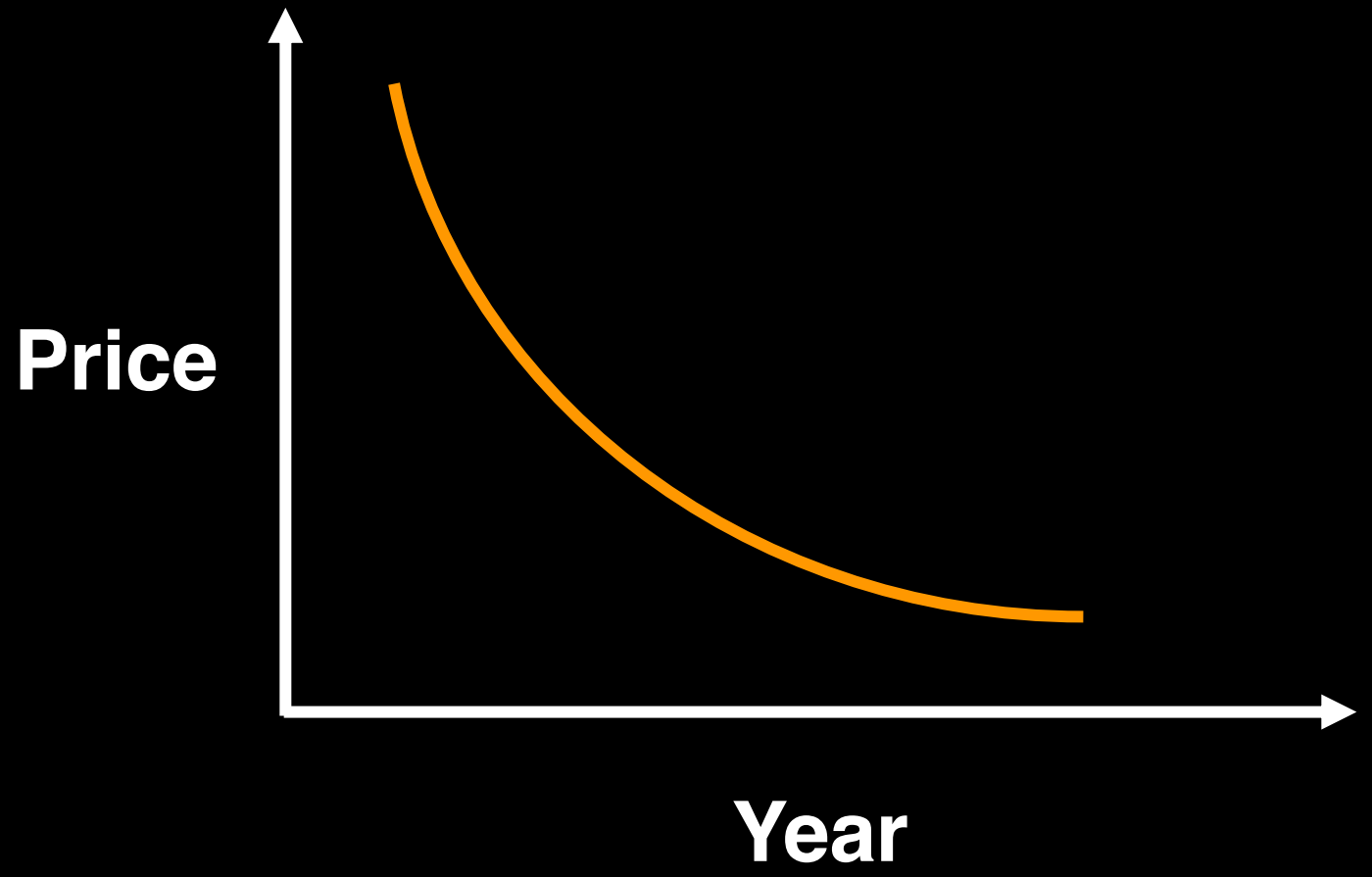
Succinct Trie Indexes Made Practical

Huanchen Zhang

David G. Andersen, Michael Kaminsky,
Andrew Pavlo, Kimberly Keeton

DRAM price won’t fall forever



Memory-efficient data structures are helpful

Smaller data structures

More data resident in faster memory

Better performance + lower costs

The limit: information-theoretic lower bound (ITLB)

➔ The minimum # of bits required to distinguish any object in a class

|S| = n ➔ $log_2 n$ bits

The limit: information-theoretic lower bound (ITLB)

➔ The minimum # of bits required to distinguish any object in a class

|S| = n ➔ $log_2 n$ bits

|n-node trie of degree k| = $\binom{kn+1}{n}/kn+1$ ➔ $n(log_2 k - (k-1)log_2(k-1))$ bits

The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

|S| = n → $log_2 n$ bits

|n-node trie of degree k| = $\binom{kn+1}{n}/kn+1$ → $n(log_2 k - (k-1)log_2(k-1))$ bits

256

9.44n

The limit: information-theoretic lower bound (ITLB)

The minimum # of bits required to distinguish any object in a class

|S| = n → $log_2 n$ bits

|n-node trie of degree k| = $\binom{kn+1}{n}/kn+1$ → $n(log_2 k - (k-1)log_2(k-1))$ bits

256

9.44n

FST = 10n

Succinct Data Structures

Use # of bits close to ITLB

Suppose ITLB = L bits

Implicit: L + O(1)

Succinct: L + o(L)

Compact: O(L) ← FST

Why aren’t succinct data structures popular?

Read-only

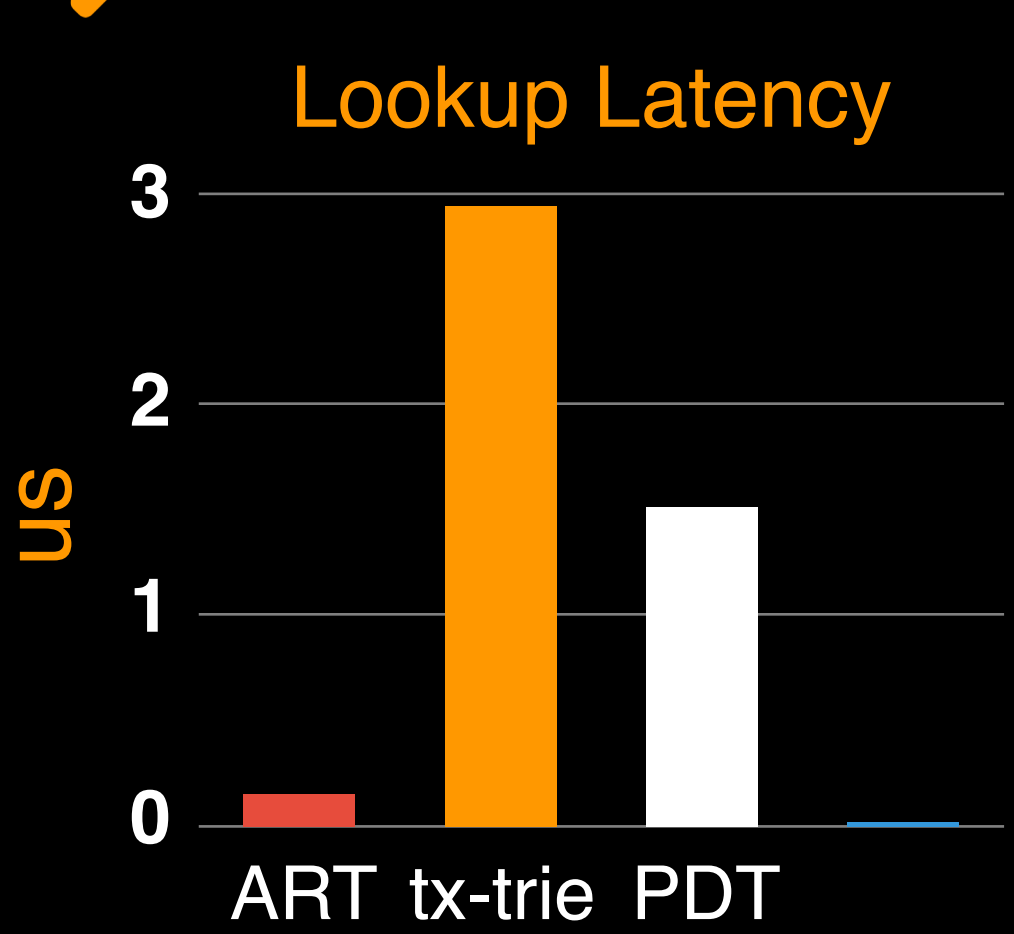
Log-structured design

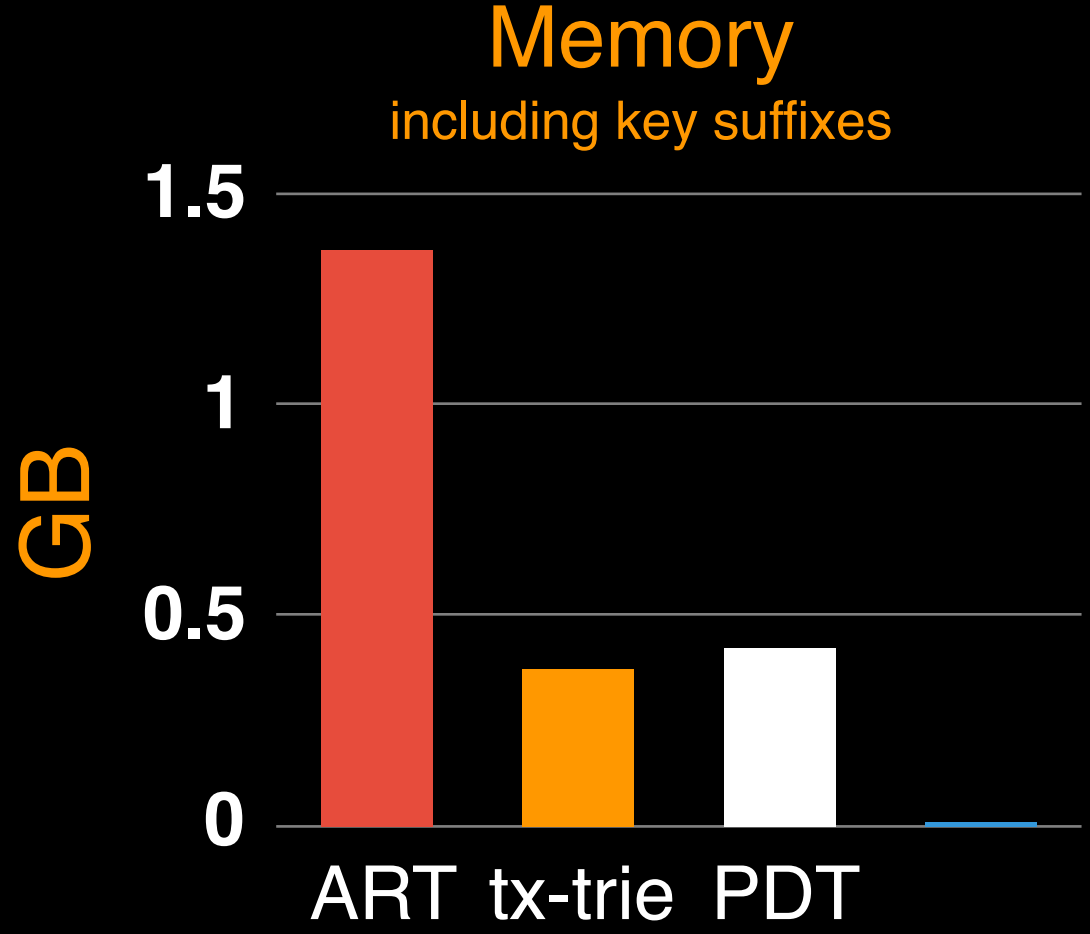
Slow

Complex

Existing succinct tries are slow

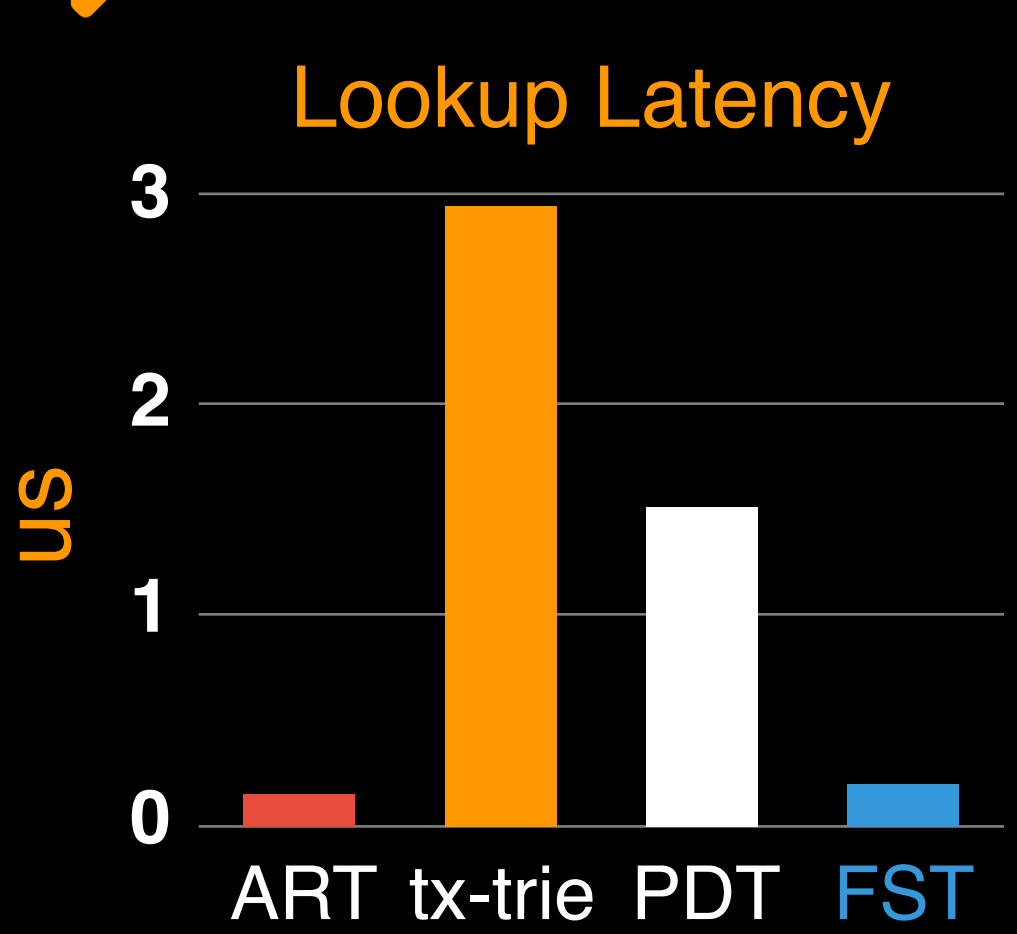
50M 64-bit integer keys

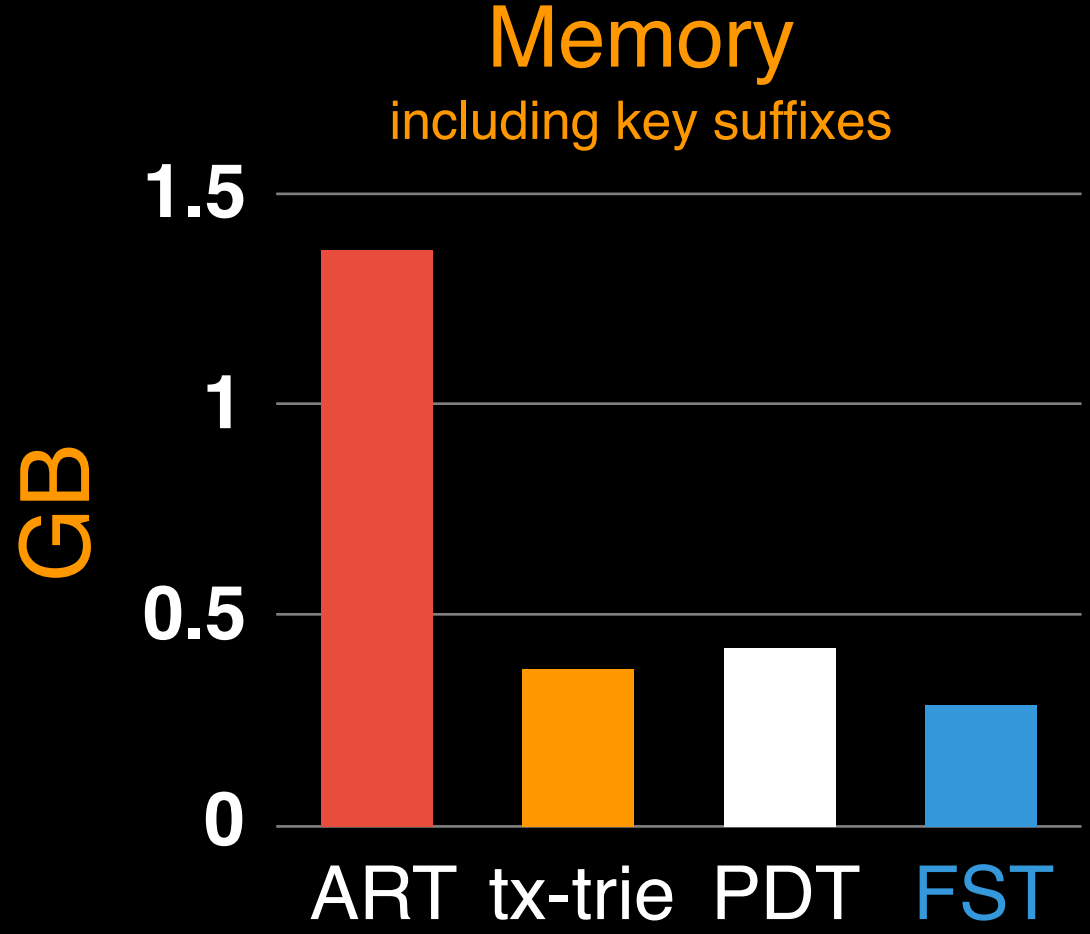




Fast Succinct Trie (FST) is fast and small

50M 64-bit integer keys

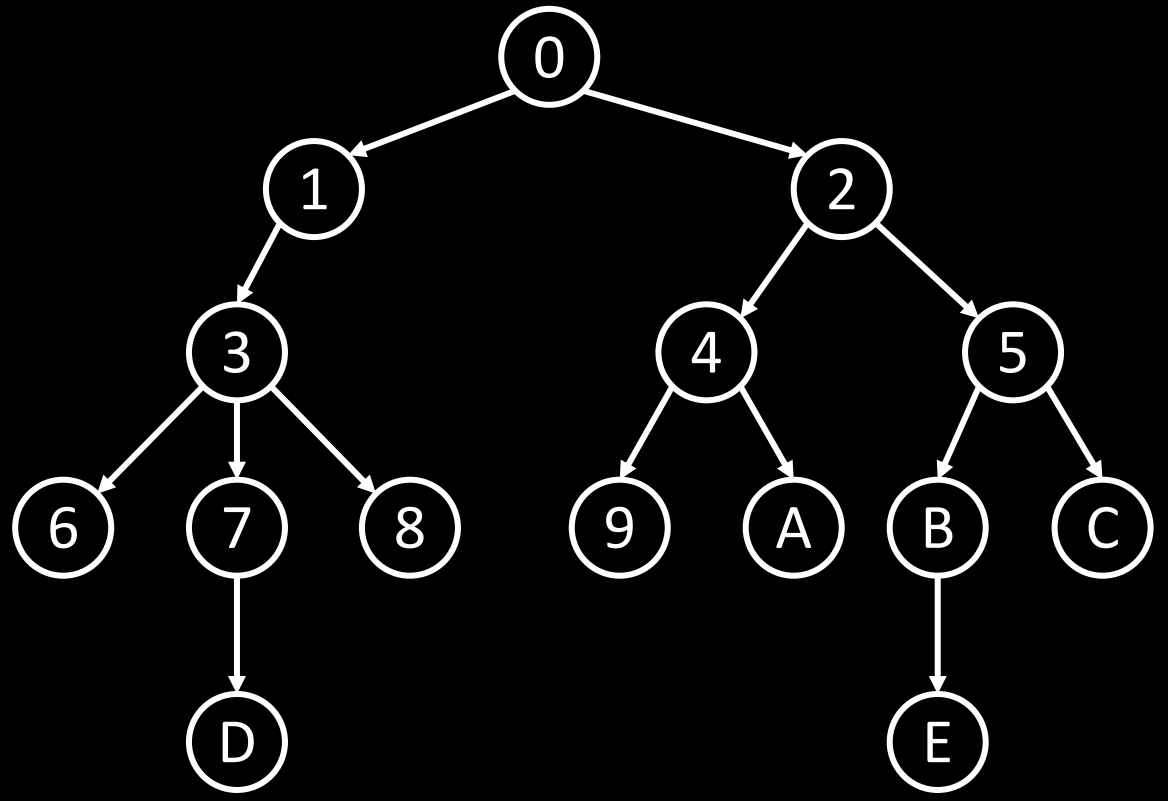




Encoding Mechanism

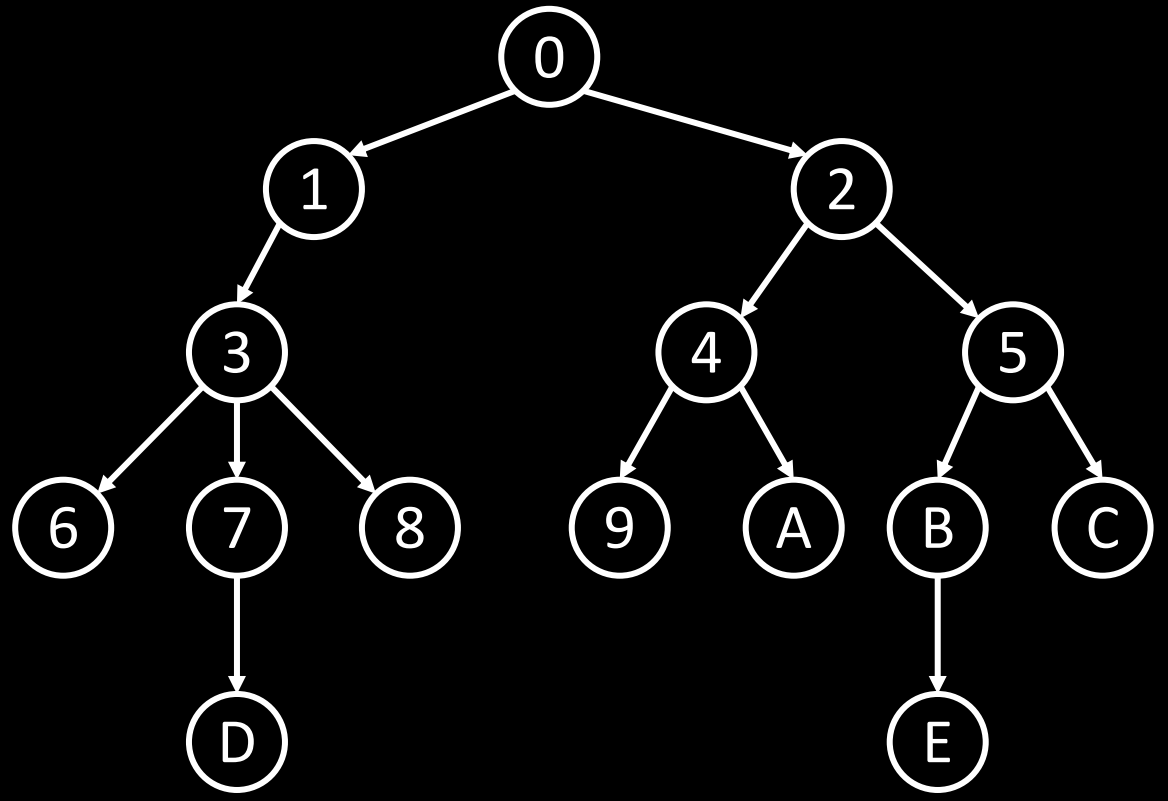
3 ways to succinctly encode ordinal trees

Ordinal tree: a rooted tree where each node can have an arbitrary # of children in order



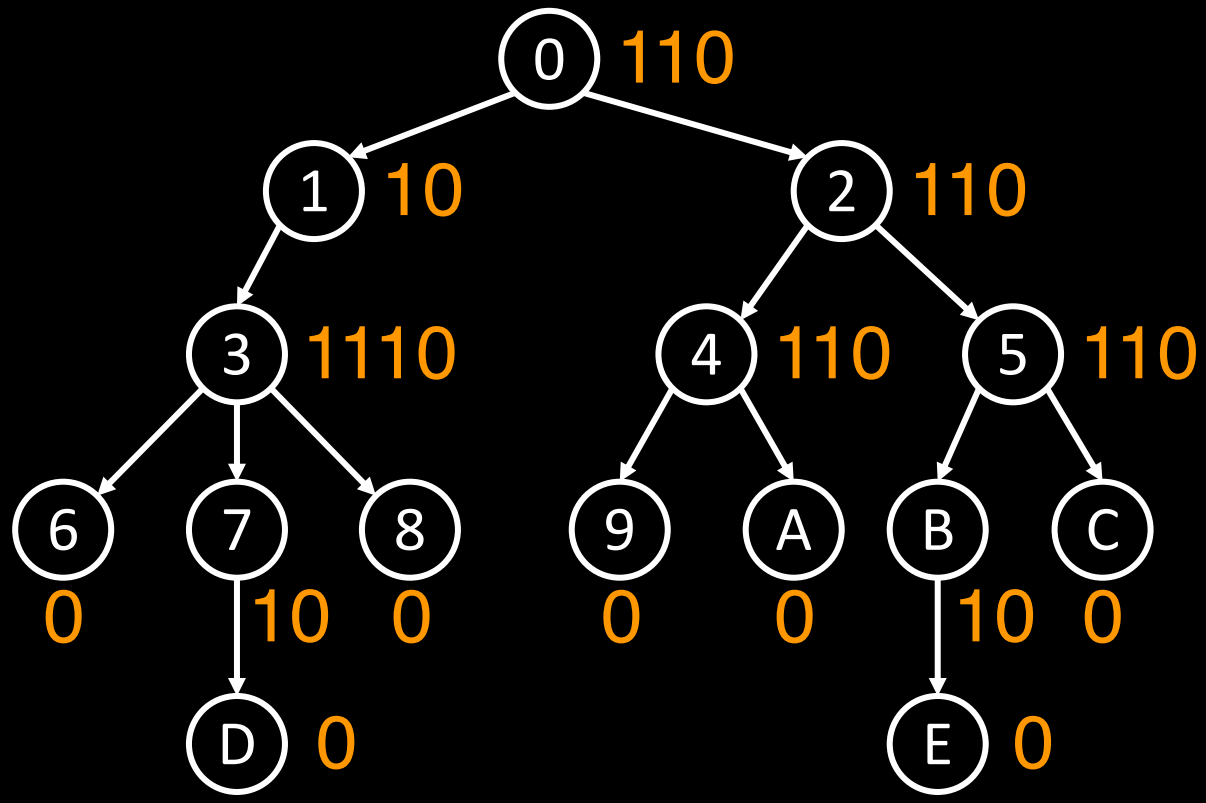
3 ways to succinctly encode ordinal trees

