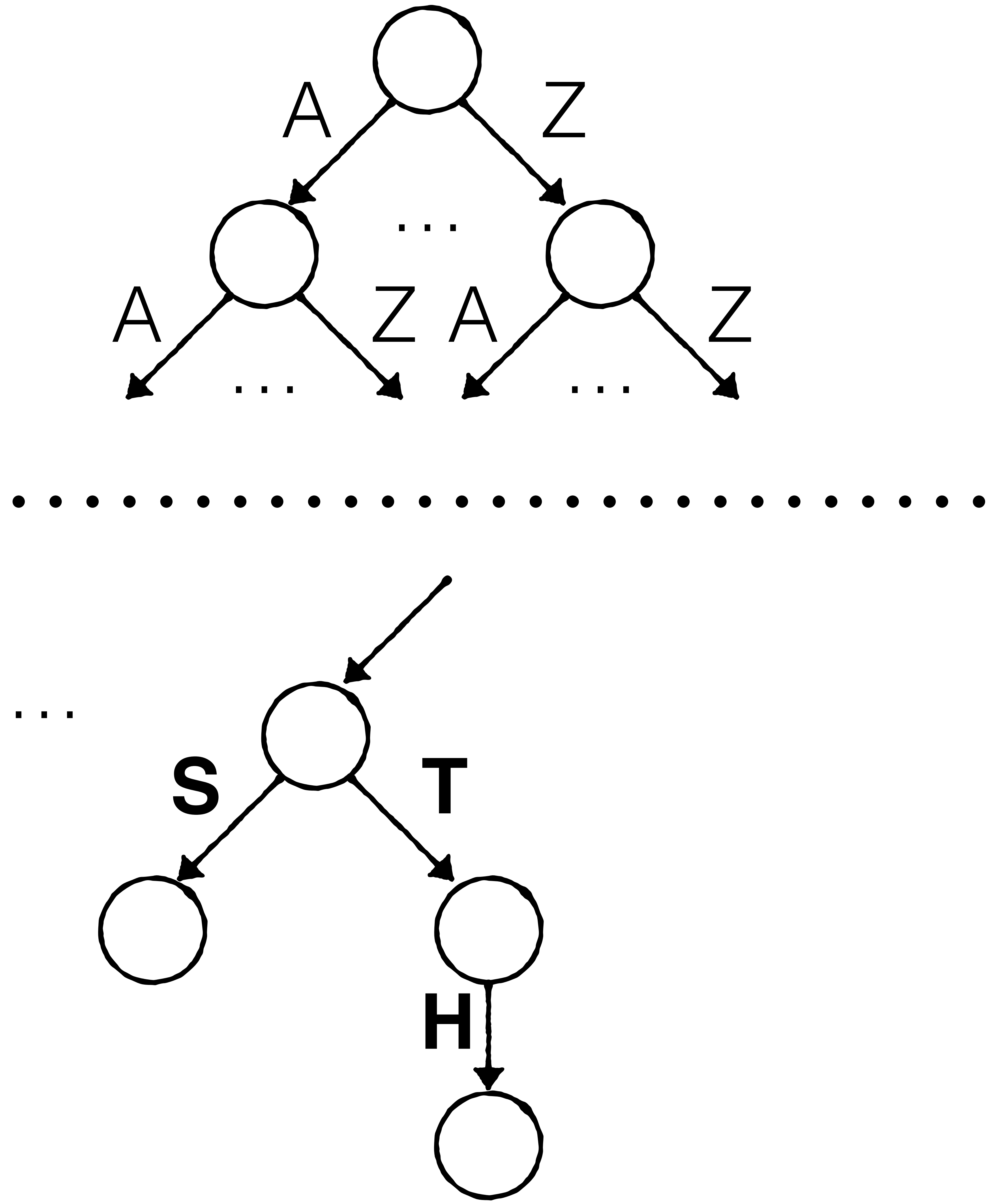
Exponentially more nodes as we move down



Optimize for speed



Optimize for space

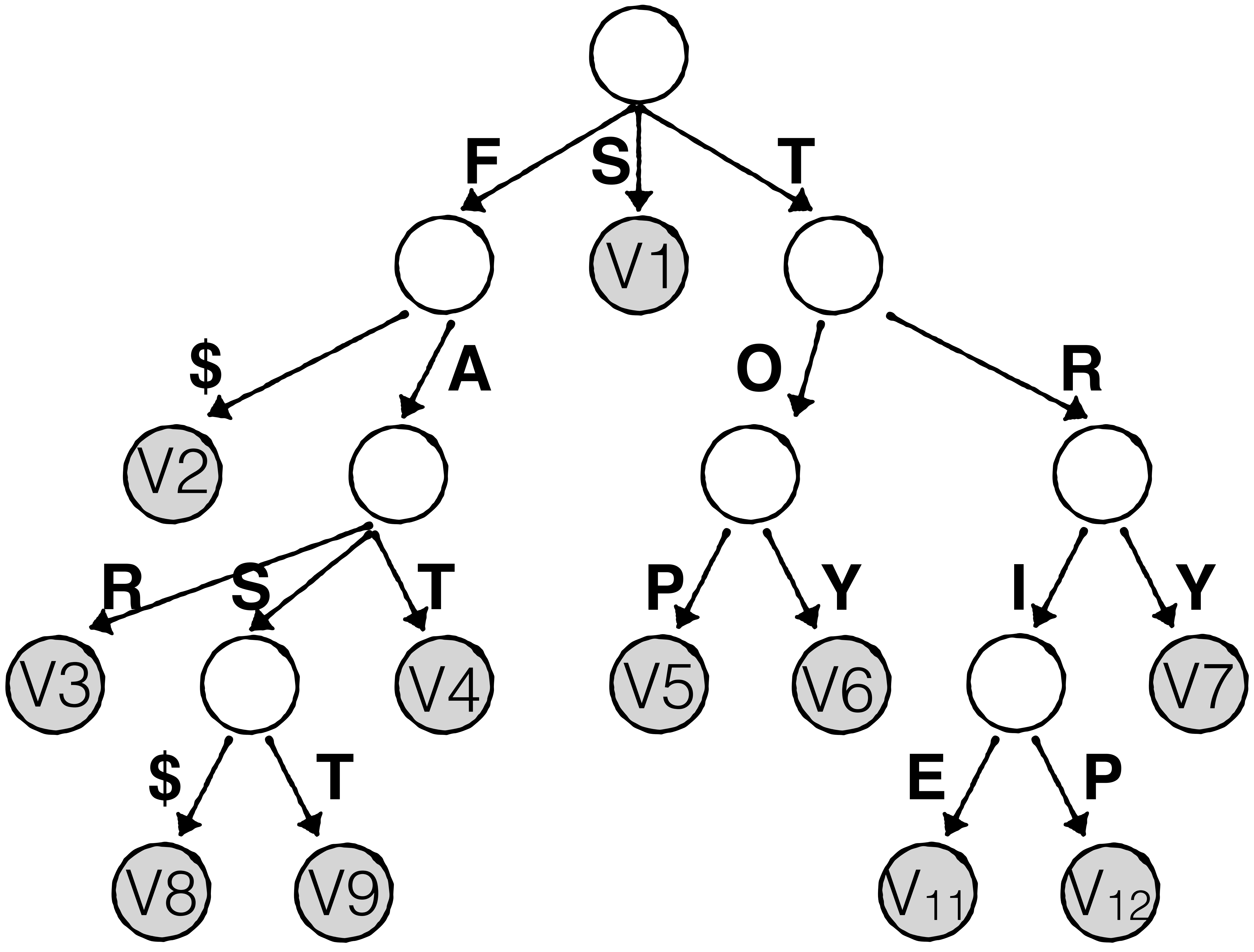


Optimize for speed

1/64

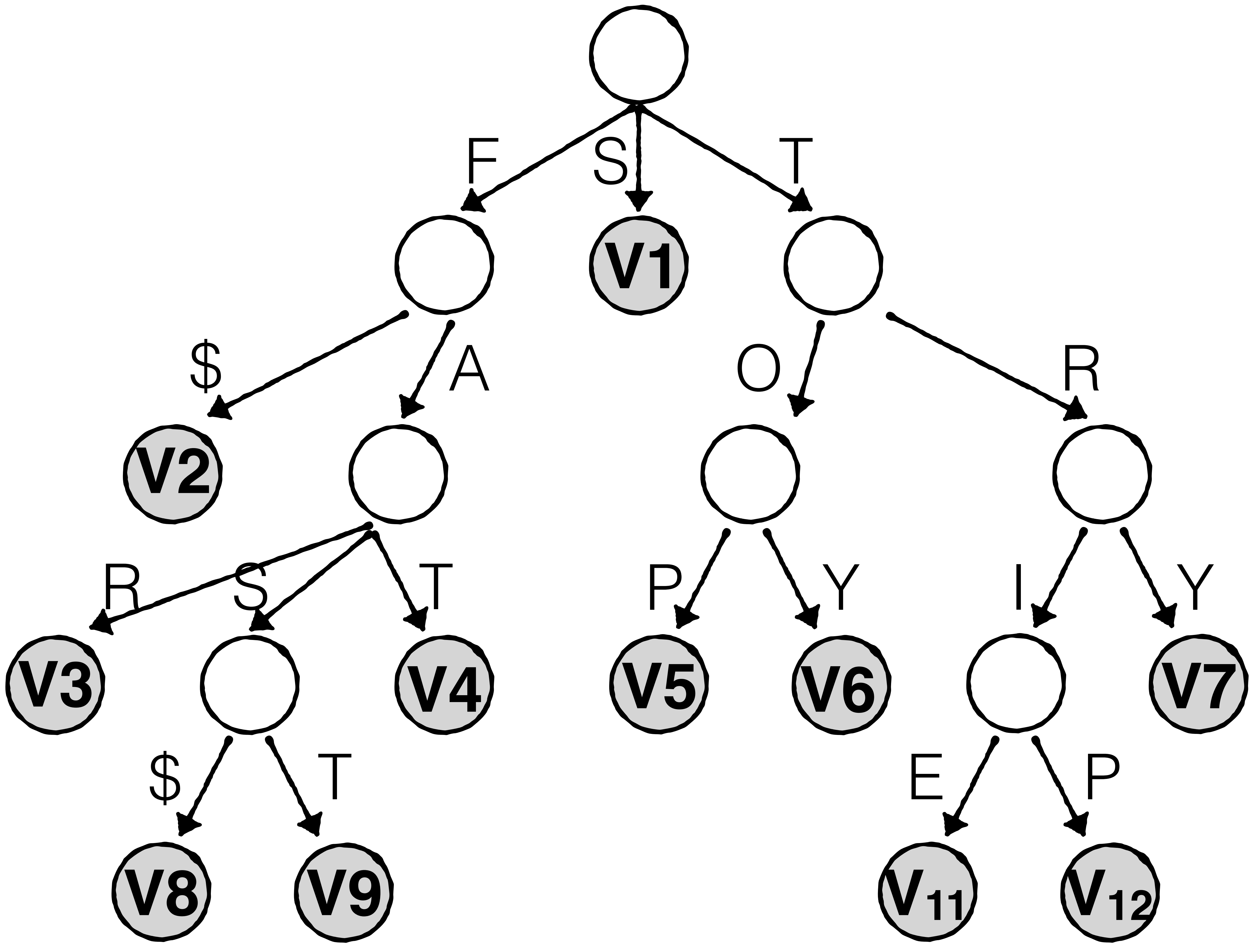
Optimize for space

63/64



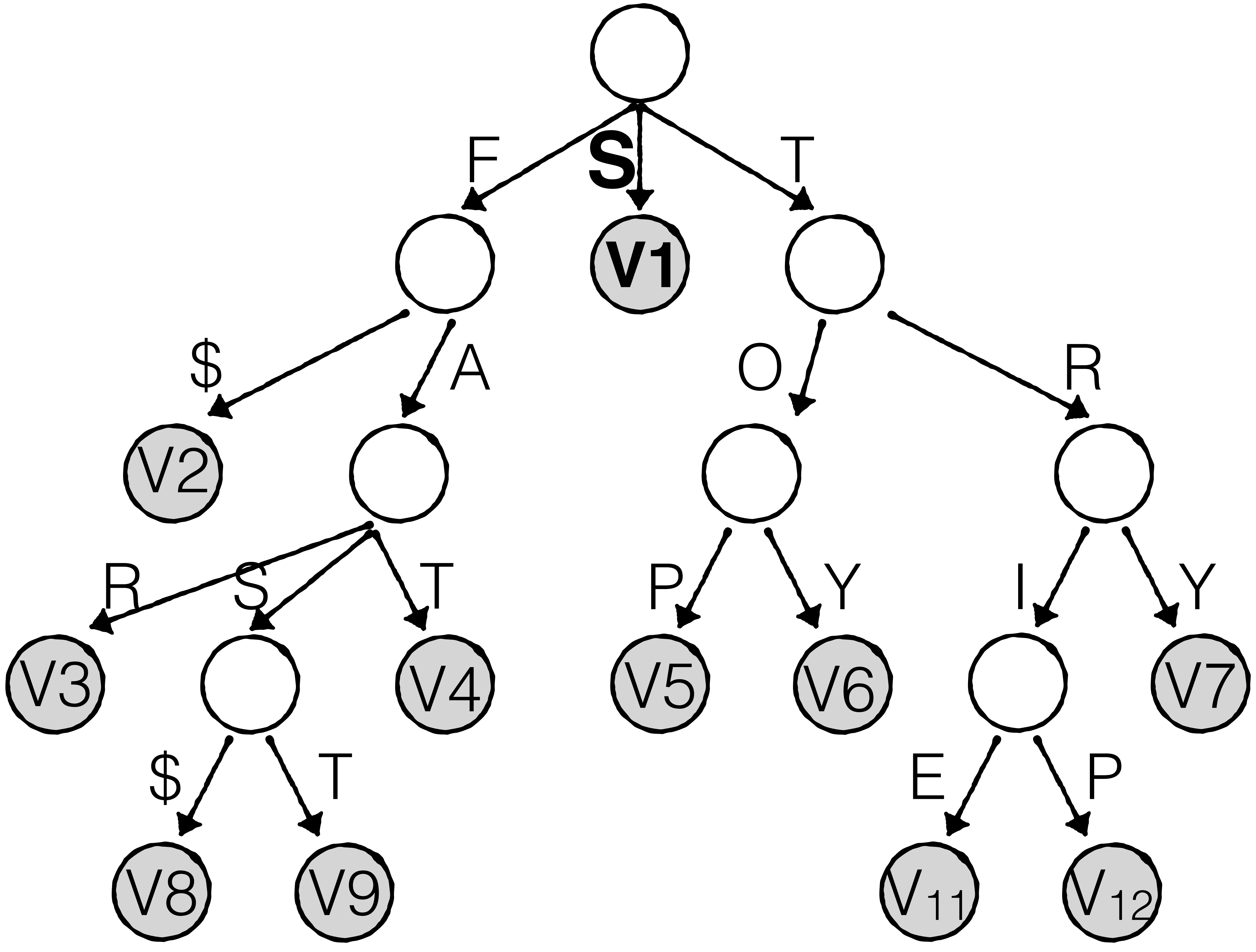
F, FAR, FAS, FAST, FAT, S, TOP, TOY, TRIE, TRIP, TRY

Each key leads to leaf payload (e.g., pointer and/or hash)



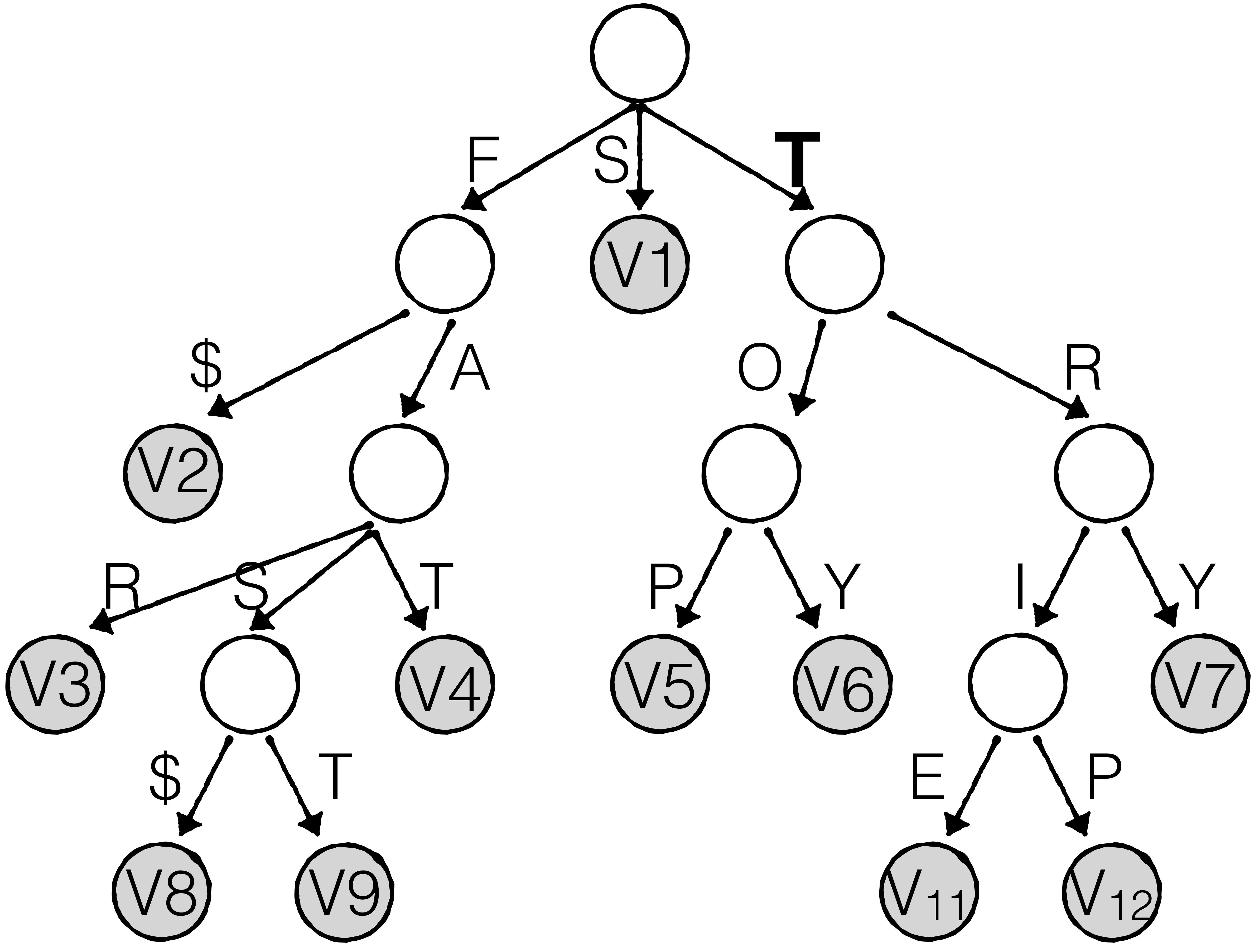
F, FAR, FAS, FAST, FAT, S, TOP, TOY, TRIE, TRIP, TRY

S leads to leaf: full key
not prefix of any other key



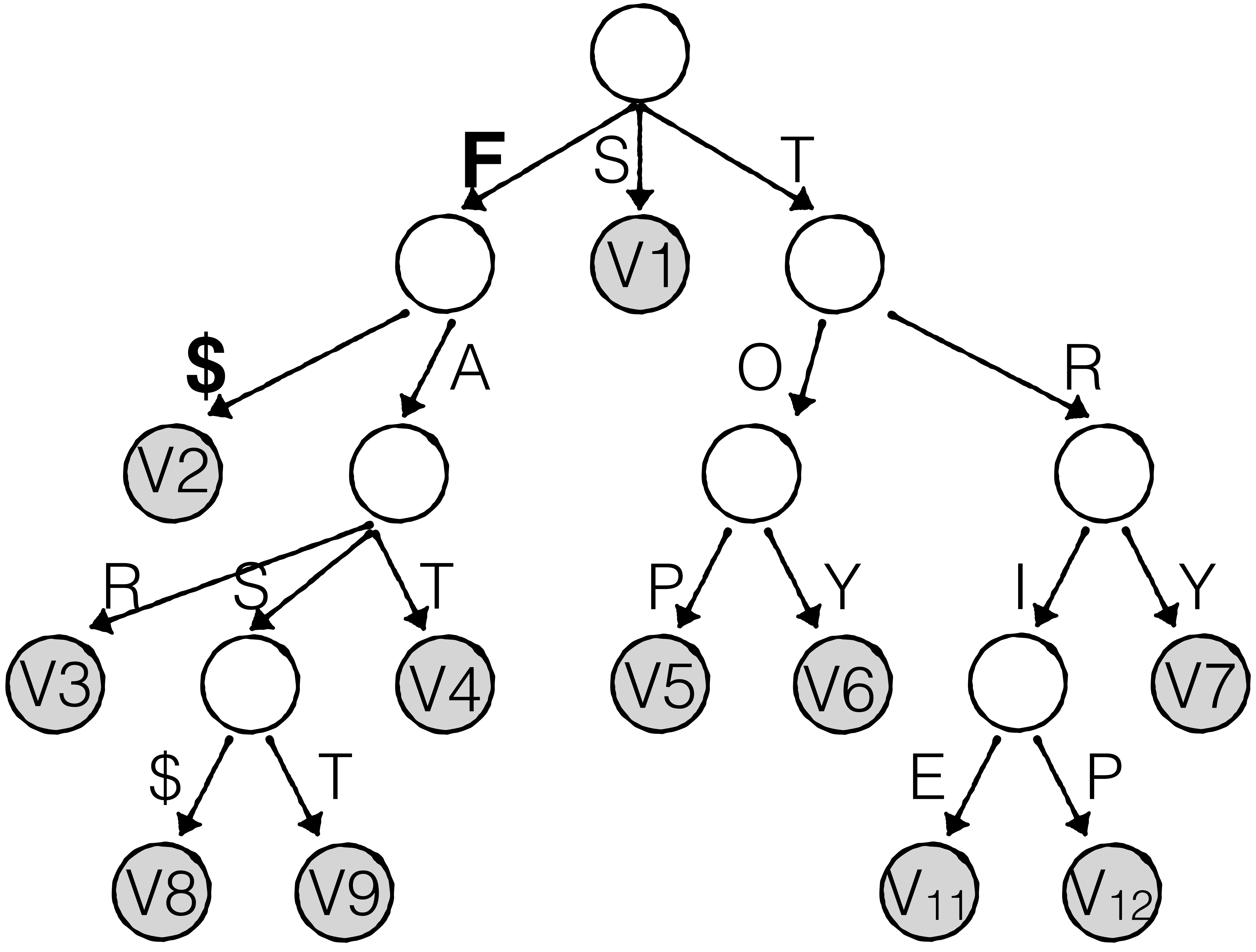
F, FAR, FAS, FAST, FAT, **S**, TOP, TOY, TRIE, TRIP, TRY

T leads to internal node: not full key
prefix of one or more other keys



F, FAR, FAS, FAST, FAT, S, **T**OP, **T**OY, **T**RIE, **T**RIP, **T**RY

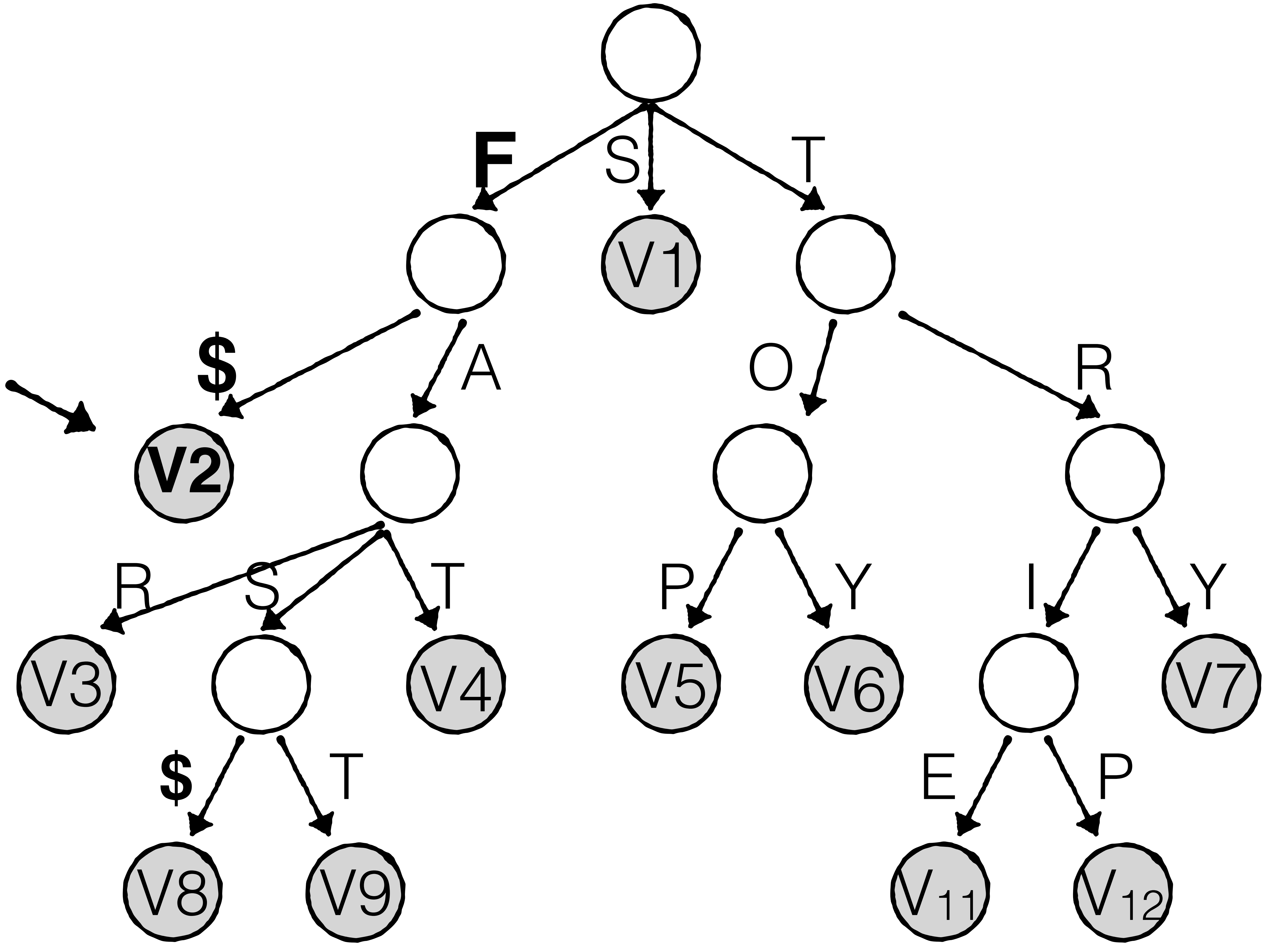
F leads to internal node with $ child: full key
prefix of one or more other keys



F, **F**AR, **F**AS, **F**AST, **F**AT, S, TOP, TOY, TRIE, TRIP, TRY

F leads to internal node with $ child: full key
prefix of one or more other keys

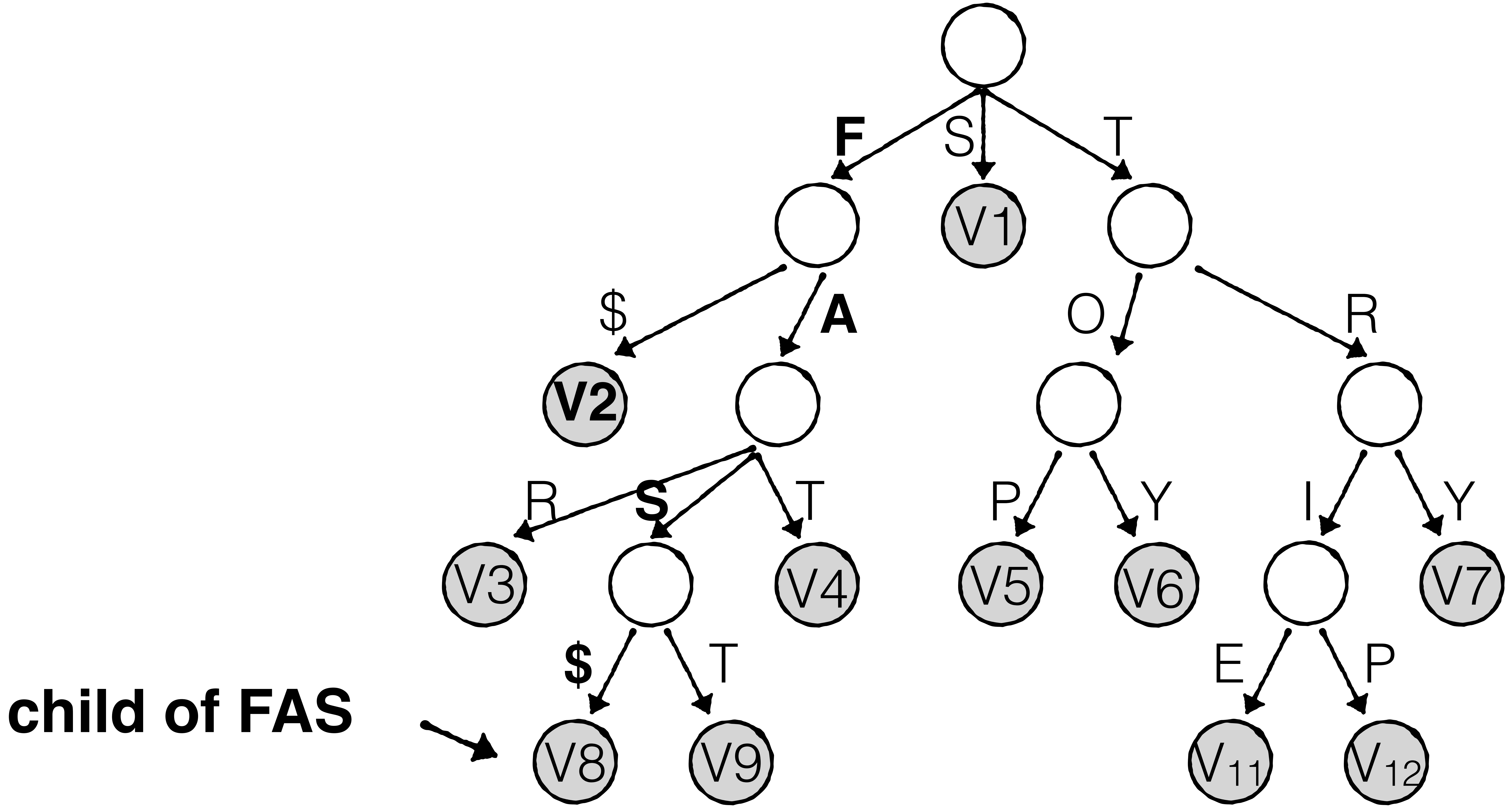
child of F



F, **F**AR, **F**AS, **F**AST, **F**AT, S, TOP, TOY, TRIE, TRIP, TRY

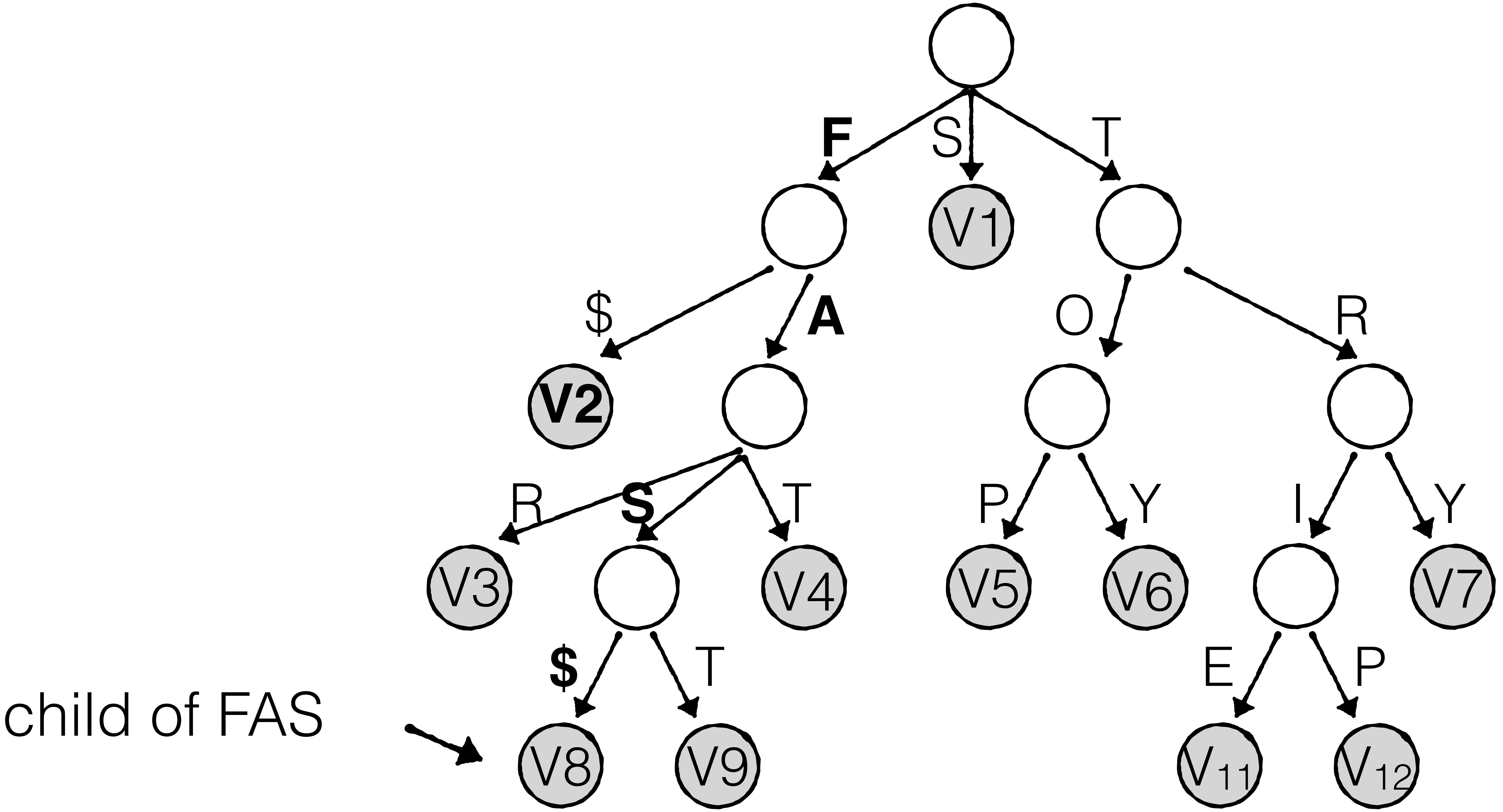
FAS leads to internal node with $ child: full key

