CS 6530: Advanced Database Systems Fall 2022

Lecture 03
In-memory indexing
(Trees, Tries, Skip Lists)

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Andy Pavlo, CMU/ Manos Athanassoulis, BU

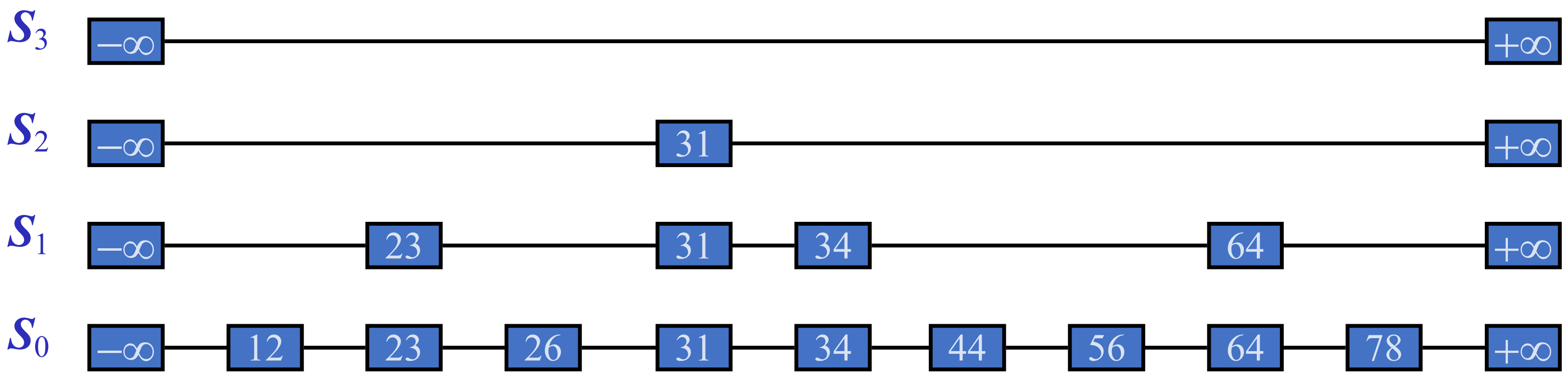
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Some reminders…

- Paper report #1 due today deadlines are posted
- Project #1 posted

What is a Skip List

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \dots, S_h$ 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.,
    $$S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$$
  - List S_h contains only the two special keys
- Skip lists are one way to implement the dictionary

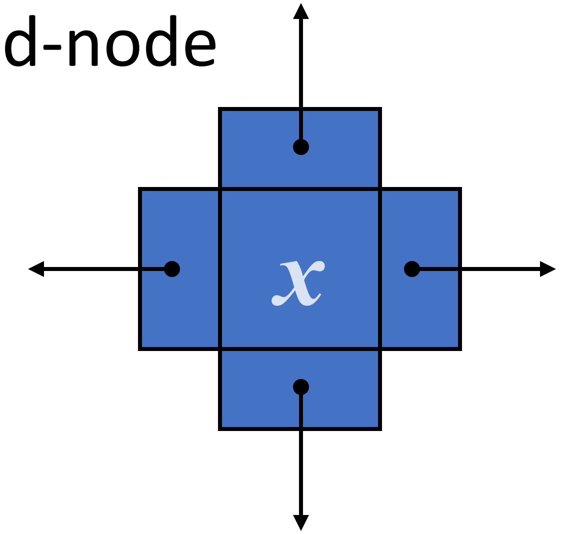


Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

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

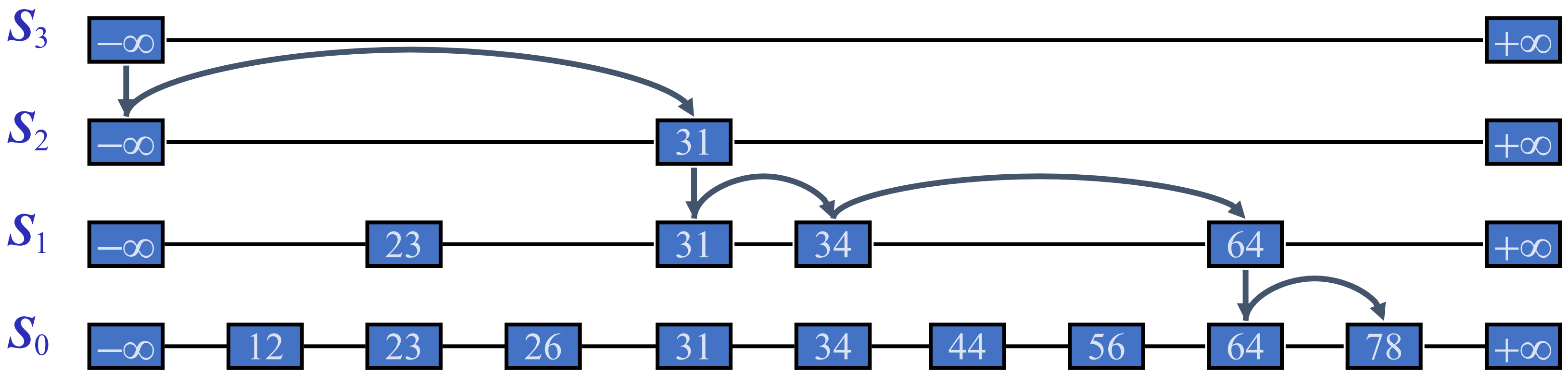
Search

- 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"

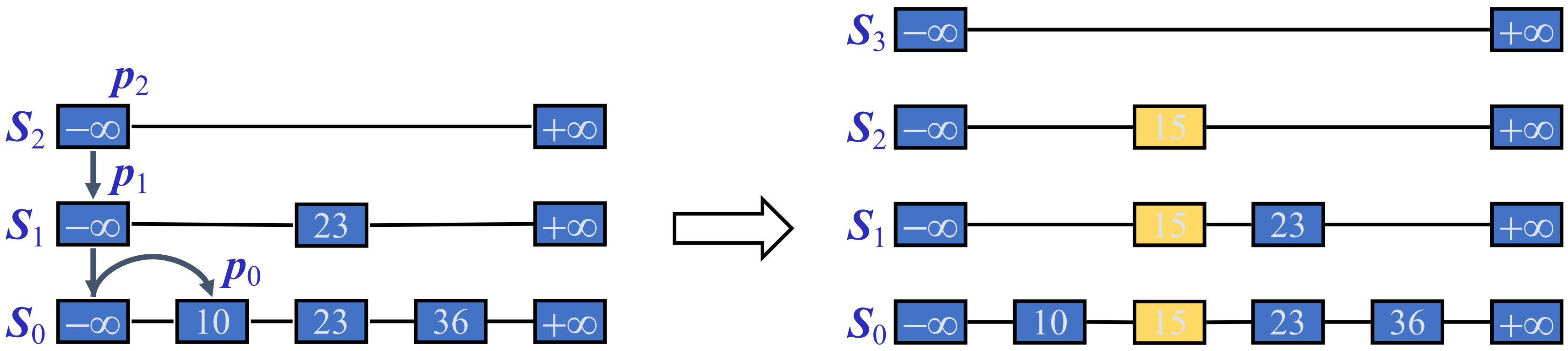
    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY
- Example: search for 78



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

Insertion

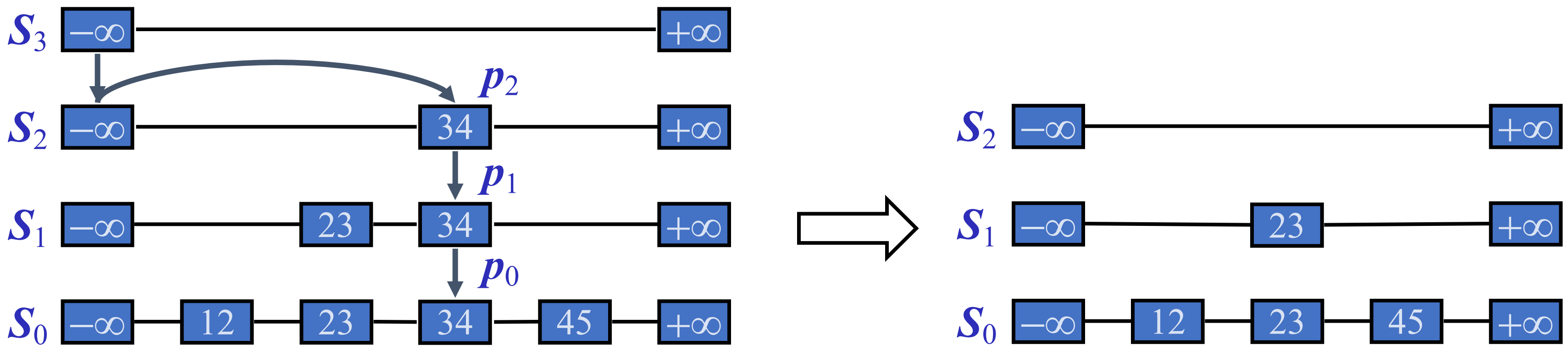
- To insert an item (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special 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