In-node ordinal tree: $C_n = \frac{1}{n+1}\binom{2n}{n}$ → $\approx 2n$ bits



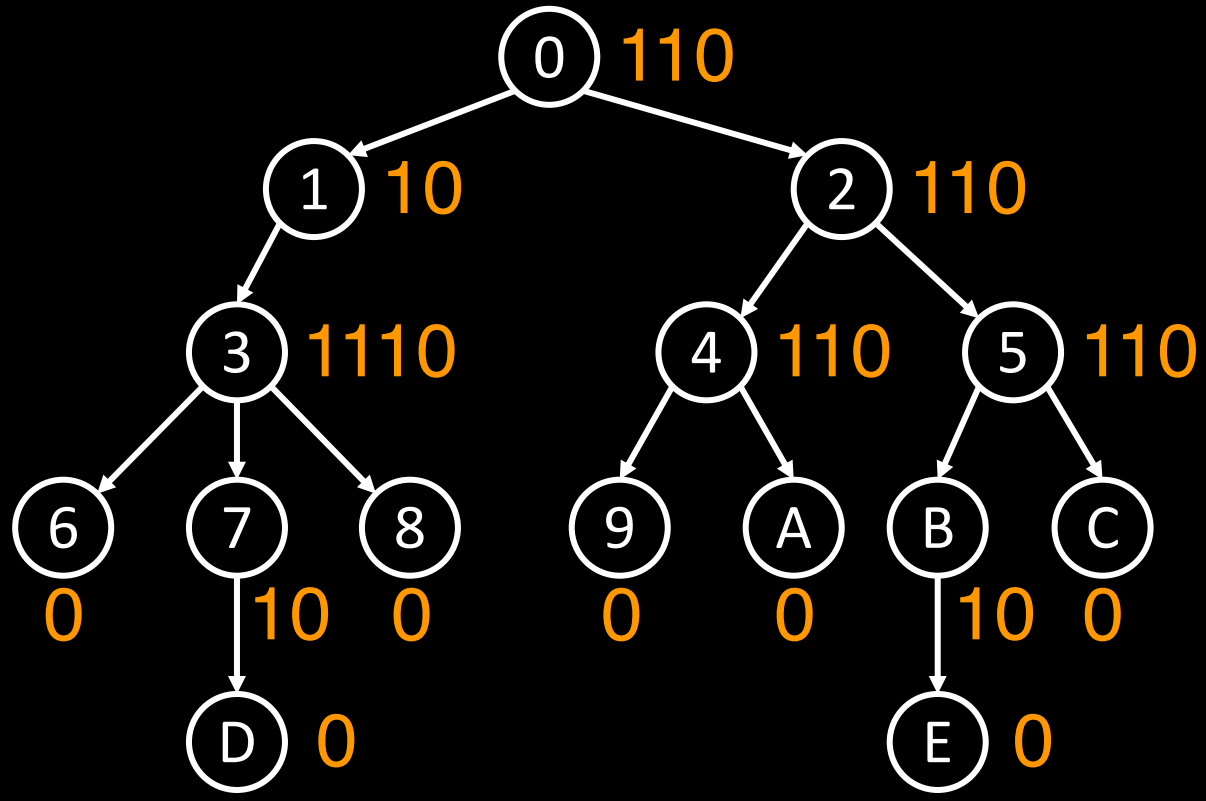
3 ways to succinctly encode ordinal trees

① **LOUDS:** level-ordered unary degree sequence



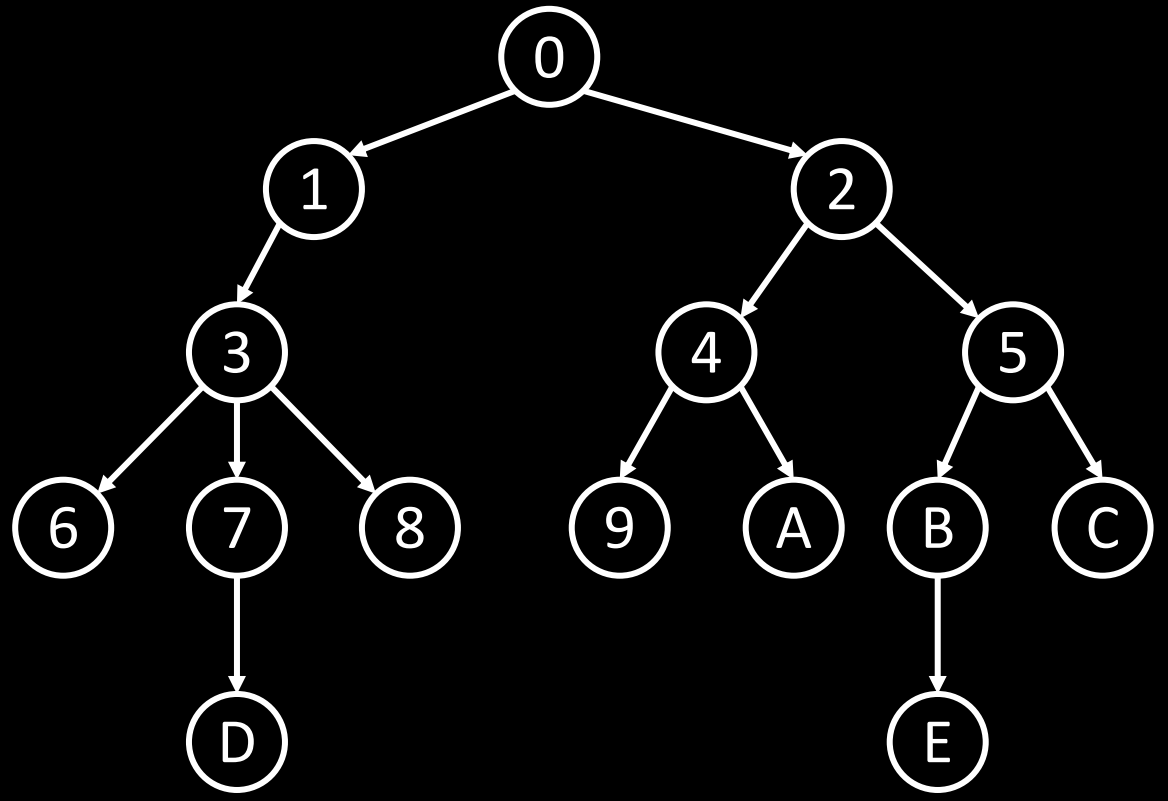
3 ways to succinctly encode ordinal trees

① **LOUDS:** 110 10 110 1110 110 110 0 10 0 0 0 10 0 0 0



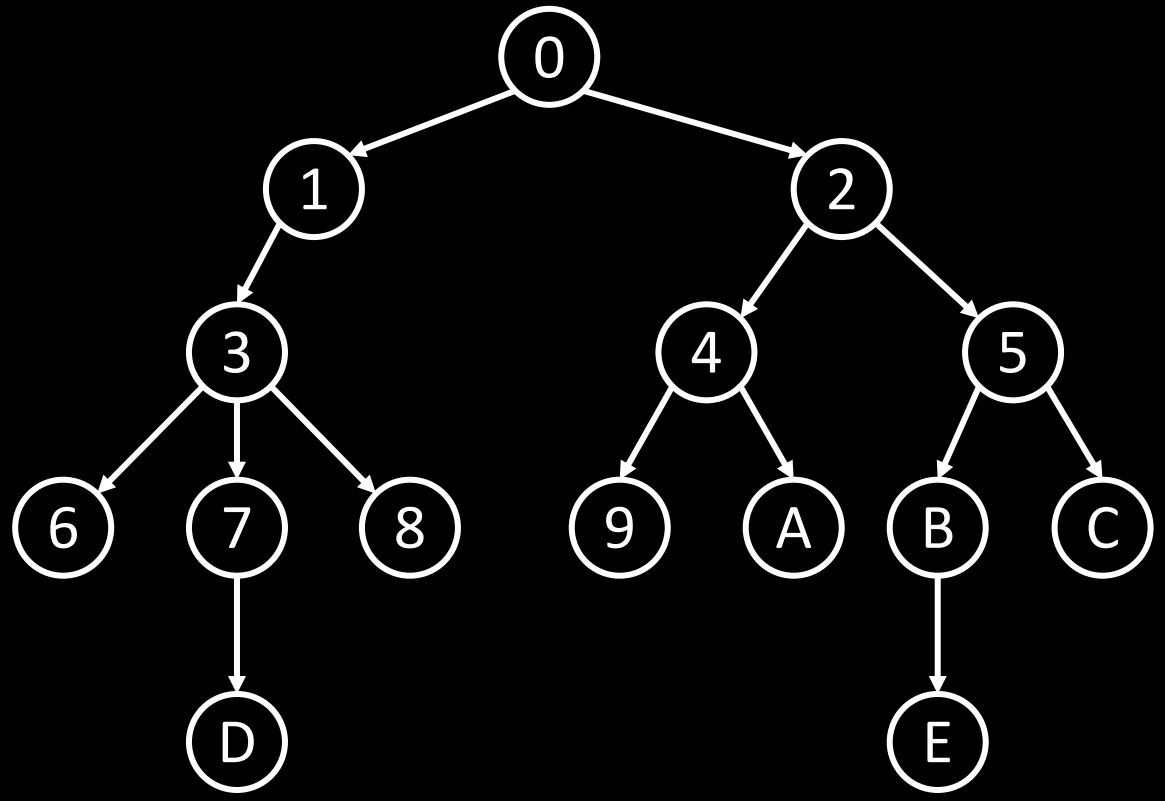
3 ways to succinctly encode ordinal trees

② **BP:** balanced parenthesis



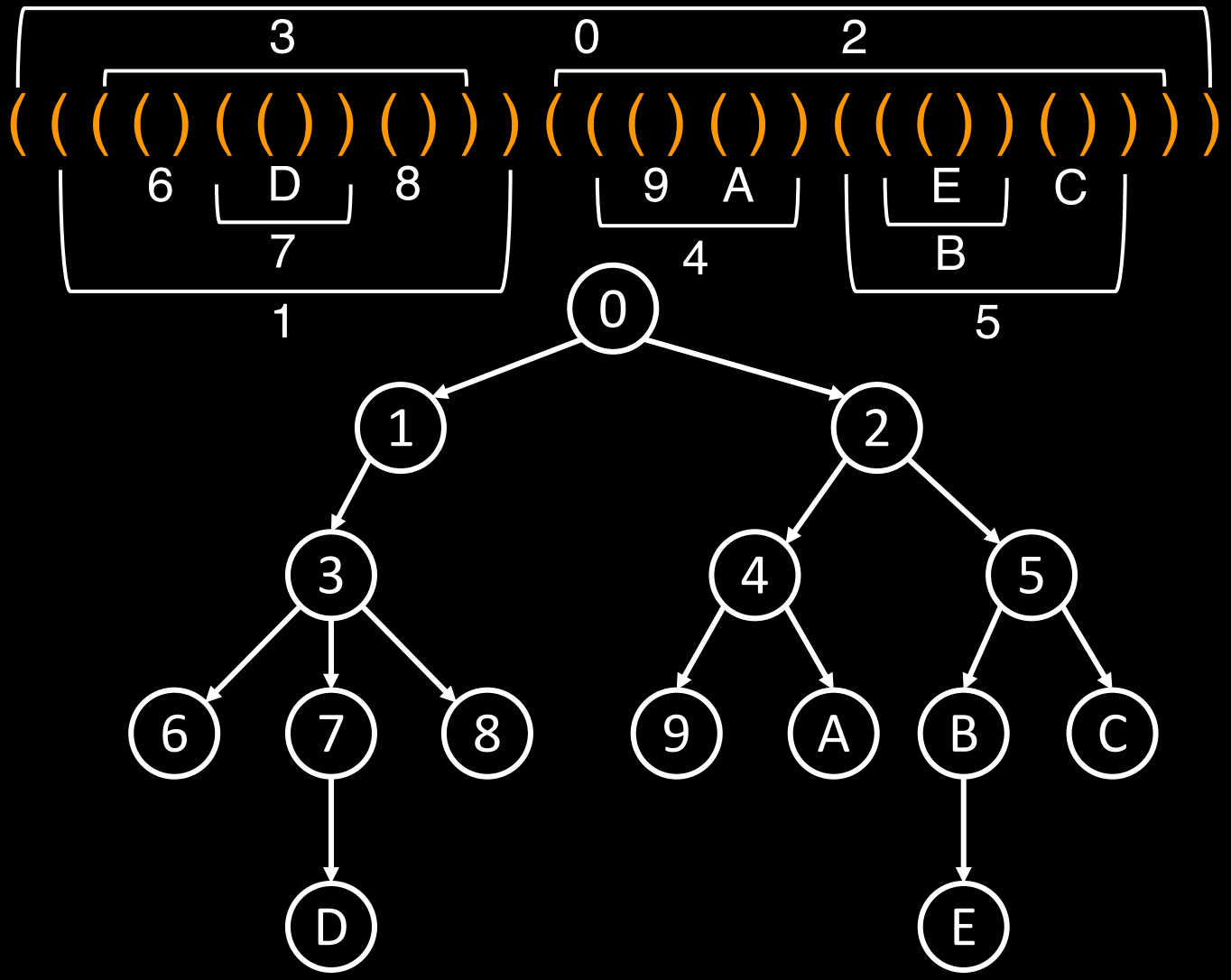
3 ways to succinctly encode ordinal trees

② **BP:** ((((()(()))()))((()())((()))()))))



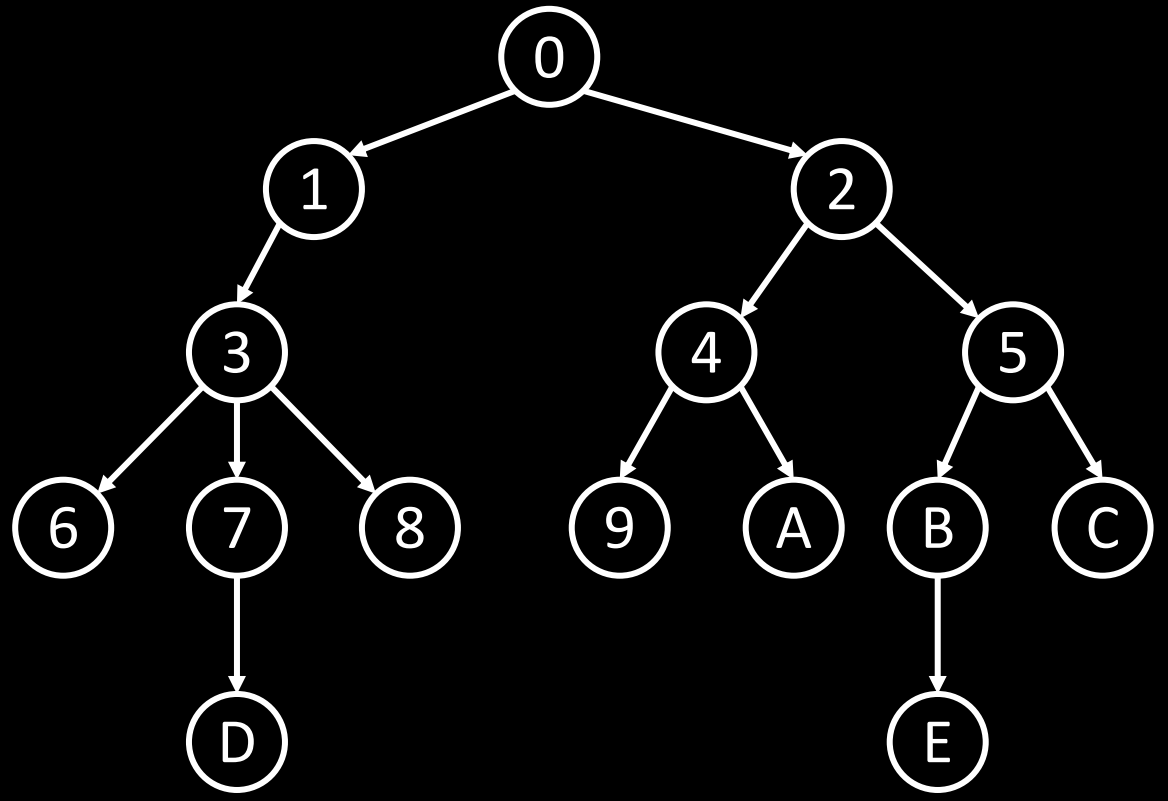
3 ways to succinctly encode ordinal trees

② **BP:** (((( ) (( )) ( ))) ((( ) ( )) ((( )) ( ))))



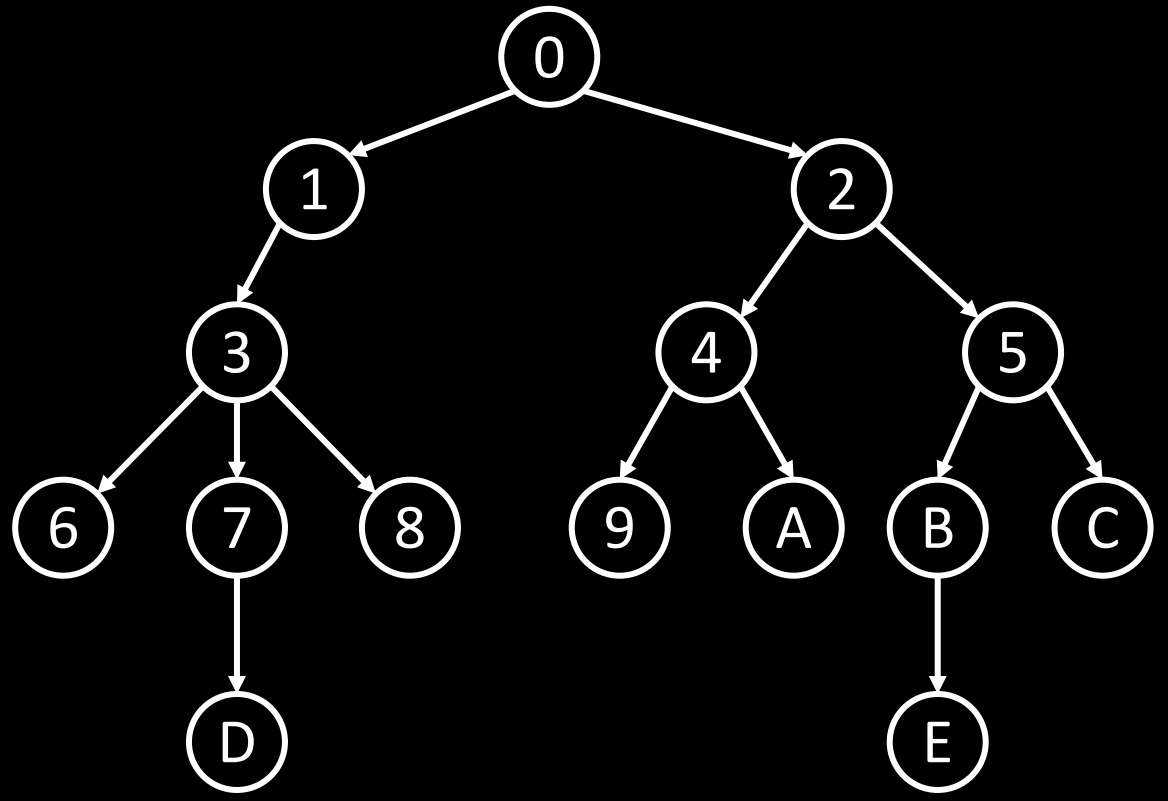
3 ways to succinctly encode ordinal trees

③ **DFUDS:** depth-first unary degree sequence



3 ways to succinctly encode ordinal trees

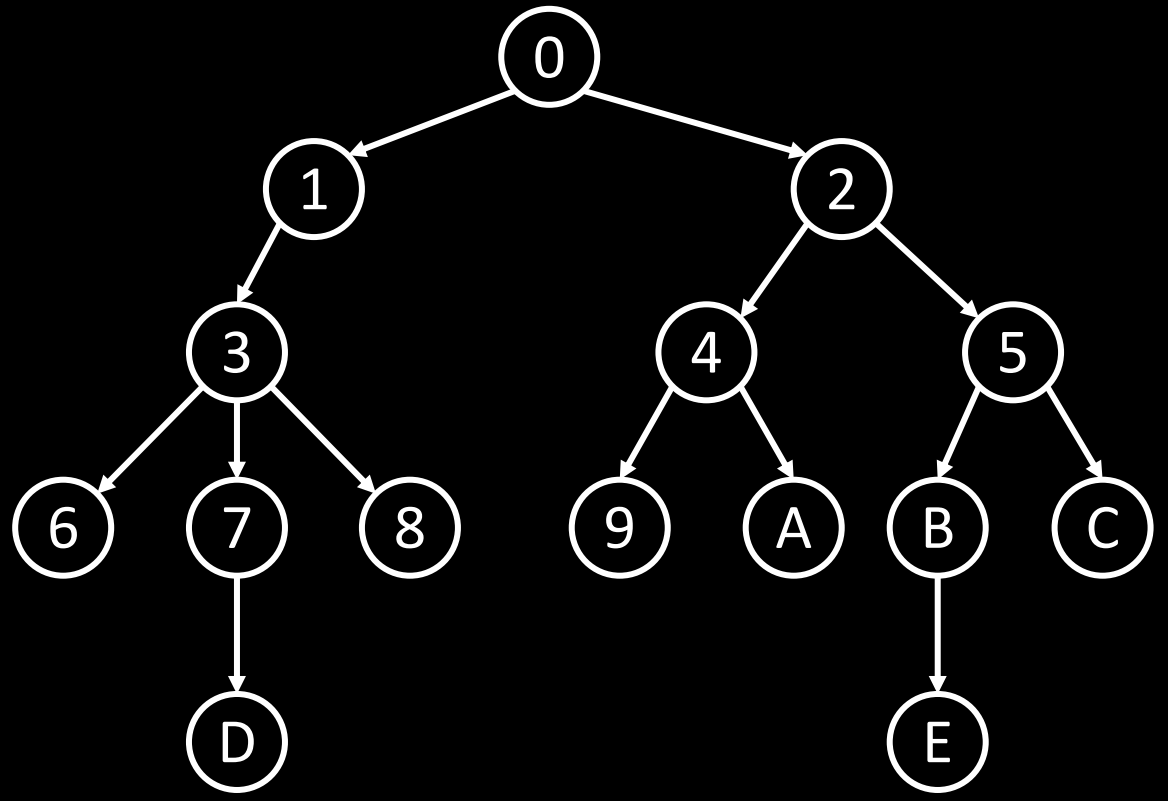
③ **DFUDS:** (( ) ( ) ((( )) ( ))) (( ) (( )) (( ) ( )))



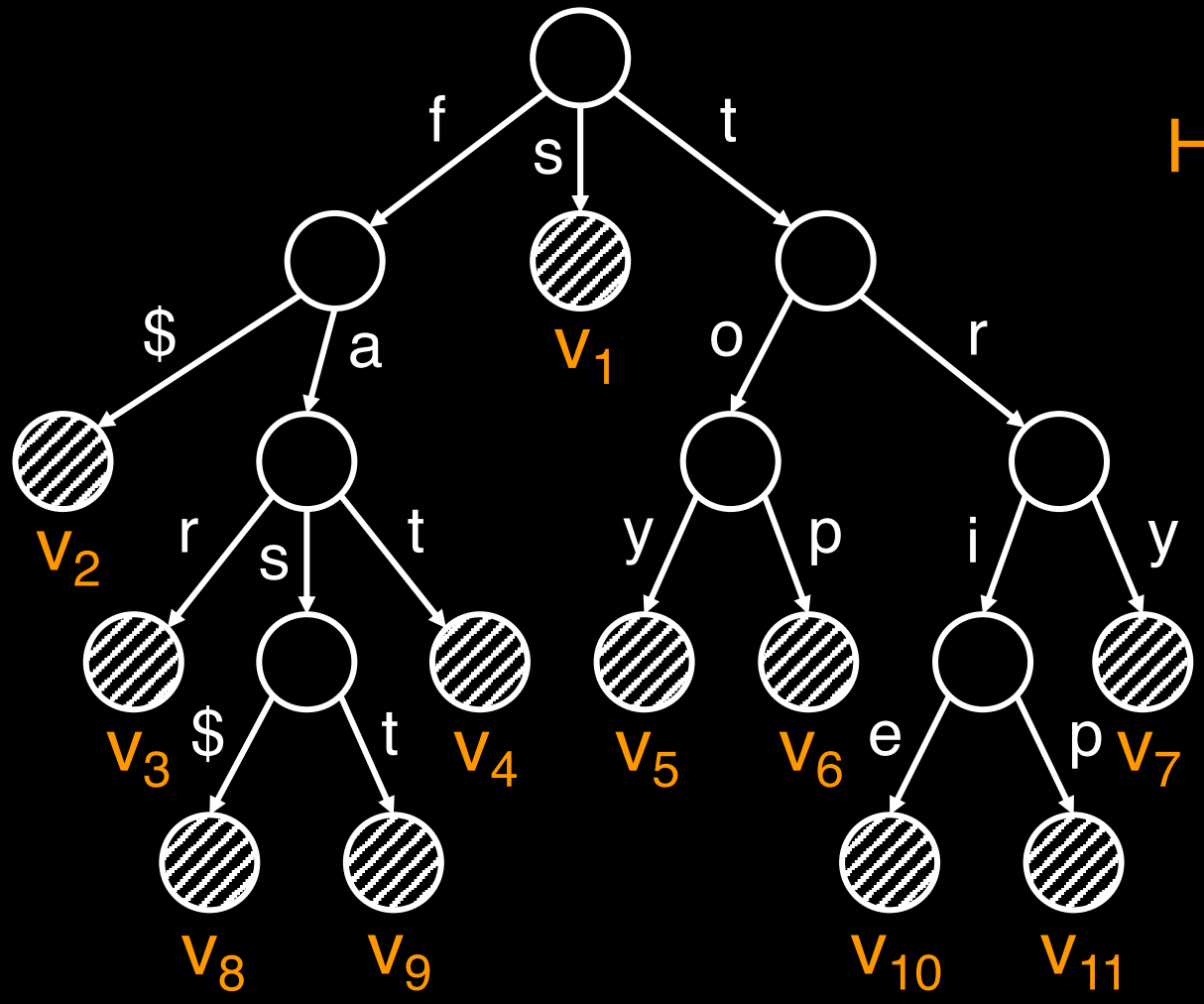
3 ways to succinctly encode ordinal trees

③ **DFUDS:** (( ) ( ) ((( )) ( ))) (( ) (( ))) (( ) ( )))

0 1 3 6 7 D 8 2 4 9 A 5 B E C



LOUDS-Sparse: succinctly encode tries



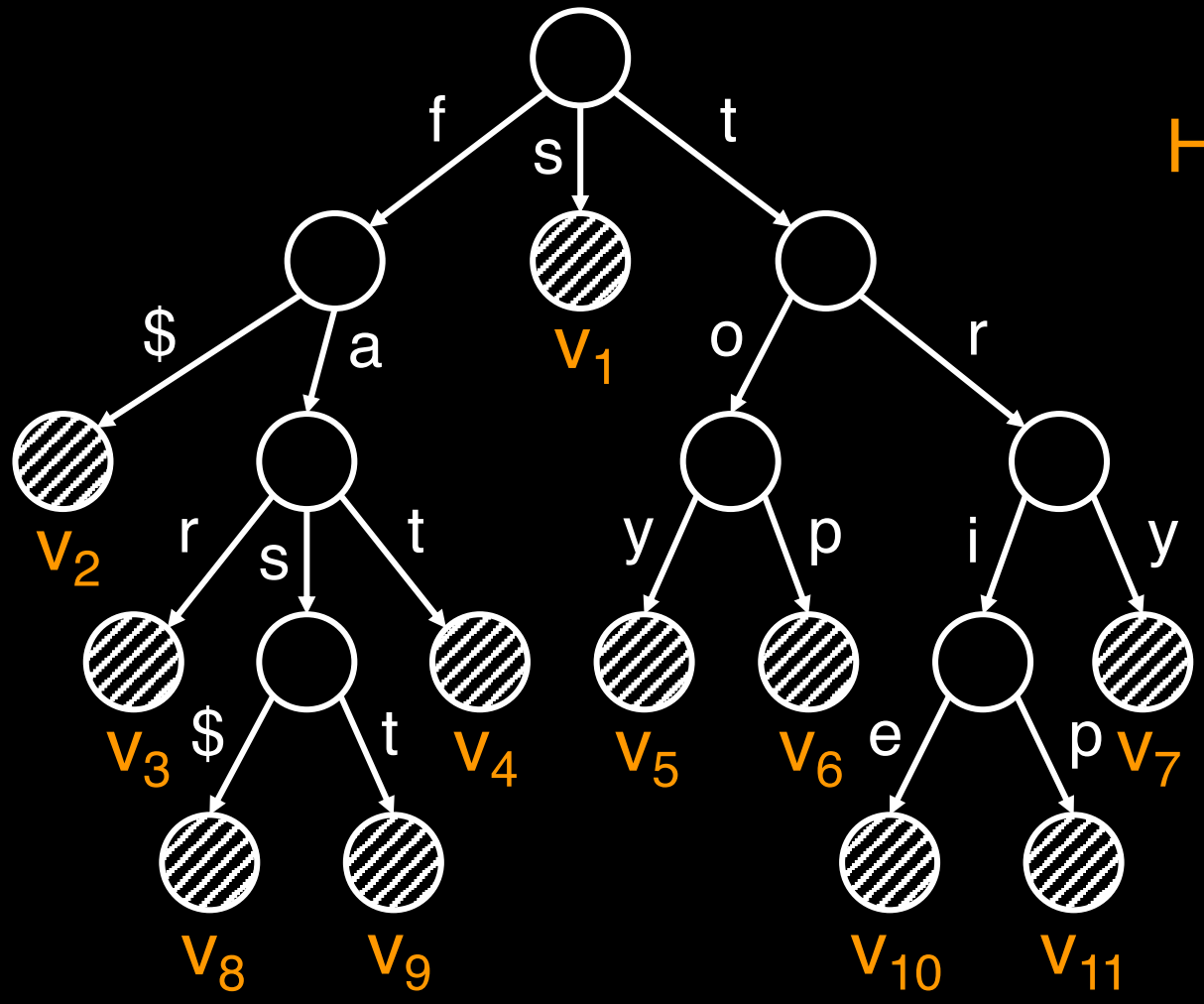
L: f s t $ a o r r s t y p i y $ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

V: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}

LOUDS-Sparse: succinctly encode tries



L: f s t $ a o r r s t y p i y $ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

V: v_1 v_2 v_3 $v_4 v_5 v_6$ $v_7 v_8 v_9 v_{10} v_{11}$

Why LOUDS?