prefix of one or more other keys



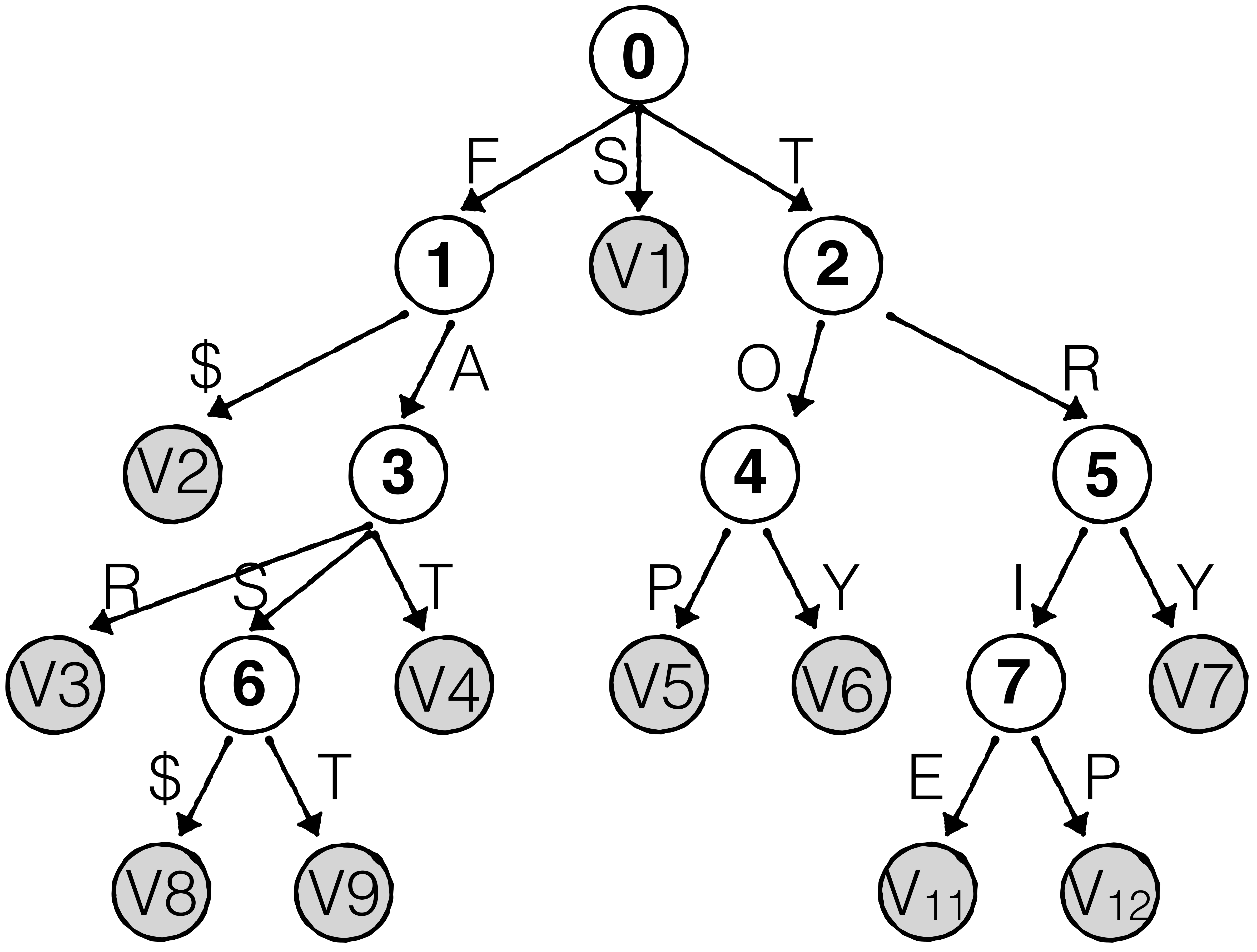
F, **F**AR, **F**AS, **F**AST, **F**AT, S, TOP, TOY, TRIE, TRIP, TRY

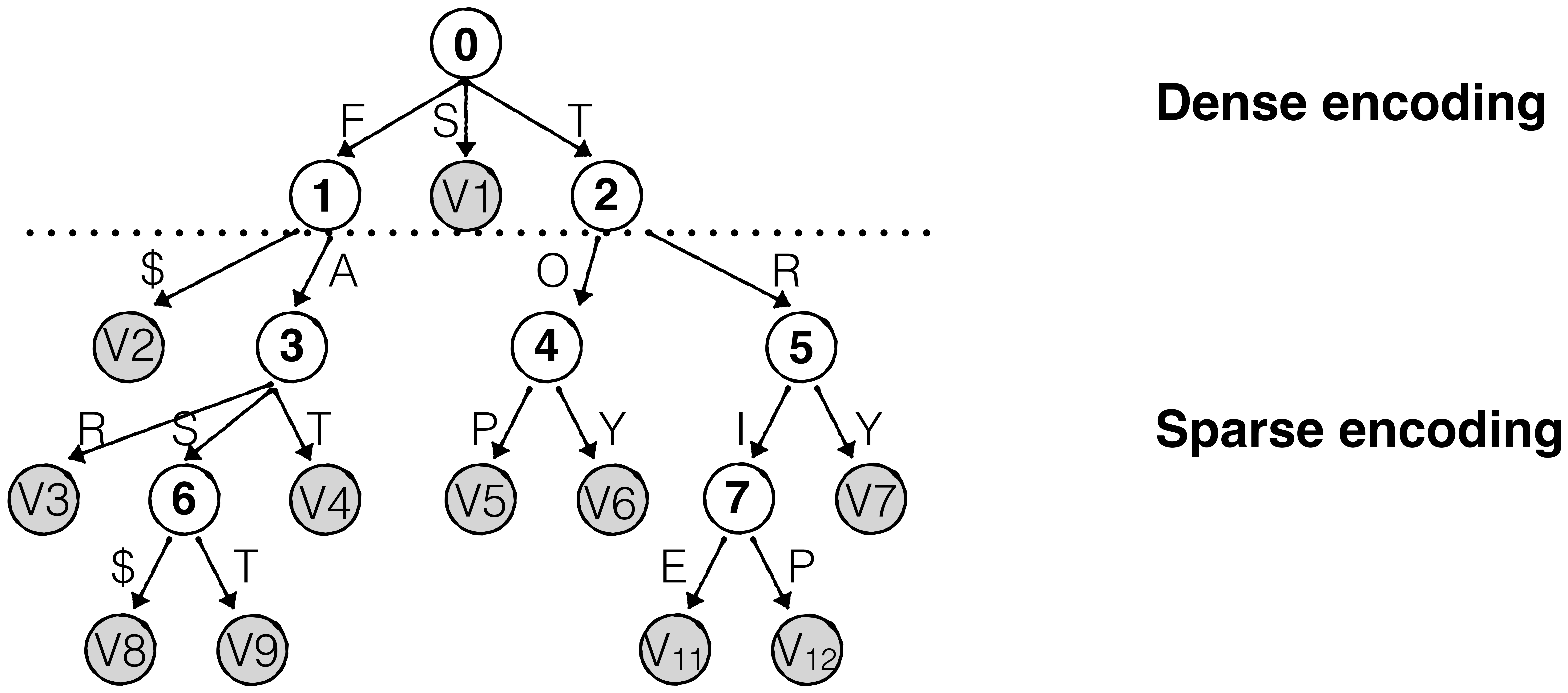
A prefix is always represented by an internal node

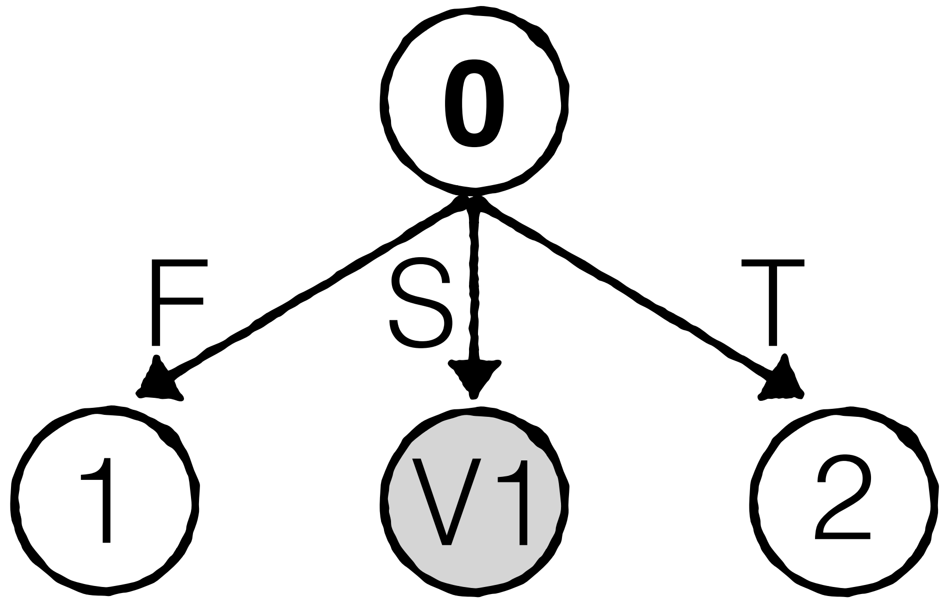


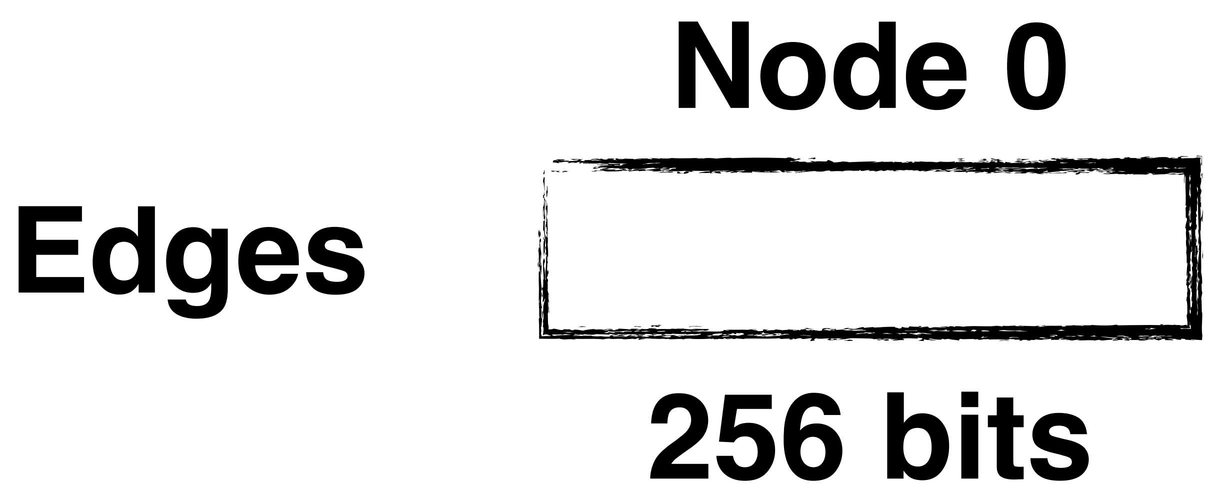
F, **F**AR, **F**AS, **F**AST, **F**AT, S, TOP, TOY, TRIE, TRIP, TRY

Label internal nodes in breadth-first order for clarity

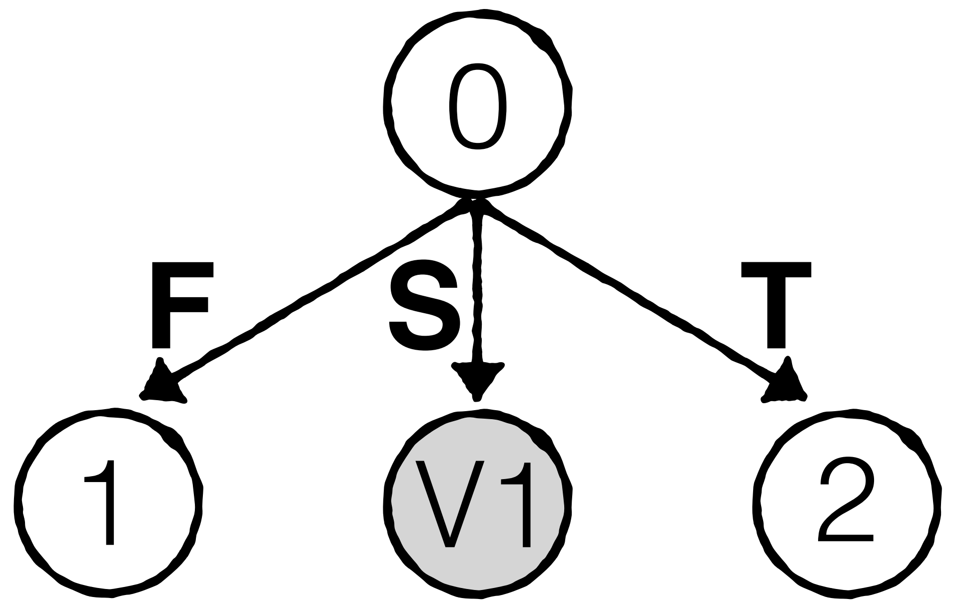


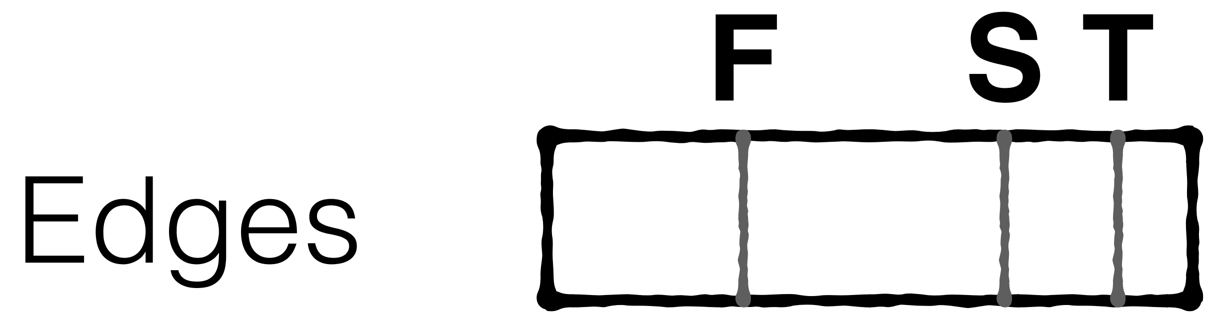

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
R
S
T
P
Y
I
Y
V3
6
V4
V5
V6
7
V7
$
T
E
P
V8
V9
V11
V12
Dense encoding
Sparse encoding


0
F
S
T
1
V1
2


Node 0
Edges
256 bits

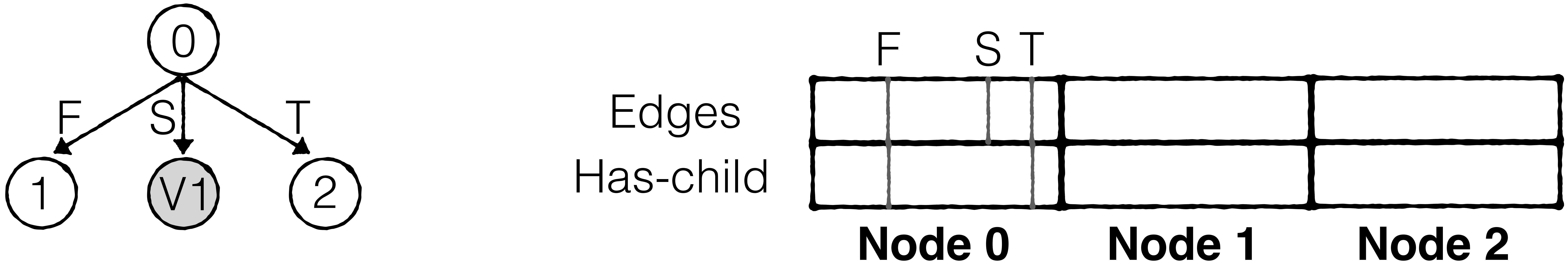
One bit set for every connected character



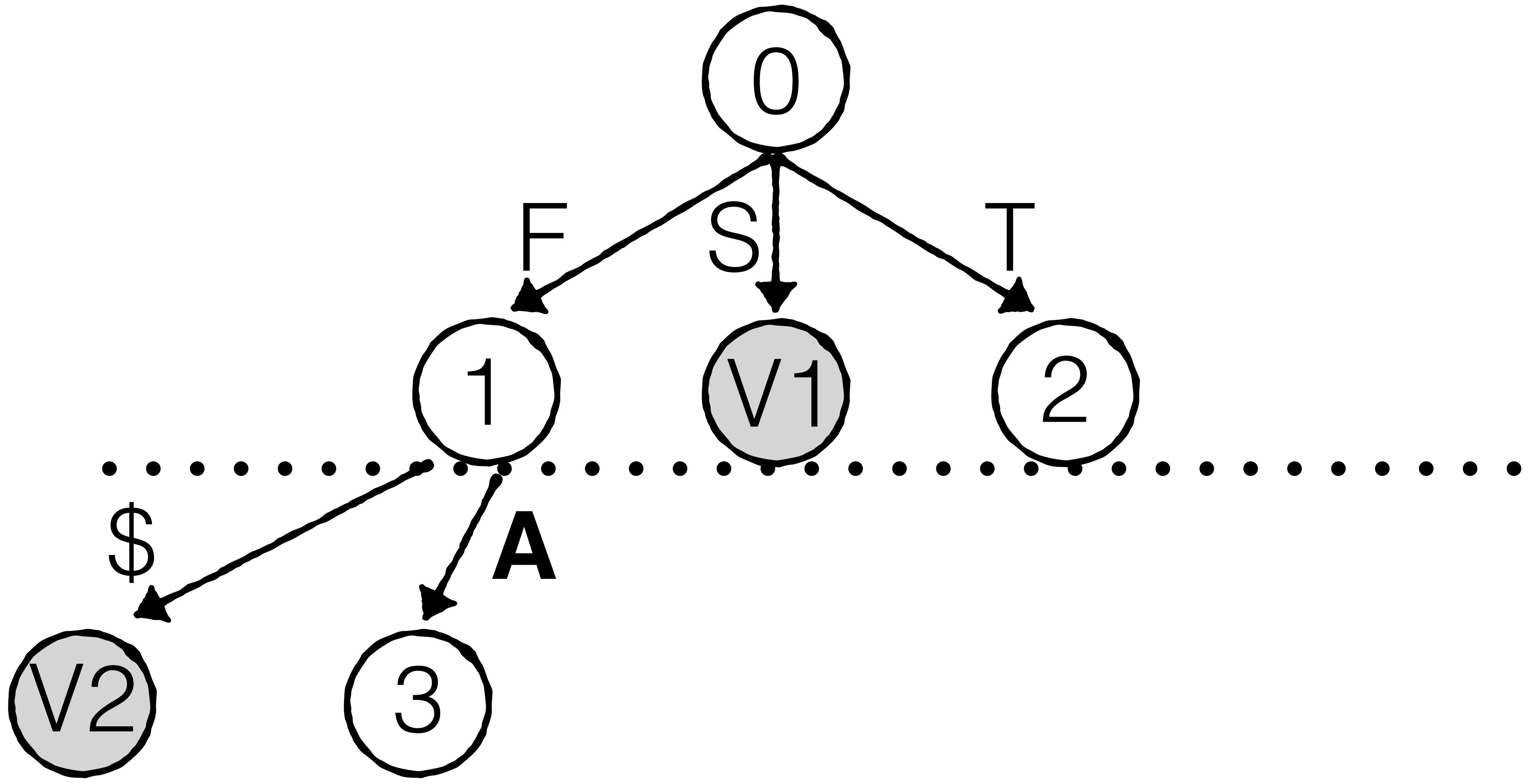


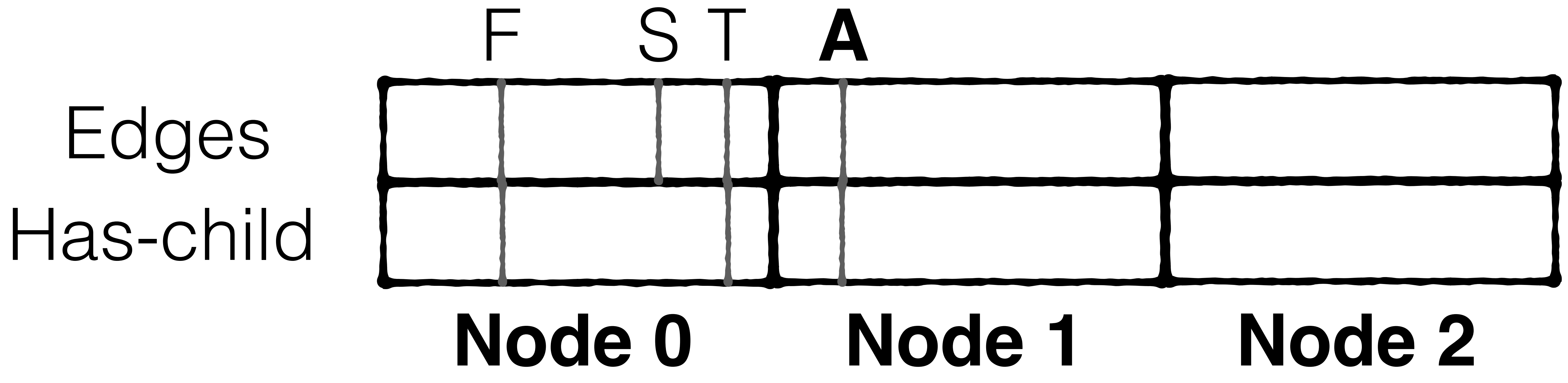

0
F
S
T
1
V1
2
F S T
Edges
Has-child


0
F
S
T
1
V1
2
F
S
T
Edges
Has-child

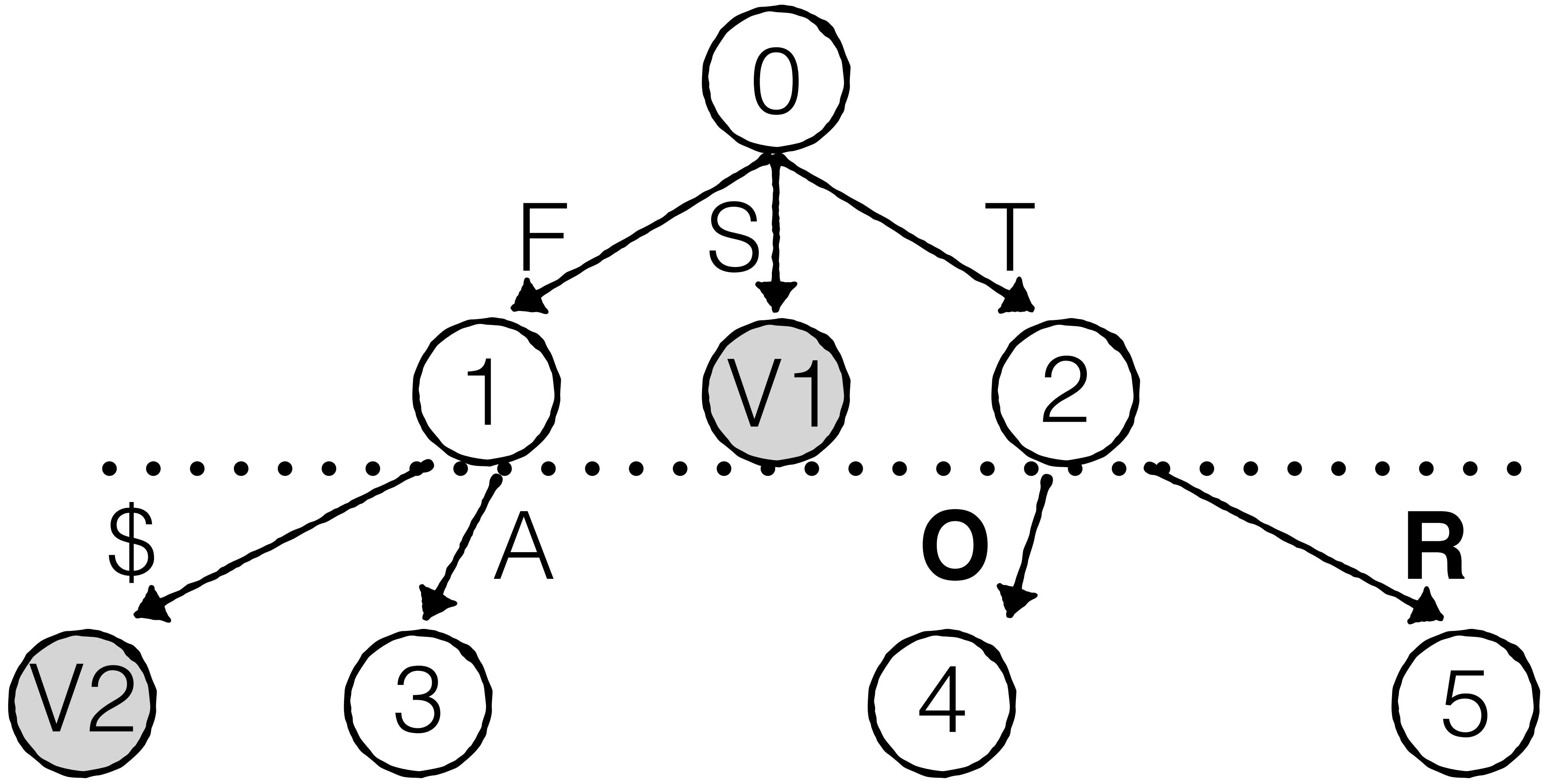


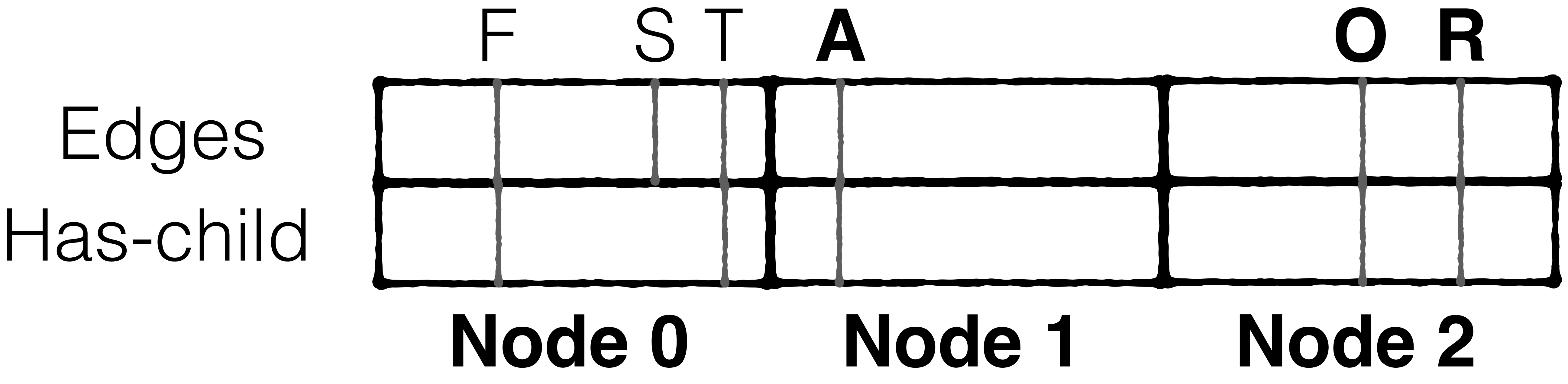
Continue in breadth-first order for each internal node



