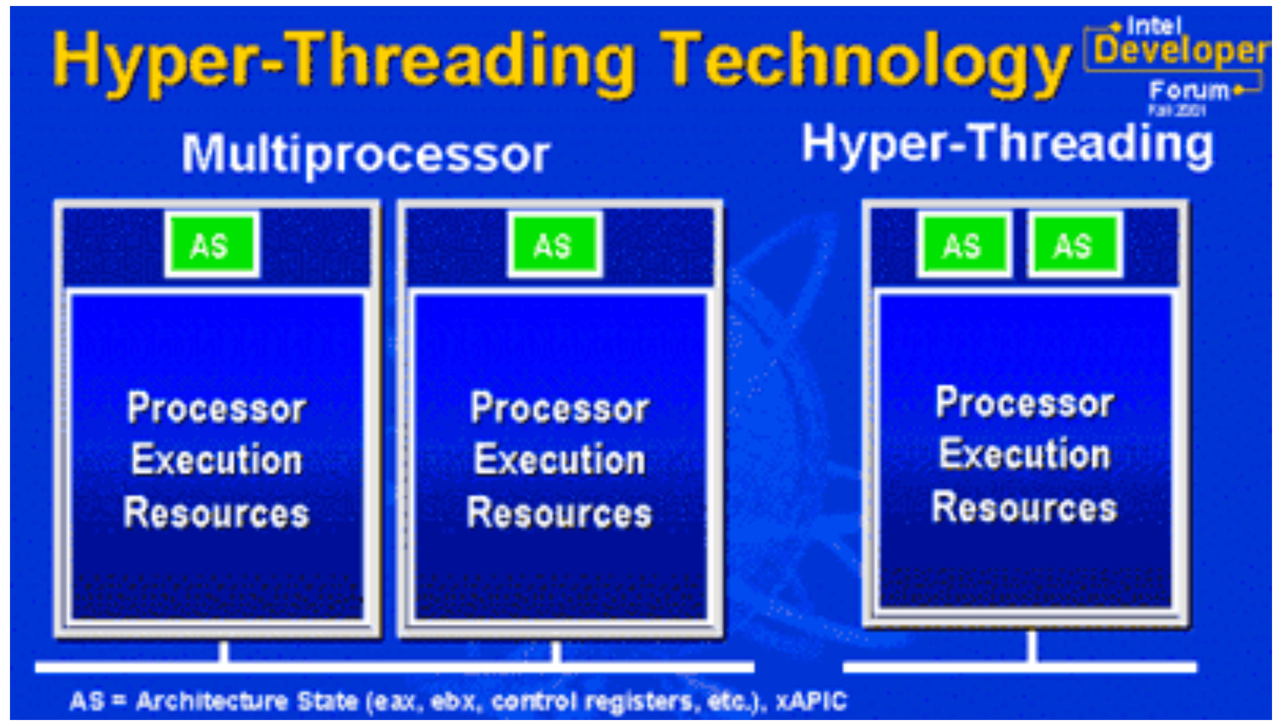
Hyperthreading

A type of **simultaneous multithreading** (SMT) - hyperthreading is the proprietary Intel version.

- SMT improves parallelization by making **two virtual (logical) cores** for each physical core by duplicating architectural state (e.g., registers) but not the main resources (e.g., ALUs)
- The first hyperthreading implementation was reported (by Intel) to use ~5% more area for 15-30% better performance.
- Performance gains are very dependent on whether the application is **memory or compute bound**.

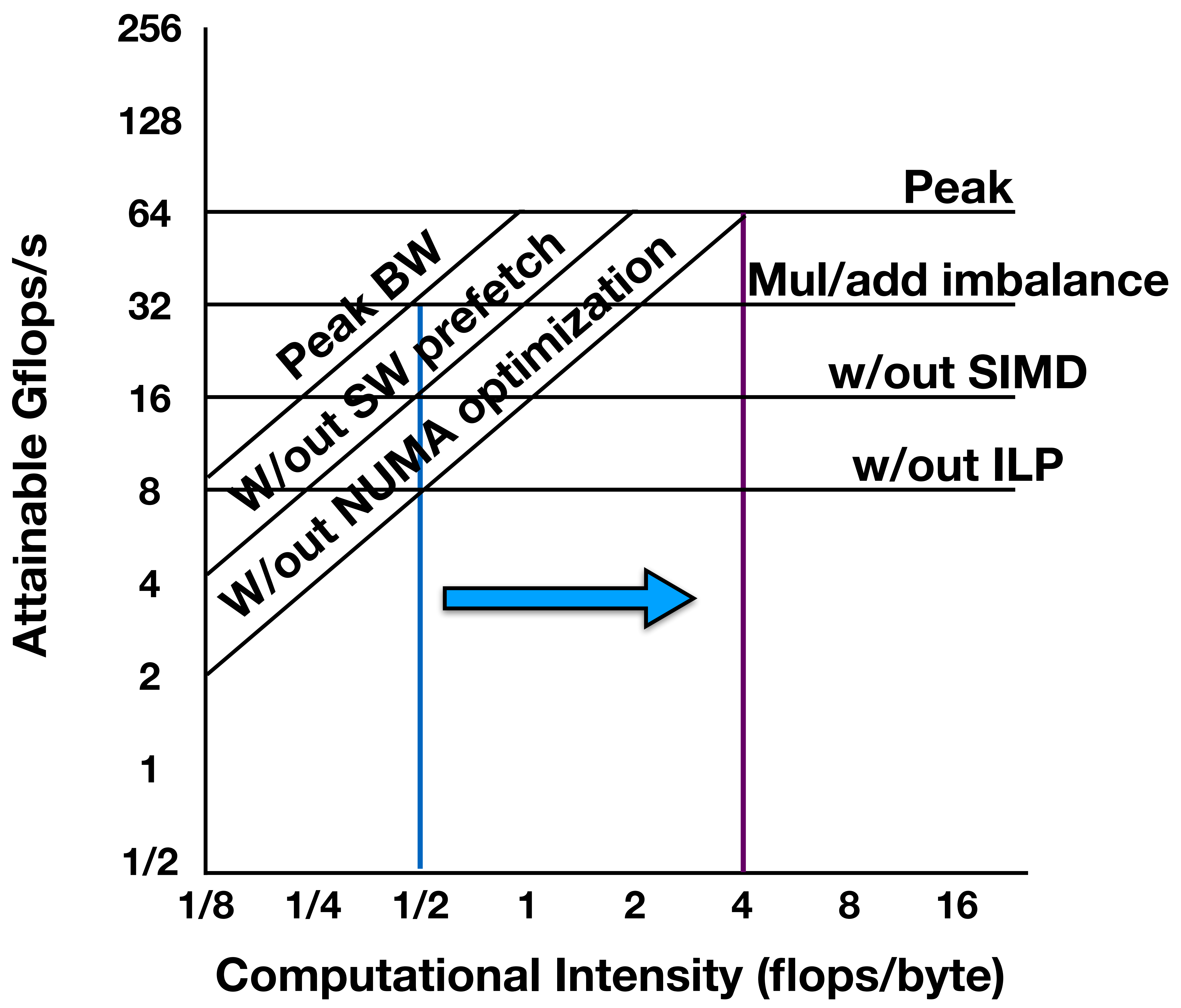


https://www.anandtech.com/show/868/3

https://www.extremetech.com/computing/133121-maximized-performance-comparing-the-effects-of-hyper-threading-software-updates

Effective Roofline (before and after)

Example machine:



Before optimization, traffic, and limited bandwidth - **performance is limited** to a very narrow window.

After optimization, ideally, performance is significantly better.

Roofline Example: Parallel loop

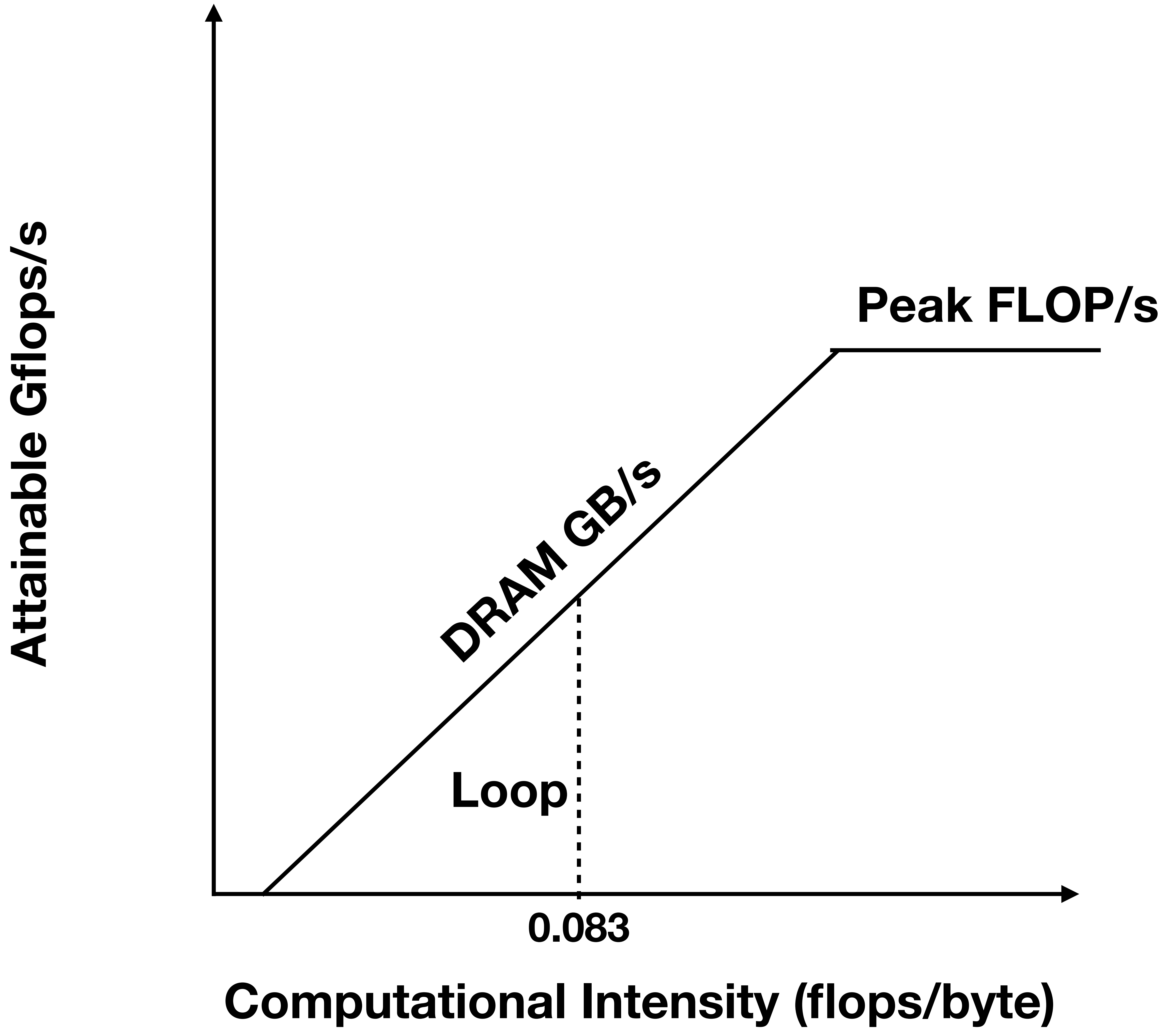
```
parallel_for (i=0;i<N;i++) {
  Z[i] = X[i] + alpha*Y[i];
}
```

2 flops, 3 memory references (2 reads and 1 write)

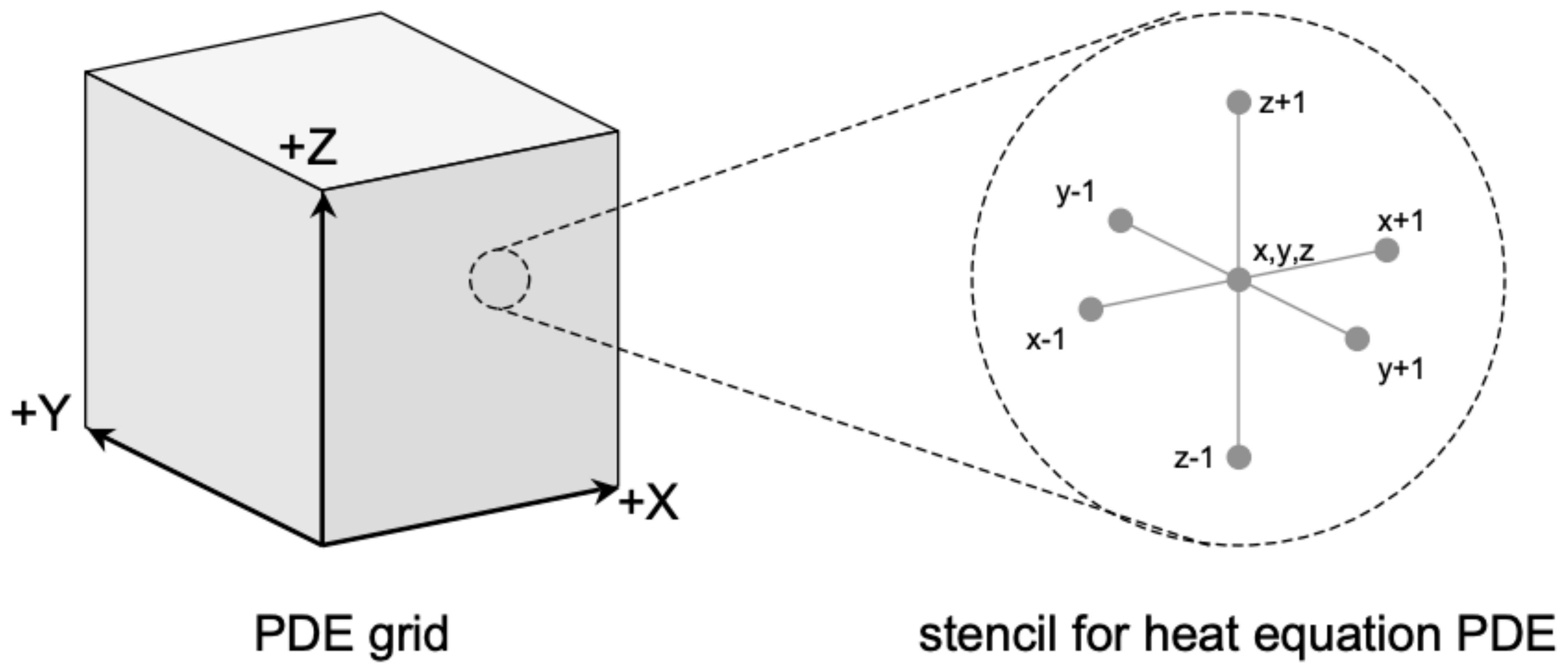
3 elements at 8 bytes each = 24 bytes

Intensity = flops / bytes = 2/24 ~ 0.083

Roofline Example: Parallel loop



Roofline Example: Heat equation stencil



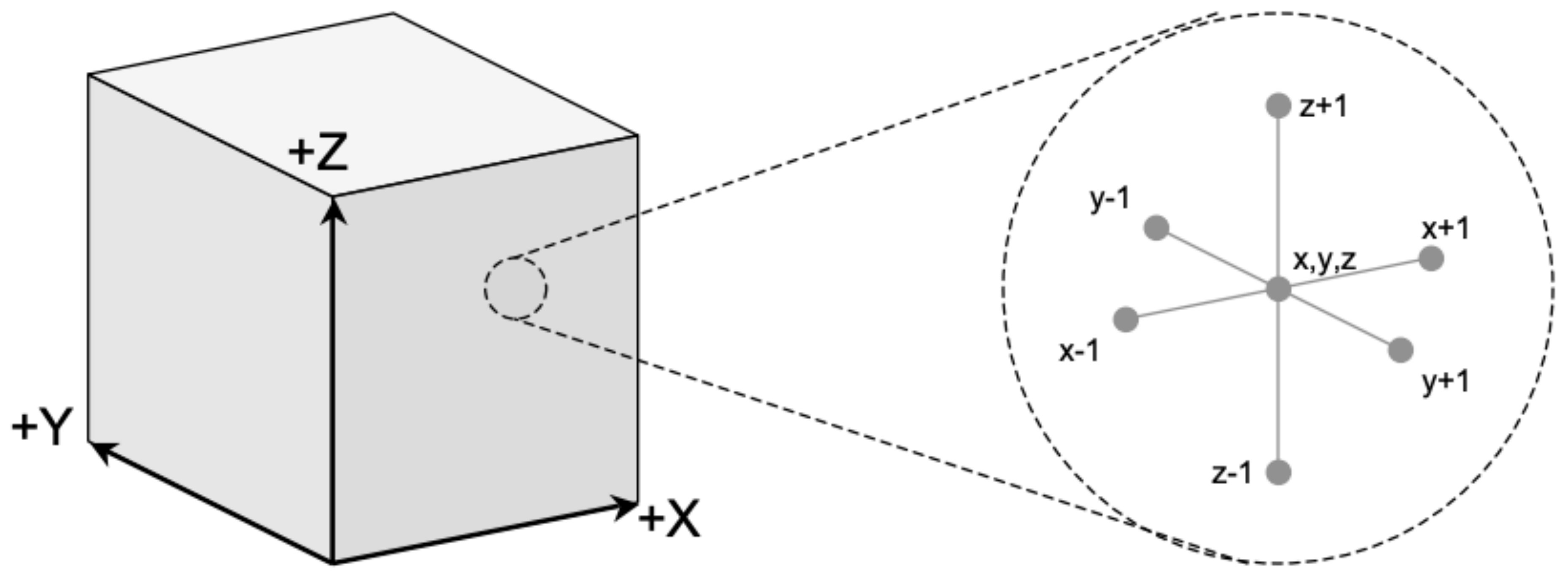

```
For x, y, z in 0 to n-1
  next[x,y,z] =
    C0 * current[x,y,z] +
    C1 * (current[x-1, y, z] +
            current[x+1, y, z] +
            current[x, y-1, z] +
            current[x, y+1, z] +
            current[x, y, z-1] +
            current[x, y, z+1]);
```

A 7-point constant coefficient stencil:

8 flops, 8 memory references (7 reads, 1 write) per point

CI = 0.125 flops / byte

Roofline Example: Heat equation stencil



PDE grid

stencil for heat equation PDE

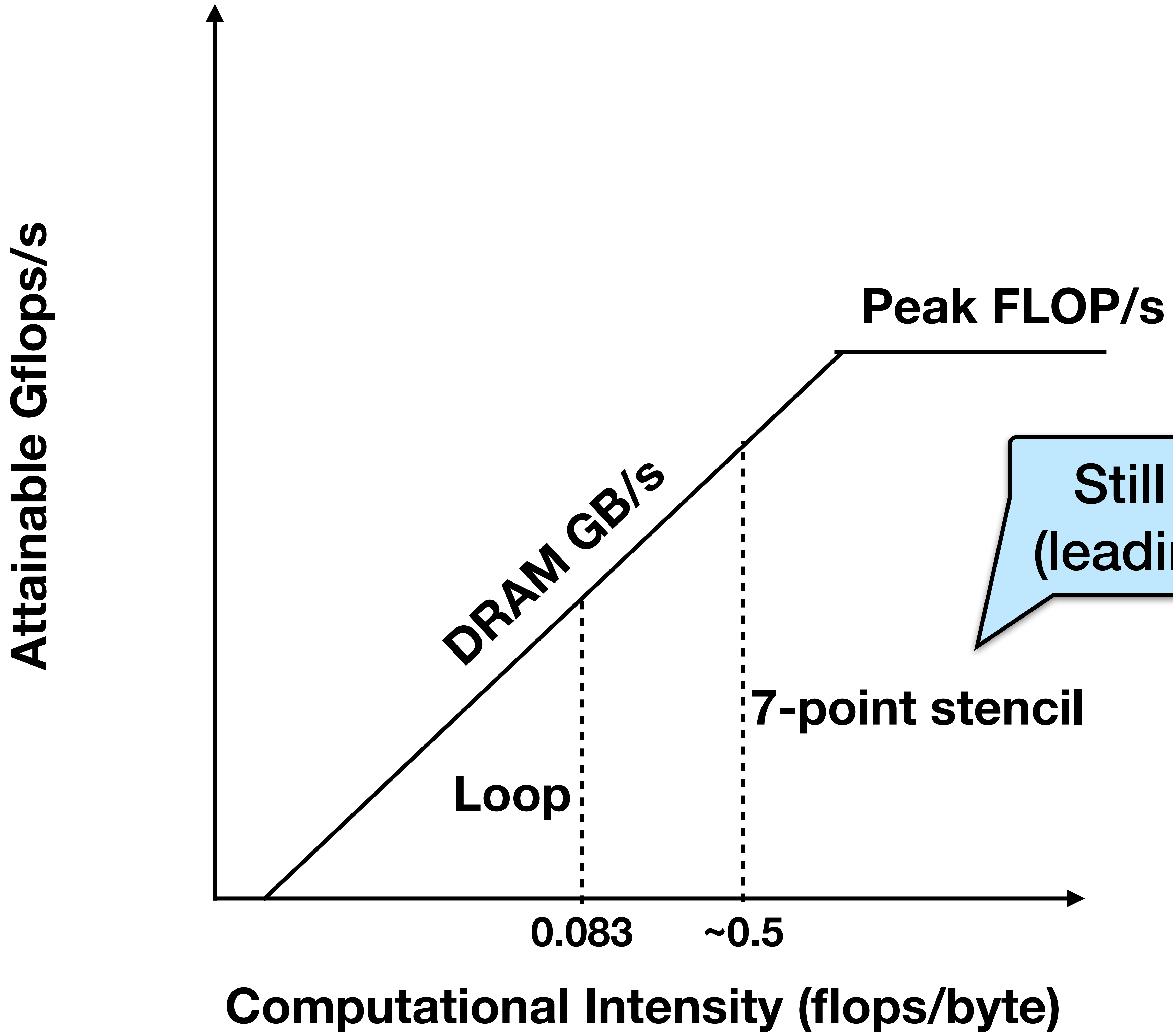
```
For x, y, z in 0 to n-1
  next[x,y,z] =
    C0 * current[x,y,z] +
    C1 * (current[x-1, y, z] +
          current[x+1, y, z] +
          current[x, y-1, z] +
          current[x, y+1, z] +
          current[x, y, z-1] +
          current[x, y, z+1]);
```

A 7-point constant coefficient stencil:

- 8 flops, 8 memory references (7 reads, 1 write) per point
- CI = 0.125 flops / byte

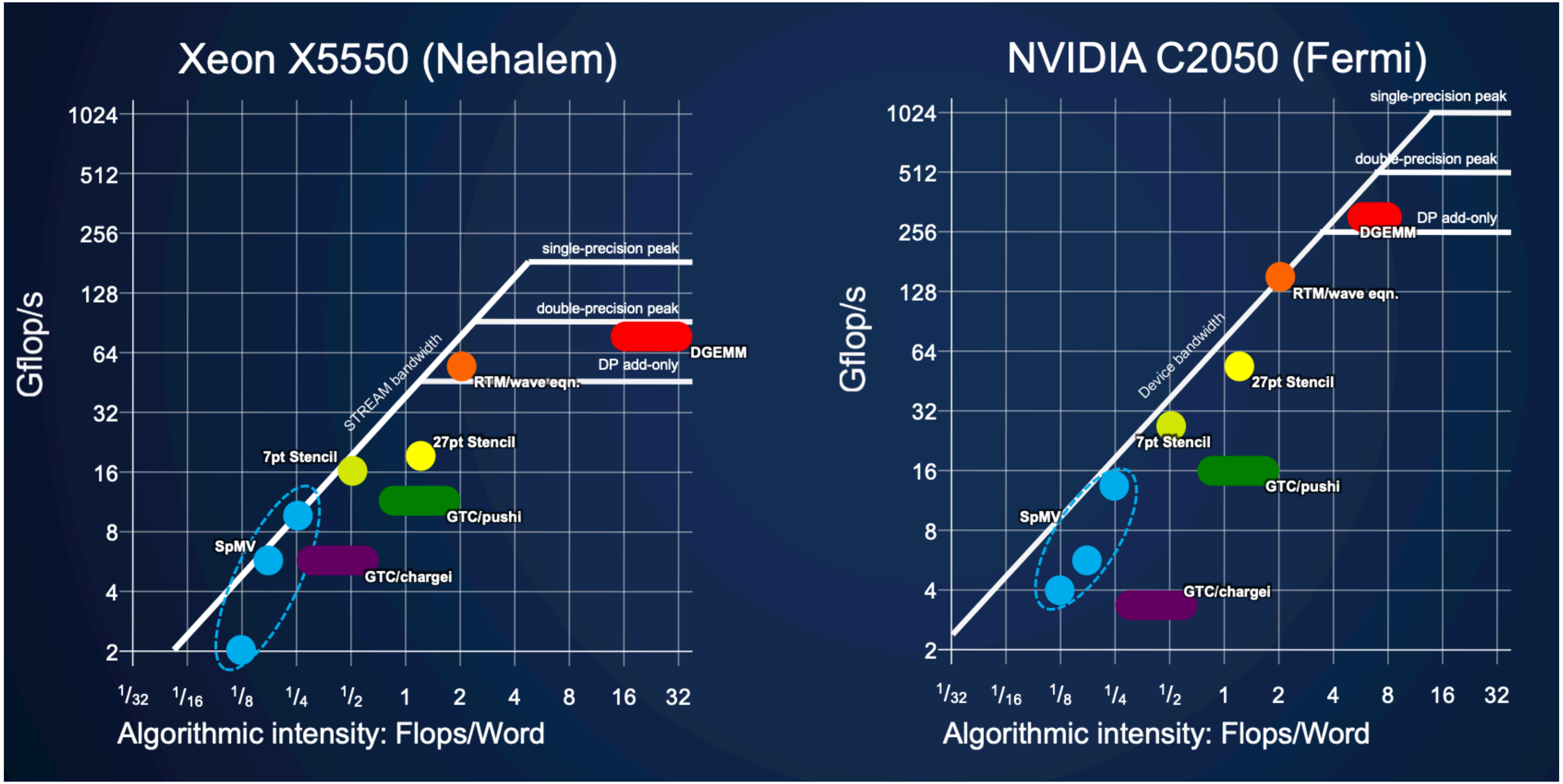
→ Cache blocking can filter out accesses to DRAM and increase the effective CI to close to 0.5

Roofline Example: Heat equation stencil



Still O(1) flops/byte, but (leading) **constants matter**

Roofline across algorithms



Work by Williams, Oliver, Shalf, Madduri, Kamil, Im, Ethier, …

Summary

Roofline captures upper bound performance with the **min of 2 upper bounds** of the machine:

- Peak flops
- Peak memory bandwidth

Computational / Arithmetic **intensity** is a key part of the model.

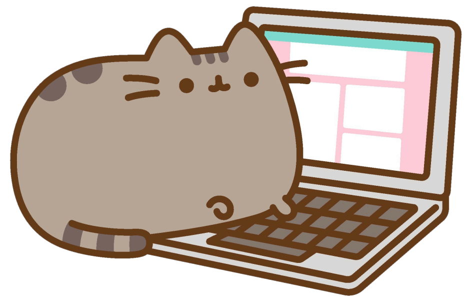
- Usually defined as best case

Originally for single processors and shared-memory machines.

Widely **used in practice** and adapted to any bandwidth/compute limited situation.