1. **Fast tree nav.**
2. **Good label locality**
3. **Easy implementation**

Rank & select on bit-vectors

```
      0         5         10        15
bv:   1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0
```

rank(bv, i) = # of 1's in bv up to position i

select(bv, i) = position of the ith 1 in bv

Examples: rank(bv, 7) = 4

select(bv, 7) = 14

Compute rank & select in constant time

The classic algorithm for computing rank

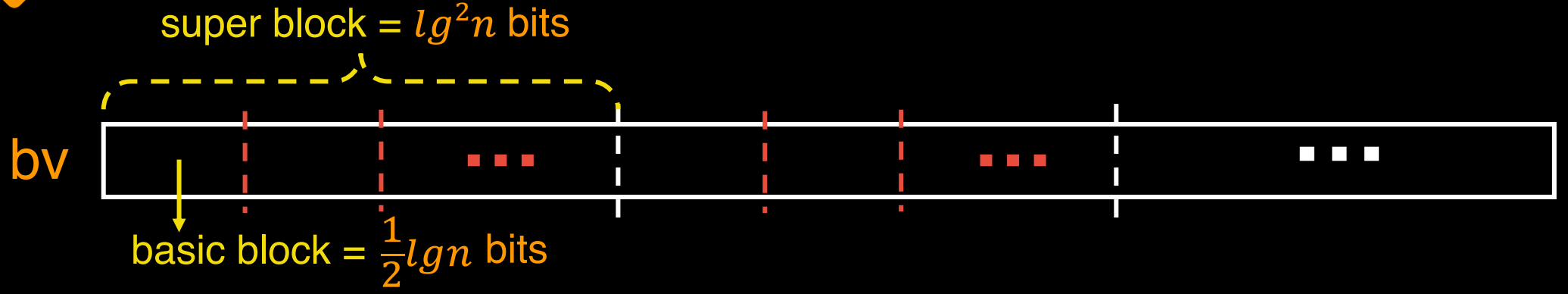
bv 

Compute rank & select in constant time

The classic algorithm for computing rank

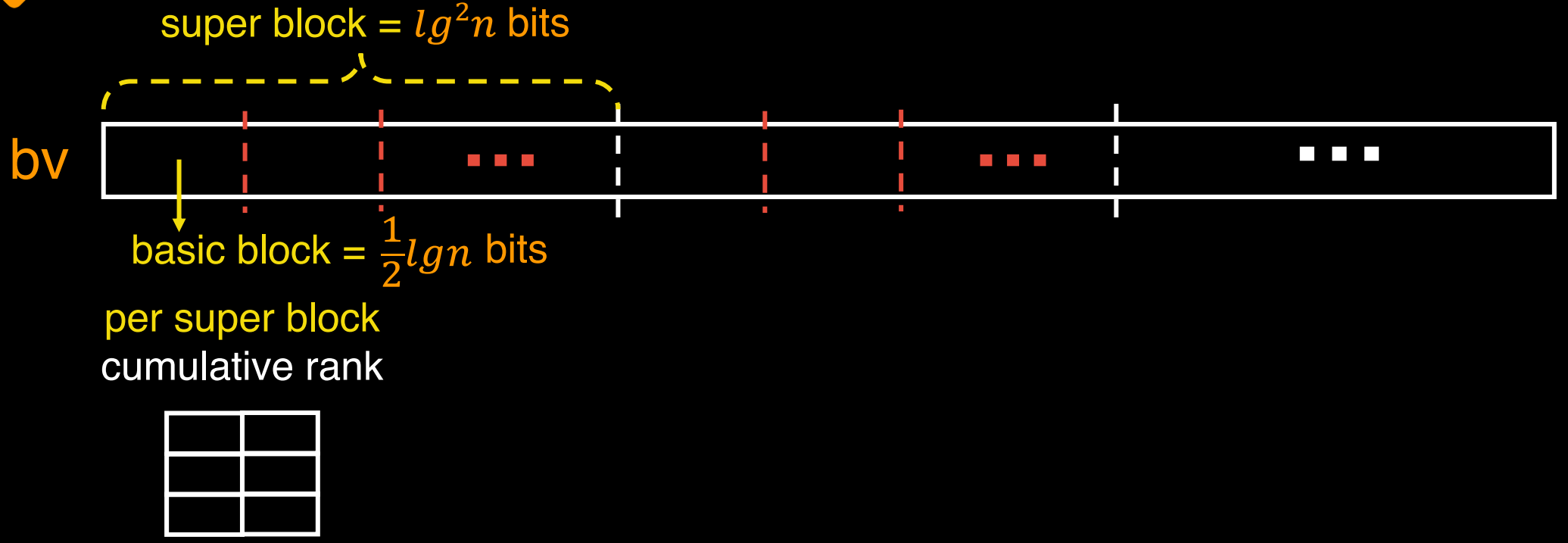
Compute rank & select in constant time

The classic algorithm for computing rank



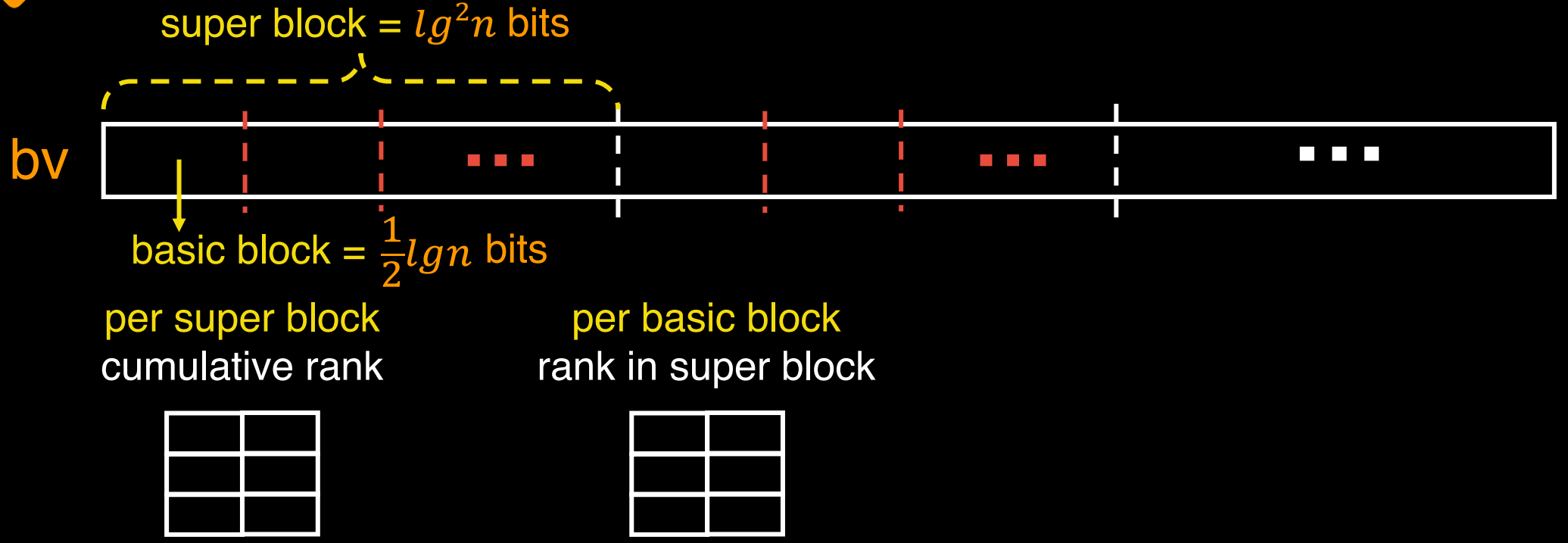
Compute rank & select in constant time

The classic algorithm for computing rank



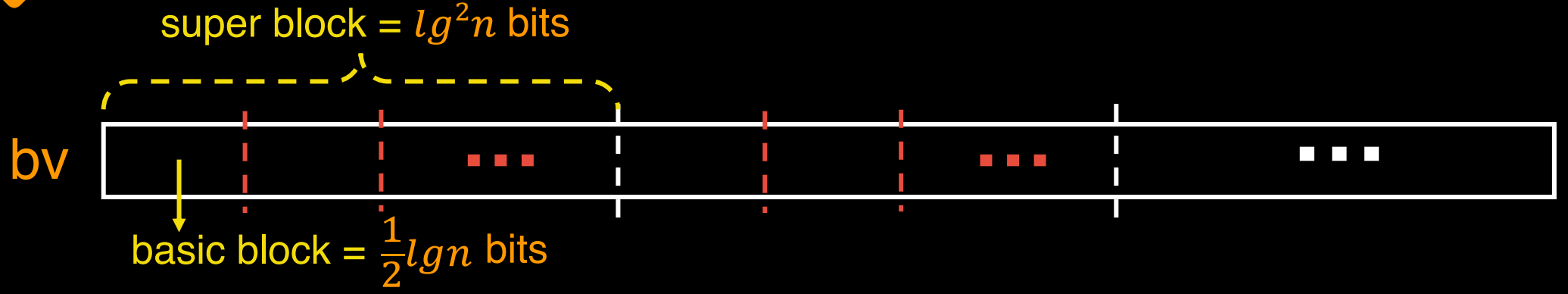
Compute rank & select in constant time

The classic algorithm for computing rank

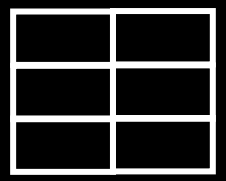


Compute rank & select in constant time

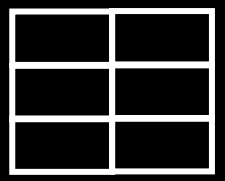
The classic algorithm for computing rank



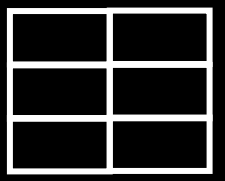
per super block
cumulative rank



per basic block
rank in super block

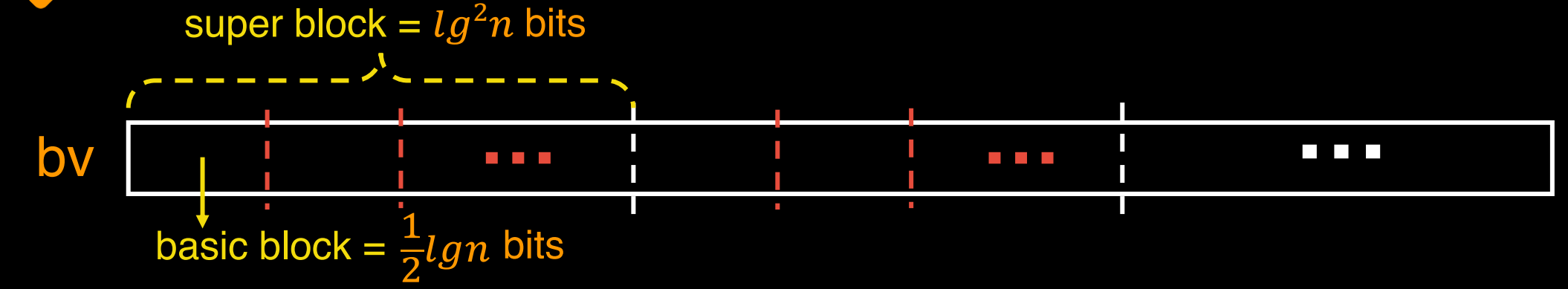


within super block
all possible queries



Compute rank & select in constant time

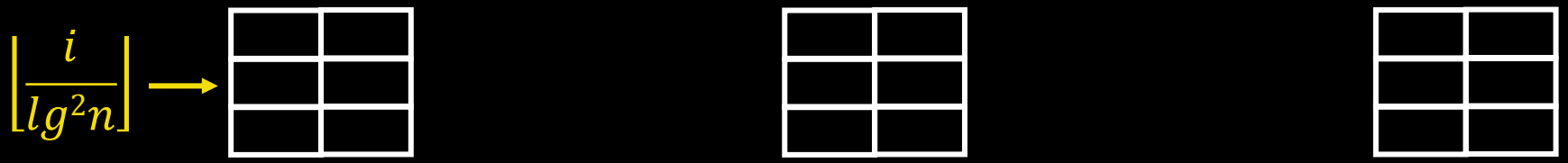
The classic algorithm for computing rank



per super block
cumulative rank

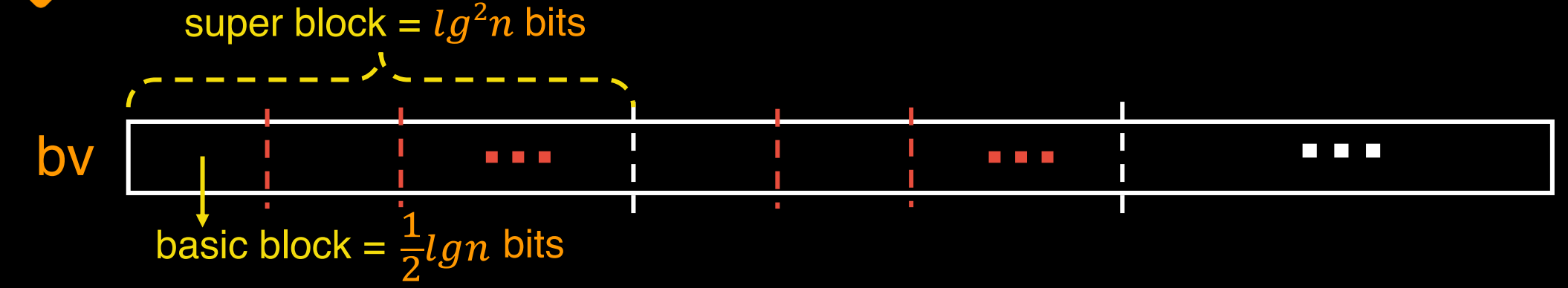
per basic block
rank in super block

within super block
all possible queries



Compute rank & select in constant time

The classic algorithm for computing rank



per super block
cumulative rank

$\left\lfloor \frac{i}{lg^2n} \right\rfloor$ →

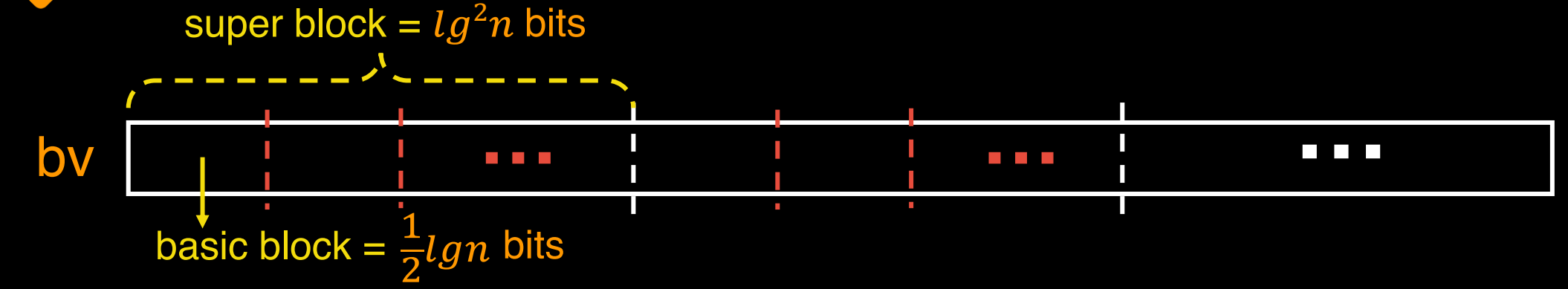
per basic block
rank in super block

$\left\lfloor \frac{i}{\frac{1}{2}lgn} \right\rfloor$ →

within super block
all possible queries

Compute rank & select in constant time

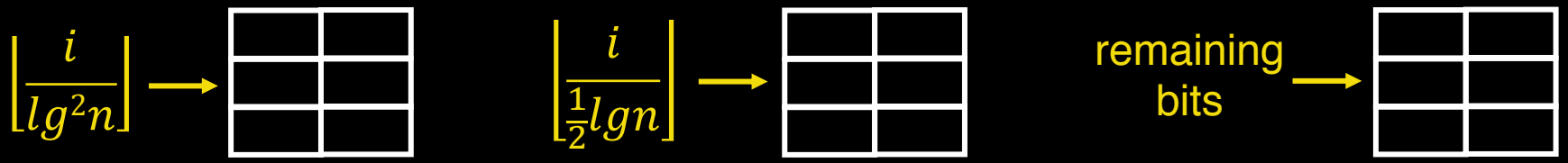
The classic algorithm for computing rank



per super block
cumulative rank

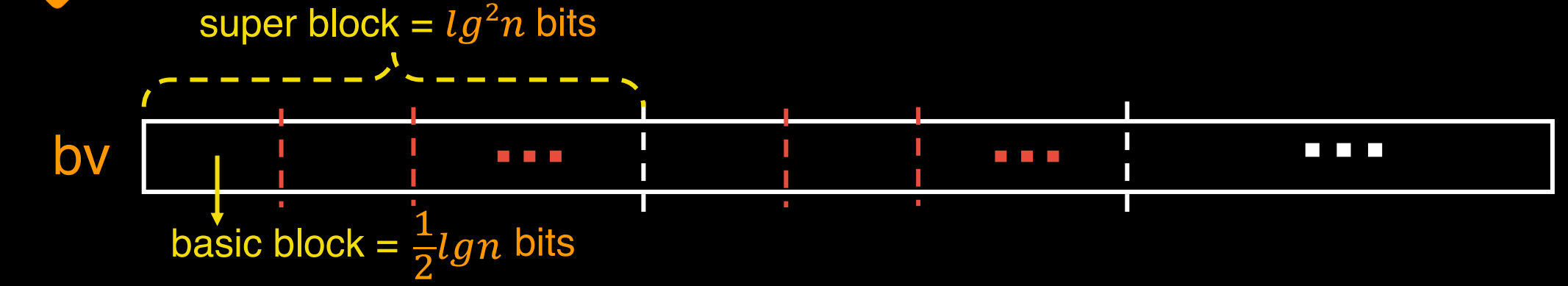
per basic block
rank in super block

within super block
all possible queries



Compute rank & select in constant time

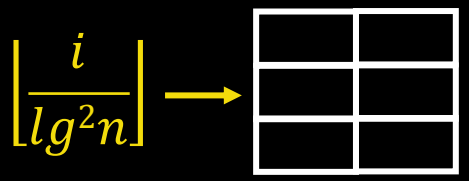
The classic algorithm for computing rank

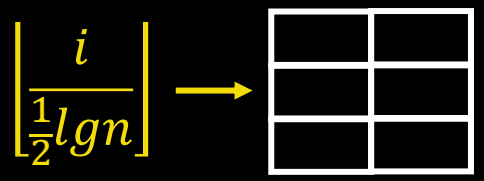


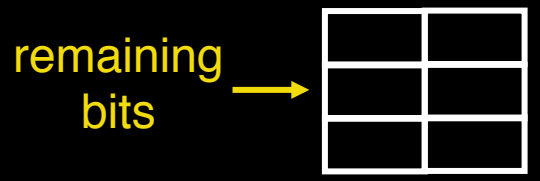
per super block
cumulative rank

per basic block
rank in super block

within super block
all possible queries



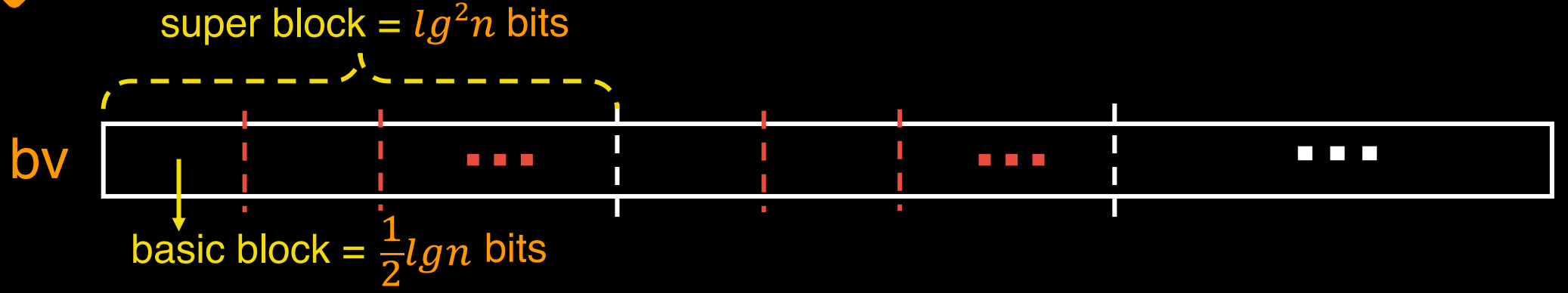




O(1) time

Compute rank & select in constant time

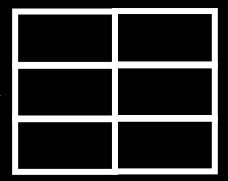
The classic algorithm for computing rank



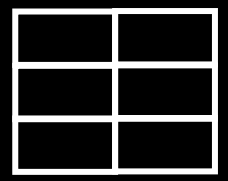
per super block
cumulative rank

per basic block
rank in super block

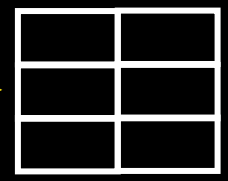
within super block
all possible queries

$\left\lfloor \frac{i}{lg^2n} \right\rfloor$ → 

space: $O(\frac{n}{lgn})$ +

$\left\lfloor \frac{i}{\frac{1}{2}lgn} \right\rfloor$ → 

$O(\frac{n}{lgn}\ lglgn)$ +

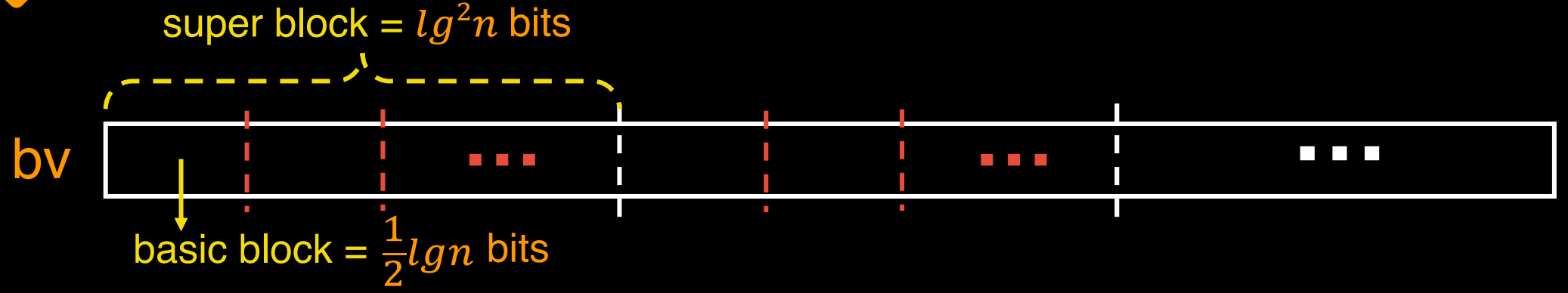
remaining bits → 

$O(\sqrt{n}\ lgn\ lglgn) = o(n)$

O(1) time

Compute rank & select in constant time

The classic algorithm for computing rank

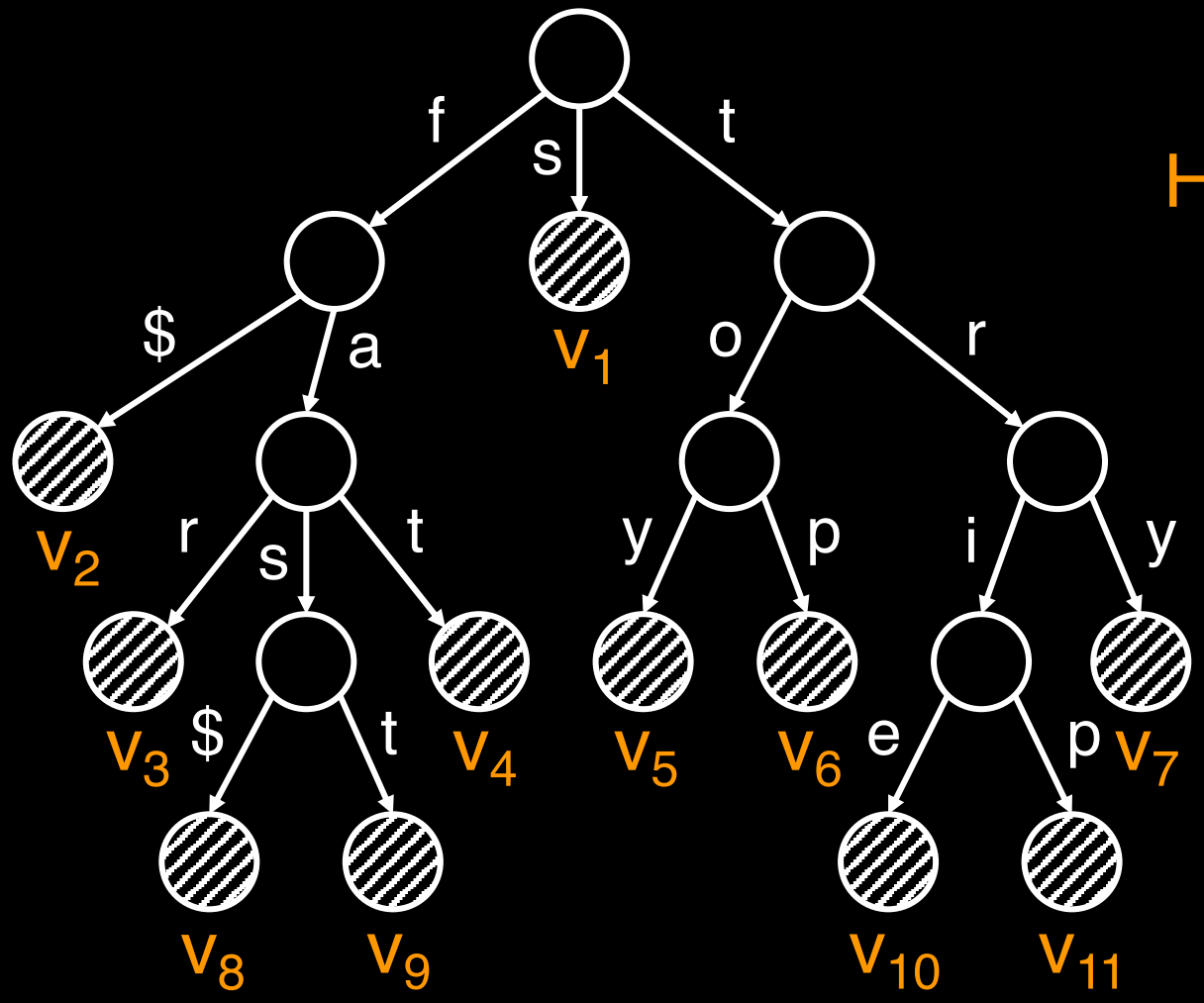


per super block	per basic block	within super block
cumulative rank	rank in super block	all possible queries
$\lfloor \frac{i}{lg^2n} \rfloor$ →	$\lfloor \frac{i}{\frac{1}{2}lgn} \rfloor$ →	remaining bits →
space: $O(\frac{n}{lgn})$	+ $O(\frac{n}{lgn}\ lglgn)$	+ $O(\sqrt{n}\ lgn\ lglgn)$ = $o(n)$