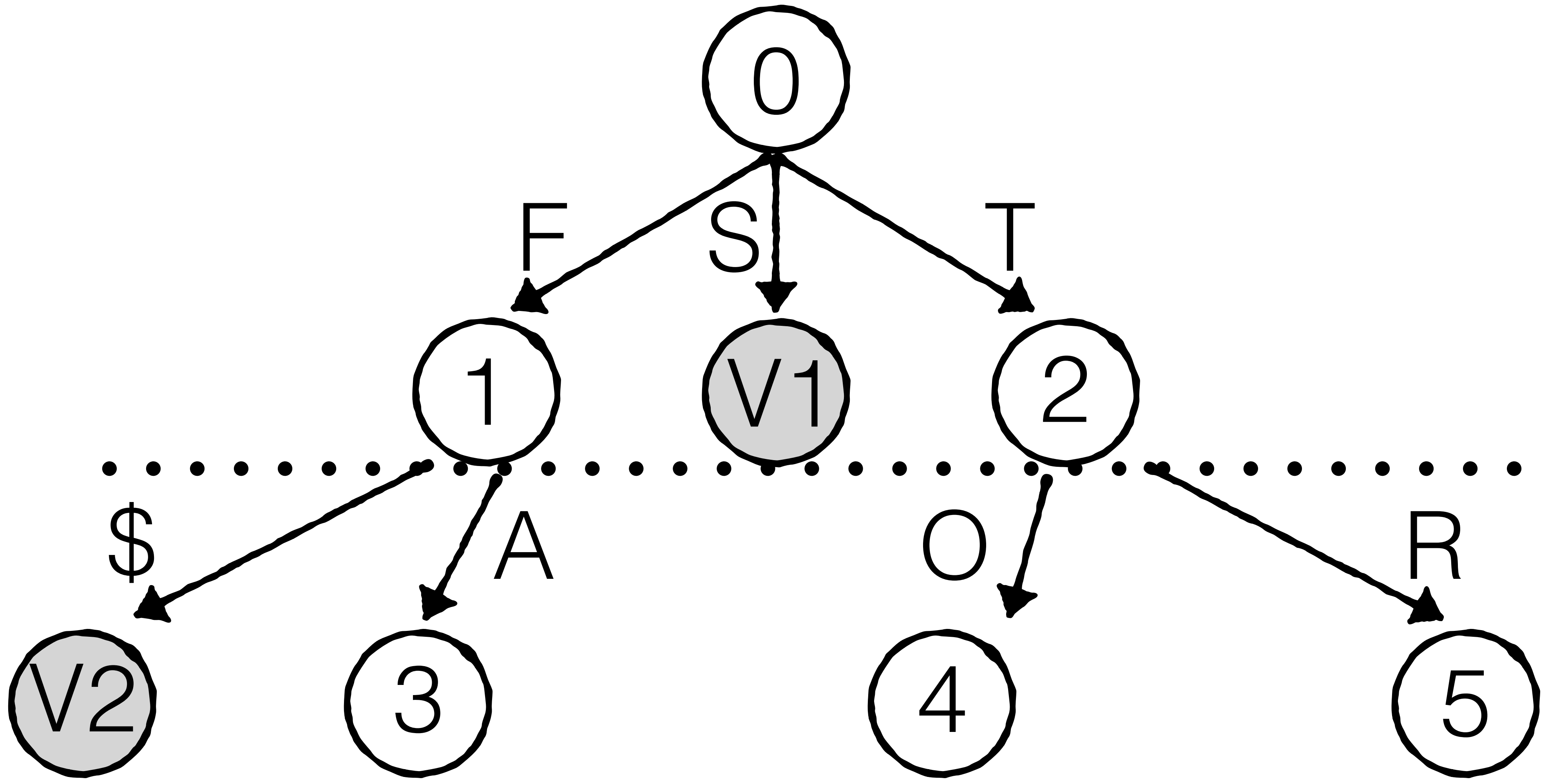


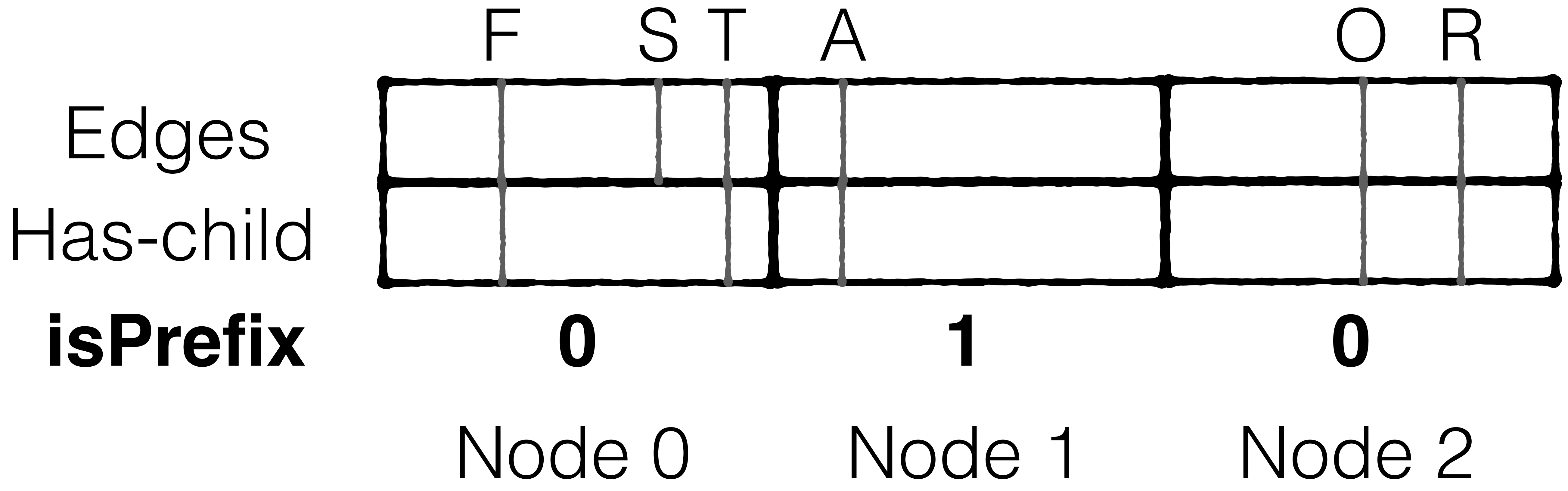
Continue in breadth-first order
for each internal node

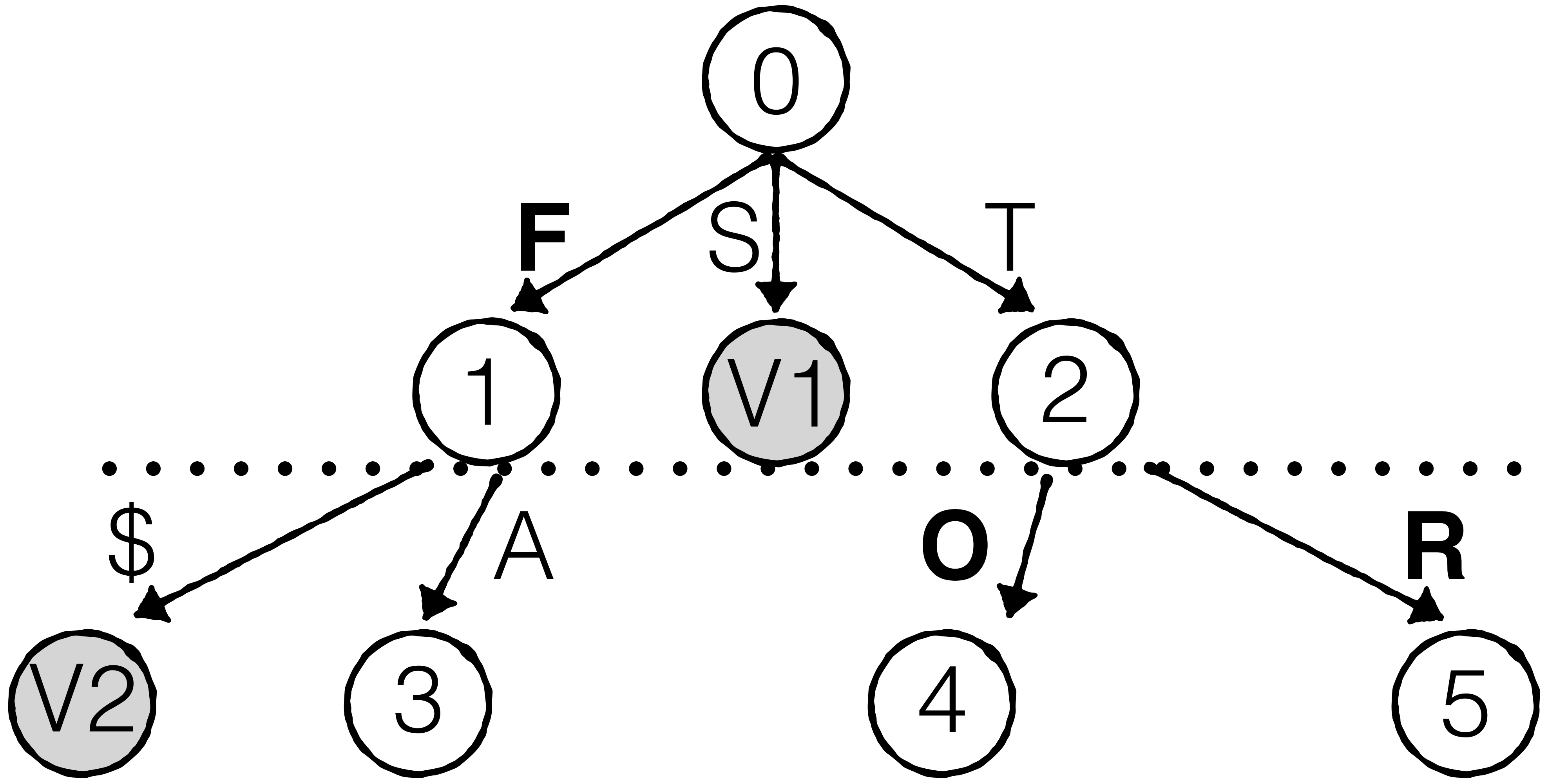


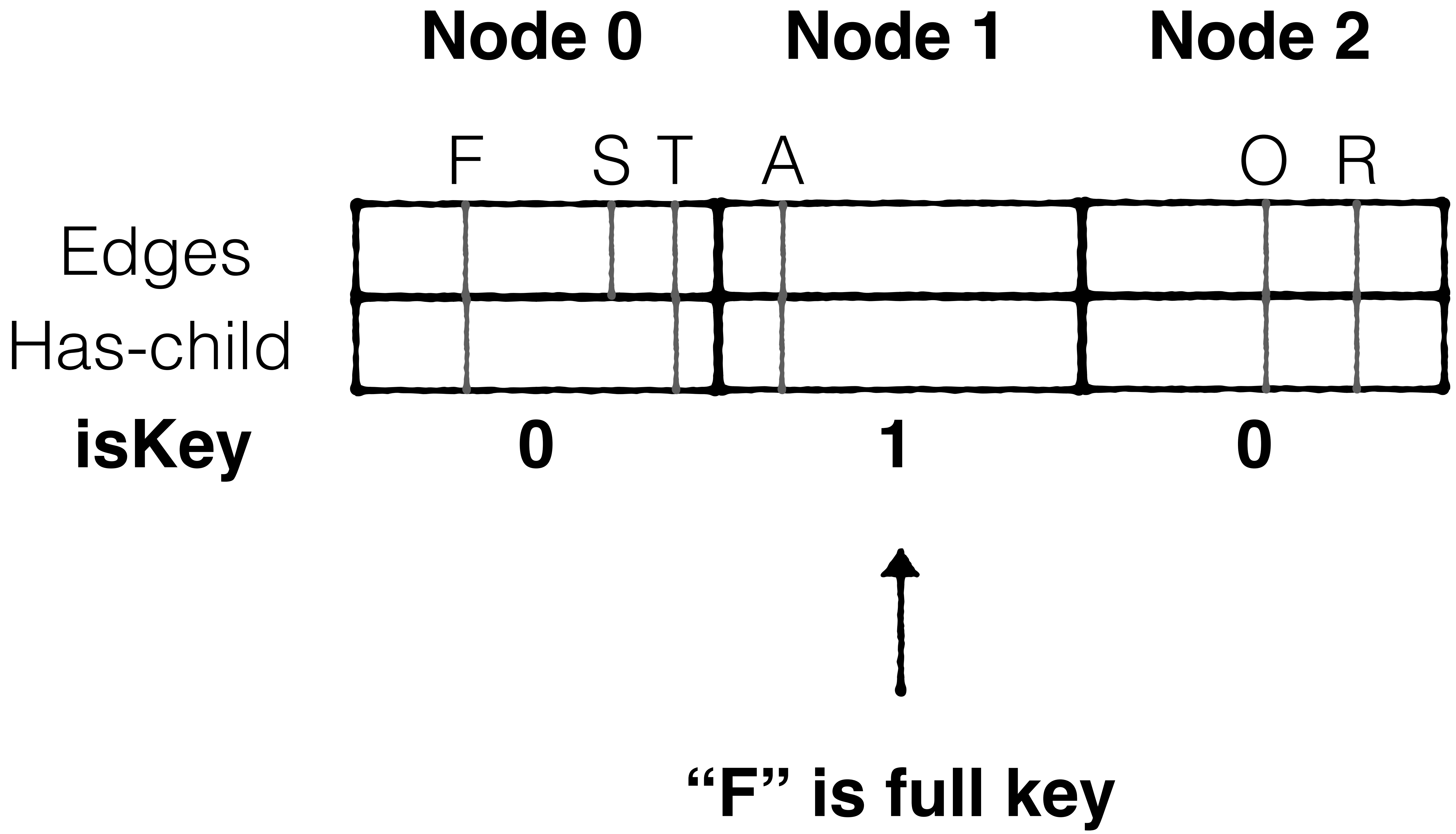


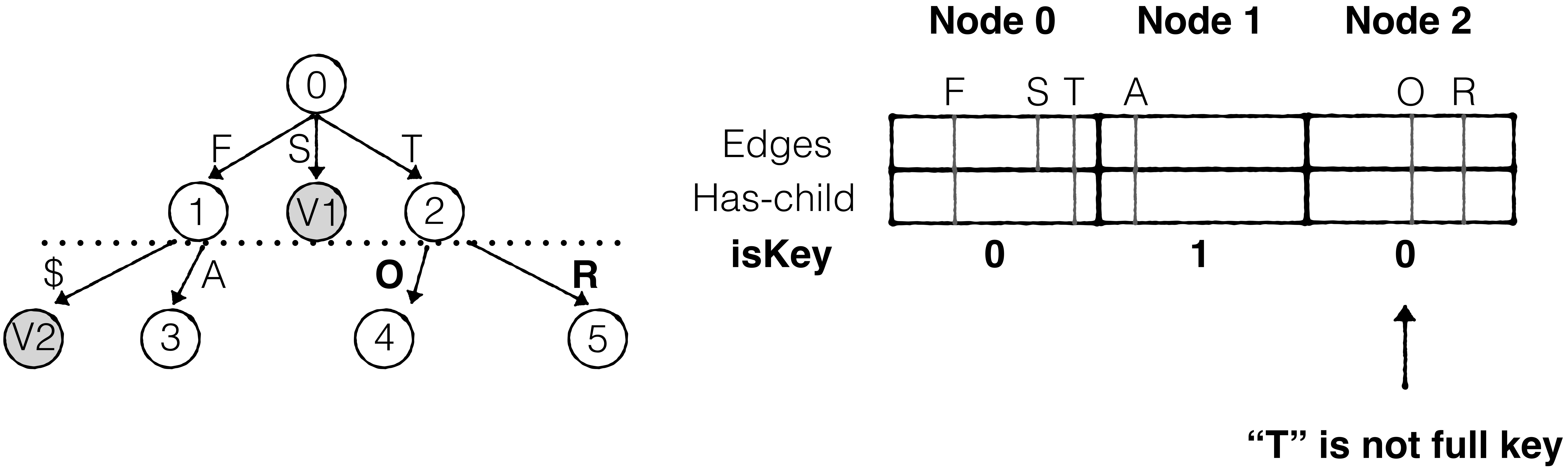
Continue in breadth-first order
for each internal node

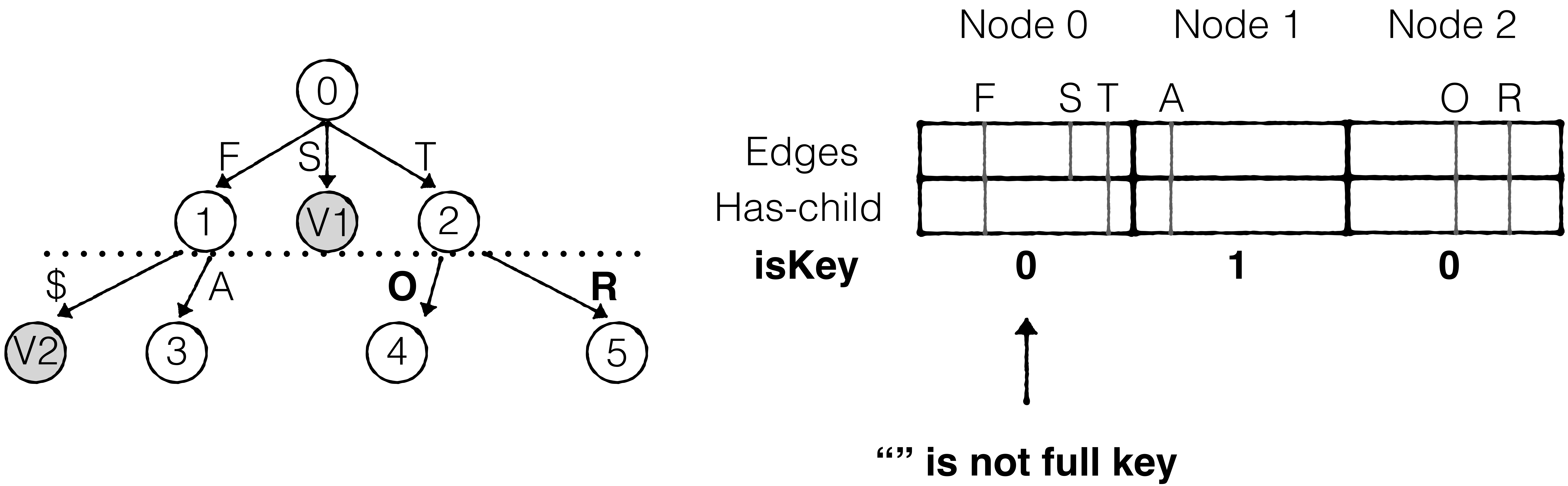

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5


F
S
T
A
O
R
Edges
Has-child
isPrefix
0
1
0
Node 0
Node 1
Node 2

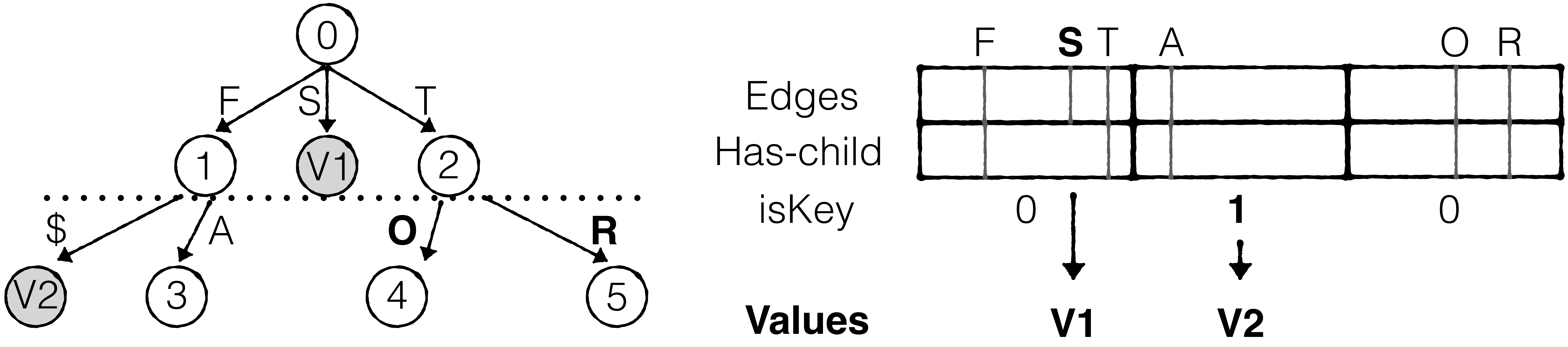

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5

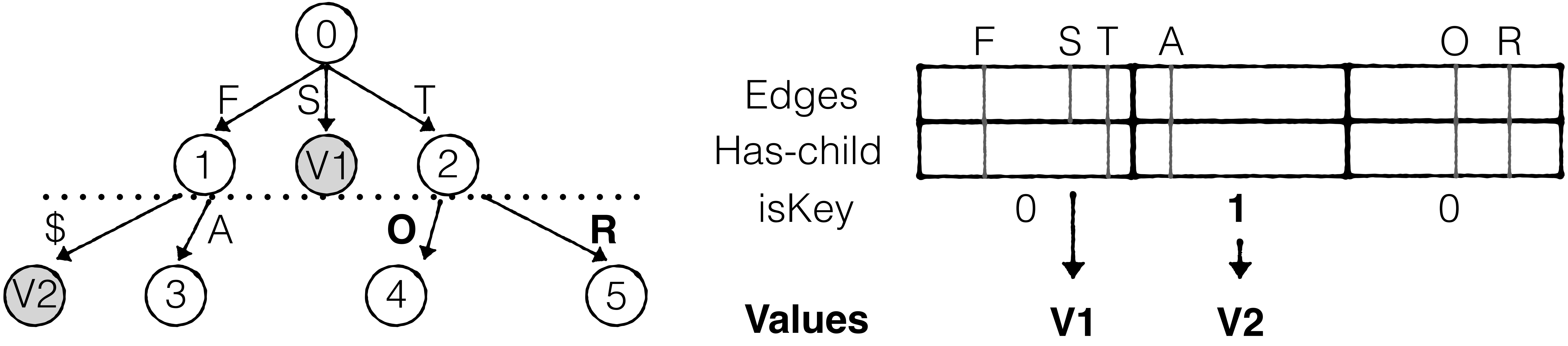

Node 0
Node 1
Node 2
F
S
T
A
O
R
Edges
Has-child
isKey
0
1
0
"F" is full key


0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
Node 0
Node 1
Node 2
F S T A O R
Edges
Has-child
isKey
0
1
0
“T” is not full key



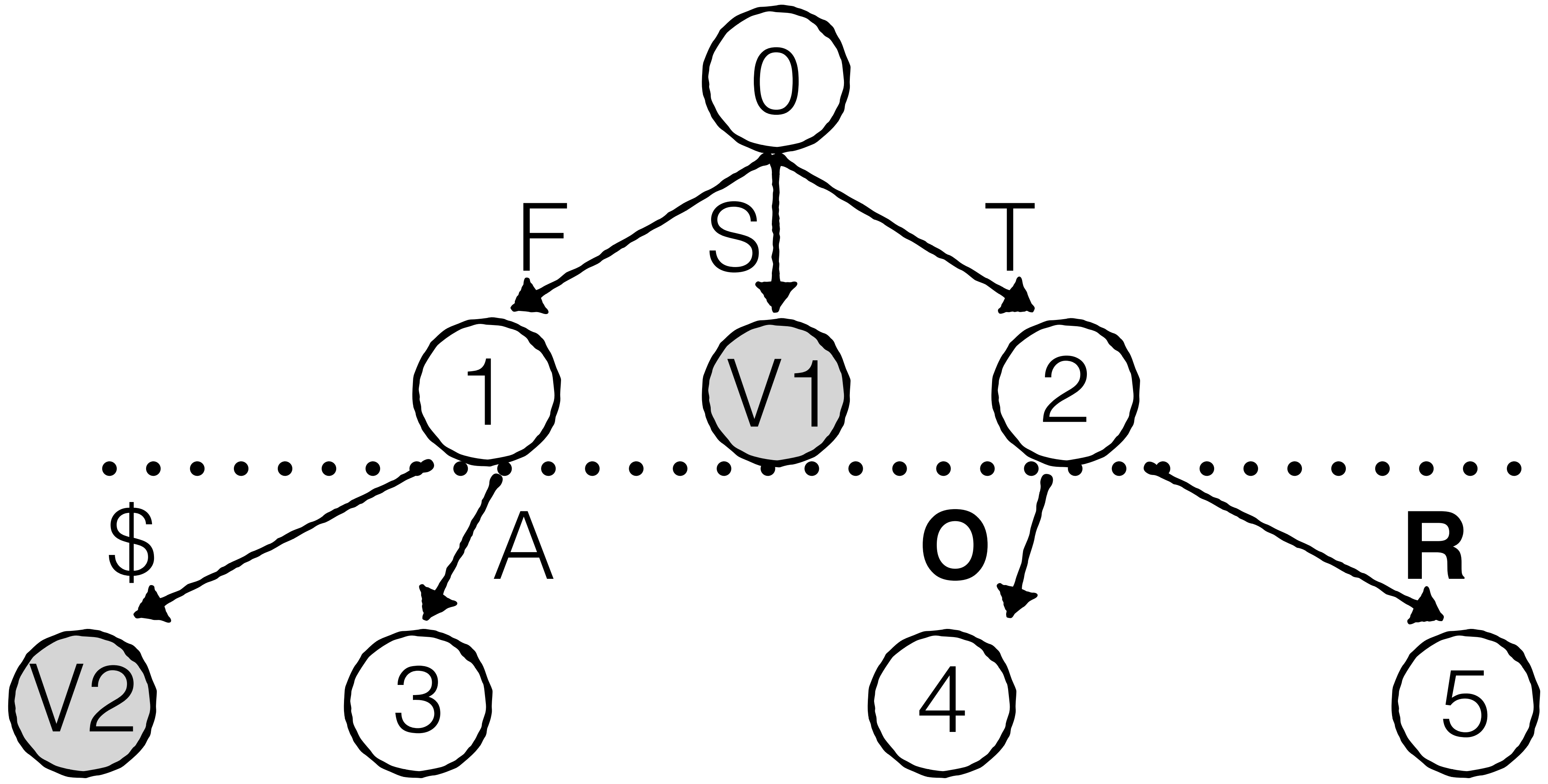
0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
Node 0
Node 1
Node 2
F
S
T
A
O
R
Edges
Has-child
isKey
0
1
0
“” is not full key


0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
F
S
T
A
O
R
Edges
Has-child
isKey
0
1
0
Values
V1
V2

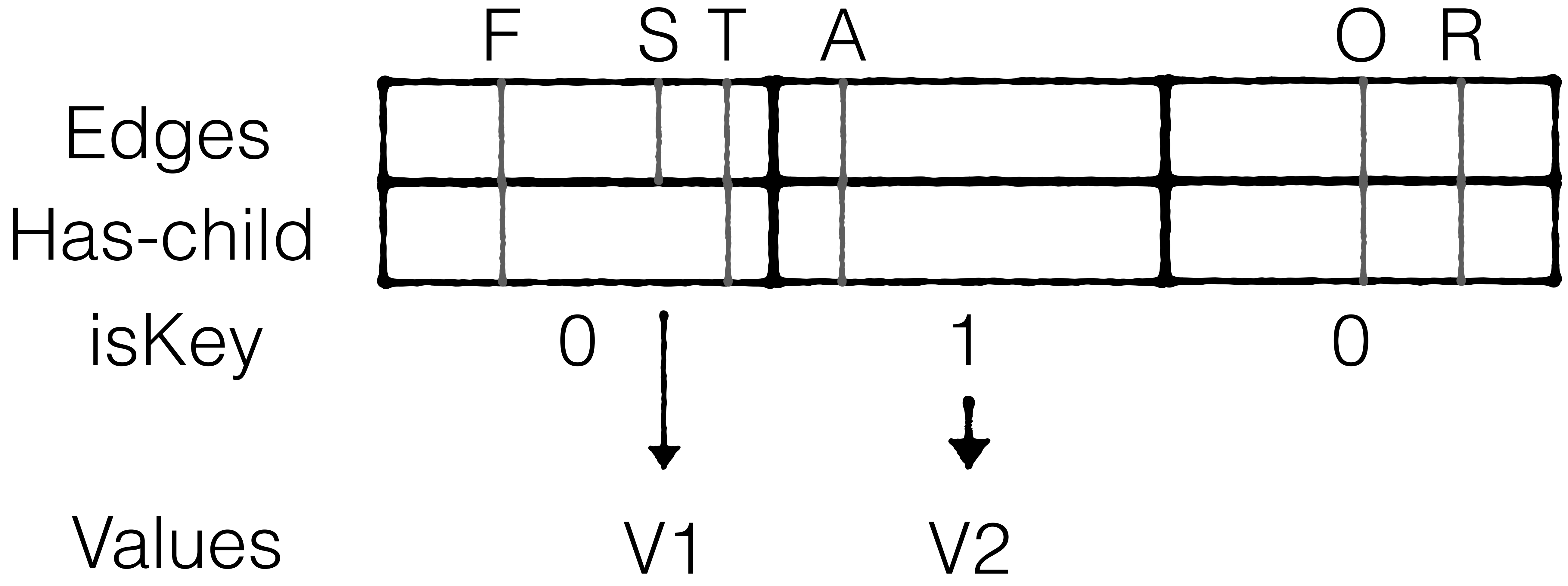


Stored contiguously in an array

Does “S” exist?

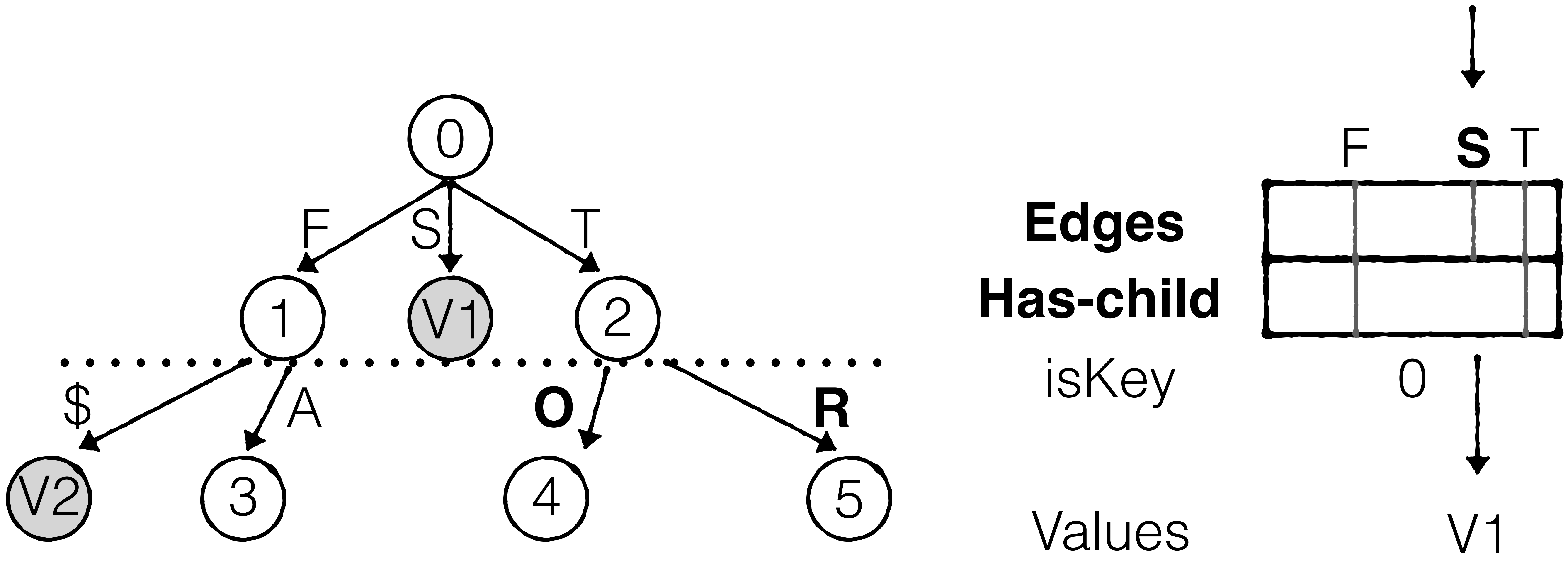


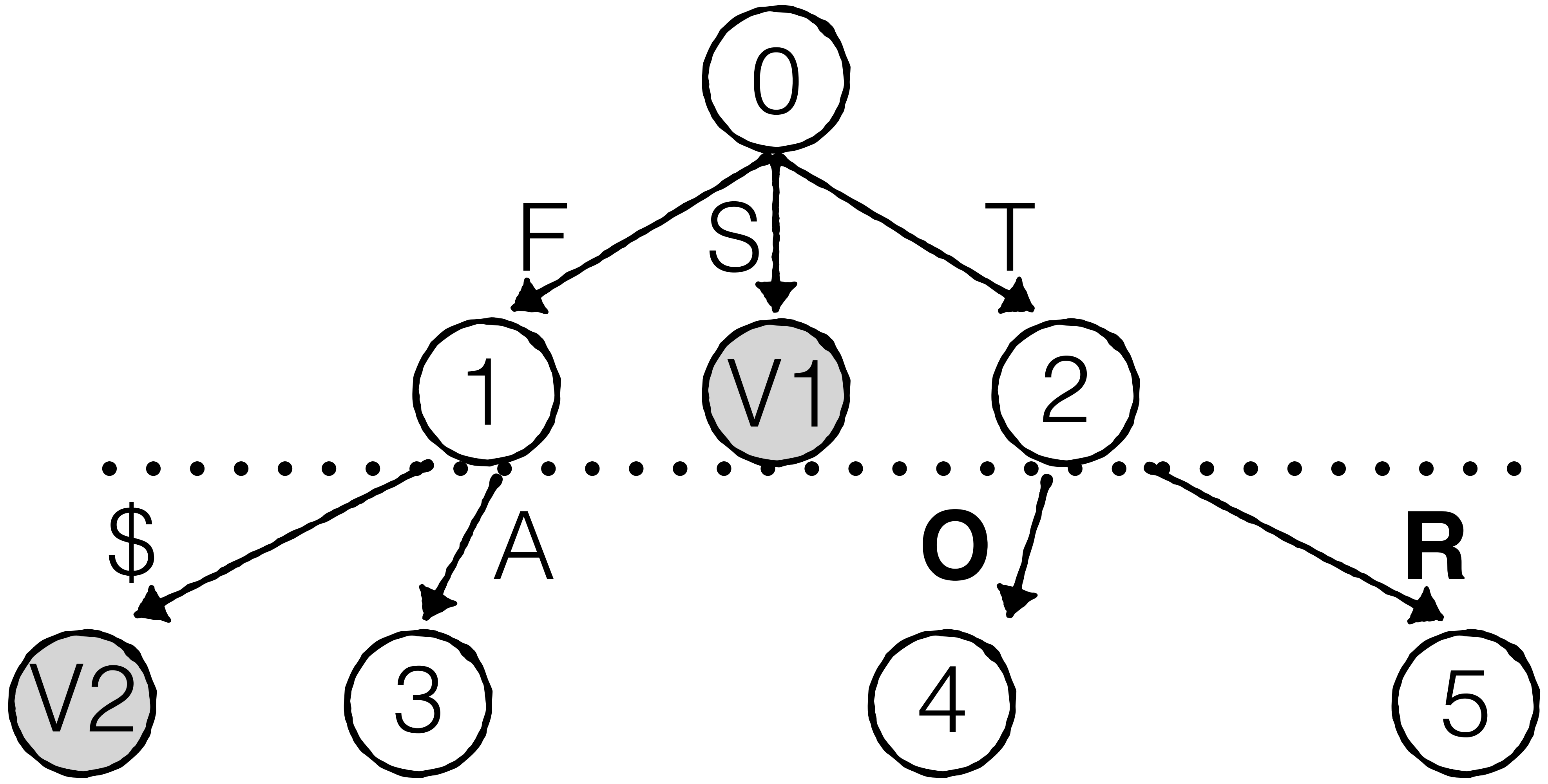
0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5

