SplinterDB:
Closing the NVMe Bandwidth Gap

Alex Conway

Vijay Chidambaram
Martin Farach-Colton
Abihsihek Gupta
Richard Spillane
Amy Tai
Rob Johnson

ATC 2020

SplinterDB:
A Key-Value Store for the Hard Cases

Key-Value Stores

What’s hard?

Fast Storage

Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency

Key-Value Stores

What’s hard?

Fast Storage

Small Key-Value Pairs

Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency

Keep key-value pairs sorted and packed into data blocks, delay merging as much as possible

Key-Value Stores

What’s hard?

Fast Storage

Small Key-Value Pairs

Small Cache

Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency

Keep key-value pairs sorted and packed into data blocks, delay merging as much as possible

Make all data structures swappable in order to gracefully degrade under cache pressure

In this talk

Fast Storage
(NVMe)

SplinterDB

Data Structures

Flush-then-Compact

In this talk

Fast Storage (NVMe)

SplinterDB

Data Structures

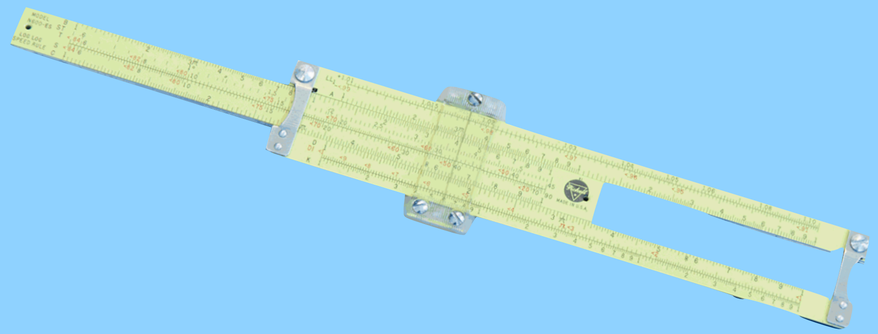
Flush-then-Compact

Back in the day…

People used to use…

Back in the day…

People used to use…

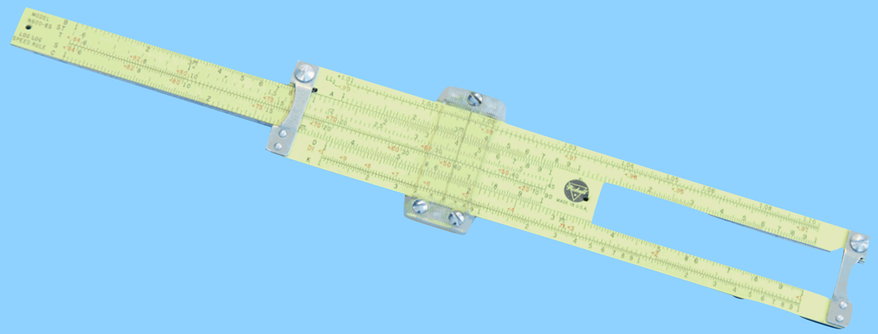


Slide rules

Back in the day…

People used to use…

VHS tapes

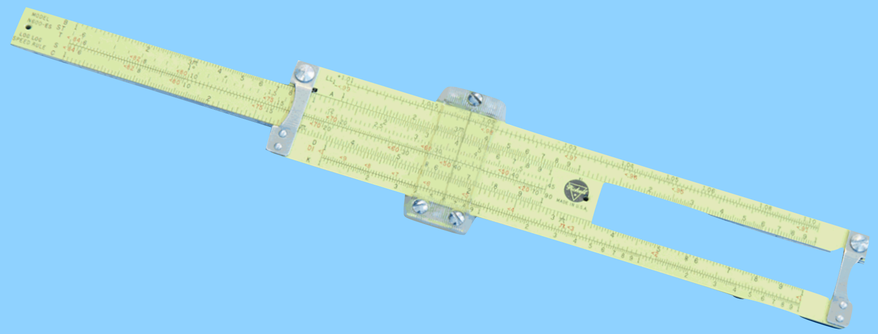


Slide rules

Back in the day…

People used to use…

VHS tapes

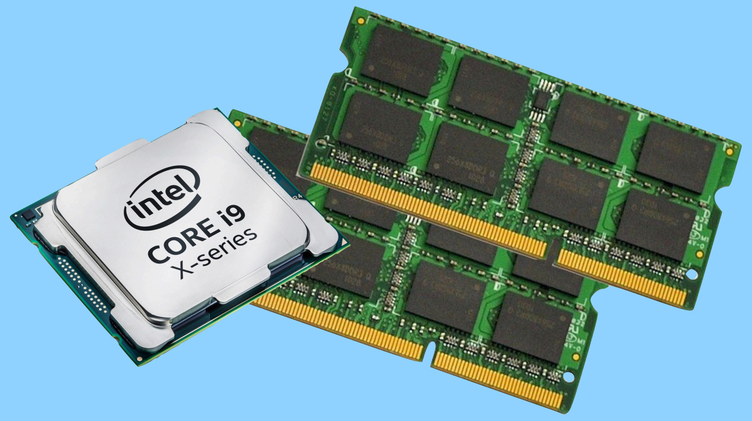


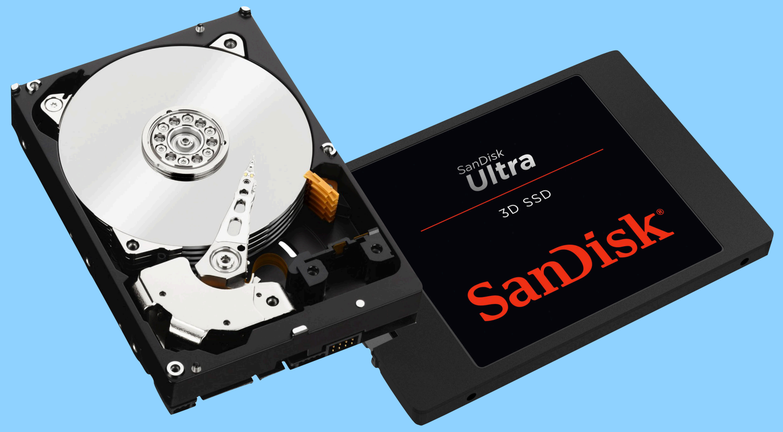
Slide rules

Fountain pens

Back in the day…

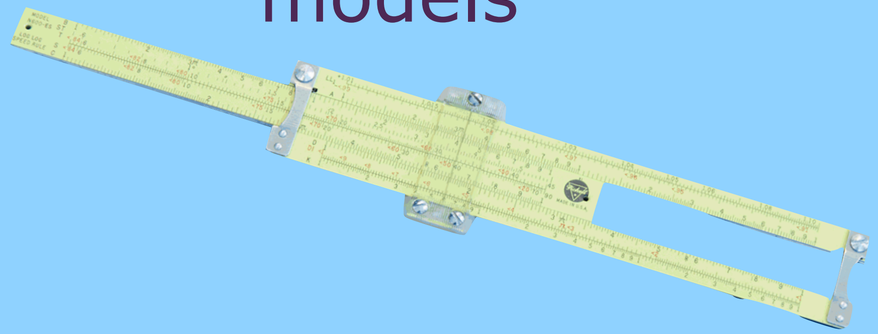
People used to use…





VHS tapes

Different Performance models

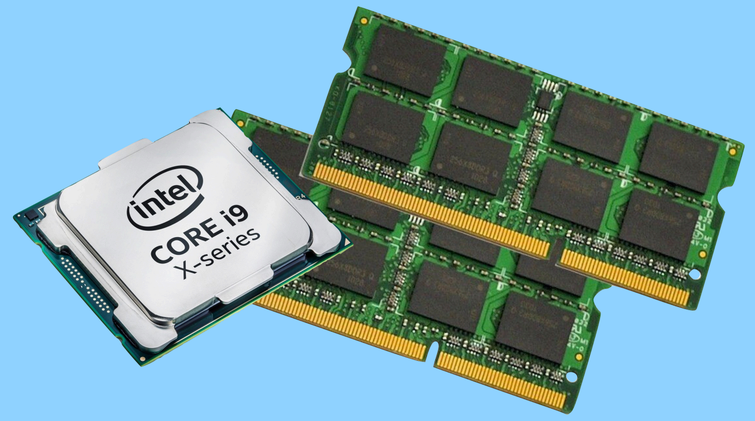


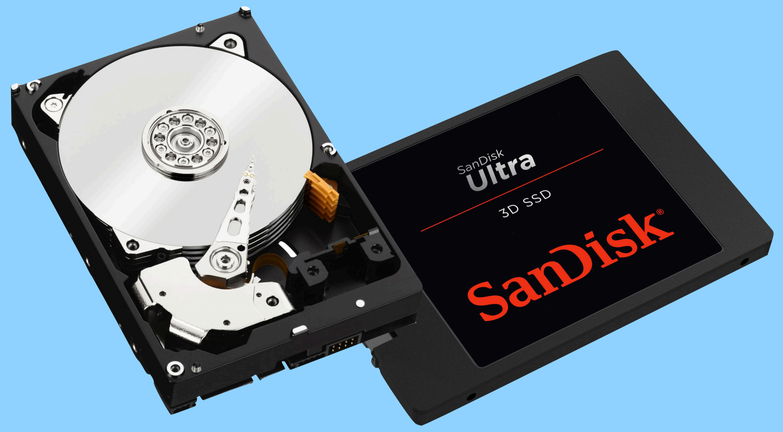
Slide rules

Fountain pens

Back in the day…

People used to use…





CPU

Different Performance models

IO

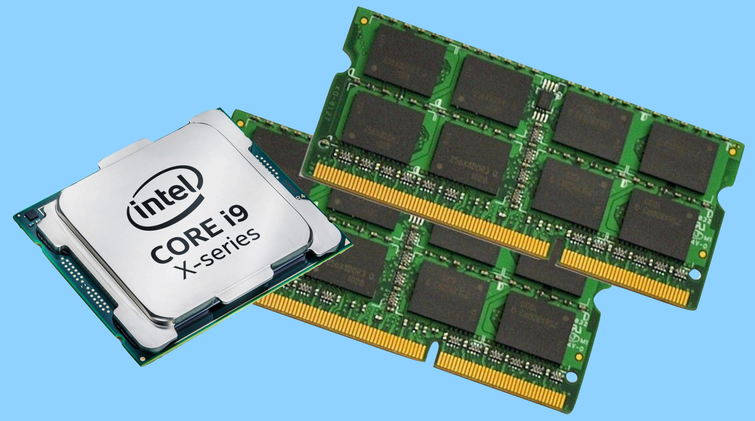
VHS tapes

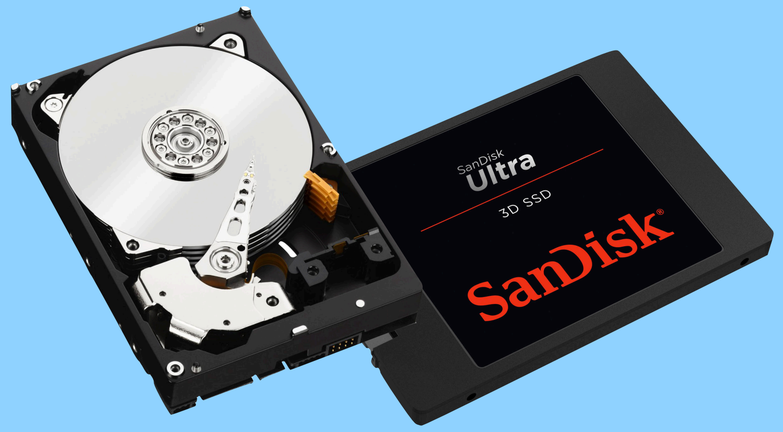
Slide rules

Fountain pens

Back in the day…

People used to use…





CPU

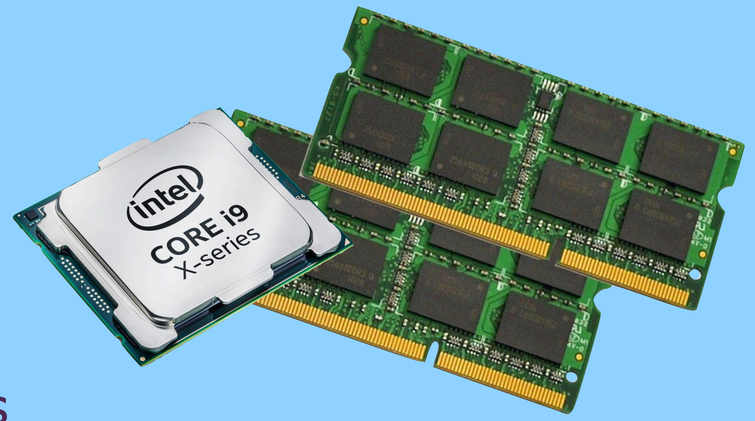
Different Performance models

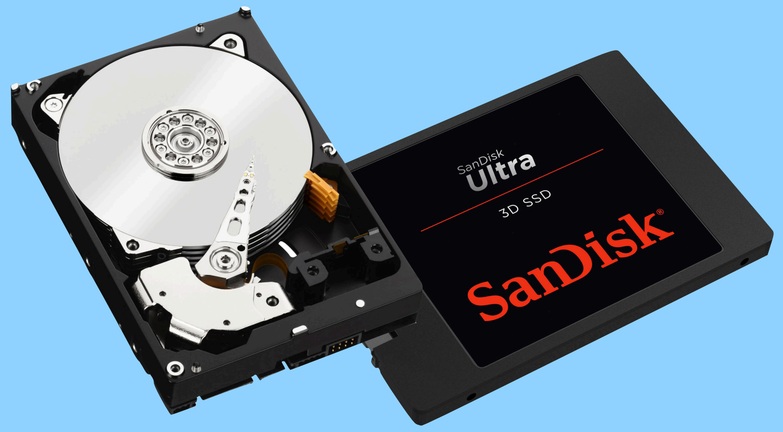
IO

Look at *e.g.* key-value stores

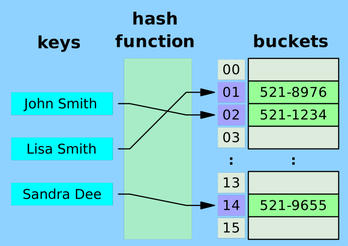
Back in the day…

People used to use…





hash tables

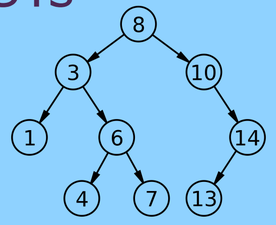


CPU

Different Performance models

IO

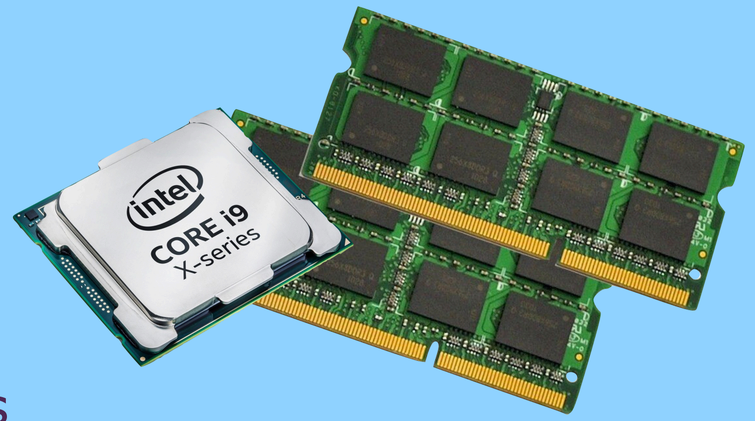
BSTs

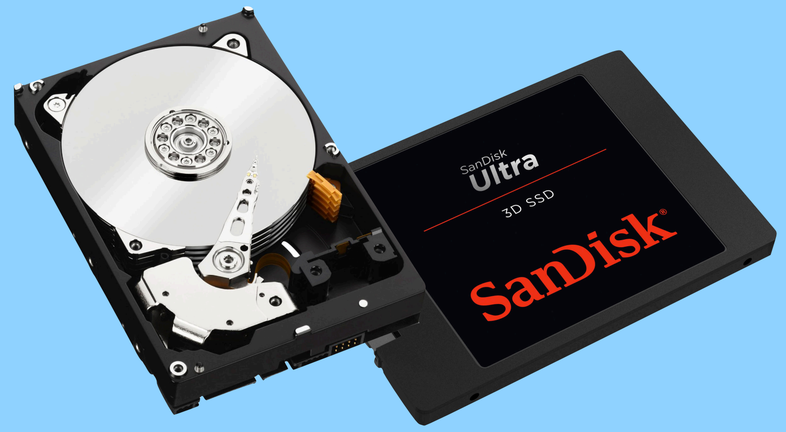


Look at *e.g.* key-value stores

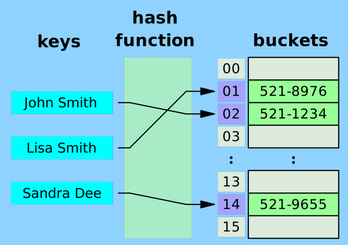
Back in the day…

People used to use…





hash tables

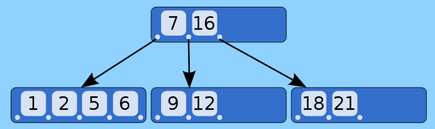


CPU

Different Performance models

IO

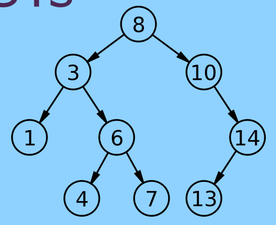
B-trees



SCYLLA.

RocksDB

BSTs

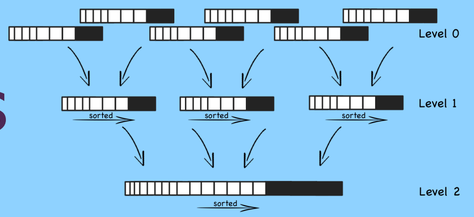


Memcached

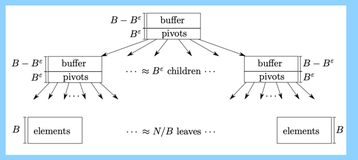
redis

Look at *e.g.* key-value stores

LSMs

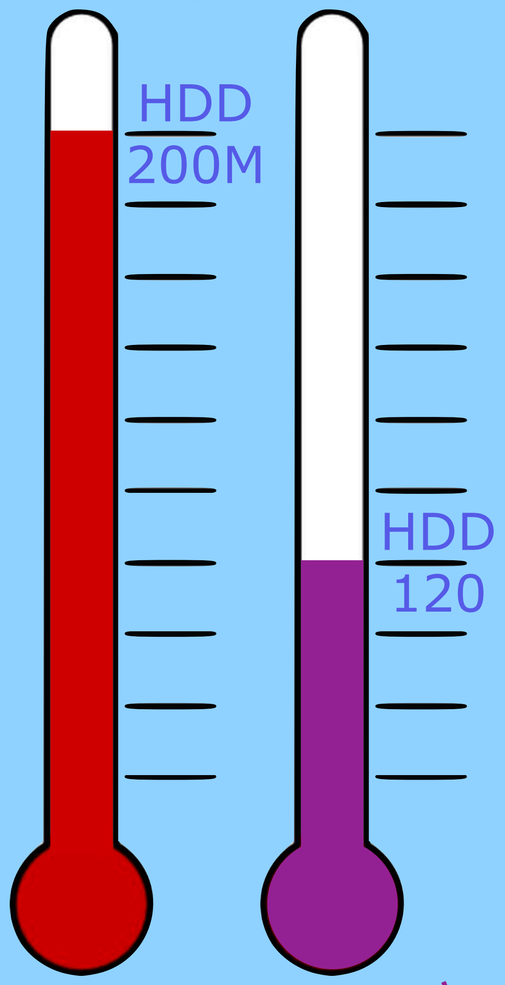


Bε-trees

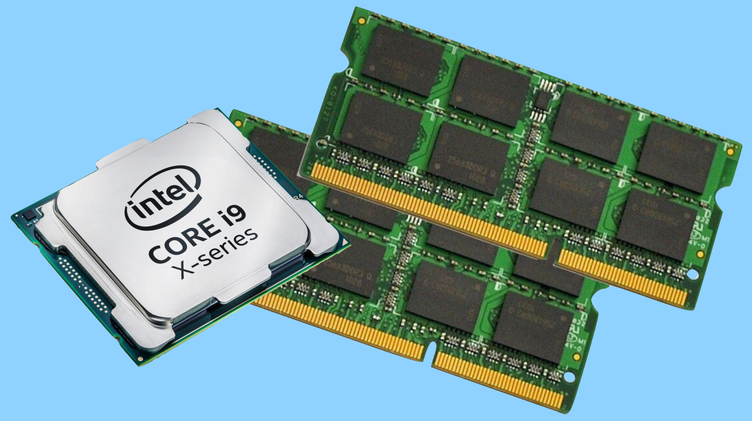


Back in the day…

Cycles per 64b word at bandwidth



People used to use…



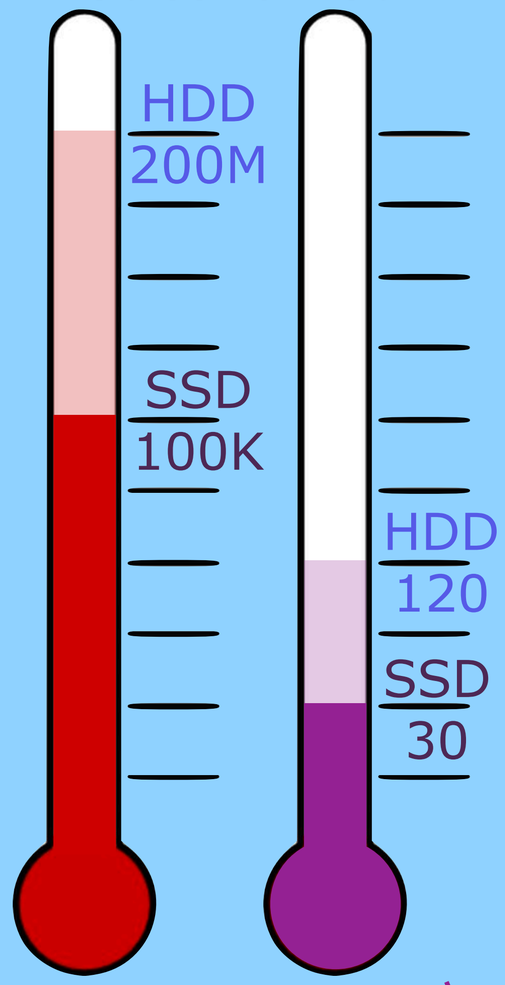
CPU

Different Performance models

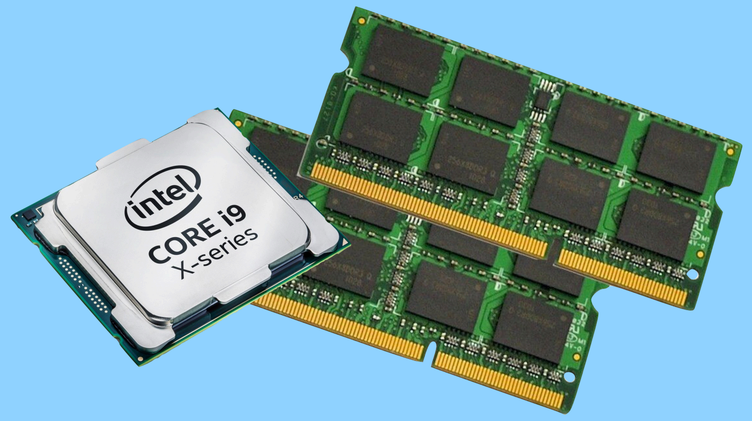
IO

Back in the day…

Cycles per 64b word at bandwidth



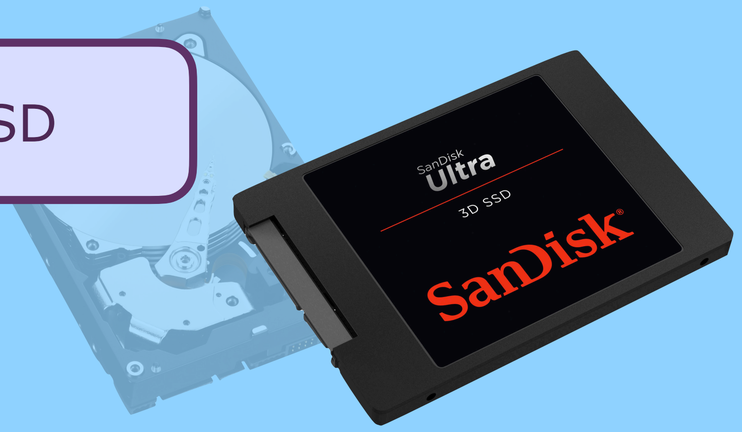
People used to use…



CPU

Different Performance models

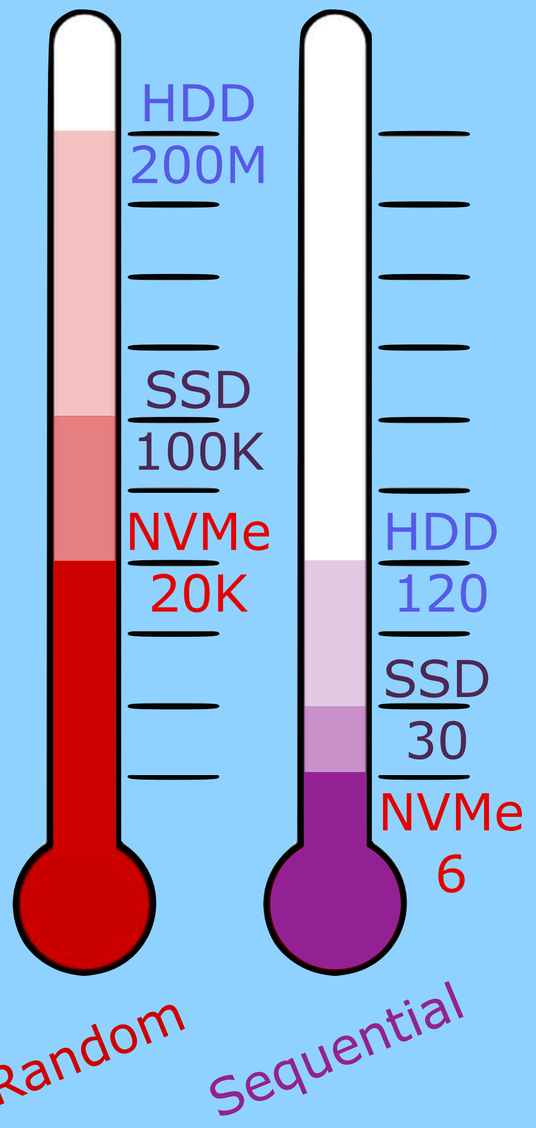
SSD



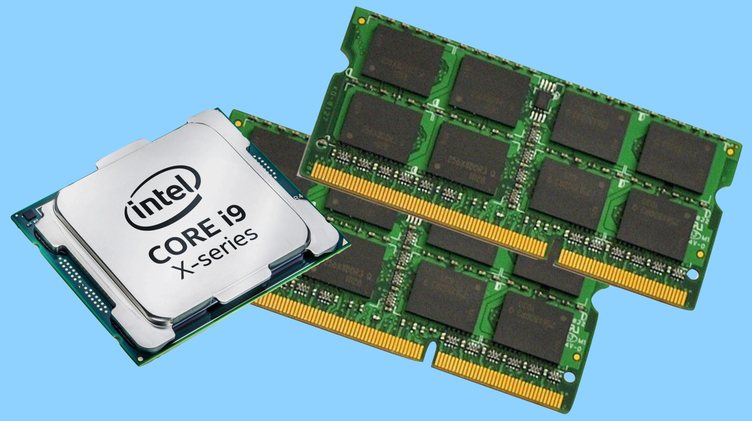
IO

Back in the day…

Cycles per 64b word at bandwidth



People used to use…



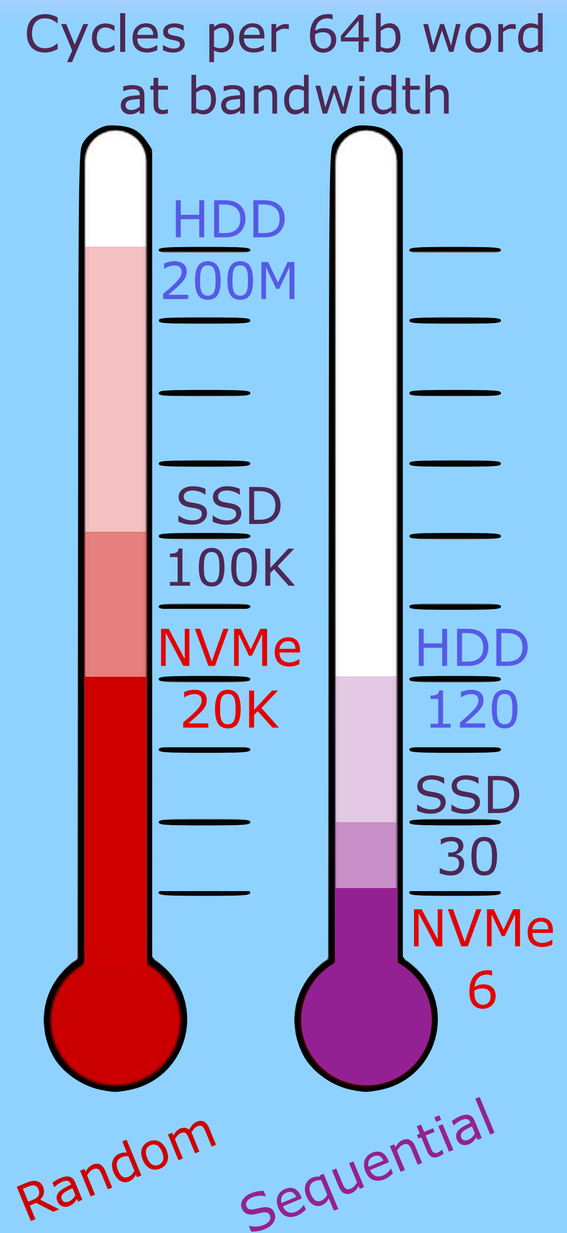
CPU

Different Performance models

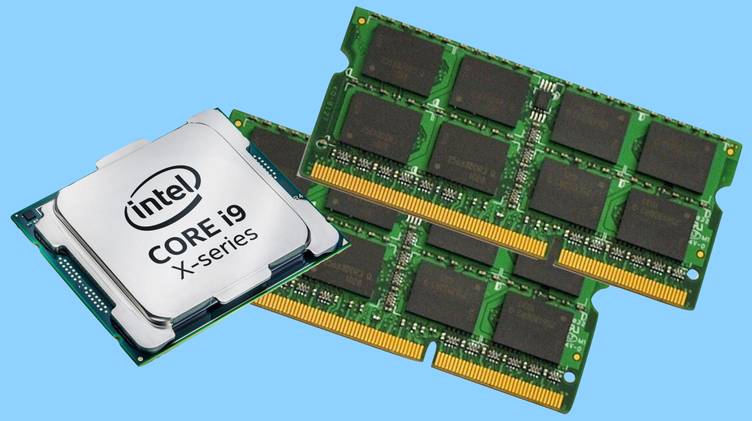
NVMe

IO

Back in the day…



People used to use…



NVMe

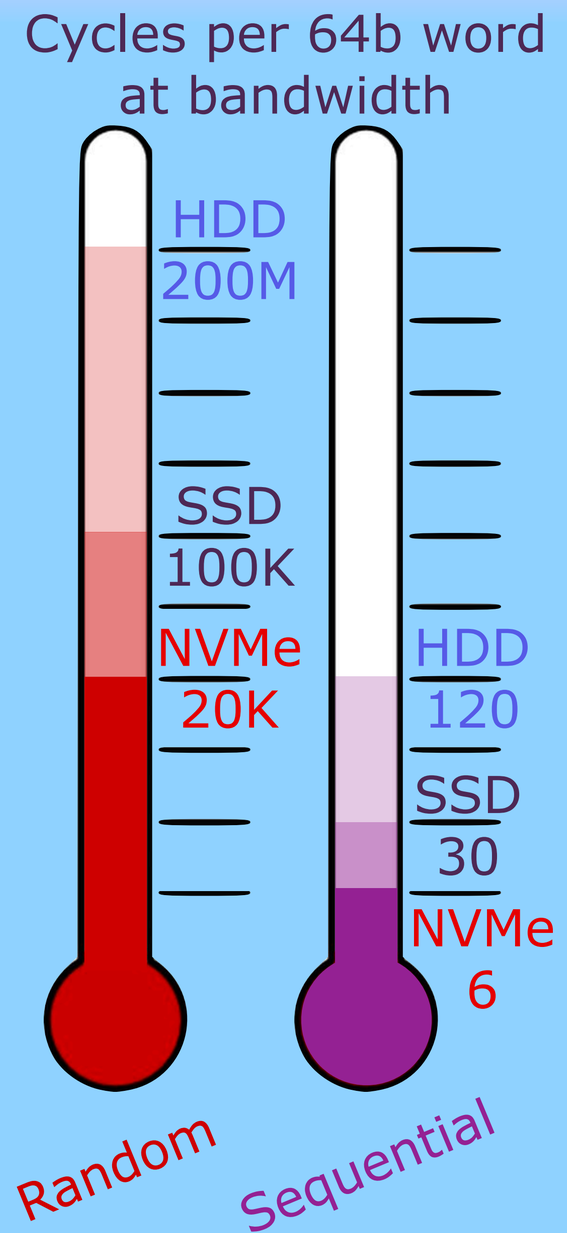
CPU

Different Performance models

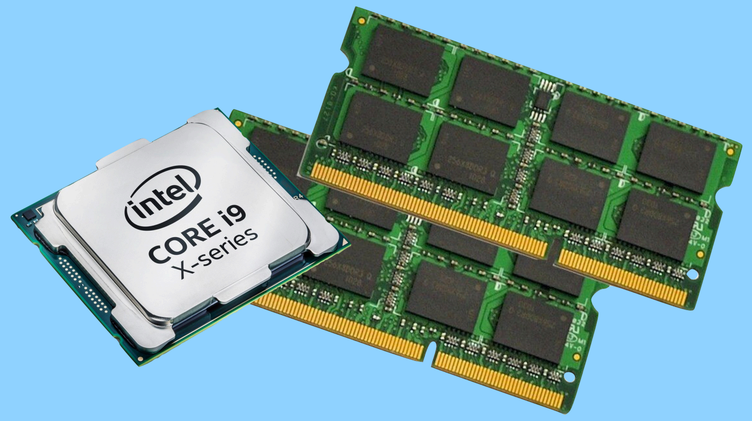
IO

For NVMe, need data structures to optimize both

Back in the day…



People used to use…



CPU

NVMe

Different Performance models

IO

For NVMe, need data structures to optimize both

~~Write Amplification~~

Work Amplification

In this talk

Fast Storage (NVMe)

SplinterDB

Data Structures

Flush-then-Compact

SplinterDB

SplinterDB is a key-value store
which handles these tough cases:

Fast Storage

Small Key-Value Pairs

Small Cache

SplinterDB

SplinterDB is a key-value store
which handles these tough cases:

Fast Storage ✓

Small Key-Value Pairs ✓

Small Cache ✓

SplinterDB

SplinterDB is a key-value store
which handles these tough cases:

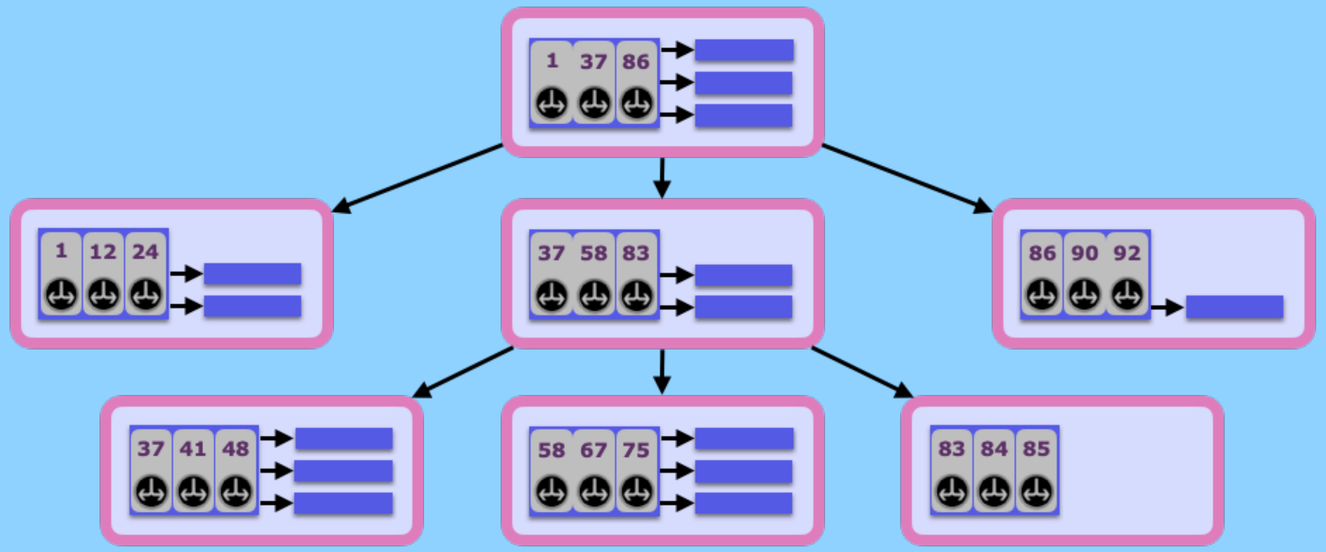
Fast Storage 

Small Key-Value Pairs 

Small Cache 

SplinterDB

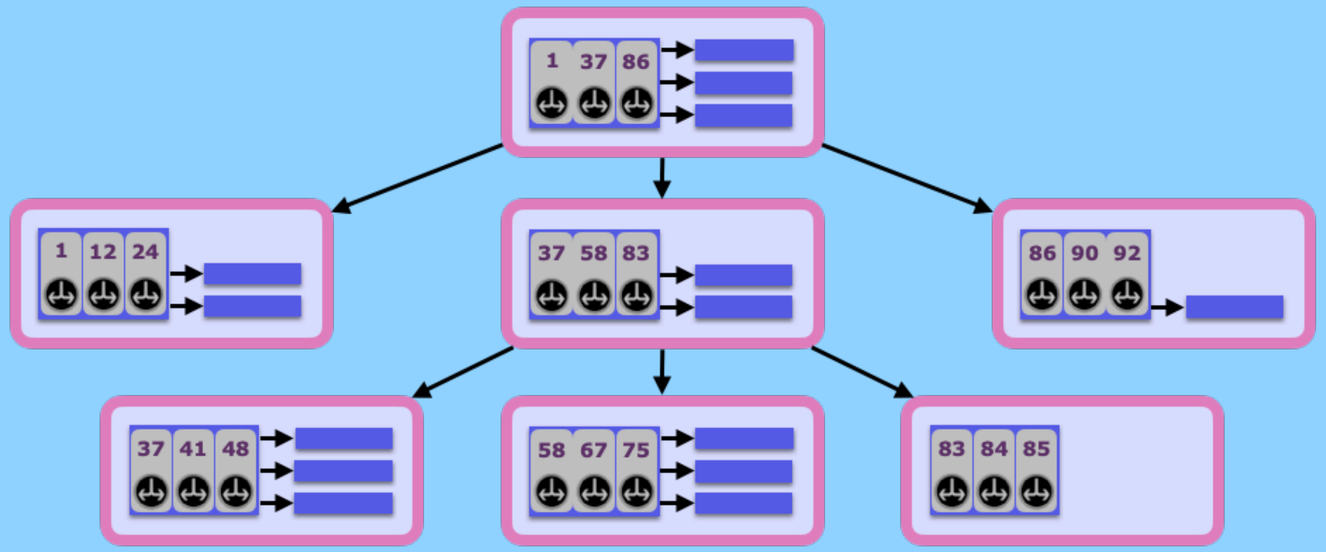
SplinterDB is a key-value store
which handles these tough cases:



Size-Tiered Bε-Tree

SplinterDB

SplinterDB is a key-value store
which handles these tough cases:

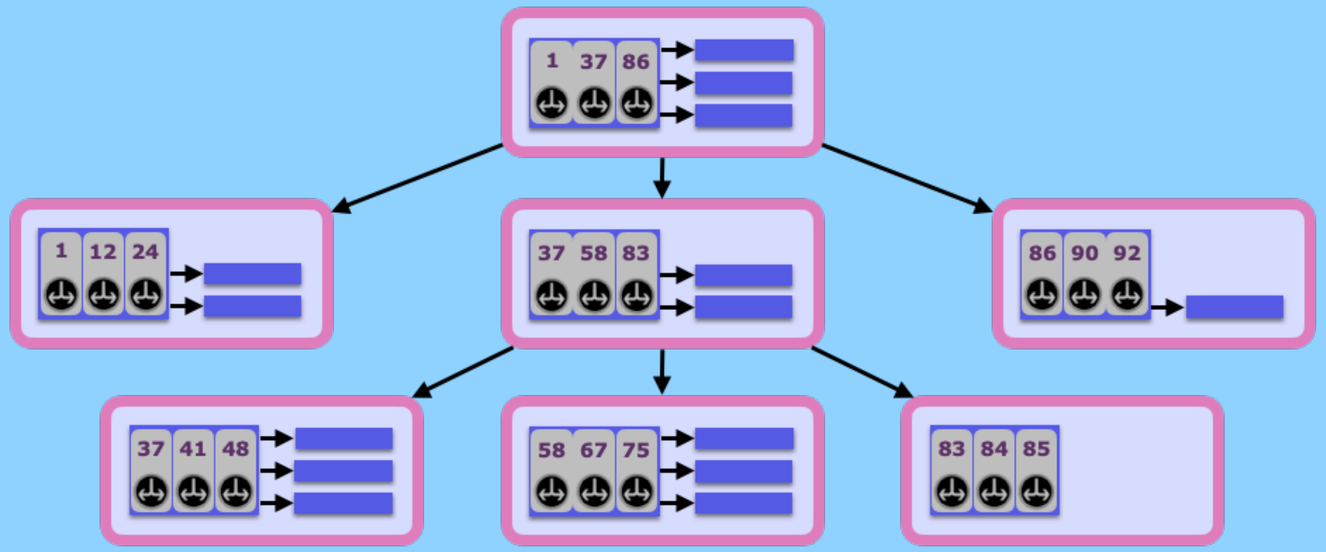


Size-Tiered Bε-Tree

Reducing Work

SplinterDB

SplinterDB is a key-value store
which handles these tough cases:



Size-Tiered Bε-Tree

Reducing Work

Concurrency

How Does SplinterDB Perform?

State of the Art: RocksDB

RocksDB

RocksDB is a high performance embedded database for key-value data. It is a fork of LevelDB by Facebook optimized to exploit many central processing unit (CPU) cores, and make efficient use of fast storage, such as solid-state drives (SSD), for input/output (I/O) bound workloads.

WIKIPEDIA
The Free Encyclopedia

- Released 2012, LevelDB traces back to 2004
- Built and maintained by full-time engineering team
- Continuous performance improvements

SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

Block-addressable NVMe

24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)

SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

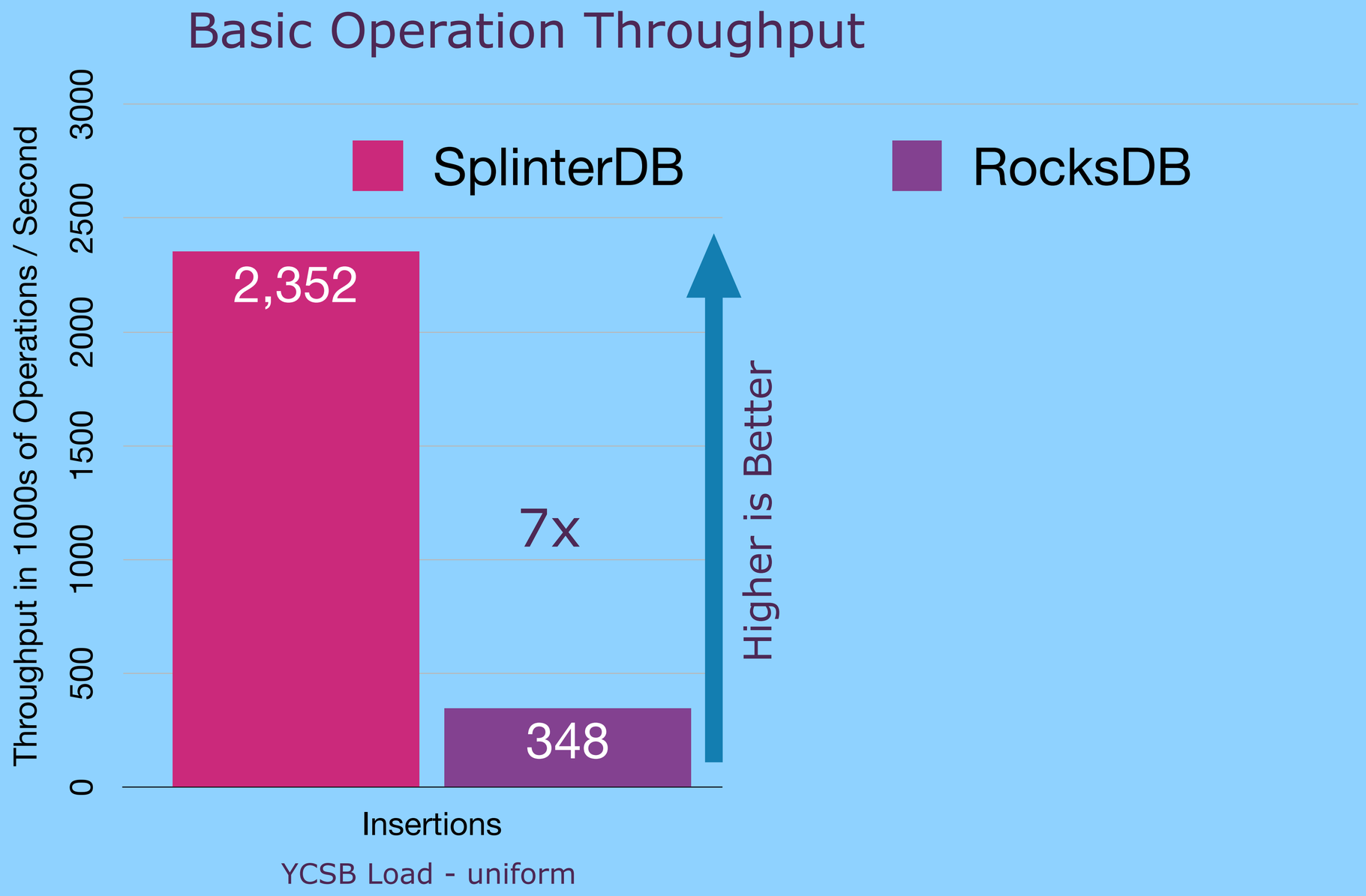
Block-addressable NVMe

24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)



SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

Block-addressable NVMe

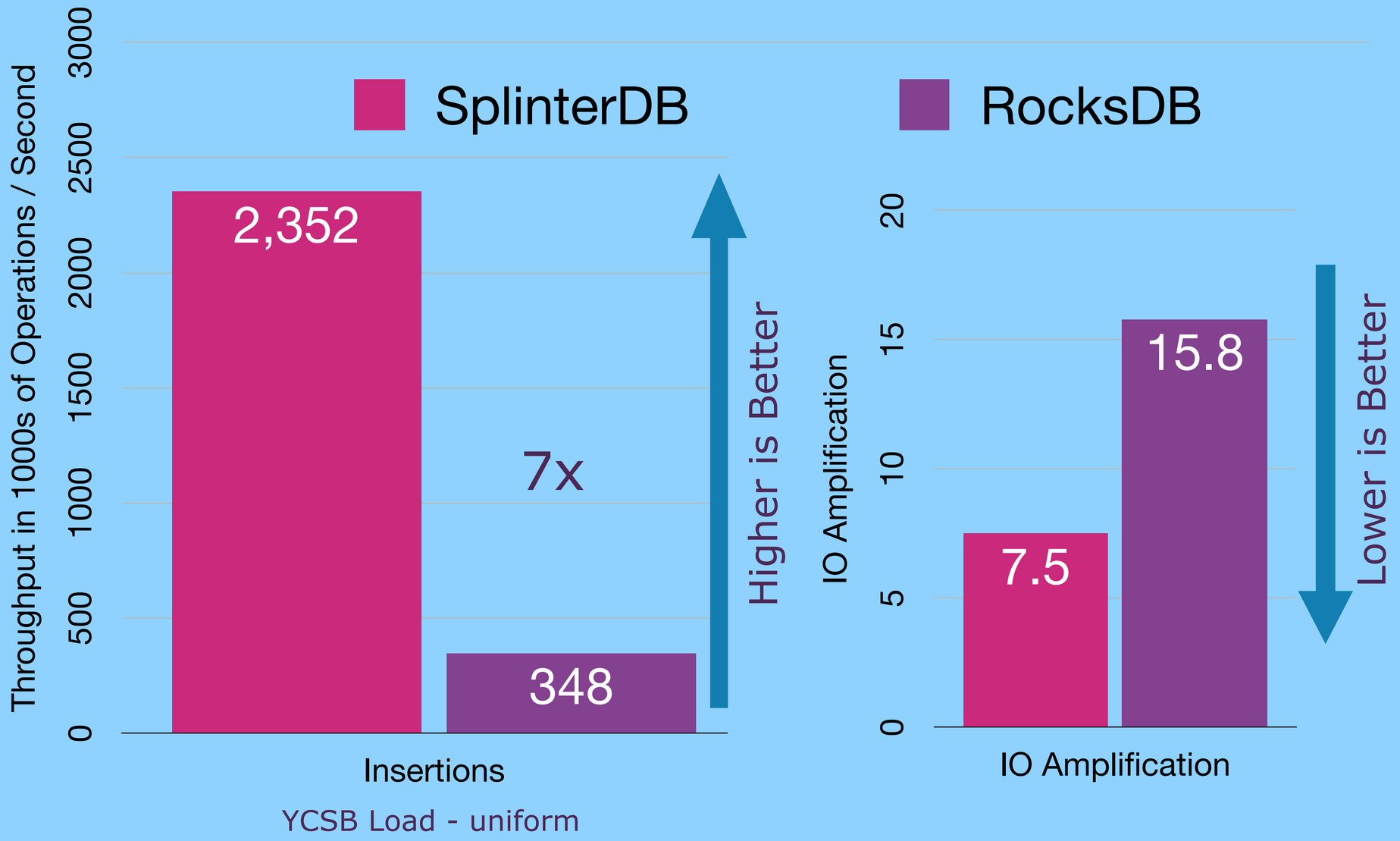
24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)

Basic Operation Throughput



SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

Block-addressable NVMe

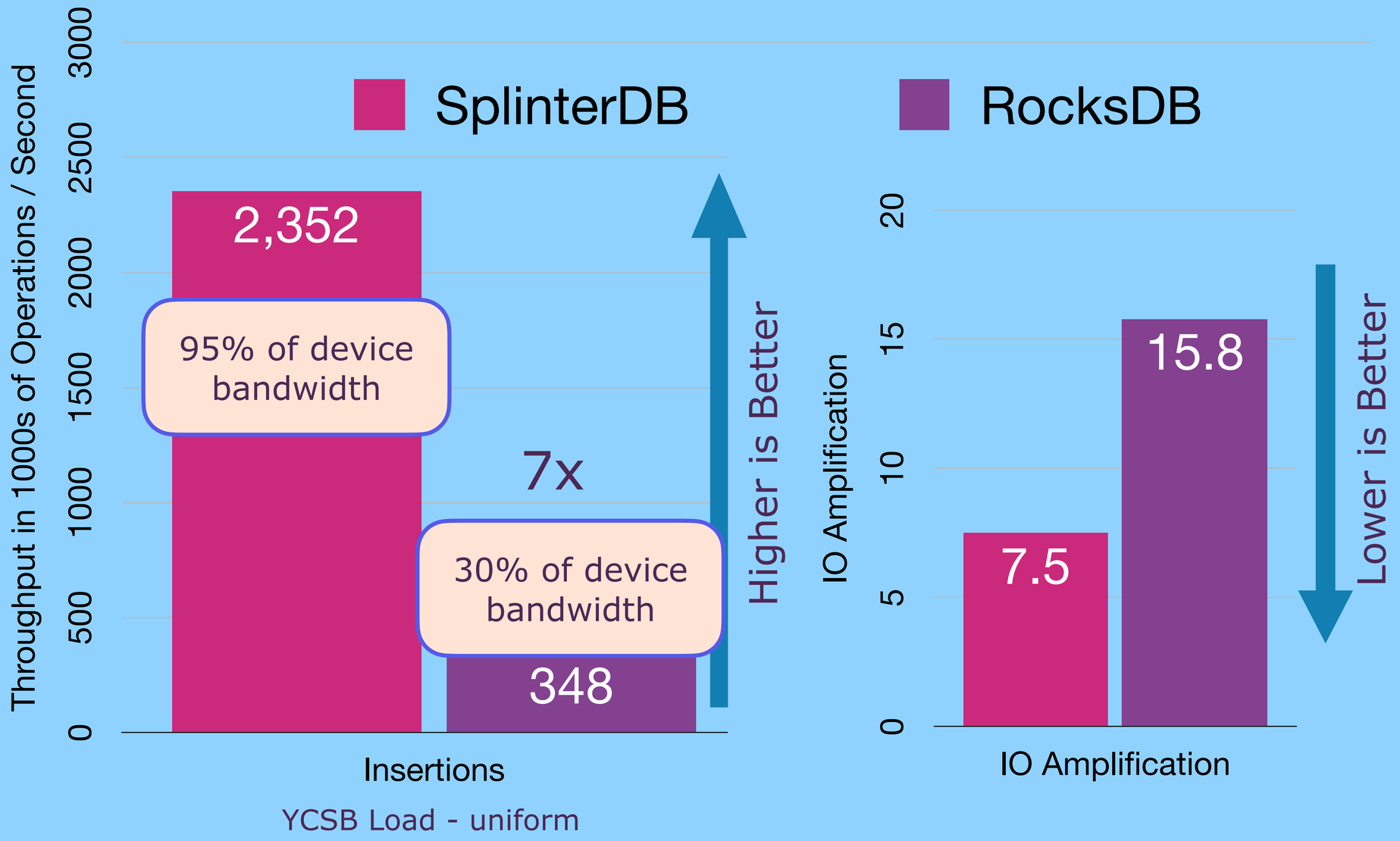
24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)

Basic Operation Throughput



SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

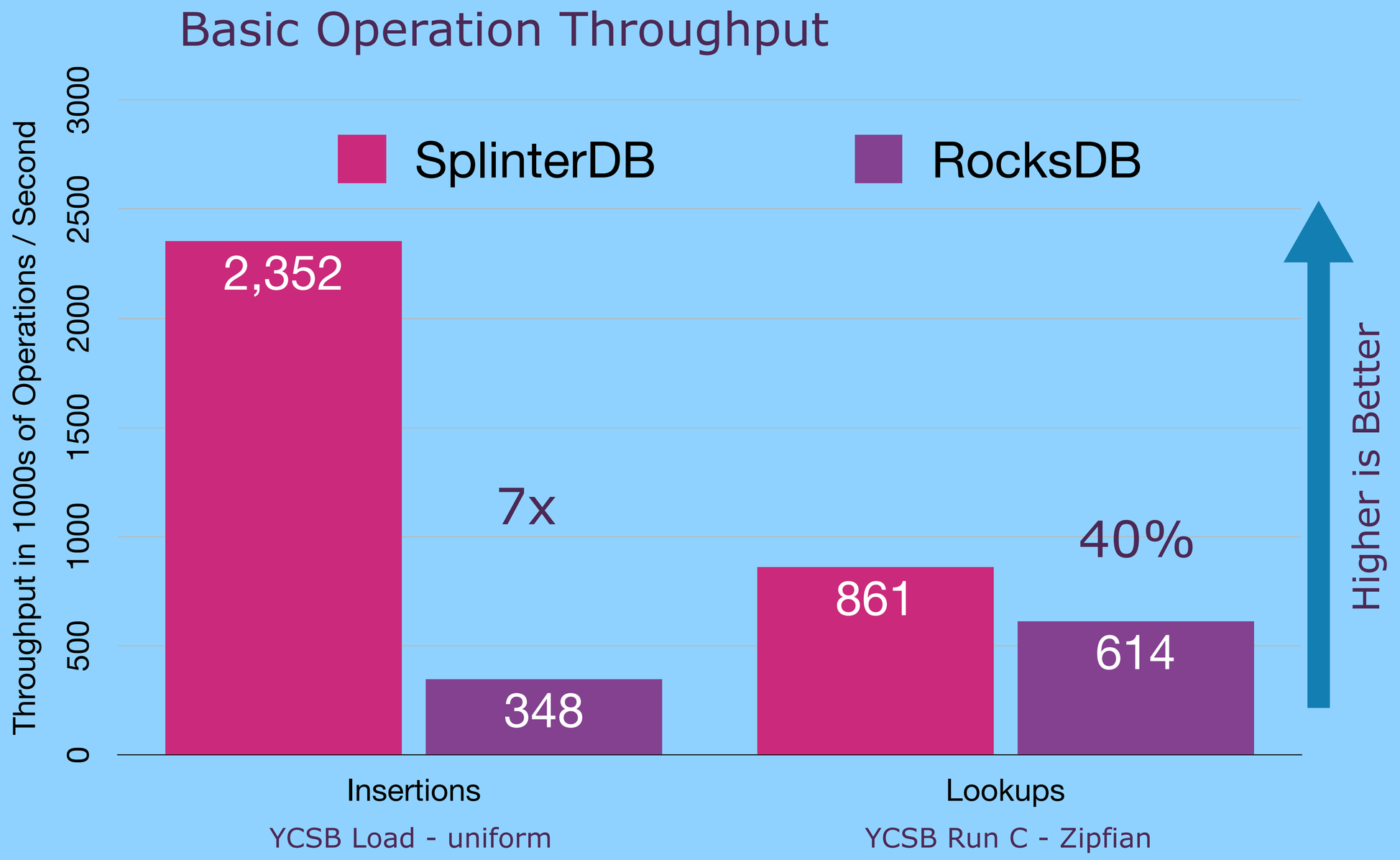
Block-addressable NVMe

24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)