O(1) time

Select is similar but trickier, often based on rank structures

Tree navigation relies on rank & select



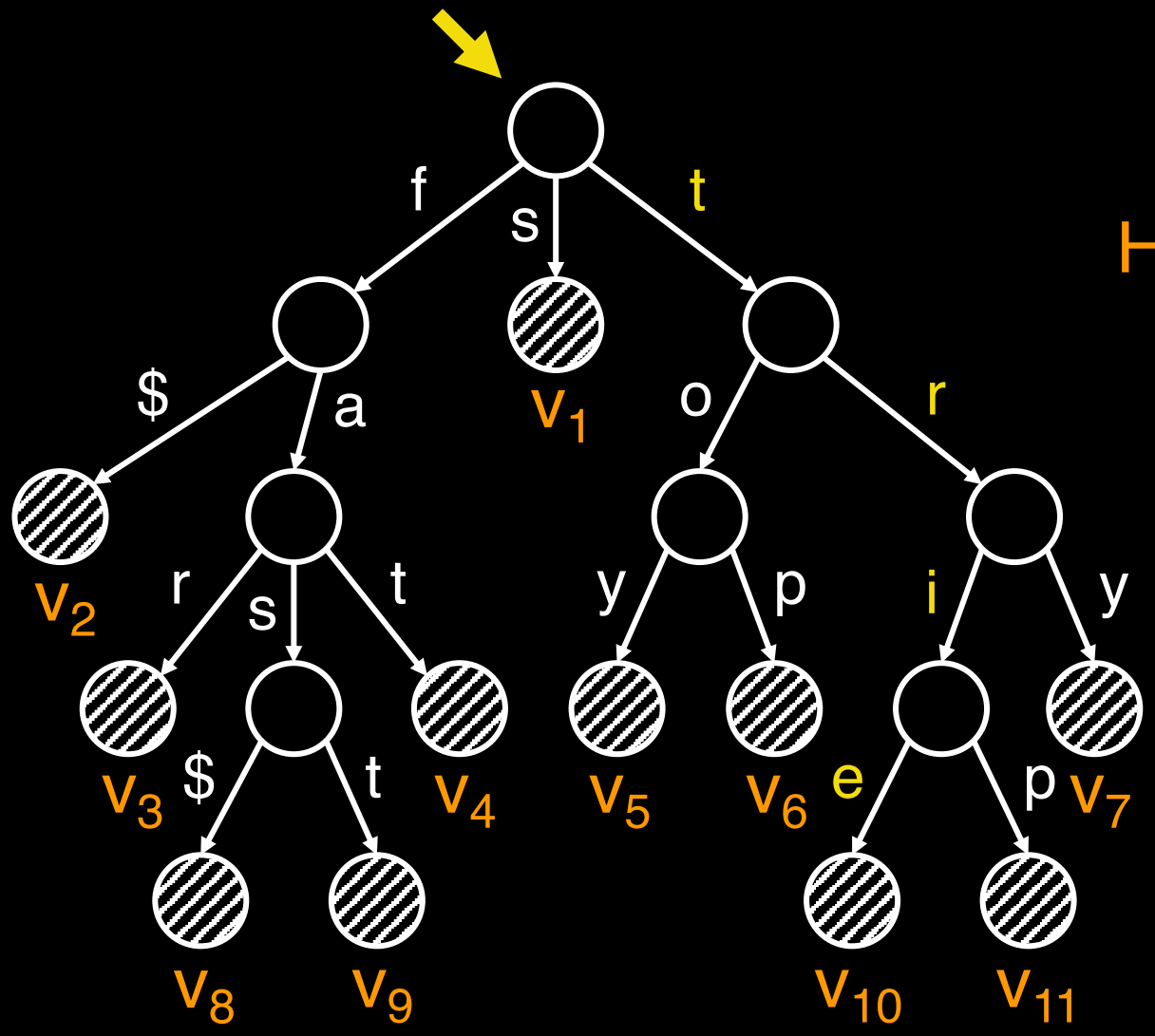
	0					5					10					15			
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p	
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	1	0	0	0	0
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	
V:		v_1	v_2				v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}				

child(i) = select(S, rank(HC, i)+1)

parent(i) = select(S, rank(S, i)-1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



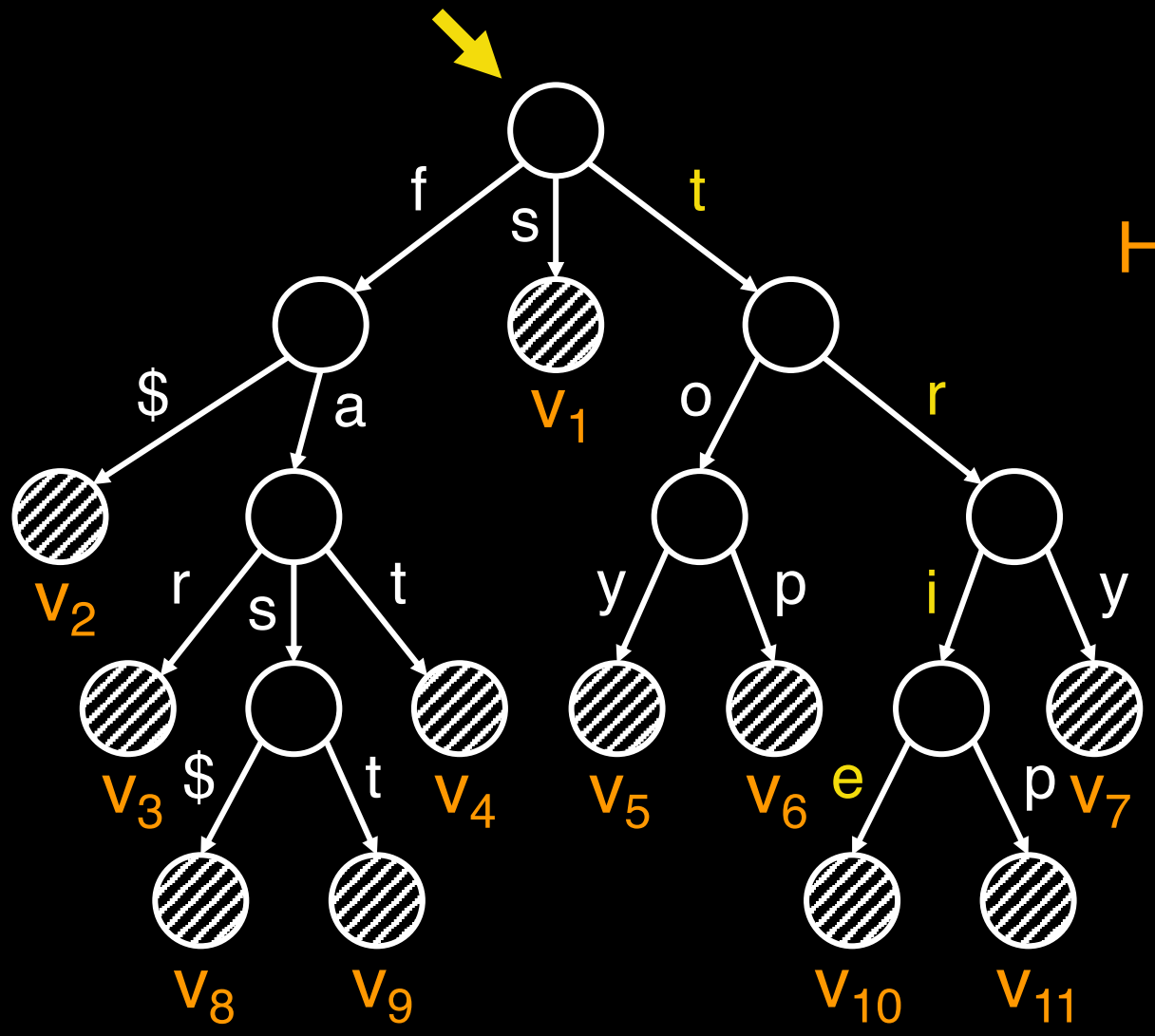
	0				5				10				15			
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0	0	0	
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0		
V:		v_1		v_2				v_3		v_4	v_5	v_6		v_7	v_8	v_9	v_{10}	v_{11}

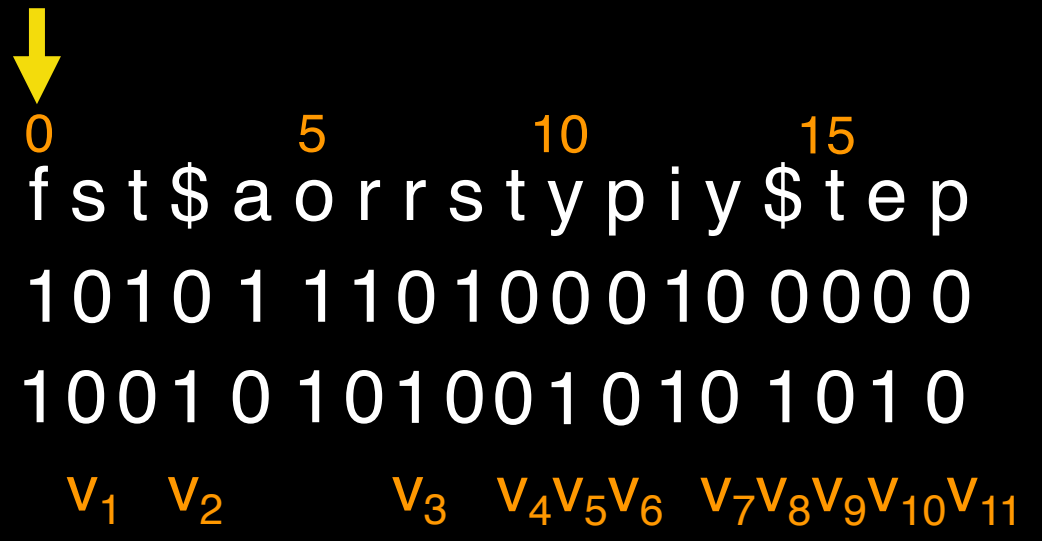
child(i) = select(S, rank(HC, i)+1)

parent(i) = select(S, rank(S, i)-1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select

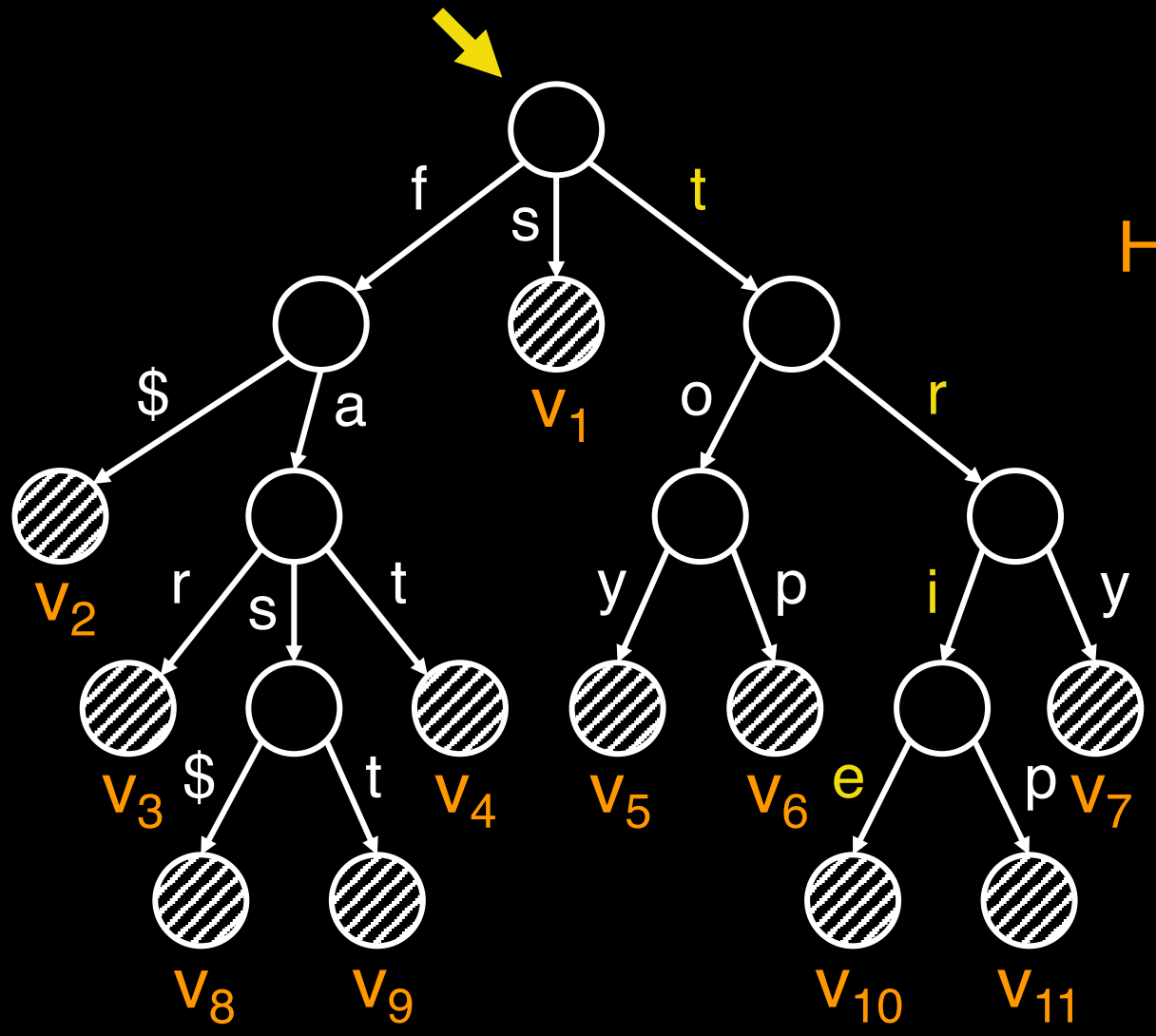




child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select

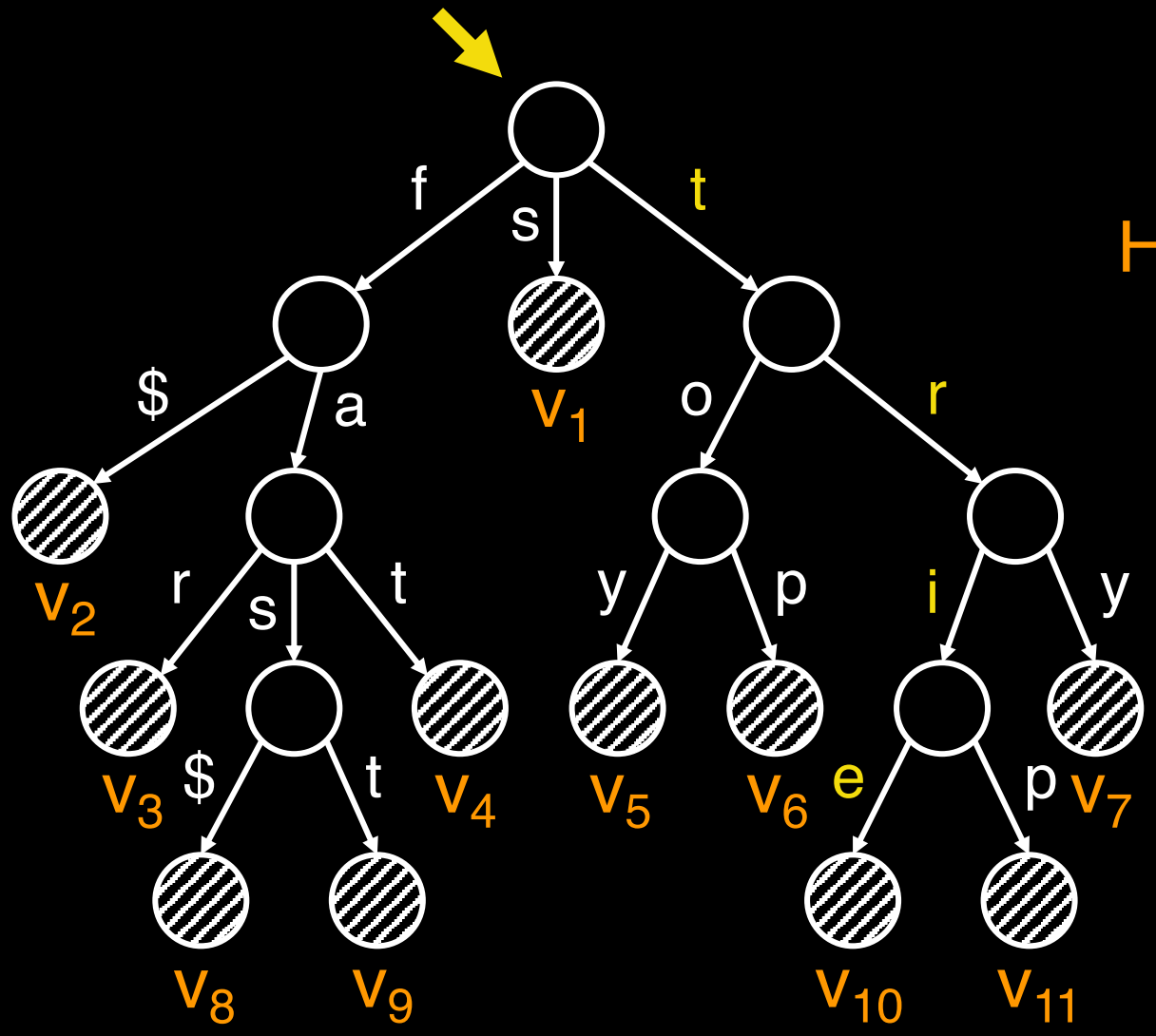


	0				5				10				15			
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0	0	0	
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0		
V:		v_1	v_2				v_3		v_4	v_5	v_6		v_7	v_8	v_9	v_{10}	v_{11}	

child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select

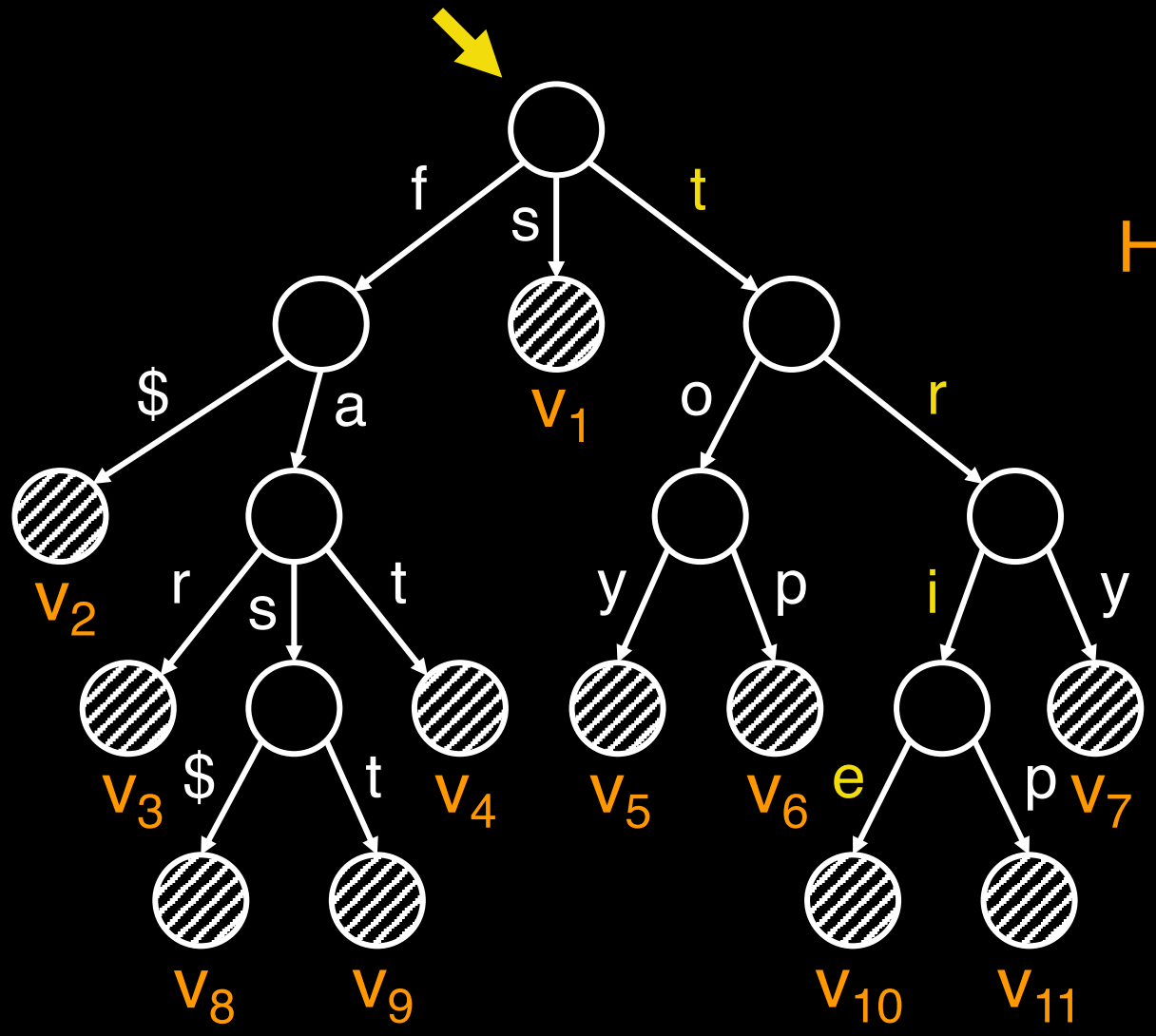


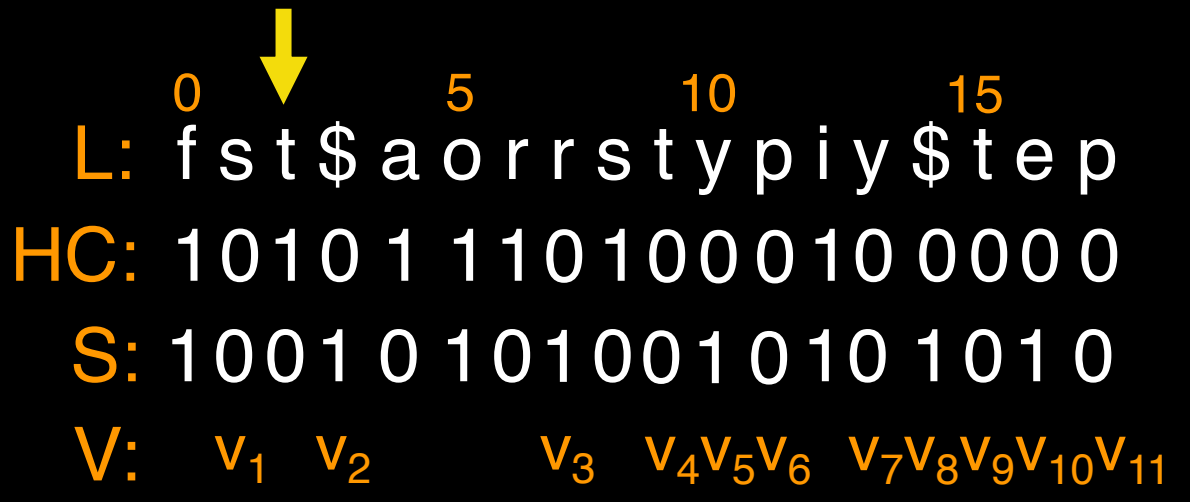
	0					5					10					15			
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p	
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	1	0	0	0	0
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	
V:		v_1		v_2					v_3		v_4	v_5	v_6		v_7	v_8	v_9	v_{10}	v_{11}

child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



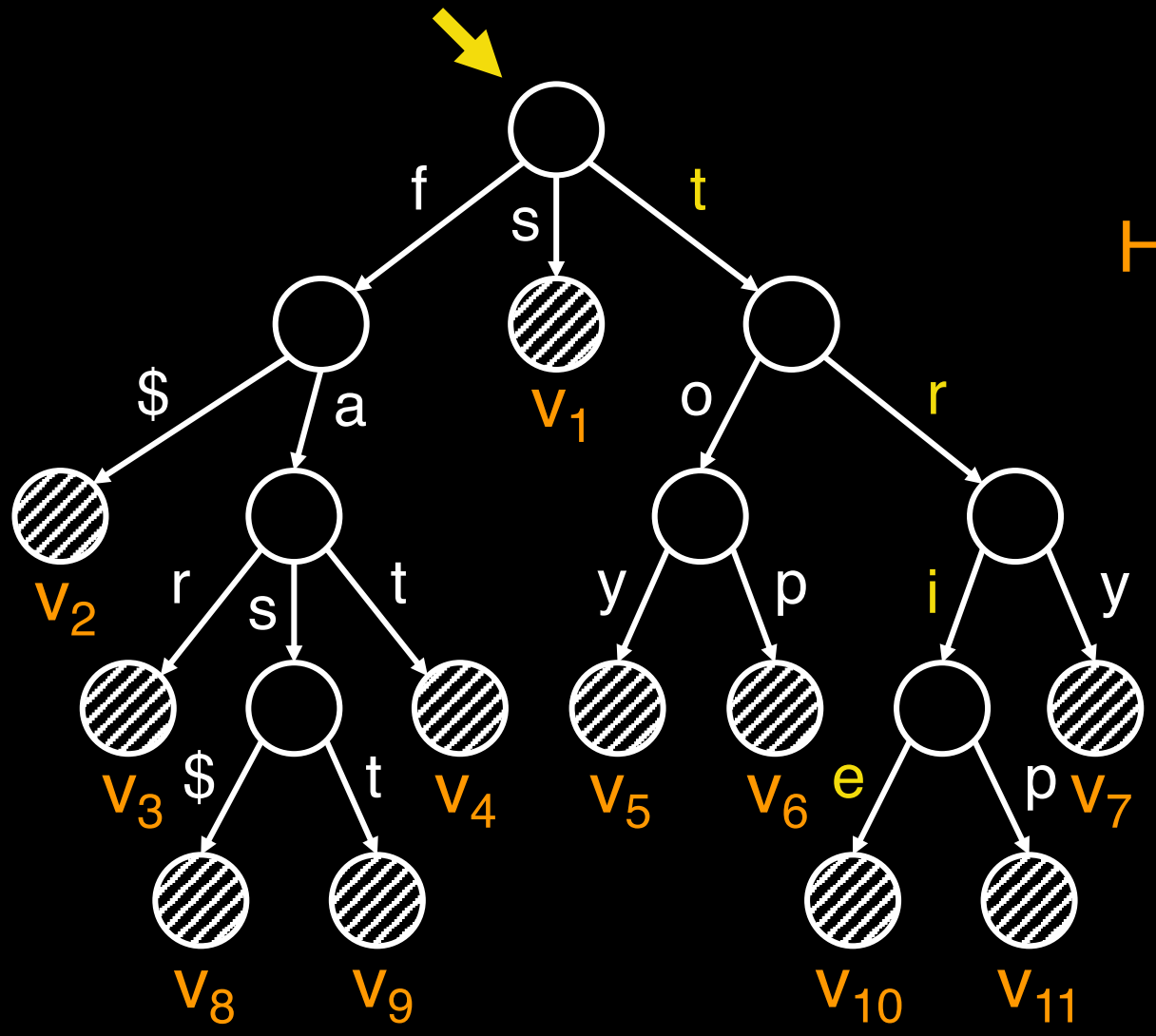


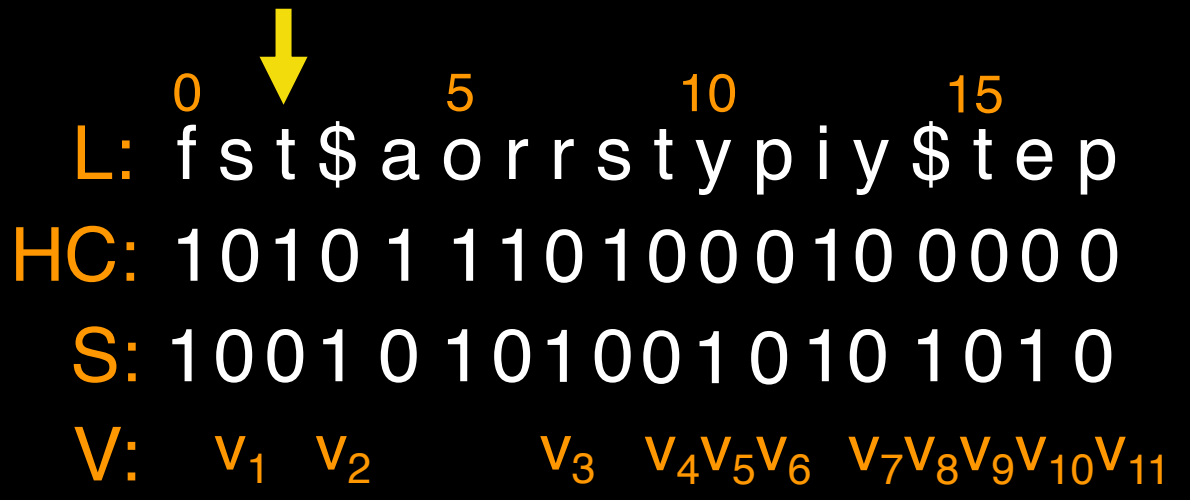
child(i) = select(S, rank(HC, i)+1)

