Write-Optimized Data Structures

Mathis Chenuet, Noah Delius

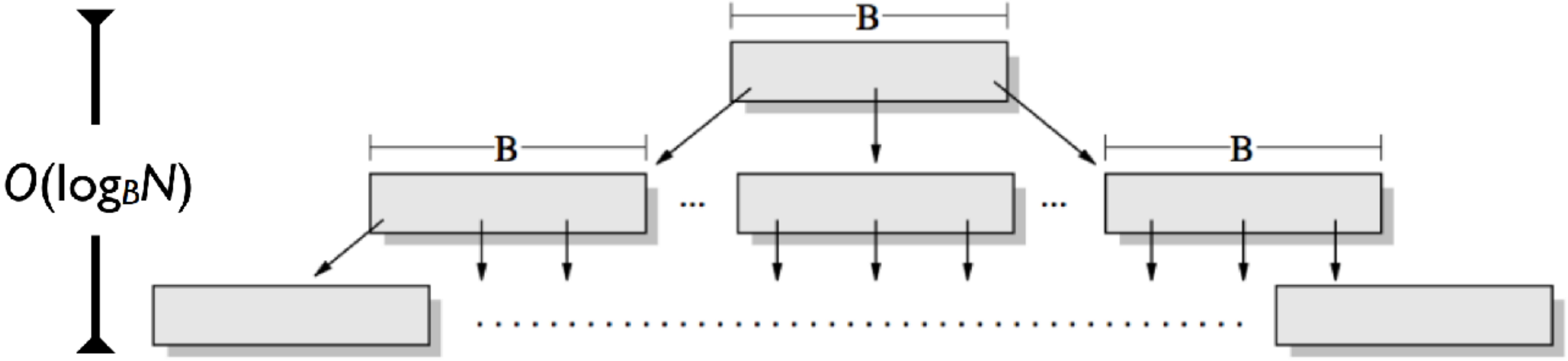
Write-Optimized Data Structures

Use cases: databases with more writes than reads;
optimize for fast writes and reasonably fast reads

- e.g. logs (with extra indices)

Idea: reducing random disk IO = minimizing number of block transfers

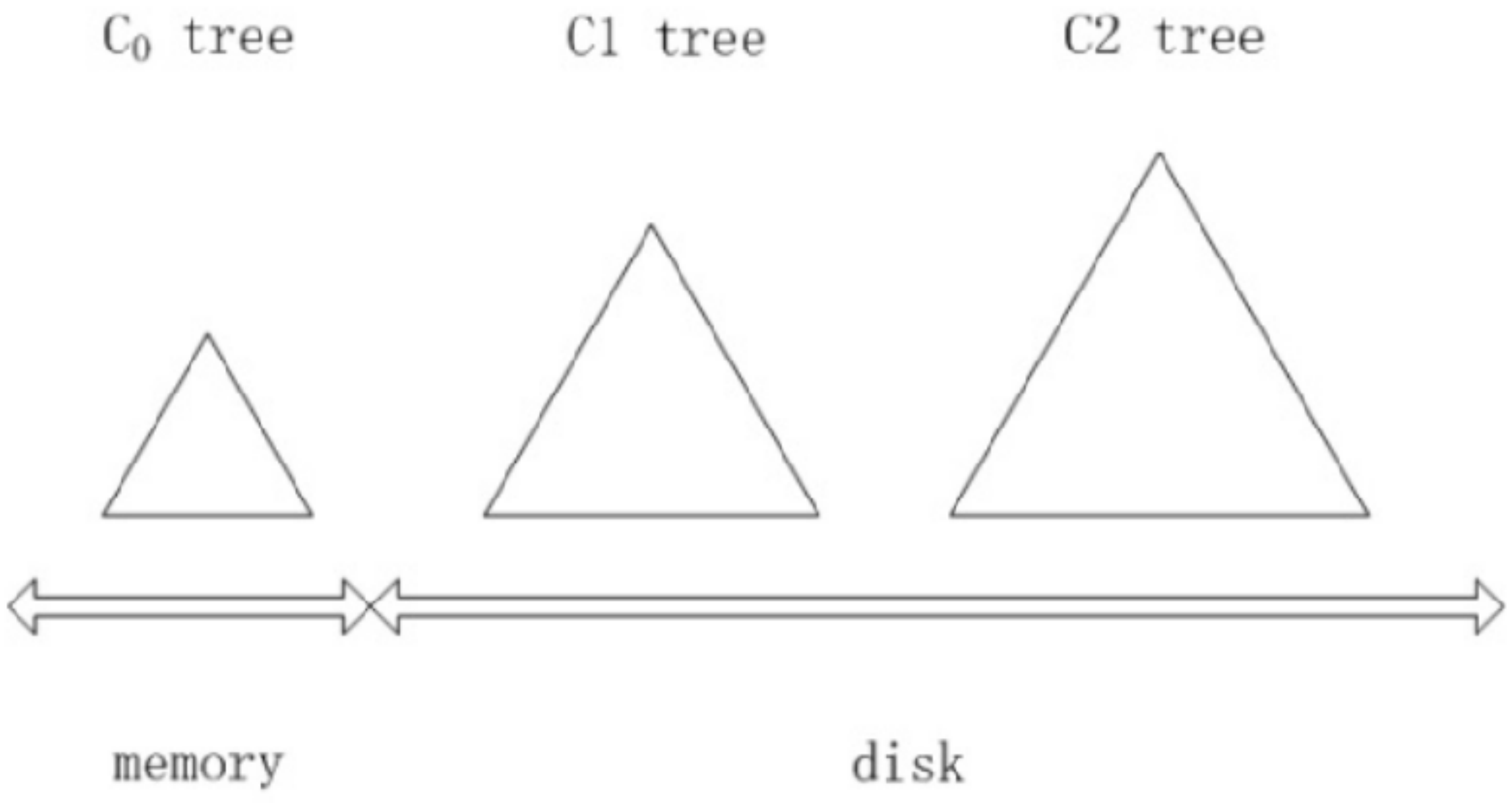
B-Trees



	B-tree	Some write-optimized structures
Insert/delete	$O(\log_B N) = O(\frac{\log N}{\log B})$	$O(\frac{\log N}{B})$

From Michael A. Bender, Stony Brook & Tokutek, “Write-Optimized Data Structures” talk, 2012

Log-Structured Merge Tree



Idea: Defer and batch operations

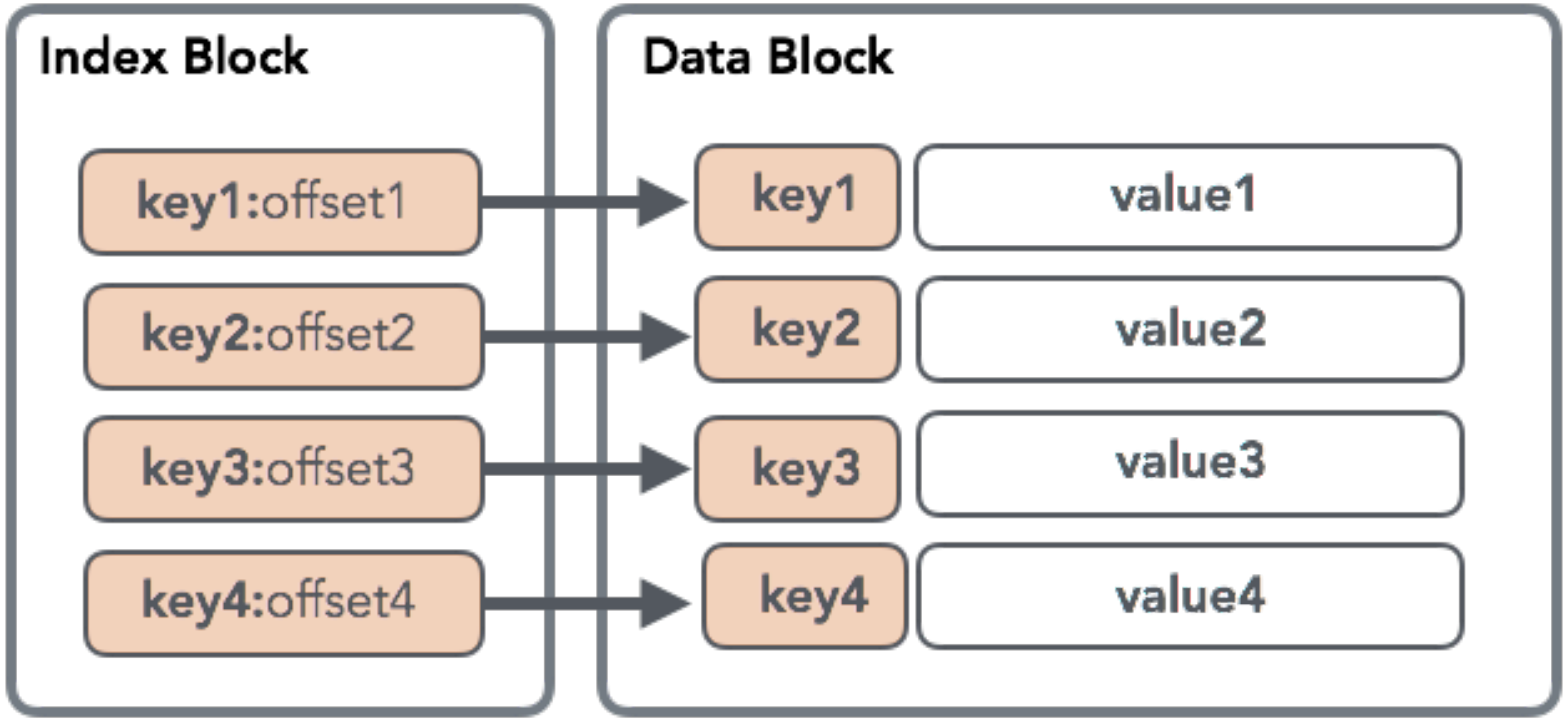
Multiple components: C_i

- C_0 is in memory
  Skiplist, B-tree, red-black tree

- $C_{i>0}$ consists of Sorted String Tables (LevelDB), B-trees (original paper), …

SSTable

Concept: sorted key-value pairs in an immutable array



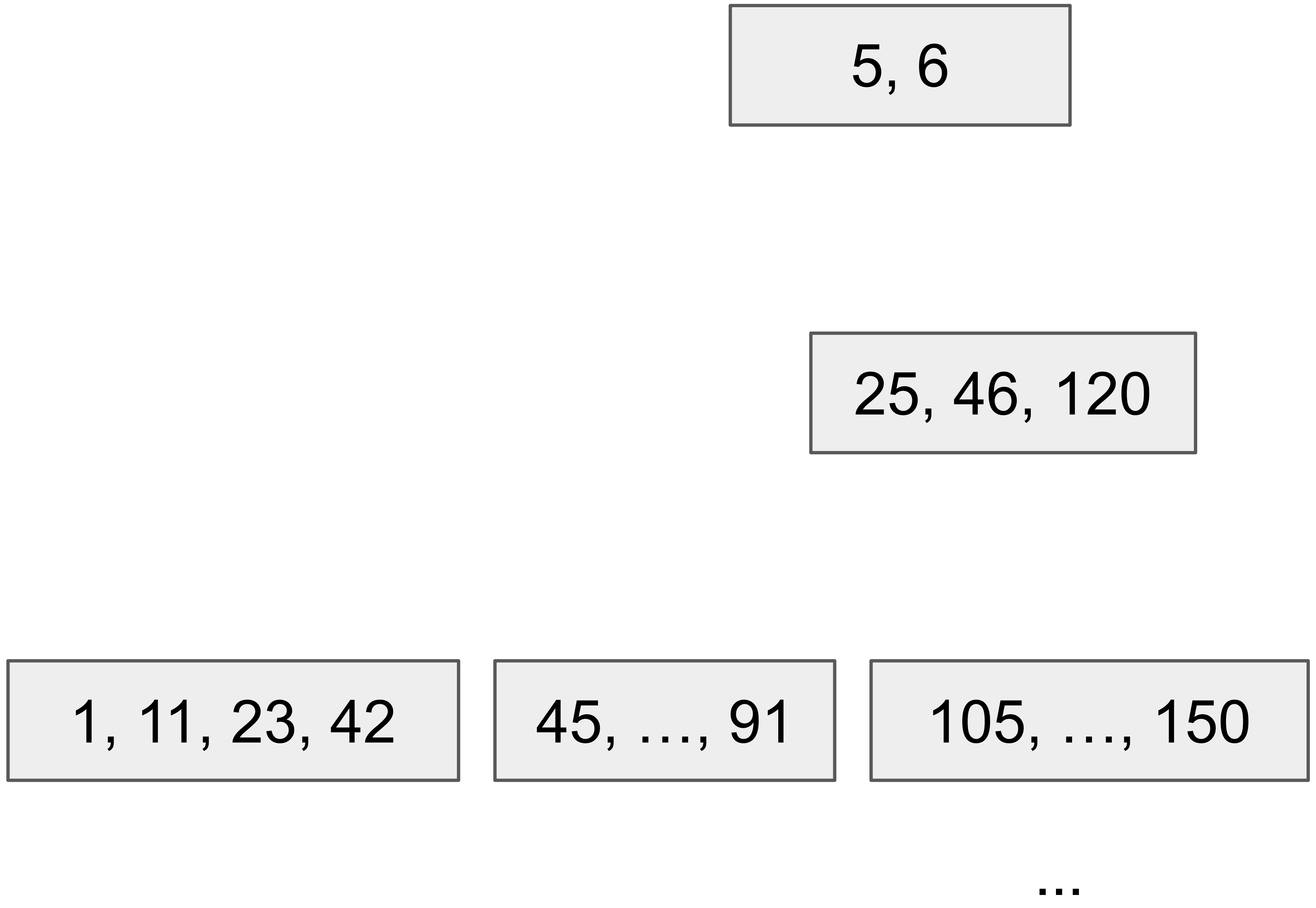
From Alex Petrov, “On Disk IO, Part 3: LSM Trees”, 2017

On disk: contiguous array

Log-Structured Merge Tree

LevelDB strategy

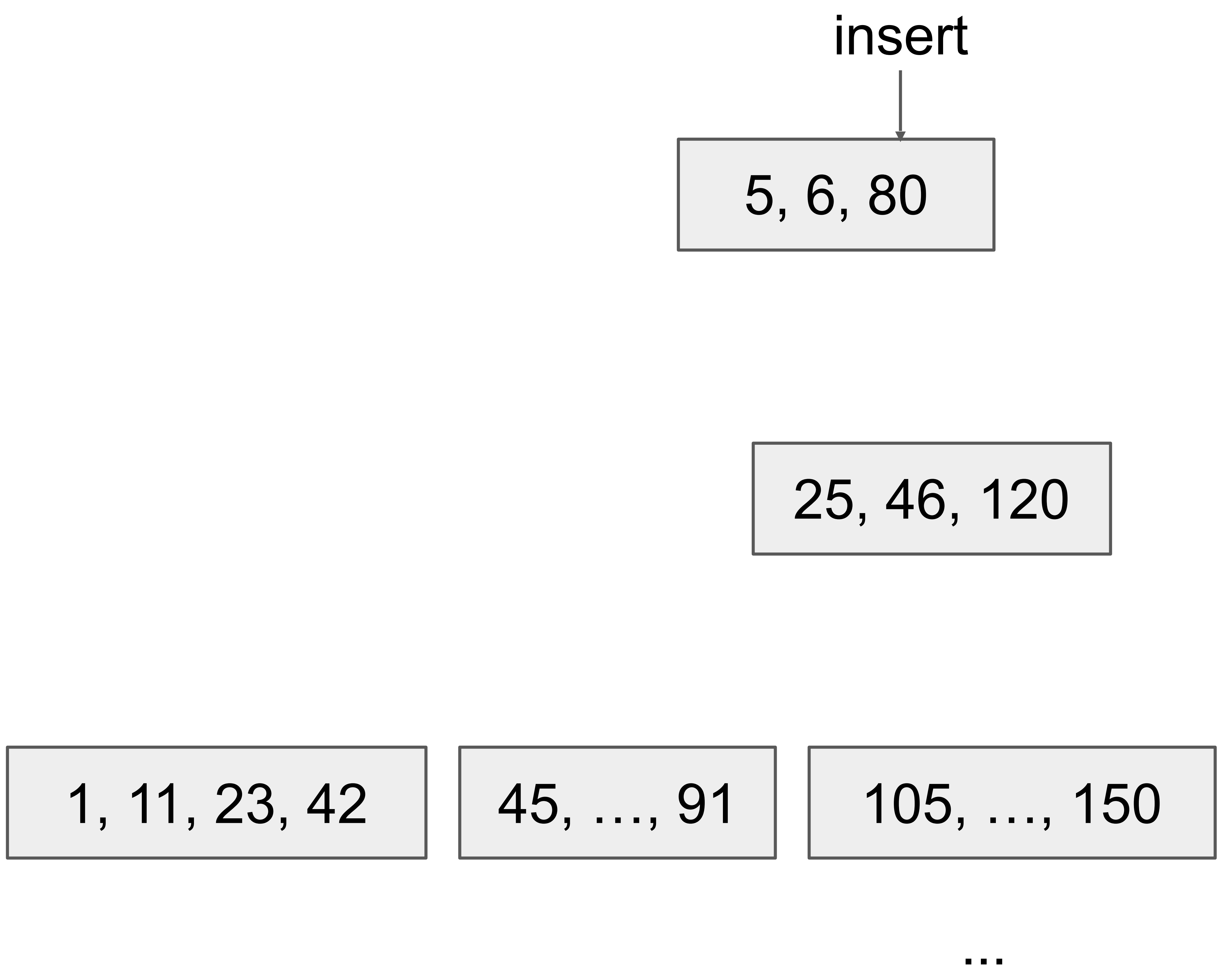
- C_0: skiplist
- C_1: SSTables with overlapping key ranges
- C_{2+}: SSTables with disjoint key ranges



Log-Structured Merge Tree

LevelDB strategy

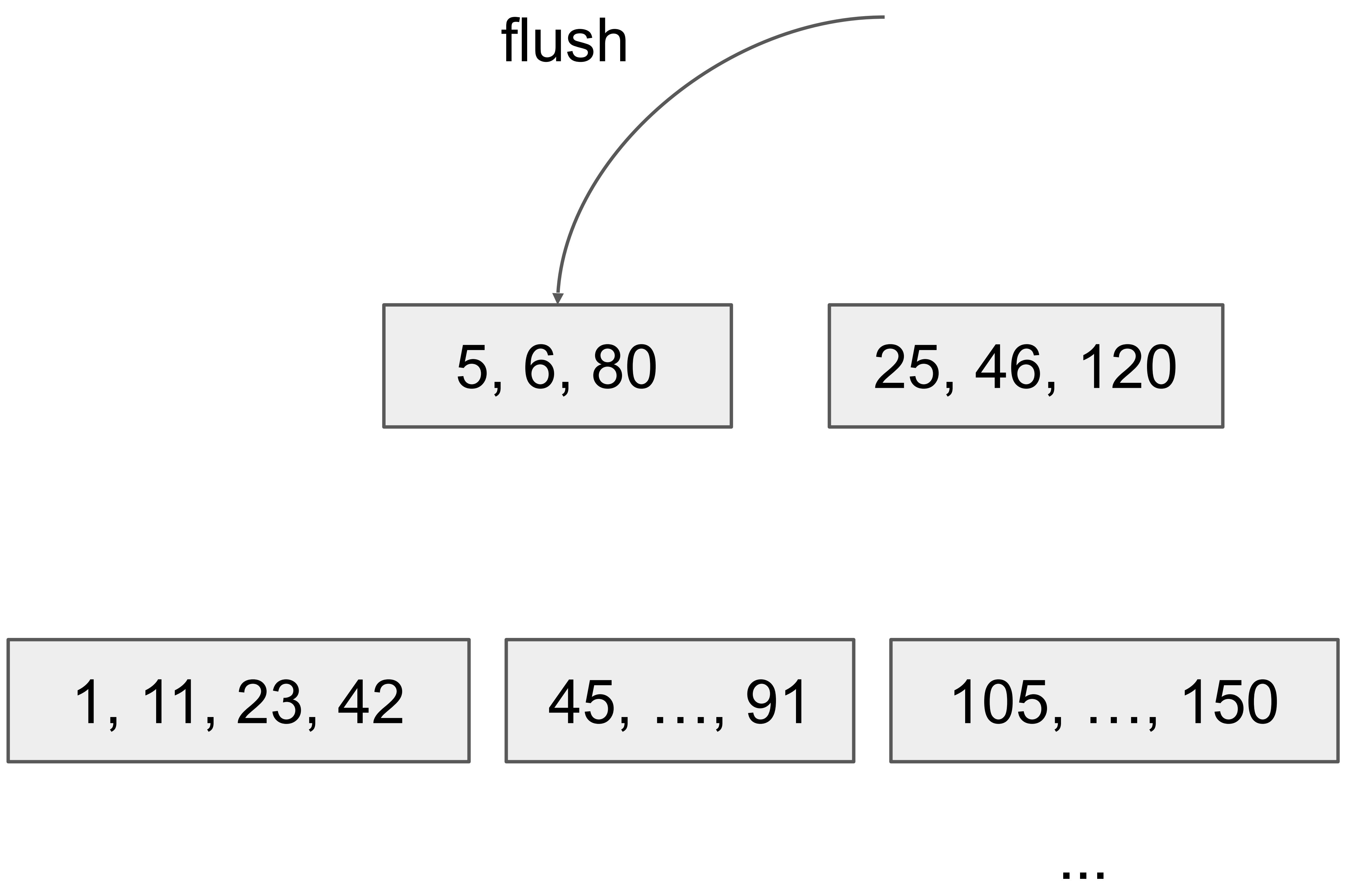
- C_0: skiplist
- C_1: SSTables with overlapping key ranges
- C_{2+}: SSTables with disjoint key ranges



Log-Structured Merge Tree

LevelDB strategy

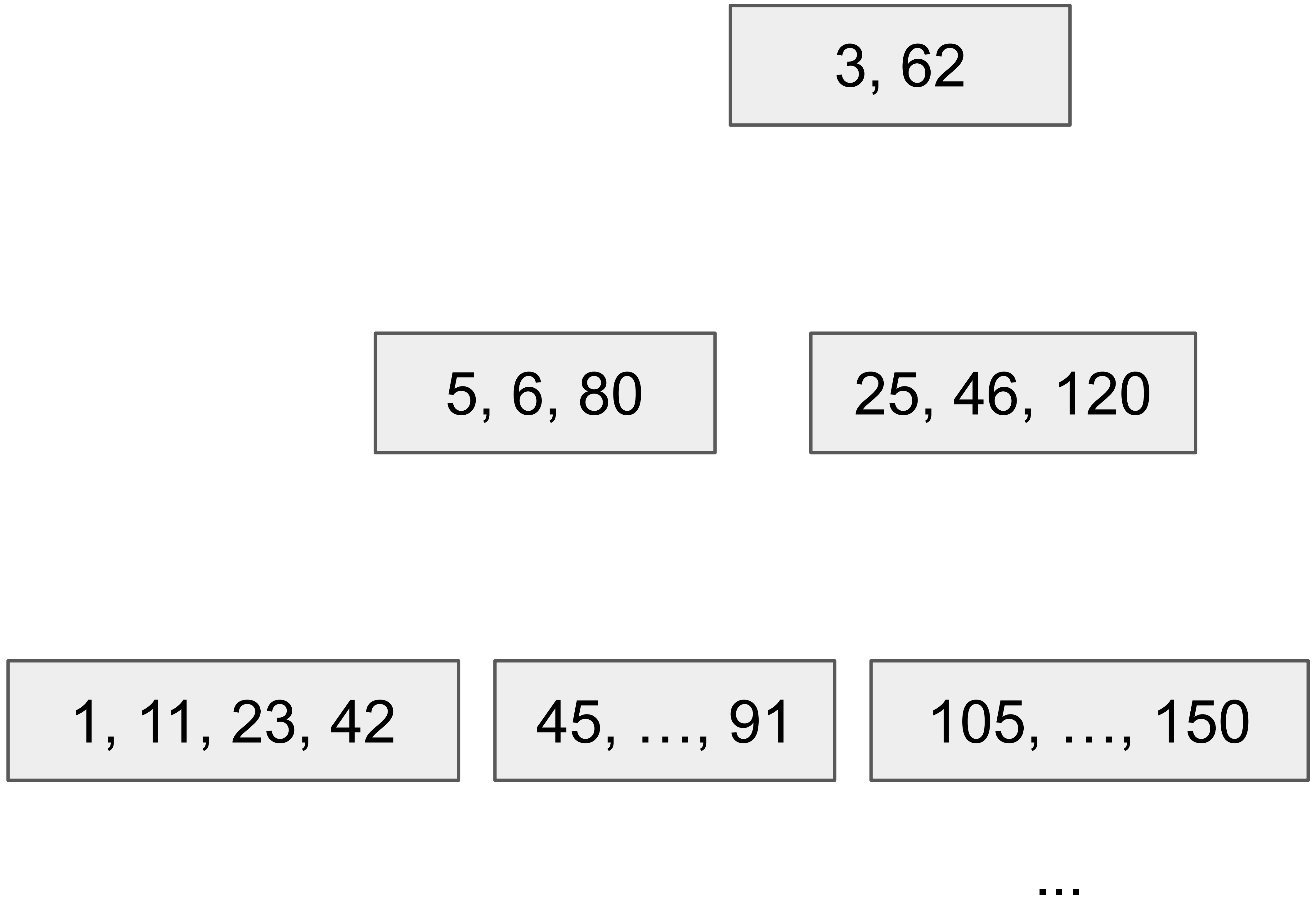
- C_0: skiplist
- C_1: SSTables with overlapping key ranges
- C_{2+}: SSTables with disjoint key ranges



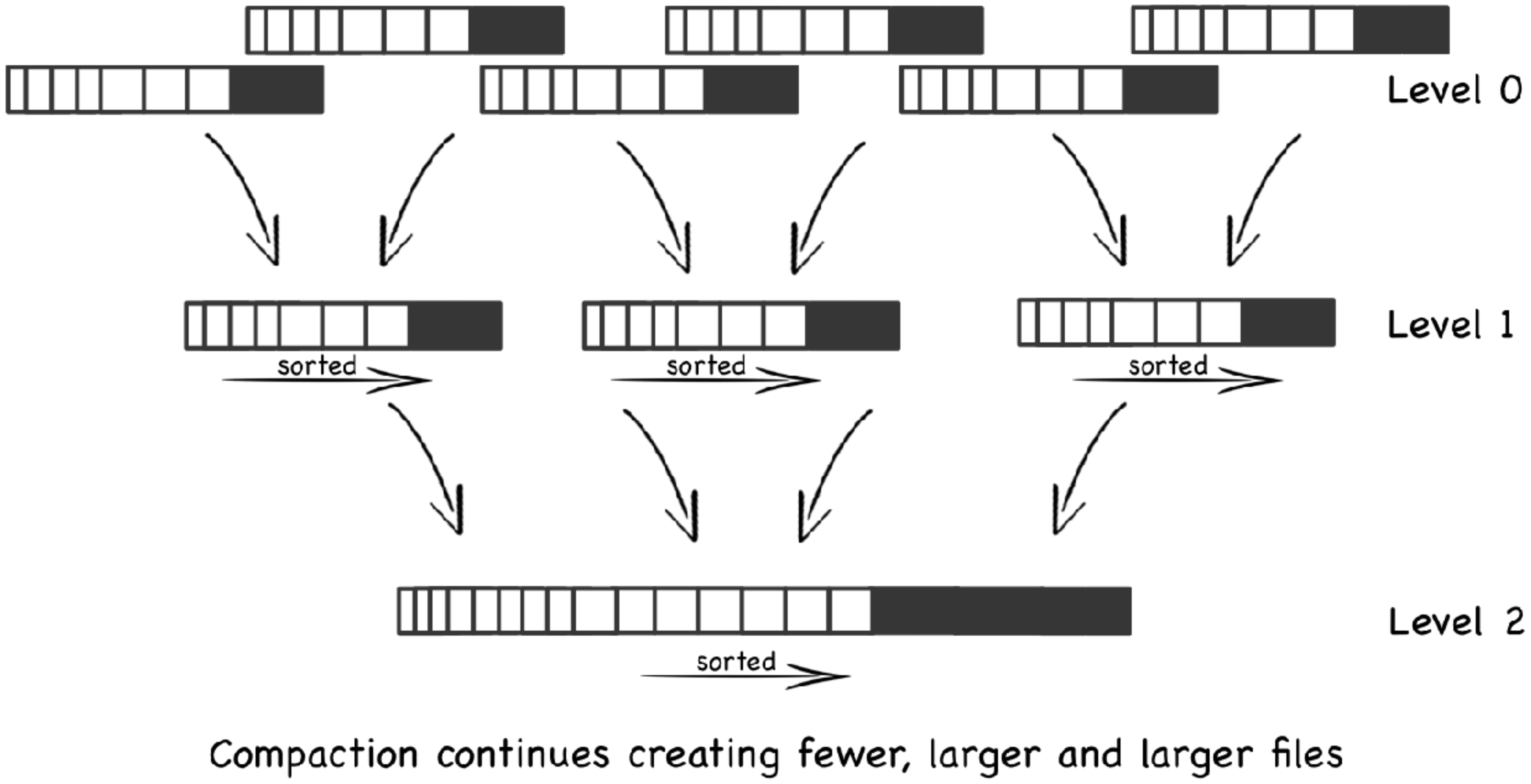
Log-Structured Merge Tree

LevelDB strategy

- C_0: skiplist
- C_1: SSTables with overlapping key ranges
- C_{2+}: SSTables with disjoint key ranges



LSM Tree merge behavior



From Ben Stopford, "Log Structured Merge Trees", 2015

Cache-**O**blivious **L**ookahead **A**rrays

Basic
Cache-Oblivious
Lookahead Arrays

Basic
Cache-**O**blivious
~~**L**ookahead~~ **A**rrays

Basic COLA

Basic COLA

- for N key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ **sorted arrays (*levels*)**

Basic COLA

- for N key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ **sorted arrays (*levels*)**
- for $0 \le i \le \lfloor \log_2 N \rfloor$, level i is empty or contains 2^i elements

Basic COLA

- for N key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ **sorted arrays (*levels*)**
- for $0 \leq i \leq \lfloor \log_2 N \rfloor$, level i is empty or contains 2^i elements
- level i full $\iff i^{th}$ least significant bit of N is 1