Deletion

- To remove an item with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



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:

  $b \leftarrow \textbf{\textit{randomBit}}()$

  if $b = 0$

  do A …

  else $\{ b = 1 \}$

  do B …

- Its running time depends on the outcomes of the coin tosses

- 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
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")
- We use a randomized algorithm to insert items into a skip list to insert in expected O(log n)-time
- When randomization is used in data structures they are referred to as probabilistic data structures

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

Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
- We use the following two basic probabilistic facts:

  Fact 1: The probability of getting $\boldsymbol{i}$ consecutive heads when flipping a coin is $1/2^{\boldsymbol{i}}$

  Fact 2: If each of $\boldsymbol{n}$ items is present in a set with probability $\boldsymbol{p}$, the expected size of the set is $\boldsymbol{np}$

- Consider a skip list with $\boldsymbol{n}$ items
  - By Fact 1, we insert an item in list $\boldsymbol{S_i}$ with probability $1/2^{\boldsymbol{i}}$
  - By Fact 2, the expected size of list $\boldsymbol{S_i}$ is $\boldsymbol{n}/2^{\boldsymbol{i}}$
- The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with $\boldsymbol{n}$ items is $\boldsymbol{O}(\boldsymbol{n})$

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

Height

- The running time of the search and insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(\log n)$
- We use the following additional probabilistic fact:

  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most

$$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus, a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

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

Height

- The running time of the search and insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(\log n)$
- We use the following additional probabilistic fact:

  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most

  $$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus, a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

With High Probability (WHP)

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

Height

- The running time of the search and insertion algorithms is affected by the height h of the skip list

Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most

at most

$$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus, a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

With High Probability (WHP)

An event that occurs ***with high probability*** (WHP) is one whose probability depends on a certain number *n* and goes to 1 as *n* goes to infinity. [Wikipedia]

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

Search and Update Times

- The search time in a skip list is proportional to
  - the number of drop-down steps, plus
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability
- To analyze the scan-forward steps, we use yet another probabilistic fact:

  Fact 4: The expected number of coin tosses required in order to get tails is 2

- When we scan forward in a list, the destination key does not belong to a higher list
  - A scan-forward step is associated with a former coin toss that gave tails
- By Fact 4, in each list the expected number of scan-forward steps is 2
- Thus, the expected number of scan-forward steps is $O(\log n)$
- We conclude that a search in a skip list takes $O(\log n)$ expected time
- The analysis of insertion and deletion gives similar results

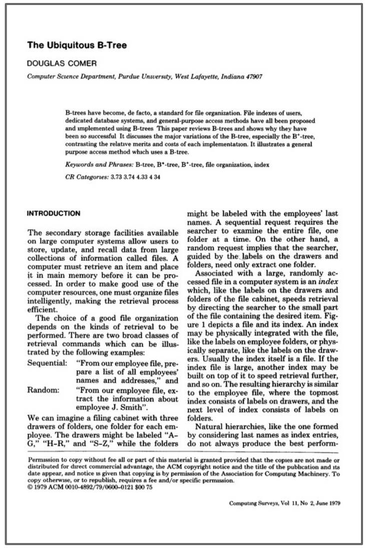
Acknowledgement: https://people.engr.tamu.edu/schaefer/teaching/221_Fall2018/Lectures/SkipLists.ppt

Question?

Are Binary trees and skip lists optimal for in-memory indexing?

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 $\mathbf{O(log_B(N))}$.
  - The fanout of the tree is B
  - Generalization of a binary search tree in that a node can have more than two children.
  - Optimized for systems that read and 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".

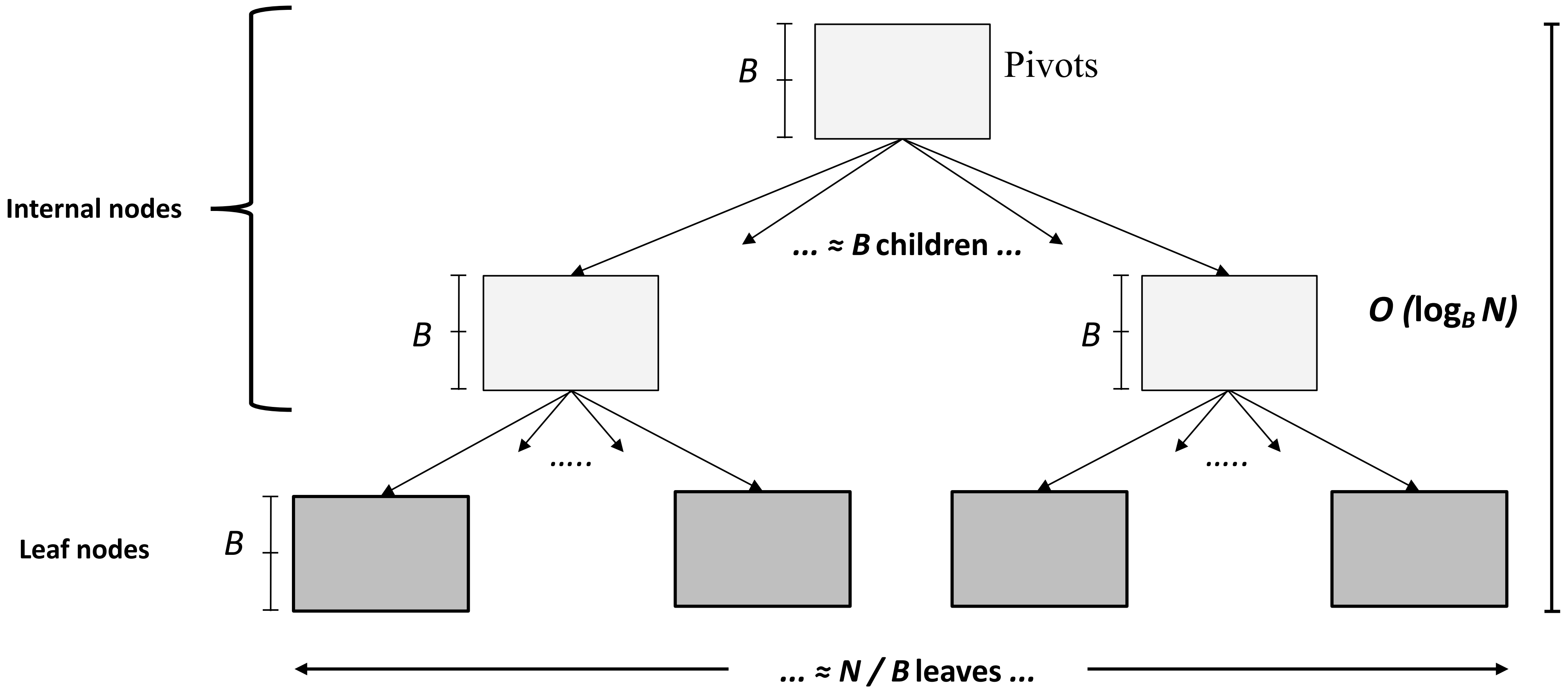
We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a 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-