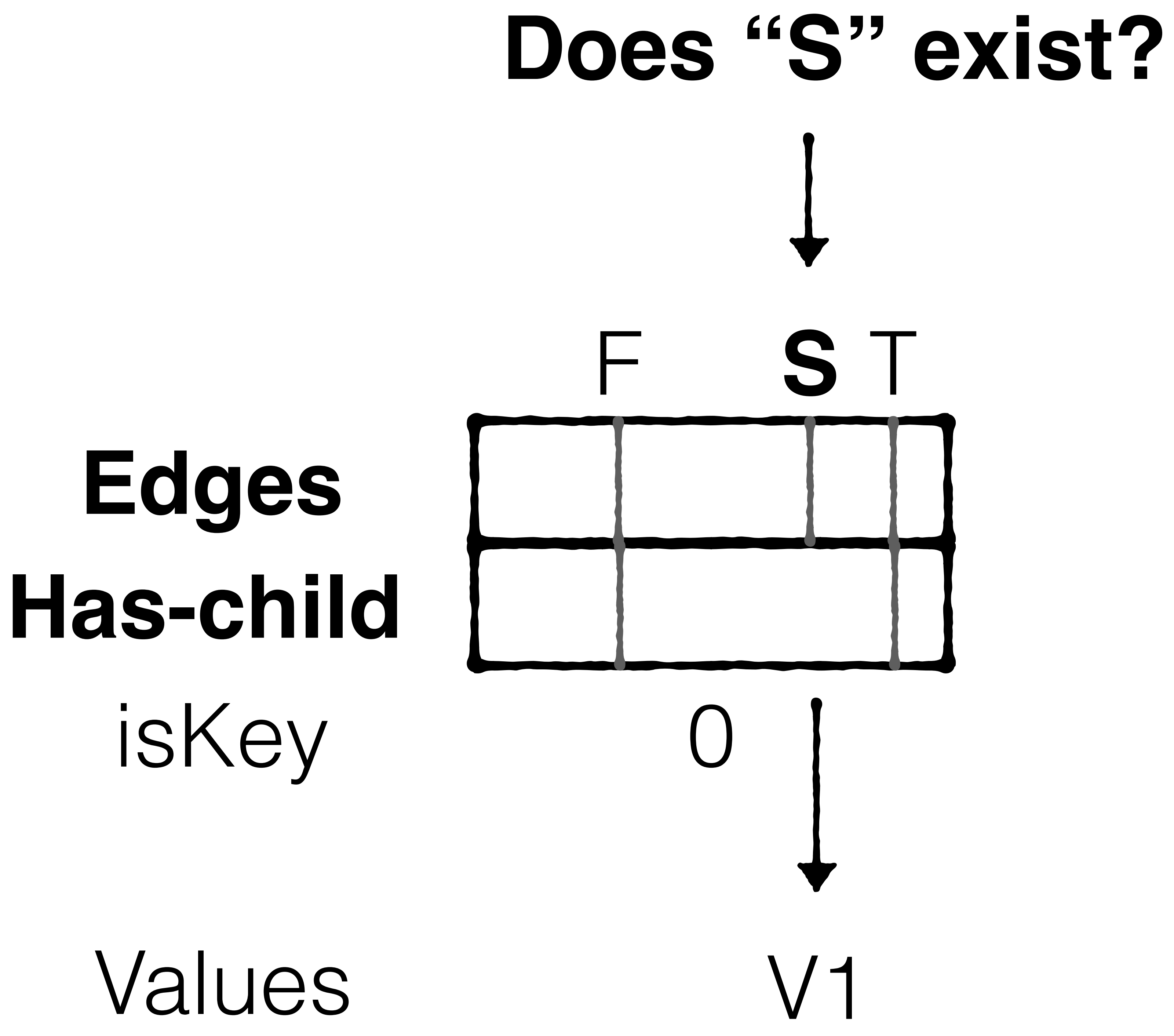


F S T A O R
Edges
Has-child
isKey
0
1
0
Values
V1
V2

Does “S” exist?

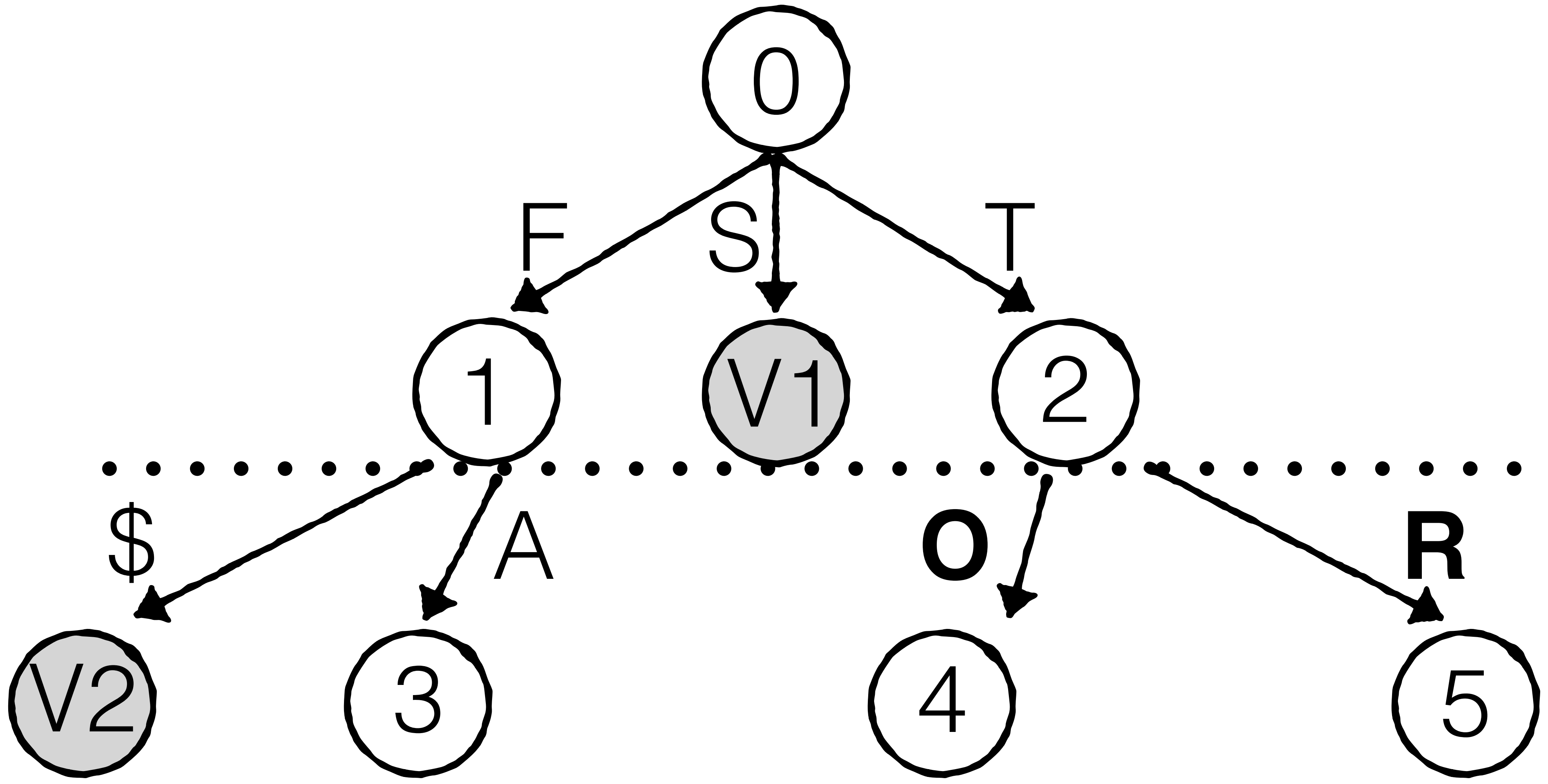


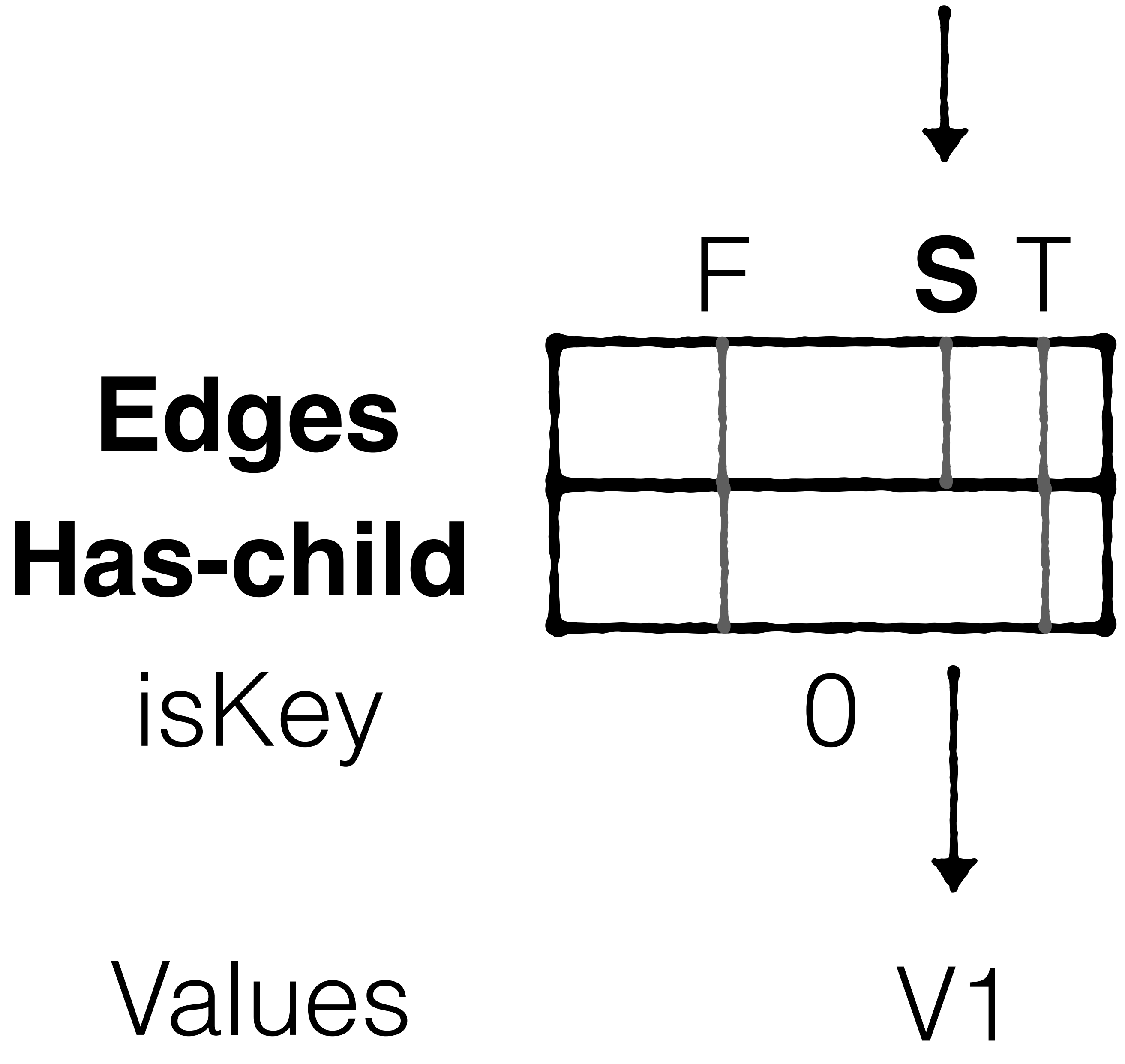




We observe that S exists and has no children, meaning it must be a full key

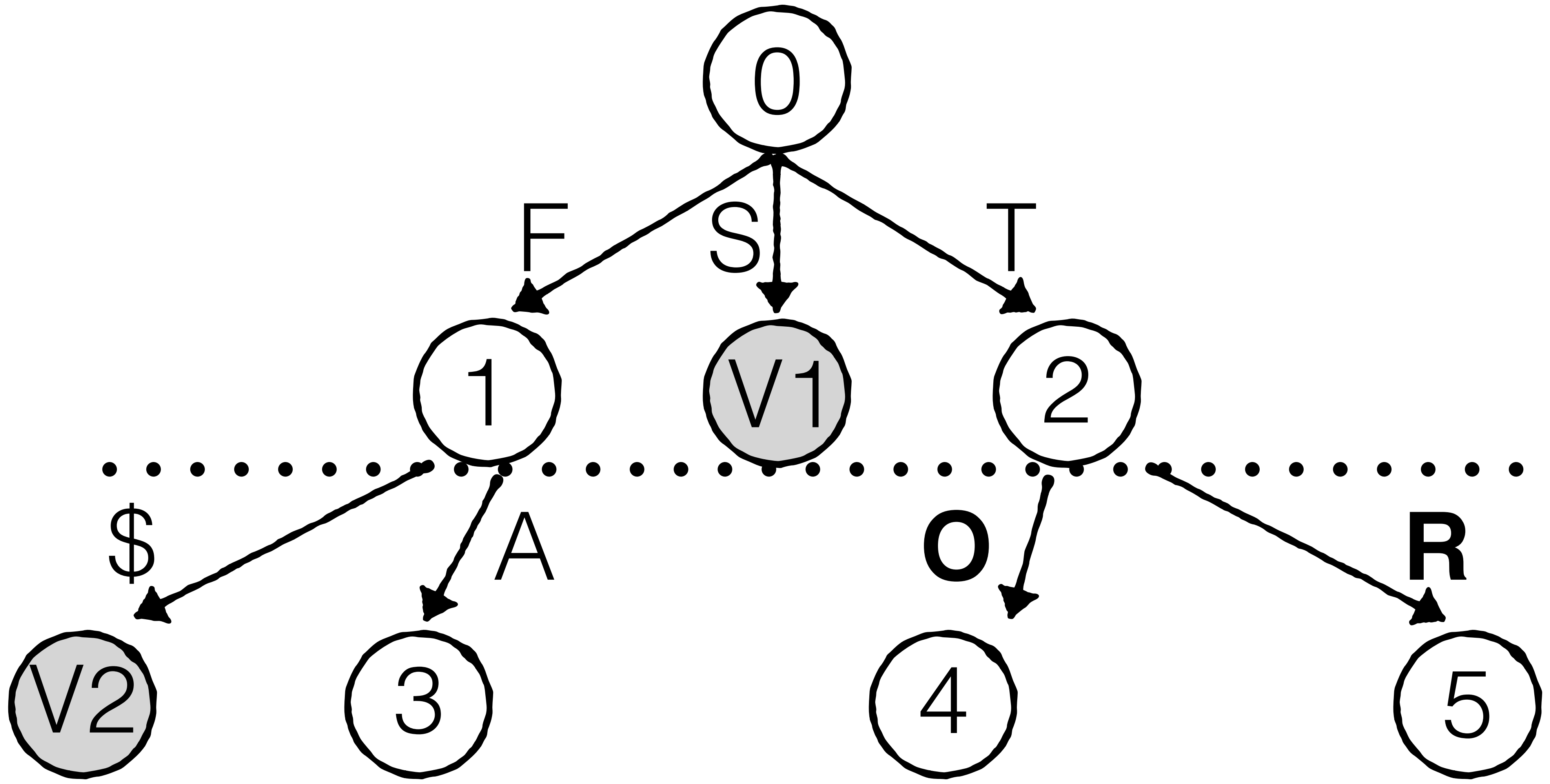
Does “S” exist?

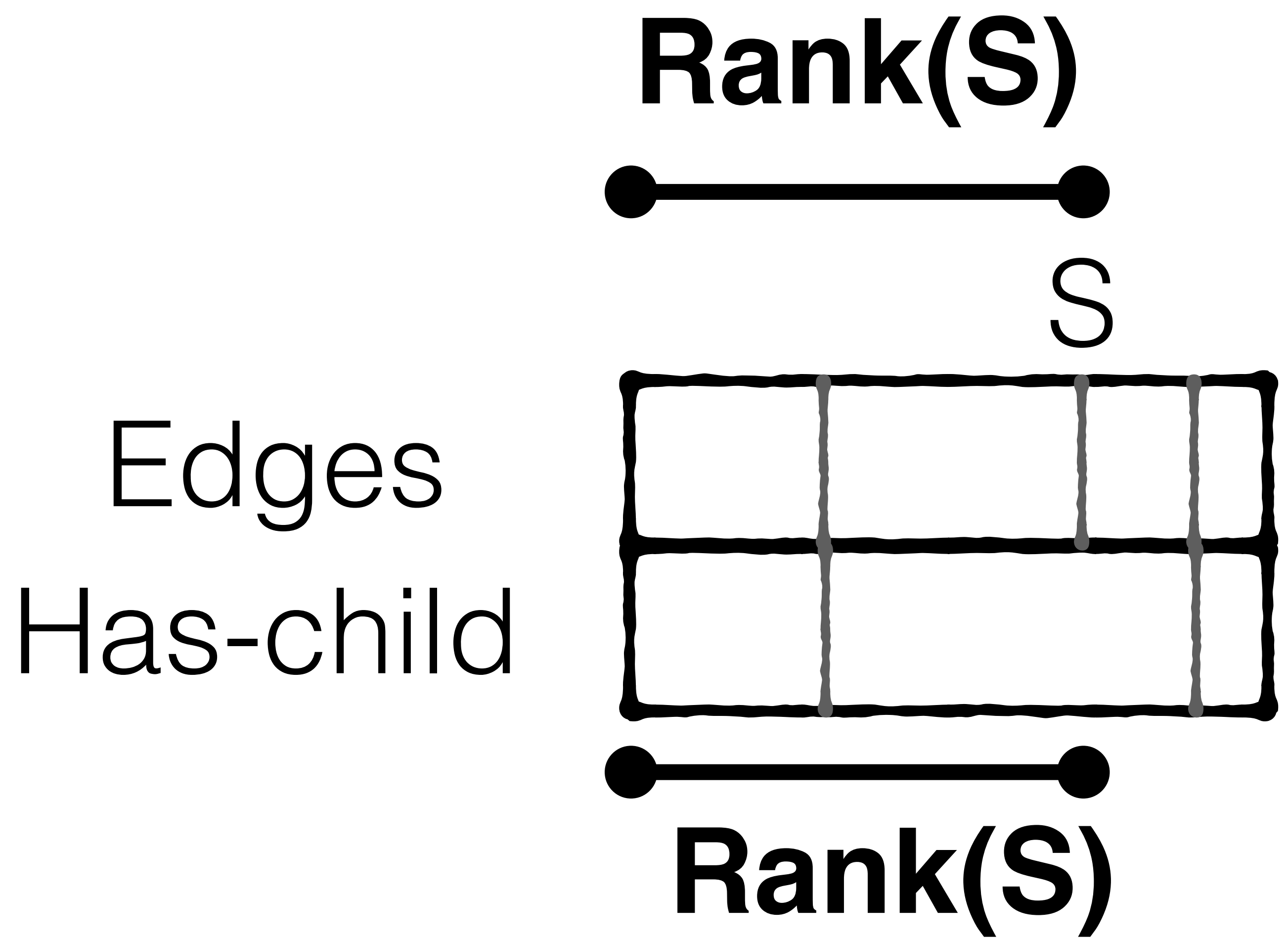


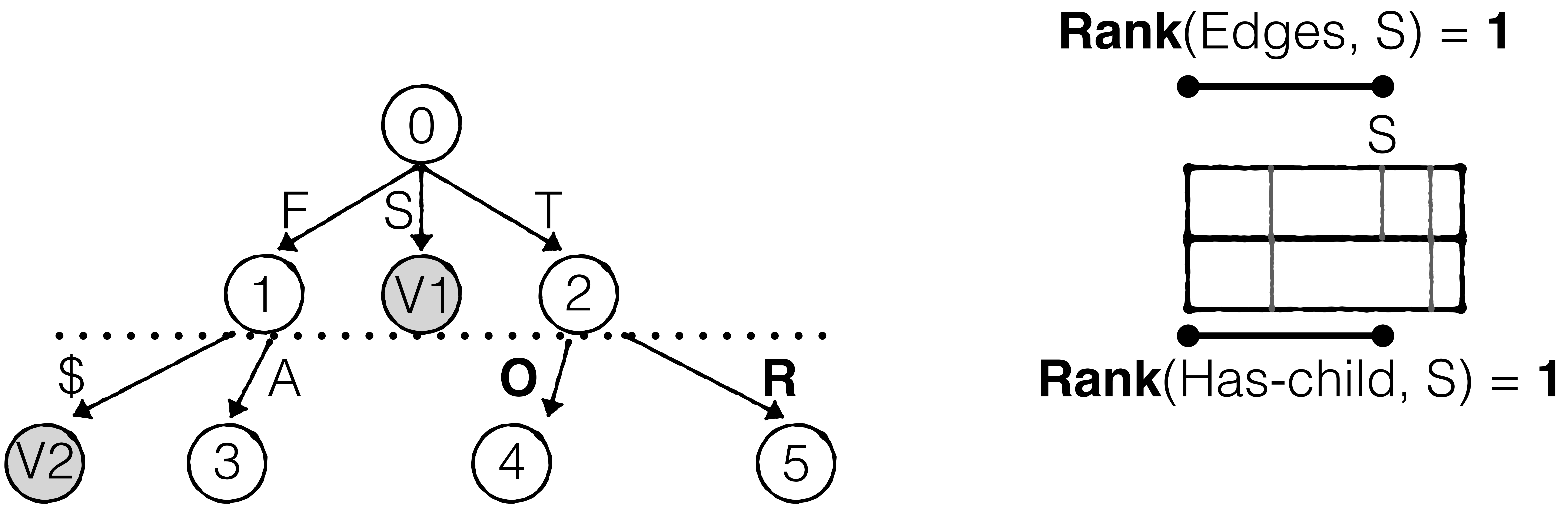


We observe that S exists and has no children, meaning it must be a full key

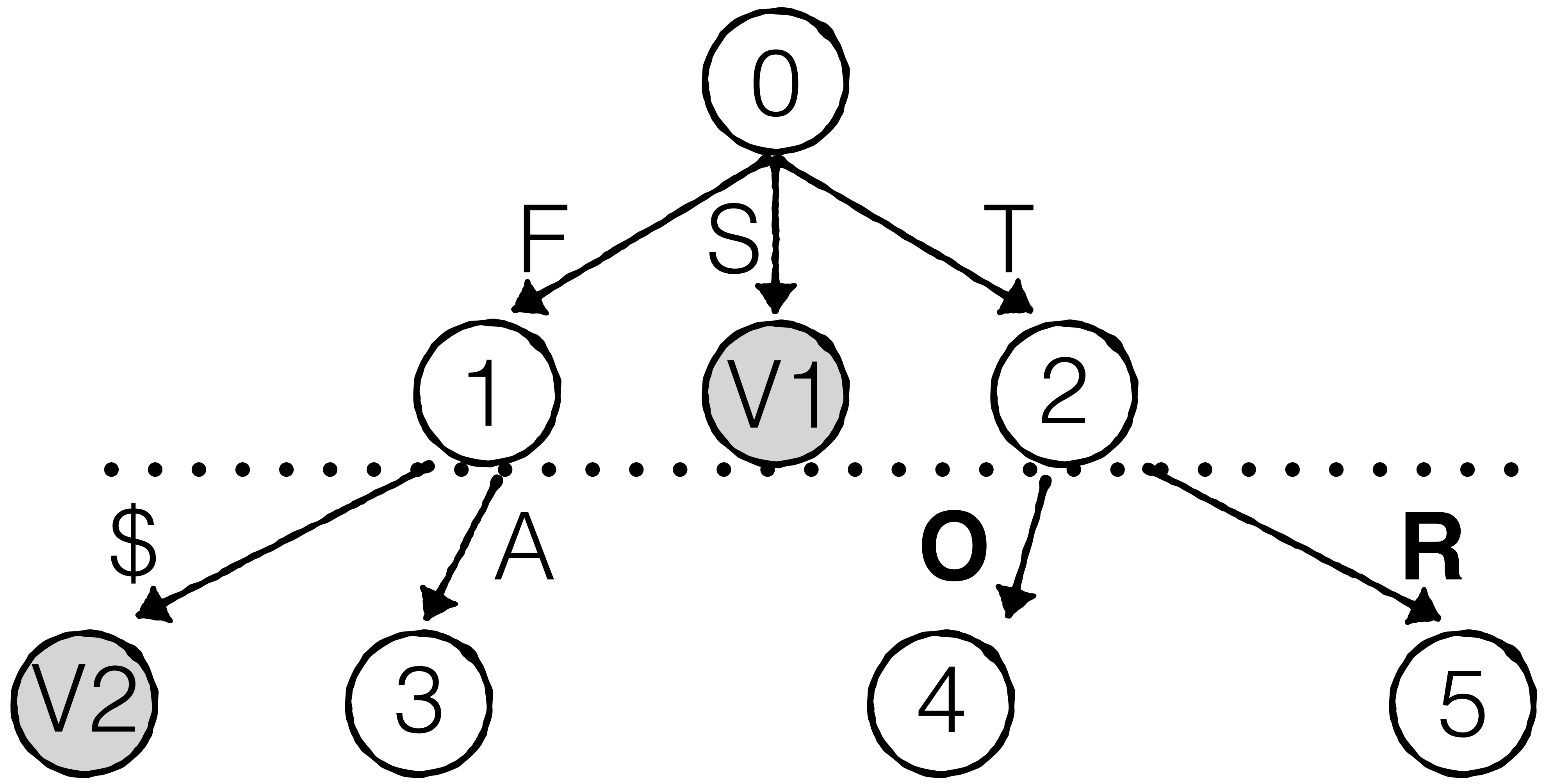
How to locate its value?

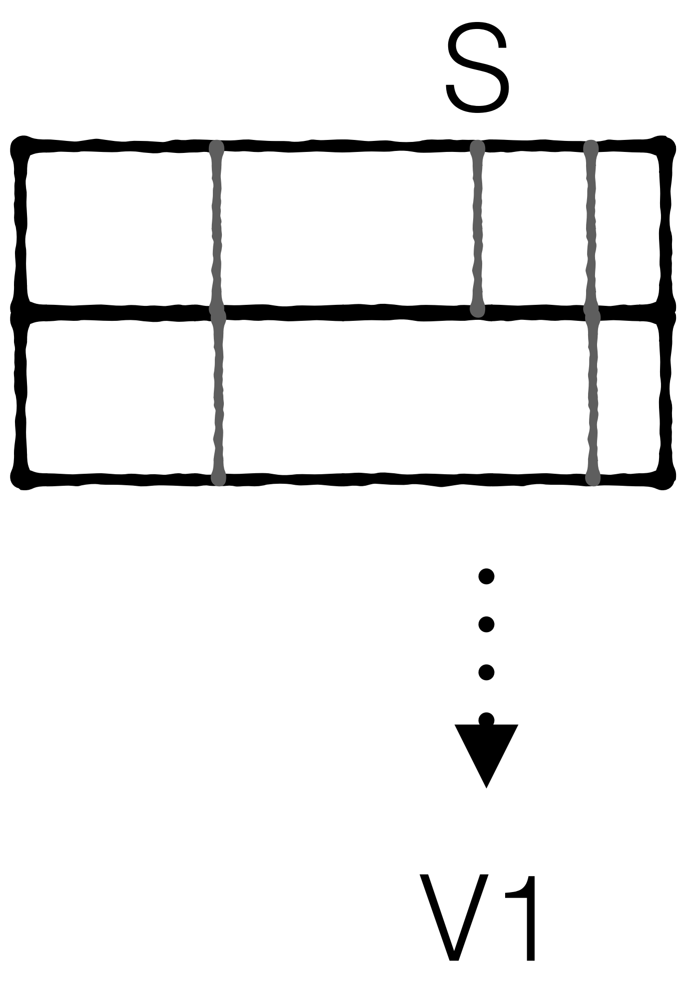

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5


Rank(S)
S
Edges
Has-child
Rank(S)

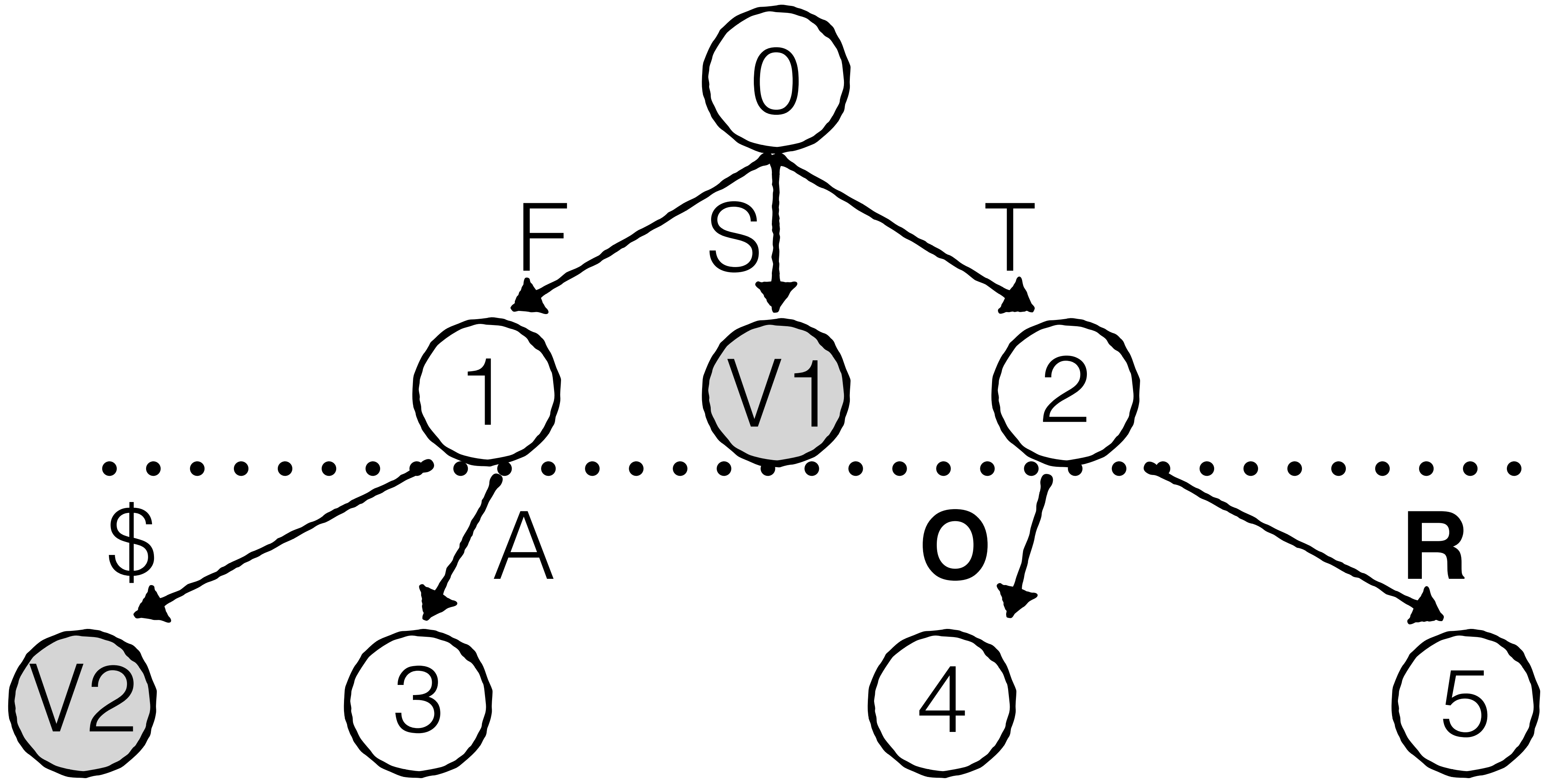

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
Rank(Edges, S) = 1
S
Rank(Has-child, S) = 1

