Learned indexes, the PGM-index, and the coding challenge

Algorithm Engineering A.Y. 2020/21

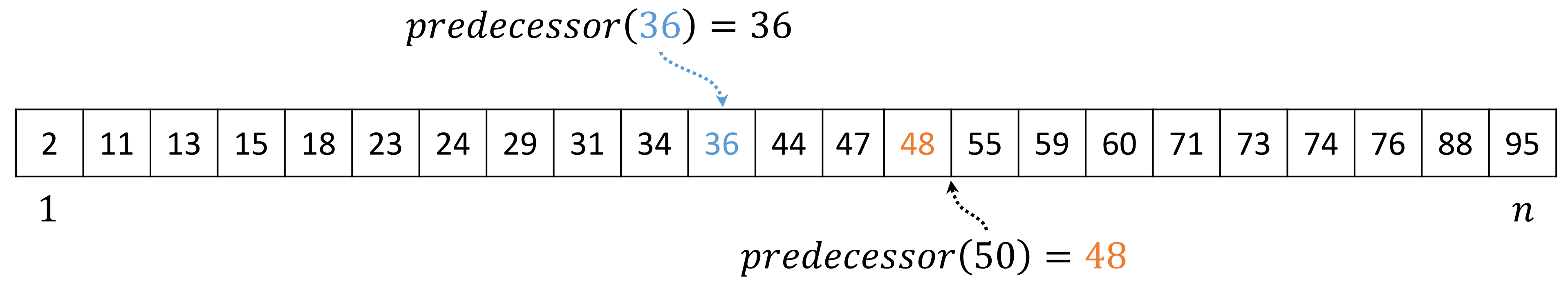
Università di Pisa

Giorgio Vinciguerra

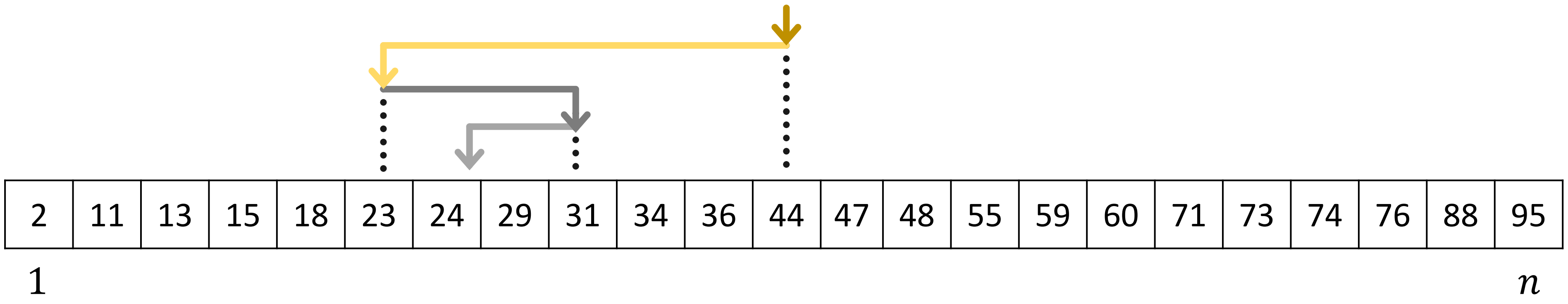
PhD student & Teaching assistant

The predecessor search problem

- Given n sorted input keys (e.g. integers), implement $predecessor(x)$ = "largest key $\leq x$"
- Range queries in DBs, lists intersection (conjunctive queries in search engines), IP routing…
- Harder than the dictionary problem: if you need to support only exact searches just use Cuckoo hashing (§8.6 of the notes)



The basic solution, binary search

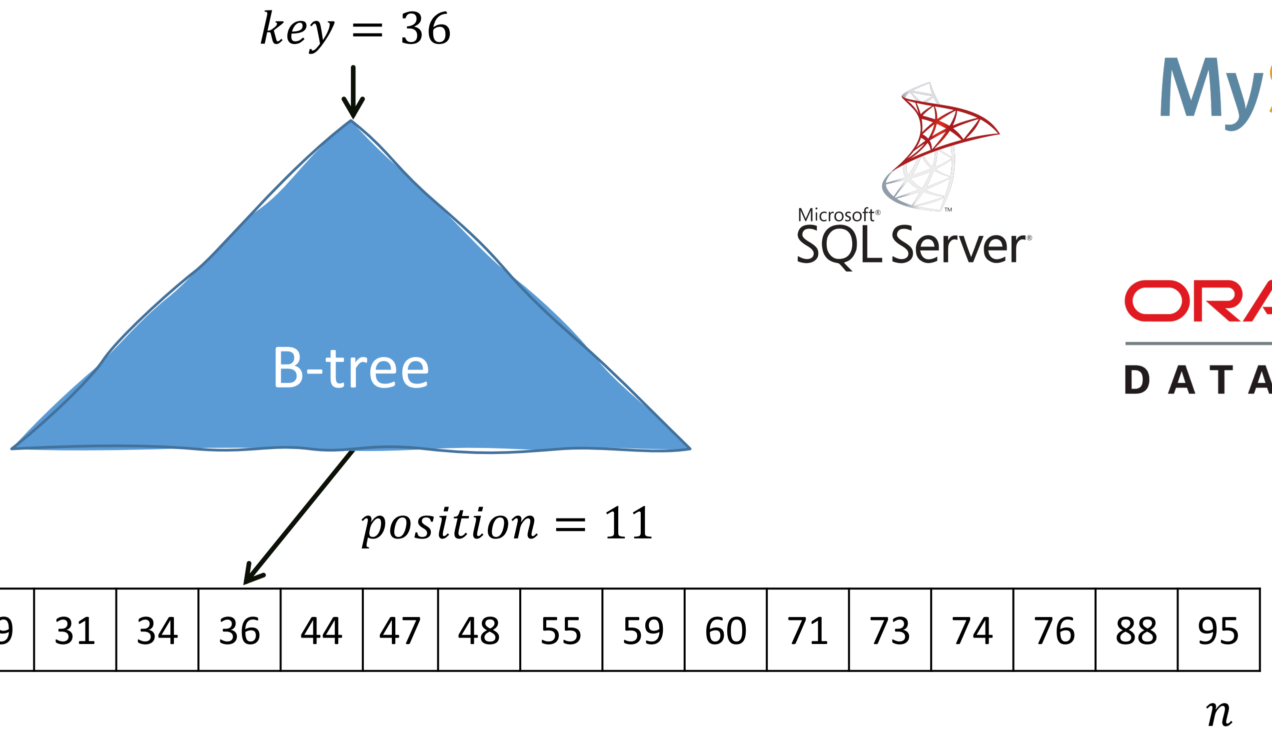


- Works well if the keys fits in the CPU cache
- Incurs costly cache misses if keys are in memory (≈ 100 ns per miss)
- Incurs very costly disk transfers if keys are on disk (≈ 16 µs per random I/O on SSDs)

Indexes

“B-trees have become, de facto, a standard for file organization”