2 2

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



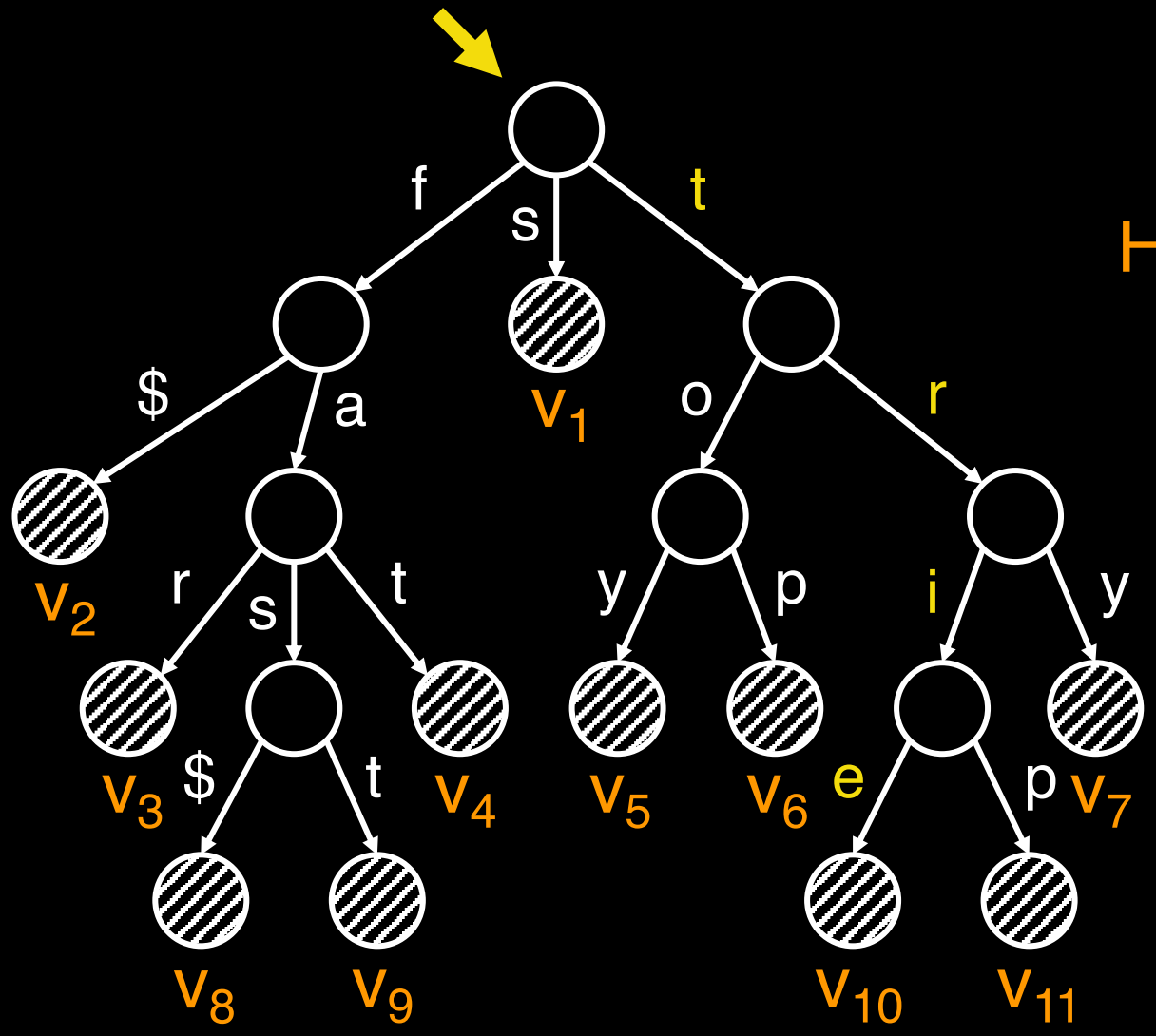


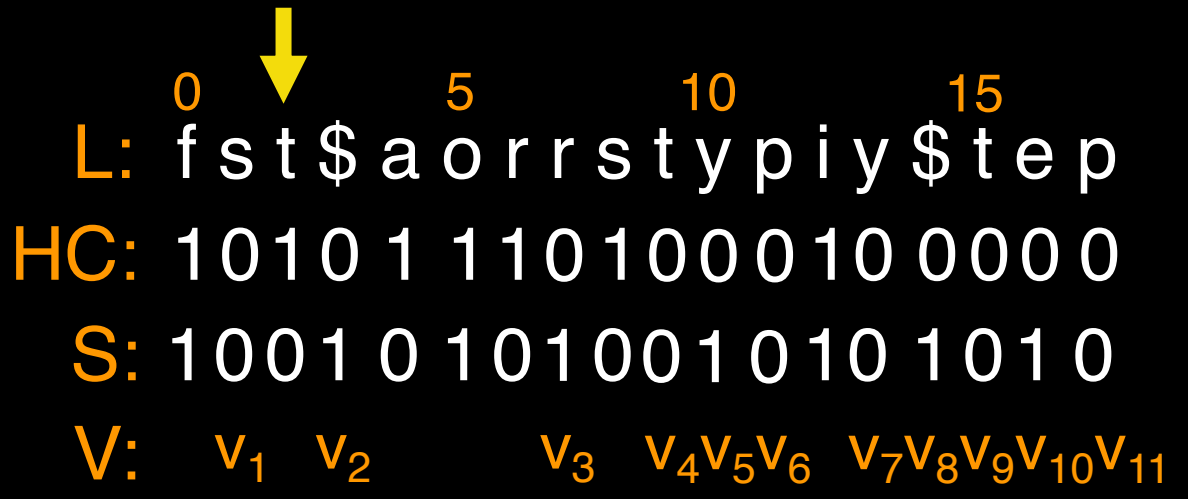
child(i) = select(S, rank(HC, i)+1)

3

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



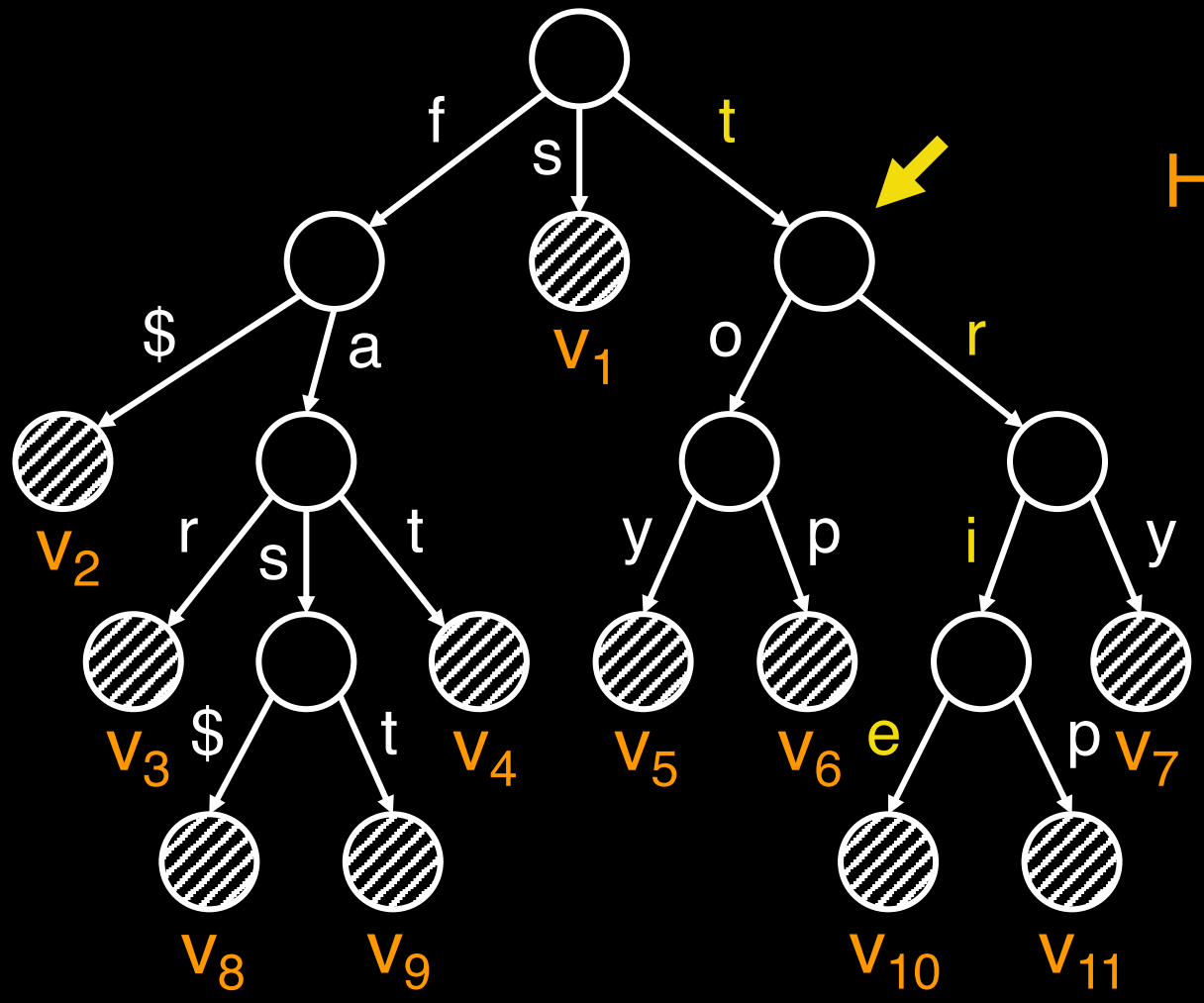


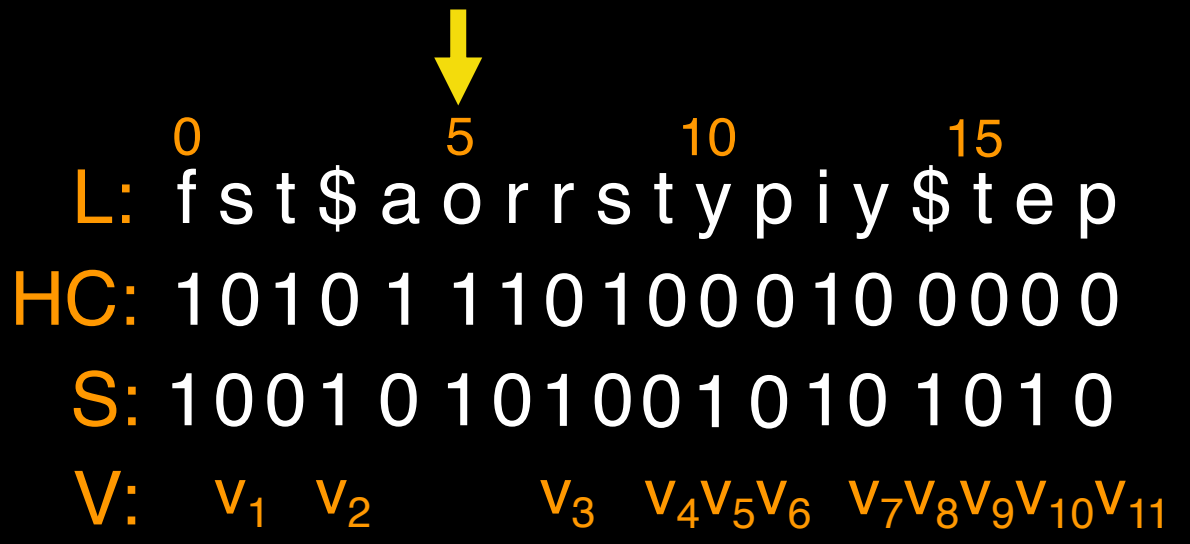
child(i) = select(S, rank(HC, i)+1)

5

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



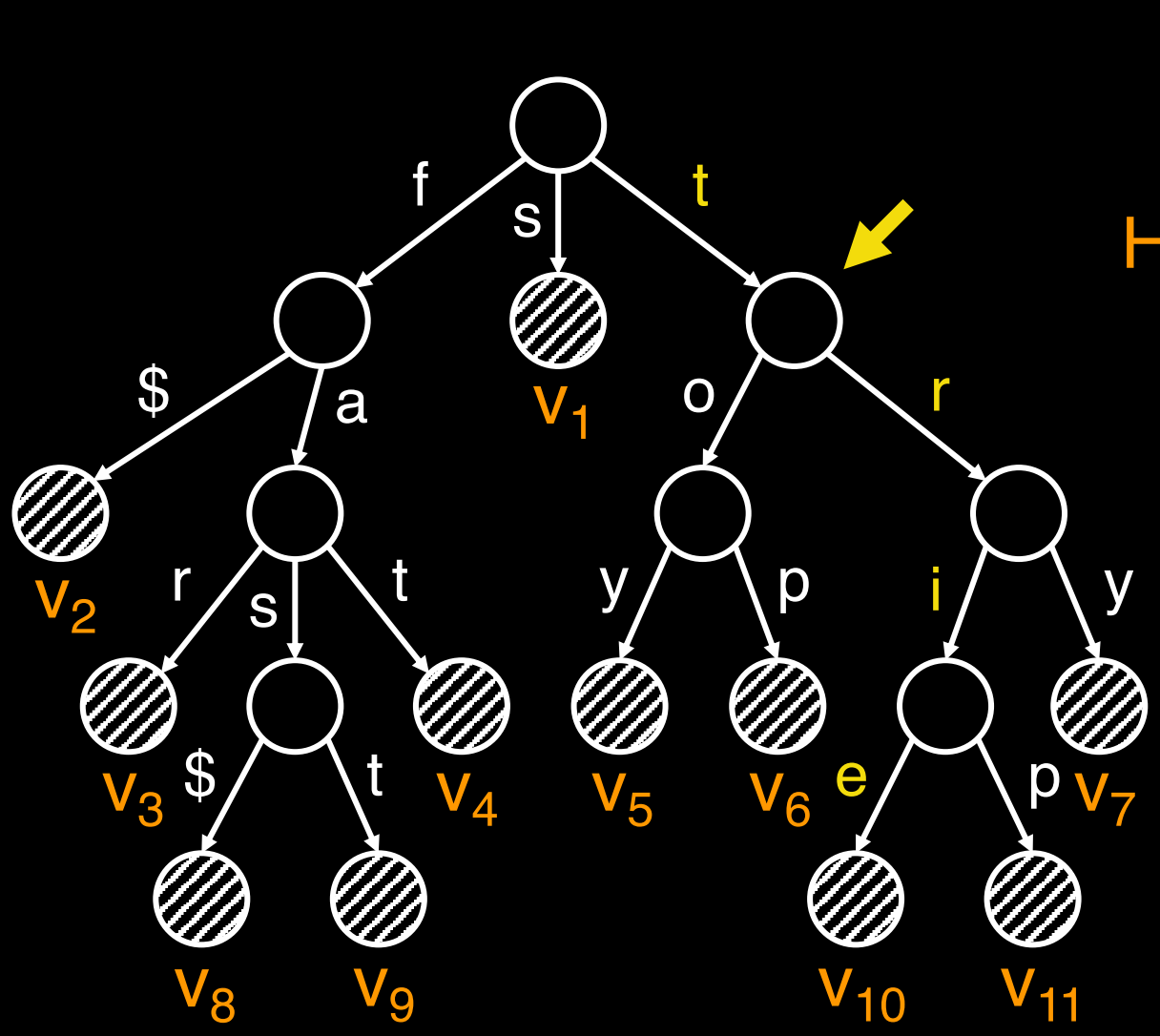


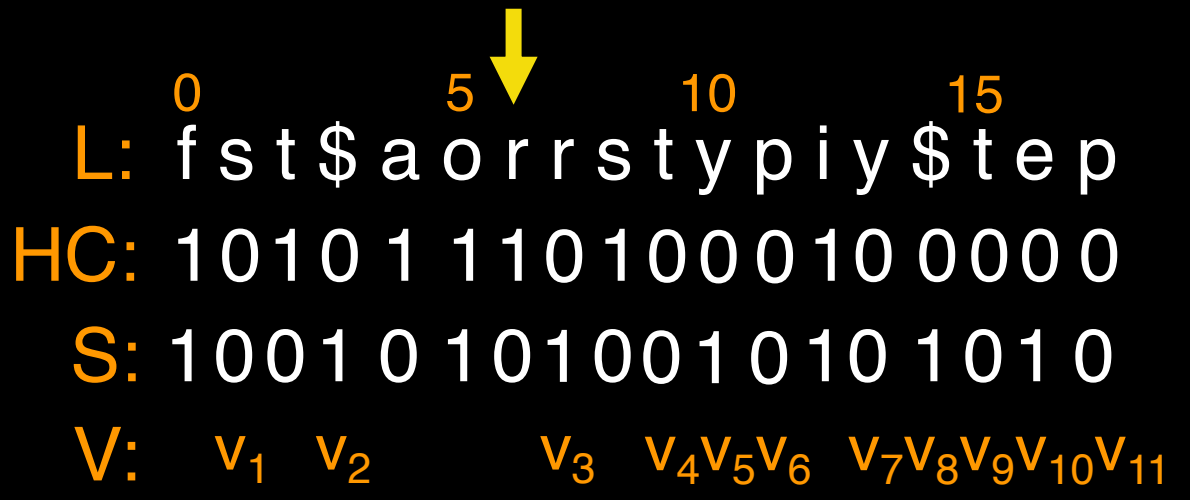
child(i) = select(S, rank(HC, i)+1)

5

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select

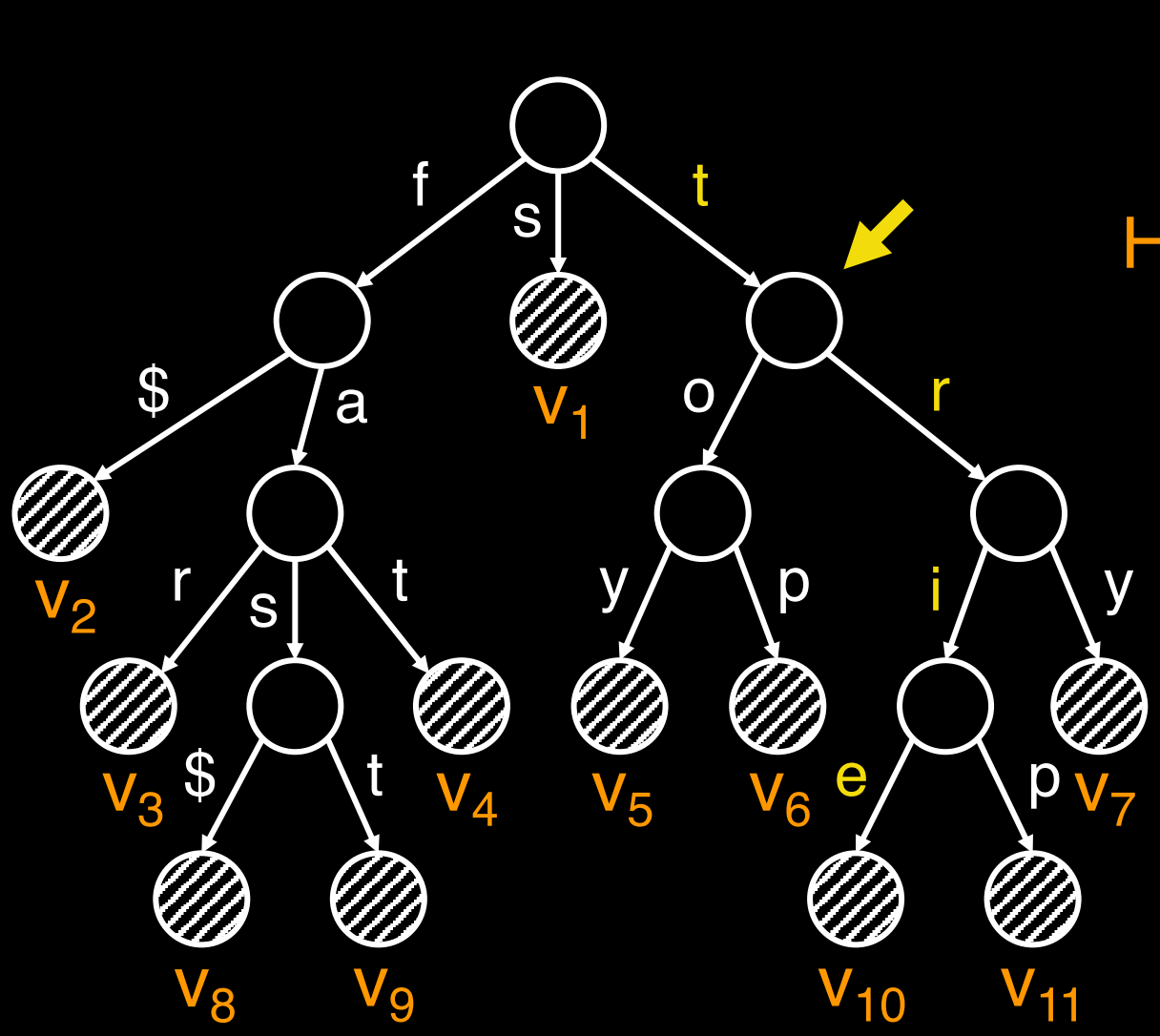


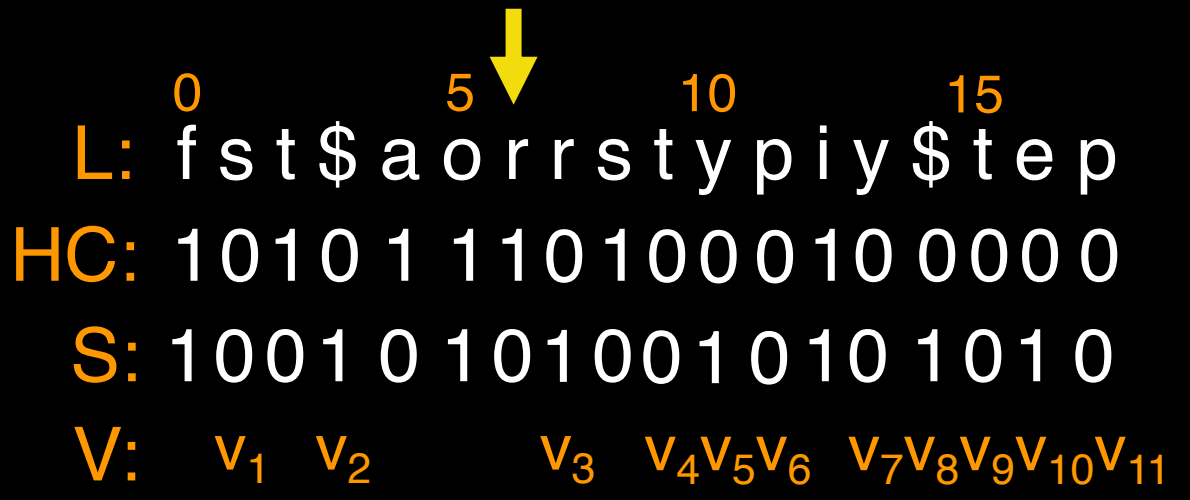


child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



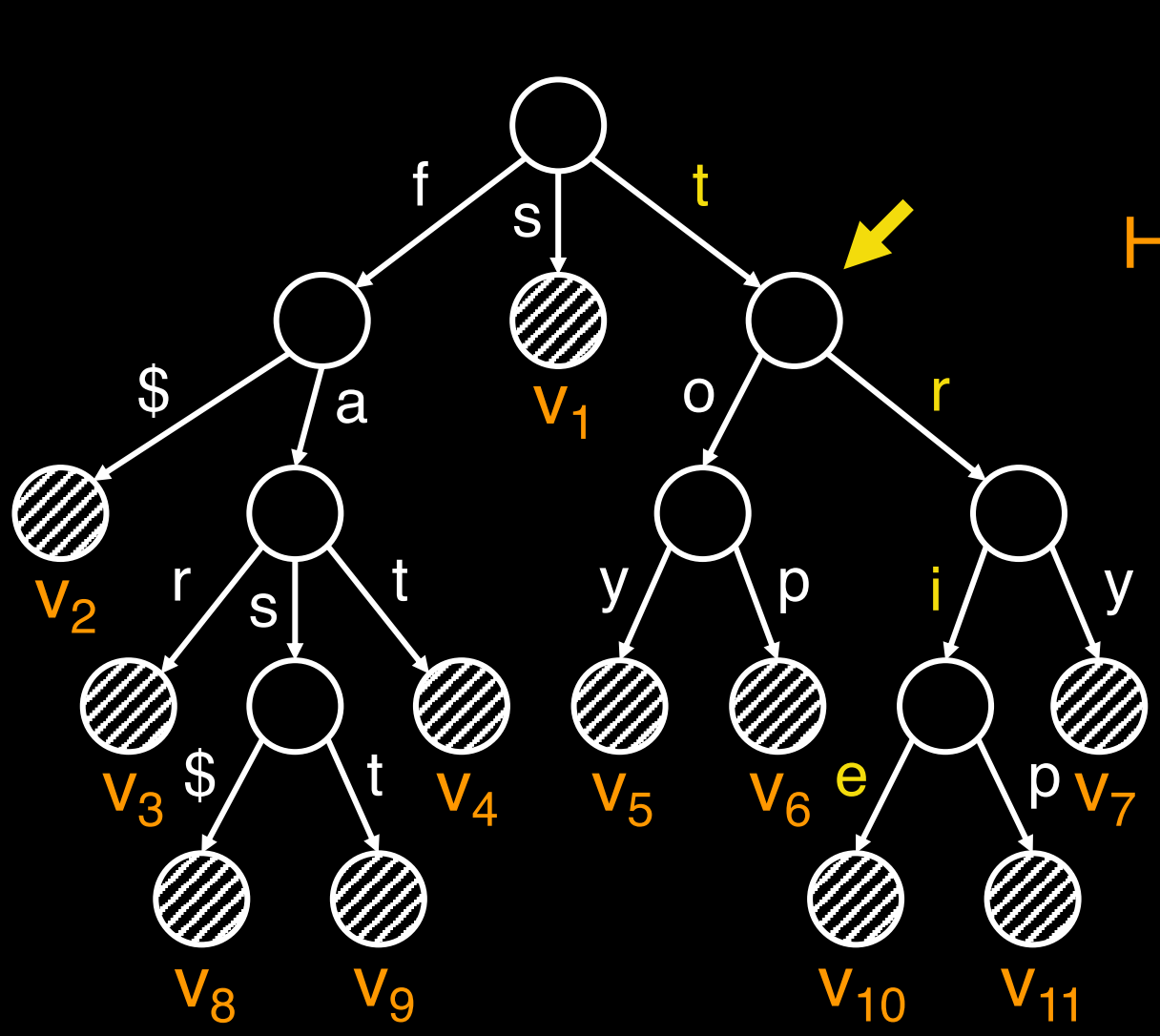


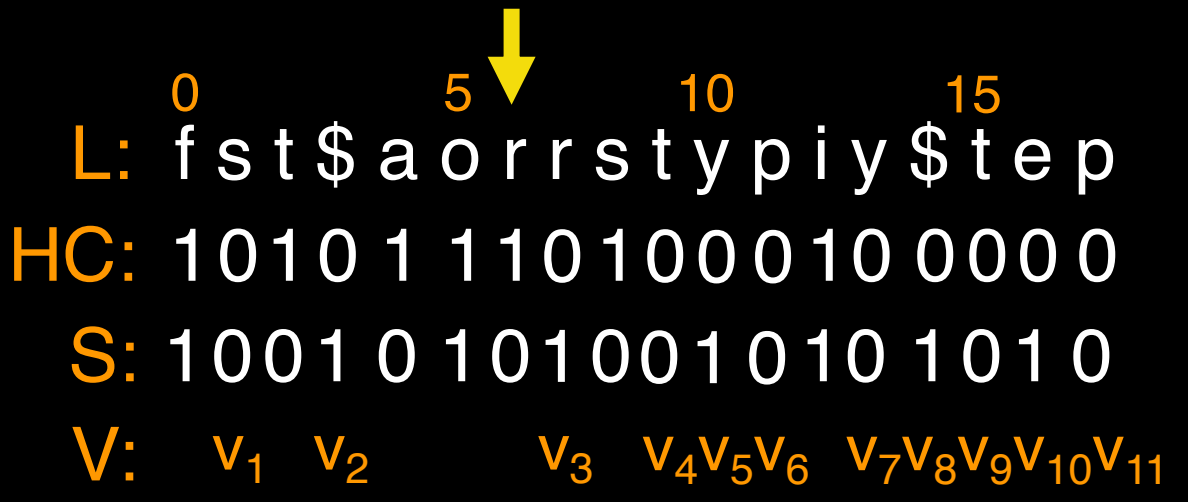
child(i) = select(S, rank(HC, i)+1)