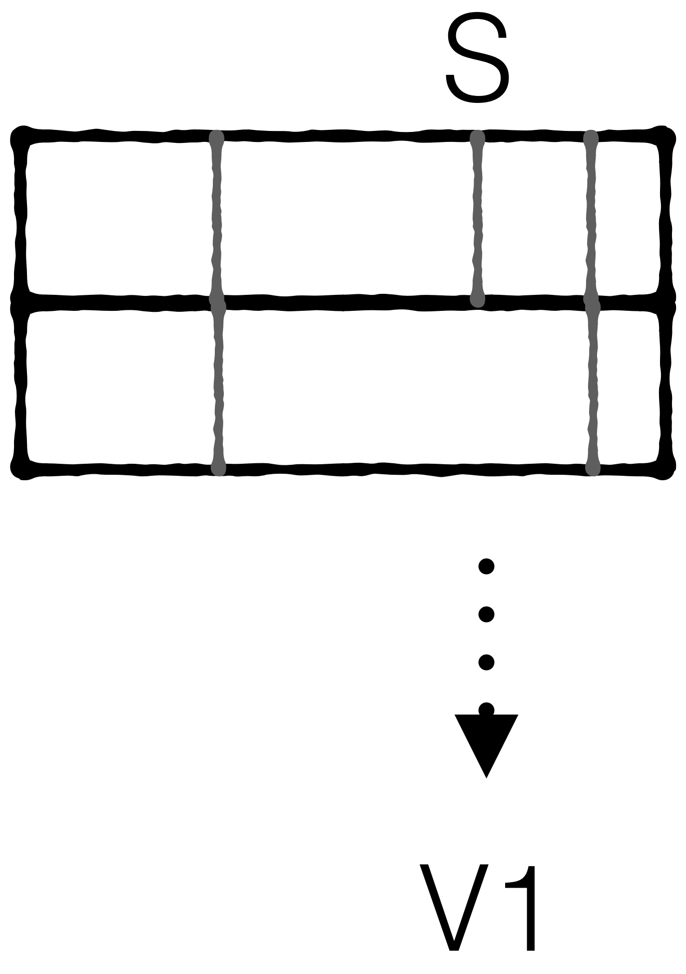
S value offset = **Rank**(Edges, S) - **Rank**(Has-child, S) = **0**





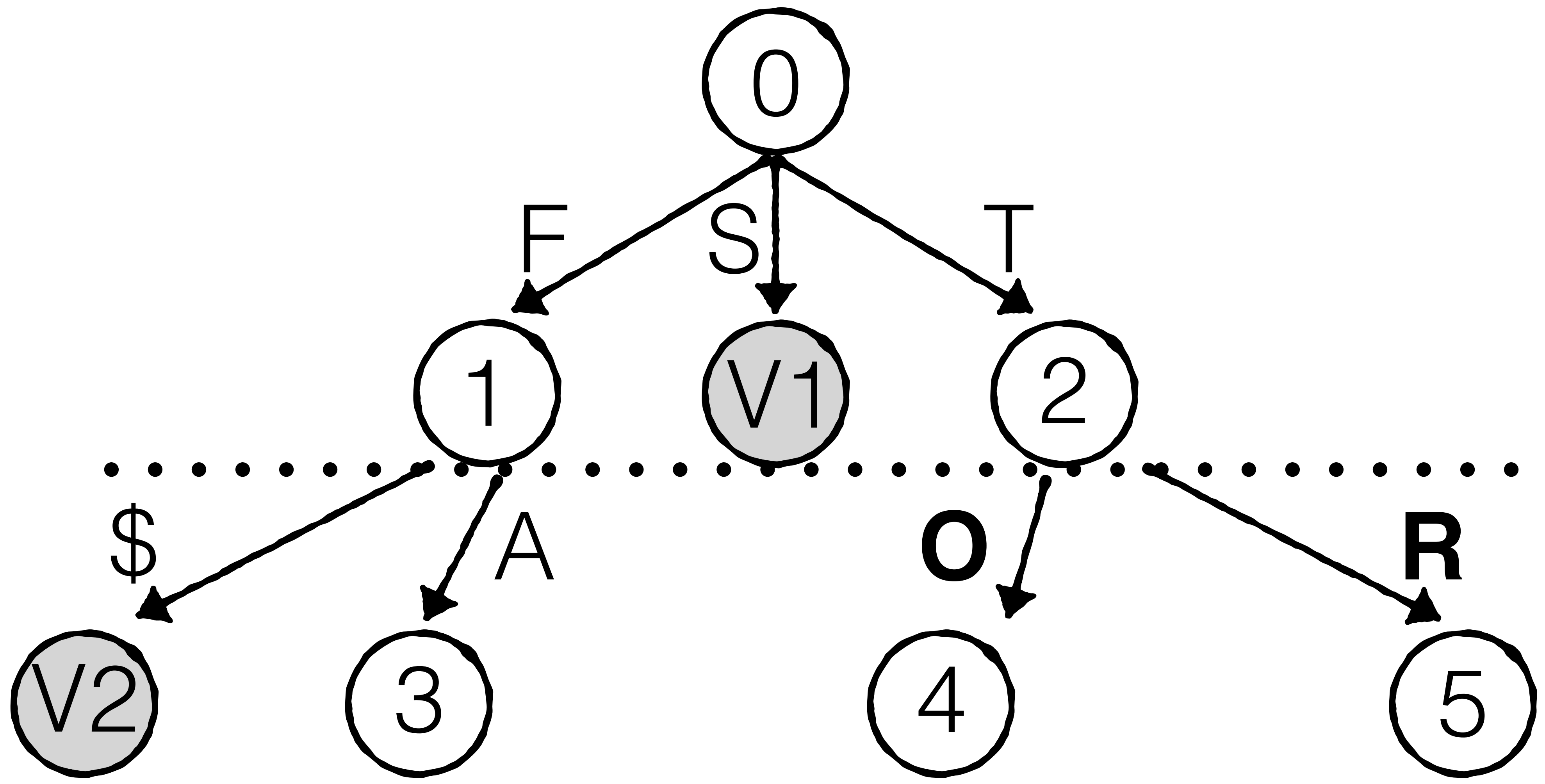
S value offset = Rank(Edges, S) - Rank(Has-child, S) = 0

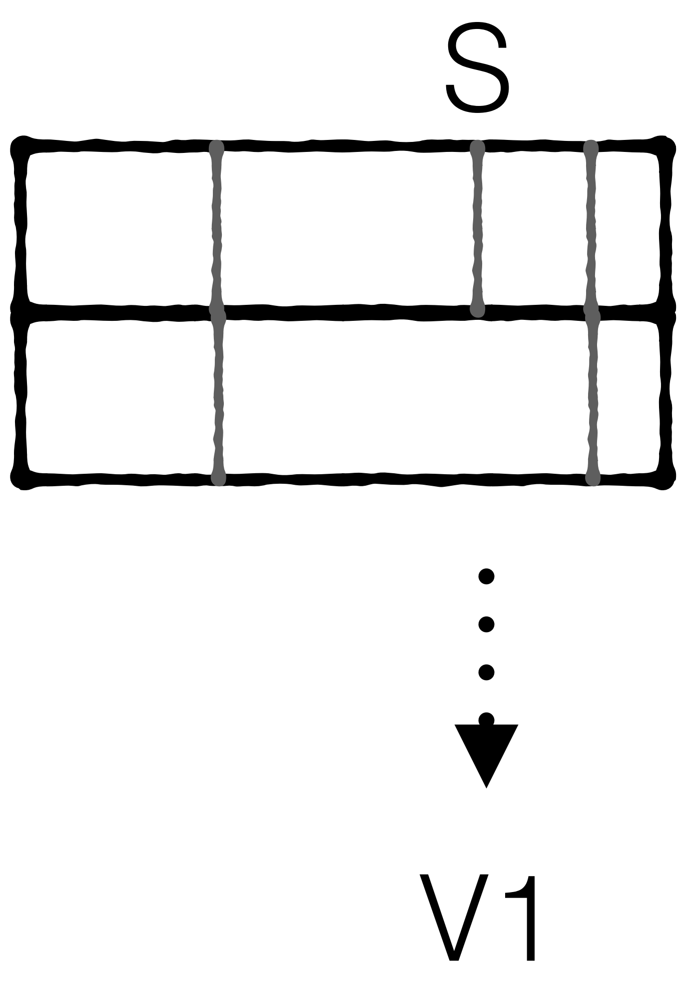




Constant time per each node access :)

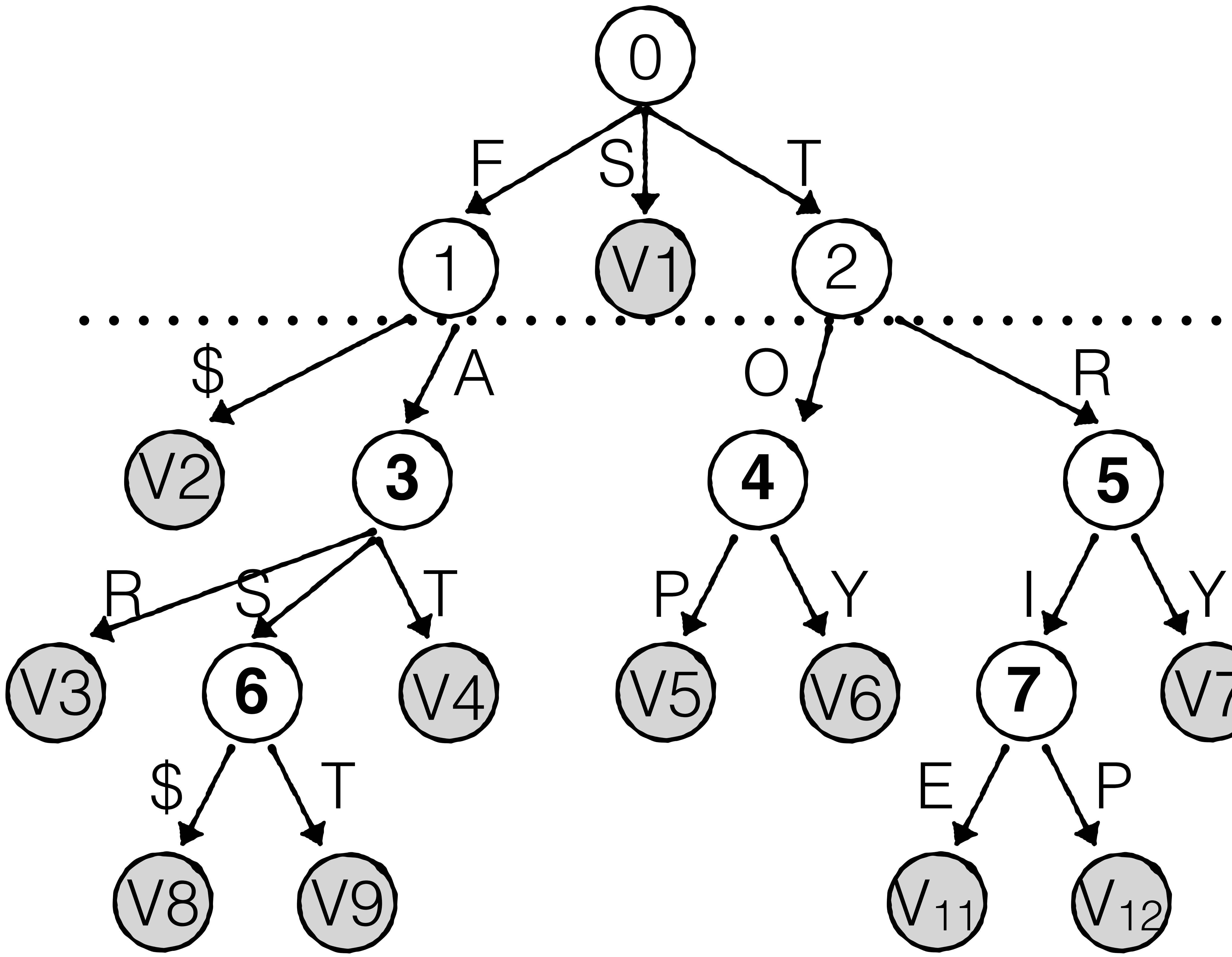
S value offset = Rank(Edges, S) - Rank(Has-child, S) = 0





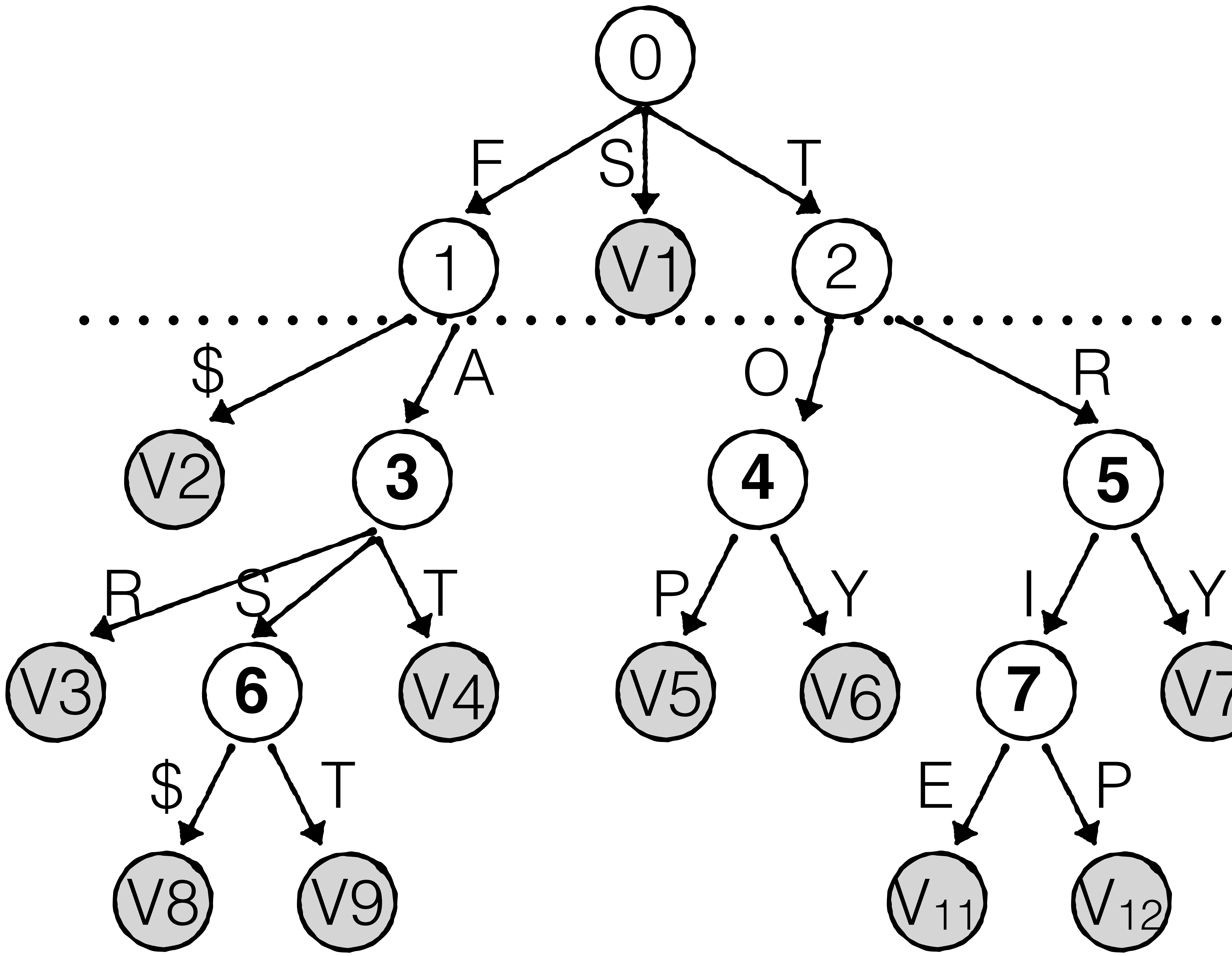
Constant time per each node access :)

Few bits per entry when base nodes have many edges

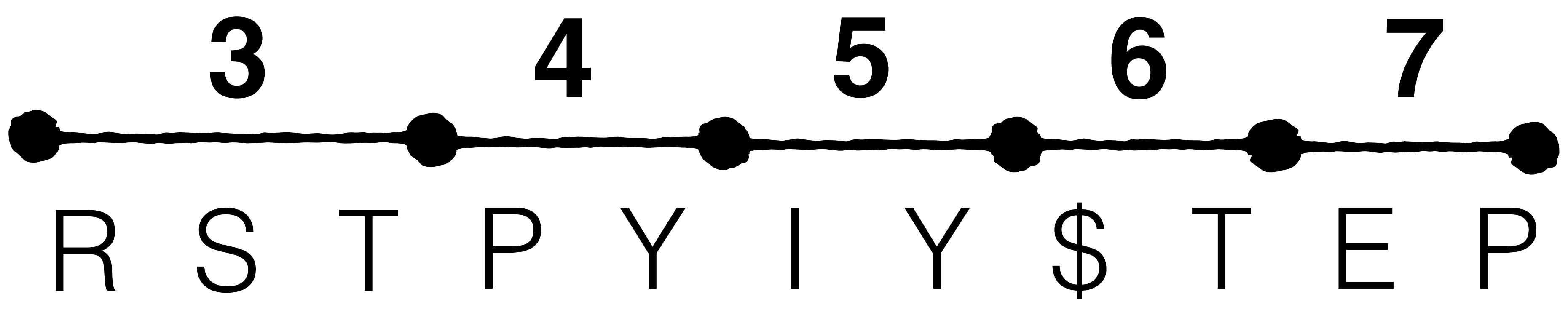


Internal Nodes in breadth first order

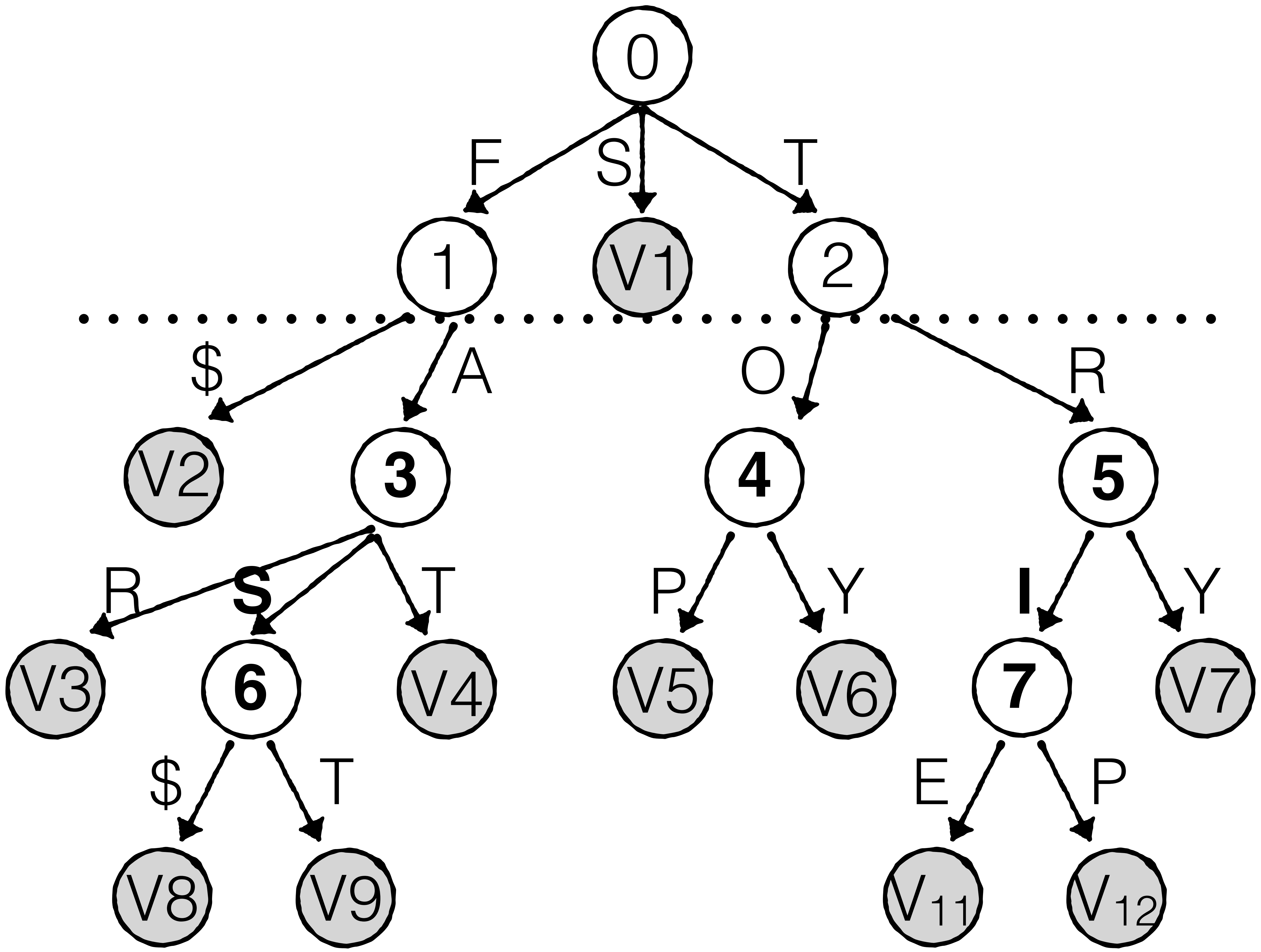
3 4 5 6 7



Internal Nodes in breadth first order

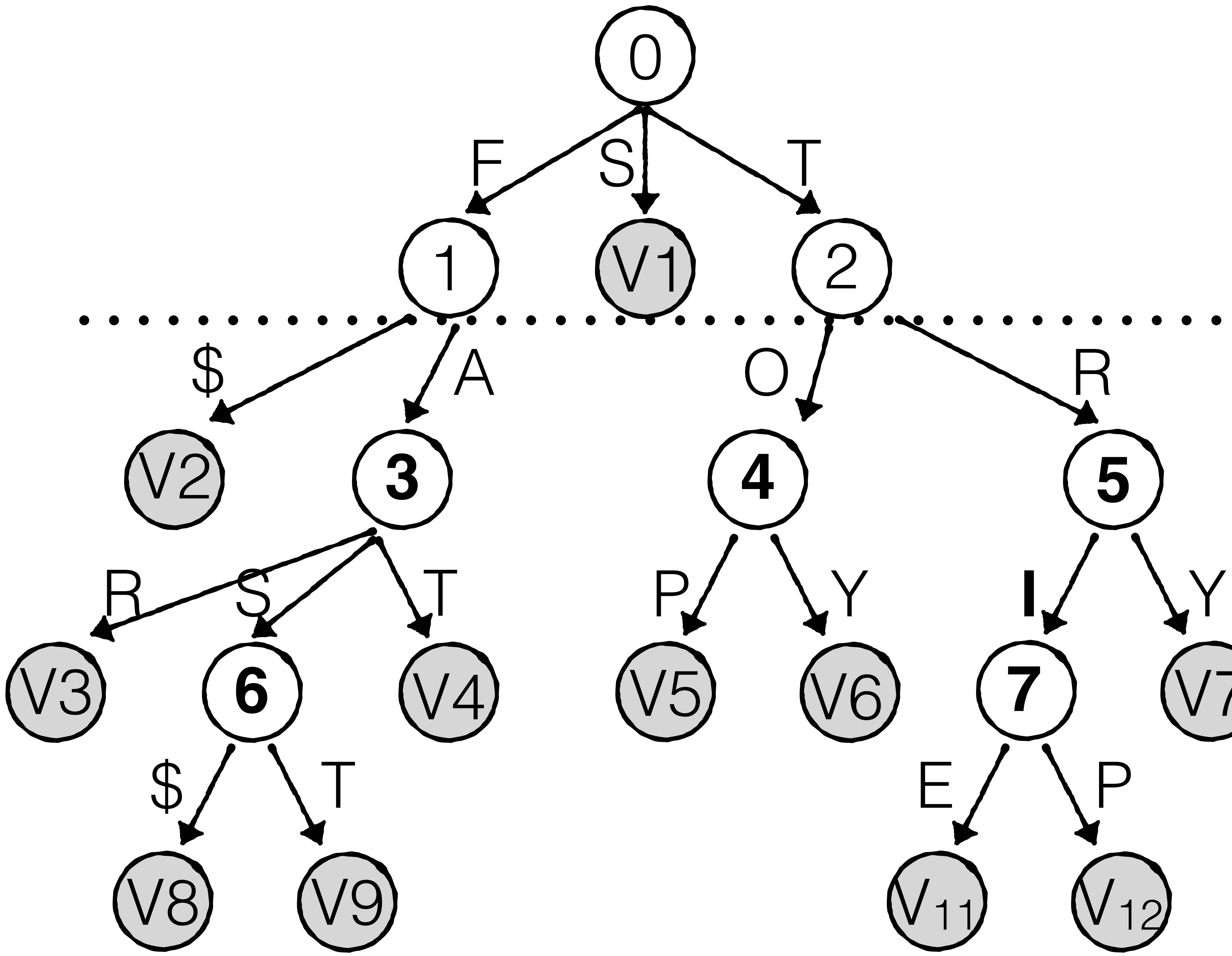


Store children contiguously



Internal Nodes

	3			**4**		**5**		**6**		**7**	
	R	**S**	T	P	Y	**I**	Y	$	T	E	P
Has-child	0	**1**	0	0	0	**1**	0	0	0	0	0

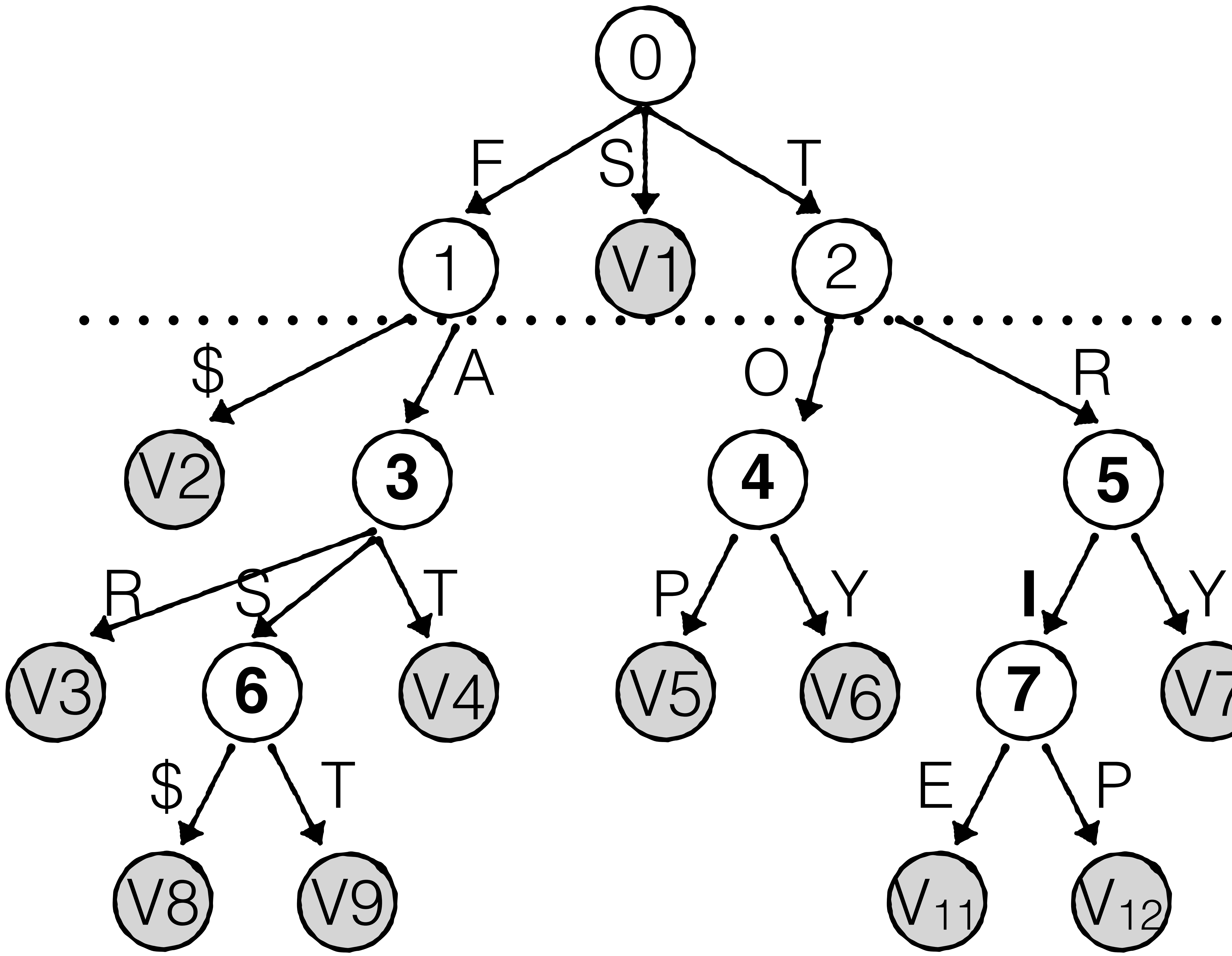


Internal Nodes

	3			4		5		6		7	
	R	S	T	**P**	Y	**I**	Y	**$**	T	**E**	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	0


0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
R
S
T
P
Y
I
Y
V3
6
V4
V5
V6
7
V7
$
T
E
P
V8
V9
V11
V12
Internal Nodes
3 4 5 6 7
R S T P Y I Y $ T E P
Has-child 0 1 0 0 0 1 0 0 0 0 0
Start 1 0 0 1 0 1 0 1 0 1 0
V3 V4 V5 V6 V7 V8 V9 V10 V11

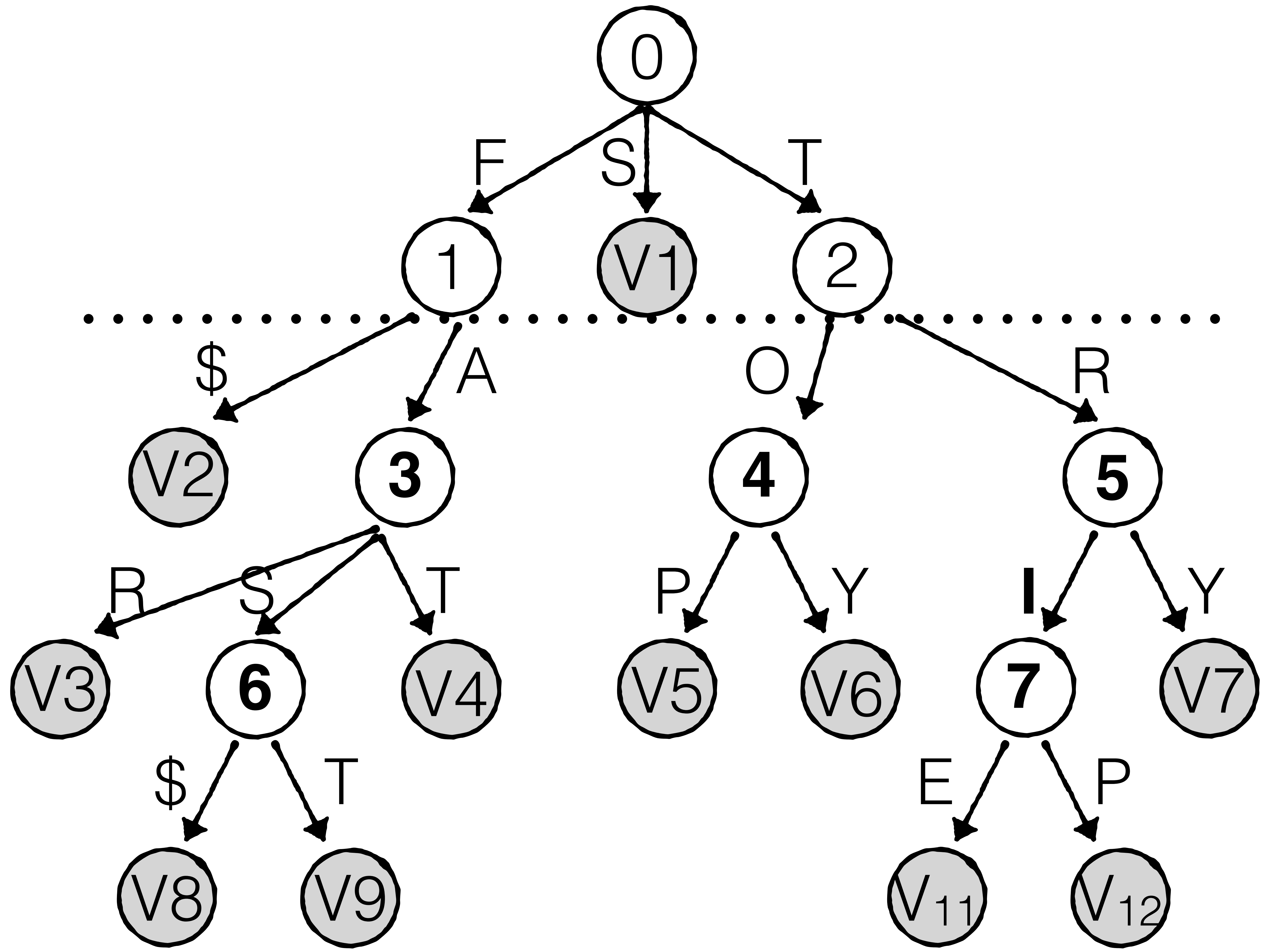

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
R
S
T
P
Y
I
Y
V3
6
V4
V5
V6
7
V7
$
T
E
P
V8
V9
V11
V12
Internal Nodes
3 4 5 6 7
R S T P Y I Y $ T E P
Has-child 0 1 0 0 0 1 0 0 0 0 0
Start 1 0 0 1 0 1 0 1 0 1 0
V3 V4 V5 V6 V7 V8 V9 V10 V11
Stored Contiguously

