CS 7270: Advanced Database Systems Fall 2025

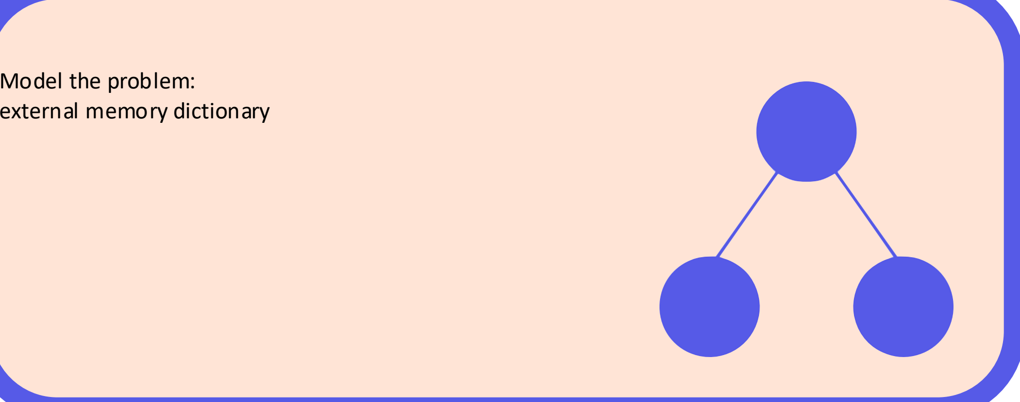
Lecture 11
B^e-tree and SplinterDB

Prashant Pandey

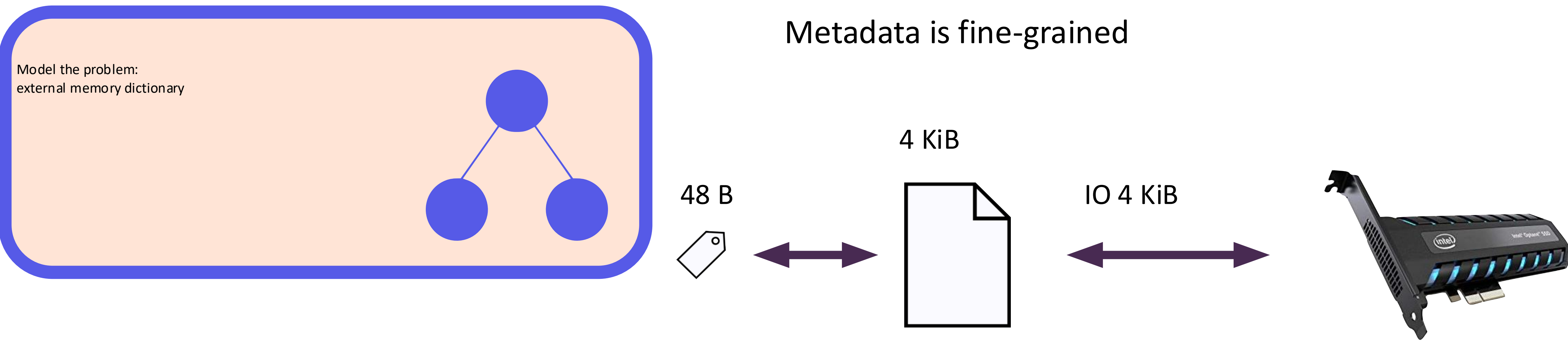
p.pandey@northeastern.edu

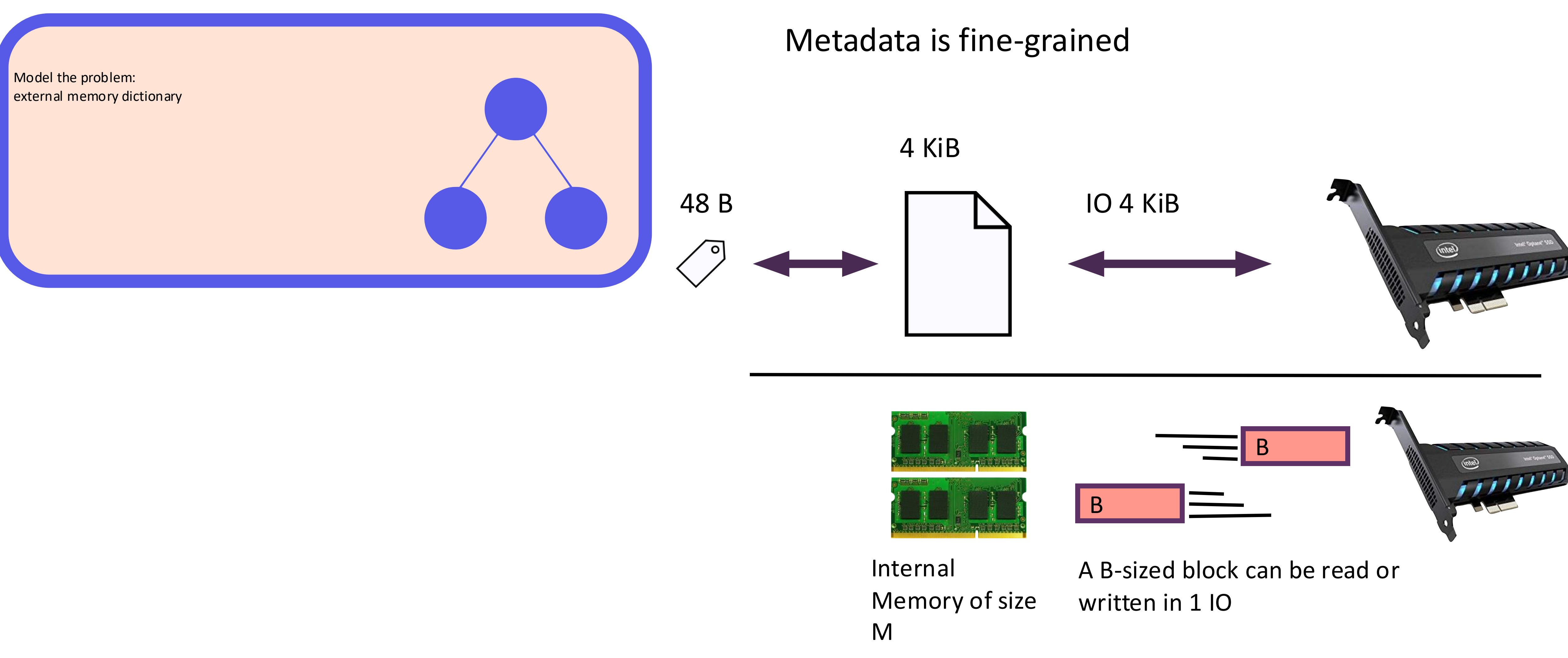
Slides taken from Prof. Alex Conway, Cornell Tech

The Story of SplinterDB

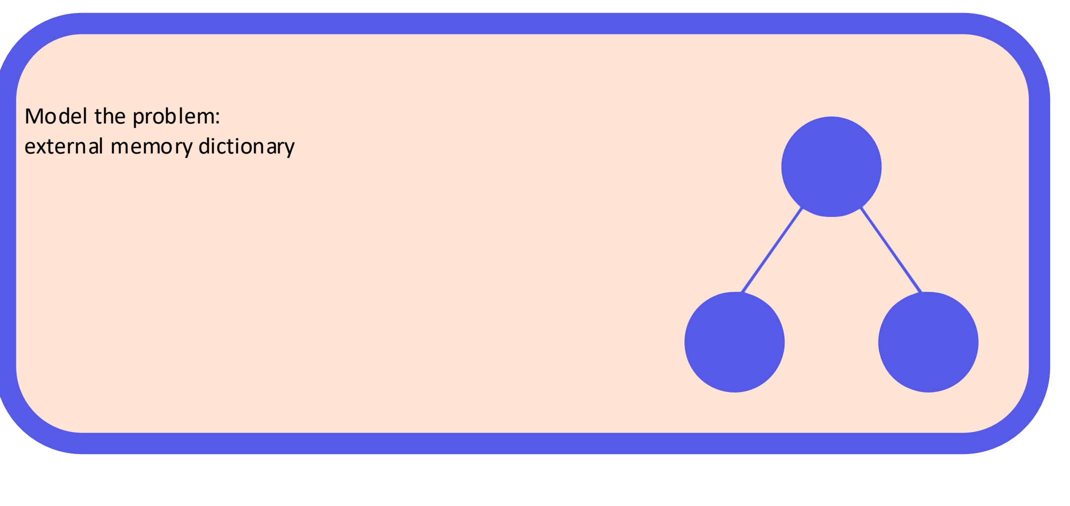


The Story of SplinterDB



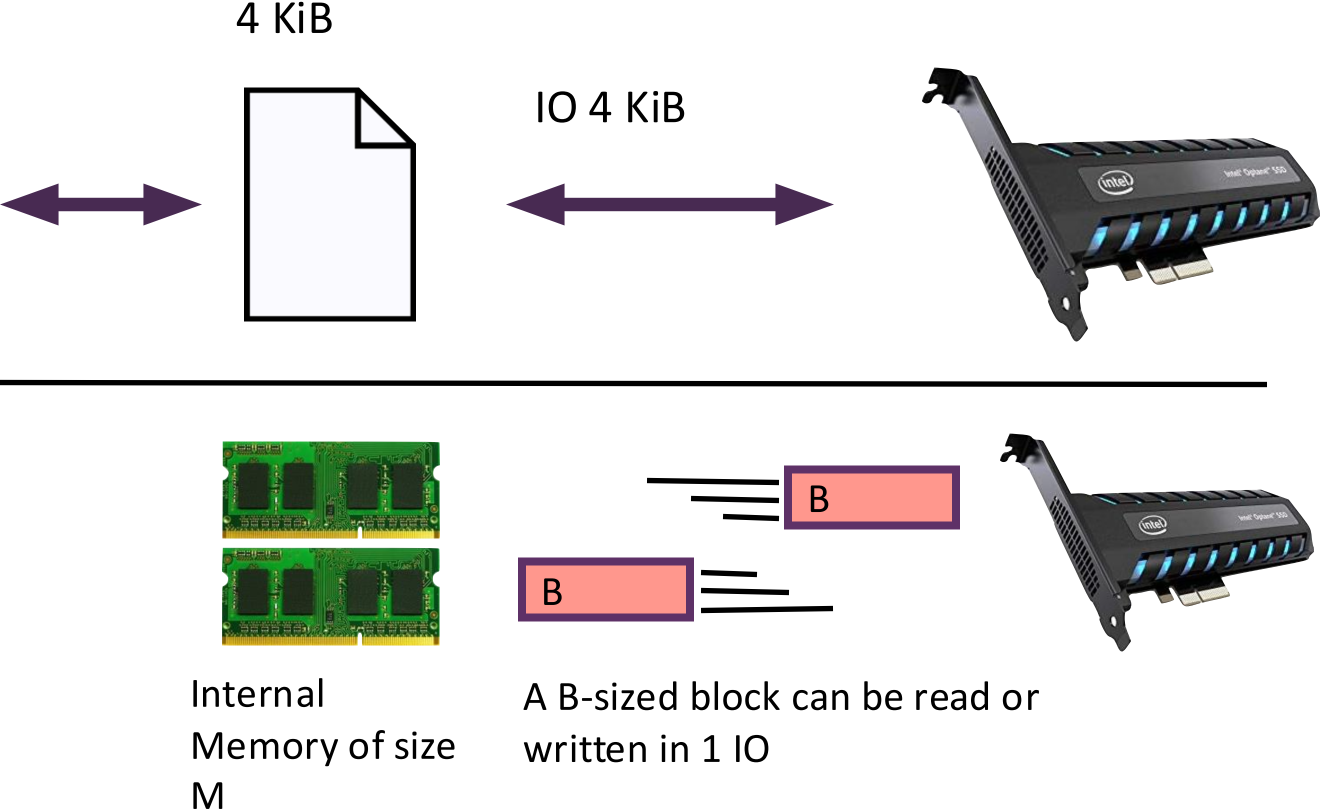

The Story of SplinterDB
Model the problem:
external memory dictionary
Metadata is fine-grained
4 KiB
48 B
IO 4 KiB
B
B
Internal
Memory of size
M
A B-sized block can be read or
written in 1 IO
External Memory Model

The Story of SplinterDB



Metadata is fine-grained

48 B

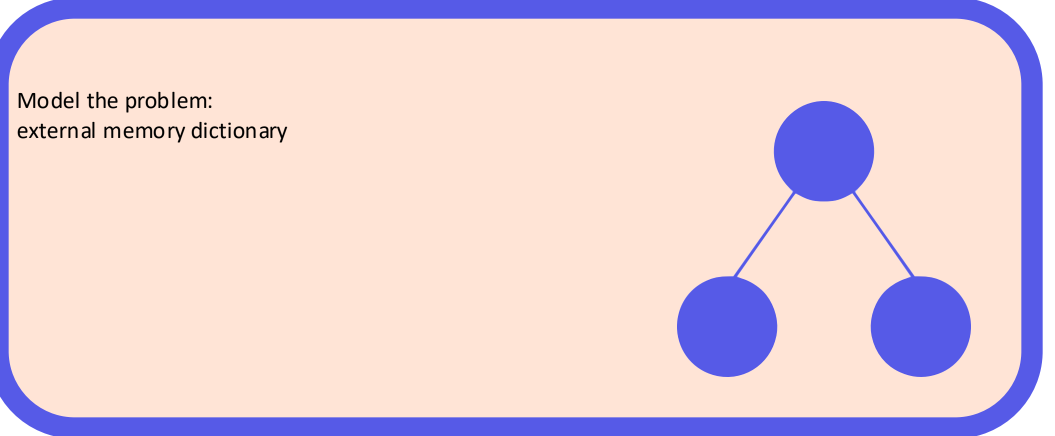


Here B is the number of items in an IO:
B = 4 KiB / 48 B

If the items were larger, the model wouldn't be as good

External Memory Model

The Story of SplinterDB



Two Flavors of
External-Memory Dictionary

Different lower bounds
(performance limits)

Comparison-Based Dictionaries

Comparison External Memory Model

```
                          <
user024299                =         user082587
                          >
```

Comparison-Based Dictionaries

Comparison External Memory Model

Comparison-Based Dictionaries

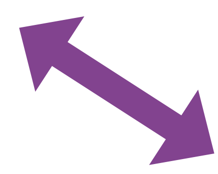
Comparison External Memory Model

Lower bounds for external memory dictionaries,
Brodal, G., Fagerberg, R. SODA '03

Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$



Lookups in

$$\Omega(\log_\lambda N)$$

where λ is a tuning parameter

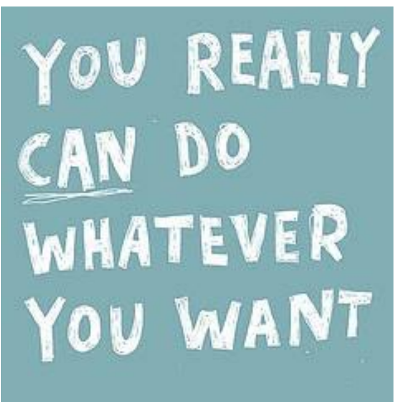
General Dictionaries

External Memory Model

General Dictionaries

External Memory Model

user024299



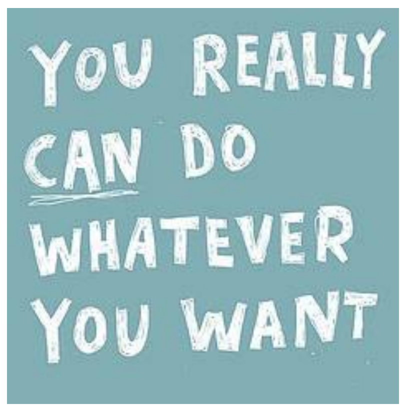
General Dictionaries

External Memory Model

`user024299`

Hashing

`XXH(user024299)`



General Dictionaries

External Memory Model

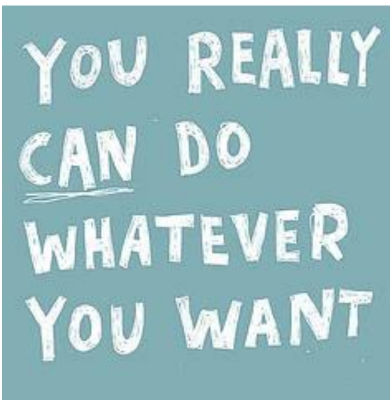
`user024299`

Hashing

`XXH(user024299)`

Filters

`qf_insert(user024299)`



General Dictionaries

External Memory Model

```
user024299
```

Hashing

```
XXH(user024299)
```

Filters

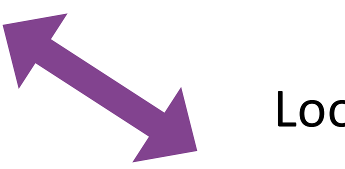
```
qf_insert(user024299)
```

Using hashing to solve the dictionary problem (in external memory), Iacono, J., Pătrașcu, M. SODA '12

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\cancel{\log_\lambda N}\right)$$



Lookups in

$$\Omega(\log_\lambda N)$$

where λ is a tuning parameter

Lower Bounds

Brodal-Fagerberg Lower Bound

Insertions in

$O\left(\frac{\lambda}{B}\log_{\lambda}N\right)$

↔

Lookups in

$\Omega(\log_{\lambda}N)$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$O\left(\frac{\lambda}{B}\right)$

↔

Lookups in

$\Omega(\log_{\lambda}N)$

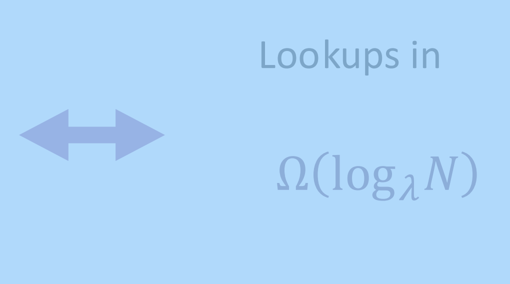
General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$

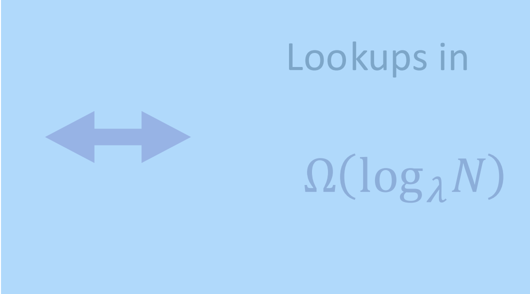


Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

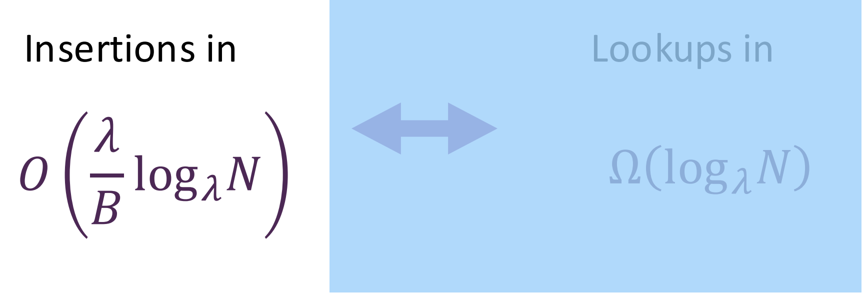
$$O\left(\frac{\lambda}{B}\right)$$



General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound



Leveled LSMs

1 component per level

L0

T times larger

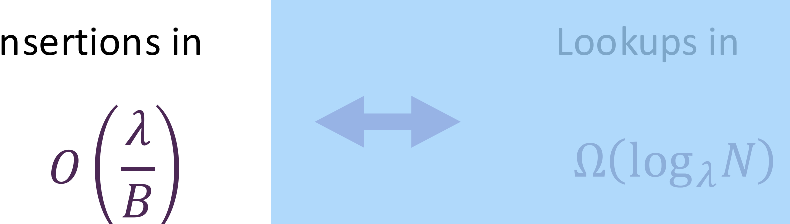
L1

L2

(a) Leveling Merge Policy

Comparison External Memory Model

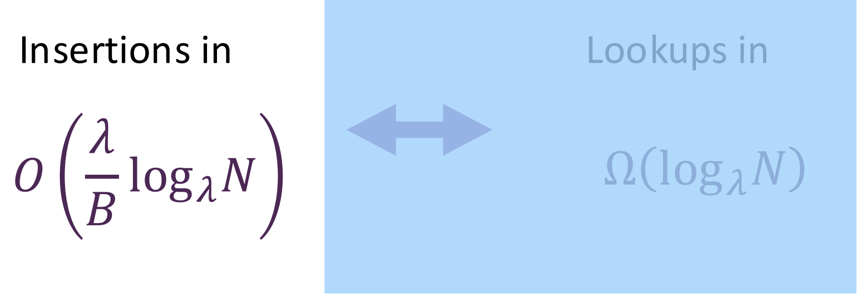
Iacono-Pătrașcu Lower Bound



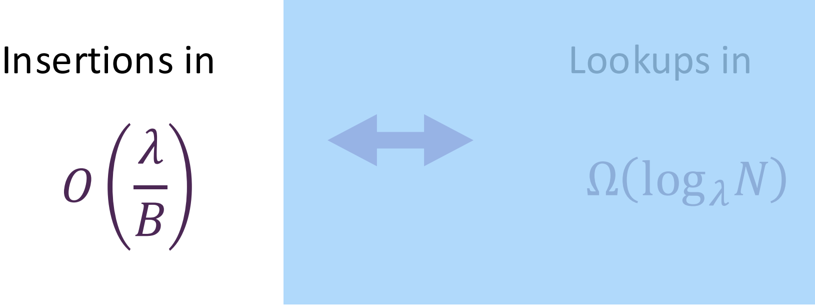
General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound



Iacono-Pătrașcu Lower Bound



Leveled LSMs

1 component per level

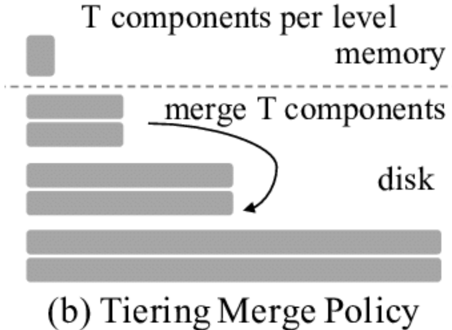
L0 T times larger

L1

L2

(a) Leveling Merge Policy

Size-tiered LSMs



(b) Tiering Merge Policy

Comparison External Memory Model

General External Memory Model

Lower Bounds

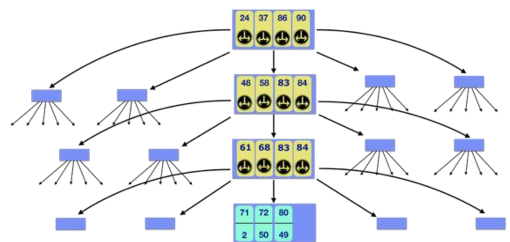
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$



B-Trees

$(\lambda = B)$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$

General External Memory Model

Lower Bounds

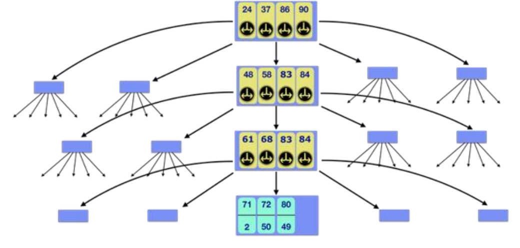
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$

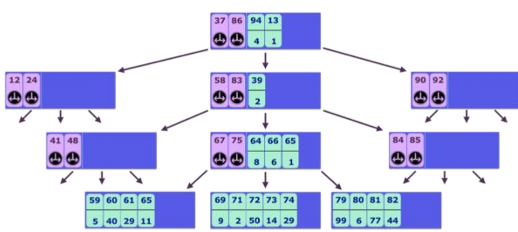
Lookups in

$$\Omega(\log_\lambda N)$$



B-Trees

$(\lambda = B)$



B^ε-Trees

$(\lambda = B^\varepsilon)$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$

General External Memory Model

Lower Bounds

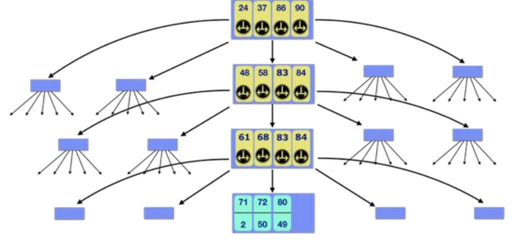
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_{\lambda}N\right)$$

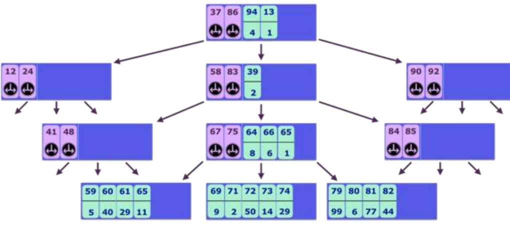
Lookups in

$$\Omega(\log_{\lambda}N)$$



B-Trees

$$(\lambda = B)$$



B^{ε}-Trees

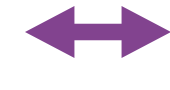
$$(\lambda = B^{\varepsilon})$$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$



Lookups in

$$\Omega(\log_{\lambda}N)$$

Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

General External Memory Model

Lower Bounds

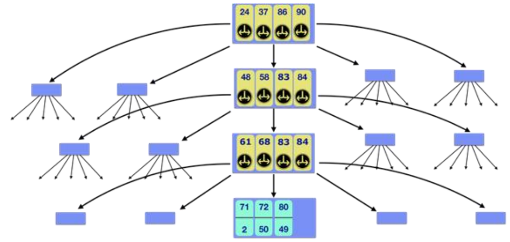
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$



B-Trees

$(\lambda = B)$

B^ε-Trees

$(\lambda = B^\varepsilon)$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$

Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

Optimal Hashing in External Memory, ***Conway***, *Farach-Colton, Shillane, ICALP 2018*

Lower Bounds

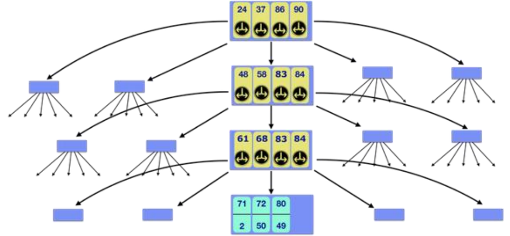
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_{\lambda}N\right)$$

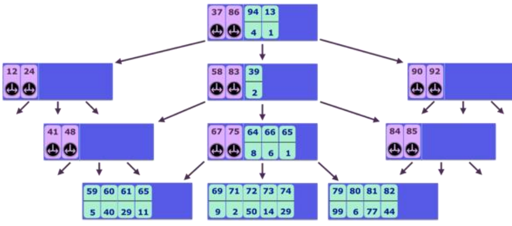
Lookups in

$$\Omega(\log_{\lambda}N)$$



B-Trees

$$(\lambda = B)$$



B^{ε}-Trees

$$(\lambda = B^{\varepsilon})$$

Comparison External Memory Model

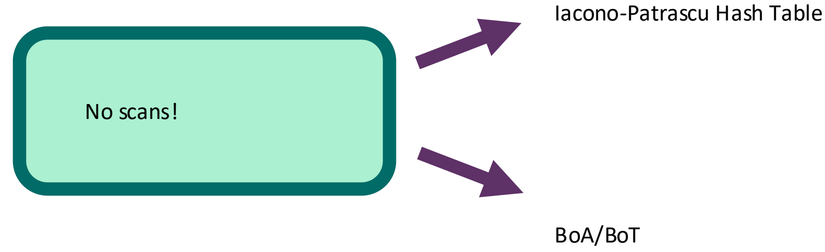
Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_{\lambda}N)$$



Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

General External Memory Model

Lower Bounds

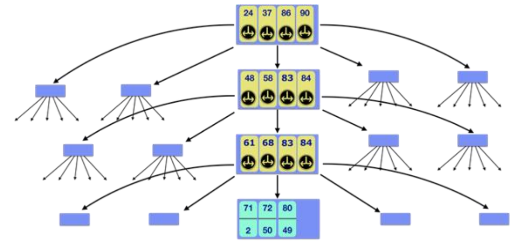
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_\lambda N\right)$$

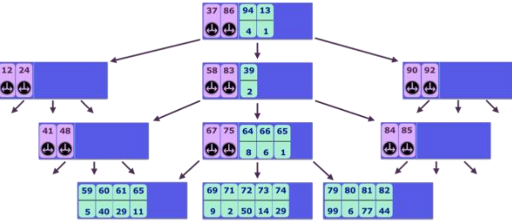
Lookups in

$$\Omega(\log_\lambda N)$$



B-Trees

$$(\lambda = B)$$



Bε-Trees

$$(\lambda = B^\varepsilon)$$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_\lambda N)$$

Mapped Bε-Trees

$$(\lambda = B^\varepsilon, B^\varepsilon = \Omega(\log_{B^\varepsilon} N))$$

Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

General External Memory Model

Lower Bounds

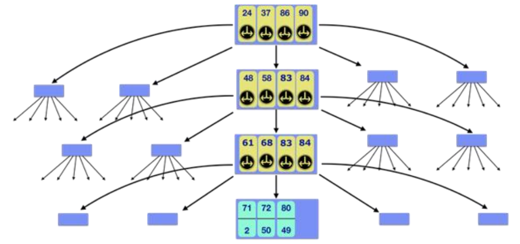
Brodal-Fagerberg Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\log_{\lambda}N\right)$$

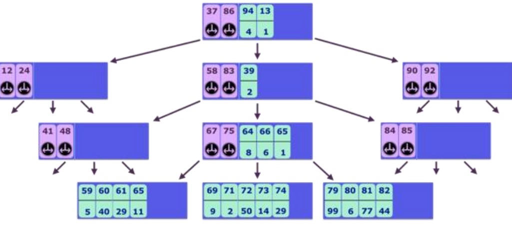
Lookups in

$$\Omega(\log_{\lambda}N)$$



B-Trees

$(\lambda = B)$



B^{ε}-Trees

$(\lambda = B^{\varepsilon})$

Comparison External Memory Model

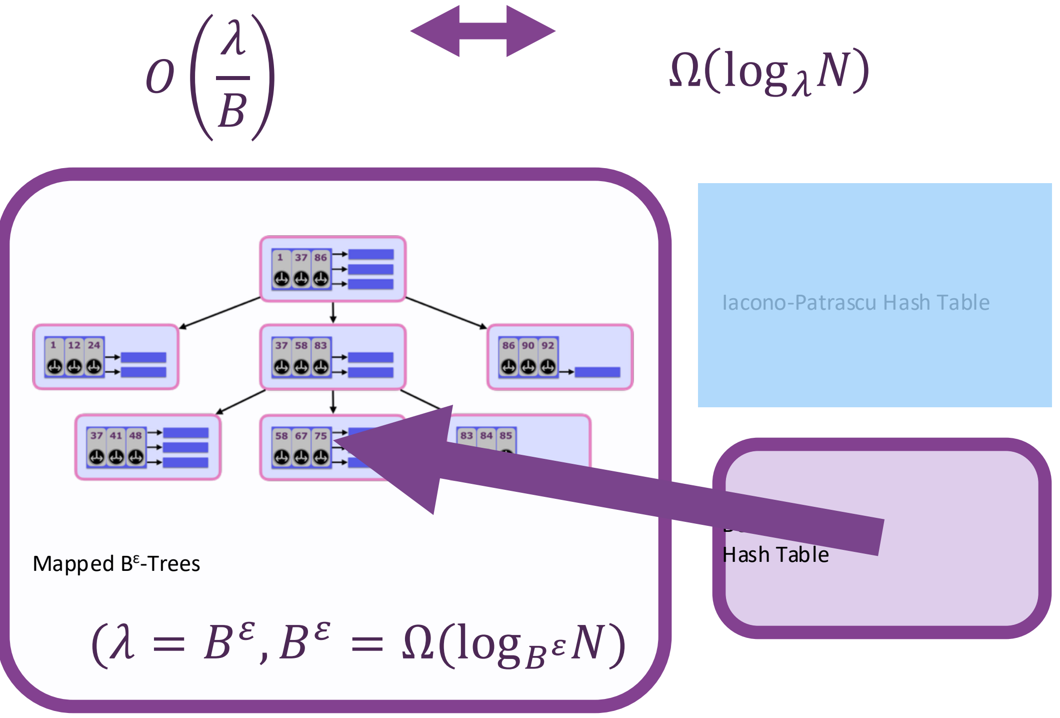
Iacono-Pătrașcu Lower Bound

Insertions in

$$O\left(\frac{\lambda}{B}\right)$$

Lookups in

$$\Omega(\log_{\lambda}N)$$



Mapped B^{ε}-Trees

$(\lambda = B^{\varepsilon}, B^{\varepsilon} = \Omega(\log_{B^{\varepsilon}}N))$

General External Memory Model

I/O Amplification

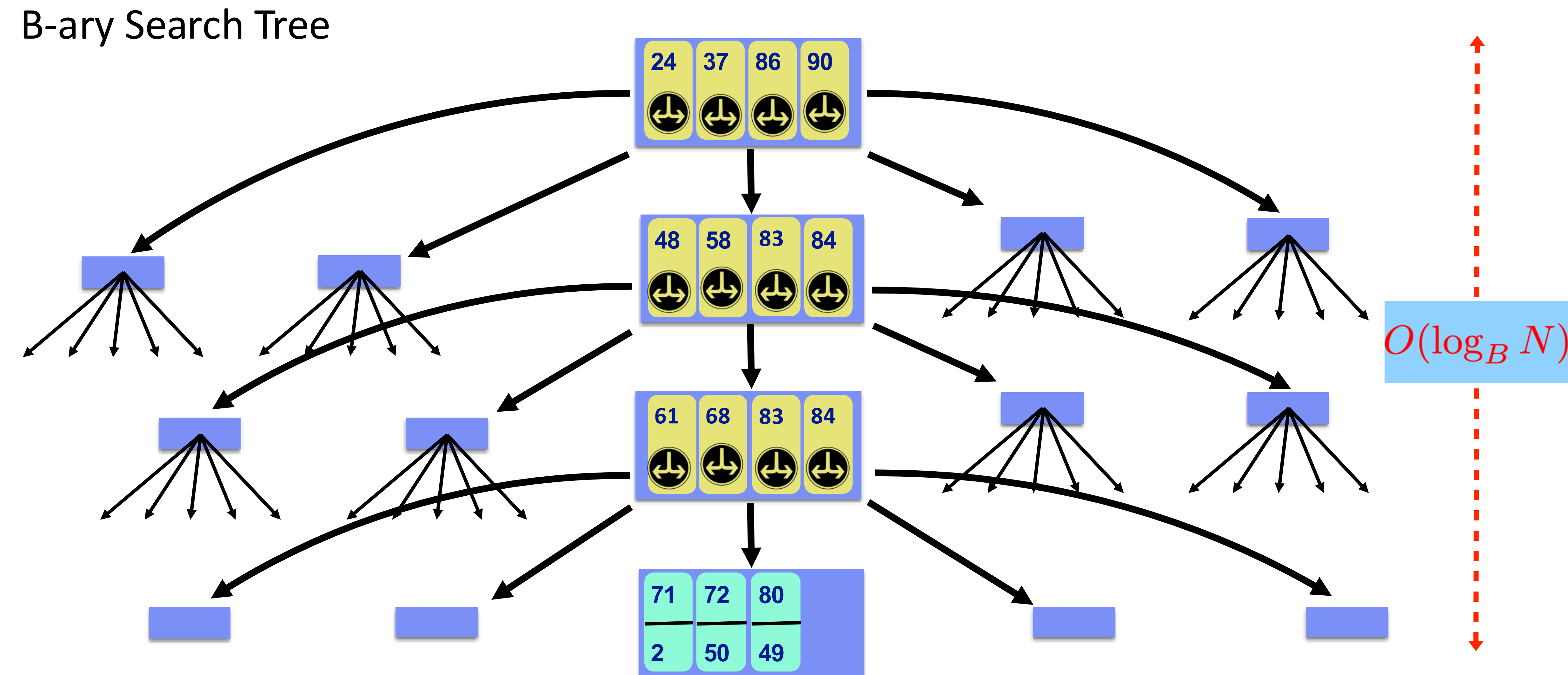
Read amplification is the ratio of the number of blocks read from the disk versus the number
of blocks required to read the key-value pair.

Write amplification is the ratio of the number of blocks written to the disk versus the number of blocks required to write the key-value pair.

