CS 6530: Advanced Database Systems Fall 2024

Lecture 9
B^e-tree and SplinterDB

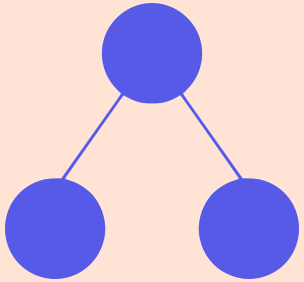
Prashant Pandey

prashant.pandey@utah.edu

Slides taken from Prof. Alex Conway, Cornell Tech

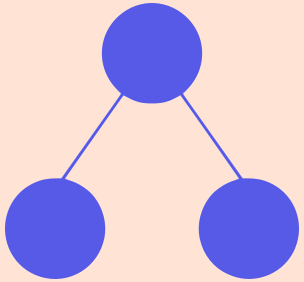
The Story of SplinterDB

Model the problem:
external memory dictionary

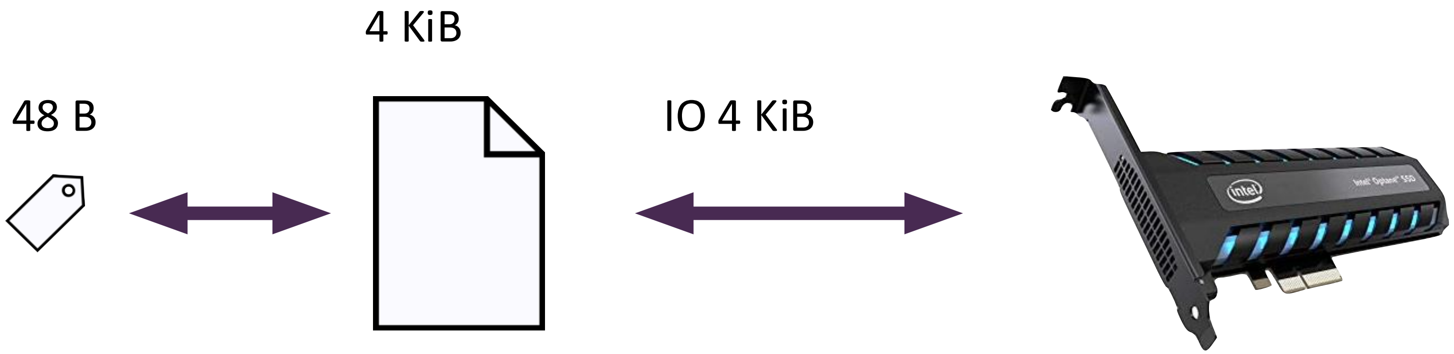


The Story of SplinterDB

Model the problem:
external memory dictionary

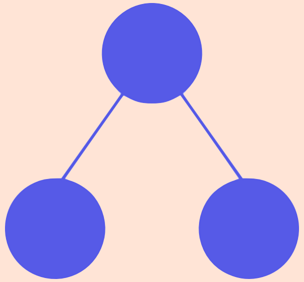


Metadata is fine-grained

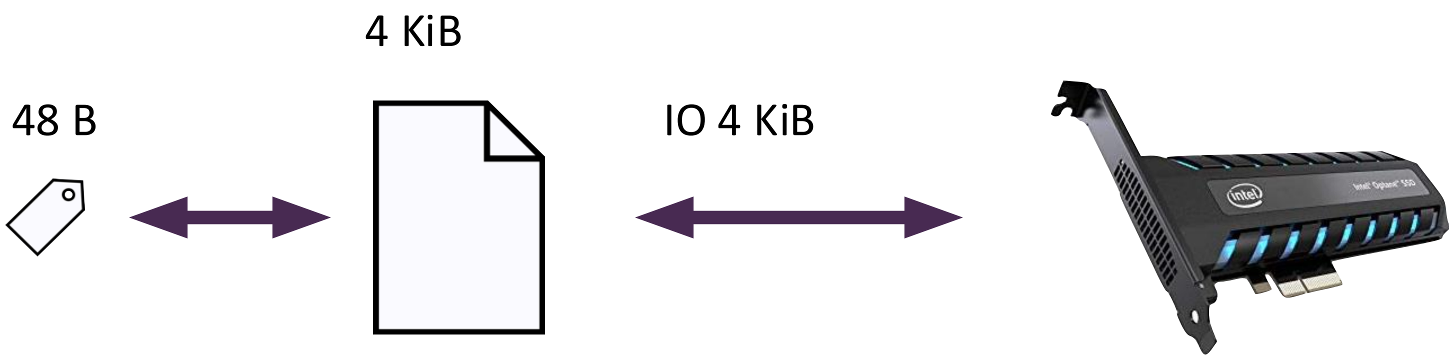


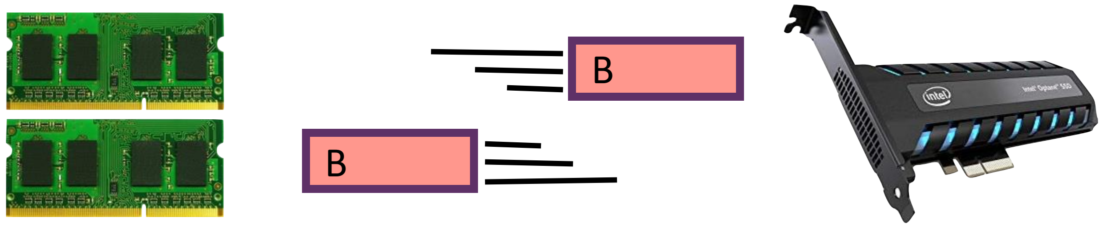
The Story of SplinterDB

Model the problem:
external memory dictionary



Metadata is fine-grained





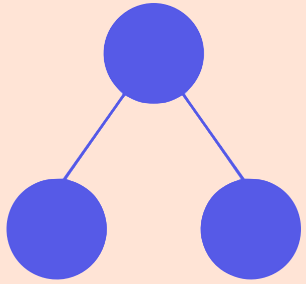
Internal Memory of size M

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

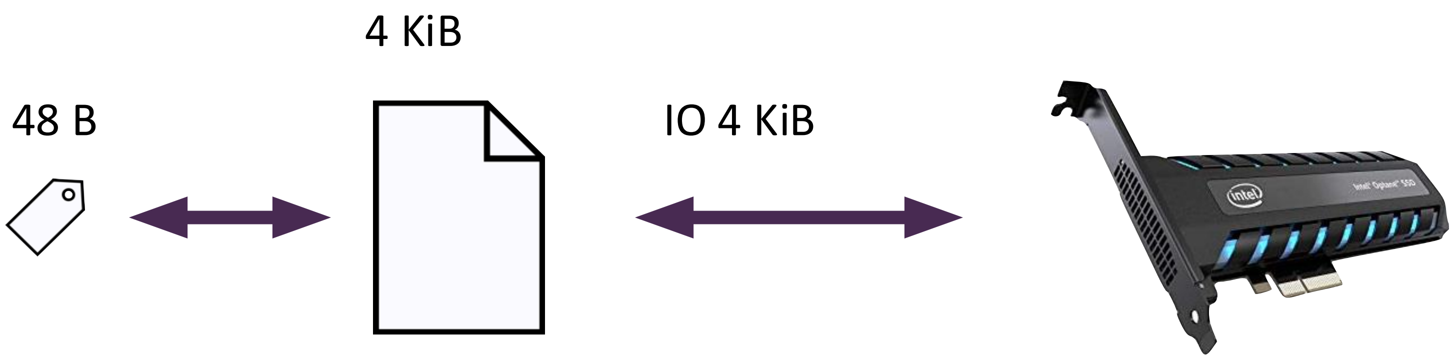
External Memory Model

The Story of SplinterDB

Model the problem:
external memory dictionary

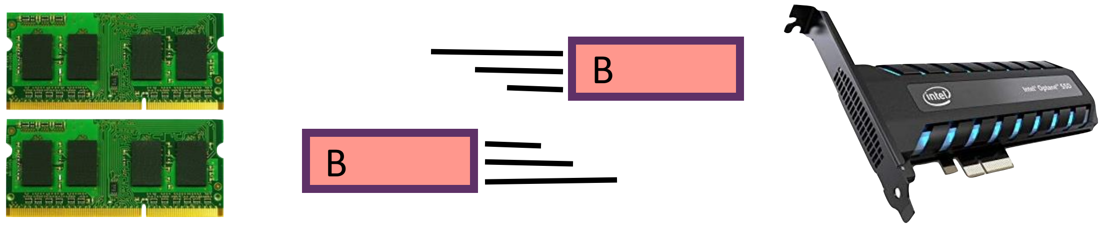


Metadata is fine-grained



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



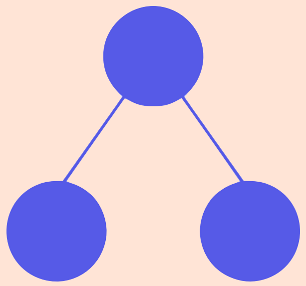
Internal Memory of size M

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)

Lower Bounds

Brodal-Fagerberg Lower Bound

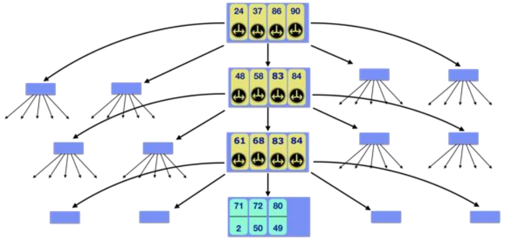
Insertions in

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

↔

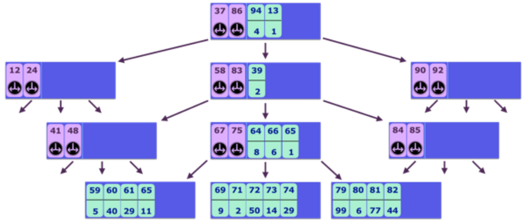
Lookups in

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



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

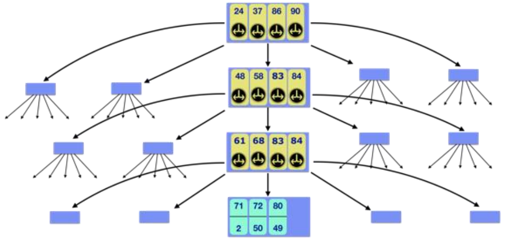
$$\Omega(\log_{\lambda} N)$$

General External Memory Model

Lower Bounds

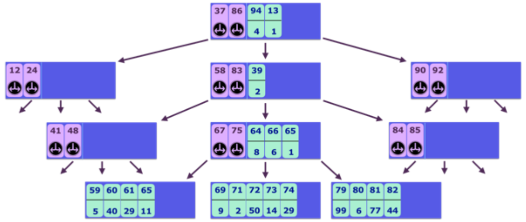
Brodal-Fagerberg Lower Bound

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

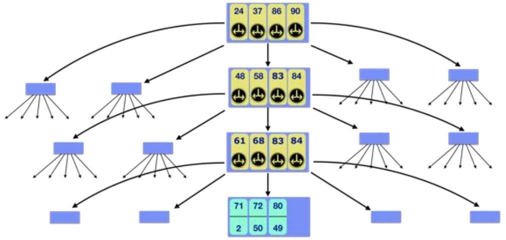
Brodal-Fagerberg Lower Bound

Insertions in

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

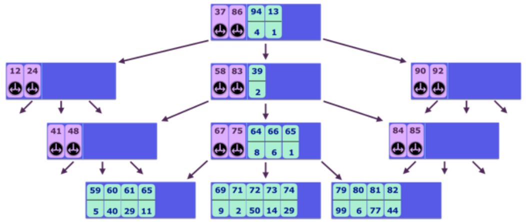
Lookups in

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



B-Trees

$$(\lambda = B)$$



Bᵋ-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

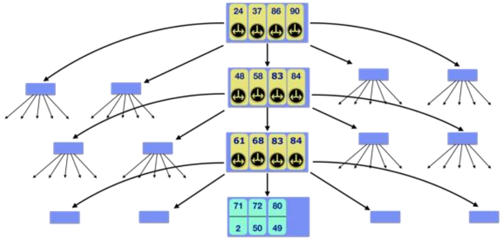
BoA/BoT
Hash Table

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

Lower Bounds

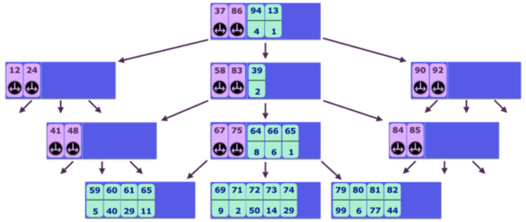
Brodal-Fagerberg Lower Bound

Insertions in $O\left(\frac{\lambda}{B}\log_\lambda N\right)$ ↔ Lookups in $\Omega(\log_\lambda N)$



B-Trees

$(\lambda = B)$



B$^\varepsilon$-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

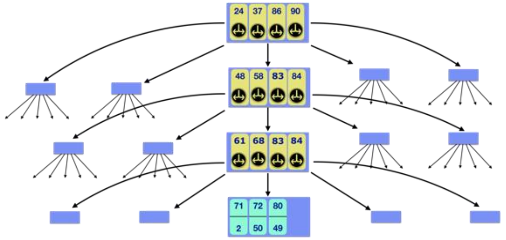
General External Memory Model

Lower Bounds

Brodal-Fagerberg Lower Bound

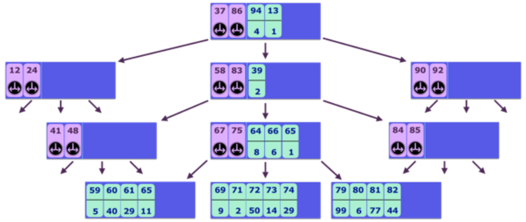
Insertions in

$O\left(\frac{\lambda}{B}\log_\lambda N\right)$ ⟷ Lookups in $\Omega(\log_\lambda N)$



B-Trees

$(\lambda = B)$



B$^{\varepsilon}$-Trees

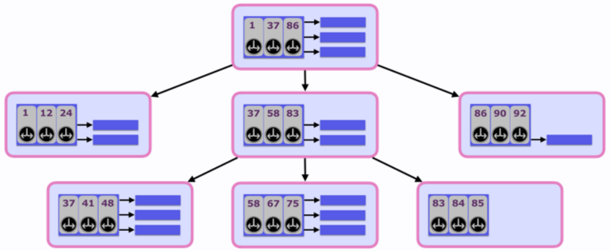
$(\lambda = B^{\varepsilon})$

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

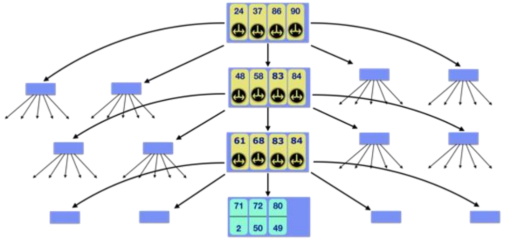
Brodal-Fagerberg Lower Bound

Insertions in

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

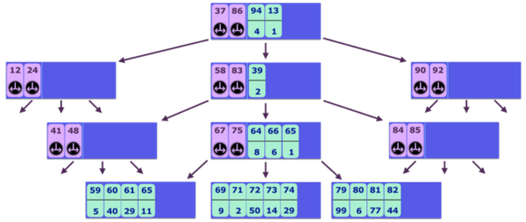
Lookups in

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



B-Trees

$$(\lambda = B)$$



Bᵋ-Trees

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

Comparison External Memory Model

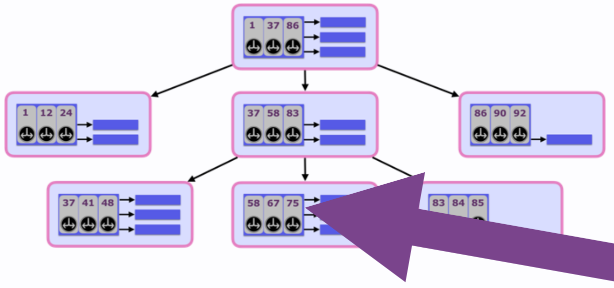
Iacono-Pătrașcu Lower Bound

Insertions in

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

Lookups in

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



Mapped Bᵋ-Trees

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

Iacono-Patrascu Hash Table

Hash Table

General External Memory Model

I/O Amplification

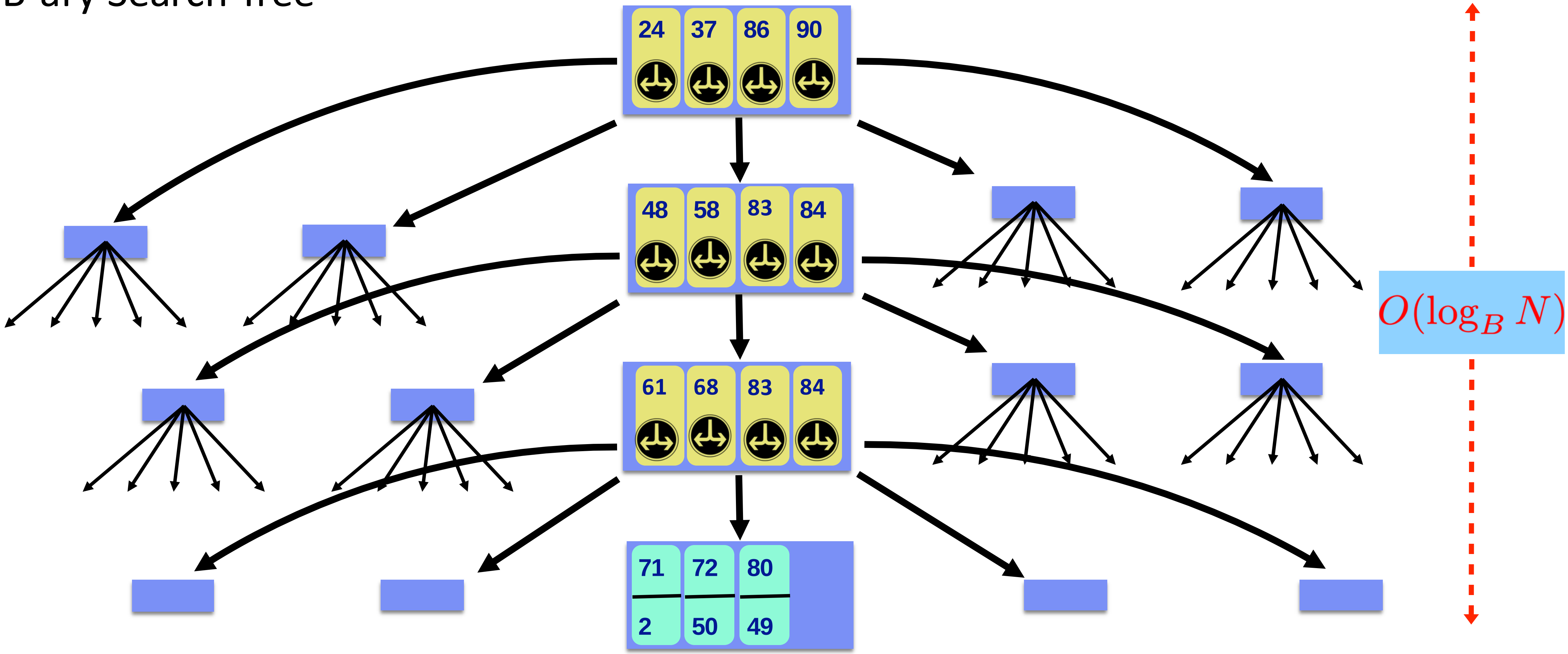
Read amplification is the ratio of the number of blocks read from the disk versus the number of blocks required to read the key-value pair.