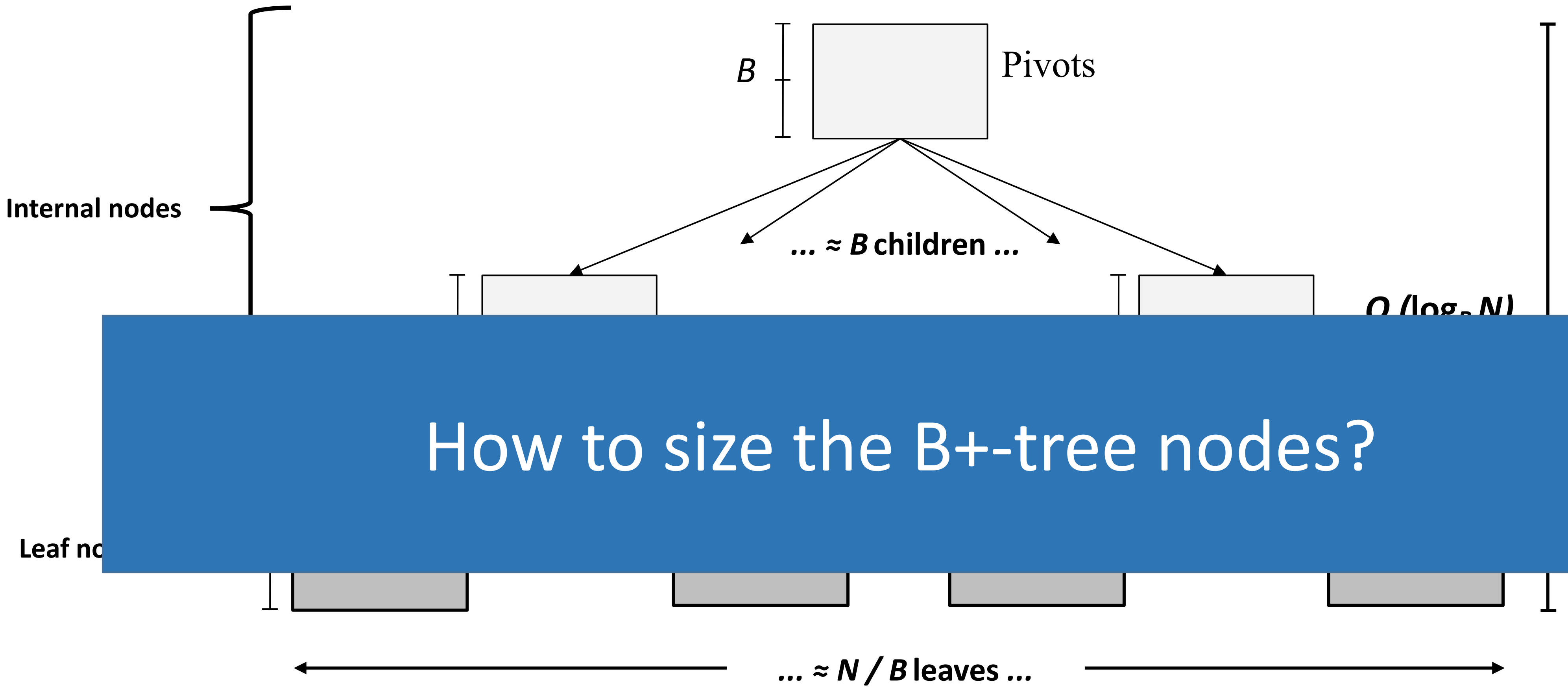
Computing Surveys, Vol 11, No 2, June 1979

B+ Trees



Internal nodes
B
Pivots
... ≈ B children ...
B
B
O (log_B N)
.....
.....
Leaf nodes
B
... ≈ N / B leaves ...

B+ Trees

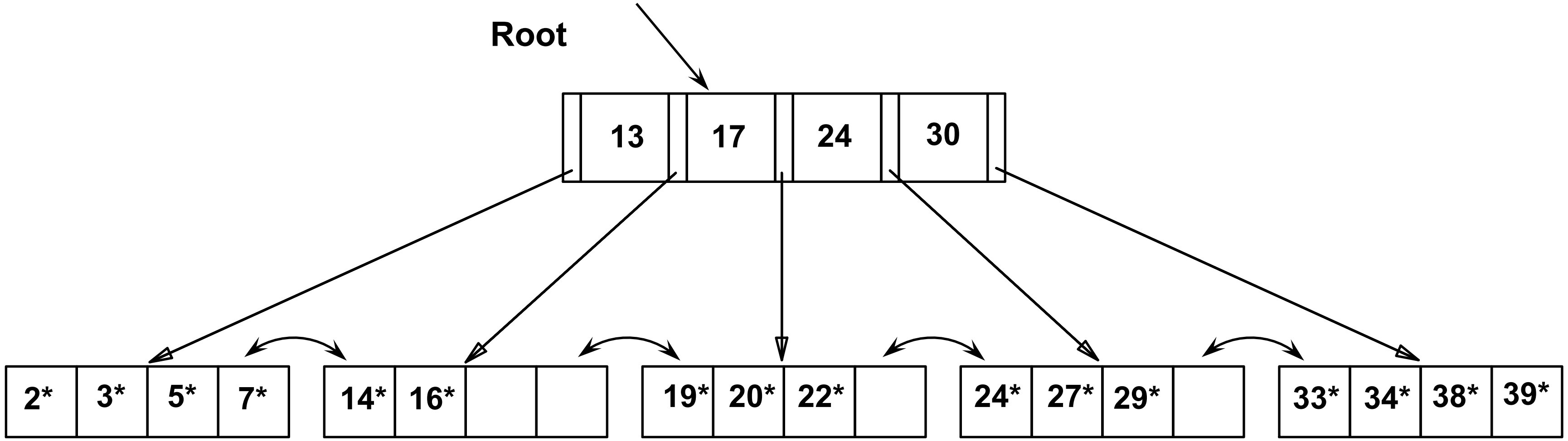


How to size the B+-tree nodes?

B+ Trees

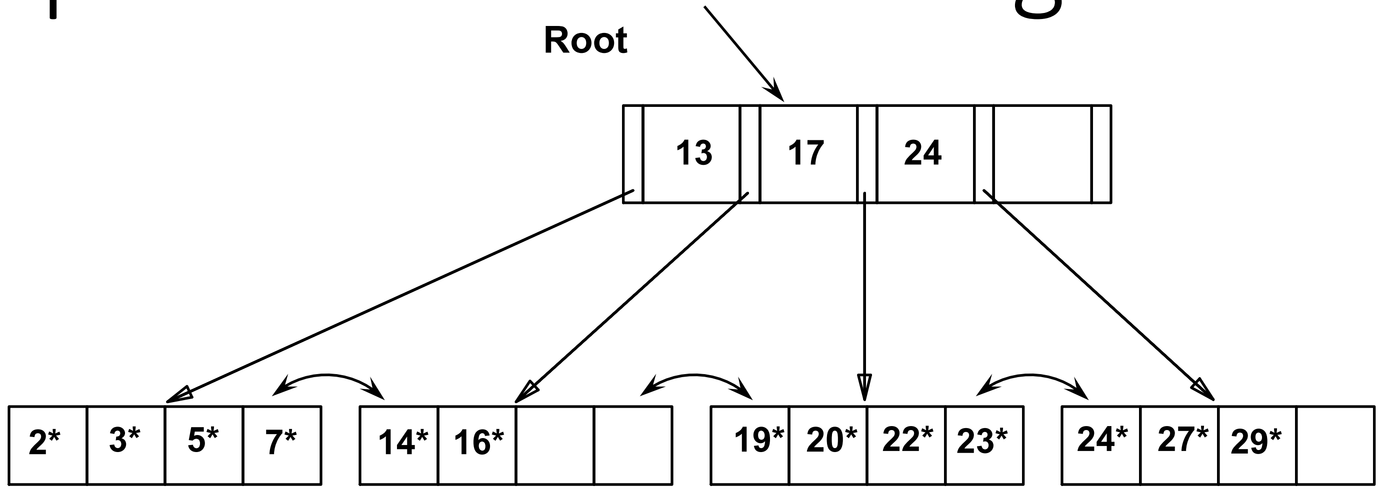
Search begins at root, and key comparisons direct it to a leaf.

Search for 5*, 15*, all data entries >= 24* ...

