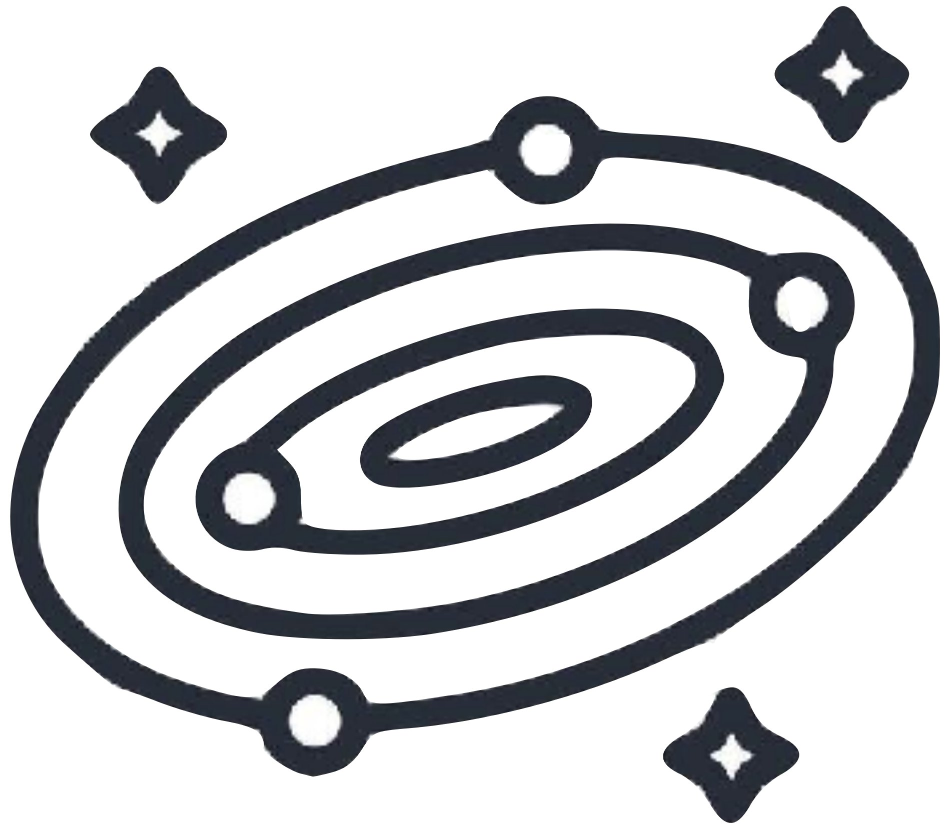
Storing Sorted Strings

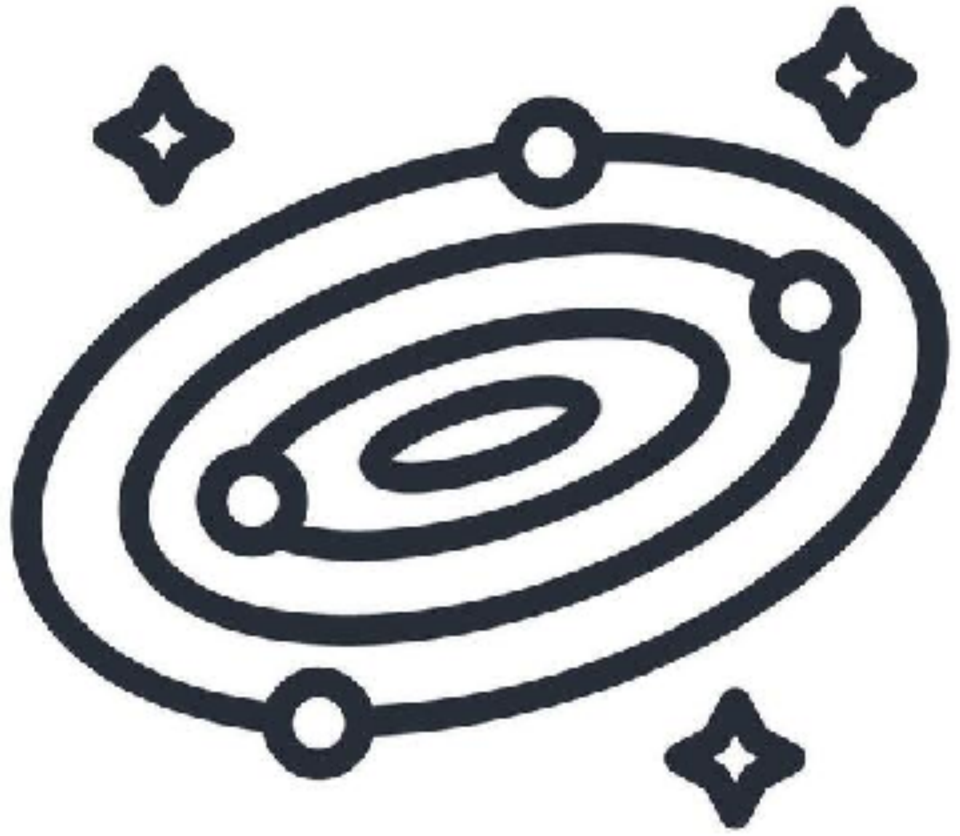
(RocksDB, Delta Compression, LZ77, Huffman Coding, etc)

Niv Dayan - CSC2525 Research Topics in Database Management

Last Week - Succinct Sets

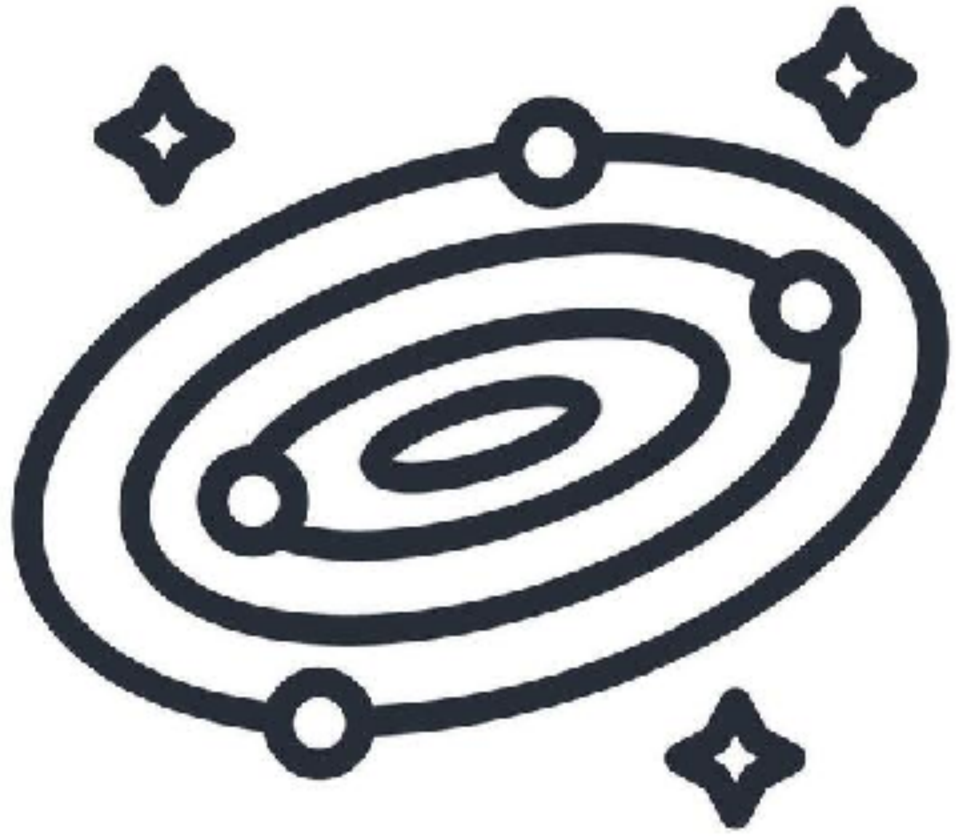


Last Week - Succinct Sets

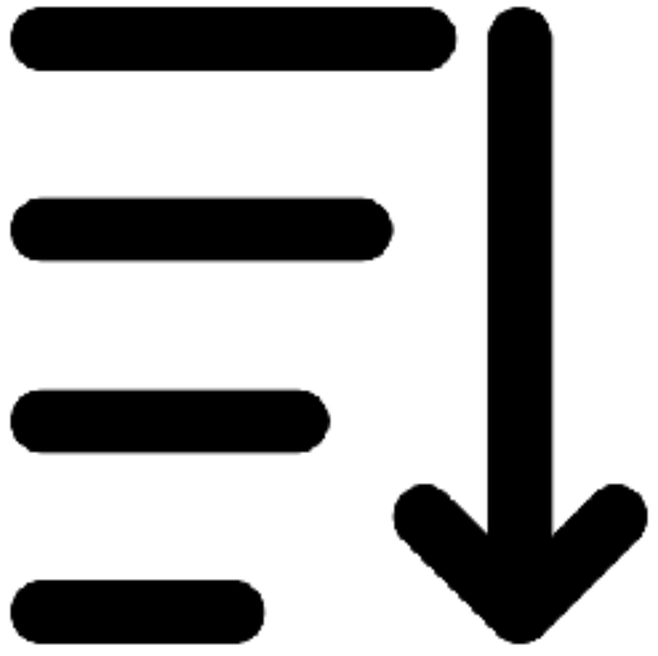


N integers out of Universe U

Last Week - Succinct Sets



N integers out
of Universe U



**Sorting Enables
Compression**

Sorting Enables Compression

Delta Compression w. Golomb Codes

Elias Fano

Sorting Enables Compression

Delta Compression
w. Golomb Codes

Encode distances between adjacent sorted integers

Elias Fano

Sorting Enables Compression

Delta Compression
w. Golomb Codes

Encode distances between adjacent sorted integers

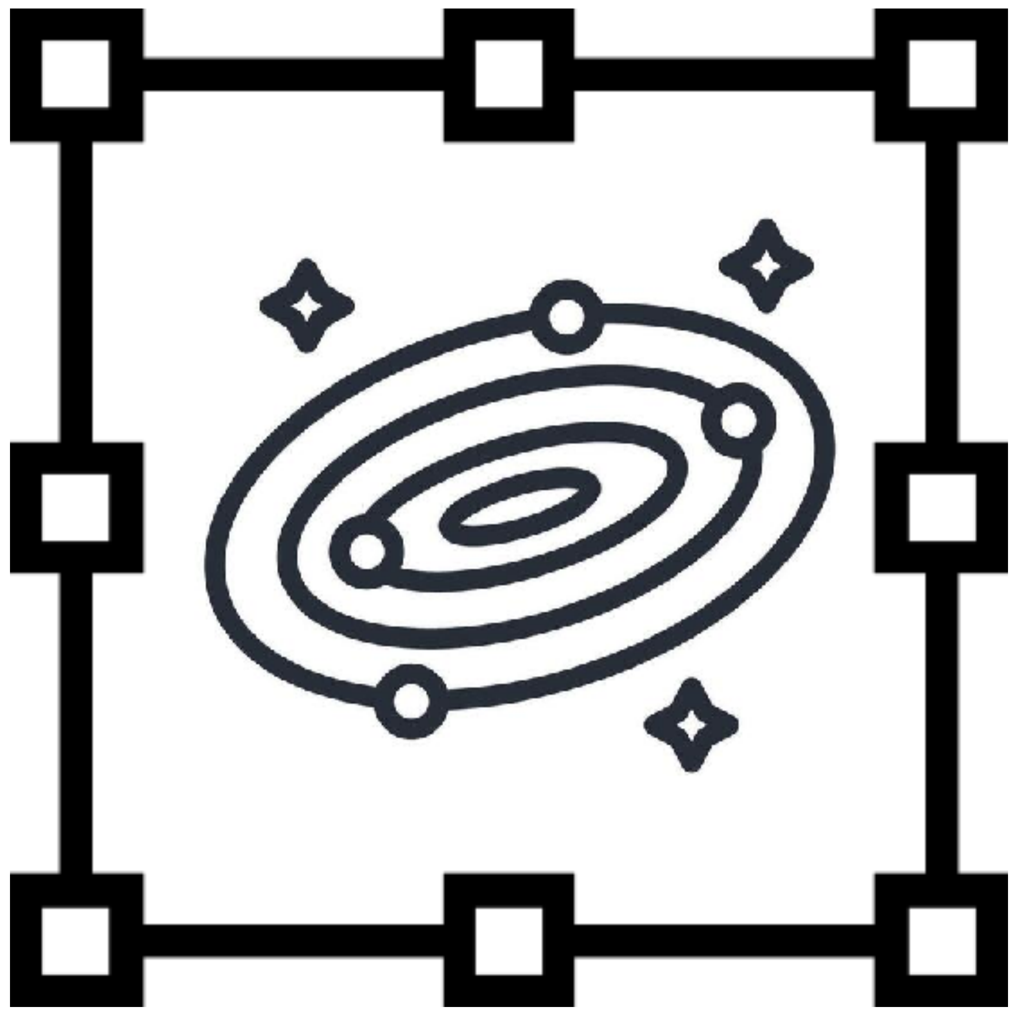
Elias Fano

Encode common prefixes of adjacent integers implicitly

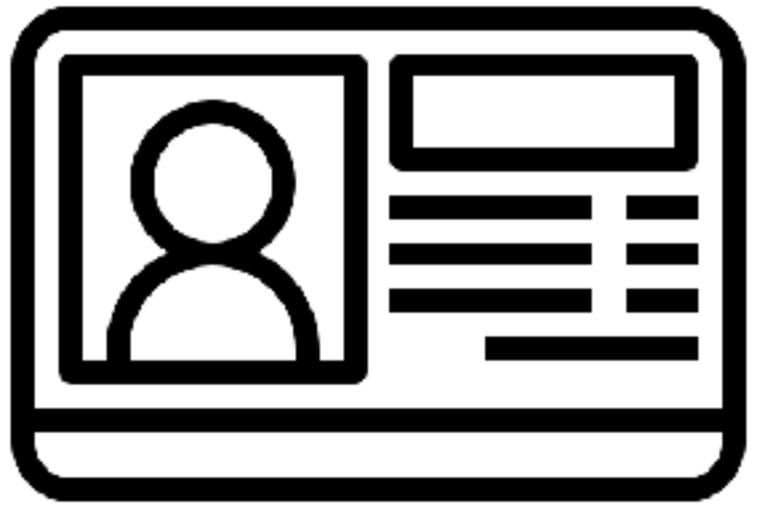
Assumptions from Last Time

Assumptions from Last Time

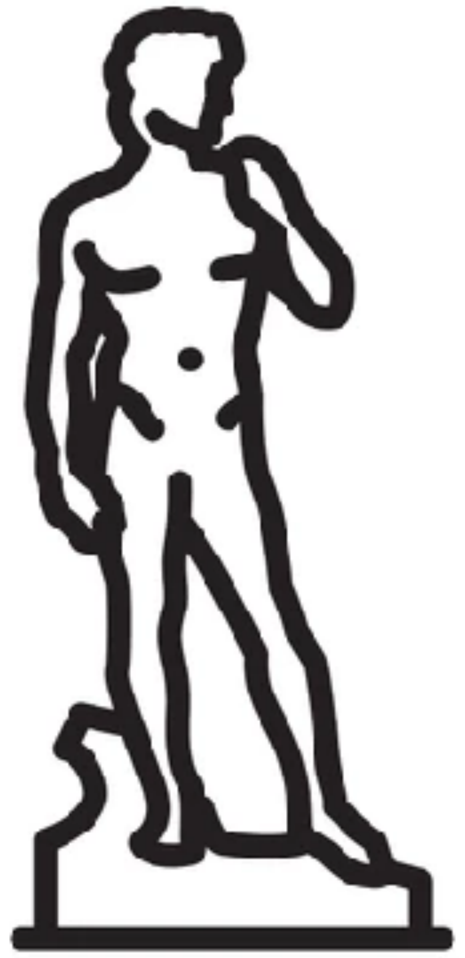
Bounded Universe



unique integer keys

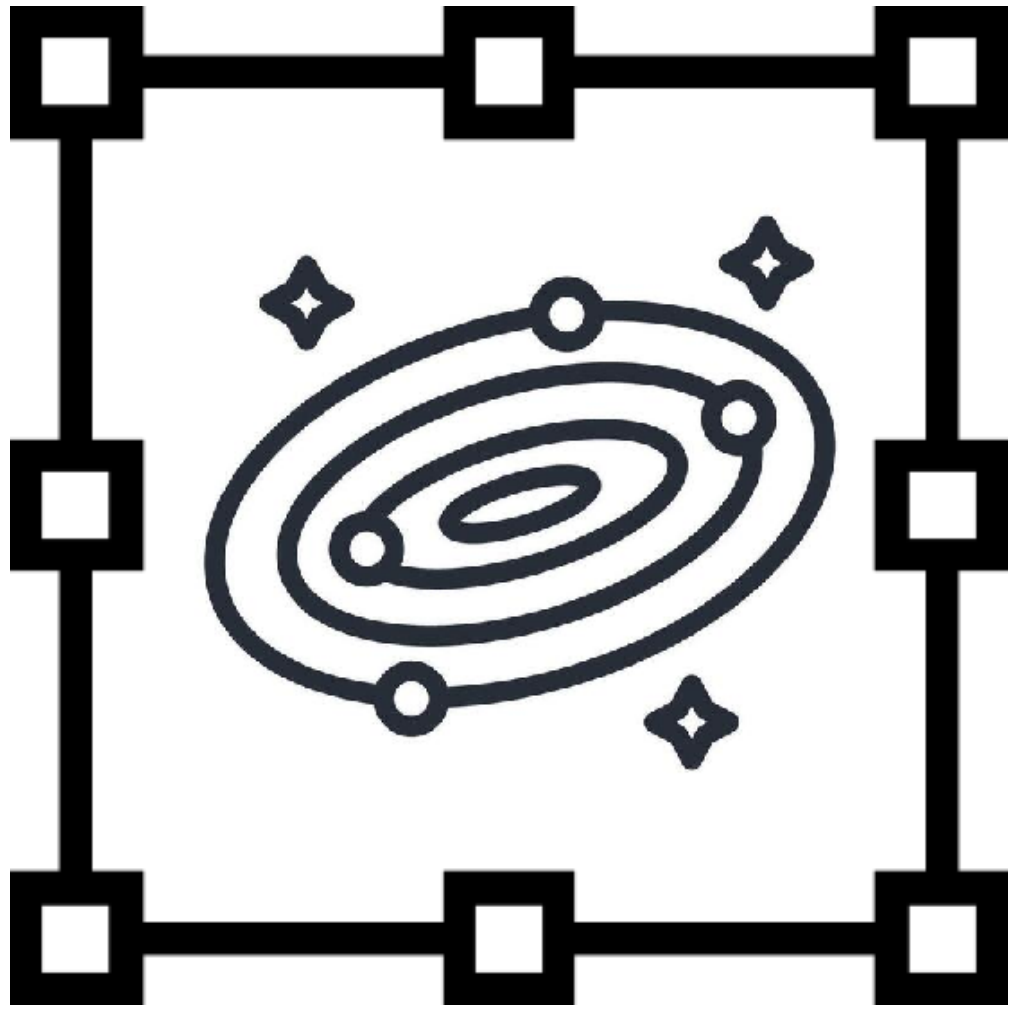


No duplicates

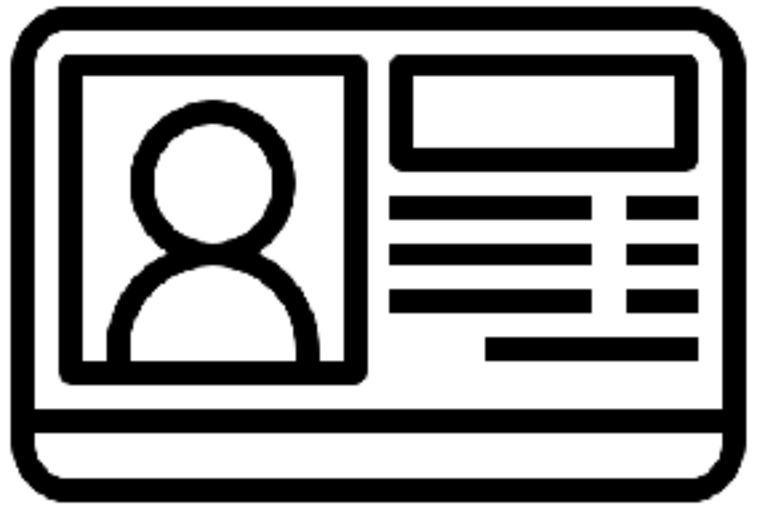


Assumptions from Last Time

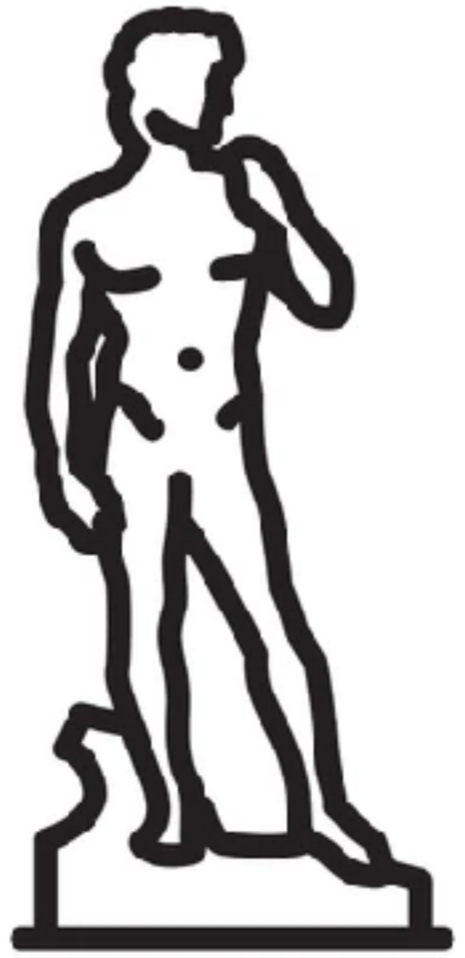
Bounded
Universe



unique
integer keys



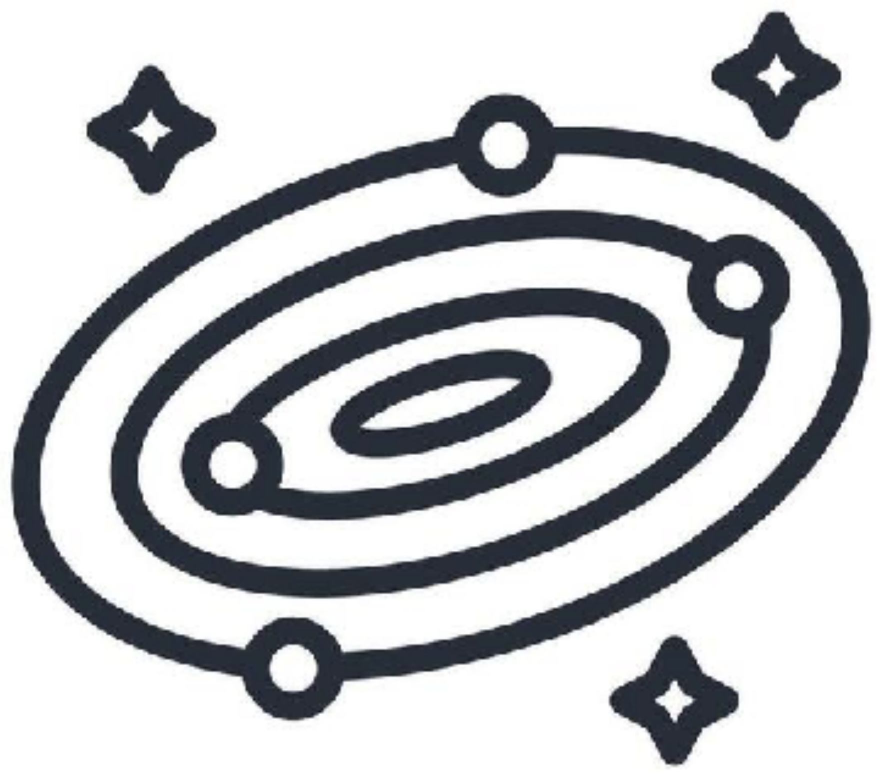
No duplicates



What if none of these hold?

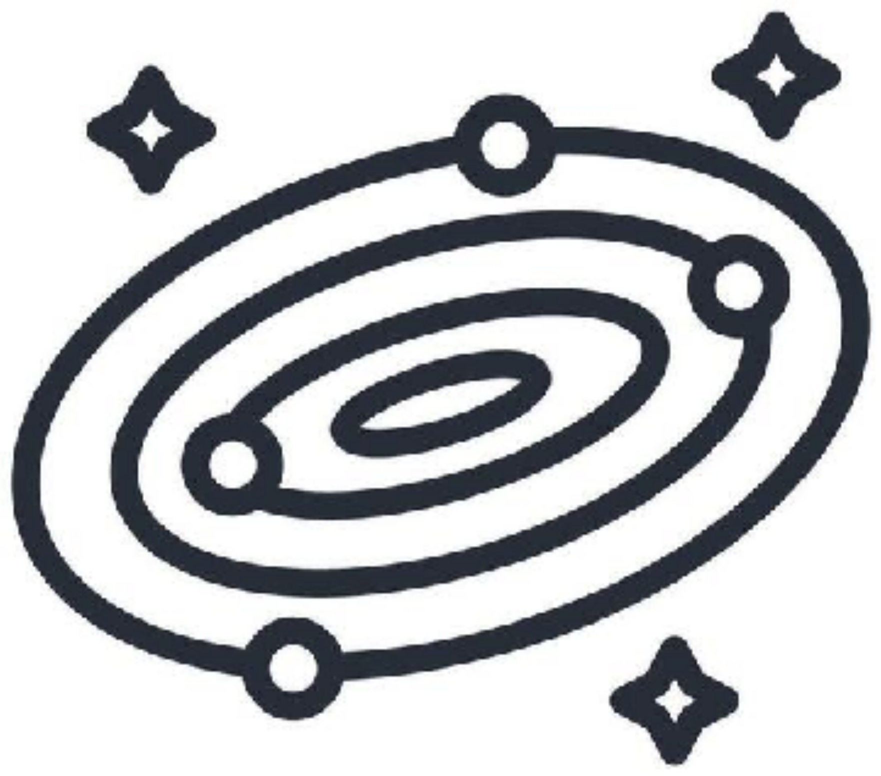
Today's Assumptions

Unbounded Universe



Today's Assumptions

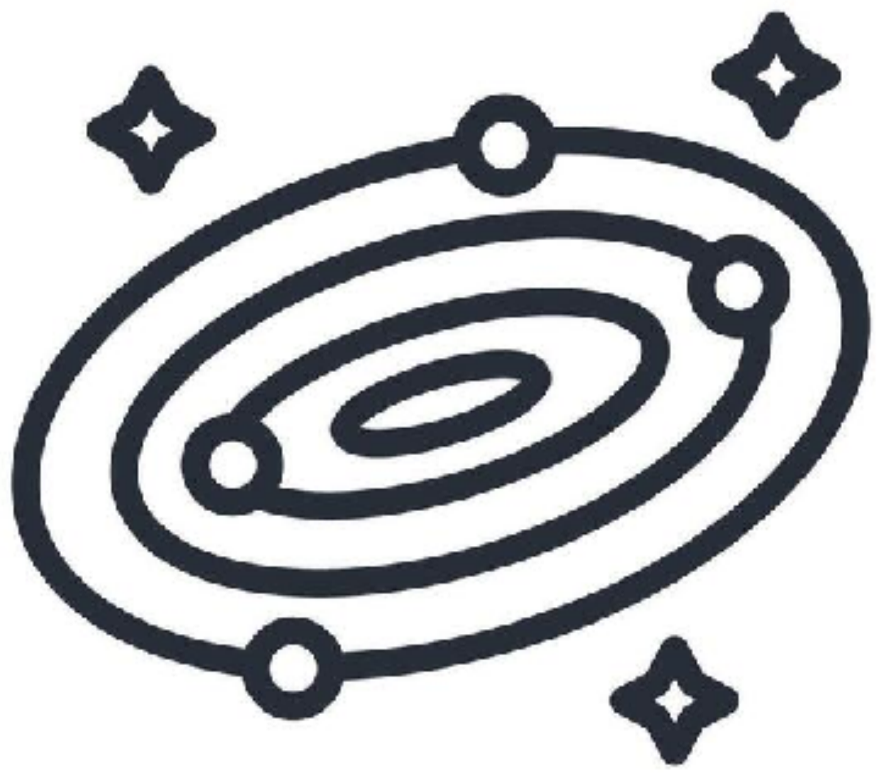
Unbounded
Universe



Var-length
String keys

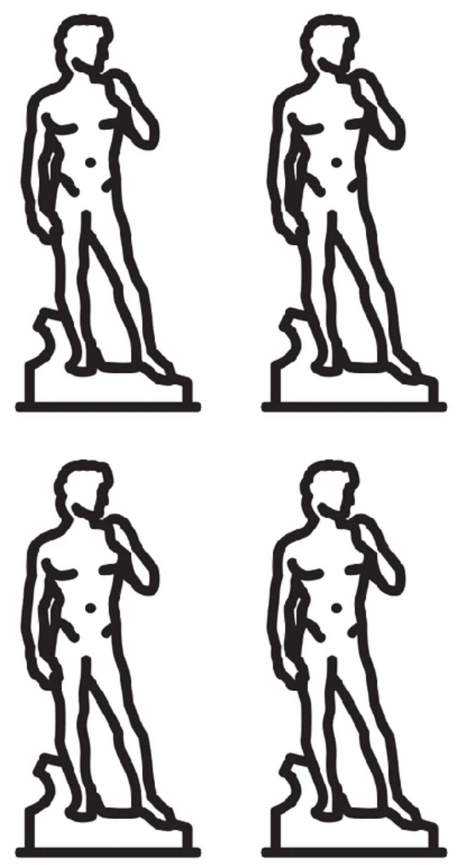
Today’s Assumptions

Unbounded
Universe

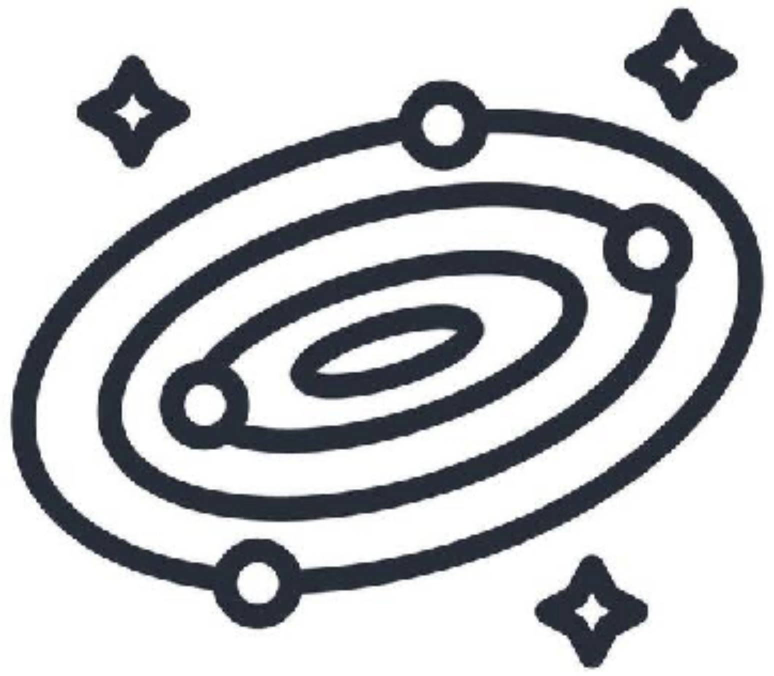


Var-length
String keys

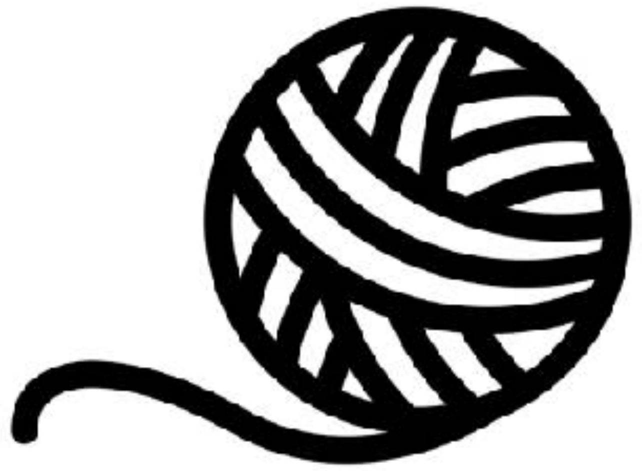
Duplicates
allowed



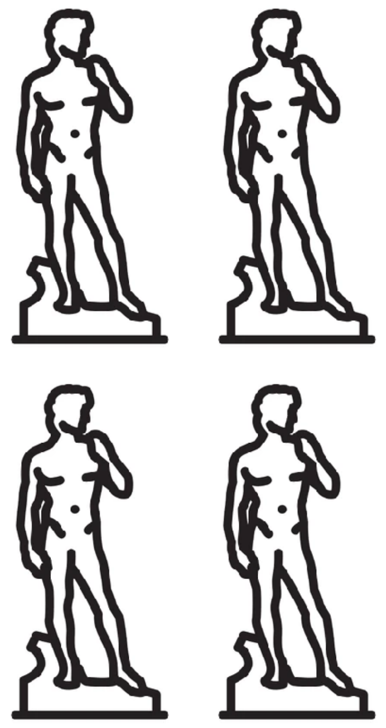
Unbounded
Universe



Var-length
String keys

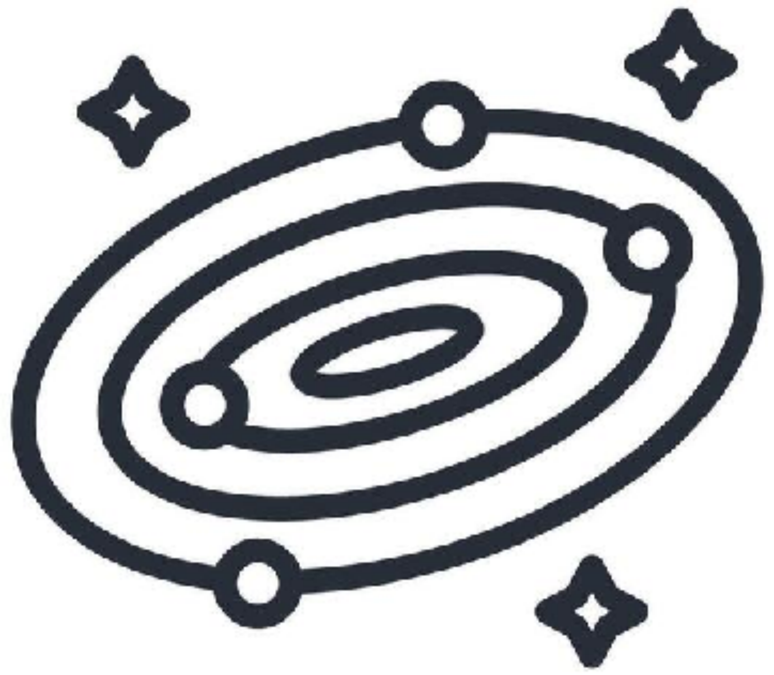


Duplicates
allowed

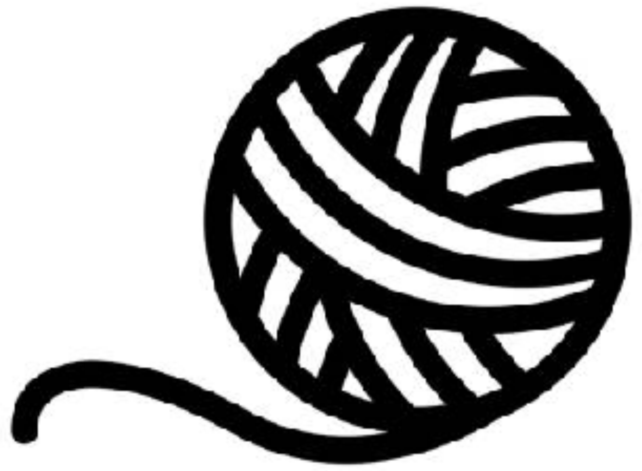


Key-value pairs

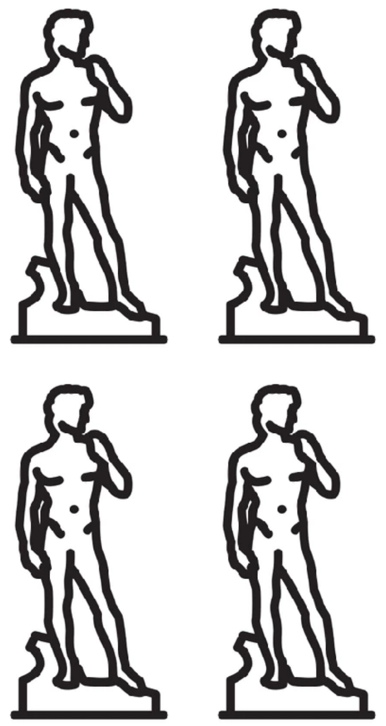
Unbounded Universe



Var-length String keys



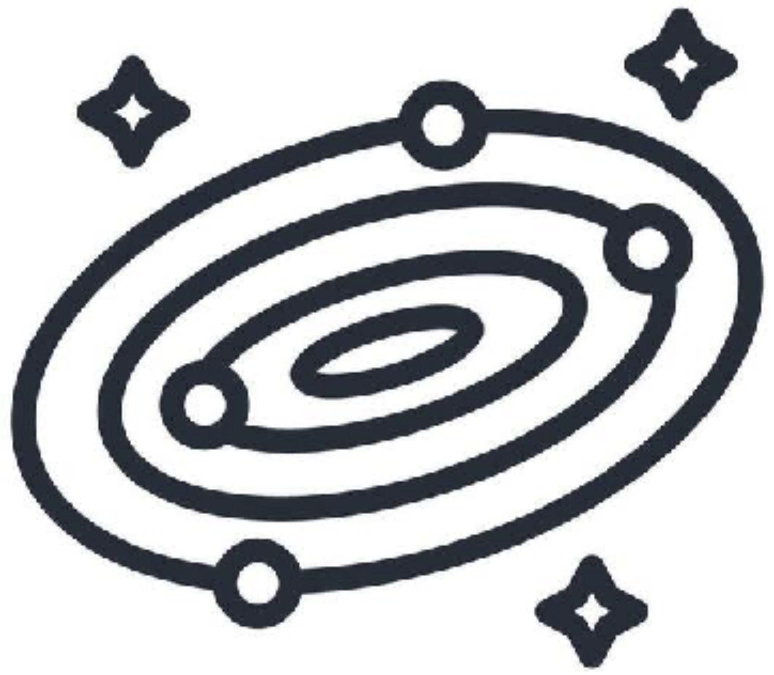
Duplicates allowed



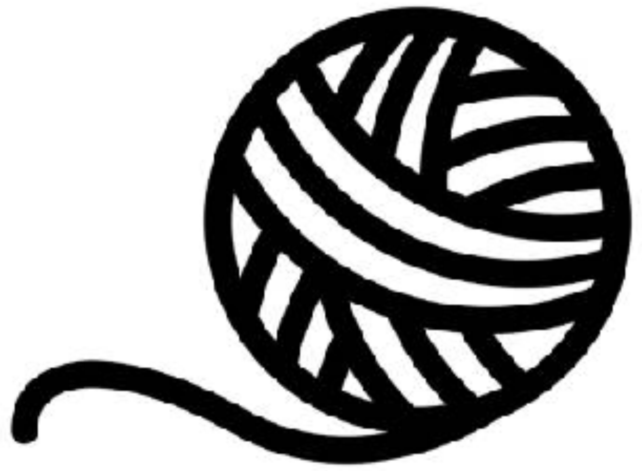
Key-value pairs

Static

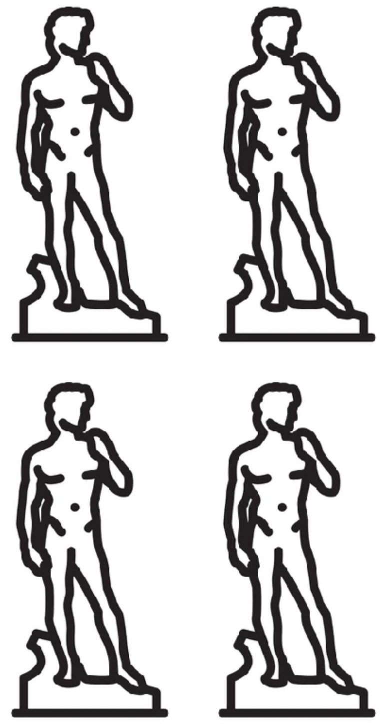
Unbounded Universe



Var-length String keys



Duplicates allowed



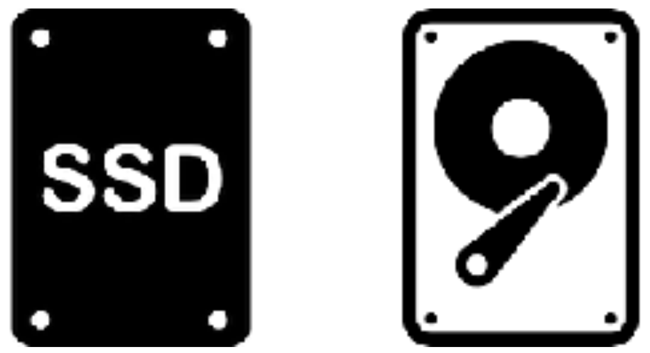
Key-value pairs

Static

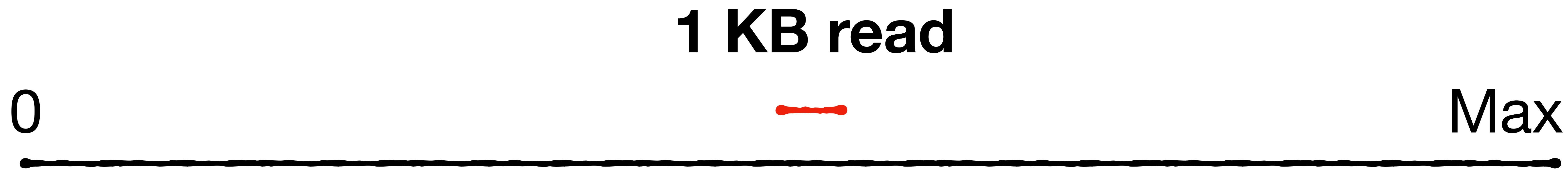
Storage

0 Max

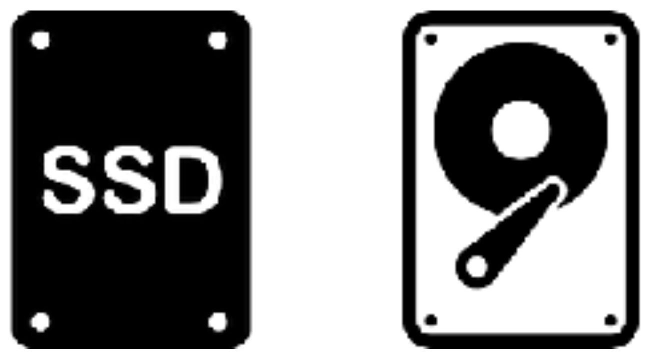
Block device interface



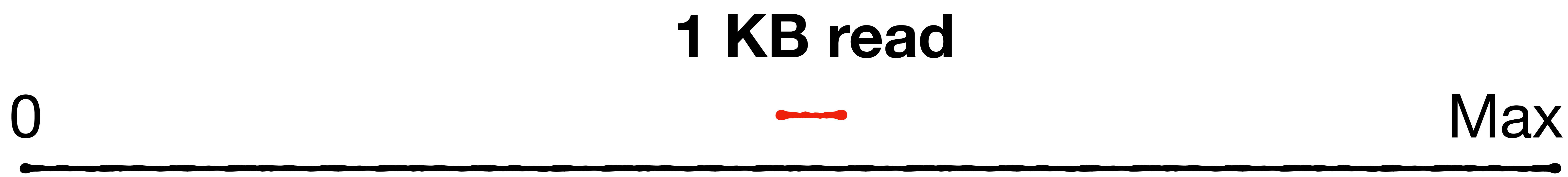
Reading/writing from storage at units of less than ≈4KB does not pay off.



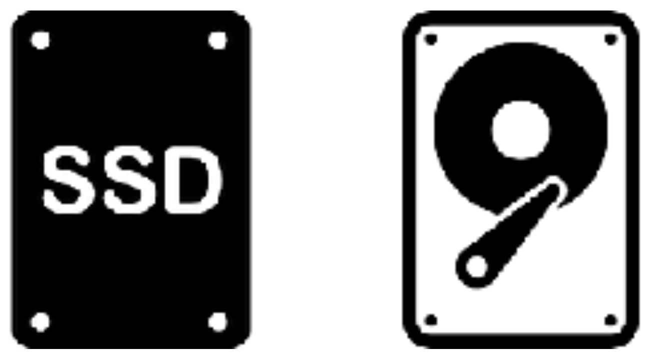
Block device interface



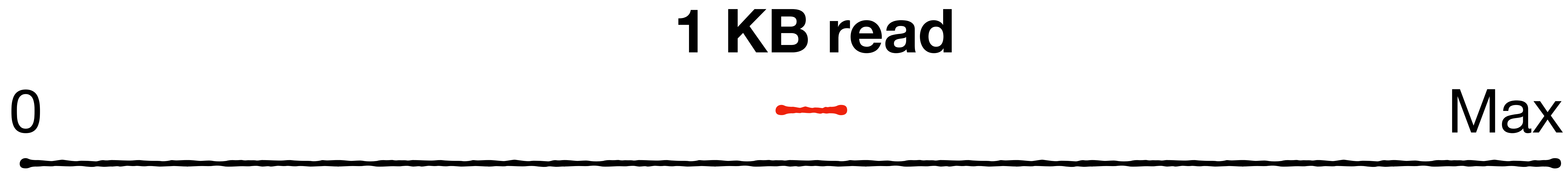
Reading/writing from storage at units of less than ≈4KB does not pay off.



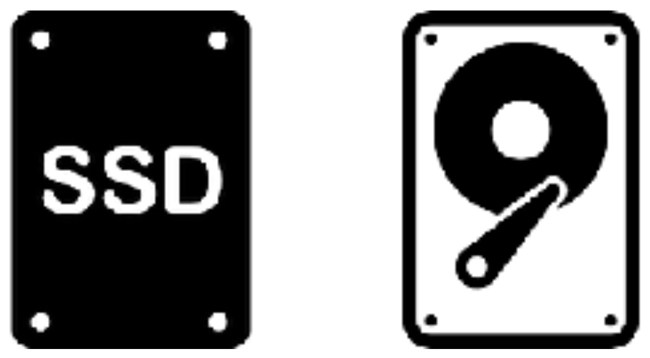
Disk: head movement dominates



Reading/writing from storage at units of less than ≈4KB does not pay off.

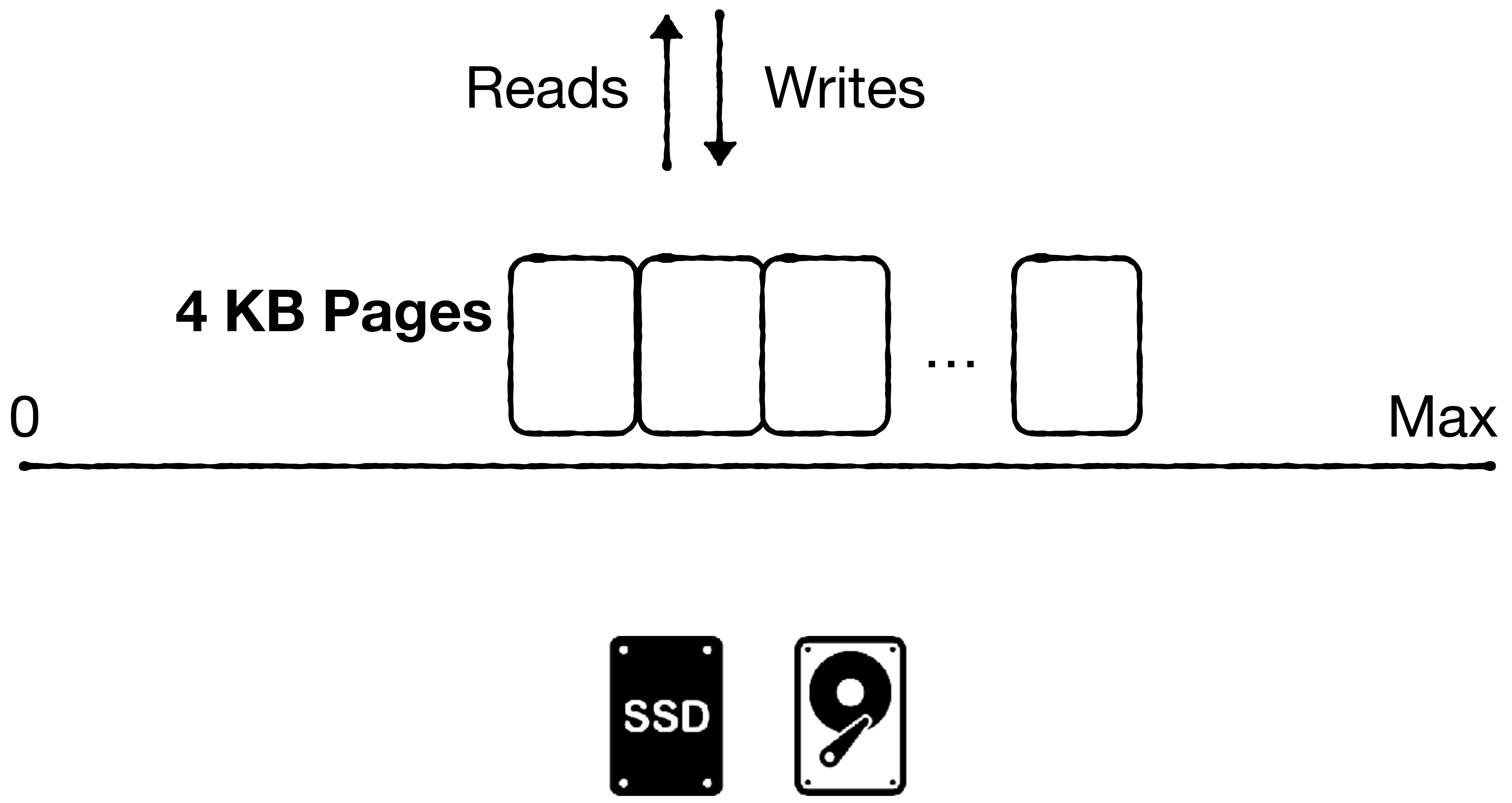


Disk: head movement dominates

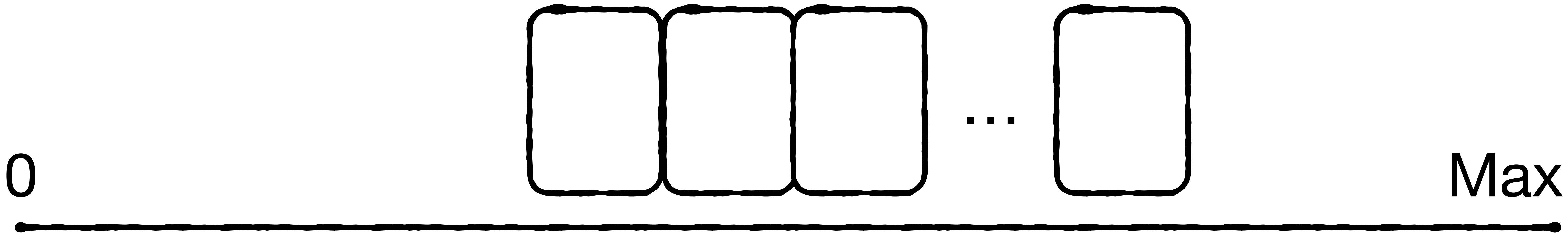


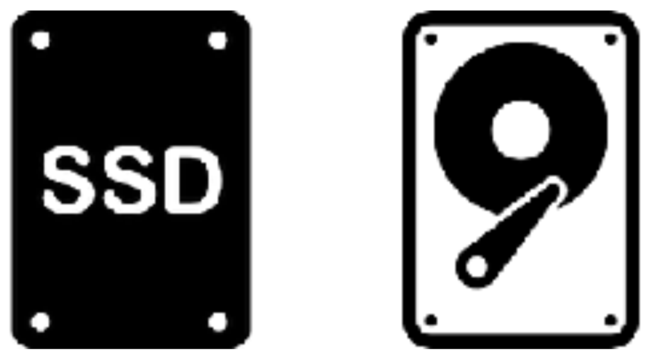
Minimum page read is 4 KB

organize data into 4KB pages. Read/write at 4KB granularity



How to store sorted static var-length kv-pairs in these pages?

