CS 6530: Advanced Database Systems Fall 2023

Lecture 10
Write-Optimized Indexes

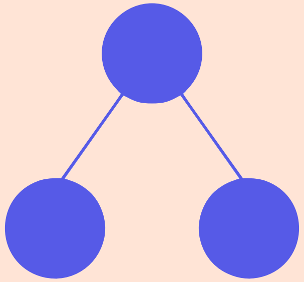
Prashant Pandey

prashant.pandey@utah.edu

Slides taken from Prof. Alex Conway, Cornell Tech

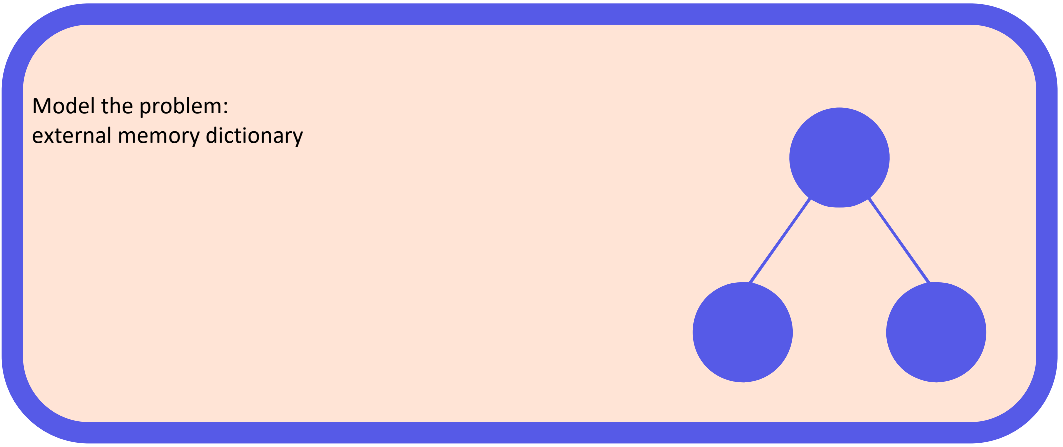
The Story of SplinterDB

Model the problem:
external memory dictionary

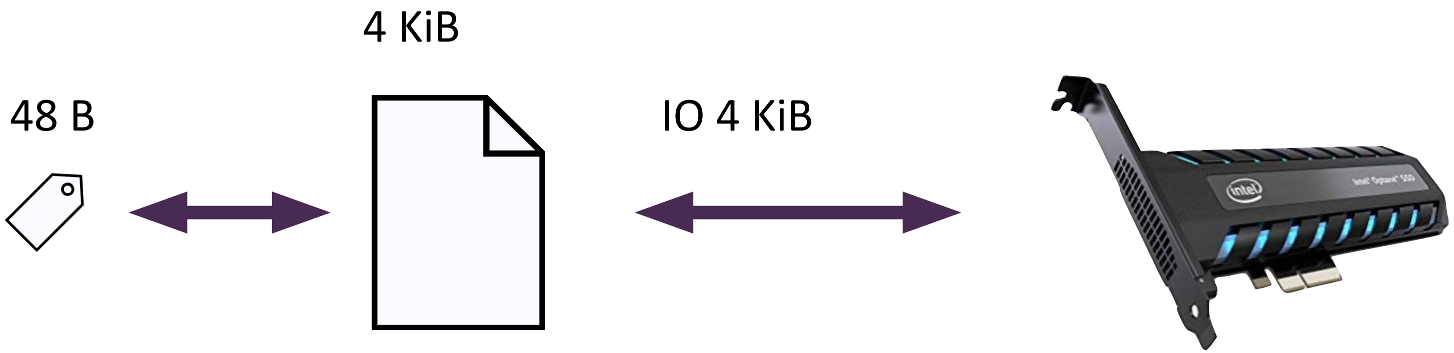


The Story of SplinterDB

Model the problem:
external memory dictionary

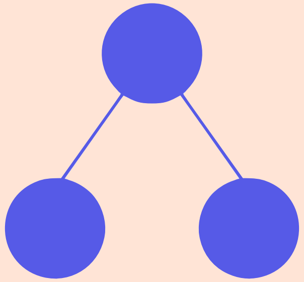


Metadata is fine-grained

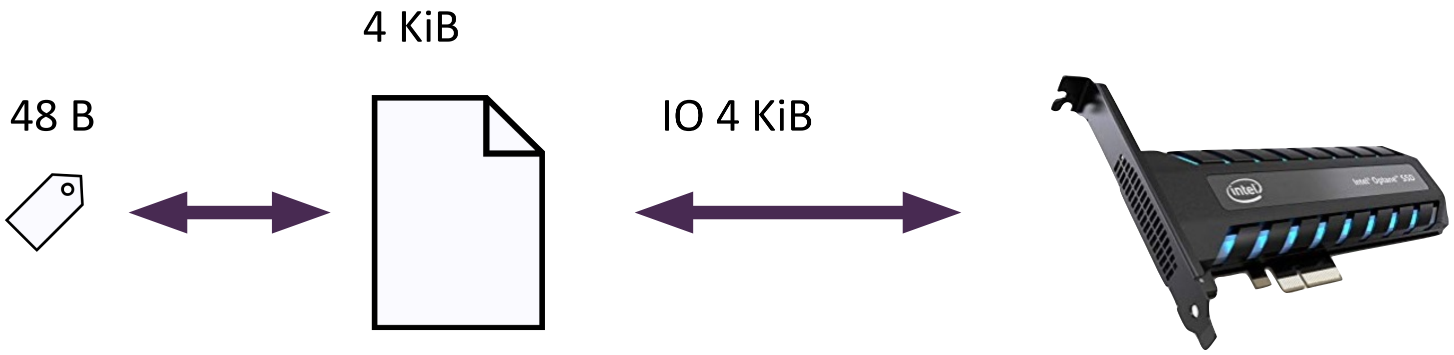


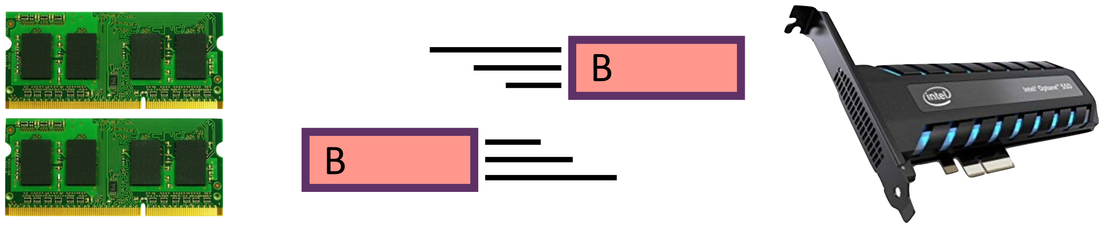
The Story of SplinterDB

Model the problem:
external memory dictionary



Metadata is fine-grained



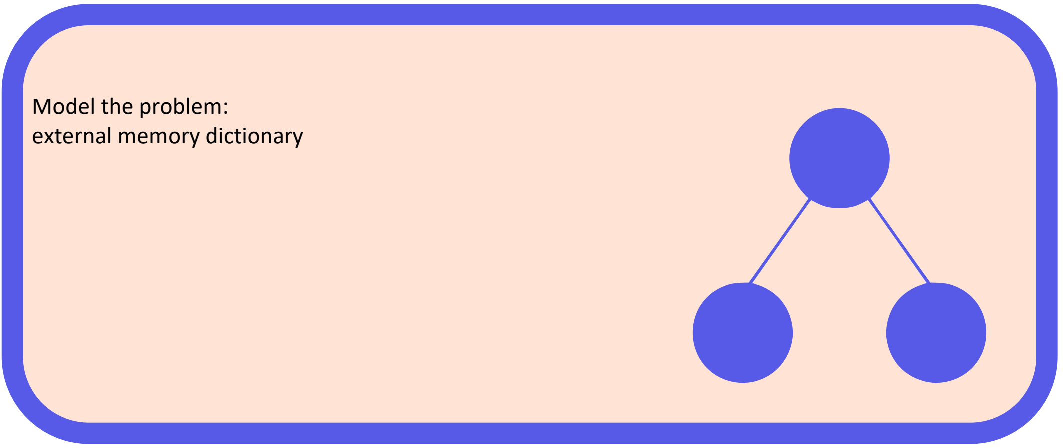


Internal Memory of size M

A B-sized block can be read or written in 1 IO

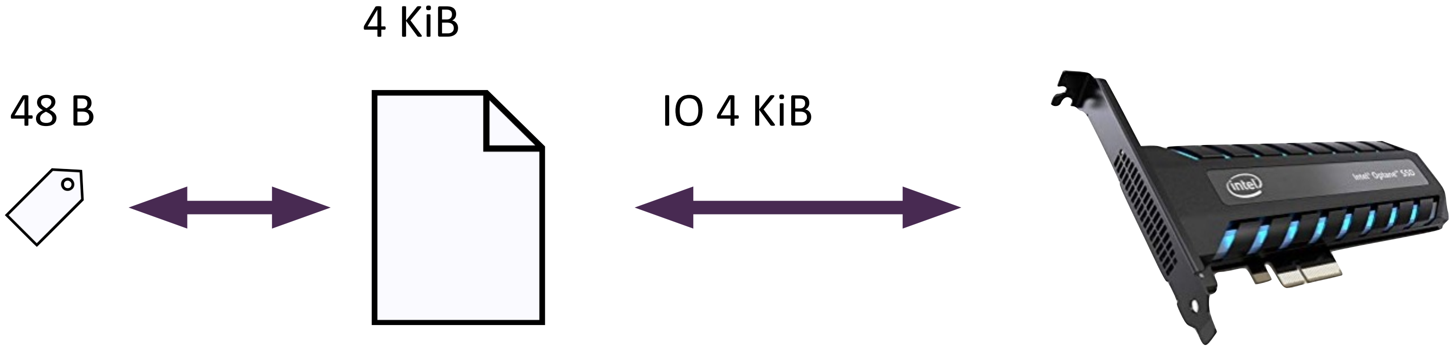
External Memory Model

The Story of SplinterDB



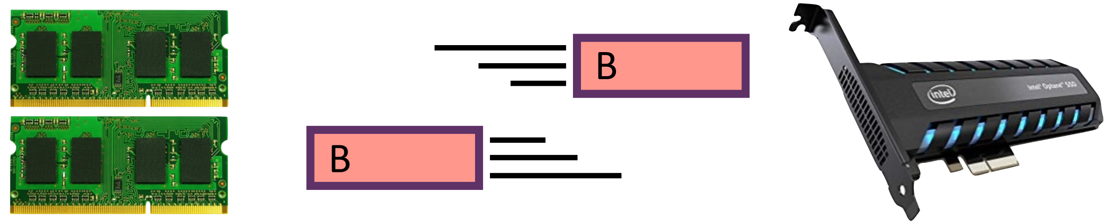
Model the problem:
external memory dictionary

Metadata is fine-grained



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



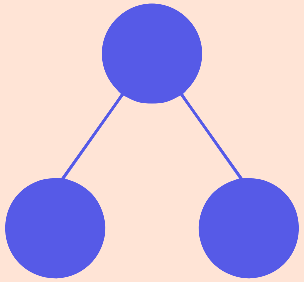
Internal Memory of size M

A B-sized block can be read or written in 1 IO

External Memory Model

The Story of SplinterDB

Model the problem:
external memory dictionary



Two Flavors of
External-Memory Dictionary

Different lower bounds
(performance limits)

Comparison-Based Dictionaries

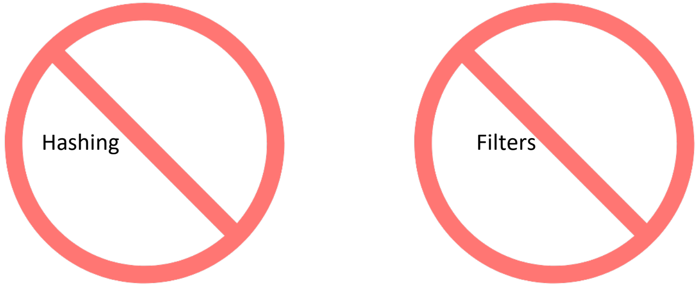
Comparison External Memory Model

```
                              <
user024299                    =        user082587
                              >
```

Comparison-Based Dictionaries

Comparison External Memory Model

```
                                          <
user024299                                =          user082587
                                          >
```



Hashing

Filters

Comparison-Based Dictionaries

Comparison External Memory Model

```
                          <
user024299                =          user082587
                          >
```

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

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

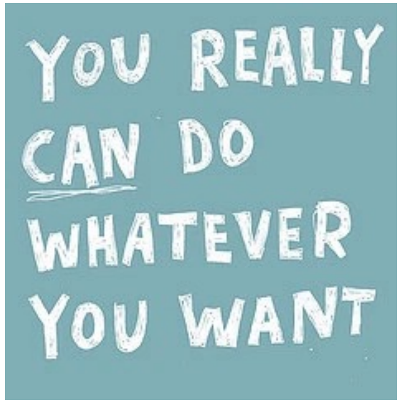
General Dictionaries

External Memory Model

General Dictionaries

External Memory Model

user024299



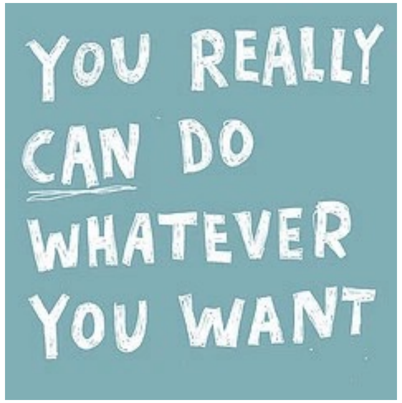
General Dictionaries

External Memory Model

`user024299`

Hashing

`XXH(user024299)`



General Dictionaries

External Memory Model

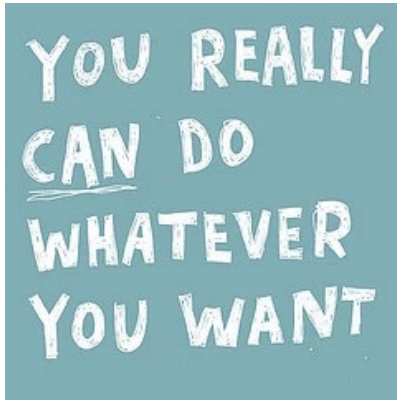
```
user024299
```

Hashing

```
XXH(user024299)
```

Filters

```
qf_insert(user024299)
```



General Dictionaries

External Memory Model

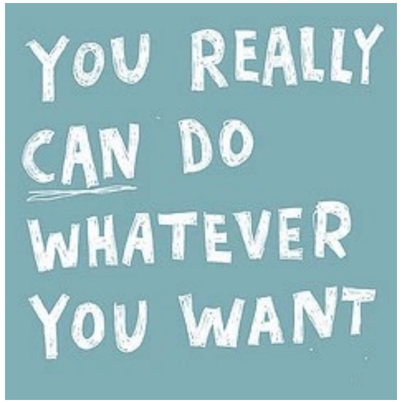
```
user024299
```

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 $\Omega(\log_\lambda N)$

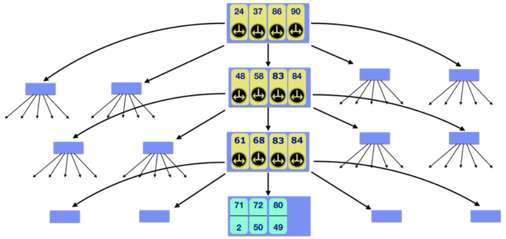
General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound

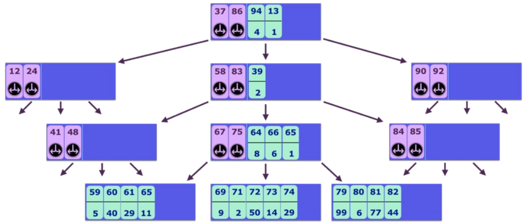
Insertions in ⟷ Lookups in

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



B-Trees

$(\lambda = B)$



B$^\varepsilon$-Trees

$(\lambda = B^\varepsilon)$

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in ⟷ Lookups in

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

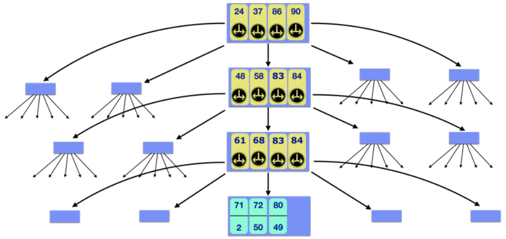
General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound

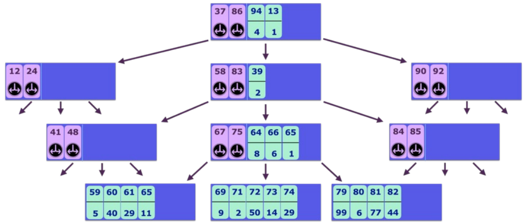
Insertions in ↔ Lookups in

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



B-Trees

$$(\lambda = B)$$



B$^{\varepsilon}$-Trees

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

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in ↔ Lookups in

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

Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound

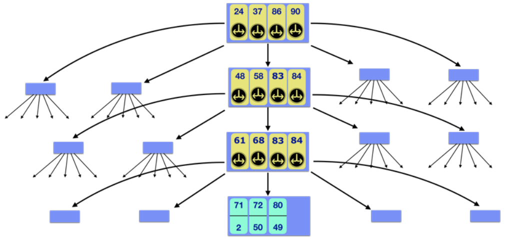
Insertions in

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

$\longleftrightarrow$

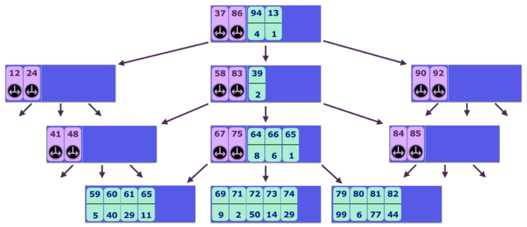
Lookups in

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



B-Trees

$$(\lambda = B)$$



B^ε-Trees

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

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

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

$\longleftrightarrow$

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

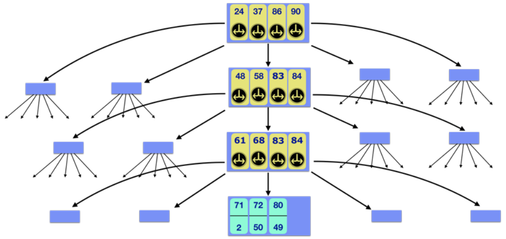
Brodal-Fagerberg Lower Bound

Insertions in

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

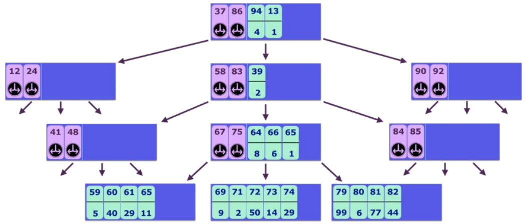
Lookups in

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



B-Trees

$$(\lambda = B)$$



B$^{\varepsilon}$-Trees

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

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

Insertions in

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

Lookups in

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

No scans!

Iacono-Patrascu Hash Table

BoA/BoT
Hash Table

General External Memory Model

Lower Bounds

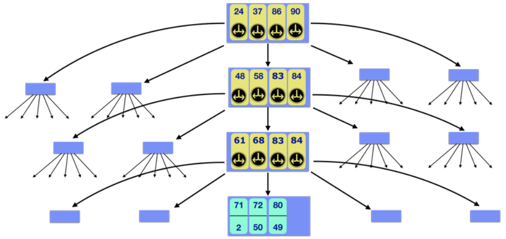
Brodal-Fagerberg Lower Bound

Insertions in

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

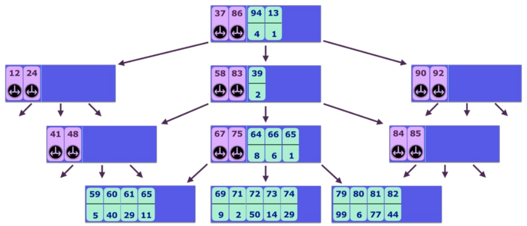
Lookups in

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



B-Trees

$$(\lambda = B)$$



B$^\varepsilon$-Trees

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

Comparison External Memory Model

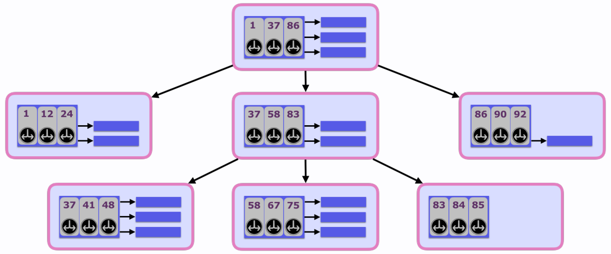
Iacono-Pătrașcu Lower Bound

Insertions in

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

Lookups in

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



Mapped B$^\varepsilon$-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

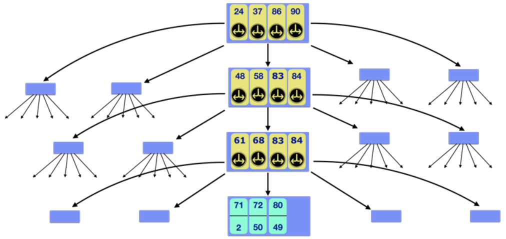
Insertions in

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

⟷

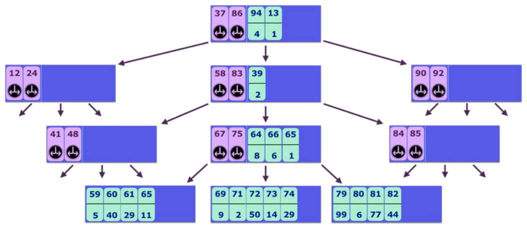
Lookups in

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



B-Trees

$$(\lambda = B)$$



B$^\varepsilon$-Trees

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

Comparison External Memory Model

Iacono-Pătrașcu Lower Bound

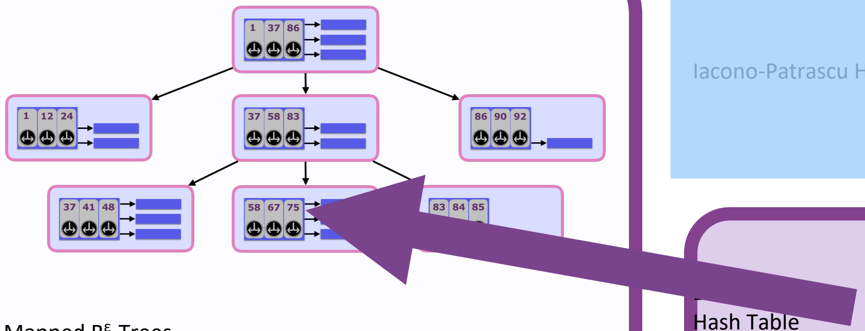
Insertions in

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

⟷

Lookups in

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



Iacono-Patrascu Hash Table

Hash Table

Mapped B$^\varepsilon$-Trees

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

General External Memory Model

I/O Amplification

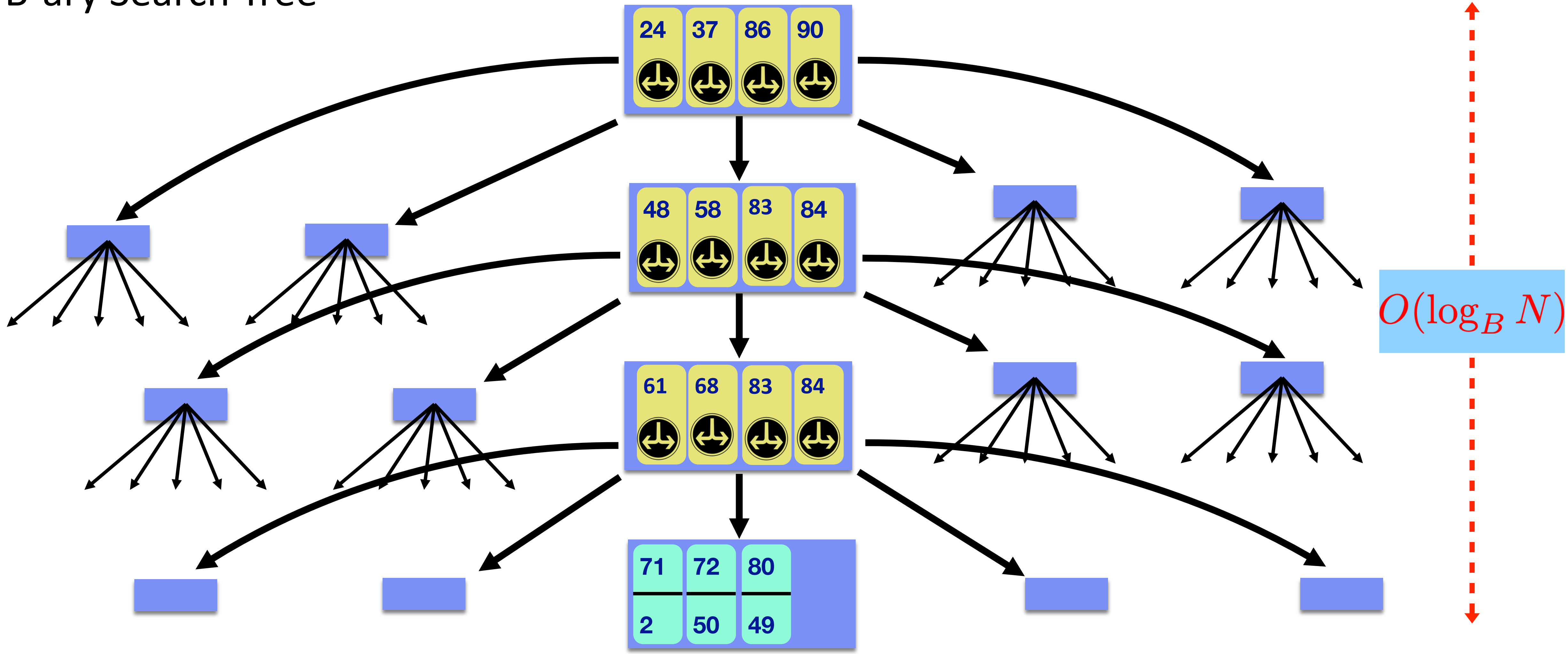
Read amplification is the ratio of the number of blocks read from the disk versus the number of blocks required to read the key-value pair.

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

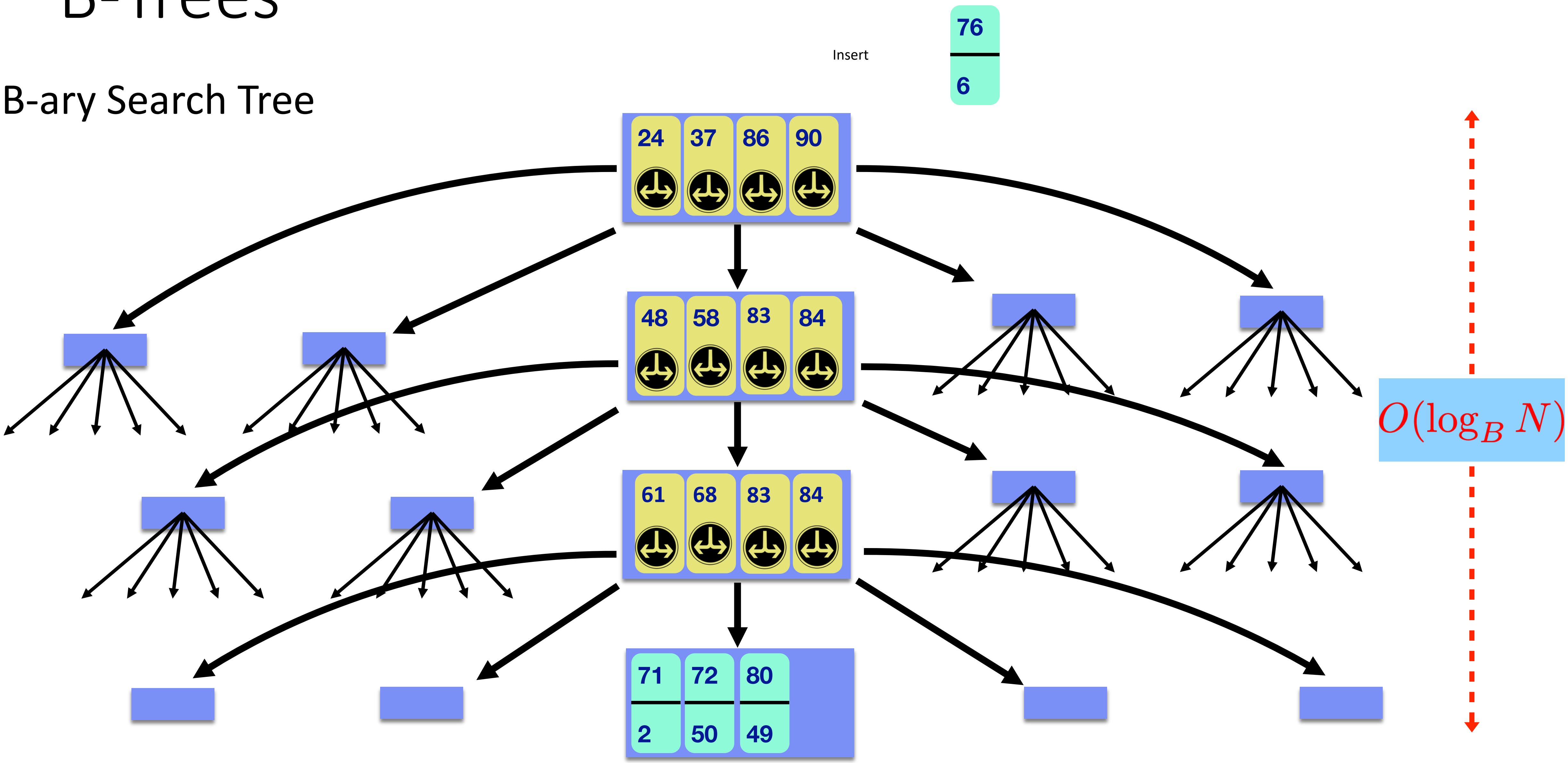
B-Trees

B-ary Search Tree



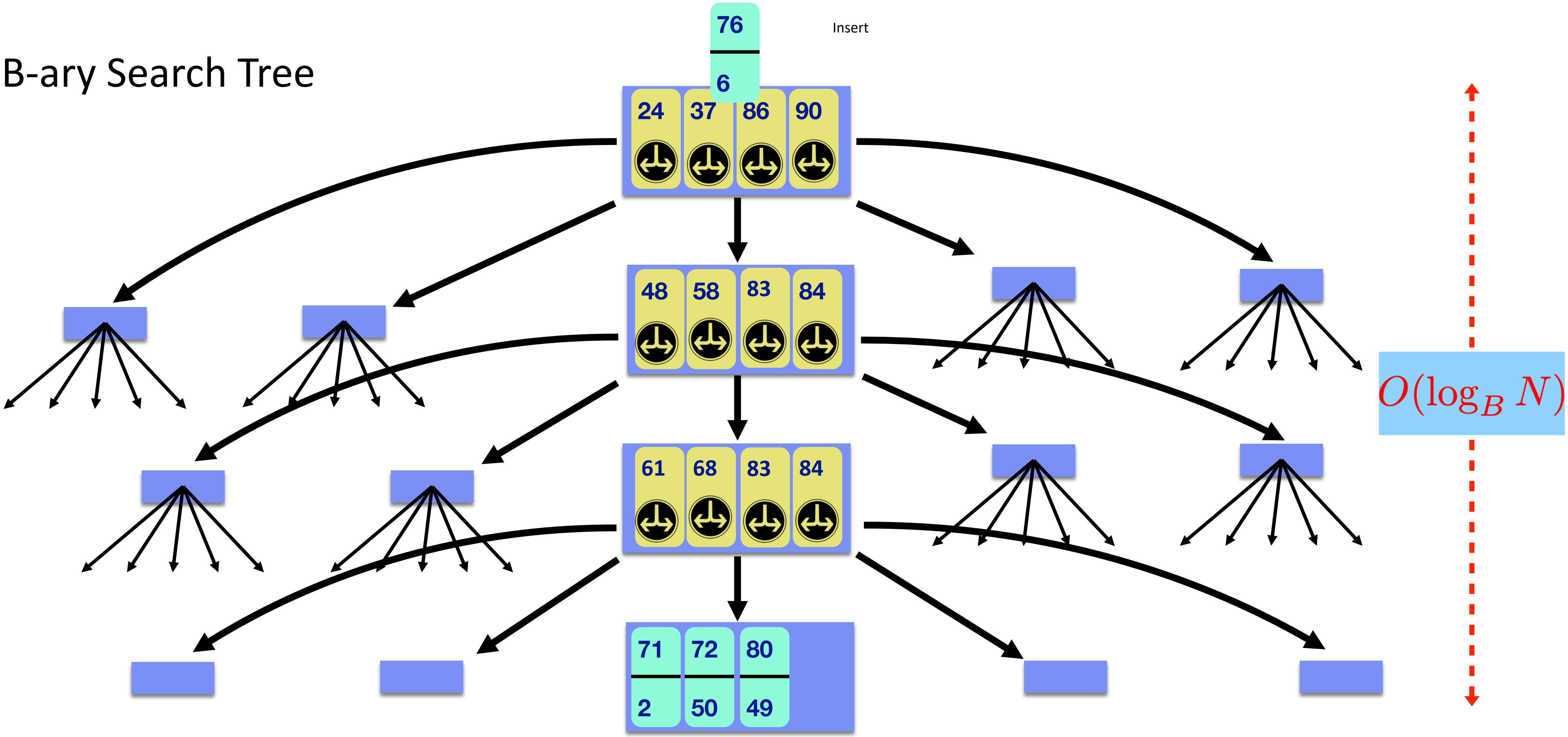
B-Trees

B-ary Search Tree



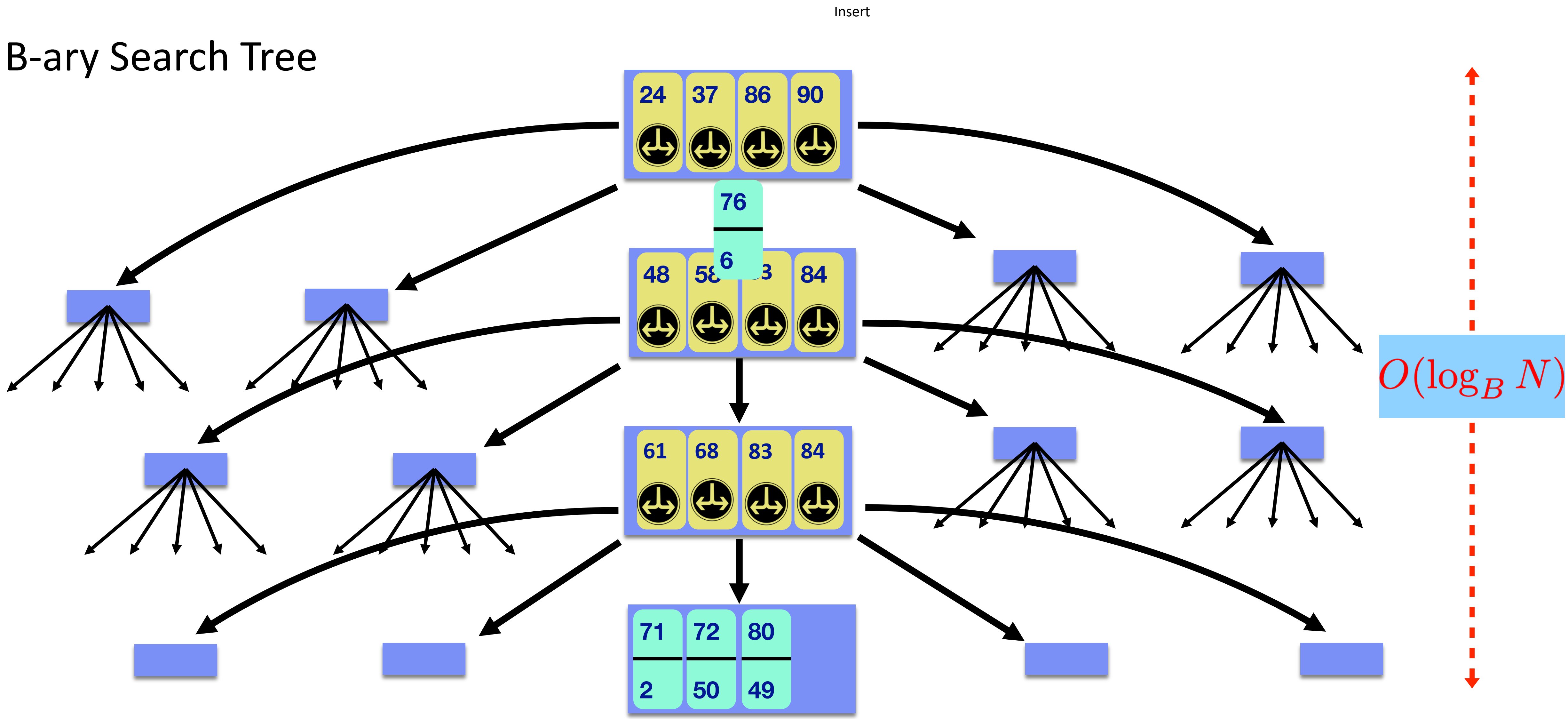
B-Trees

B-ary Search Tree



B-Trees

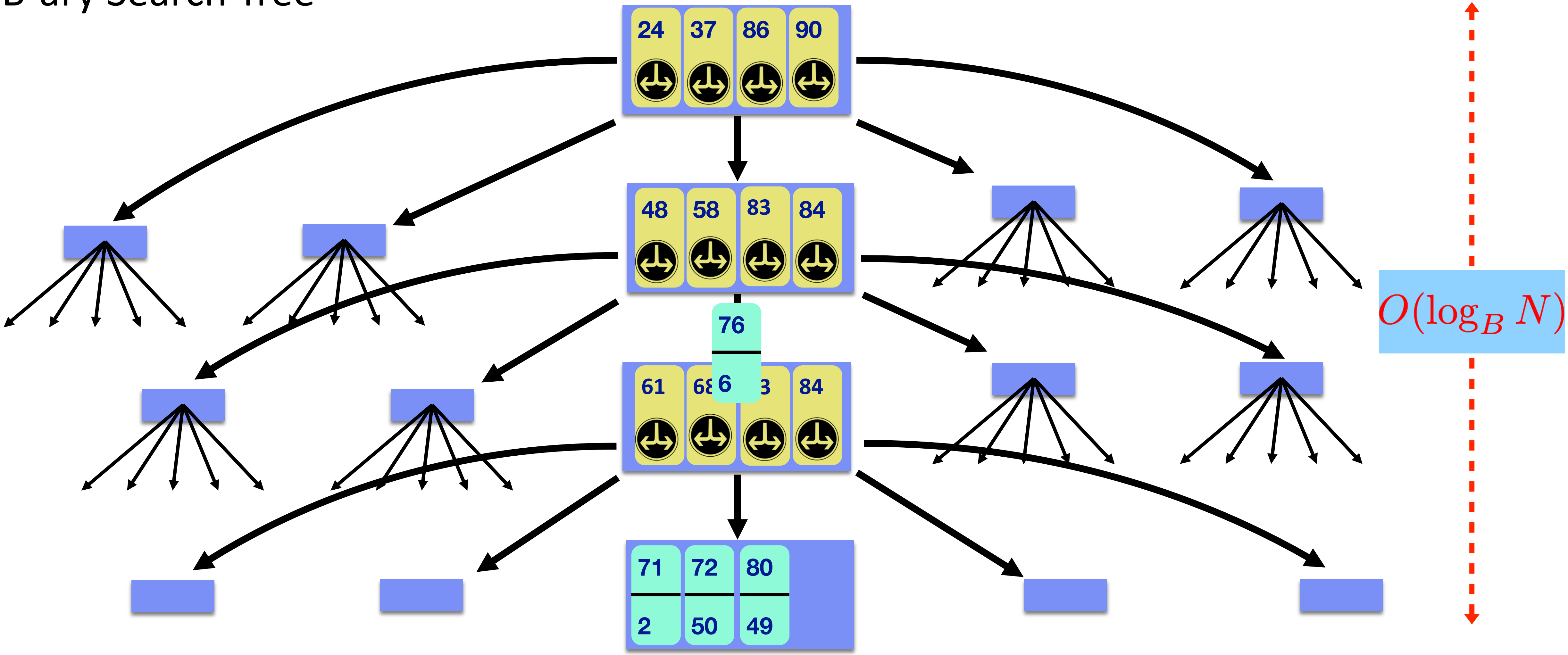
B-ary Search Tree



B-Trees

Insert

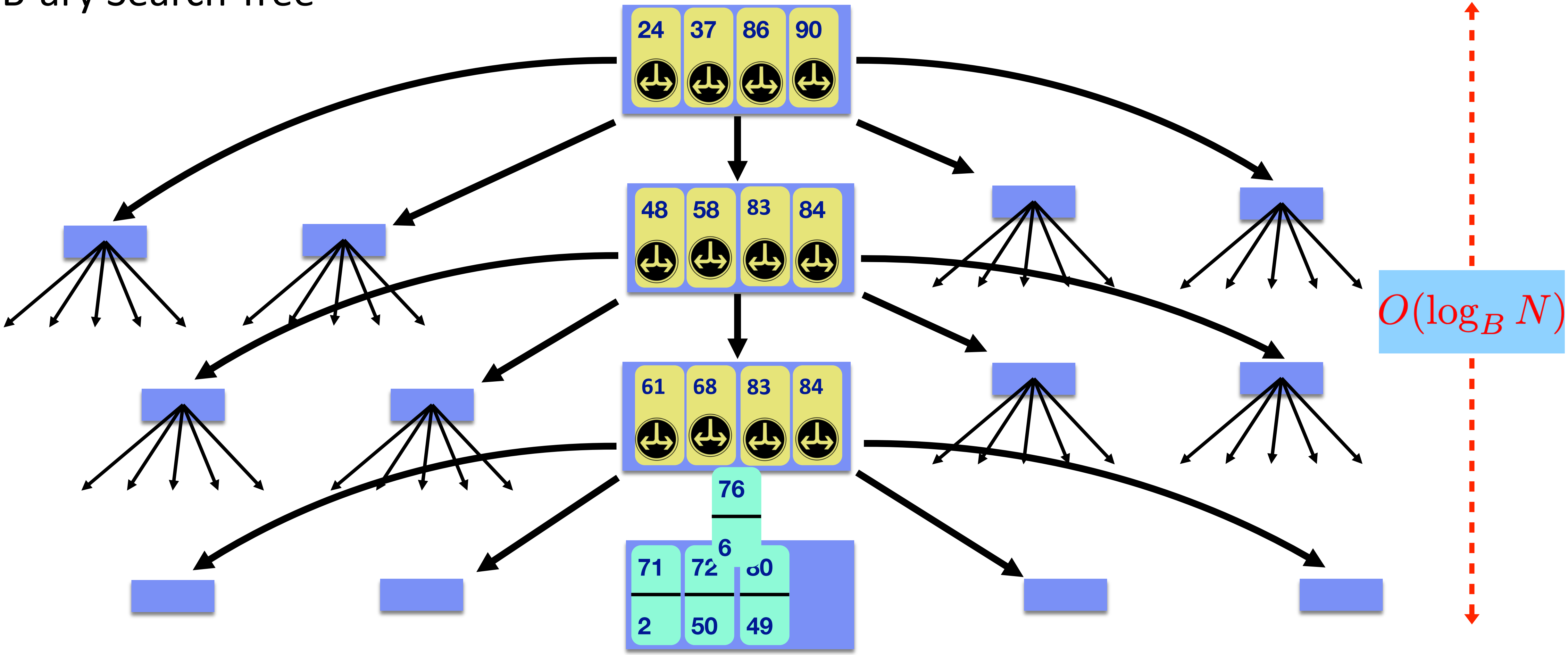
B-ary Search Tree



B-Trees

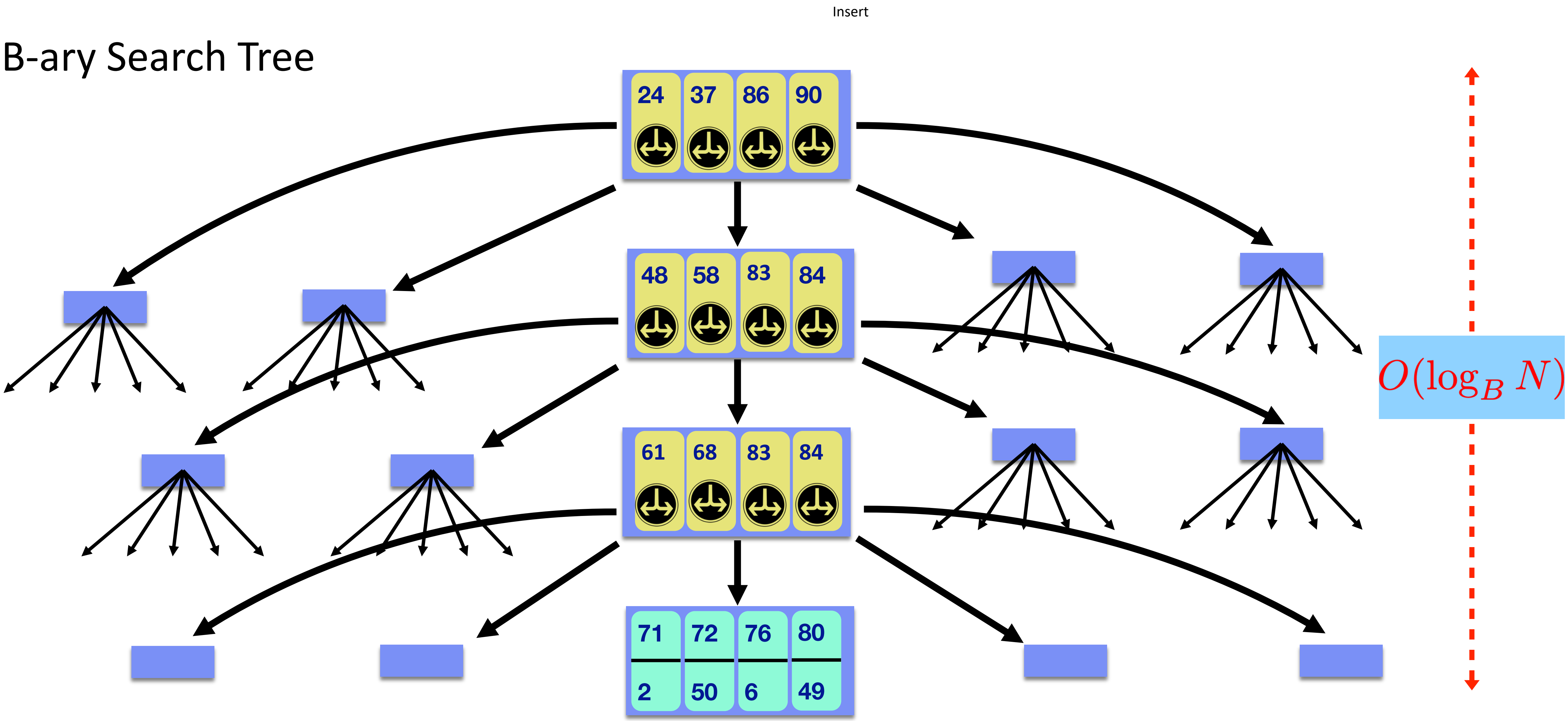
Insert

B-ary Search Tree



B-Trees

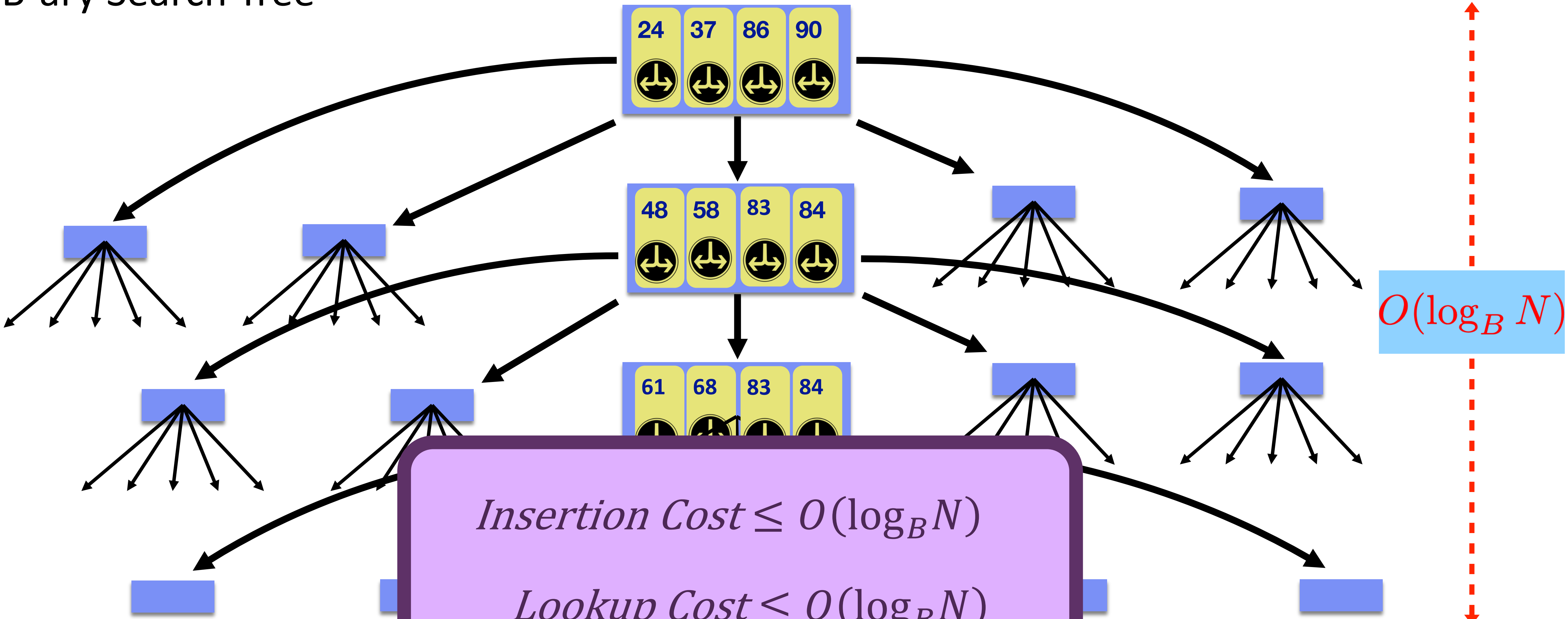
B-ary Search Tree



B-Trees

Insert

B-ary Search Tree



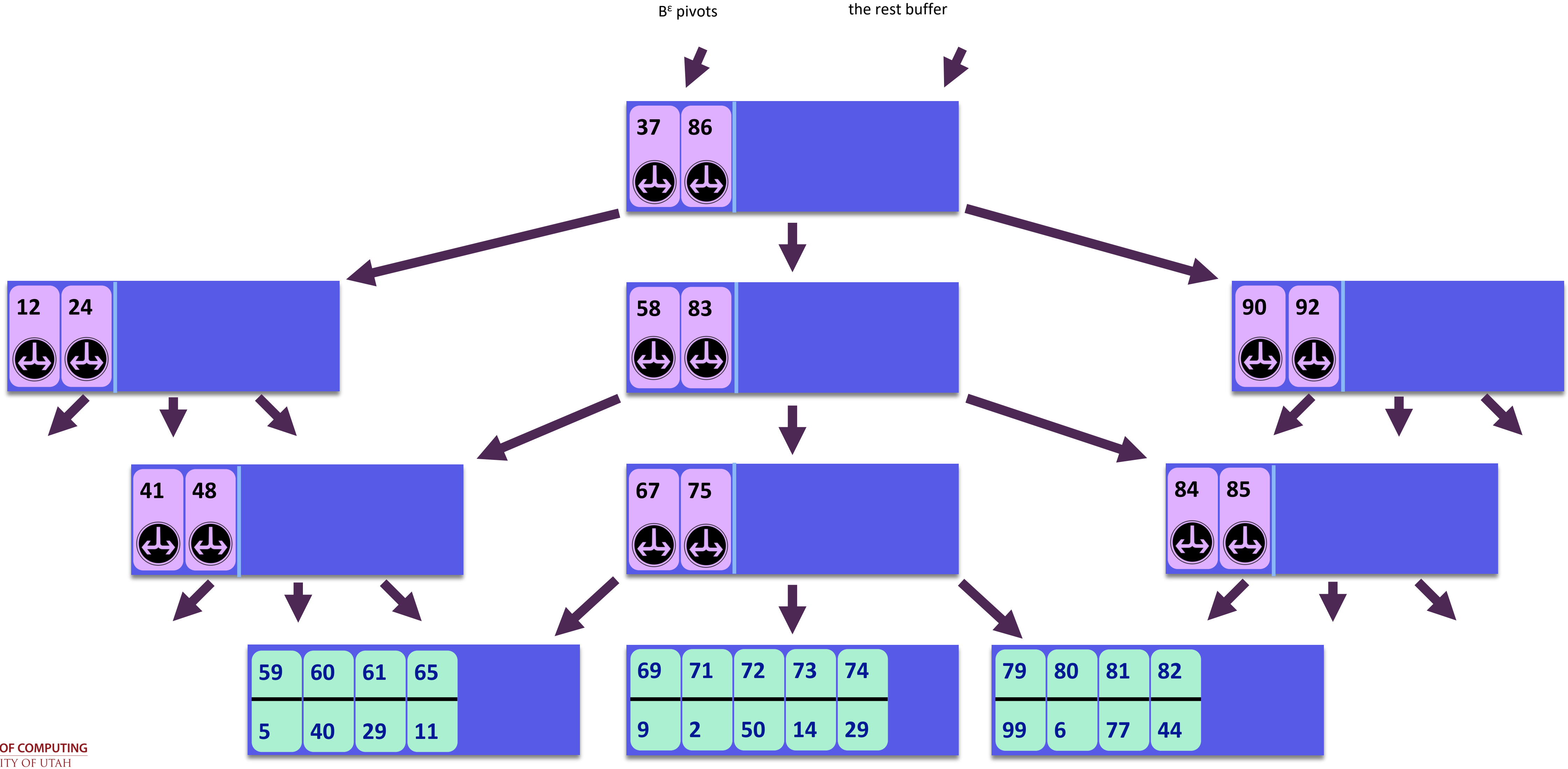
$$\mathit{Insertion\ Cost} \leq O(\log_B N)$$

$$\mathit{Lookup\ Cost} \leq O(\log_B N)$$

B$^\varepsilon$-Trees

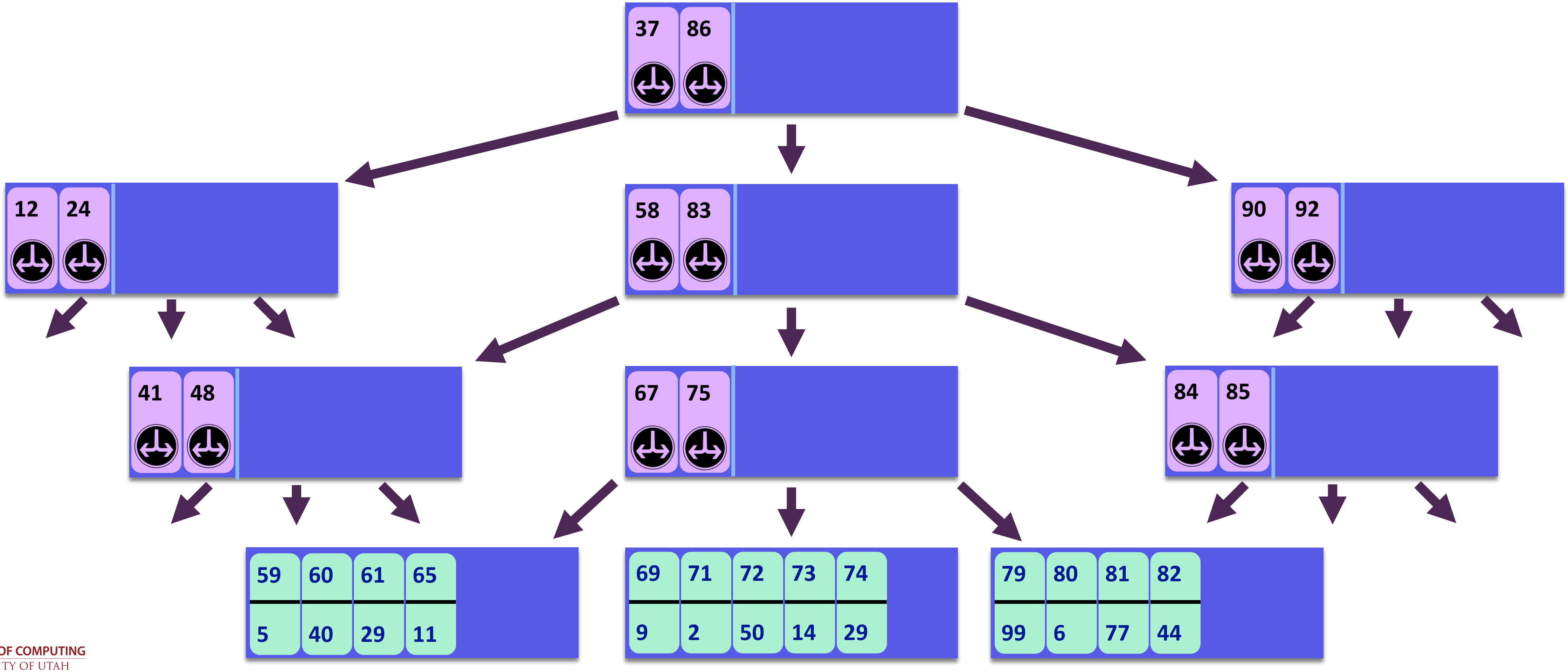
B^{ε}-Trees

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



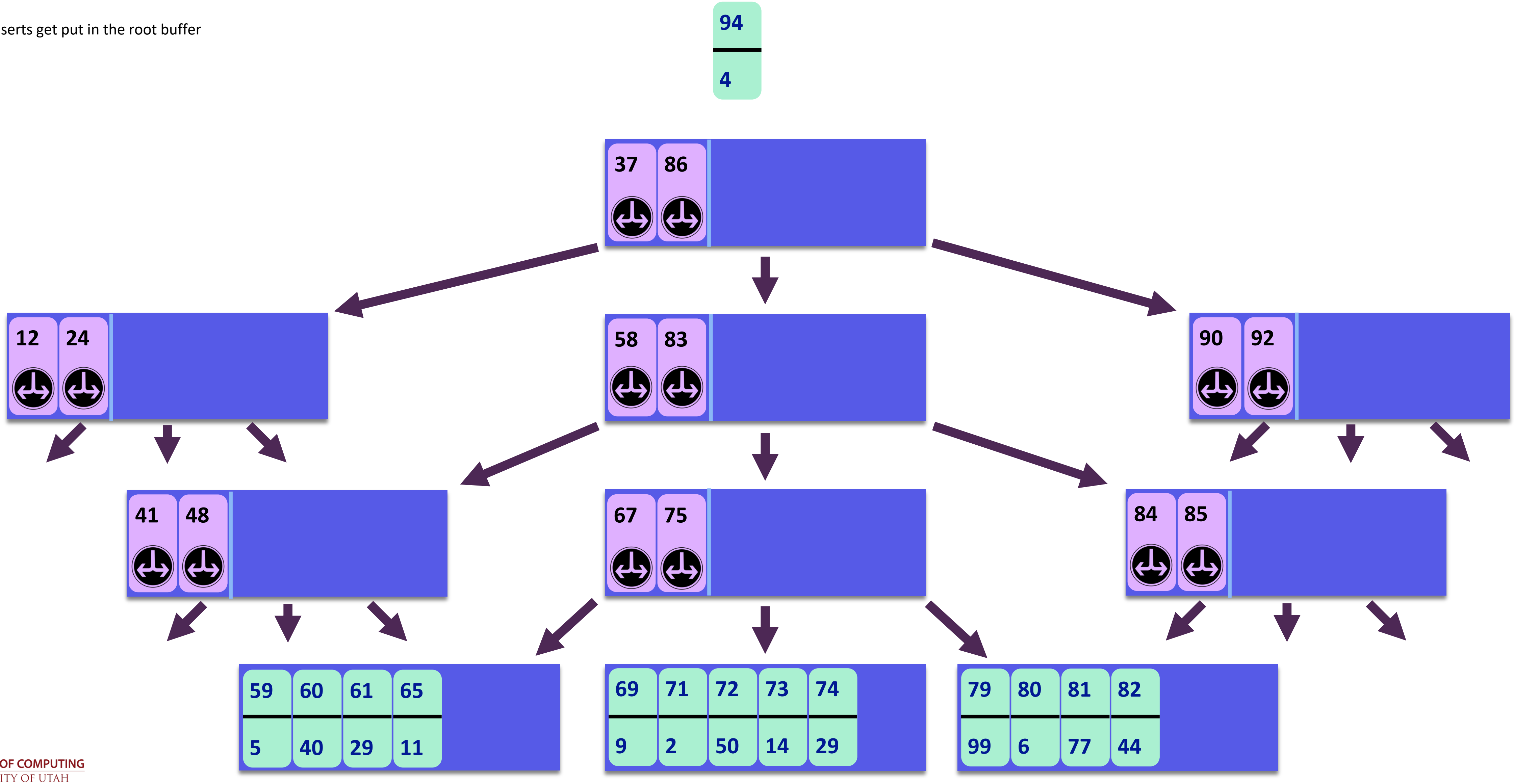
B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