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

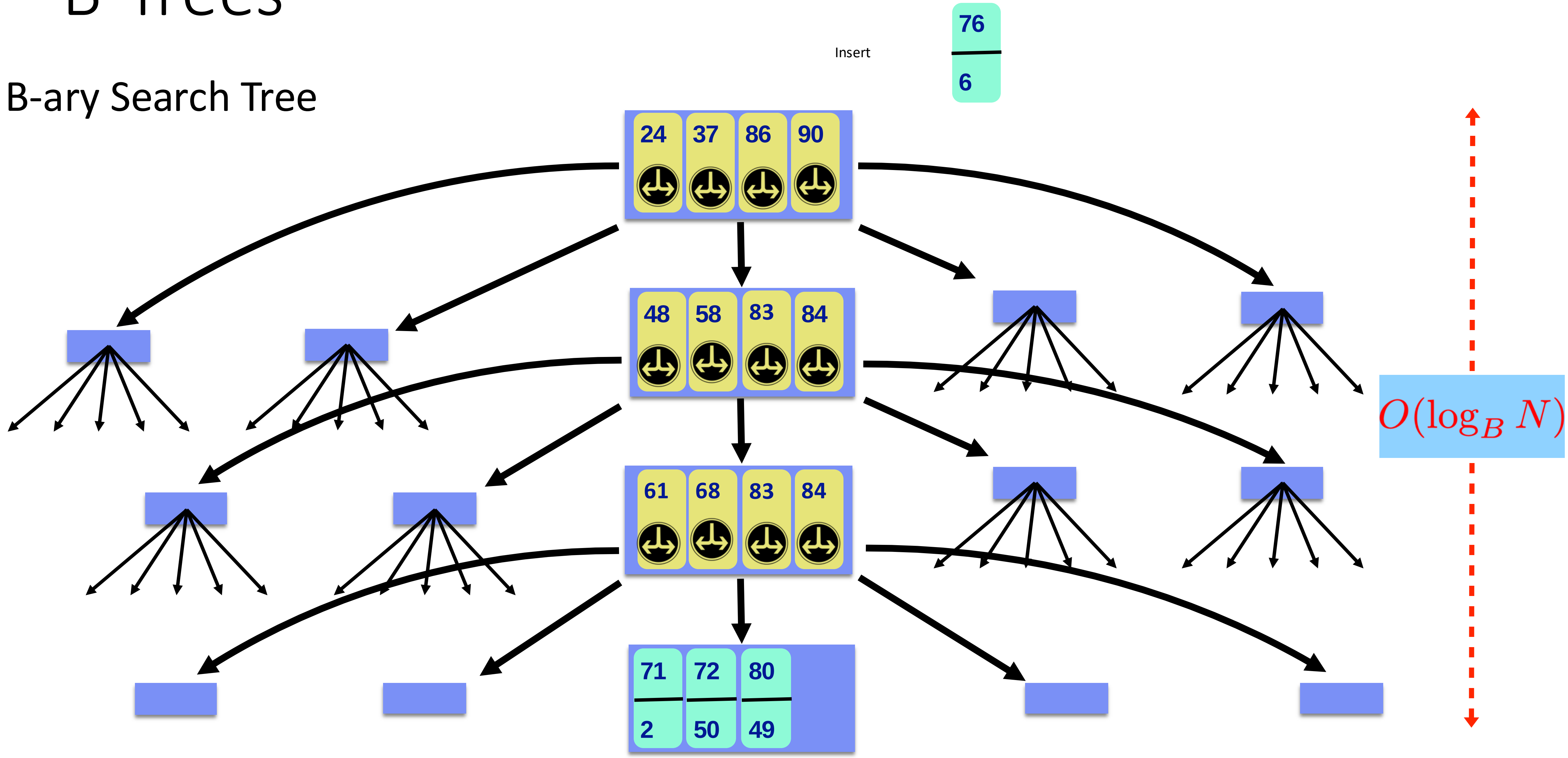
B-Trees

B-ary Search Tree



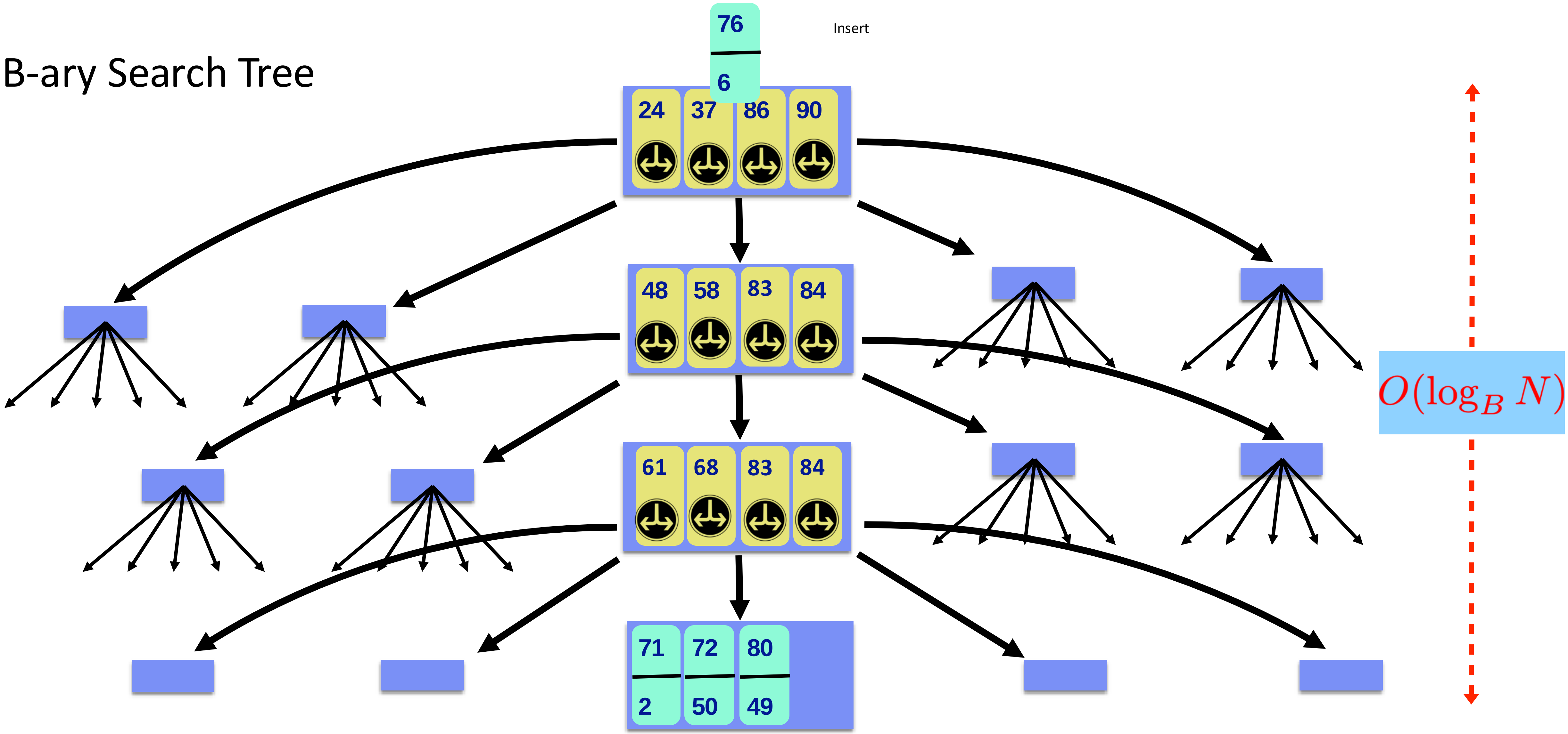
B-Trees

B-ary Search Tree



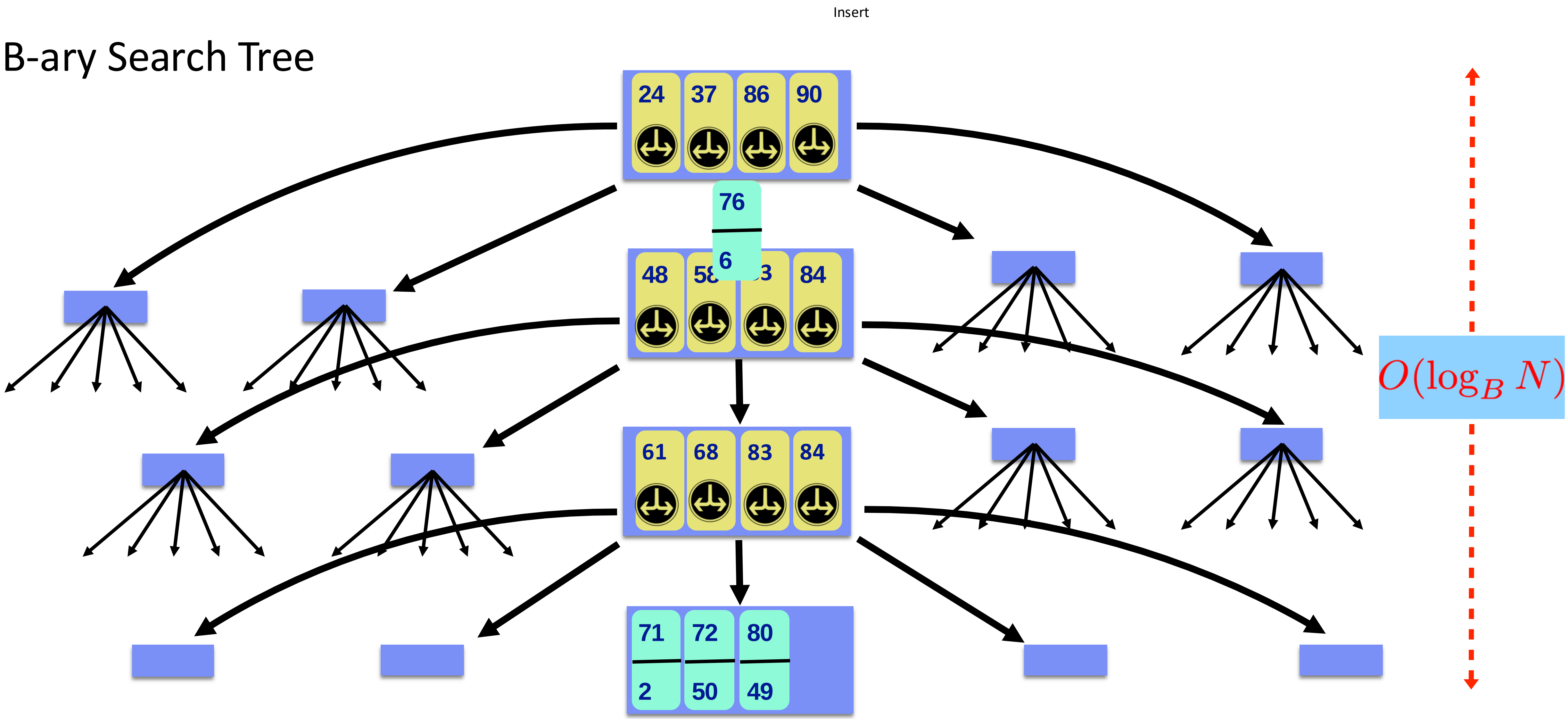
B-Trees

B-ary Search Tree



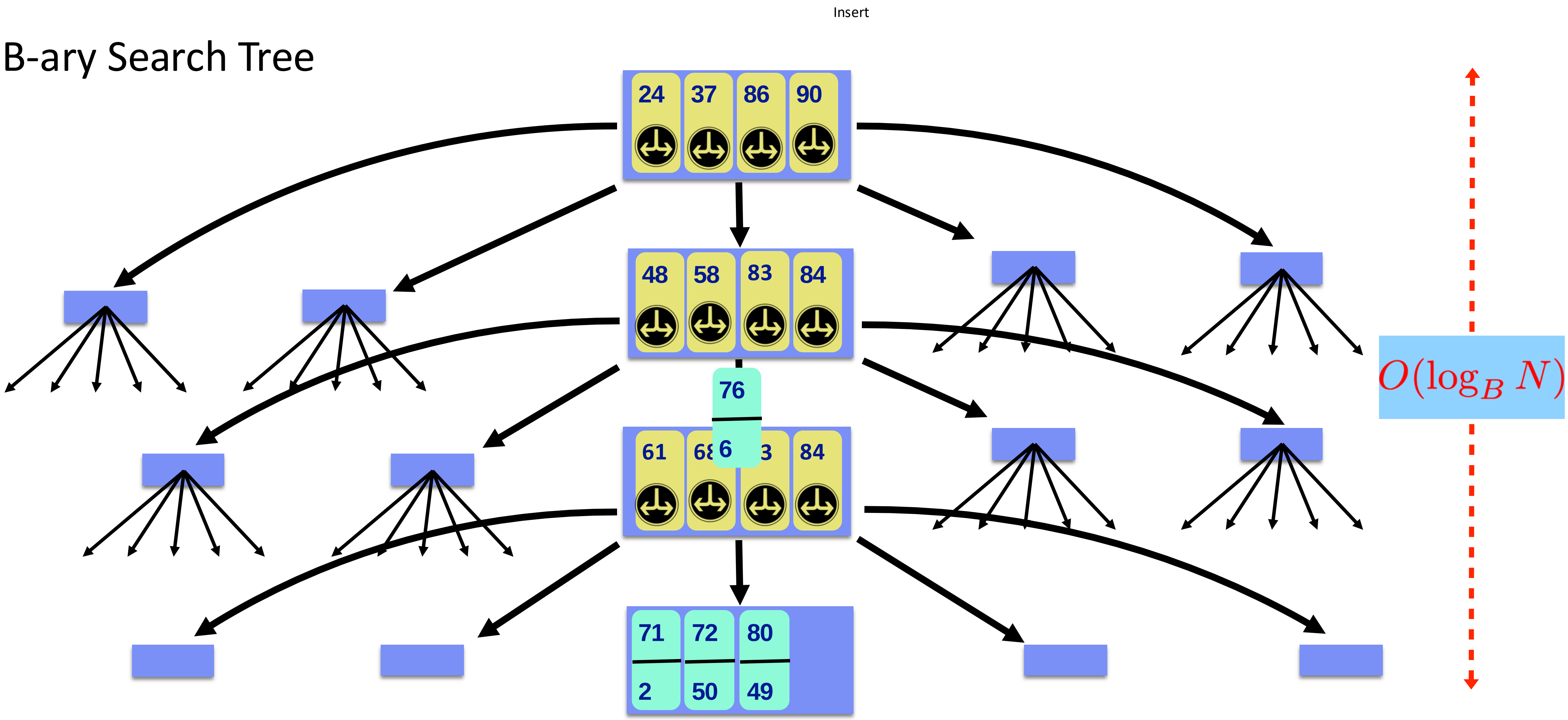
B-Trees

B-ary Search Tree



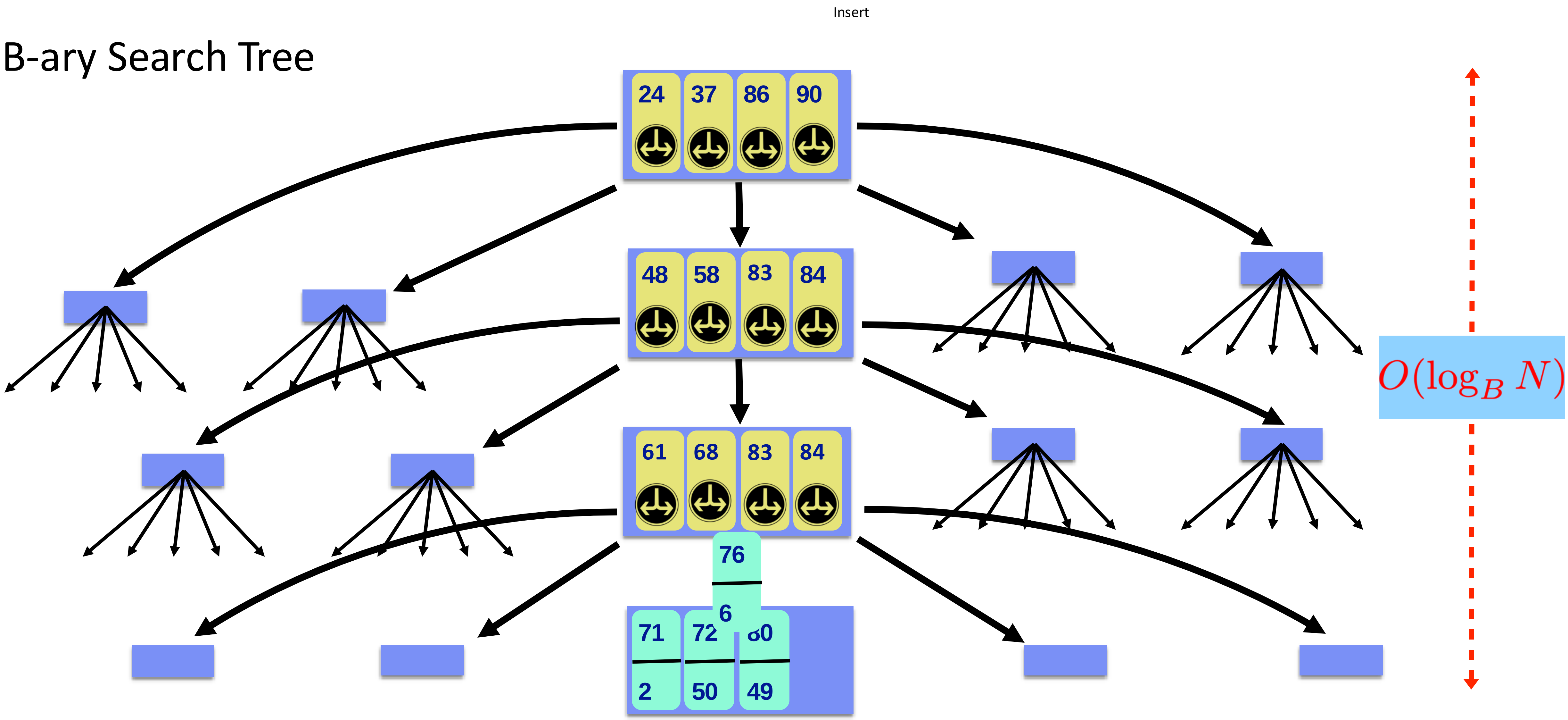
B-Trees

B-ary Search Tree



B-Trees

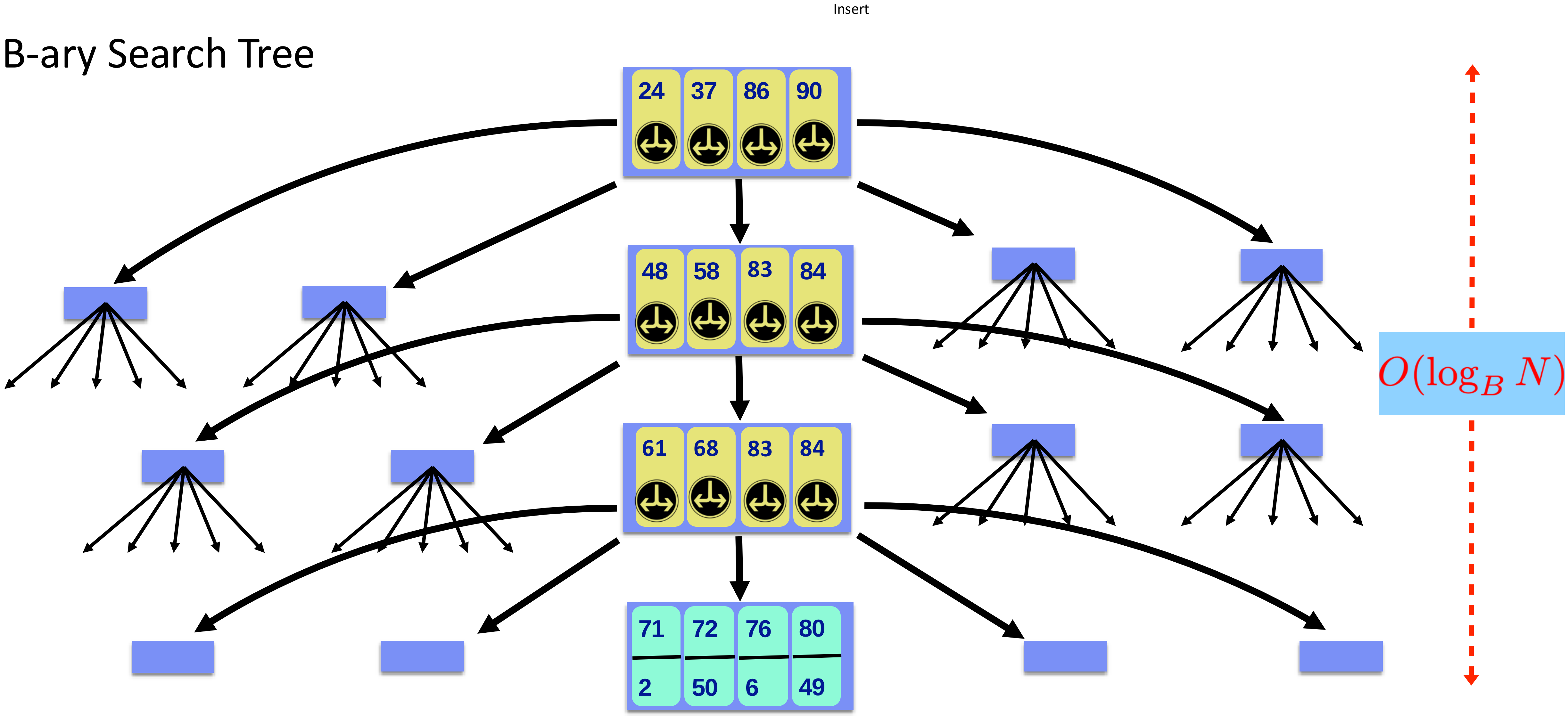
B-ary Search Tree



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B-Trees

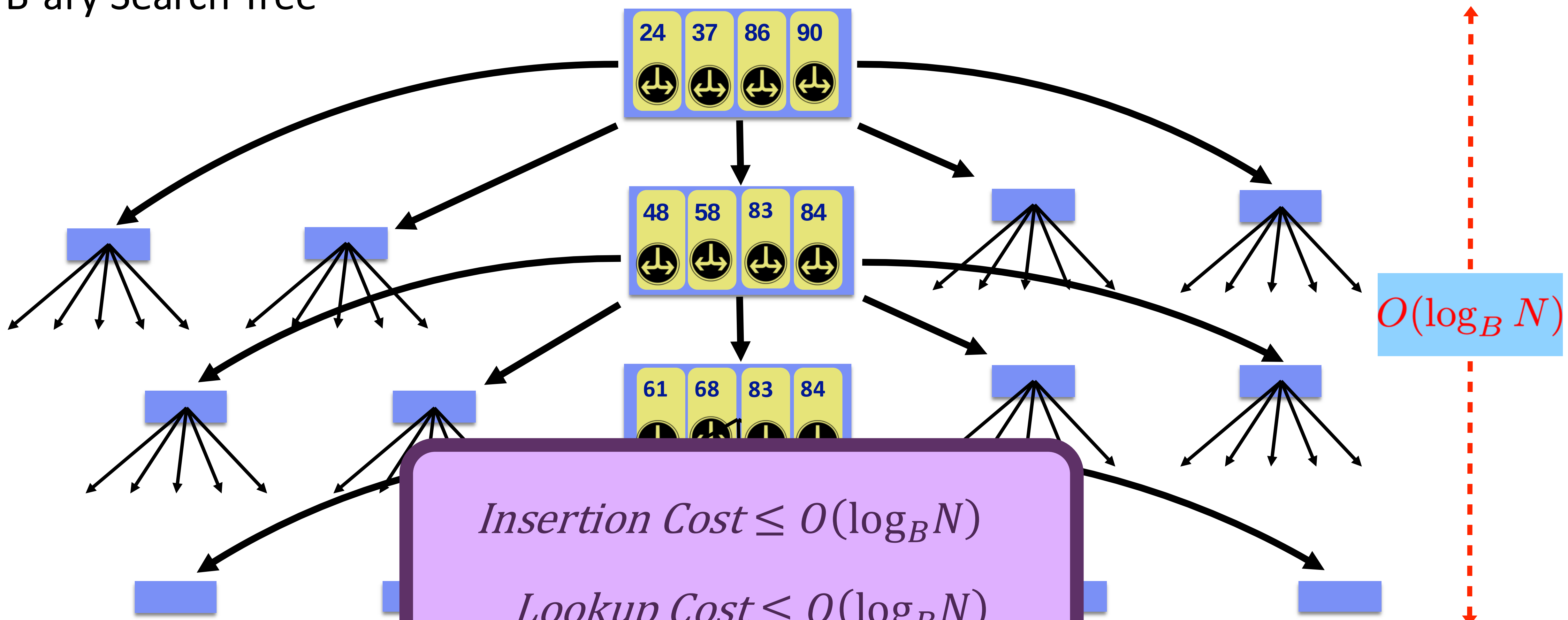
B-ary Search Tree



B-Trees

Insert

B-ary Search Tree



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

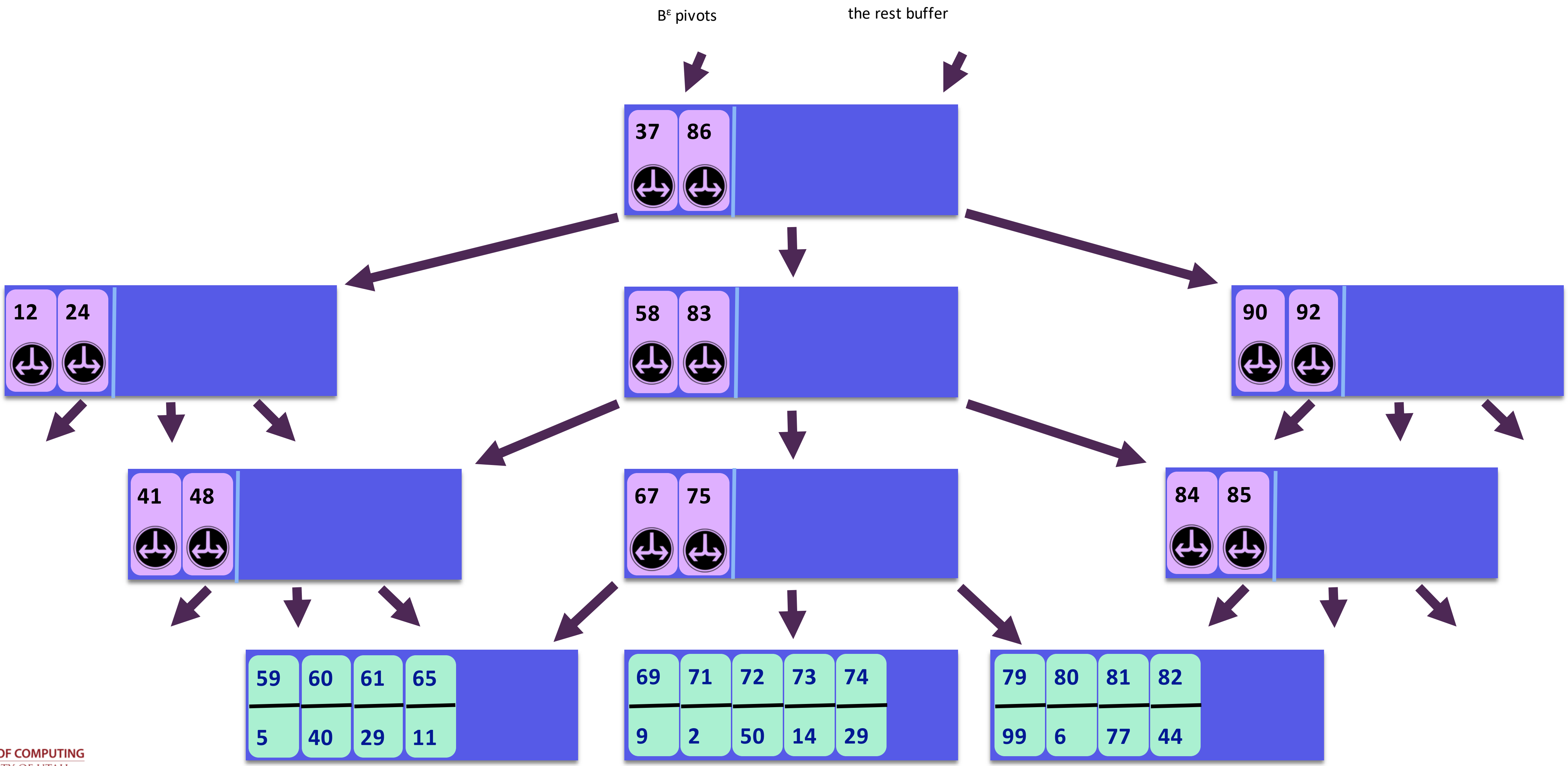
$$Lookup\ Cost \leq O(\log_B N)$$

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B^{ε}-Trees

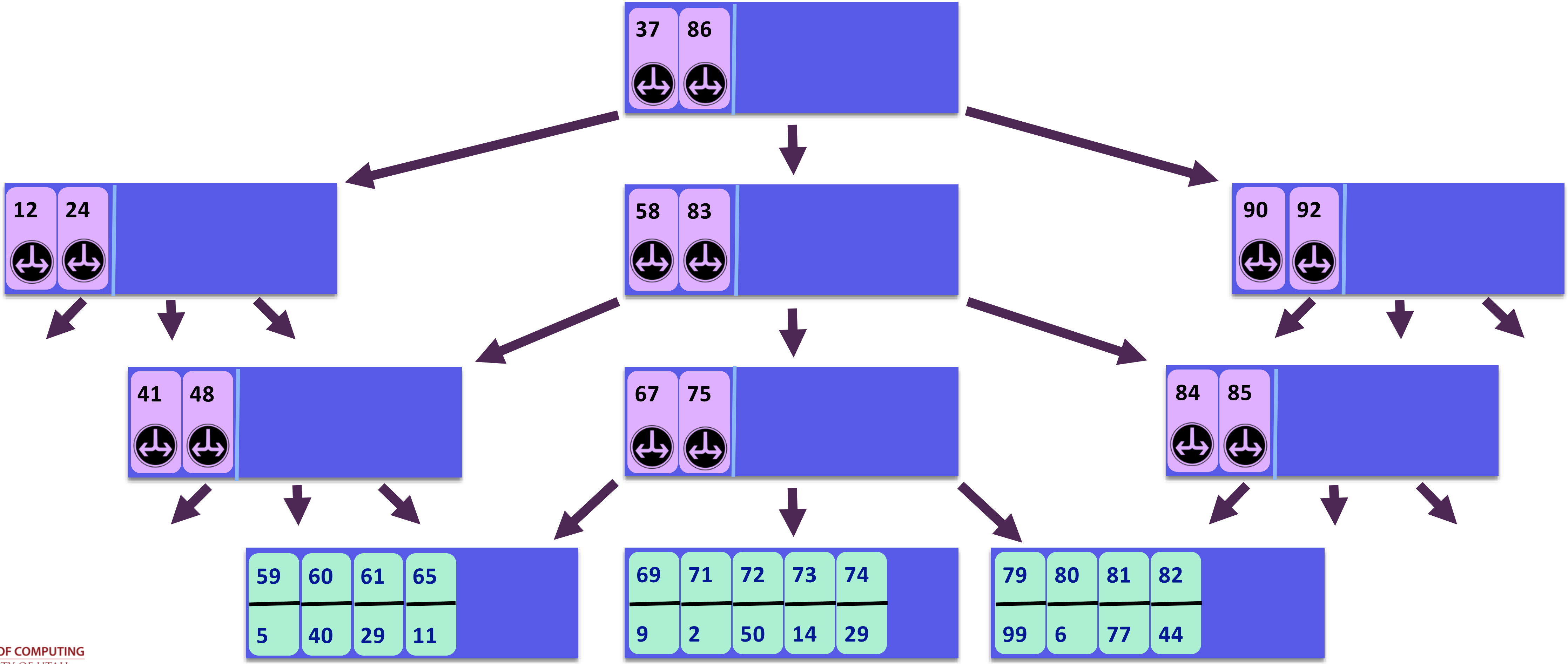
B^{ε}-Trees

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



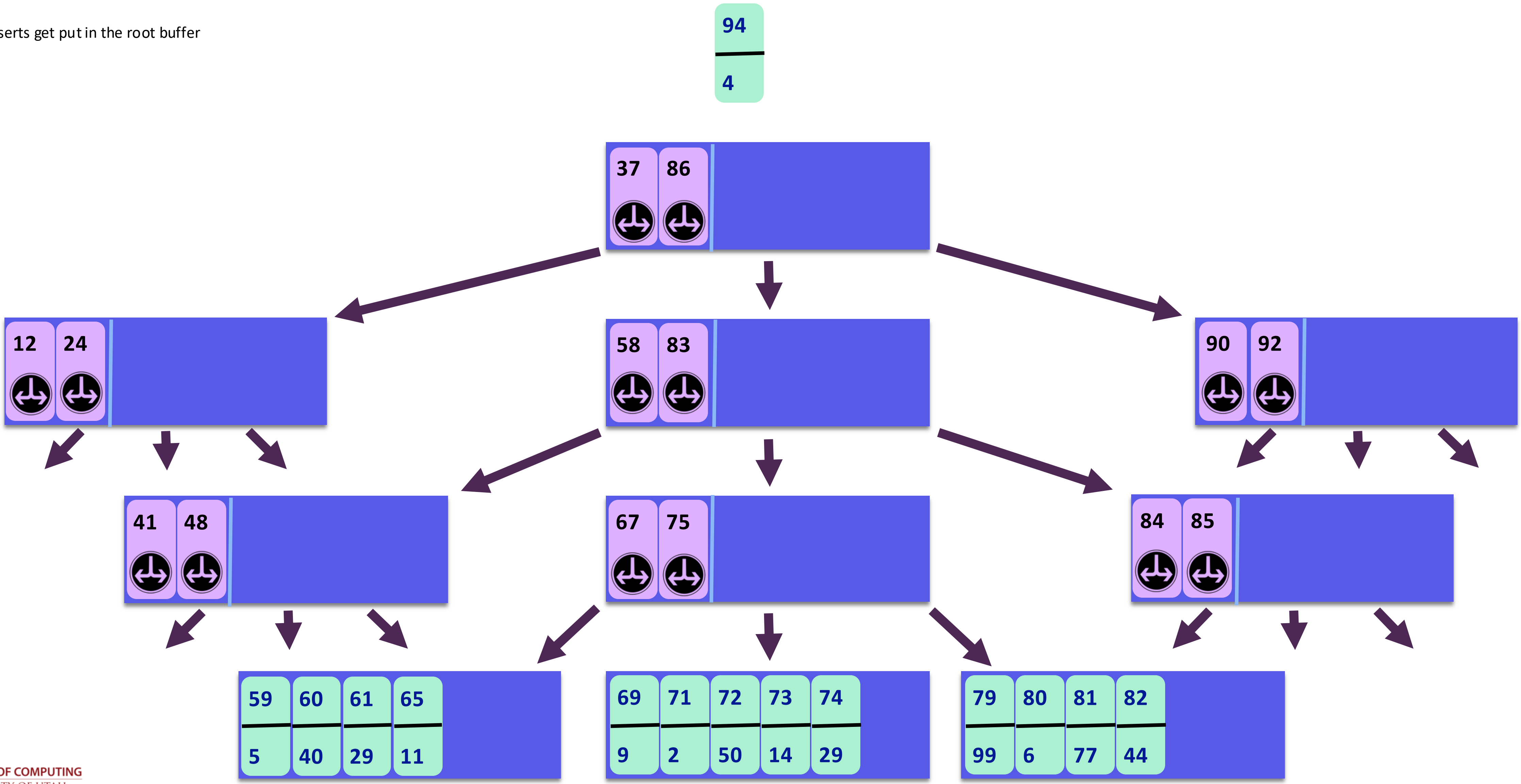
B^{ε}-Trees

Inserts get put in the root buffer



B$^{\varepsilon}$-Trees

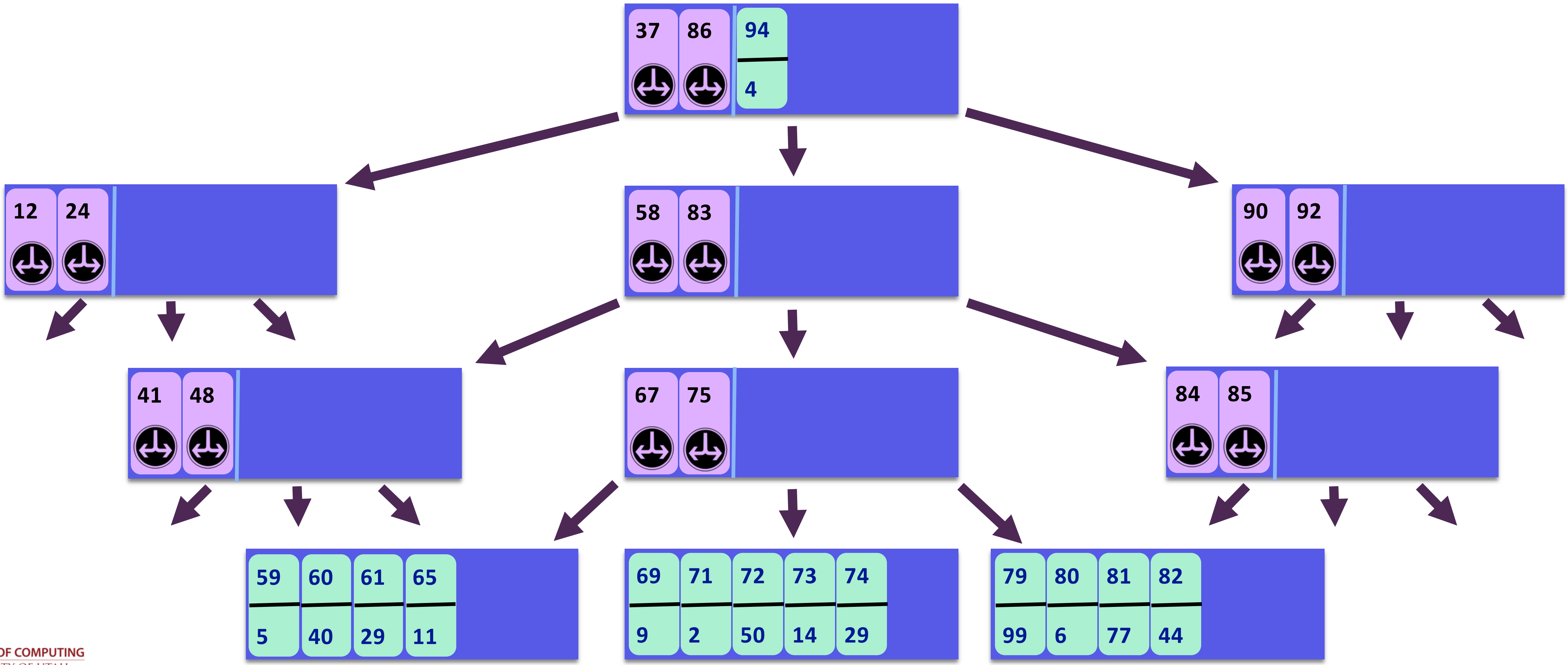
Inserts get put in the root buffer



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

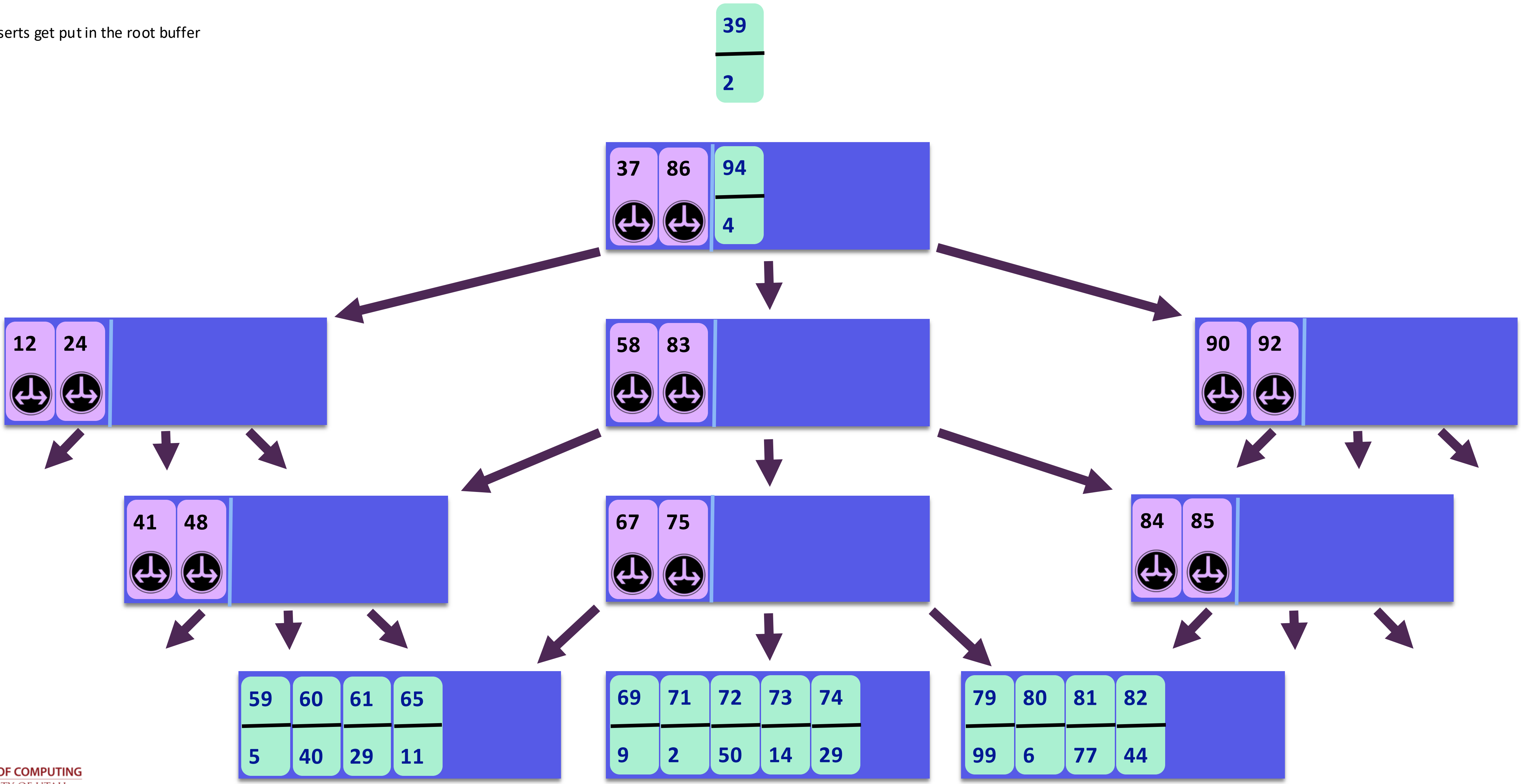
Bε-Trees

Inserts get put in the root buffer



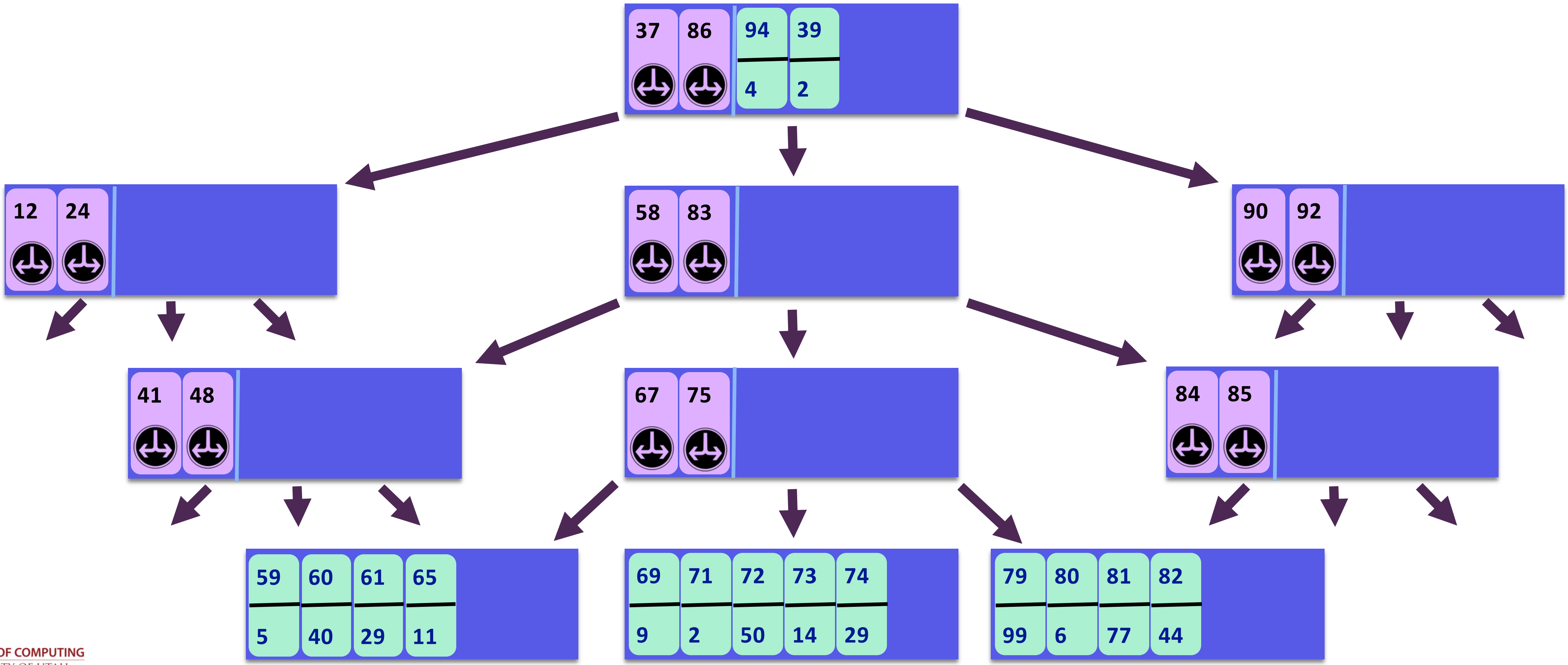
Bε-Trees

Inserts get put in the root buffer



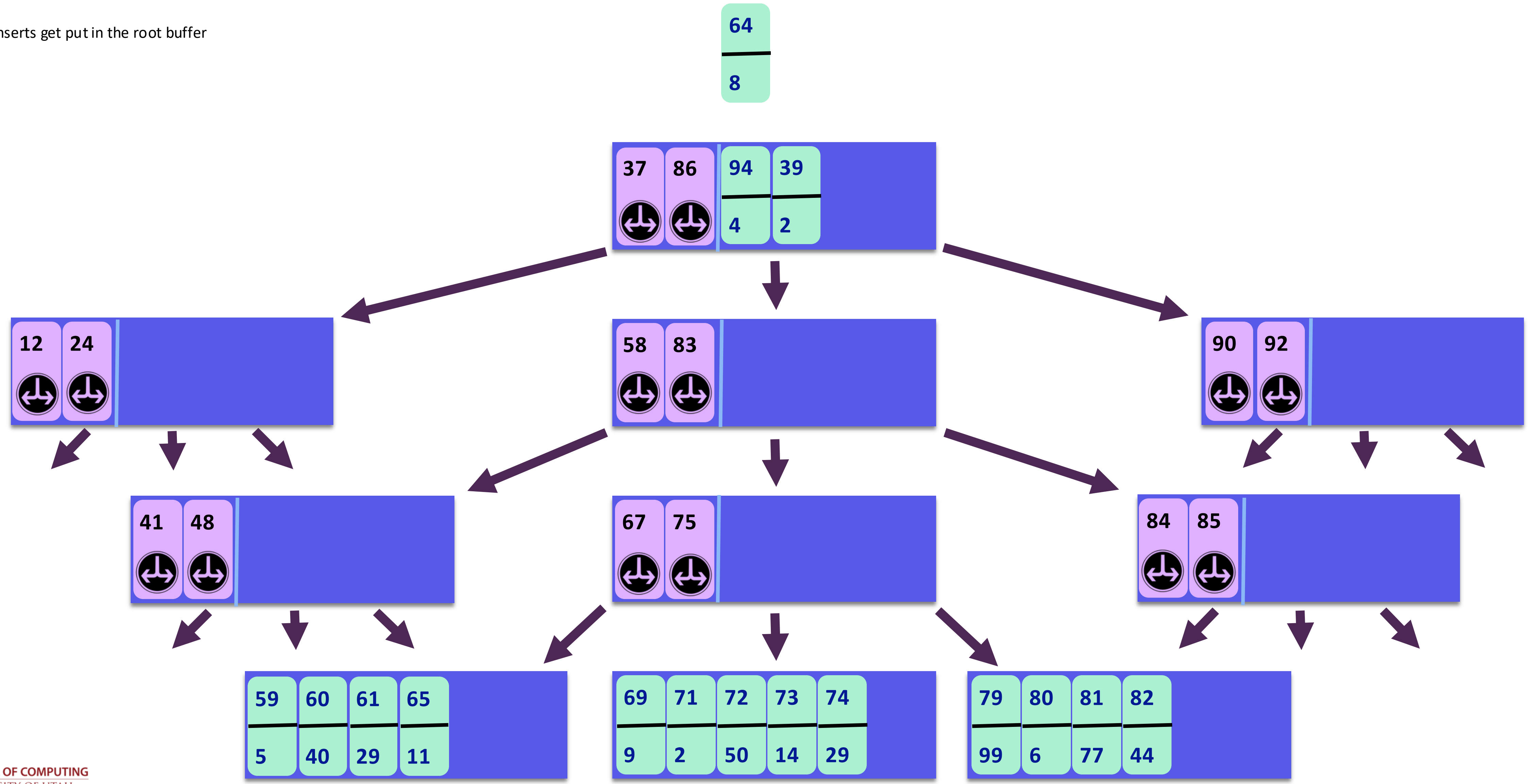
B^{ε}-Trees

Inserts get put in the root buffer



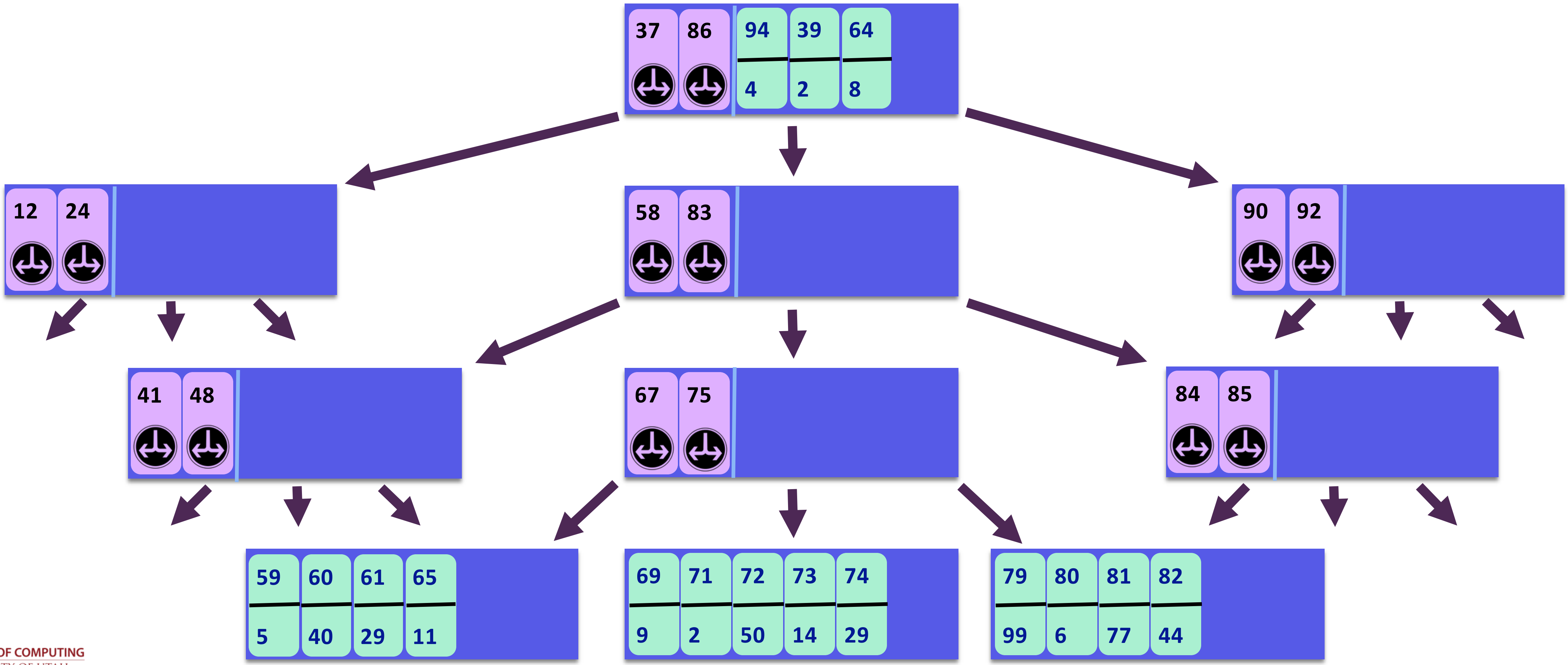