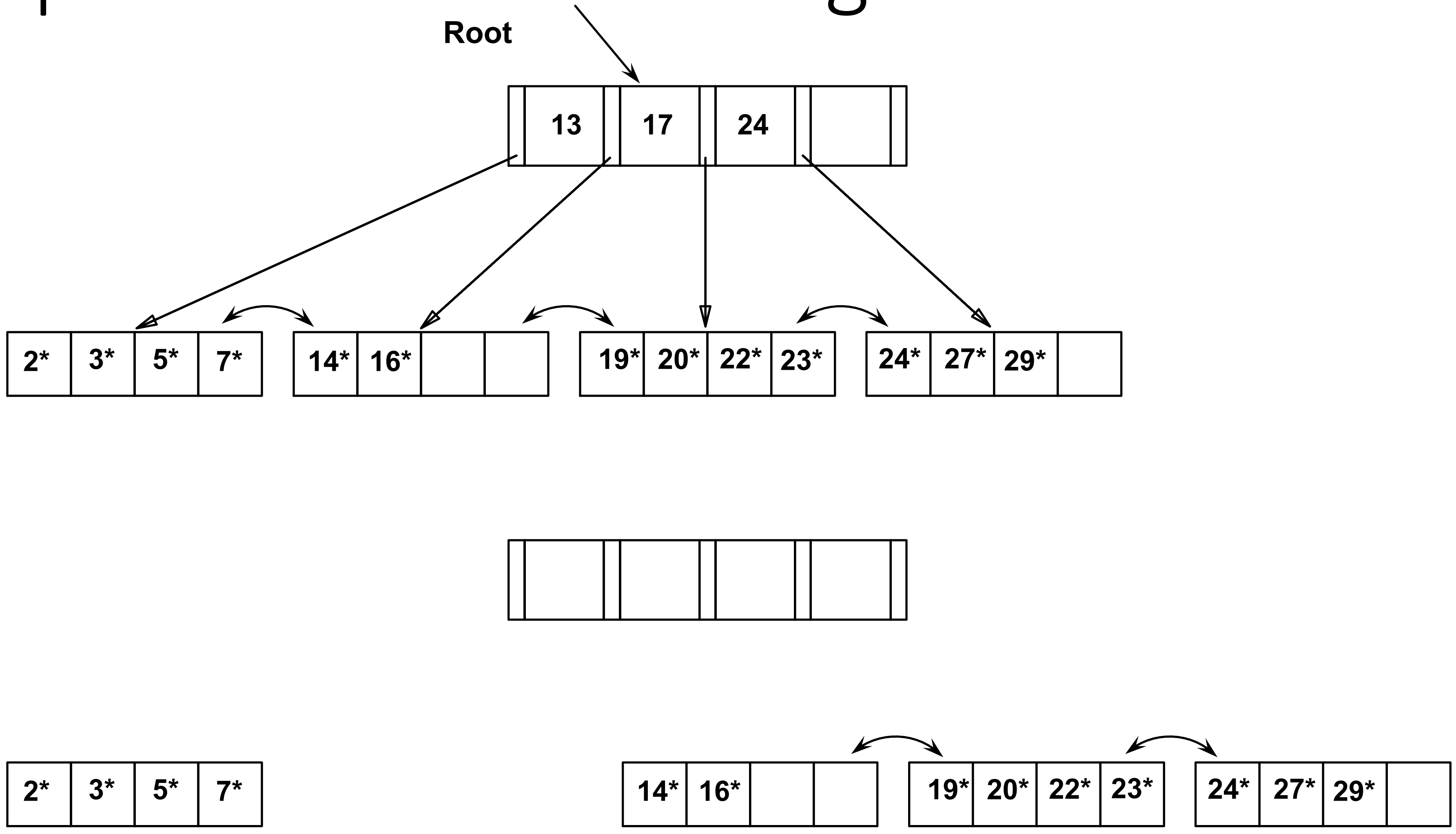


Based on the search for 15, we know it is not in the tree!*

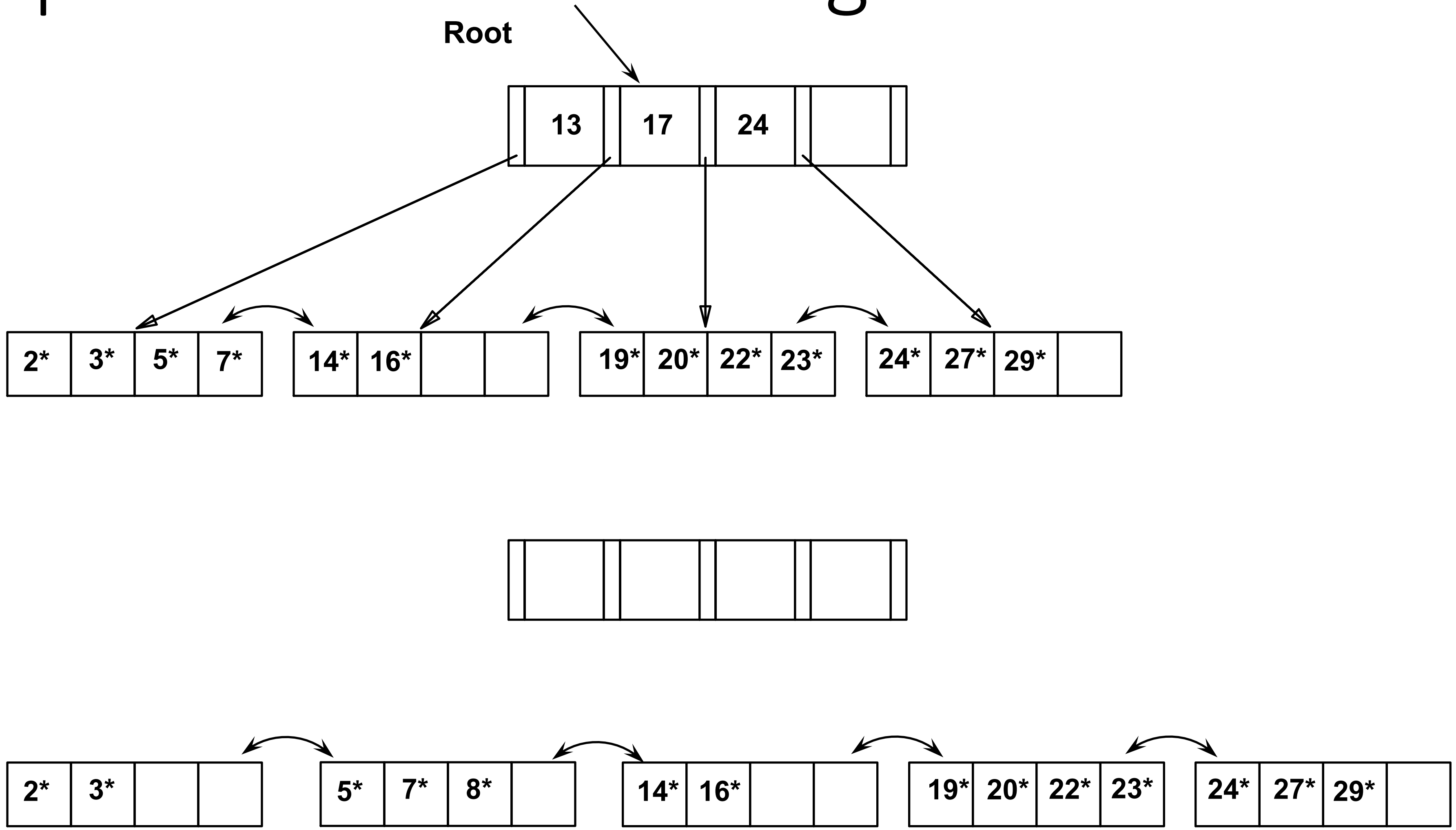
Example B+ Tree - Inserting 8*



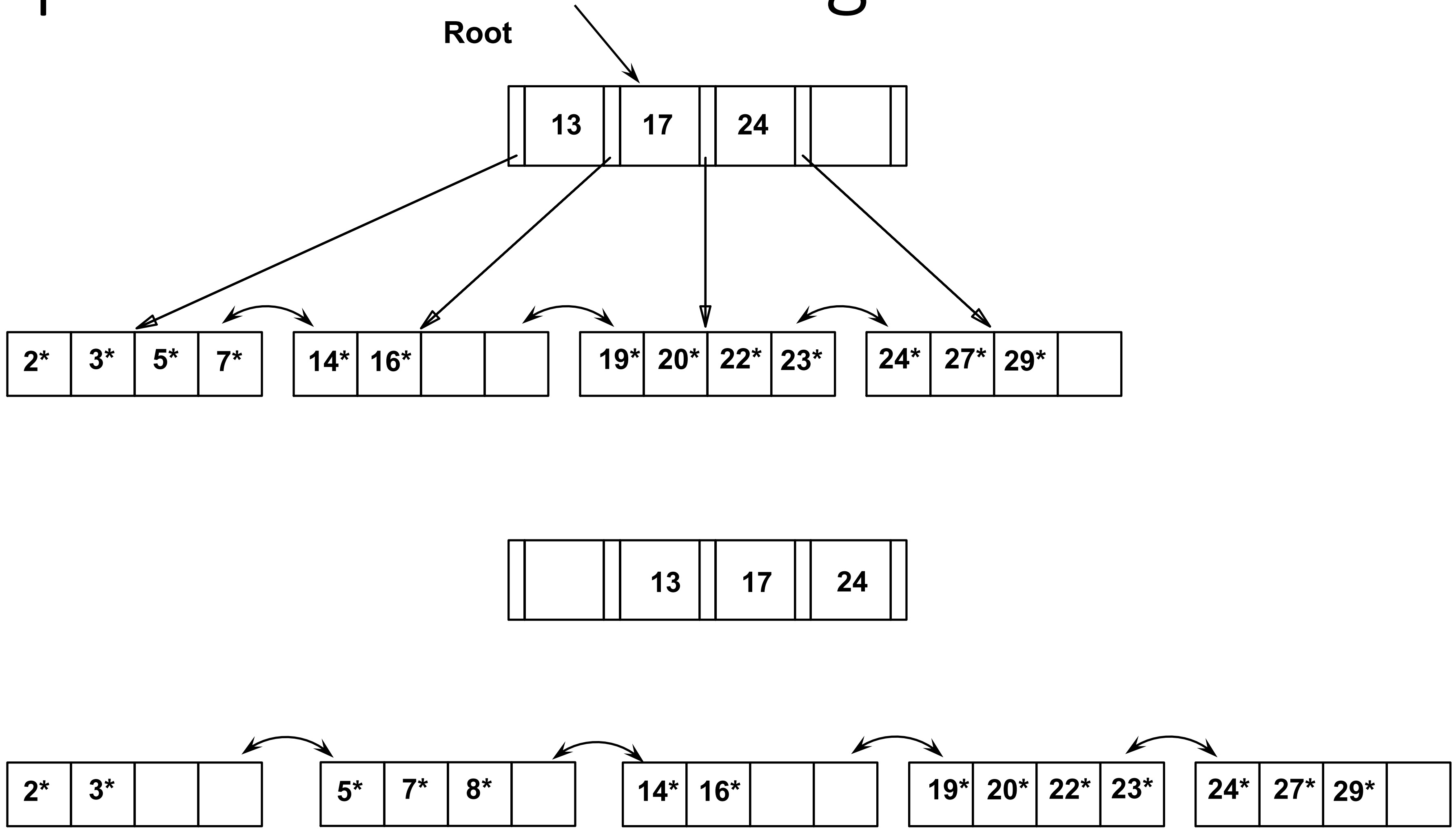
Example B+ Tree - Inserting 8*



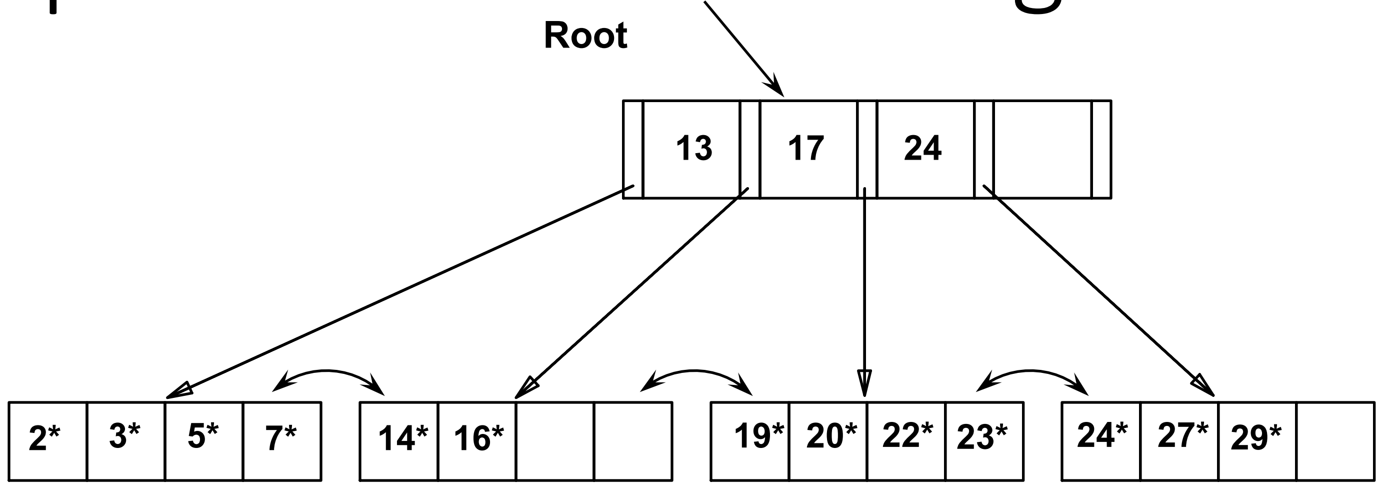
Example B+ Tree - Inserting 8*

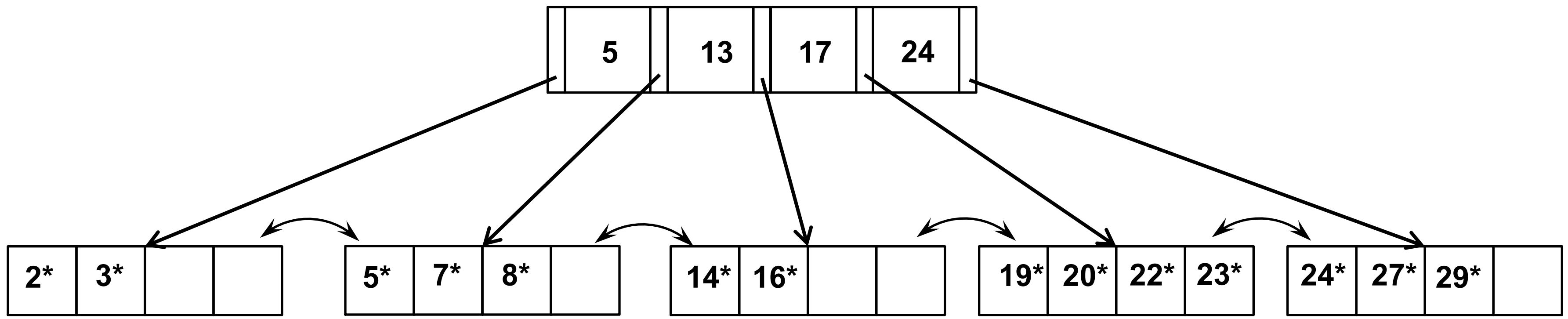


Example B+ Tree - Inserting 8*

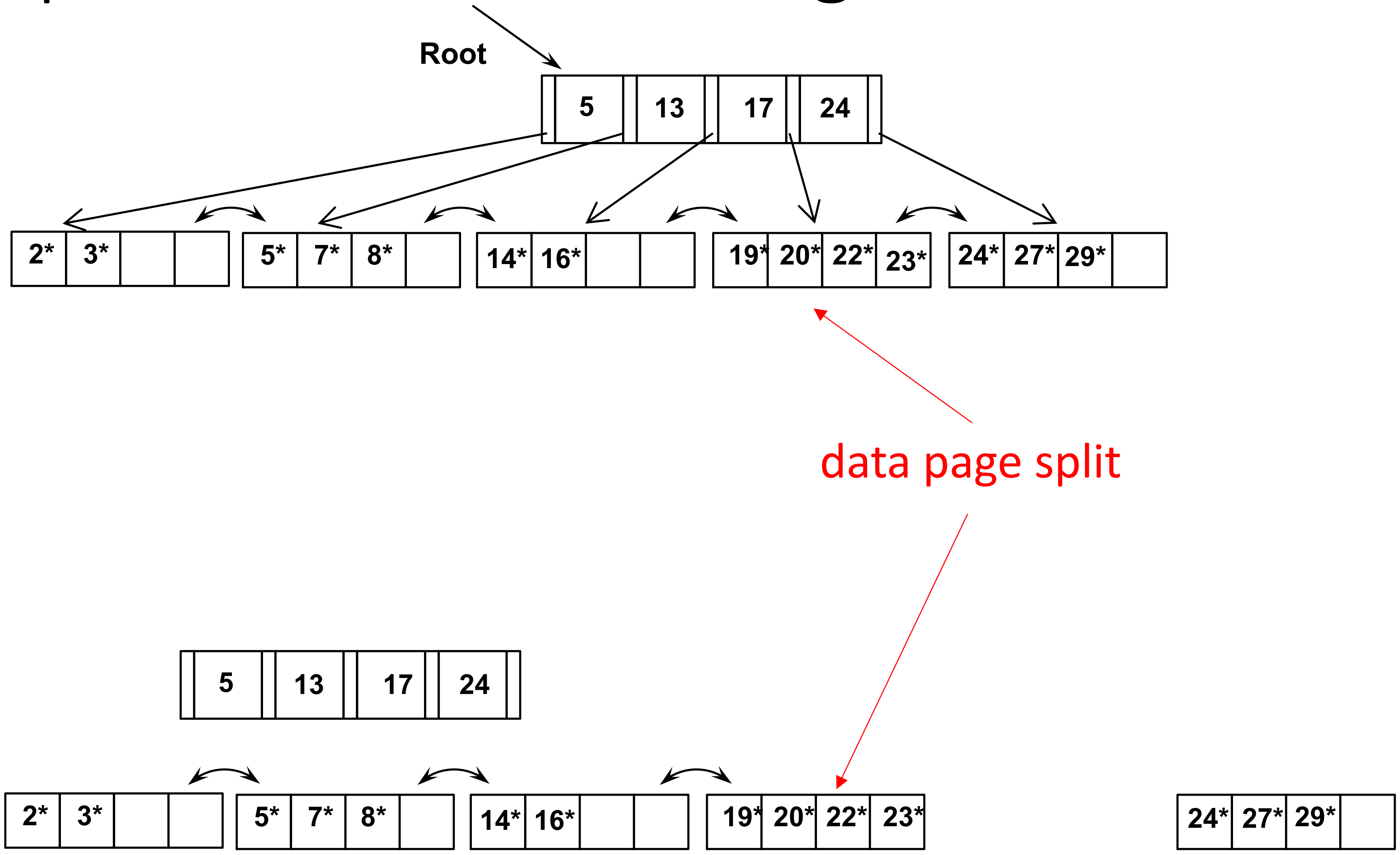


Example B+ Tree - Inserting 8*

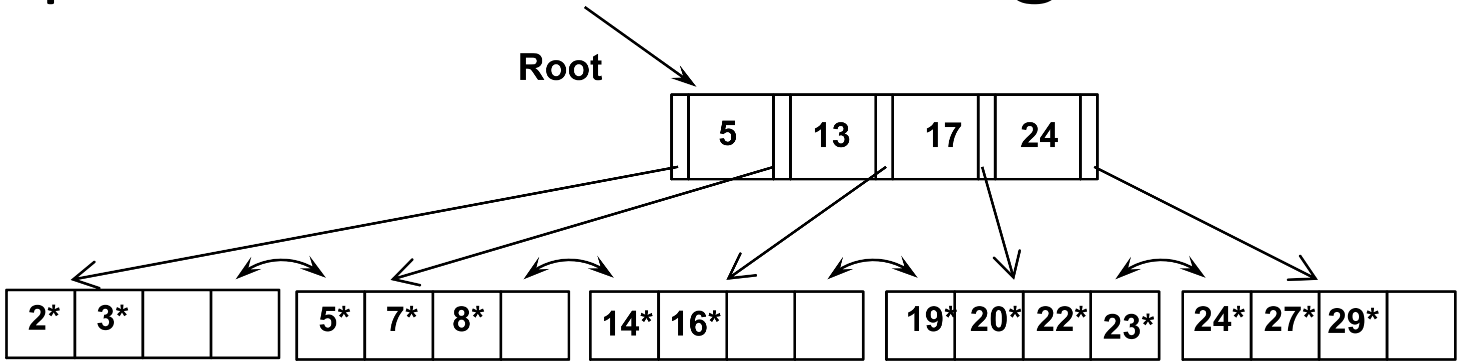