— Comer. *Ubiquitous B-tree*. ACM Computing Surveys. ’79

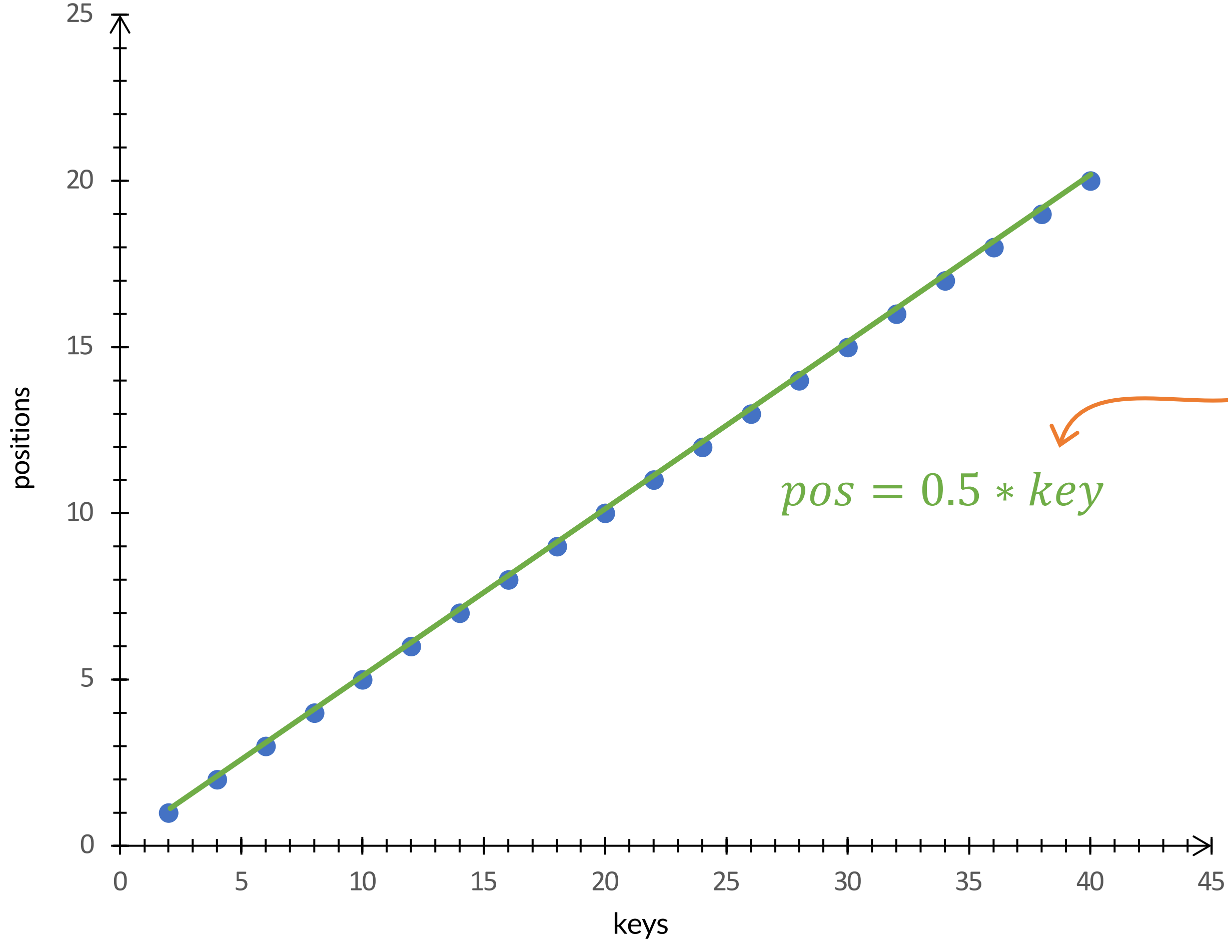


(values associated to keys are not shown)

A different look at the data

2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Map data to points
$(key, position)$

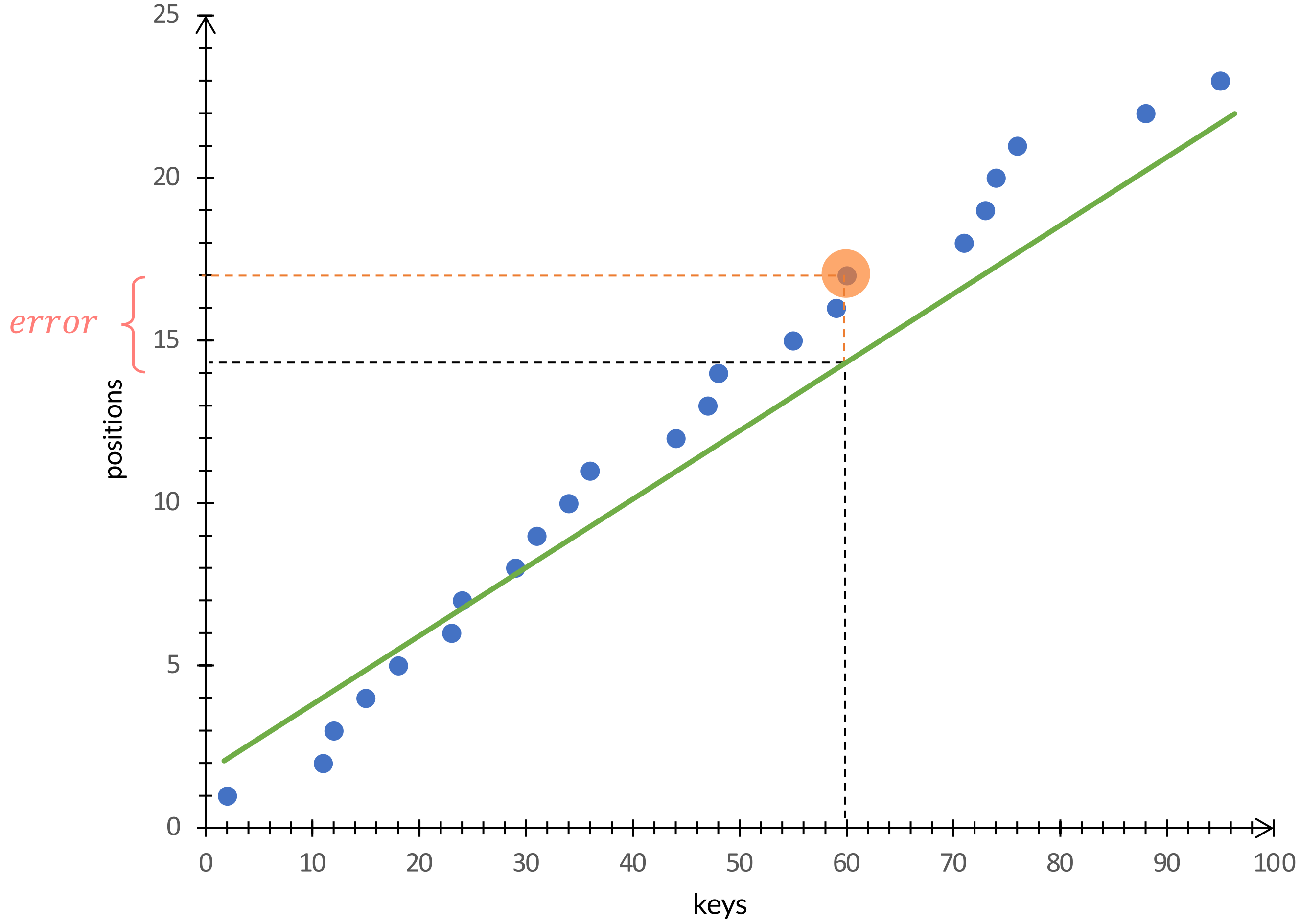


Indexes an arbitrary number of keys in $O(1)$ space and time

A different look at *realistic* data

error

2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23



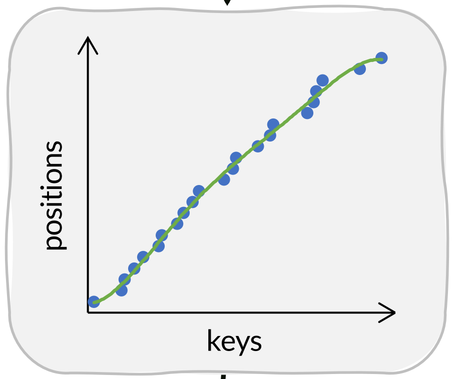
Learned indexes

Query latency = time to output a position + time to “fix the error” via binary search

How to strike a good balance between the model complexity and the query latency?