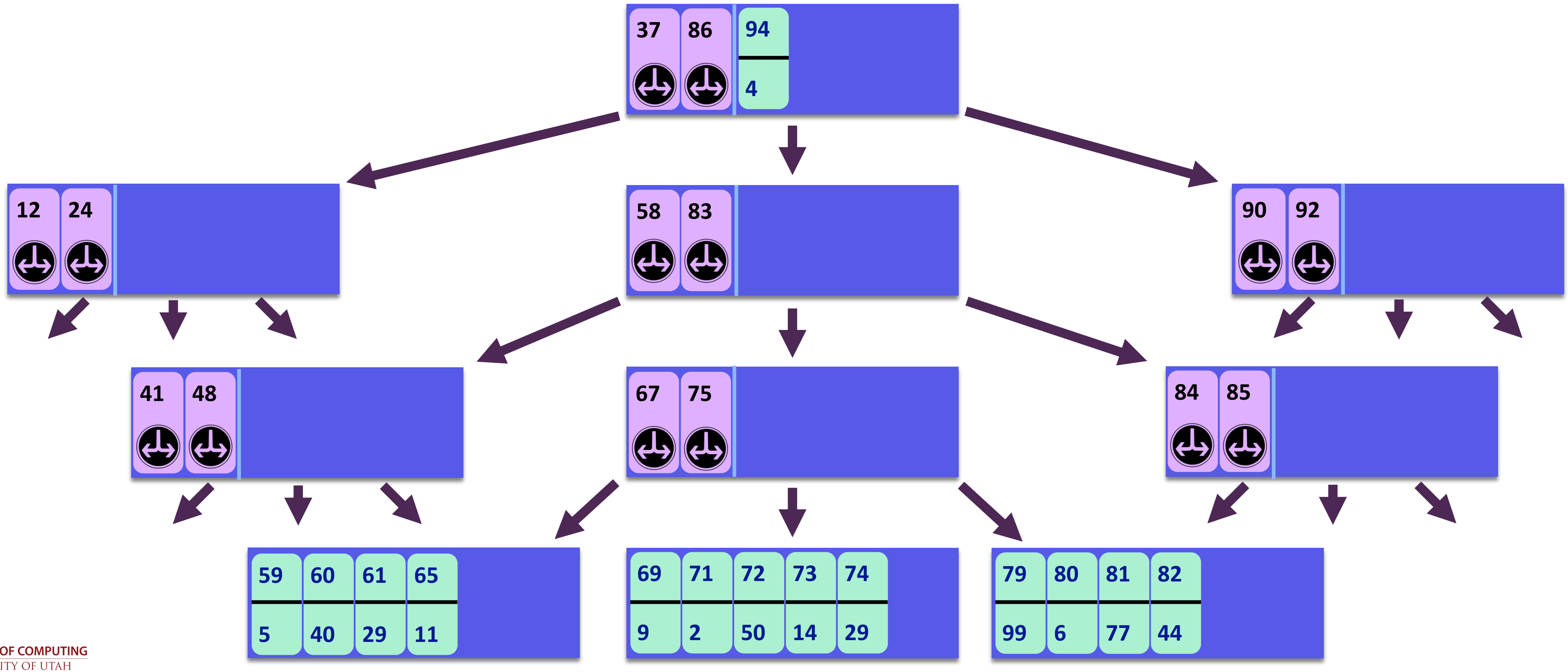
B^{ε}-Trees

Inserts get put in the root buffer



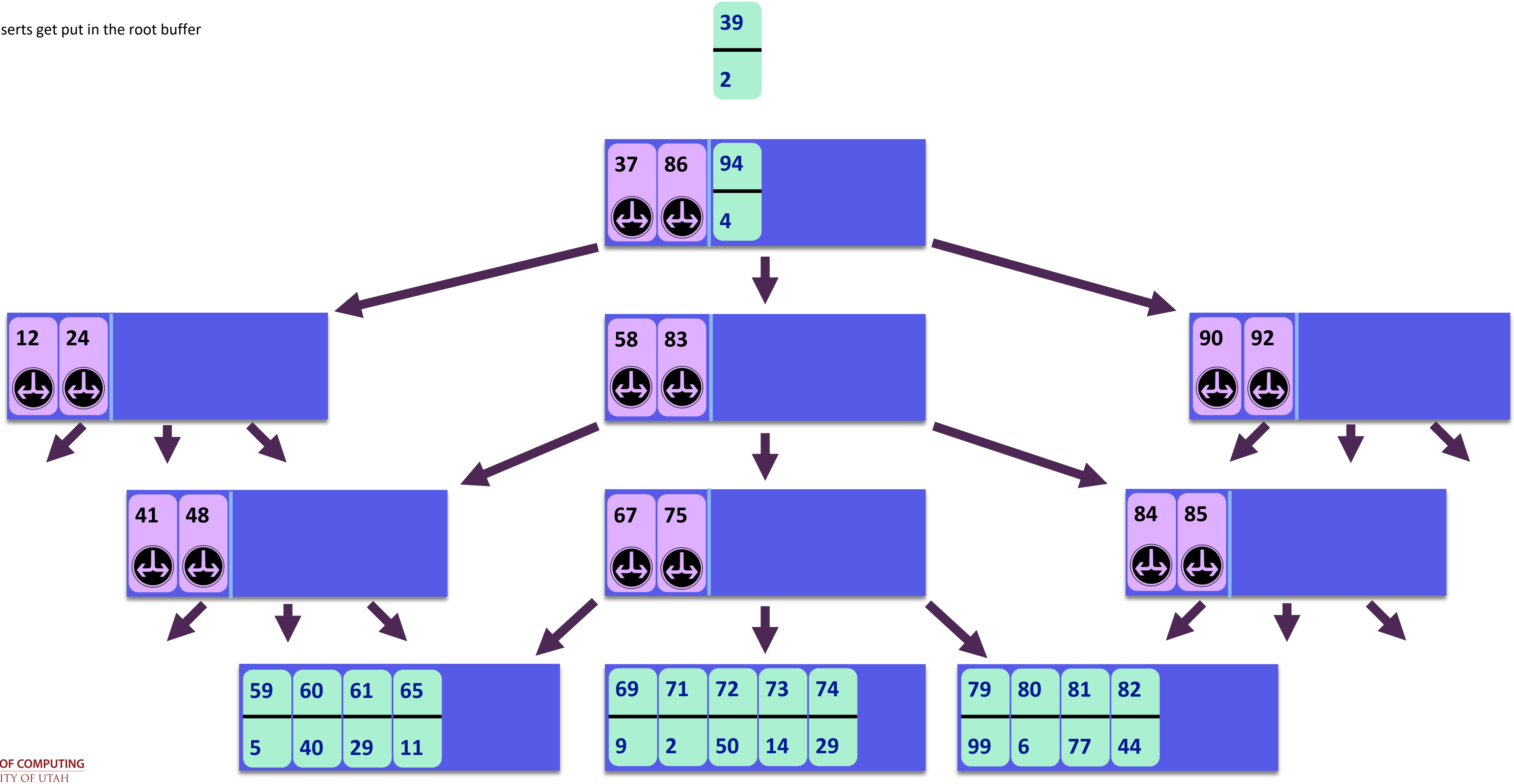
B^{ε}-Trees

Inserts get put in the root buffer



B^{ε}-Trees

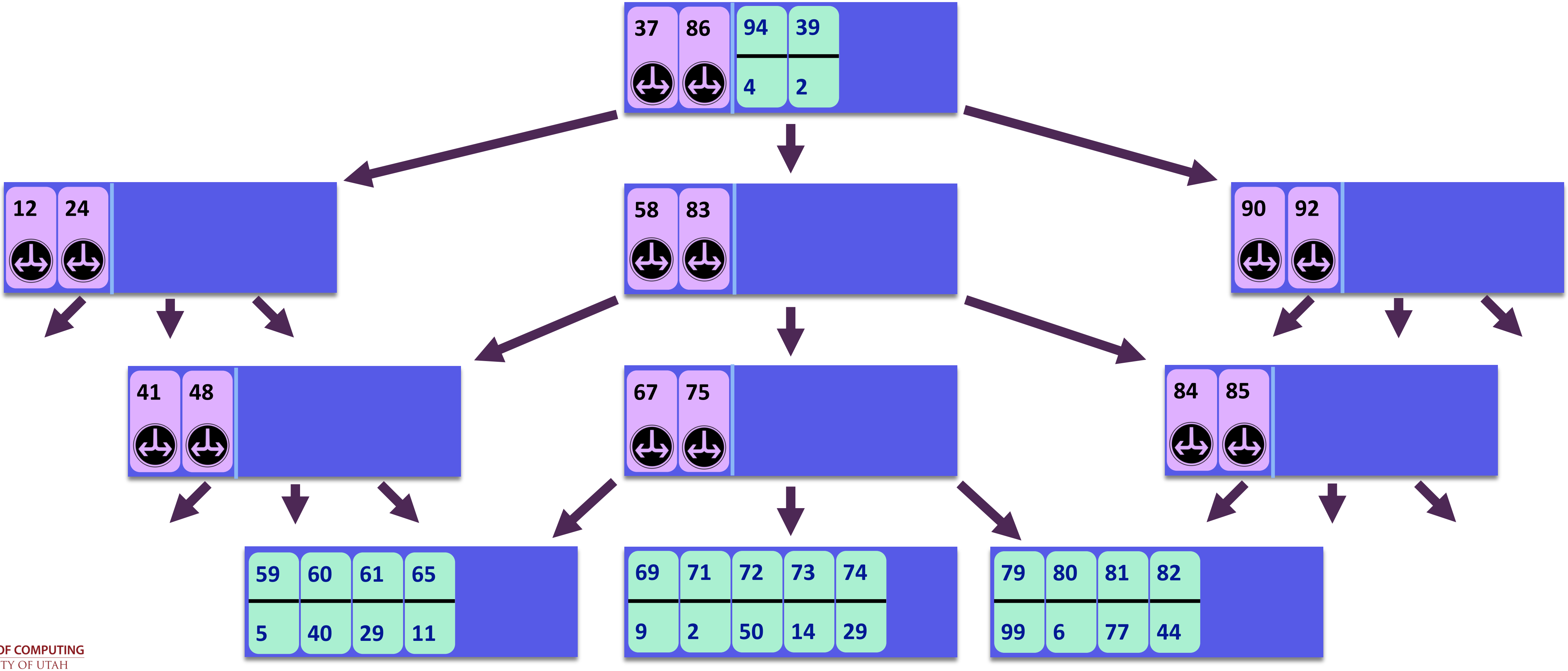
Inserts get put in the root buffer



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

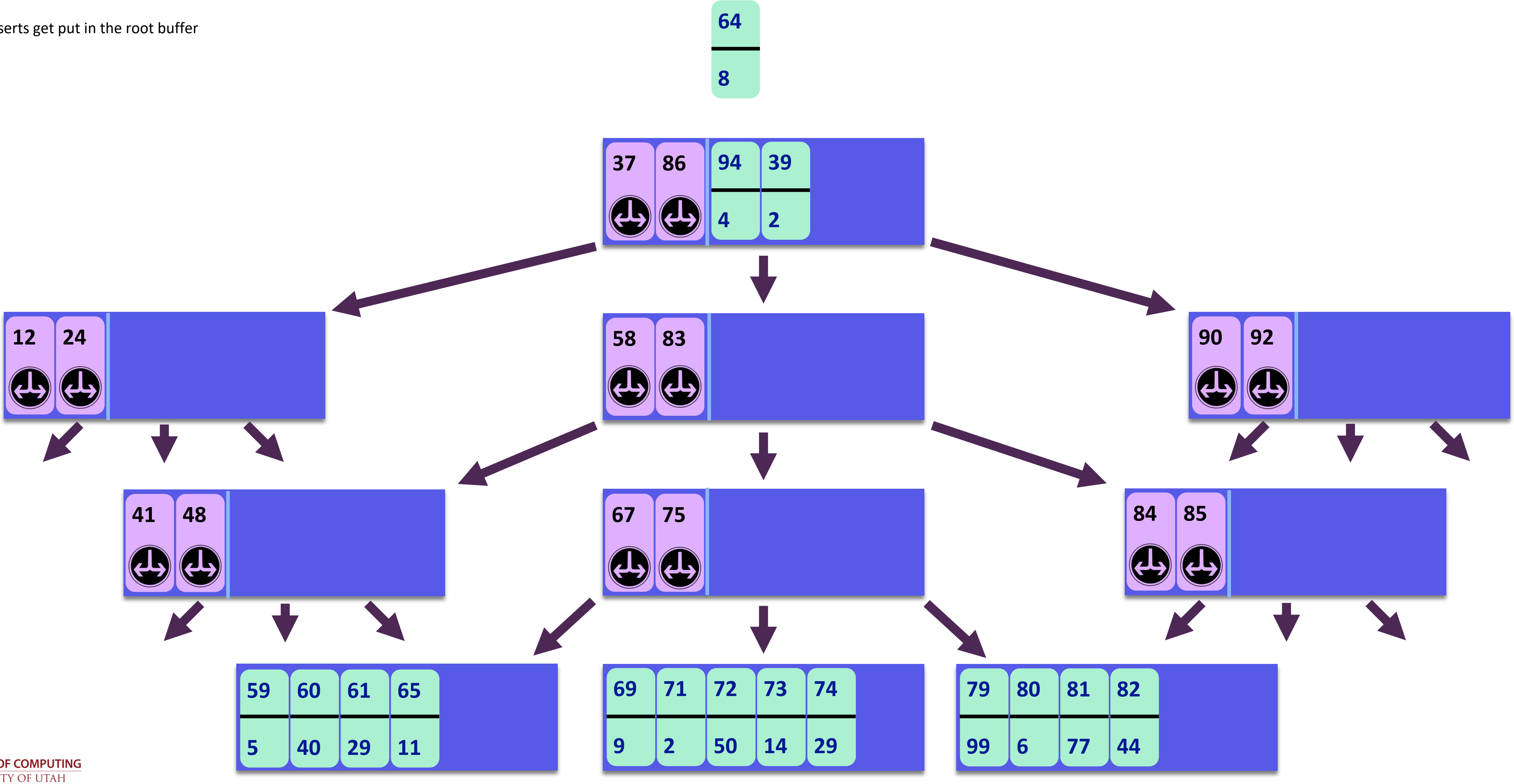
B^{ε}-Trees

Inserts get put in the root buffer



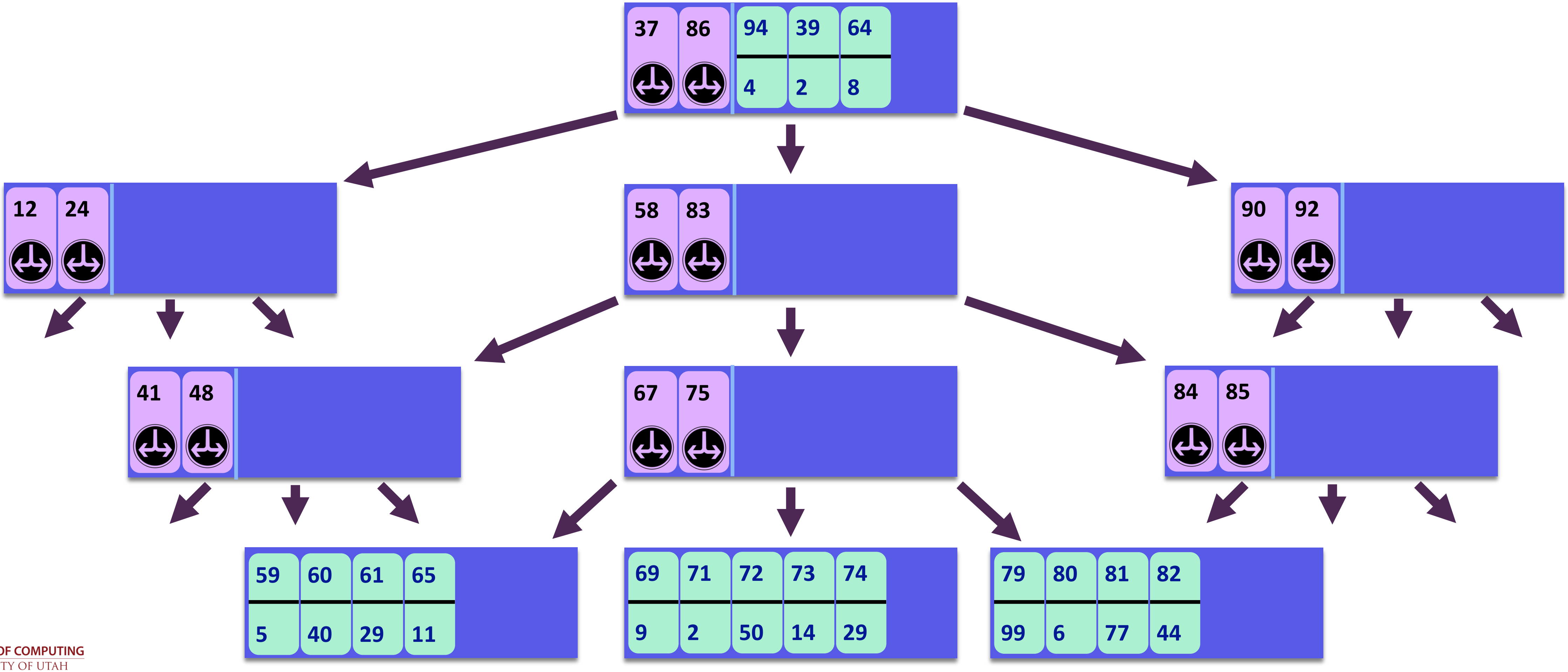
B^ε-Trees

Inserts get put in the root buffer



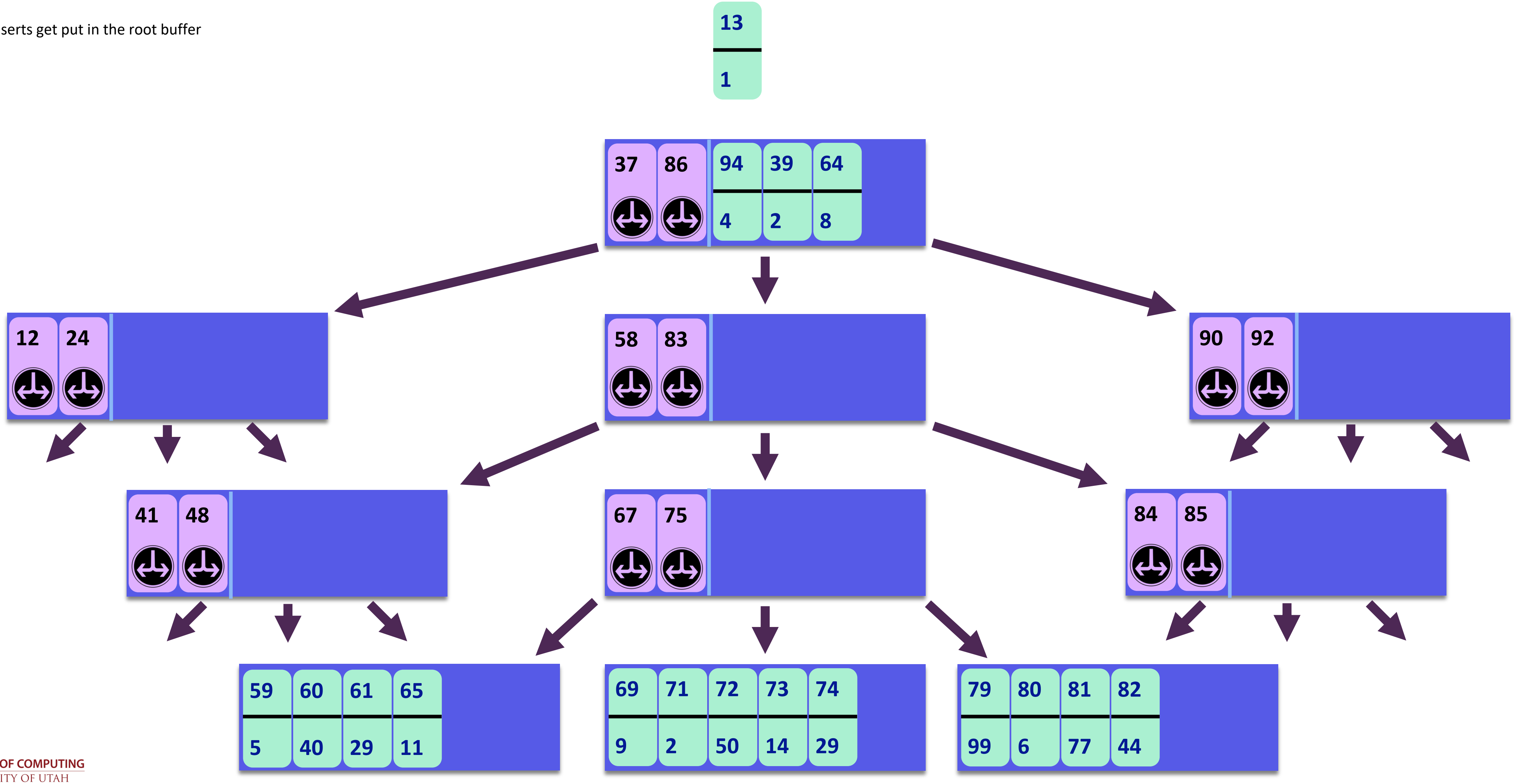
B^{ε}-Trees

Inserts get put in the root buffer



B$^{\varepsilon}$-Trees

Inserts get put in the root buffer

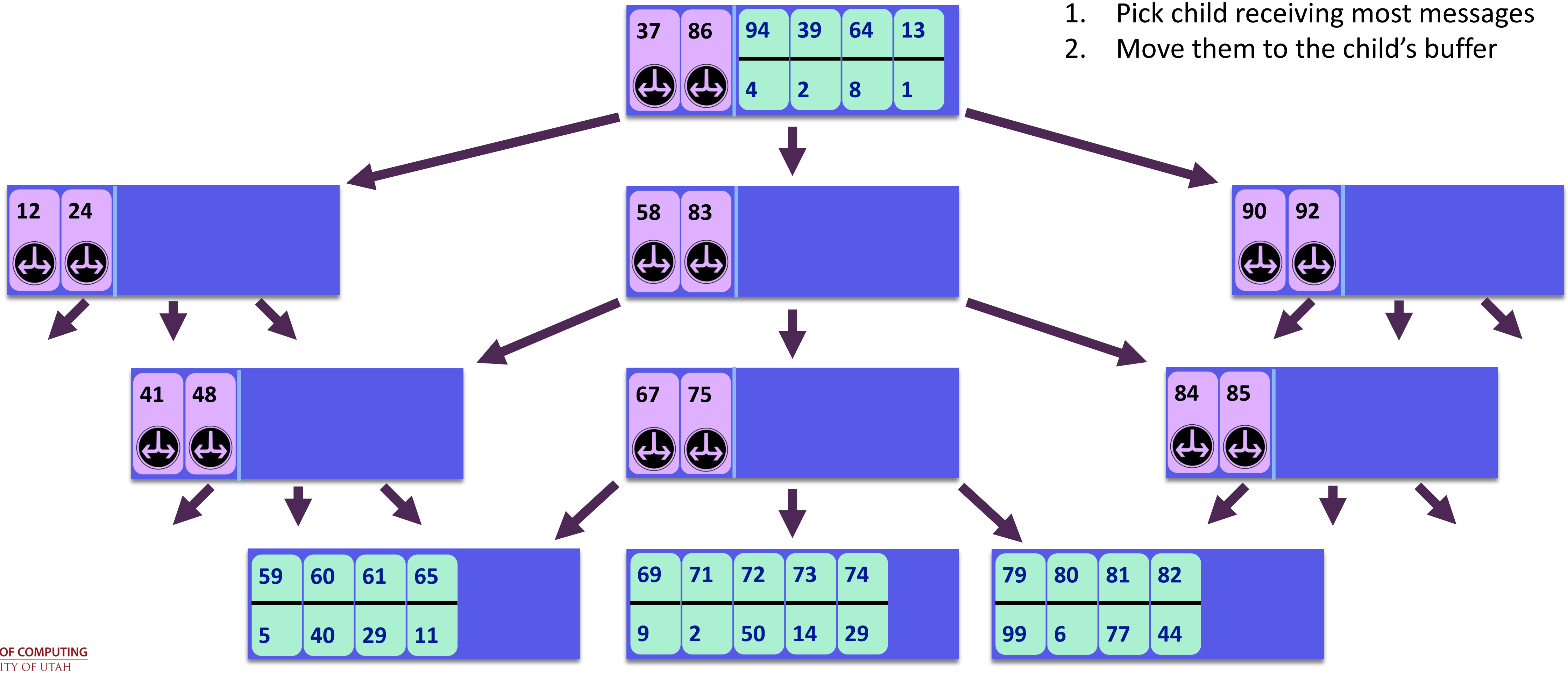


B^{ε}-Trees

Inserts get put in the root buffer

When a buffer is full:

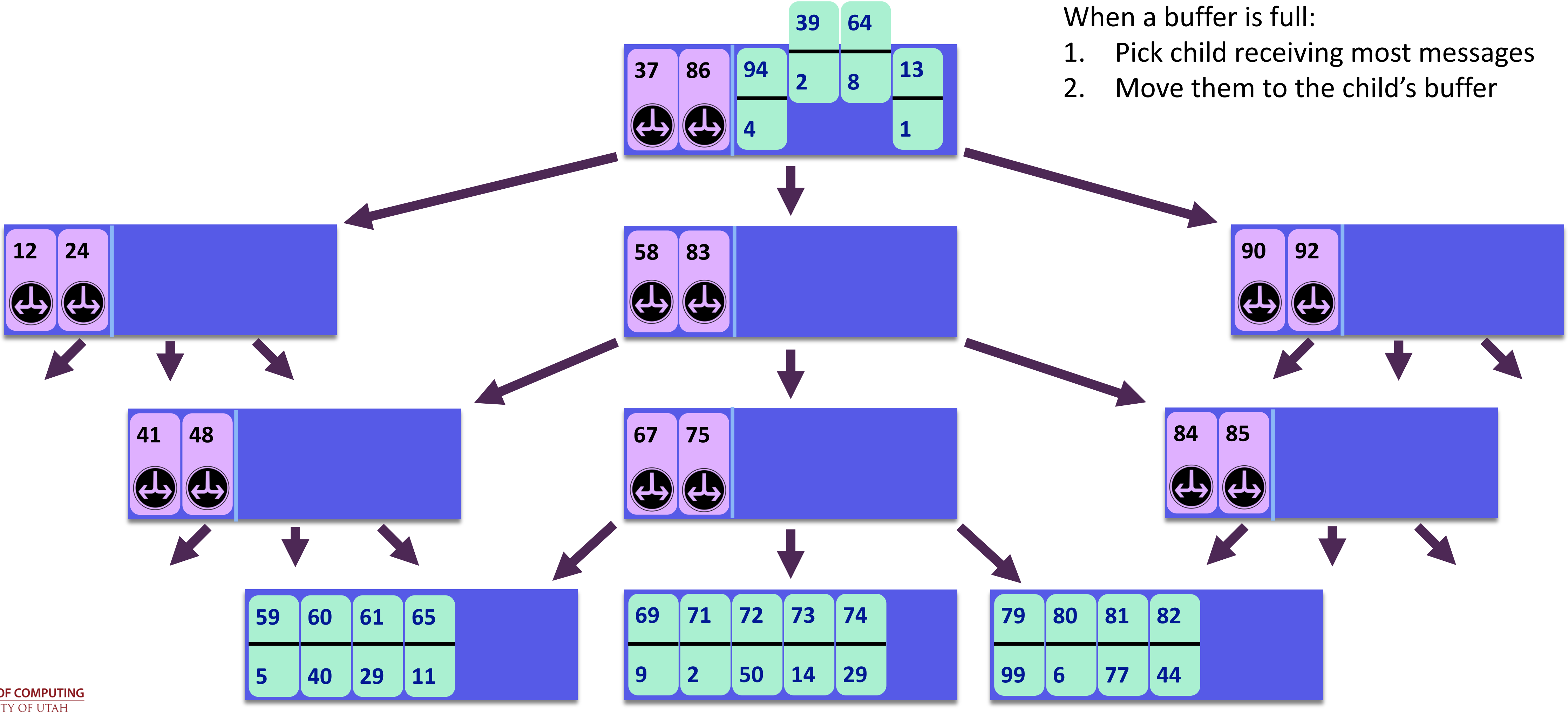
1. Pick child receiving most messages
2. Move them to the child's buffer



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B^{ε}-Trees

Inserts get put in the root buffer

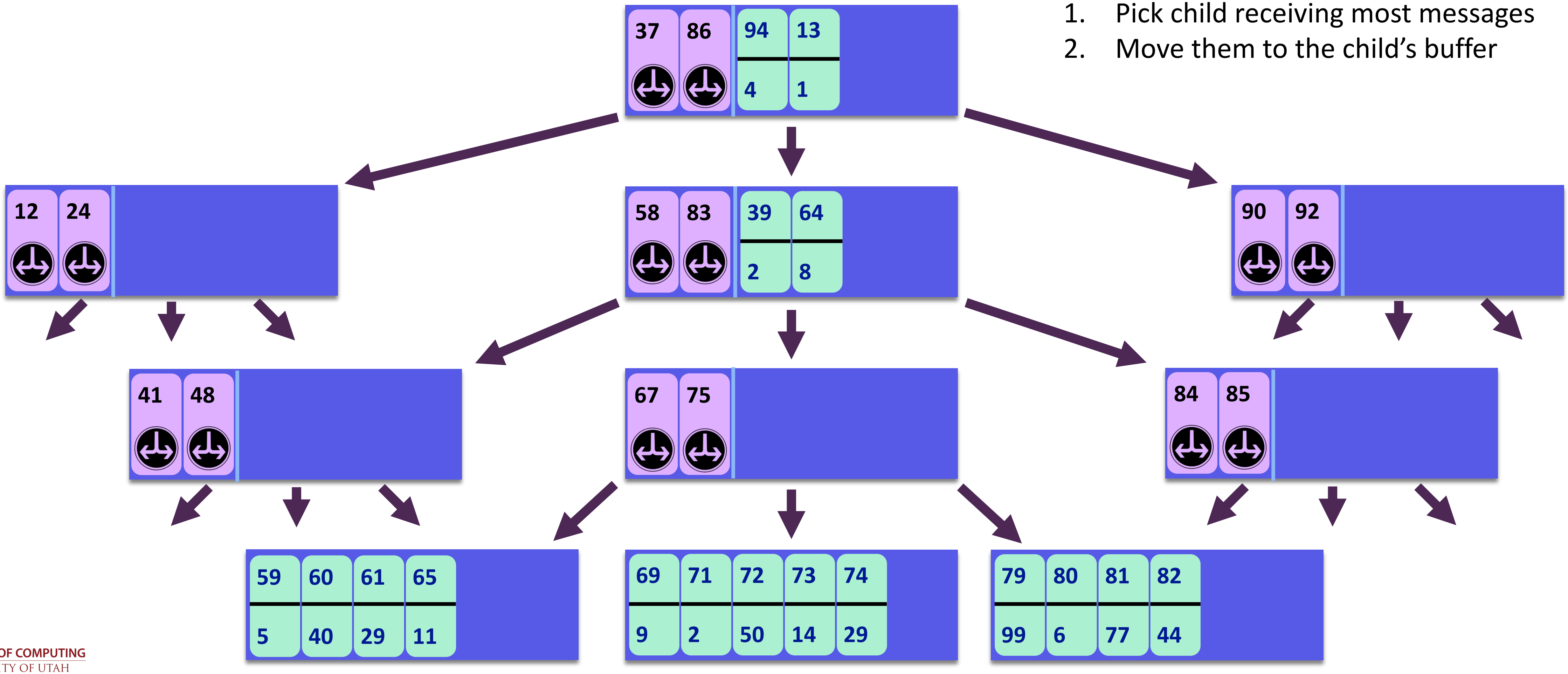


When a buffer is full:

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

B^{ε}-Trees

Inserts get put in the root buffer

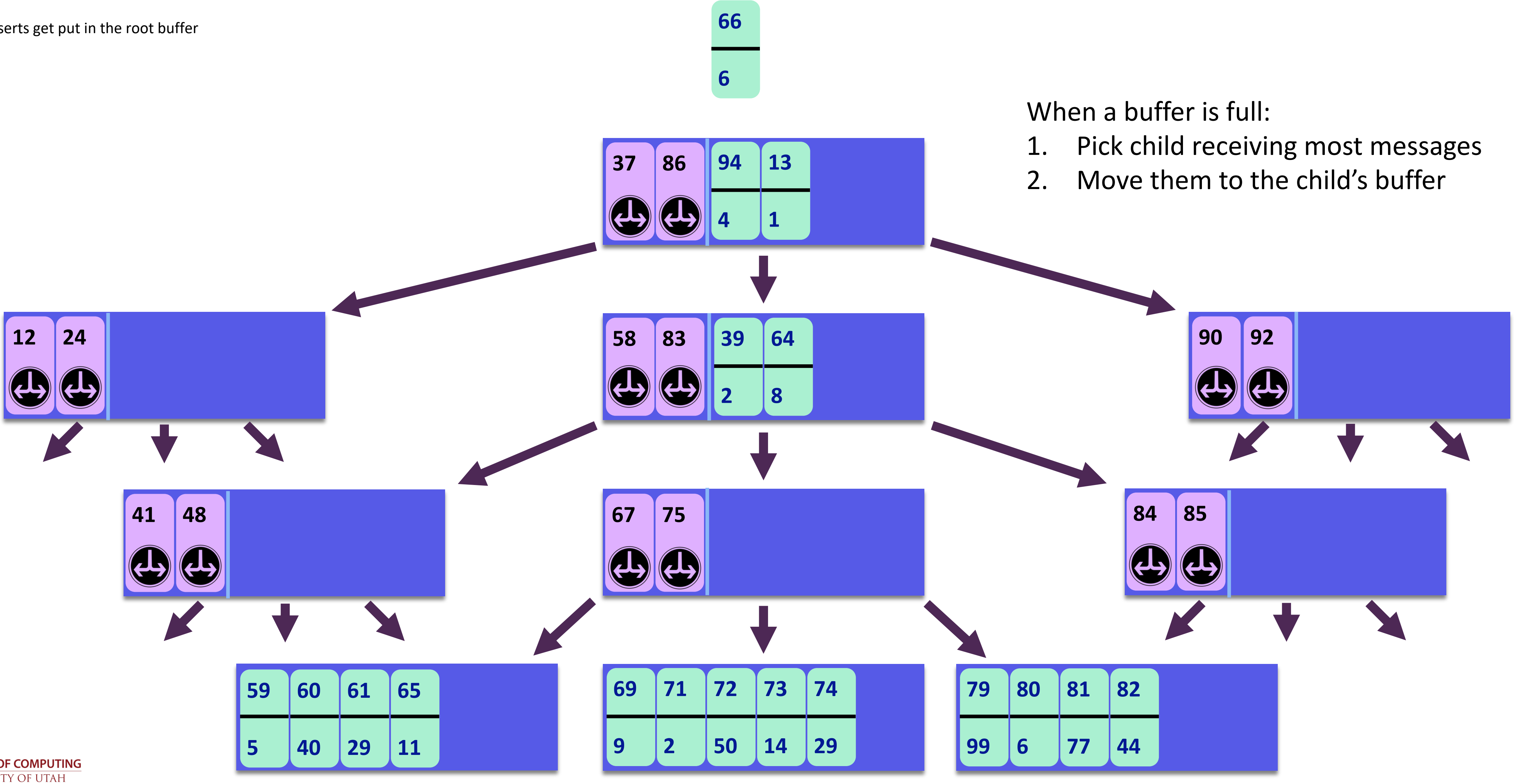


When a buffer is full:

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

B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



When a buffer is full:

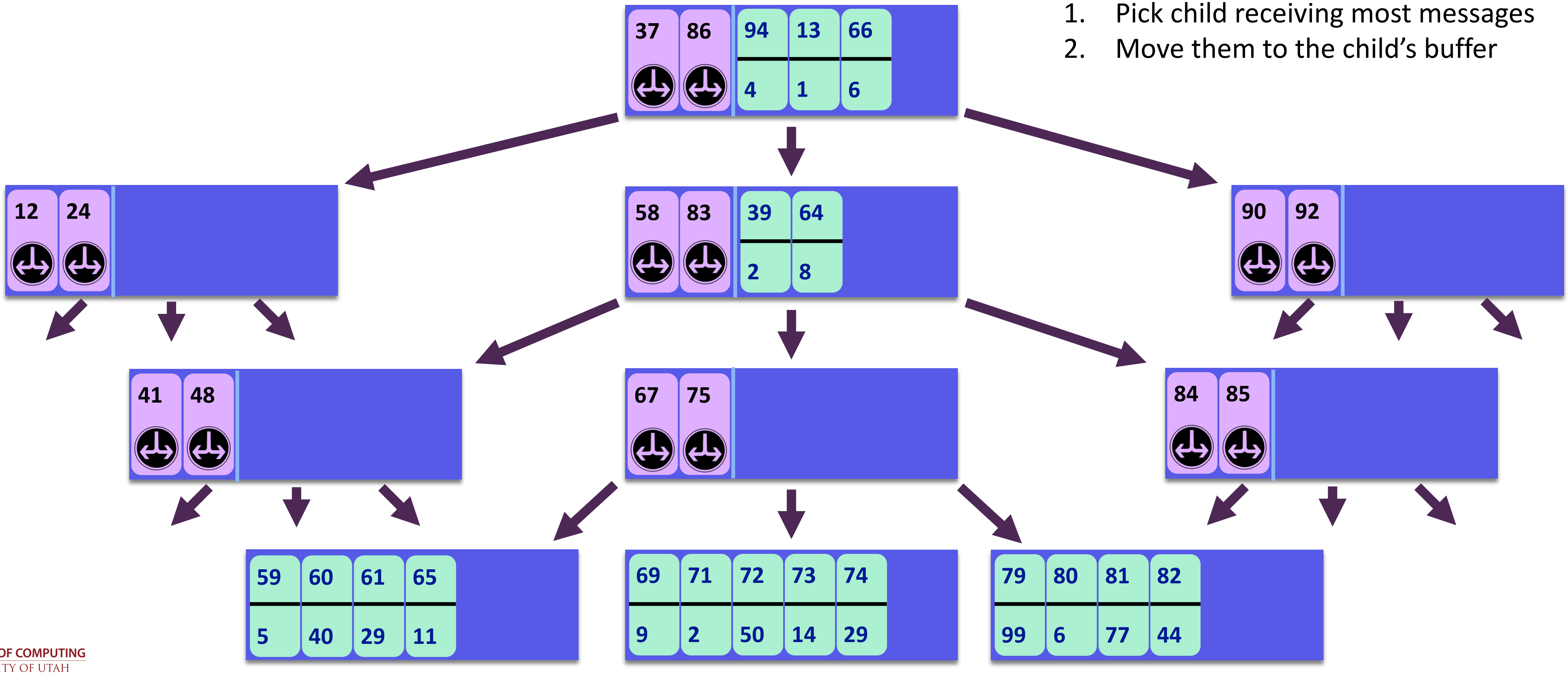
1. Pick child receiving most messages
2. Move them to the child's buffer

B-Trees

Inserts get put in the root buffer

When a buffer is full:

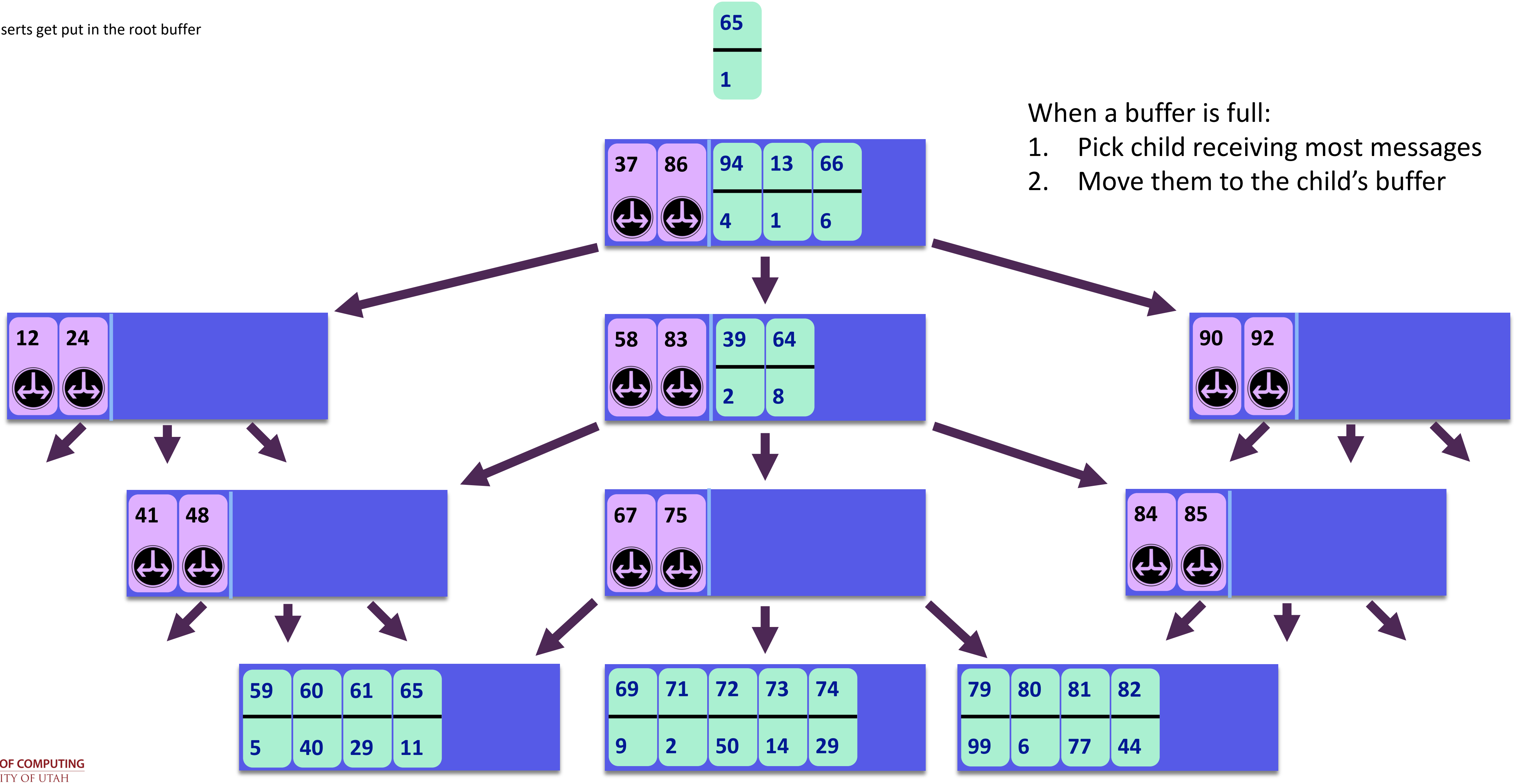
1. Pick child receiving most messages
2. Move them to the child's buffer



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B$^{\varepsilon}$-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

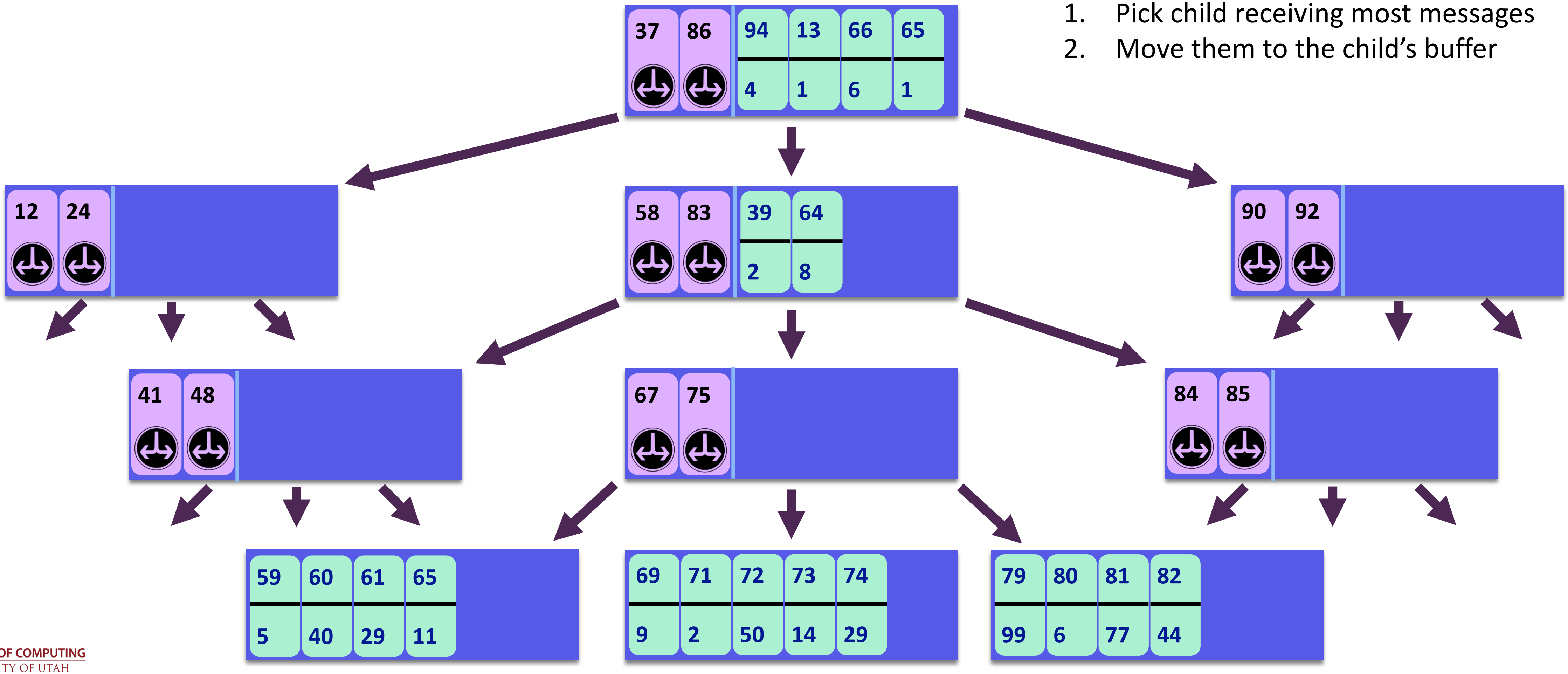
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B^{ε}-Trees

Inserts get put in the root buffer

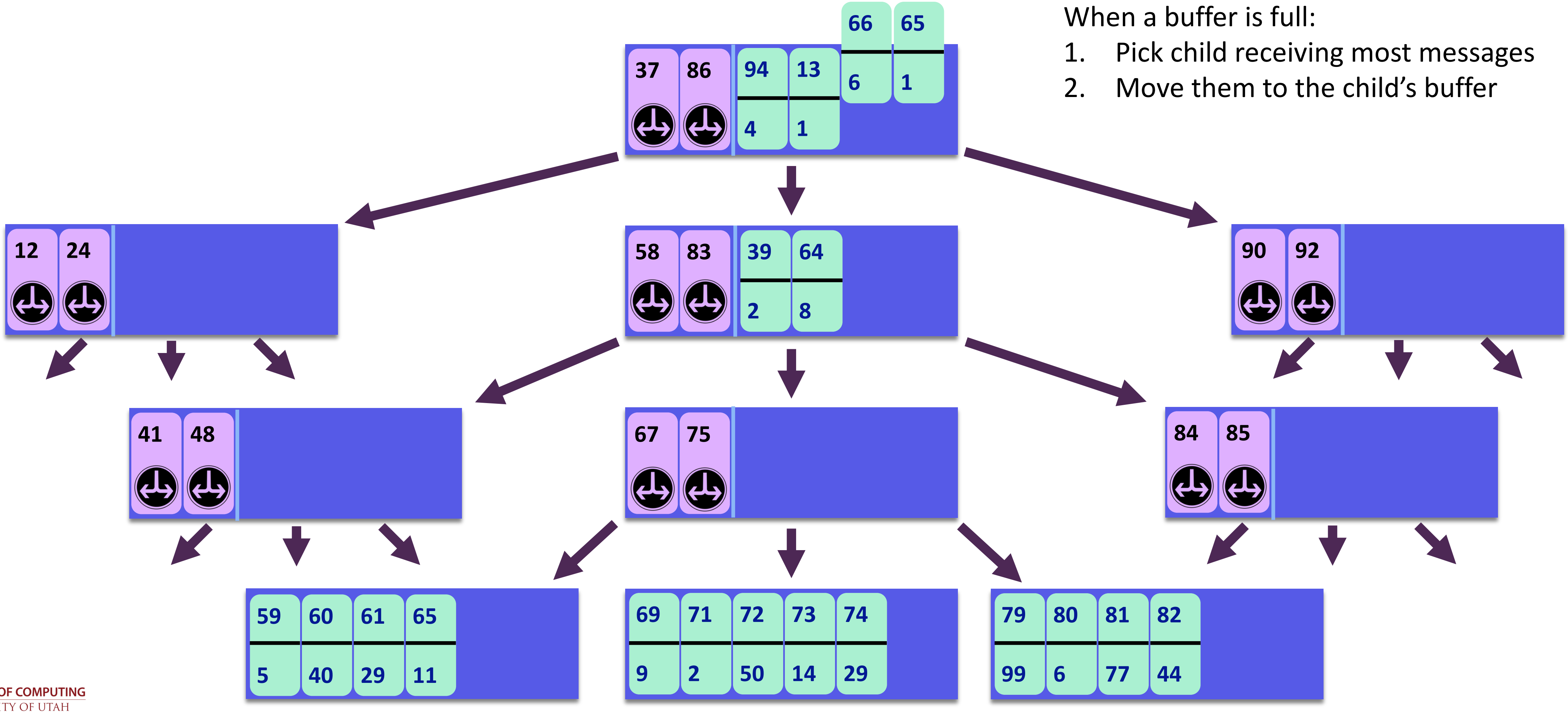
When a buffer is full:

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



B^{ε}-Trees

Inserts get put in the root buffer



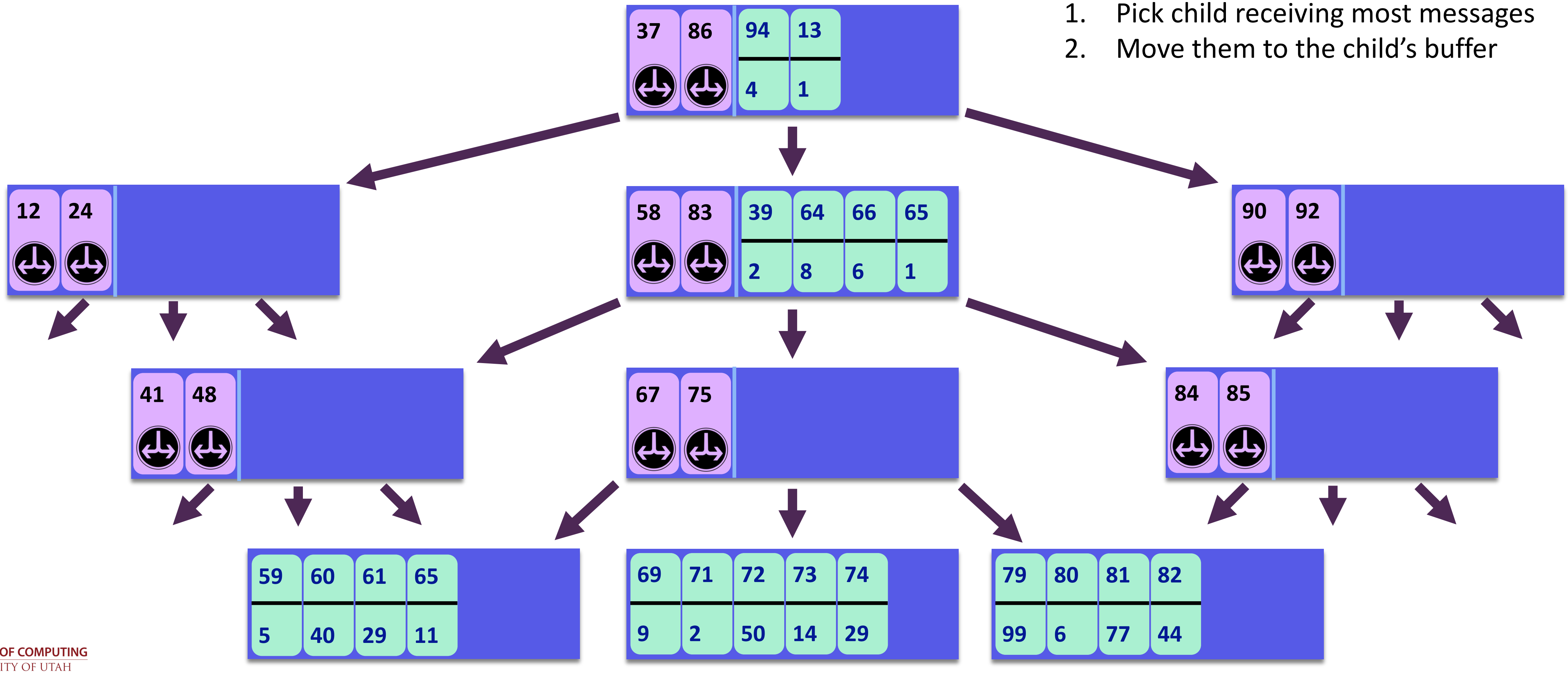
When a buffer is full:

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

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



When a buffer is full:

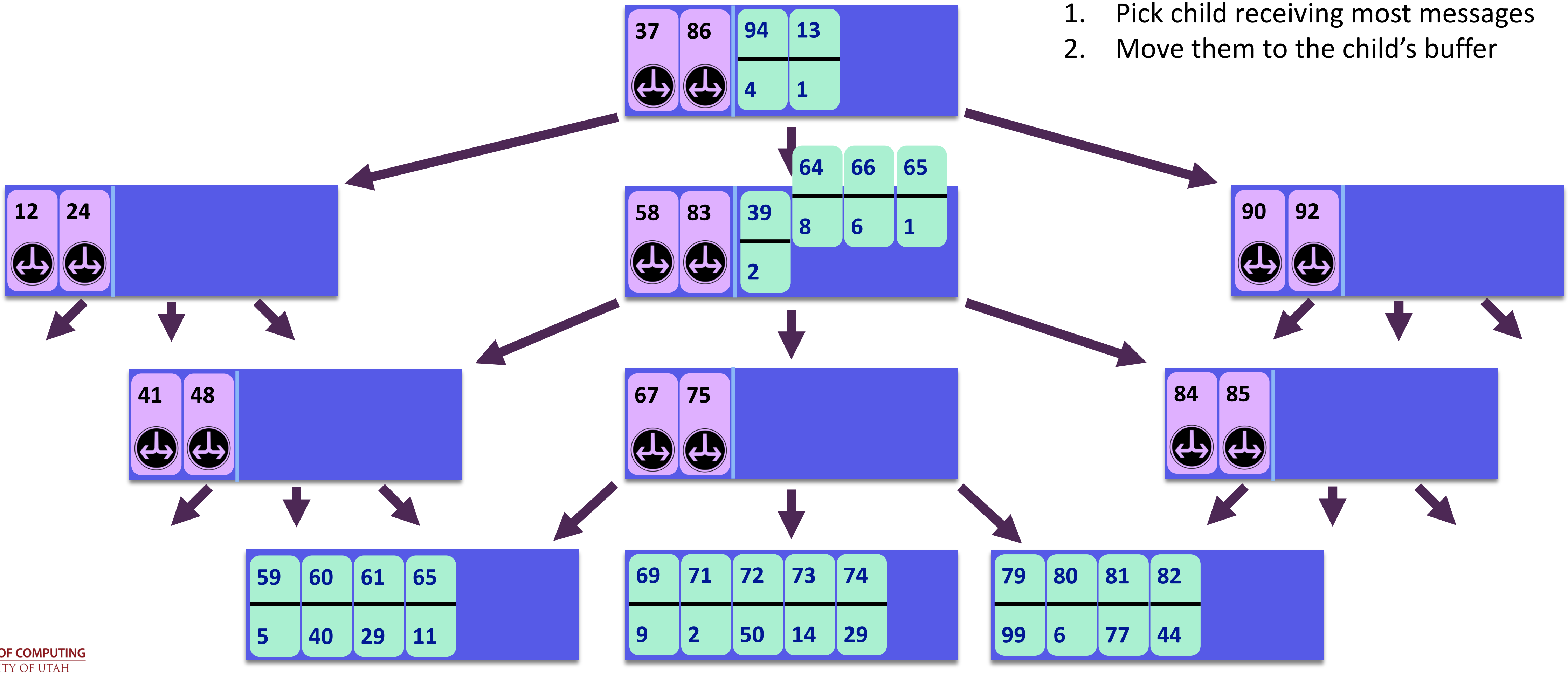
1. Pick child receiving most messages
2. Move them to the child's buffer

B$^{\varepsilon}$-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

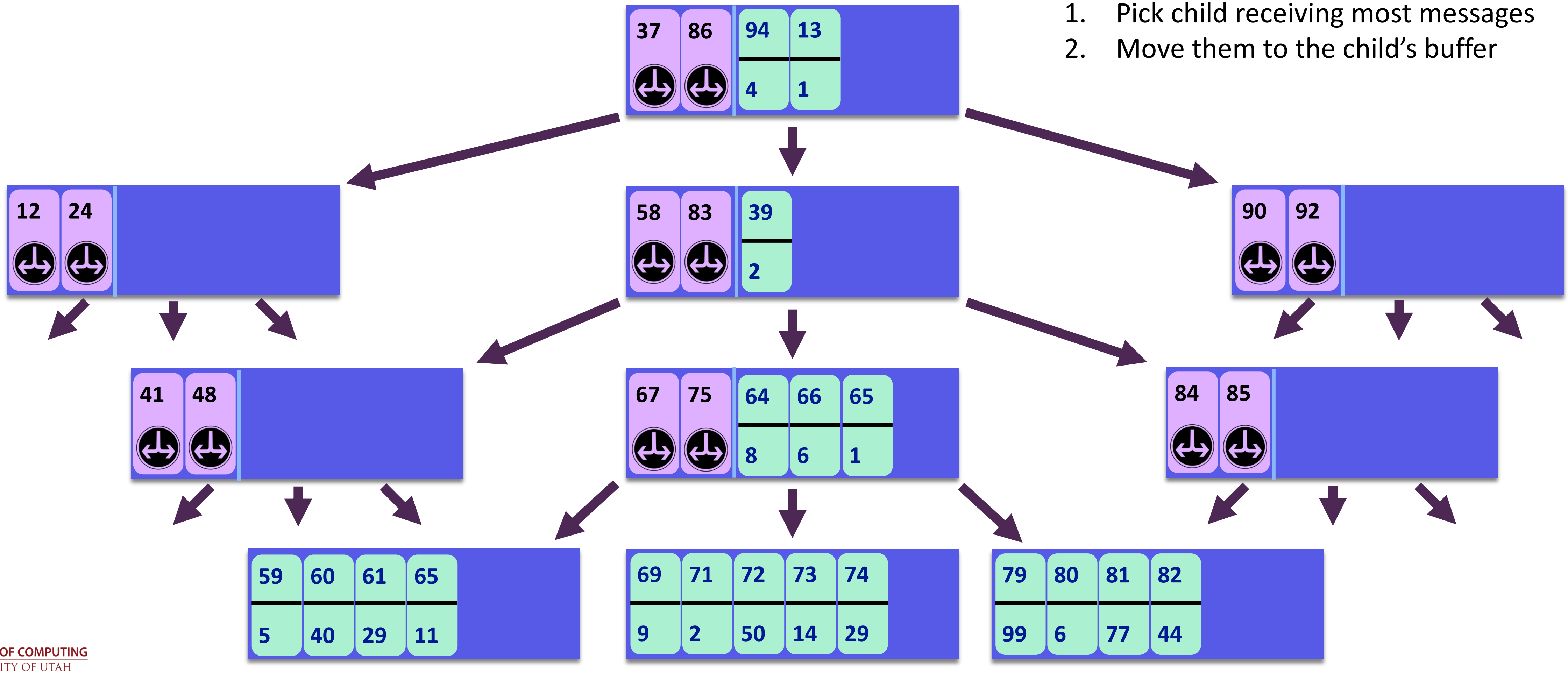


B^{ε}-Trees

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

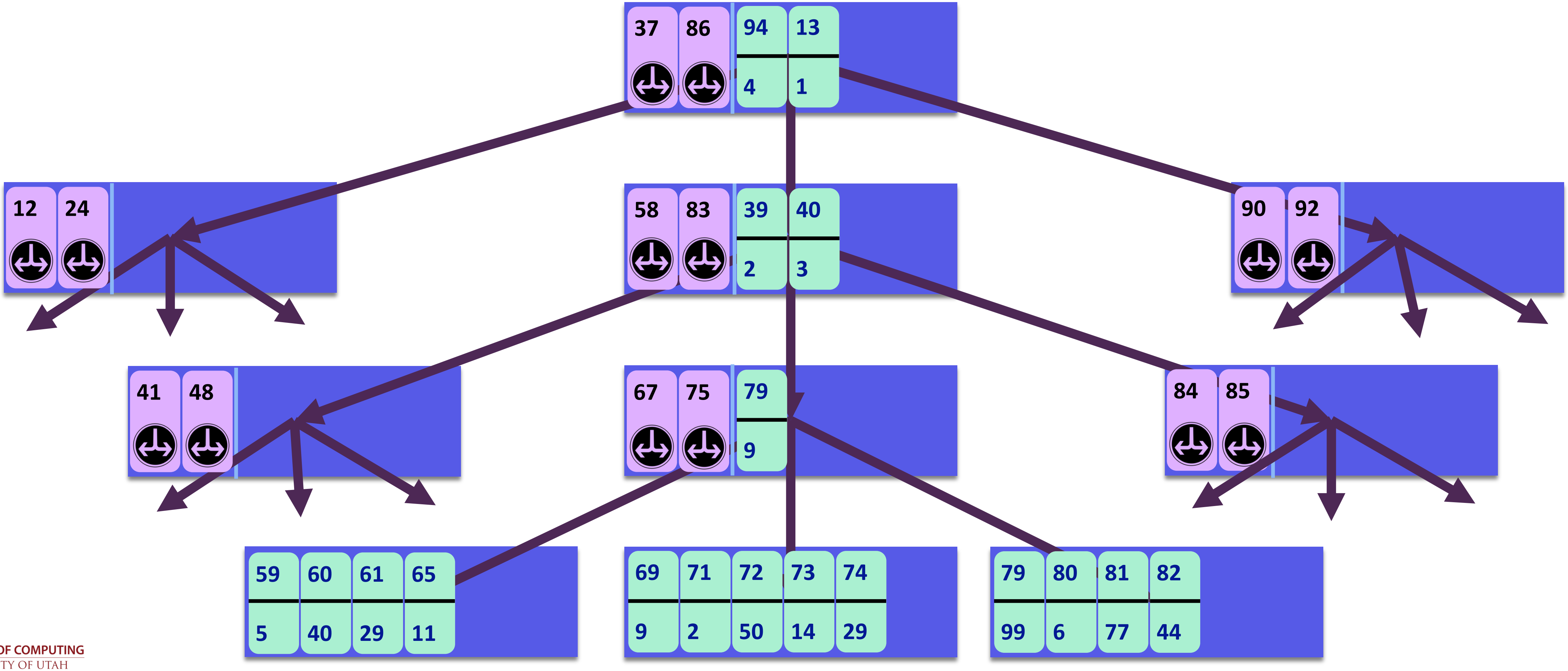


Lookups in B$^{\varepsilon}$-Trees

B^{ε}-Trees

Query(71)

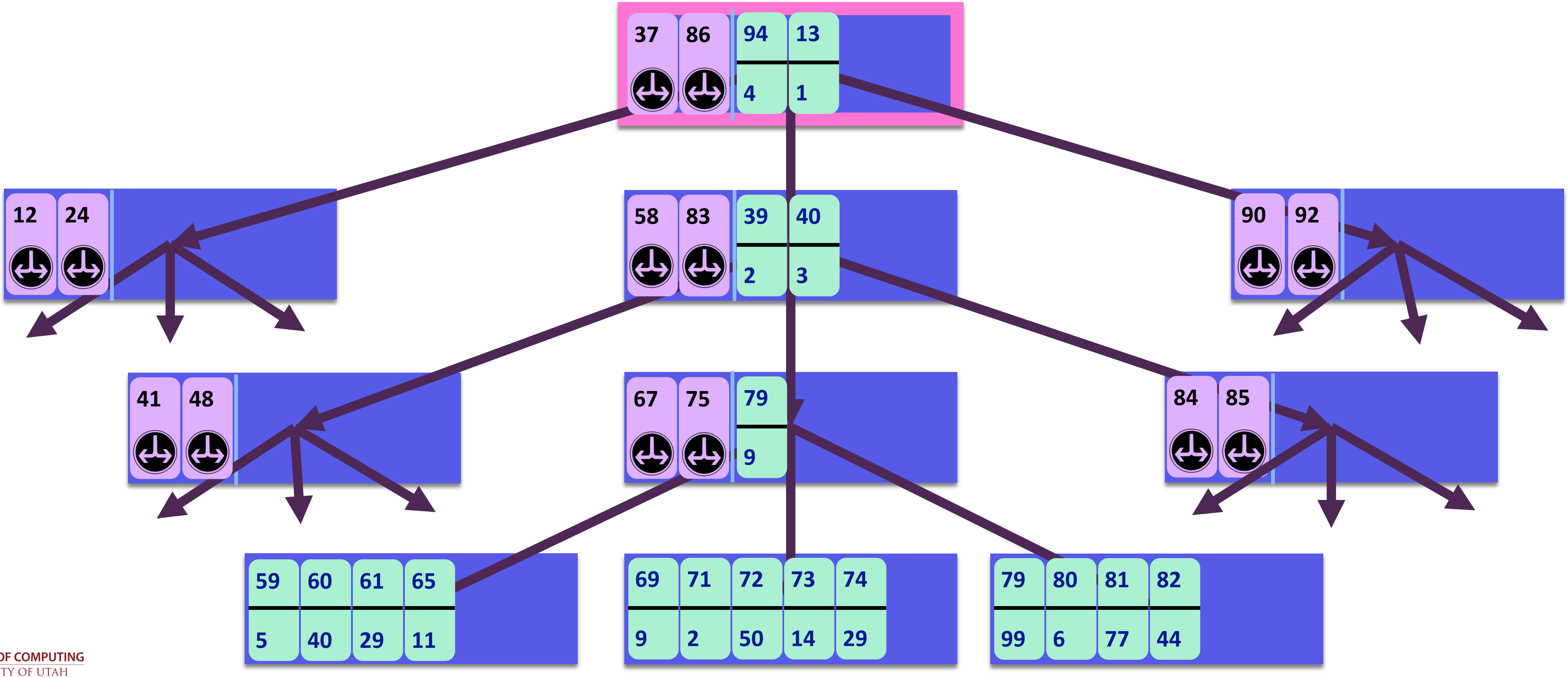
Lookups follow pivots, but check buffers along the way



B$^{\varepsilon}$-Trees

Query(71)

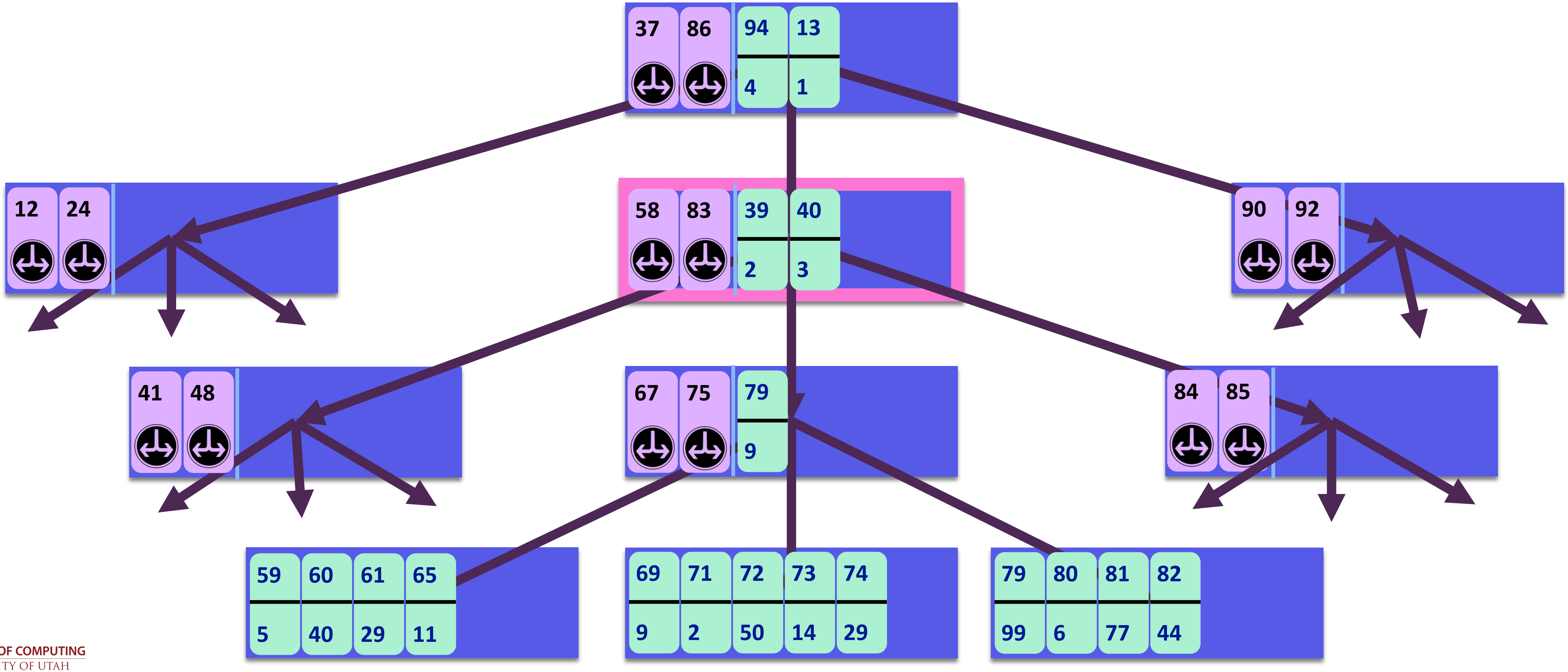
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Query(71)