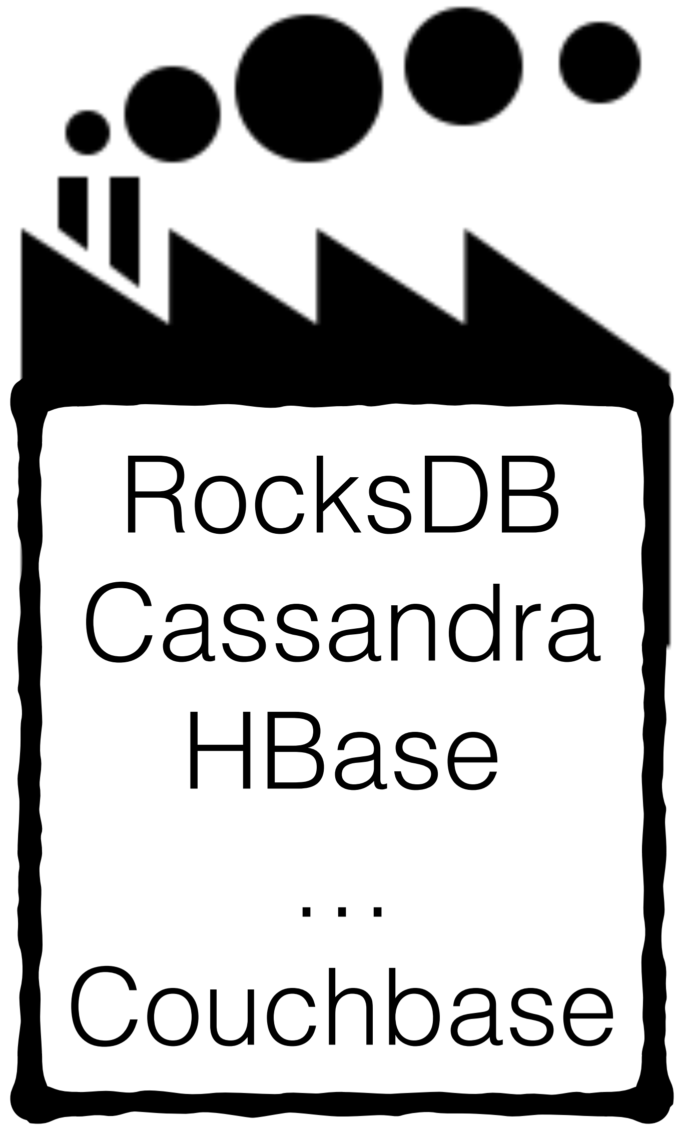


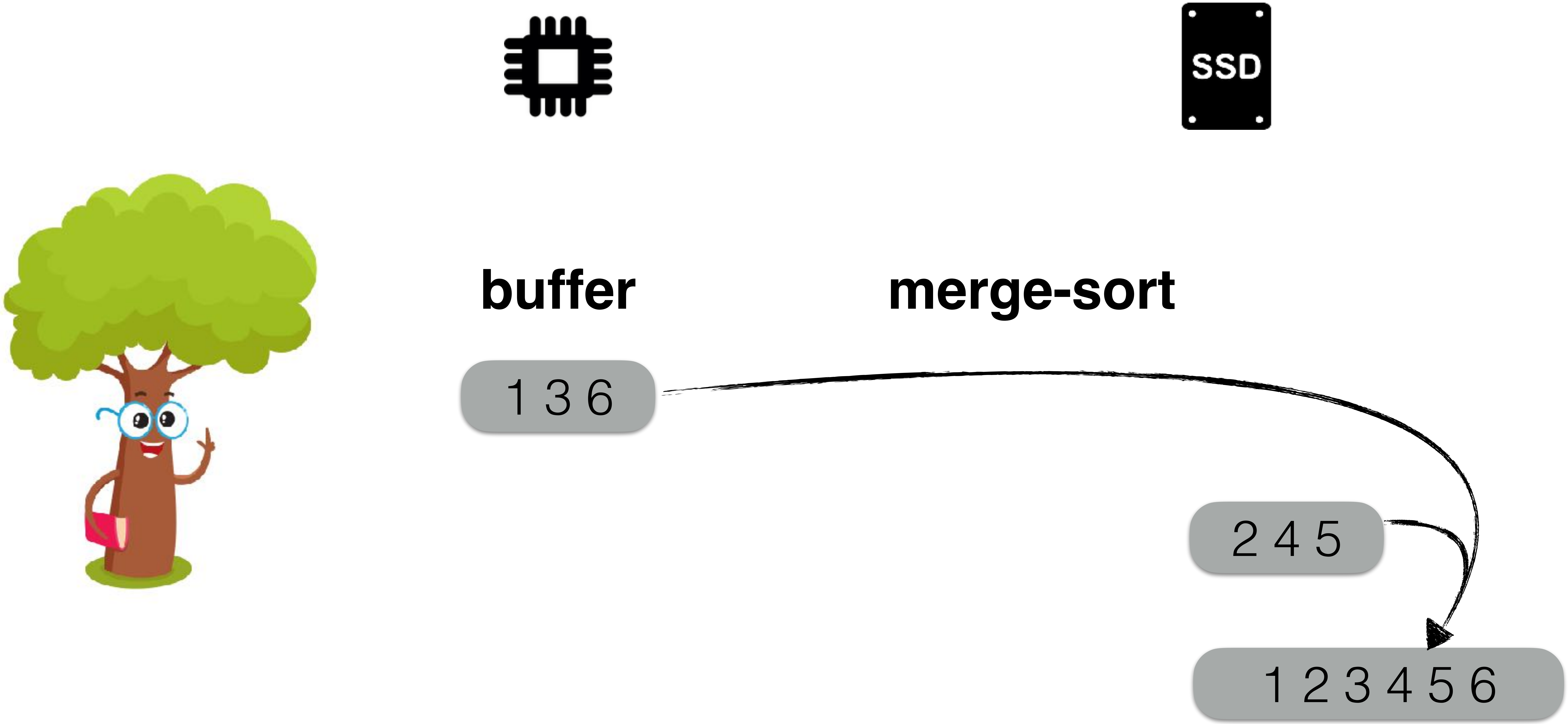


Application?

Application - LSM-Trees

Application - LSM-Trees





SSD
buffer
merge-sort
1 3 6
2 4 5
1 2 3 4 5 6

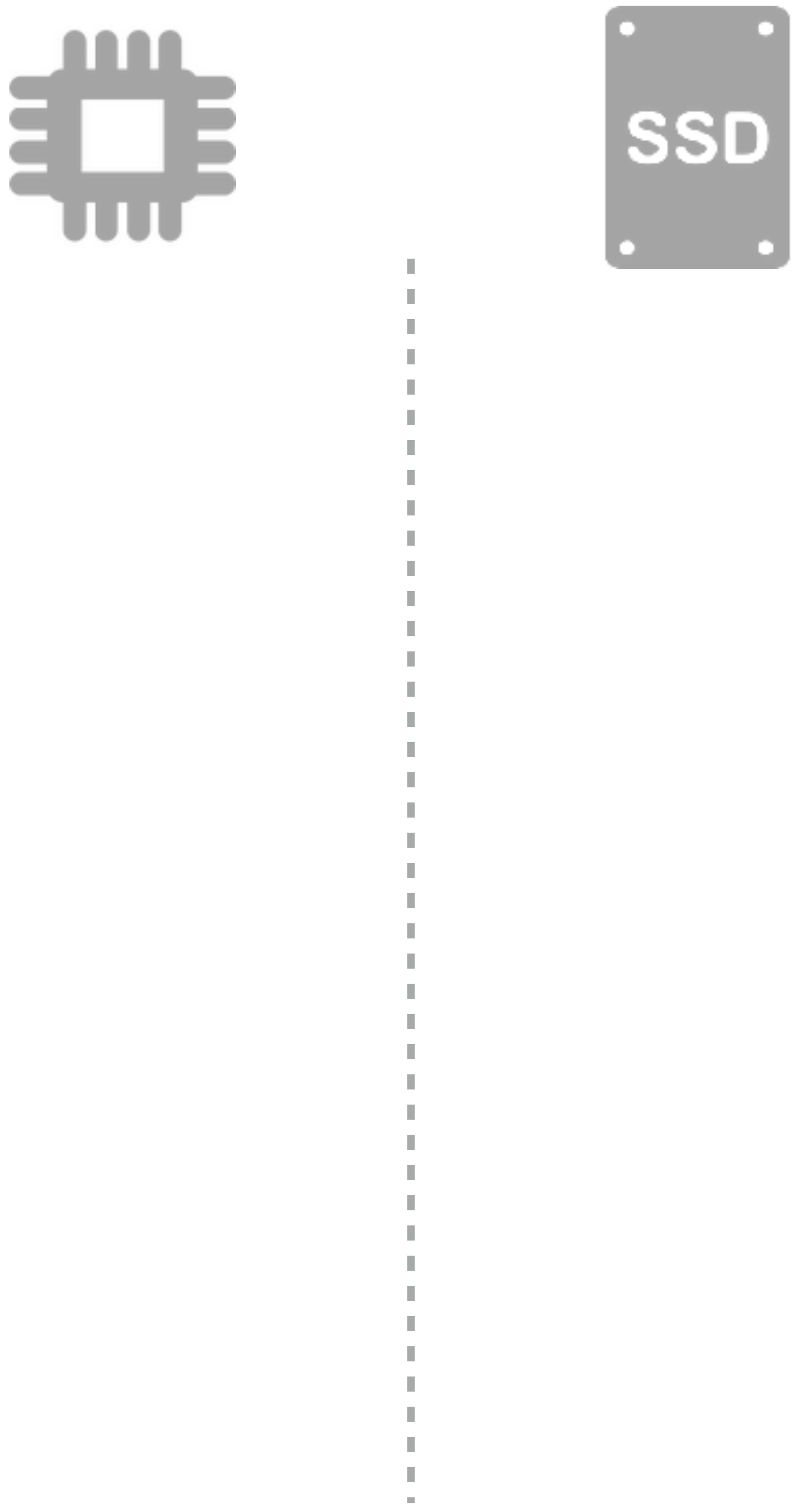
Only sequential writes

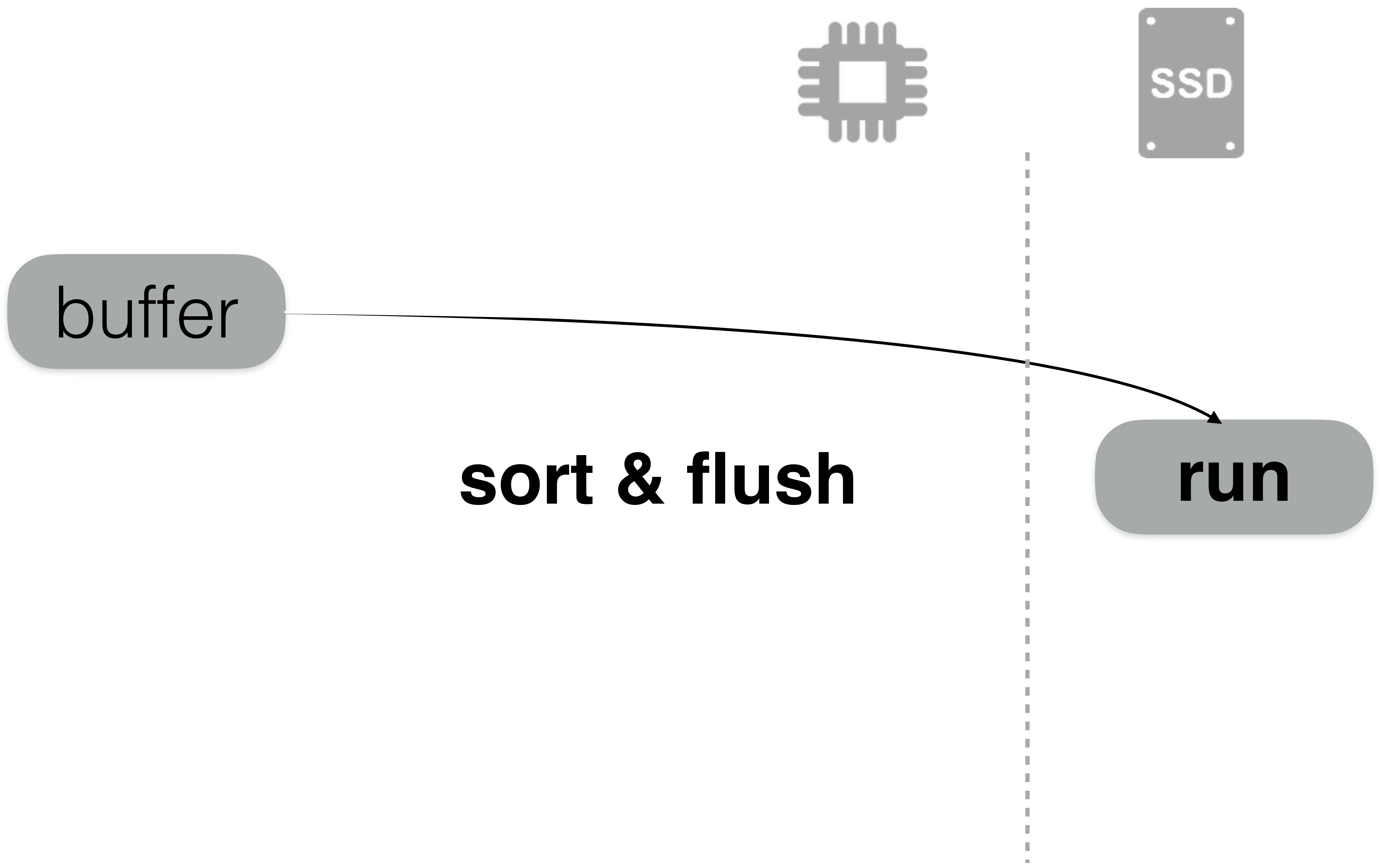
Inserts

of kv-pairs

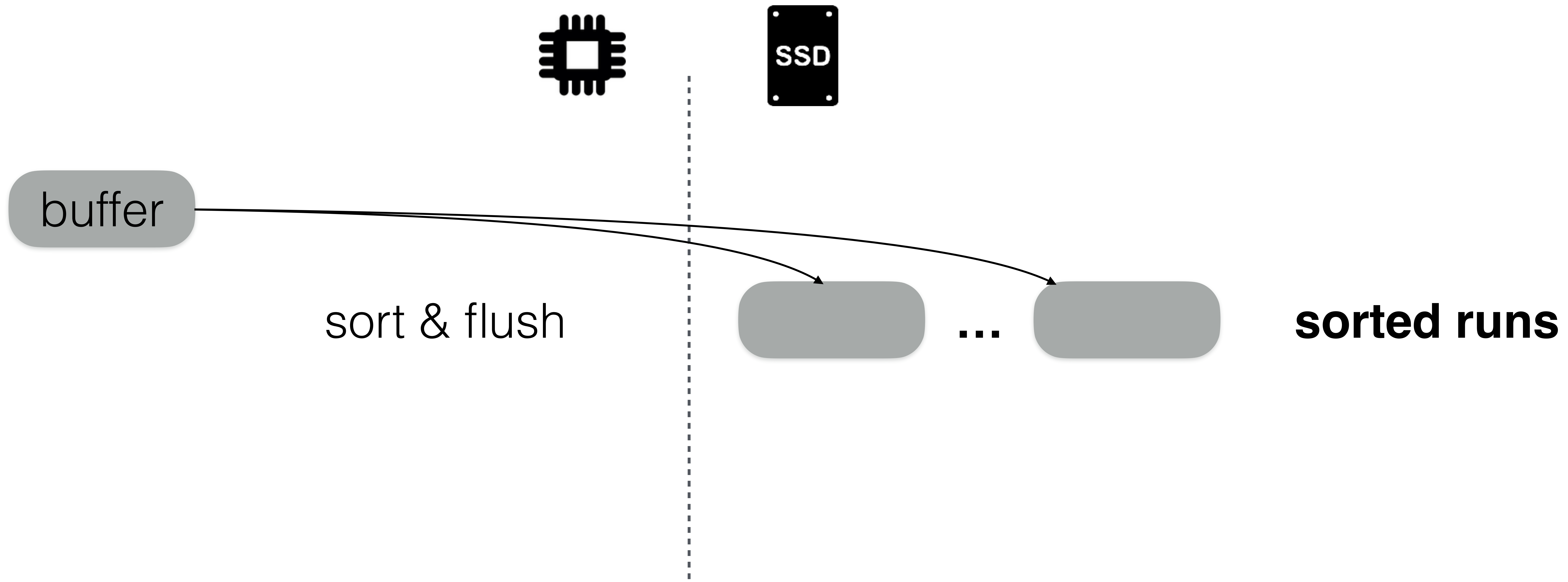
↓

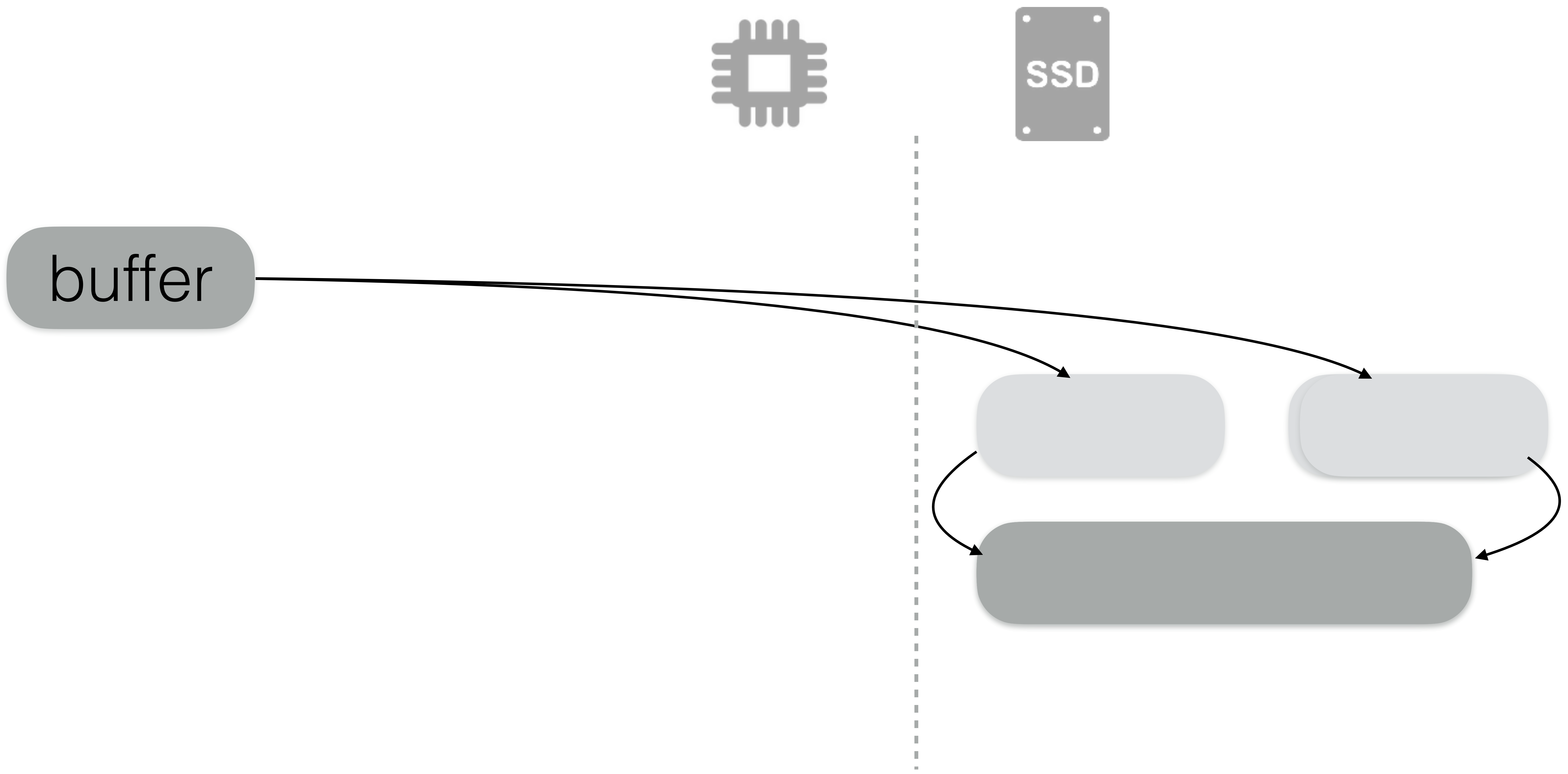
buffer



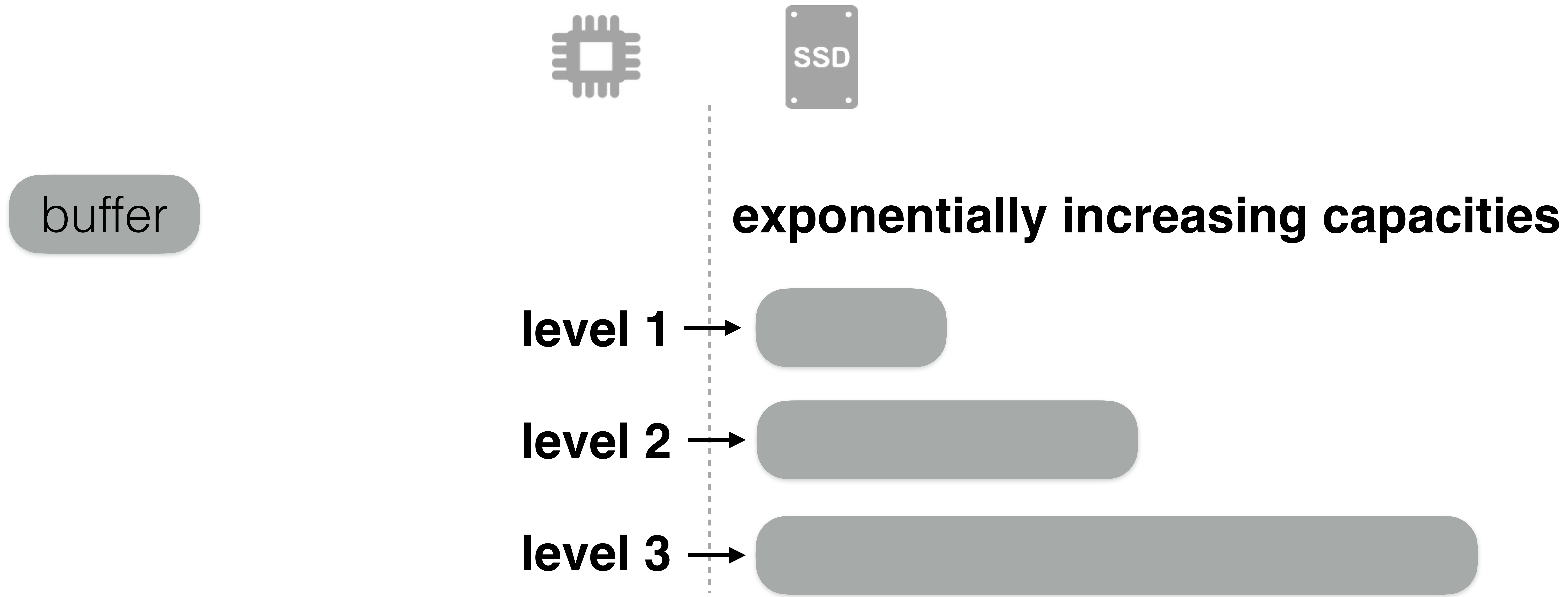


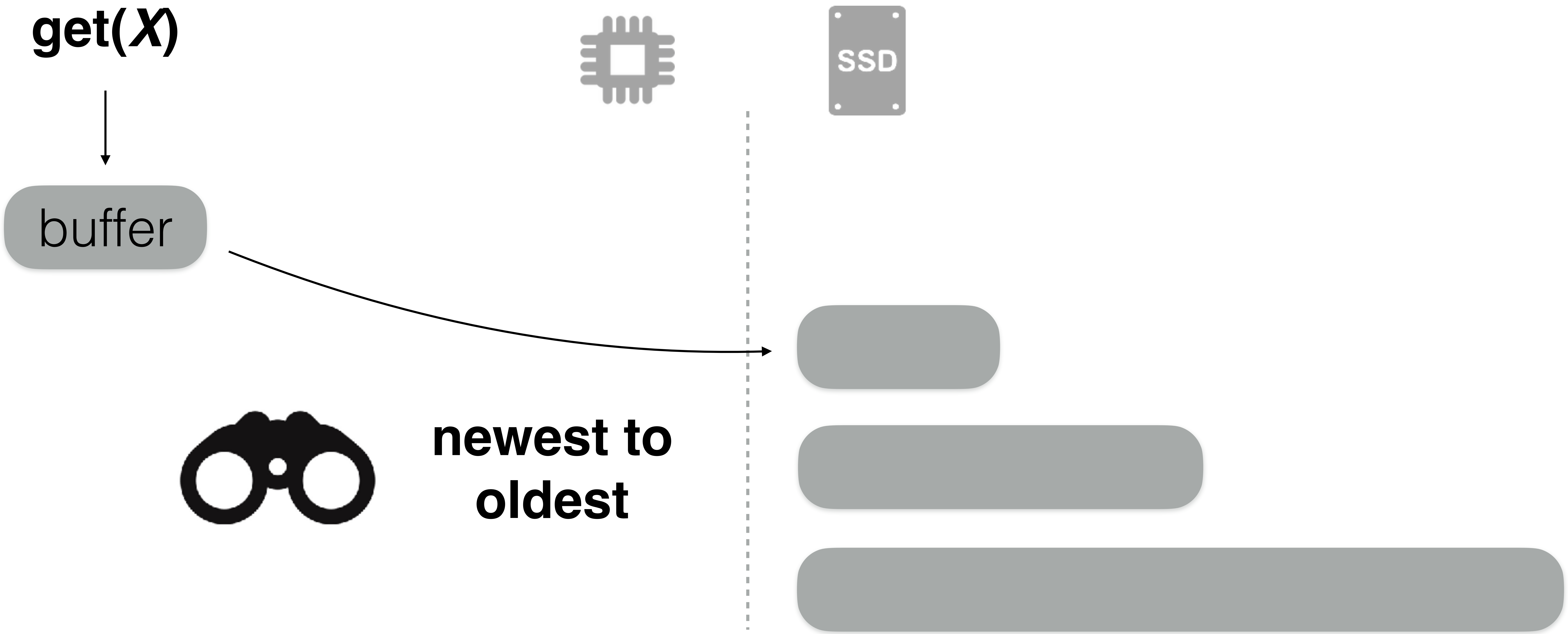
SSD
buffer
sort & flush
run

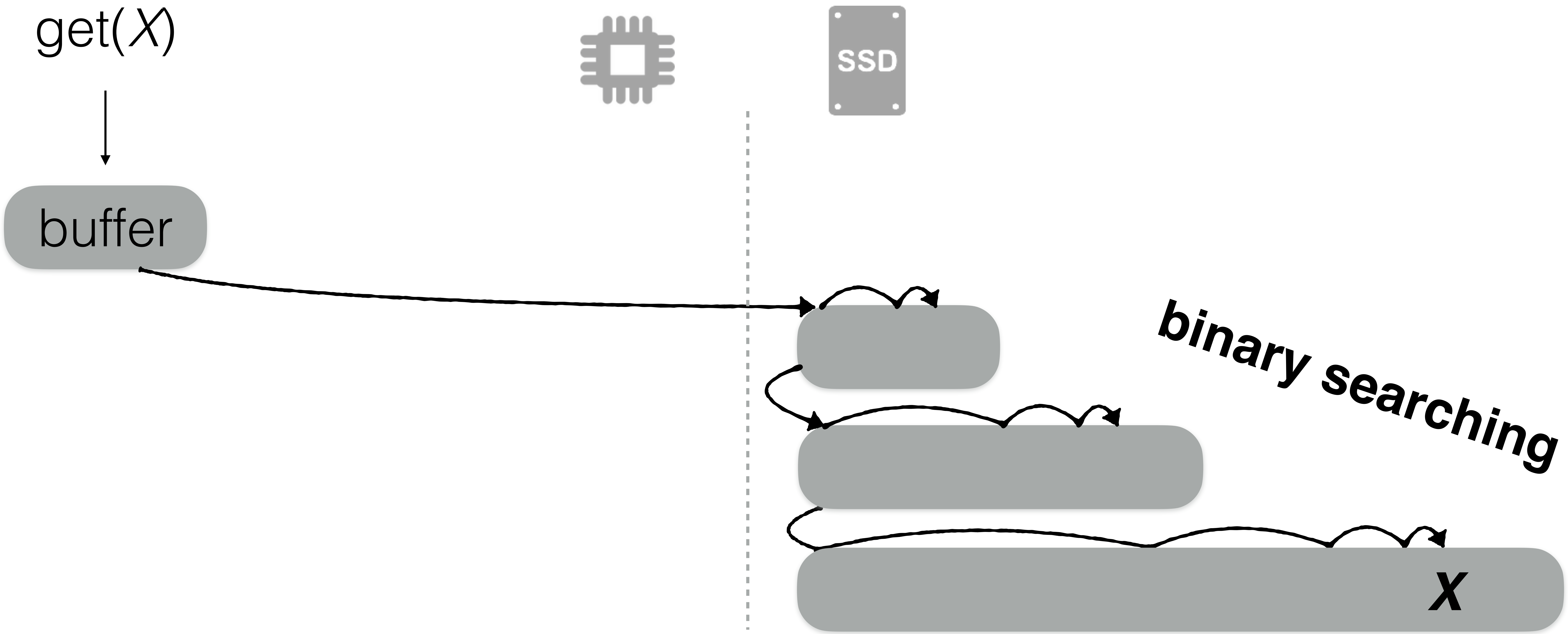

SSD
buffer
sort & flush
...
sorted runs

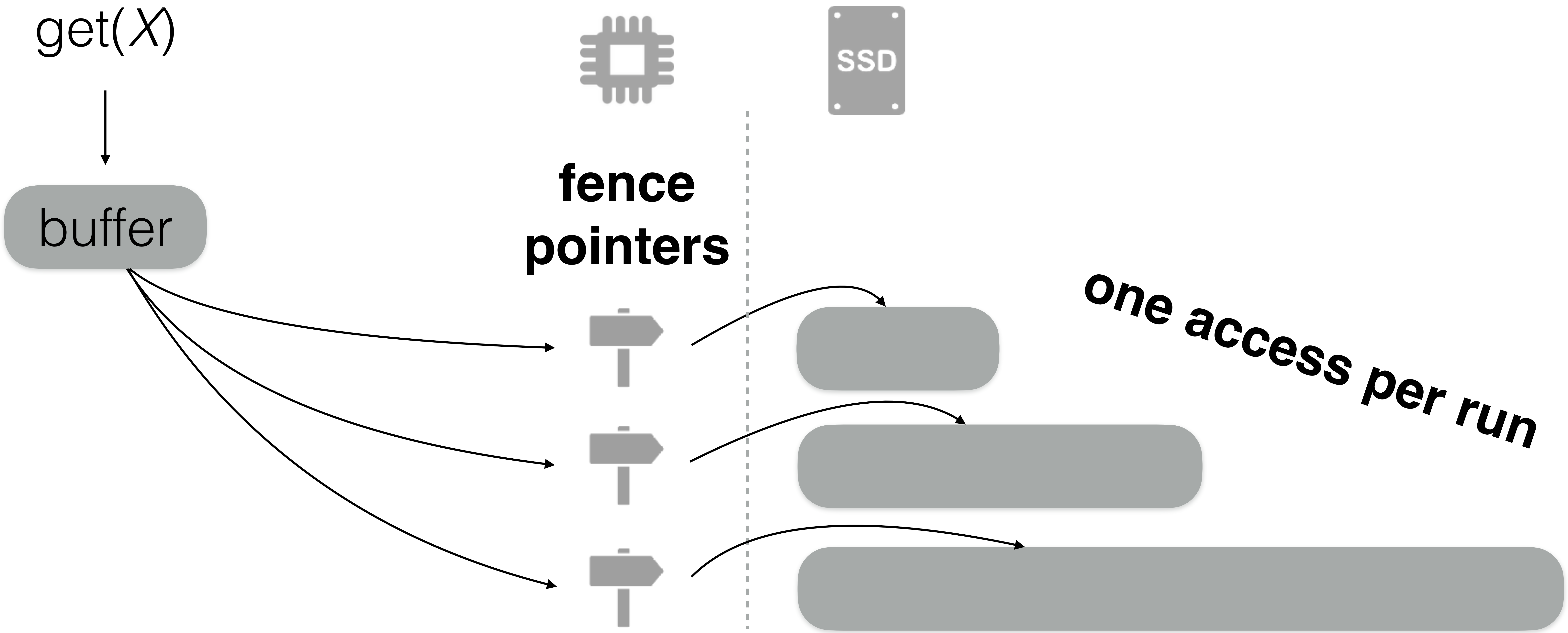


sort-merge

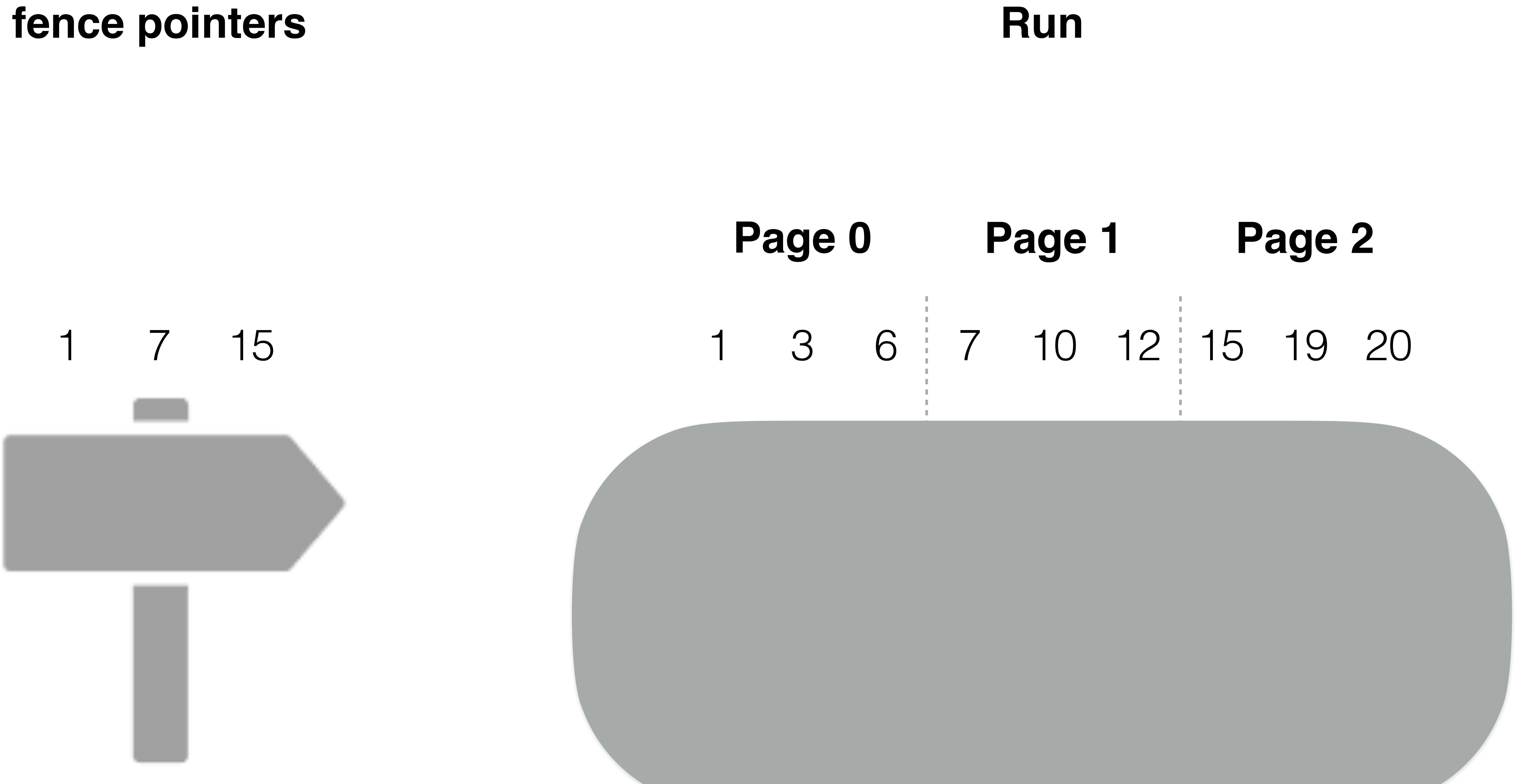

SSD
buffer
exponentially increasing capacities
level 1
level 2
level 3


get(*X*)
SSD
buffer
newest to oldest

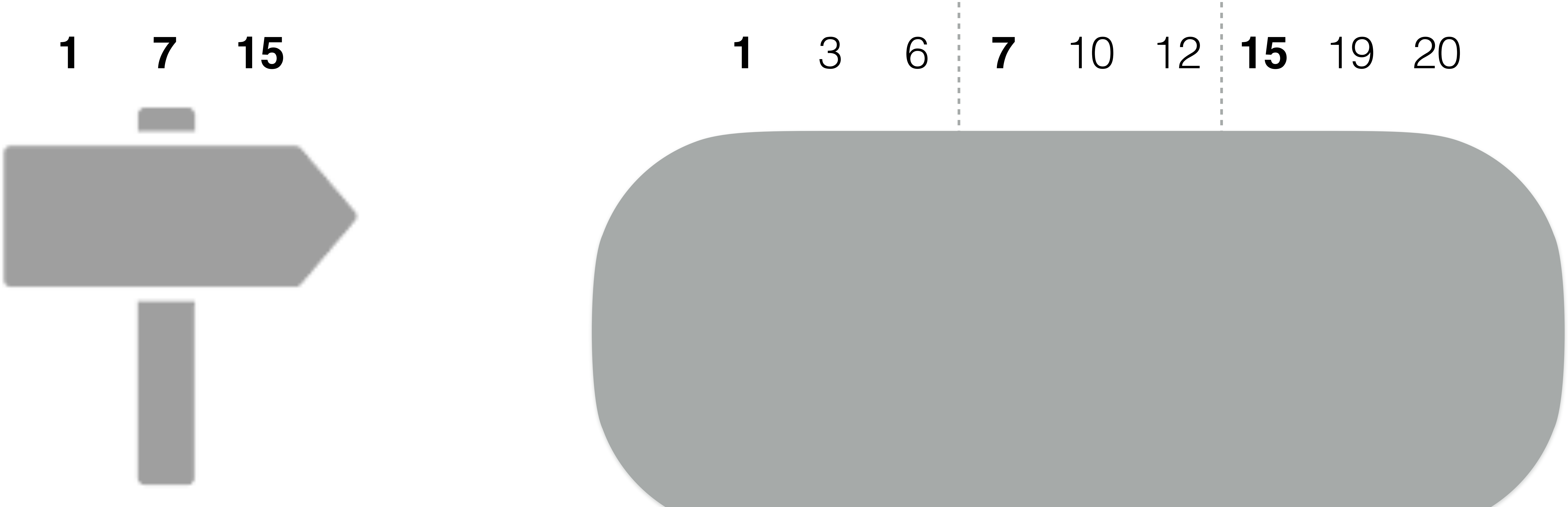

get(X)
SSD
buffer
binary searching
X



get(X)
buffer
fence
pointers
SSD
one access per run


fence pointers
Run
Page 0
Page 1
Page 2
1 7 15
1 3 6
7 10 12
15 19 20

First key of each page

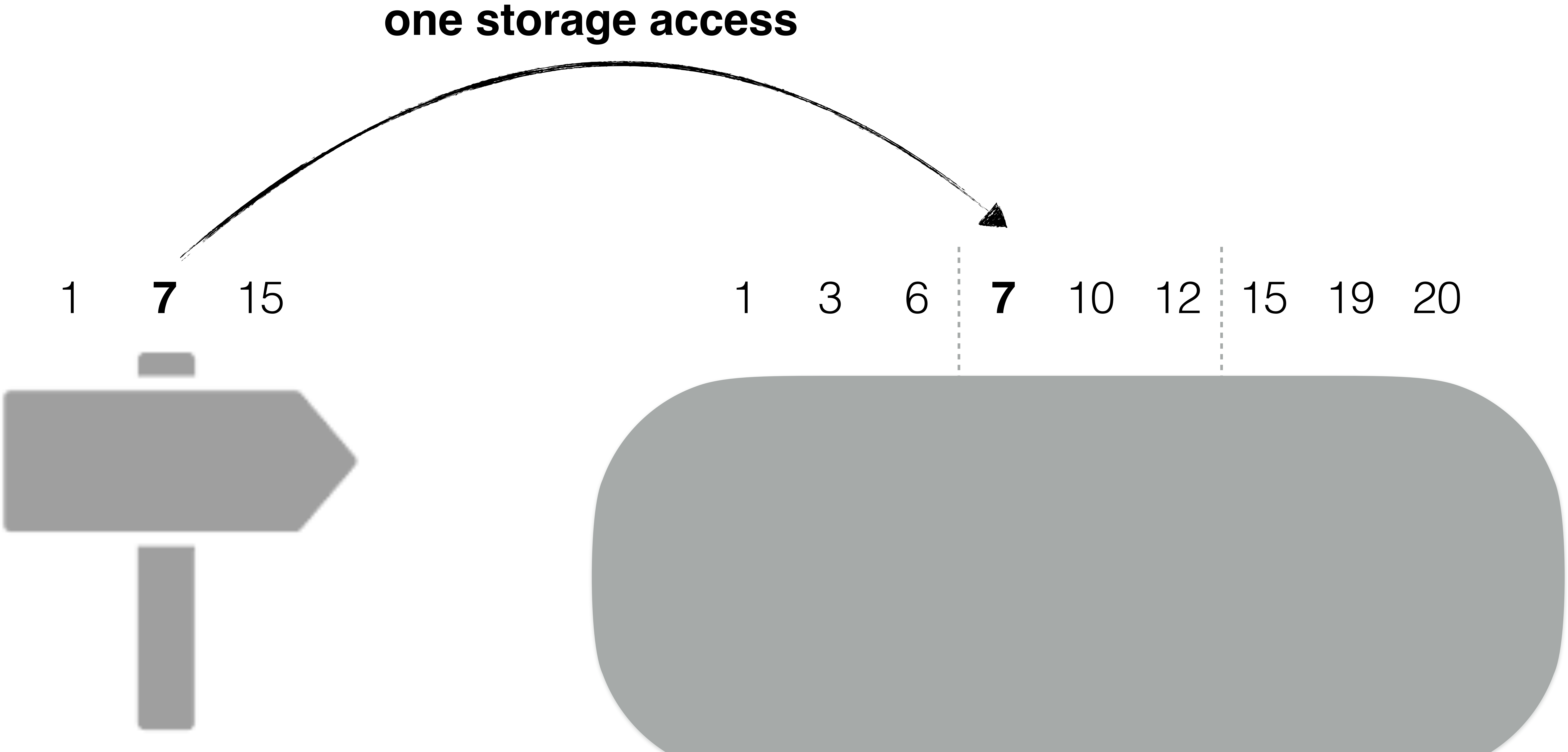


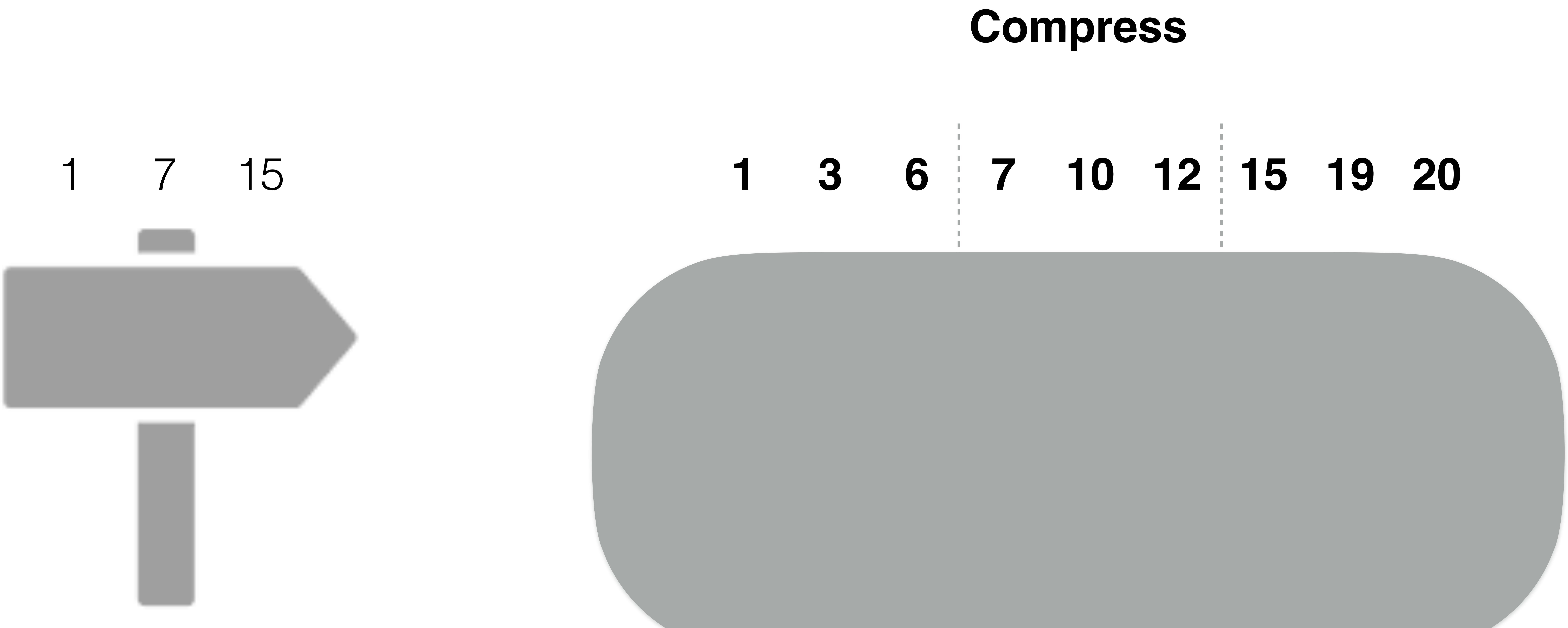
First key of each page

binary search

1 7 15

1 3 6 | 7 10 12 | 15 19 20

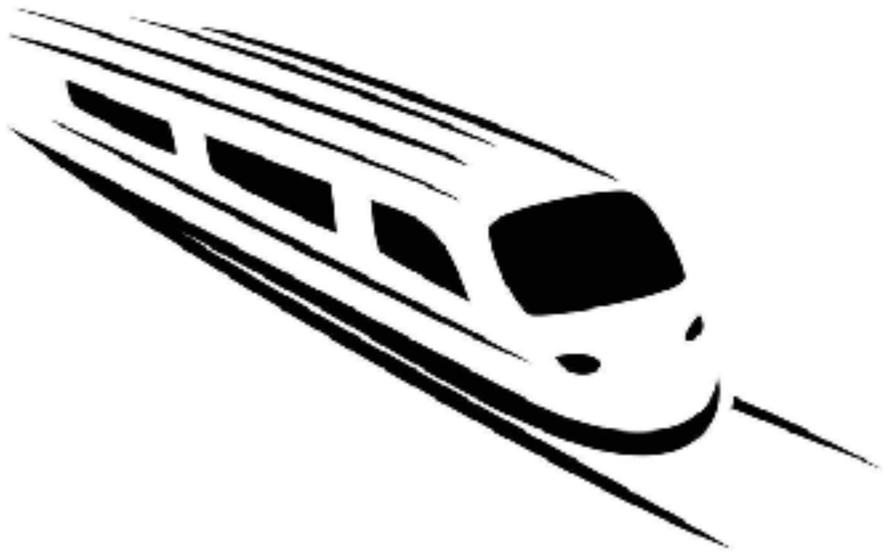

one storage access
1 7 15
1 3 6 7 10 12 15 19 20


Compress
1 7 15
1 3 6 7 10 12 15 19 20

Why Compress Data in Storage?

Why Compress Data in Storage?

Space efficiency

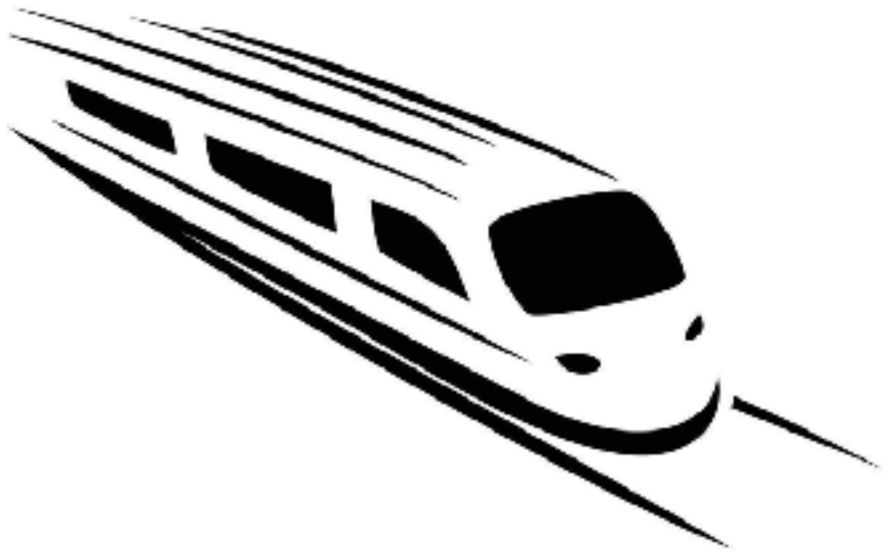


Bandwidth

Improve SSD lifetime

Why Compress Data in Storage?

Space efficiency

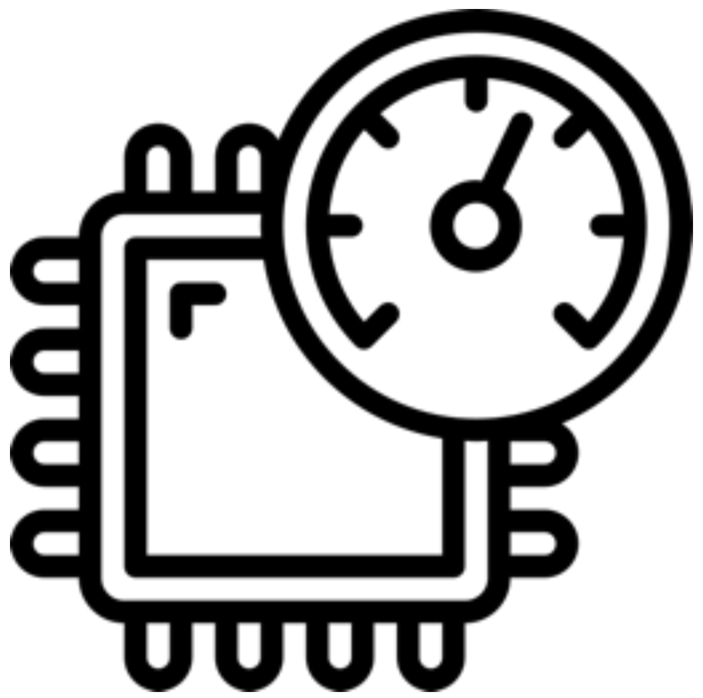


Bandwidth

Improve SSD lifetime

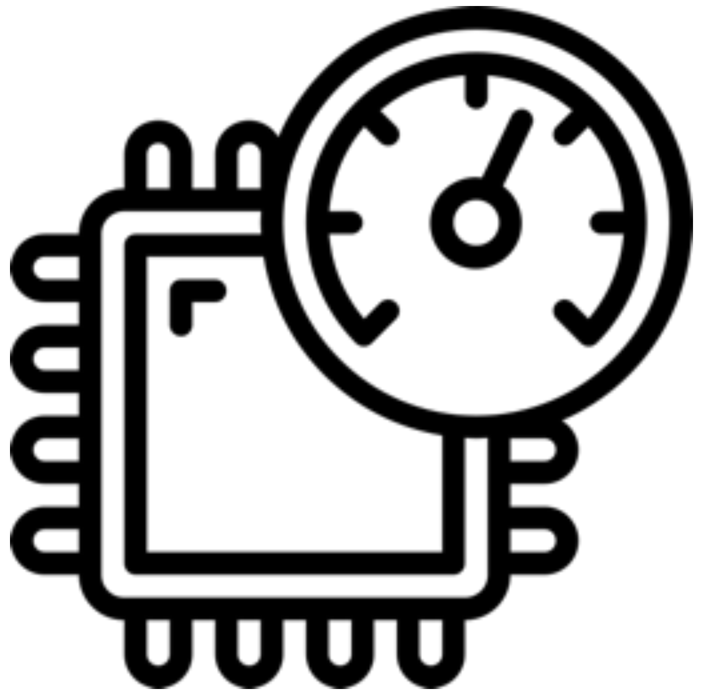
Downsides?