B^{ε}-Trees

Inserts get put in the root buffer



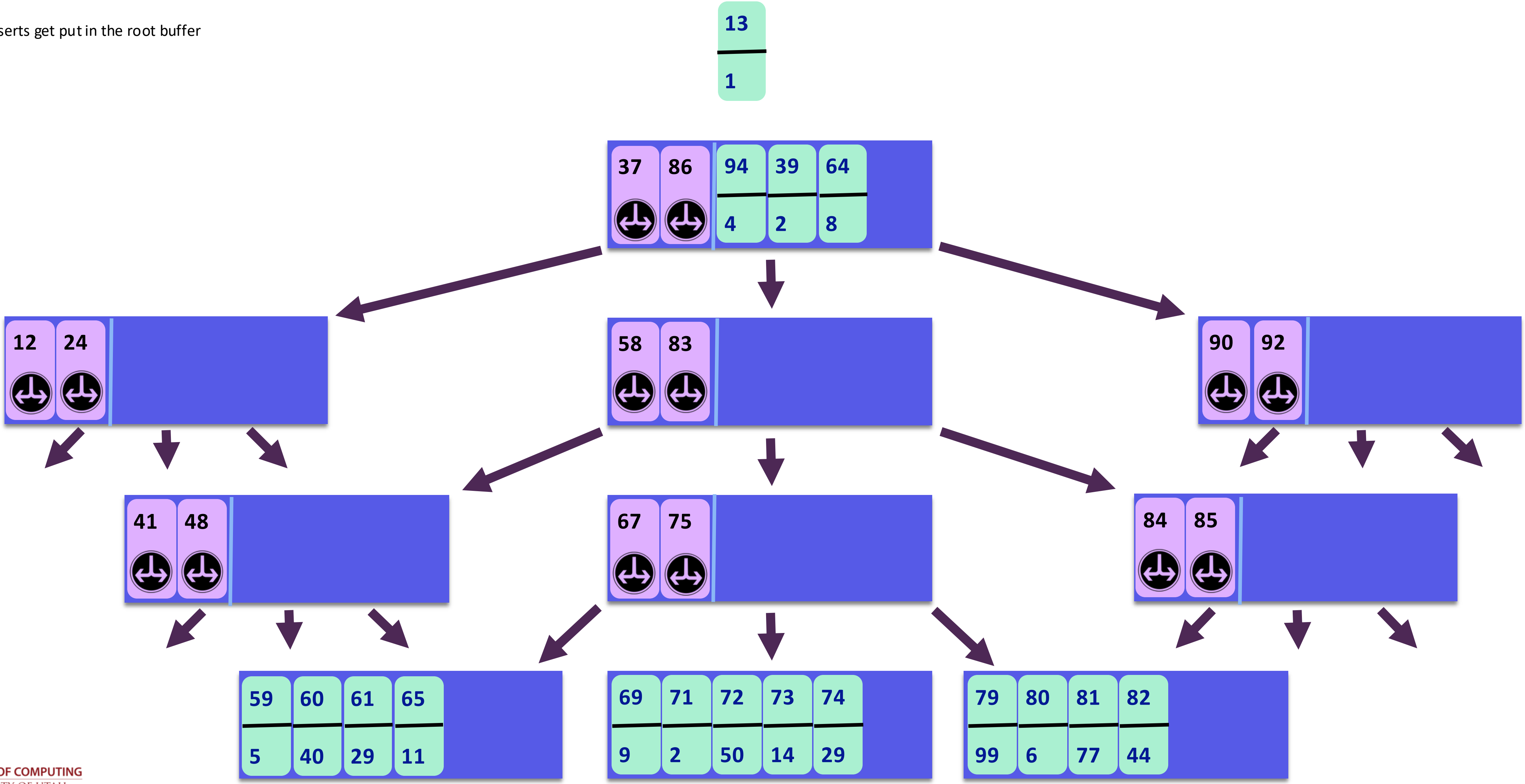
B^{ε}-Trees

Inserts get put in the root buffer



B^{ε}-Trees

Inserts get put in the root buffer

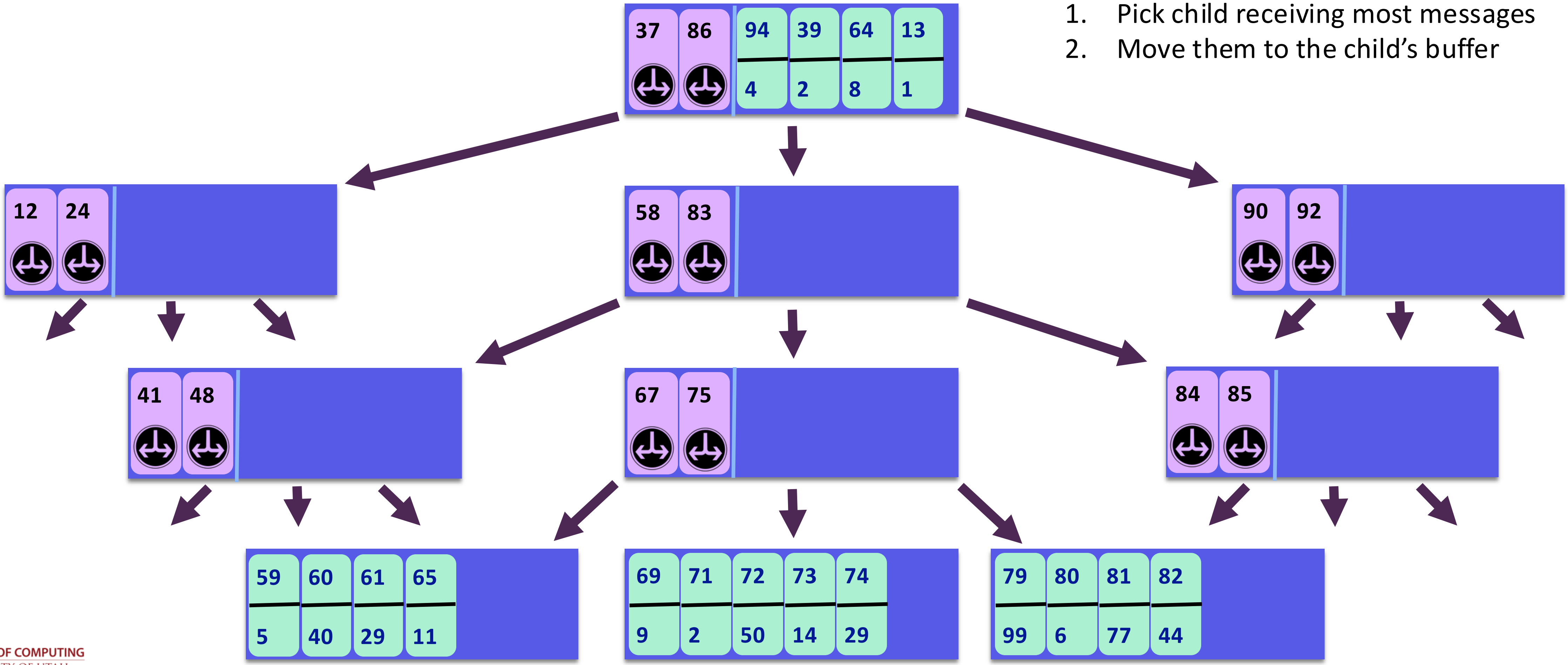


B$^{\varepsilon}$-Trees

Inserts get put in the root buffer

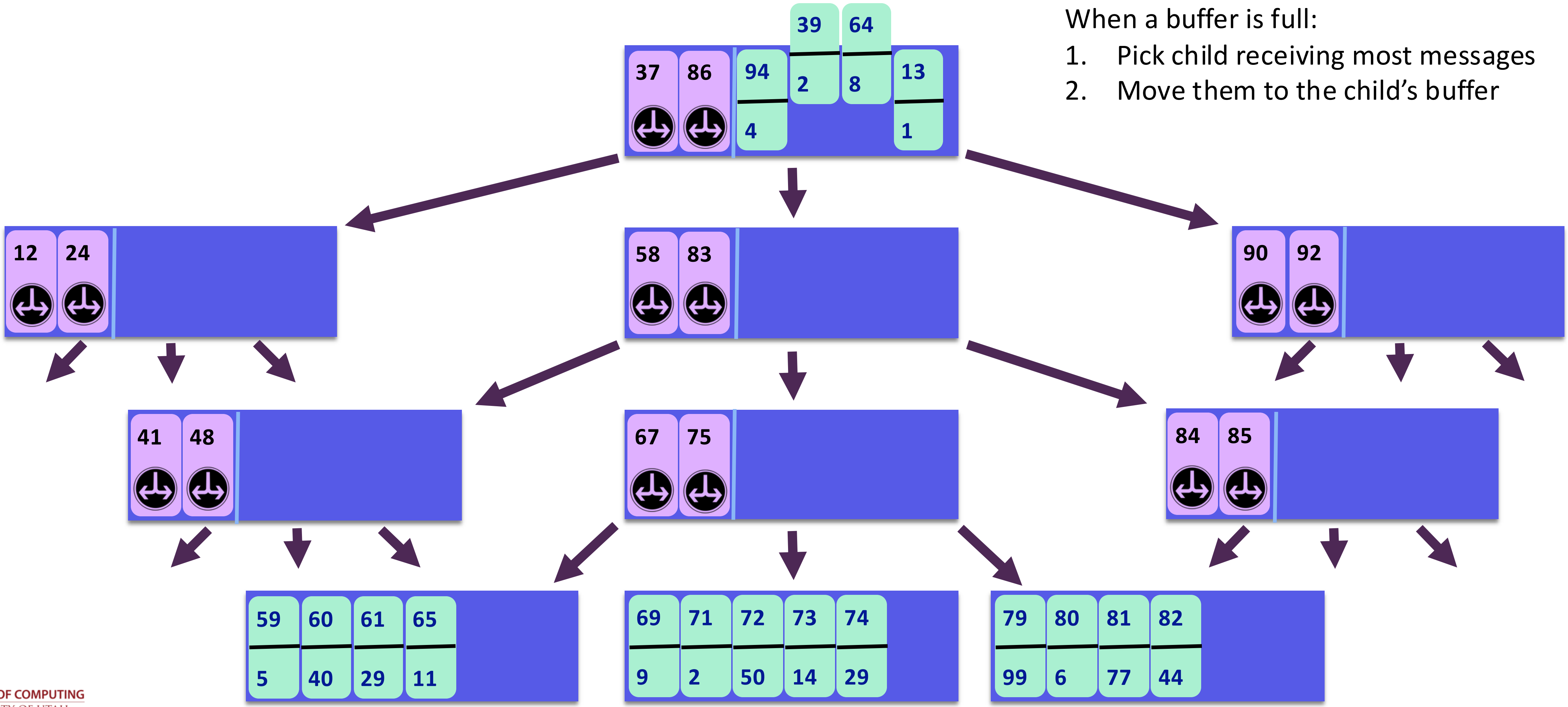
When a buffer is full:

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



B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



When a buffer is full:

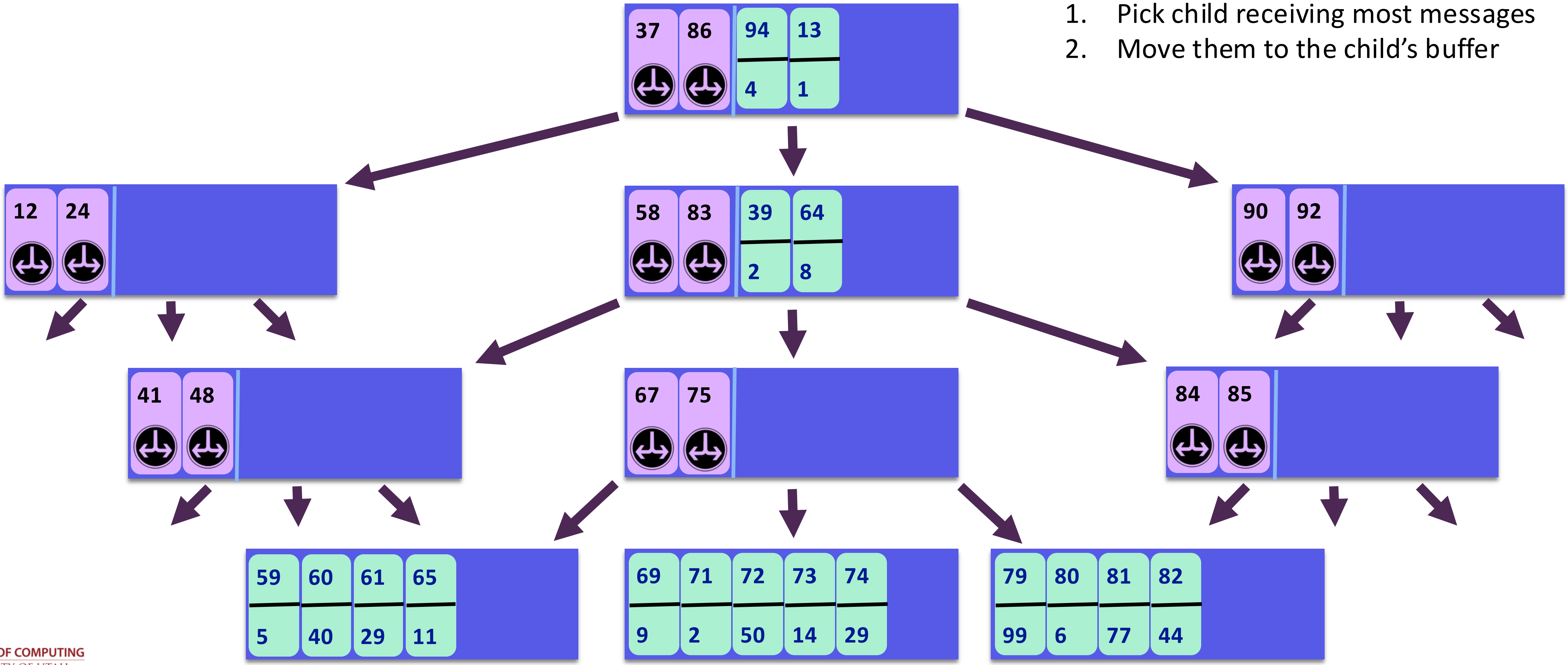
1. Pick child receiving most messages
2. Move them to the child's buffer

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

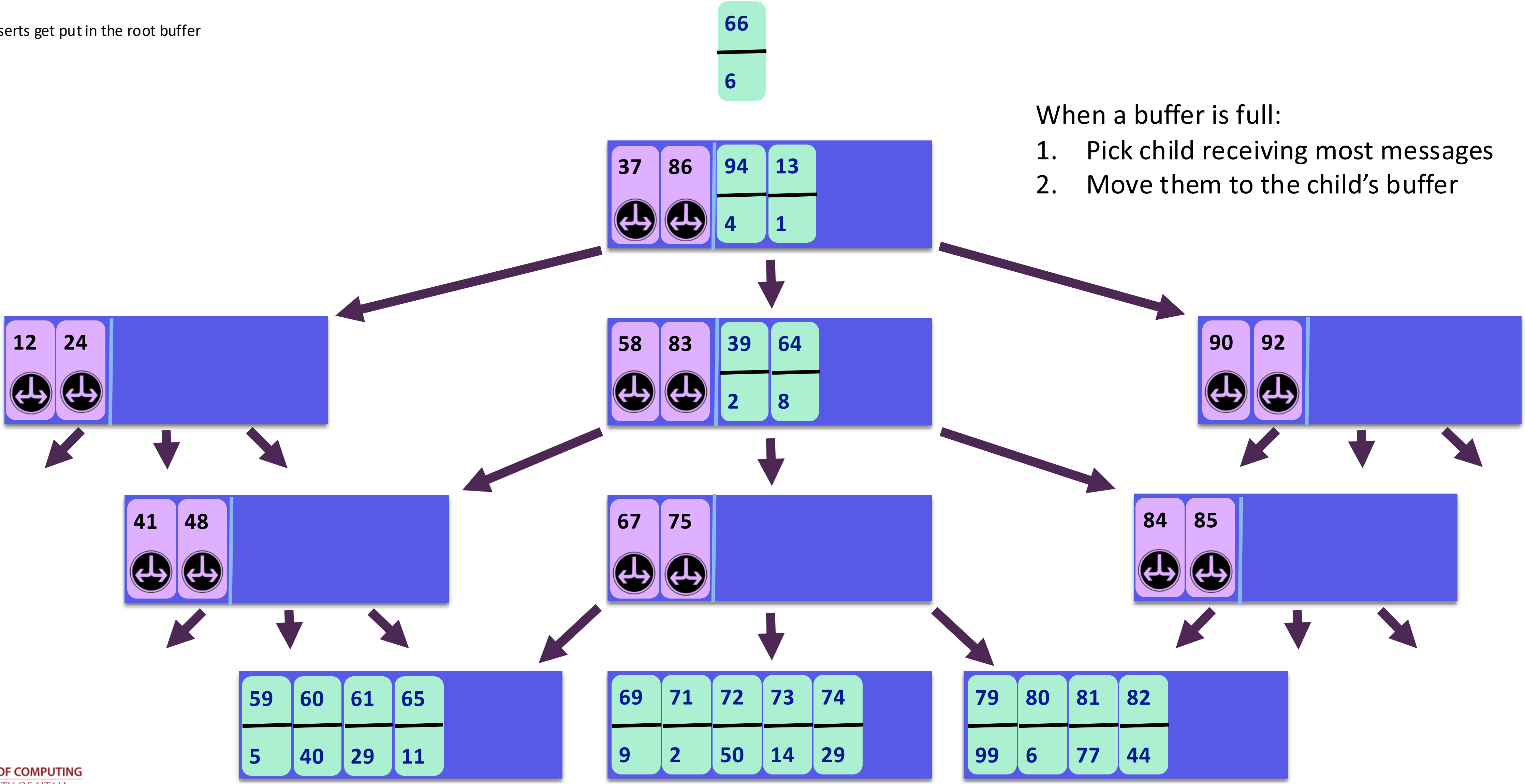


B^{ε}-Trees

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



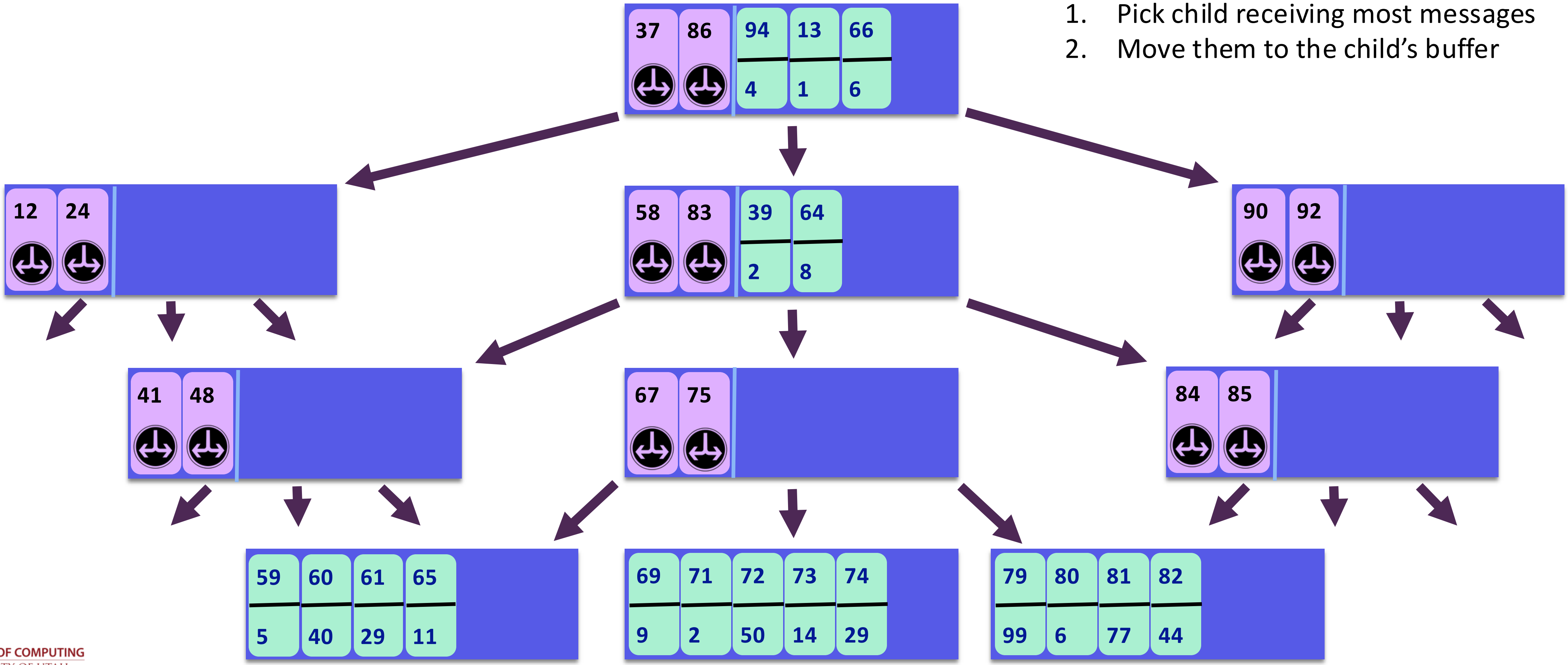
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

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



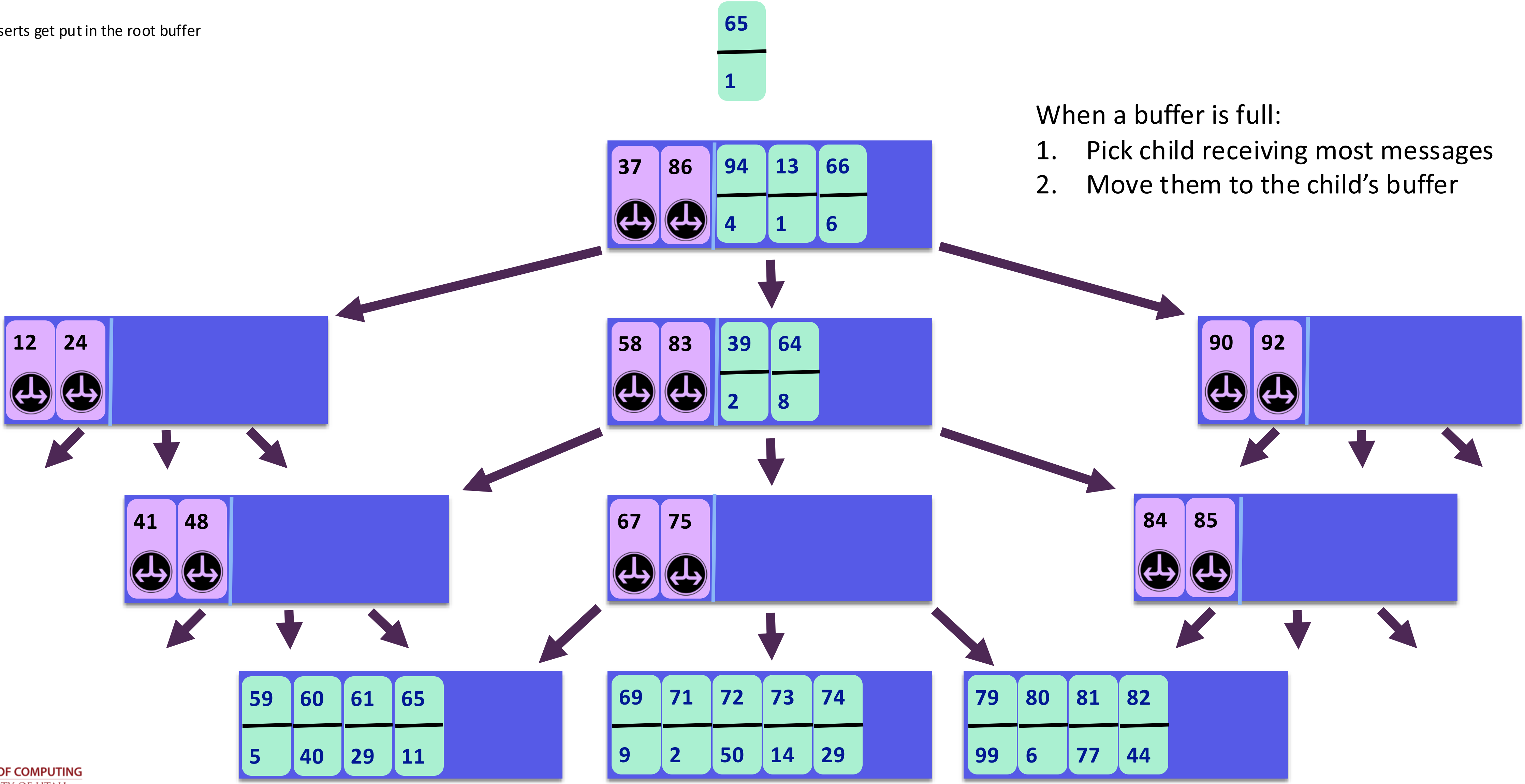
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

B^{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

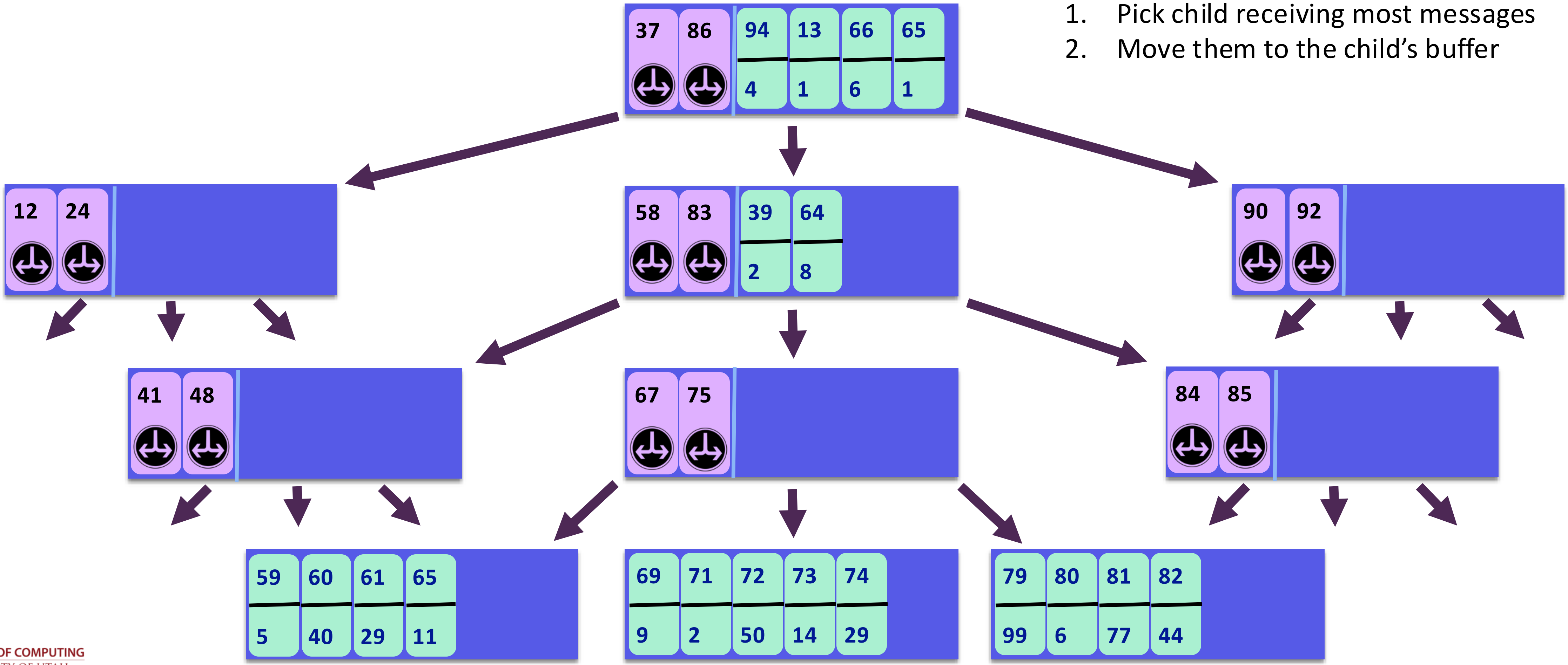


Bε-Trees

Inserts get put in the root buffer

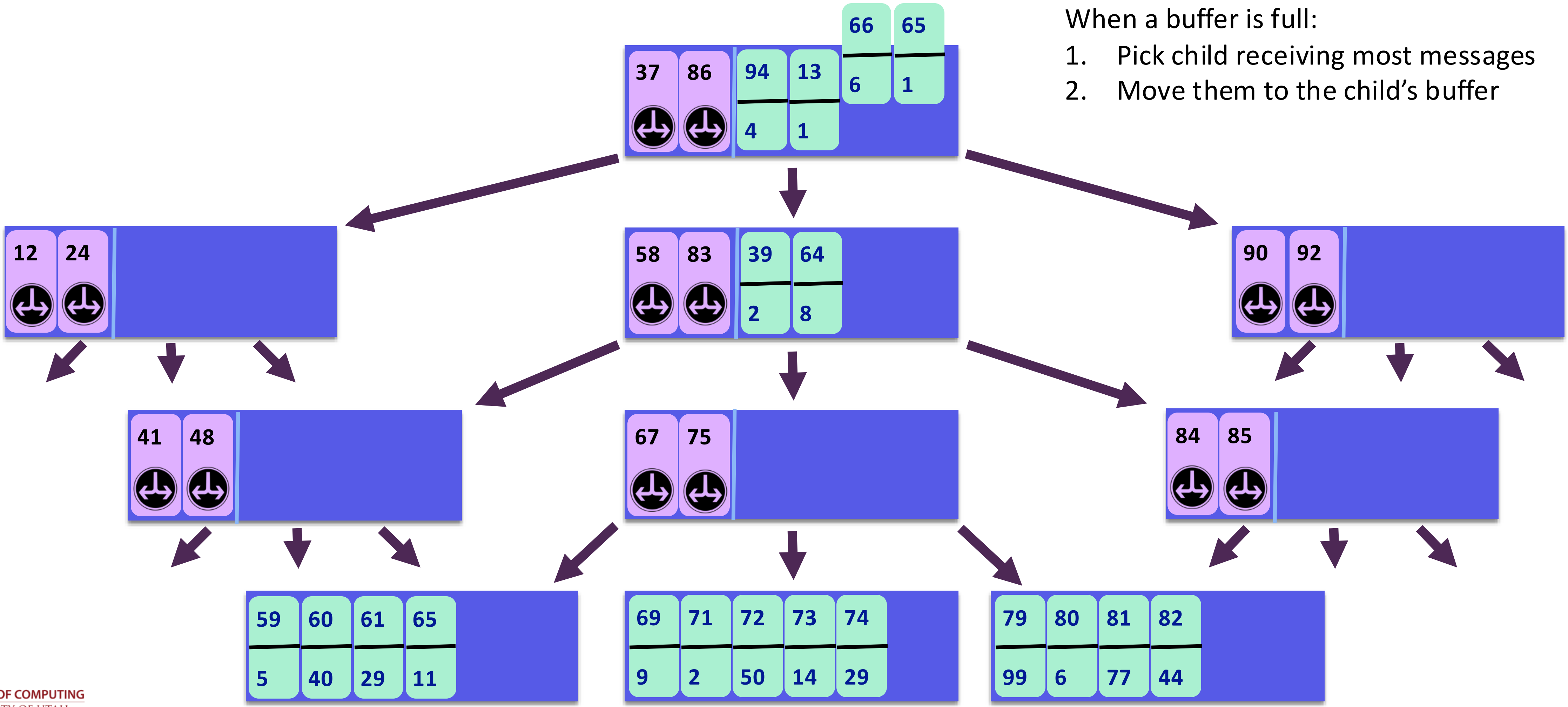
When a buffer is full:

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



B$^{\varepsilon}$-Trees

Inserts get put in the root buffer



When a buffer is full:

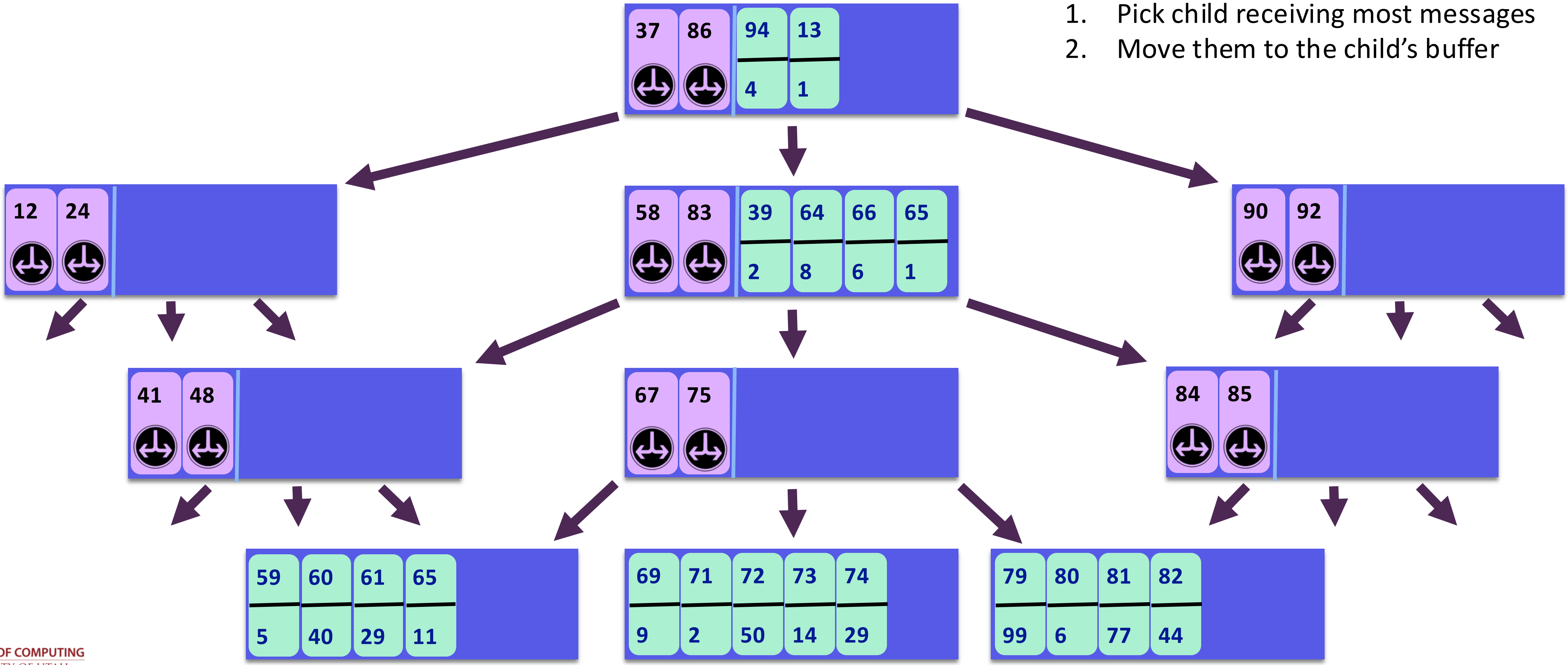
1. Pick child receiving most messages
2. Move them to the child's buffer