CSE 6230:
HPC Tools and Applications

+



Lecture 4: I/O-efficient Data Structures (Trees, Skip Lists, and Tries)

Helen Xu

hxu615@gatech.edu

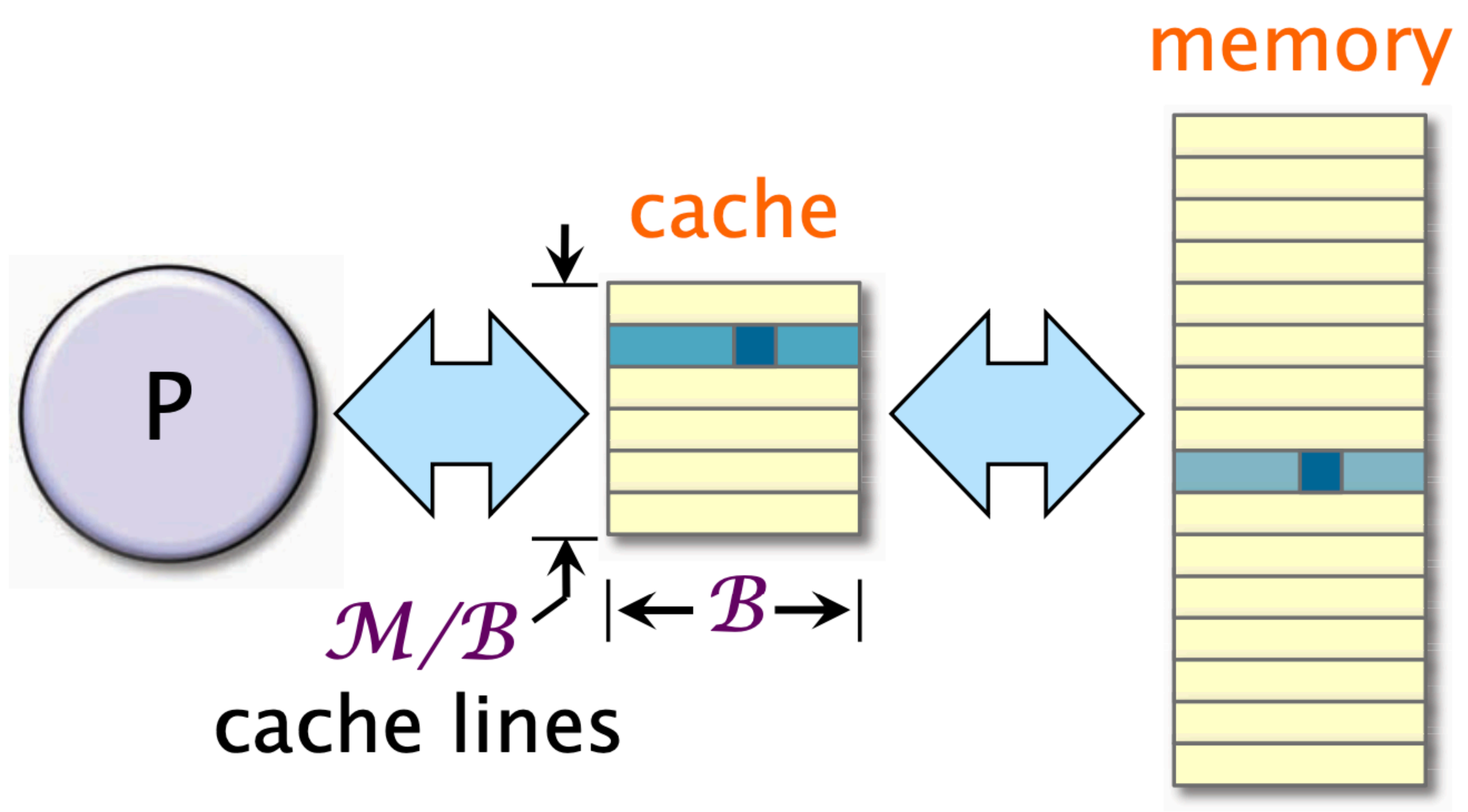
Georgia Tech College of Computing
School of Computational Science and Engineering

Acknowlegement: some slides from University of Utah CS 6530

Recall: Ideal-Cache Model

Parameters

- Two-level hierarchy
- Cache size of *M* bytes
- Cache-line length of *B* bytes
- Fully associative
- Optimal, omniscient replacement.



Performance Measures

- **Work** W (ordinary running time)
- **Cache misses** Q (number of cache lines that need to be transferred between cache and memory)

Dictionary data structures

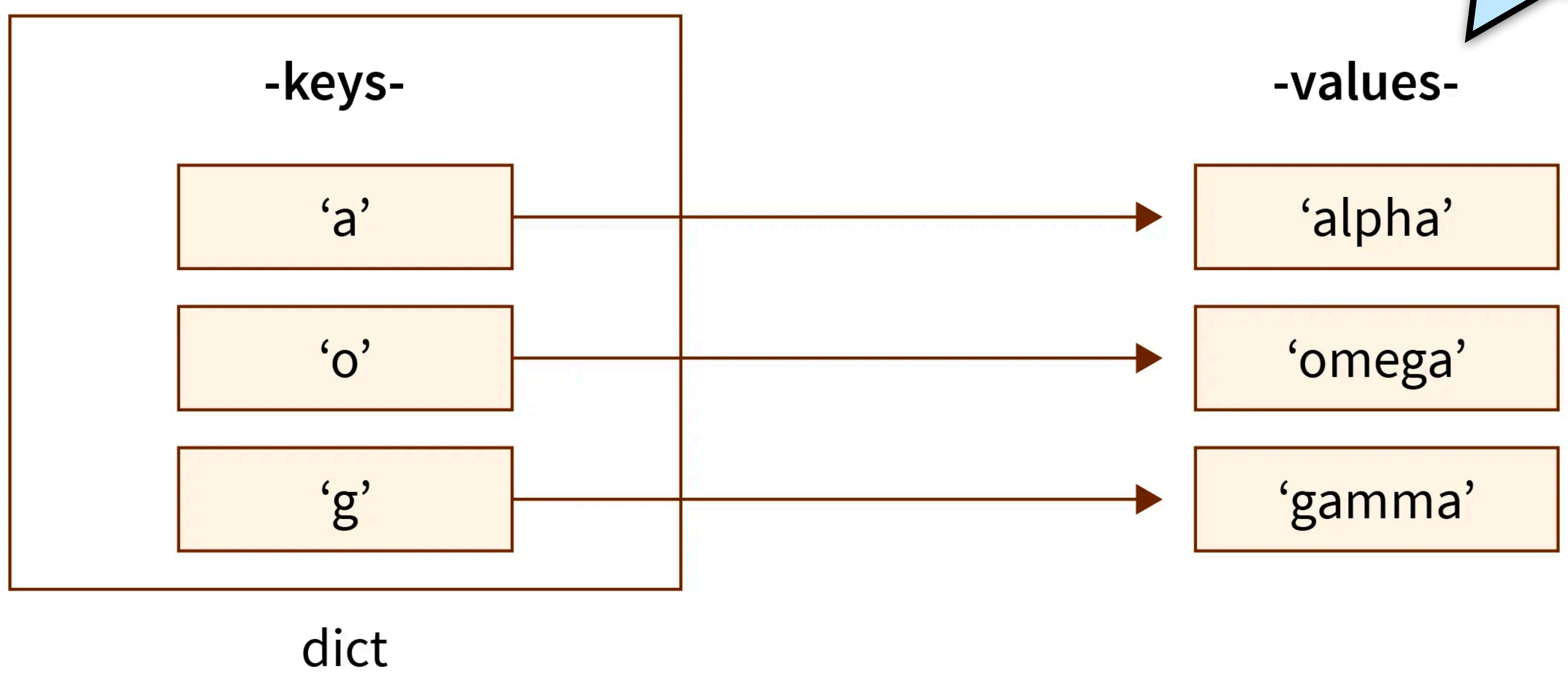
A **dictionary data structure** is a general-purpose data structure for supporting a group of objects.

Dynamic dictionaries typically support the following operations:

- **Search** for the existence of an element
- **Insert** an element
- **Remove** an element

Key-value stores map keys to values

If just keys without values, it is a **key store**



https://www.scaler.com/topics/dictionary-in-data-structure/
https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Dictionaries

Review: Self-balancing binary search trees

Self-balancing binary search trees support the basic dynamic-dictionary operations (search, insert, delete) in $O(\log n)$ time.

Examples include: red-black trees, AVL trees, etc.

For a more in-depth review, see CLRS chapters 12-13 (pdf on Canvas).

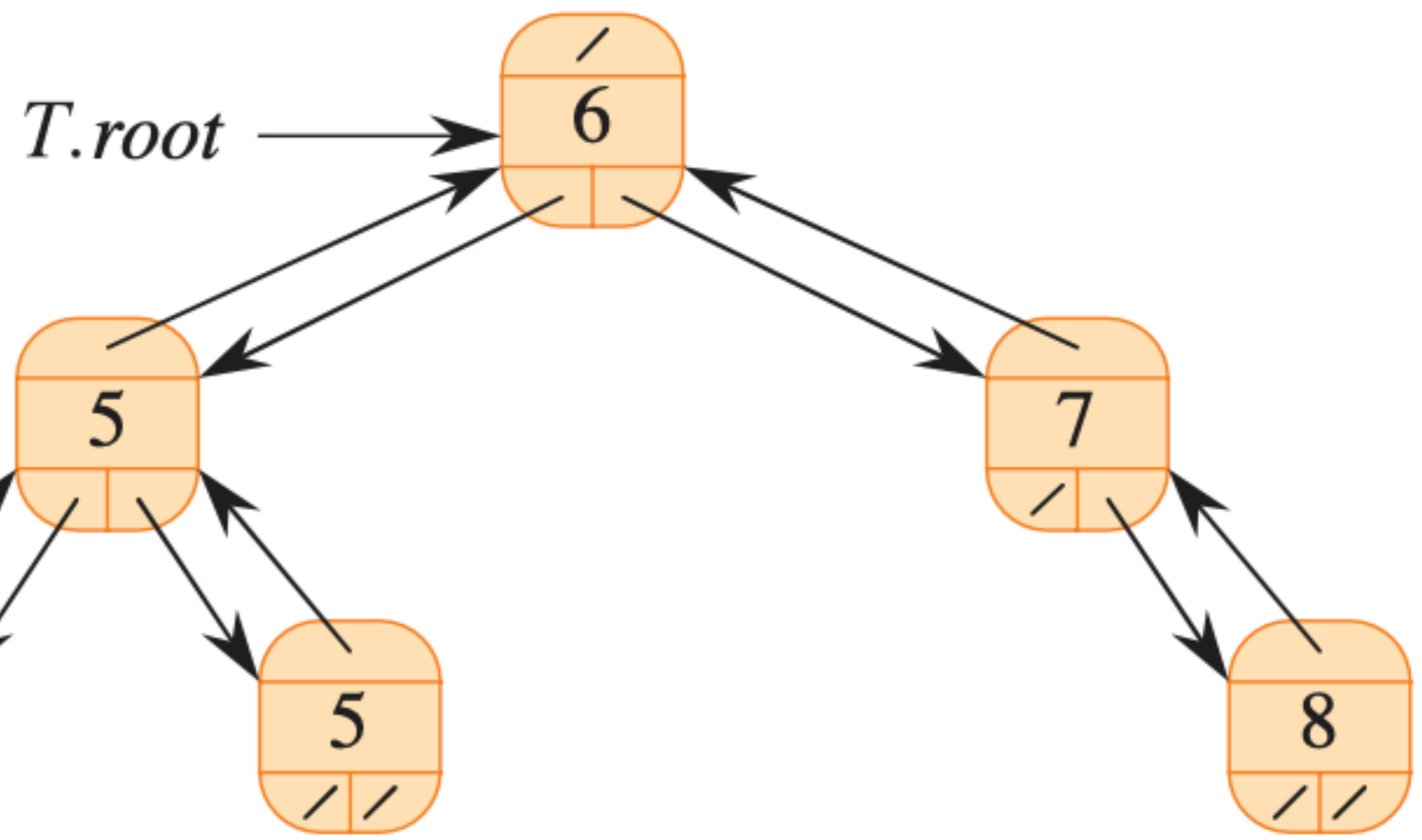


Diagram from CLRS

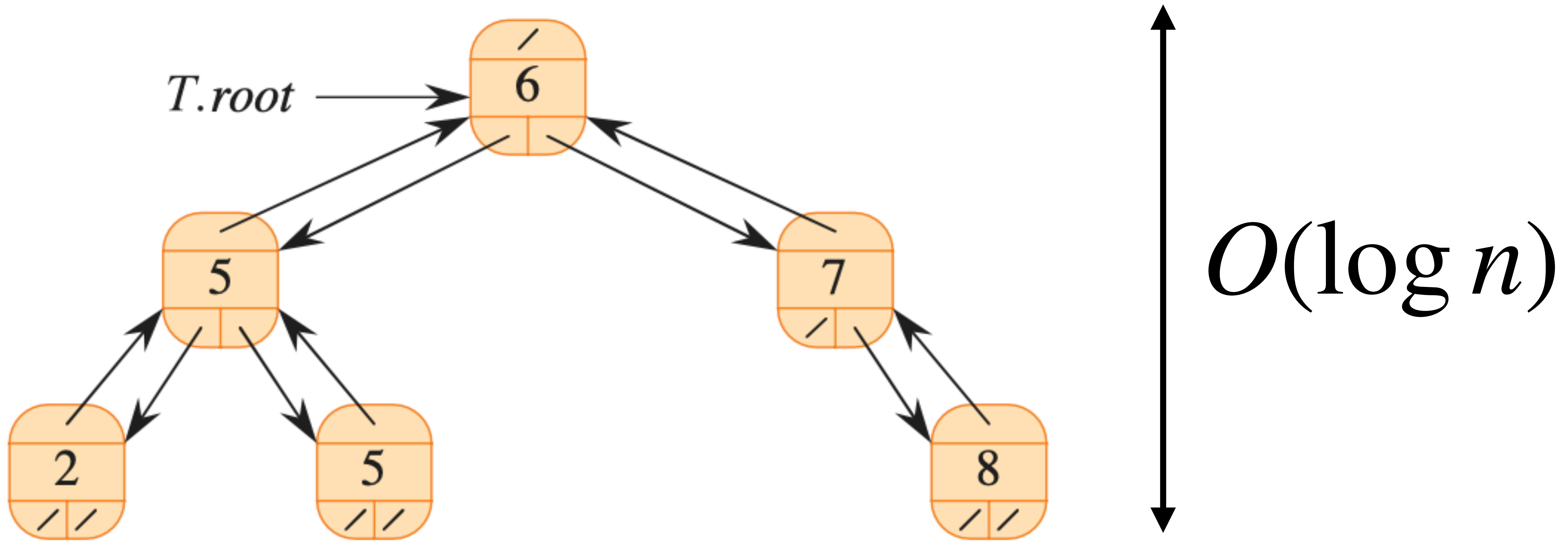
Warmup Question

How many **cache misses** does it take to search/insert/delete in a balanced binary tree?

Cache misses in binary trees

A balanced binary tree follows one pointer and therefore incurs one cache miss per level.

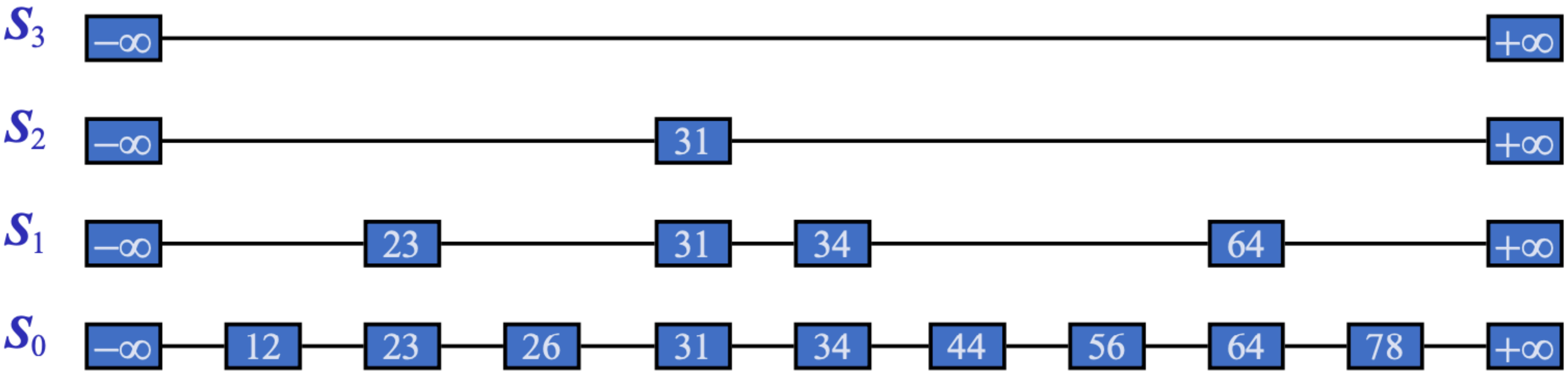
The height of the tree is $O(\log n)$, so every operation takes $O(\log n)$ cache misses.



Skip lists

What is a skip list?

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys
- Skip lists are one way to implement the dictionary

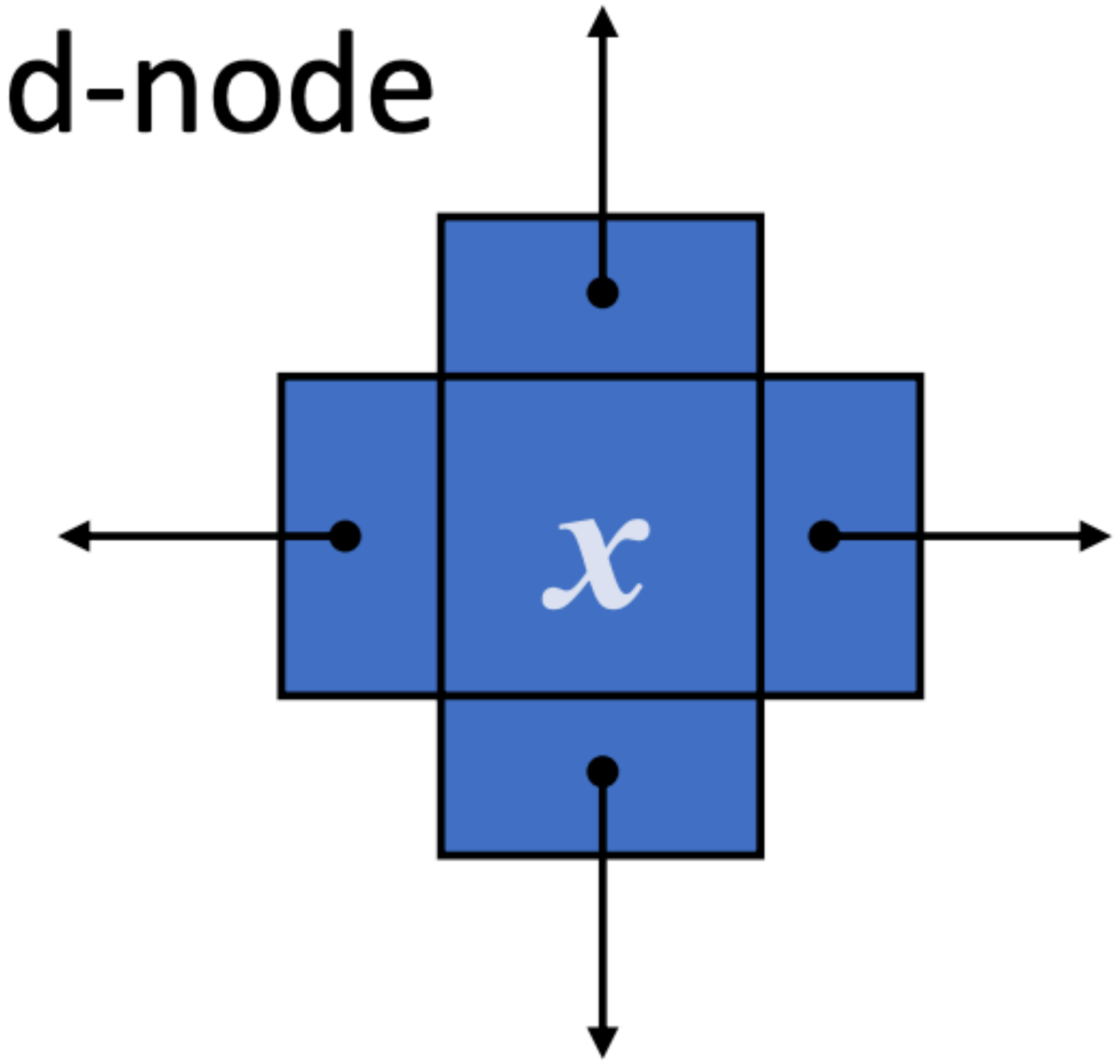


Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Skip list implementation

- We can implement a skip list with quad-nodes
- A quad-node stores:
  - item
  - link to the node before
  - link to the node after
  - link to the node below
- Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them

quad-node



Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

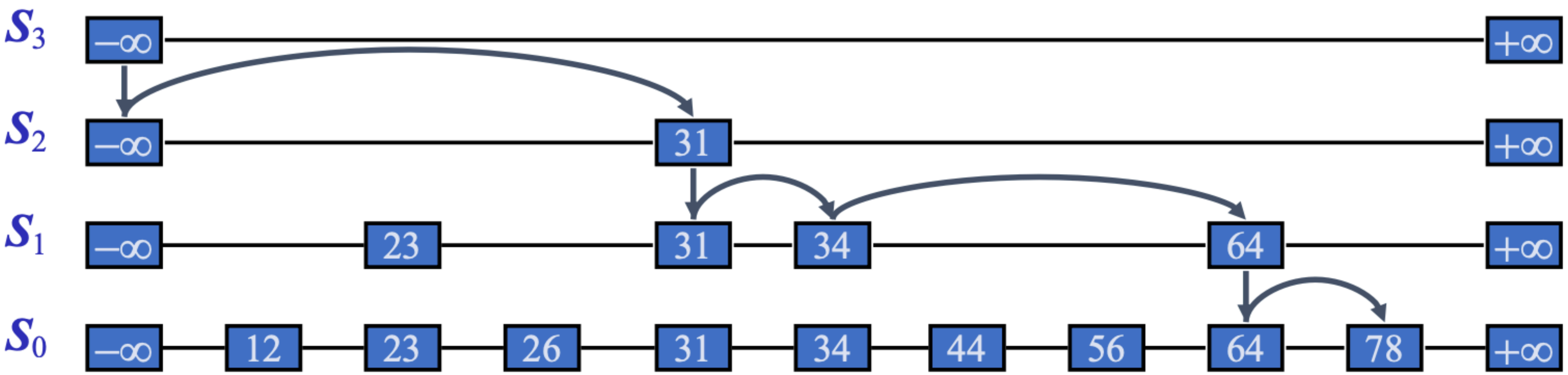
Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(after(p))$

    $x = y$: we return $element(after(p))$

    $x > y$: we "scan forward"

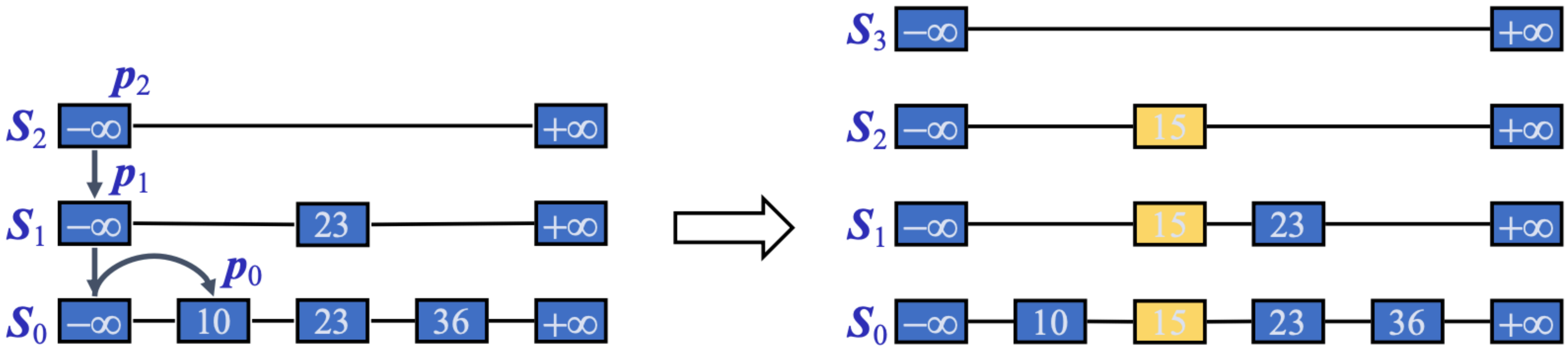
    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY
- Example: search for 78



Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Insertion

- To insert an item (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
  - For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Randomized algorithms

A **randomized algorithm** controls its execution through random selection (e.g., coin tosses).

It contains statements like:

$$\boldsymbol{b} \leftarrow \boldsymbol{randomBit}()$$

if $\boldsymbol{b} = 0$

  do A …

else $\{ \boldsymbol{b} = 1 \}$

  do B …

Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

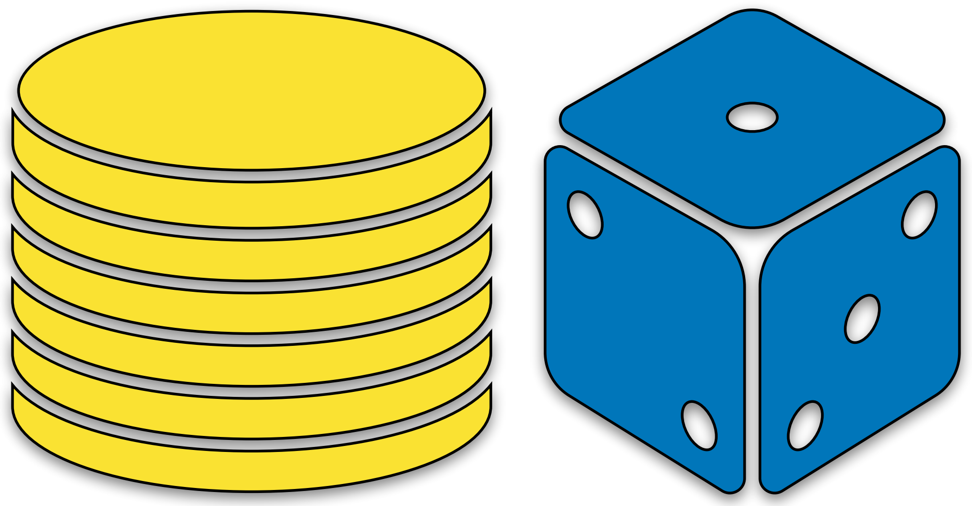
Analyzing randomized algorithms

The runtime of a randomized algorithm depends on the **outcomes of the coin tosses** (or dice rolls, etc.)

Through **probabilistic analysis**, we can derive the expected running time of a randomized algorithm.

We make the following assumptions in the analysis:

- the coins are **unbiased**, and
- the coin tosses are **independent**.



The worst-case running time is often large but has very low probability (e.g., it occurs when all coin tosses give “heads”).

Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Randomized algorithms and skip lists

When randomization is used in data structures, they are referred to as **probabilistic data structures**.

We use a randomized algorithm to insert (and delete) elements into a skip list in **expected** $O(\log n)$ time.

The height of a skip list is $O(\log n)$ **with high probability**.

Probability depends on a certain number n and goes to 1 as n goes to infinity

The **expected space usage** of a skip list is $O(n)$.

Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Question

Are binary trees and skip lists **optimal** in the ideal-cache model?

B-trees and B+-trees

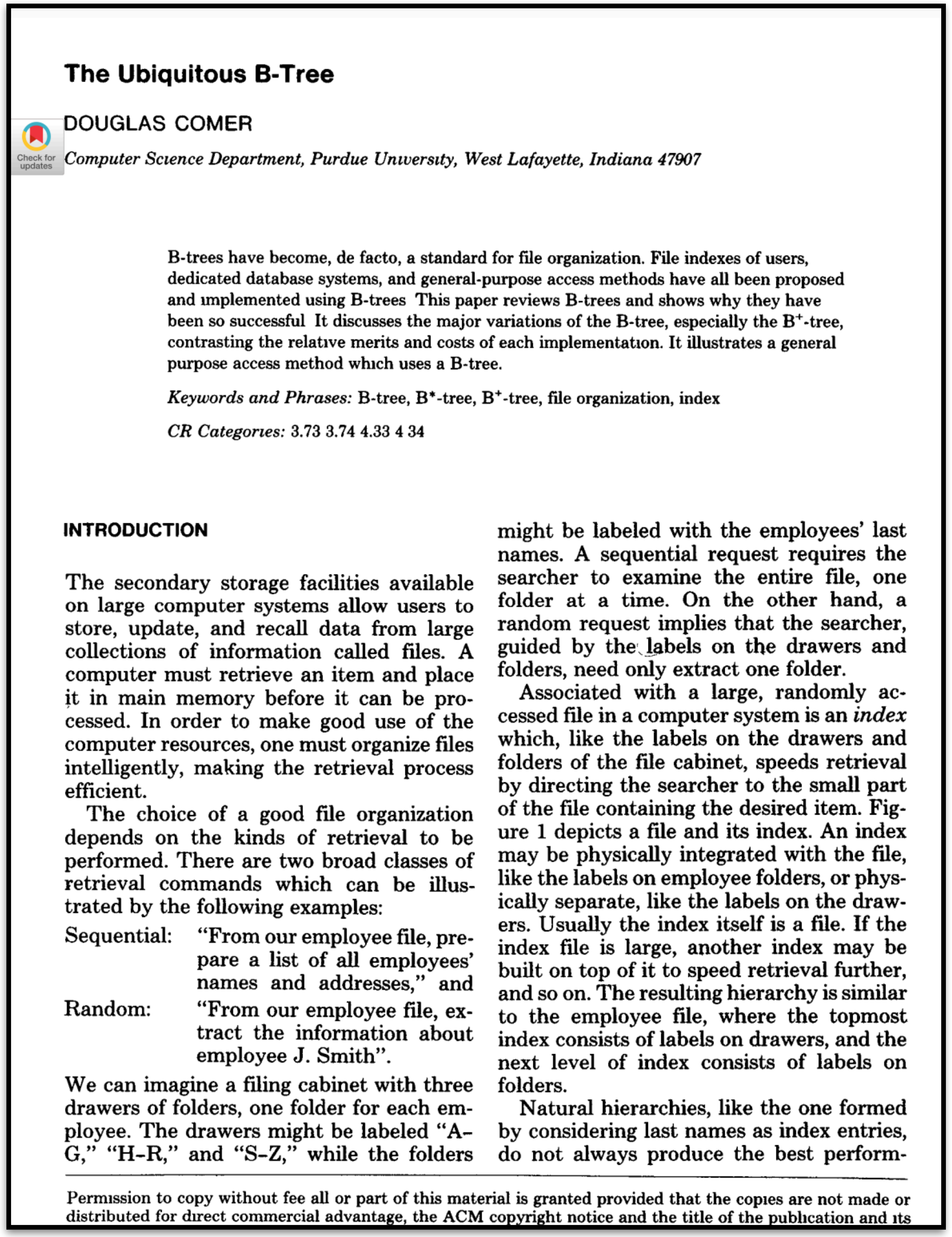
B-trees

A B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in $O(\log_B(N))$ cache-line misses in the ideal-cache model.