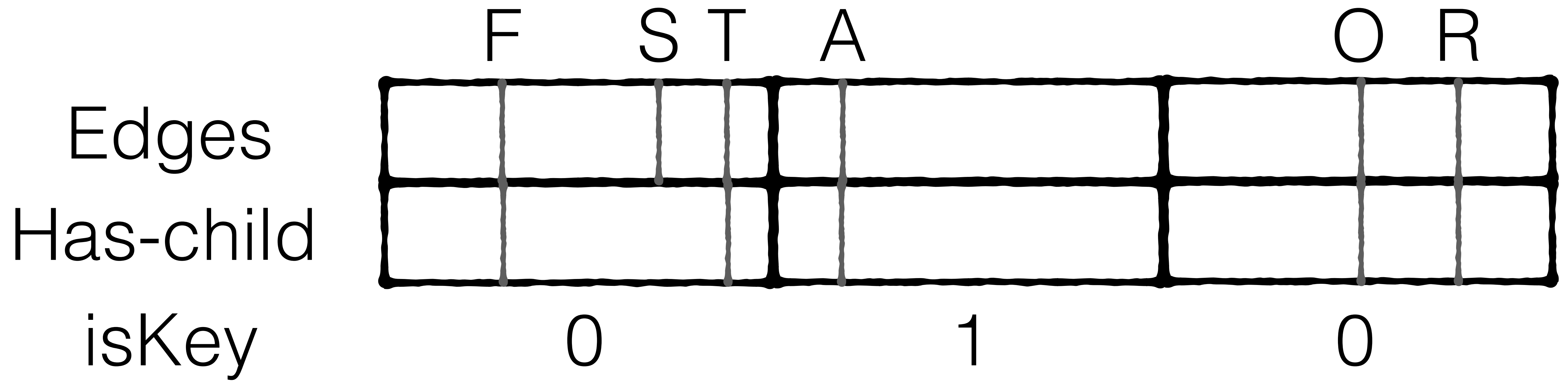


Internal Nodes

	3			4		5		6		7	
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0
	V_3		**V_4**	**V_5**	**V_6**		**V_7**	**V_8**	**V_9**	**V_{10}**	**V_{11}**

Each leaf connected to payload

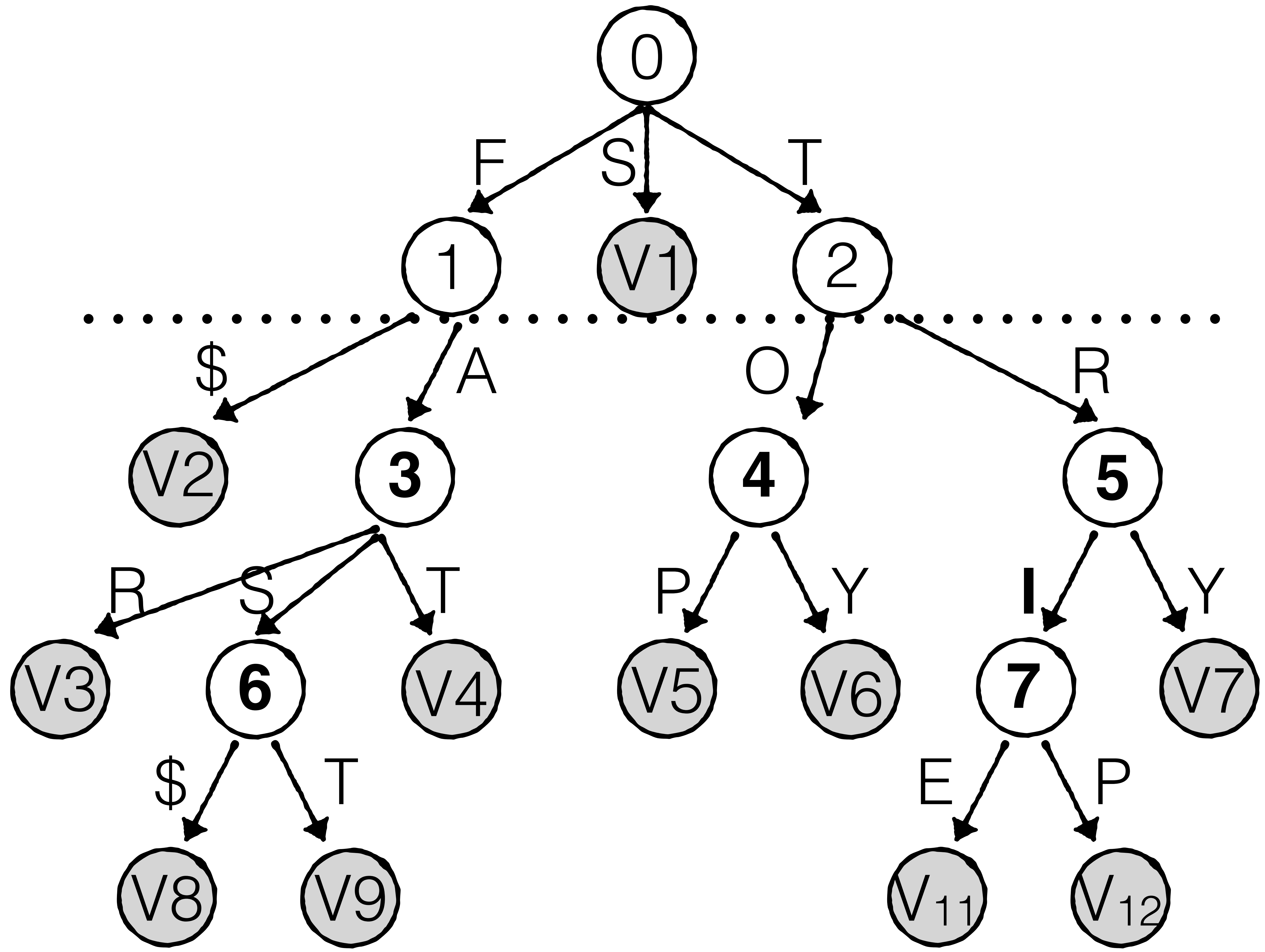


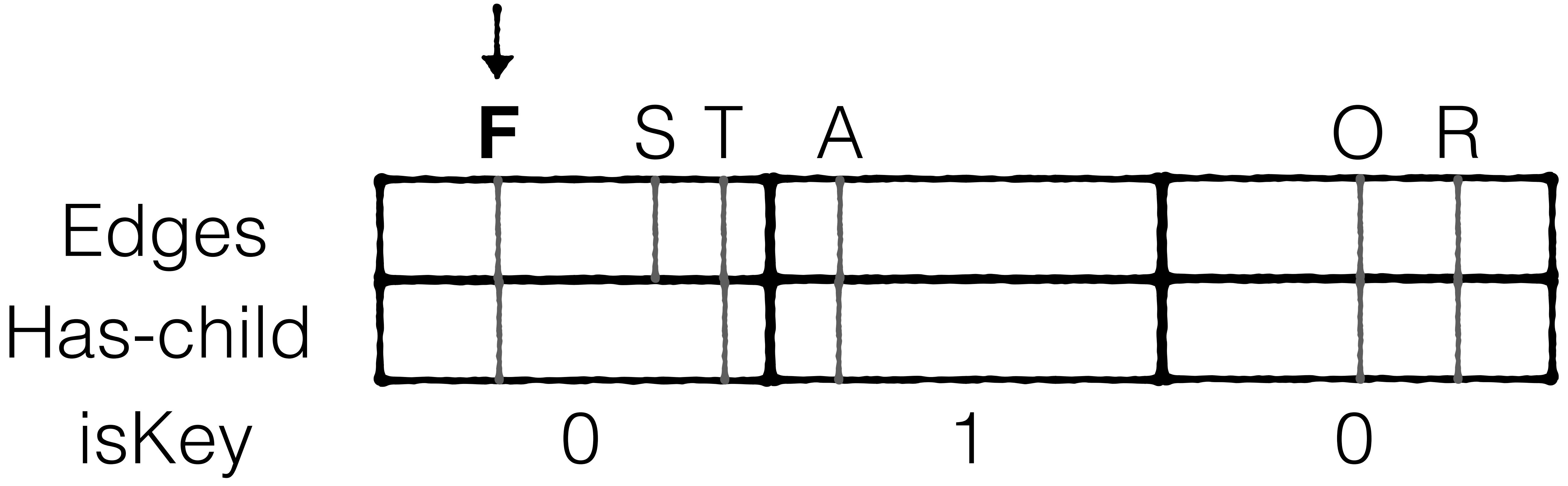


	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0

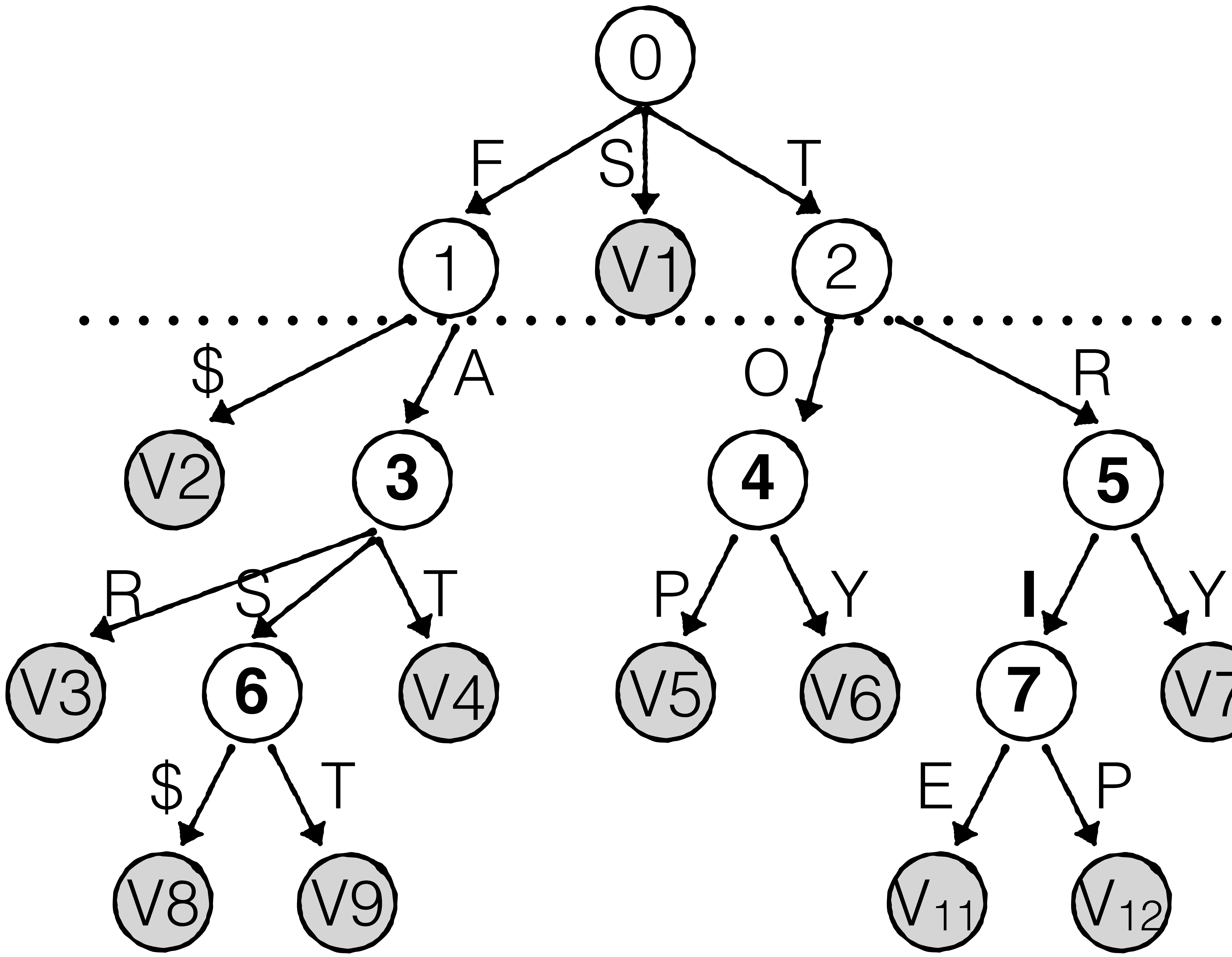
Whole filter :)

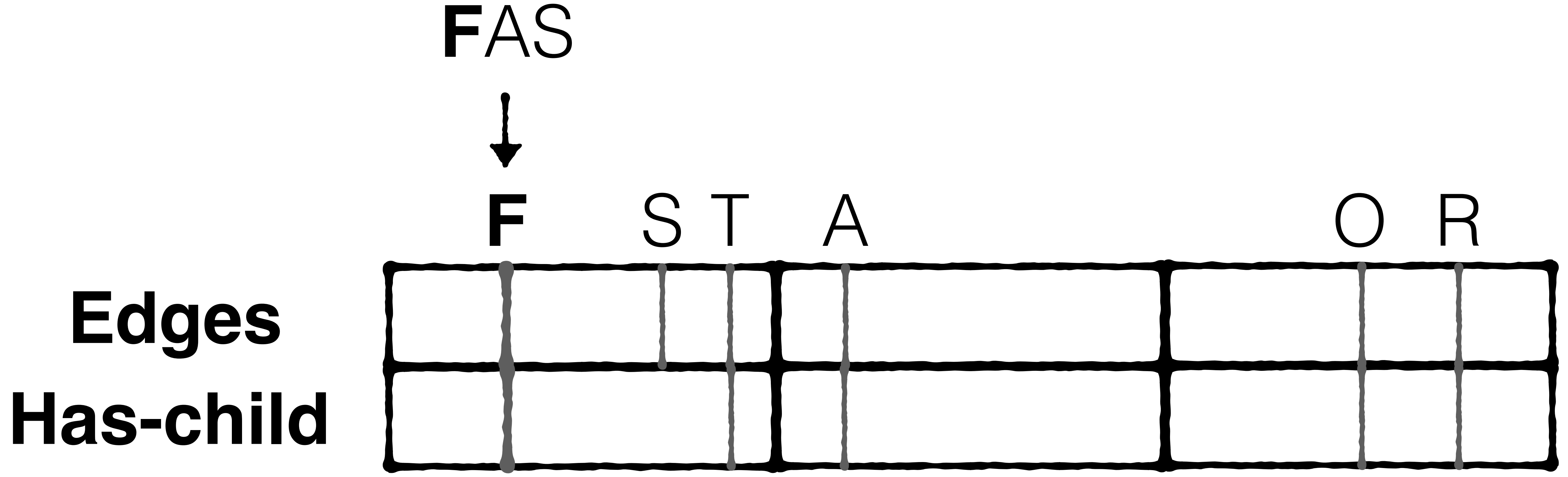
Does “FAS” exist?



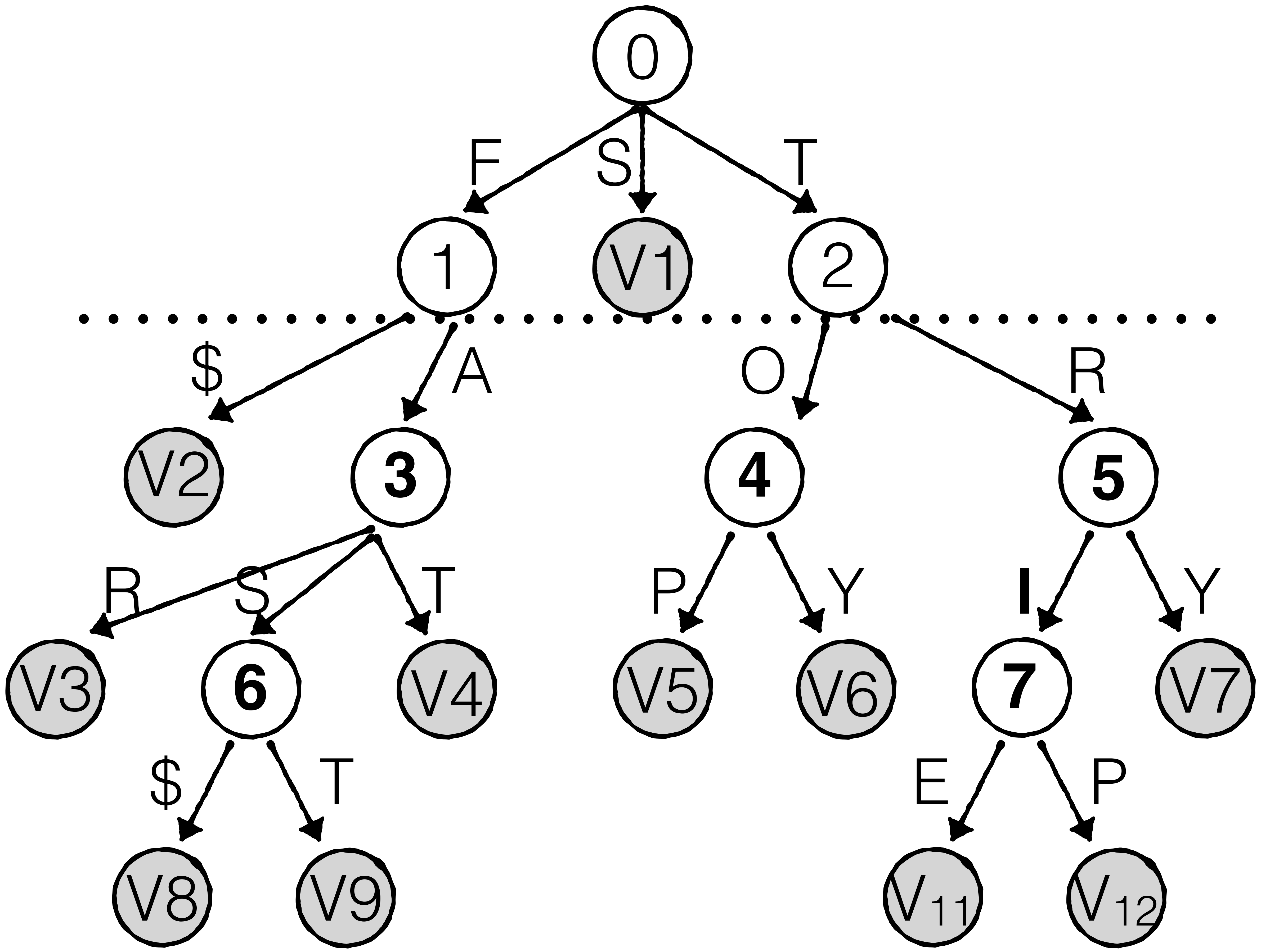


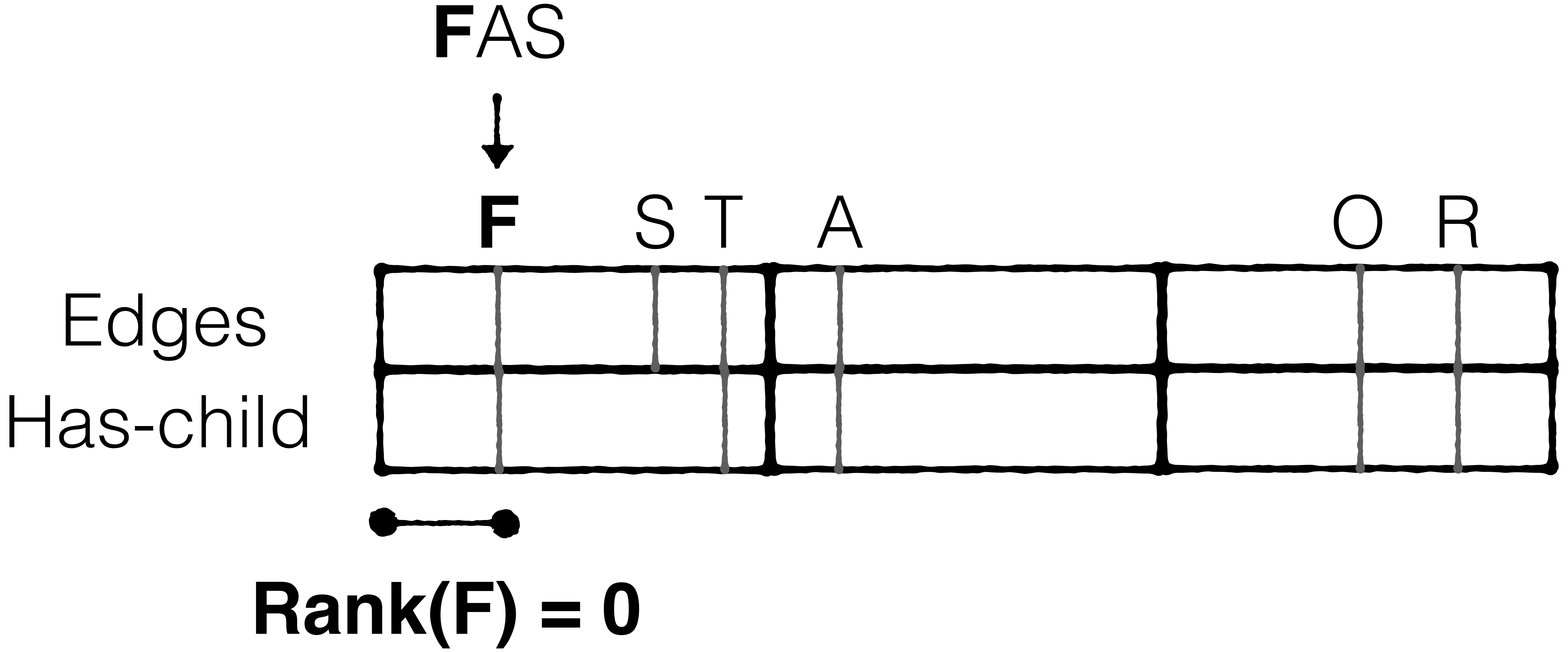
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0



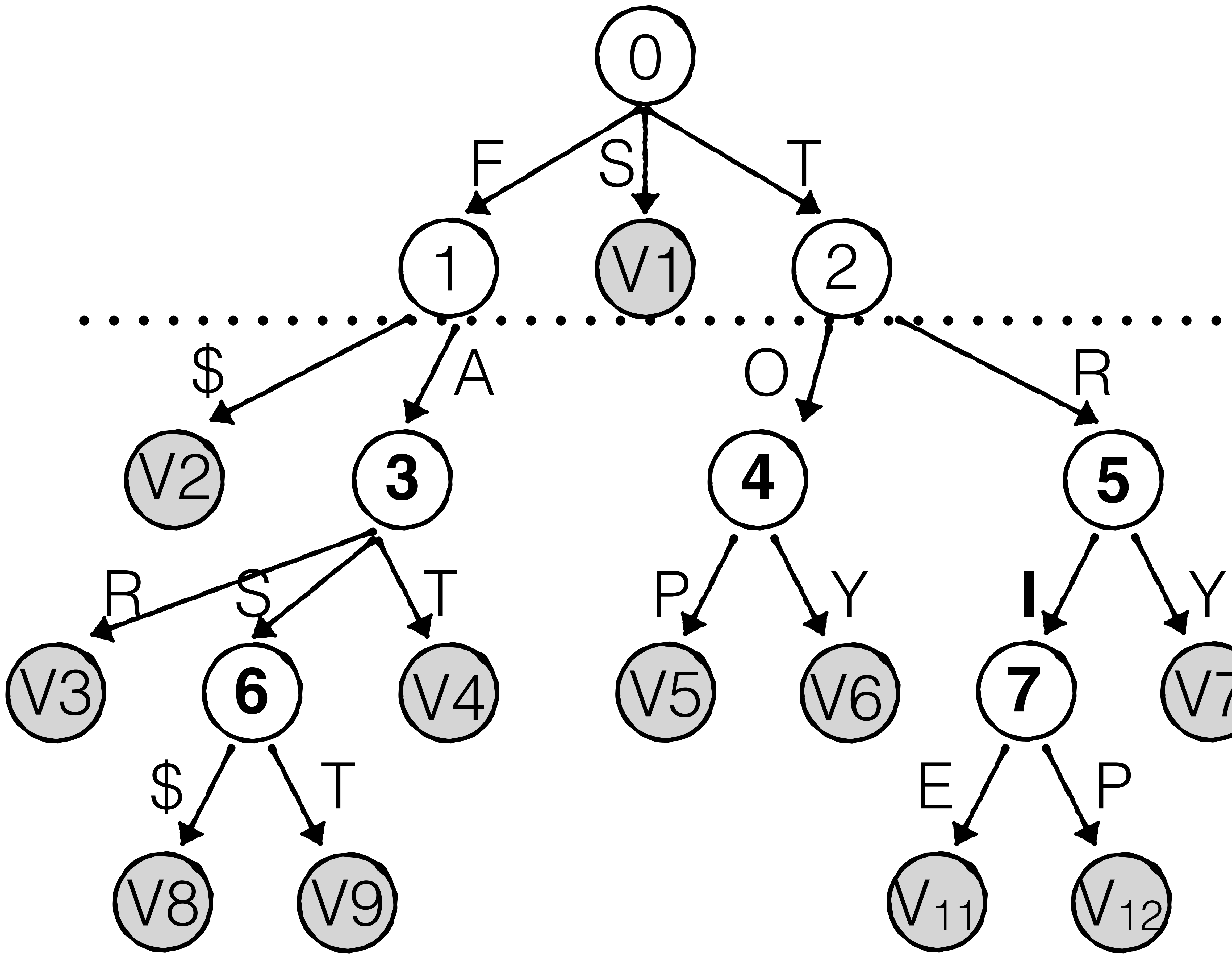


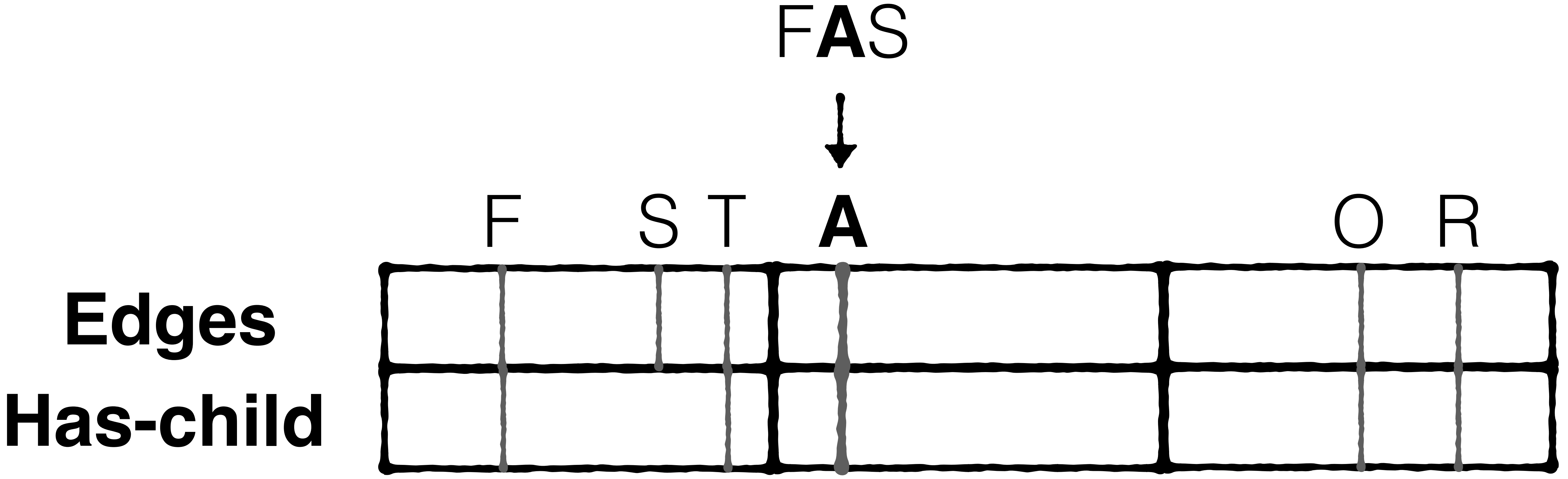
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0