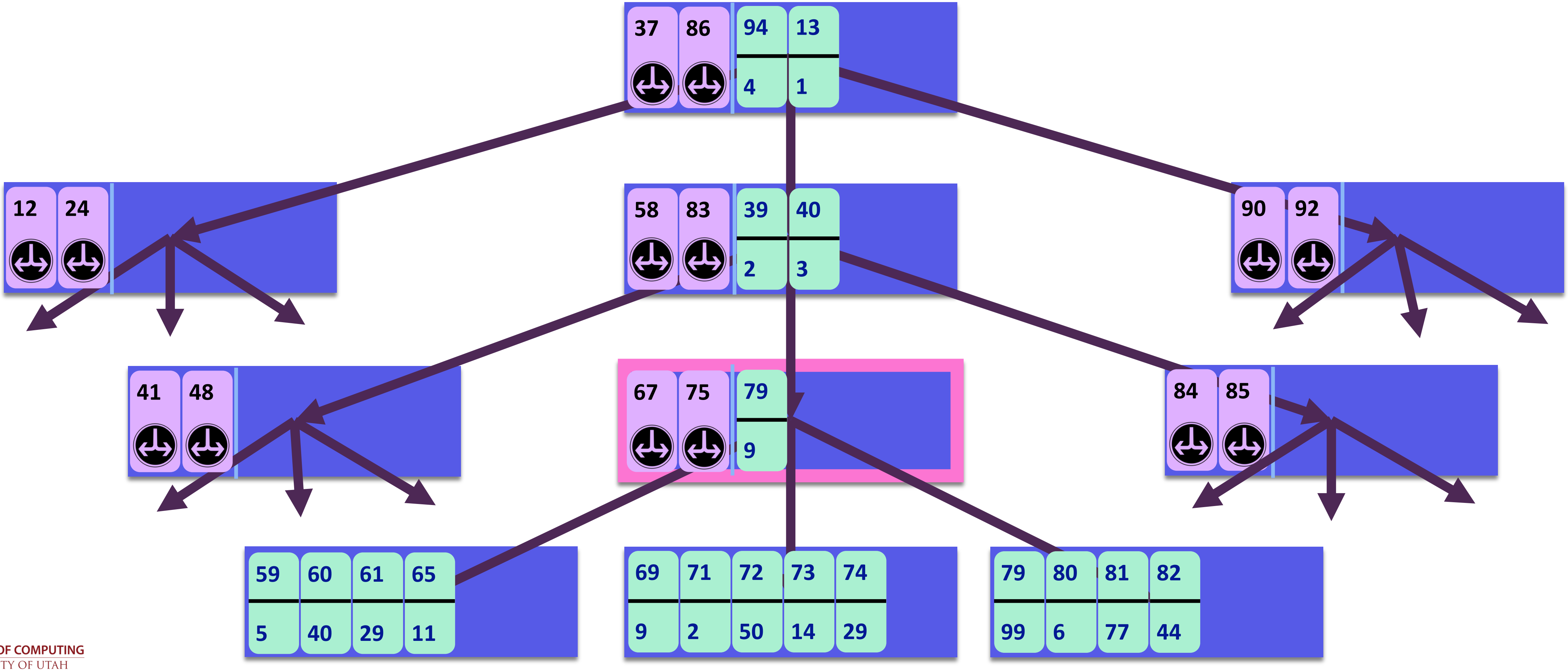
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Query(71)

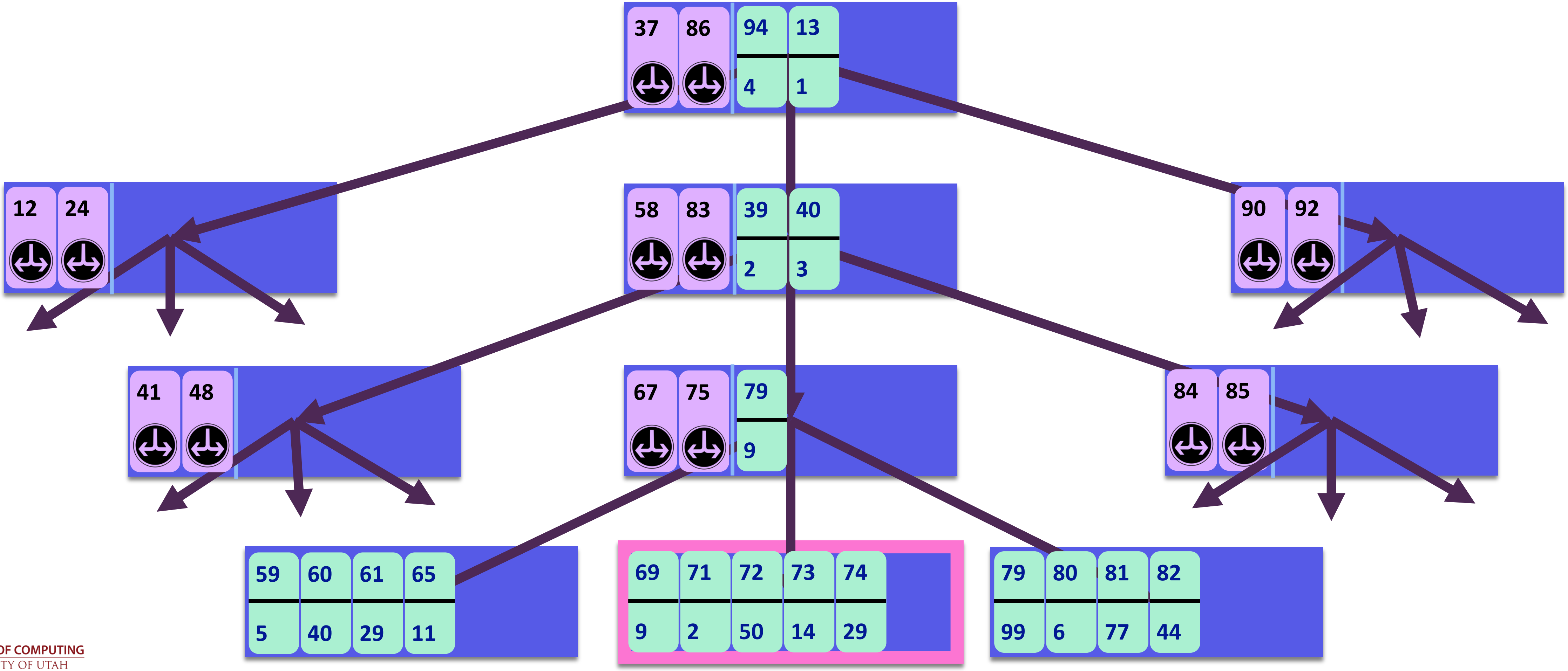
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Lookups follow pivots, but check buffers along the way

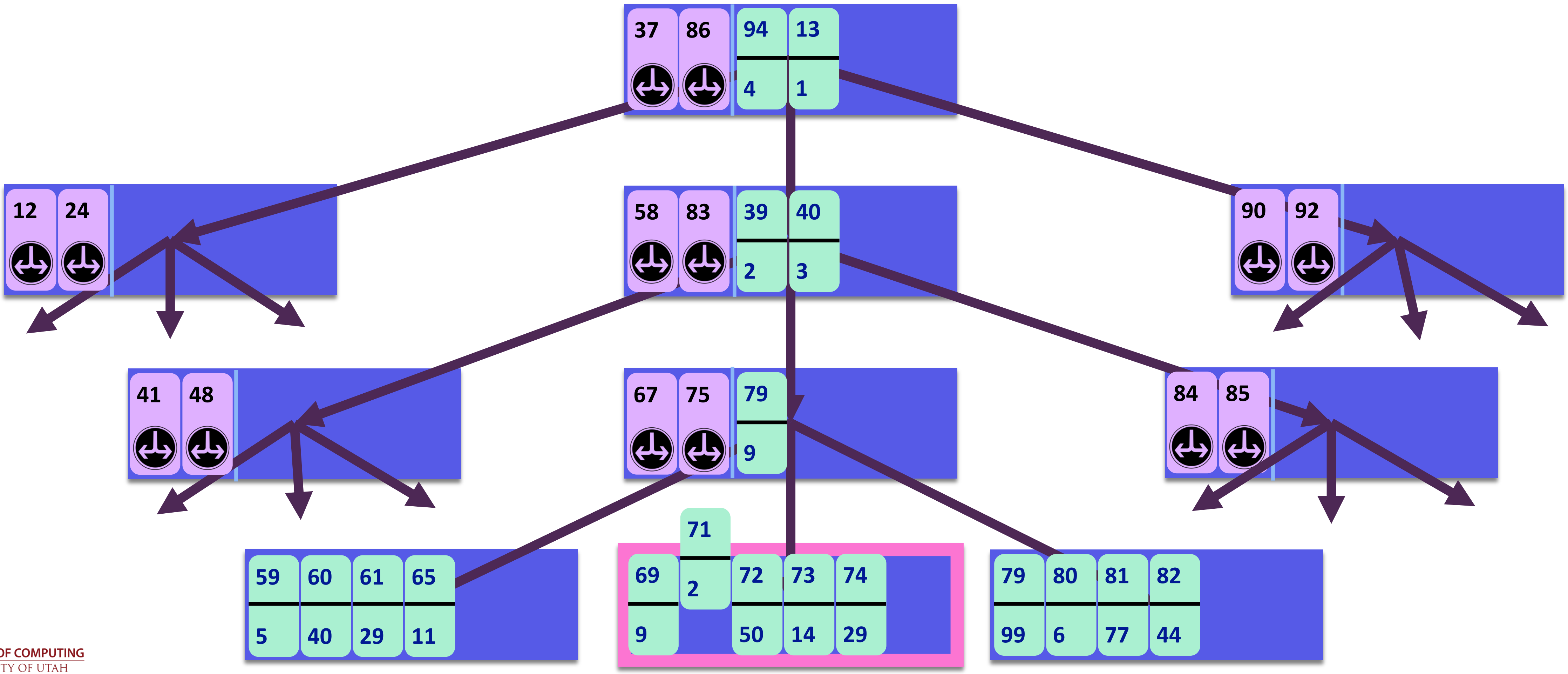
Query(71) → 2



B$^{\varepsilon}$-Trees

Lookups follow pivots, but check buffers along the way

Query(71) ⟶ 2



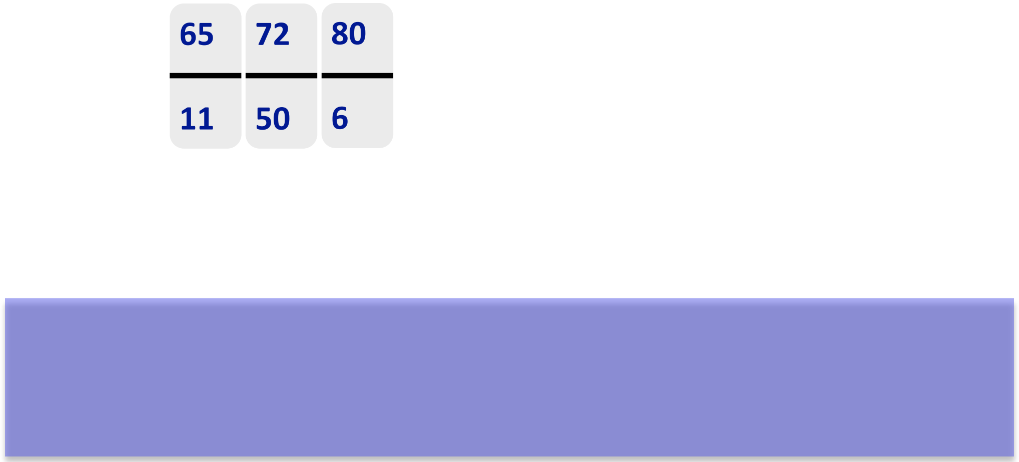
Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

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

(Also most LSMS)

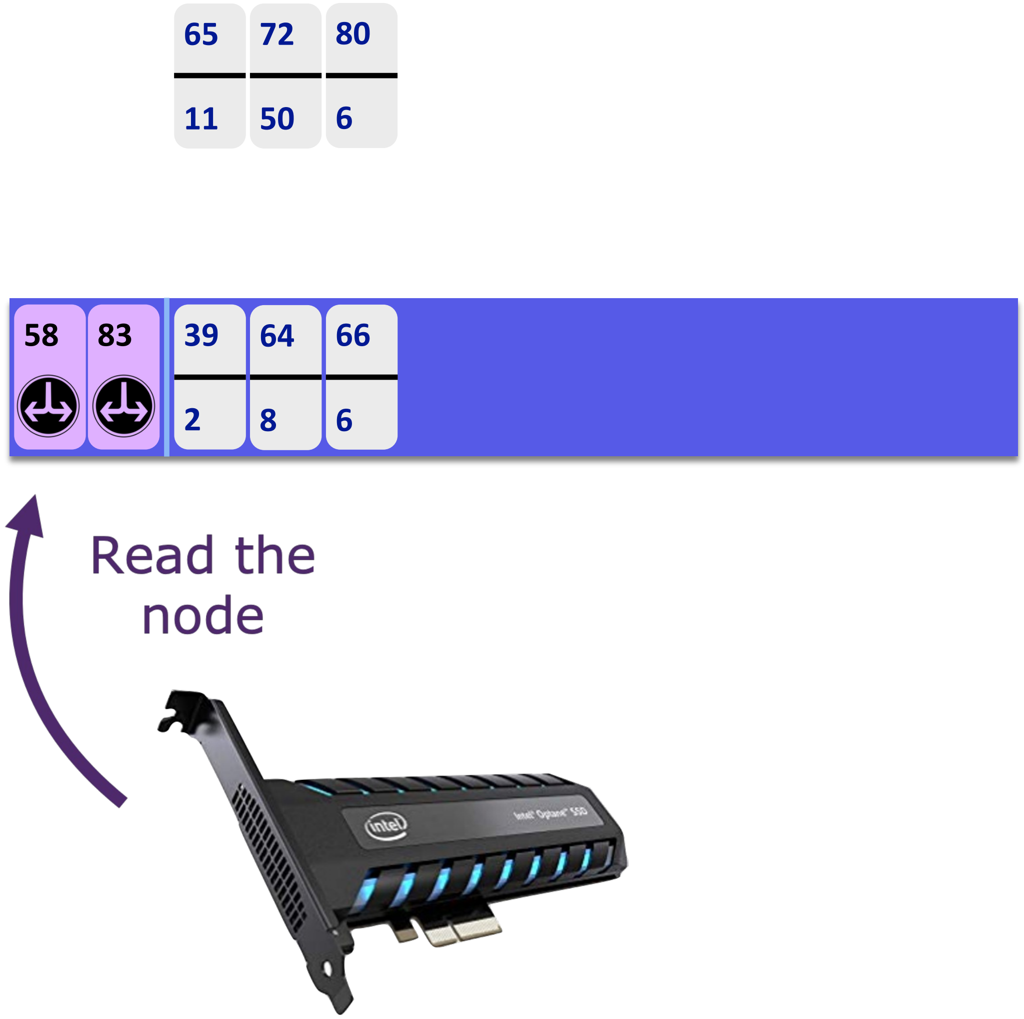
Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



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

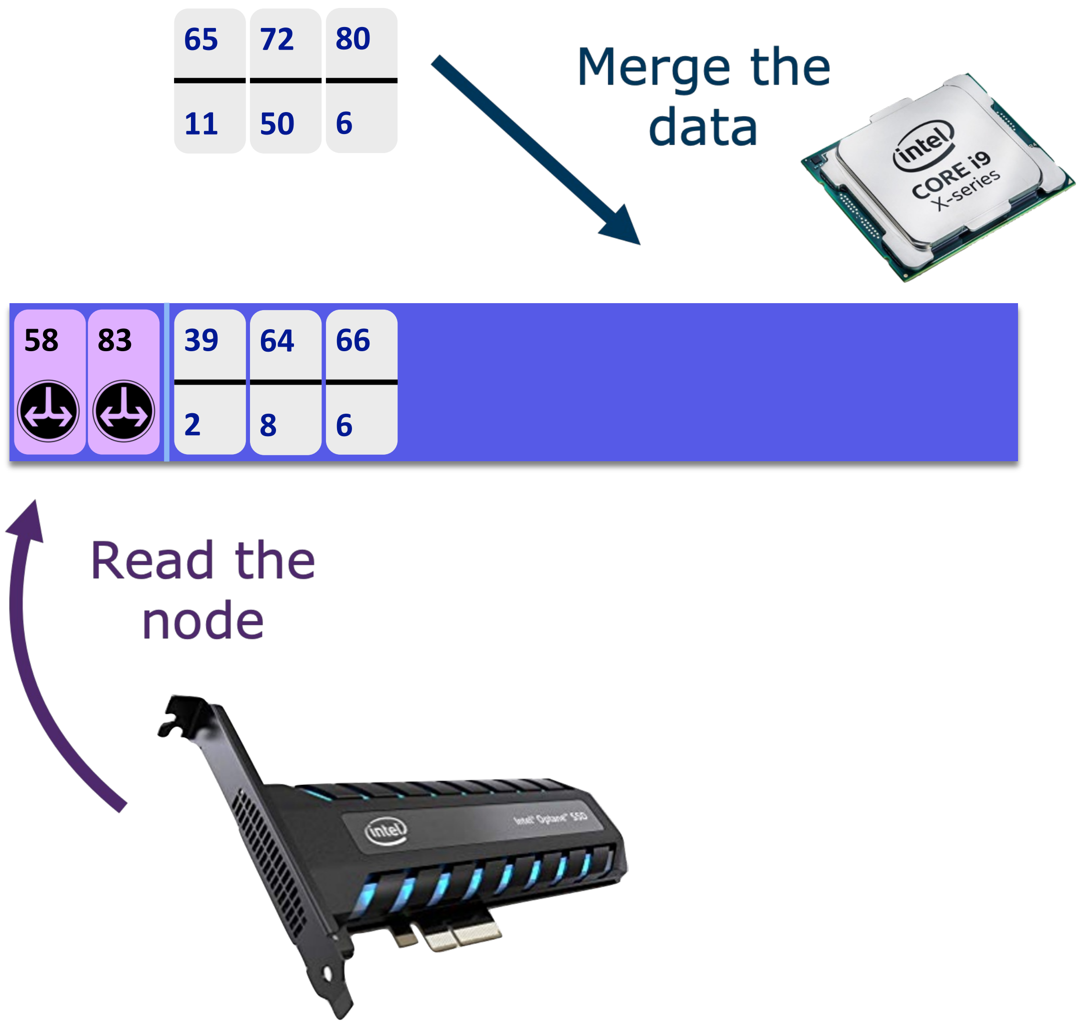
Recall: Insertions in B$^{\varepsilon}$-trees



Read the node

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

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



65	72	80
11	50	6

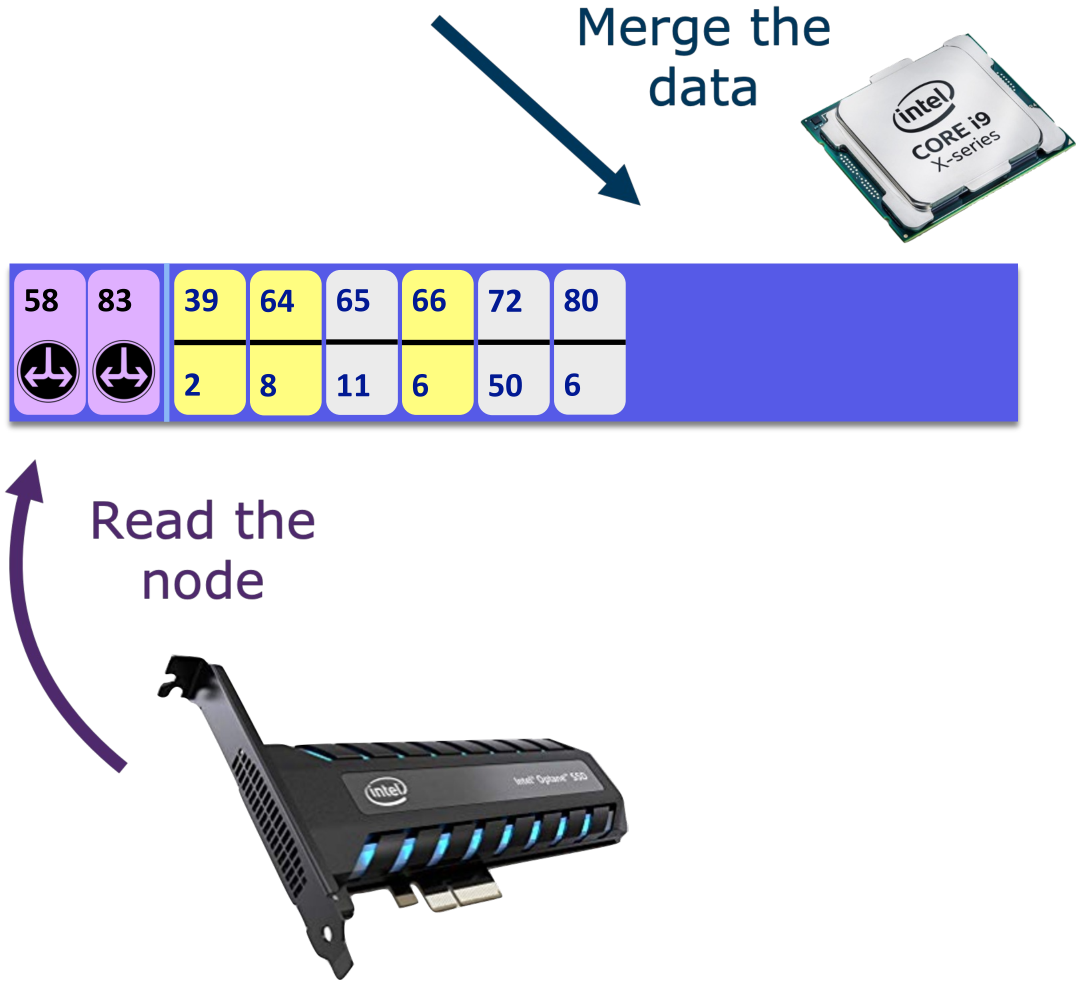
Merge the data

58	83	39	64	66
		2	8	6

Read the node

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

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



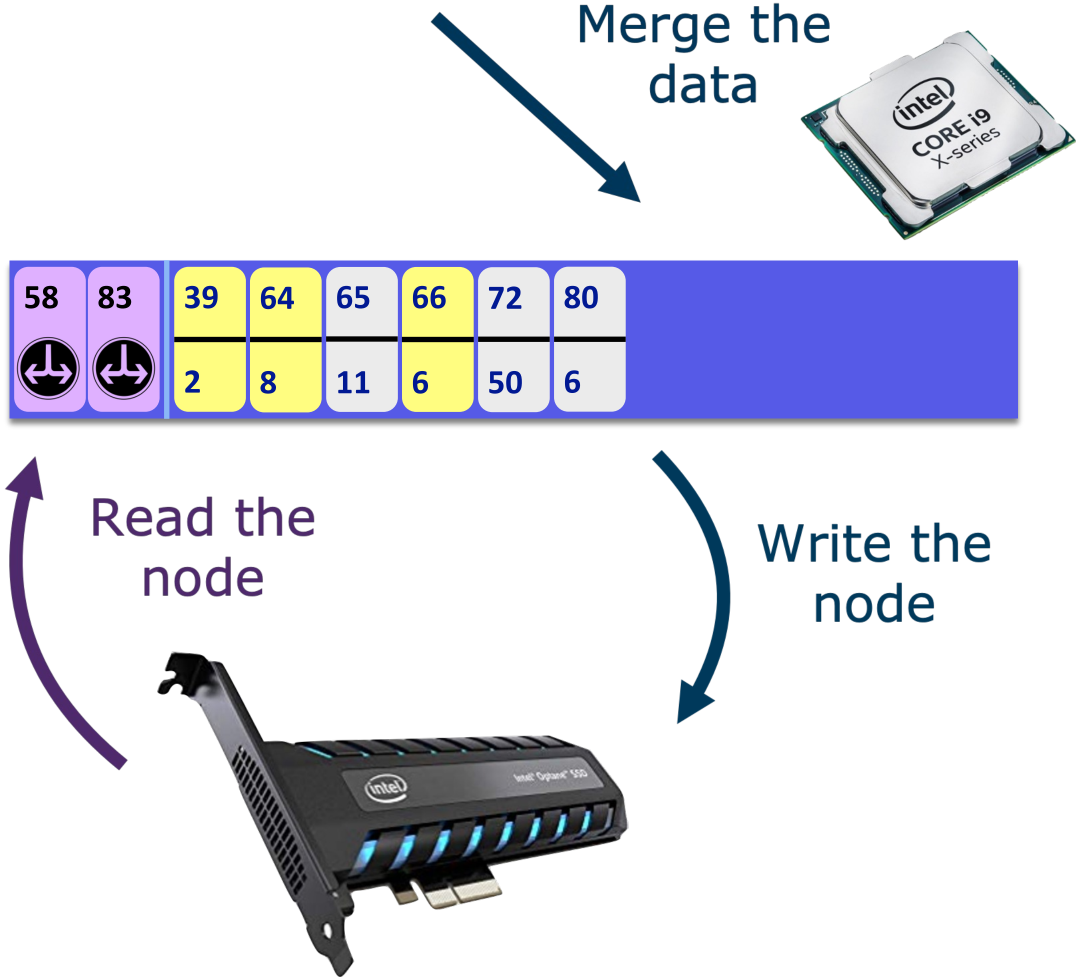
Merge the data

58	83	39	64	65	66	72	80
		2	8	11	6	50	6

Read the node

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

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



Merge the data

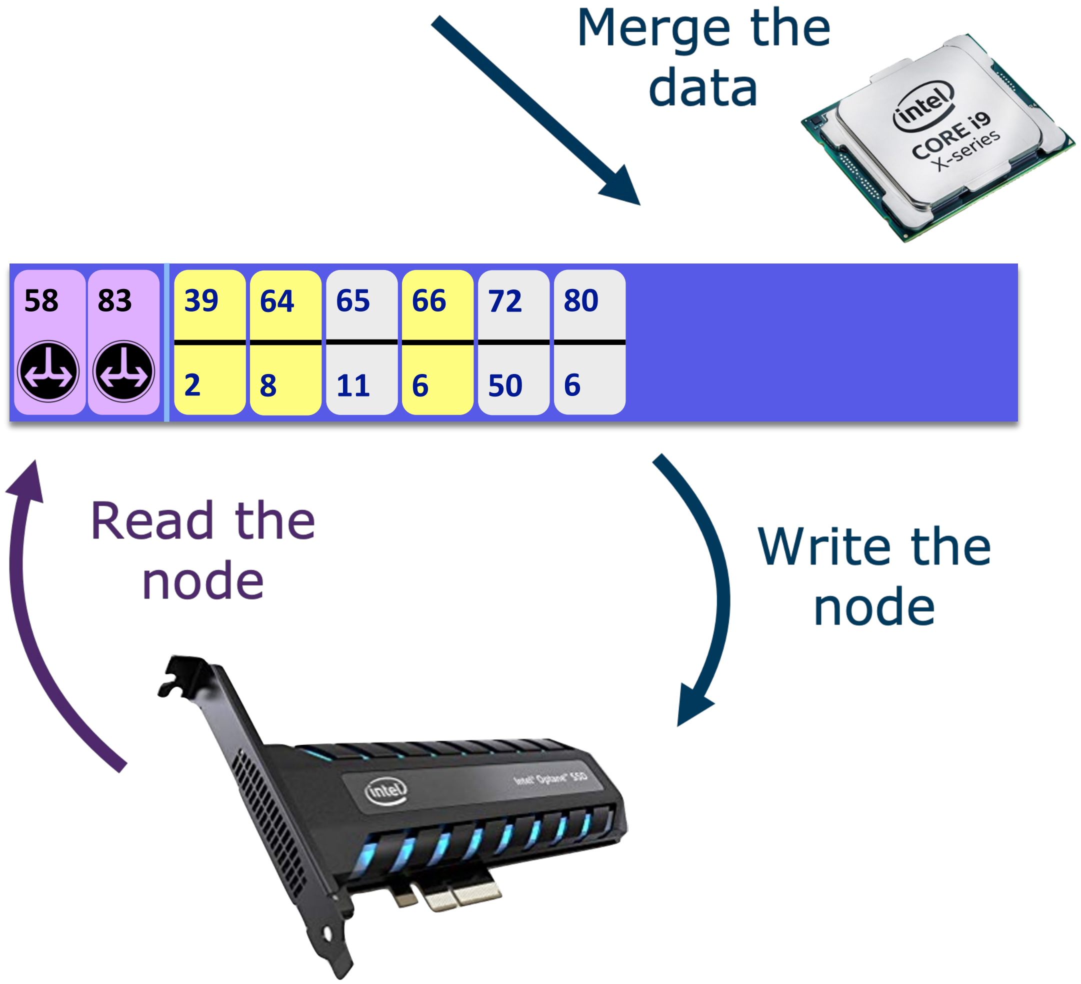
58	83	39	64	65	66	72	80
		2	8	11	6	50	6

Read the node

Write the node

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

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



Merge the data

58	83	39	64	65	66	72	80
		2	8	11	6	50	6

Read the node

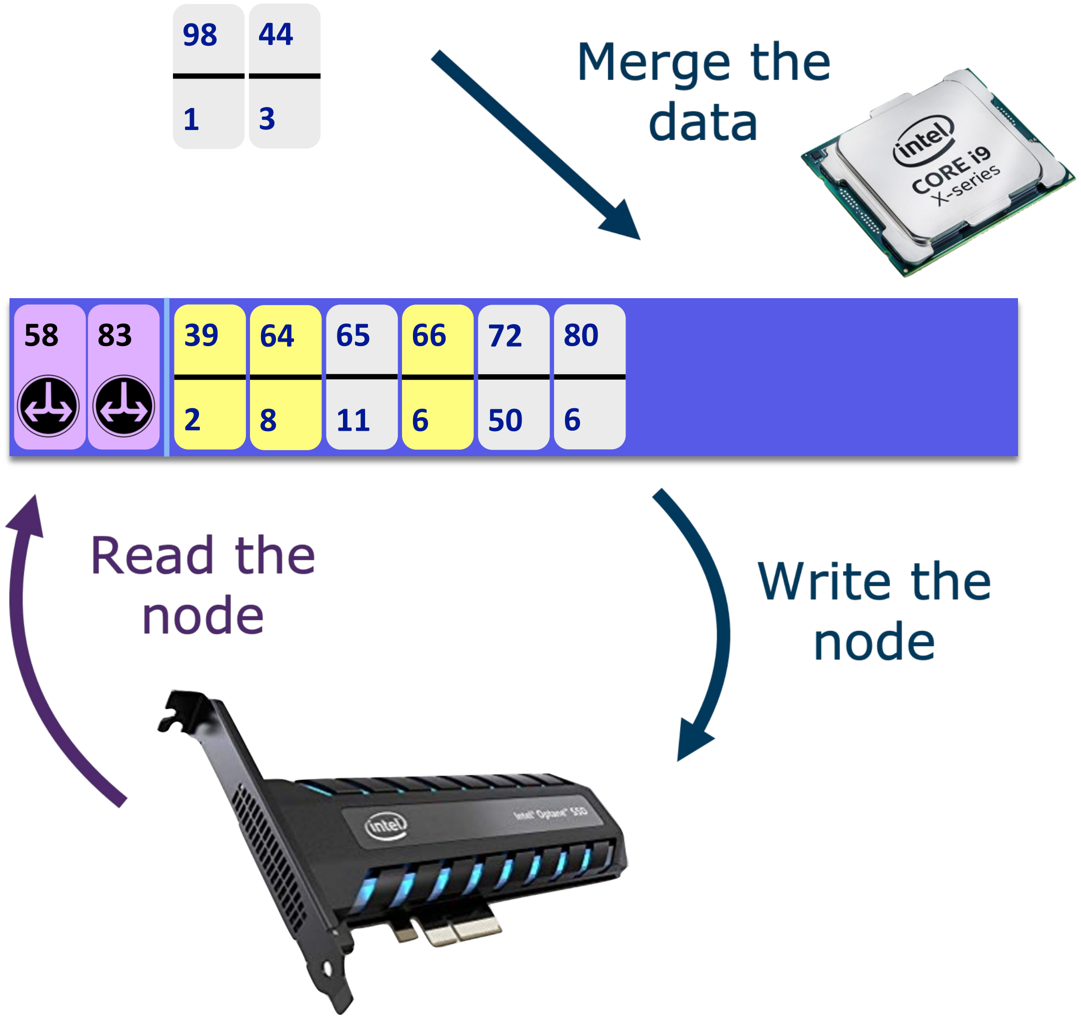
Write the node

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



98	44
1	3

58	83	39	64	65	66	72	80
		2	8	11	6	50	6

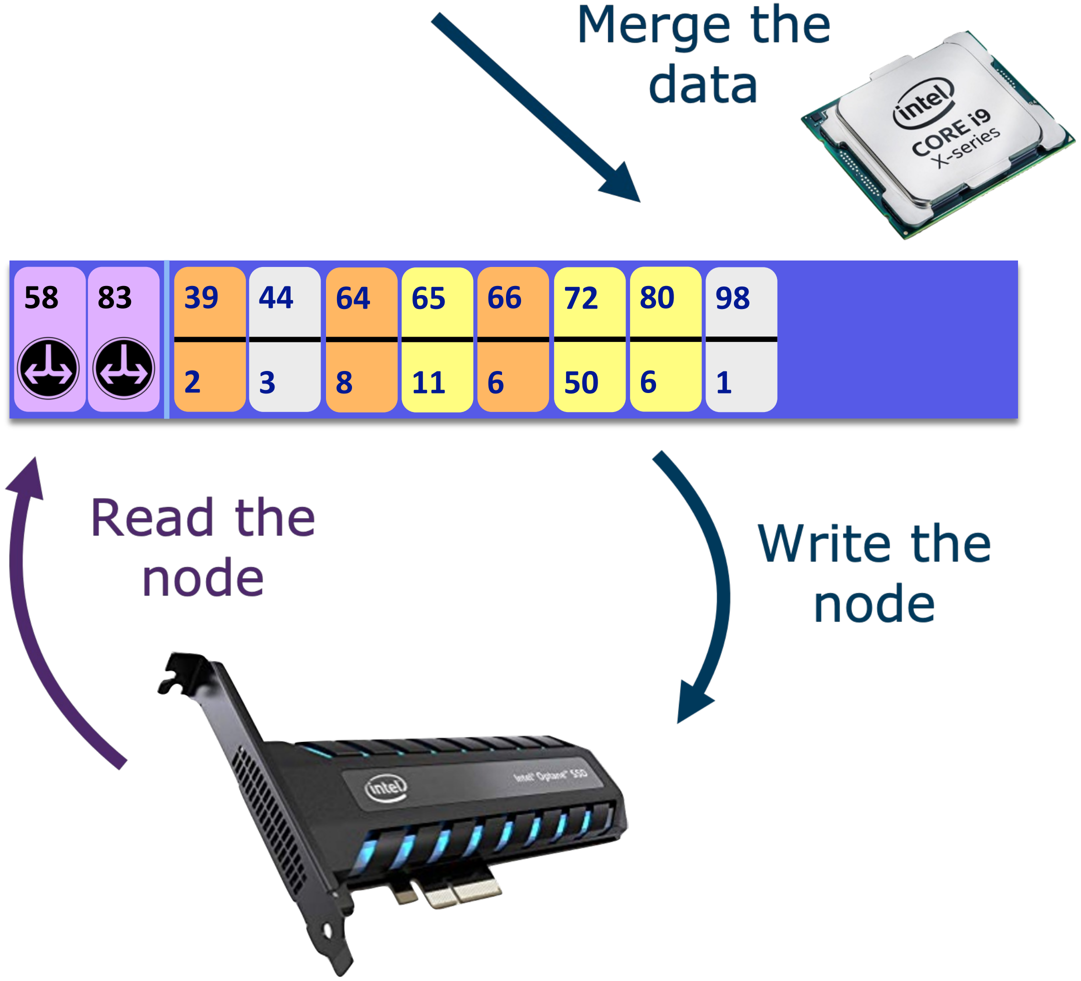
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

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

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



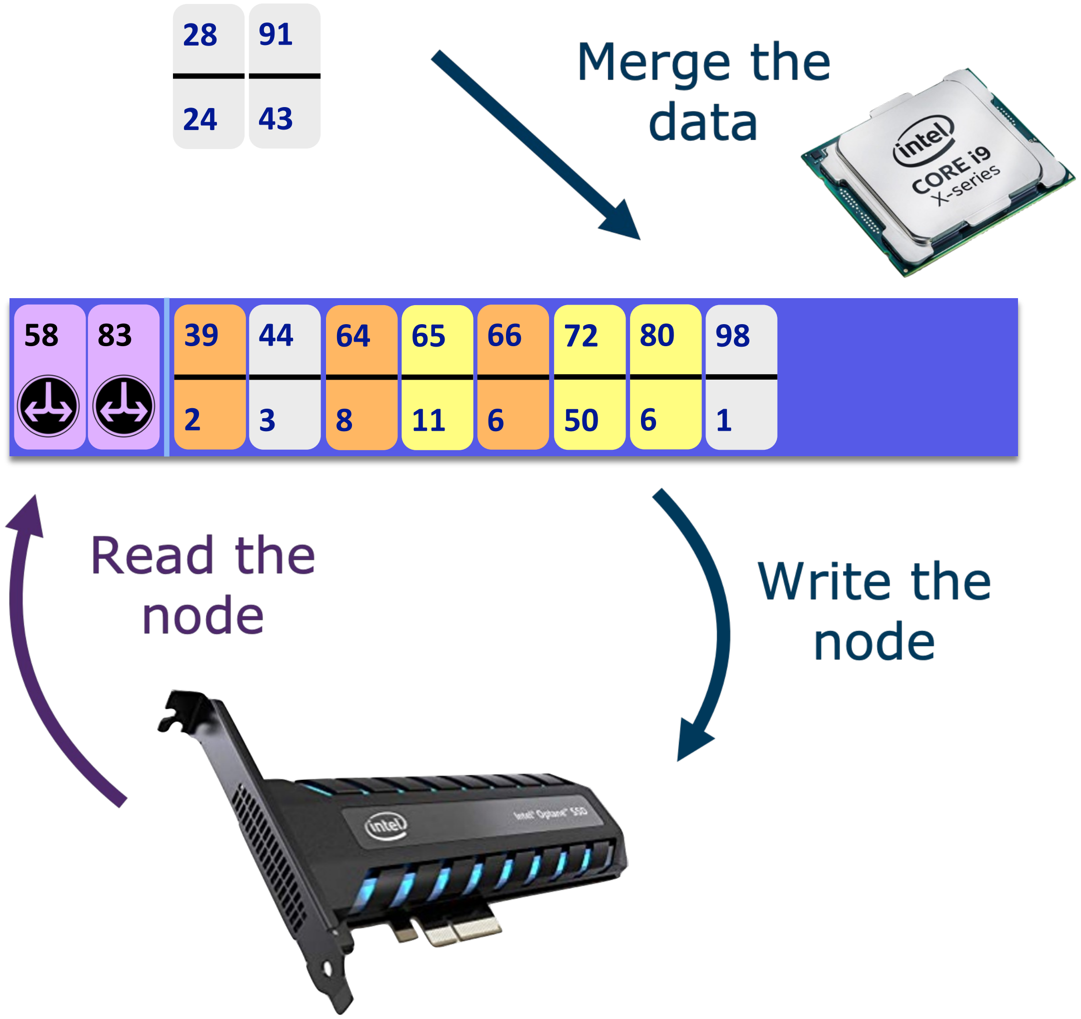
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

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

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



28	91
24	43

58	83	39	44	64	65	66	72	80	98
		2	3	8	11	6	50	6	1

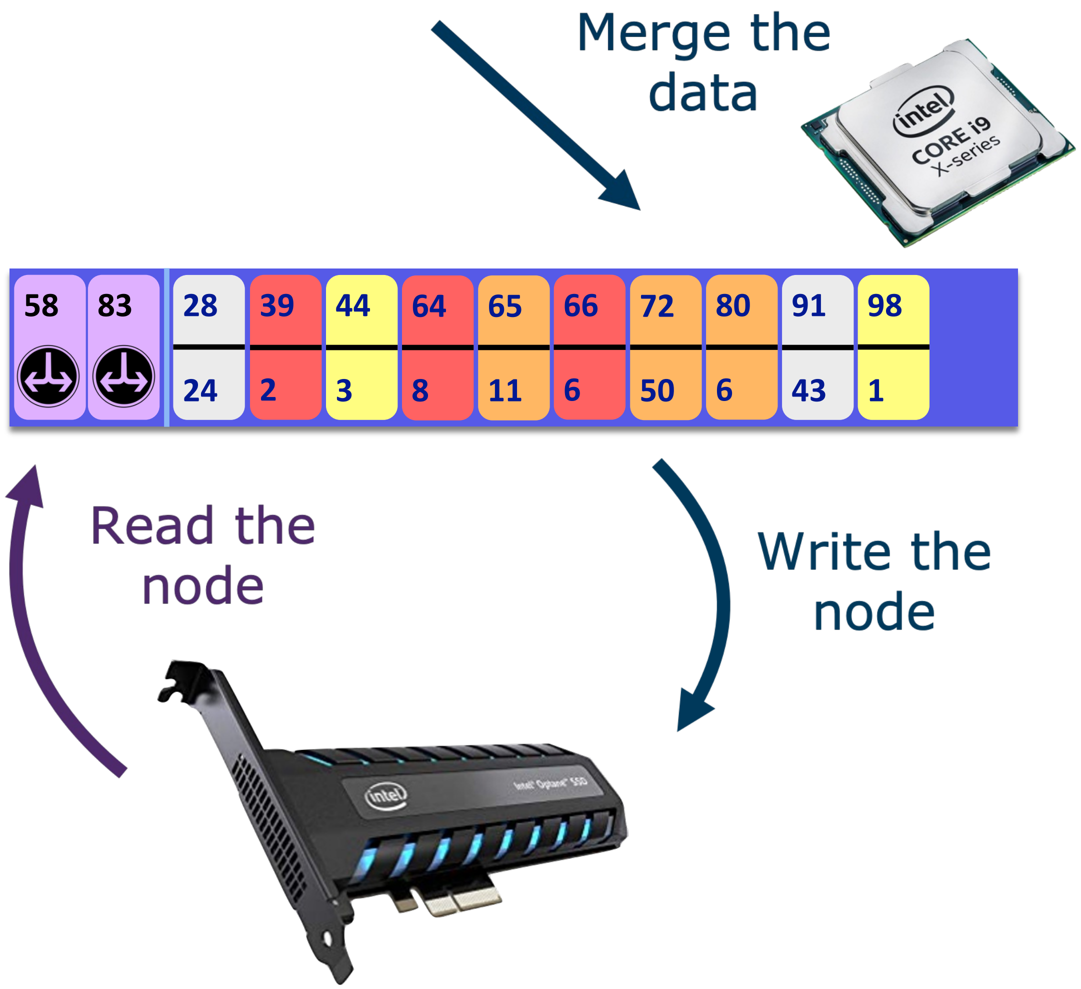
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

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

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



58	83	28	39	44	64	65	66	72	80	91	98
		24	2	3	8	11	6	50	6	43	1

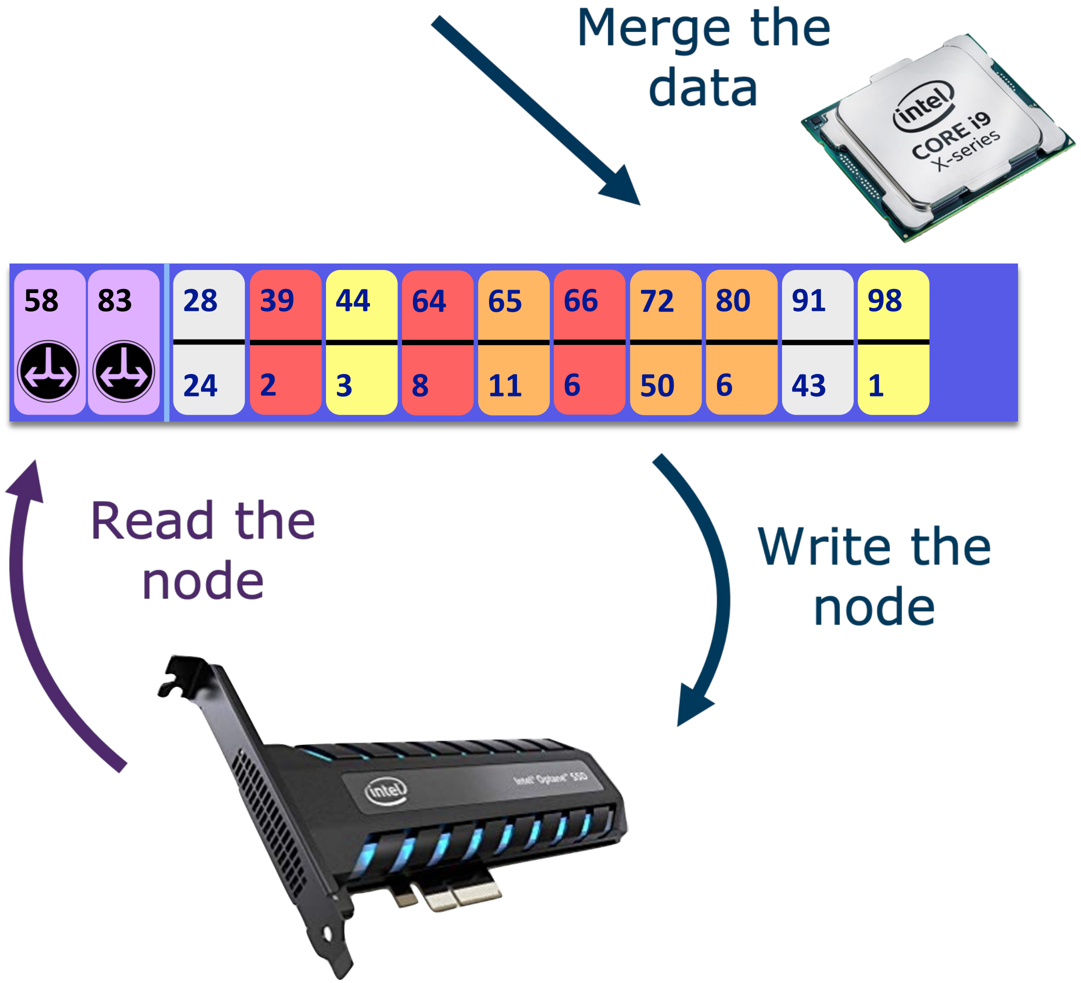
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

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

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



Merge the data

58	83	28	39	44	64	65	66	72	80	91	98
		24	2	3	8	11	6	50	6	43	1

Read the node

Write the node

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!

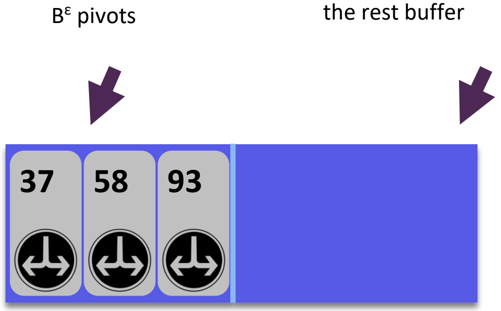
Size-Tiered B^{ε}-Trees

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

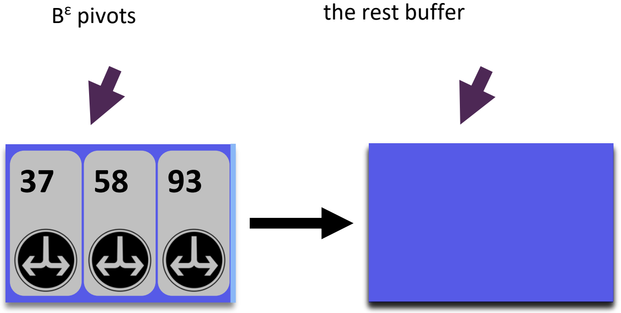
Recall:
a B^{ε}-tree node has pivots and a buffer



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

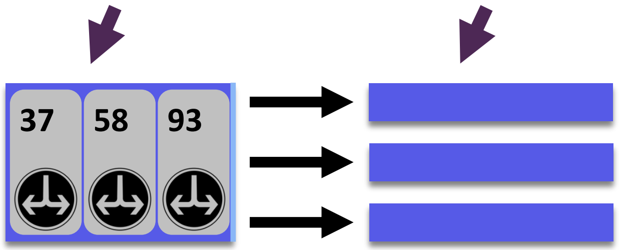


In an STB$^{\varepsilon}$-tree, the buffer is stored separately

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



B^ε pivots

the rest buffer

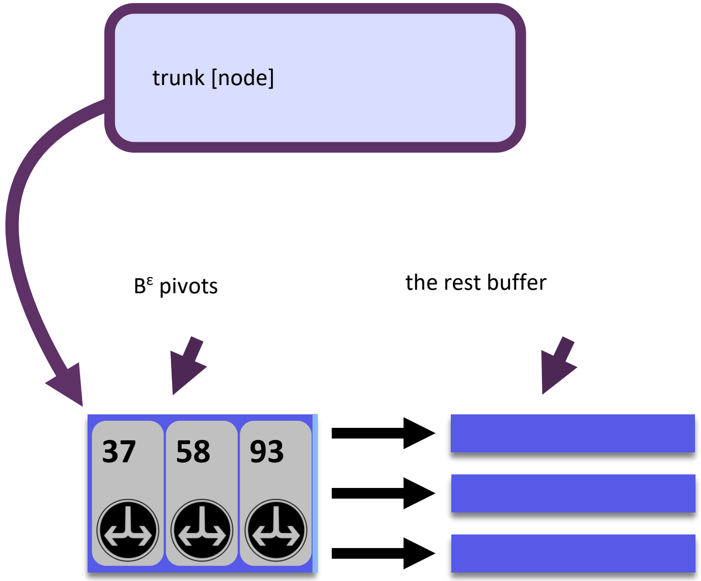
and in several discontiguous pieces

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

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

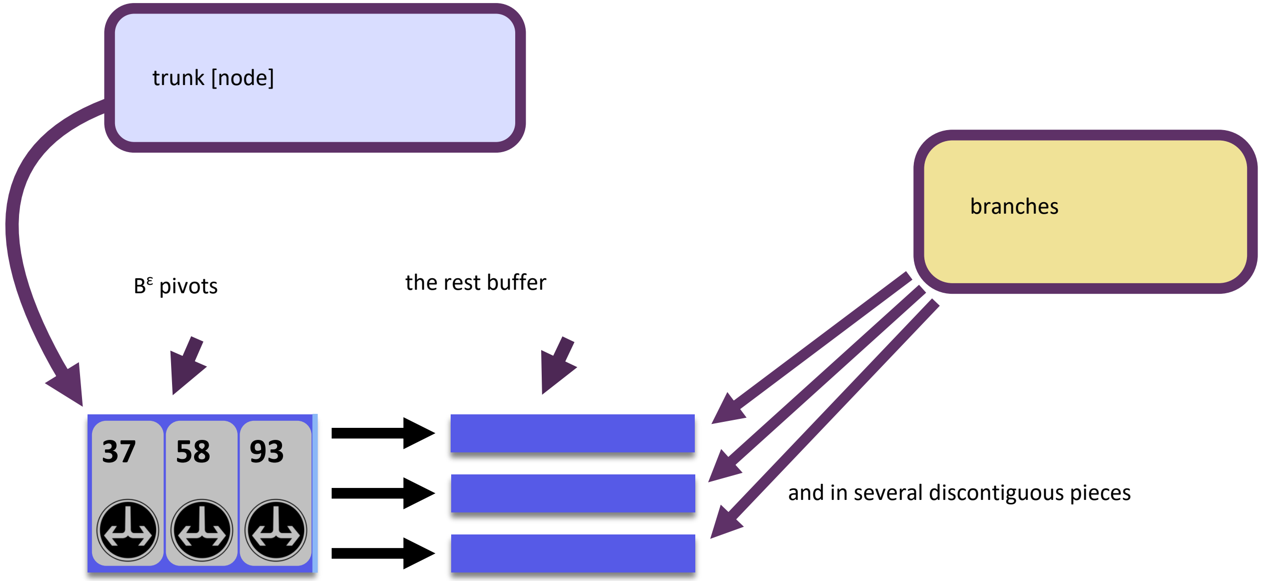


and in several discontiguous pieces

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

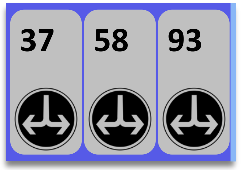
In an STB$^{\varepsilon}$-tree, the buffer is stored separately

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

Size-Tiered B^{ε}-Trees

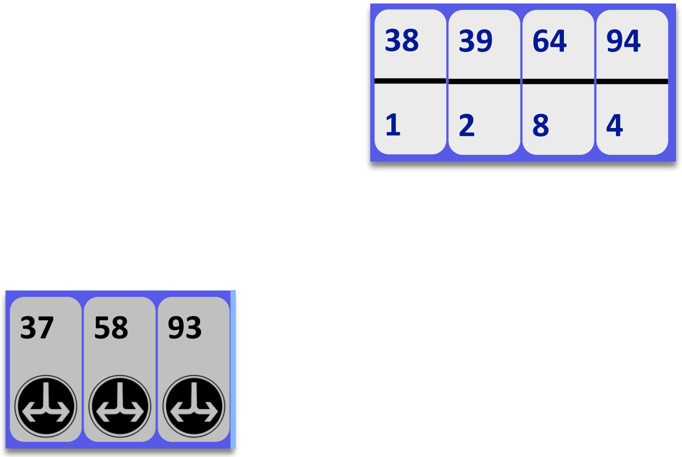
A Size-Tiered B^{ε}-tree is a B^{ε}-tree where the buffer is stored discontiguously

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



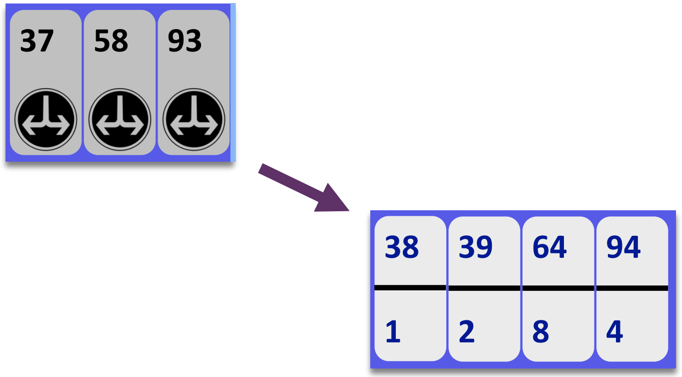
When new data is flushed into the trunk node…

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

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

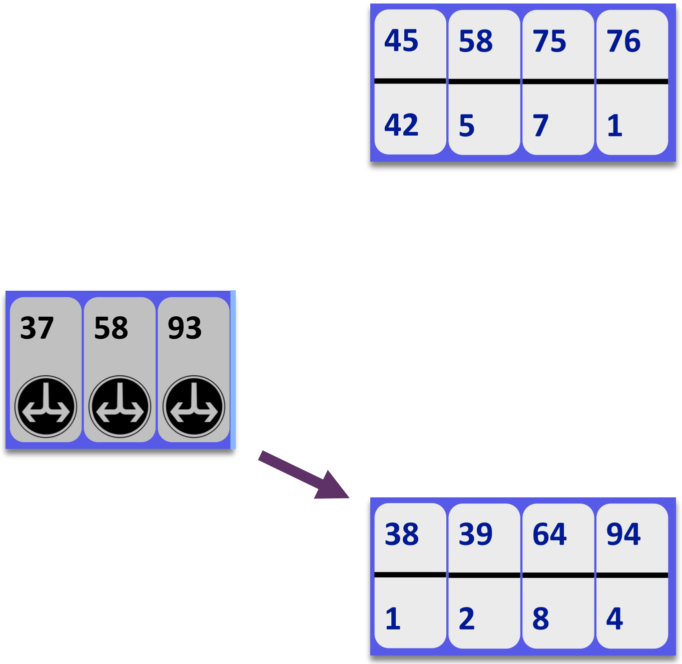
When new data is flushed into the trunk node…

…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



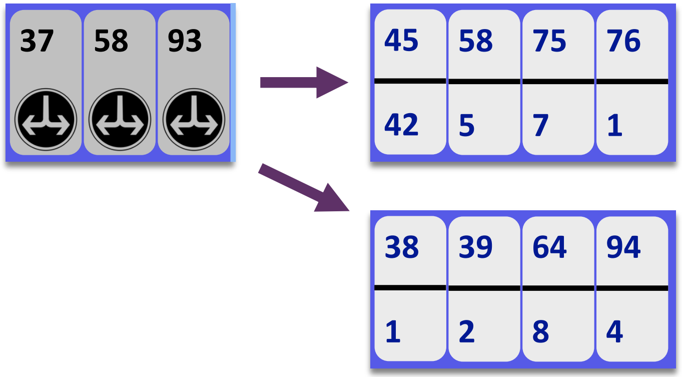
When new data is flushed into the trunk node…

…it is added as a new branch

Size-Tiered B$^\epsilon$-Trees

A Size-Tiered B$^\epsilon$-tree is a B$^\epsilon$-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

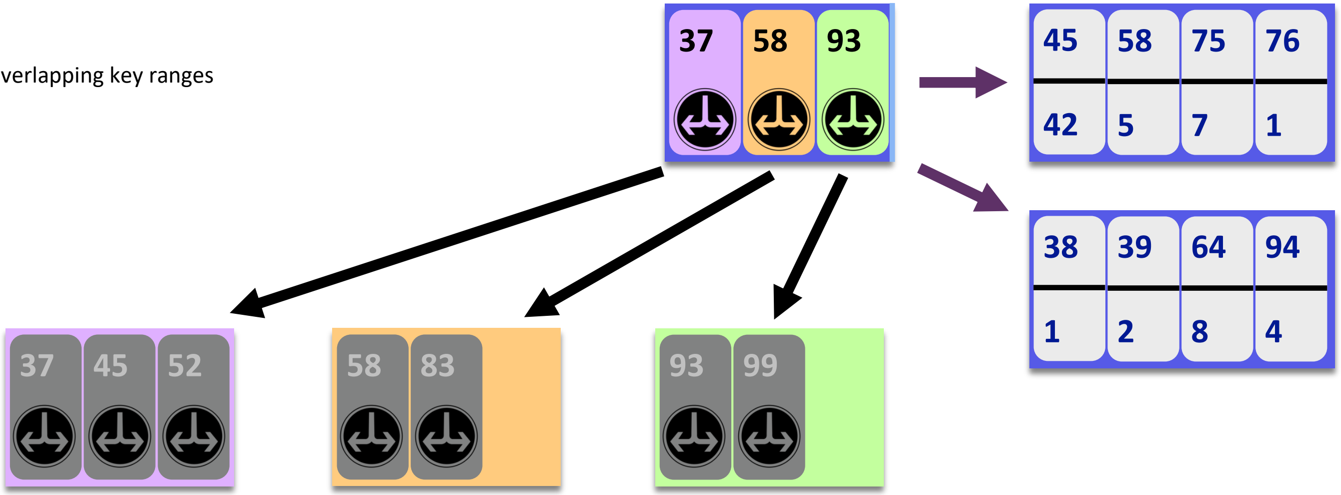
A Size-Tiered B$^\varepsilon$-tree is a B$^\varepsilon$-tree where the buffer is stored discontiguously

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

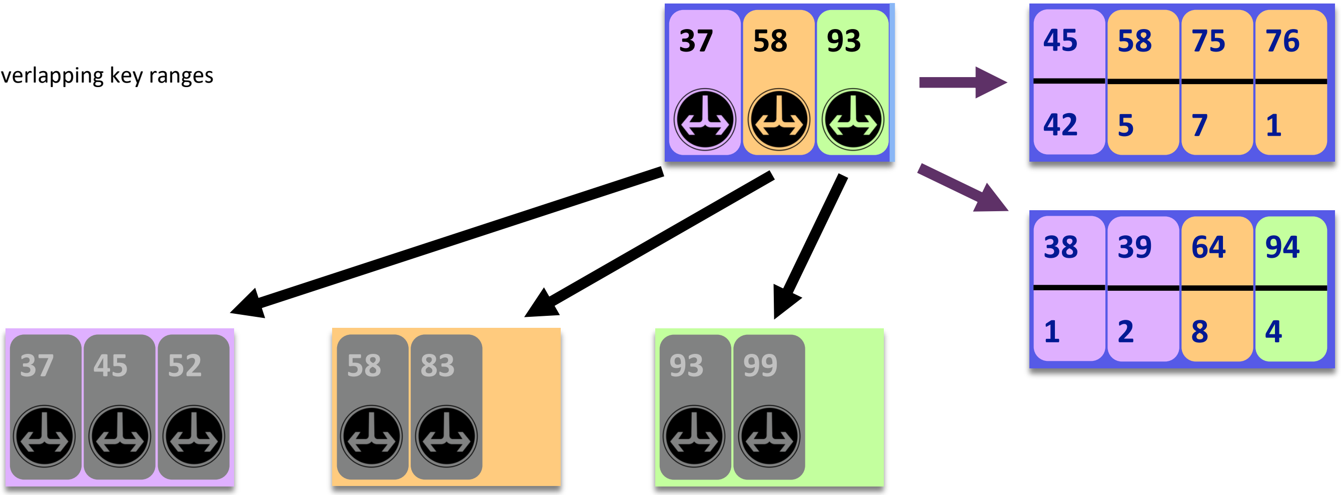
A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node…

…it is added as a new branch

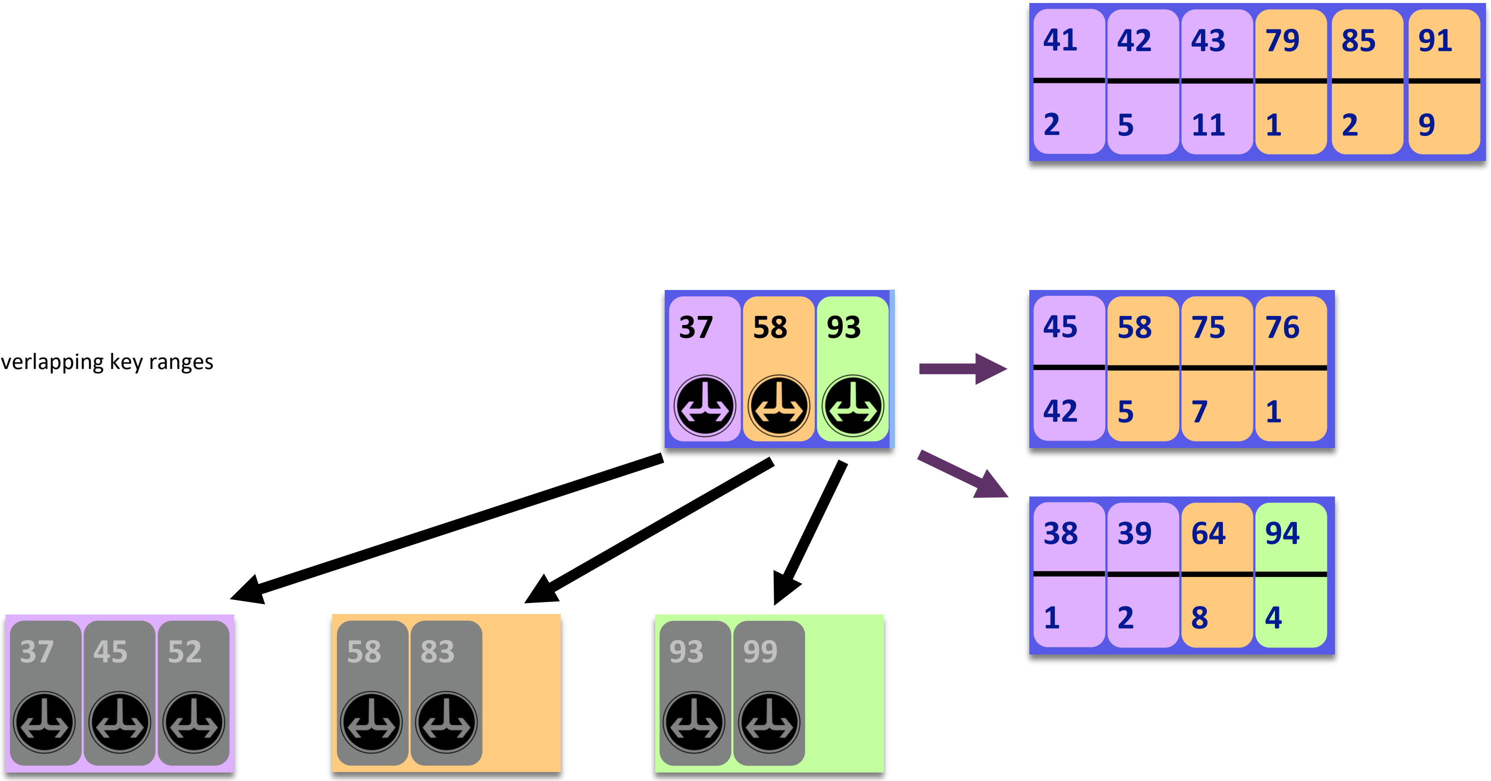
Branches may have overlapping key ranges