Compression Downsides



CPU for de/compression

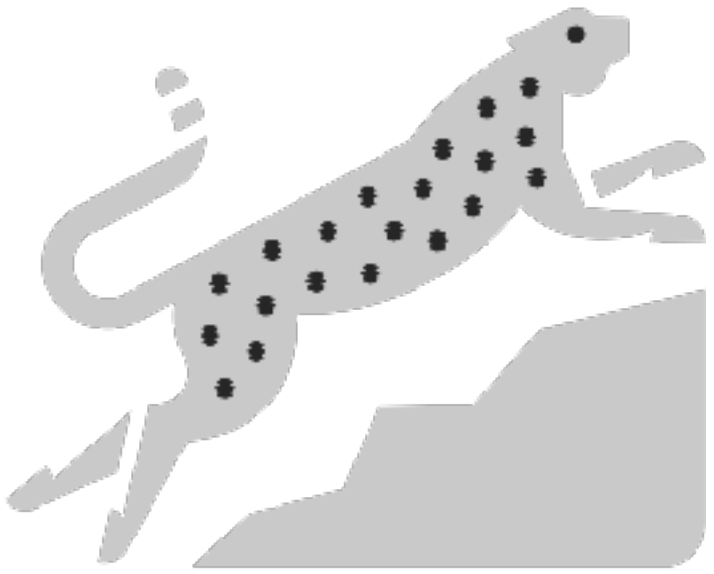
Compression Downsides



CPU for
de/compression

But since data is in storage, I/O rather than CPU is the bottleneck

Case Study



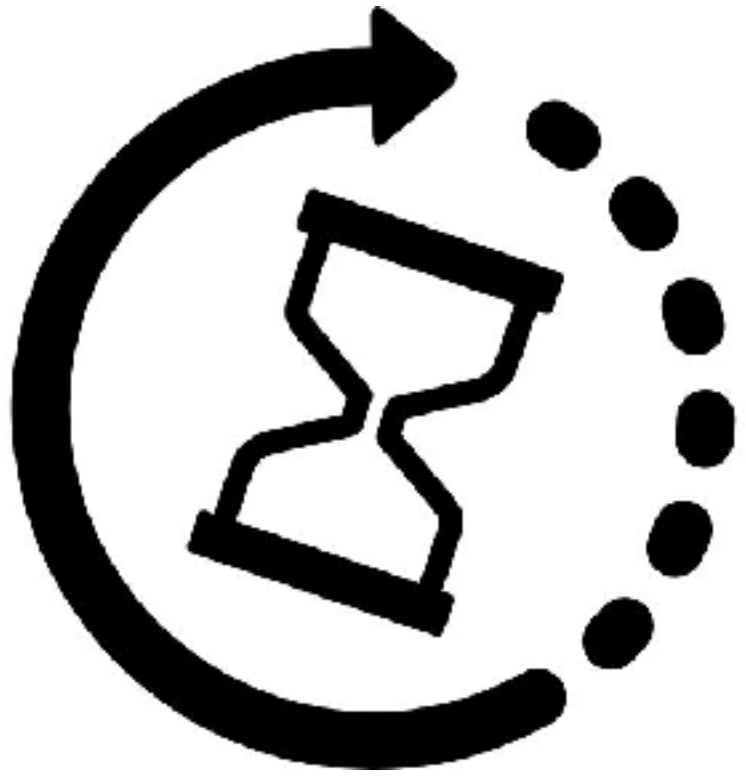
RocksDB

Combines many Techniques

Used in Practice

Disclaimer

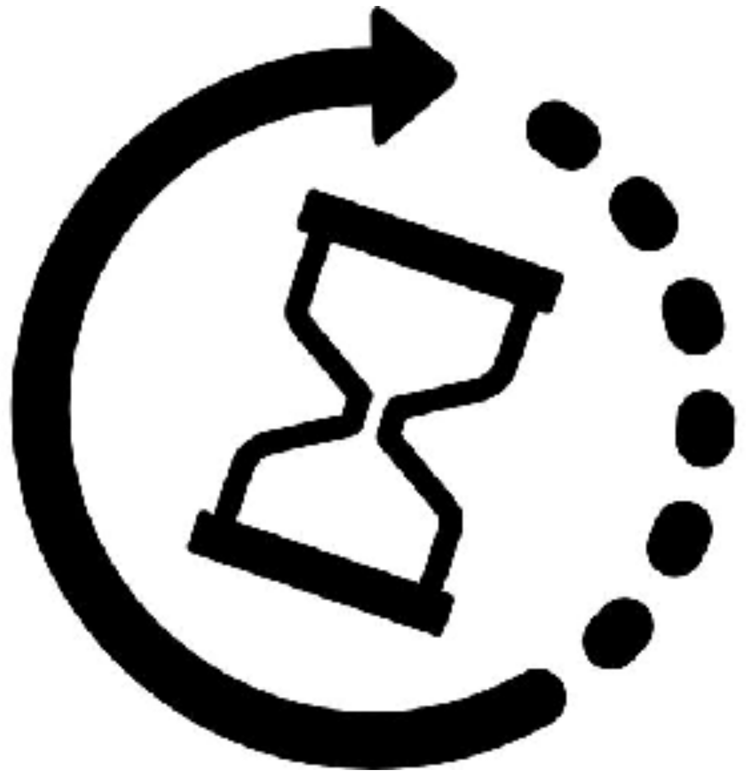
Best of my knowledge



Reflects 2 years ago

Disclaimer

Best of my knowledge



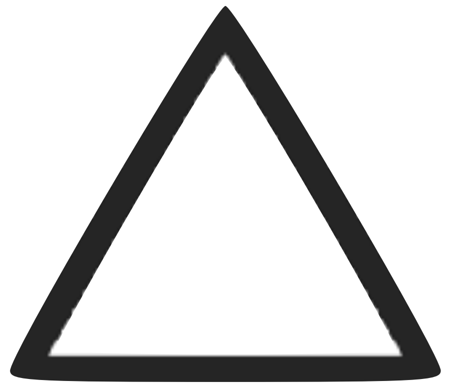
Reflects 2 years ago

Super interesting design & any misrepresentation is unintentional

Agenda

Page Sizing

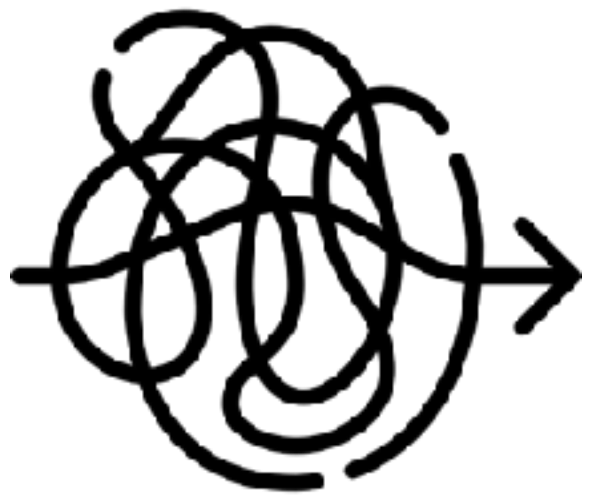
Delta Encoding



Restarts

Page Hash Index

LZ77

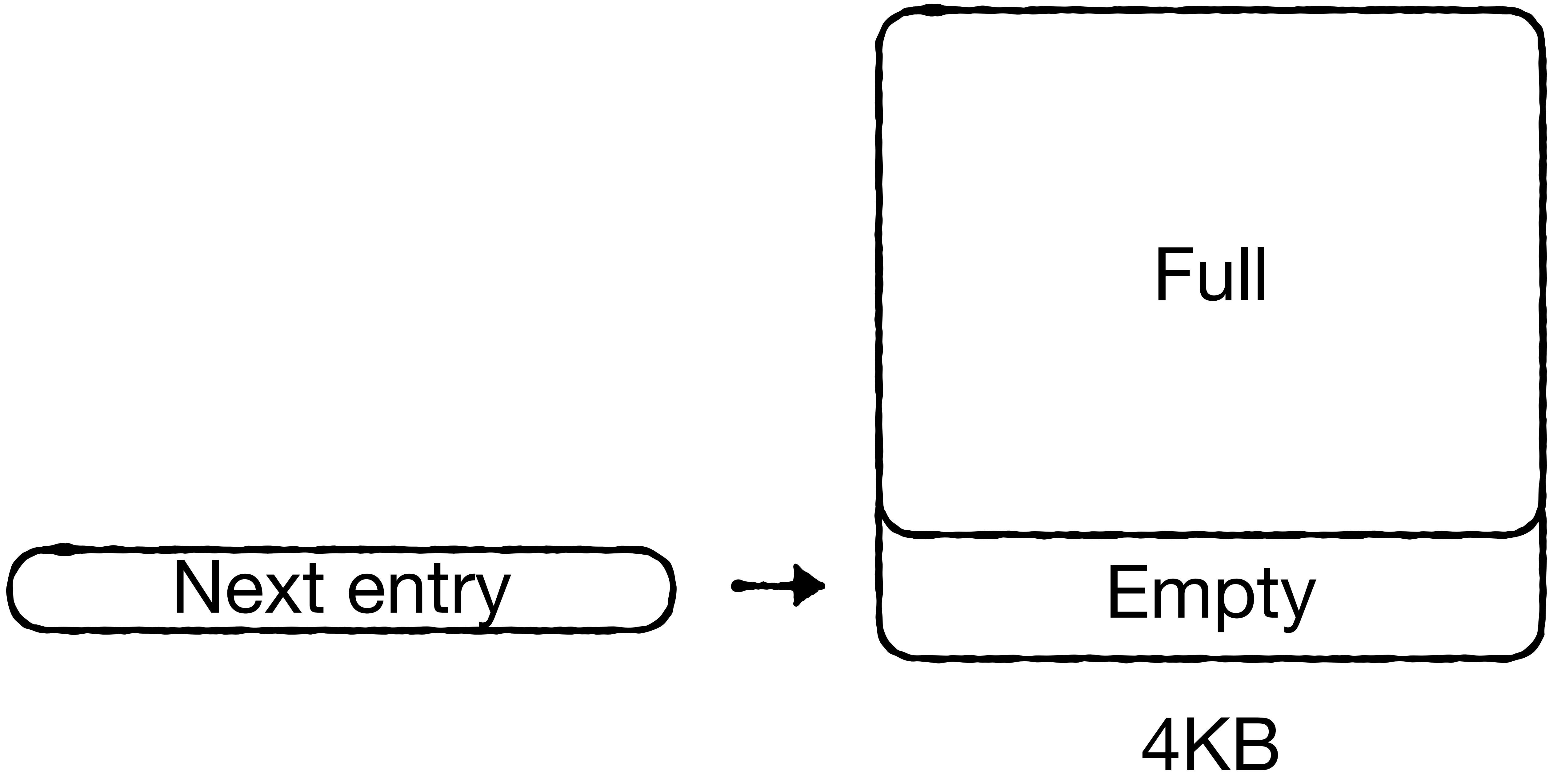


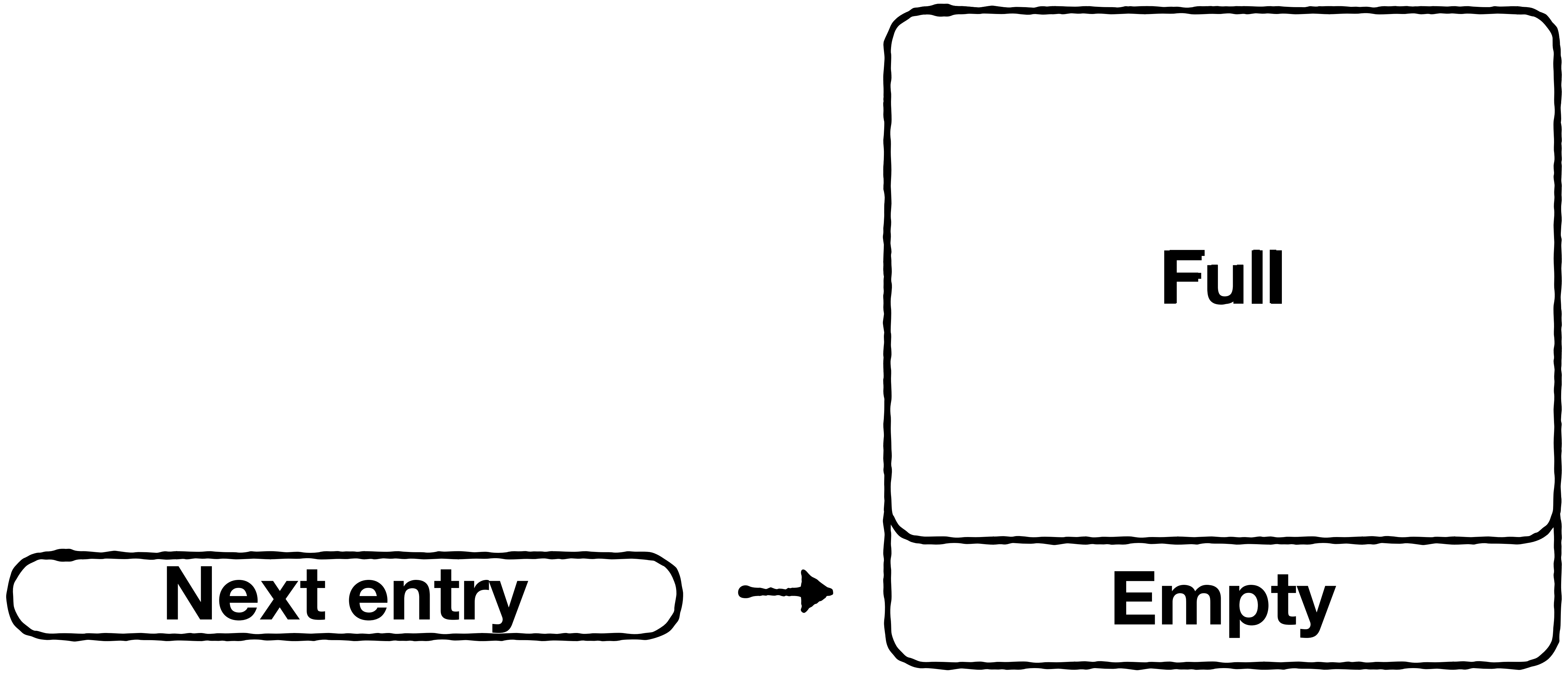
Huffman coding

Dictionary training

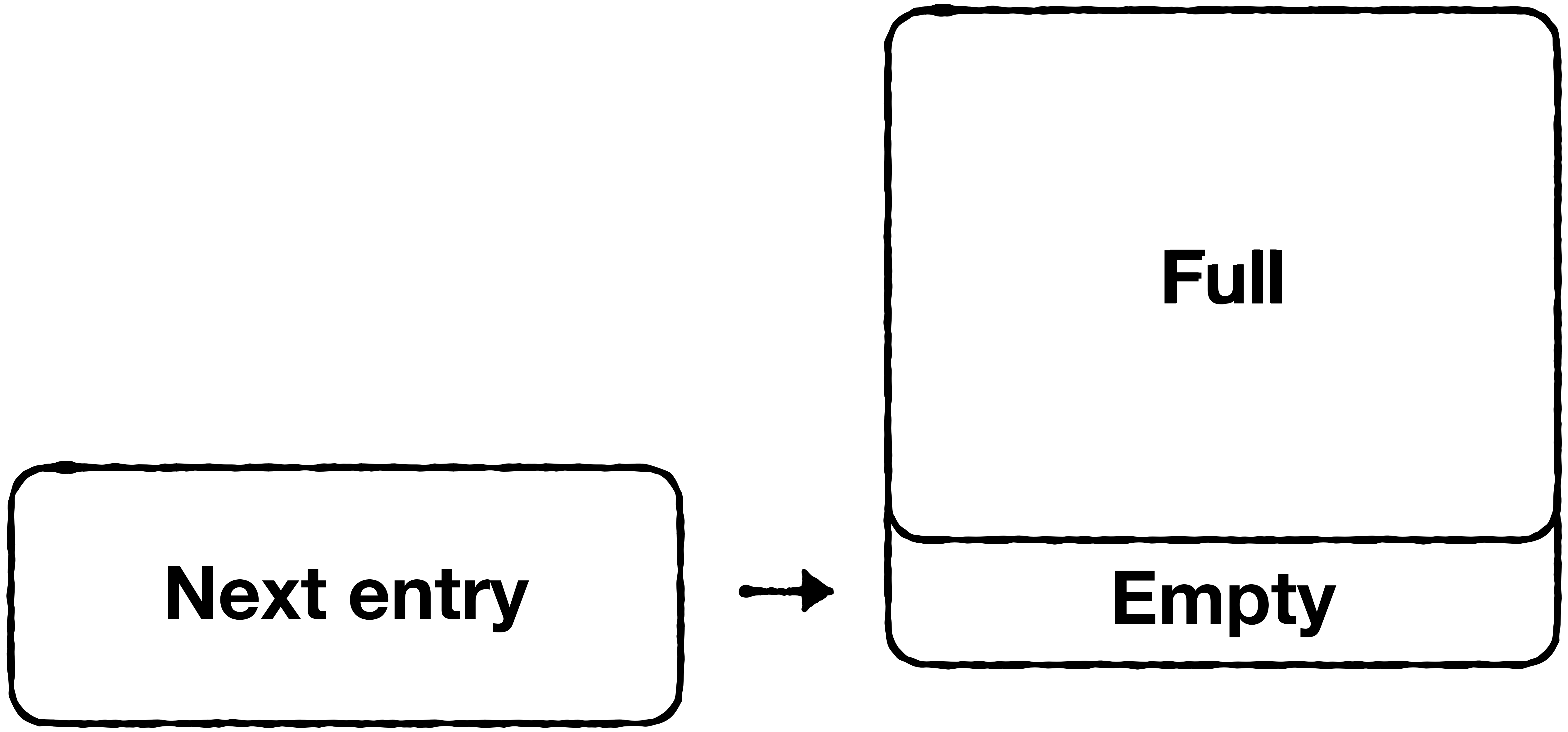
Page Sizing

How to pack var-length entries into a page?



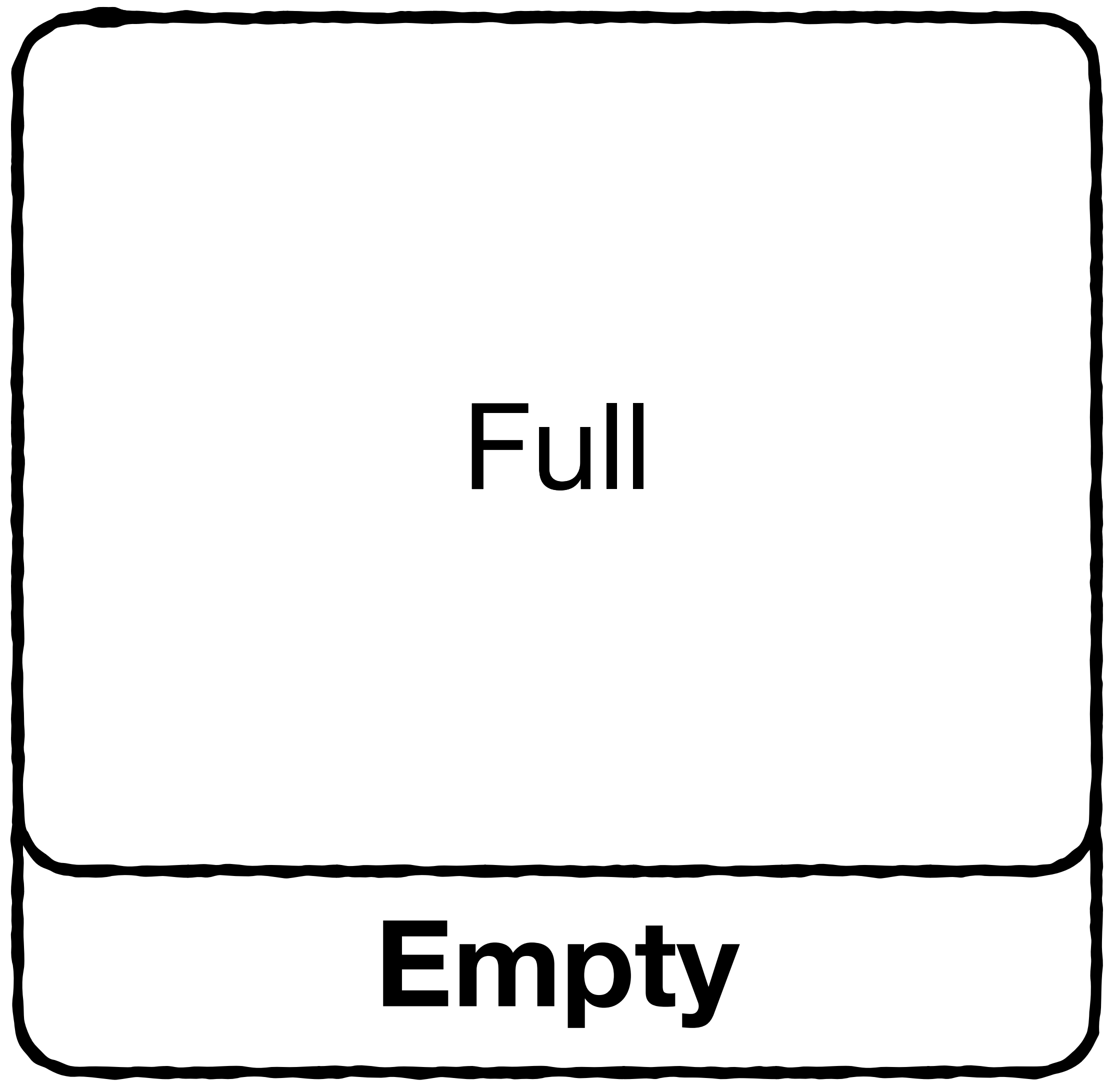


As long as next entry fits, add it

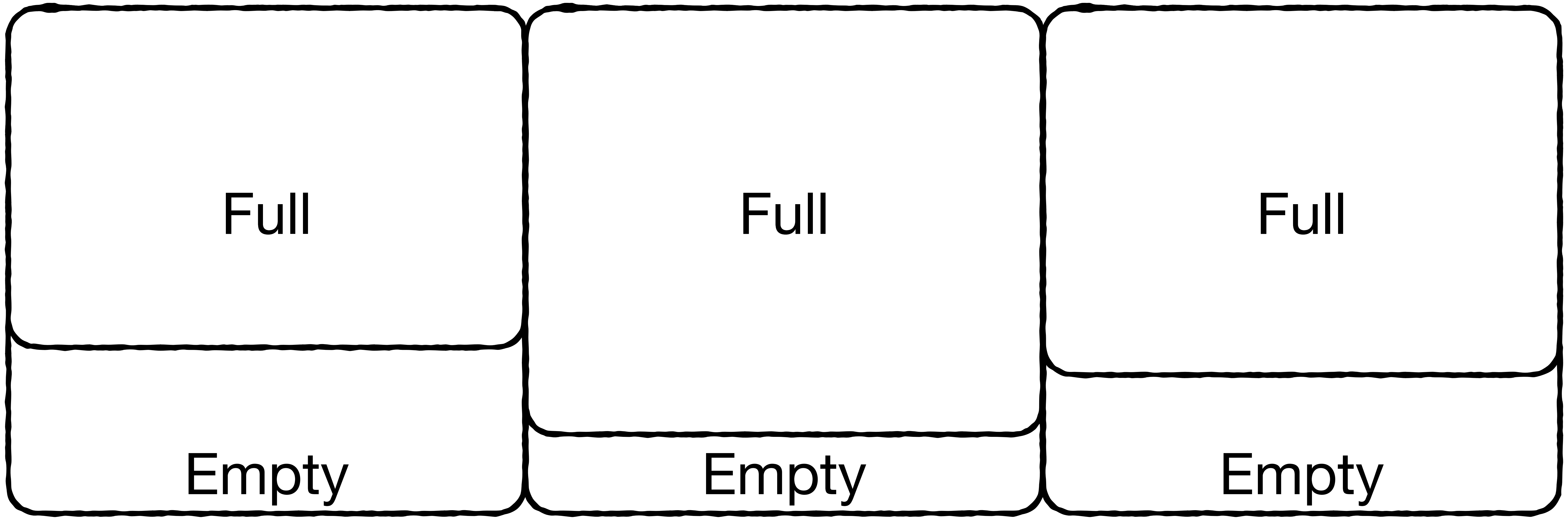


But what if it’s too big?

Option 1: Seal page and waste space



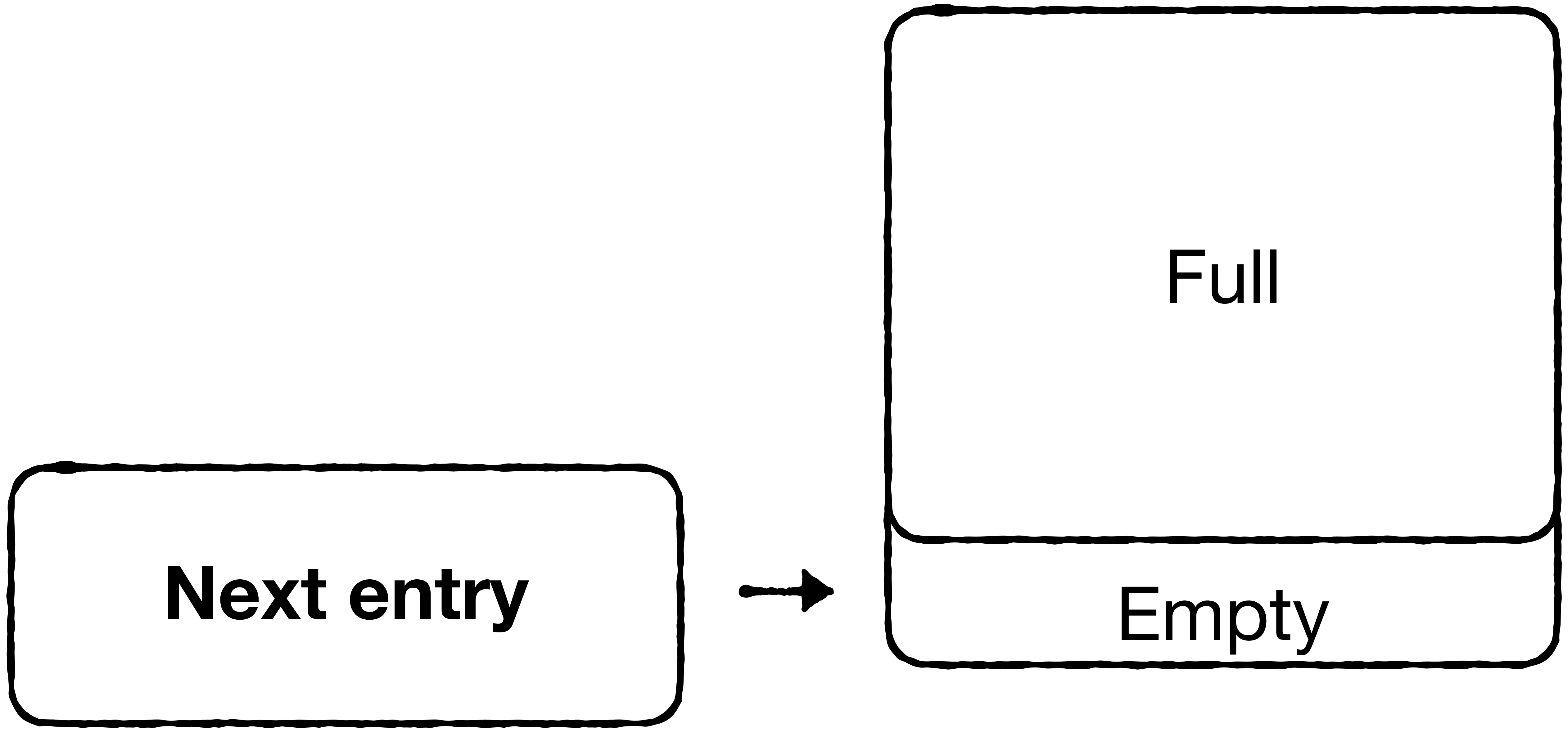
Option 1: Seal page and waste space



pages are aligned on SSD. Each can be read with one I/O

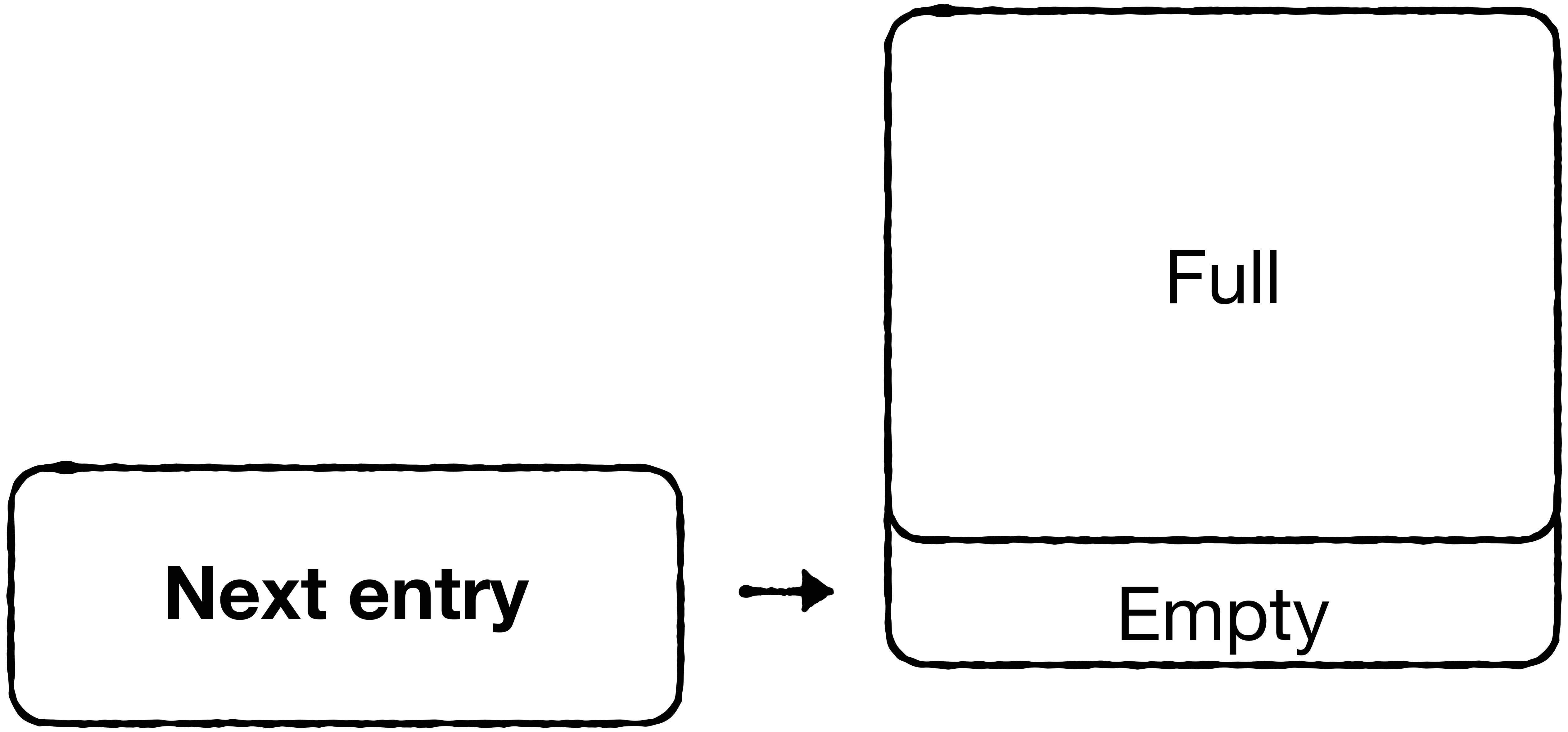
waste some space

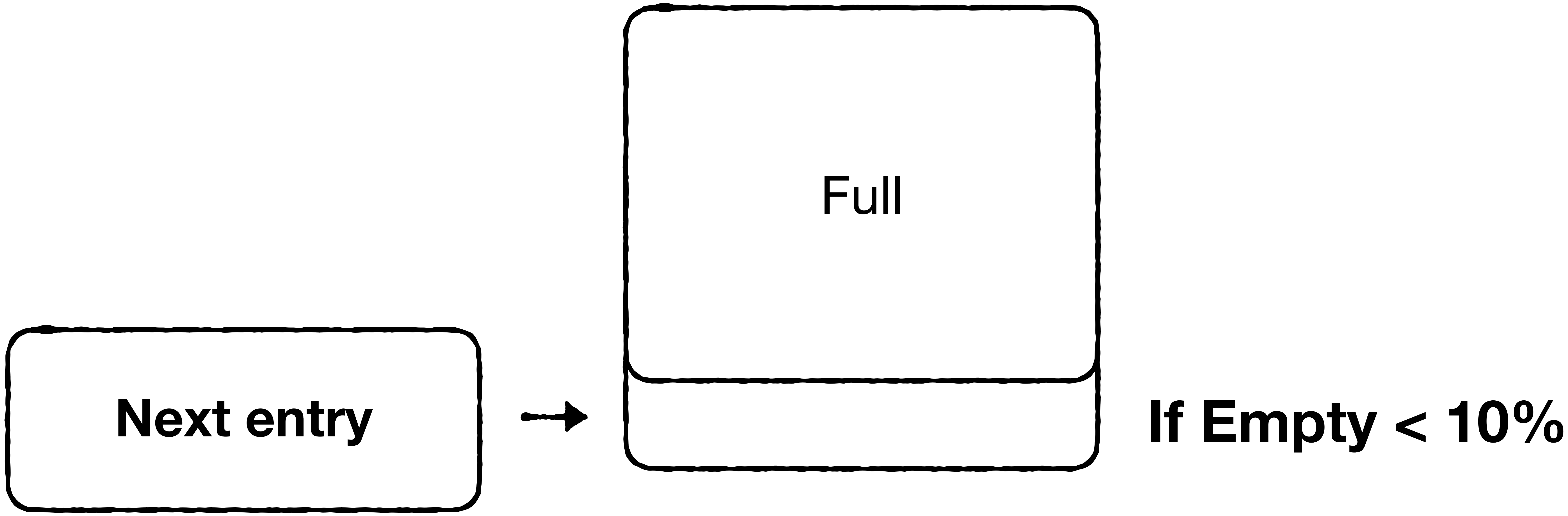
Option 2: allow variable-sized pages

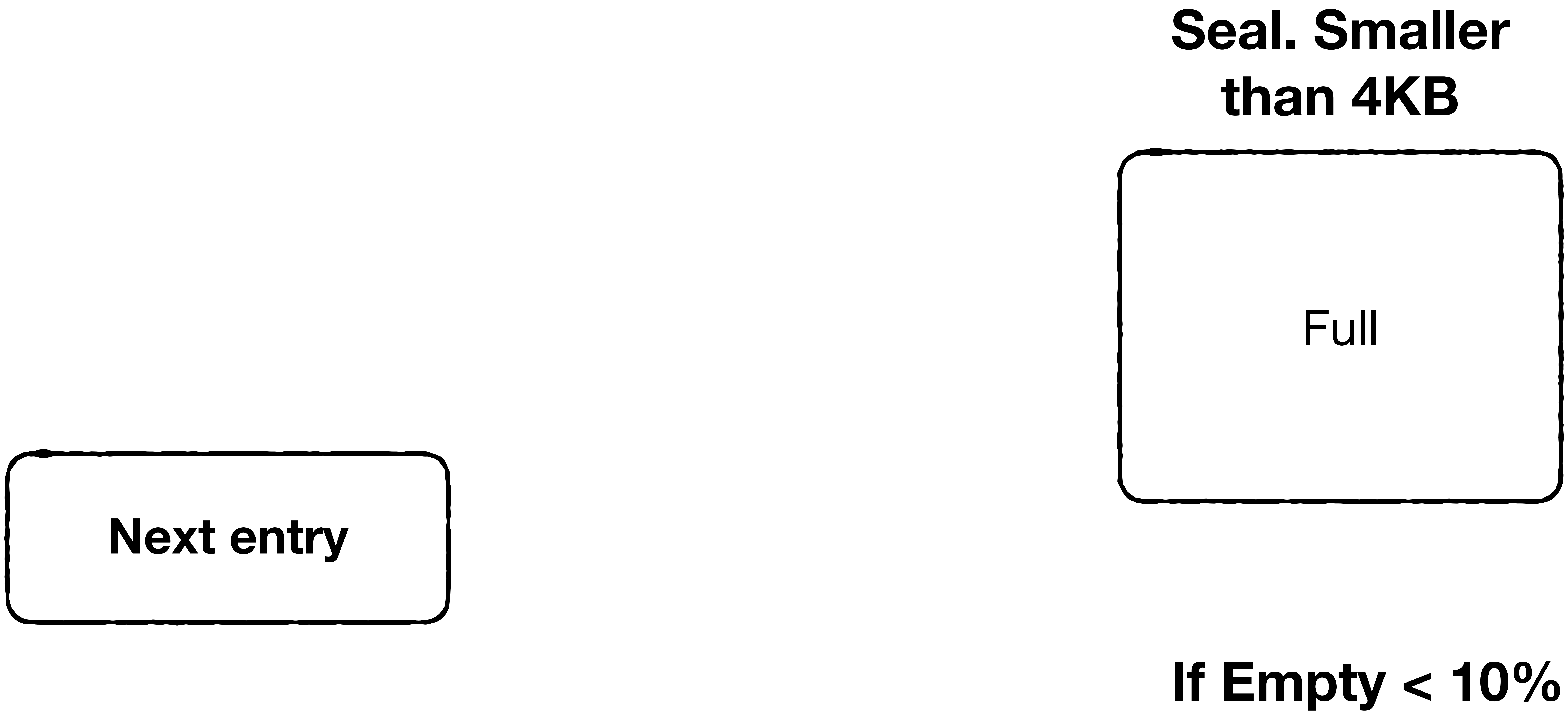


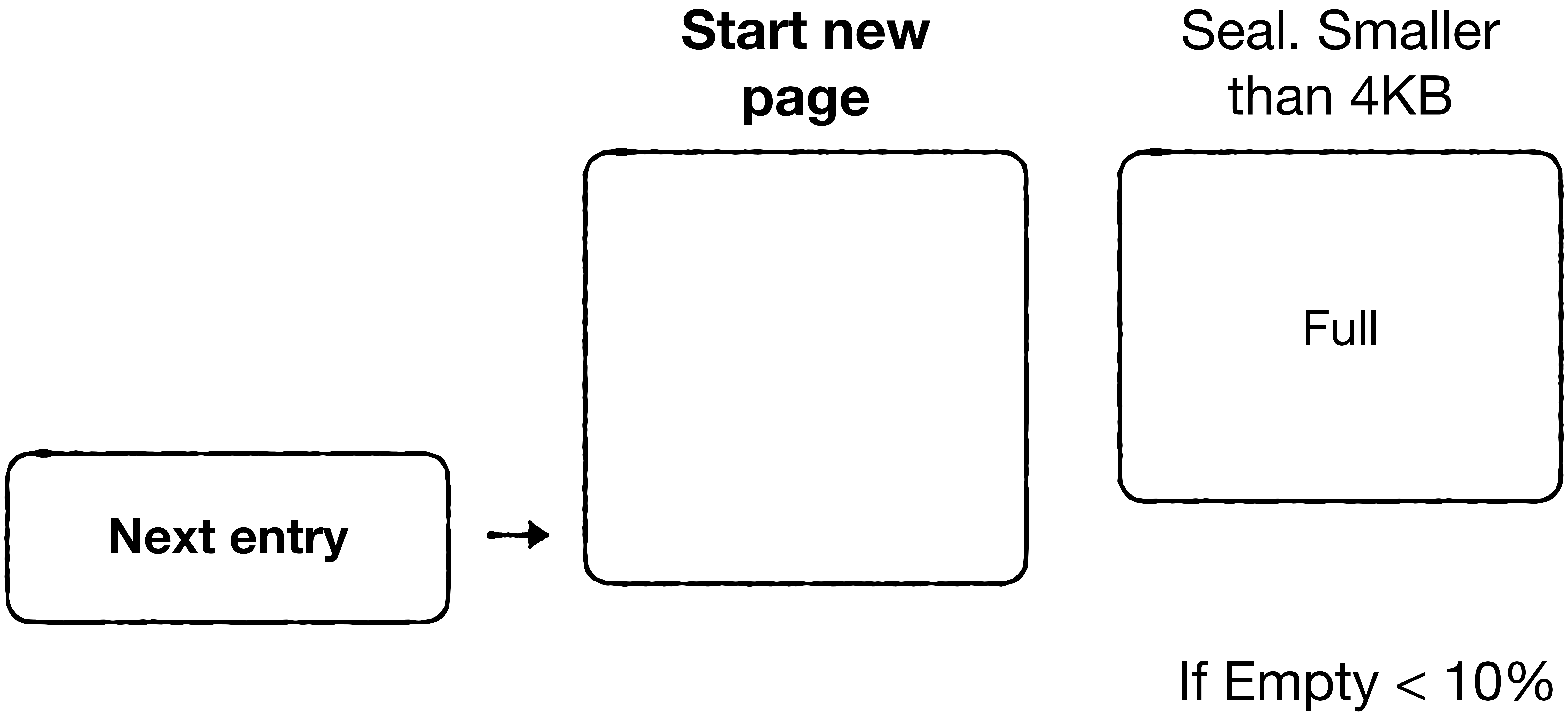
Option 2: allow variable-sized pages

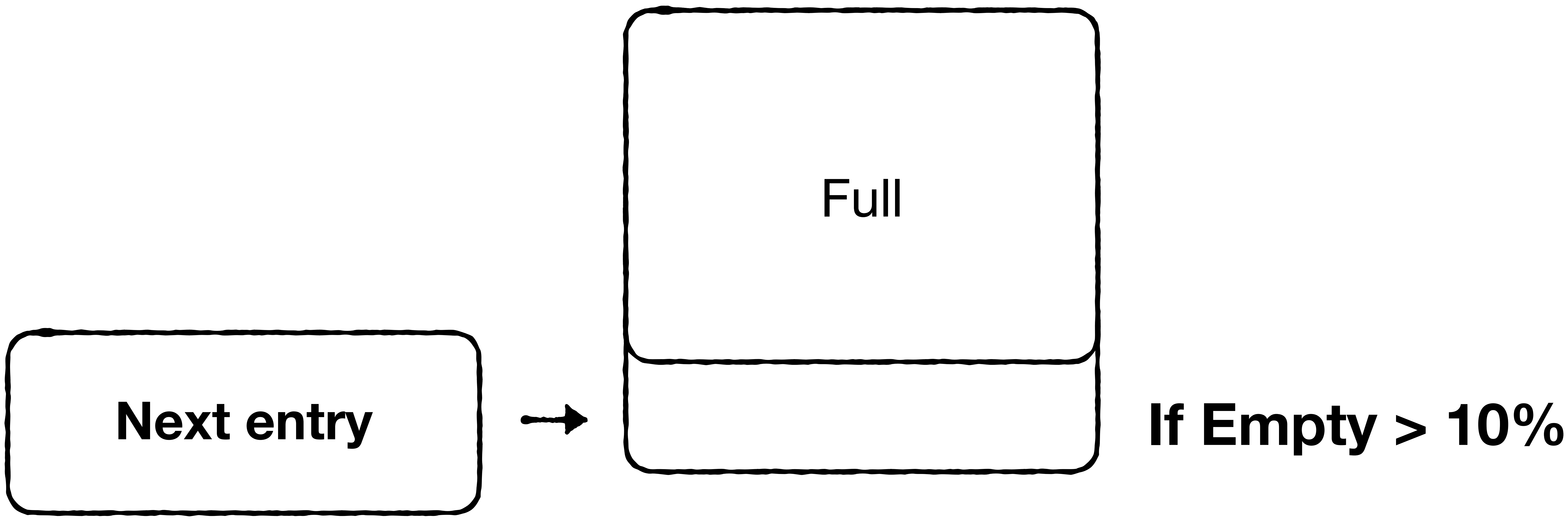
block_size_deviation: 10%



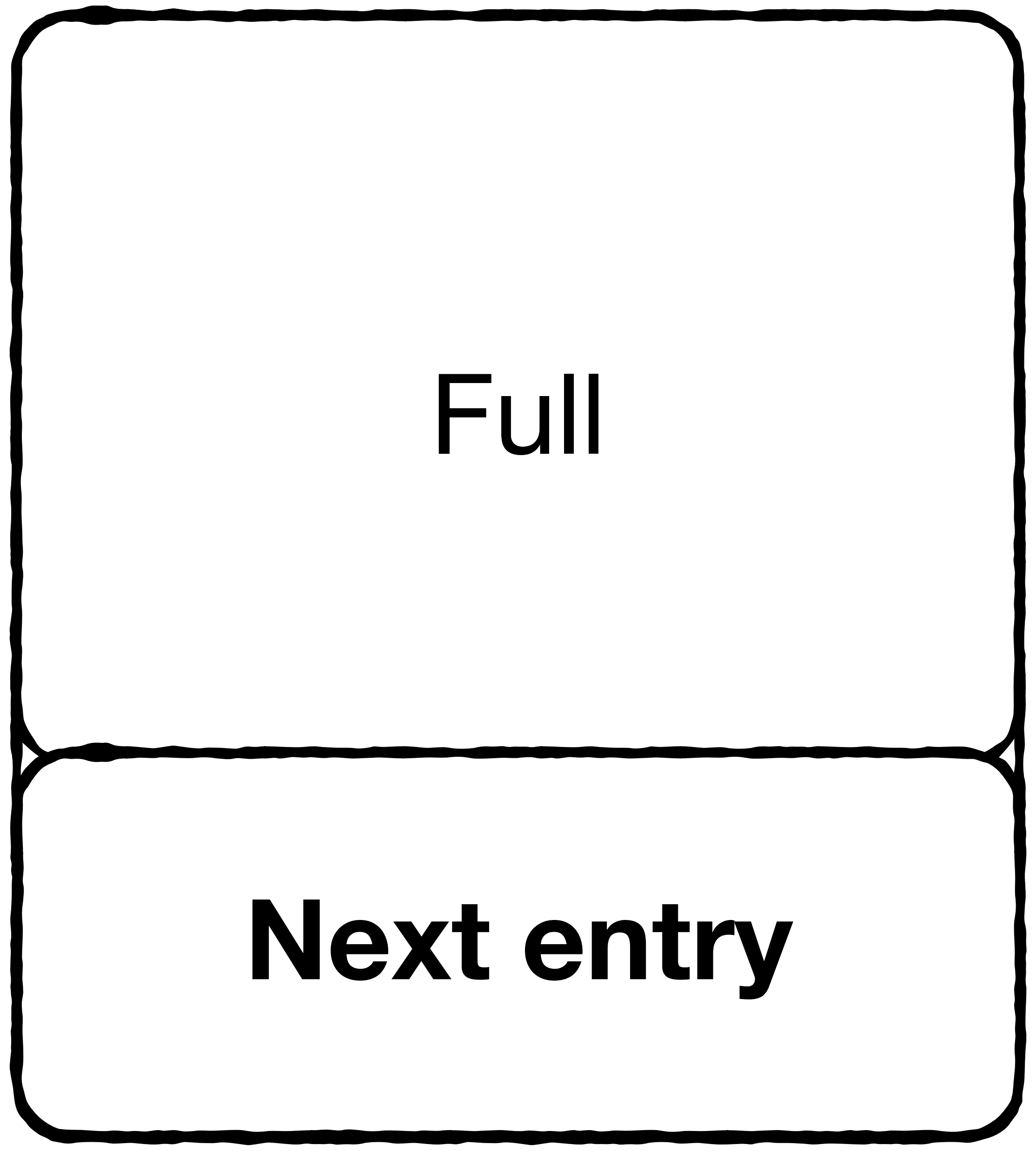

Full
Next entry
If Empty < 10%


Seal. Smaller than 4KB
Full
Next entry
If Empty < 10%


Start new page
Seal. Smaller than 4KB
Full
Next entry
If Empty < 10%

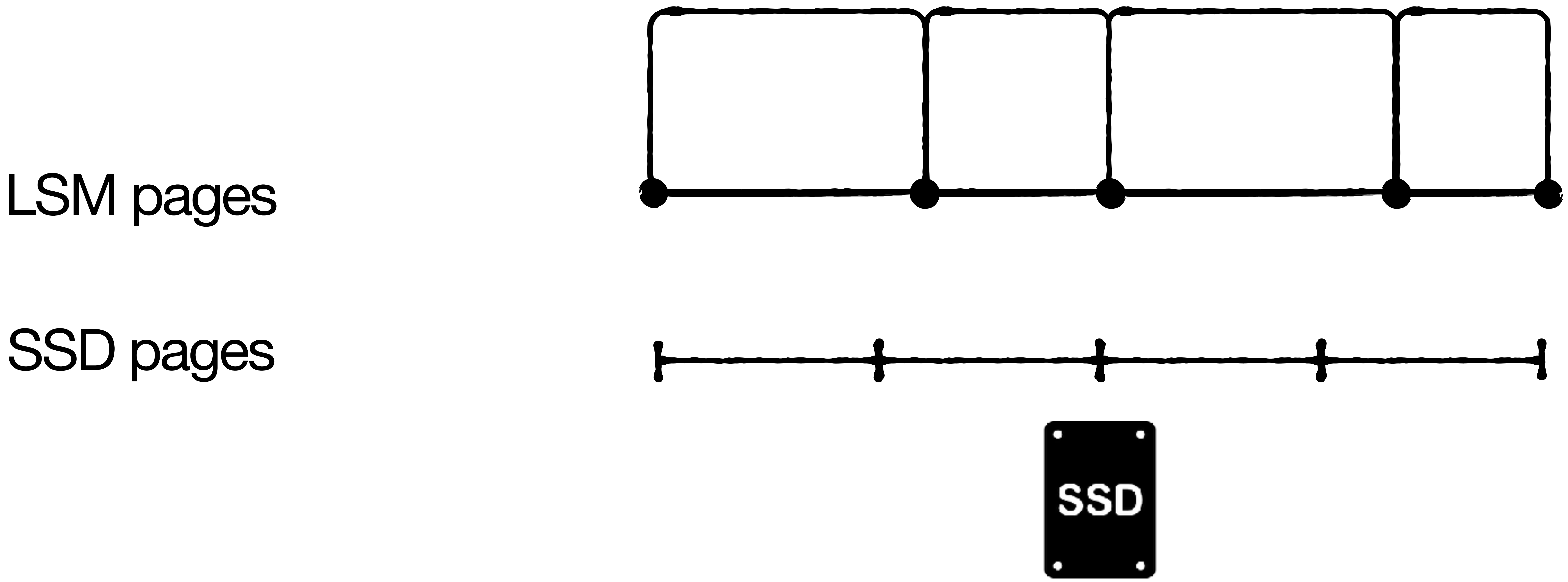

Next entry
Full
If Empty > 10%

Let the page be larger than 4KB



If Empty > 10%

Hence, RocksDB pages are ≈4KB but exhibit variation



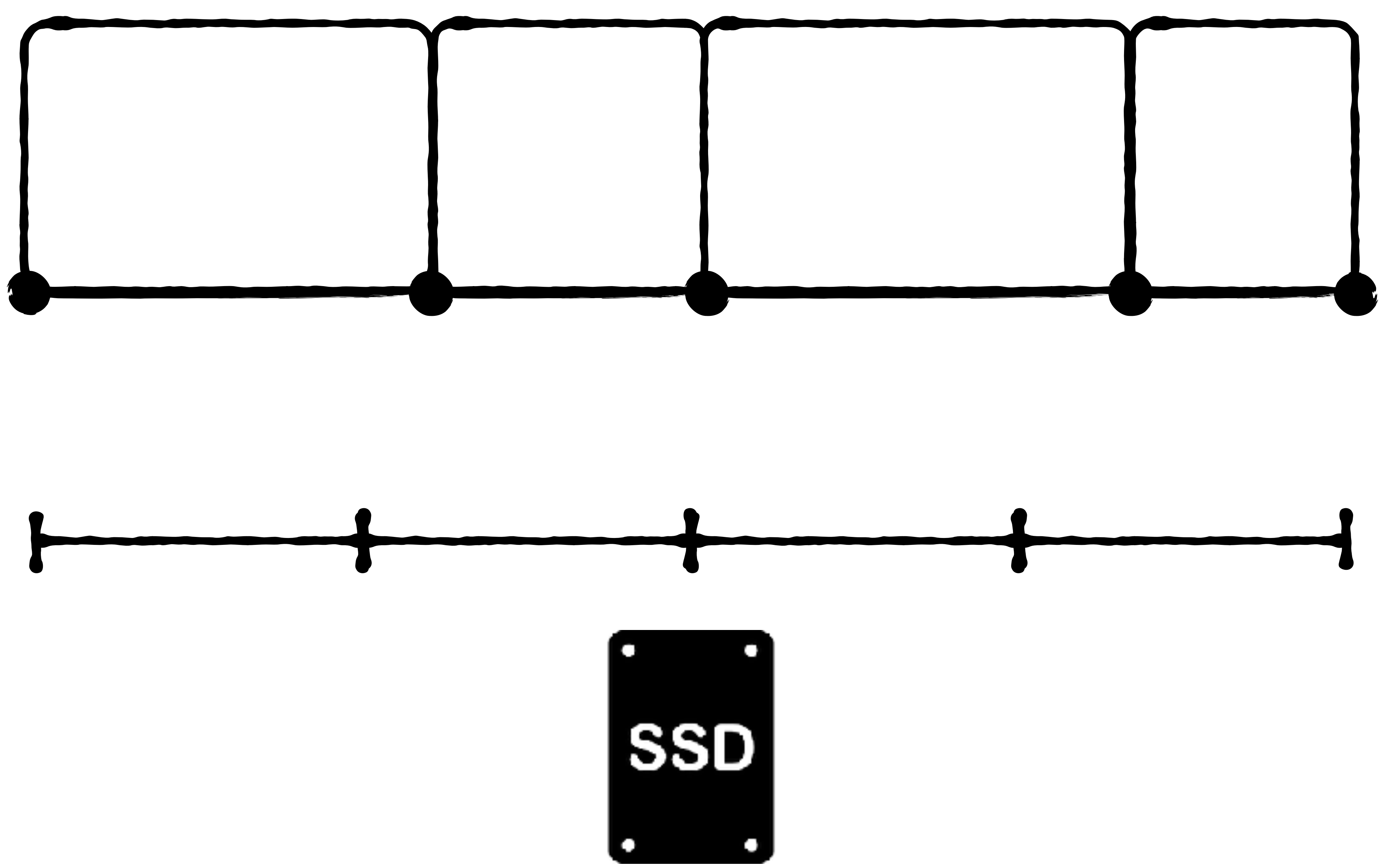
Hence, RocksDB pages are ≈4KB but exhibit variation

no wasted space

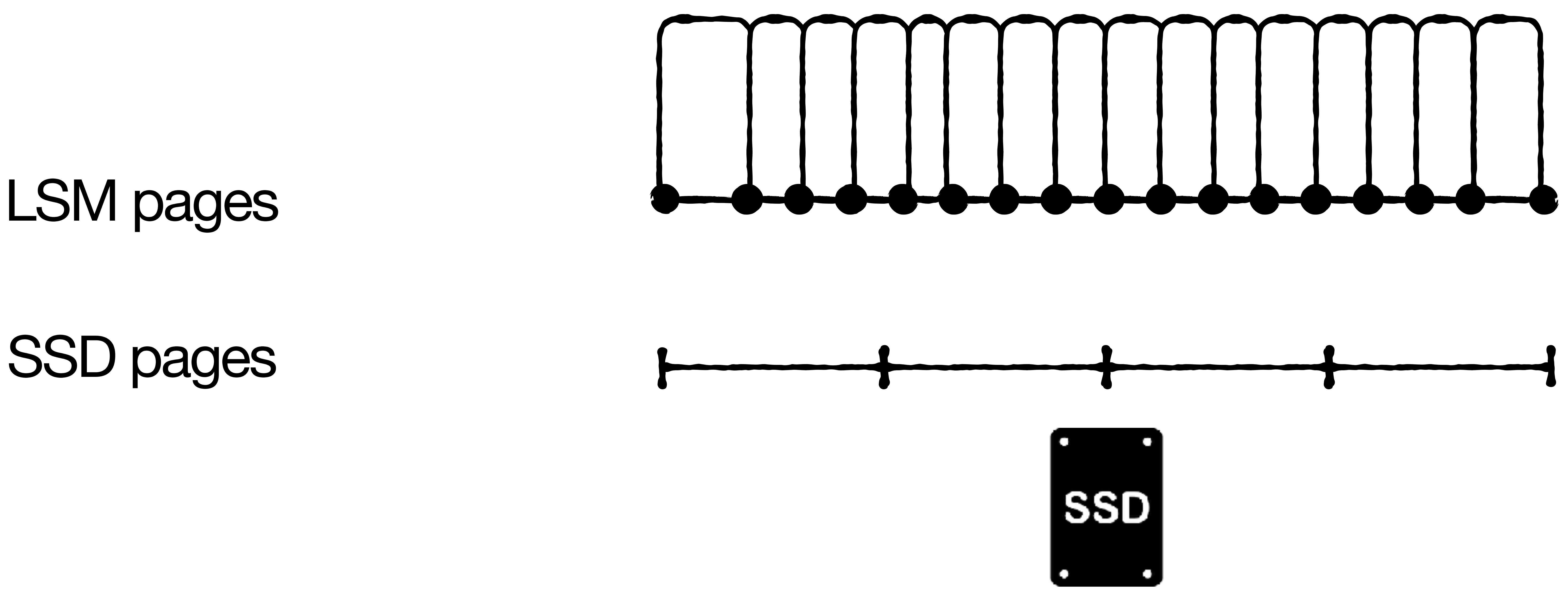
no SSD page alignment -> possibly 2x read-amplification