B-Trees

B-Trees

B-ary Search Tree



B-Trees

B-ary Search Tree

Insert 76 / 6

24 37 86 90

48 58 83 84

61 68 83 84

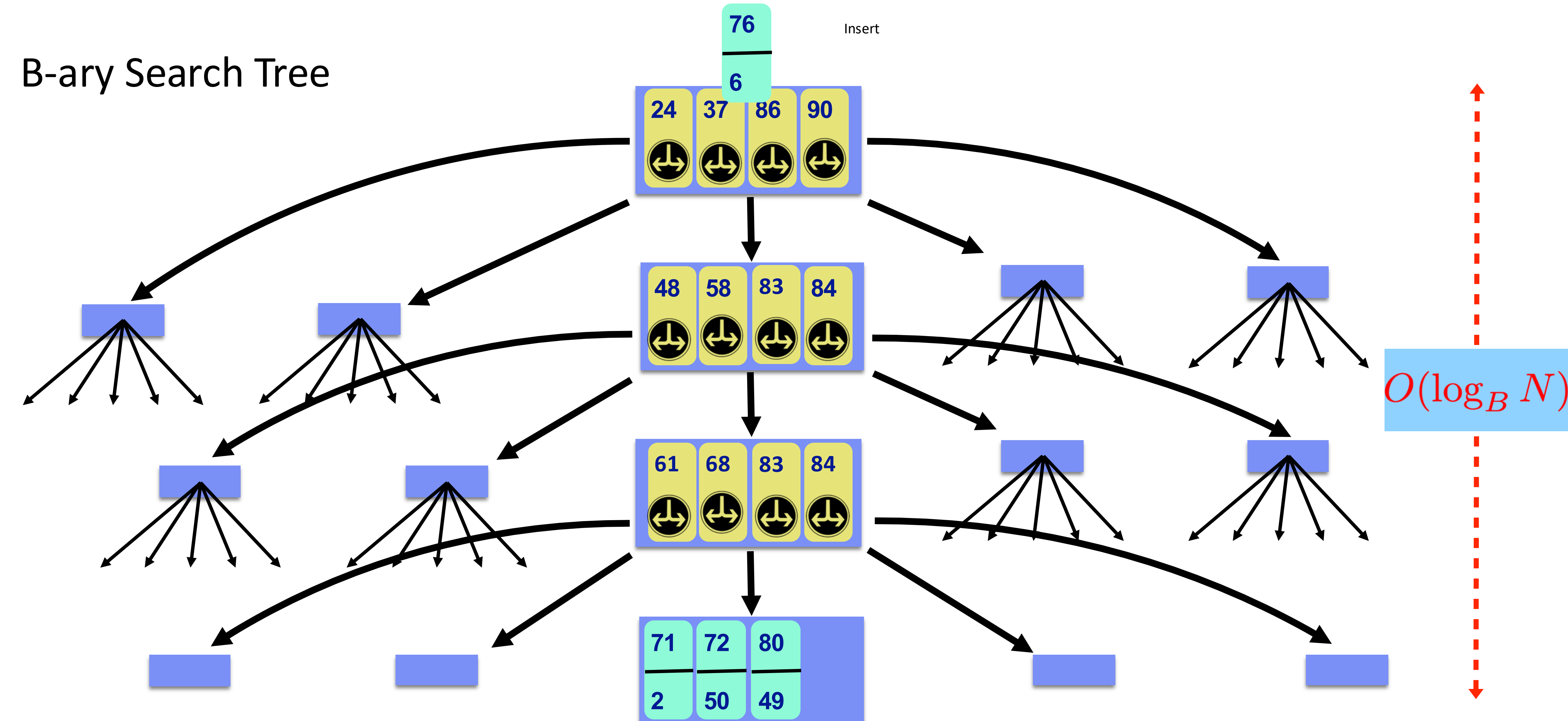
71 72 80

2 50 49

$O(\log_B N)$

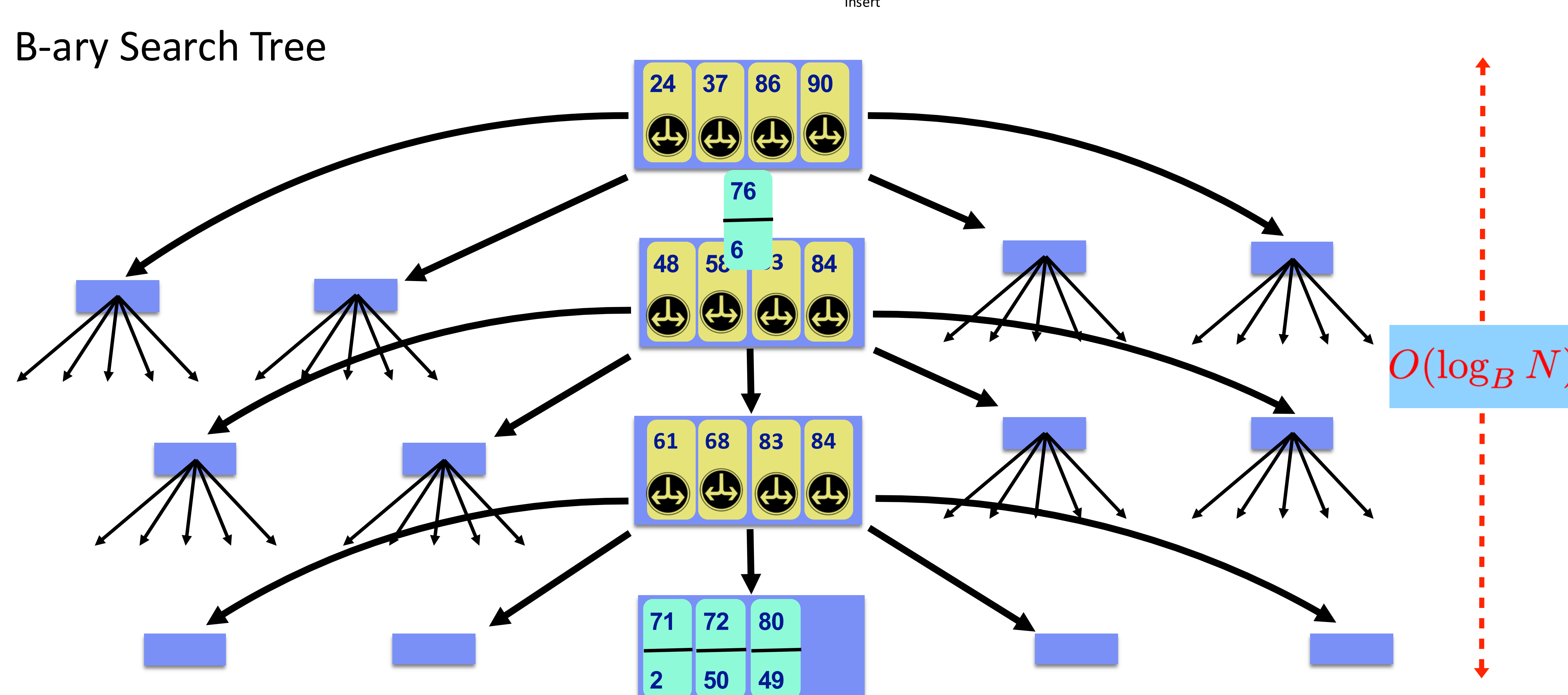
B-Trees

B-ary Search Tree



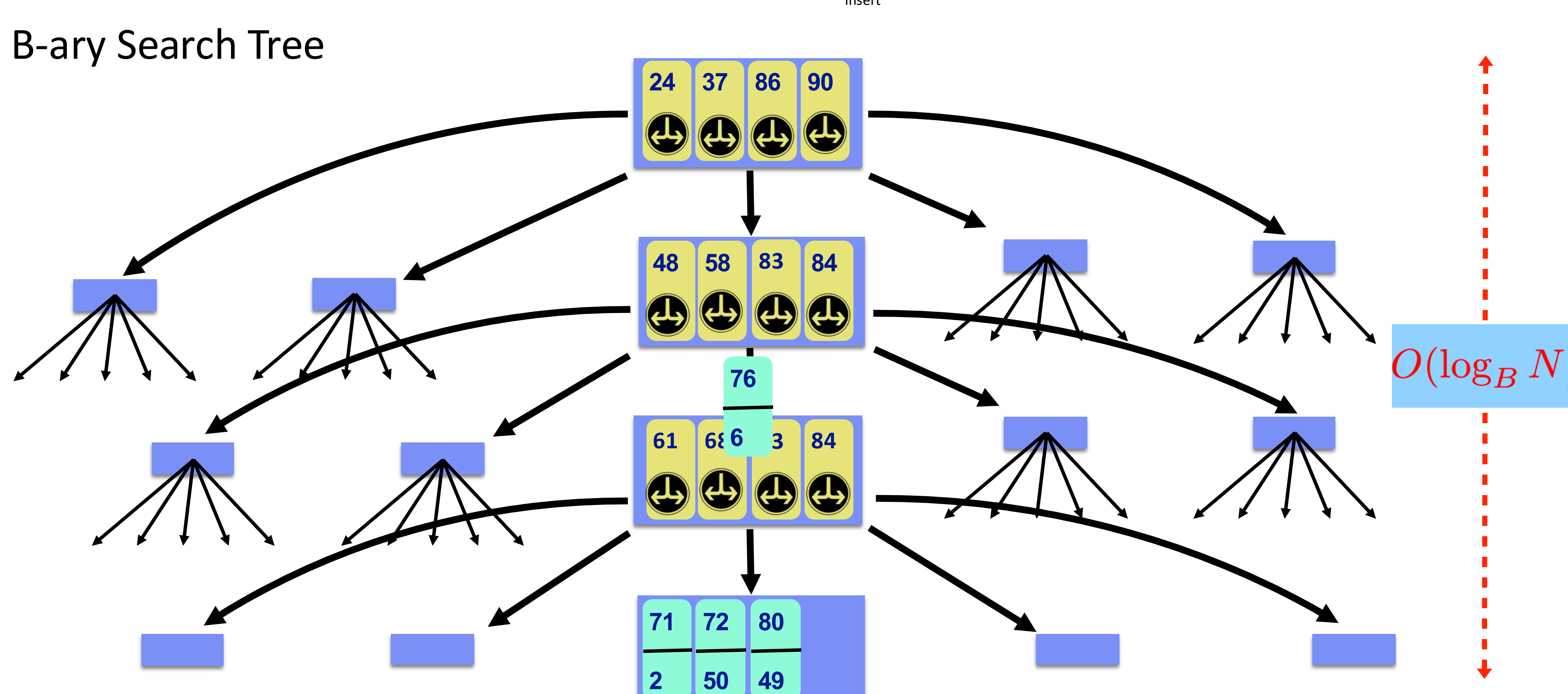
B-Trees

B-ary Search Tree



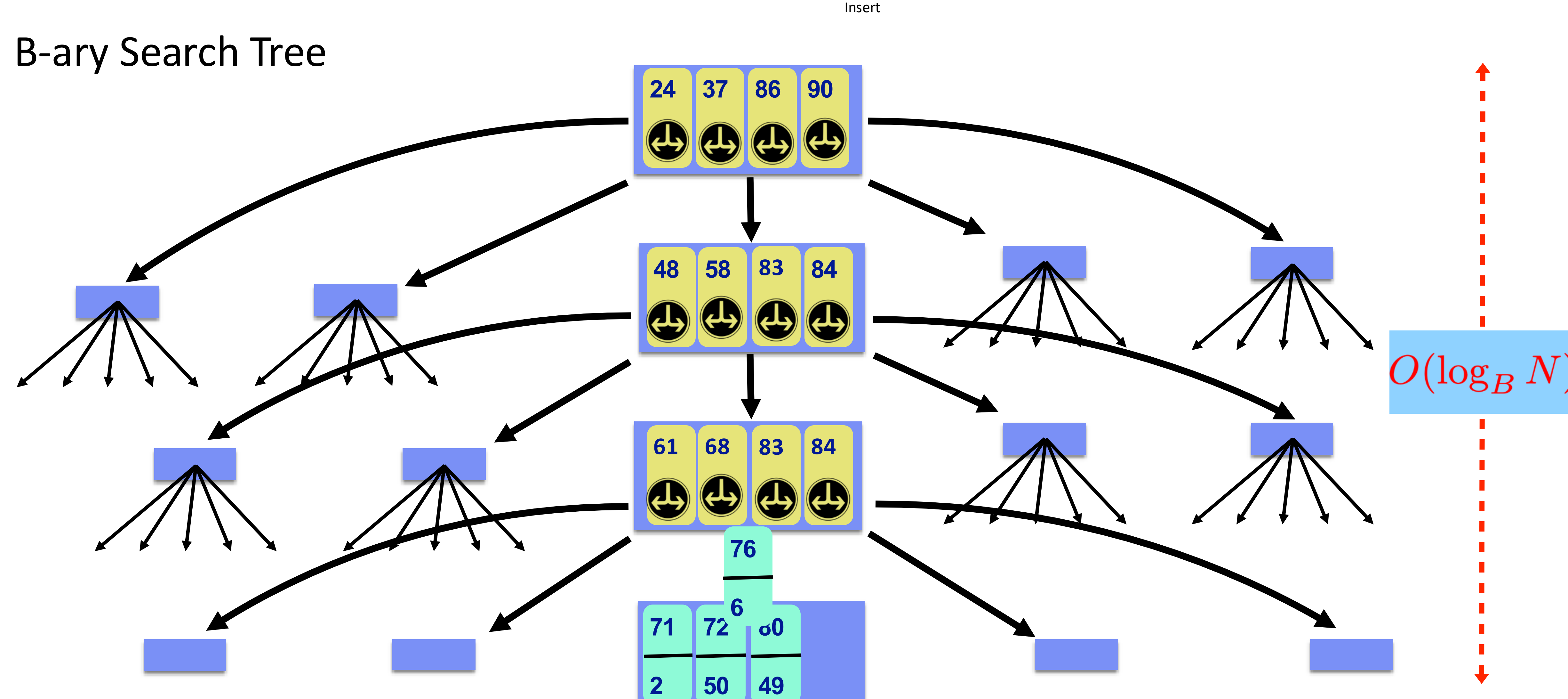
B-Trees

B-ary Search Tree



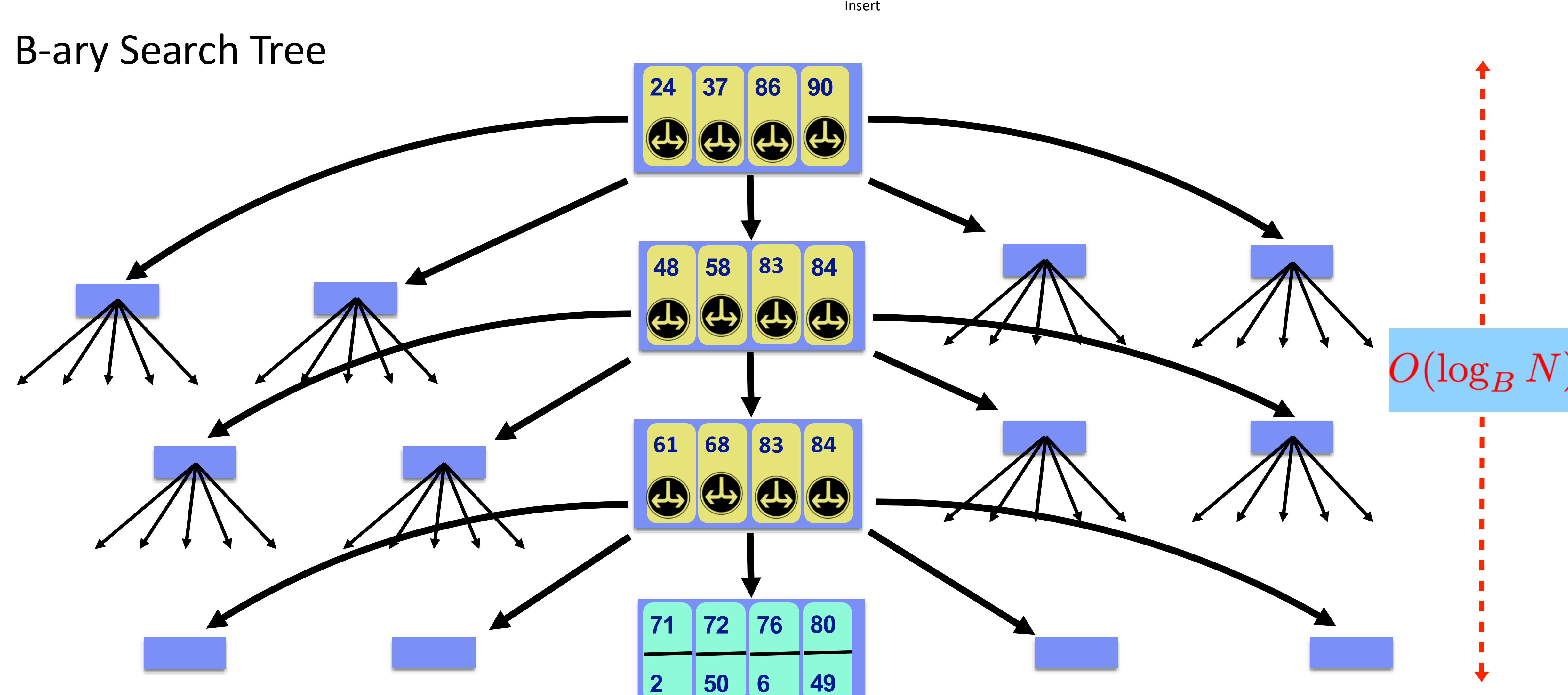
B-Trees

B-ary Search Tree



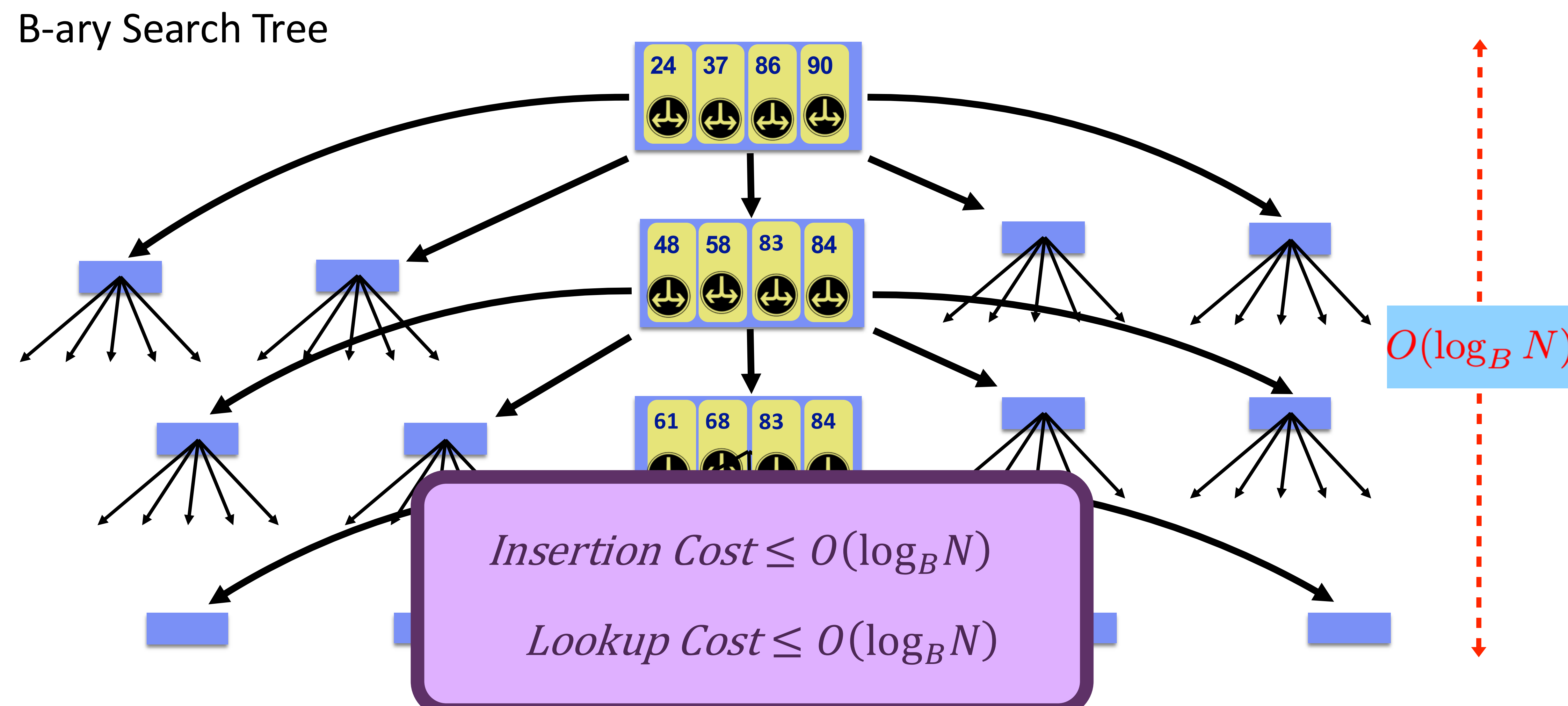
B-Trees

B-ary Search Tree



B-Trees

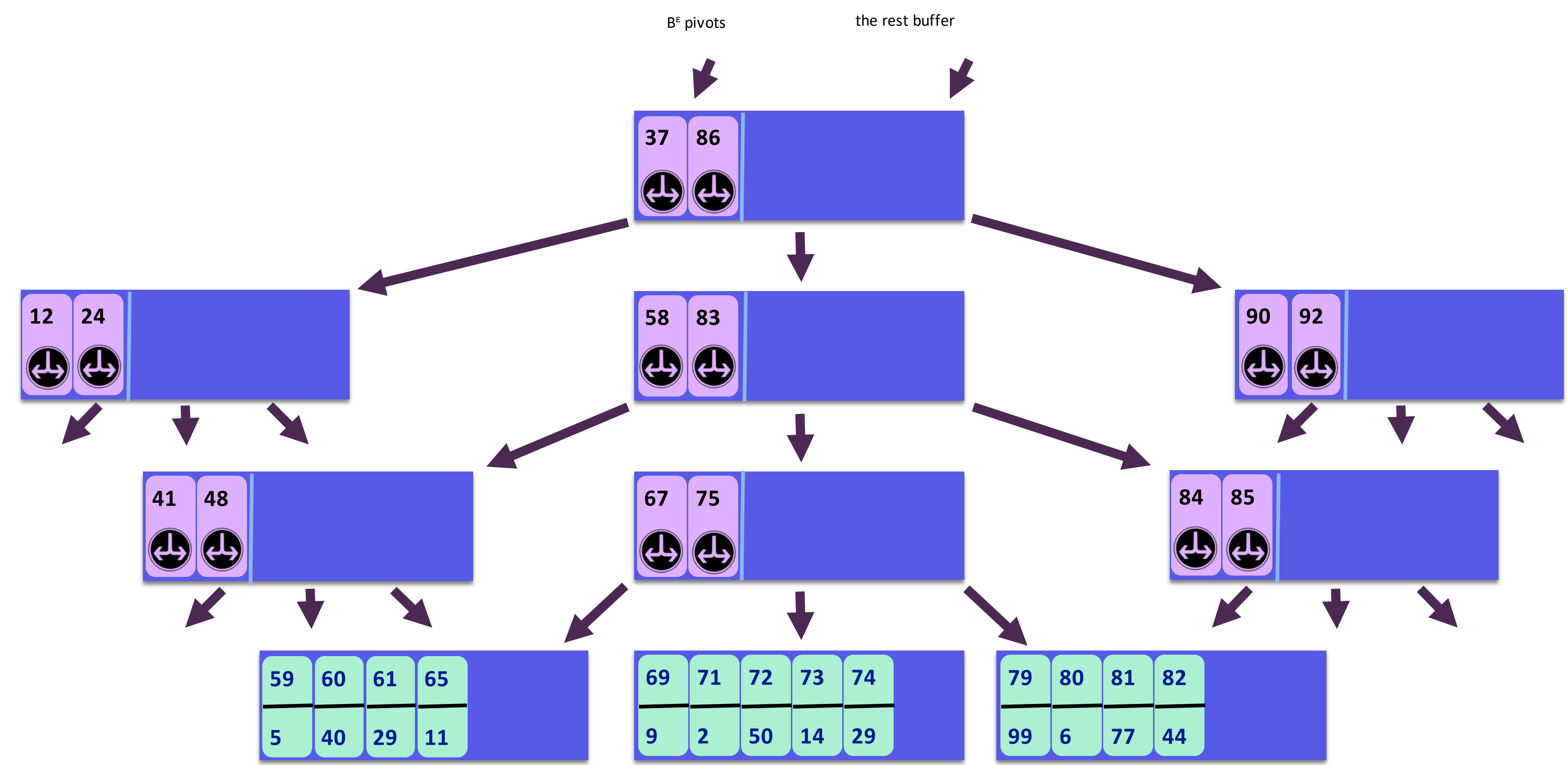
B-ary Search Tree



B^{ε}-Trees

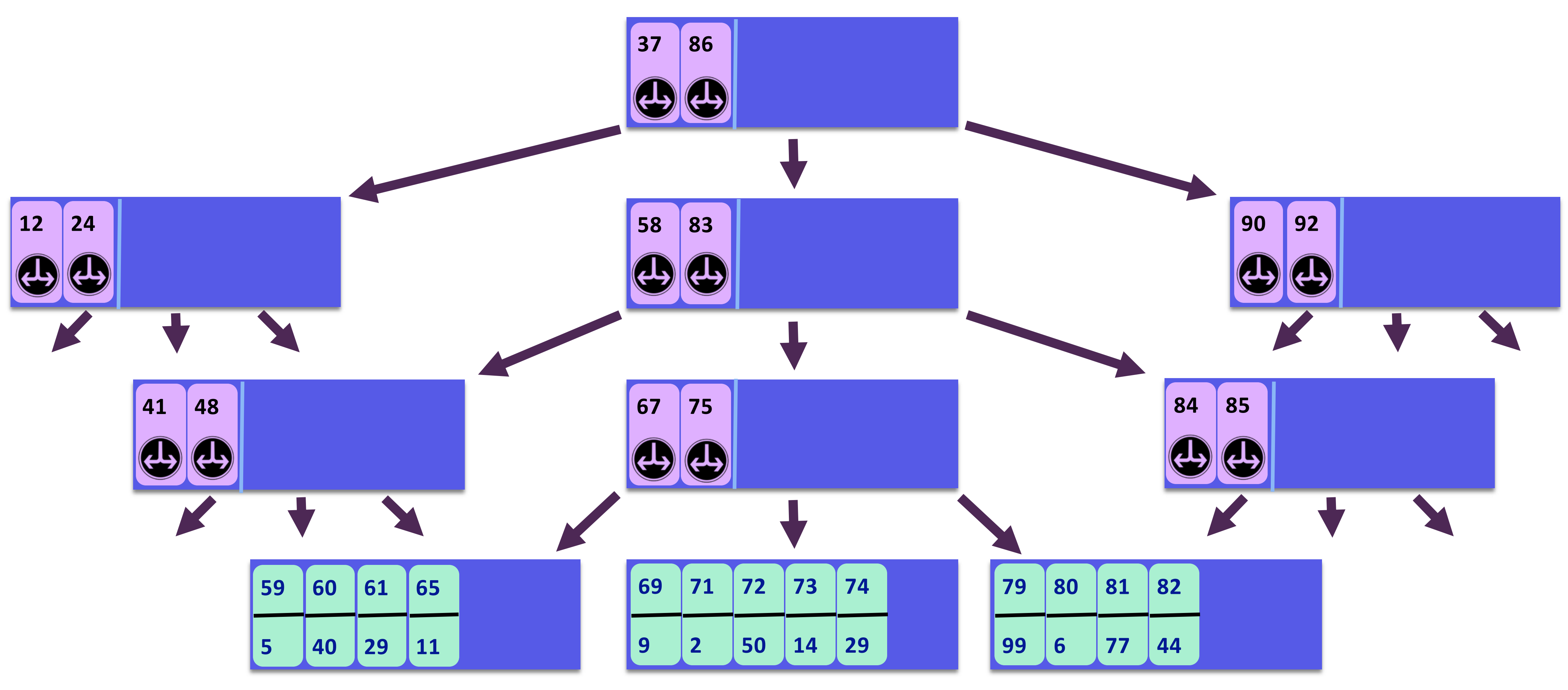
B^{ε}-Trees

A B^{ε}-tree is a search tree (like a B-tree)



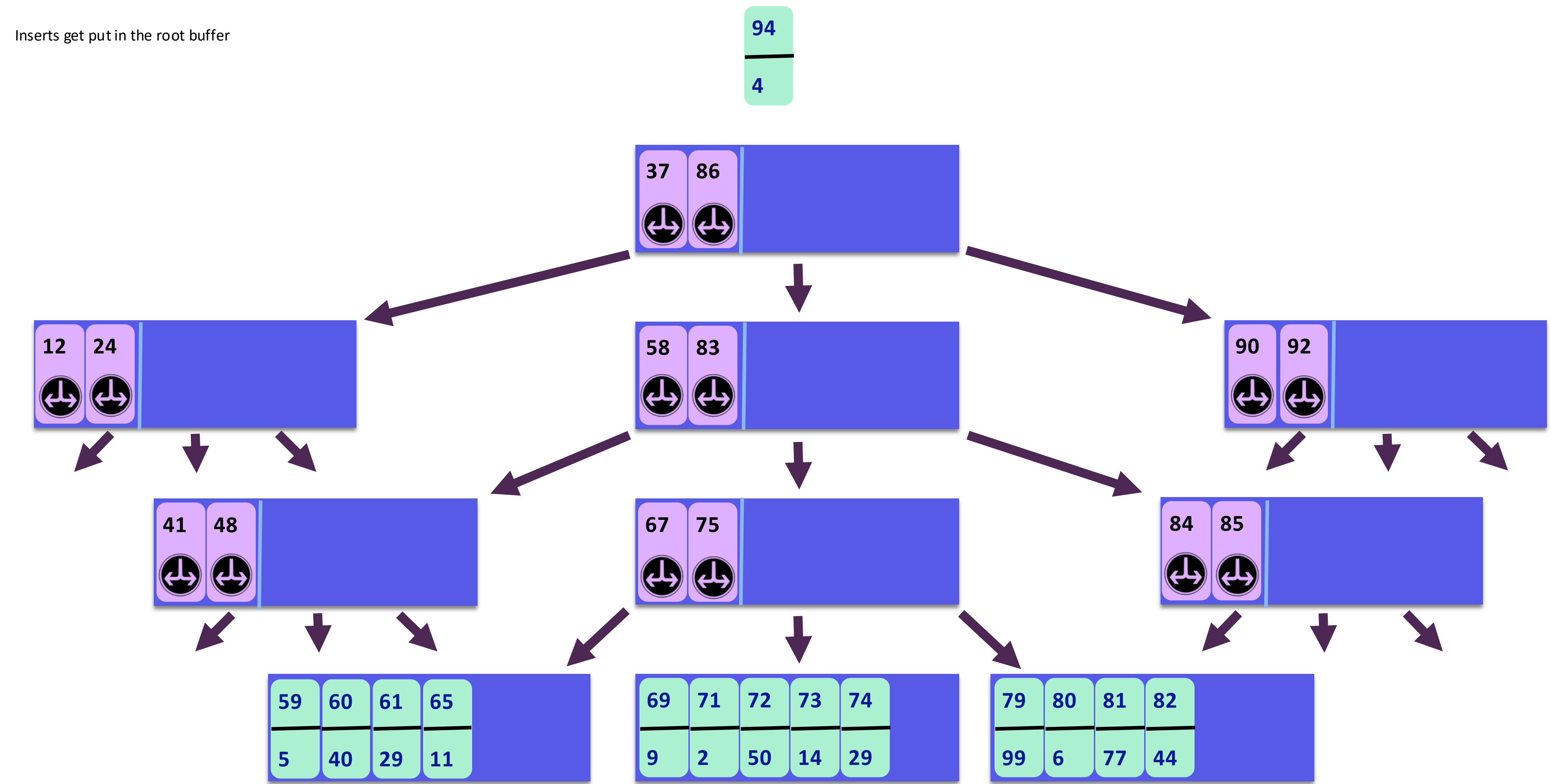
B^{ε}-Trees

Inserts get put in the root buffer



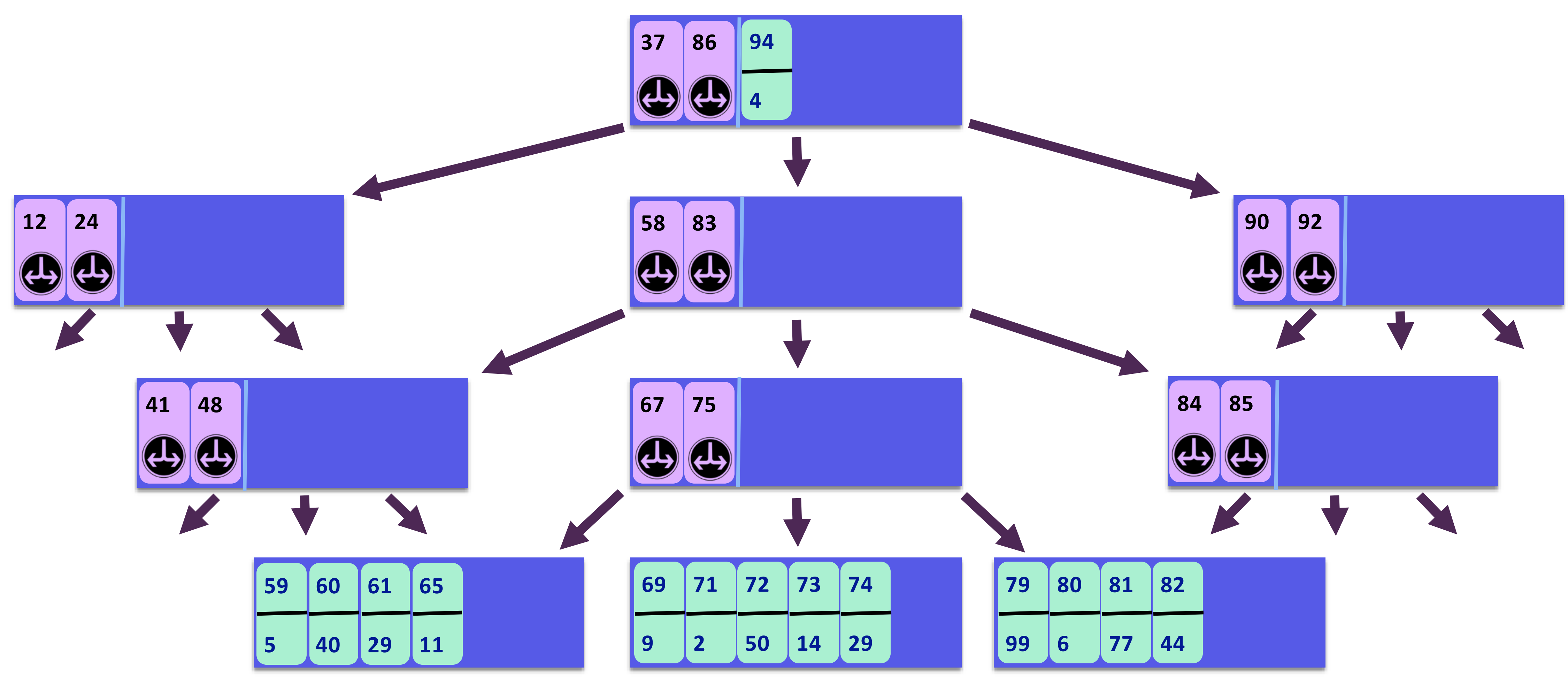
B^{ε}-Trees

Inserts get put in the root buffer



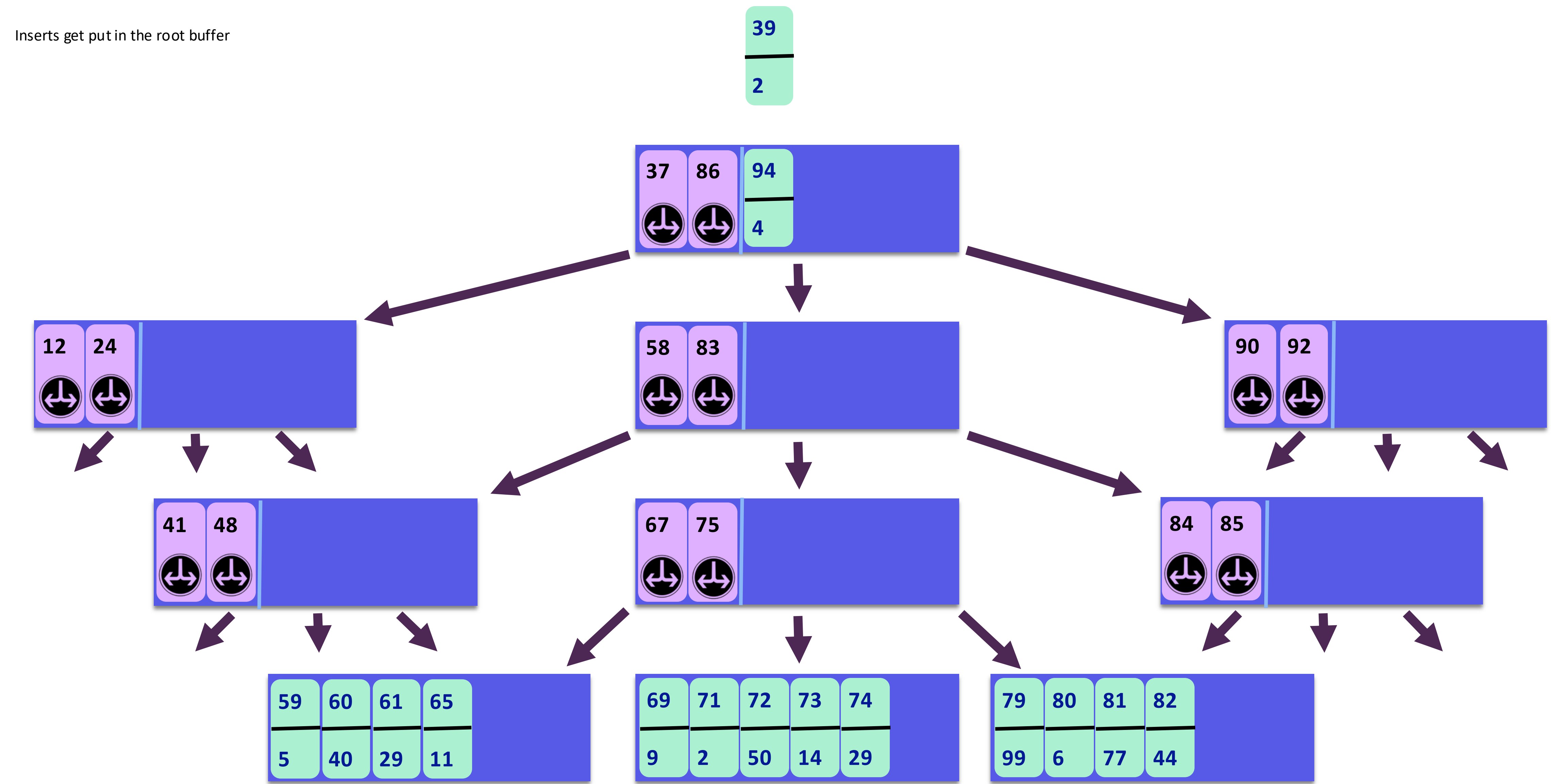
B^{ε}-Trees

Inserts get put in the root buffer



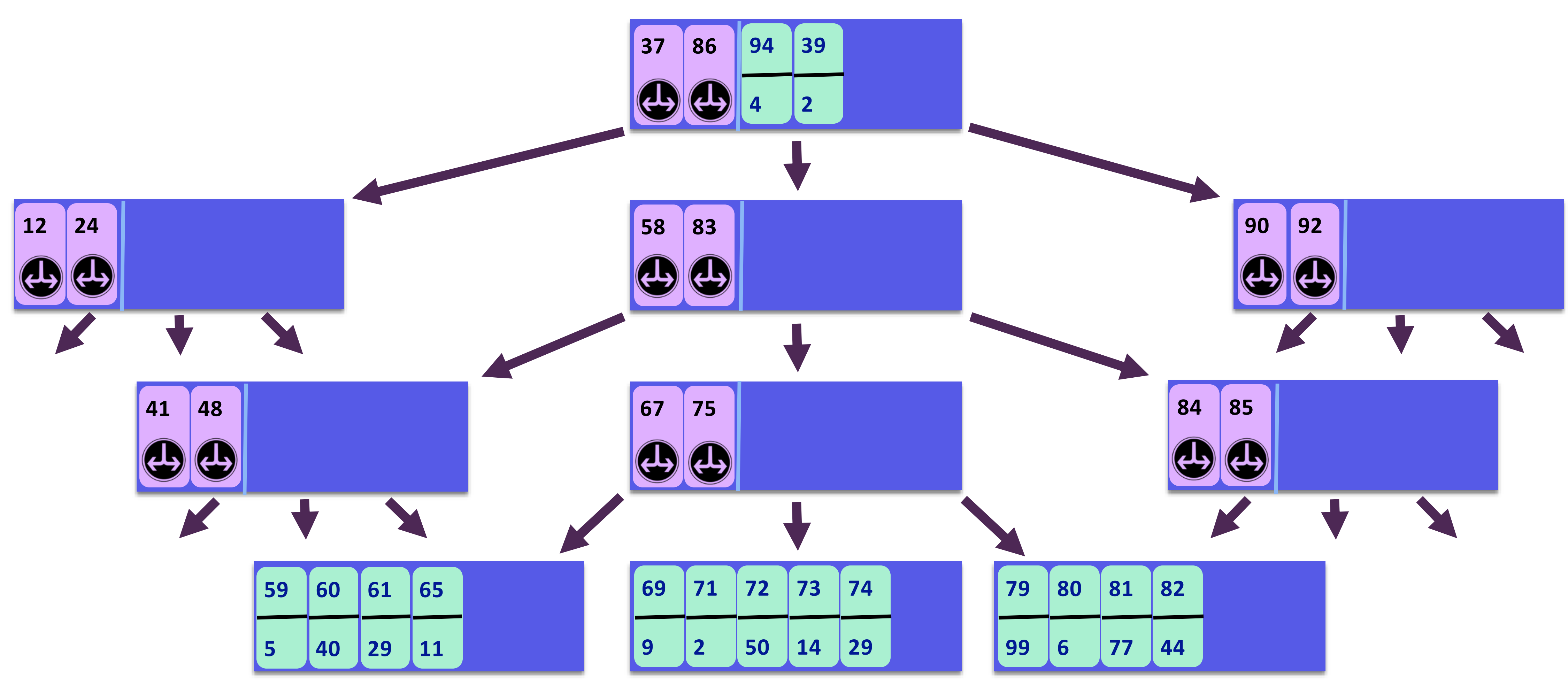
B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