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

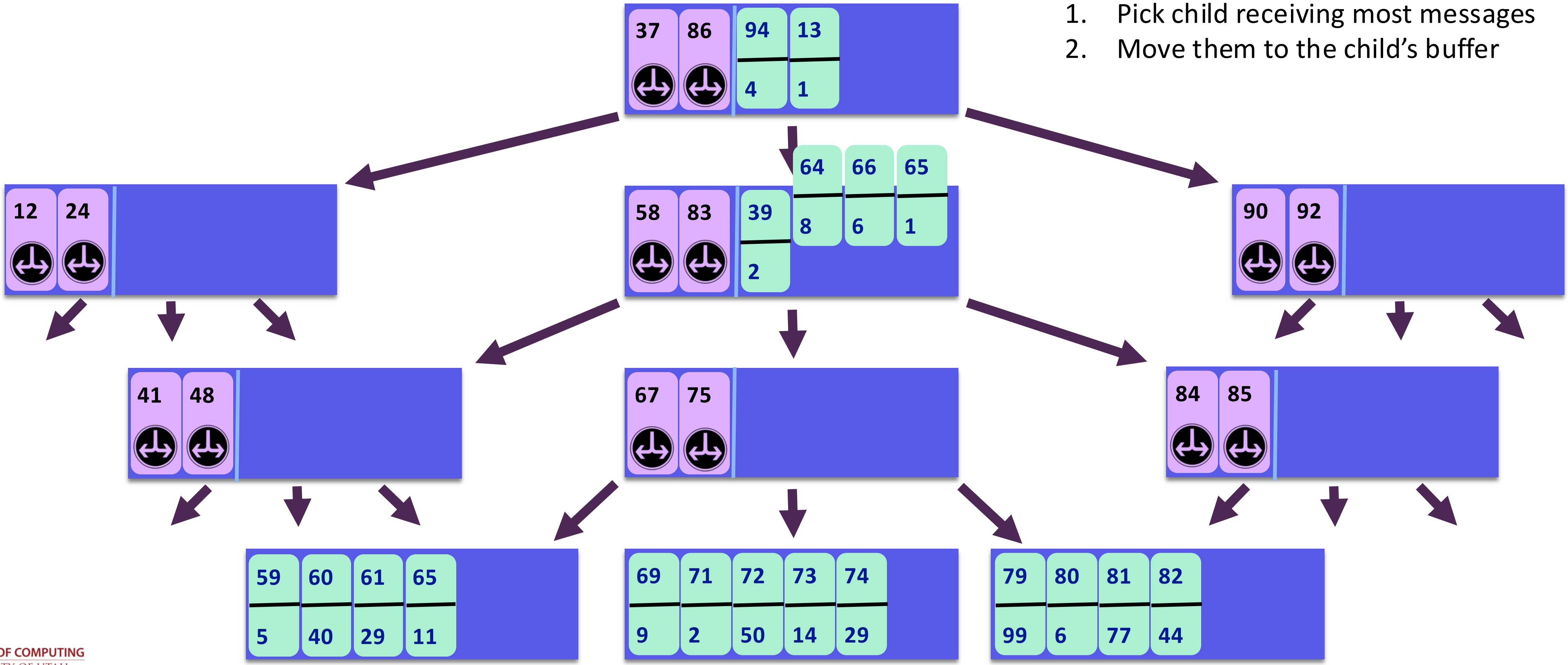


B$^{\varepsilon}$-Trees

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

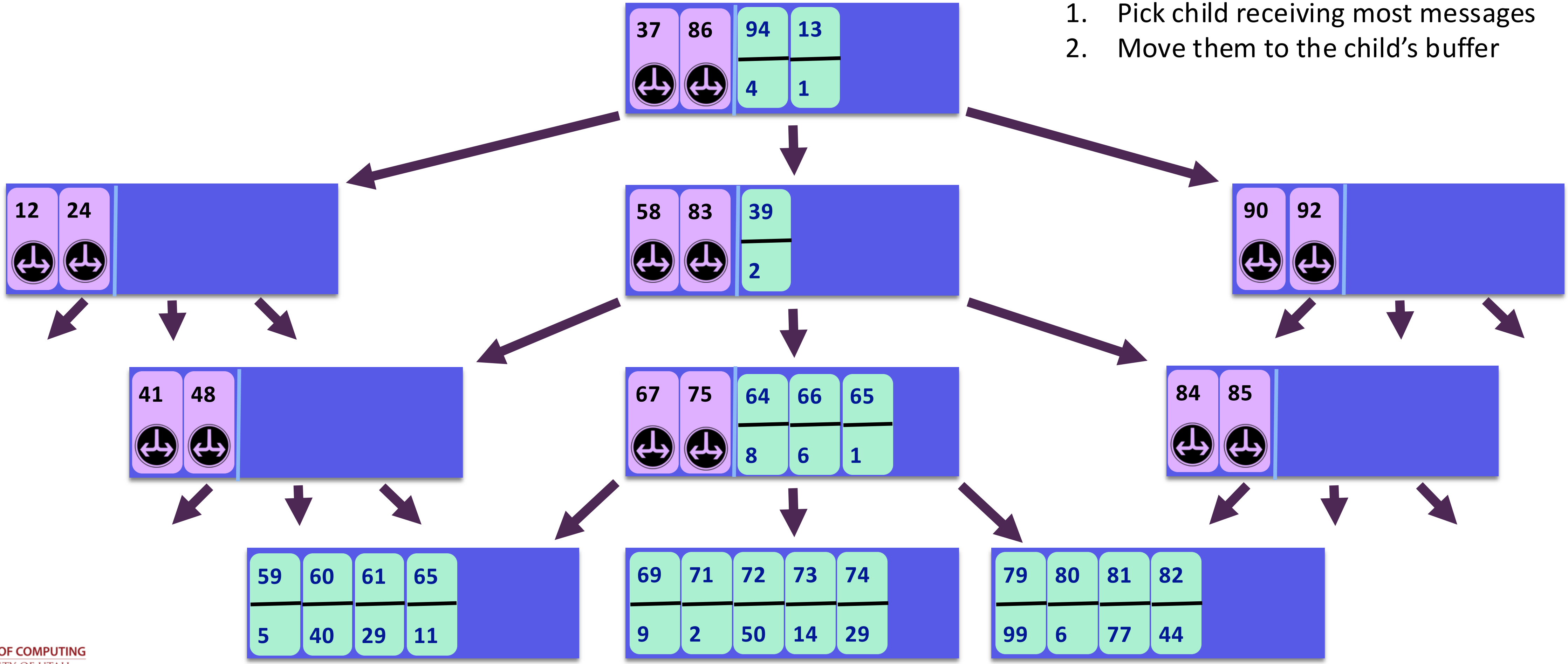


B^{ε}-Trees

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



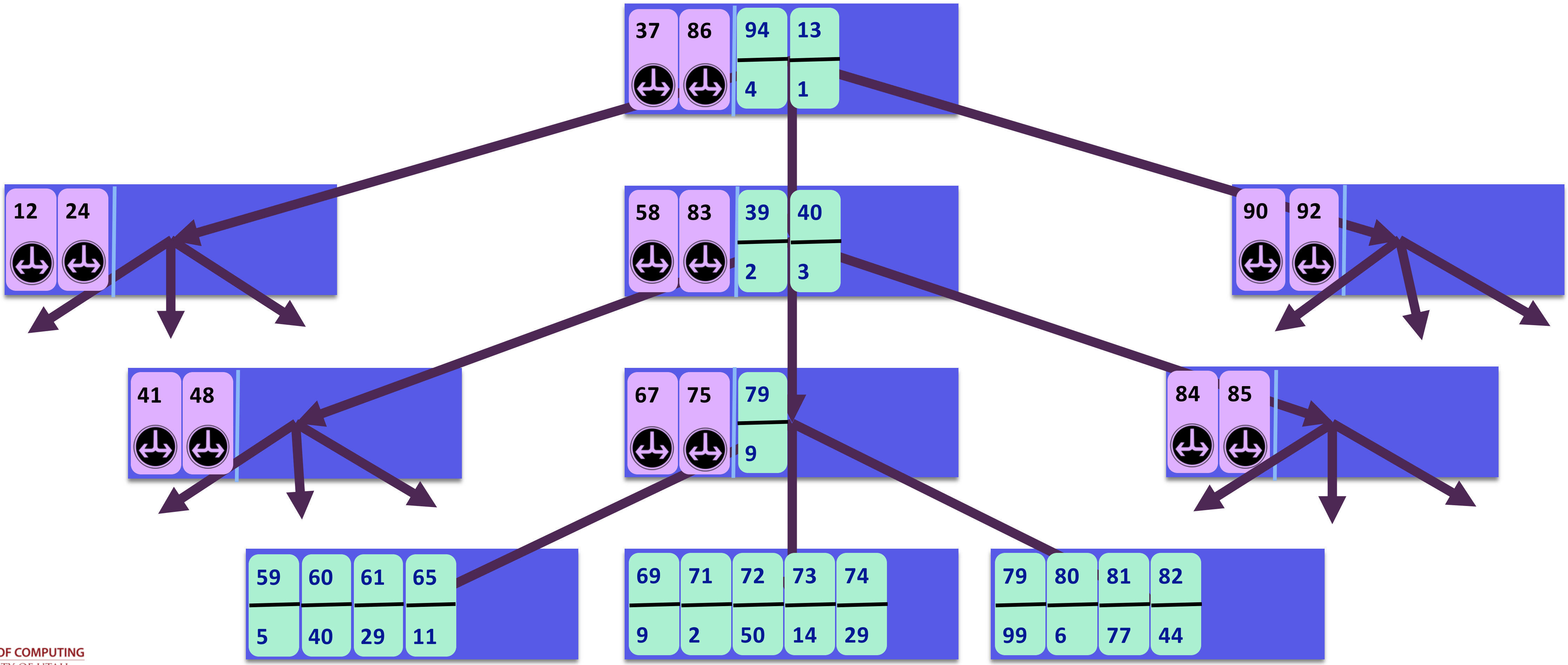
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Lookups in B^{ε}-Trees

Bε-Trees

Query(71)

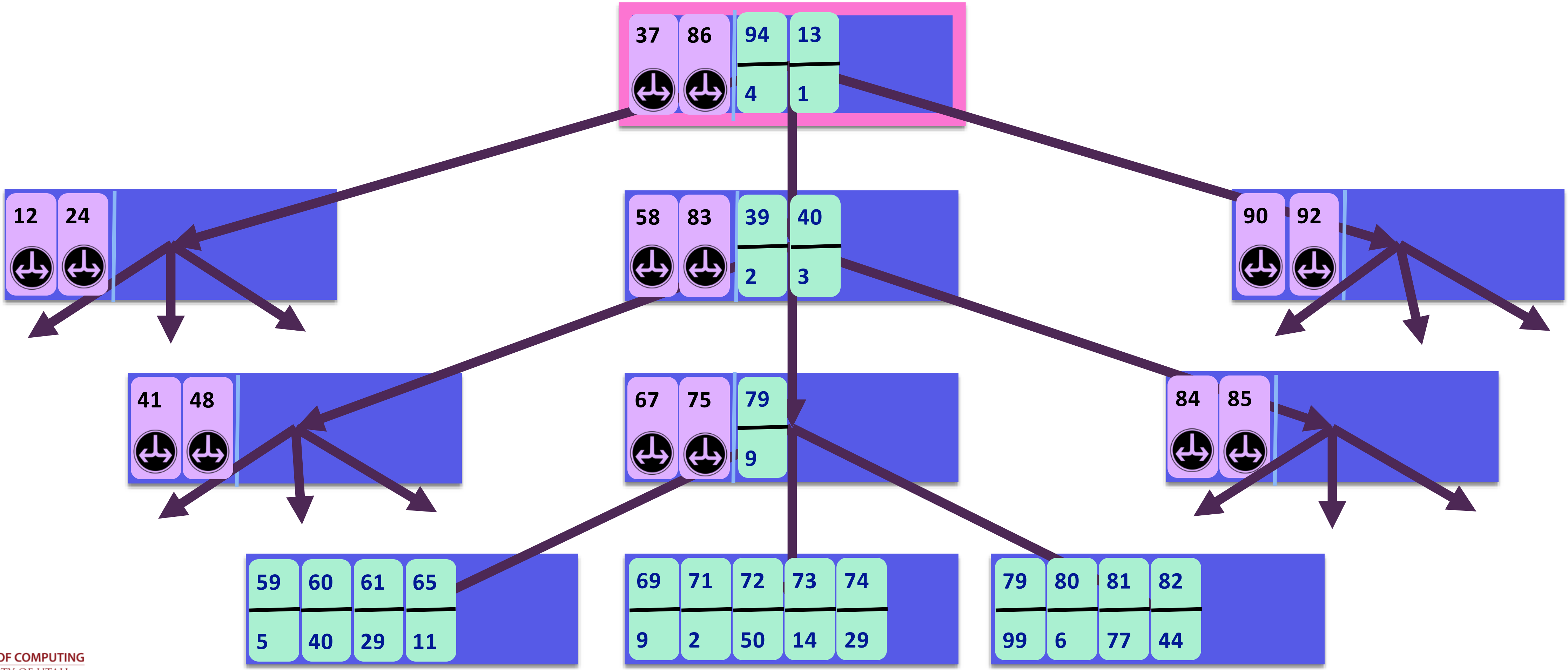
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Query(71)

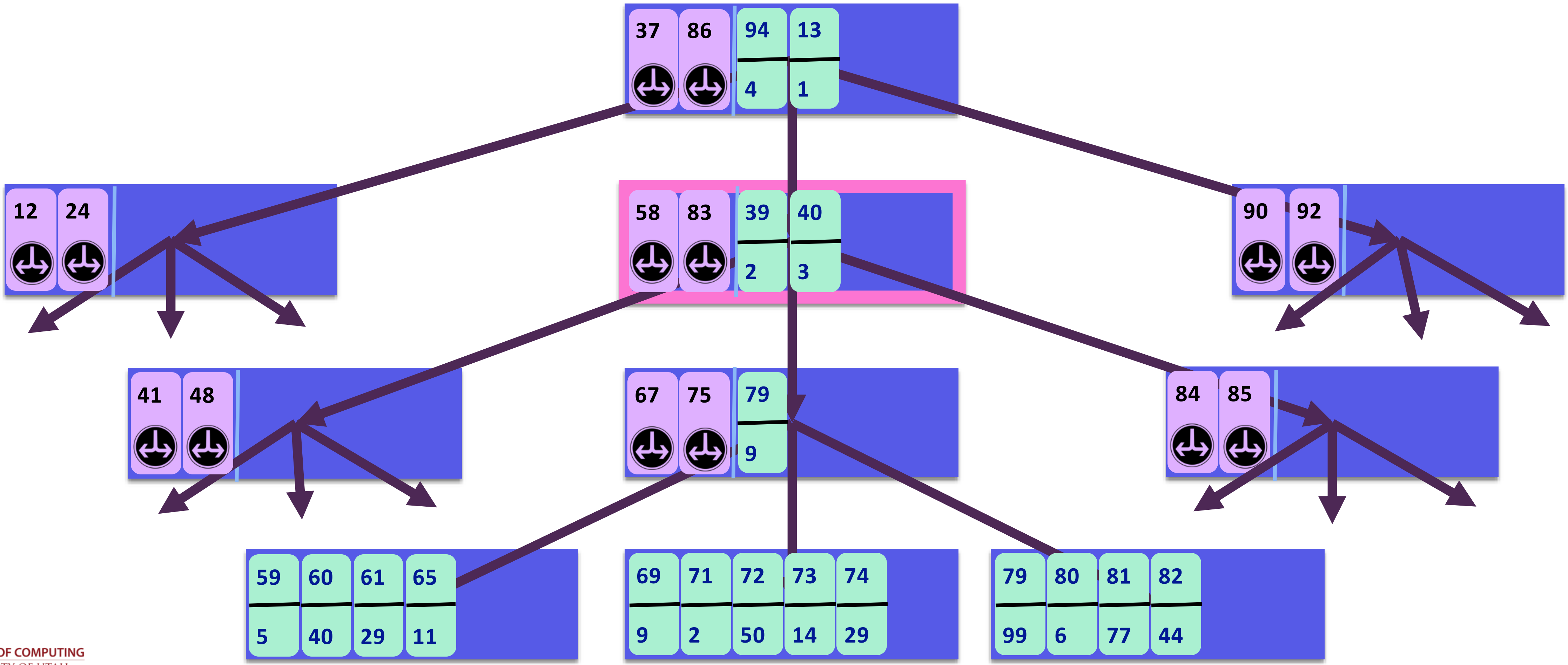
Lookups follow pivots, but check buffers along the way



B$^{\varepsilon}$-Trees

Query(71)

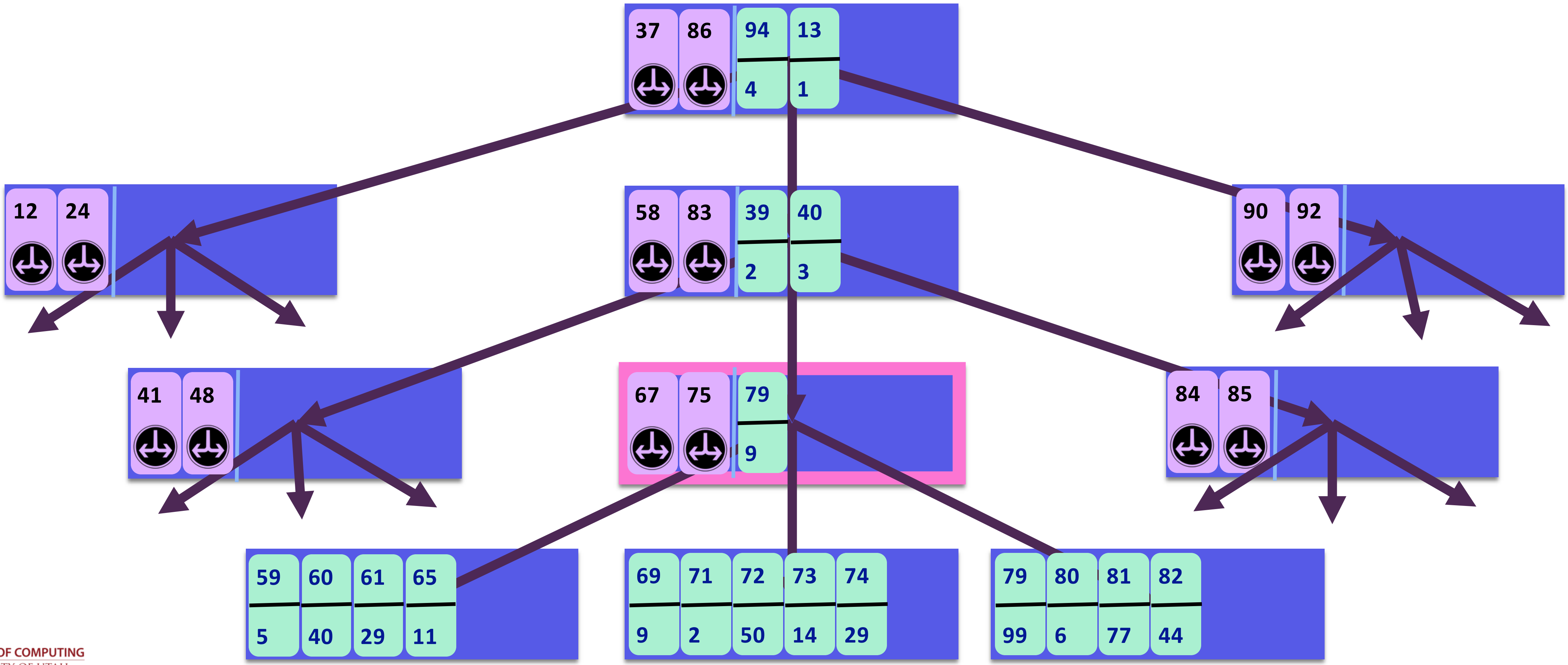
Lookups follow pivots, but check buffers along the way



B^{ε}-Trees

Query(71)

Lookups follow pivots, but check buffers along the way

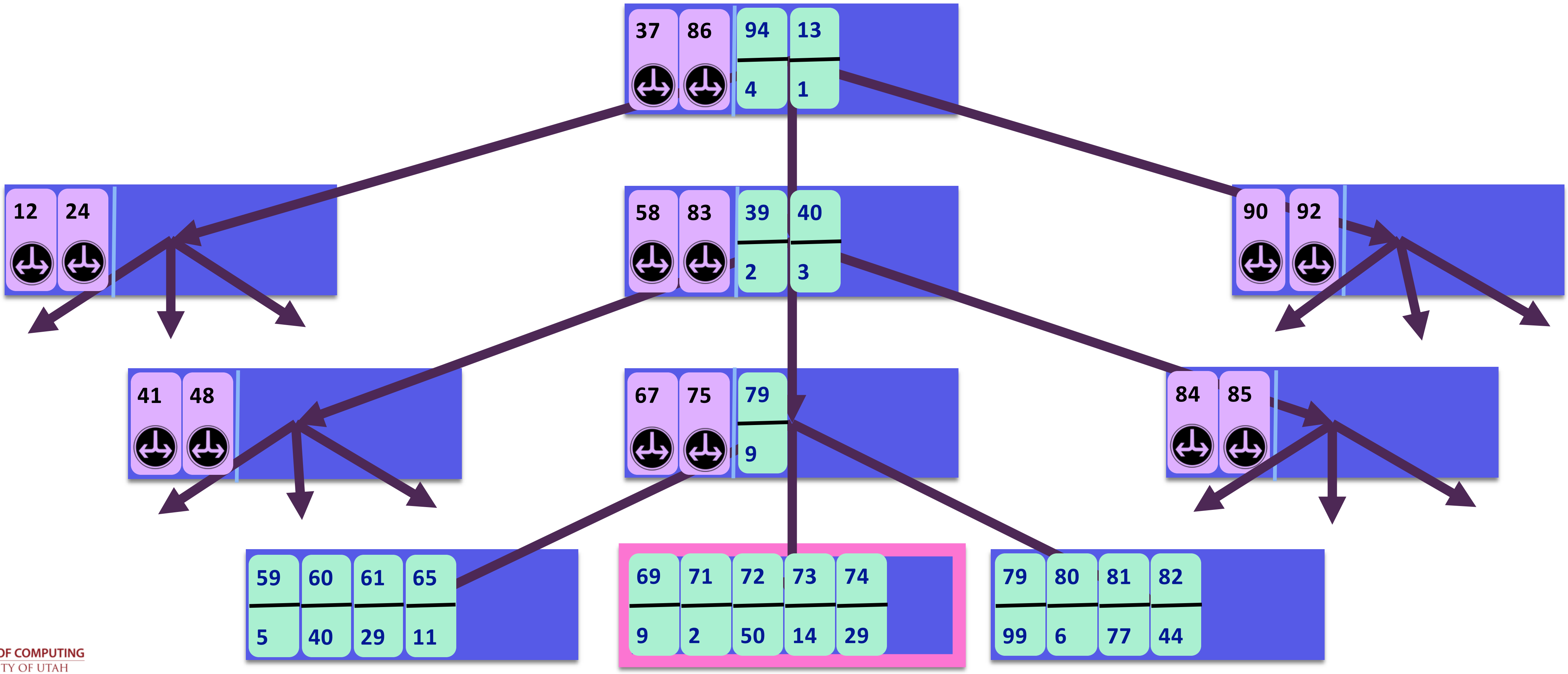


SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Bε-Trees

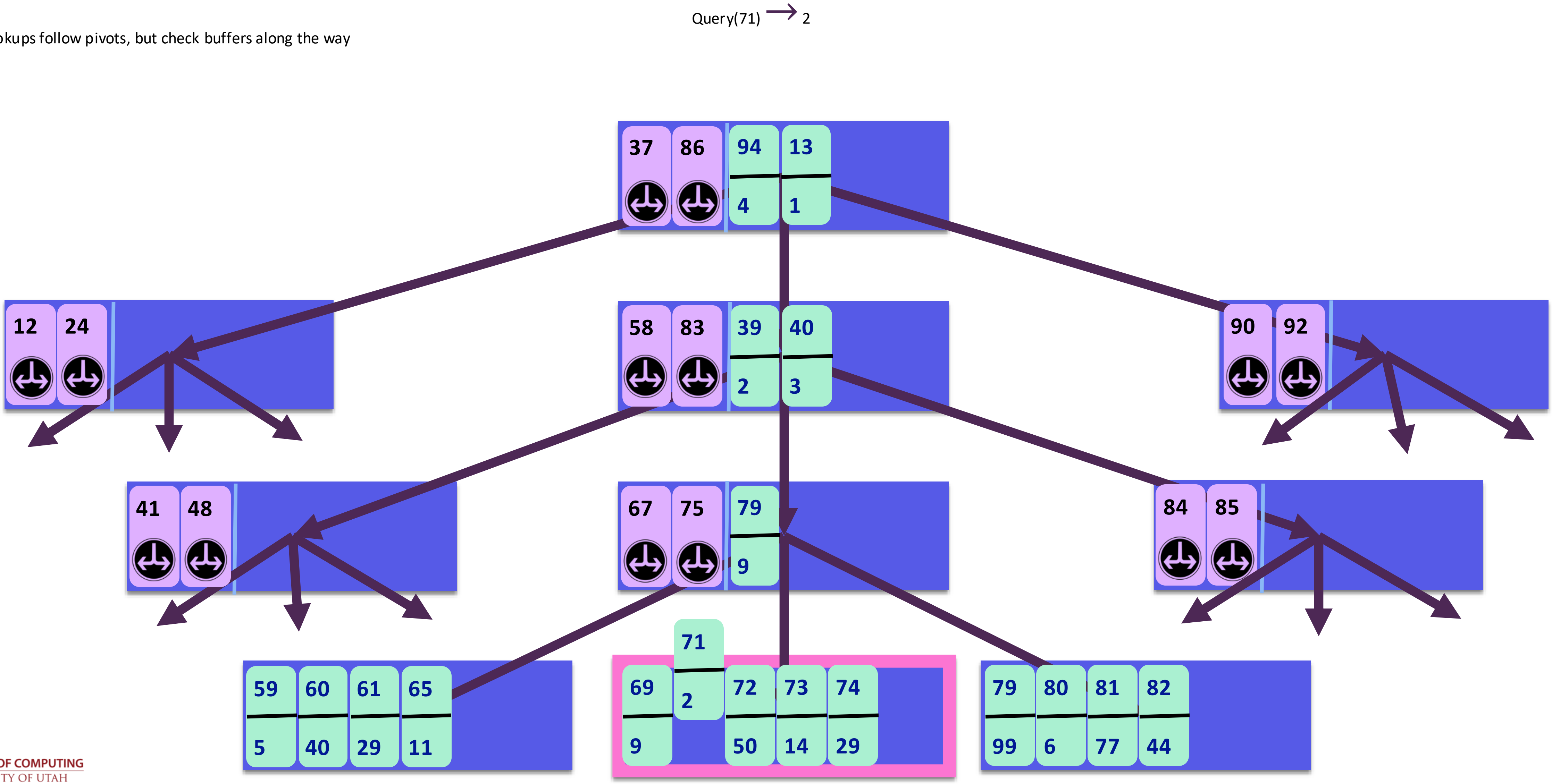
Lookups follow pivots, but check buffers along the way

Query(71) → 2



Bε-Trees

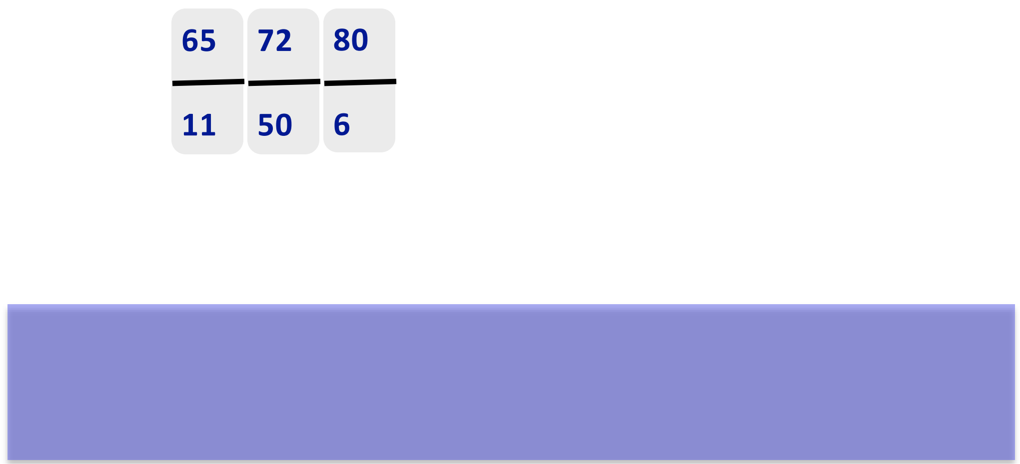
Lookups follow pivots, but check buffers along the way



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

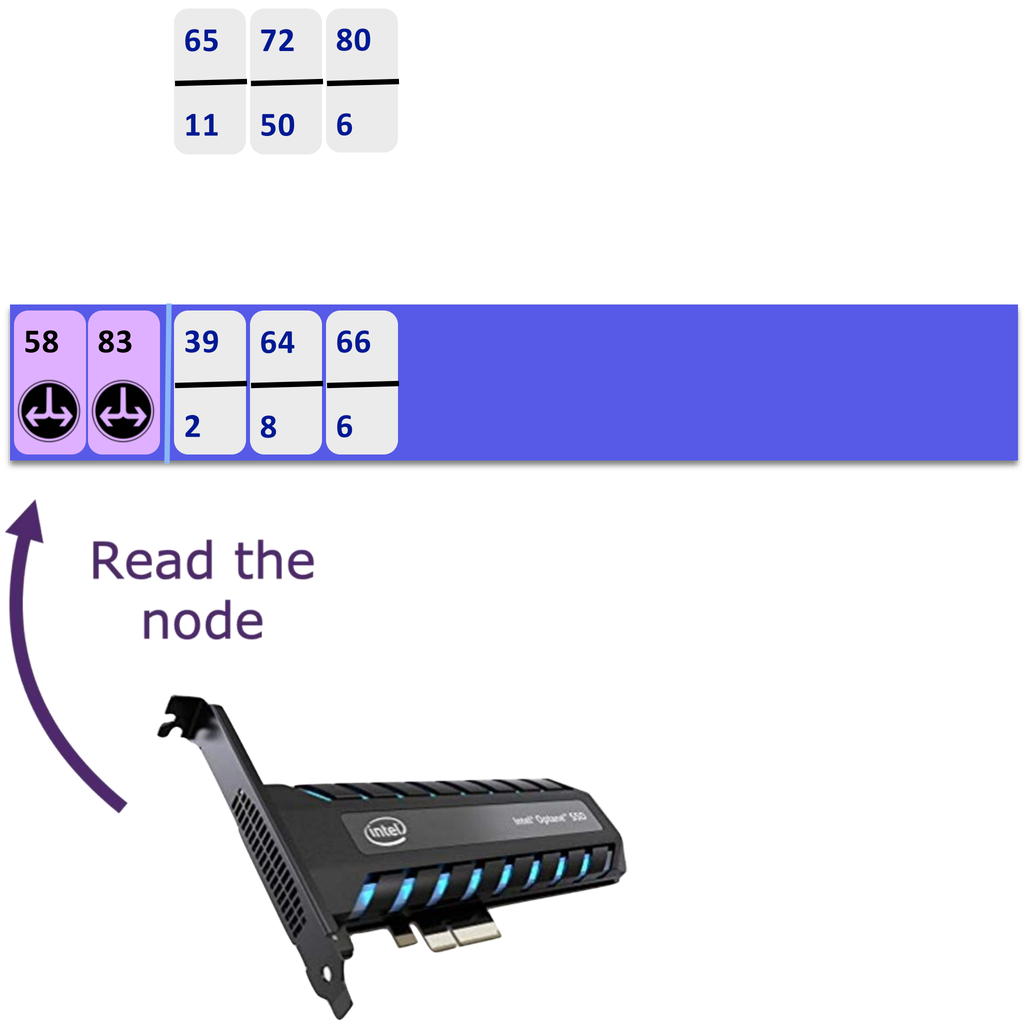
Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

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



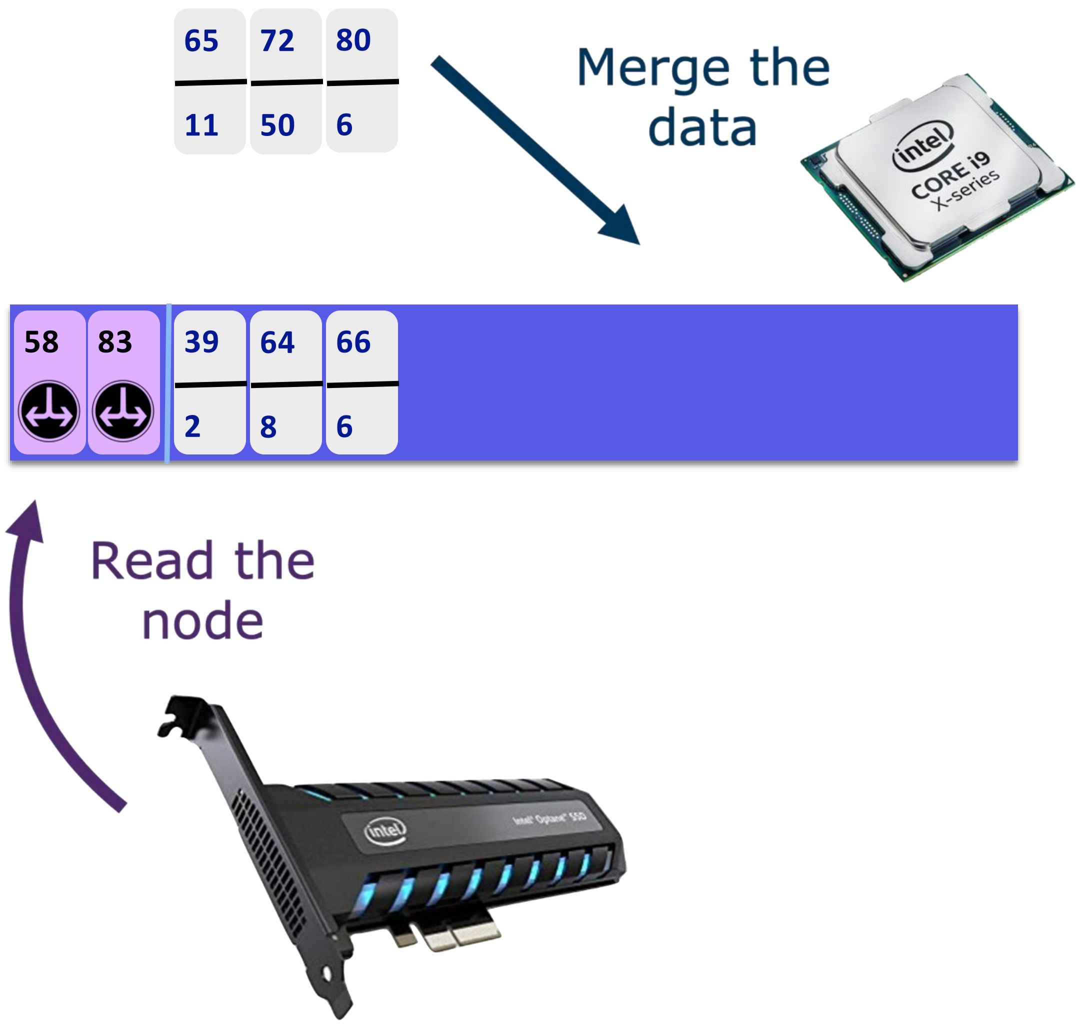
Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