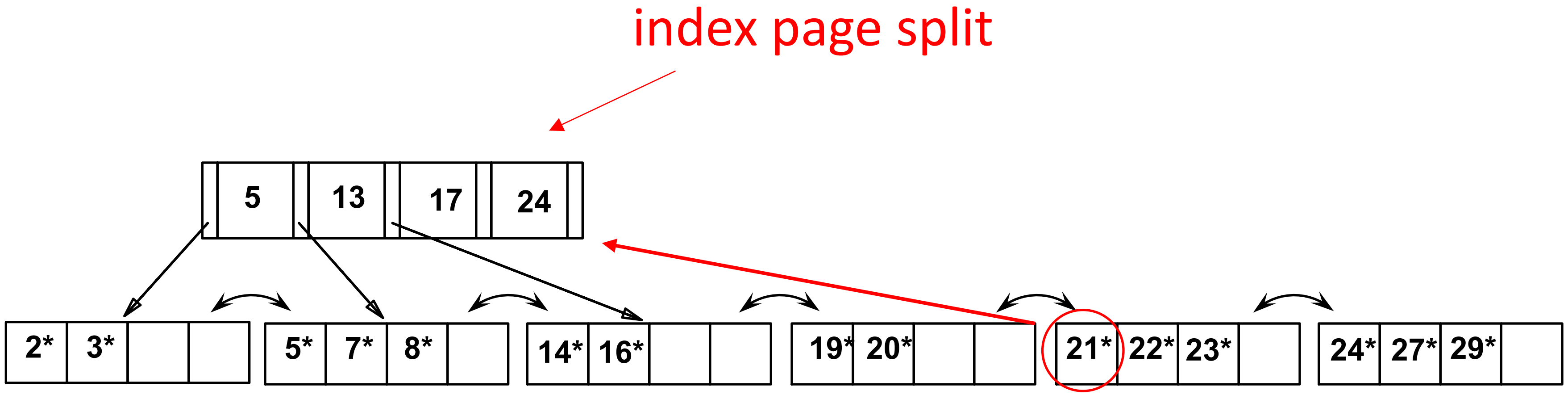


Example B+ Tree - Inserting 21*

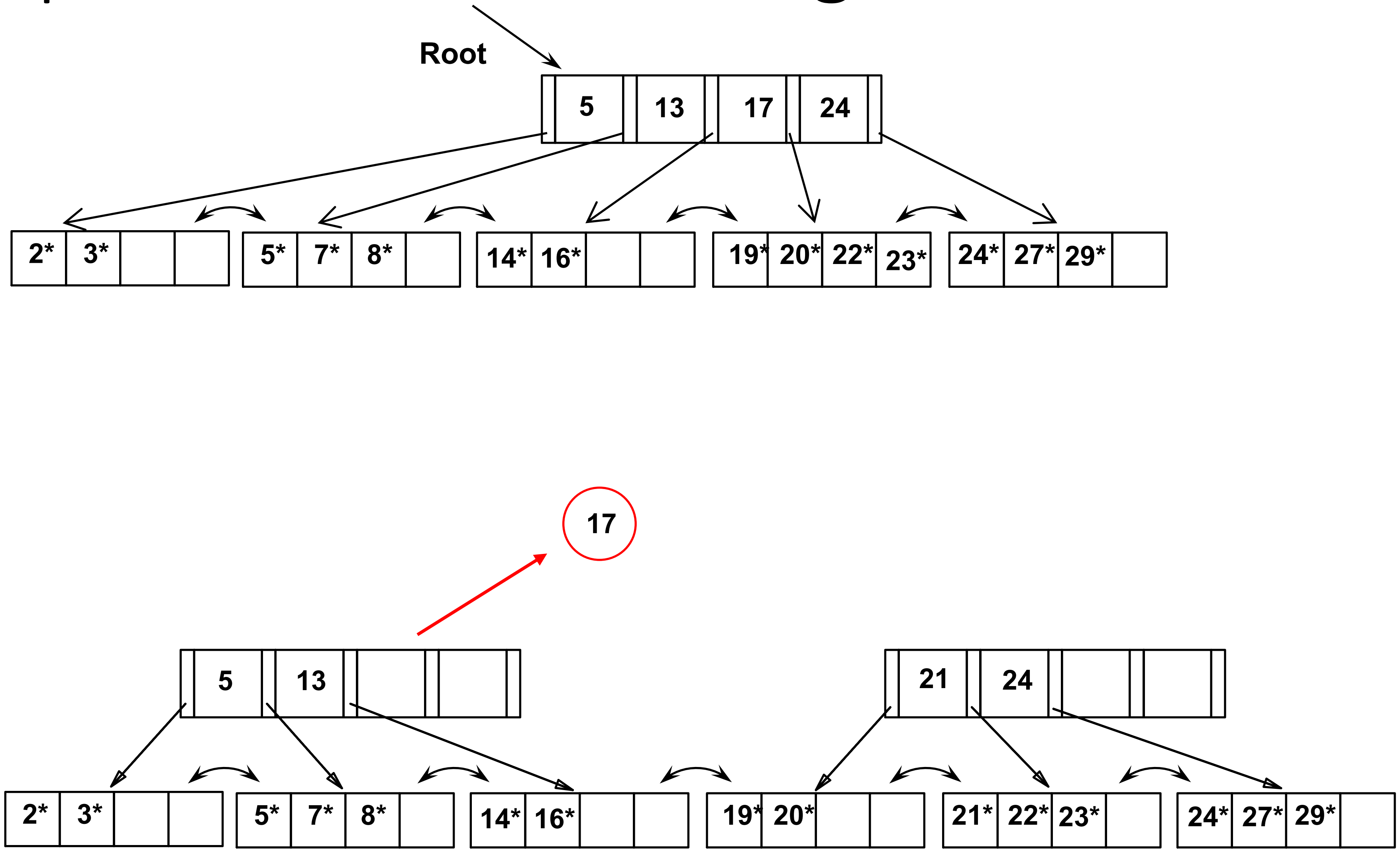


Example B+ Tree - Inserting 21*

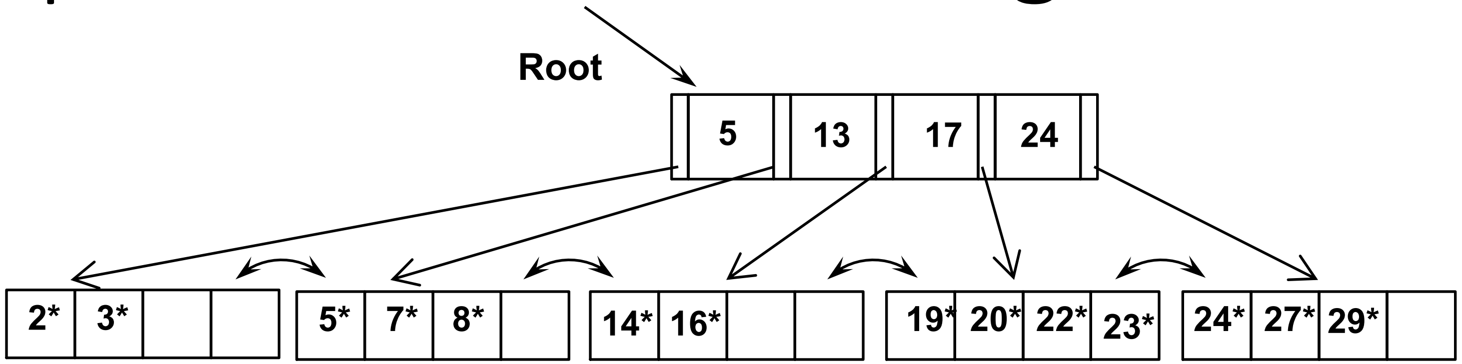




Example B+ Tree - Inserting 21*



Example B+ Tree - Inserting 21*

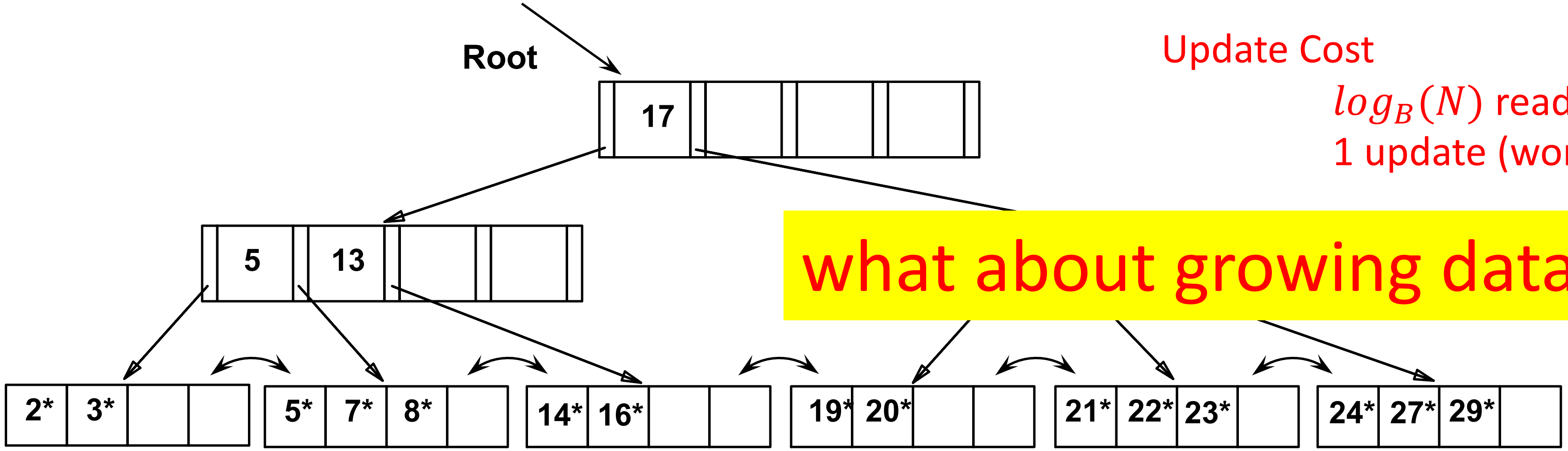


Read Cost: $log_B(N)$

Update Cost

$log_B(N)$ reads

1 update (worse case $log_B(N)$)



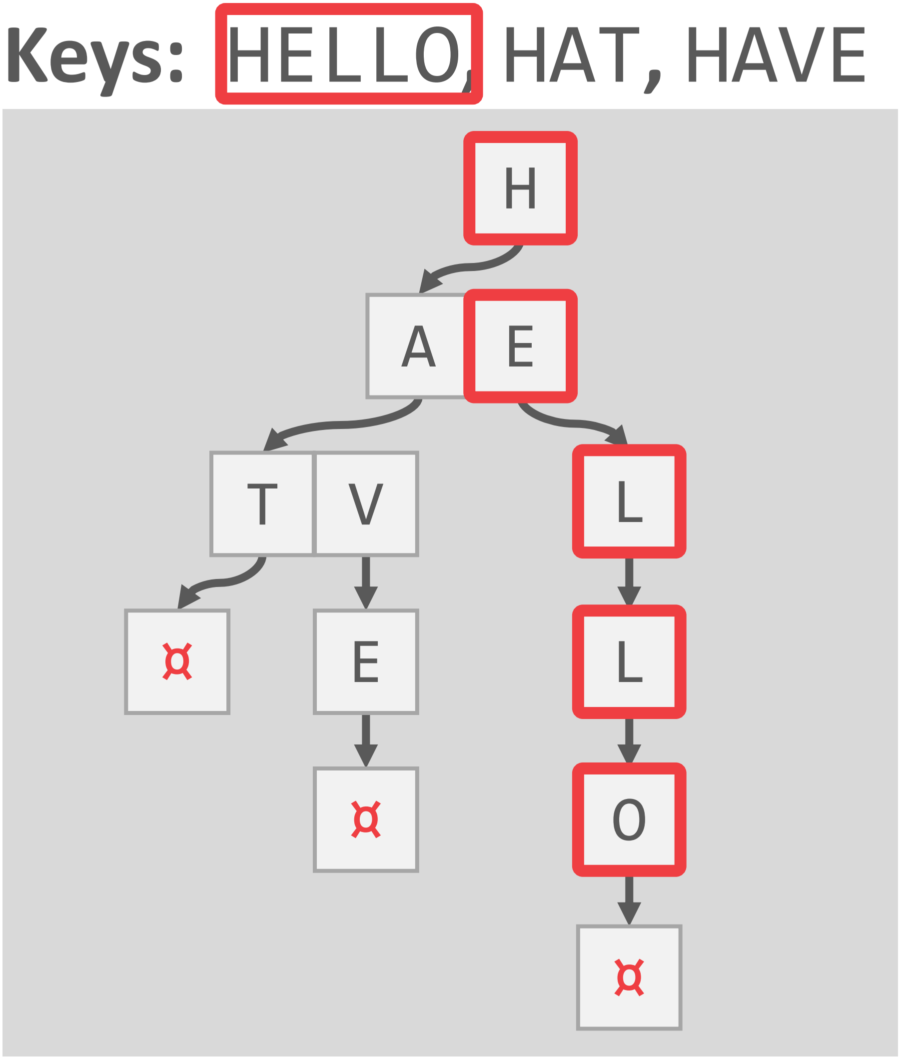
what about growing dataset size?

Observation

- The inner node keys in a B+tree cannot tell you whether a key exists in the index. You always must traverse to the leaf node.

- This means that you could have (at least) one cache miss per level in the tree.

Trie index



- Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
  - Also known as ***Digital Search Tree***, ***Prefix Tree***.

Trie index properties

- Shape only depends on key space and lengths.
  - Does not depend on existing keys or insertion order.
  - Does not require rebalancing operations.

- All operations have **O(*k*)** complexity where ***k*** is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are stored implicitly and can be reconstructed from paths.

Trie index properties

- Shape only depends on key space and lengths.
  - Does not depend on existing keys or insertion order. ← **History independent**
  - Does not require rebalancing operations.

- All operations have **O(*k*)** complexity where ***k*** is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are stored implicitly and can be reconstructed from paths.