Model trained on a dataset of pairs (key, pos)
$\mathcal{D} = \{(2,1), (11,2), \ldots, (95, n)\}$

key



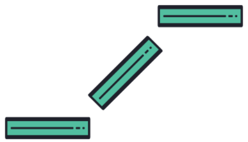
$position$ (approximate)

2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95

1 n

Binary search in
$[position - error, position + error]$

An optimal solution: the PGM-index

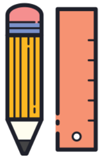


Opt. piecewise linear model

Fast to construct, captures non-linearities

Fixed model “error” ε

Control the size of the search range

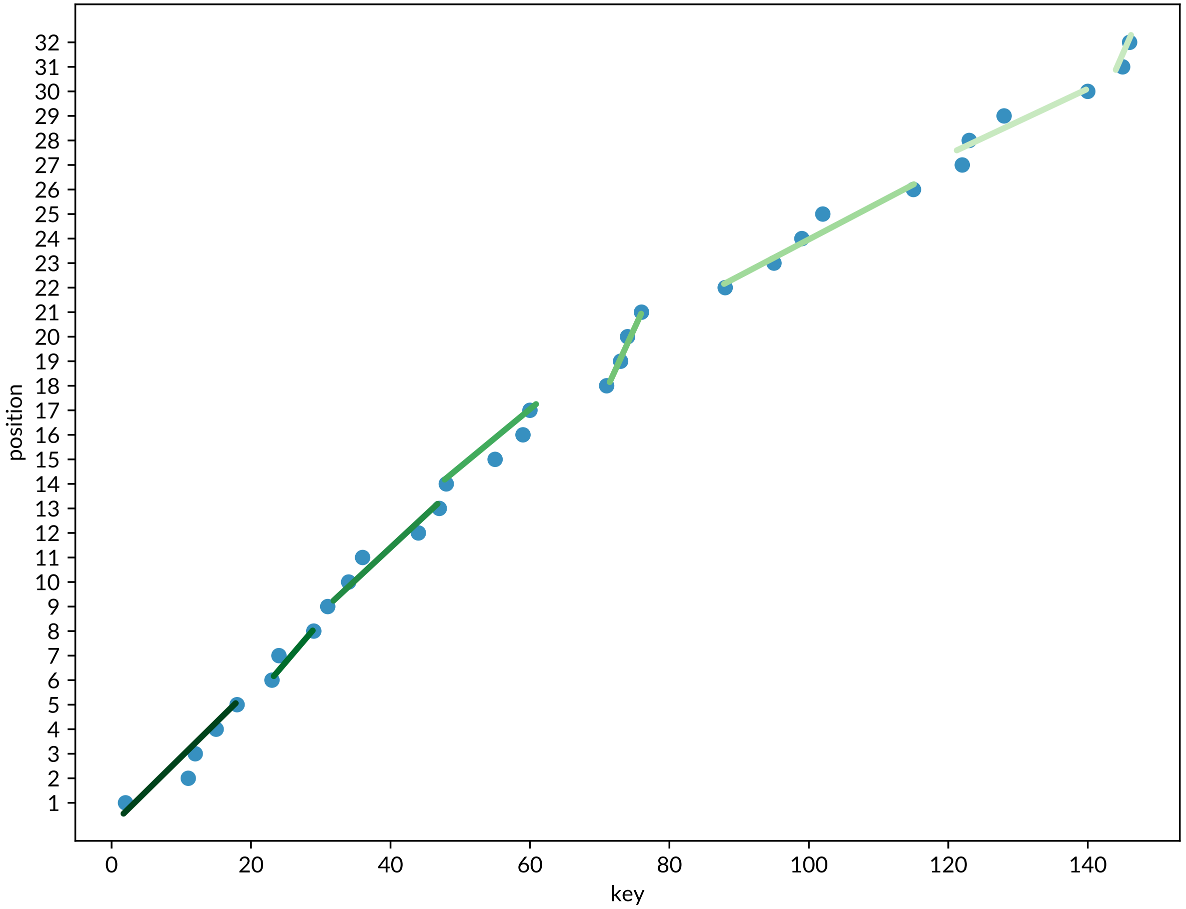


Recursive design