LSM pages

SSD pages

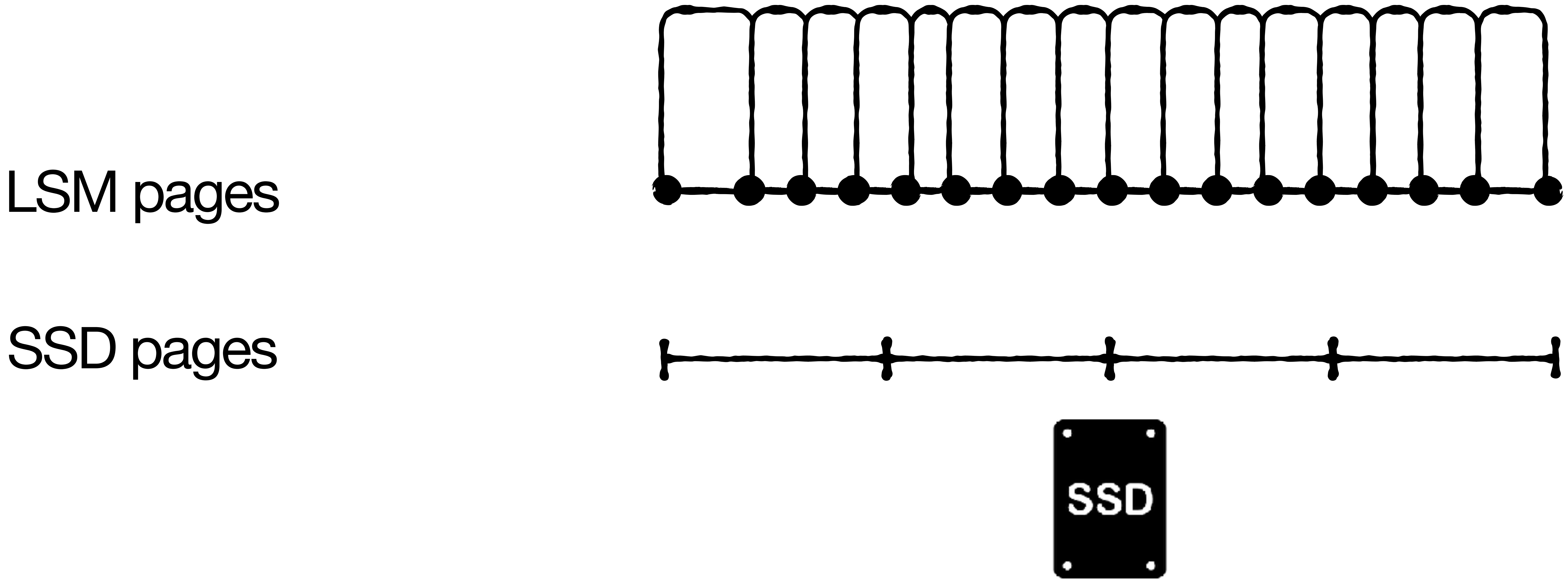


But suppose compression rate is X (e.g., X=4)



But suppose compression rate is X (e.g., X=4)

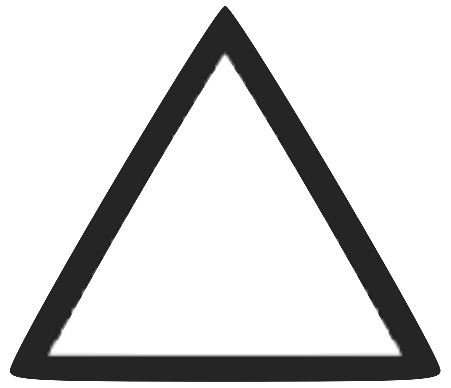
only read 1 + 1/X pages on avg. as some DB pages are fully contained in SSD pages



Agenda

Page Sizing

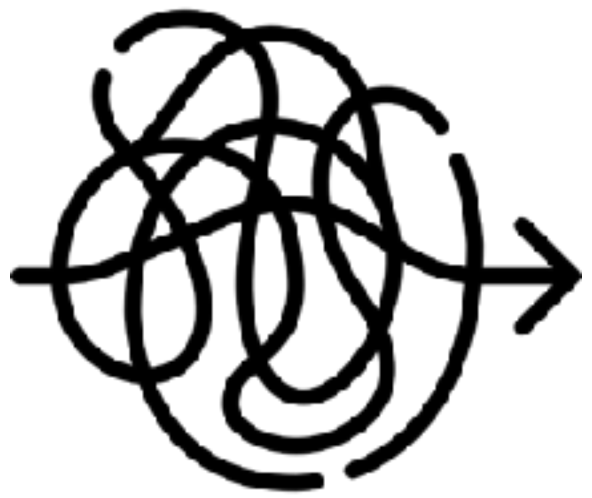
Delta Encoding



Restarts

Page Hash Index

LZ77

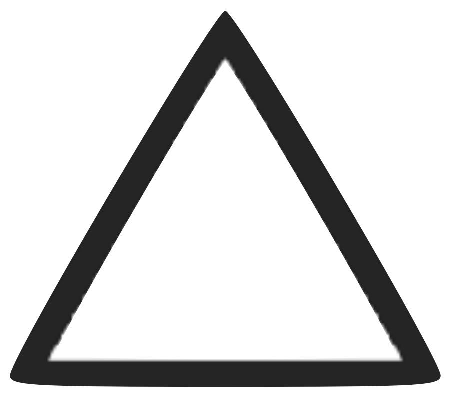


Huffman coding

Dictionary training

Delta Encoding

Remove common prefixes of adjacent sorted keys



Delta Encoding

Remove common prefixes of adjacent sorted keys

△

Now with strings as opposed to integers :)

AAABCD AAABCEF AAABD

AAABCD **AAABC**EF AAABD

5 bytes match

AAABCD **AAAB**CEF **AAAB**D

4 bytes match

How to encode common prefixes?

AAABCD AAABCEF AAABD

For each key add two fields: **shared_bytes S**

unshared_bytes U

AAABCD AAABCEF AAABD

For each key add two fields: **shared_bytes S**

unshared_bytes U

S **U** AAABCD **S** **U** AAABCEF **S** **U** AAABD

For each key add two fields: shared_bytes S

unshared_bytes U

0 6 AAABCD **5 2** AAABCEF **4 1** AAABD

0 **6** AAABCD **5** **2** **AAABC**EF **4** **1** **AAAB**D

Remove redundant prefixes

0 **6** AAABCD **5** **2** **EF** **4** **1** **D**

Any issues?

0 6 AAABCD **5 2 EF** **4 1 D**

How to encode the un/shared fields?

0 **6** AAABCD **5** **2** **EF** **4** **1** **D**

How to encode the un/shared fields?

Fixed-sized integer (L bytes)?

0 **6** AAABCD **5** **2** **EF** **4** **1** **D**

How to encode the un/shared fields?

Fixed-sized integer (L bytes)?

2L bytes per key & can encode keys of at most 2^L bytes

0 6 AAABCD   5 2 EF   4 1 D

How to encode the un/shared fields?

Fixed-sized integer (L bytes)?

2L bytes per key & can encode **keys of at most 2^L bytes**

Wasteful for small keys

Restrictive

0 6 AAABCD

5 2 EF

4 1 D

Solutions?

How to encode the un/shared fields?

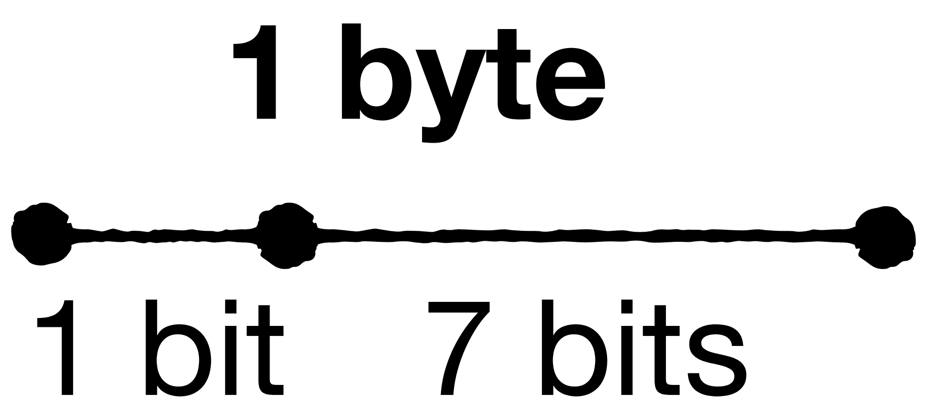
Fixed-sized integer (L bytes)?

2L bytes per key & can encode **keys of at most 2^L bytes**

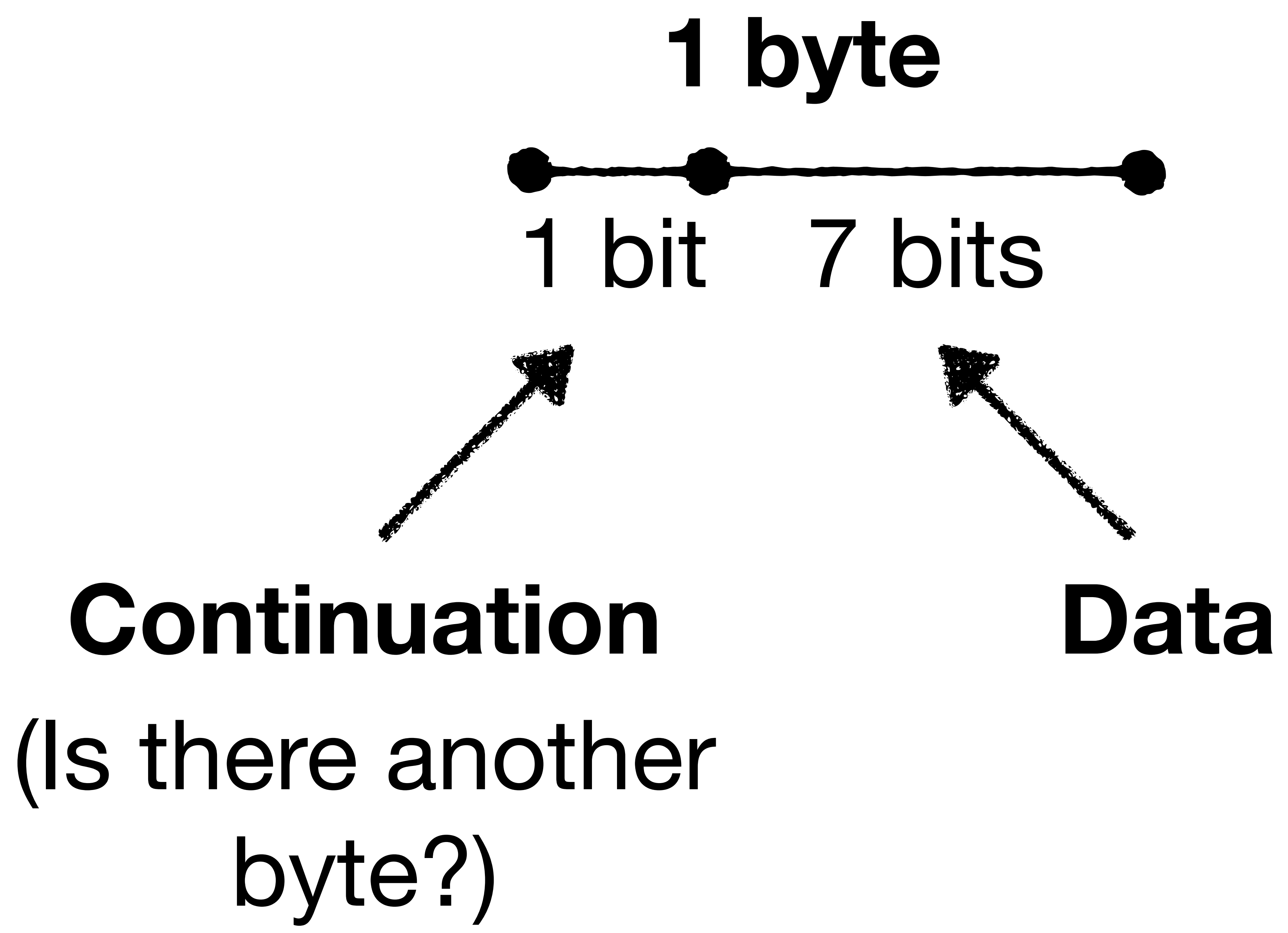
Wasteful for small keys **Restrictive**

0 6 AAABCD 5 2 EF 4 1 D

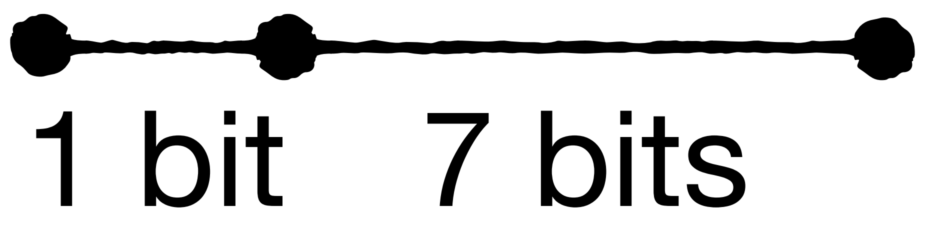
Varint



Varint



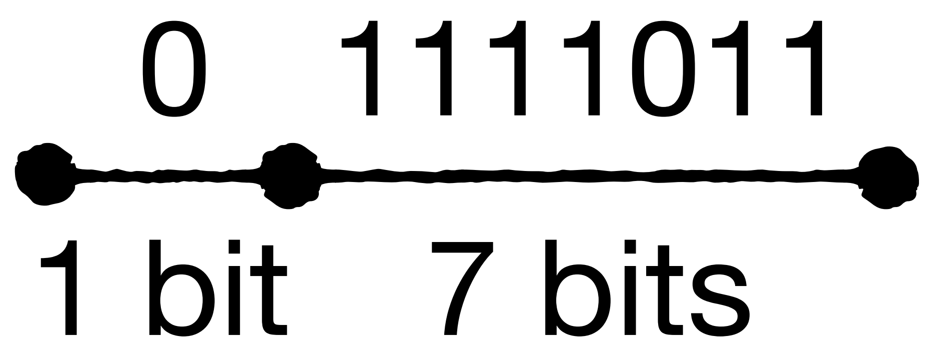
E.g., encode **123**, in binary: **1111011**



E.g., encode 123, in binary:

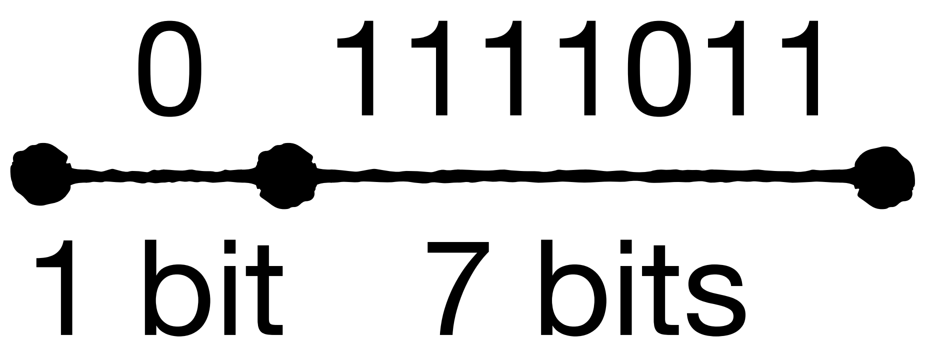
$$\underbrace{\mathbf{0}}_{\text{1 bit}}\ \underbrace{\mathbf{1111011}}_{\text{7 bits}}$$

E.g., encode 123, in binary:

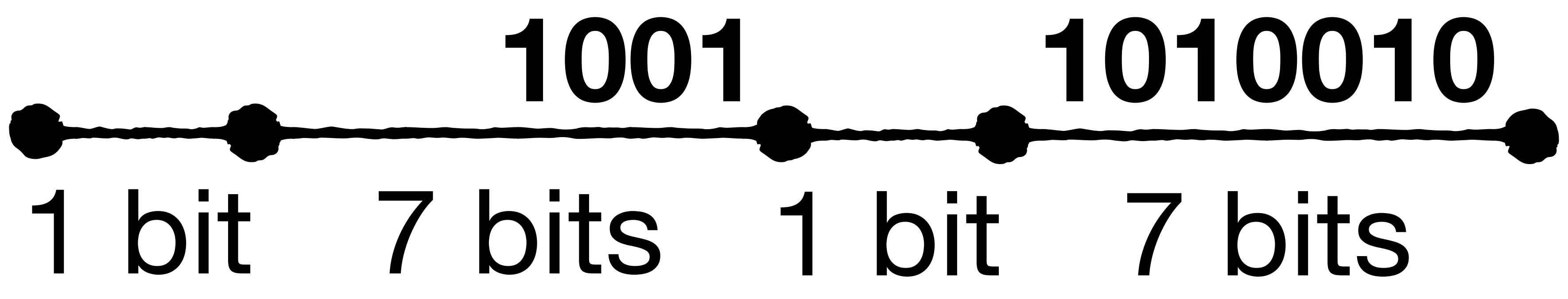


E.g., encode 1234, in binary: 10011010010

E.g., encode 123, in binary:

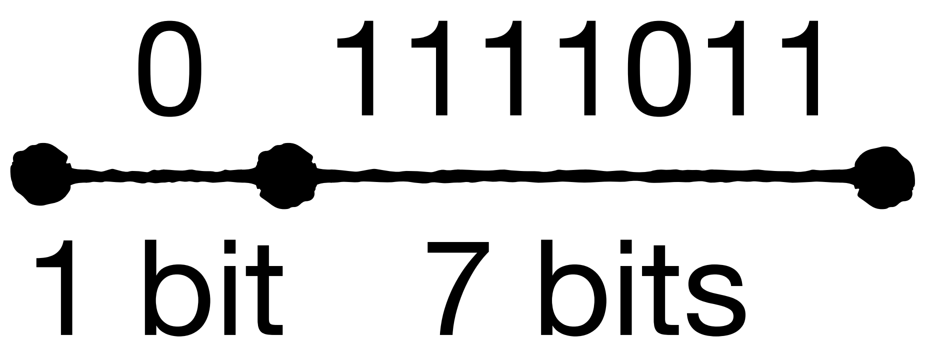


E.g., encode 1234, in binary:

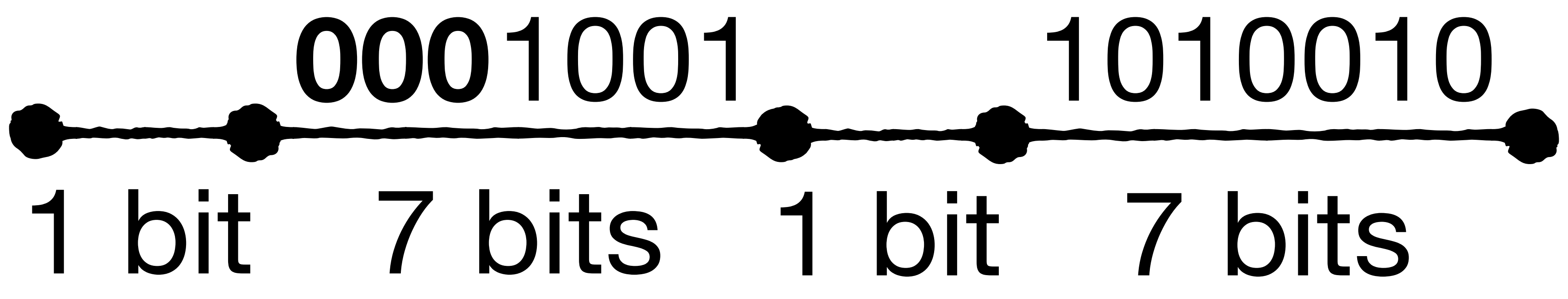


Split into 7 bit chunks

E.g., encode 123, in binary:



E.g., encode 1234, in binary:



Add padding

E.g., encode 123, in binary:

0	1111011
1 bit	7 bits

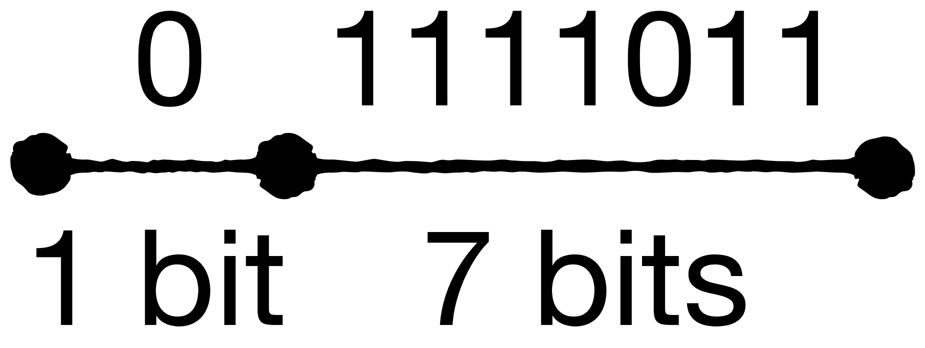
E.g., encode 1234, in binary:

1	0001001	**0**	1010010
1 bit	7 bits	1 bit	7 bits

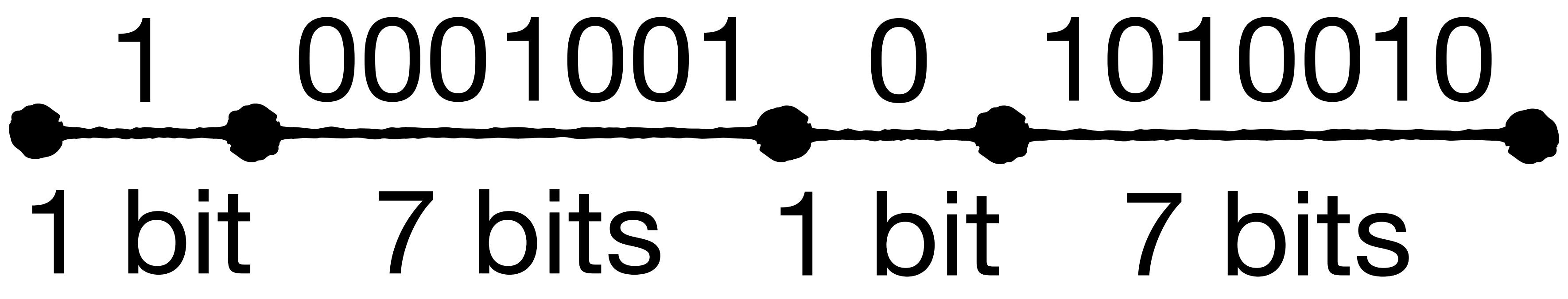
Add continuation bits

Lose 1/8 space but get flexibility

E.g., encode 123, in binary:



E.g., encode 1234, in binary:



Back to the example:

0 **6** AAABCD **5** **2** EF **4** **1** D

Back to the example:

0 6 AAABCD **5 2** EF **4 1** D

Encode as varints

1 varint byte represents count of up to $2^7 = 128$

0 6 AAABCD 5 2 EF 4 1 D

Encode as varints

1 varint byte represents count of up to $2^7 = 128$

Keys are typically smaller than 128 bytes

0 6 AAABCD 5 2 EF 4 1 D

1 varint byte represents count of up to $2^7 = 128$

Keys are typically smaller than 128 bytes

Typically 1 byte each

0 6 AAABCD **5 2** EF 4 1 D

1 varint byte represents count of up to $2^7 = 128$

Keys are typically smaller than 128 bytes

Typically 1 byte each

0 6 AAABCD **5 2** EF 4 1 D

But can still support keys larger than 128 bytes :)

KV-Pair Encoding

shared_bytes	unshared_bytes	value_bytes	key_delta	value
varint	varint	varint	char[unshared_bytes]	char[value_bytes]

KV-Pair Encoding

shared_bytes	**unshared_bytes**	value_bytes	**key_delta**	value
varint	varint	varint	char[unshared_bytes]	char[value_bytes]

Focused on these so far

Entire Key-Value Pair Encoding

shared_bytes	unshared_bytes	**value_bytes**	key_delta	**value**
varint	varint	**varint**	char[unshared_bytes]	**char[value_bytes]**

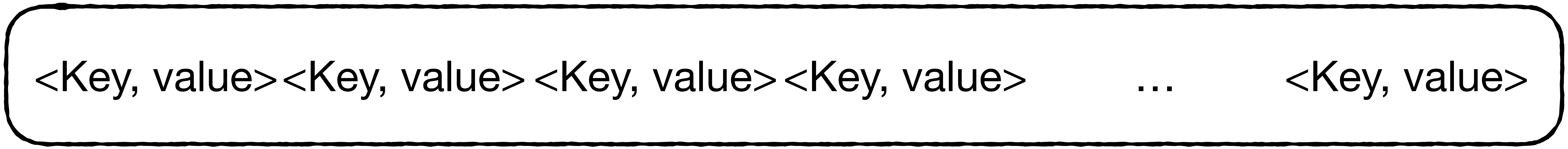
Value size also encoded as varint, but not delta encoded

Entire Key-Value Pair Encoding

shared_bytes	unshared_bytes	value_bytes	key_delta	value
varint	varint	varint	char[unshared_bytes]	char[value_bytes]

Any performance issues?

potentially decode whole page to answer a query



4KB page

potentially decode whole page to answer a query

<Key, value><Key, value><Key, value><Key, value> … **<Key, value>**

Suppose a query targets the last KV-pair in a 4 KB page

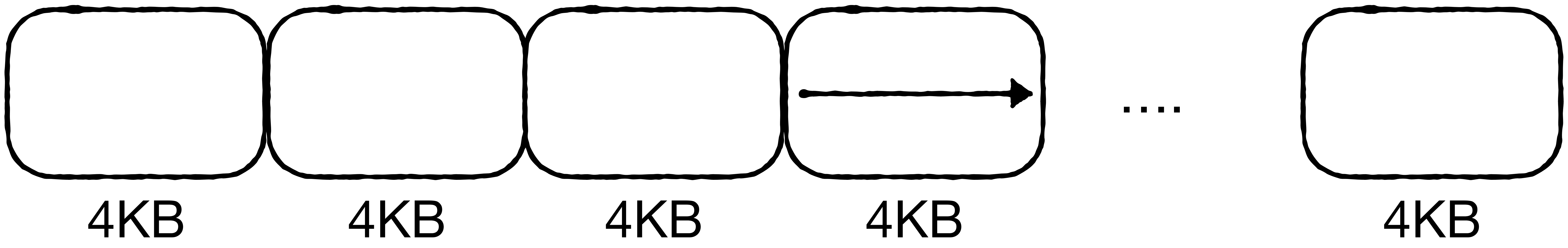
We must potentially decode an entire block to answer a query

<Key, value><Key, value><Key, value><Key, value> … <Key, value>

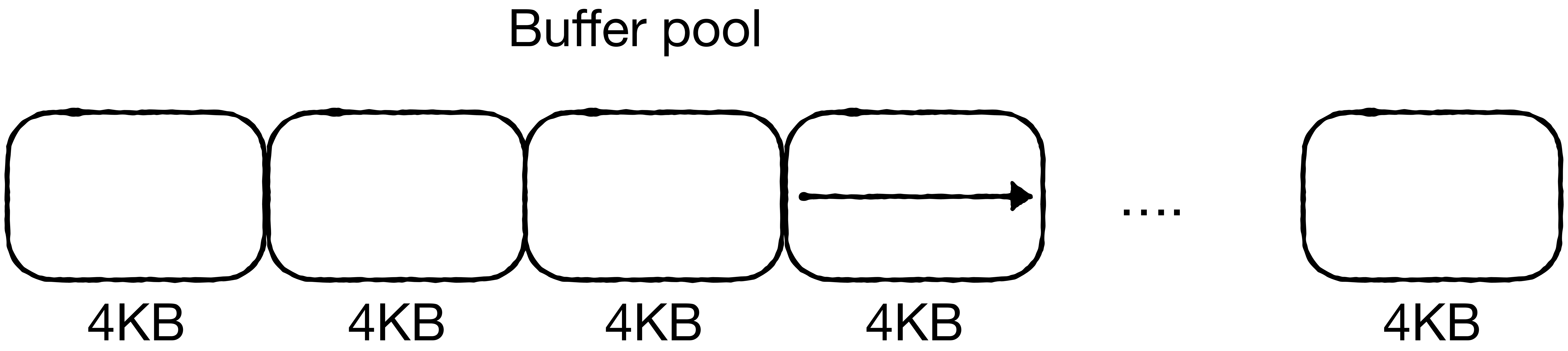
Must traverse and decode whole page due to delta encoding

Suppose a query targets the last KV-pair in a 4 KB page

Buffer pool



As we maintain this page format in the buffer pool, it may significantly slow down queries to hot keys

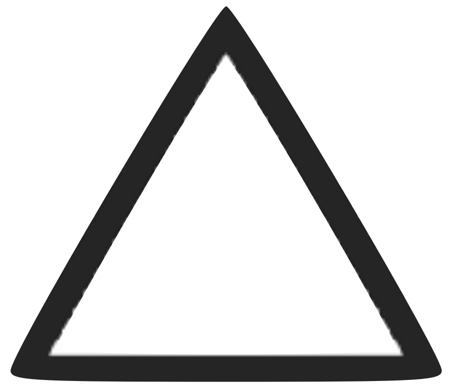


As we maintain this page format in the buffer pool, it may significantly slow down queries to hot keys

Solutions?

Page Sizing

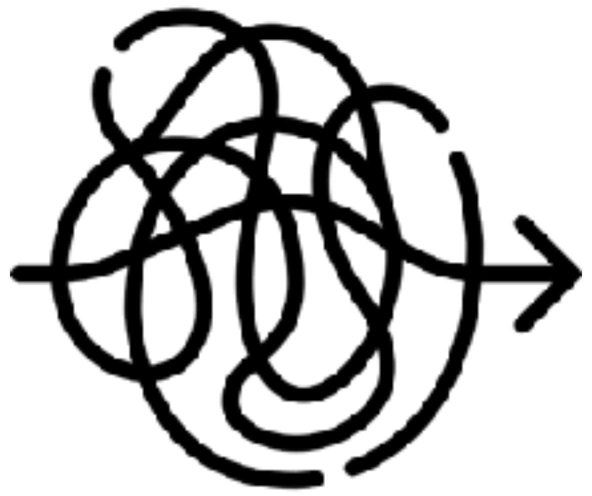
Delta Encoding



Restarts

Page Hash Index

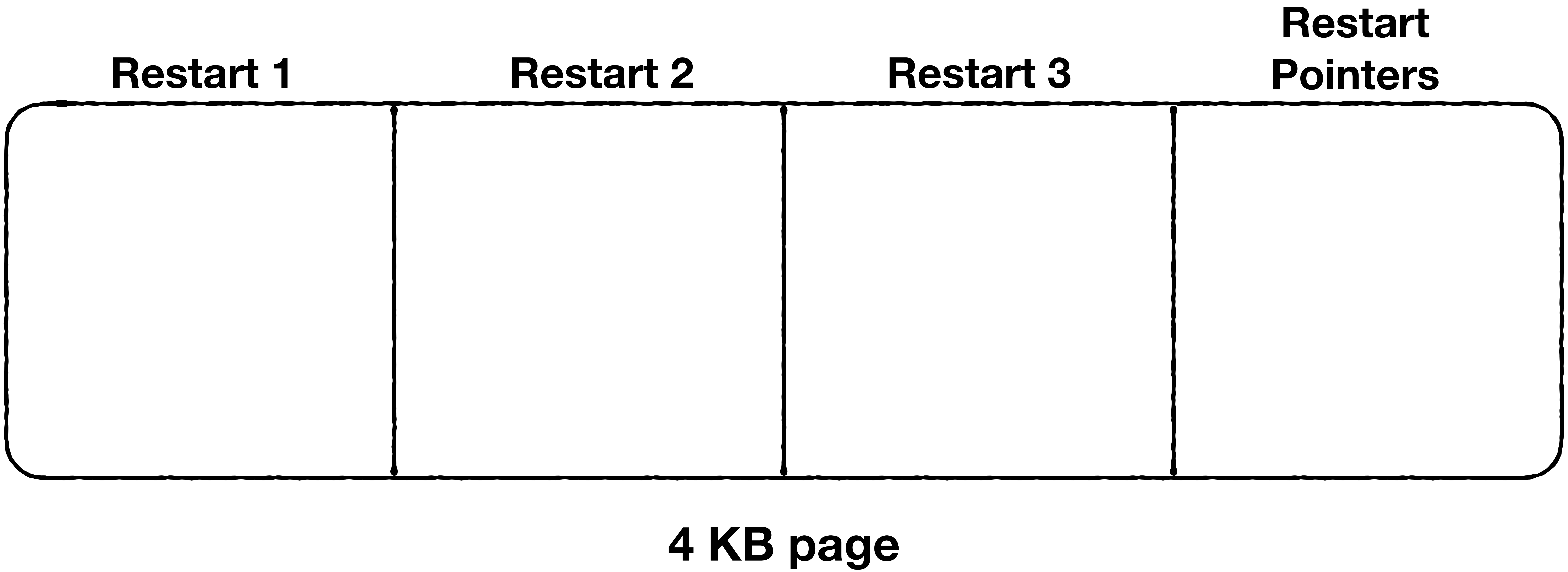
LZ77



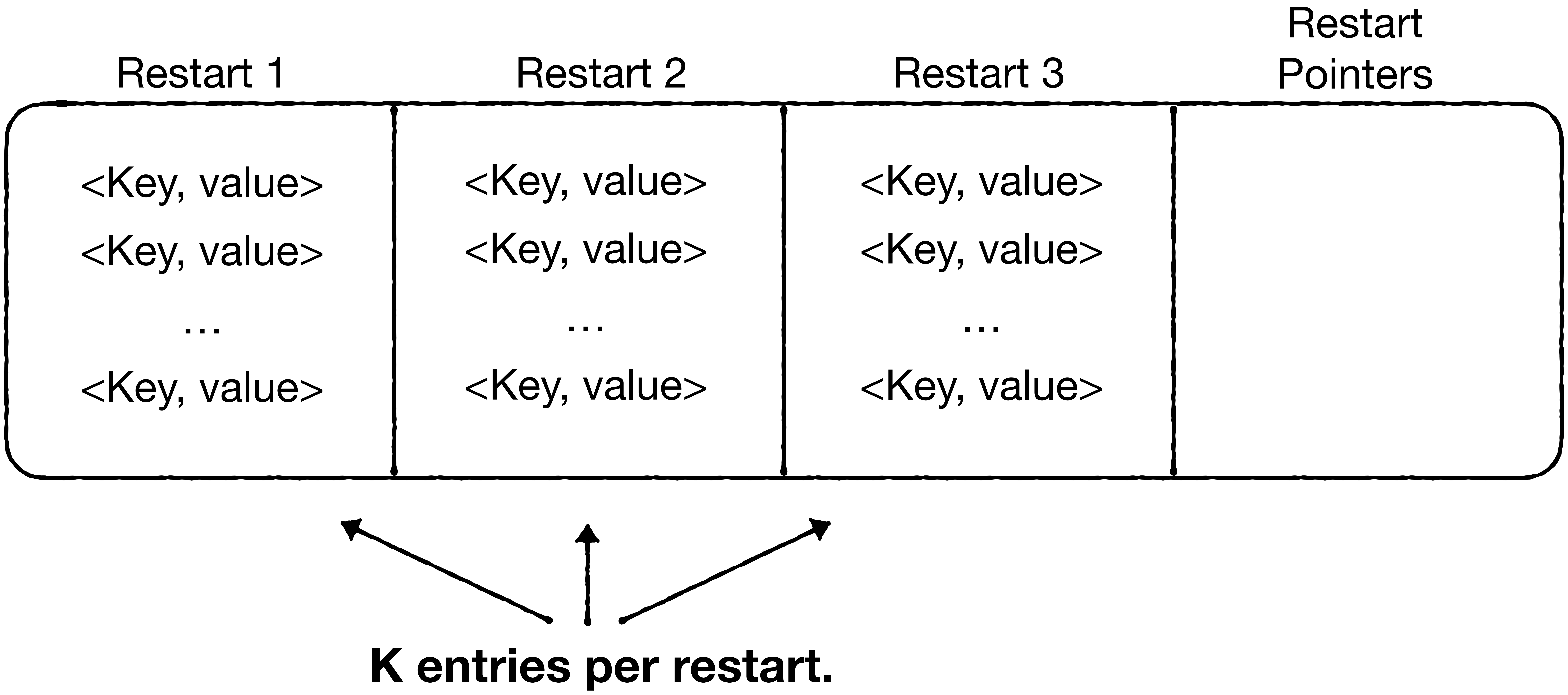
Huffman coding

Dictionary training

Restarts

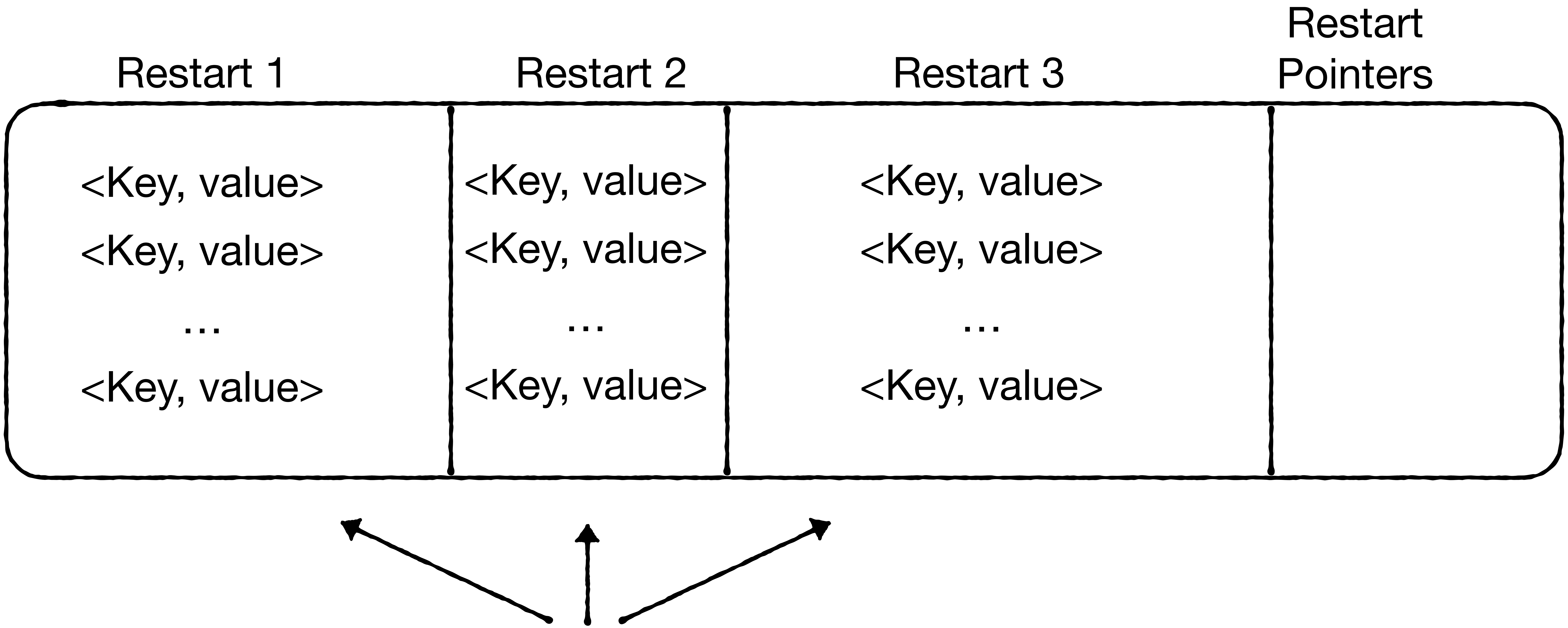


Restarts



Restart 1
Restart 2
Restart 3
Restart Pointers
<Key, value>
<Key, value>
…
<Key, value>
<Key, value>
<Key, value>
…
<Key, value>
<Key, value>
<Key, value>
…
<Key, value>
K entries per restart.

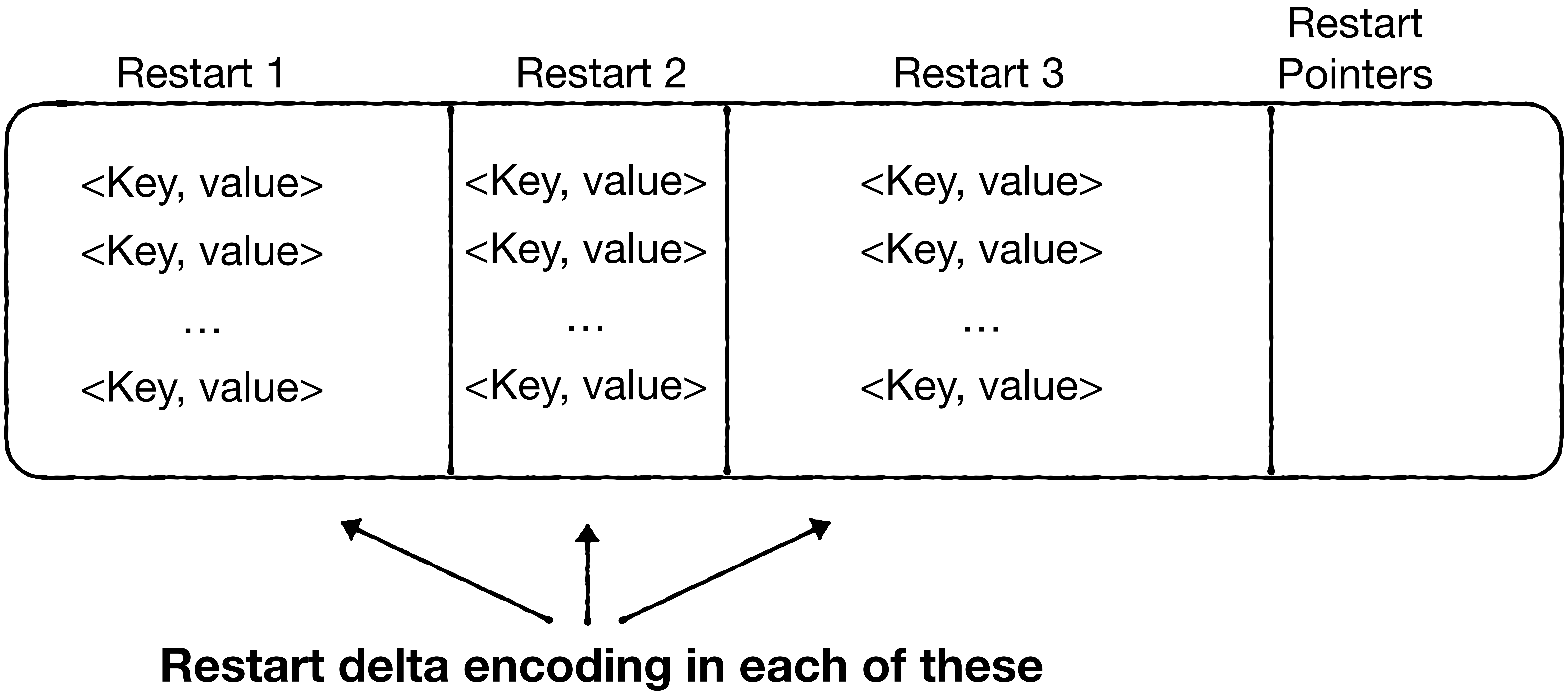
Restarts



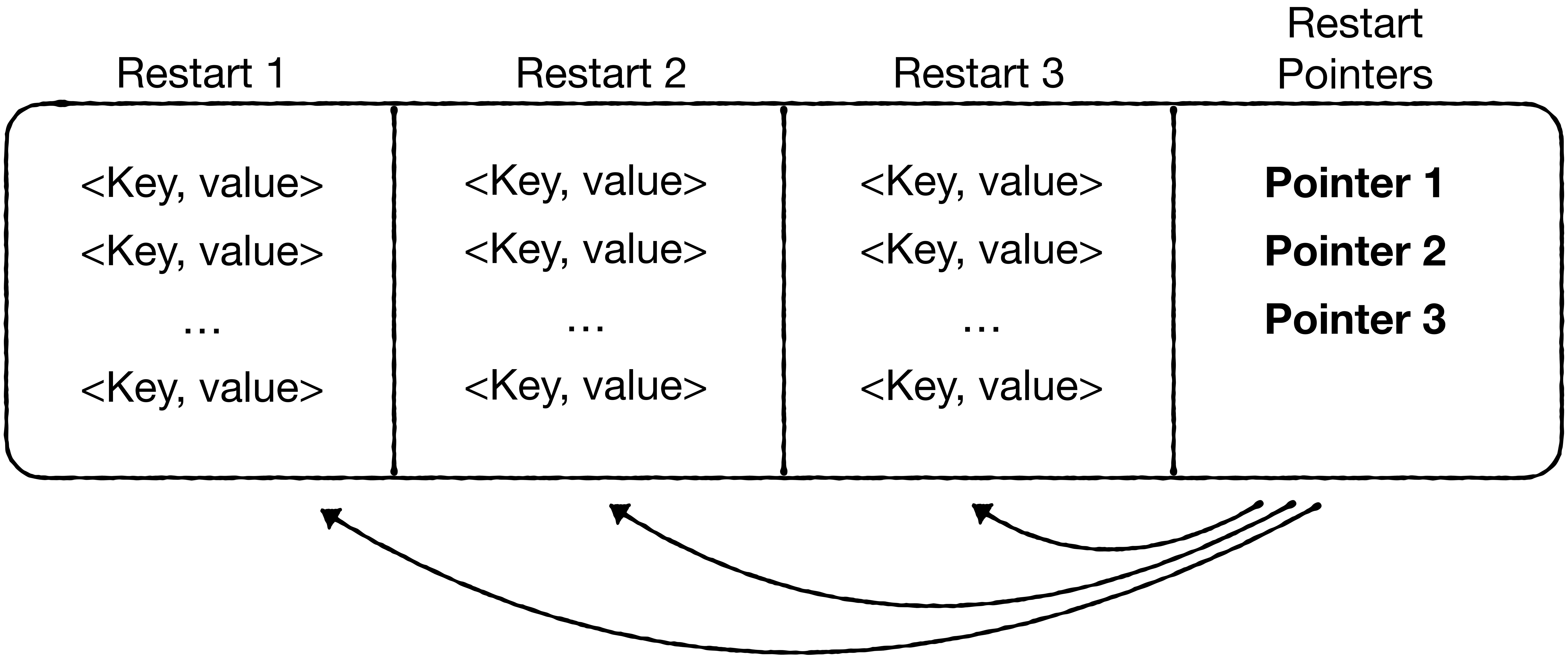
K entries per restart.