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…

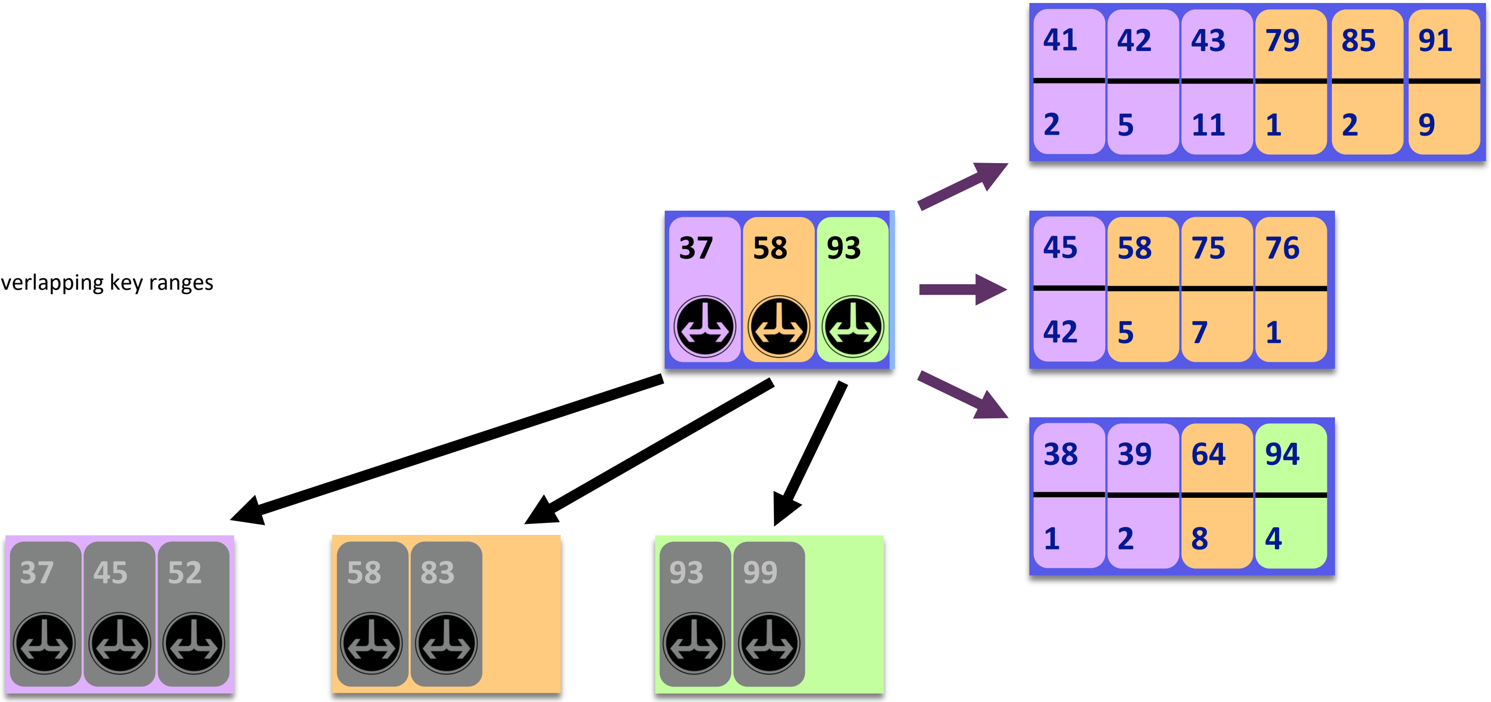
…it is added as a new branch

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

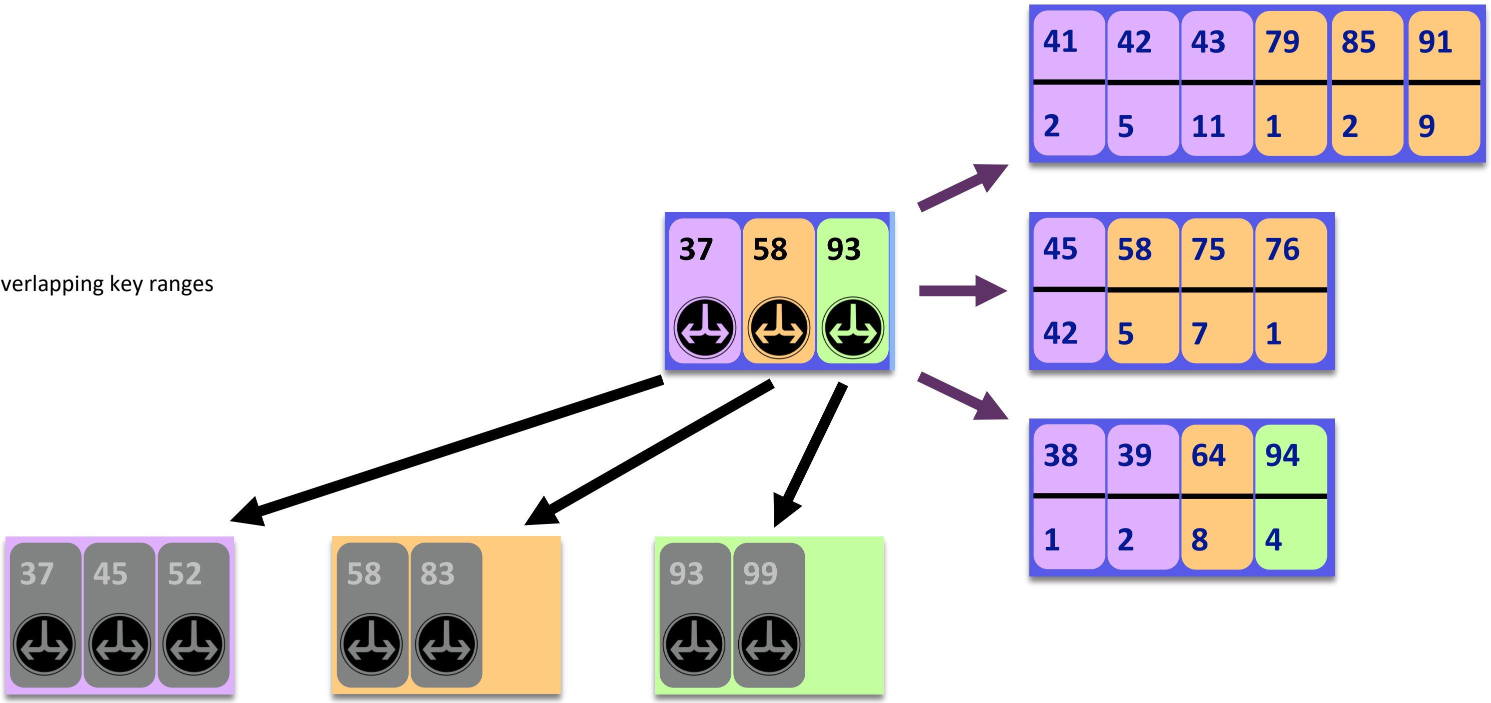
The old branches do not need to be rewritten

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



When new data is flushed into the trunk node…

…it is added as a new branch

Branches may have overlapping key ranges

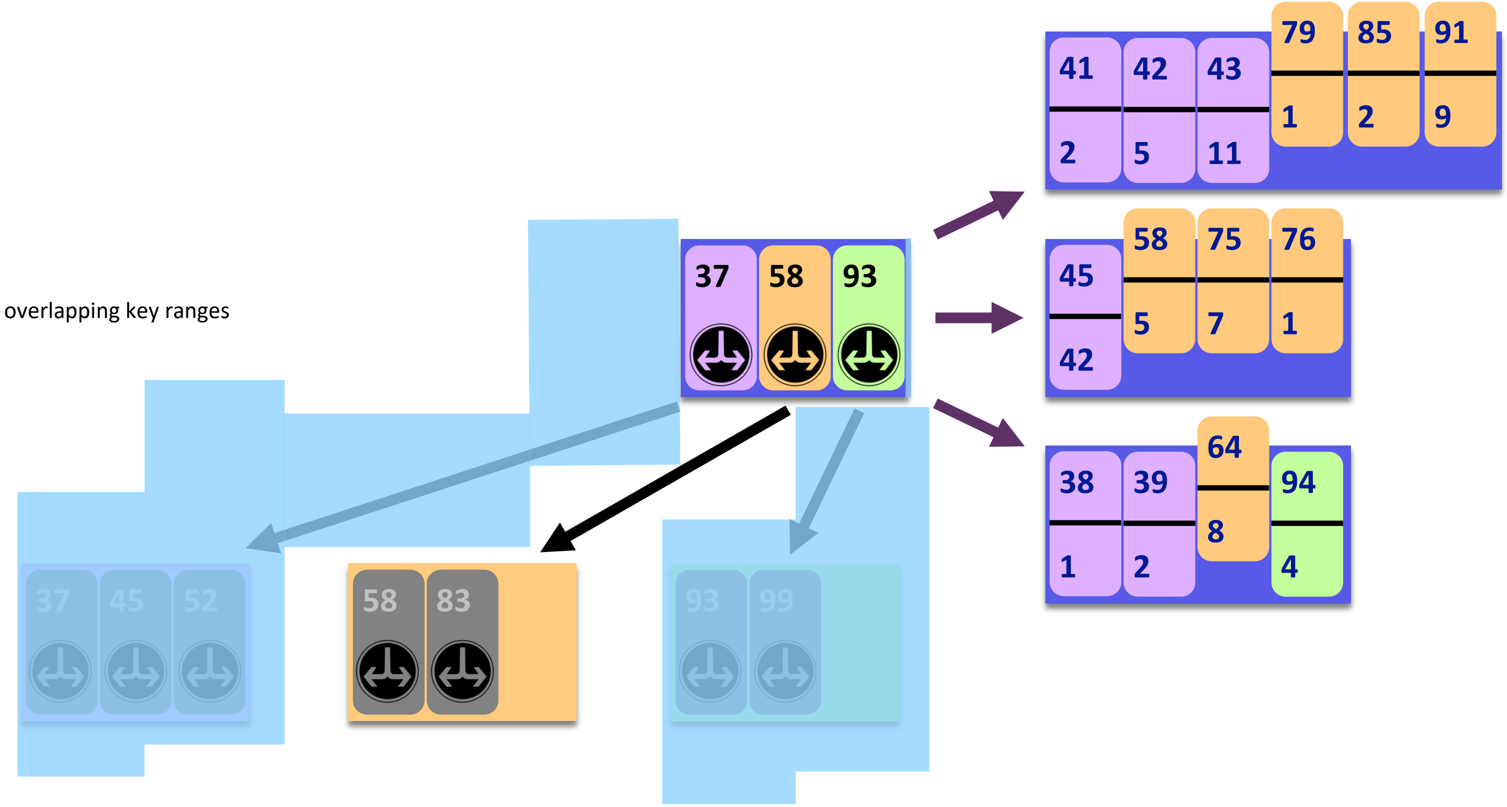
The old branches do not need to be rewritten

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



When new data is flushed into the trunk node…

…it is added as a new branch

Branches may have overlapping key ranges

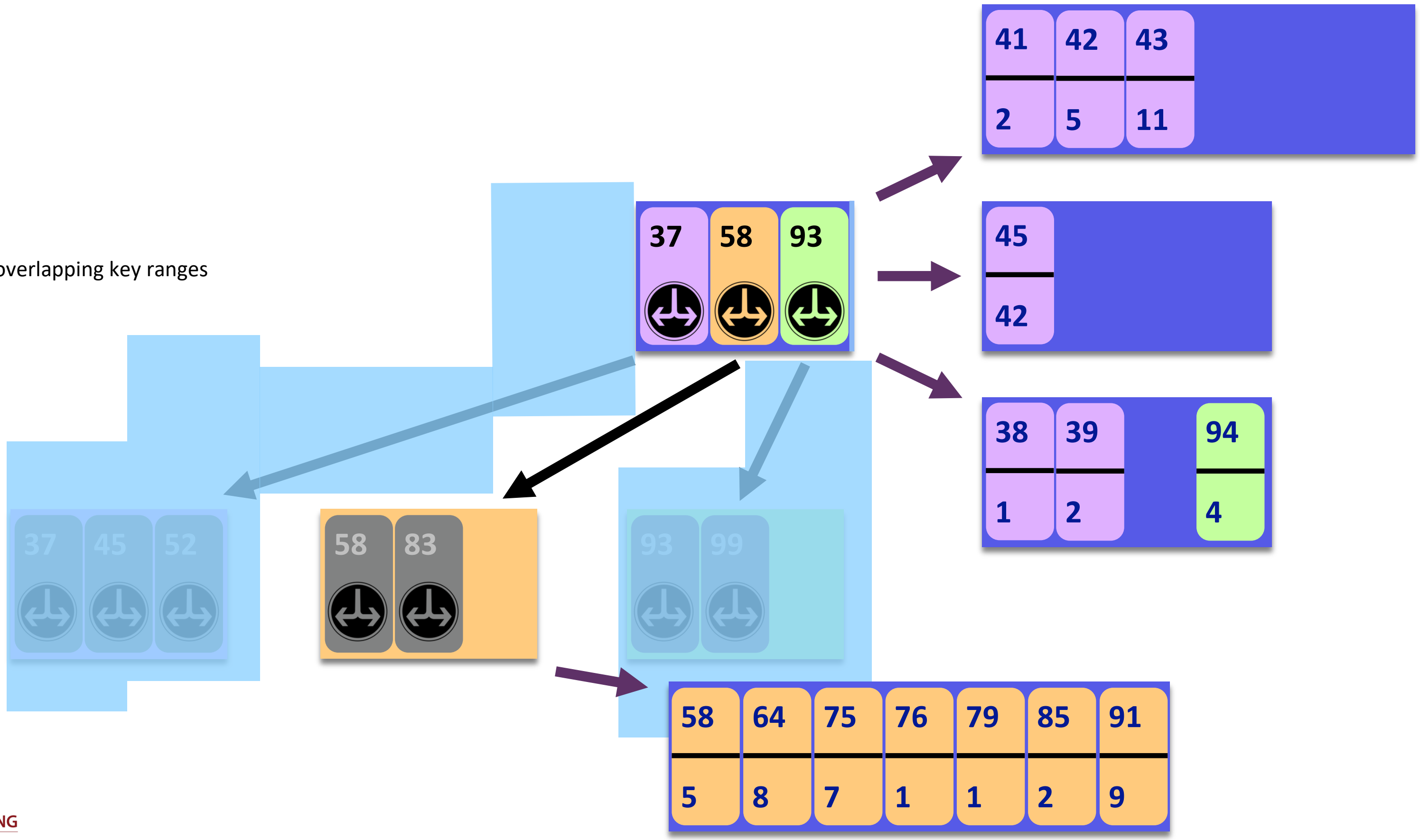
The old branches do not need to be rewritten

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



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

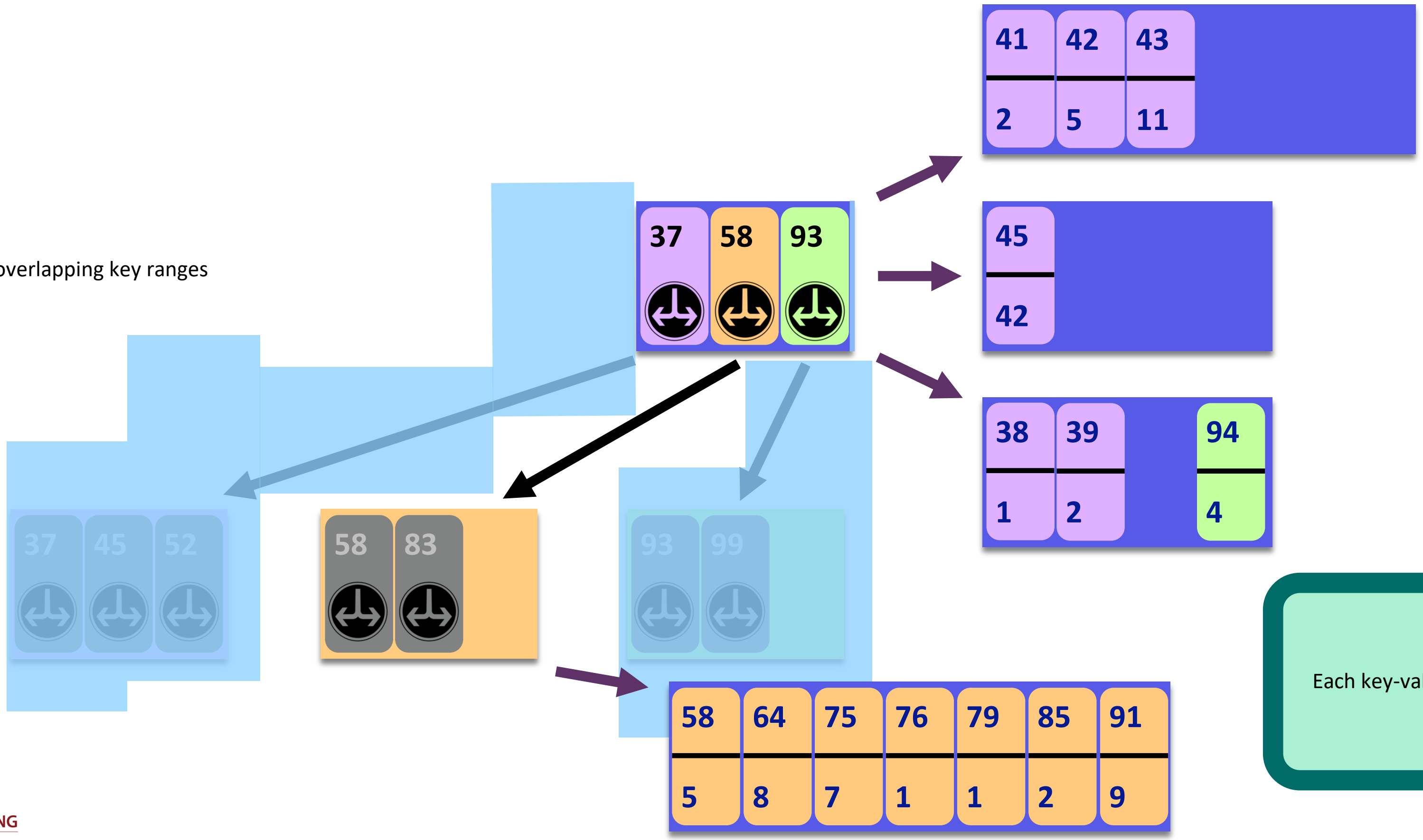
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

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

Branches may have overlapping key ranges

The old branches do not need to be rewritten

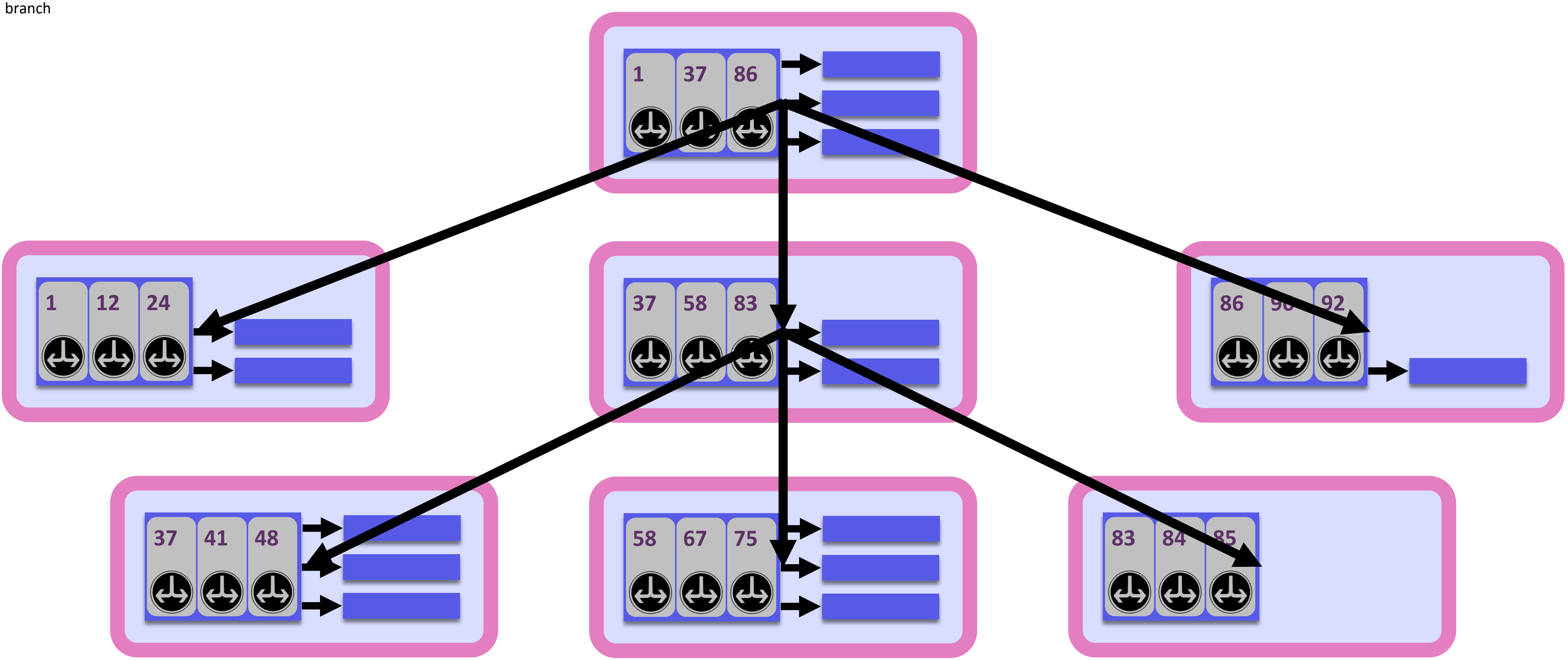
Each key-value pair is read/written once per trunk node

Lookups in 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

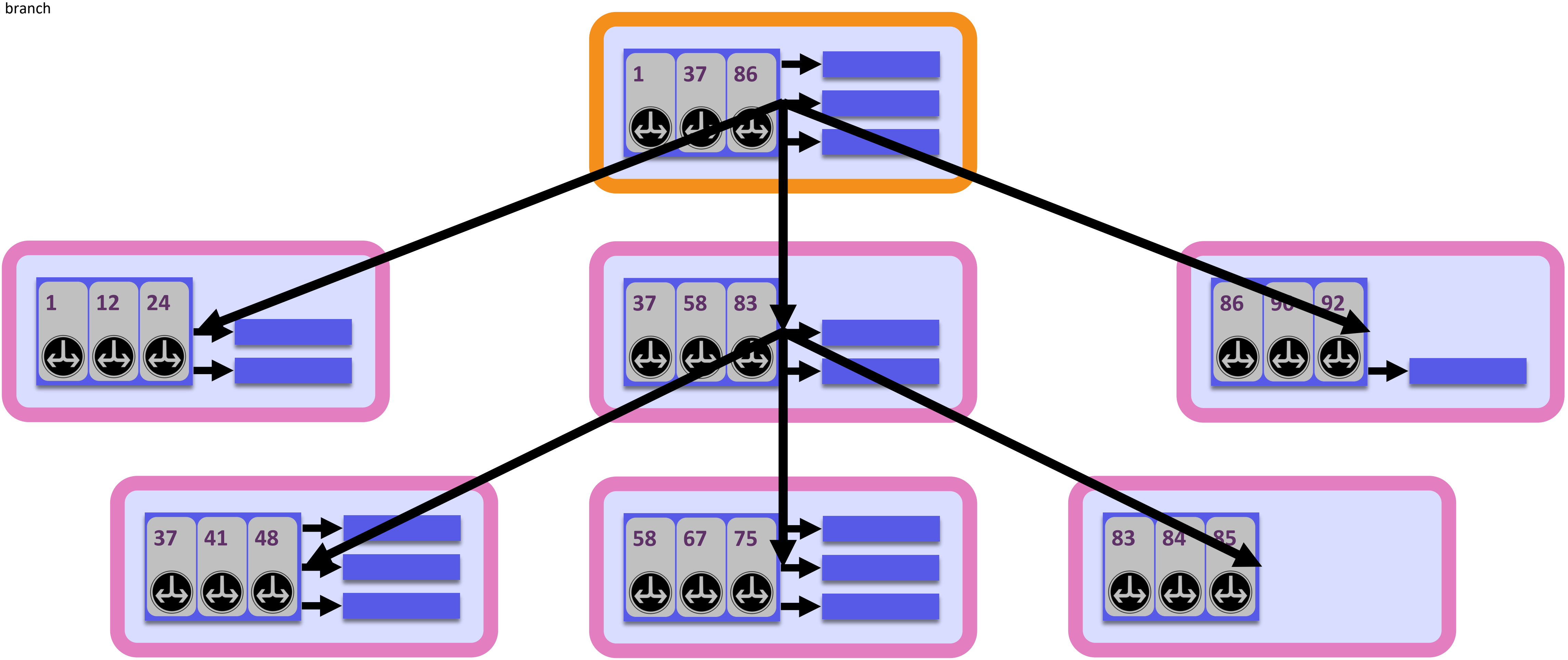
Query(71)



Size-Tiered B^{ε}-Trees

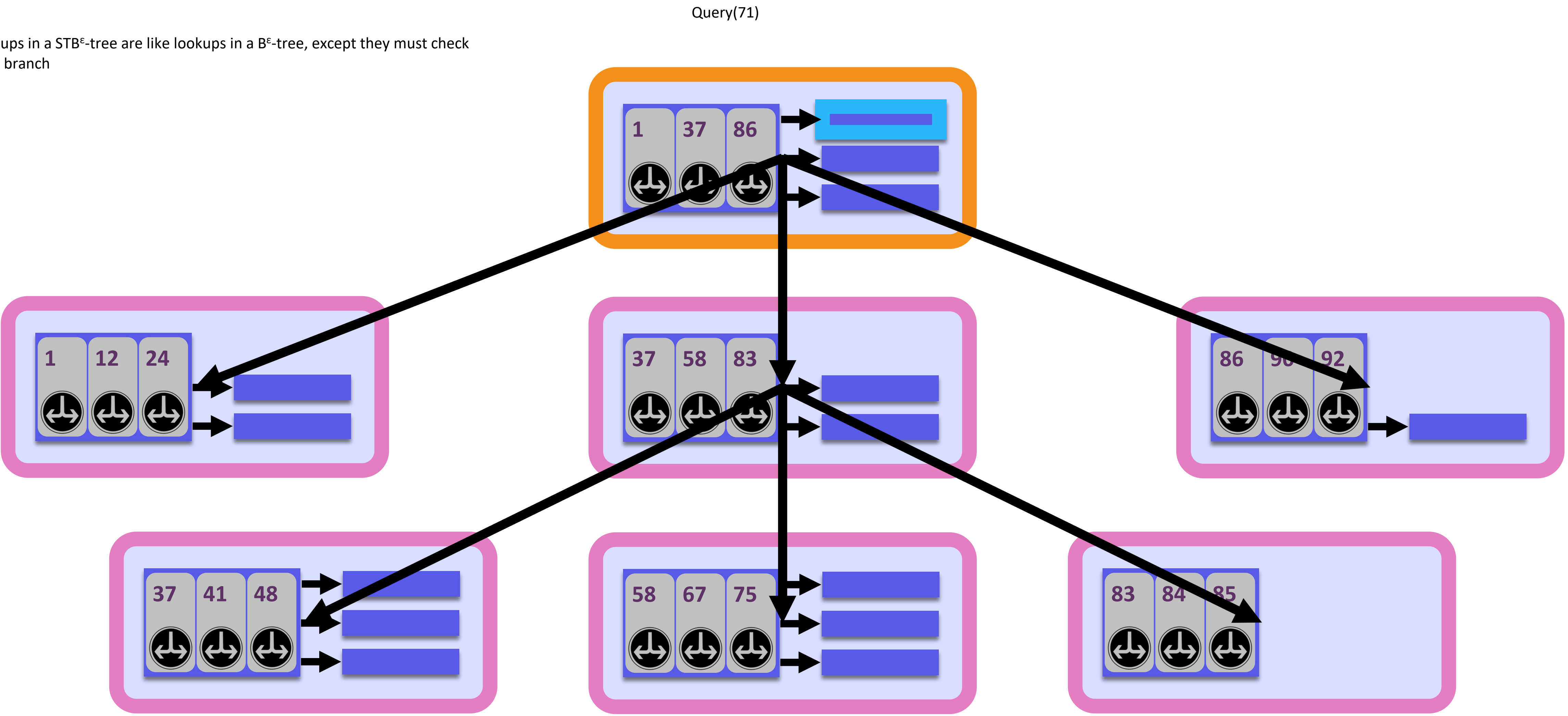
Lookups in a STB^{ε}-tree are like lookups in a B^{ε}-tree, except they must check each branch

Query(71)



Size-Tiered B$^\varepsilon$-Trees

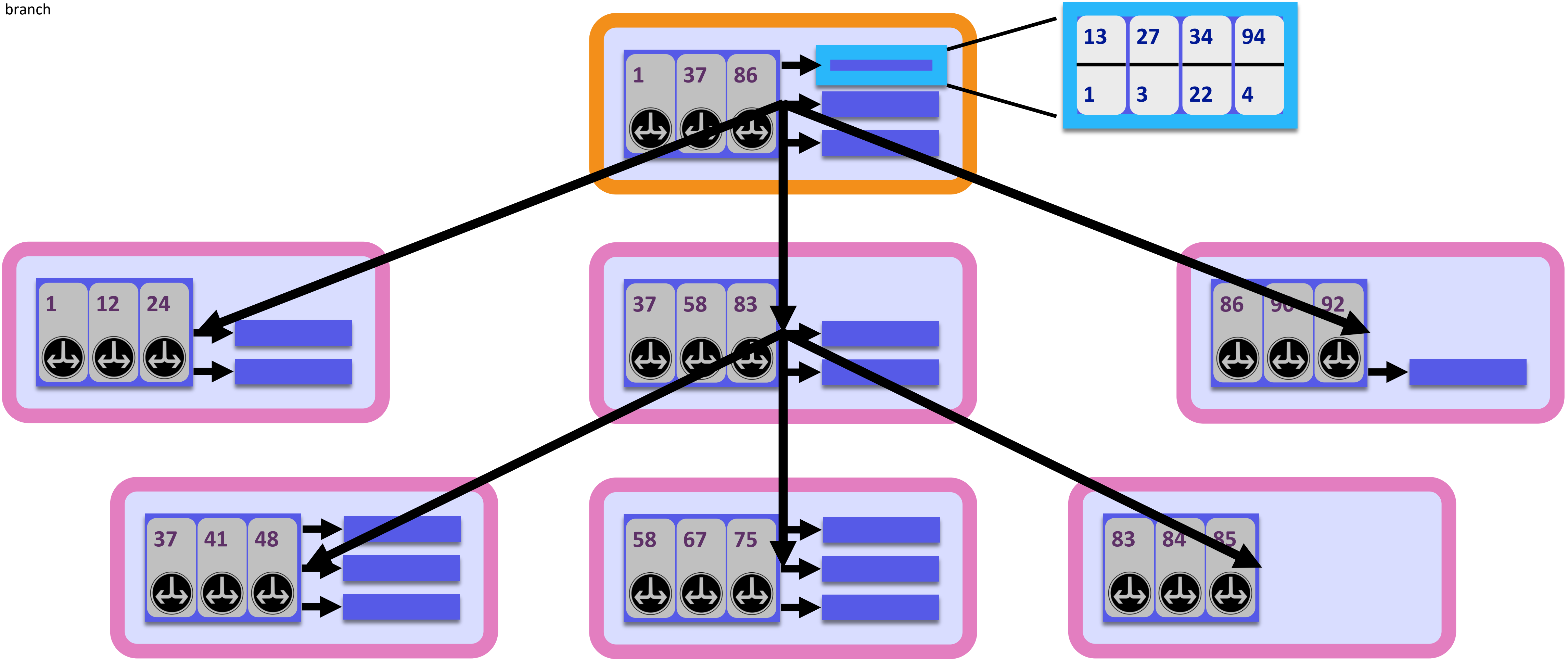
Lookups in a STB$^\varepsilon$-tree are like lookups in a B$^\varepsilon$-tree, except they must check each branch



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

Query(71)

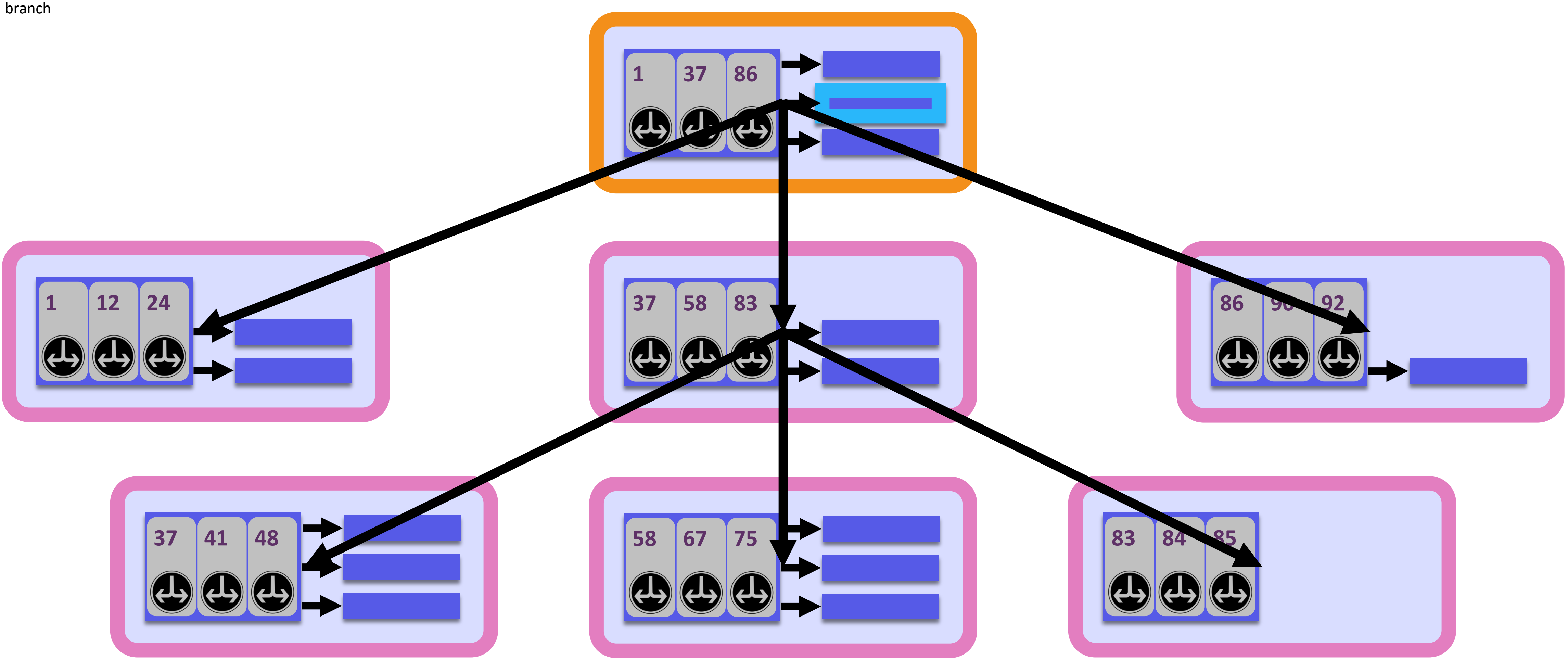
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

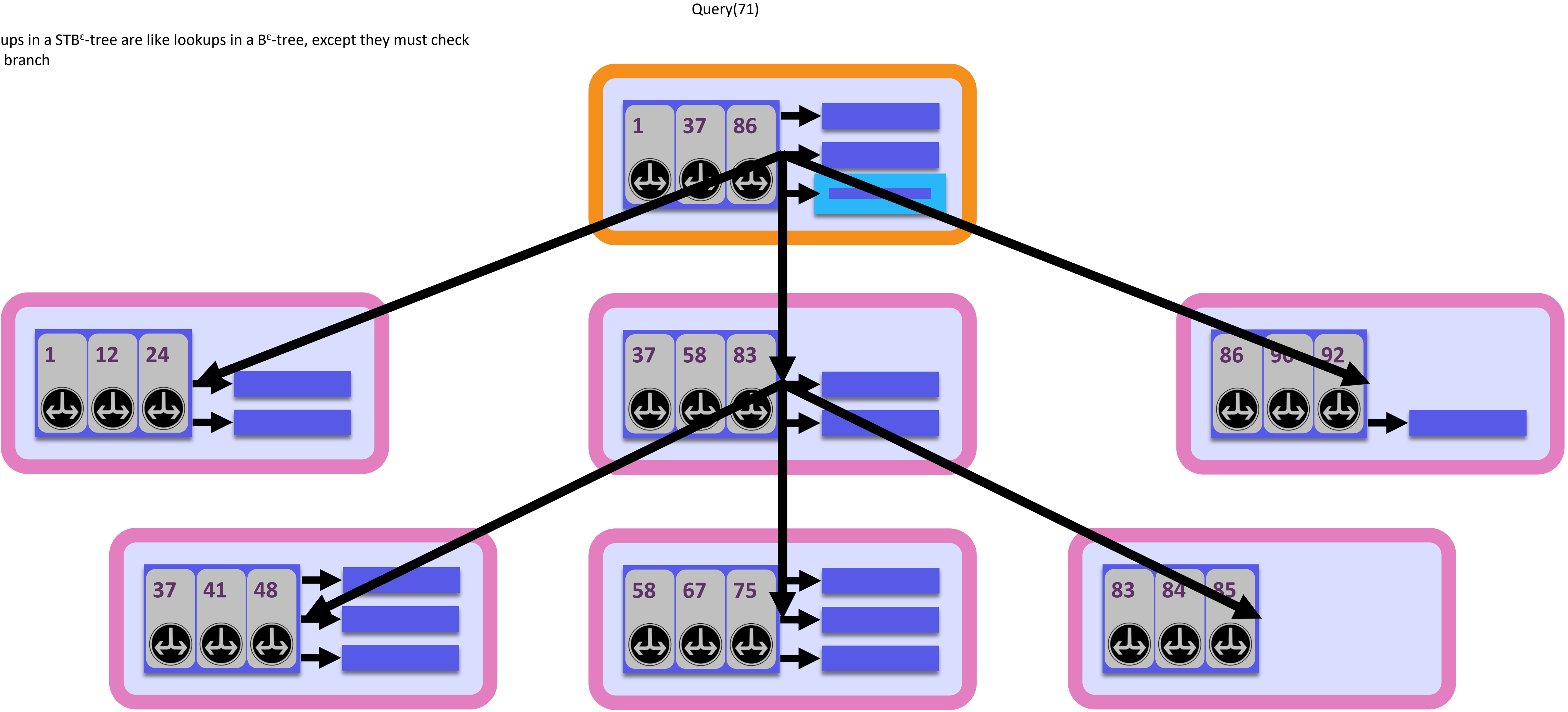
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

Query(71)



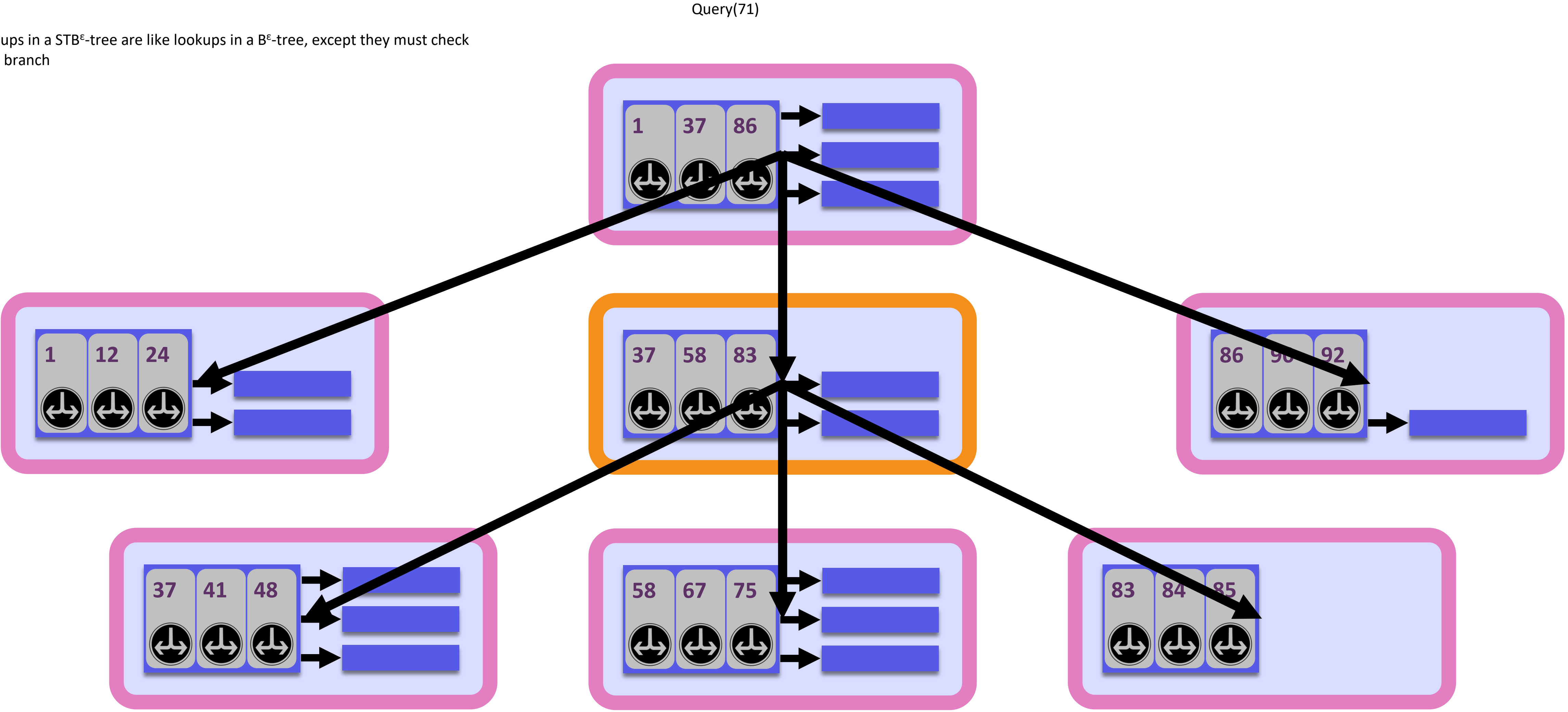
Size-Tiered Bᵋ-Trees

Lookups in a STBᵋ-tree are like lookups in a Bᵋ-tree, except they must check each branch



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

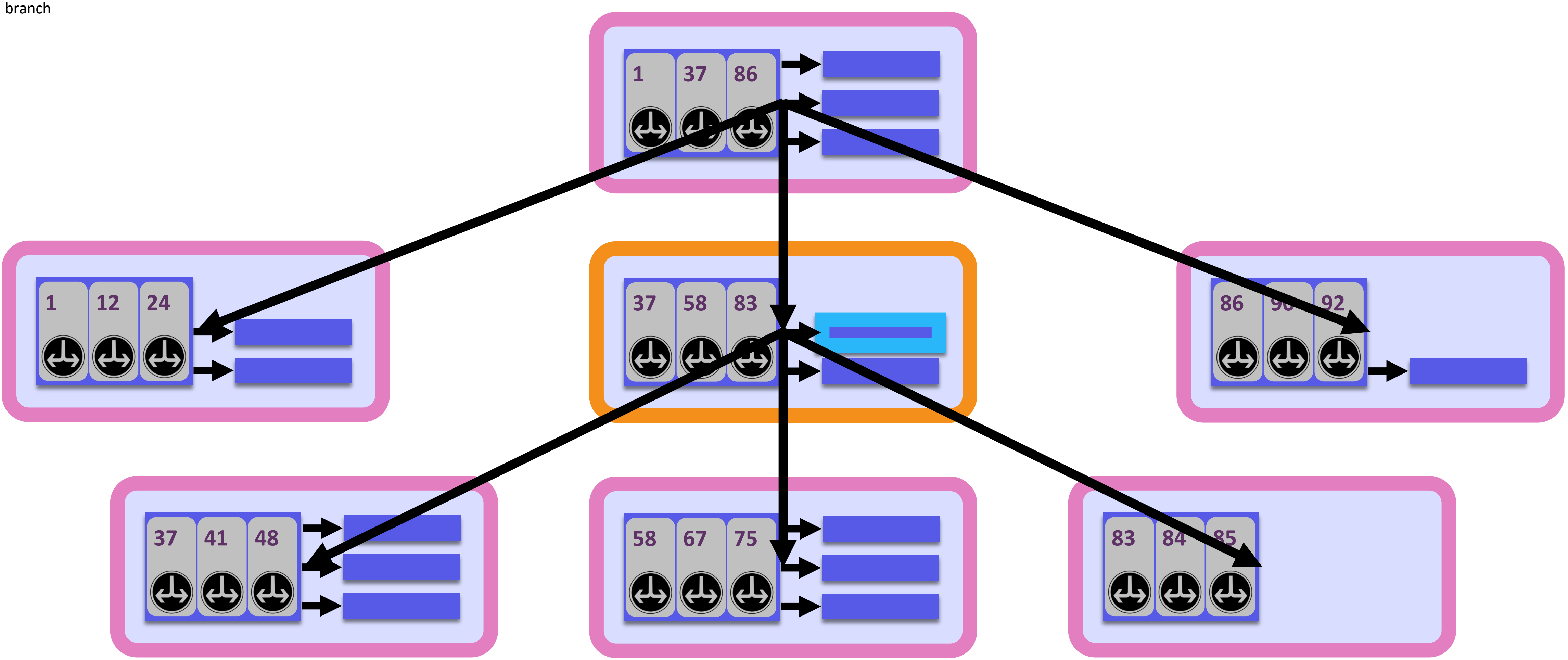
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

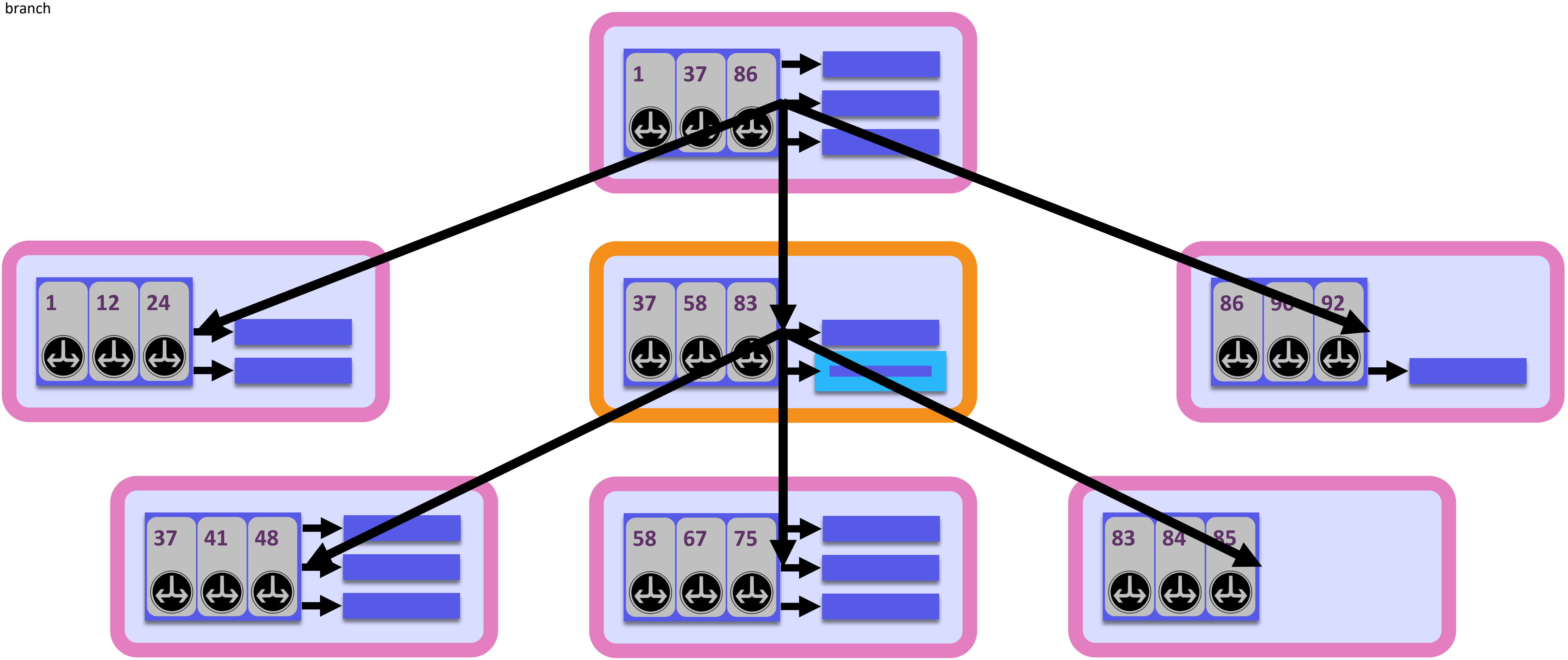
Query(71)



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

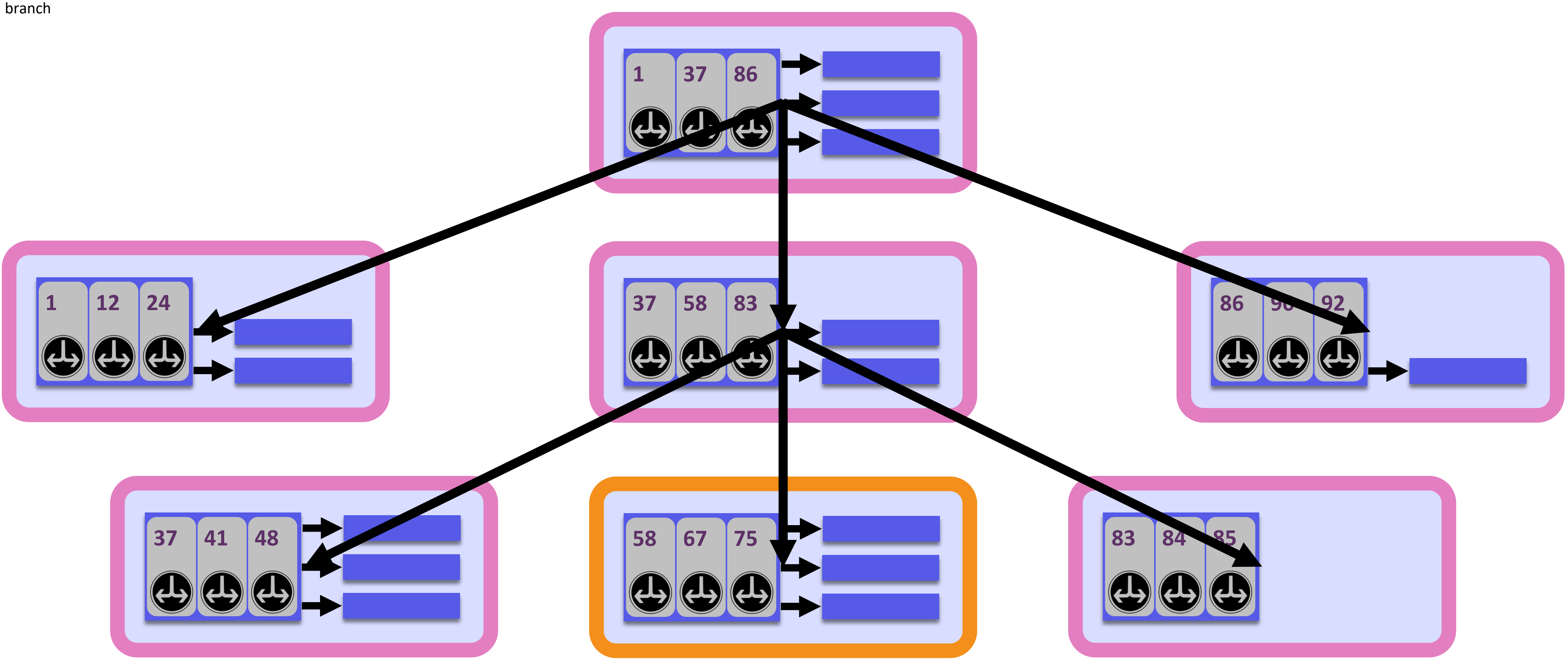
Query(71)



Size-Tiered B$^\varepsilon$-Trees

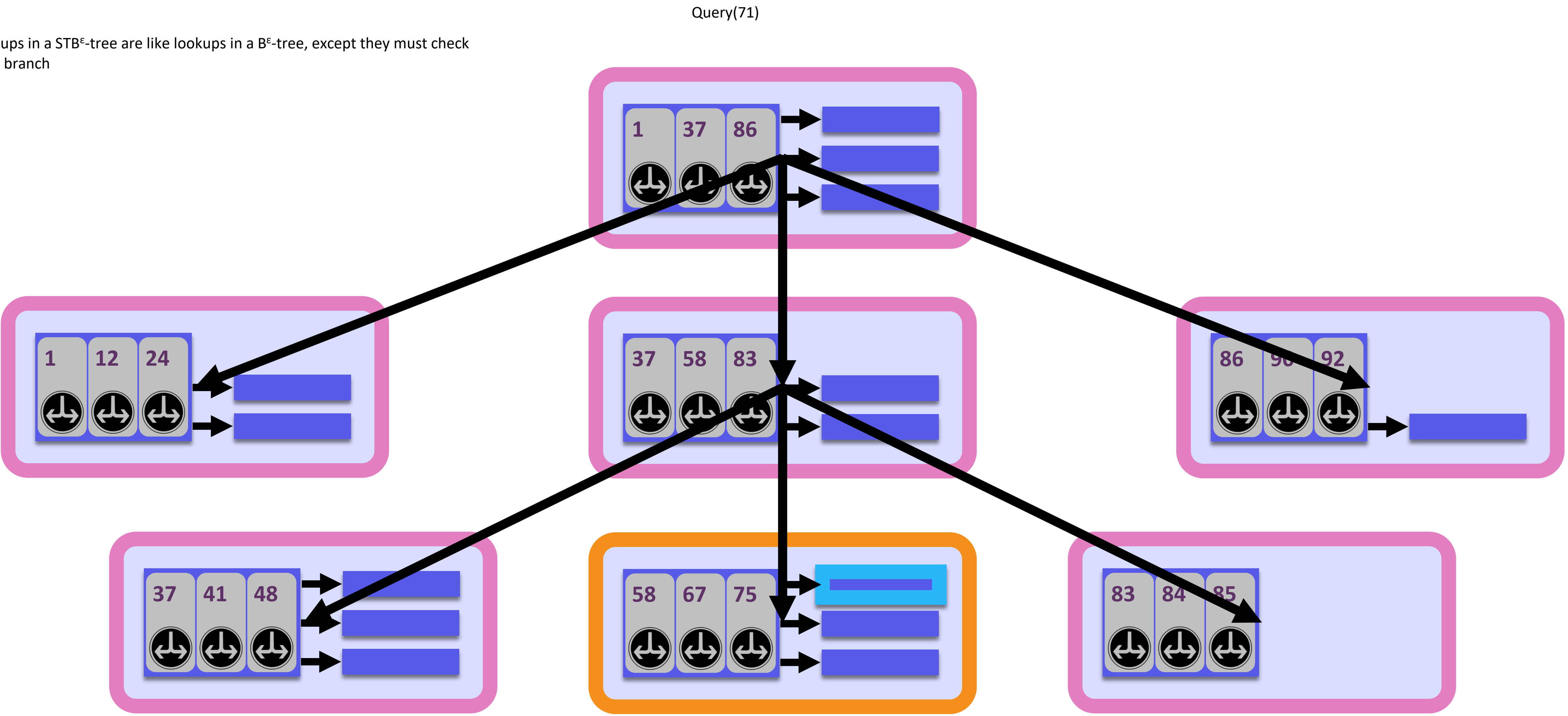
Lookups in a STB$^\varepsilon$-tree are like lookups in a B$^\varepsilon$-tree, except they must check each branch

Query(71)



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

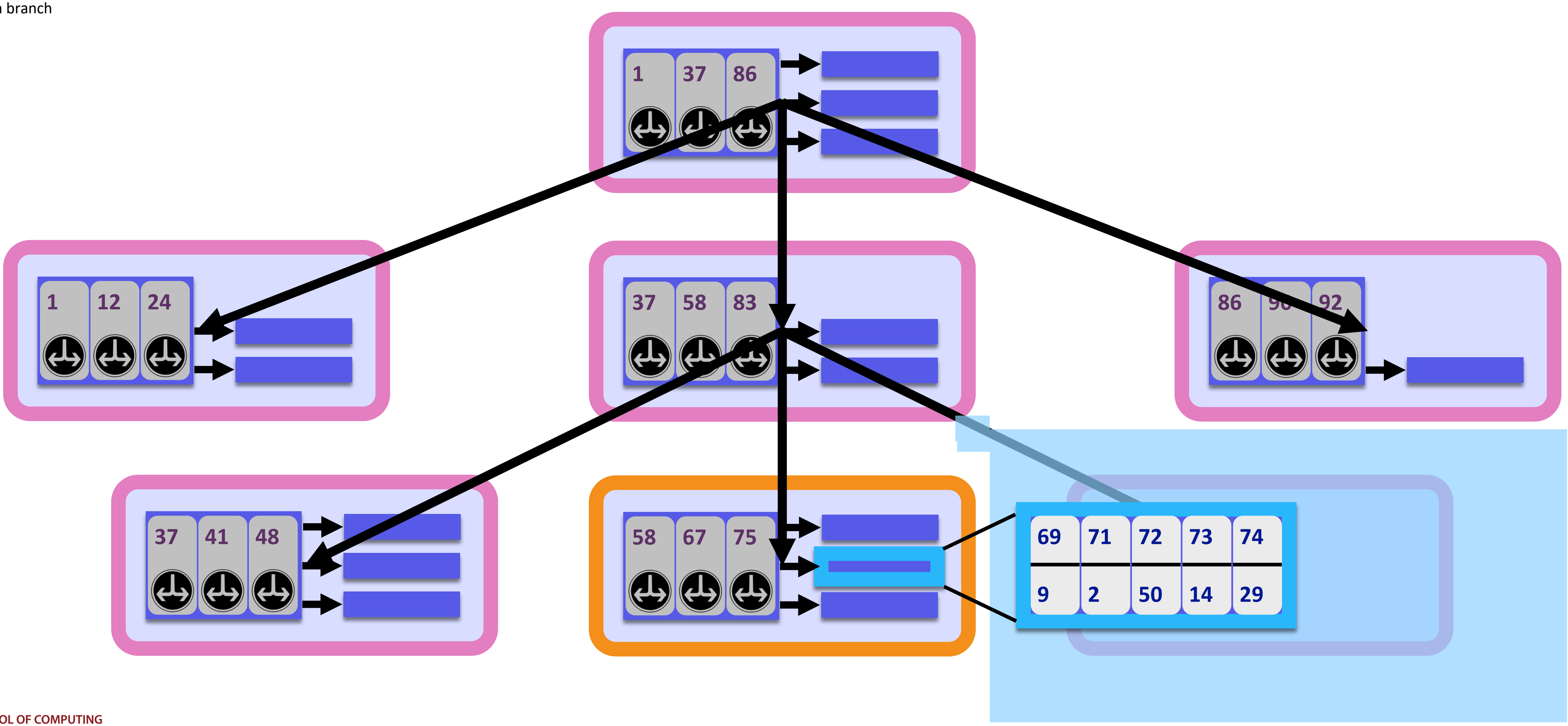
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

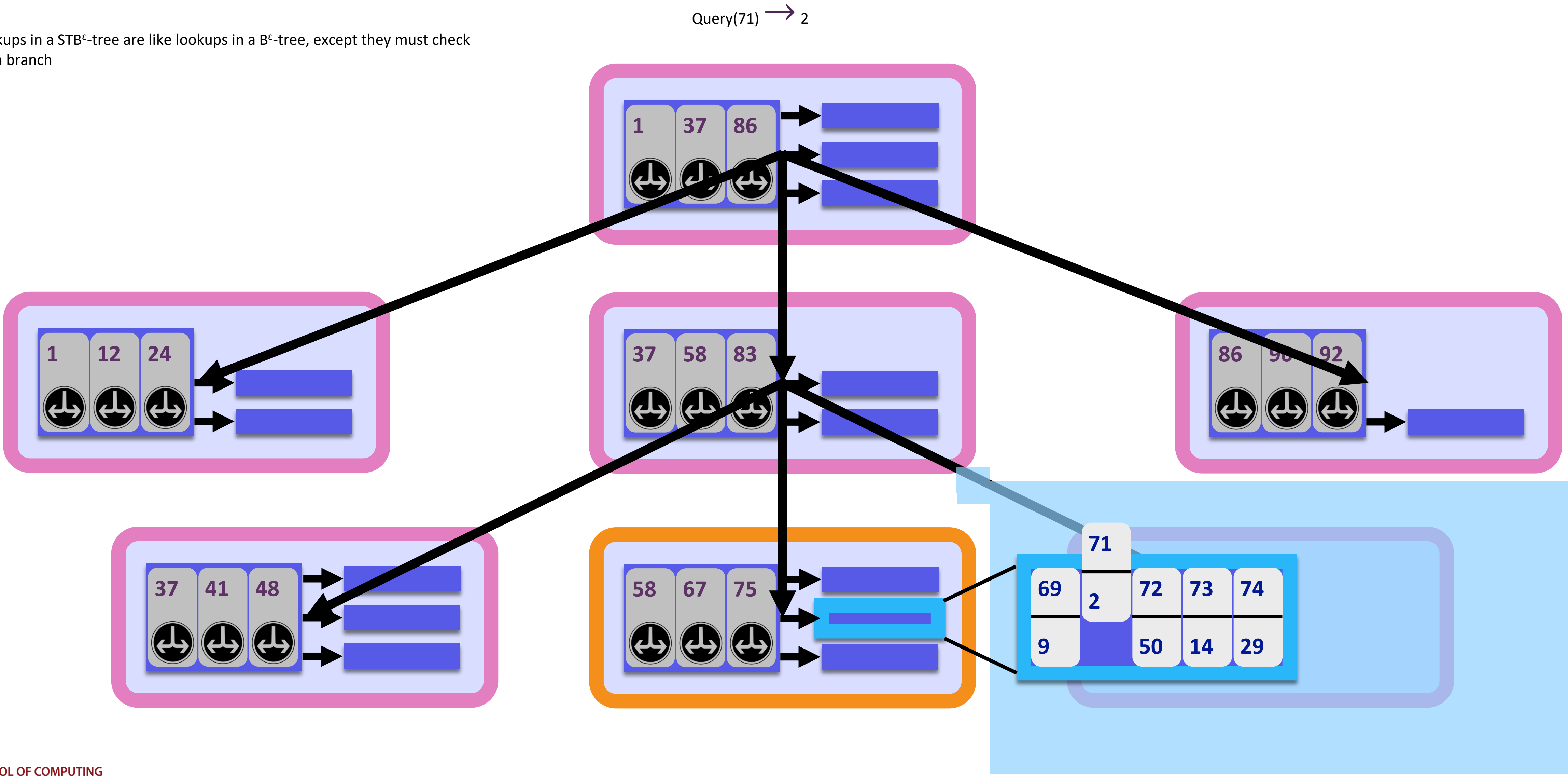
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

Query(71)



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

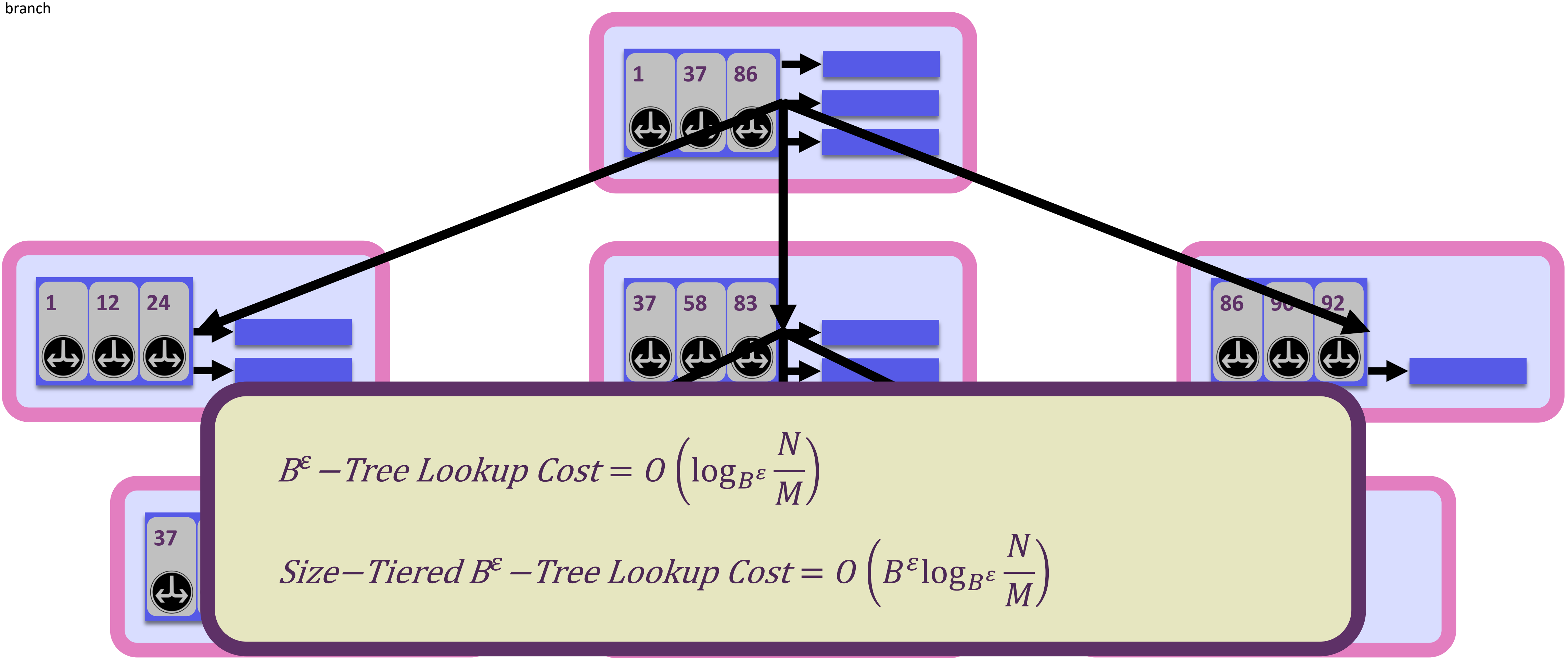
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

Query(71)