The fanout of the tree is **B**

Generalization of a binary search tree - a node can have more than 2 children

Optimized for systems that read/write large blocks of data



The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B$^+$-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B$^+$-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

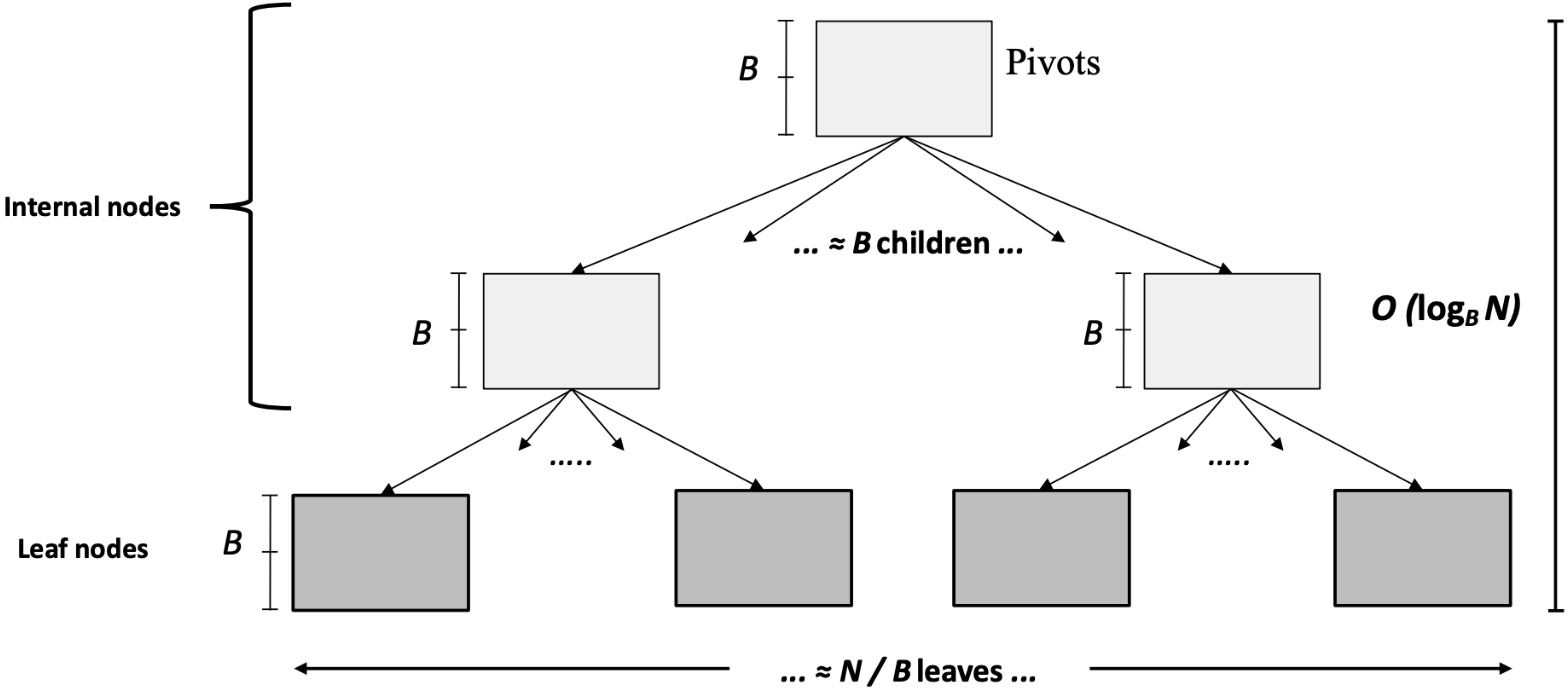
Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

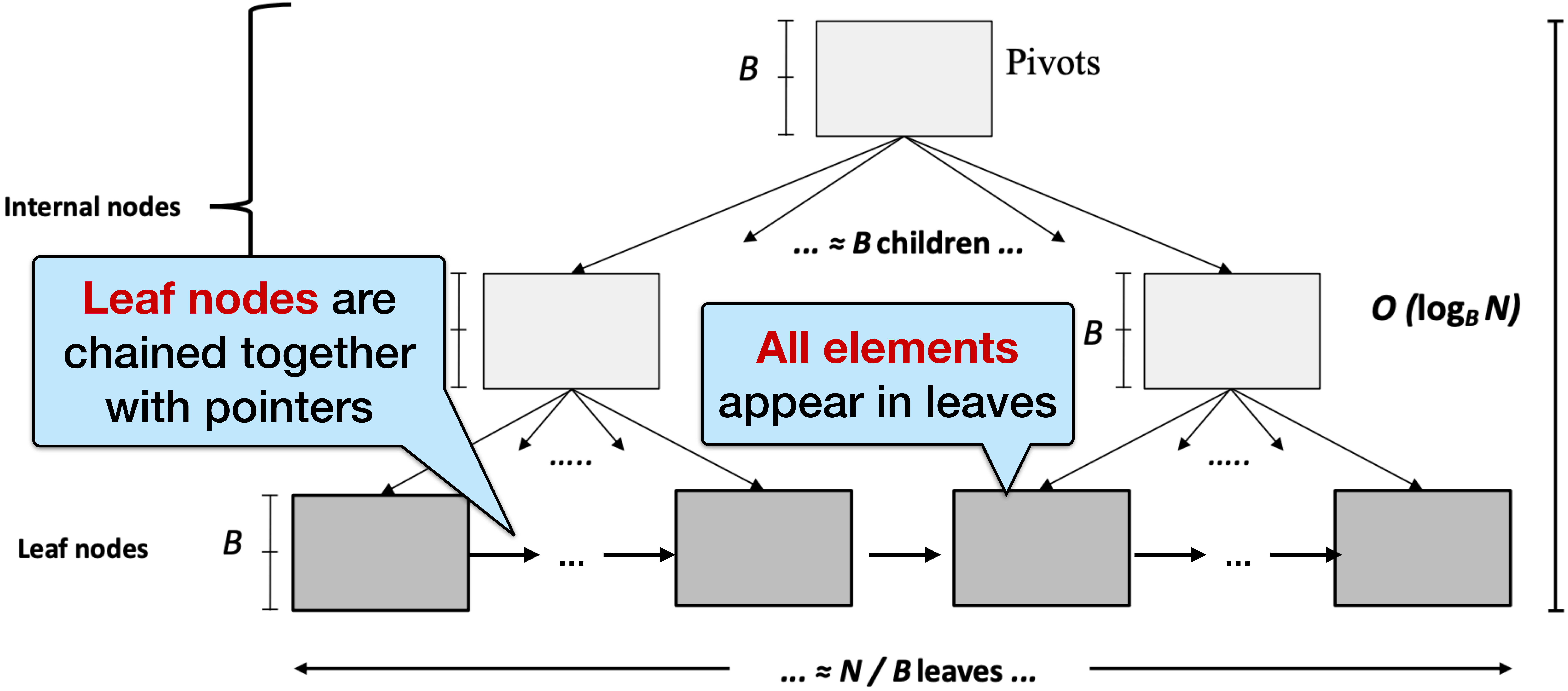
Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best perform-

B-tree structure



Often used in practice

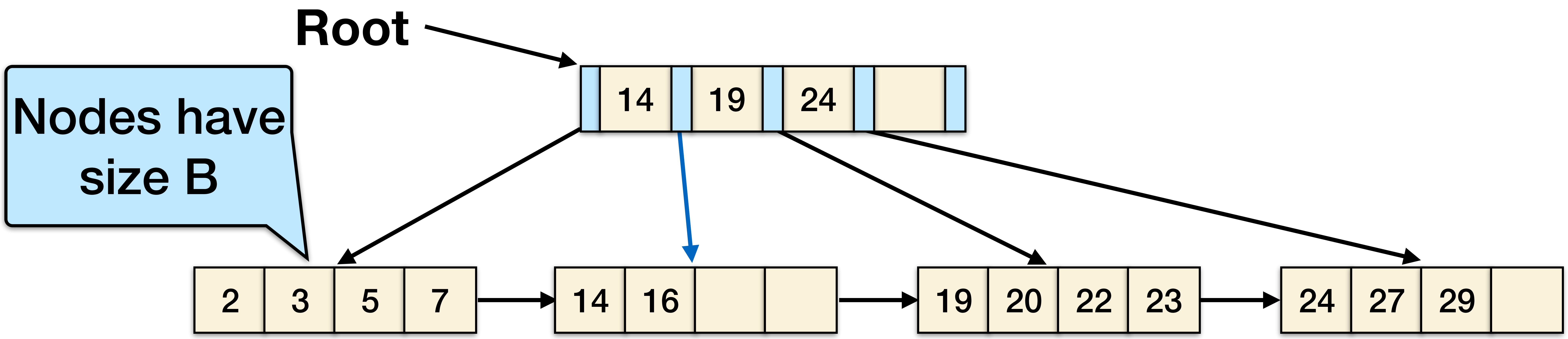
B+-tree structure



Search in B+-trees

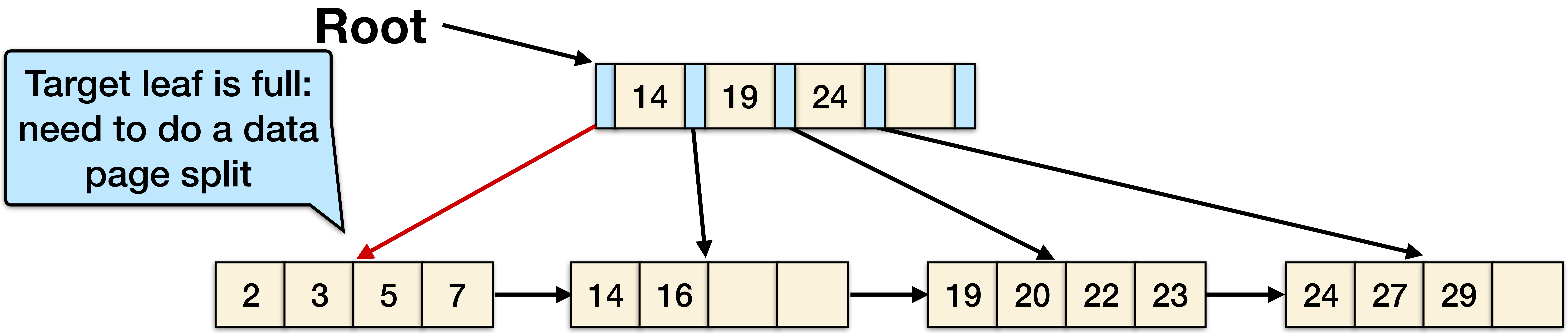
Searches in a B+-tree begin at the root, and **key comparisons** direct it to a leaf.

Example: search for 15

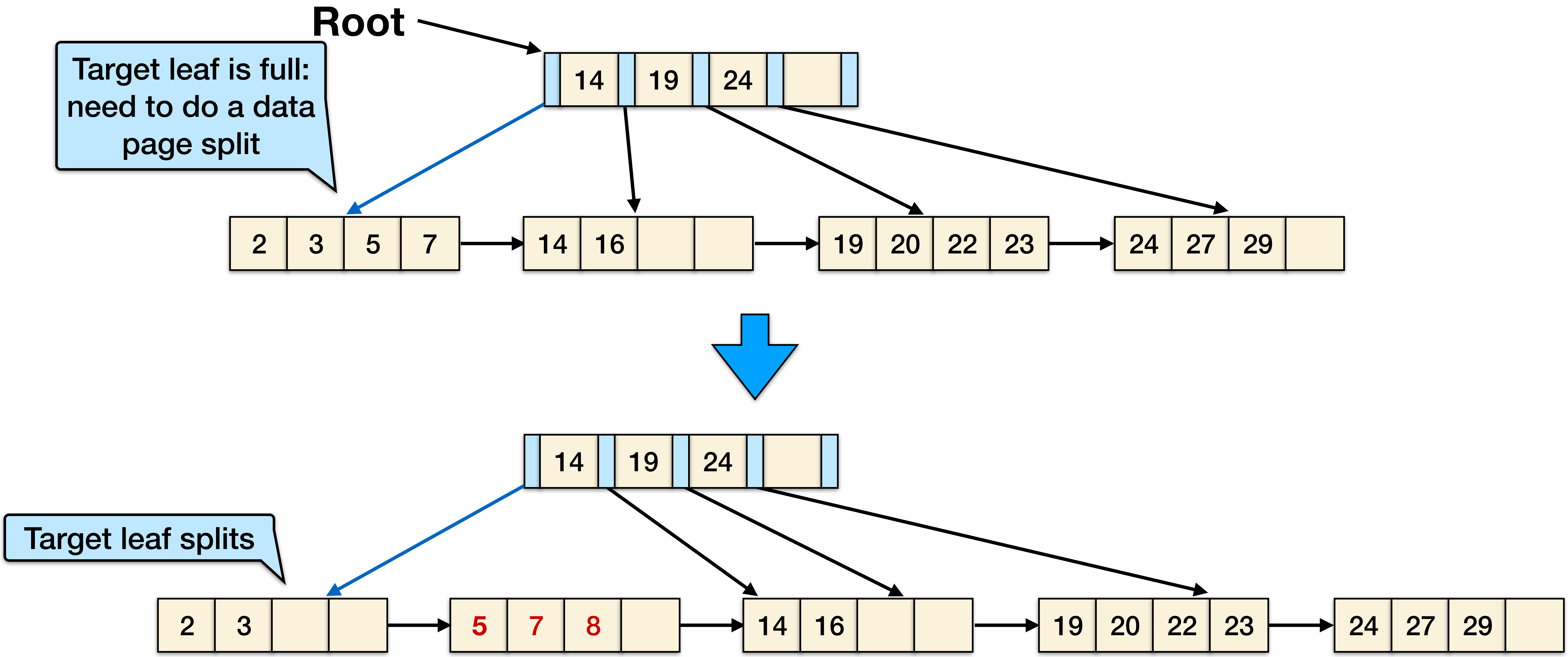


Based on the search, we know 15 is not in the tree!

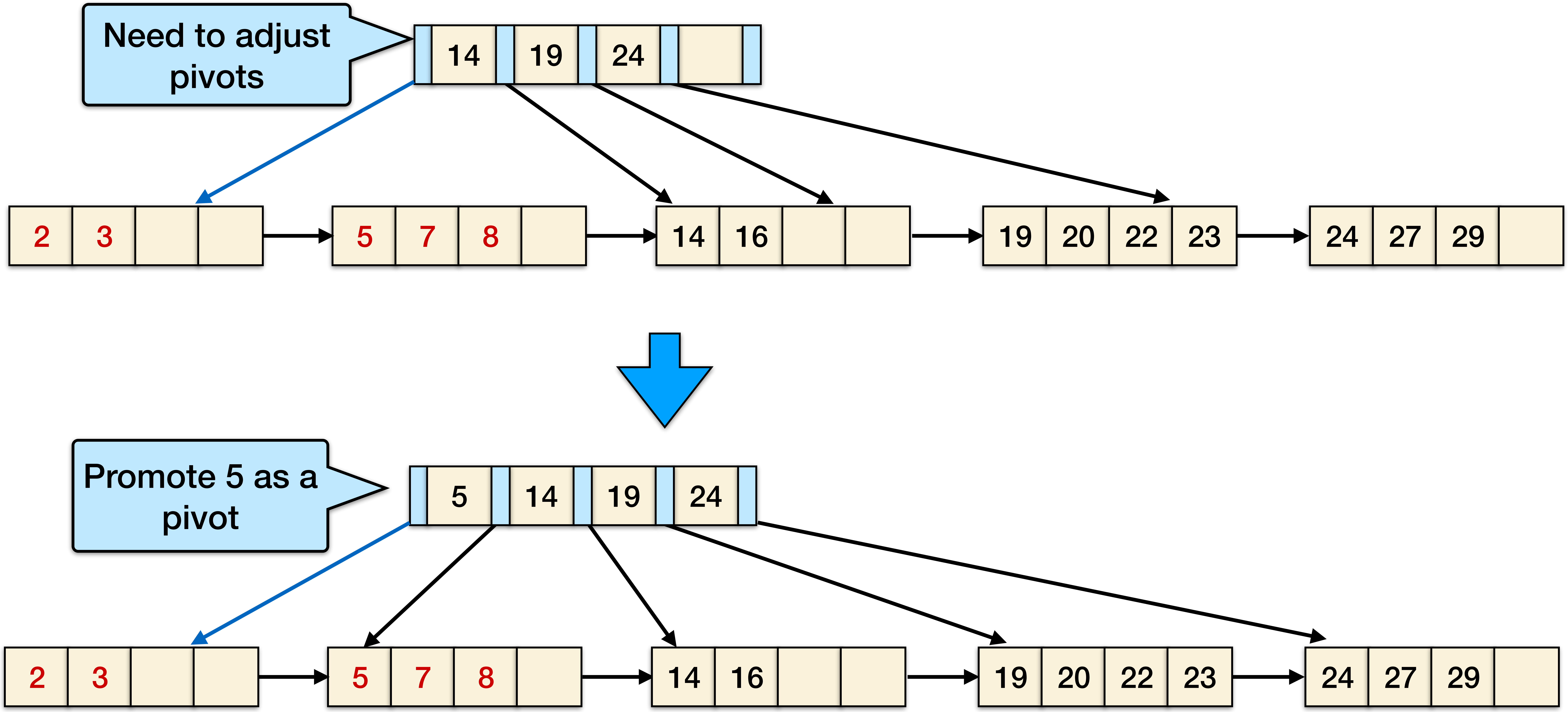
Example: Insert 8 into B+-tree



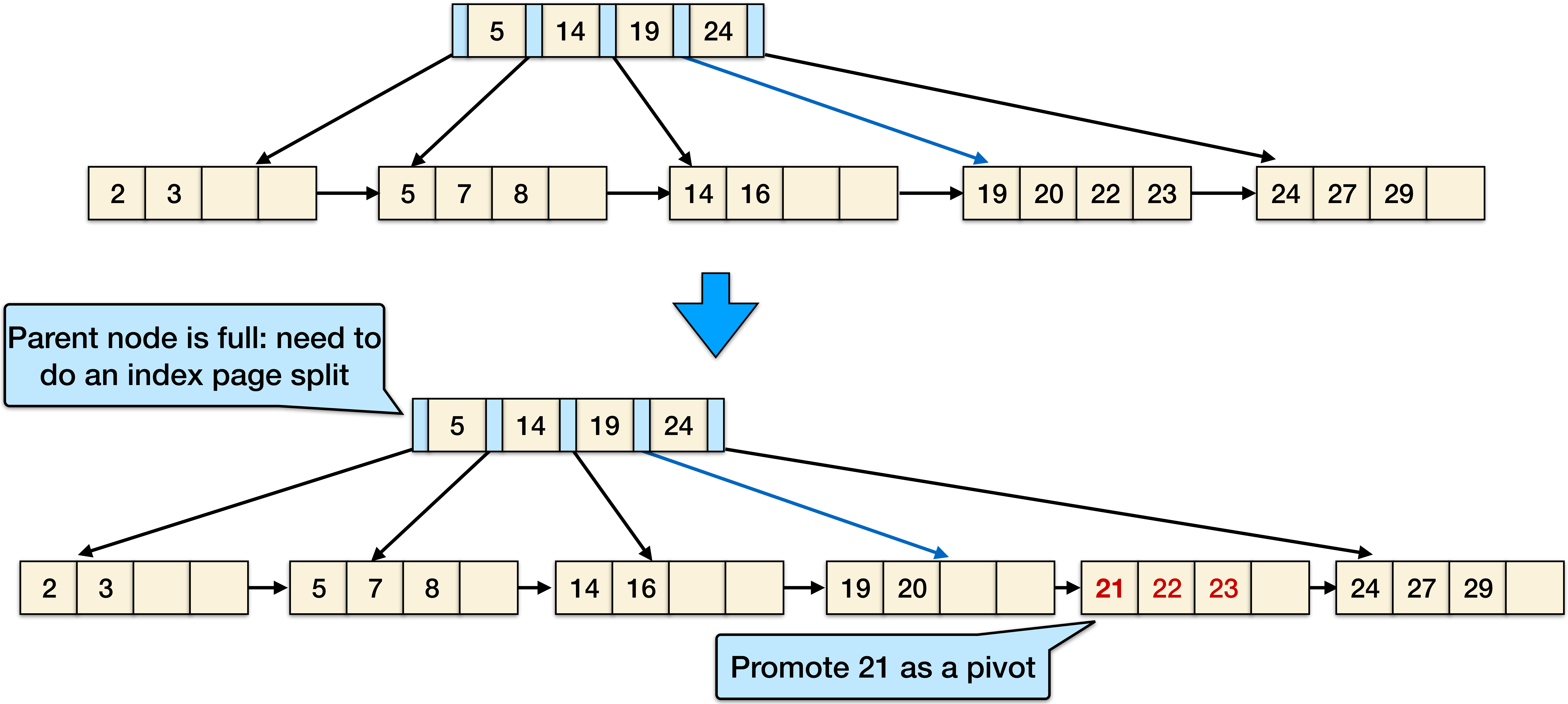
Example: Insert 8 into B+-tree



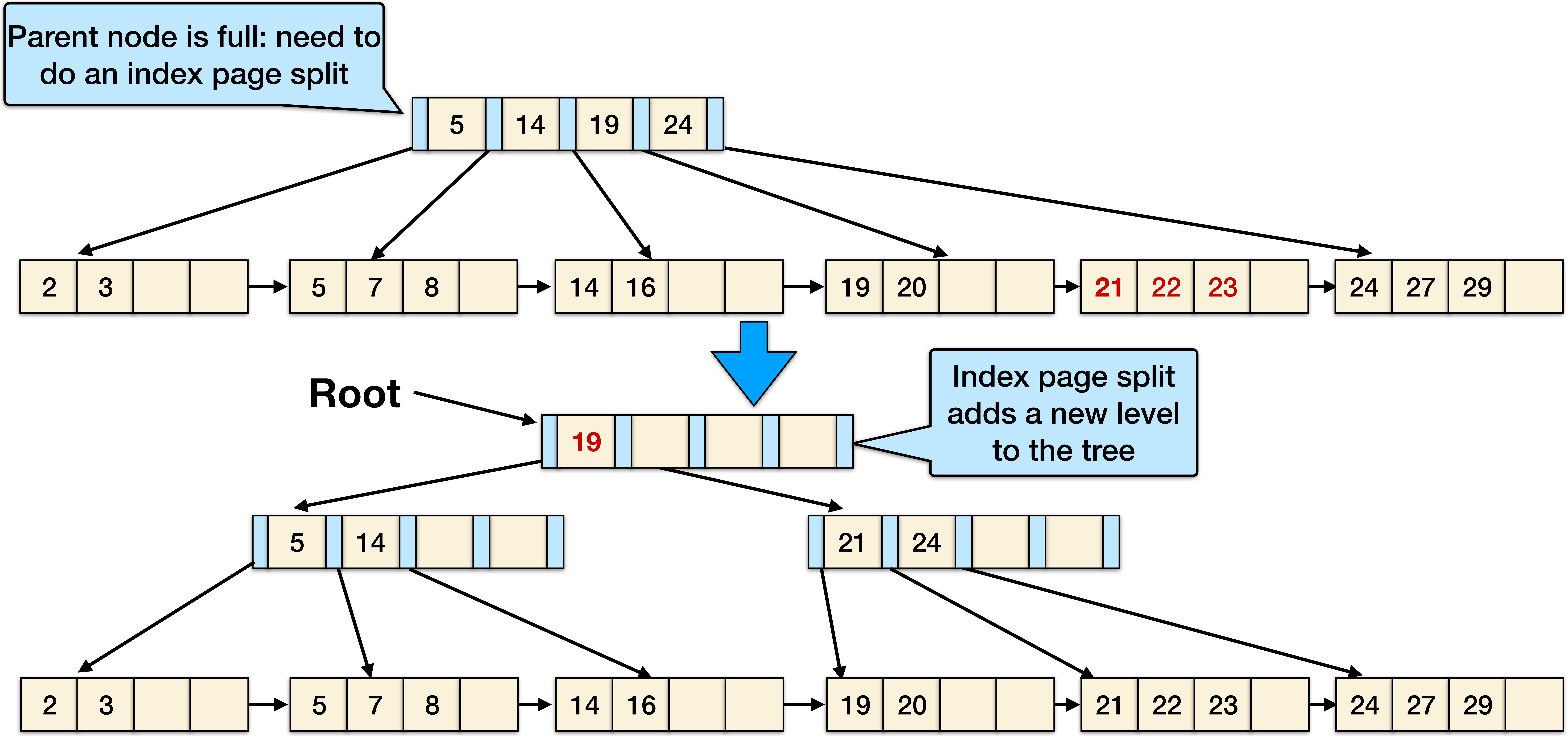
Example: Insert 8 into B+-tree



Next example: Insert 21 into B+-tree



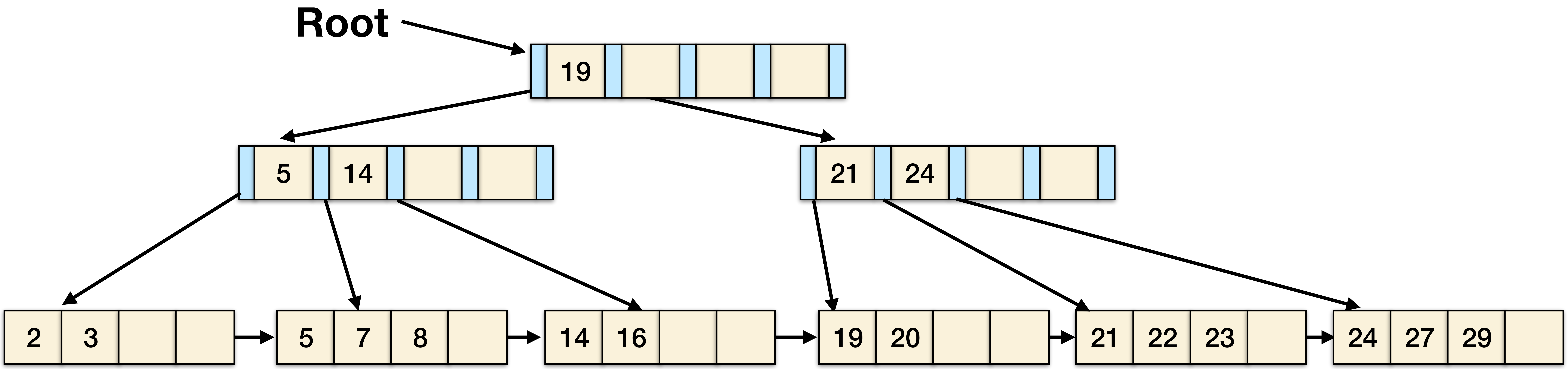
Next example: Insert 21 into B+-tree



B-tree bounds

B-trees support searches (reads) and inserts (updates) in $O(\log_B N)$ cache-line transfers.

Most updates only write to the leaves, but in the worst case, an update may propagate up the tree for $O(\log_B N)$ writes.

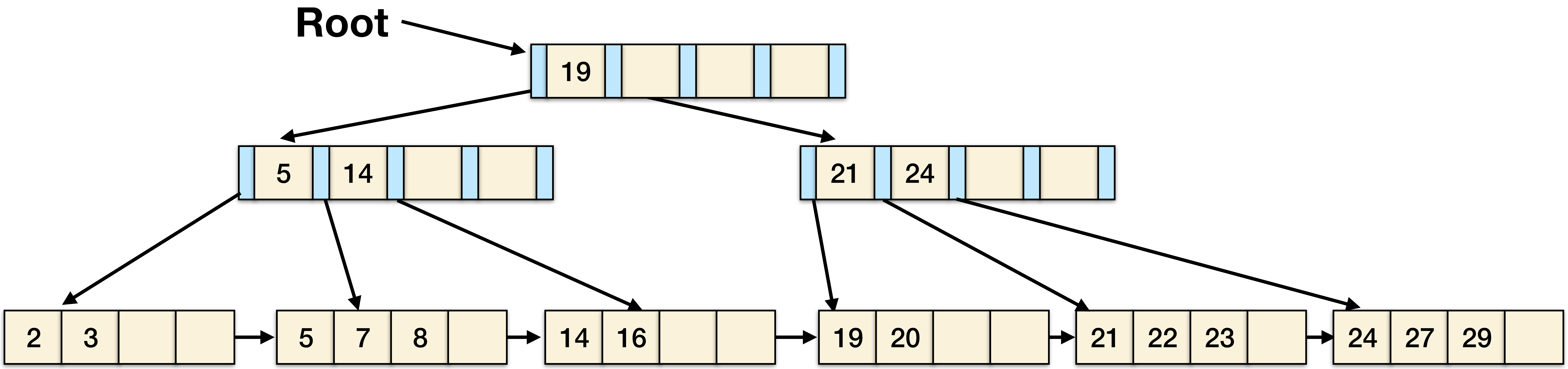


B-tree bounds

Asymptotically better than binary trees

B-trees support searches (reads) and inserts (updates) in $O(\log_B N)$ cache-line transfers.