Restarts are variable-length (if entries are var-length)

Restarts

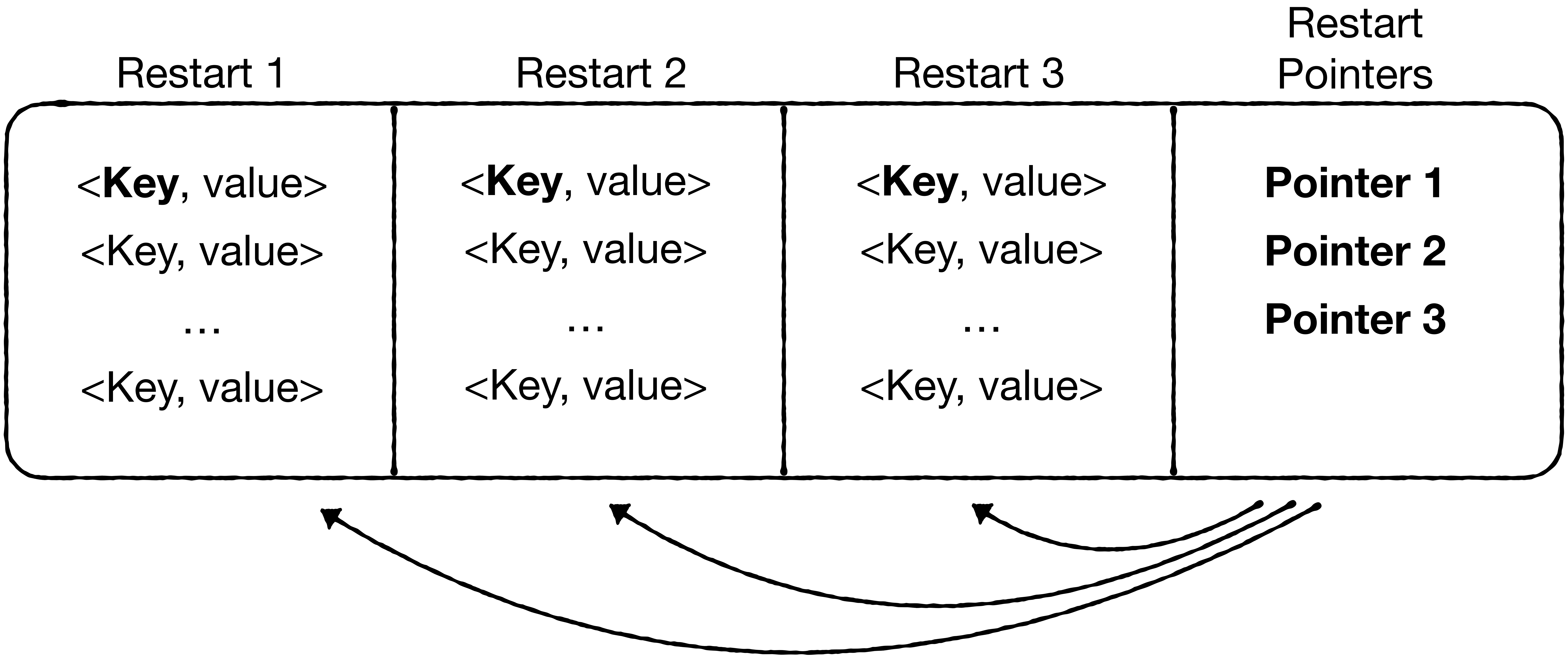

Restart 1
Restart 2
Restart 3
Restart Pointers
<Key, value>
<Key, value>
…
<Key, value>
<Key, value>
<Key, value>
…
<Key, value>
<Key, value>
<Key, value>
…
<Key, value>
Restart delta encoding in each of these

Restarts

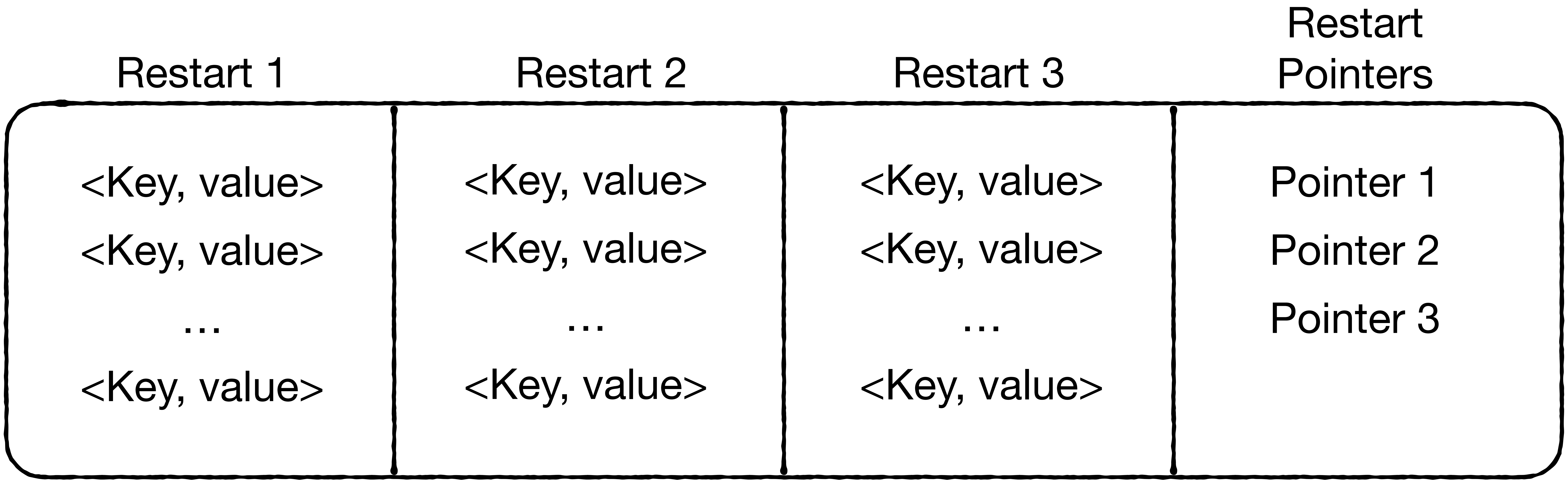


end page contains pointers to start of each restart

Restarts



Can binary search first key in each restart

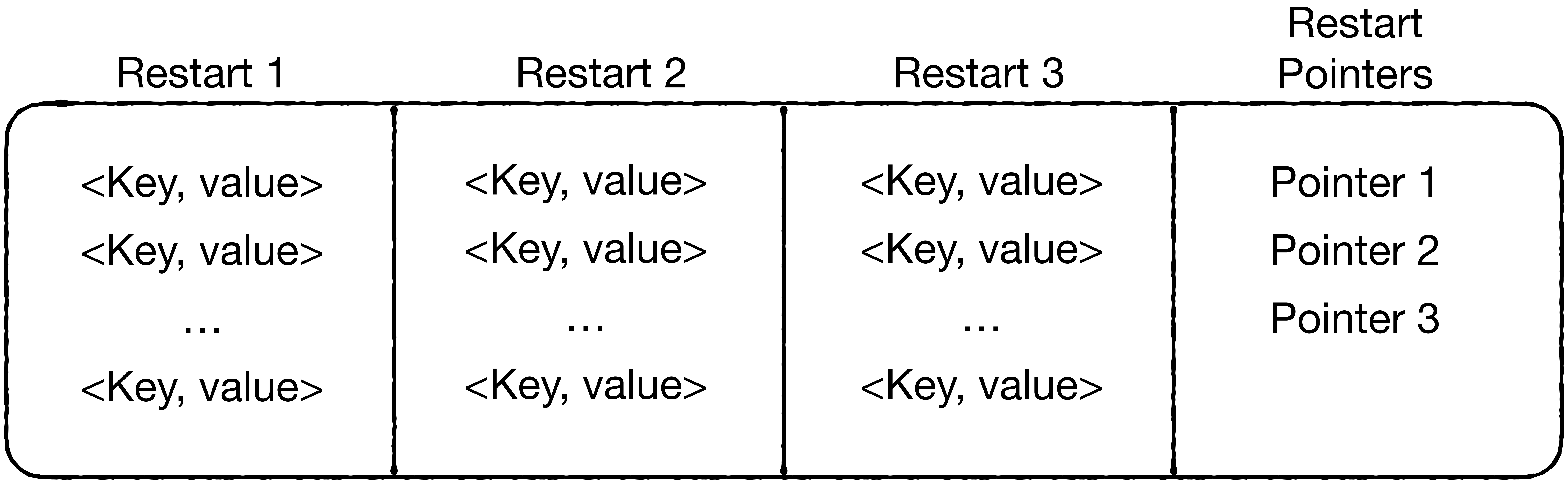


Faster queries (don’t have to traverse/decode the whole page)

Restart 1	Restart 2	Restart 3	Restart Pointers
<Key, value>	<Key, value>	<Key, value>	Pointer 1
<Key, value>	<Key, value>	<Key, value>	Pointer 2
…	…	…	Pointer 3
<Key, value>	<Key, value>	<Key, value>	

Faster queries (don’t have to traverse/decode the whole page)

More space (for restarting delta encodings and storing pointers)

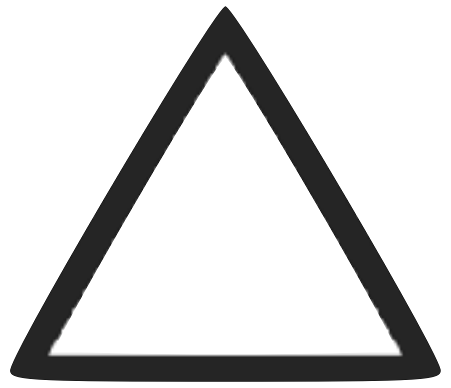


Faster queries (don’t have to traverse/decode the whole page)

Can we be even faster?

Page Sizing

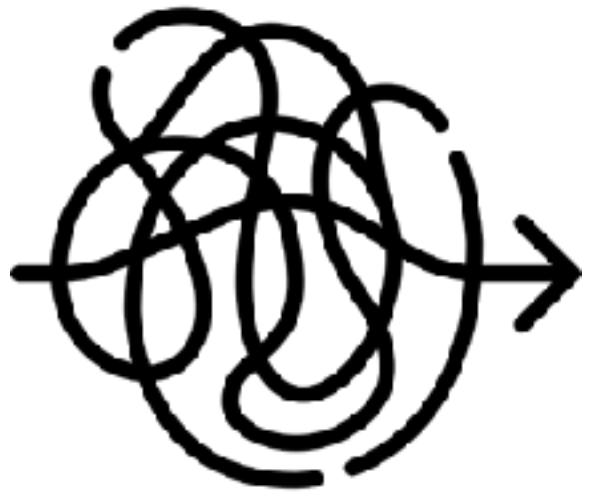
Delta Encoding



Restarts

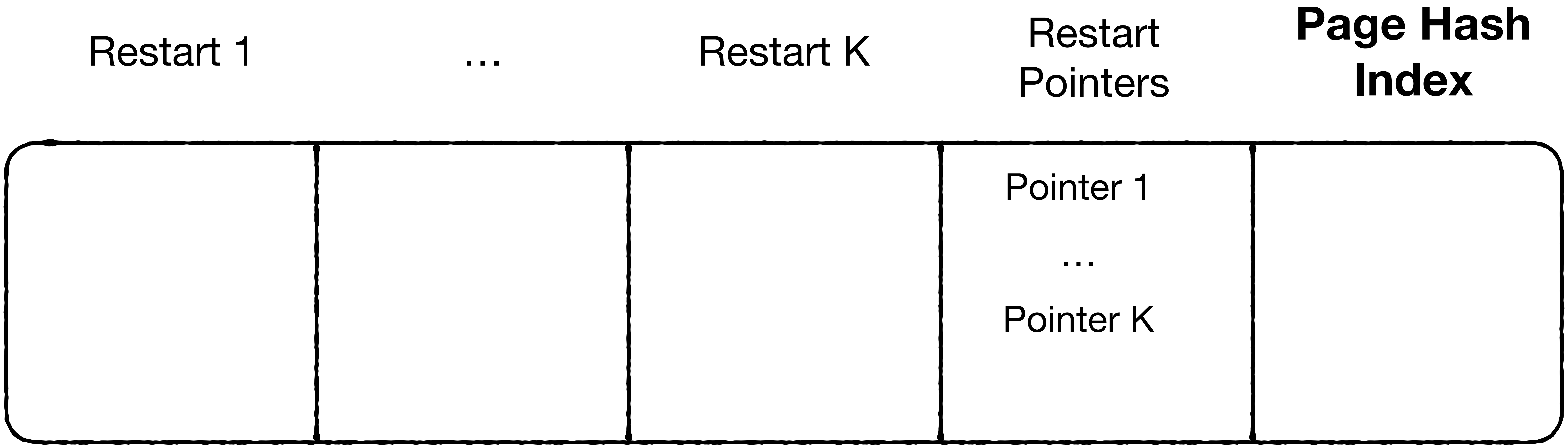
Page Hash Index

LZ77

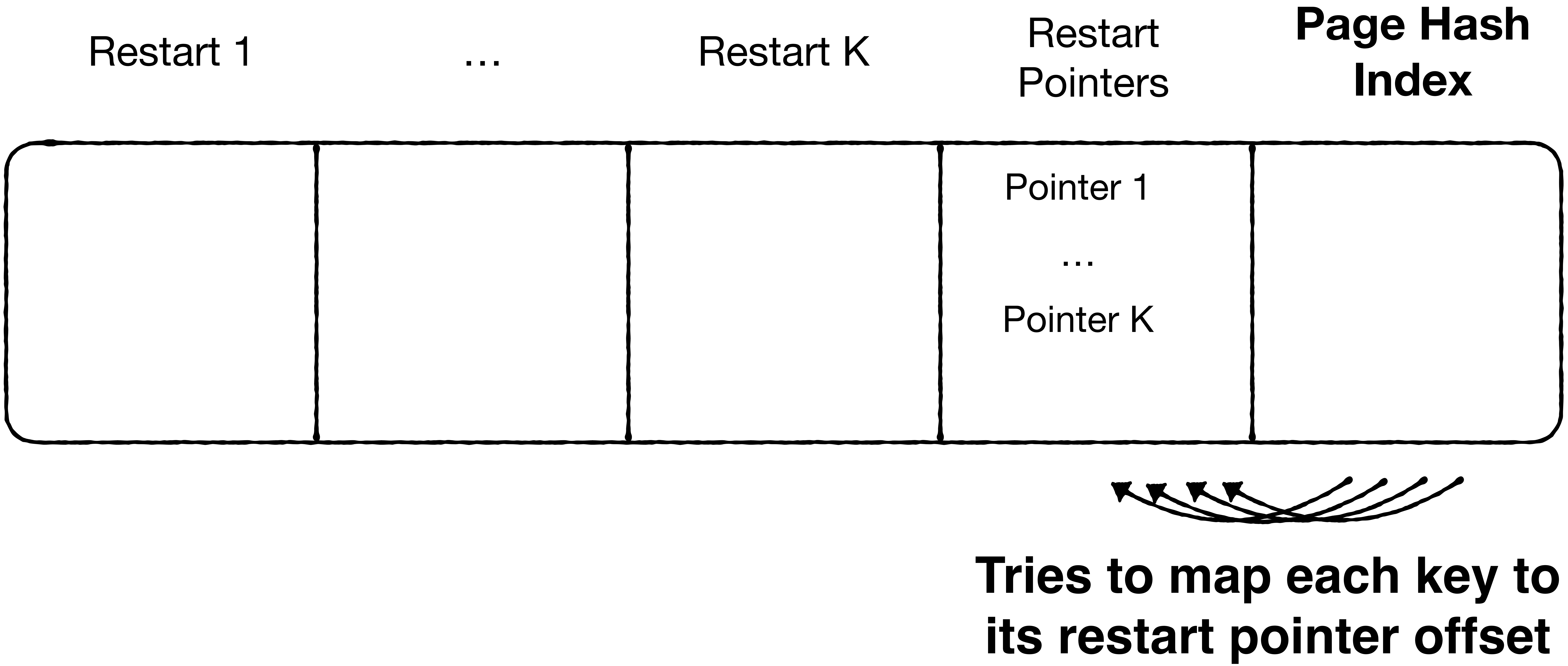


Huffman coding

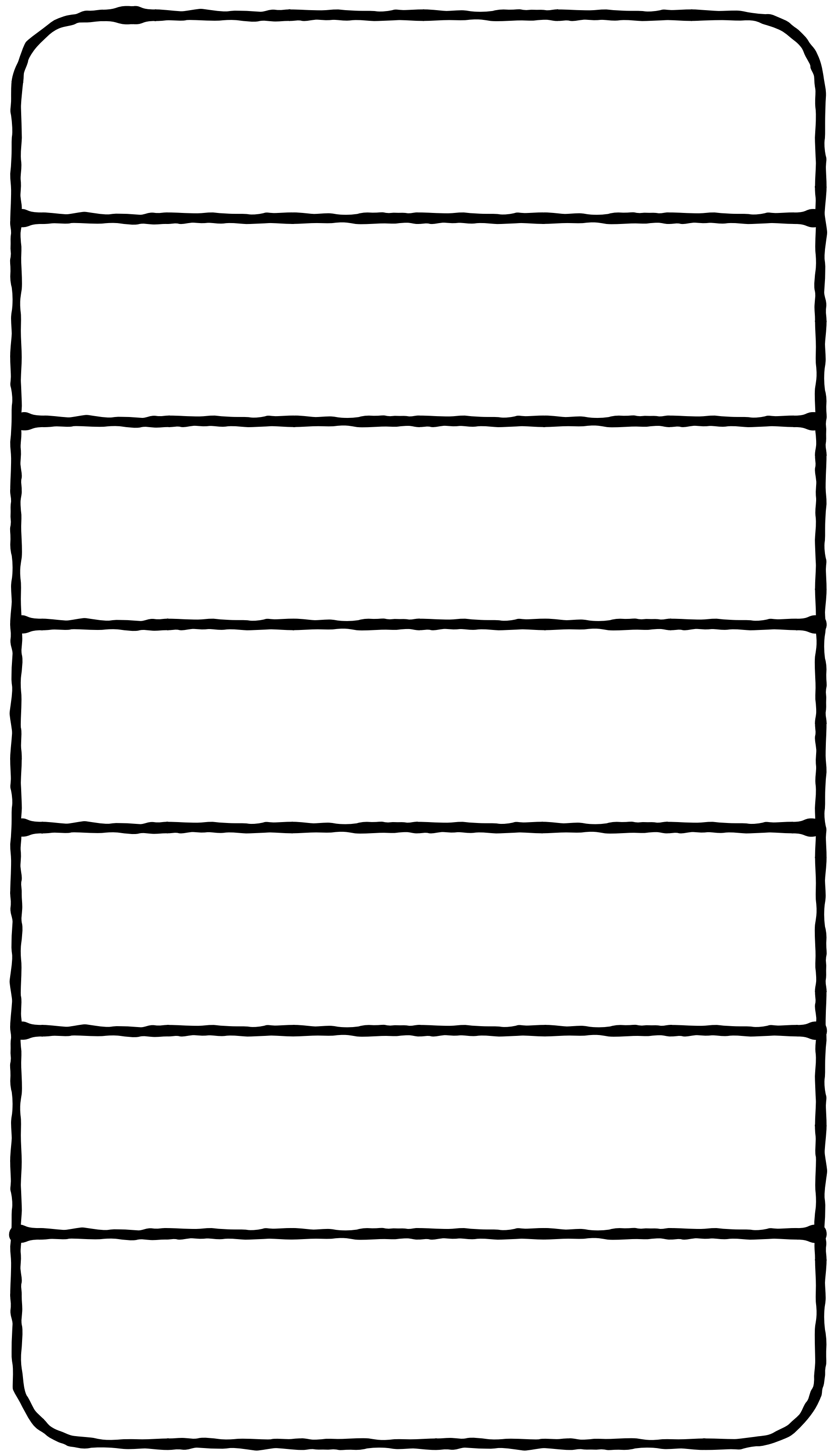
Dictionary training



Restart 1
…
Restart K
Restart Pointers
Page Hash Index
Pointer 1
…
Pointer K

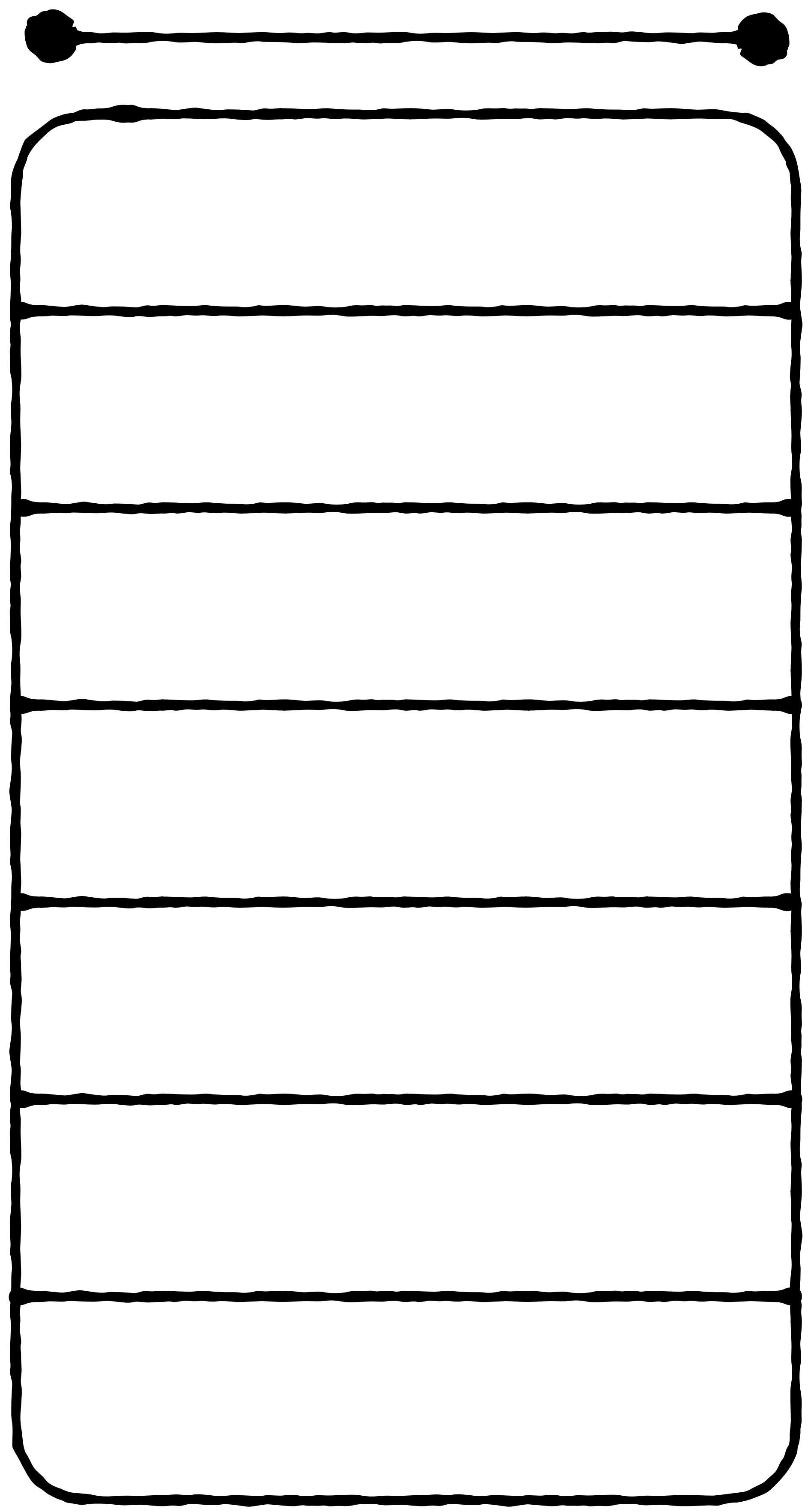

Restart 1
…
Restart K
Restart Pointers
Page Hash Index
Pointer 1
…
Pointer K
Tries to map each key to its restart pointer offset

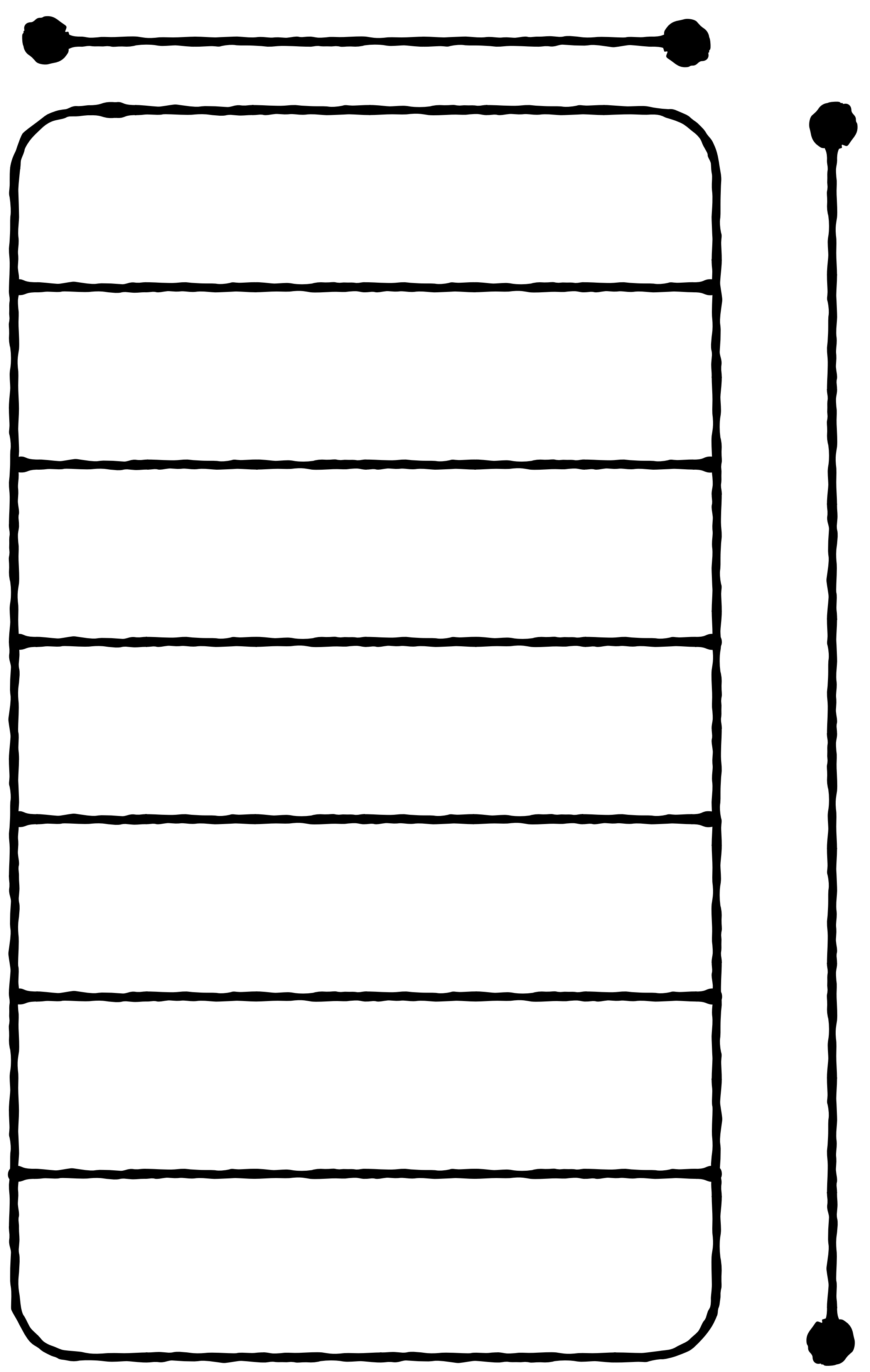
Page Hash Index

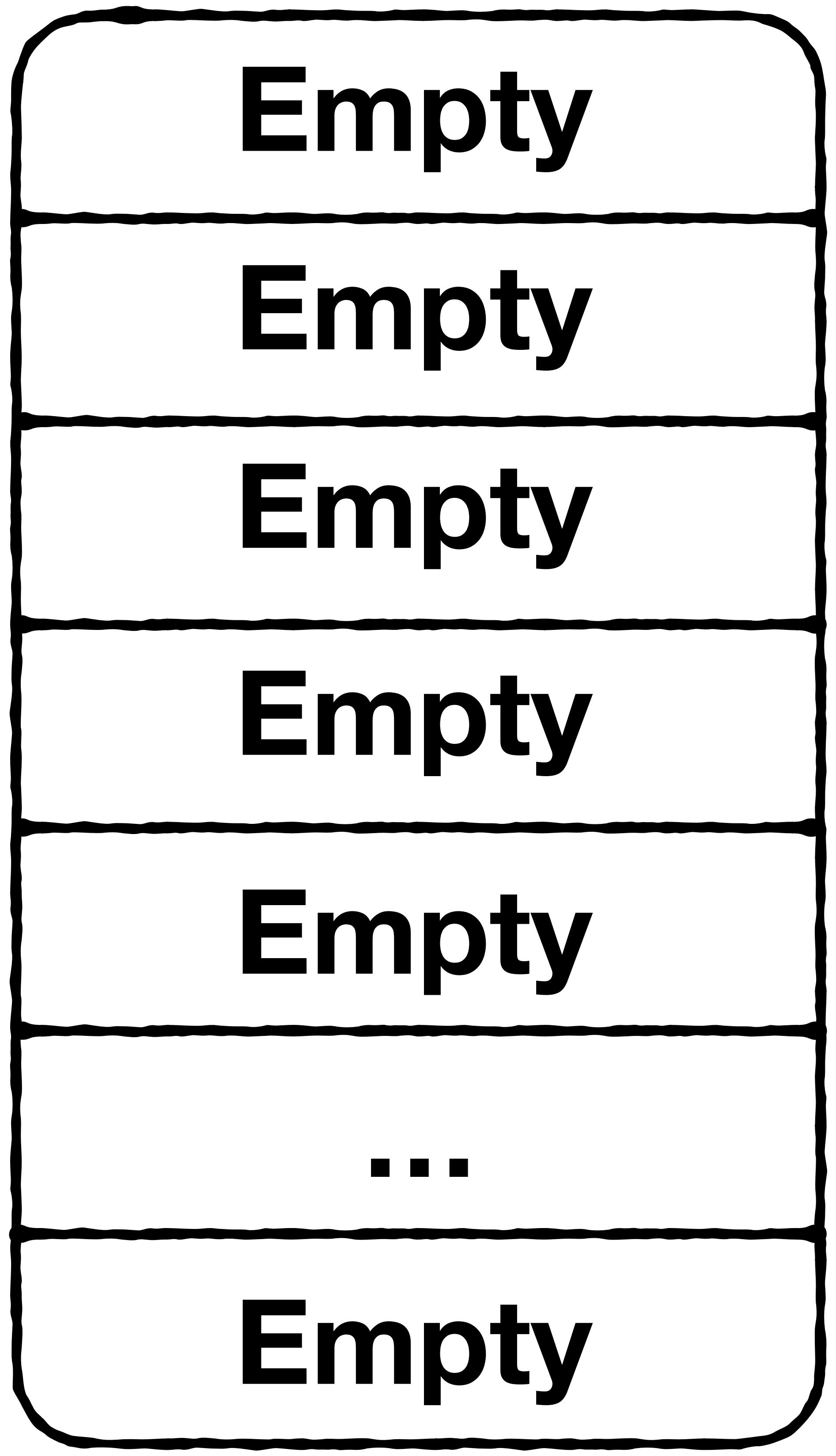


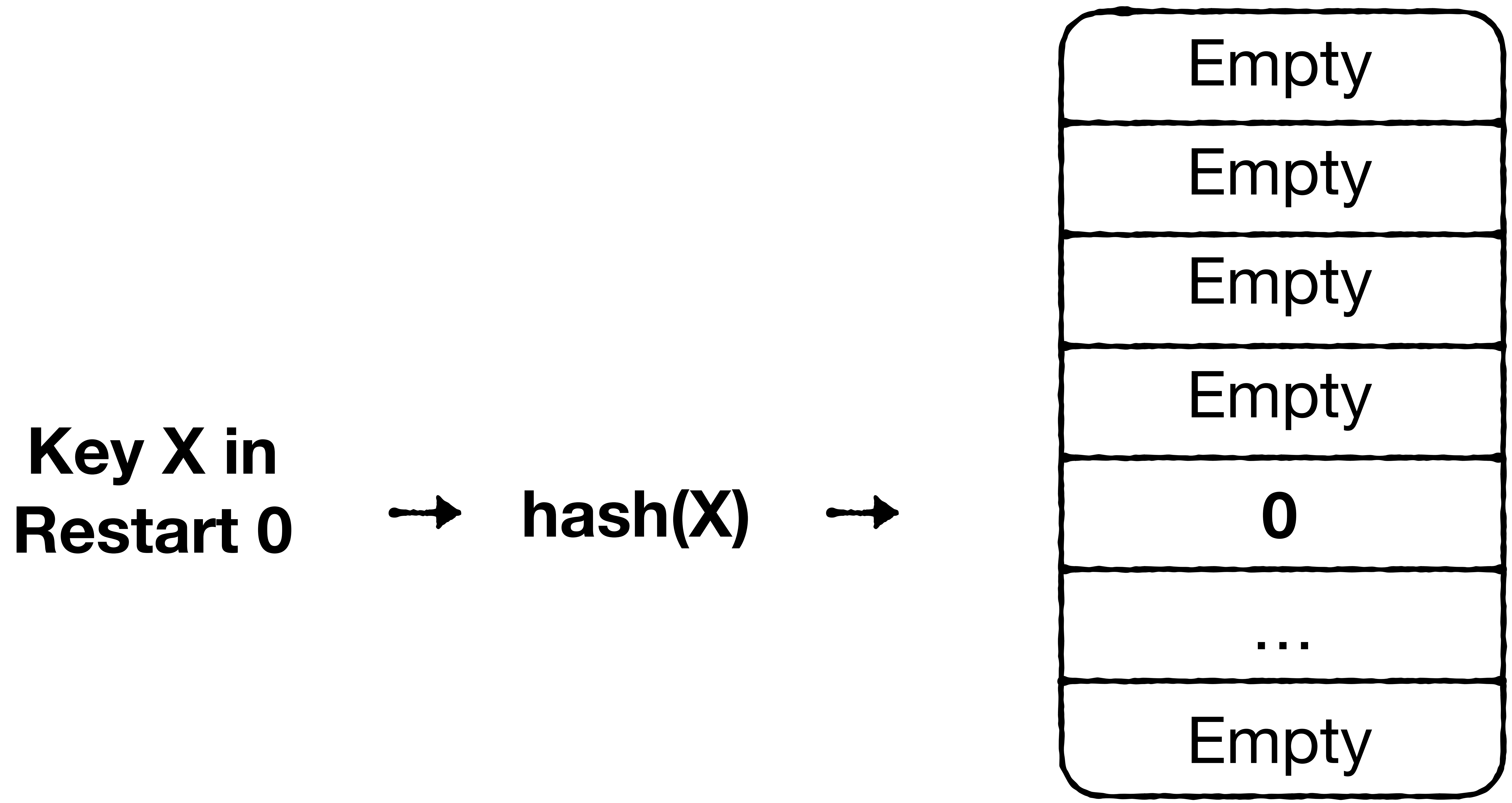
Page Hash Index

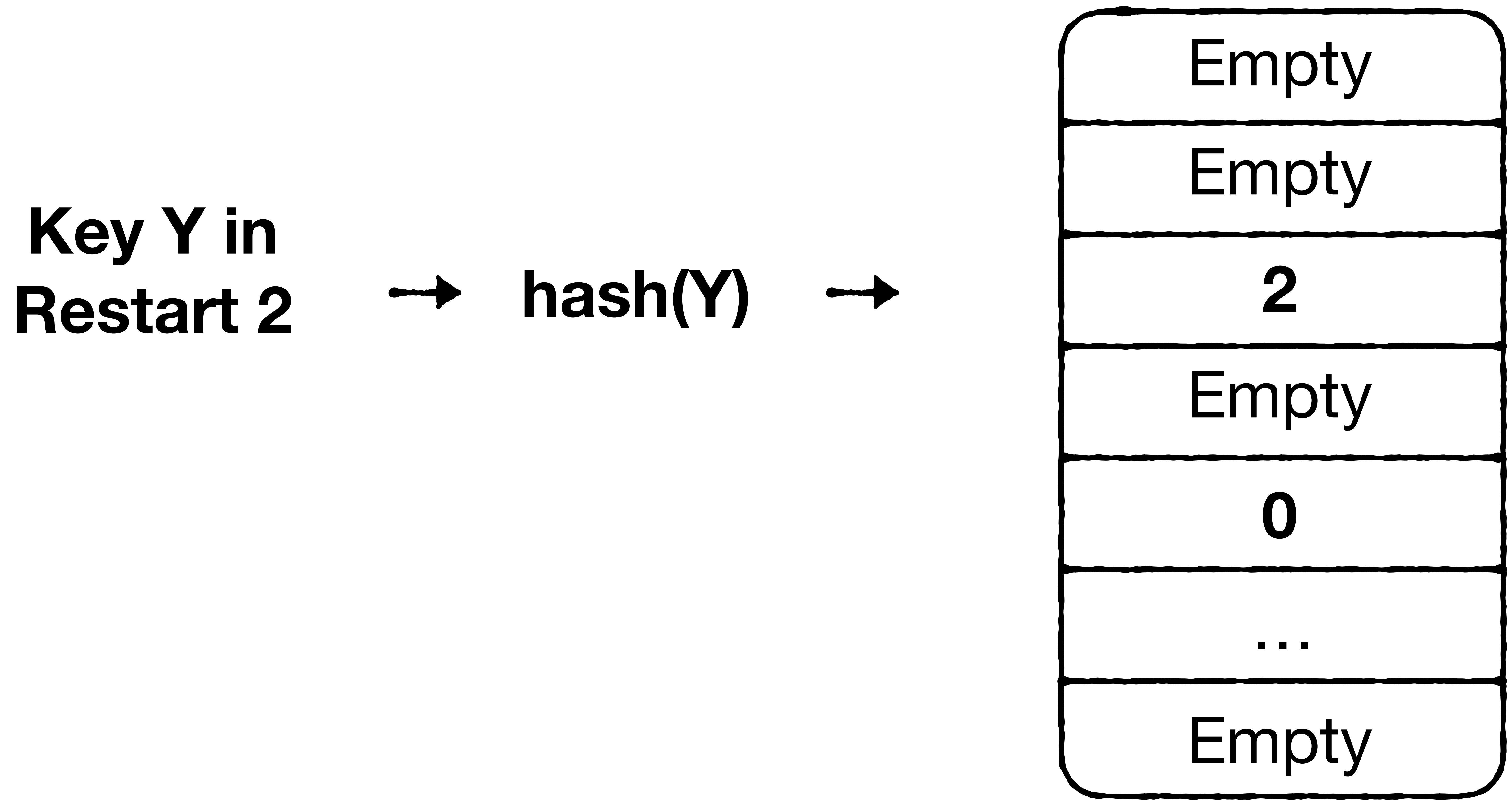
8 bit width (identifies at most 256 restarts)

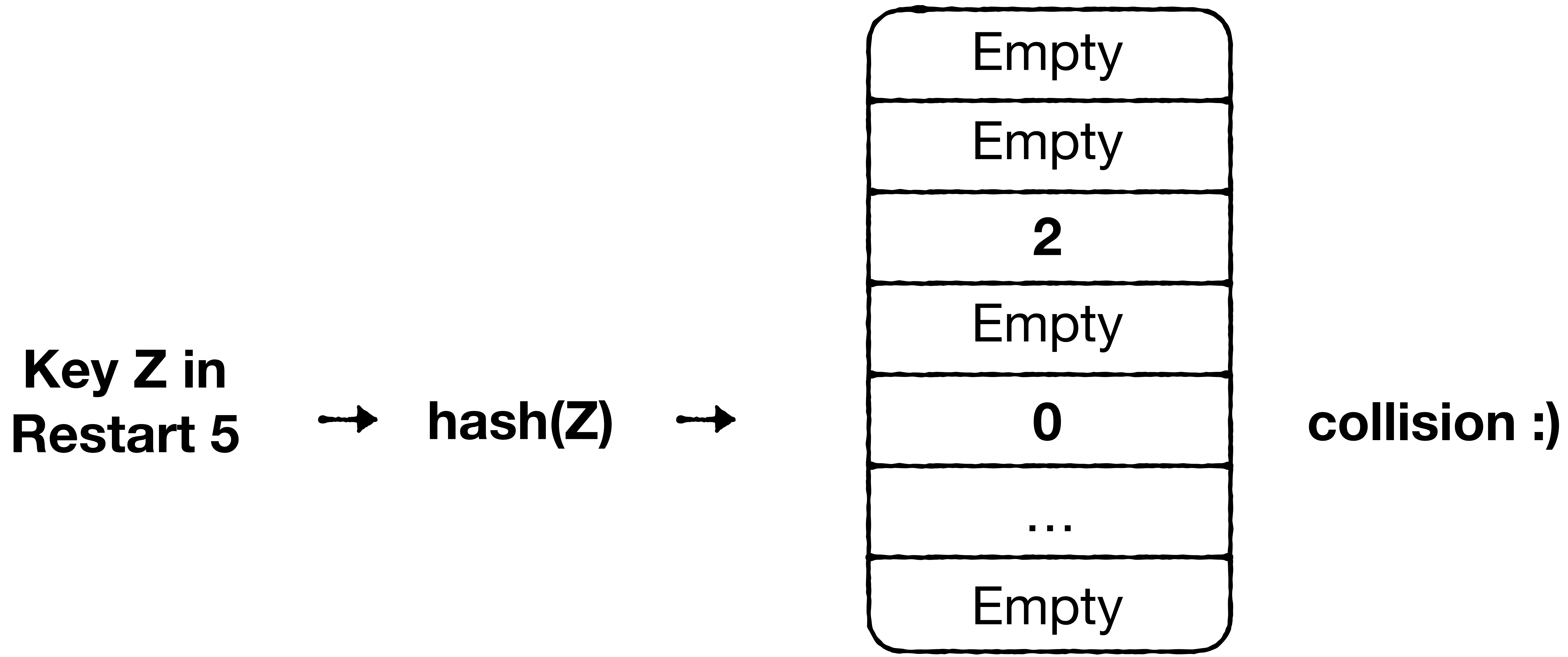



8 bit width (identifies at most 256 restarts)
As many slots as entries in page


Empty
Empty
Empty
Empty
Empty
...
Empty

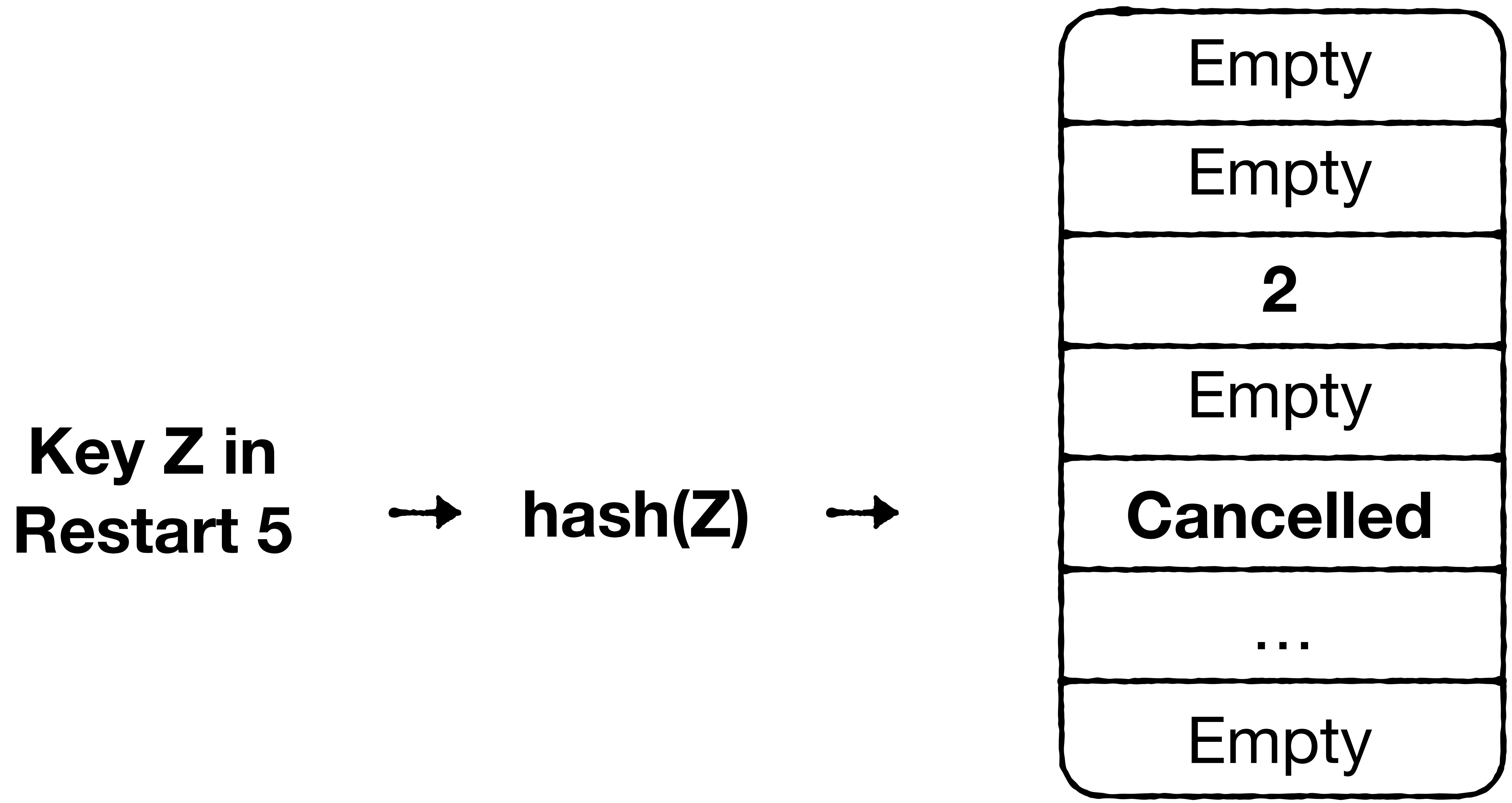

Key X in
Restart 0
hash(X)
Empty
Empty
Empty
Empty
0
...
Empty


Key Y in
Restart 2
hash(Y)
Empty
Empty
2
Empty
0
...
Empty

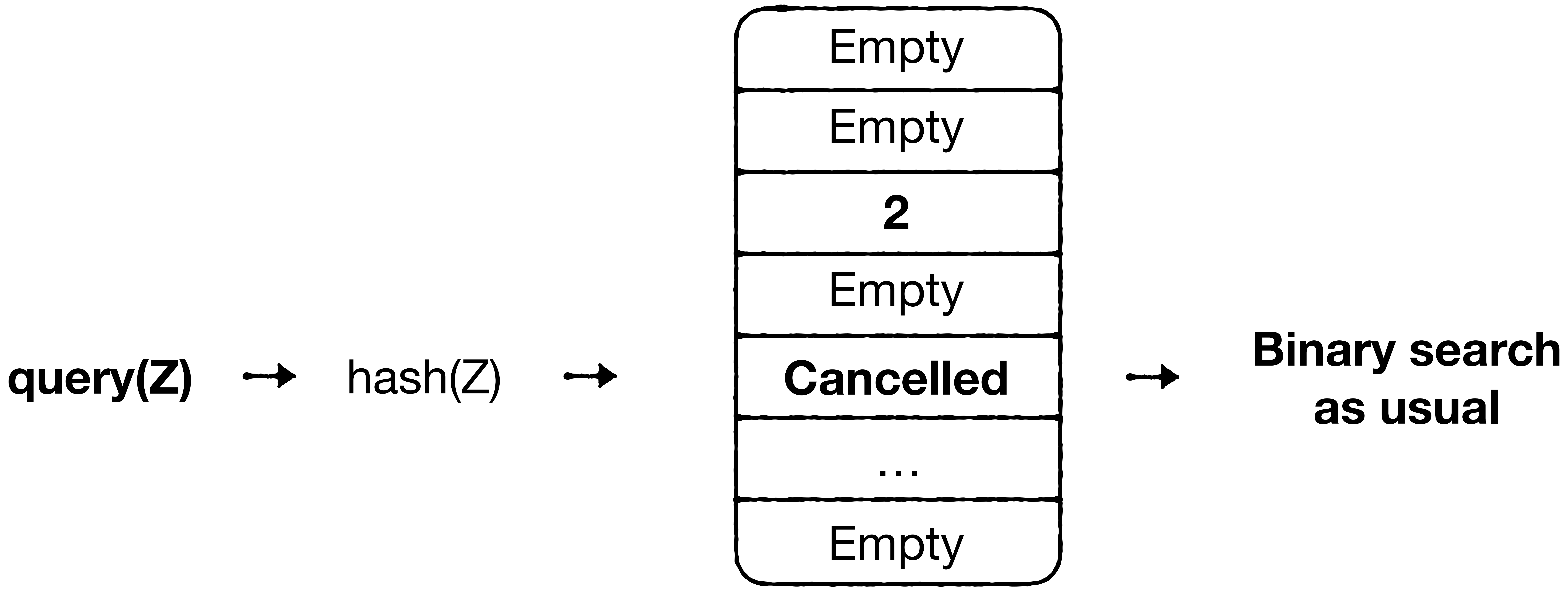

Key Z in
Restart 5
hash(Z)
Empty
Empty
2
Empty
0
...
Empty
Empty
collision :)

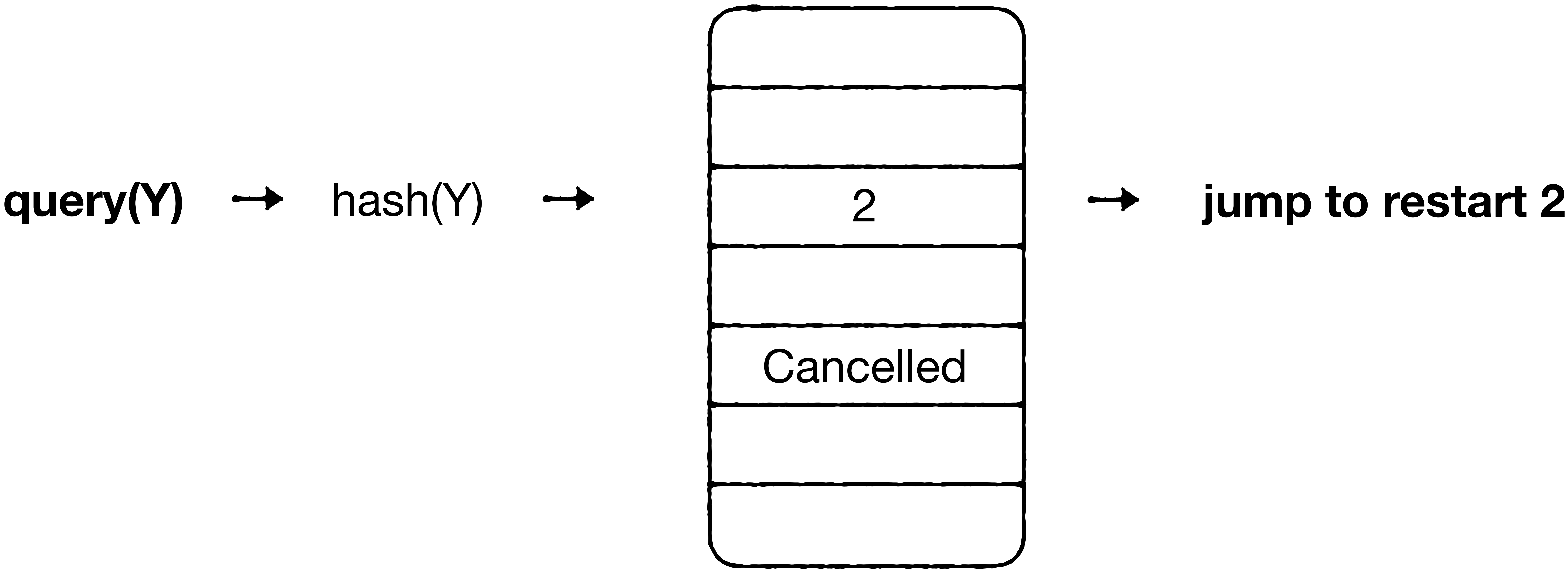
collision cancels a slot

(using special symbol, e.g., 11111111)

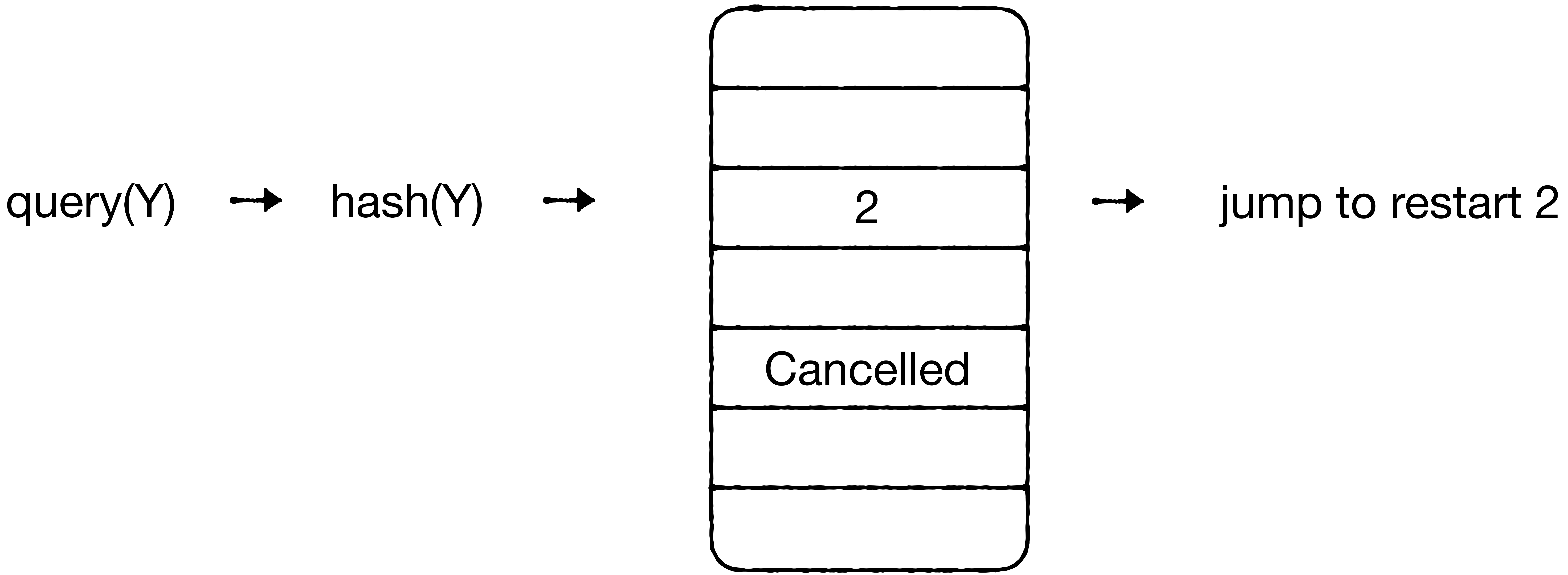


collision cancels a slot
(using special symbol, e.g., 11111111)

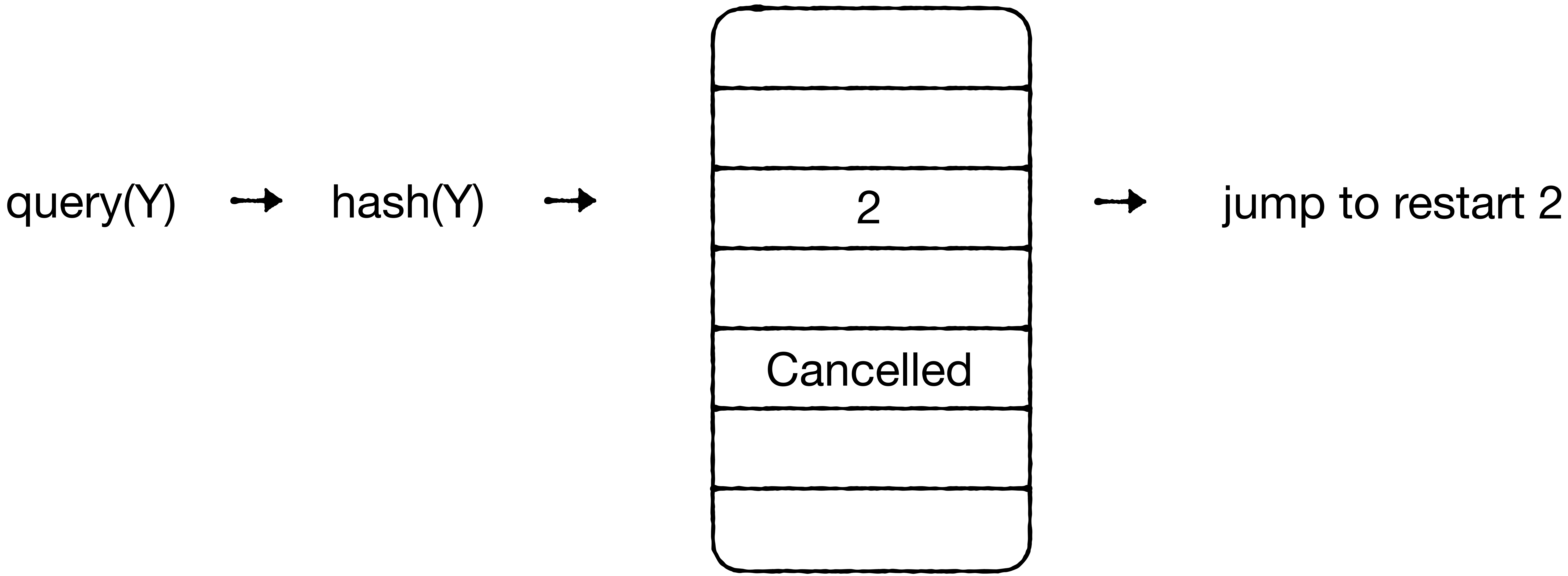



query(Y)
hash(Y)
2
Cancelled
jump to restart 2

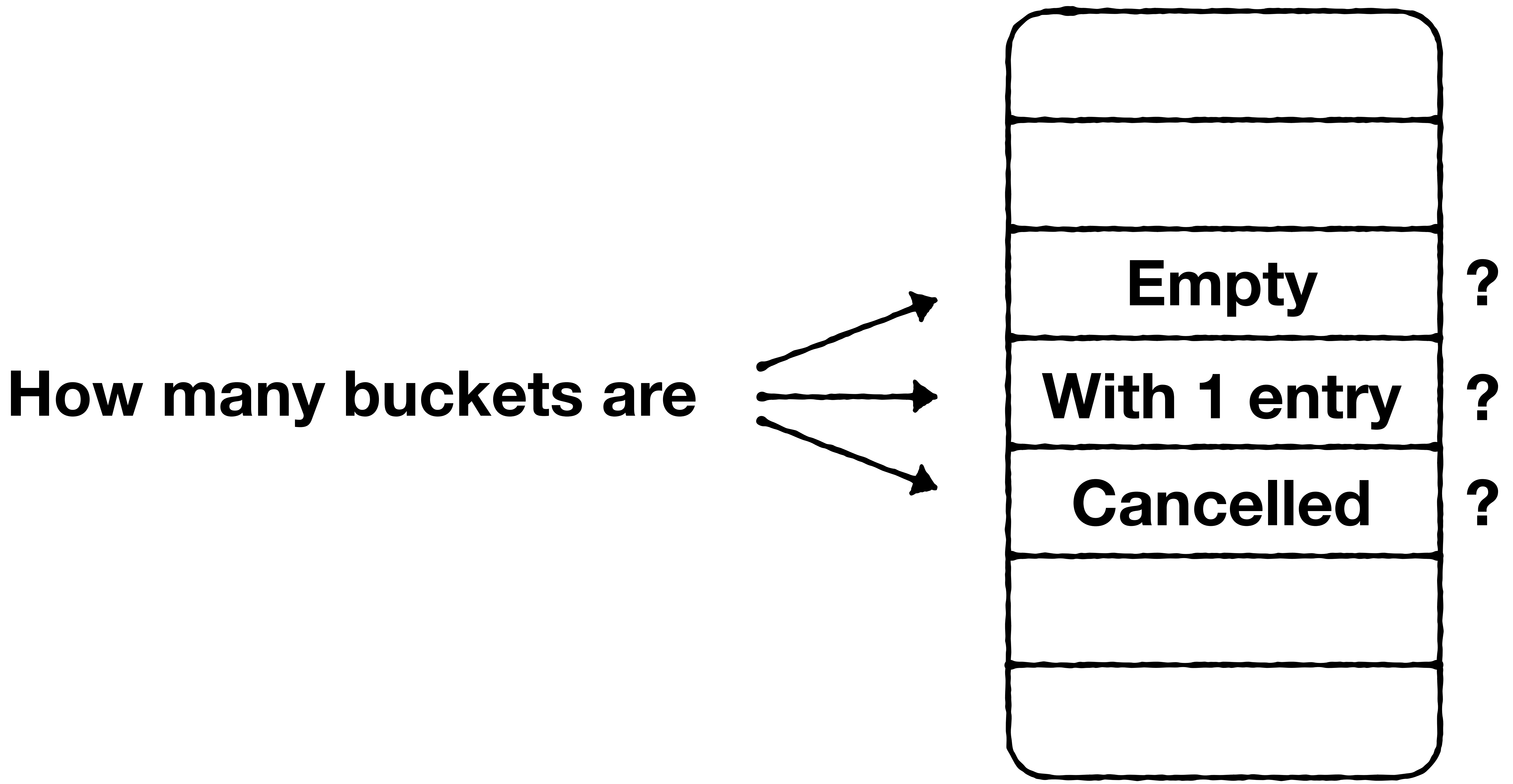
Non-cancelled slots guarantee we jump to the correct location



What’s the likelihood to hit a non-cancelled slot?