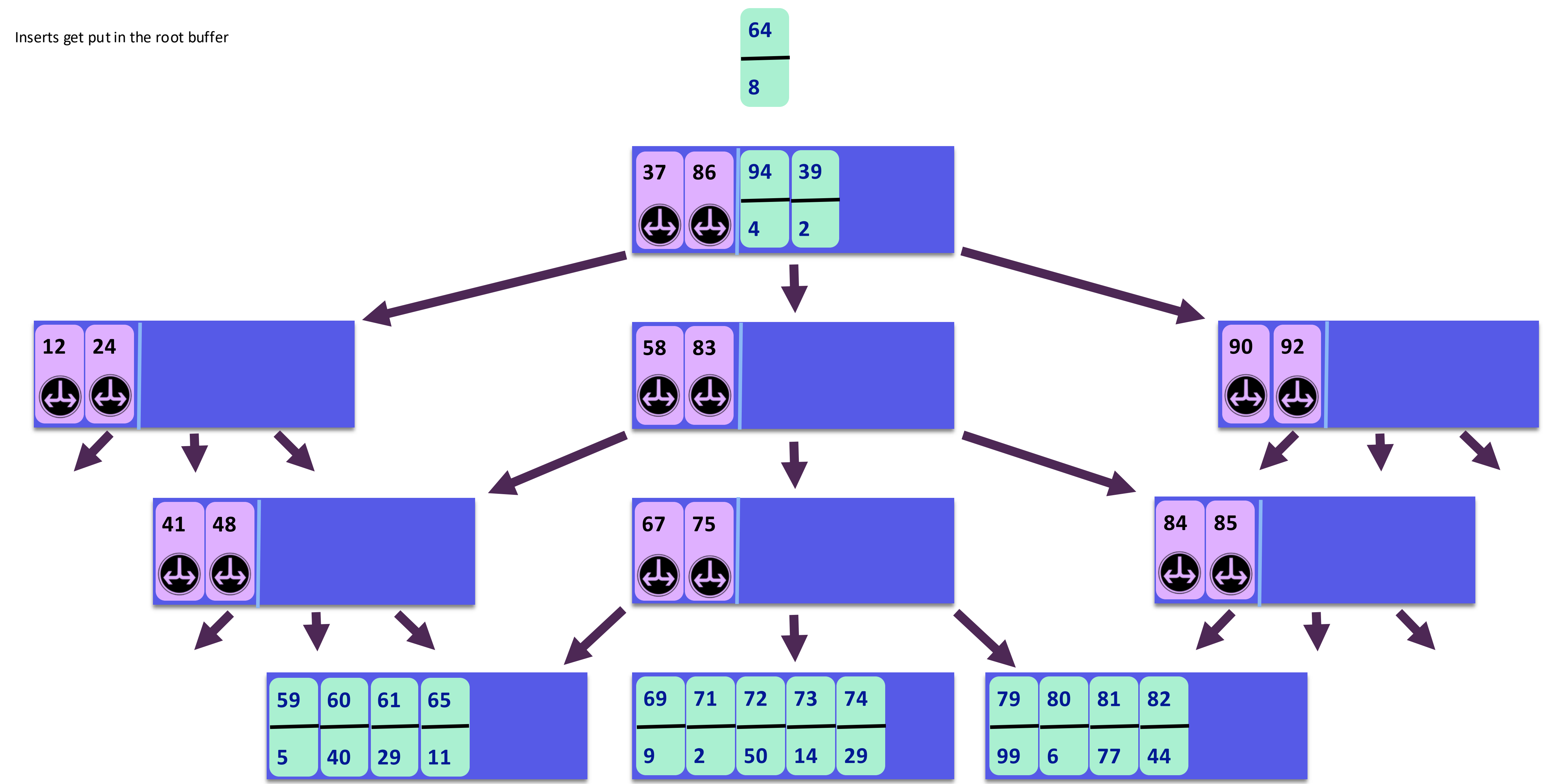
B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



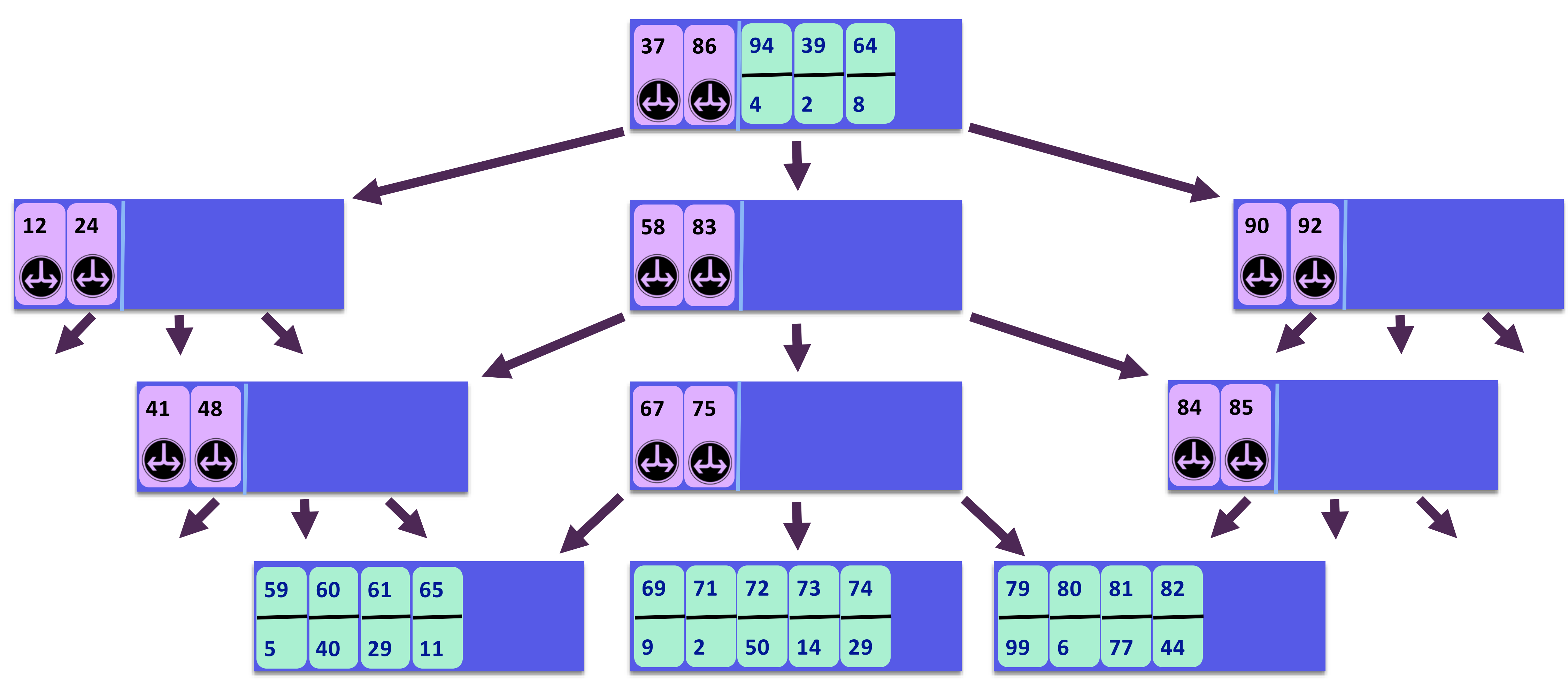
Bε-Trees

Inserts get put in the root buffer



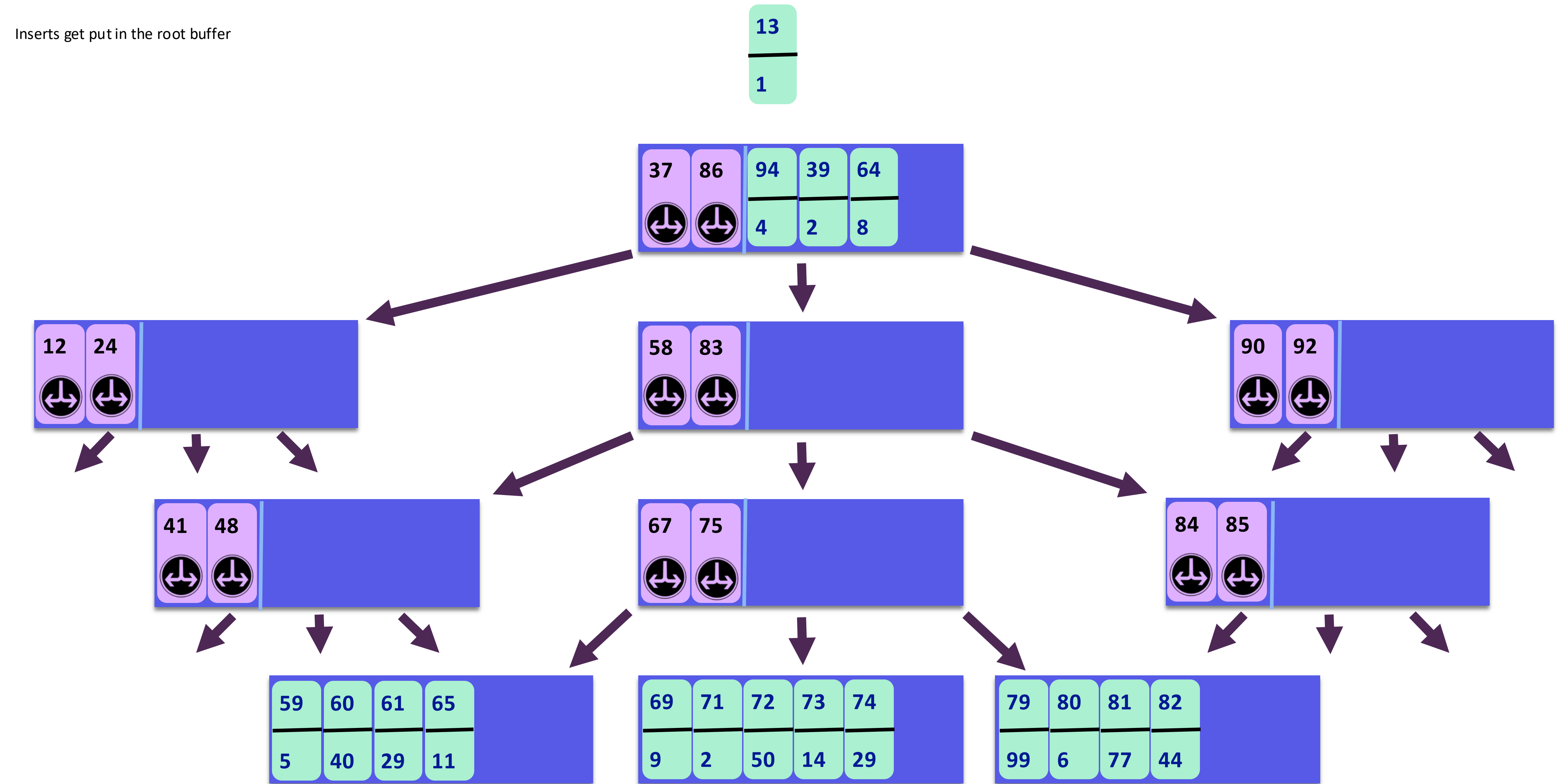
B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



B^{ε}-Trees

Inserts get put in the root buffer

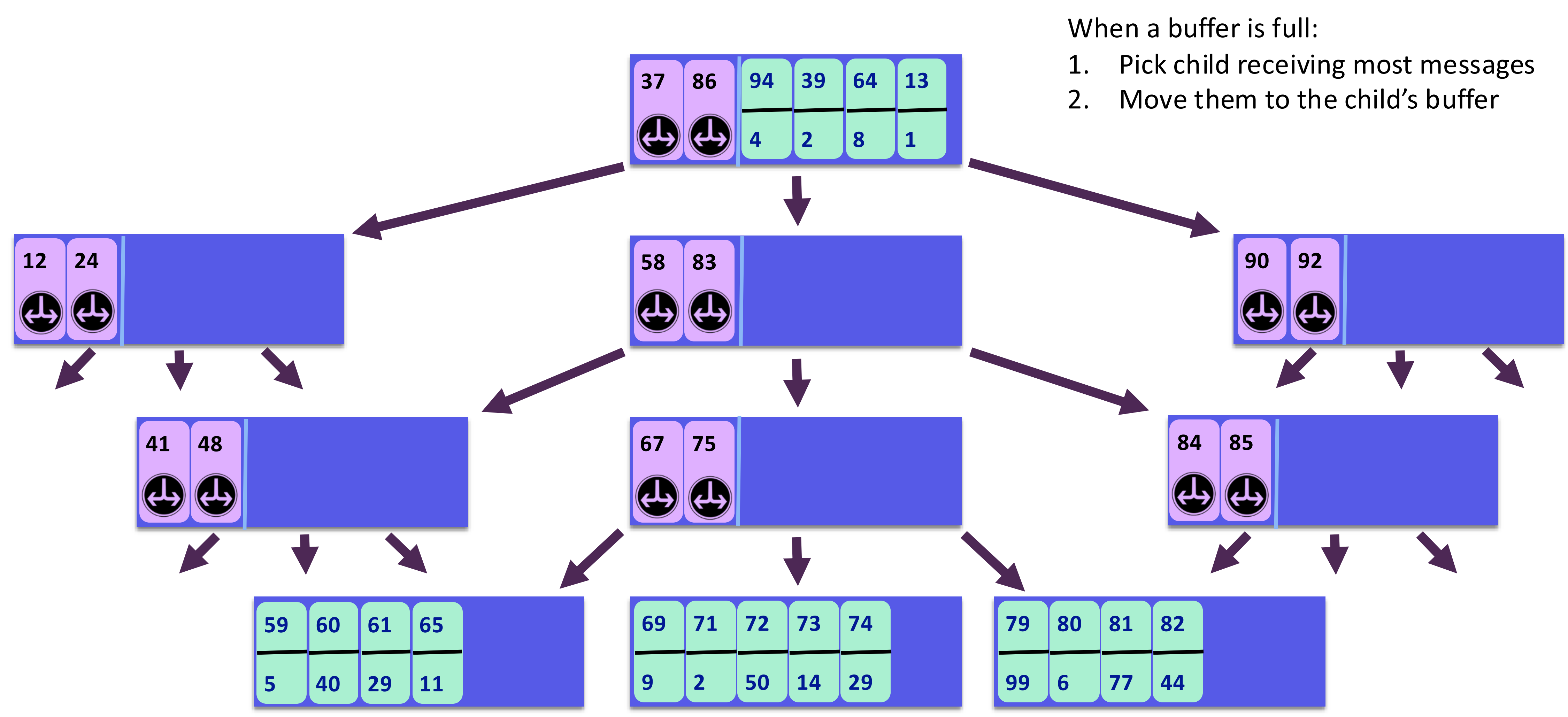


B^{ε}-Trees

Inserts get put in the root buffer

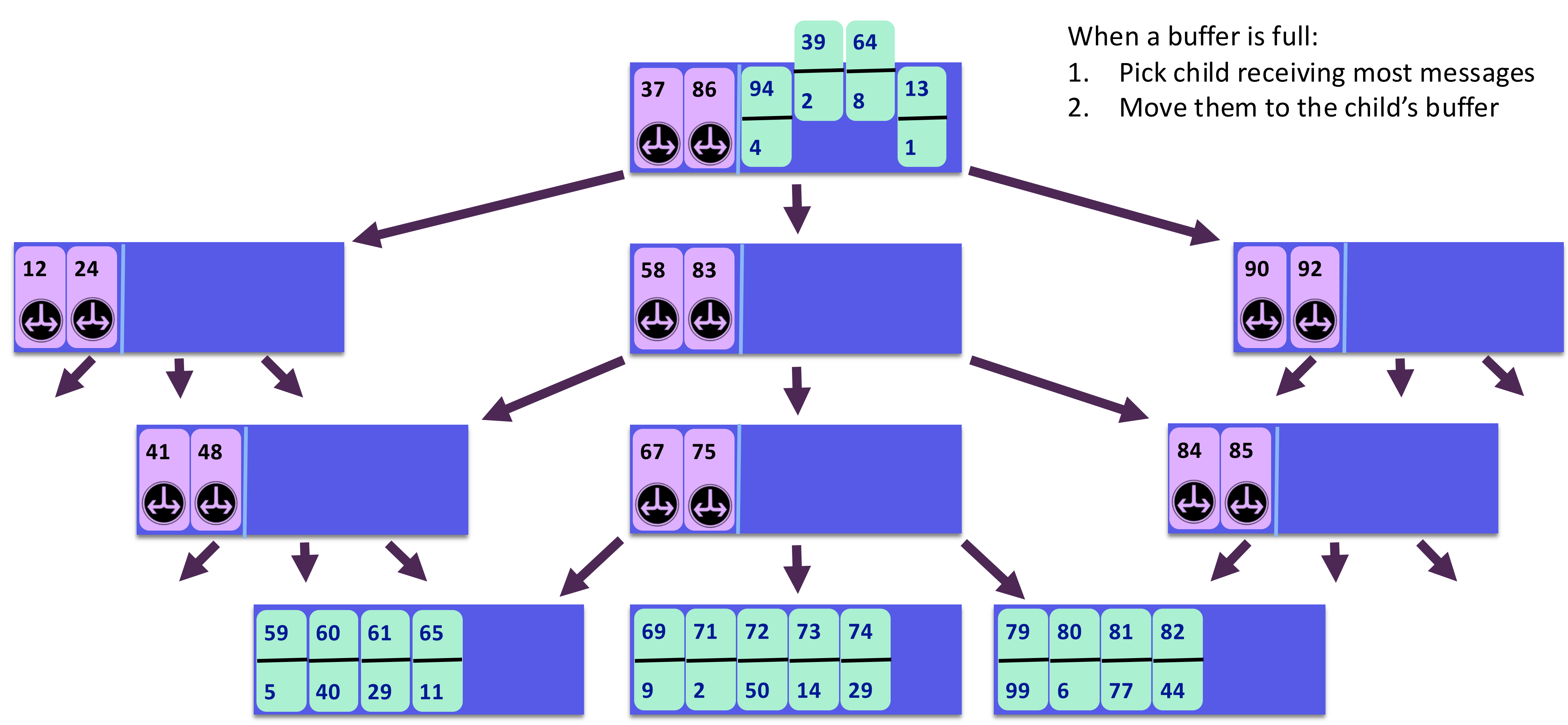
When a buffer is full:

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



B^{ε}-Trees

Inserts get put in the root buffer

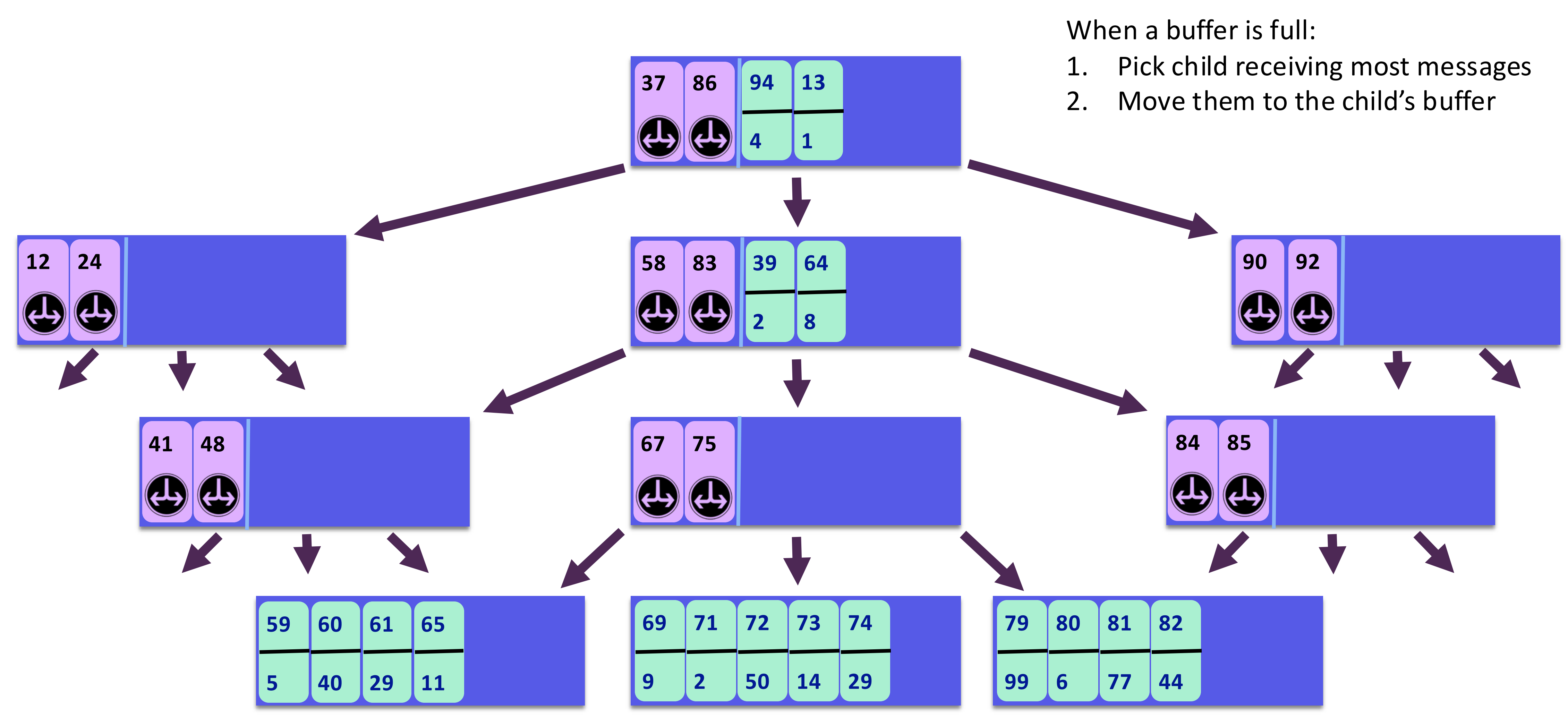


B^{ε}-Trees

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

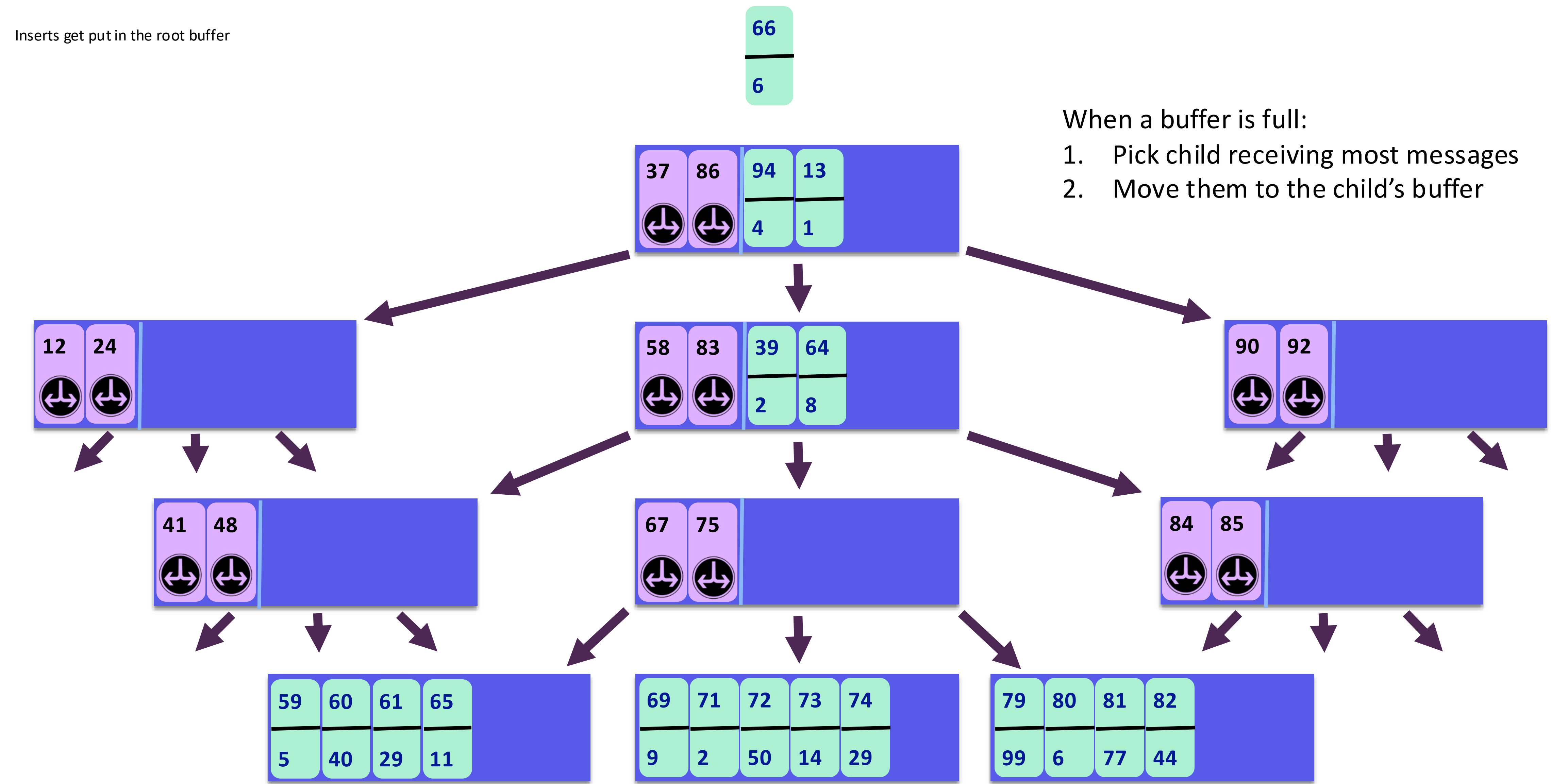


B^{ε}-Trees

Inserts get put in the root buffer

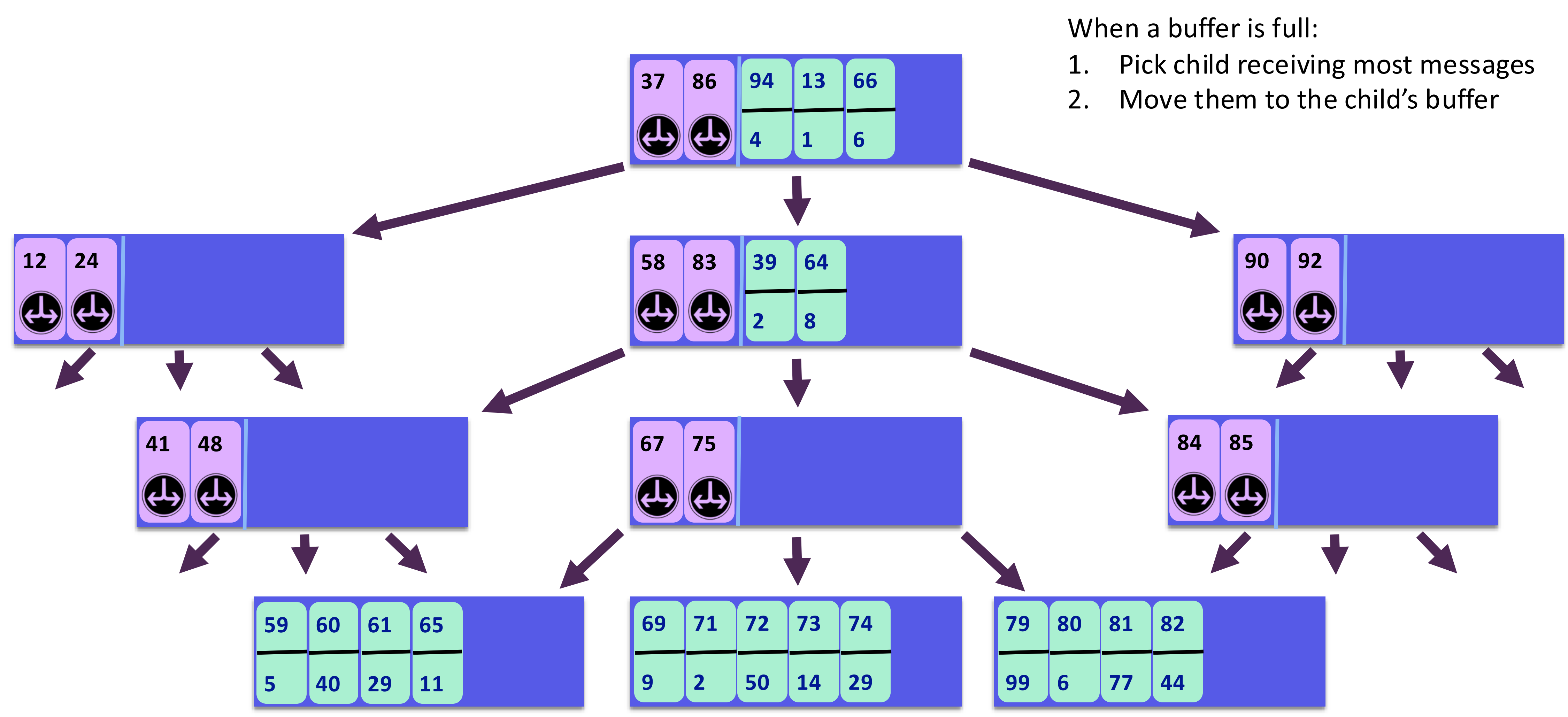
When a buffer is full:

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



B-Trees

Inserts get put in the root buffer

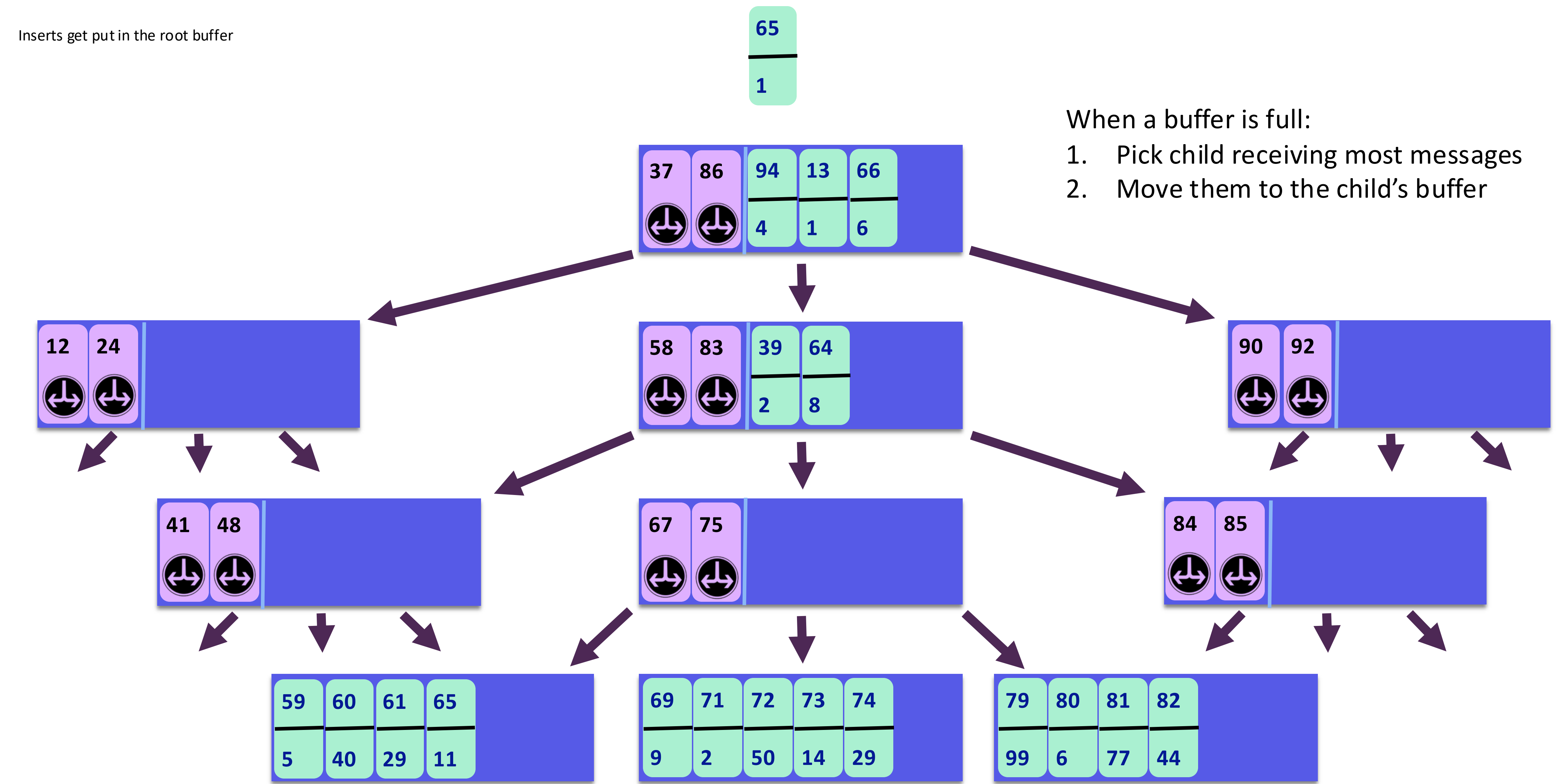


B^{ε}-Trees

Inserts get put in the root buffer

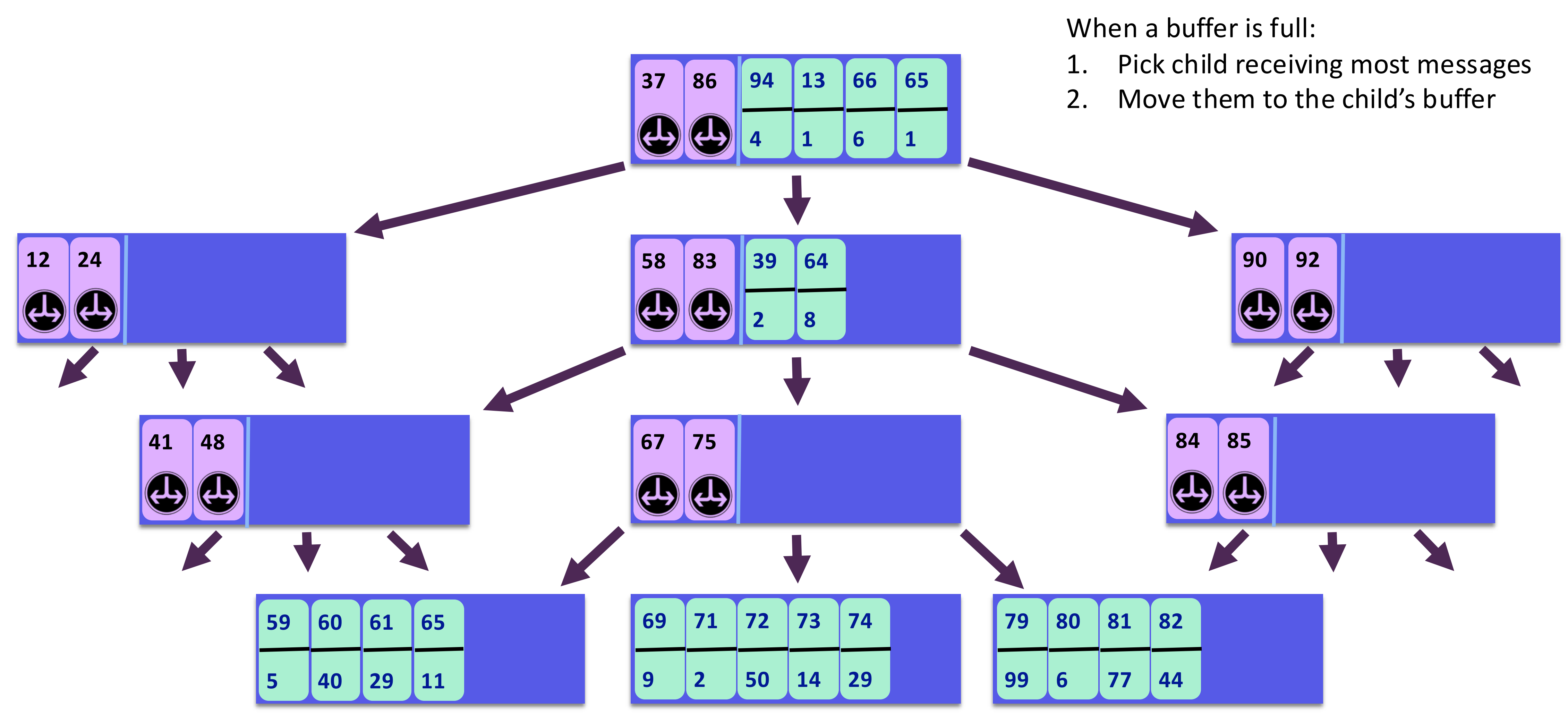
When a buffer is full:

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



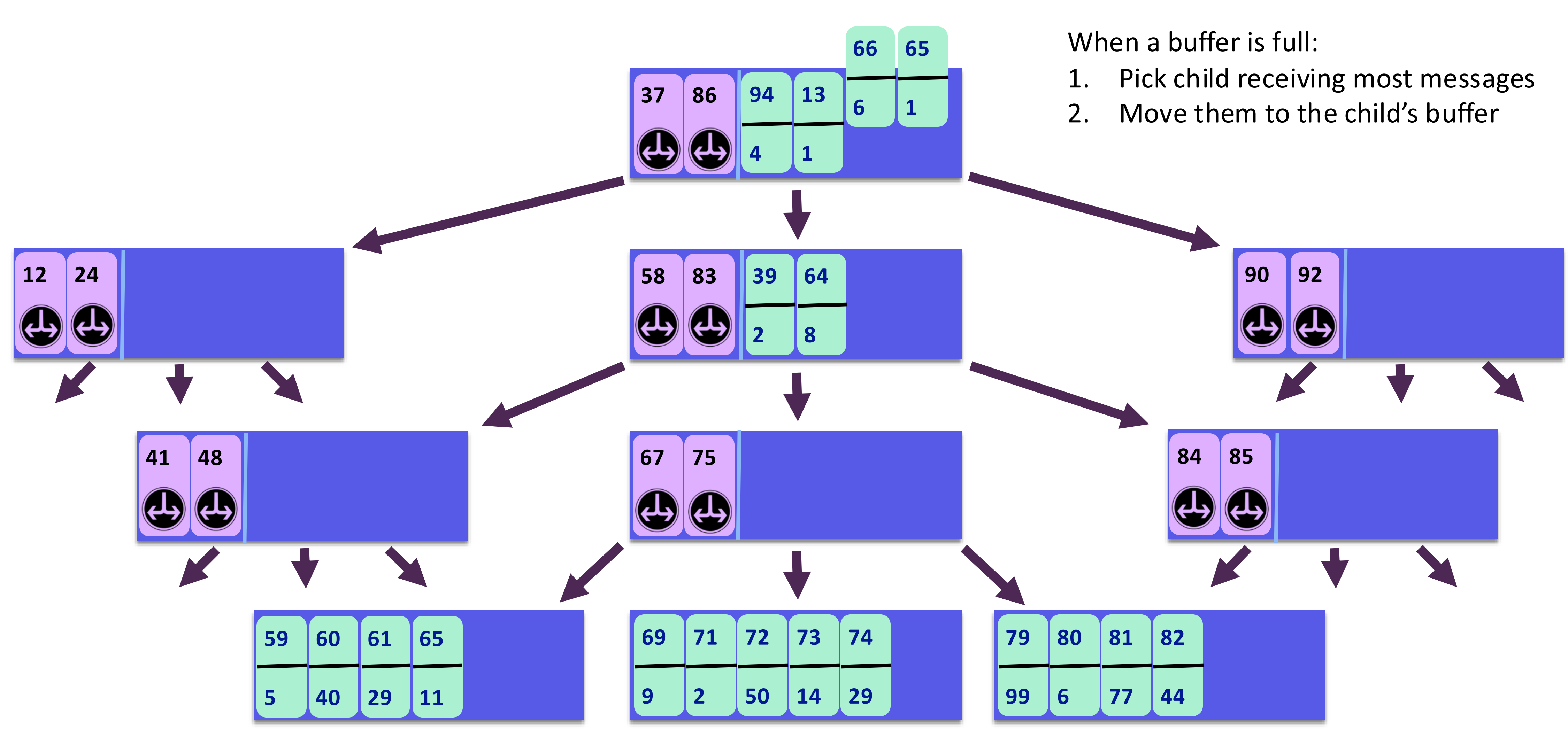
B^{ε}-Trees

Inserts get put in the root buffer



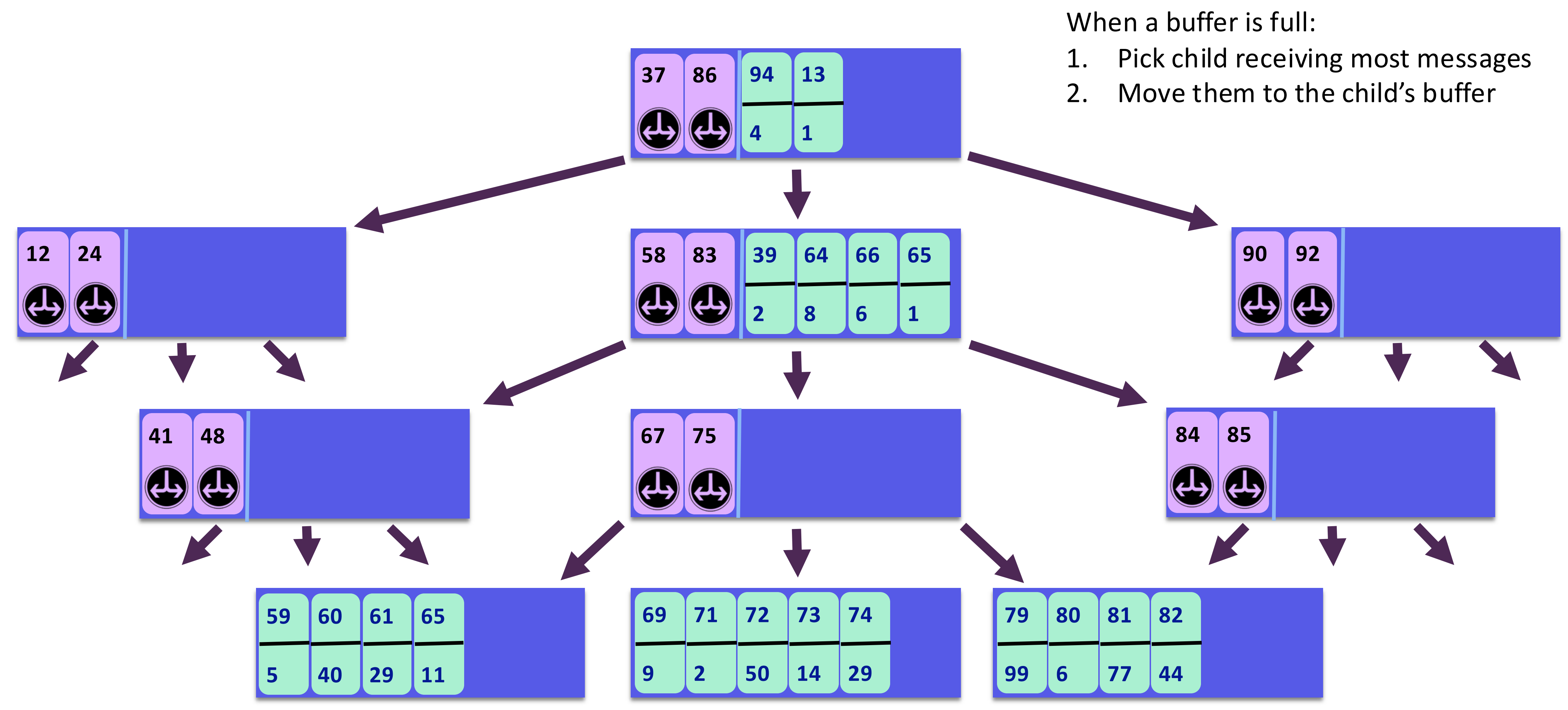
B^{ε}-Trees

Inserts get put in the root buffer



B^{ε}-Trees

Inserts get put in the root buffer

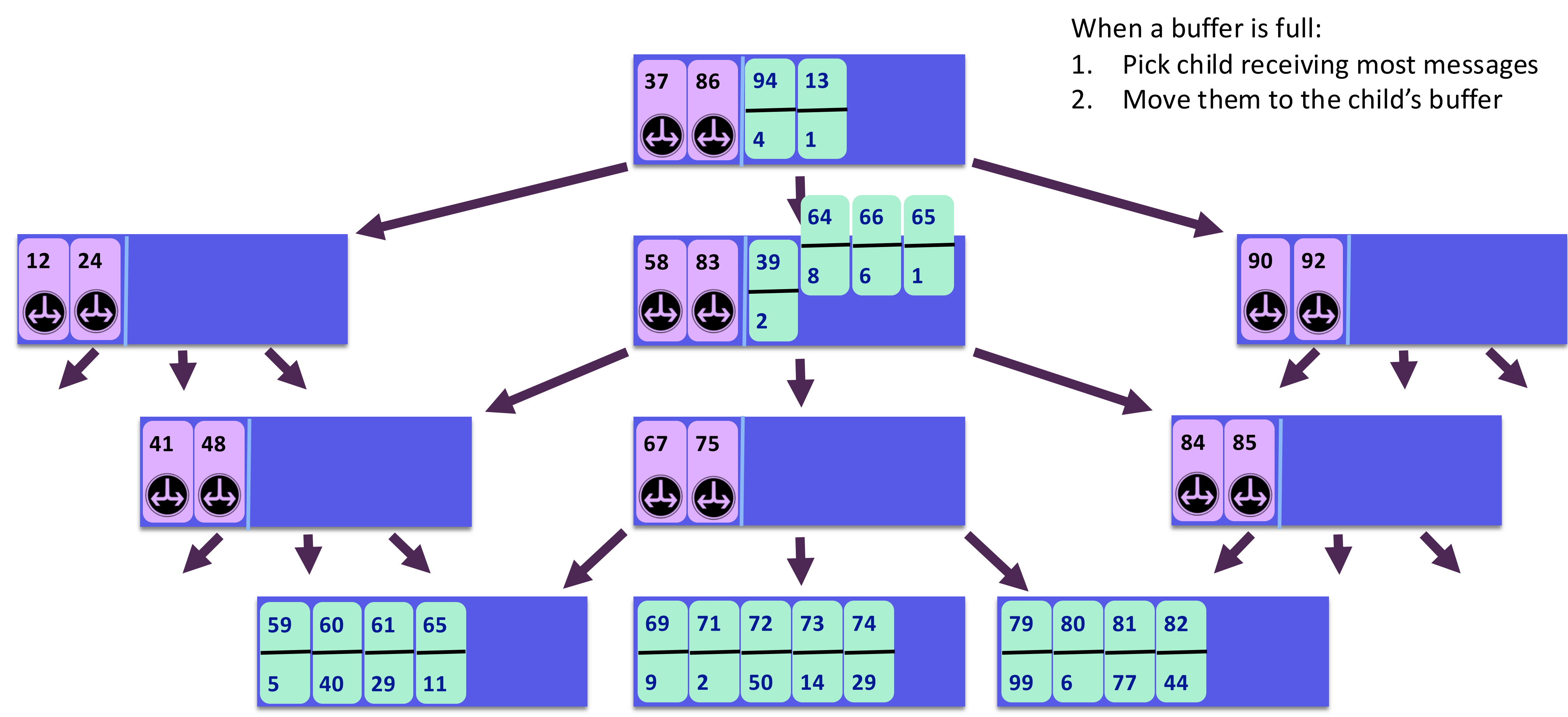


B^{ε}-Trees

Inserts get put in the root buffer

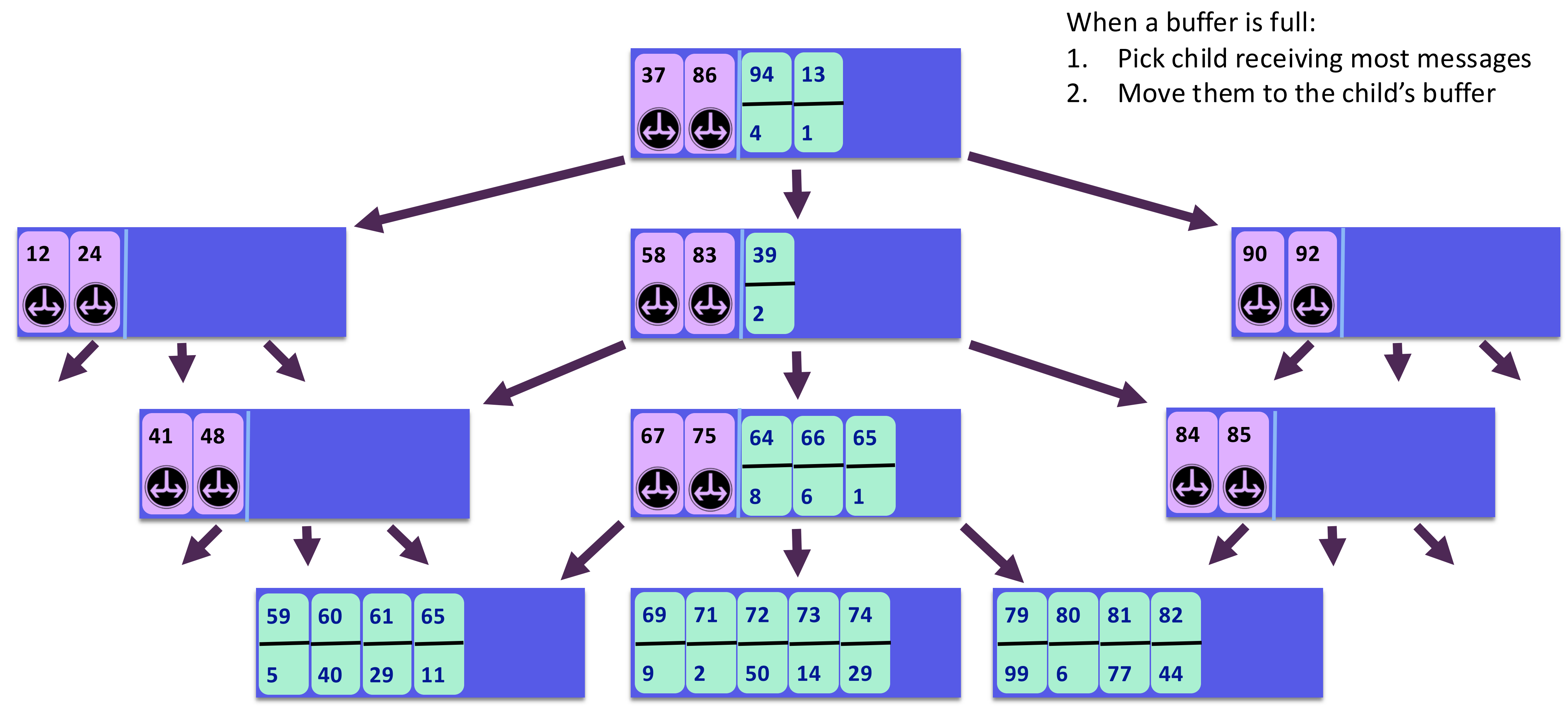
When a buffer is full:

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



B^{ε}-Trees

Inserts get put in the root buffer

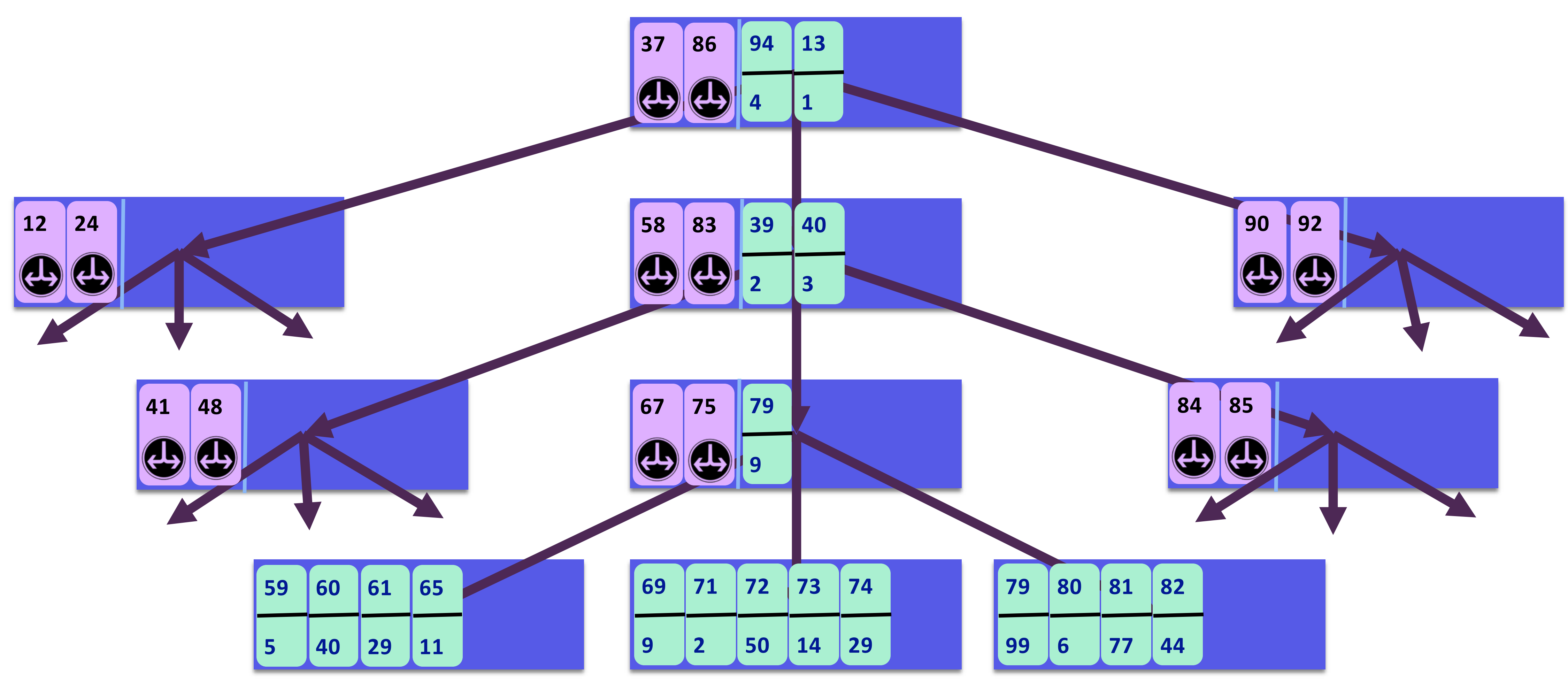


Lookups in B^{ε}-Trees

Bε-Trees

Query(71)

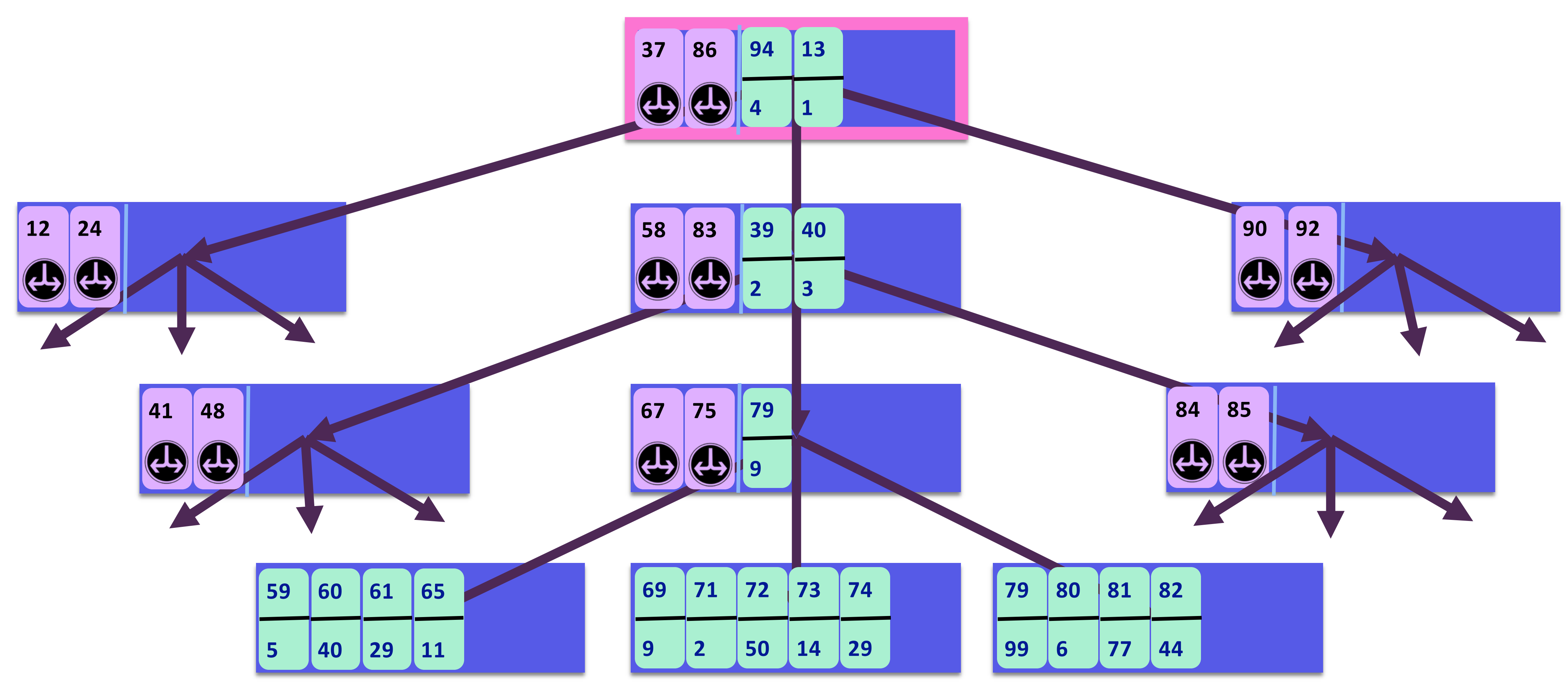
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Query(71)

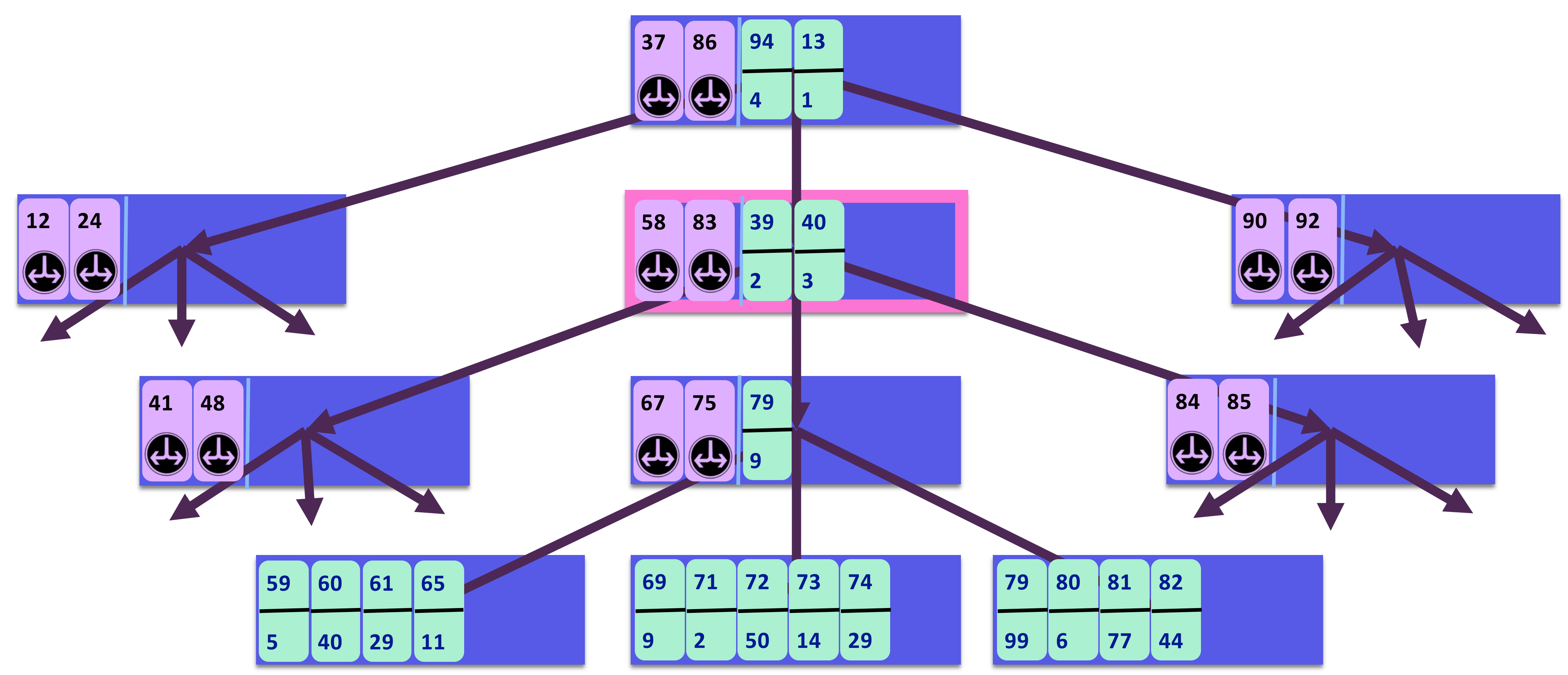
Lookups follow pivots, but check buffers along the way



Bε-Trees

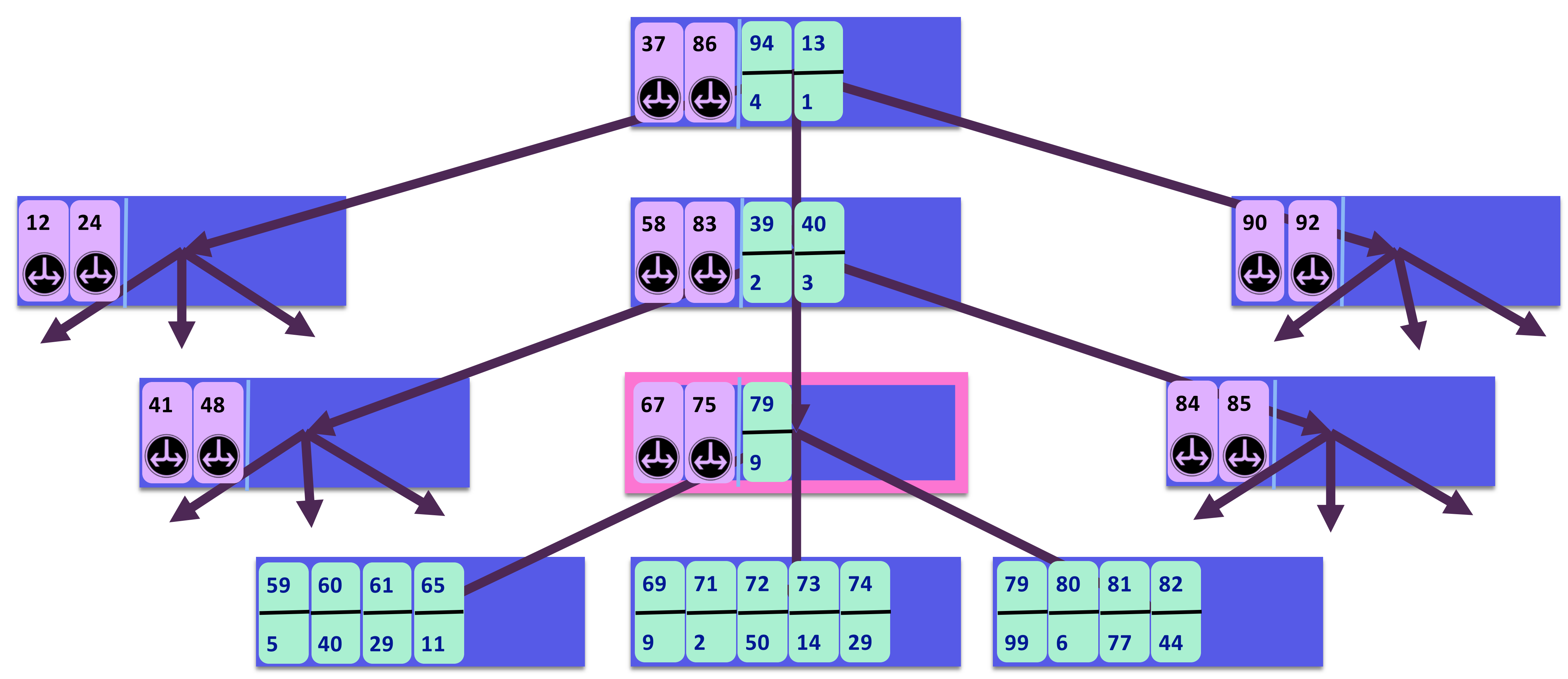
Query(71)

Lookups follow pivots, but check buffers along the way



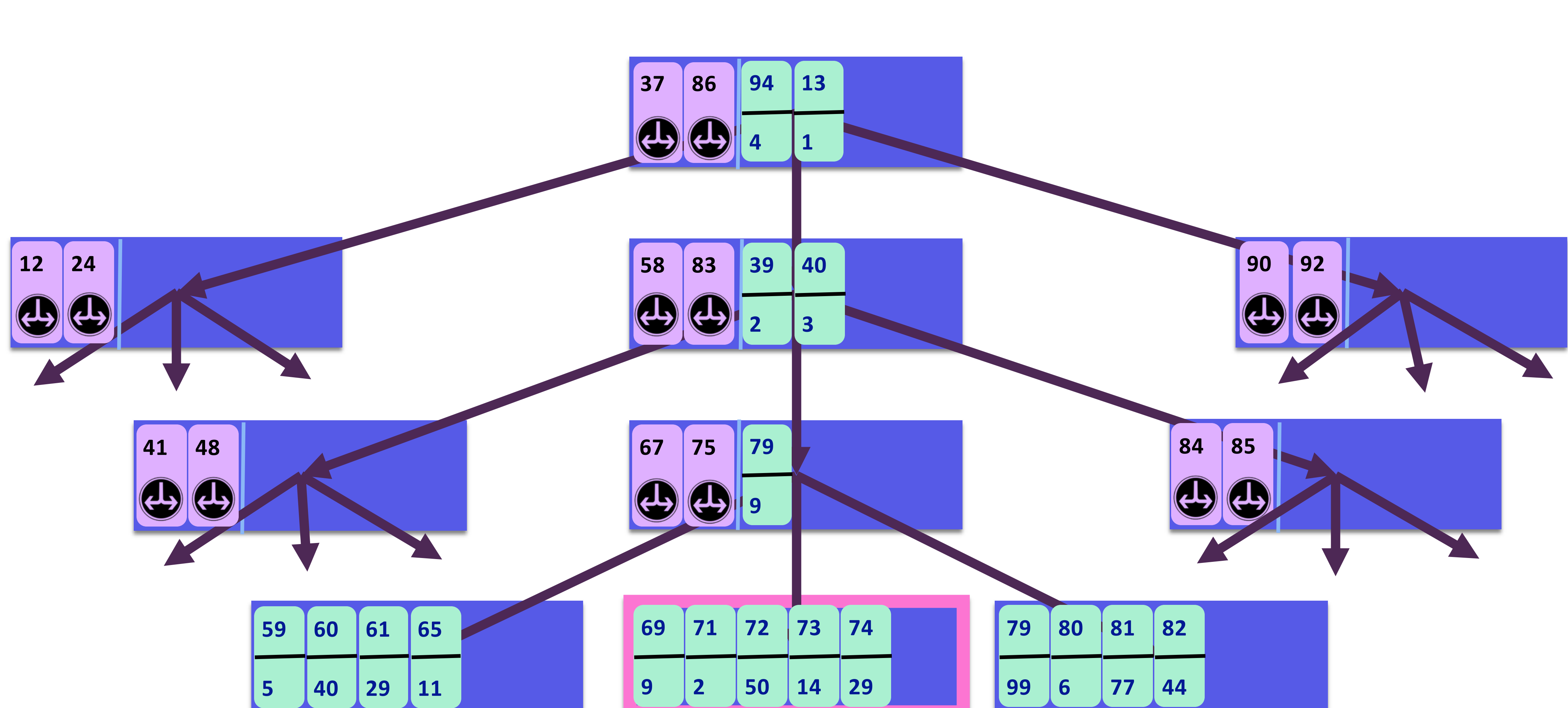
B^{ε}-Trees

Lookups follow pivots, but check buffers along the way



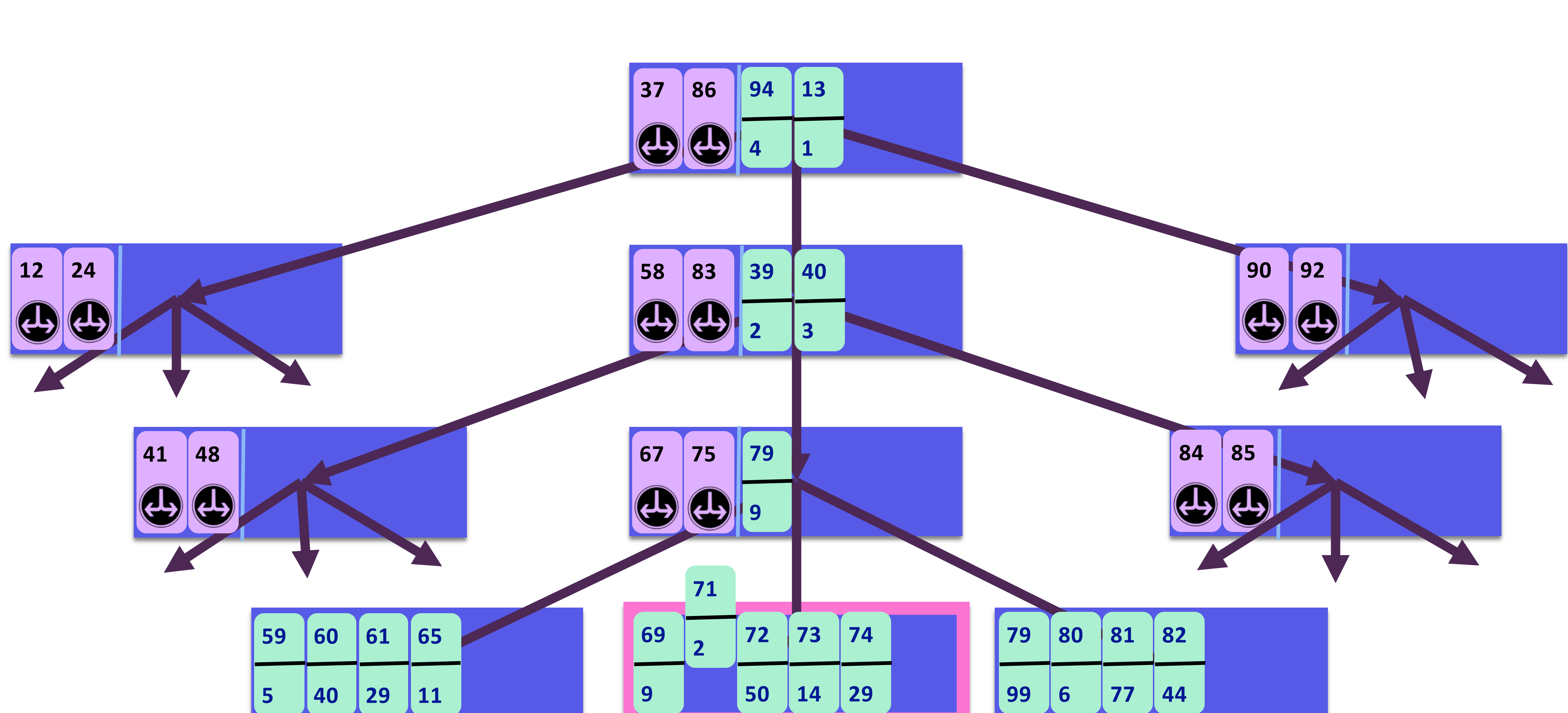
B$^{\varepsilon}$-Trees

Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Lookups follow pivots, but check buffers along the way



Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

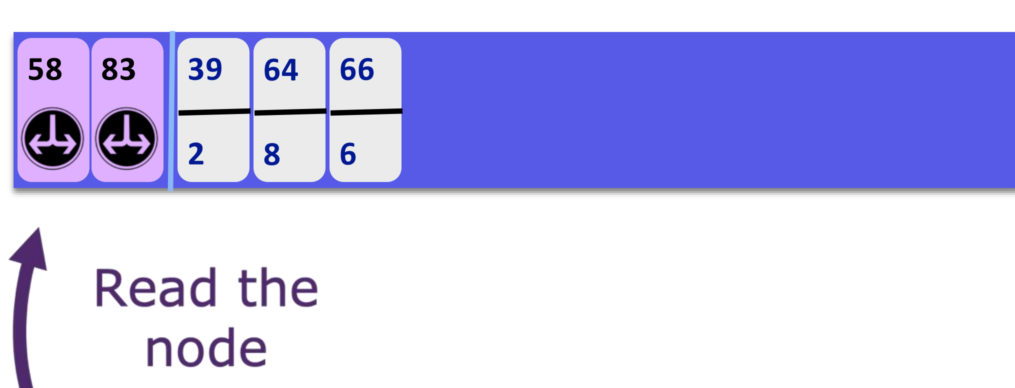
Insertions in B^ε-Trees are more expensive than they look