Most updates only write to the leaves, but in the worst case, an update may propagate up the tree for $O(\log_B N)$ writes.

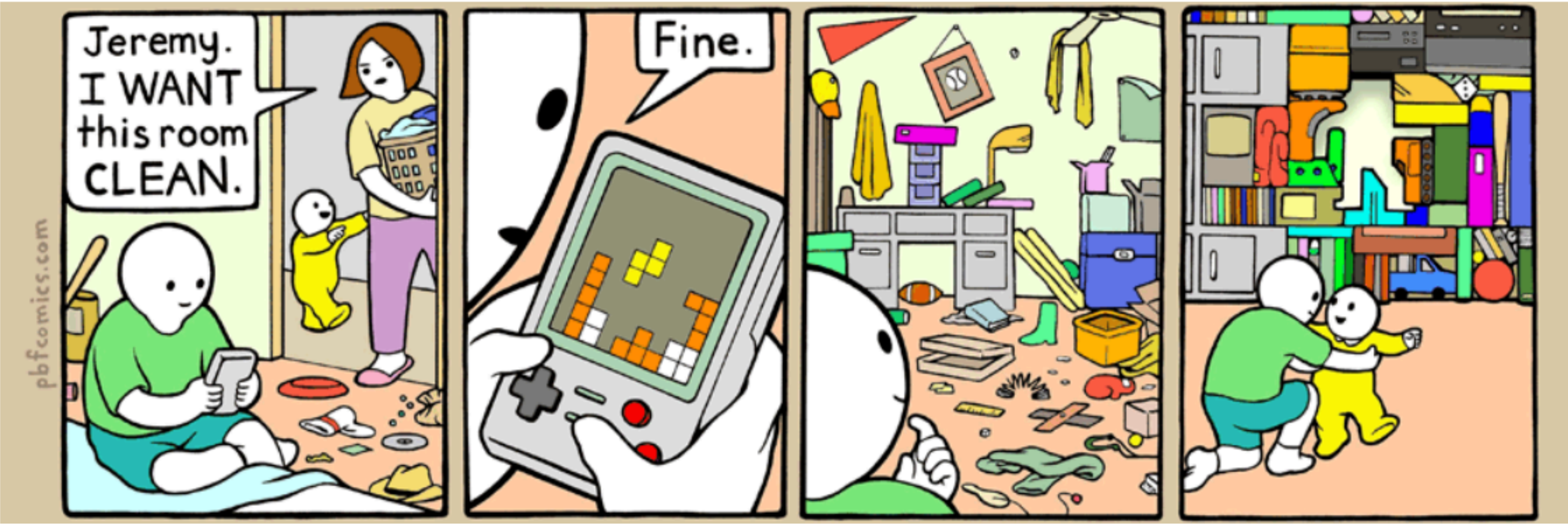


Question

Are B-trees **optimal** in the ideal-cache model?

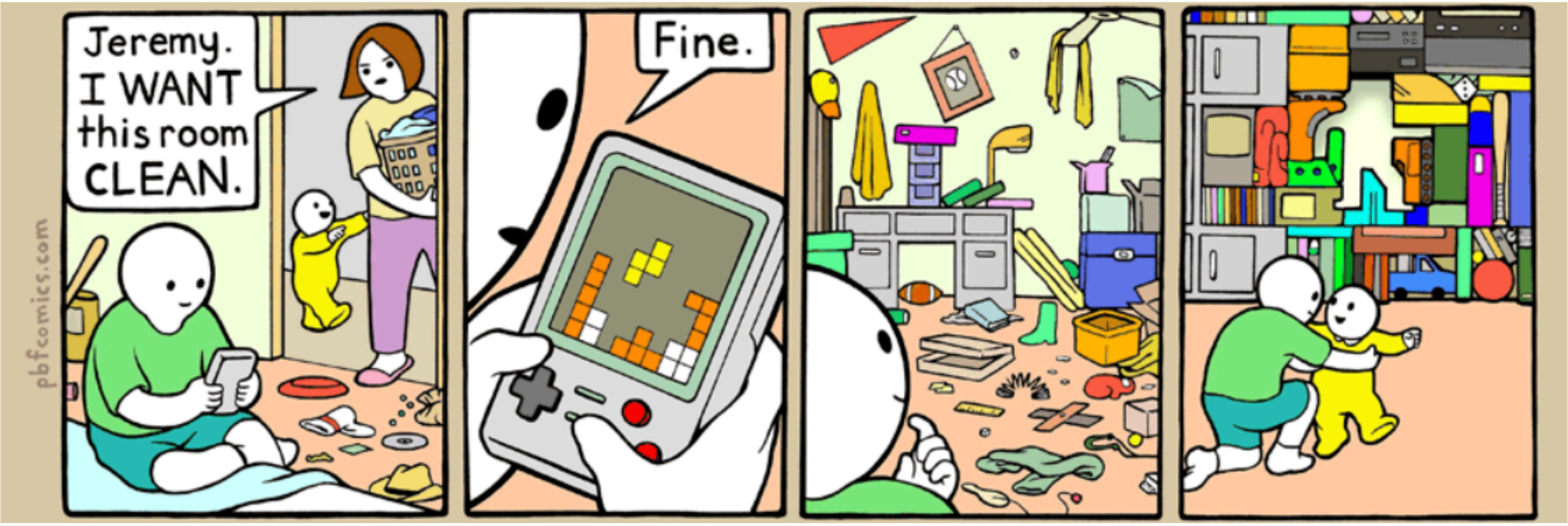
How can/should we organize data?

There are many different approaches…



https://pbfcomics.com/comics/game-boy/

How should we organize data?



Logging
e.g., with an array

Indexing
e.g., B-trees

Slide from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture11.pdf

How should we organize data?

	Insert	Search
Logging	$O(1/B)$	$O(N/B)$
Indexing (assuming B-tree)	$O(\log_B N)$	$O(\log_B N)$

Append to end of log

Scan entire log

↓ 😡

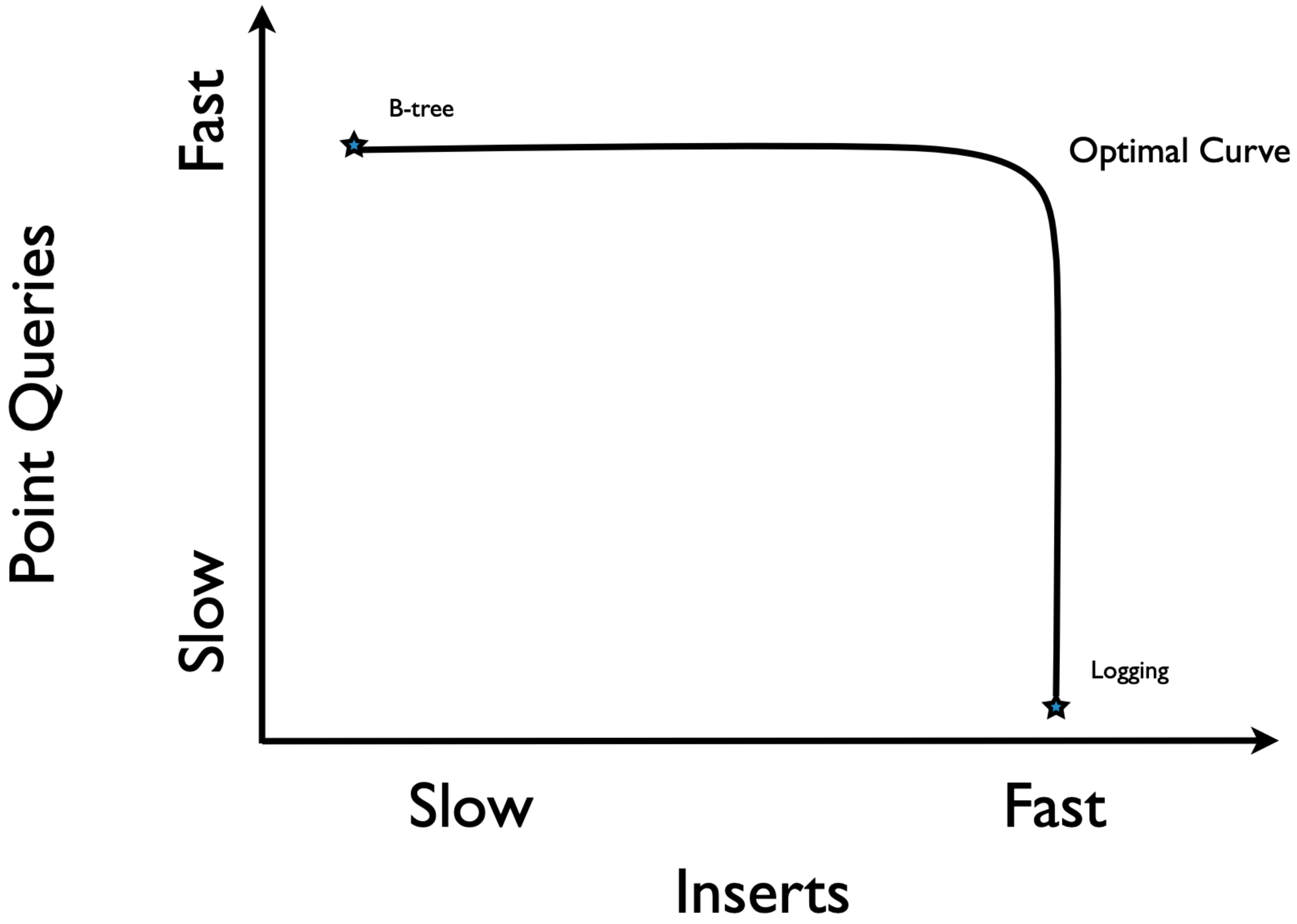
💚 ↓

Insert into leaf (root-to-leaf traversal)

Locate in leaf (root-to-leaf traversal)

Slide from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture11.pdf

Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]



Slides from: https://www3.cs.stonybrook.edu/~bender/talks/2012-Bender-Dagstuhl-write-optimized-talk.pdf

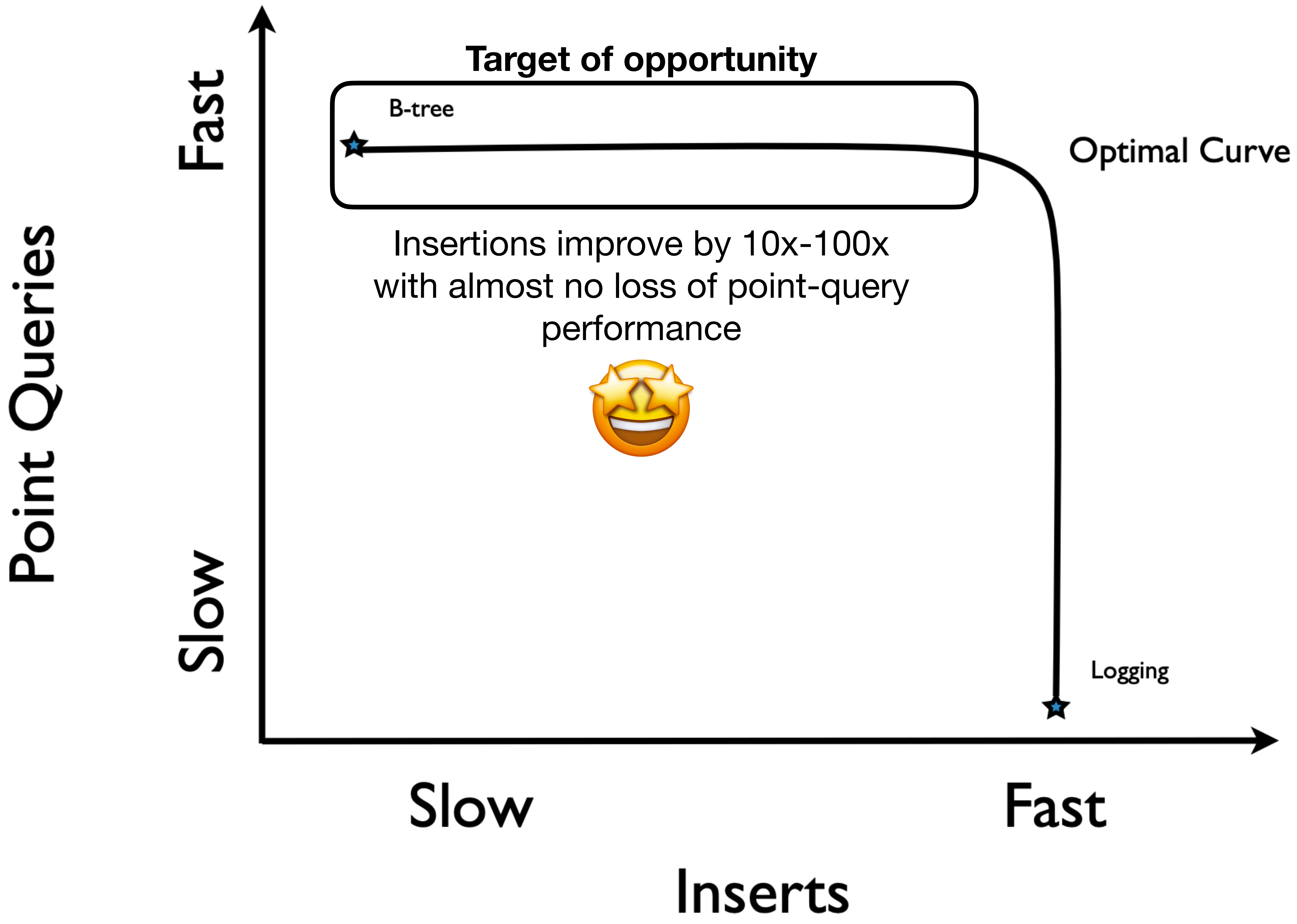
Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]

	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^\varepsilon} N\right)$
B-tree (ε=1)	$O\left(\log_B N\right)$	$O\left(\log_B N\right)$
ε=1/2	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left(\log_B N\right)$
ε=0	$O\left(\frac{\log N}{B}\right)$	$O\left(\log N\right)$

10x-100x faster inserts

Slides from: https://www3.cs.stonybrook.edu/~bender/talks/2012-Bender-Dagstuhl-write-optimized-talk.pdf

Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]

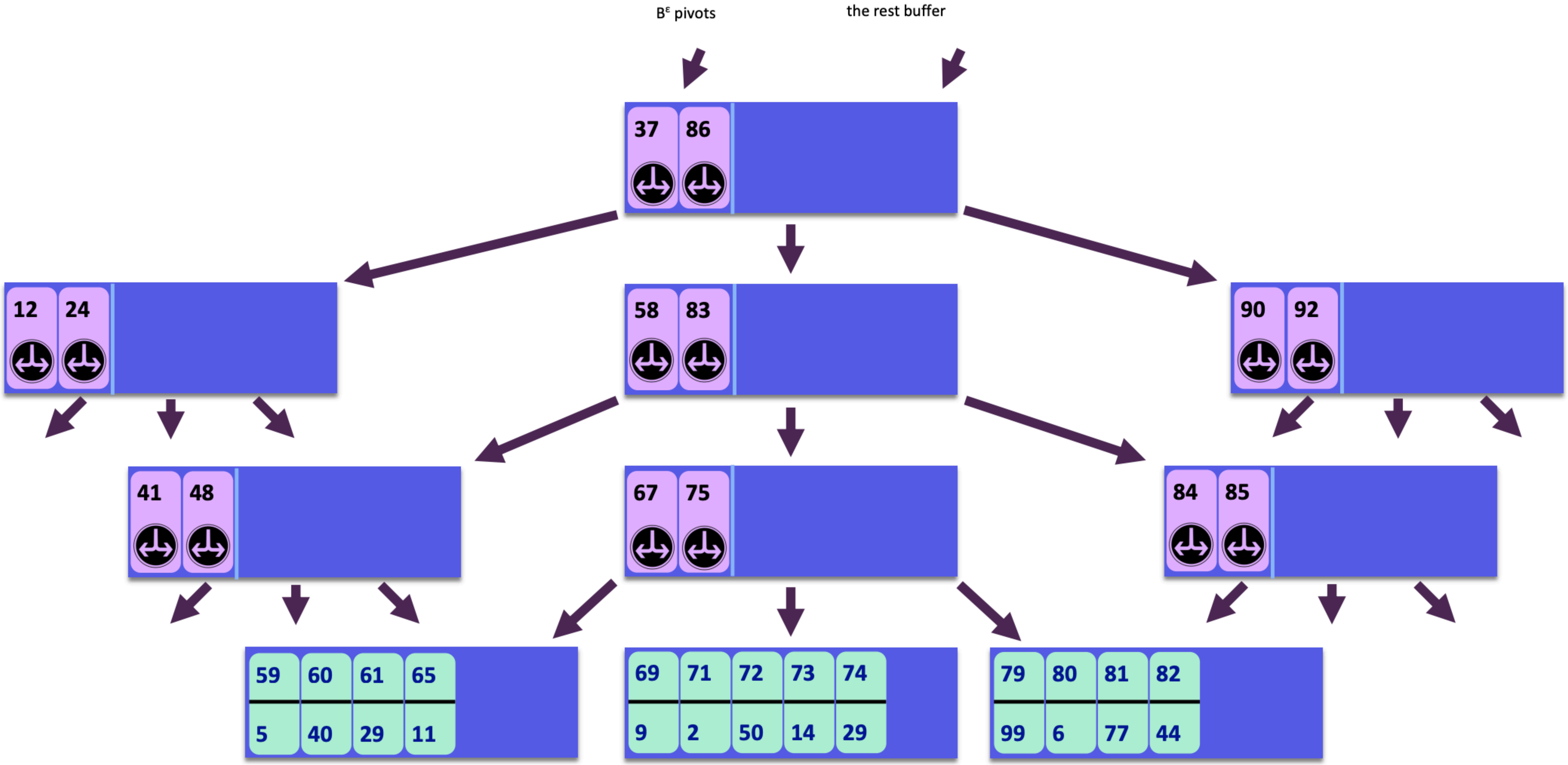


Slides from: https://www3.cs.stonybrook.edu/~bender/talks/2012-Bender-Dagstuhl-write-optimized-talk.pdf

B$^{\mathcal{E}}$ trees
(and write optimization)

$B^{\mathcal{E}}$ trees

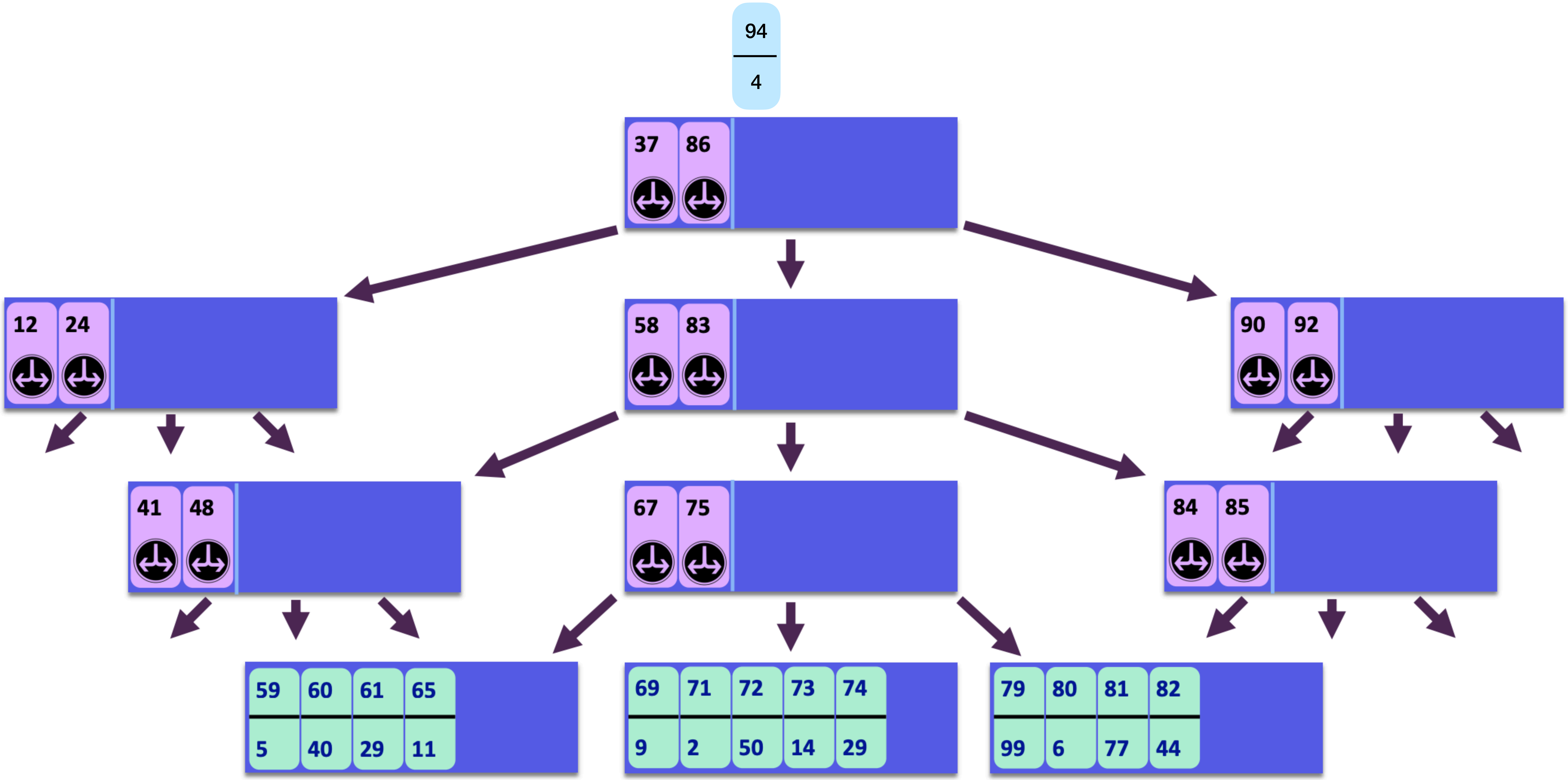
$B^{\mathcal{E}}$ trees are search trees (like B-trees)



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\varepsilon}$ trees

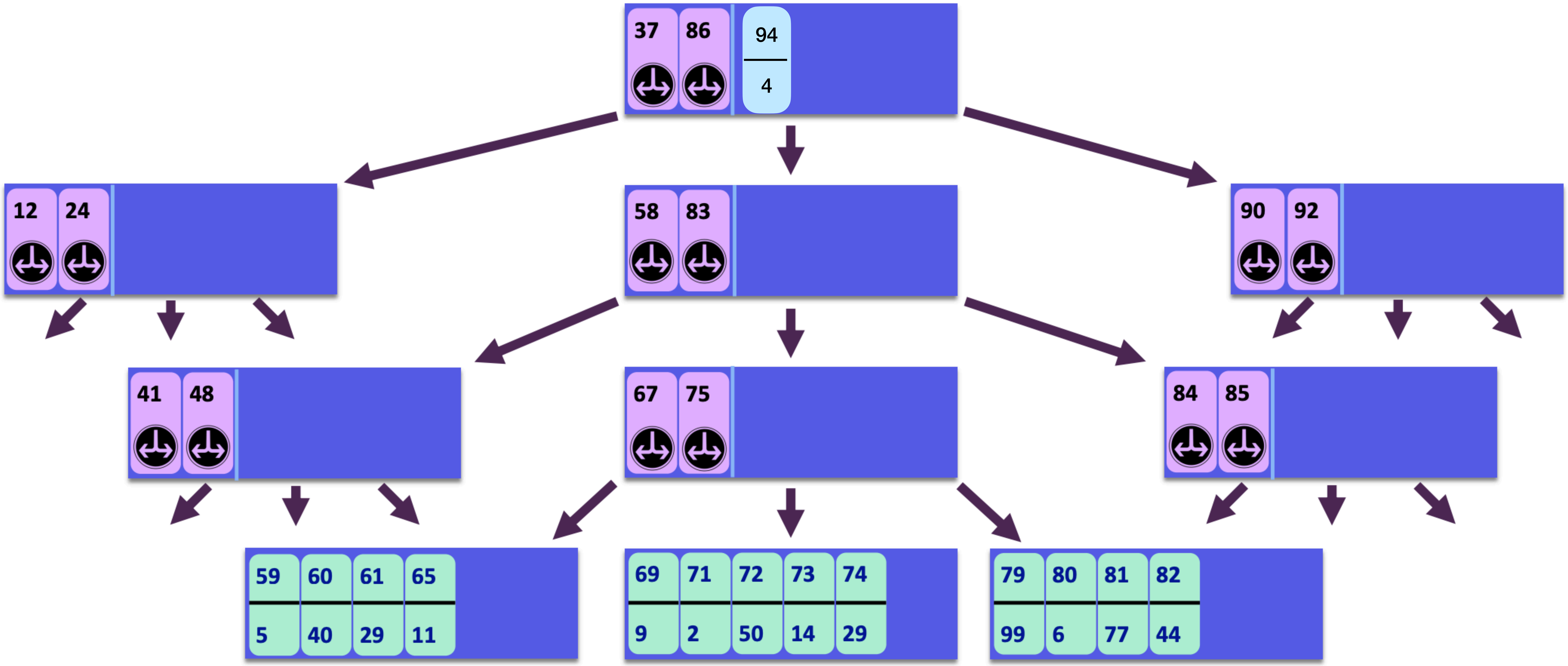
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\varepsilon}$ trees

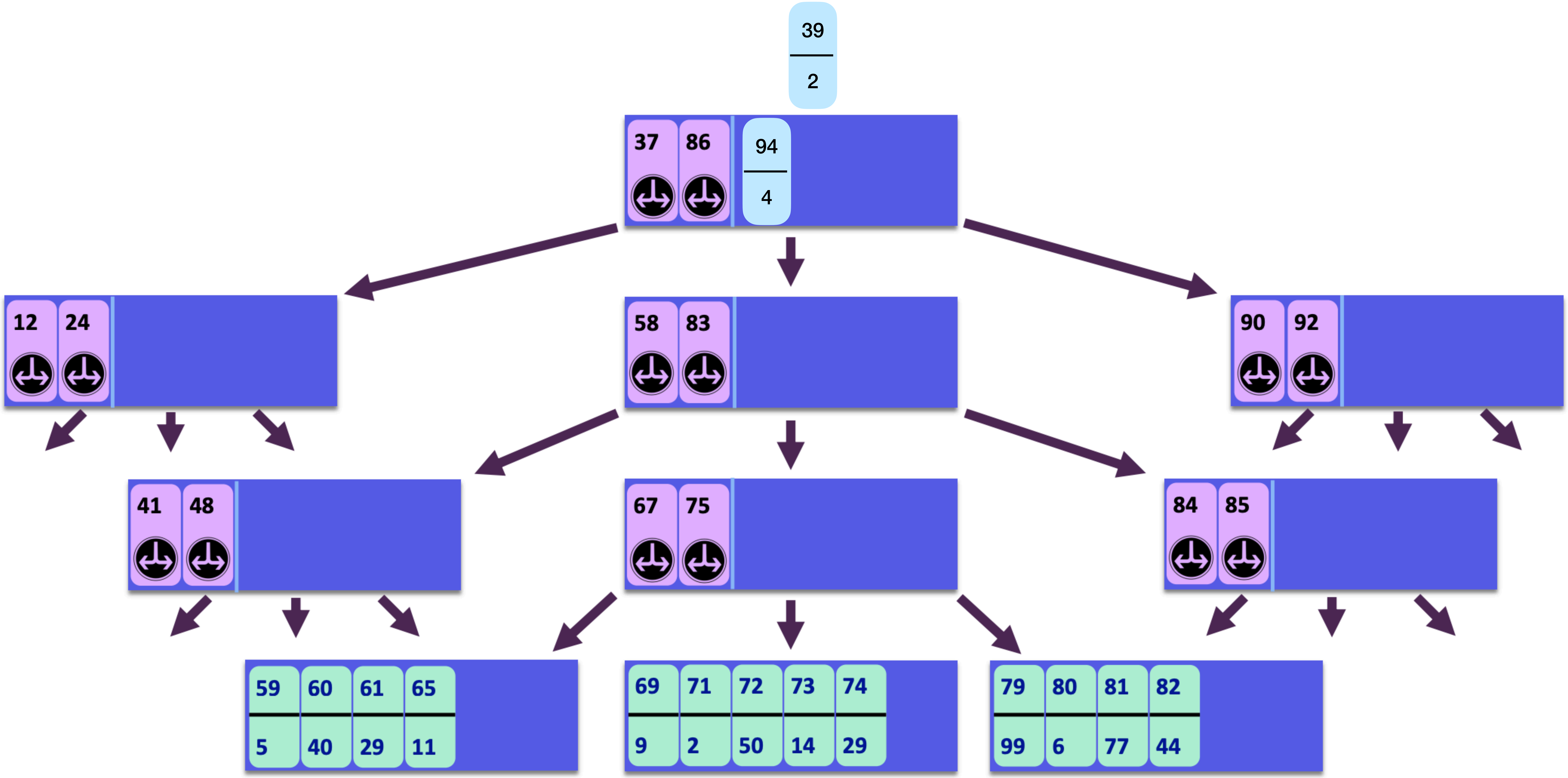
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\varepsilon}$ trees