SplinterDB Performance

32 2Ghz cores

Intel Optane 905P

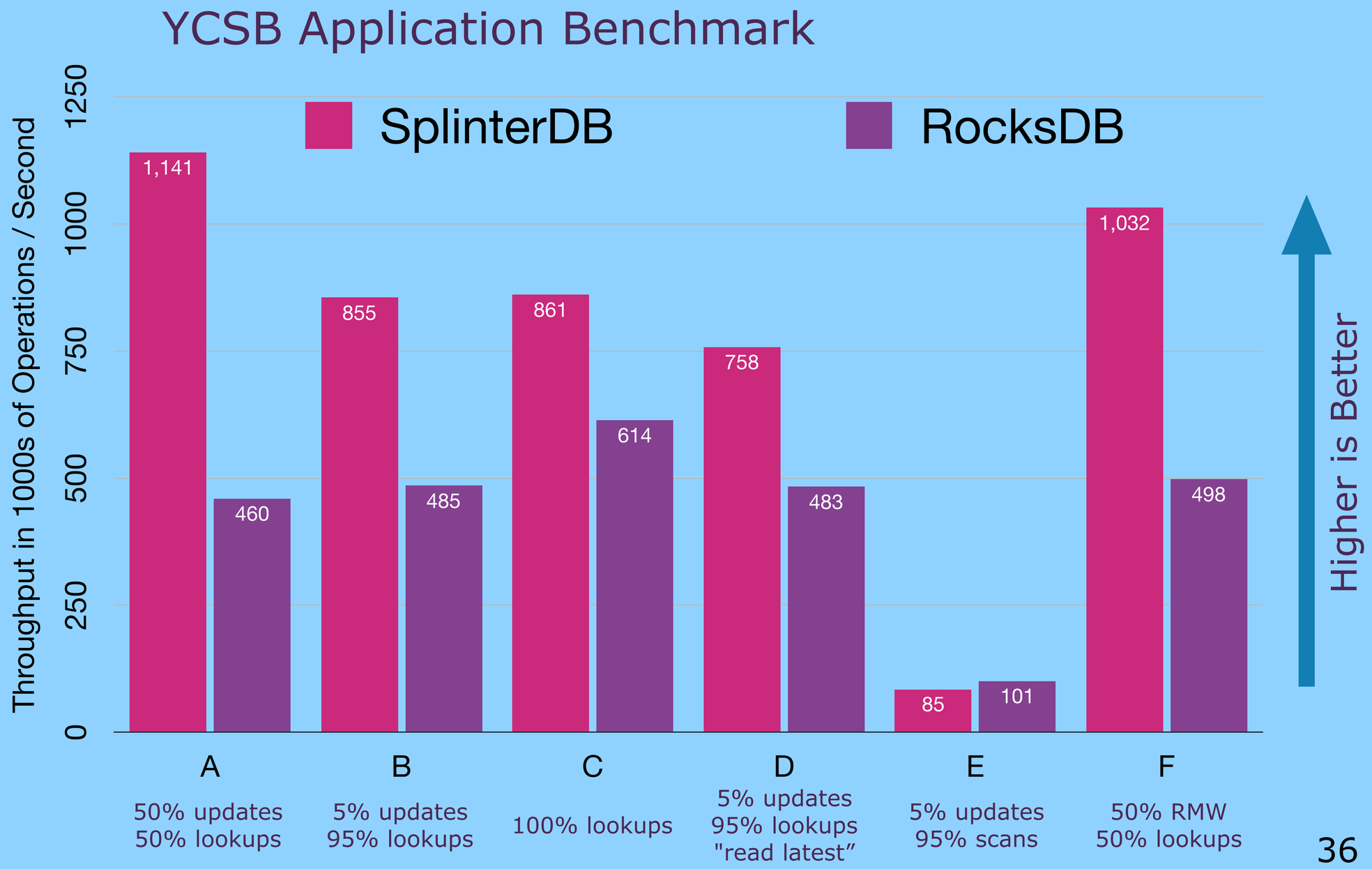
Block-addressable NVMe

24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset

Small cache
(using cgroup)



In this talk

Fast Storage
(NVMe)

SplinterDB

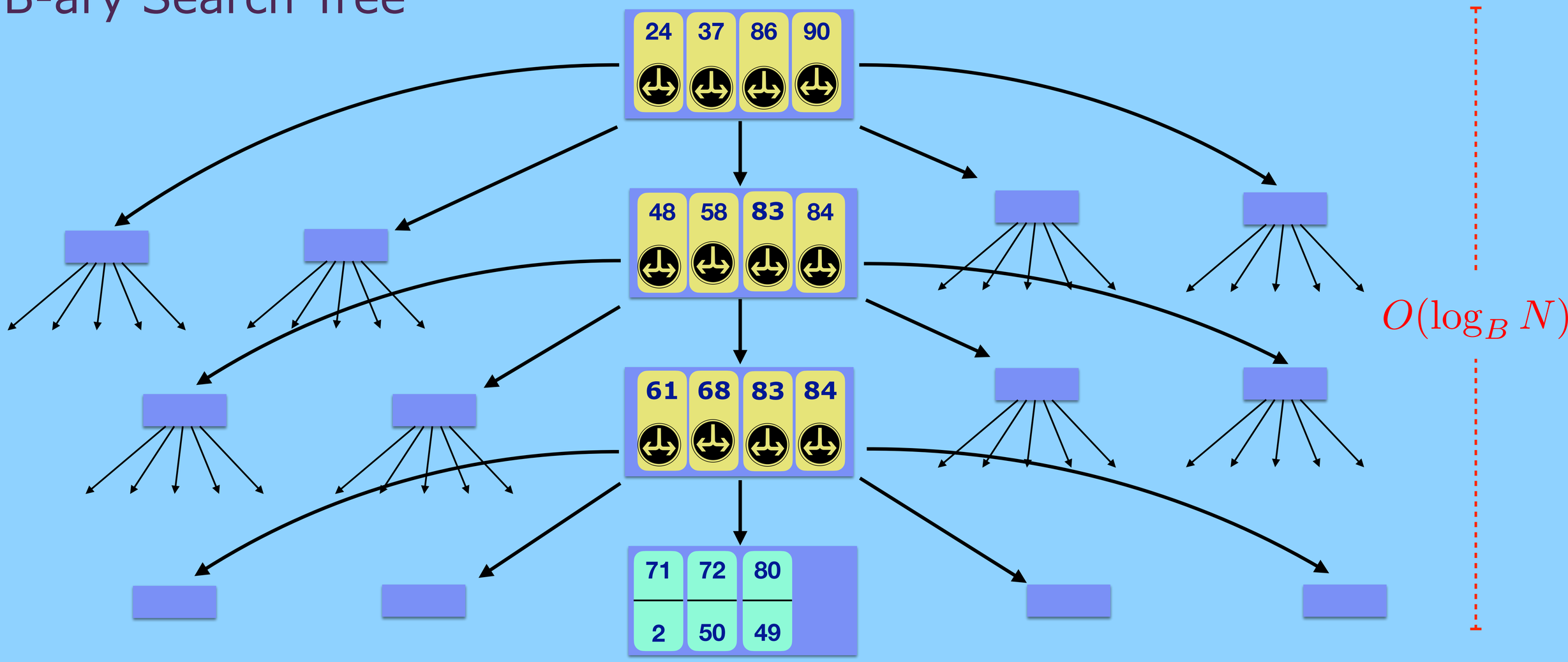
Data Structures

Flush-then-Compact

B-Trees

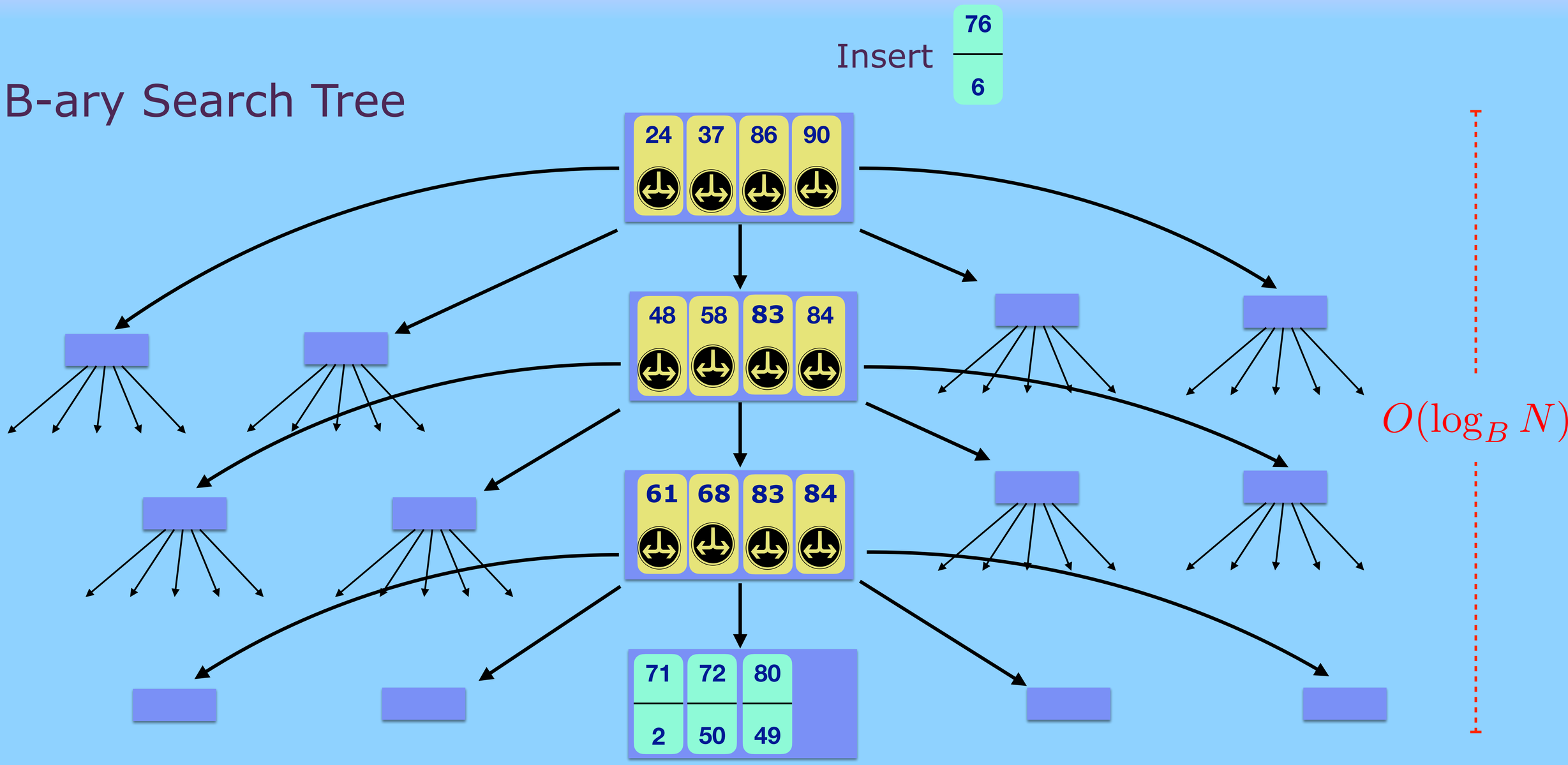
B-Trees

B-ary Search Tree



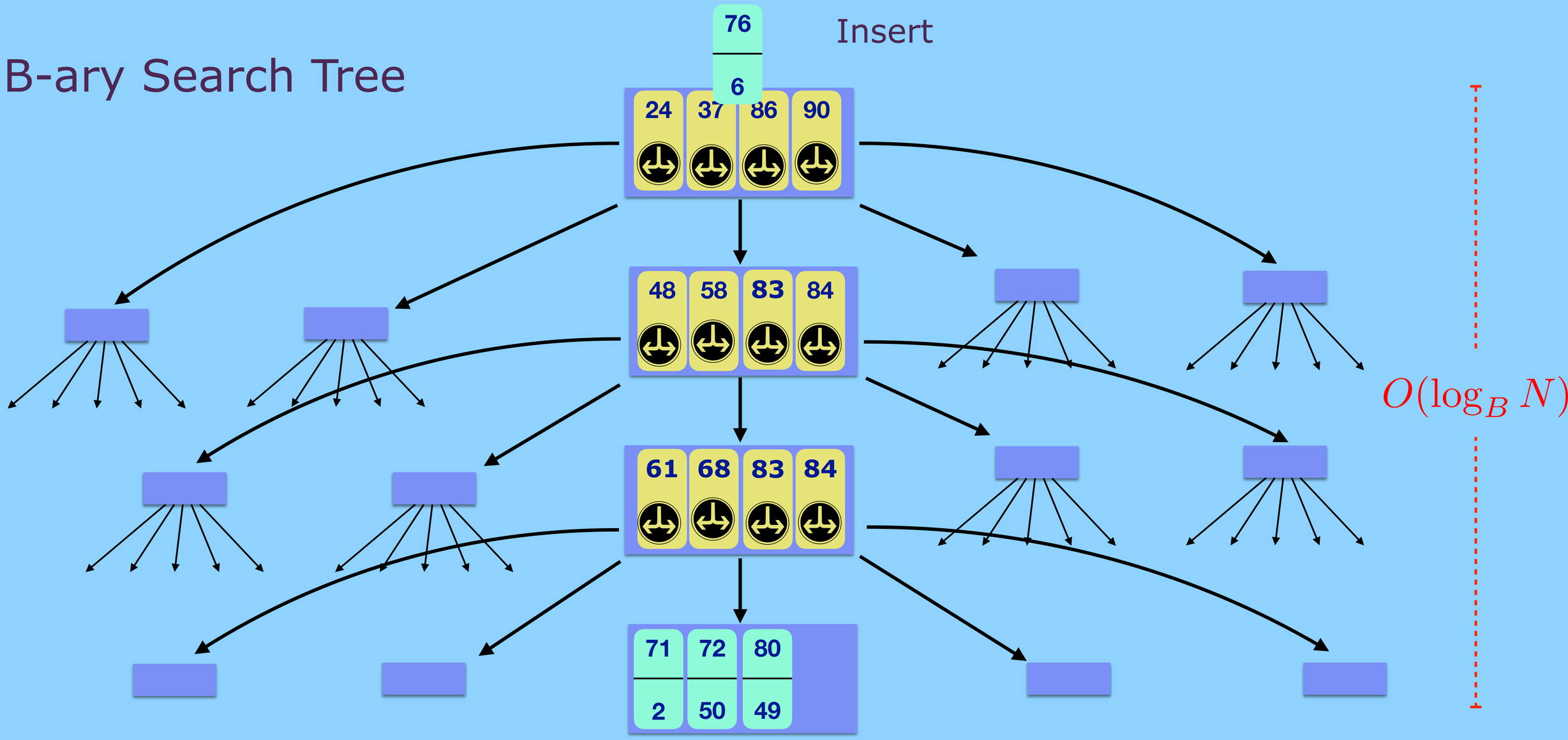
B-Trees

B-ary Search Tree



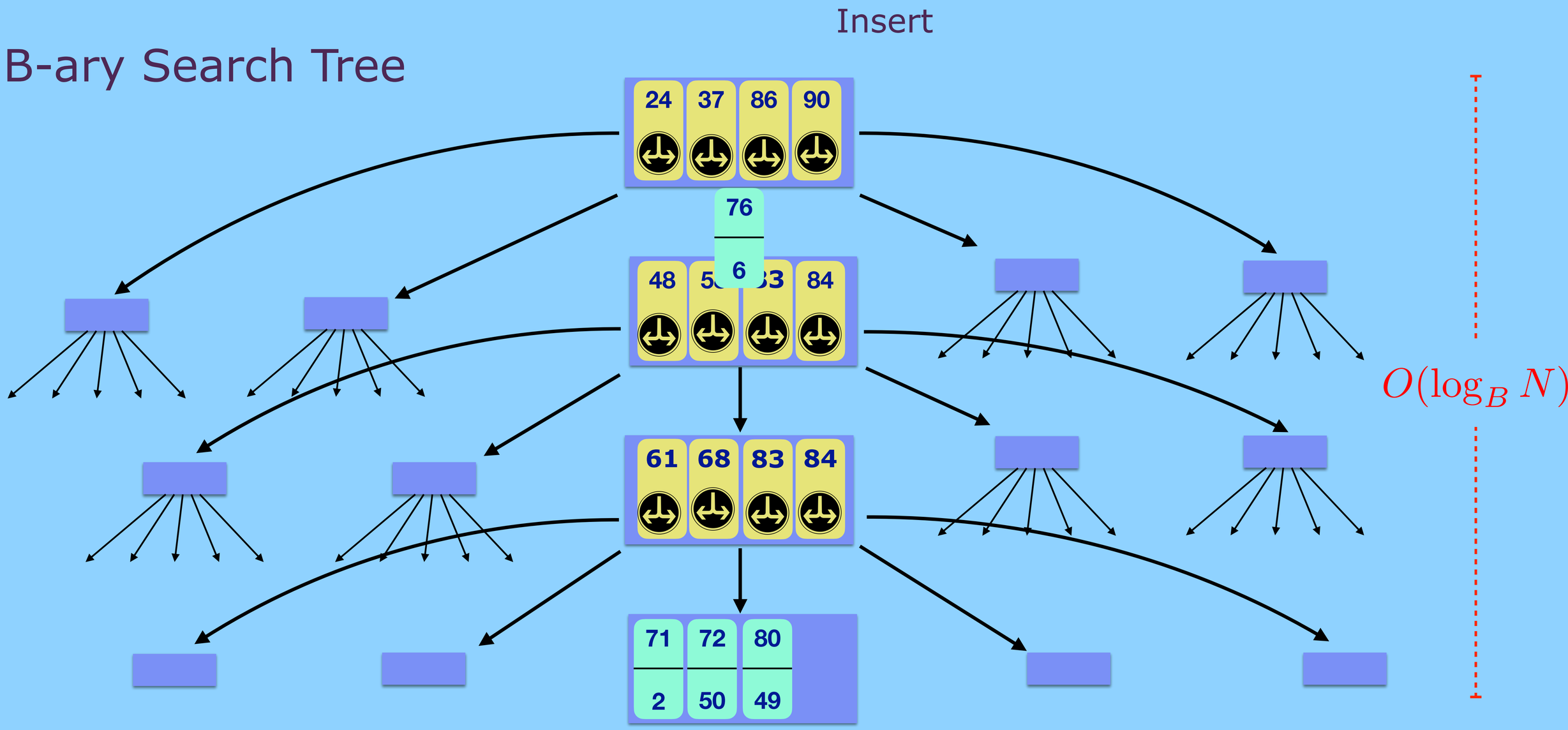
B-Trees

B-ary Search Tree



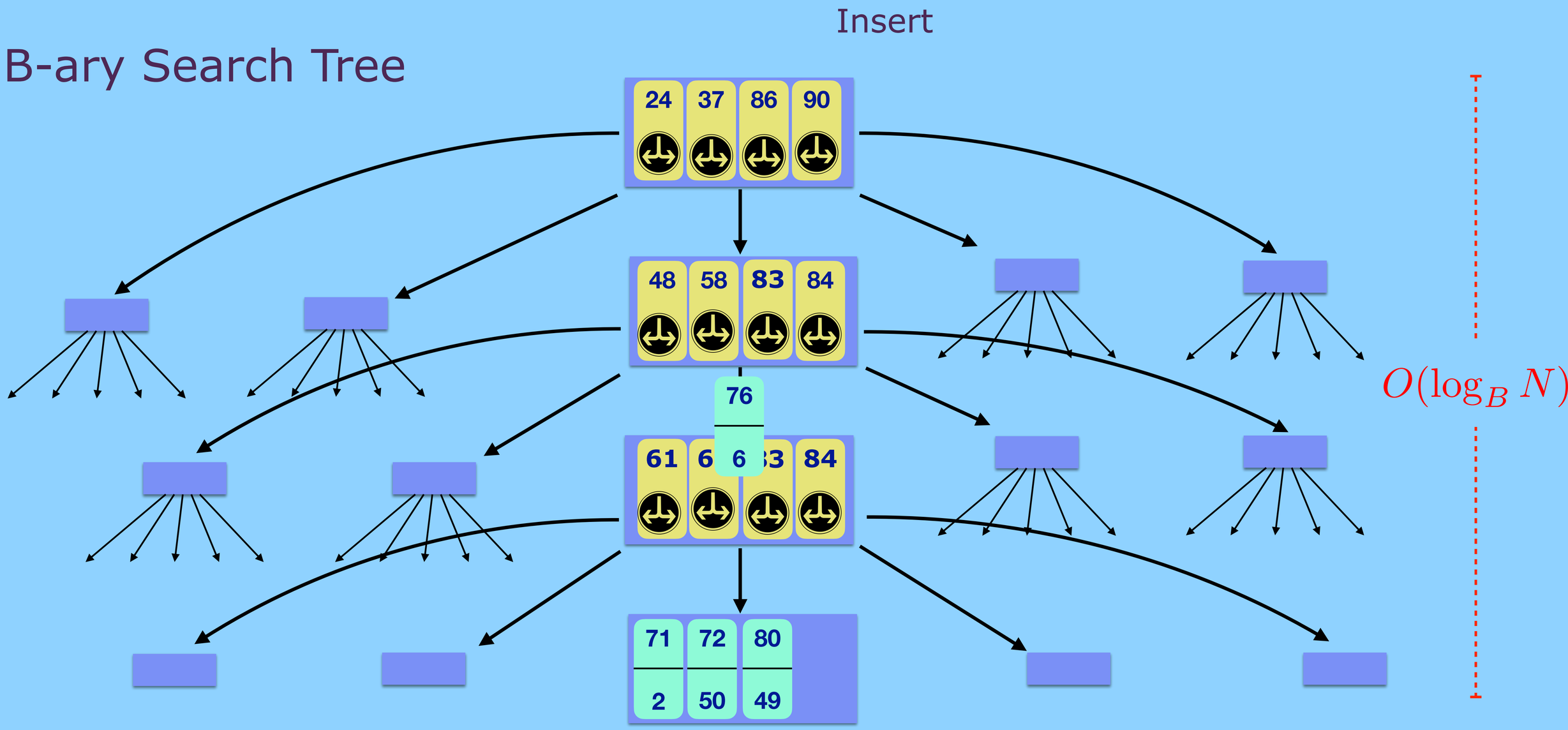
B-Trees

B-ary Search Tree



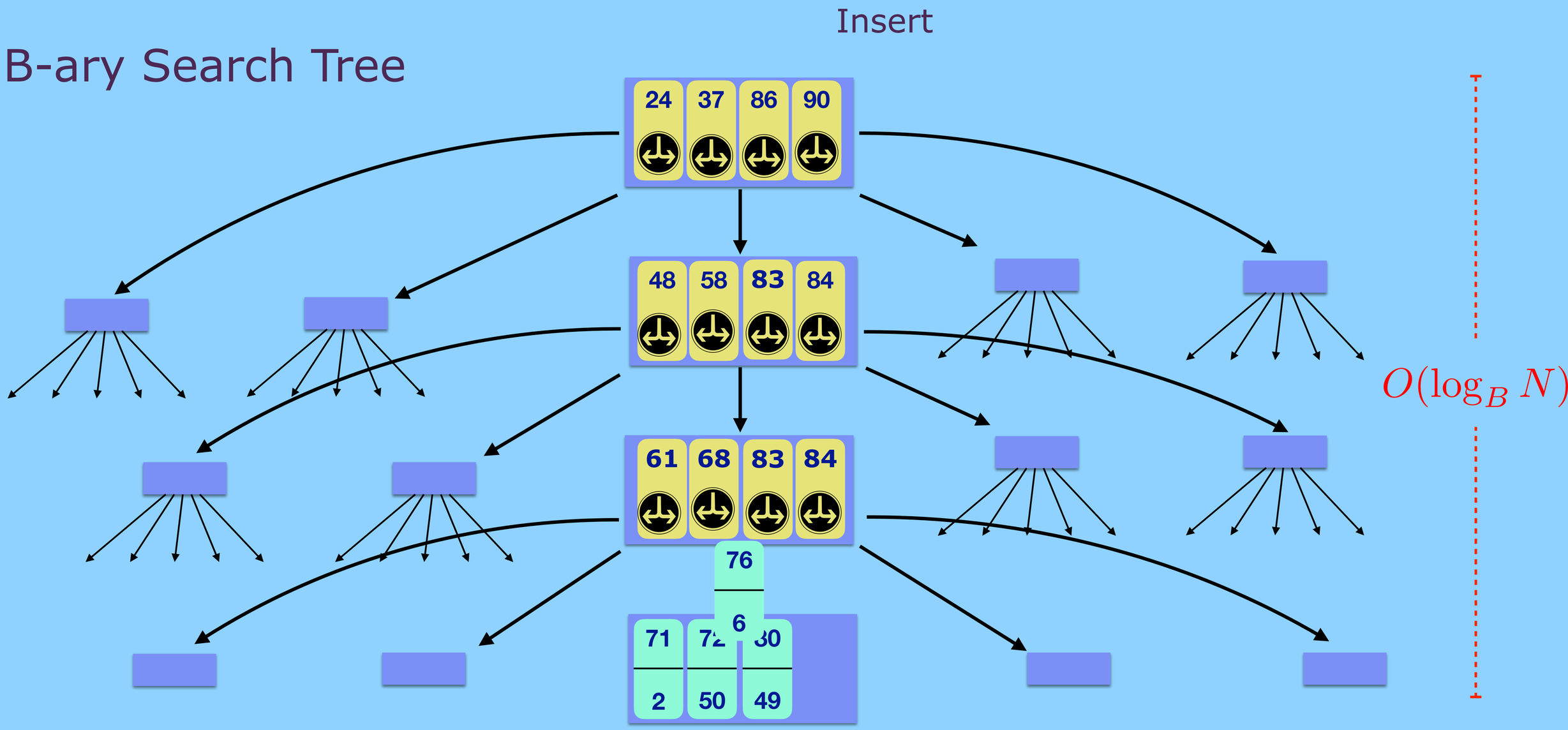
B-Trees

B-ary Search Tree



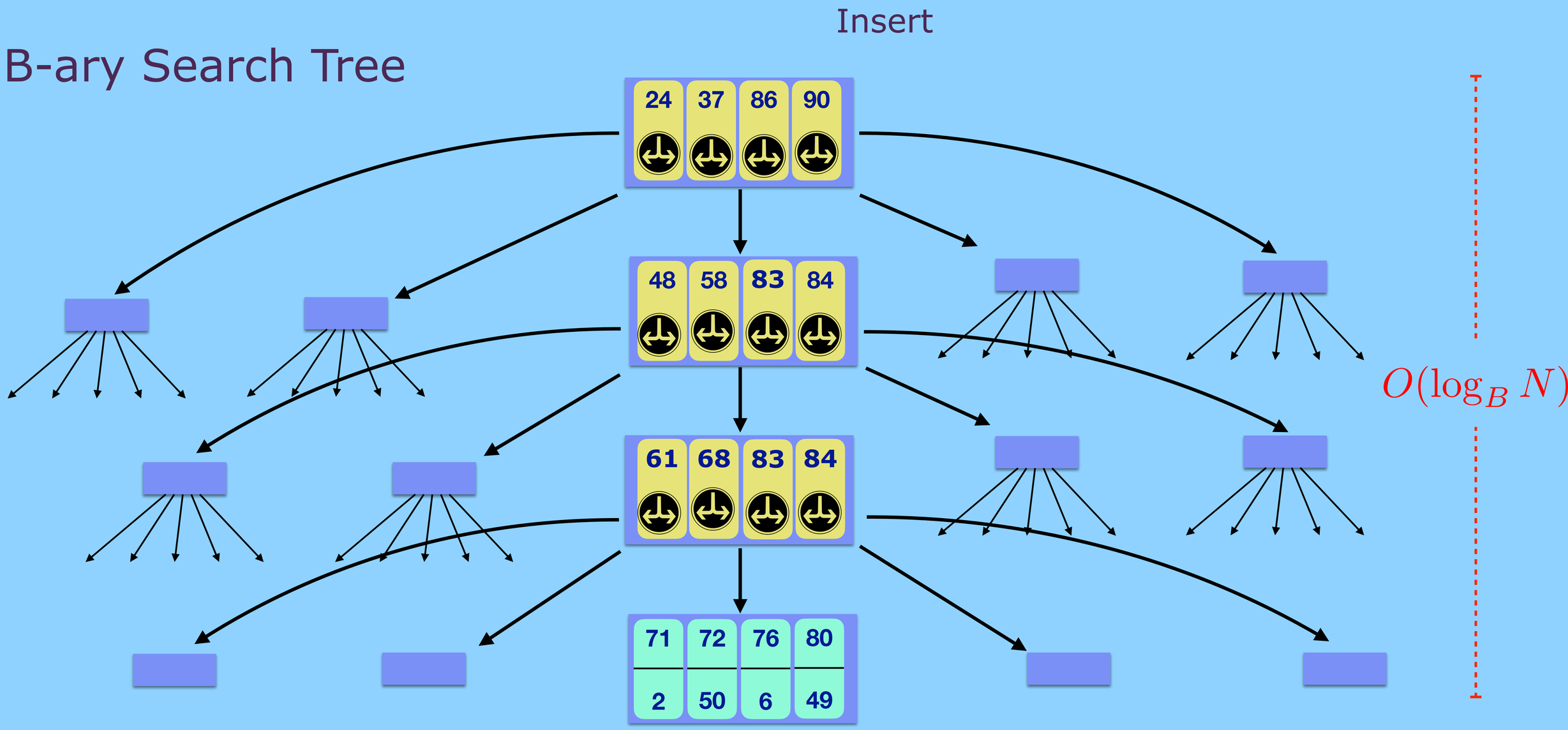
B-Trees

B-ary Search Tree



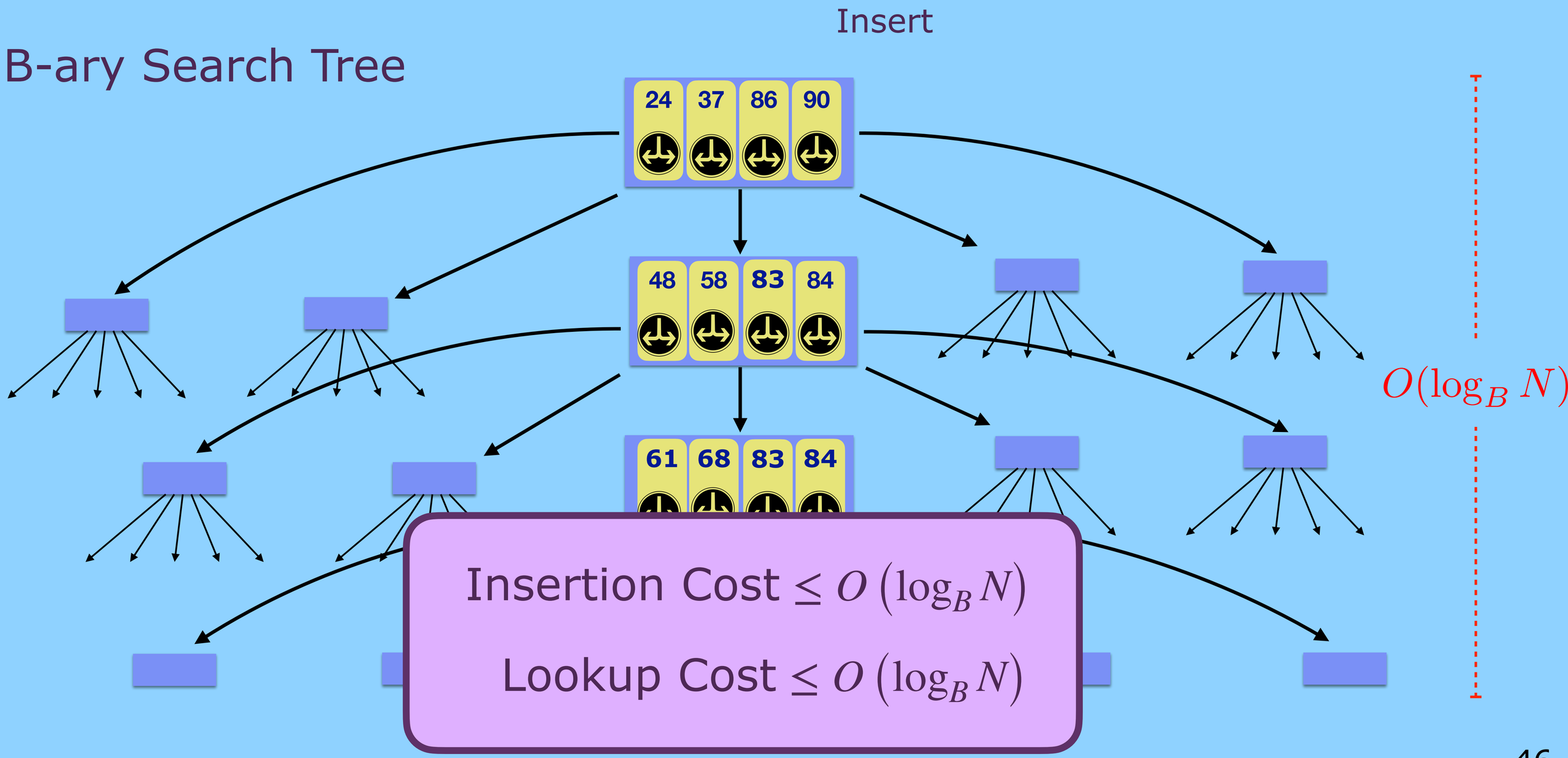
B-Trees

B-ary Search Tree



B-Trees

B-ary Search Tree



Insertion Cost $\leq O\left(\log_B N\right)$

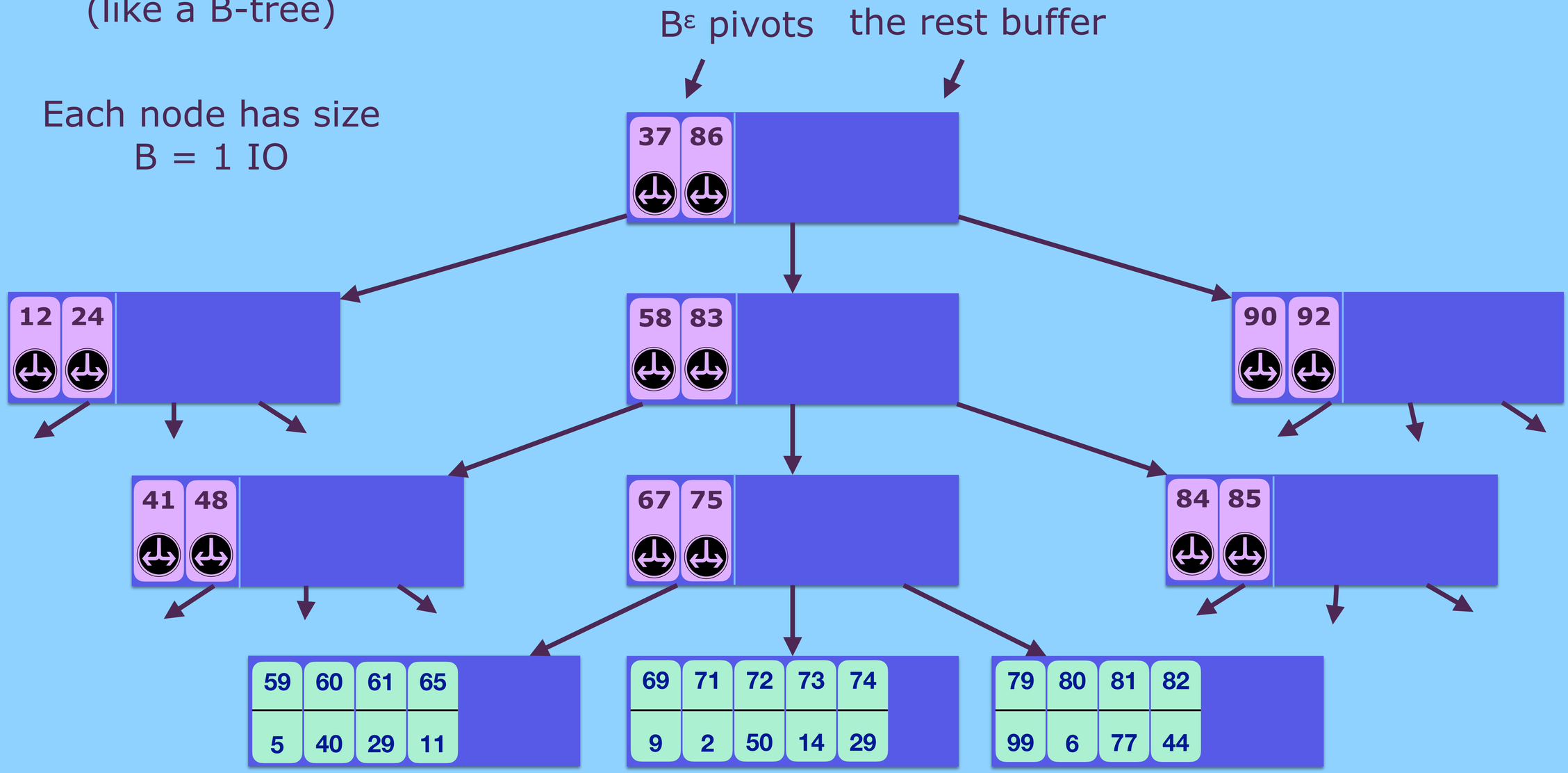
Lookup Cost $\leq O\left(\log_B N\right)$

Bε-Trees

Bε-Trees

A Bε-tree is a search tree
(like a B-tree)

