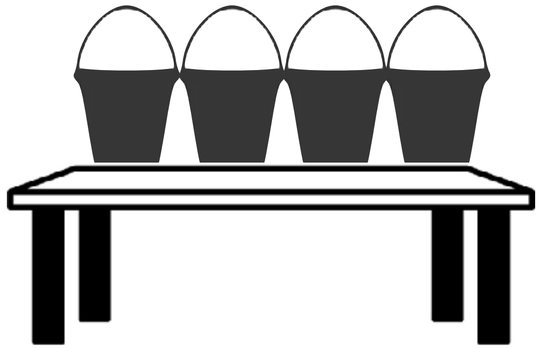
Practical Perfect Hashing

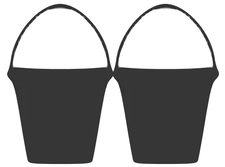
(Minimal & Dynamic)

Niv Dayan - CSC2525 Research Topics in Database Management

Hash Tables



Hash Tables



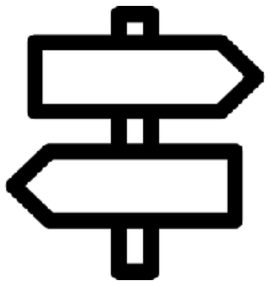
Maps keys to random buckets

Resolves collisions

Expected constant time operations

Many DB applications

Hash Join



In-memory index

Key-value Stores

Collision Resolution

Chaining

Open addressing
(Linear probing, etc)

Cuckoo hashing

Collision Resolution **relies on storing full keys**

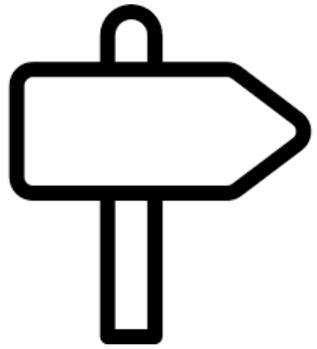
Chaining

Open addressing
(Linear probing, etc)

Cuckoo hashing

Problems storing full keys

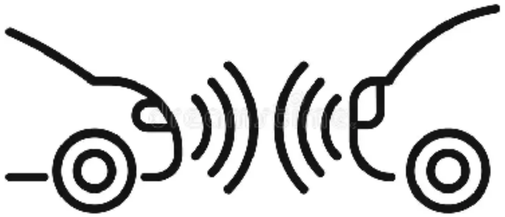
space
Keys may be large



Requires indirection
if keys are var-length

Perfect Hashing

Does not store the keys

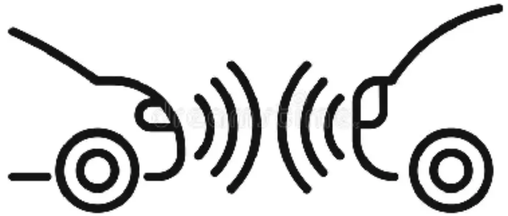


Collision-free queries from key to payload

Higher construction/ insertion overheads to resolve collisions

Perfect Hashing

Does not store the keys



Collision-free queries from key to payload

Higher construction/ insertion overheads to resolve collisions

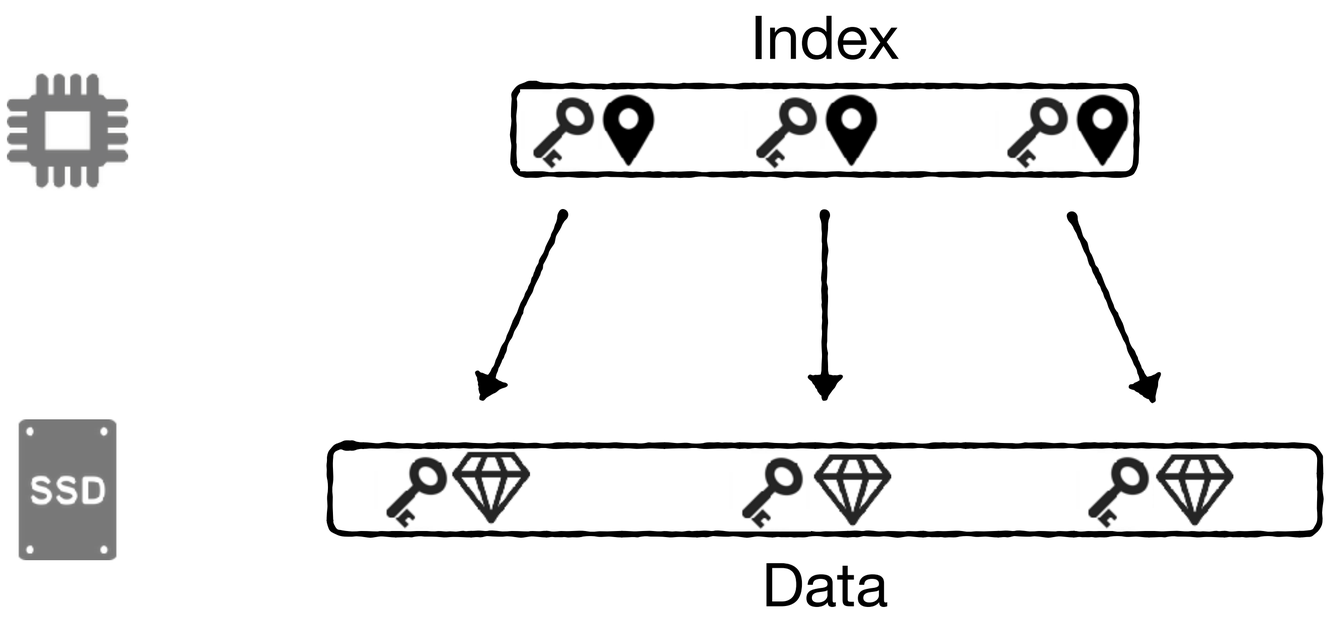
Only effective when we query existing keys! Why?

Application Example: Key-value Stores

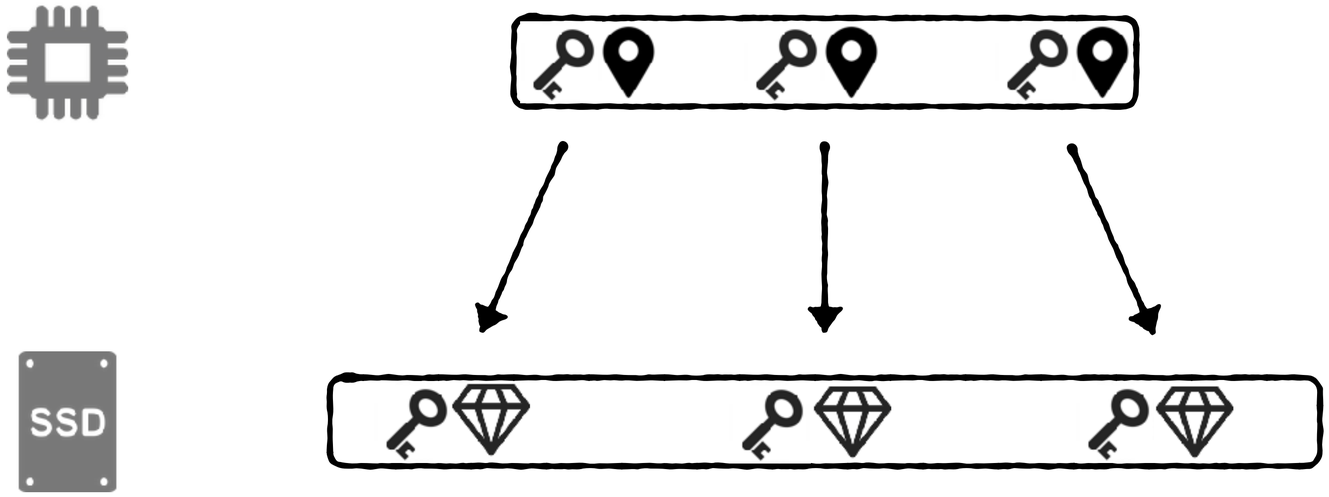
Index

Data

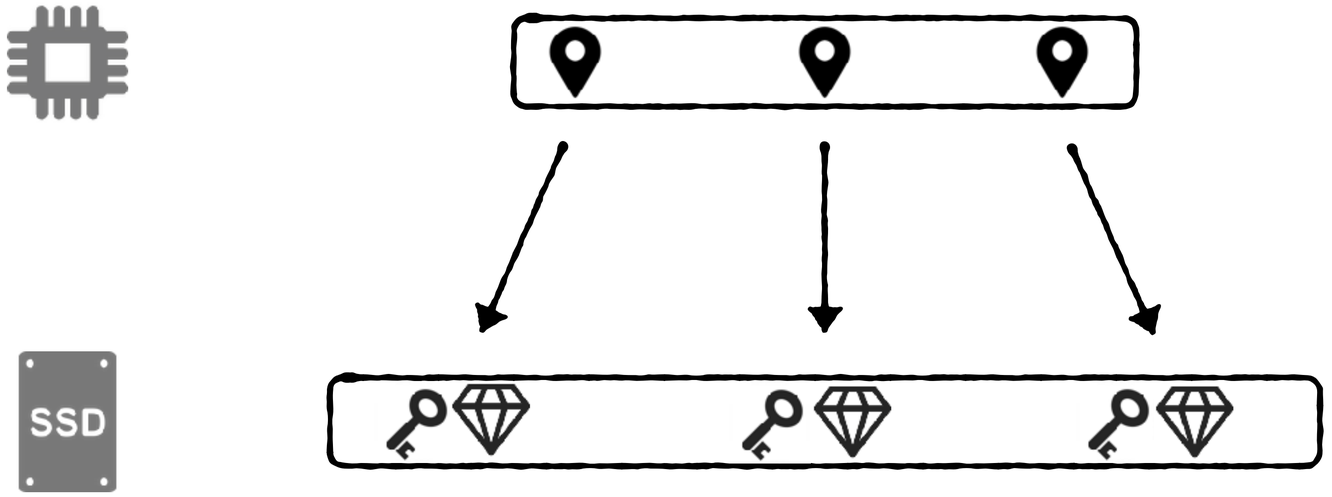
Key-value Stores



Index can be hash table containing keys (e.g., bitcask)

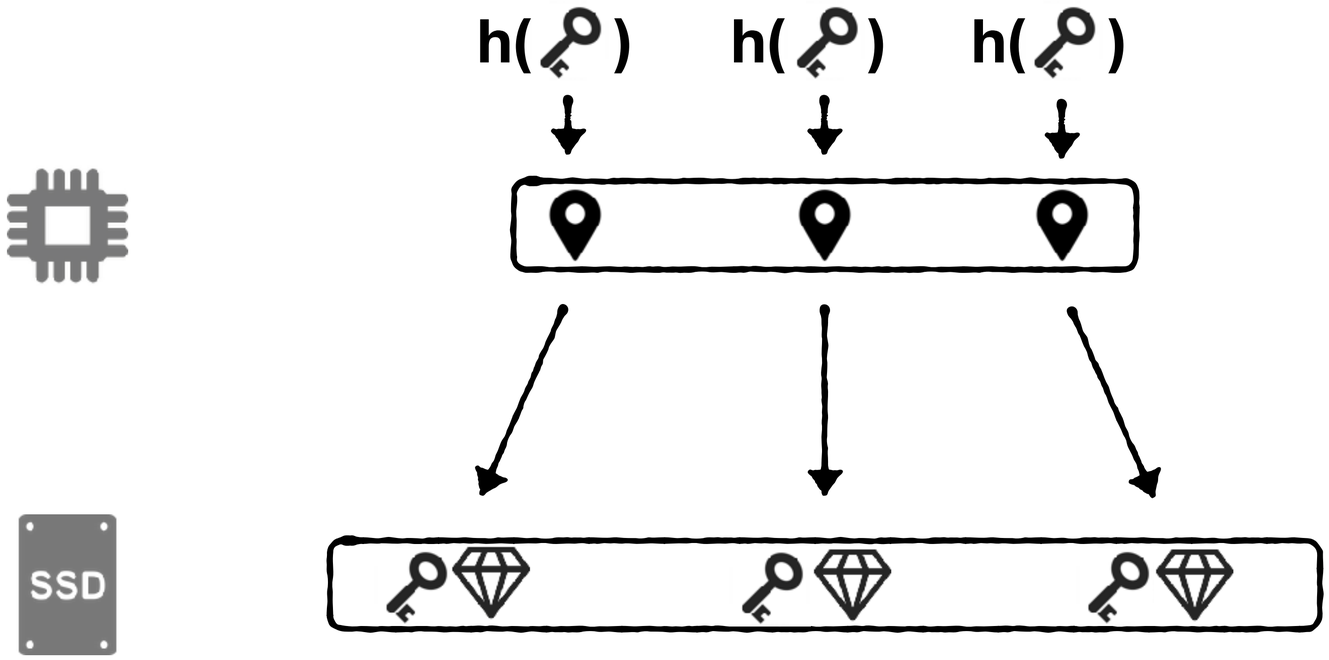


can we get away with not storing keys?

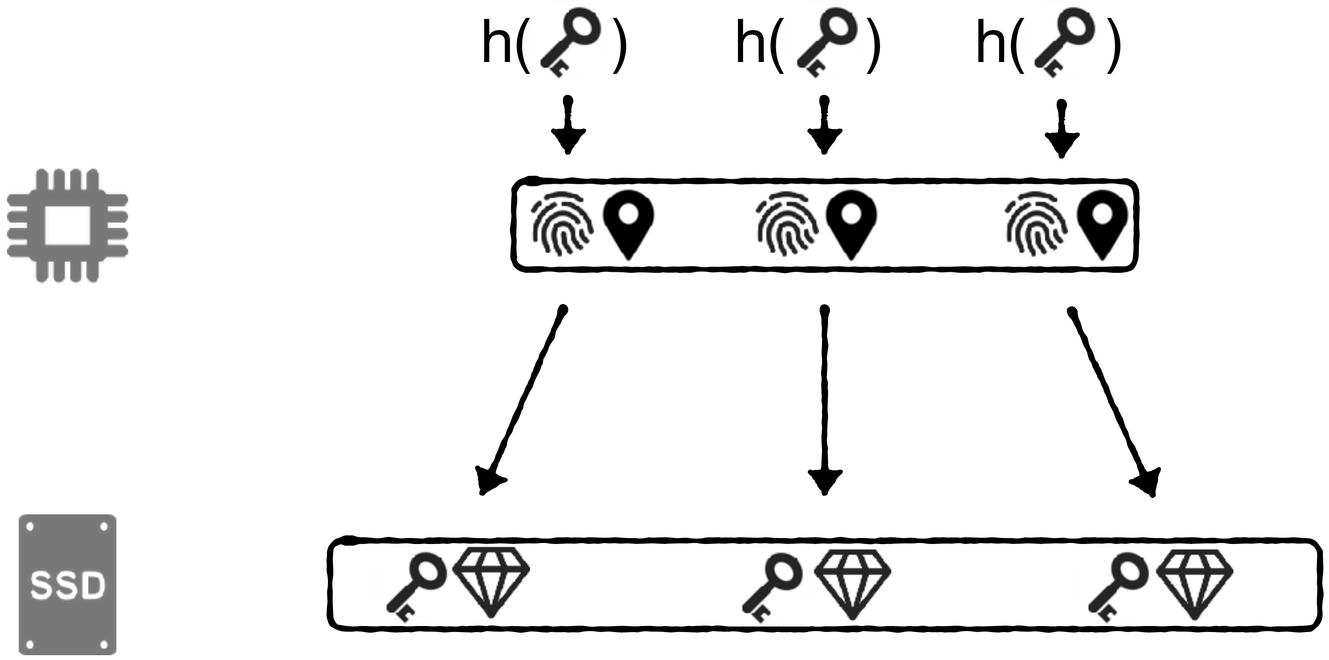


can we get away with not storing keys?

We have seen one solution earlier in the course :)

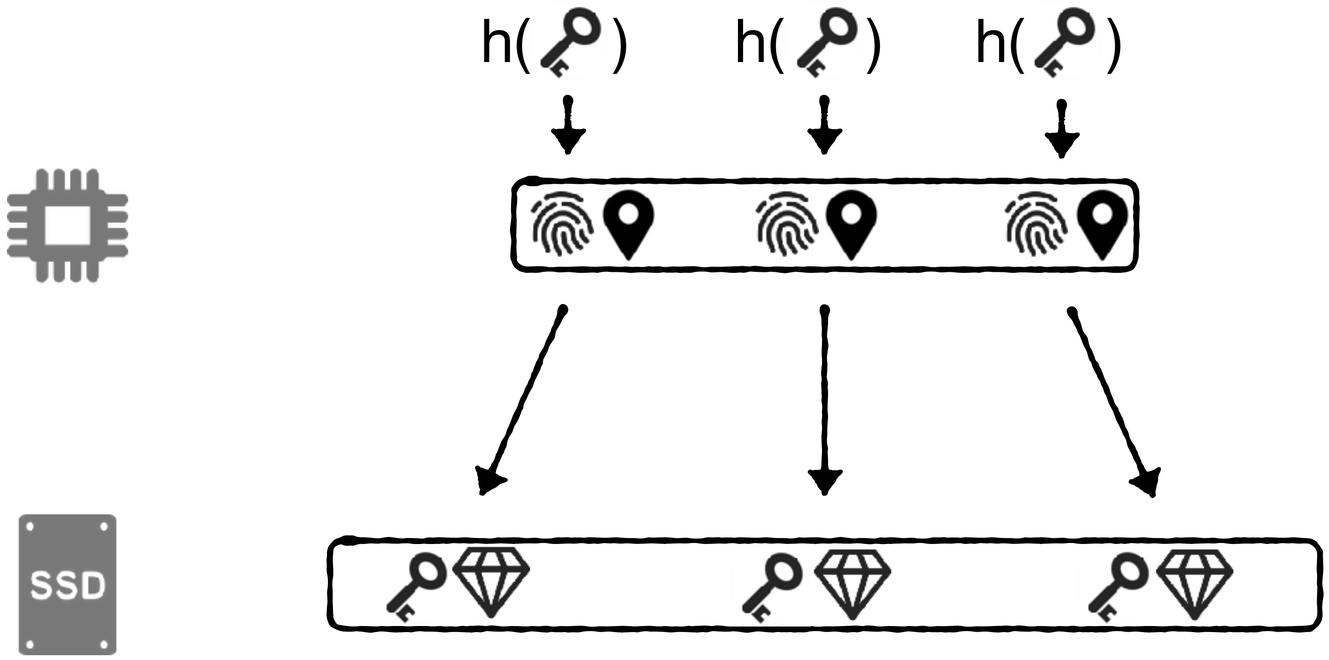


Use a filter (e.g., quotient filter)



Use a filter (e.g., quotient filter)

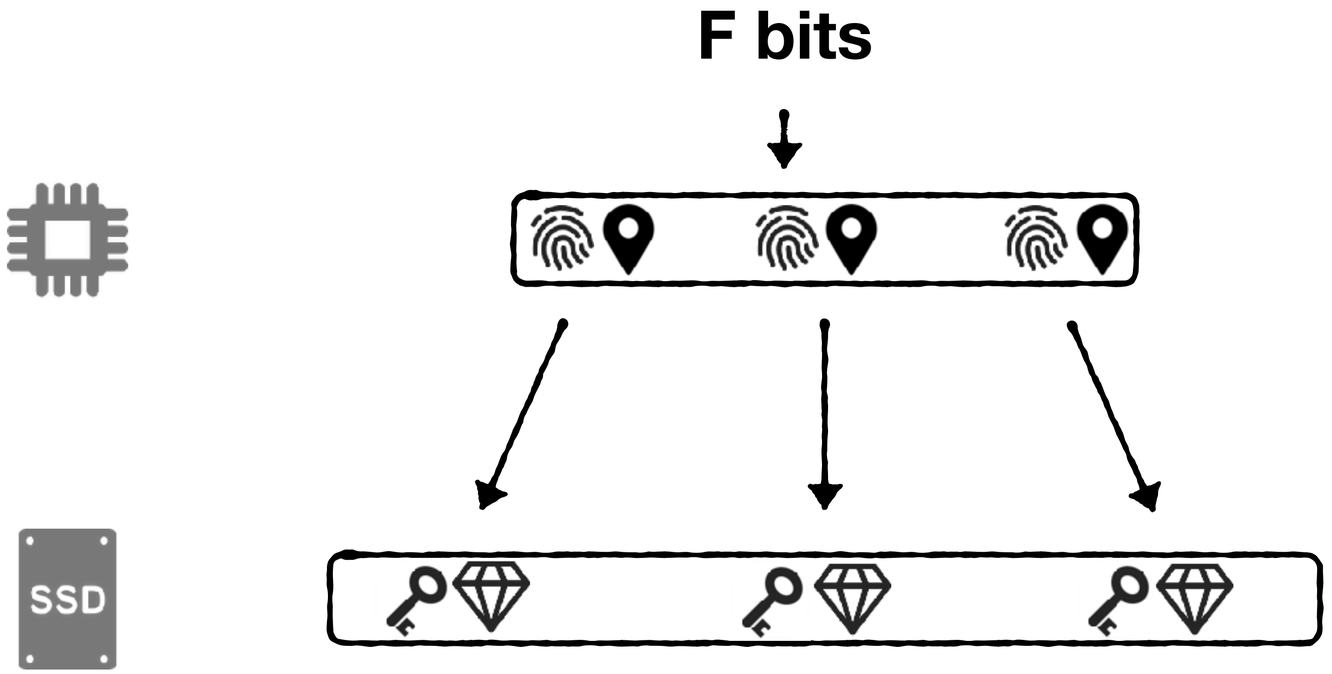
Replace keys with fingerprints to save space



Query I/O costs

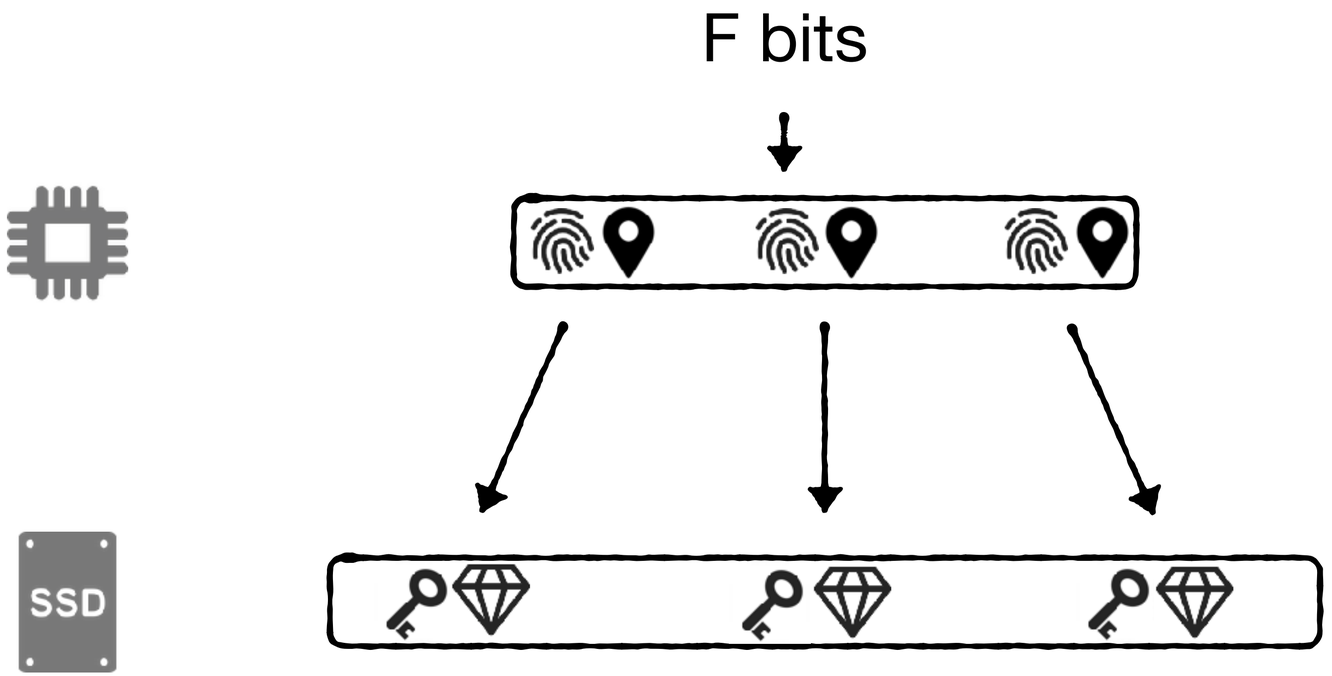
non-existing key?

existing key?



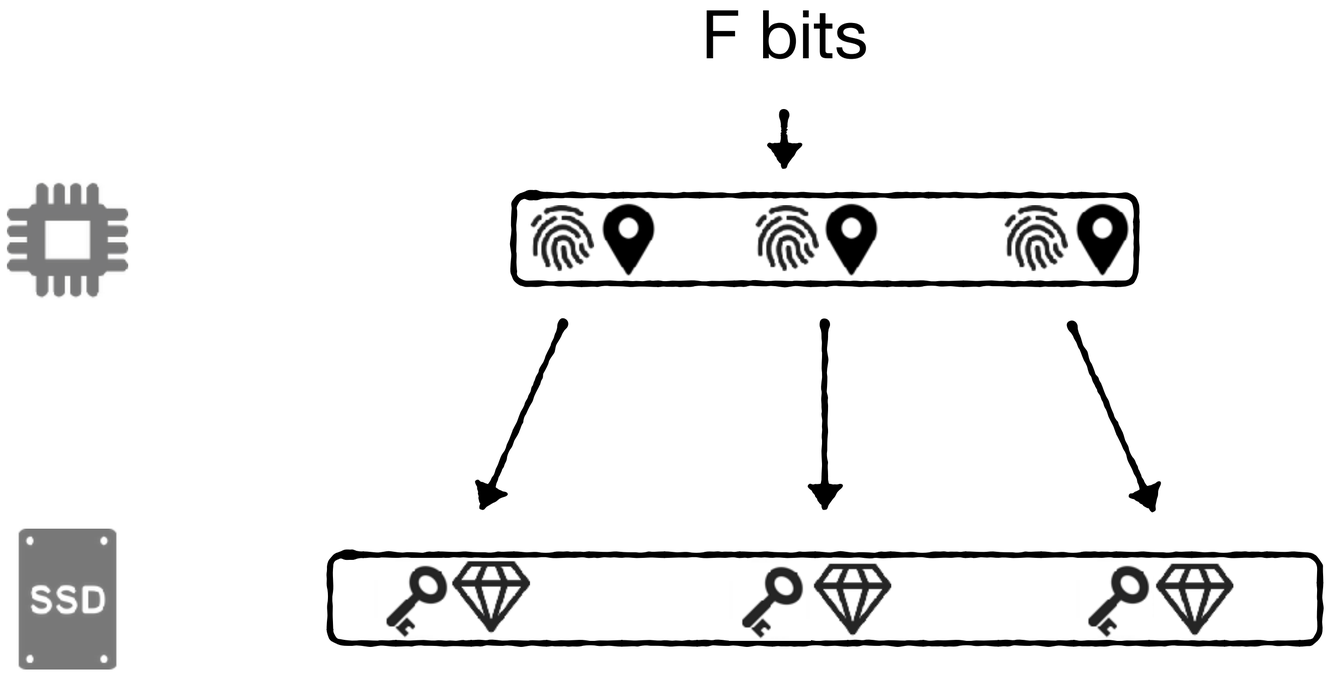
Query I/O costs **non-existing key?** 2^{-F}

existing key?



Query I/O costs

non-existing key?	2^{-F}
existing key?	**$1+2^{-F}$**



Query I/O costs

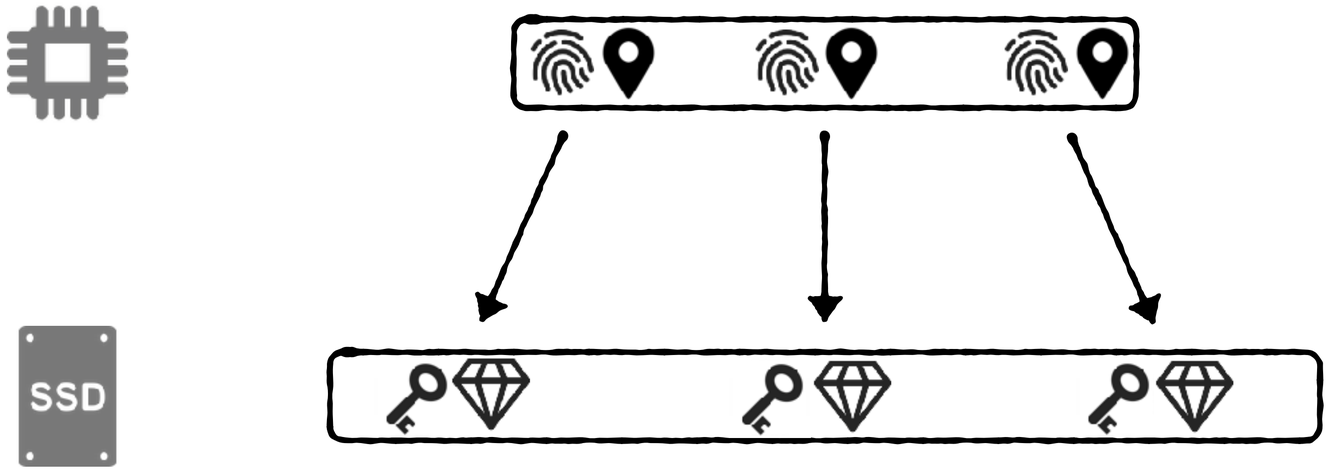
non-existing key? 2^{-F}

existing key? **$1+2^{-F}$**

Let's focus on queries to existing keys

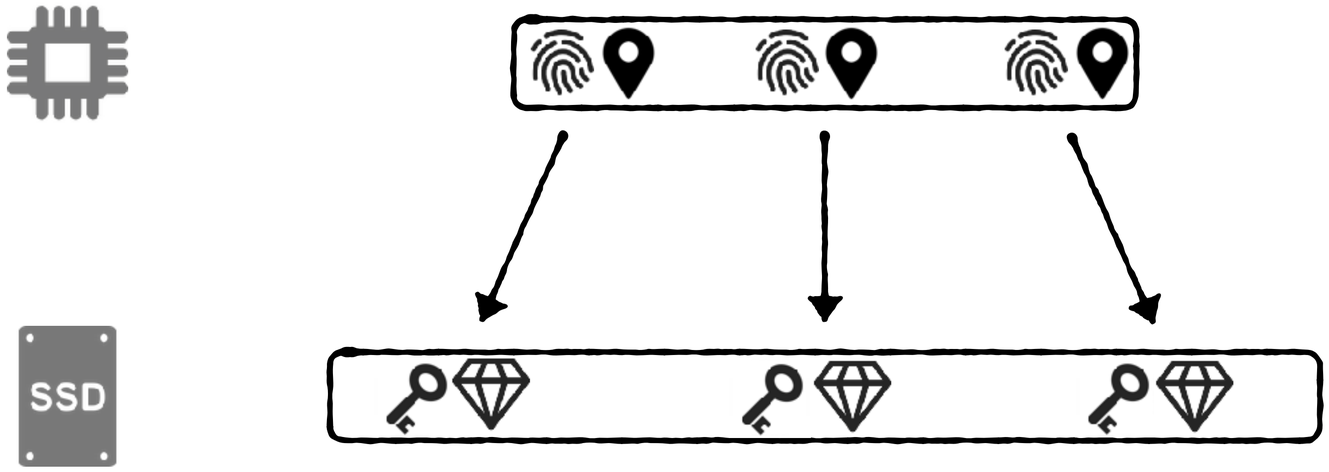
(1) more common

(2) minimize latency for useful work



Query I/O costs existing key? $1+2^{-F}$

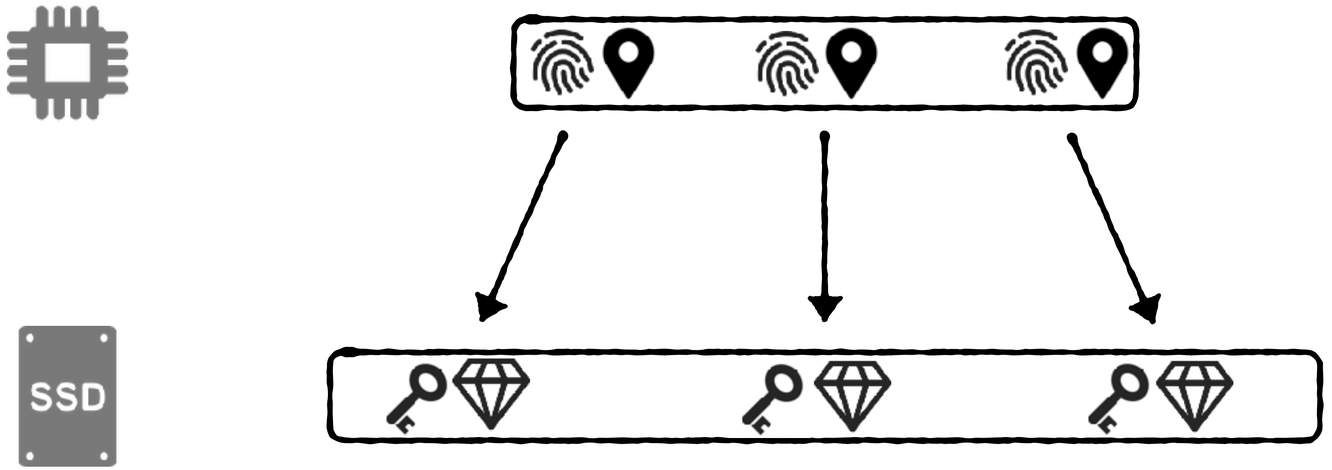
Due to fingerprint collisions



Query I/O costs existing key? $1+2^{-F}$

Due to fingerprint collisions

Can we reduce by increasing F

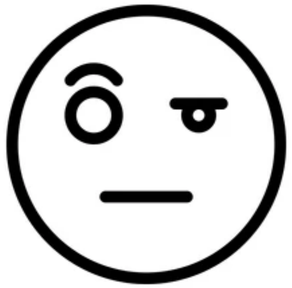


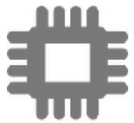
Query I/O costs existing key? $1+2^{-F}$

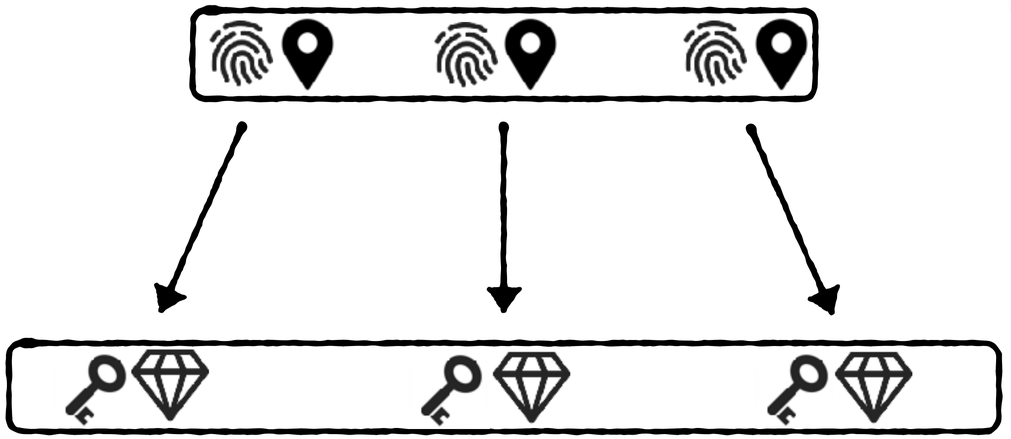
Due to fingerprint collisions

Can we reduce by increasing F

Is there a better way?







Perfect Hashing

Minimal

space-efficient
static data

Dynamic

more space
supports updates

Minimal Perfect Hashing

Array with N slots

Minimal Perfect Hashing

N keys

A B C D E F G H

Array with N slots

Minimal Perfect Hashing

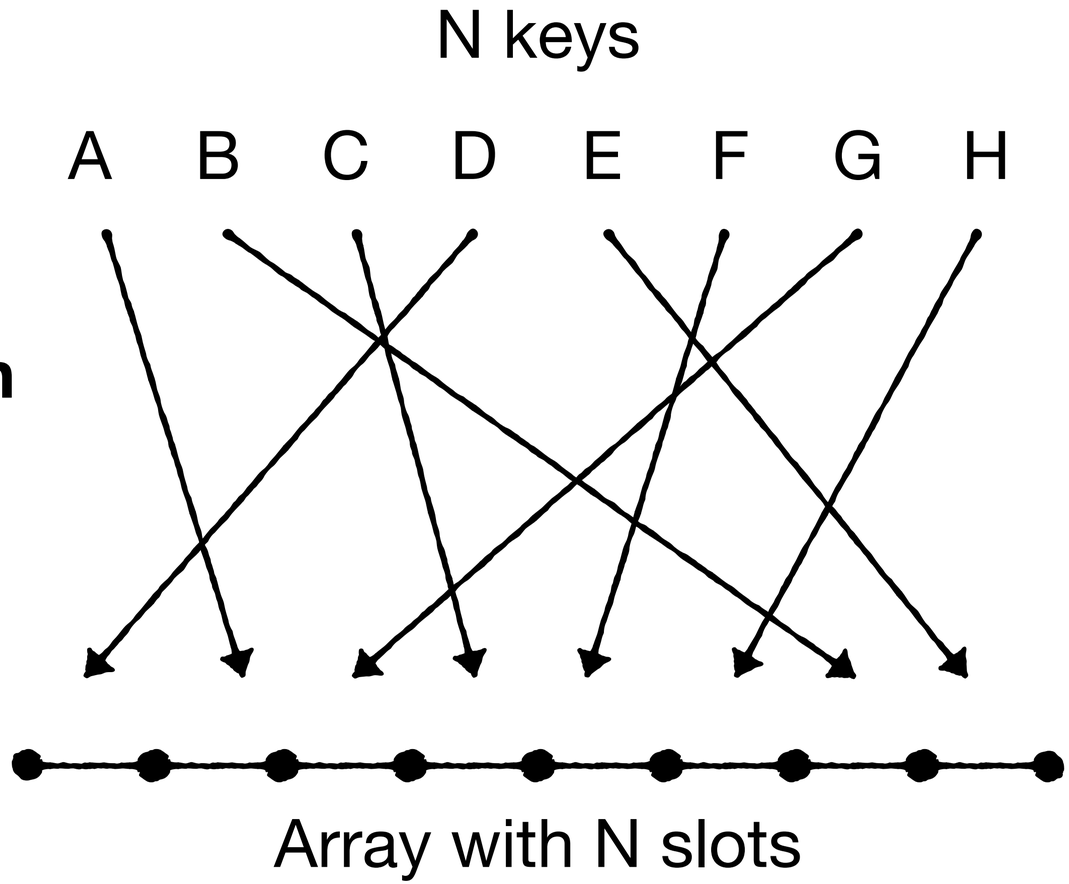
N keys

A B C D E F G H

Array with N slots

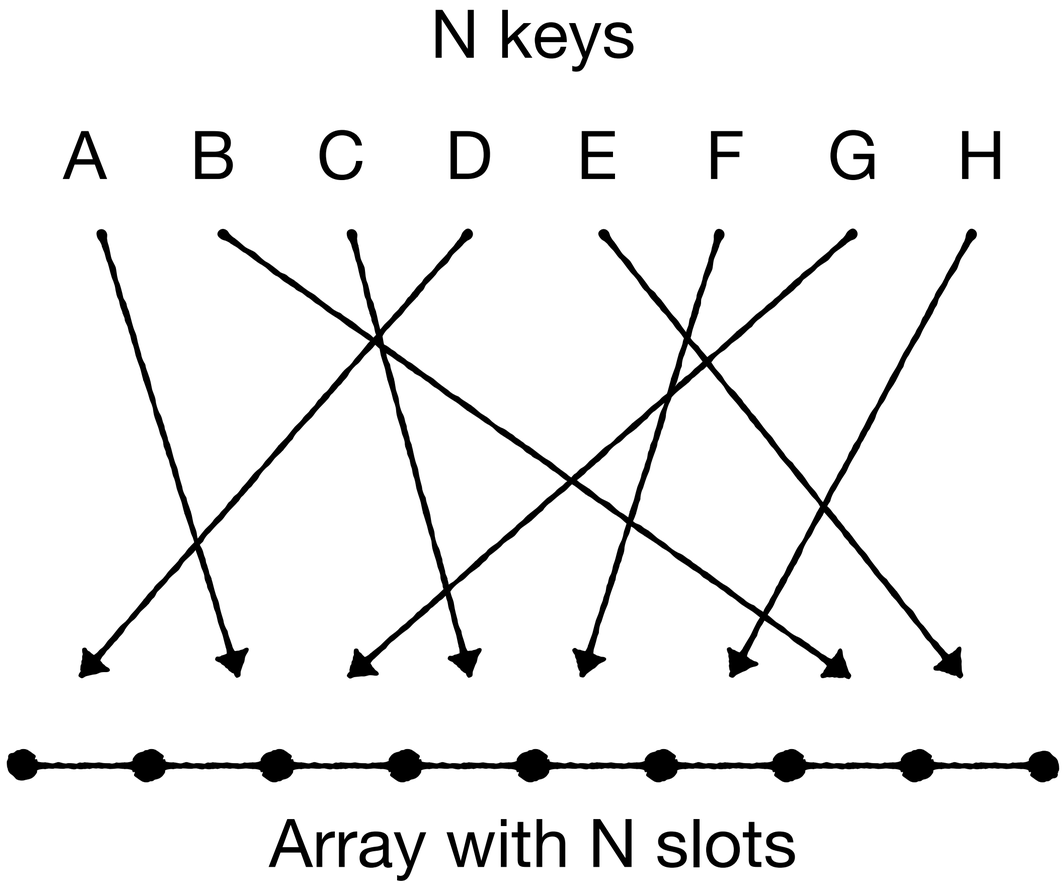
100% load factor! No extra capacity as with normal hash tables

Goal: Establish bijection (one-to-one mapping)

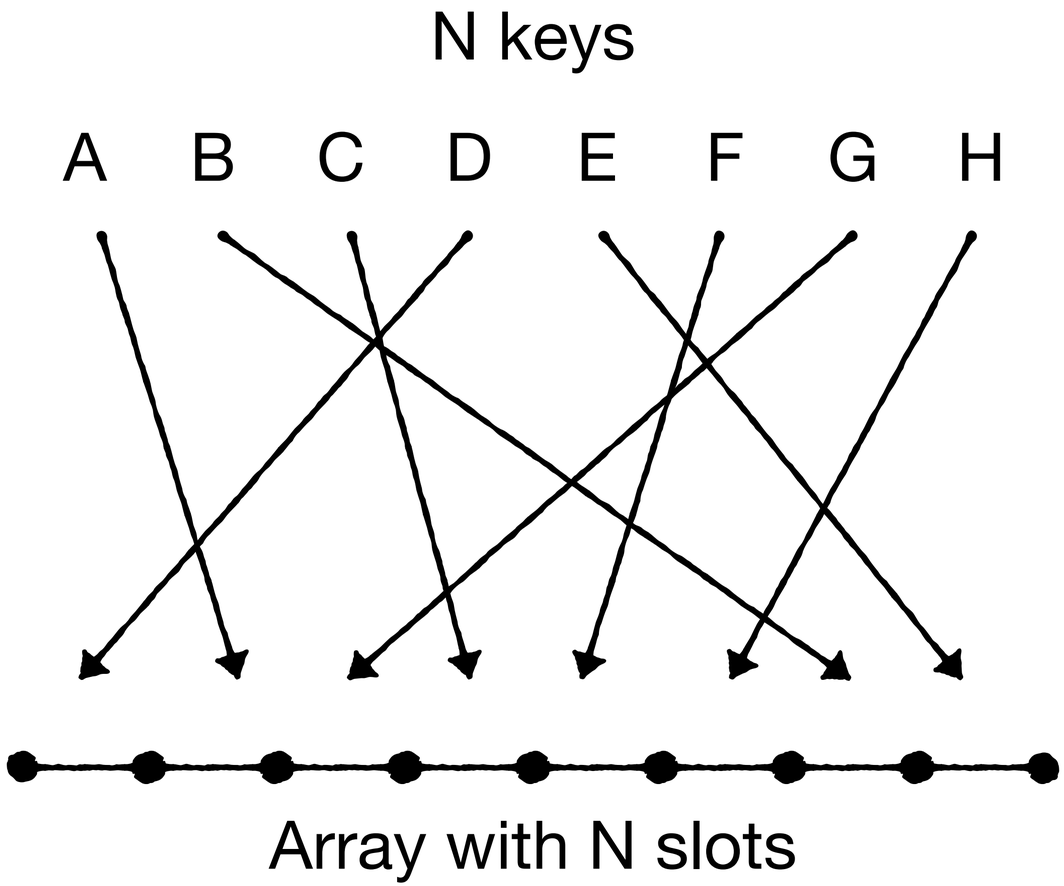


Goal: Establish bijection
(one-to-one mapping)

Collision-free



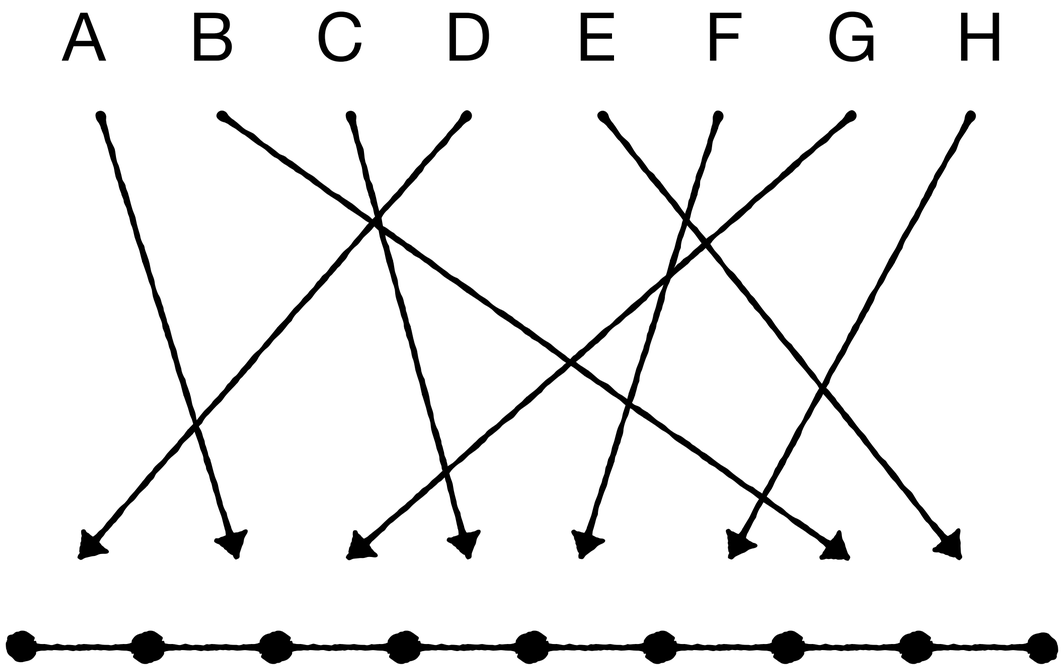
What's the probability of general hash function creating bijection?



What's the probability of general hash function creating bijection?

 # possible bijections (permutations)?

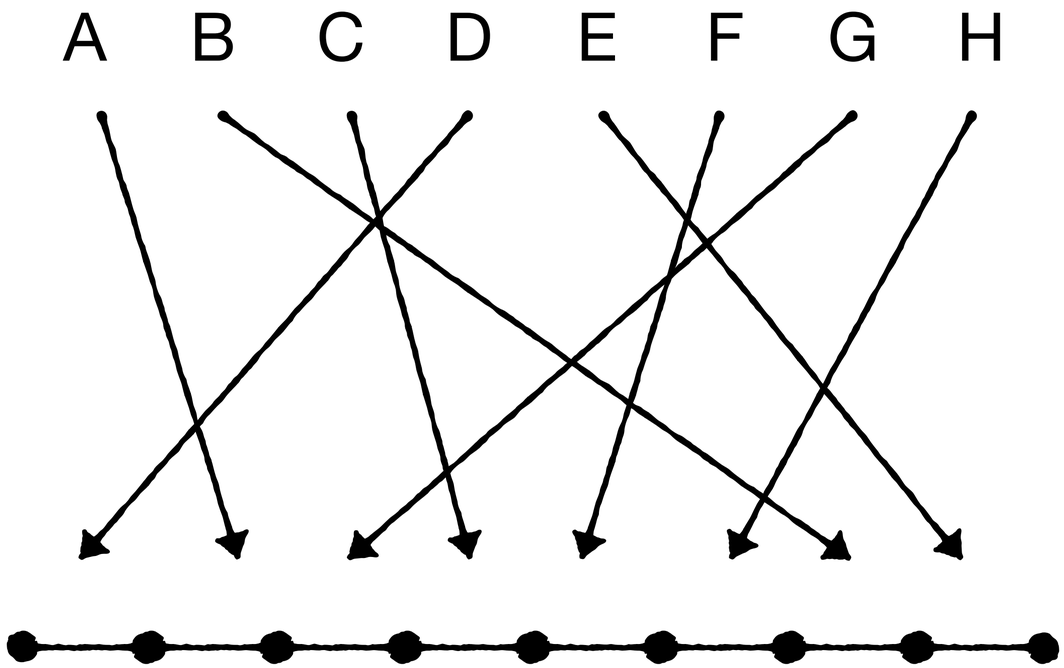
 # possible assignments?



What's the probability of general hash function creating bijection?

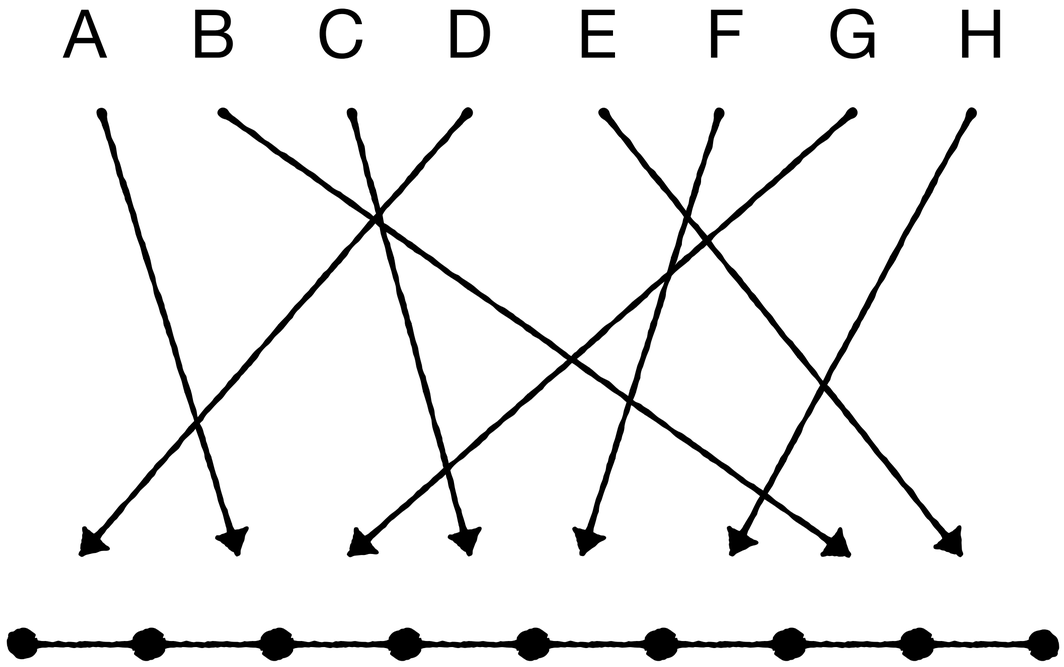
 # possible bijections (permutations)? N!

possible assignments?