6 6

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



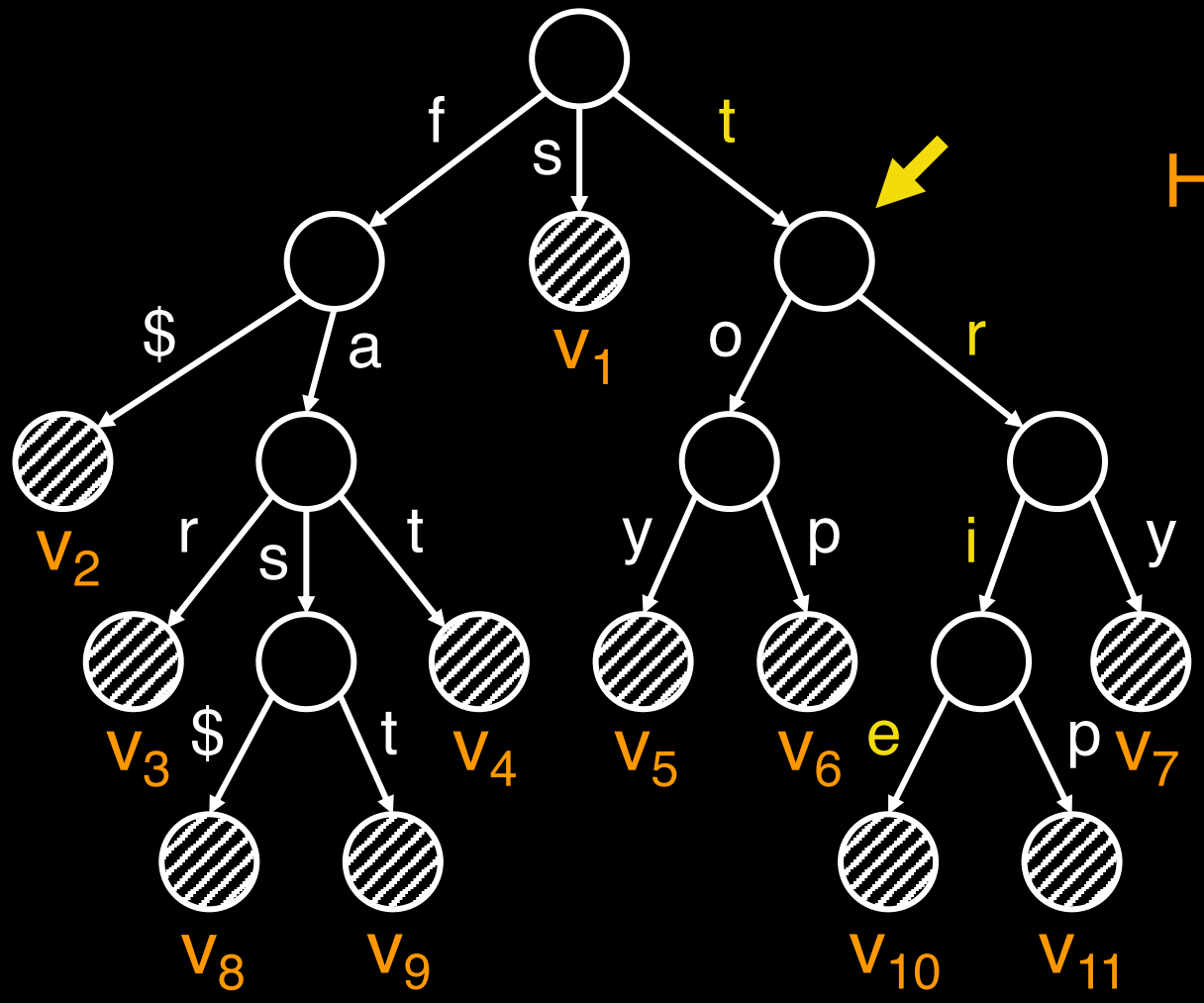


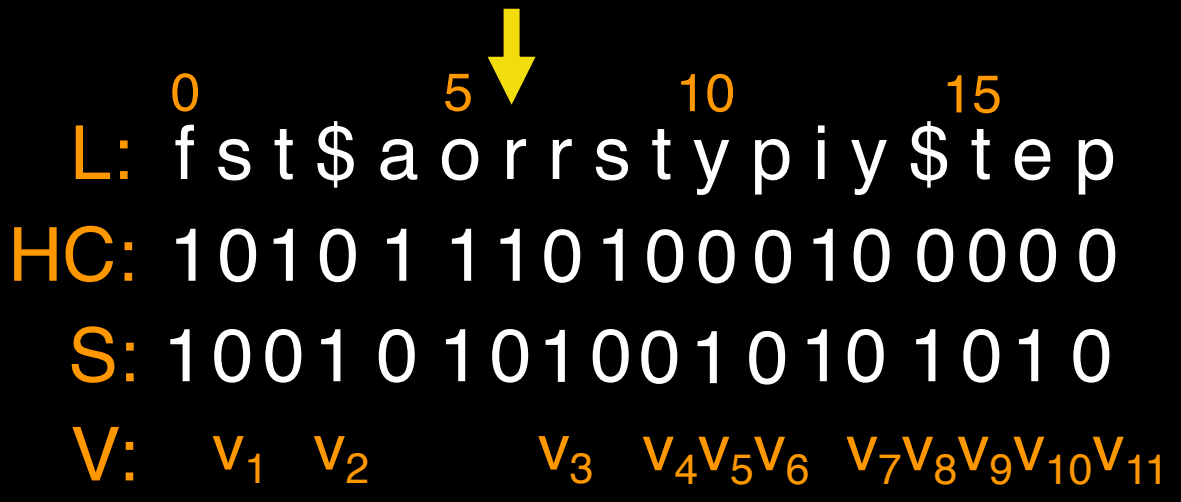
child(i) = select(S, rank(HC, i)+1)

6

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



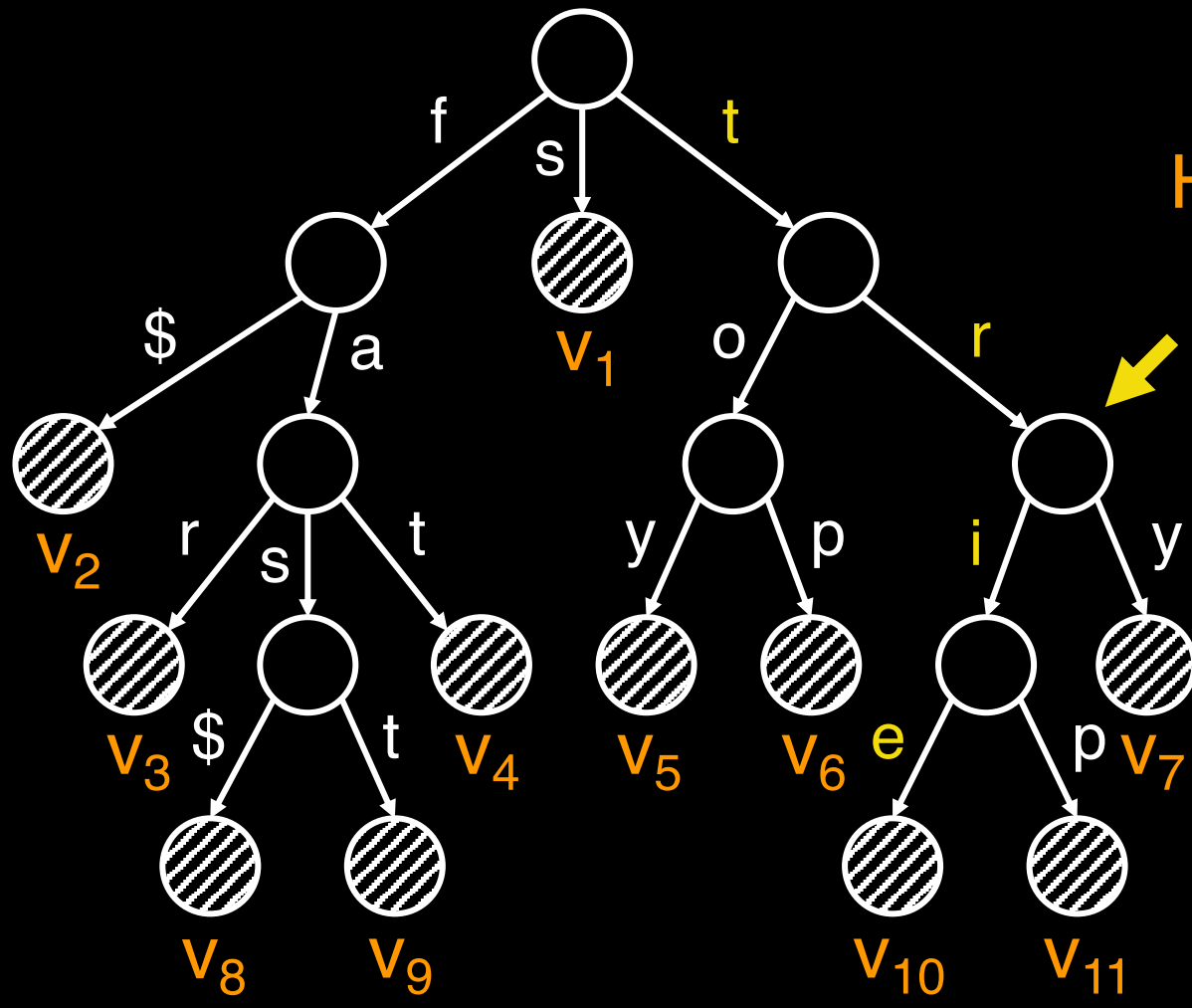


child(i) = select(S, rank(HC, i)+1)

12

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



0 5 10 15

L: f s t $ a o r r s t y p i y $ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

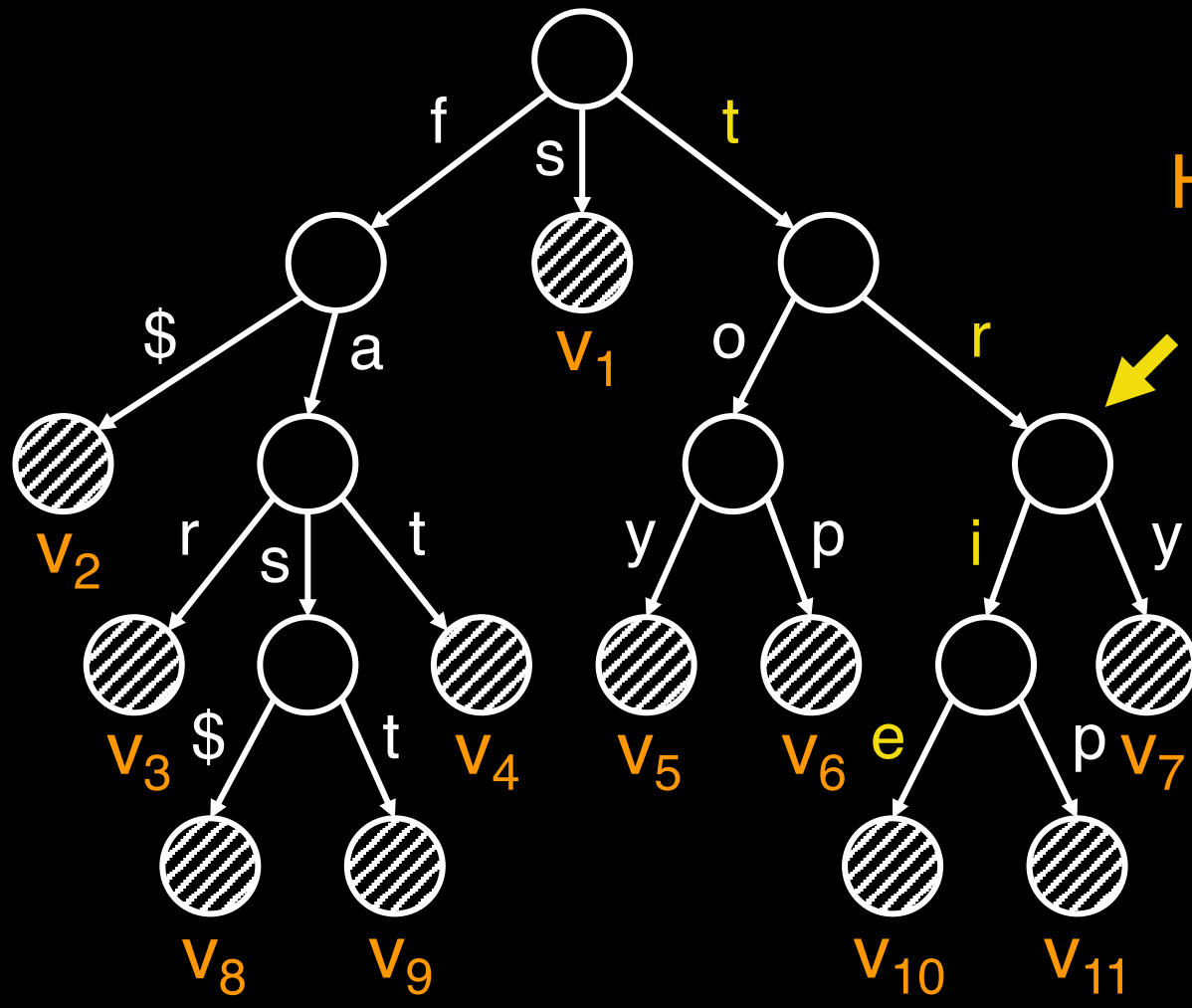
V: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}

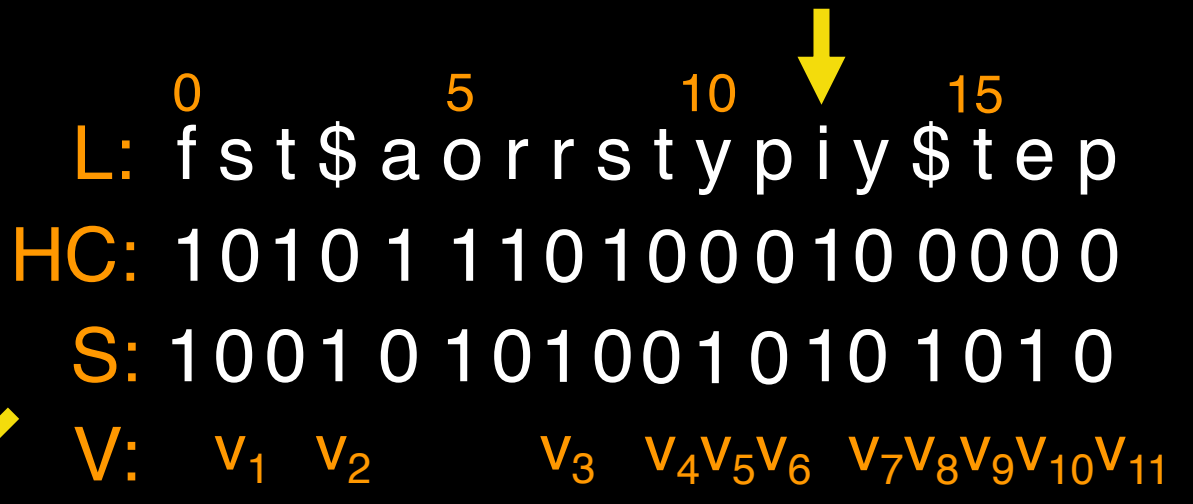
child(i) = select(S, rank(HC, i)+1)

12

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



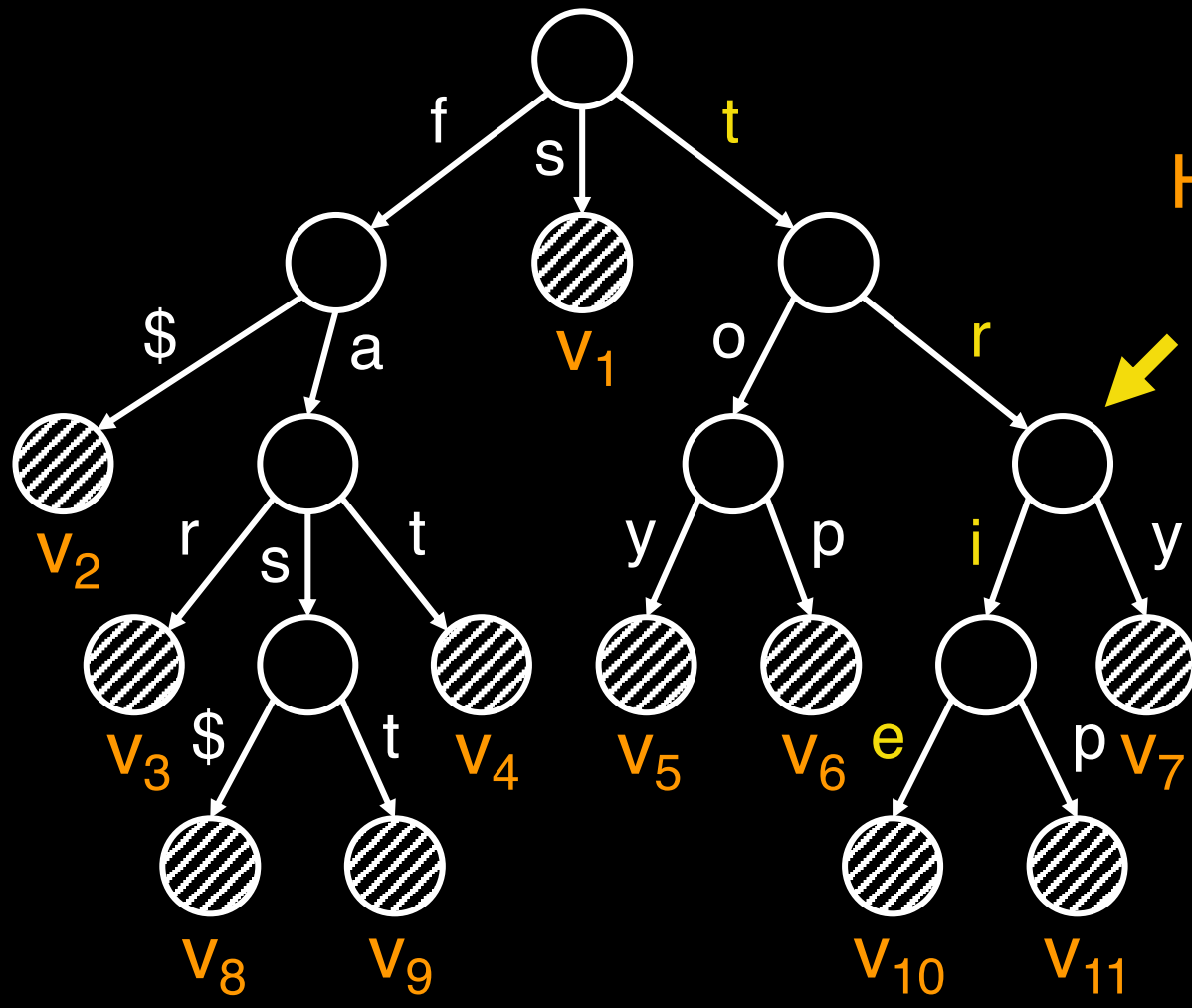


child(i) = select(S, rank(HC, i)+1)

12 12

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



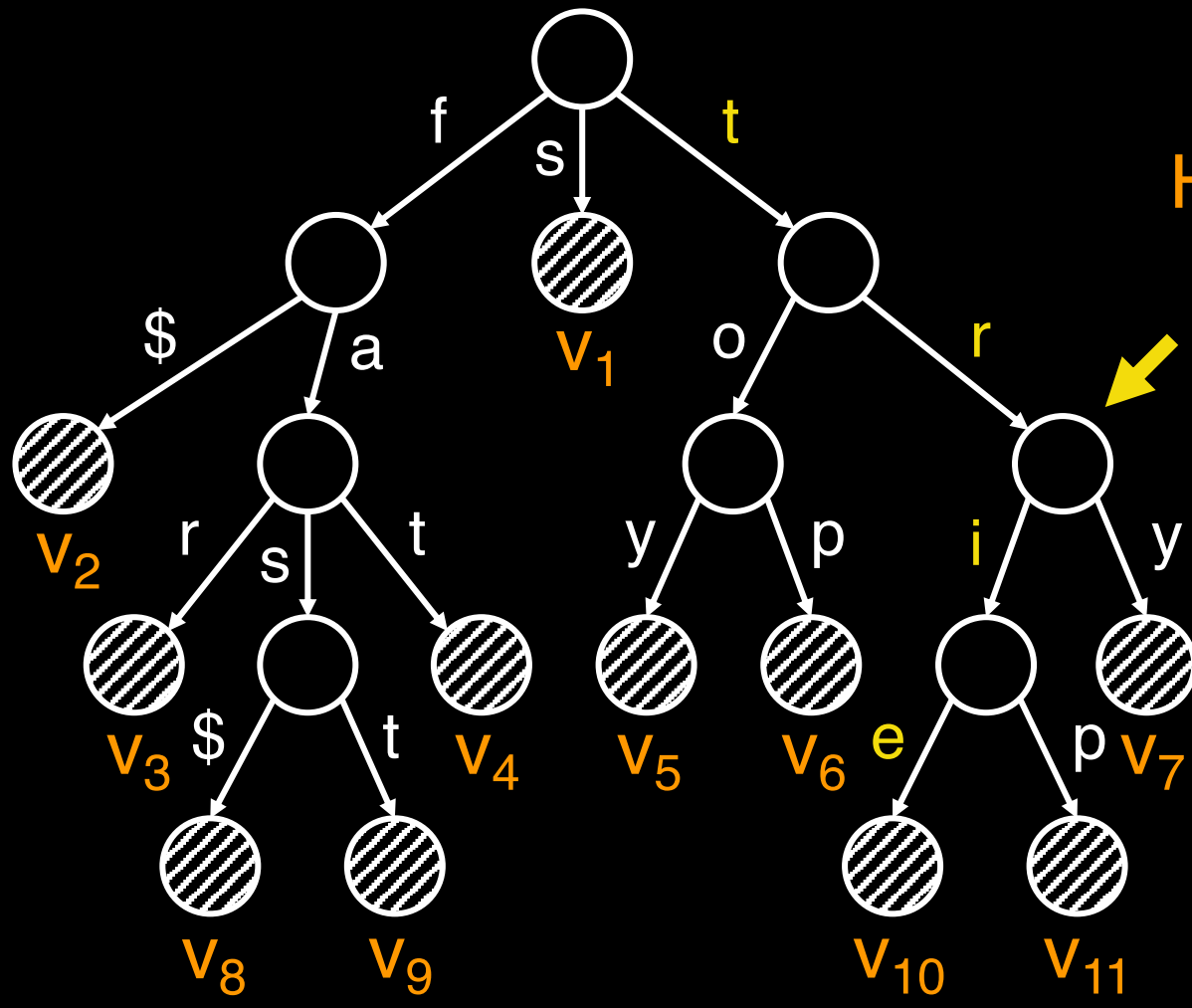

```
          0         5         10        15
L:  f s t $ a o r r s t y p i y $ t e p
HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0
S:  1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0
V:  v1  v2      v3  v4 v5 v6  v7 v8 v9 v10 v11
```

child(i) = select(S, rank(HC, i)+1)

8

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



0 5 10 15

L: f s t $ a o r r s t y p i y $ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