Recall: Insertions in B^ε-trees

65	72	80
11	50	6

Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

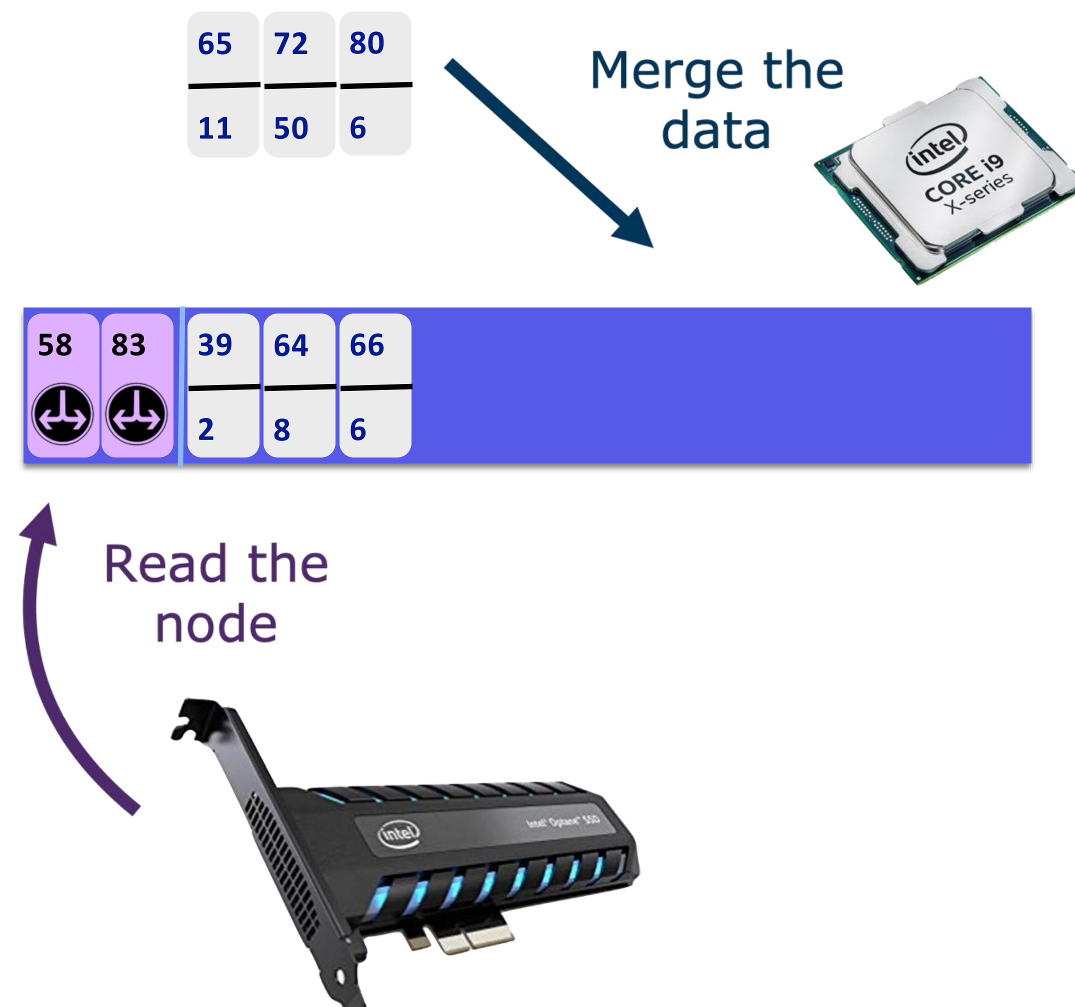
Recall: Insertions in B$^{\varepsilon}$-trees



Read the node

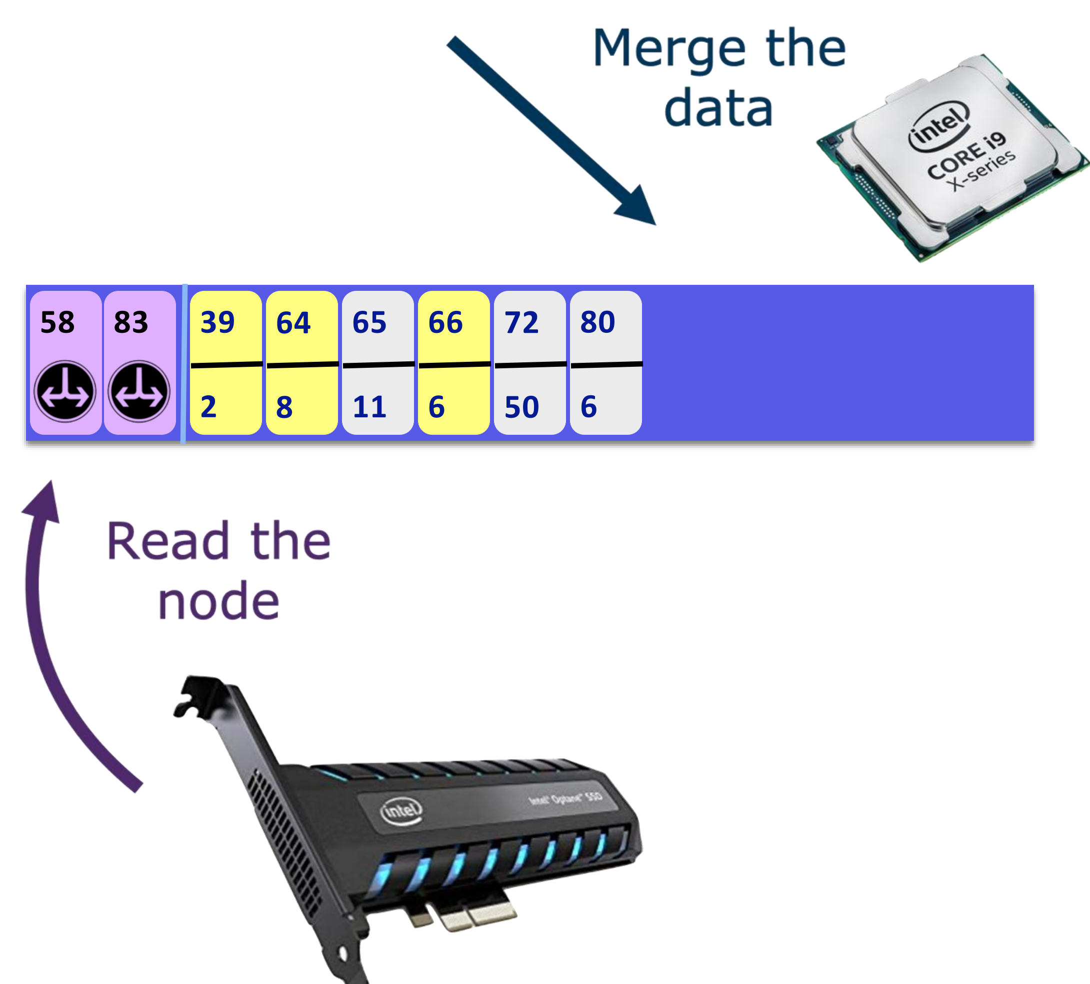
Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



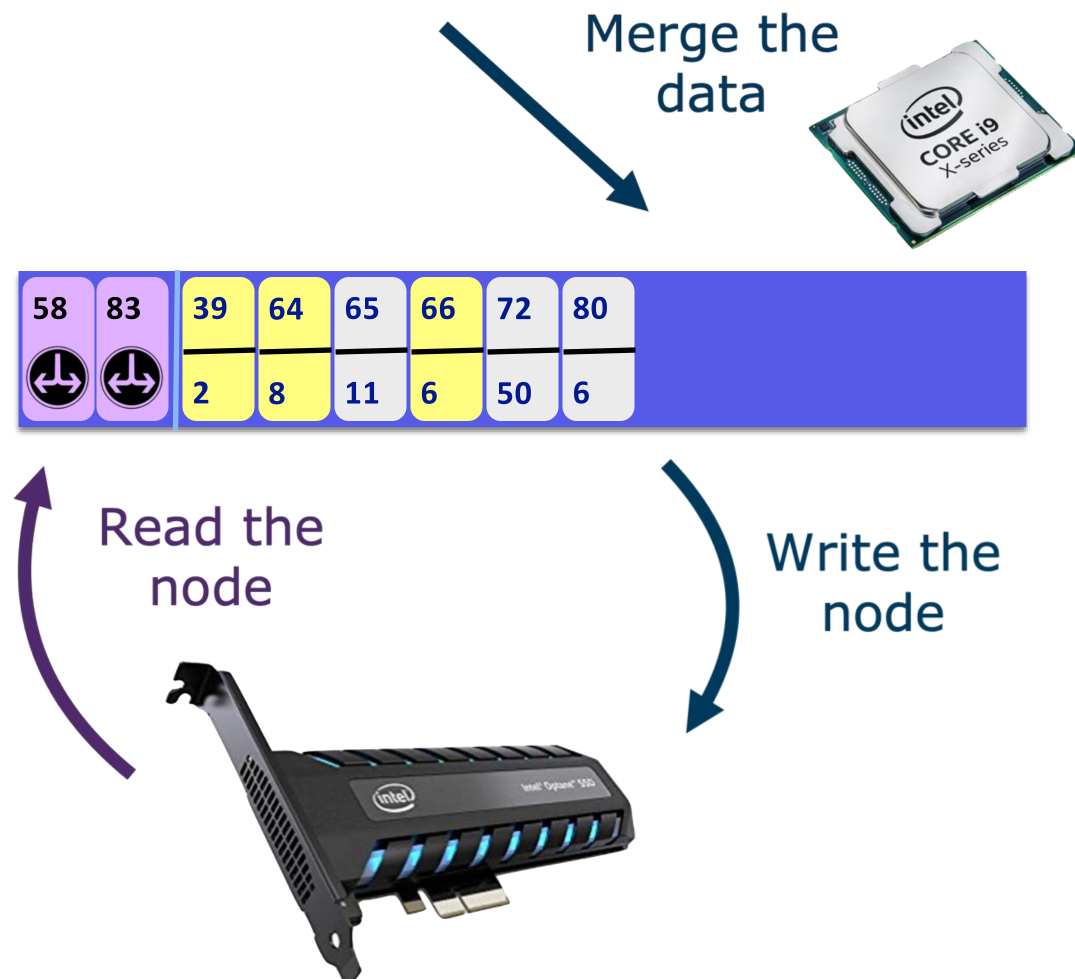
Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

Recall: Insertions in B$^{\varepsilon}$-trees



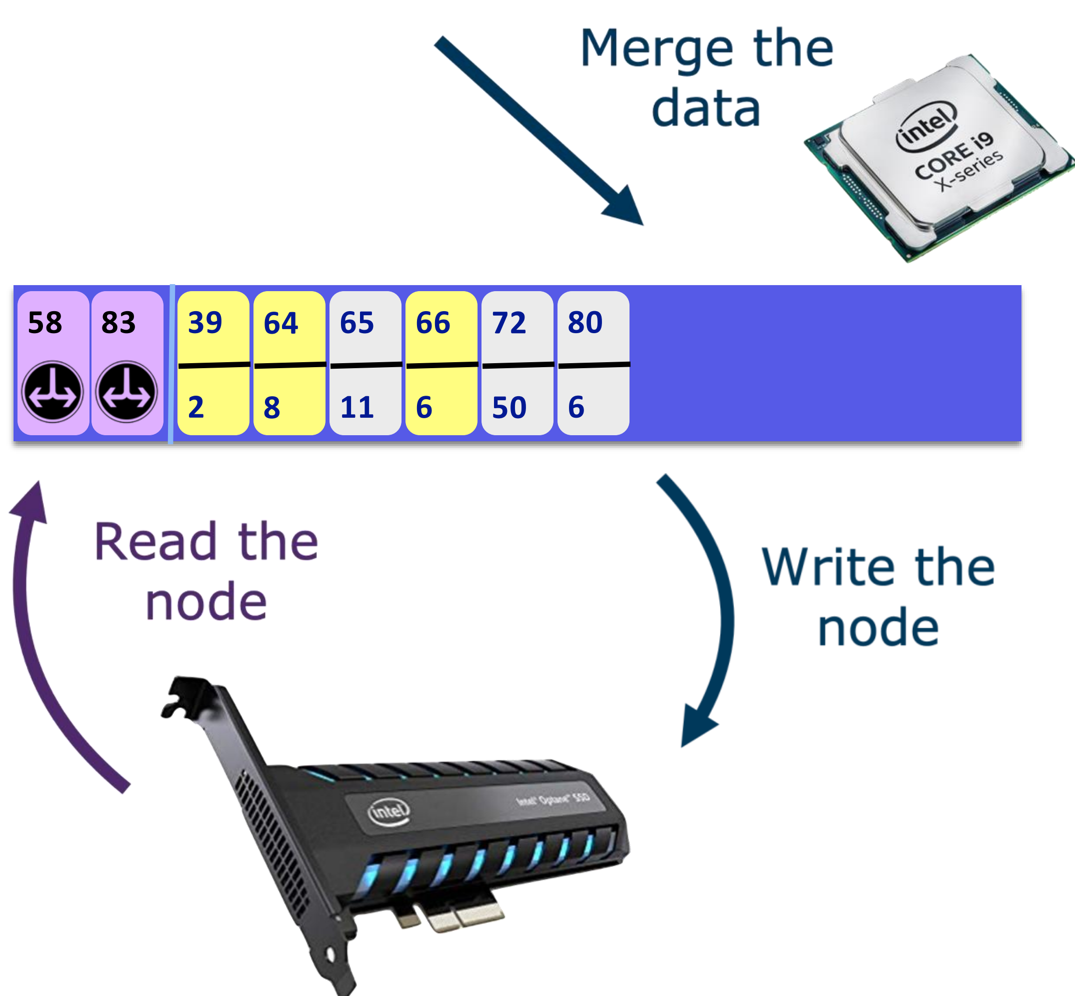
Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees

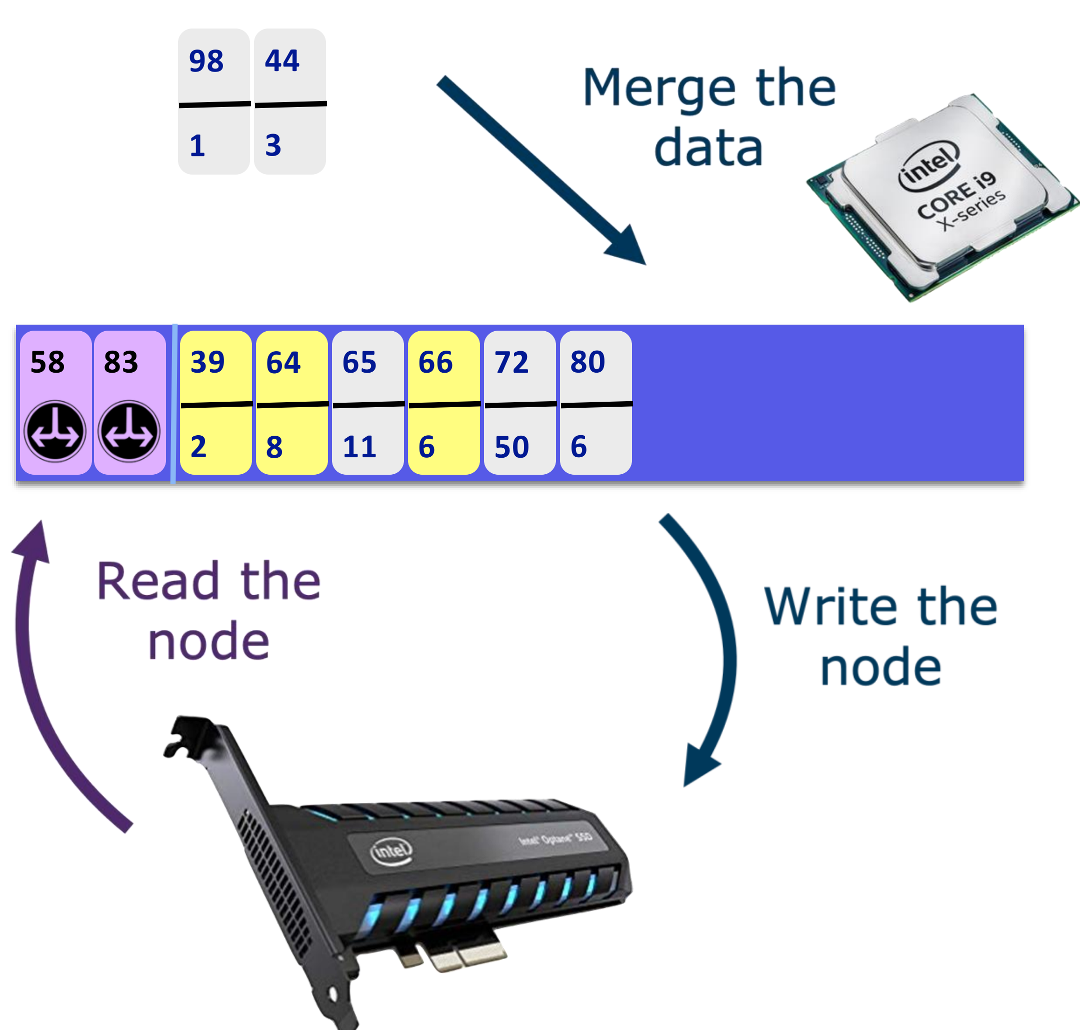


CPU Work = O(old + new)

Volume of IO = O(old + new)

Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

Recall: Insertions in B$^{\varepsilon}$-trees



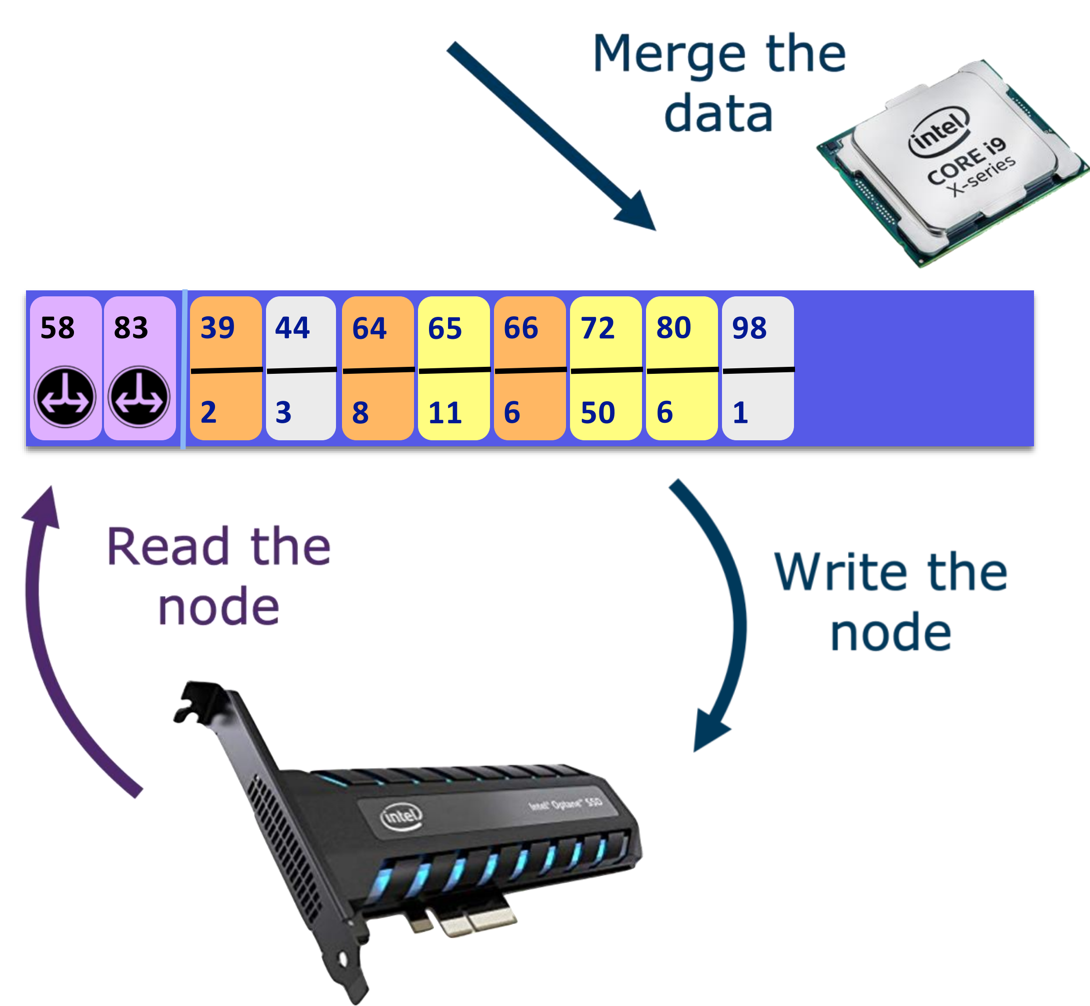
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



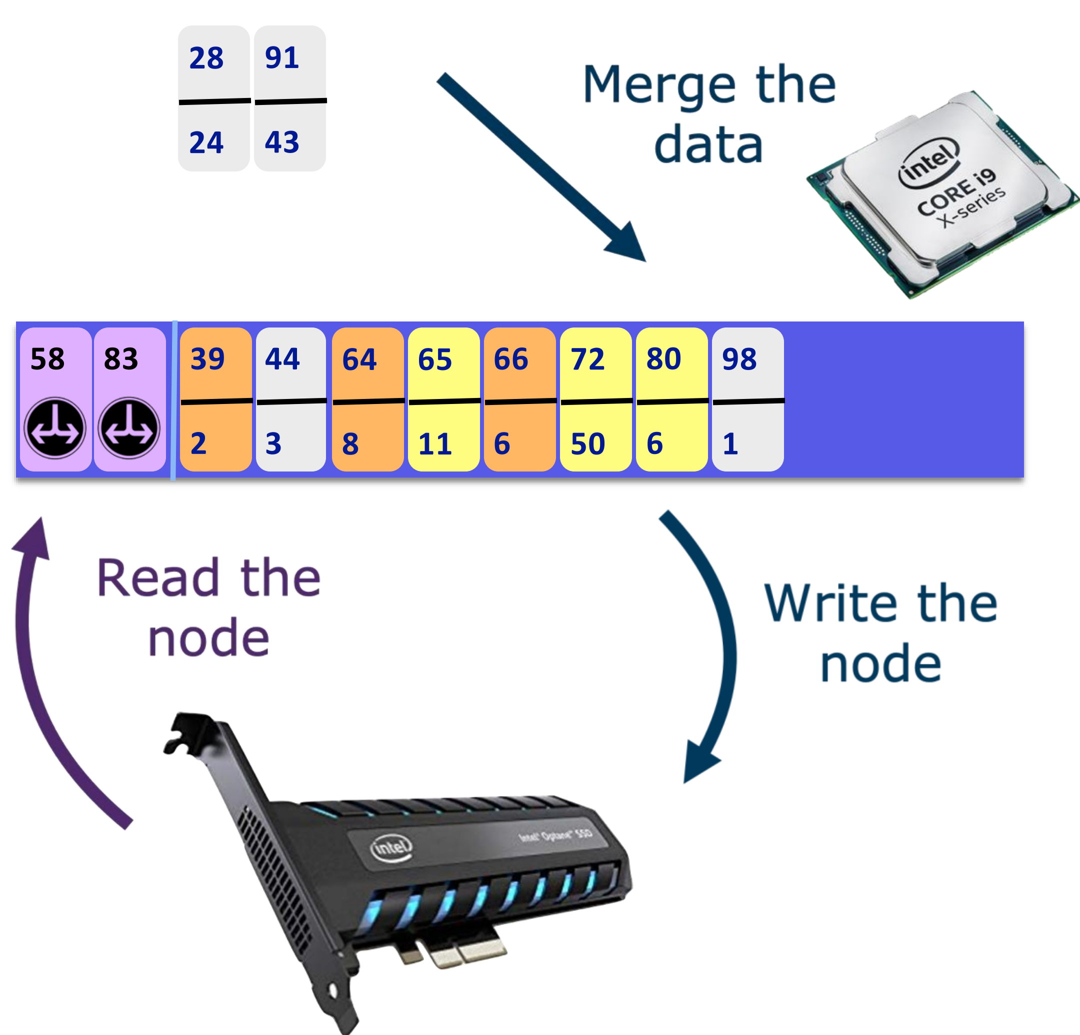
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



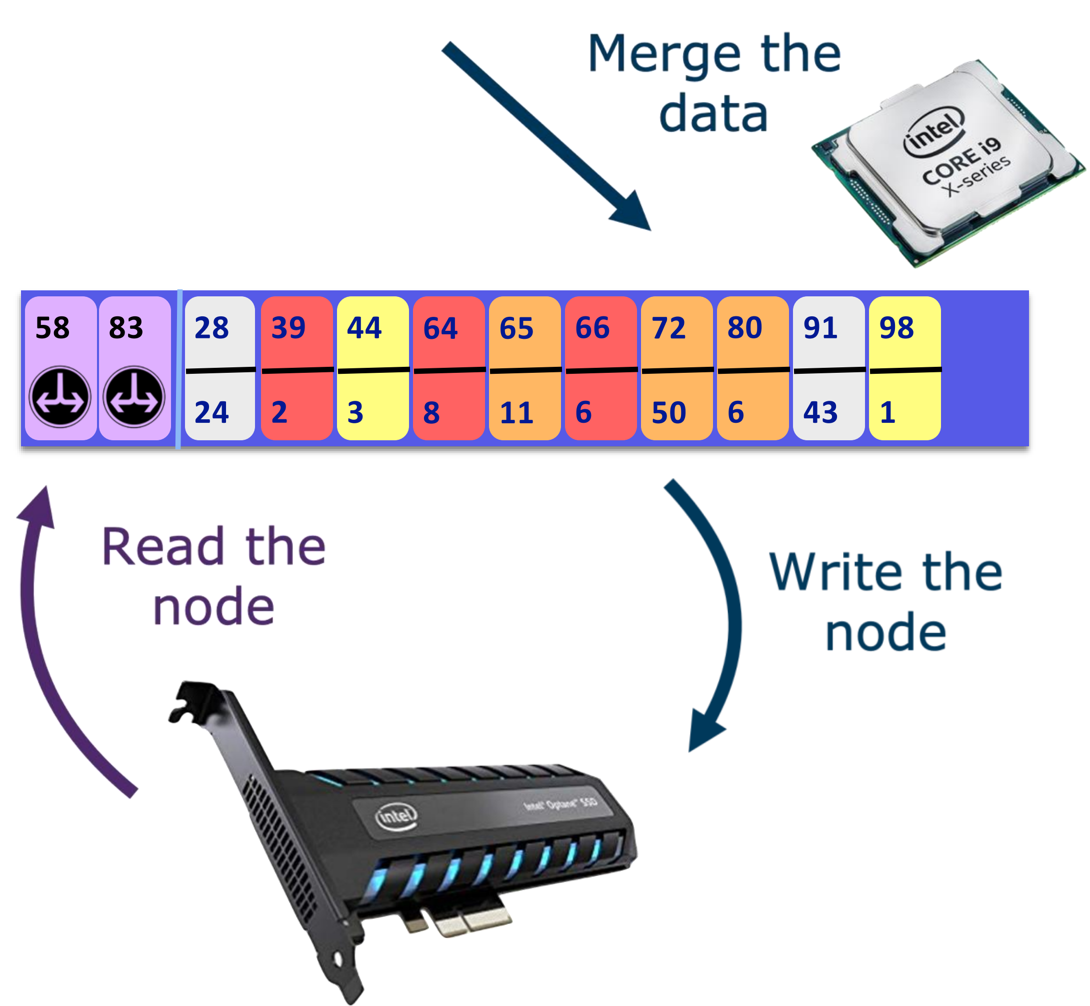
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



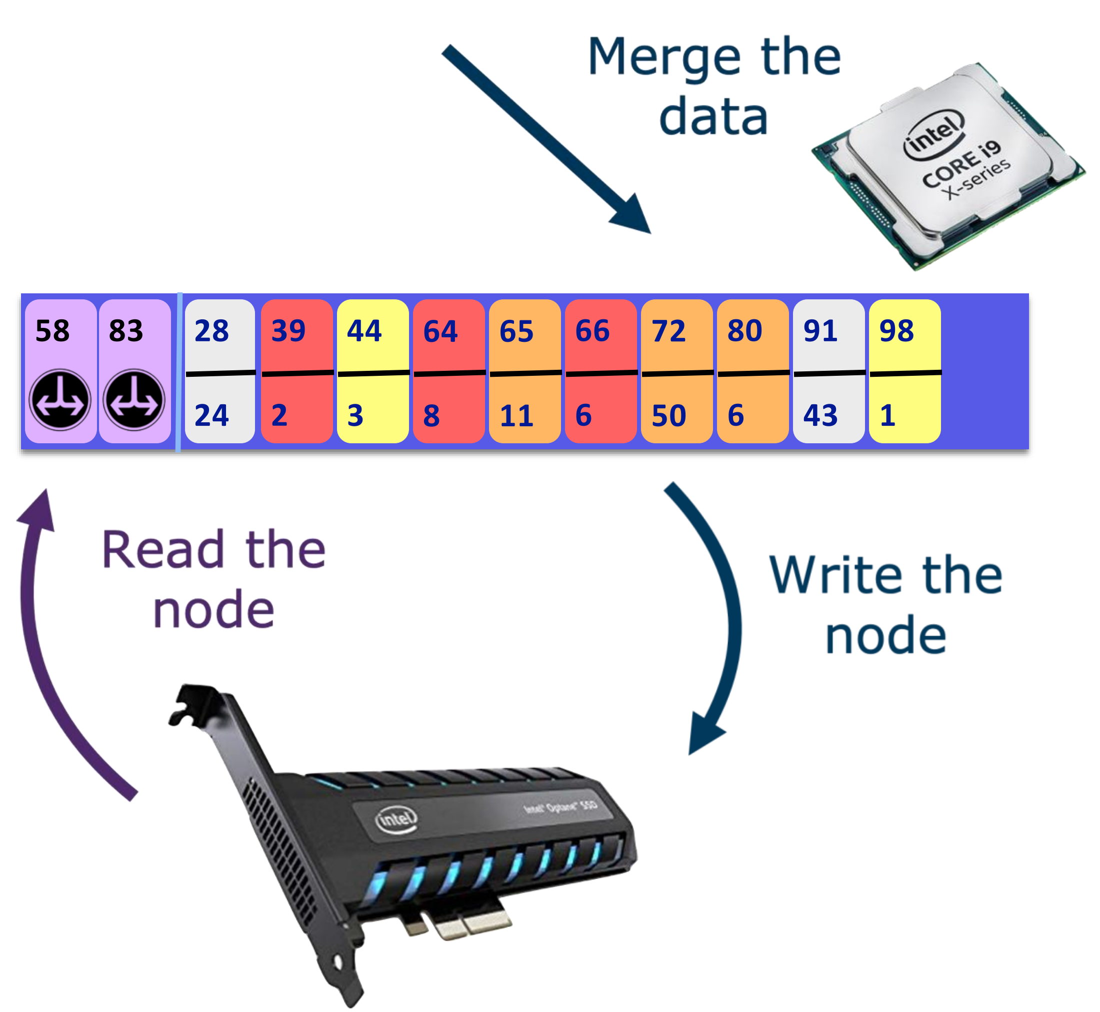
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Up to B^{ε} times per node!

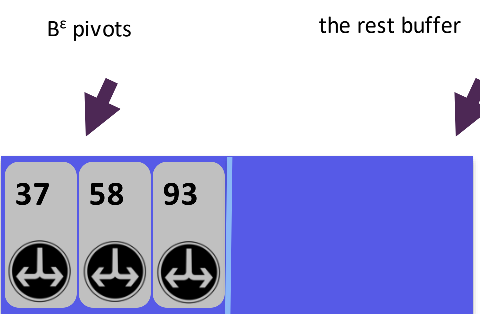
Size-Tiered B^{ε}-Trees

SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores
Conway, Gupta, Chidambaram, Farach-Colton, Spillane, Tai, Johnson, ATC 2020

Size-Tiered B^ε-Trees

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

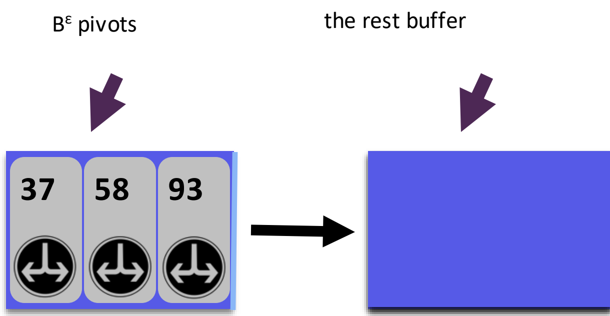
Recall:
a B^ε-tree node has pivots and a buffer



Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

Recall:
a B$^{\varepsilon}$-tree node has pivots and a buffer

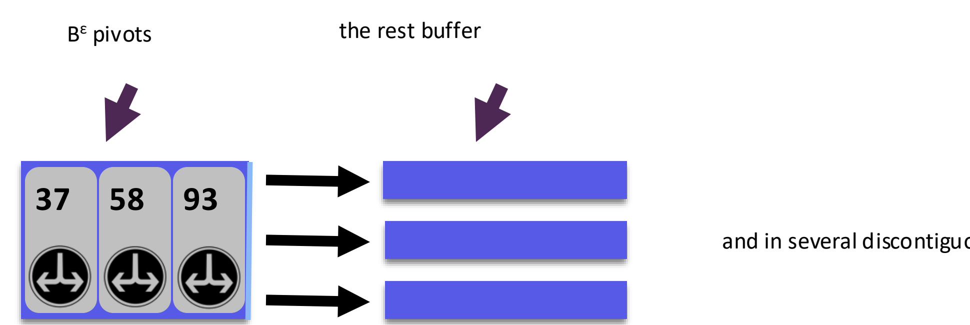


In an STB$^{\varepsilon}$-tree, the buffer is stored separately

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

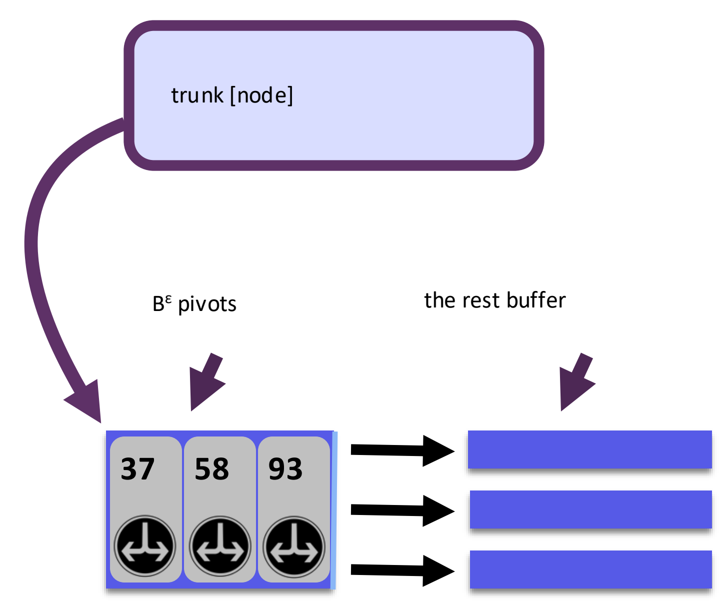


and in several discontiguous pieces

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

Size-Tiered B^{ε}-Trees

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



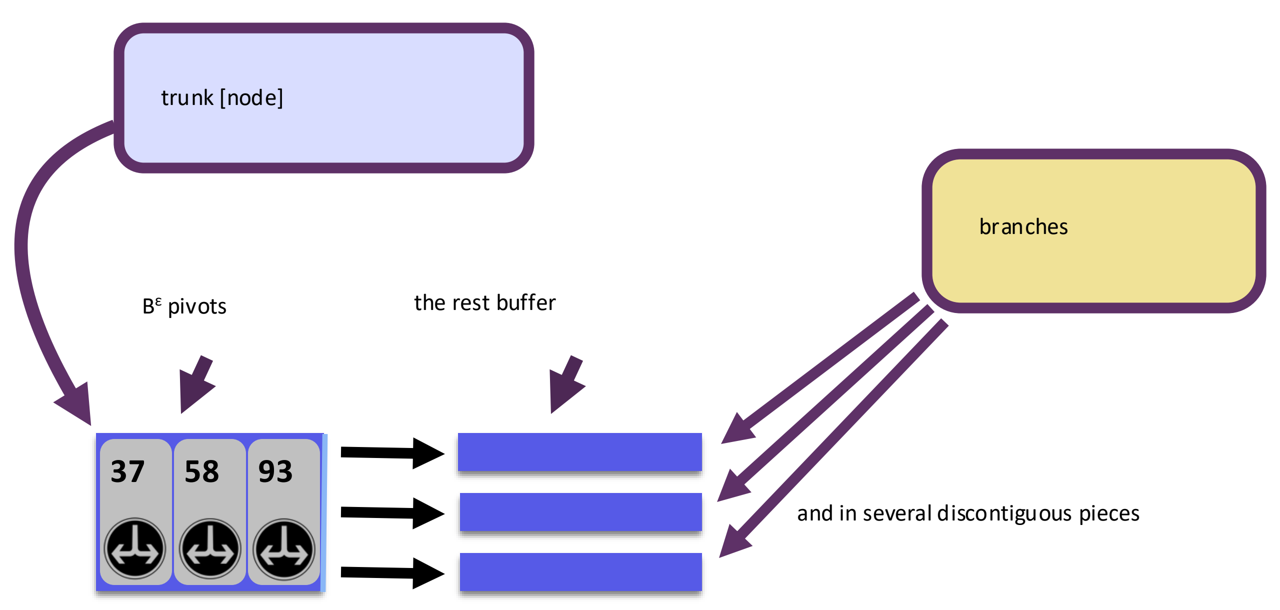
Recall:
a B^{ε}-tree node has pivots and a buffer

and in several discontiguous pieces

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

Size-Tiered 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

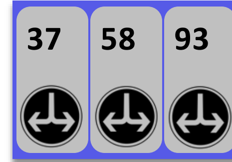
In an STB^ε-tree, the buffer is stored separately

Insertions in Size-Tiered B^{ε}-Trees

Size-Tiered B^{ε}-Trees

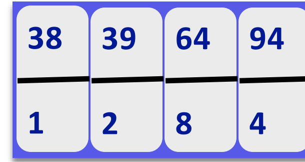
A Size-Tiered B^{ε}-tree is a B^{ε}-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node…



Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously



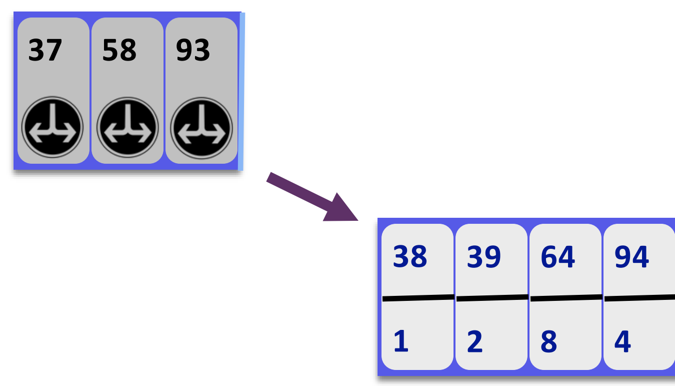
When new data is flushed into the trunk node…