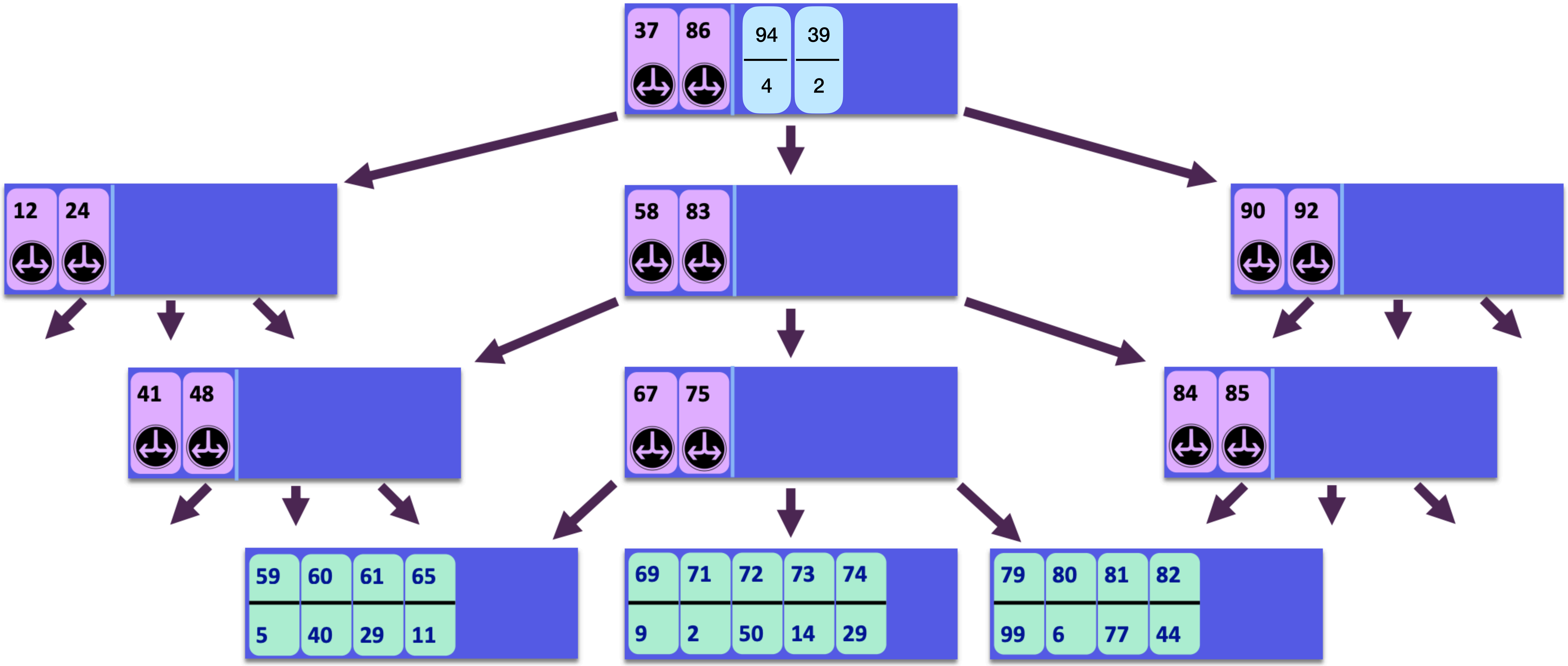
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\varepsilon}$ trees

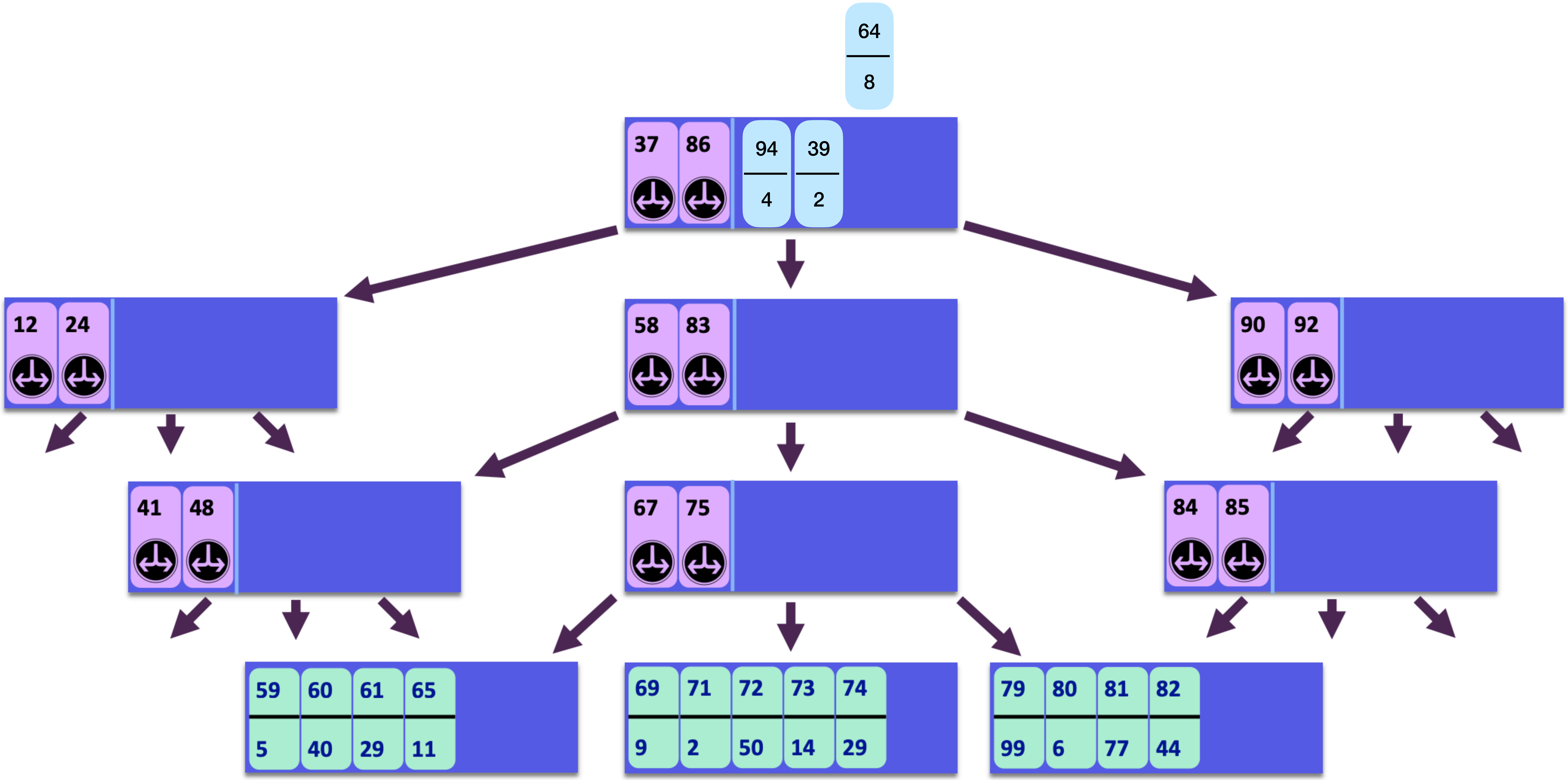
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees

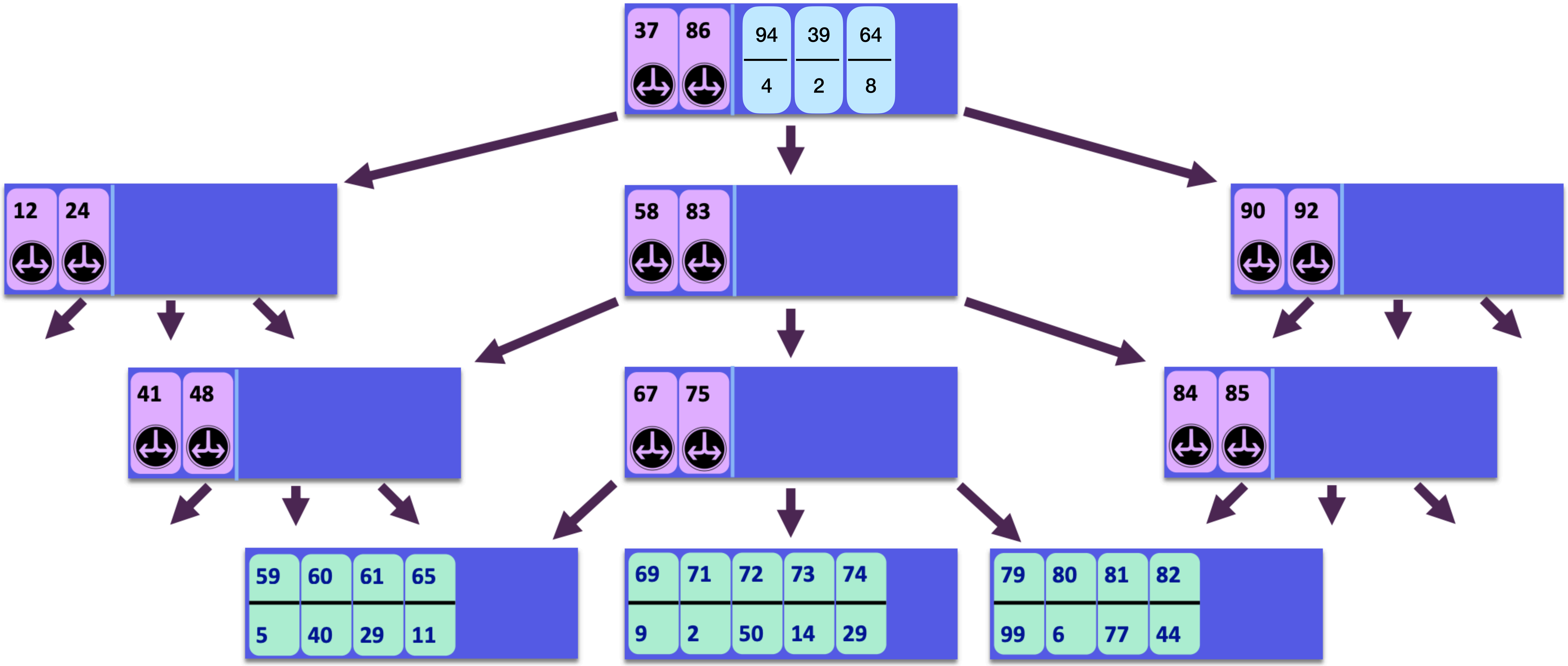
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

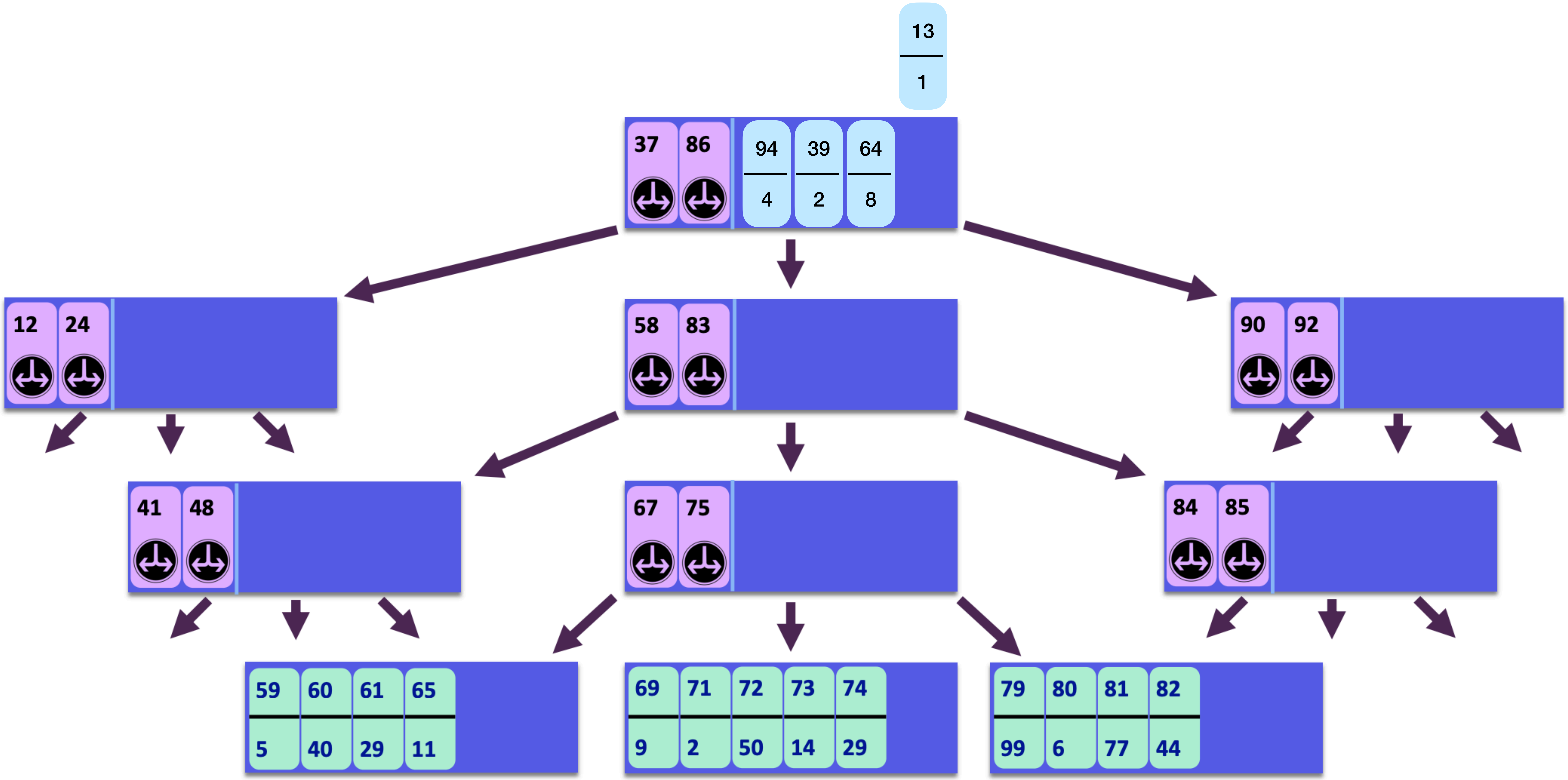
Insertion in B$^{\varepsilon}$ trees

Insertions get put into the root buffer



Insertion in B^{ε} trees

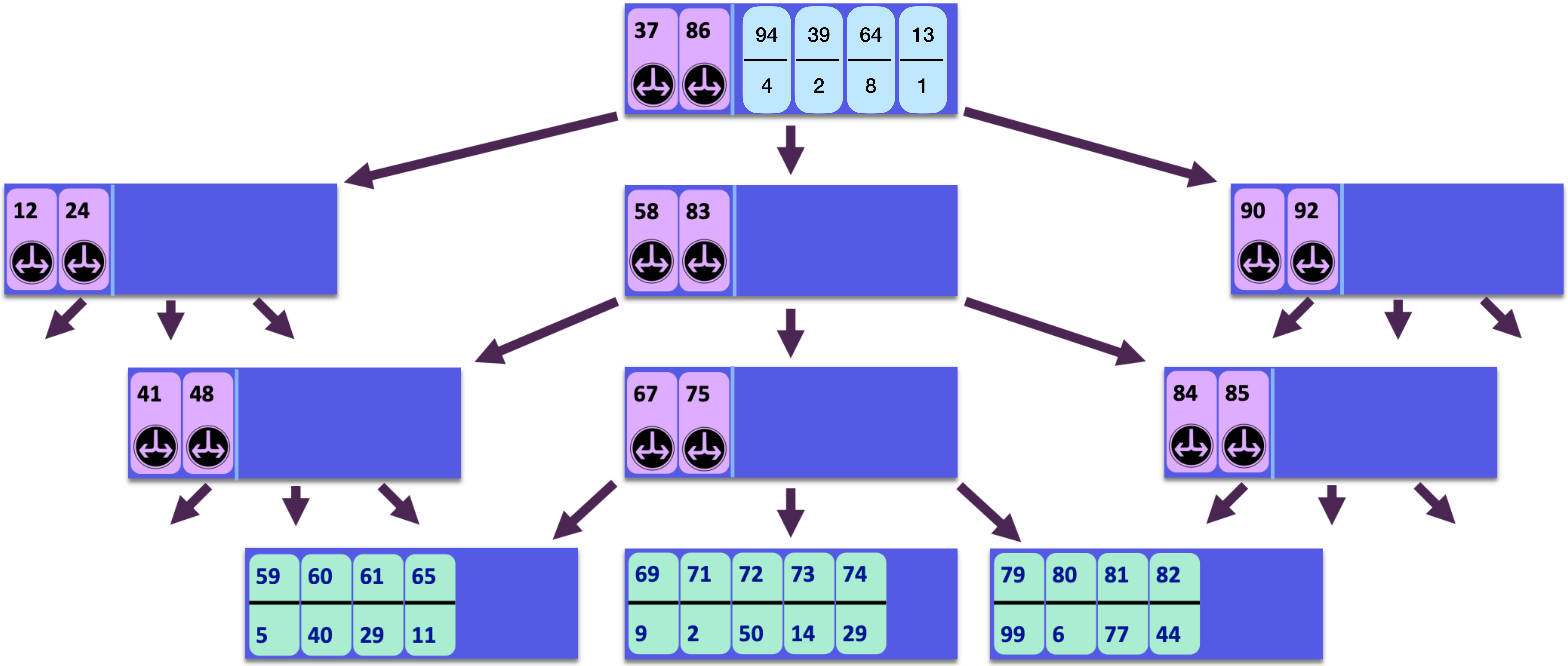
Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

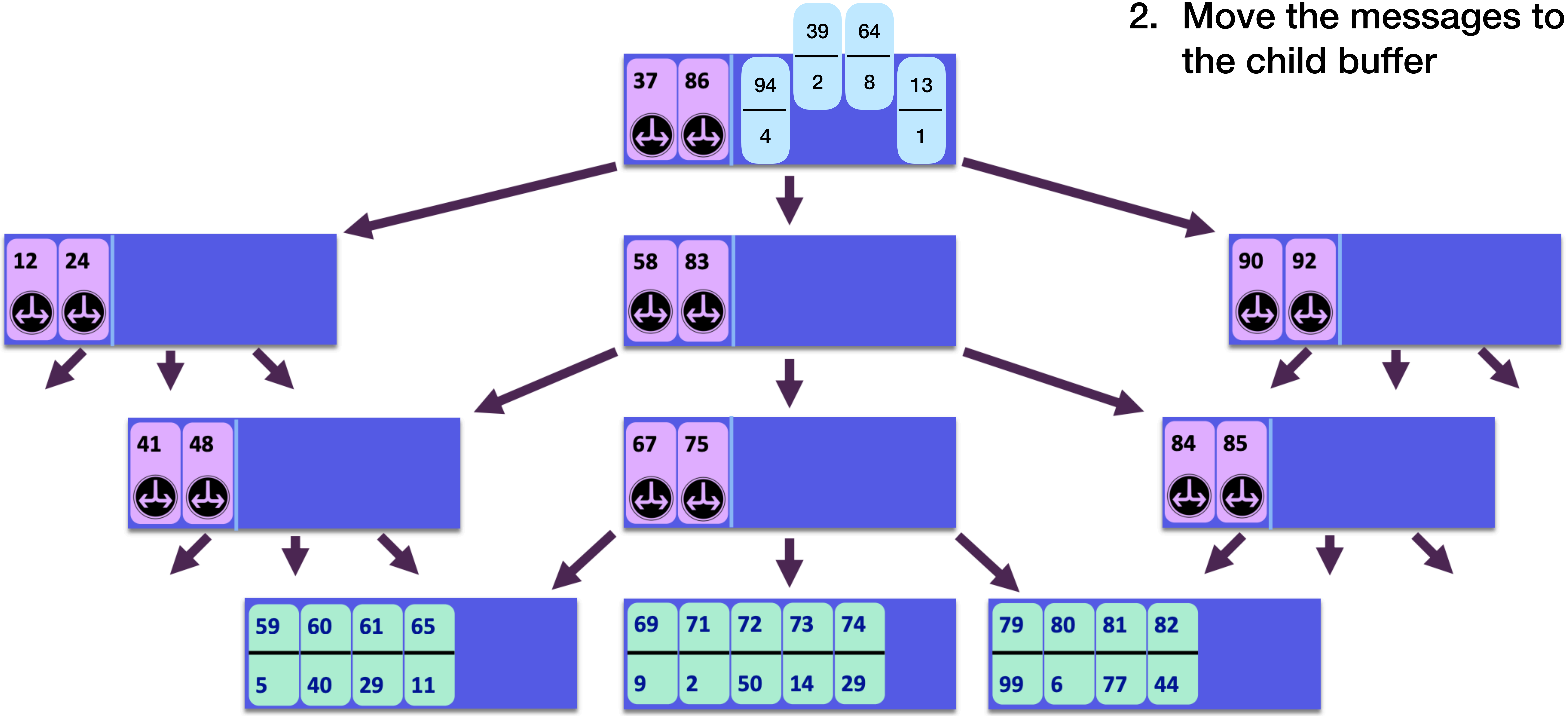
Insertion in B^{ε} trees

Insertions get put into the root buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\mathcal{E}}$ trees



When a buffer is full:

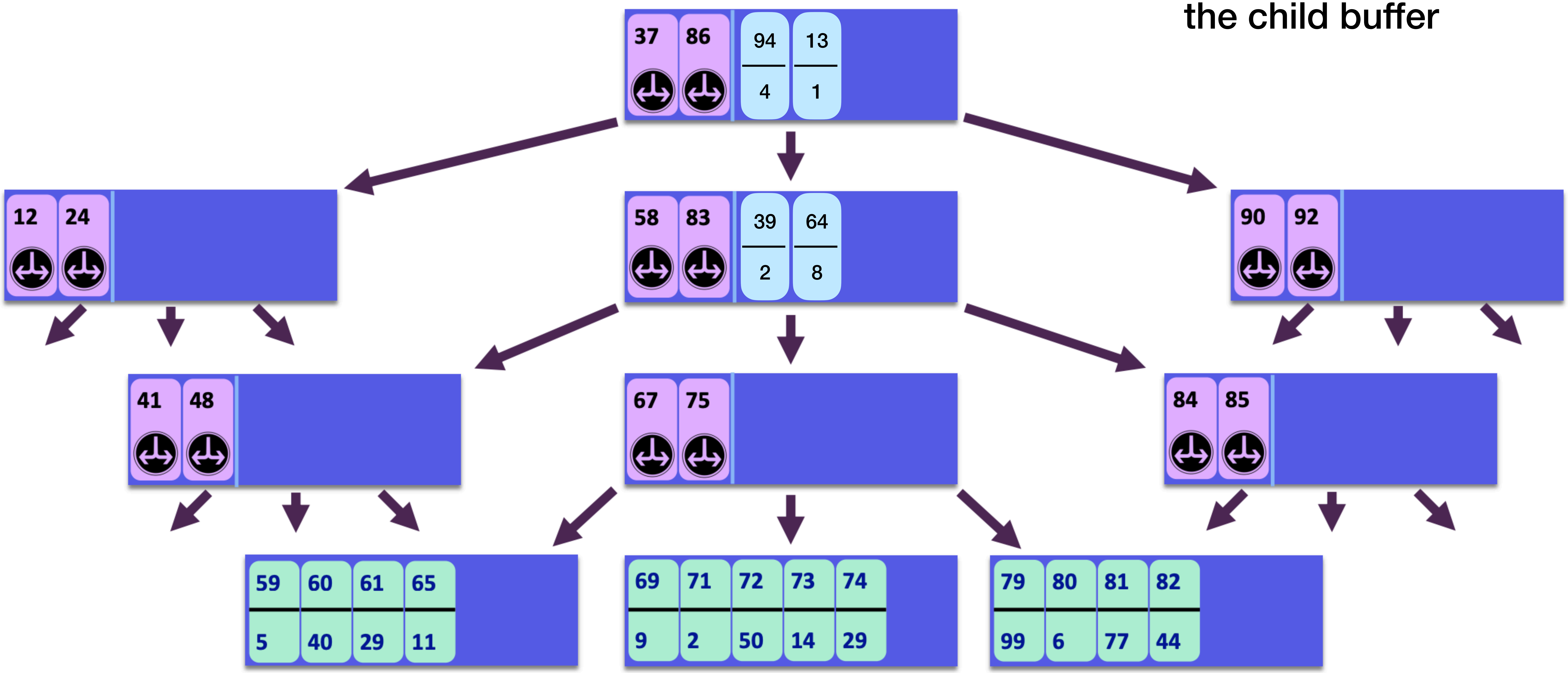
1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

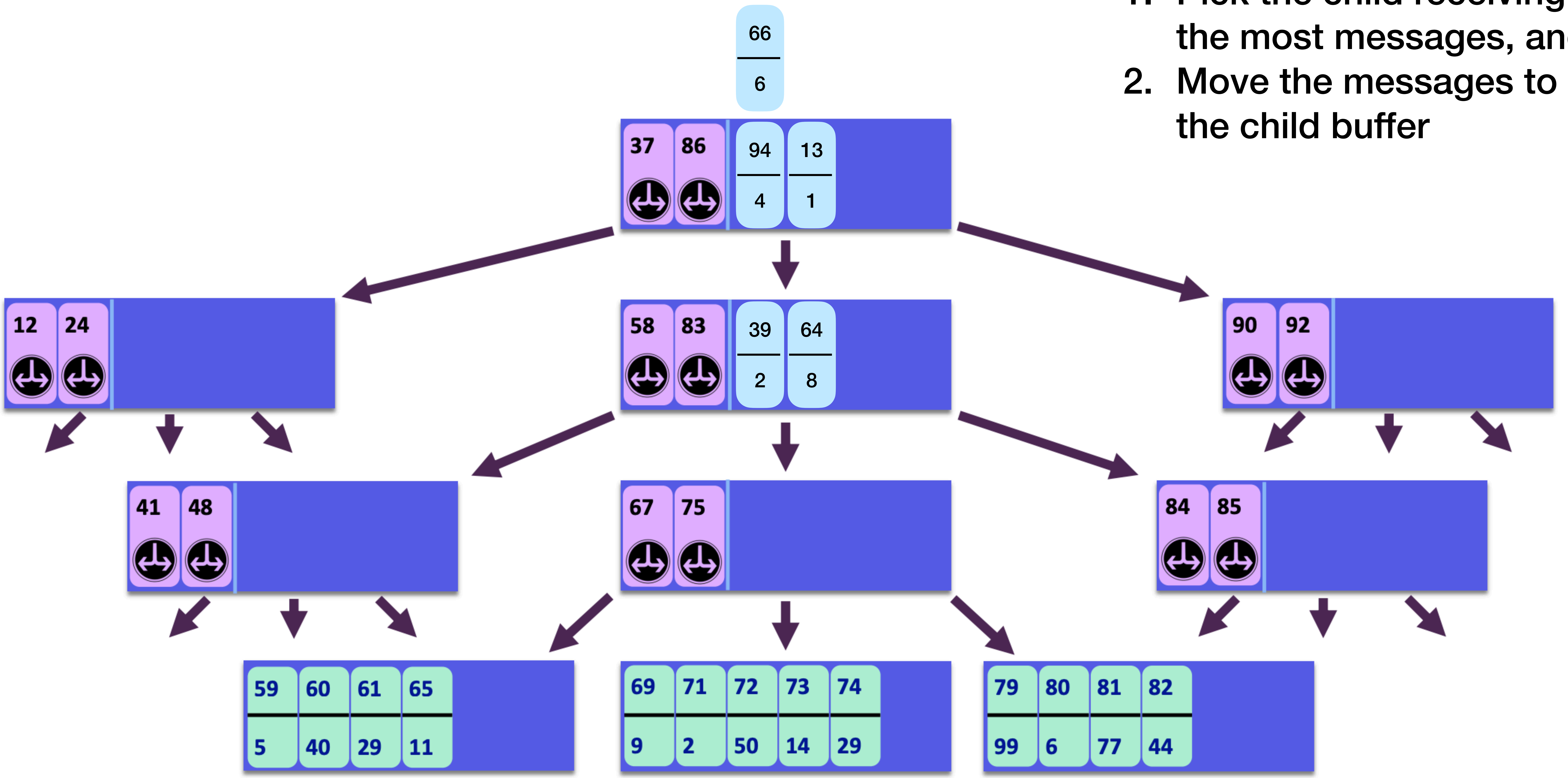
Insertion in B^{ε} trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer



Insertion in B$^{\varepsilon}$ trees



When a buffer is full:

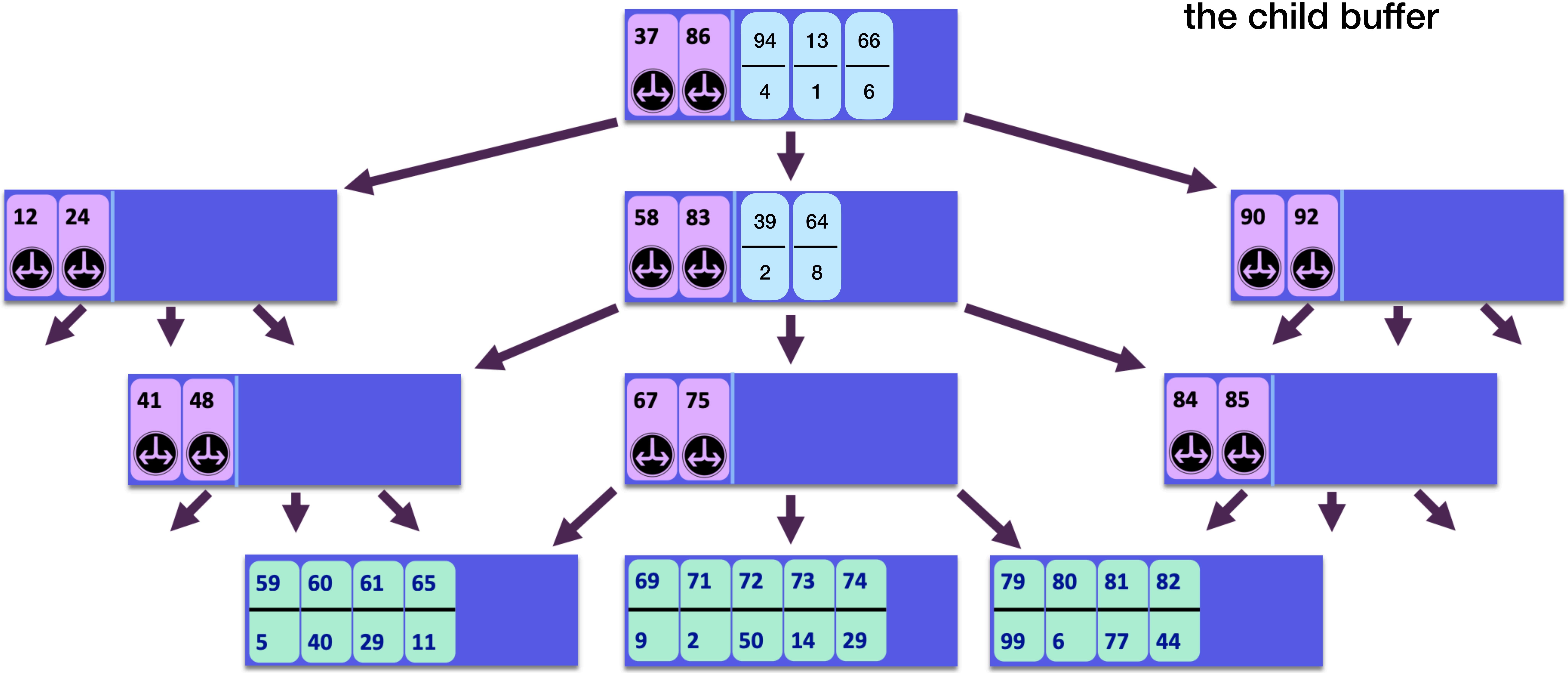
1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\mathcal{E}}$ trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

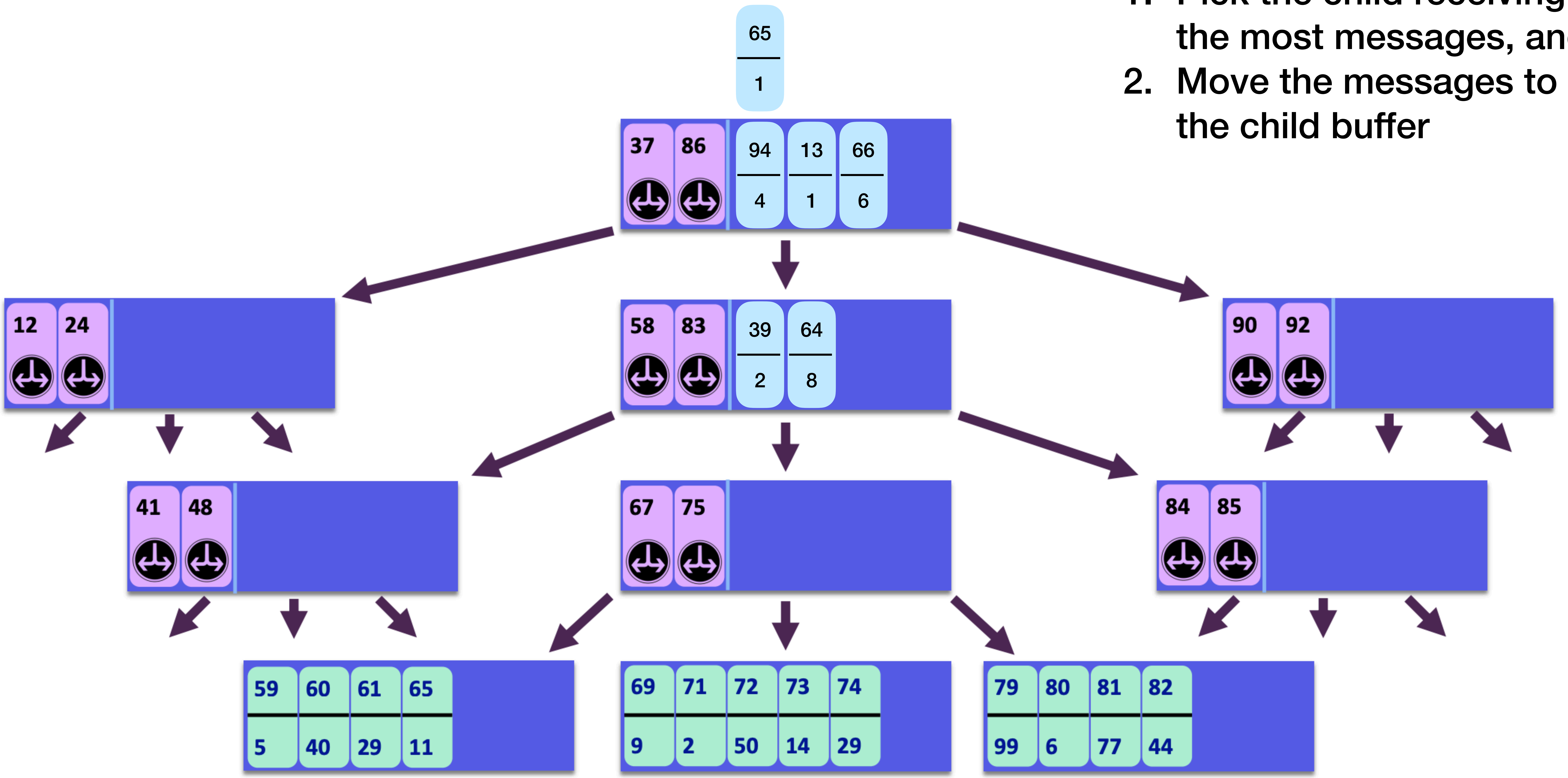


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B$^{\varepsilon}$ trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

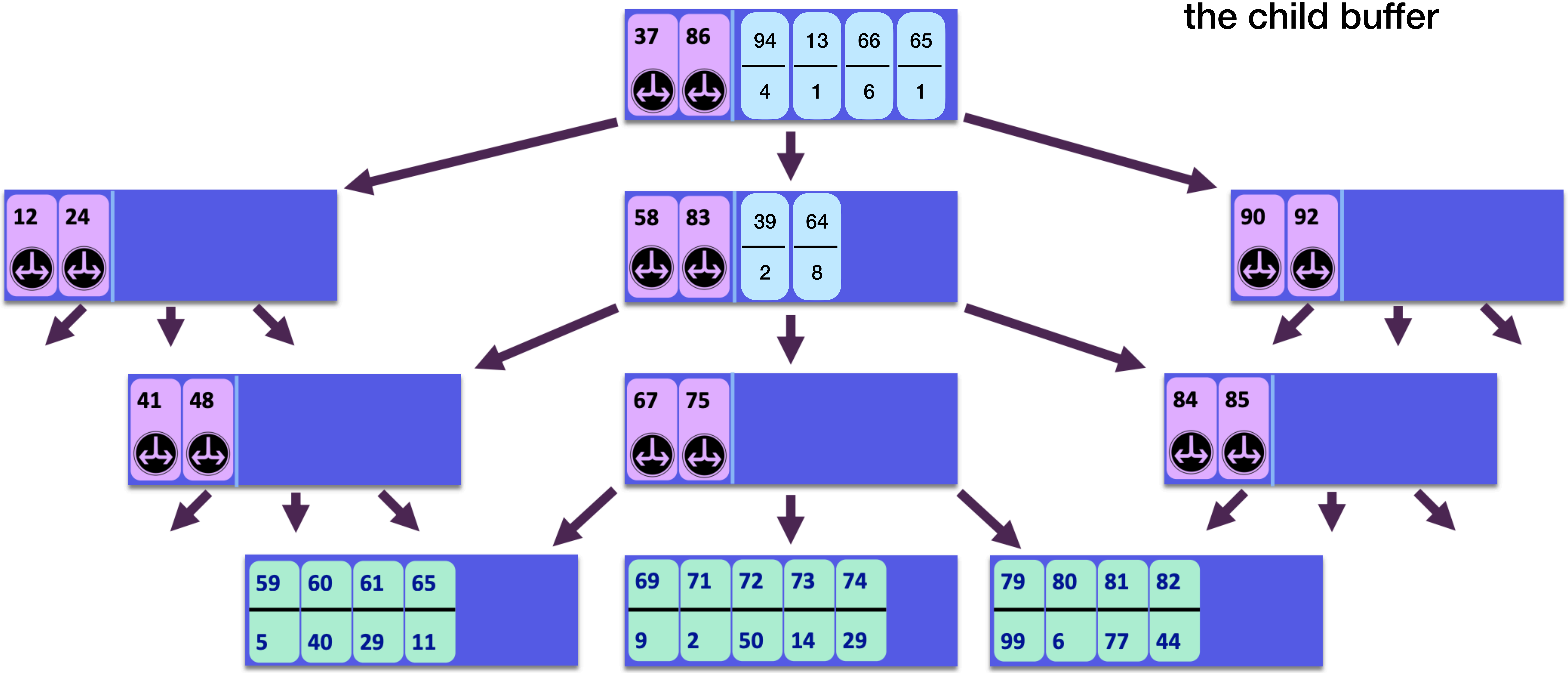


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees

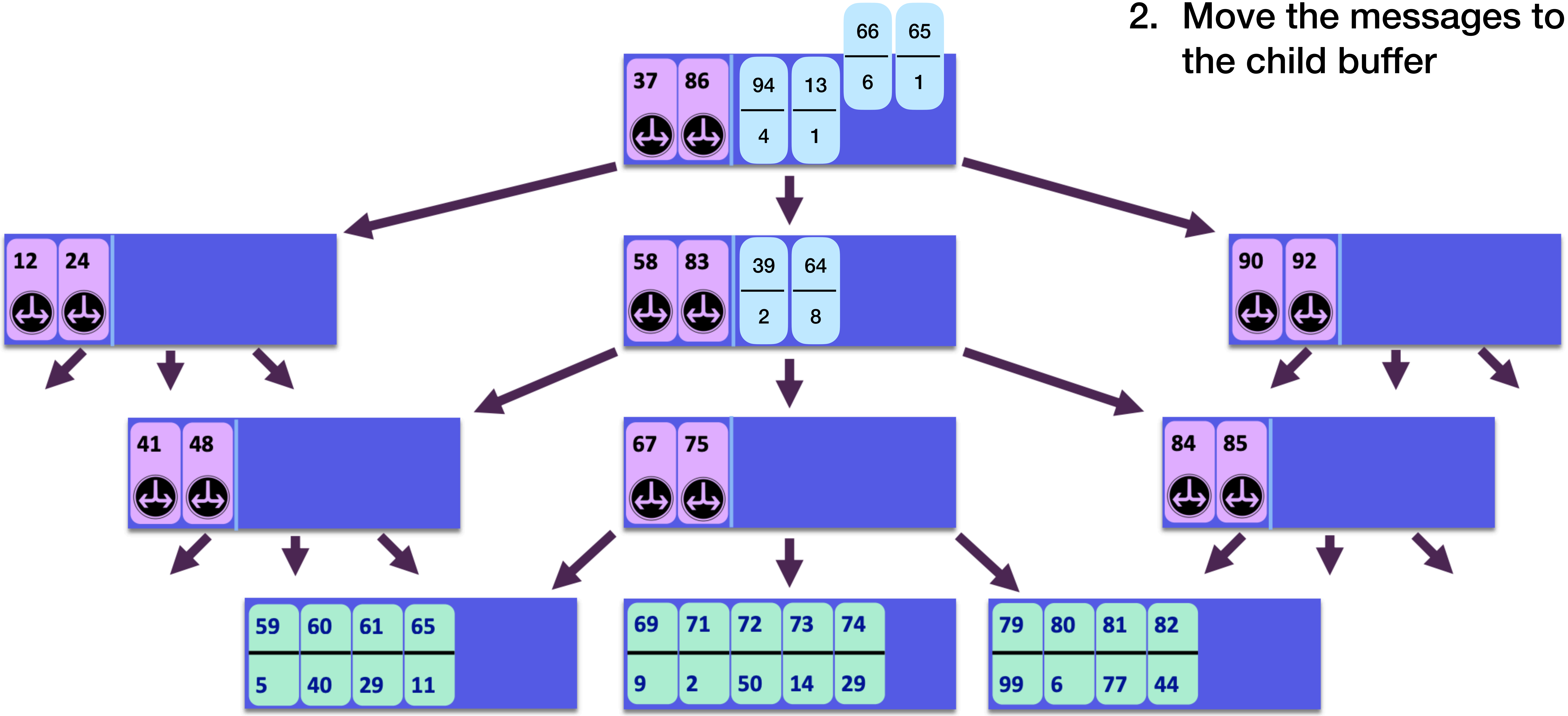
When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees



When a buffer is full:

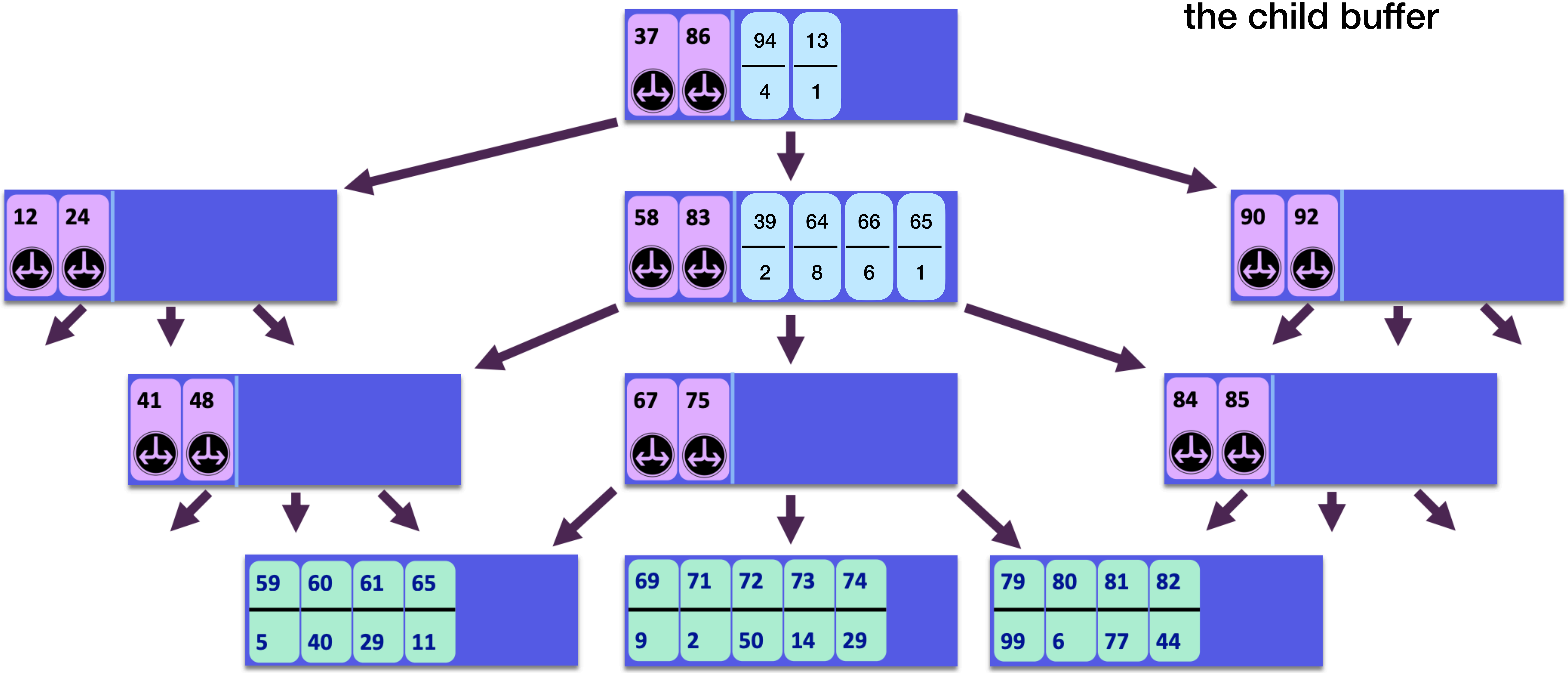
1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

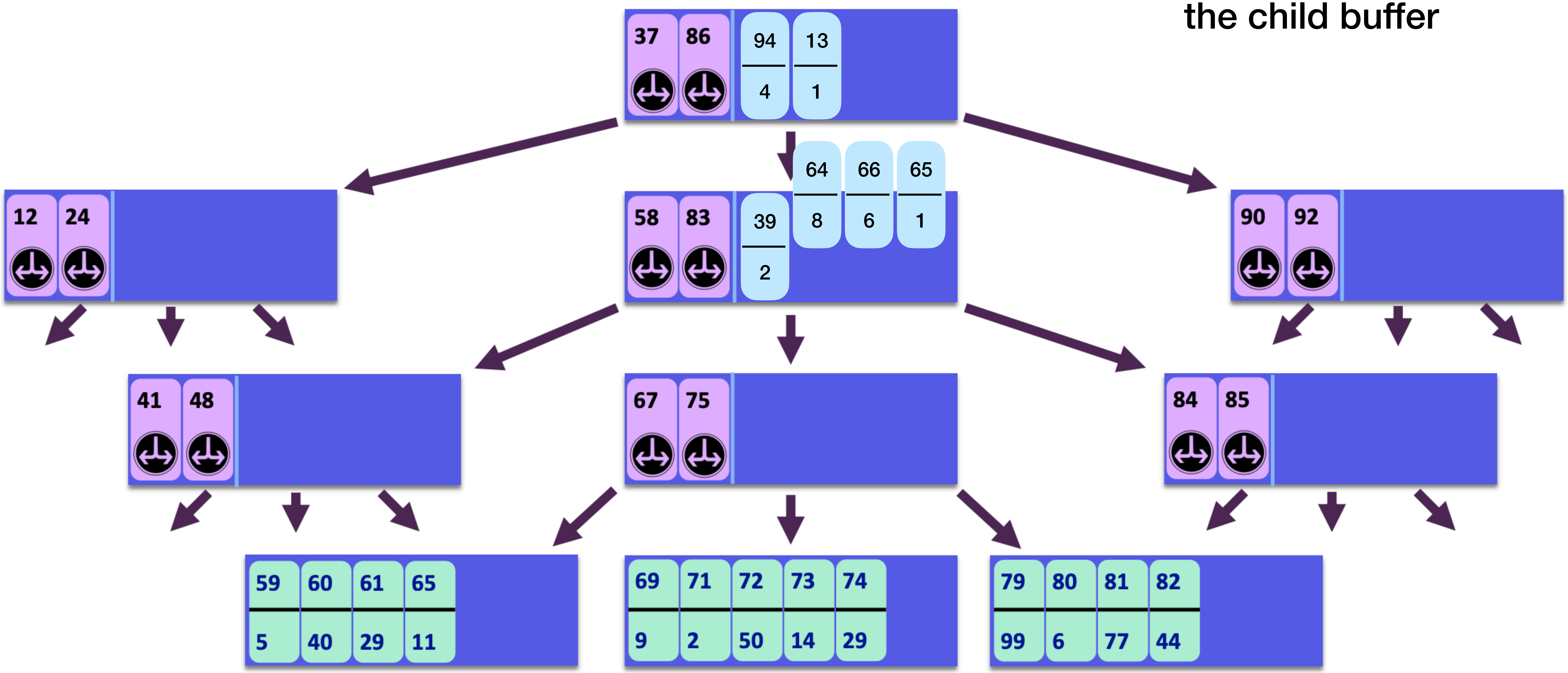


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

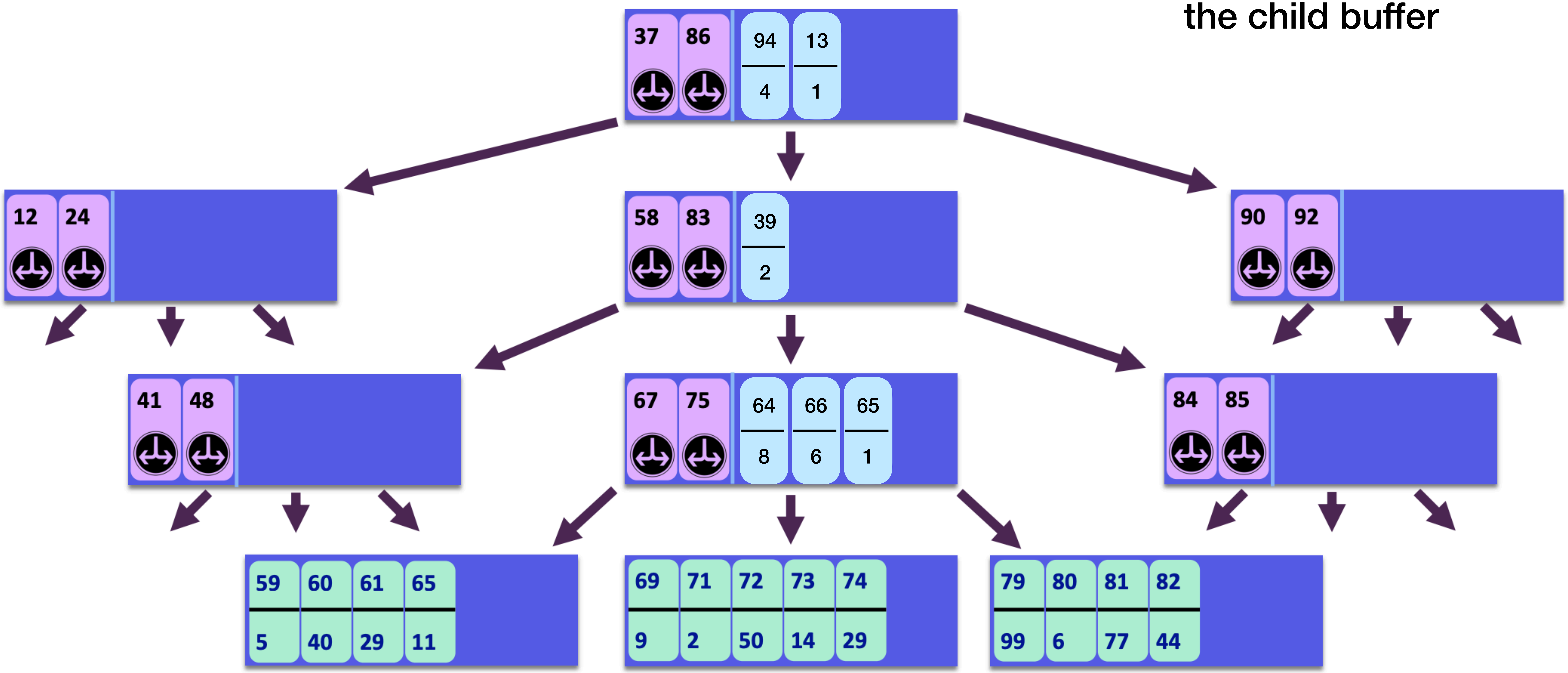


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Insertion in B^{ε} trees

When a buffer is full:

1. Pick the child receiving the most messages, and
2. Move the messages to the child buffer

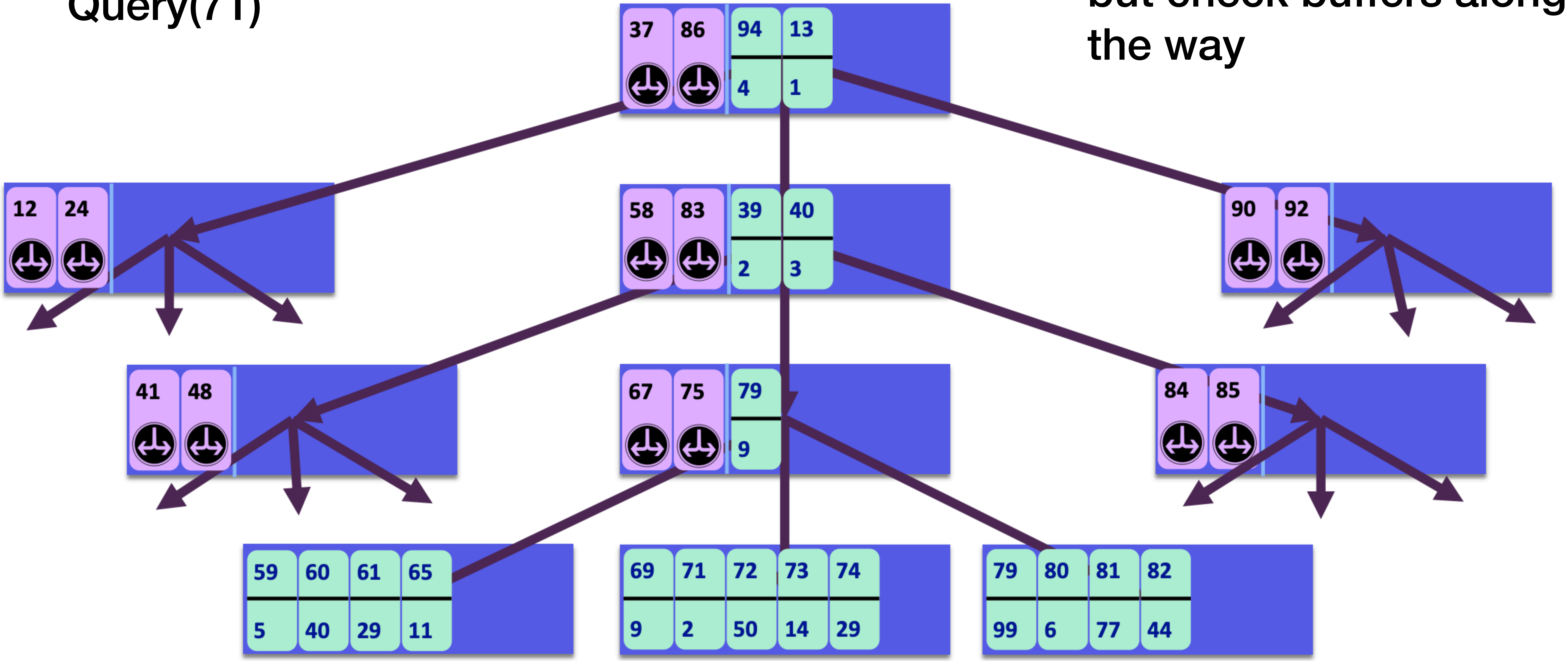


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Lookups in B^{ε} trees

Query(71)