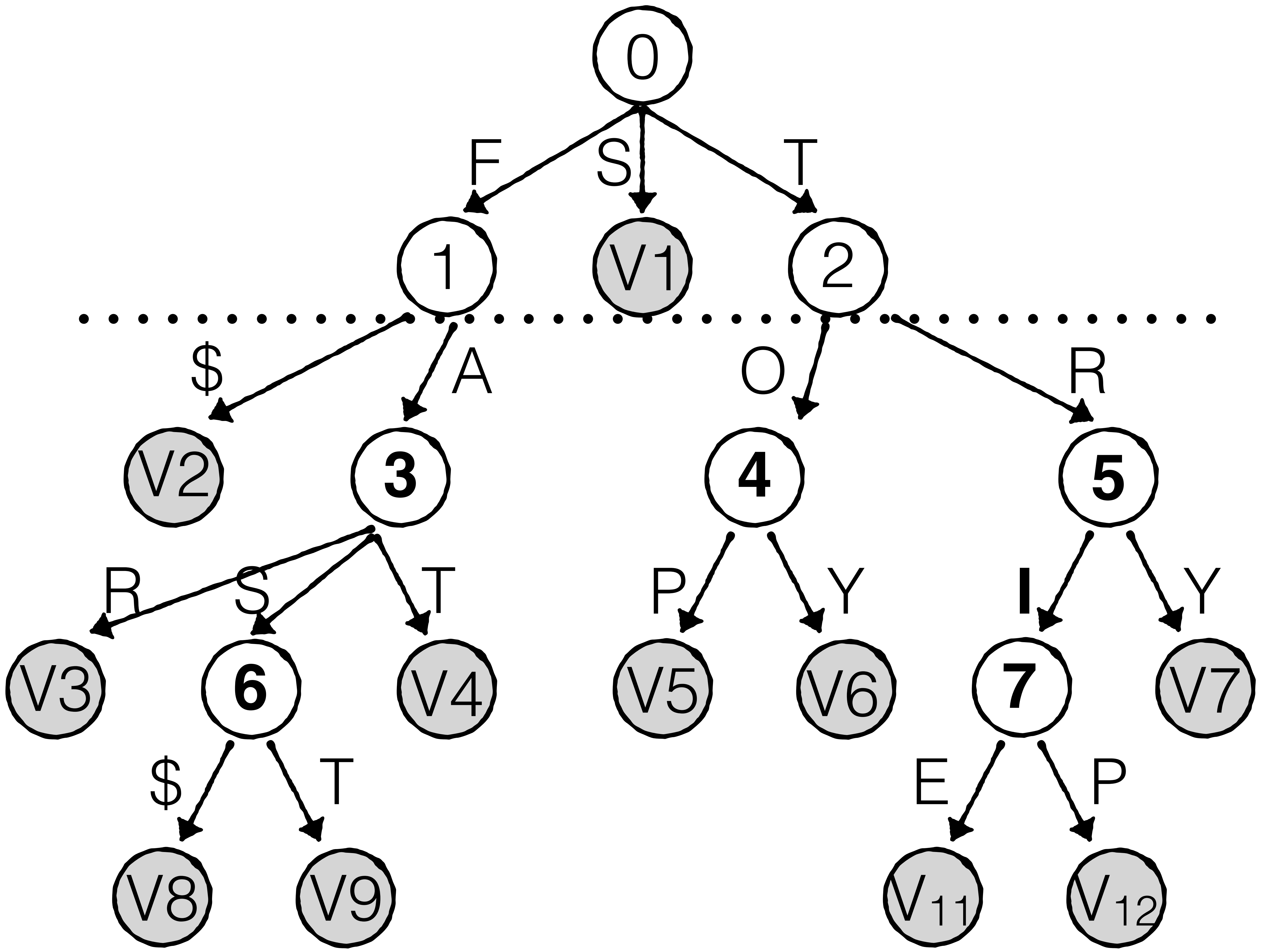


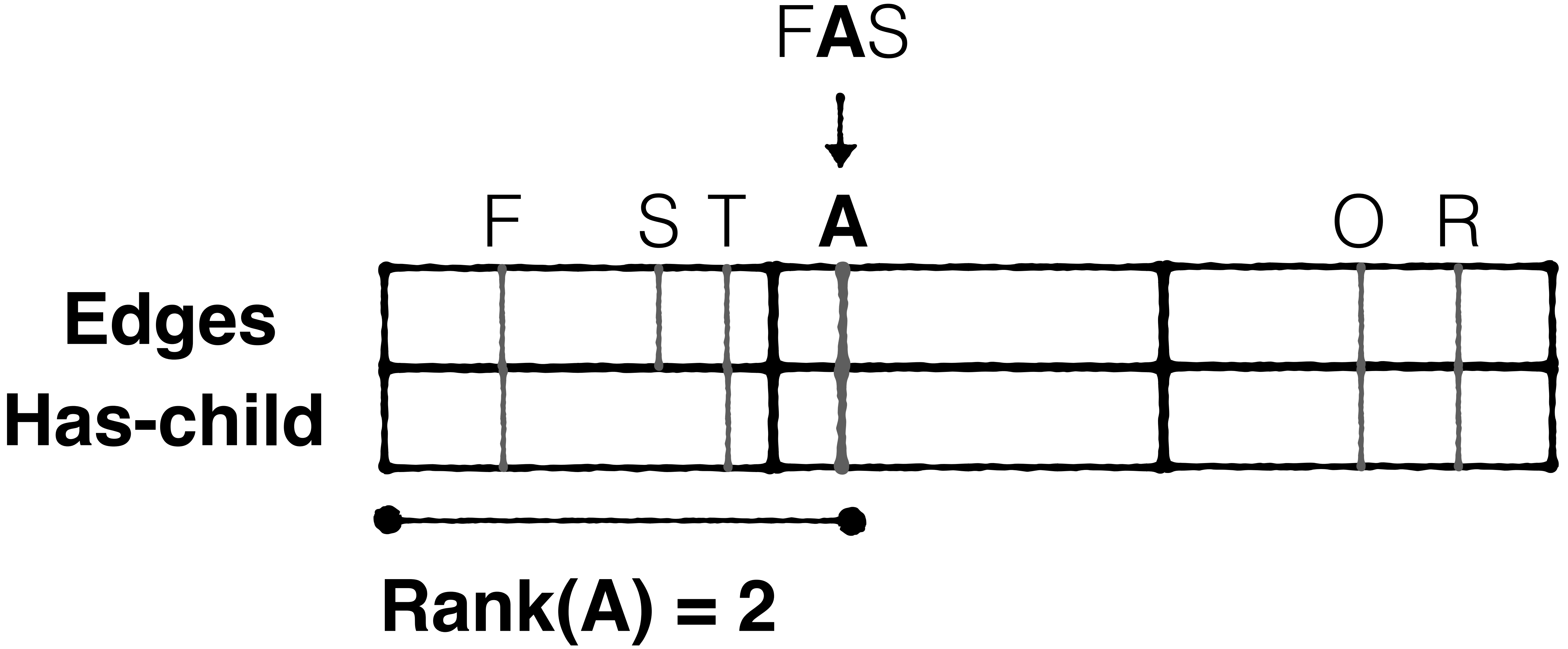
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0



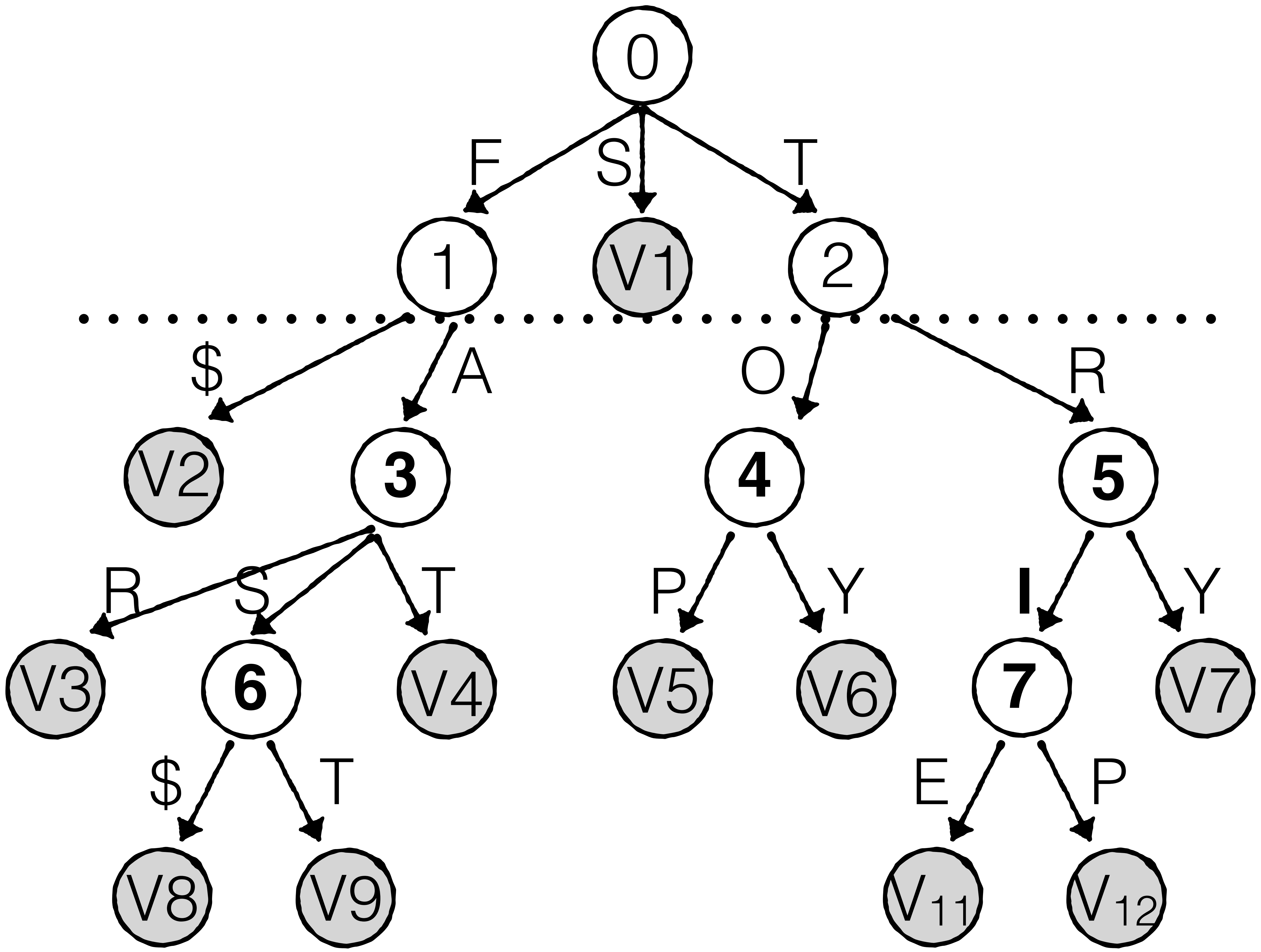


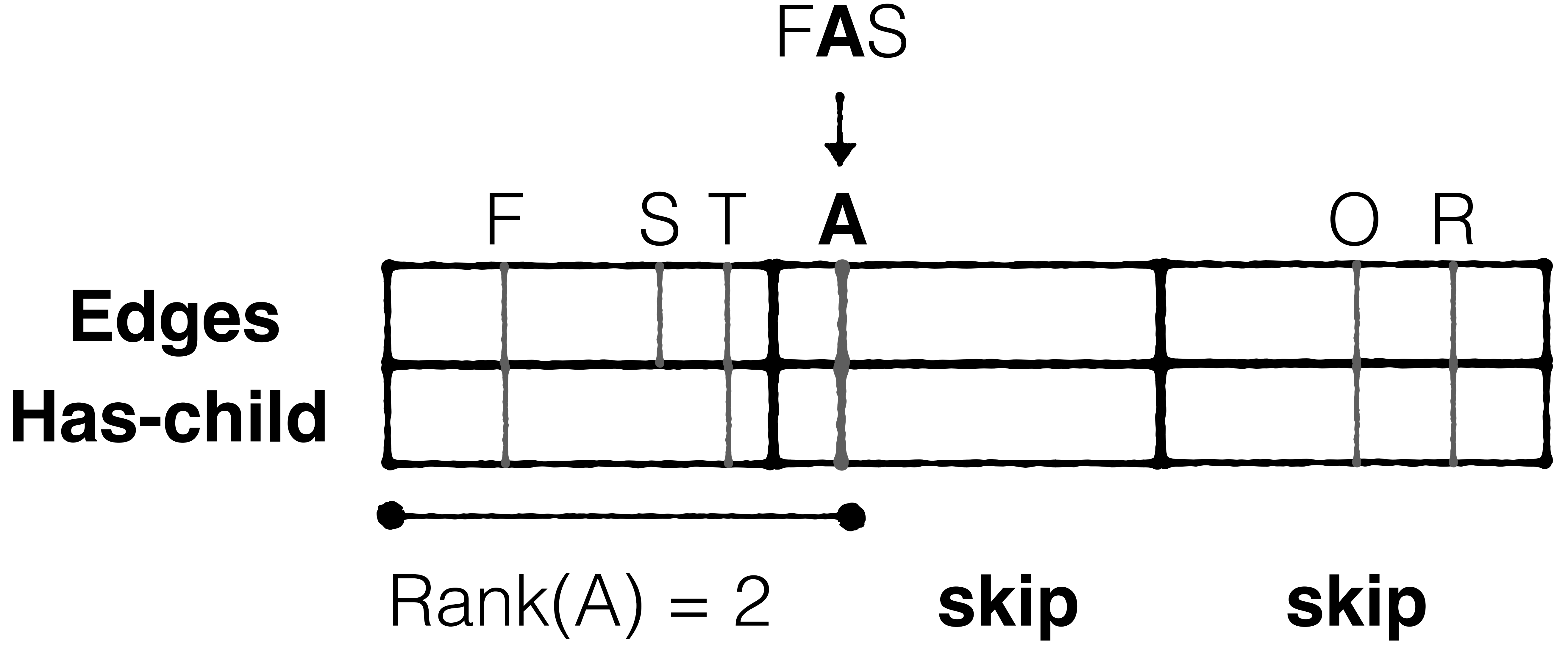
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0



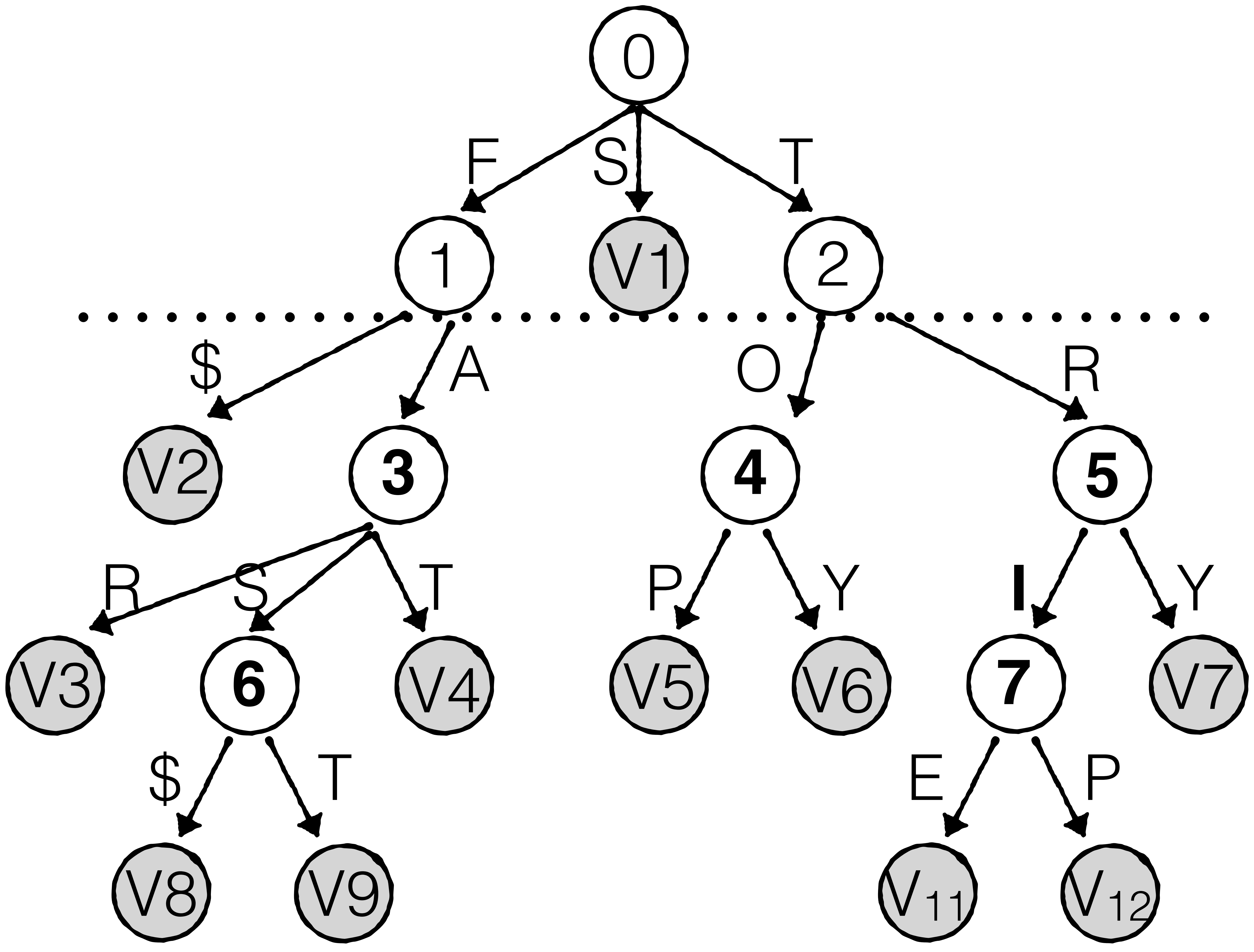


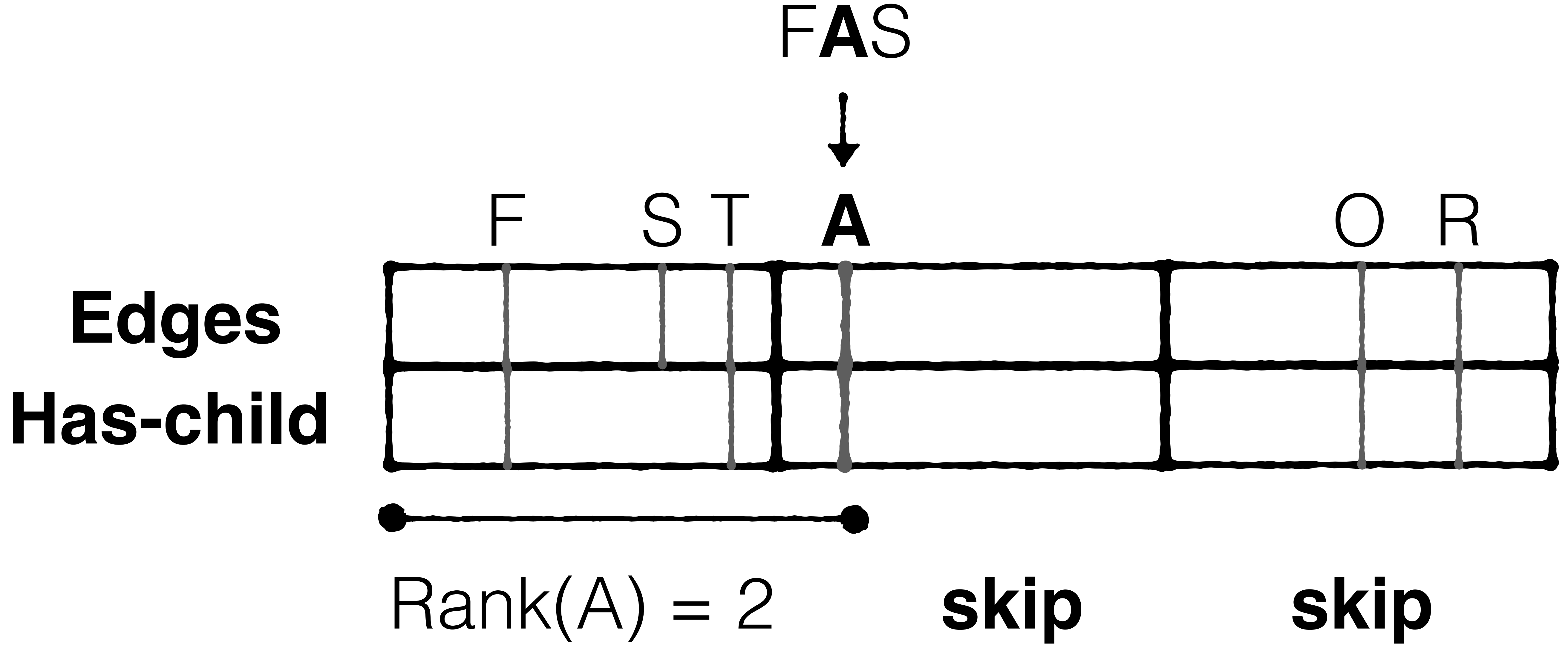
	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0





	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0

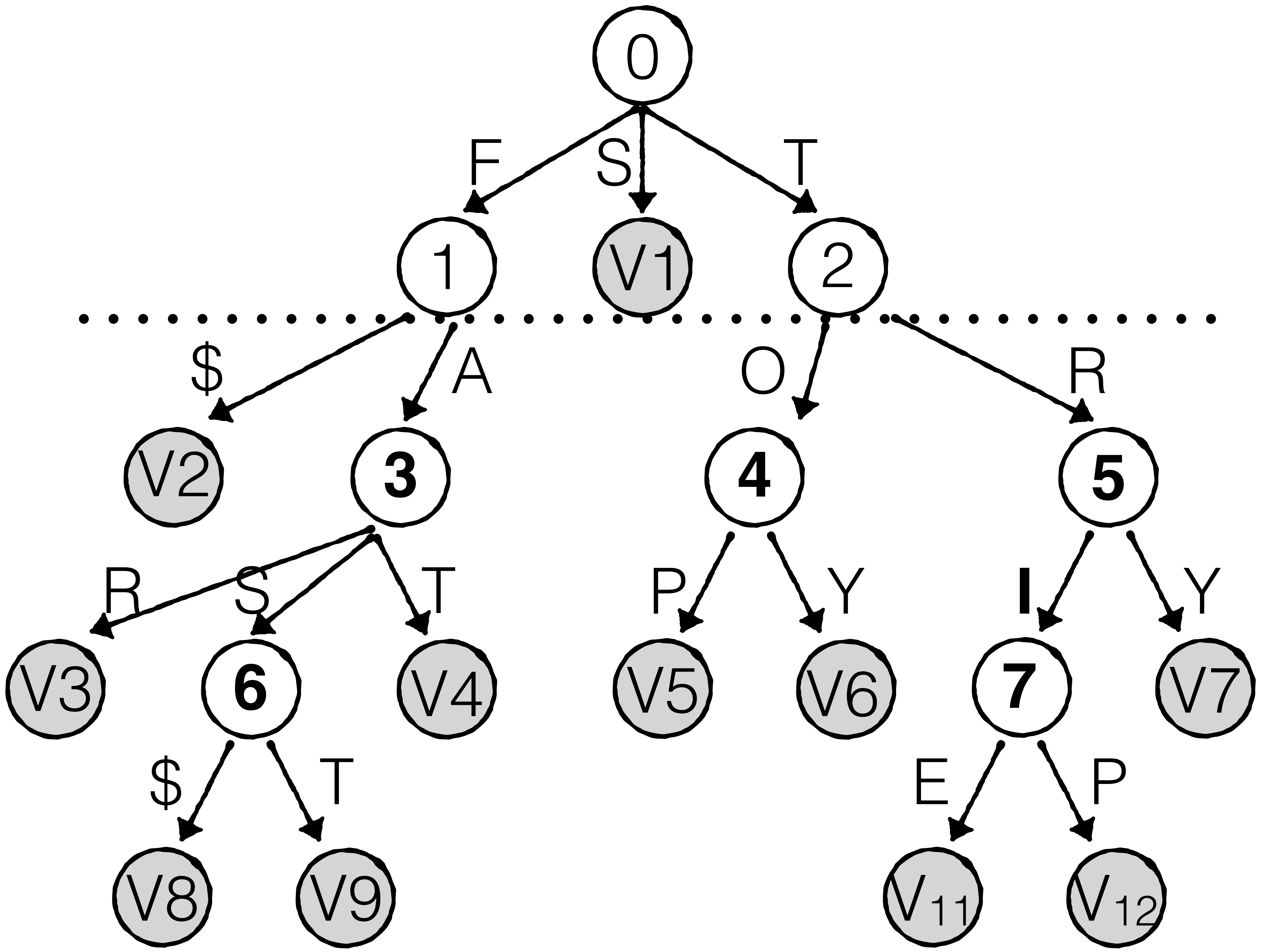


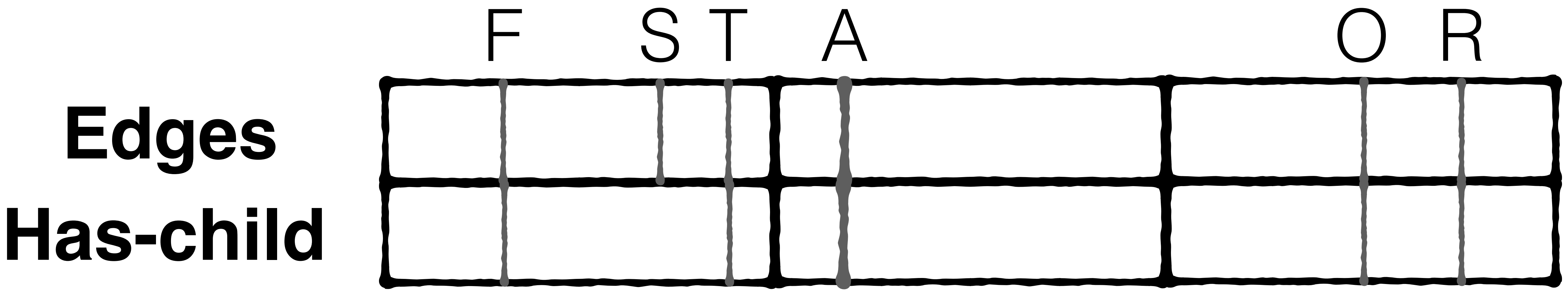


	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**

Select(2 - #dense nodes - 1)


FAS
F S T A O R
Edges
Has-child
Rank(A) = 2
skip
skip
0
F S T
1 V1 2
$ A O R
V2 3 4 5
R S T P Y I Y
V3 6 V4 V5 V6 7 V7
$ T E P
V8 V9 V11 V12
R S T P Y I Y $ T E P
Has-child 0 1 0 0 0 1 0 0 0 0 0
Start 1 0 0 1 0 1 0 1 0 1 0
Select(0) = 0



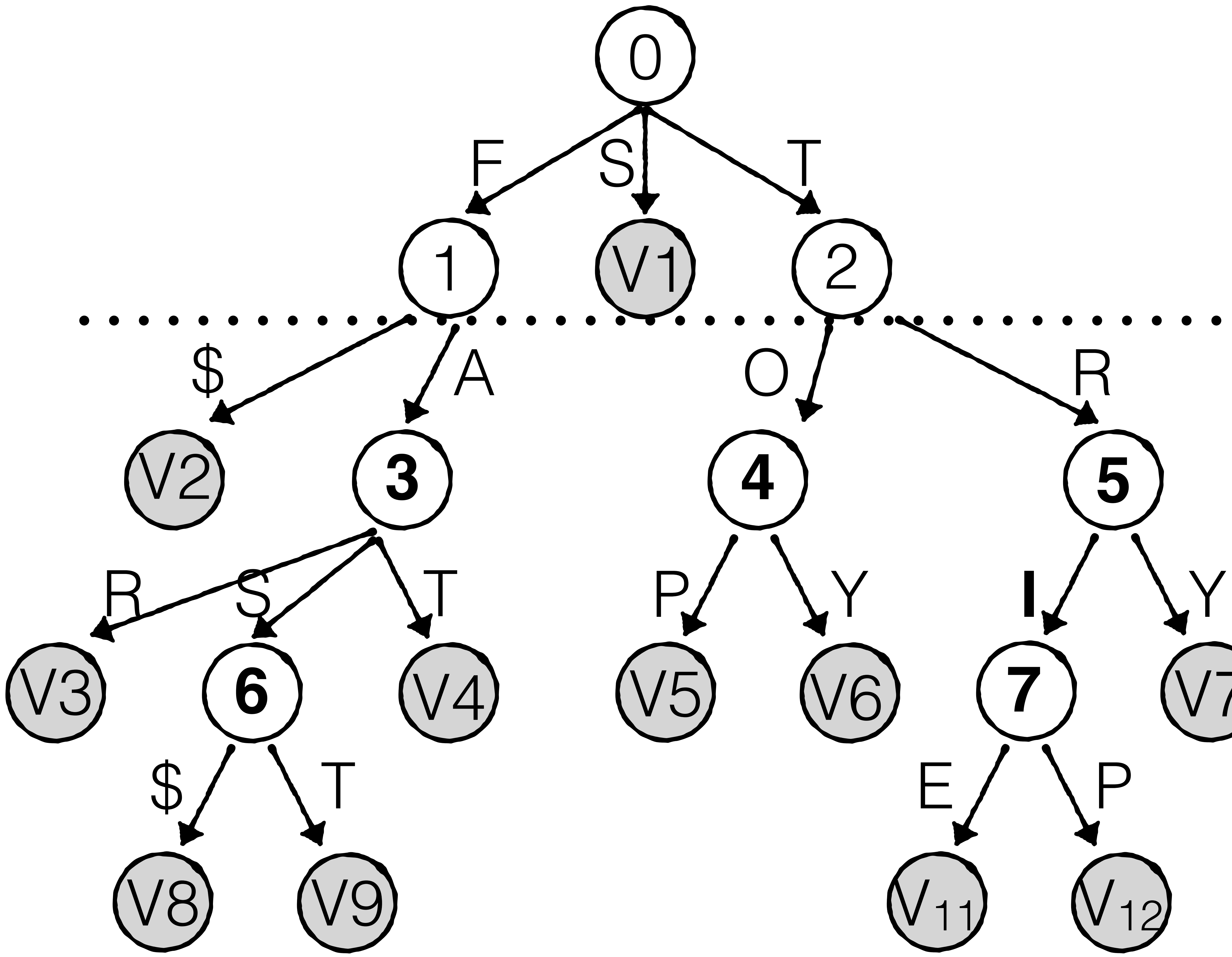


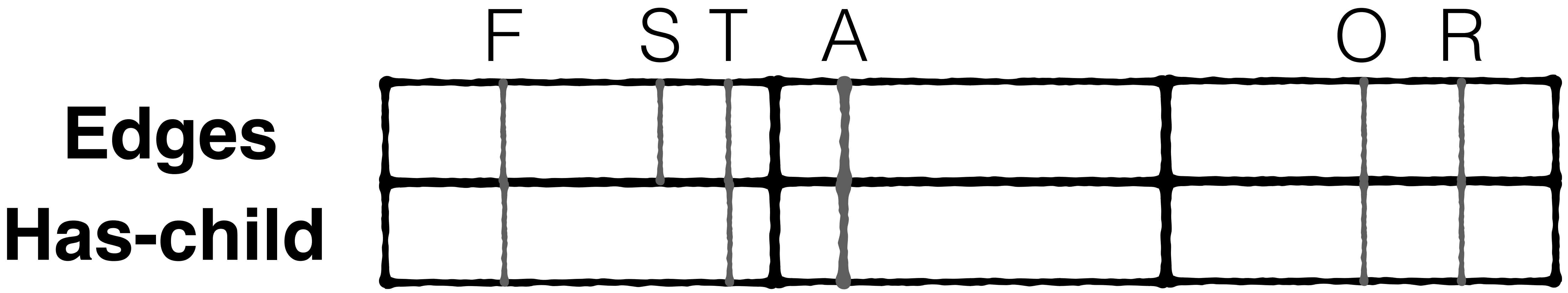
Node 3

	R	**S**	**T**	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	**1**	0	0	1	0	1	0	1	0	1	0

FA**S**

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
R
S
T
P
Y
I
Y
V3
6
V4
V5
V6
7
V7
$
T
E
P
V8
V9
V11
V12
F S T A O R
Edges
Has-child
Node 3
R S T P Y I Y $ T E P
Has-child 0 1 0 0 0 1 0 0 0 0 0
Start 1 0 0 1 0 1 0 1 0 1 0
FAS
Scan