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



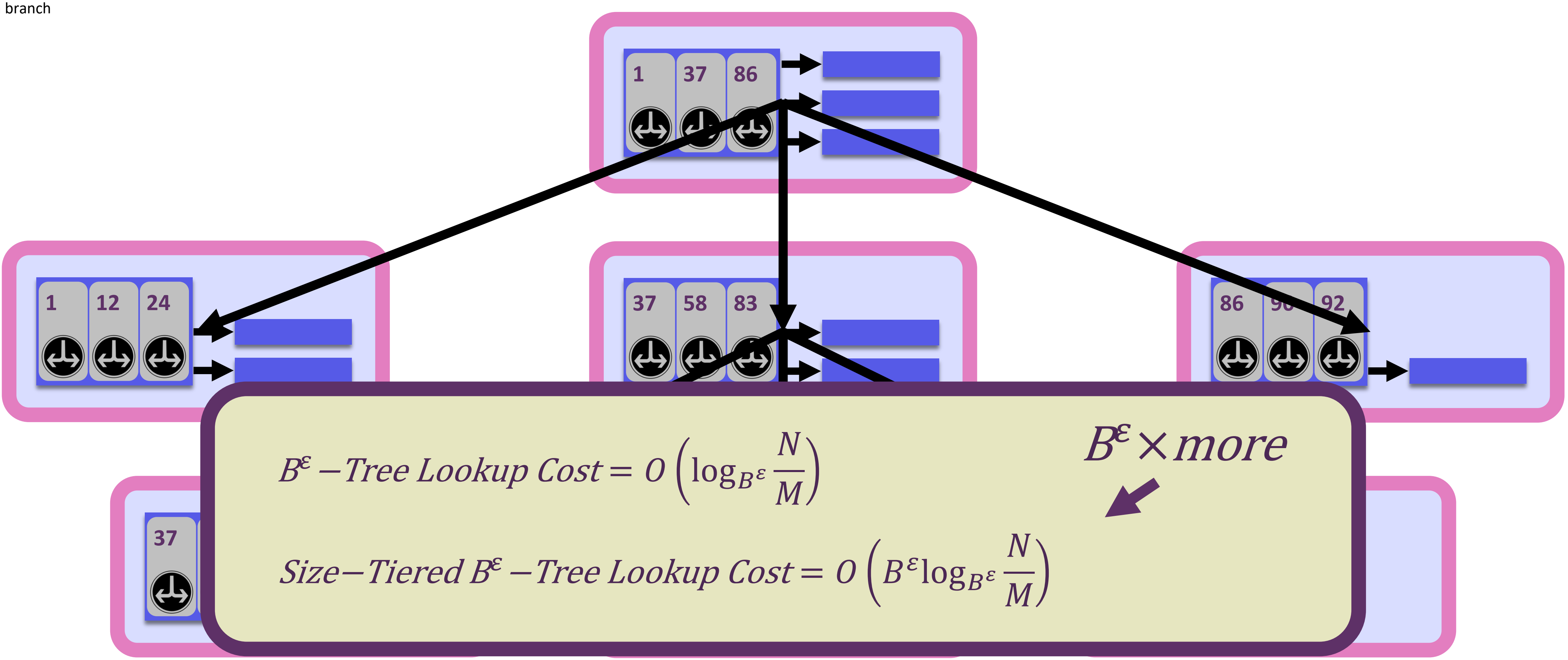
$$B^{\varepsilon} - Tree\ Lookup\ Cost = O\left(\log_{B^{\varepsilon}} \frac{N}{M}\right)$$

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

Size-Tiered B^ε-Trees

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

Query(71)



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

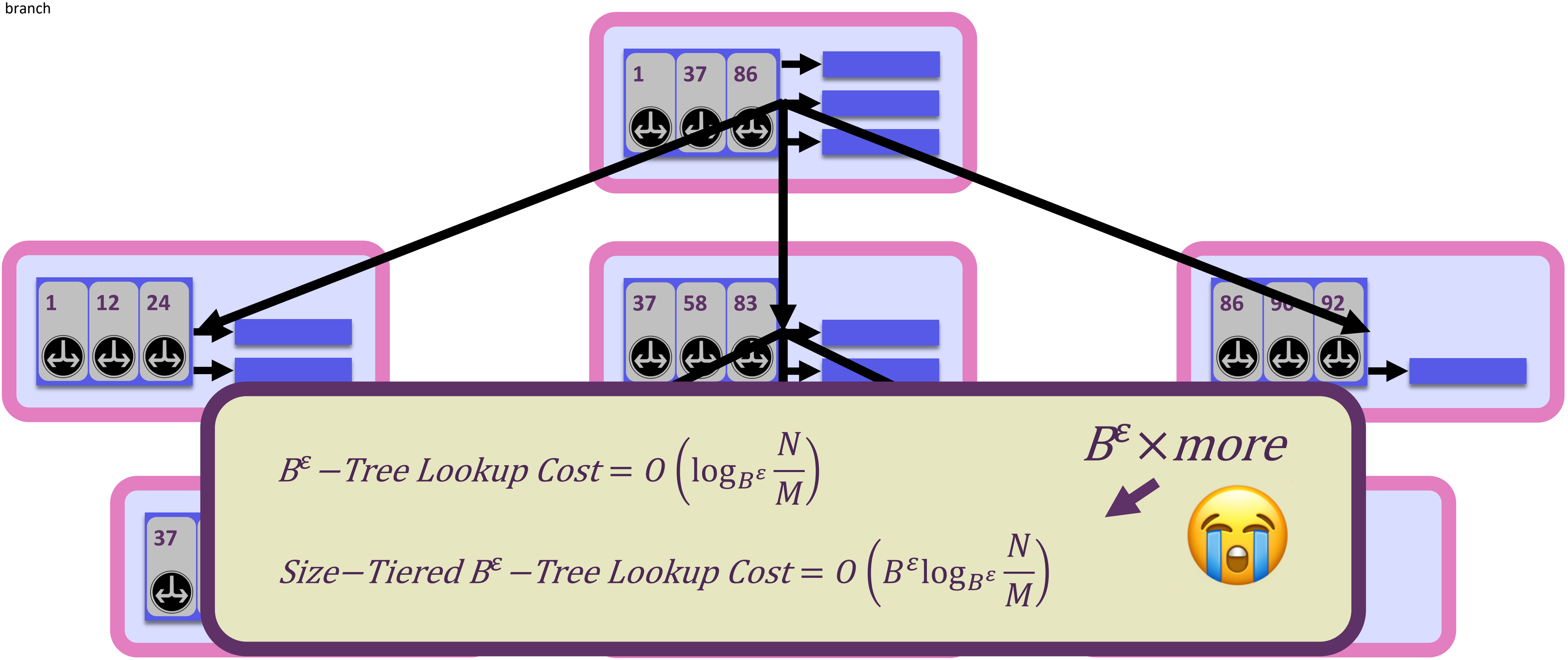
$B^{\varepsilon} \times more$

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

Size-Tiered B^ε-Trees

Query(71)

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



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

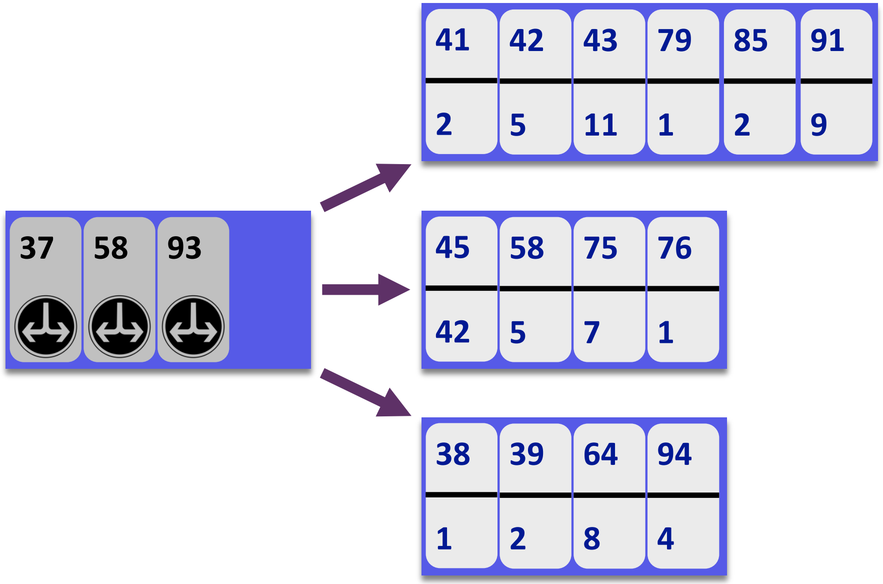
$$Size{-}Tiered\ B^\varepsilon - Tree\ Lookup\ Cost = O\left(B^\varepsilon \log_{B^\varepsilon}\frac{N}{M}\right)$$

$B^\varepsilon \times more$ 😭

Fixing Lookups (almost)

Fixing Lookups (almost)

The problem is that each node has multiple branches



37	58	93

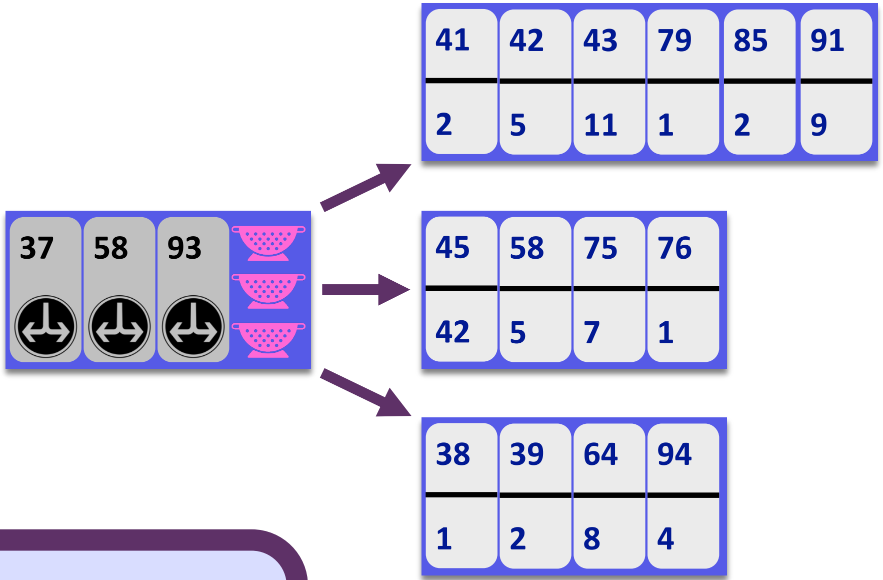
41	42	43	79	85	91
2	5	11	1	2	9

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

Fixing Lookups (almost)

The problem is that each node has multiple branches



37	58	93

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

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

Idea: use filters to avoid searching them

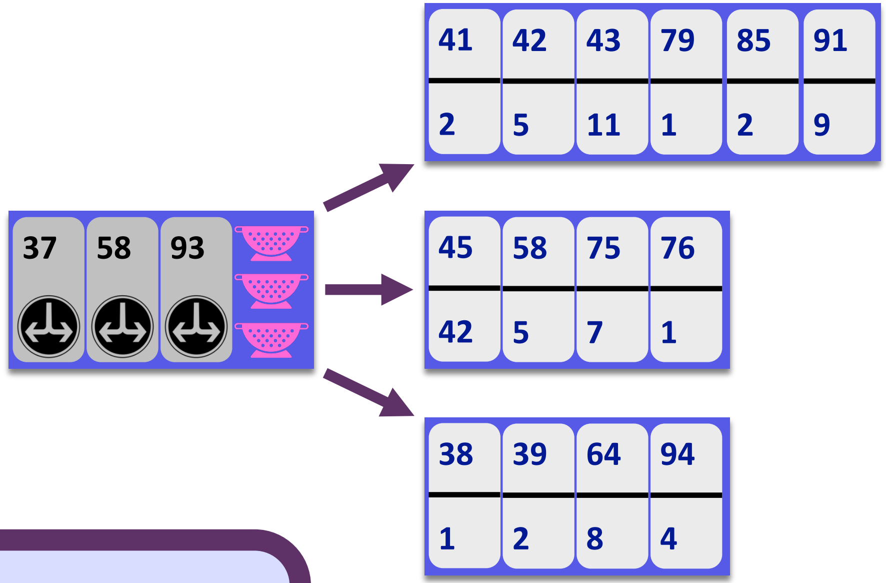
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

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

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)

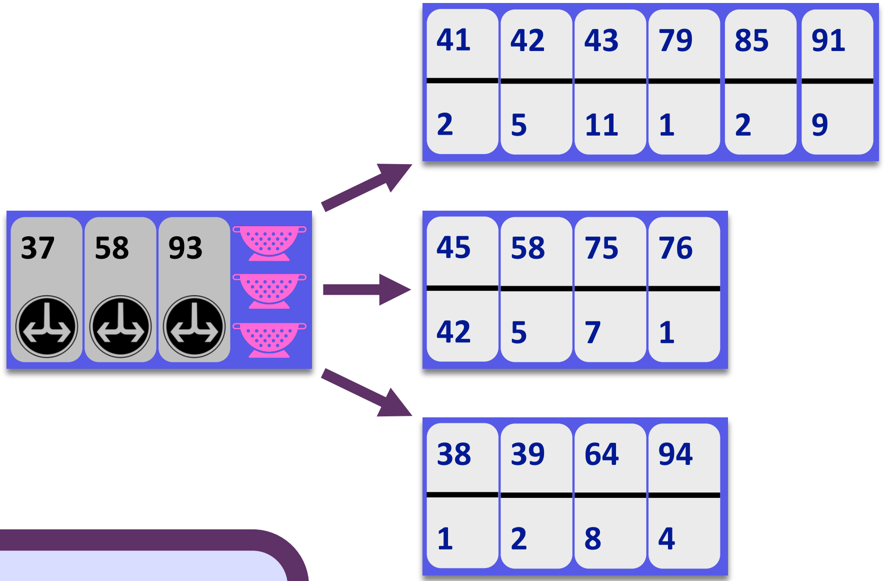
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches



37	58	93

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

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)

Idea: use filters to avoid searching them

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

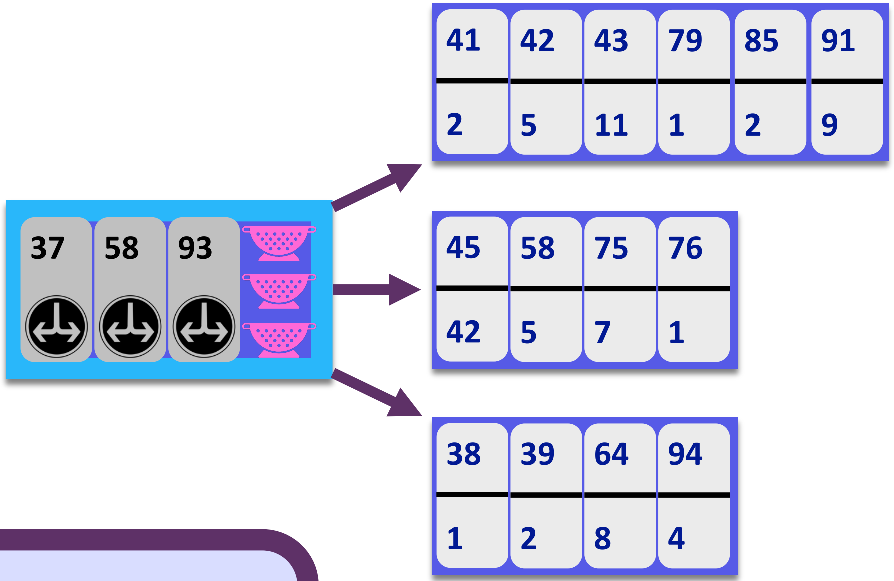
Examples: Bloom, cuckoo, quotient

Fixing Lookups (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

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)

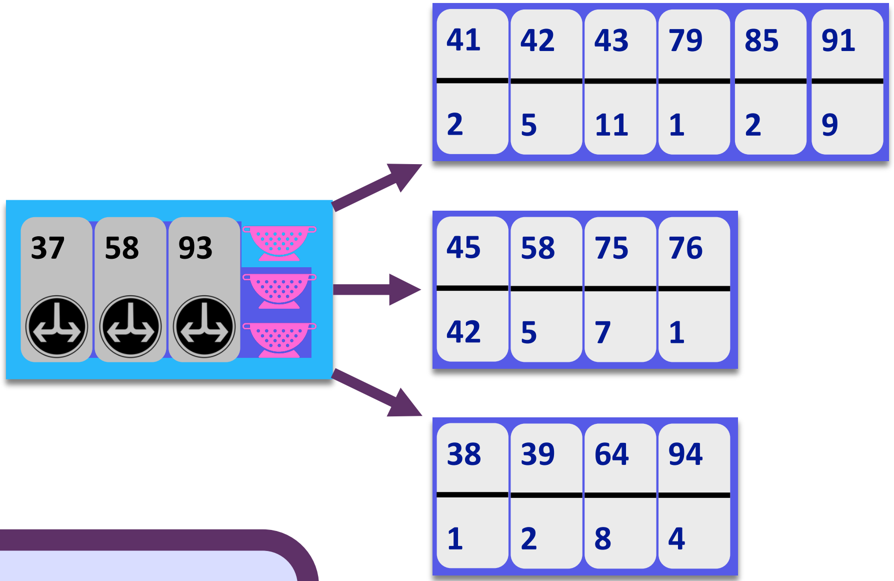
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches



37	58	93

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

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)

Idea: use filters to avoid searching them

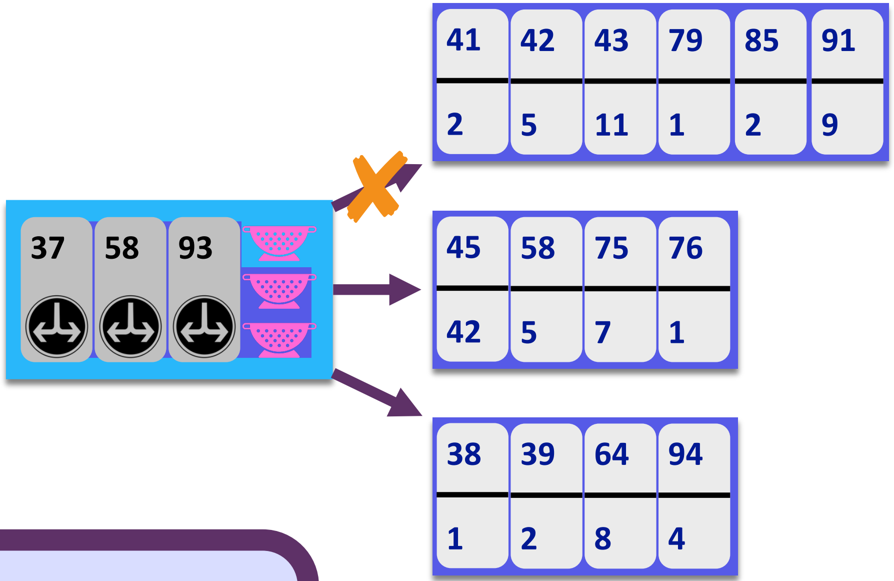
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches



37	58	93

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

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)

Idea: use filters to avoid searching them

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

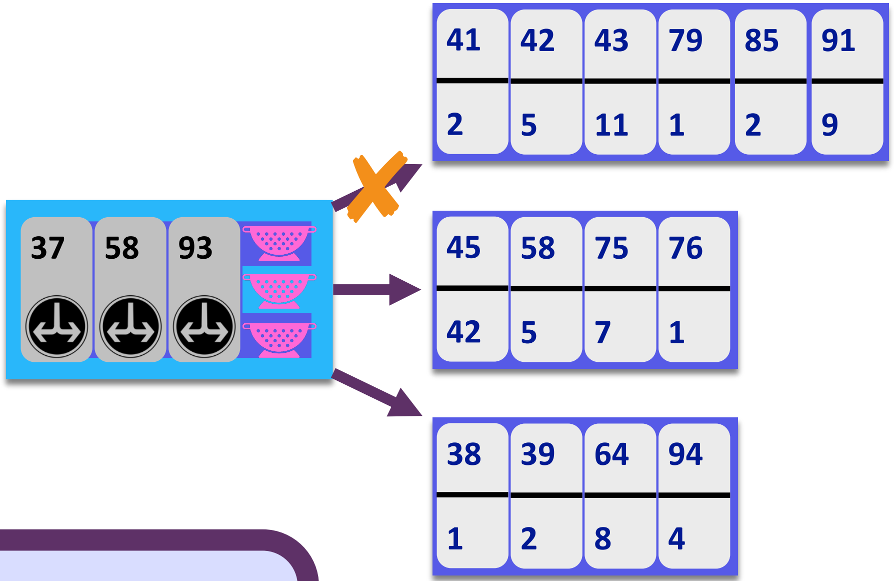
Examples: Bloom, cuckoo, quotient

Fixing Lookups (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

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

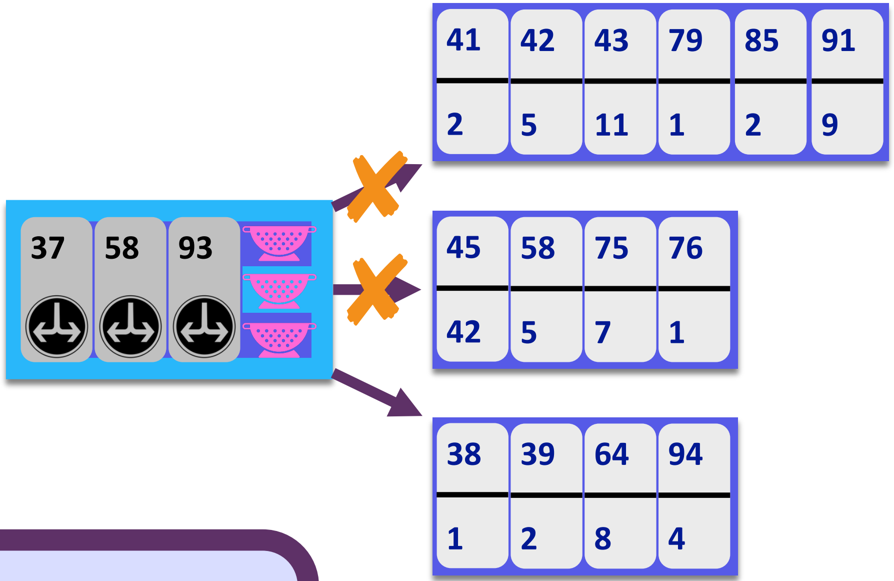
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



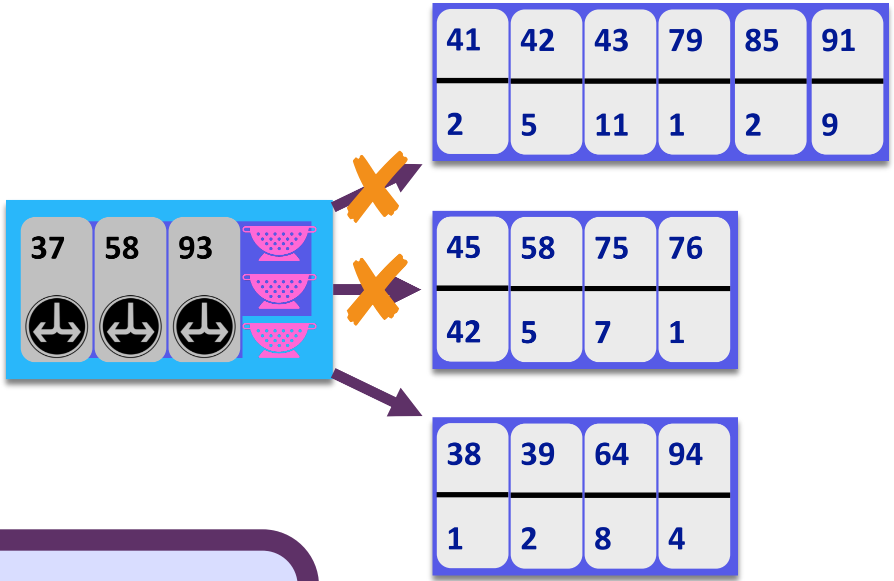
Now a lookup will only search those branches which contain the key (plus rare false positives)

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



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

Idea: use filters to avoid searching them

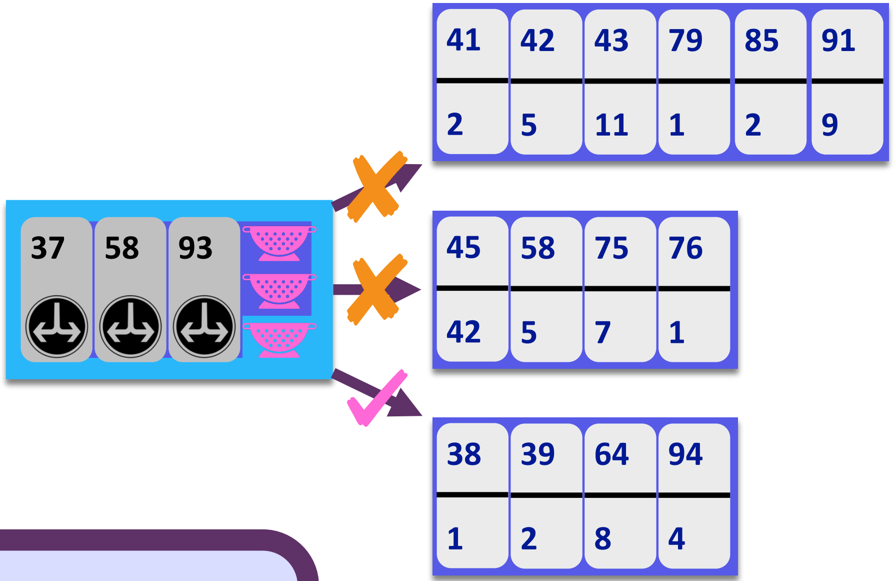
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches



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

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)

Idea: use filters to avoid searching them

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

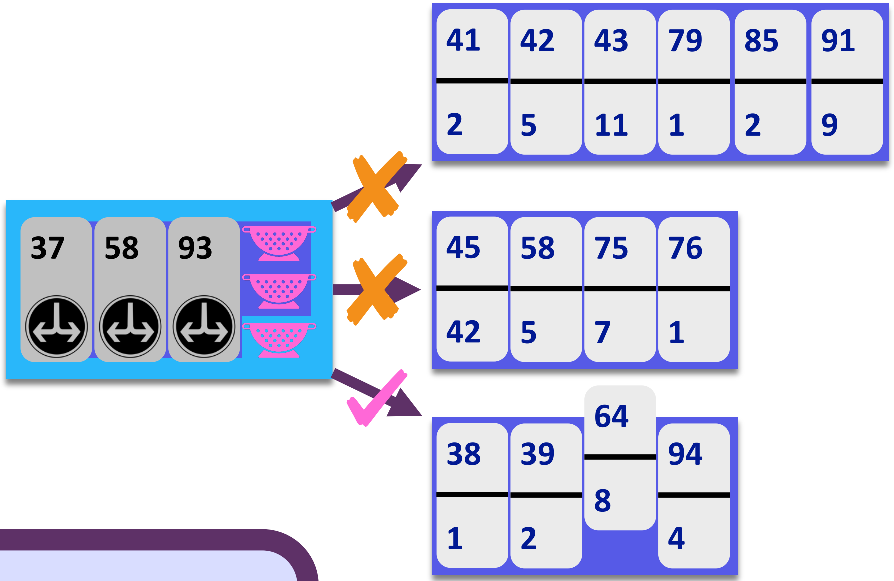
Examples: Bloom, cuckoo, quotient

Fixing Lookups (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

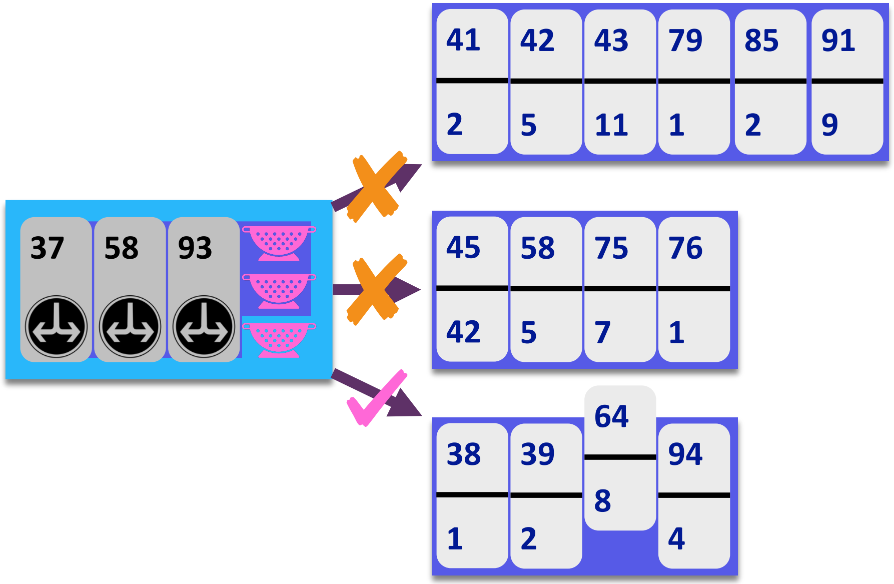
Examples: Bloom, cuckoo, quotient

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)

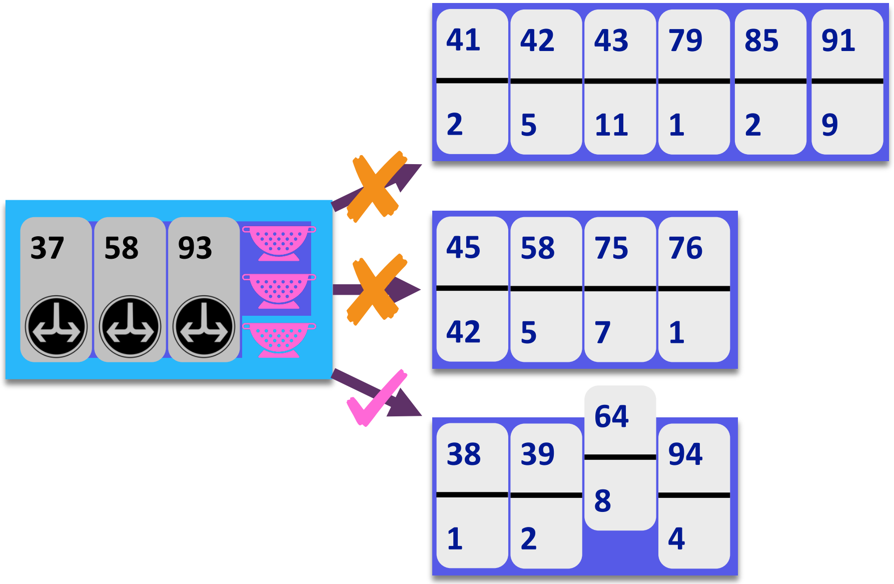
$$False\ Positive\ Rate \leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_B N}\right)$$

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)

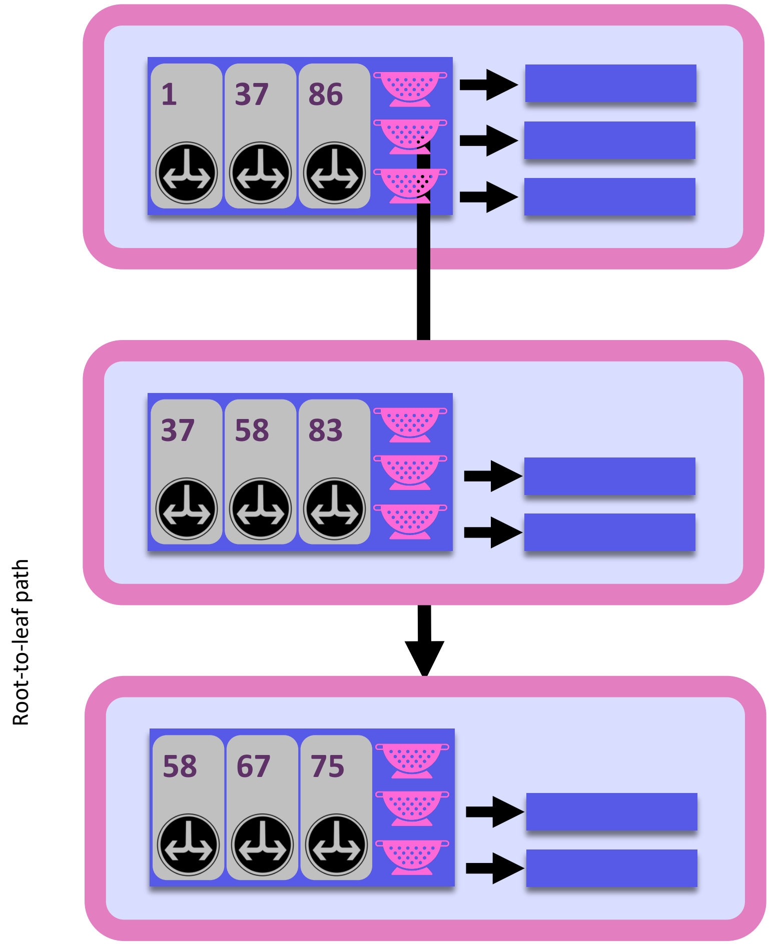
$$\mathit{False\ Positive\ Rate} \leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_B N}\right) \quad \Longrightarrow$$

Lookups in O(1) IOs

Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

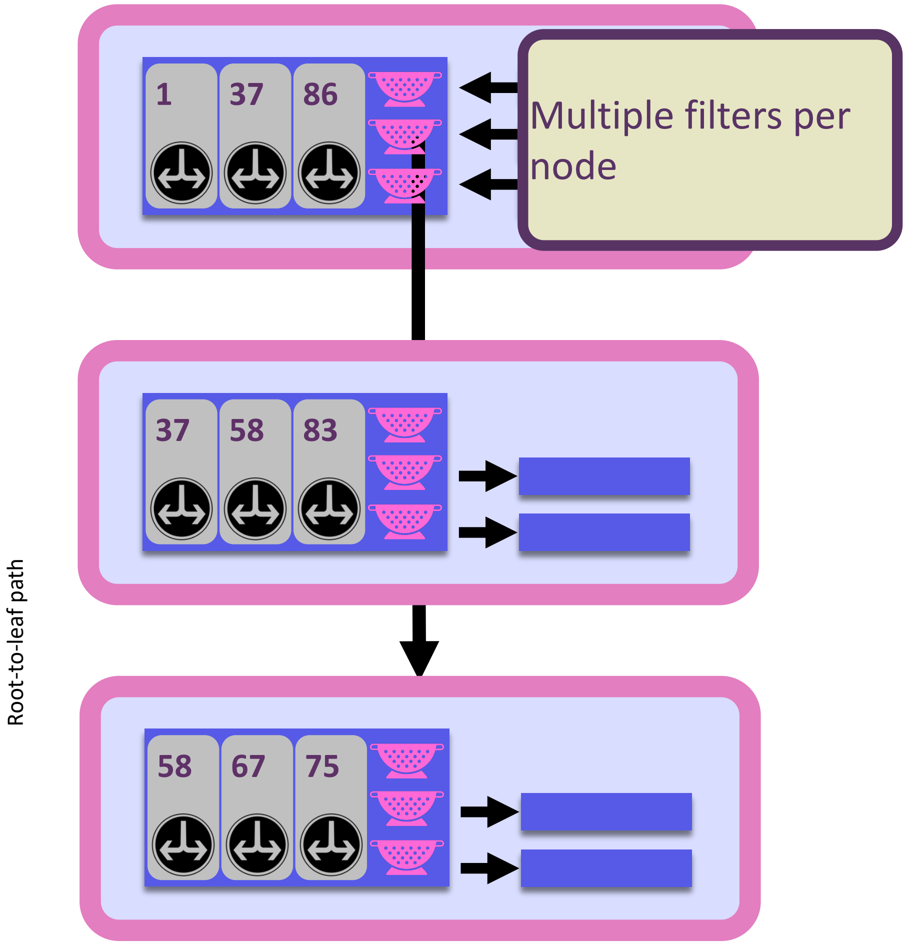
Querying all these filters is expensive



Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

Querying all these filters is expensive

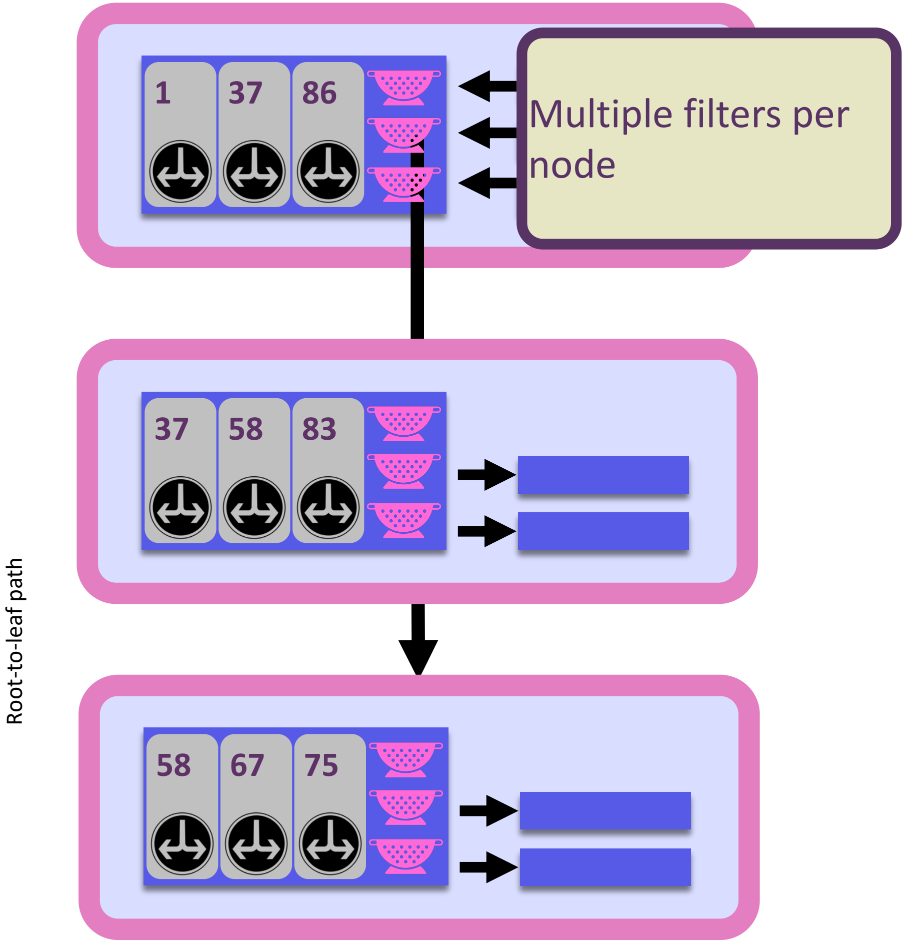
In practice, we see 15-40 filter lookups per point query



Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

Querying all these filters is expensive

In practice, we see 15-40 filter lookups per point query



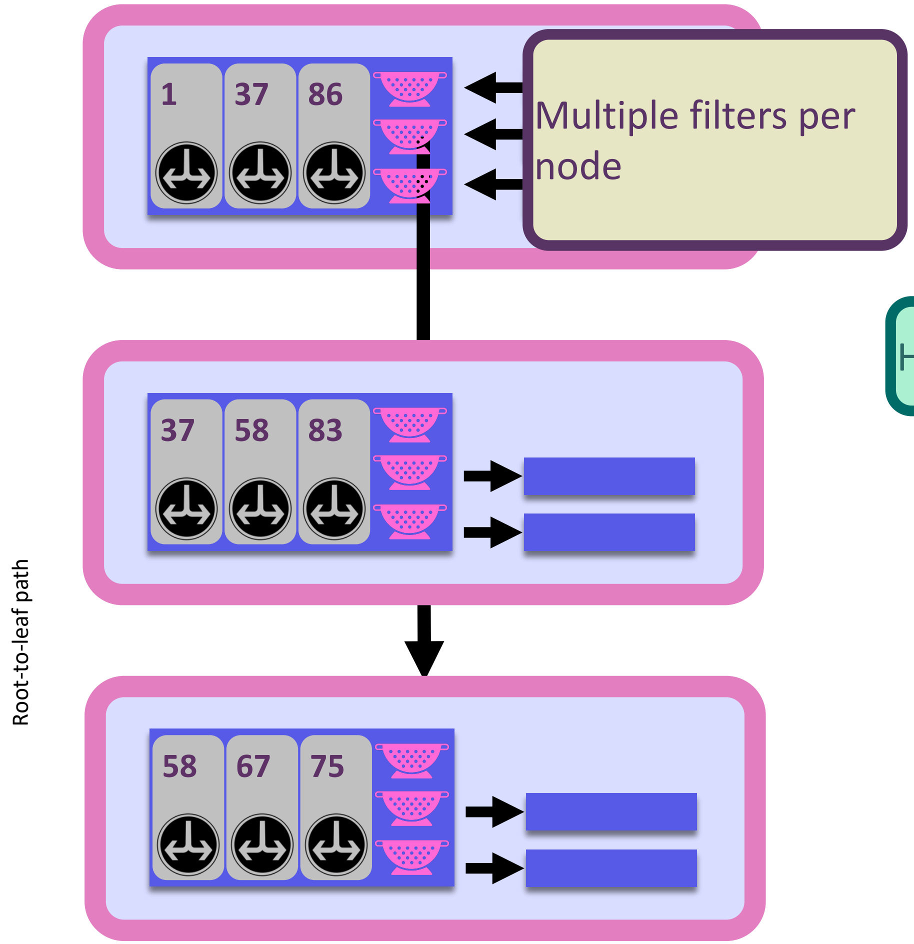
We could hope to amortize against IO

BUT…

Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

Querying all these filters is expensive

In practice, we see 15-40 filter lookups per point query



Multiple filters per node

We could hope to amortize against IO

BUT…

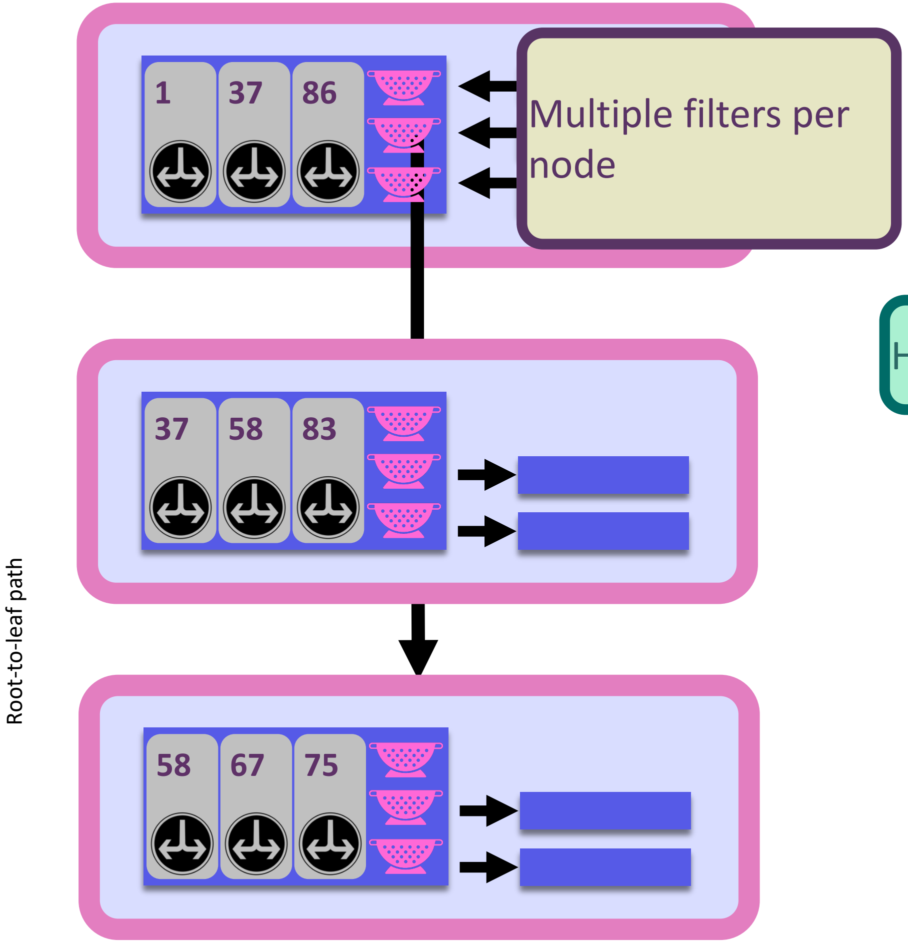
High Memory/Hot Queries

No IO, performance limited by CPU

Really Fixing Lookups in Size-Tiered B$^{\varepsilon}$-Trees

Querying all these filters is expensive

In practice, we see 15-40 filter lookups per point query



Multiple filters per node

We could hope to amortize against IO

BUT…

High Memory/Hot Queries

No IO, performance limited by CPU

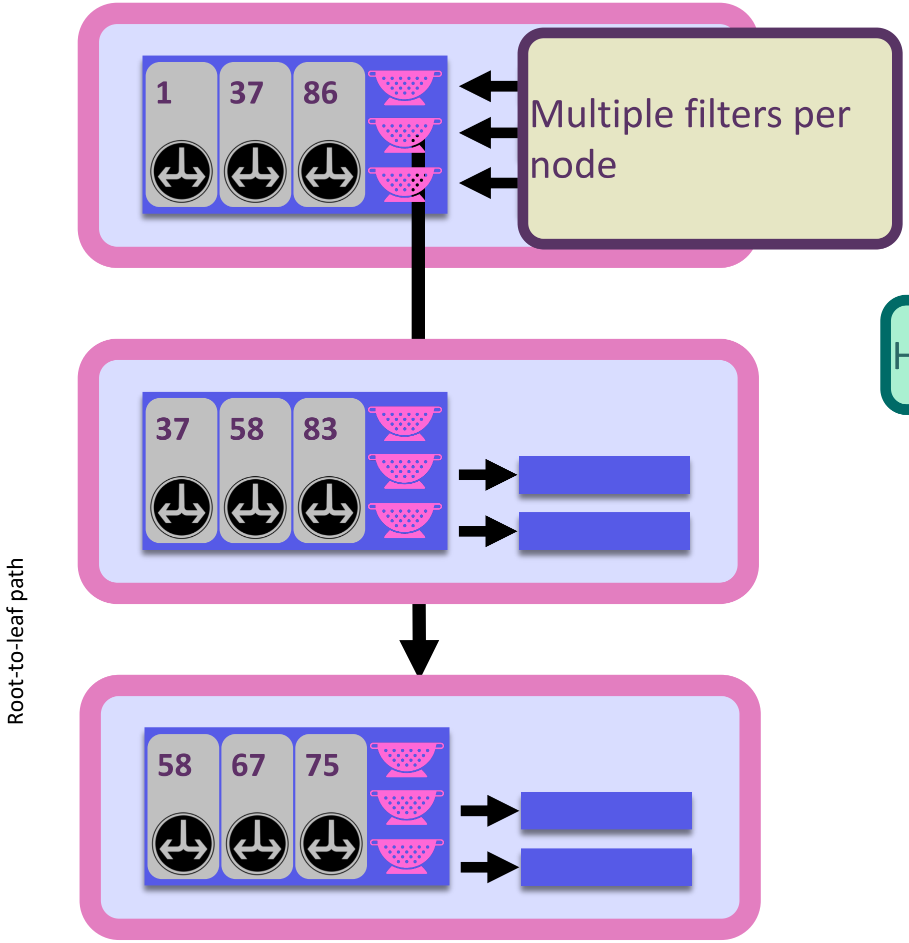
Medium Memory

1 IO per query,
CPU cost of filter lookups ⇒ more threads

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

Querying all these filters is expensive

In practice, we see 15-40 filter lookups per point query



We could hope to amortize against IO

BUT…

High Memory/Hot Queries — No IO, performance limited by CPU

Medium Memory — 1 IO per query,
CPU cost of filter lookups ⇒ more threads

Low Memory — Filters paged out to storage,
Lookup performance degrades

Maplets

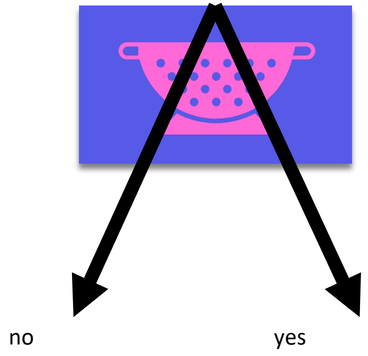
Maplets

A maplet is a filter which can also store small values

Maplets

A maplet is a filter which can also store small values

Is X in the set?

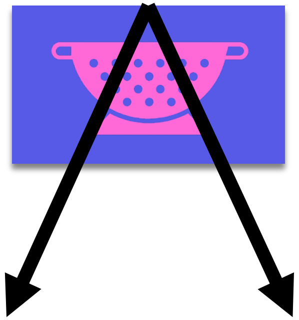


Filter

Maplets

A maplet is a filter which can also store small values

Is X in the set?

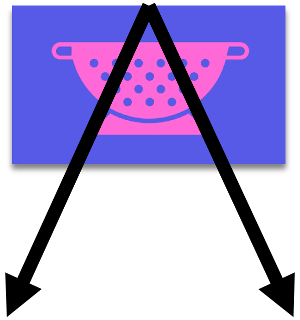


no

yes

Filter

Is X in the set?



no

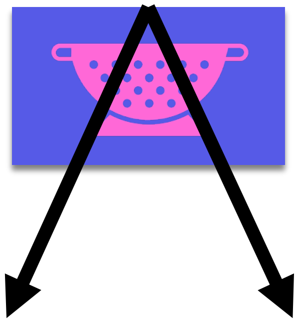
yes, 4

Maplet

Maplets

A maplet is a filter which can also store small values

Is X in the set?

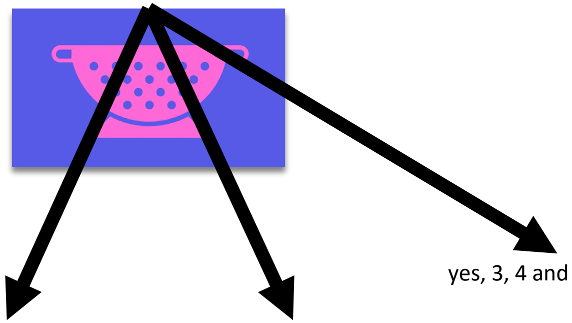


no

yes

Filter

Is X in the set?



no

yes, 4

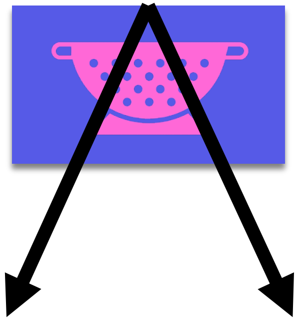
yes, 3, 4 and 7

Maplet

Maplets

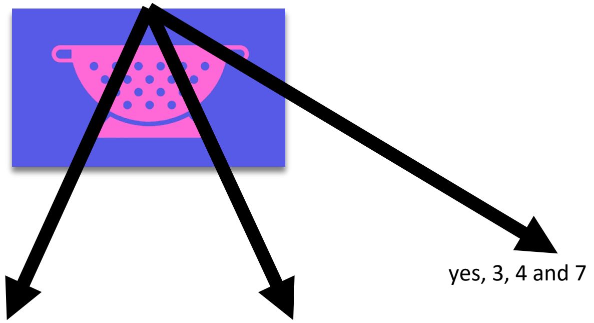
A maplet is a filter which can also store small values

Is X in the set?



Filter

Is X in the set?



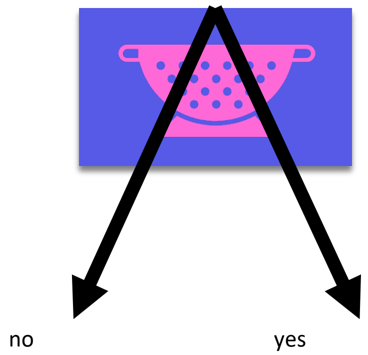
Maplet

No false negatives, same false positive guarantee

Maplets

A maplet is a filter which can also store small values

Is X in the set?

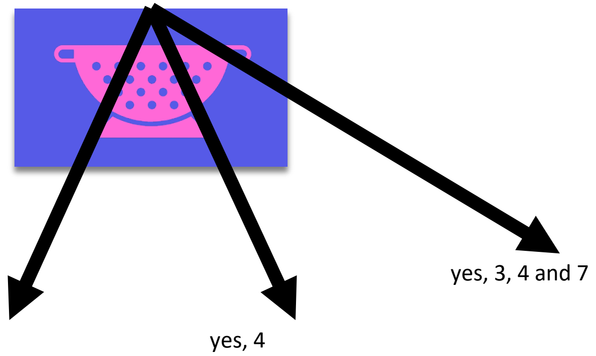


no

yes

Filter

Is X in the set?



no

yes, 4

yes, 3, 4 and 7

Maplet

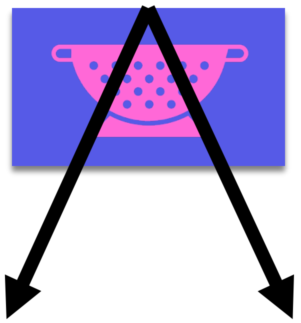
No false negatives, same false positive guarantee

Same memory footprint as multiple filters

Maplets

A maplet is a filter which can also store small values

Is X in the set?

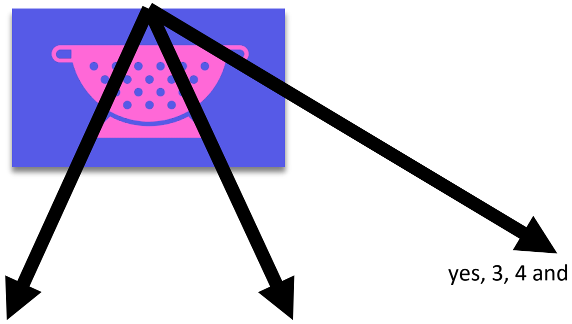


no

yes

Filter

Is X in the set?



no

yes, 4

yes, 3, 4 and 7

Maplet

No false negatives, same false positive guarantee

Same memory footprint as multiple filters

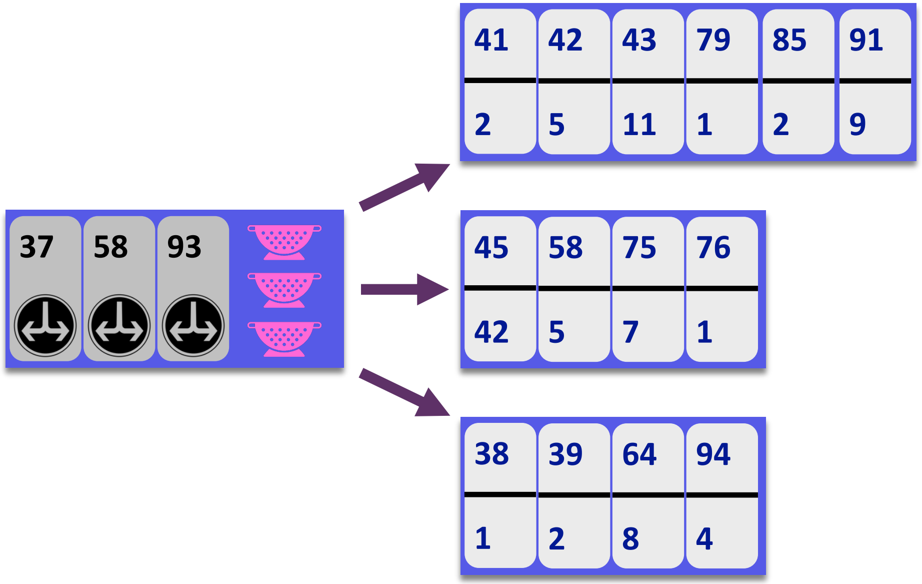
Lookups same cost as 1 quotient filter:
2 cache line misses
1 IO

Mapped B^{ε}-Trees

SplinterDB and Maplets: Improving the Trade-Offs in LSM Compaction Policy
Conway, Farach-Colton, Johnson,
SIGMOD 2023

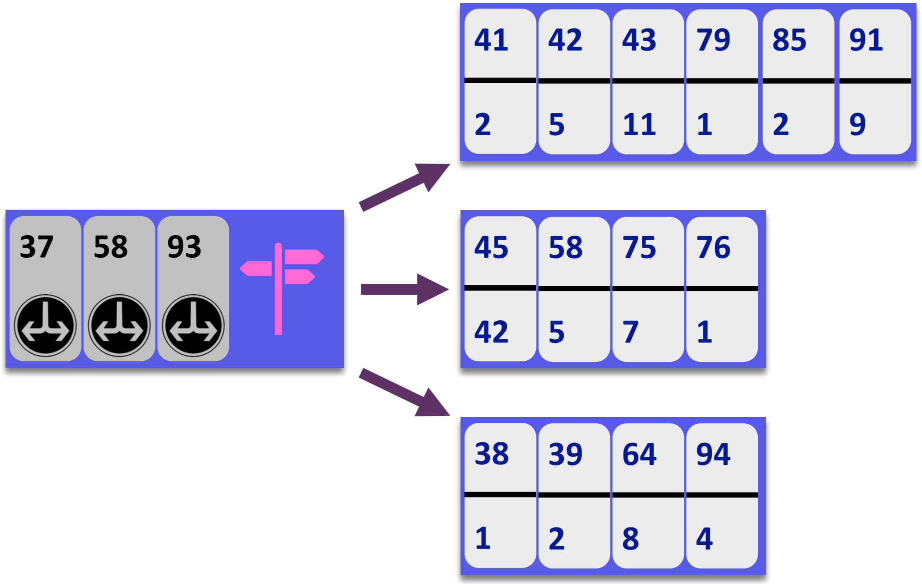
Mapped B^{ε}-Trees

Replace individual filters with a single maplet



Mapped B^{ε}-Trees

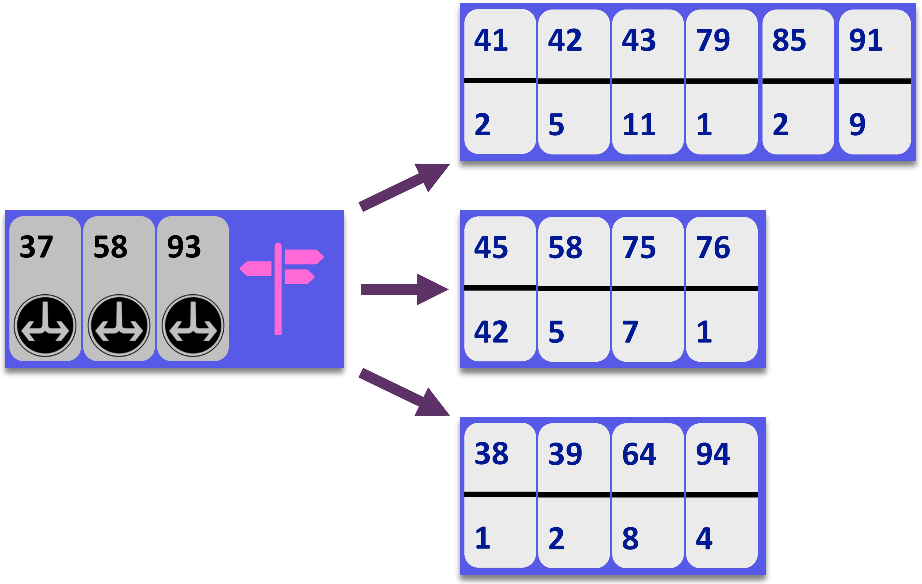
Replace individual filters with a single maplet



Mapped B^{ε}-Trees

Replace individual filters with a single maplet

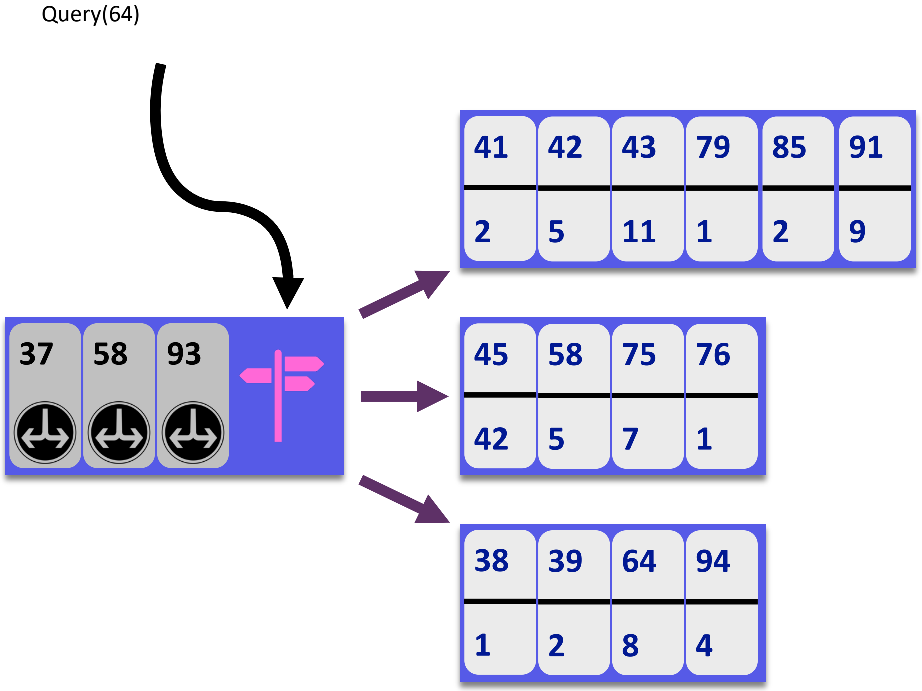
Use the values to store which buffers contain matching keys



Mapped B^{ε}-Trees

Replace individual filters with a single maplet

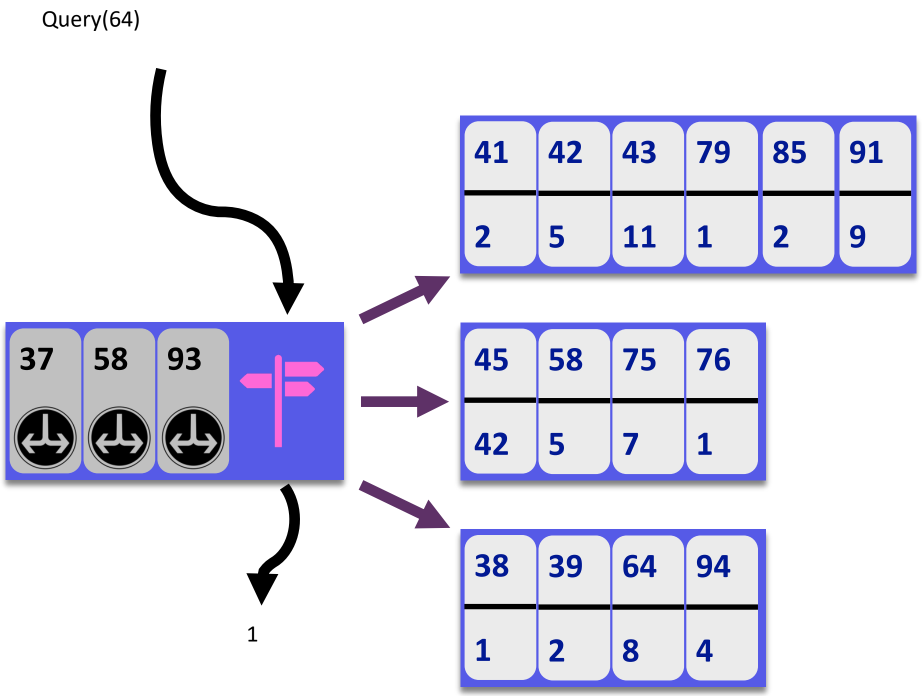
Use the values to store which buffers contain matching keys



Mapped B^{ε}-Trees

Replace individual filters with a single maplet

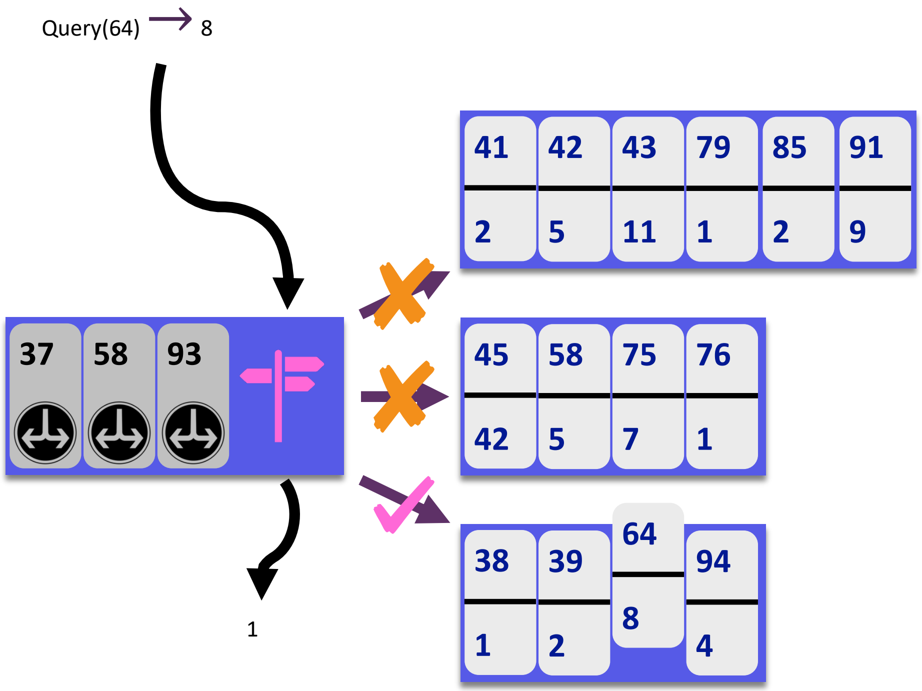
Use the values to store which buffers contain matching keys



Mapped B^{ε}-Trees

Replace individual filters with a single maplet

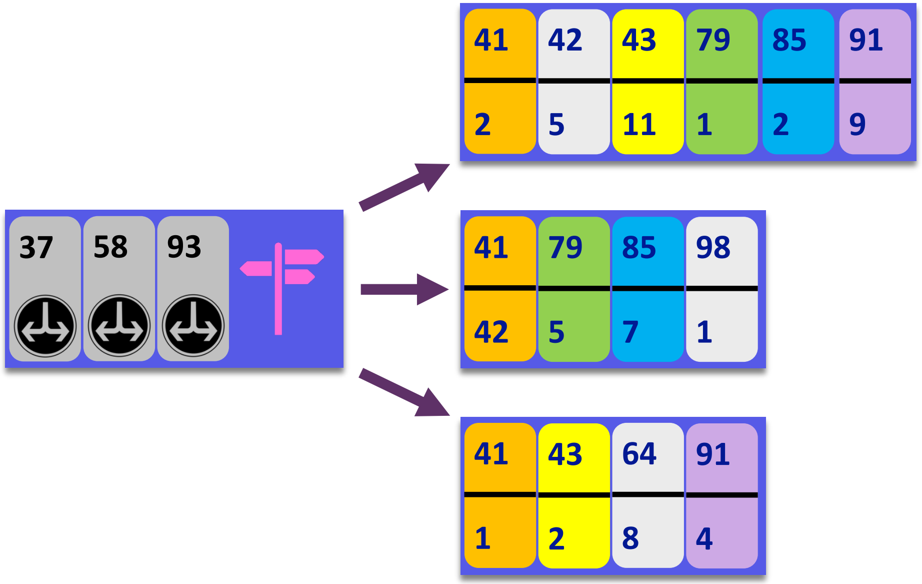
Use the values to store which buffers contain matching keys