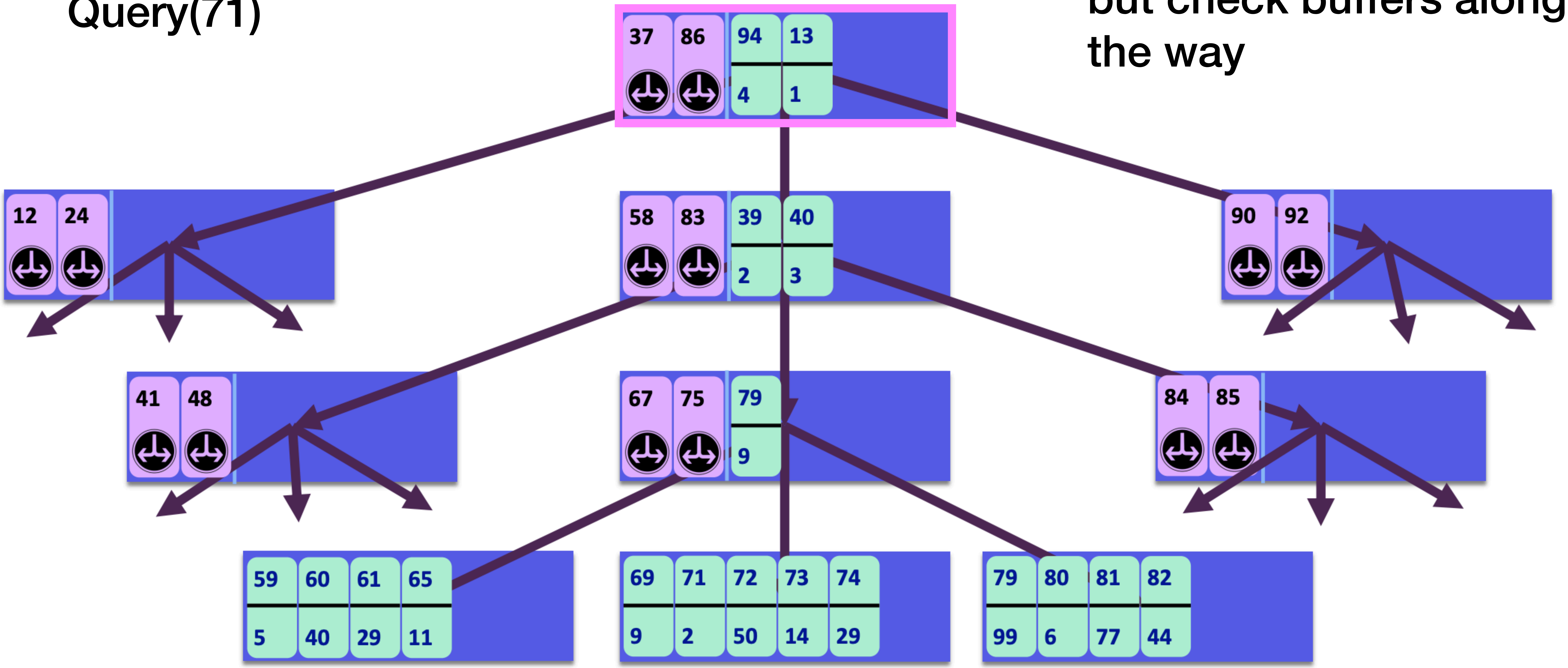
Lookups follow pivots, but check buffers along the way



Lookups in B^{ε} trees

Query(71)

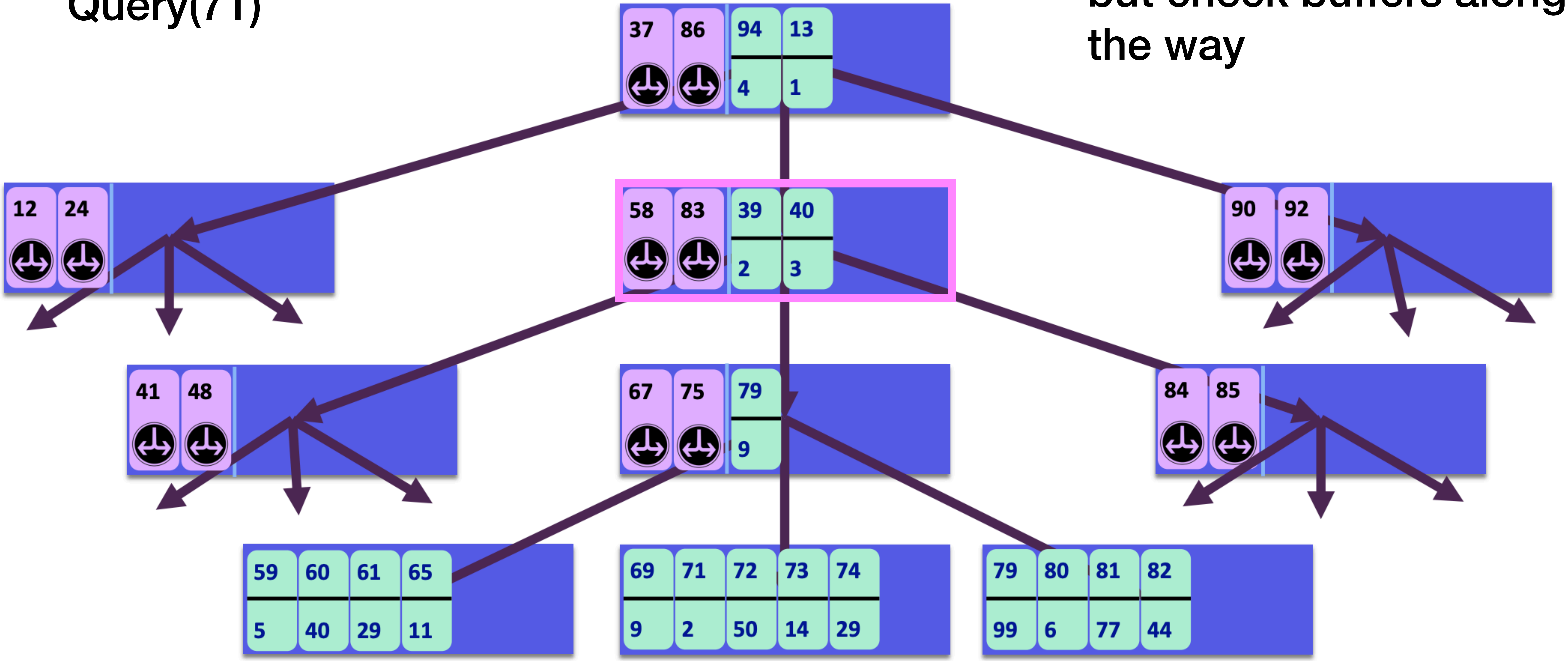
Lookups follow pivots, but check buffers along the way



Lookups in B^{ε} trees

Query(71)

Lookups follow pivots, but check buffers along the way

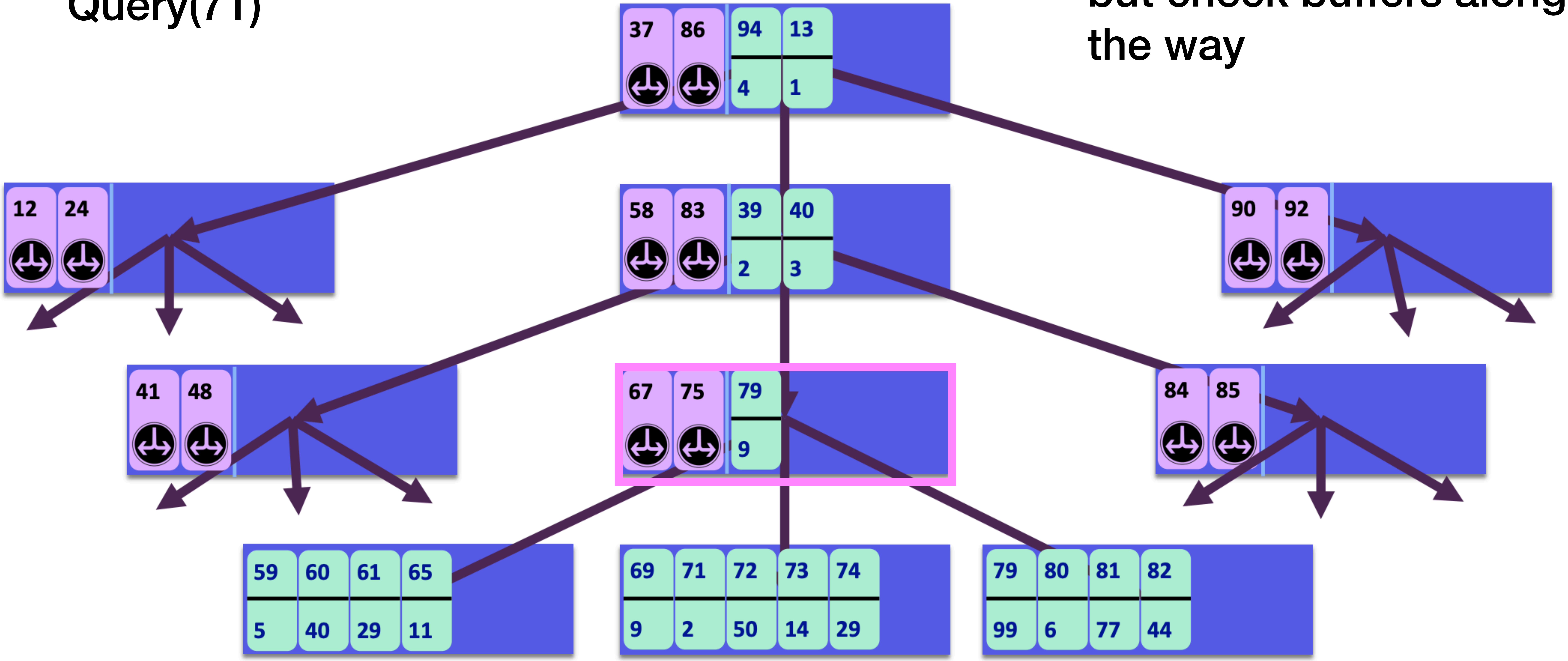


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Lookups in B$^{\varepsilon}$ trees

Query(71)

Lookups follow pivots, but check buffers along the way

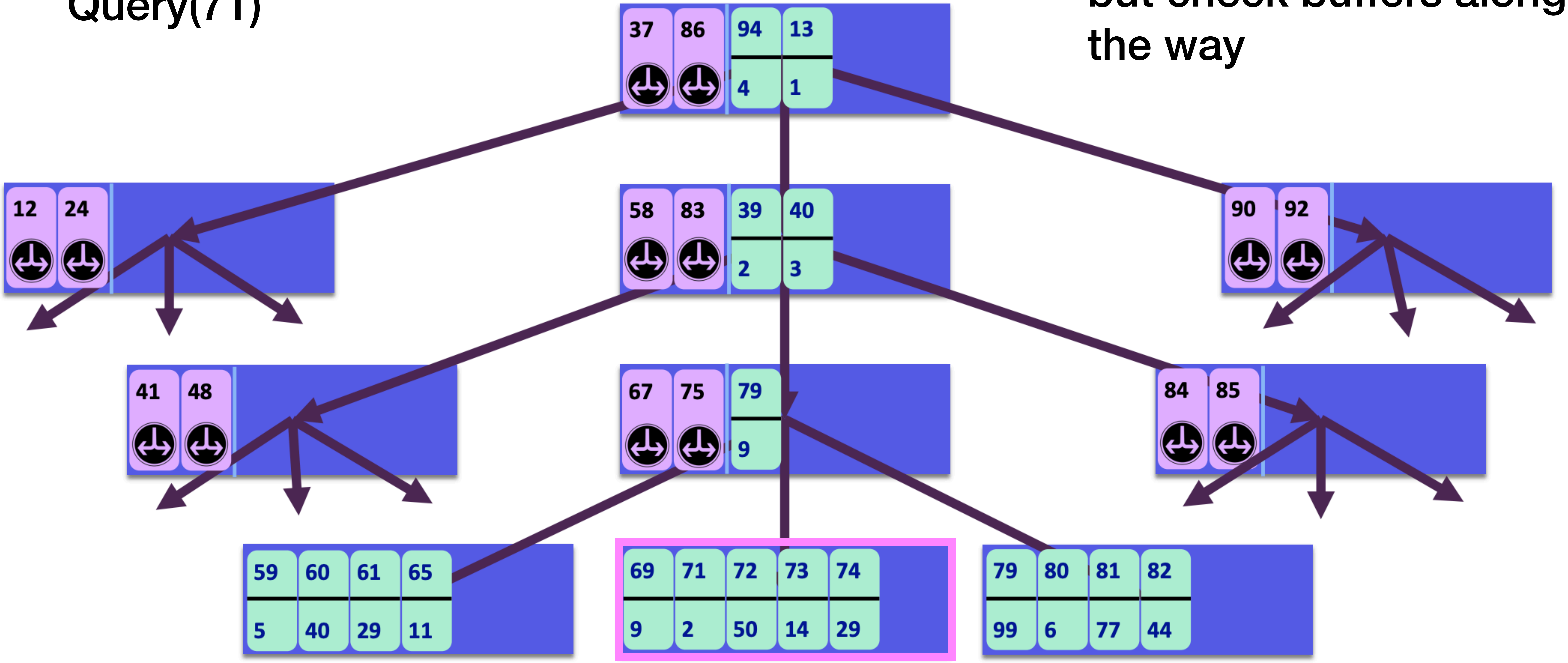


Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Lookups in B$^{\varepsilon}$ trees

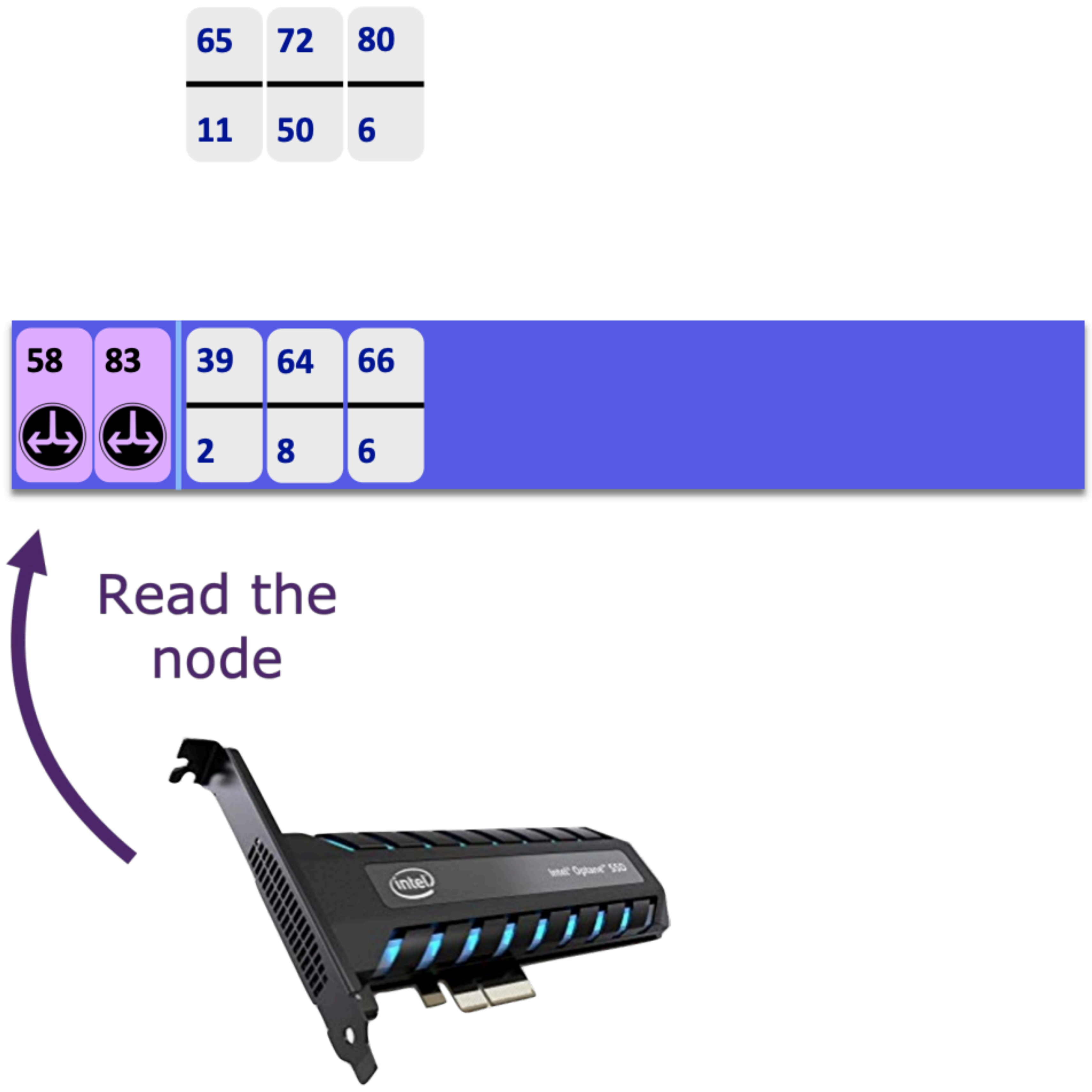
Query(71)

Lookups follow pivots, but check buffers along the way



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

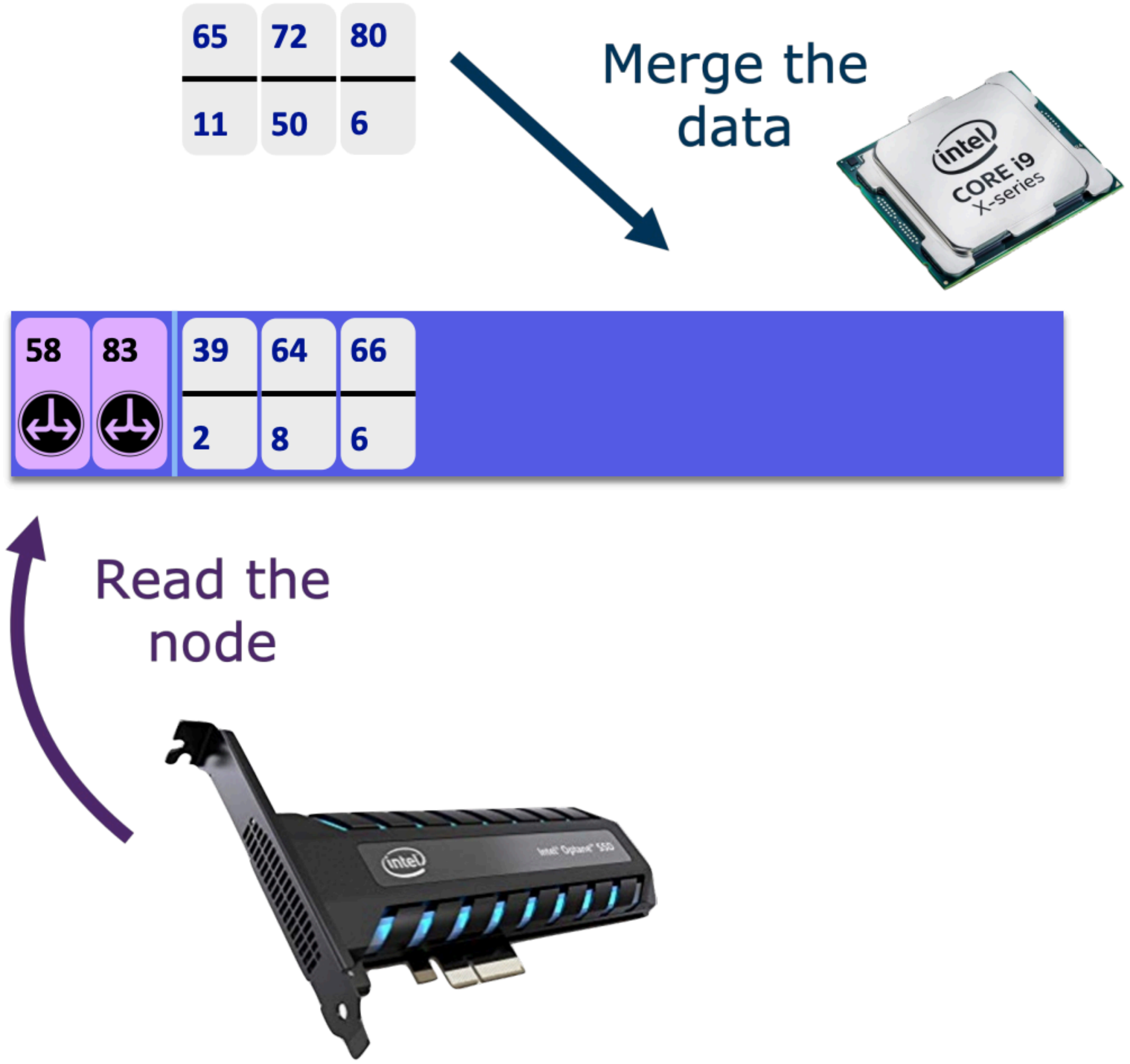
Recall: Insertions in B$^{\varepsilon}$ trees



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

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

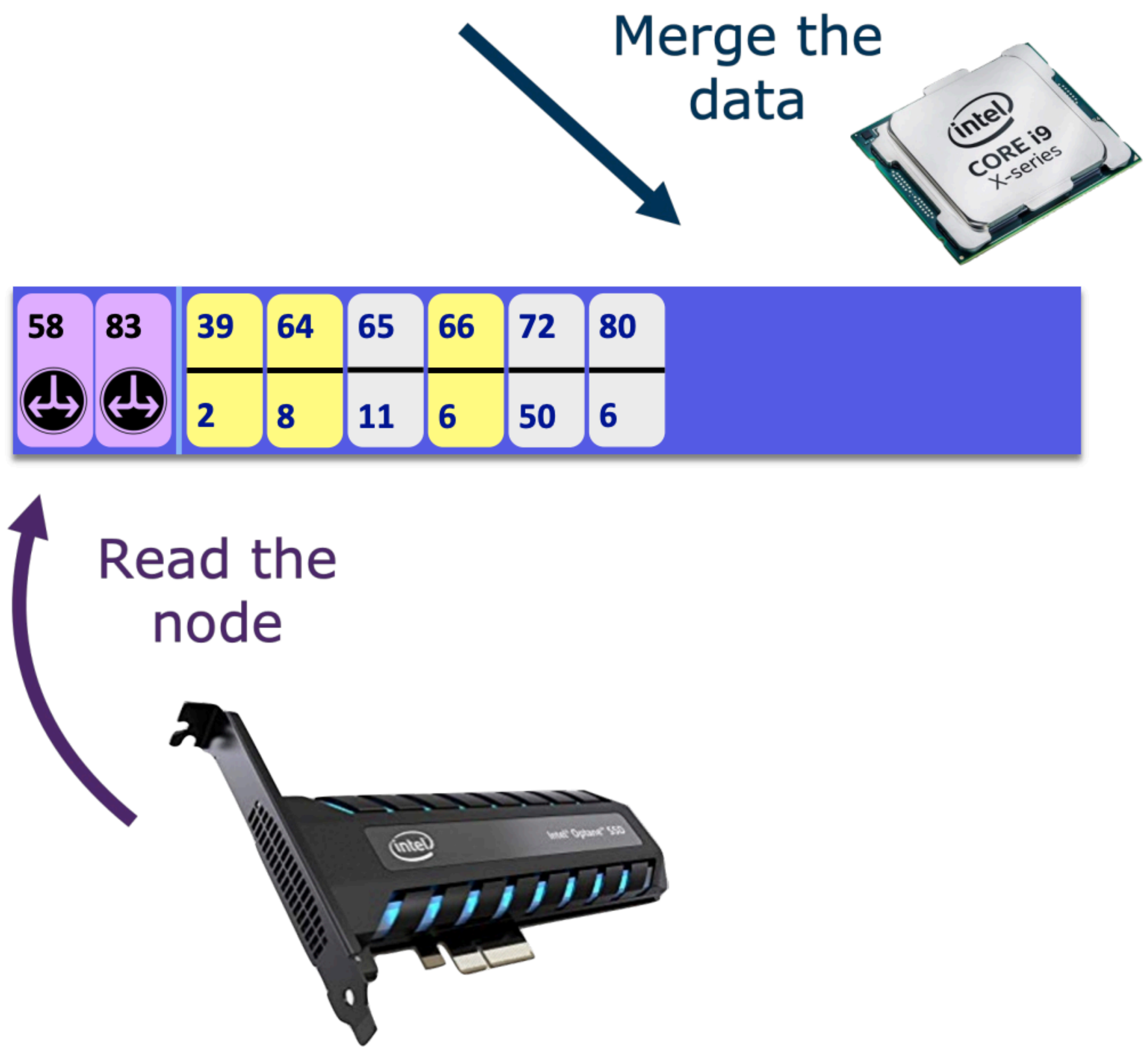
Recall: Insertions in B$^{\varepsilon}$ trees



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

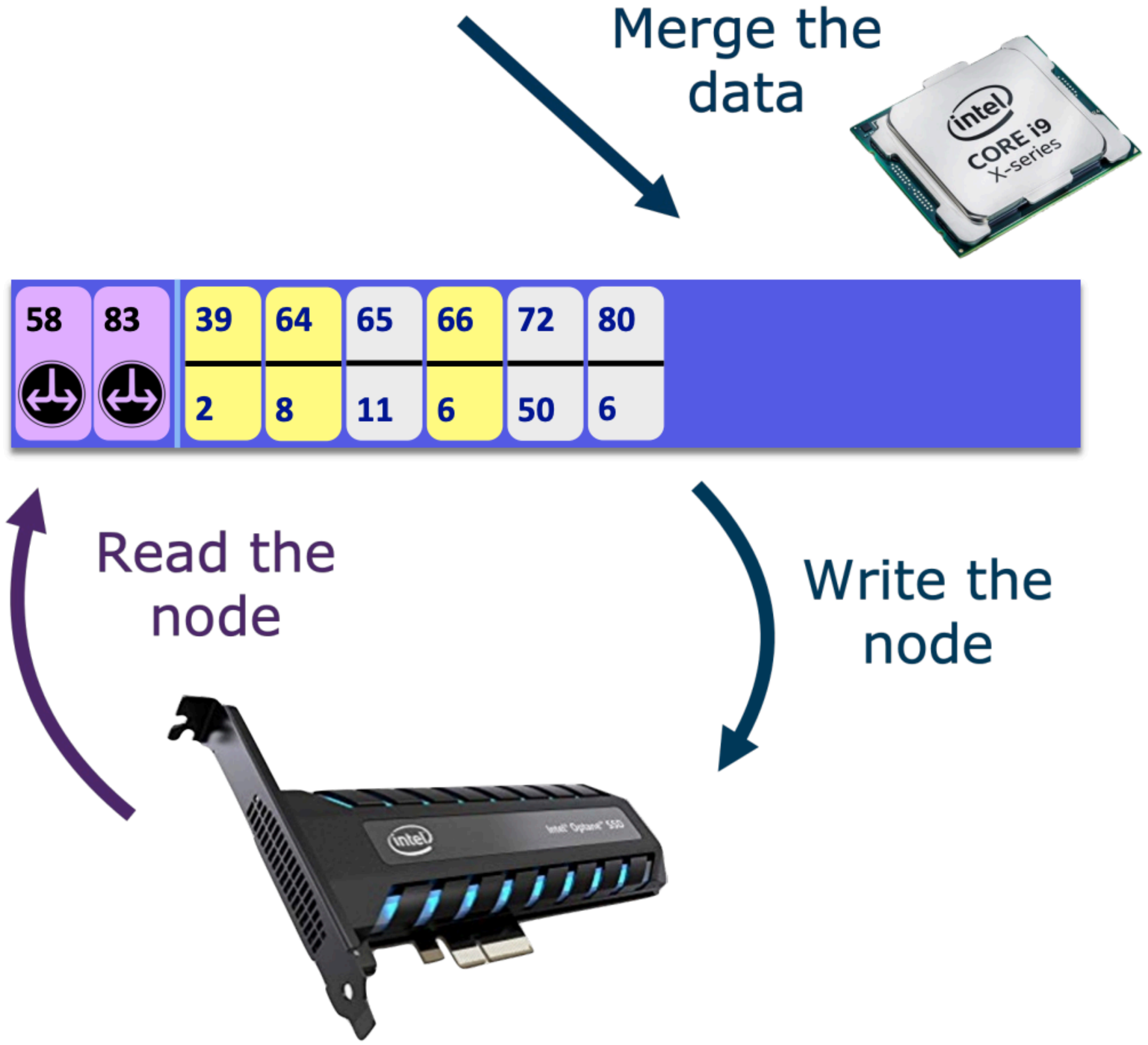
Insertions in B^{ε} trees are more expensive than they look