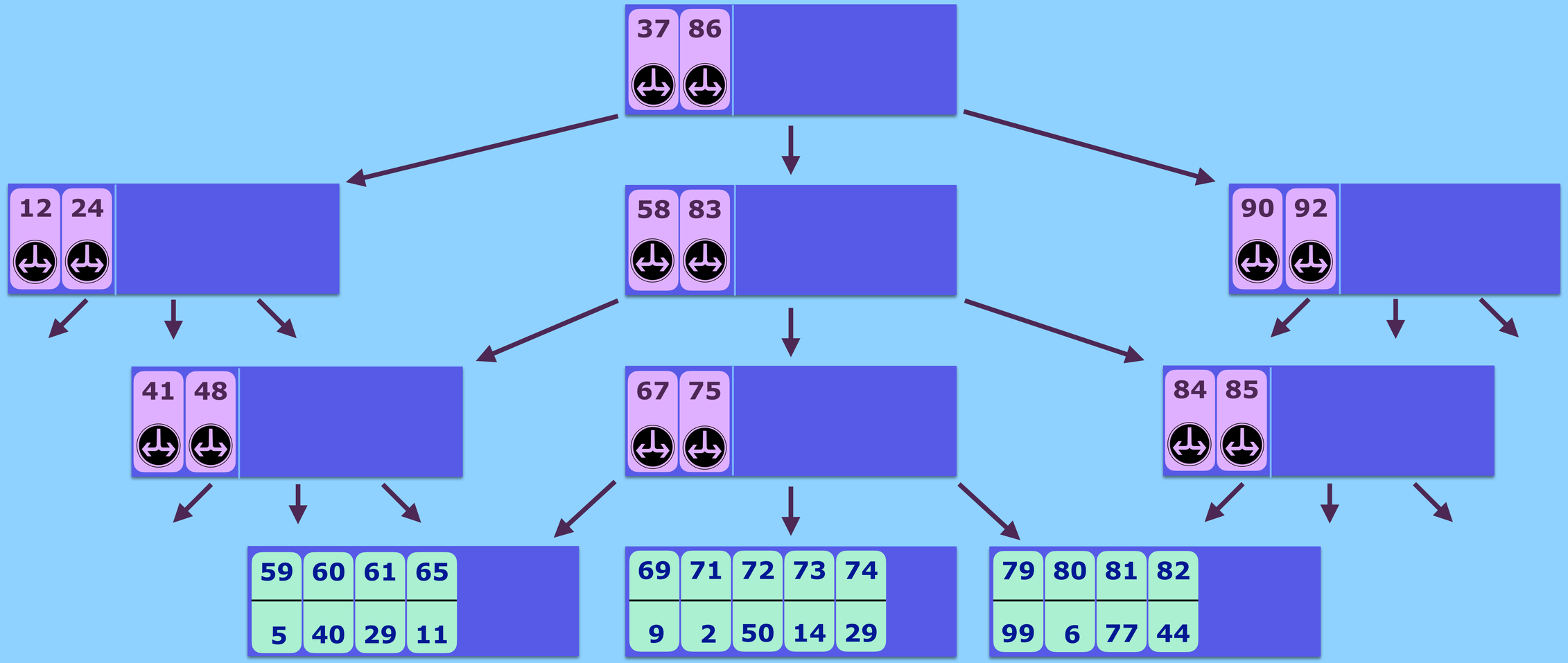
Each node has size
B = 1 IO



Insertions in B^{ε}-Trees

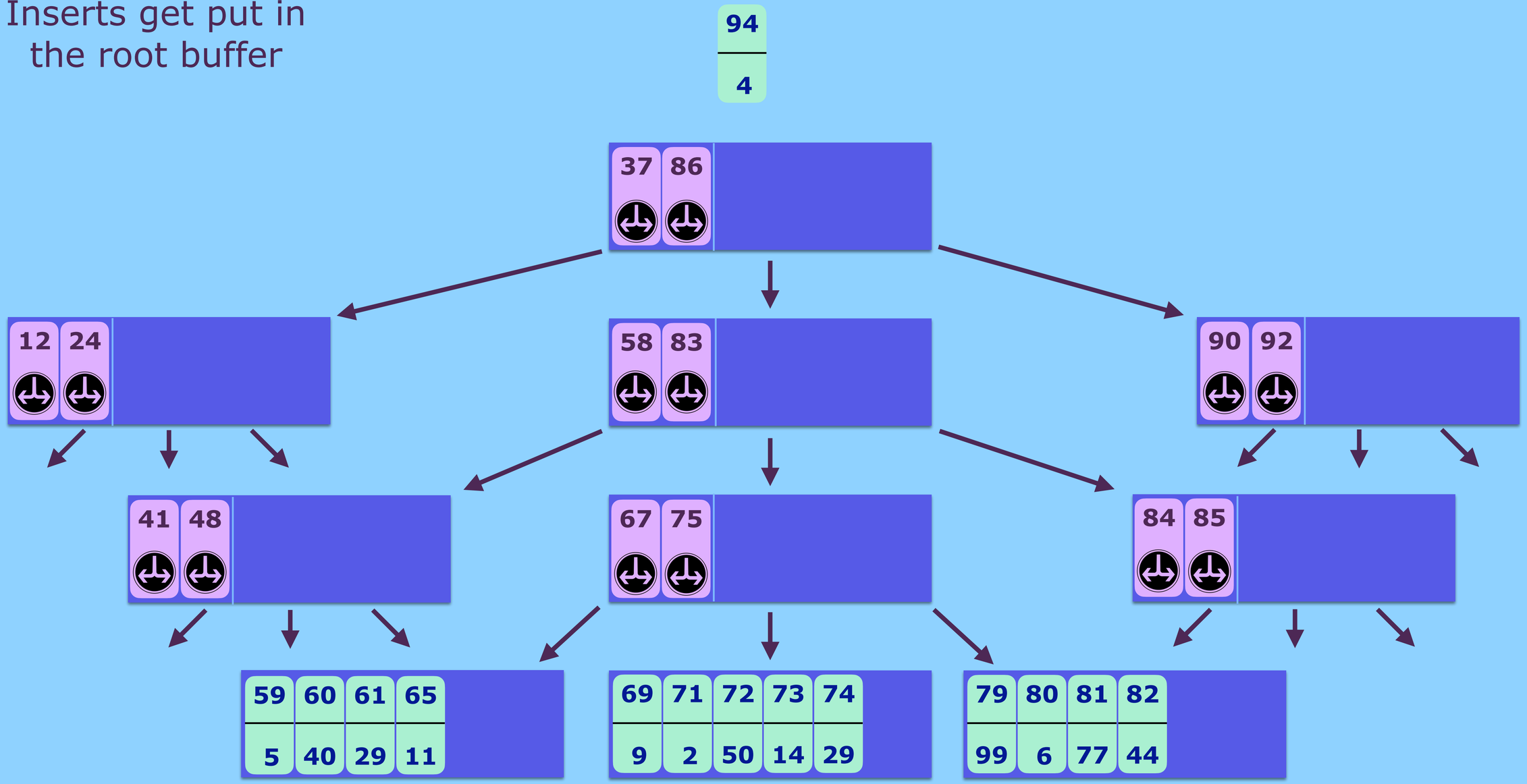
Bε-Trees

Inserts get put in the root buffer



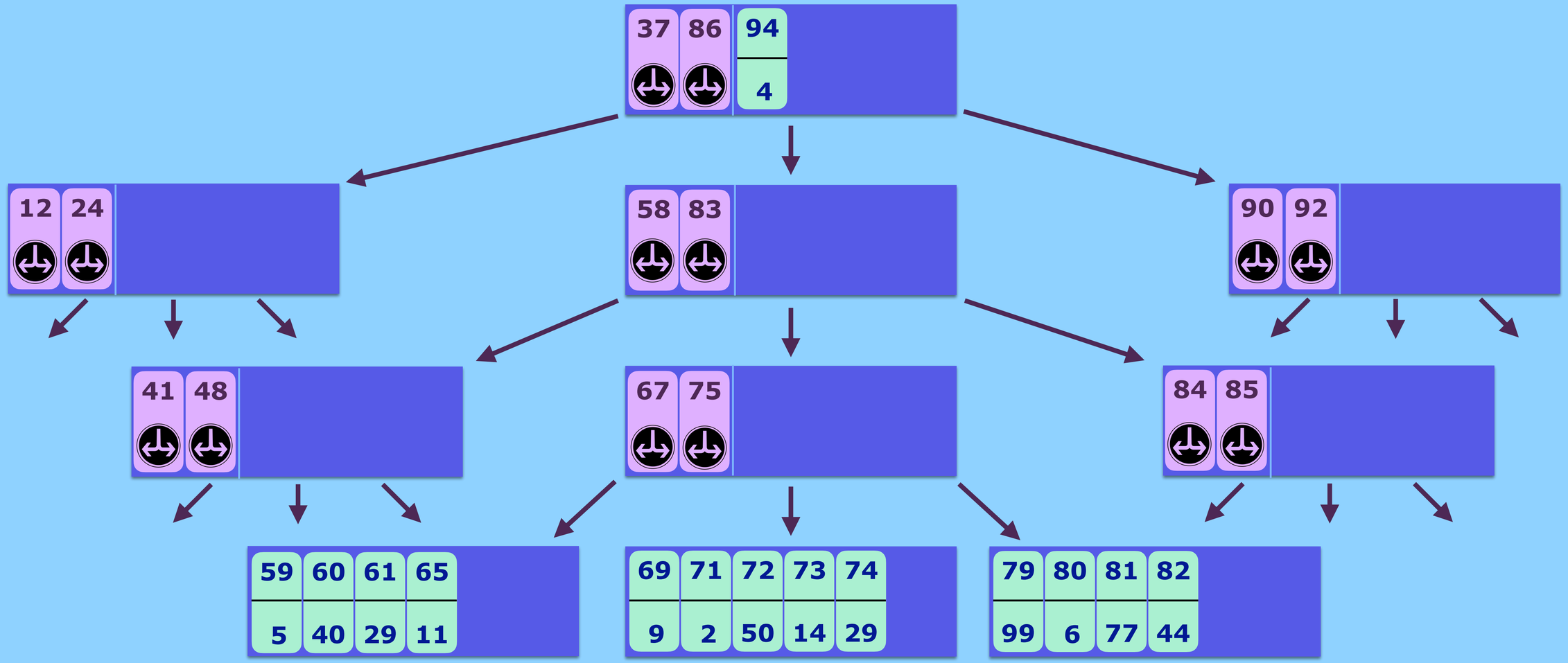
Bε-Trees

Inserts get put in the root buffer



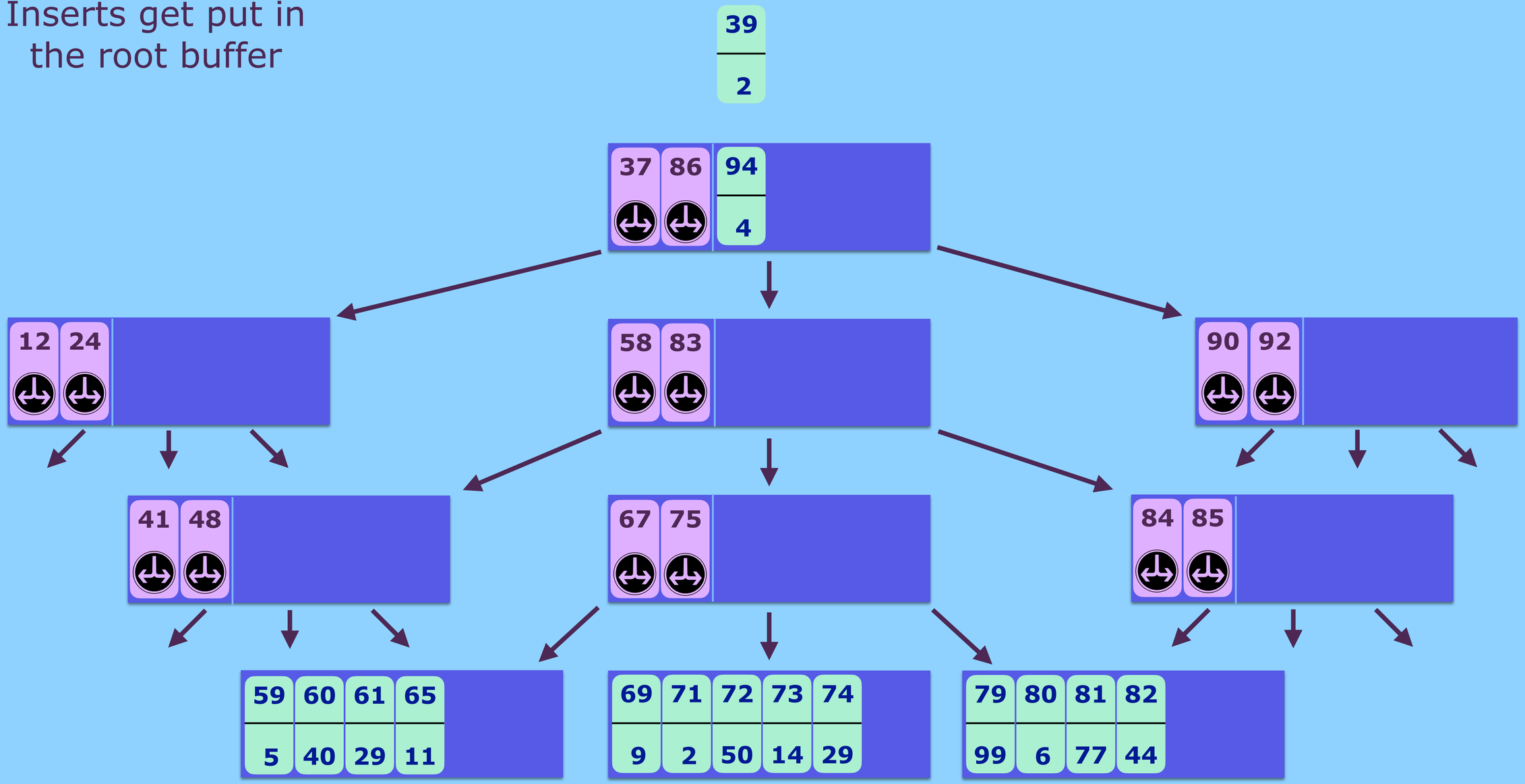
Bε-Trees

Inserts get put in the root buffer



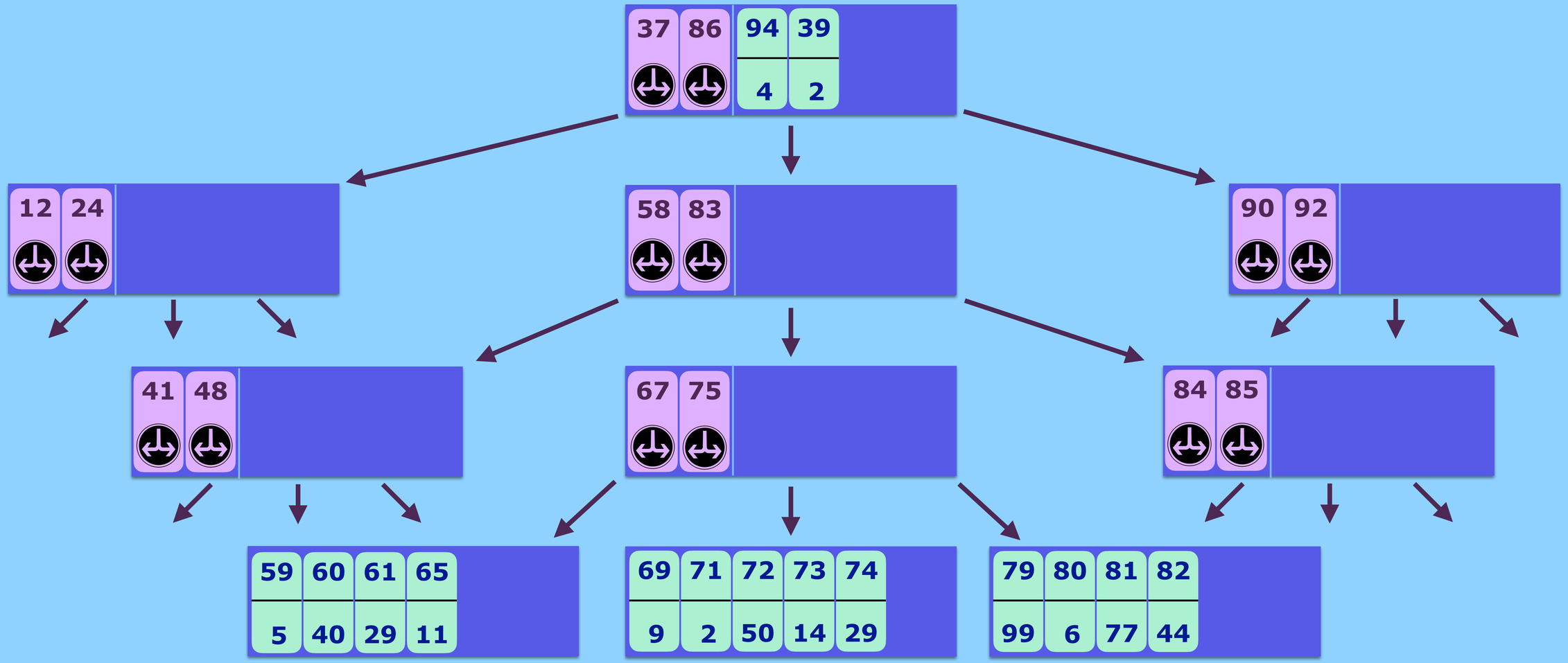
Bε-Trees

Inserts get put in the root buffer



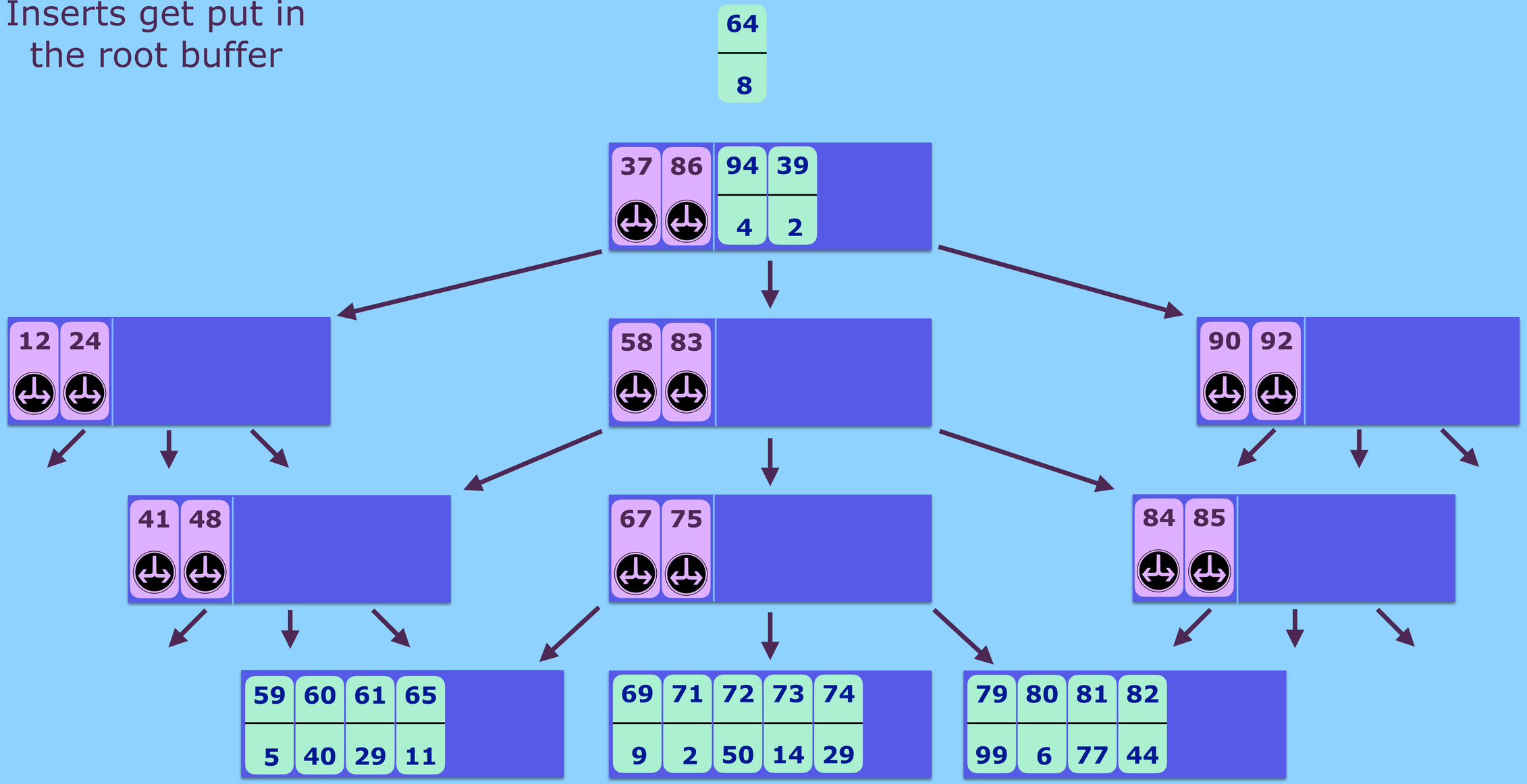
Bε-Trees

Inserts get put in the root buffer



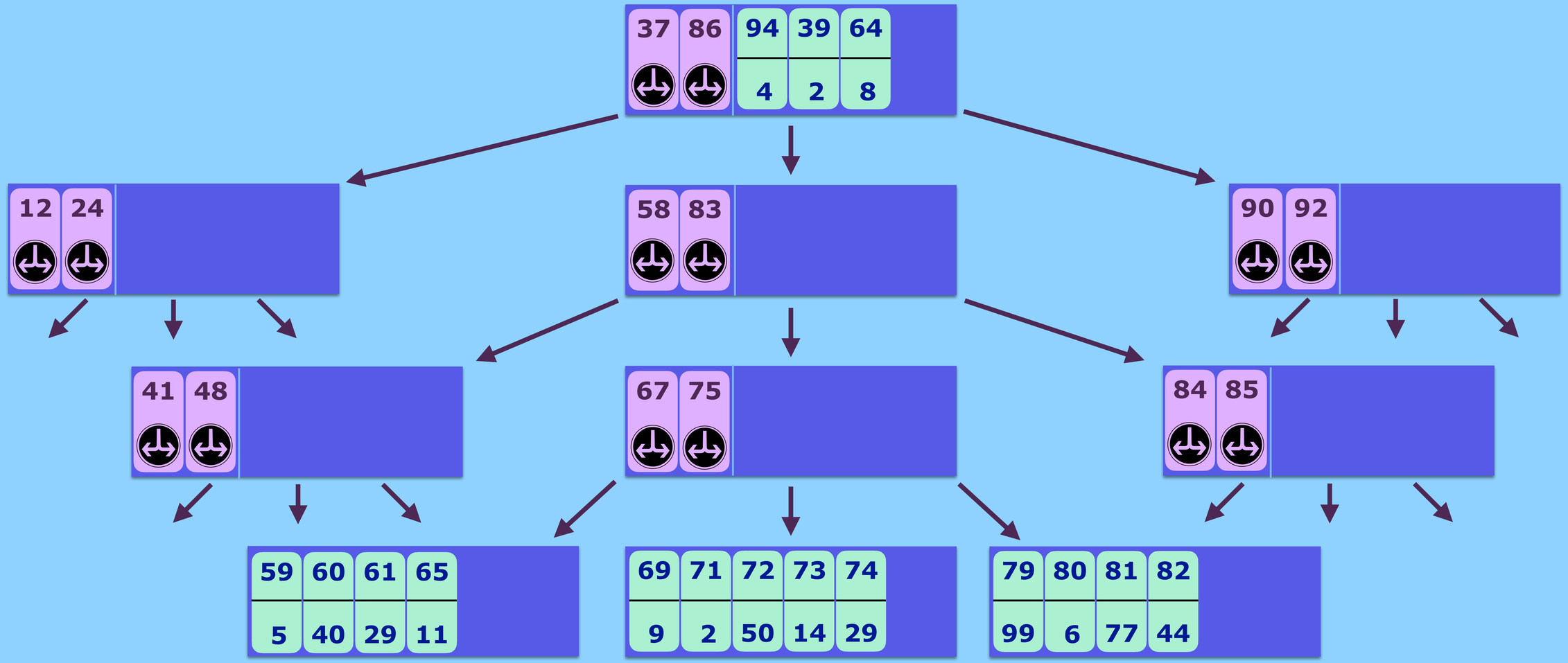
Bε-Trees

Inserts get put in the root buffer



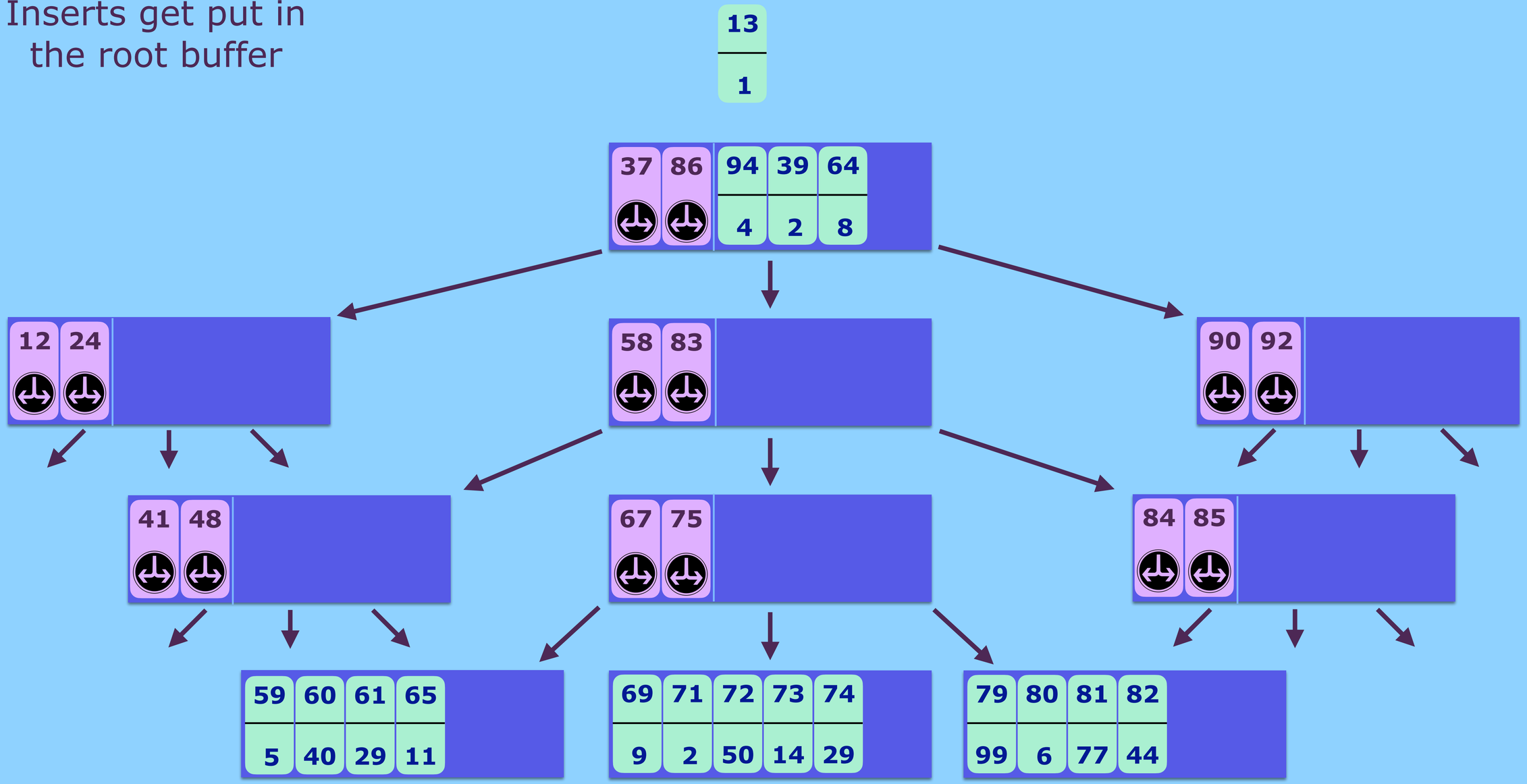
Bε-Trees

Inserts get put in the root buffer



Bε-Trees

Inserts get put in the root buffer

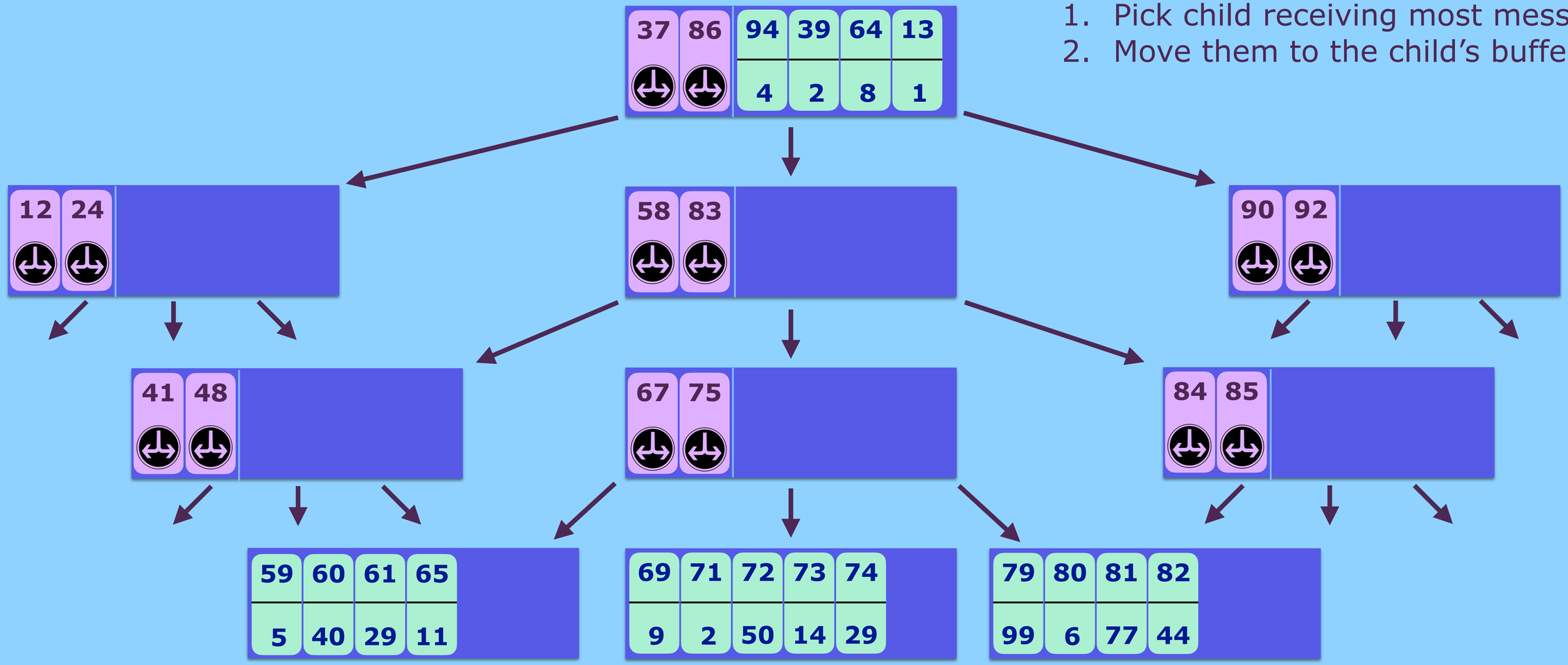


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

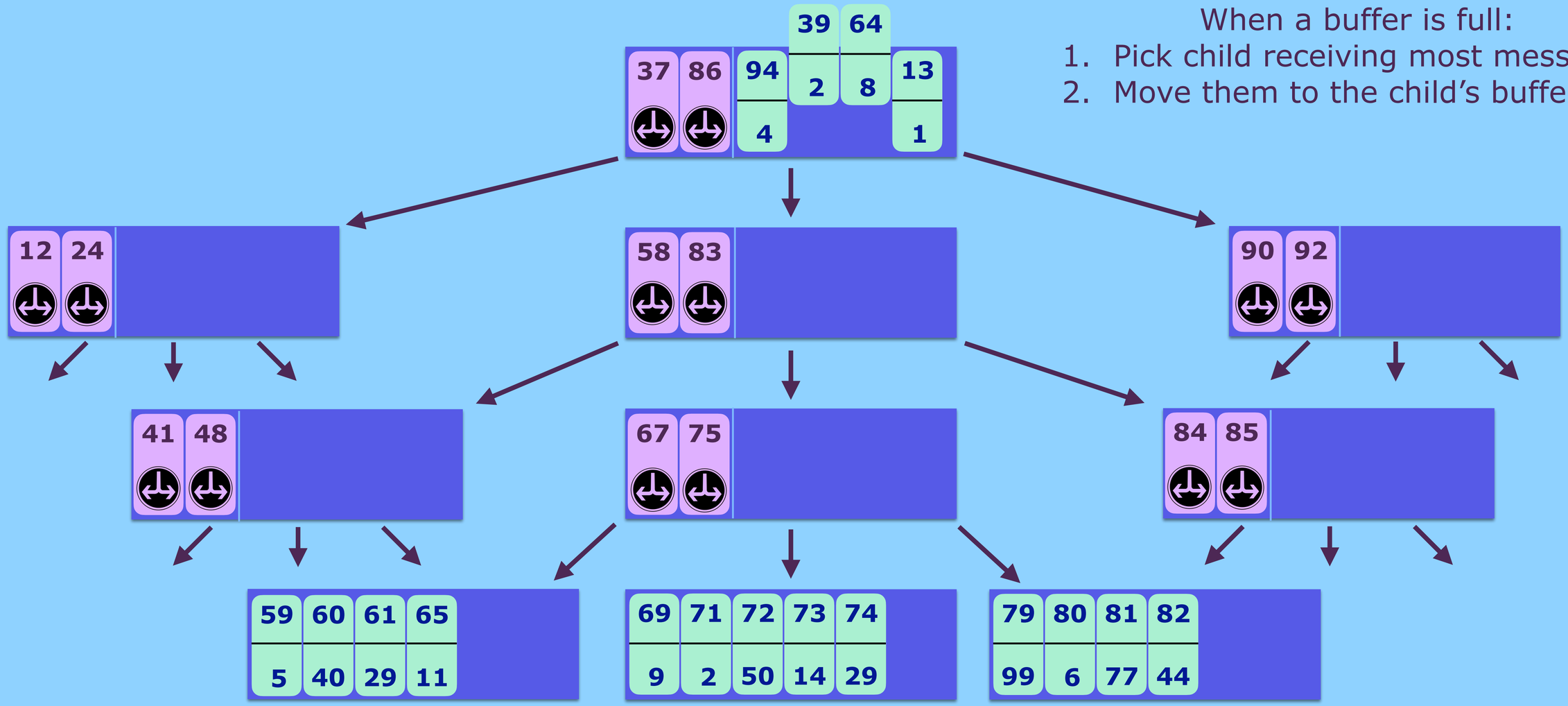


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

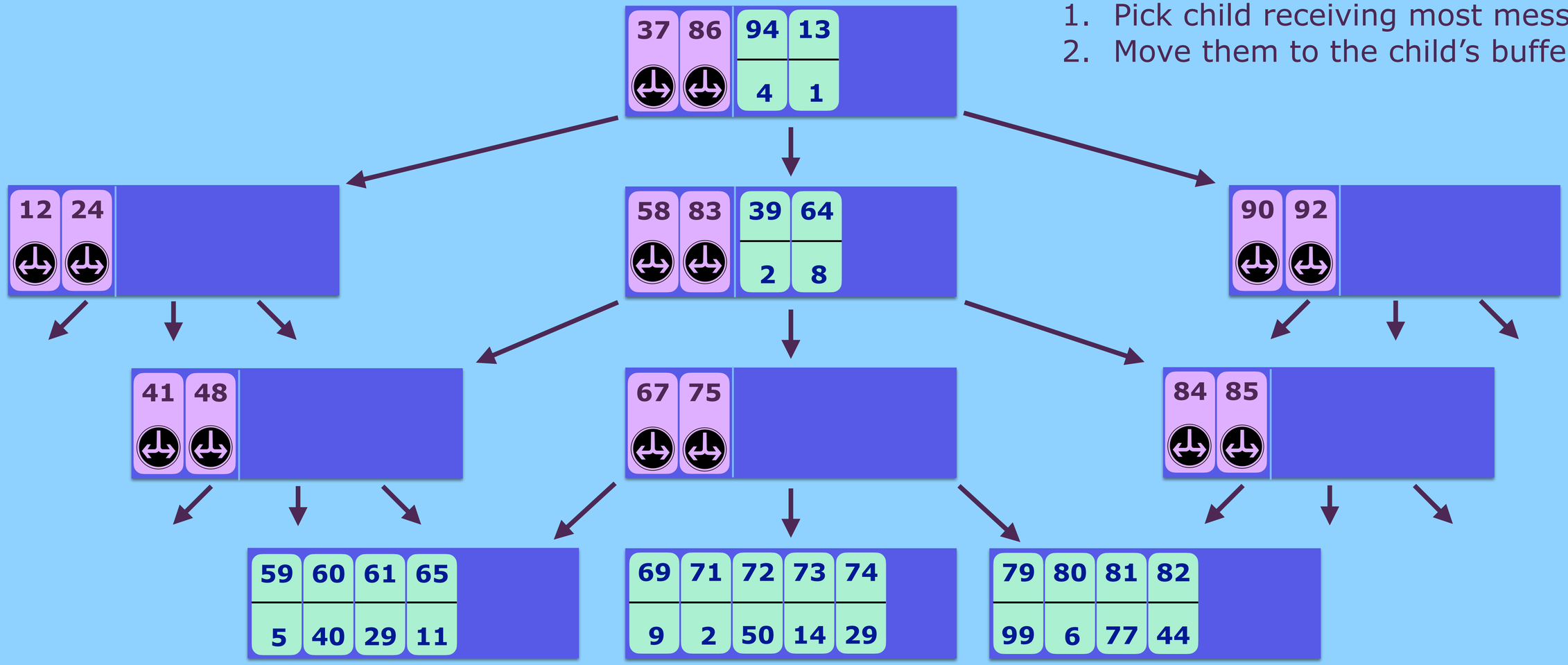


Bε-Trees

Inserts get put in the root buffer

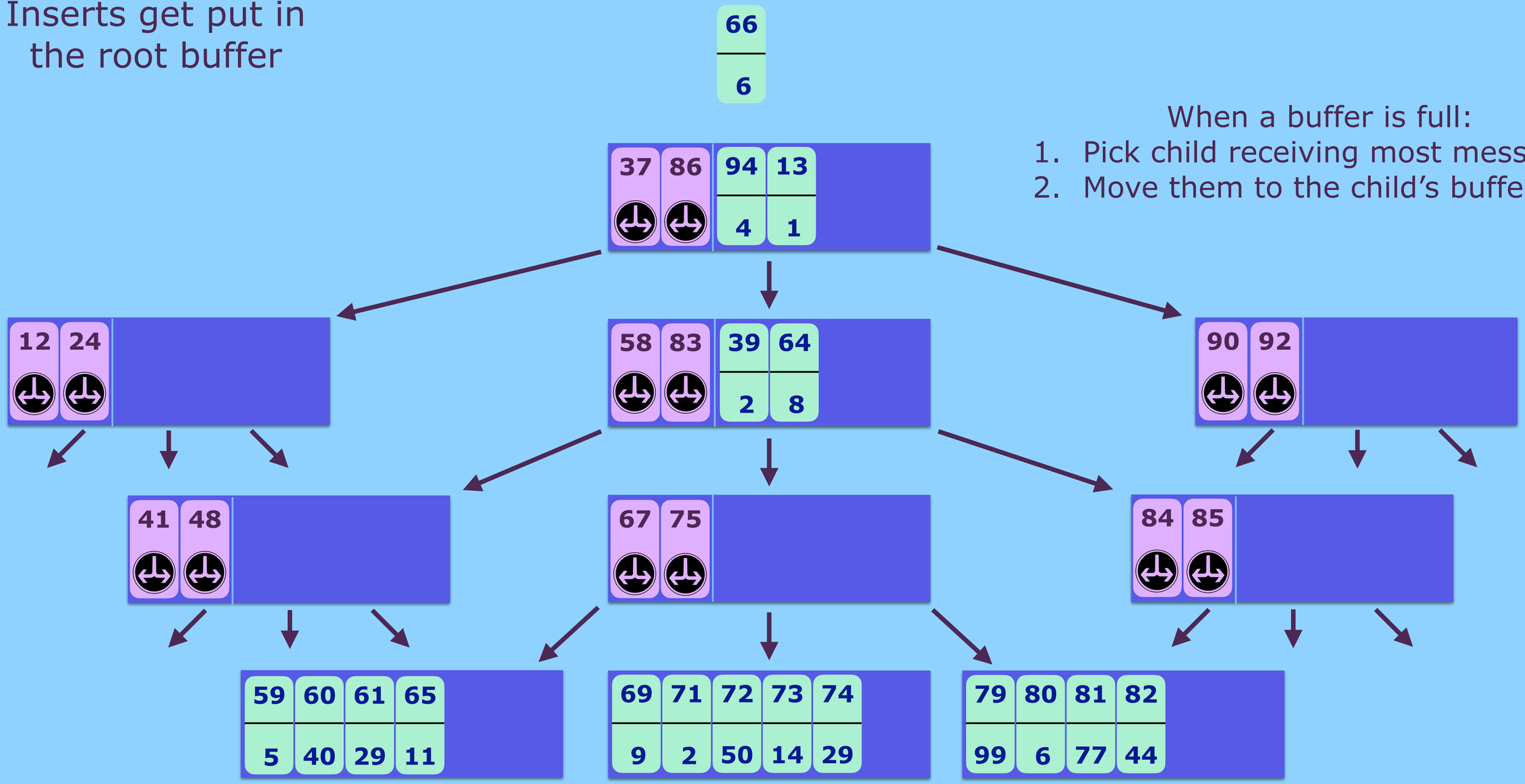
When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer



Bε-Trees

Inserts get put in the root buffer



When a buffer is full:

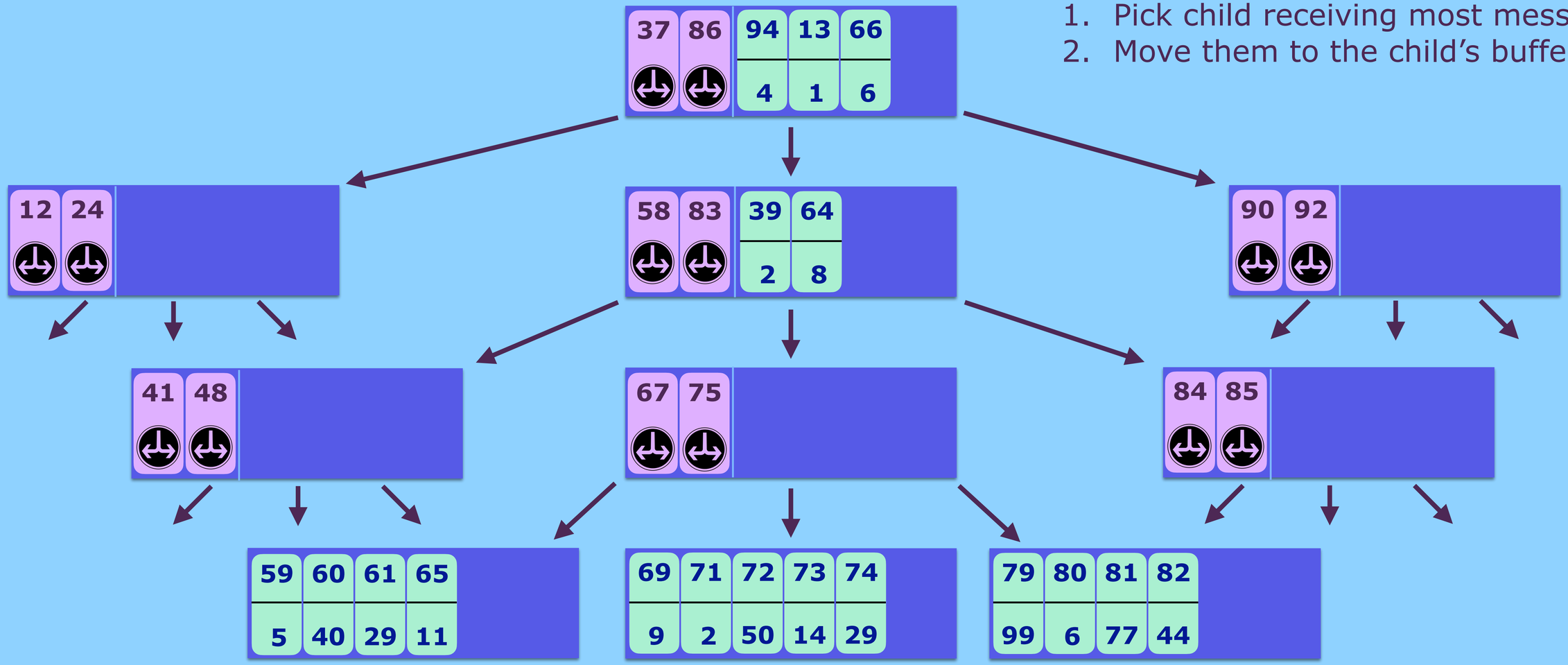
1. Pick child receiving most messages
2. Move them to the child's buffer

Bε-Trees

Inserts get put in the root buffer

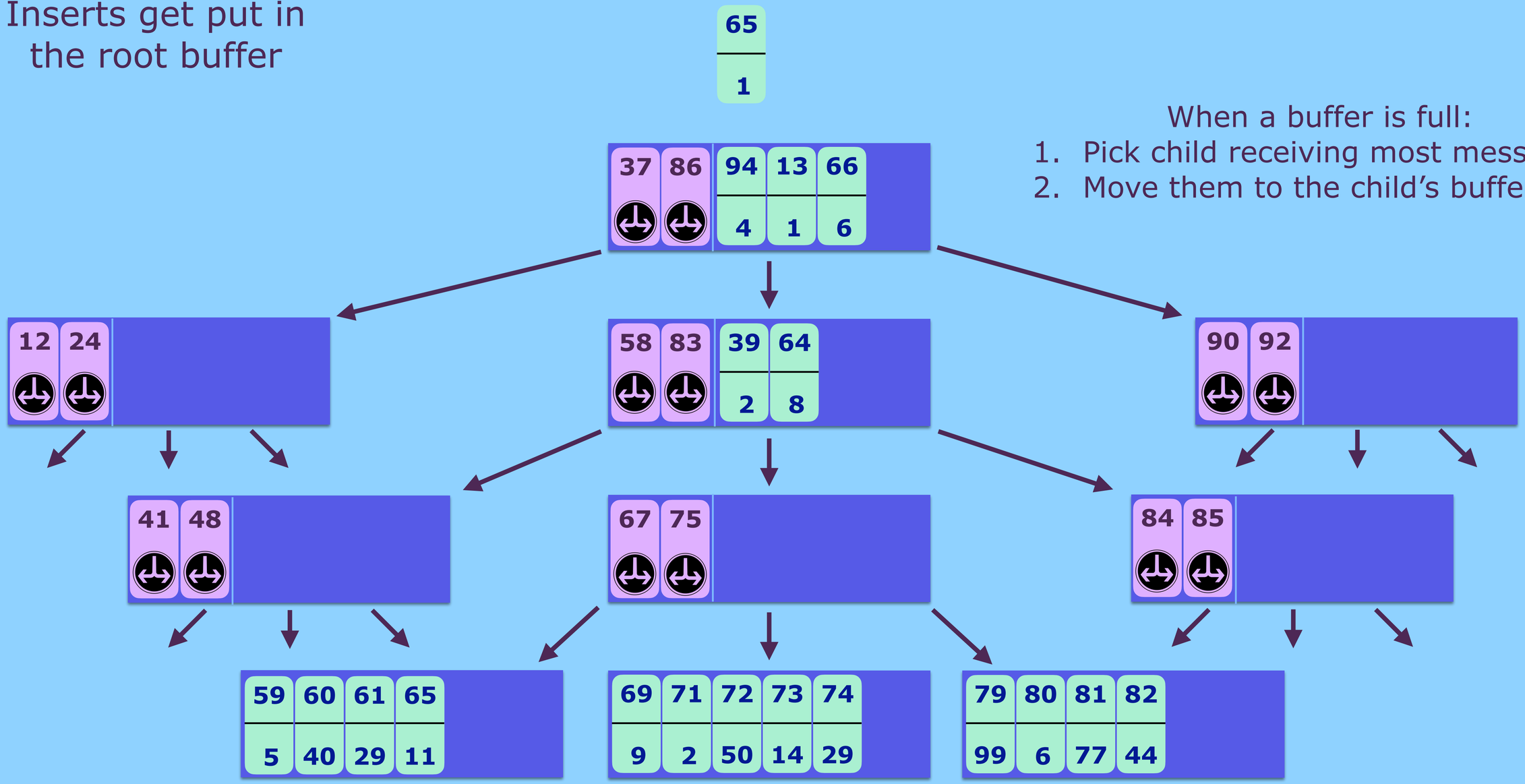
When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer



Bε-Trees

Inserts get put in the root buffer



When a buffer is full:

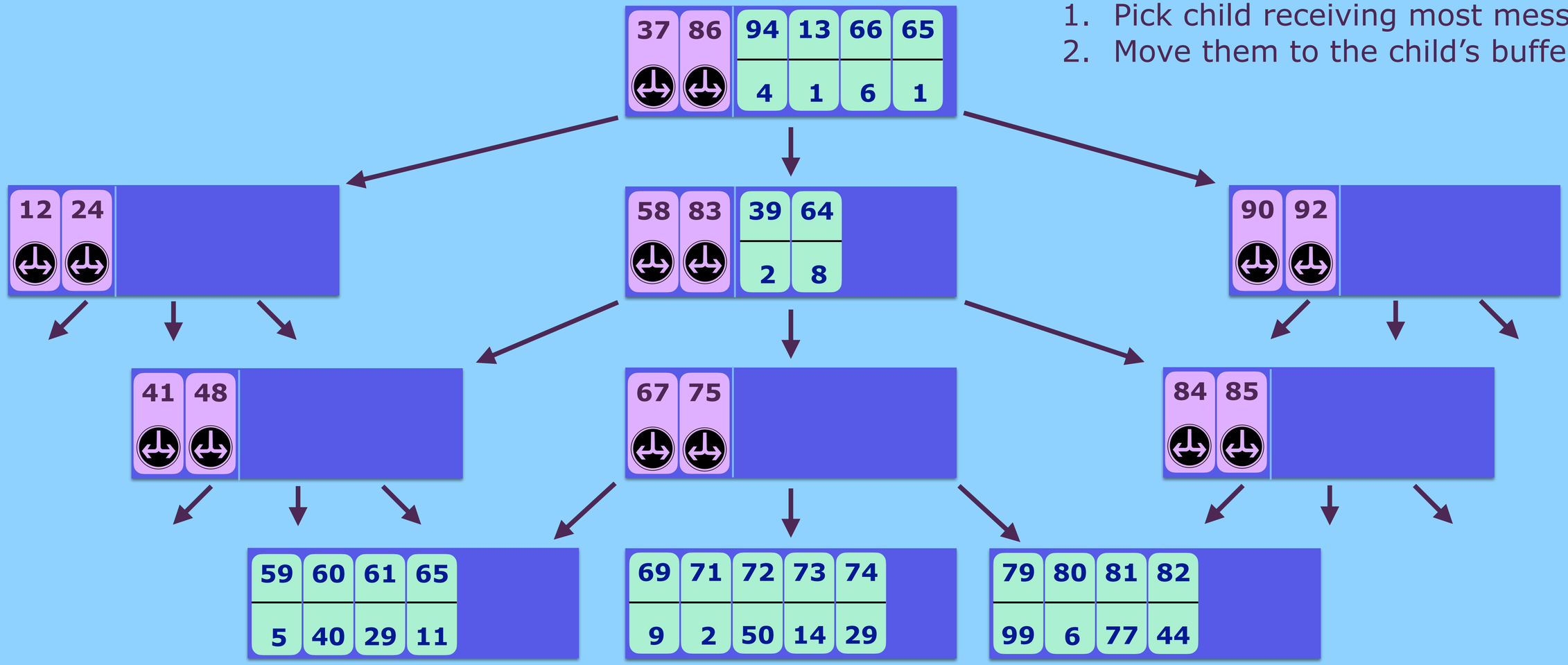
1. Pick child receiving most messages
2. Move them to the child's buffer

Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

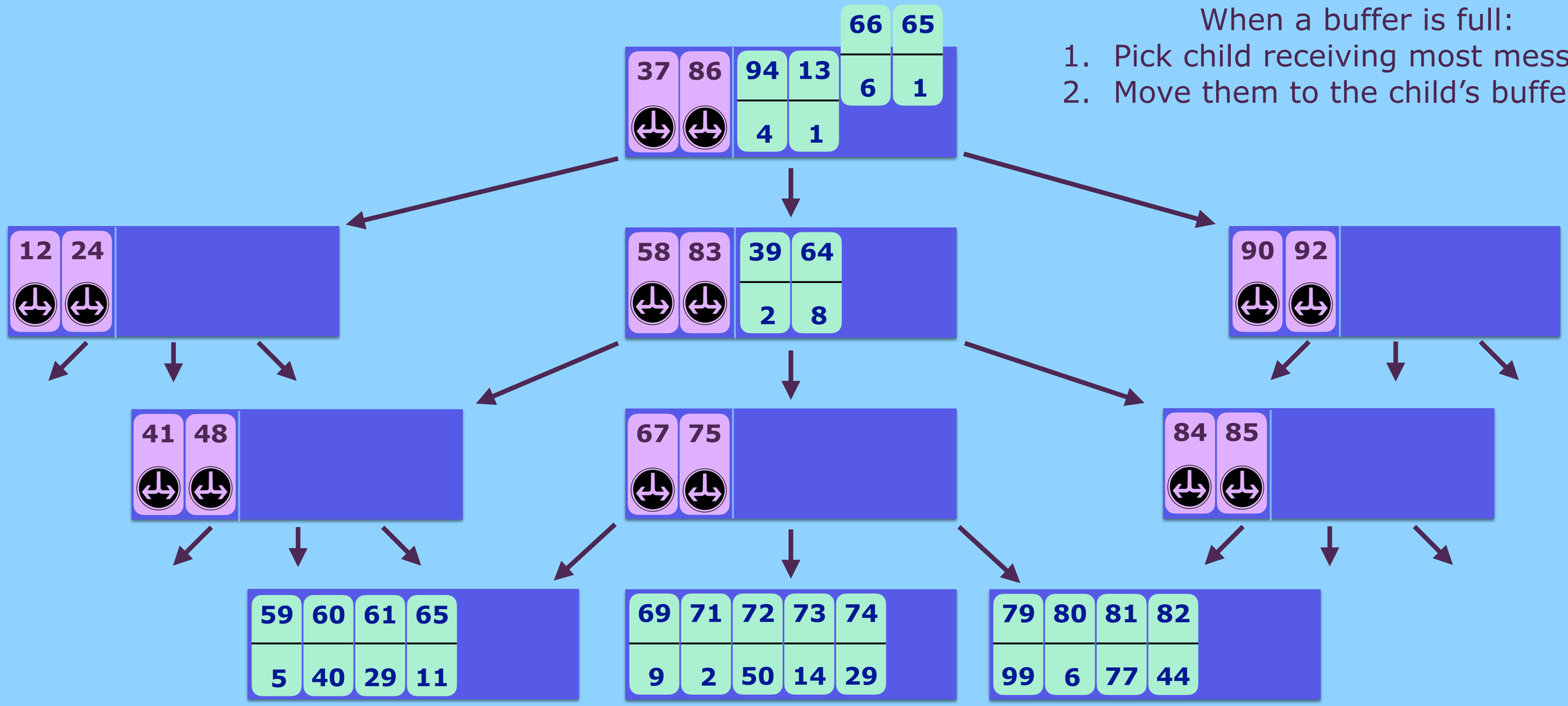


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

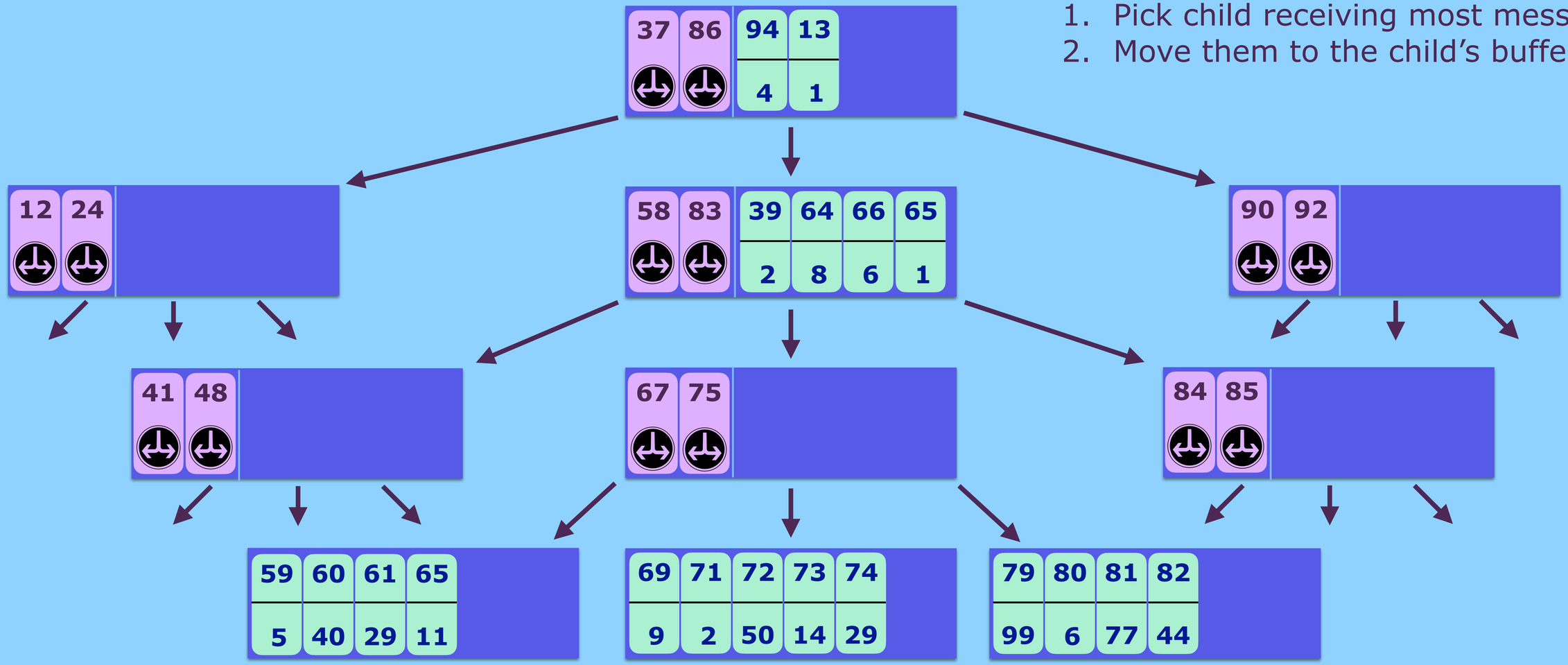


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

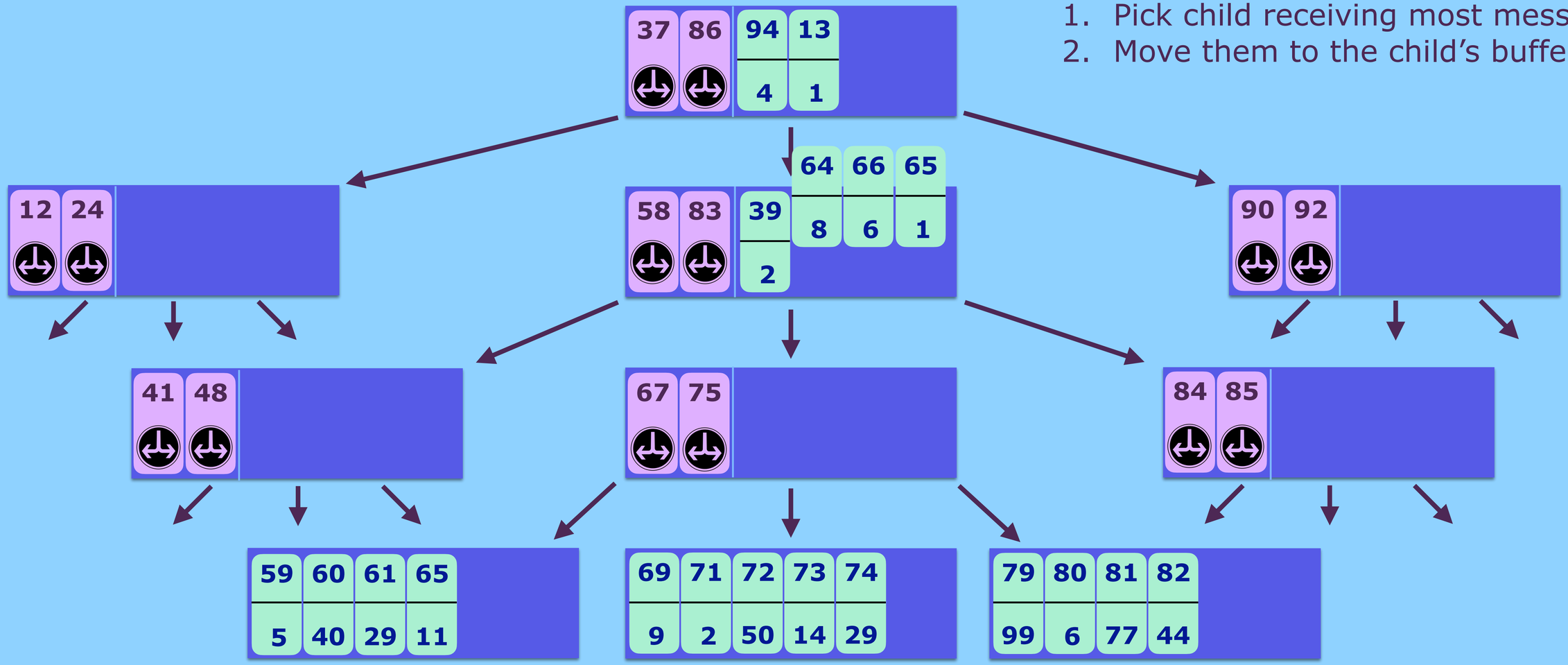


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

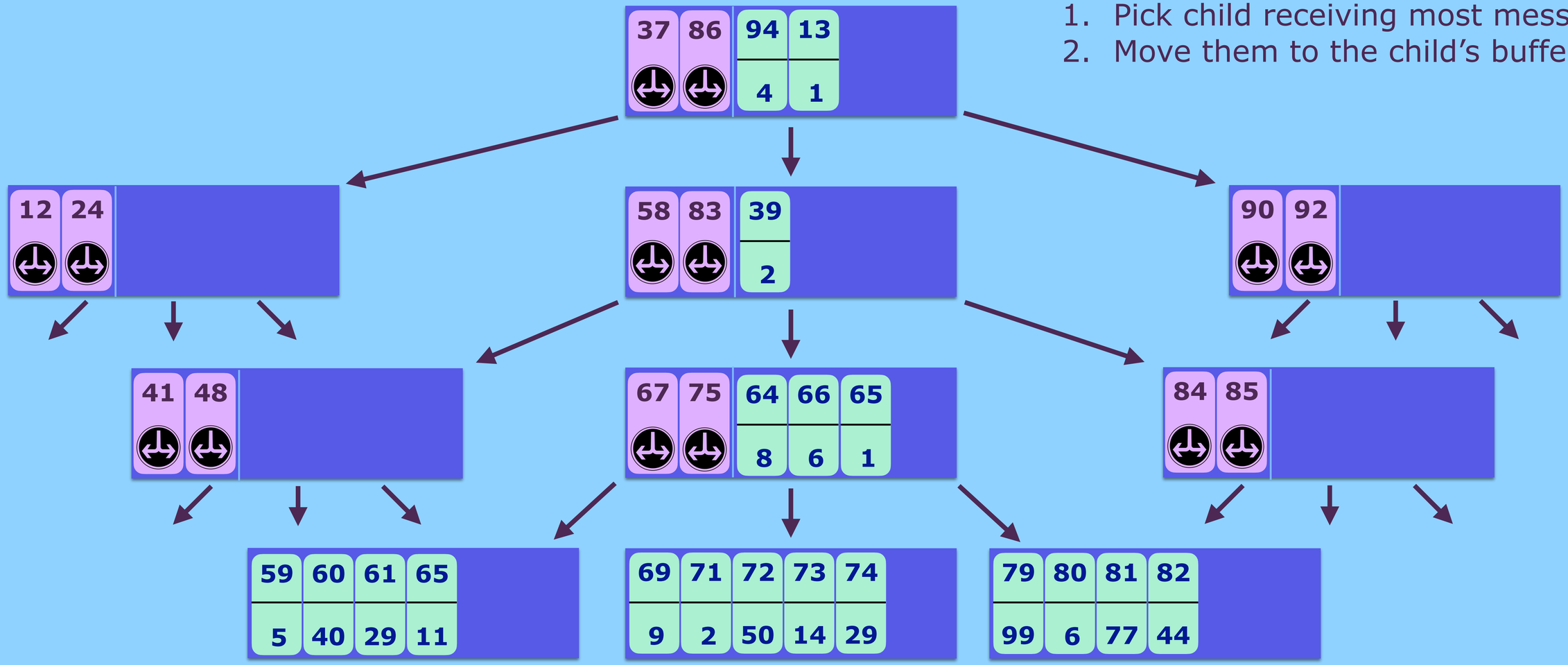


Bε-Trees

Inserts get put in the root buffer

When a buffer is full:

1. Pick child receiving most messages
2. Move them to the child's buffer

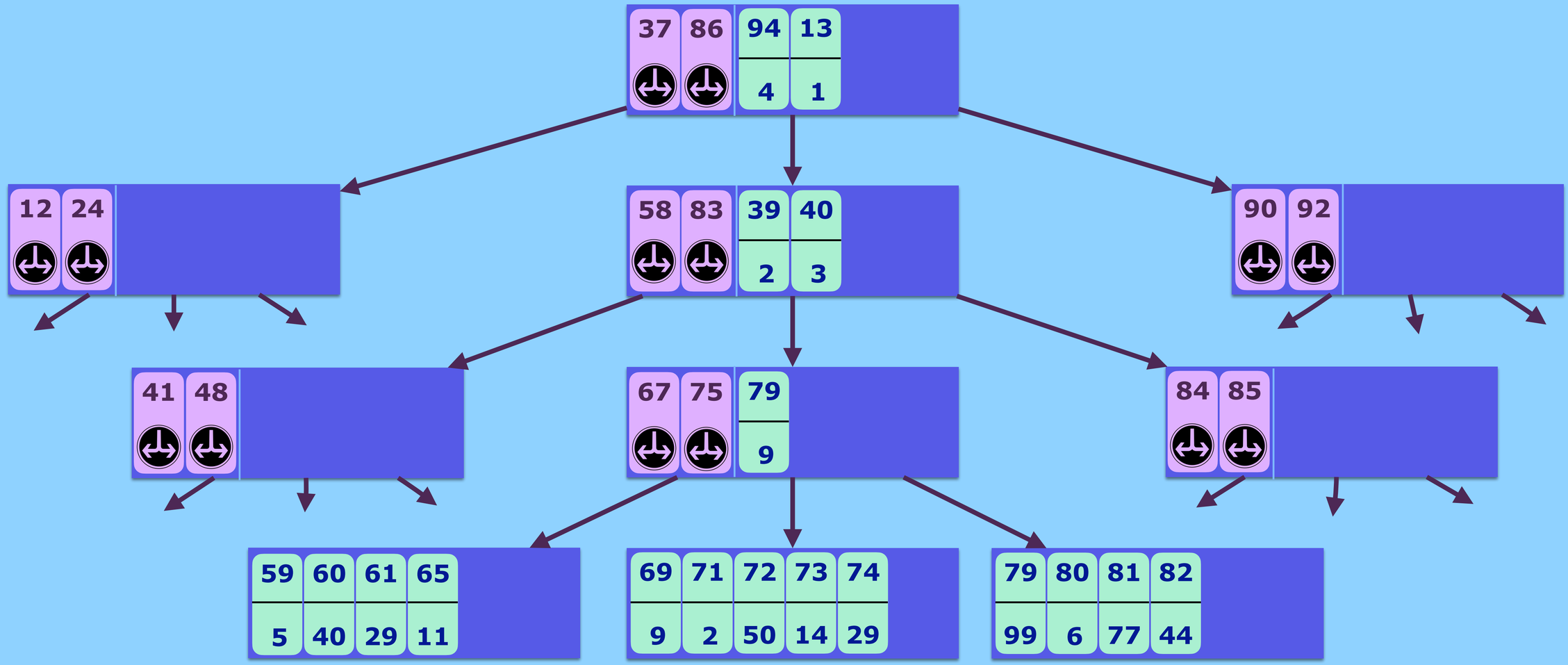


Lookups in Bε-Trees

Bε-Trees

Lookups follow pivots, but check buffers along the way

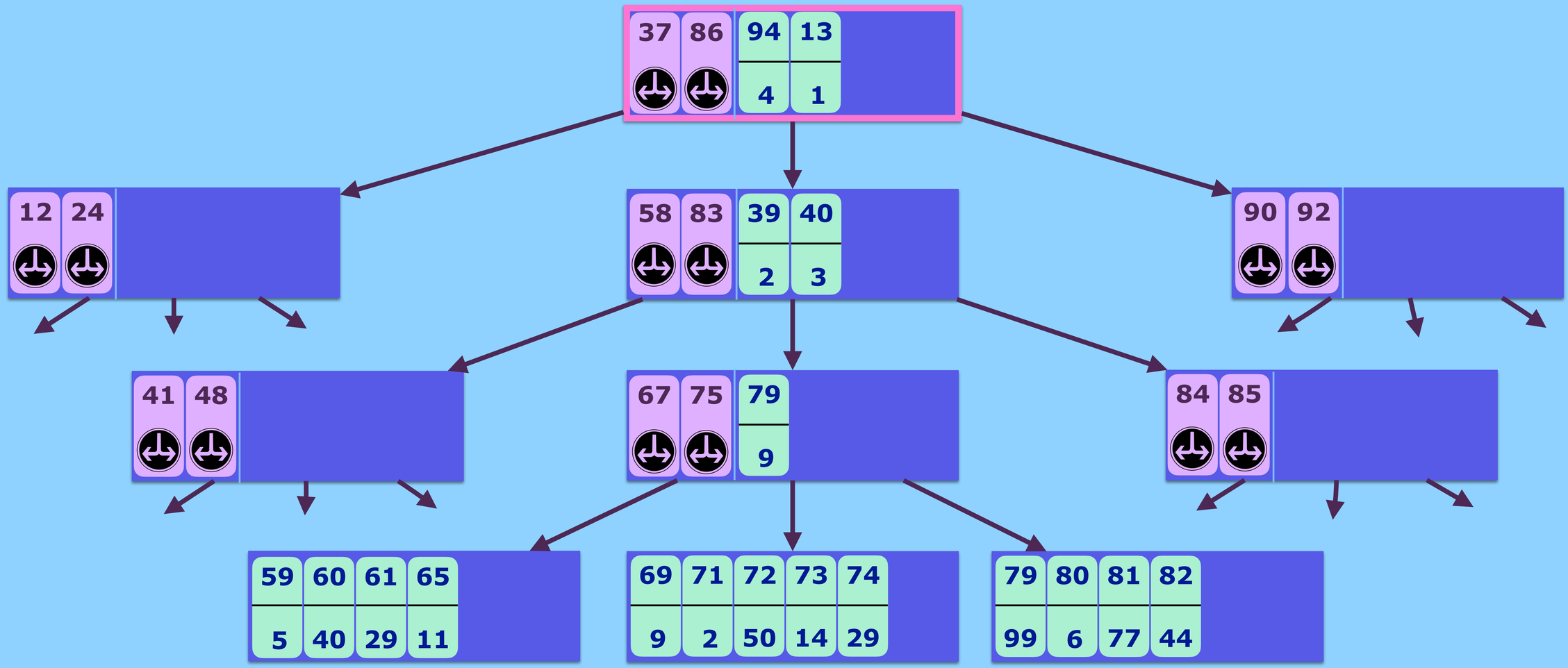
Query(71)



Bε-Trees

Lookups follow pivots, but check buffers along the way

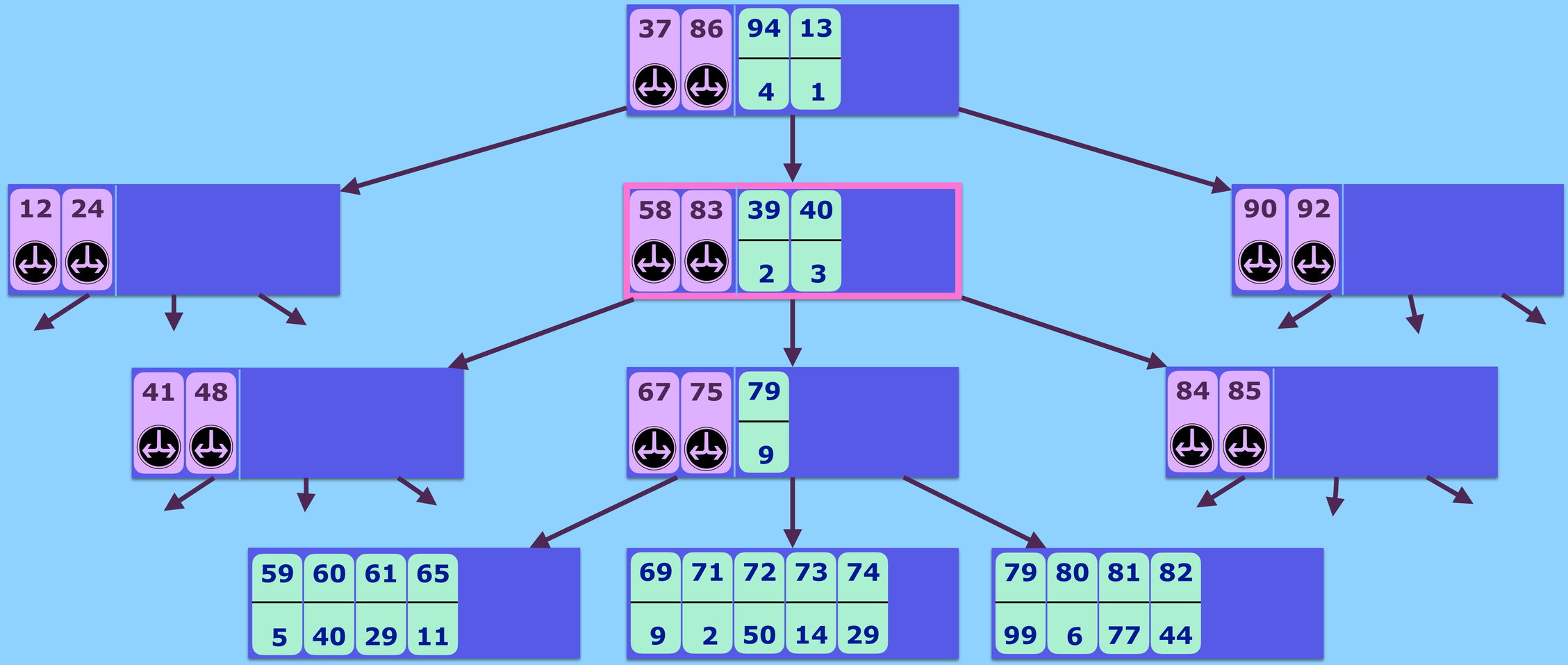
Query(71)



Bε-Trees

Lookups follow pivots, but check buffers along the way

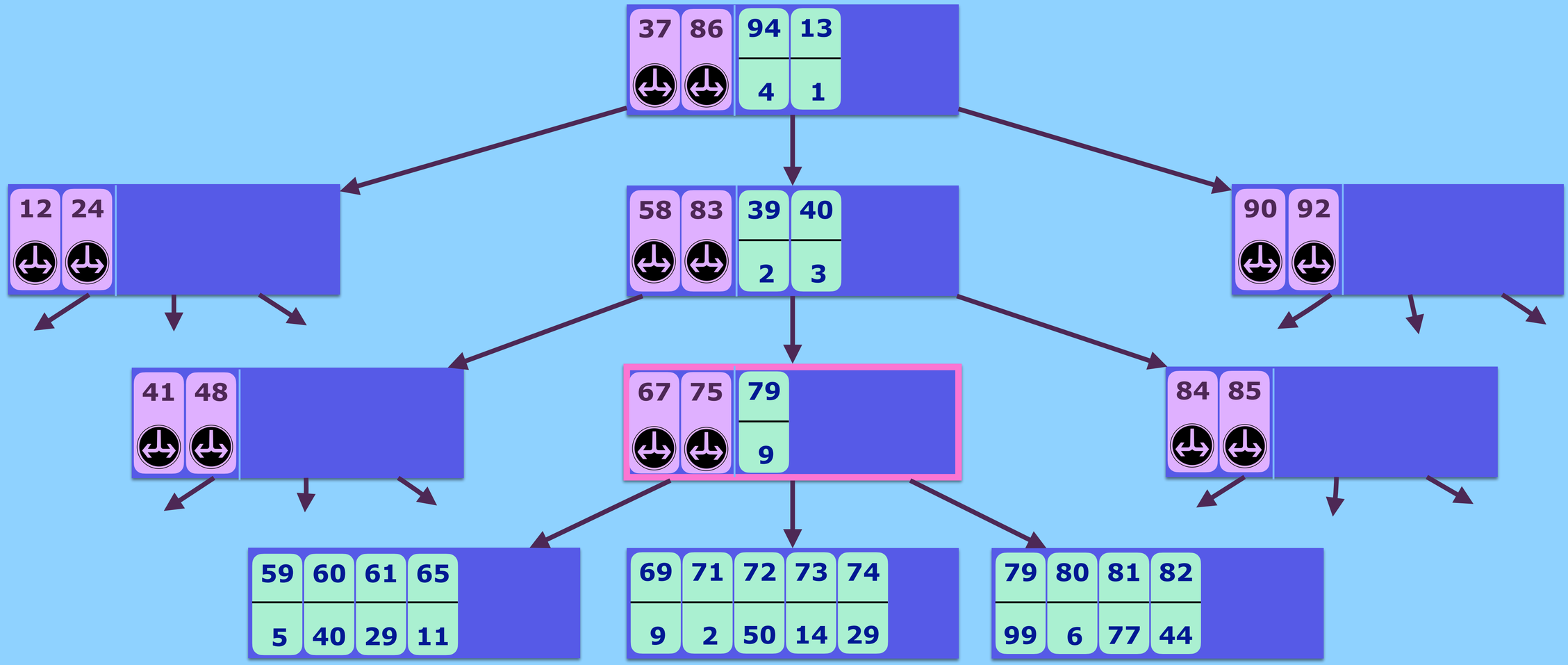
Query(71)



Bε-Trees

Lookups follow pivots, but check buffers along the way

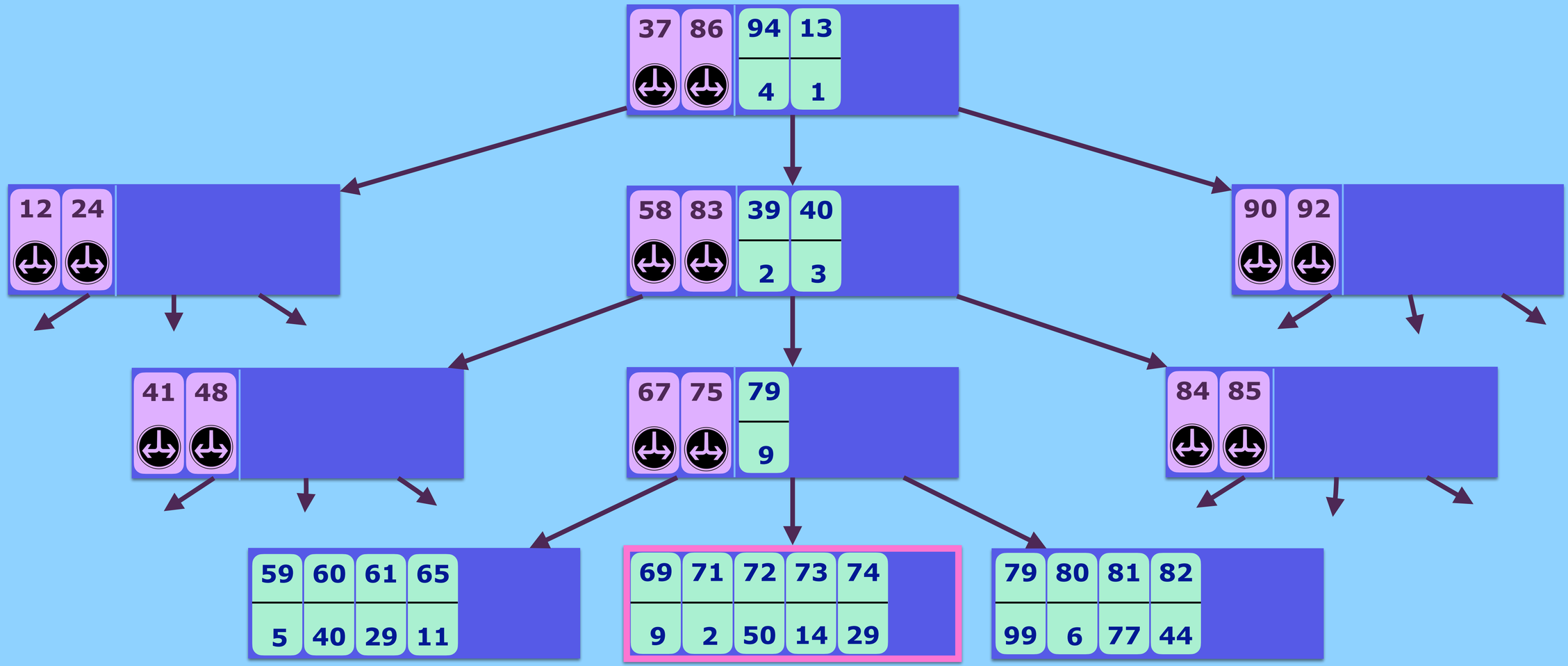
Query(71)