Size-Tiered B^{ε}-Trees

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

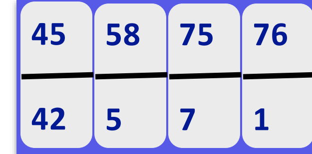
When new data is flushed into the trunk node…

…it is added as a new branch



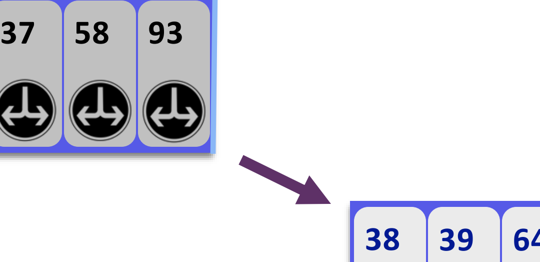
Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously



When new data is flushed into the trunk node…

…it is added as a new branch



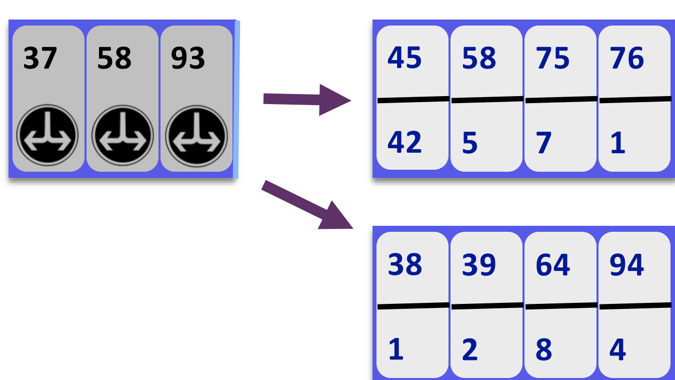
Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-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



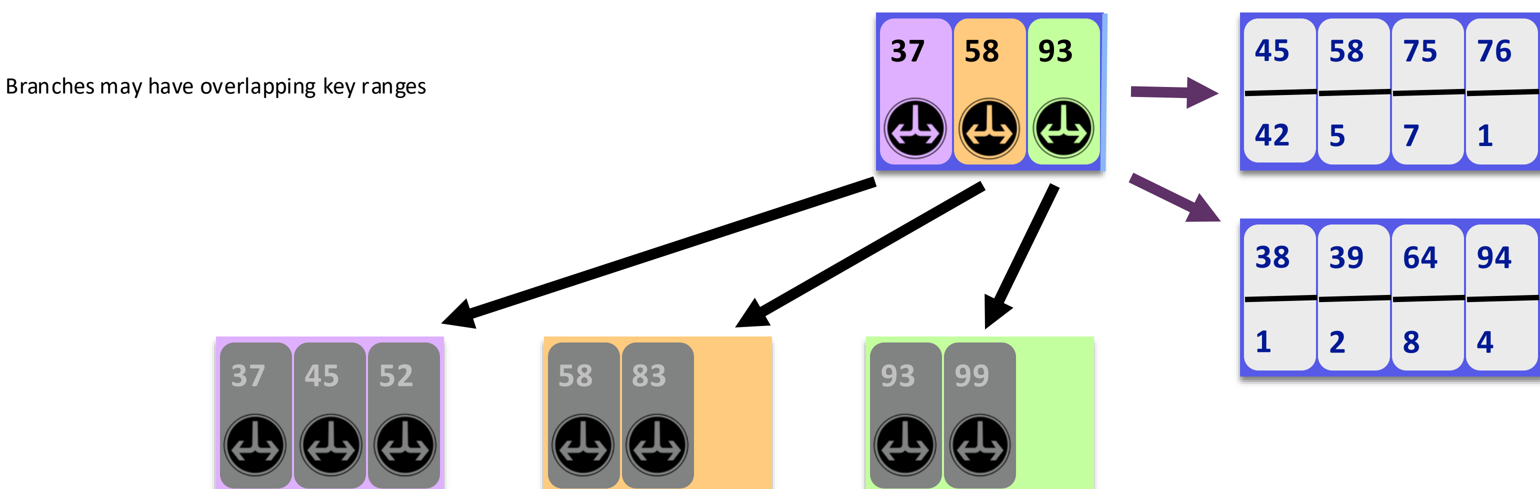
Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-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 B$^\varepsilon$-Trees

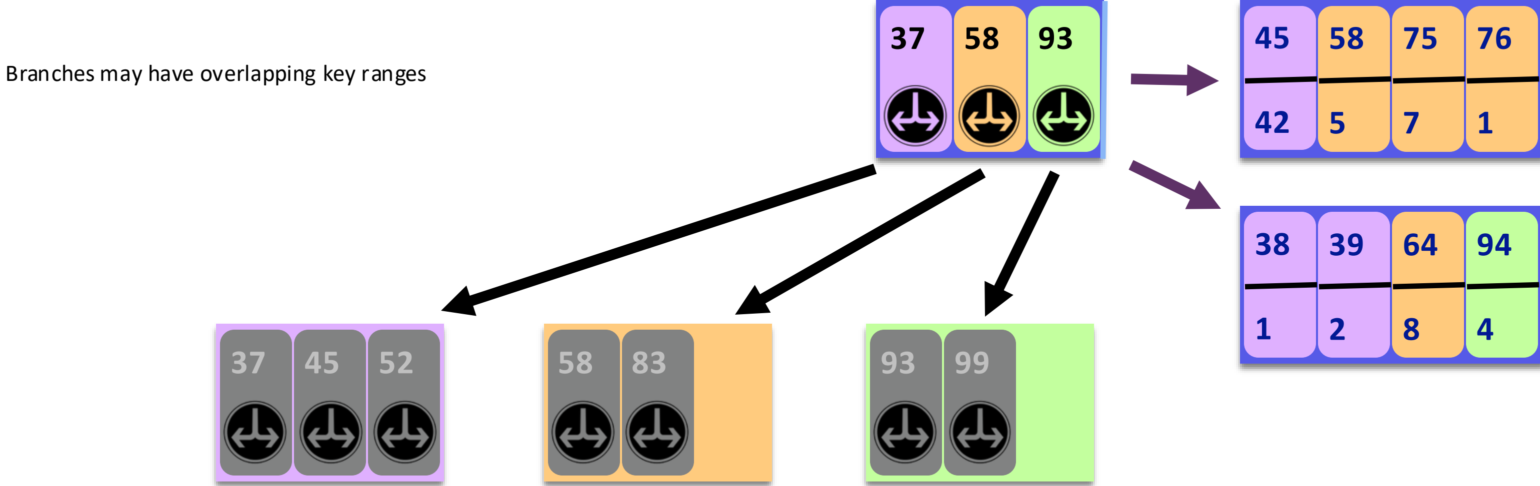
A Size-Tiered B$^\varepsilon$-tree is a B$^\varepsilon$-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node…

…it is added as a new branch

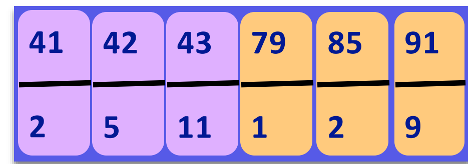
Branches may have overlapping key ranges

The old branches do not need to be rewritten



Size-Tiered B$^{\varepsilon}$-Trees

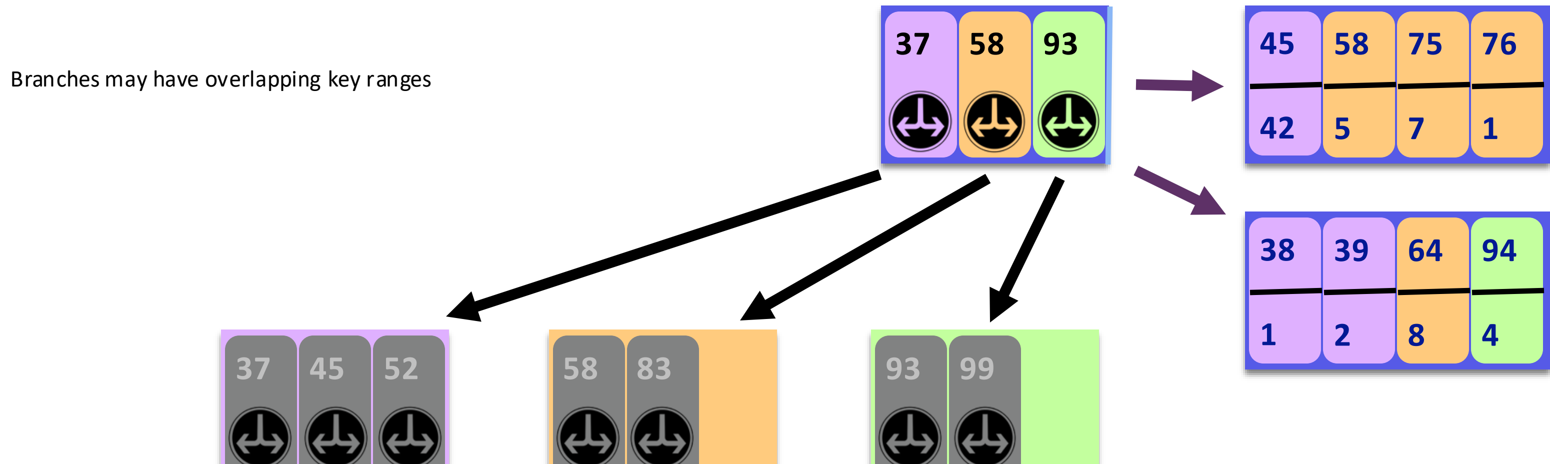
A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously



When new data is flushed into the trunk node…

…it is added as a new branch

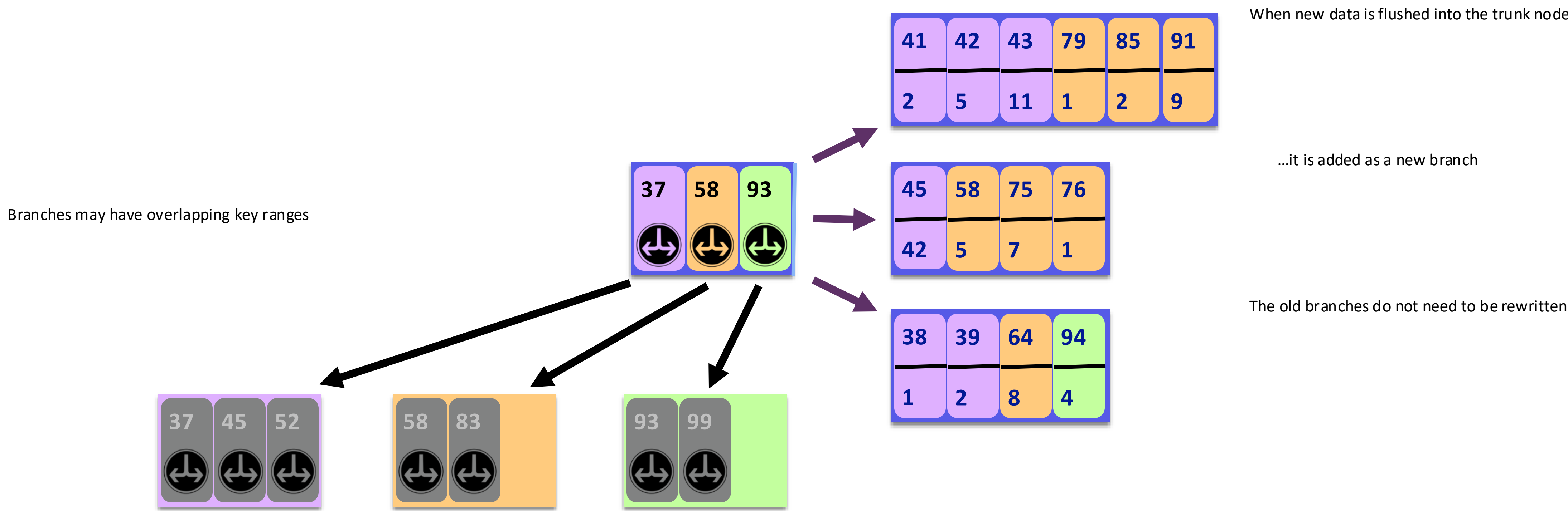
Branches may have overlapping key ranges



The old branches do not need to be rewritten

Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-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

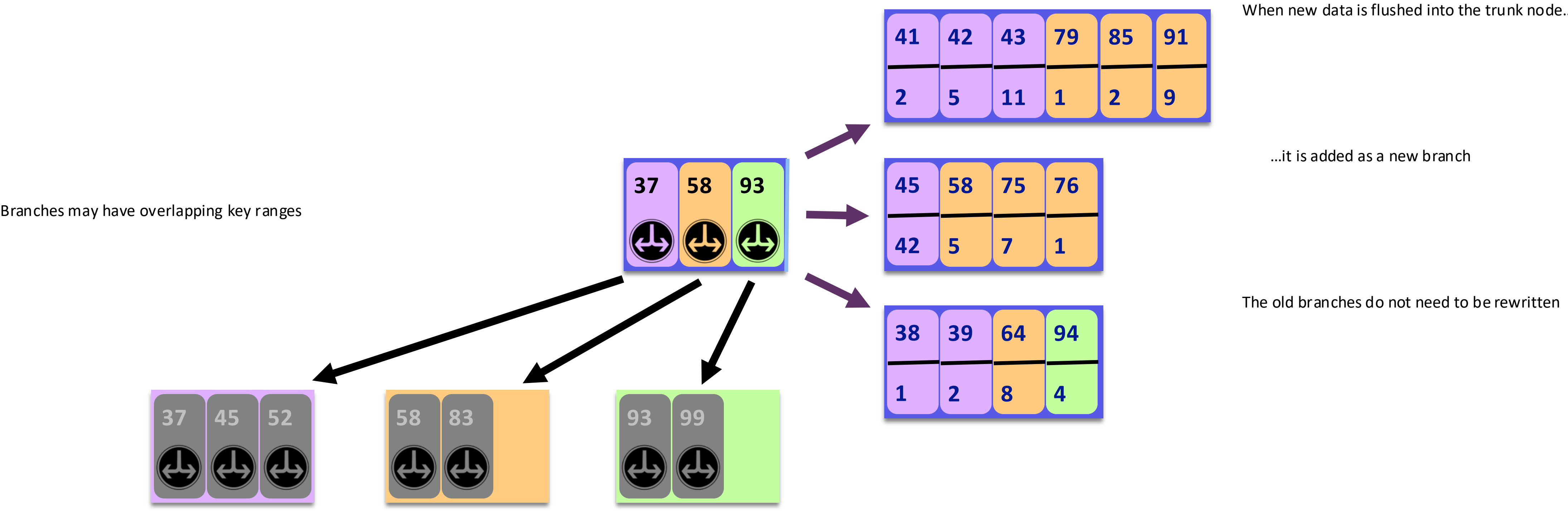
The old branches do not need to be rewritten

Size-Tiered B$^\varepsilon$-Trees

A Size-Tiered B$^\varepsilon$-tree is a B$^\varepsilon$-tree where the buffer is stored discontiguously

When the node is full:

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

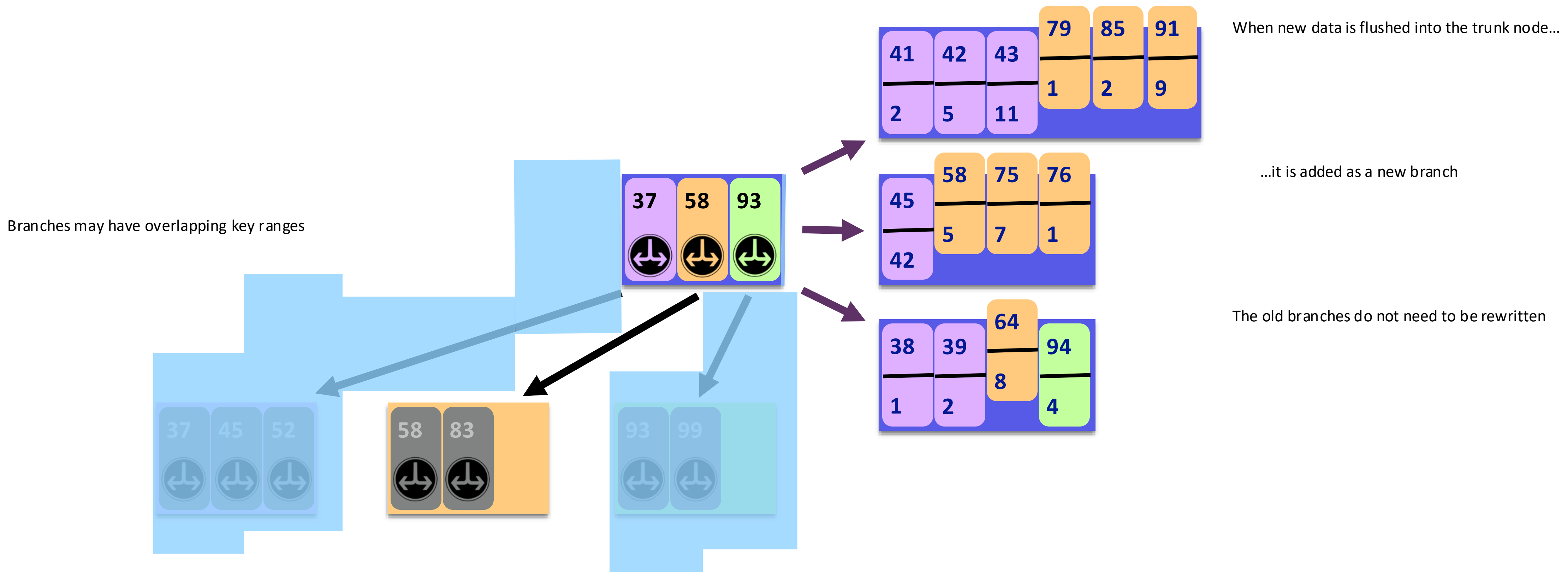


Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

When the node is full:

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



When new data is flushed into the trunk node…

…it is added as a new branch

Branches may have overlapping key ranges

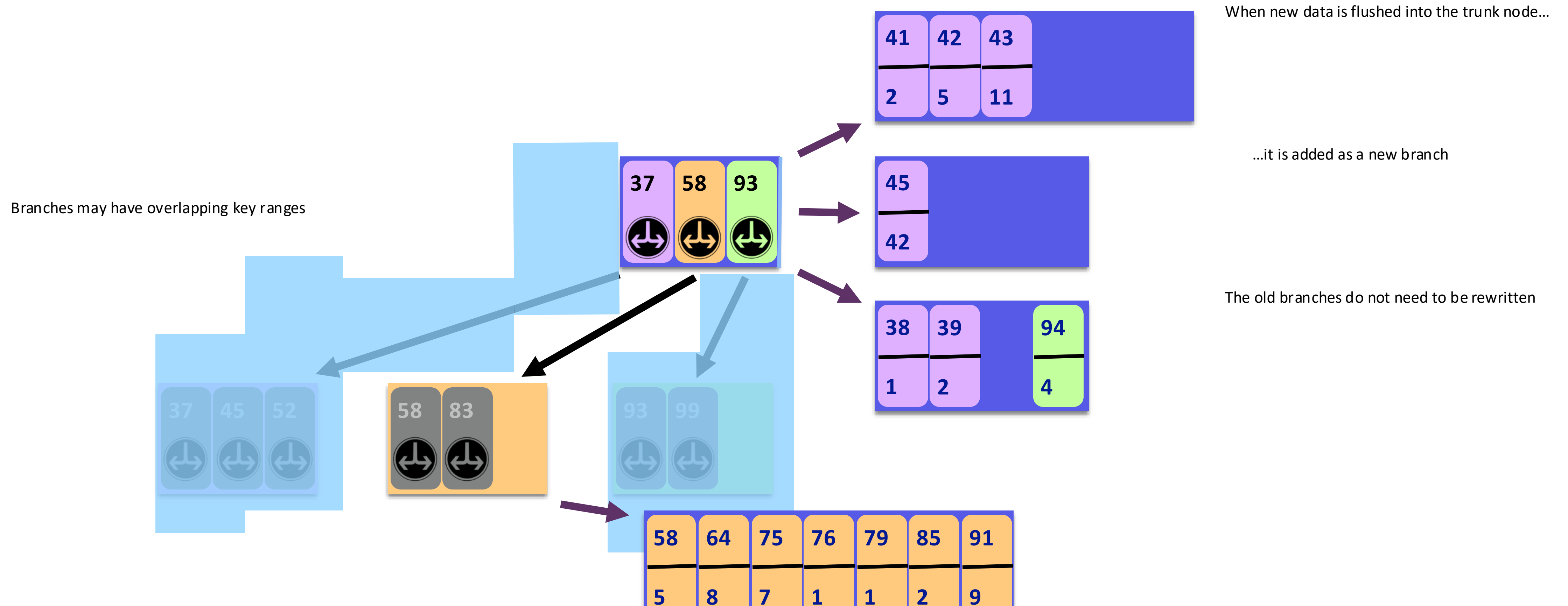
The old branches do not need to be rewritten

Size-Tiered B^{ε}-Trees

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

When the node is full:

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



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

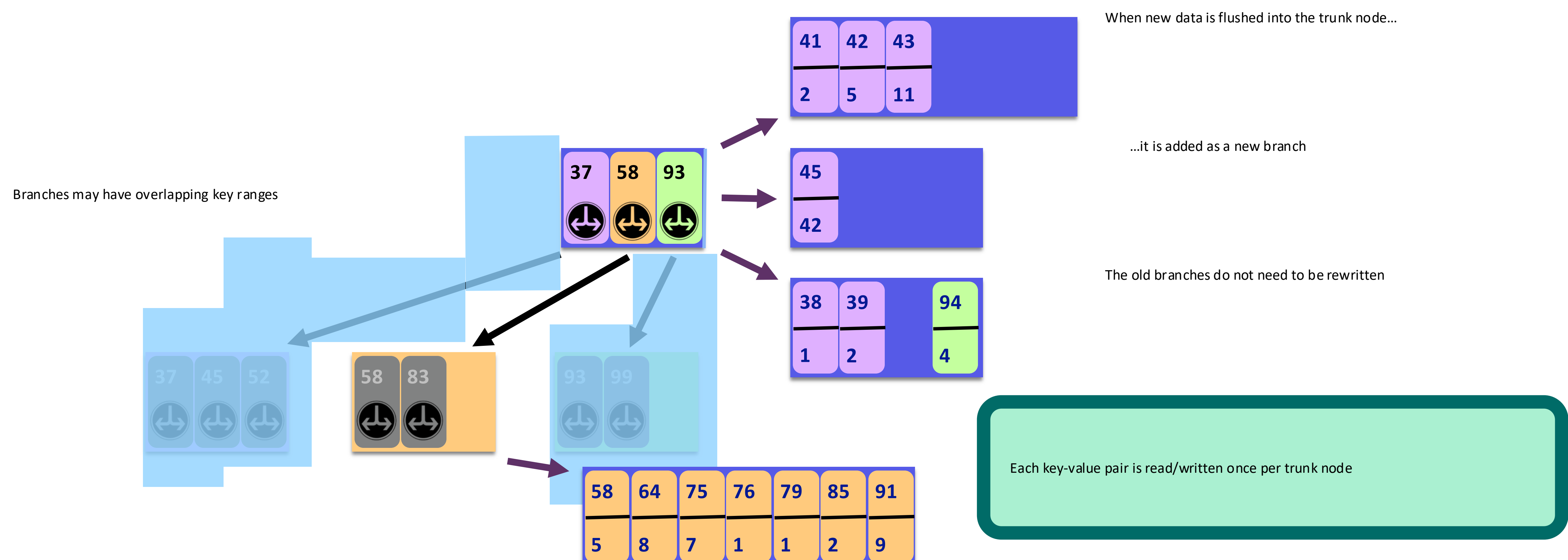
Branches may have overlapping key ranges

Size-Tiered B$^{\varepsilon}$-Trees

A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

When the node is full:

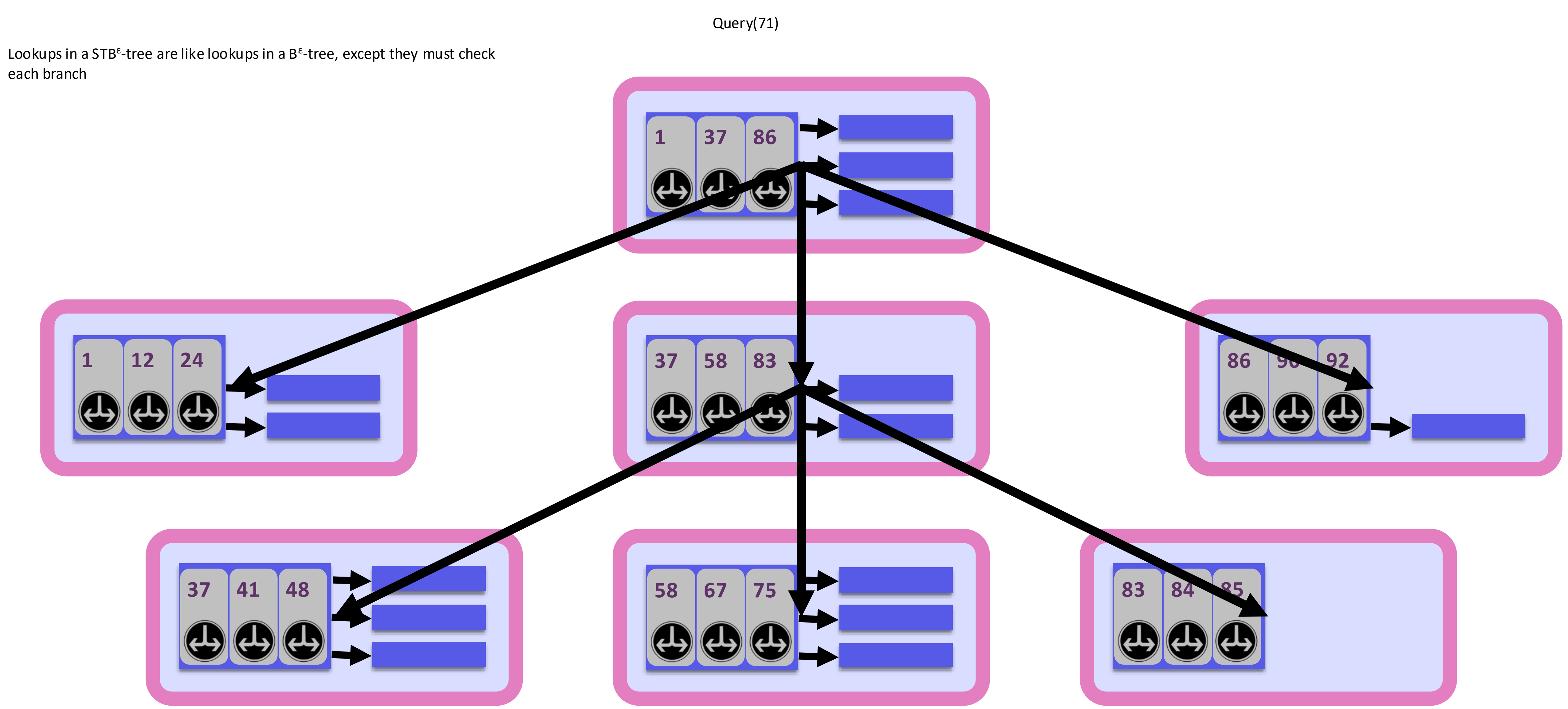
1. Pick child receiving most messages
2. Merge them into a new branch for the child