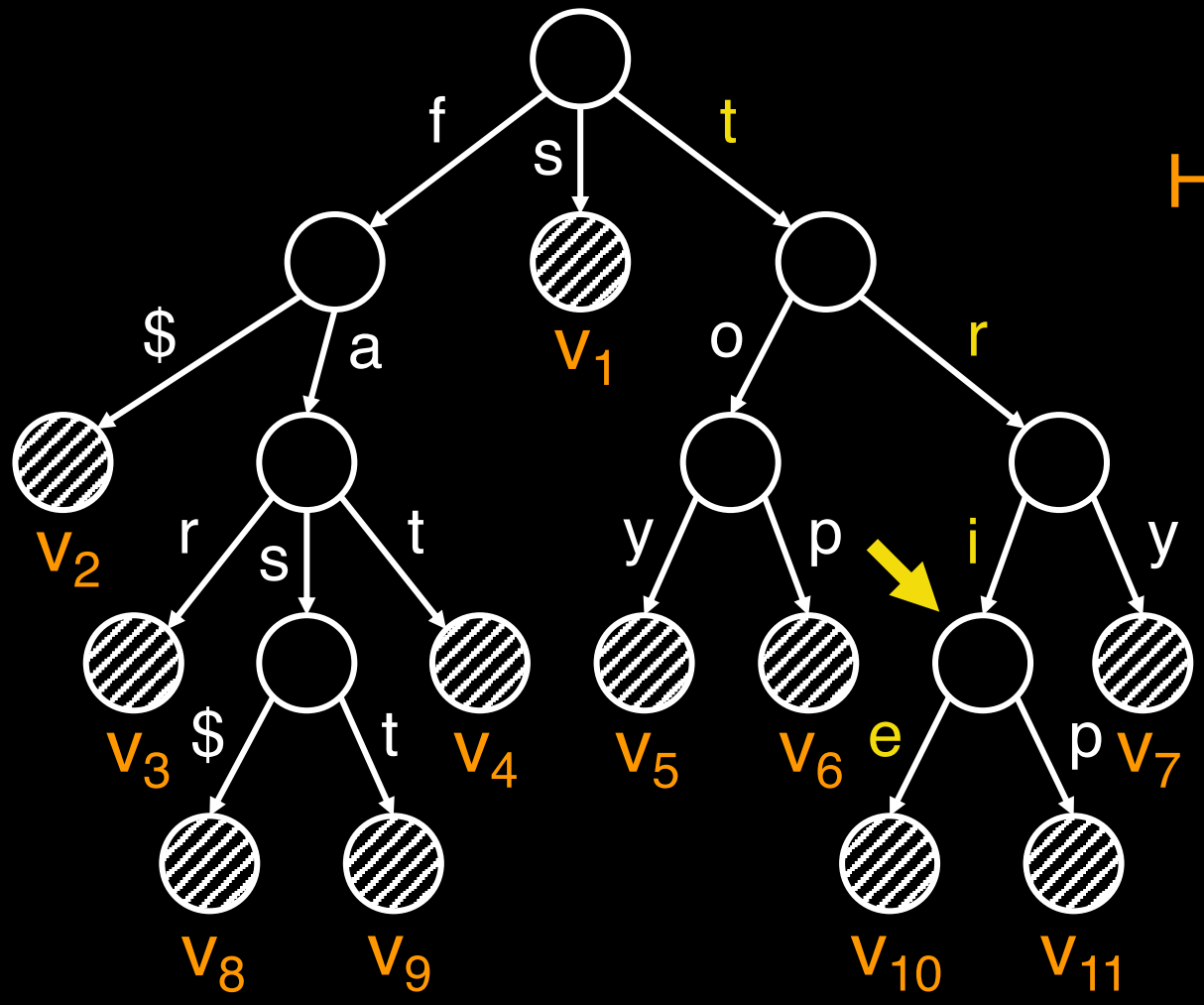
V: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}

child(i) = select(S, rank(HC, i)+1)

16

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



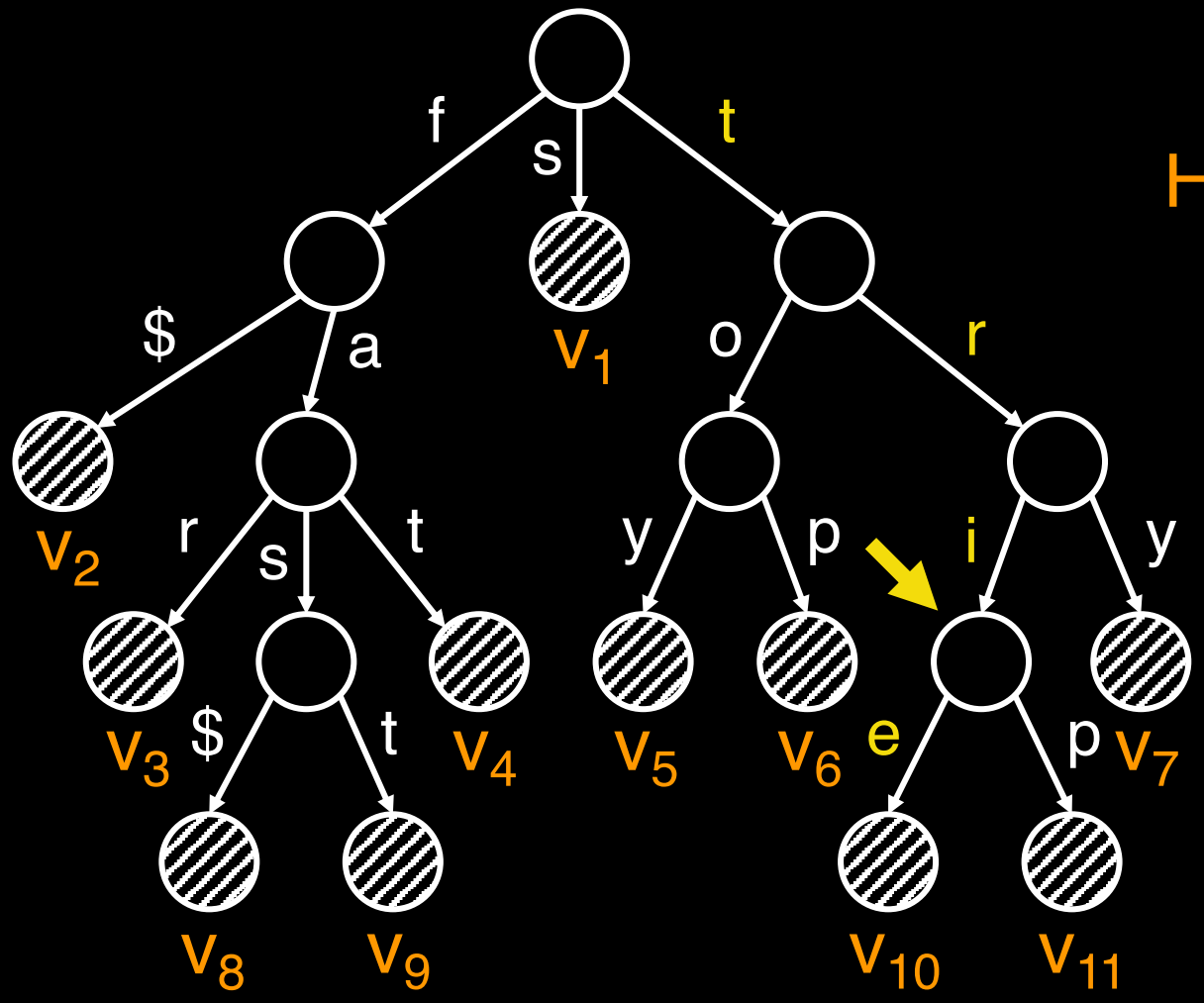
	0				5				10				15
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0	0	0	
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0
V:		v_1		v_2				v_3		v_4	v_5	v_6		v_7	v_8	v_9	v_{10}	v_{11}

child(i) = select(S, rank(HC, i)+1)

16

value(i) = i - rank(HC, i)

Tree navigation relies on rank & select



0 5 10 15

L: f s t $ a o r r s t y p i y $ t e p

HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0

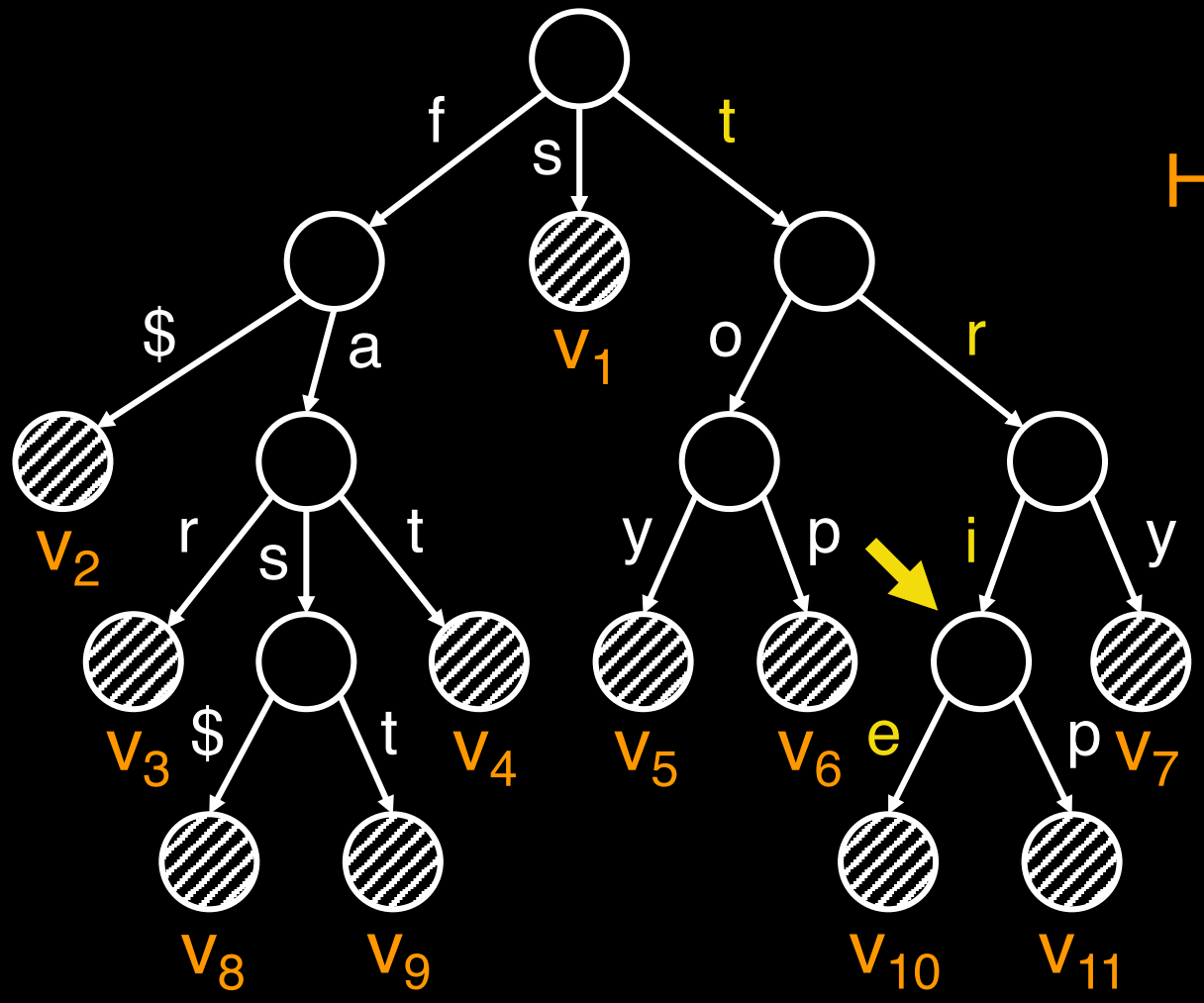
V: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}

child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

16 16 16

Tree navigation relies on rank & select



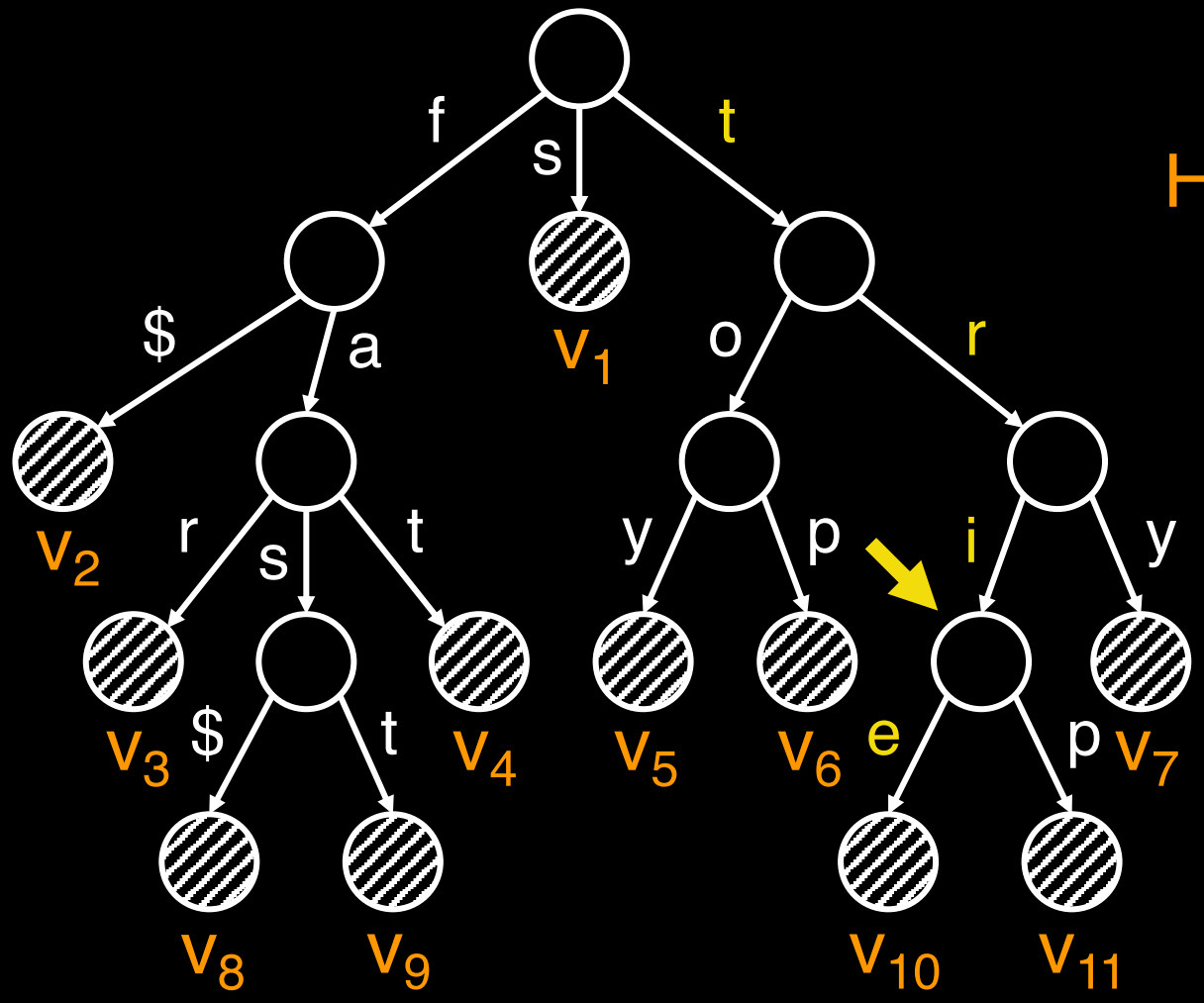
	0				5				10				15
L:	f	s	t	$	a	o	r	r	s	t	y	p	i	y	$	t	e	p
HC:	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0	0	0	
S:	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0		
V:		v_1		v_2				v_3		v_4	v_5	v_6		v_7	v_8	v_9	v_{10}	v_{11}

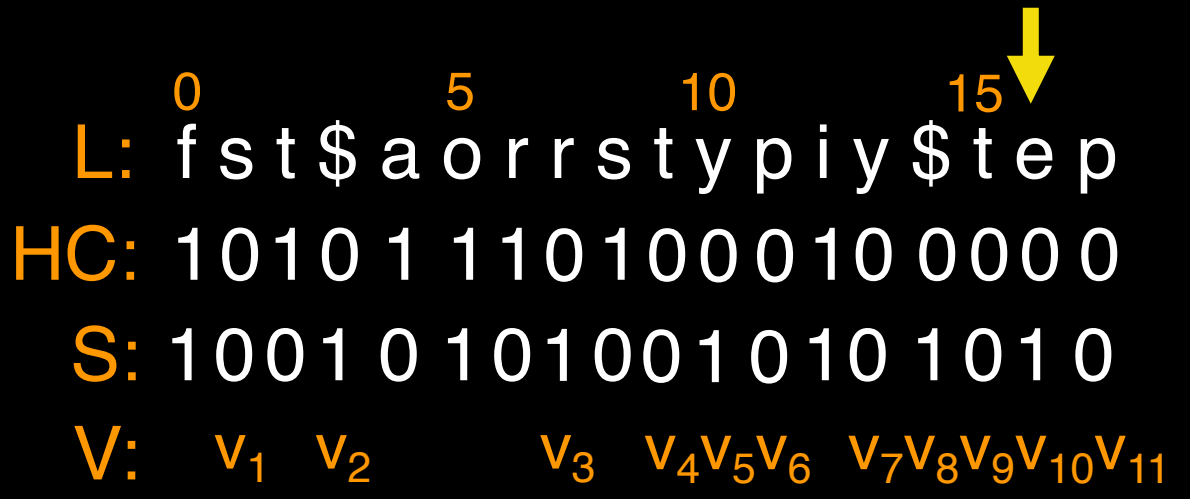
child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

16 7

Tree navigation relies on rank & select



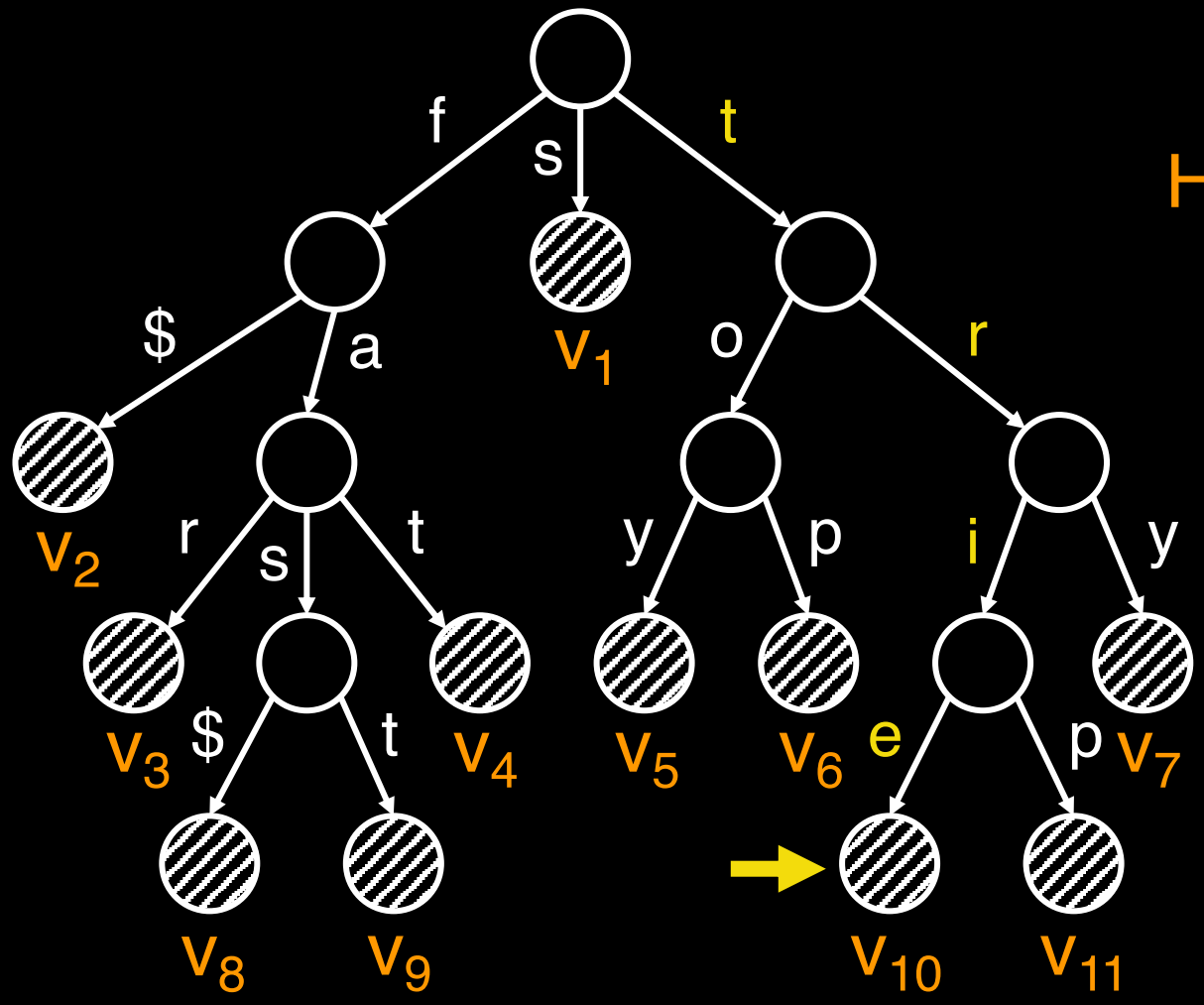


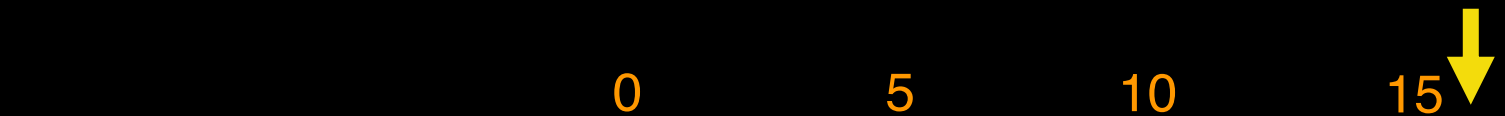
child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

9

Tree navigation relies on rank & select





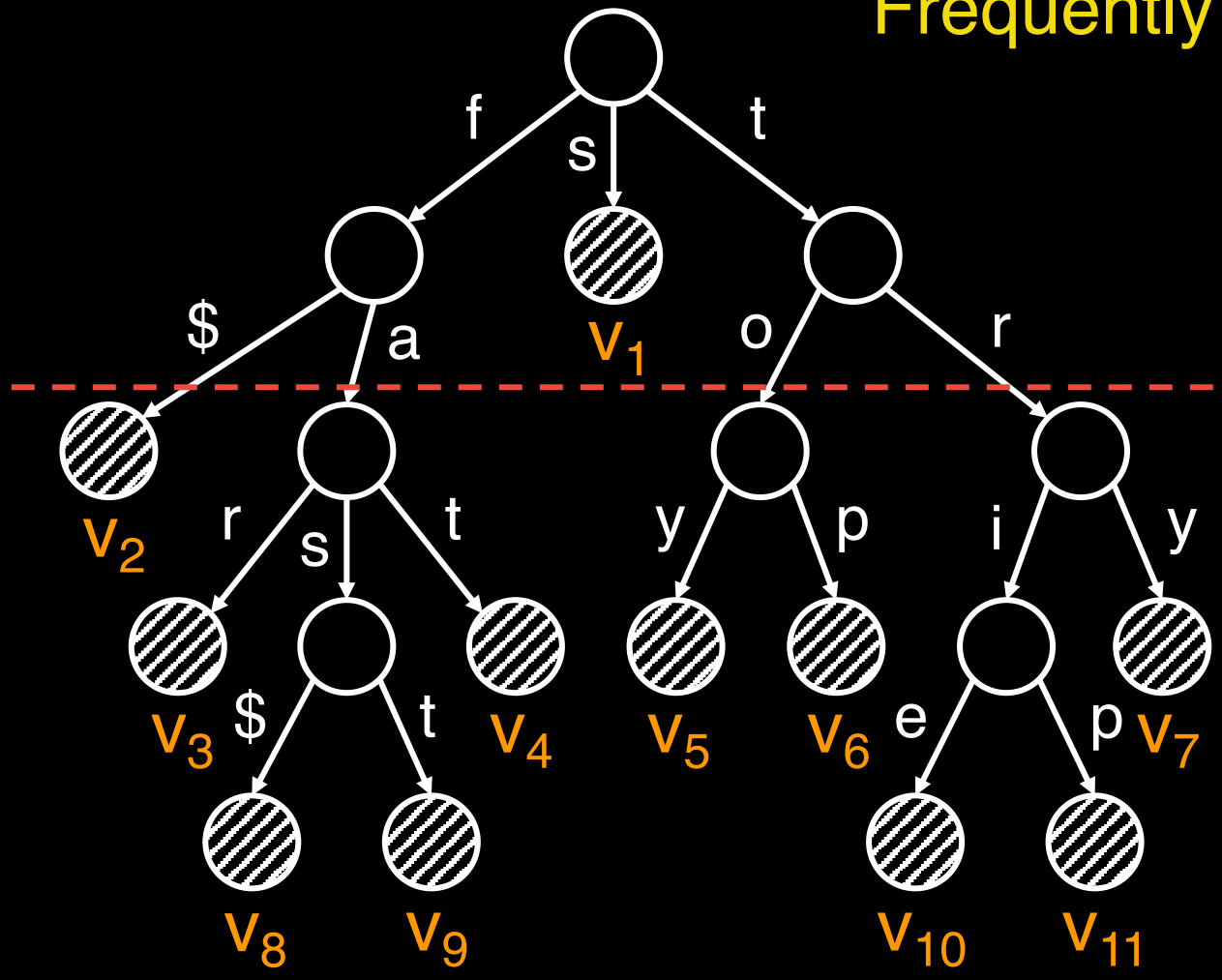
child(i) = select(S, rank(HC, i)+1)

value(i) = i - rank(HC, i)