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



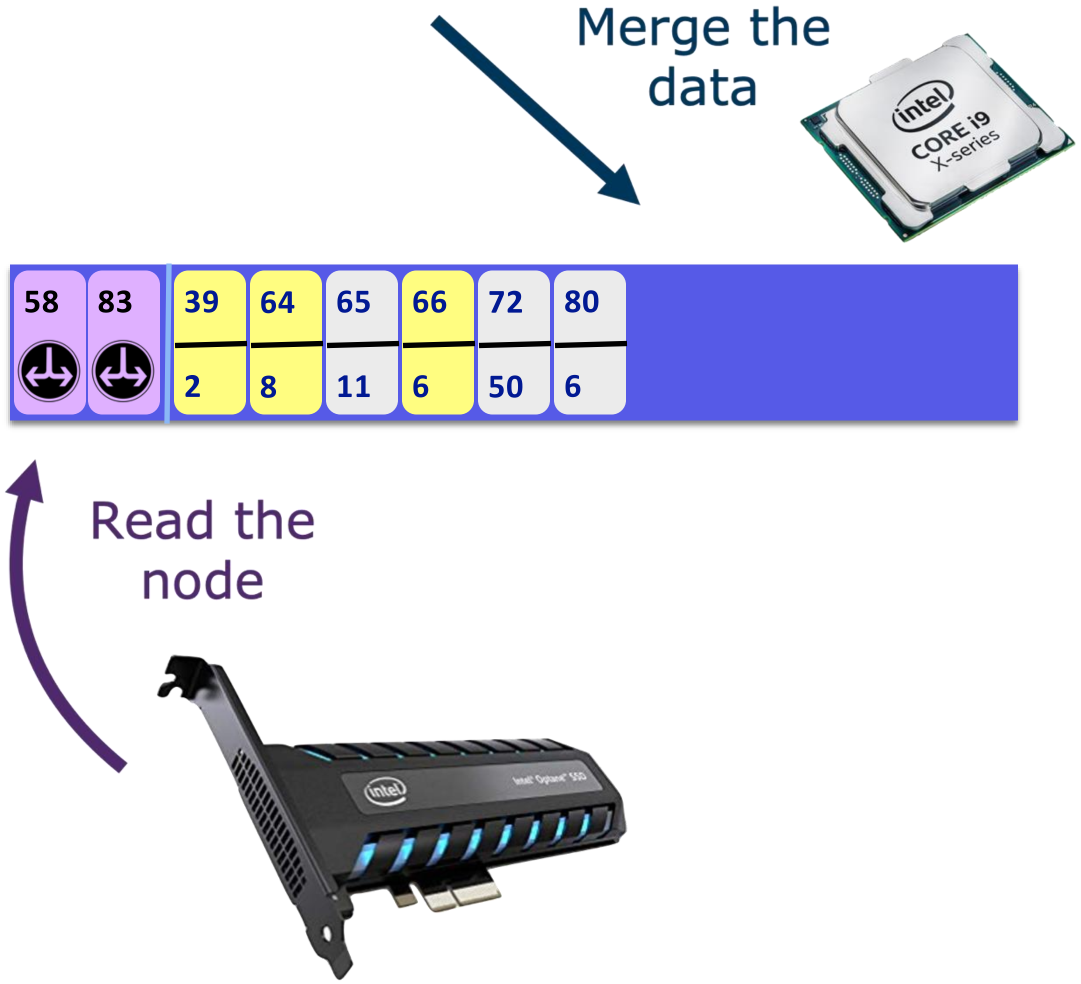
Insertions in B$^{\varepsilon}$-Trees are more expensive than they look

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



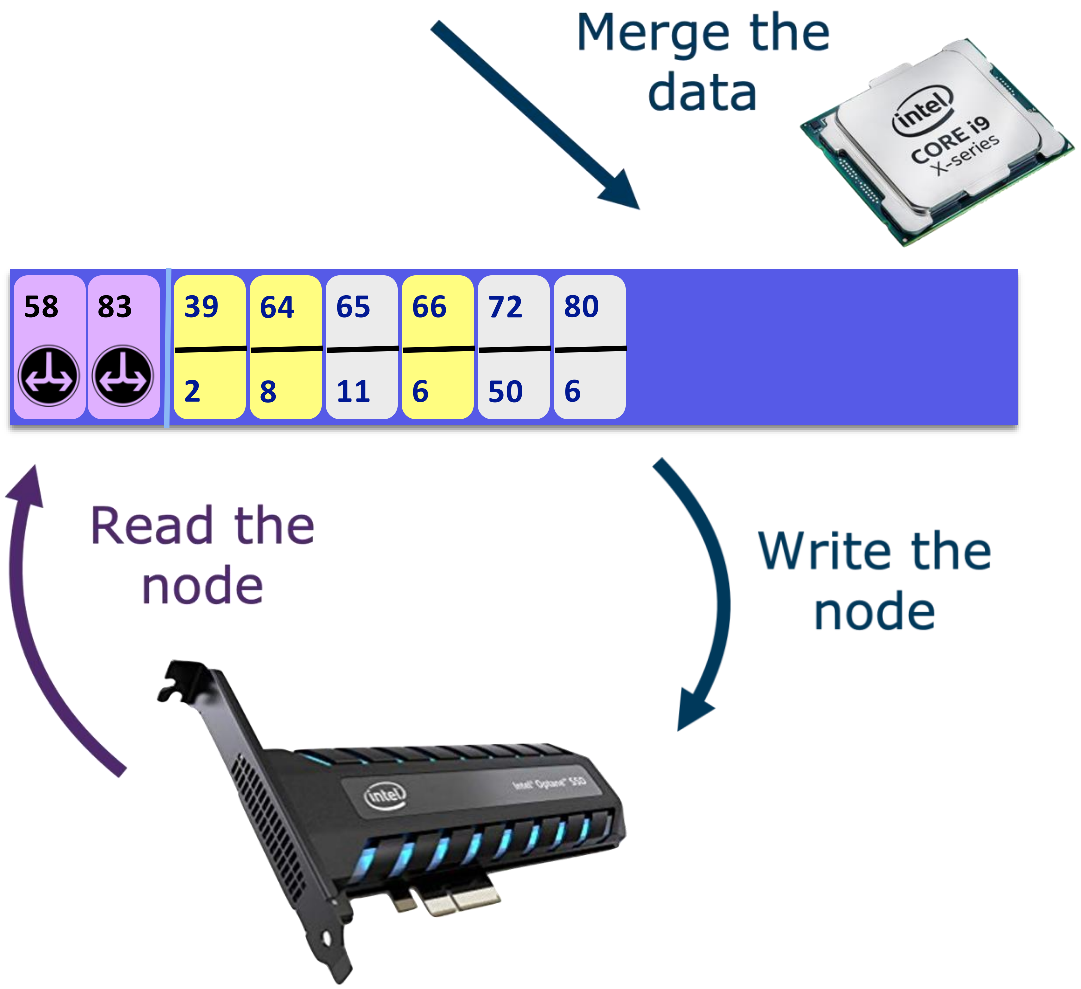
Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



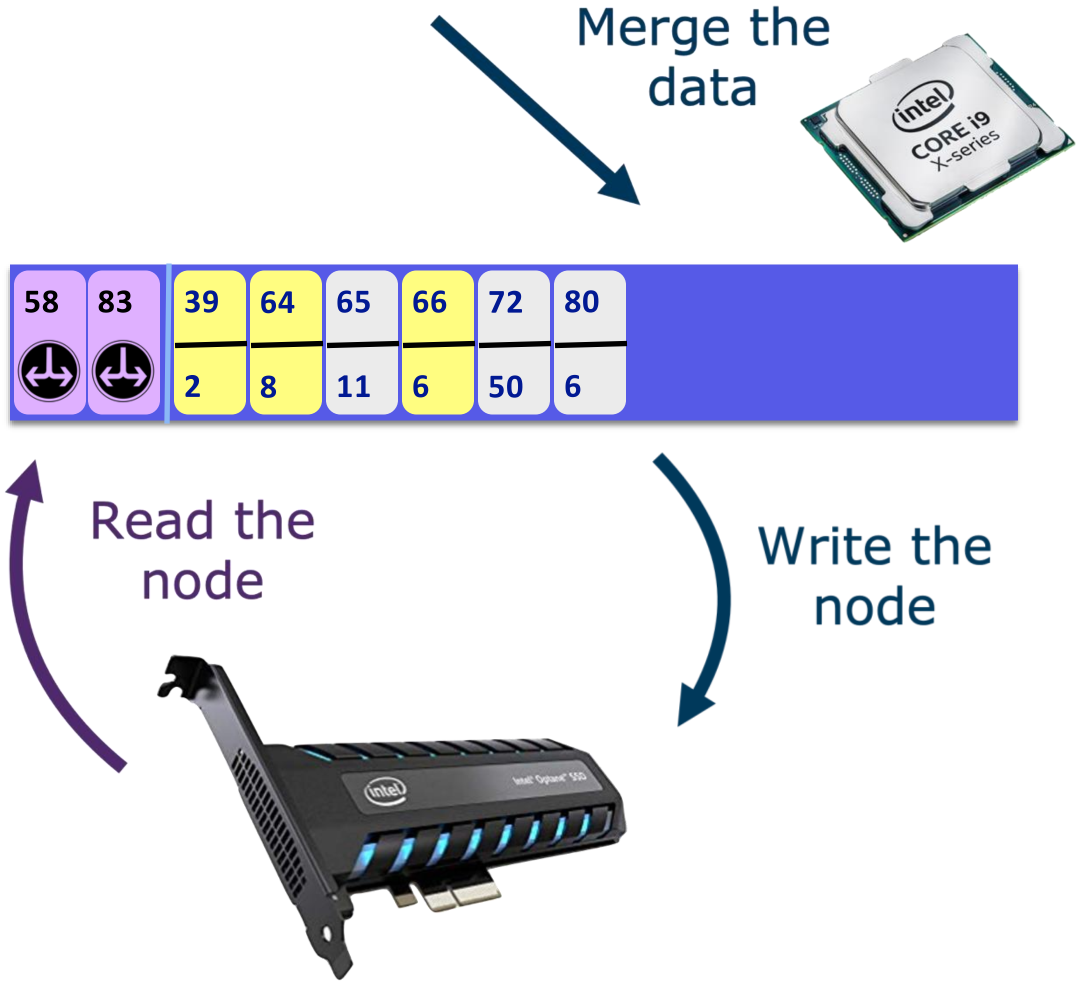
Insertions in B^{ε}-Trees are more expensive than they look

Recall: Insertions in B^{ε}-trees



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

Recall: Insertions in B^{ε}-trees

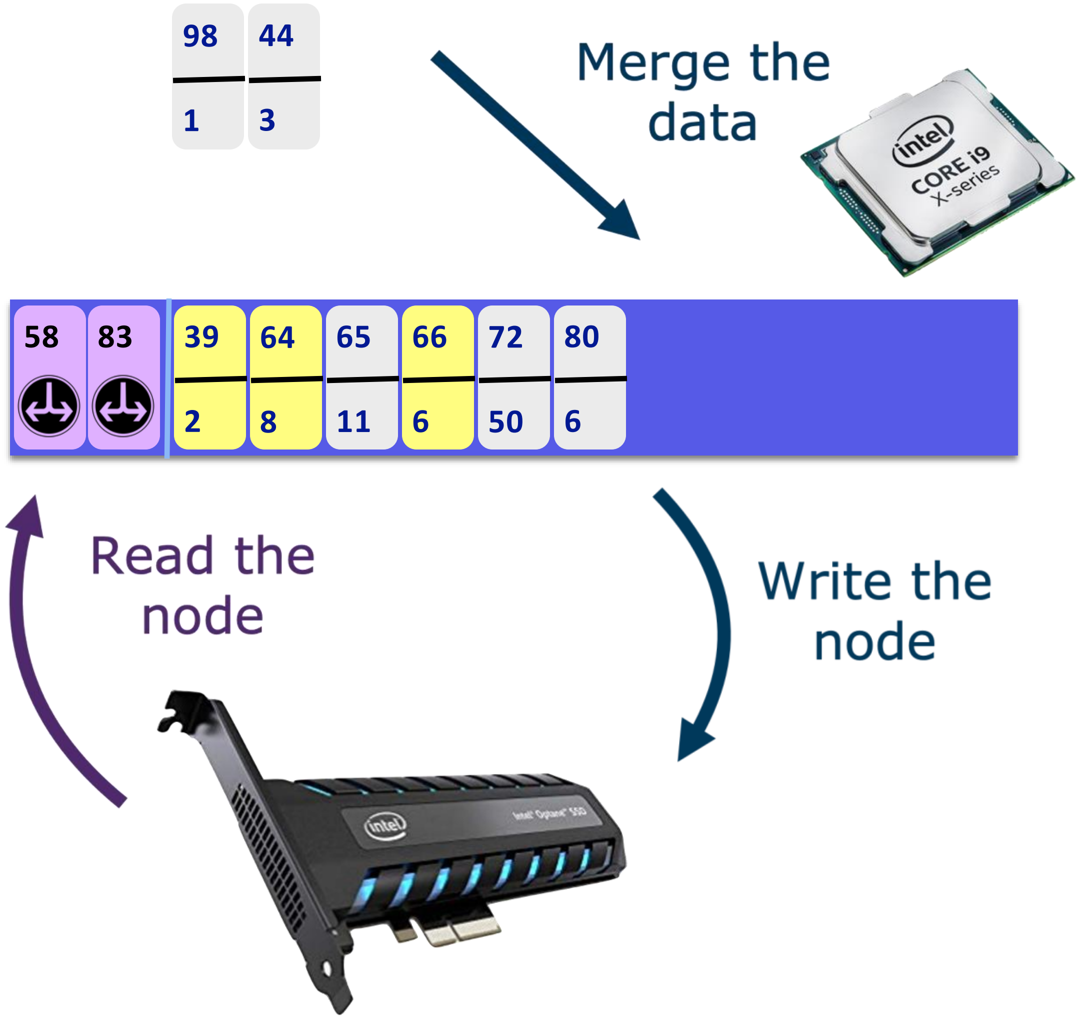


CPU Work = O(old + new)

Volume of IO = O(old + new)

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

Recall: Insertions in B^{ε}-trees



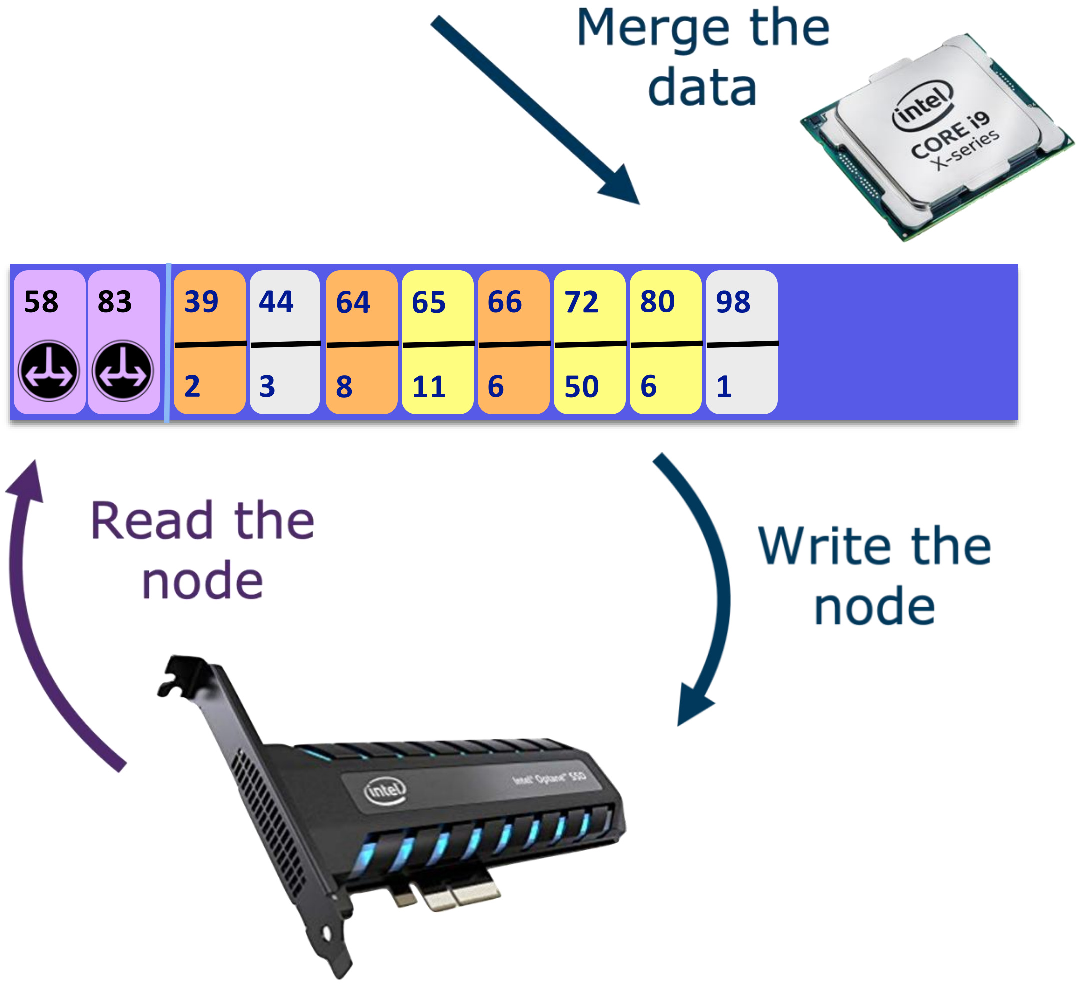
CPU Work = O(old + new)

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



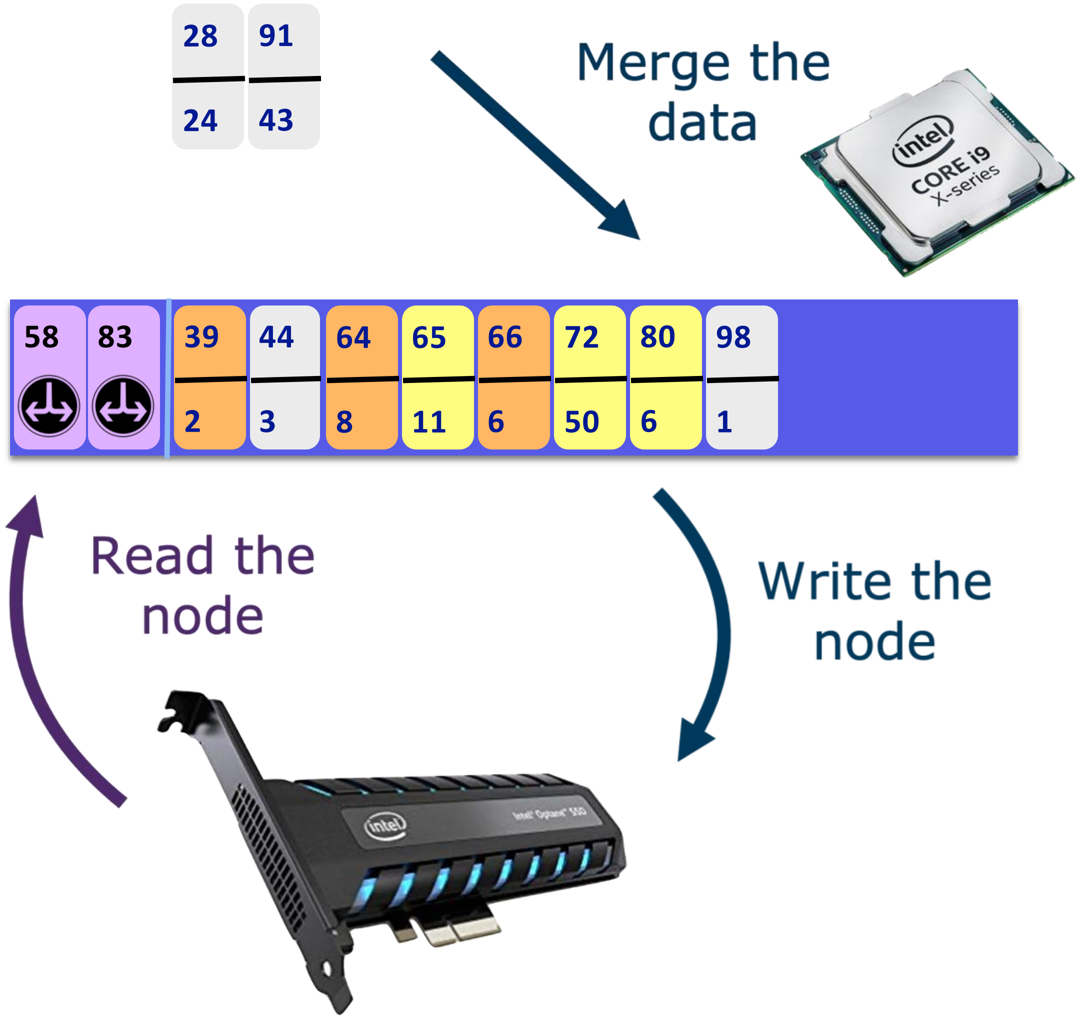
CPU Work = O(old + new)

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



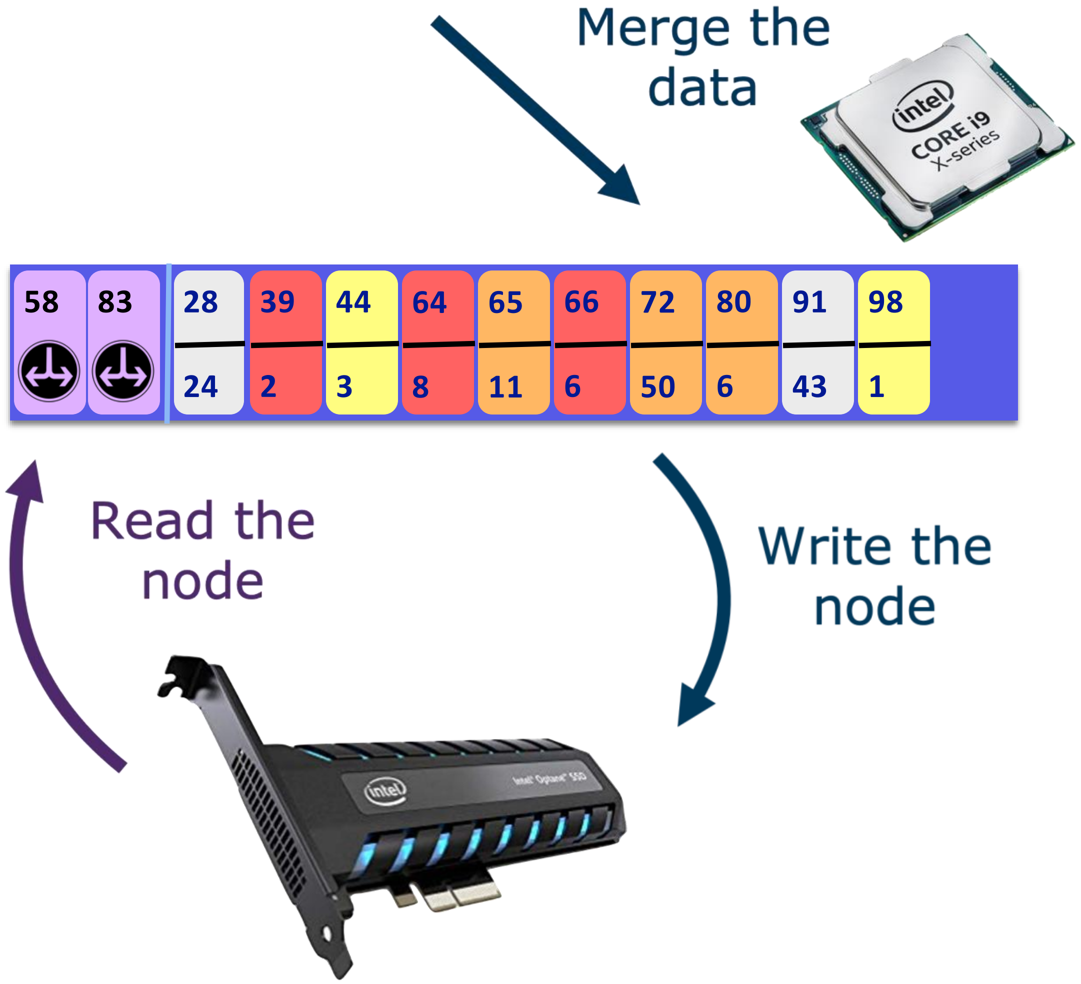
CPU Work = O(old + new)

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



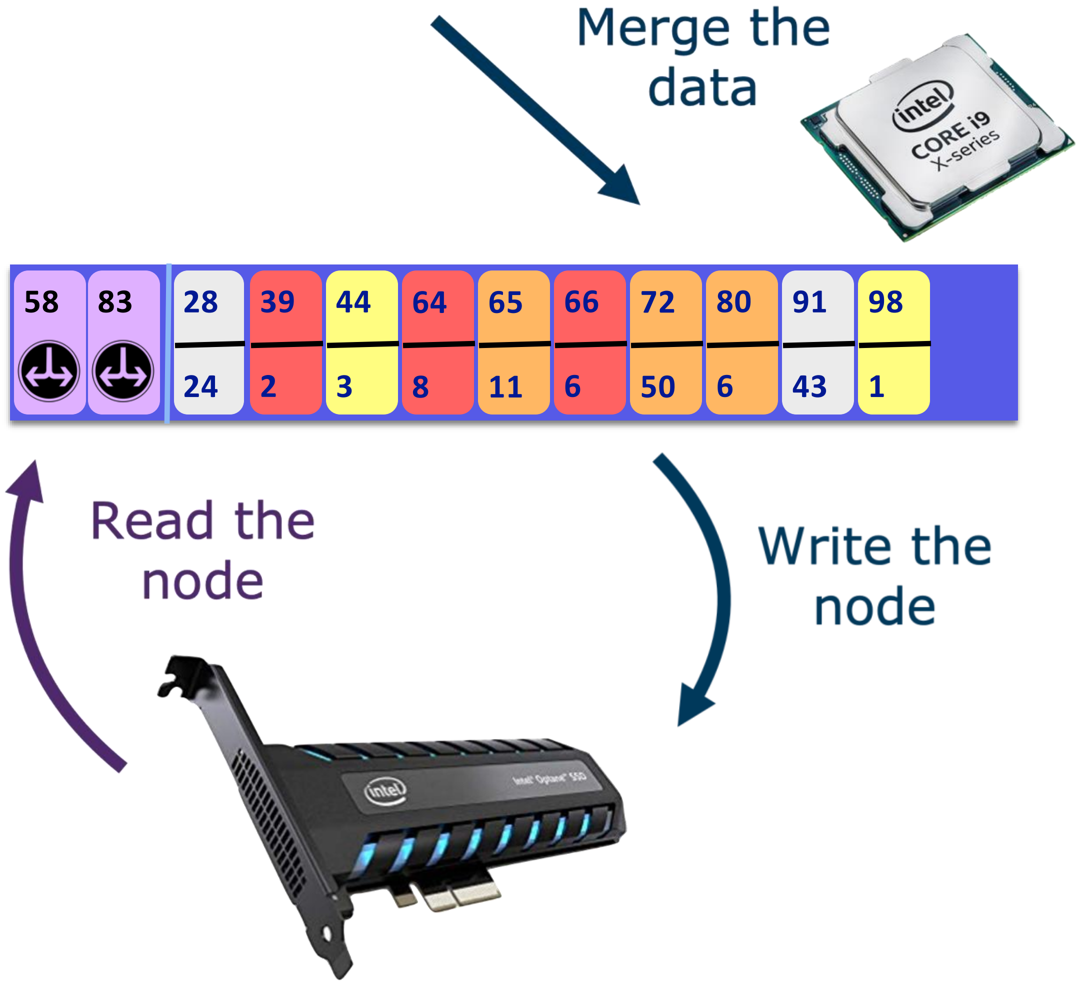
CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

Insertions in Bᵋ-Trees are more expensive than they look

Recall: Insertions in Bᵋ-trees



CPU Work = O(old + new)

Volume of IO = O(old + new)

Older data gets written over and over again

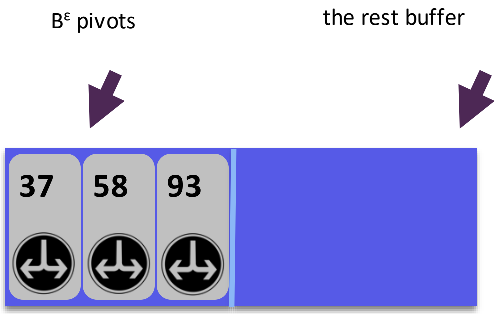
Up to B^{ε} times per node!

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

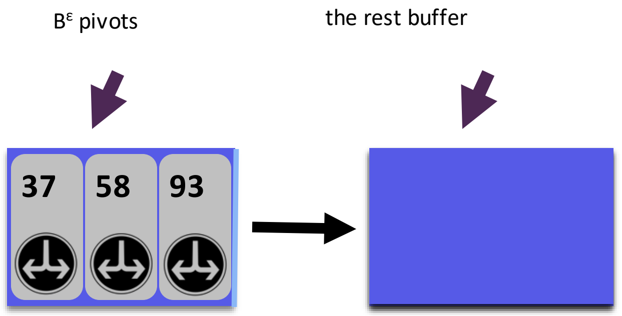


Recall:
a B^{ε}-tree node has pivots and a buffer

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

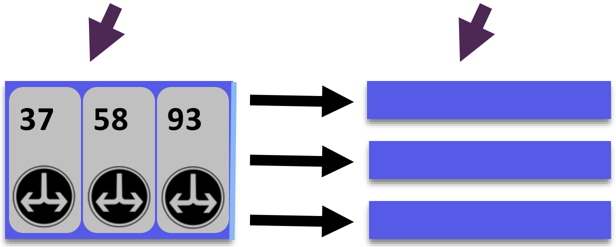
Size-Tiered B^{ε}-Trees

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

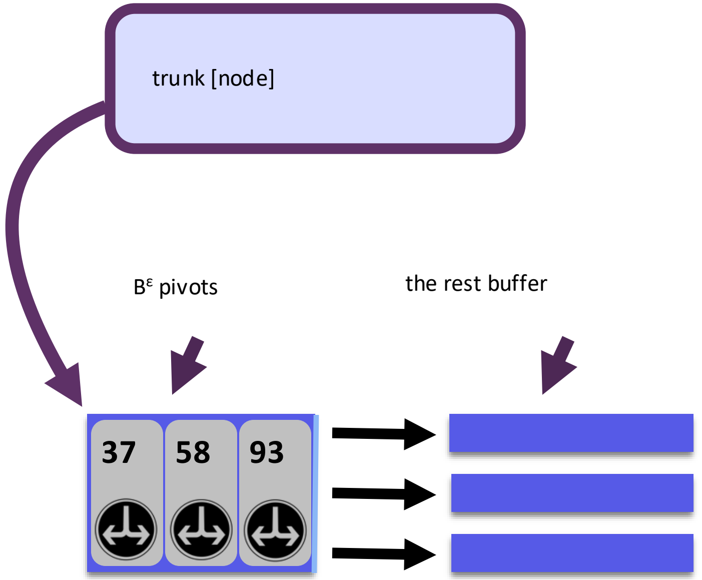


and in several discontiguous pieces

In an STB^{ε}-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

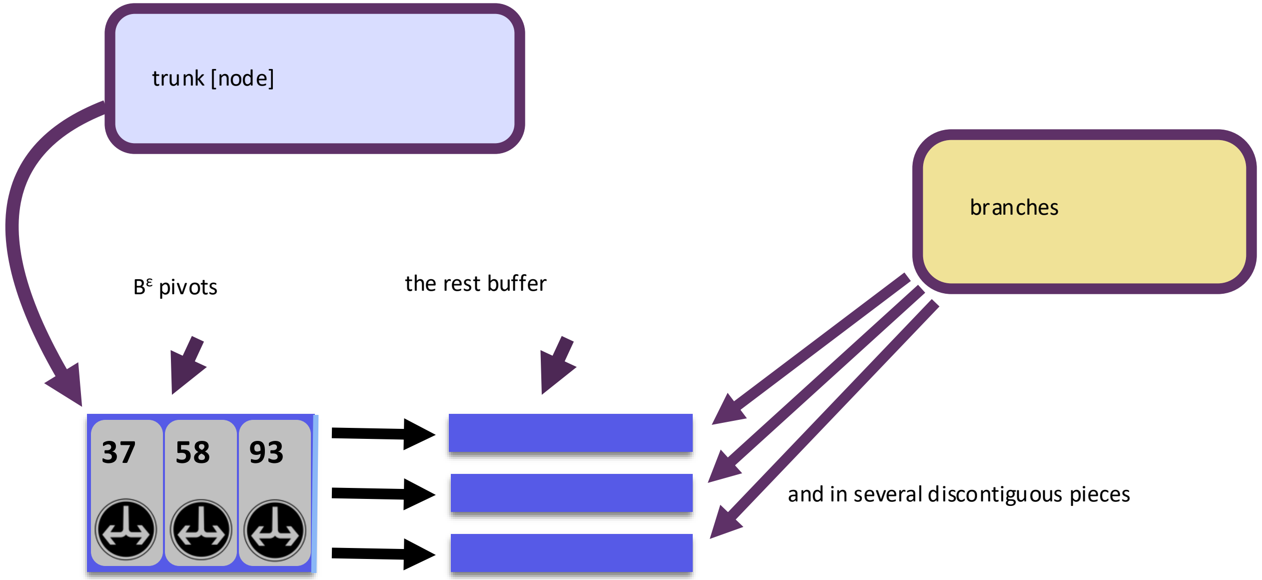
and in several discontiguous pieces

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

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



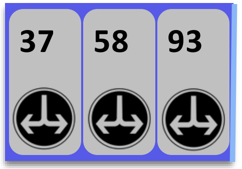
In an STB$^{\varepsilon}$-tree, the buffer is stored separately

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

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

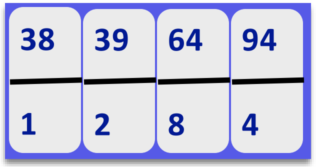
A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node…

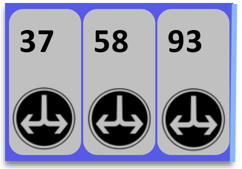


Size-Tiered B$^{\varepsilon}$-Trees

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…

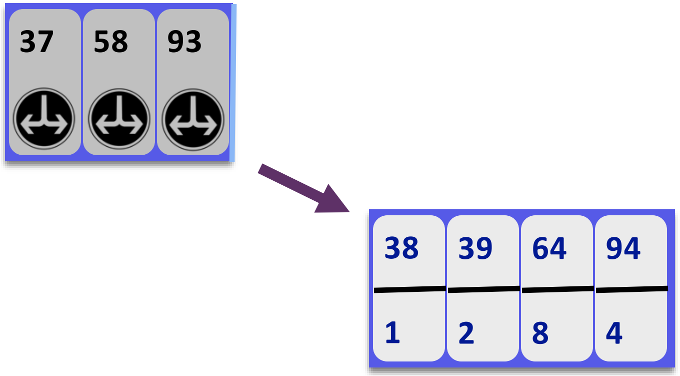


Size-Tiered B$^{\varepsilon}$-Trees

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…

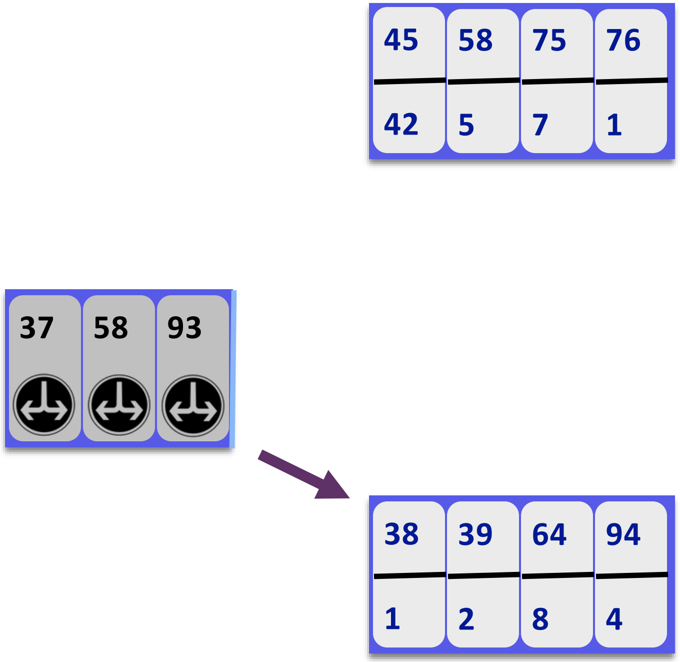
…it is added as a new branch



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

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…

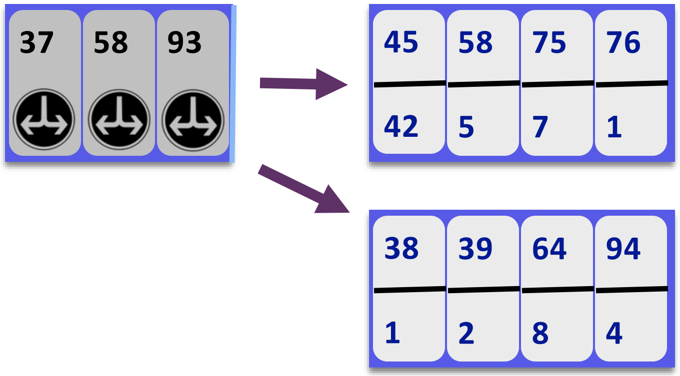
…it is added as a new branch

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

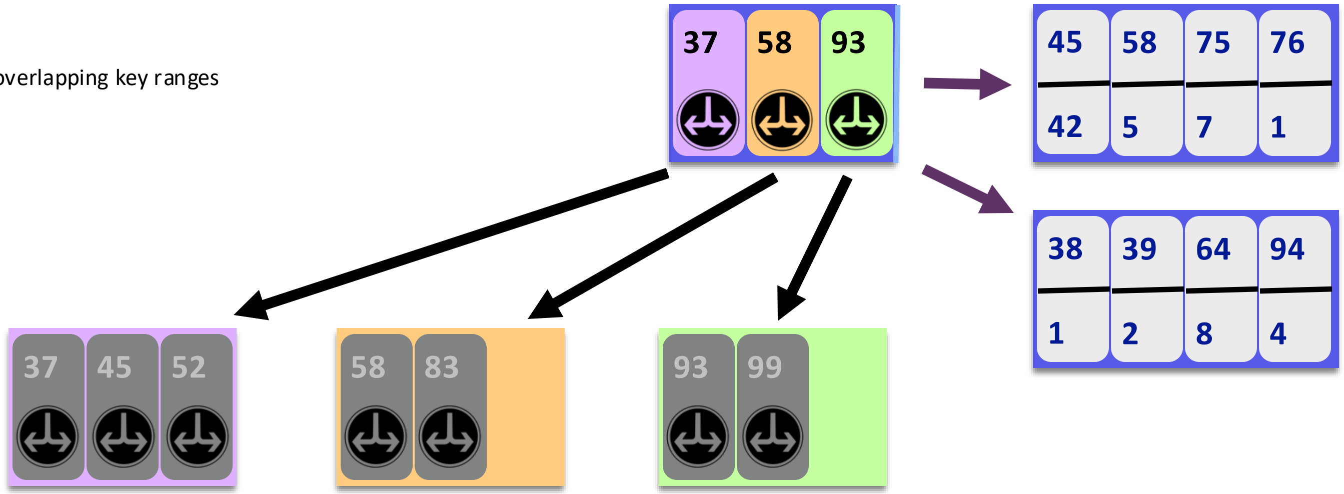
A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