Basic COLA

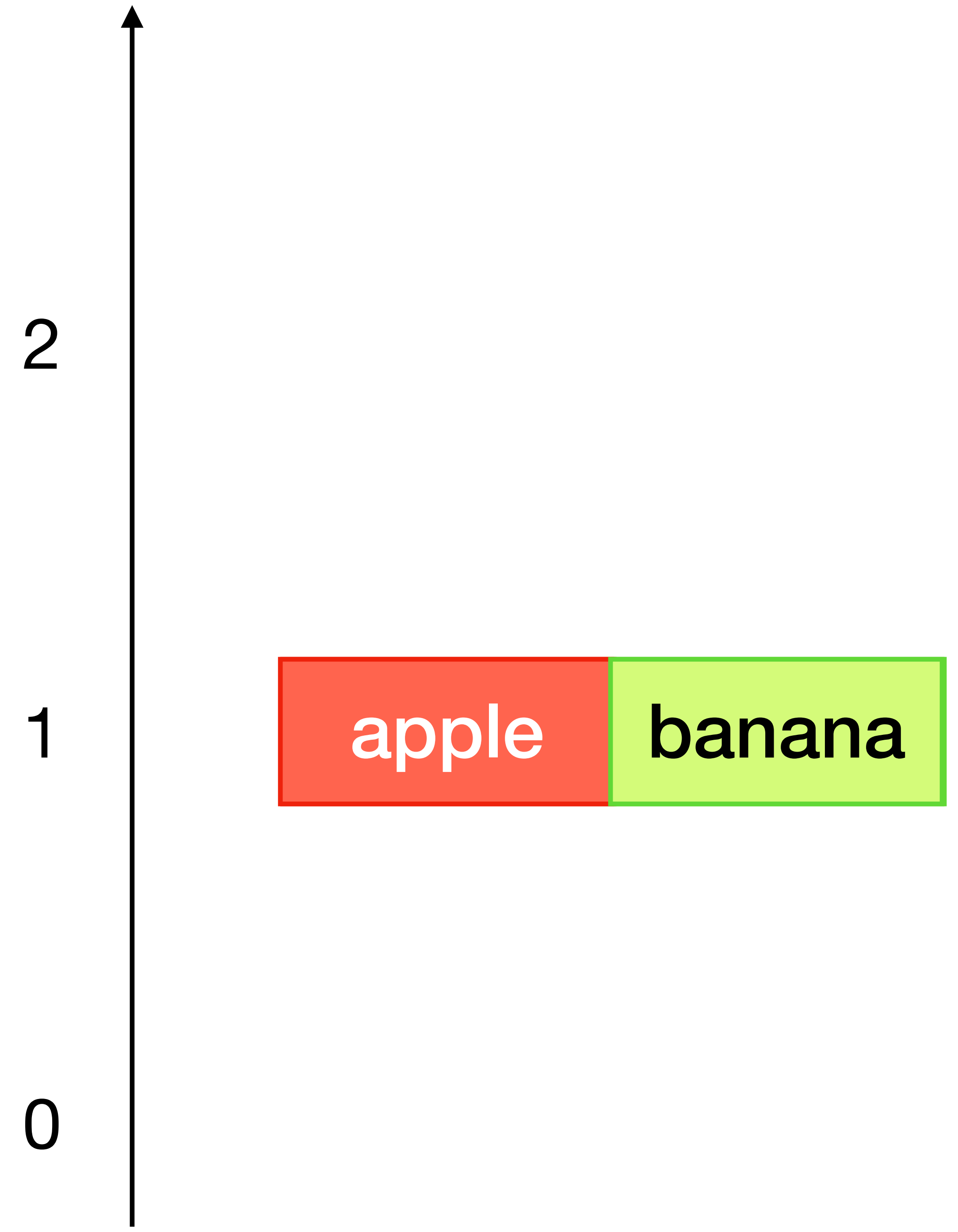
- for N key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ **sorted arrays (*levels*)**
- for $0 \leq i \leq \lfloor \log_2 N \rfloor$, level i is empty or contains 2^i elements
- level i full $\iff i^{th}$ least significant bit of N is 1
- insert: find smallest empty array, merge all levels below (and the new element) into it, clear lower levels

Basic COLA

- for N key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ **sorted arrays (*levels*)**
- for $0 \leq i \leq \lfloor \log_2 N \rfloor$, level i is empty or contains 2^i elements
- level i full $\iff i^{th}$ least significant bit of N is 1
- insert: find smallest empty array, merge all levels below (and the new element) into it, clear lower levels
- query: perform binary search on all levels