Adapt to the memory hierarchy

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

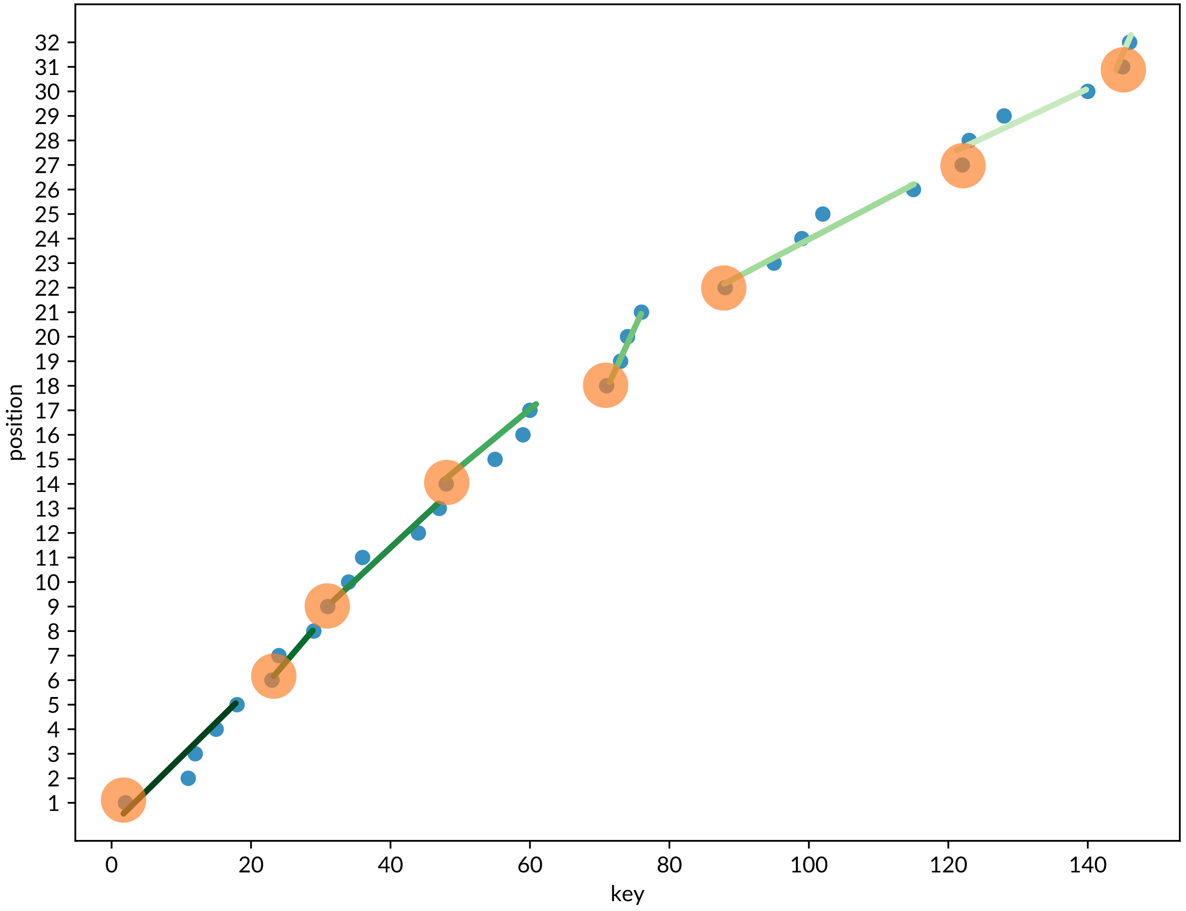


2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

1 $\quad\quad$ n

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time



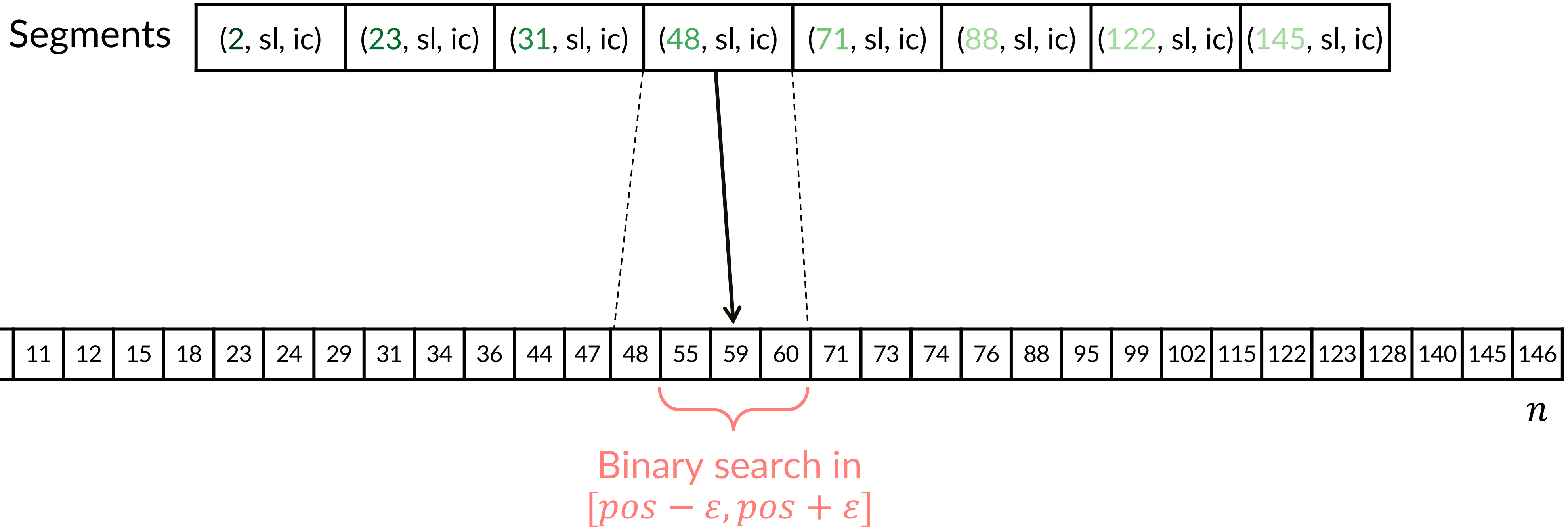
Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

1 ... n

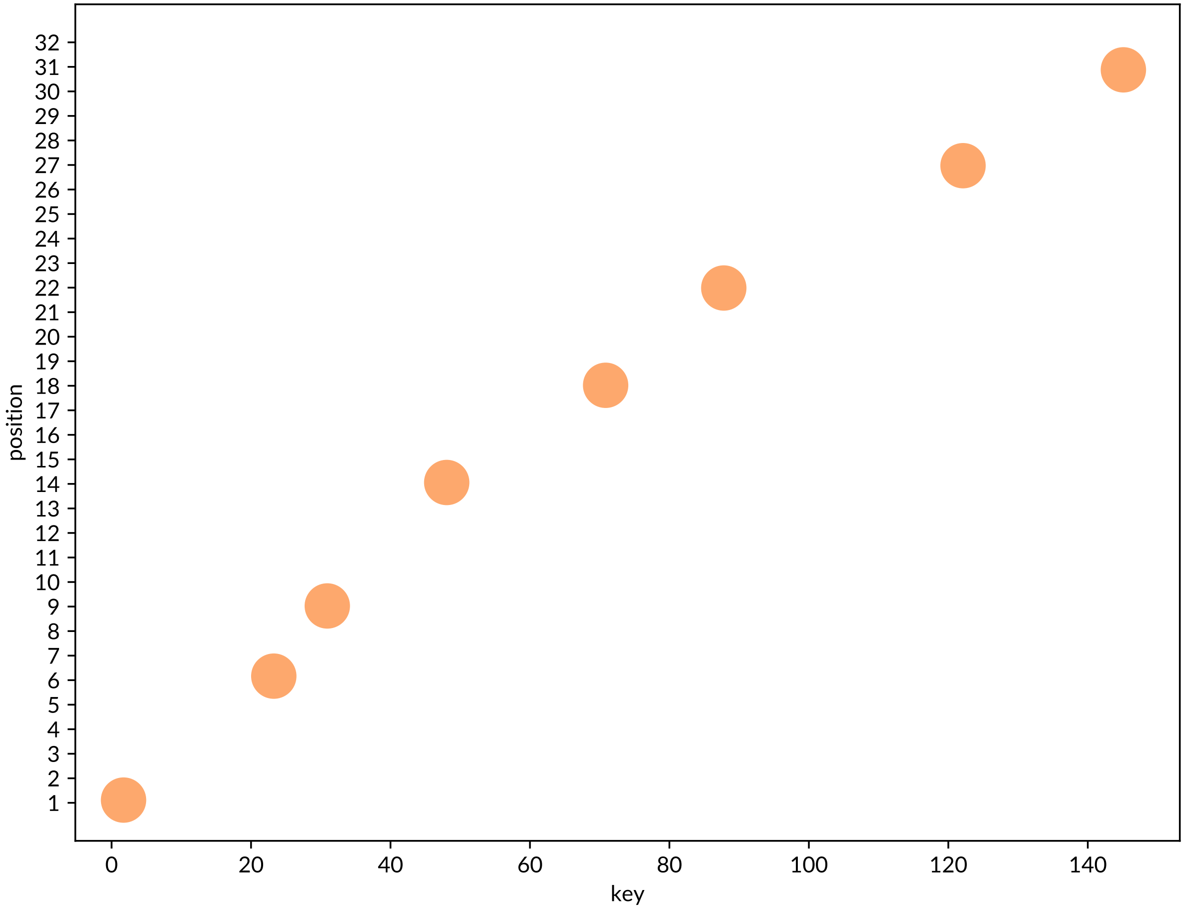
Partial memory layout of the PGM-index

Each segment indexes a variable and potentially large sequence of keys
while guaranteeing a search range size of $2\varepsilon + 1$



PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time



Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

Step 3. Keep only $s_i.key$

2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

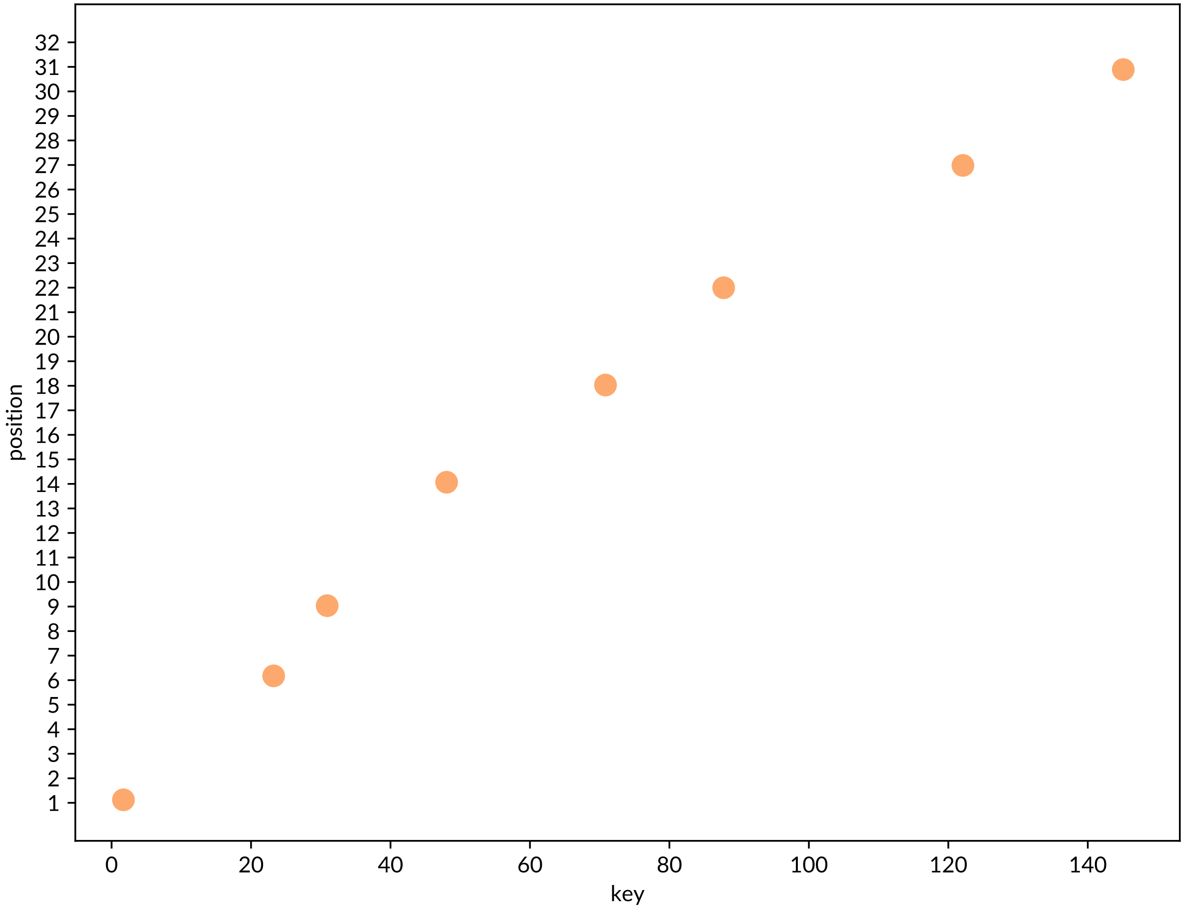
1 $\qquad$ n

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

Step 3. Keep only $s_i.key$

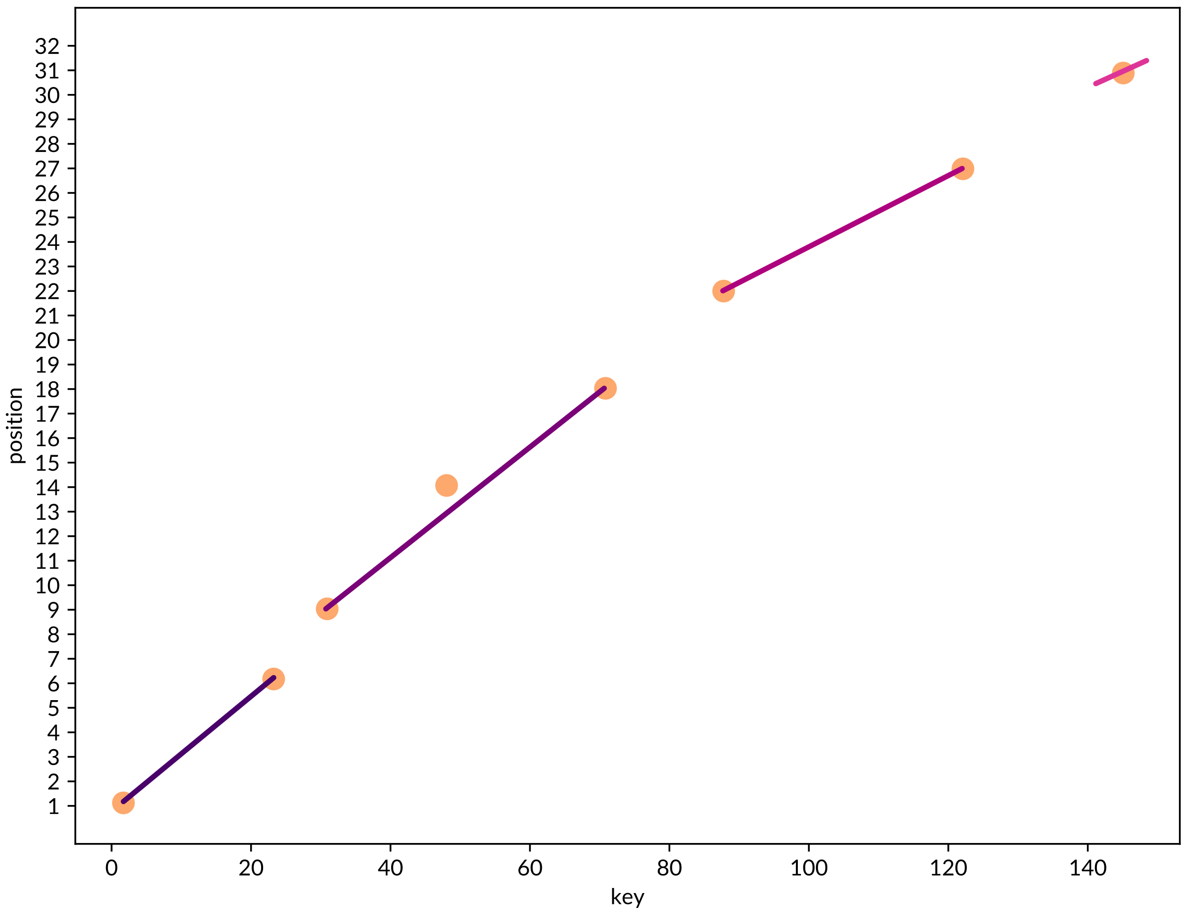


2	23	31	48	71	88	122	145

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 2. Store the segments as triples $s_i = (key, slope, intercept)$