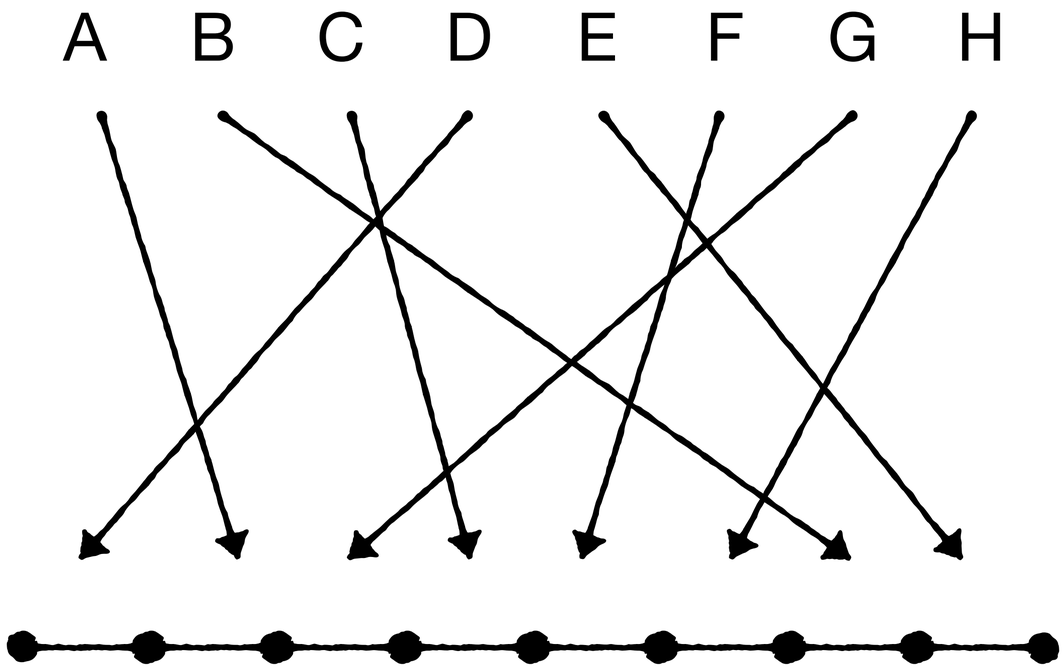
What's the probability of general hash function creating bijection?

# possible bijections (permutations)?	**N!**
# possible assignments?	**N^N**



What's the probability of general hash function creating bijection?

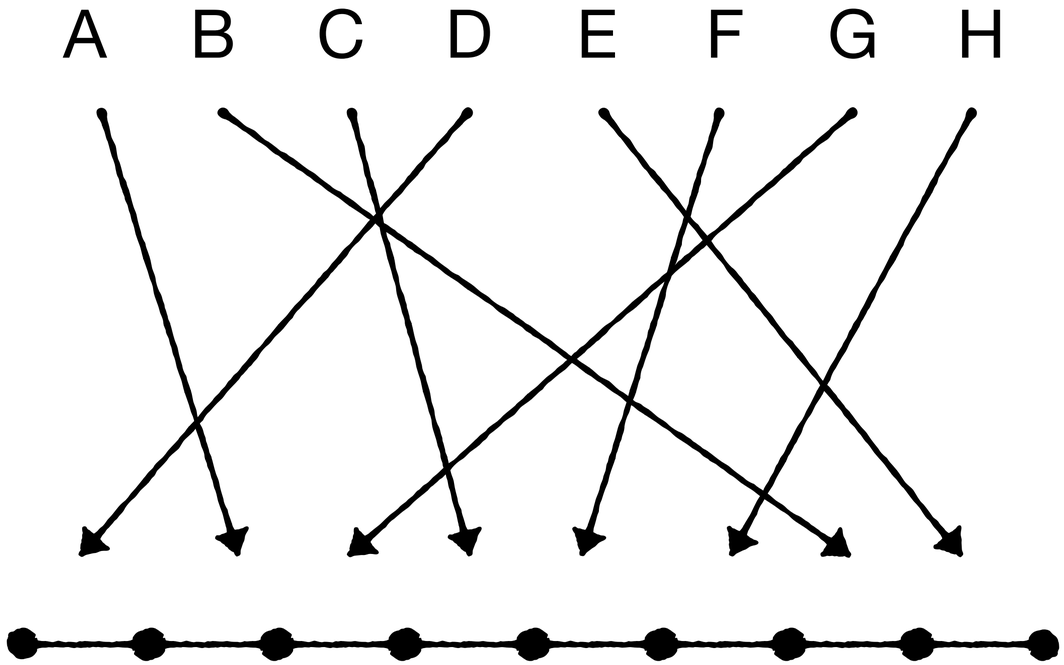
$$\frac{N!}{N^N}$$



What’s the probability of general hash function creating bijection?

$$\frac{N!}{N^N} \approx \sqrt{2\pi N} \cdot e^{-N}$$

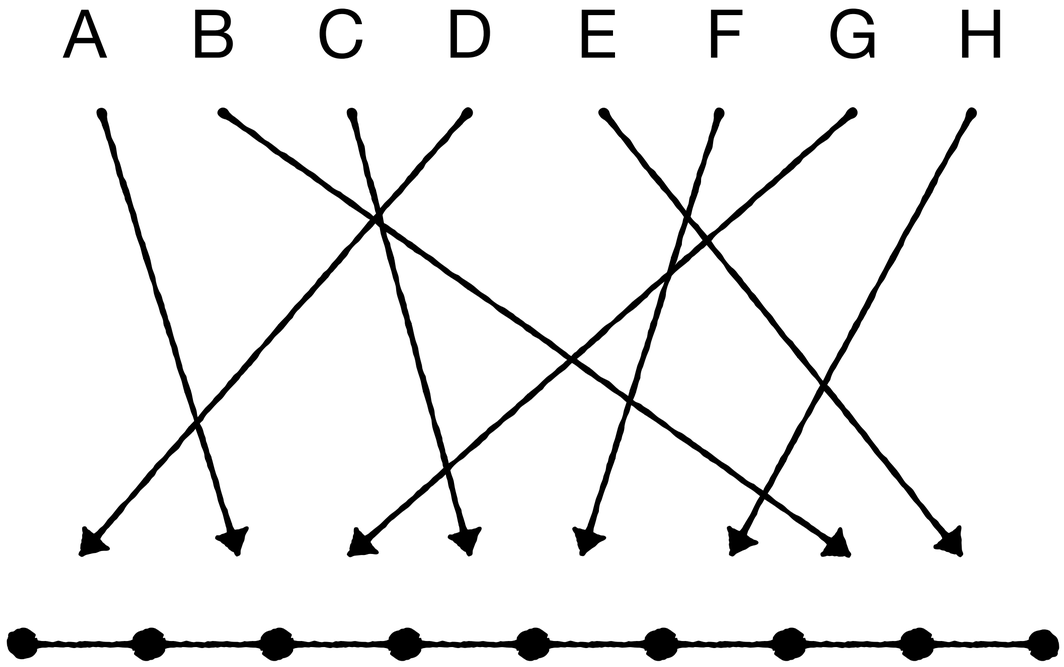
By Stirling’s approximation

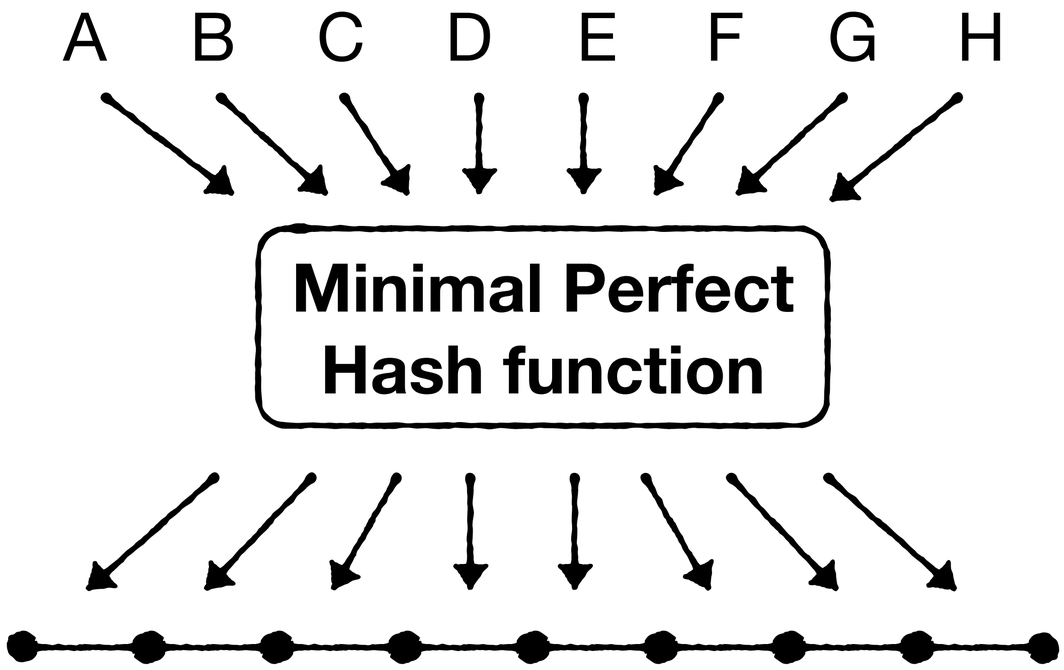


What's the probability of general hash function creating bijection?

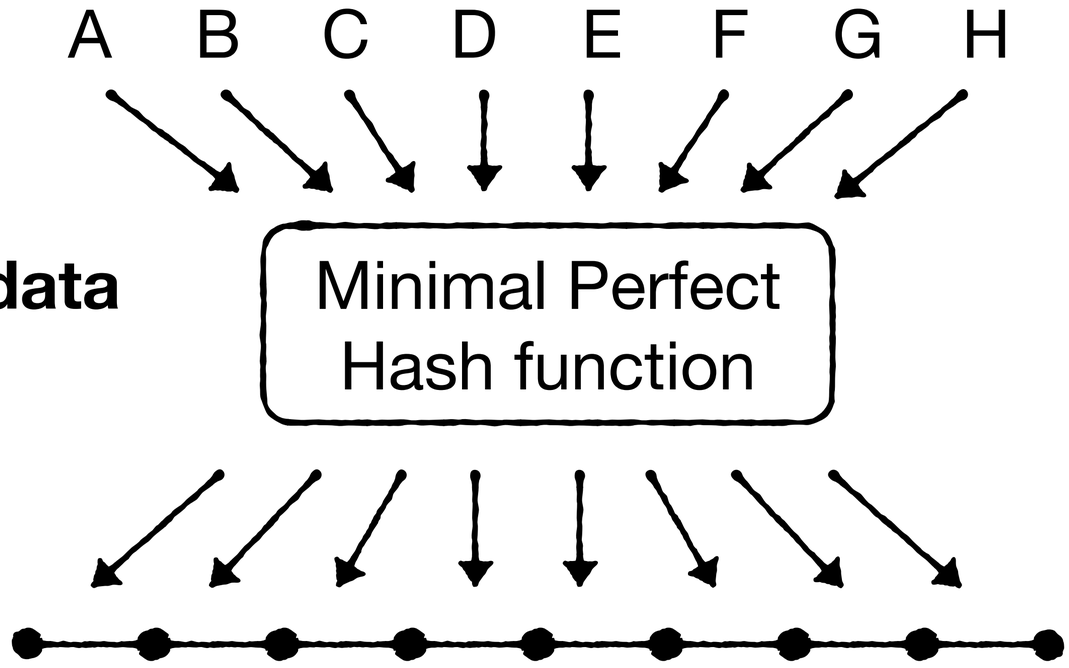
$$\lim_{N \to \infty} \approx \sqrt{2 \pi N} \cdot e^{-N} = 0$$

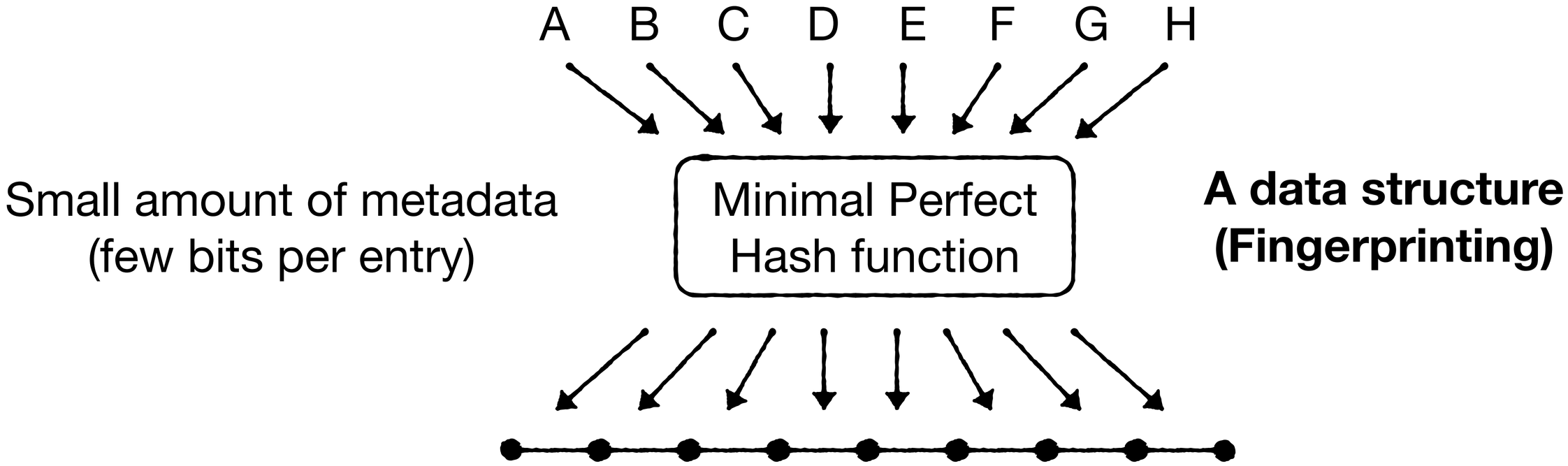
What can we do instead?




A B C D E F G H
Minimal Perfect
Hash function

Small amount of metadata (few bits per entry)

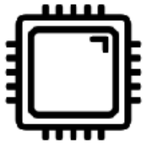




Small amount of metadata
(few bits per entry)

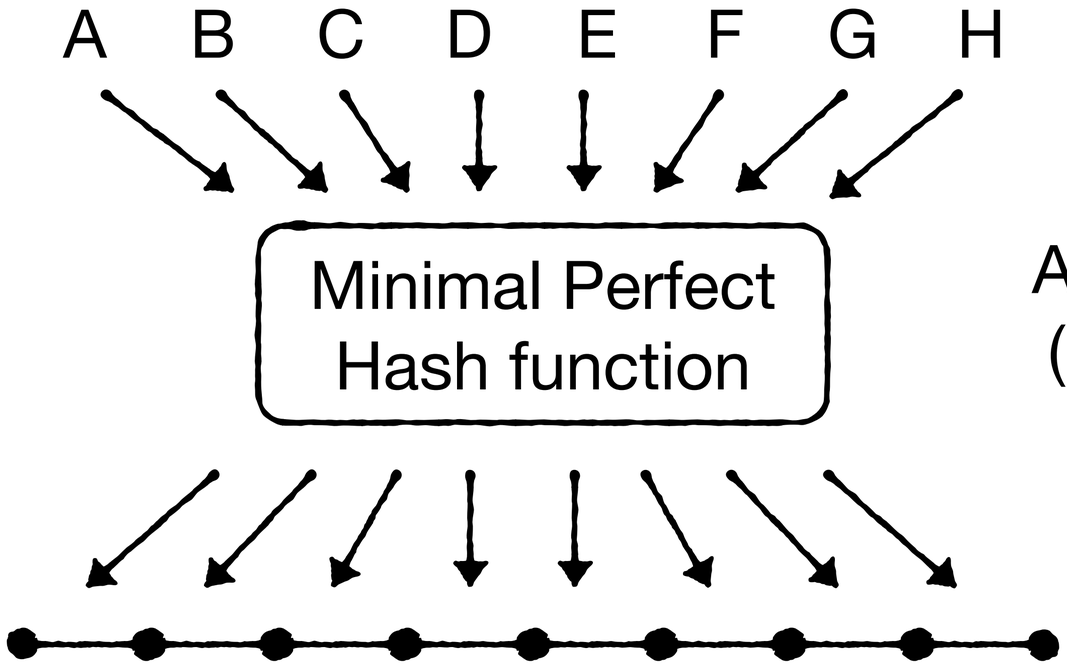
A data structure
(Fingerprinting)

Memory
(Bits / entry)



Query cost

Construction time



A data structure
(Fingerprinting)

Fingerprinting

Fingerprinting-based Minimal Perfect Hashing Revisited. JEA 2023.
Piotr Beling.

Retrieval and Perfect Hashing using Fingerprinting. JEA 2014.
Ingo Müller, Peter Sanders, Robert Schulze & Wei Zhou.

Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. SEA 2017.
Antoine Limasset, Guillaume Rizk, Rayan Chikhi, Pierre Peterlongo.

Meraculous: de novo genome assembly with short paired-end reads. PloS one 2011.
Jarrod A. Chapman ,Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, Daniel S. Rokhsar

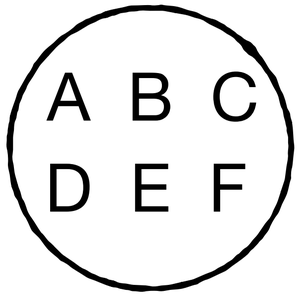
Perfect Hashing for Network Applications. ISIT 2006.
Yi Lu, Balaji Prabhakar, Flavio Bonomi.

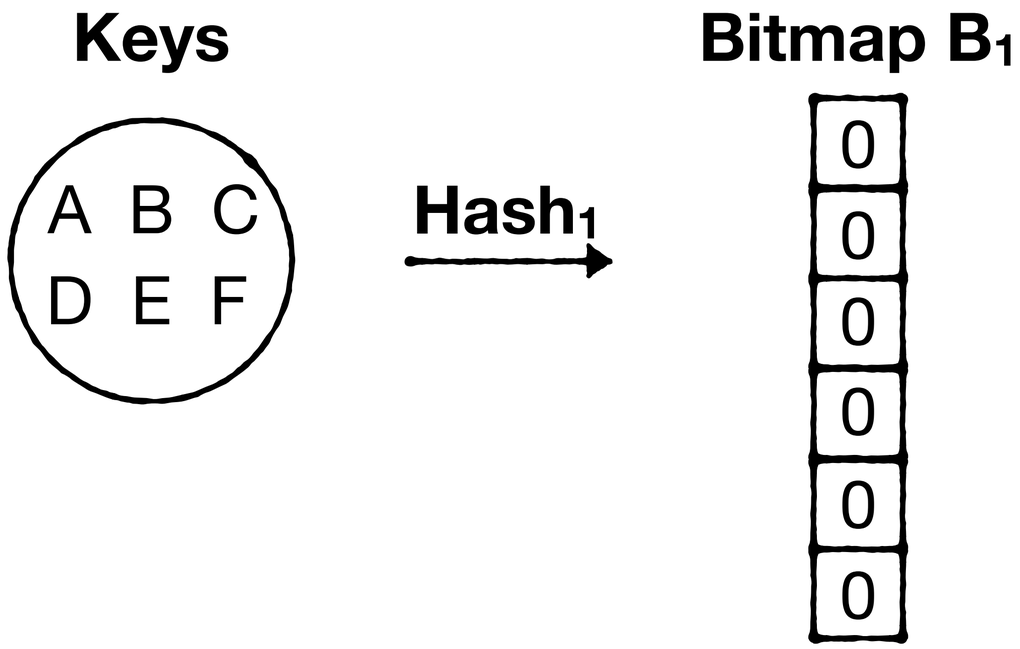
Fingerprinting

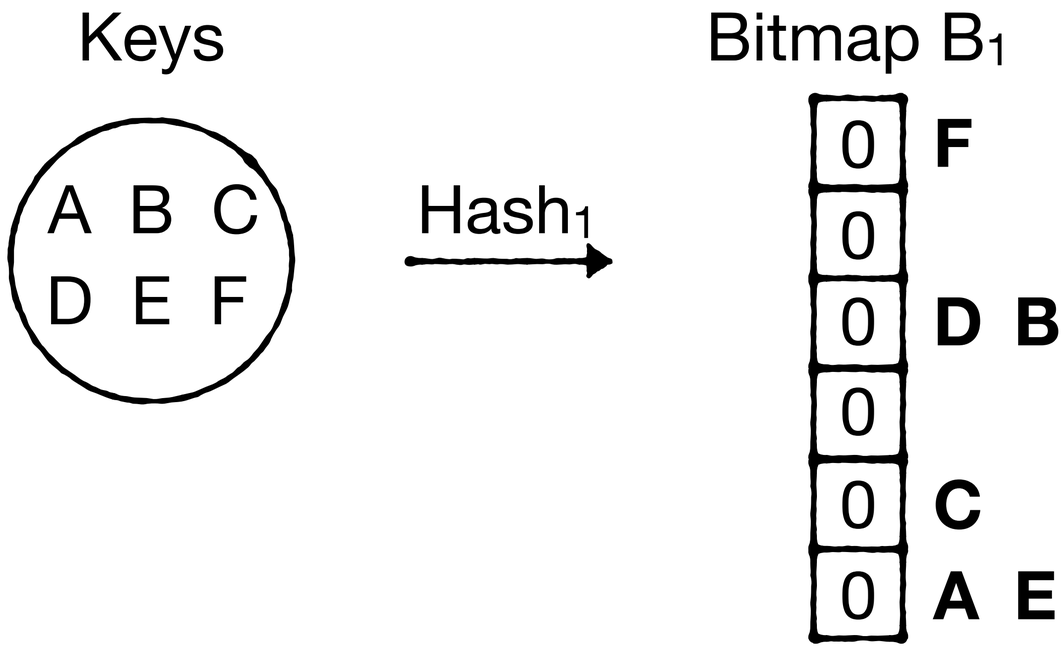
Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. SEA 2017.
Antoine Limasset, Guillaume Rizk, Rayan Chikhi, Pierre Peterlongo.

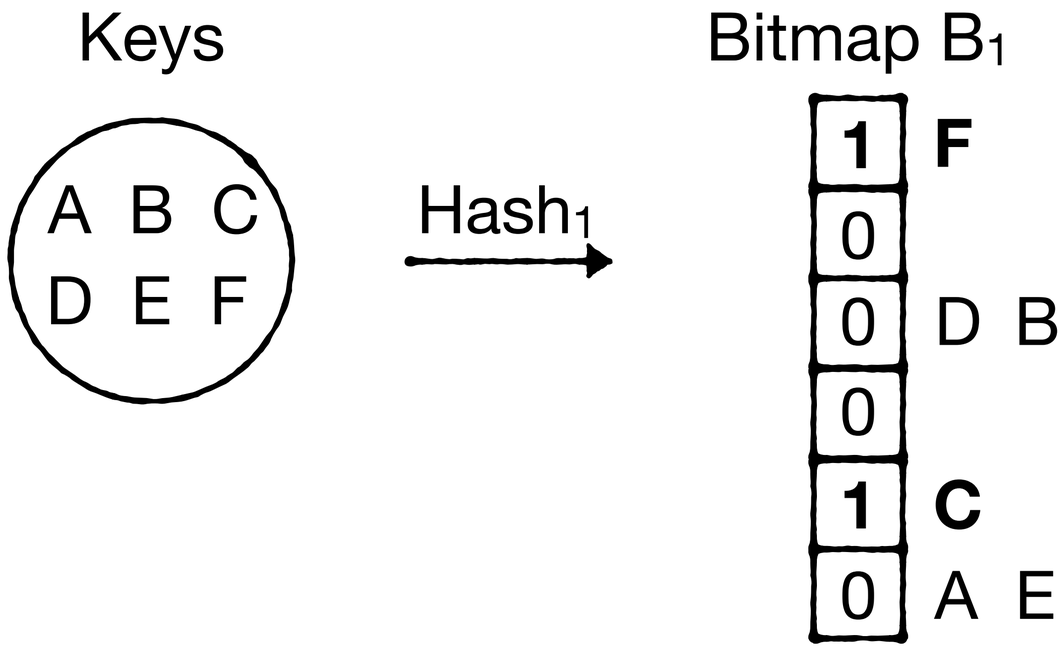
Accessible & Experimental

Keys

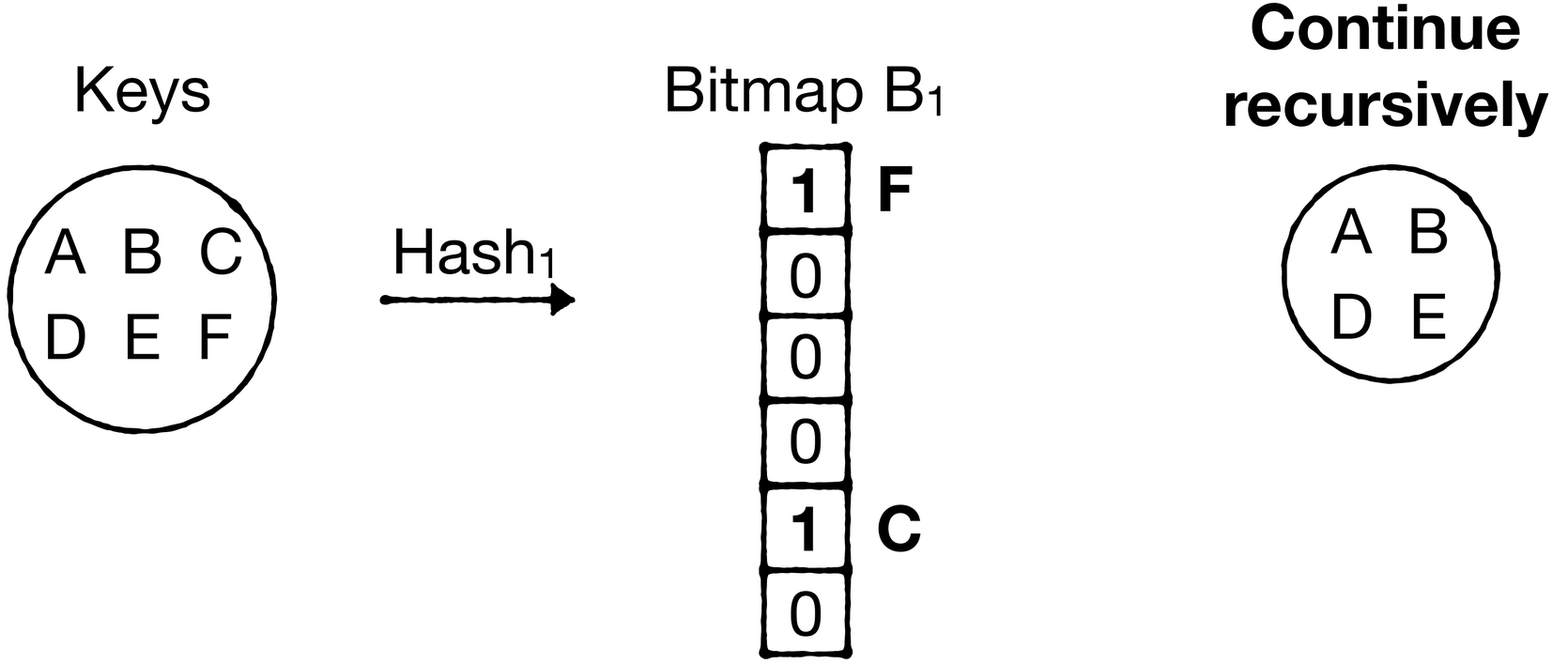


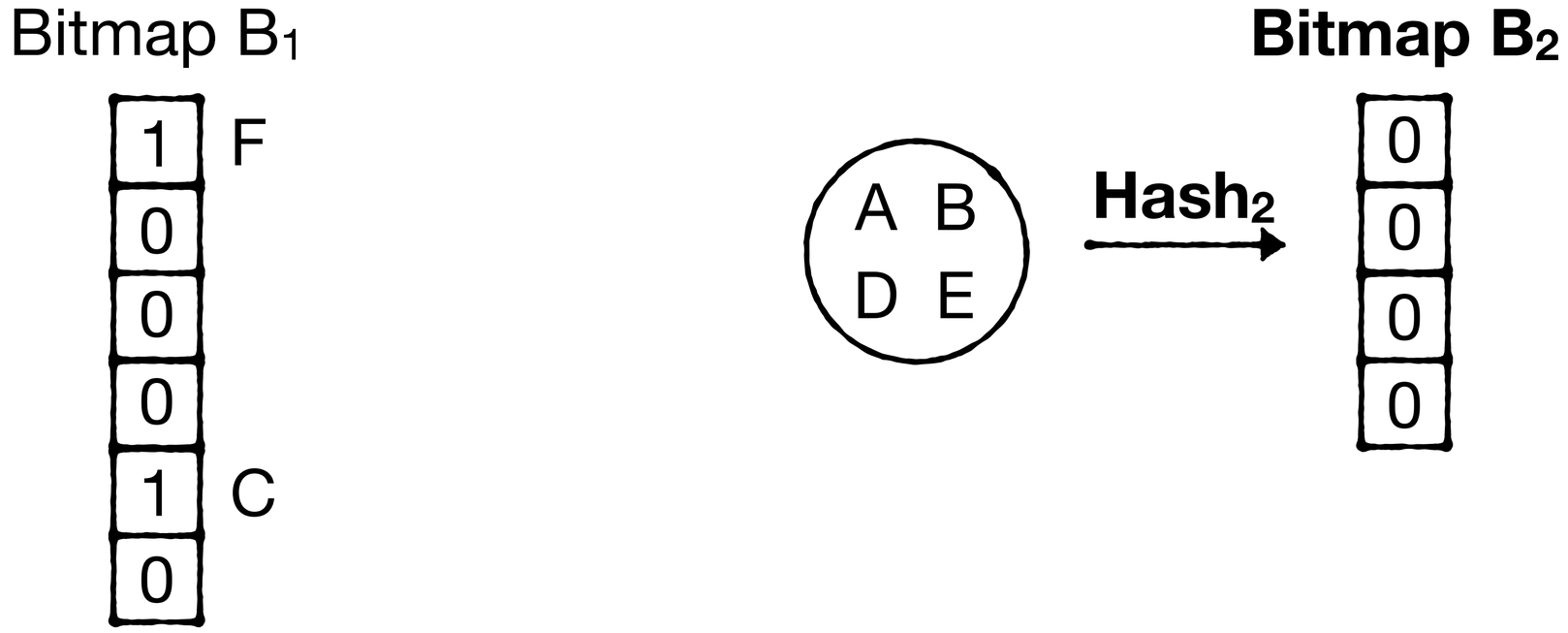

Keys
A B C
D E F
Hash1
Bitmap B1
0
0
0
0
0
0

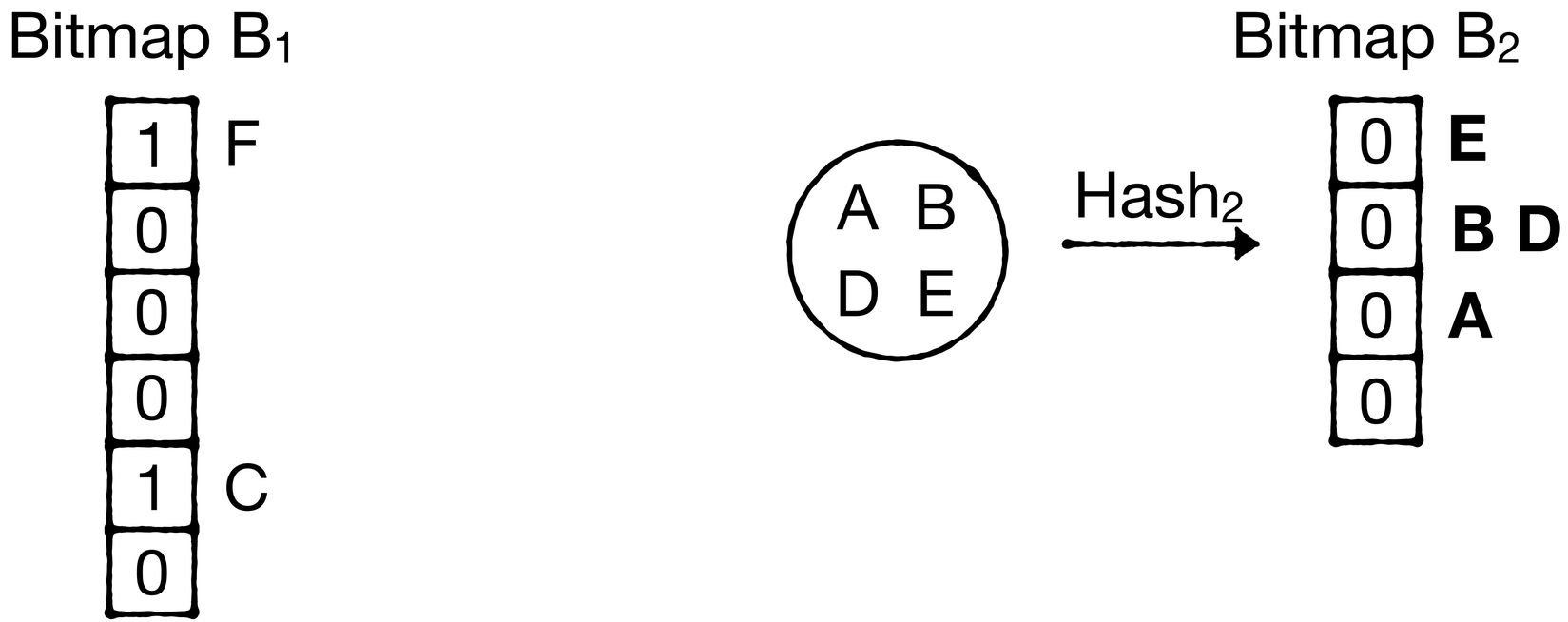

Keys
A B C
D E F
Hash1
Bitmap B1
0 F
0
0 D B
0
0 C
0 A E

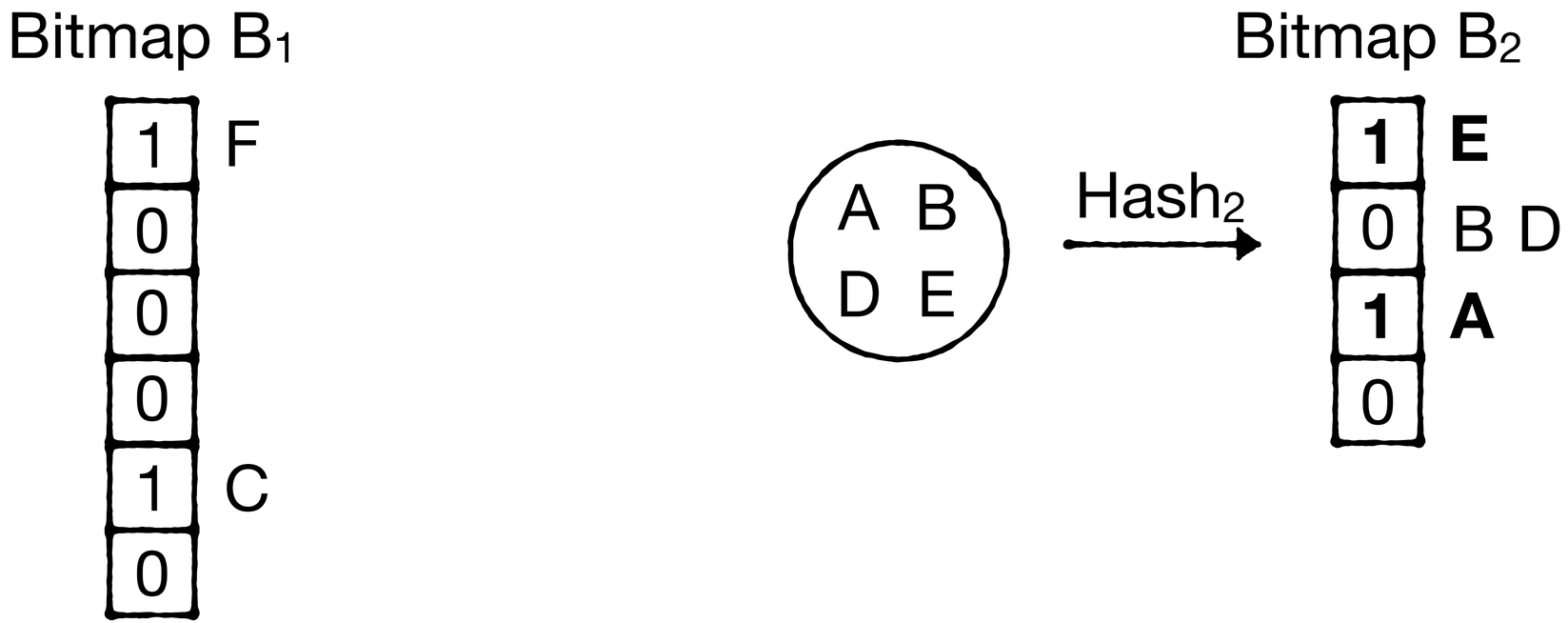


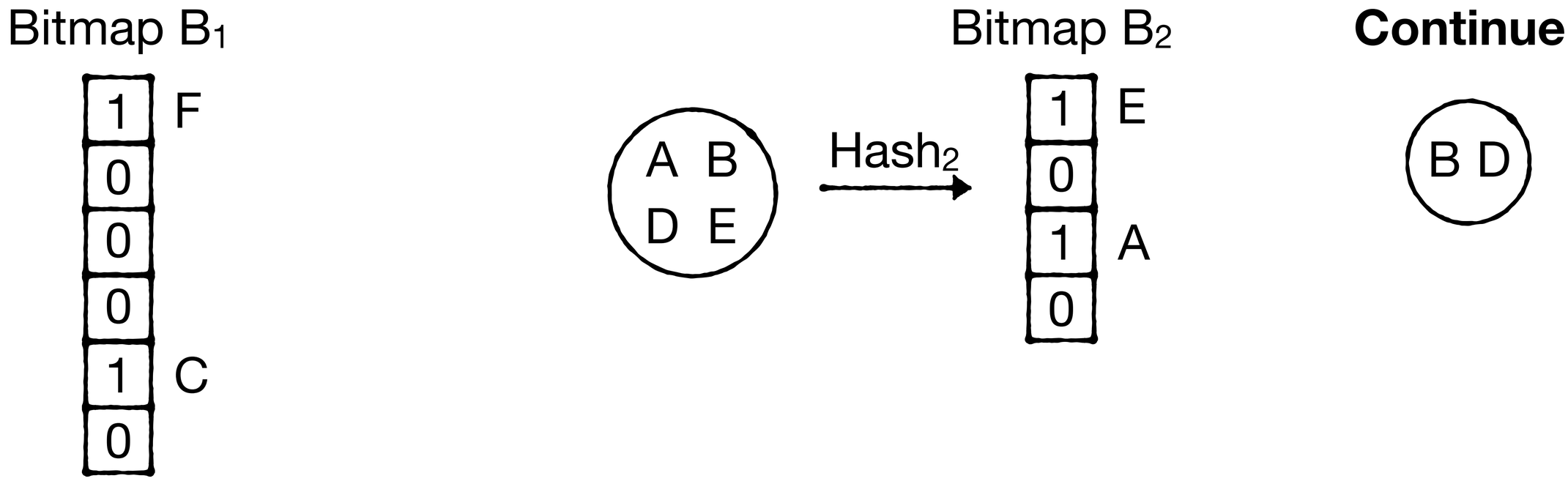
Set to 1 only if one entry mapped to this bit


Keys
A B C
D E F
Hash1
Bitmap B1
1
0
0
0
1
0
F
C
Continue recursively
A B
D E

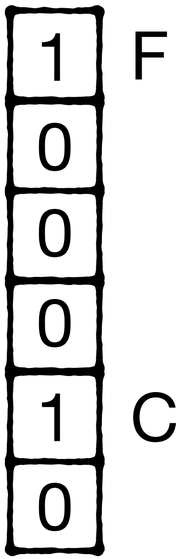

Bitmap B_1
1
F
0
0
0
1
C
0
A B
D E
Hash$_2$
Bitmap B_2
0
0
0
0


Bitmap B_1
1
0
0
0
1
0
F
C
A B
D E
$Hash_2$
Bitmap B_2
0
0
0
0
E
B D
A


Bitmap B_1
1
0
0
0
1
0
F
C
A B
D E
$Hash_2$
Bitmap B_2
1
0
1
0
E
B D
A


Bitmap B_1
1
0
0
0
1
0
F
C
A B
D E
$Hash_2$
Bitmap B_2
1
0
1
0
E
A
Continue
B D

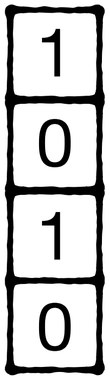
Bitmap B_1



F

C

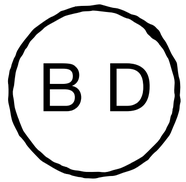
Bitmap B_2

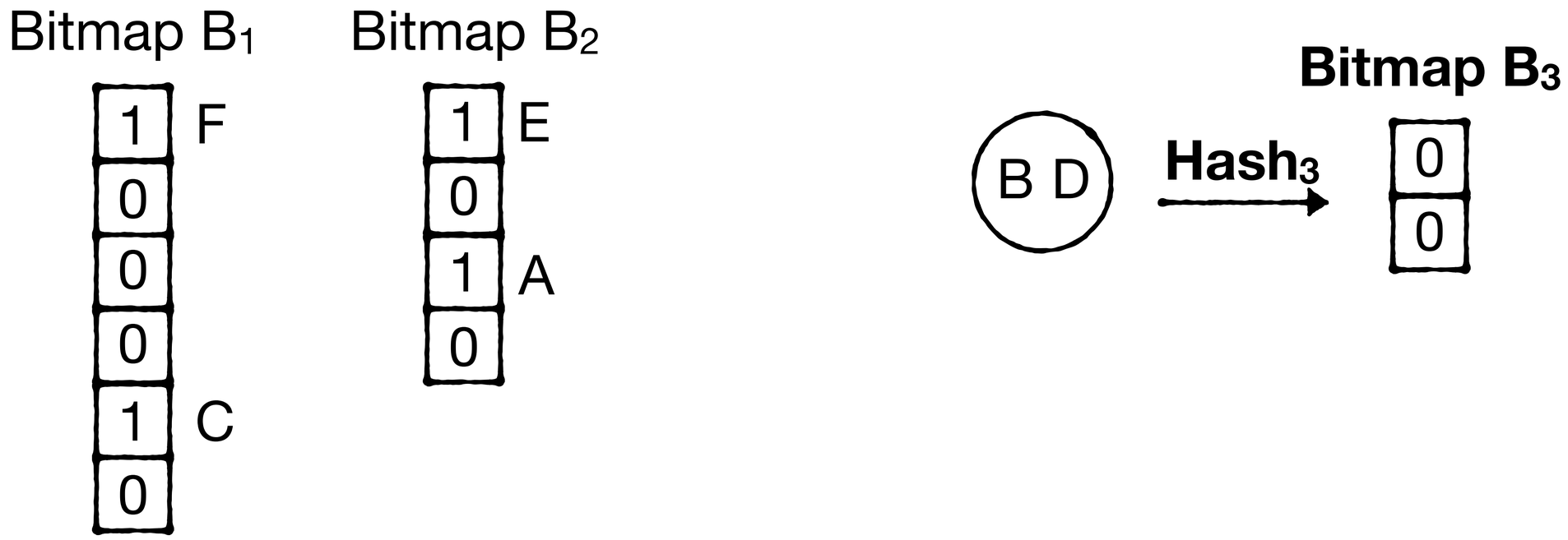


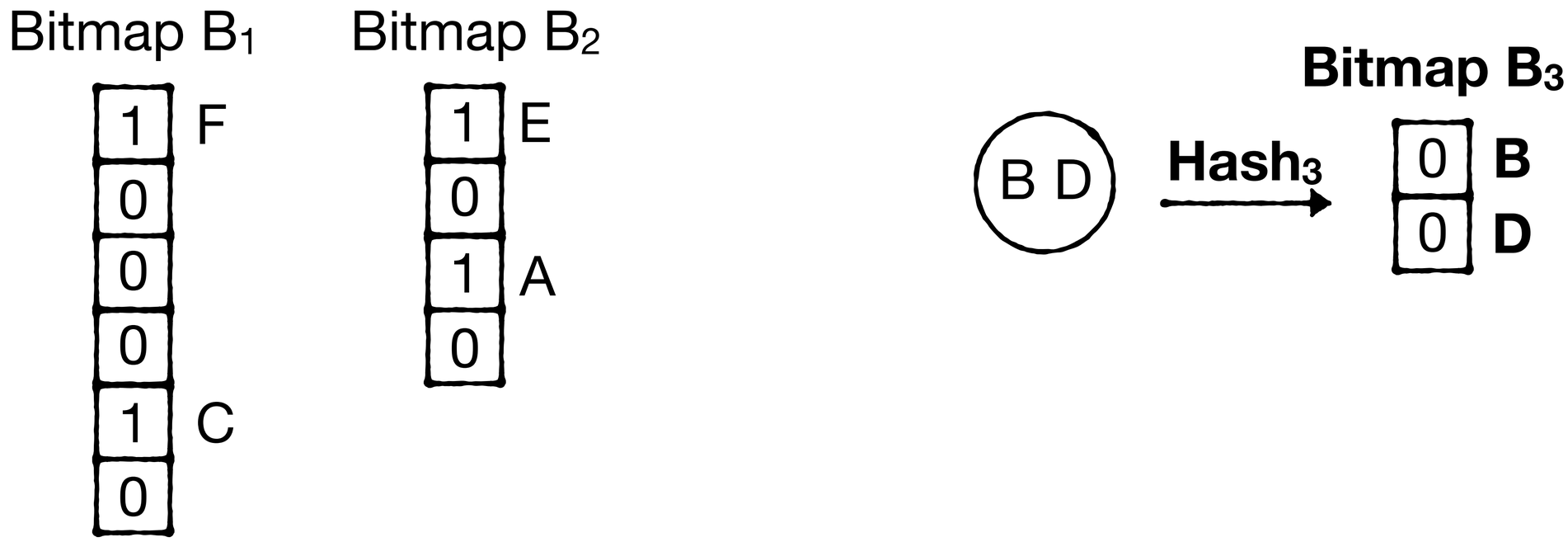
E

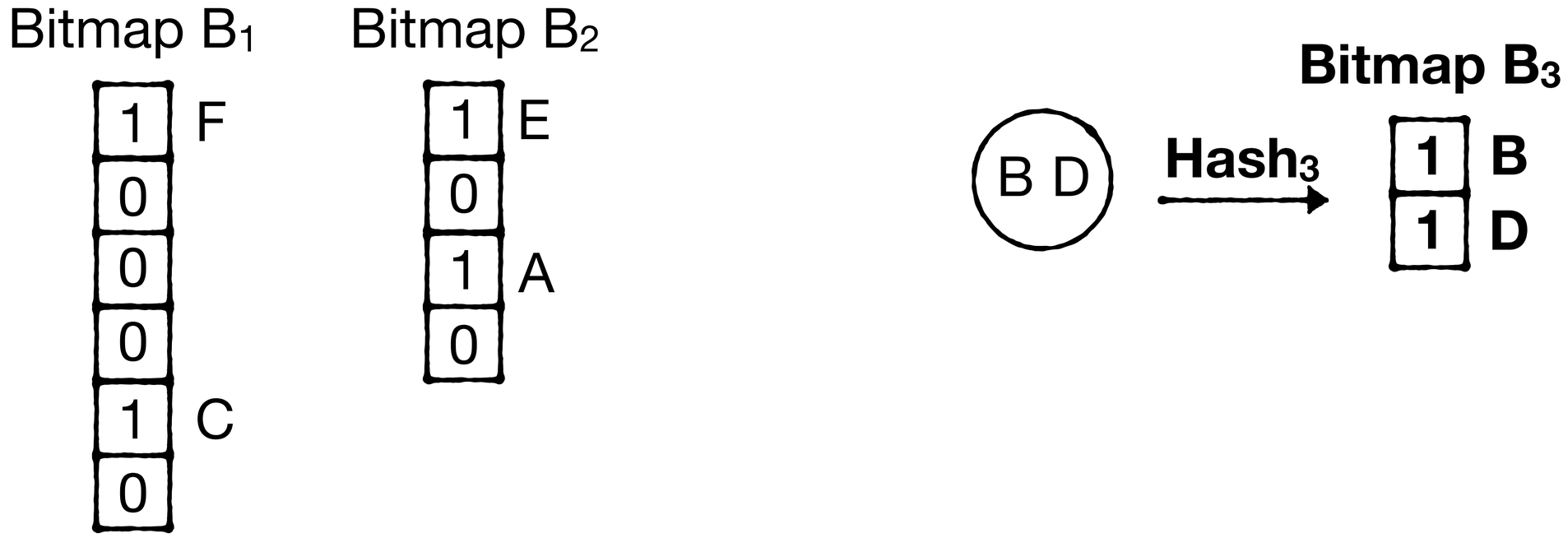
A

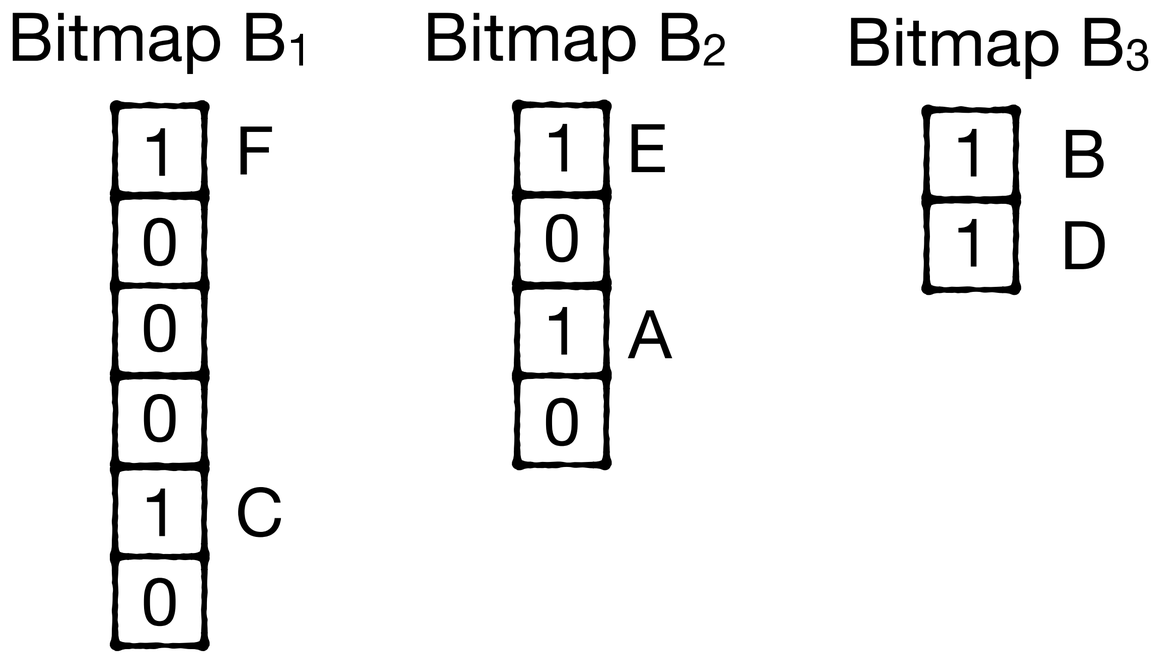
Continue



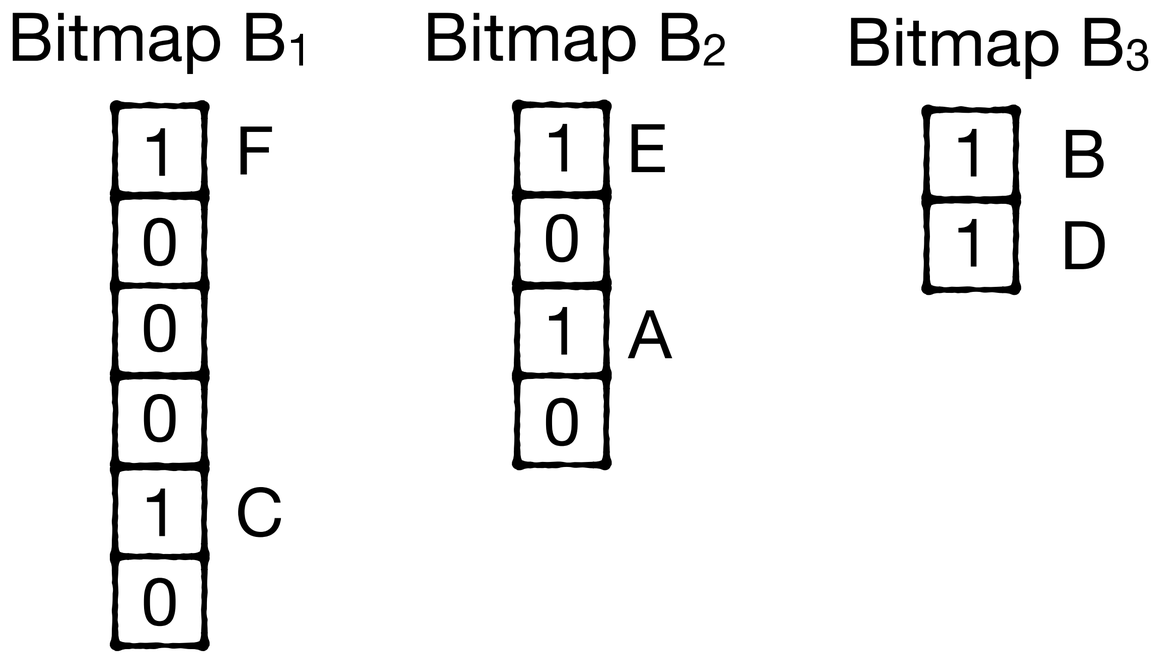

Bitmap B_1
1
0
0
0
1
0
F
C
Bitmap B_2
1
0
1
0
E
A
B D
$Hash_3$
Bitmap B_3
0
0


Bitmap B_1
1
0
0
0
1
0
F
C
Bitmap B_2
1
0
1
0
E
A
B D
$Hash_3$
Bitmap B_3
0
0
B
D