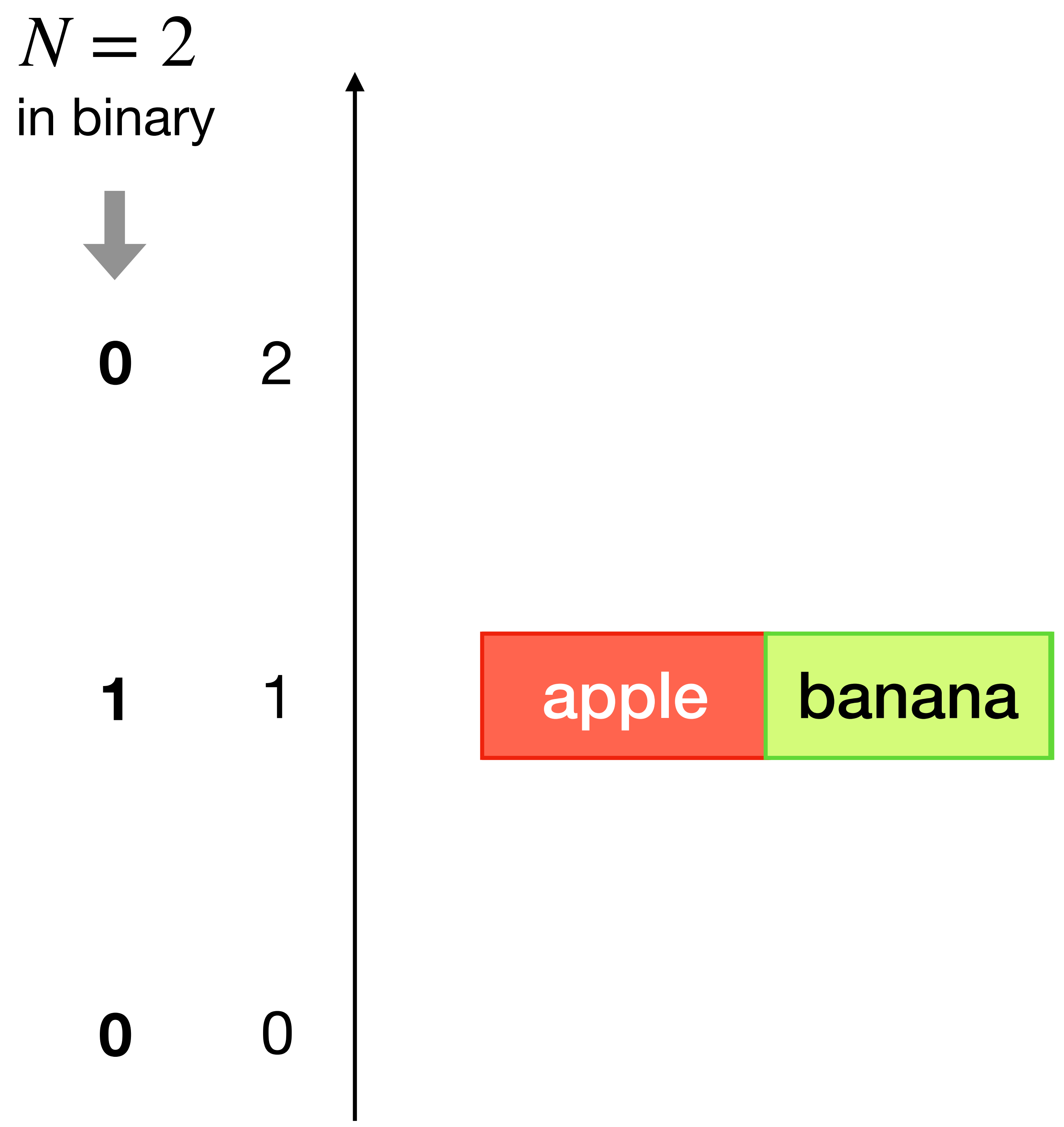
Basic COLA — Insertions

example: keys are names, values are colors



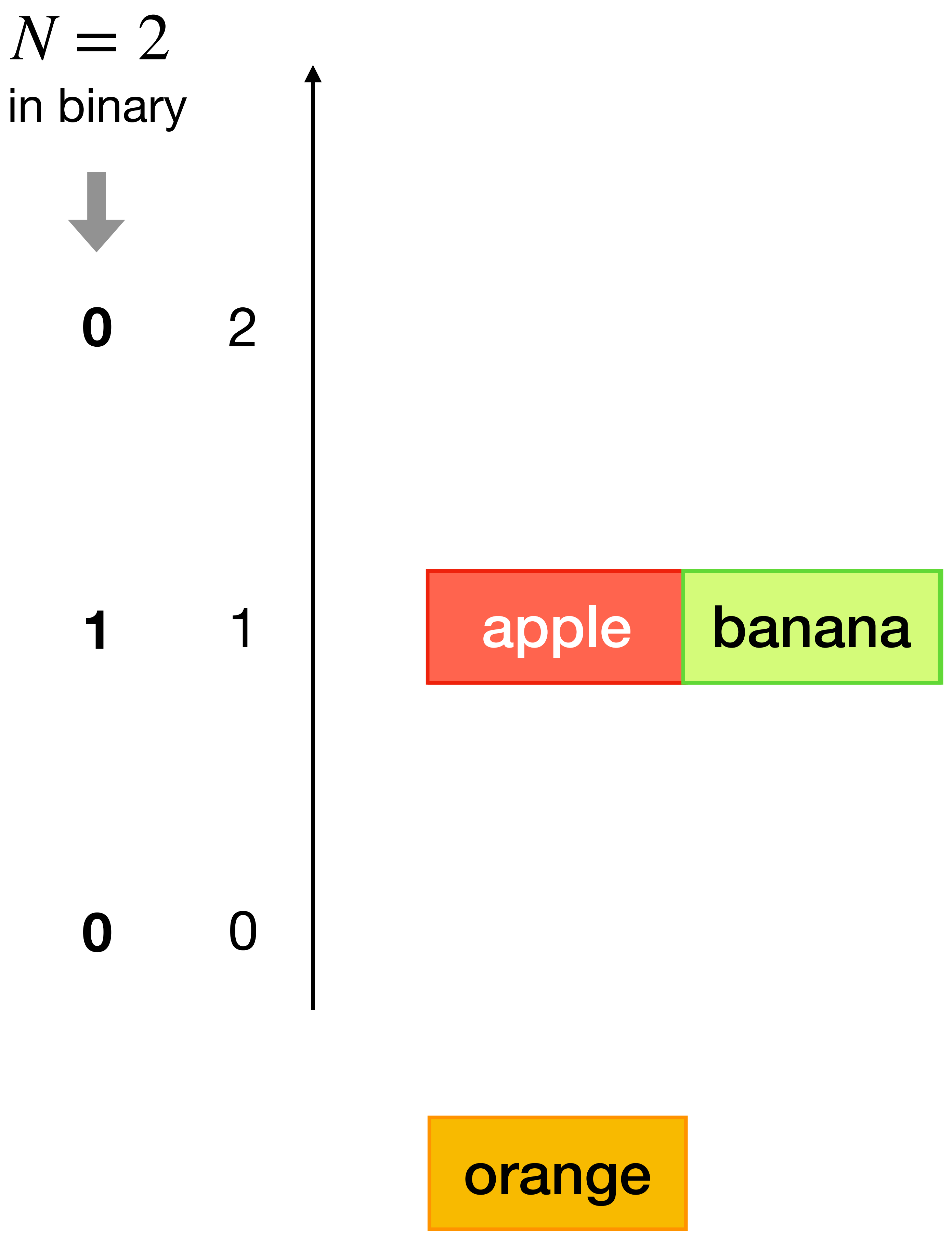
Basic COLA — Insertions

example: keys are names, values are colors



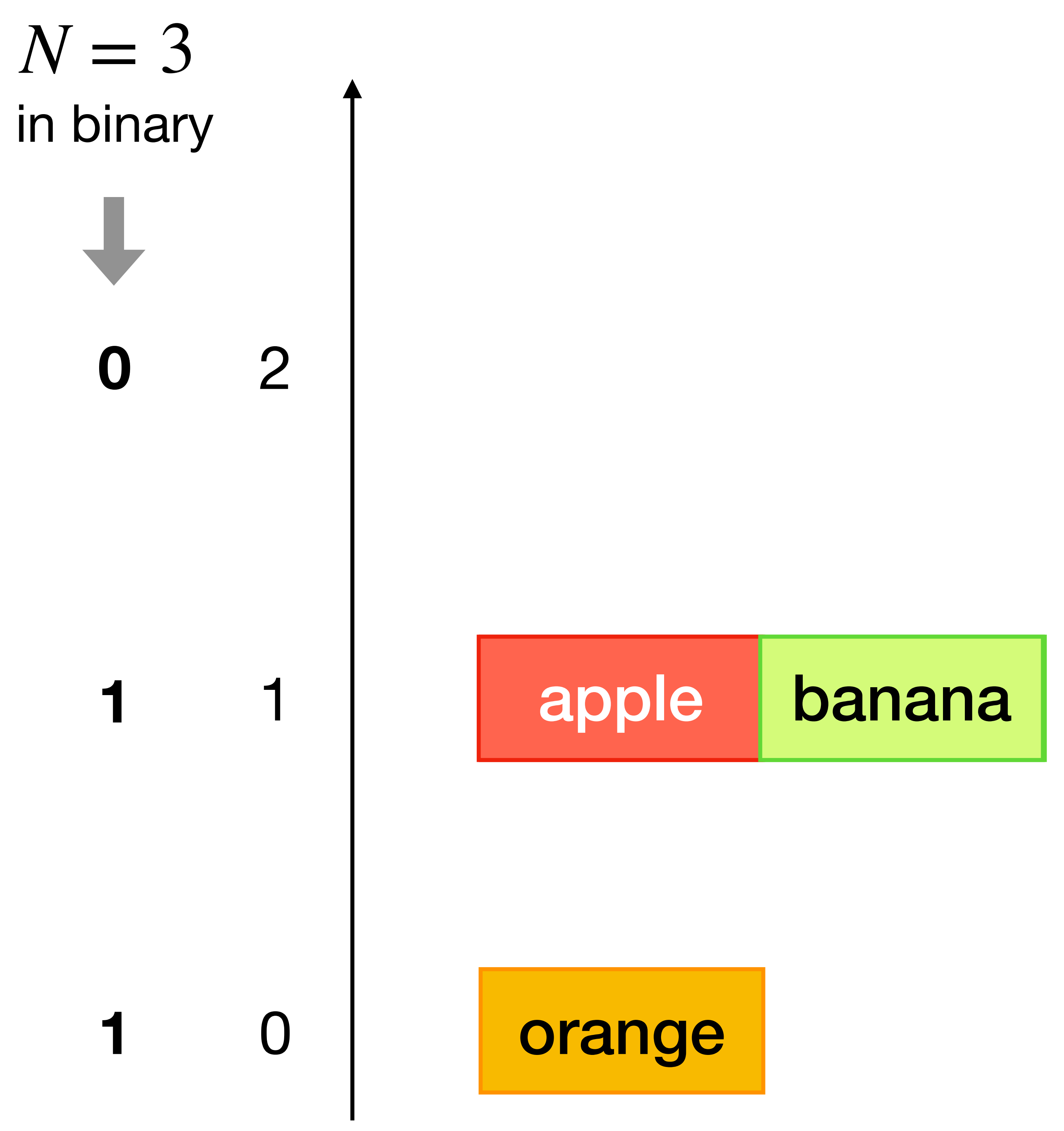
Basic COLA — Insertions

example: keys are names, values are colors



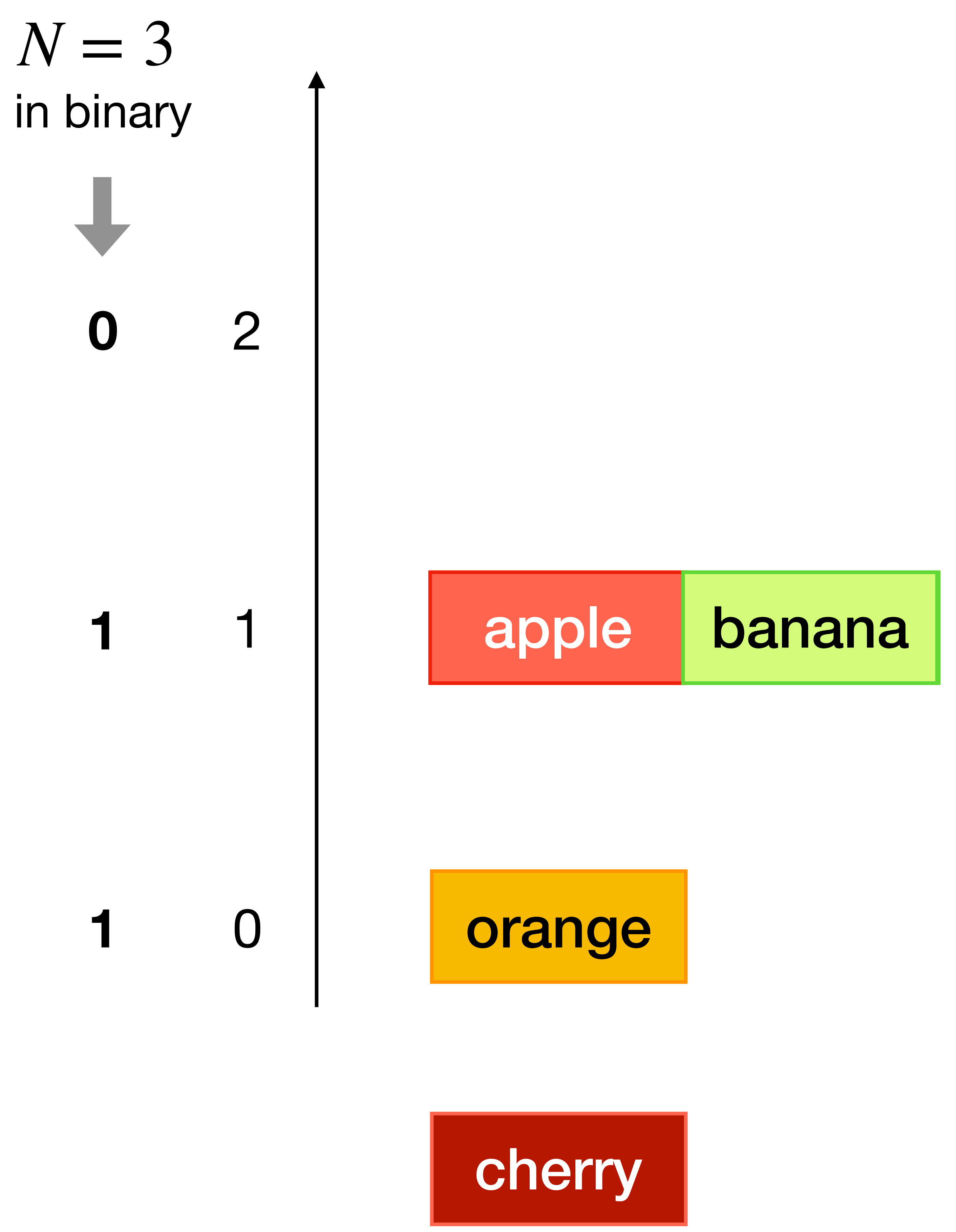
Basic COLA — Insertions

example: keys are names, values are colors



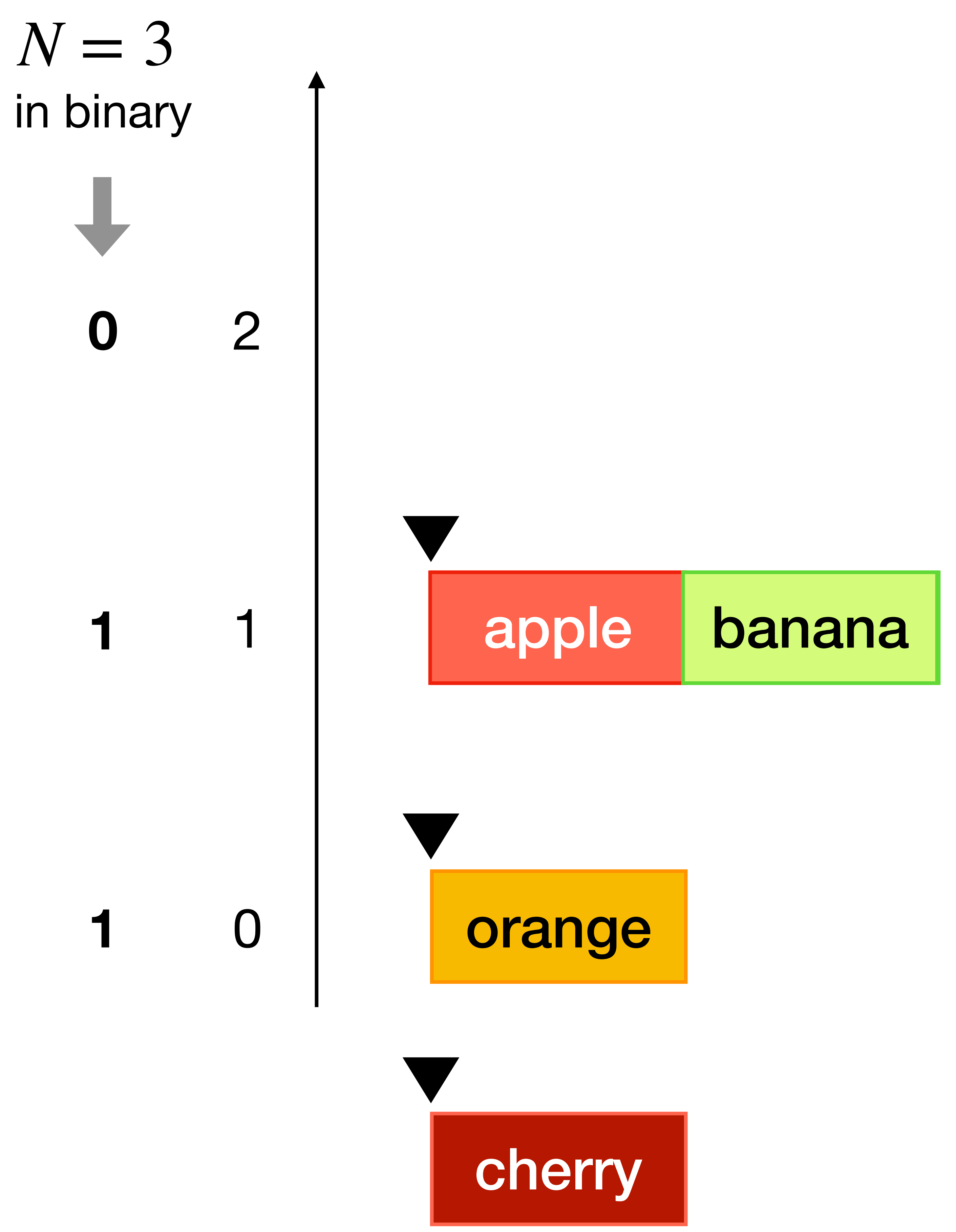
Basic COLA — Insertions

example: keys are names, values are colors



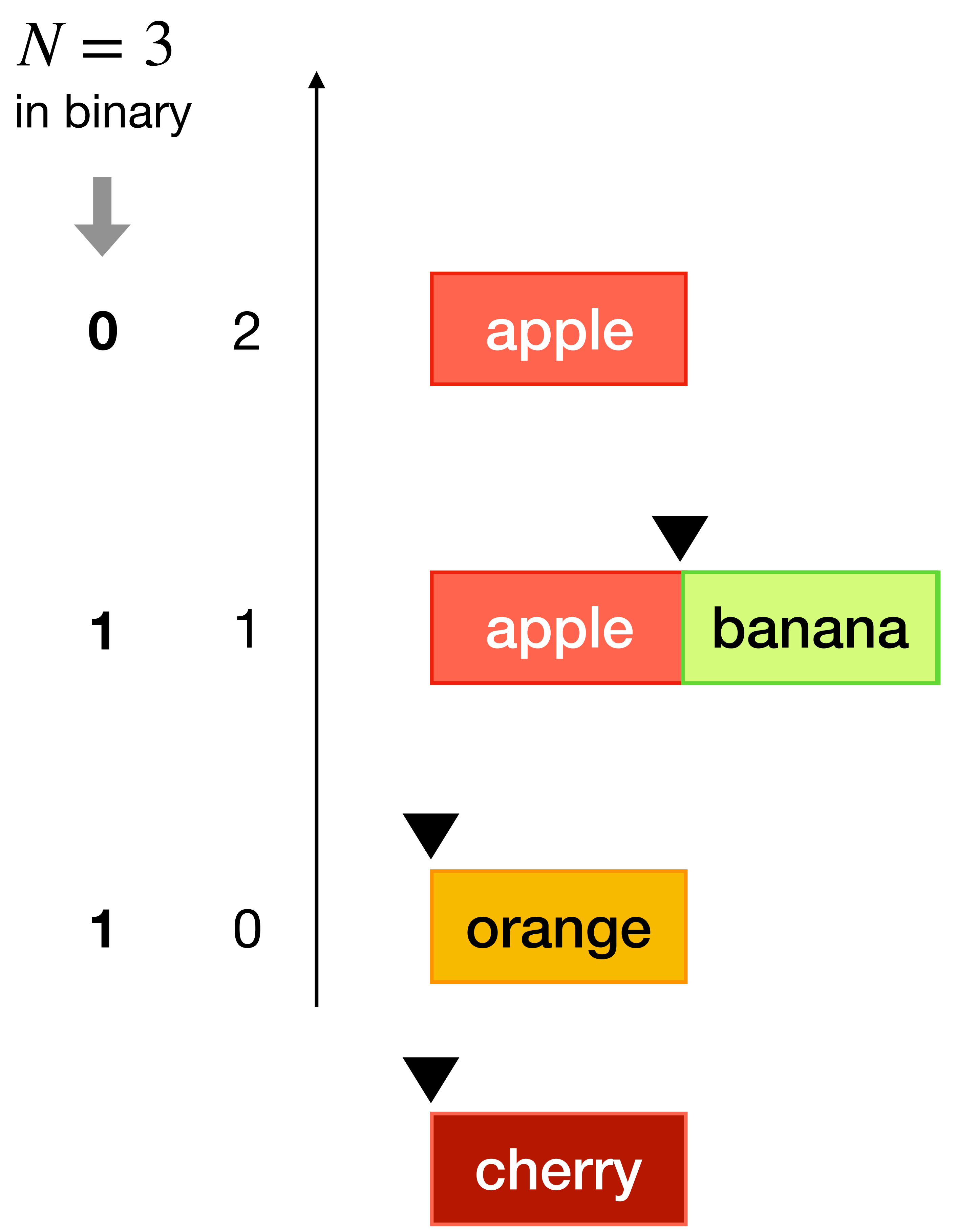
Basic COLA — Insertions

example: keys are names, values are colors



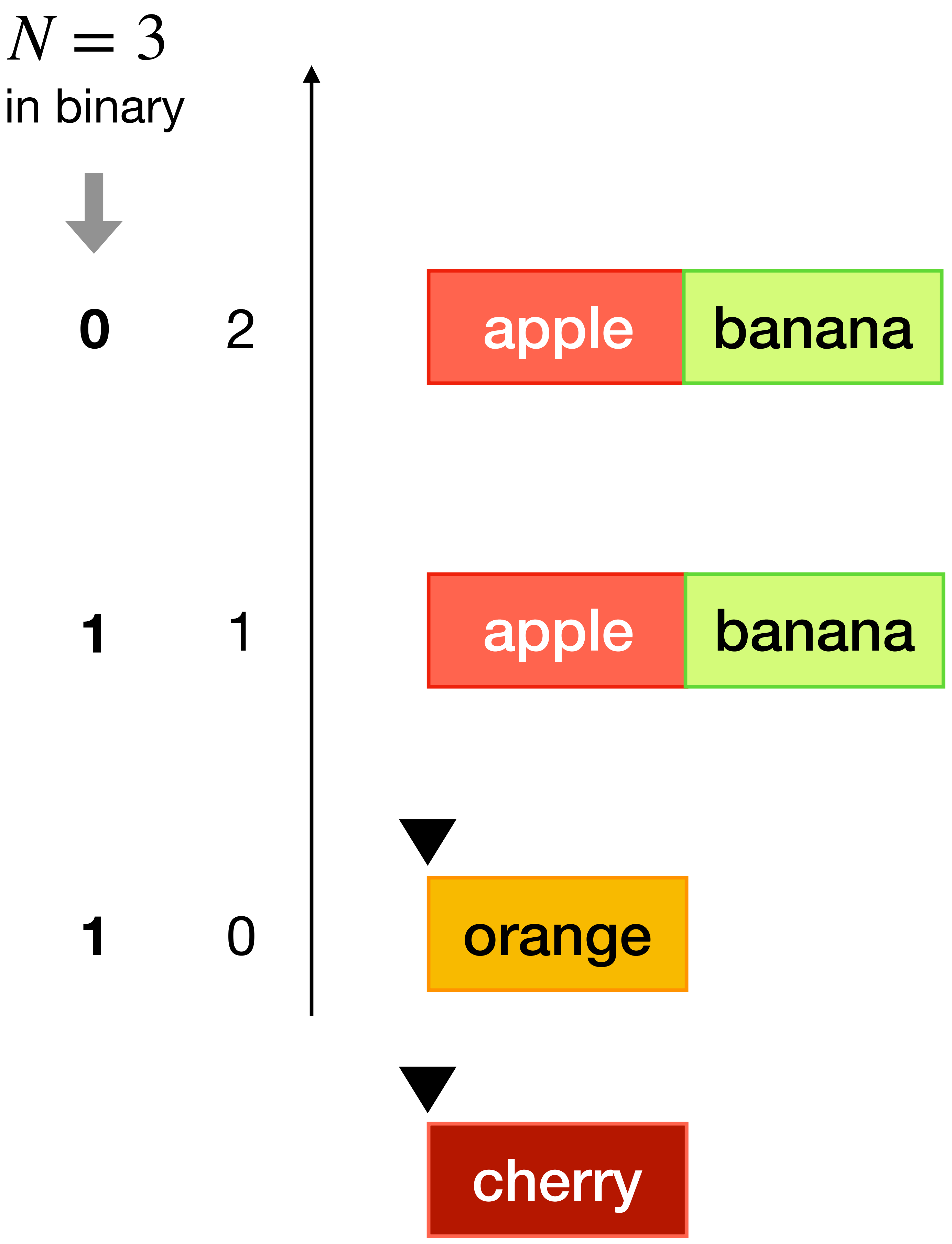
Basic COLA — Insertions

example: keys are names, values are colors



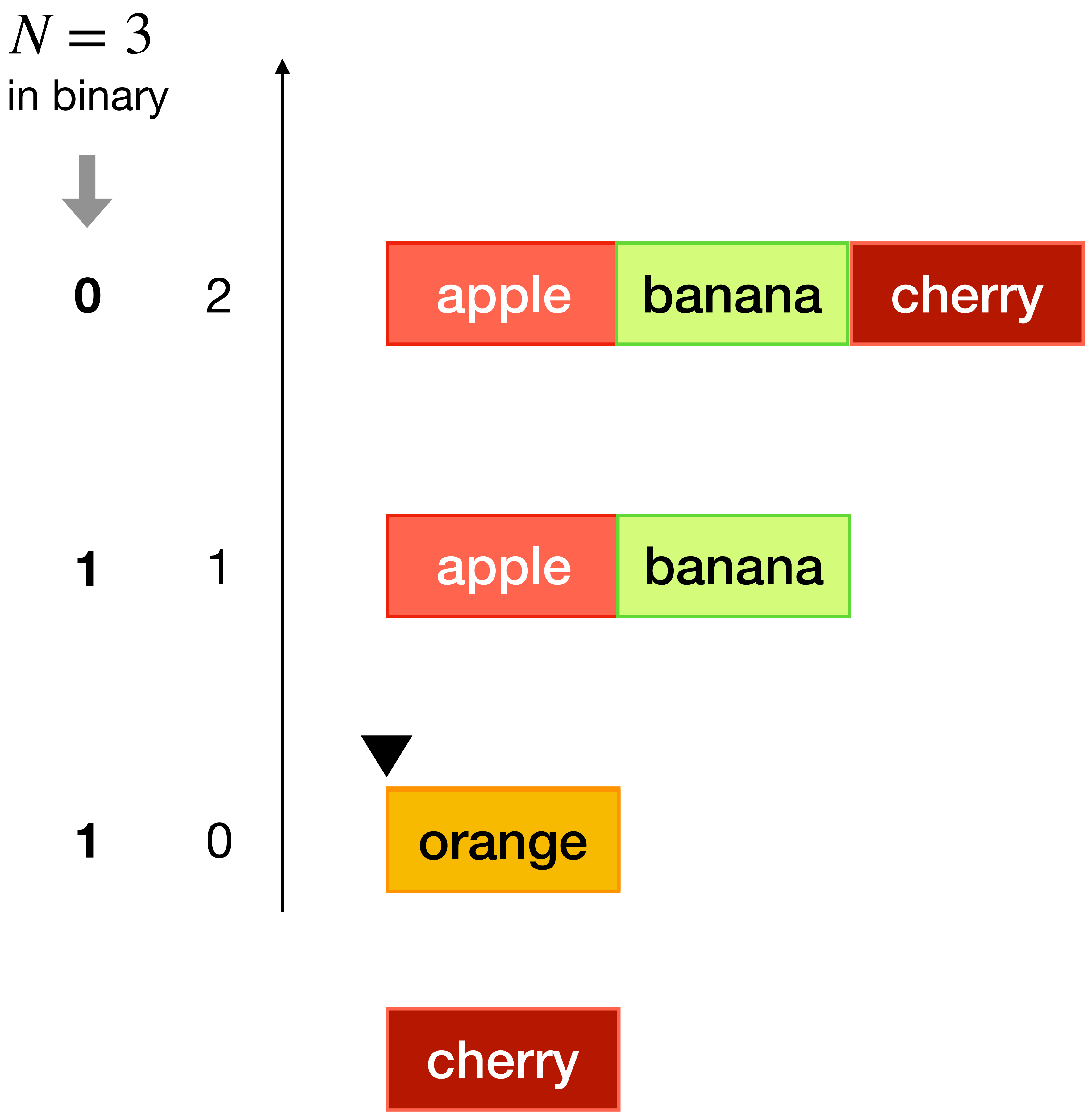
Basic COLA — Insertions

example: keys are names, values are colors



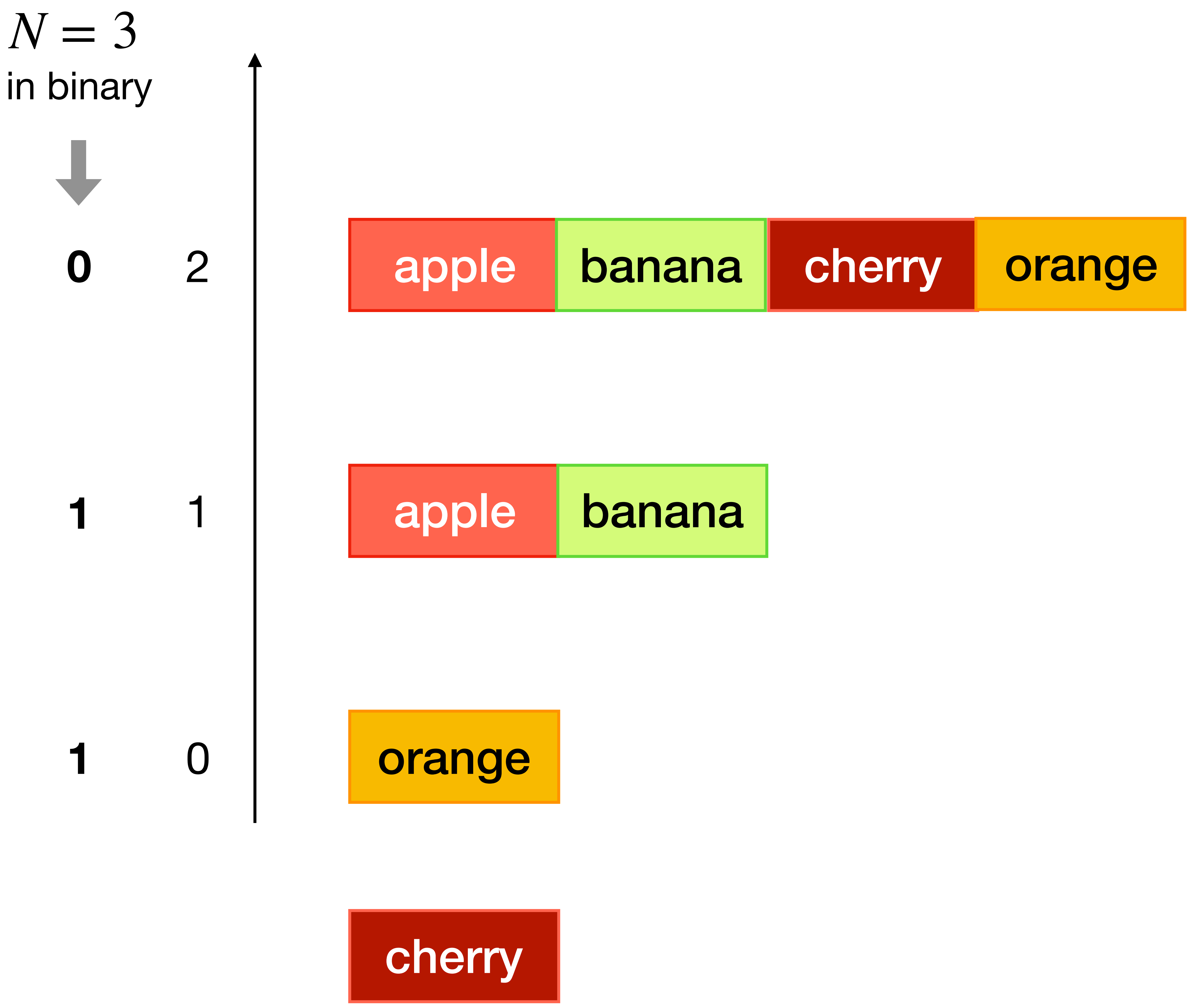
Basic COLA — Insertions

example: keys are names, values are colors



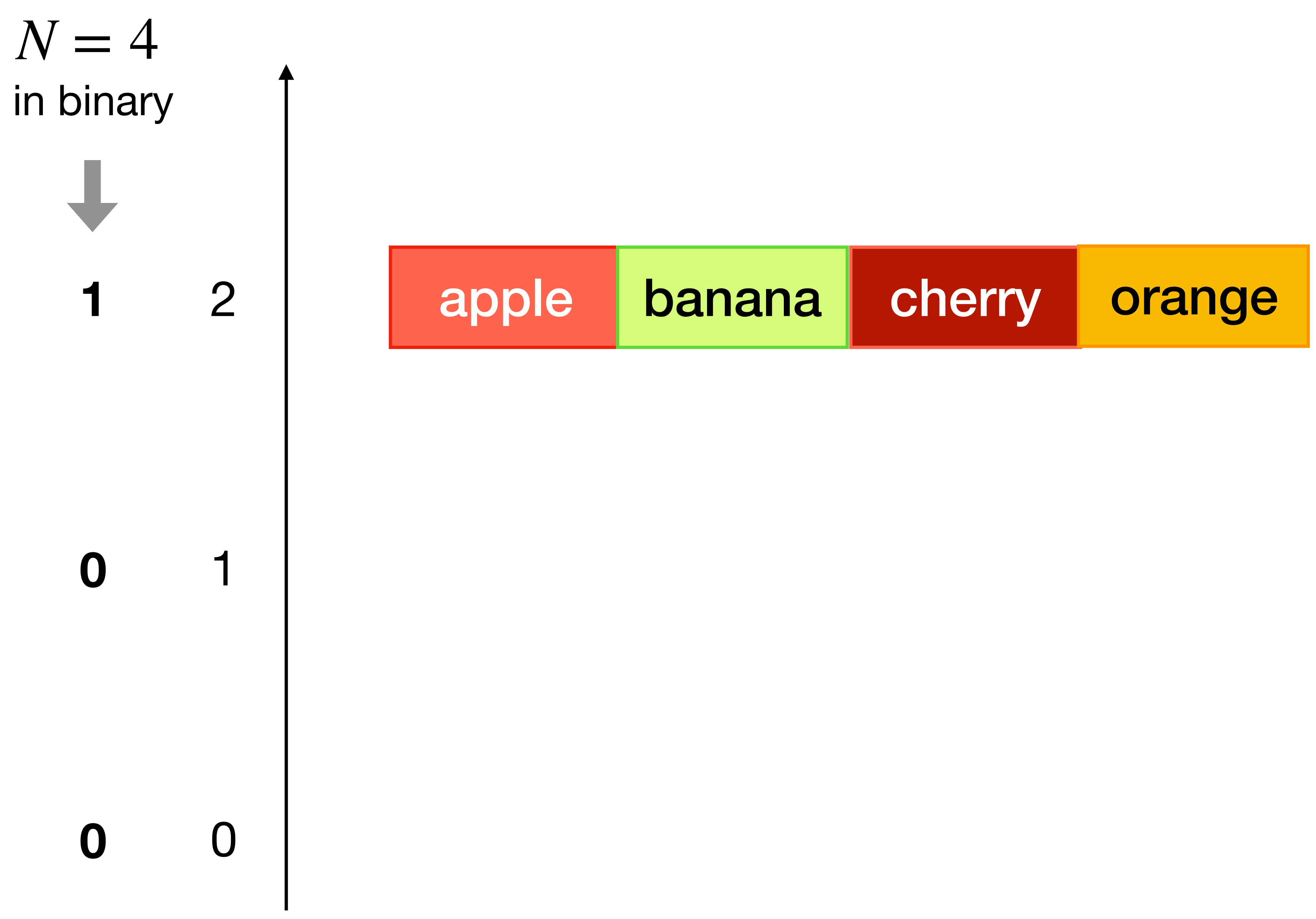
Basic COLA — Insertions

example: keys are names, values are colors



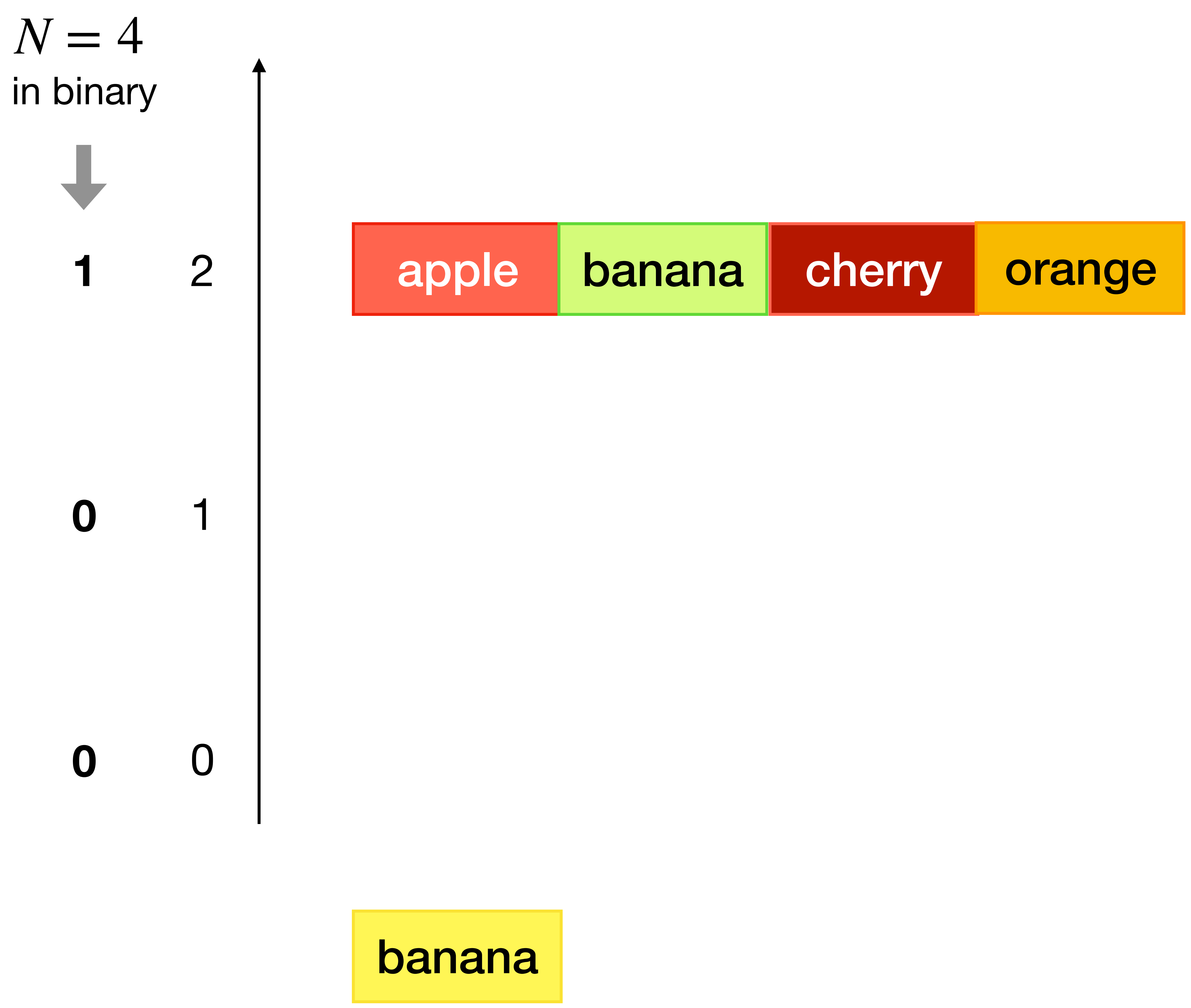
Basic COLA — Insertions

example: keys are names, values are colors



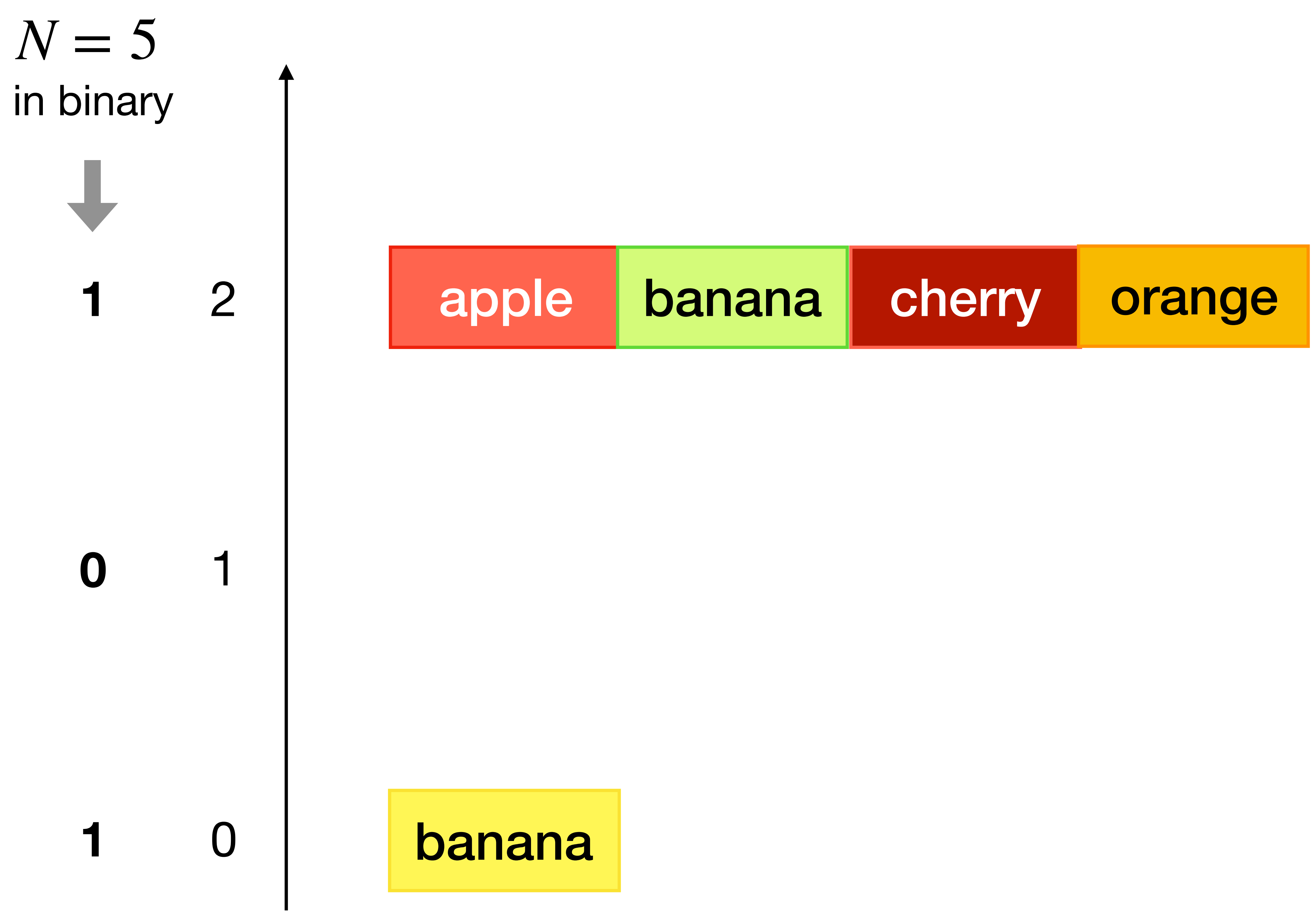
Basic COLA — Insertions

example: keys are names, values are colors



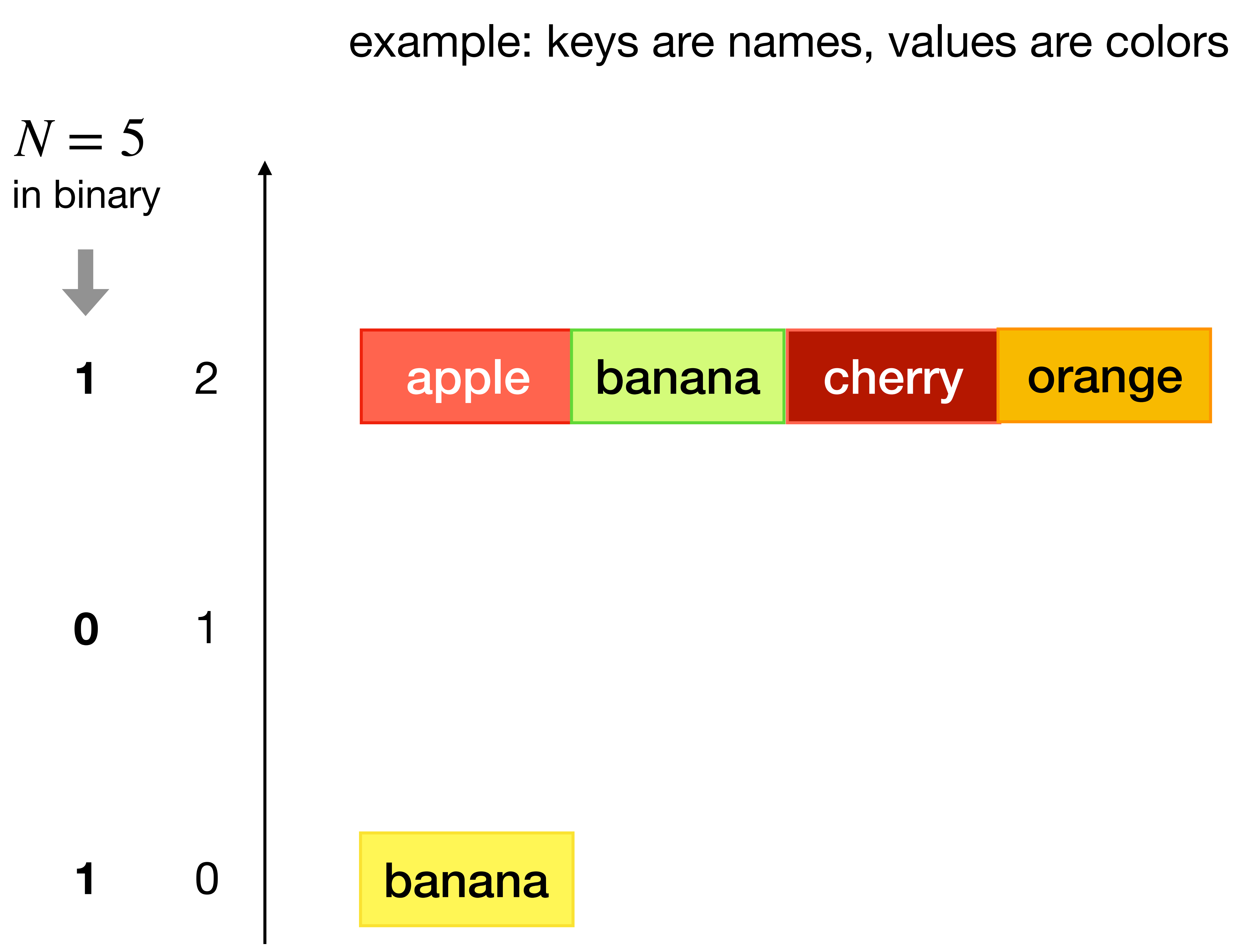
Basic COLA — Insertions

example: keys are names, values are colors



Complexity (block transfers)

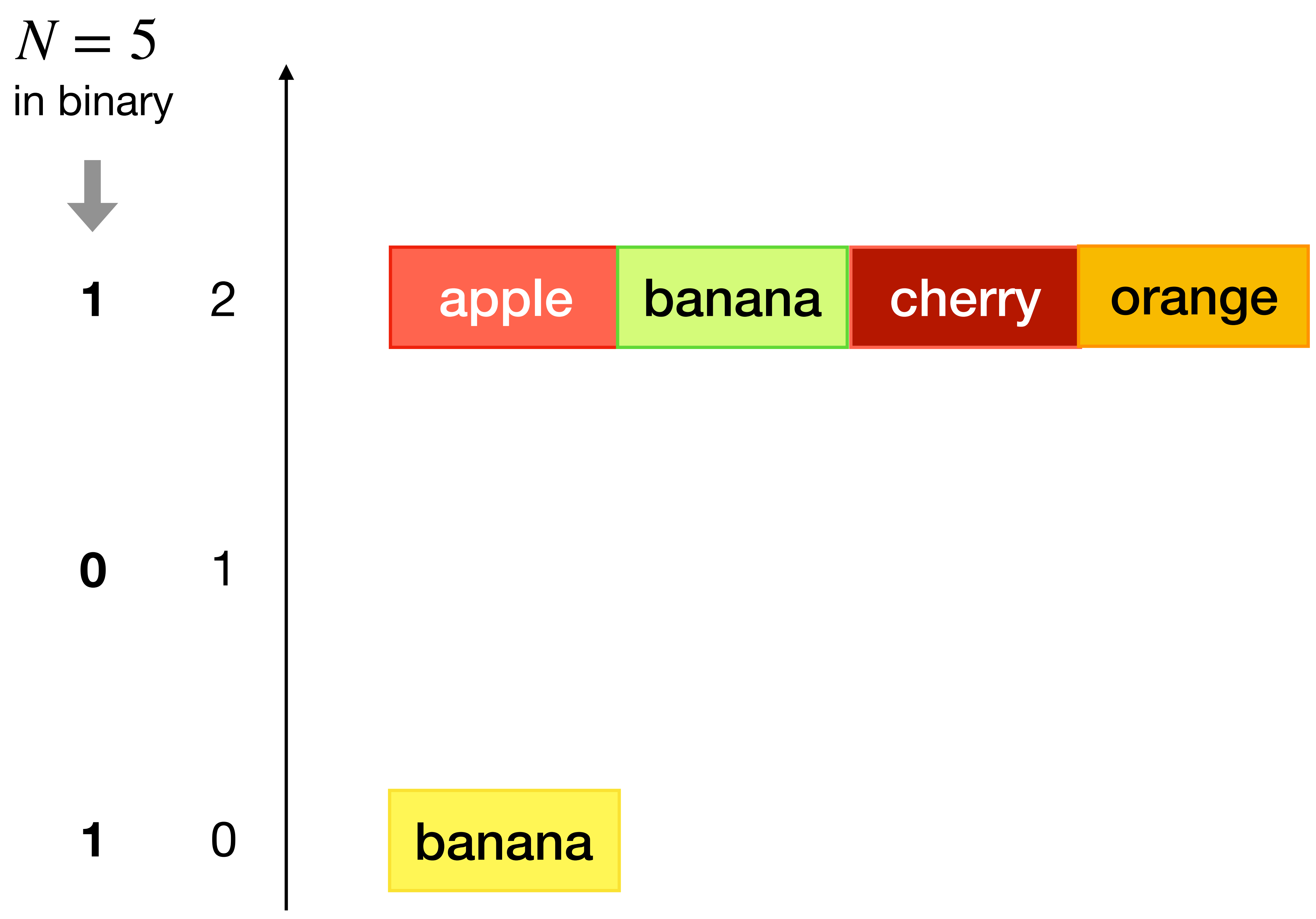
Basic COLA — Insertions



Complexity (block transfers)