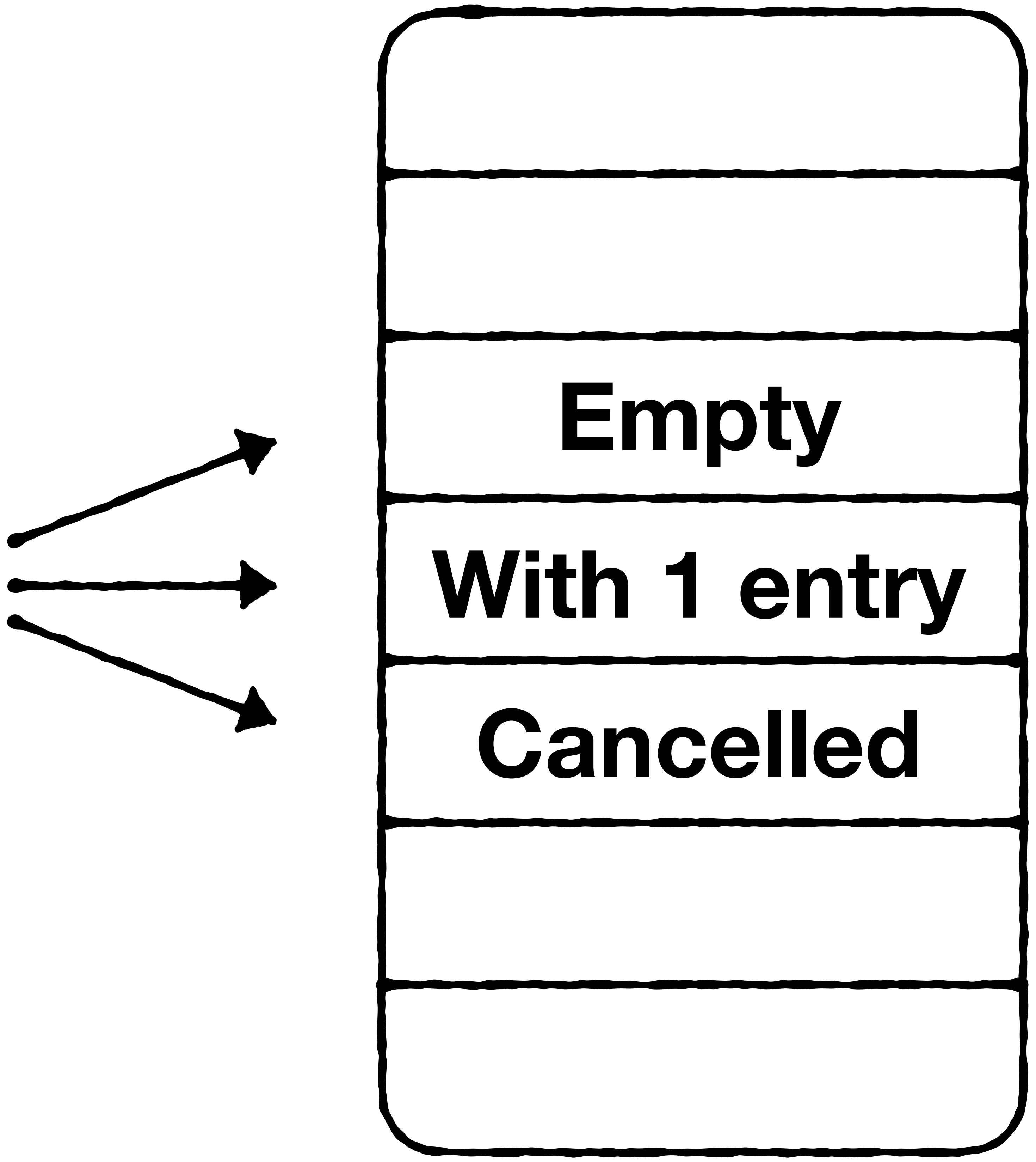
What’s the likelihood to hit a non-cancelled slot?



Poisson distribution

λ=1

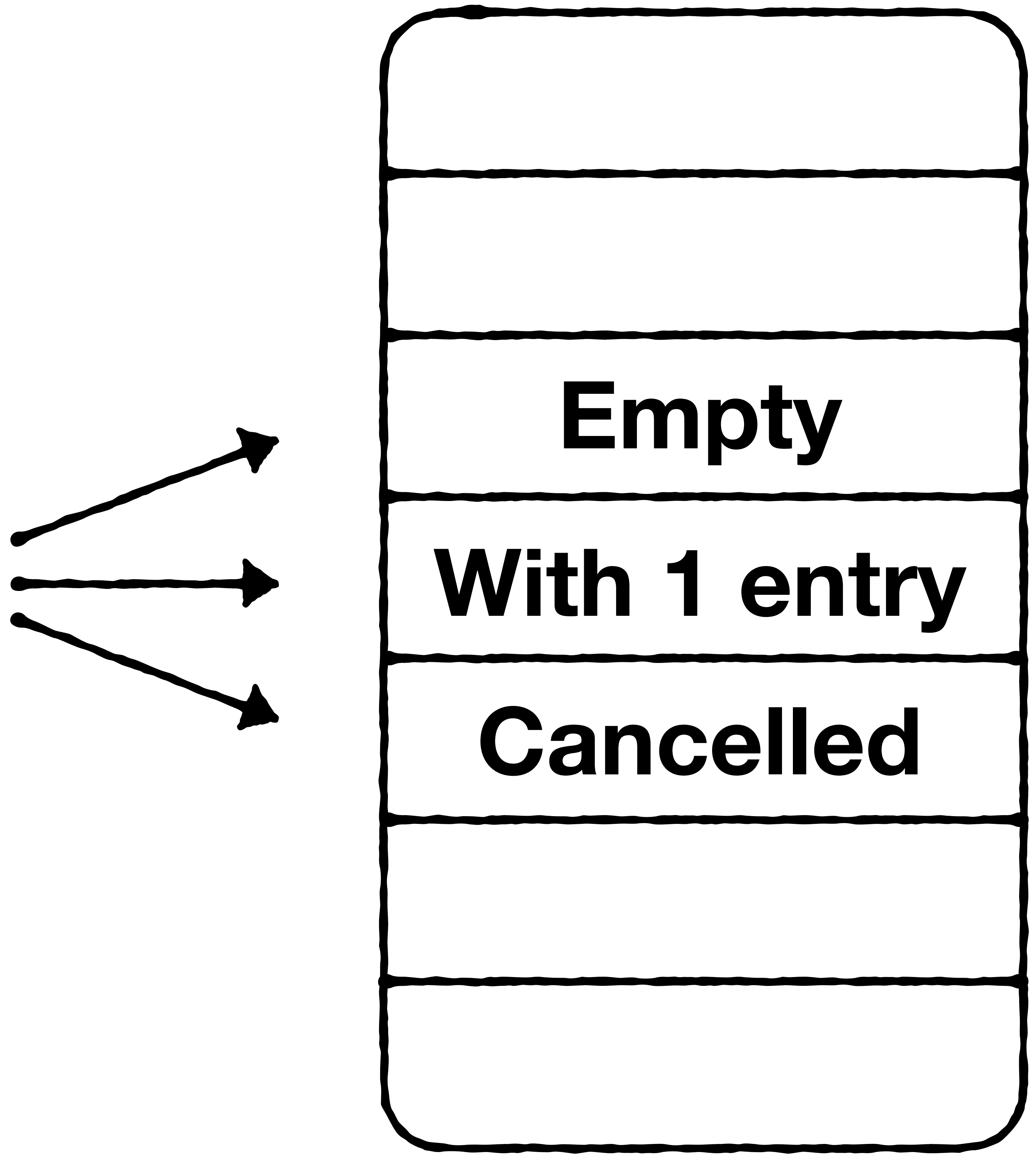
How many buckets are



Poisson distribution

λ=1

How many buckets are



$P(\lambda, 0)$

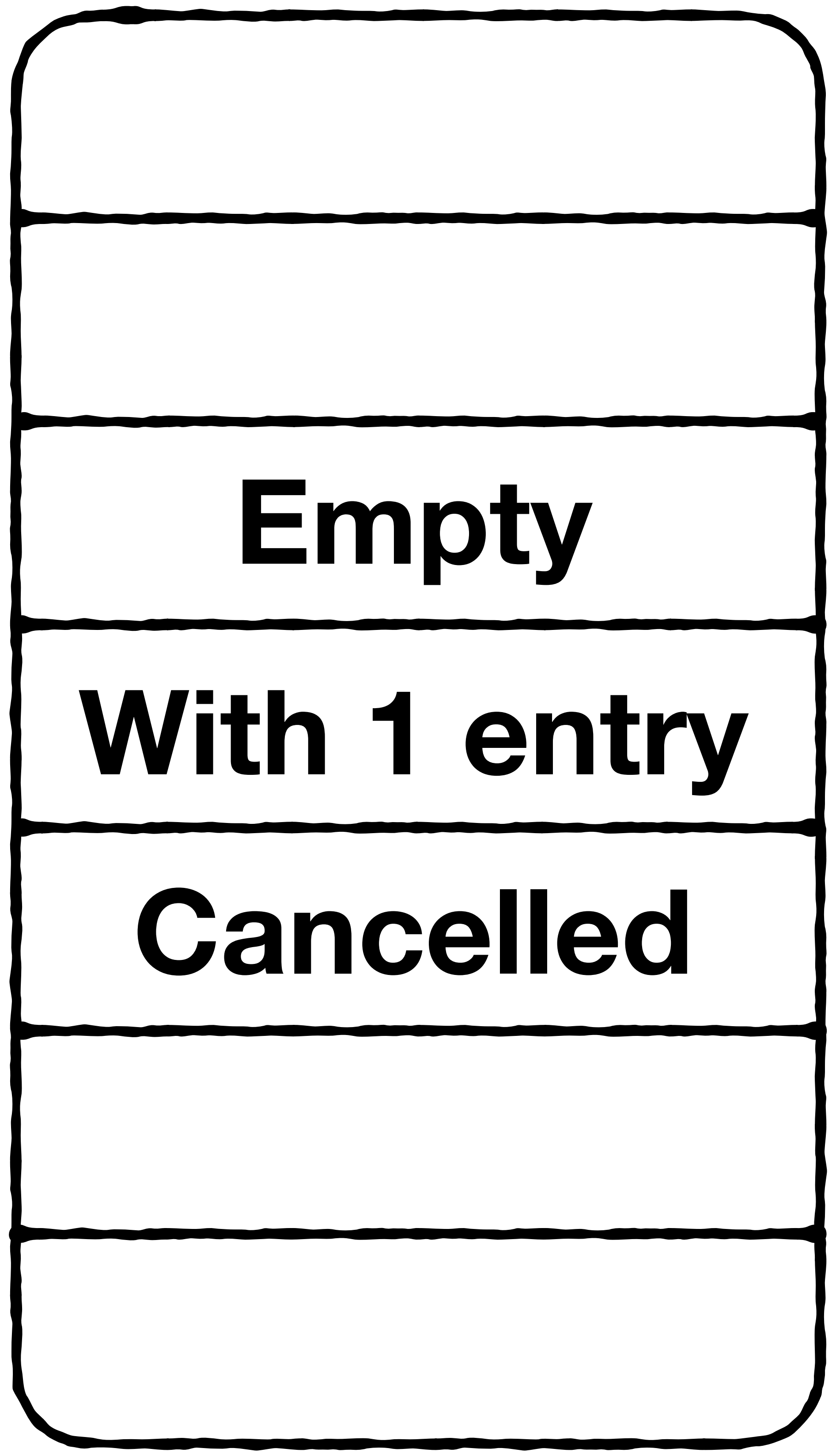
$P(\lambda, 1)$

$1 - P(\lambda, 0) - P(\lambda, 1)$

Poisson distribution

$\lambda=1$

How many buckets are:



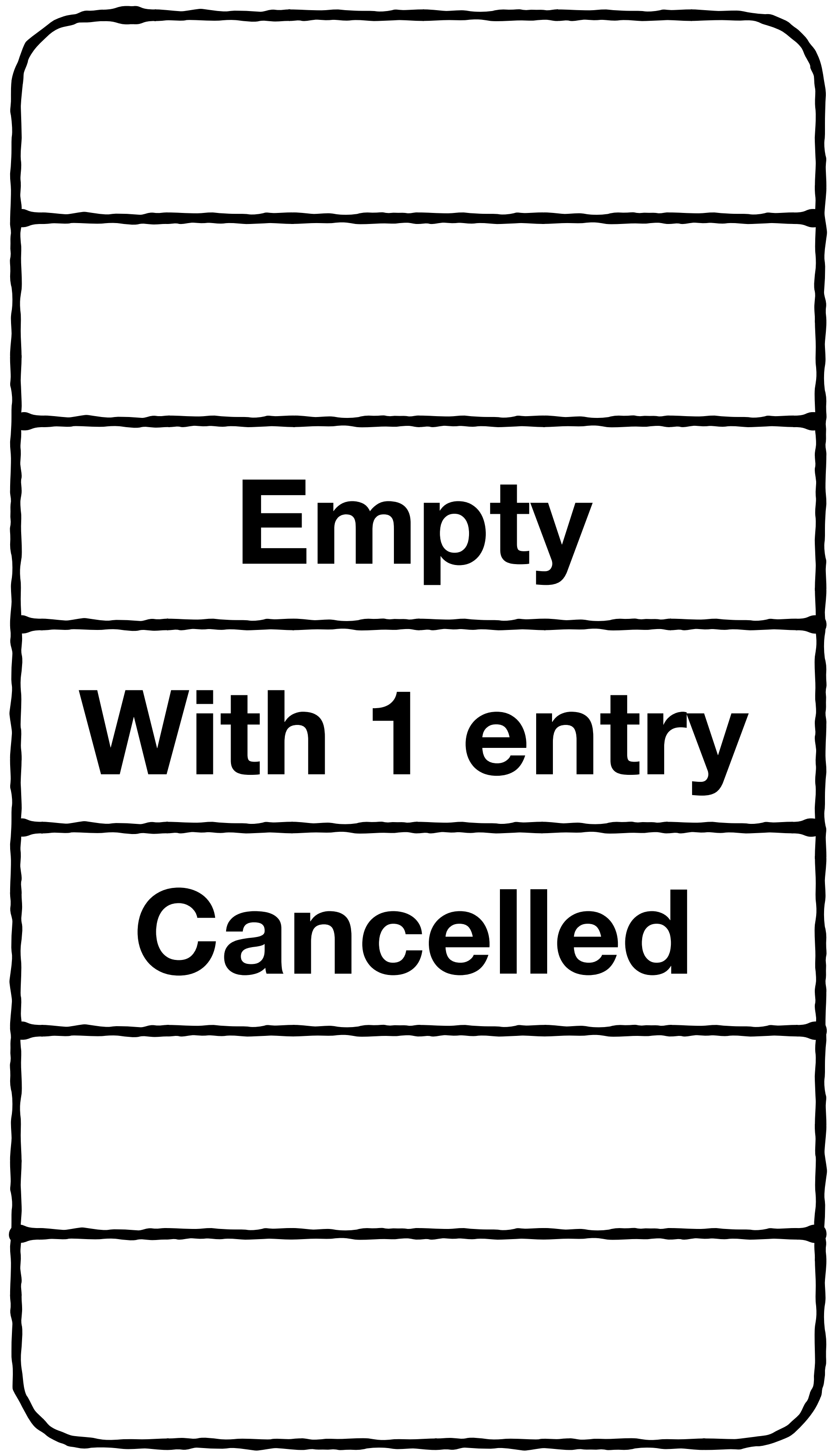
$e^{-1} = 0.37$

$e^{-1} = 0.37$

$1 - 2 \cdot e^{-1} = 0.26$

What's the likelihood to hit a non-cancelled slot?

How many buckets are:



$e^{-1} = 0.37$

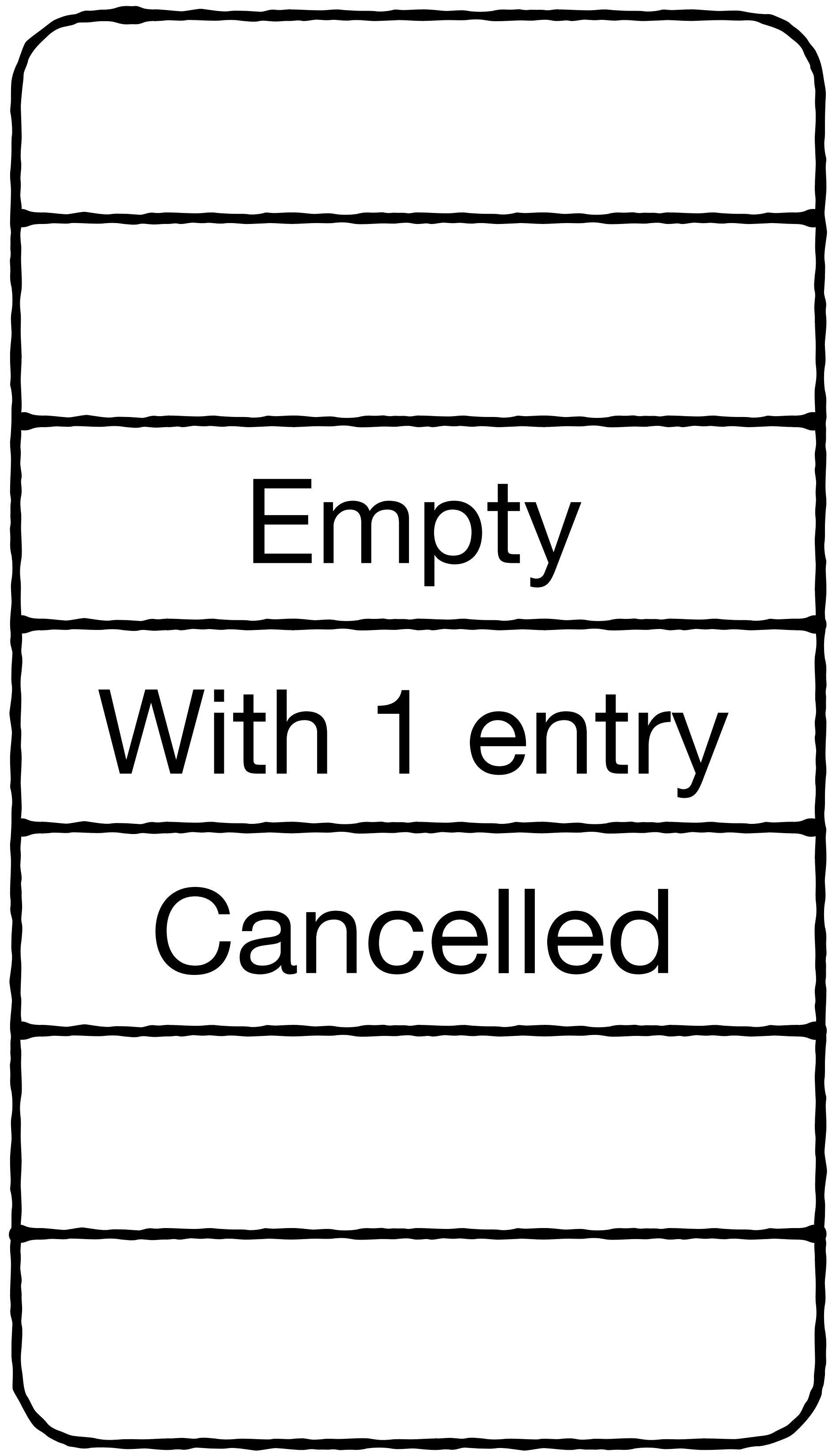
$e^{-1} = 0.37$

$1 - 2 \cdot e^{-1} = 0.26$

What’s the likelihood to hit a non-cancelled slot?

$$\frac{\textbf{P(With 1 entry)}}{\textbf{P(With 1 entry) + P(cancelled)}}$$

How many buckets are:



$e^{-1} = 0.37$

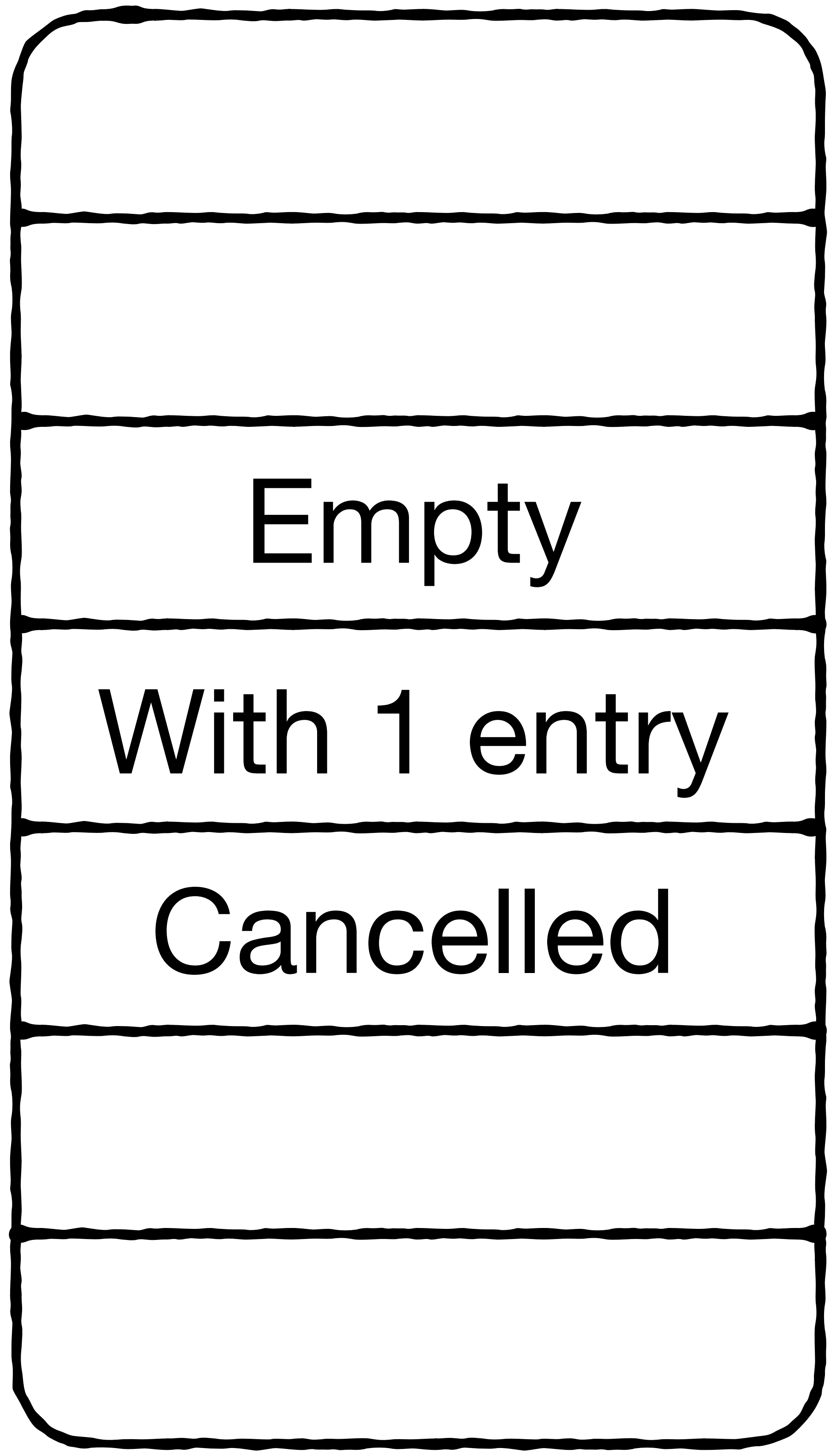
$e^{-1} = 0.37$

$1 - 2 \cdot e^{-1} = 0.26$

What's the likelihood to hit a non-cancelled slot?

$$\frac{\textbf{P(With 1 entry)}}{\textbf{P(With 1 entry) + P(cancelled)}} = \frac{\textbf{0.37}}{\textbf{0.37 + 0.26}} = \textbf{0.59}$$

How many buckets are:



$e^{-1} = 0.37$

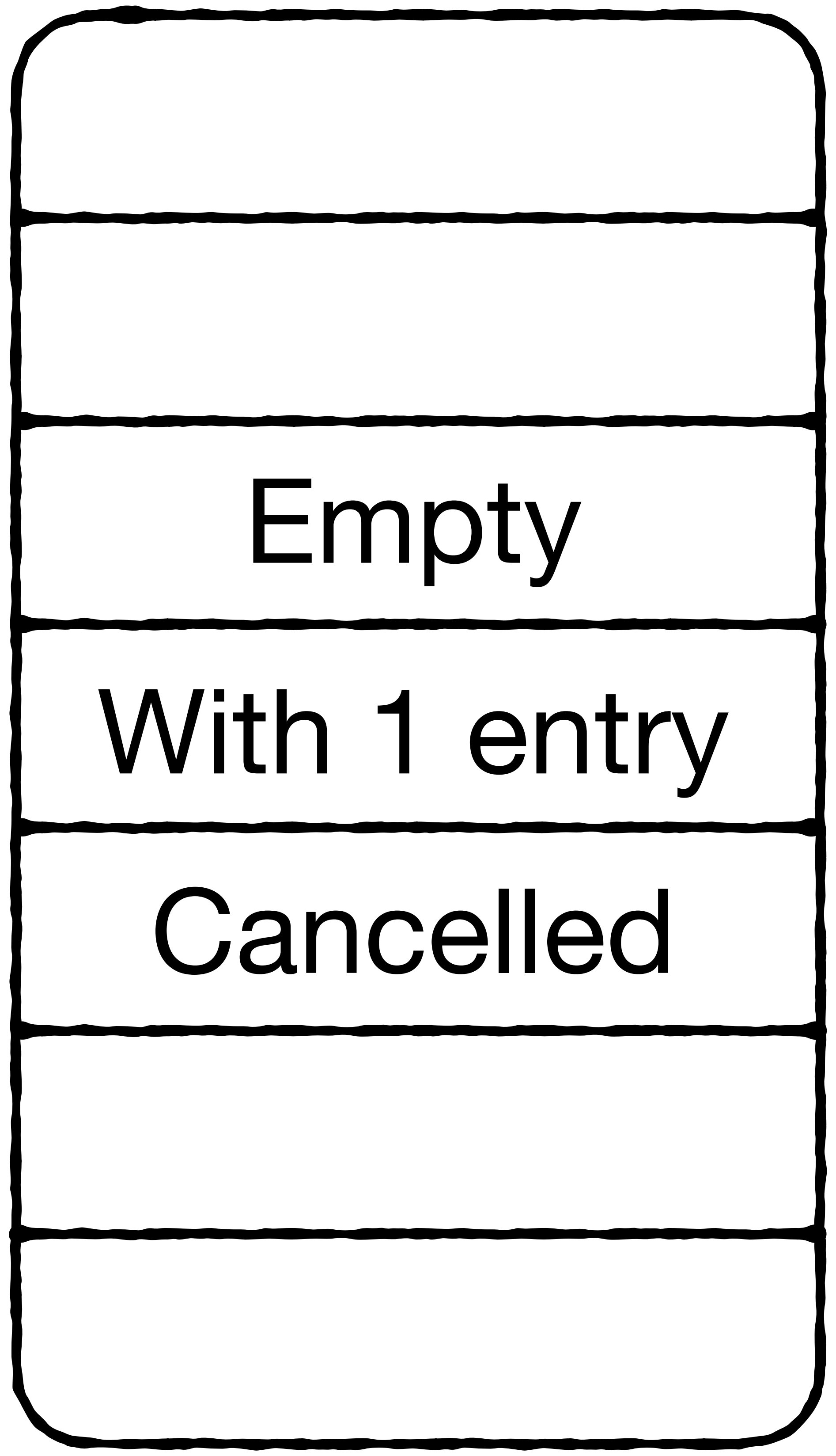
$e^{-1} = 0.37$

$1 - 2 \cdot e^{-1} = 0.26$

Challenge: how to improve this?

$$\frac{P(\text{With 1 entry})}{P(\text{With 1 entry}) + P(\text{cancelled})} = \frac{0.37}{0.37 + 0.26} = \mathbf{0.59}$$

How many buckets are:



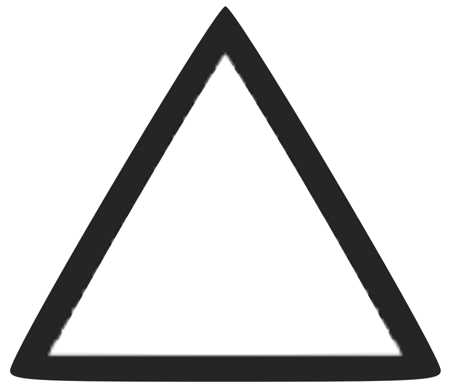
$e^{-1} = 0.37$

$e^{-1} = 0.37$

$1 - 2 \cdot e^{-1} = 0.26$

Page Sizing

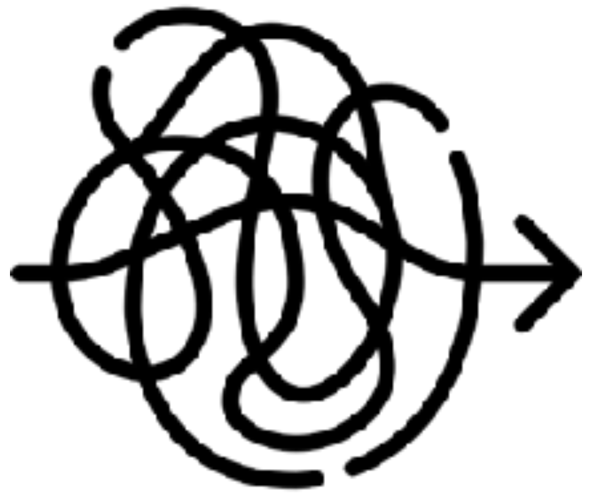
Delta Encoding



Restarts

Page Hash Index

LZ77



Huffman coding

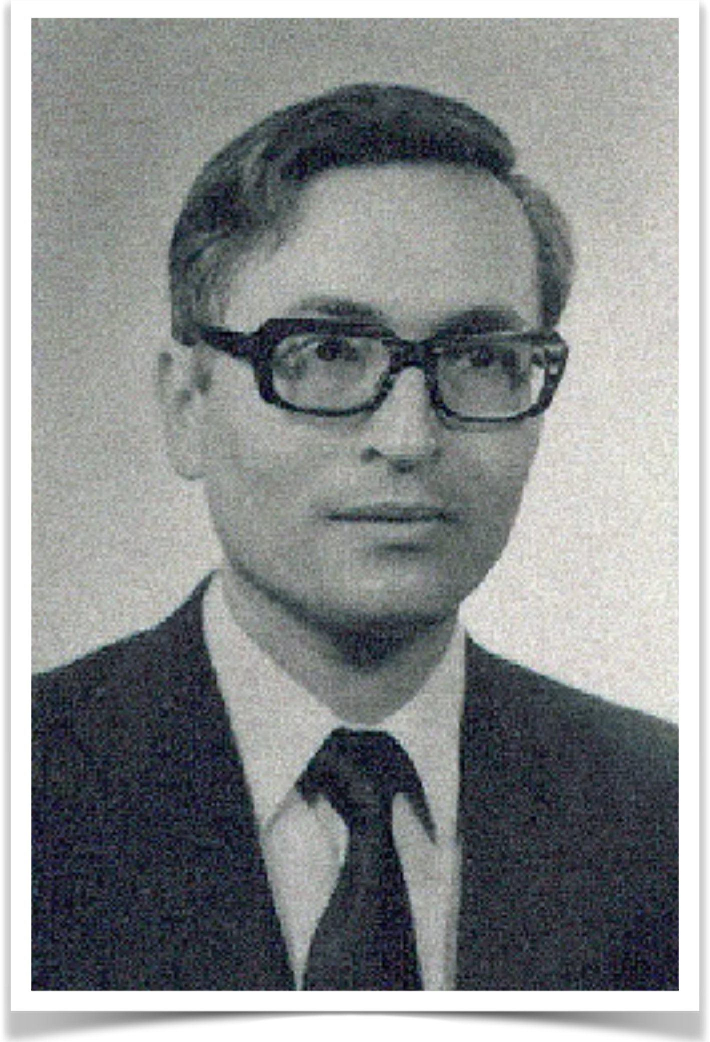
Dictionary training

LZ77 - Lempel-Ziv 1977

Abraham Lempel

1936 - 2023

Technion



Jacob Ziv

1931 - 2023

Technion

LZ77 - Lempel-Ziv 1977

delta encoding is only for sorted keys.

data values aren't sorted but often contain redundancy

<K1, **Hello**> <K2, **Hell**> <K3, **Jello**>

K1 **Hello** K2 **Hell** K3 **Jello**

View whole page as string of characters

Slide a window over input string and eliminates repeating substrings

K1 **Hello** K2 **Hell** K3 **Jello**

Principles

H e l l o H e l l J e l l o

Principles

Consider a substring

H e l l o H e l l J **e l l o**

Any substring can be viewed as:

[substring we have seen] … [new character]

H e l l o H e l l J **e l l o**

Any substring can be viewed as:

[**substring we have seen**] … [new character]

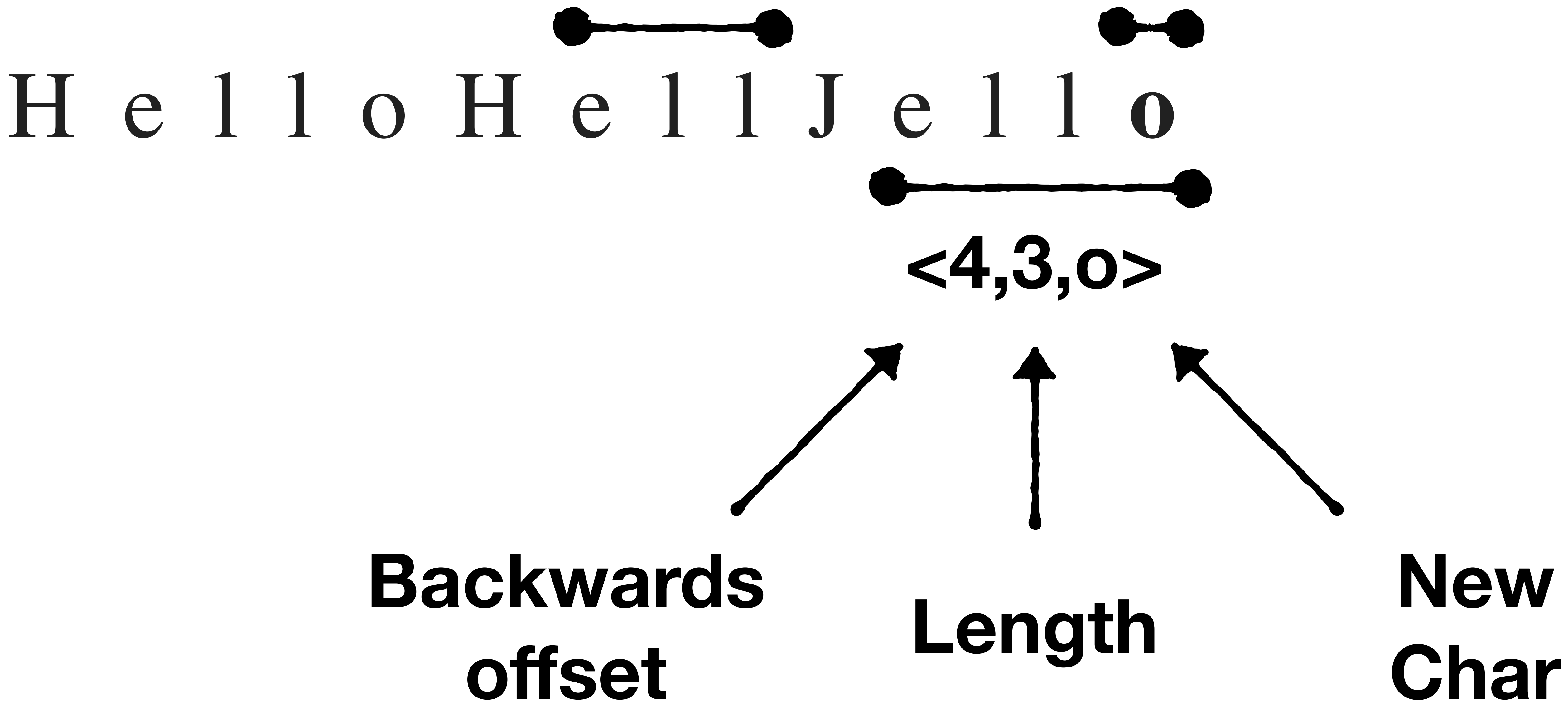
H e l l o H **e l l** J **e l l** o

Any substring can be viewed as:

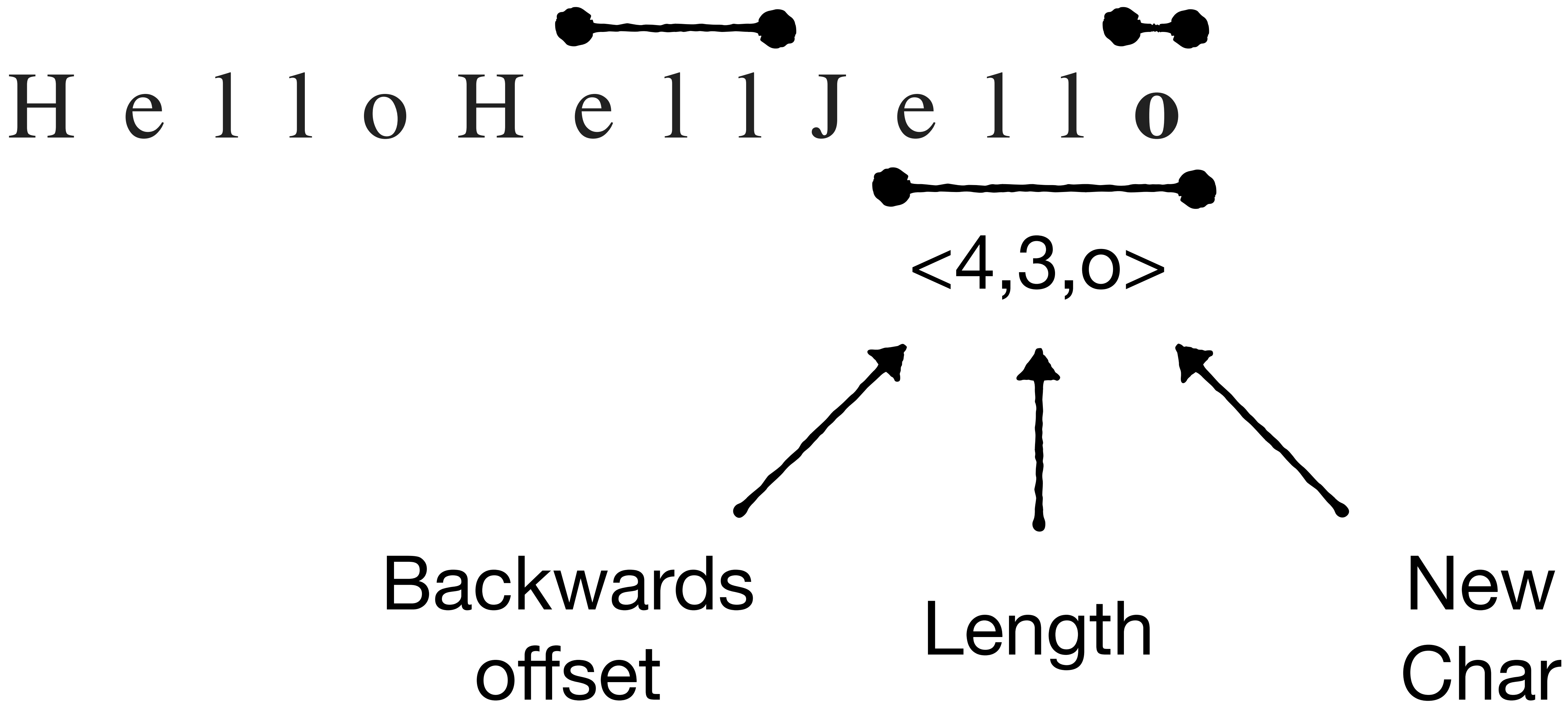
[**substring we have seen**] … [new character]