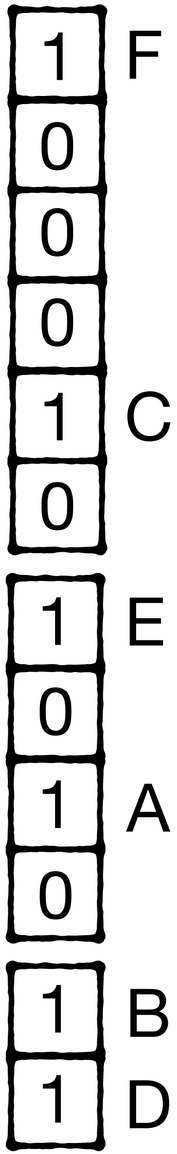

Bitmap B_1
1
0
0
0
1
0
F
C
Bitmap B_2
1
0
1
0
E
A
B D
$Hash_3$
Bitmap B_3
1
1
B
D


Bitmap B_1
1
0
0
0
1
0
F
C
Bitmap B_2
1
0
1
0
E
A
Bitmap B_3
1
1
B
D

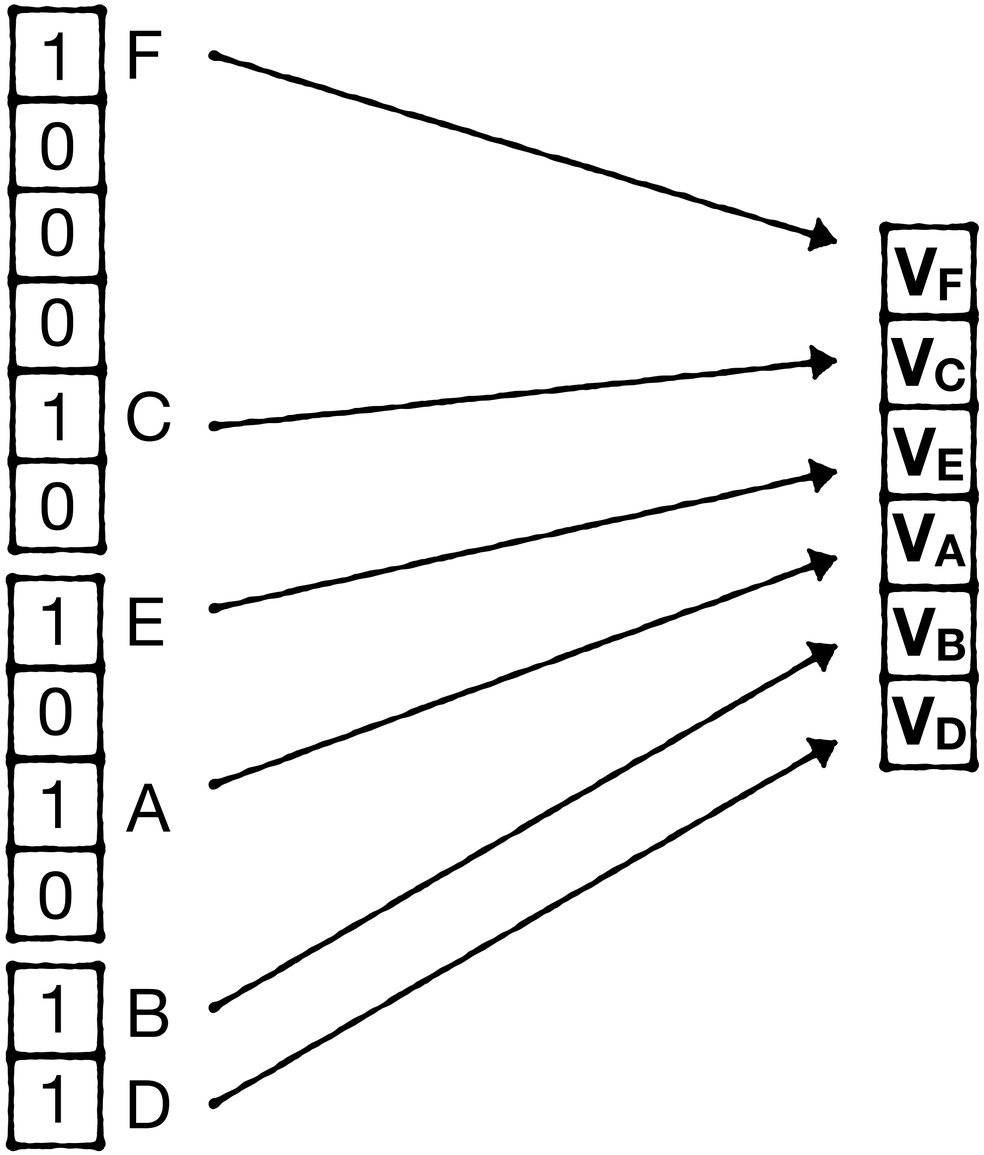
Concatenate bitmaps



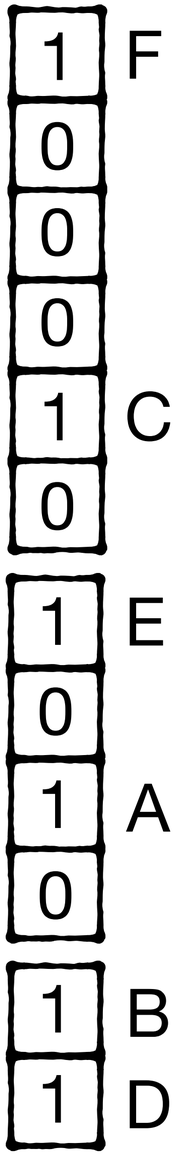
Concatenate bitmaps



Bijection is now established :)

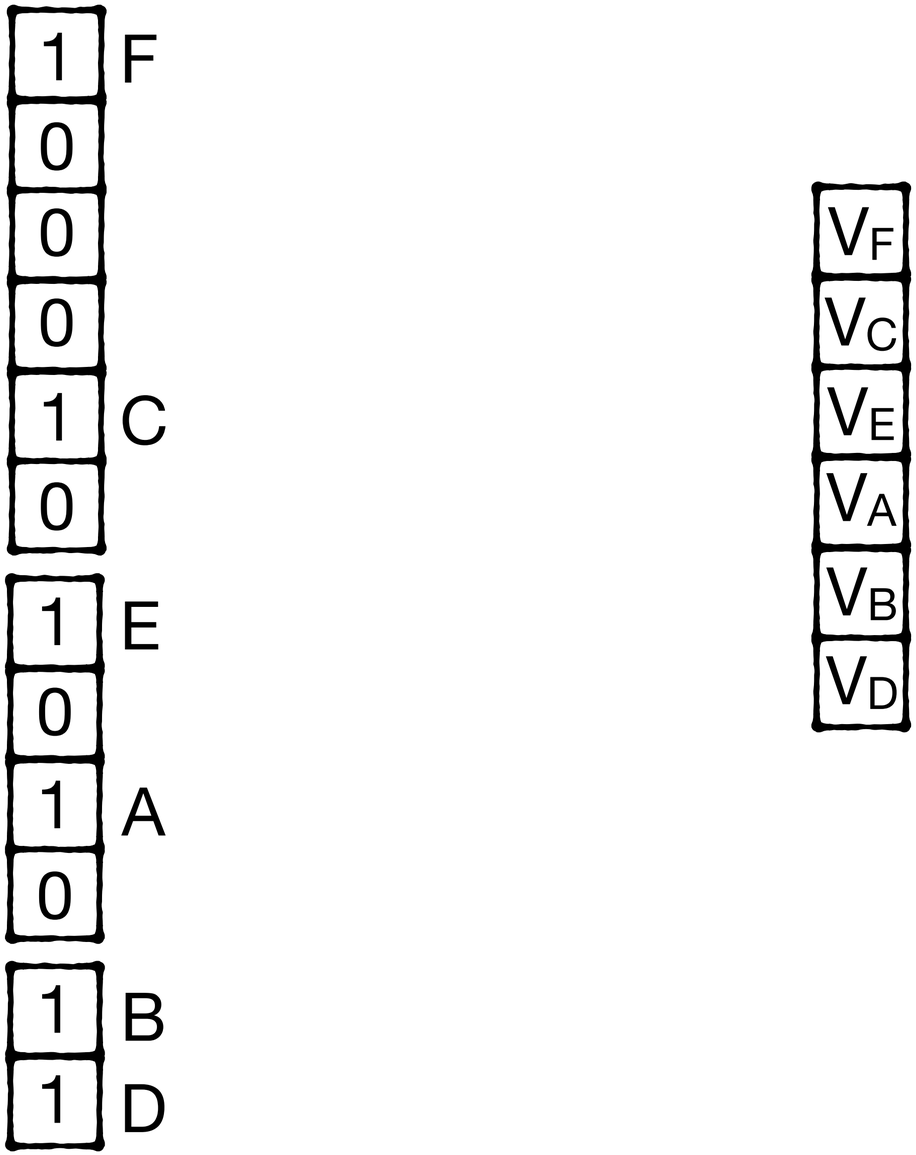


Bijection is now established :)

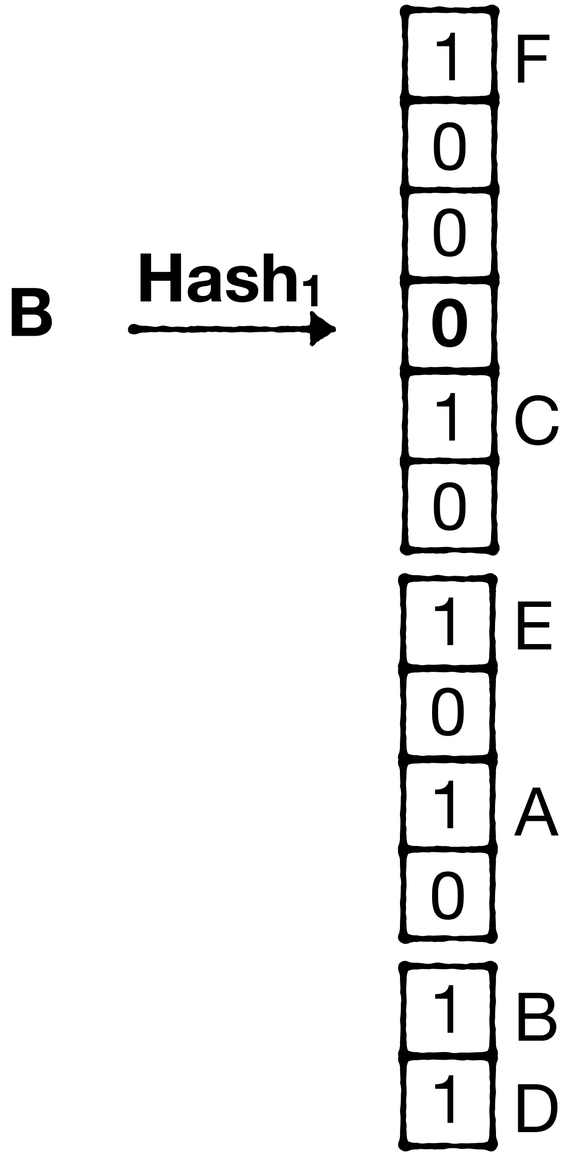


Array stores a value/pointer associated with each key

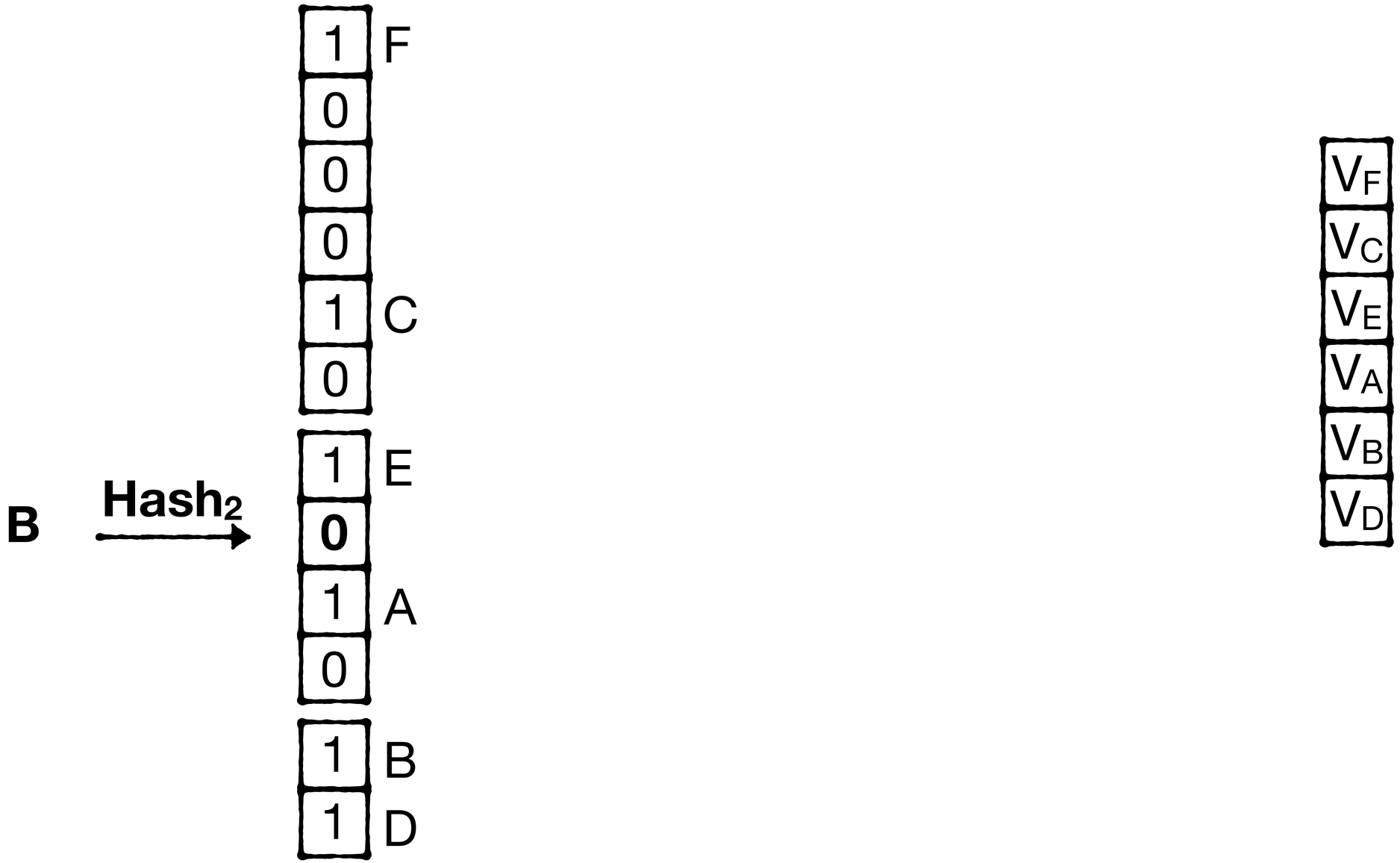
How to query?



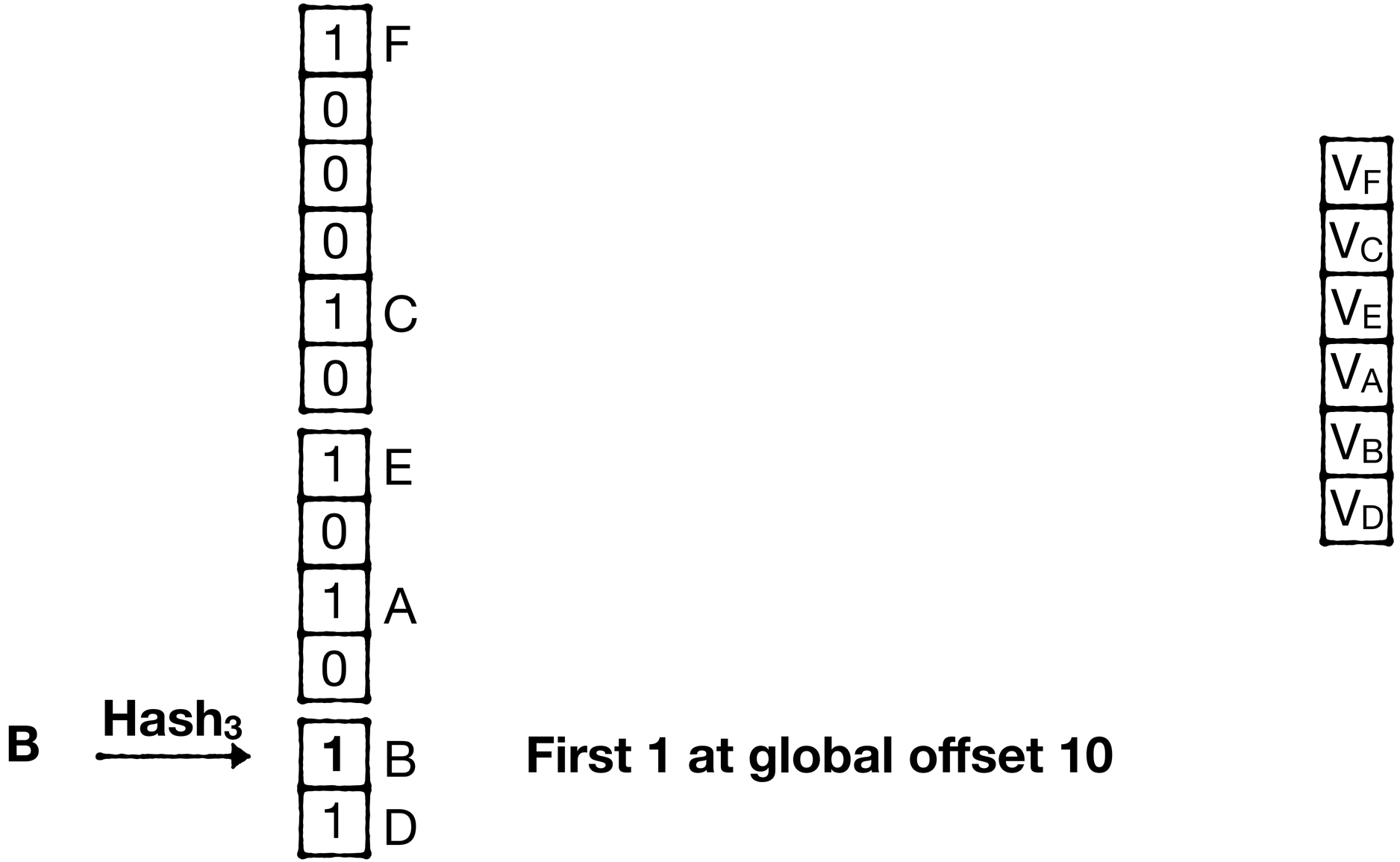
How to query? **Check bitmaps until finding 1**



How to query? **Check bitmaps until finding 1**

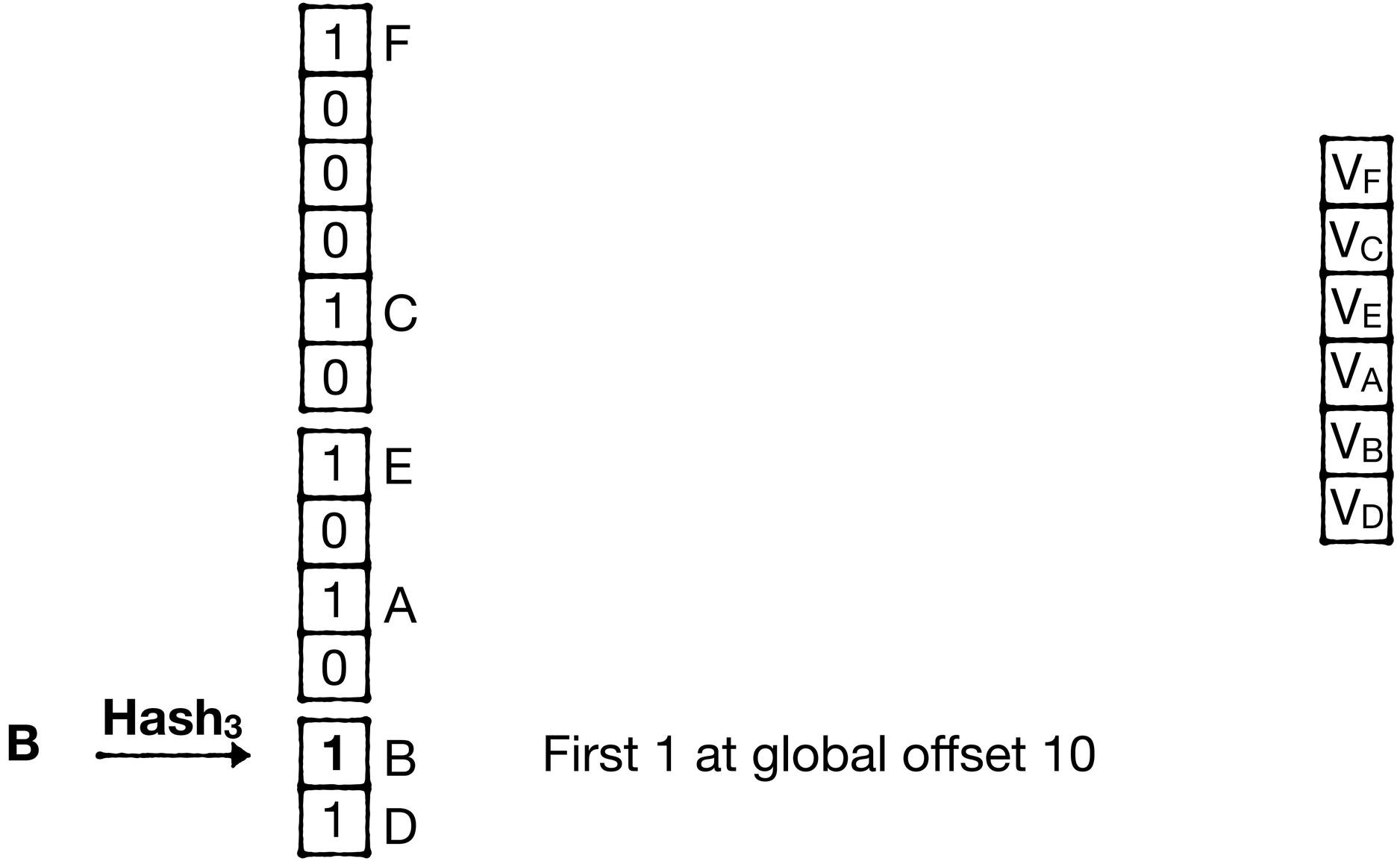


How to query? **Check bitmaps until finding 1**



How to query? Check bitmaps until finding 1

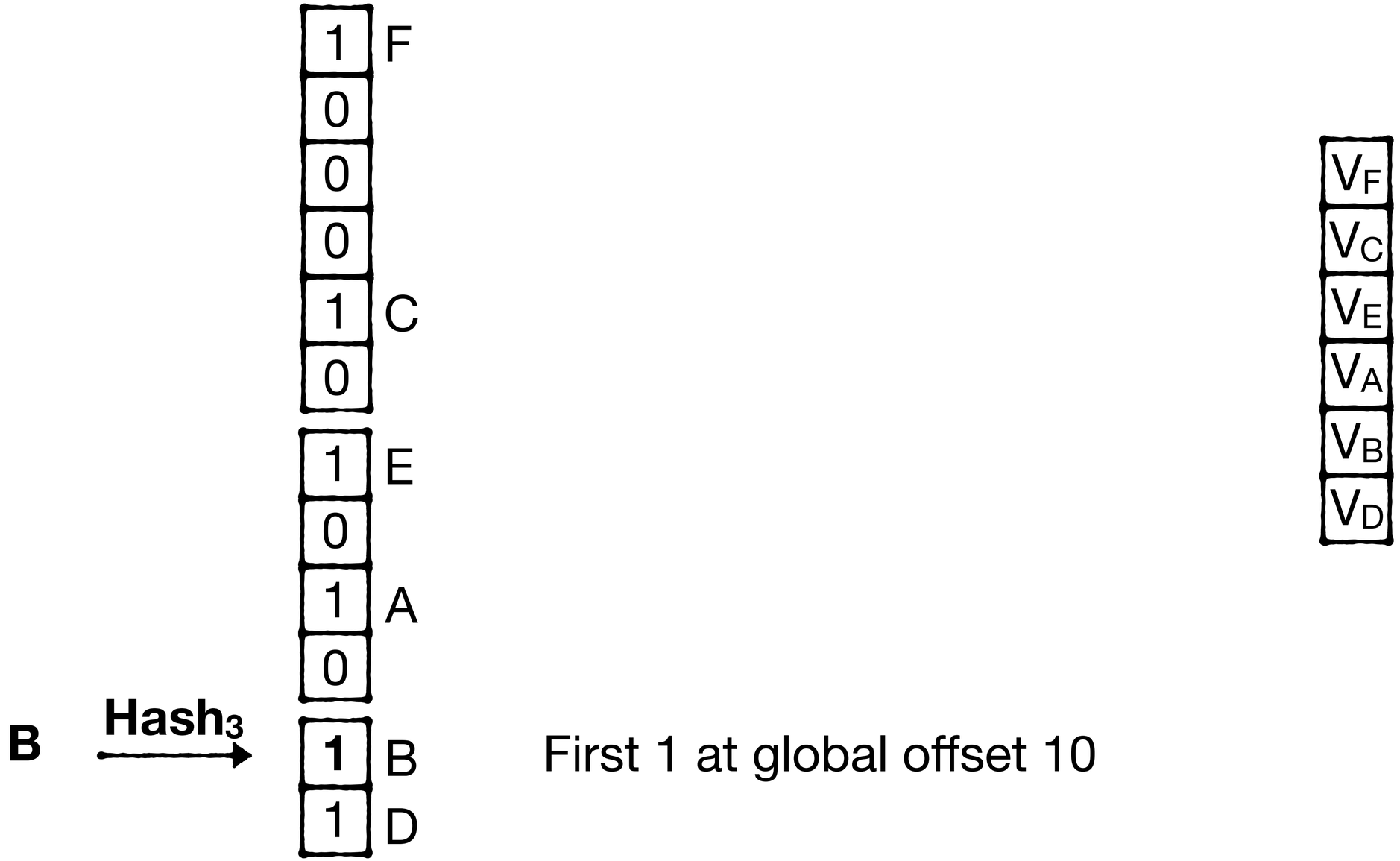
Next? :)



First 1 at global offset 10

How to query? Check bitmaps until finding 1

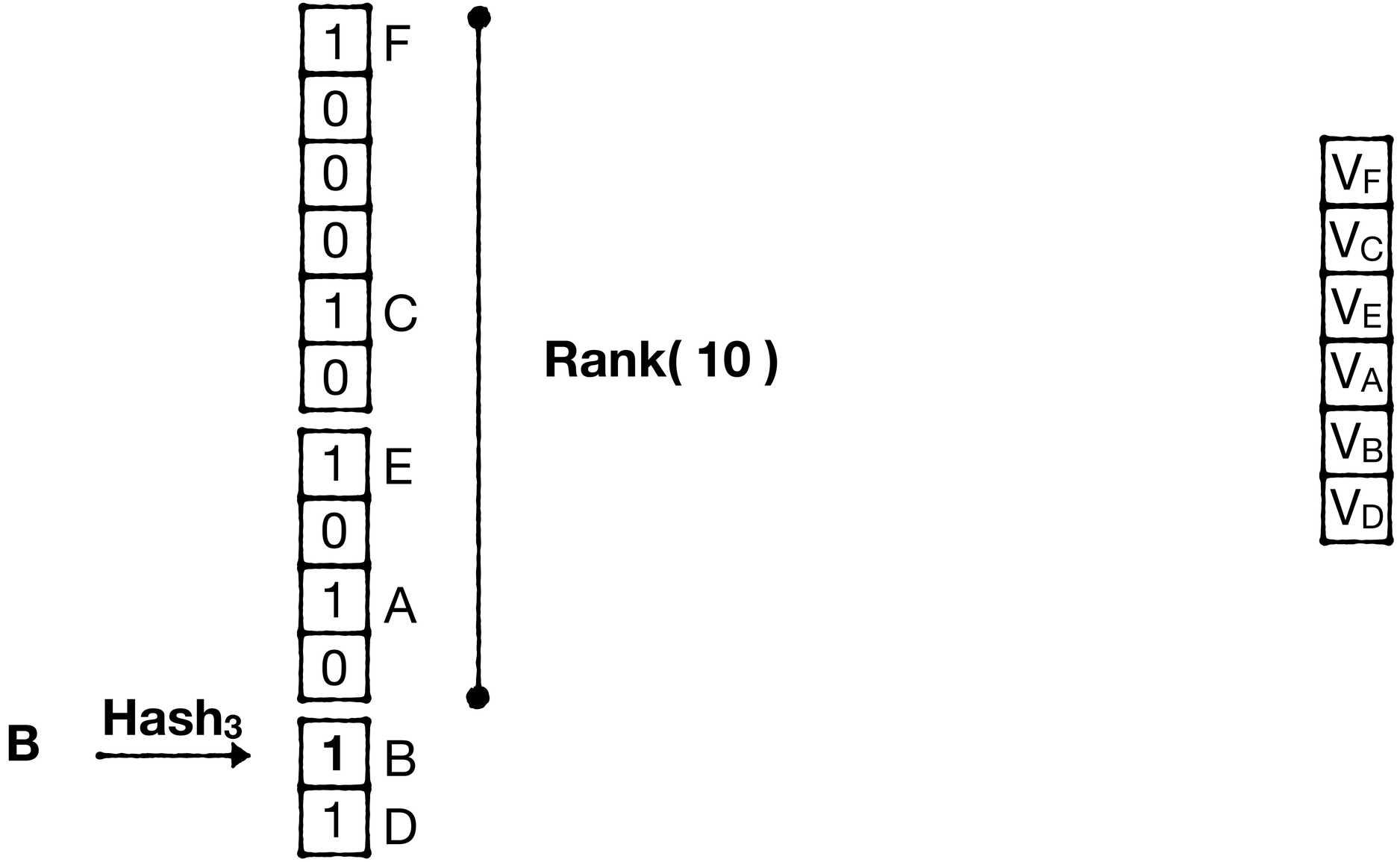
Rank on offset of first 1



First 1 at global offset 10

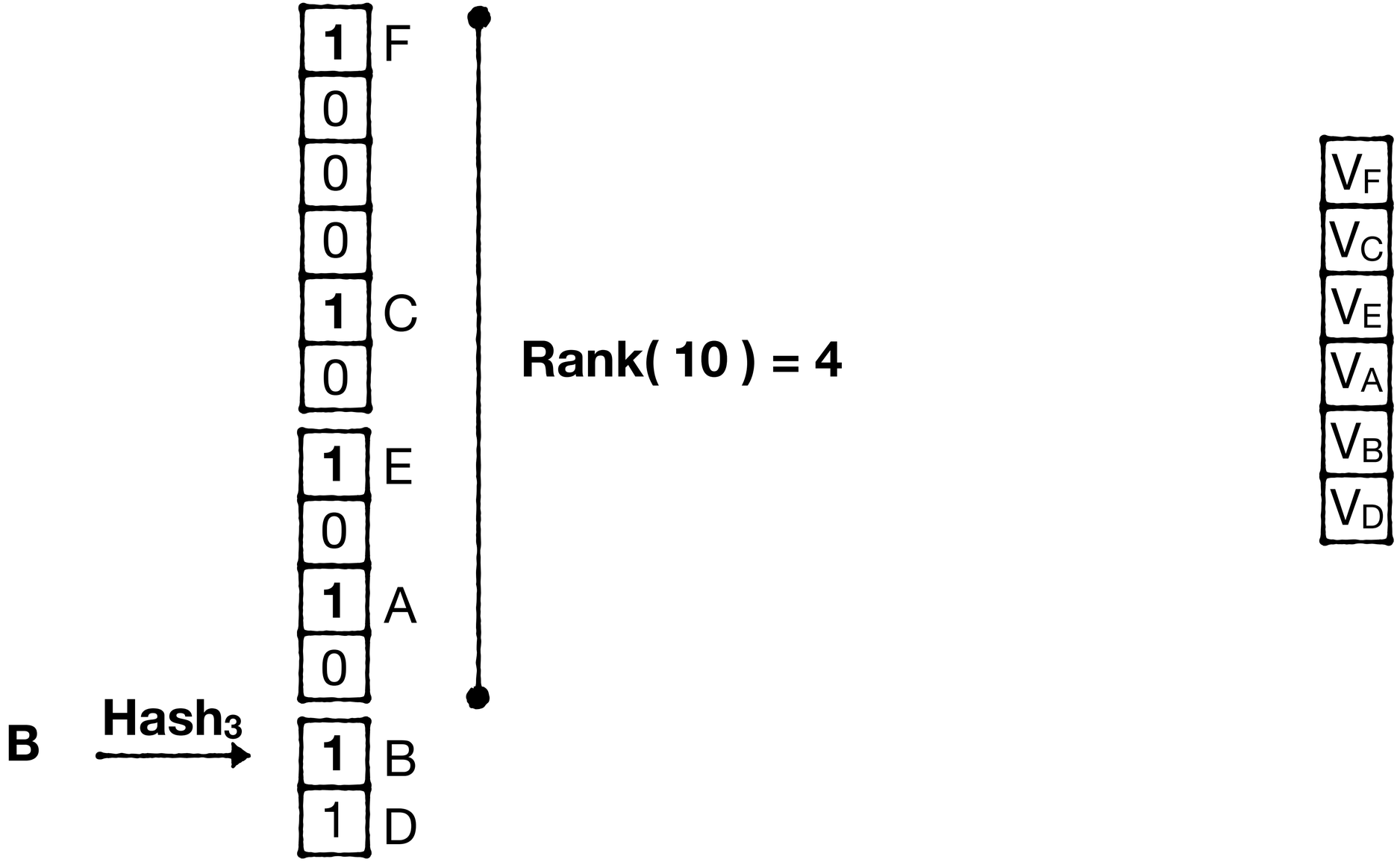
How to query? Check bitmaps until finding 1

Rank on offset of first 1



How to query? Check bitmaps until finding 1

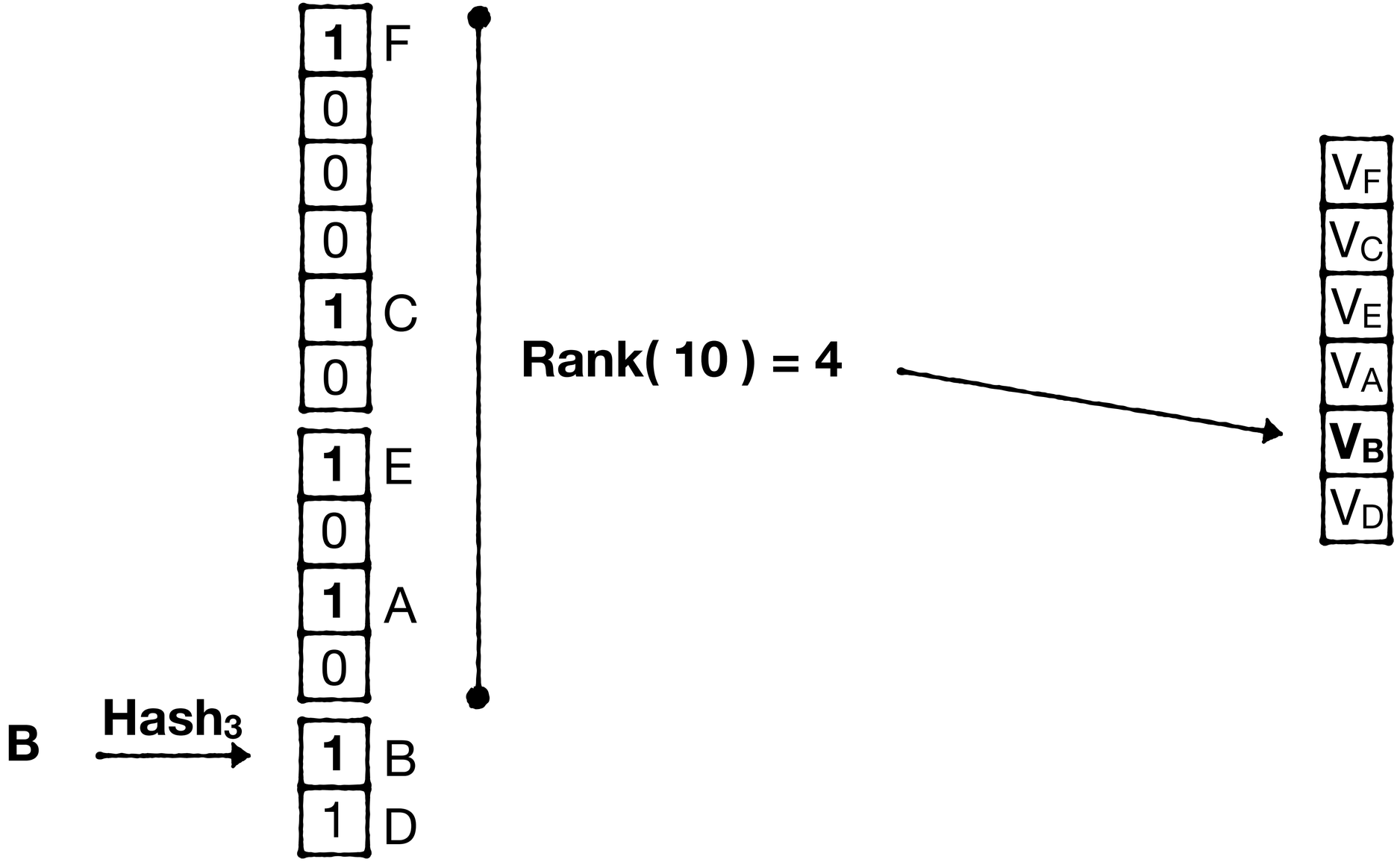
Rank on offset of first 1



How to query?

Check bitmaps until finding 1

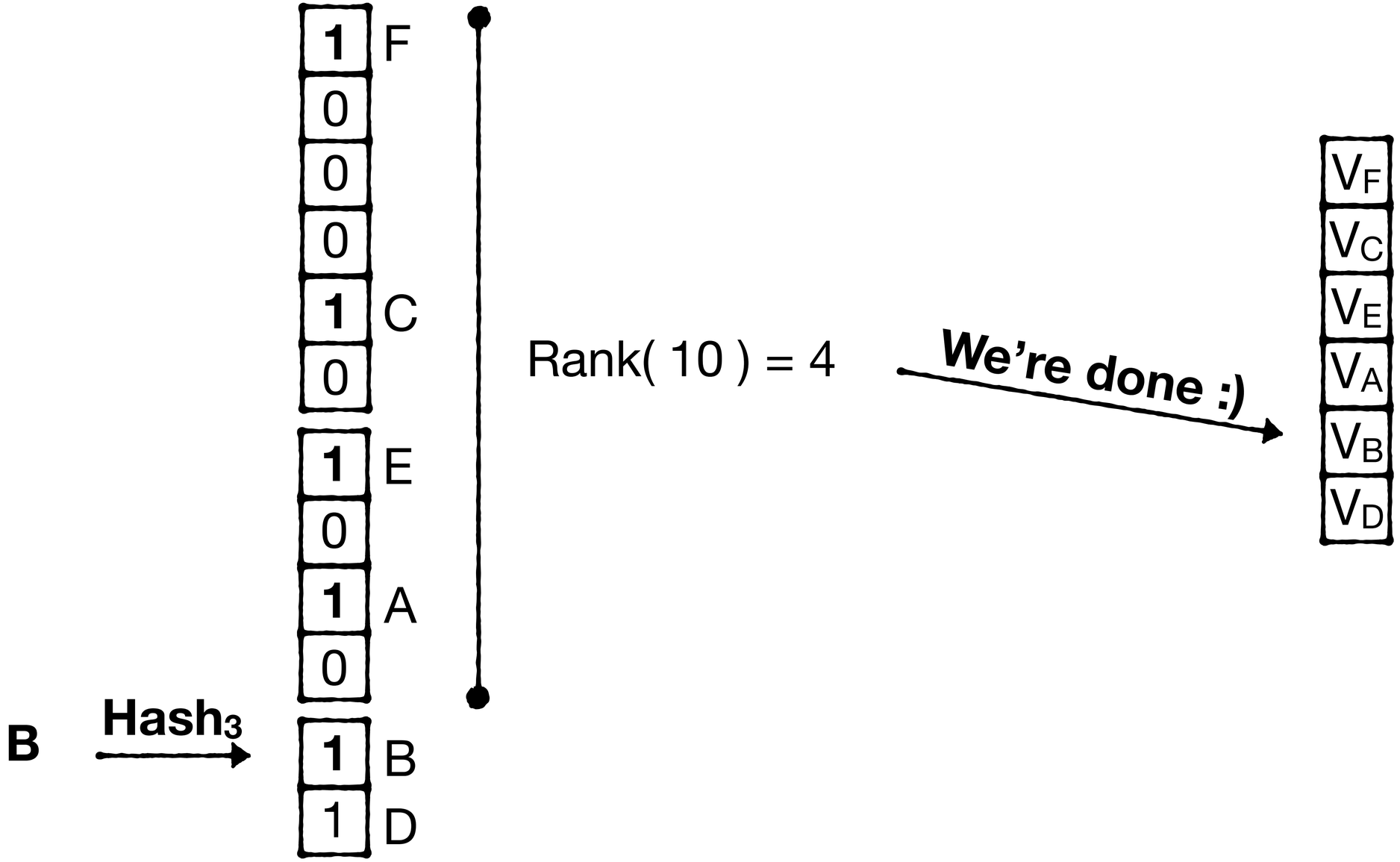
Rank on offset of first 1

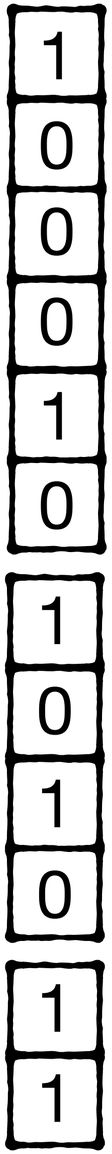


How to query?

Check bitmaps until finding 1

Rank on offset of first 1



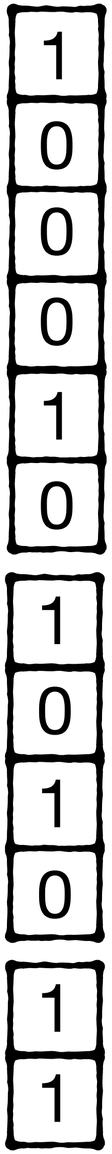


How to run rank quickly?

Rank(0) = 0

Rank(4) = 1

Rank(8) = 3

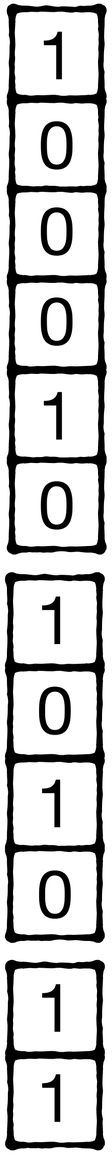


How to run rank quickly?