Bε-Trees

Lookups follow pivots, but check buffers along the way

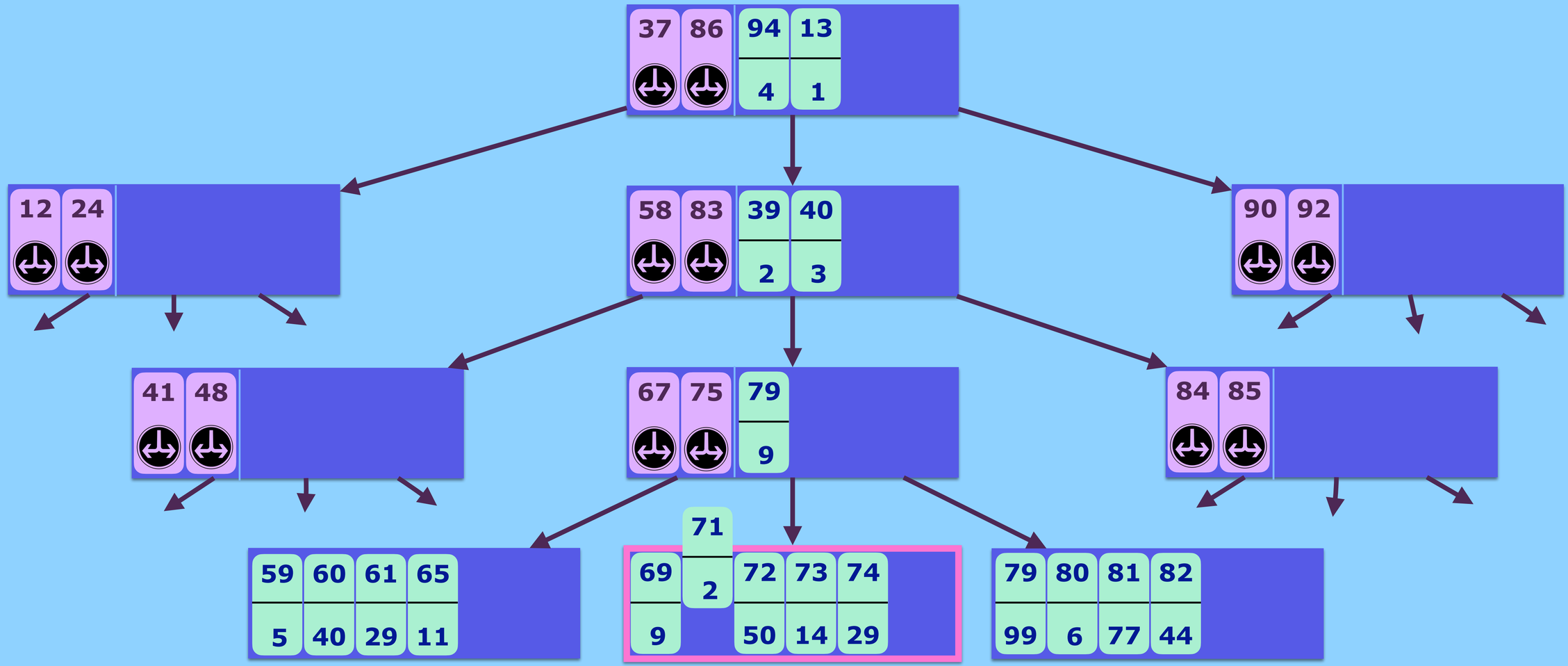
Query(71) → 2



Bε-Trees

Lookups follow pivots, but check buffers along the way

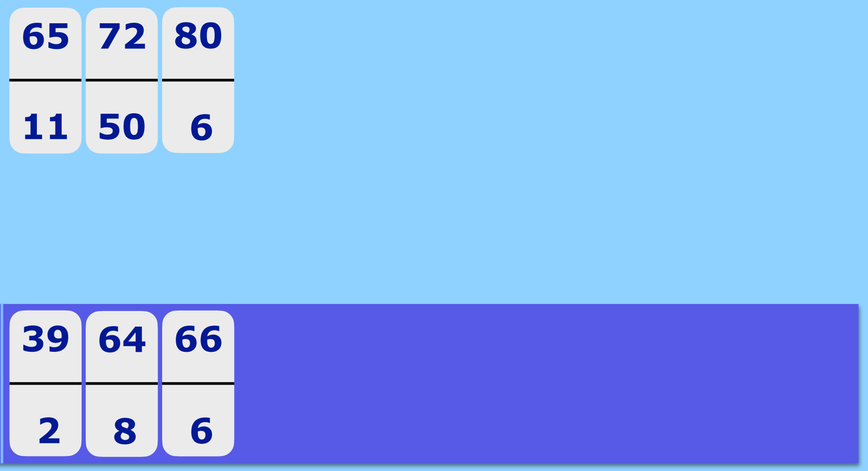
Query(71) → 2



Work
~~Write~~ Amplification
in B^{ε}-Trees

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees



To add new data to a Bε-tree node, the node must be rewritten

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

And again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

And again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

And again

And again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees

To add new data to a Bε-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

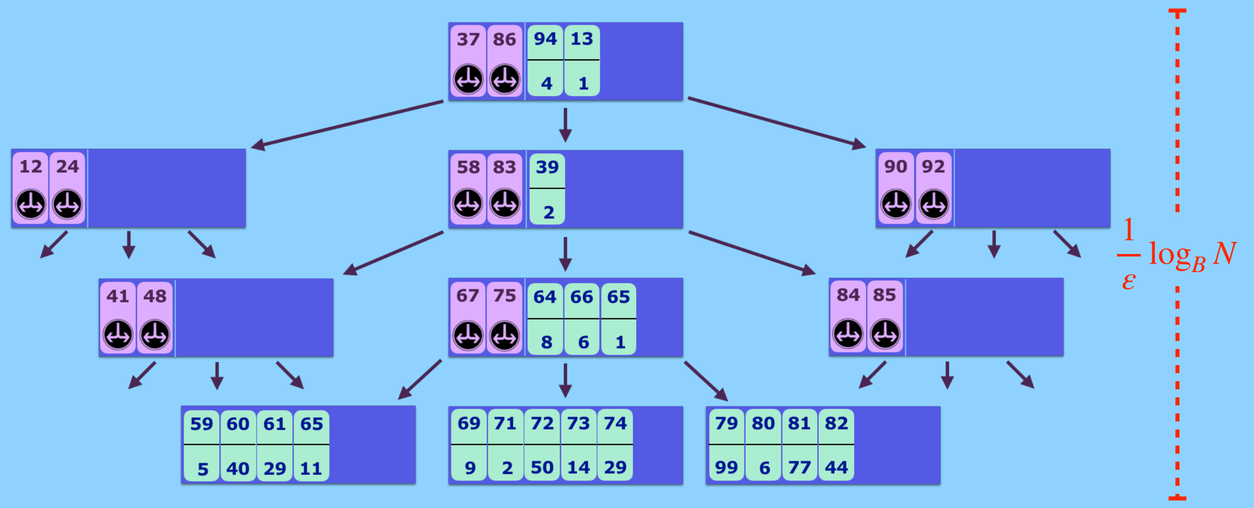
And again

In the worst case, the average message is rewritten Bε/2 times in each node

And again

Work Amplification in Bε-Trees

Recall: Insertions in Bε-trees



In the worst case, the average message is rewritten Bε/2 times in each node

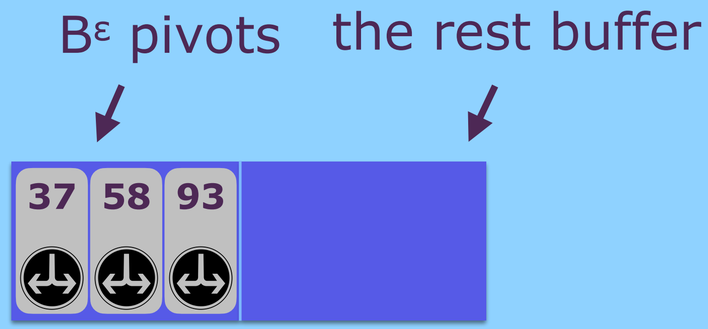
Bε-Tree Work Amplification = $O\left(B^{\varepsilon} \times \log_{B^{\varepsilon}} N\right)$

Size-Tiered Bε-Trees (SplinterDB)

Size-Tiered B^ε-Trees

A Size-Tiered B^ε-tree is a B^ε-tree where the buffer is stored discontiguously

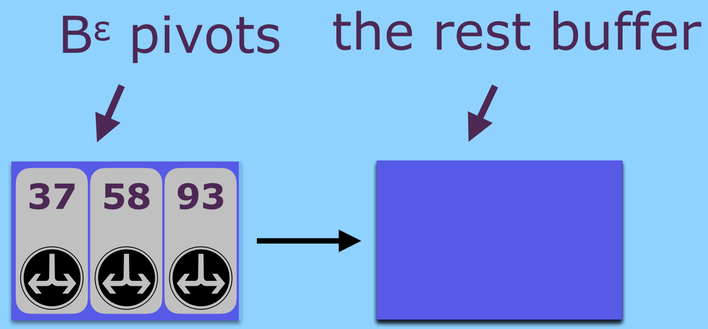
Recall:
a B^ε-tree node has pivots and a buffer



Size-Tiered B^ε-Trees

A Size-Tiered B^ε-tree is a B^ε-tree where the buffer is stored discontiguously

Recall:
a B^ε-tree node has pivots and a buffer

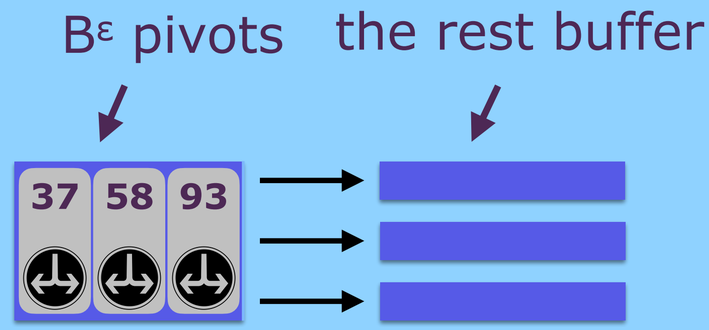


In an STB$^\varepsilon$-tree, the buffer is stored separately

Size-Tiered B^ε-Trees

A Size-Tiered B^ε-tree is a B^ε-tree where the buffer is stored discontiguously

Recall:
a B^ε-tree node has pivots and a buffer

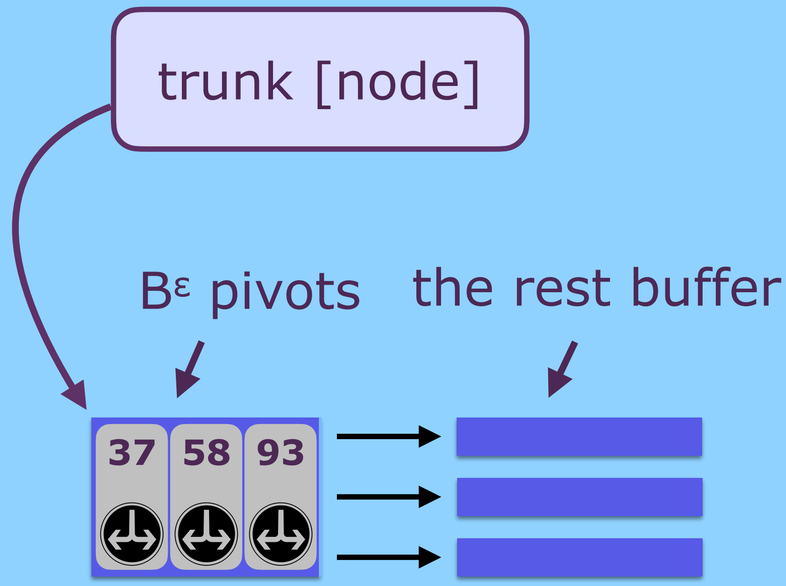


and in several discontiguous pieces

In an STB$^\varepsilon$-tree, the buffer is stored separately

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously



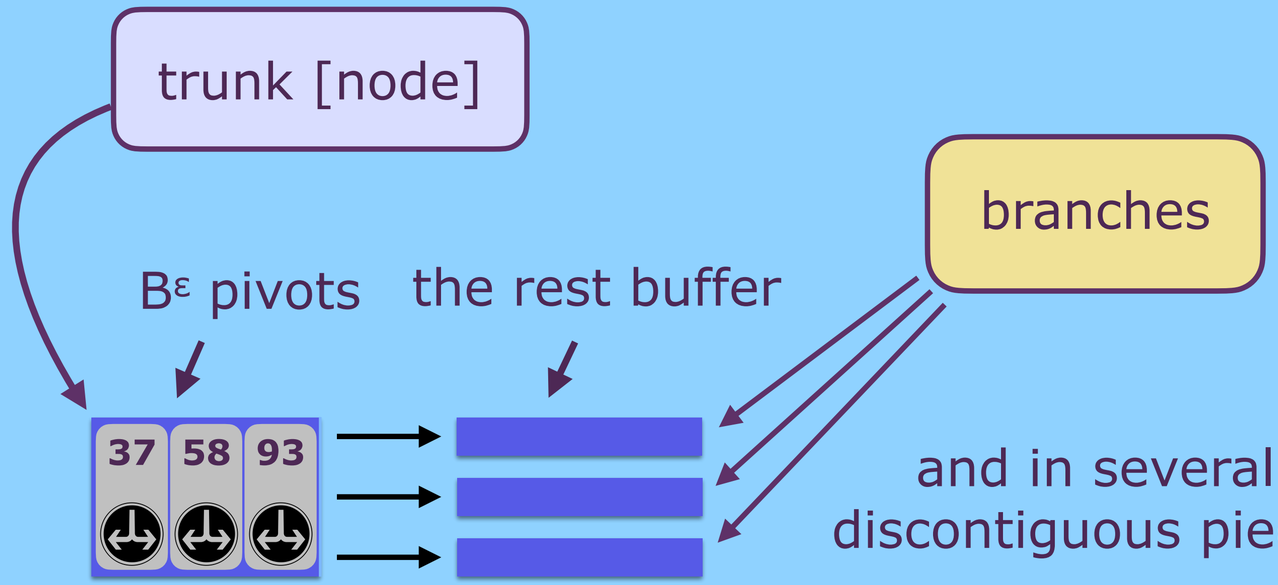
Recall:
a Bε-tree node has pivots and a buffer

and in several discontiguous pieces

In an STBε-tree, the buffer is stored separately

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously



Recall:
a Bε-tree node has pivots and a buffer

In an STBε-tree, the buffer is stored separately

Insertions in Size-Tiered B^ε-Trees

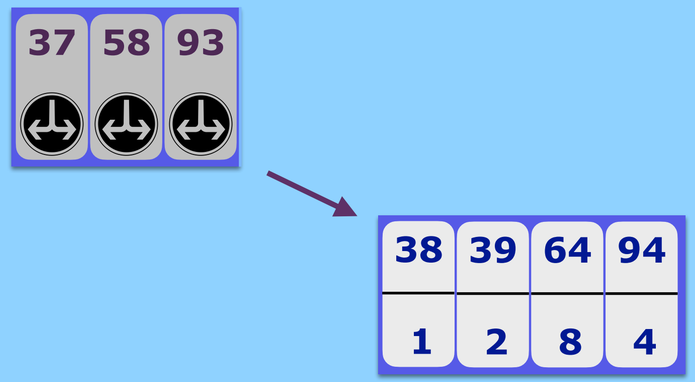
Size-Tiered B^ε-Trees

A Size-Tiered B^ε-tree is a B^ε-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node…

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

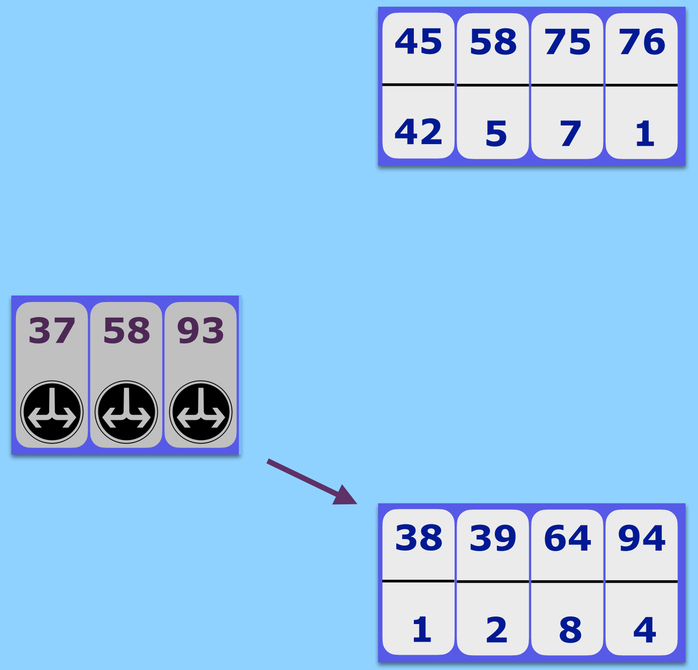


When new data is flushed into the trunk node…

…it is added as a new branch

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

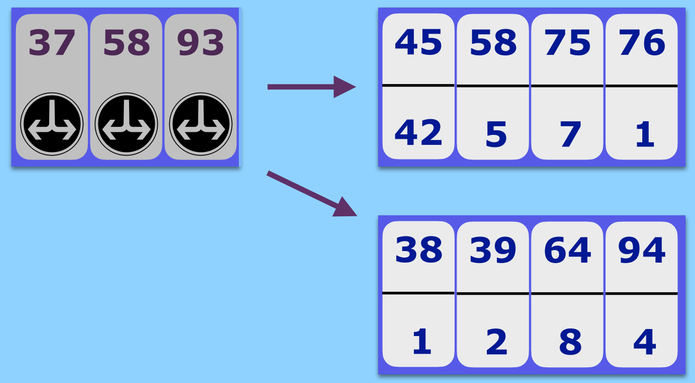


When new data is flushed into the trunk node…

…it is added as a new branch

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously



When new data is flushed into the trunk node…

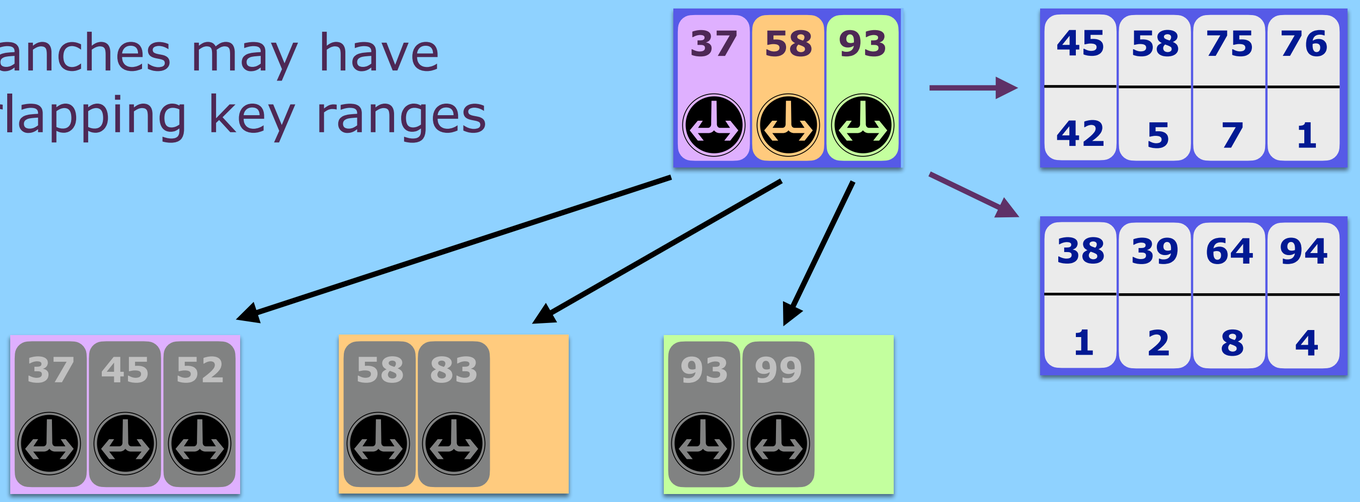
…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

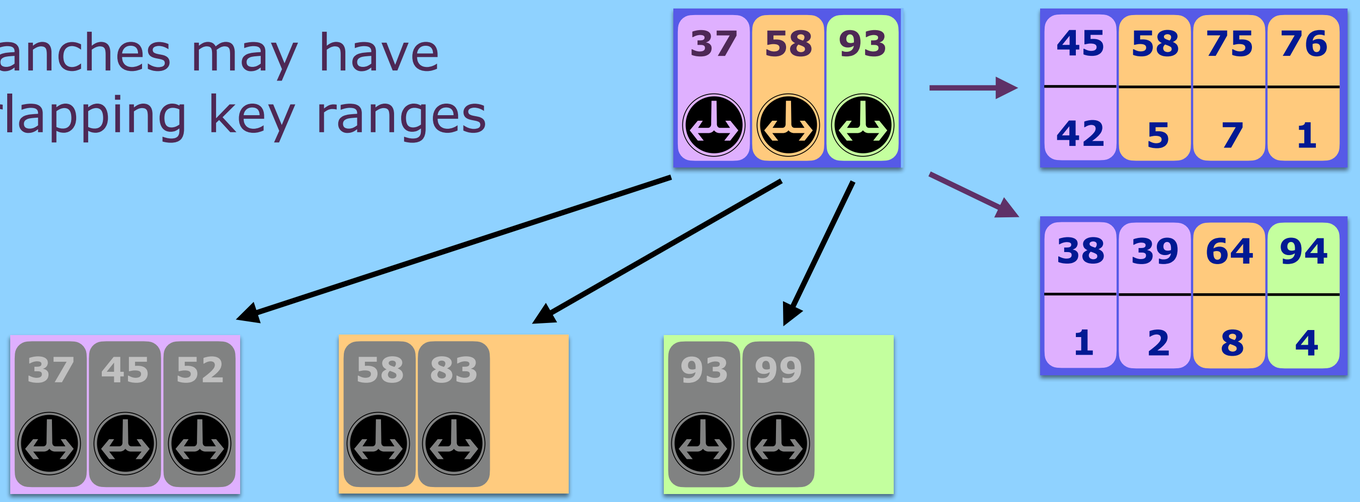
…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Branches may have overlapping key ranges



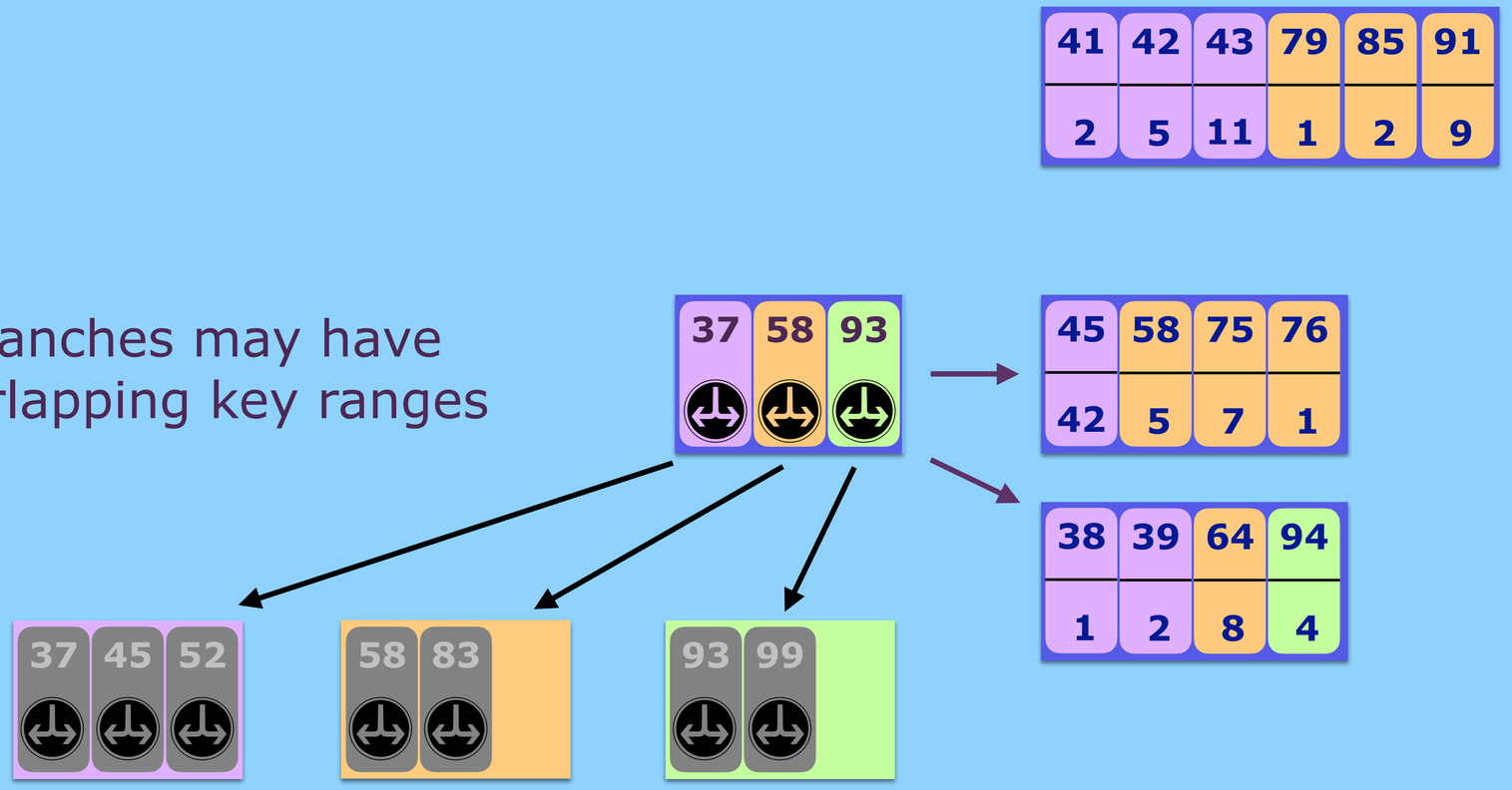
When new data is flushed into the trunk node…

…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously



When new data is flushed into the trunk node…

…it is added as a new branch

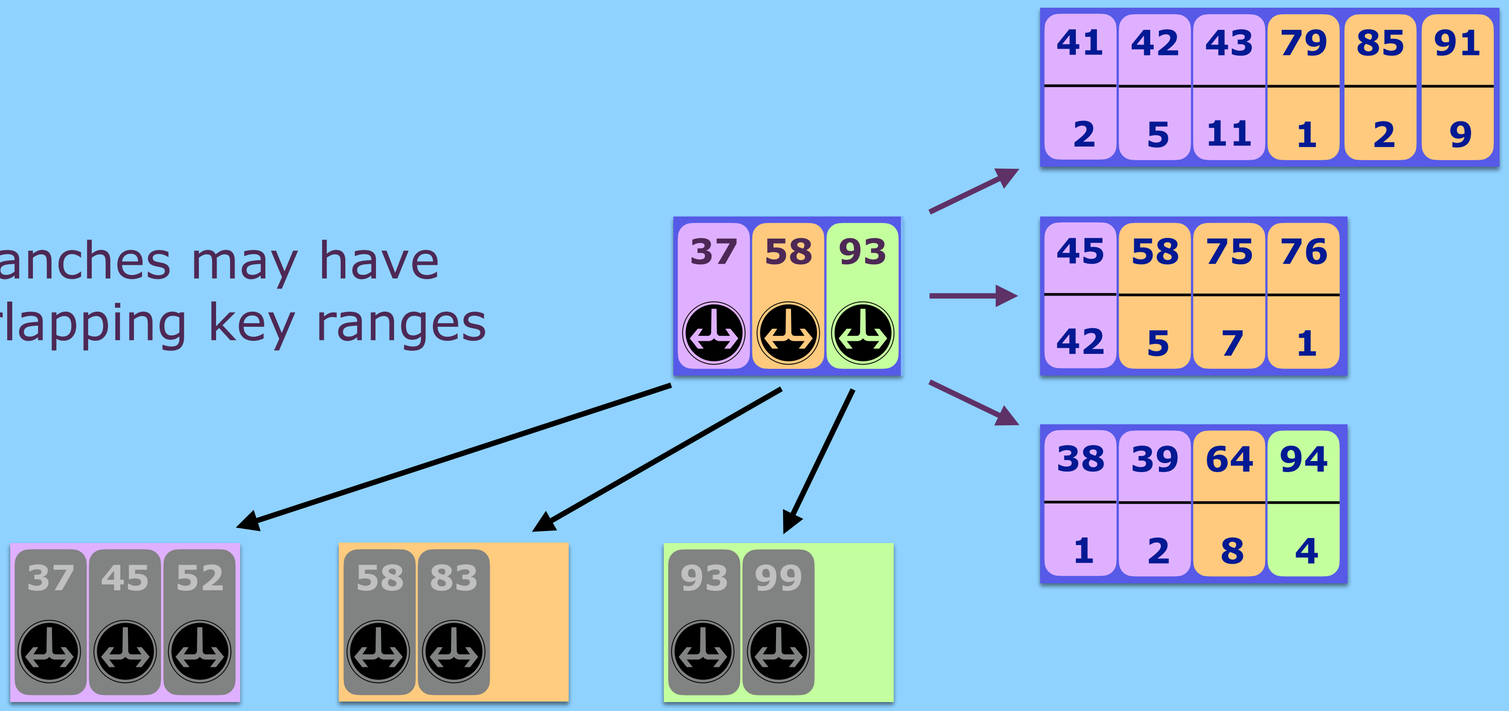
Branches may have overlapping key ranges

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

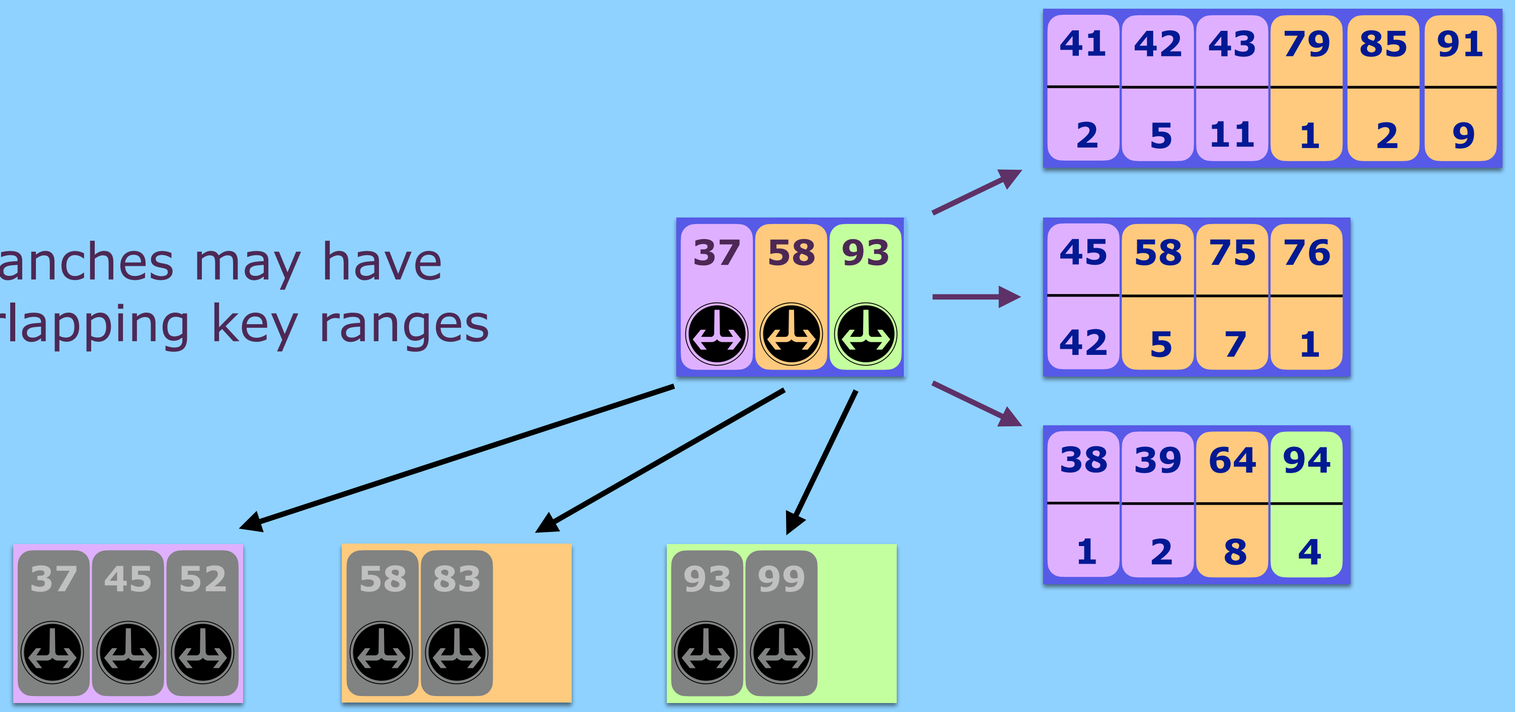
…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

The fullness threshold is:
Fanout × Average Buffer Size



Branches may have overlapping key ranges

When new data is flushed into the trunk node…

…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

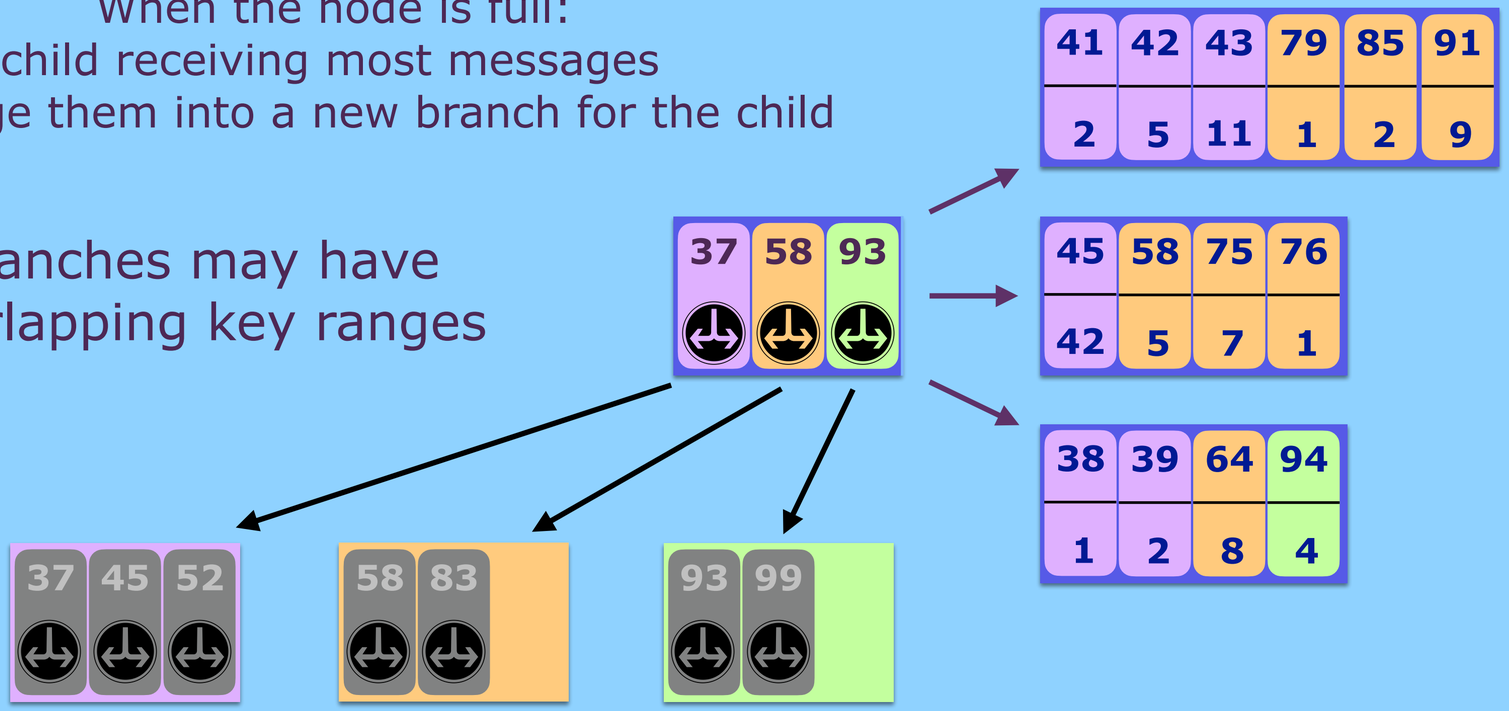
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

The fullness threshold is:
Fanout × Average Buffer Size

When the node is full:

1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

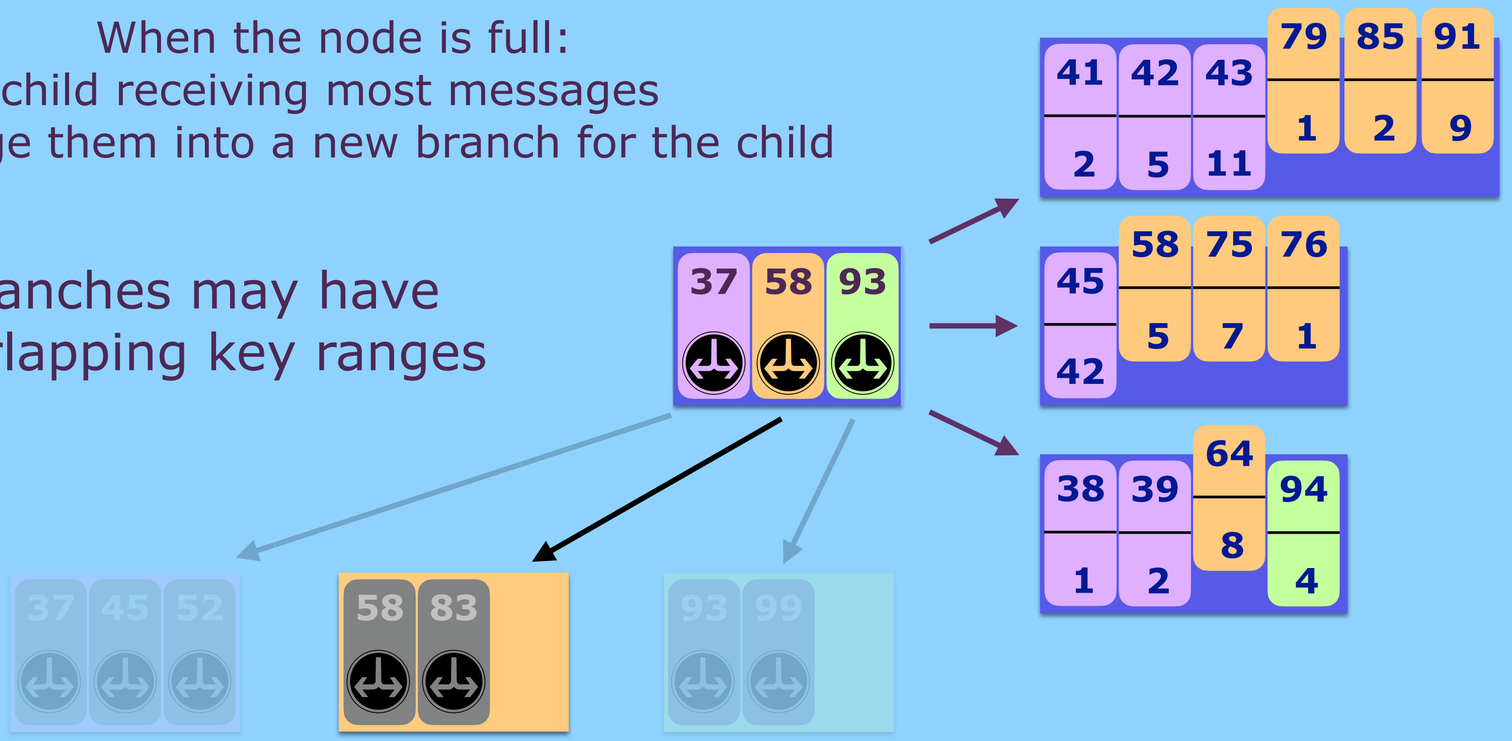
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

The fullness threshold is:
Fanout × Average Buffer Size

When the node is full:

1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

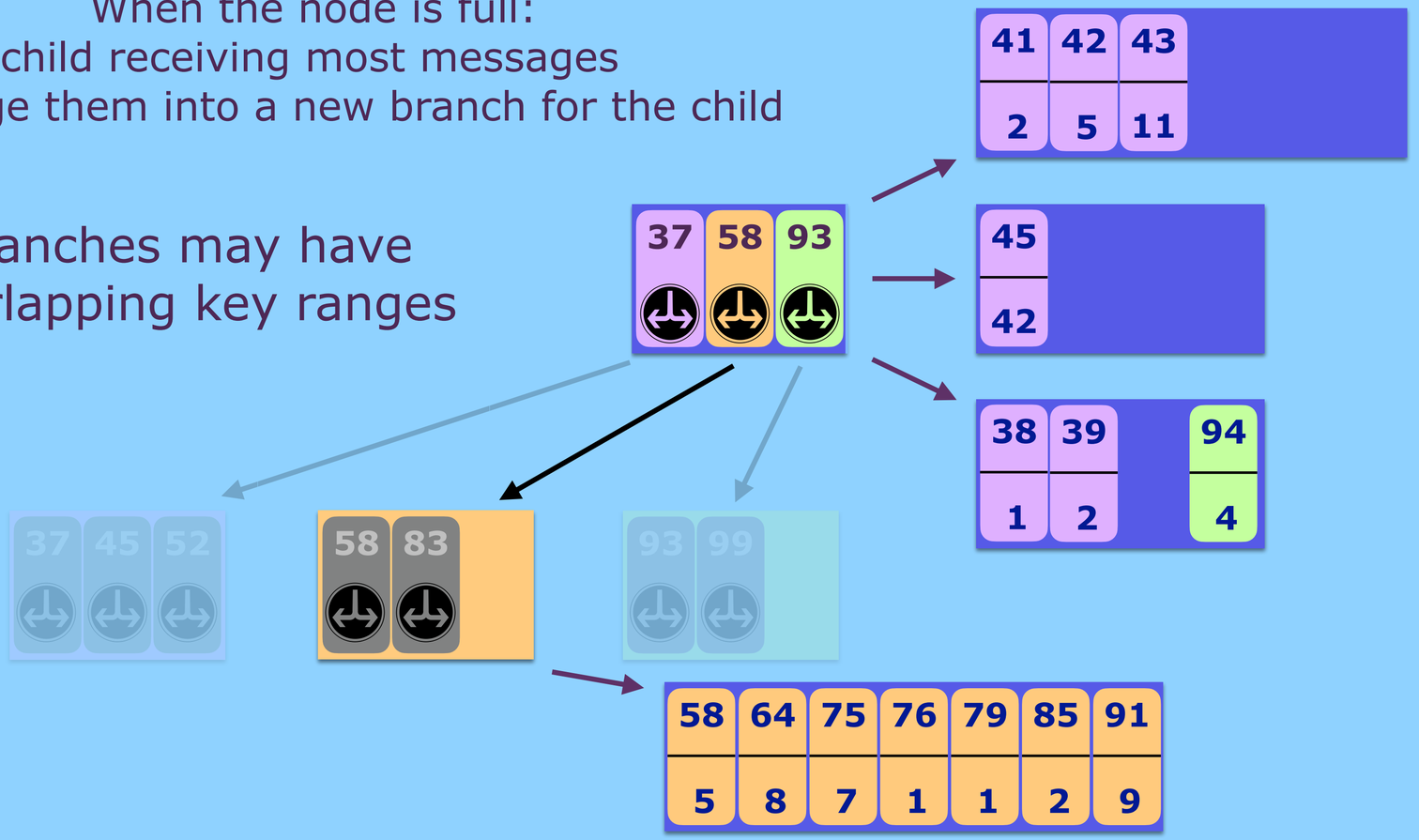
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

The fullness threshold is:
Fanout × Average Buffer Size

When the node is full:

1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

…it is added as a new branch

The old branches do not need to be rewritten

Size-Tiered Bε-Trees

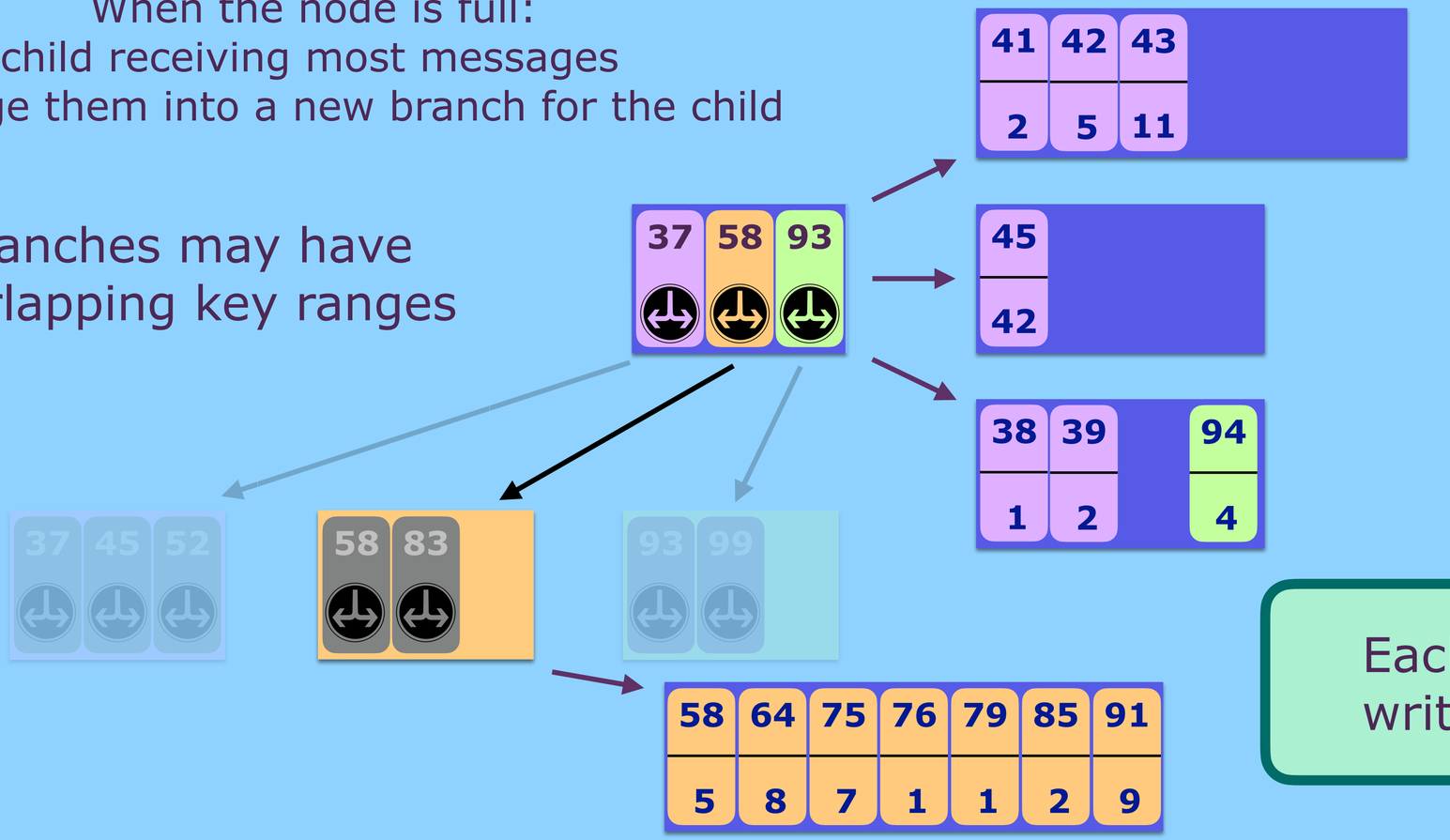
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

The fullness threshold is:
Fanout × Average Buffer Size

When the node is full:

1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

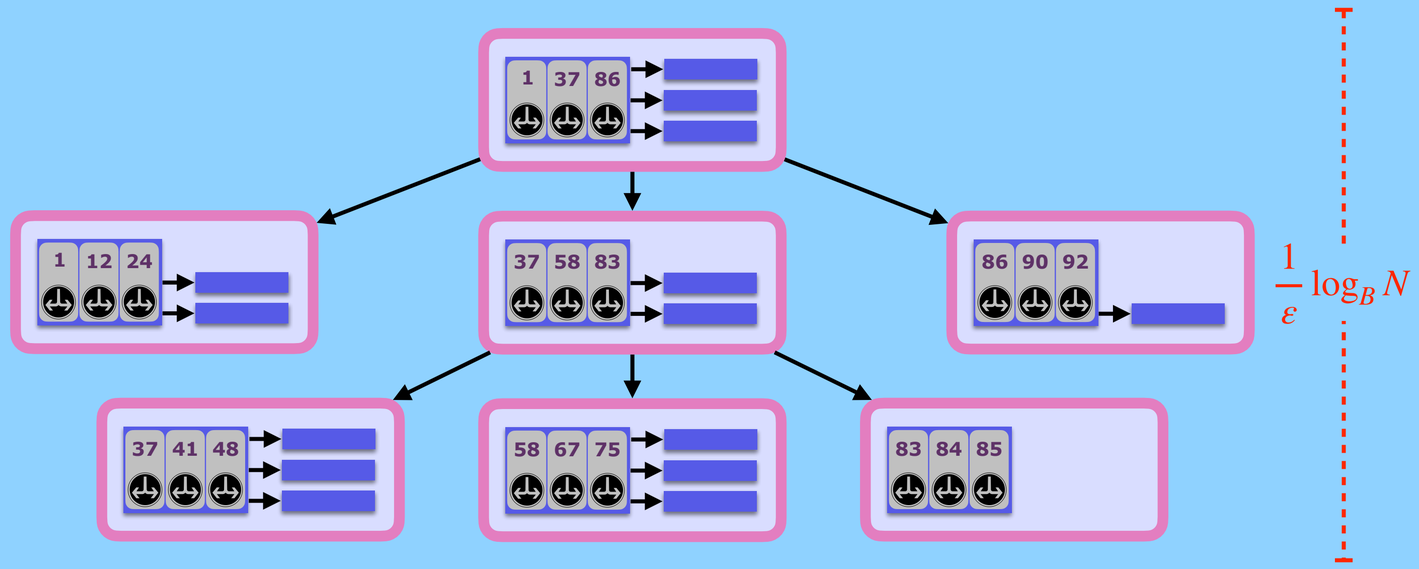
…it is added as a new branch

The old branches do not need to be rewritten

Each key-value pair is read/ written once per trunk node

Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously



Each key-value pair is read/ written once per trunk node

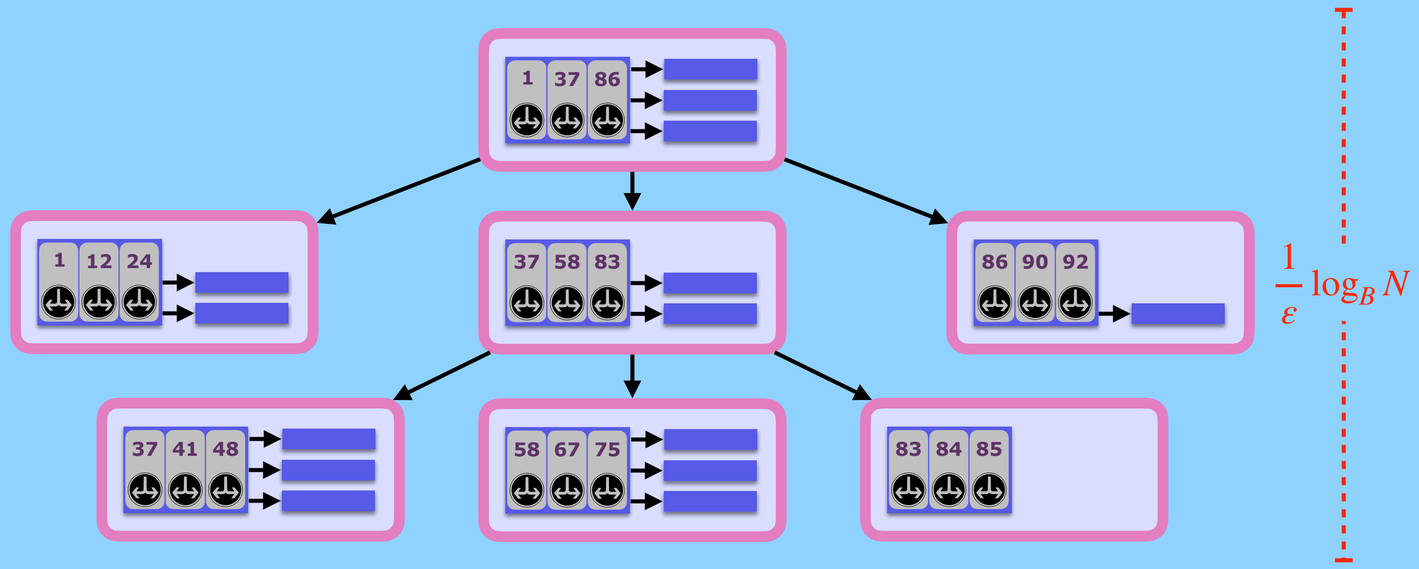
Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Work Amplification

B$^{\varepsilon}$-Tree: $O\left(B^{\varepsilon} \times \log_{B^{\varepsilon}} N\right)$

Size-Tiered B$^{\varepsilon}$-Tree: $O\left(\log_{B^{\varepsilon}} N\right)$



Each key-value pair is read/ written once per trunk node

Size-Tiered Bε-Trees

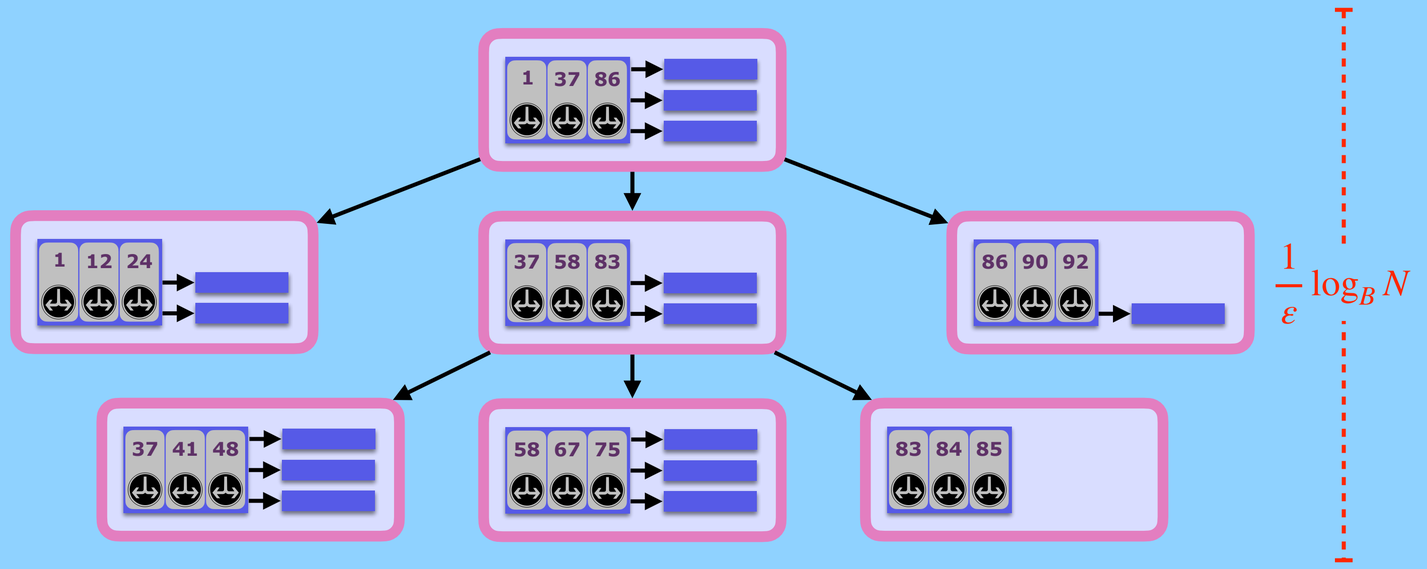
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Work Amplification

Bε-Tree: $O\left(B^\varepsilon \times \log_{B^\varepsilon} N\right)$

$B^\varepsilon \times$ less

Size-Tiered Bε-Tree: $O\left(\log_{B^\varepsilon} N\right)$



Each key-value pair is read/ written once per trunk node

Size-Tiered Bε-Trees

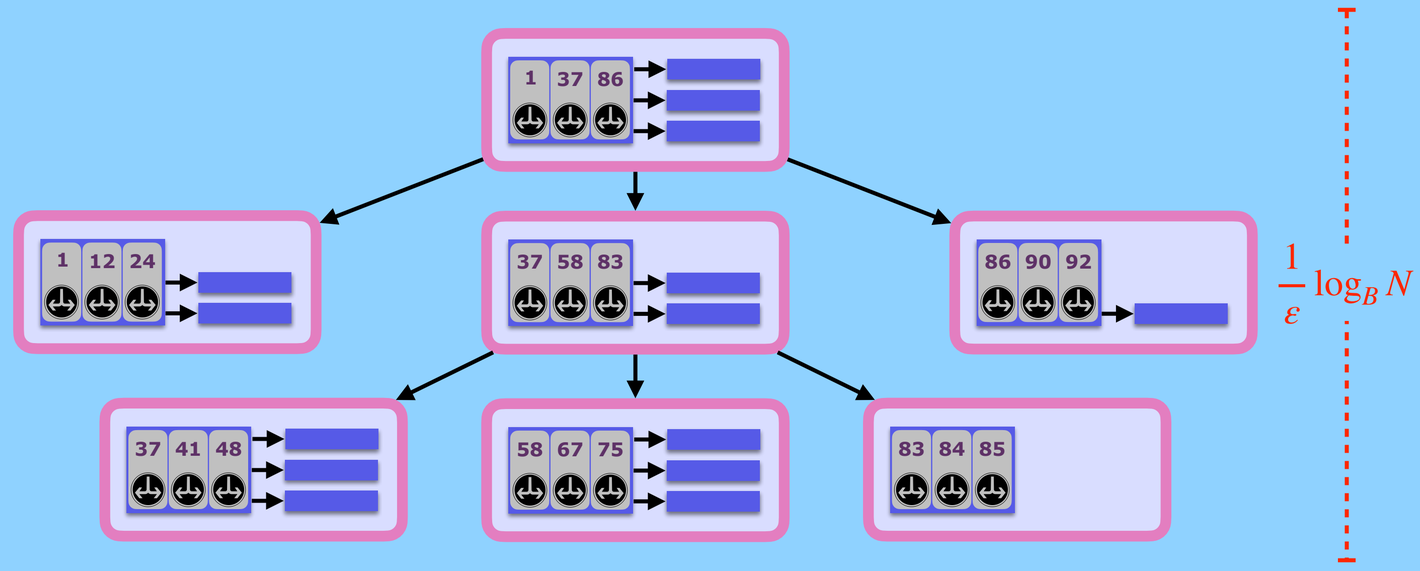
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Work Amplification