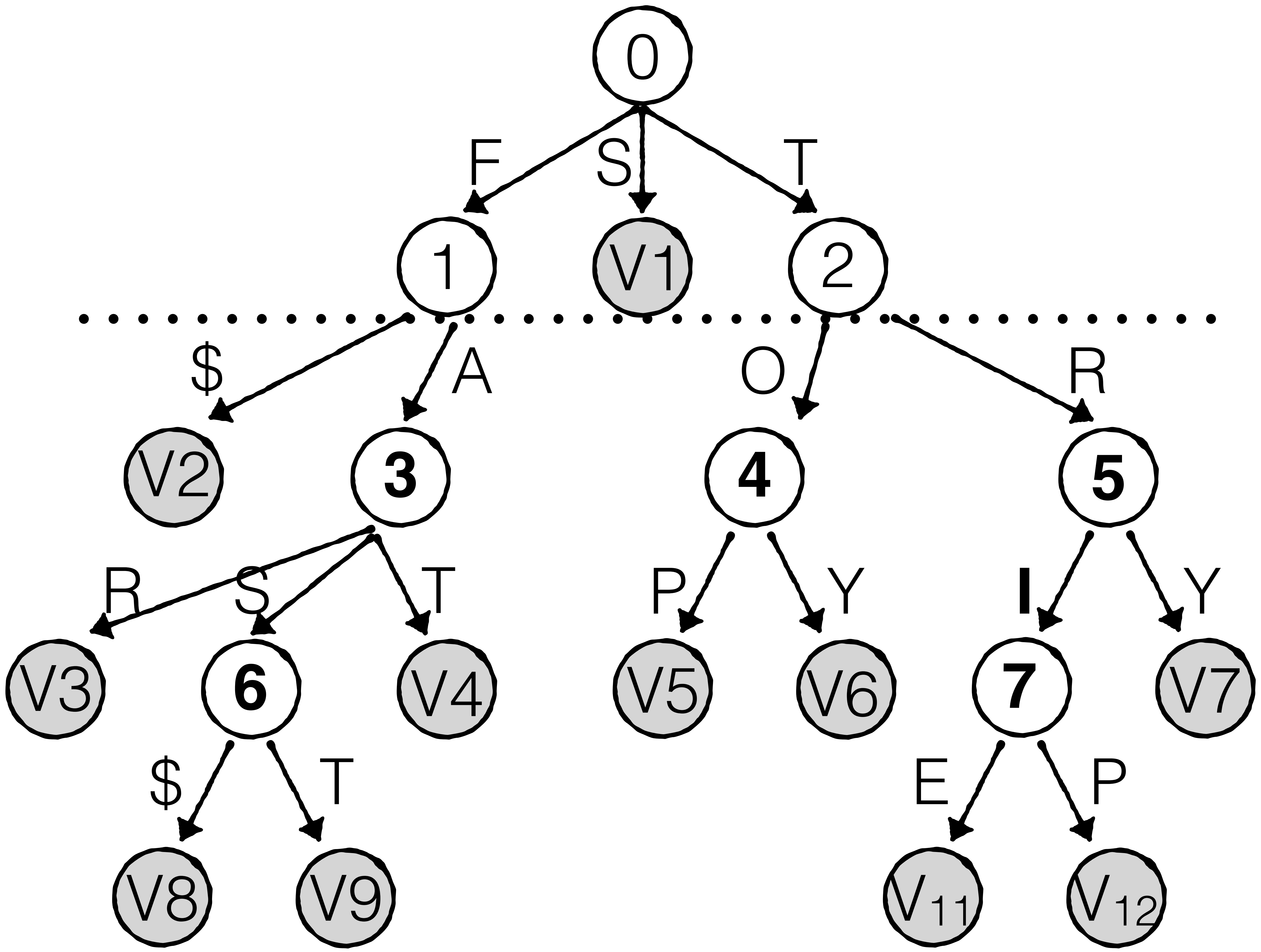


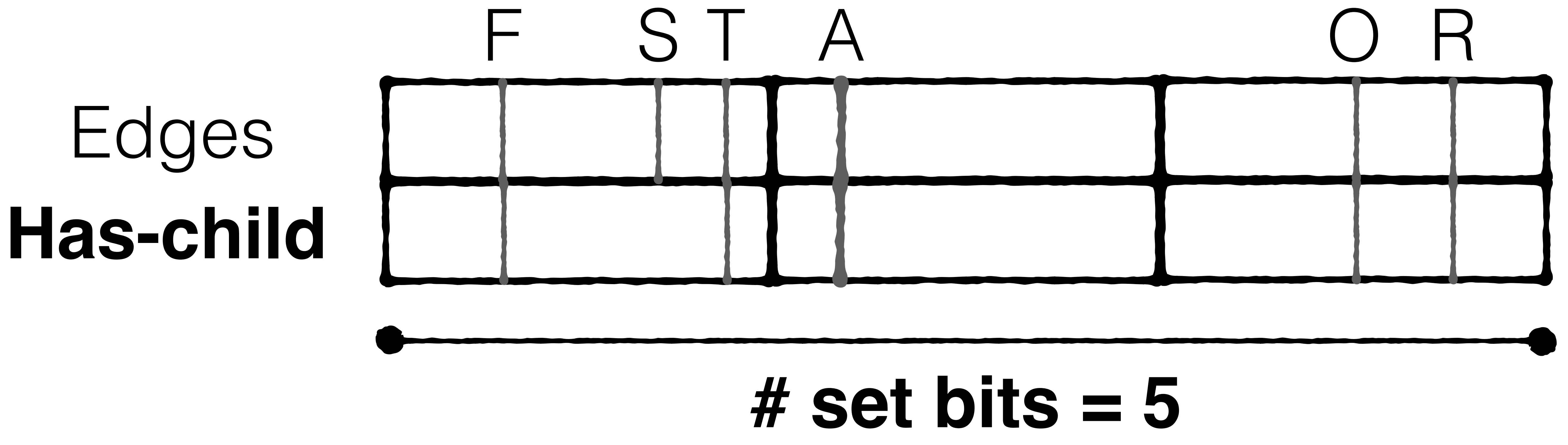


Node 3

	R	**S**	T	P	Y	I	Y	$	T	E	P
Has-child	0	**1**	0	0	0	1	0	0	0	0	0
Start	1	**0**	0	1	0	1	0	1	0	1	0

FA**S**

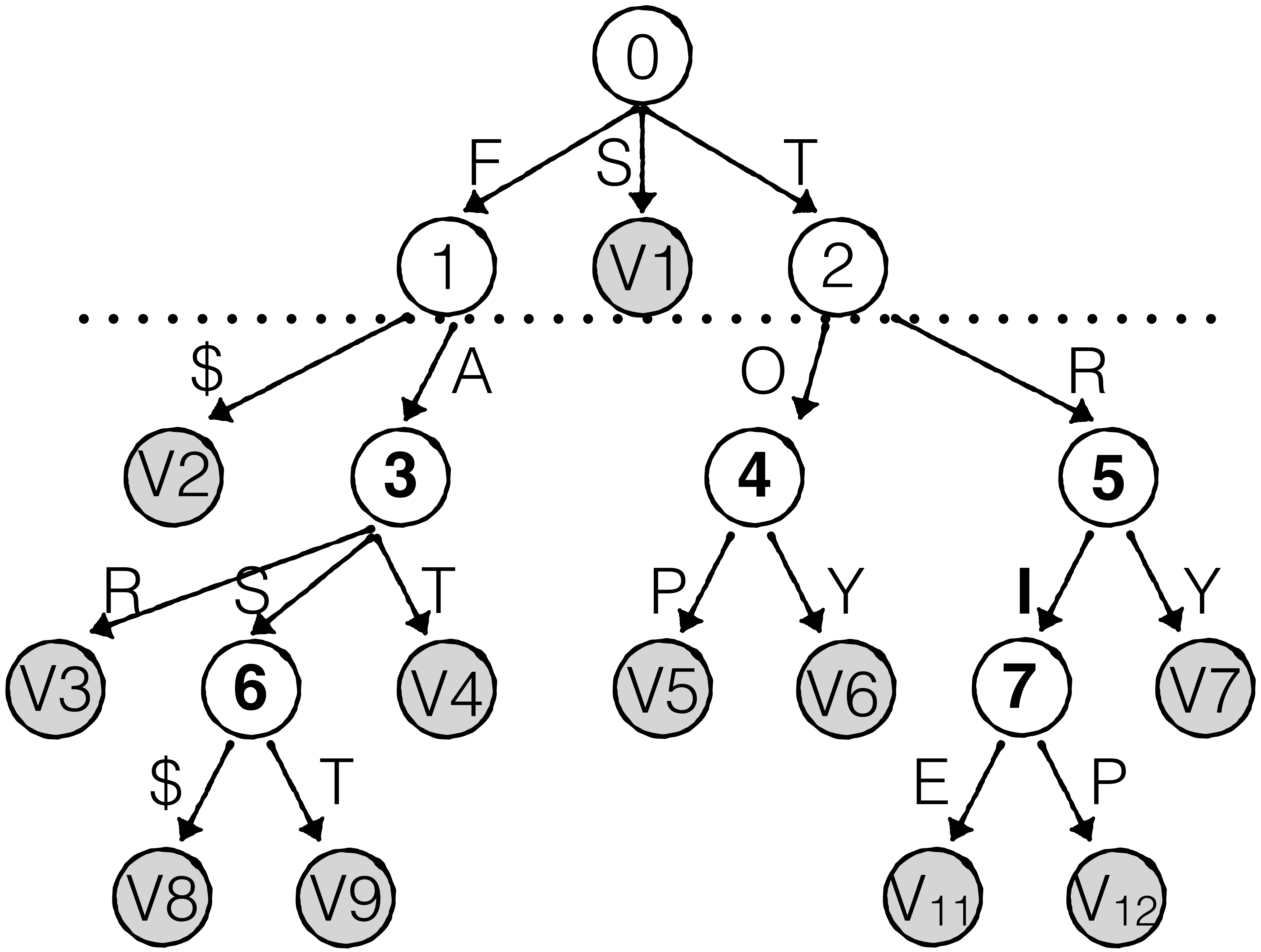


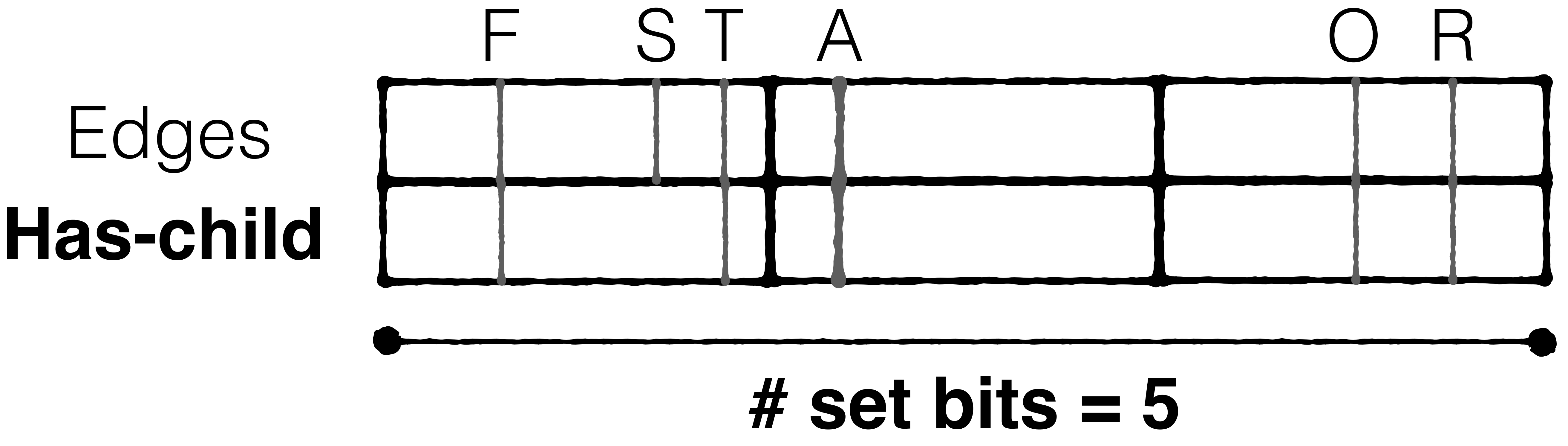


Node 3

	R	**S**	T	P	Y	I	Y	$	T	E	P
Has-child	0	**1**	0	0	0	1	0	0	0	0	0
Start	1	**0**	0	1	0	1	0	1	0	1	0

FA**S**



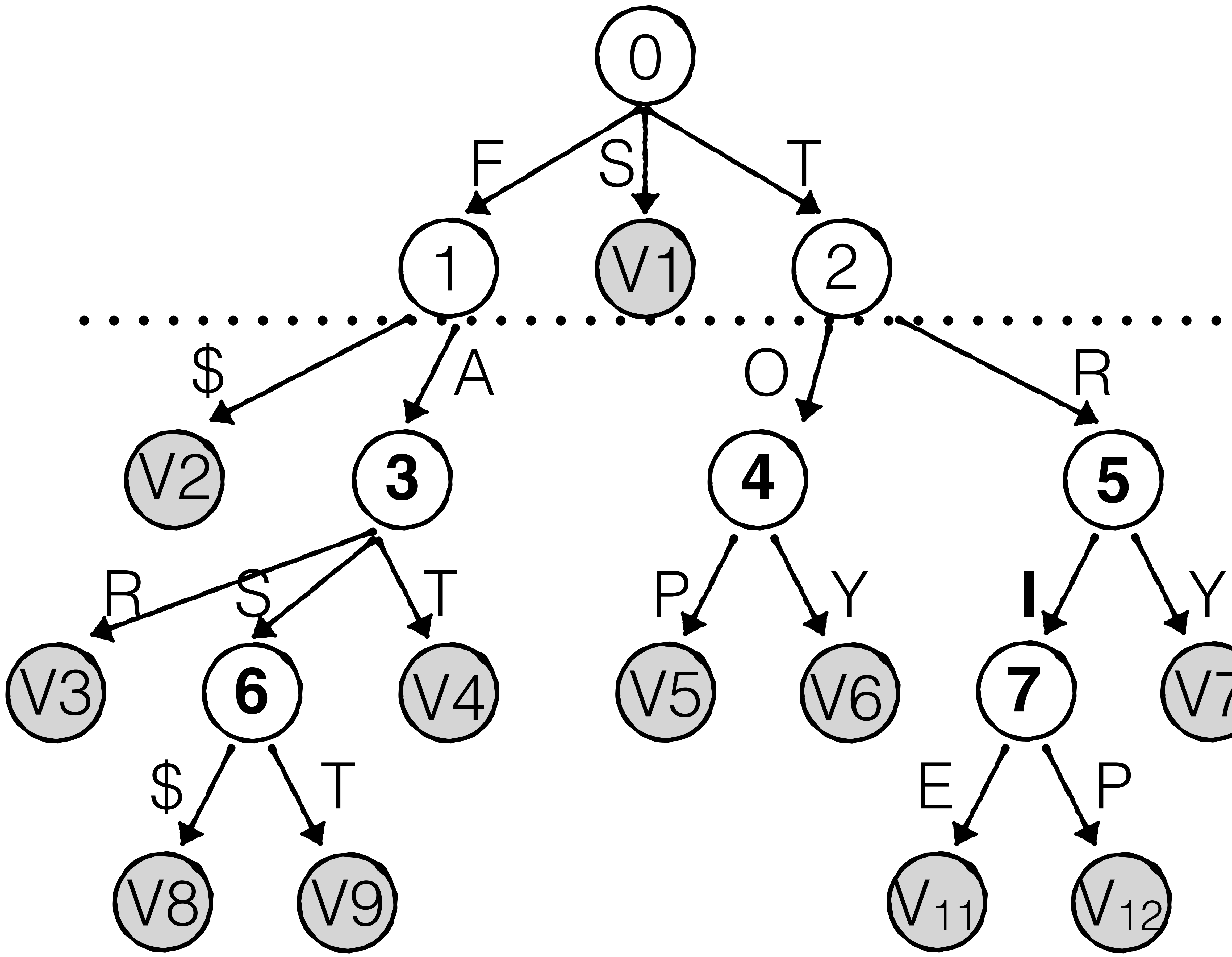


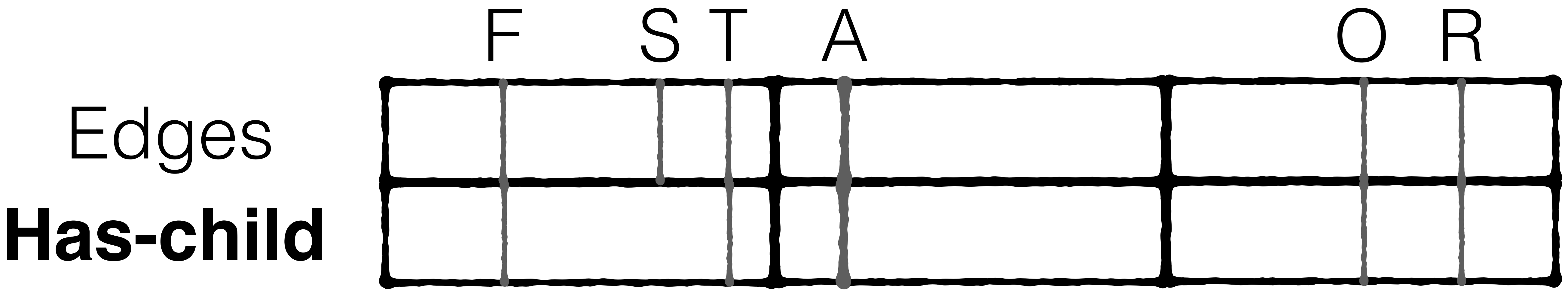
Rank(1) = 0

	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	**0**	**1**	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0

FA**S**

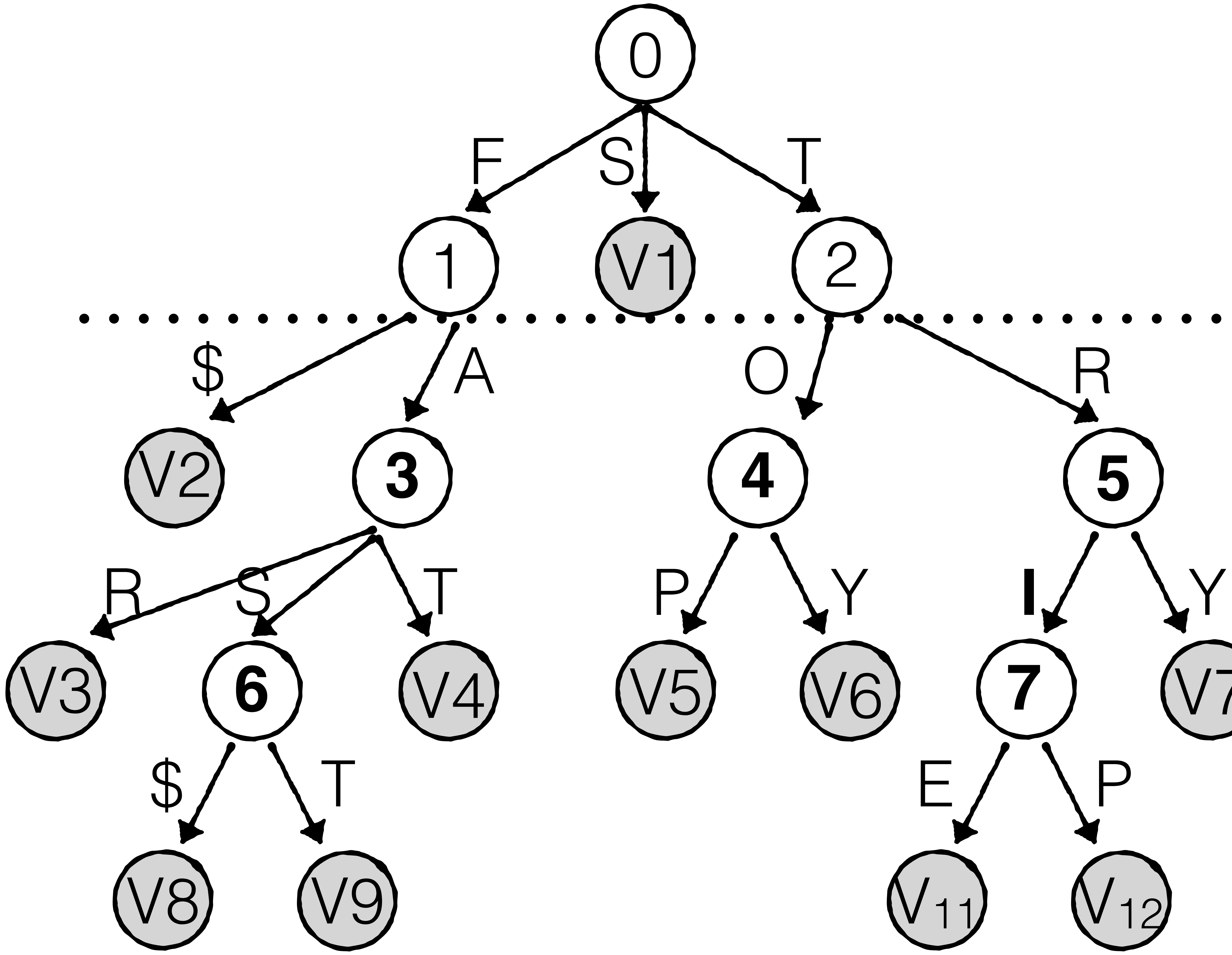

0
F
S
T
1
V1
2
$
A
O
R
V2
3
4
5
R
S
T
P
Y
I
Y
V3
6
V4
V5
V6
7
V7
$
T
E
P
V8
V9
V11
V12
F S T A O R
Edges
Has-child
R S T P Y I Y $ T E P
Has-child 0 1 0 0 0 1 0 0 0 0 0
Start 1 0 0 1 0 1 0 1 0 1 0
select(5 - 2) = 7
FAS

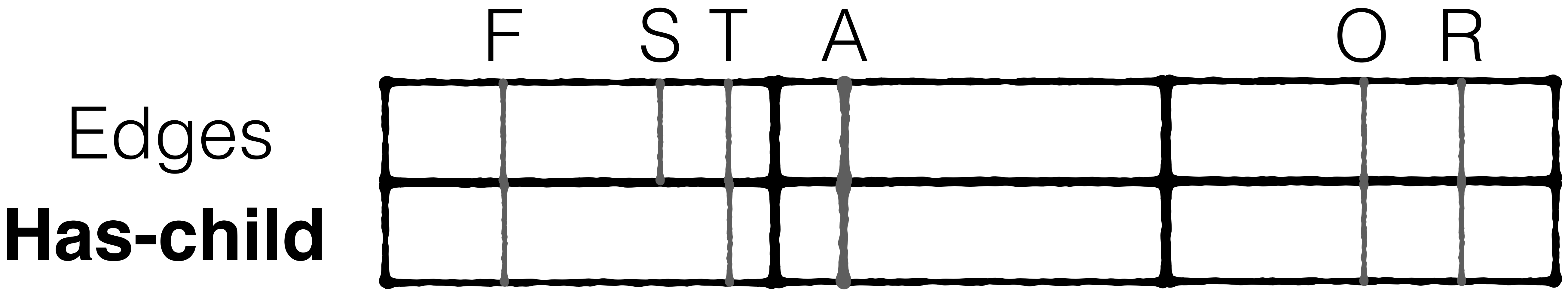




	R	S	T	P	Y	I	Y	**$**	T	E	P
Has-child	0	1	0	0	0	1	0	**0**	0	0	0
Start	1	0	0	1	0	1	0	**1**	0	1	0

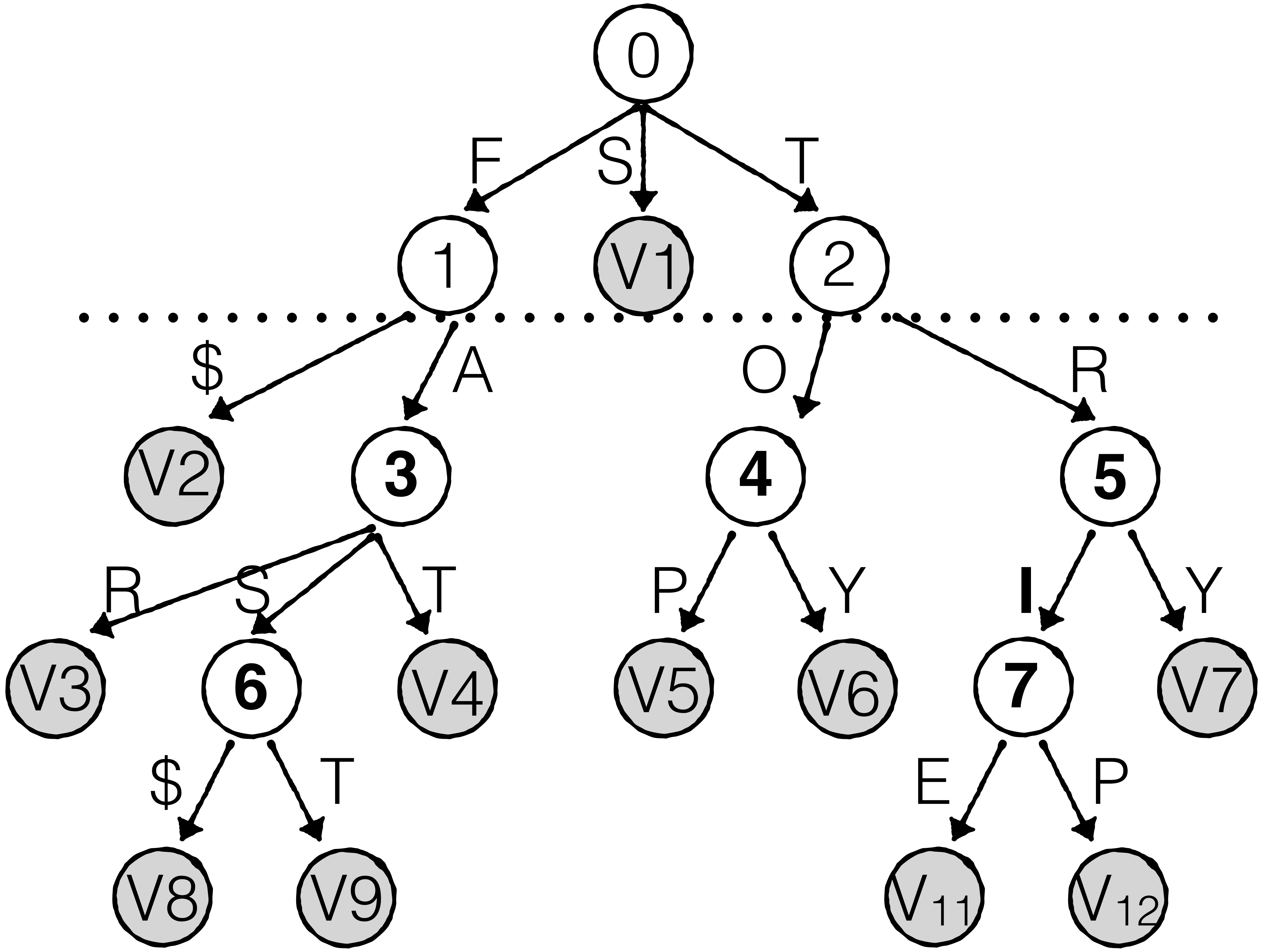
FA**S**

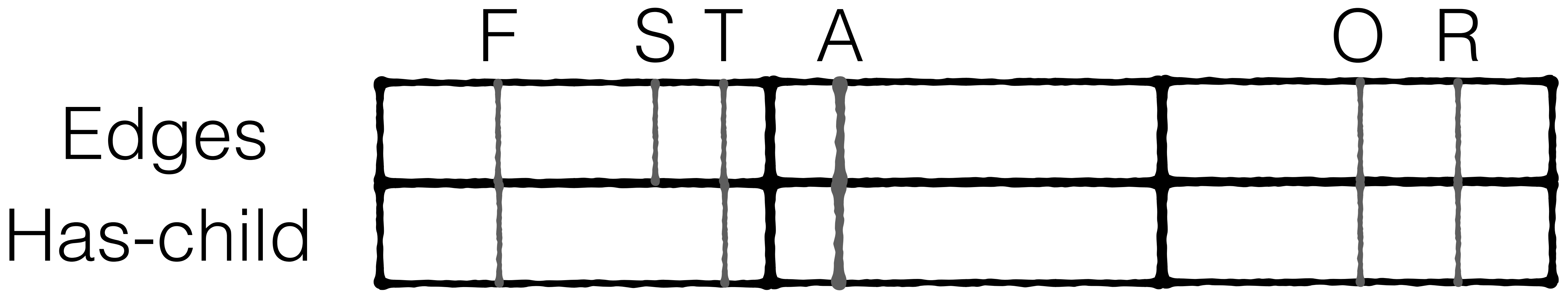




	R	S	T	P	Y	I	Y	**$**	T	E	P
Has-child	0	1	0	0	0	1	0	**0**	0	0	0
Start	1	0	0	1	0	1	0	**1**	0	1	0

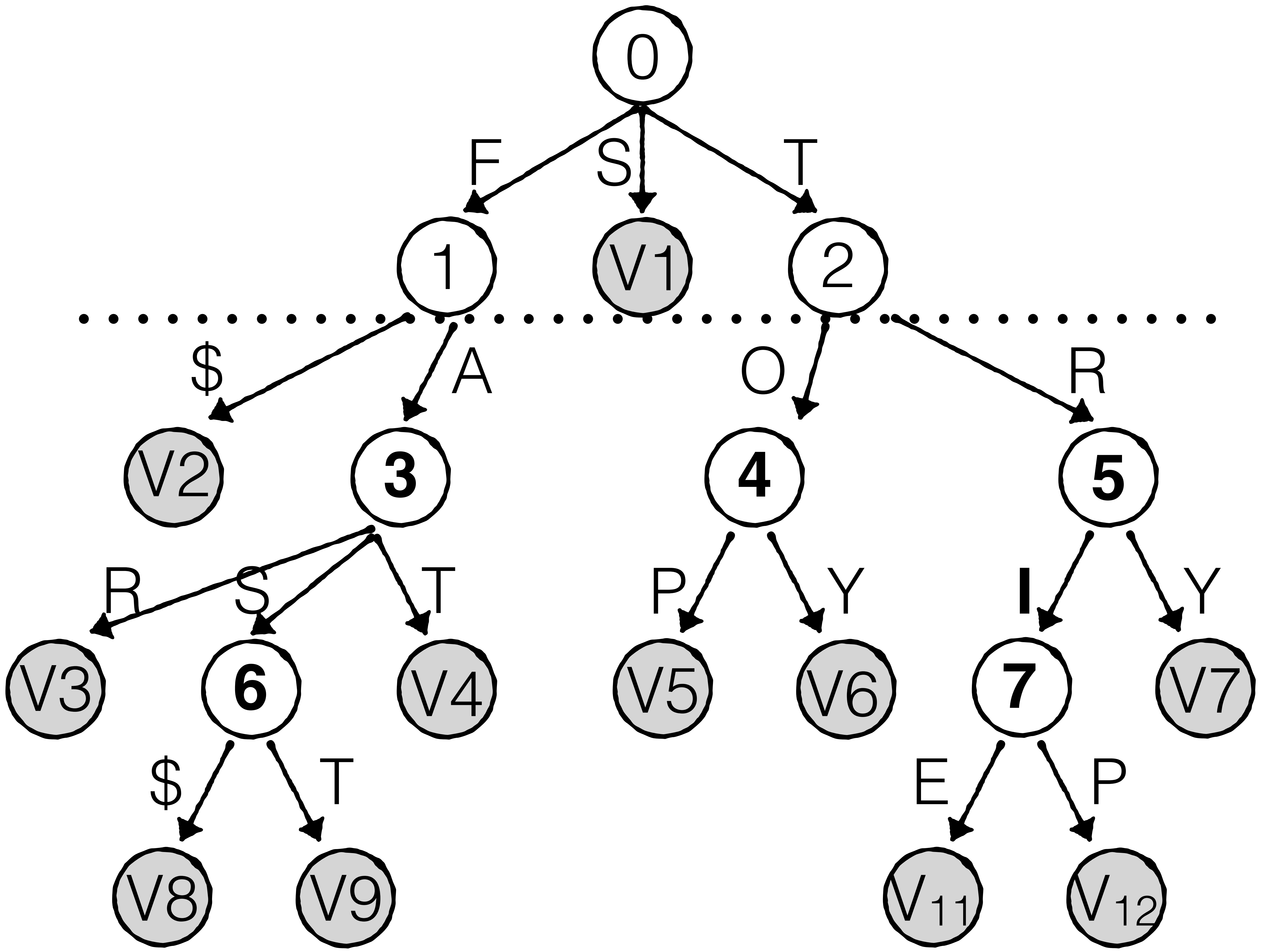
V_8

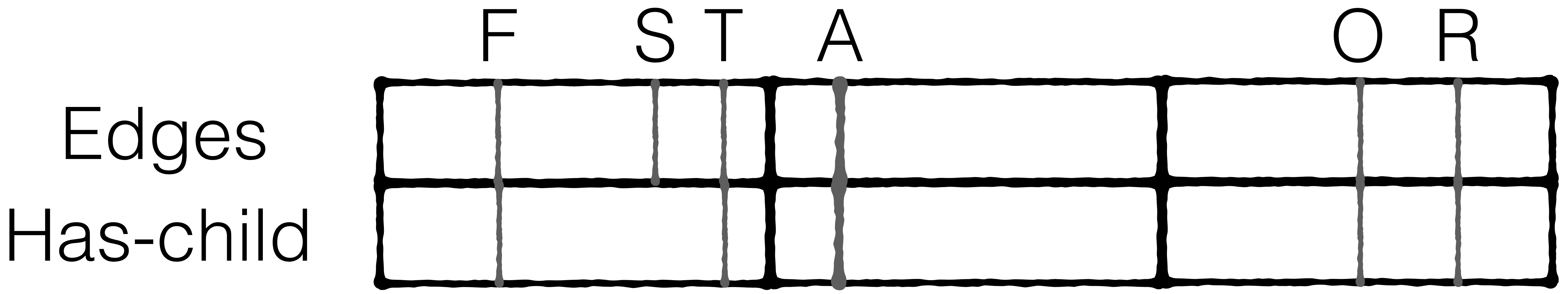




	R	S	T	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0

Sparse encoding is space efficient for nodes within many edges





	R	**S**	**T**	P	Y	I	Y	$	T	E	P
Has-child	0	1	0	0	0	1	0	0	0	0	0
Start	1	0	0	1	0	1	0	1	0	1	0

Sparse encoding is space efficient for nodes within many edges

But slower as we must scan edges for each node

	SuRF
	
Var-length keys & queries	**Yes**
FPR guarantee	**None**
Query speed	**O(L)** (L = key length)
Dynamic	**No**

SuRF

Memento

Diva