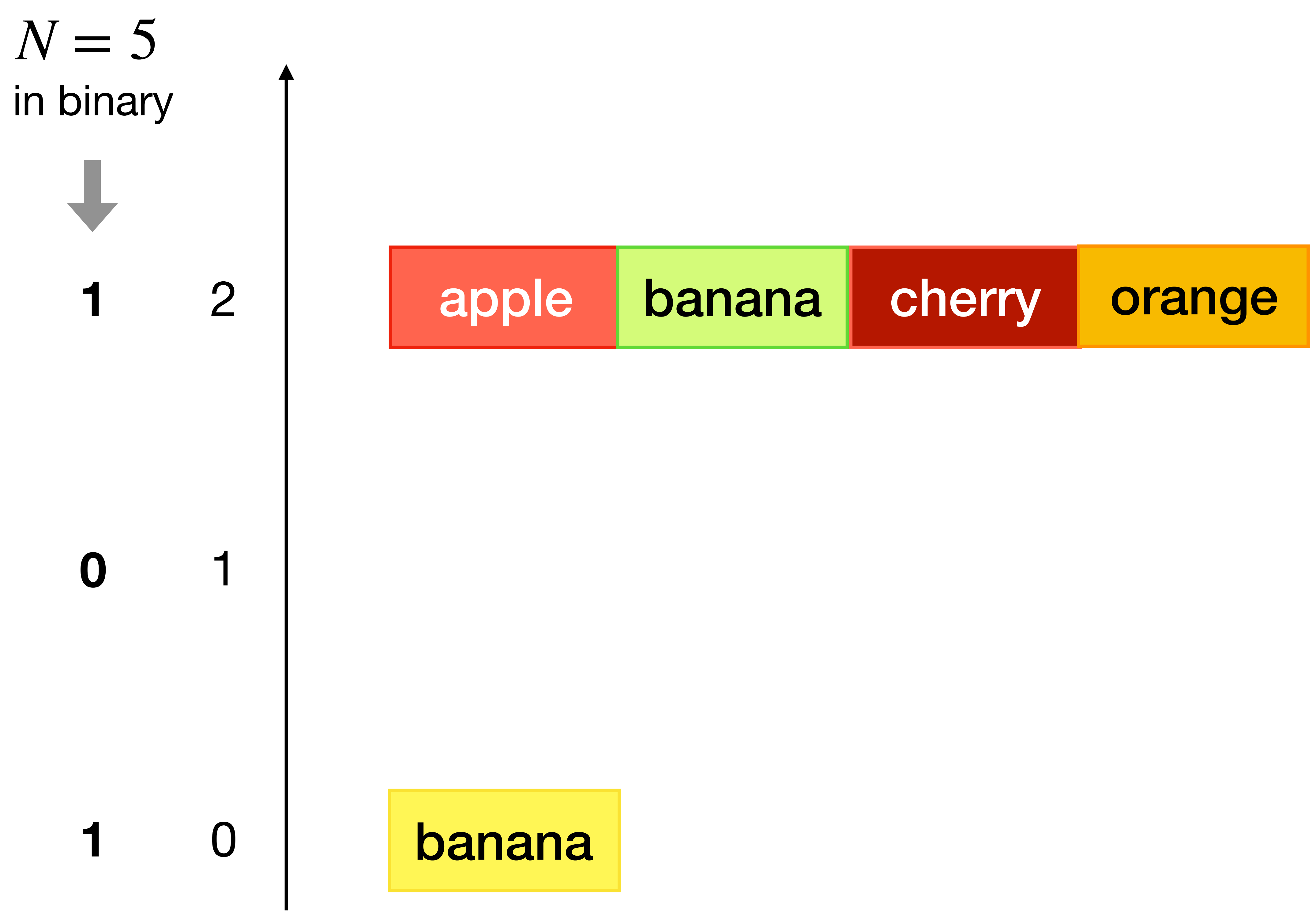
Basic COLA — Insertions

example: keys are names, values are colors



Basic COLA — Insertions

example: keys are names, values are colors

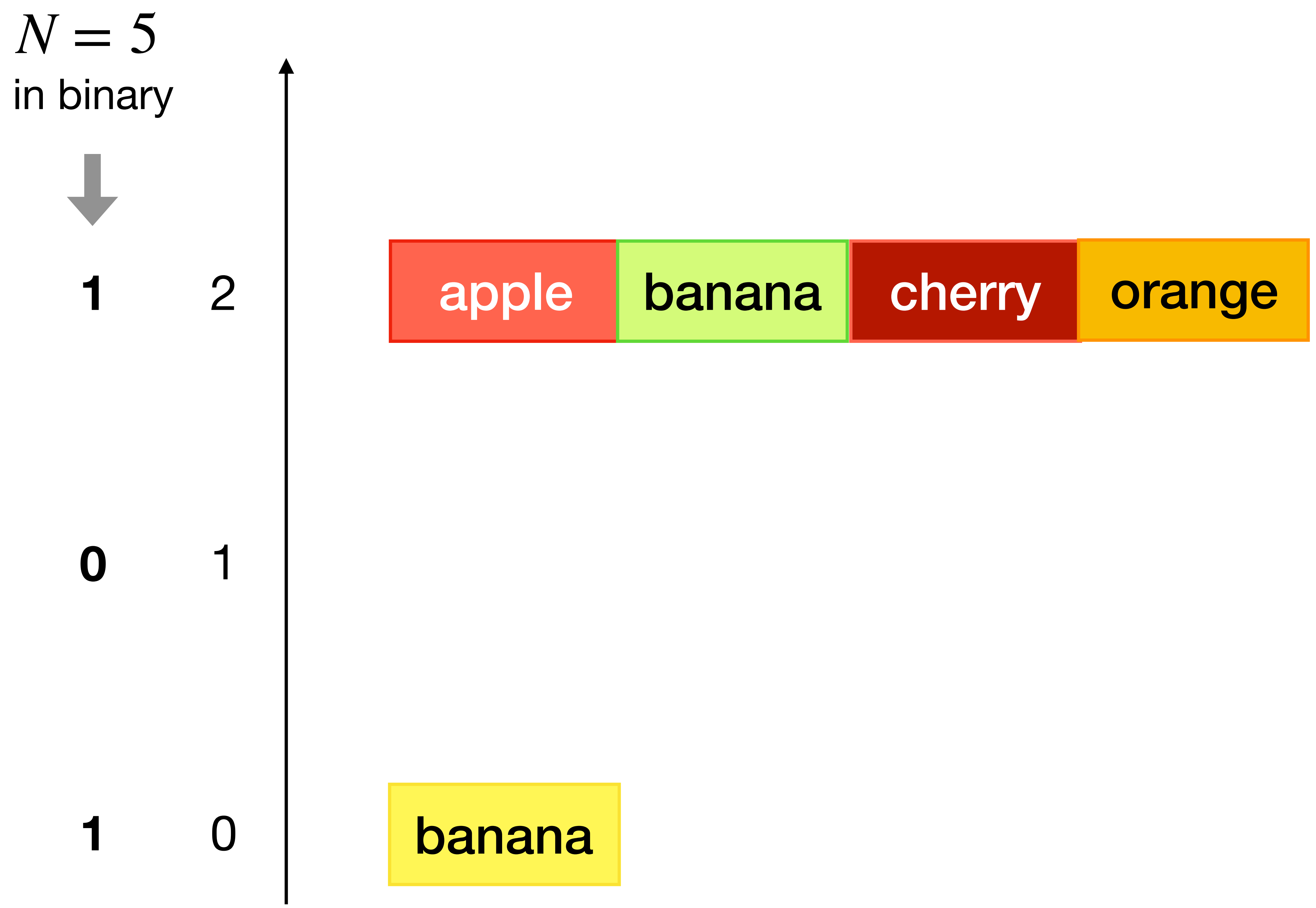


Complexity (block transfers)

- assume elements have size $O(1)$

Basic COLA — Insertions

example: keys are names, values are colors

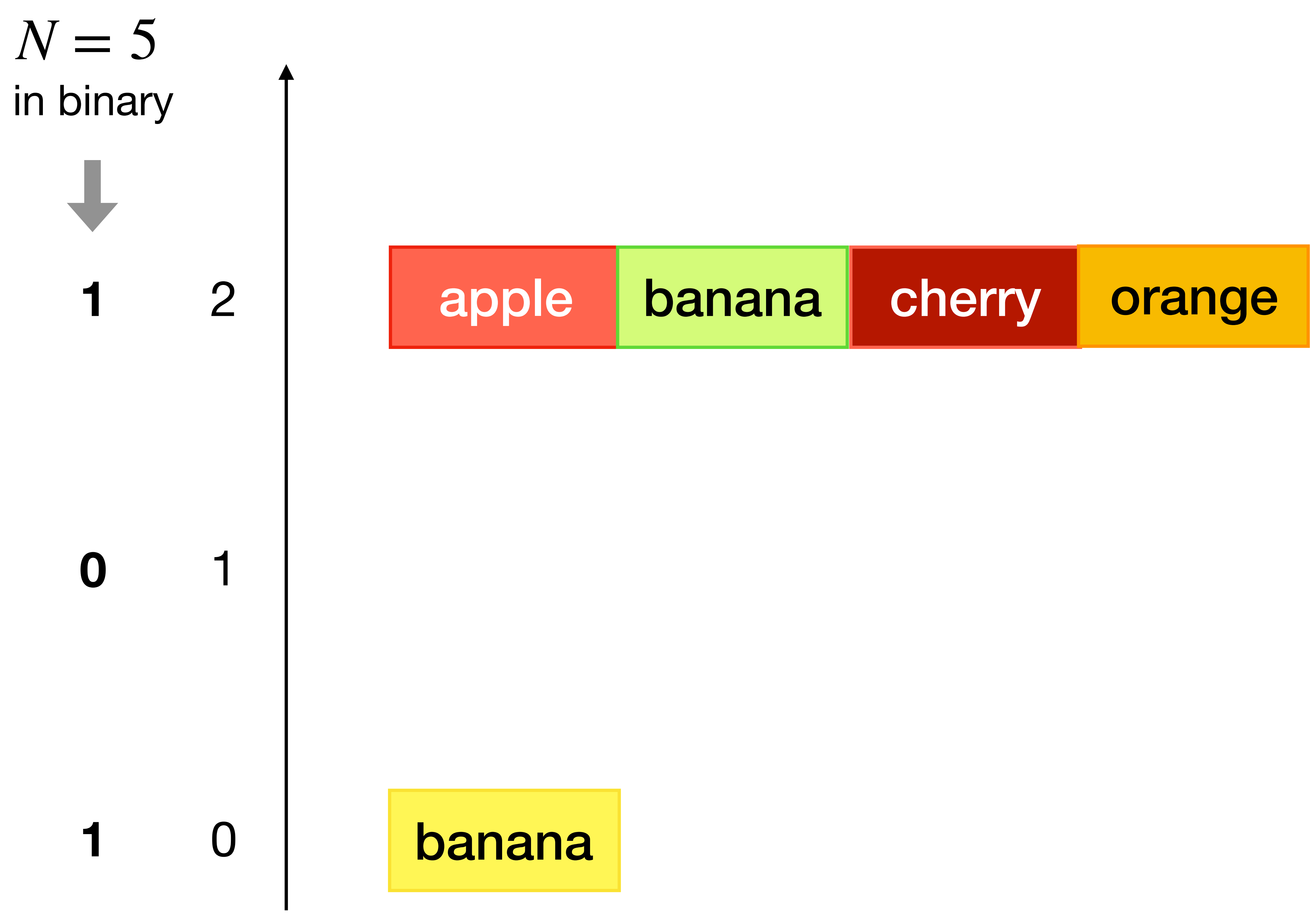


Complexity (block transfers)

- assume elements have size $O(1)$
- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)

Basic COLA — Insertions

example: keys are names, values are colors

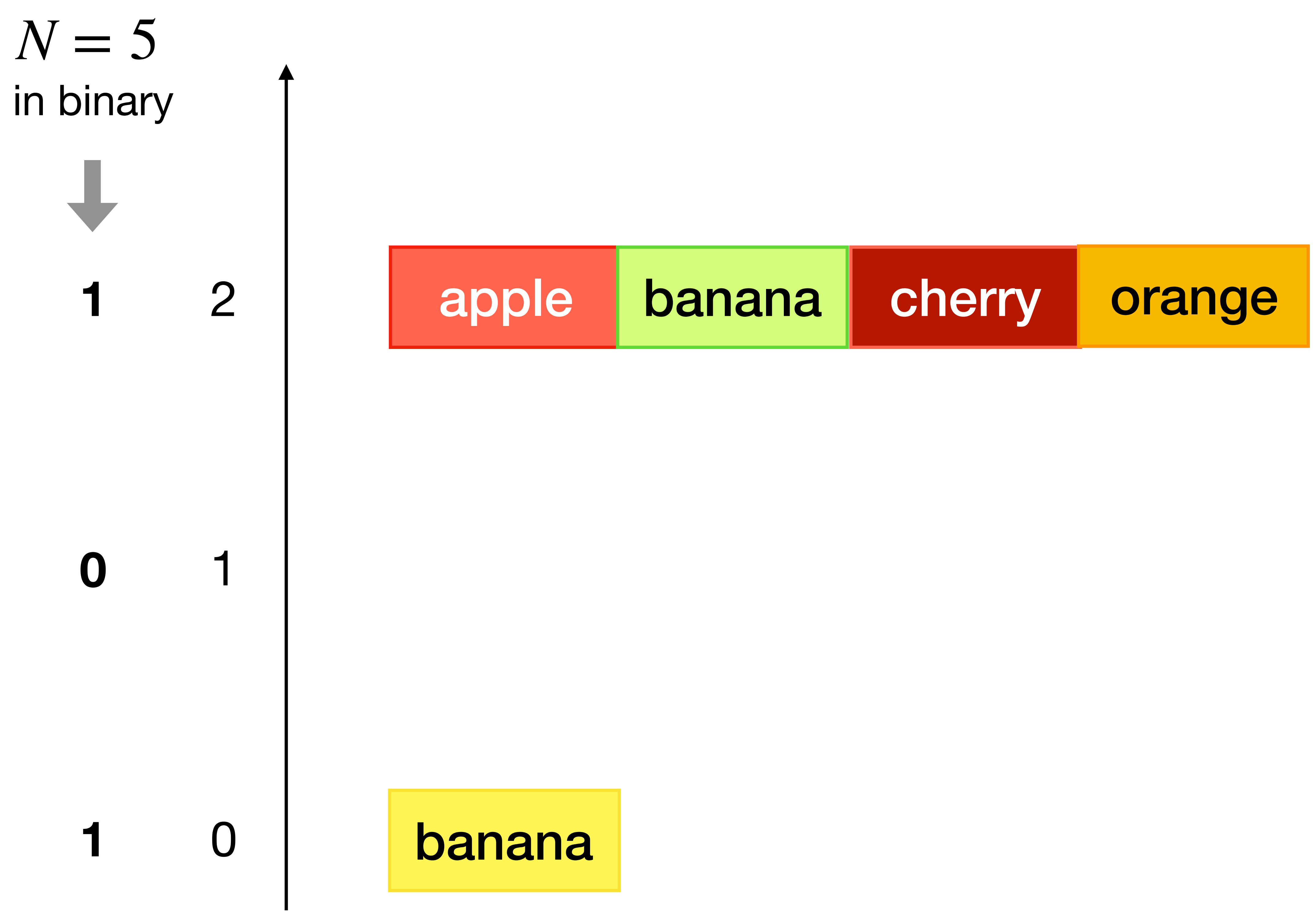


Complexity (block transfers)

- assume elements have size $O(1)$
- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)
- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)

Basic COLA — Insertions

example: keys are names, values are colors



Complexity (block transfers)

- assume elements have size $O(1)$
- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)
- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)
- amortized: $O\left(\frac{\log N}{B}\right)$

Complexity (block transfers)

- assume elements have size $O(1)$
- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)
- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)
- amortized: $O\left(\frac{\log N}{B}\right)$

Complexity (block transfers)

- assume elements have size $O(1)$

- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)

- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)

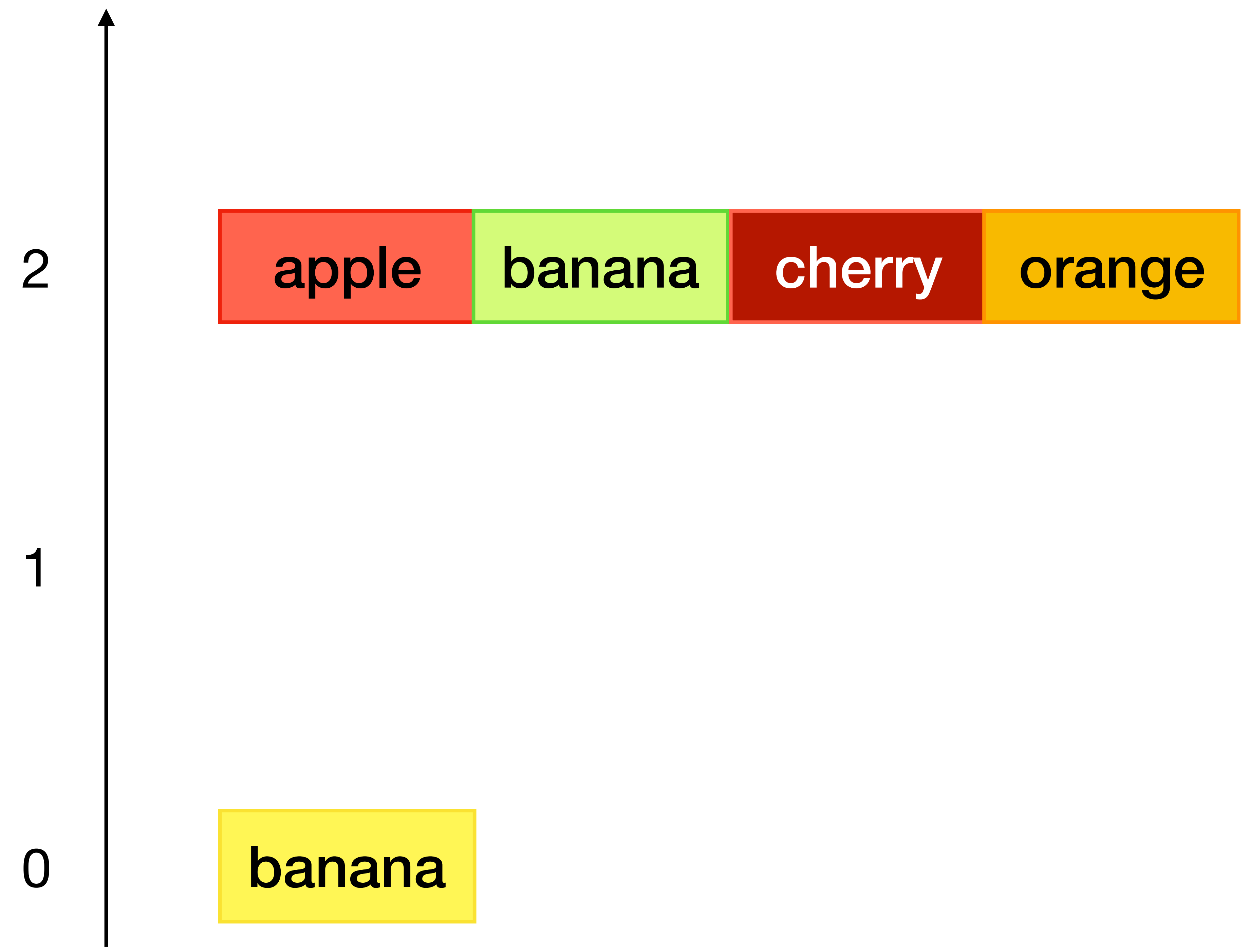
- amortized: $O\left(\frac{\log N}{B}\right)$

  amortized per-element cost for sequential writes is $O\left(\frac{1}{B}\right)$,

  an element is only written when it's merged,

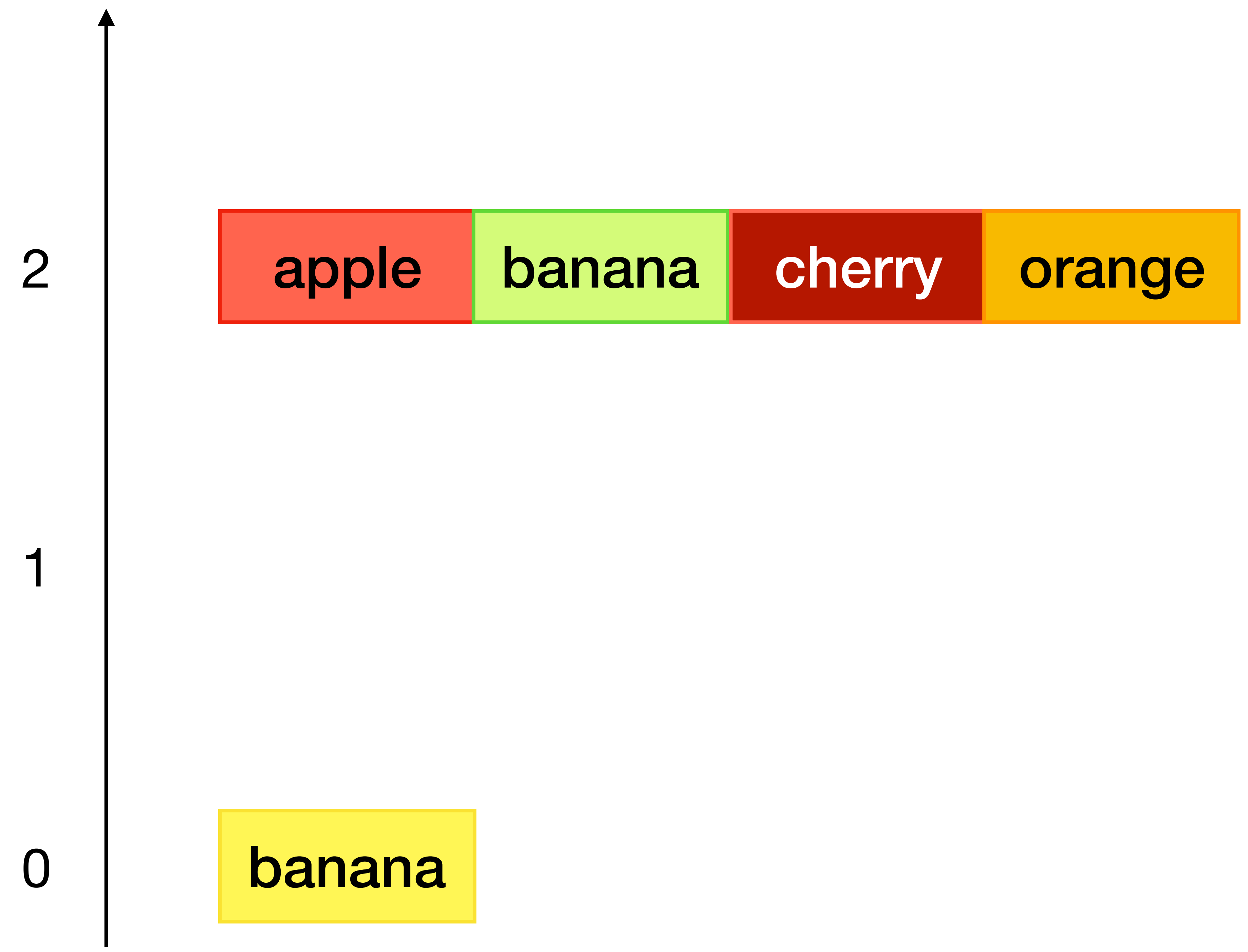
  an element can only be merged $O(\log N)$ times

Basic COLA — Queries



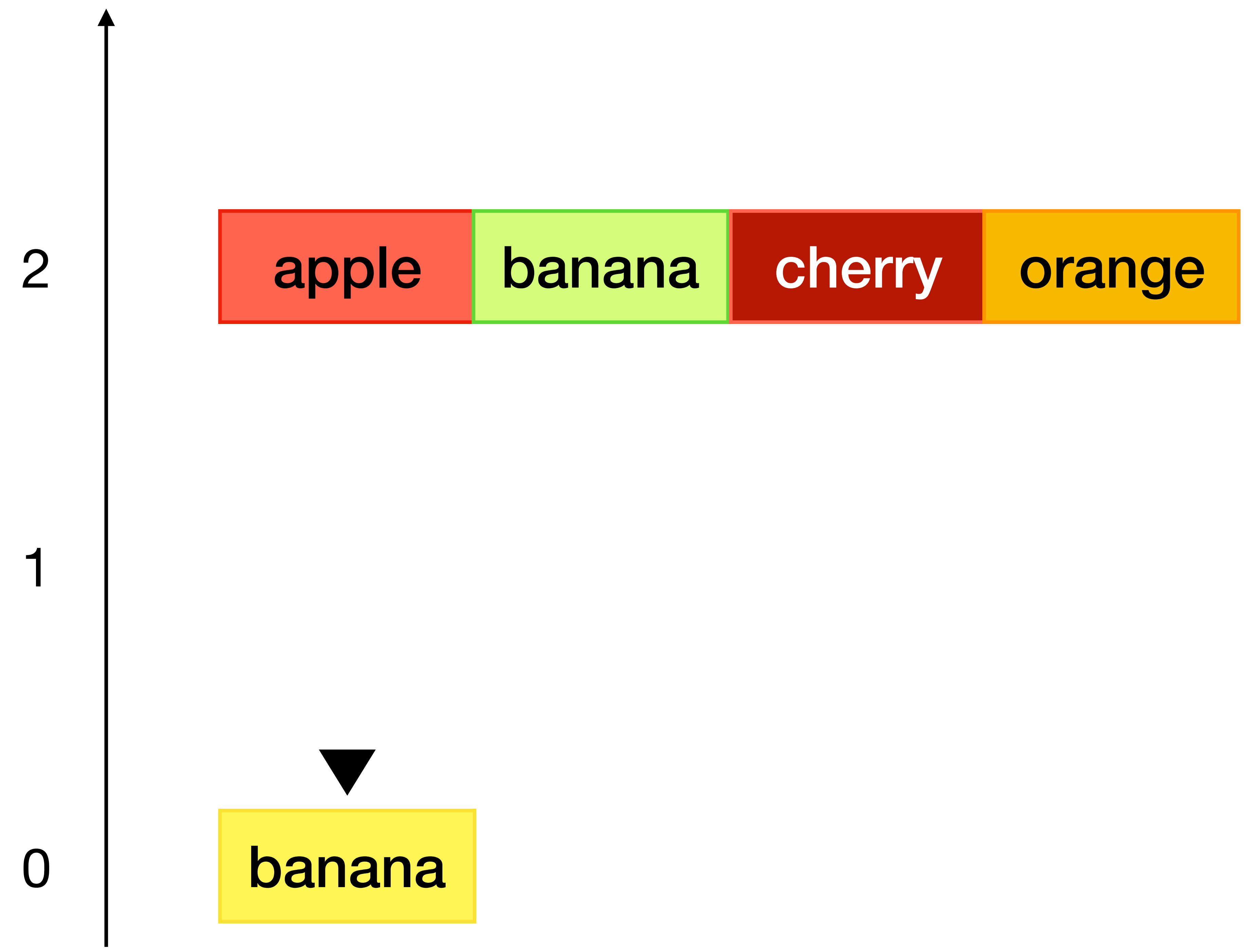
Basic COLA — Queries

query “**banana**”



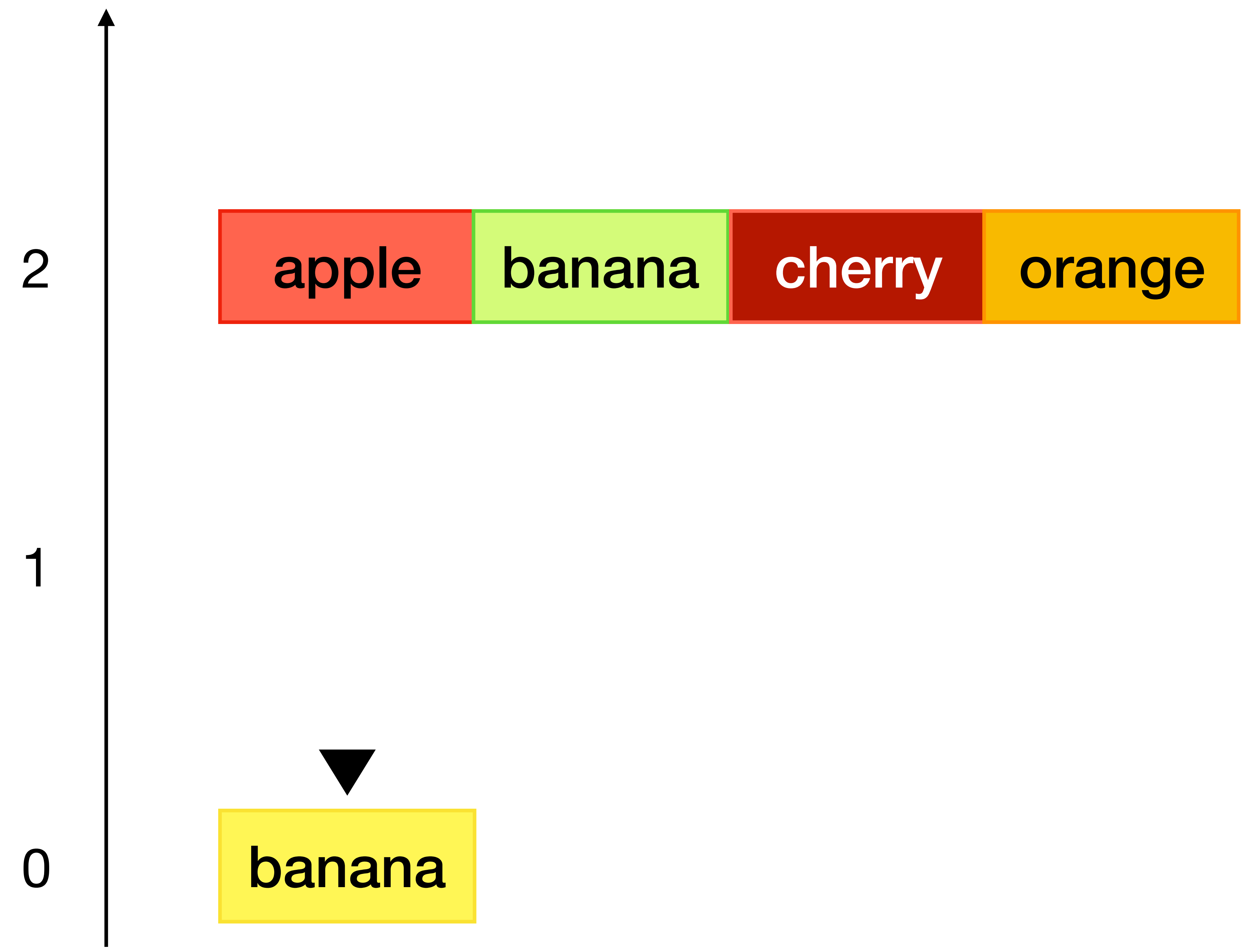
Basic COLA — Queries

query “banana”

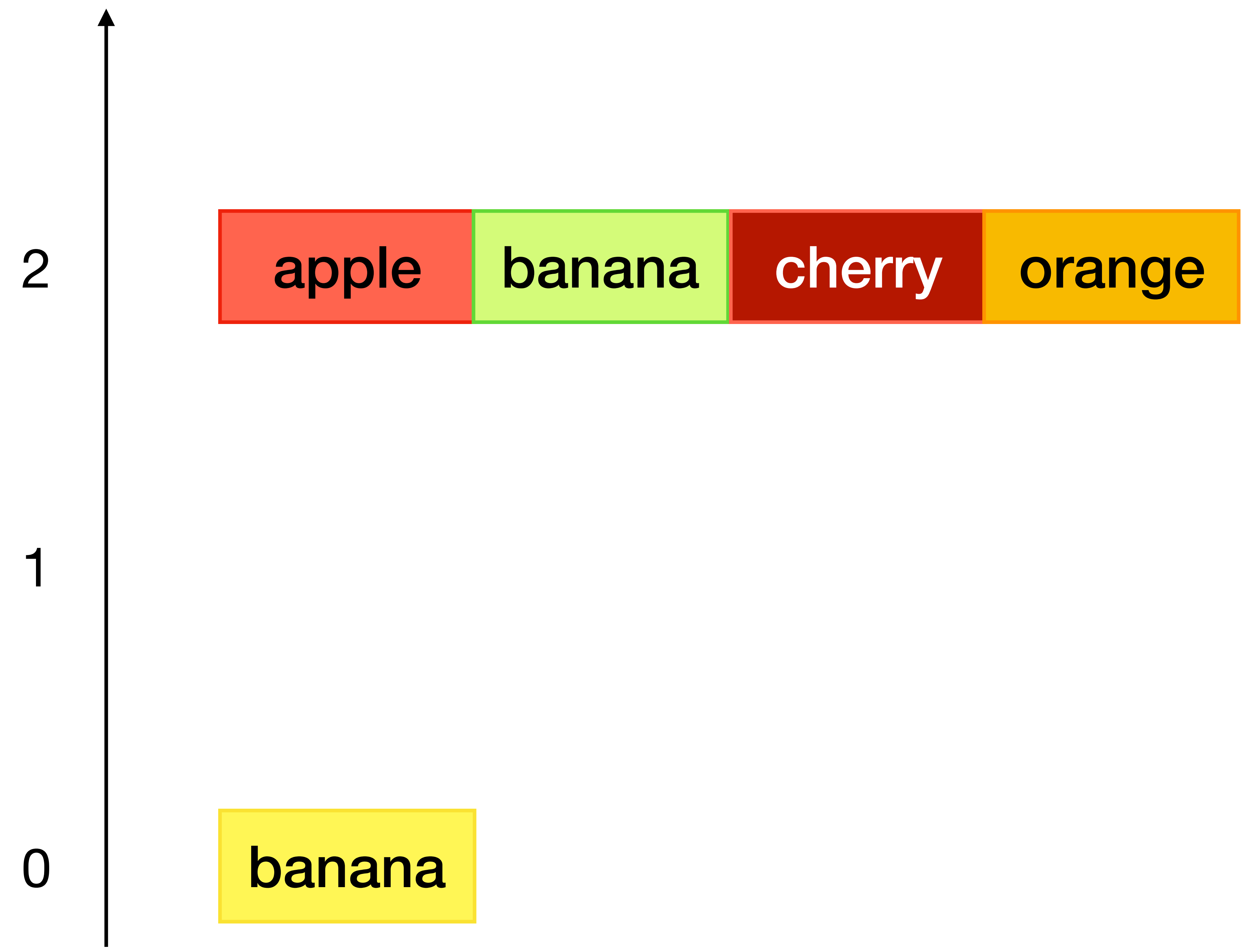


Basic COLA — Queries

query “**banana**”