Using Maplets to Manage Space

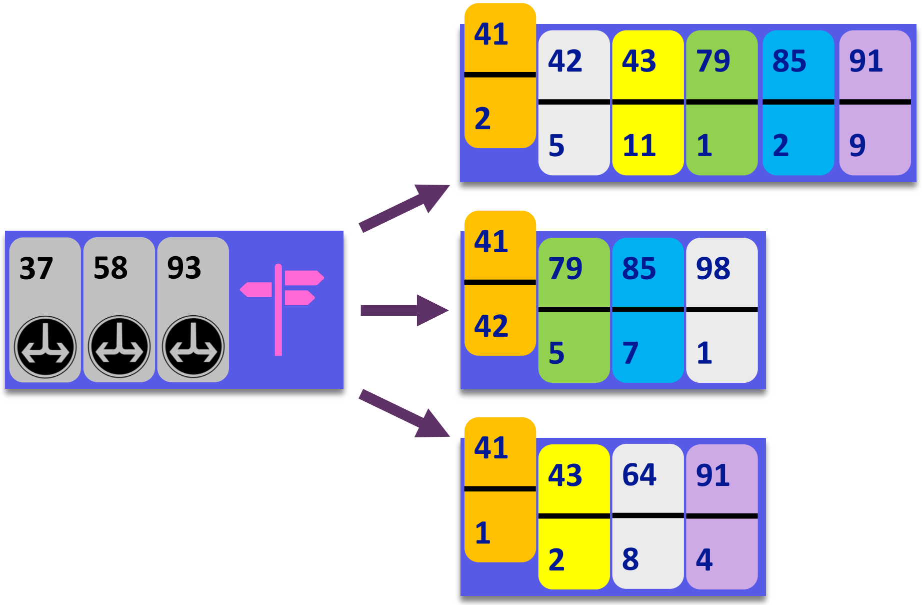
Using Maplets to Manage Space

Size-tiering can lead to redundant data, wasting space



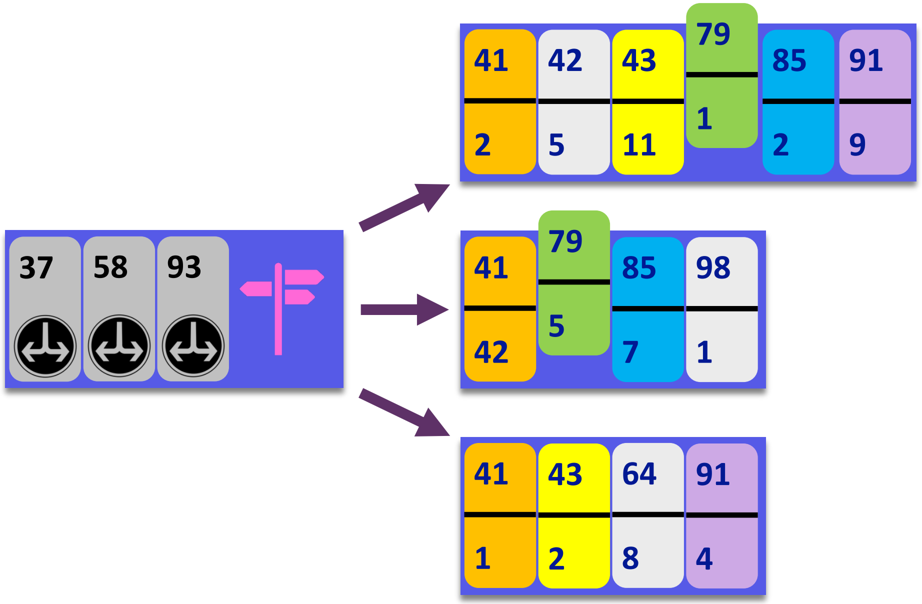
Using Maplets to Manage Space

Size-tiering can lead to redundant data, wasting space



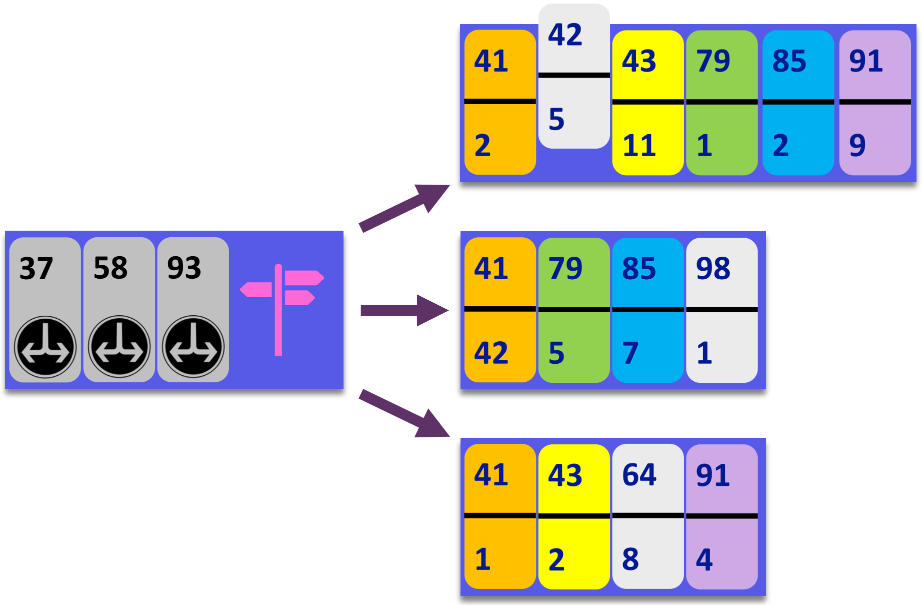
Using Maplets to Manage Space

Size-tiering can lead to redundant data, wasting space



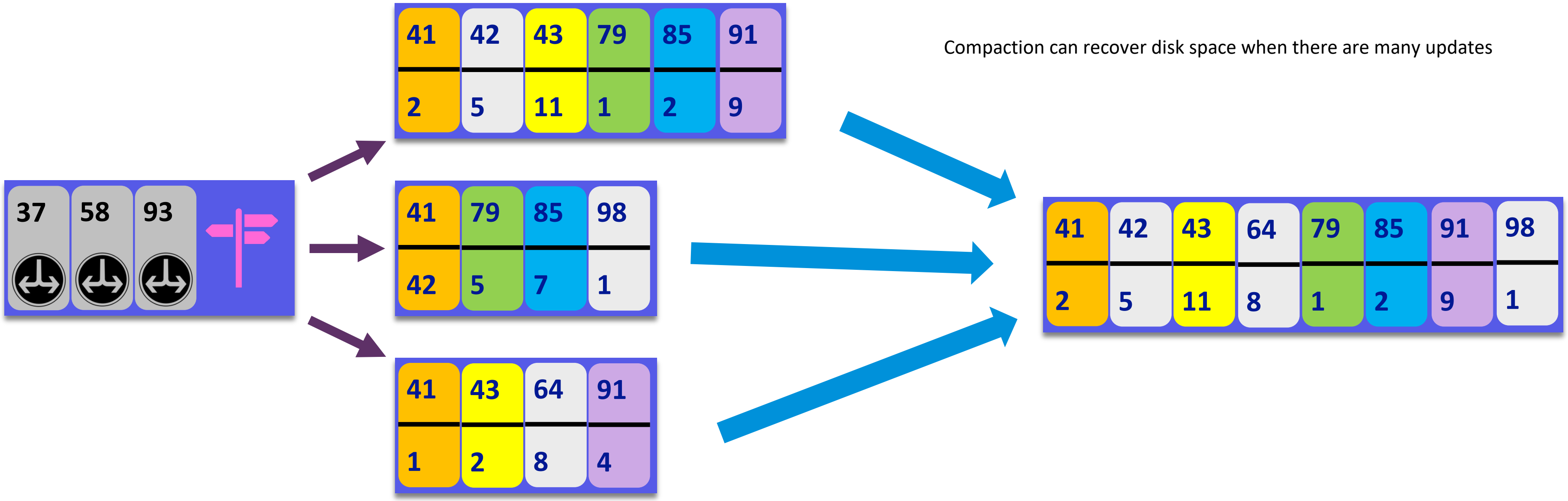
Using Maplets to Manage Space

Size-tiering can lead to redundant data, wasting space



Using Maplets to Manage Space

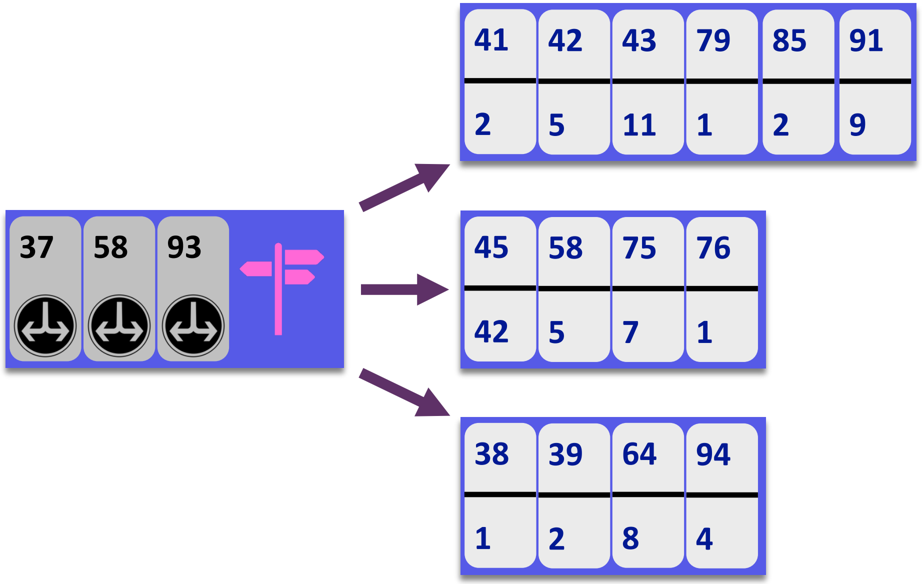
Size-tiering can lead to redundant data, wasting space



Compaction can recover disk space when there are many updates

Using Maplets to Manage Space

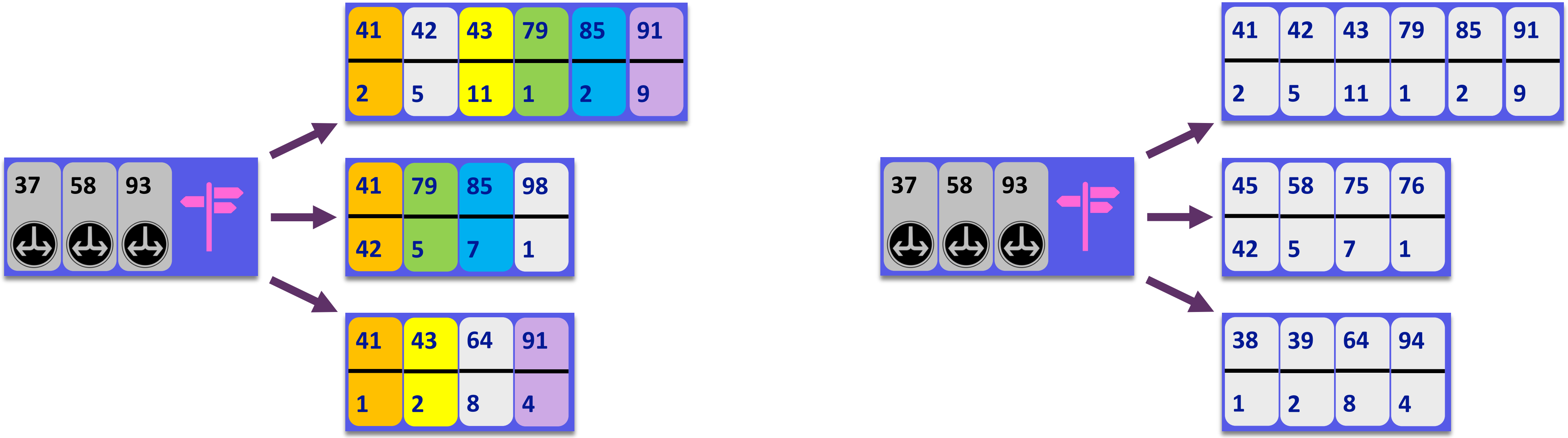
Compaction saves little space when there is little redundant data



So we don't want to waste time compacting branches with few updates

Using Maplets to Manage Space

Maplets can tell us how much redundant data there is



37	58	93

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

41	79	85	98
42	5	7	1

41	43	64	91
1	2	8	4

37	58	93

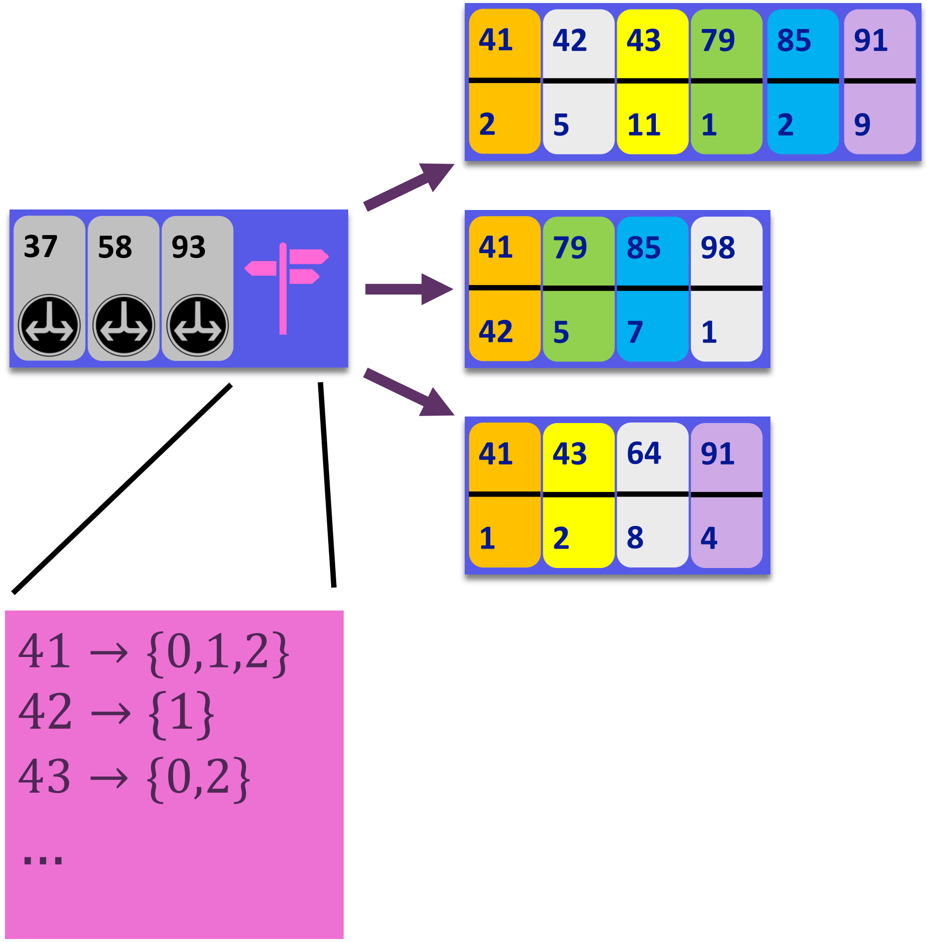
41	42	43	79	85	91
2	5	11	1	2	9

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

Using Maplets to Manage Space

Maplets can tell us how much redundant data there is

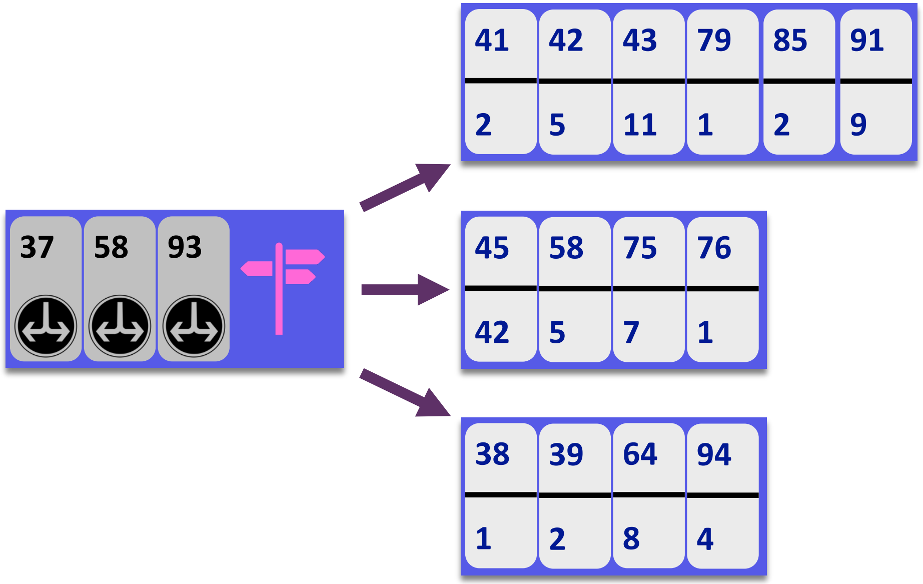


41	42	43	79	85	91
2	5	11	1	2	9

41	79	85	98
42	5	7	1

41	43	64	91
1	2	8	4

41 → {0,1,2}
42 → {1}
43 → {0,2}
…



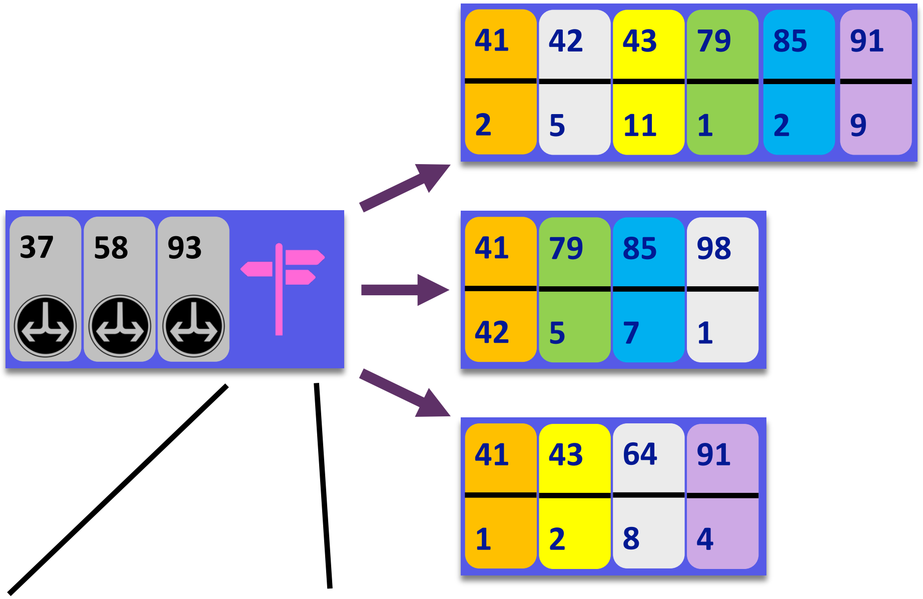
41	42	43	79	85	91
2	5	11	1	2	9

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

Using Maplets to Manage Space

Maplets can tell us how much redundant data there is



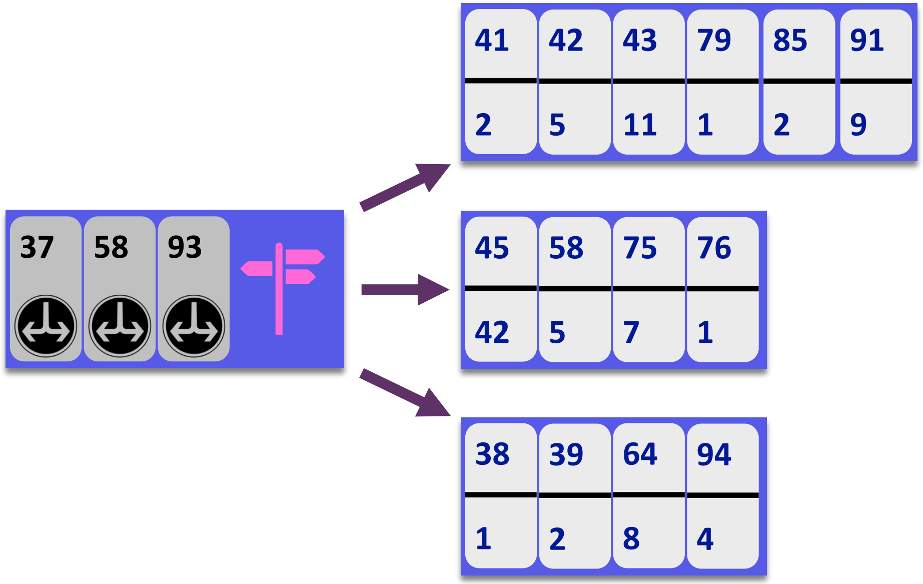
41	42	43	79	85	91
2	5	11	1	2	9

41	79	85	98
42	5	7	1

41	43	64	91
1	2	8	4

41 → {0,1,2}
42 → {1}
43 → {0,2}
…

Lots of multiple entries



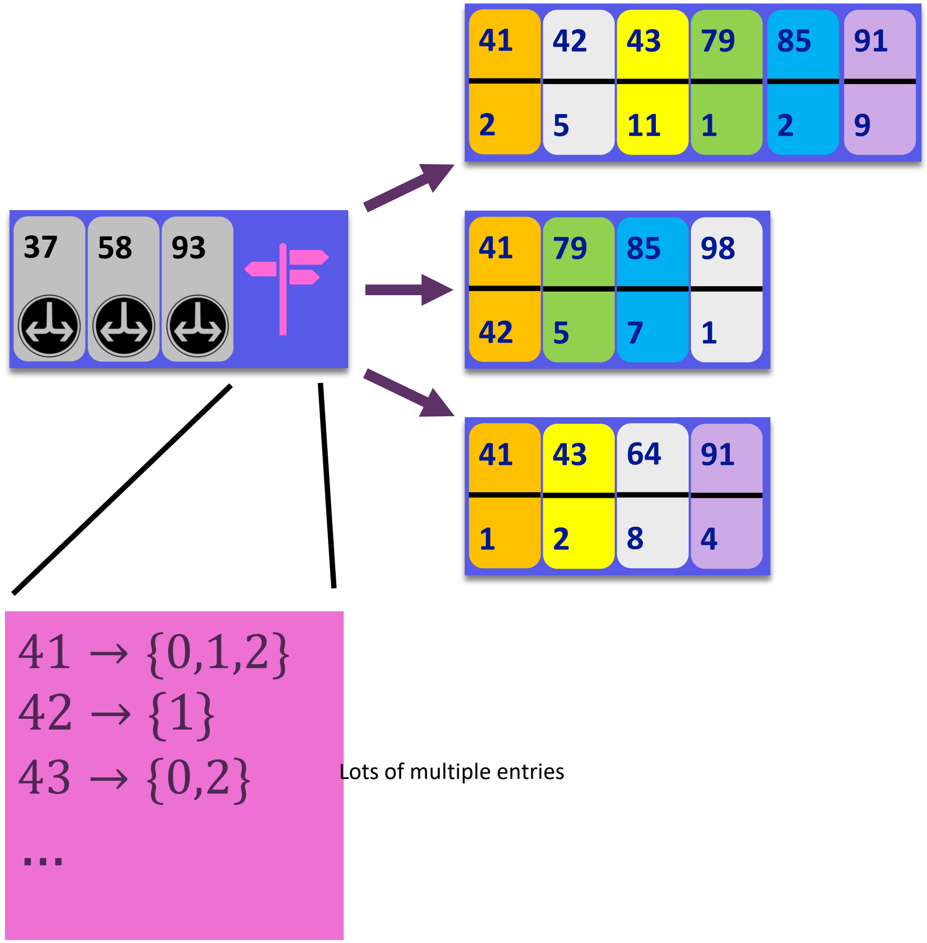
41	42	43	79	85	91
2	5	11	1	2	9

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

Using Maplets to Manage Space

Maplets can tell us how much redundant data there is



37	58	93

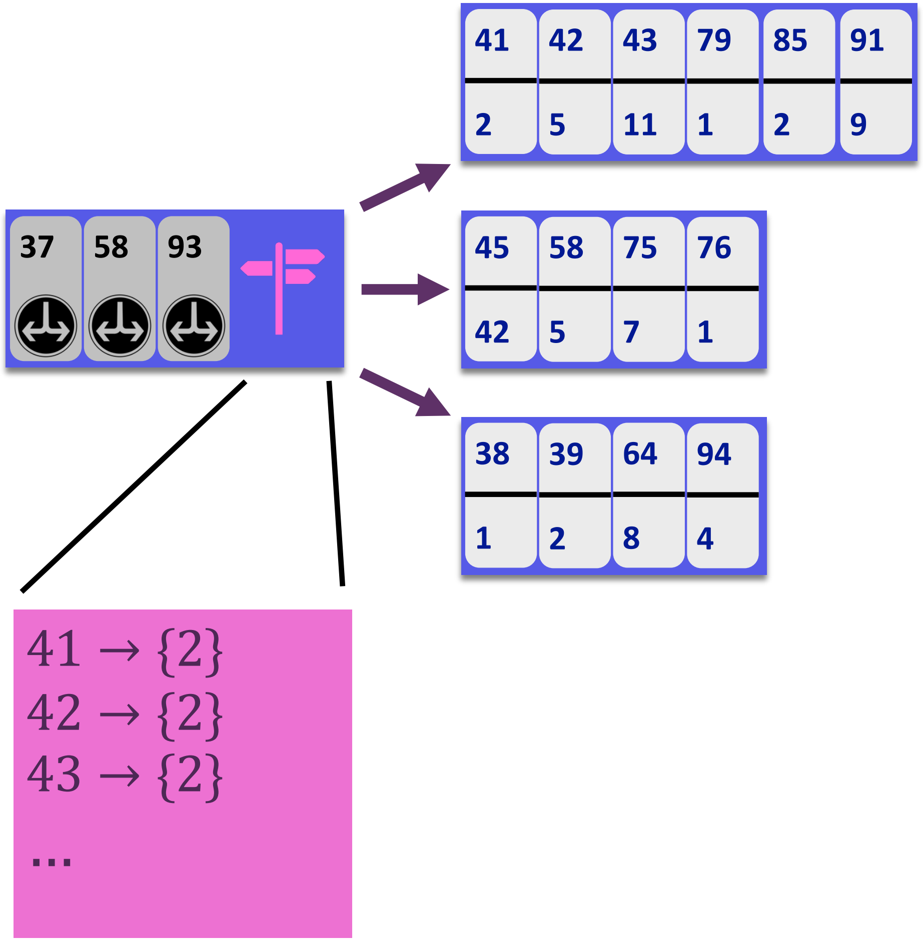
41	42	43	79	85	91
2	5	11	1	2	9

41	79	85	98
42	5	7	1

41	43	64	91
1	2	8	4

$41 \rightarrow \{0,1,2\}$
$42 \rightarrow \{1\}$
$43 \rightarrow \{0,2\}$
…

Lots of multiple entries



37	58	93

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

45	58	75	76
42	5	7	1

38	39	64	94
1	2	8	4

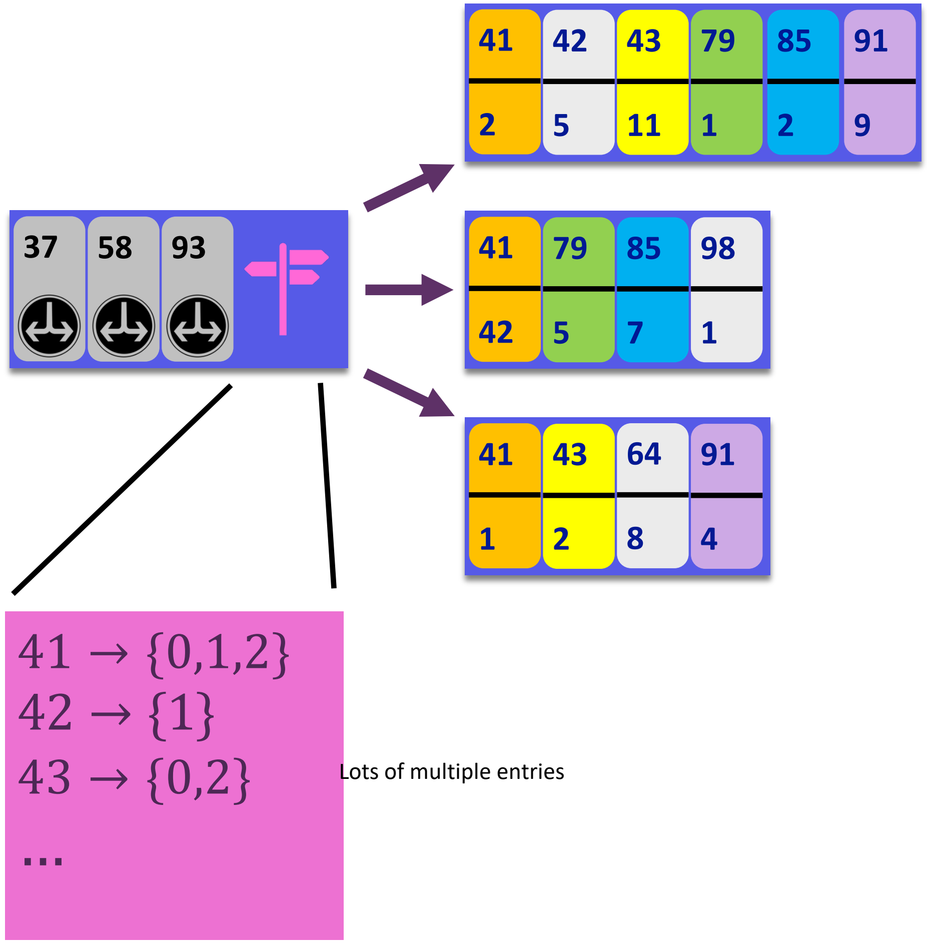
$41 \rightarrow \{2\}$
$42 \rightarrow \{2\}$
$43 \rightarrow \{2\}$
…

t

Maplet

Using Maplets to Manage Space

Maplets can tell us how much redundant data there is



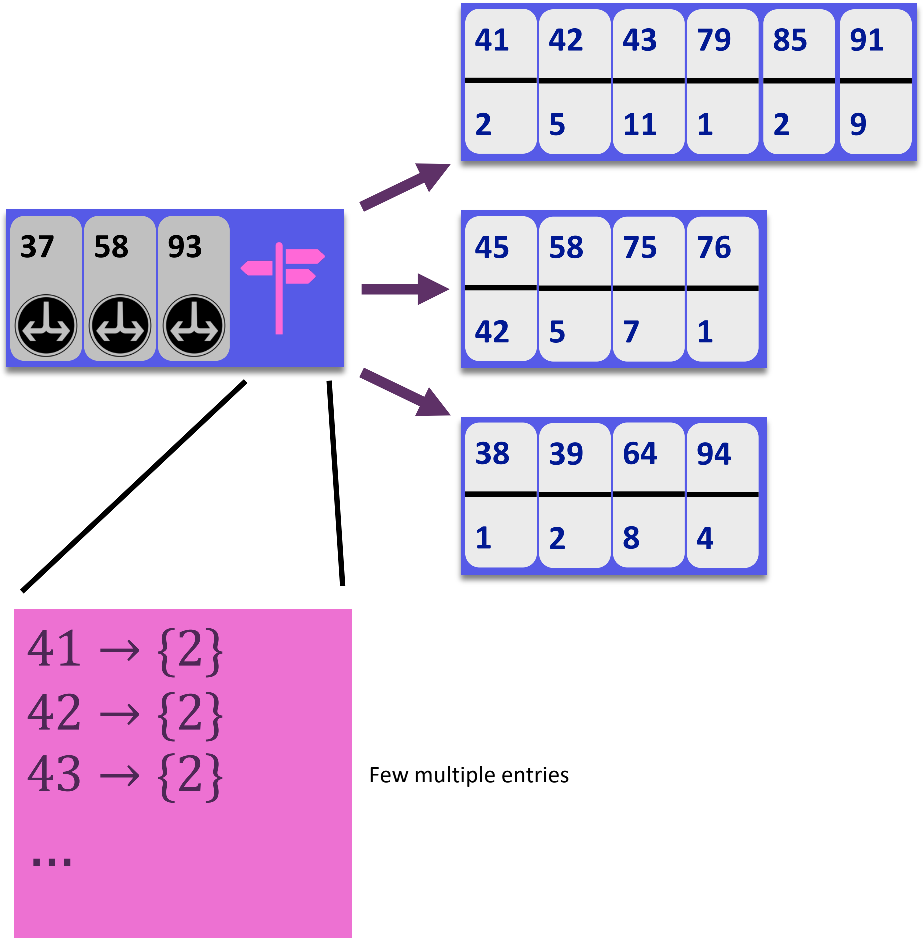
41	42	43	79	85	91
2	5	11	1	2	9

41	79	85	98
42	5	7	1

41	43	64	91
1	2	8	4

$41 \rightarrow \{0,1,2\}$
$42 \rightarrow \{1\}$
$43 \rightarrow \{0,2\}$
…

Lots of multiple entries



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

45	58	75	76
42	5	7	1

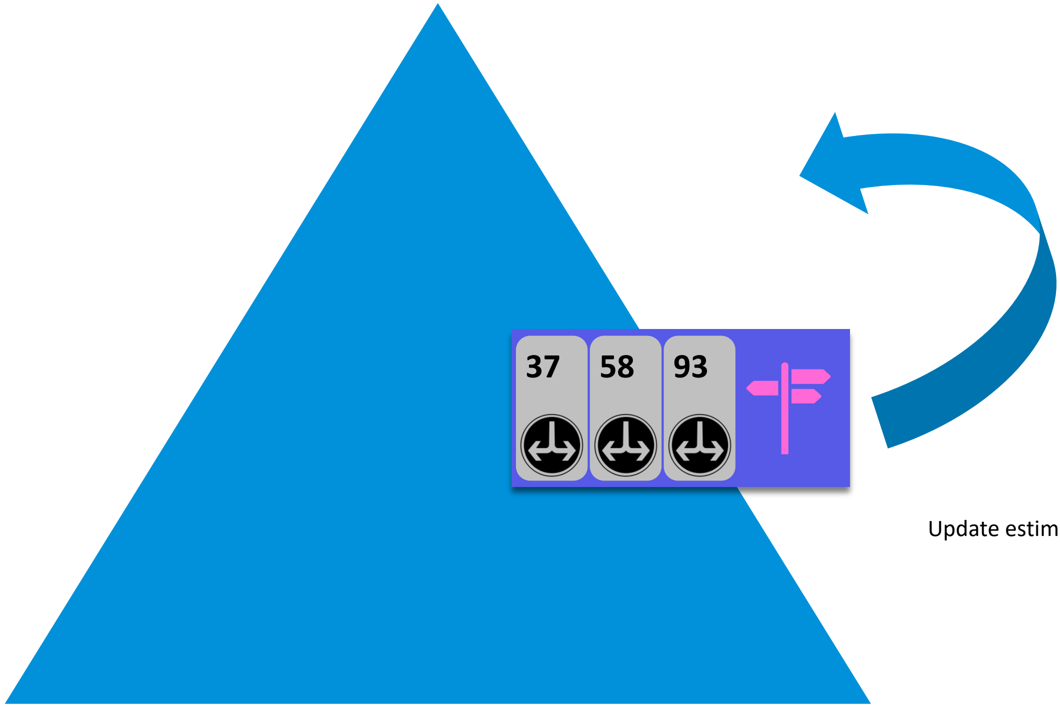
38	39	64	94
1	2	8	4

$41 \rightarrow \{2\}$
$42 \rightarrow \{2\}$
$43 \rightarrow \{2\}$
…

Few multiple entries

SplinterDB Adaptive Space Reclamation

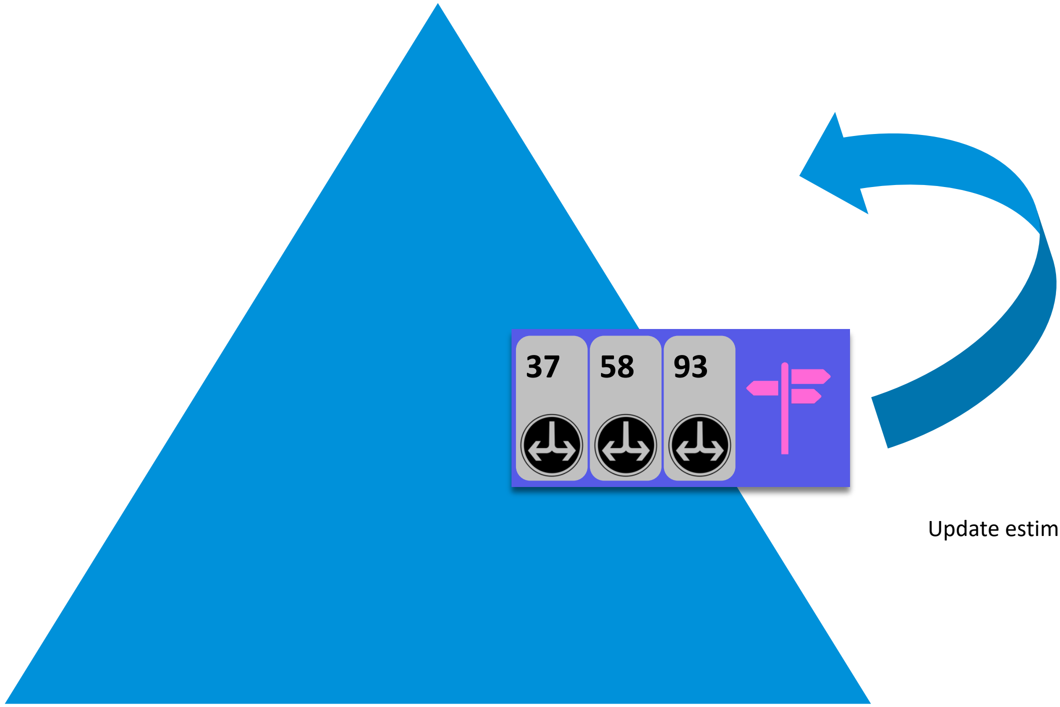
SplinterDB maintains a heap of trunk nodes, sorted by estimated amount of redundant data



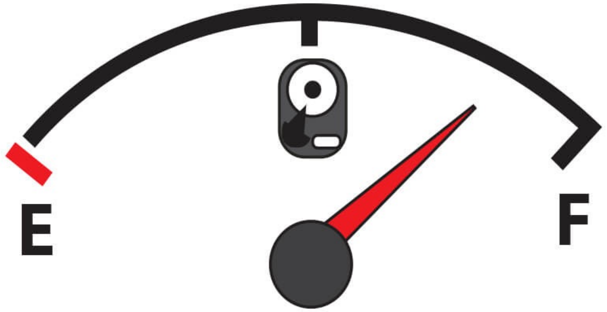
Update estimate every time we rebuild maplet

SplinterDB Adaptive Space Reclamation

SplinterDB maintains a heap of trunk nodes, sorted by estimated amount of redundant data



Update estimate every time we rebuild maplet



Whenever disk usage gets too high, SplinterDB initiates compaction on top node of the heap.