Bε-Tree: $O\left(B^\varepsilon \times \log_{B^\varepsilon} N\right)$

Size-Tiered Bε-Tree: $O\left(\log_{B^\varepsilon} N\right)$

$B^\varepsilon \times$ less



Each key-value pair is read/ written once per trunk node

Size-Tiered Bε-Trees

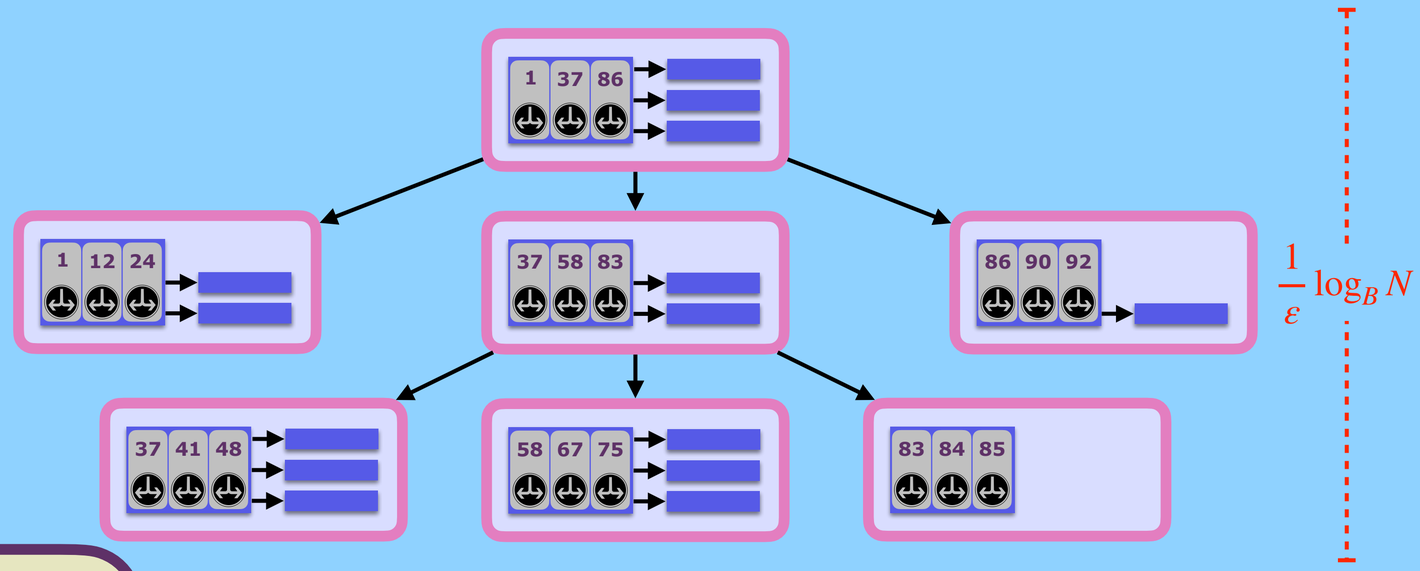
A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously

Work Amplification

Bε-Tree: $O\left(B^\varepsilon \times \log_{B^\varepsilon} N\right)$

Size-Tiered Bε-Tree: $O\left(\log_{B^\varepsilon} N\right)$

$B^\varepsilon \times$ less



Insertion Cost

Bε-Tree: $= O\left(\frac{1}{B} B^\varepsilon \times \log_{B^\varepsilon} \frac{N}{M}\right)$

Size-Tiered Bε-Tree: $= O\left(\frac{1}{B} \log_{B^\varepsilon} \frac{N}{M}\right)$

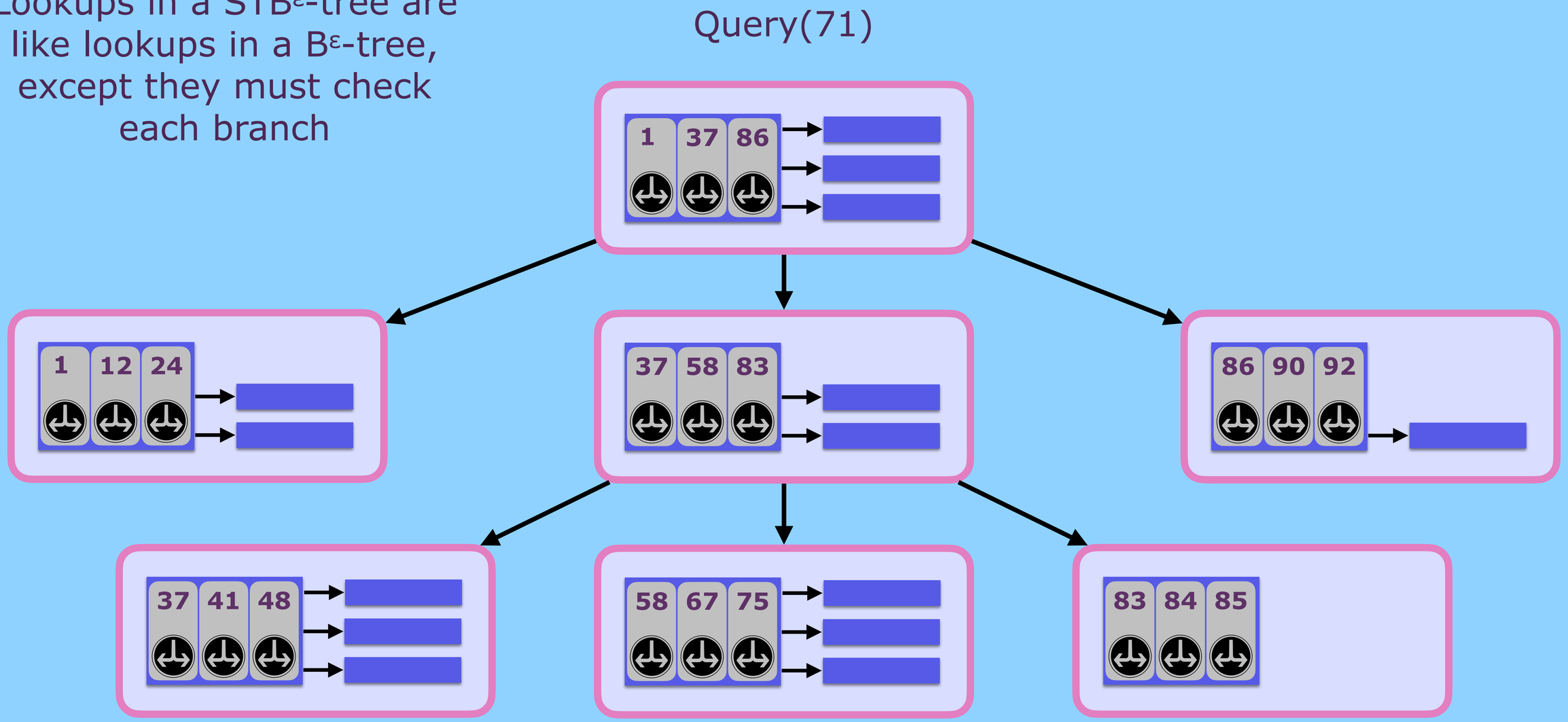
$B^\varepsilon \times$ less

Each key-value pair is read/ written once per trunk node

Lookups in Size-Tiered B^{ε}-Trees

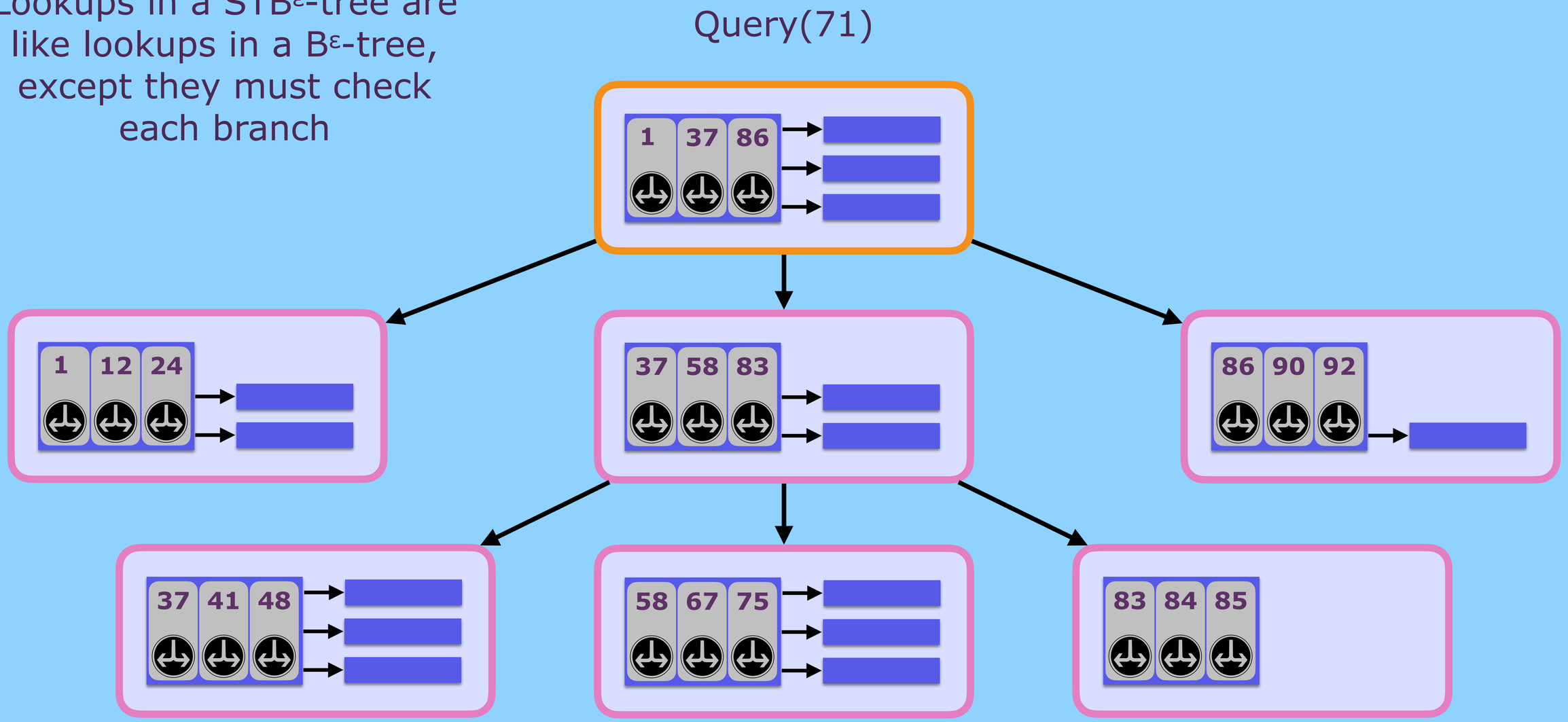
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



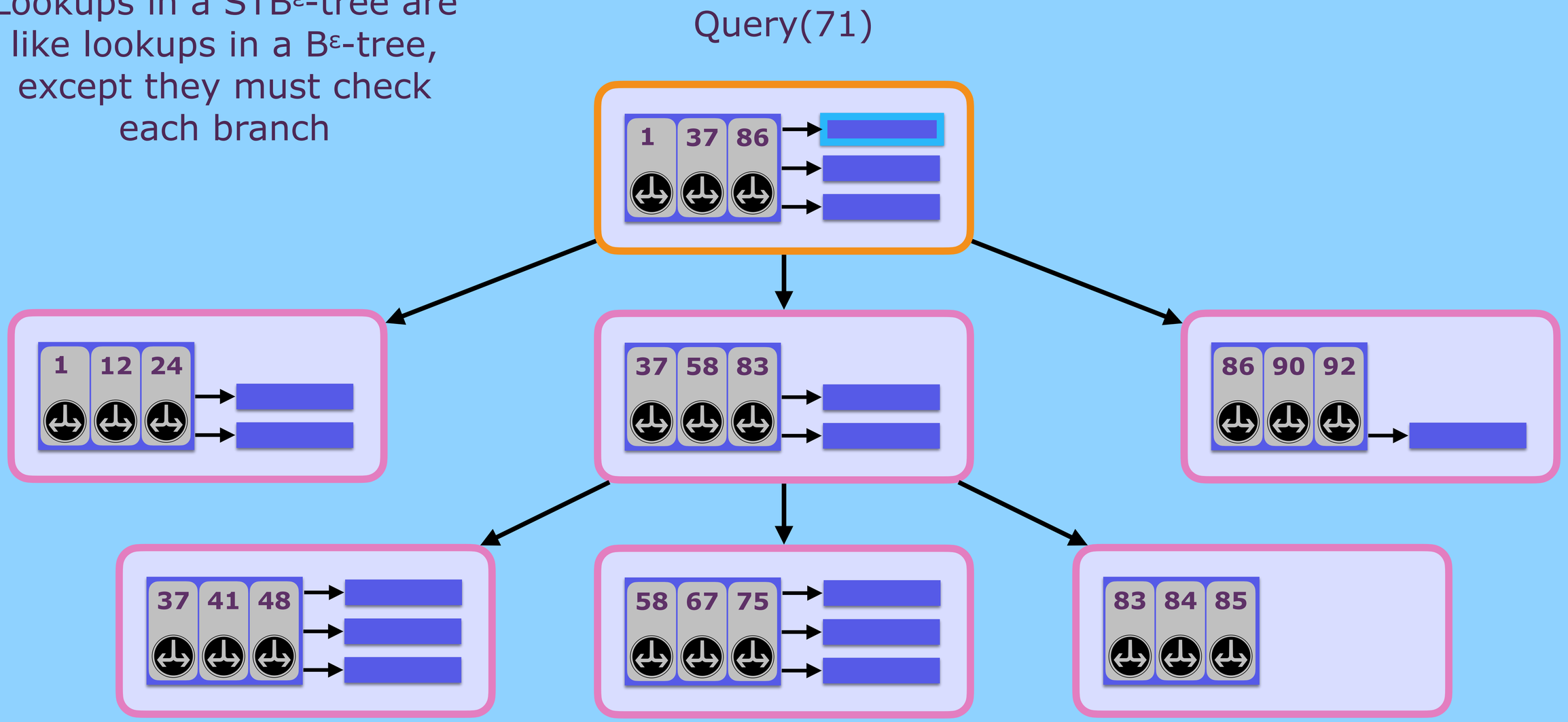
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



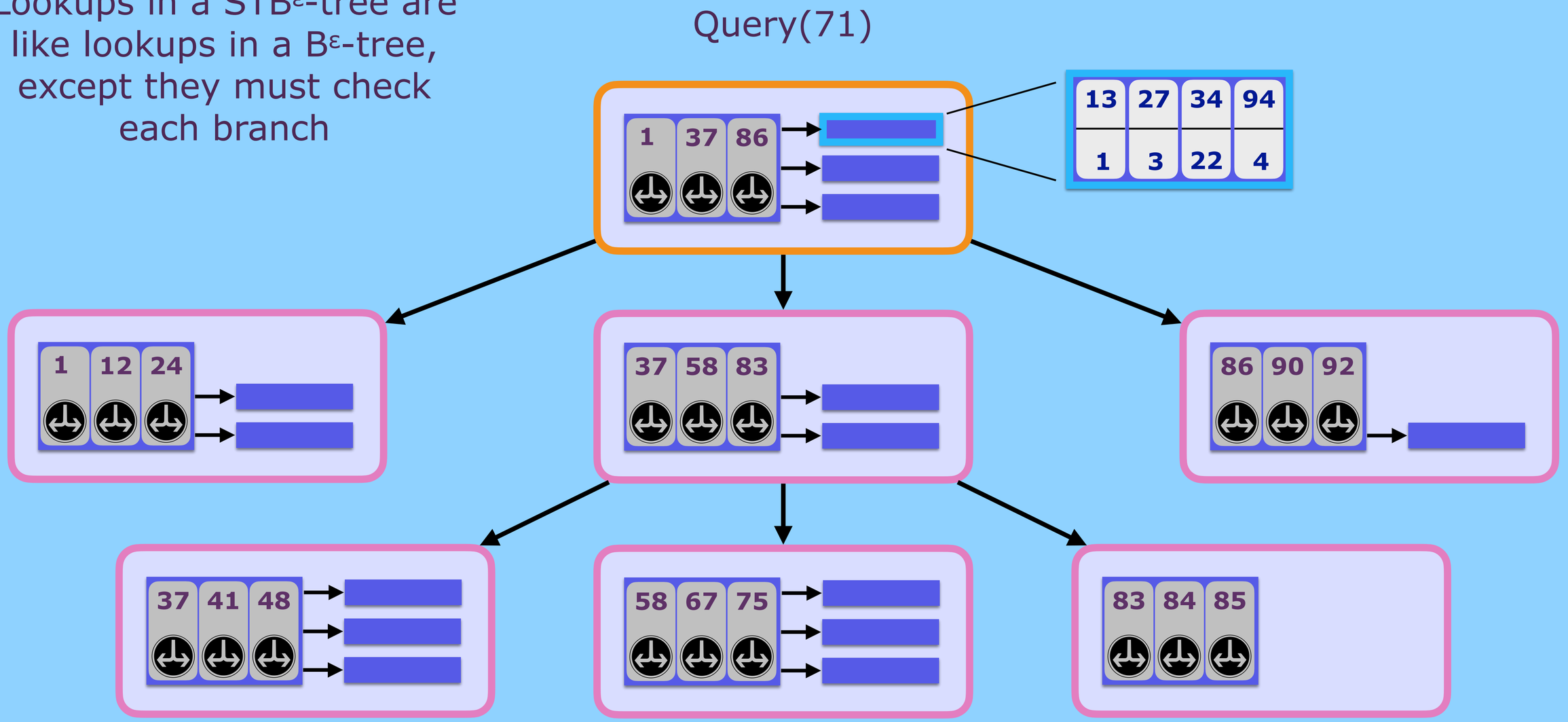
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



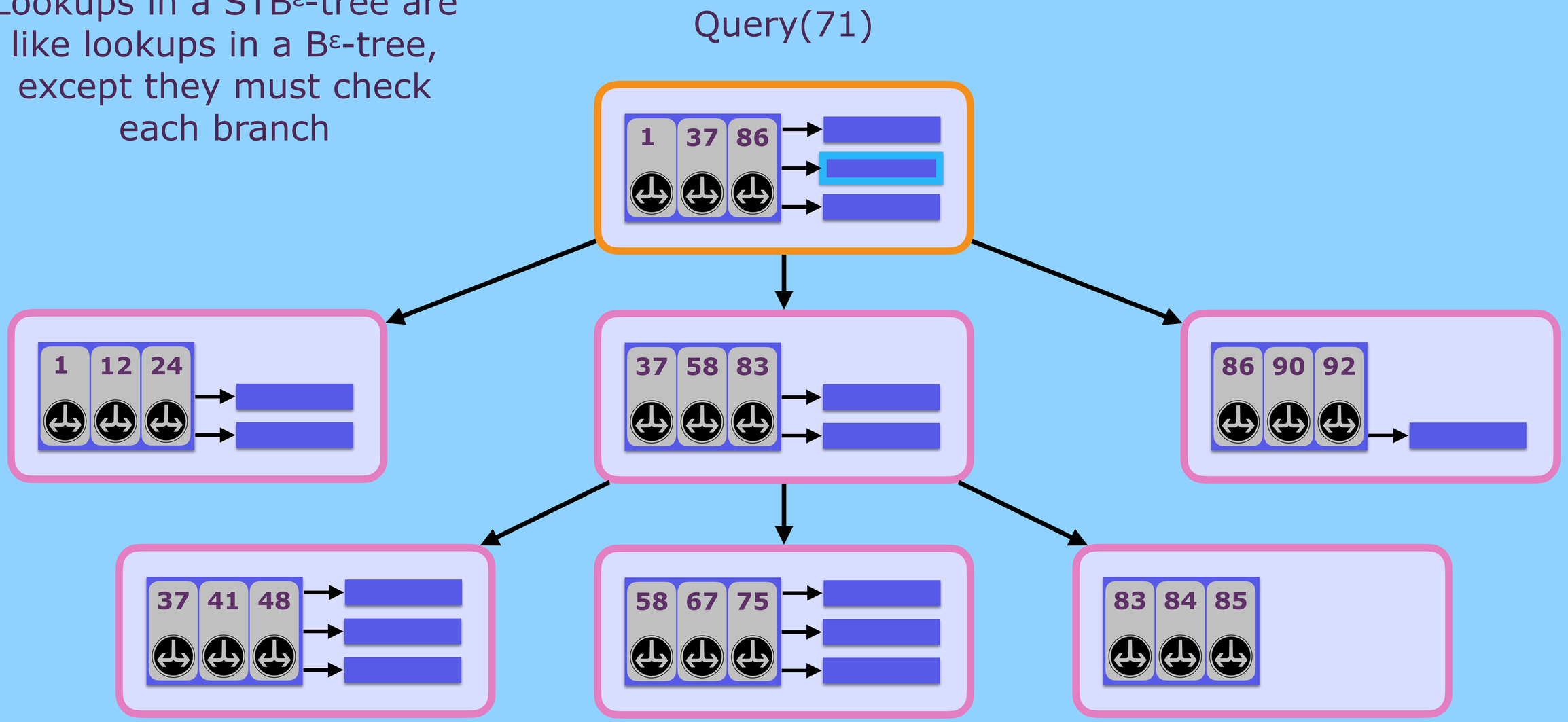
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



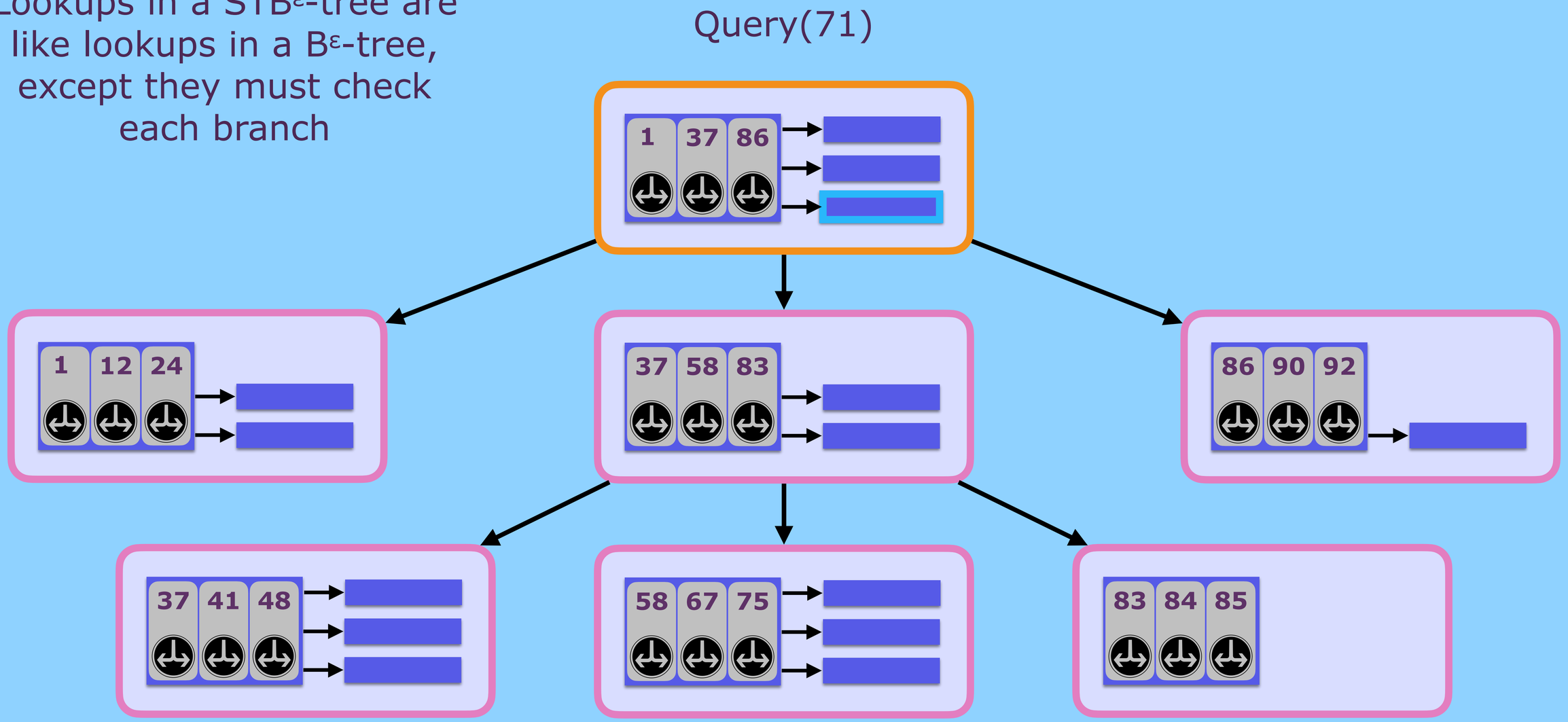
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



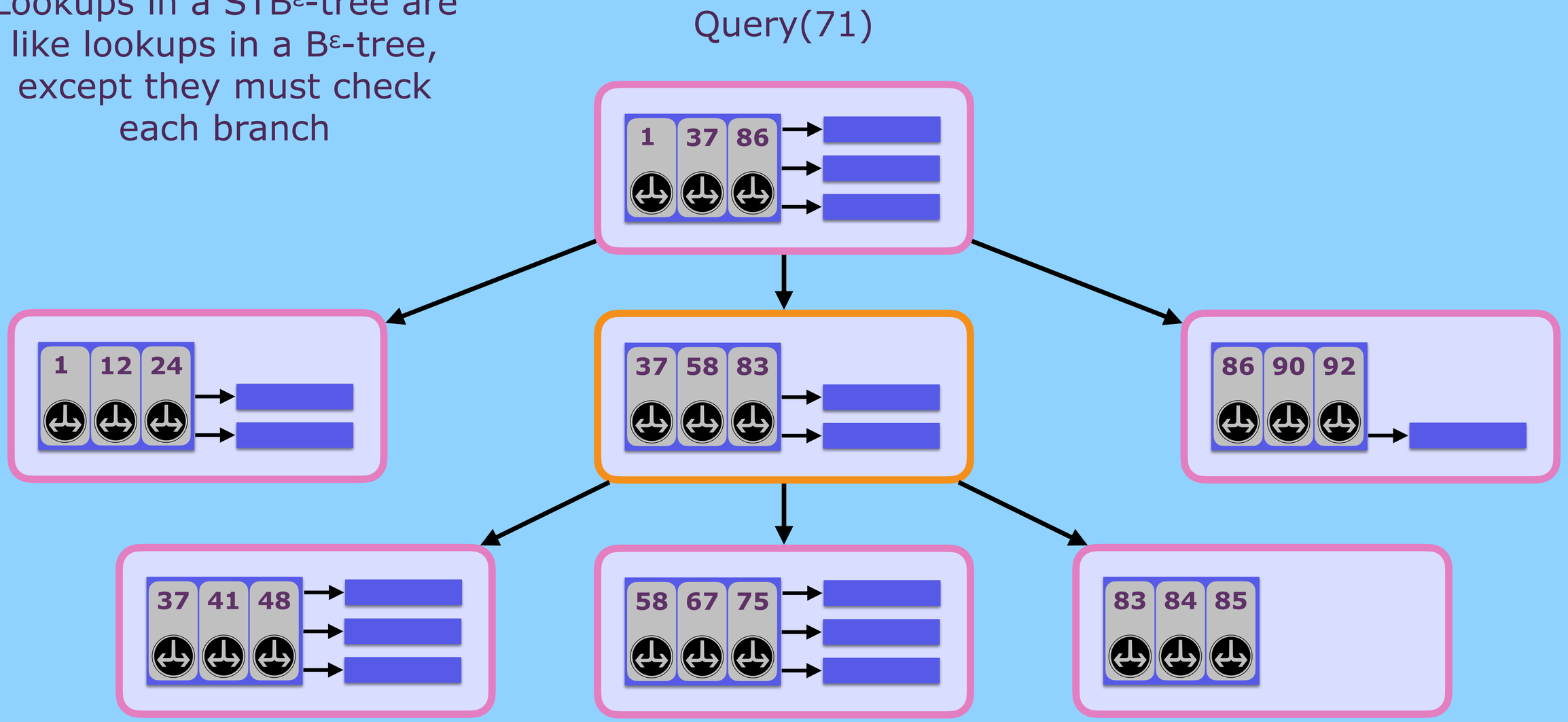
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



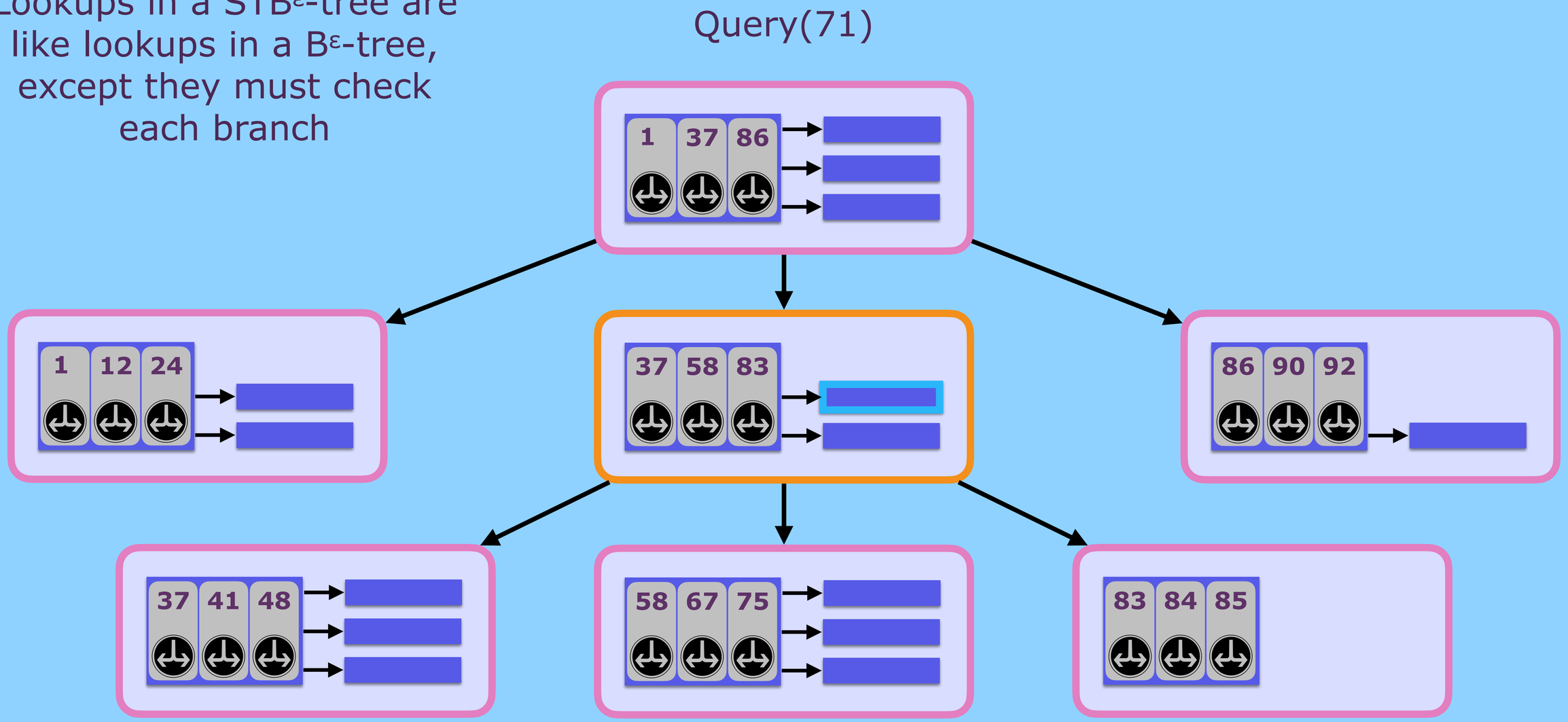
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



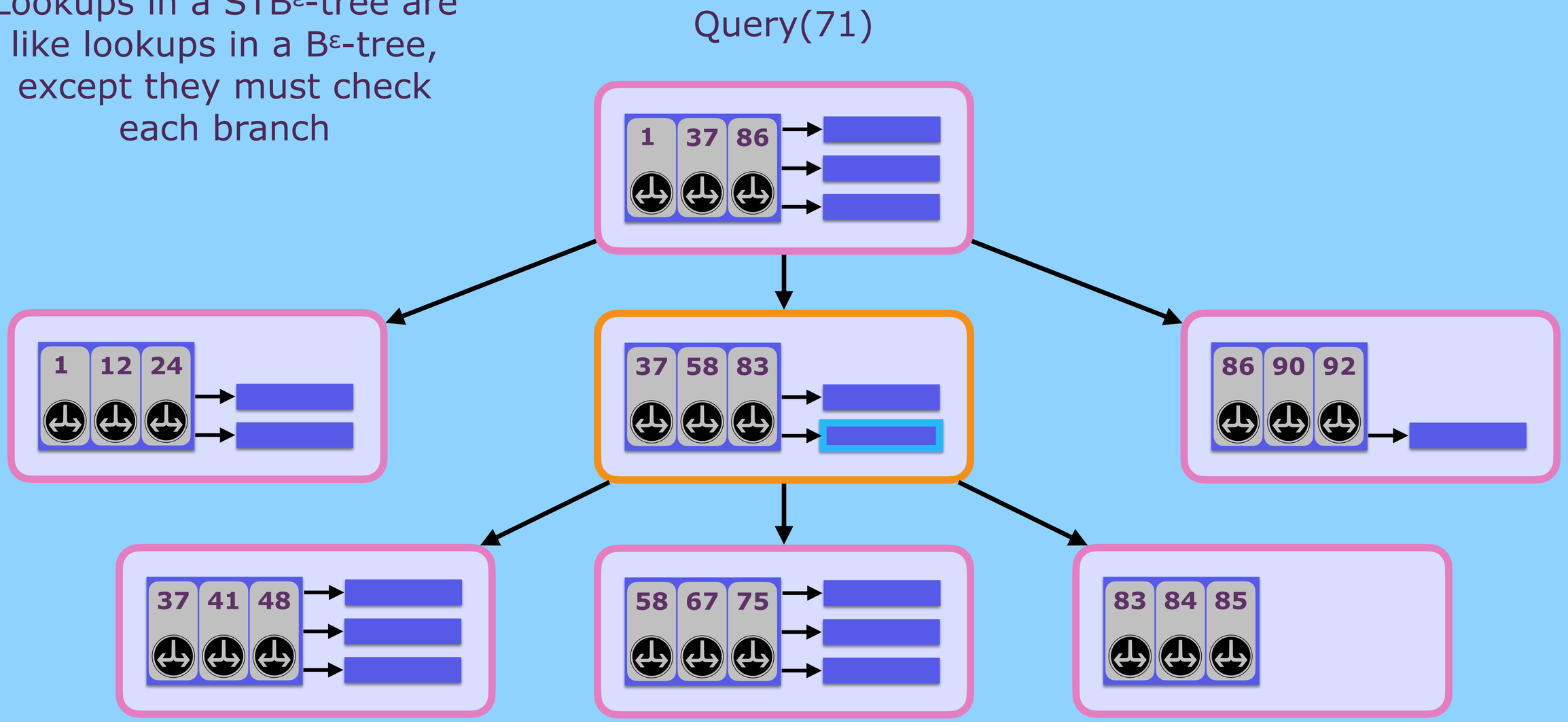
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



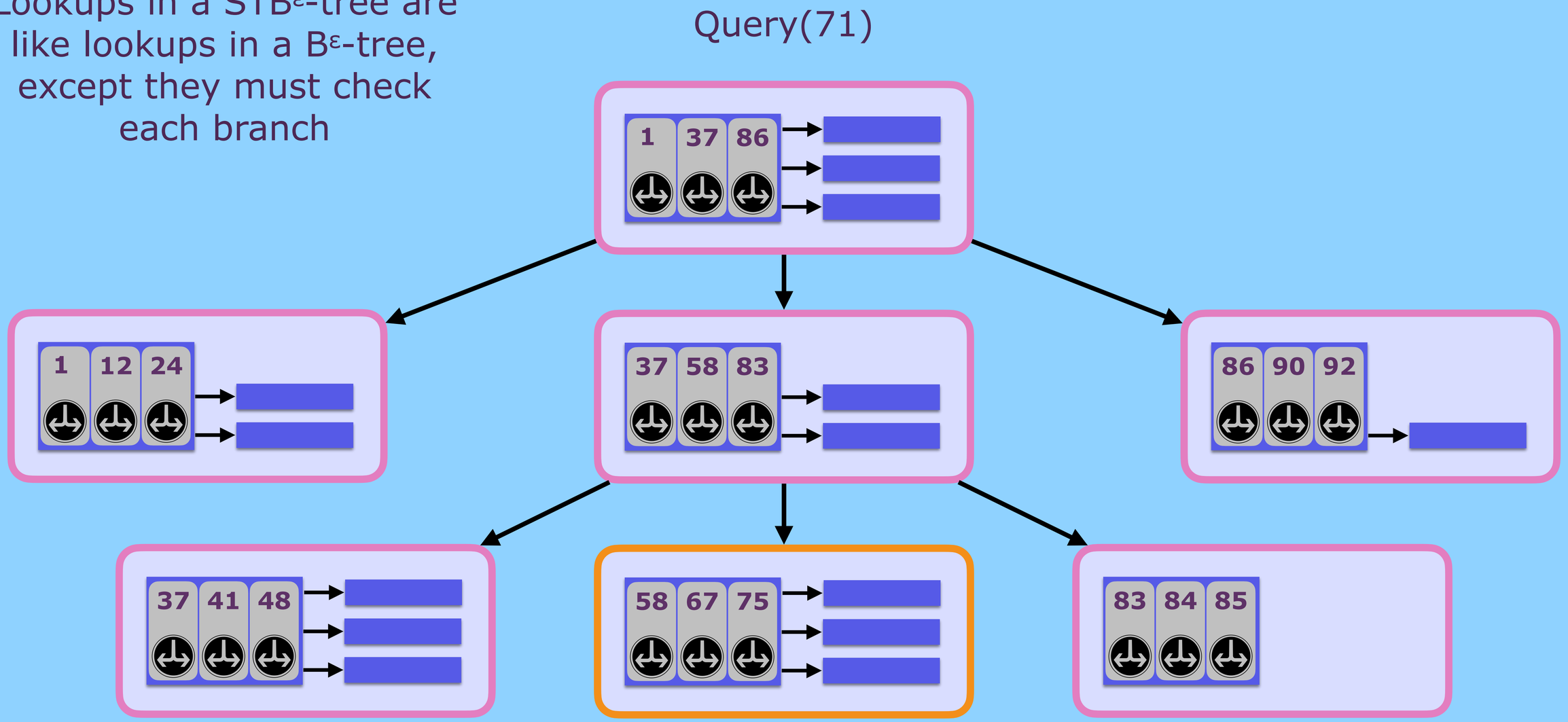
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



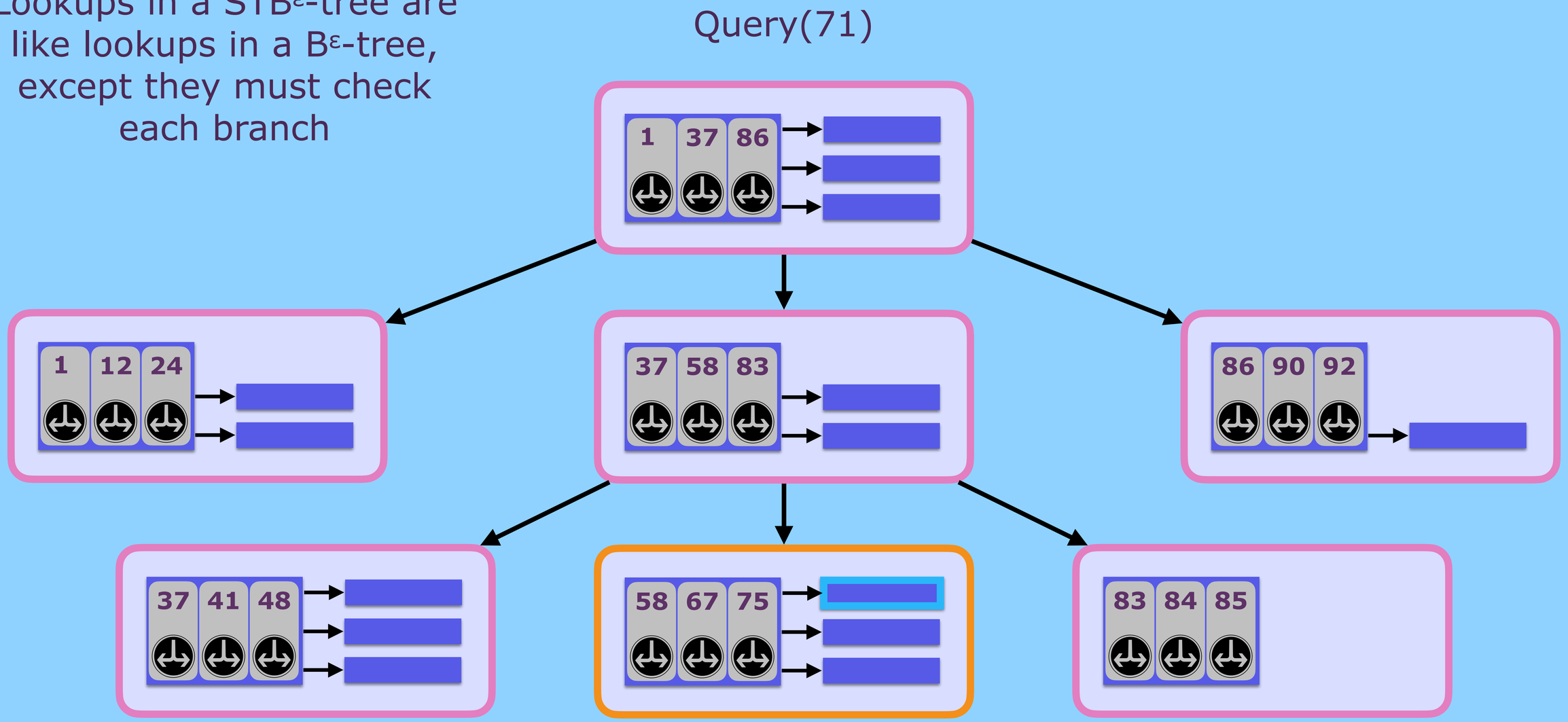
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



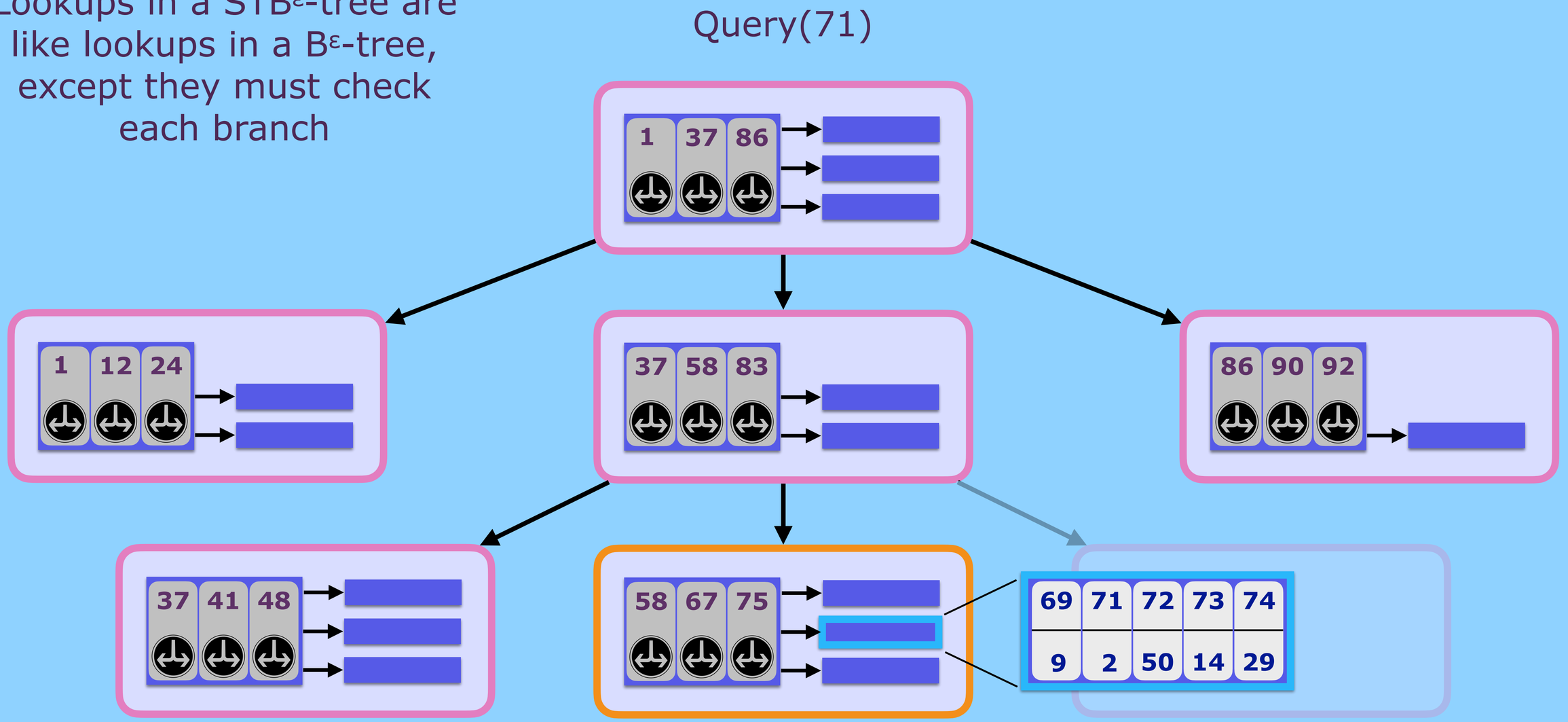
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



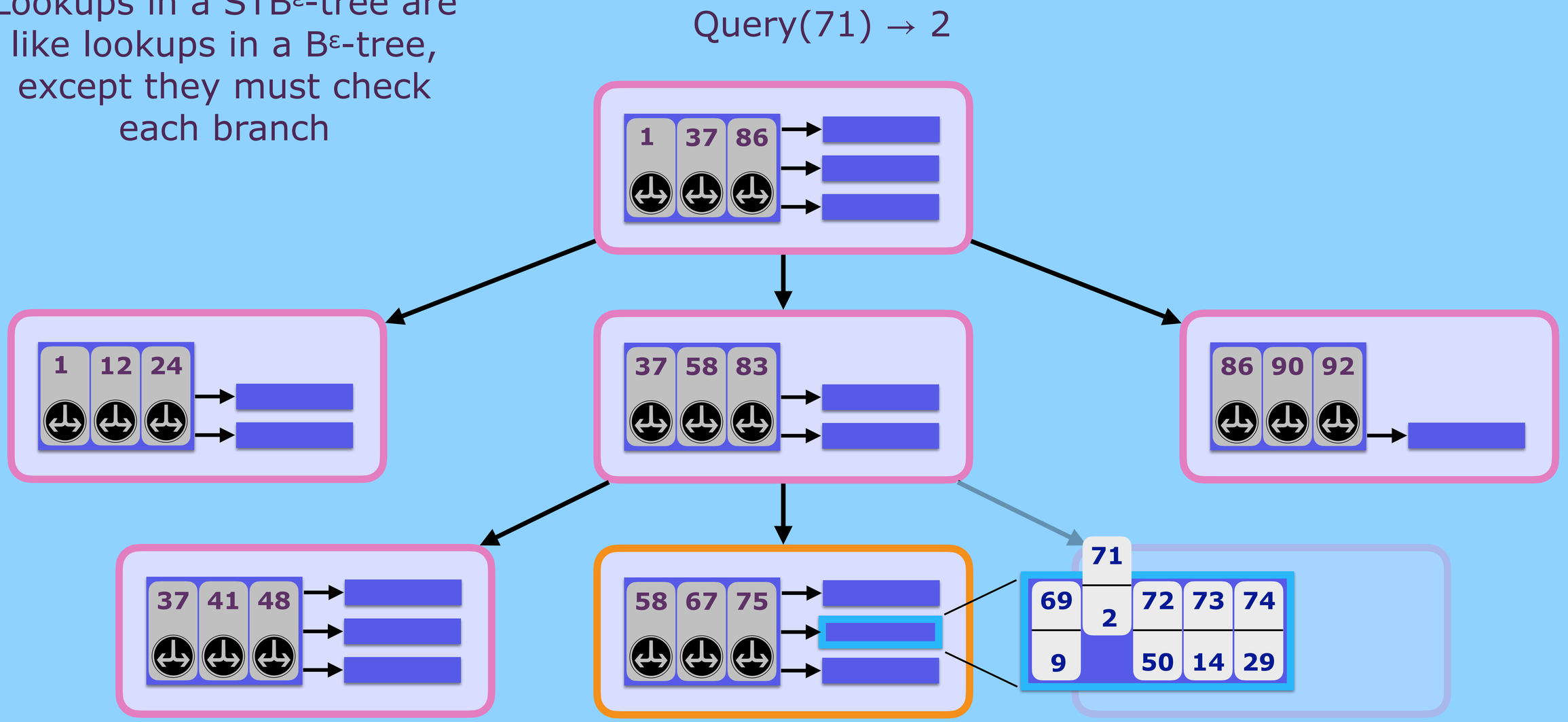
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



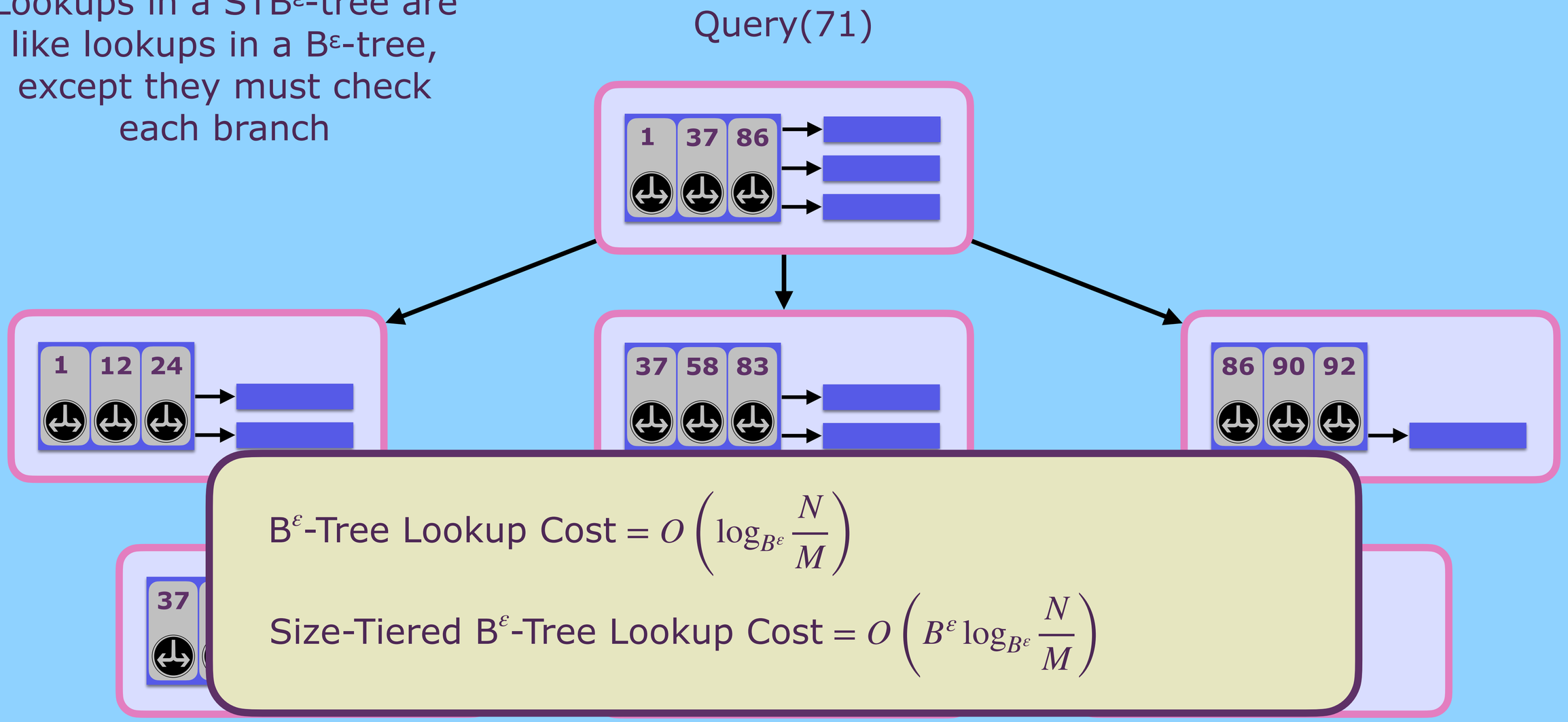
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch

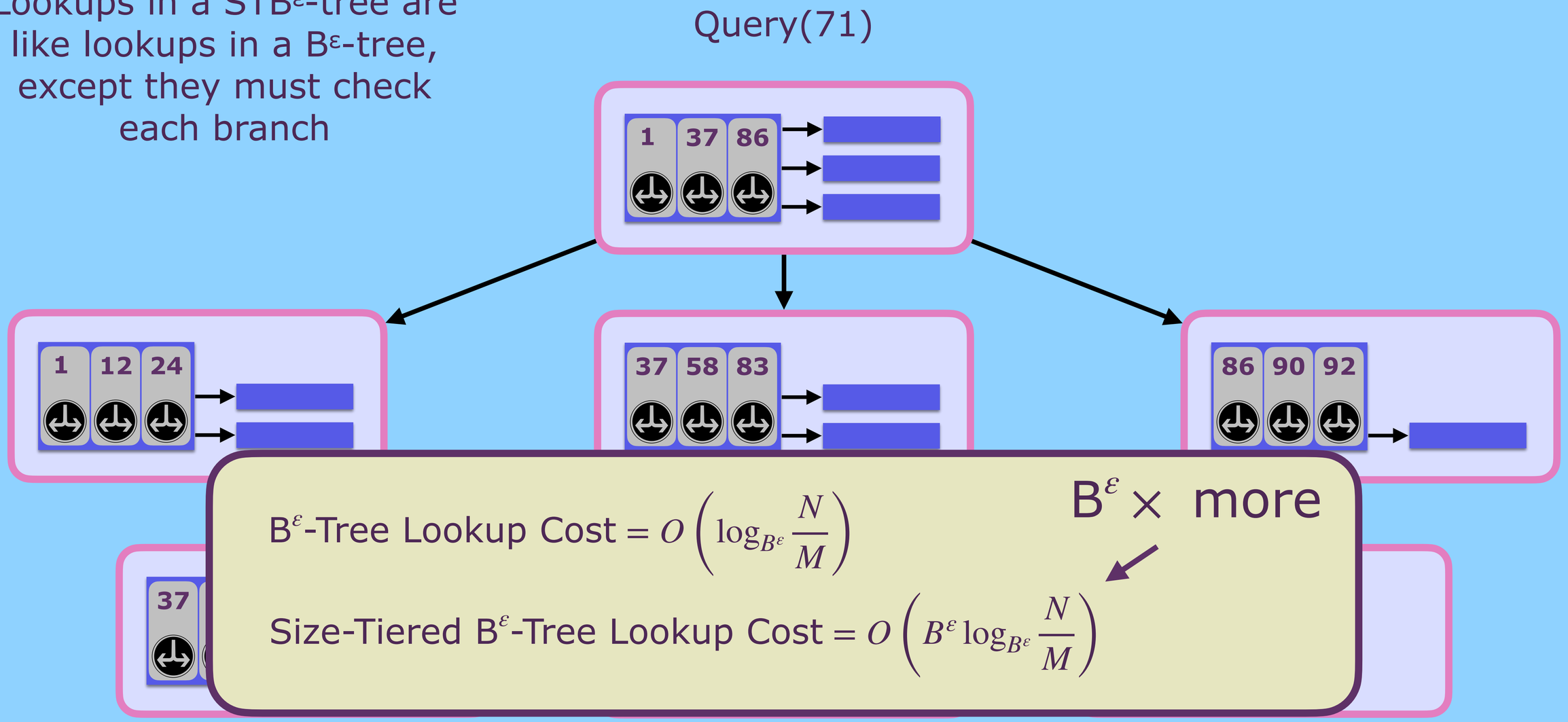


B^ε-Tree Lookup Cost $= O\left(\log_{B^\varepsilon} \frac{N}{M}\right)$

Size-Tiered B^ε-Tree Lookup Cost $= O\left(B^\varepsilon \log_{B^\varepsilon} \frac{N}{M}\right)$

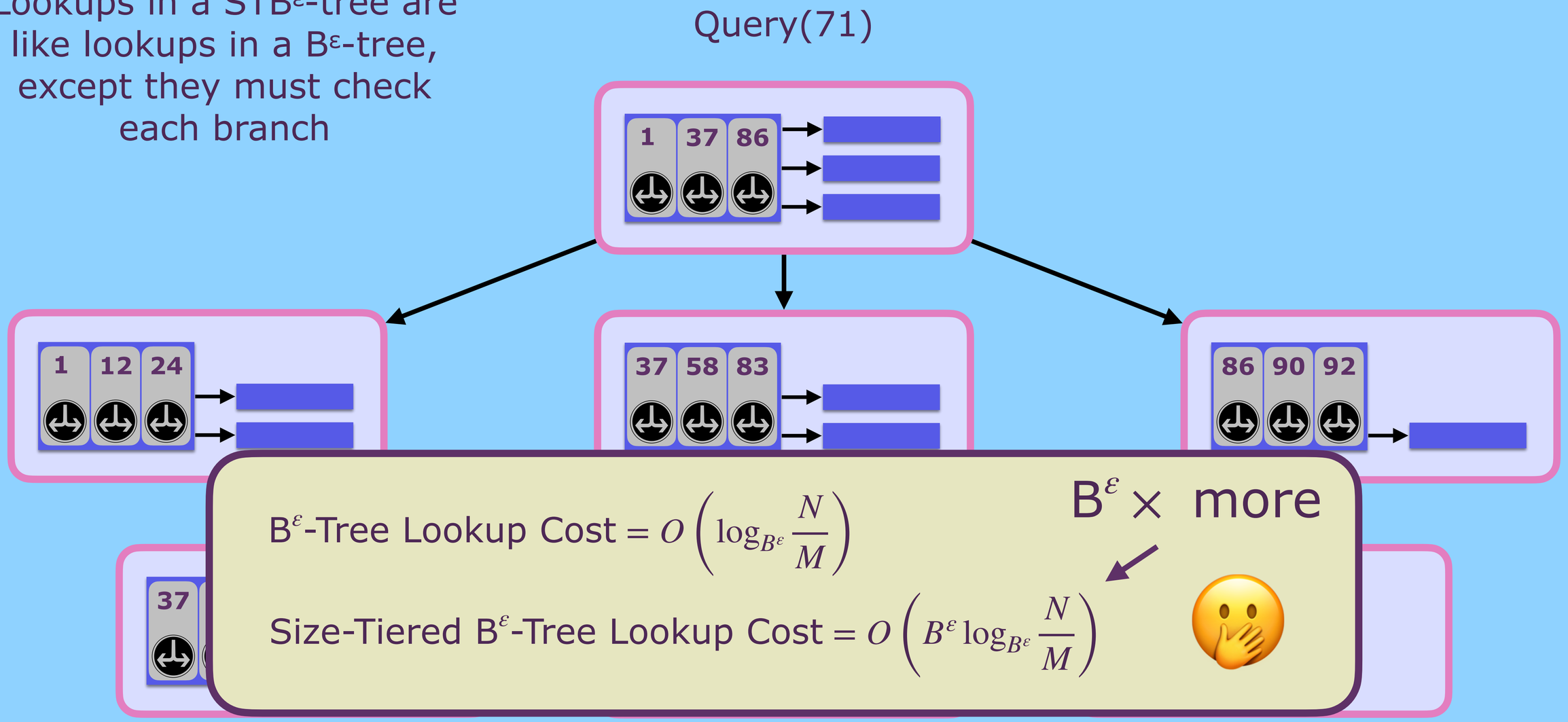
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