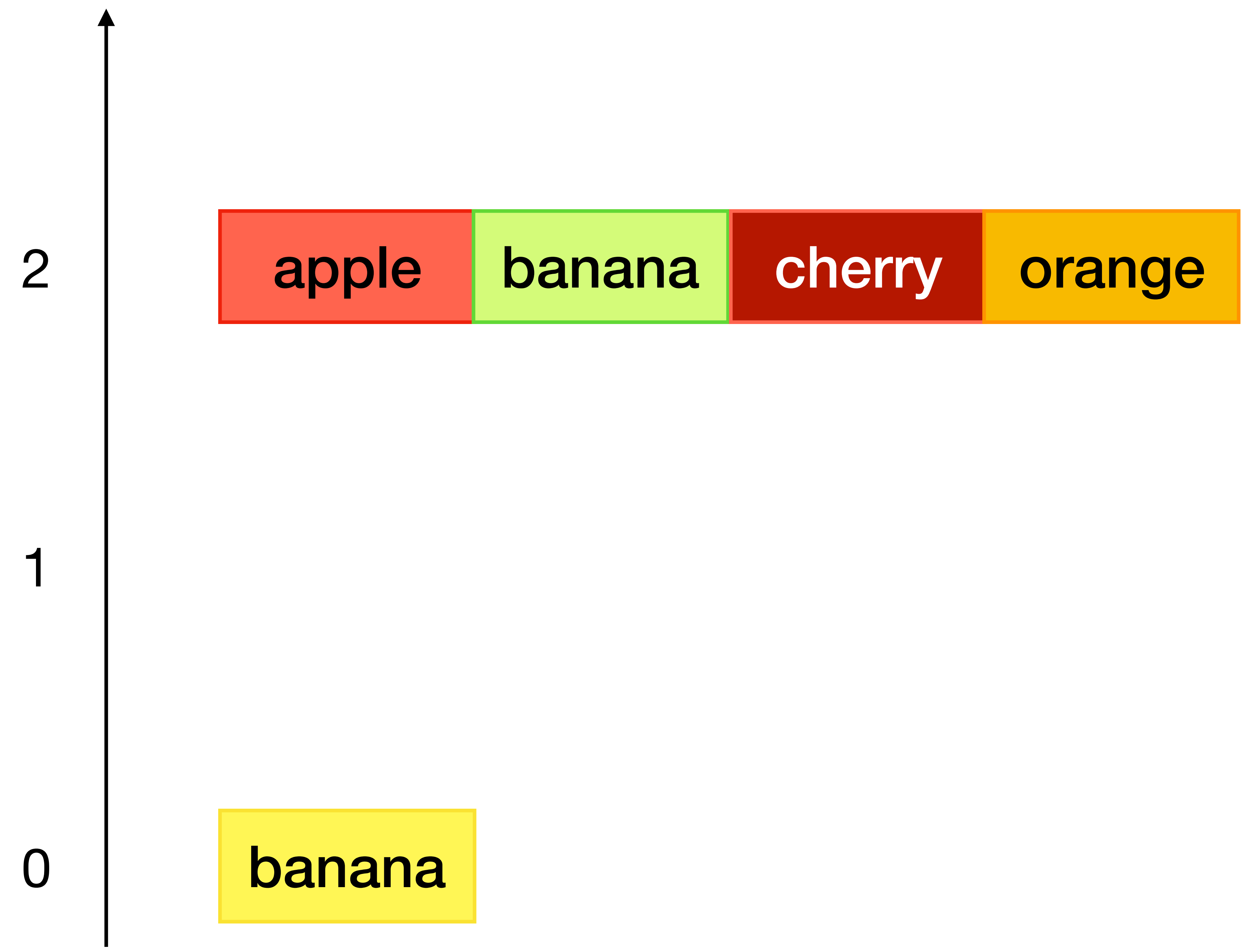


Basic COLA — Queries



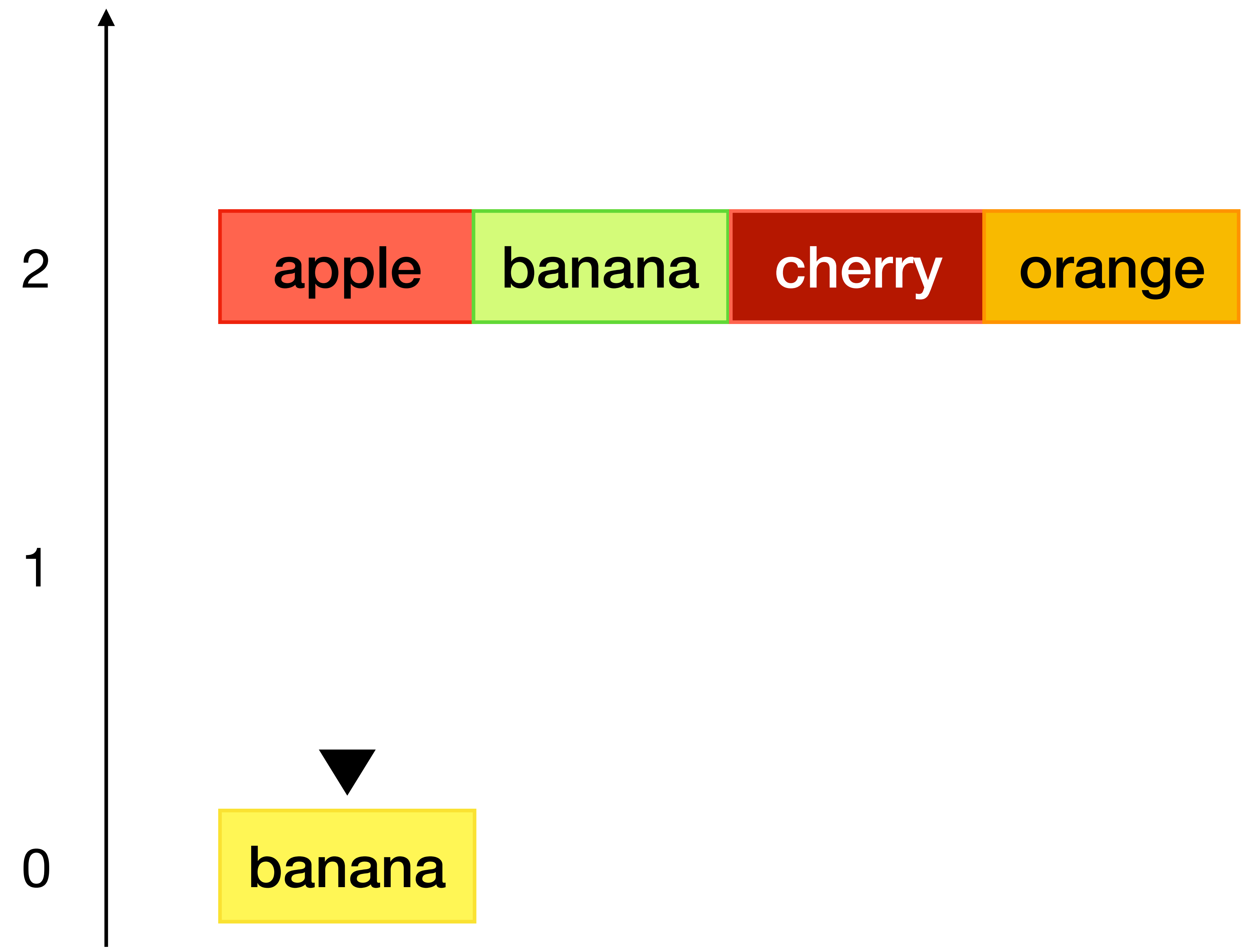
Basic COLA — Queries

query "orange"



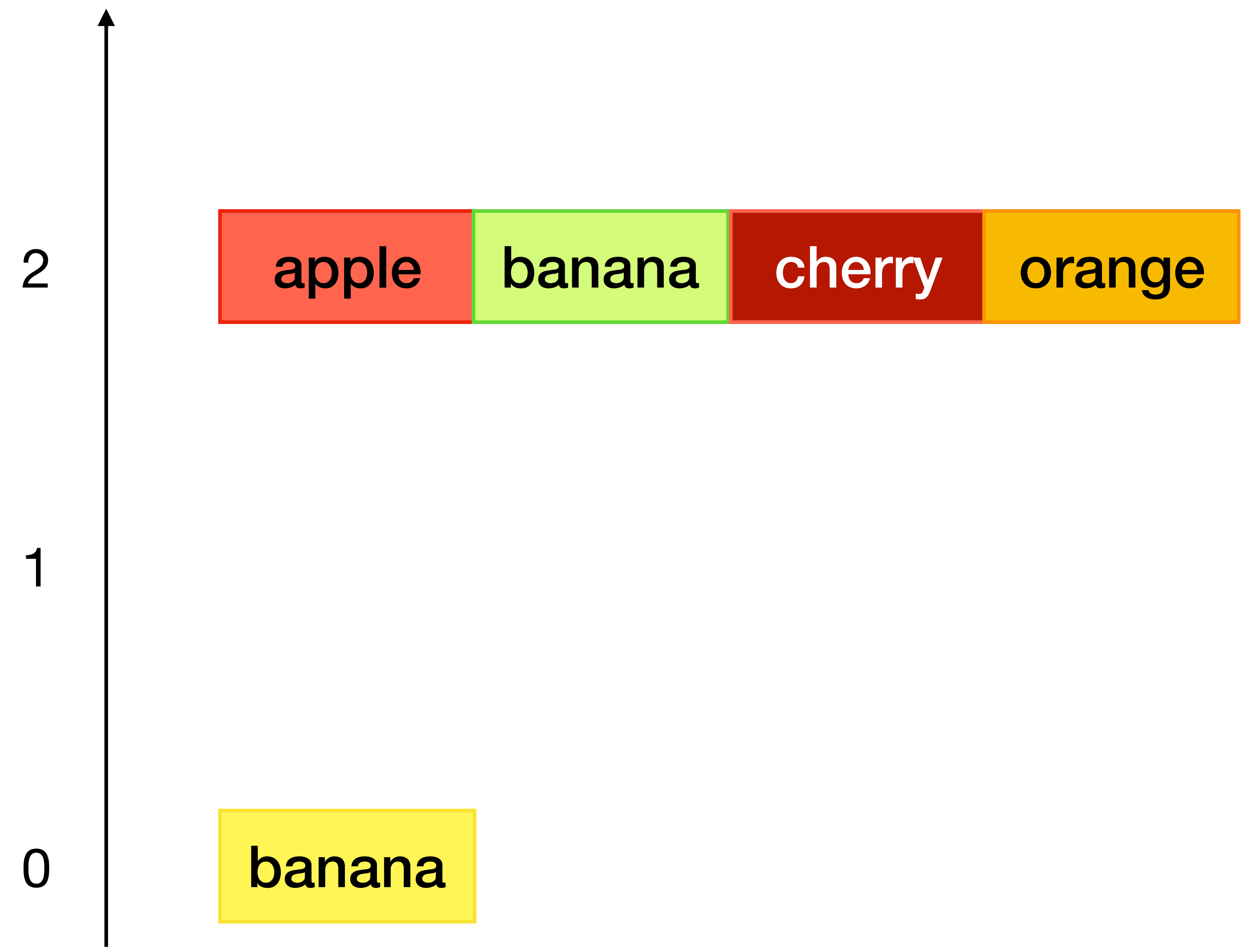
Basic COLA — Queries

query “orange”



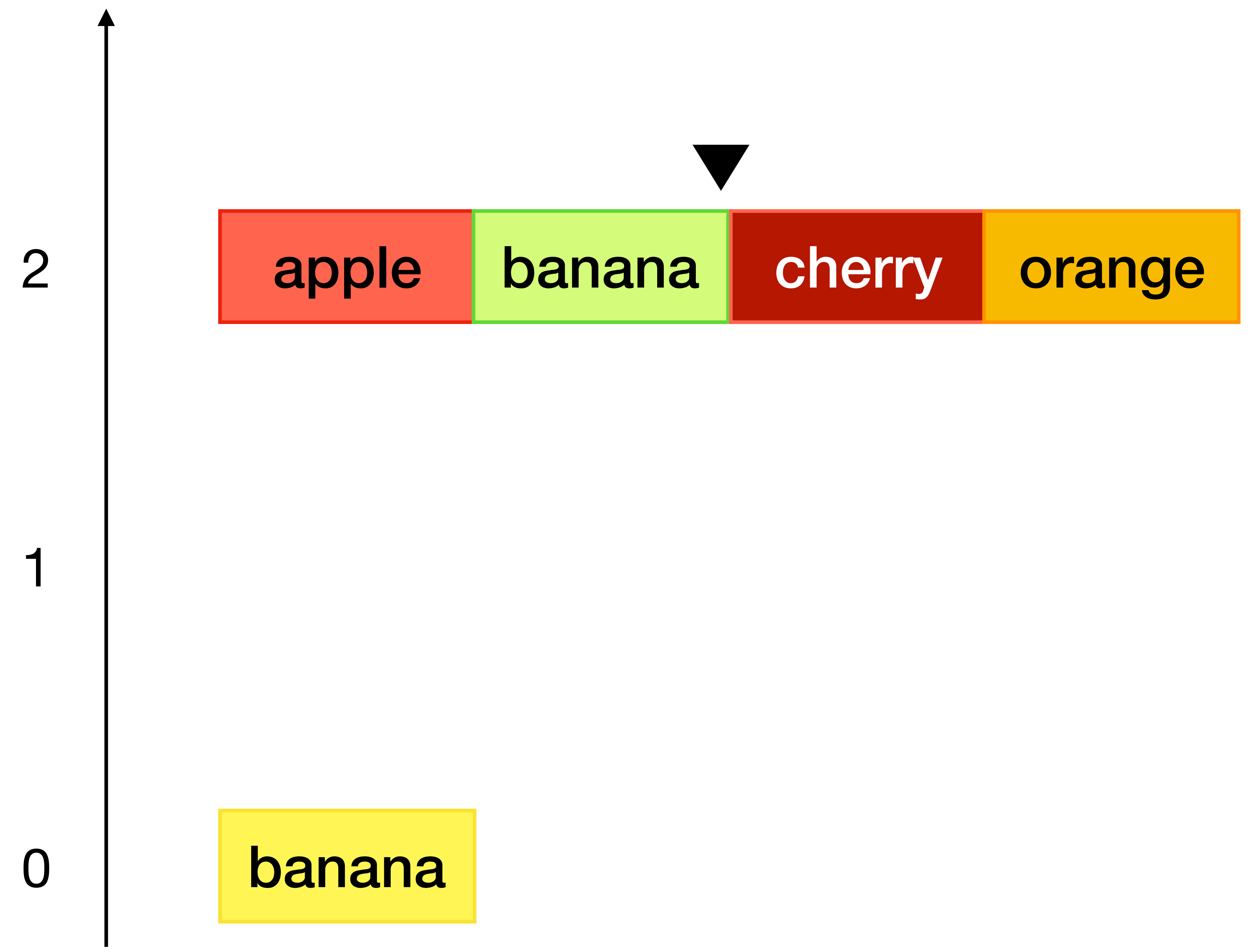
Basic COLA — Queries

query "orange"



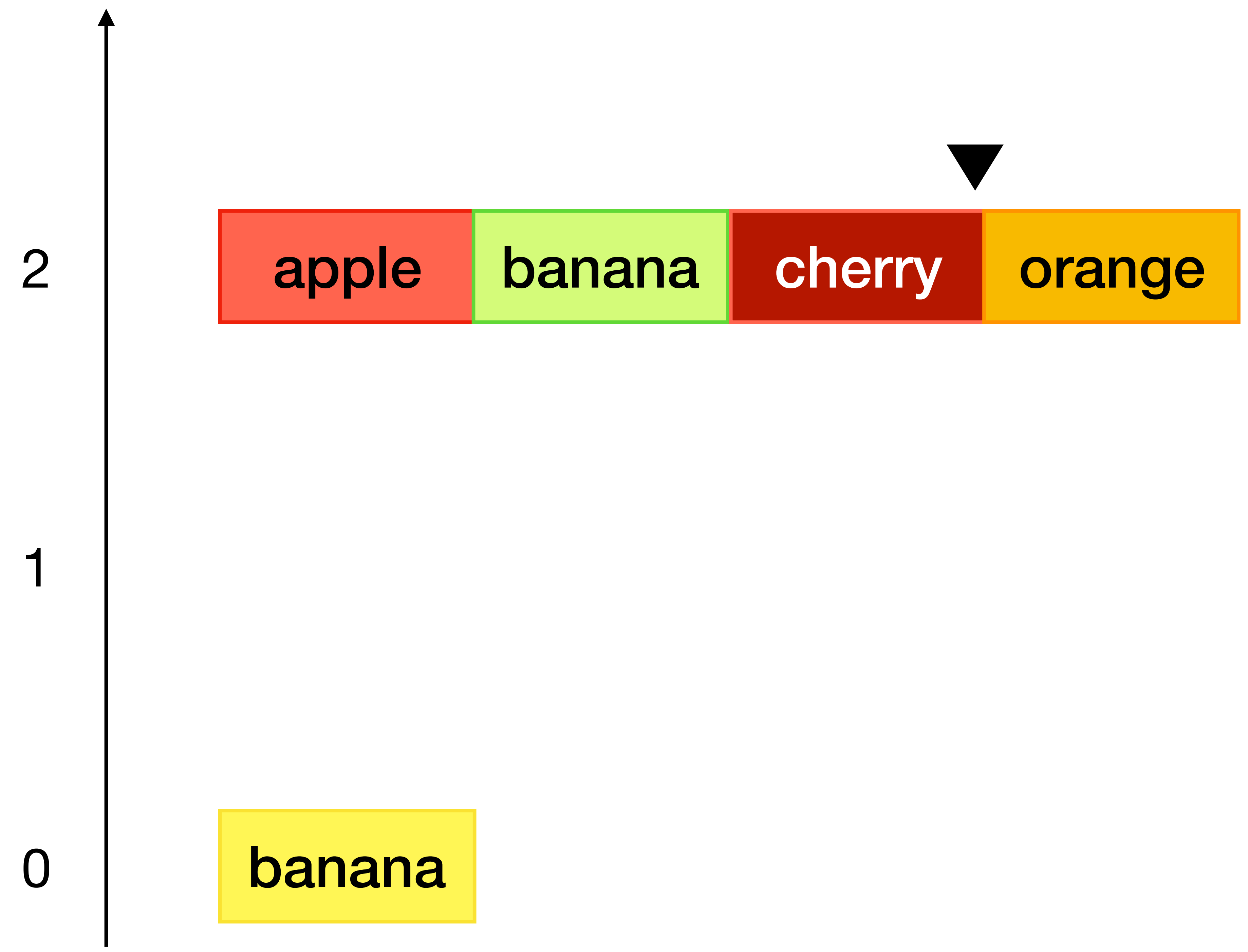
Basic COLA — Queries

query “orange”



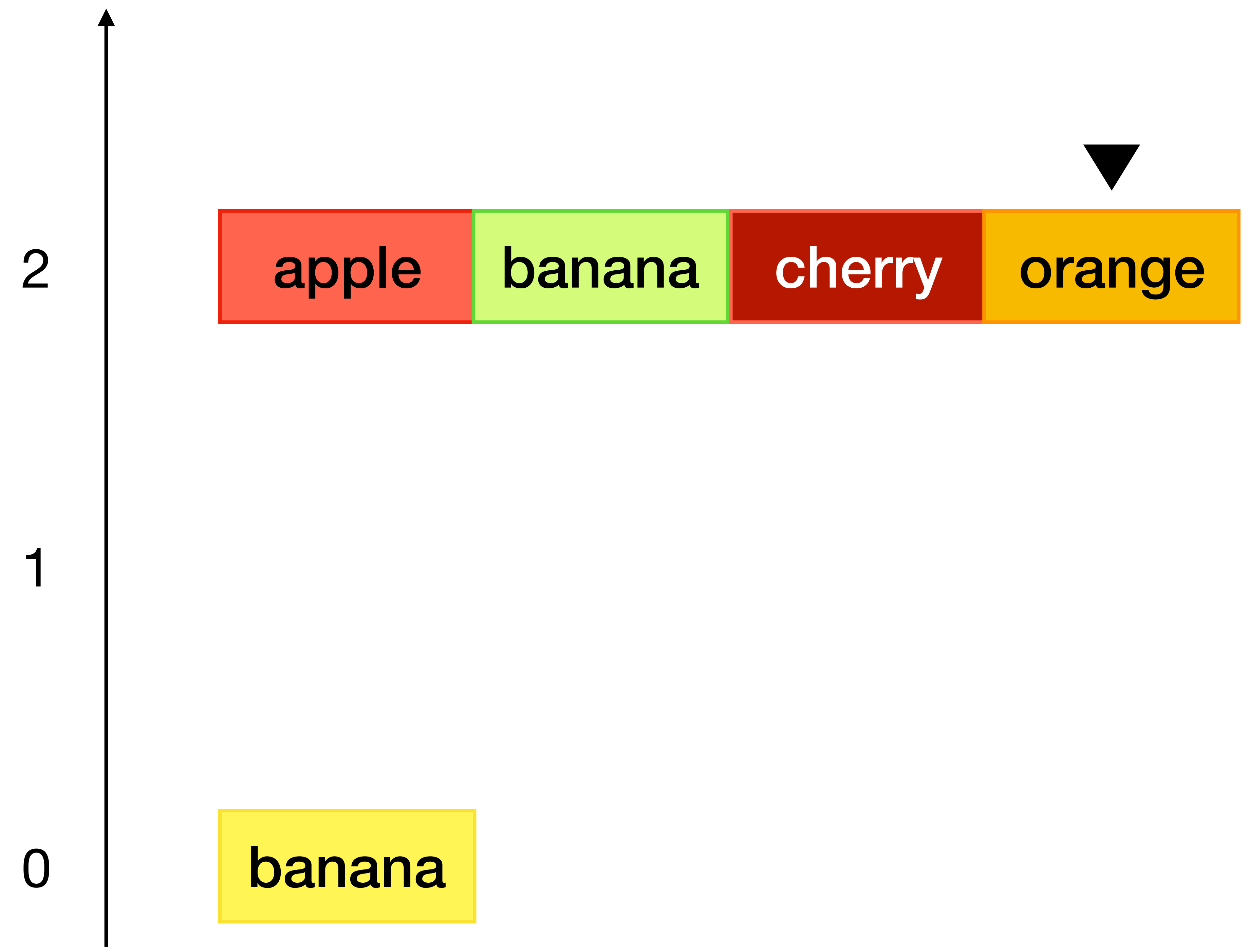
Basic COLA — Queries

query “orange”



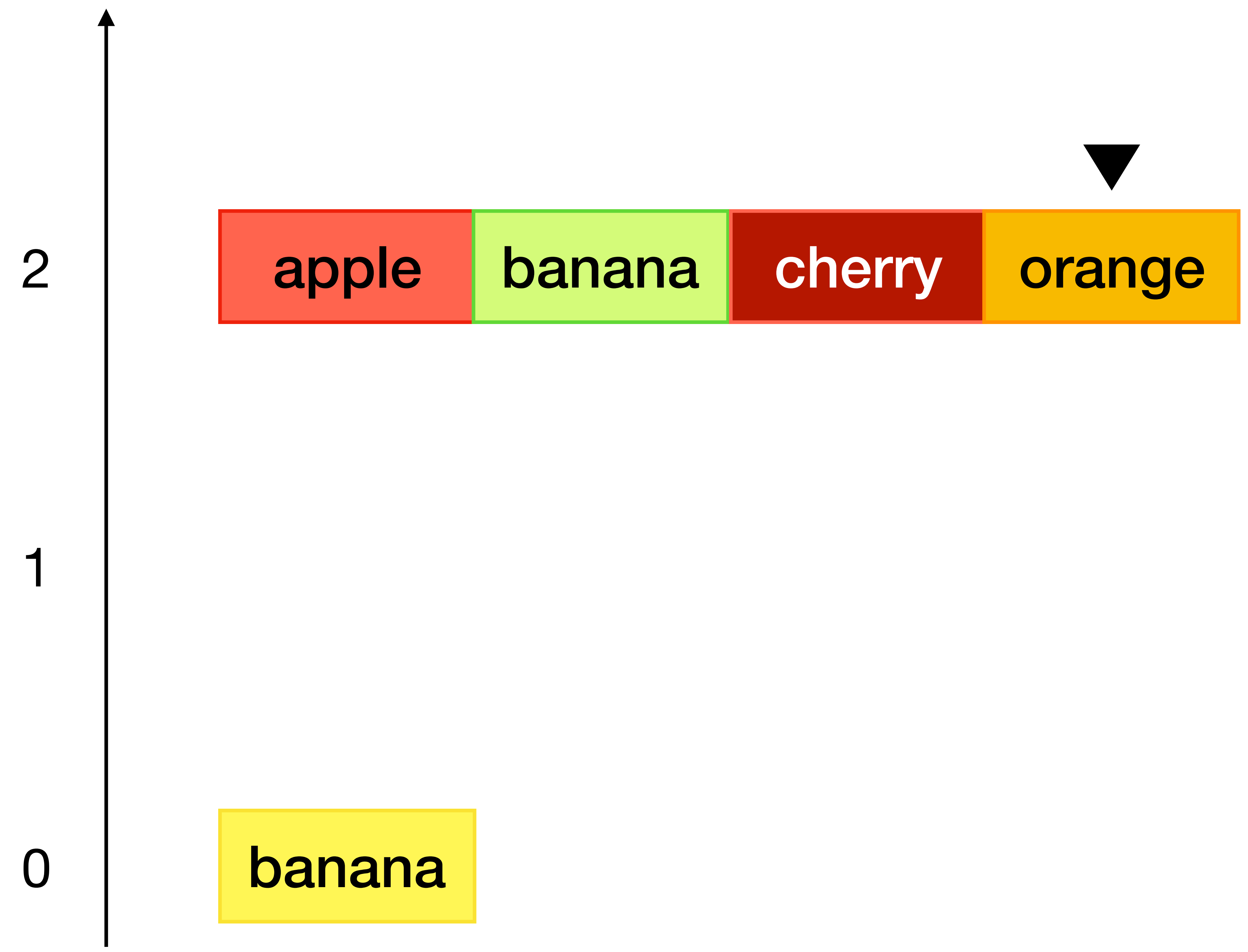
Basic COLA — Queries

query "orange"

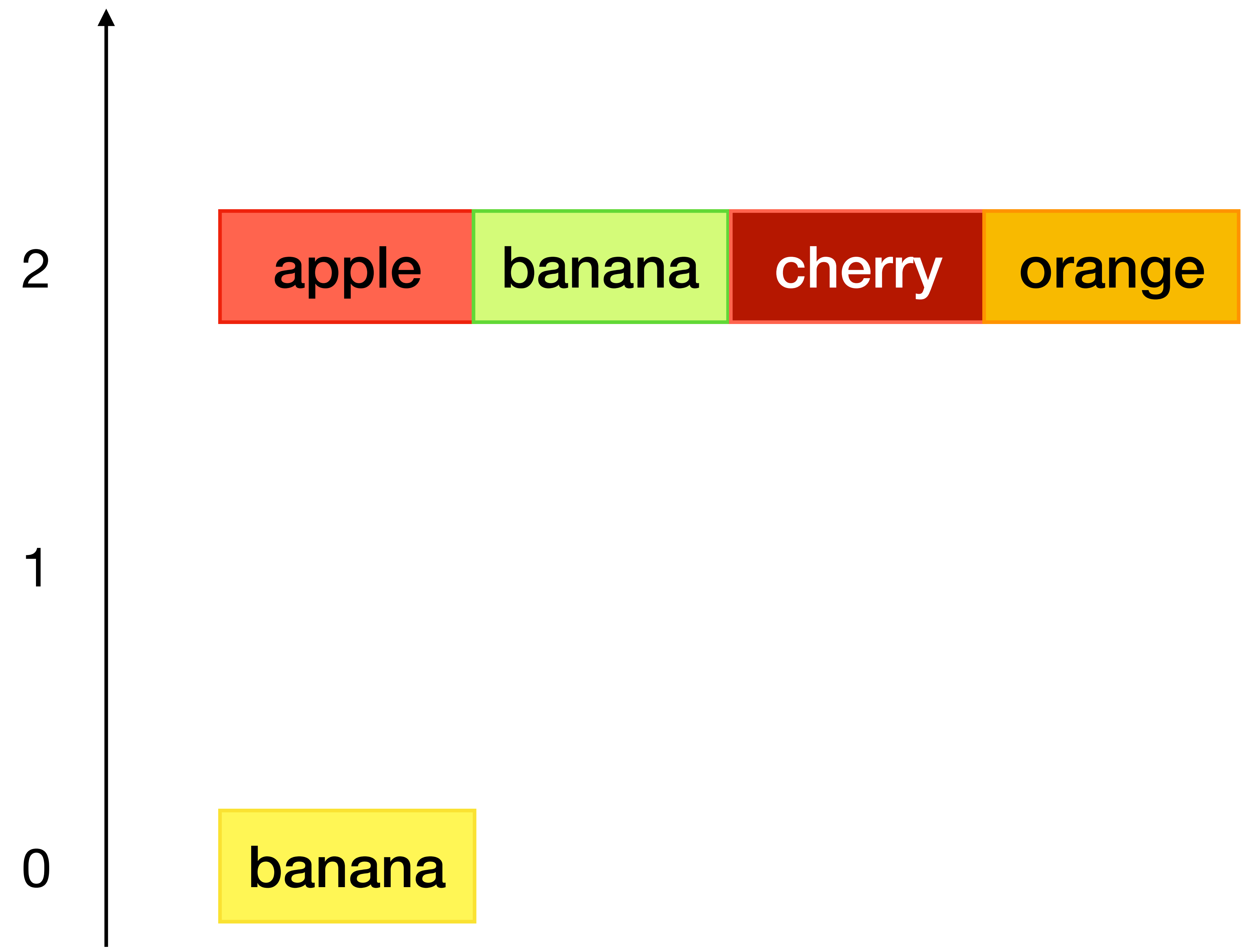


Basic COLA — Queries

query “orange”