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



SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

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

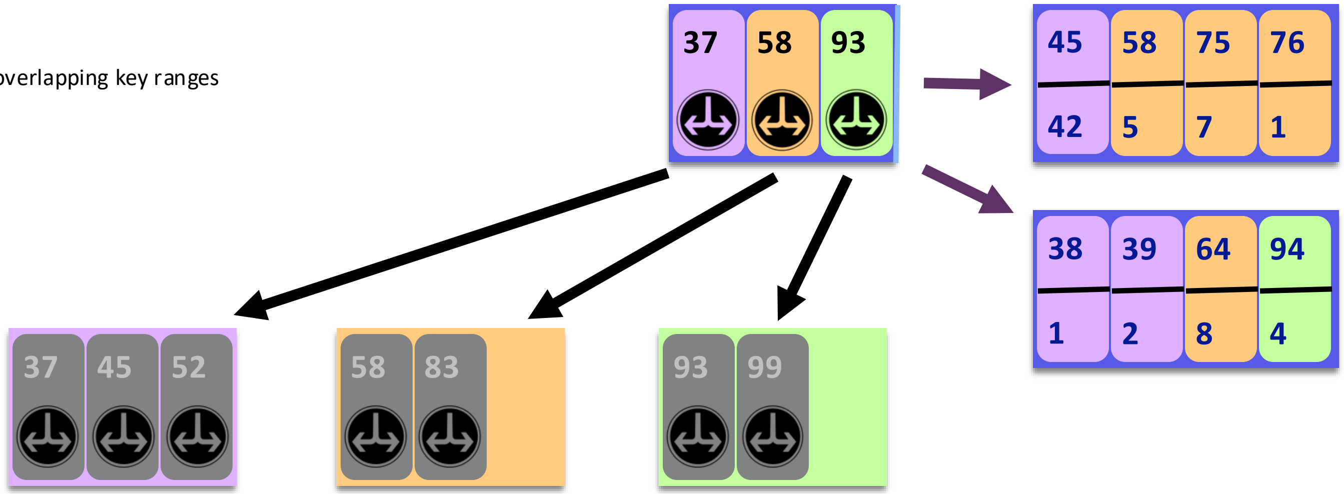
A Size-Tiered B$^{\varepsilon}$-tree is a B$^{\varepsilon}$-tree where the buffer is stored discontiguously

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

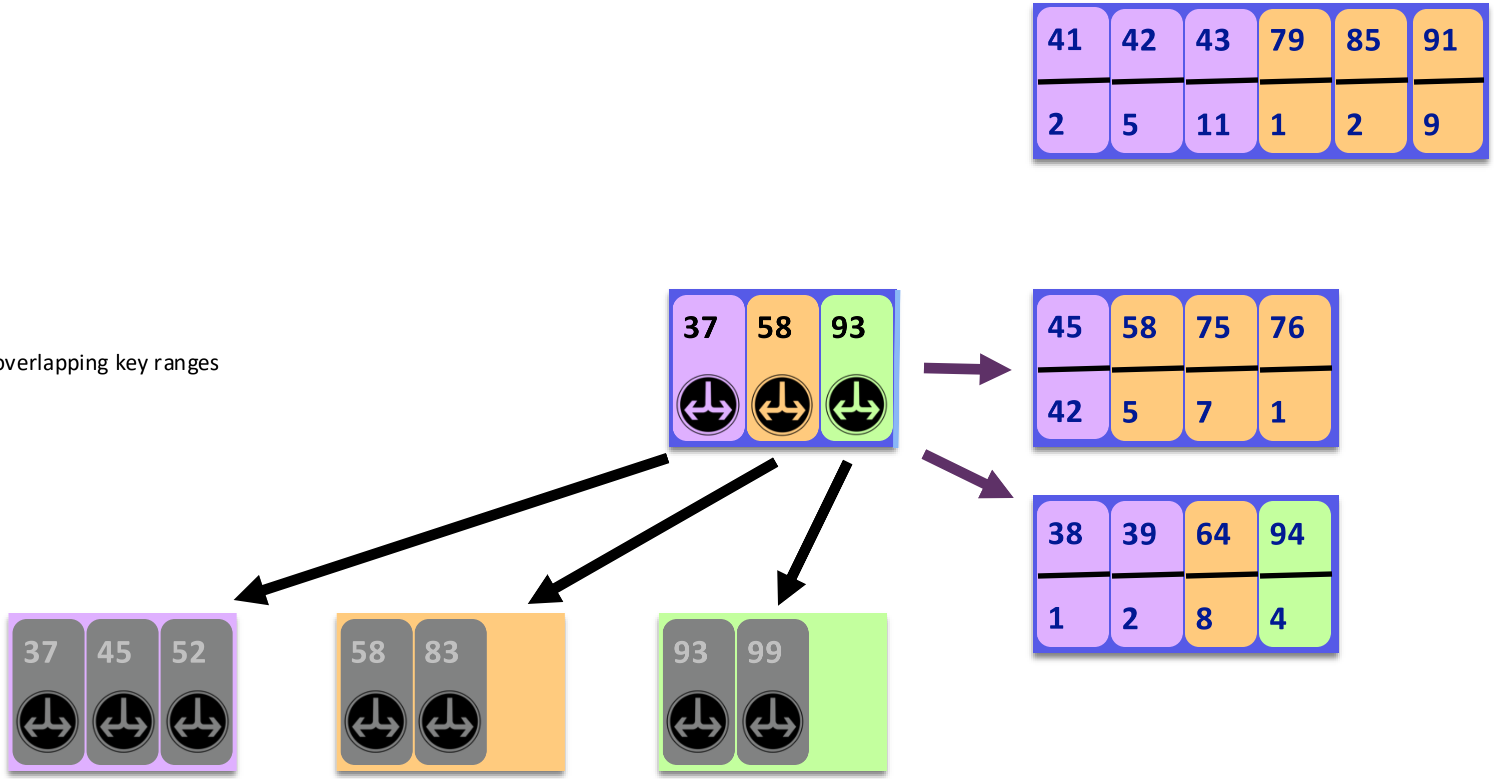
A Size-Tiered B$^\varepsilon$-tree is a B$^\varepsilon$-tree where the buffer is stored discontiguously

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



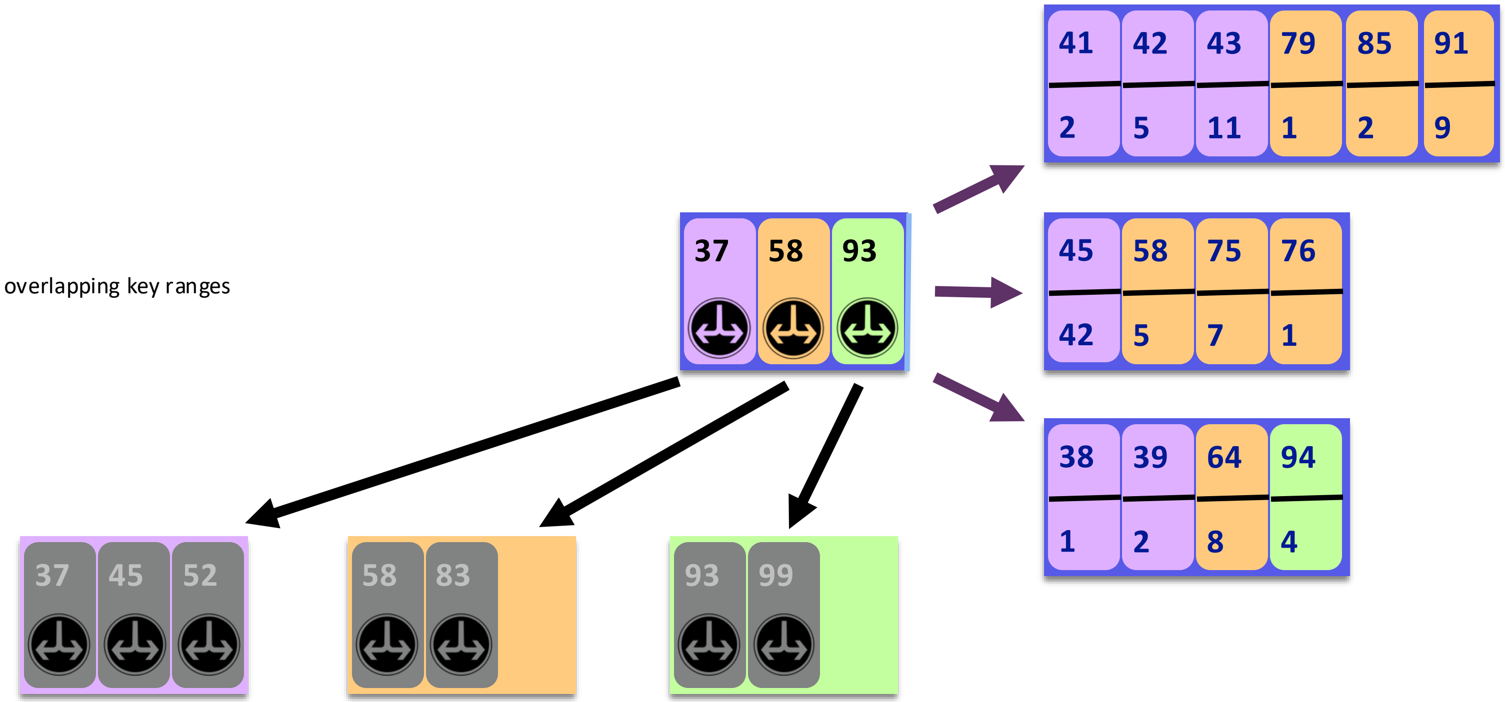
Size-Tiered B$^{\varepsilon}$-Trees

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



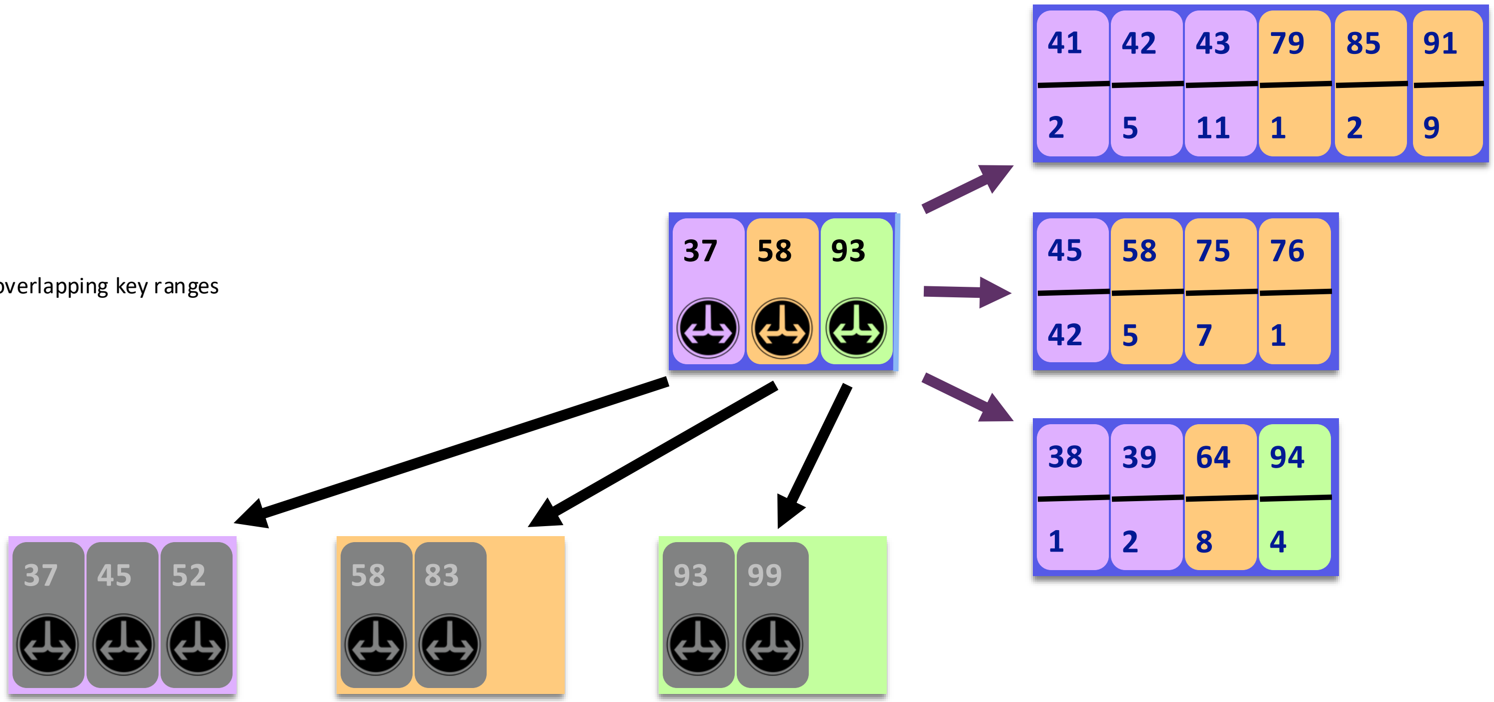
The old branches do not need to be rewritten

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

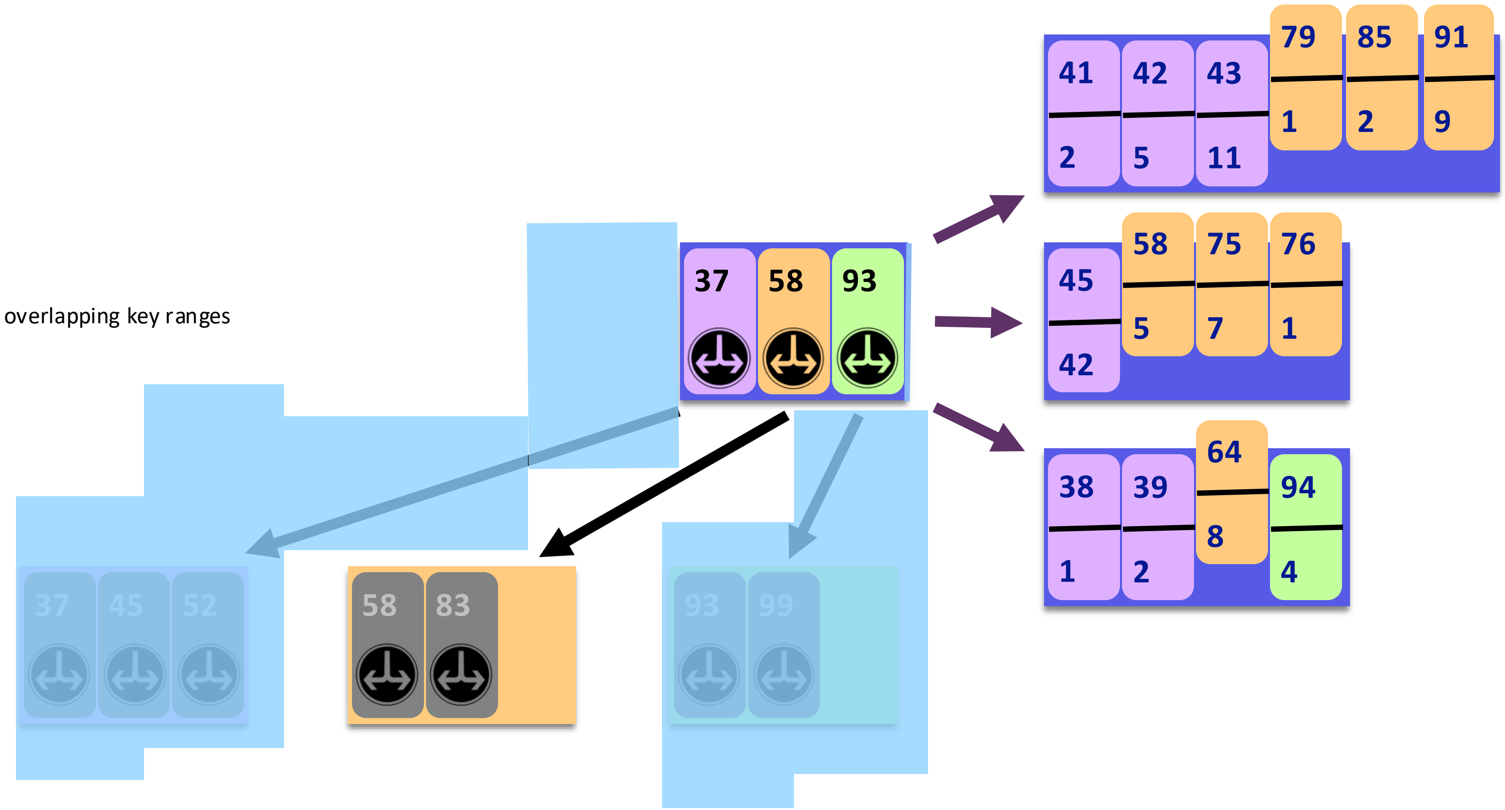
Size-Tiered B$^{\varepsilon}$-Trees

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

Branches may have overlapping key ranges



When new data is flushed into the trunk node…

…it is added as a new branch

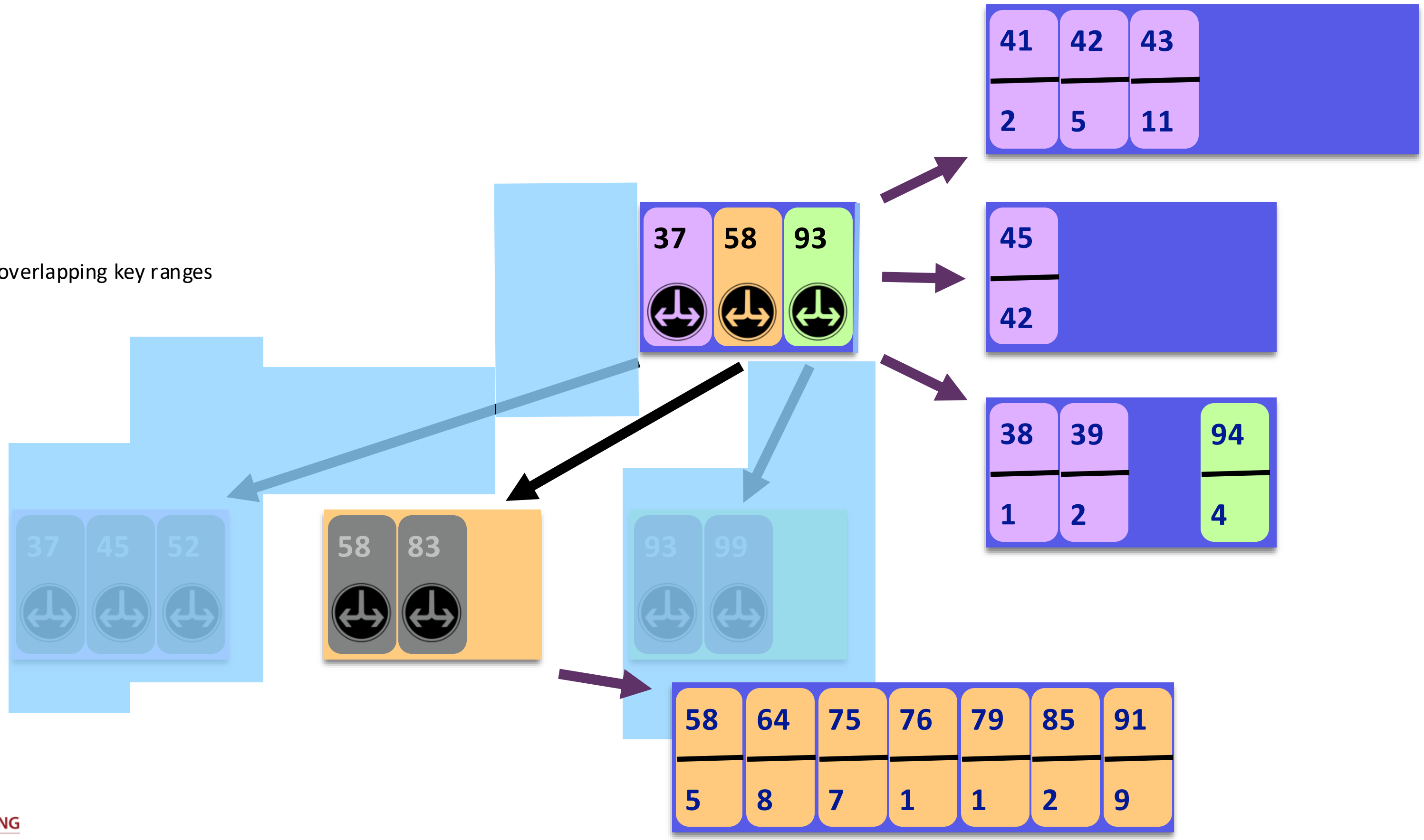
The old branches do not need to be rewritten

Size-Tiered 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

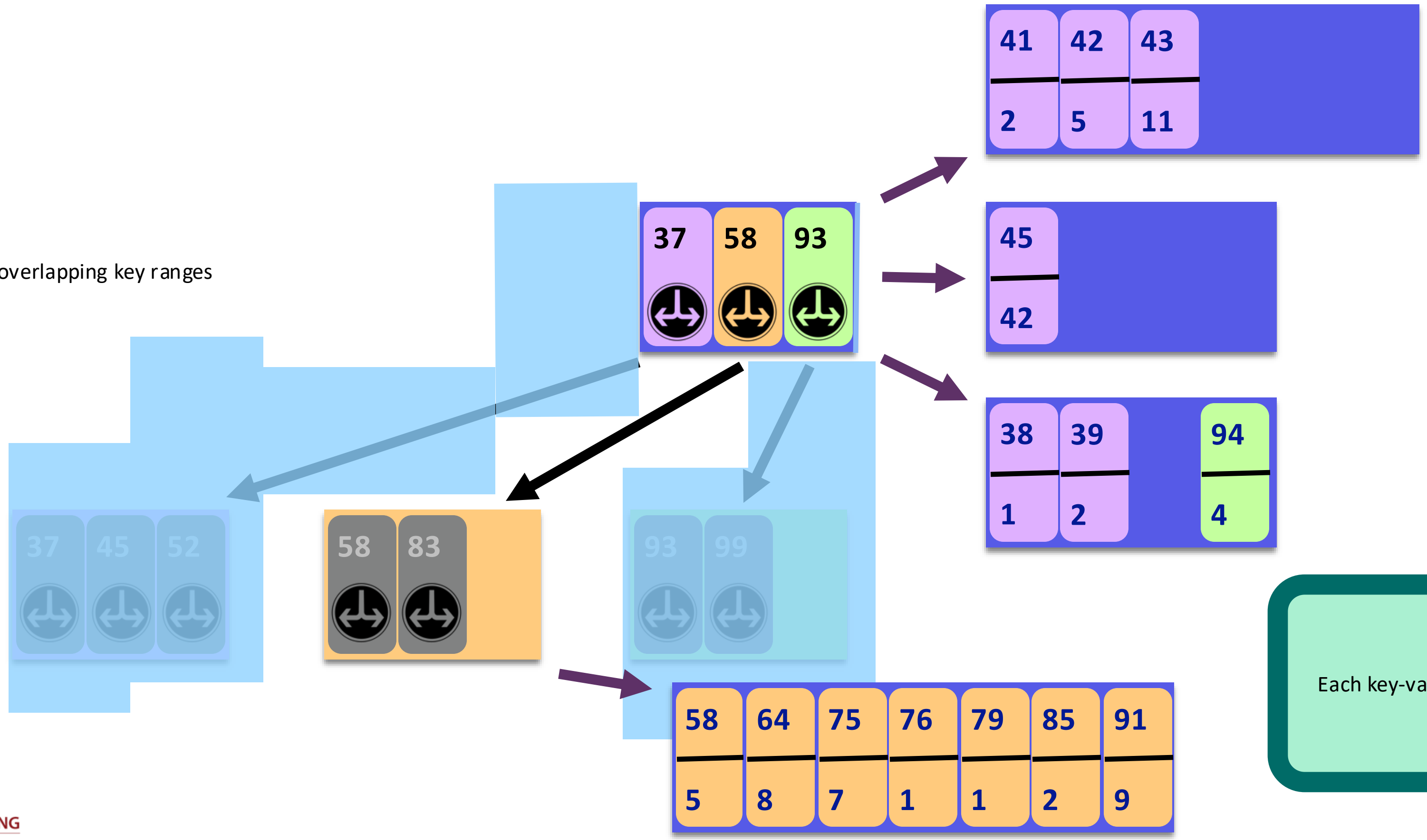
The old branches do not need to be rewritten

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

The old branches do not need to be rewritten

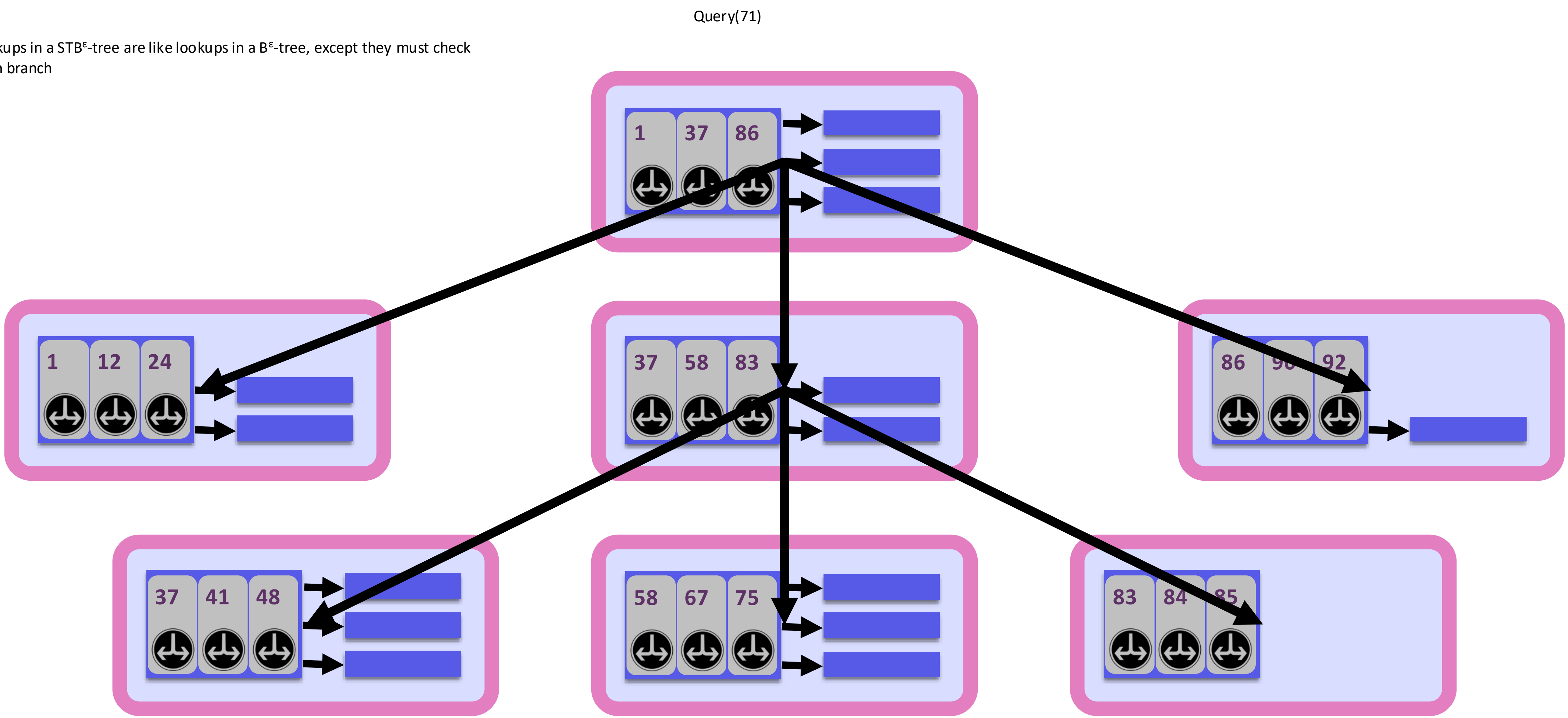
Branches may have overlapping key ranges

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

Lookups in Size-Tiered B^{ε}-Trees

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

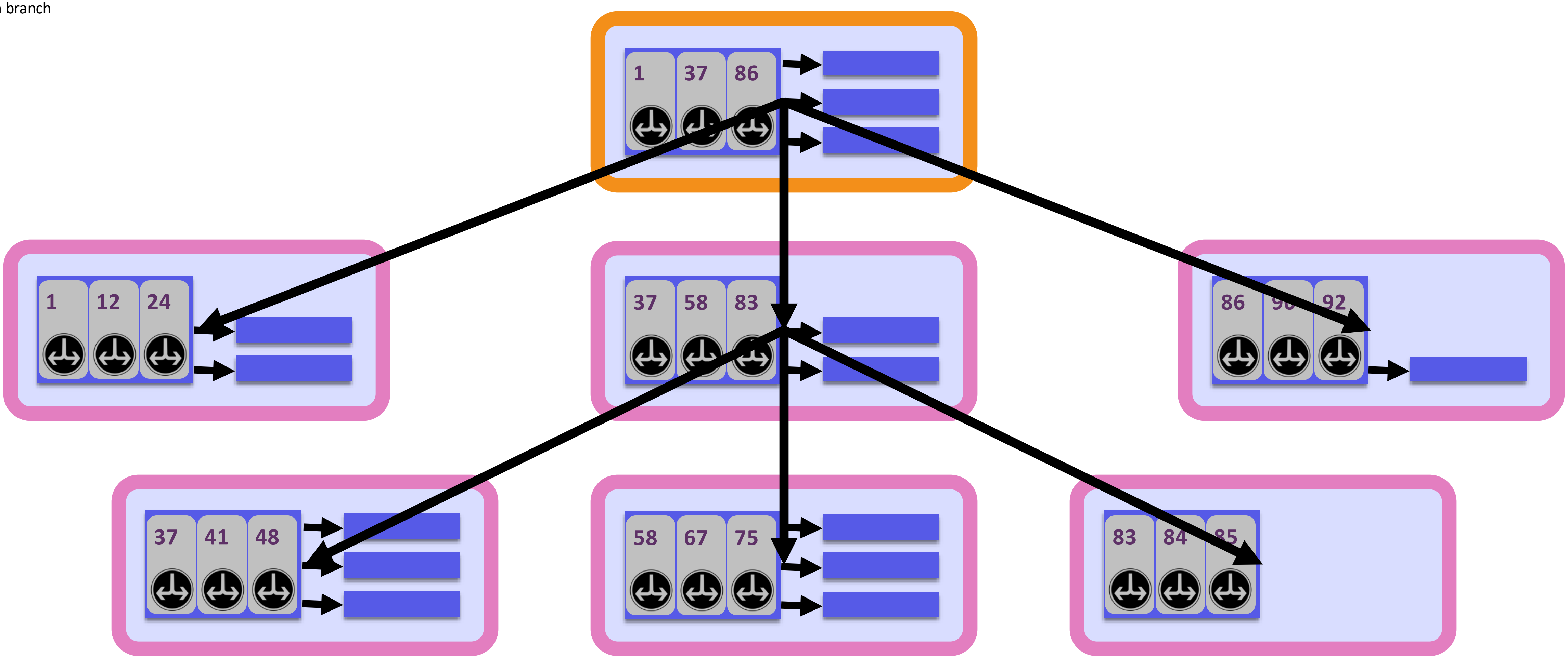
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

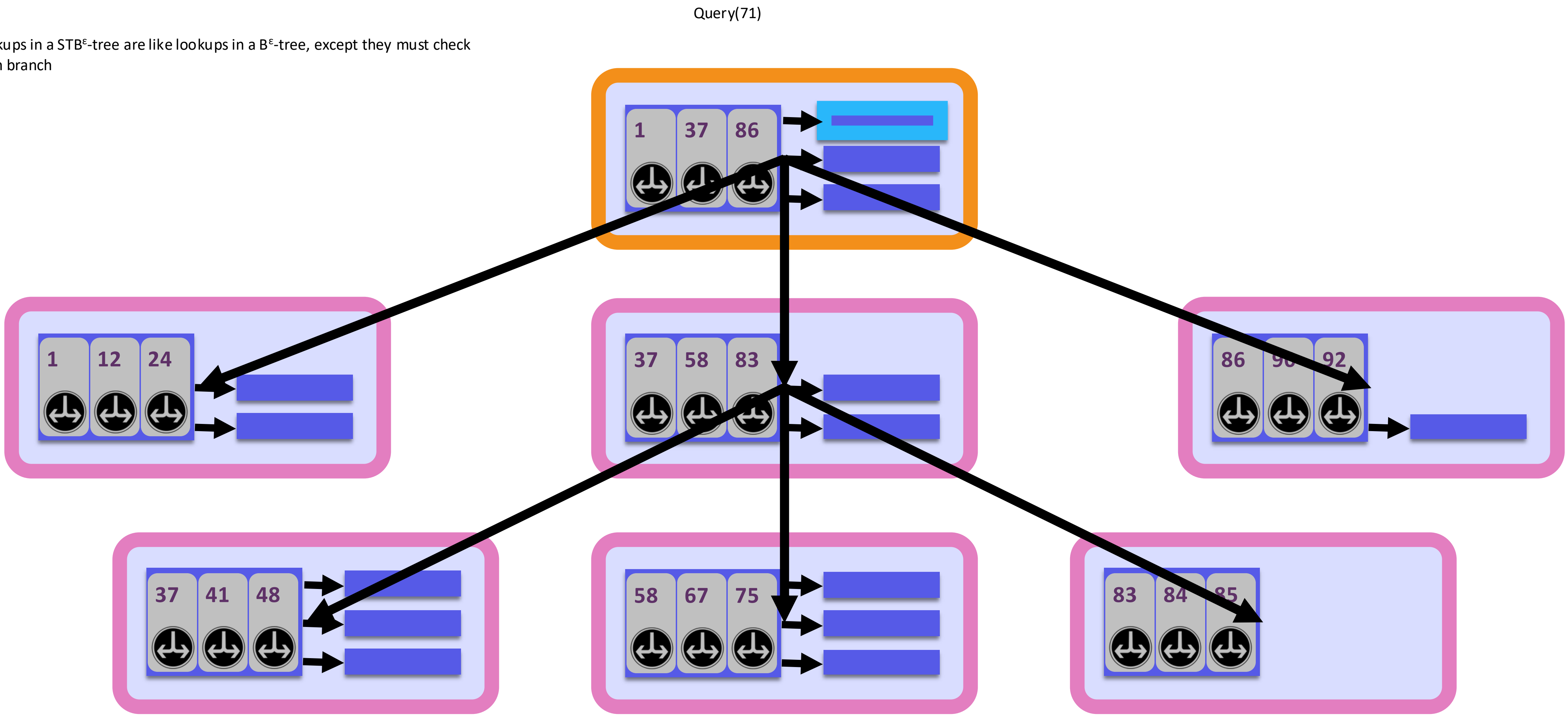
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch

Query(71)



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

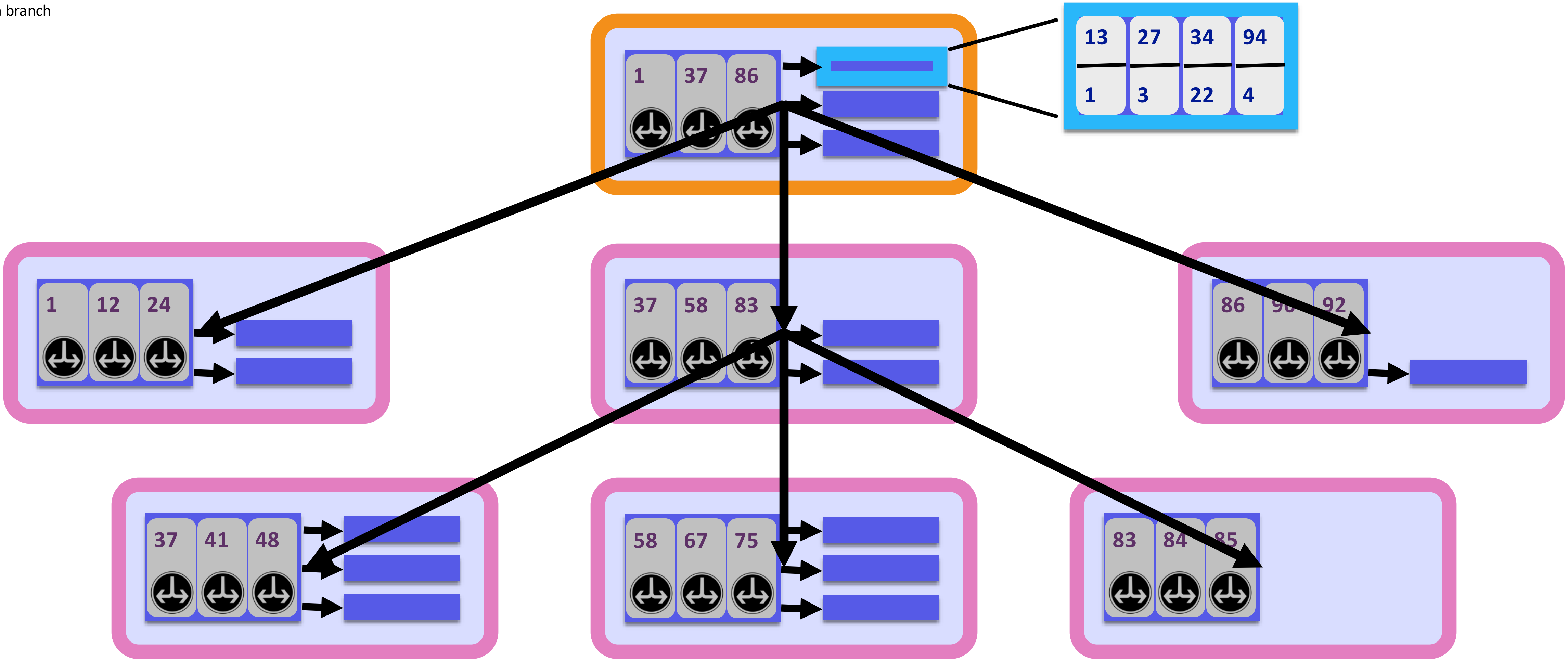
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



Size-Tiered B^{ε}-Trees

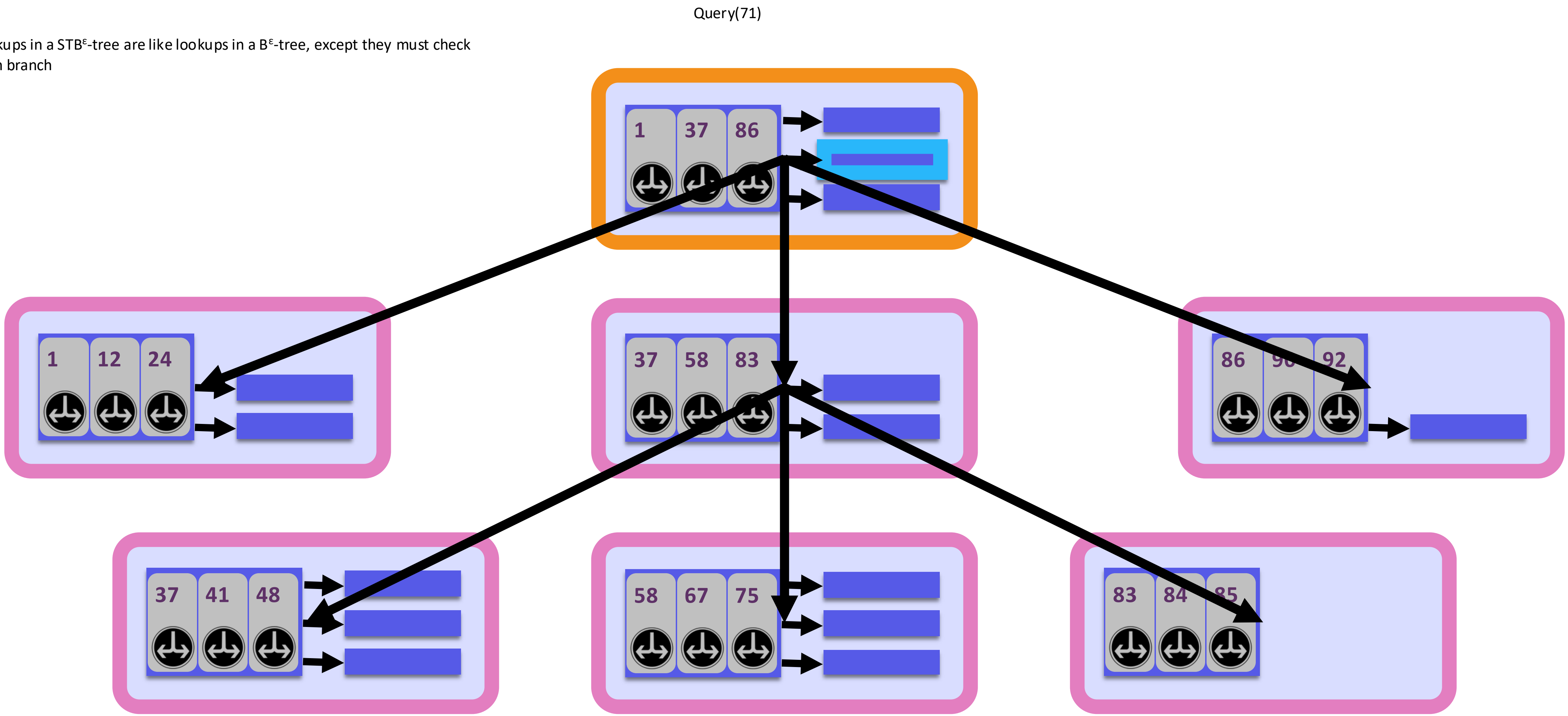
Query(71)

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



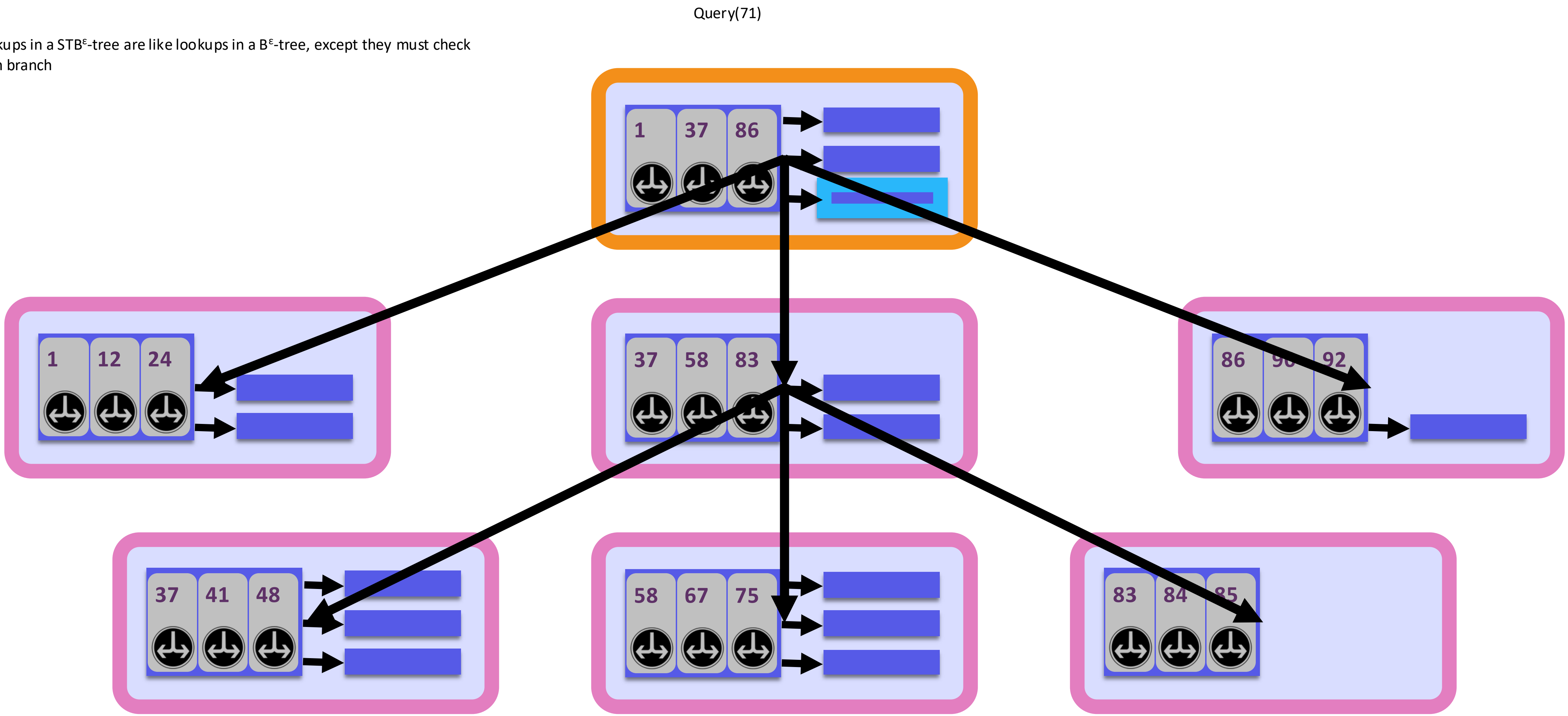
Size-Tiered B$^{\varepsilon}$-Trees

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



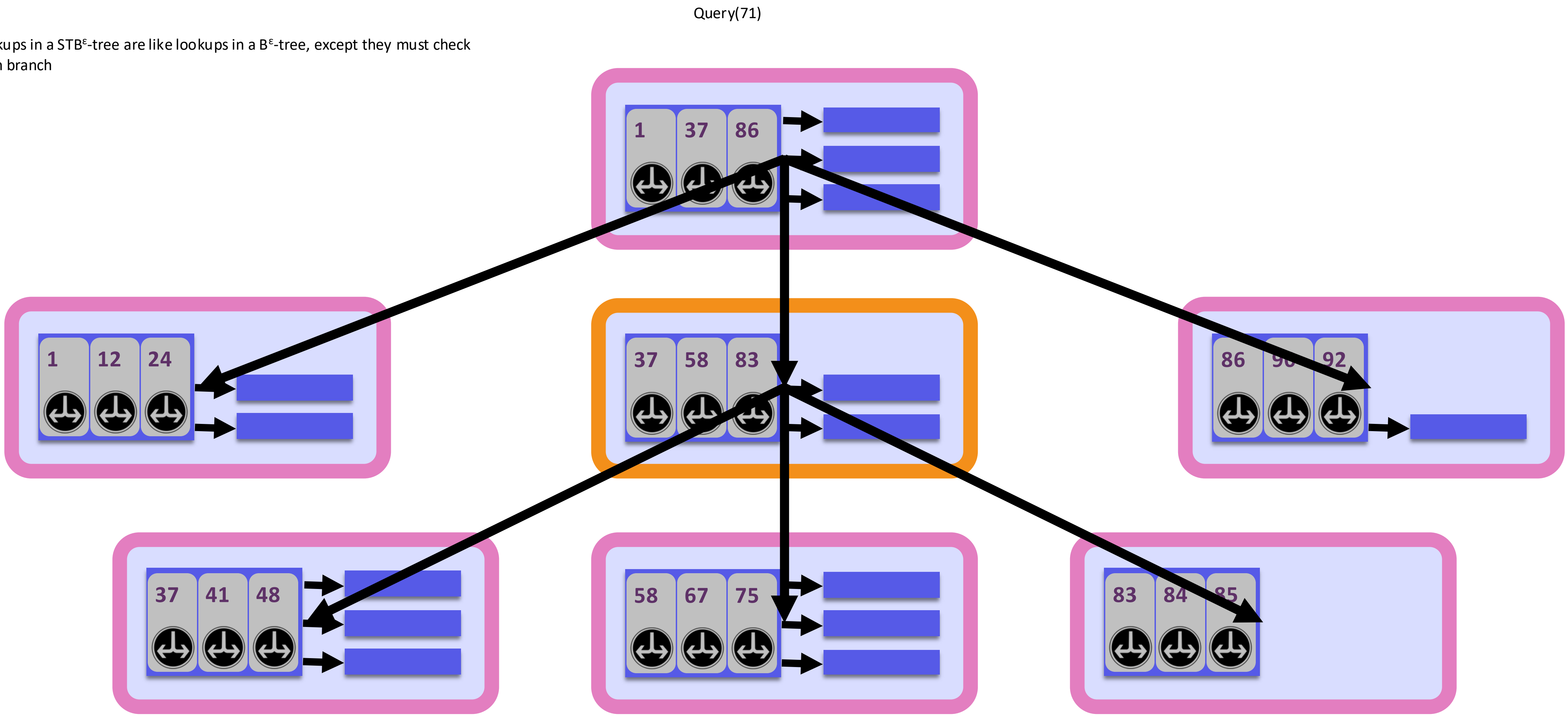
Size-Tiered B^{ε}-Trees

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



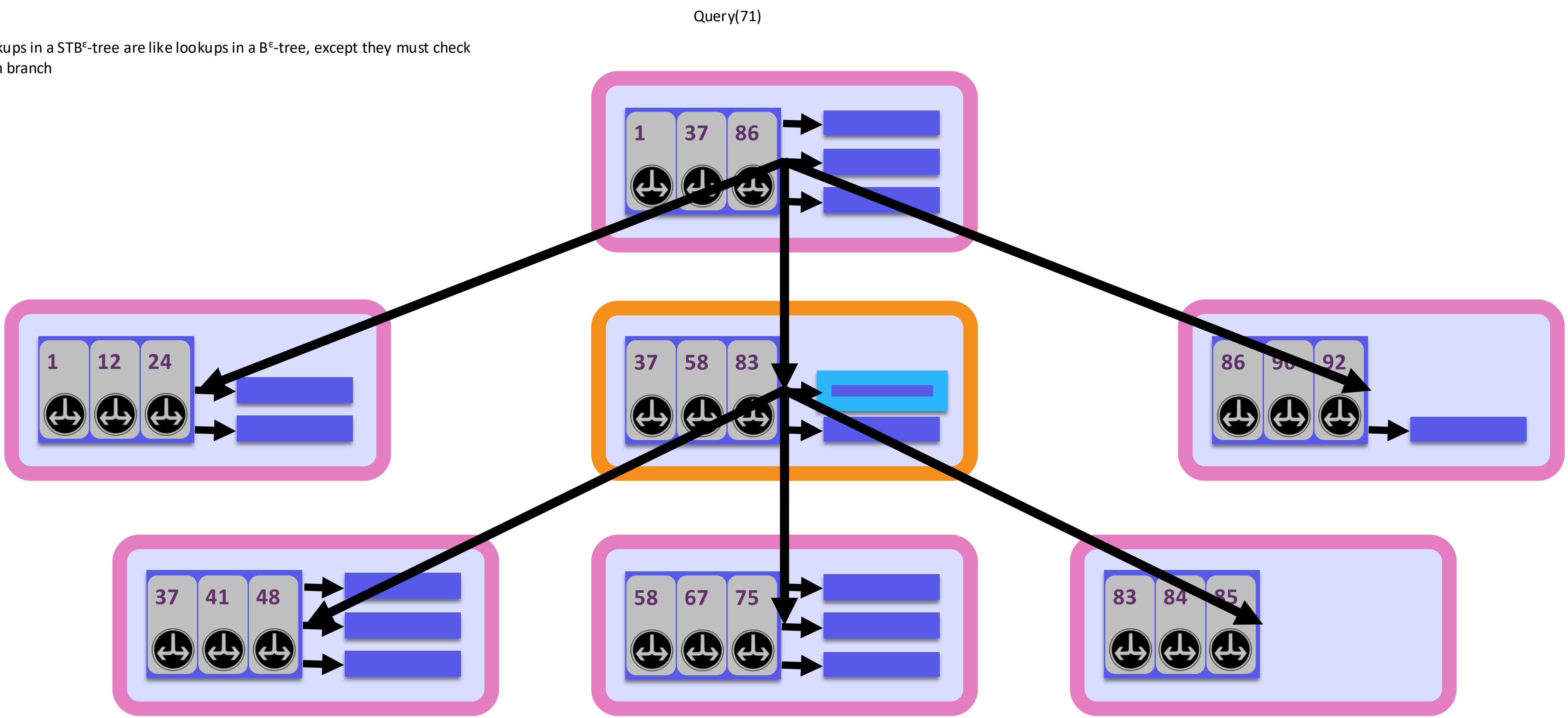
Size-Tiered B$^{\varepsilon}$-Trees

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



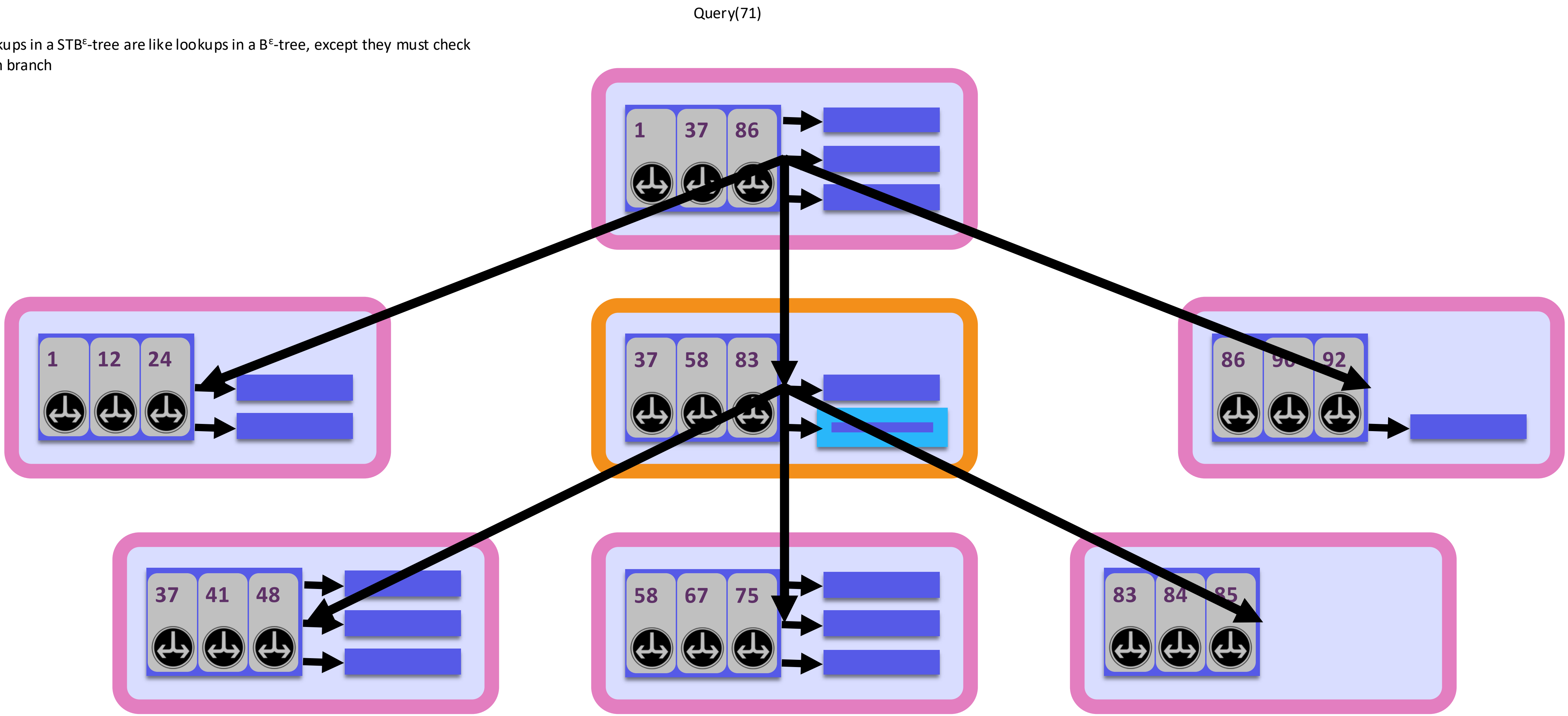
Size-Tiered B$^{\varepsilon}$-Trees

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



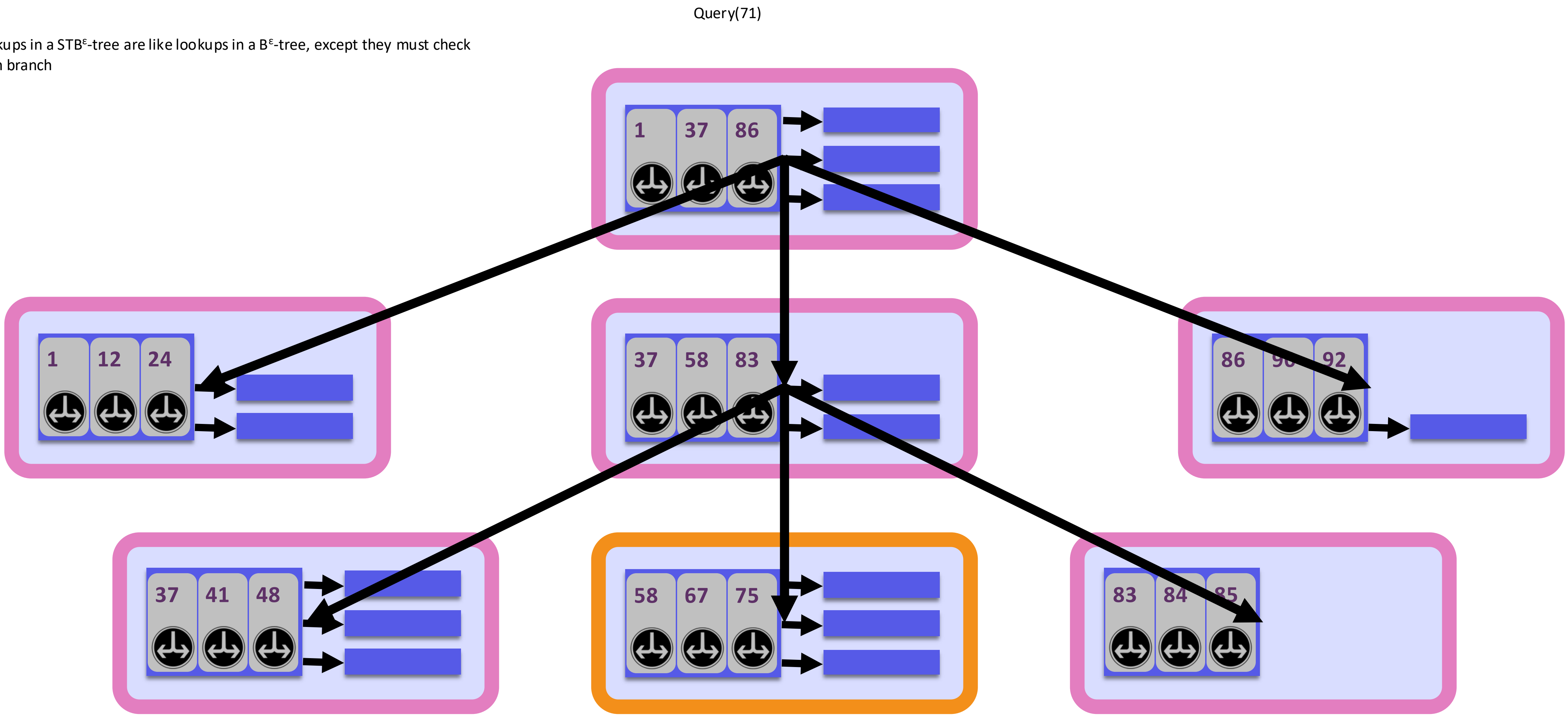
Size-Tiered B$^{\varepsilon}$-Trees

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



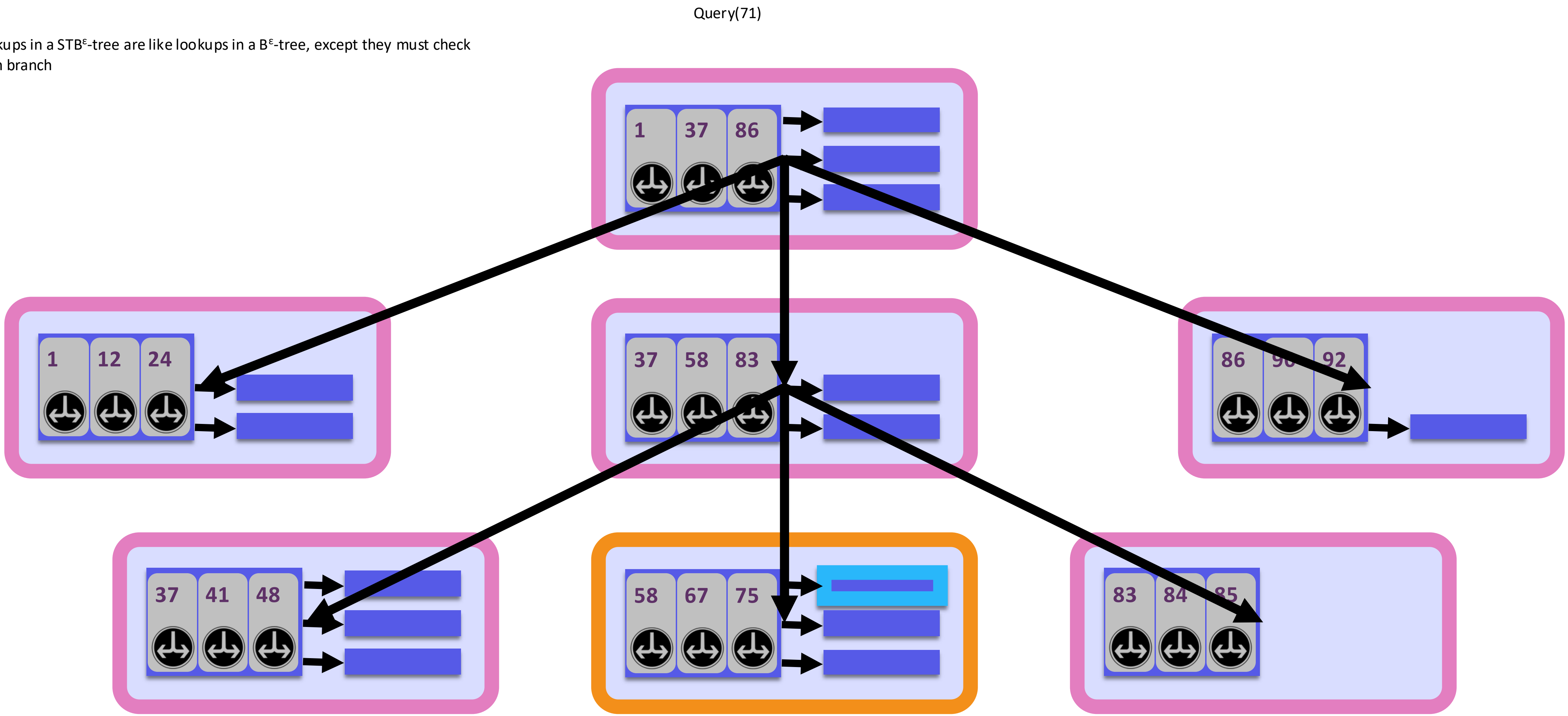
Size-Tiered B$^{\varepsilon}$-Trees

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

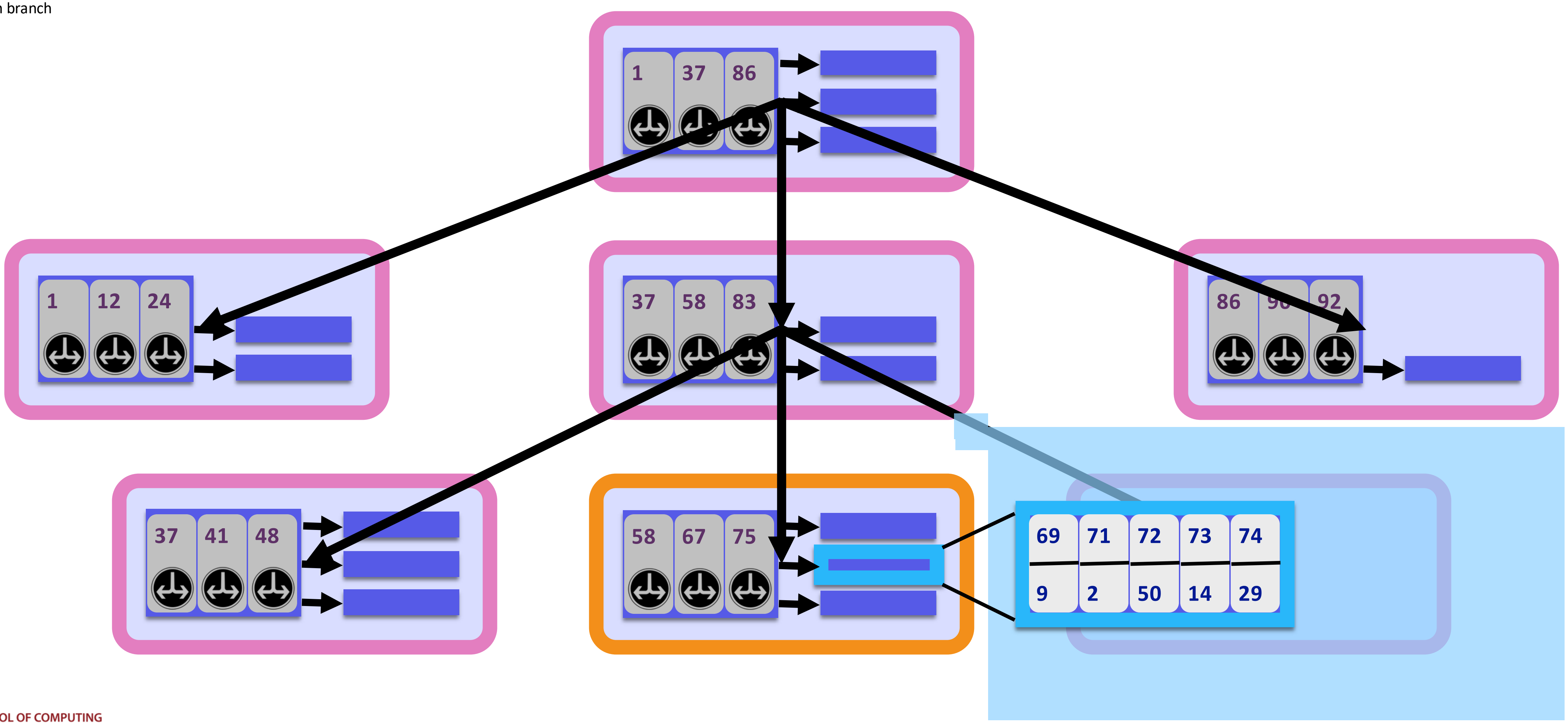
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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

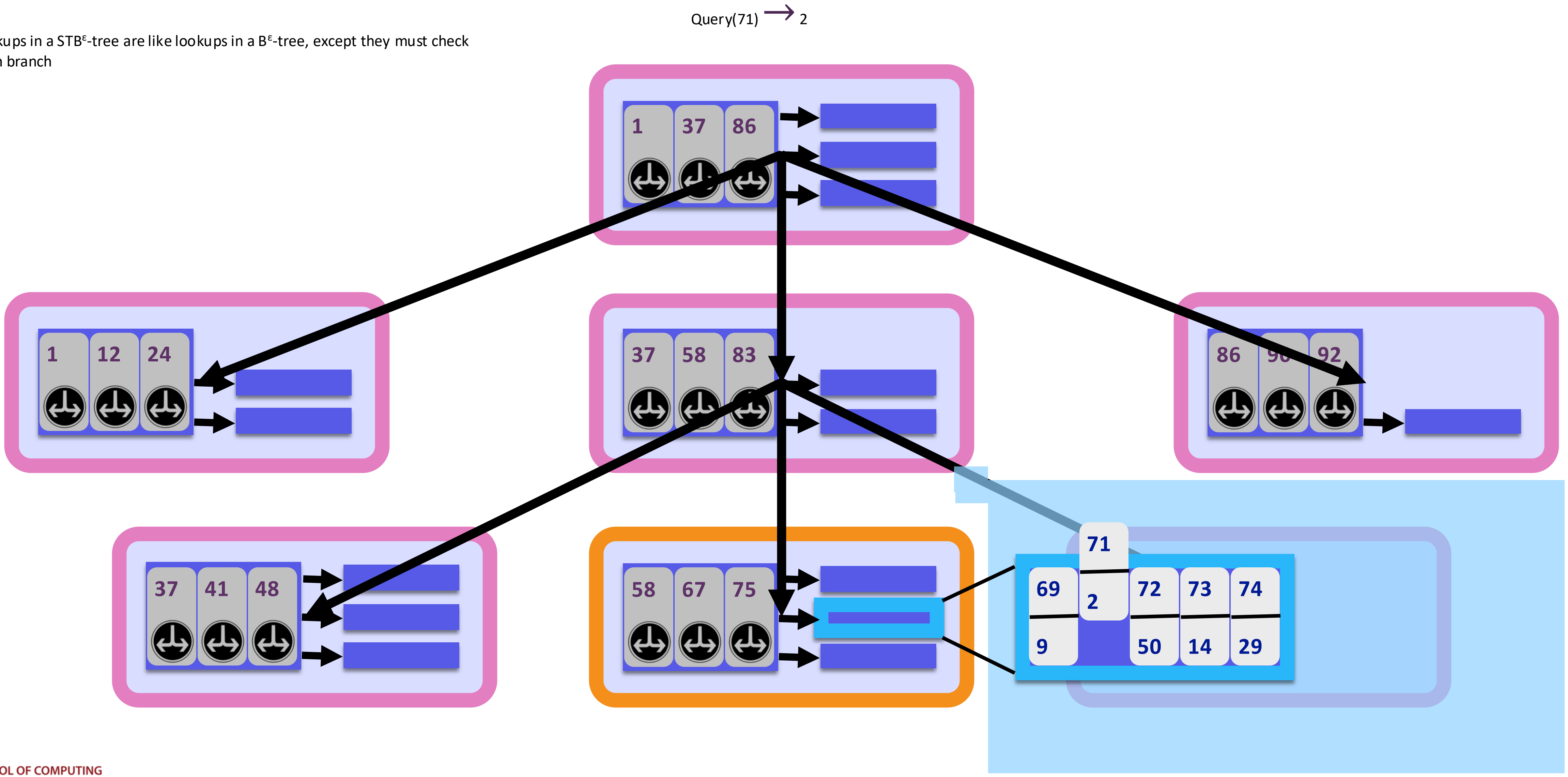
Query(71)