Size-Tiered Bε-Trees

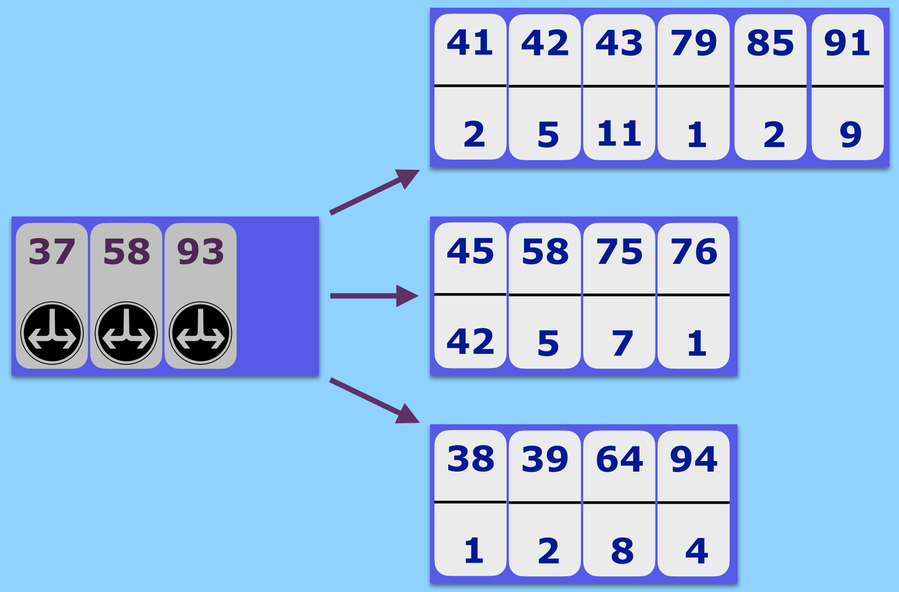
Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch



Fixing Lookups in Size-Tiered B^{ε}-Trees

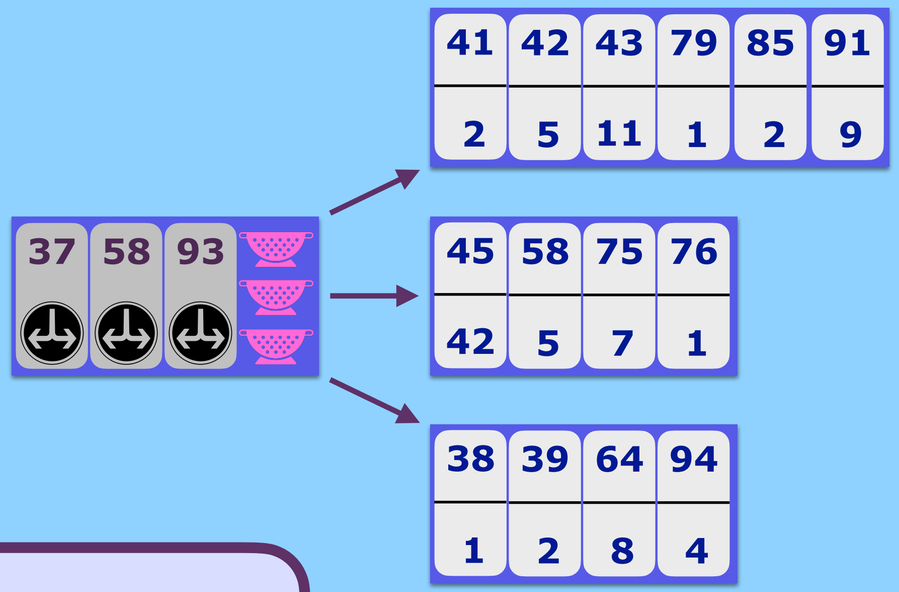
Fixing Lookups in Size-Tiered Bε-Trees

The problem is that each node has multiple branches

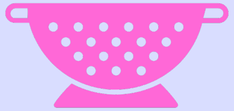


Fixing Lookups in Size-Tiered Bε-Trees

The problem is that each node has multiple branches



Idea: use filters to avoid searching them



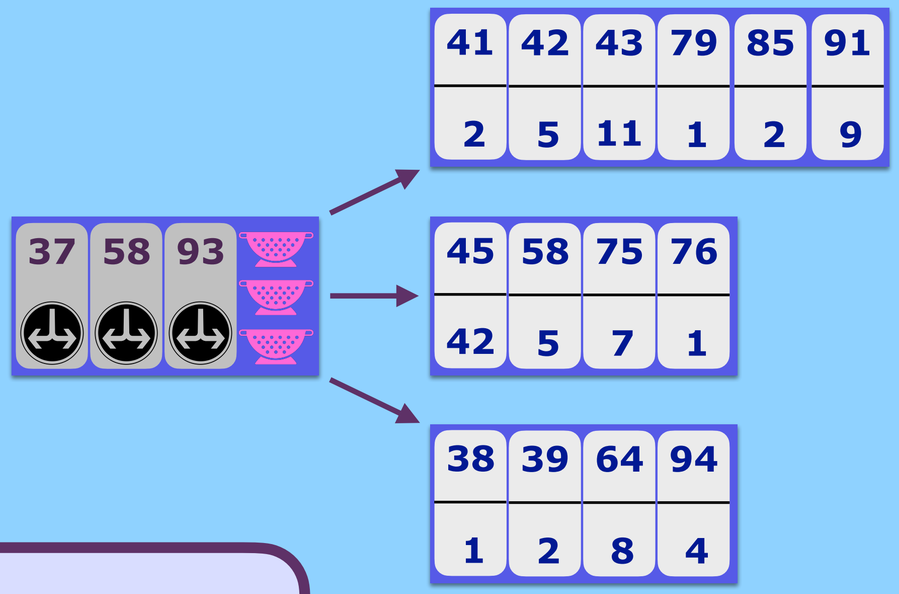
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

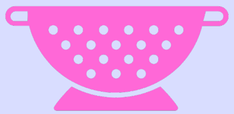
Fixing Lookups in Size-Tiered Bε-Trees

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



Now a lookup will only search those branches which contain the key (plus rare false positives)



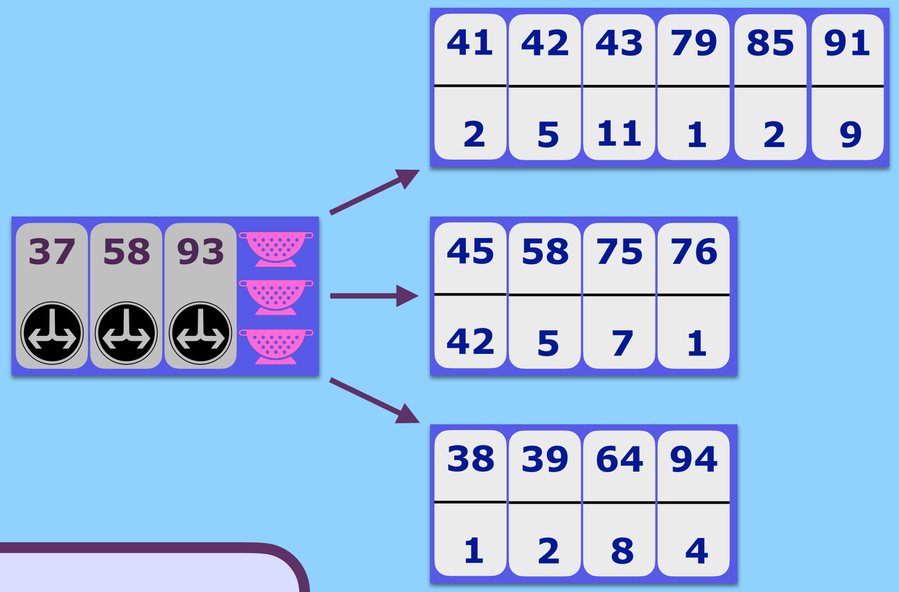
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

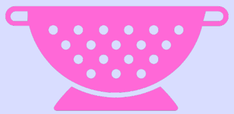
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



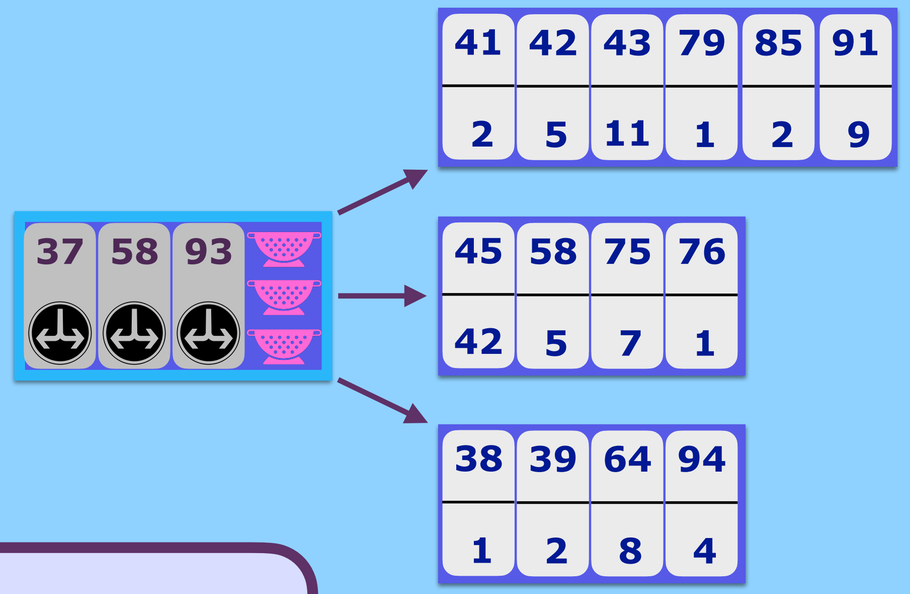
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

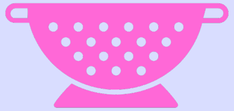
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



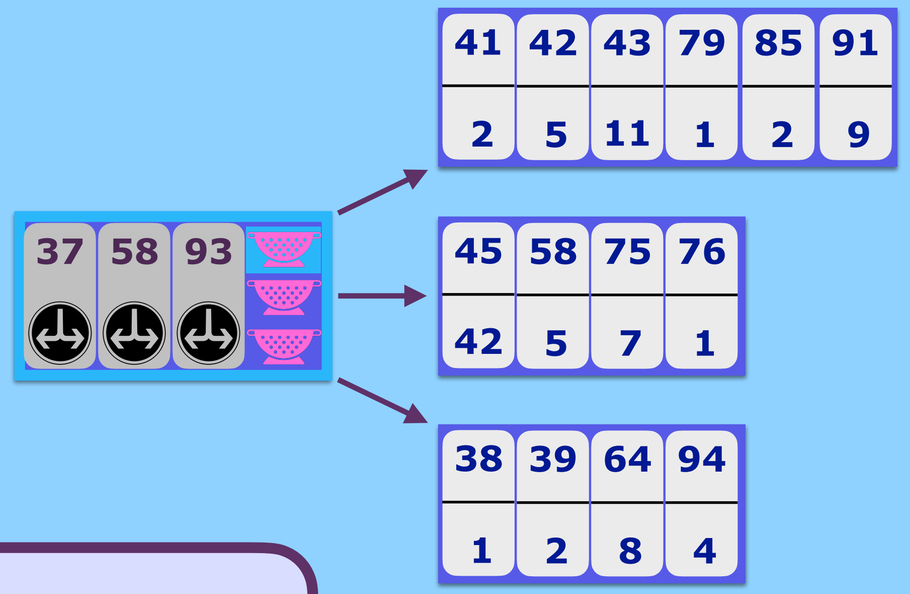
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

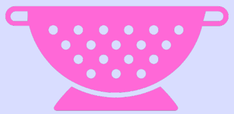
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



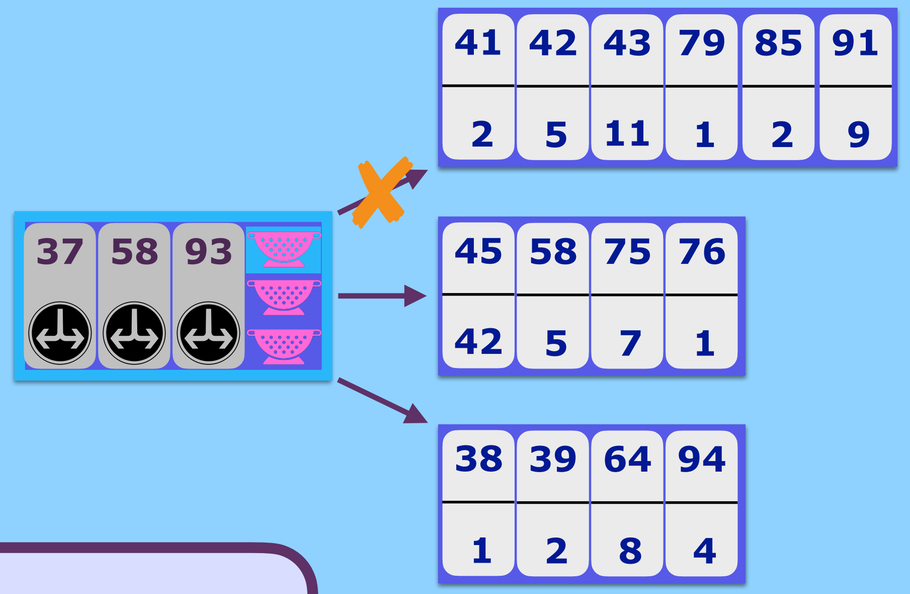
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

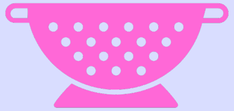
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



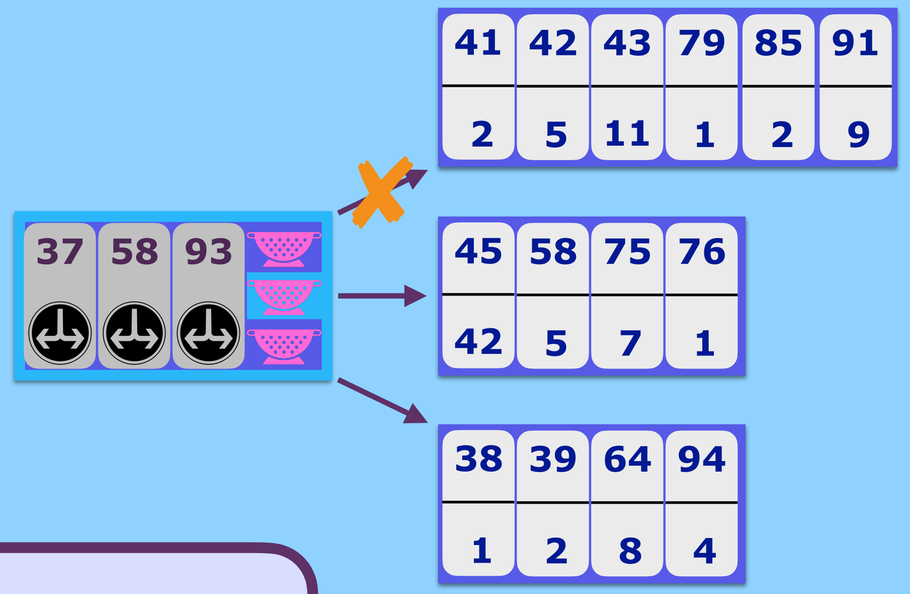
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

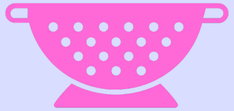
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



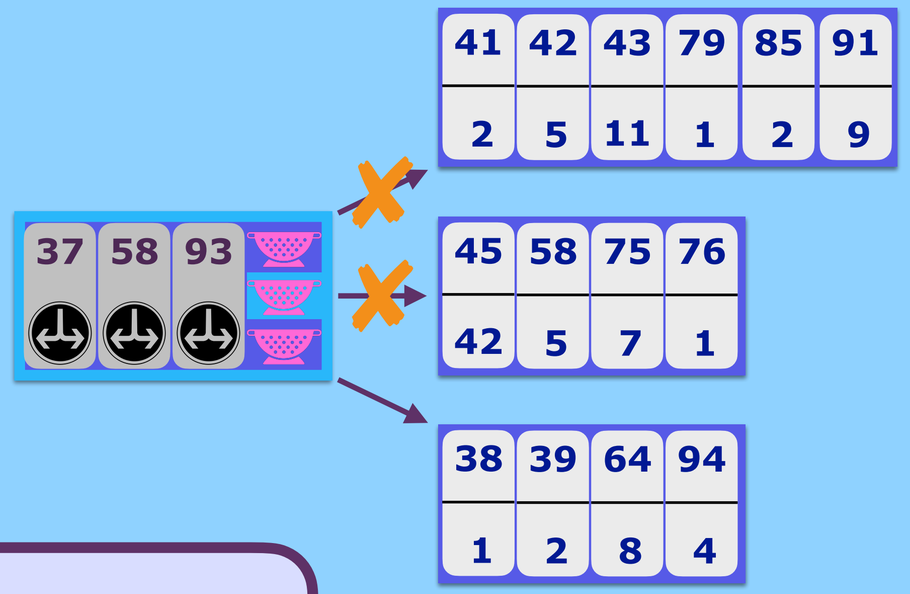
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

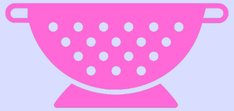
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



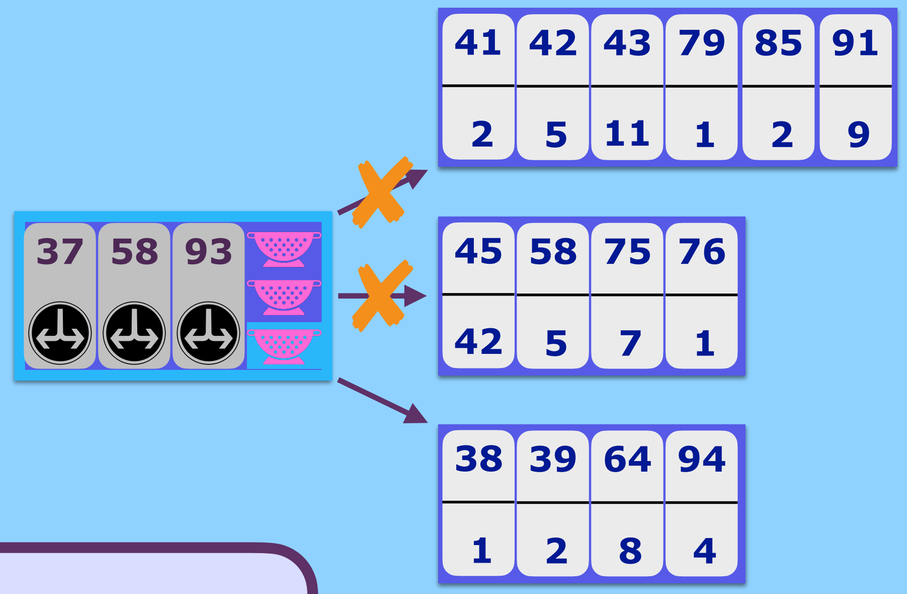
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

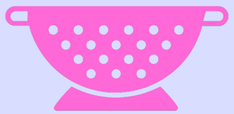
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



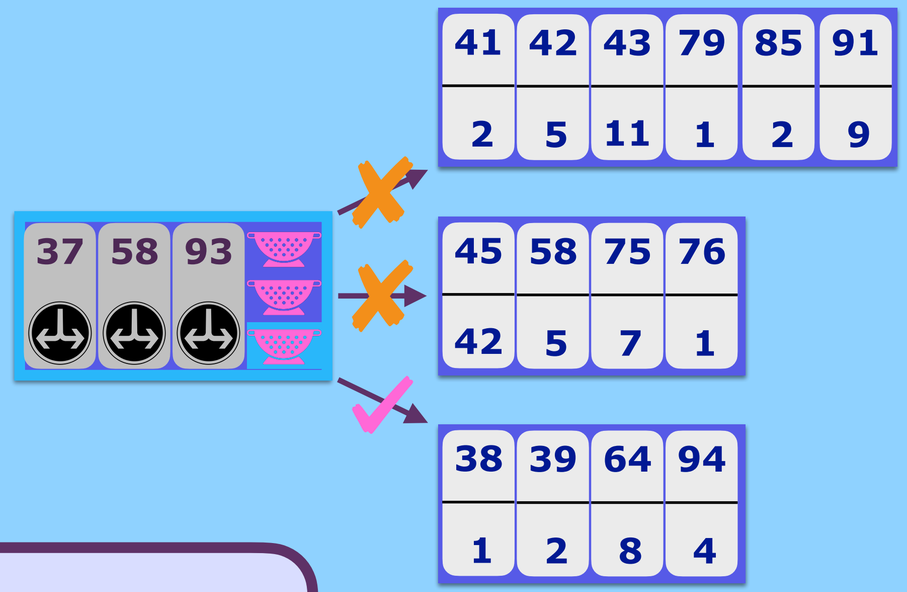
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

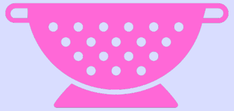
Query(64)

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



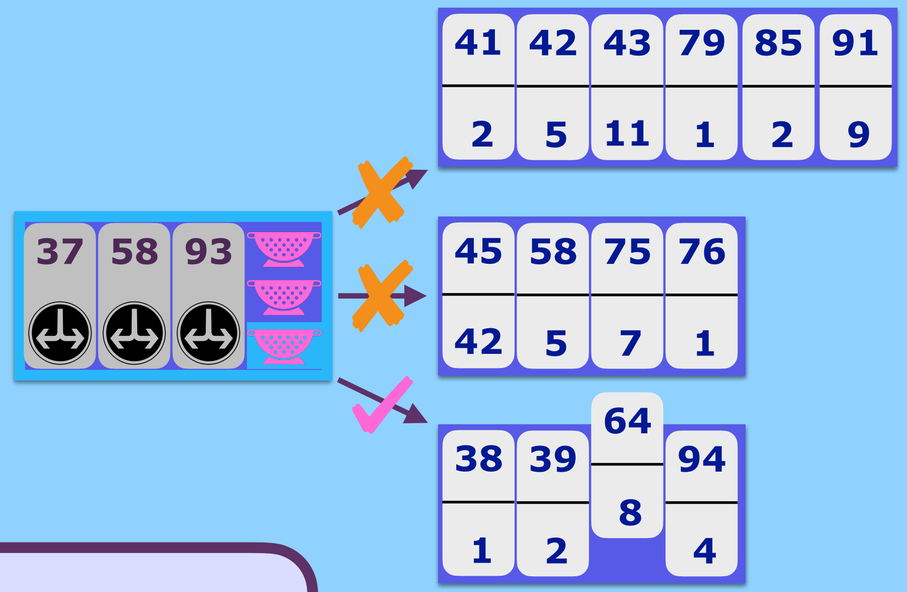
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

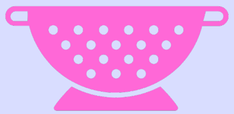
Query(64) → 8

The problem is that each node has multiple branches



Now a lookup will only search those branches which contain the key (plus rare false positives)

Idea: use filters to avoid searching them



A filter is a probabilistic data structure with answers membership with no false negatives

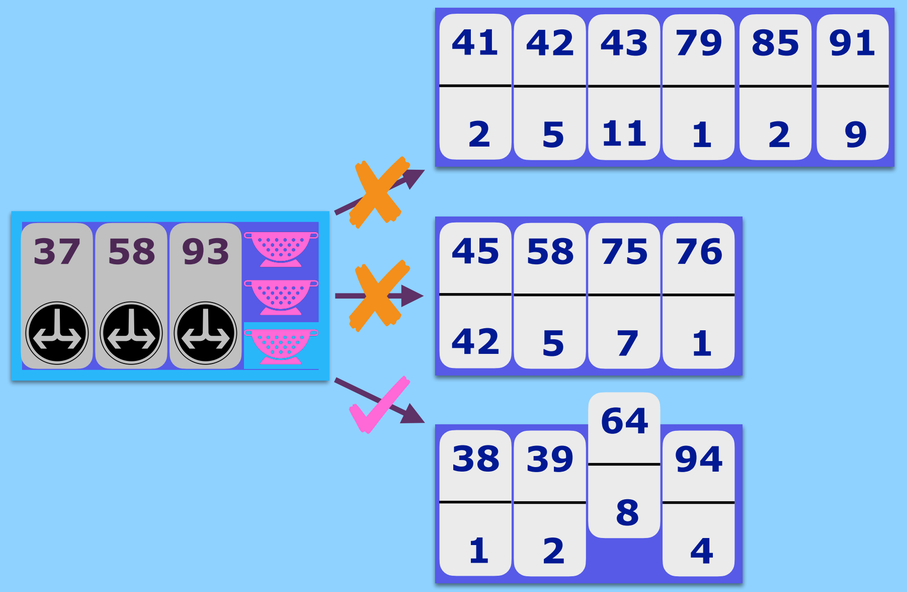
Examples: Bloom, cuckoo, quotient

Fixing Lookups in Size-Tiered Bε-Trees

Query(64) → 8

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



Now a lookup will only search those branches which contain the key (plus rare false positives)

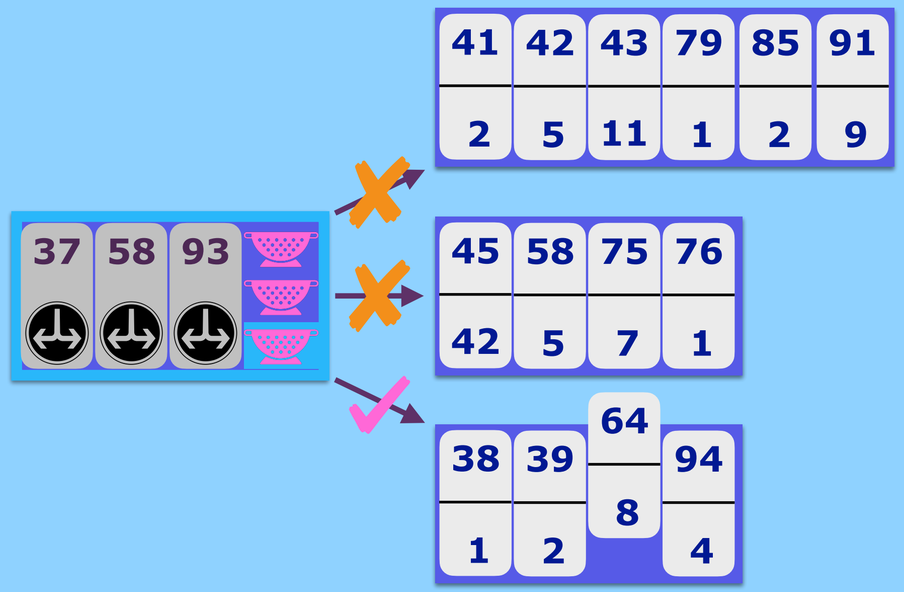
False Positive Rate $\leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_B N}\right)$

Fixing Lookups in Size-Tiered Bε-Trees

Query(64) → 8

The problem is that each node has multiple branches

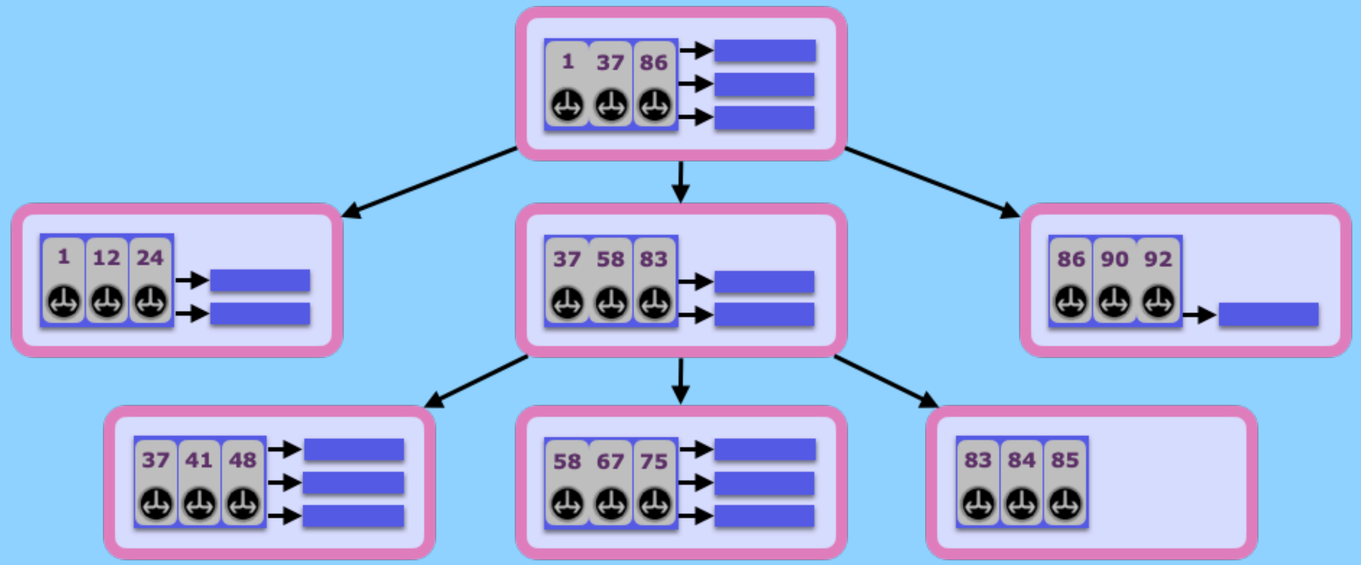
Idea: use filters to avoid searching them



Now a lookup will only search those branches which contain the key (plus rare false positives)

False Positive Rate $\leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_B N}\right)$ $\Rightarrow$ Lookups in O(1) IOs

Size-Tiered Bε-Tree



Less compaction

Less IO

Less CPU

Low lookup cost

Scans

See the text!

Short — more expensive

Long — disk bandwidth

In this talk

Fast Storage
(NVMe)