Rank array as we saw 2 weeks ago

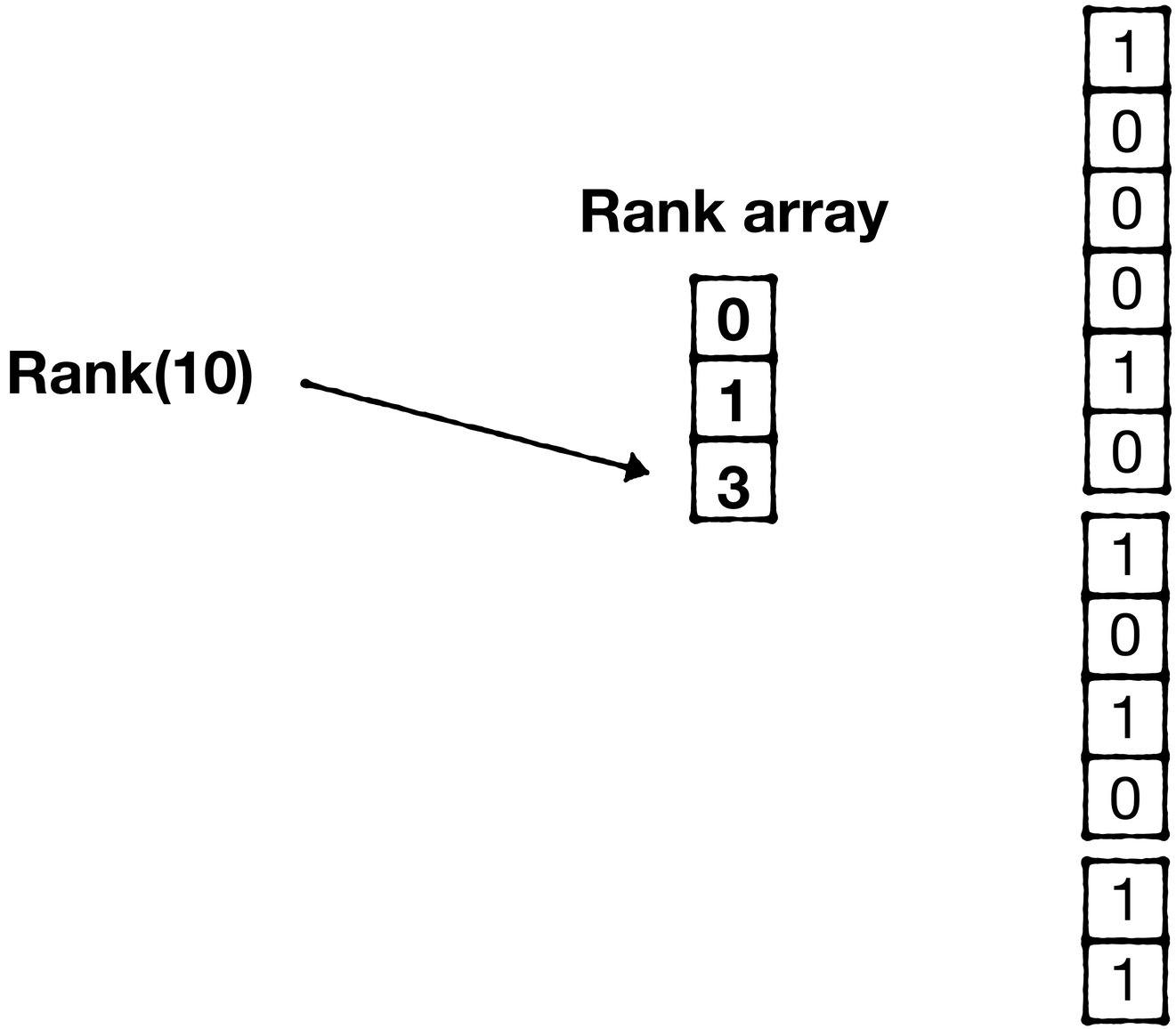
Rank array

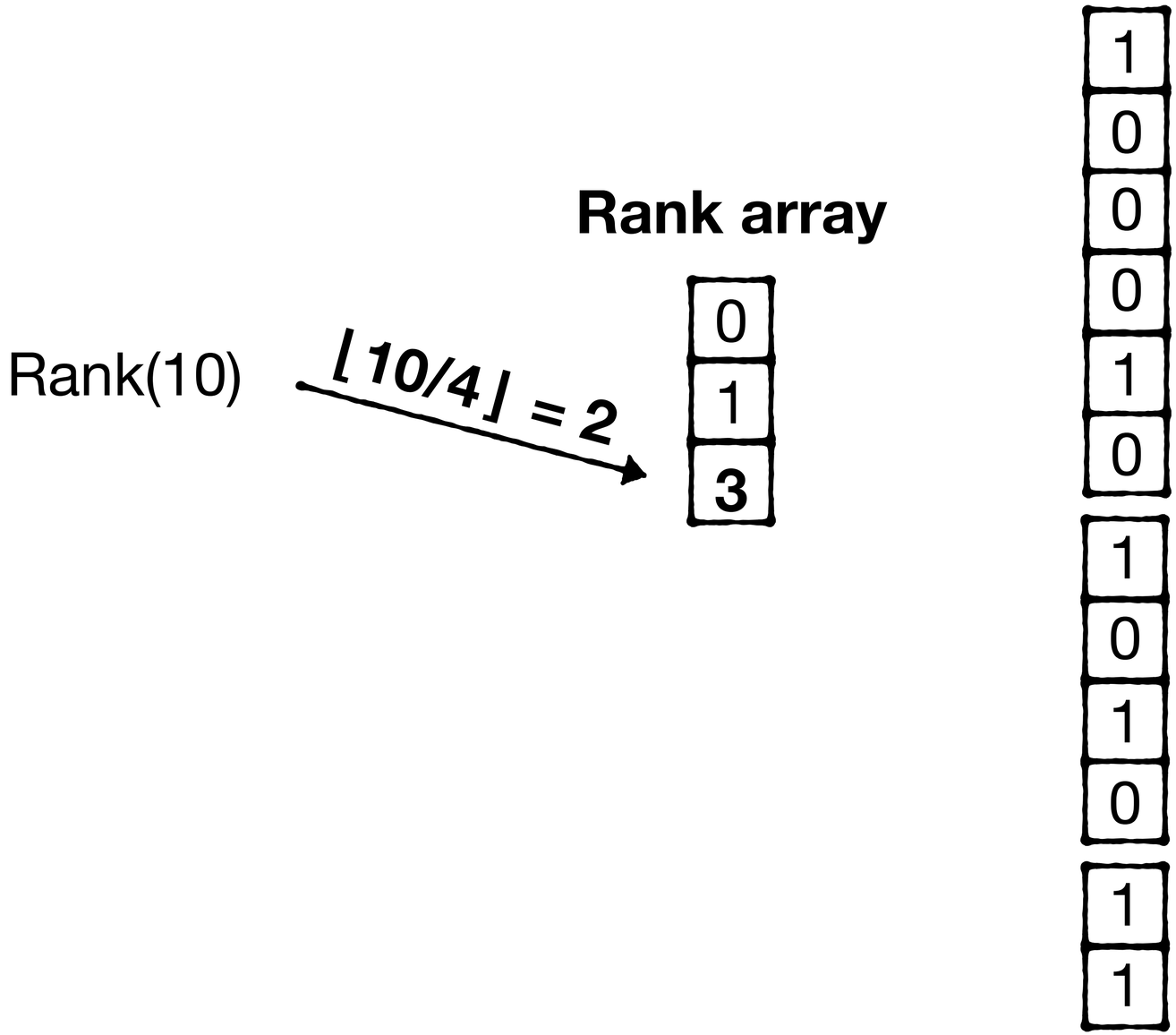
0
1
3

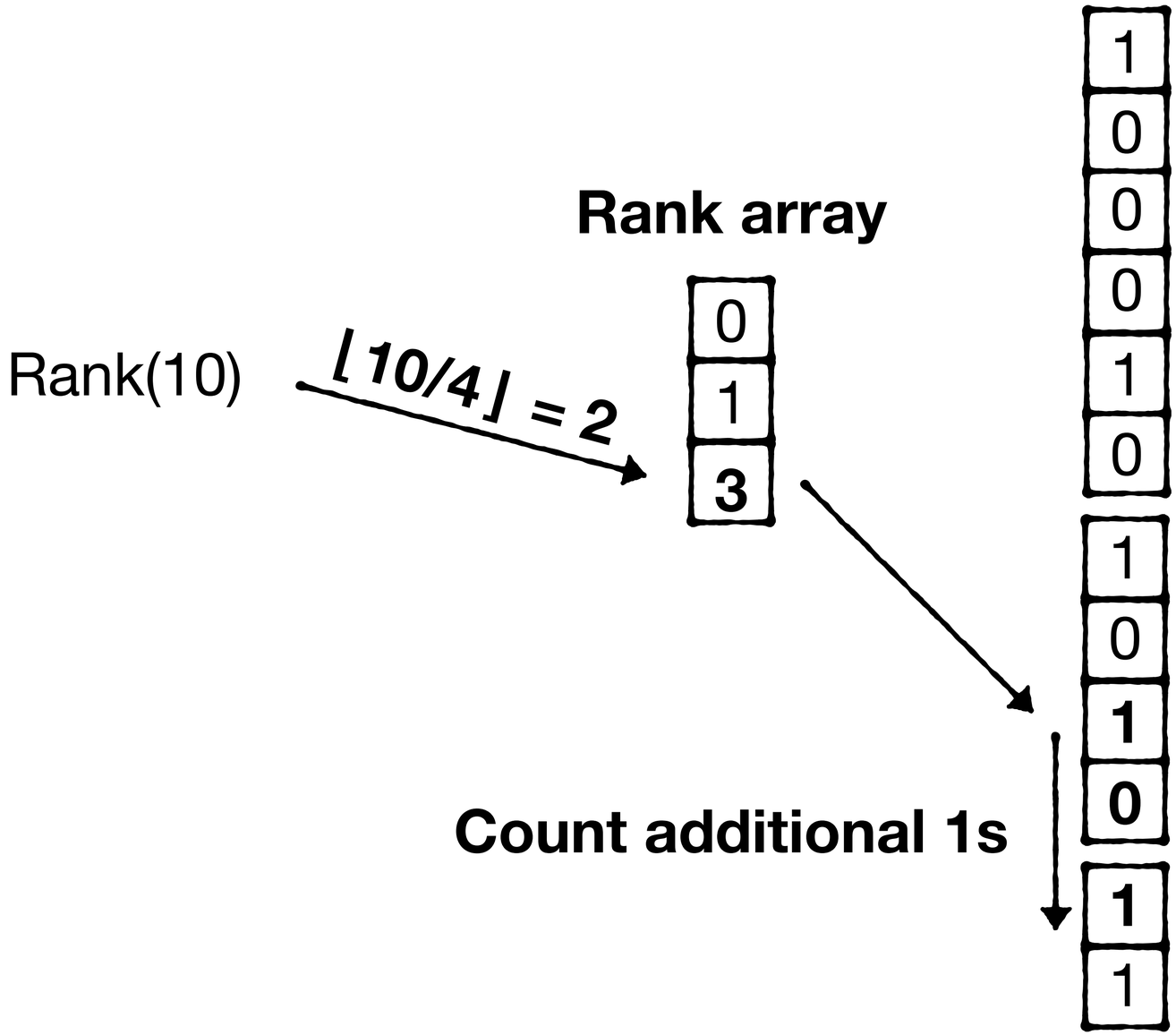


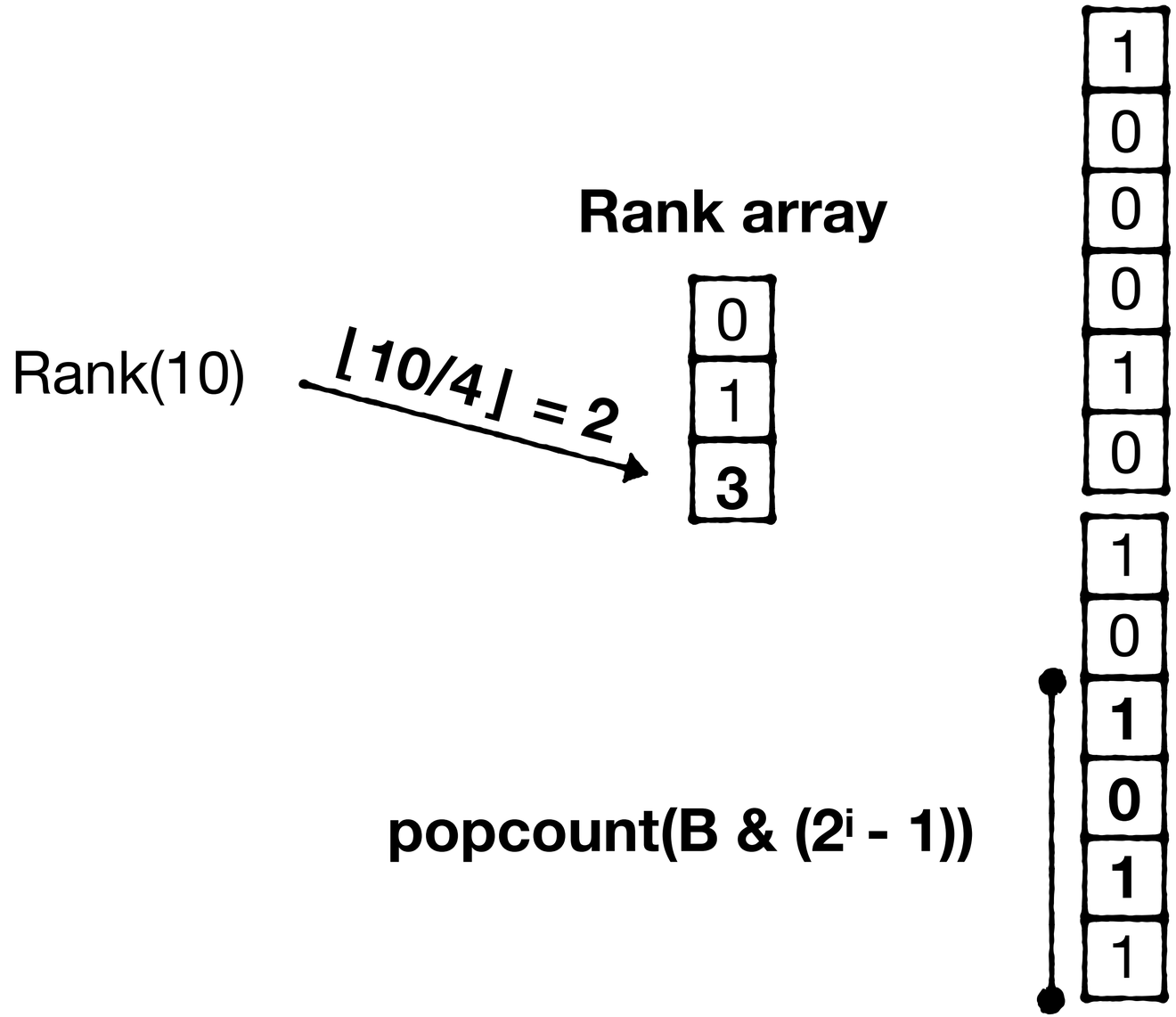
How to run rank quickly?

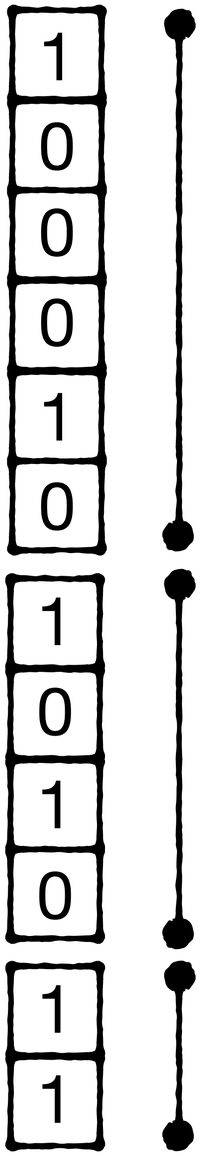
Rank array as we saw 2 weeks ago


Rank array
Rank(10)
0
1
3
1
0
0
0
1
0
1
0
1
0
1
1


Rank array
Rank(10)
⌊10/4⌋ = 2
0
1
3
1
0
0
0
1
0
1
0
1
0
1
1

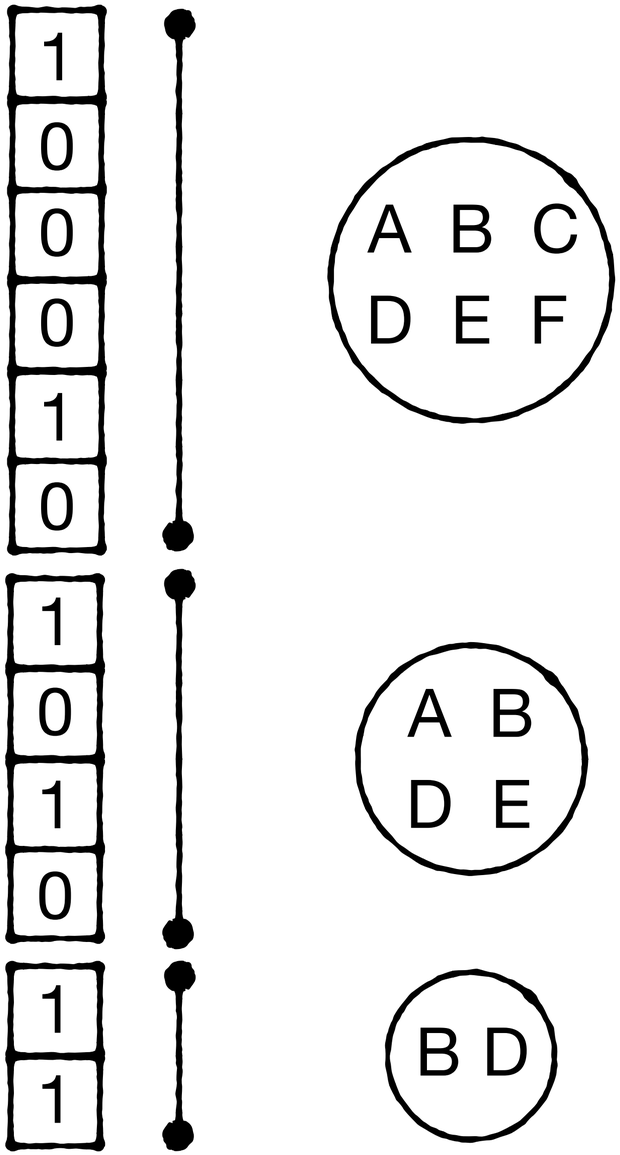

Rank array
Rank(10)
⌊10/4⌋ = 2
0
1
3
1
0
0
0
1
0
1
0
1
0
Count additional 1s
1
1


Rank array
Rank(10)
⌊10/4⌋ = 2
0
1
3
1
0
0
0
1
0
1
0
1
0
1
1
popcount(B & (2^i - 1))



Total bitmap size?

Total bitmap size?



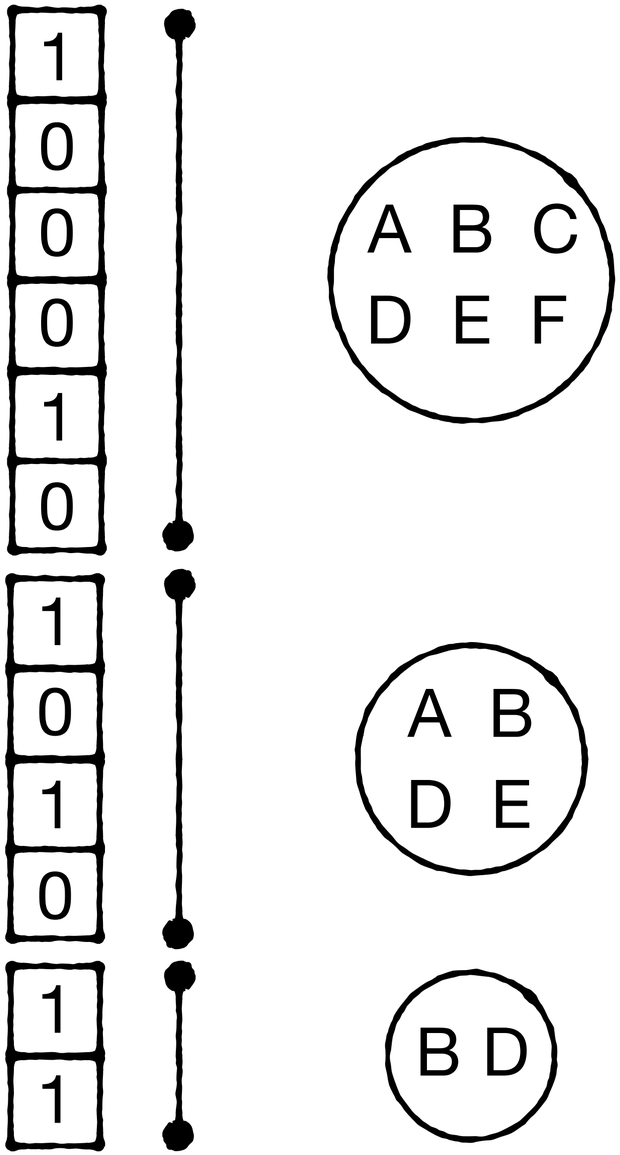
Tried 6 entries. 2 succeeded

Tried 4 entries. 2 succeeded

Tried 2 entries. 2 succeeded

Total bitmap size?

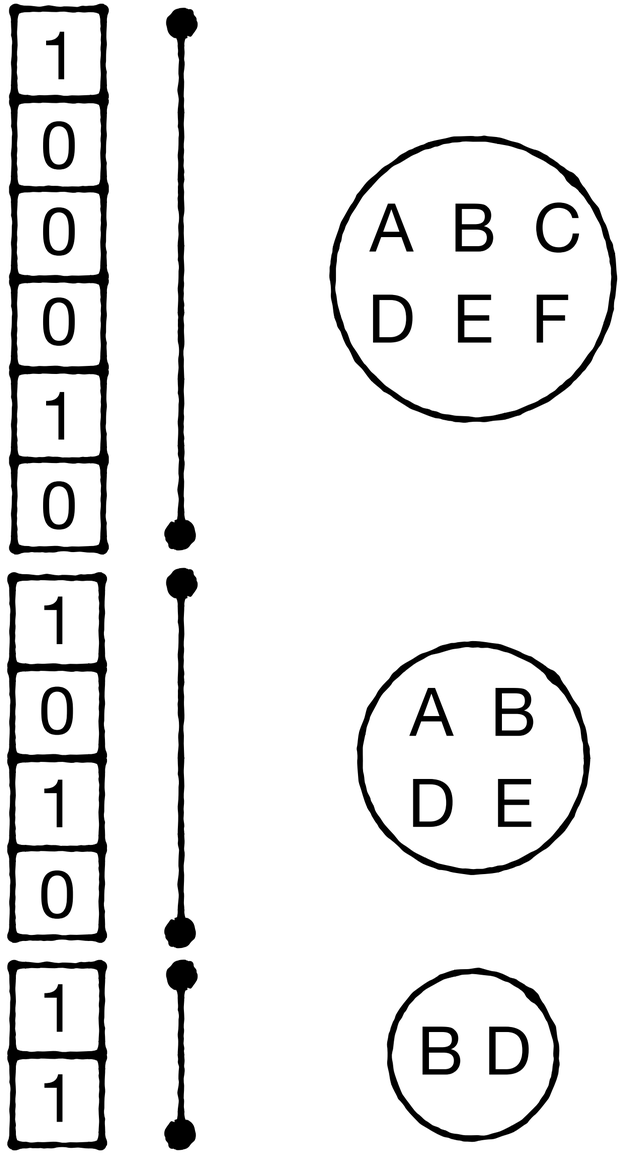
 # trials across all entries until success



Tried 6 entries. 2 succeeded

Tried 4 entries. 2 succeeded

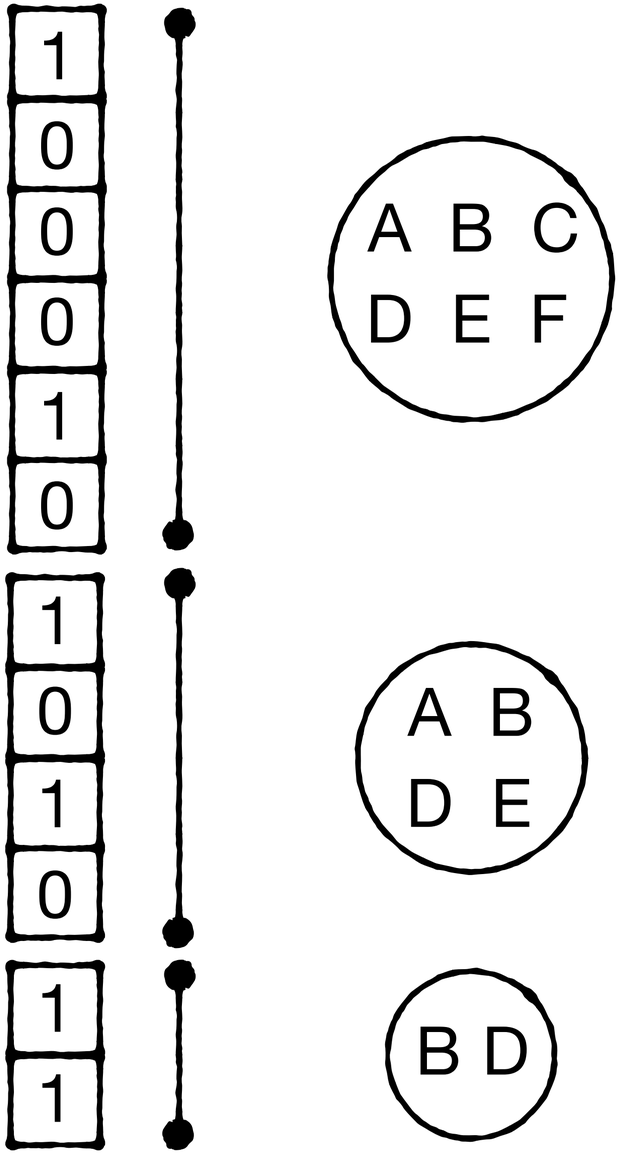
Tried 2 entries. 2 succeeded



Total bitmap size?

trials across all entries until success

P[entry succeeds in given iteration]?



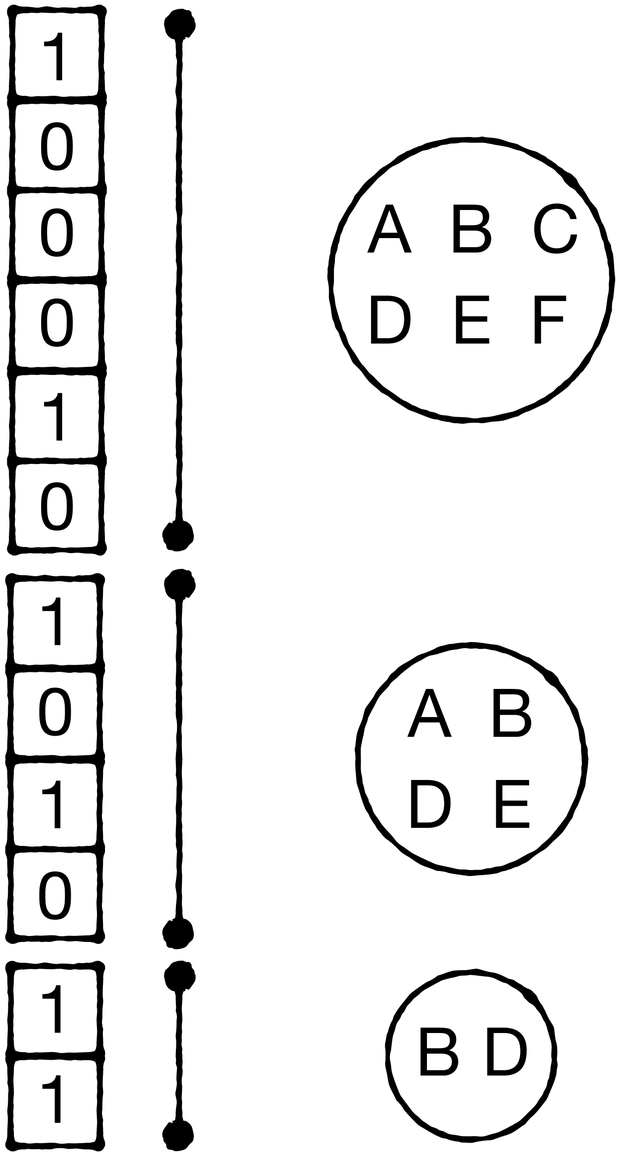
Total bitmap size?

trials across all entries until success

P[entry succeeds in given iteration]

Poisson(λ, 1)

λ= 1

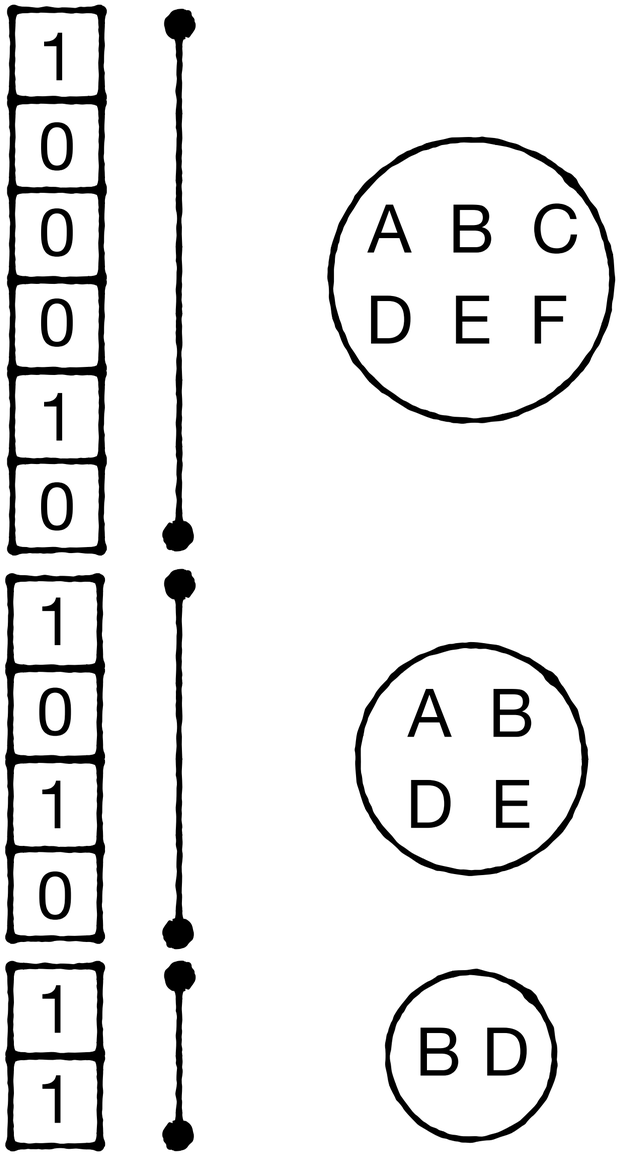


Total bitmap size?

trials across all entries until success

P[entry succeeds in given iteration]

Poisson(1,1) = e^{-1}



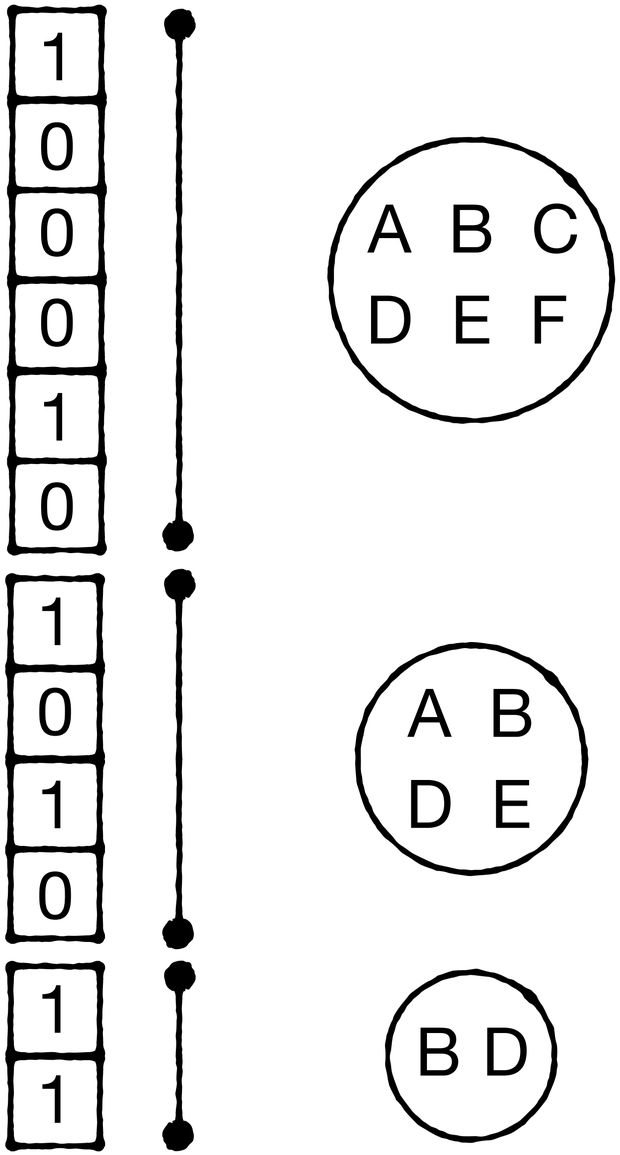
Total bitmap size?

trials across all entries until success

P[entry succeeds in given iteration]

Poisson(1,1) = e^{-1}

E[# iterations until succeeding]?

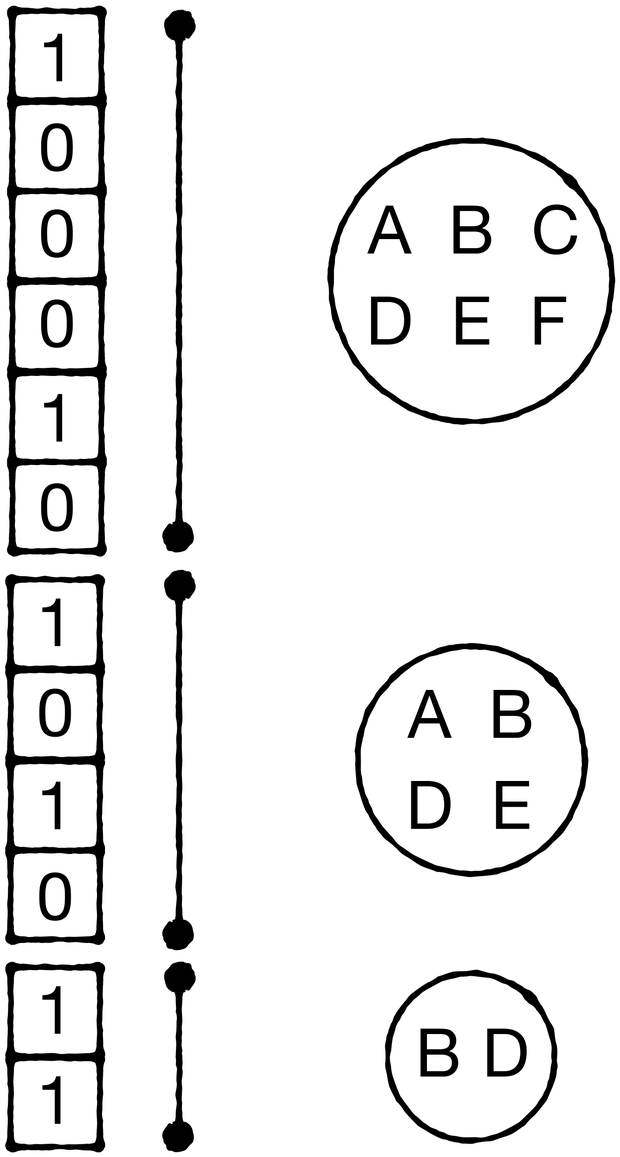


Total bitmap size?

trials across all entries until success

E[# iterations until succeeding]?

$$\frac{1}{e^{-1}}$$

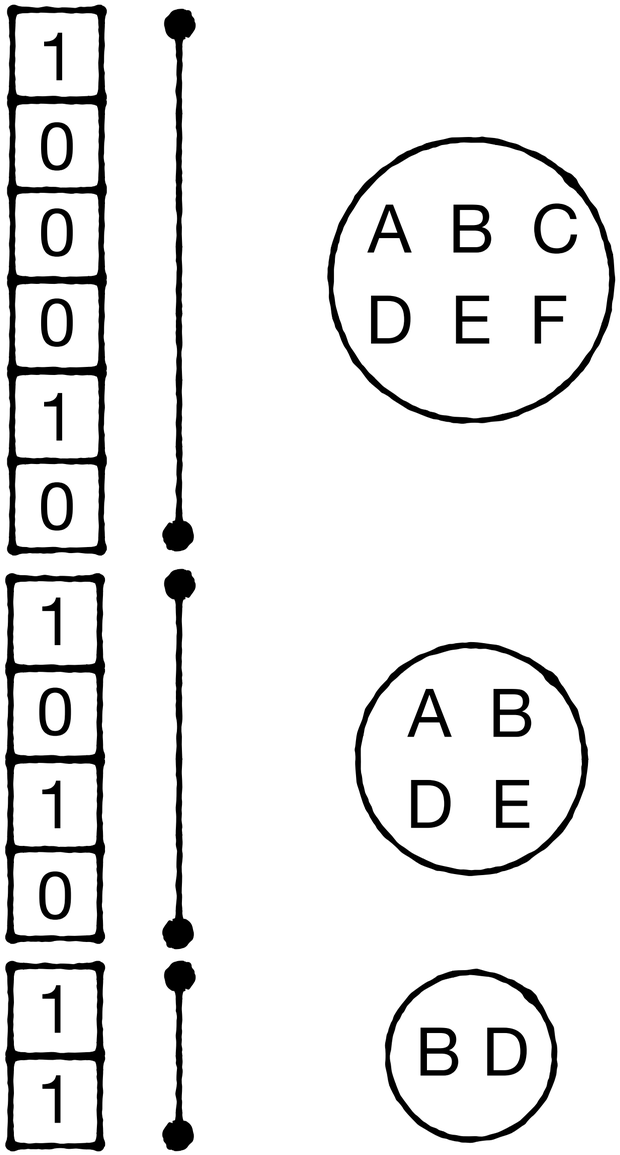


Total bitmap size?

trials across all entries until success

E[# iterations until succeeding]?

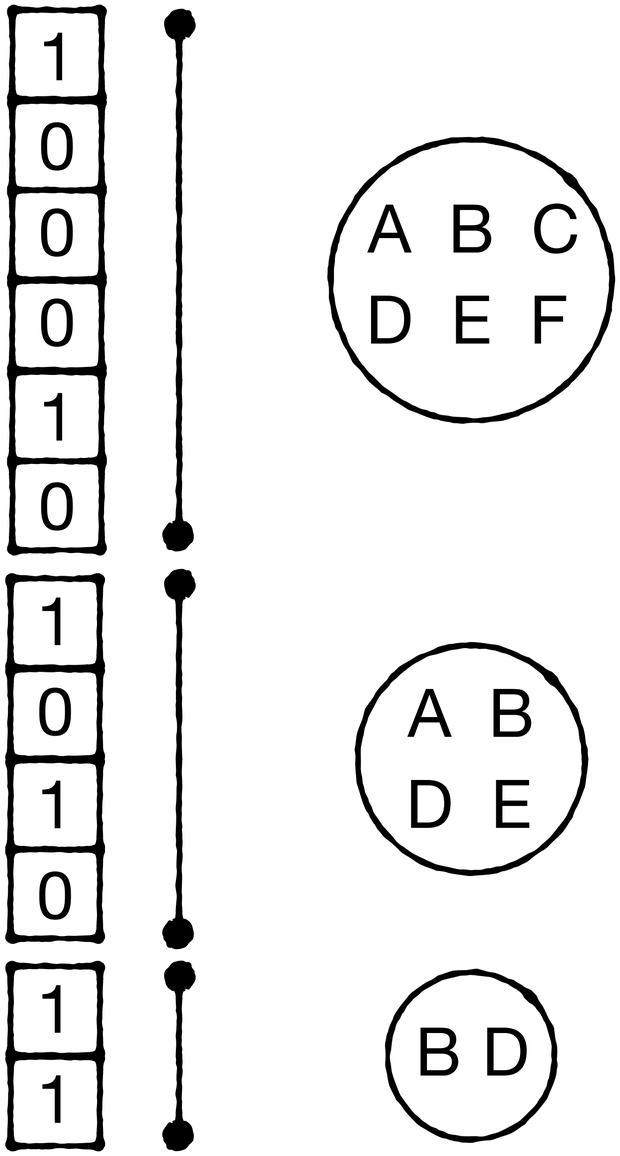
e



Total bitmap size?

 # trials across all entries until success

$$N \cdot e \quad \text{bits}$$



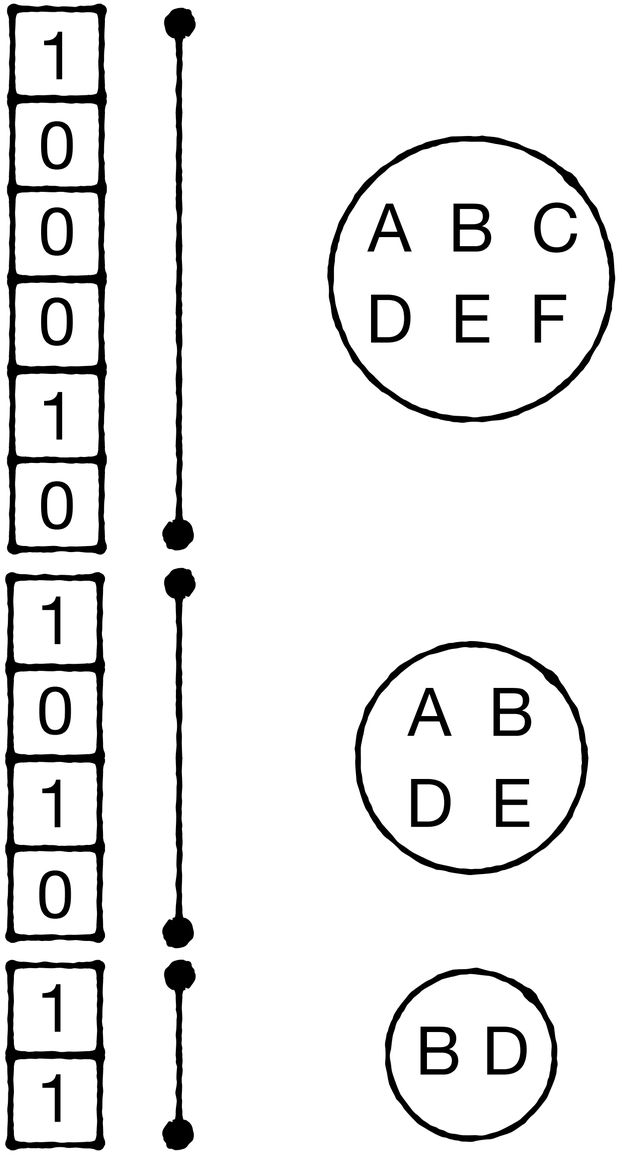
Total bitmap size?

trials across all entries until success

$$\mathbf{N \cdot e}$$

↑

Also our construction time :)



Total bitmap size?

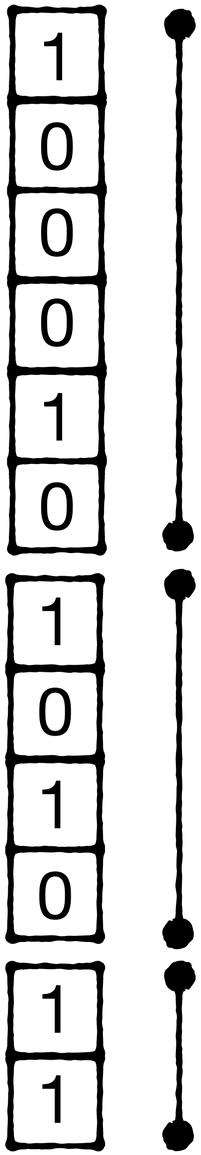
trials across all entries until success

$$\mathbf{N \cdot e}$$

Also our construction time :)

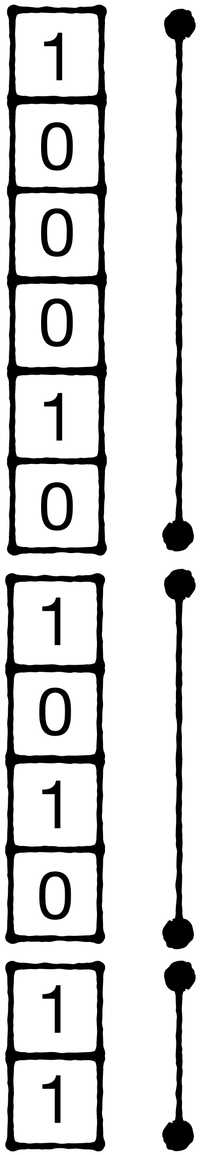
 # times each entry is hashed

Worst Case Query cost?

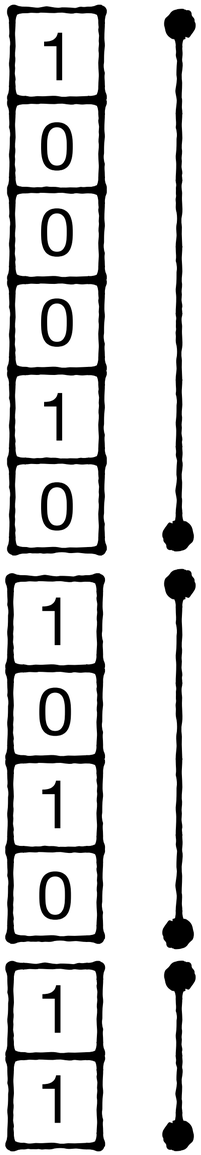


Worst Case Query cost?

= # bitmaps to traverse



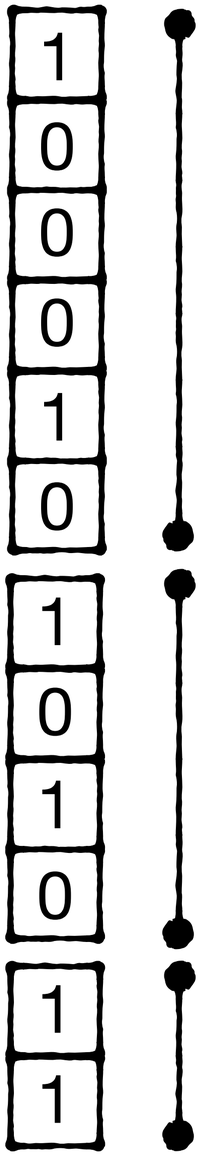
Worst Case Query cost?



= # bitmaps to traverse

= log **base** (N)

Worst Case Query cost?



= # bitmaps to traverse

= log **base** (N)

↑

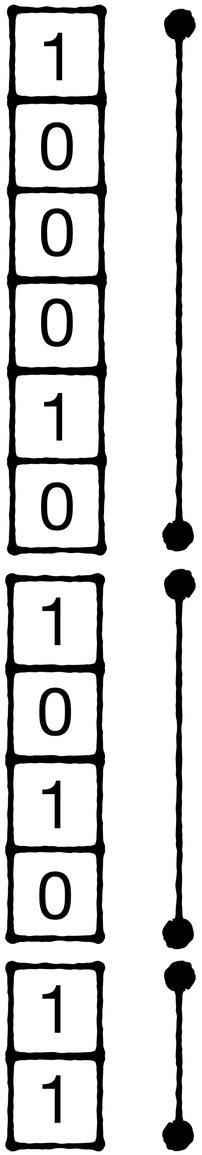
?

Worst Case Query cost?

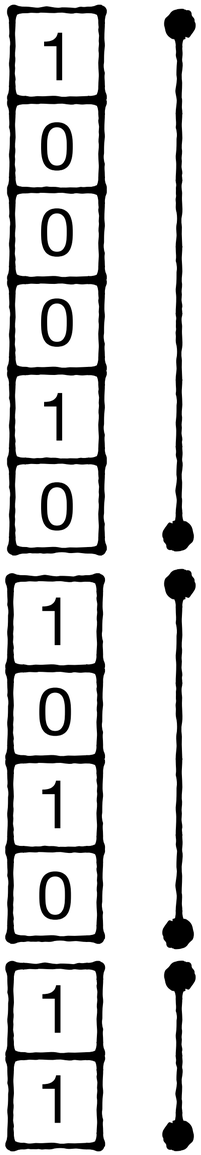
= # bitmaps to traverse

$$= \log_{\textbf{base}}(N)$$

Fraction of entries failing each iteration $\rightarrow \dfrac{1}{1 - e^{-1}}$



Worst Case Query cost?

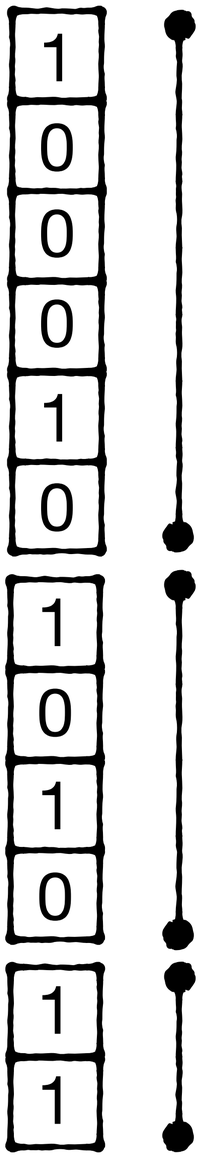


= # bitmaps to traverse

= log **base** (N)

1.58

Worst Case Query cost?

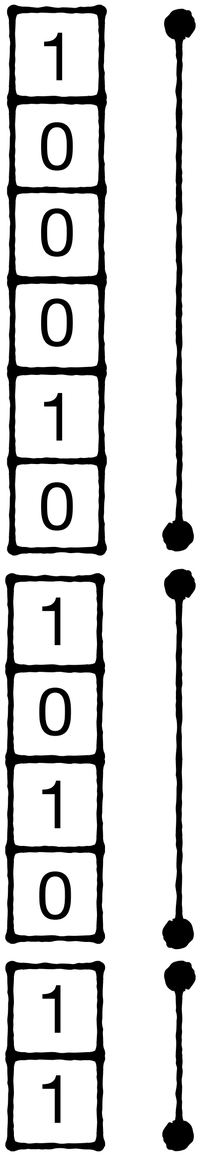


= log $_{1.58}$ (N)

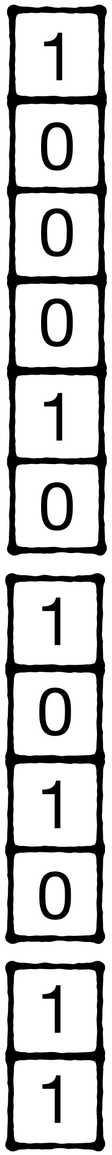
Worst Case Query cost?

$= \log_{1.58}(N)$

Is it really so bad?



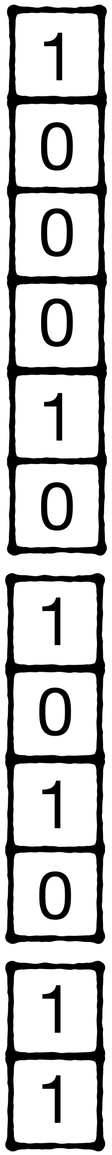
Is it really so bad?



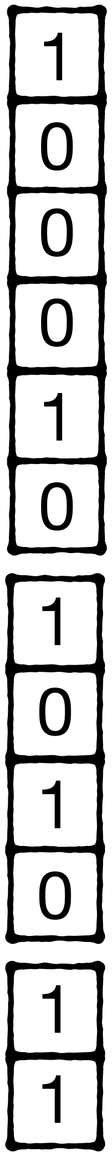
← Entries that hit 1 terminate immediately.

← **Only for very few entries we must go to end**

expected worst-case query cost?

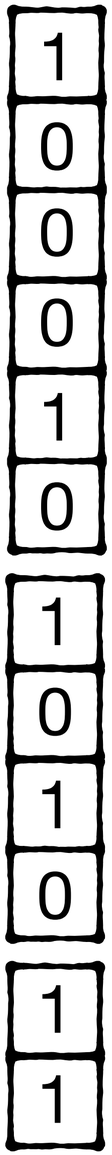


expected worst-case query cost?



1 access to first bit map

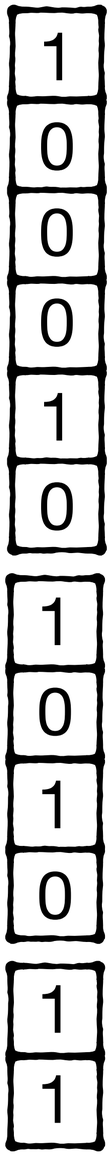
expected worst-case query cost?



1 access to first bit map

(1 - e^{-1}) chance to access second

expected worst-case query cost?

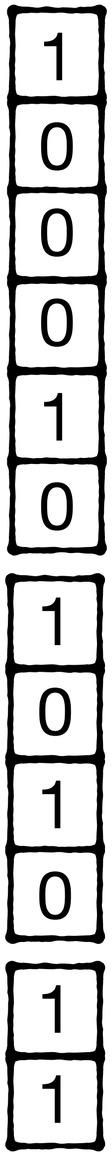


1 access to first bit map

$(1 - e^{-1})$ chance to access second

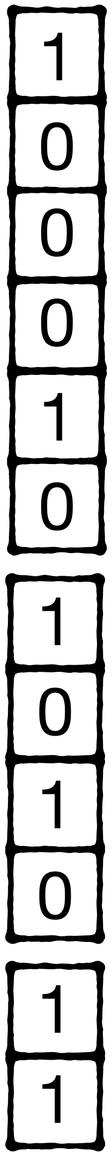
$(1 - e^{-1})^2$ chance to access third

expected worst-case query cost?



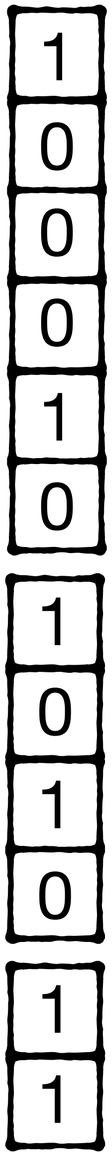
$$= 1 + (1 - e^{-1}) + (1 - e^{-1})^2 + (1 - e^{-1})^3 + \ldots$$

expected worst-case query cost?



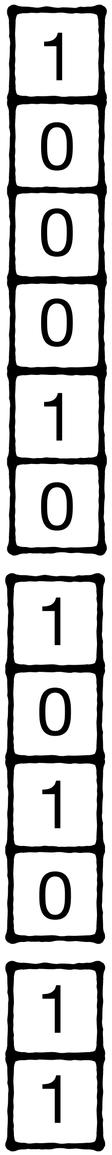
$$= 1 + (1 - e^{-1}) + (1 - e^{-1})^2 + (1 - e^{-1})^3 + \ldots$$

$$= \frac{\mathbf{1}}{\mathbf{1 - e^{-1}}} = \mathbf{1.58} = \mathbf{O(1)}$$



Memory (bits)	**N · e**
worst-case query cost	**O(log N)**
expected worst-case query cost	**O(1)**
Construction	**O(N)**

Is this the best we can do?

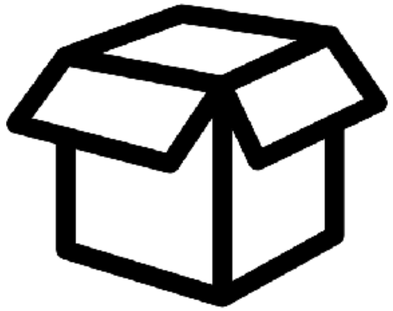


Memory (bits)	 **N · e**
worst-case query cost	O(log N)
expected worst-case query cost	O(1)
Construction	O(N)

Lower Bound for Minimal Perfect Hashing

Lower Bound for Minimal Perfect Hashing

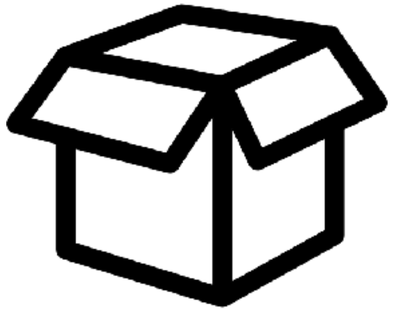
Assume nothing about implementation



Analyze with respect to specification

Lower Bound for Minimal Perfect Hashing

Assume nothing
about implementation



Analyze with respect
to specification

N - # entries
Bijective (one-to-one)

Lower Bound for Minimal Perfect Hashing (MPH)

$$|MPH| + ??? \geq |Permutation|$$

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| \; + \; \textbf{???} \; \geq \; |\text{Permutation}|$$

What data must we add to transform MPH into permutation? :)

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| \; + \; \mathbf{???} \; \geq \; |\text{Permutation}|$$

What data must we add to transform MPH into permutation? :)

The data itself! $\mathbf{N \cdot \log_2(N)}$

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| + \mathbf{N \cdot \log_2(N)} \geq |\text{Permutation}|$$

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| + N \cdot \log_2(N) \geq \textbf{|Permutation|}$$

How big is this?

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| + N \cdot \log_2(N) \geq \mathbf{\log_2(N!)}$$

How big is this?