2	23	31	48	71	88	122	145

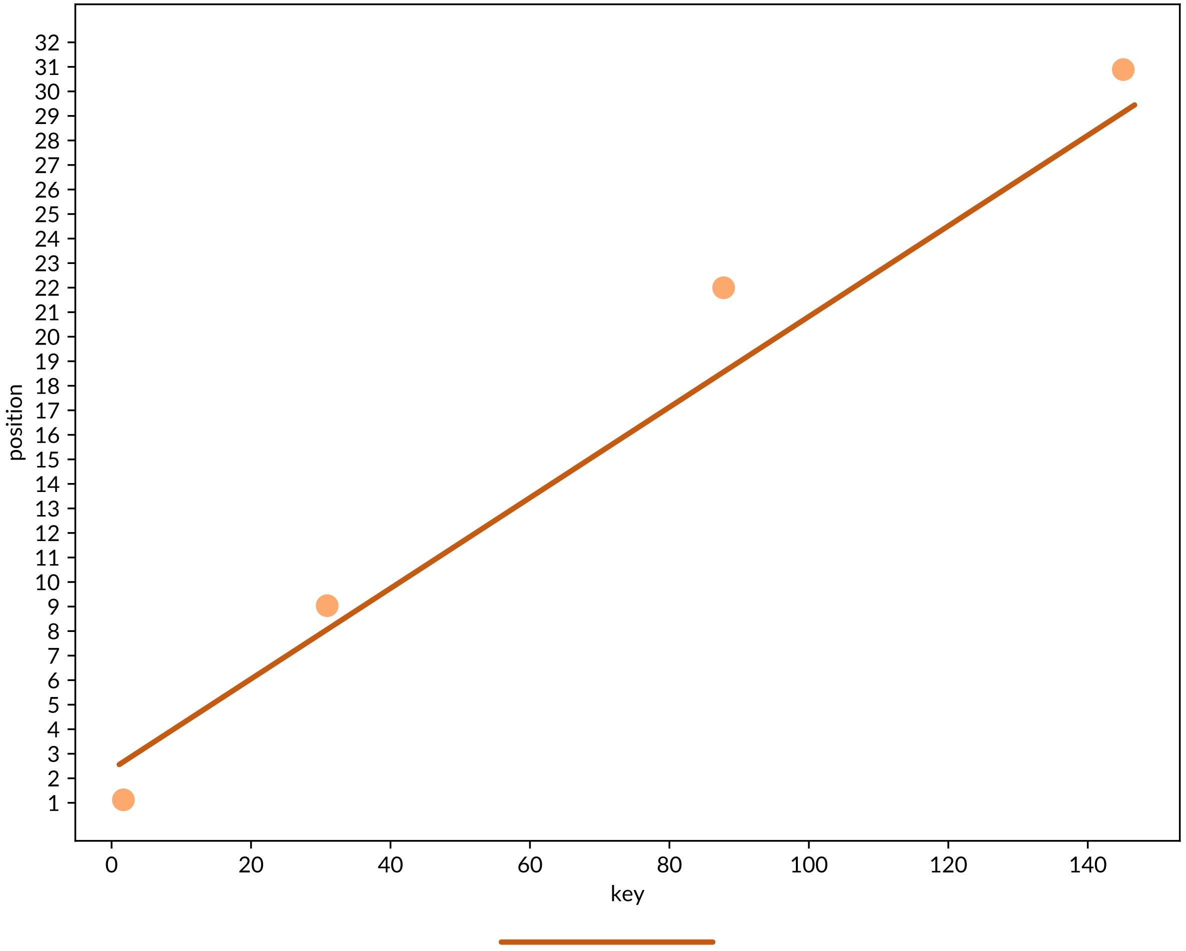
Step 3. Keep only $s_i.key$

Step 4. Repeat recursively

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

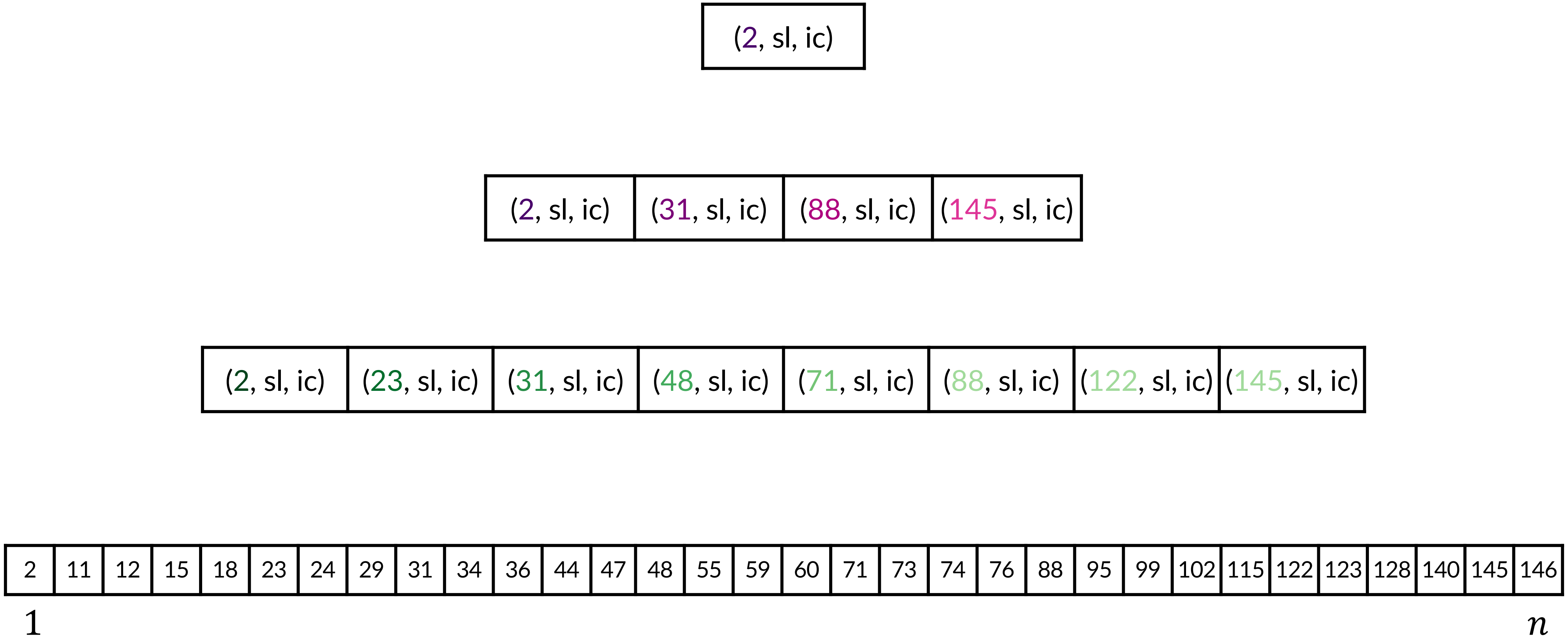


2	31	88	145

Step 3. Keep only $s_i.key$

Step 4. Repeat recursively

Memory layout of the PGM-index

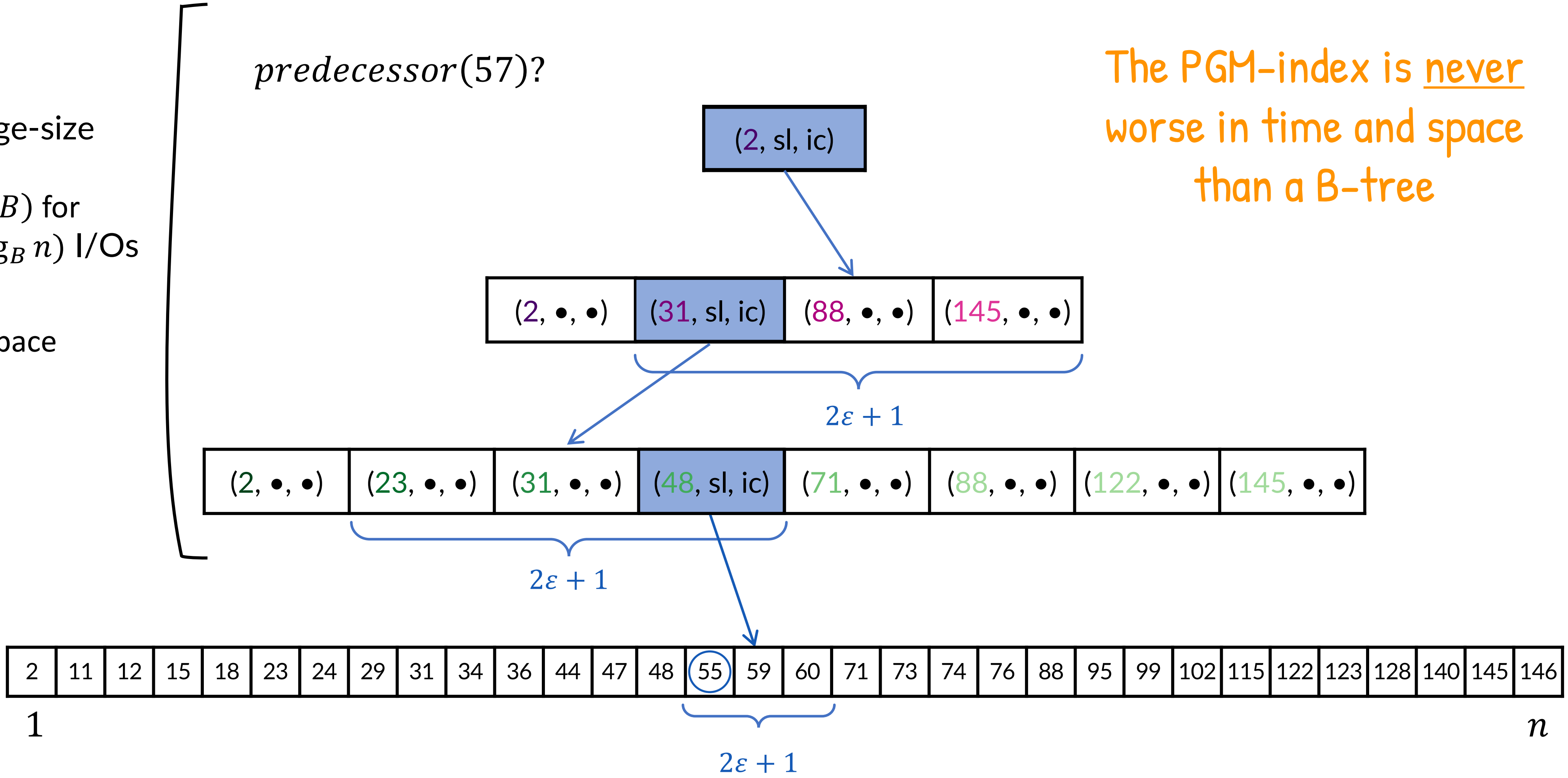


Predecessor search with $\varepsilon = 1$

B = disk page-size

Set $\varepsilon = \Theta(B)$ for queries in $O(\log_B n)$ I/Os

$O(n/\varepsilon)$ space