H e l l o H e l l J e l l **o**

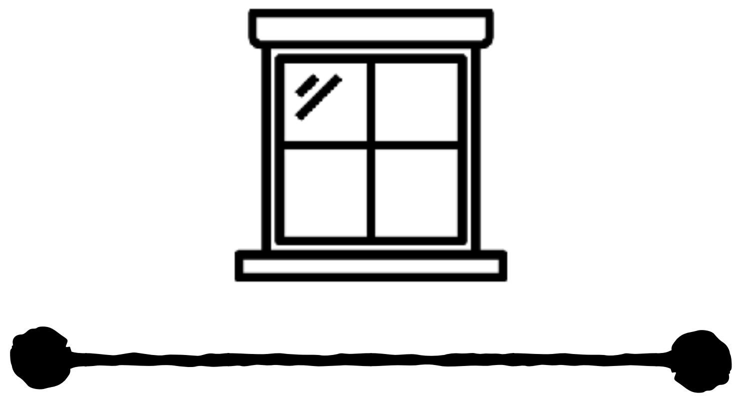
Represent as triplet



Transform string into sequence of triplets



Full Example



H e l l o H e l l J e l l o

Example

Lookahead

Search

H e l l o H e l l J e l l o

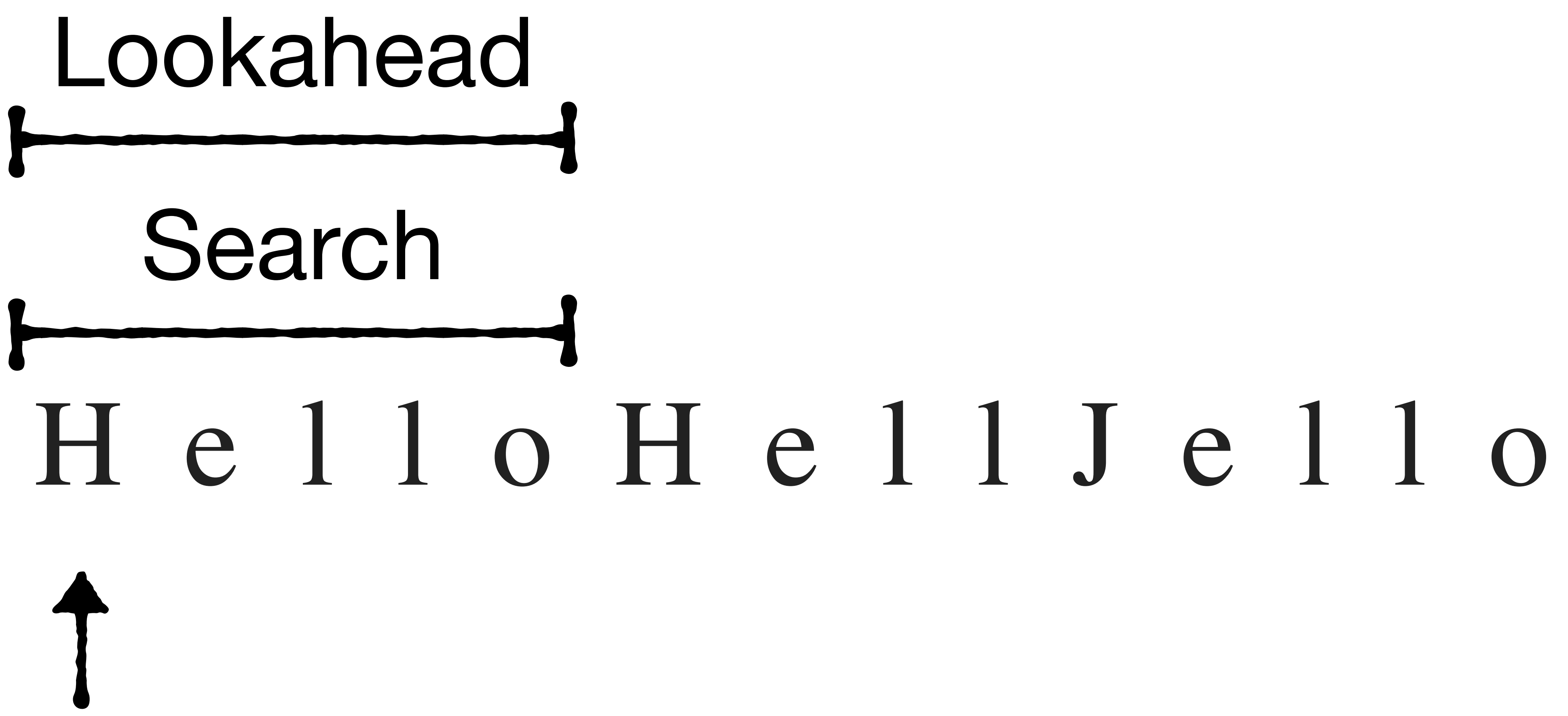
Example

Lookahead

Search

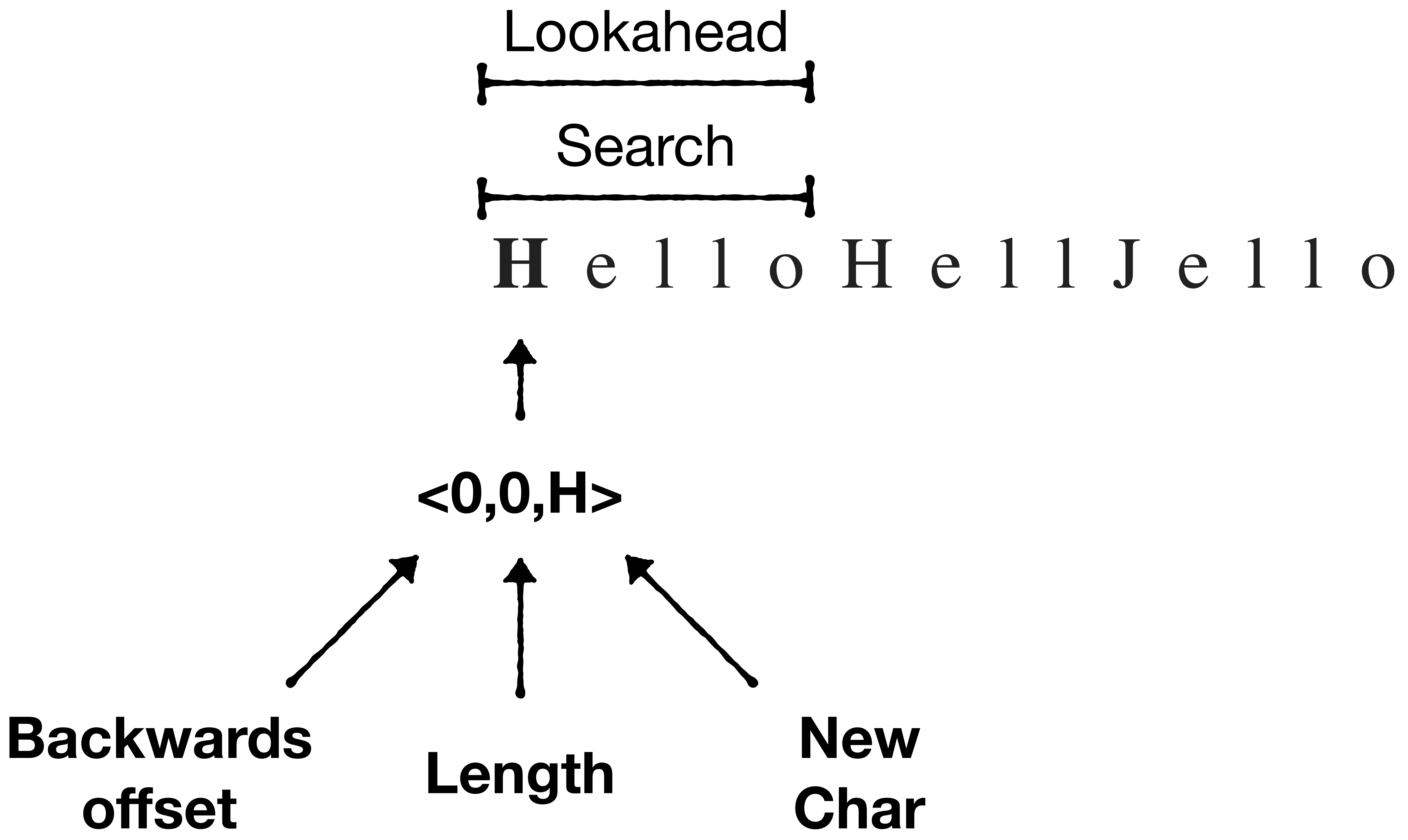
H e l l o H e l l J e l l o

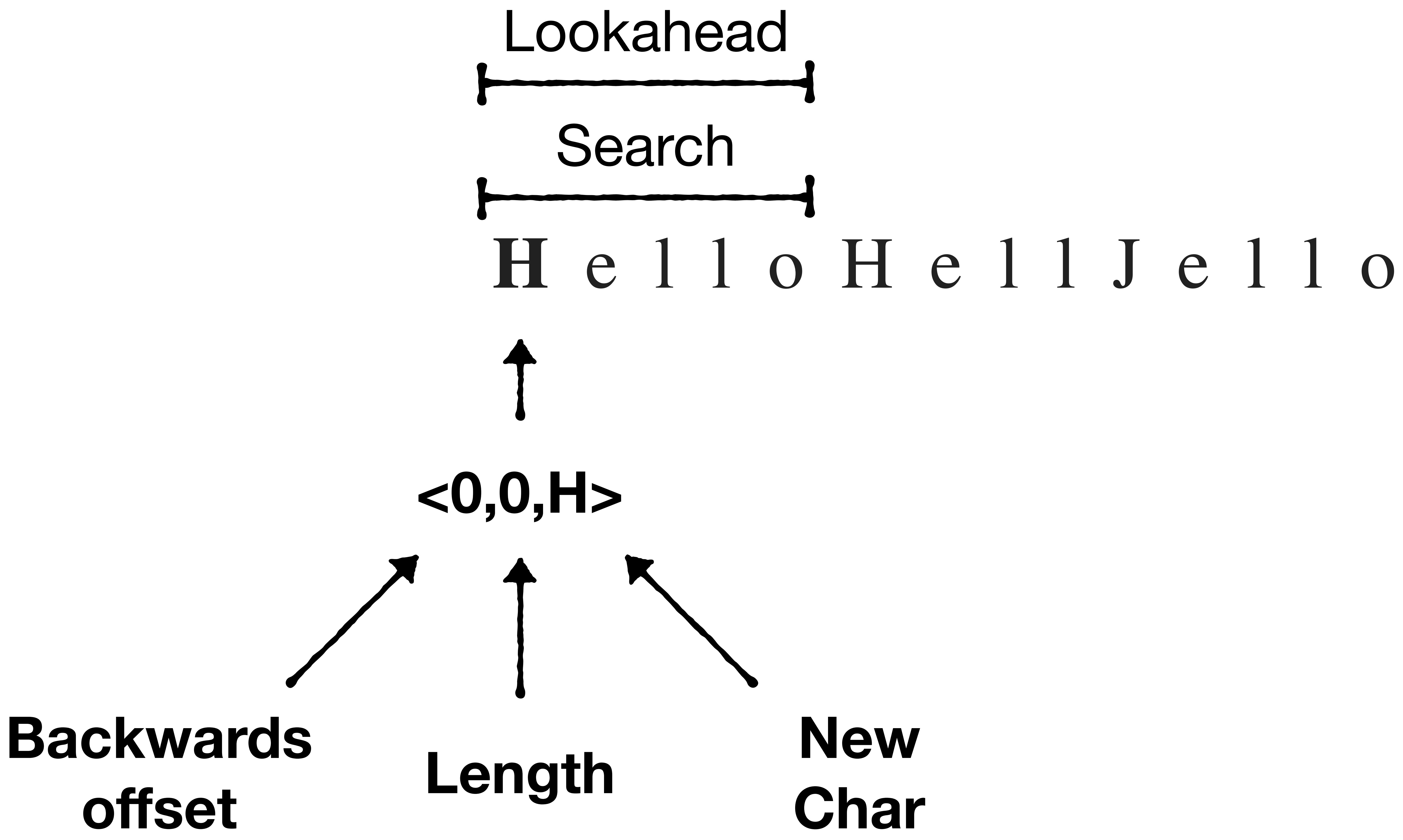
Overlap initially, but usually consecutive

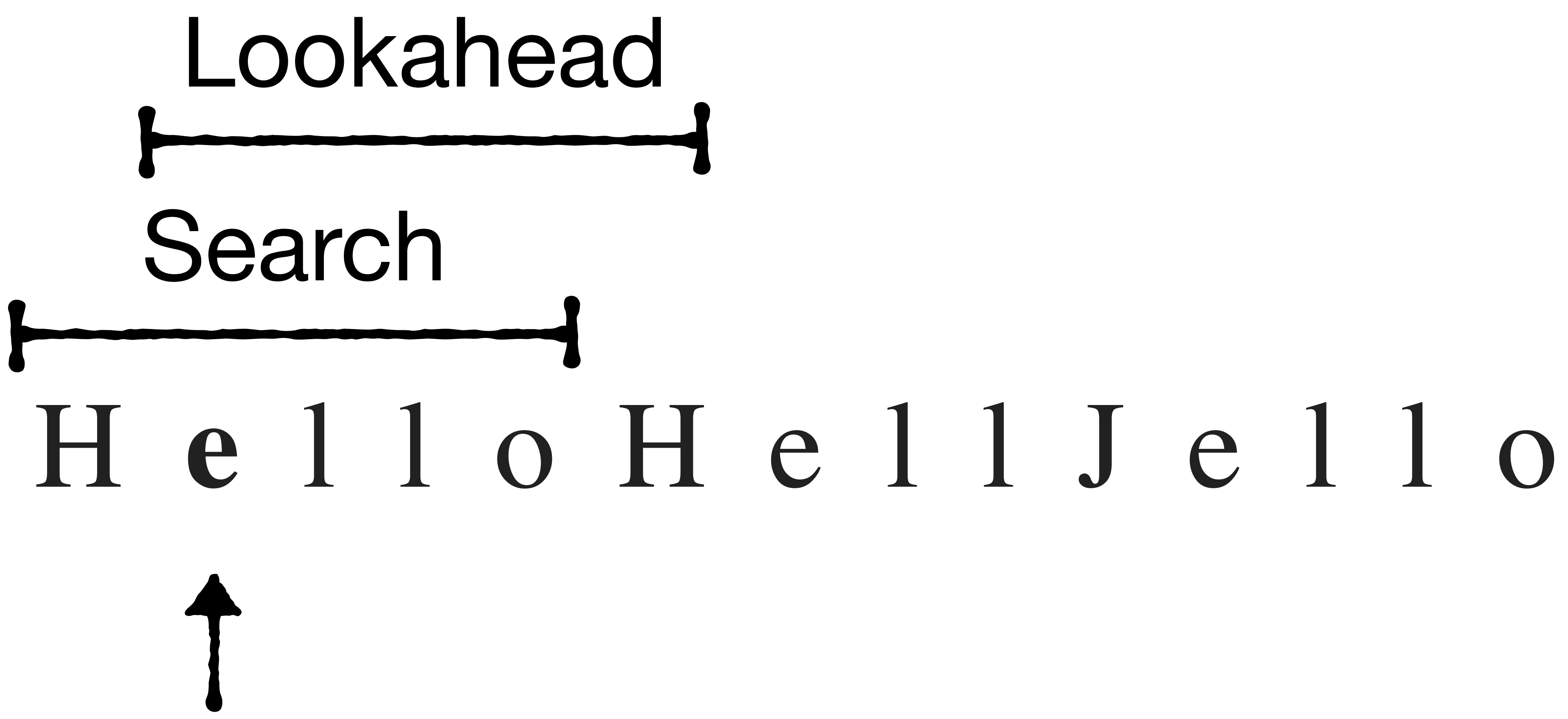


Initialize pointer to start of string

For every substring we haven’t seen yet, generate triplet




Lookahead
Search
H e l l o H e l l J e l l o
<0,0,H>
Backwards offset
Length
New Char



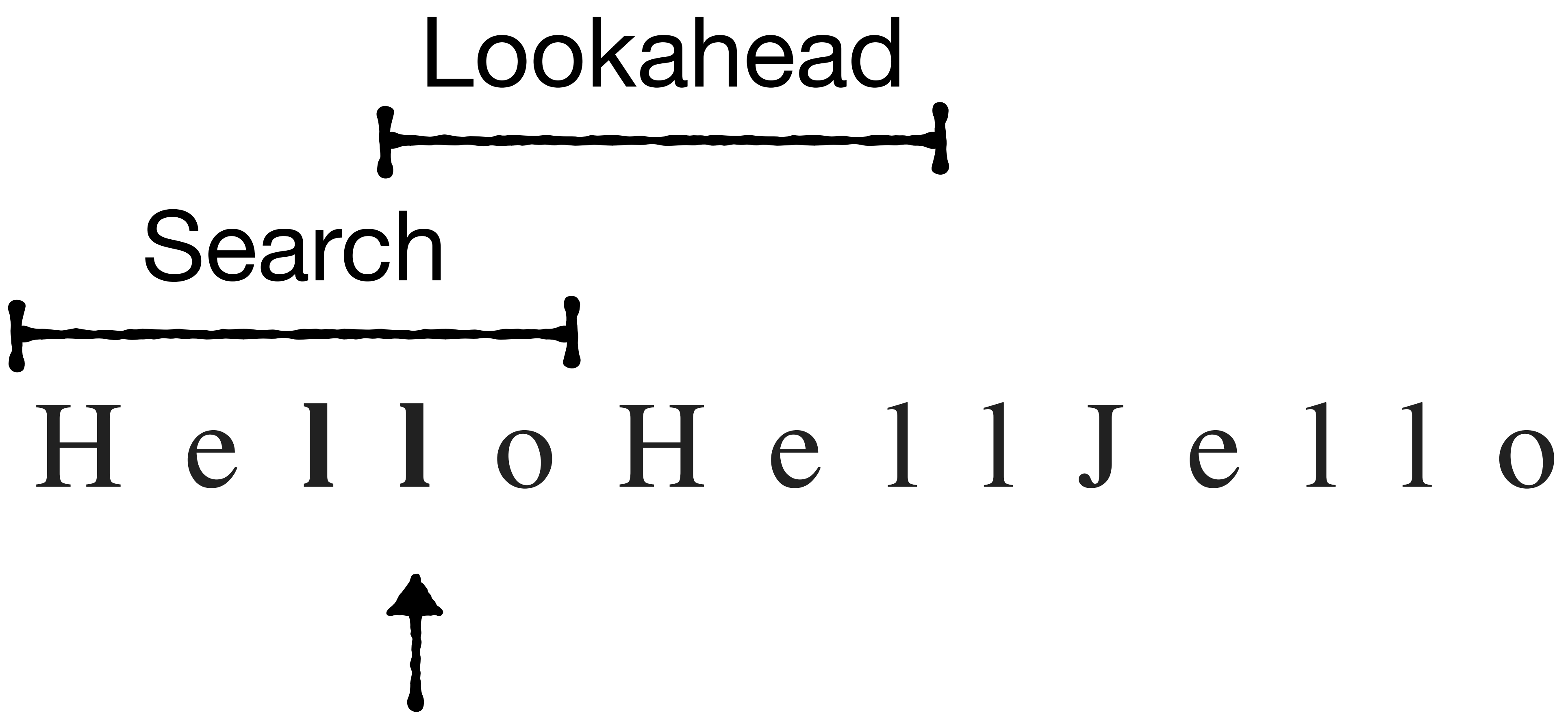
<0,0,H> **<0,0,e>**

Lookahead

Search

H e **l** l o H e l l J e l l o

<0,0,H><0,0,e> **<0,0,l>**



<0,0,H><0,0,e><0,0,l> **<1,1,>**

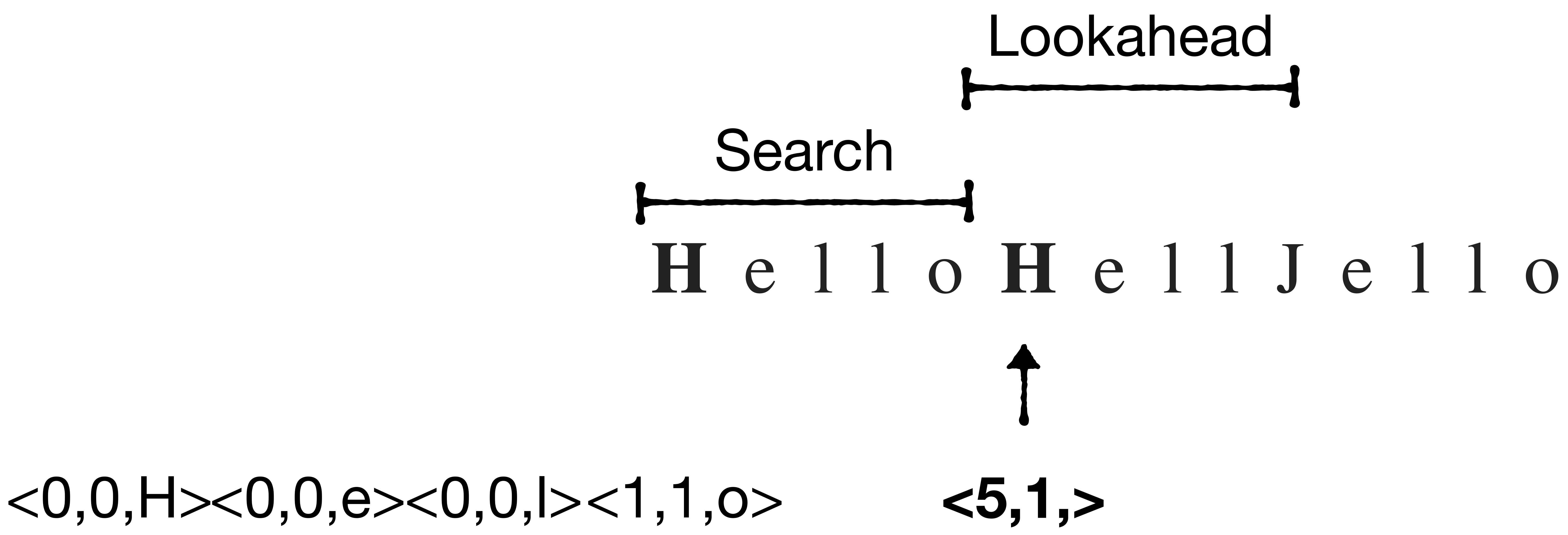
Lookahead

Search

H e **l l** o H e l l J e l l o

<0,0,H><0,0,e><0,0,l>

<1,1,o>

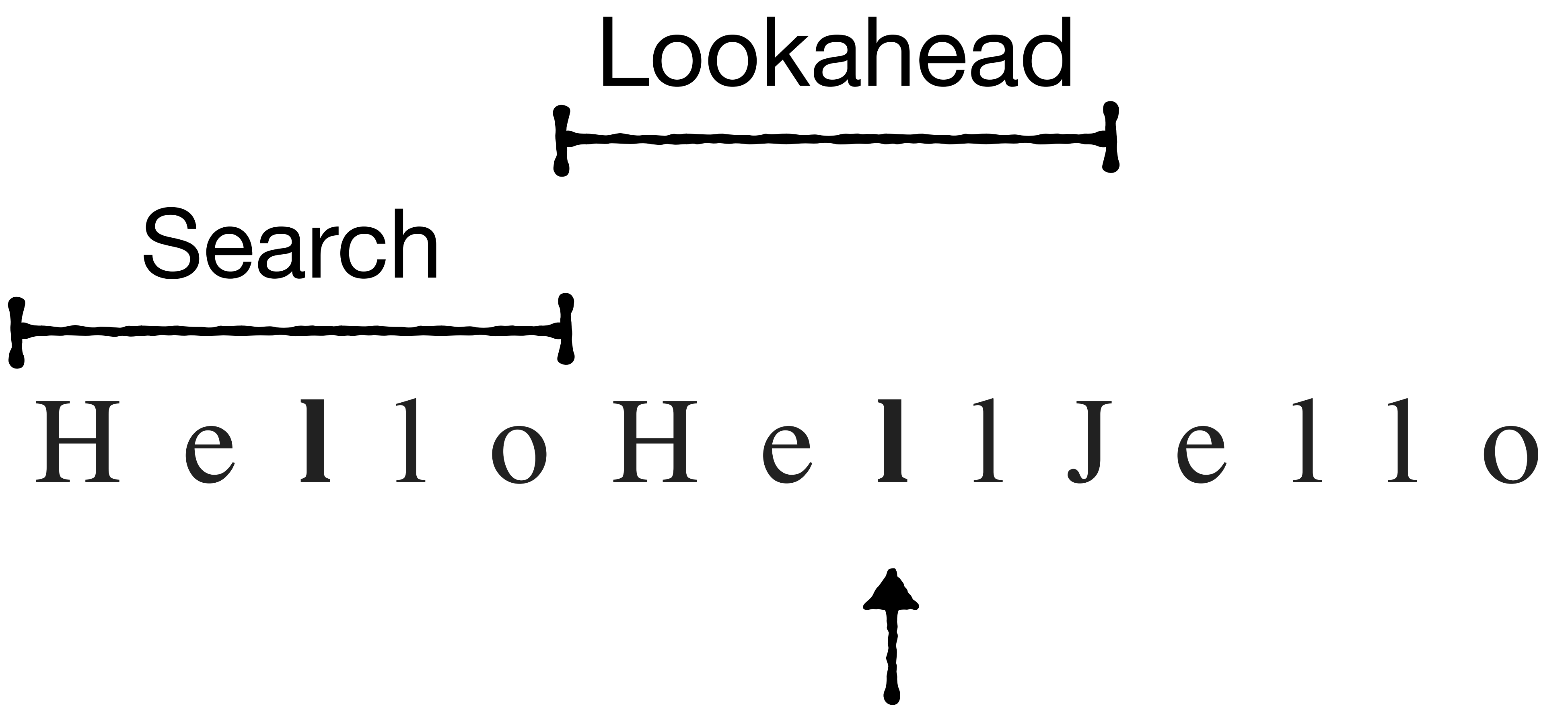

Lookahead
Search
H e l l o H e l l J e l l o
<0,0,H><0,0,e><0,0,l><1,1,o>
<5,1,>

Lookahead

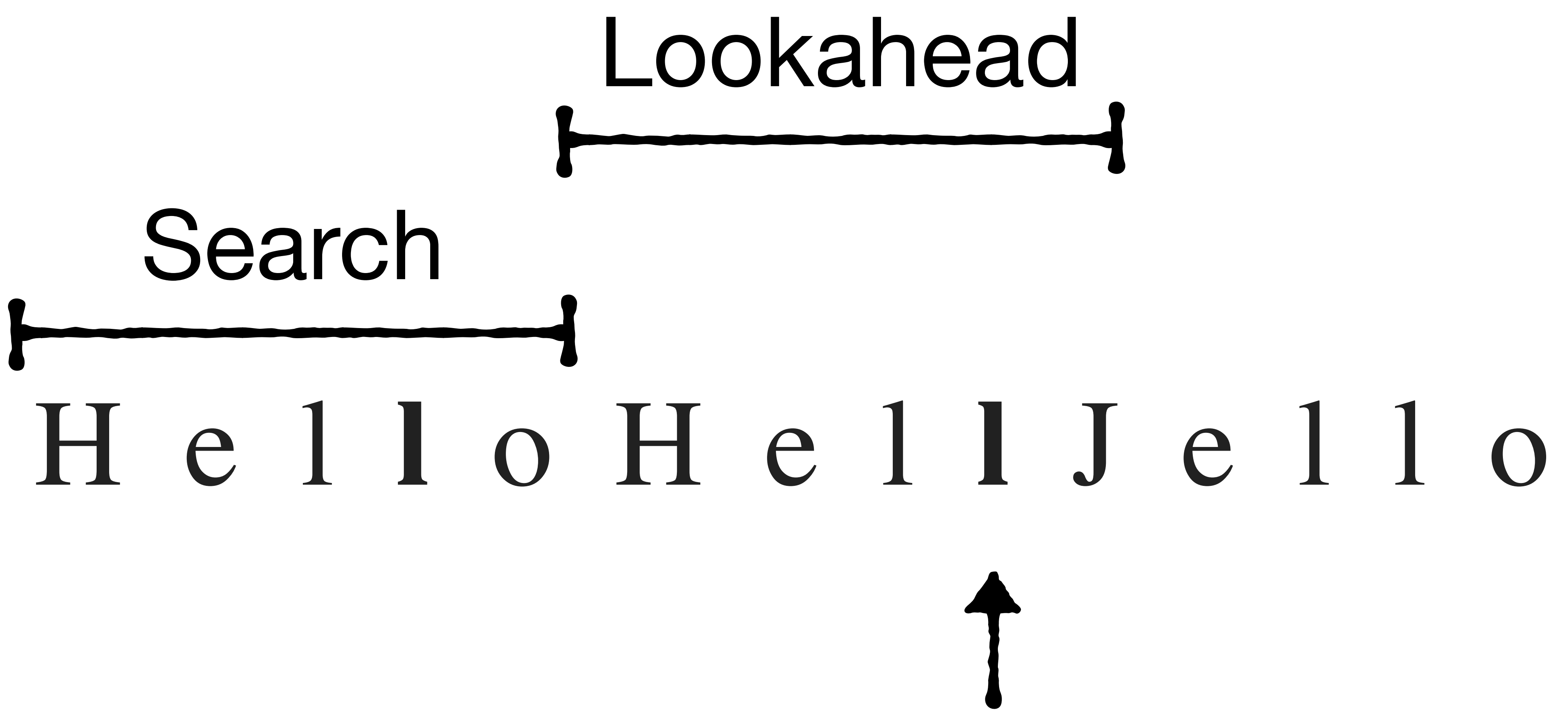
Search

H **e** l l o H **e** l l J e l l o

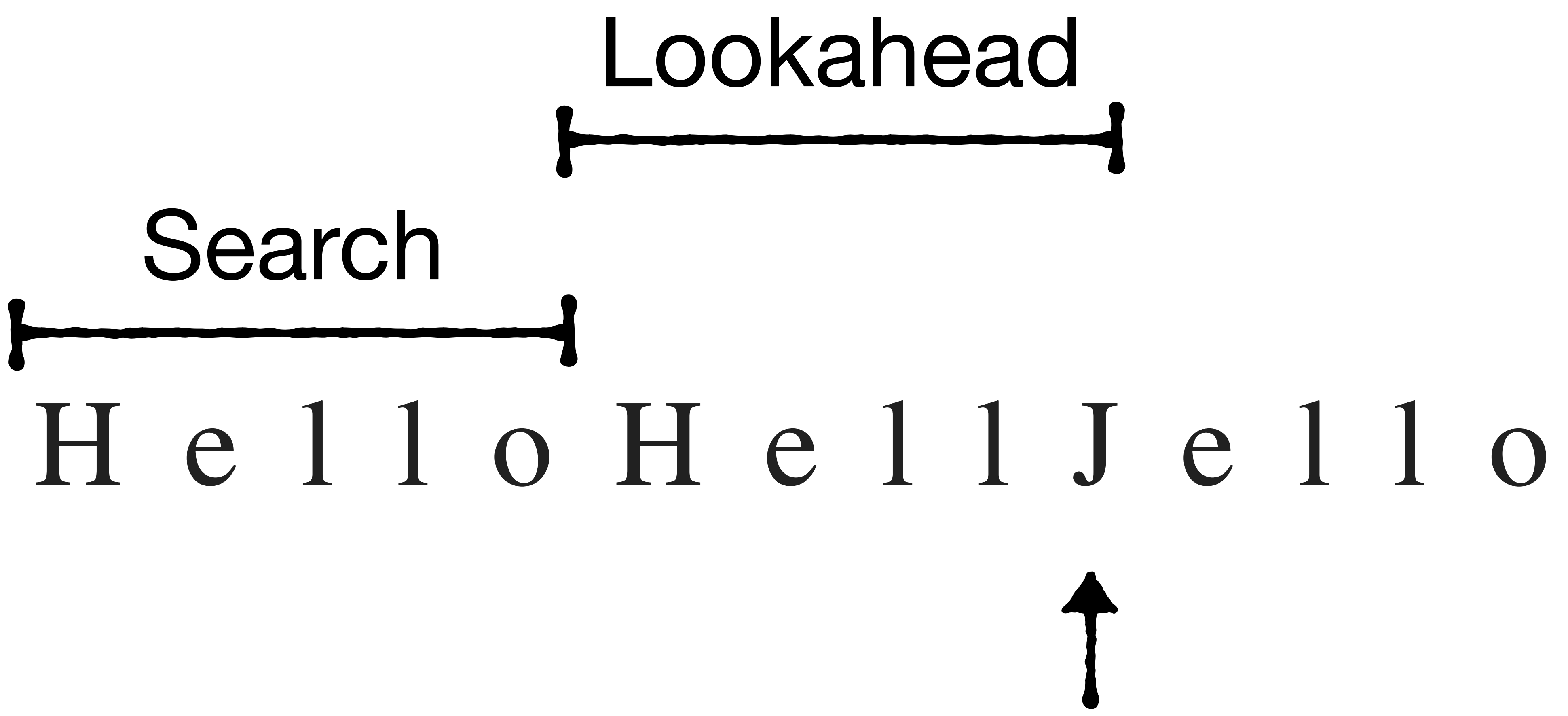
<0,0,H><0,0,e><0,0,l><1,1,o> **<5,2,>**



<0,0,H><0,0,e><0,0,l><1,1,o> **<5,3,>**



<0,0,H><0,0,e><0,0,l><1,1,o> **<5,4,>**



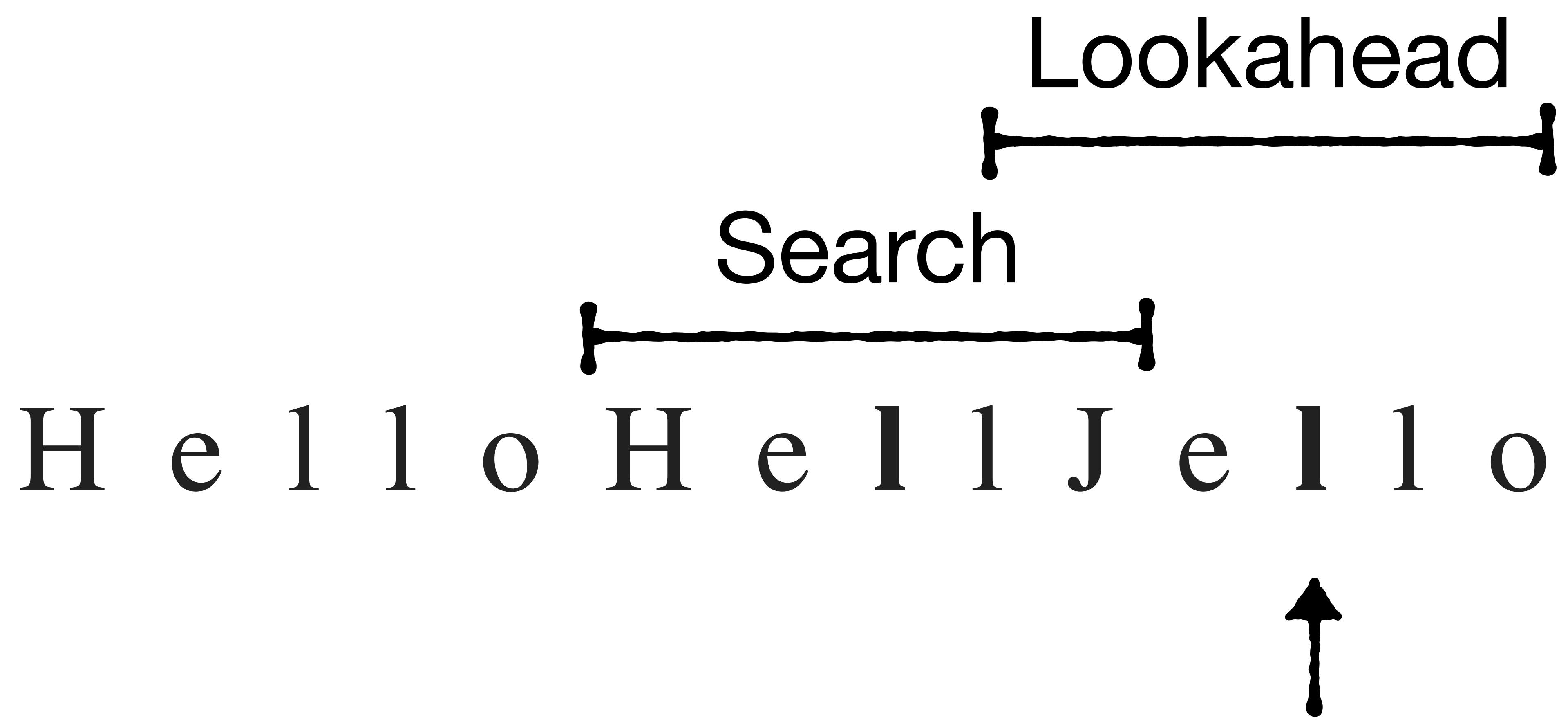
<0,0,H><0,0,e><0,0,l><1,1,o> **<5,4,J>**

Lookahead

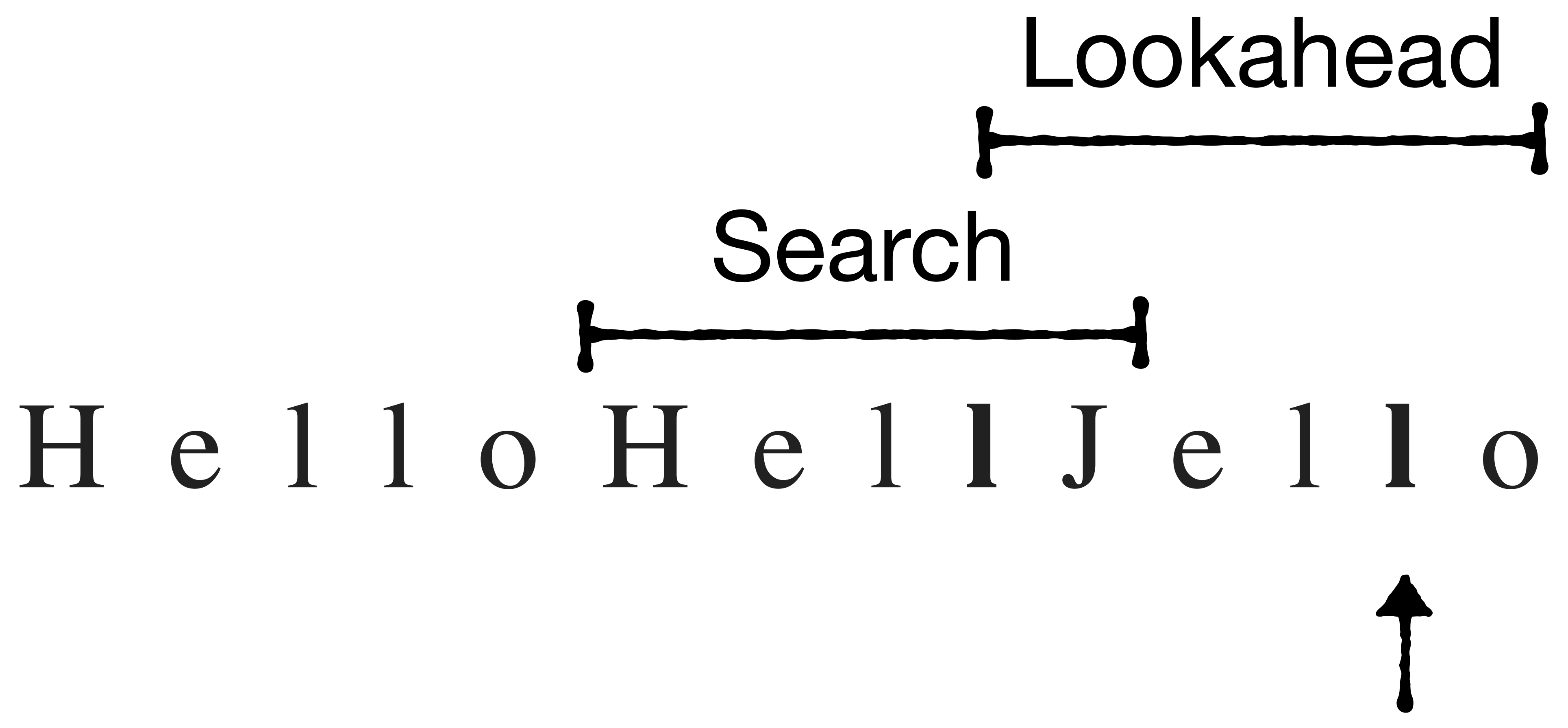
Search

H e l l o H **e** l l J **e** l l o

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J> **<4,1,>**



<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J> **<4,2,>**



<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J>

<4,3,>

Lookahead

Search

H e l l o H e l l J e l l **o**

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J> **<4,3,o>**

Lookahead

Search

H e l l o H e l l J e l l o

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J><4,3,o> **<0,0,end>**

H e l l o H e l l J e l l o

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J><4,3,o><0,0,end> **We’re done :)**

LZ77: Basis of most modern compression libraries

Snappy

LZ4

ZSTD

How big is each field in a triplet?

<backwards offset, length, char>

How big is each field in a triplet?

<**backwards offset**, length, char>

$\log_2$(search buffer size)

How big is each field in an LZ77 triplet?

<backwards offset, **length**, char>

$\log_2$(lookahead buffer size)

How big is each field in an LZ77 triplet?

<backwards offset, length, **char**>

?

How big is each field in an LZ77 triplet?

<backwards offset, length, **char**>

?

The smaller this is, the more triplets there are, yet the more fine-grained compression we get

How big is each field in an LZ77 triplet?

<backwards offset, length, **char**>

1-4 bytes (can be var-length)

The smaller this is, the more triplets there are, yet the more fine-grained compression we get

What’s the runtime with respect to:

Window size W

H e l l o H e l l J e l l o

The number of characters N

What’s the runtime with respect to:

Window
size **W**

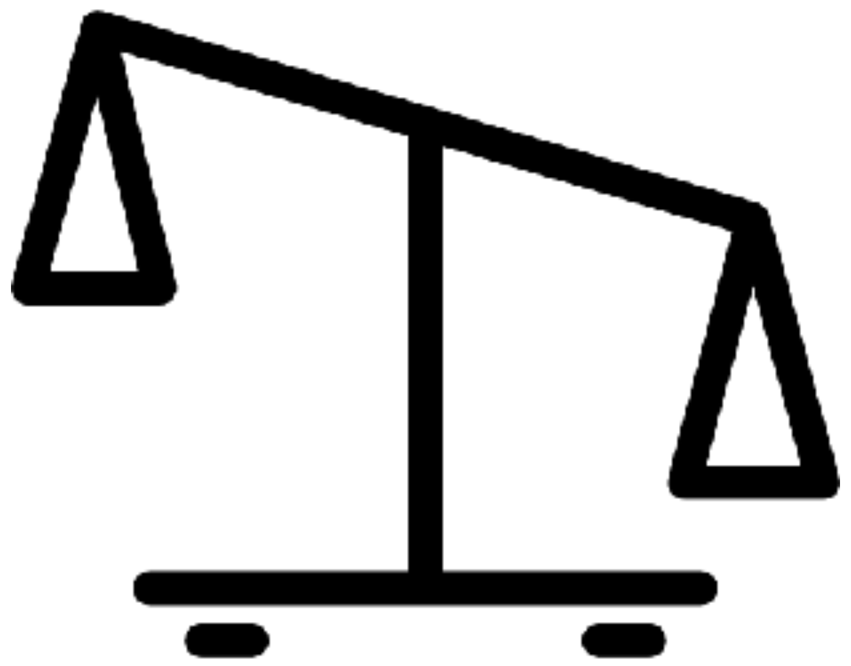
H e l l o H e l l J e l l o

The number of characters **N**

O(N · W)

Trade-off governed by window size

Compression rate



CPU

$O(N \cdot W)$

Page Sizing

Delta Encoding

Restarts

Page Hash Index

LZ77

Huffman coding

Dictionary training

Consider the following compressed sequence

<0,0,C> <0,0,A> <0,0,B> <1,4,A> <5,4,B> <5,4,A> …

<0,0,C> <0,0,A> <0,0,B> <1,4,A> <**5**,4,B> <**5**,4,A> …

There may be some patterns in the backwards offset

<0,**0**,C> <0,**0**,A> <0,**0**,B> <1,**4**,A> <5,**4**,B> <5,**4**,A> …

0 **4**

Some lengths may repeat more frequently

<0,0,C> <0,0,**A**> <0,0,B> <1,4,**A**> <5,4,B> <5,4,**A**> …

Some characters may repeat more than others

Can we assign smaller codes to more frequent symbols?

<0,0,C> <0,0,**A**> <0,0,B> <1,4,**A**> <5,4,B> <5,4,**A**> …

Example: **A - 1**

B - 01

C - 00

Huffman Coding (HF)

Frequent symbols $\longrightarrow$ Smaller codes

Huffman Coding (HF)

Frequent symbols $\longrightarrow$ Smaller codes

Infrequent symbols $\longrightarrow$ Longer codes

Huffman Coding (HF)

Frequent symbols → Smaller codes

Infrequent symbols → Longer codes

Goal: encode each symbol succinctly using the symbol's relative frequency

Huffman Coding (HF)

Frequent symbols $\longrightarrow$ Smaller codes

Infrequent symbols $\longrightarrow$ Longer codes

Goal: encode each symbol succinctly using the symbol's relative frequency

In contrast, LZ77 removes repeating substrings

Huffman Coding

B C A A C C B A B B C A D E C A A C B C B B C A B A B A A D

Huffman Coding

B C A A C C B A B B C A D E C A A C B C B B C A B A B A A D

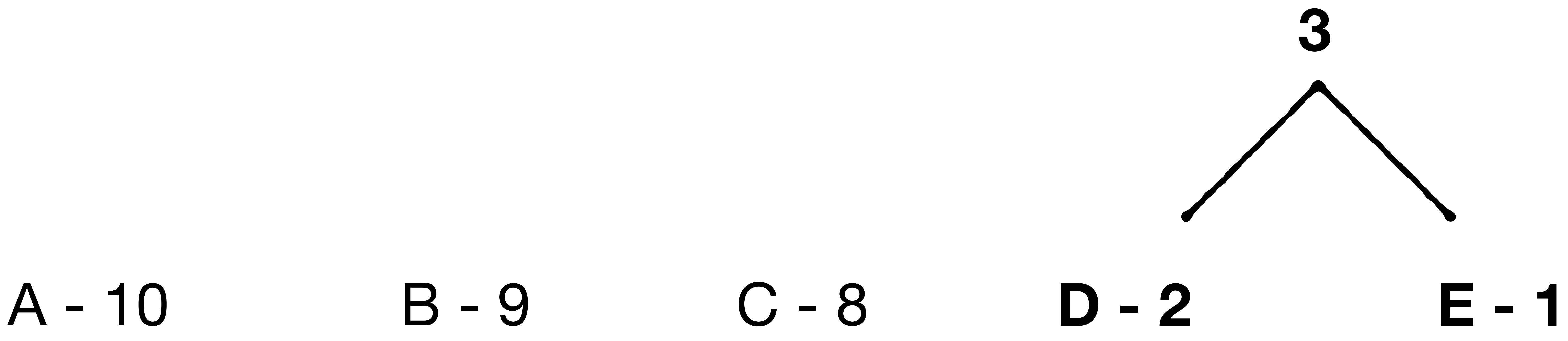
Character:	**A**	**B**	**C**	**D**	**E**
Histogram:	**10**	**9**	**8**	**2**	**1**

Huffman Coding

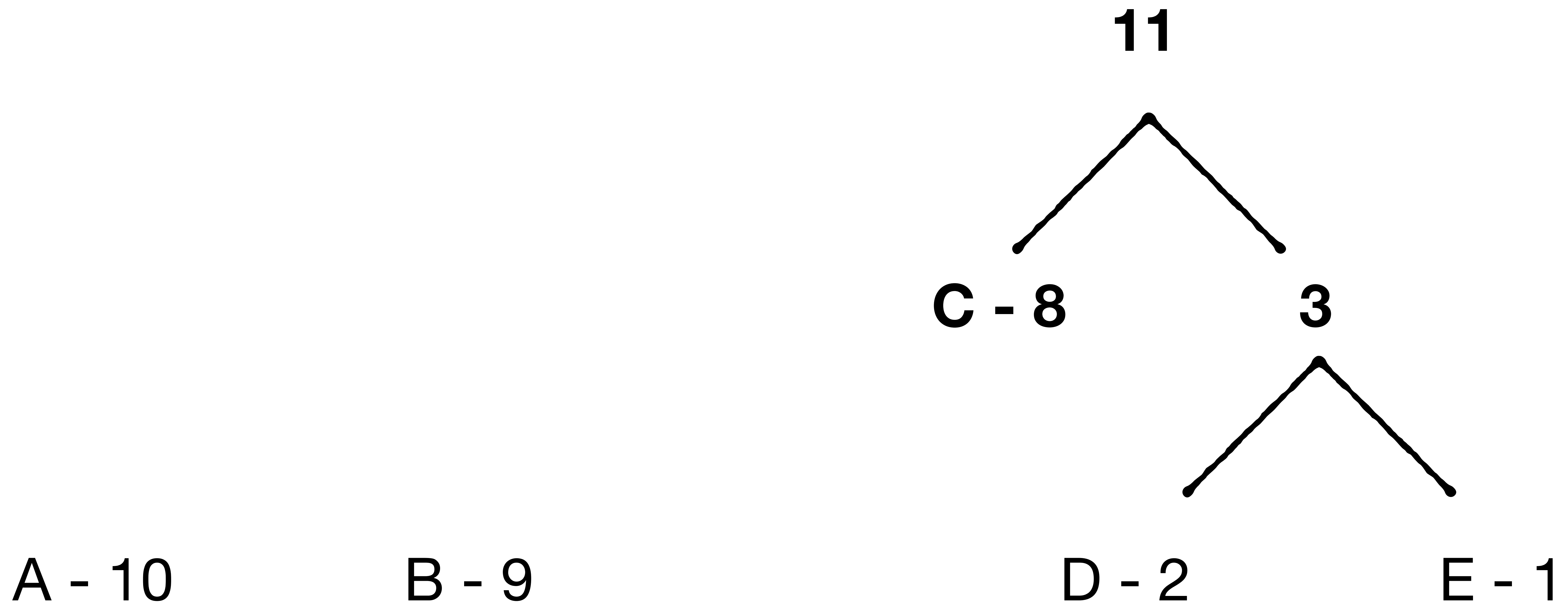
Let each character be a subtree

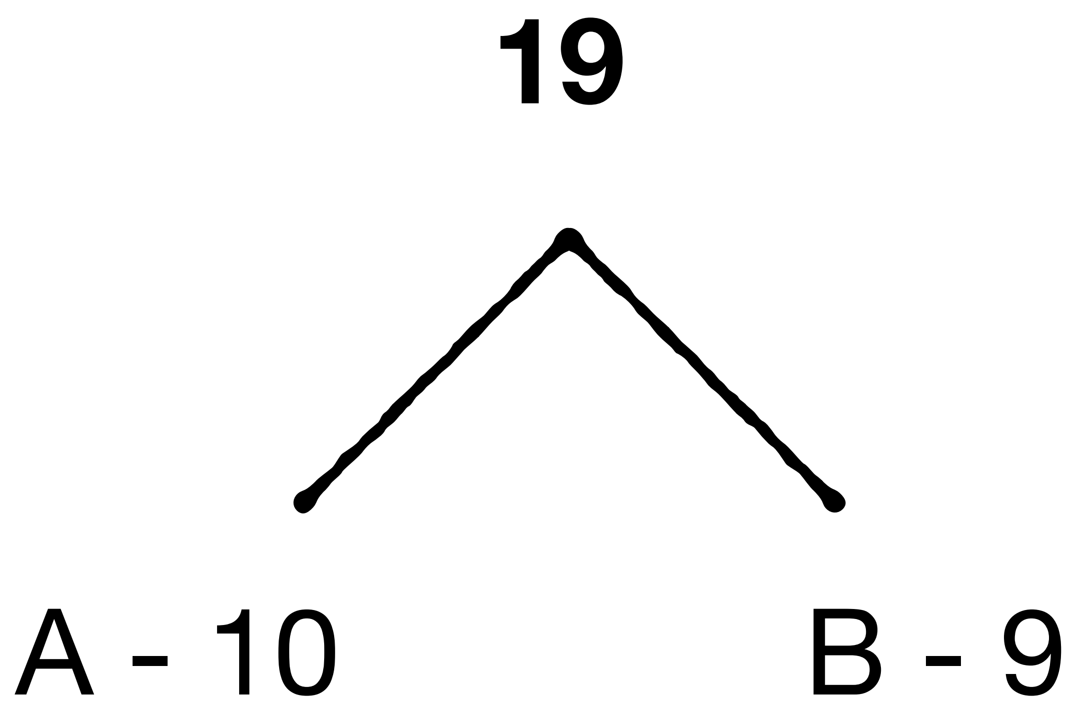
A - 10 **B - 9** **C - 8** **D - 2** **E - 1**

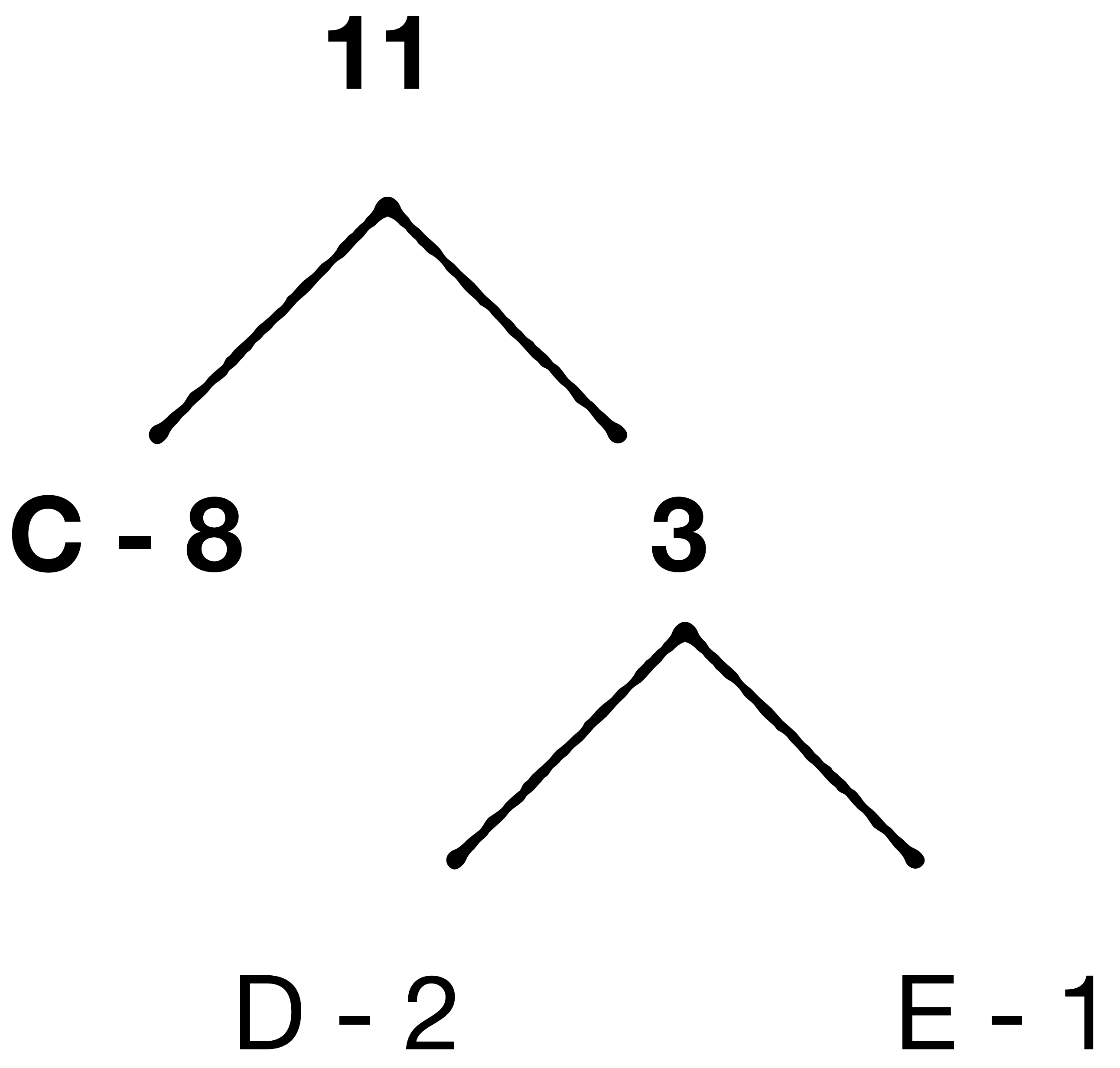
Connect sub-trees with lowest cumulative frequency