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

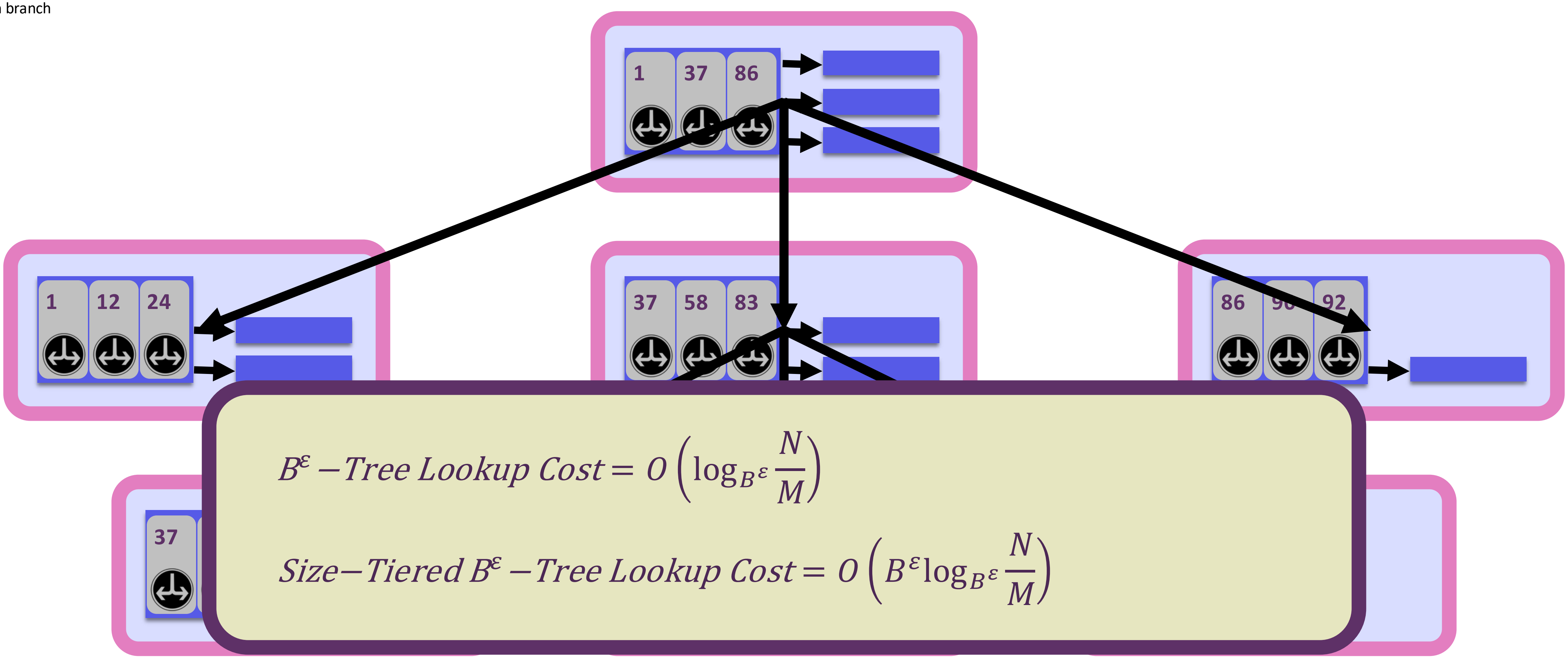
Lookups in a STB$^{\varepsilon}$-tree are like lookups in a B$^{\varepsilon}$-tree, except they must check each branch



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)

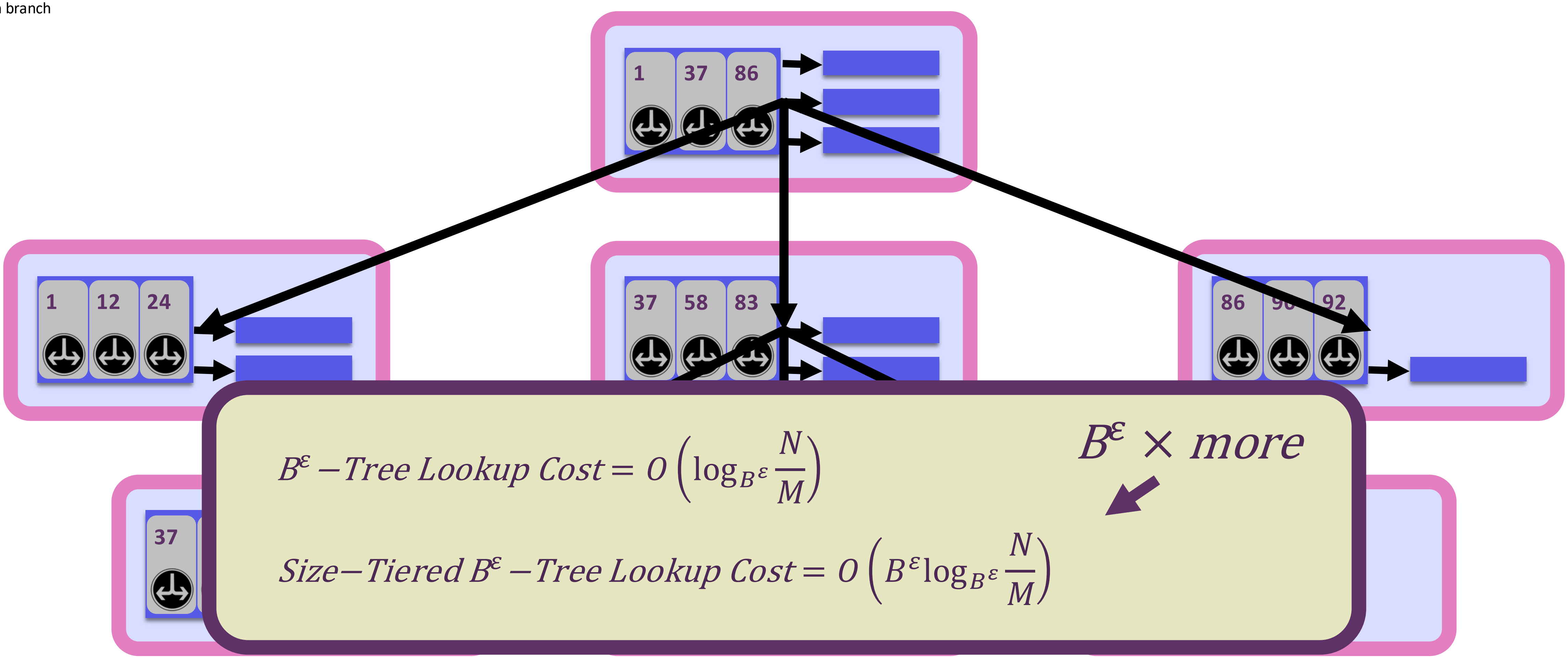


$$B^{\varepsilon} - Tree\ Lookup\ Cost = O\left(\log_{B^{\varepsilon}} \frac{N}{M}\right)$$

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

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



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

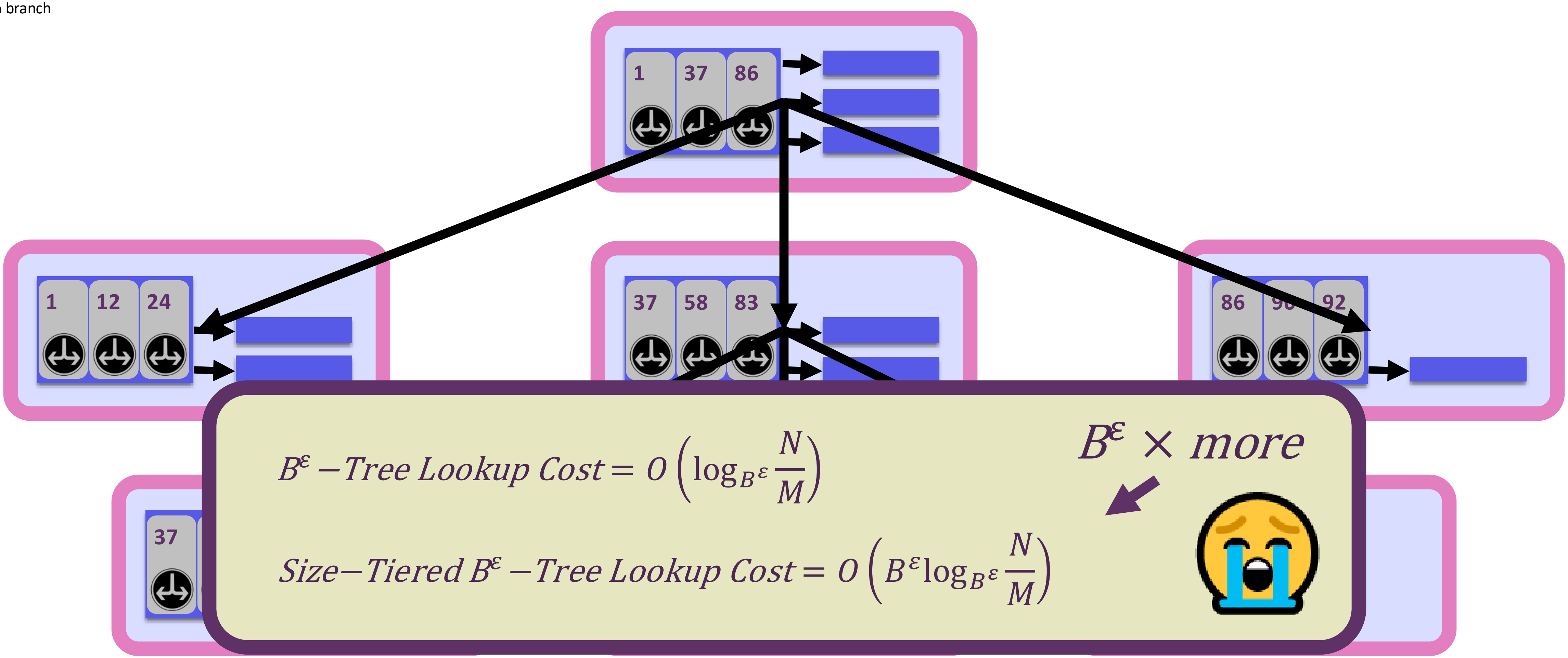
$B^{\varepsilon} \times more$

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

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

Query(71)

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



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

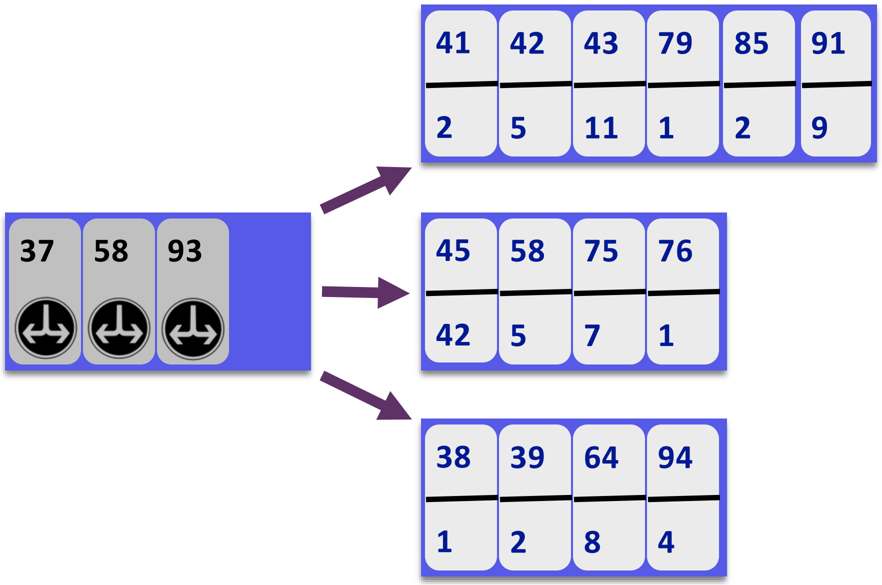
$$Size{-}Tiered\ B^{\varepsilon} - Tree\ Lookup\ Cost = O\left(B^{\varepsilon} \log_{B^{\varepsilon}} \frac{N}{M}\right)$$

$B^{\varepsilon} \times more$

Fixing Lookups (almost)

Fixing Lookups (almost)

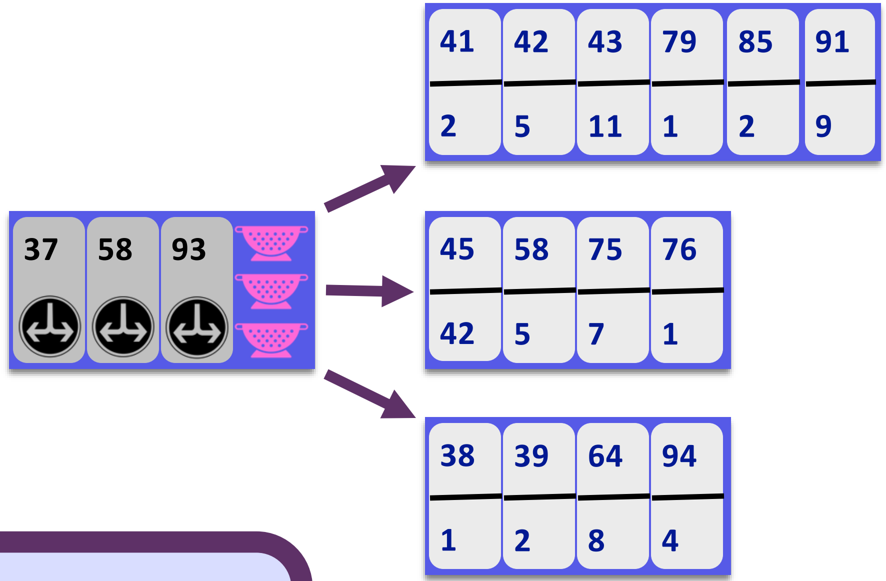
The problem is that each node has multiple branches

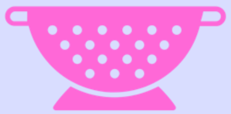


Fixing Lookups (almost)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



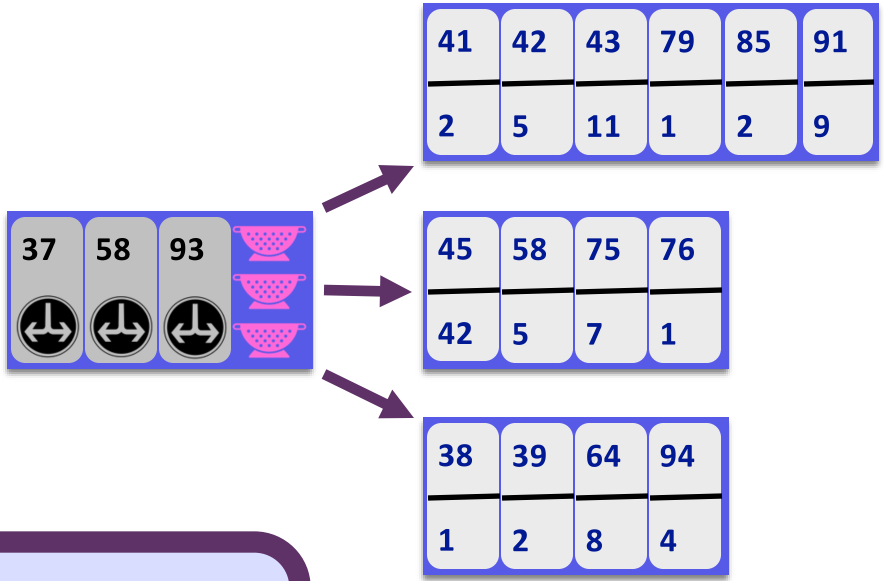


A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

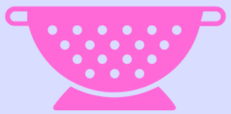
Fixing Lookups (almost)

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

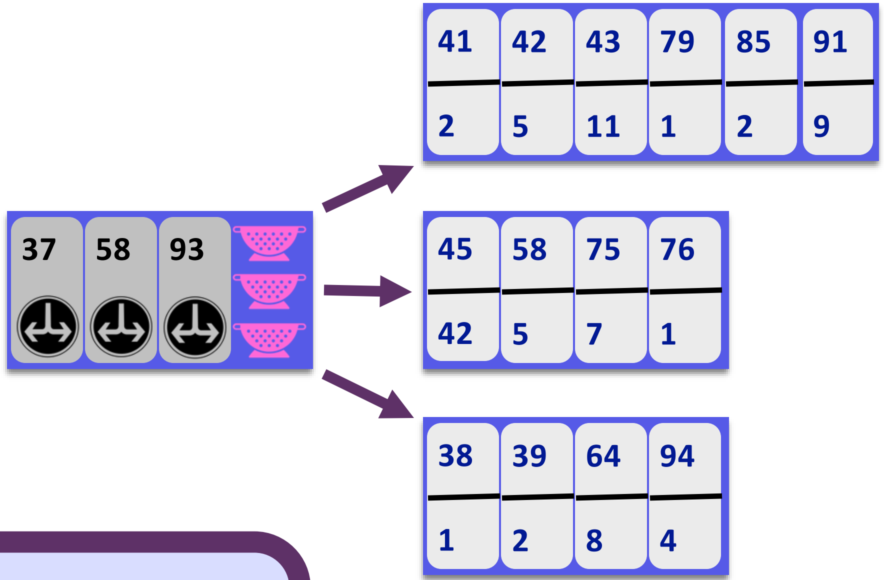
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

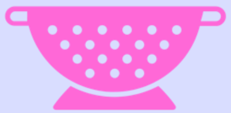
Query(64)

The problem is that each node has multiple branches

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

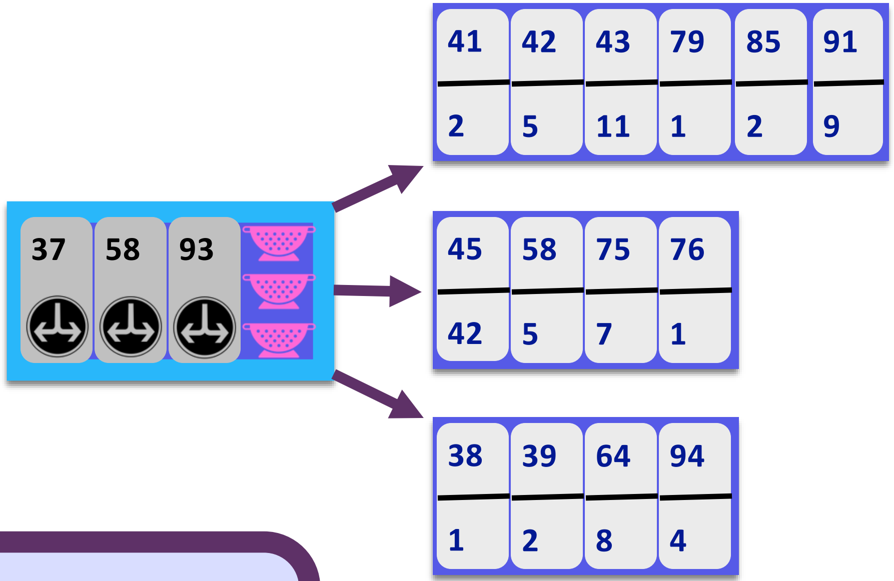
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

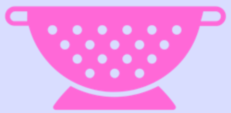
Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



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

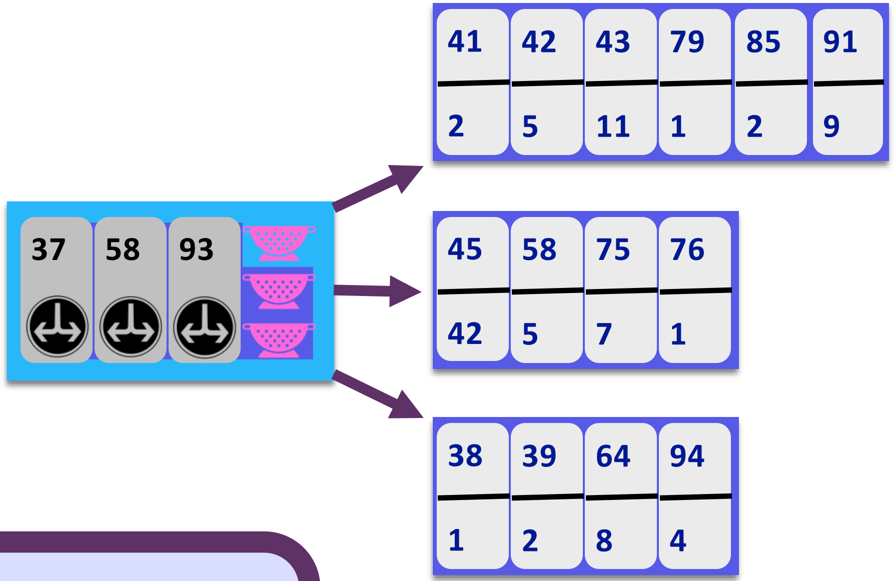
Examples: Bloom, cuckoo, quotient

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Fixing Lookups (almost)

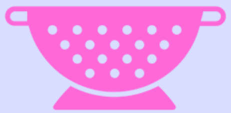
Query(64)

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



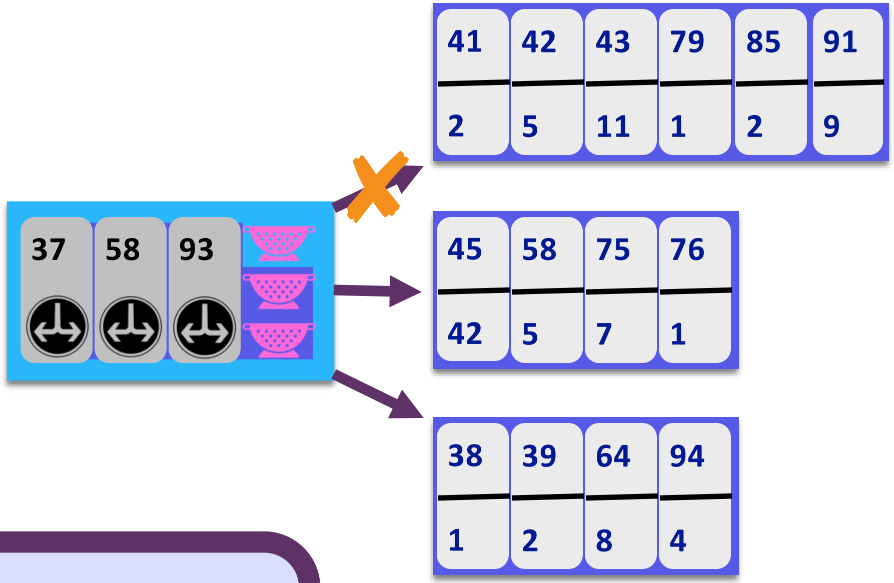
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

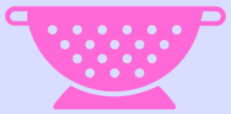
Query(64)

The problem is that each node has multiple branches



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

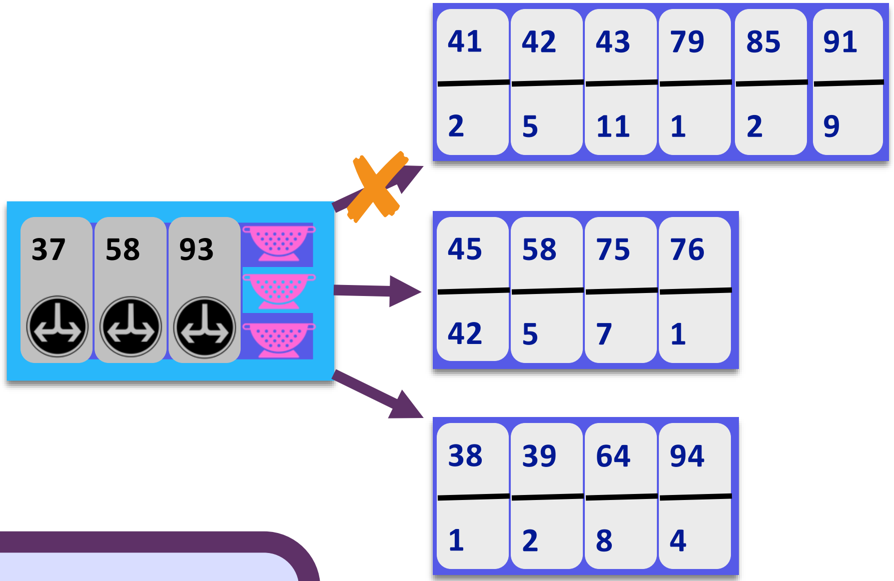
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

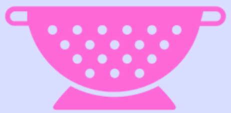
Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



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



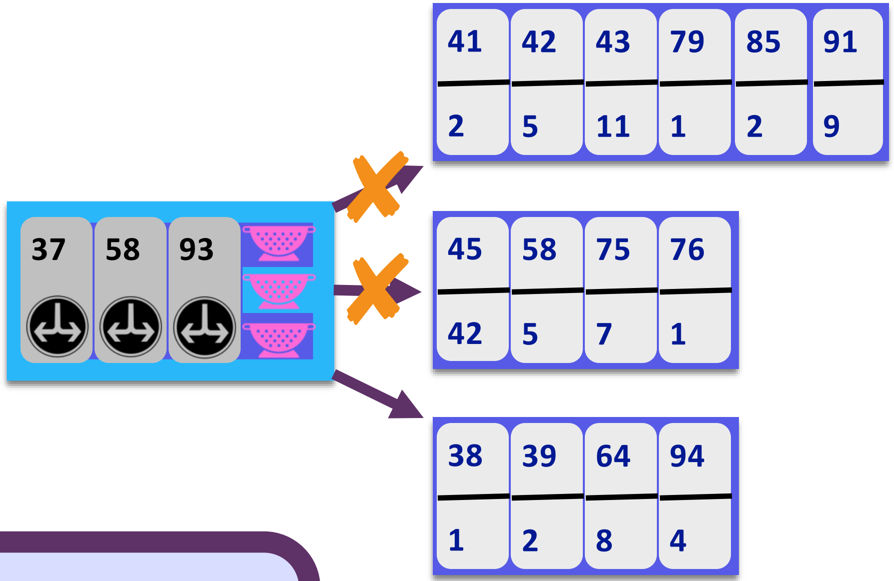
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

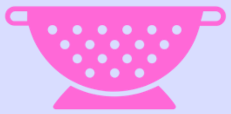
Query(64)

The problem is that each node has multiple branches



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

Idea: use filters to avoid searching them



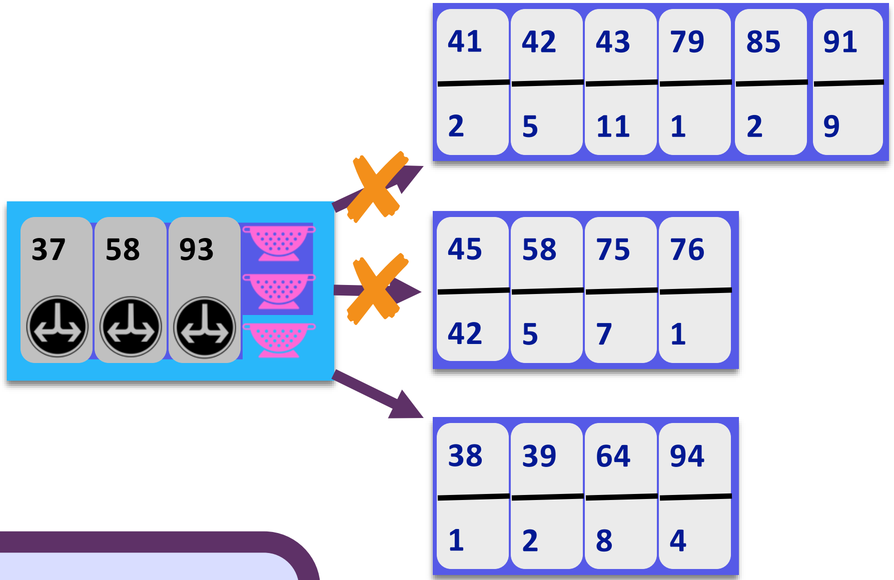
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

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

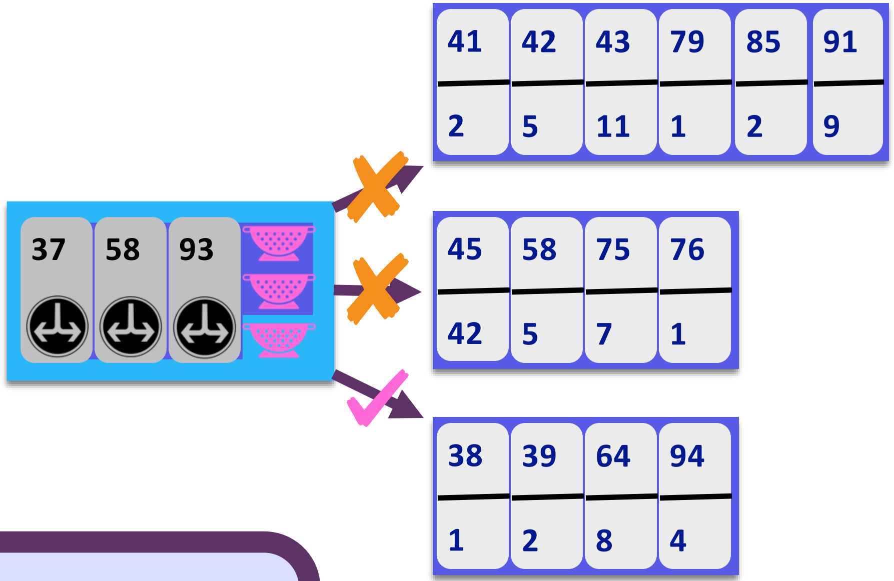
Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64)

The problem is that each node has multiple branches

Idea: use filters to avoid searching them



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

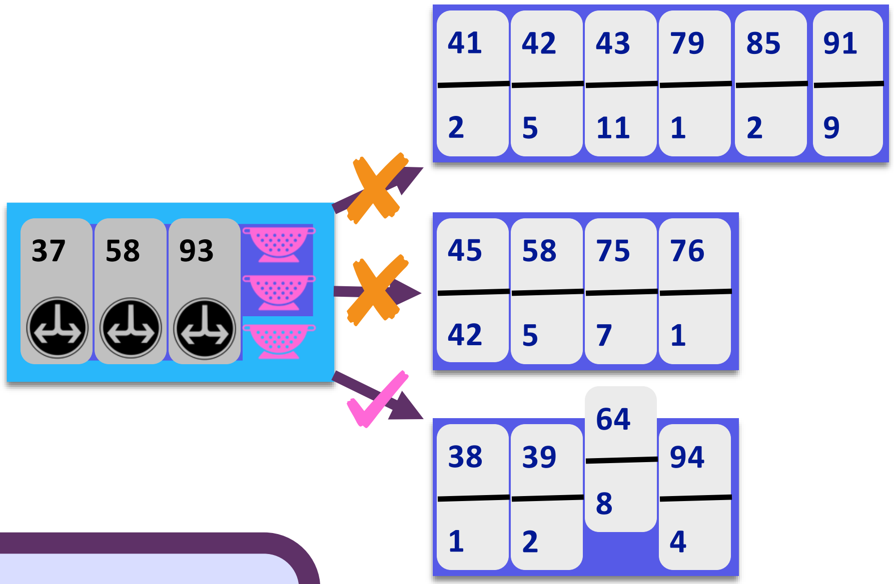
Examples: Bloom, cuckoo, quotient

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)

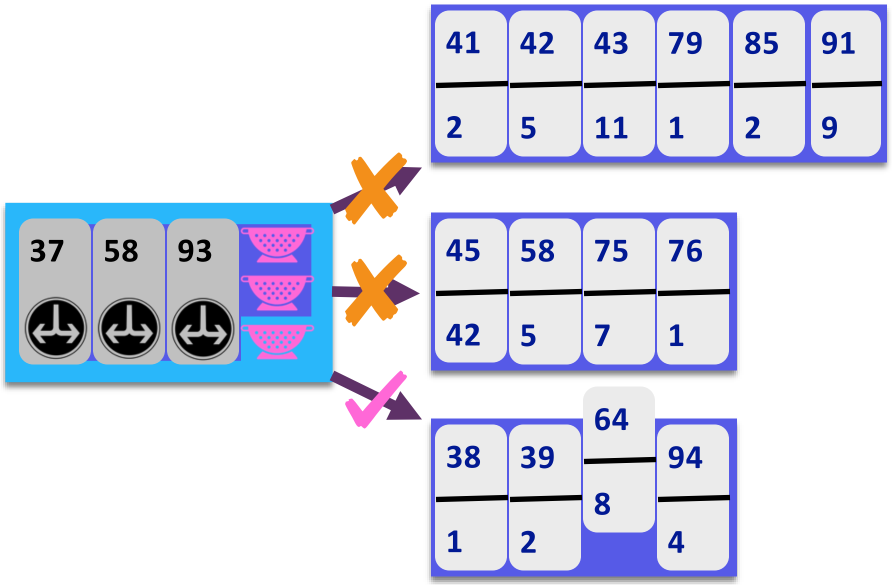
A filter is a probabilistic data structure with answers membership with no false negatives

Examples: Bloom, cuckoo, quotient

Fixing Lookups (almost)

Query(64) ⟶ 8

The problem is that each node has multiple branches



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

Idea: use filters to avoid searching them

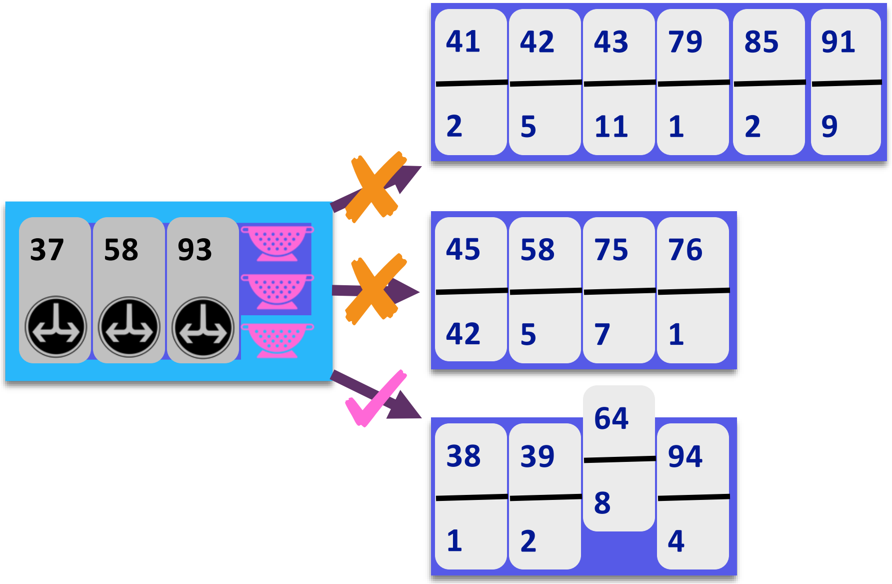
$$False\ Positive\ Rate \leq O\left(\frac{\varepsilon}{B^{\varepsilon}\log_B N}\right)$$

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)

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

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