Lookups in Size-Tiered B^ε-Trees

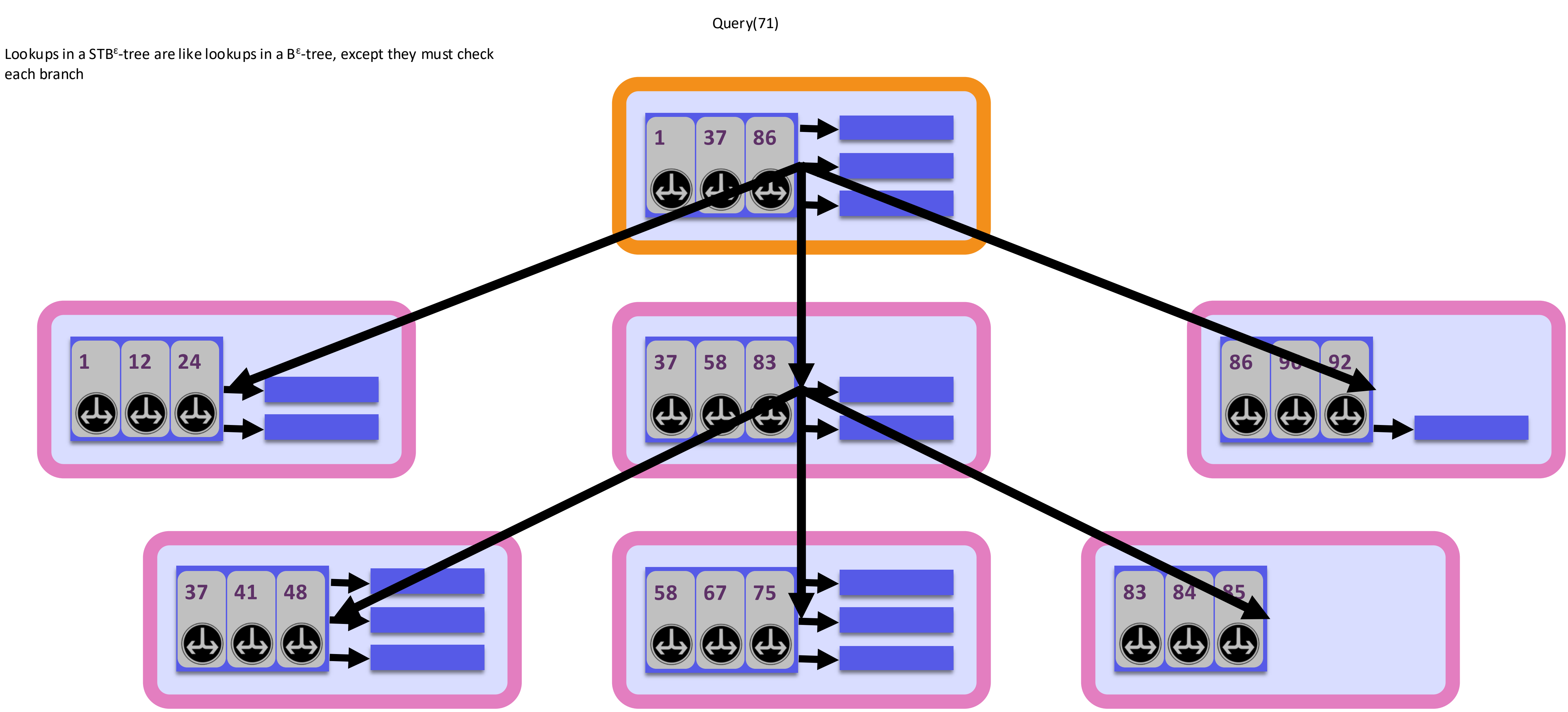
Size-Tiered B^{ε}-Trees

Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch



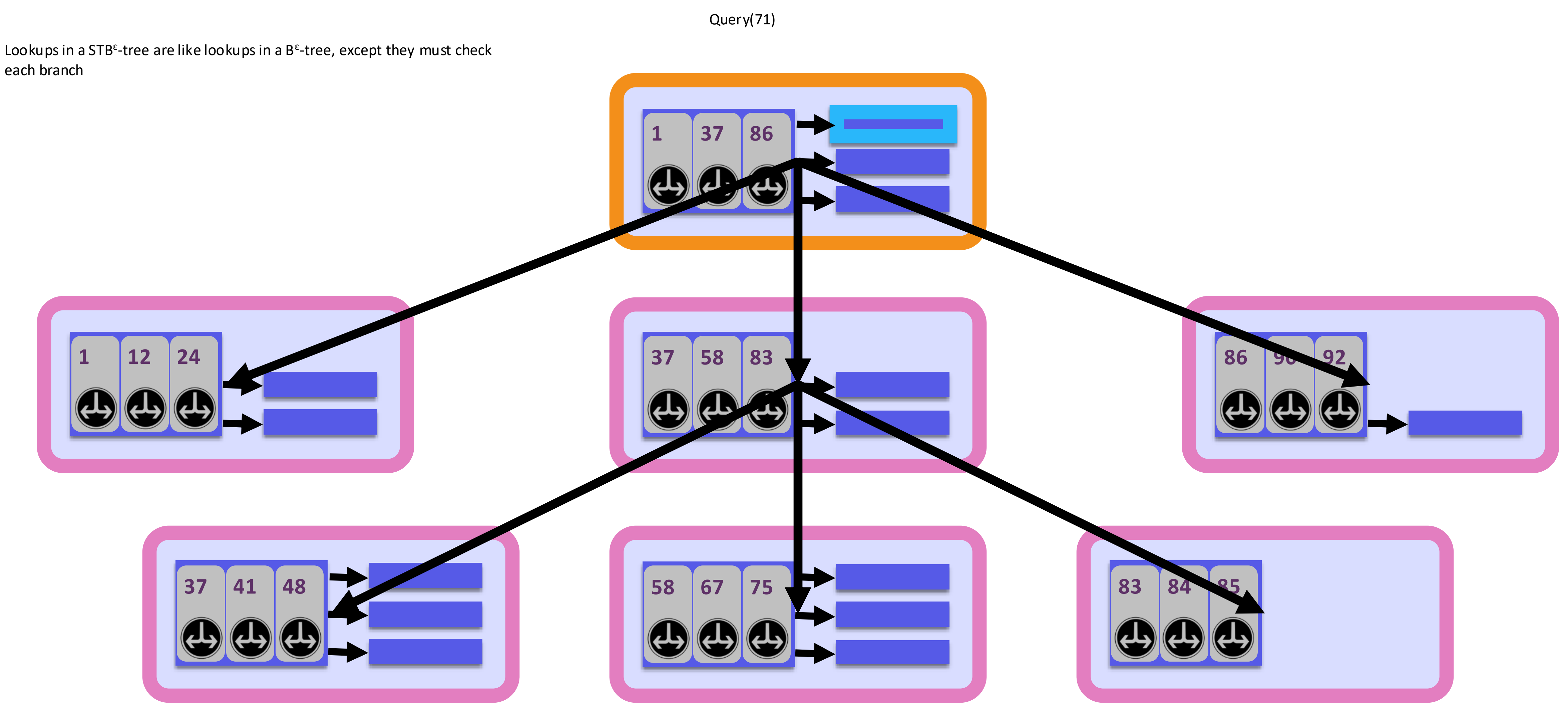
Size-Tiered B^{ε}-Trees

Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch



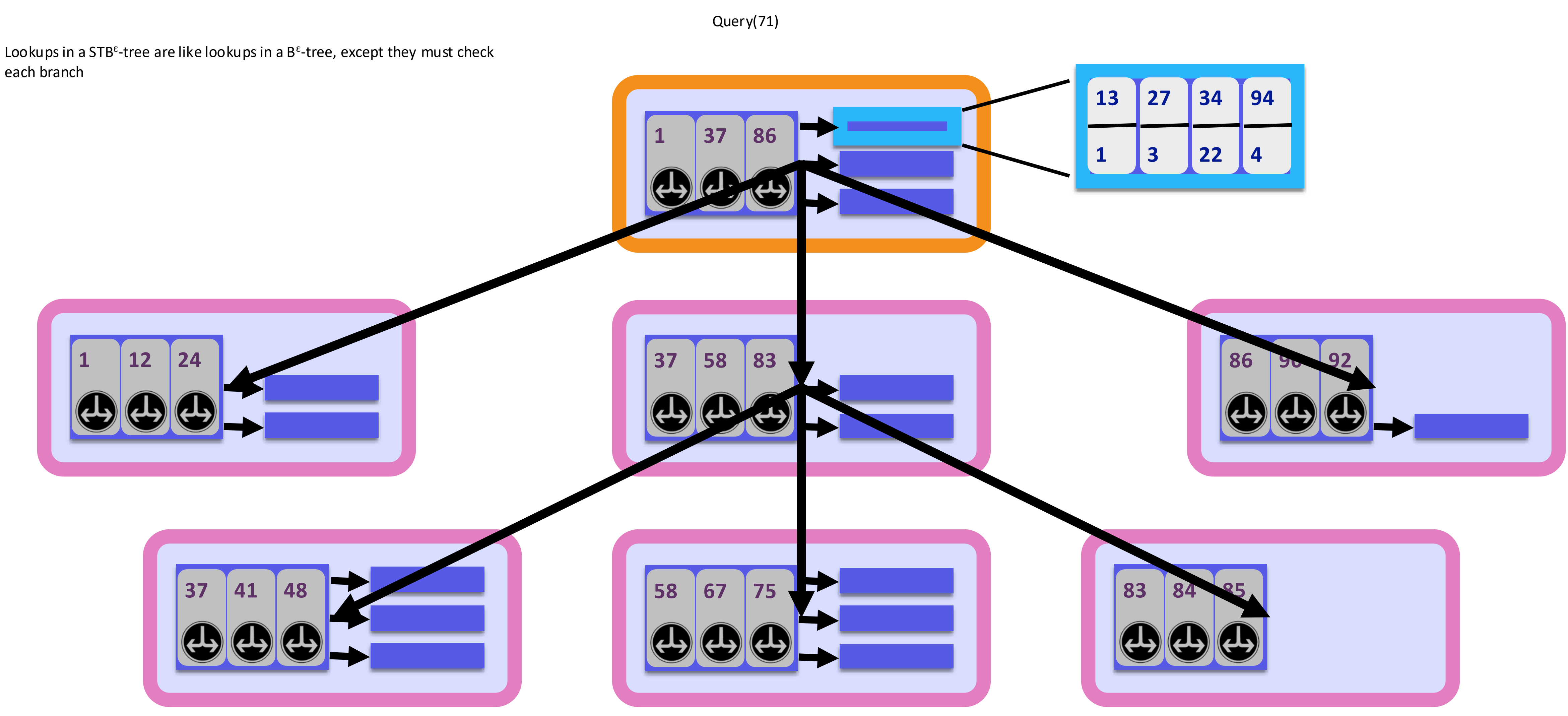
Size-Tiered B^{ε}-Trees

Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch



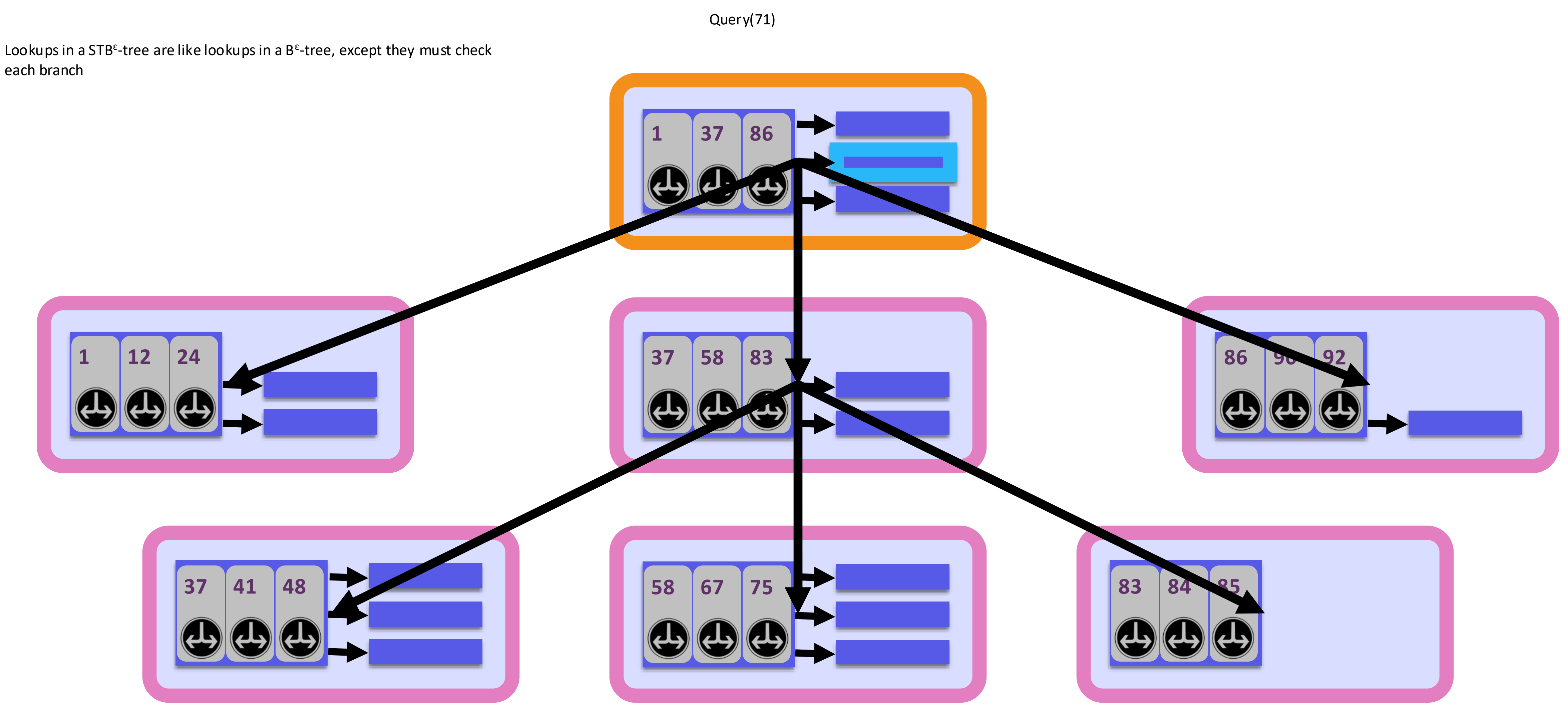
Size-Tiered B^{ε}-Trees

Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch



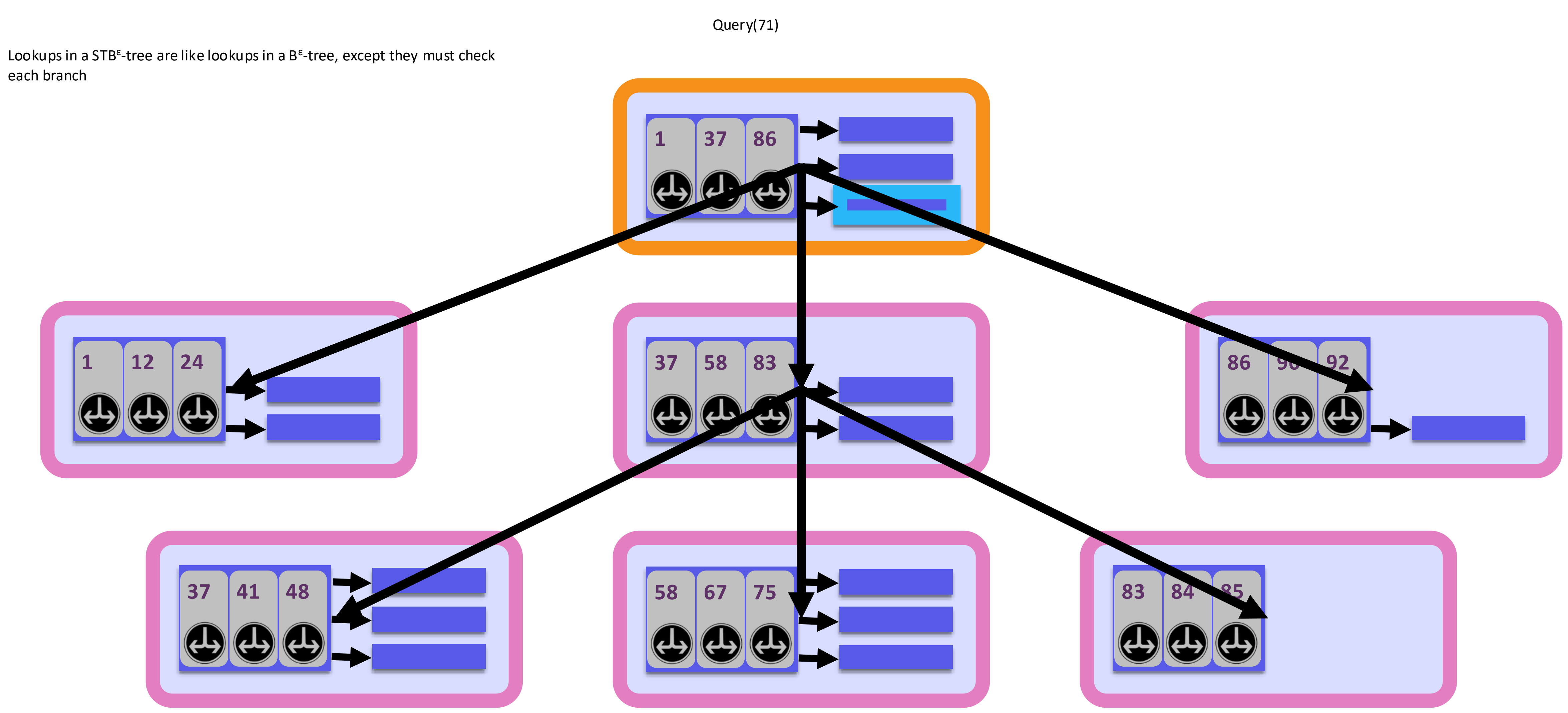
Size-Tiered B^{ε}-Trees

Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch



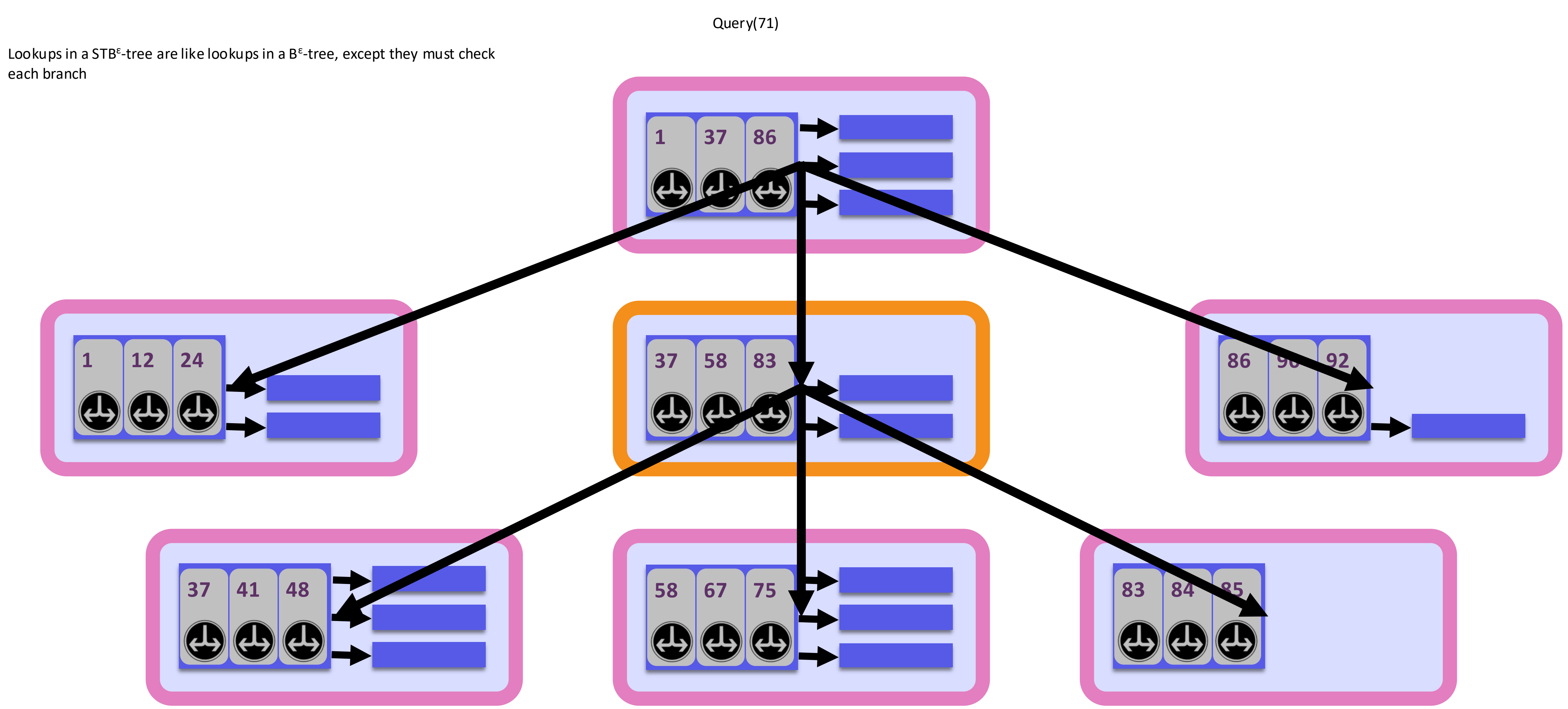
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



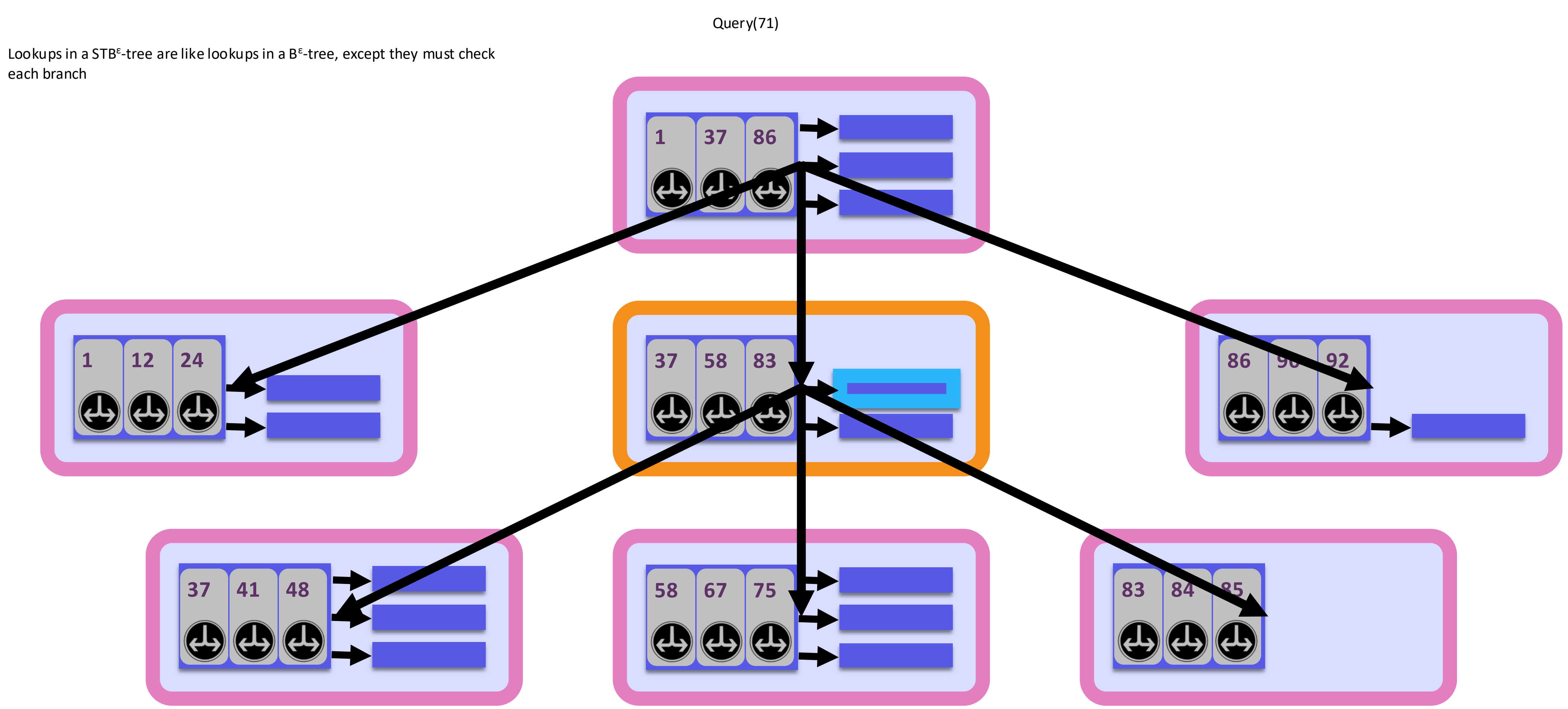
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



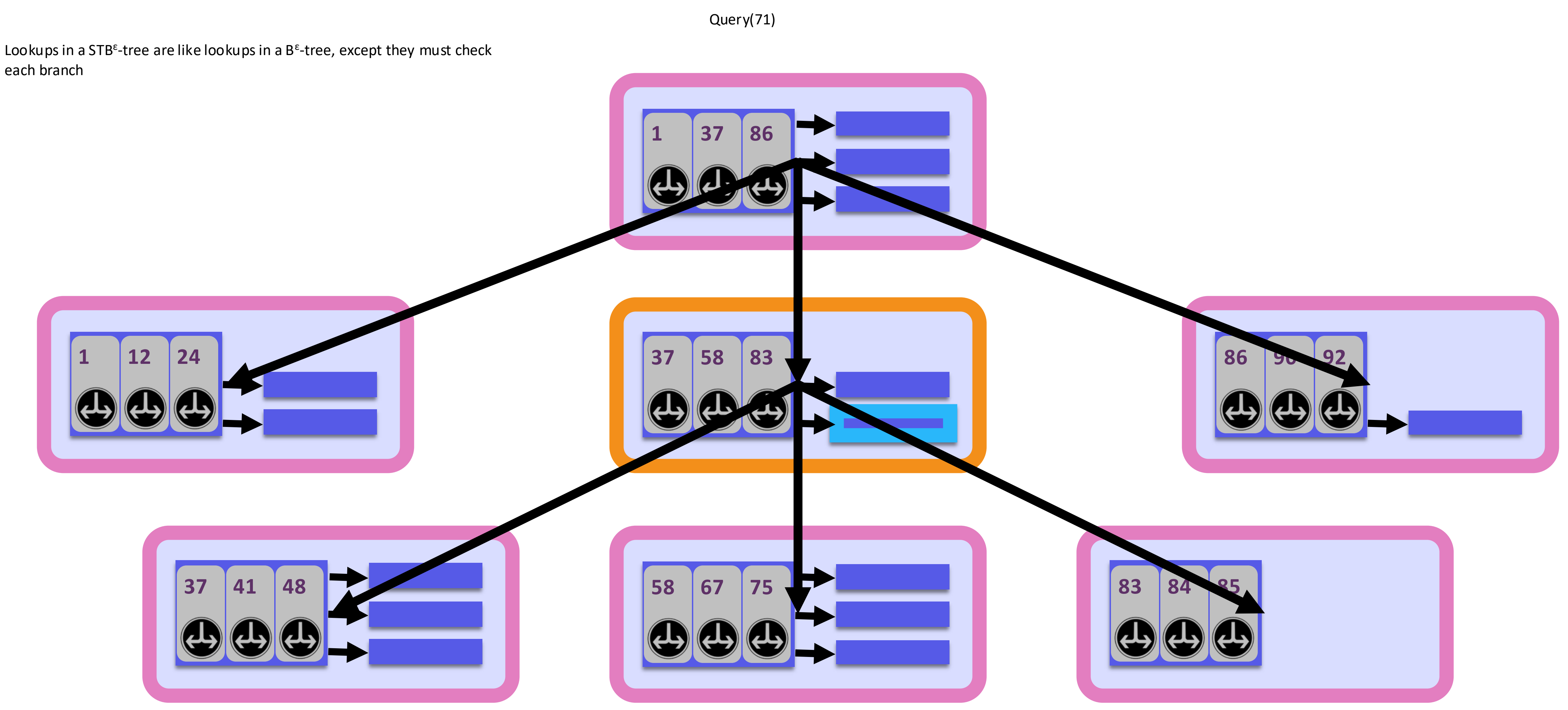
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



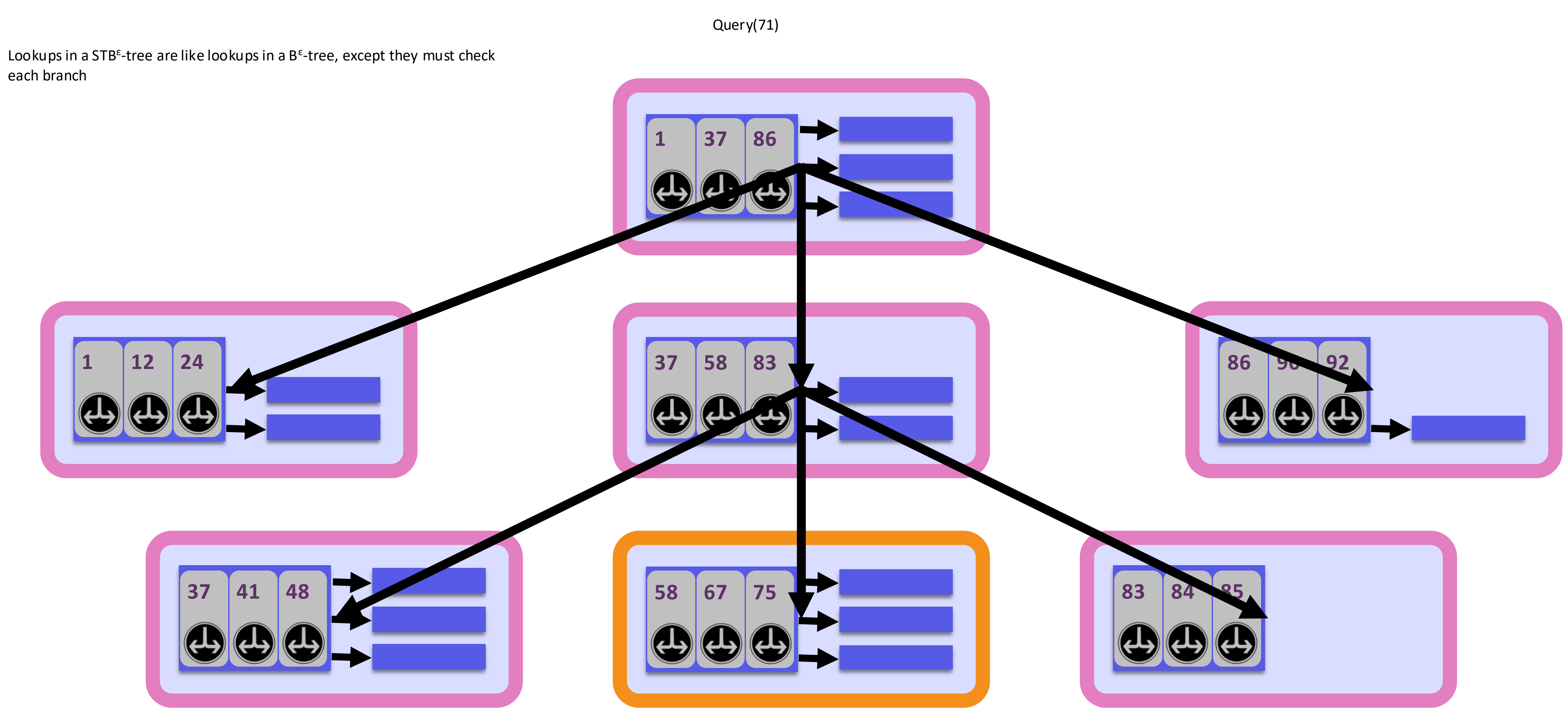
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



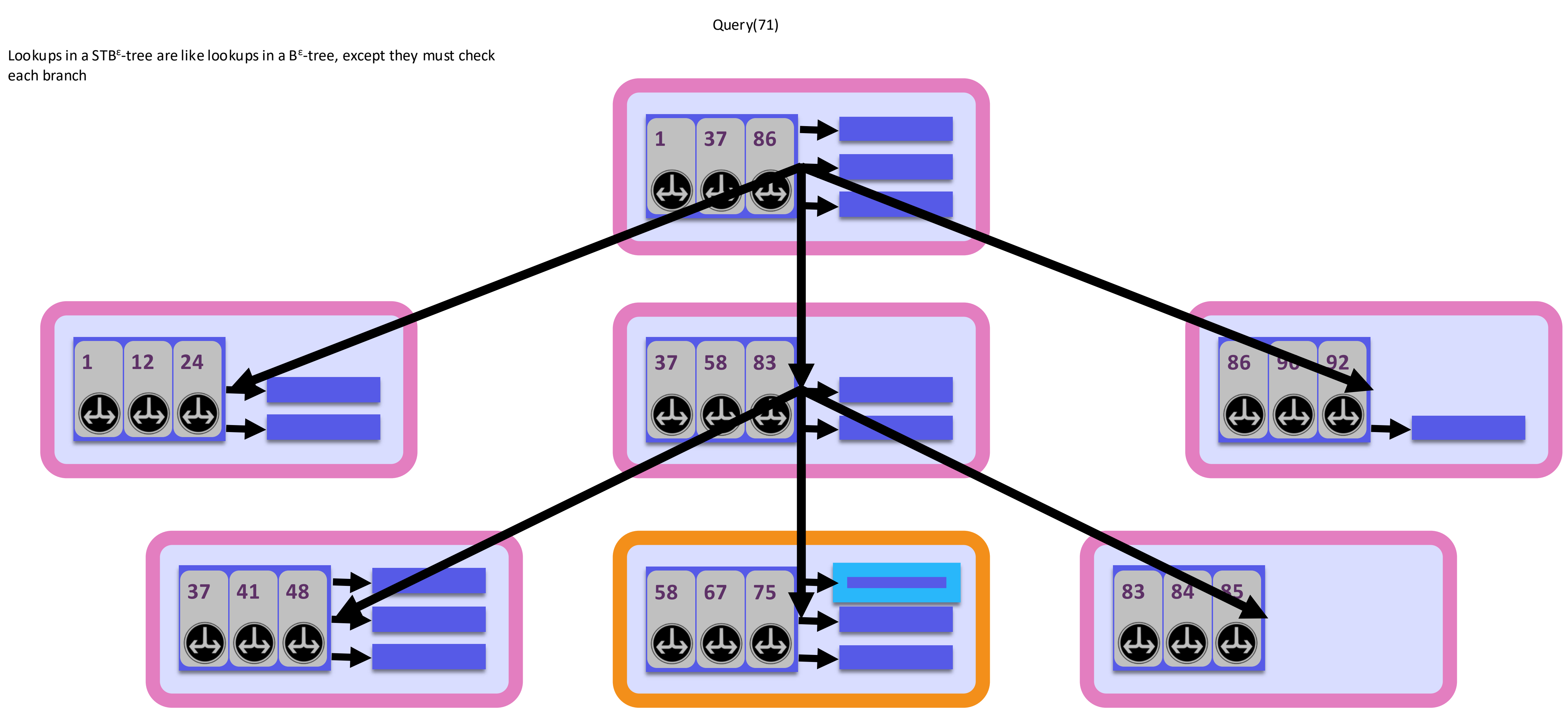
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

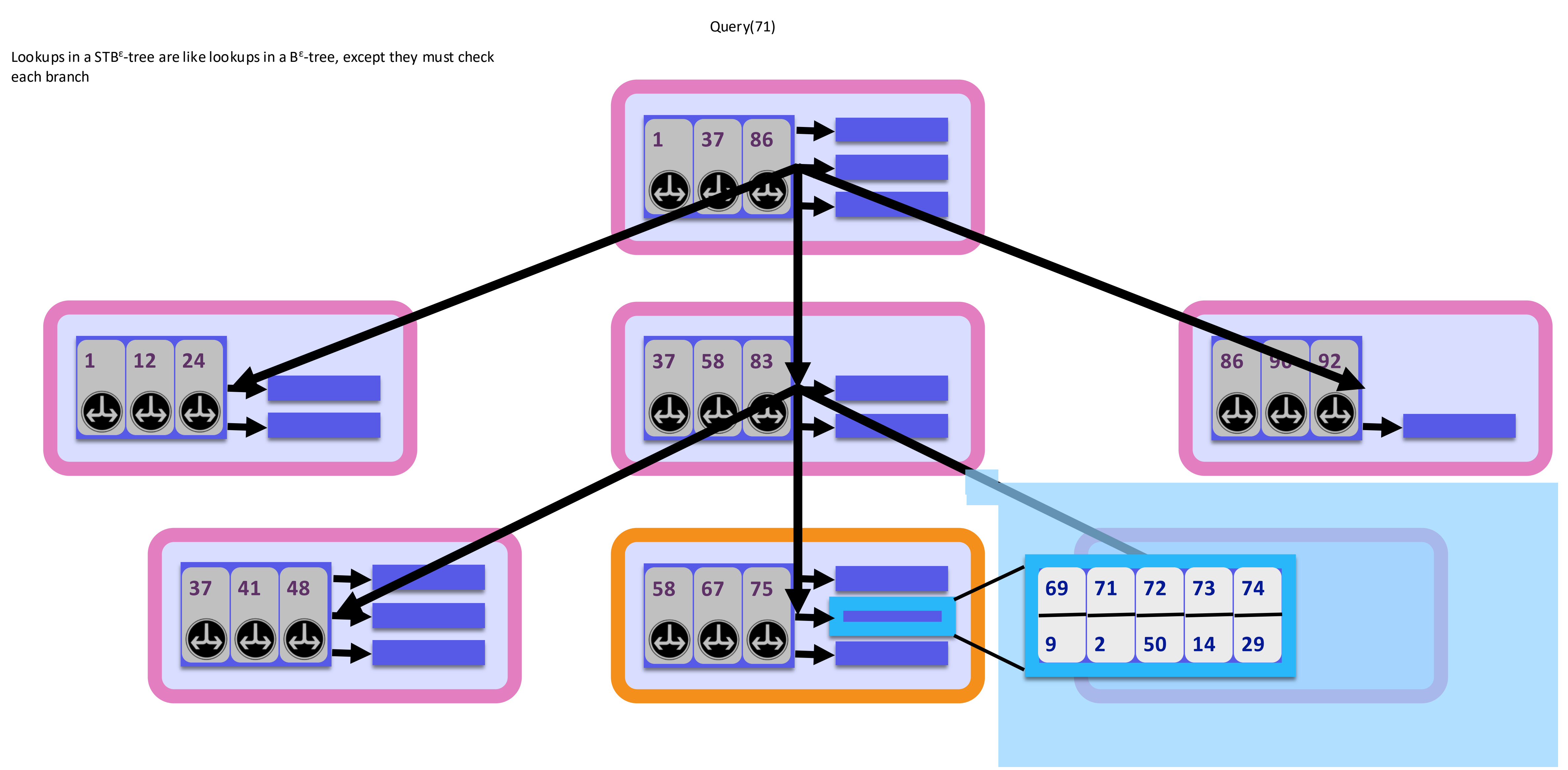


Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

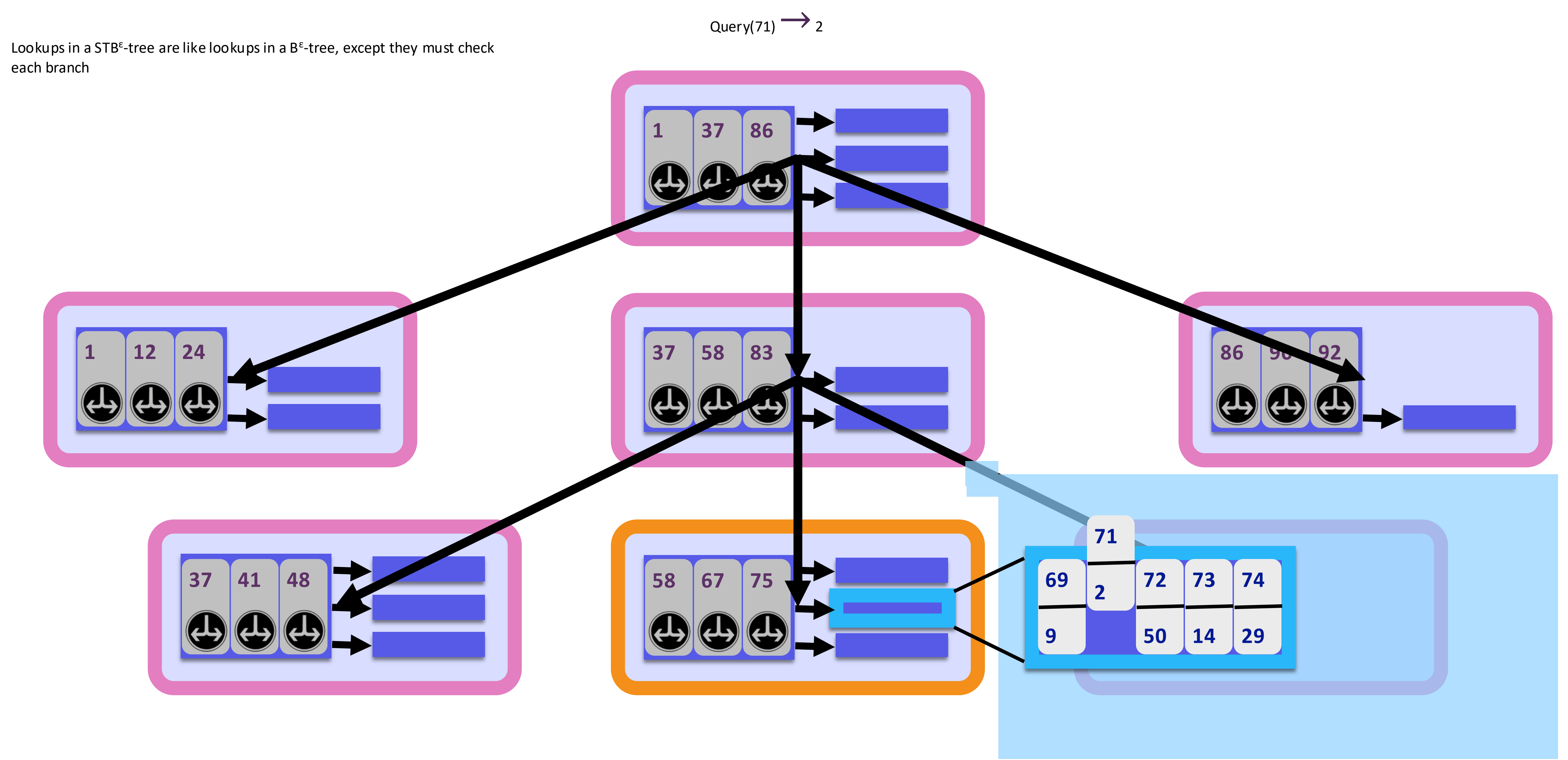


Size-Tiered B$^{\varepsilon}$-Trees



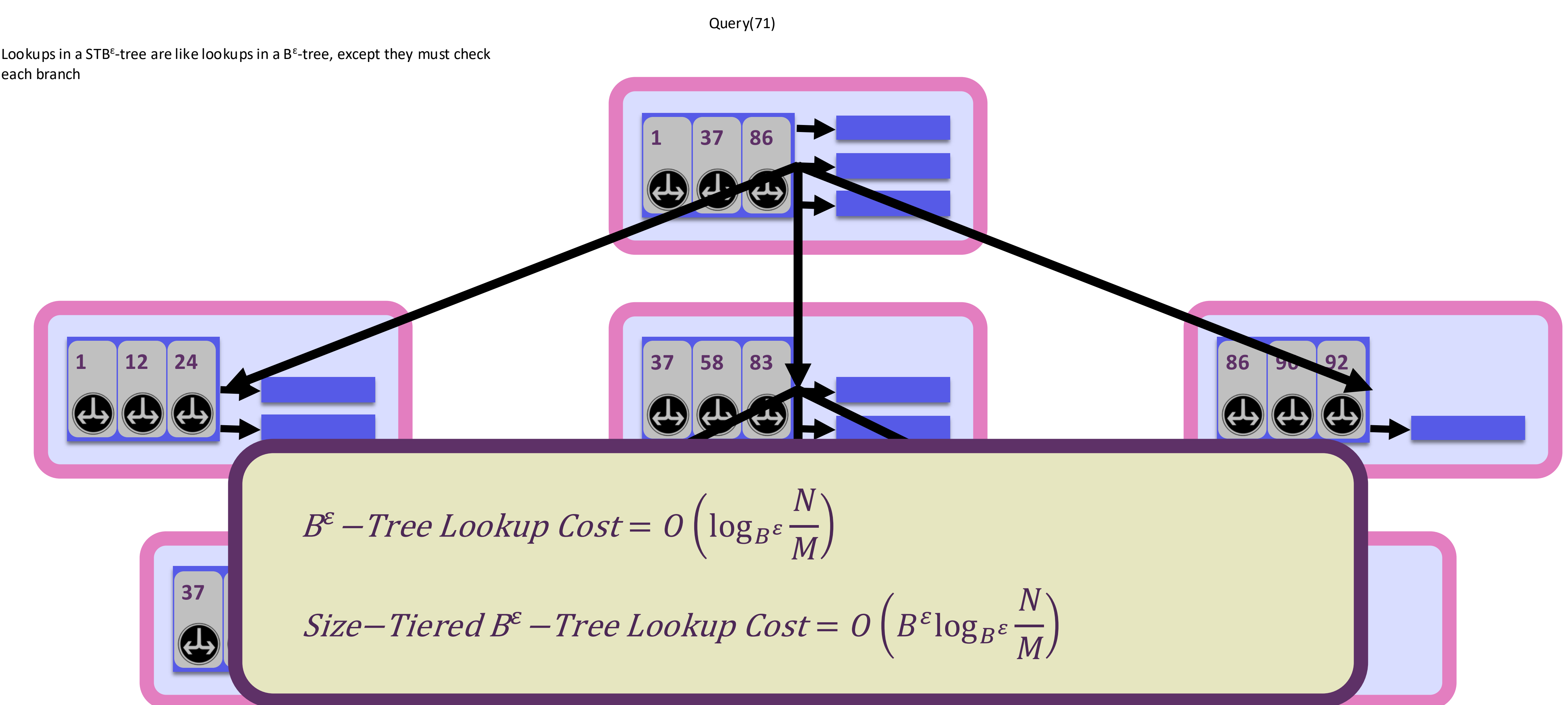
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

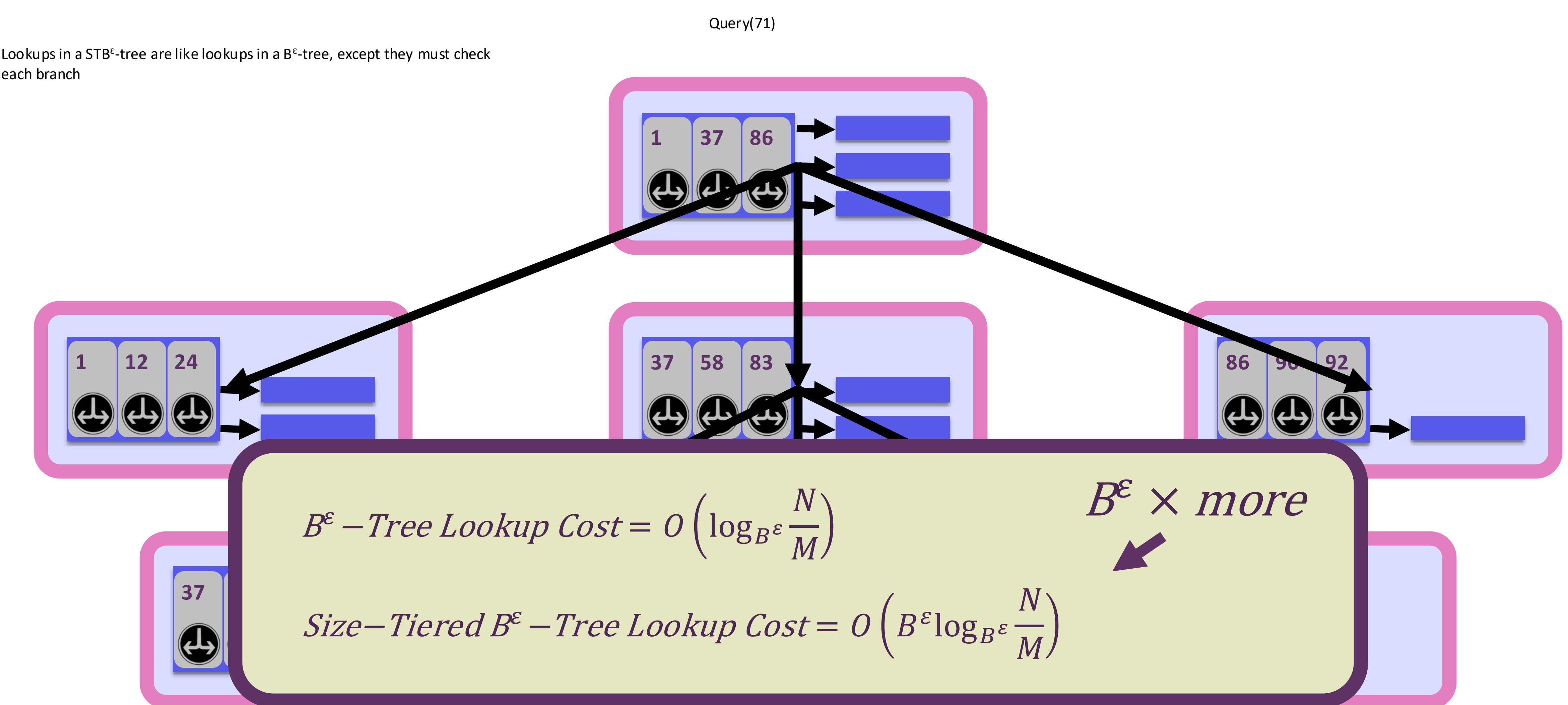


$$B^{\varepsilon}-Tree\ Lookup\ Cost = O\left(\log_{B^{\varepsilon}}\frac{N}{M}\right)$$