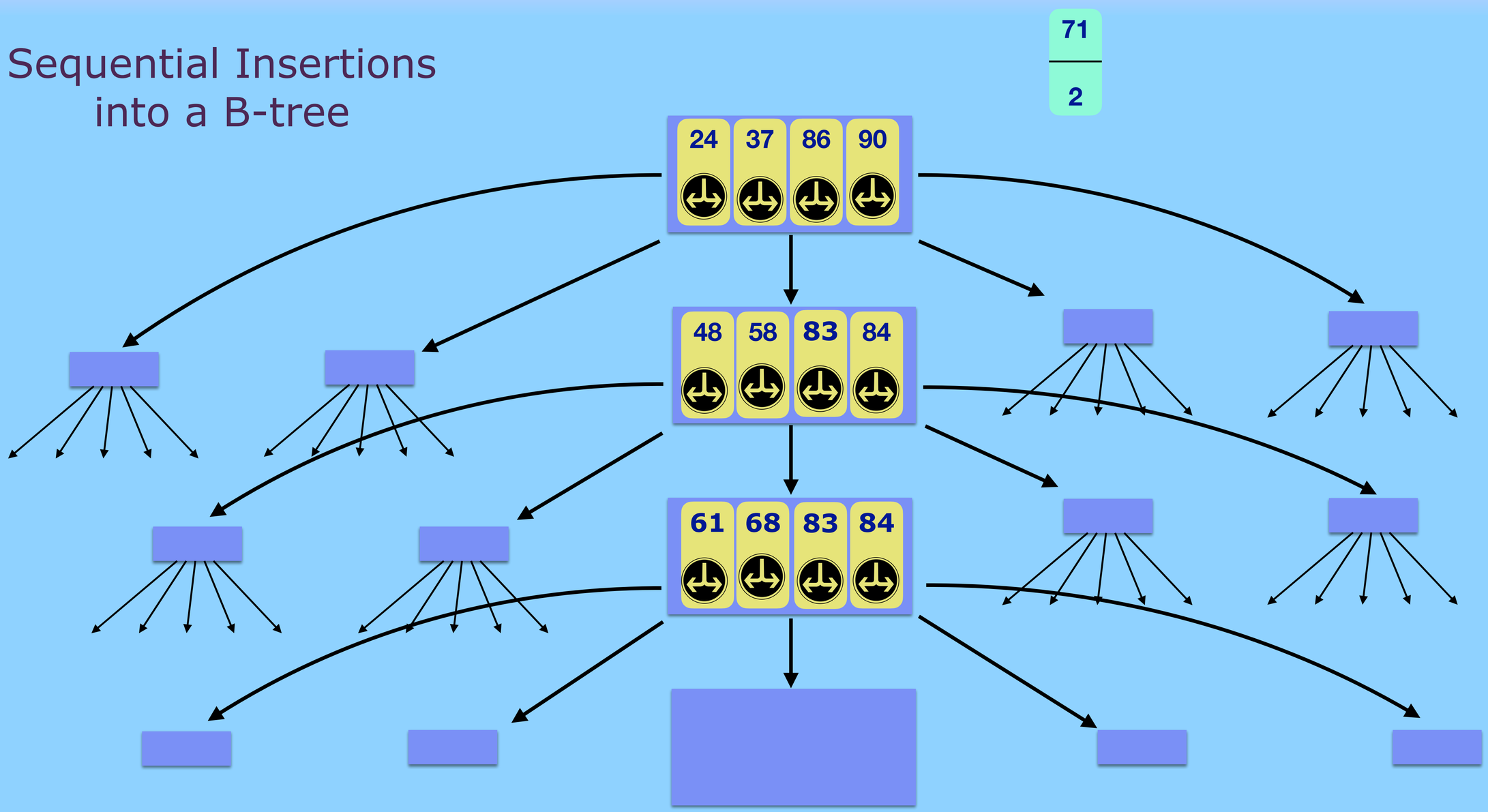
SplinterDB

Data Structures

Flush-then-Compact

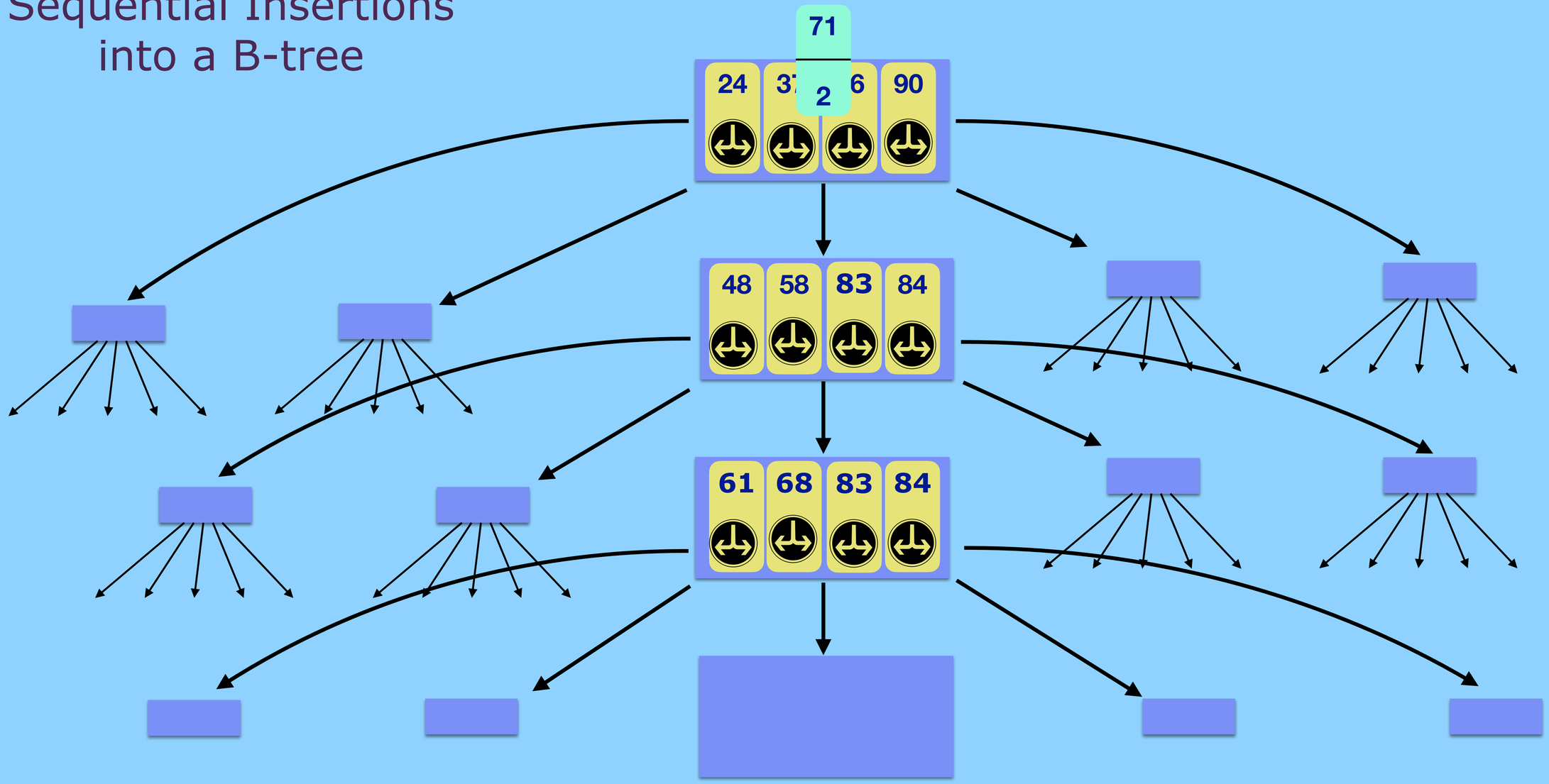
Flush-Then-Compact

Sequential Insertions
into a B-tree



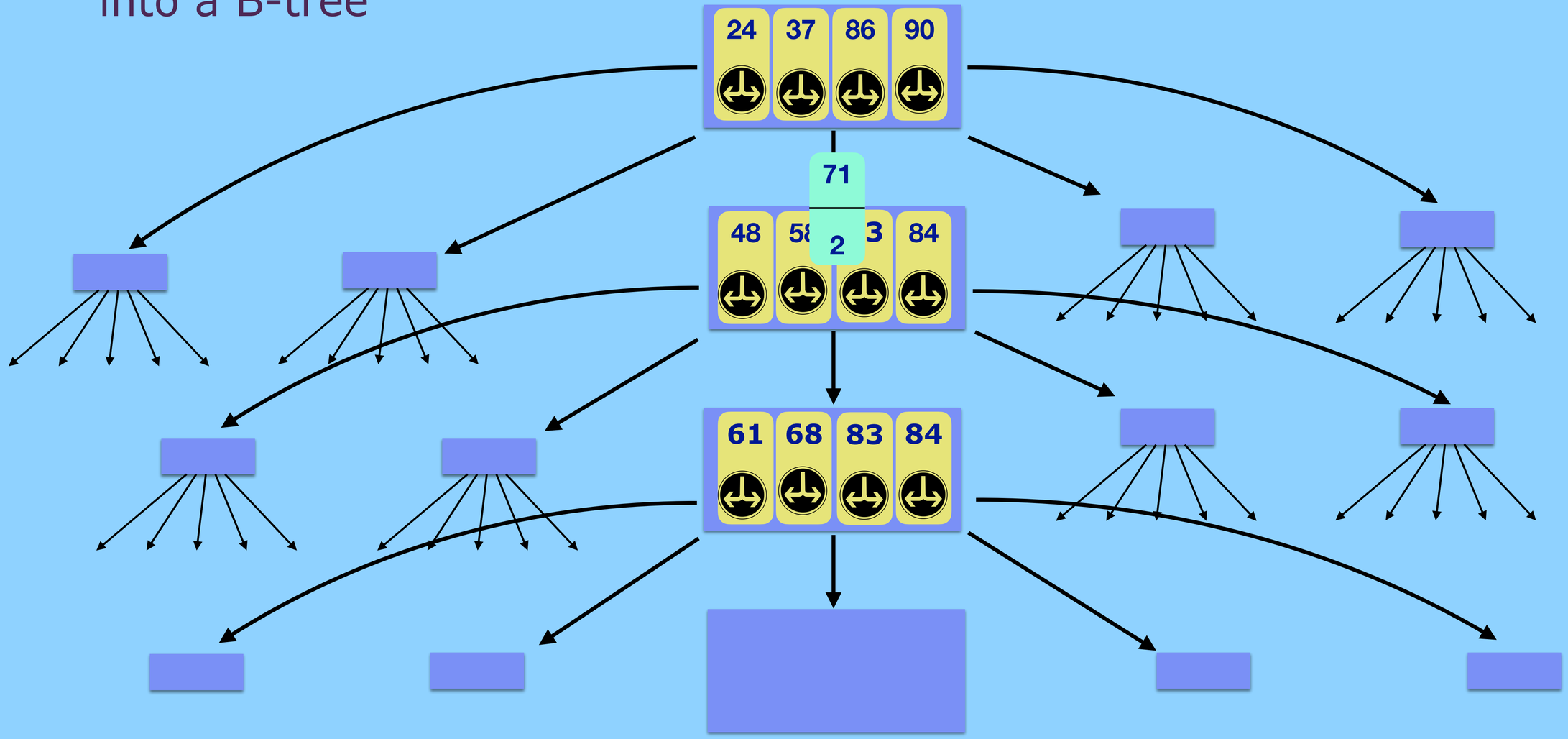
Flush-Then-Compact

Sequential Insertions
into a B-tree



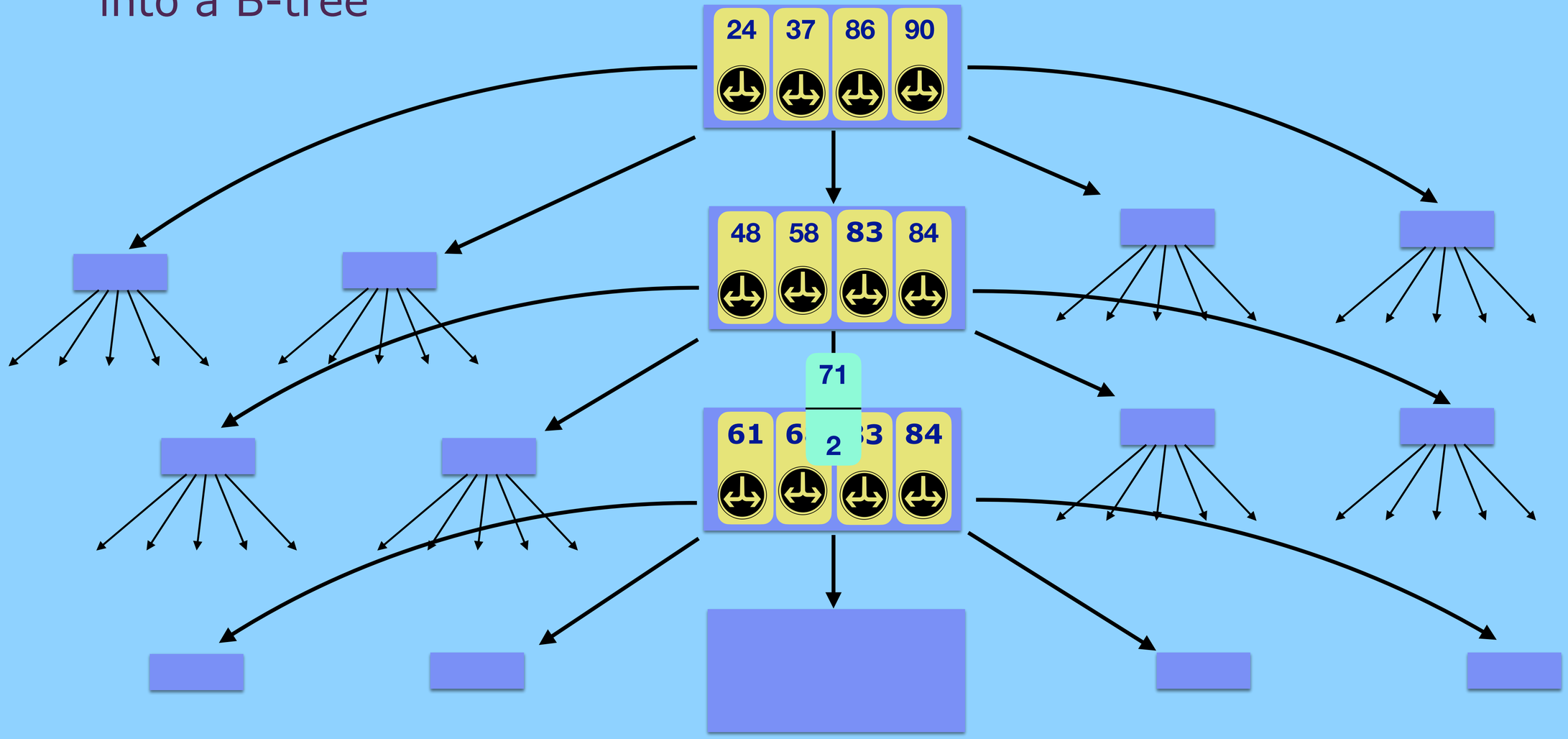
Flush-Then-Compact

Sequential Insertions
into a B-tree



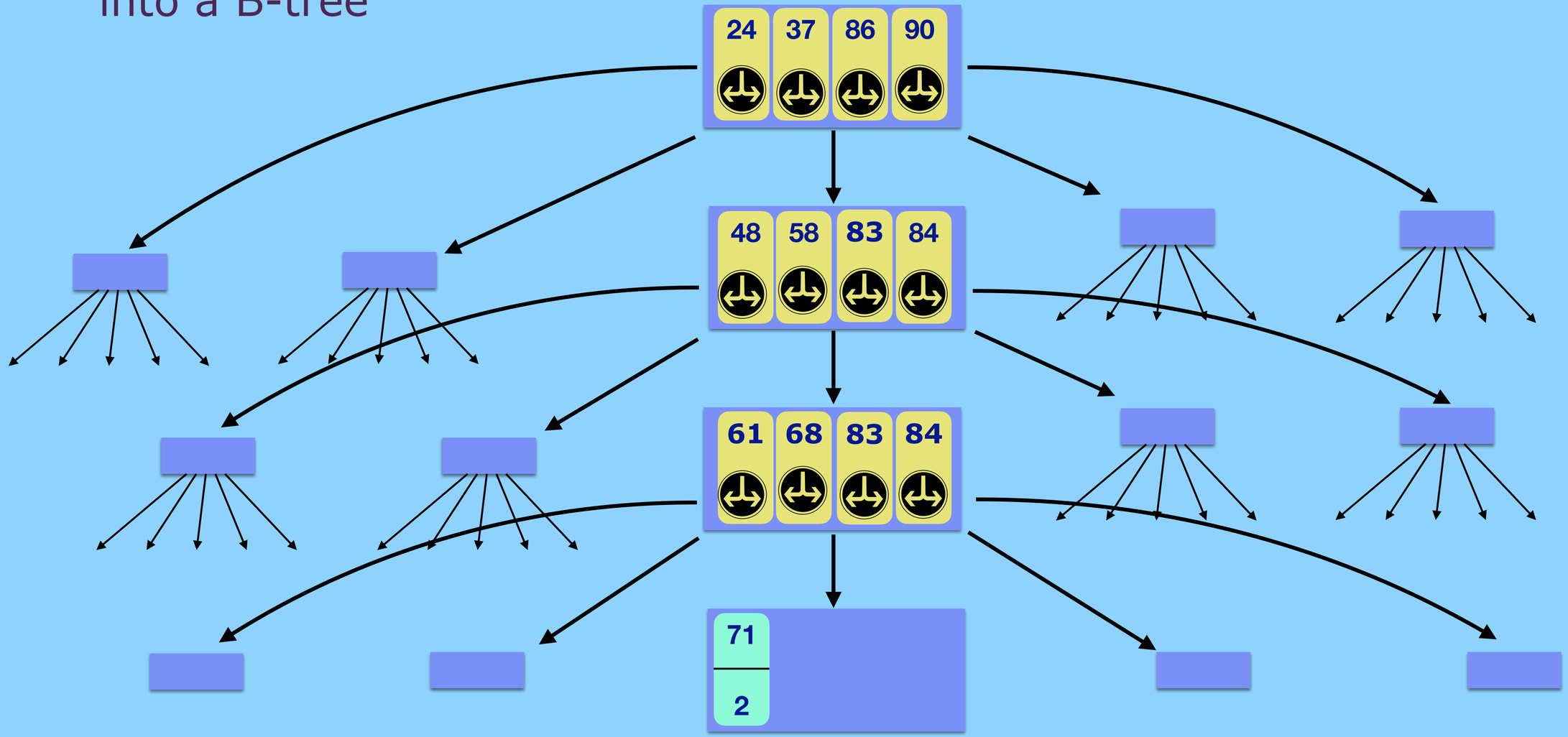
Flush-Then-Compact

Sequential Insertions
into a B-tree



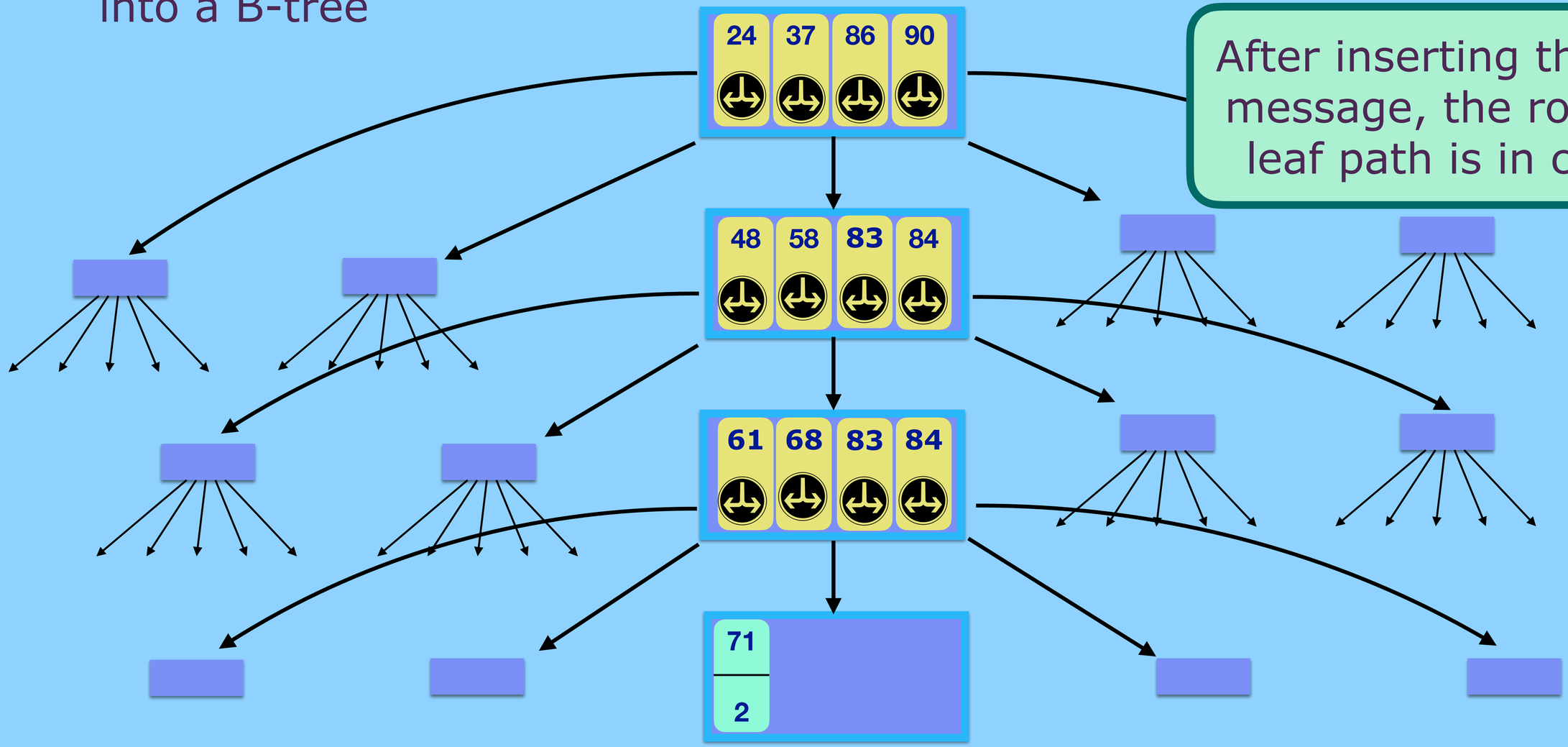
Flush-Then-Compact

Sequential Insertions
into a B-tree



Flush-Then-Compact

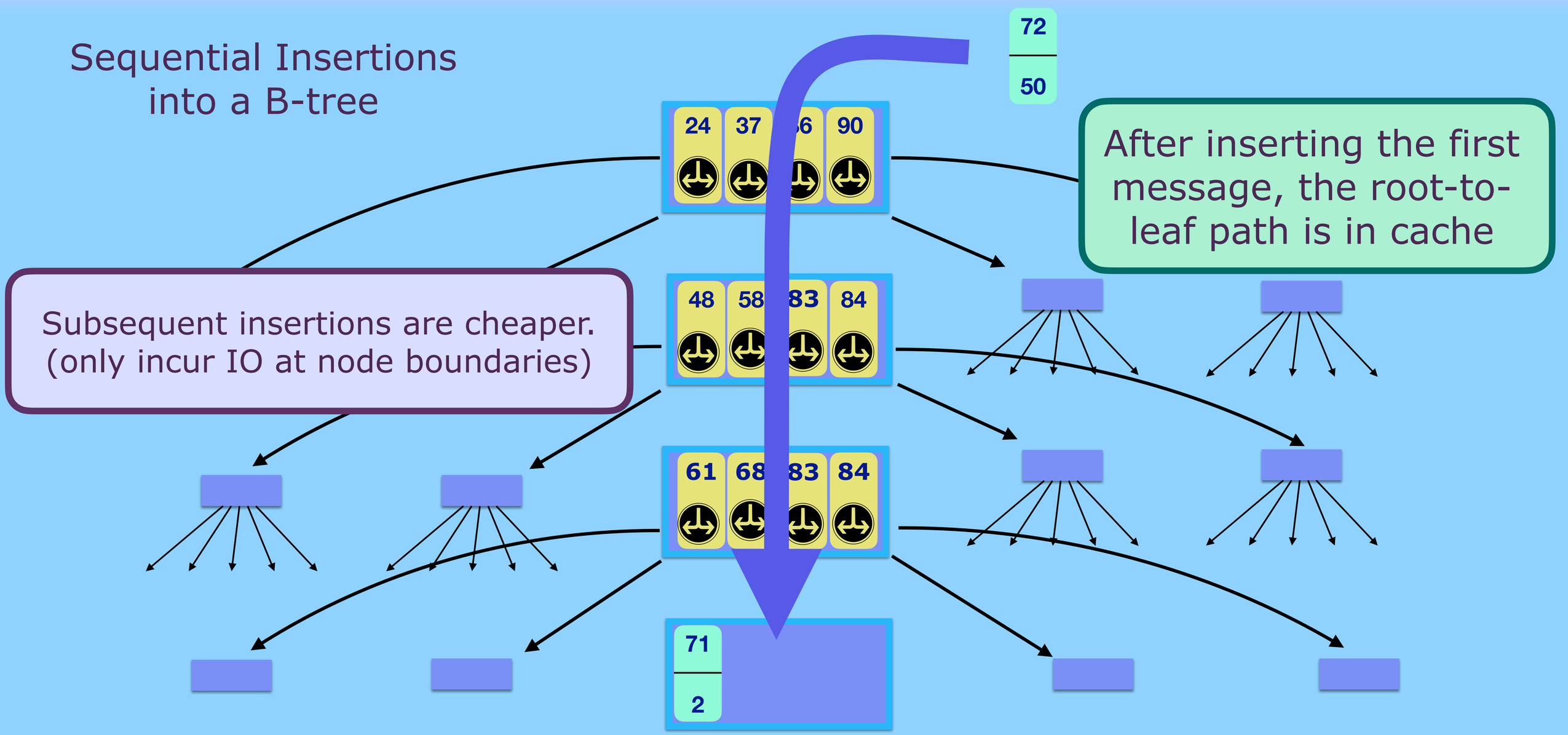
Sequential Insertions into a B-tree



After inserting the first message, the root-to-leaf path is in cache

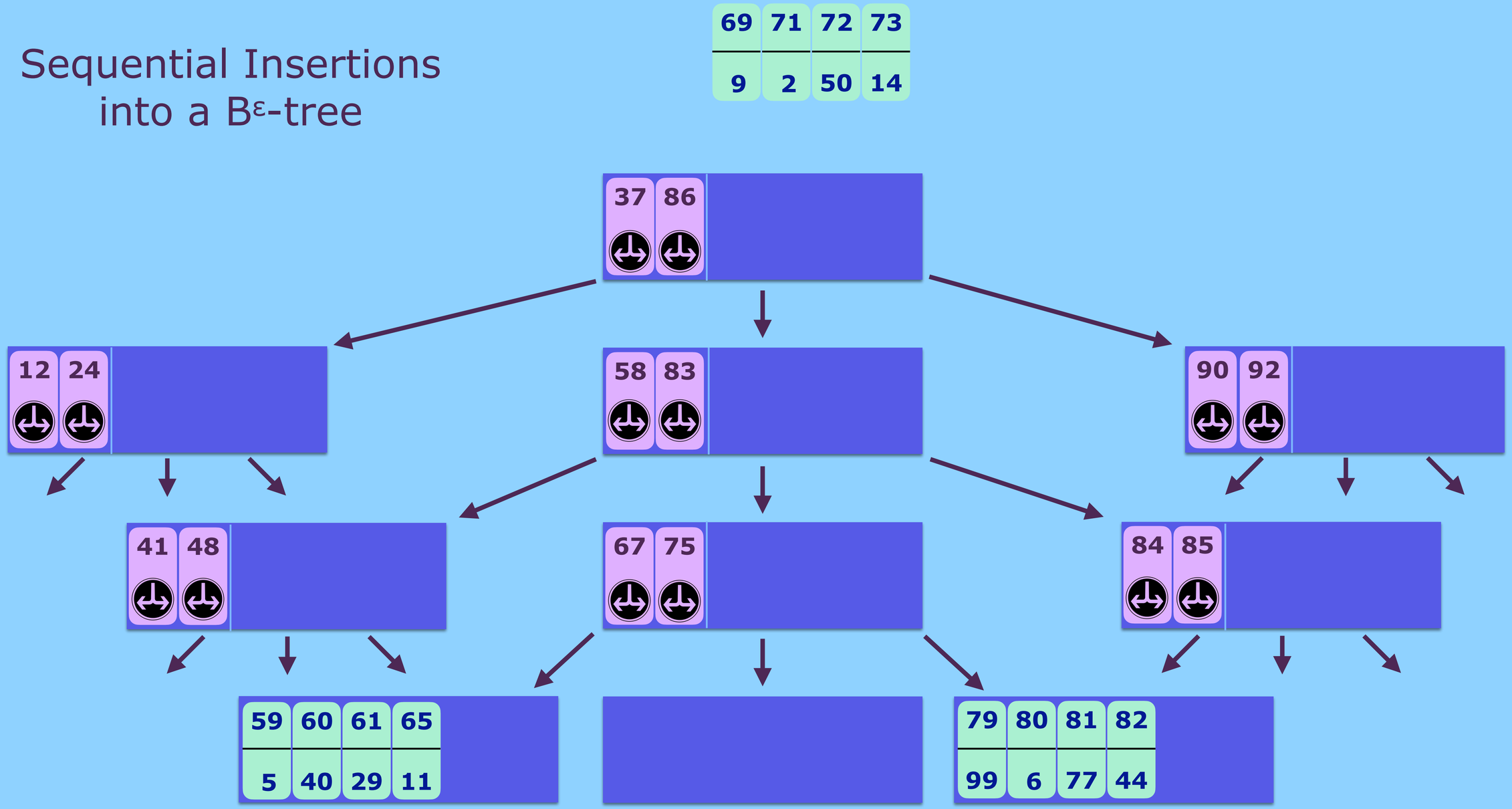
Flush-Then-Compact

Sequential Insertions into a B-tree



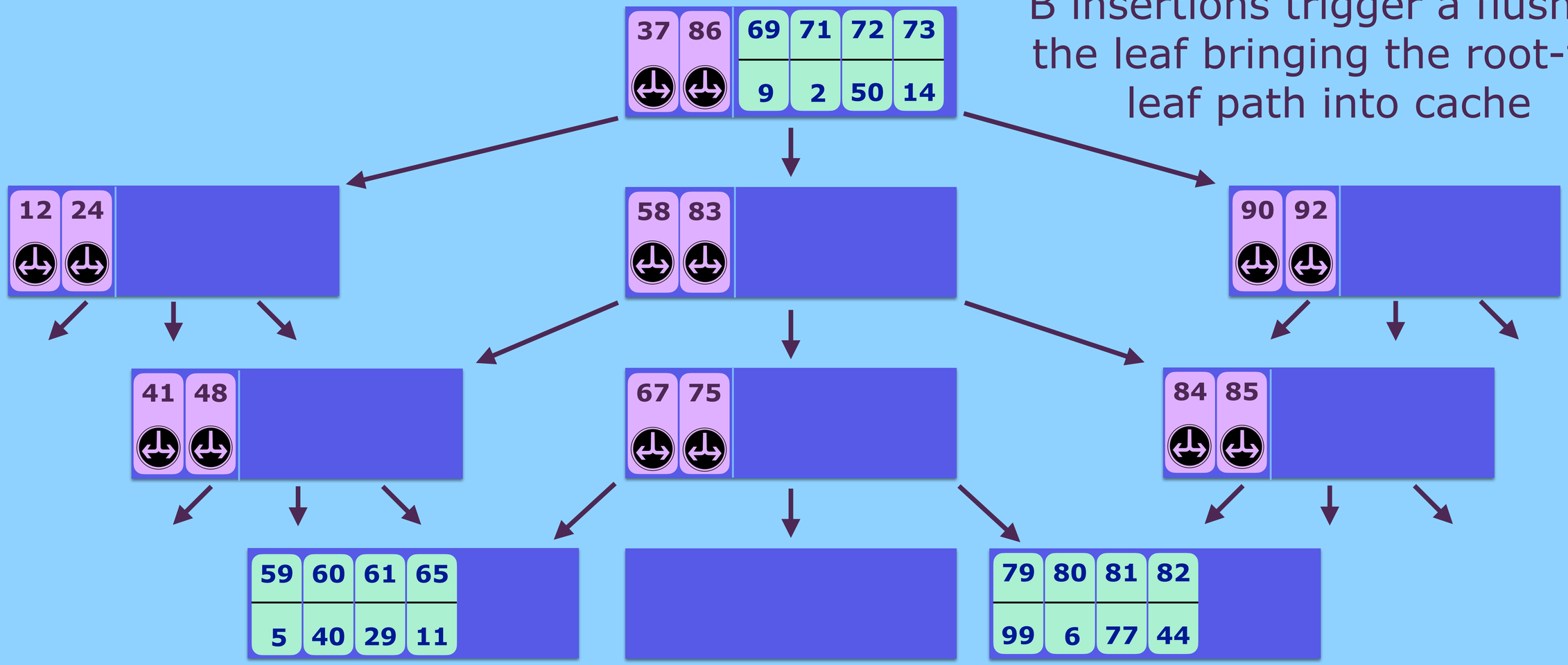
Flush-Then-Compact

Sequential Insertions into a B$^{\varepsilon}$-tree



Flush-Then-Compact

Sequential Insertions into a B$^{\varepsilon}$-tree

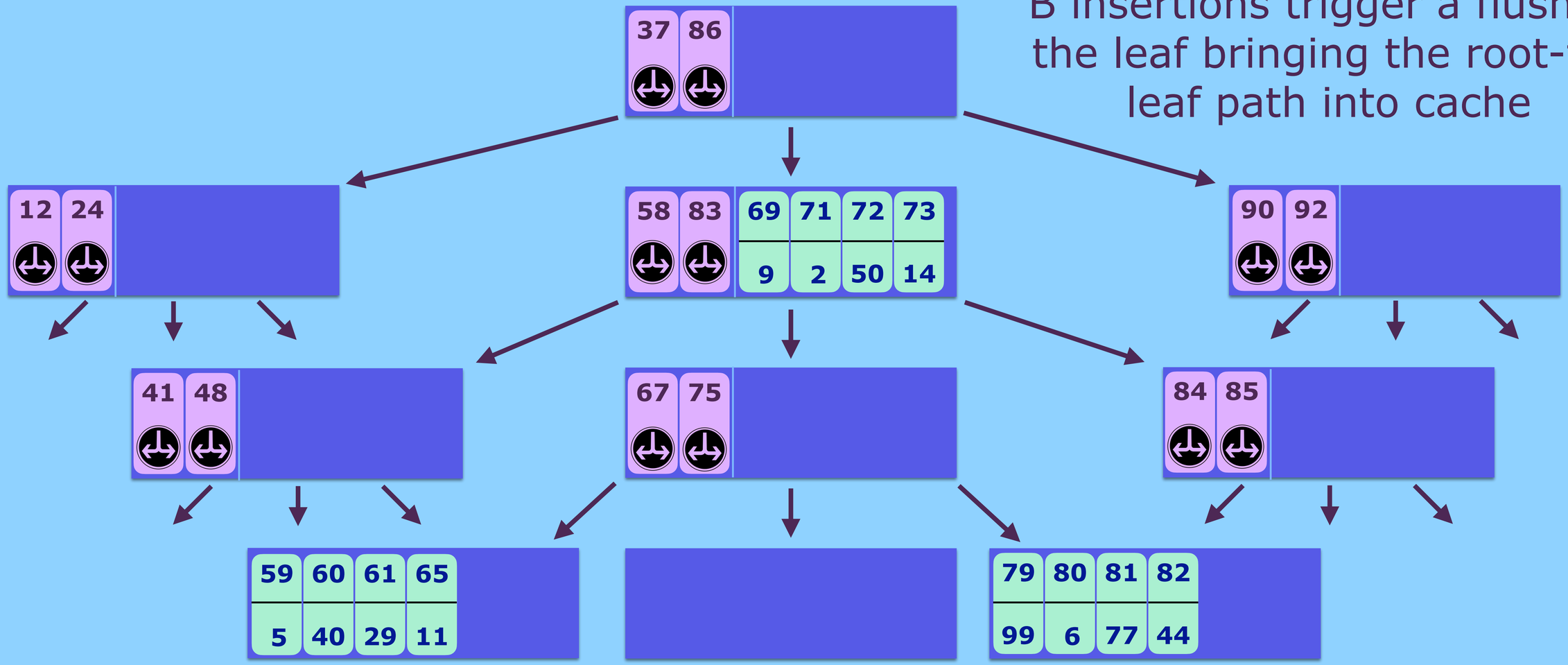


B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

Flush-Then-Compact

Sequential Insertions into a B$^{\varepsilon}$-tree

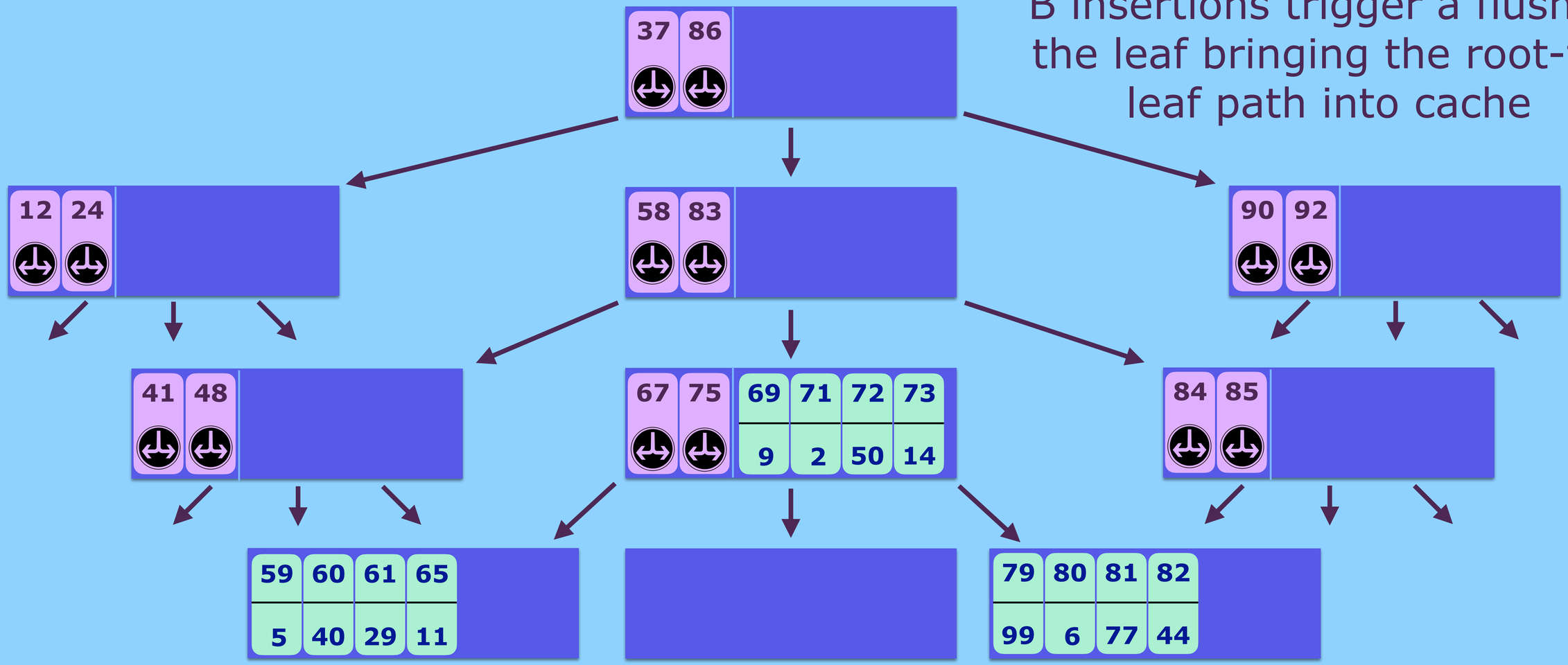
B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache



Flush-Then-Compact

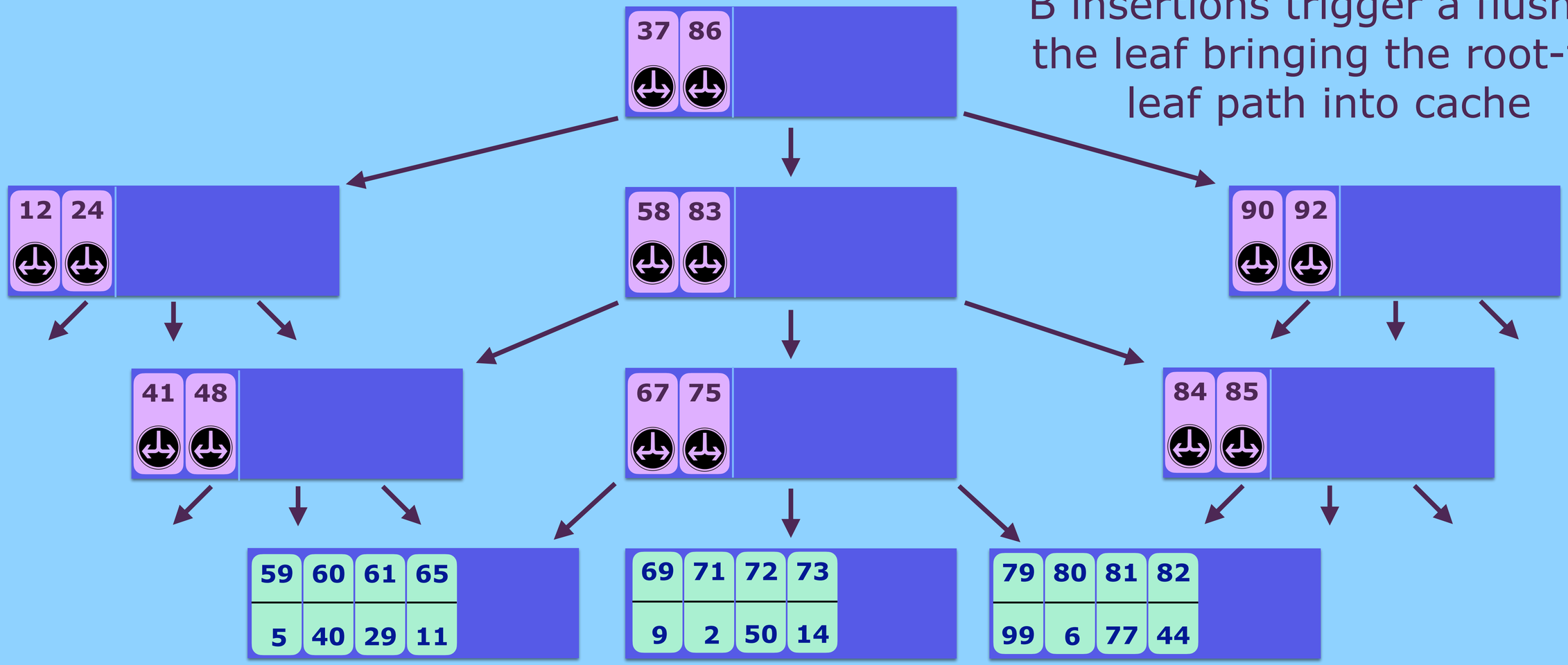
Sequential Insertions into a B$^{\varepsilon}$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache



Flush-Then-Compact

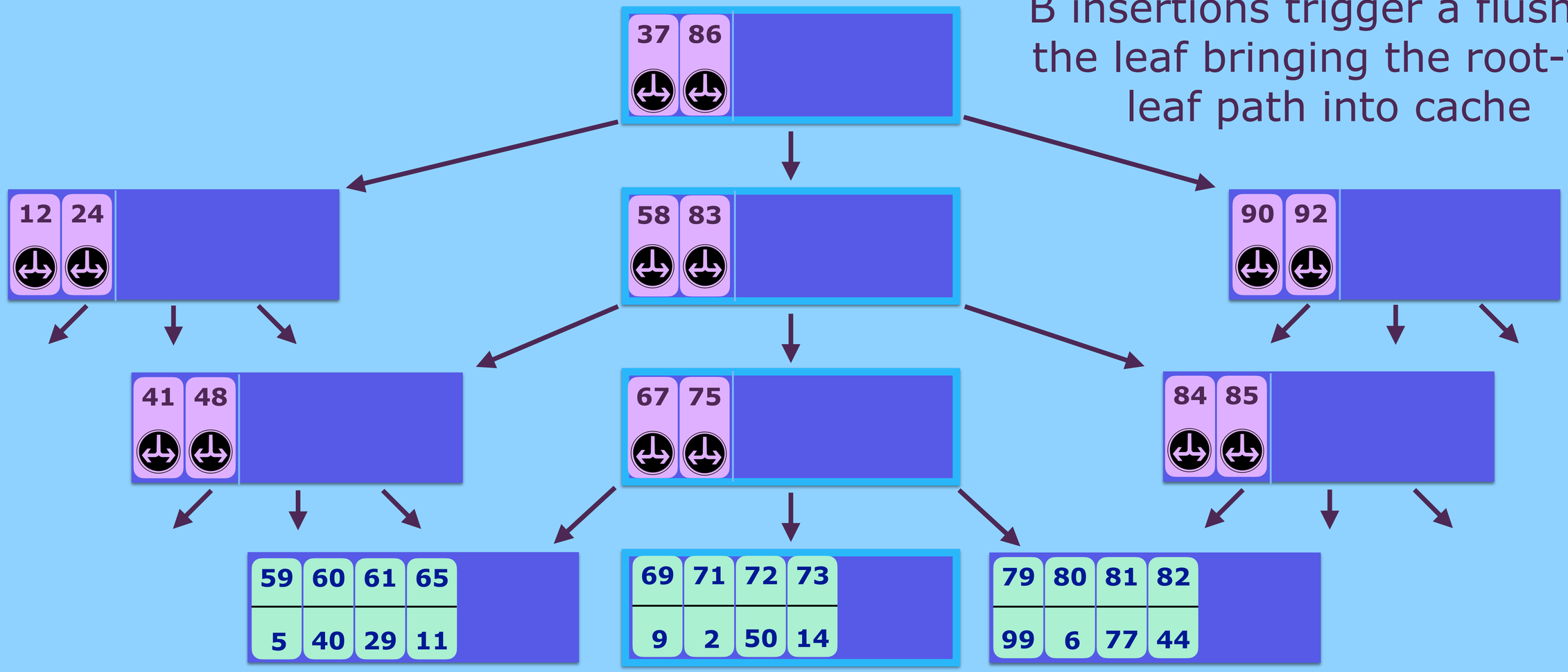
Sequential Insertions into a B$^{\varepsilon}$-tree



B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

Flush-Then-Compact

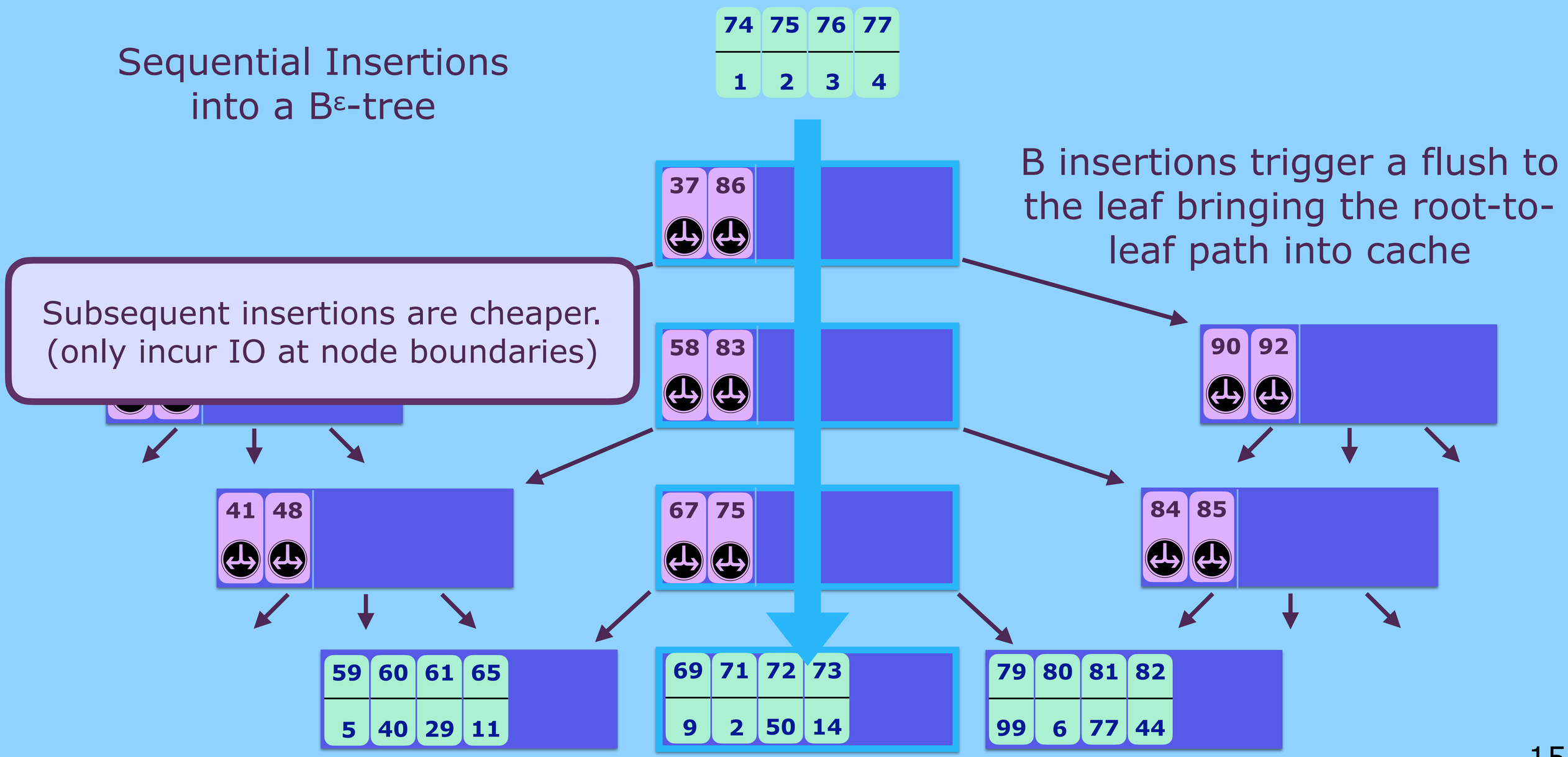
Sequential Insertions into a B$^{\varepsilon}$-tree



B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

Flush-Then-Compact

Sequential Insertions into a B$^{\varepsilon}$-tree

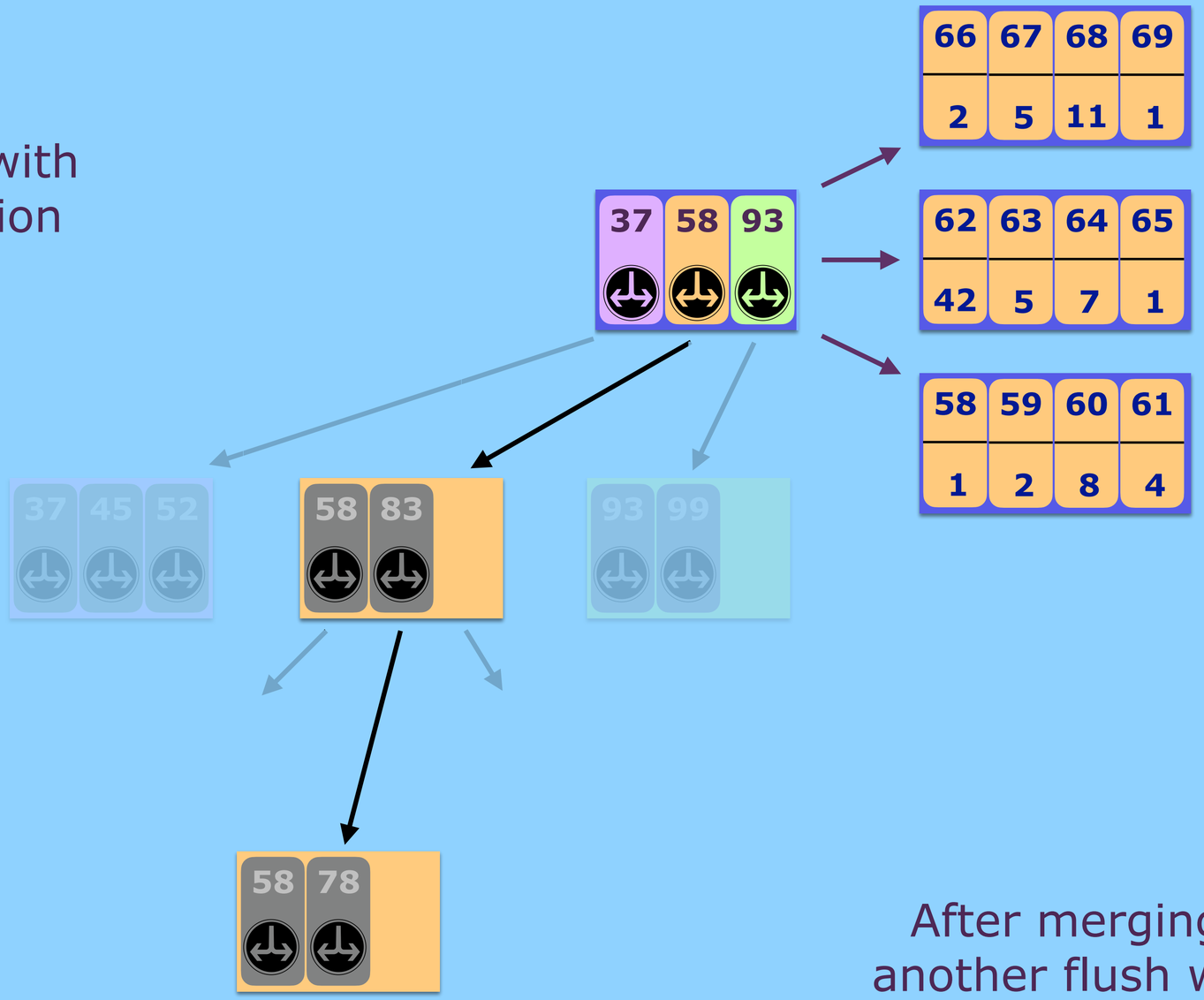


B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

Subsequent insertions are cheaper. (only incur IO at node boundaries)

Flush-Then-Compact

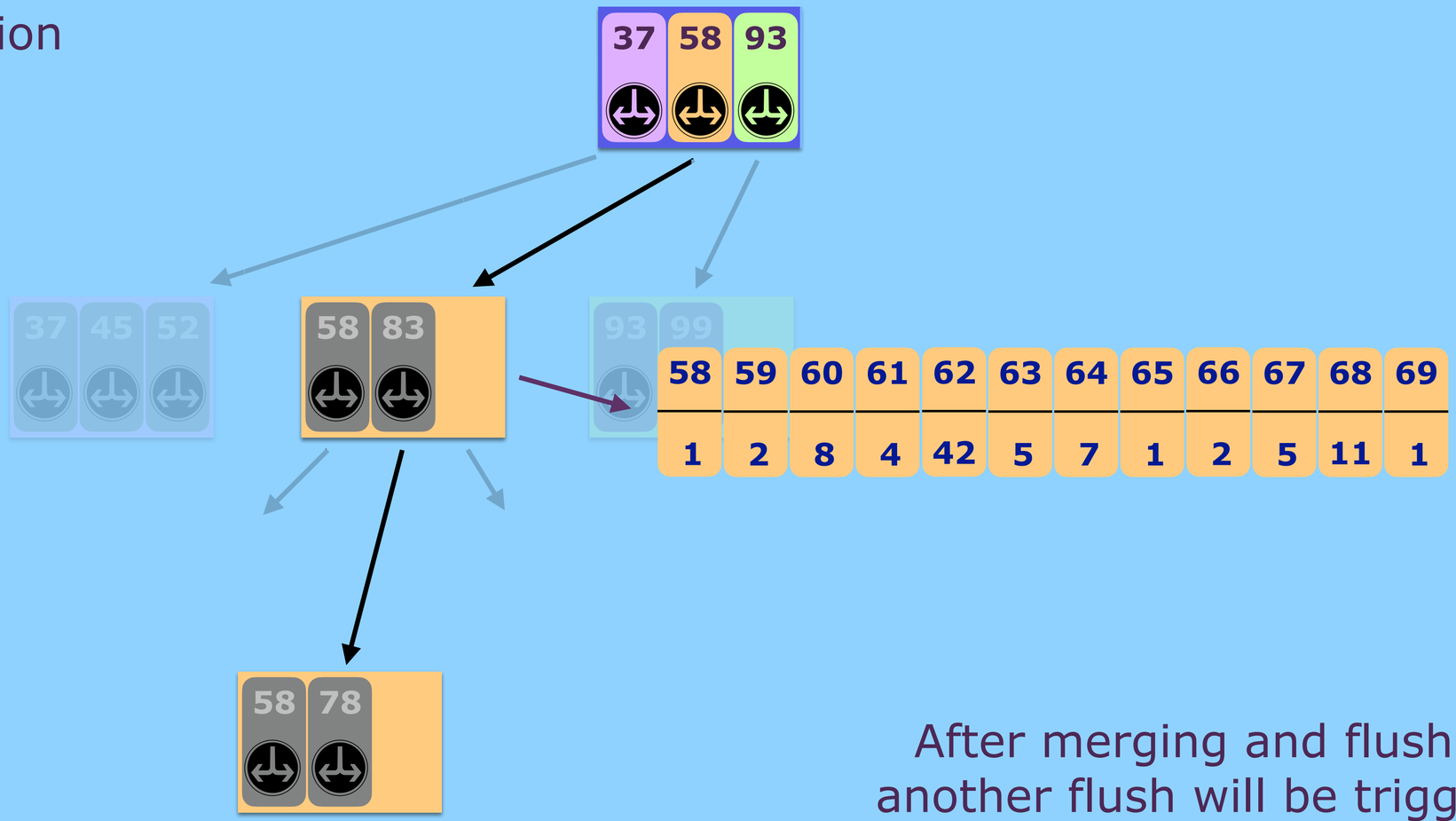
Want:
Sequential insertions with
lower work amplification



After merging and flushing
another flush will be triggered

Flush-Then-Compact

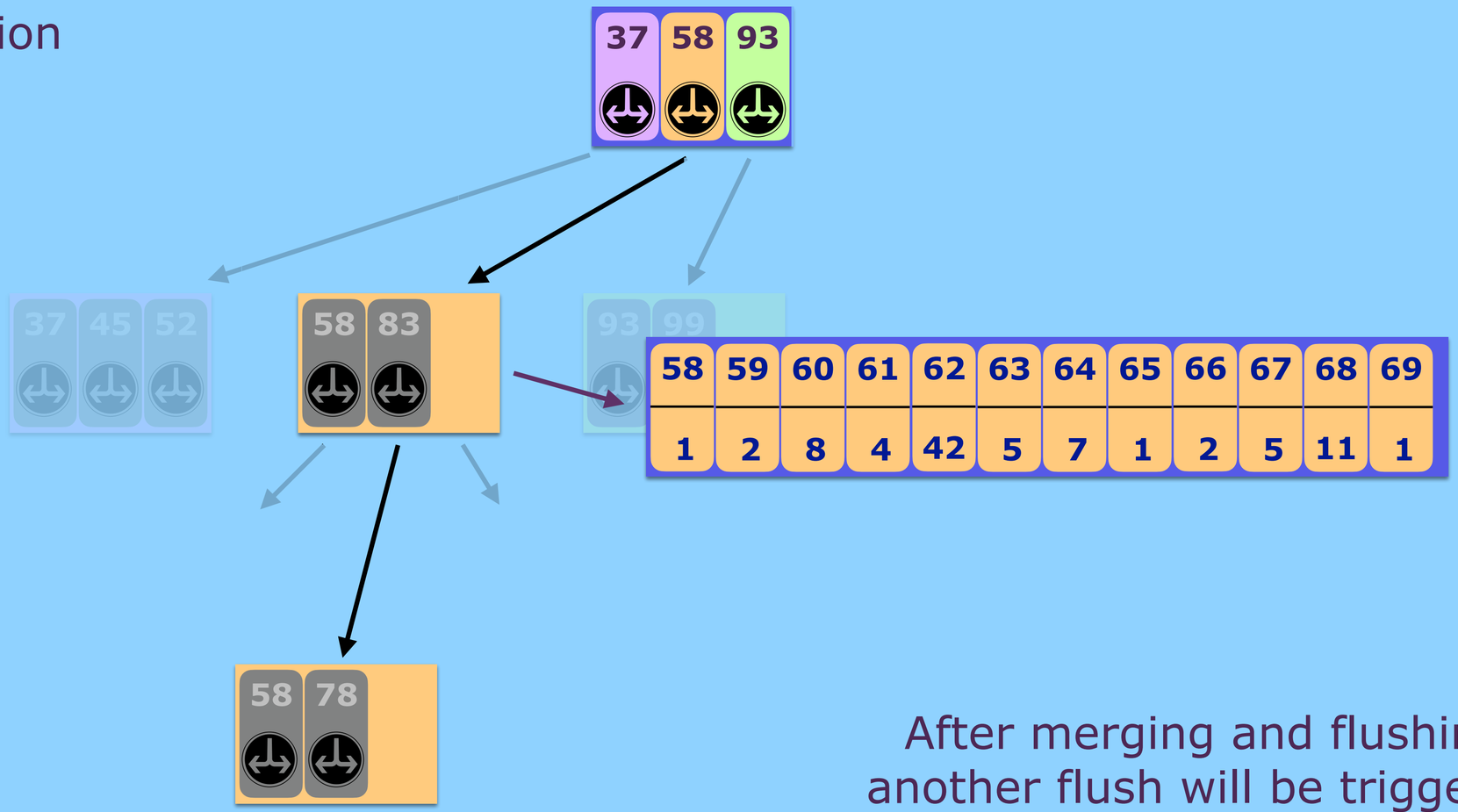
Want:
Sequential insertions with
lower work amplification



After merging and flushing
another flush will be triggered

Flush-Then-Compact

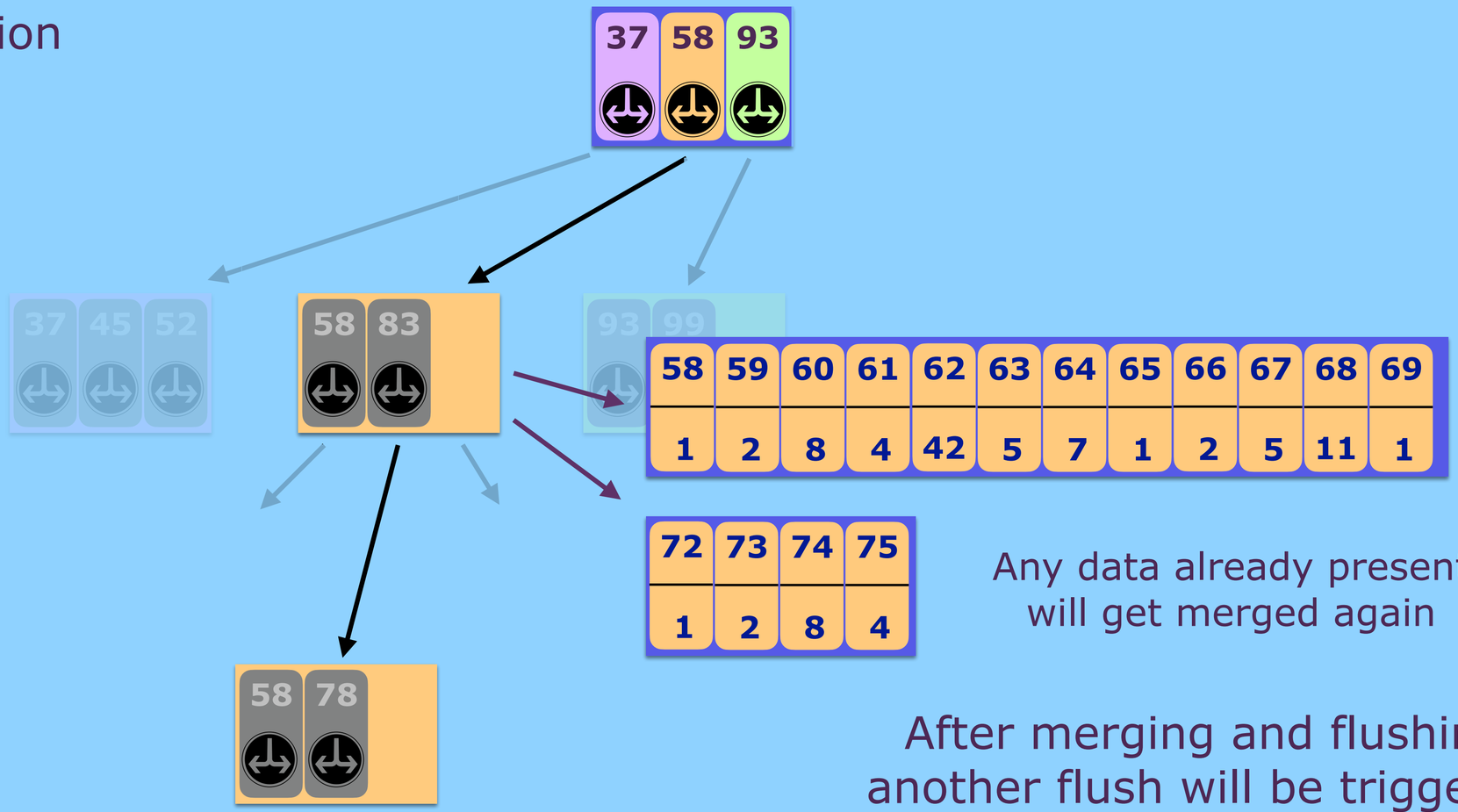
Want:
Sequential insertions with
lower work amplification



After merging and flushing
another flush will be triggered

Flush-Then-Compact

Want:
Sequential insertions with
lower work amplification

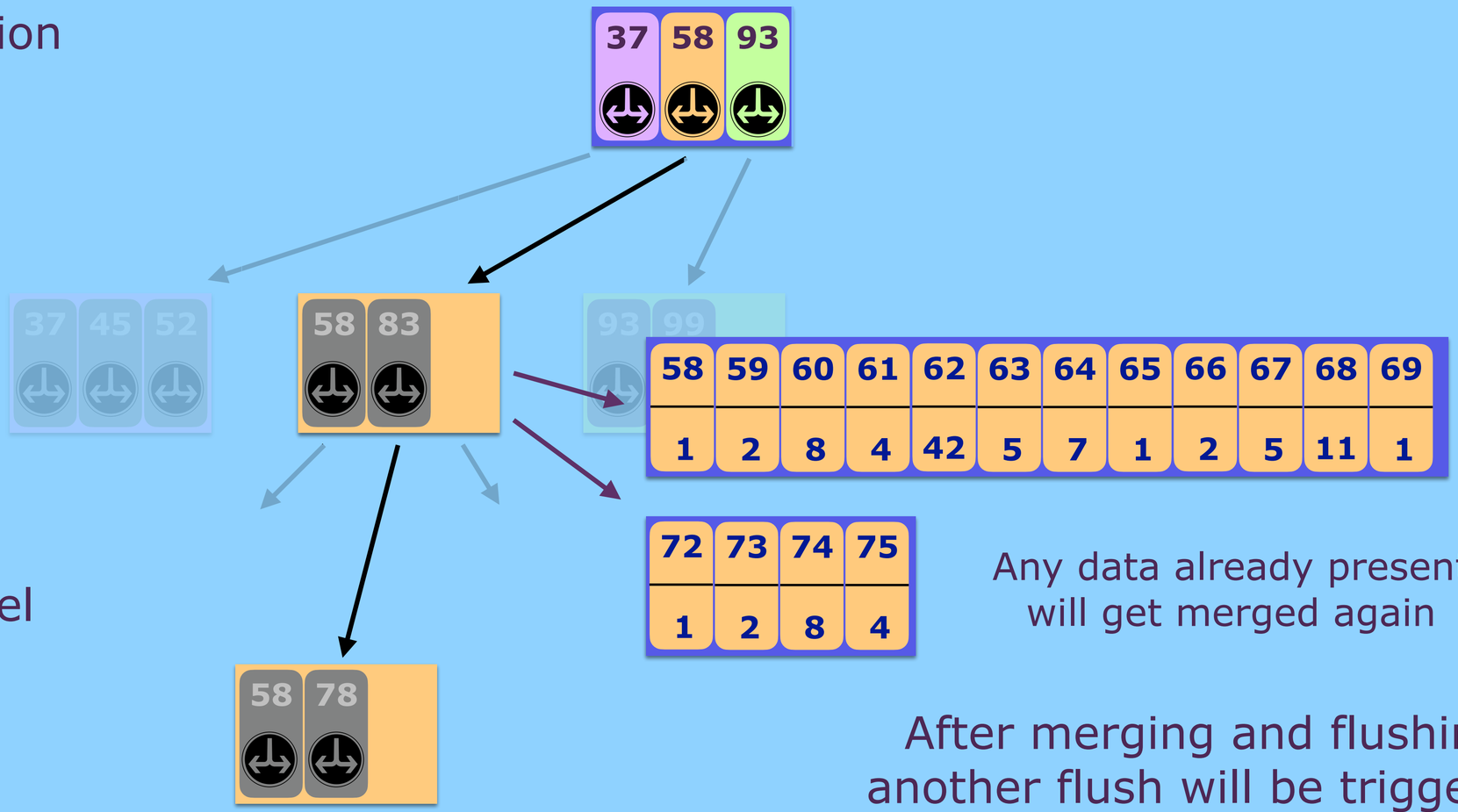


Any data already present
will get merged again

After merging and flushing
another flush will be triggered

Flush-Then-Compact

Want:
Sequential insertions with
lower work amplification



Can still end up
merging on each level

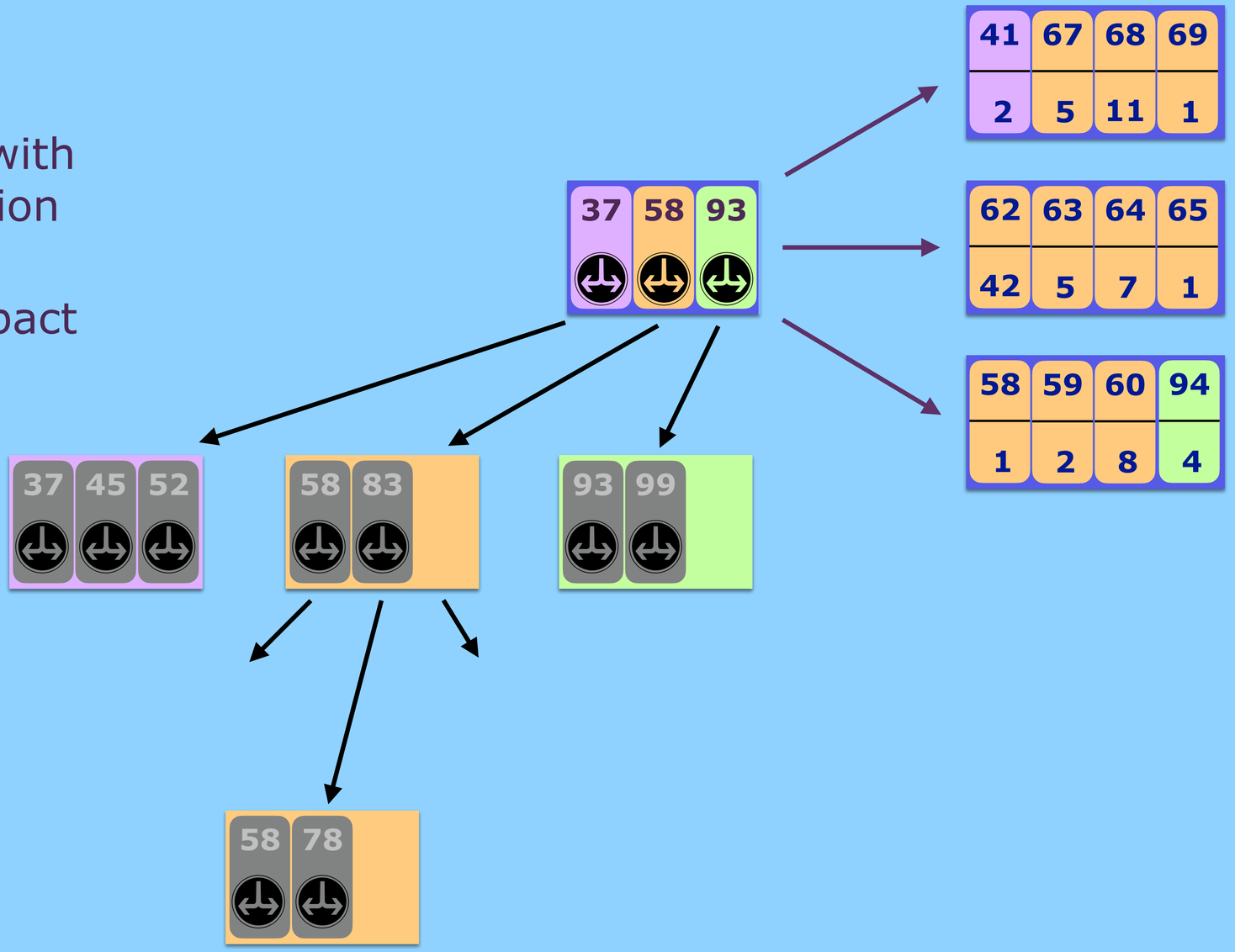
Any data already present
will get merged again

After merging and flushing
another flush will be triggered

Flush-Then-Compact

Want:
Sequential insertions with
lower work amplification

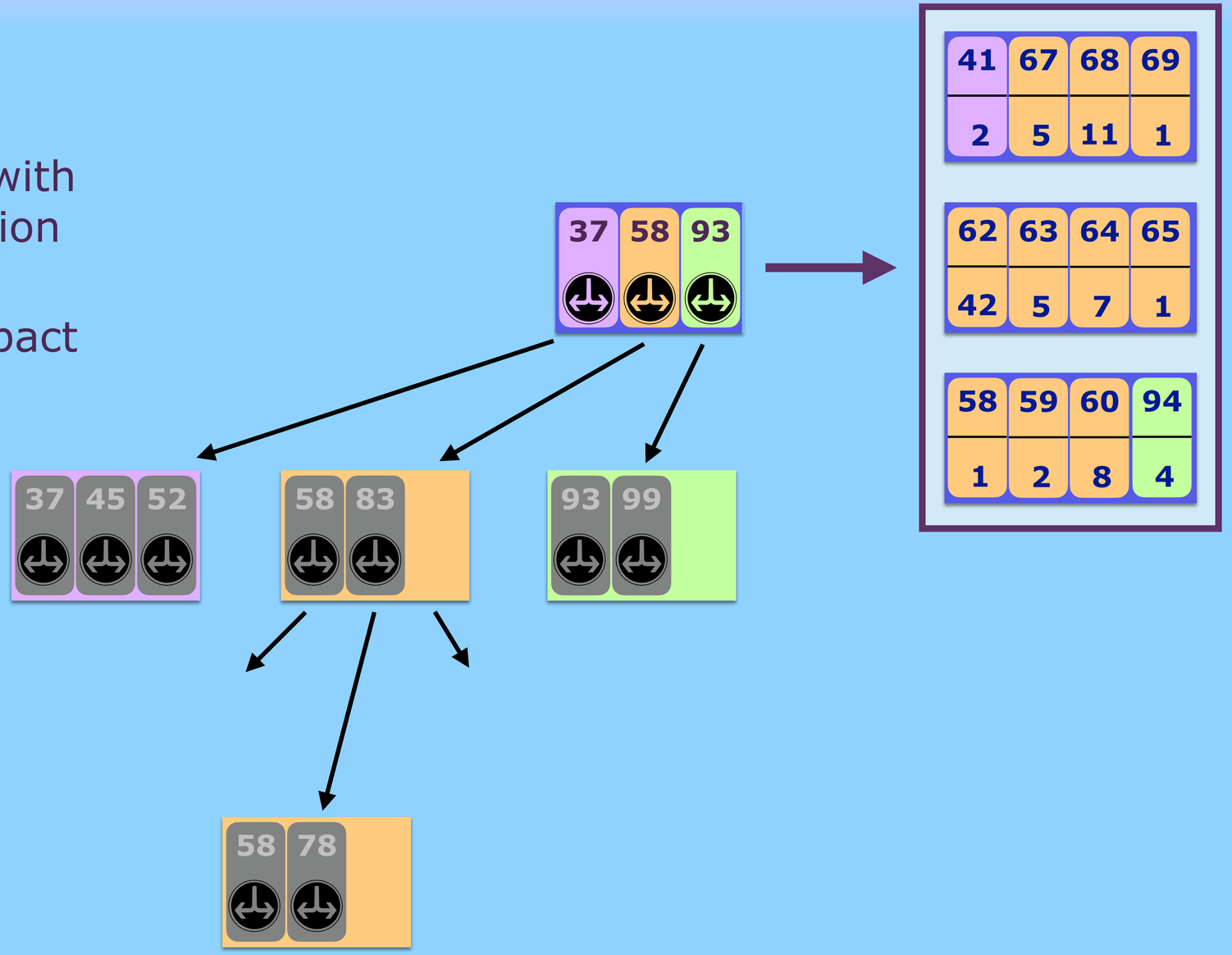
Idea: Flush-then-compact



Flush-Then-Compact

Want:
Sequential insertions with
lower work amplification

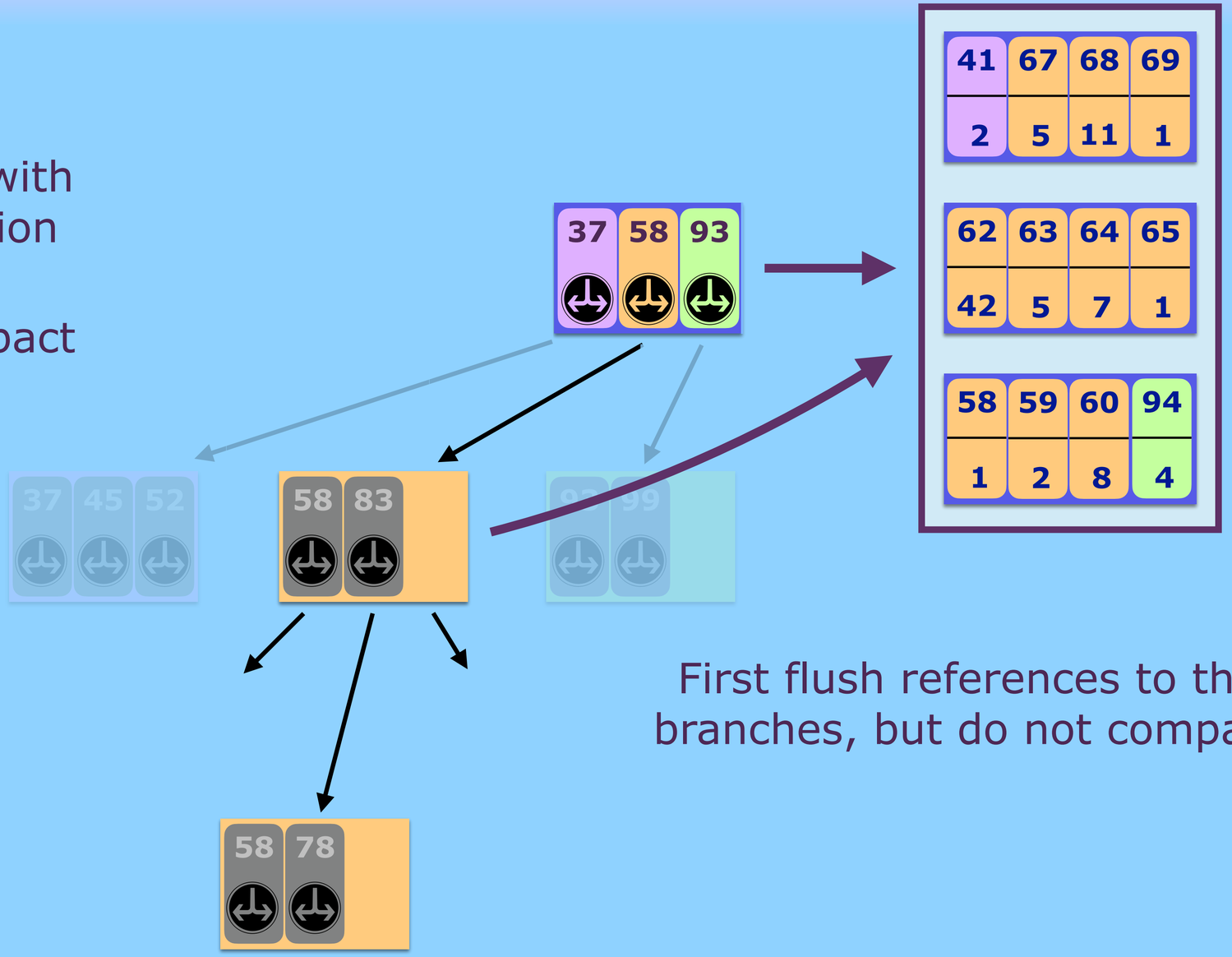
Idea: Flush-then-compact



Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

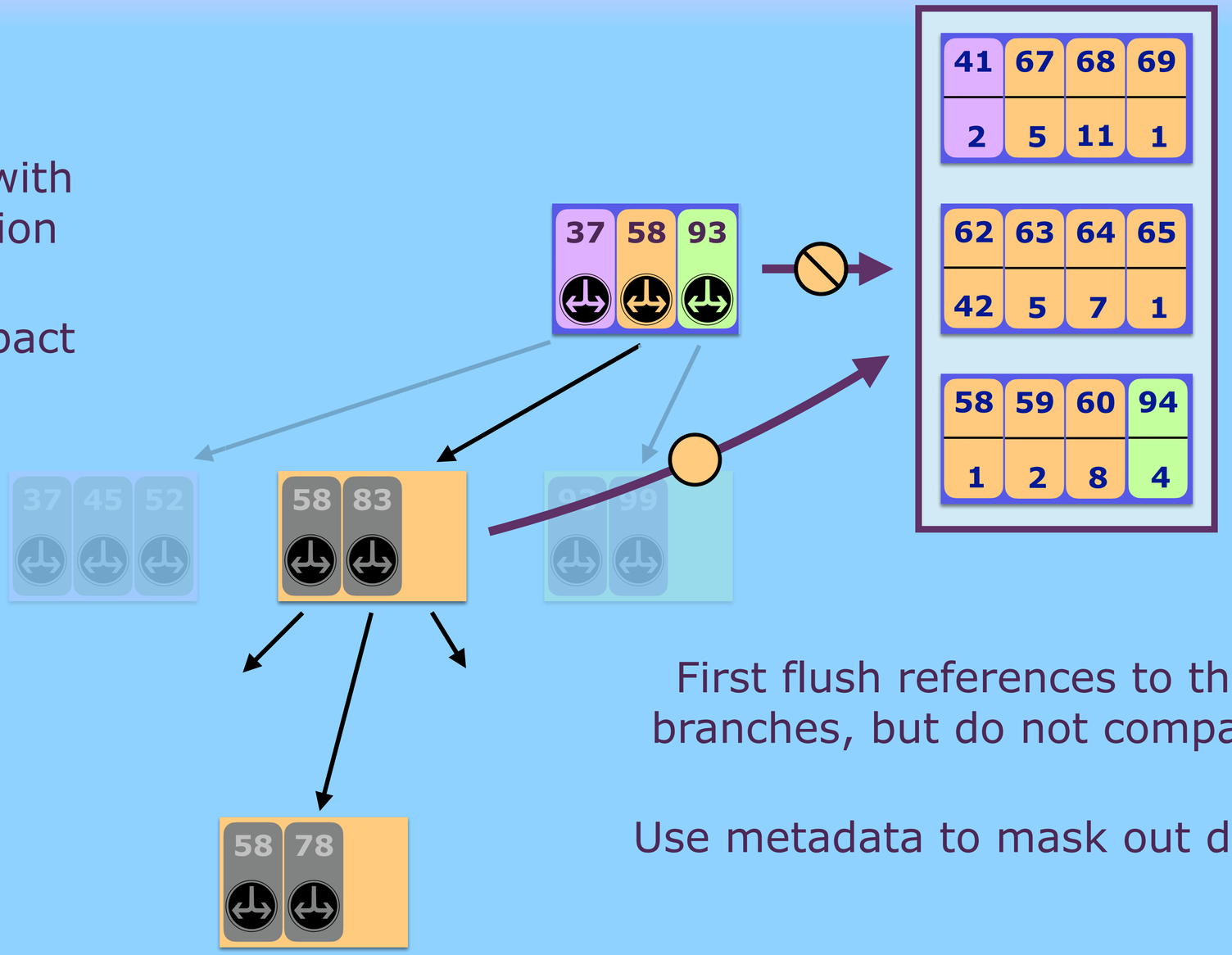


First flush references to the branches, but do not compact

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



First flush references to the branches, but do not compact

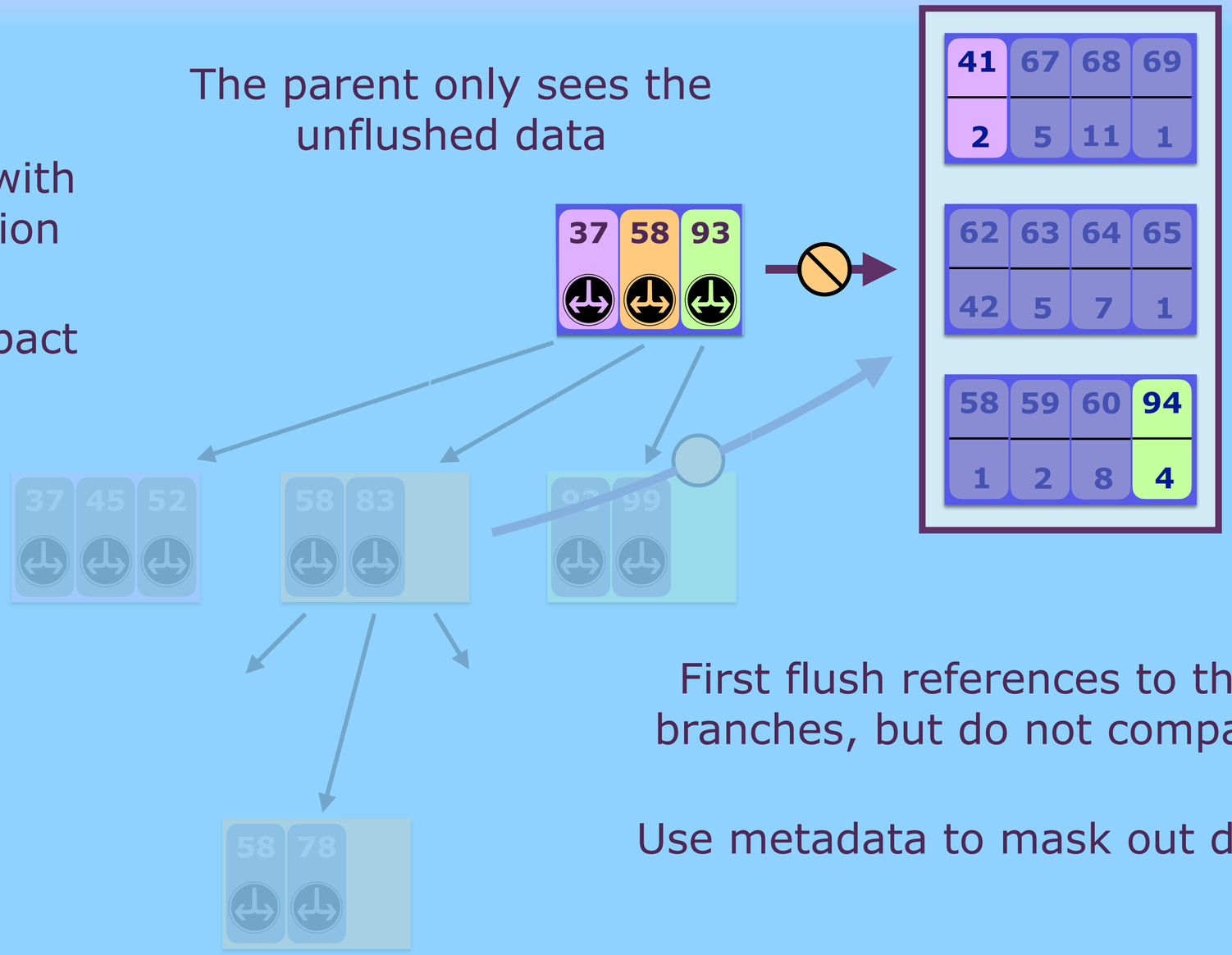
Use metadata to mask out data

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

The parent only sees the unflushed data



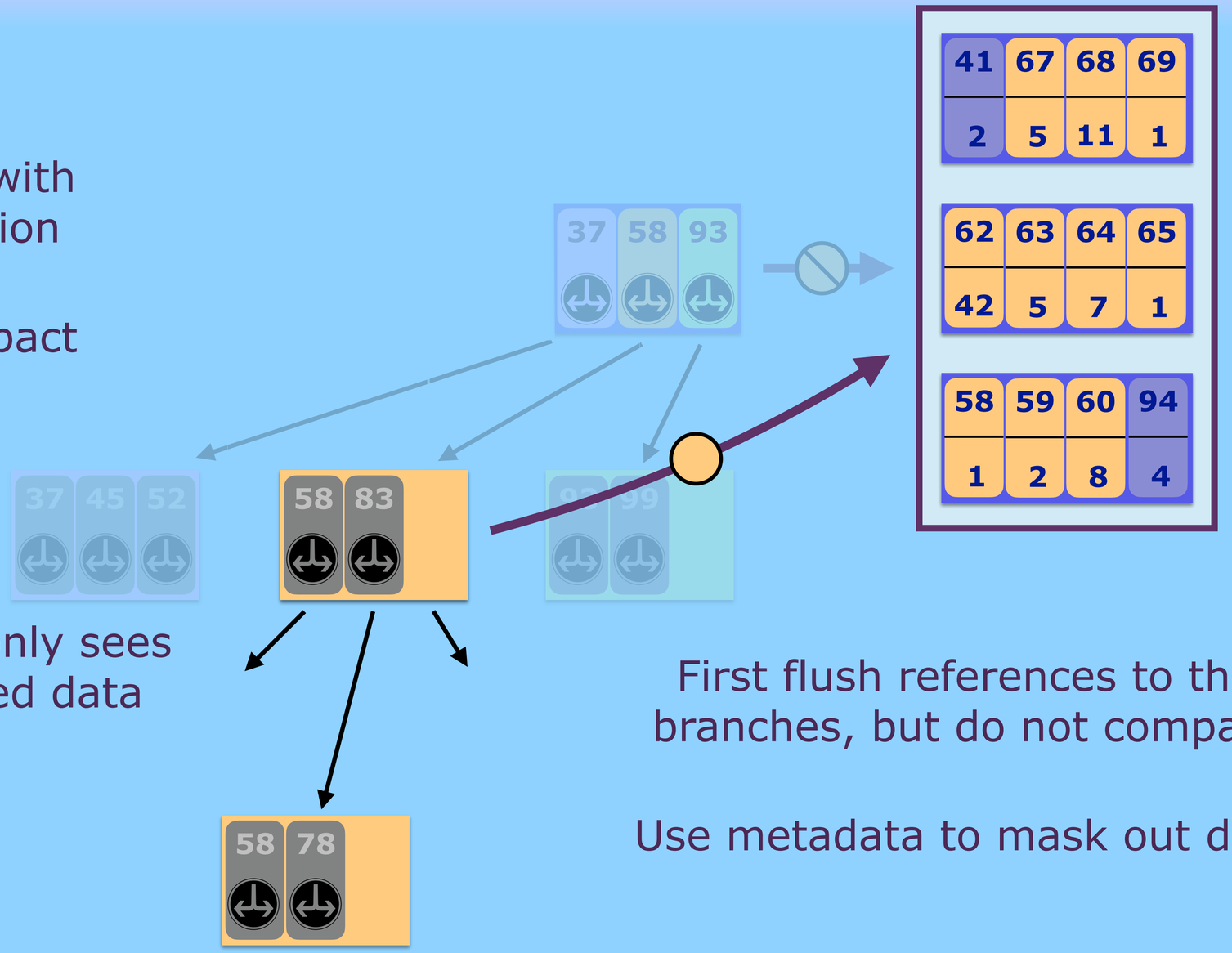
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