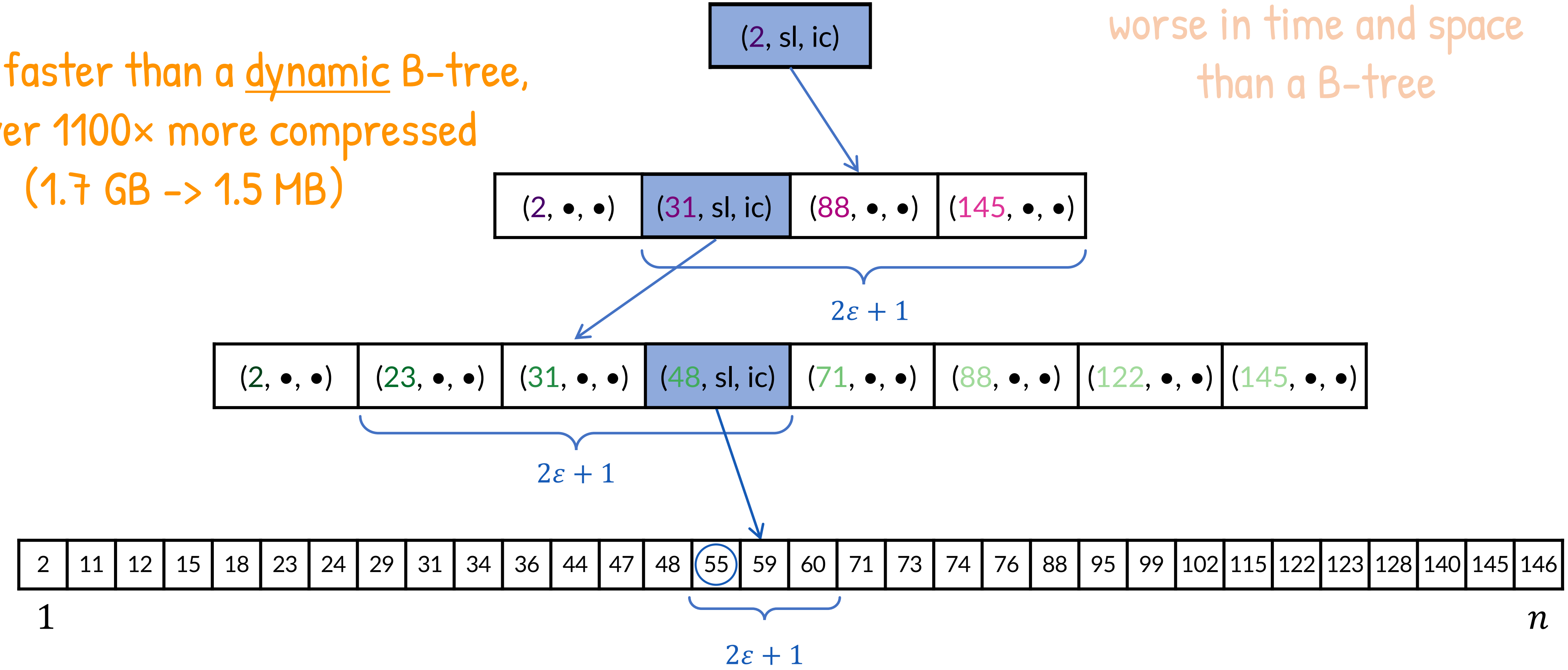
The PGM-index is never worse in time and space than a B-tree

The PGM-index, in practice

As fast as a static cache-optimised B-tree but 80× more compressed

Up to 3× faster than a dynamic B-tree, and over 1100× more compressed (1.7 GB -> 1.5 MB)

The PGM-index is never worse in time and space than a B-tree



A stronger theoretical result on the space of a PGM

Theorem. Consider iid gaps between consecutive input keys with finite mean μ and variance σ^2.