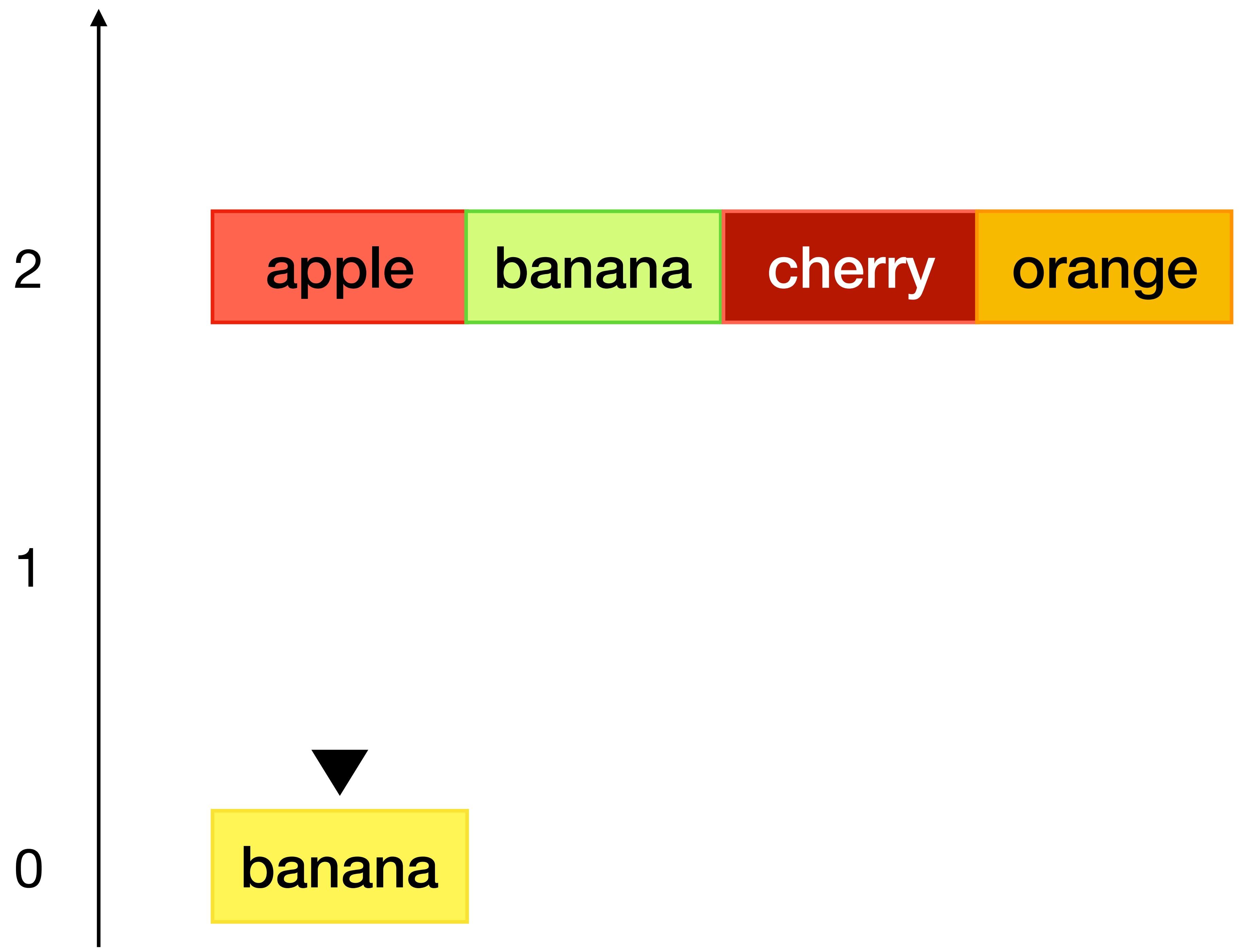


Basic COLA — Queries



<u>Complexity</u> (block transfers)

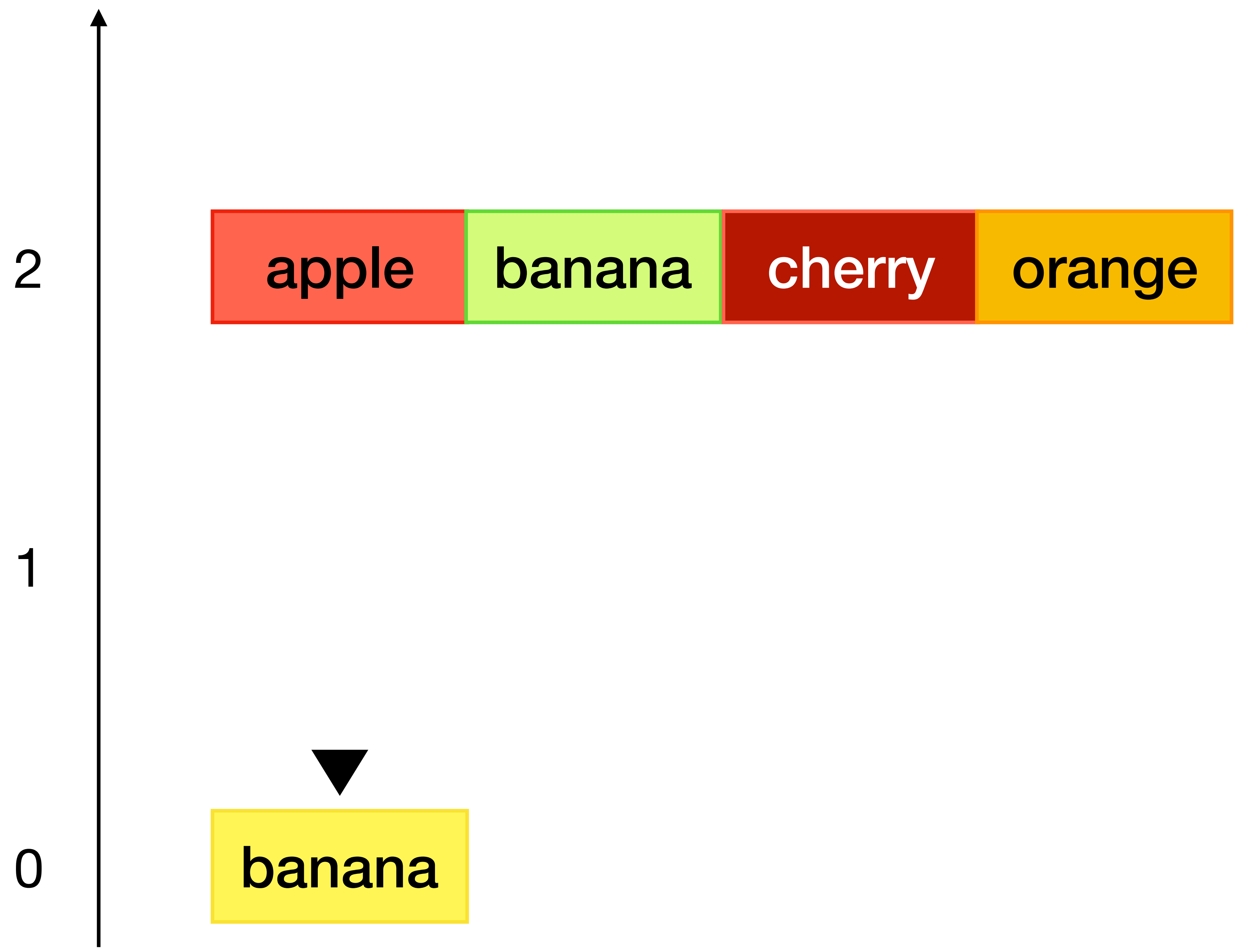
- assume elements have ***the same*** size $O(1)$
  - otherwise, binary search doesn't work



Complexity (block transfers)

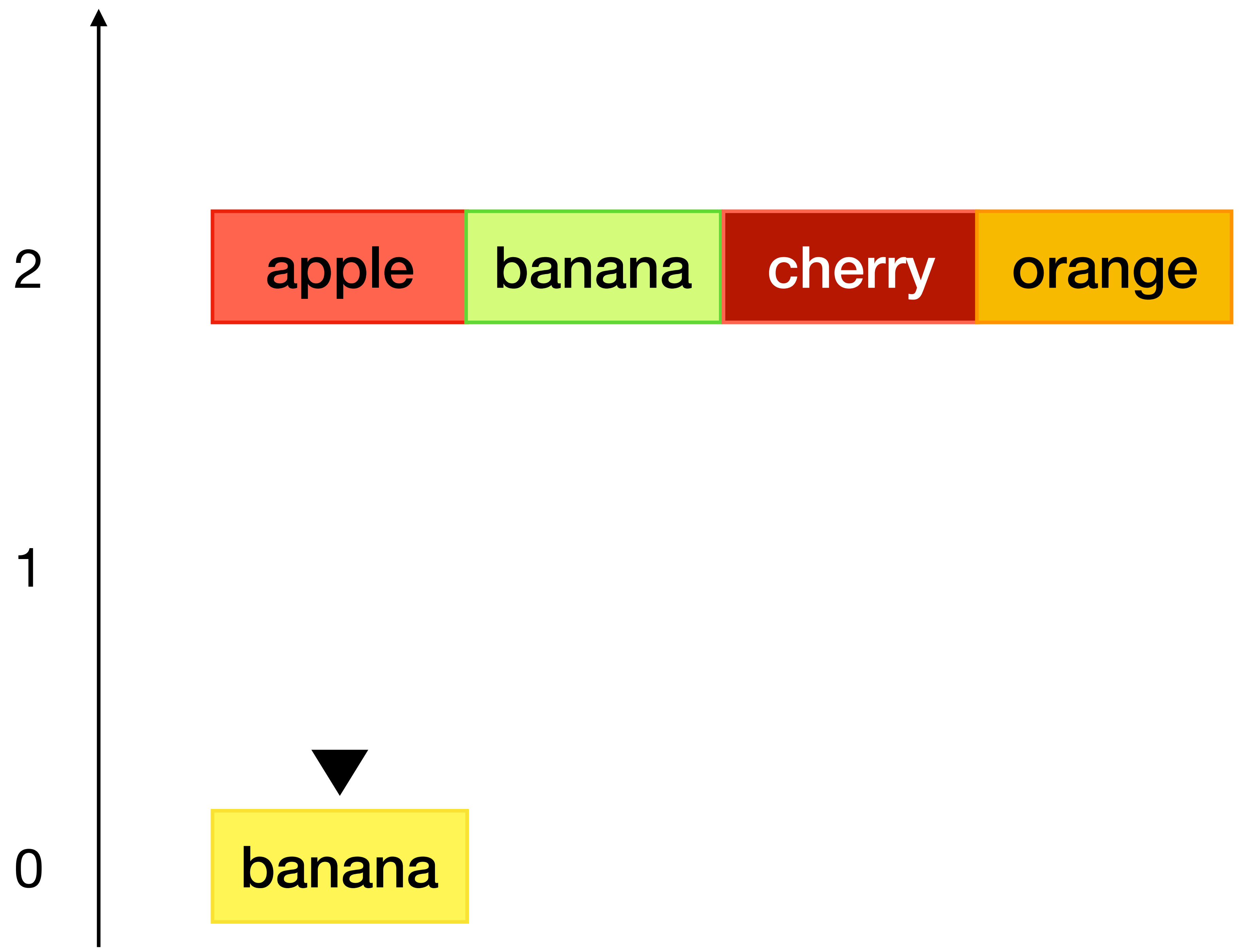
- assume elements have ***the same*** size $O(1)$
  - otherwise, binary search doesn't work

- binary search costs $O\left(\log \frac{N}{B}\right)$ $= O(\log N - \log B)$ per array



Complexity (block transfers)

- assume elements have ***the same*** size $O(1)$
  - otherwise, binary search doesn't work

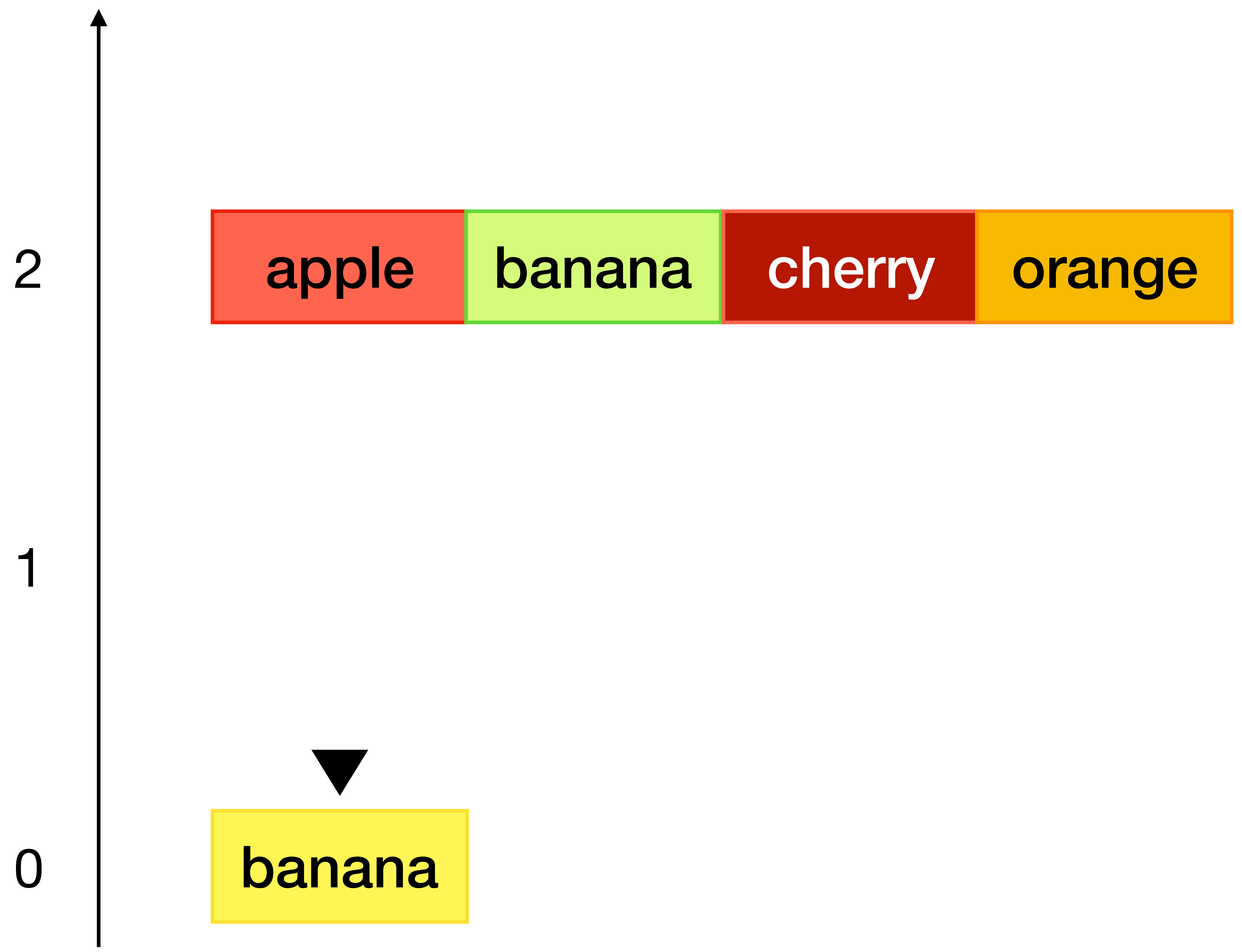


- binary search costs $O\left(\log \frac{N}{B}\right)$ $= O(\log N - \log B)$ per array

- have $O(\log N - \log B)$ arrays of size $\geq B$

Complexity (block transfers)

- assume elements have ***the same*** size $O(1)$
  - otherwise, binary search doesn't work



- binary search costs $O\left(\log \frac{N}{B}\right)$ $= O(\log N - \log B)$ per array
- have $O(\log N - \log B)$ arrays of size $\geq B$
- $\Rightarrow$ query complexity is $O\left(\left(\log N - \log B\right)^2\right)$

Lookaheads

Lookaheads

- idea: speed up queries using ***fractional cascading***

Lookaheads

- idea: speed up queries using ***fractional cascading***
- every eighth key of array i is replicated in array $i - 1$ as a ***lookahead pointer***

Lookaheads

- idea: speed up queries using ***fractional cascading***
- every eighth key of array i is replicated in array $i-1$ as a ***lookahead pointer***
  - sorted order of array maintained

Lookaheads

- idea: speed up queries using ***fractional cascading***
- every eighth key of array i is replicated in array $i-1$ as a ***lookahead pointer***
  - sorted order of array maintained
- every fourth element of array i is a ***duplicate lookahead pointer***, points to the nearest left and right lookaheads

Lookaheads

- idea: speed up queries using ***fractional cascading***
- every eighth key of array i is replicated in array $i - 1$ as a ***lookahead pointer***
    - sorted order of array maintained
- every fourth element of array i is a ***duplicate lookahead pointer***, points to the nearest left and right lookaheads
- queries: sequentially scan through arrays, using lookahead pointers to determine upper/lower bounds

Lookaheads

0	1	3	6	7	12	17	18	23	24	26	31	32	34	36	37	38	43	44	46	47	51	53	54	57	61	63	68	69	70	72	77

8	16	23	29	32	34	42	46	50	60	67	70	73	79	80	84

48	50	57	62	66	70	78	86

46	52	56	63

1	8

31

Lookaheads

0	1	3	6	7	12	17	**18**	23	24	26	31	32	34	36	**37**	38	43	44	46	47	51	53	**54**	57	61	63	68	69	70	72	**77**

8	16	23	29	32	34	42	46	50	60	67	70	73	79	80	84

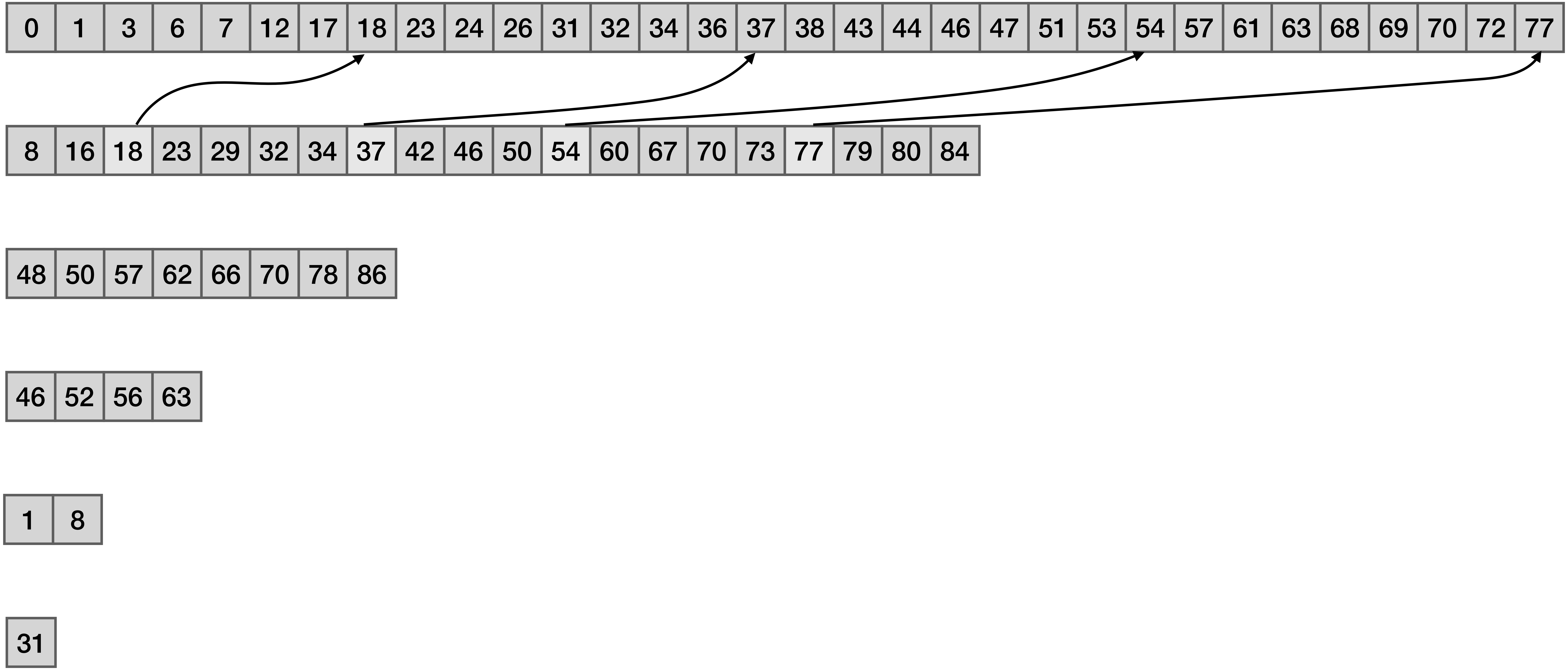
48	50	57	62	66	70	78	86

46	52	56	63

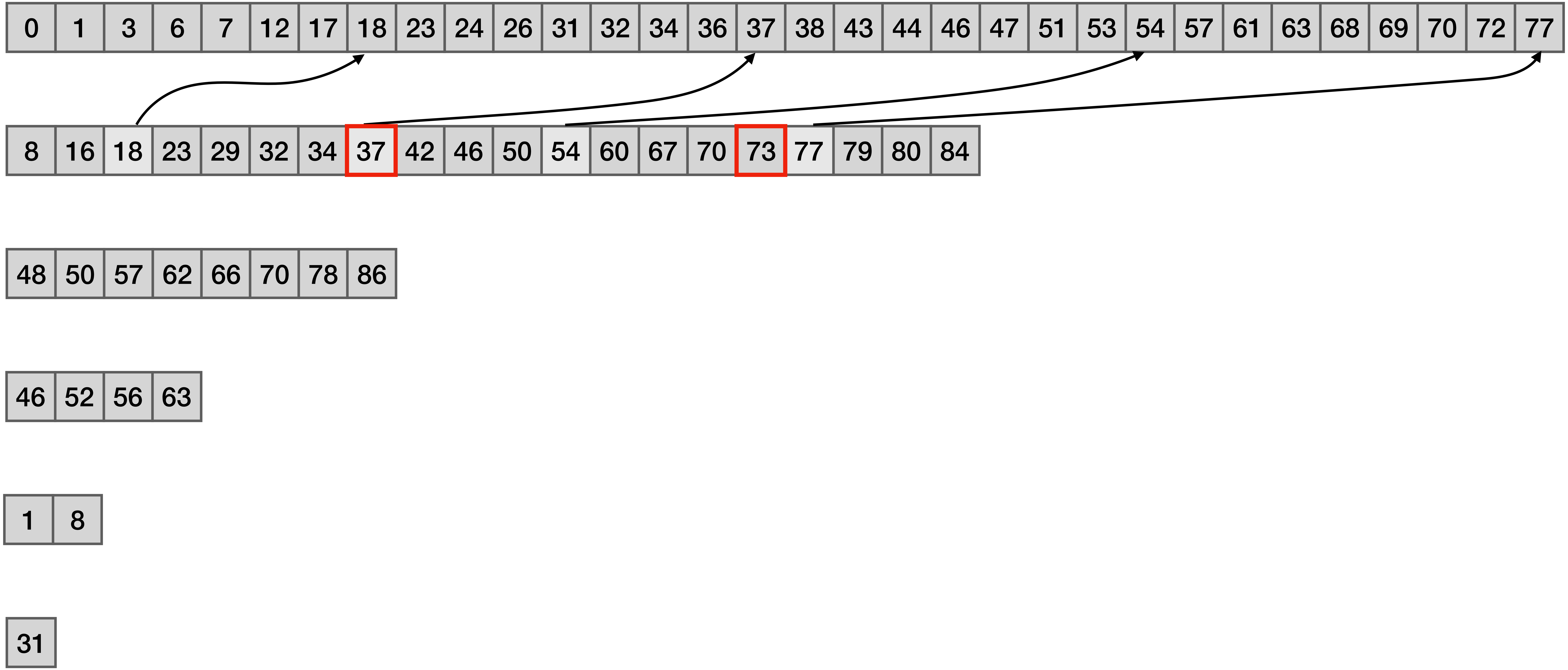
1	8

31

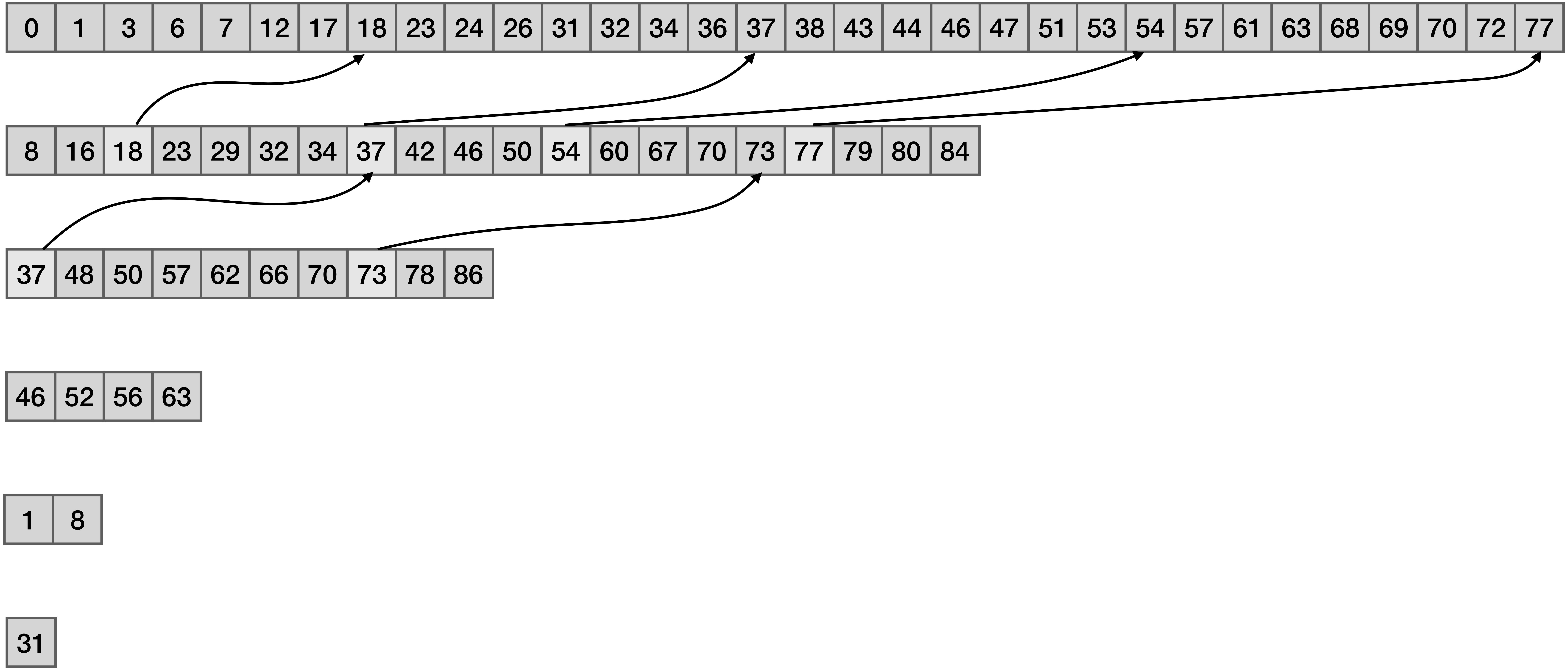
Lookaheads


0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 46 47 51 53 54 57 61 63 68 69 70 72 77
8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84
48 50 57 62 66 70 78 86
46 52 56 63
1 8
31