Lower Bound for Minimal Perfect Hashing (MPH)

$$|\text{MPH}| + N \cdot \log_2(N) \geq \mathbf{N \cdot \log_2(N) + N \cdot \log_2(e)}$$

By Stirling's approximation

Lower Bound for Minimal Perfect Hashing (MPH)

$$|MPH| \geq N \log_2(e)$$

We're done :)

Lower Bound

$$N \cdot \log_2(e)$$

Fingerprinting

$$N \cdot e$$

Lower Bound	Fingerprinting
$\mathbf{N \cdot \log_2(e)}$	$\mathbf{N \cdot e}$

Not far off, but also not there…

Lower Bound	Fingerprinting
$N \cdot \log_2(e)$	**$N \cdot e$**

Not far off, but also not there…

Other methods push memory footprint lower :)

Perfect Hashing

Minimal

space-efficient
static data

Dynamic

more space
supports updates

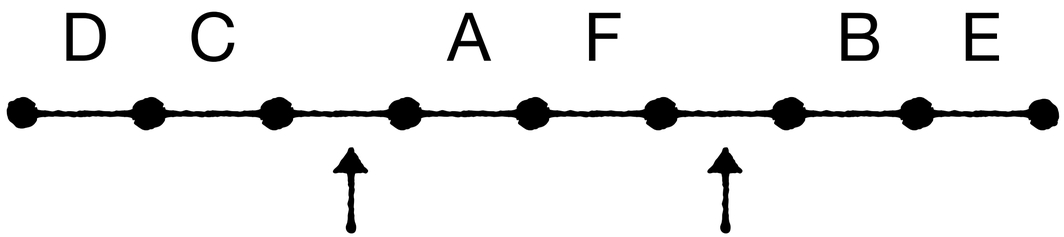
Dynamic Perfect Hashing

N keys

A B C D E F

more than N slots

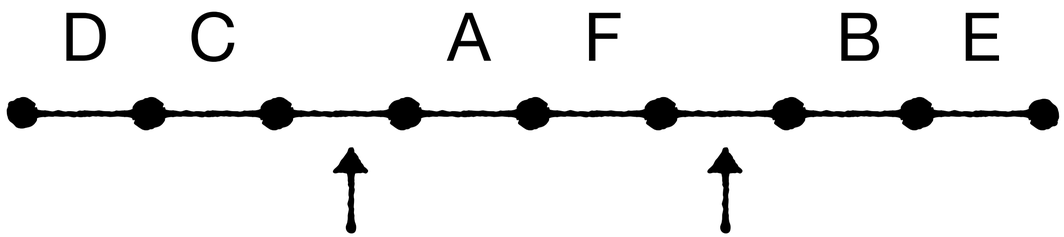
Dynamic Perfect Hashing



Some slots can stay free

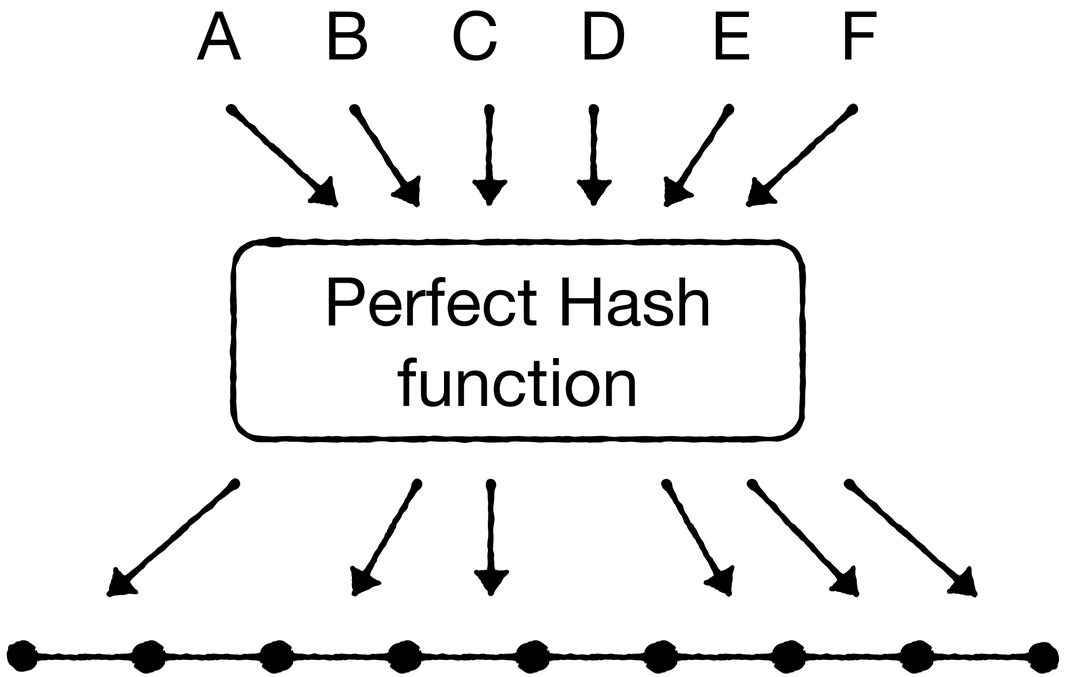
Dynamic Perfect Hashing

More flexibility :)



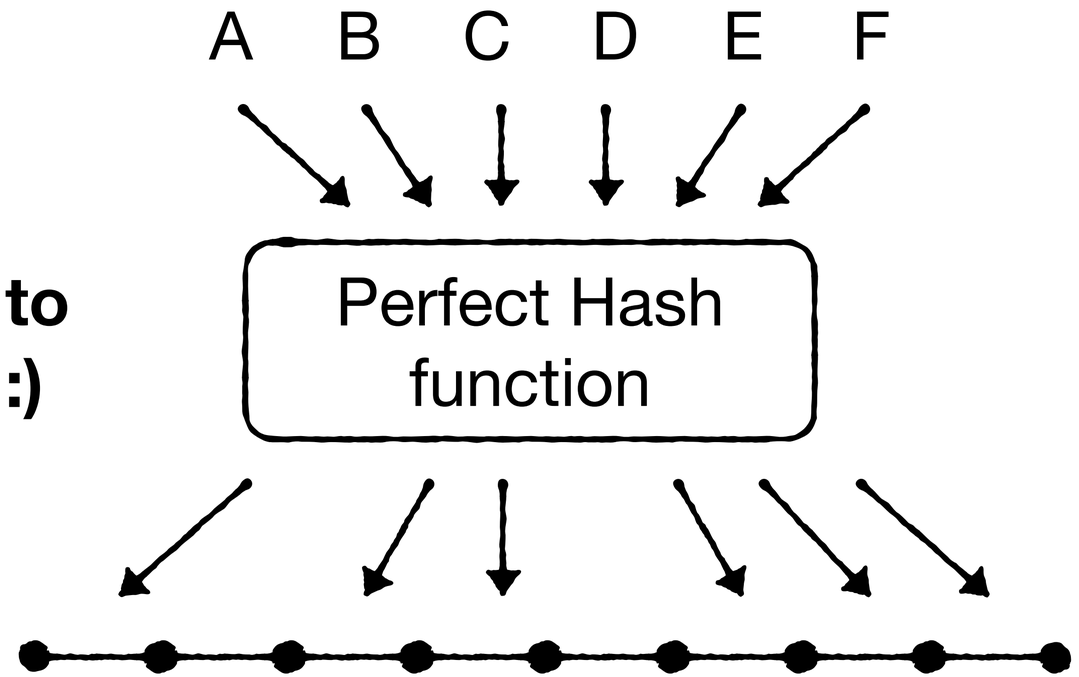
Some slots can stay free

Dynamic Perfect Hashing



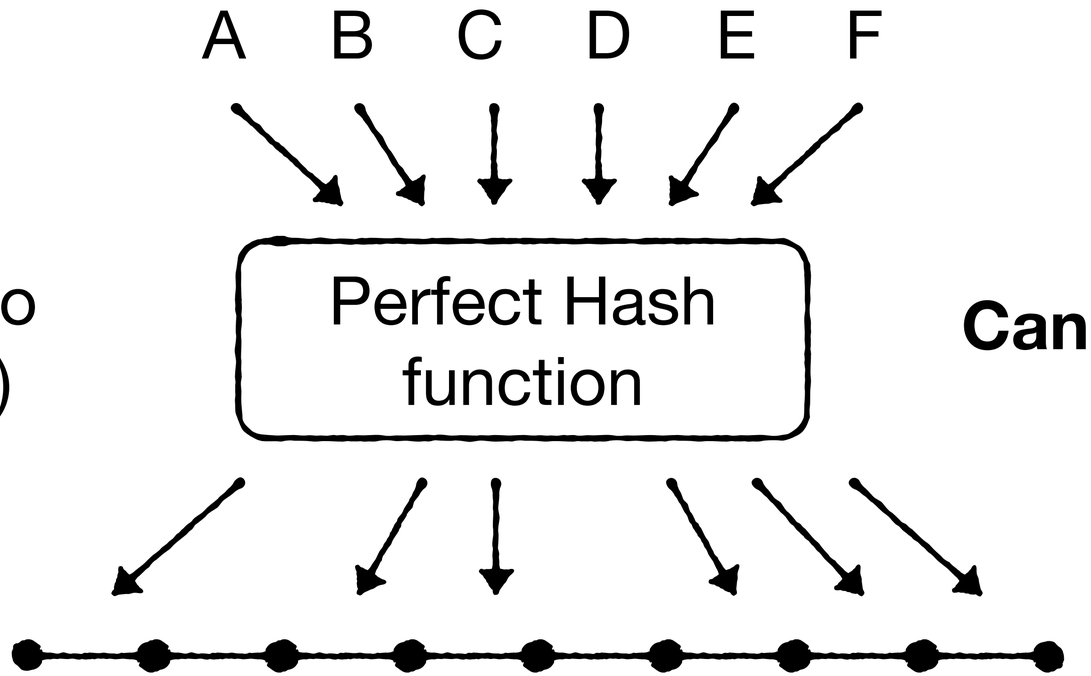
Dynamic Perfect Hashing

Just a data structure to ensure no collisions :)



Dynamic Perfect Hashing

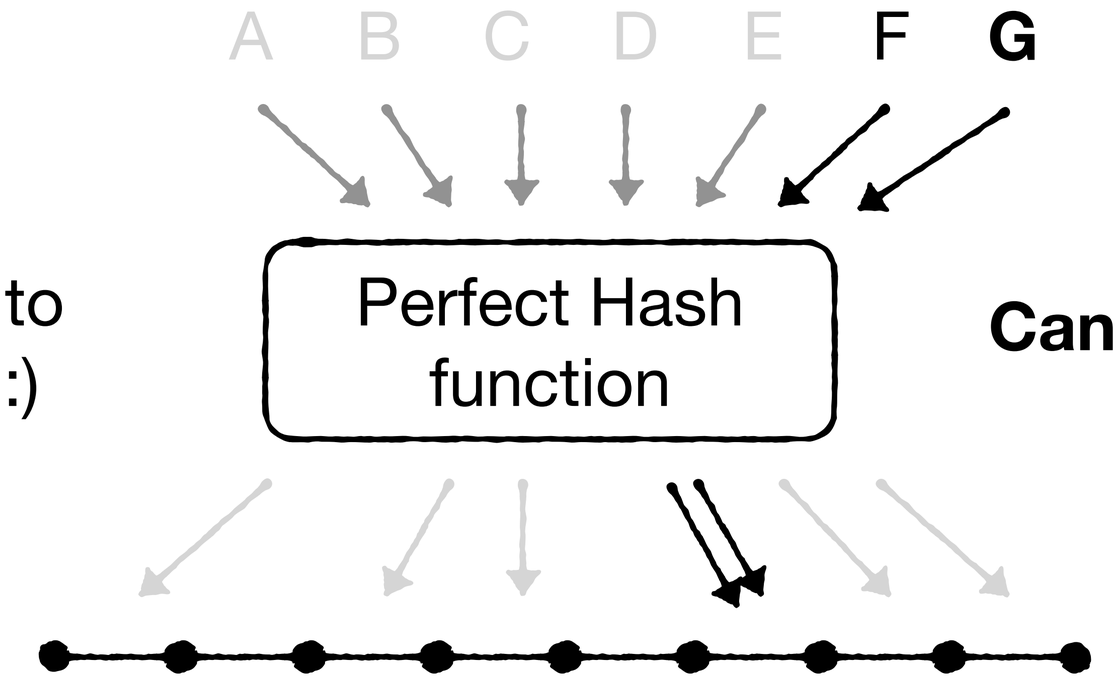
Just a data structure to ensure no collisions :)



Can handle insertions!

Dynamic Perfect Hashing

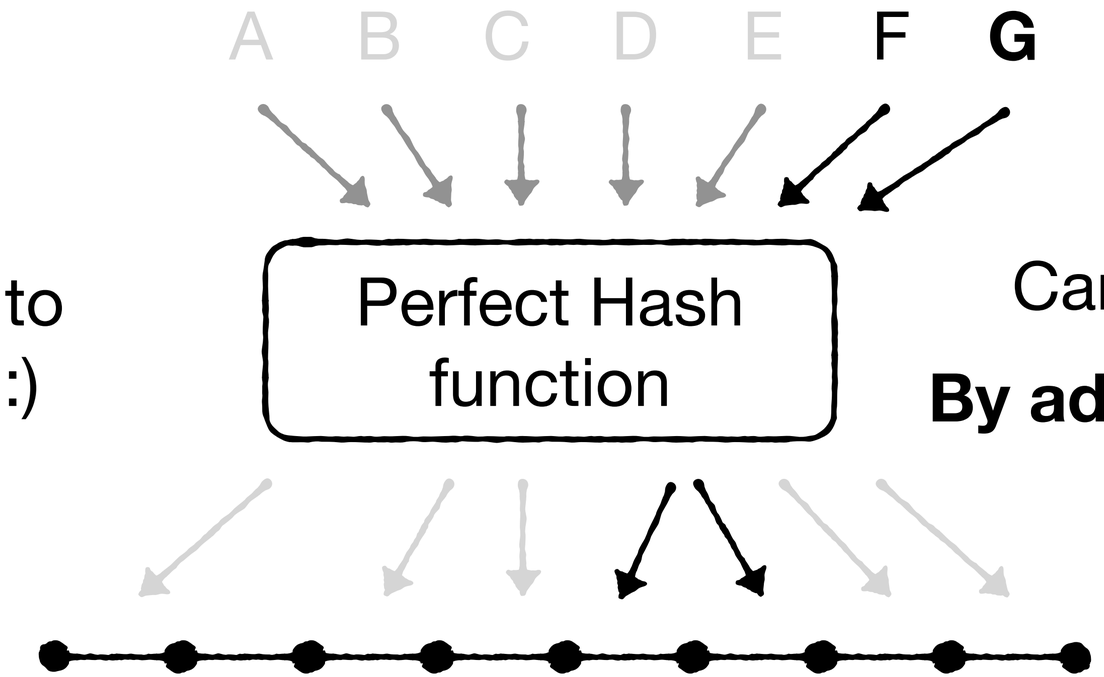
Just a data structure to ensure no collisions :)



Can handle insertions!

Dynamic Perfect Hashing

Just a data structure to ensure no collisions :)



Can handle insertions!

By adapting hash function

Dynamic Perfect Hashing

Storing a Sparse Table with 0(1) Worst Case Access Time. JACM 1984.
ML Fredman, J Komlós, E Szemerédi

The End of Moore's Law and the Rise of the Data Processor. VLDB 2021.
Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, Avraham Meir, Mark Mokryn, Iddo Naiss, Noam Rabinovich

Many more…

Dynamic Perfect Hashing

The End of Moore's Law and the Rise of the Data Processor. VLDB 2021.
Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, Avraham Meir, Mark Mokryn, Iddo Naiss, Noam Rabinovich

space-efficient, used in practice

Delta Hash Table

hash(X) = 0 1 0 1 0 0 1 1 0 1 0 1 1 0 0 …

hash(Y) = 0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 …

Delta Hash Table

hash(X) = **0 1 0 1 0 0 1 1 0** 1 0 1 1 0 0 ...

hash(Y) = **0 1 0 1 0 0 1 1 0** 1 0 1 1 0 1 ...

Same slot

Delta Hash Table

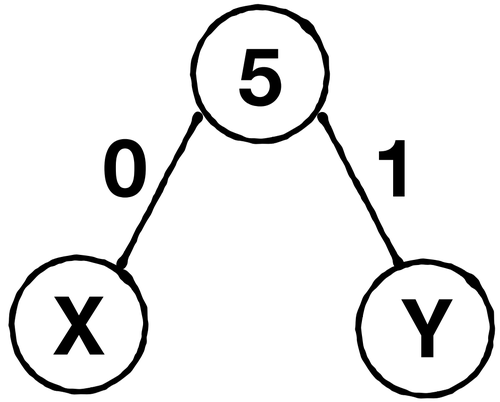
hash(X) = 1 0 1 1 0 0 ...

hash(Y) = 1 0 1 1 0 1 ...

Build Trie capturing index of first different bit

hash(X) = 1 0 1 1 0 **0** ...

hash(Y) = 1 0 1 1 0 **1** ...

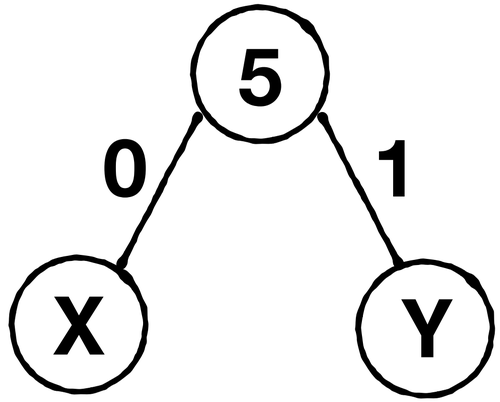


Build Trie capturing index of first different bit

hash(X) = **1** 0 1 1 0 0 ...

hash(Y) = **1** 0 1 1 0 1 ...

hash(Z) = **0** 1 0 0 1 1 ...

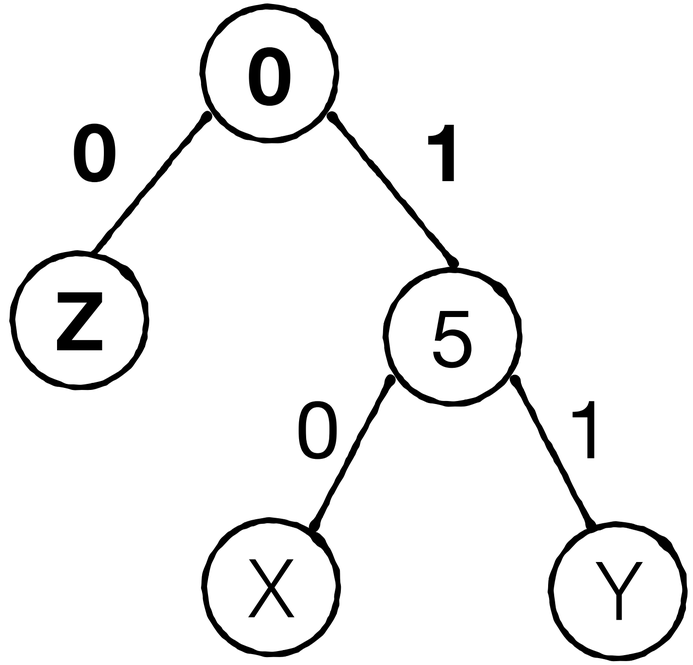


Build Trie capturing index of first different bit

hash(X) = **1** 0 1 1 0 0 ...

hash(Y) = **1** 0 1 1 0 1 ...

hash(Z) = **0** 1 0 0 1 1 ...



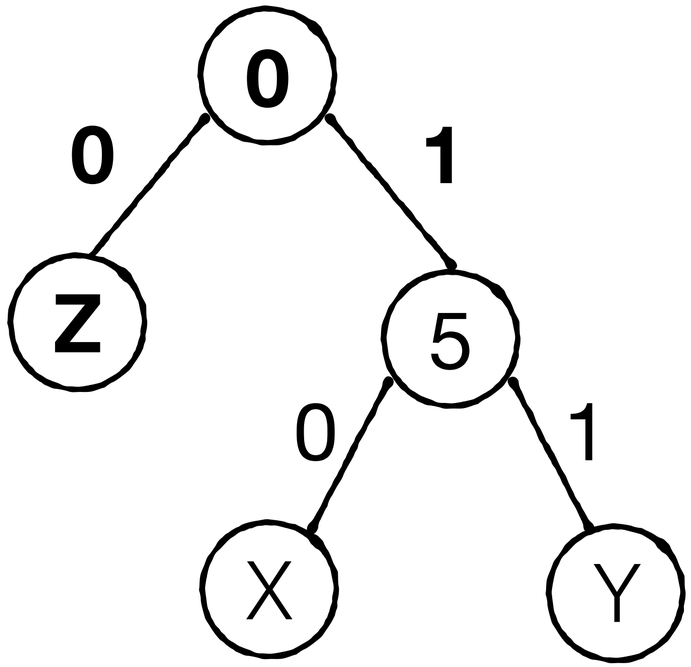
Build Trie capturing index of first different bit

hash(X) = 1 0 1 1 0 0 ...

hash(Y) = 1 0 1 1 0 1 ...

hash(Z) = 0 1 **0** 0 1 1 ...

hash(Q) = 0 1 **1** 0 0 1 ...



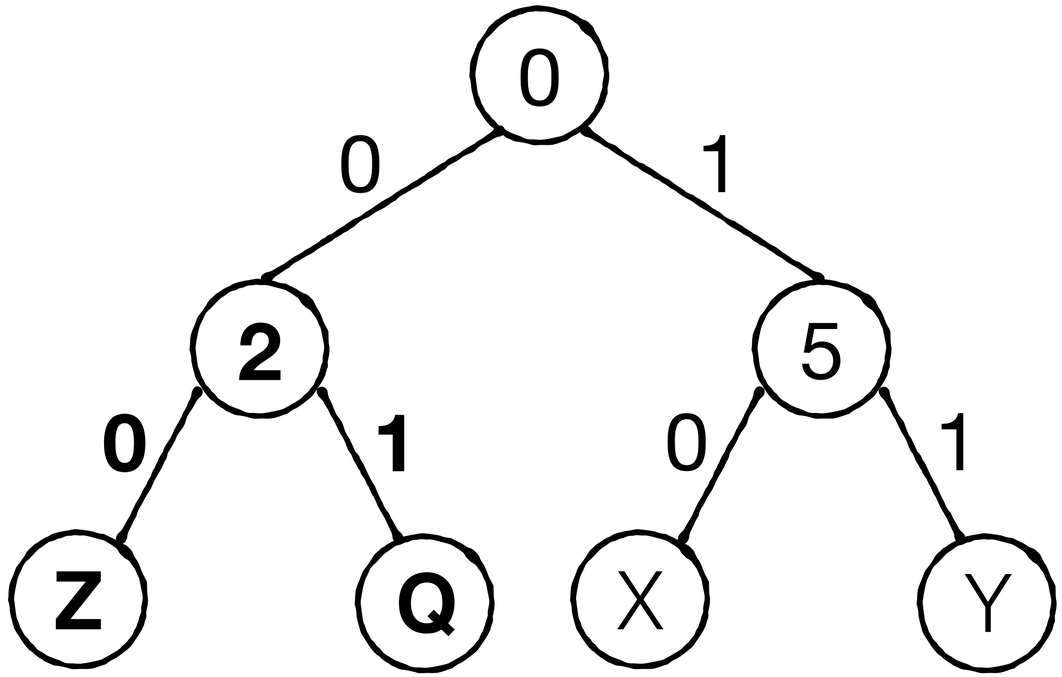
Build Trie capturing index of first different bit

hash(X) = 1 0 1 1 0 0 ...

hash(Y) = 1 0 1 1 0 1 ...

hash(Z) = 0 1 **0** 0 1 1 ...

hash(Q) = 0 1 **1** 0 0 1 ...



Build Trie capturing index of first different bit

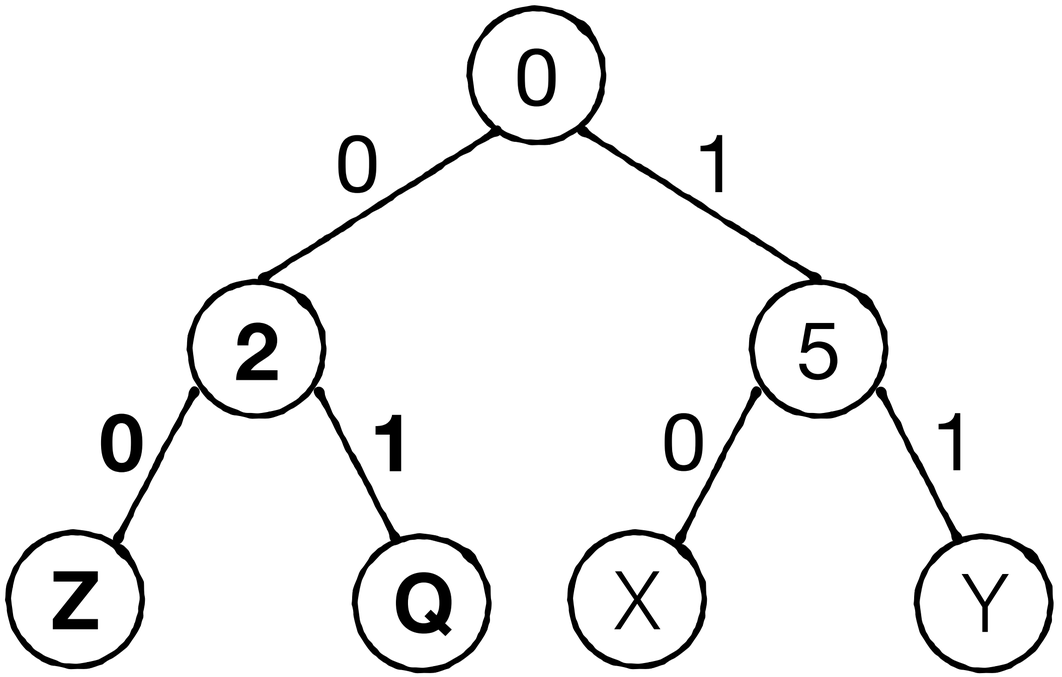
hash(X) = 1 0 1 1 0 0 ...

hash(Y) = 1 0 1 1 0 1 ...

hash(Z) = 0 **1** 0 0 1 1 ...

hash(Q) = 0 **1** 1 0 0 1 ...

hash(W) = 0 **0** 1 0 0 1 ...



Build Trie capturing index of first different bit

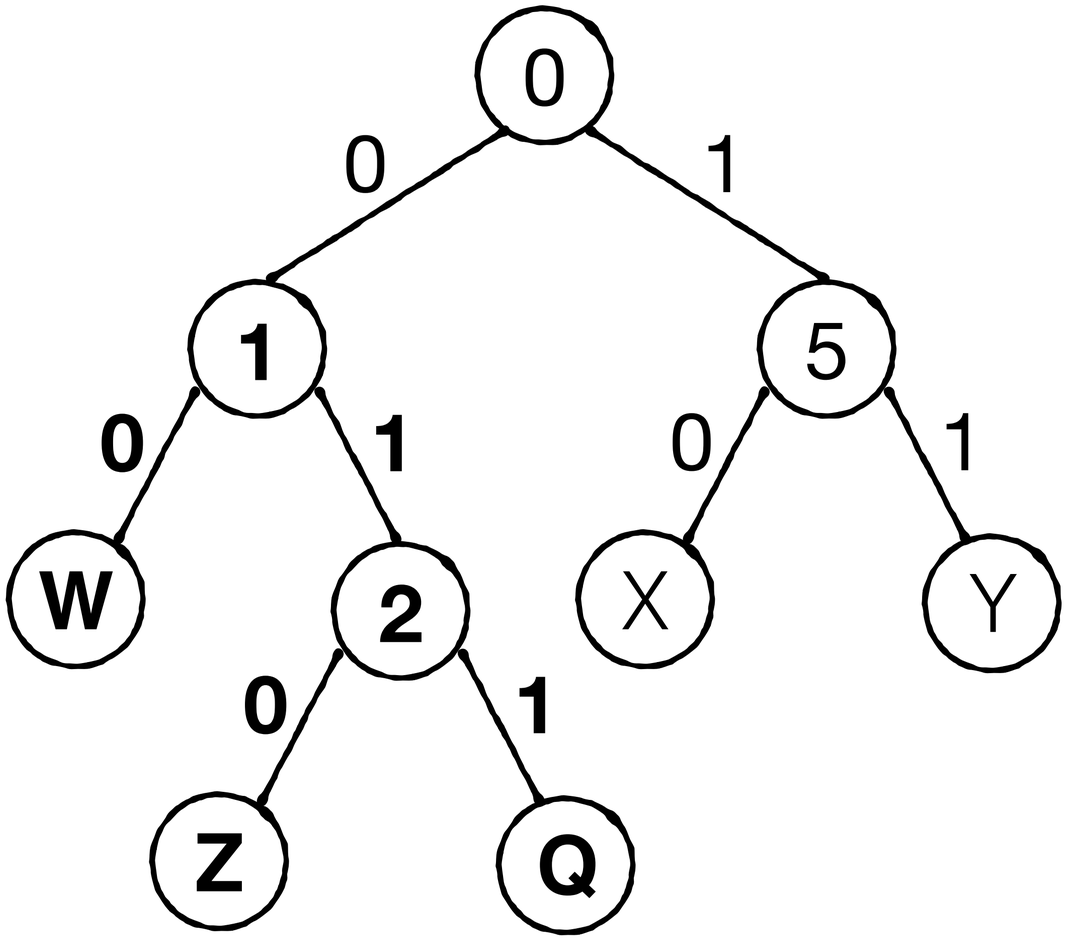
hash(X) = 1 0 1 1 0 0 ...

hash(Y) = 1 0 1 1 0 1 ...

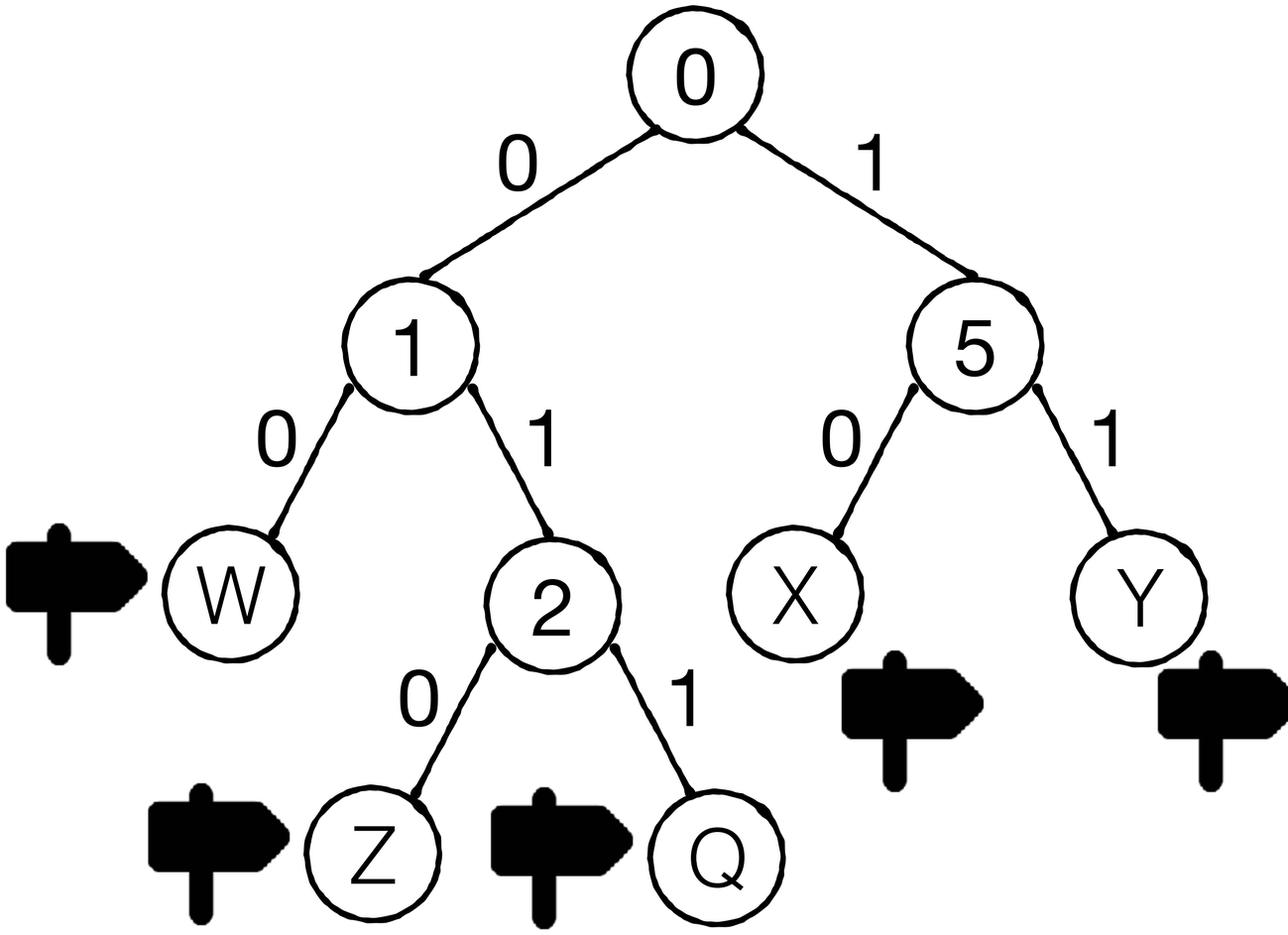
hash(Z) = 0 **1** 0 0 1 1 ...

hash(Q) = 0 **1** 1 0 0 1 ...

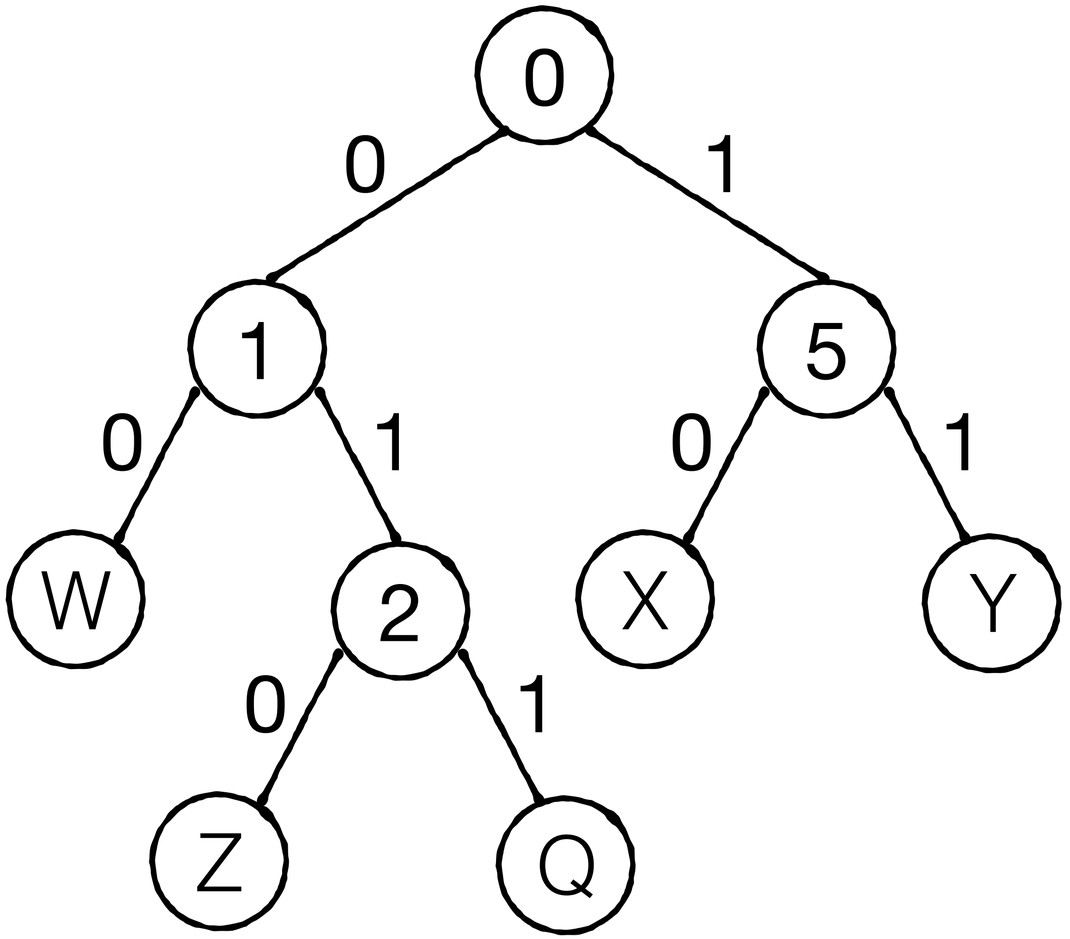
hash(W) = 0 **0** 1 0 0 1 ...



Place pointer/payload in leafs

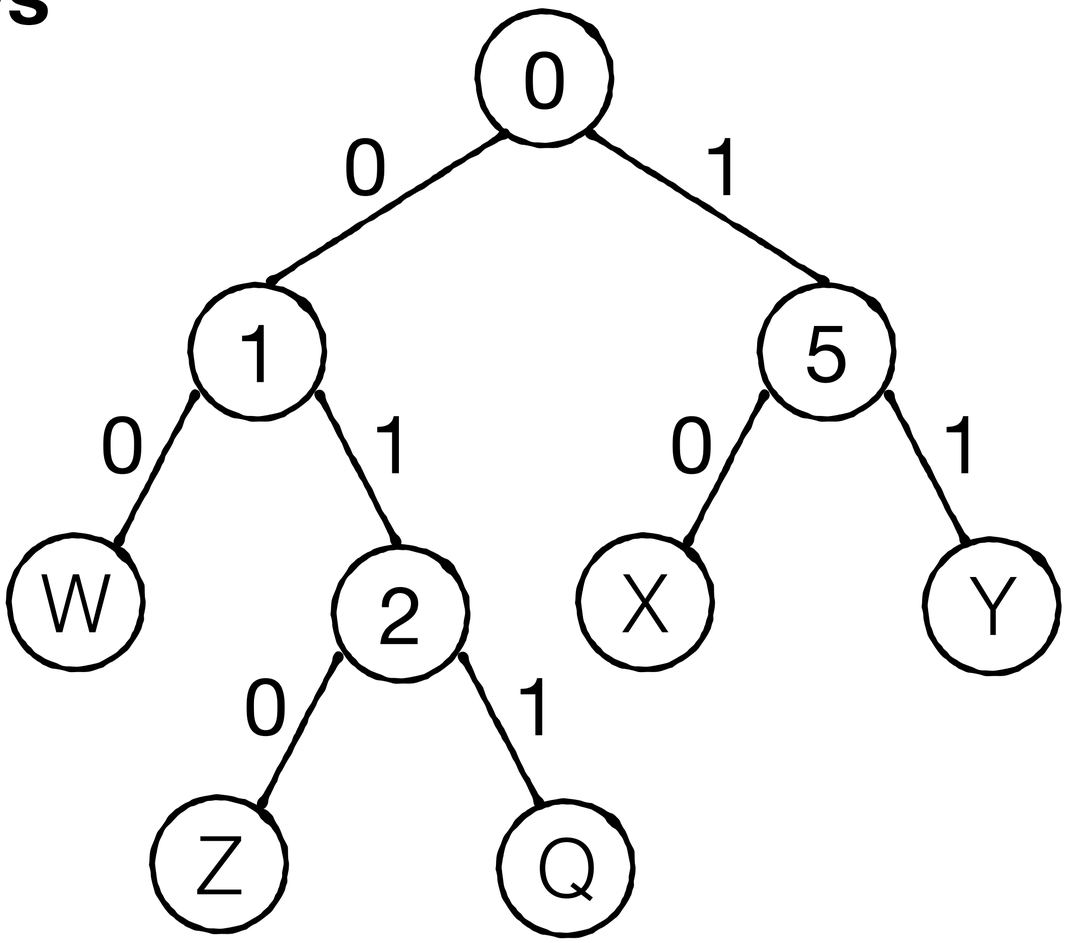


Existing keys are fully differentiated

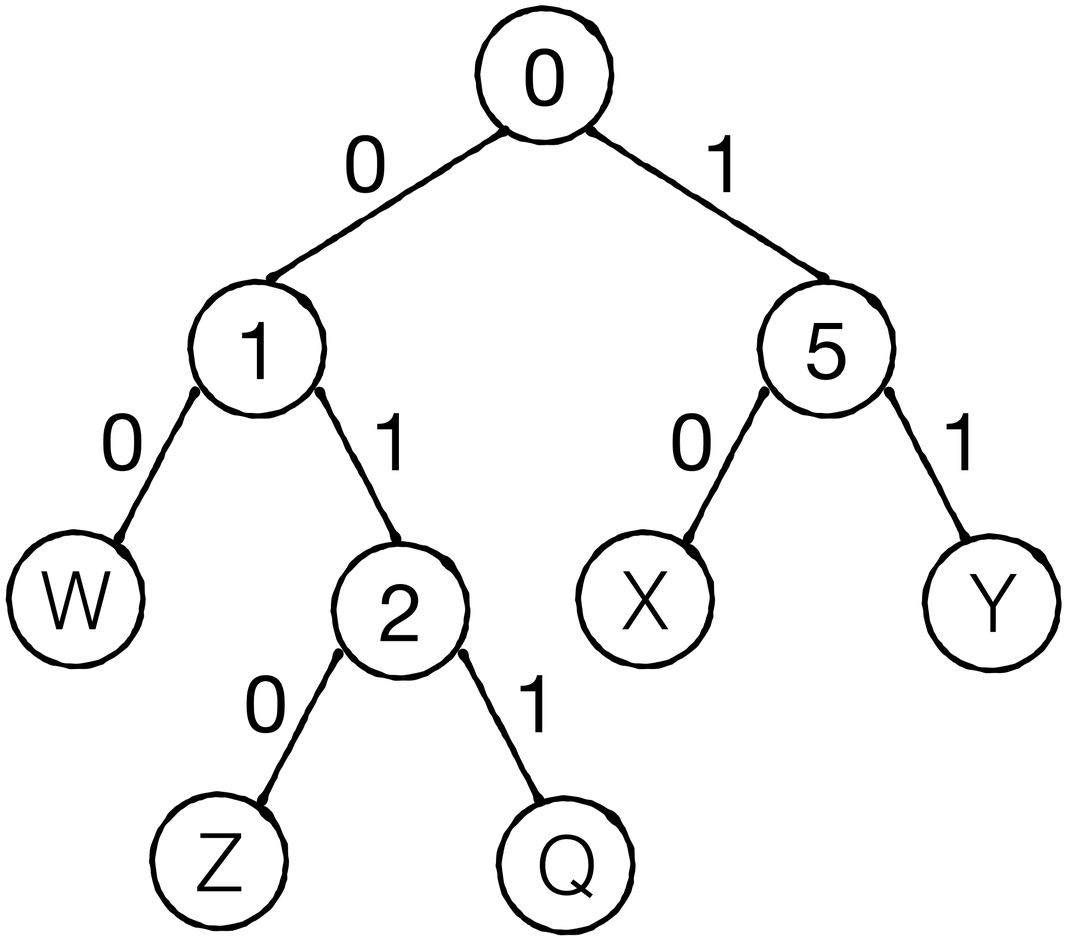


Existing keys are fully differentiated

A query to an existing key always finds correct payload

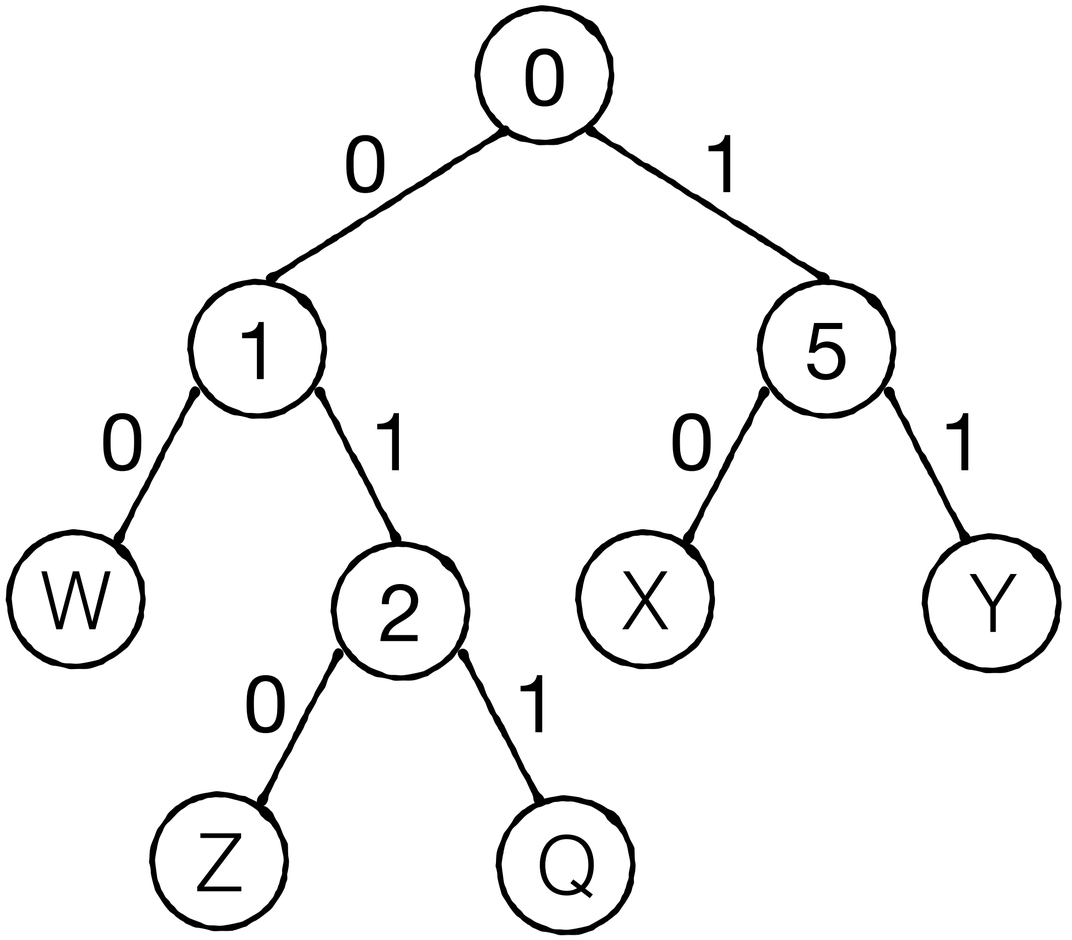


How about a query to non-existing key?

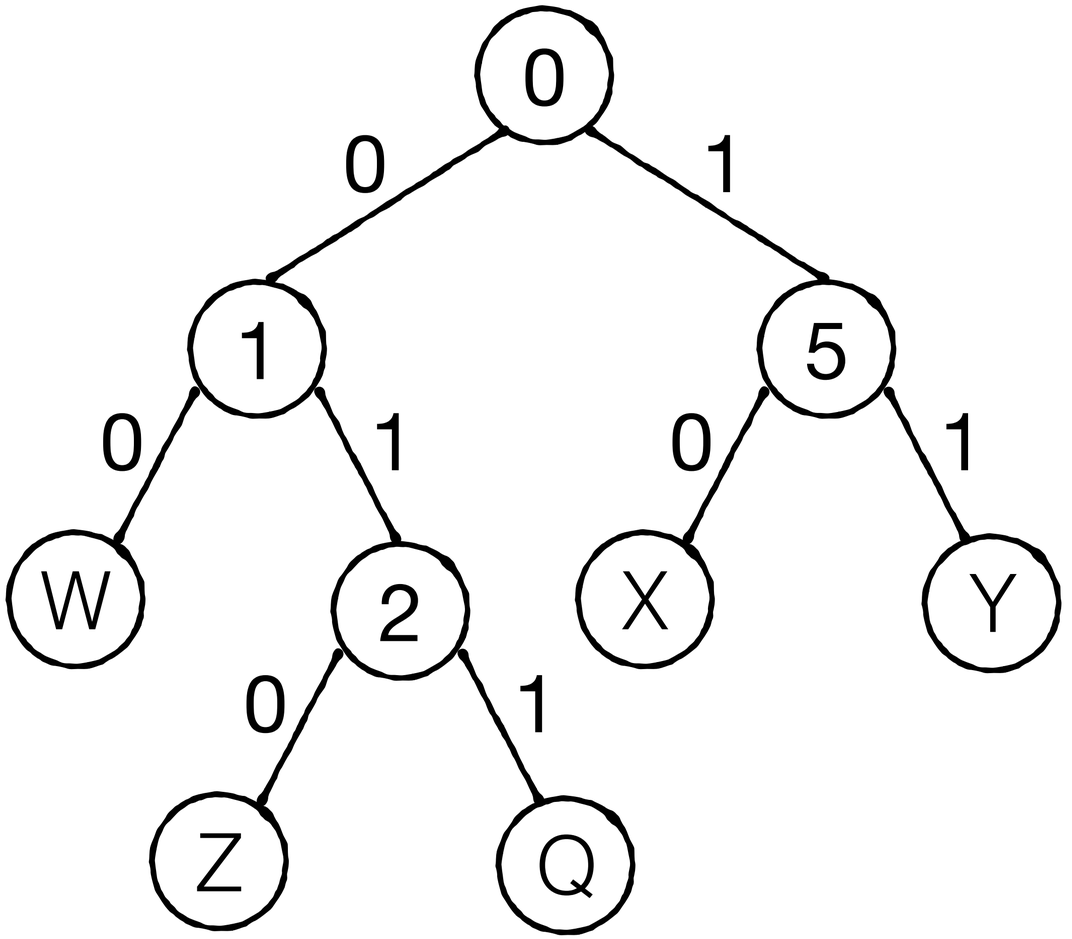


How about a query to non-existing key?

Finds empty slot or returns payload associated with some other key

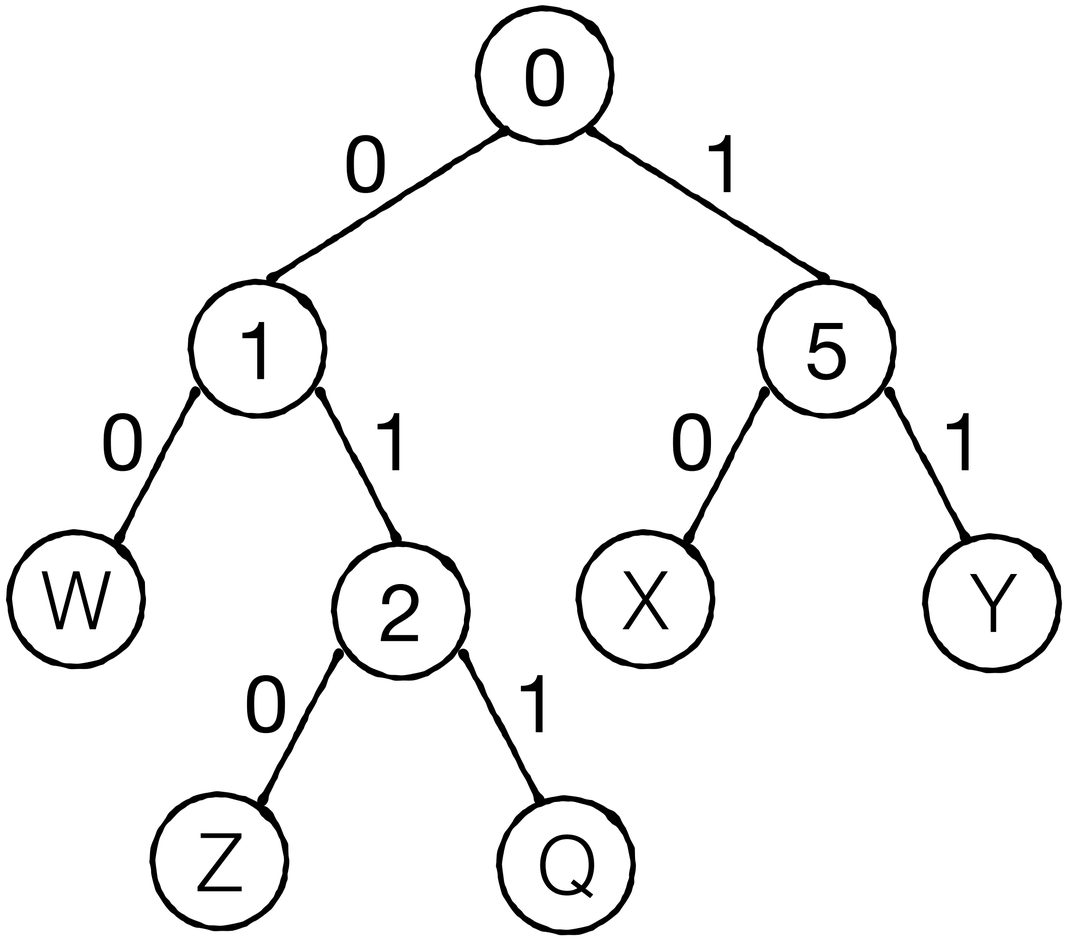


How to encode this trie efficiently within a hash table bucket?