If ε is sufficiently large, the number of segments (≈ the space of a PGM) on n input keys is, with high probability,

$$\frac{\sigma^2}{\mu^2}\,\frac{n}{\varepsilon^2}$$

Corollary. Under the assumption above, the PGM-index with $\varepsilon = \Theta(B)$ improves the space of a B-tree from $\Theta(n/B)$ to $O(n/B^2)$

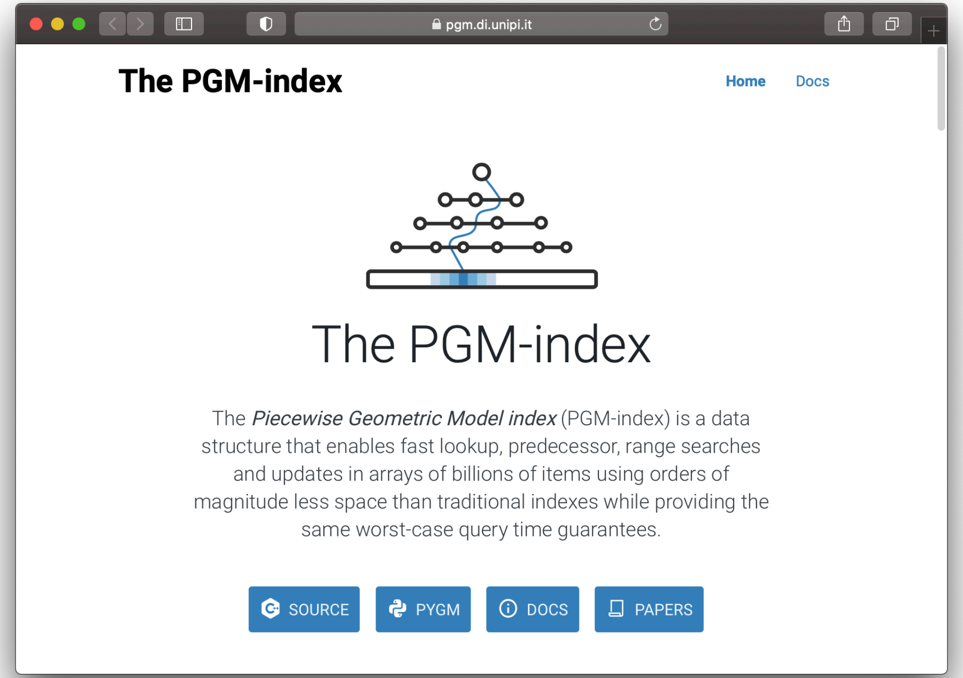
Website and reference implementation

Website: https://pgm.di.unipi.it

Library (C++17): https://github.com/gvinciguerra/PGM-index

Library (Python): https://github.com/gvinciguerra/PyGM

Documentation: https://pgm.di.unipi.it/docs/



API of the PGM-index

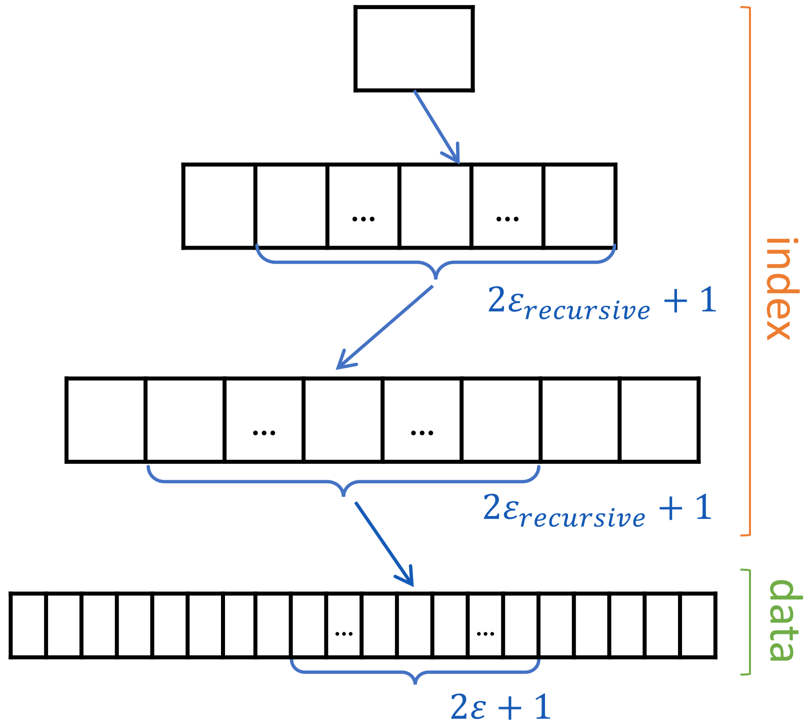
```
#include <vector>
#include <iostream>
#include <algorithm>
#include "pgm_index.hpp"

int main() {
    // Generate some random data
    std::vector<uint32_t> data(1000000);
    std::generate(data.begin(), data.end(), std::rand);
    std::sort(data.begin(), data.end());

    // Construct the PGM-index
    const int eps = 128;
    const int eps_recursive = 8;
    pgm::PGMIndex<uint32_t, eps, eps_recursive> index(data);

    // Query the PGM-index
    auto q = 42;
    auto range = index.search(q);
    auto lo = data.begin() + range.lo;
    auto hi = data.begin() + range.hi;
    std::cout << *std::lower_bound(lo, hi, q) << std::endl;
    std::cout << index.size_in_bytes() << std::endl;

    return 0;
}
```



API of the piecewise linear model

- To construct the vector of segments

```
std::vector&lt;Segment&gt;
make_pgm_segments(
    const std::vector<uint64_t> &data,
    size_t epsilon
);
```

- To compute a prediction

```
size_t pos = segments[i](key)
```

(this API is available only on the challenge website)

The challenge

Beat the space-time Pareto curve of the C++ reference implementation of the PGM-index

- Use all the tools (including PGM) you have seen in the lab lectures as Lego bricks to design your solution
- Propose new ideas and code them
- Required methods:
  - `MyIndex(const std::vector<uint64_t> &data)`
  - `size_t size_in_bytes()`
  - `uint64_t nextGEQ(uint64_t x) // assume: data.front()≤x<data.back()`

The challenge (cont.)

- Submit your `index.hpp` to ae2020challenge.di.unipi.it and check the real-time leaderboard
- Three datasets with possibly repeated keys (read-only plain vector)
- Time-space performance is important, but more important are originality and elegance of the solution