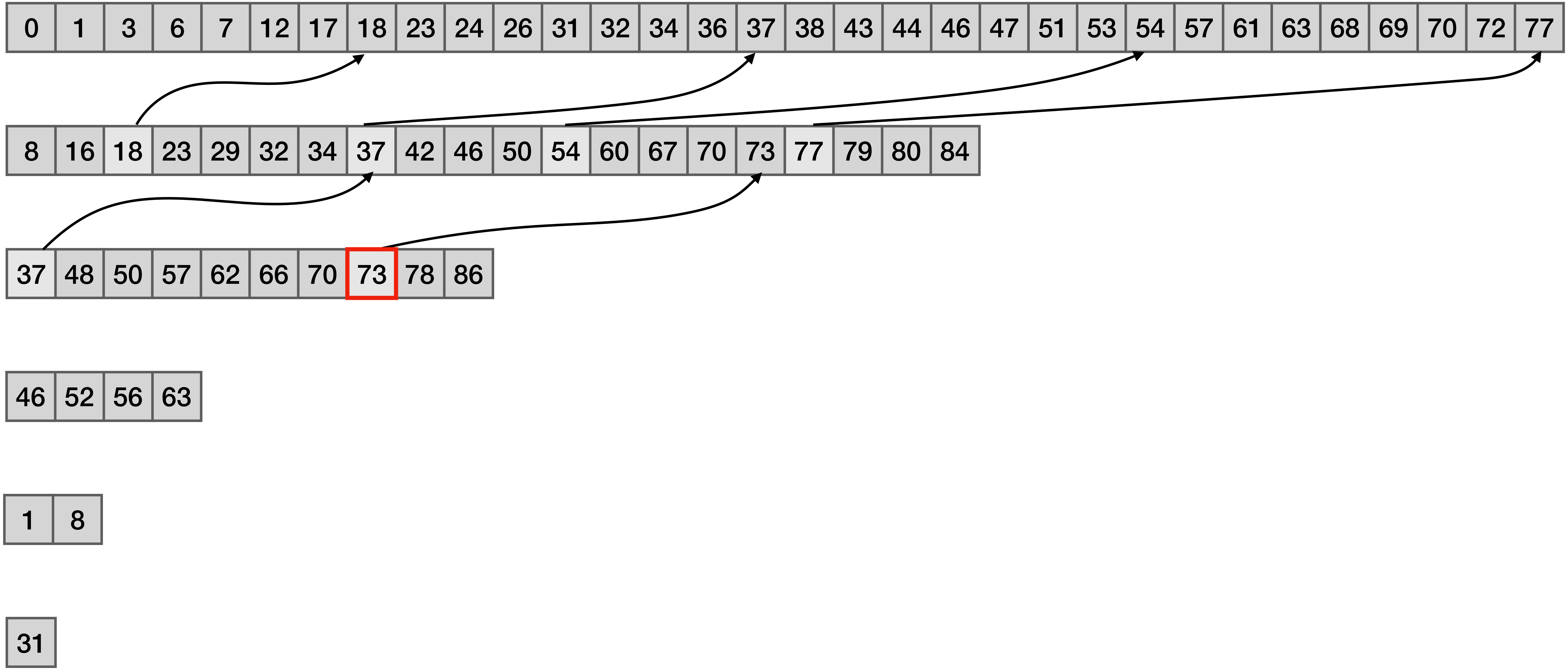
Lookaheads


0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 46 47 51 53 54 57 61 63 68 69 70 72 77
8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84
48 50 57 62 66 70 78 86
46 52 56 63
1 8
31

Lookaheads

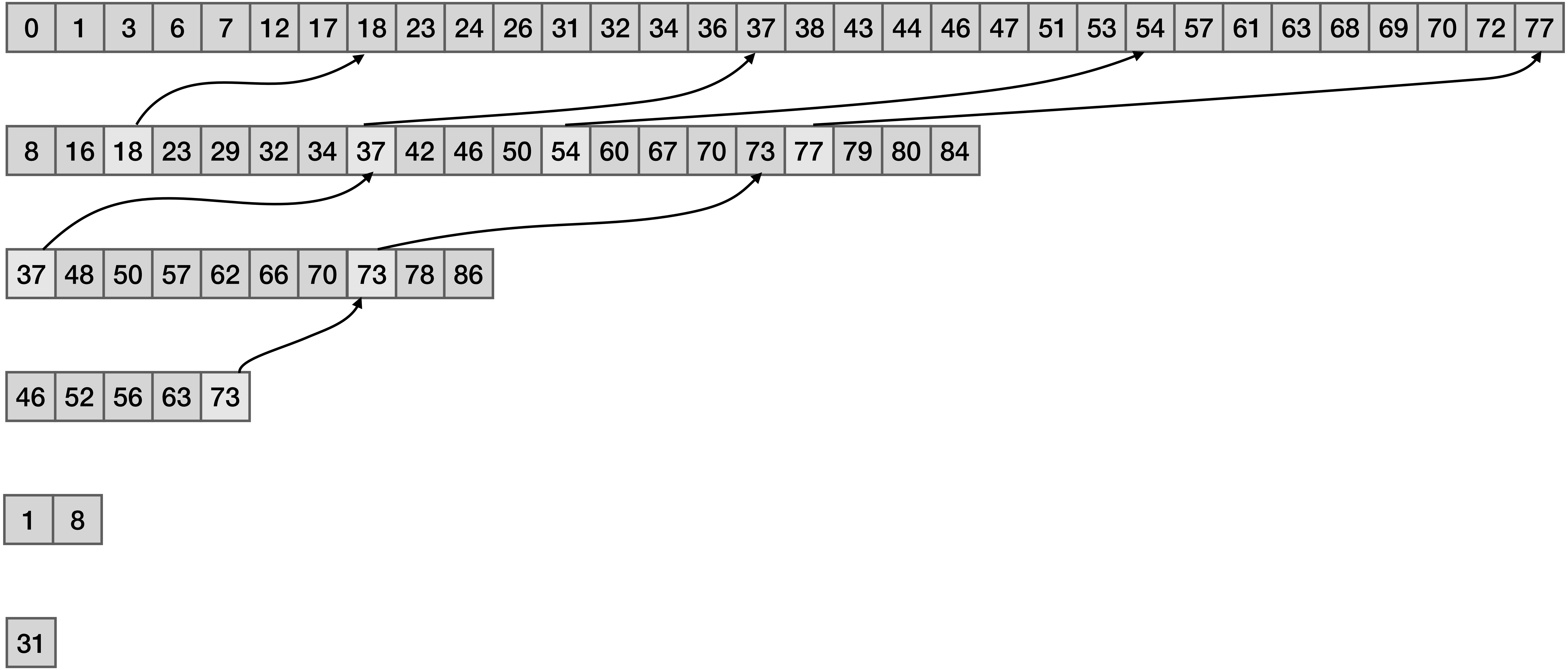

0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 46 47 51 53 54 57 61 63 68 69 70 72 77
8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84
37 48 50 57 62 66 70 73 78 86
46 52 56 63
1 8
31

Lookaheads

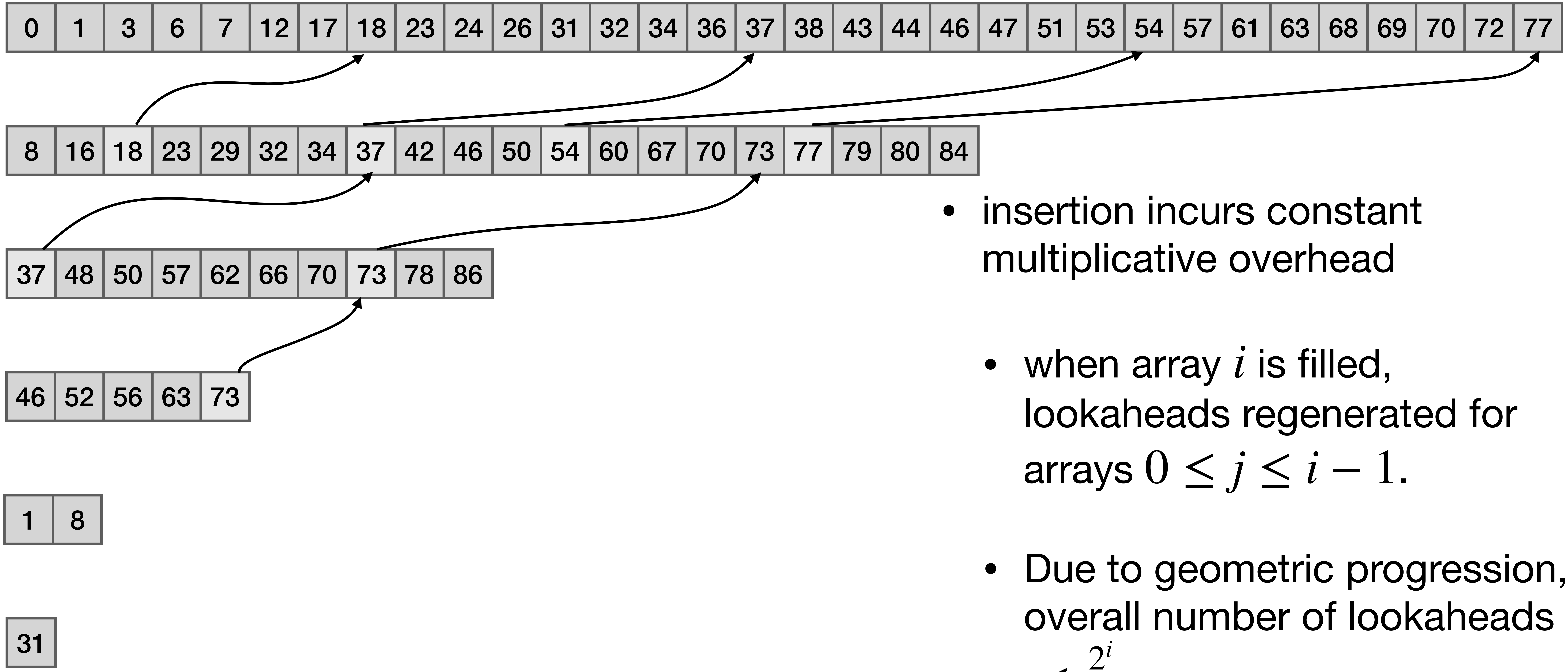


0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 46 47 51 53 54 57 61 63 68 69 70 72 77
8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84
37 48 50 57 62 66 70 73 78 86
46 52 56 63
1 8
31

Lookaheads

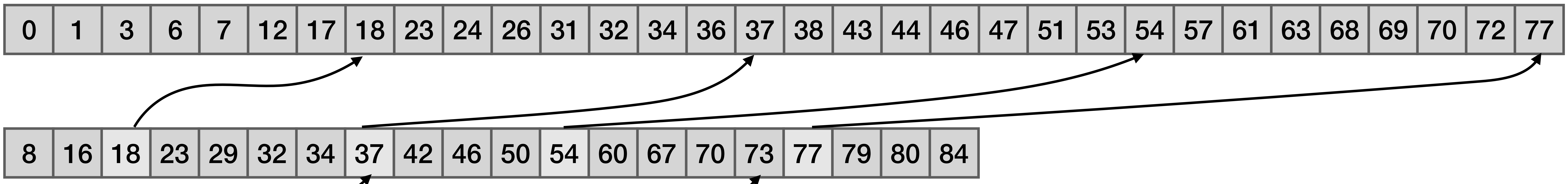

0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 46 47 51 53 54 57 61 63 68 69 70 72 77
8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84
37 48 50 57 62 66 70 73 78 86
46 52 56 63 73
1 8
31

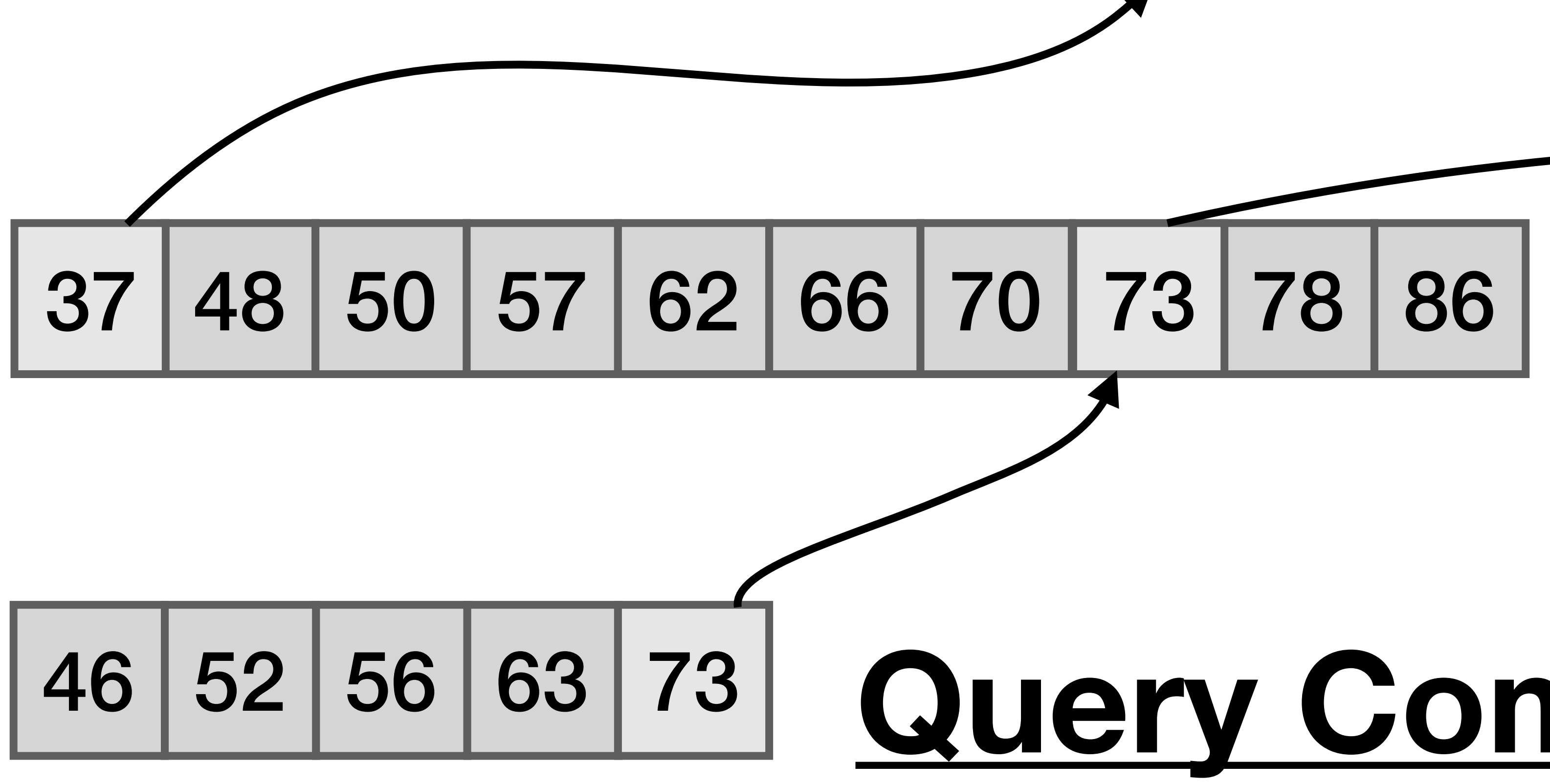
Lookaheads

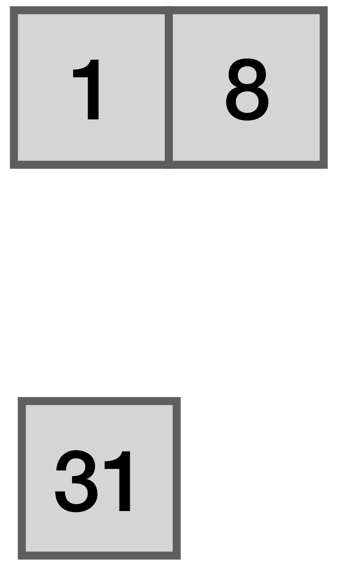


- insertion incurs constant multiplicative overhead
  - when array i is filled, lookaheads regenerated for arrays $0 \leq j \leq i - 1$.
  - Due to geometric progression, overall number of lookaheads $< \frac{2^i}{4}$

Lookaheads





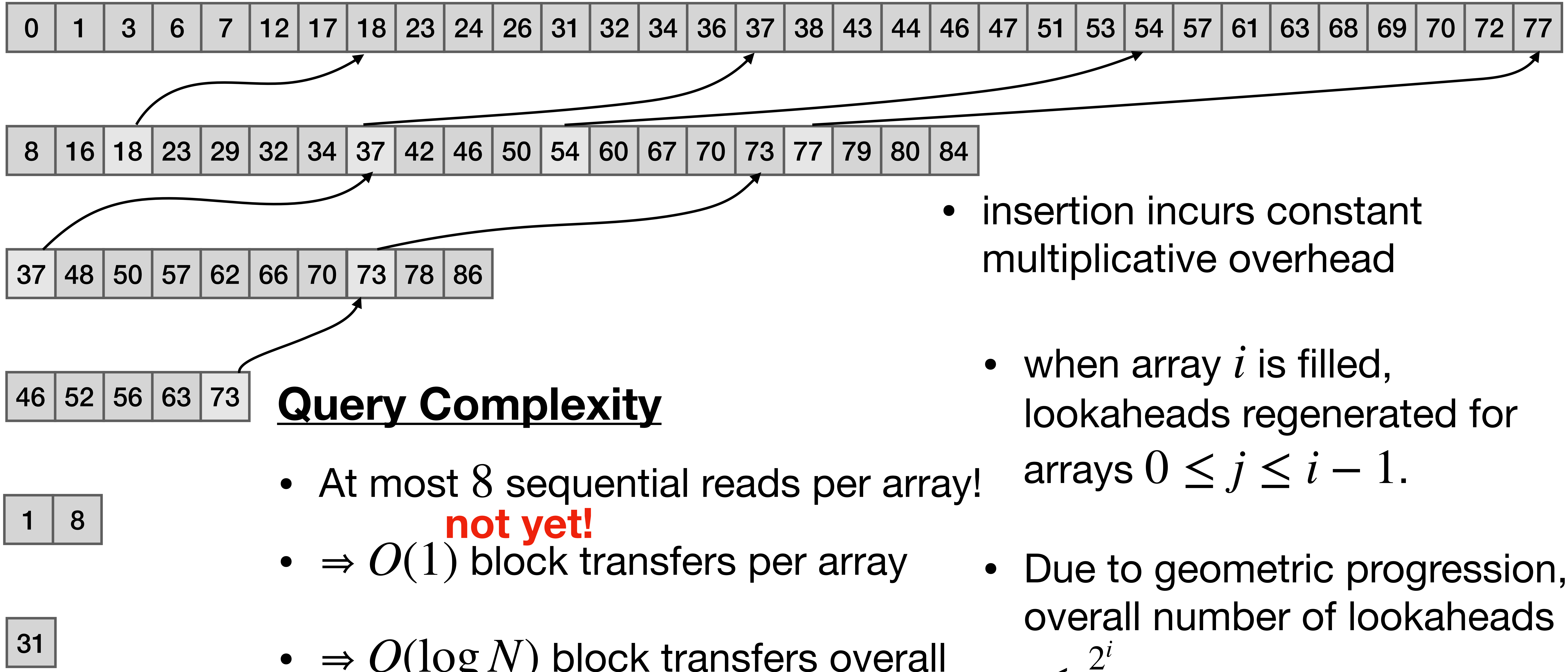


- insertion incurs constant multiplicative overhead
- when array i is filled, lookaheads regenerated for arrays $0 \leq j \leq i-1$.
- Due to geometric progression, overall number of lookaheads $< \frac{2^i}{4}$

<u>Query Complexity</u>

- At most 8 sequential reads per array!
- $\Rightarrow O(1)$ block transfers per array
- $\Rightarrow O(\log N)$ block transfers overall

Lookaheads



- insertion incurs constant multiplicative overhead

Query Complexity

- At most 8 sequential reads per array!
- $\Rightarrow O(1)$ block transfers per array **not yet!**
- $\Rightarrow O(\log N)$ block transfers overall

- when array i is filled, lookaheads regenerated for arrays $0 \leq j \leq i - 1$.
- Due to geometric progression, overall number of lookaheads $< \frac{2^i}{4}$

Lookaheads

1	2	3	4	5	6	7	8	16	32	64	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

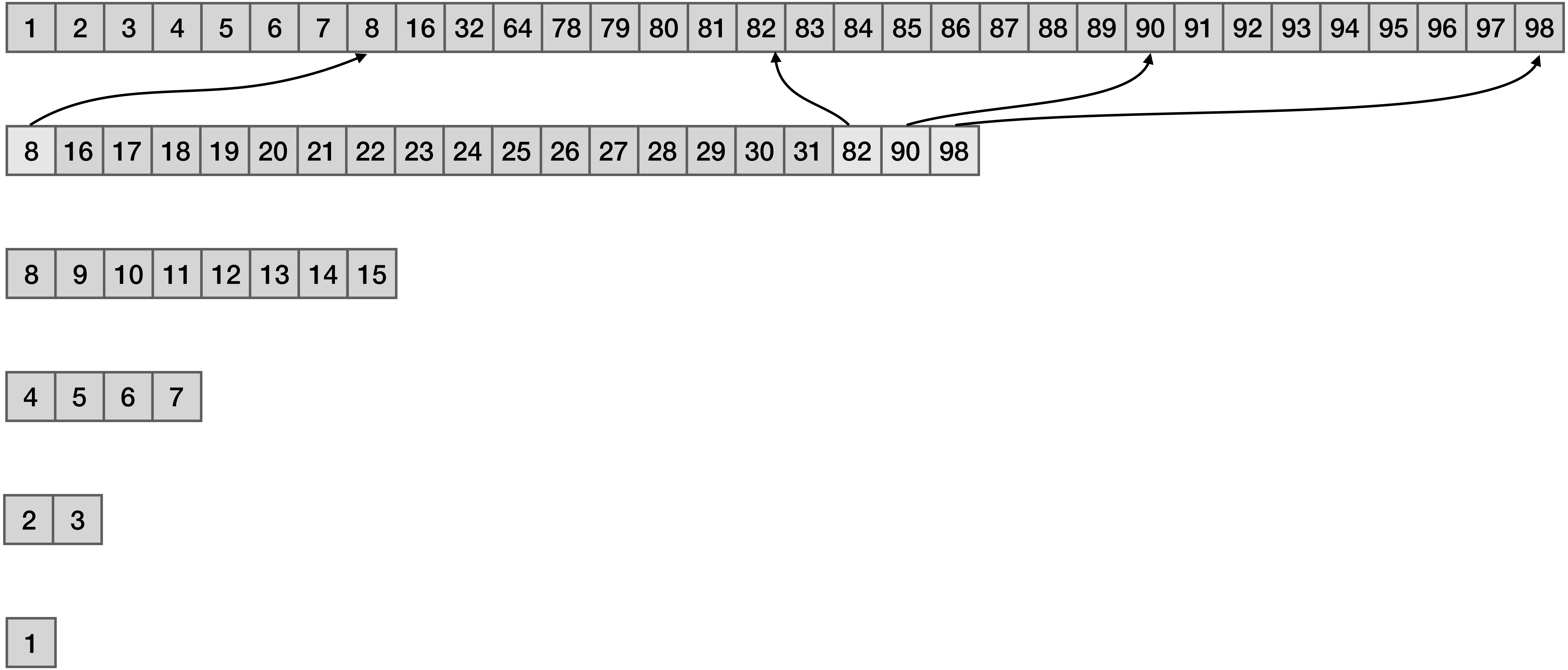
8	9	10	11	12	13	14	15

4	5	6	7

2	3

1

Lookaheads

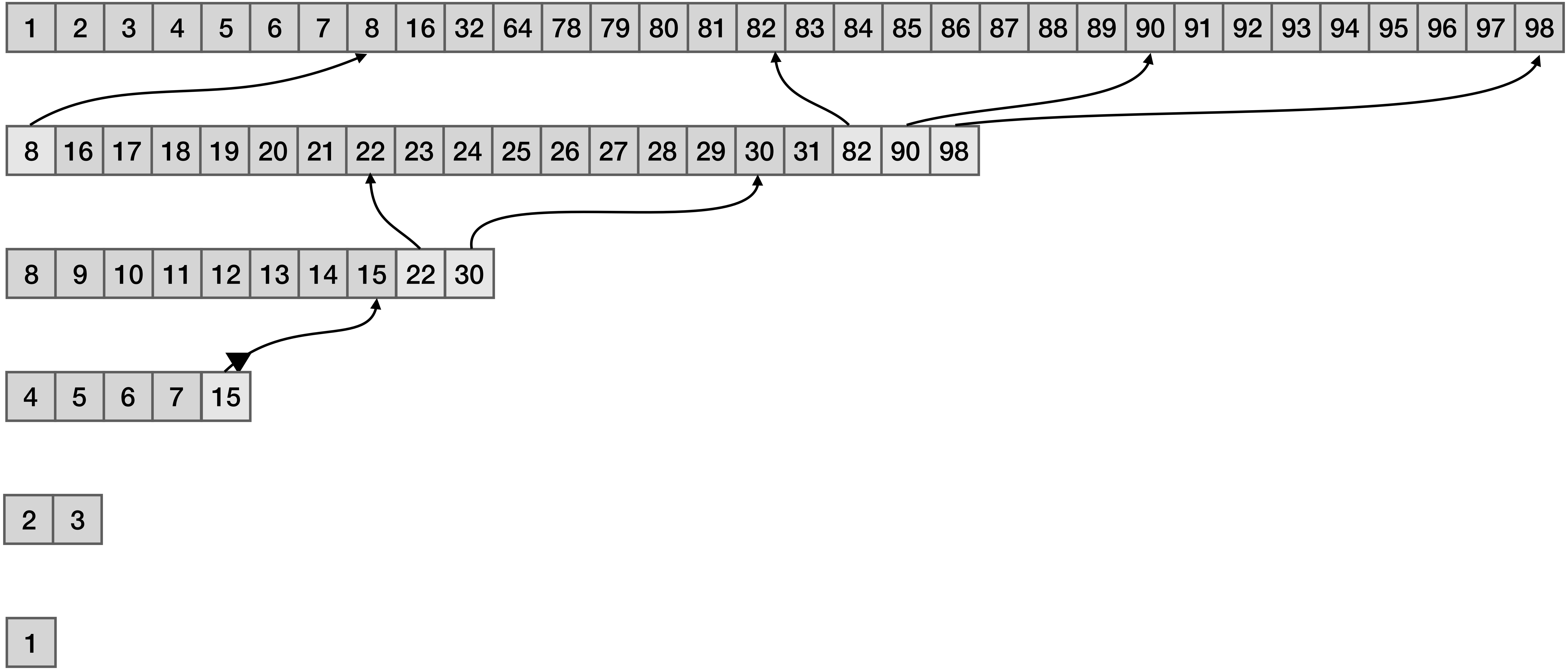

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98
8 9 10 11 12 13 14 15
4 5 6 7
2 3
1