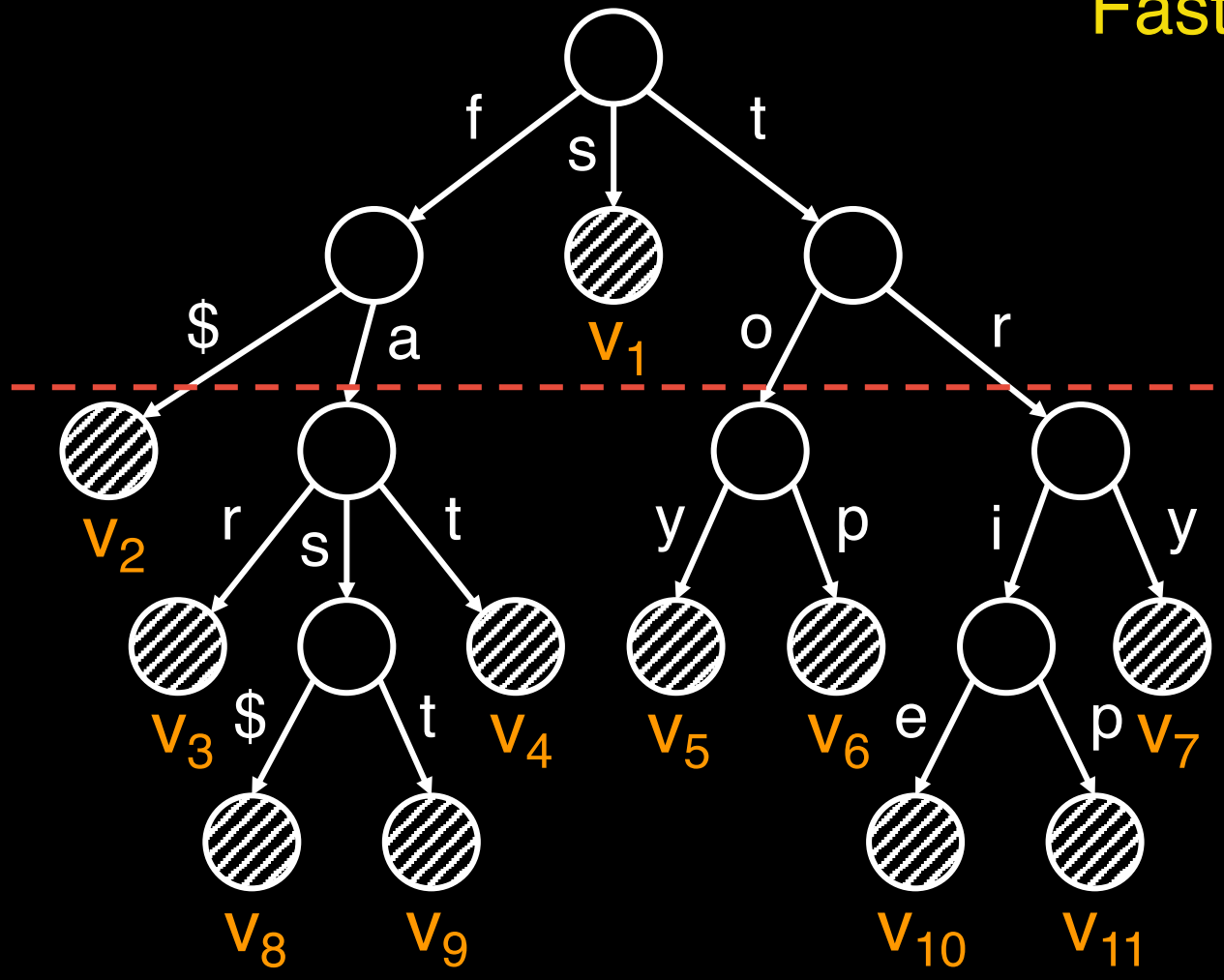
9

Performance Optimization

LOUDS-Dense: optimize for the common case



LOUDS-Dense: optimize for the common case

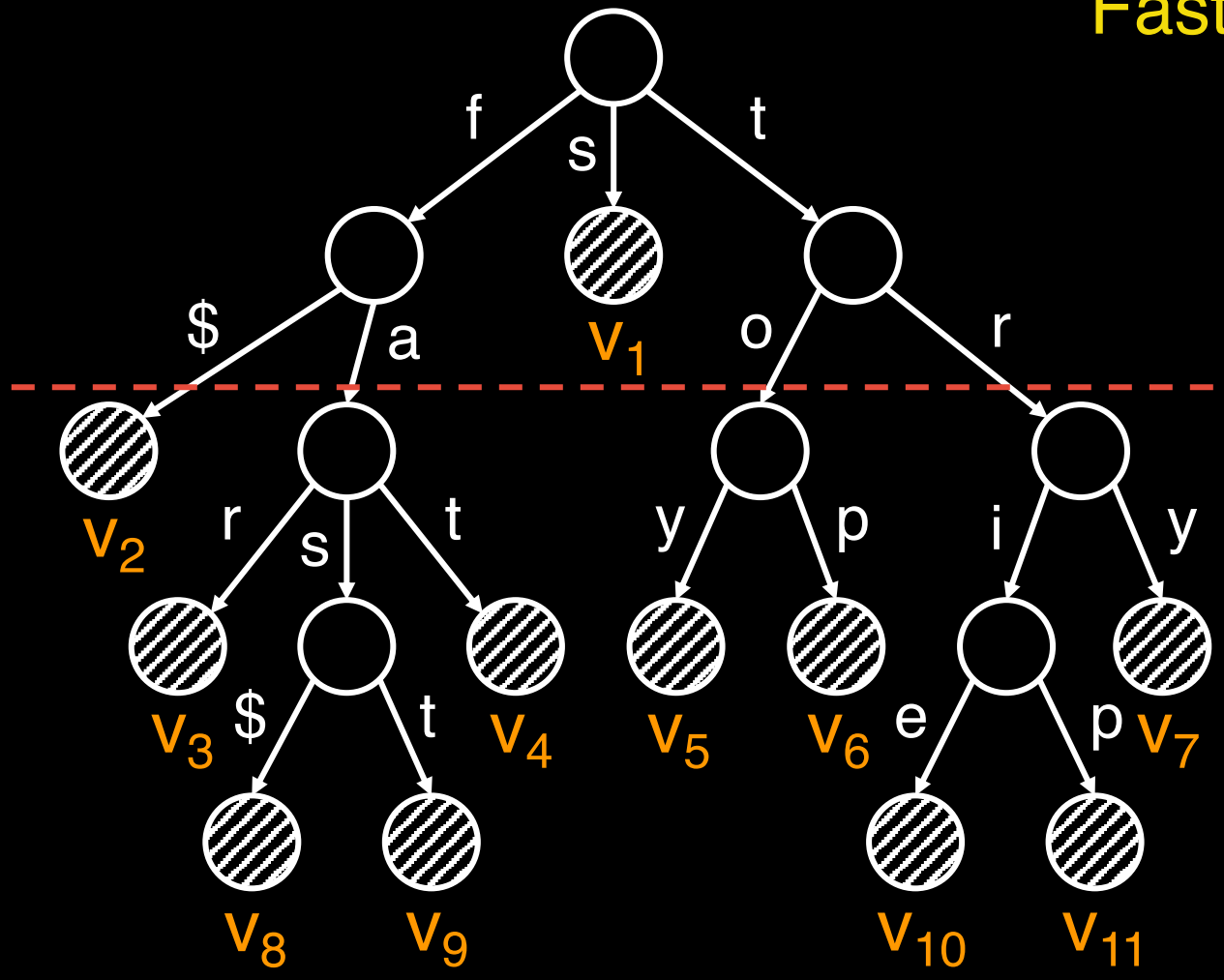


Fast

Divided by size ratio

Space-efficient

LOUDS-Dense: optimize for the common case



Fast

L: f s t $ a o r

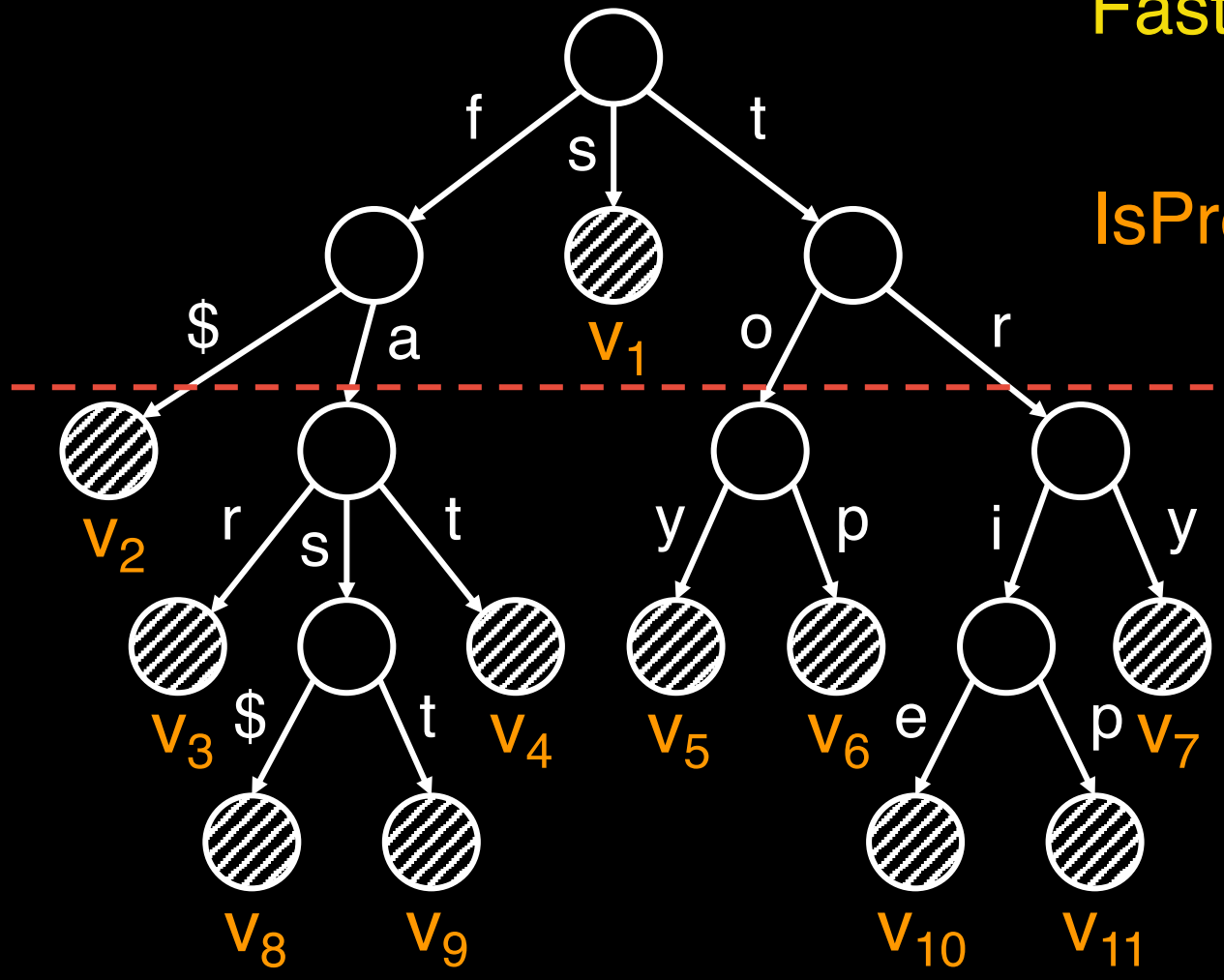
HC: 1 0 1 0 1 1 1

S: 1 0 0 1 0 1 0

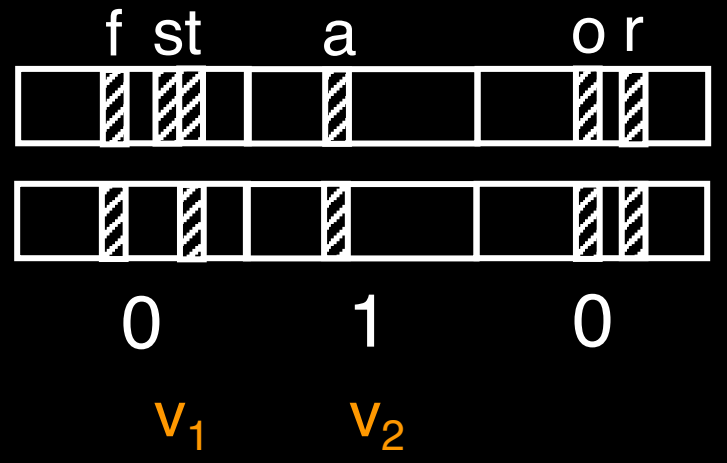
V: v_1 v_2

Space-efficient

LOUDS-Dense: optimize for the common case

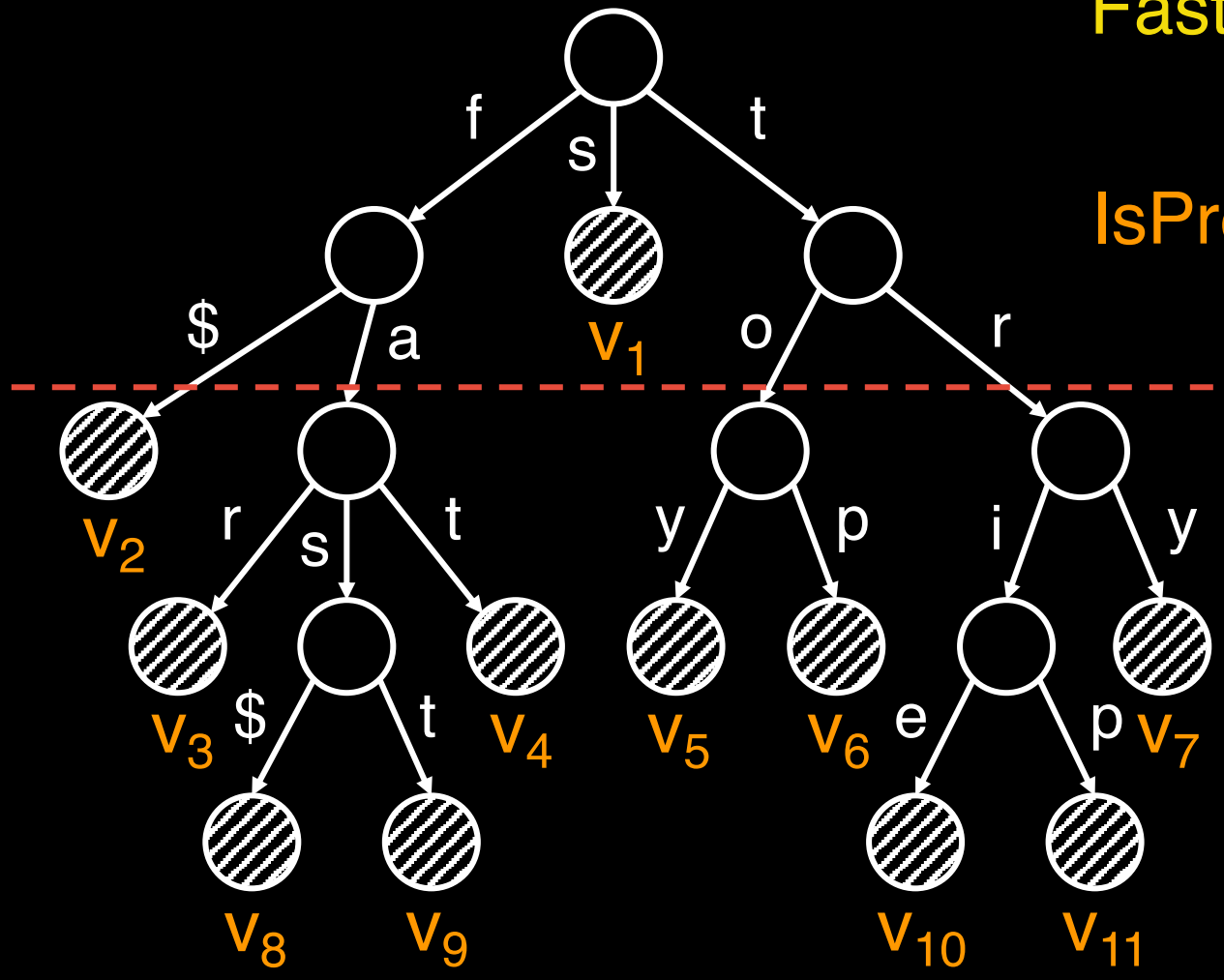


Fast



Space-efficient

LOUDS-DS: best of both worlds



Fast

L: f st a o r

HC:

IsPrefixKey: 0 1 0

V: v_1 v_2

L: r s t y p i y $ t e p

HC: 0 1 0 0 0 1 0 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 1 0

V: v_3 $v_4 v_5 v_6$ $v_7 v_8 v_9 v_{10} v_{11}$

Space-efficient

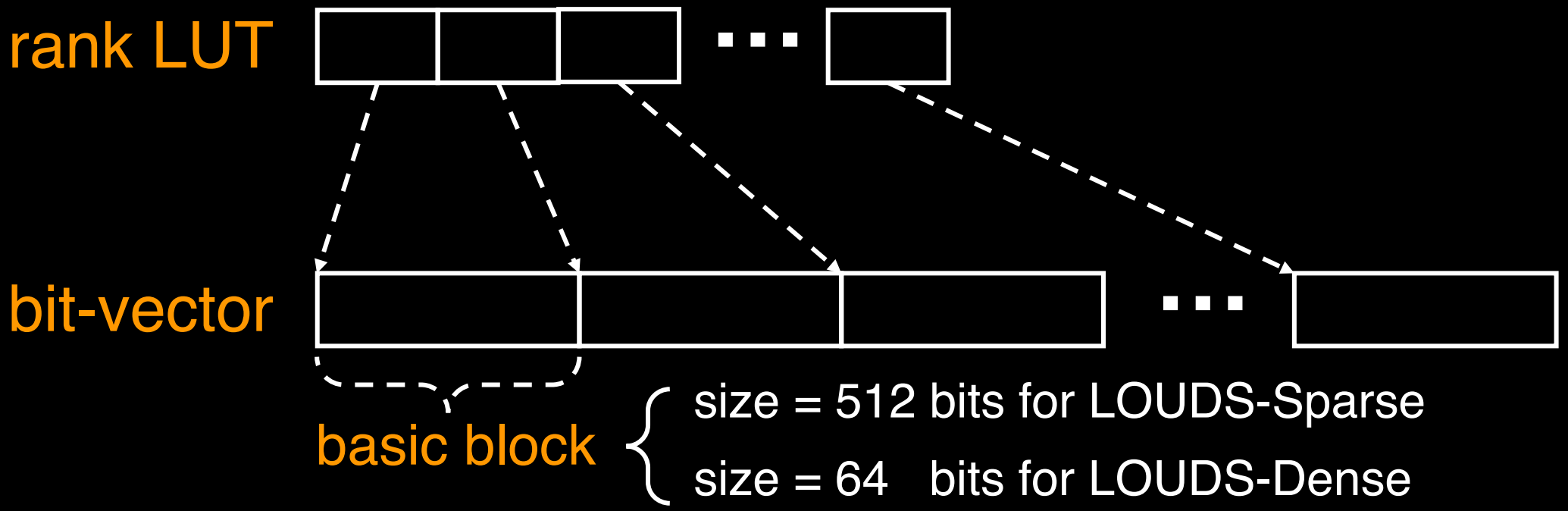
Observations: rank & select on LOUDS-DS

① Either rank or select is required, but not both

② Space taken by auxiliary structures is small

③ The bit vector (S) that requires select is dense

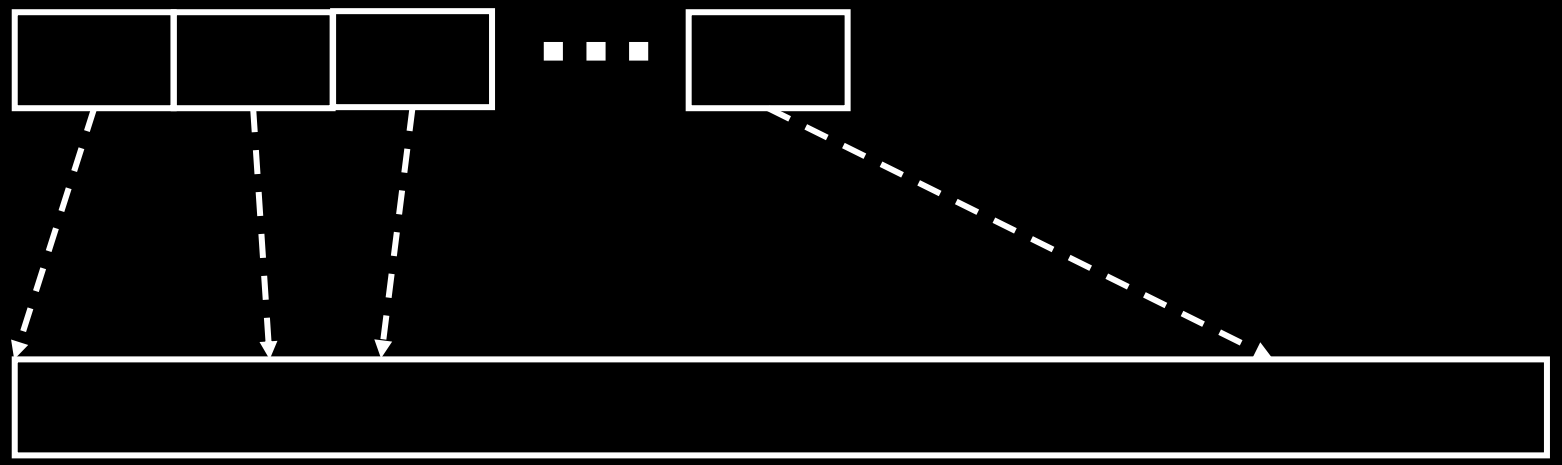
Optimizing rank structure for LOUDS-DS



within basic block: use popcount instruction

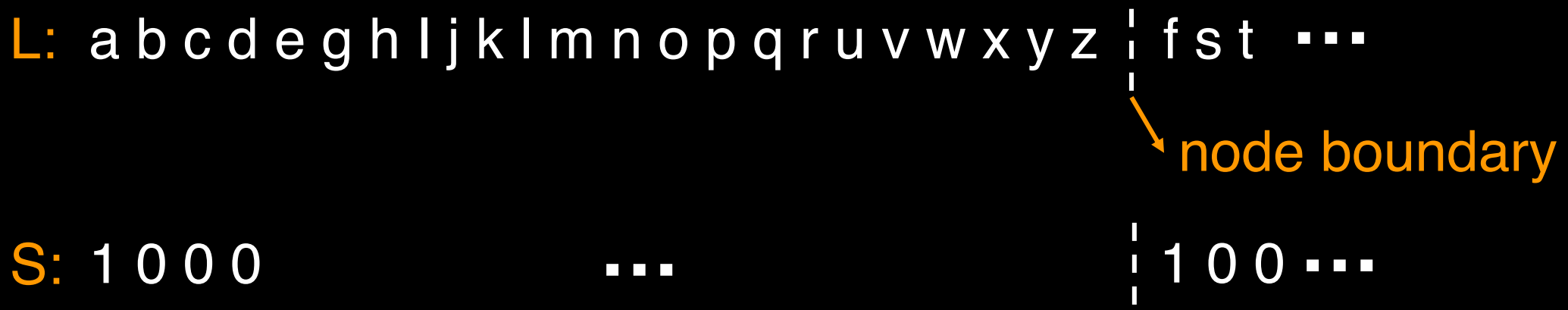
Optimizing rank structure for LOUDS-DS

select samples
(every x 1's)



bit-vector

Optimizing label search algorithm using SIMD



Optimizing label search algorithm using SIMD

Optimizing label search algorithm using SIMD

Using prefetching to hide memory latency

L: f s t $ a o r r s t y p i y $ t e p

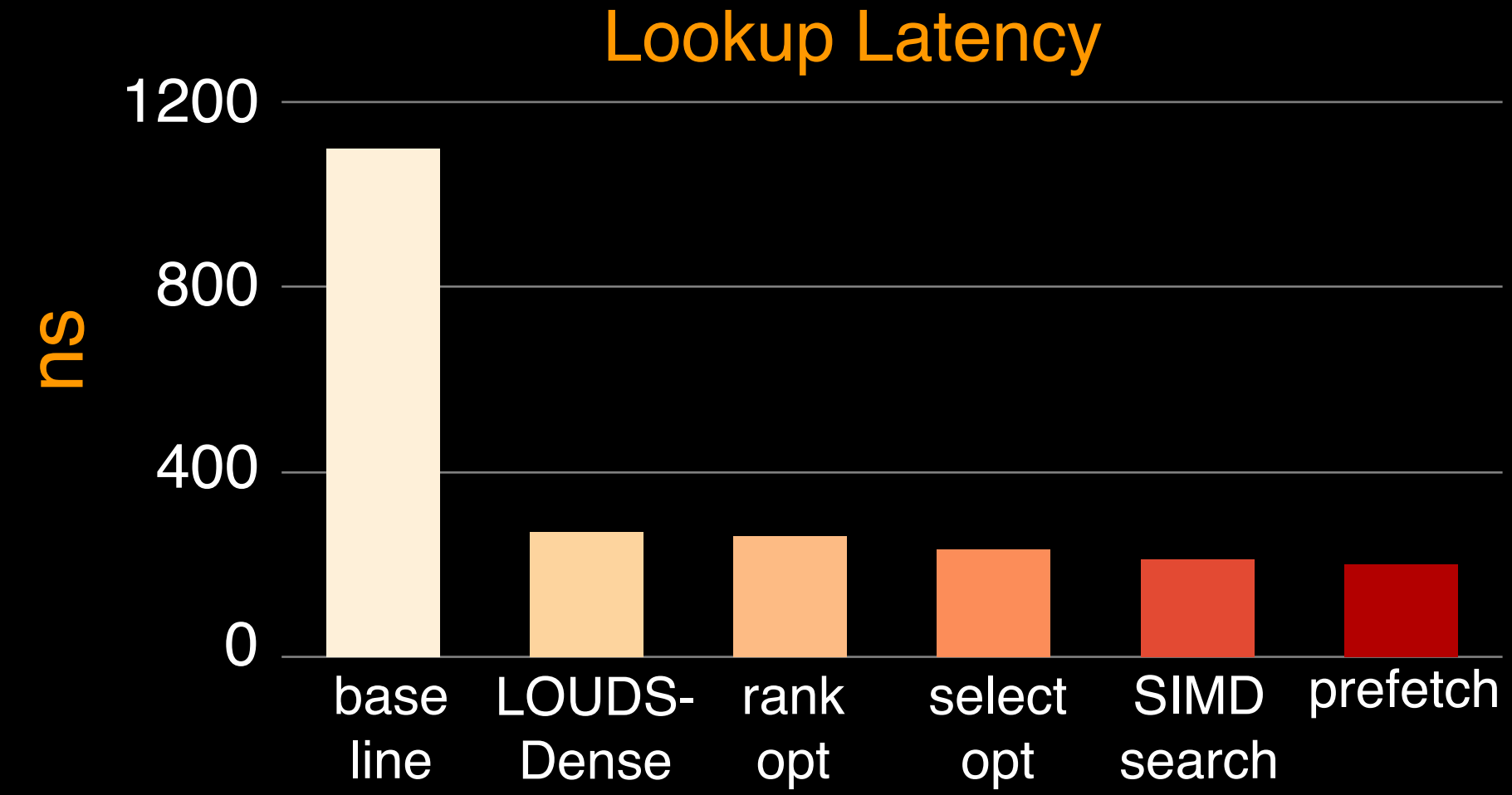
HC: 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0

S: 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0

V: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11}

What makes FST fast

50M 64-bit integer keys



Where is FST in the performance-space trade-off

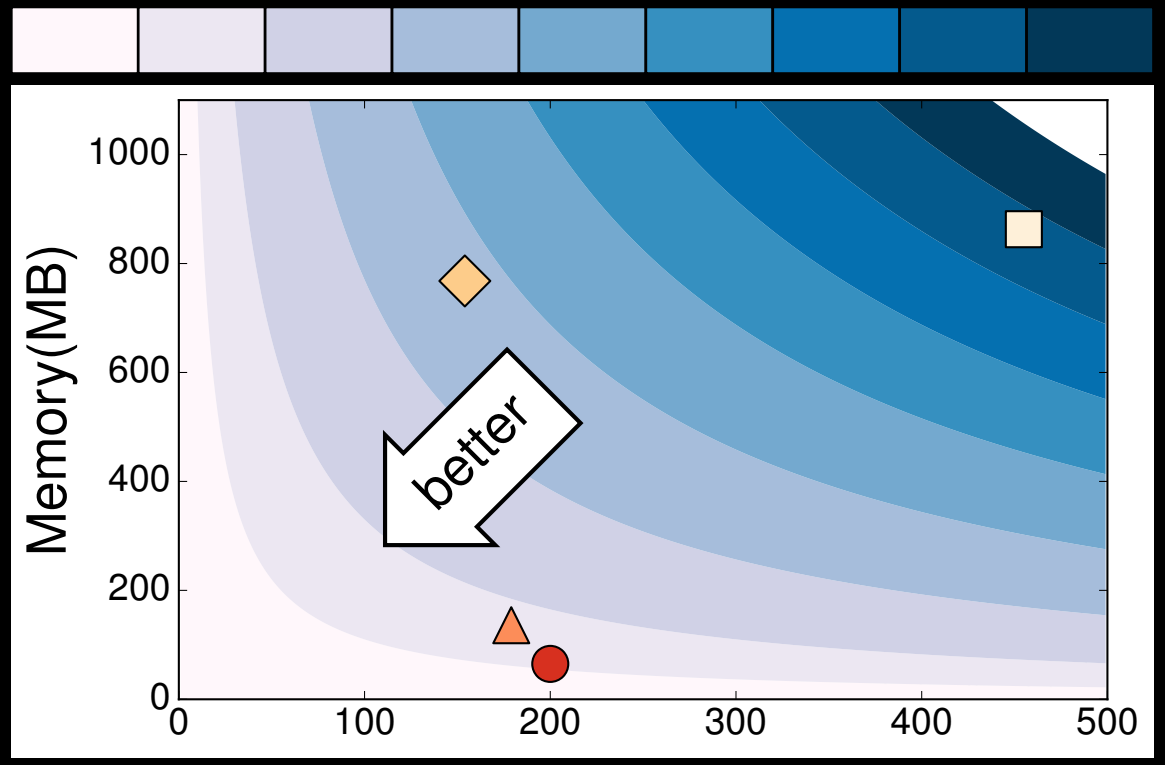
- 50M 64-bit integer keys

Cost function: $C = PrS$

r > 1 → favors performance
r < 1 → favors space

low cost → high cost

$r = 1$ →



B+tree
ART
C-ART
FST

Conclusion

① LOUDS-DS
- A hybrid approach to succinctly encode tries

② FST
- Matches performance of pointer-based tries
- 4 – 15x faster than existing succinct tries
- Consumes only close-to-optimal space