Recall: Insertions in B^{ε} trees



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

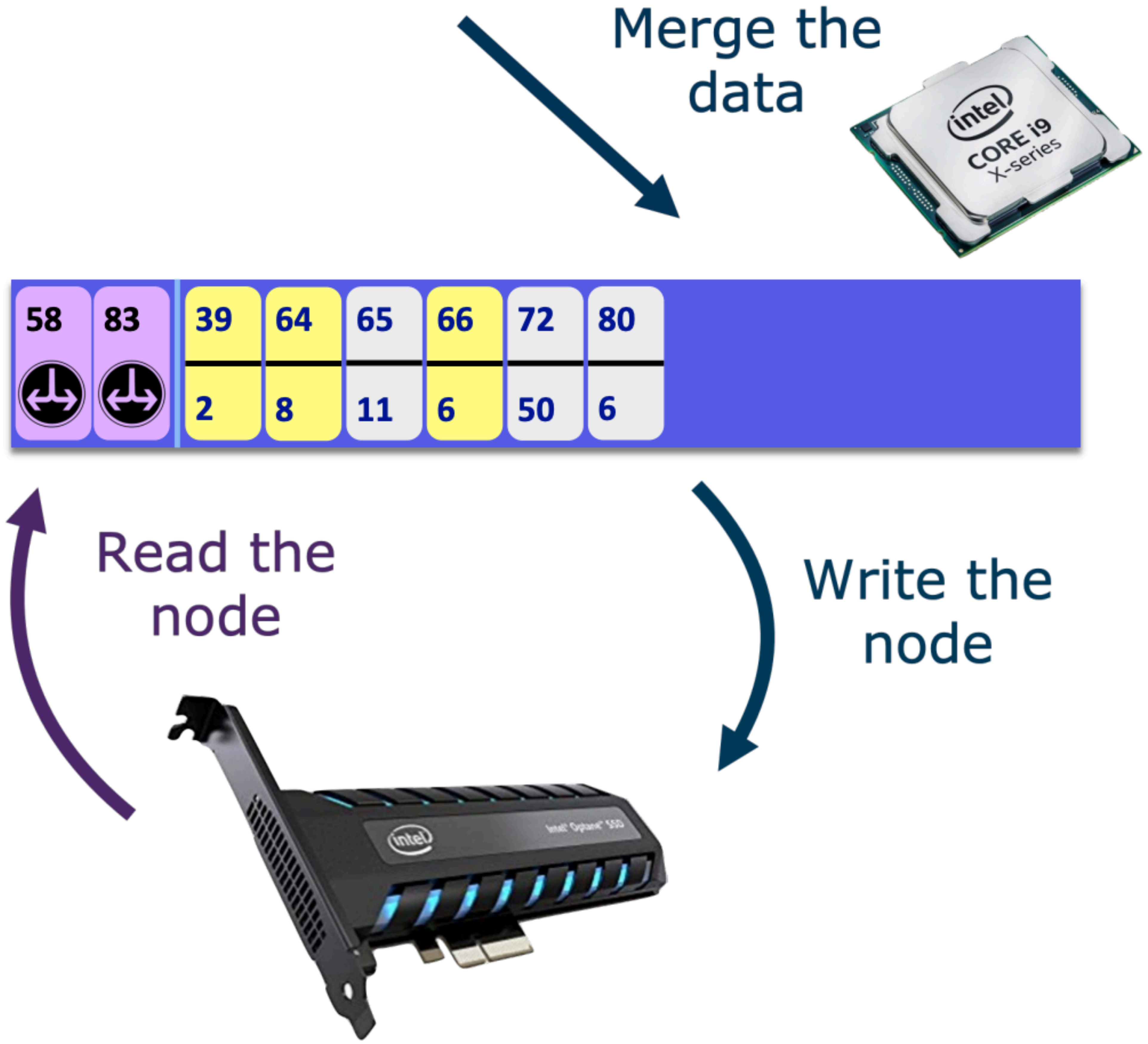
Recall: Insertions in B^{ε} trees



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

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

Recall: Insertions in B^{ε} trees

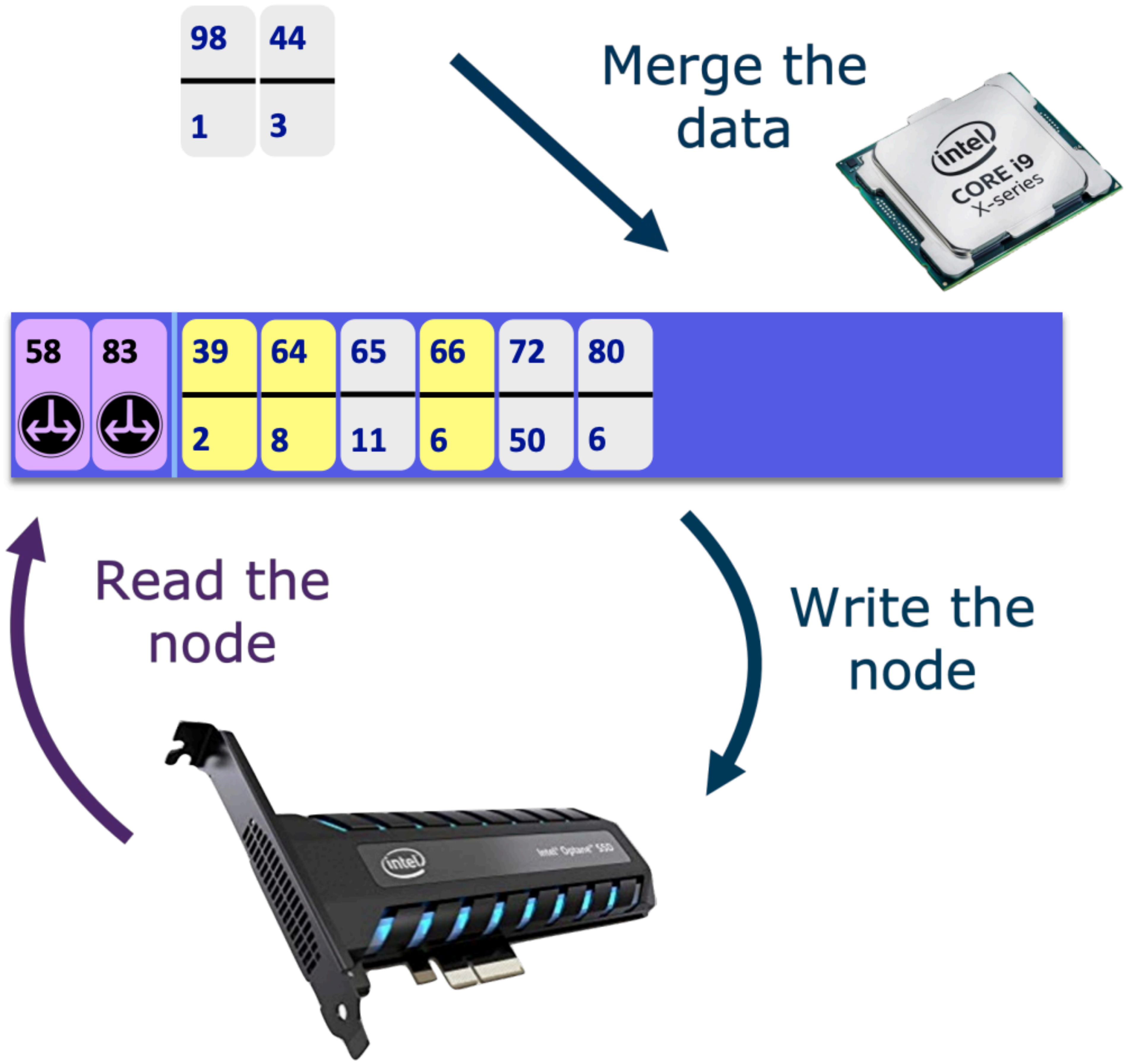


CPU work = O(old + new messages)

Volume of IO = O(old + new messages)

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

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



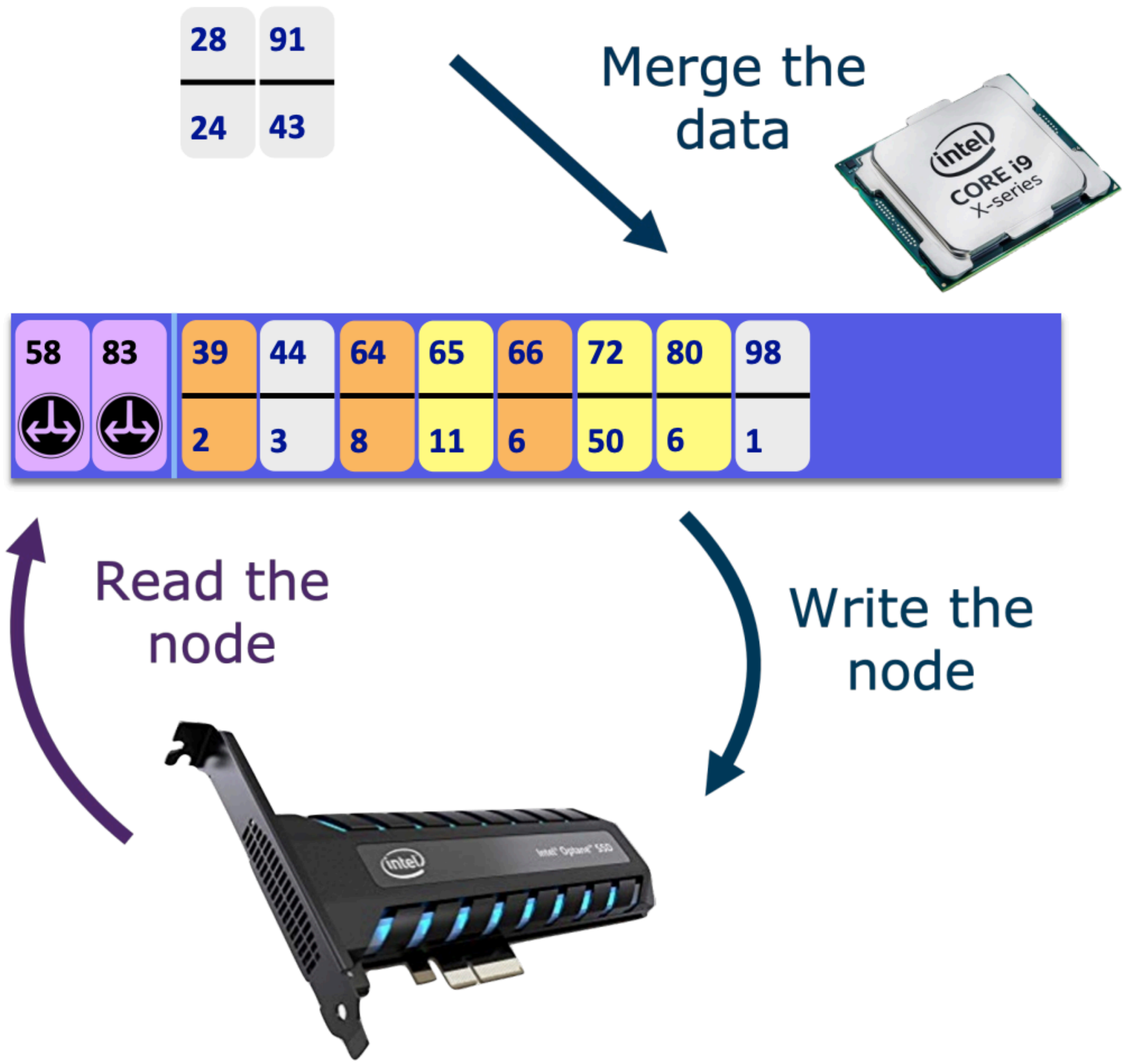
CPU work = O(old + new messages)

Volume of IO = O(old + new messages)

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 messages)

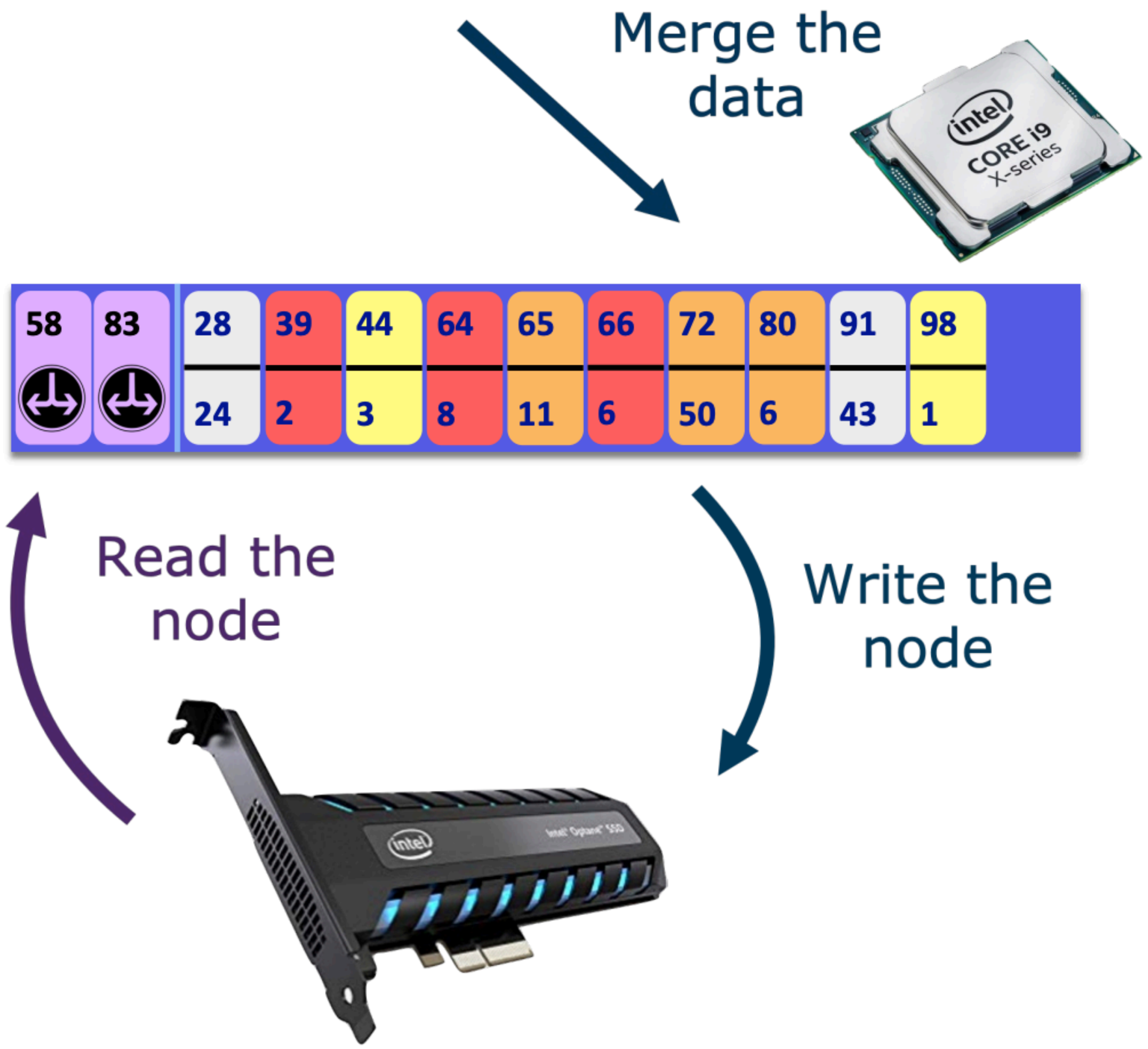
Volume of IO = O(old + new messages)

Older data gets written over and over again

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

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

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



CPU work = O(old + new messages)

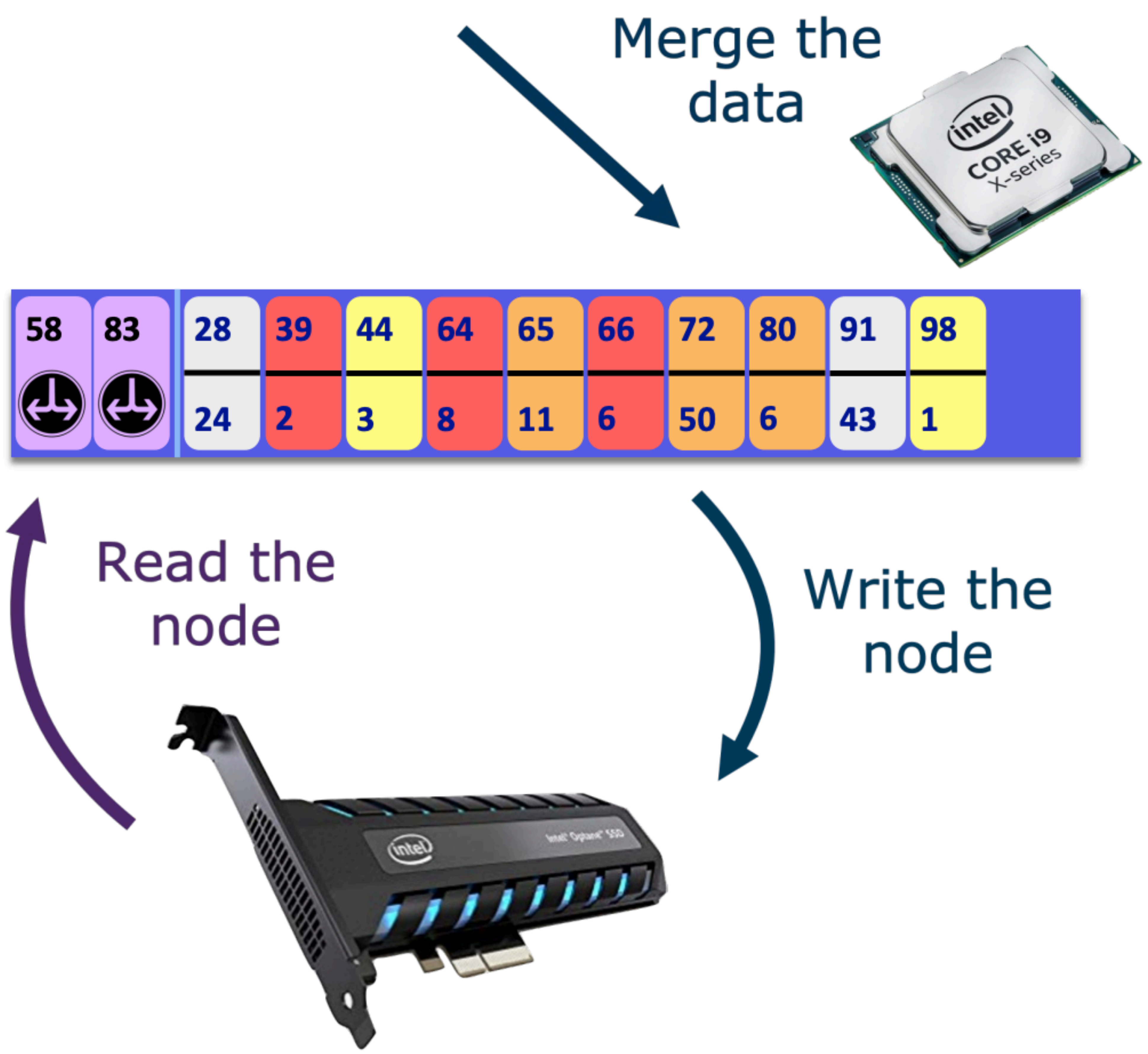
Volume of IO = O(old + new messages)

Older data gets written over and over again

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

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

Recall: Insertions in B^{ε} trees



CPU work = O(old + new messages)

Volume of IO = O(old + new messages)

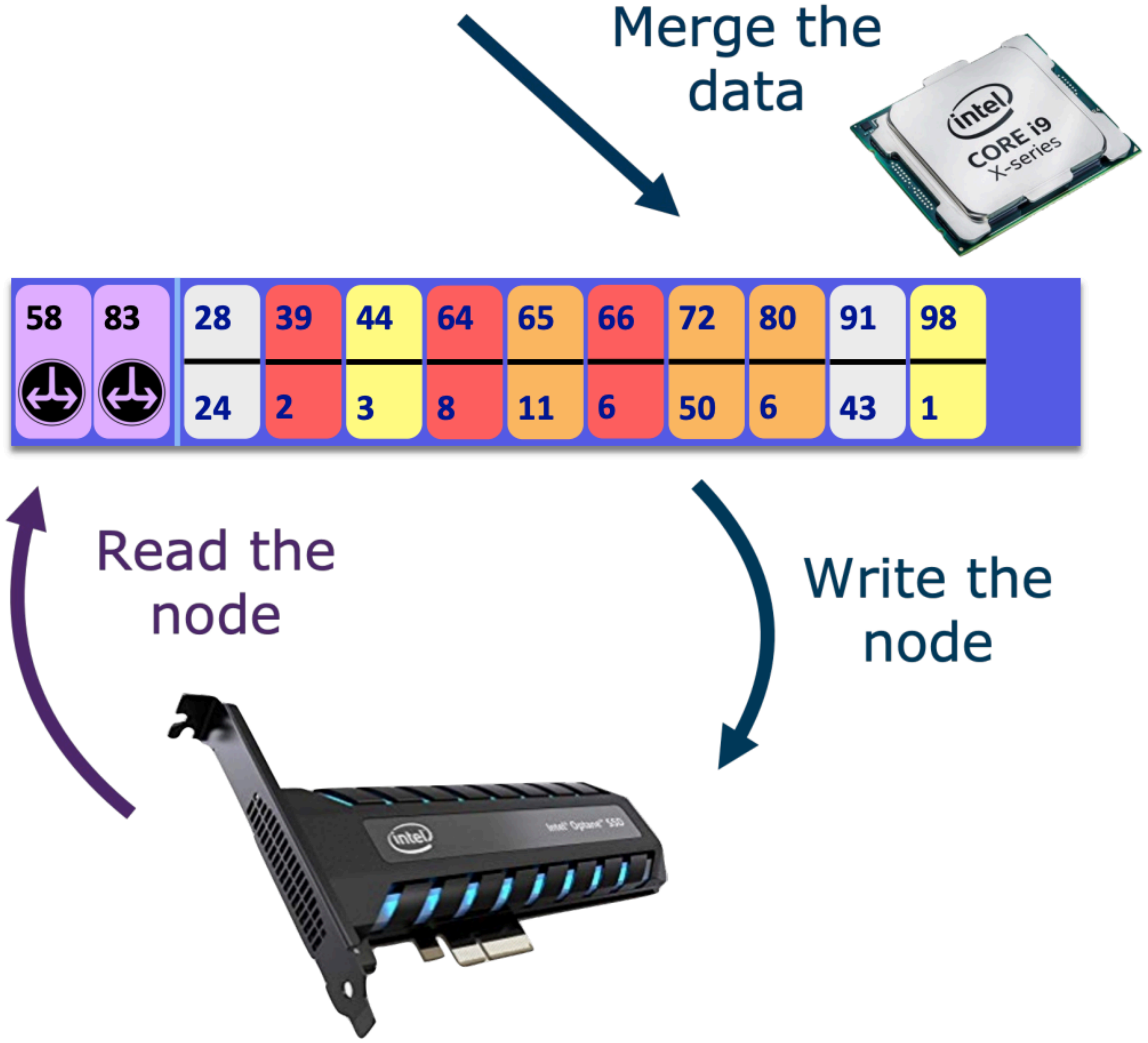
Older data gets written over and over again

Up to B^{ε} times per node!

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

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

Recall: Insertions in B$^\varepsilon$ trees



CPU work = O(old + new messages)

Volume of IO = O(old + new messages)

Older data gets written over and over again

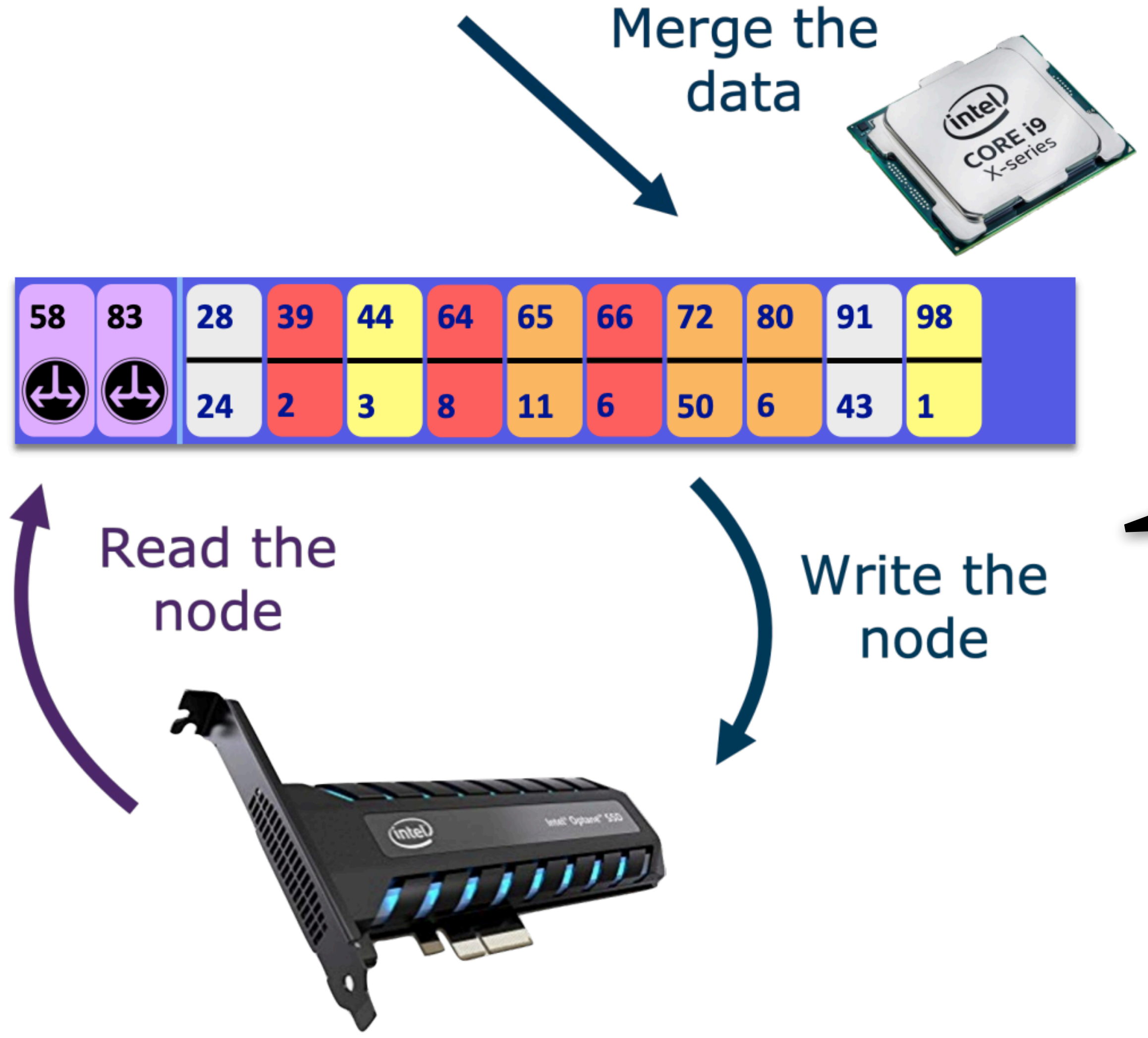
Up to B$^\varepsilon$ times per node!

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.

Shortens disk (e.g., flash, SSD) lifespan



Buffering increases write amplification

I/O amplification

Read amplification is the ratio of the number of blocks read from the disk vers

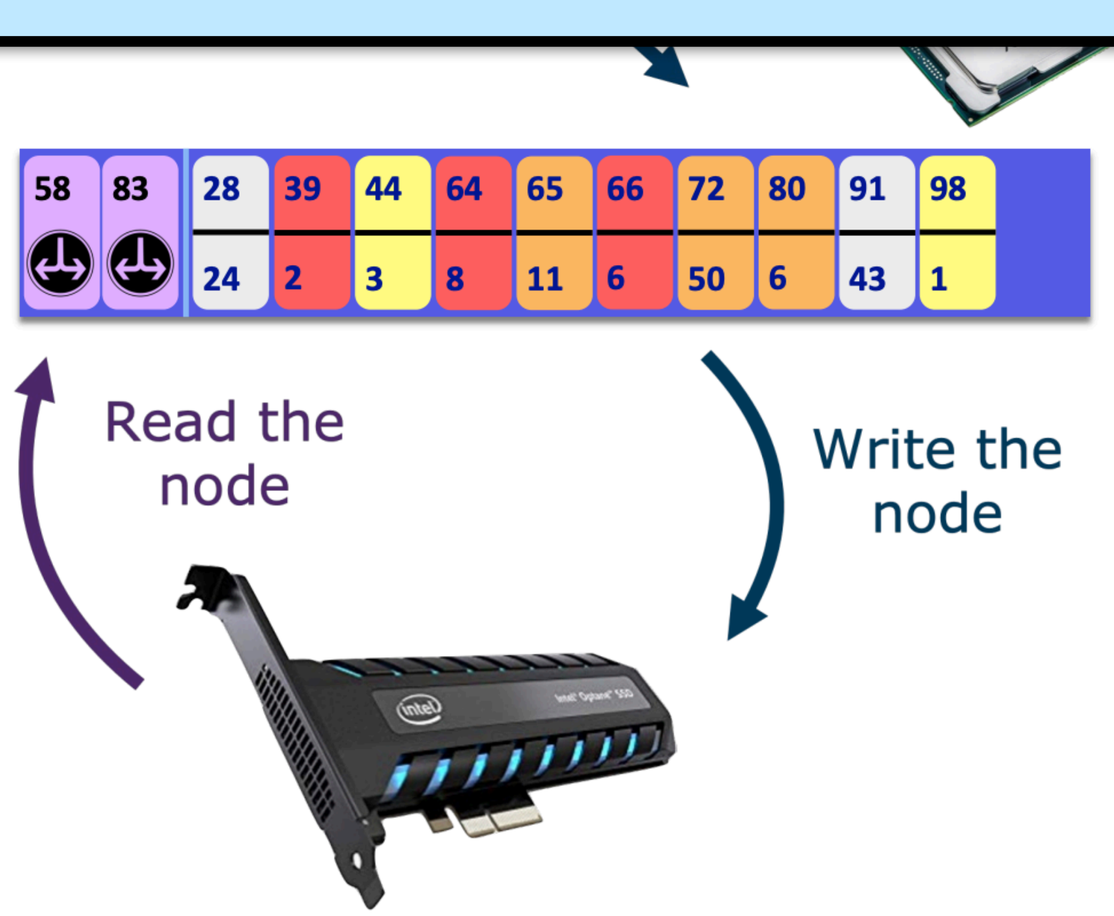
Wri k

vers

Shorte
(e.g., fla
lifespan

How to fix it?

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



Buffering increases write amplification