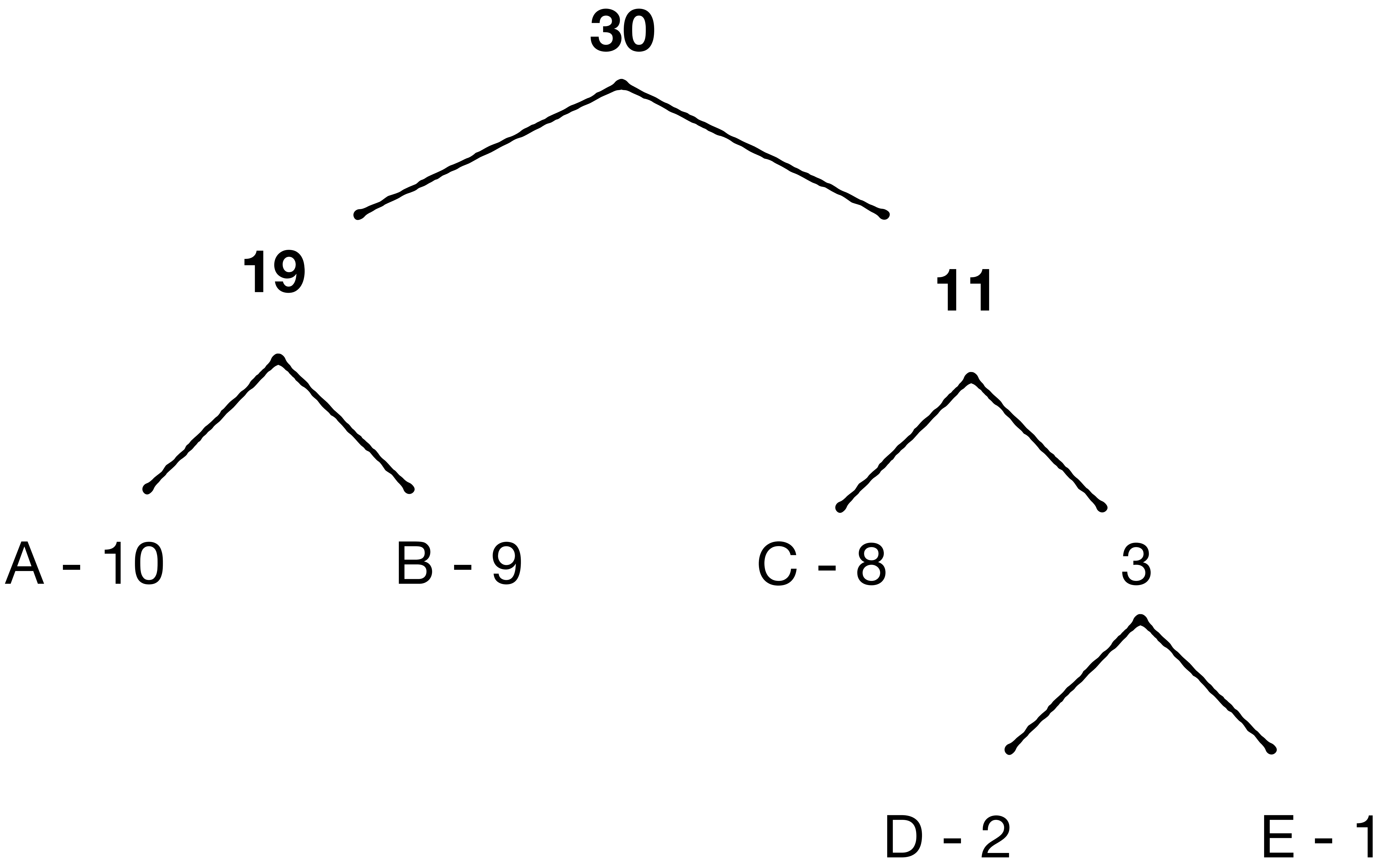


Connect sub-trees with lowest cumulative frequency



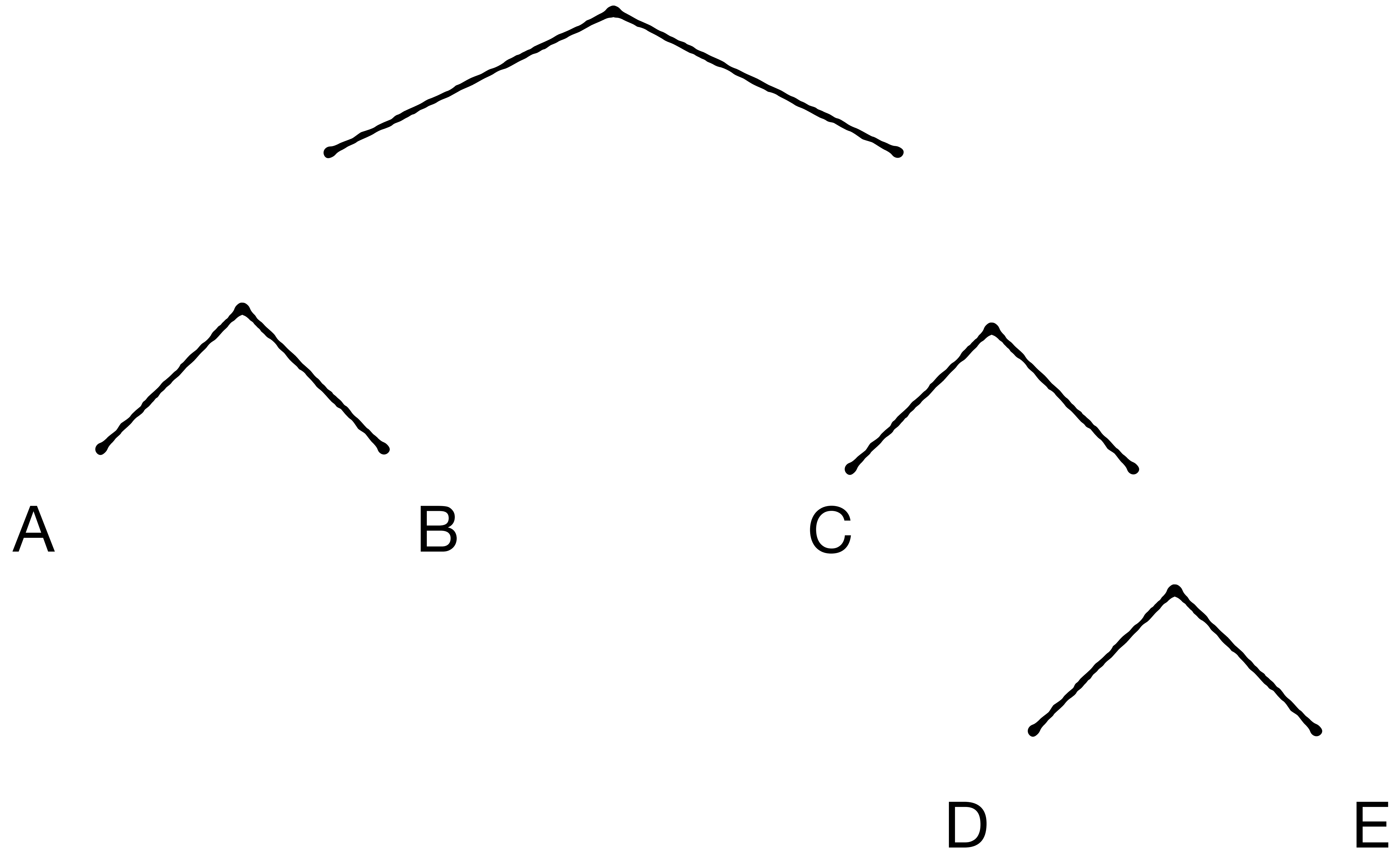

19
A - 10
B - 9


11
C - 8
3
D - 2
E - 1

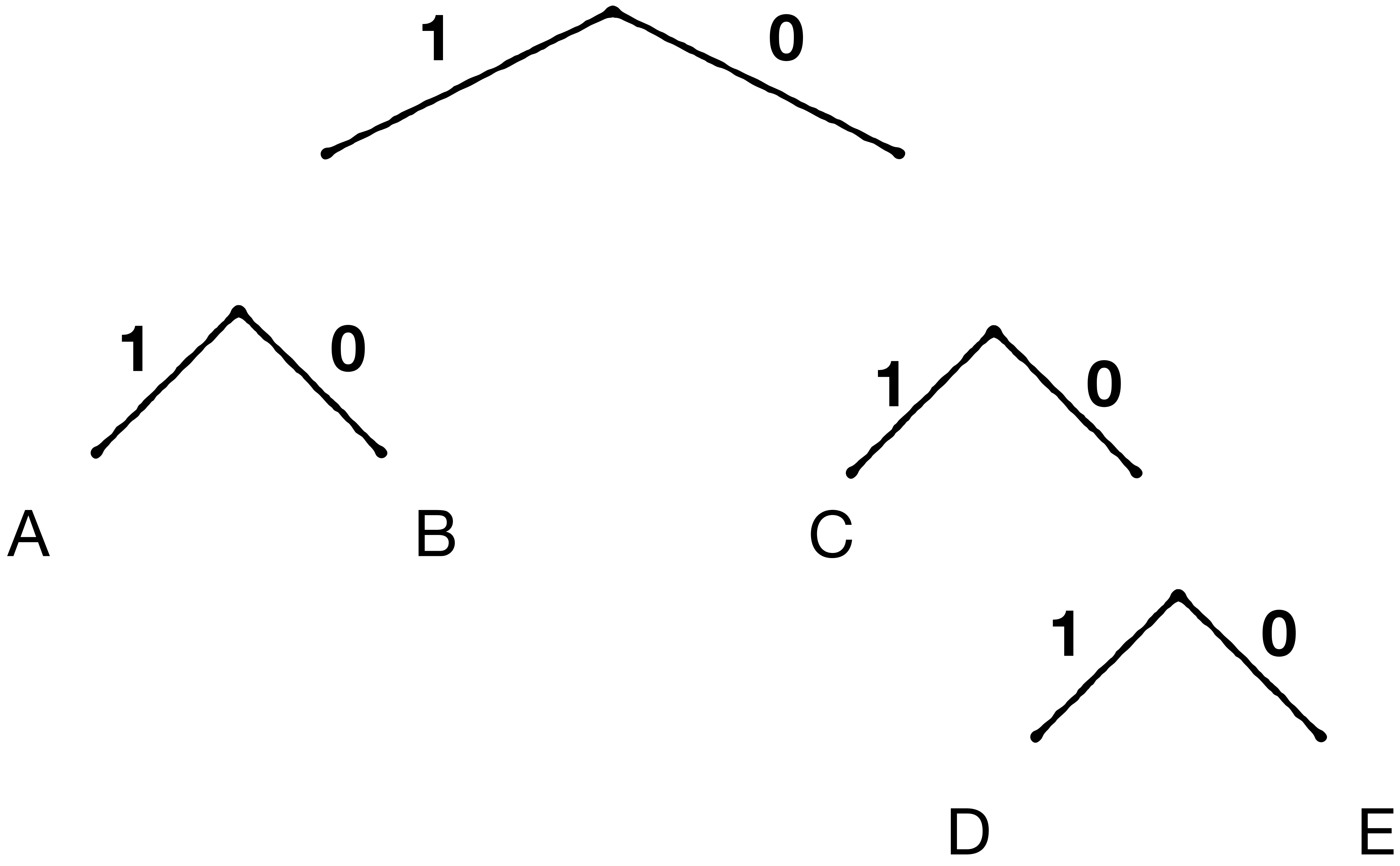


30
19
11
A - 10
B - 9
C - 8
3
D - 2
E - 1

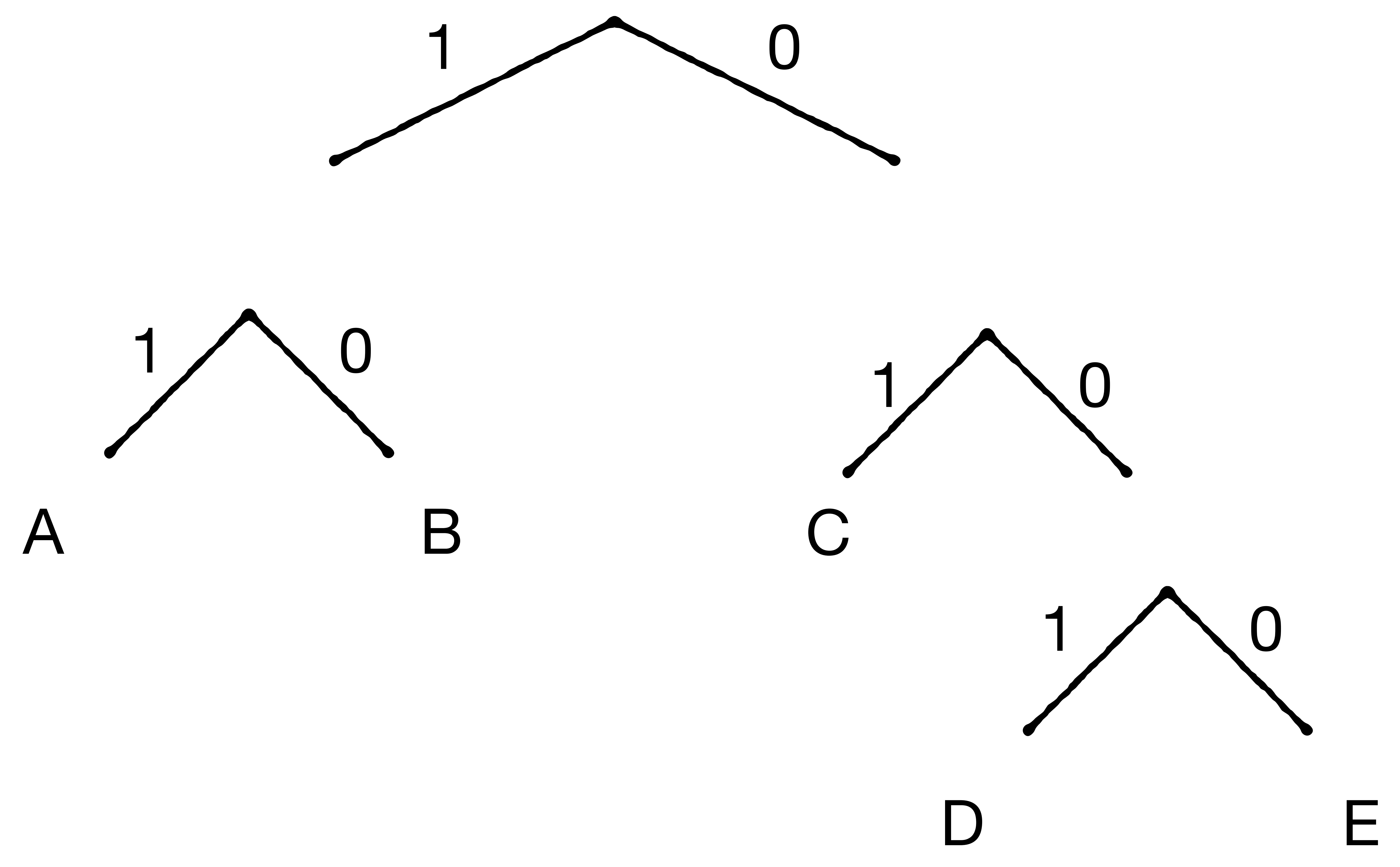
Remove frequency labels



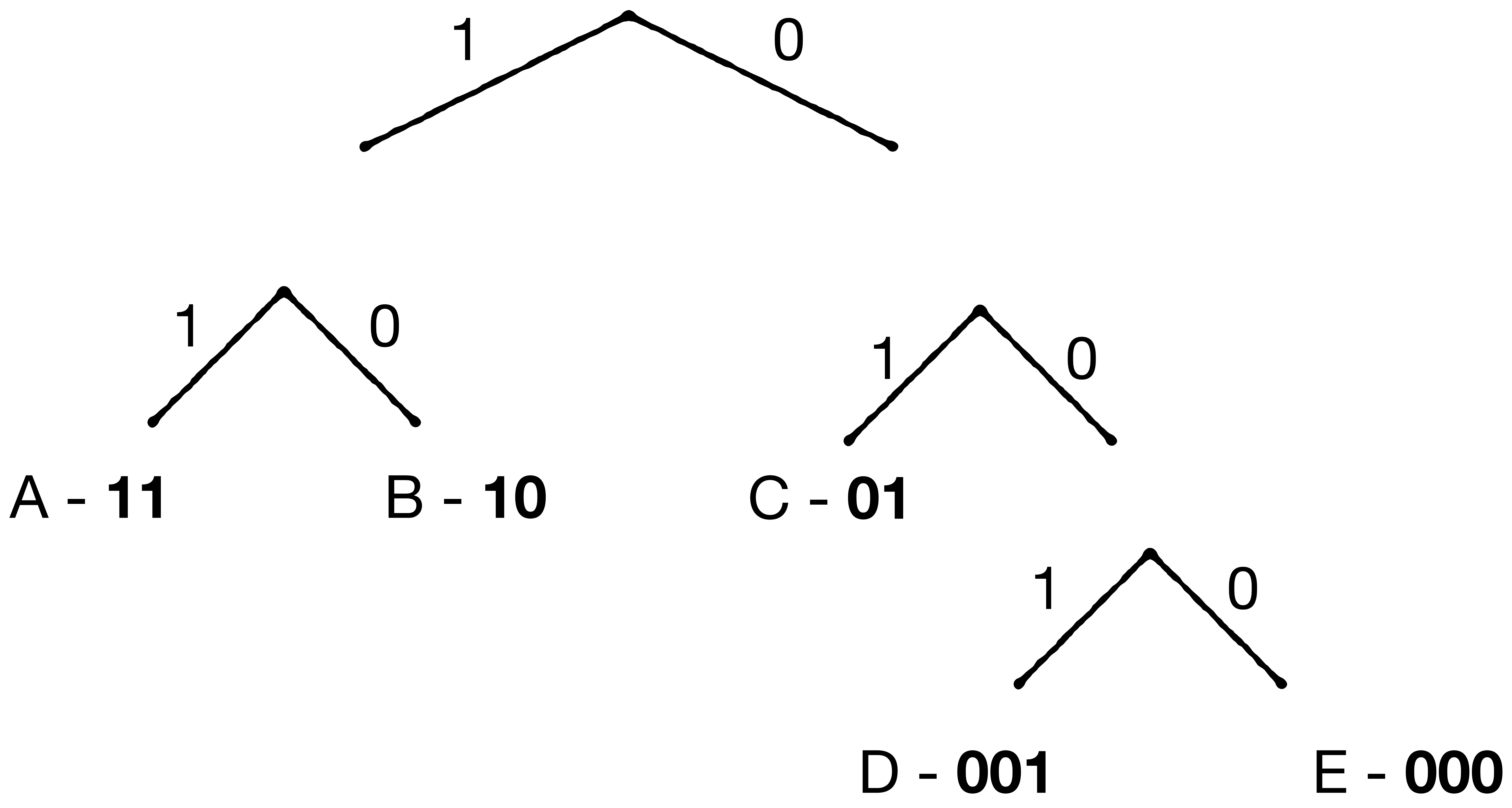
Label edges with zeros and ones



The code for each entry is given by its path from the room



The code for each entry is given by its path from the room



Final codes

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

Prefix code - no code is a prefix of another code

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

Prefix code - no code is a prefix of another code

Allows to uniquely decode a bit string

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

More frequent symbols are assigned smaller codes

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

More frequent symbols are assigned smaller codes

Optimal avg. code length

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

Construction Algorithm & Runtime?

A - **11**

B - **10**

C - **01**

D - **001**

E - **000**

Algorithm: **Use priority queue to always give two sub-trees with lowest frequency**

Runtime: **$O(s \cdot \log_2(s))$** **For S symbols**

Back to LZ77

Suppose we now have triplets

<0,0,C> <0,0,A> <0,0,B> <1,4,A> <5,4,B> <5,4,A> …

Back to LZ77

Suppose we now have triplets

<0,0,C> <0,0,A> <0,0,B> <1,4,A> <5,4,B> <5,4,A> …

A - 1

B - 01

C - 00

And we have Huffman codes

Plug the Huffman codes into the triplets

<0,0,**C**> <0,0,**A**> <0,0,**B**> <1,4,**A**> <5,4,**B**> <5,4,**A**> …

A - 1

B - 01

C - 00

Plug the Huffman codes into the triplets

<0,0,**00**><0,0,**1**><0,0,**01**><1,4,**1**><5,4,**01**><5,4,**1**> …

<0,0,**00**><0,0,**1**><0,0,**01**><1,4,**1**><5,4,**01**><5,4,**1**> …

Improves compression rate

Degrades de/compression speed

<0,0,**00**><0,0,**1**><0,0,**01**><1,4,**1**><5,4,**01**><5,4,**1**> …

- Improves compression rate
- Degrades de/compression speed
  - **Building the Huffman tree takes time**
  - **Triplets are not byte-aligned**
  - **Decompressing each code requires a Huffman tree search**

Page Sizing

Delta Encoding

Restarts

Page Hash Index

LZ77

Huffman coding

Dictionary Training

Dictionary Training

LZ77 + Huffman eliminates redundancy in a single page

<K1, Hello> <K2, Hell> <K3, Jello>

Dictionary Training

But the same redundancy may exist across multiple adjacent pages

<K1, Hello> <K2, Hell> <K3, Jello>

<K4, Hello> <K5, Hell> <K6, Jello>

But the same redundancy may exist across multiple adjacent pages

<K1, Hello> <K2, Hell> <K3, Jello>	<K4, Hello> <K5, Hell> <K6, Jello>

How can we eliminate redundancy across pages to improve the compression rate?

Example:

H e l l o H e l l J e l l o

H e l l o H e l l J e l l o

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J><4,3,o><0,0,end><0,0,H><0,0,e><0,0,l><1,1,o><5,4,J><4,3,o><0,0,end>

Example:

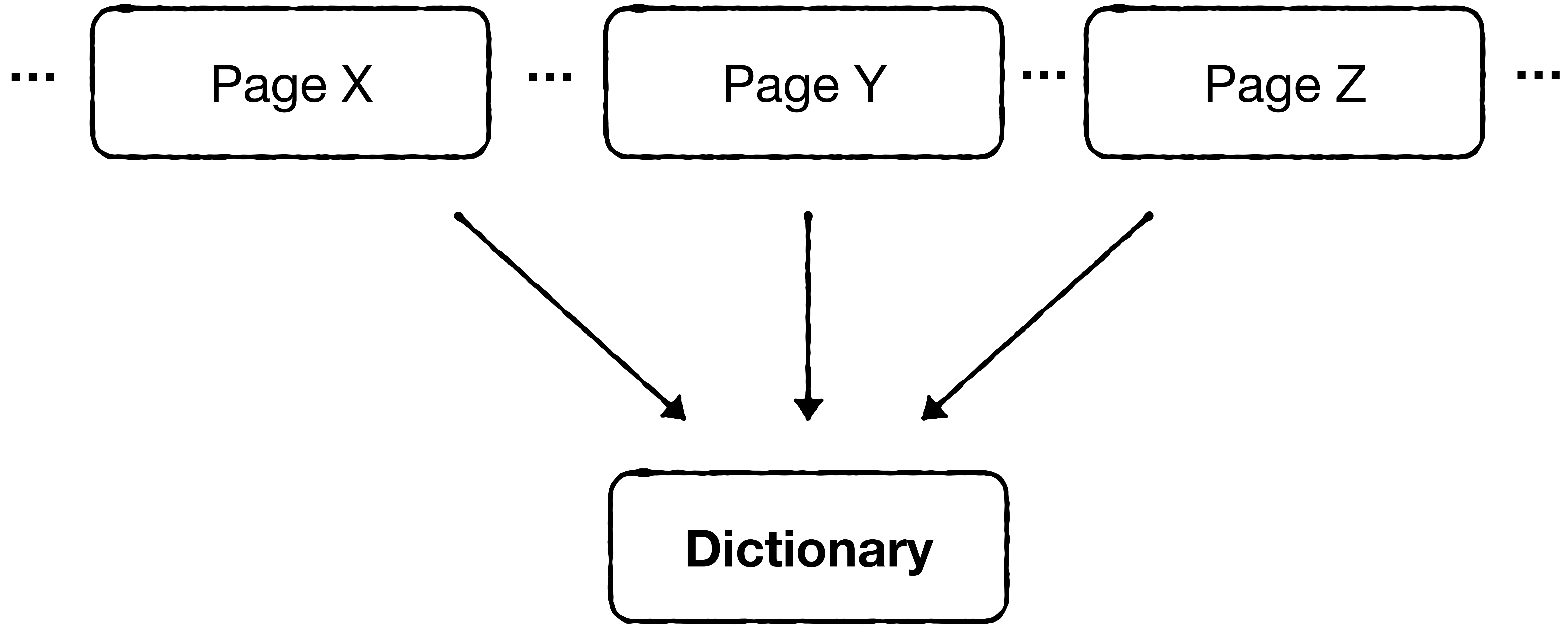
H e l l o H e l l J e l l o | H e l l o H e l l J e l l o

<0,0,H><0,0,e><0,0,l><1,1,o><5,4,J><4,3,o><0,0,end>

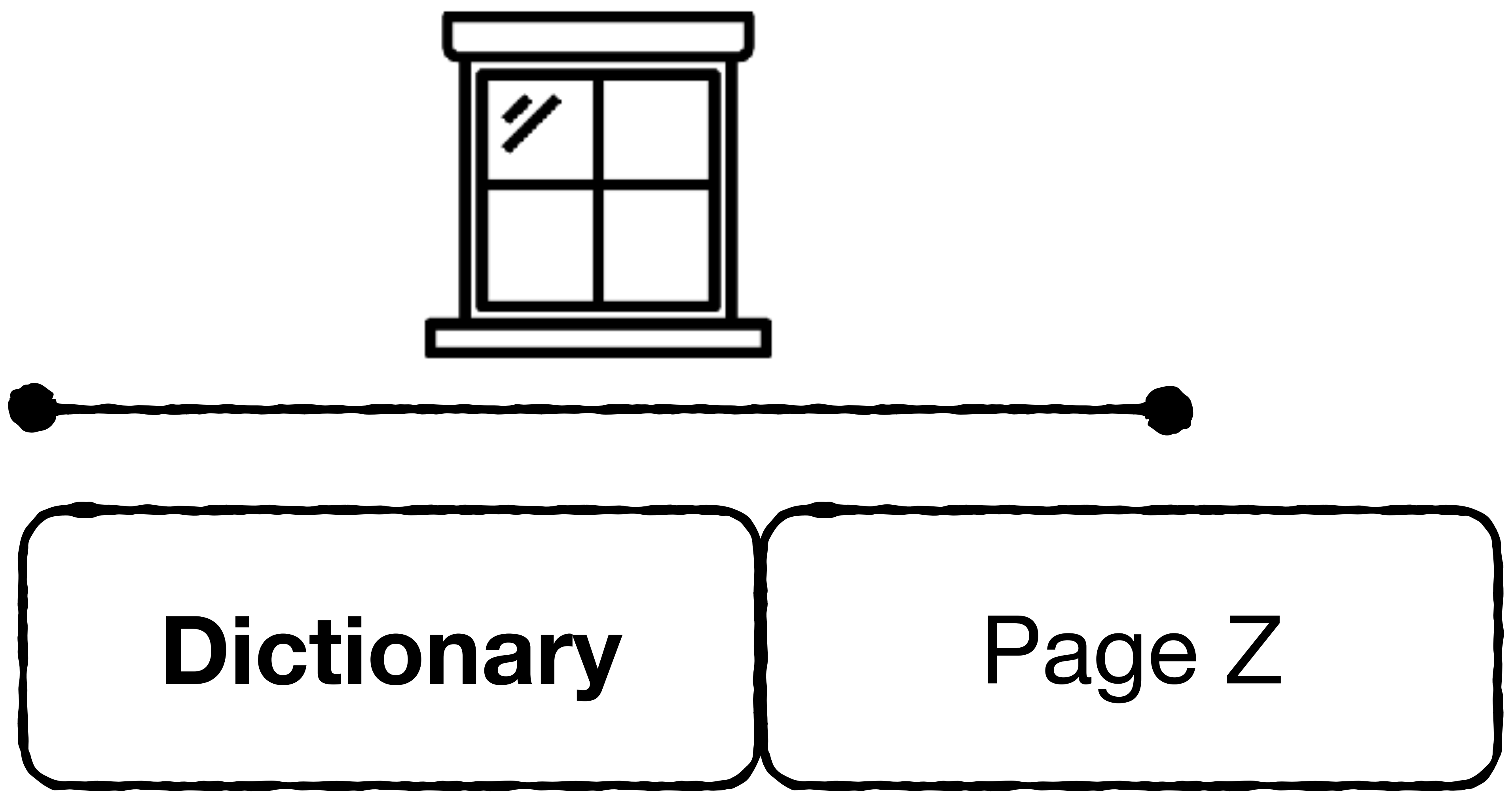
<14,14,end>

Fewer triplets if we could compress relative to redundancy we have already seen

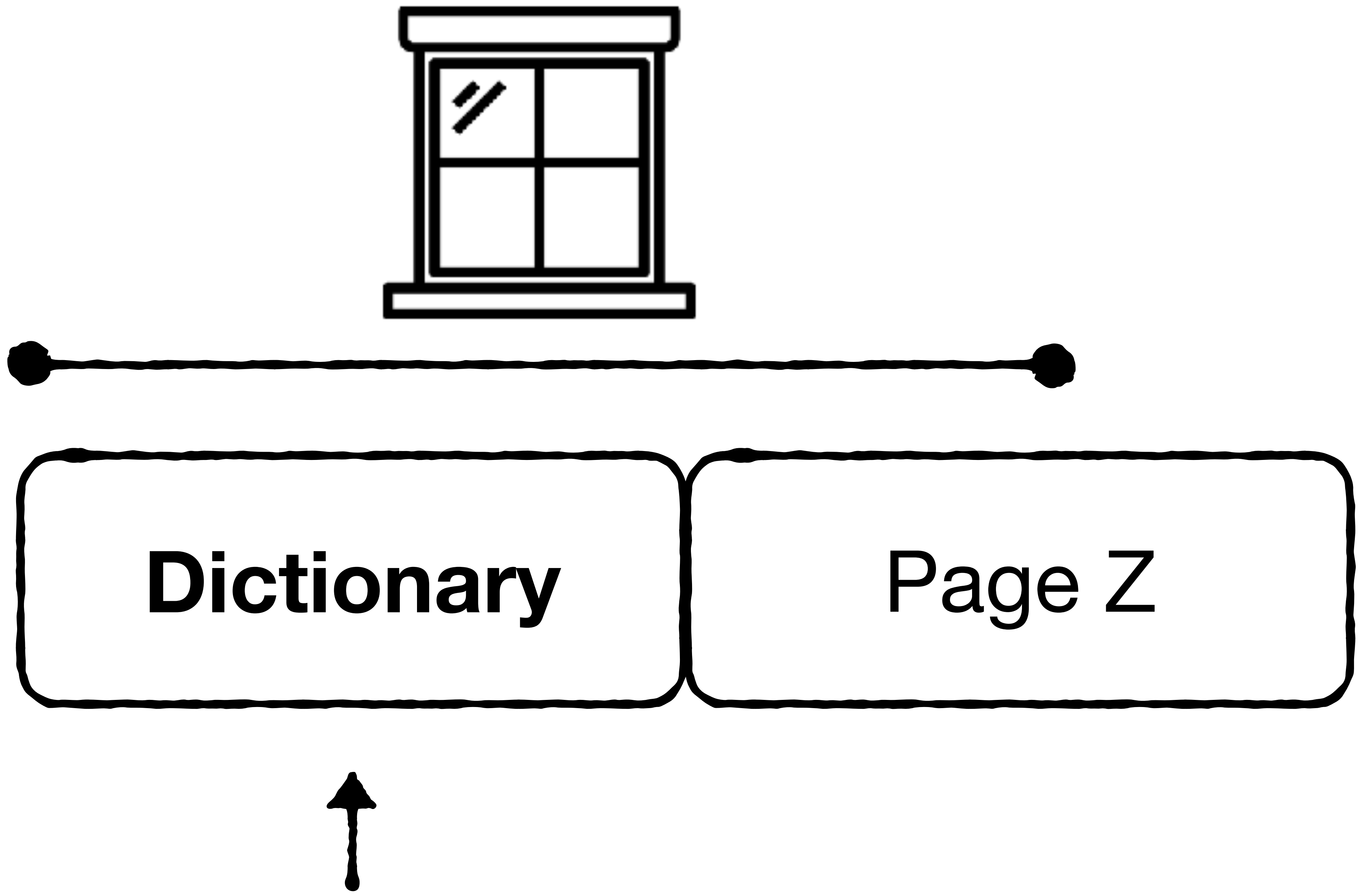
Capture redundancy in a dictionary



Compress each page using LZ77 with respect to dictionary



Compress each page using LZ77 with respect to dictionary

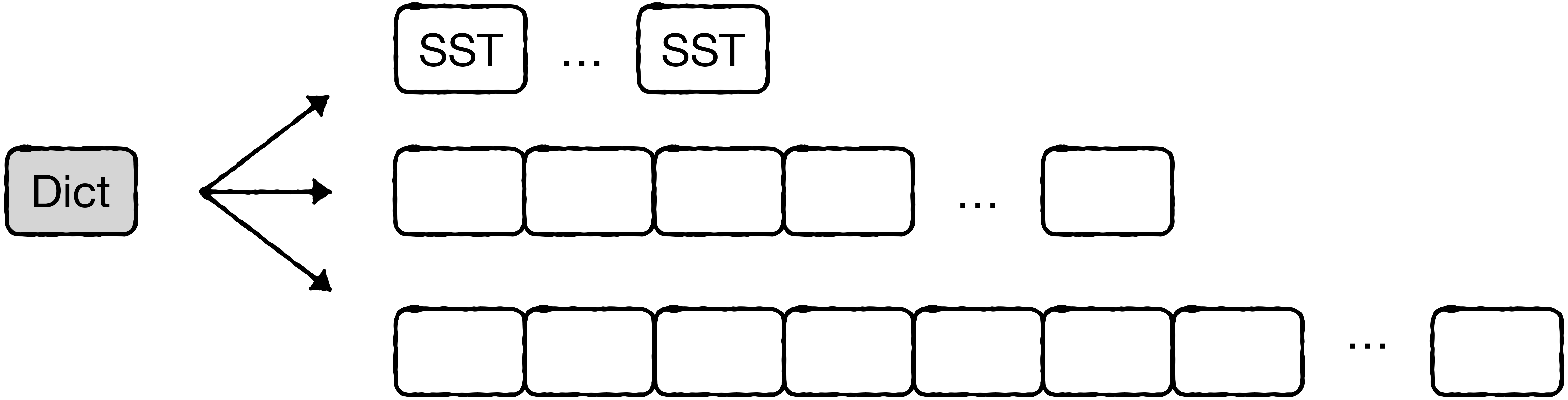


Needs to fit in the L1 or L2 caches

Dictionary Scopes

Dictionary Scopes

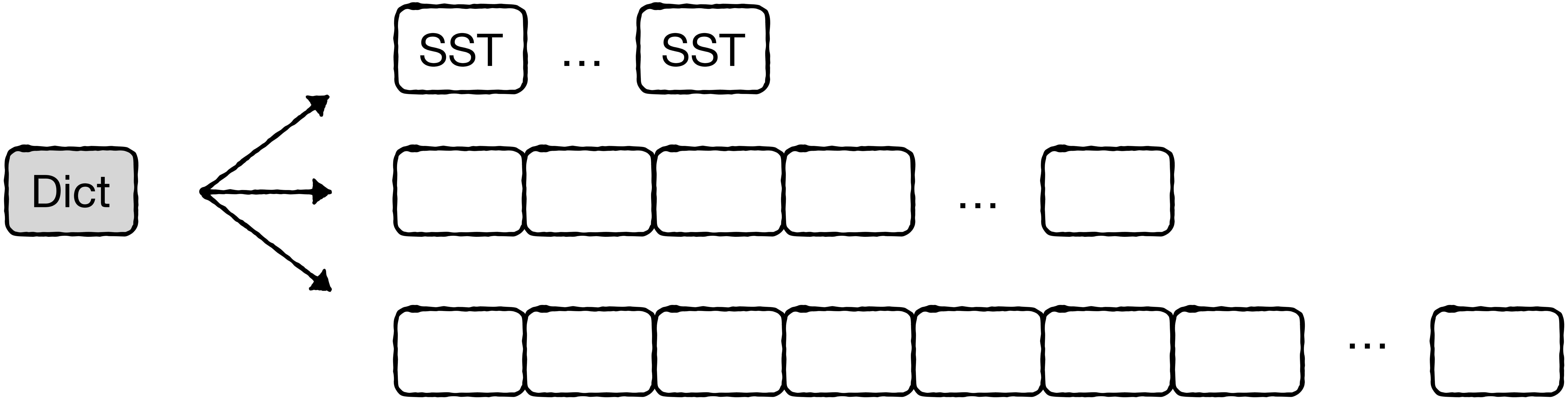
We can have one dictionary for the whole LSM-tree



Dictionary Scopes

We can have one dictionary for the whole LSM-tree

Problem: data distribution vary across/within levels and change over time

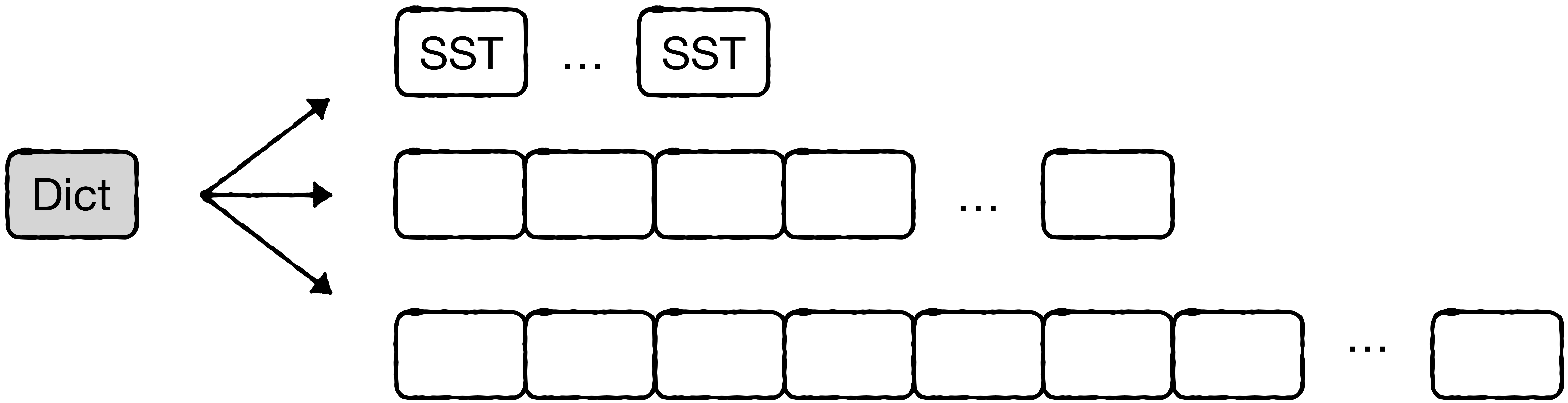


Dictionary Scopes

We can have one dictionary for the whole LSM-tree

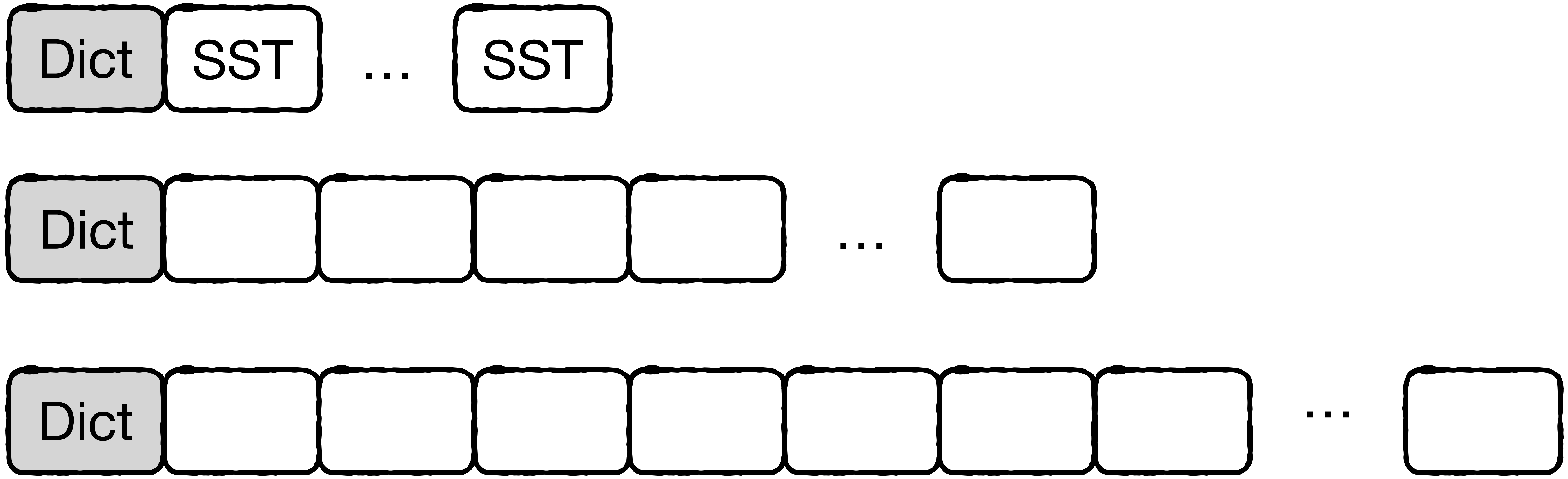
Problem: data distribution vary across/within levels and change over time

Updating the dictionary is problematic for decompression



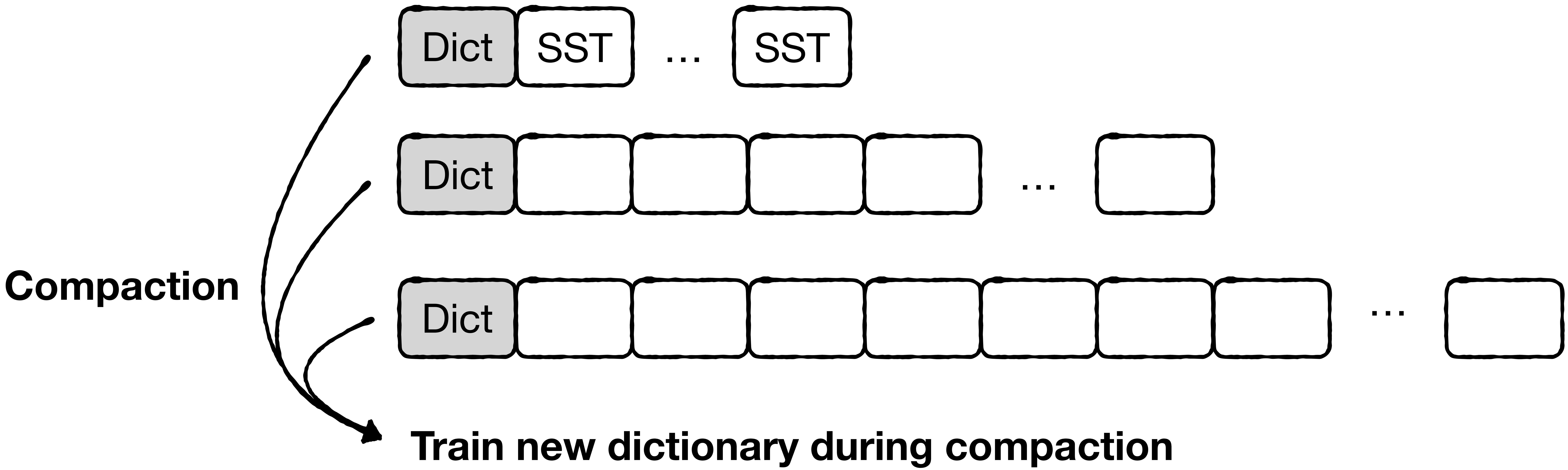
Dictionary Scopes

We can also have a dictionary per level

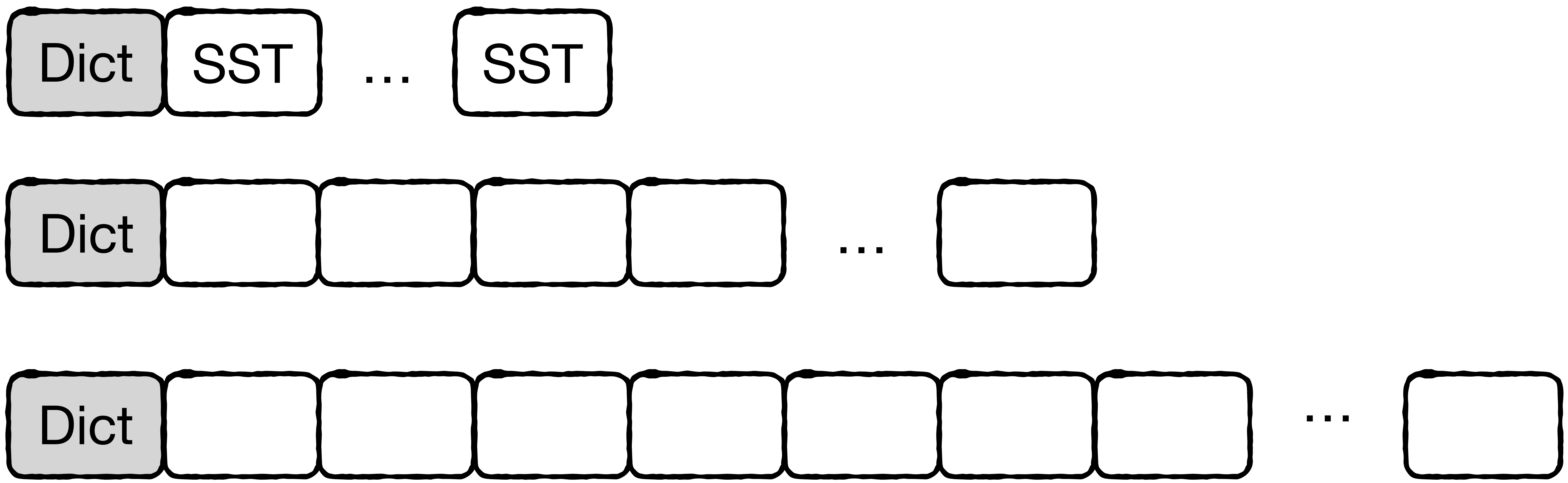


Dictionary Scopes

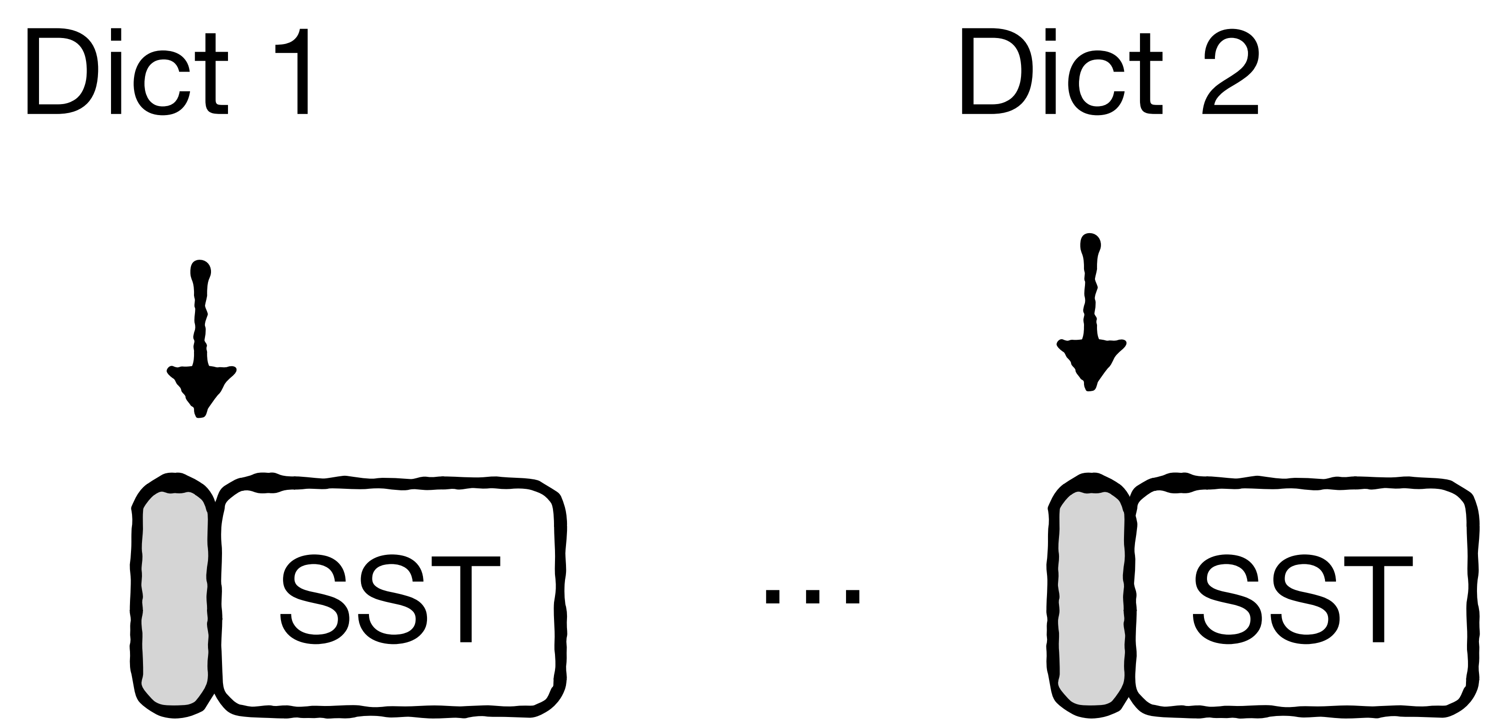
We can also have a dictionary per level



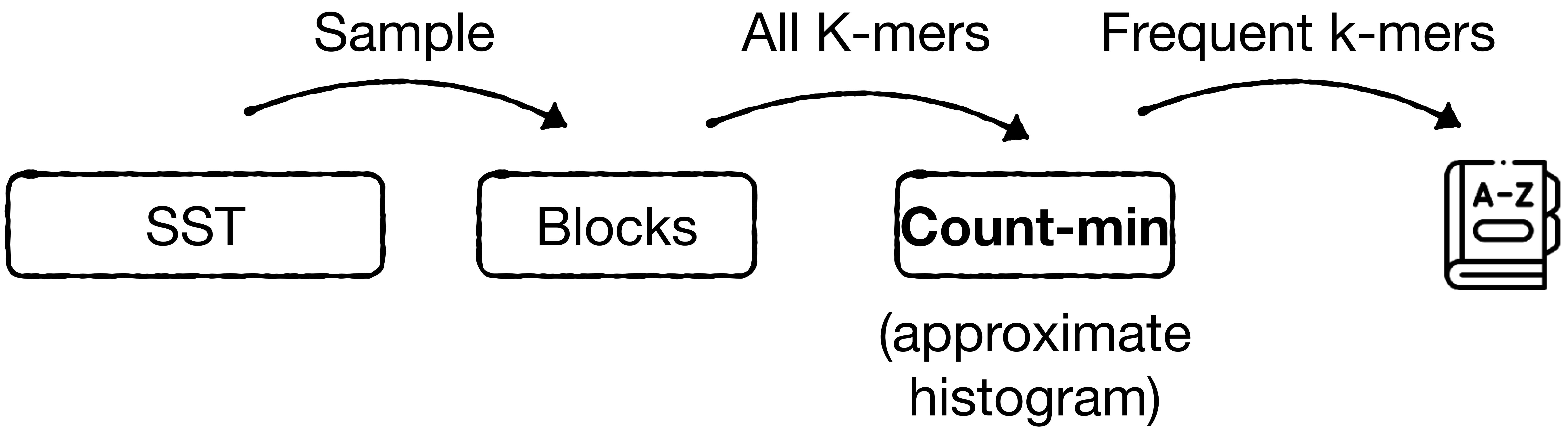
But the data distribution per level may also vary



RocksDB trains dictionary per SST, which is a spatial partition of a level

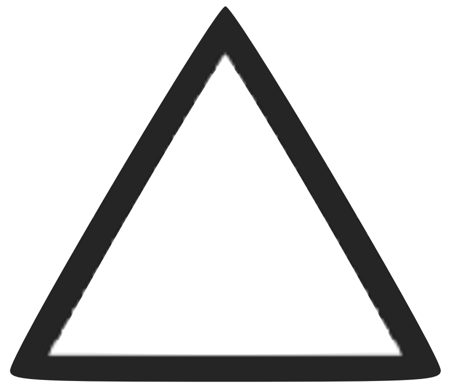


How is the dictionary trained?



Page
Sizing

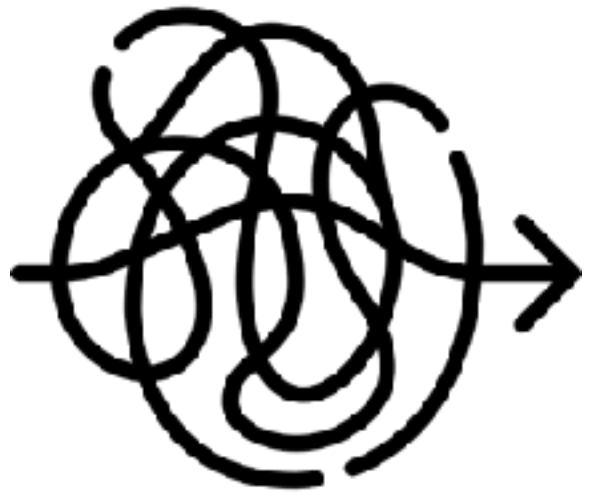
Delta
Encoding



Restarts

Page Hash
Index

LZ77



Huffman
coding

Dictionary
training

And now - a student presentation