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

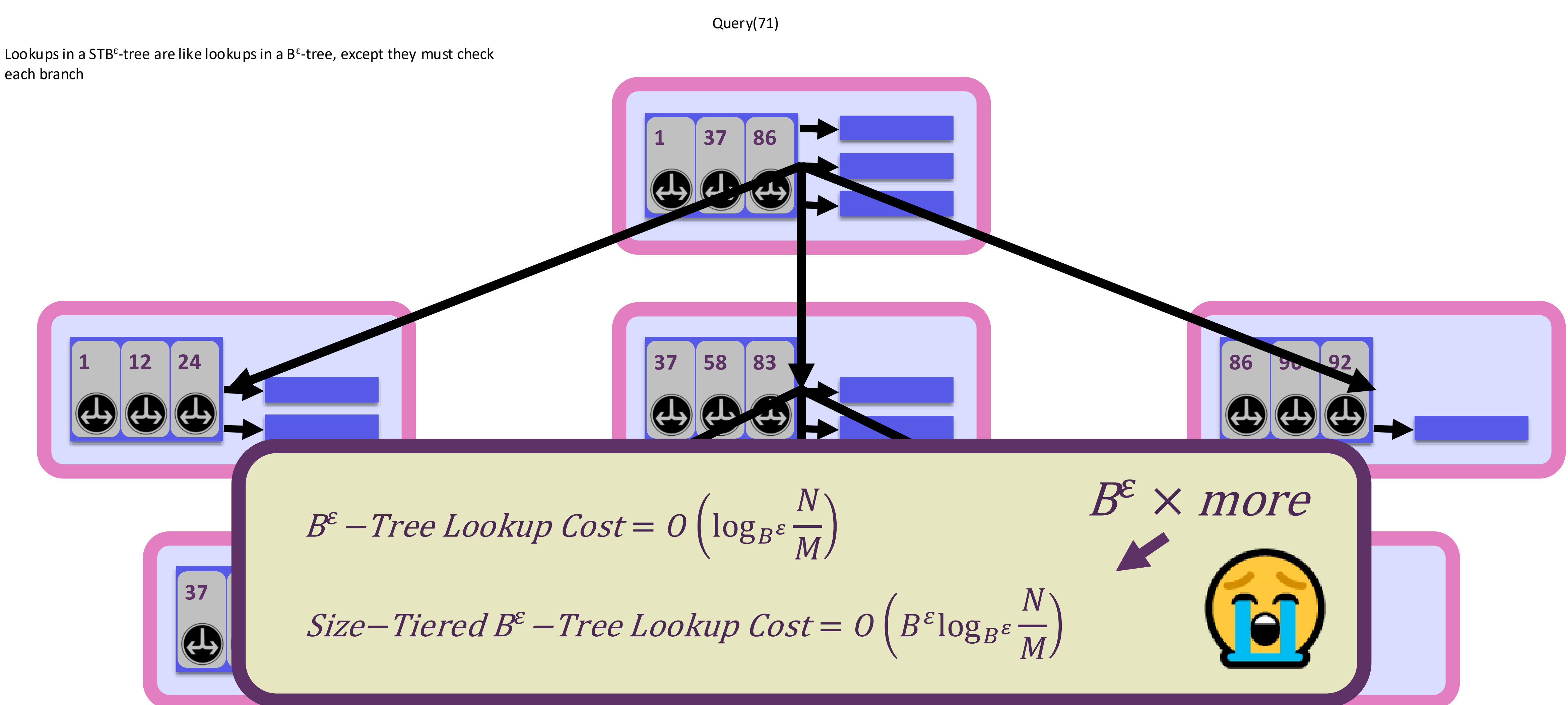
Size-Tiered B$^{\varepsilon}$-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



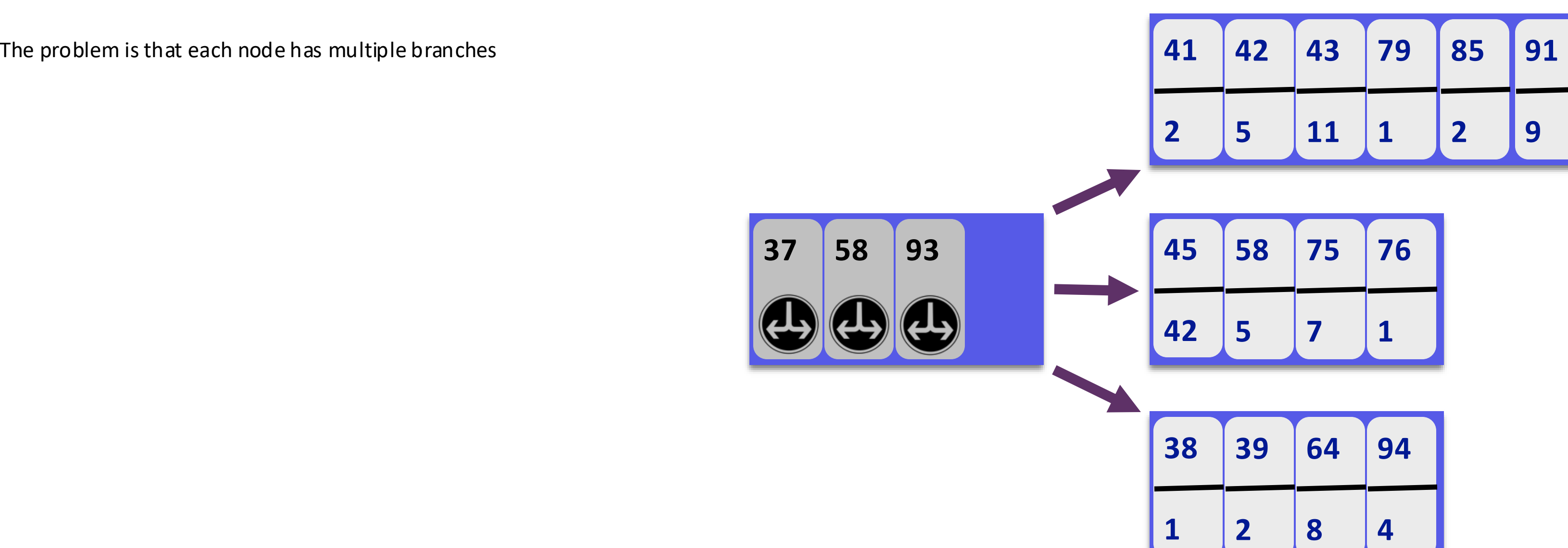
Size-Tiered B^{ε}-Trees

Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



Fixing Lookups (almost)

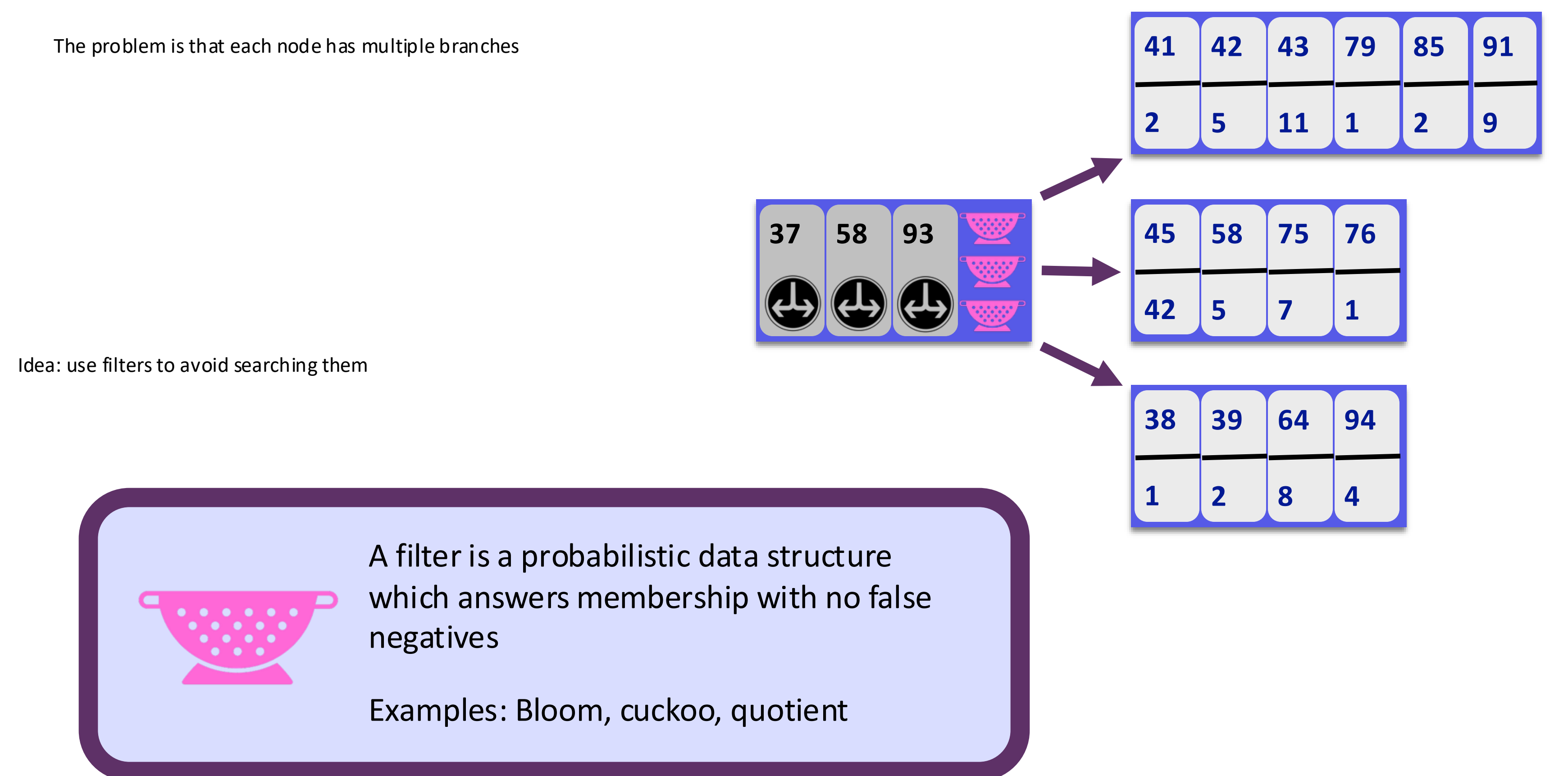
Fixing Lookups (almost)



Fixing Lookups (almost)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



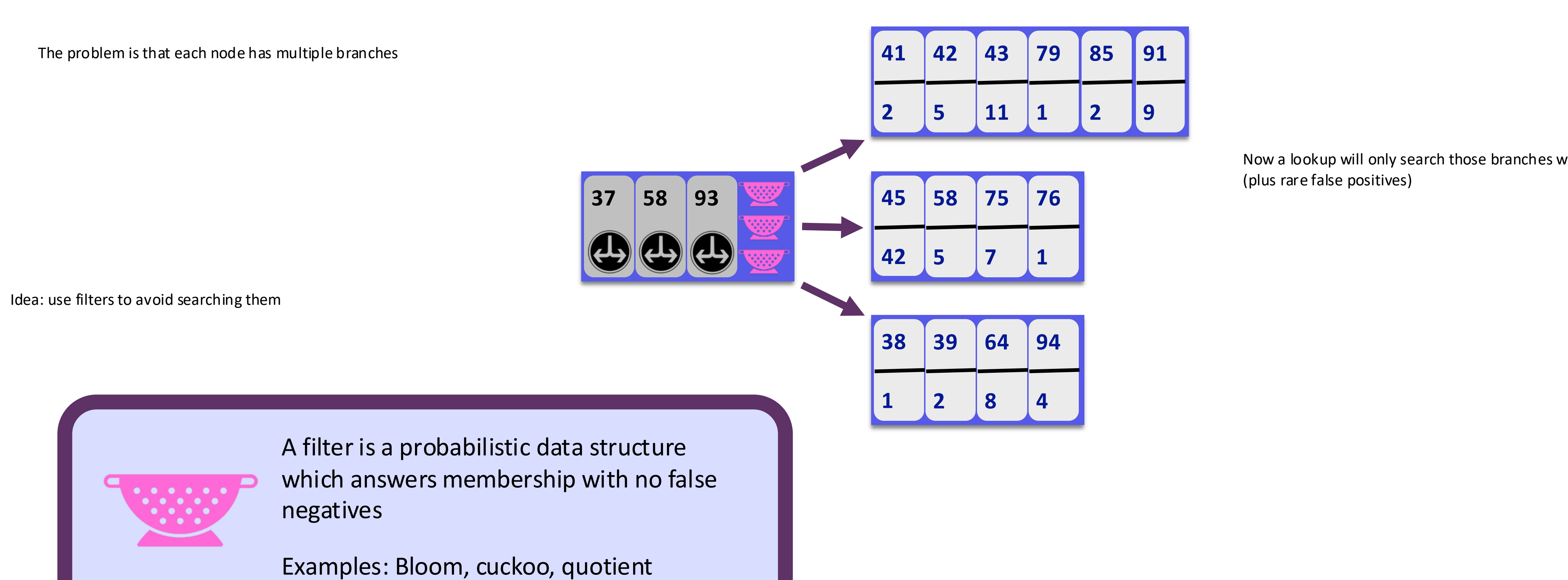
A filter is a probabilistic data structure which answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

The problem is that each node has multiple branches

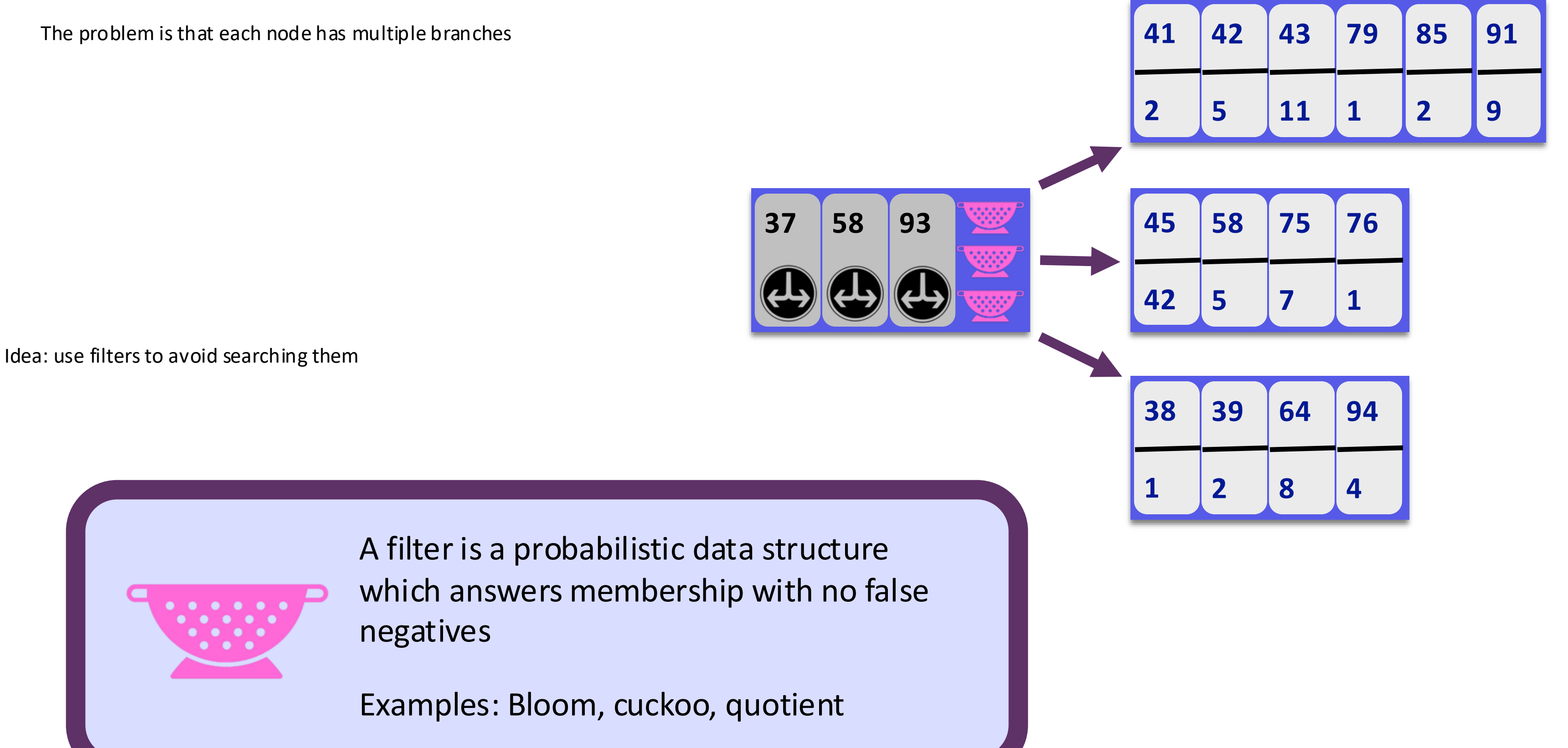
Idea: use filters to avoid searching them



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

Fixing Lookups (almost)

Query(64)



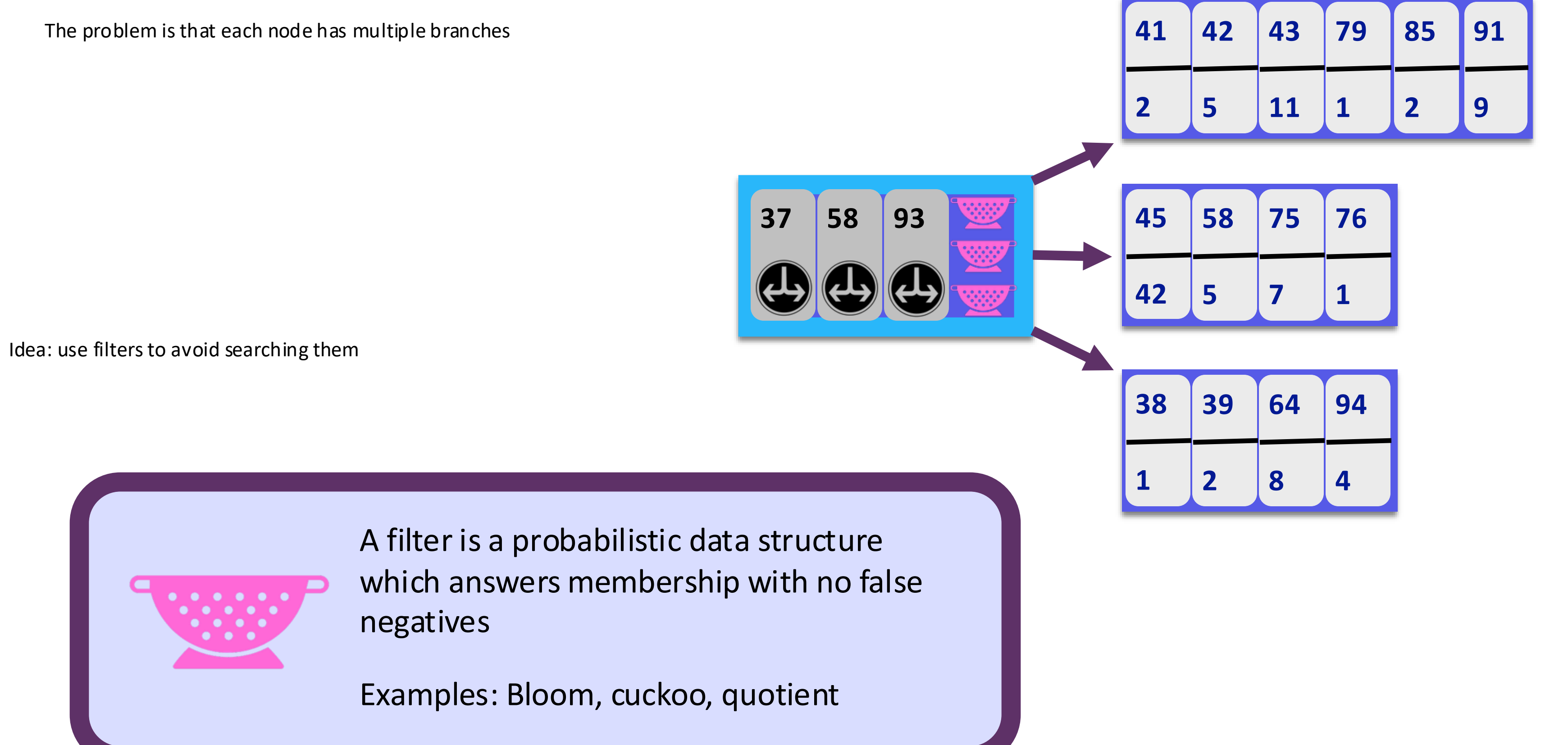
Now a lookup will only search those branches which contain the key (plus rare false positives)

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



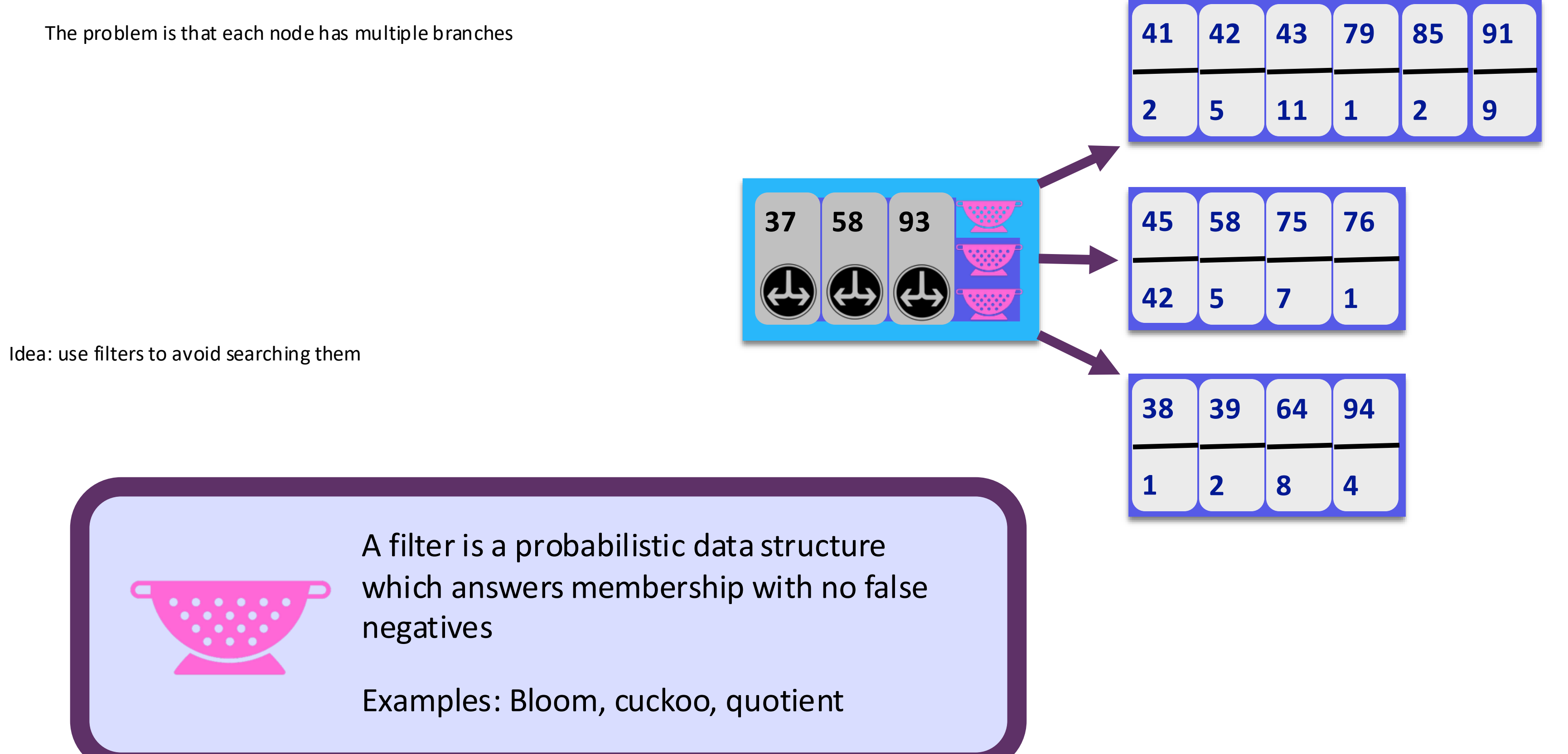
Now a lookup will only search those branches which contain the key (plus rare false positives)

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



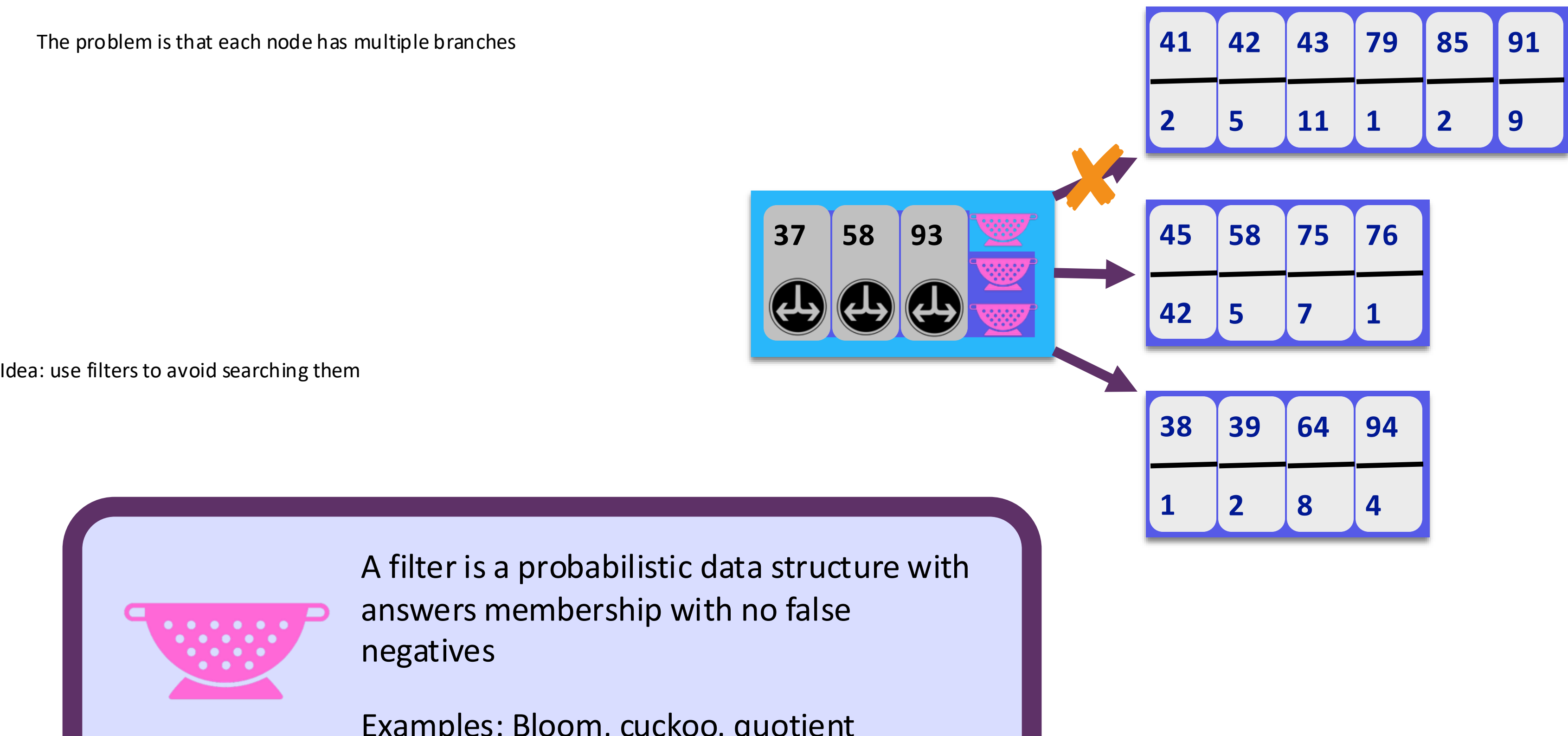
Now a lookup will only search those branches which contain the key (plus rare false positives)

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



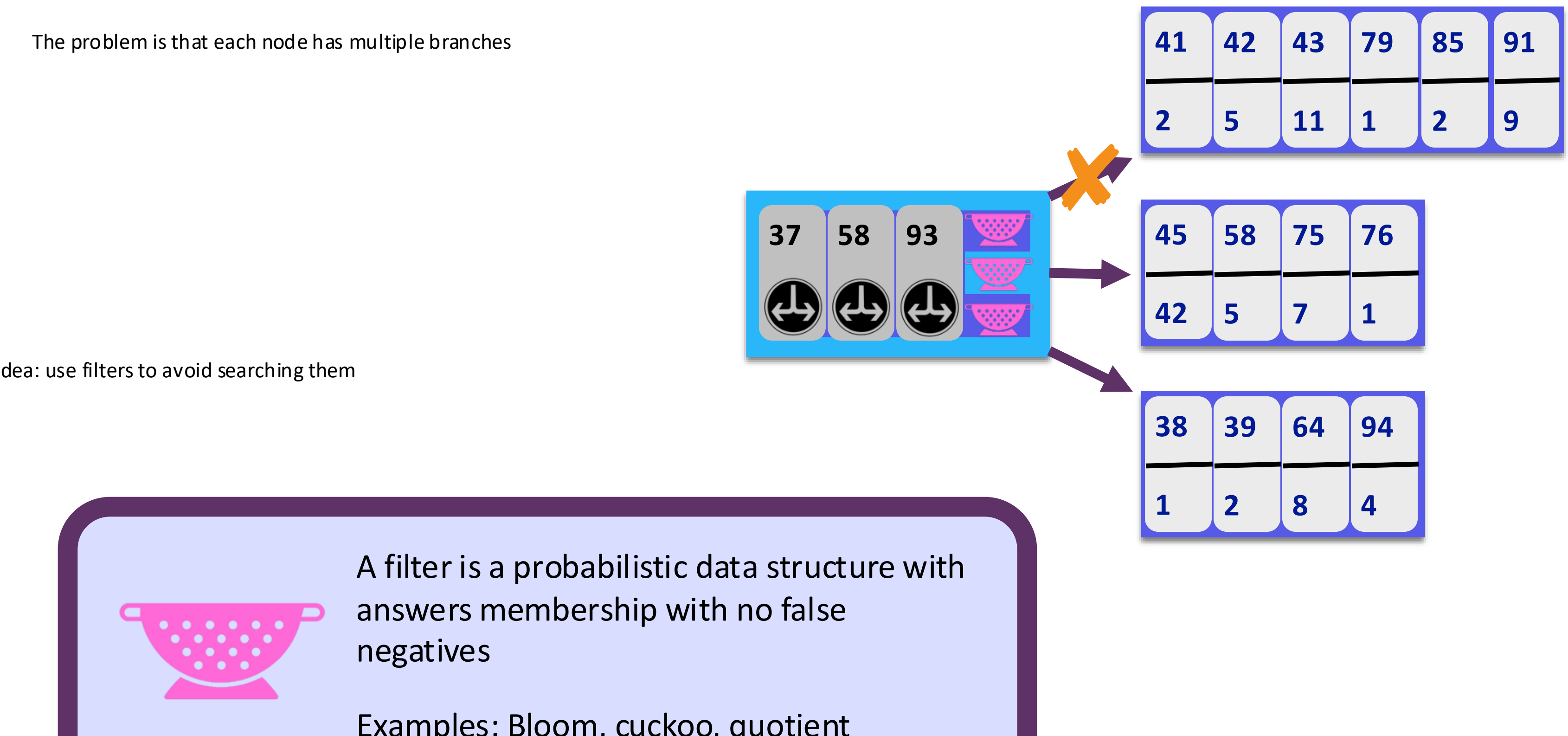
Now a lookup will only search those branches which contain the key (plus rare false positives)

Fixing Lookups (almost)

Query(64)

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)

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them

41	42	43	79	85	91
2	5	11	1	2	9

37	58	93

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

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

Query(64)

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 (almost)

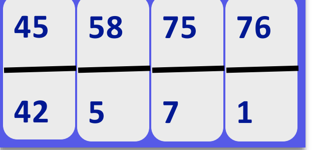
Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them

37 58 93

41 42 43 79 85 91
2 5 11 1 2 9



38 39 64 94
1 2 8 4

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

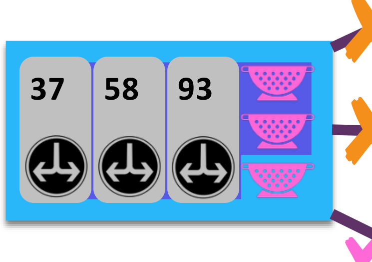
Examples: Bloom, cuckoo, quotient

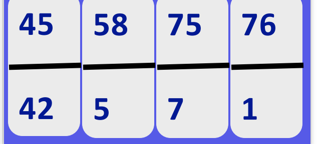
Fixing Lookups (almost)

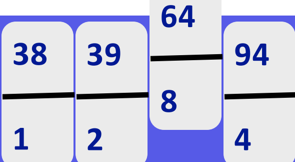
Query(64) → 8

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)

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

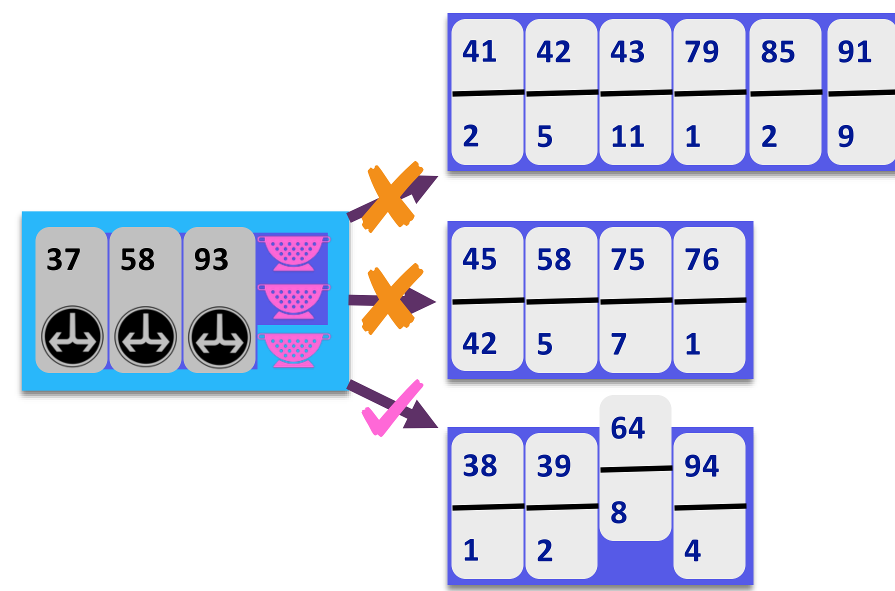
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

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)

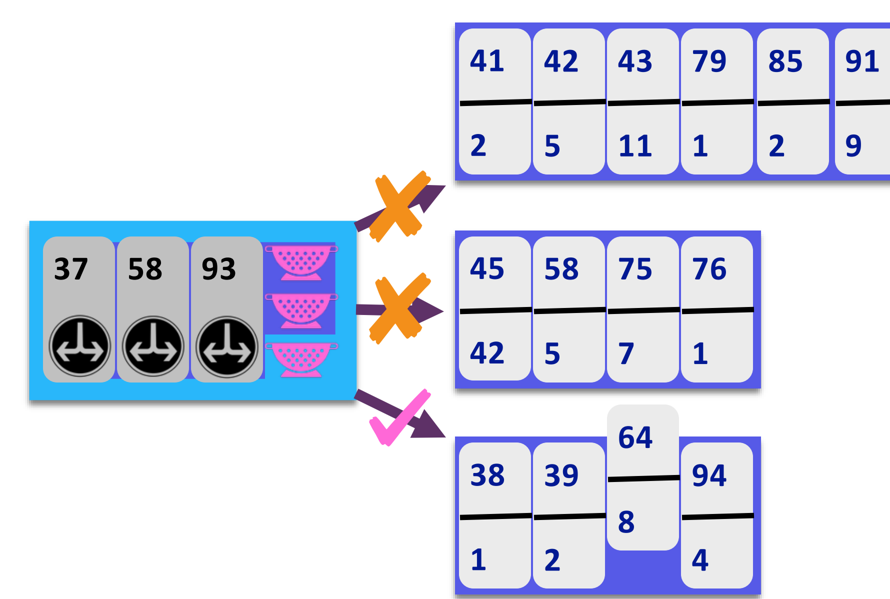
$$\textit{False Positive Rate} \leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_{B} N}\right)$$

Fixing Lookups (almost)

Query(64) $\longrightarrow$ 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)

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

Lookups in O(1) IOs

Conclusion

- B^e-trees are asymptotically faster than B-trees for insertions.
- They are appropriate for OLTP workloads
- Size-tiered B^e-trees help reduce write amplification
- Filter data structure can help reduce read amplification