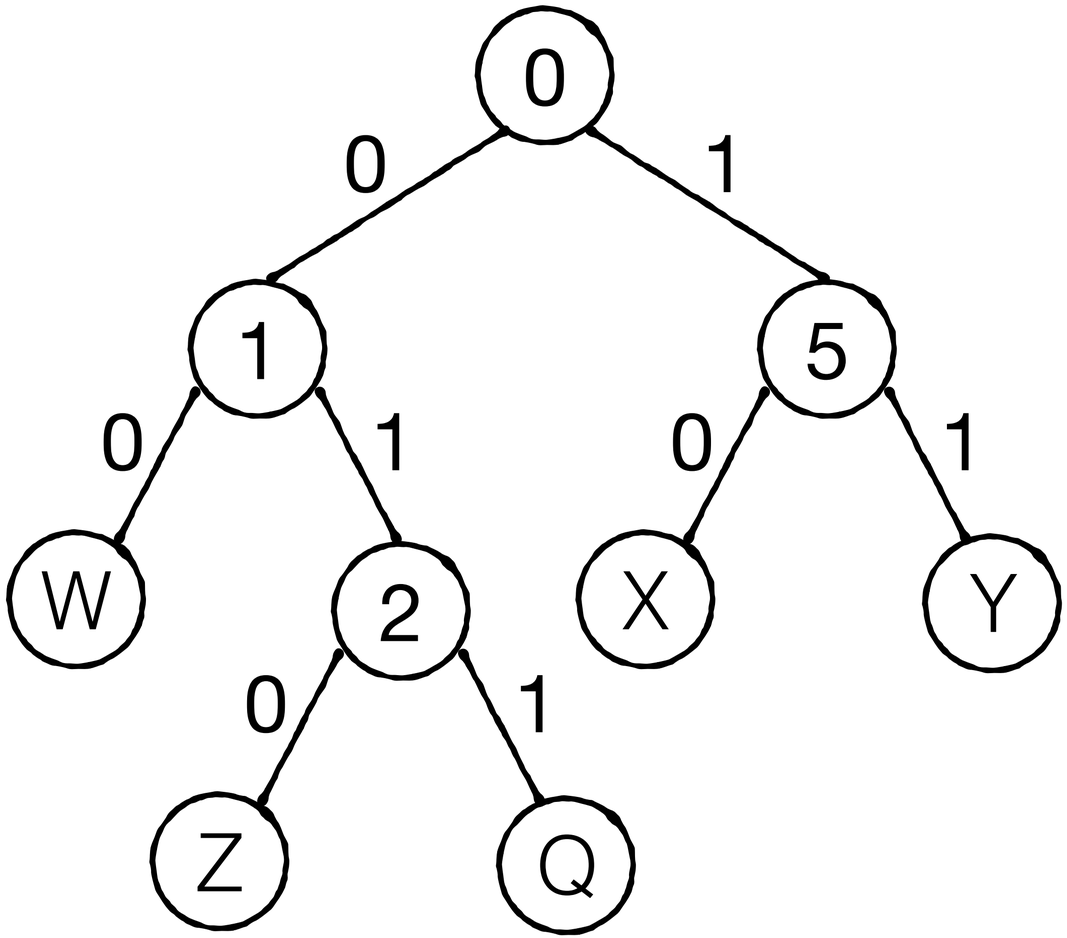


How to encode this trie efficiently within a hash table bucket?

Encode # entries in unary: 111110



How to encode this trie efficiently within a hash table bucket?

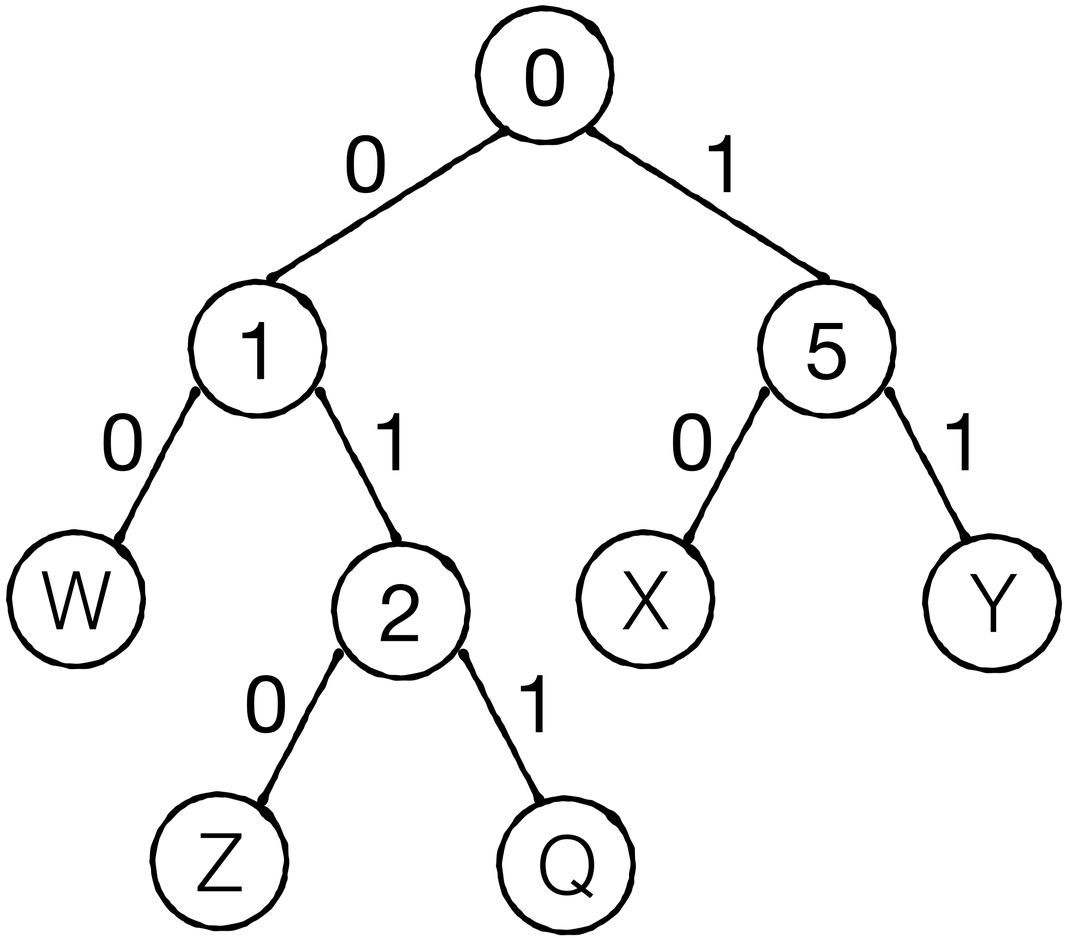


Encode # entries in unary: 111110

Unary is ideal since trie is usually small (0-2 entries)

entries

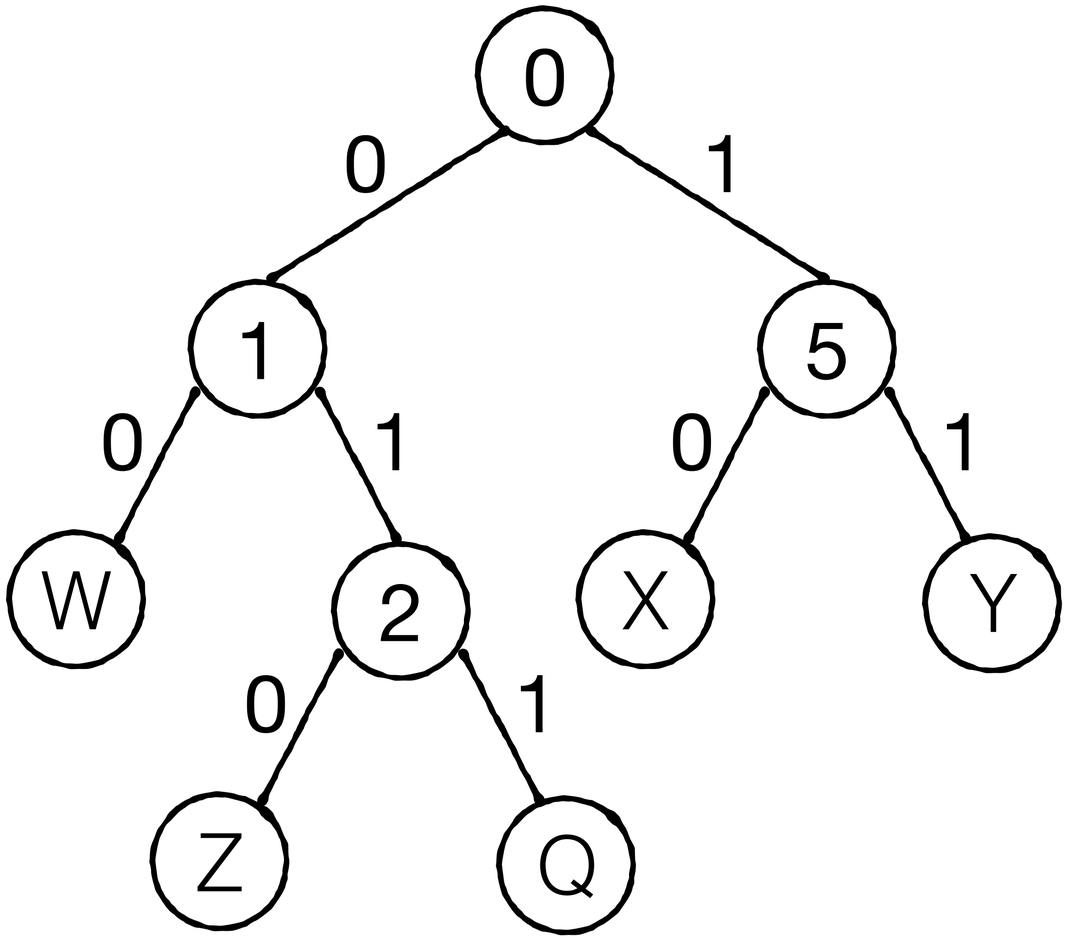
111110



entries

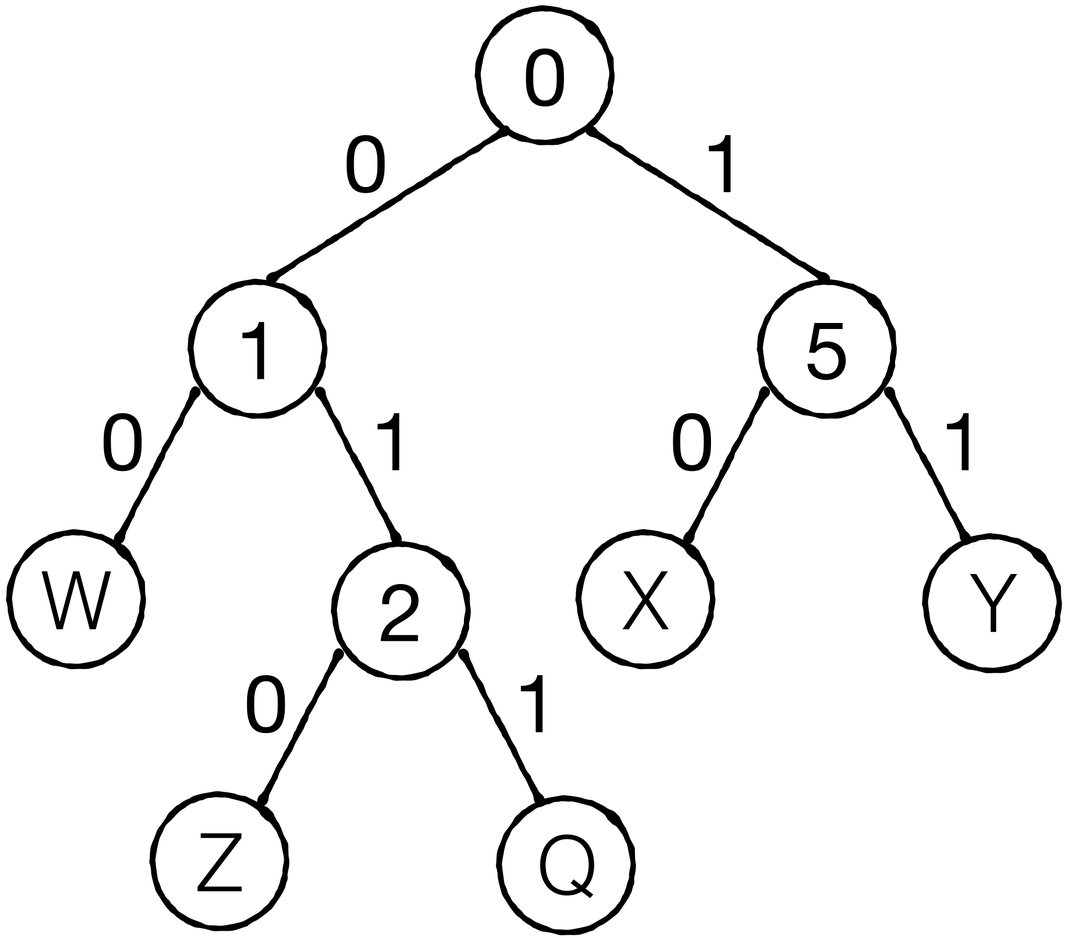
111110

Trie topology?



entries

111110



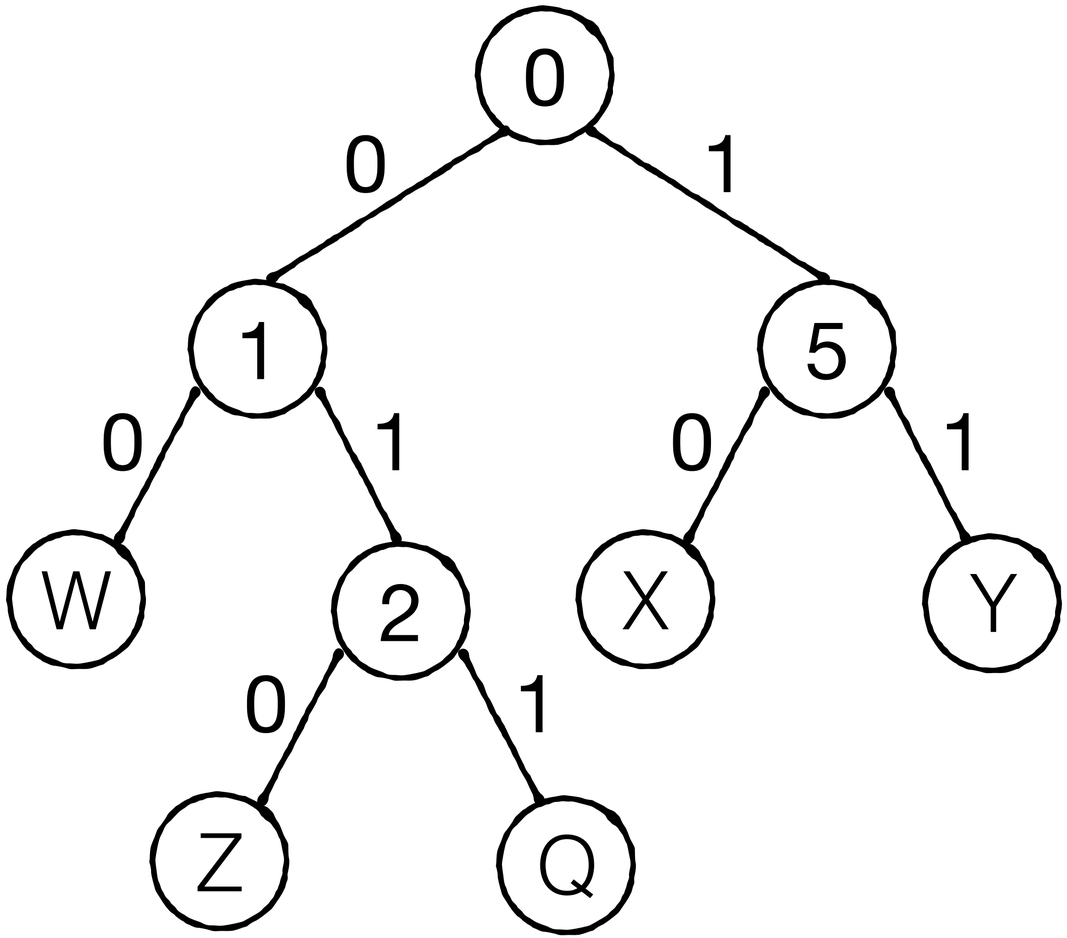
Trie topology?

00 - no children
01 - one right child node
10 - one left child node
11 - two children

entries

111110

Trie topology?



00 - no children
01 - one right child node
10 - one left child node
11 - two children

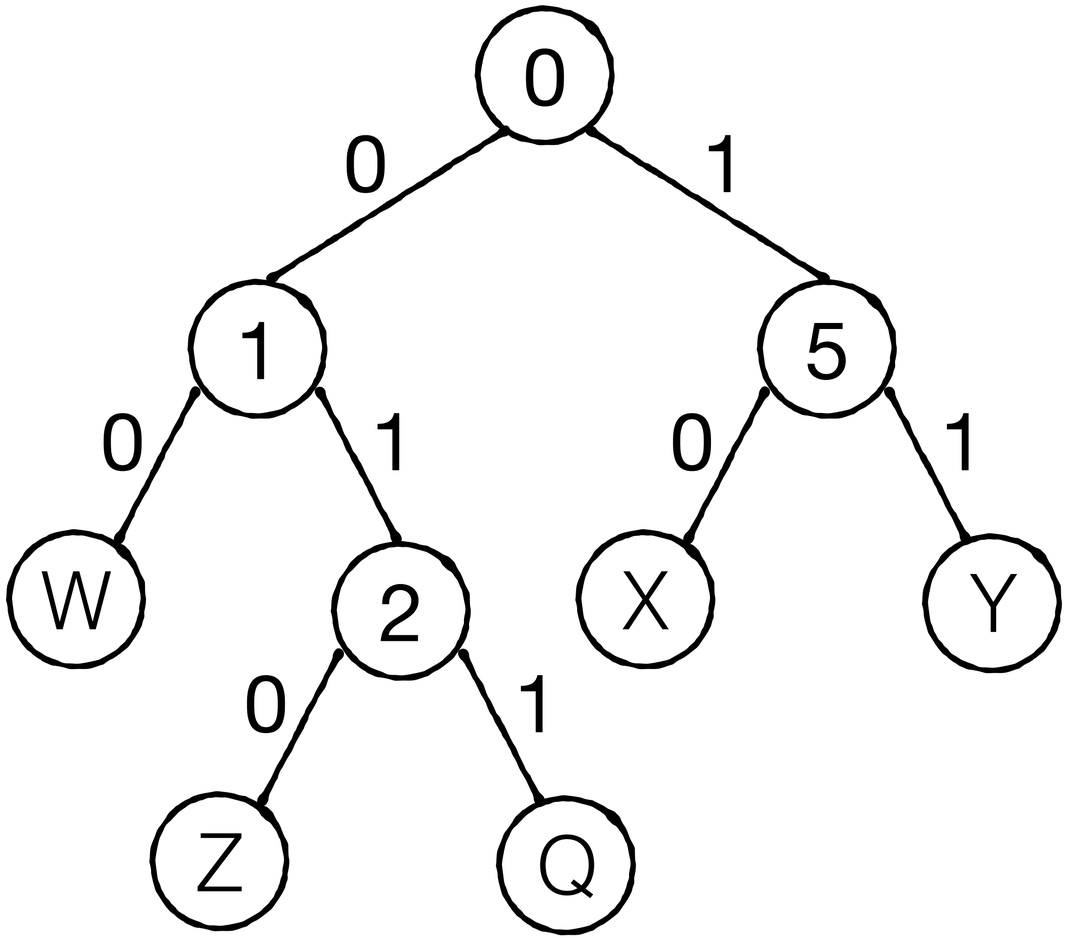
Encode structure depth-first

11 01 00 00

entries

111110

Trie topology?



00 - no children
01 - one right child node
10 - one left child node
11 - two children

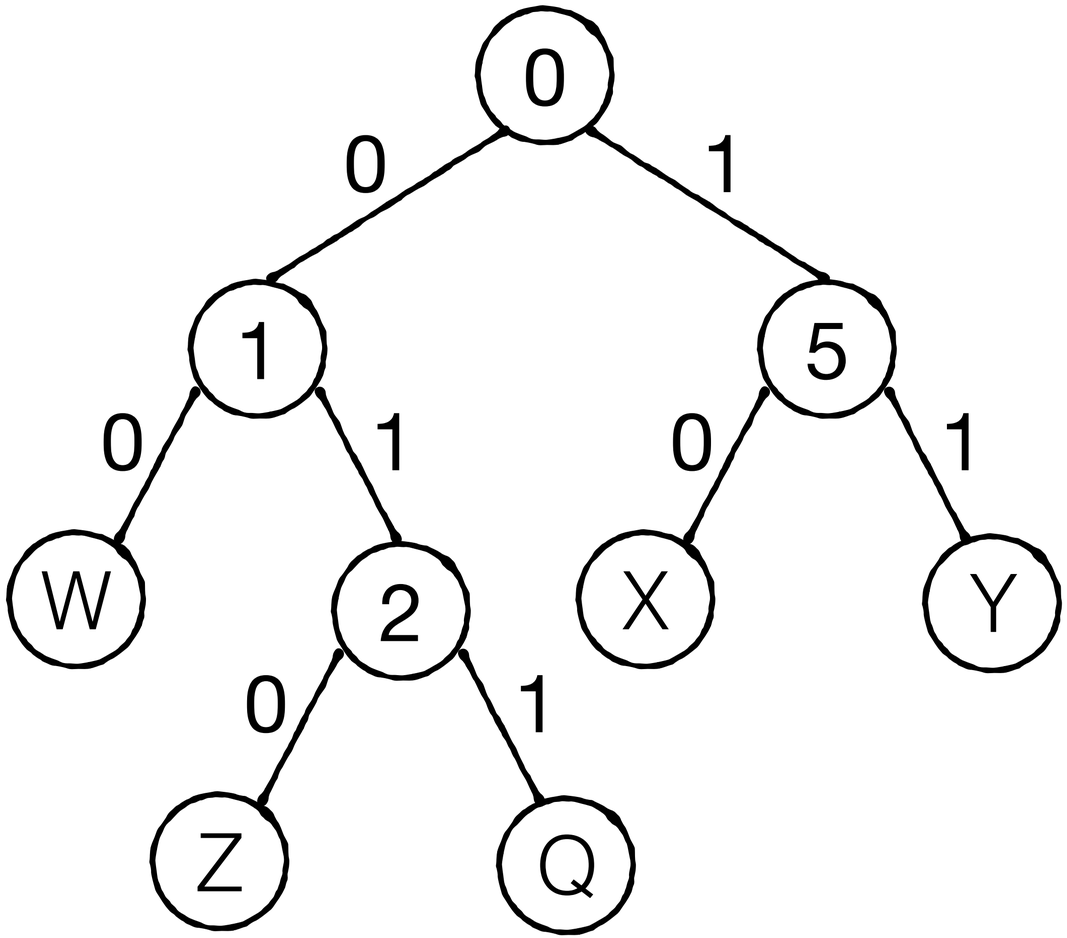
Encode structure depth-first

11 01 00 ~~00~~

Last bits always zero

entries

111110



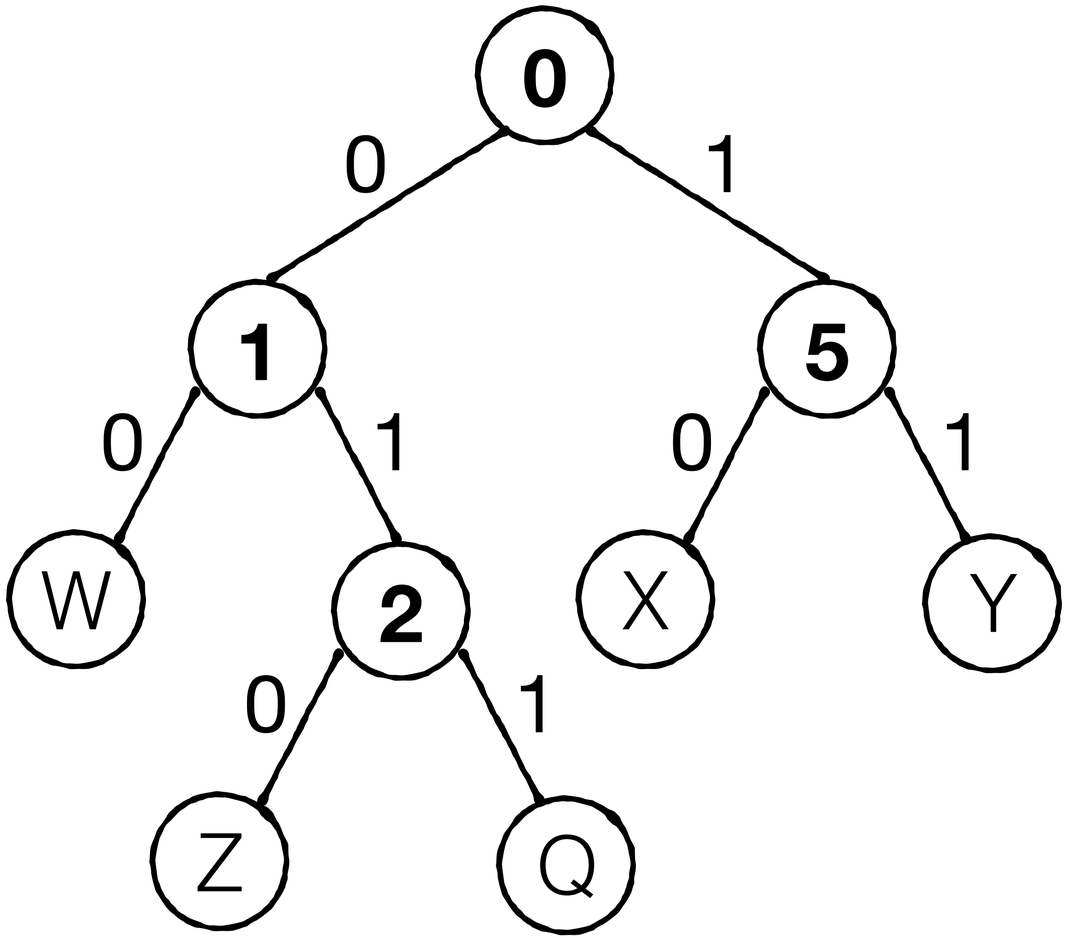
Trie topology?

00 - no children
01 - one right child node
10 - one left child node
11 - two children

Encode structure depth-first

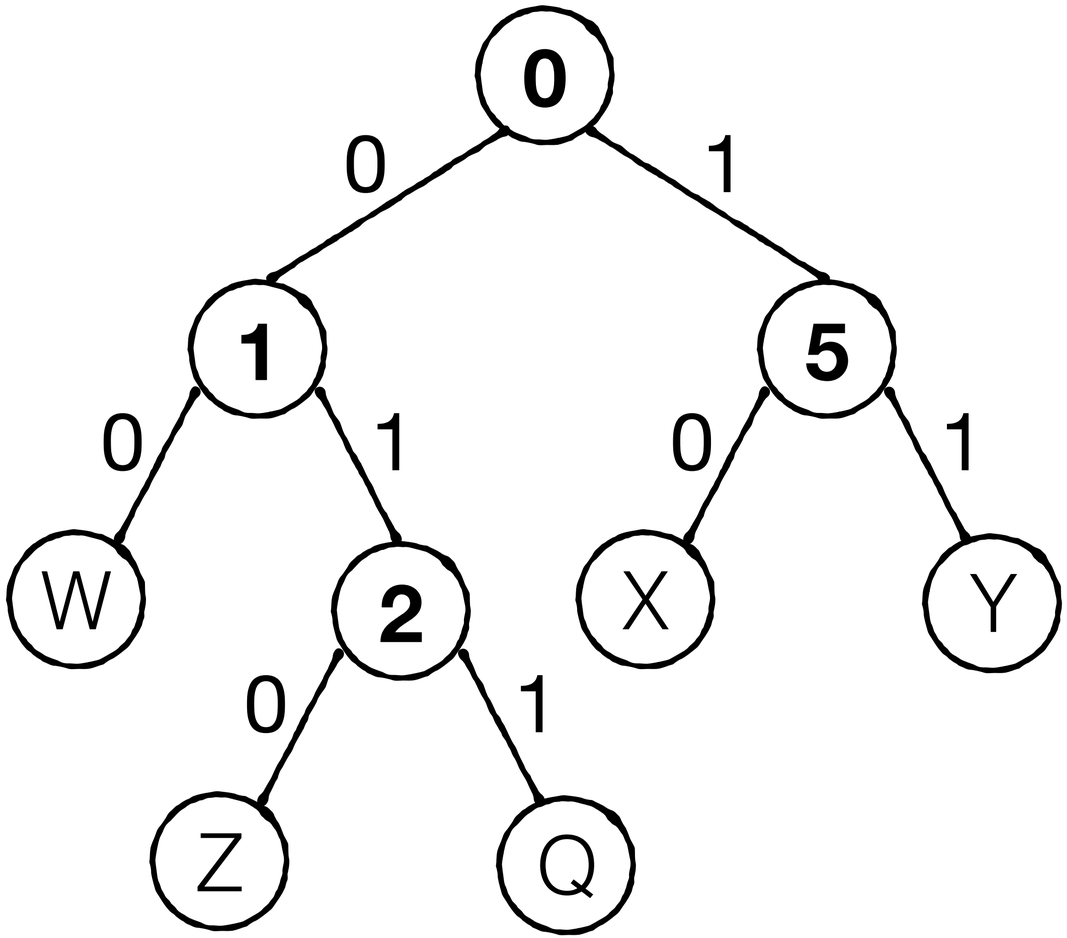
11 01 00

# entries	**Topology**
111110	**11 01 00**



# entries	Topology
111110	11 01 00

Differentiating bit indexes?



entries

111110

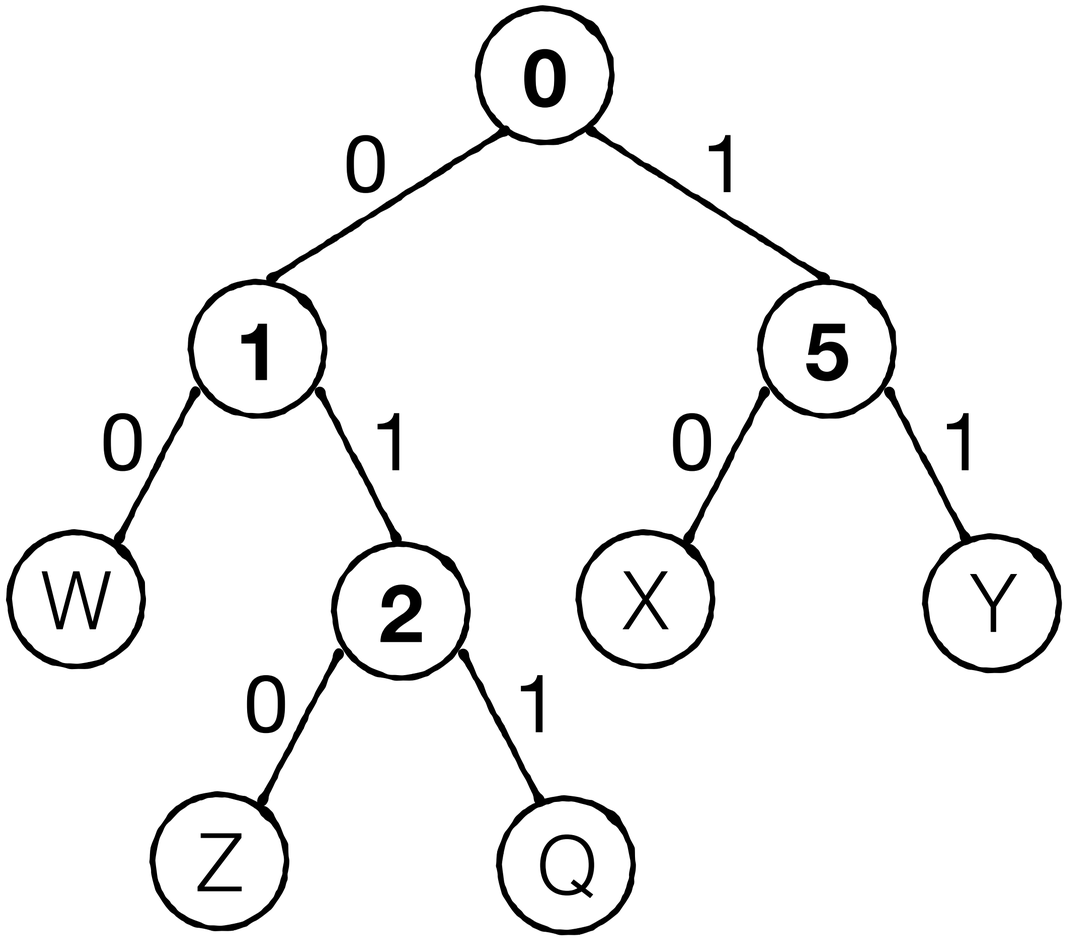
Topology

11 01 00

Differentiating bit indexes?

Depth-first, unary

0 10 110 111110

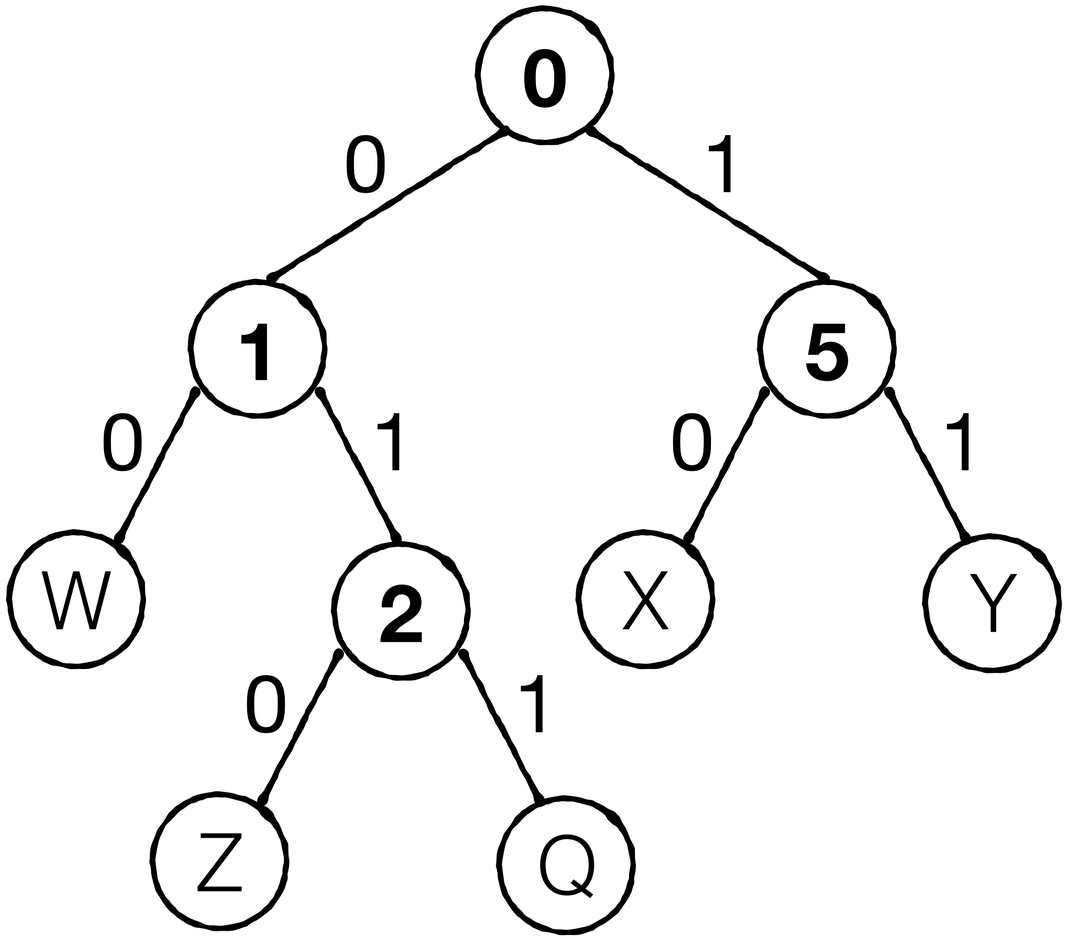


entries

111110

Topology

11 01 00



Differentiating bit indexes?

Depth-first, unary

0 10 110 111110

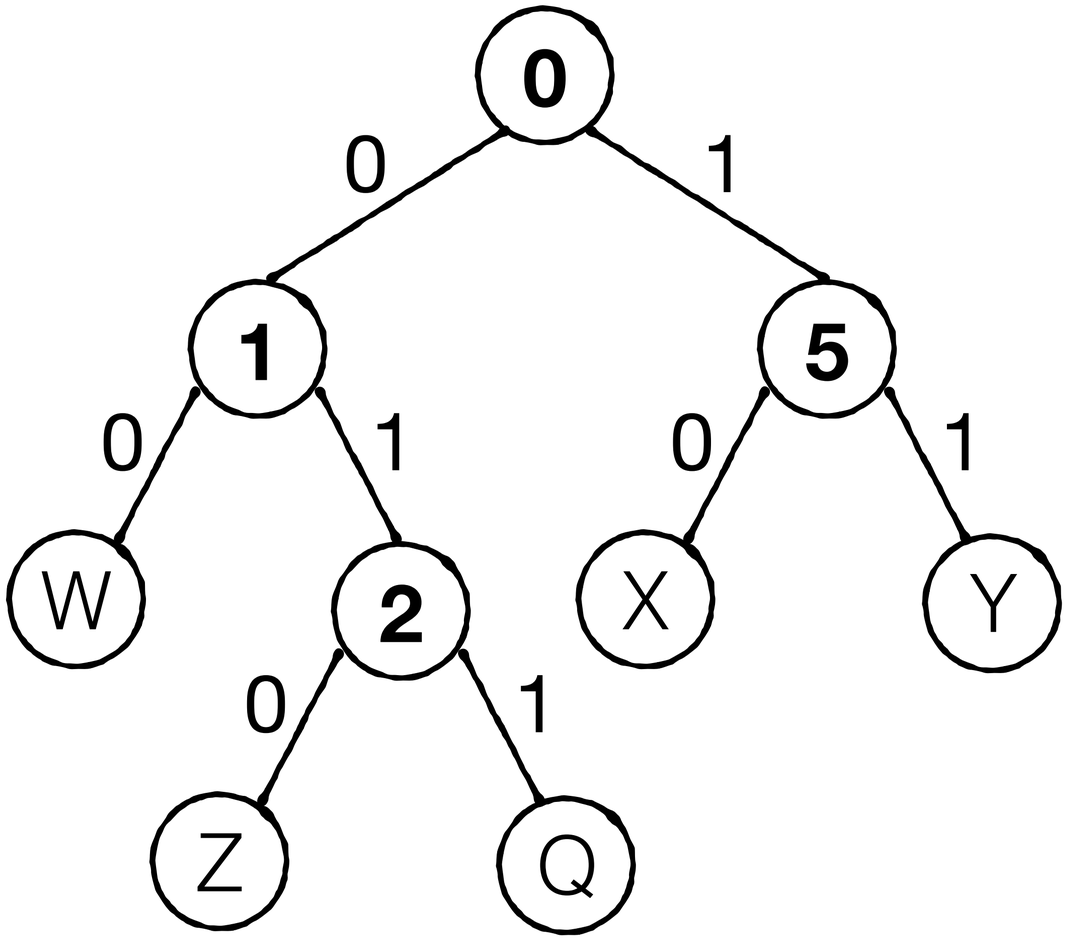
0 1 2 5

entries

111110

Topology

11 01 00



Differentiating bit indexes?

Depth-first, unary

0 10 110 111110

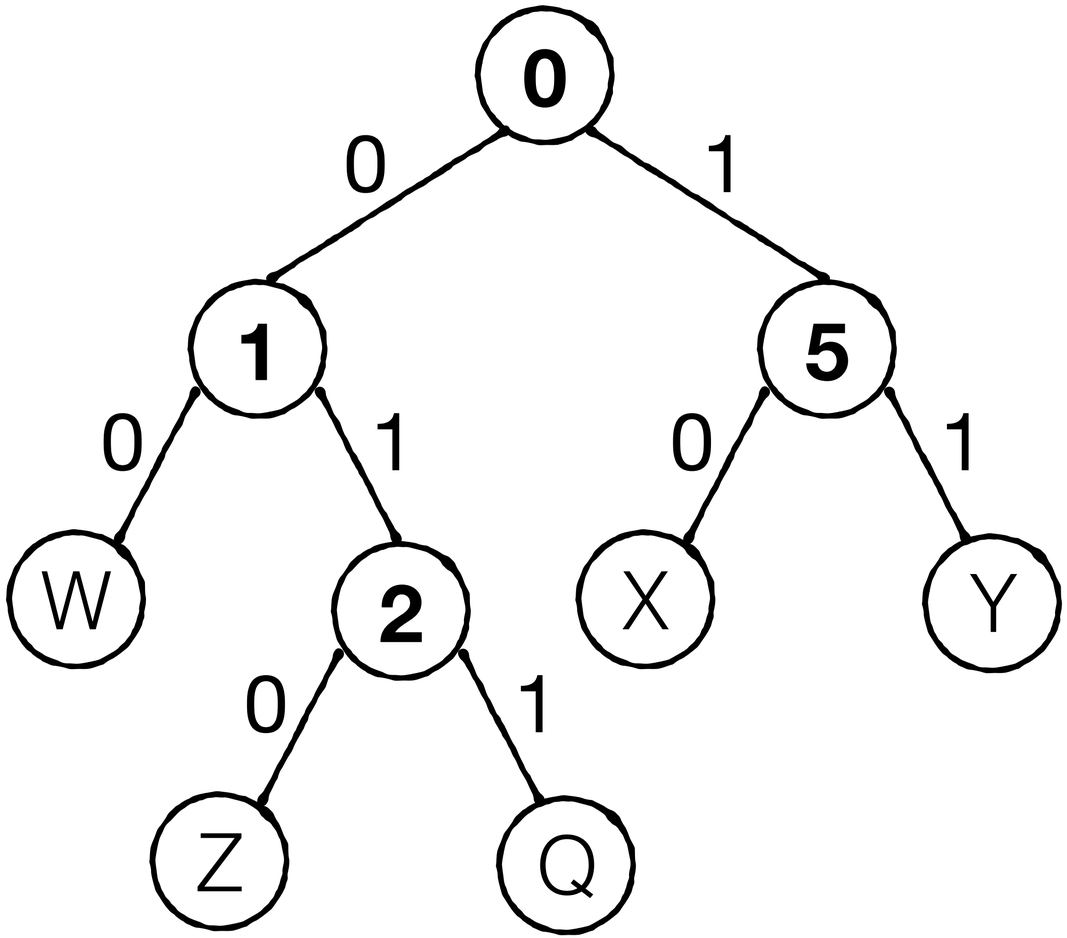
child’s index is larger than parent’s by at least 1

entries

111110

Topology

11 01 00



Differentiating bit indexes?

Depth-first, unary

0 10 110 111110

child's index is larger than parent's by at least 1

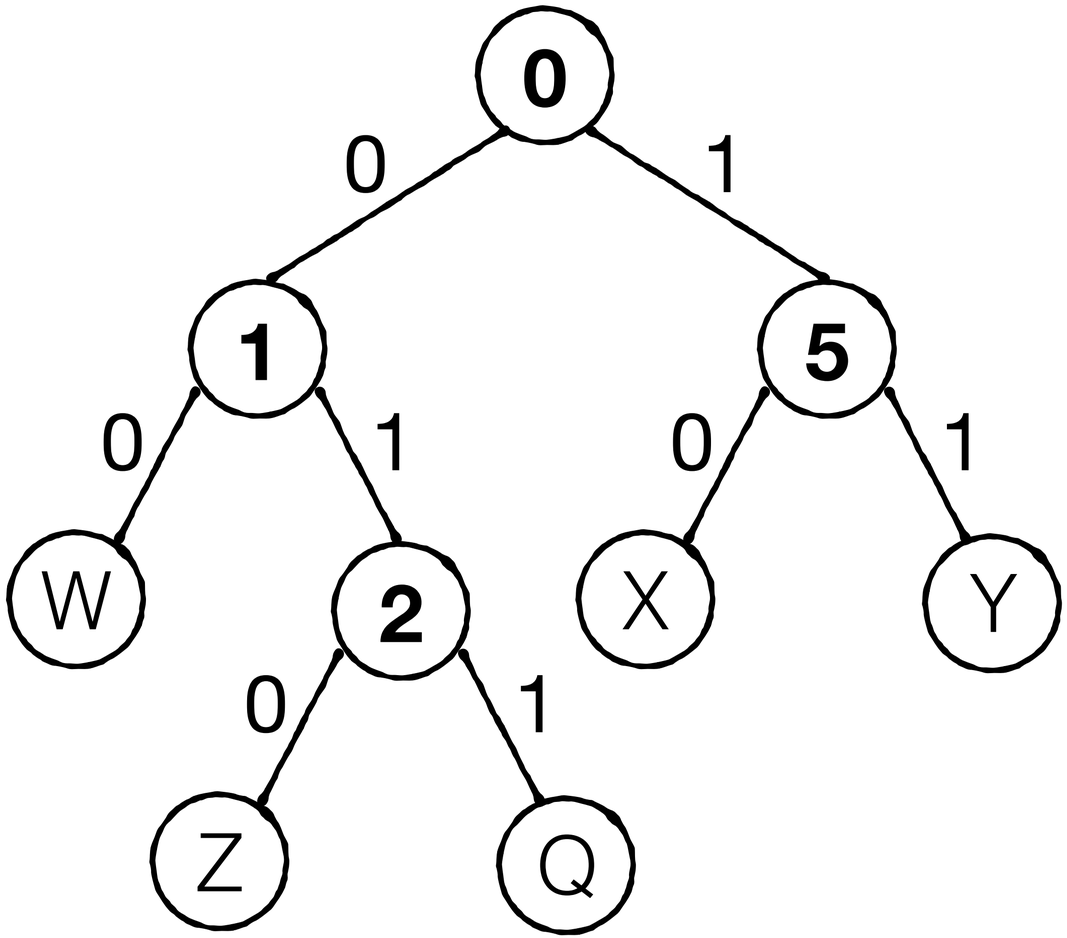
So let's encode index as Δ - 1

entries

111110

Topology

11 01 00



Differentiating bit indexes?

Depth-first, unary

0 ~~1~~0 ~~11~~0 ~~1~~11110

child’s index is larger than parent’s by at least 1

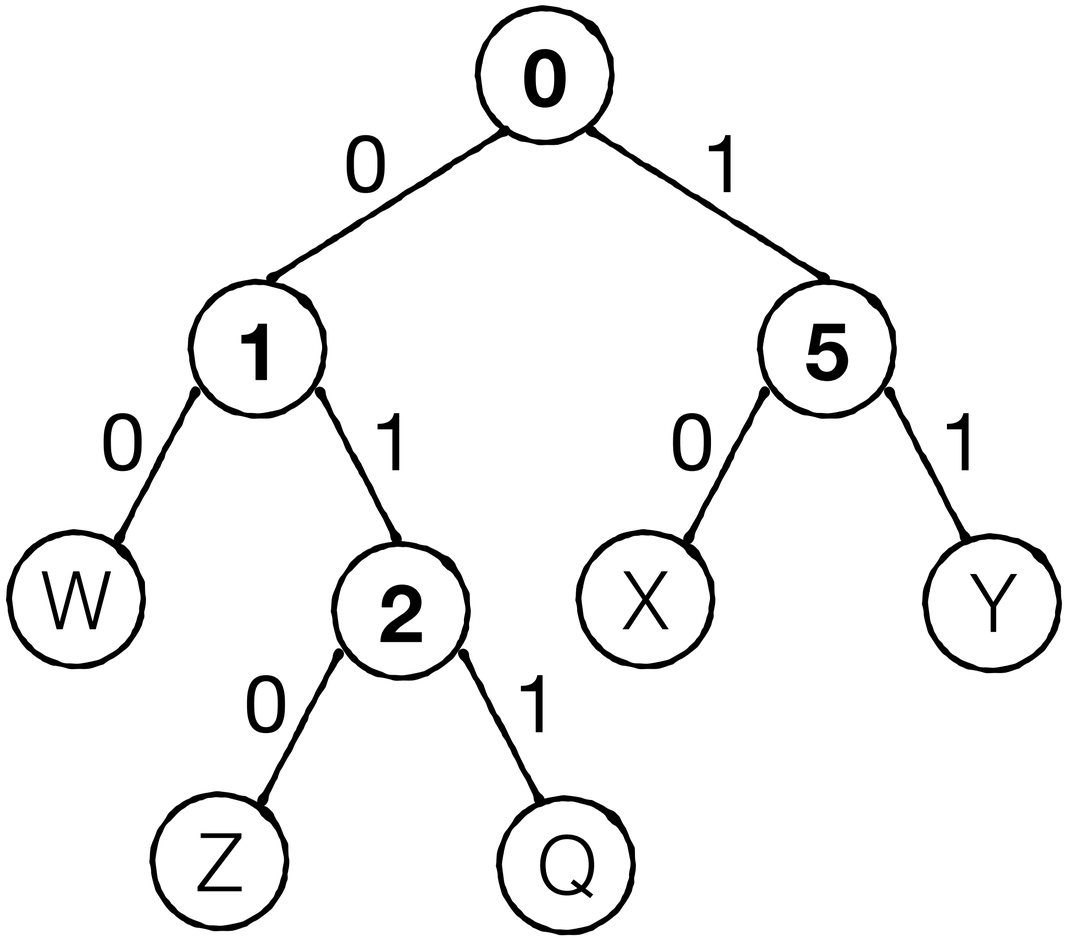
So let’s encode index as Δ - 1

entries

111110

Topology

11 01 00



Differentiating bit indexes?