The child only sees the flushed data

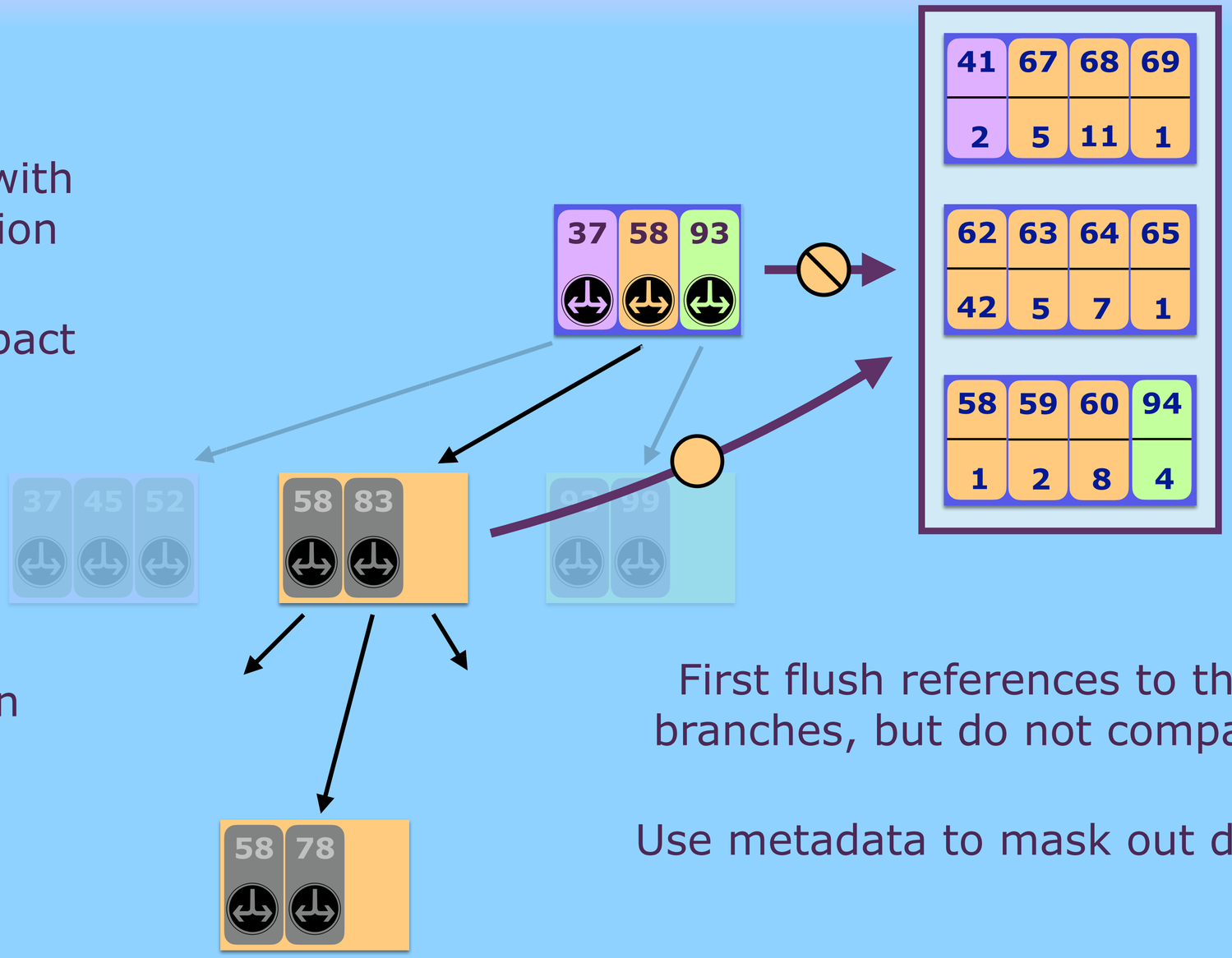
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



First flush references to the branches, but do not compact

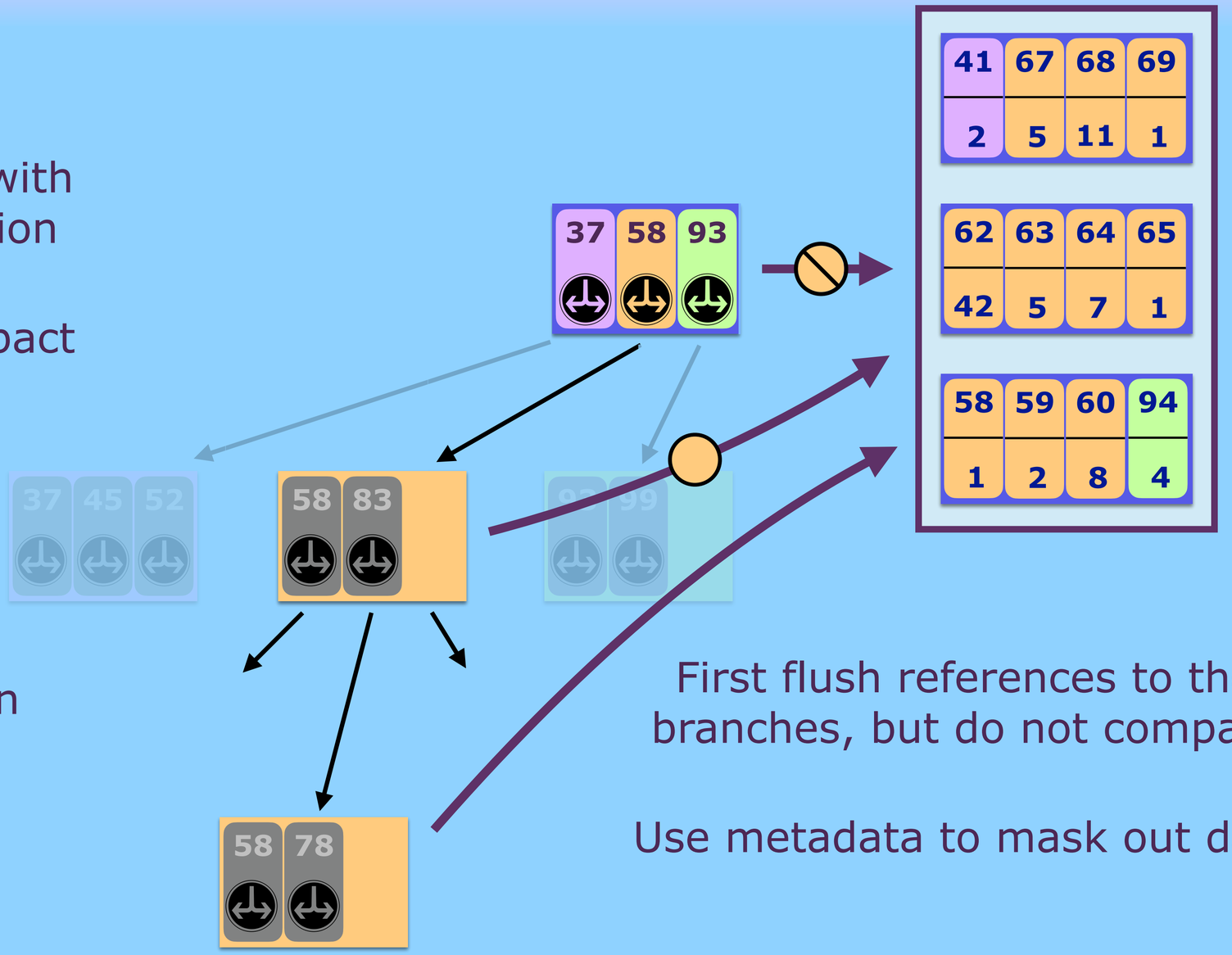
Use metadata to mask out data

Then can flush again

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



Then can flush again

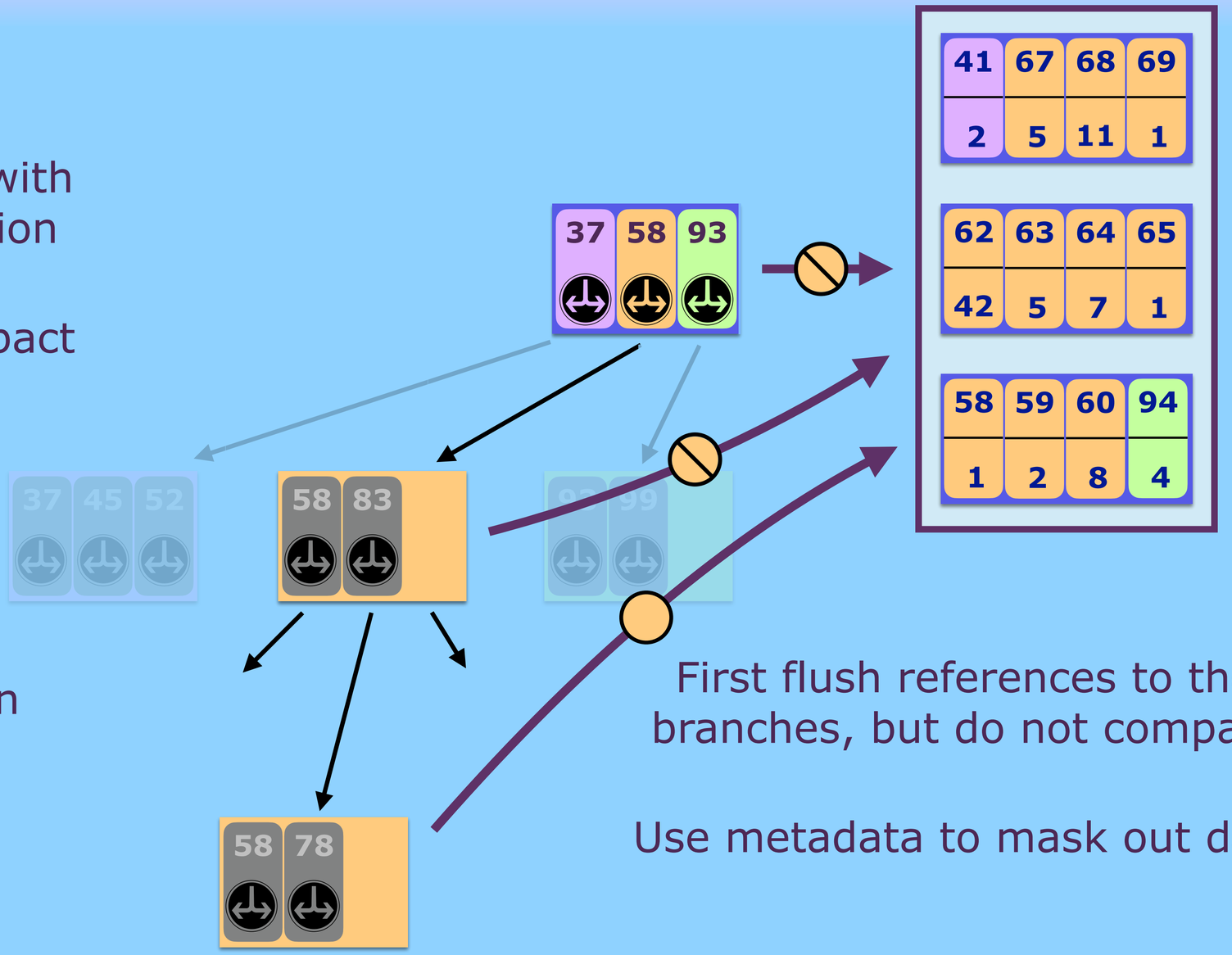
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



Then can flush again

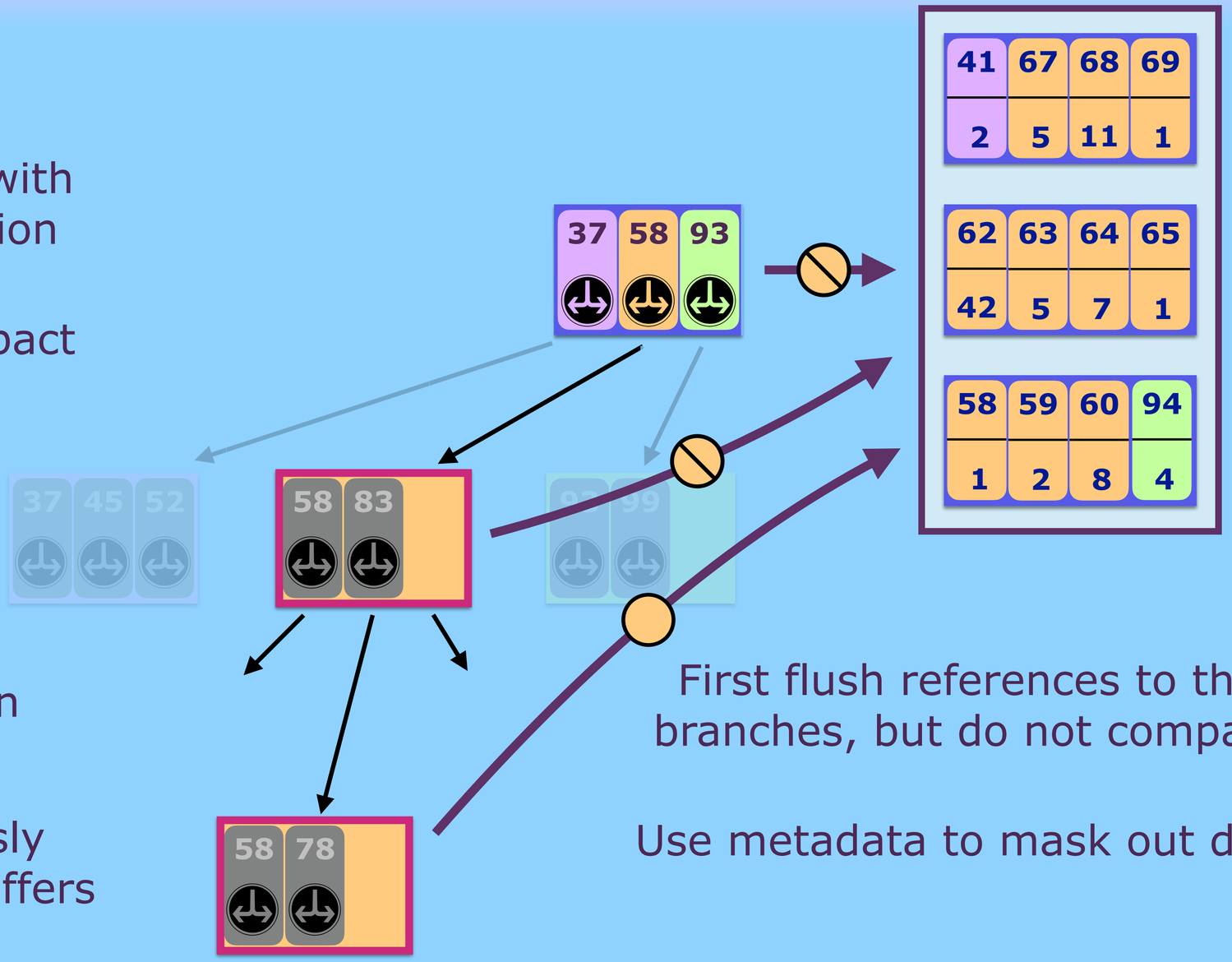
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact



First flush references to the branches, but do not compact

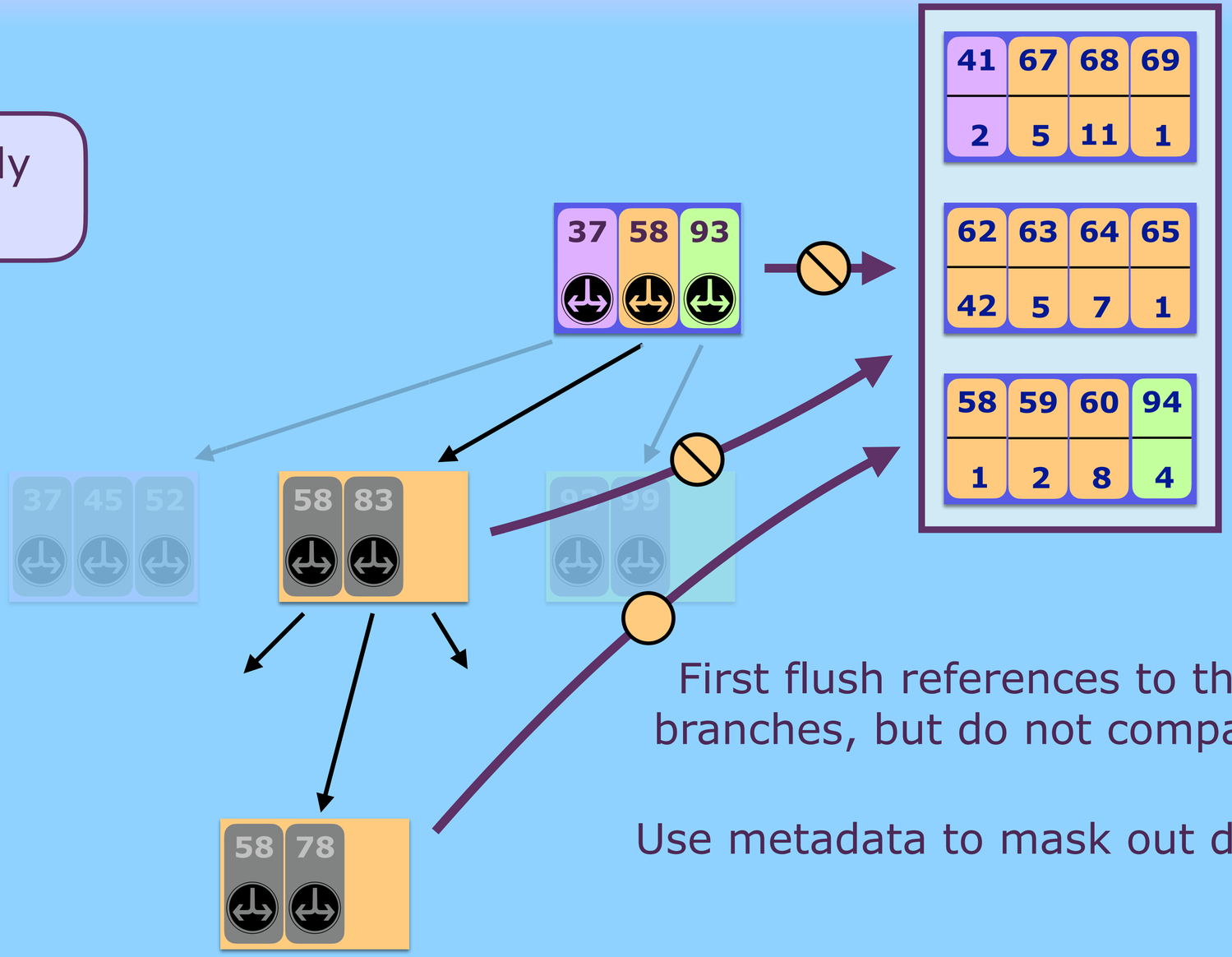
Use metadata to mask out data

Then can flush again

Finally, asynchronously compact the flushed buffers in each node

Flush-Then-Compact

No work on immediately flushed data



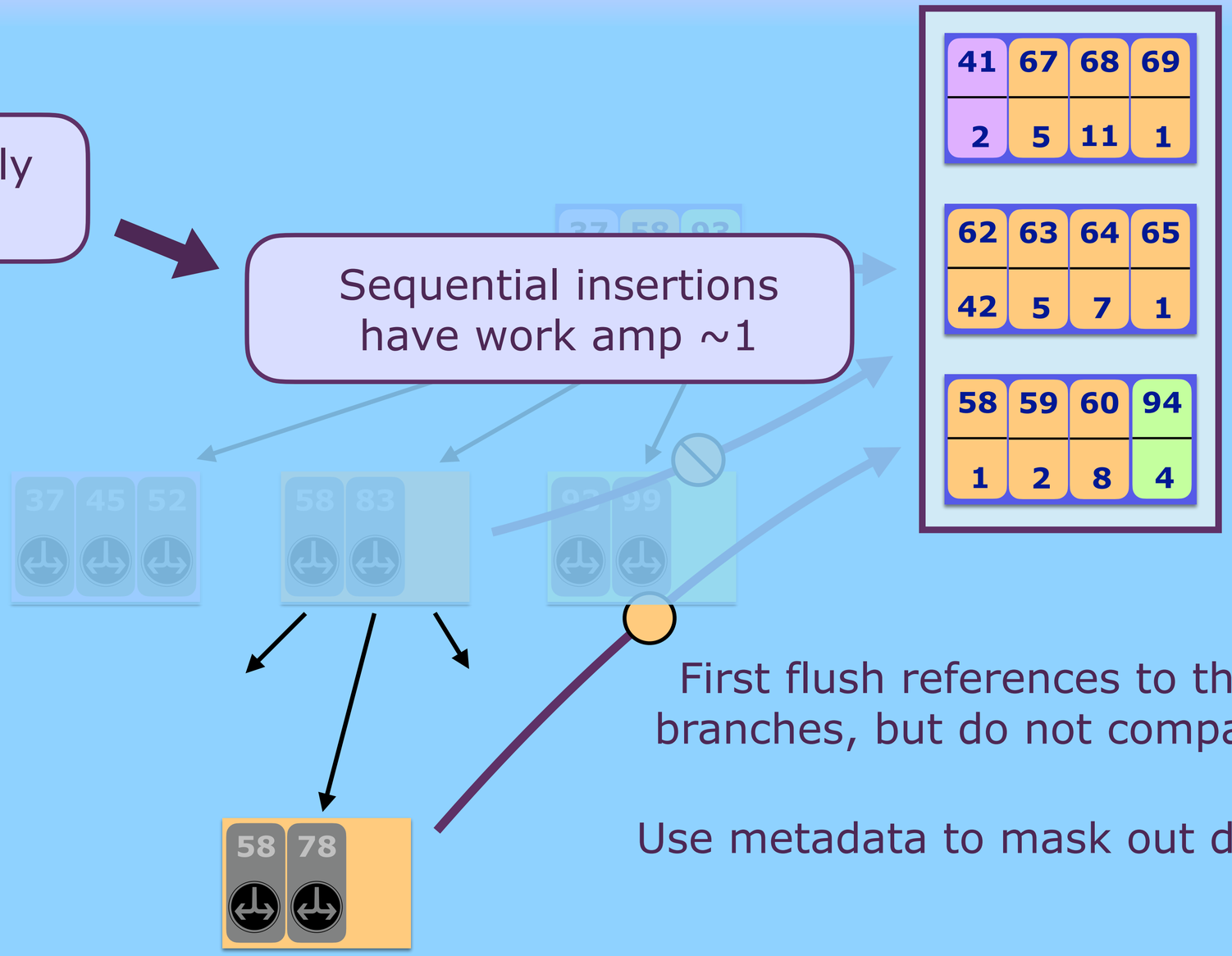
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Sequential insertions have work amp ~1



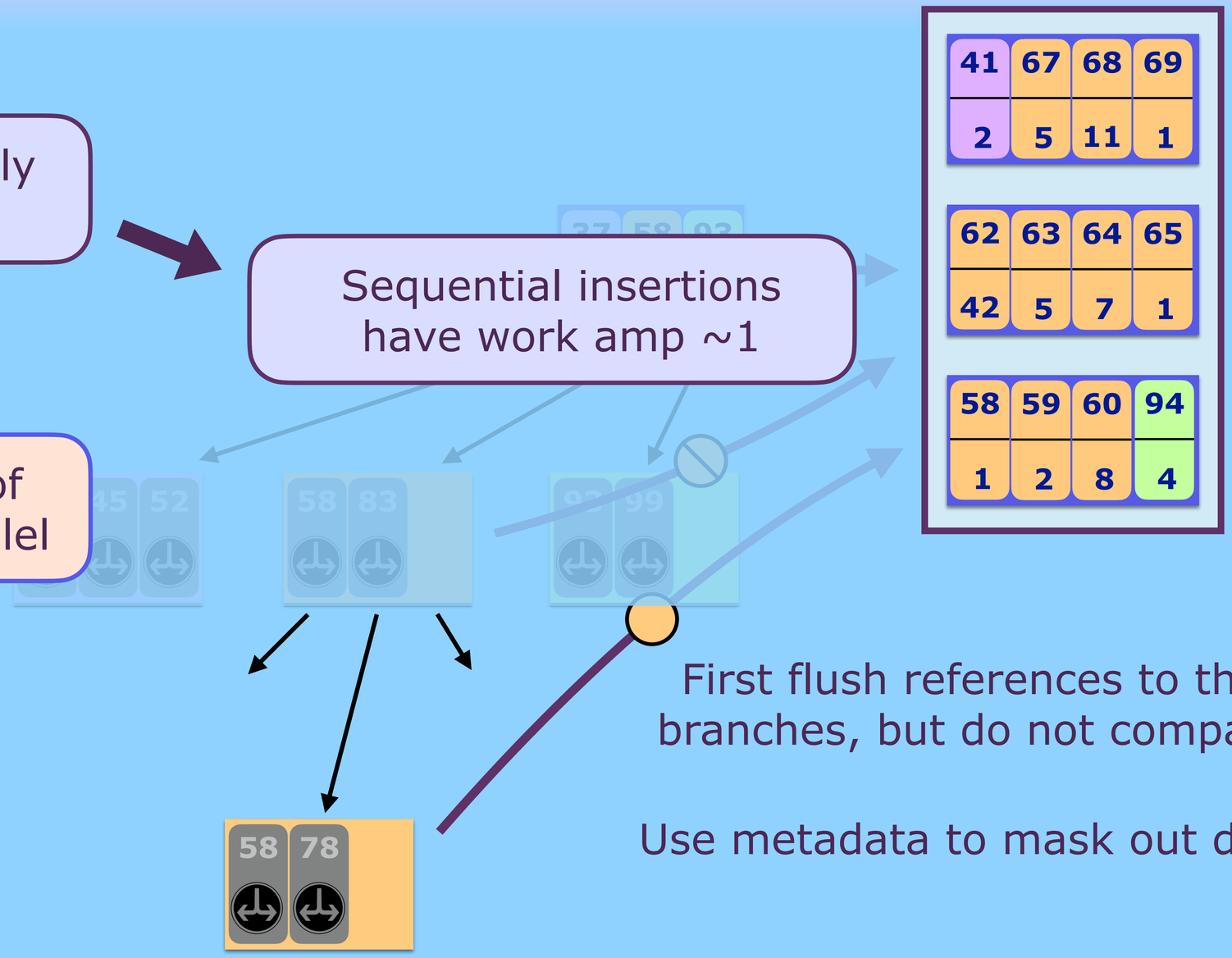
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Break a serial chain of compactions into parallel



First flush references to the branches, but do not compact

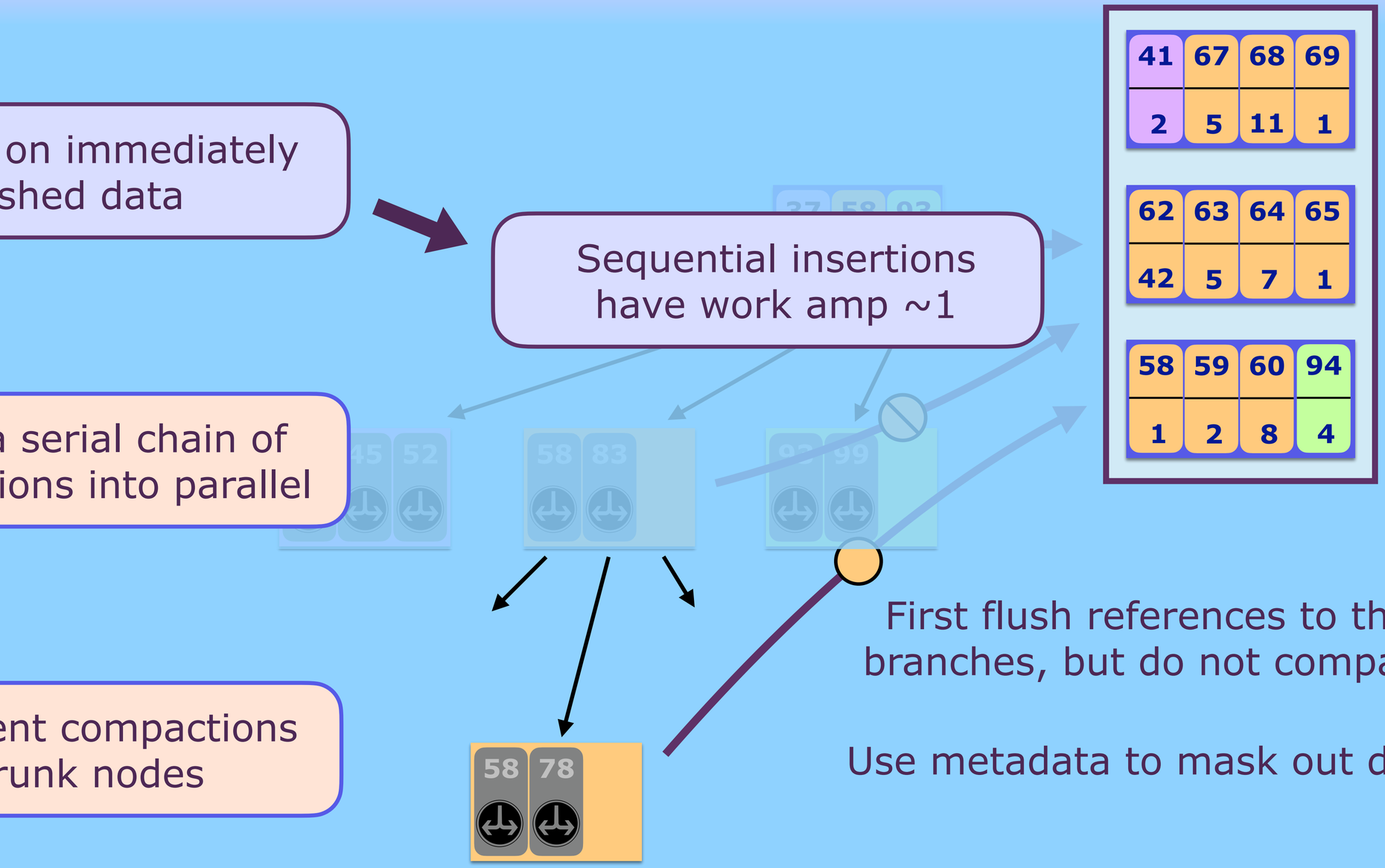
Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Break a serial chain of compactions into parallel

Concurrent compactions in trunk nodes



Flush-Then-Compact

No work on immediately flushed data

Break a serial chain of compactions into parallel

Concurrent compactions in trunk nodes

Sequential insertions have work amp ~1

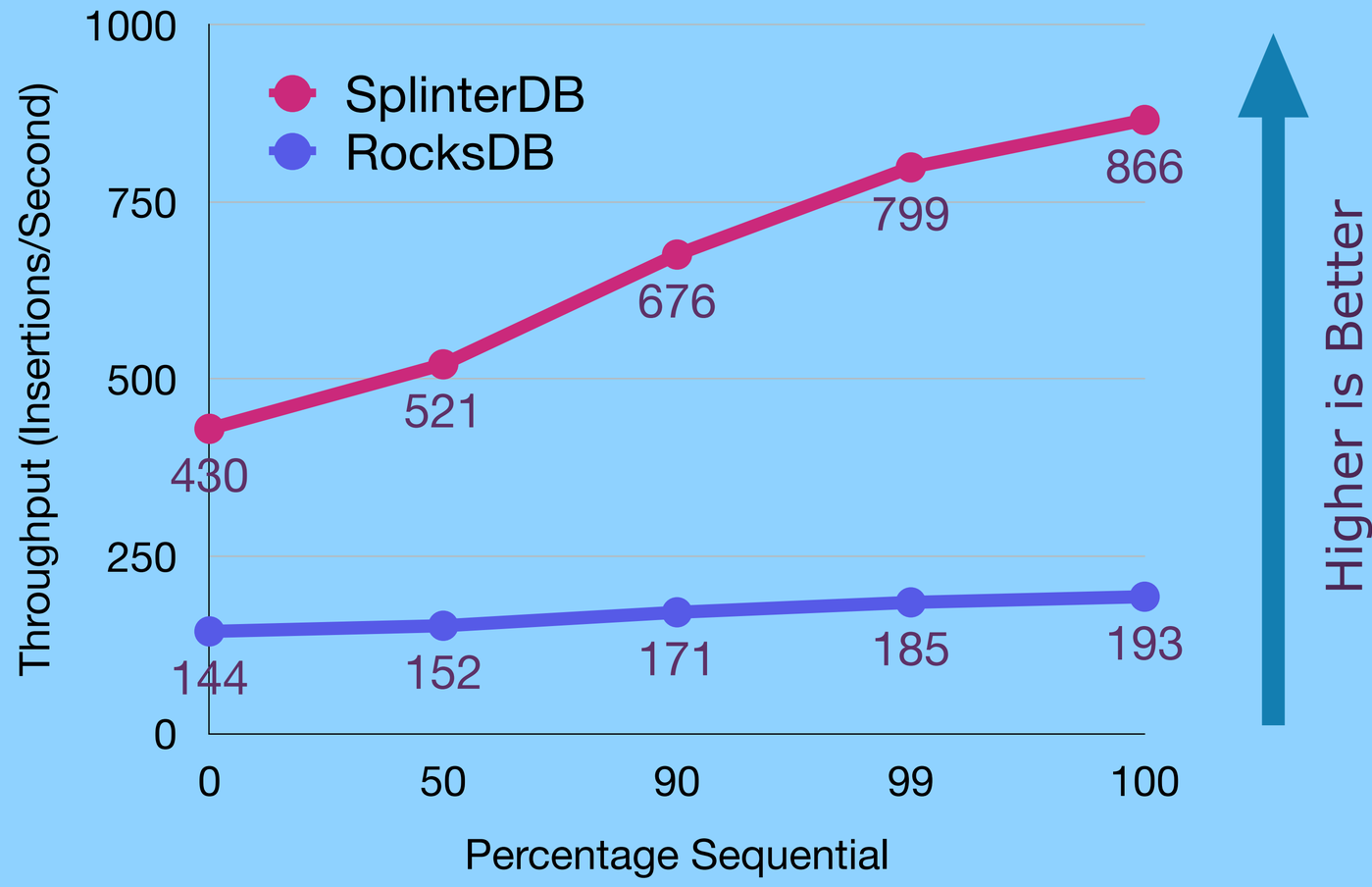
Improve insertion concurrency

sh references to the
, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Run a single-threaded workload with a percentage sequential insertions and the rest random

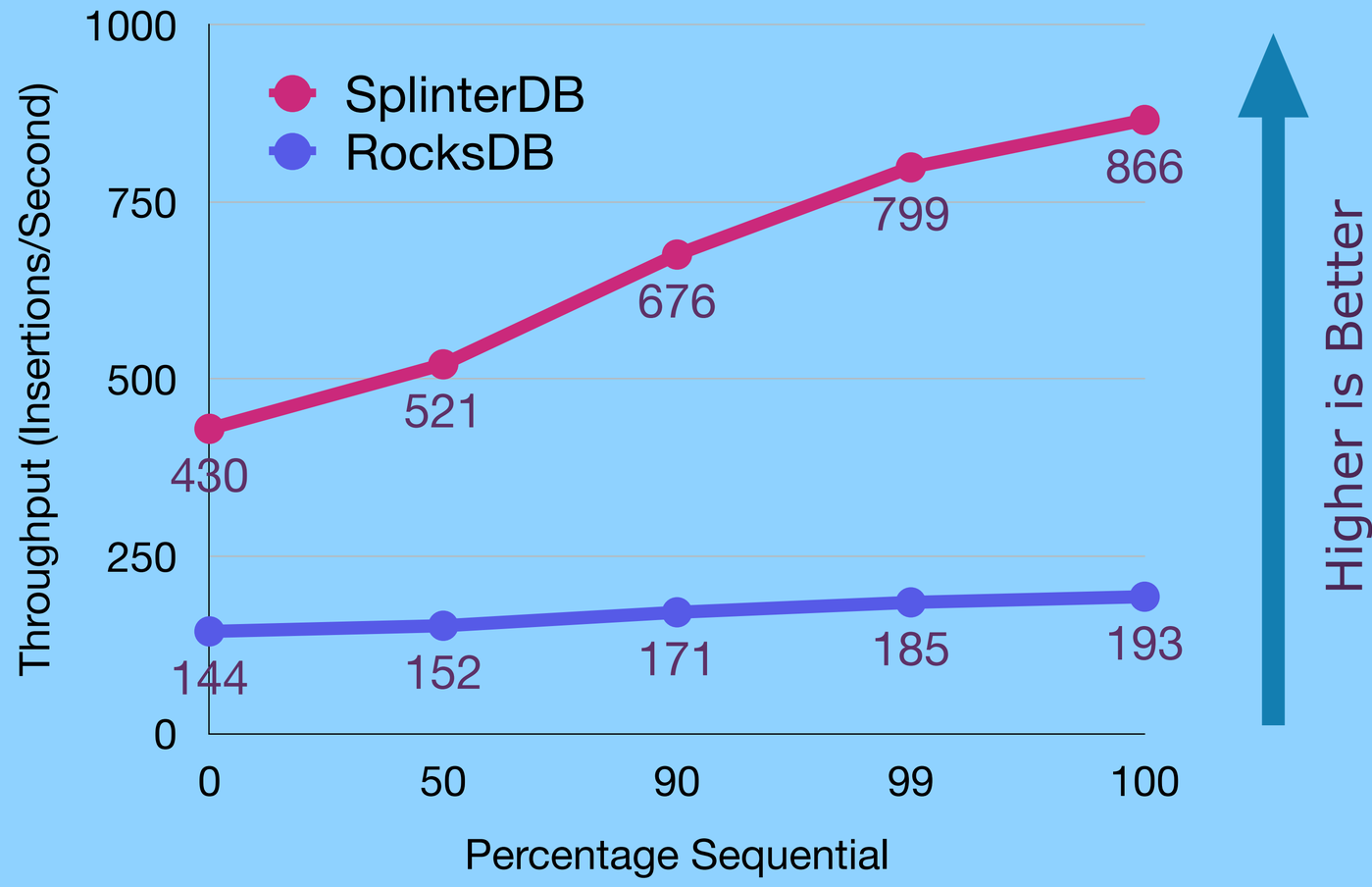


X-axis not to scale

Flush-Then-Compact

Run a single-threaded workload with a percentage sequential insertions and the rest random

Because of flush-then-compact, SplinterDB smoothly increases throughput as the workload gets more sequential



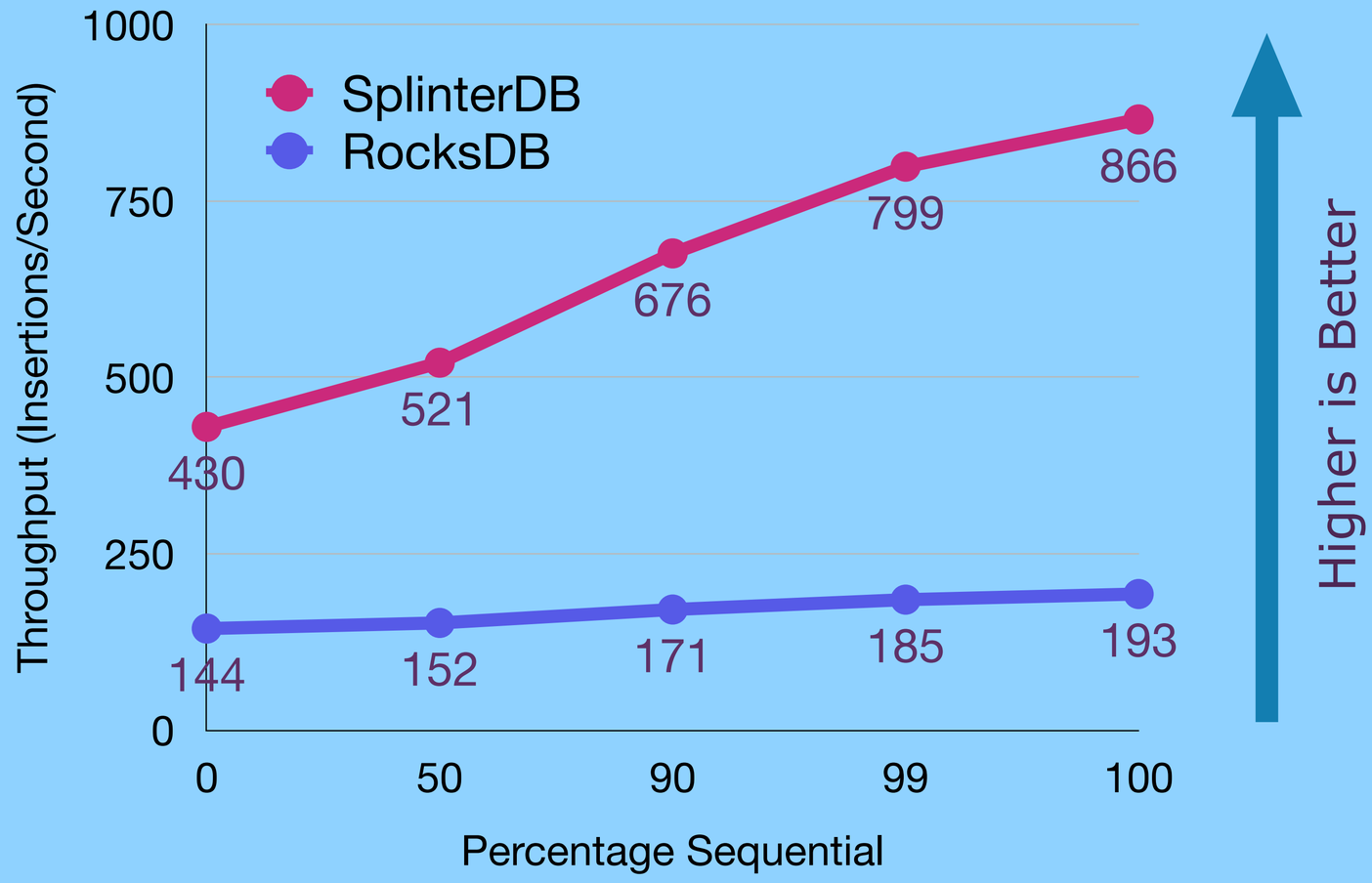
X-axis not to scale

Flush-Then-Compact

Run a single-threaded workload with a percentage sequential insertions and the rest random

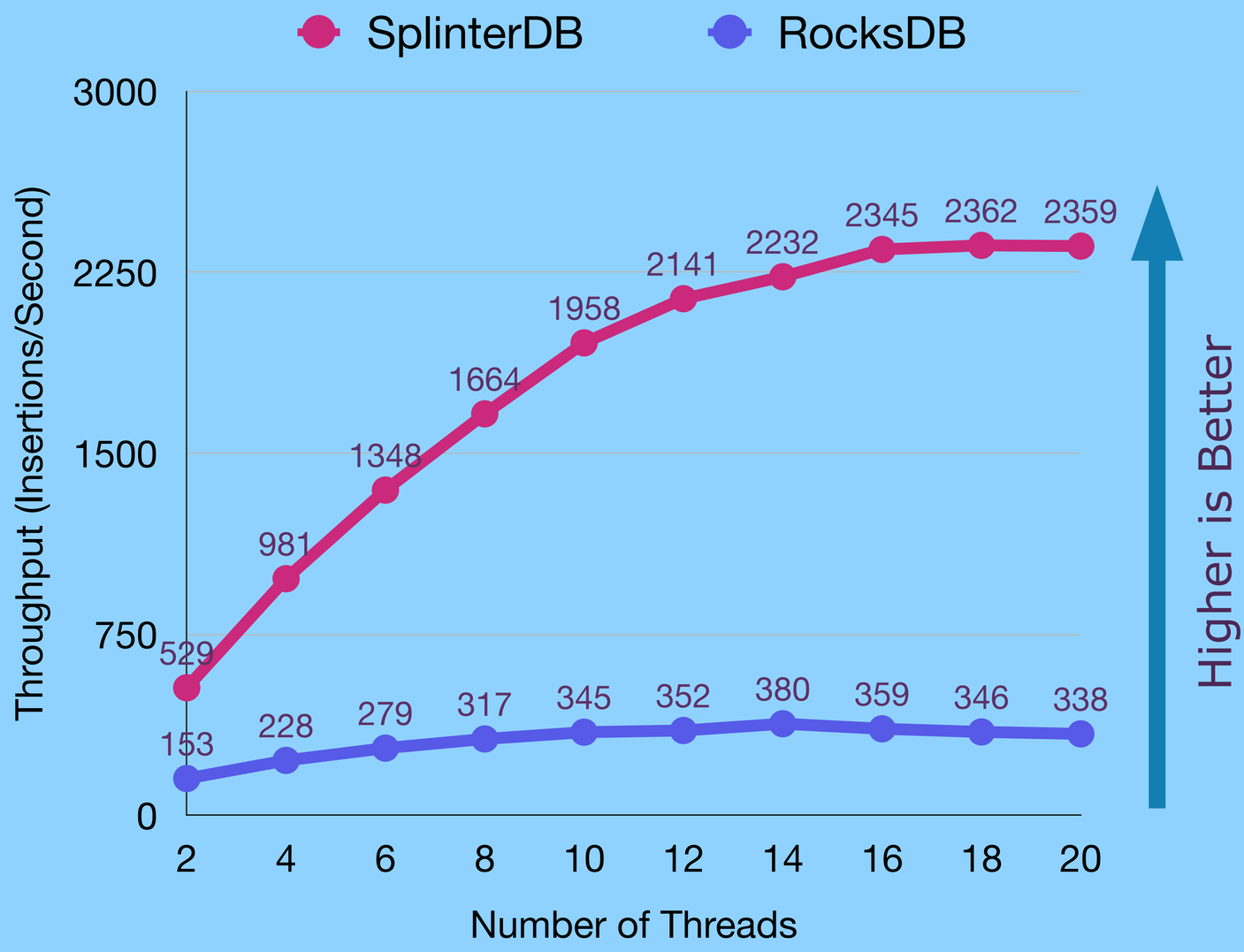
Because of flush-then-compact, SplinterDB smoothly increases throughput as the workload gets more sequential

RocksDB improves, but at a much lower rate



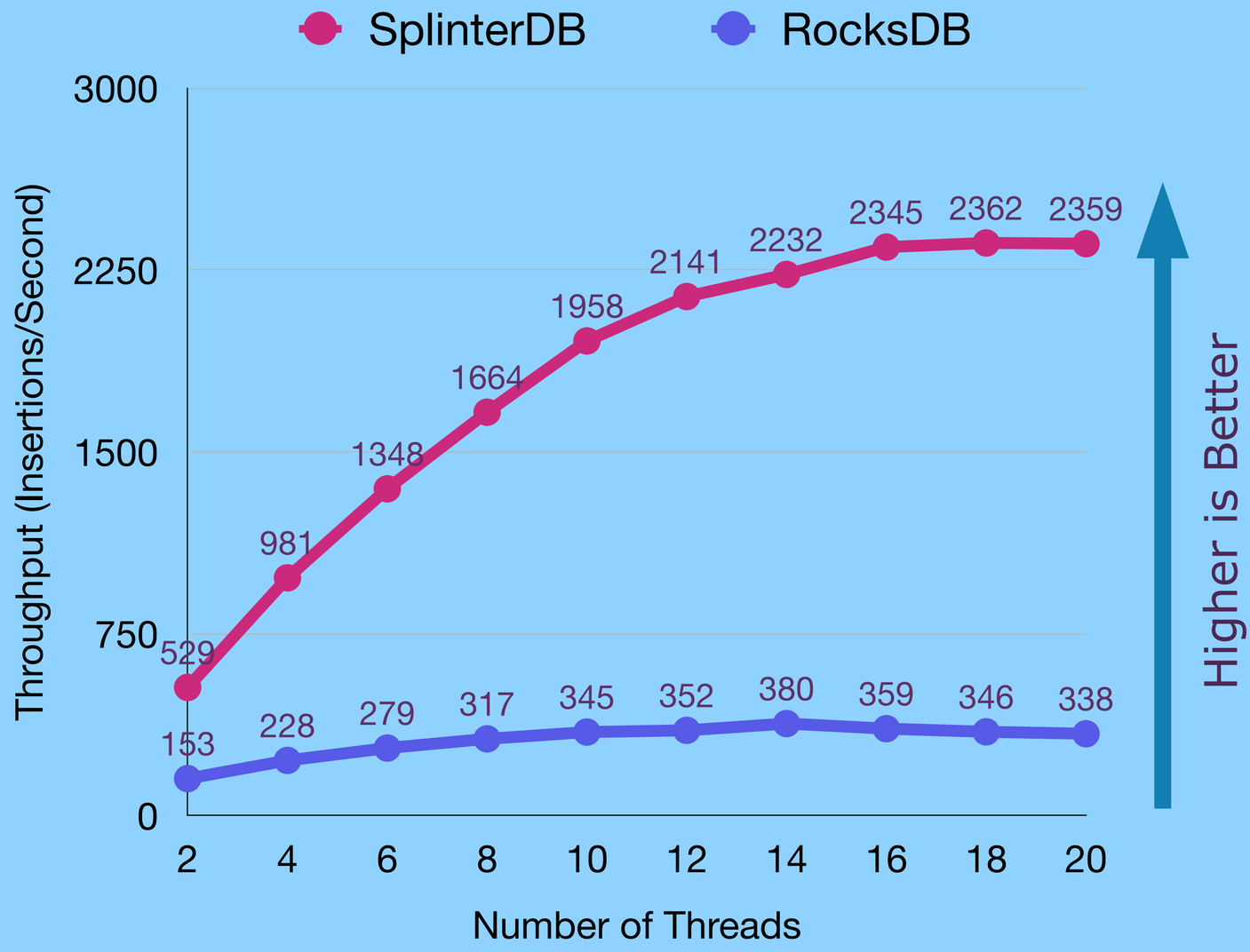
X-axis not to scale

Flush-then-Compact



Insertions in SplinterDB scale well

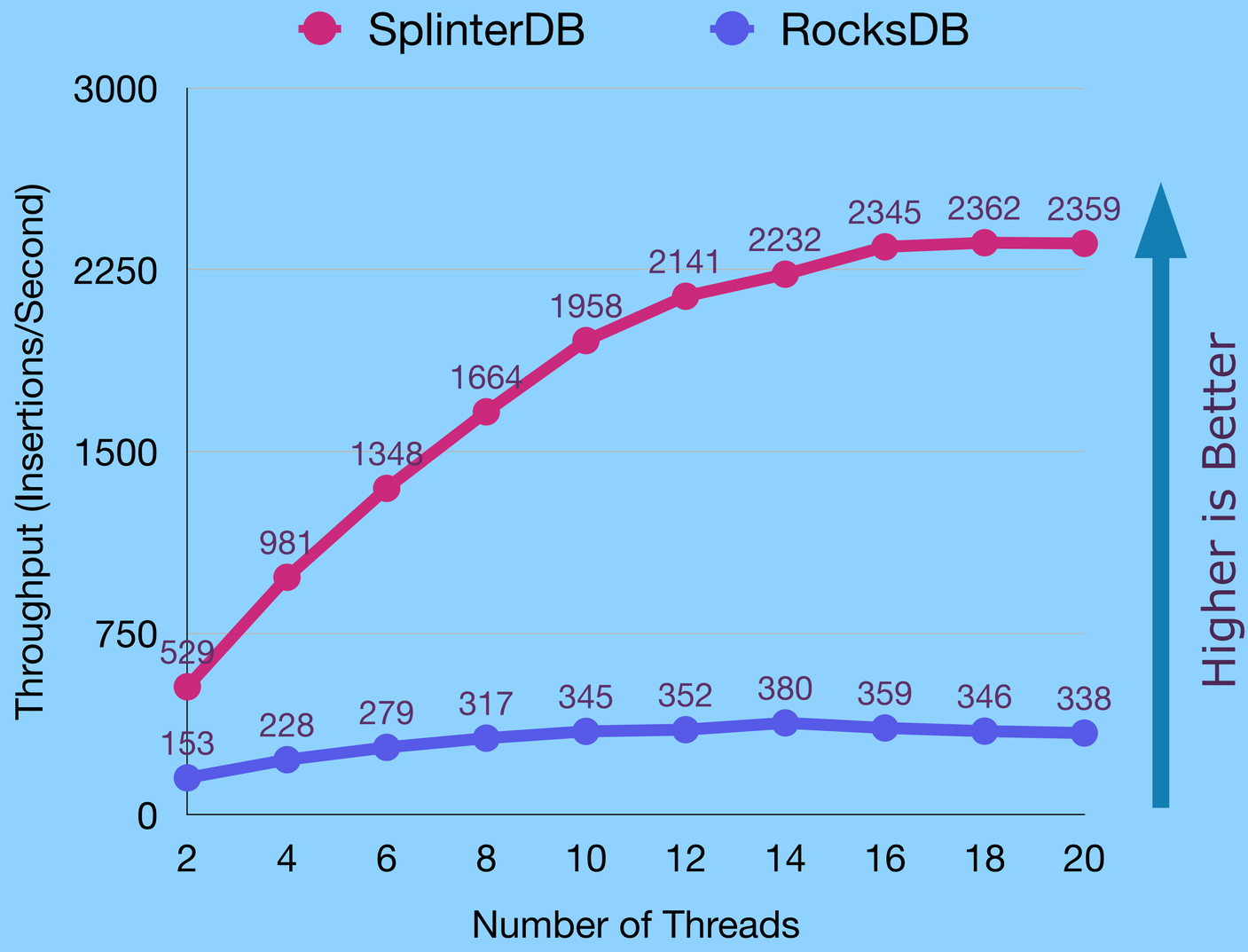
Flush-then-Compact



Insertions in SplinterDB scale well

At 12 threads, SplinterDB has 7x the throughput of 1 thread

Flush-then-Compact



Insertions in SplinterDB scale well

At 12 threads, SplinterDB has 7x the throughput of 1 thread

At 12+ threads, SplinterDB uses 85%+ of the device bandwidth

Conclusion

Conclusion

SplinterDB is a key-value store
which handles these tough cases:

Fast Storage ✓

Small Key-Value Pairs ✓

Small Cache ✓

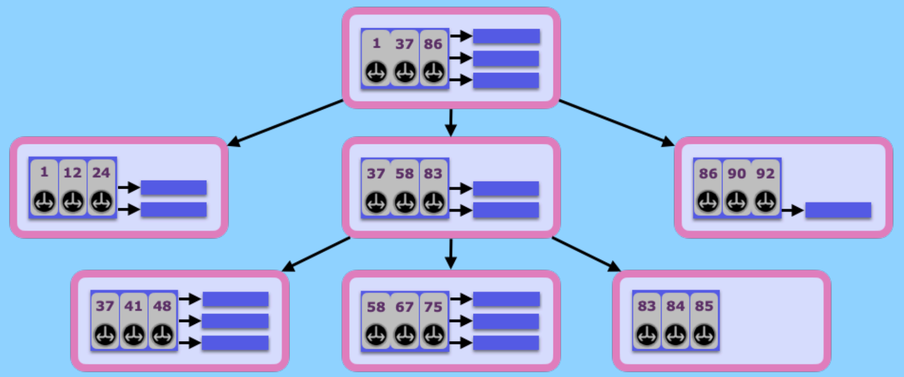
Conclusion

SplinterDB is a key-value store
which handles these tough cases:

Fast Storage ✓

Small Key-Value Pairs ✓

Small Cache ✓



Size-Tiered Bε-Tree

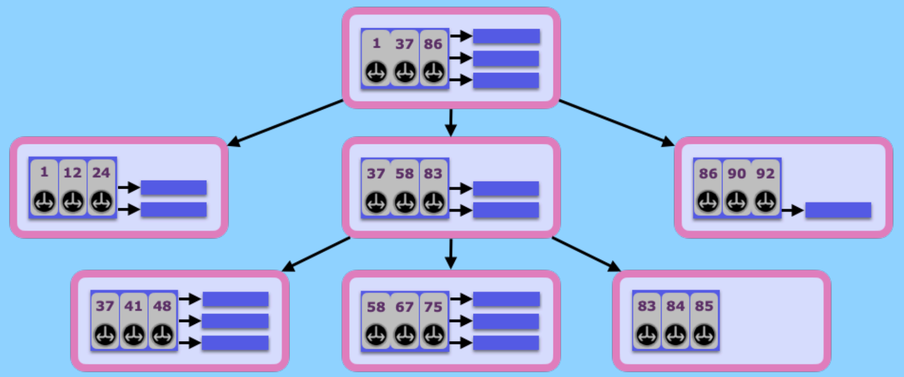
Conclusion

SplinterDB is a key-value store
which handles these tough cases:

Fast Storage ✓

Small Key-Value Pairs ✓

Small Cache ✓



Size-Tiered B^{ε}-Tree

Flush-then-Compact

Thank you!!!

Alex Conway

ajhconway.com

aconway@vmware.com