query **81**

Lookaheads

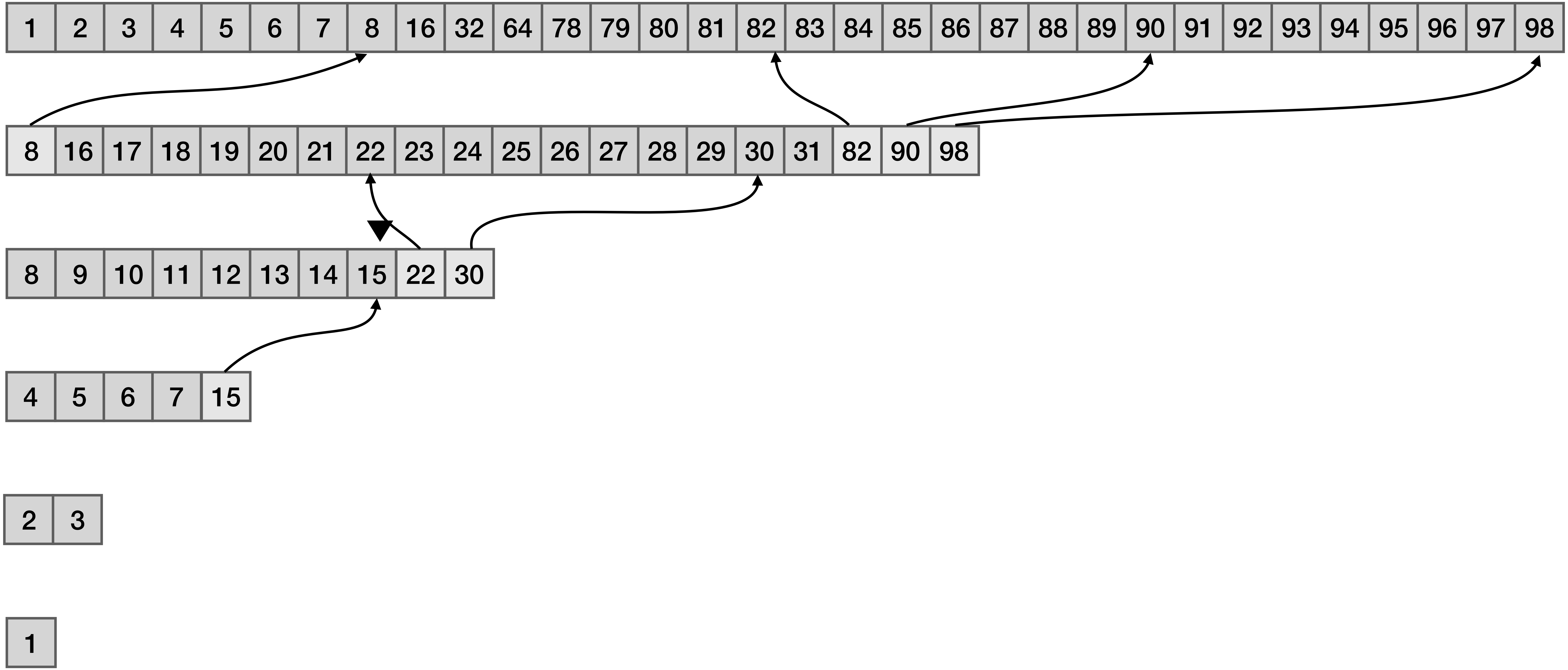
Lookaheads

query **81**



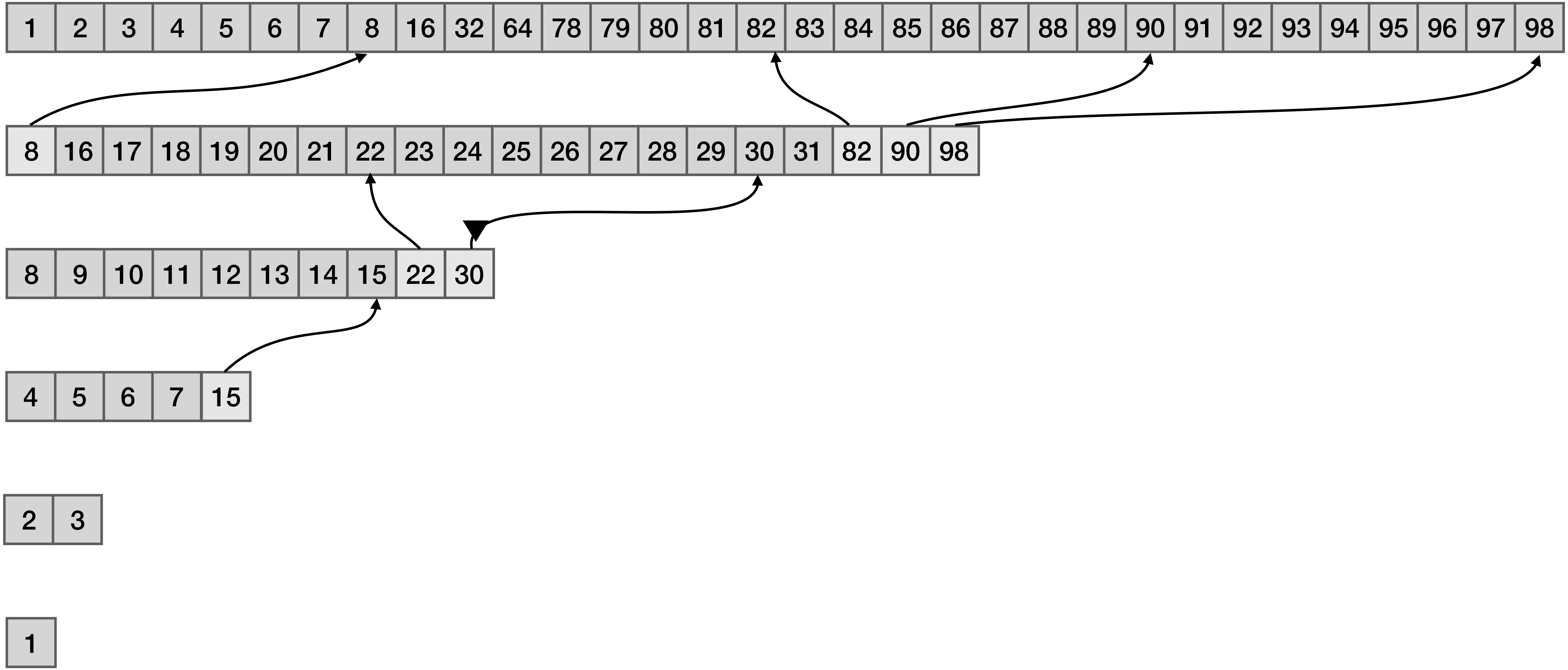
query **81**

Lookaheads



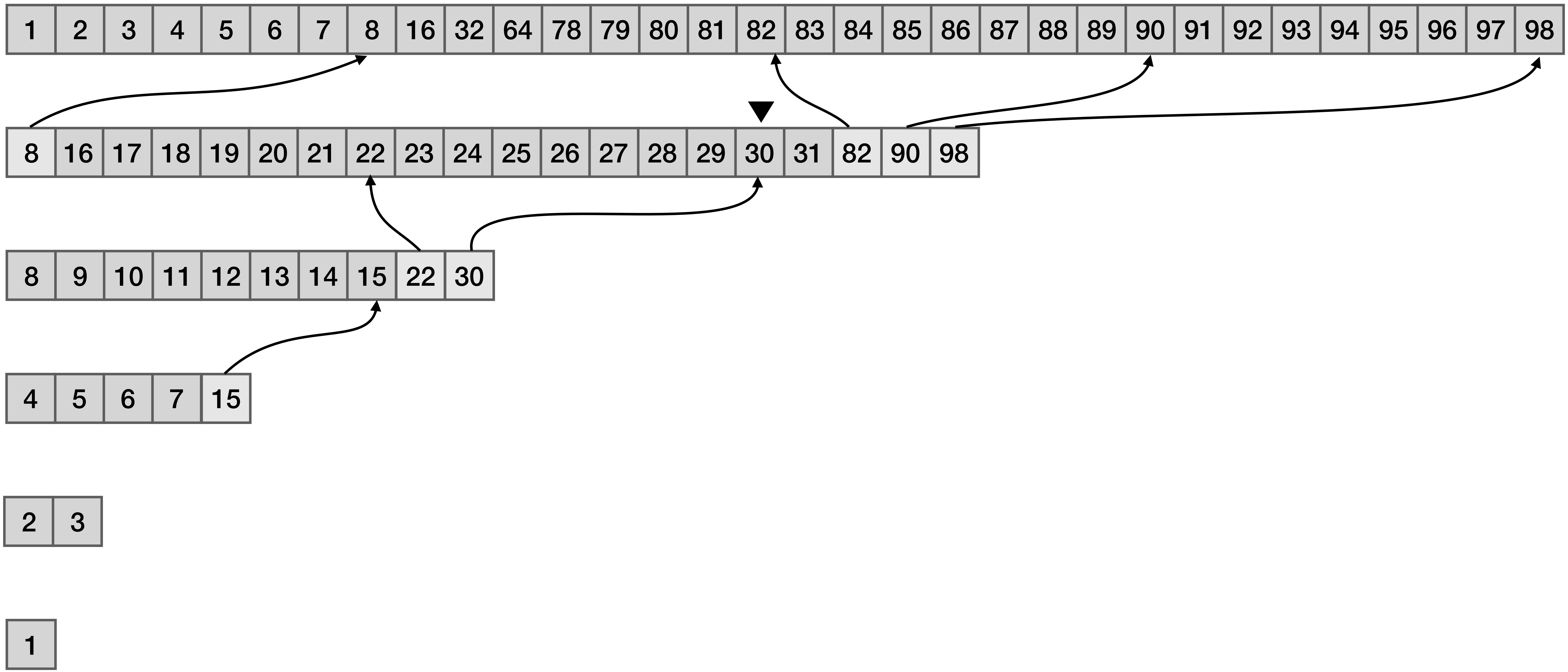
query **81**

Lookaheads



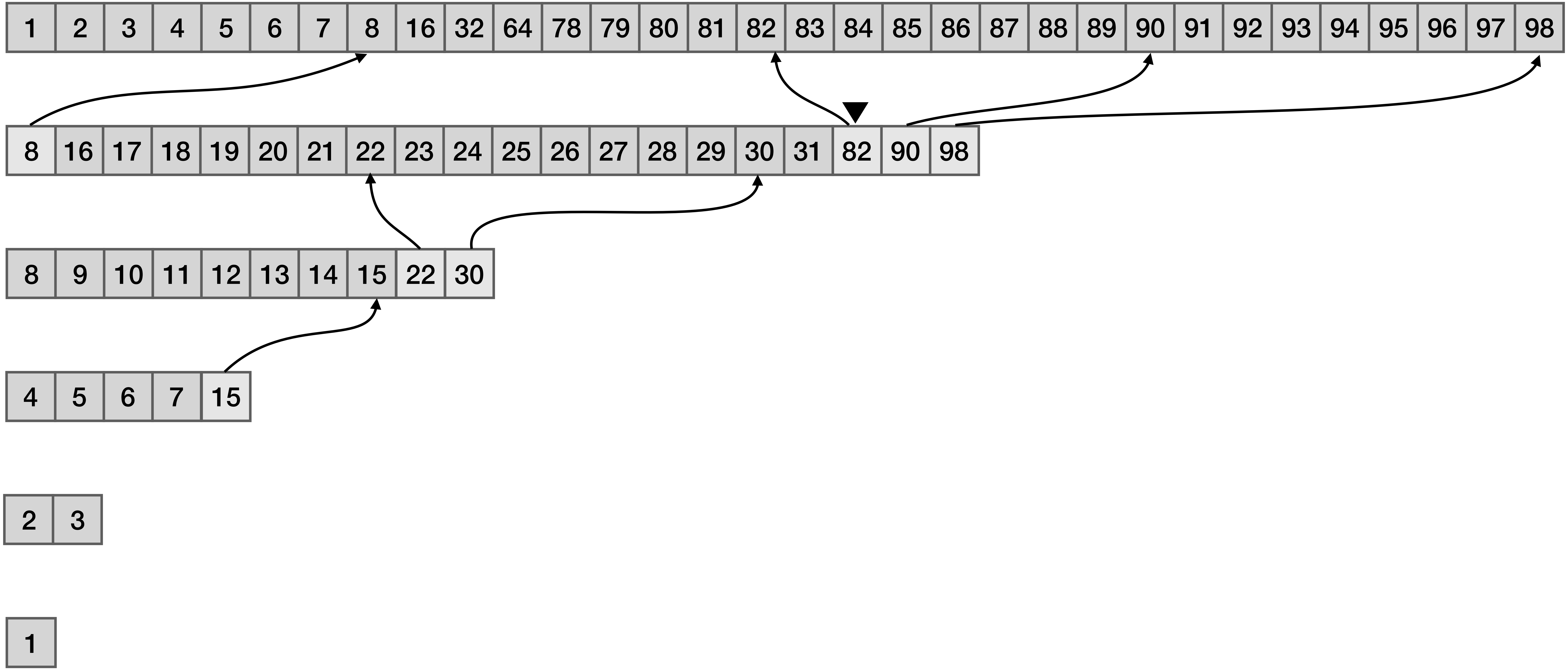
query **81**

Lookaheads



query **81**

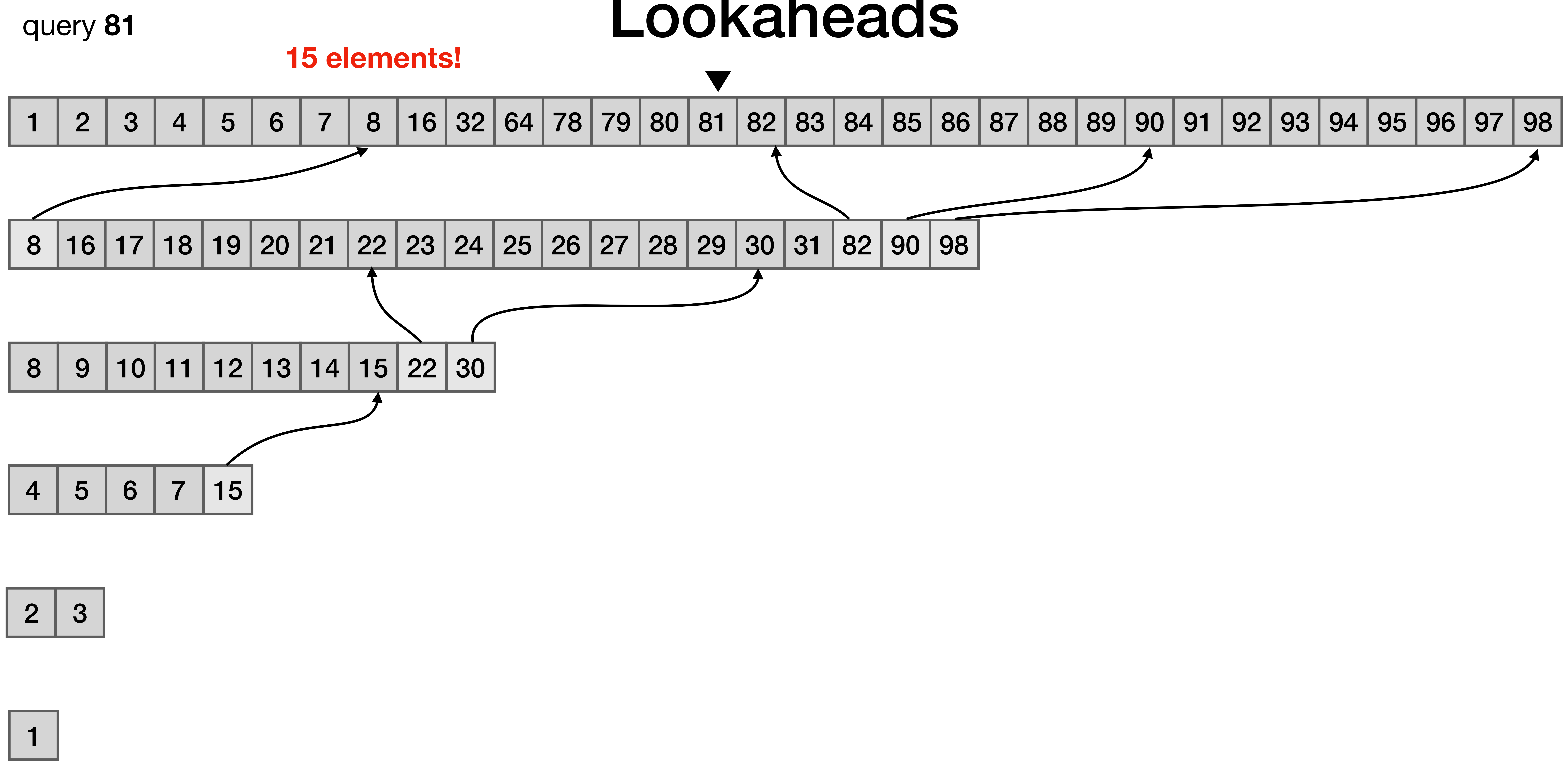
Lookaheads



Lookaheads

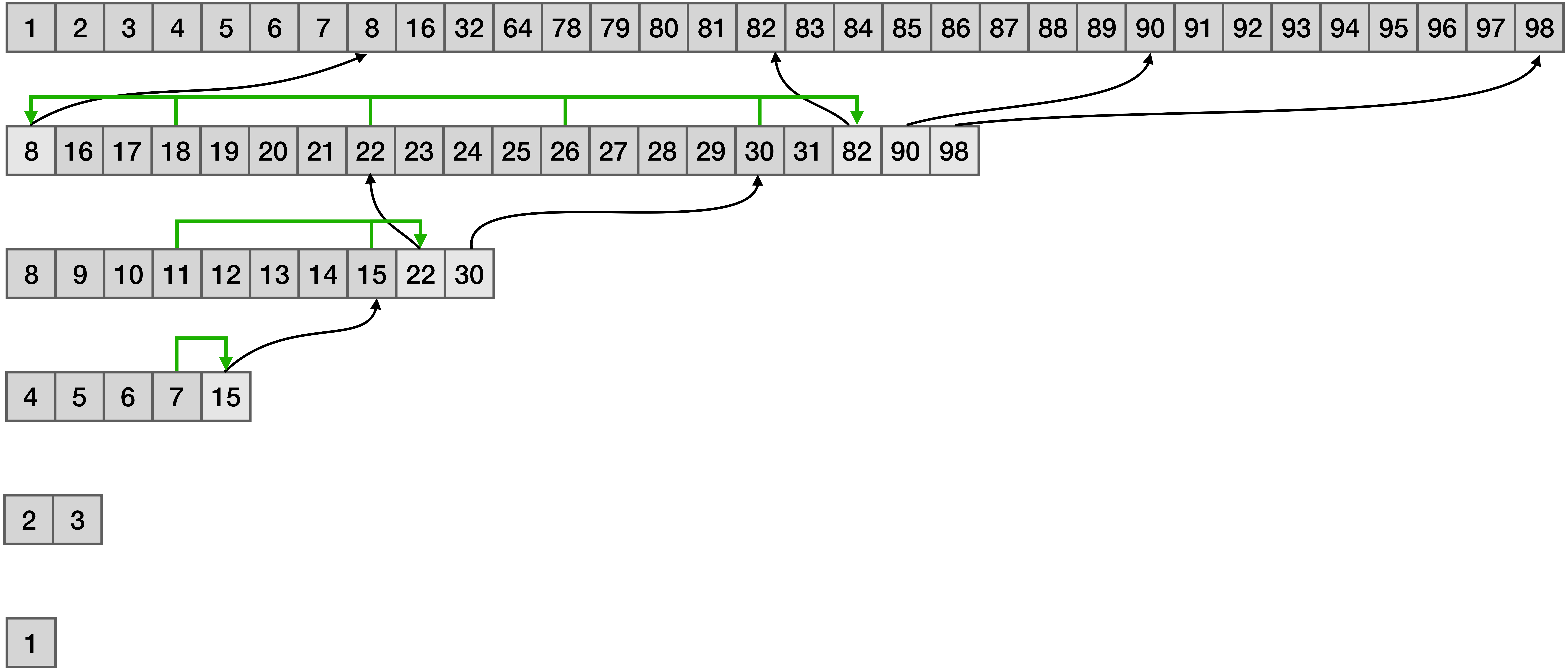

query **81**
1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98
8 9 10 11 12 13 14 15 22 30
4 5 6 7 15
2 3
1

Lookaheads

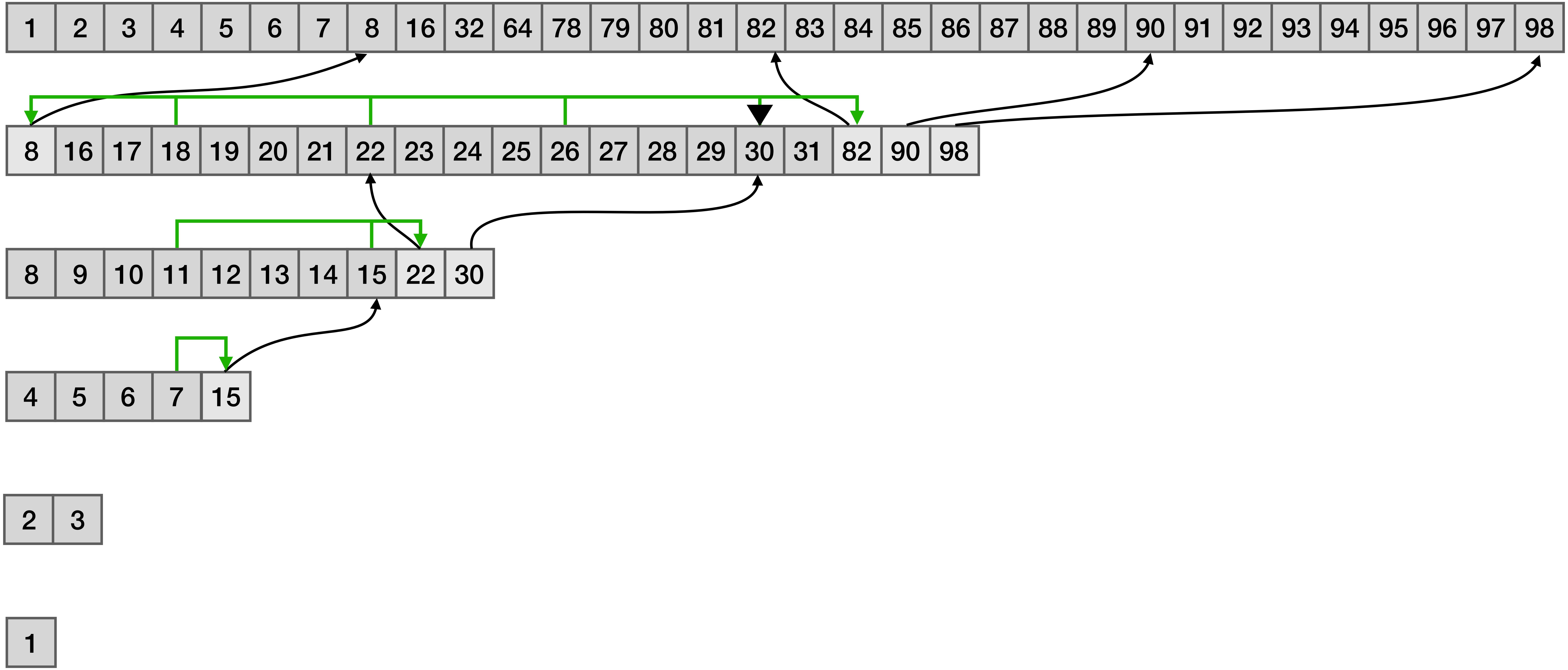


query **81**
15 elements!
1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98
8 9 10 11 12 13 14 15 22 30
4 5 6 7 15
2 3
1

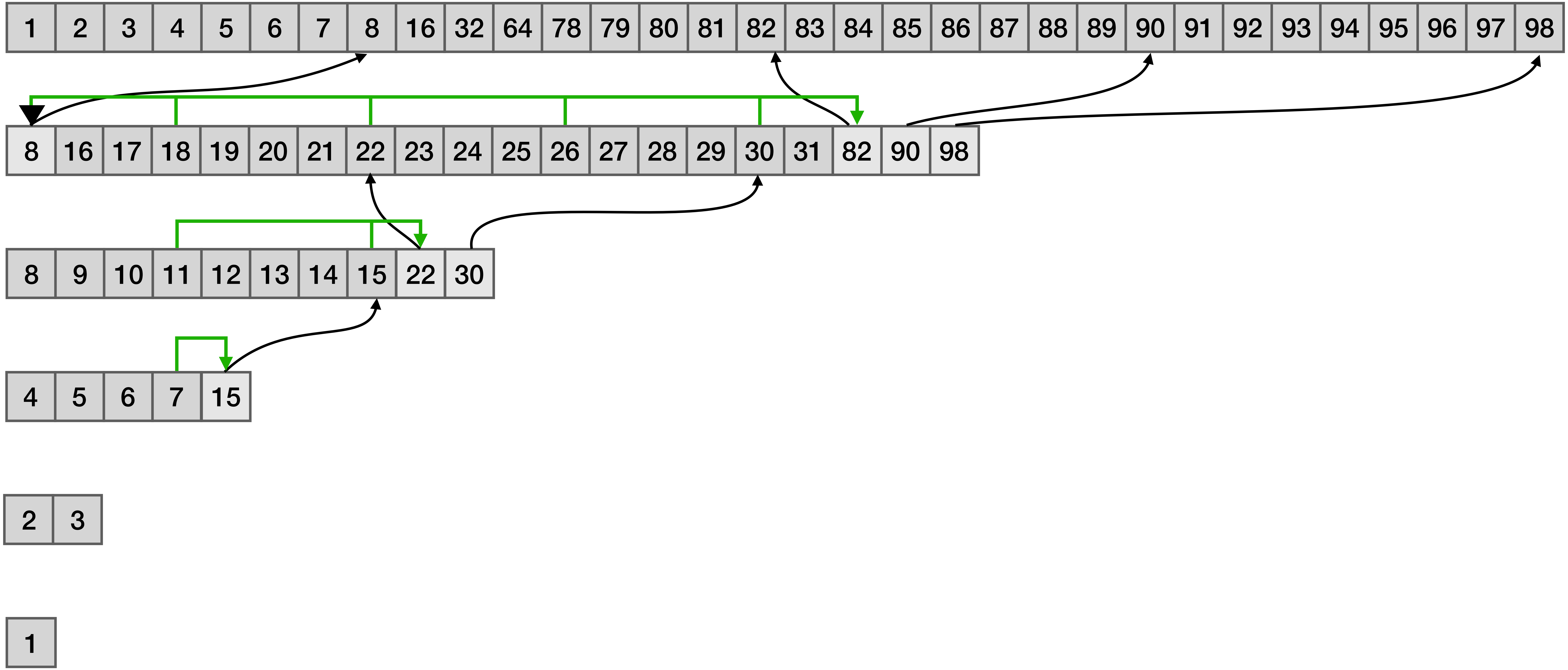
Duplicate Lookaheads



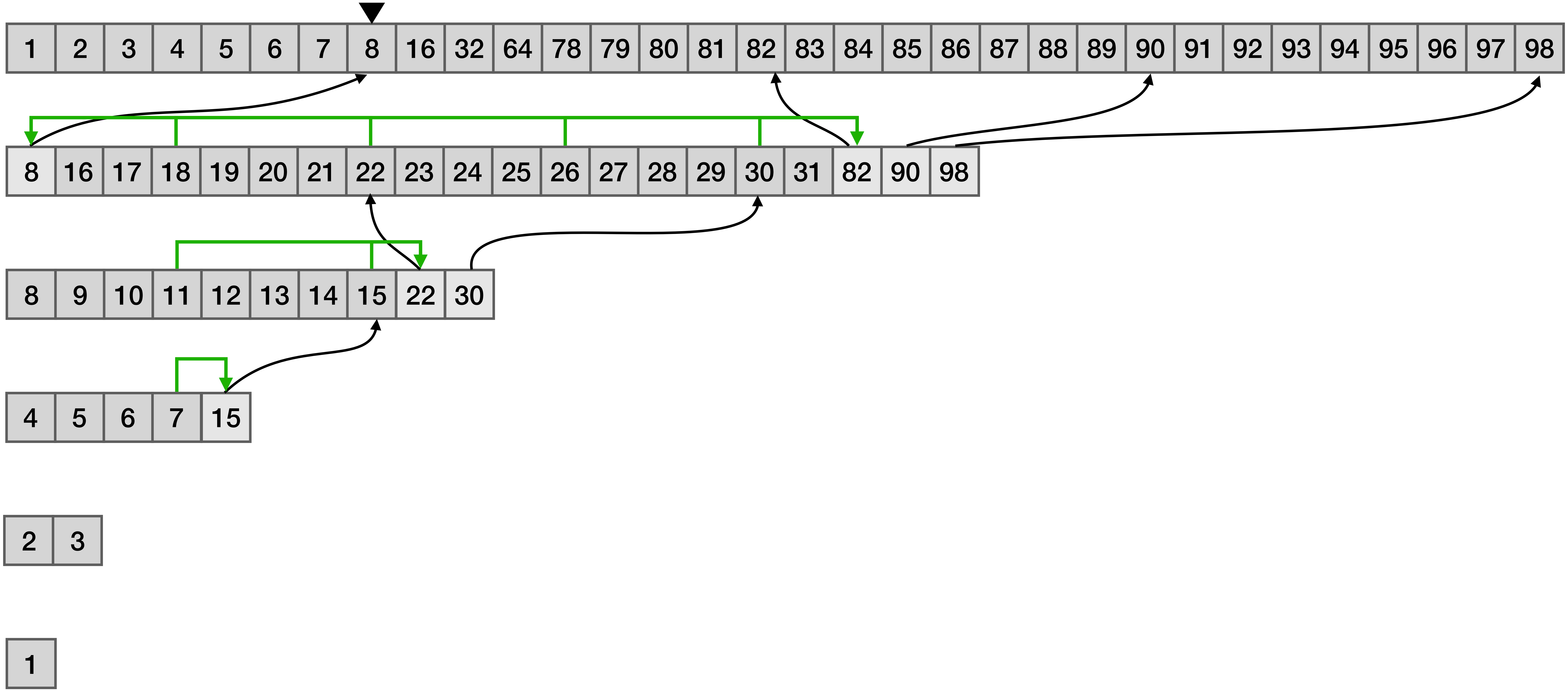
Duplicate Lookaheads



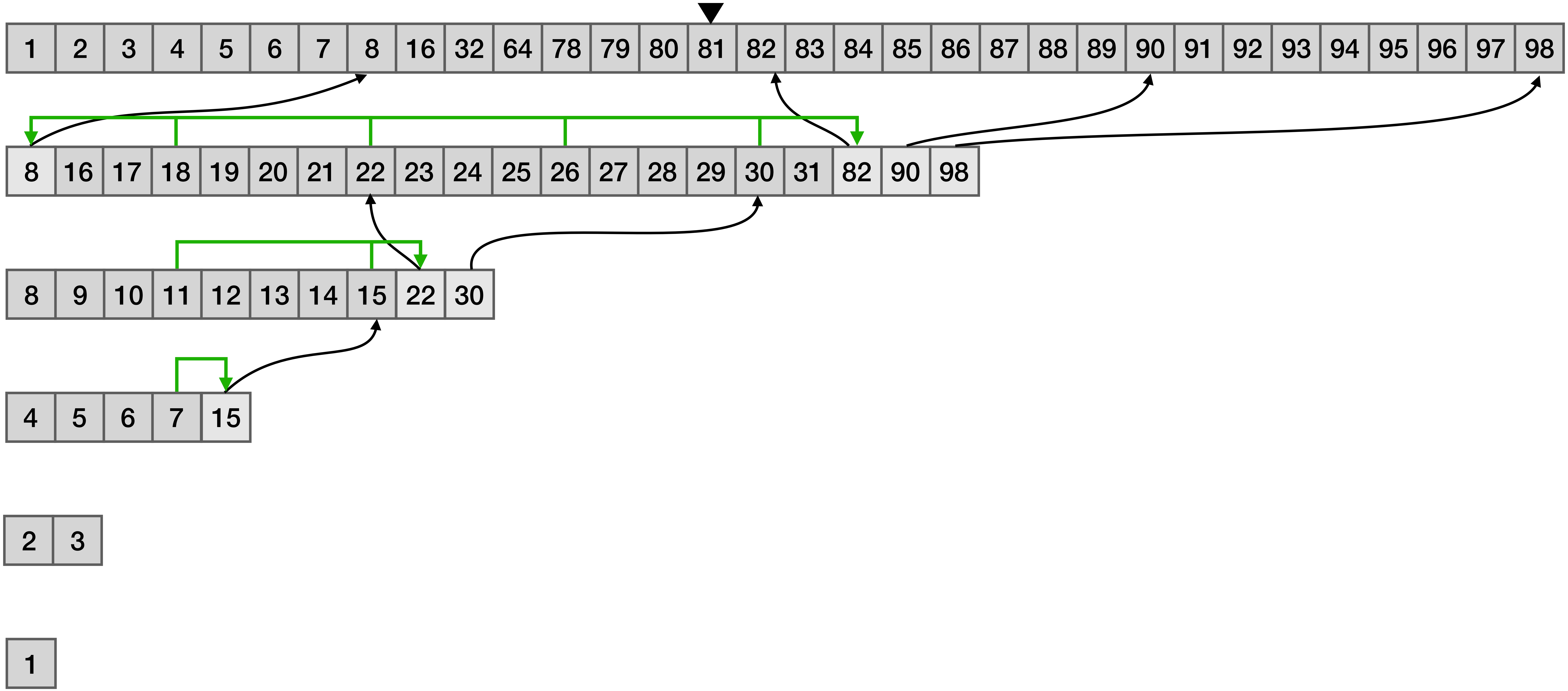
Duplicate Lookaheads



Duplicate Lookaheads



Duplicate Lookaheads



Thanks for listening!

References

[1] Michael A. Bender et al. "Cache-oblivious streaming B-trees". In: *SPAA*. 2007.

[2] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *TOCS*. 2006.

[3] Sanjay Ghemawat and Jeffrey Dean. *LevelDB*. https://github.com/google/leveldb/. 2011–2019.

[4] Chen Luo and Michael J. Carey. "LSM-based storage techniques: a survey". In: *The VLDB Journal* (2018), pp. 1–26.

[5] Patrick E. O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33 (1996), pp. 351–385.

[6] Russell Sears and Raghu Ramakrishnan. "bLSM: a general purpose log structured merge tree". In: *SIGMOD Conference*. 2012.