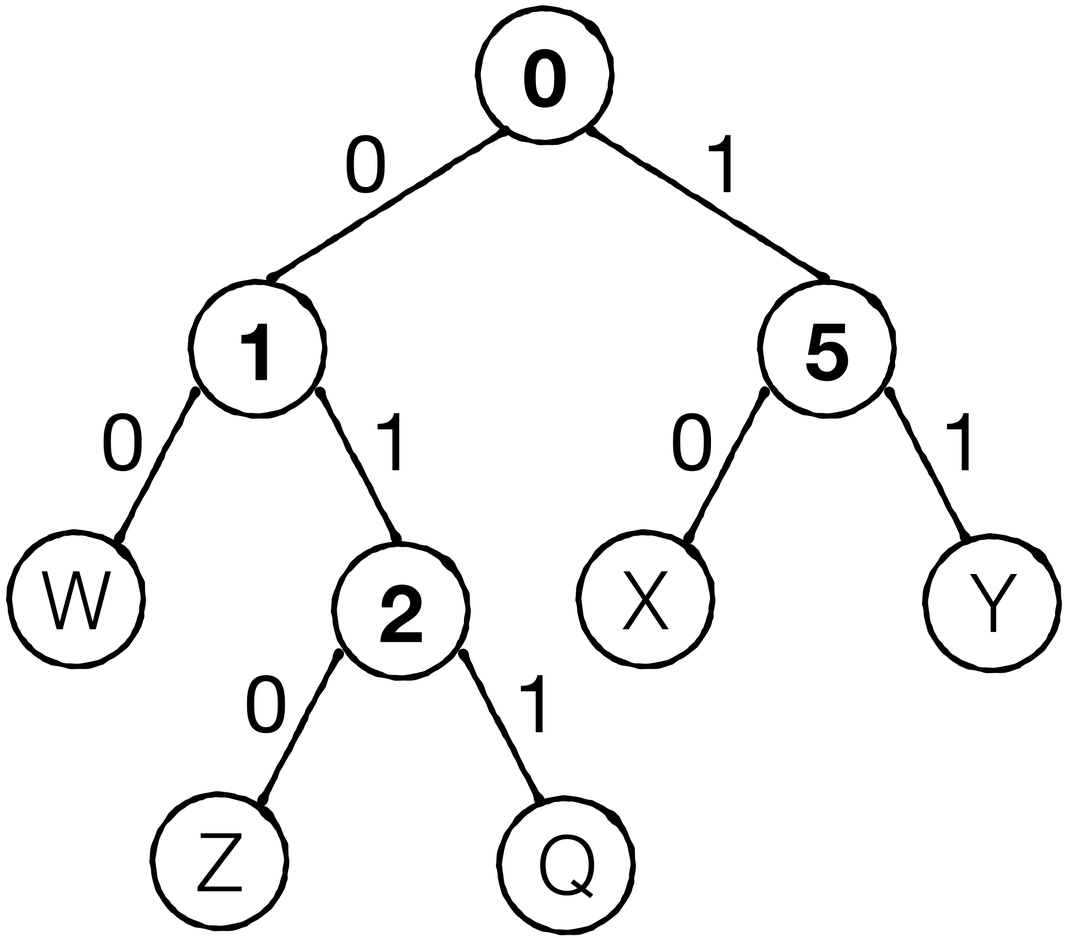
Depth-first, unary

0 0 0 11110

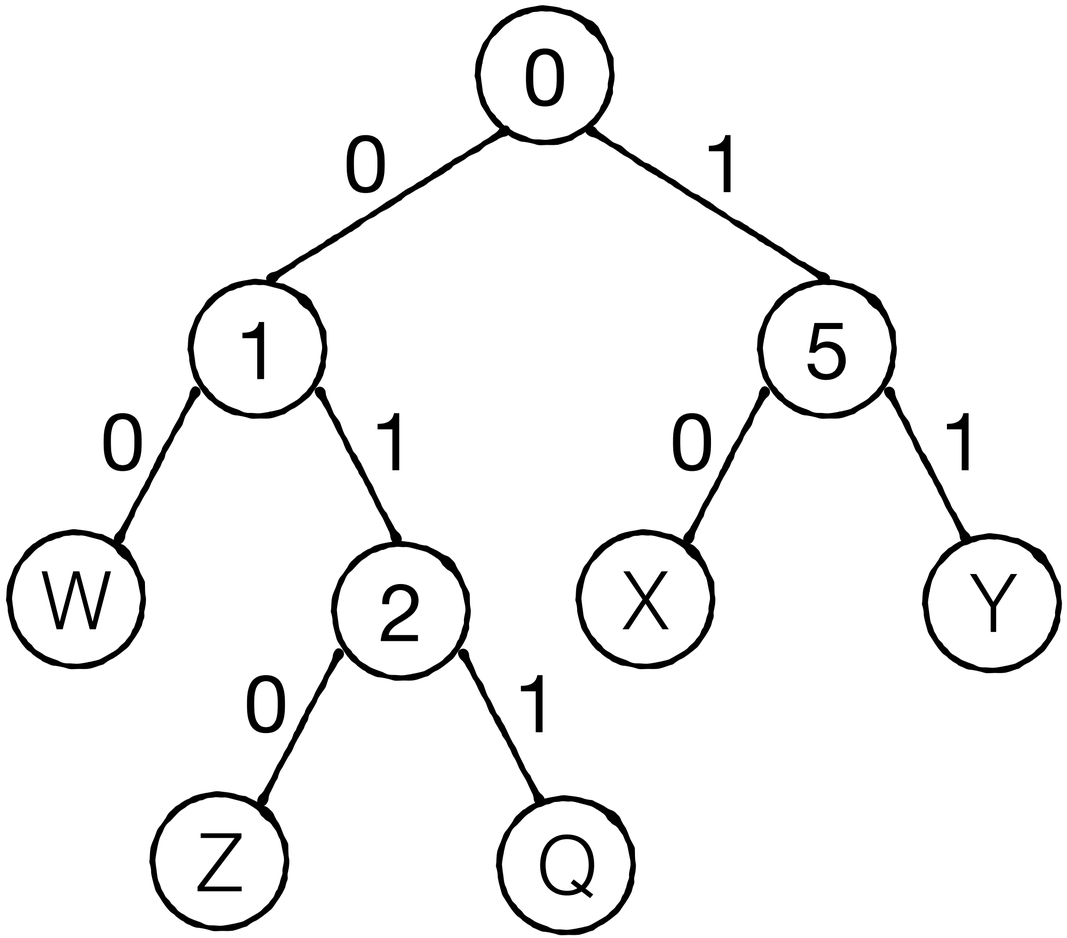
child’s index is larger than parent’s by at least 1

So let’s encode index as Δ - 1

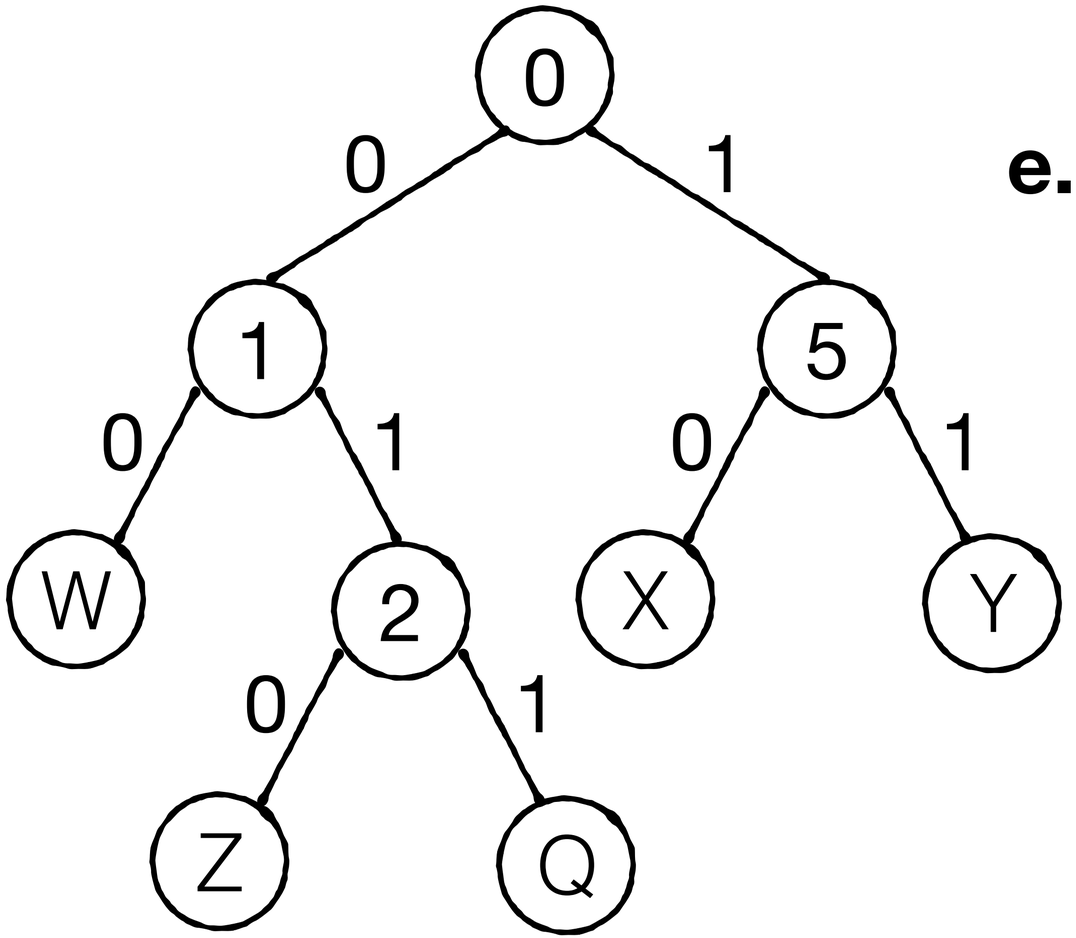
# entries	Topology	**Indices**
111110	11 01 00	**0 0 0 11110**



# entries	Topology	Indices	**Pointers**
111110	11 01 00	0 0 0 11110	**W Z Q X Y**

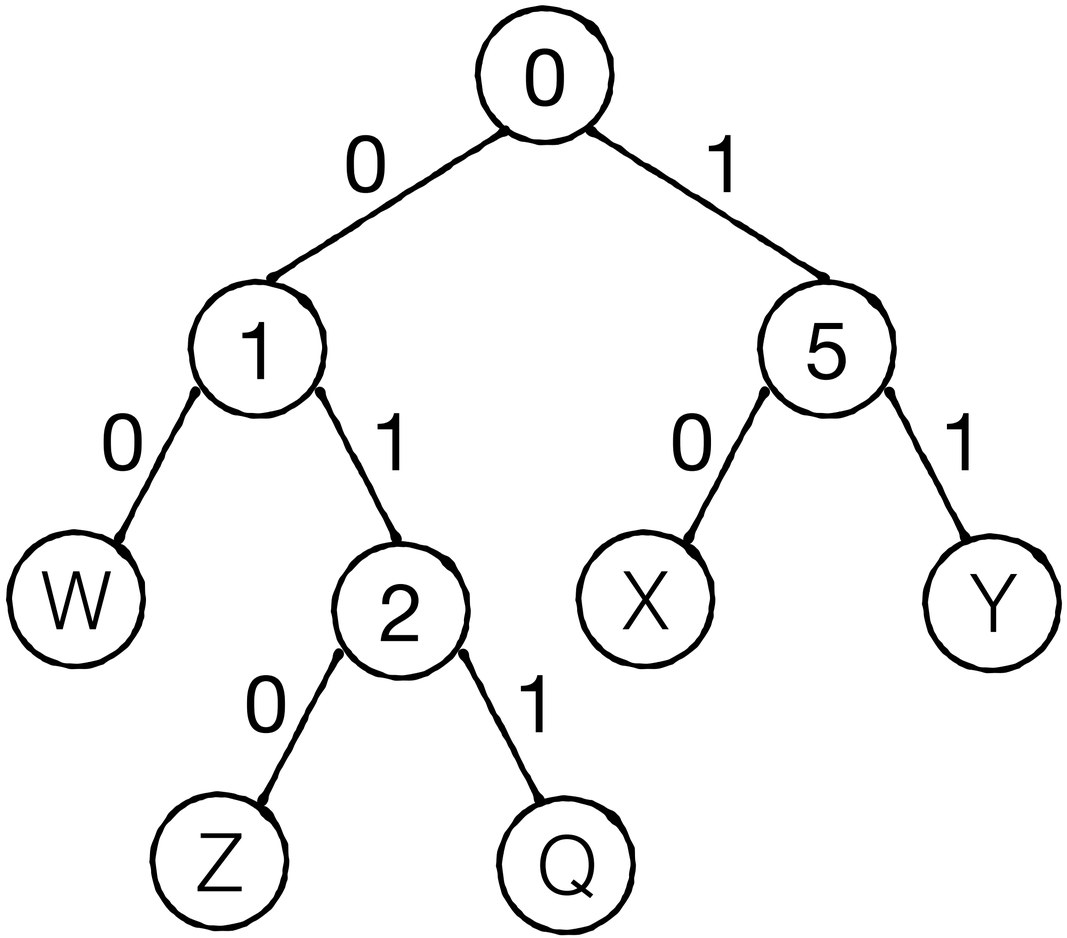


# entries	Topology	Indices	Pointers
111110	11 01 00	0 0 0 11110	W Z Q X Y



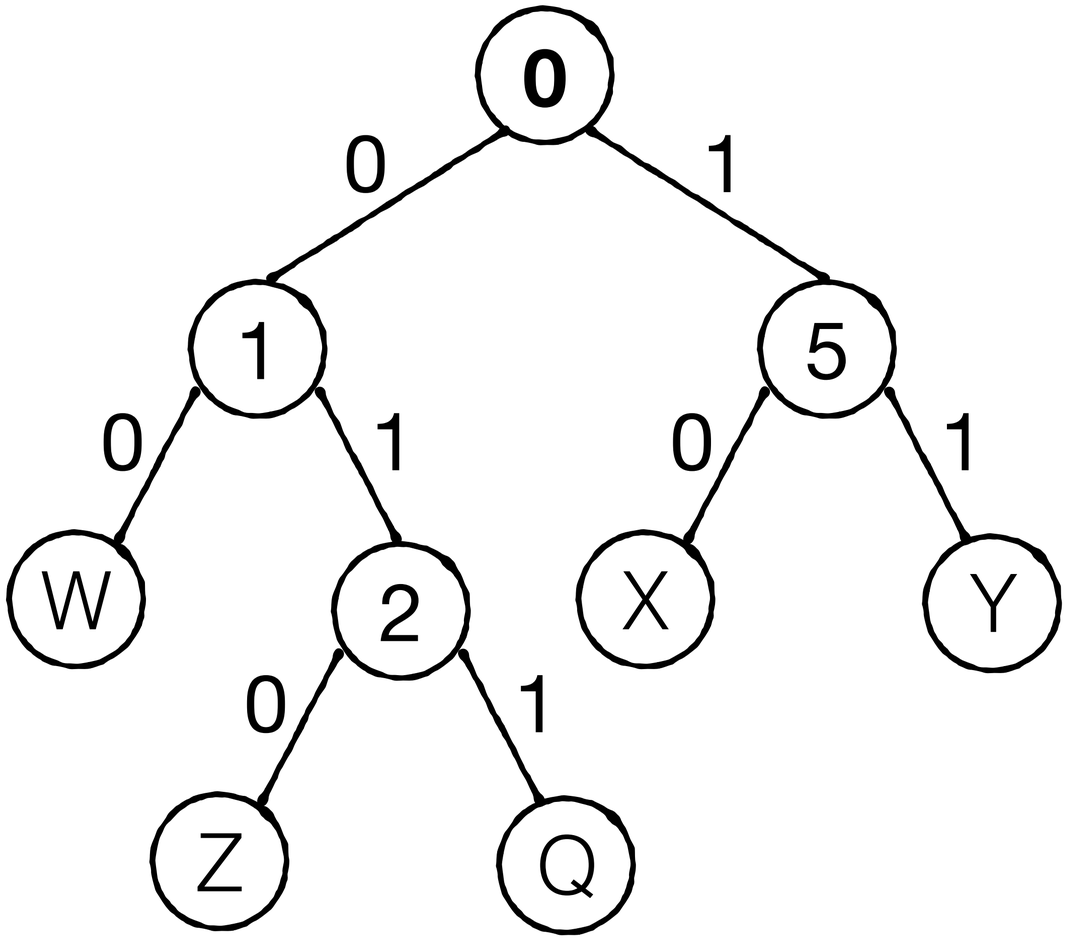
e.g., get(Q) where FP(Q) = 0 1 1 0 0 1

# entries	Topology	Indices	Pointers
111110	11 01 00	0 0 0 11110	W Z Q X Y



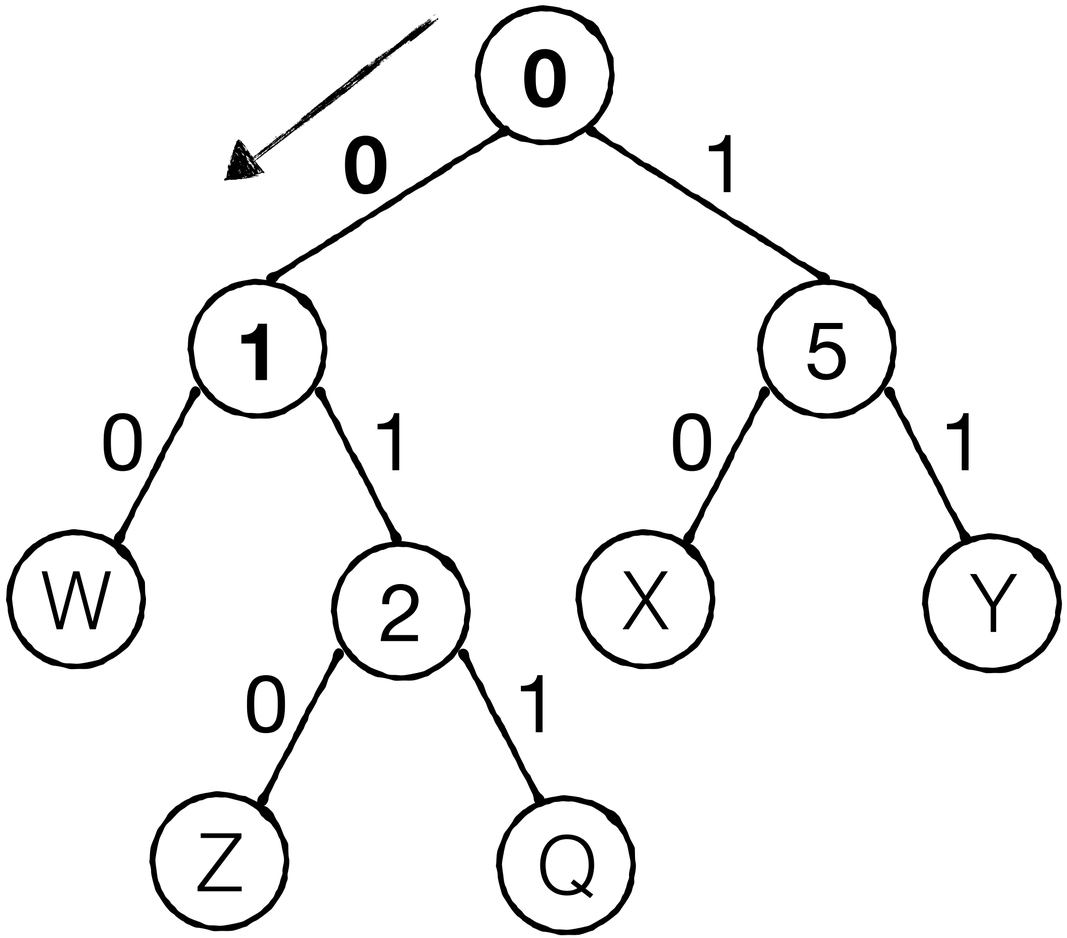
FP(Q) = 0 1 1 0 0 1

# entries	**Topology**	Indices	Pointers
111110	**11** 01 00	**0** 0 0 11110	W Z Q X Y



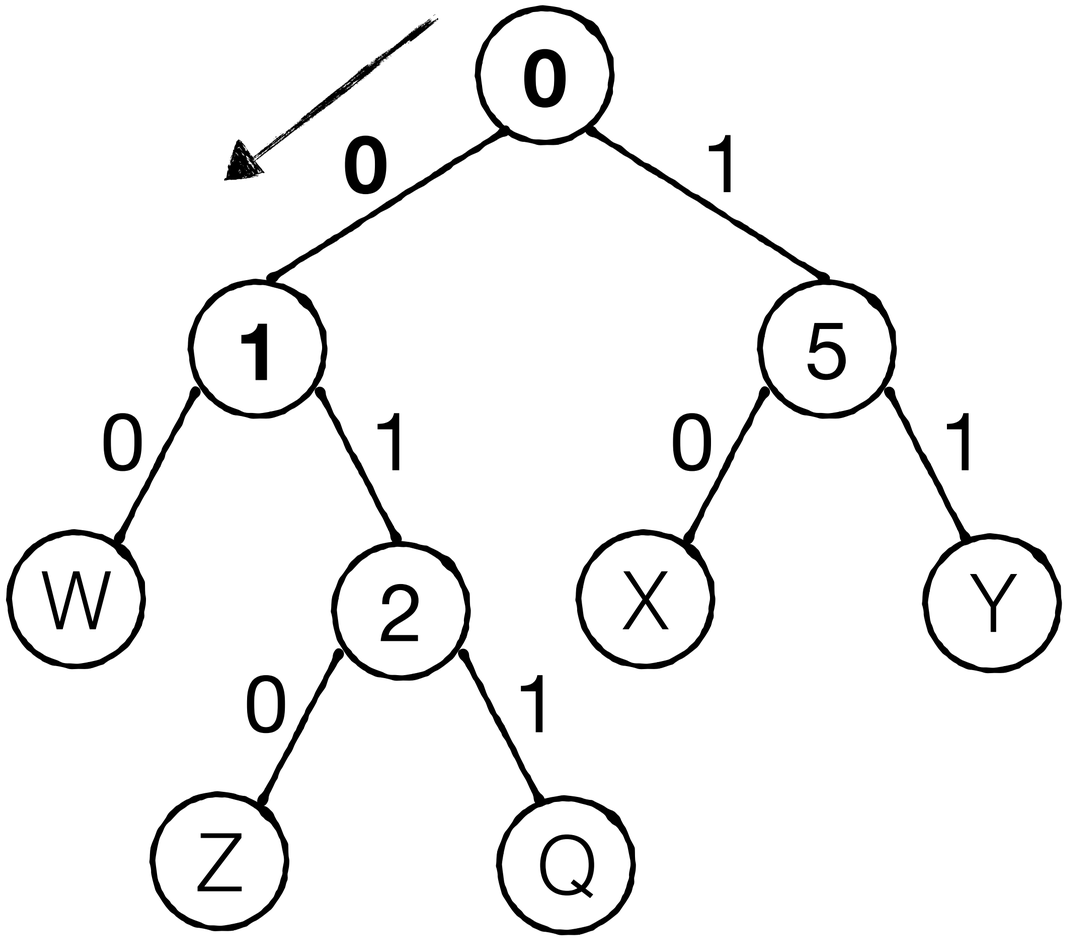
FP(Q) = **0** 1 1 0 0 1

# entries	**Topology**	Indices	Pointers
111110	**11** 01 00	**0** 0 0 11110	W Z Q X Y



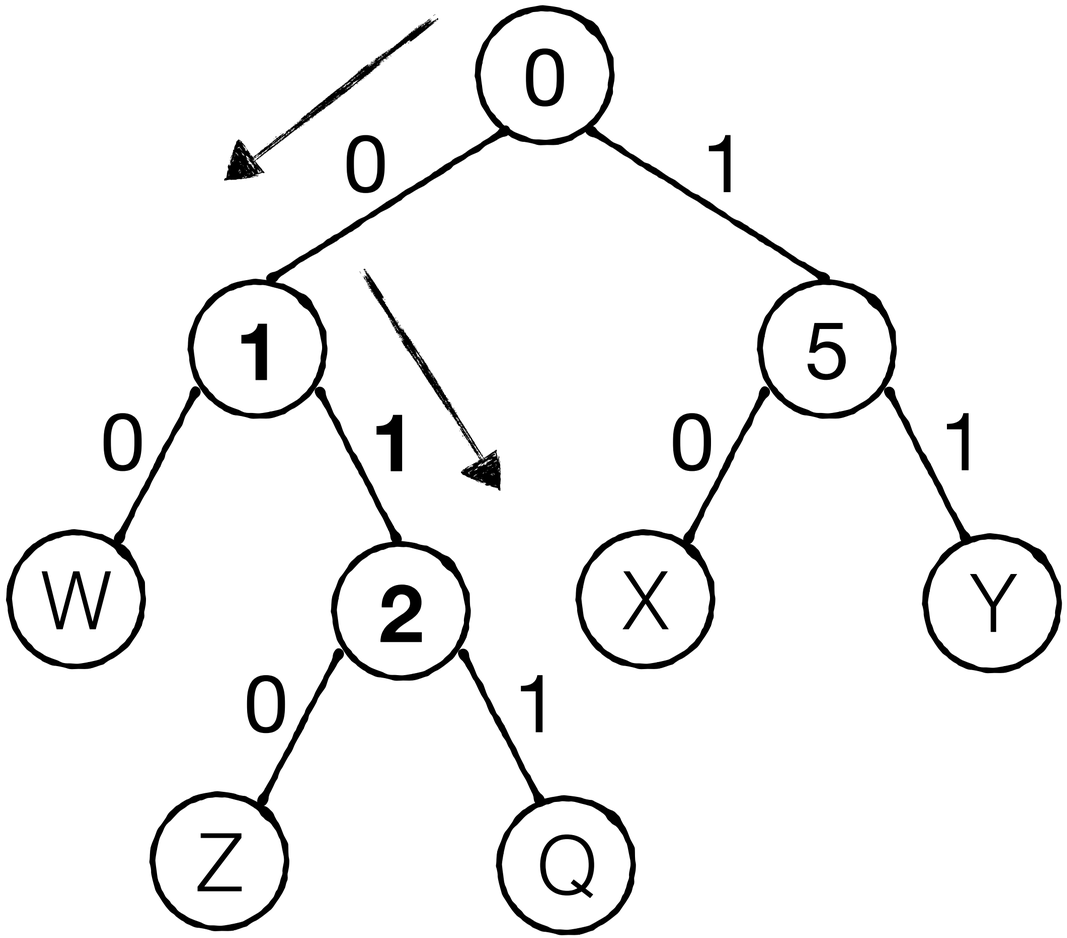
FP(Q) = **0** 1 1 0 0 1

# entries	**Topology**	Indices	Pointers
111110	11 **01** 00	0 **0** 0 11110	W Z Q X Y



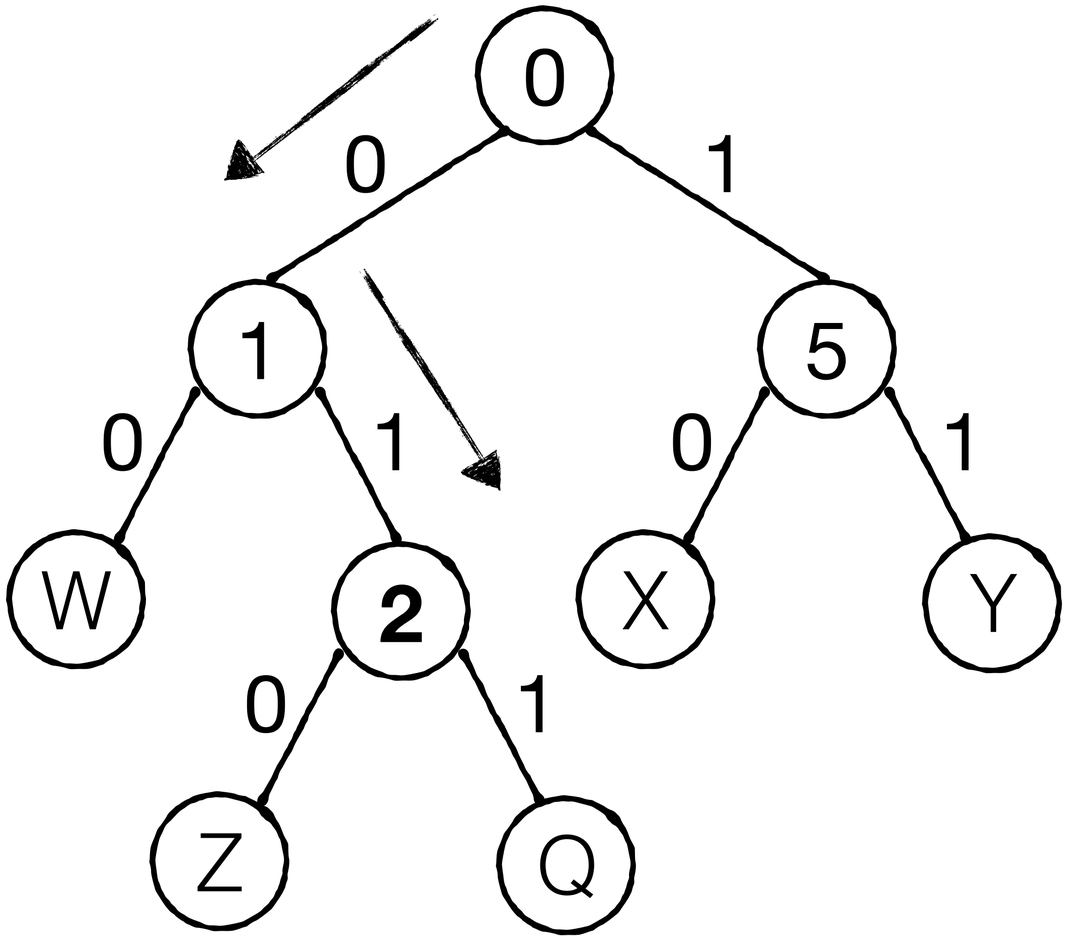
FP(Q) = 0 **1** 1 0 0 1

# entries	**Topology**	Indices	Pointers
111110	11 **01** 00	0 **0** 0 11110	W Z Q X Y



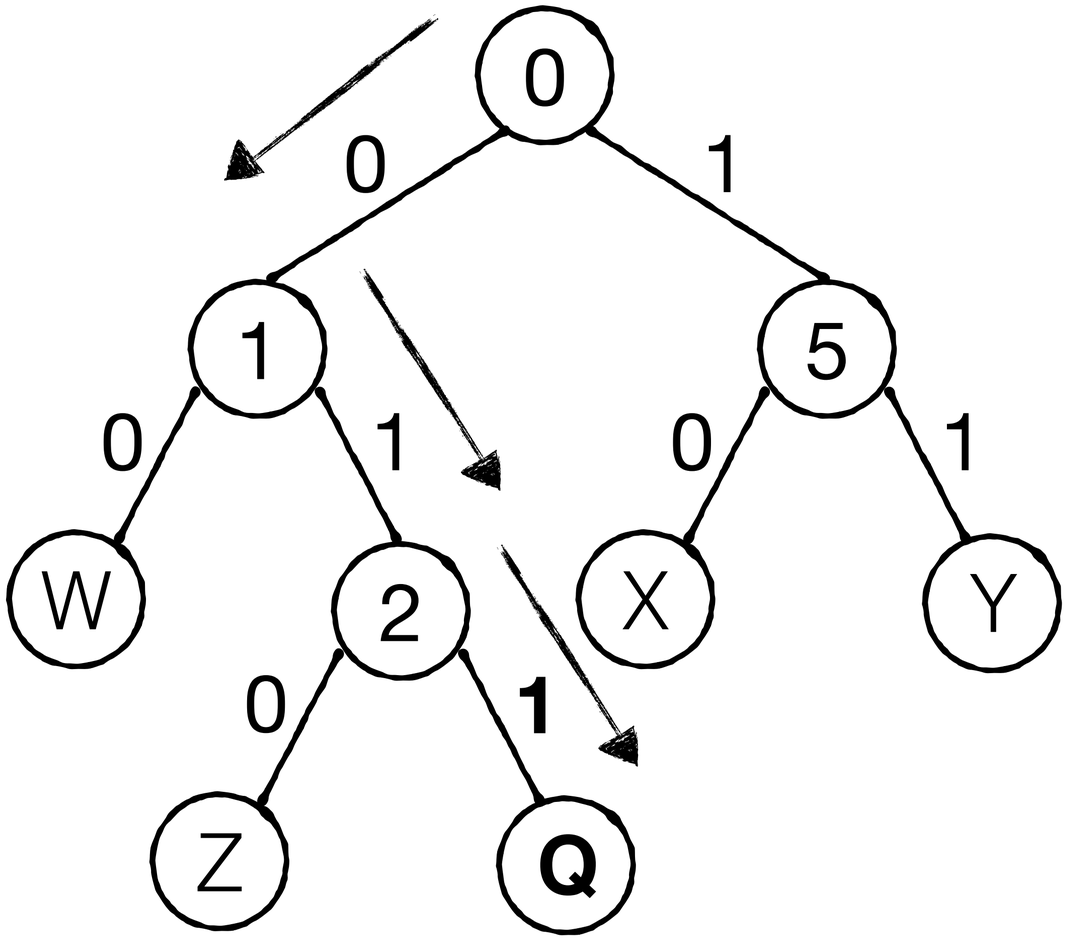
FP(Q) = 0 **1** 1 0 0 1

# entries	**Topology**	Indices	Pointers
111110	11 01 **00**	0 0 **0** 11110	W Z Q X Y



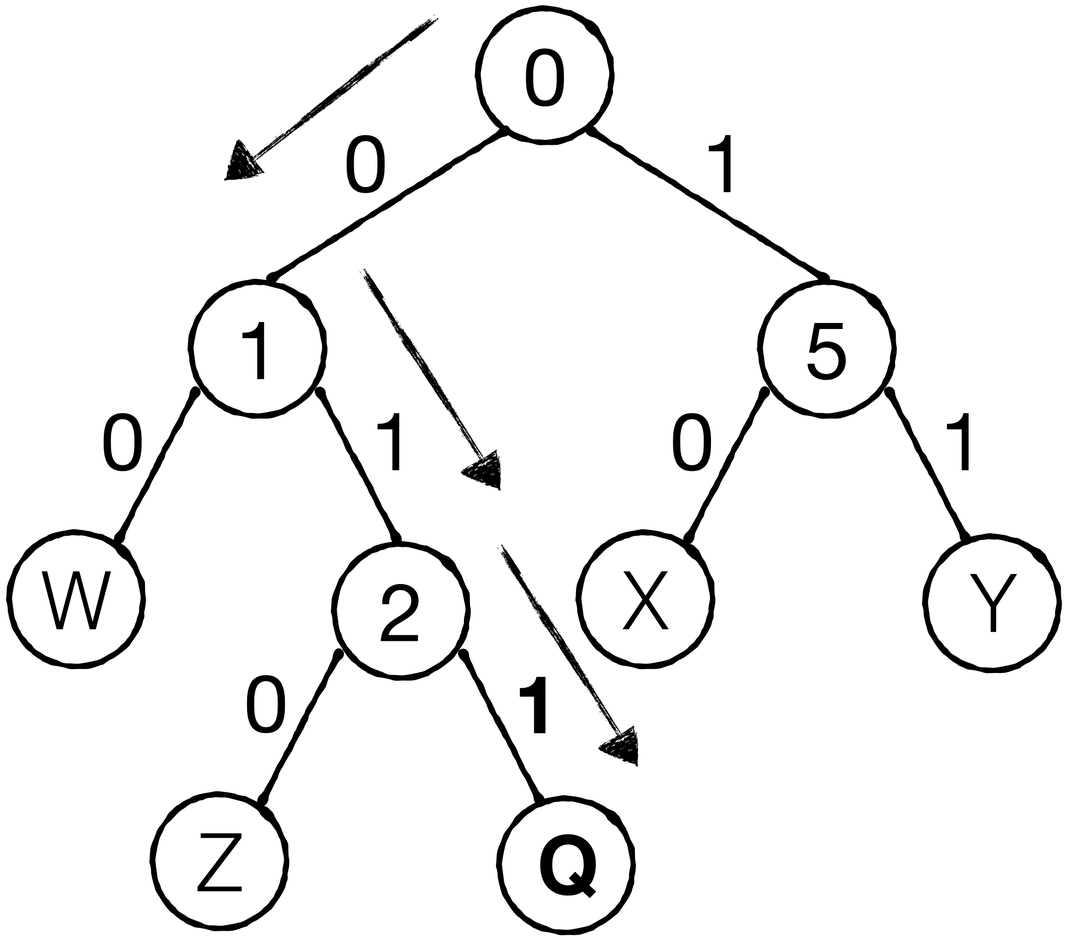
FP(Q) = 0 1 **1** 0 0 1

# entries	**Topology**	Indices	Pointers
111110	11 01 **00**	0 0 **0** 11110	W Z Q X Y

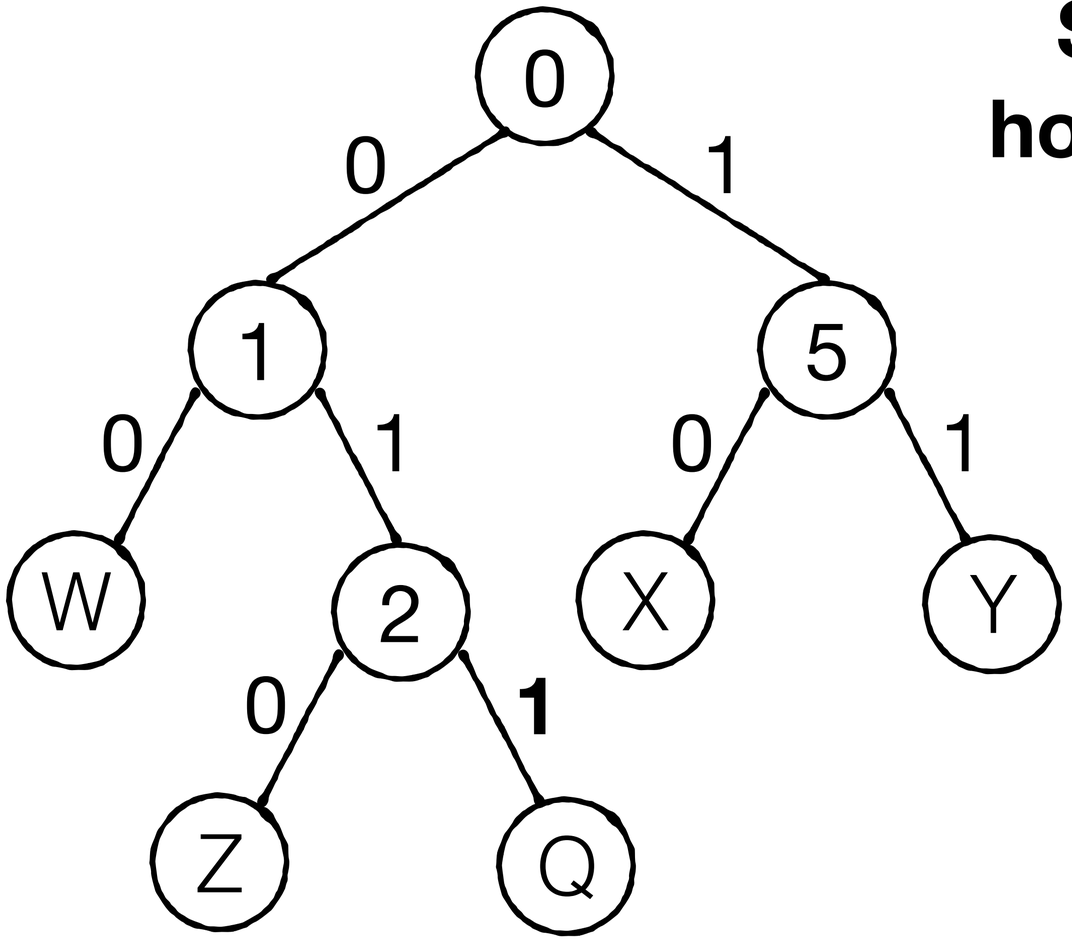


FP(Q) = 0 1 **1** 0 0 1

# entries	**Topology**	Indices	Pointers
111110	11 01 **00**	0 0 0 11110	W Z **Q** X Y

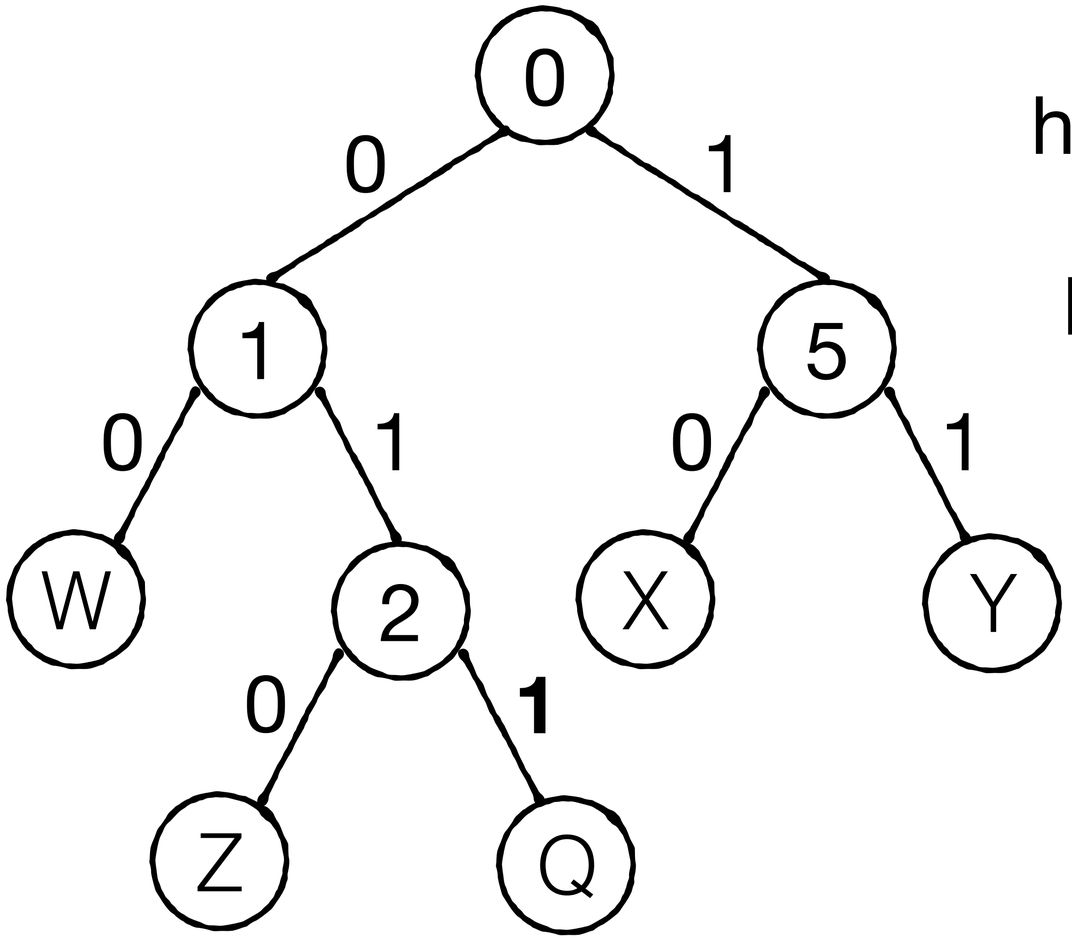


# entries	Topology	Indices	Pointers
111110	11 01 00	0 0 0 11110	W Z Q X Y



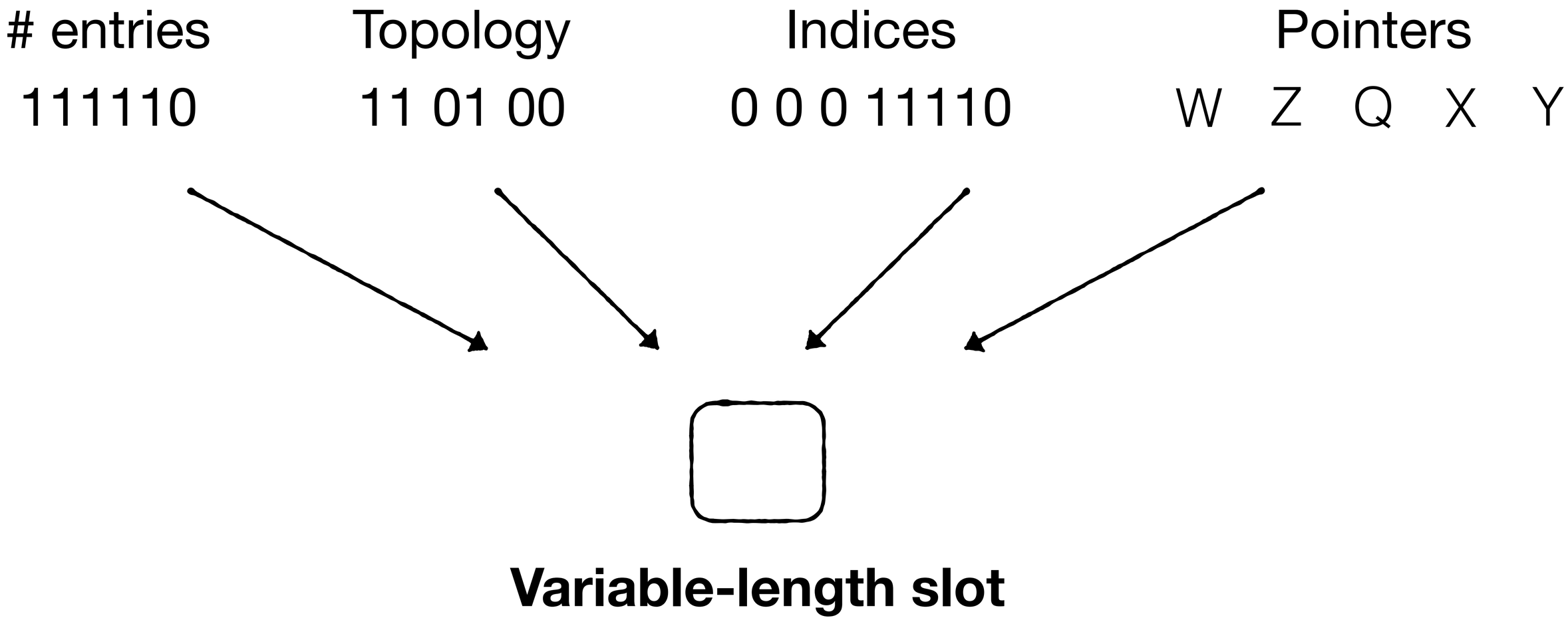
Since this encoding is var-length, how do we store it in the hash table?

# entries	Topology	Indices	Pointers
111110	11 01 00	0 0 0 11110	W Z Q X Y

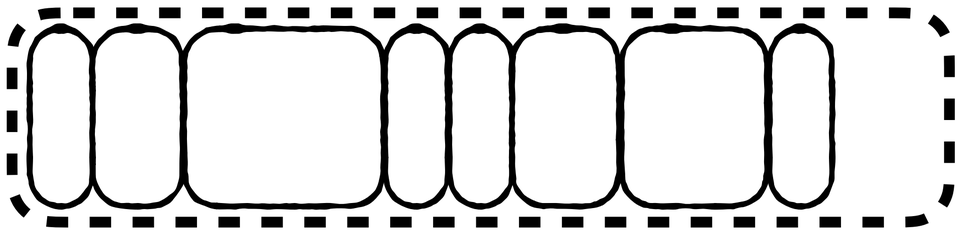


Since this encoding is var-length,
how do we store it in the hash table?