Goal: maximal gains, minimal pains

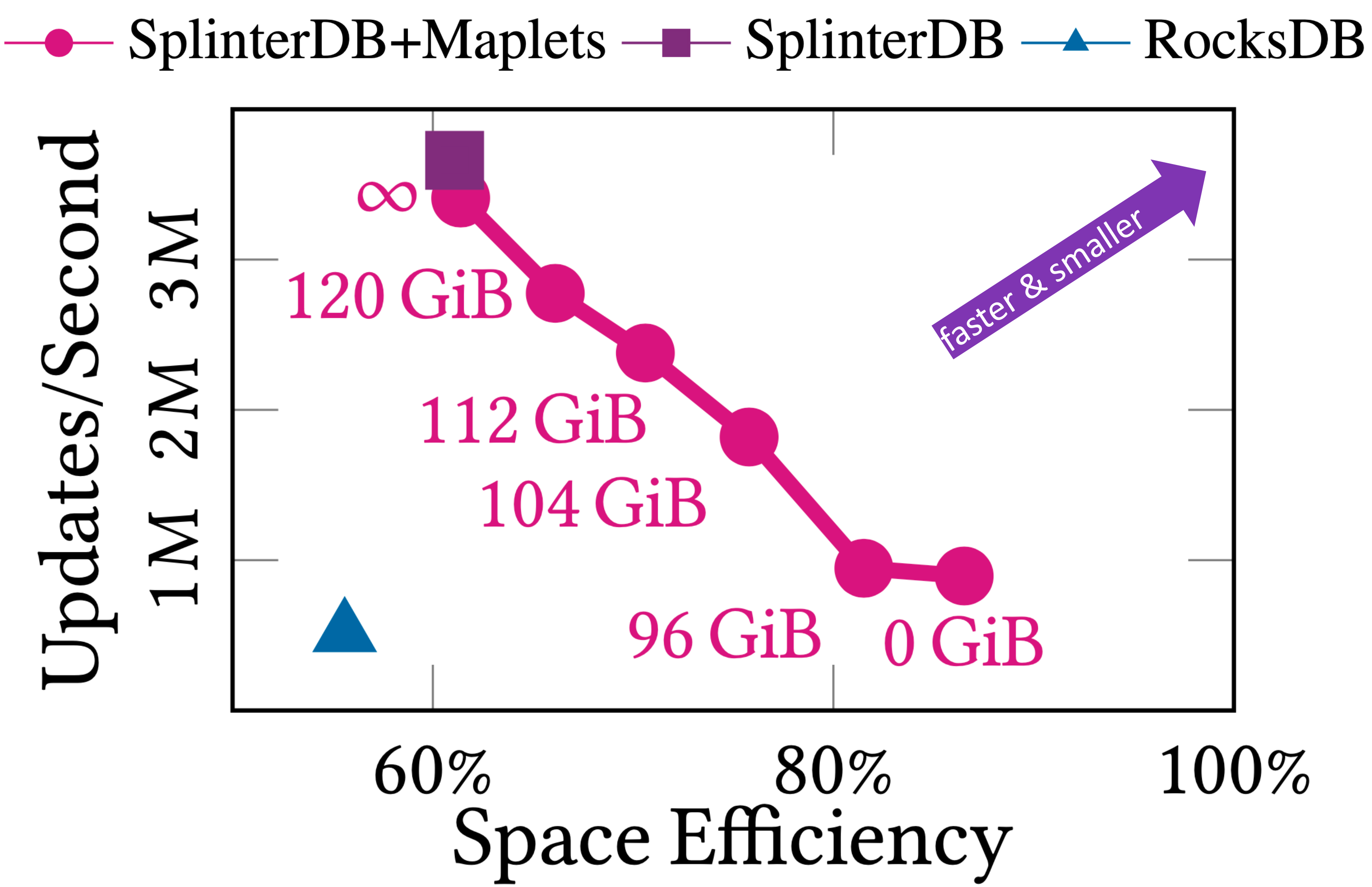
SplinterDB

28 2Ghz cores

Intel Optane 905P

24B keys 100B values

25GiB RAM
80GiB dataset

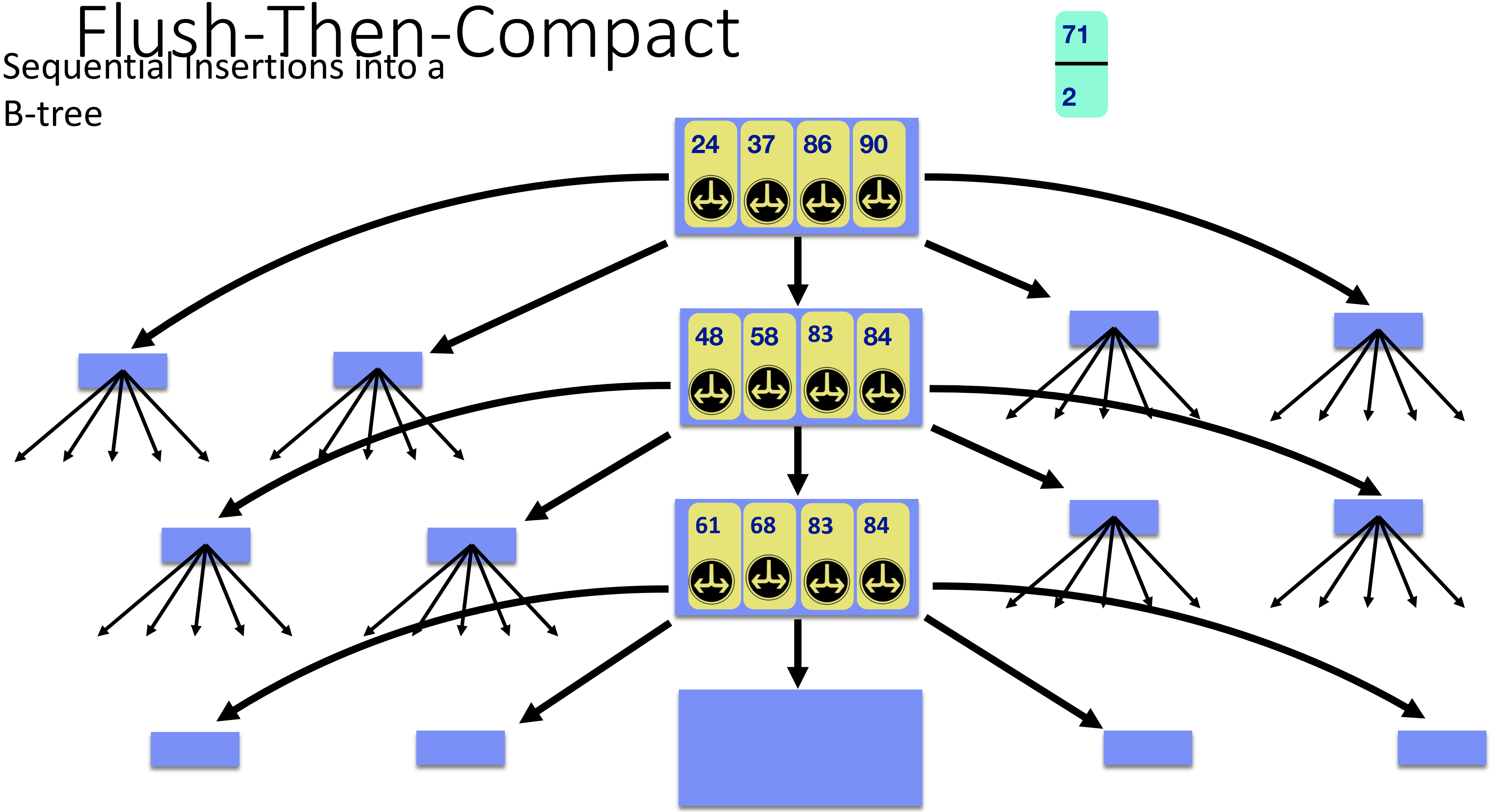


100% uniform updates

Flush-Then-Compact

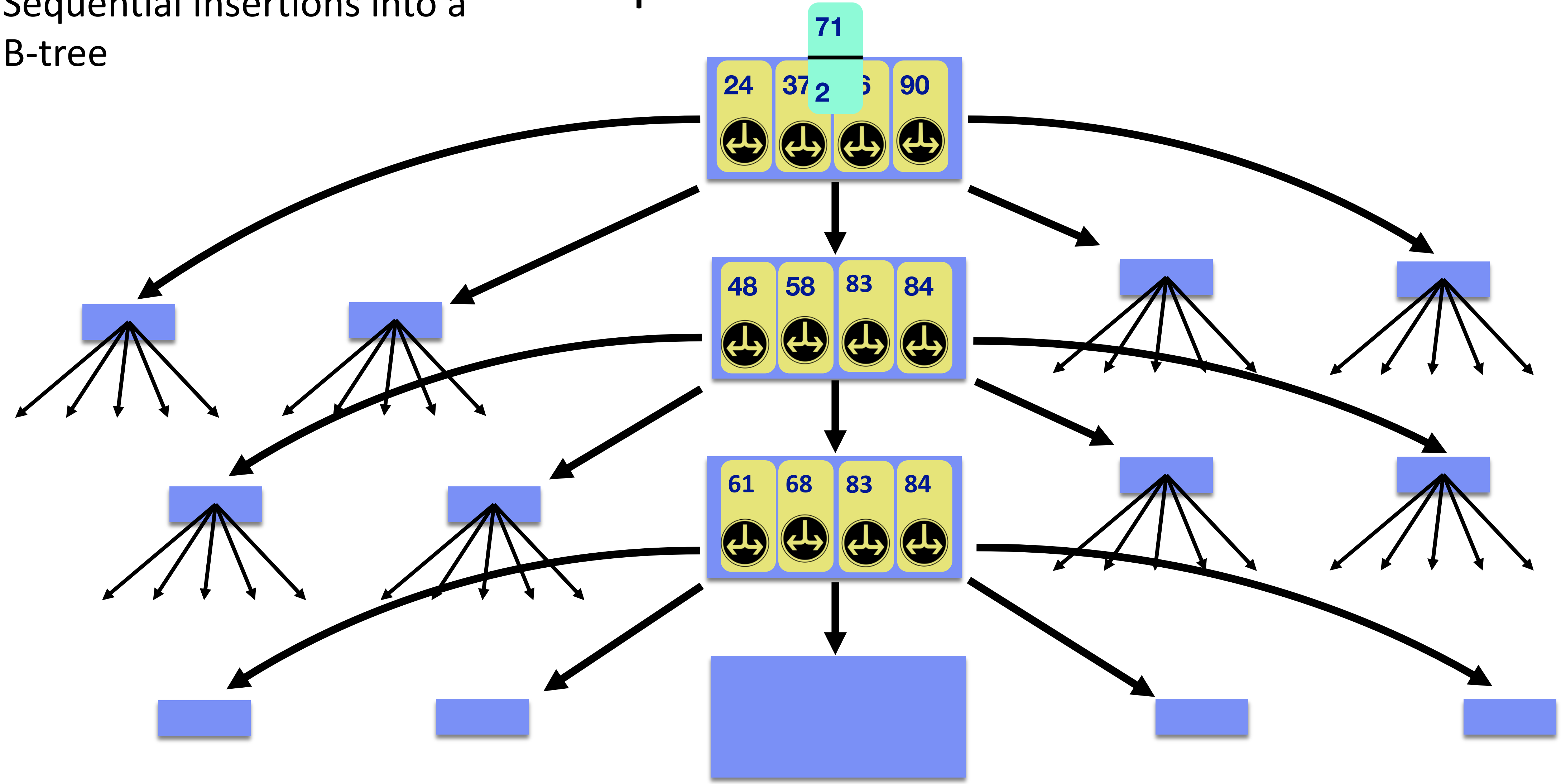
Flush-Then-Compact

Sequential Insertions into a B-tree



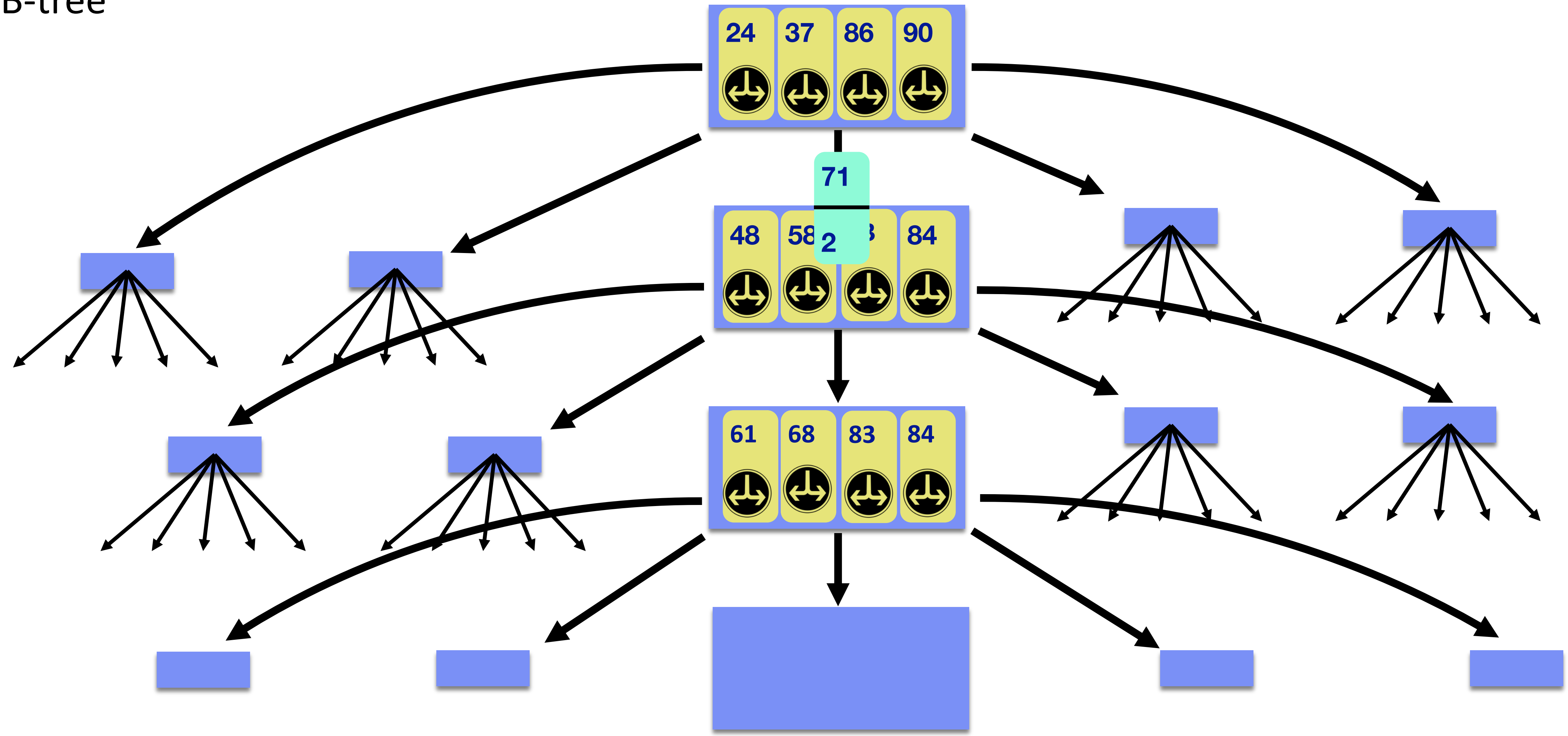
Flush-Then-Compact

Sequential Insertions into a B-tree



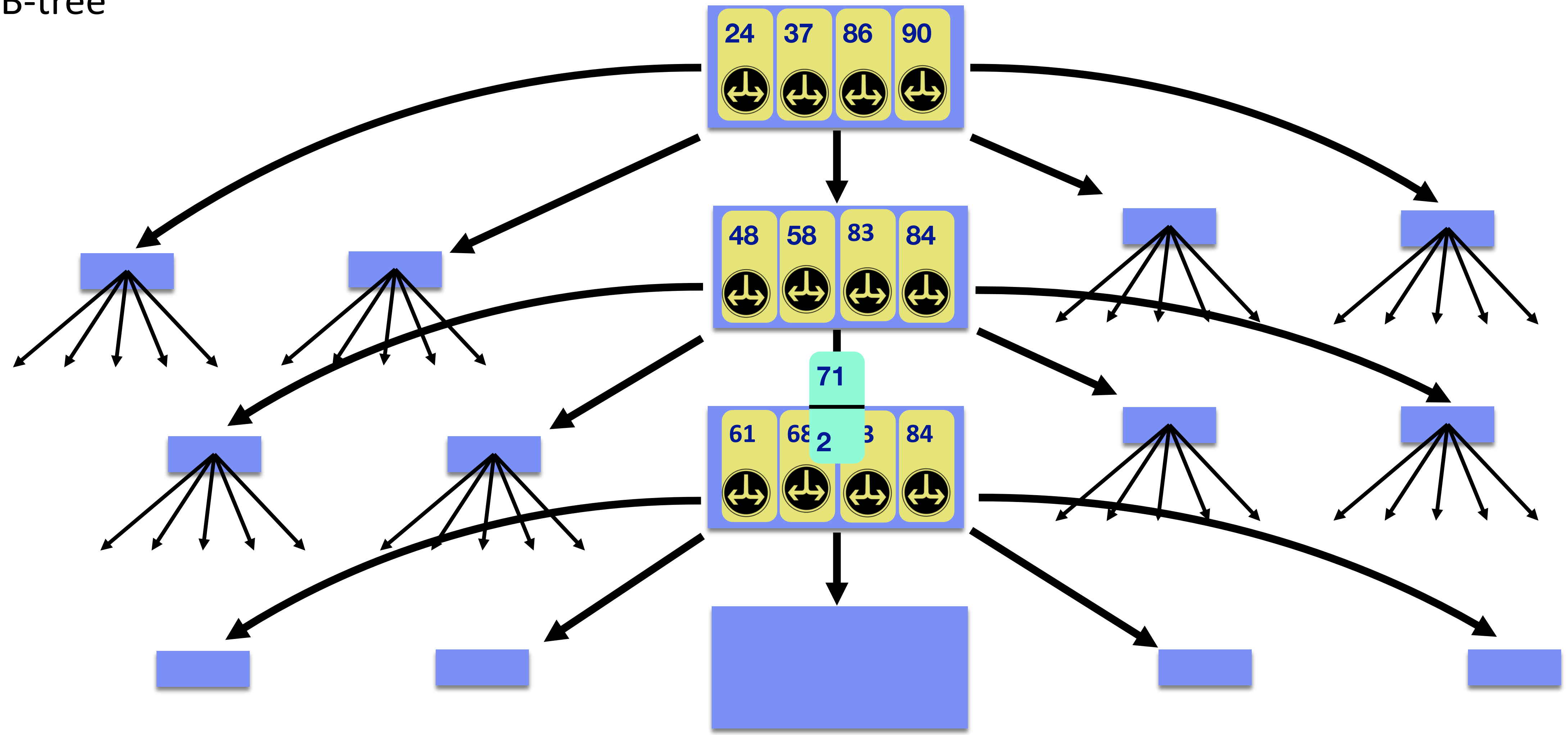
Flush-Then-Compact

Sequential Insertions into a B-tree



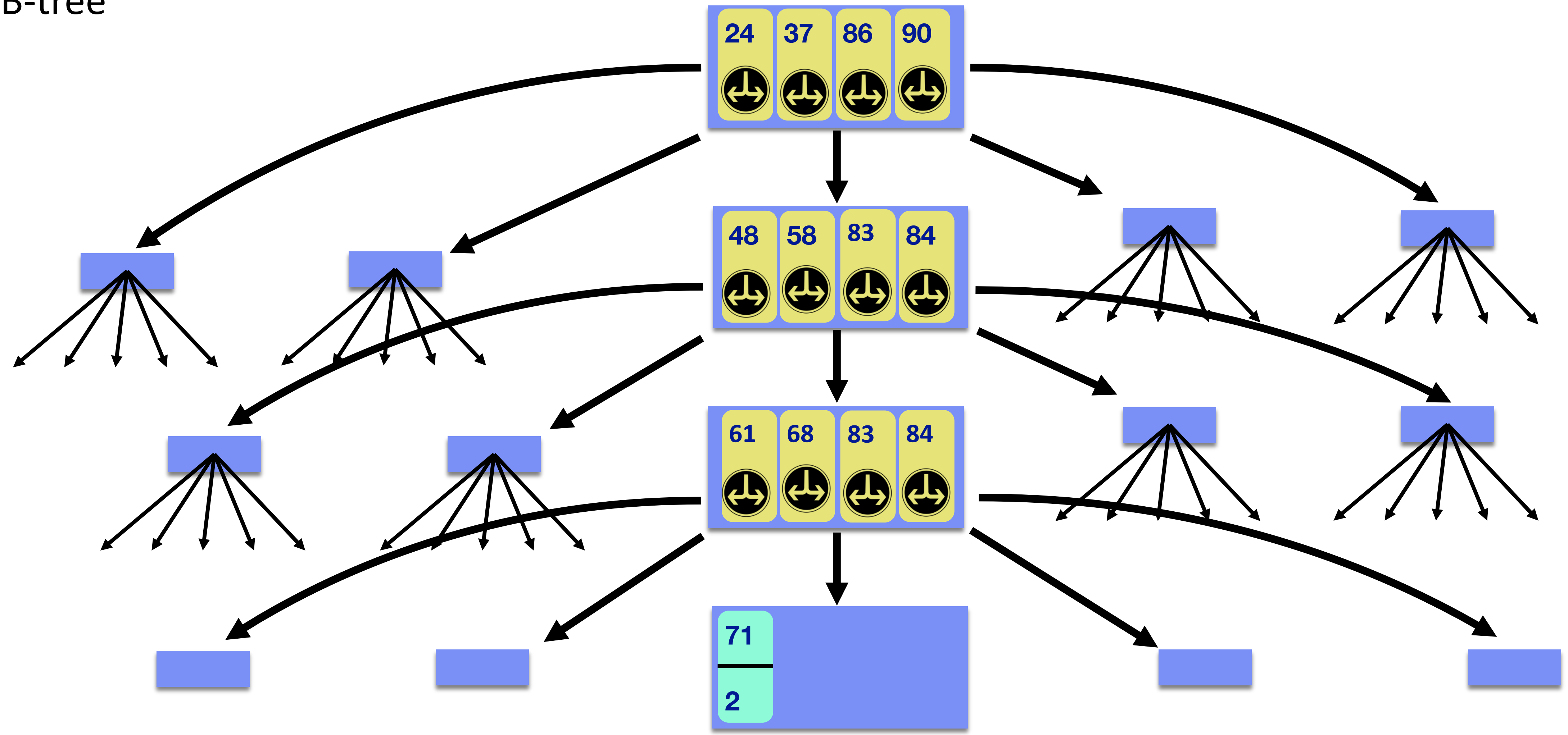
Flush-Then-Compact

Sequential Insertions into a B-tree



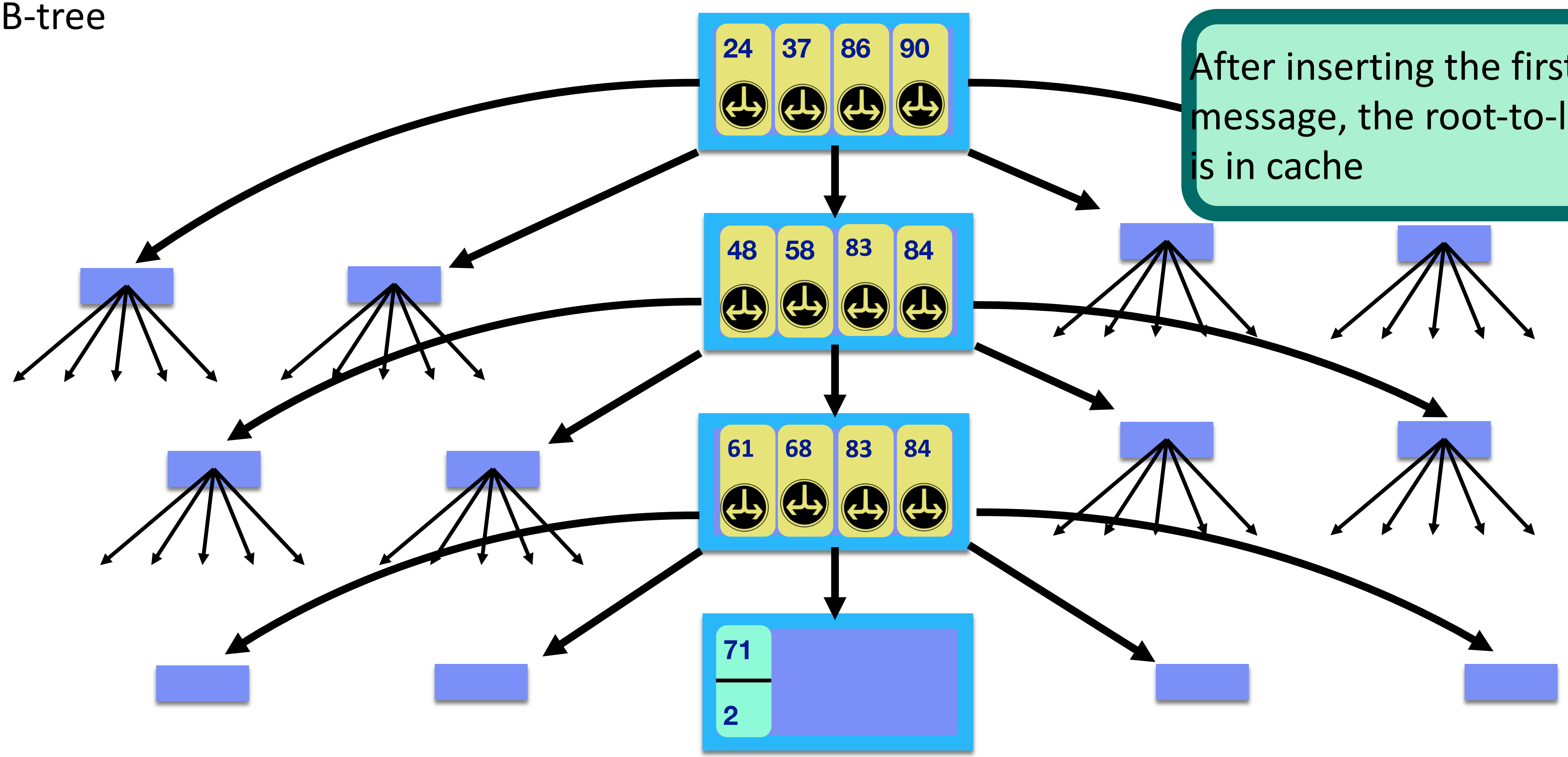
Flush-Then-Compact

Sequential Insertions into a B-tree



Flush-Then-Compact

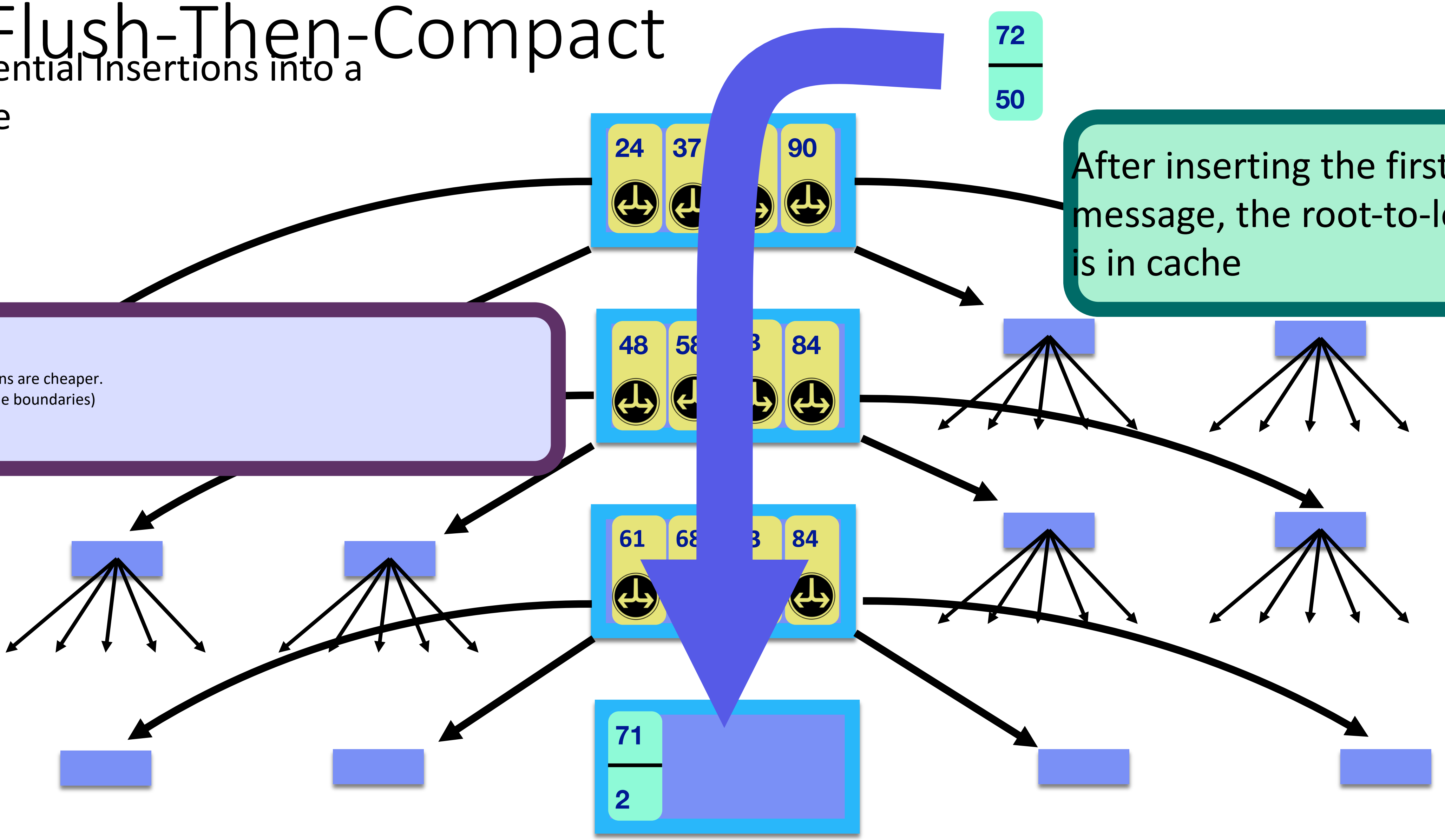
Sequential Insertions into a B-tree



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

Flush-Then-Compact

Sequential Insertions into a B-tree

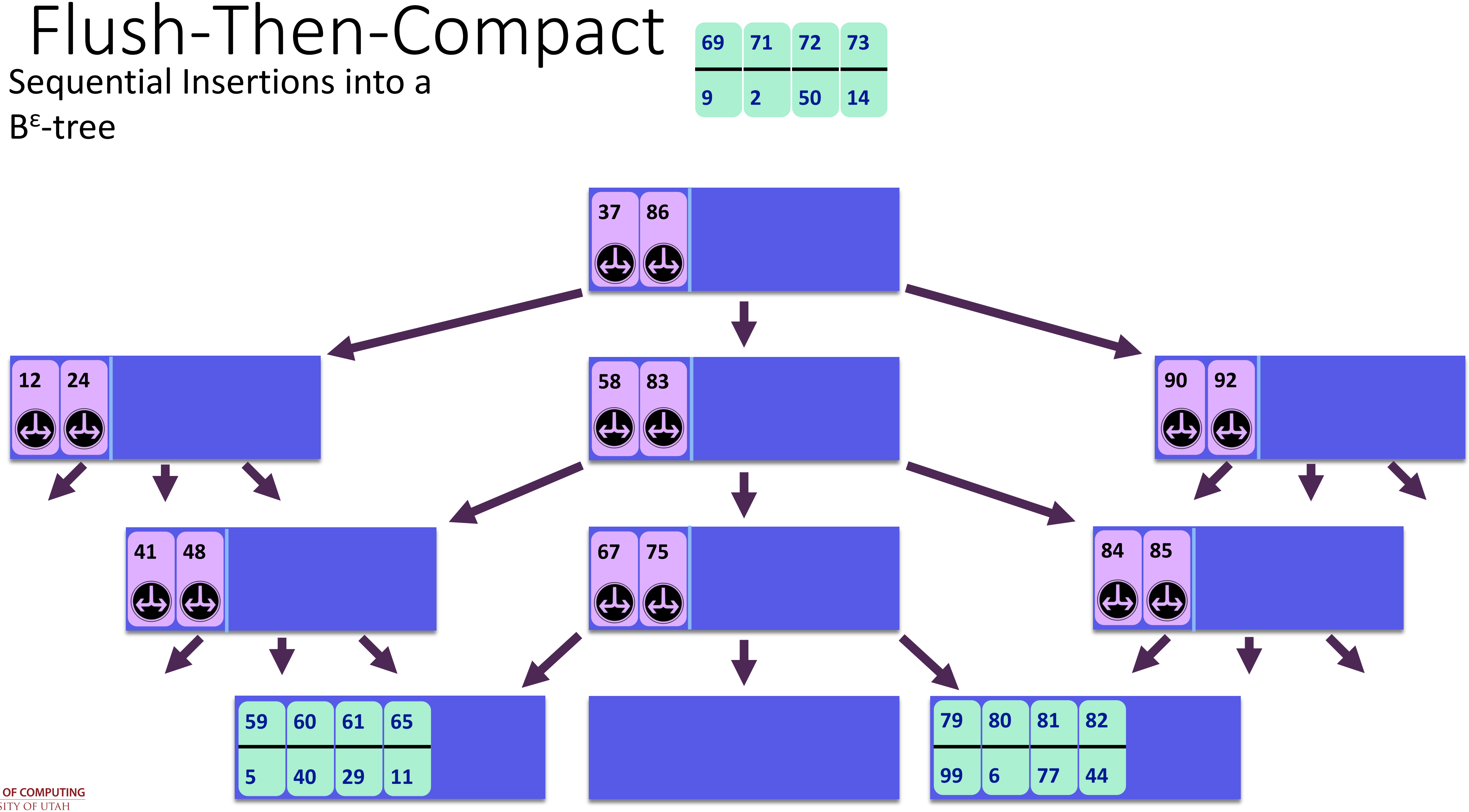


After inserting the first message, the root-to-leaf path is in cache

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

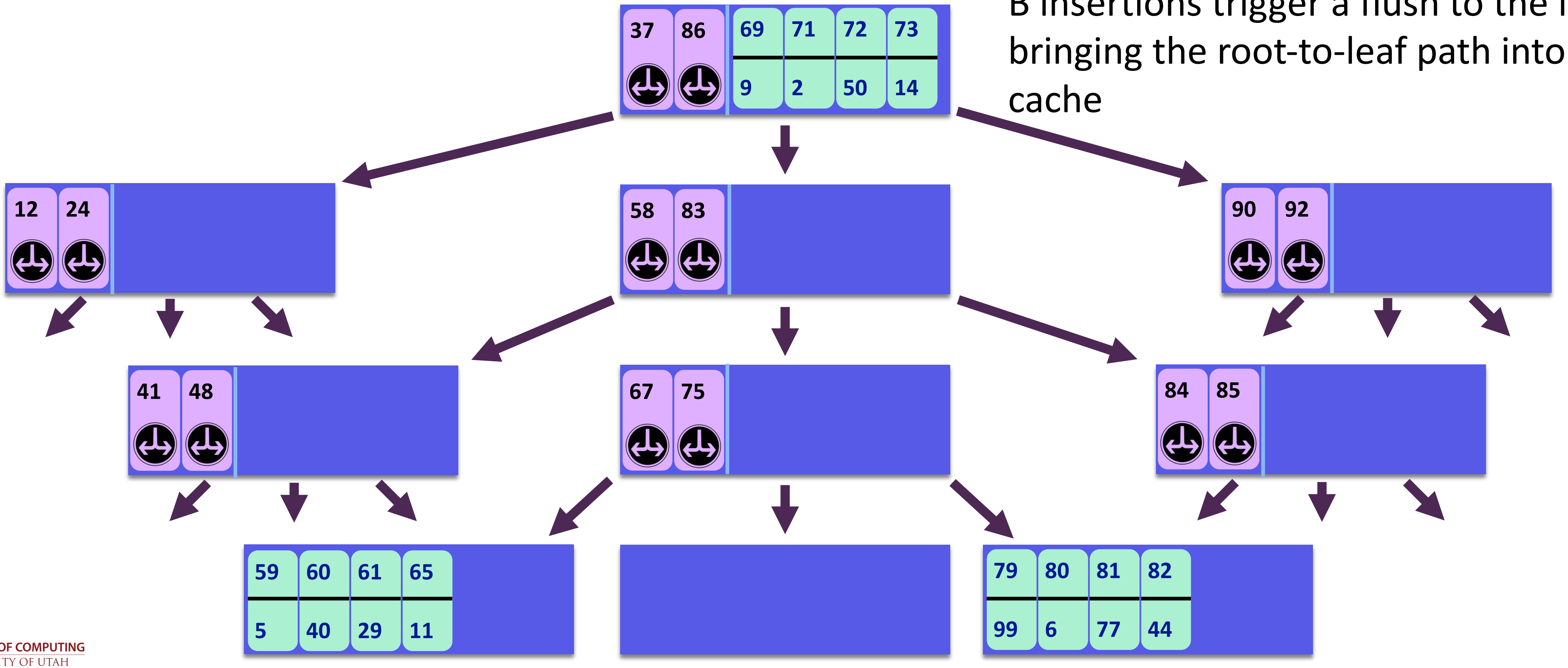
Flush-Then-Compact

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



Flush-Then-Compact

Sequential Insertions into a B^{ε}-tree

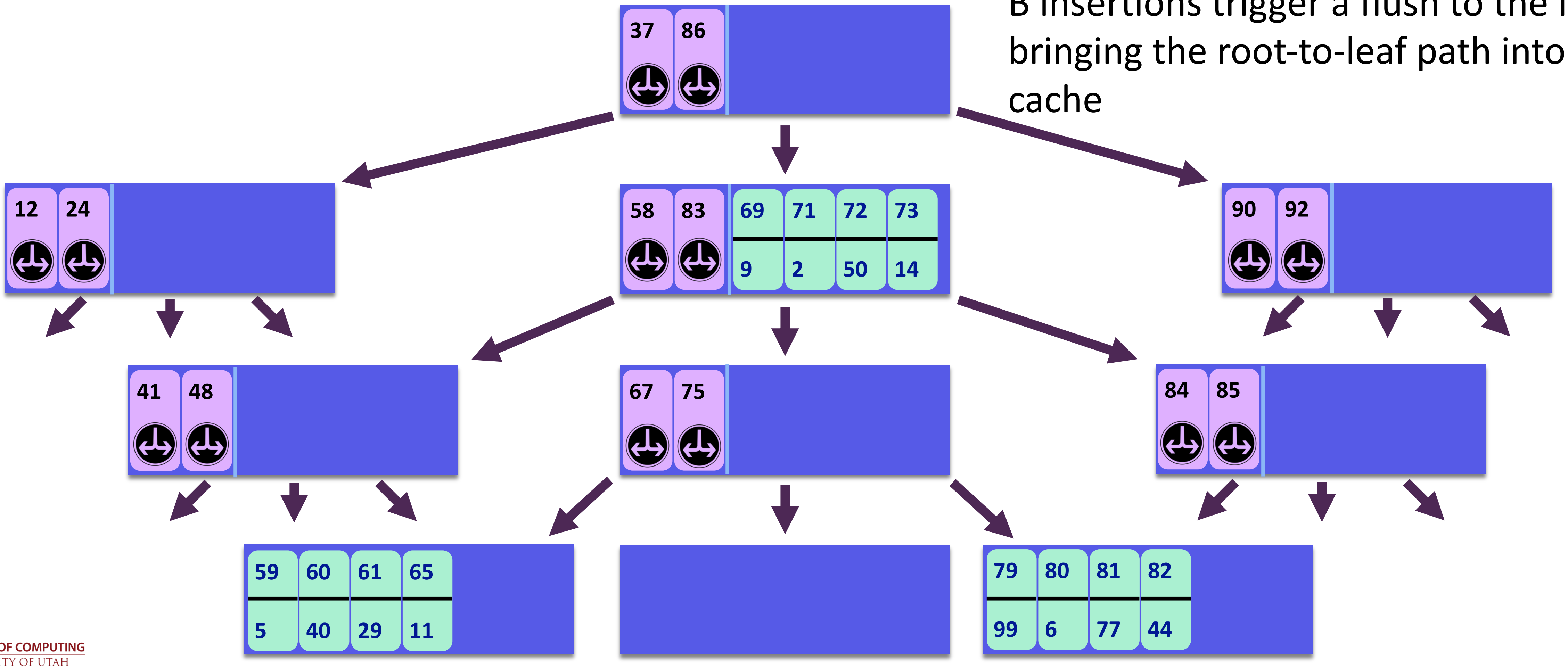


B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

Flush-Then-Compact

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

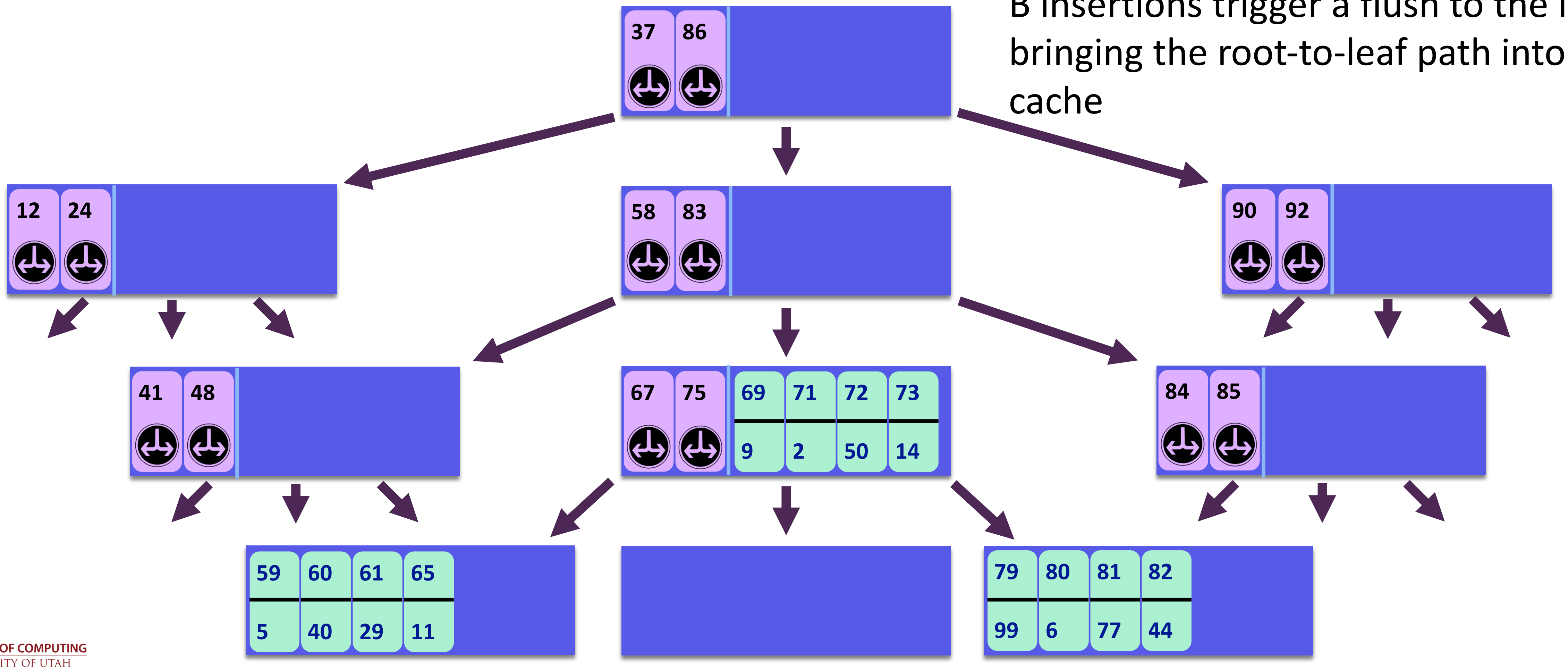
B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Flush-Then-Compact

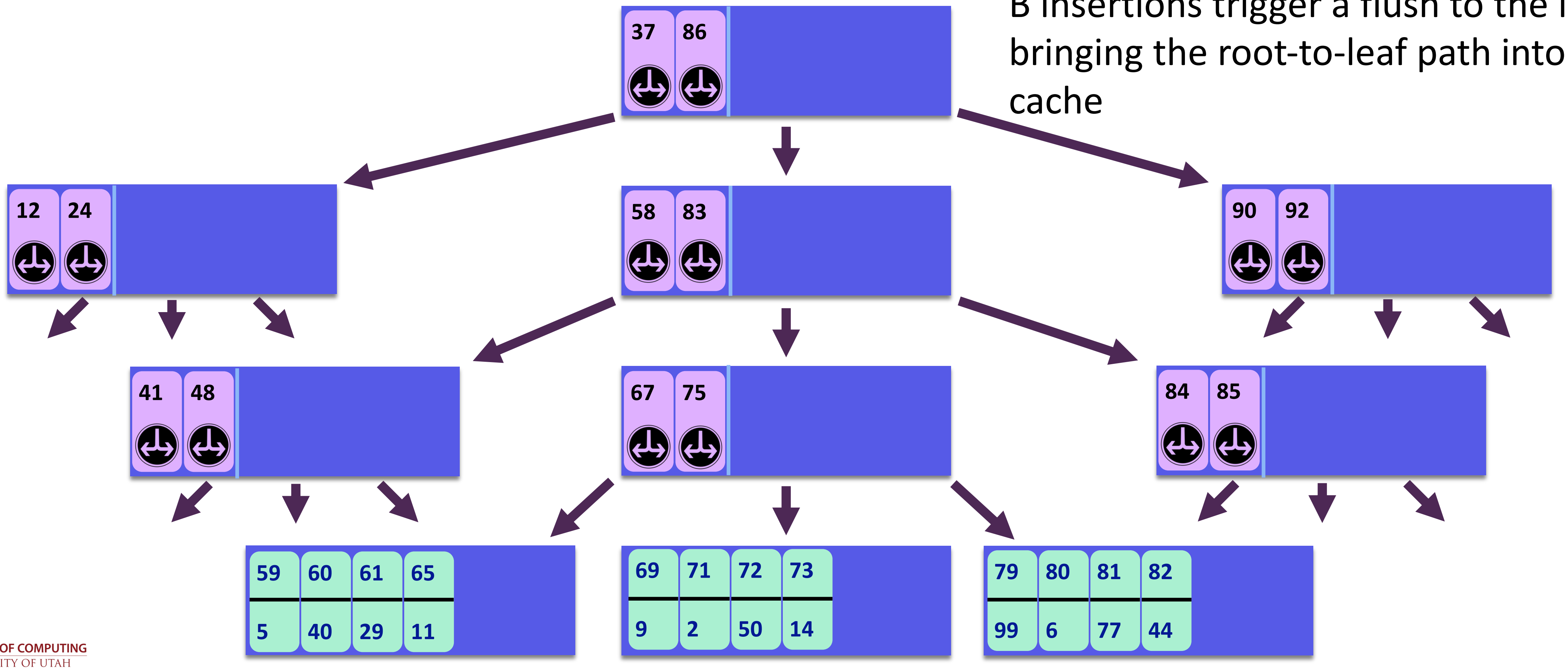
Sequential Insertions into a B^ε-tree



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

Flush-Then-Compact

Sequential Insertions into a Bε-tree

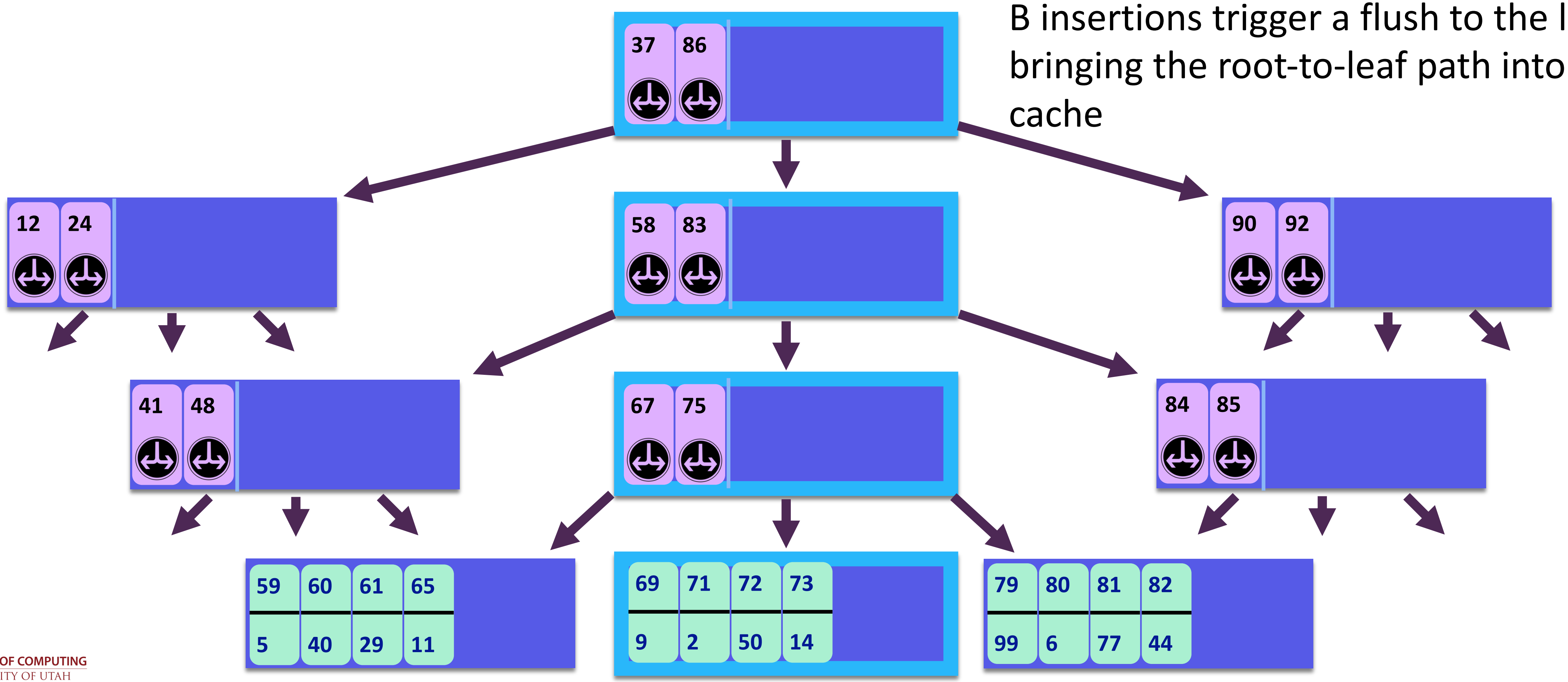


B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Flush-Then-Compact

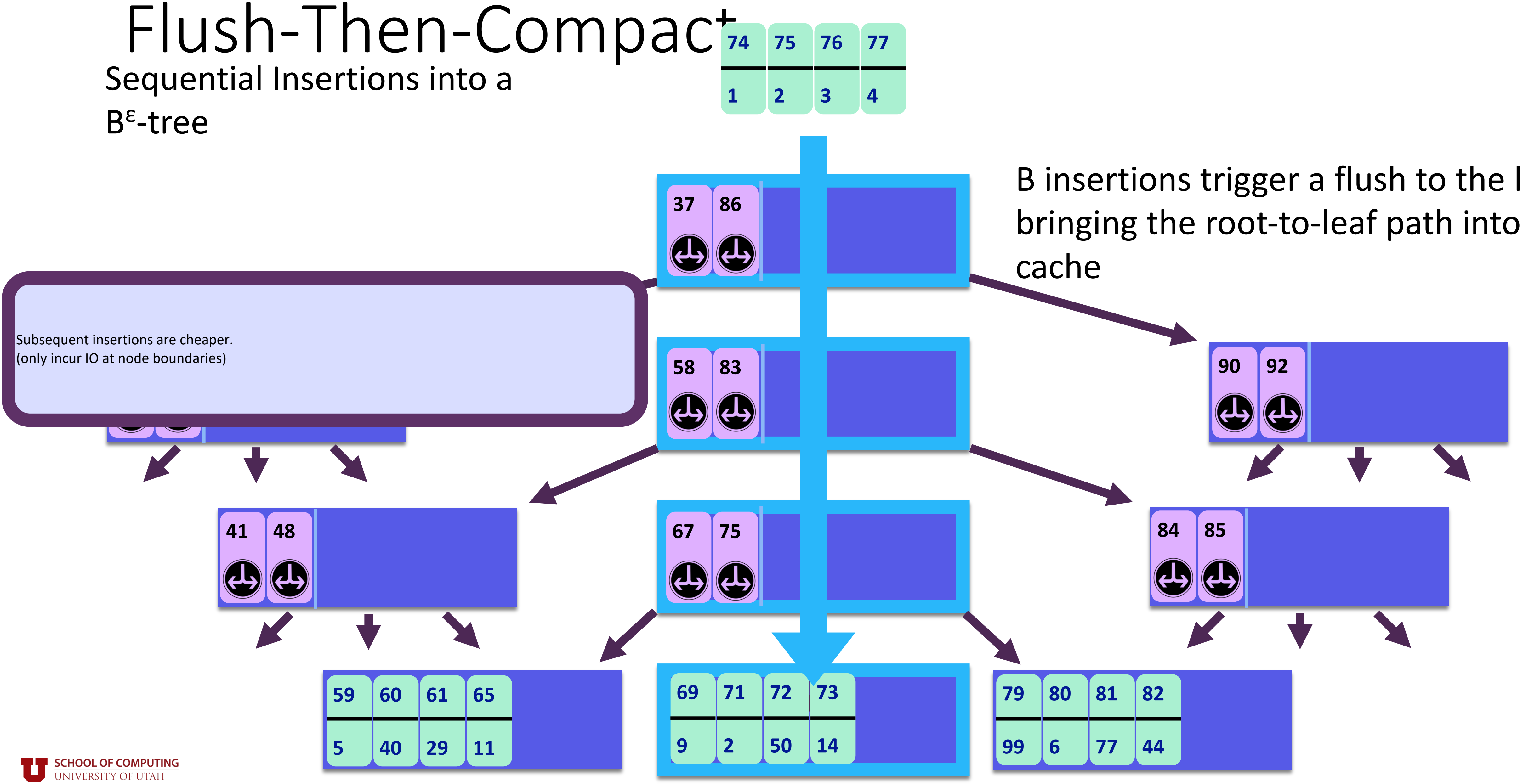
Sequential Insertions into a
B$^{\varepsilon}$-tree



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

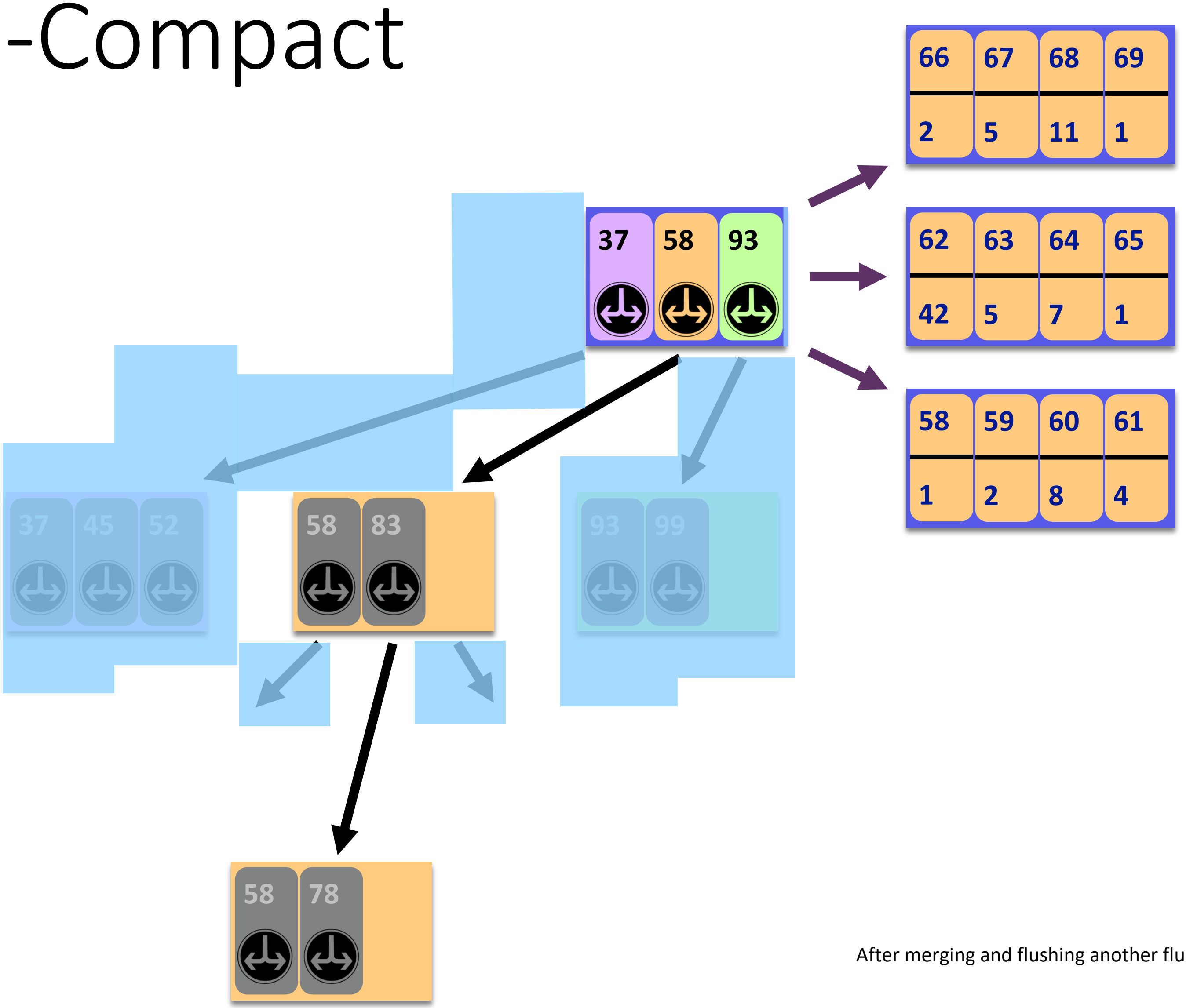


B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache

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

Flush-Then-Compact

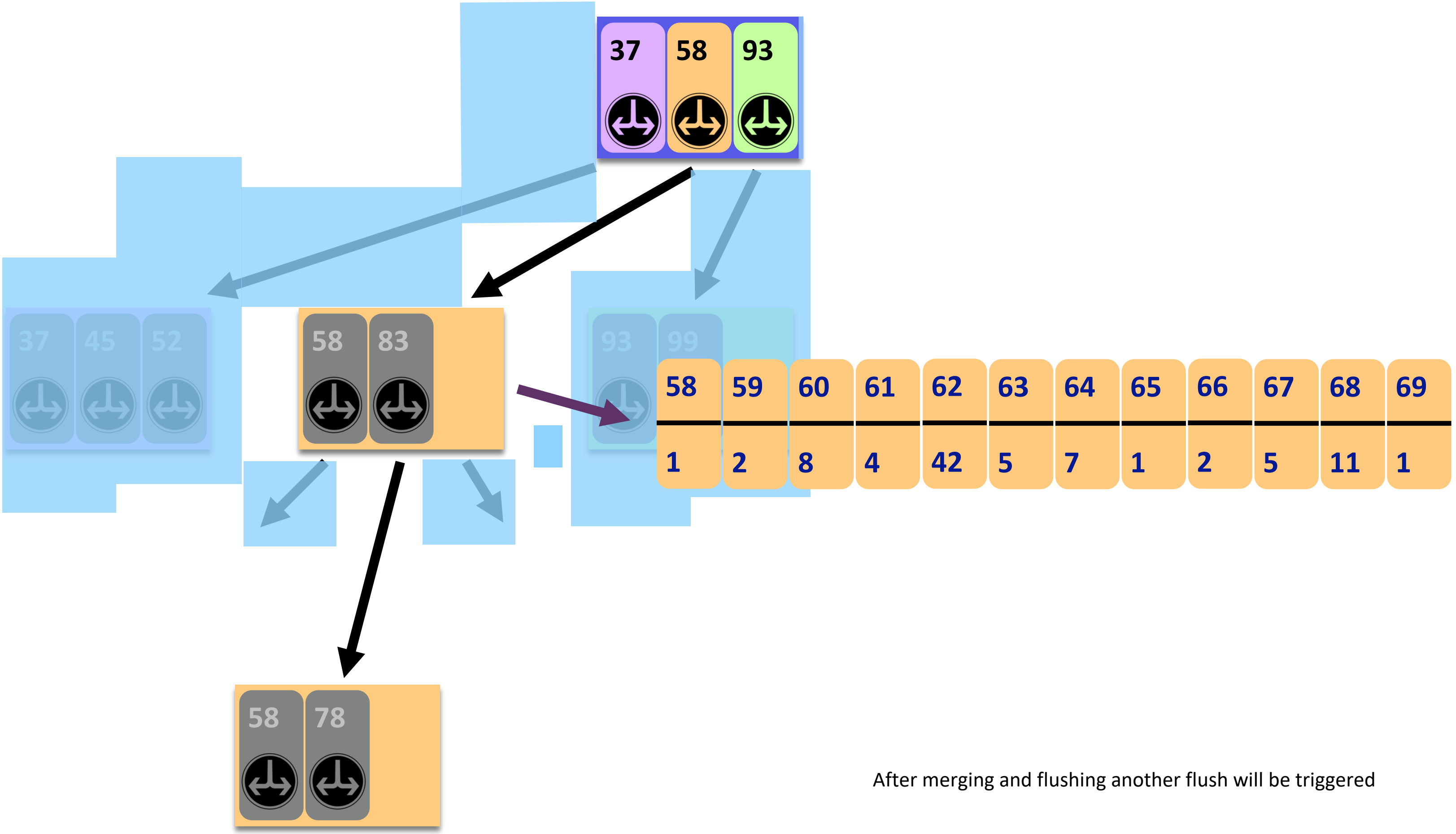
Want:
Cheap sequential insertions



After merging and flushing another flush will be triggered

Flush-Then-Compact

Want:
Cheap sequential insertions

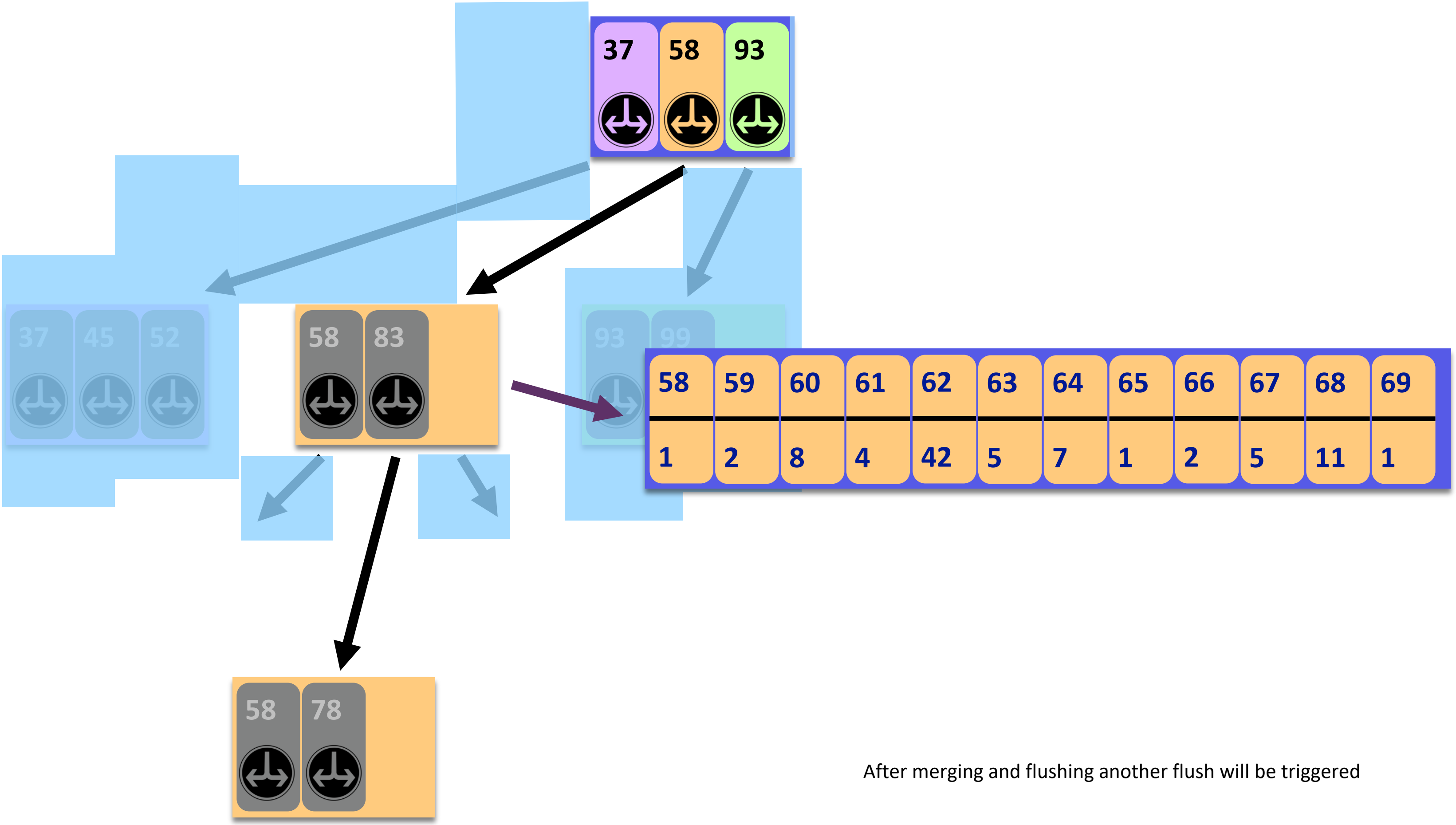


58	59	60	61	62	63	64	65	66	67	68	69
1	2	8	4	42	5	7	1	2	5	11	1

After merging and flushing another flush will be triggered

Flush-Then-Compact

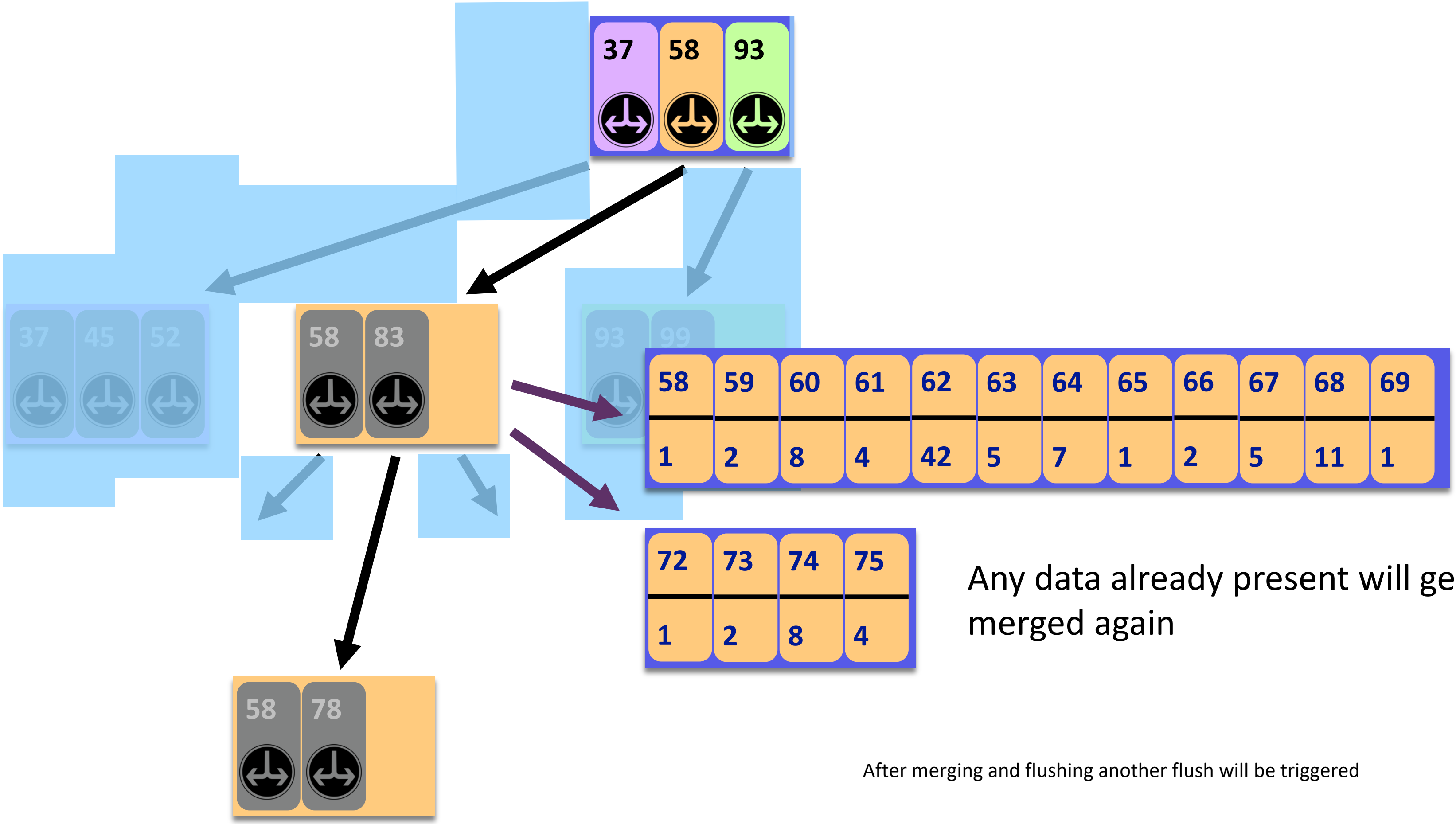
Want:
Cheap sequential insertions



After merging and flushing another flush will be triggered

Flush-Then-Compact

Want:
Cheap sequential insertions



58	59	60	61	62	63	64	65	66	67	68	69
1	2	8	4	42	5	7	1	2	5	11	1

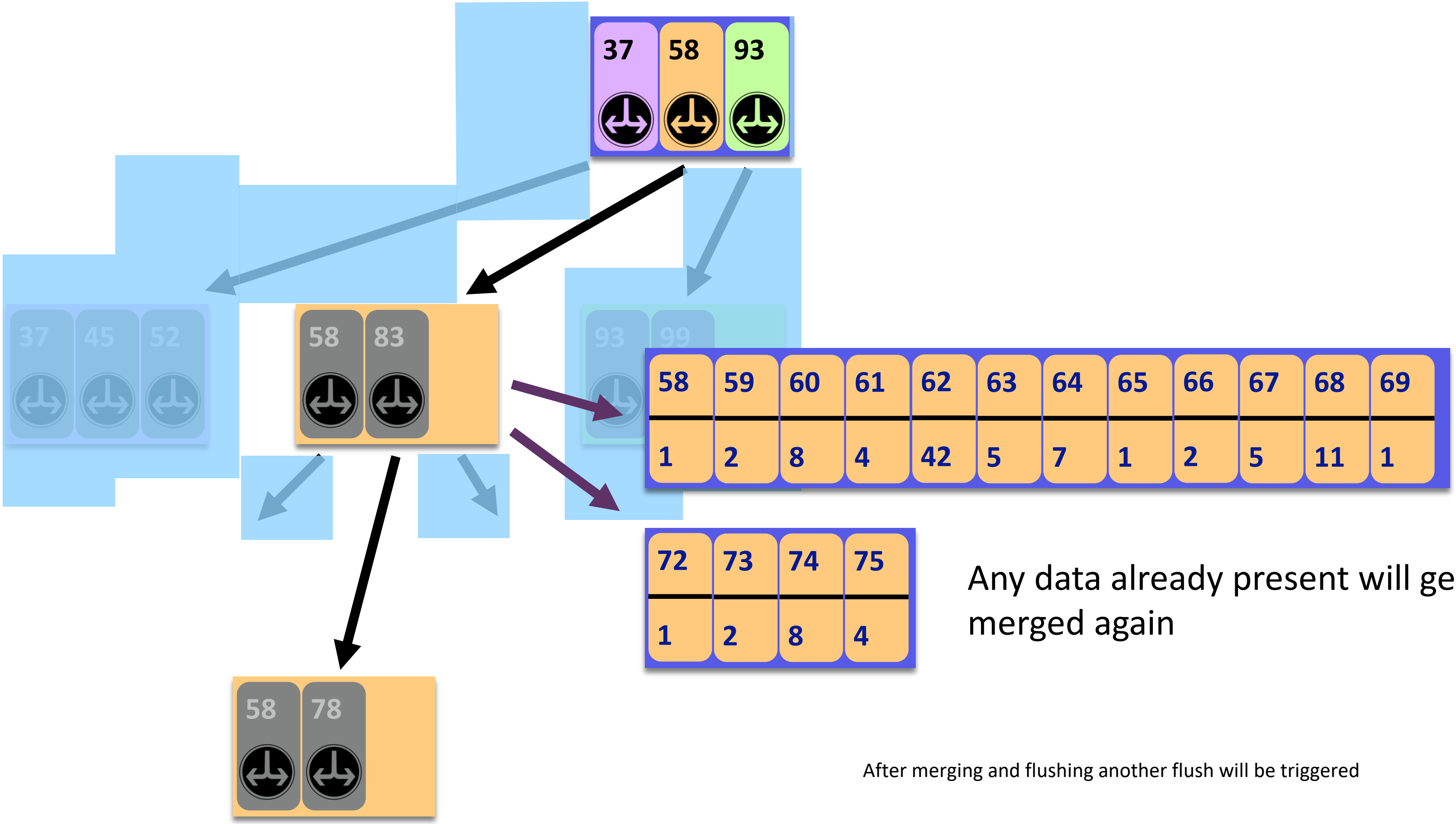
72	73	74	75
1	2	8	4

Any data already present will get merged again

After merging and flushing another flush will be triggered

Flush-Then-Compact

Want:
Cheap sequential insertions



58	59	60	61	62	63	64	65	66	67	68	69
1	2	8	4	42	5	7	1	2	5	11	1

72	73	74	75
1	2	8	4

Any data already present will get merged again

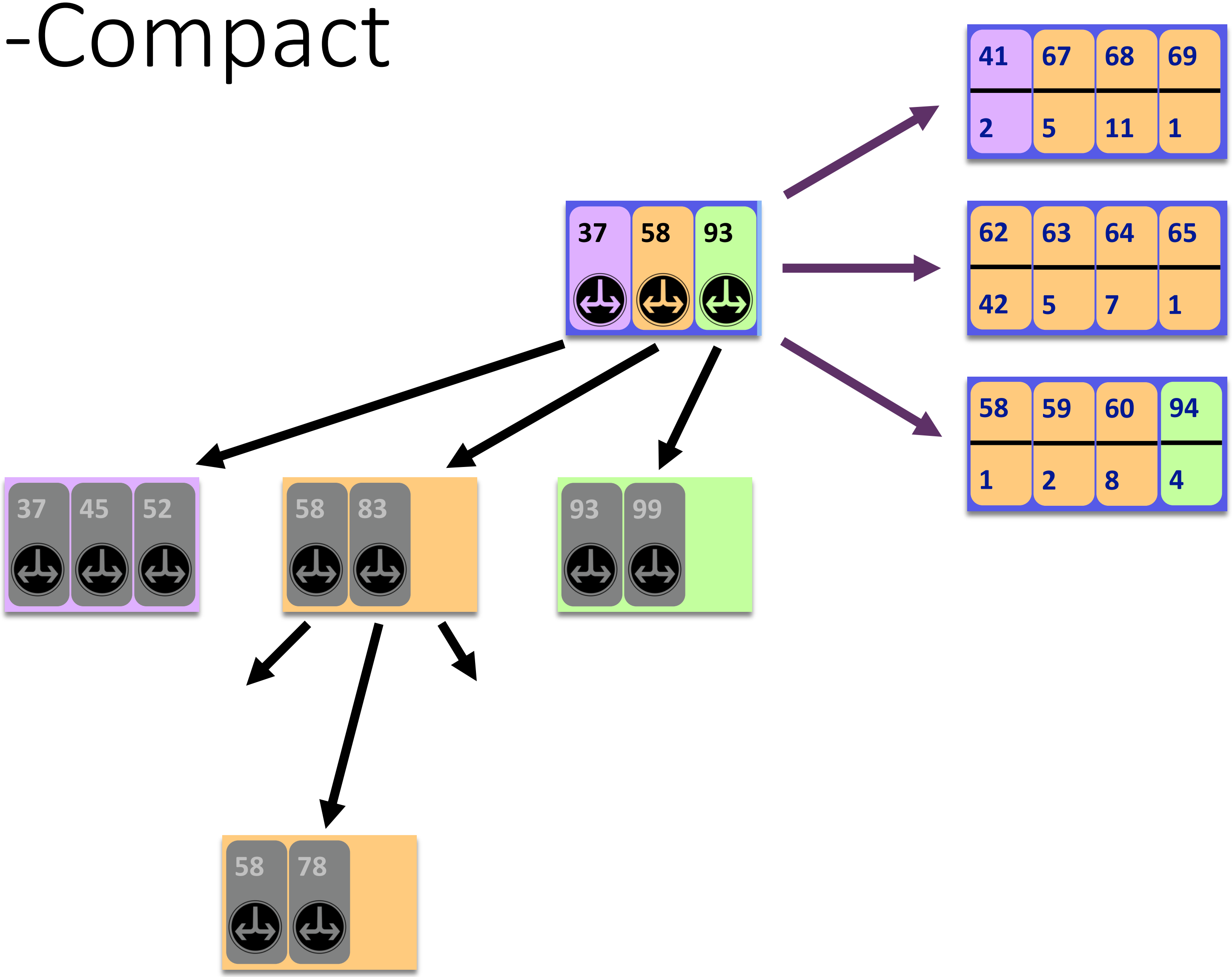
Can still end up merging on each level

After merging and flushing another flush will be triggered

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact

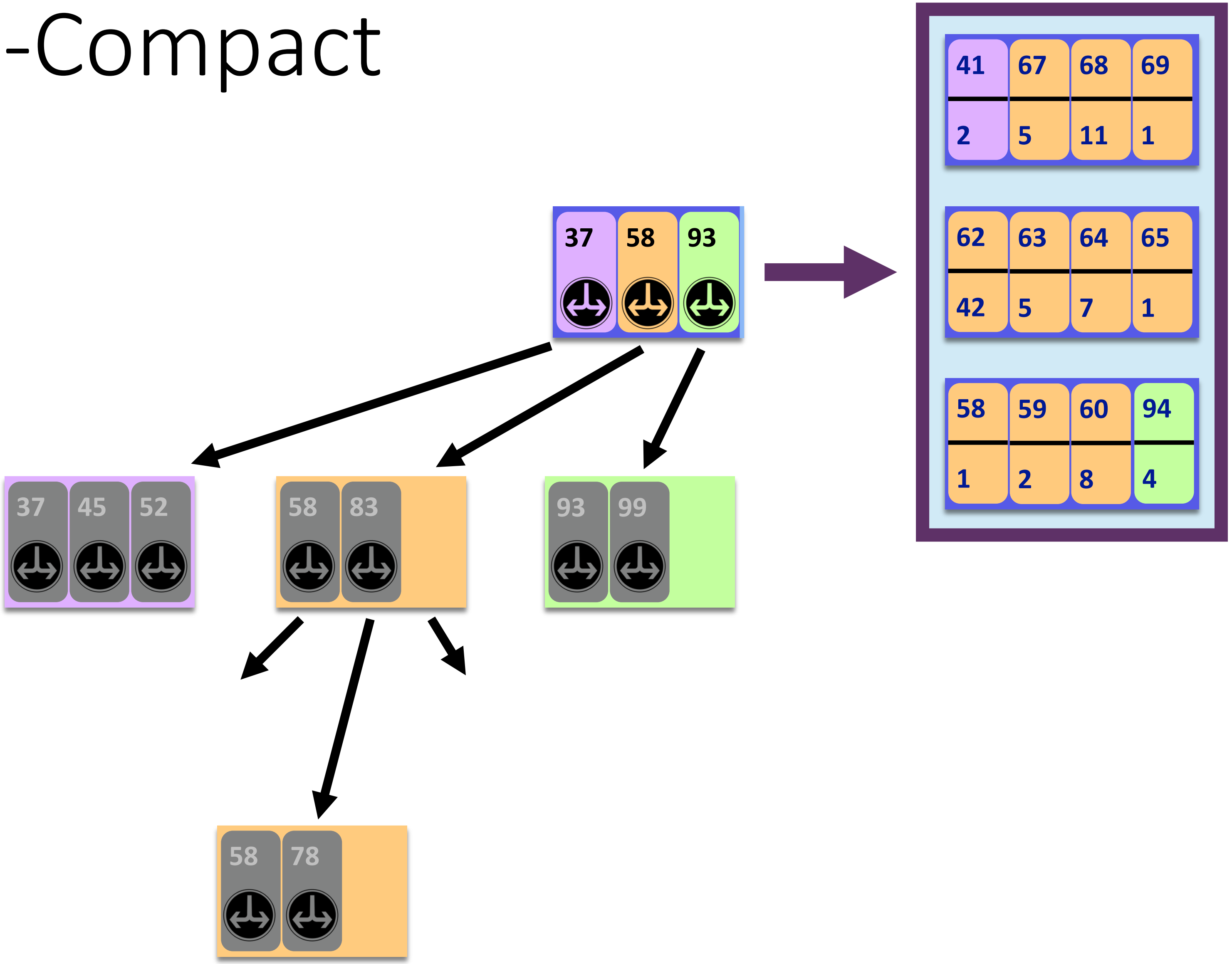


SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Flush-Then-Compact

Want:
Cheap sequential insertions

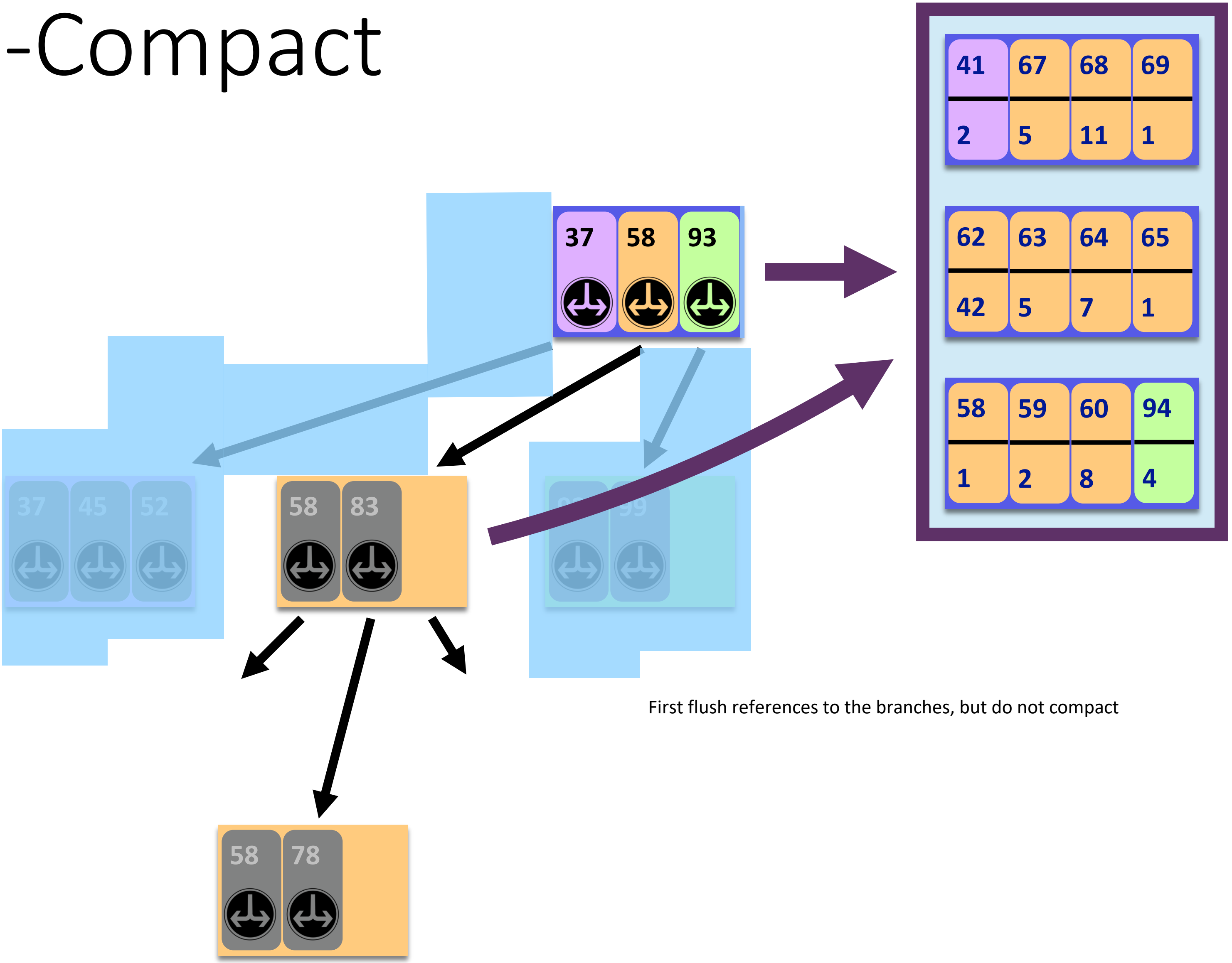
Idea: Flush-then-compact



Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact

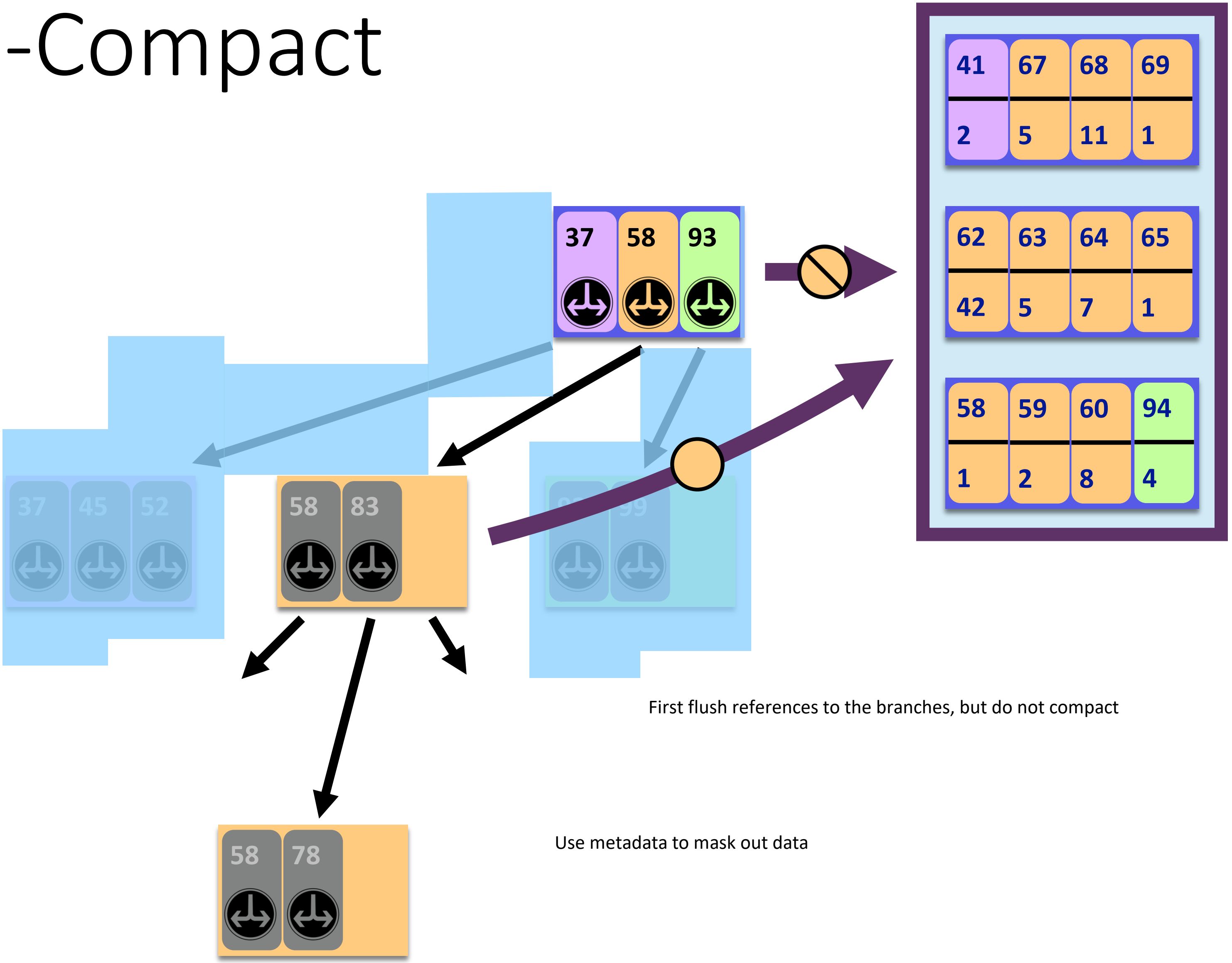


First flush references to the branches, but do not compact

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact



First flush references to the branches, but do not compact

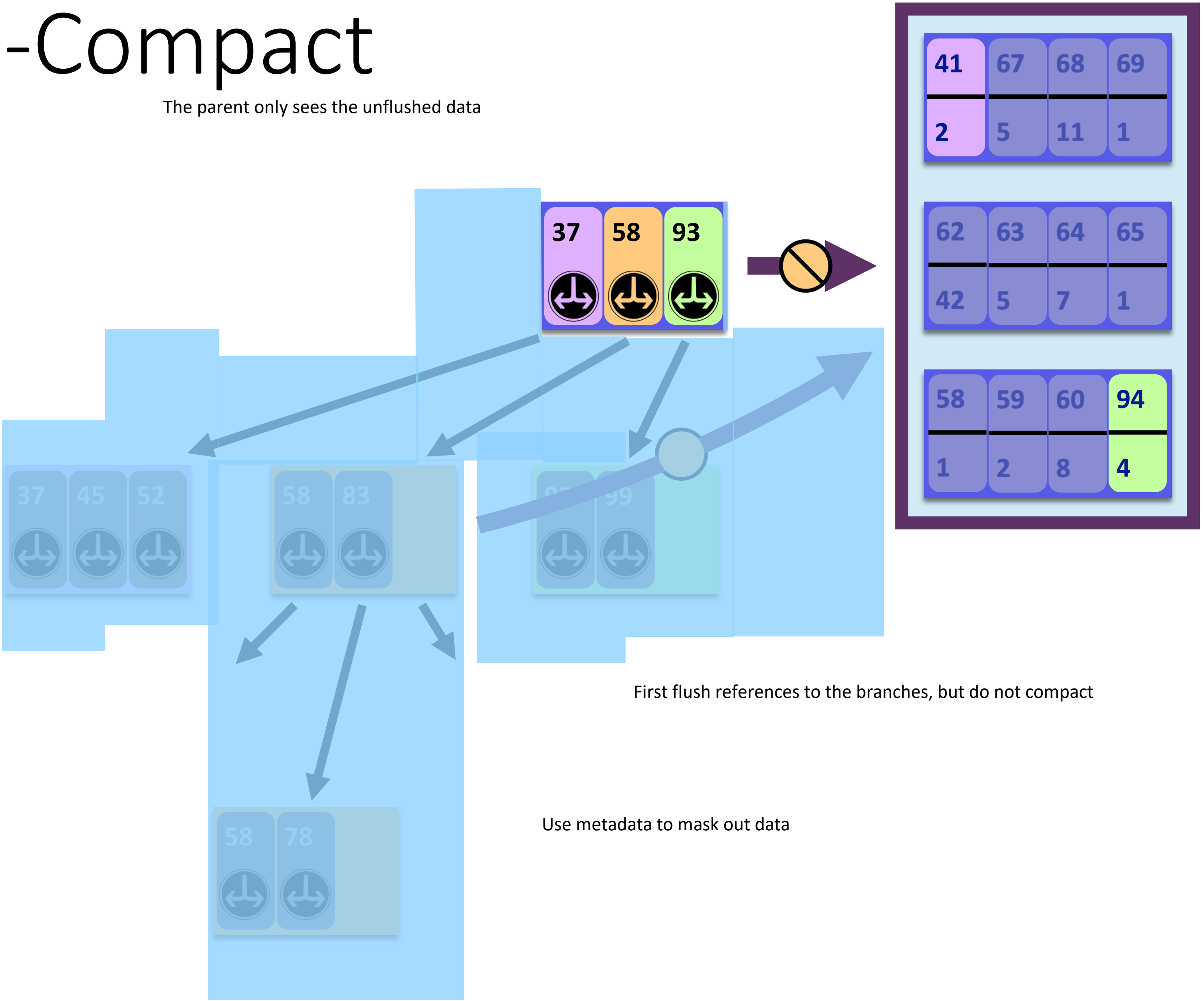
Use metadata to mask out data

Flush-Then-Compact

The parent only sees the unflushed data

Want:
Cheap sequential insertions

Idea: Flush-then-compact



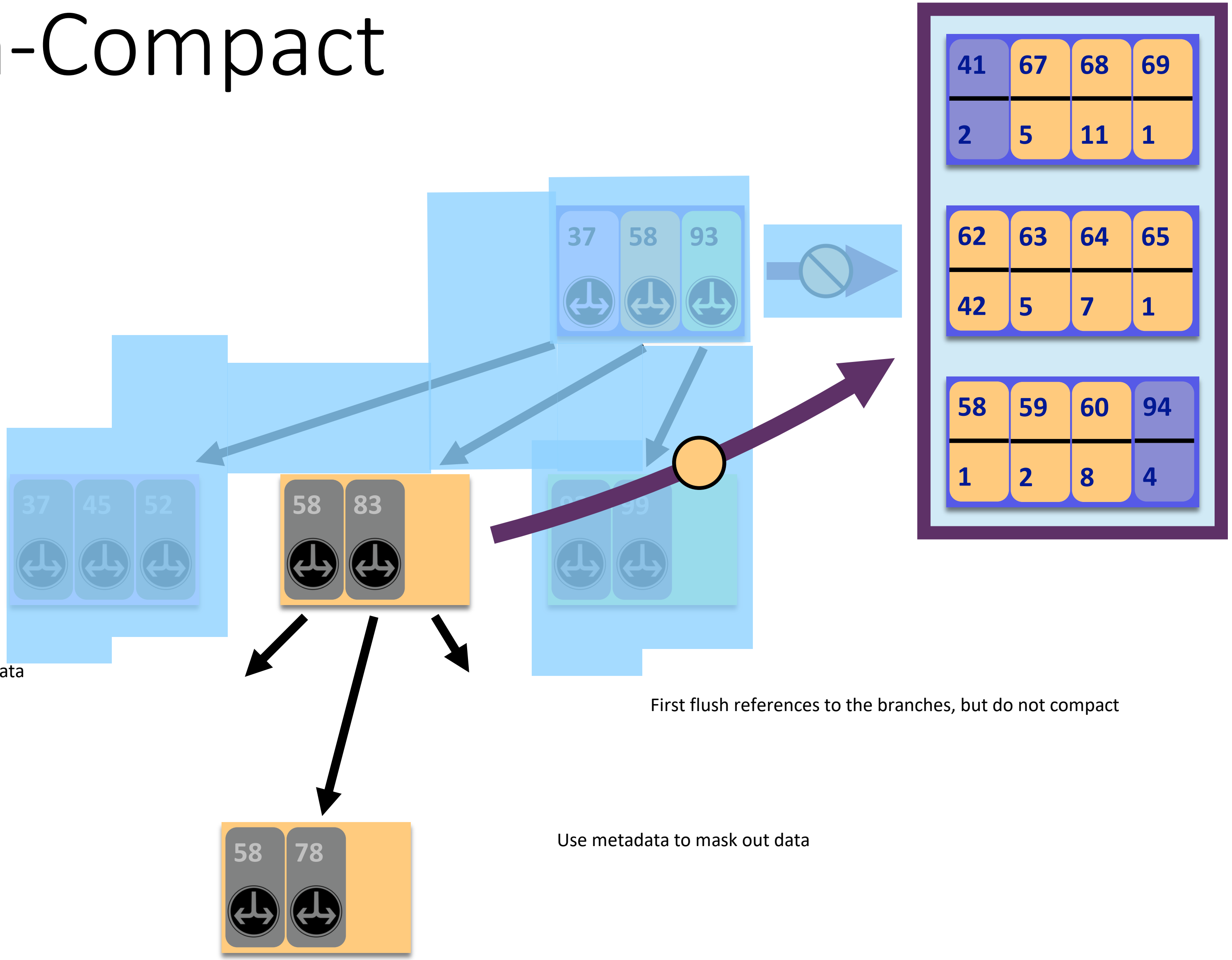
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact



41	67	68	69
2	5	11	1

62	63	64	65
42	5	7	1

58	59	60	94
1	2	8	4

The child only sees the flushed data

First flush references to the branches, but do not compact

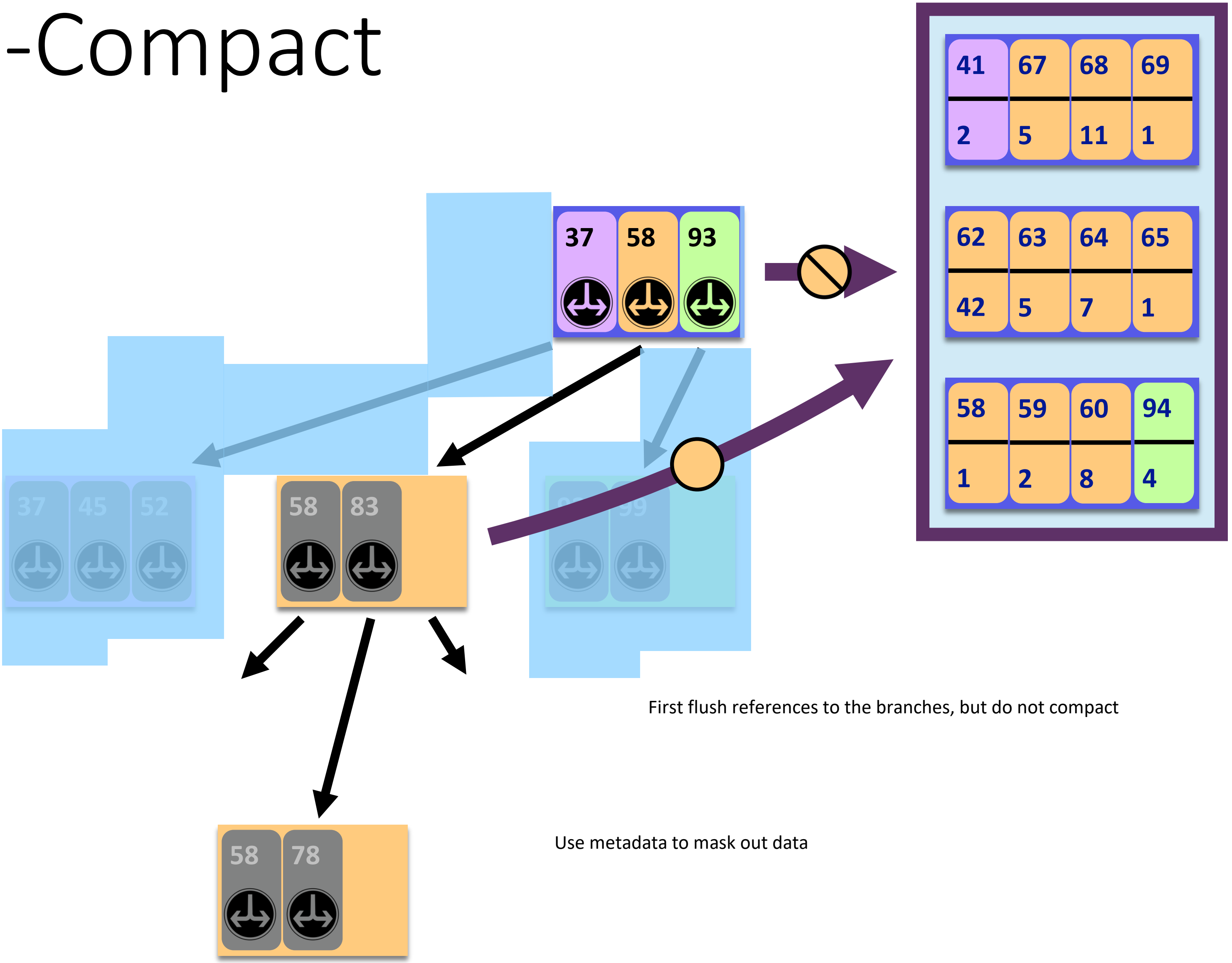
Use metadata to mask out data

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact

Then can flush again



First flush references to the branches, but do not compact

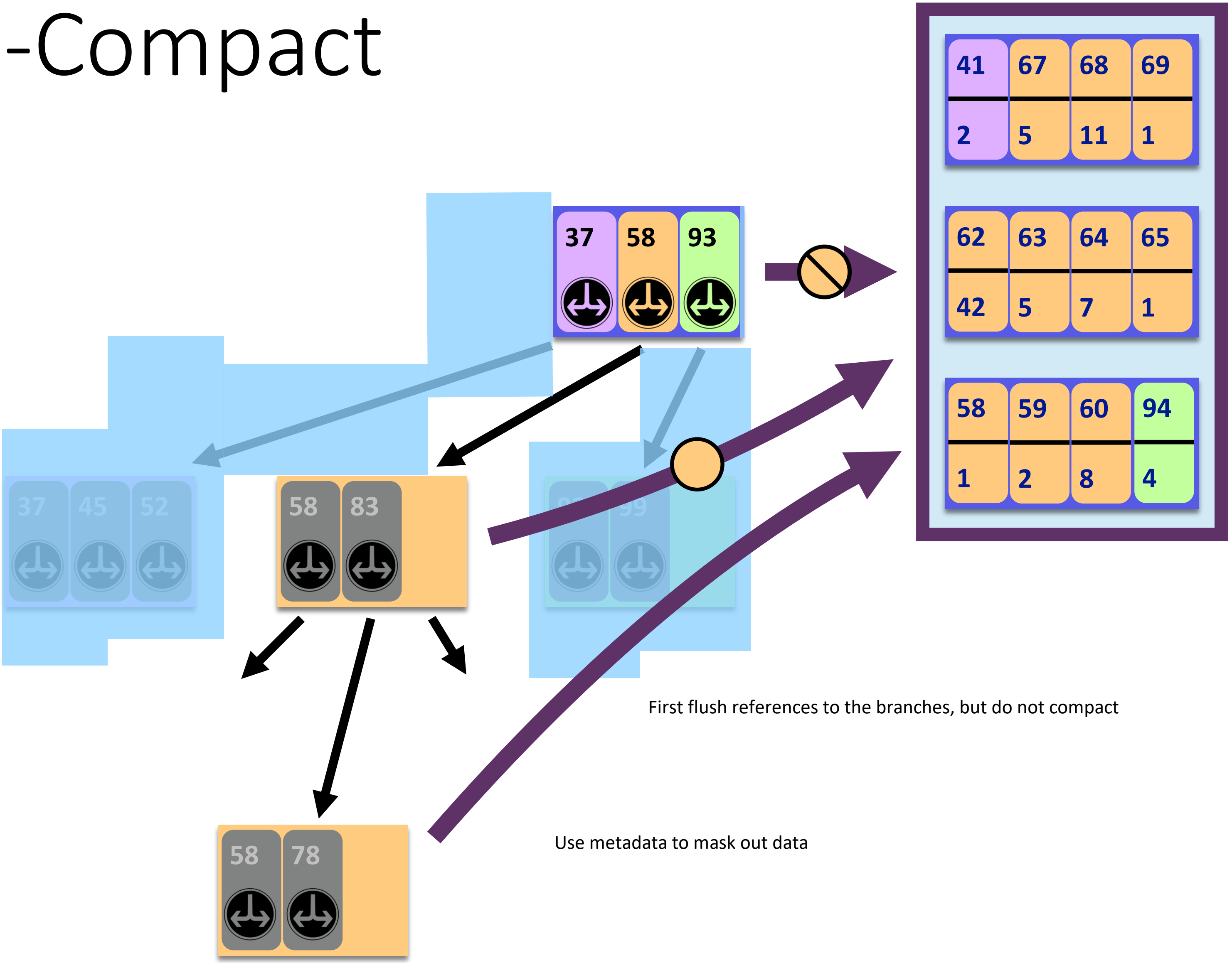
Use metadata to mask out data

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact

Then can flush again



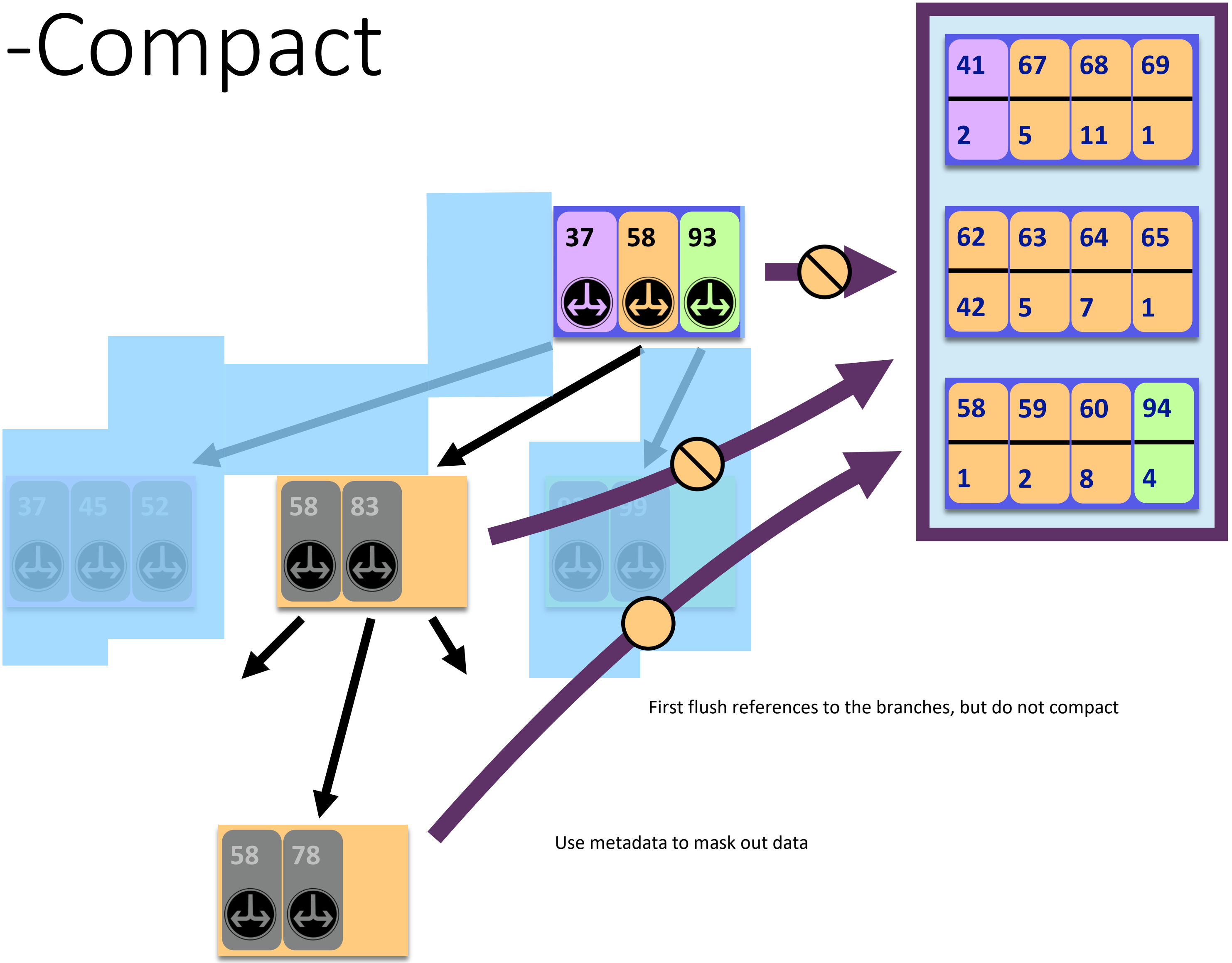
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact



Then can flush again

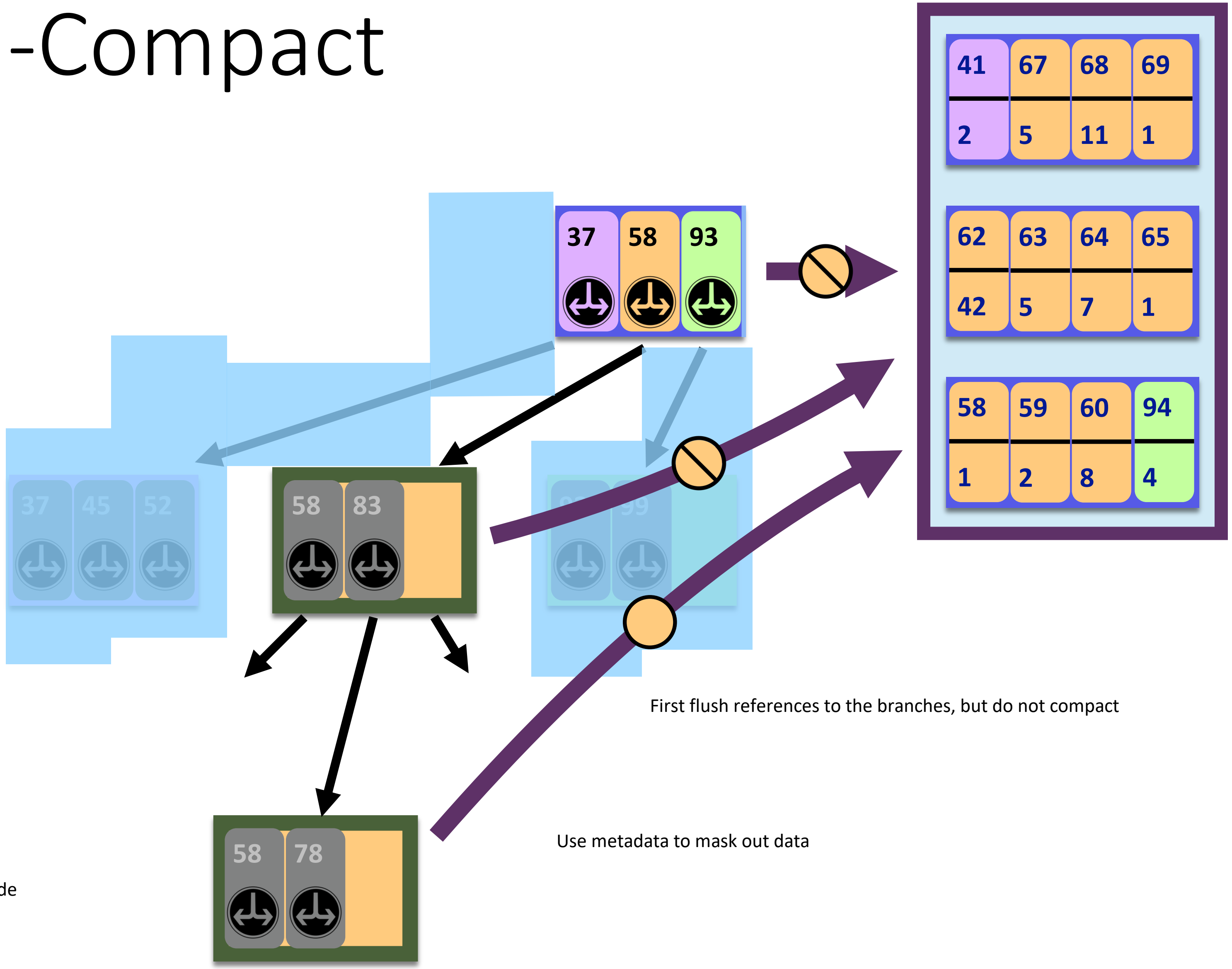
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

Want:
Cheap sequential insertions

Idea: Flush-then-compact



First flush references to the branches, but do not compact

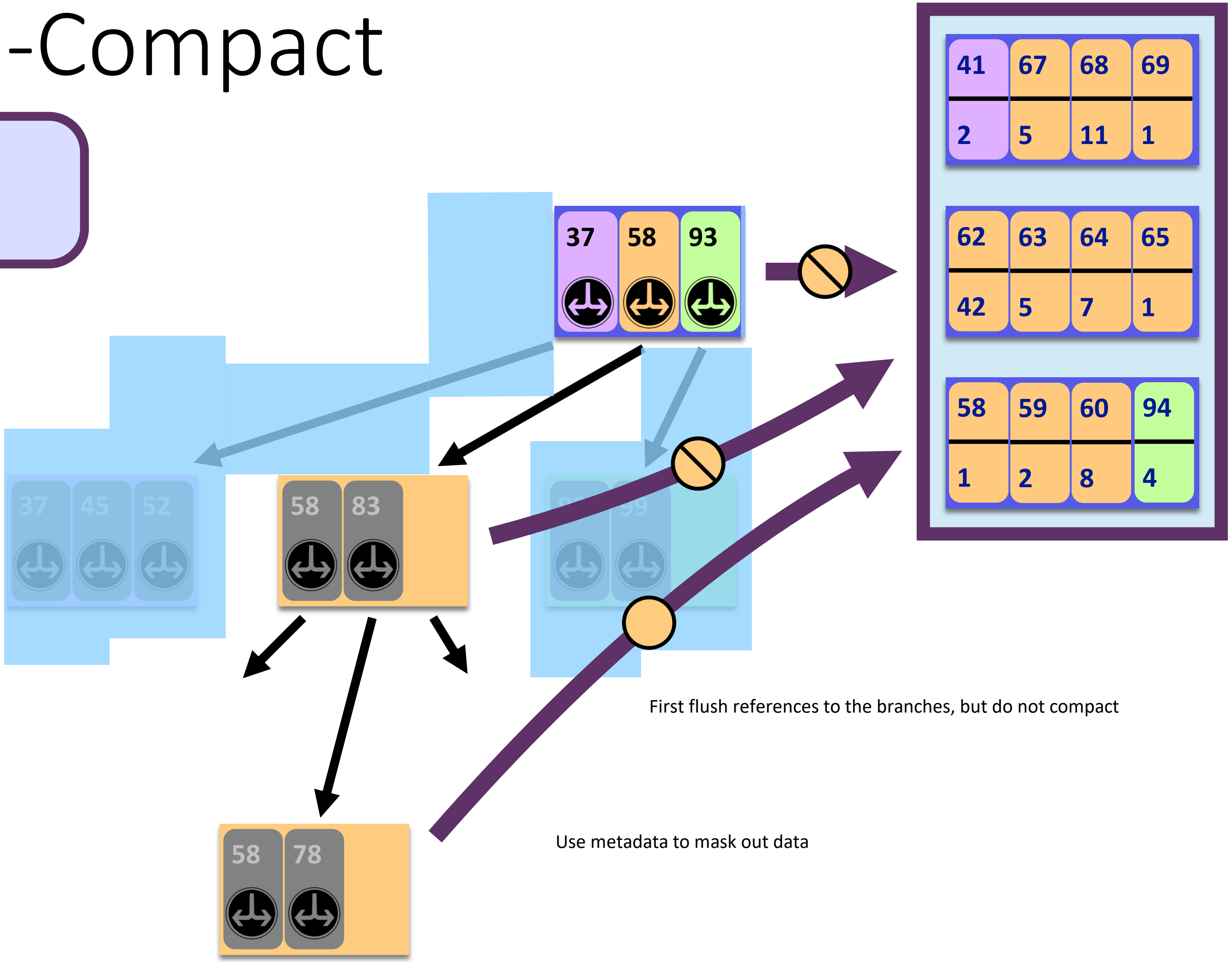
Use metadata to mask out data

Then can flush again

Finally, asynchronously compact the flushed buffers in each node

Flush-Then-Compact

No work on immediately flushed data



41	67	68	69
2	5	11	1

62	63	64	65
42	5	7	1

58	59	60	94
1	2	8	4

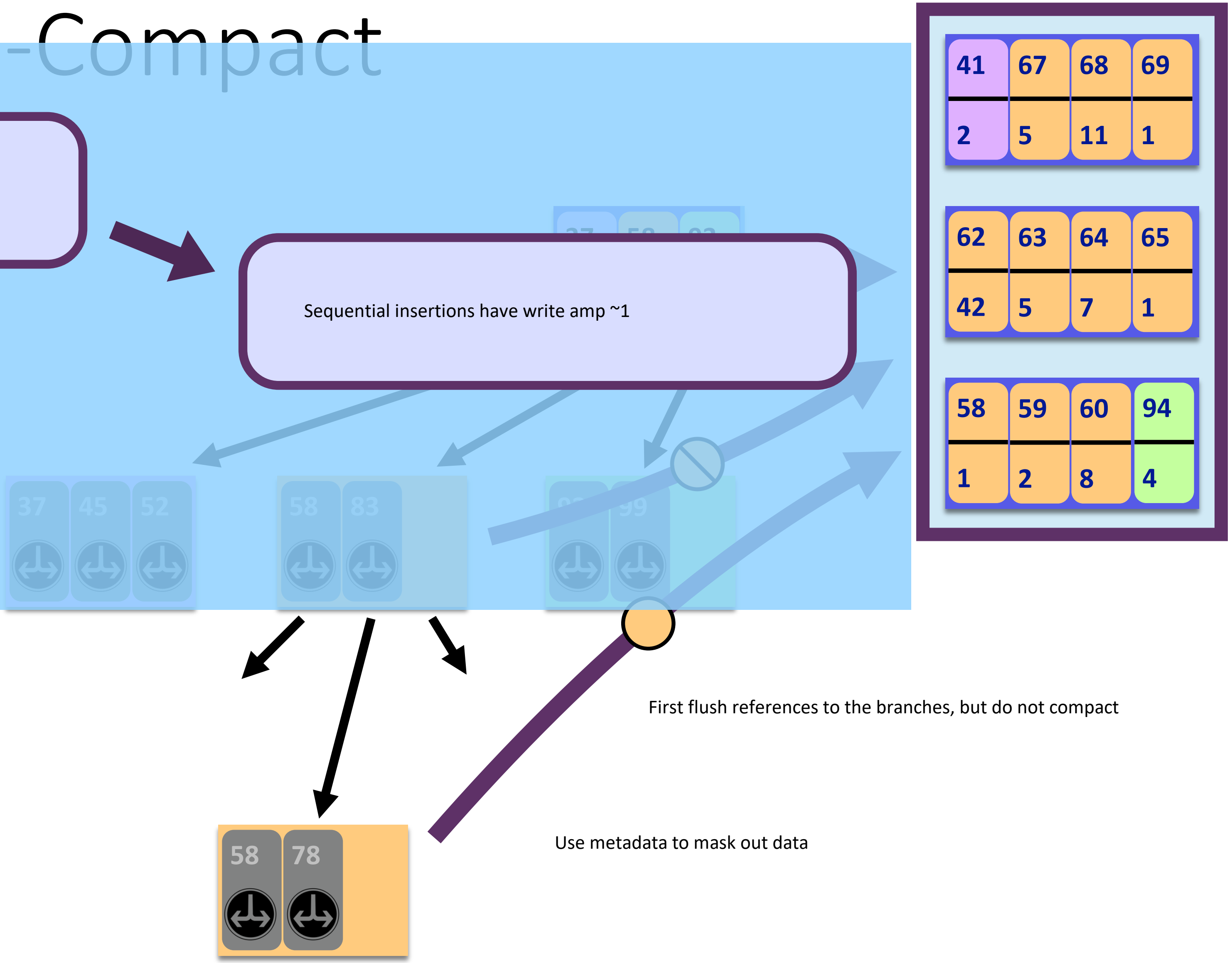
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Sequential insertions have write amp ~1



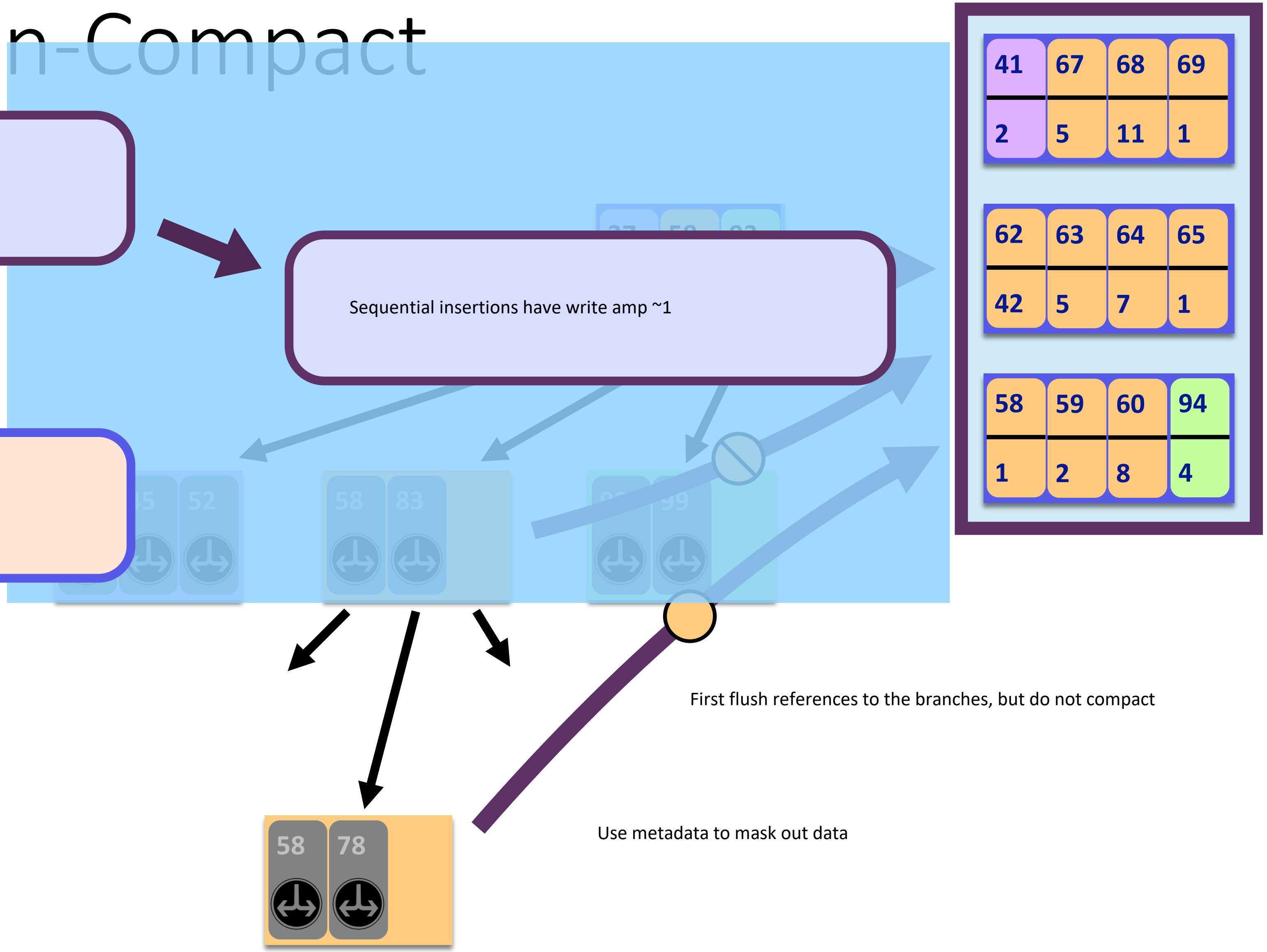
First flush references to the branches, but do not compact

Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Break a serial chain of compactions into parallel



Sequential insertions have write amp ~1

First flush references to the branches, but do not compact

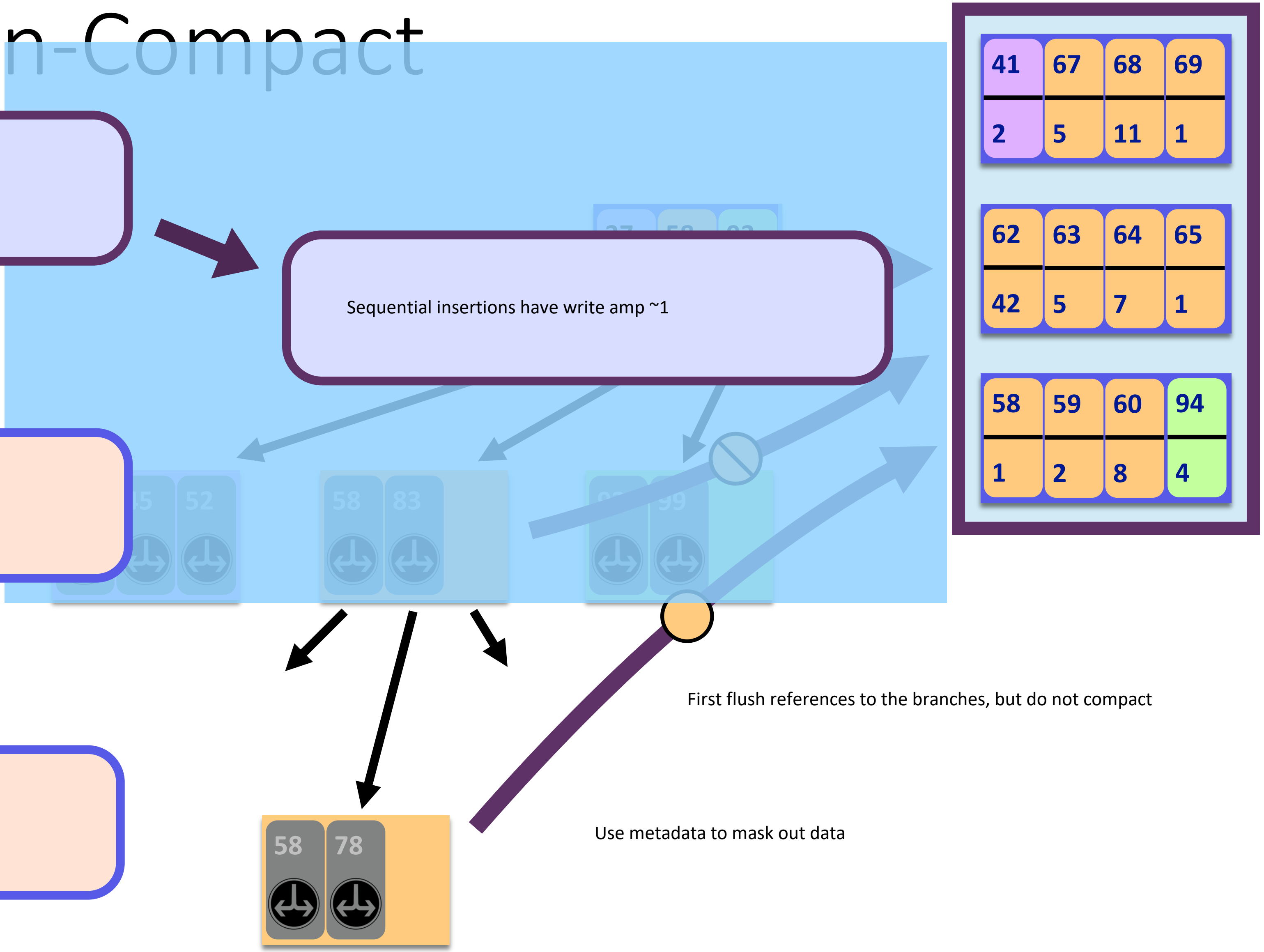
Use metadata to mask out data

Flush-Then-Compact

No work on immediately flushed data

Break a serial chain of compactions into parallel

Concurrent compactions in trunk nodes

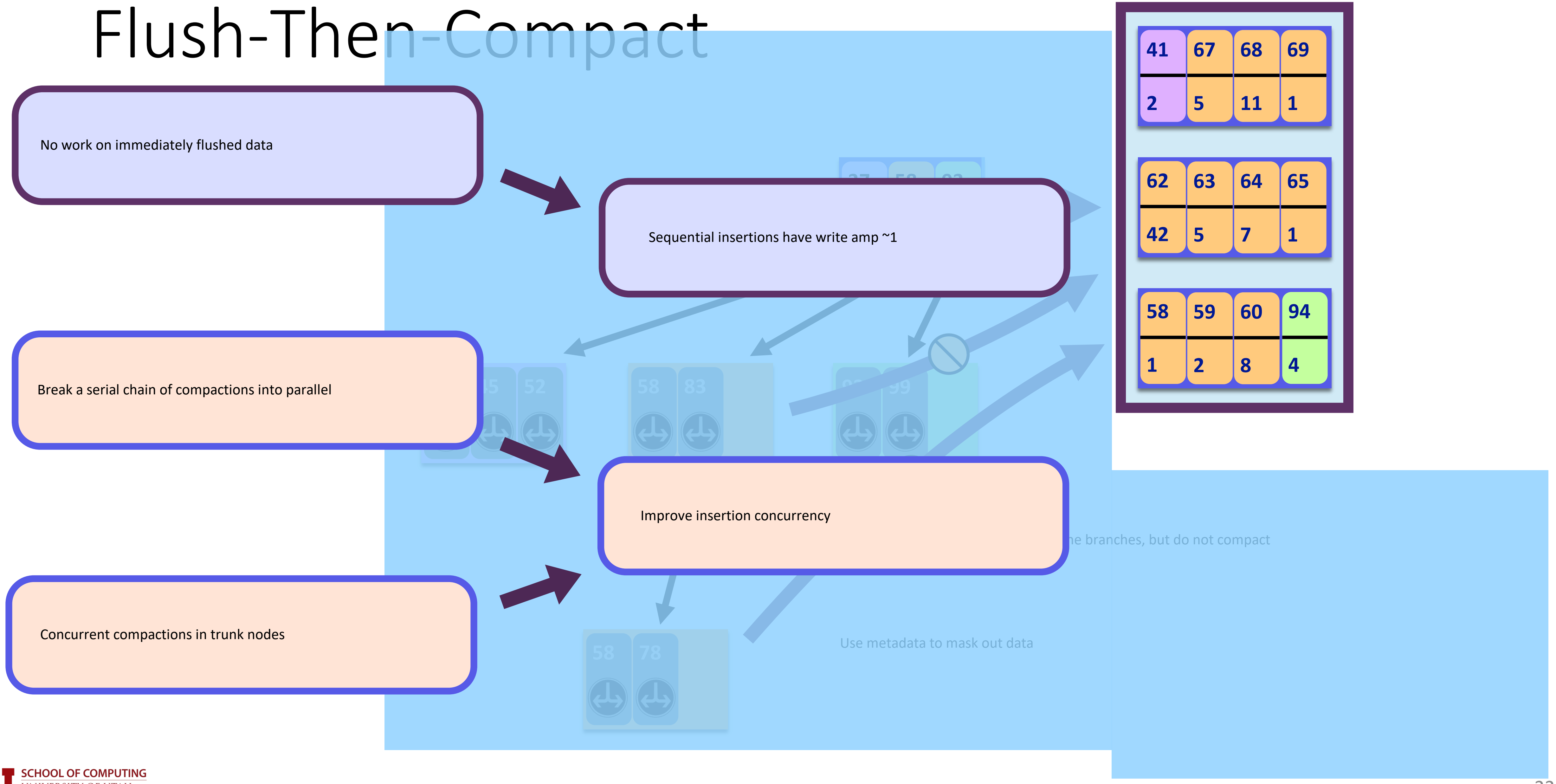


Sequential insertions have write amp ~1

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

Sequential insertions have write amp ~1

Break a serial chain of compactions into parallel

Improve insertion concurrency

Concurrent compactions in trunk nodes

he branches, but do not compact

Use metadata to mask out data

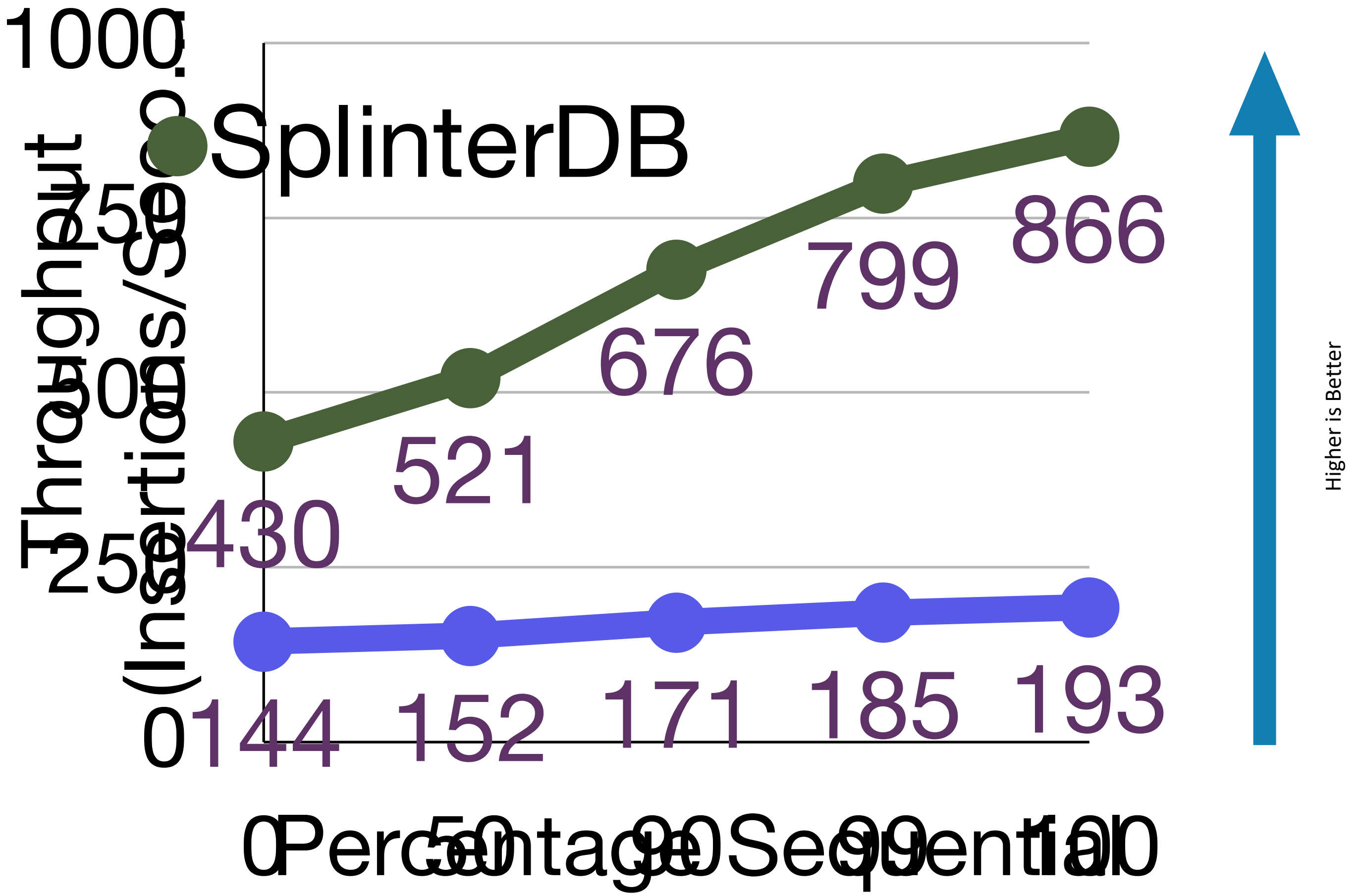
41	67	68	69
2	5	11	1

62	63	64	65
42	5	7	1

58	59	60	94
1	2	8	4

Flush-Then-Compact

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

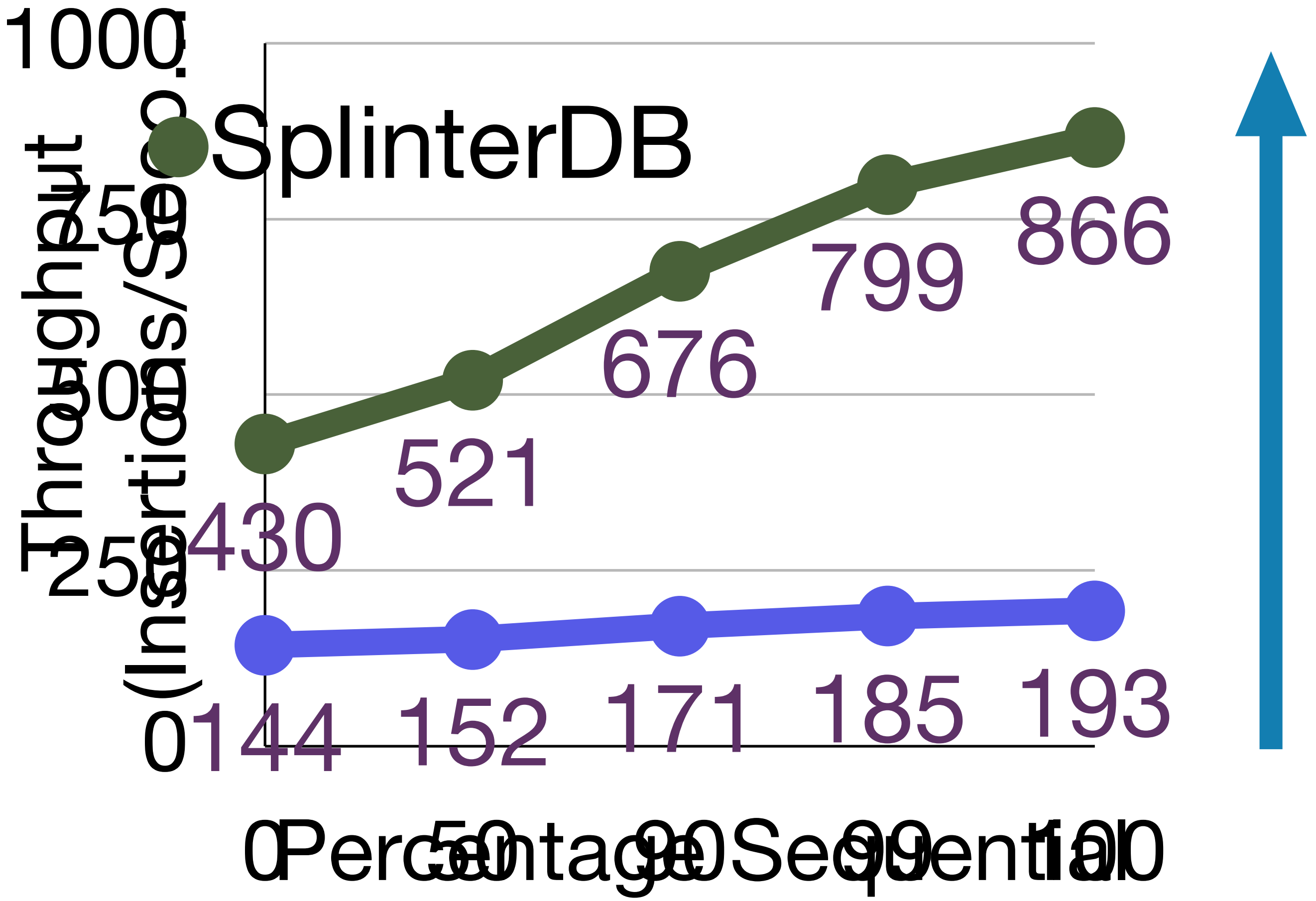


X-axis not to scale

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



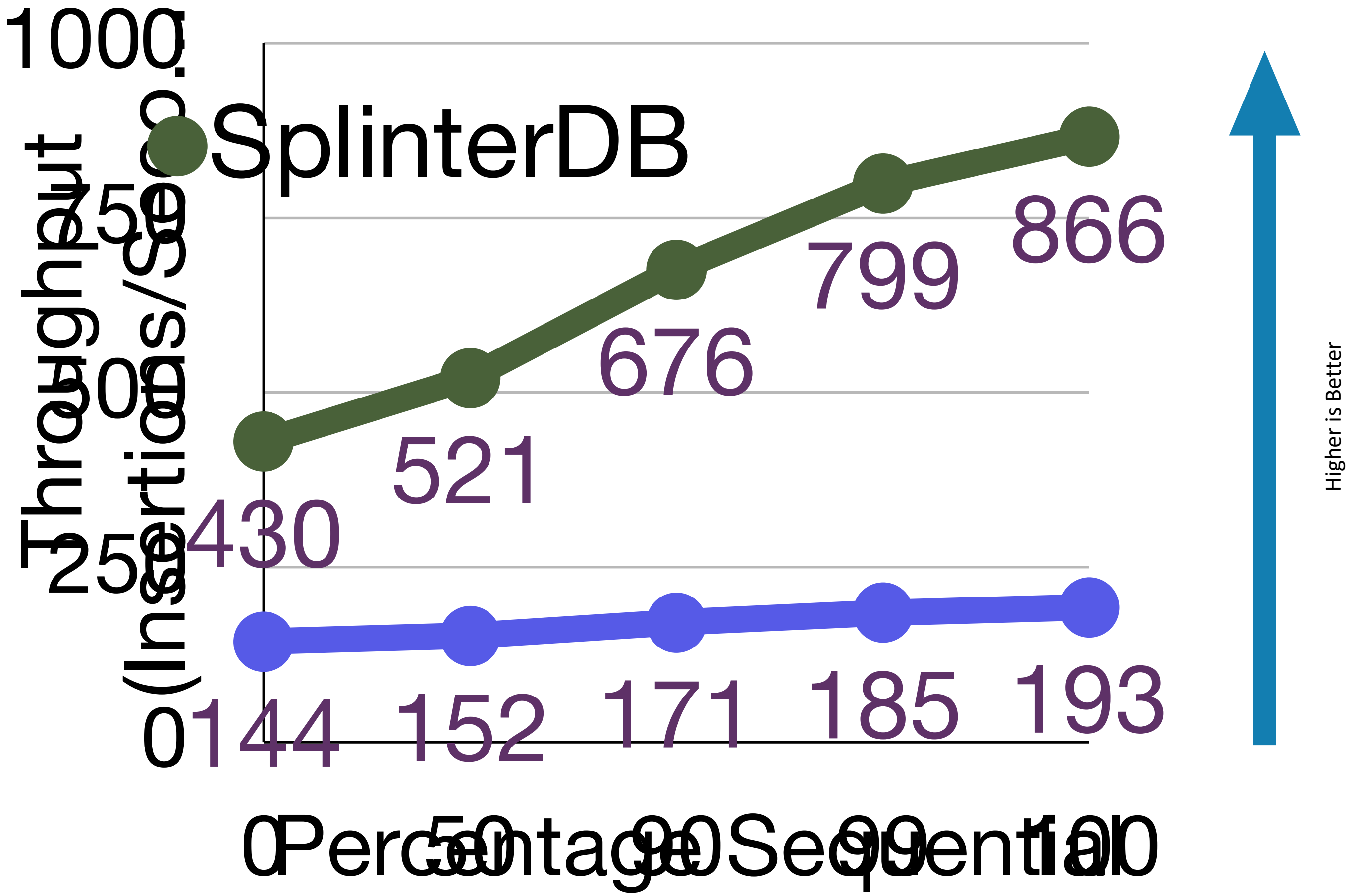
X-axis not to scale

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

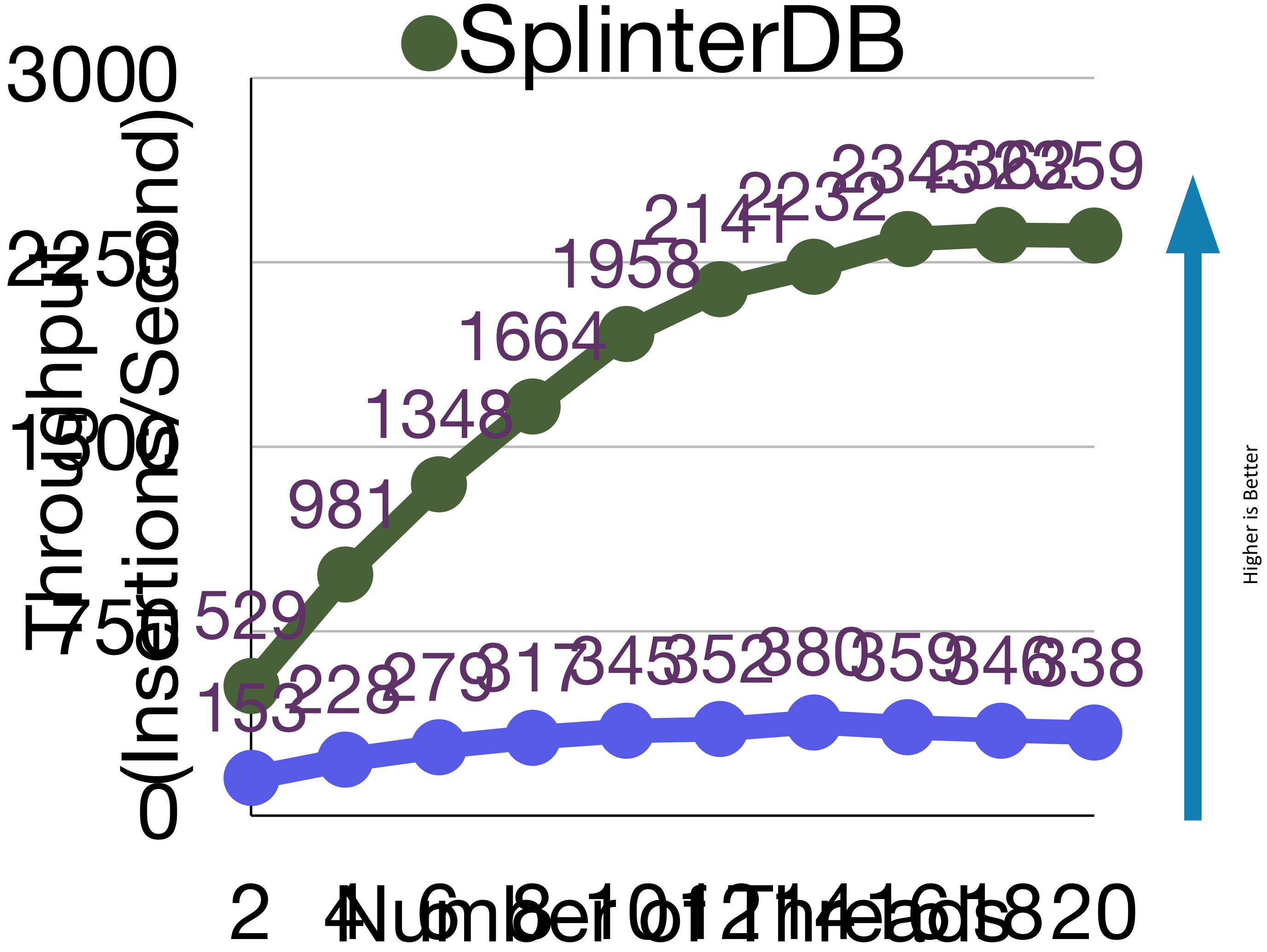
RocksDB improves, but at a much lower rate



X-axis not to scale

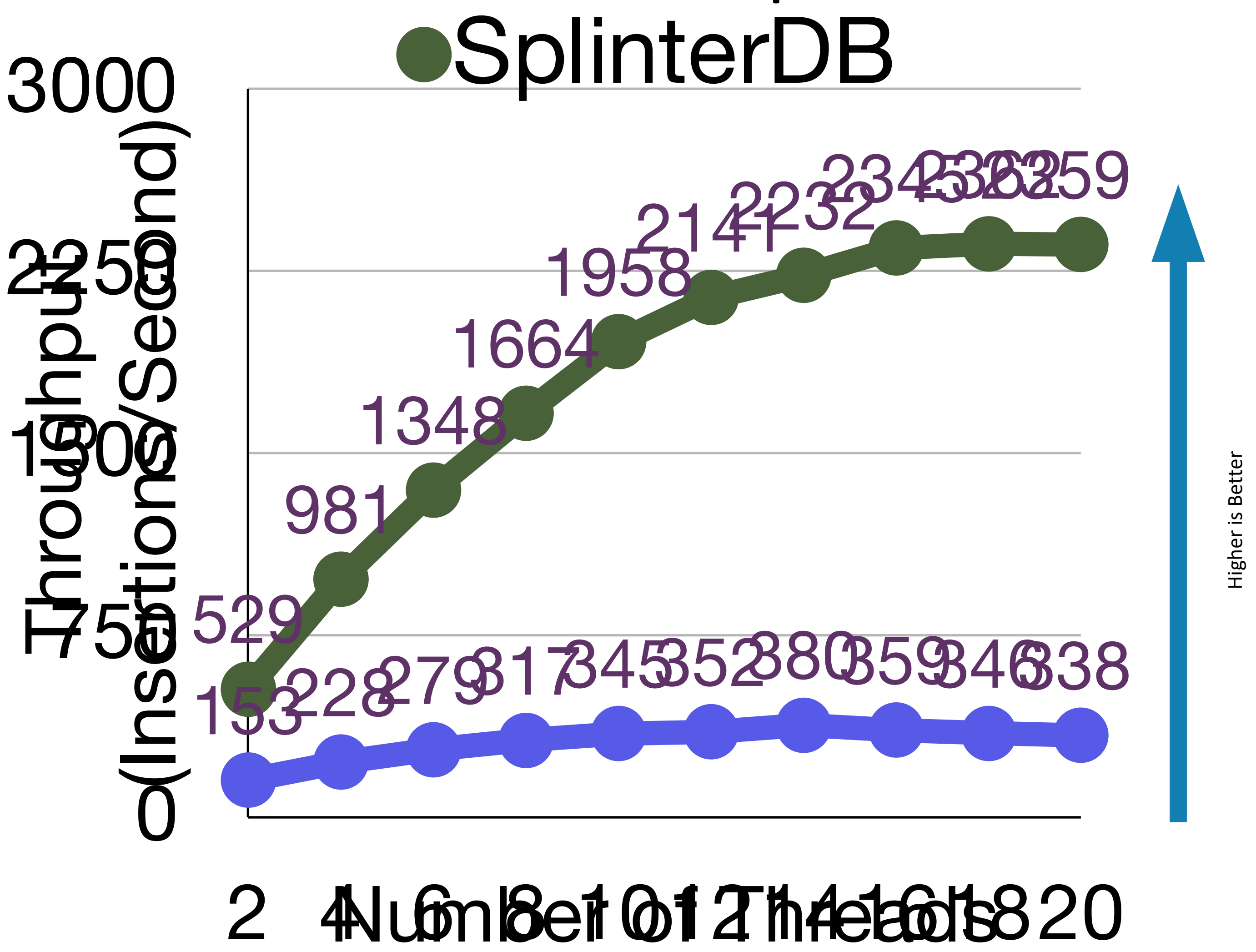
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Flush-then-Compact



Insertions in SplinterDB scale well

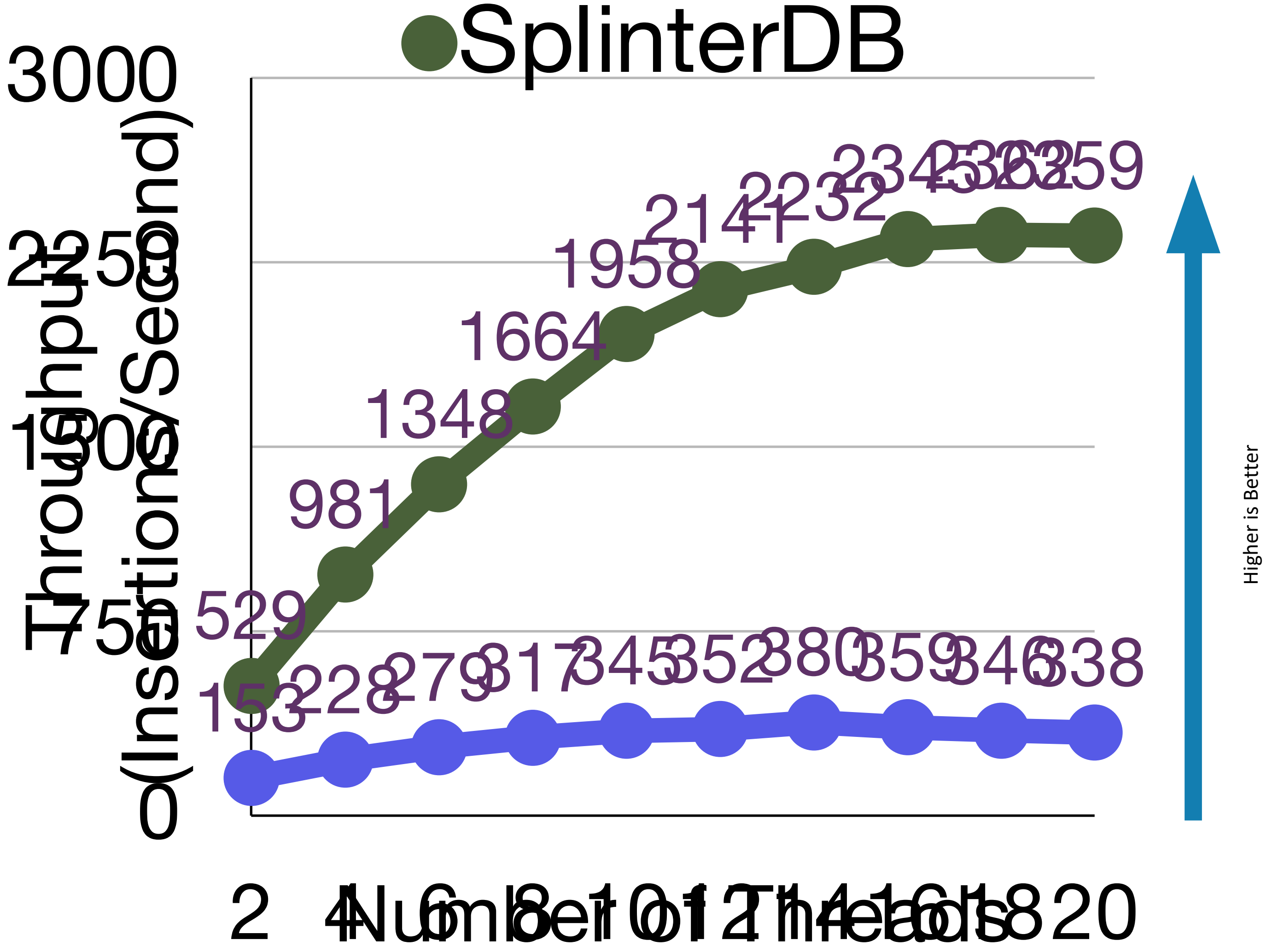
Flush-then-Compact



Insertions in SplinterDB scale well

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

Flush-then-Compact



Insertions in SplinterDB scale well

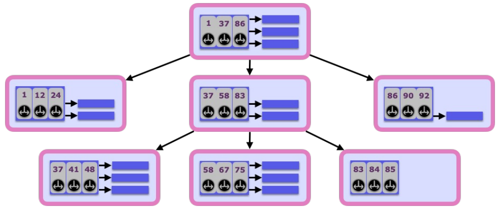
At 12 threads, SplinterDB has 7x the throughput of 1 thread

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

Conclusion

Model the problem:
external memory dictionary

Mapped B^{ε}-tree



Theory

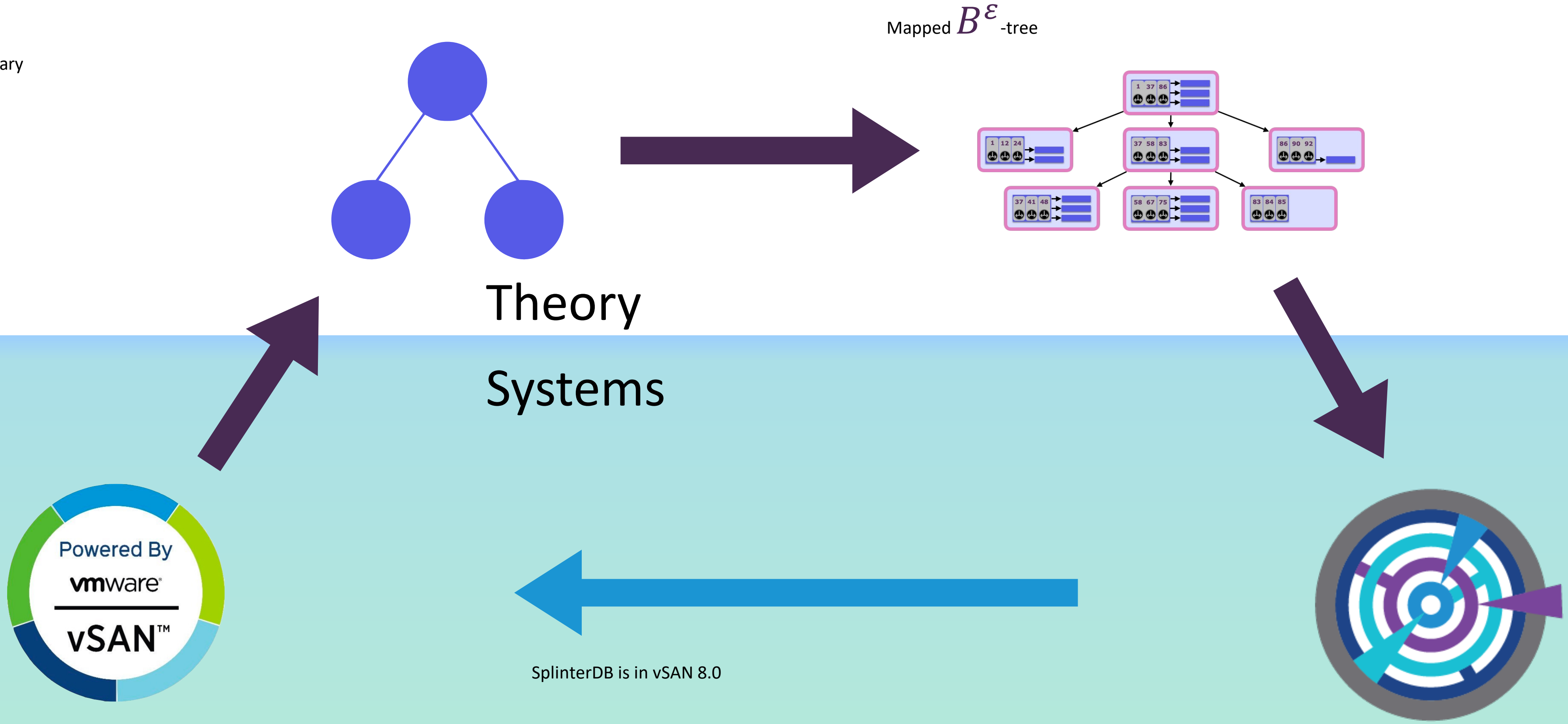
Systems

vSAN needed a new way of storing metadata

SplinterDB

Conclusion

Model the problem:
external memory dictionary



Mapped B^{ε}-tree

Theory

Systems

SplinterDB is in vSAN 8.0

vSAN needed a new way of storing metadata

Open-source at
https://github.com/vmware/splinterdb

SplinterDB