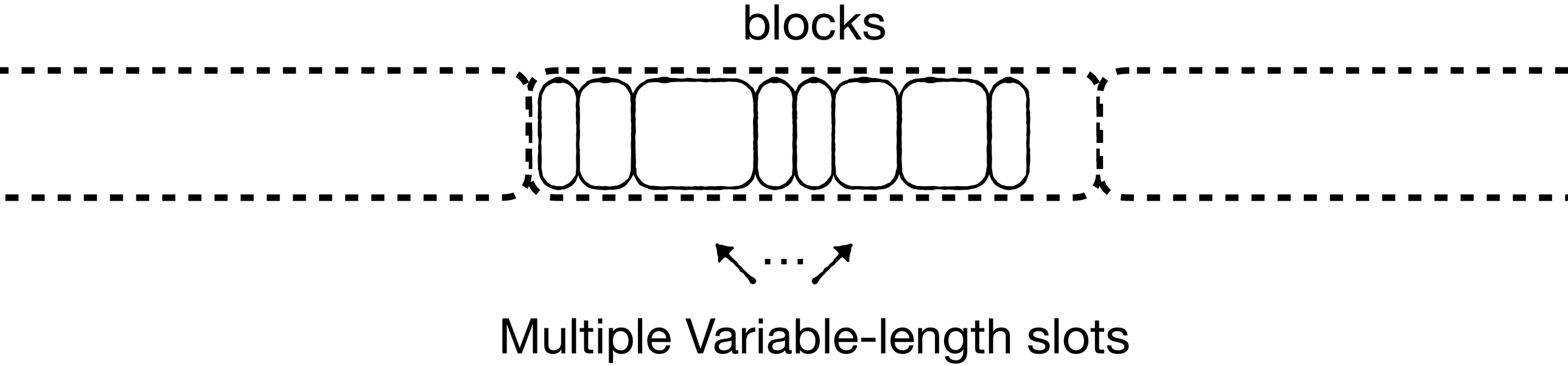
Note: all fields are self-delimiting

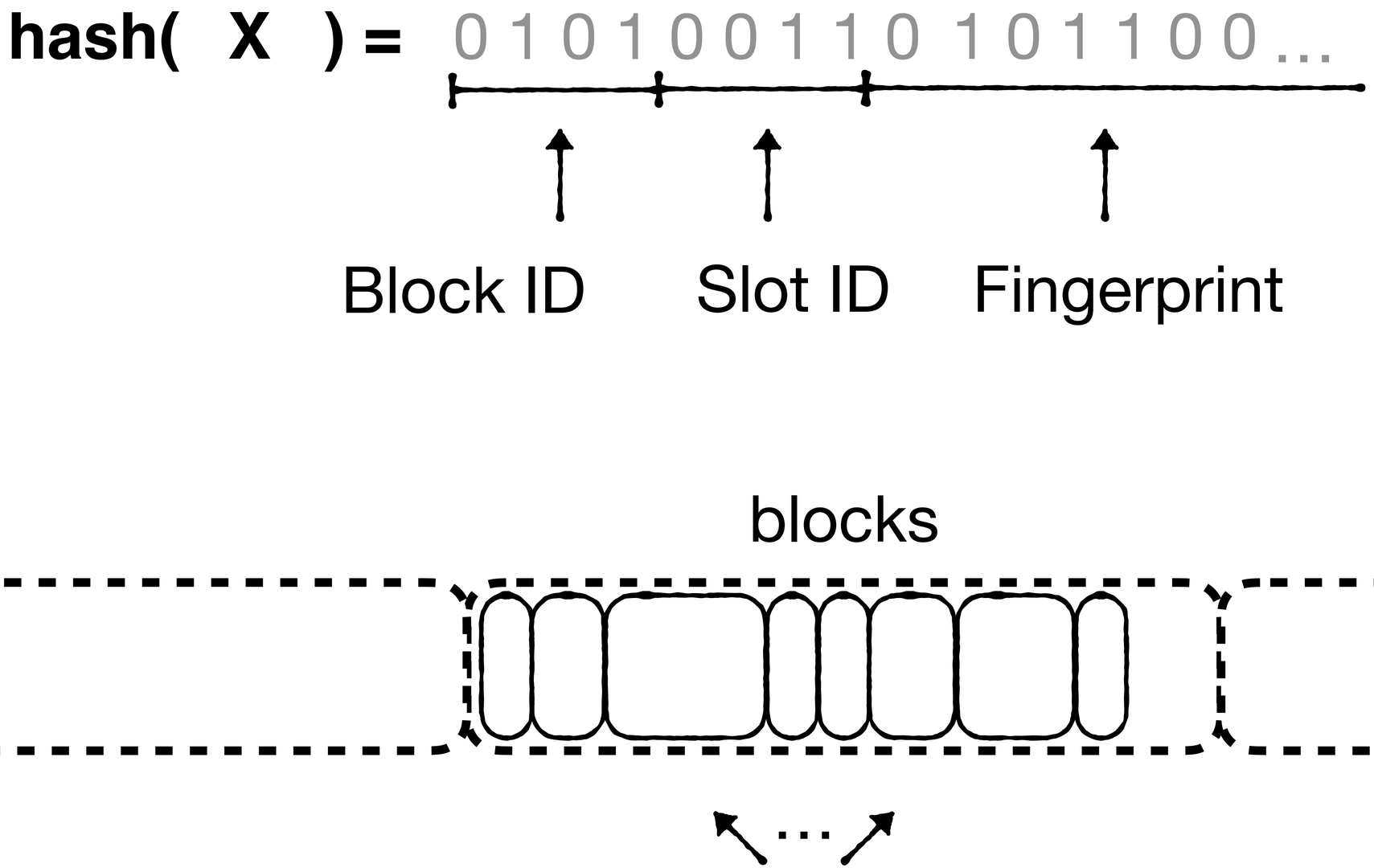

entries
Topology
Indices
Pointers
111110
11 01 00
0 0 0 11110
W Z Q X Y
Variable-length slot

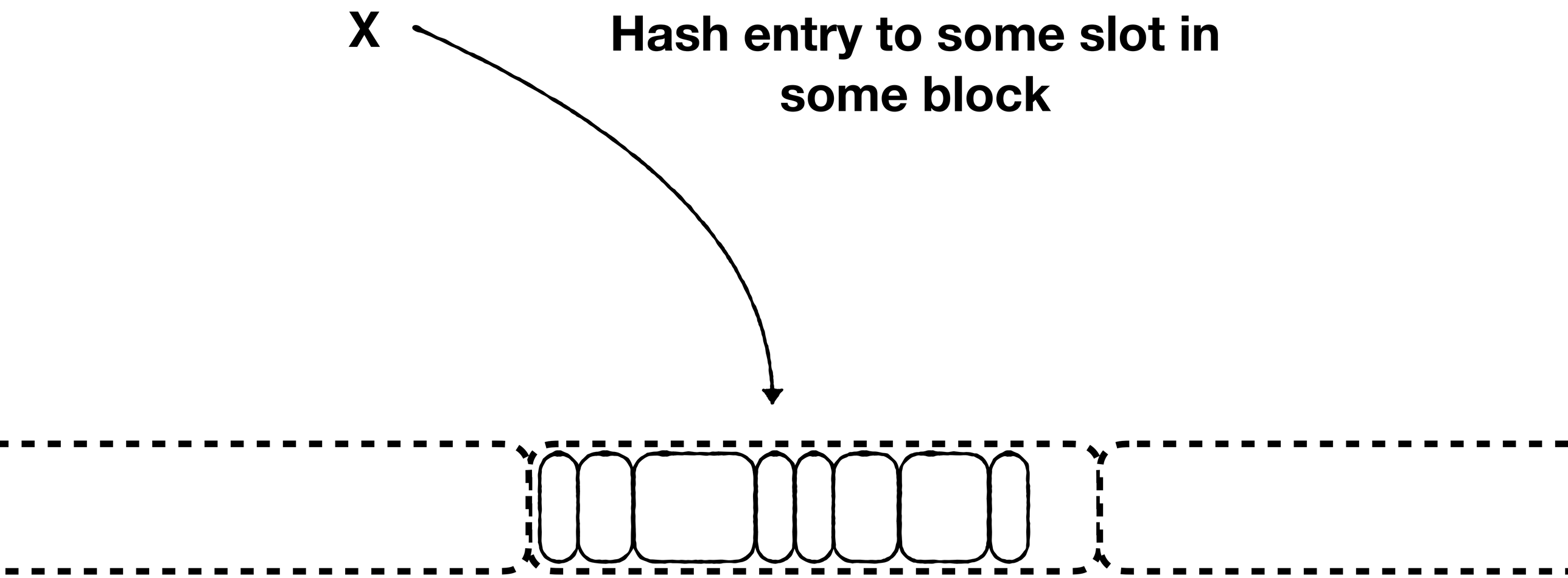
Fixed-sized block

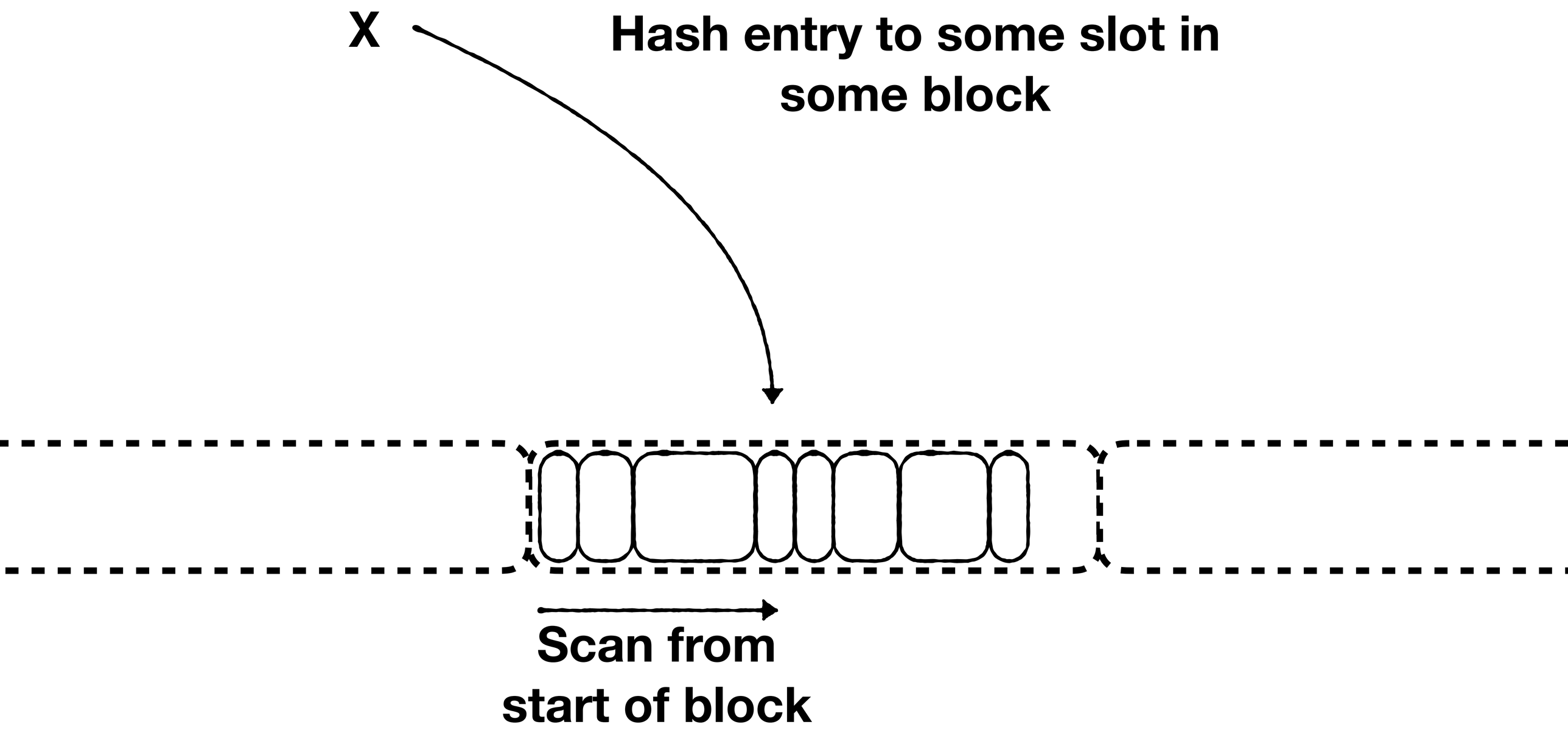


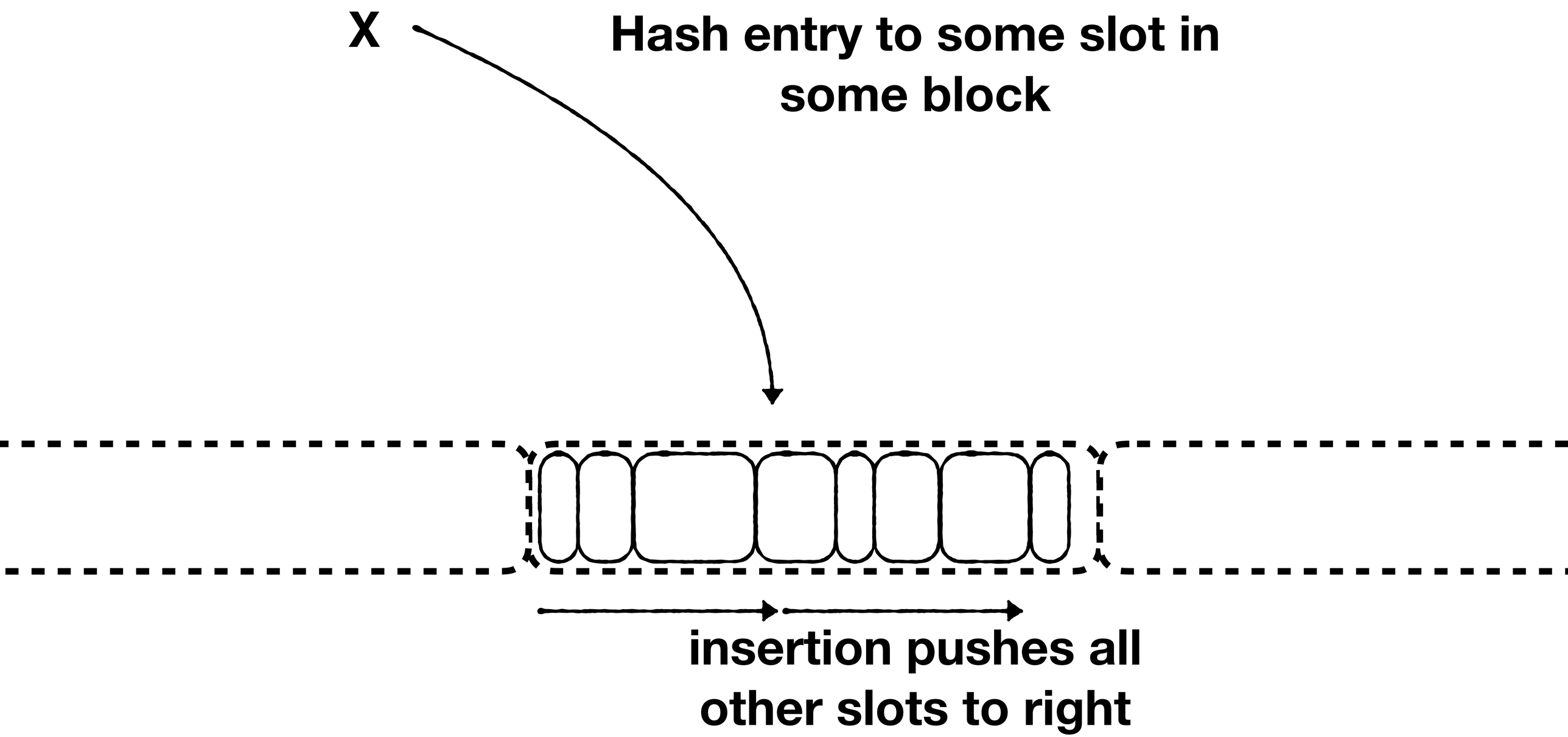
Multiple Variable-length slots

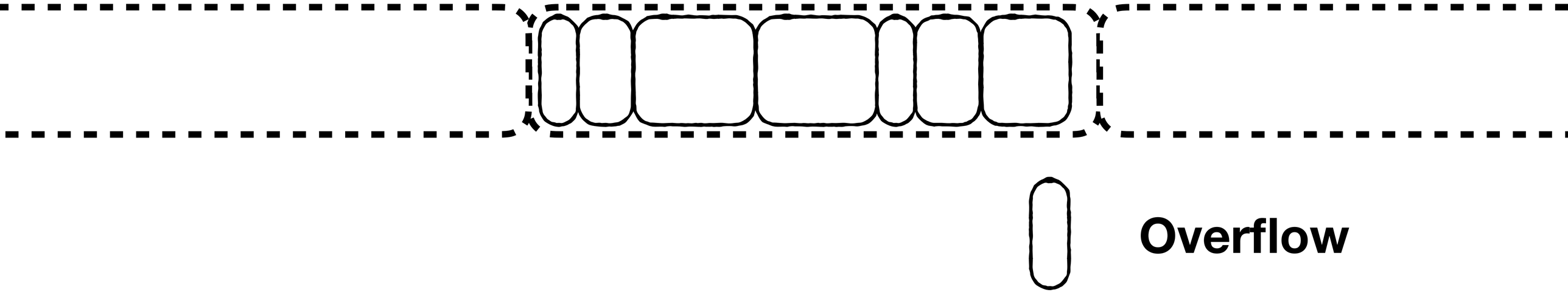

blocks
Multiple Variable-length slots

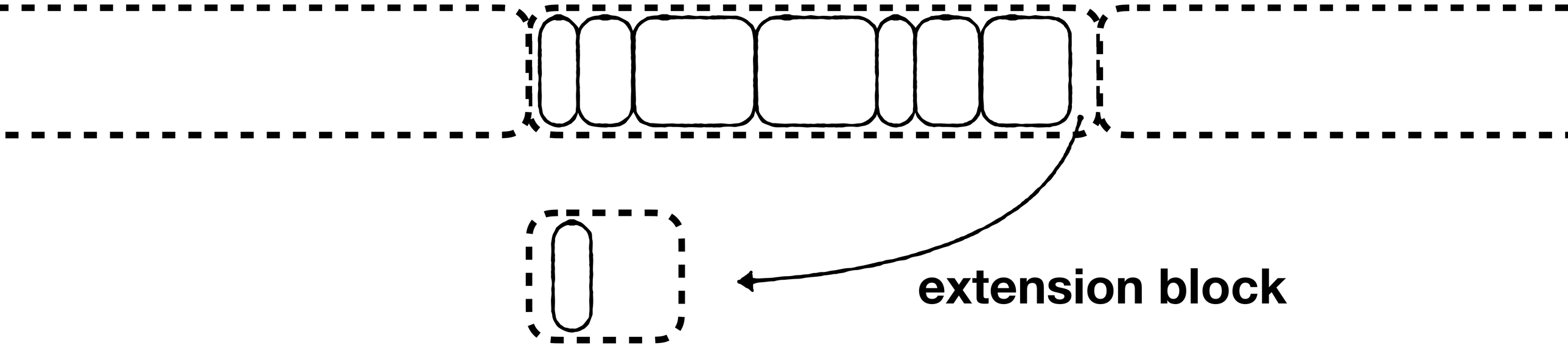

hash(X) = 0 1 0 1 0 0 1 1 0 1 0 1 1 0 0 ...
Block ID
Slot ID
Fingerprint
blocks
Multiple Variable-length slots


X
Hash entry to some slot in some block


X
Hash entry to some slot in some block
Scan from start of block

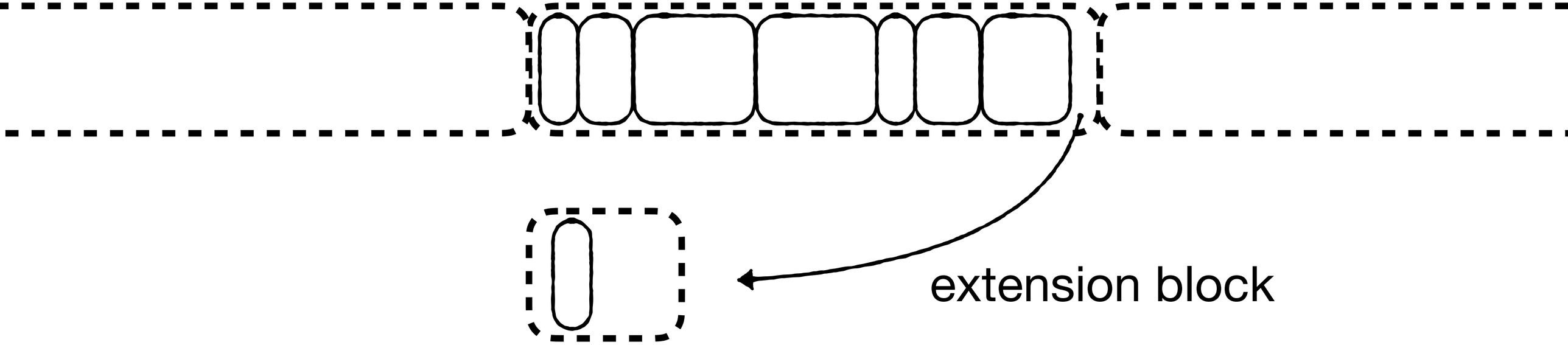

X
Hash entry to some slot in some block
insertion pushes all other slots to right


Overflow


extension block

With more slots per block…

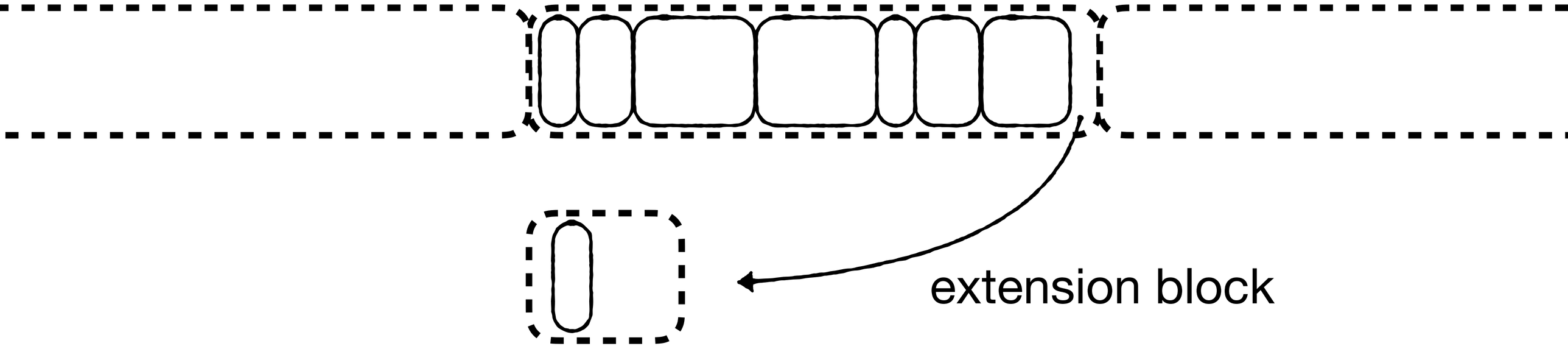
(1) **(2)**



With more slots per block…

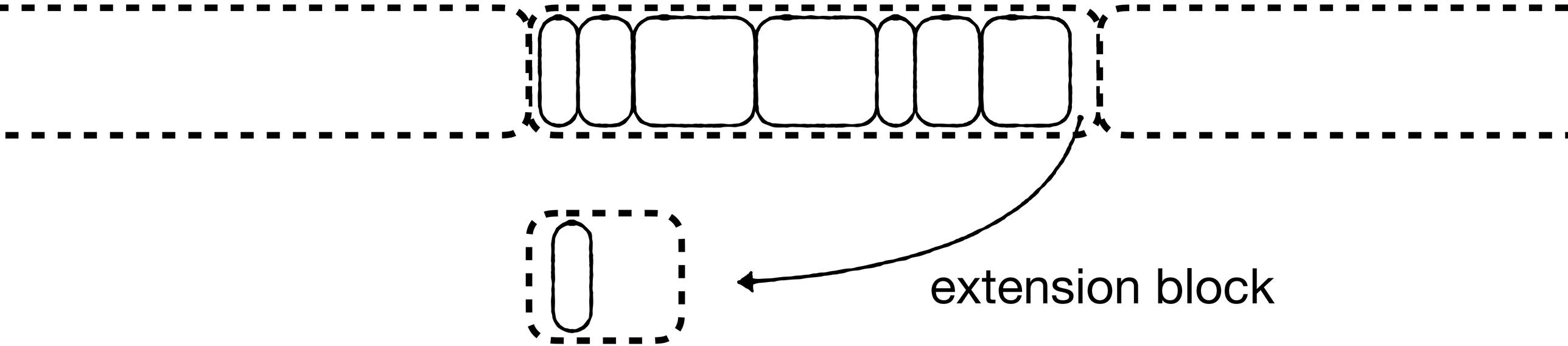
(1) Less size variability
-> fewer overflows

(2) More to traverse
for queries/inserts

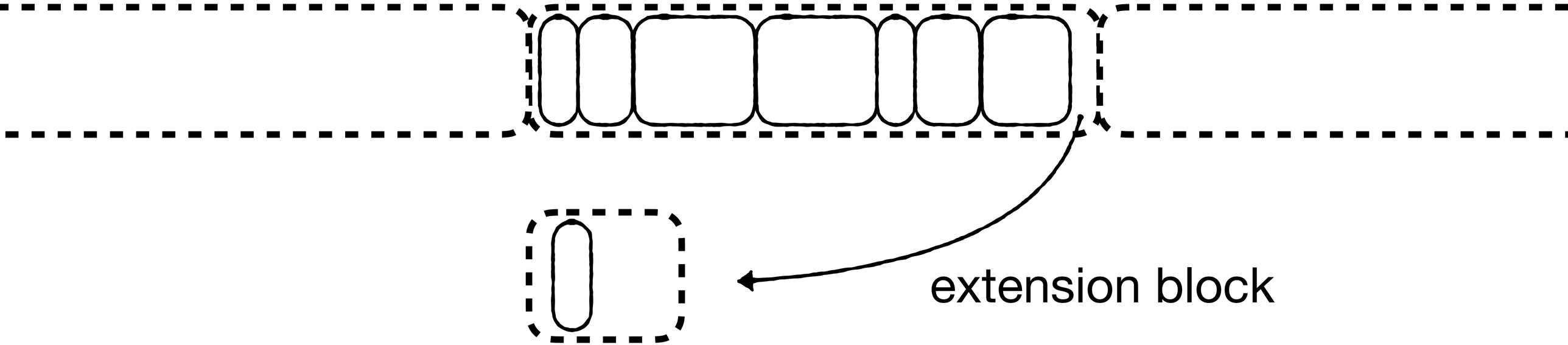


The structure is not expandable and tuned to support on avg. 1 entry per slot

e.g., 64 slots per block



How large is each slot?



How large is each slot?

#entries Topology Indices Pointers

#entries

Slot size X **Encoding** **#Bits**

#entries

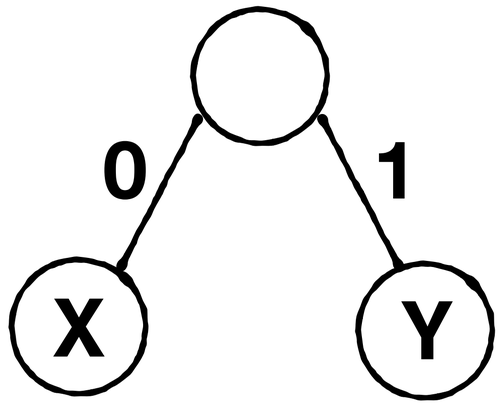
Slot size X	Encoding	#Bits
0	0	1

#entries

Slot size X	Encoding	#Bits
0	0	1
1	10	2

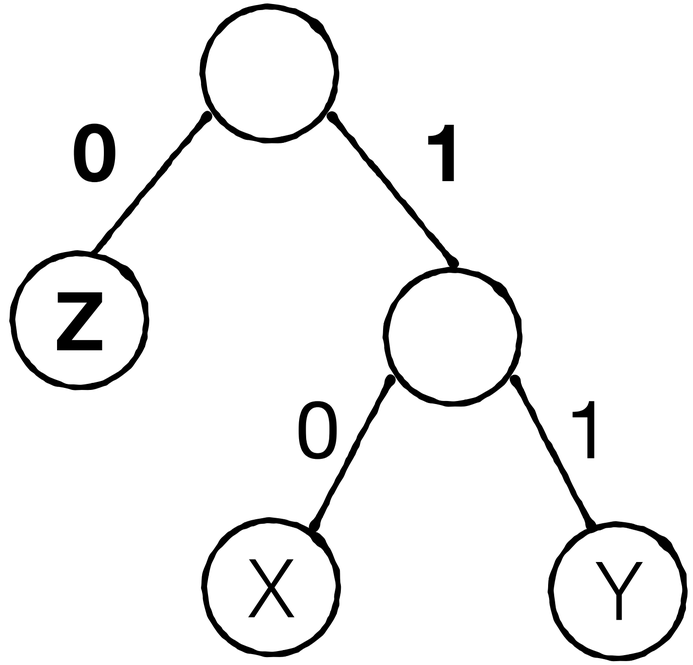
#entries

Slot size X	Encoding	#Bits
0	0	1
1	10	2
2	110	3



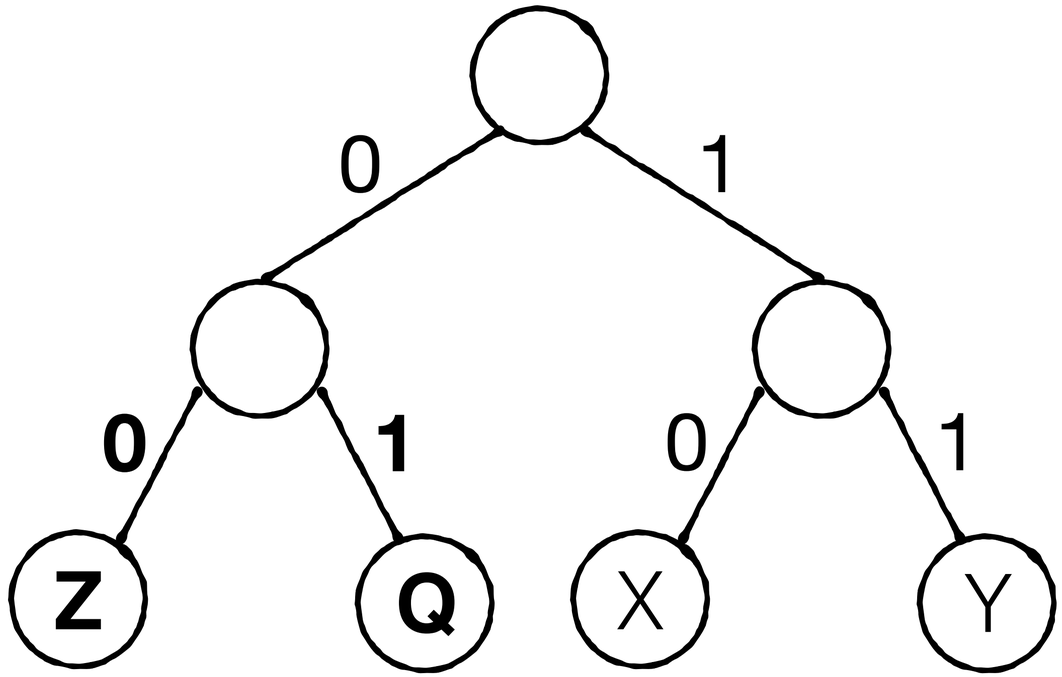
#entries

Slot size X	Encoding	#Bits
0	0	1
1	10	2
2	110	3
3	1110	4



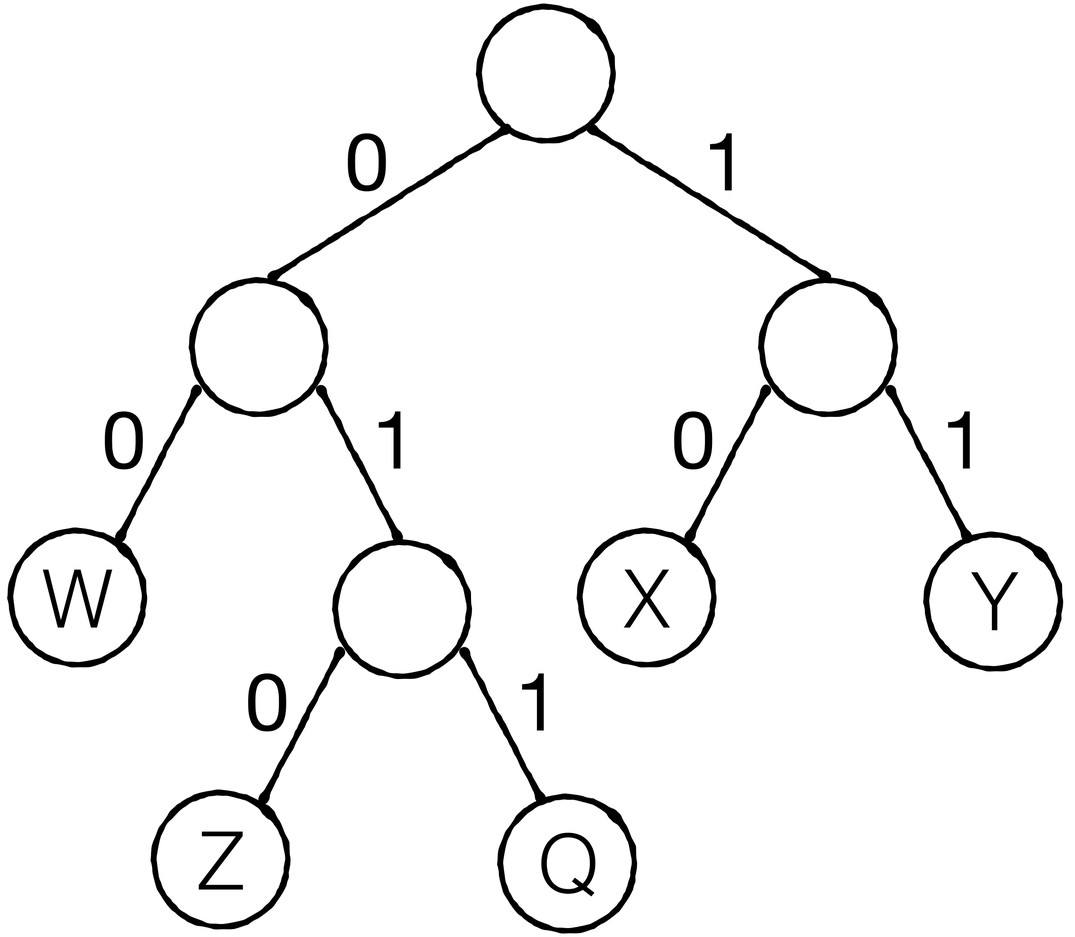
#entries

Slot size X	Encoding	#Bits
0	0	1
1	10	2
2	110	3
3	1110	4
4	11110	5



#entries

Slot size X	Encoding	#Bits
0	0	1
1	10	2
2	110	3
3	1110	4
4	11110	5
5	111110	6



How large is each slot?

#entries **Topology** Indices Pointers

Topology

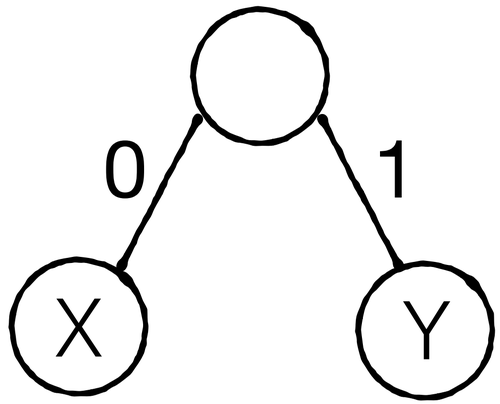
Slot size X	Encoding	#Bits
0	-	0

Topology

Slot size X	Encoding	#Bits
0	-	0
1	-	0

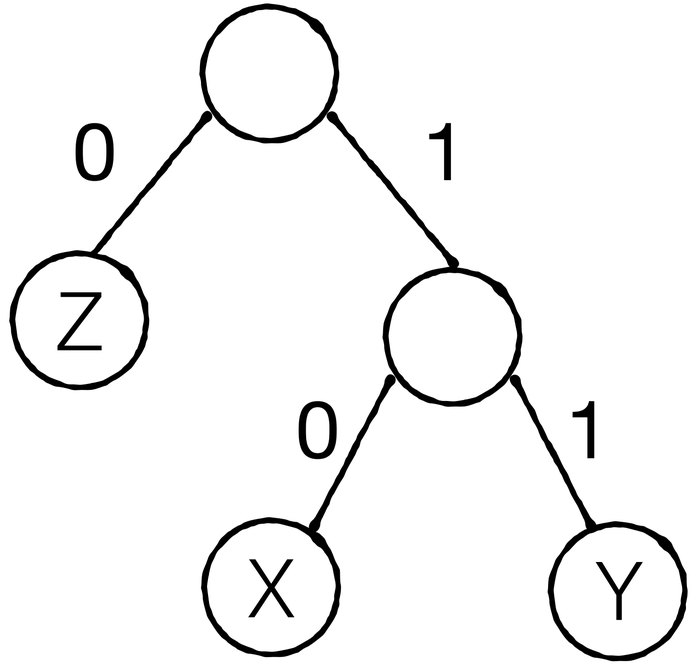
Topology

Slot size X	Encoding	#Bits
0	-	0
1	-	0
2	~~00~~	0



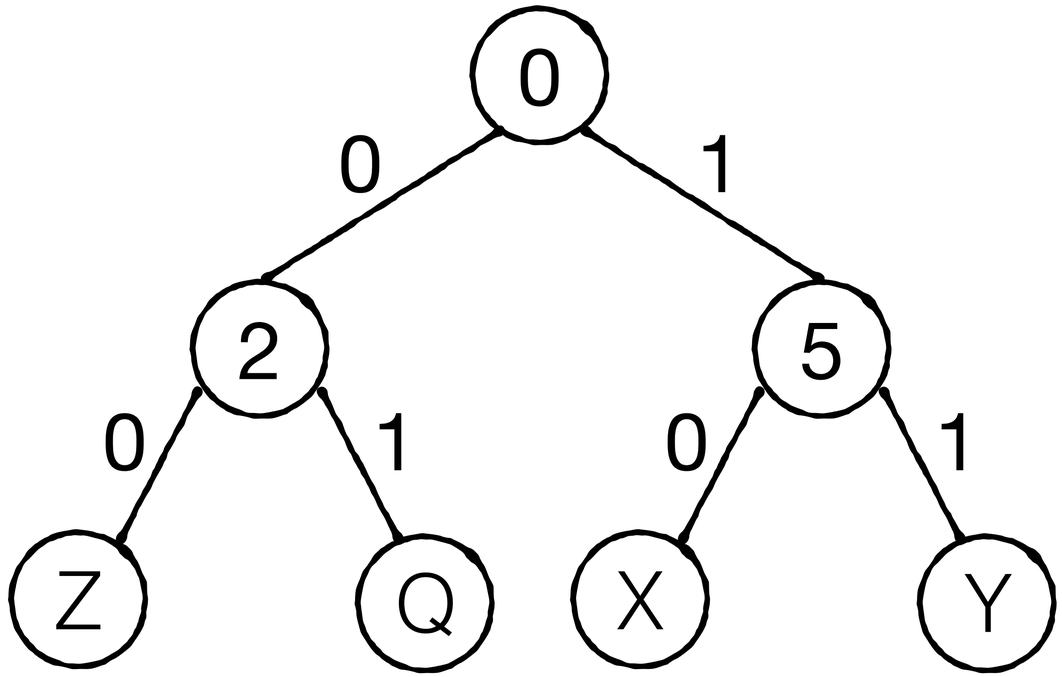
Topology

Slot size X	Encoding	#Bits
0	-	0
1	-	0
2	~~00~~	0
3	01~~00~~	2



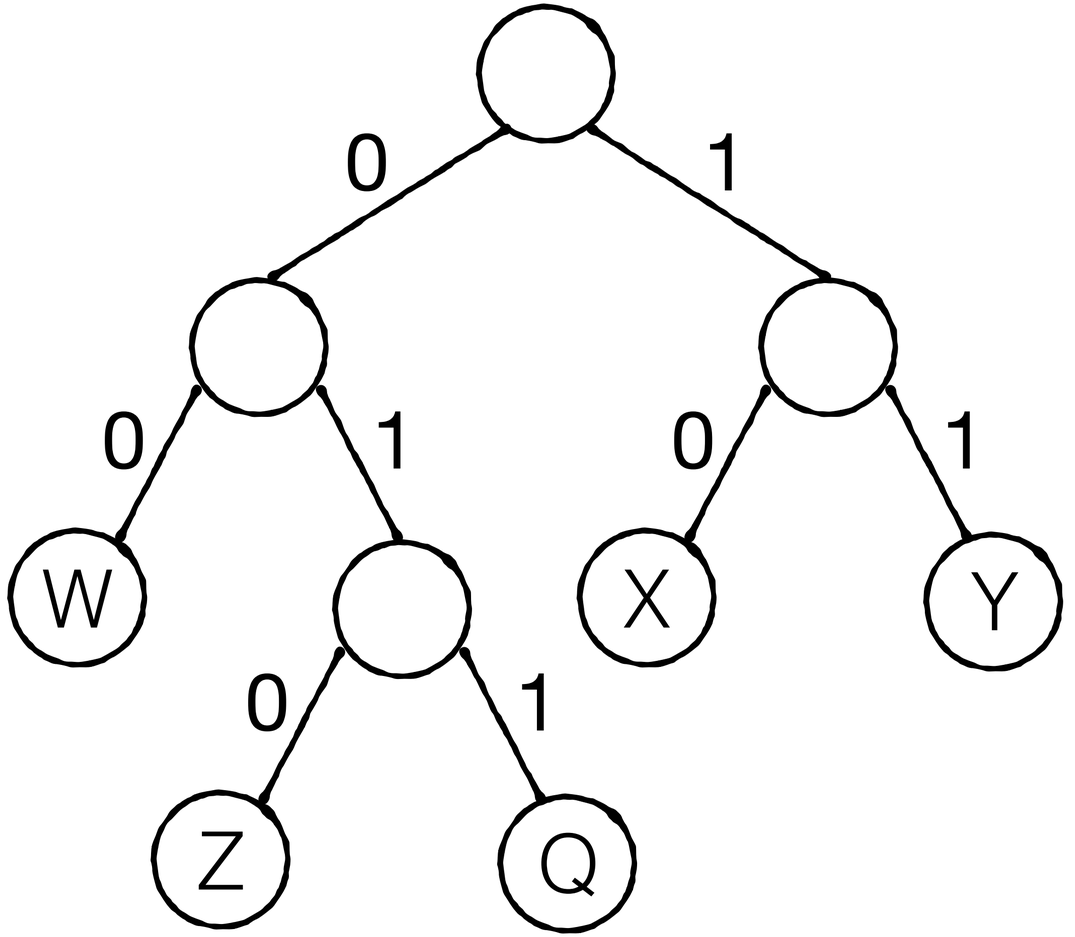
Topology

Slot size X	Encoding	#Bits
0	-	0
1	-	0
2	~~00~~	0
3	01~~00~~	2
4	11 00~~00~~	4



Topology

Slot size X	Encoding	#Bits
0	-	0
1	-	0
2	~~00~~	0
3	01~~00~~	2
4	11 00~~00~~	4
5	11 01 00~~00~~	6



How large is each slot?

#entries Topology **Indices** Pointers

Indices

Slot size X	#Bits

Indices

Slot size X	#Bits
0	0

Indices

Slot size X	#Bits
0	0
1	0

Indices

Slot size X	#Bits
0	0
1	0
2	**?**

hash(X) = **0** 1 0 1 0 …

diff with prob 0.5

Indices

Slot size X	#Bits
0	0
1	0
2	**?**

hash(X) = 0 **1** 0 1 0 …

diff with prob 0.5

Indices

Slot size X	#Bits
0	0
1	0
2	**?**

hash(X) = 0 1 **0** 1 0 …

diff with prob 0.5

Indices

Slot size X	#Bits
0	0
1	0
2	**?**

hash(X) = 0 1 **0** 1 0 ...

diff with prob 0.5

First diff bit occurs after 2 bits in expectation

Indices

Slot size X	#Bits
0	0
1	0
2	**?**

hash(X) = 0 1 **0** 1 0 ...

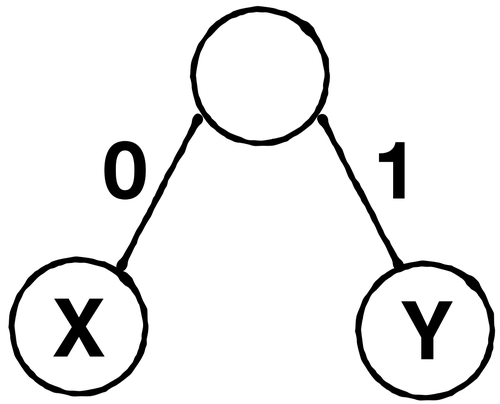
diff with prob 0.5

First diff bit occurs after
2 bits in expectation

Avg. of geometric dist. with prob 0.5

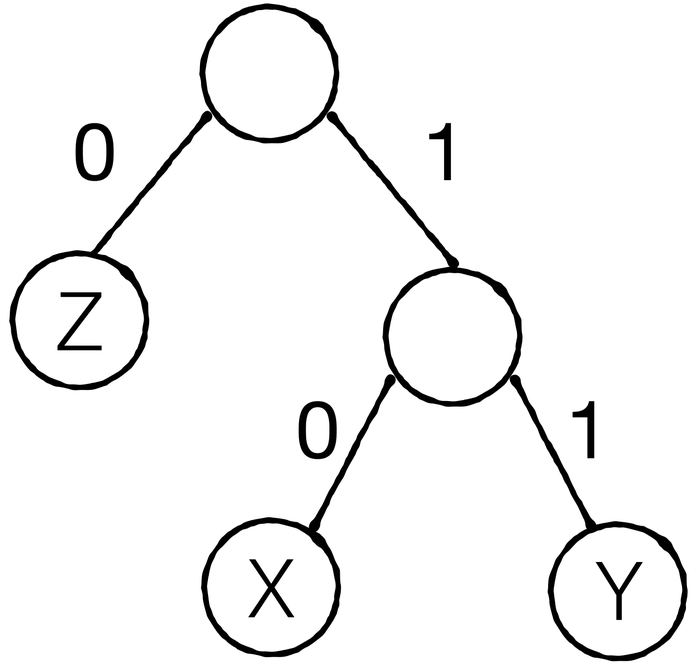
Indices

Slot size X	Avg. #Bits
0	0
1	0
2	**2**



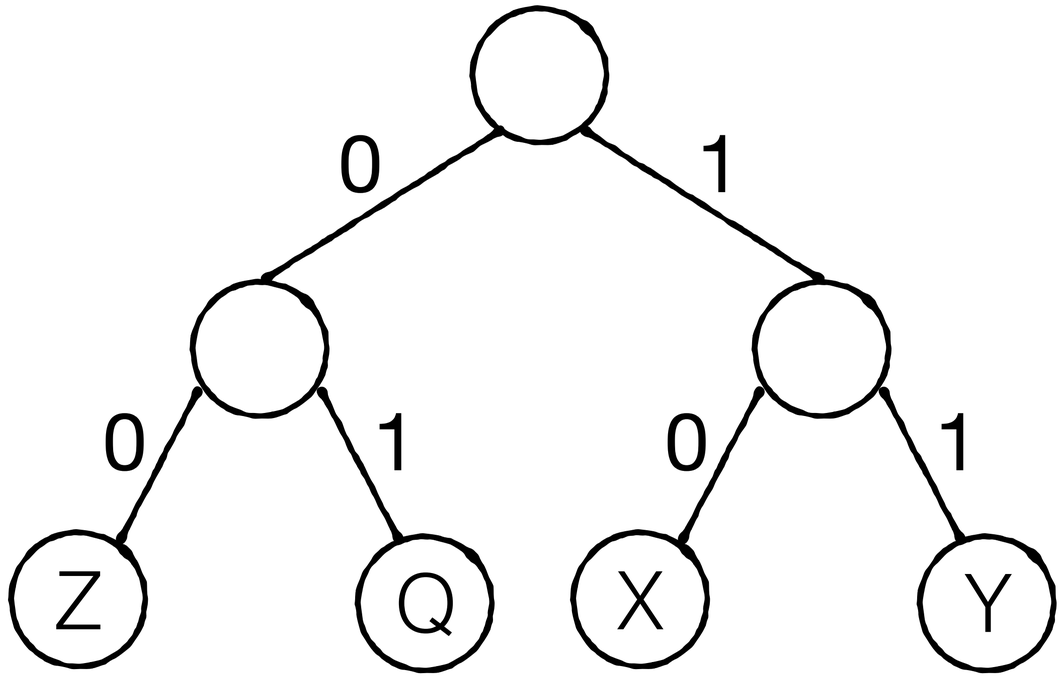
Indices

Slot size X	Avg. #Bits
0	0
1	0
2	2
3	4



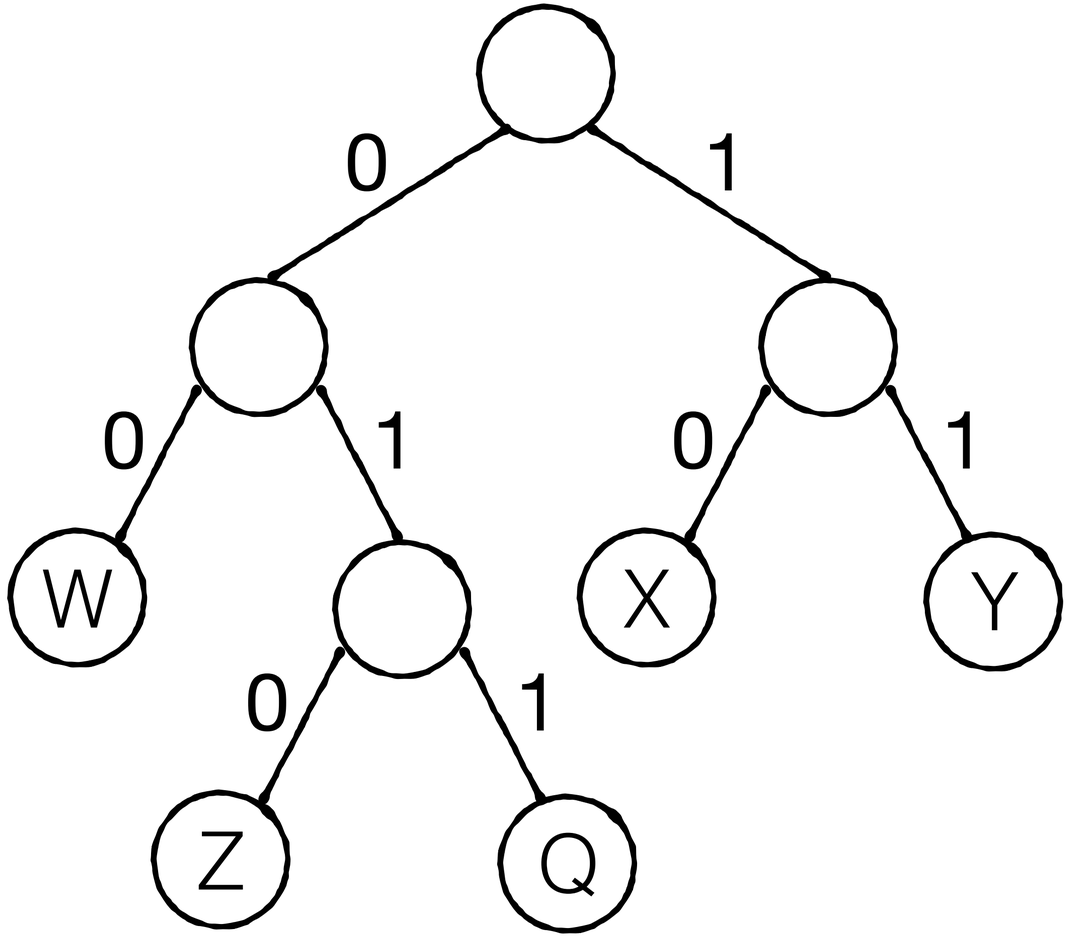
Indices

Slot size X	Avg. #Bits
0	0
1	0
2	2
3	4
4	6



Indices

Slot size X	Avg. #Bits
0	0
1	0
2	2
3	4
4	6
5	8



How large is each slot?

#entries Topology Indices

Average

$$\left(\#\text{entries}(i) \quad + \quad \text{Topology}(i) \quad + \quad \text{Indices}(i) \right)$$

Average

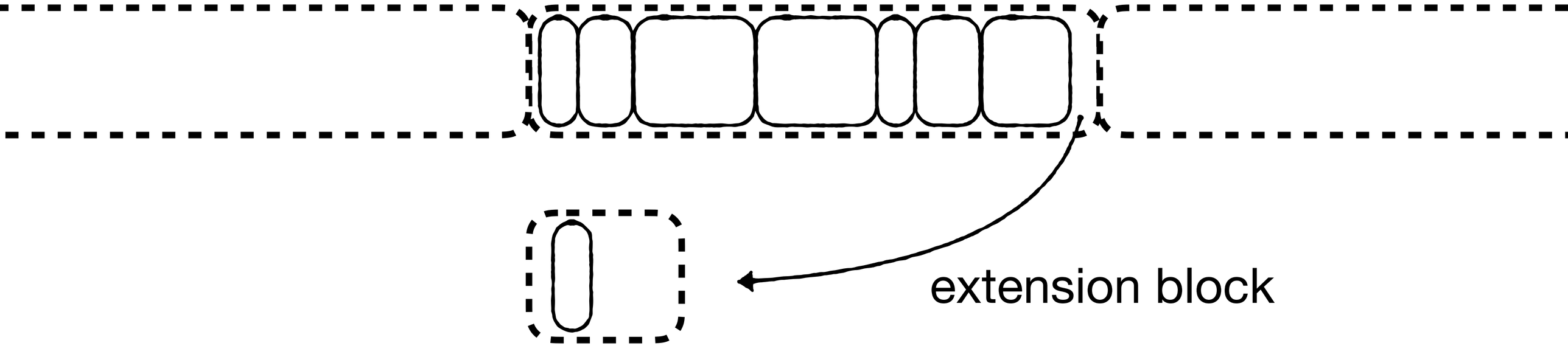
$$\sum_{i=0}^{\infty} \text{Poisson}(1, i) \cdot \left(\#\text{entries}(i) + \text{Topology}(i) + \text{Indices}(i) \right)$$

Average

$$\sum_{i=0}^{\infty} \text{Poisson}(1, i) \cdot \Big(\#\text{entries}(i) + \text{Topology}(i) + \text{Indices}(i) \Big) \approx \textbf{3 bits}$$

 # bits in block containing X slots should be at least

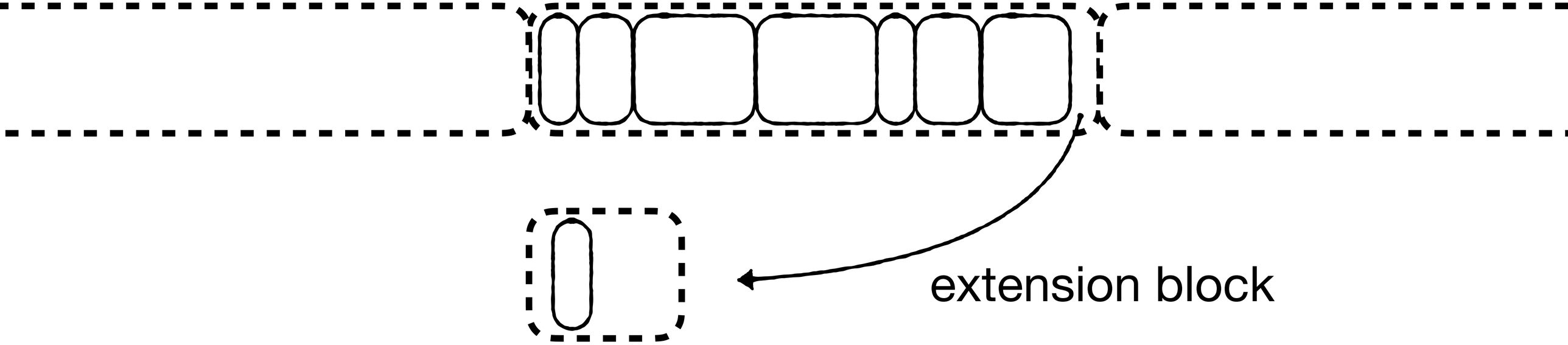
$$X \cdot (3 + \text{pointer\_size})$$



bits in block containing X slots should be at least

$$X \cdot (3 + \text{pointer\_size} + \mathbf{1})$$

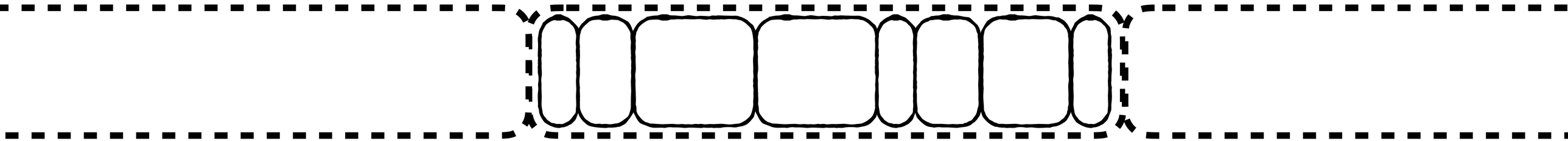
Reduce overflows



bits in block containing X slots should be at least

$$X \cdot (3 + \text{pointer\_size} + \mathbf{1})$$

Reduce overflows



Summary

	Fingerprinting	Delta Hash Table
Memory	$\approx e \cdot N$	$\approx 4 \cdot N$
Load factor	100%	$\approx$ 90%
Avg. query	O(1)	O(1)
Insertions	N/A	O(1)

	Fingerprinting	Delta Hash Table
Memory	$\approx e \cdot N$	$\approx 4 \cdot N$
Load factor	100%	$\approx$ 90%
Avg. query	O(1)	O(1)
Insertions	N/A	O(1)
Construction	O(N)	O(N)

And now, a student presentation