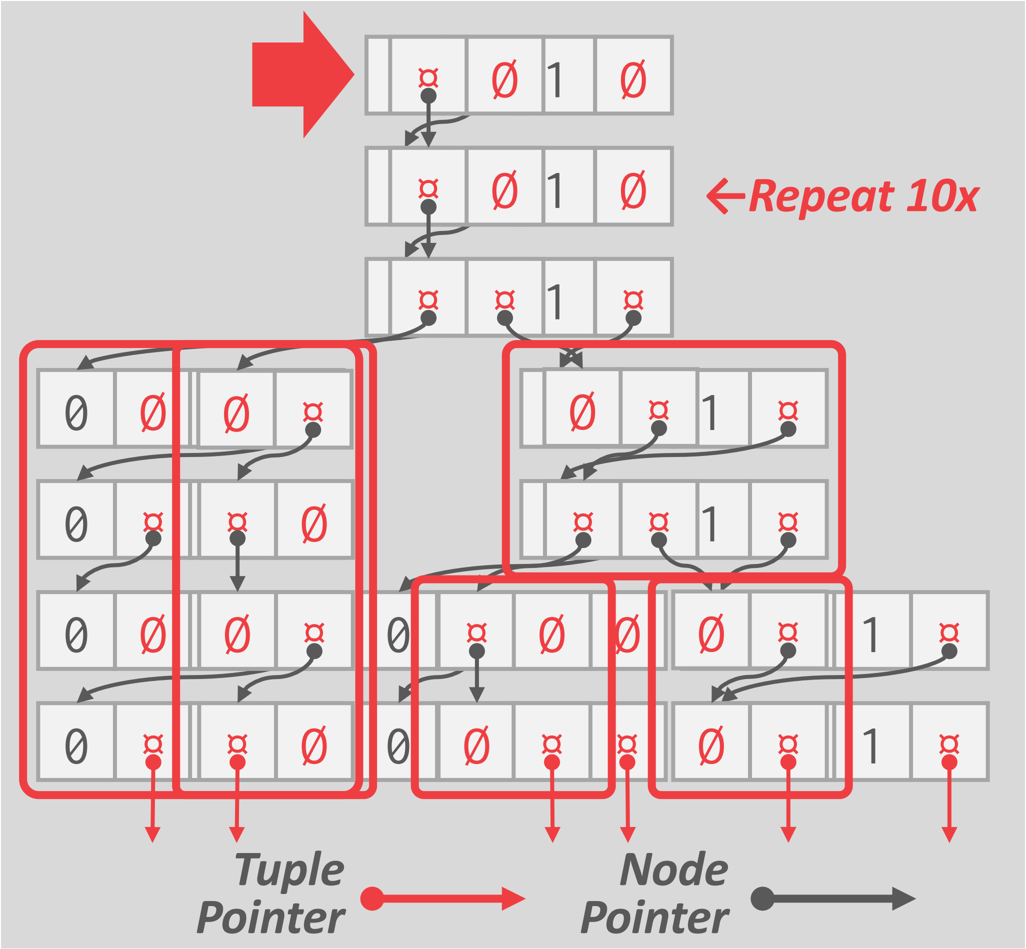
Trie key span

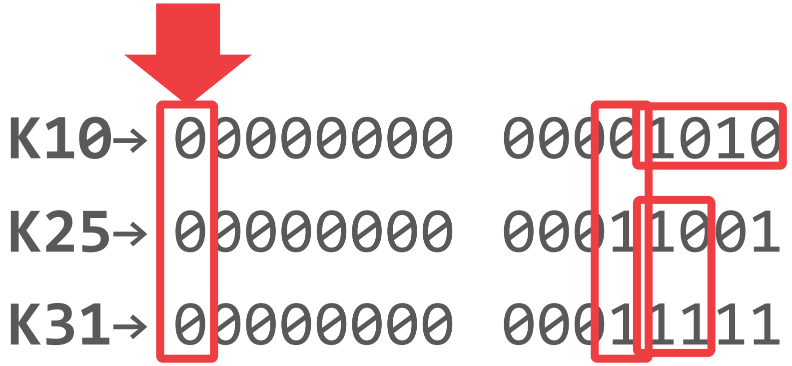
- The **span** of a trie level is the number of bits that each partial key / digit represents.
  - If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

- This determines the **fan-out** of each node and the physical **height** of the tree.
  - ***n***-way Trie = Fan-Out of ***n***

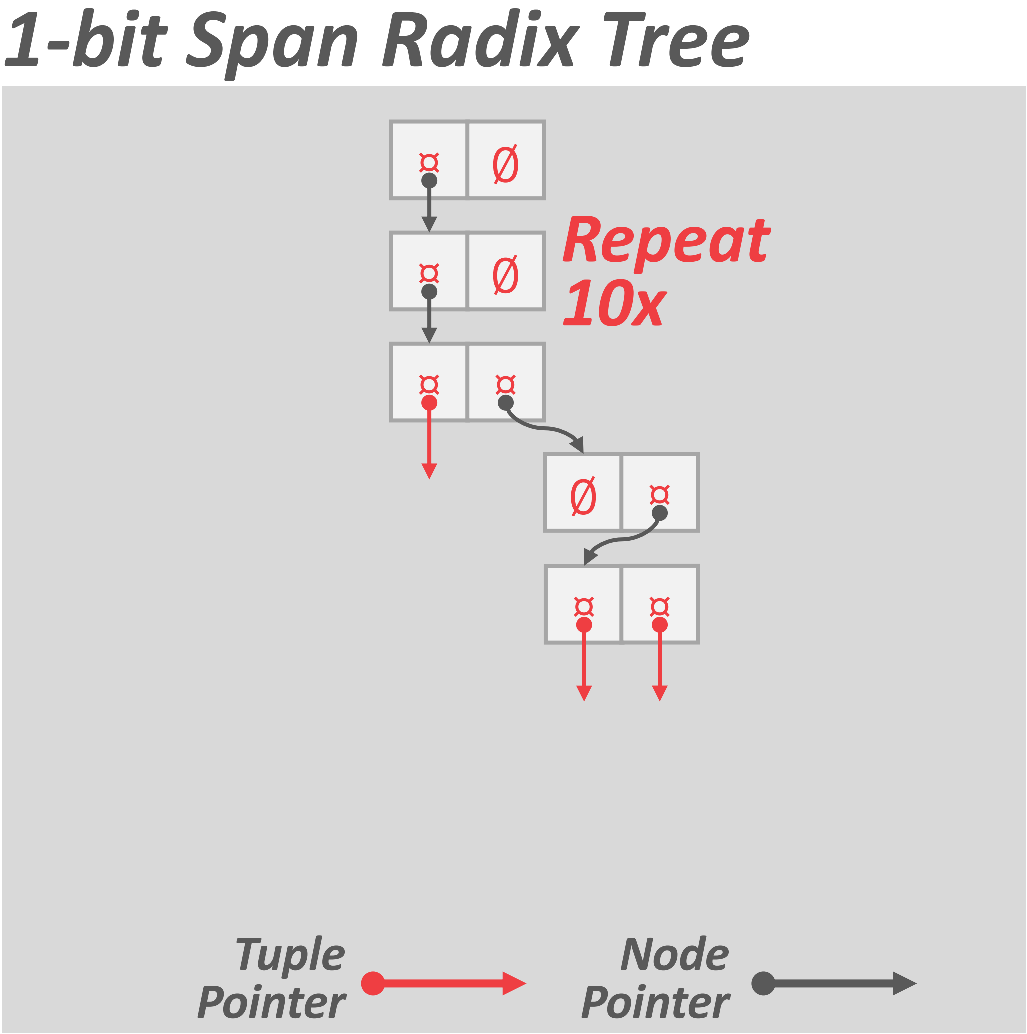
Trie key span

1-bit Span Trie





Radix tree



- Omit all nodes with only a single child.
  - Also known as ***Patricia Tree***.
- Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

Trie variants

- Judy Arrays (HP)
- ART Index (HyPer)
- Masstree (Silo)

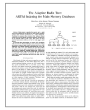
Judy arrays

- Variant of a 256-way radix tree. First known radix tree that supports adaptive node representation.
- Three array types
  - **Judy1**: Bit array that maps integer keys to true/false.
  - **JudyL**: Map integer keys to integer values.
  - **JudySL**: Map variable-length keys to integer values.

- Open-Source Implementation (LGPL). Patented by HP in 2000. Expires in 2022.
  - Not an issue according to authors.
  - http://judy.sourceforge.net/

Adaptive radix tree (ART)

- Developed for TUM HyPer DBMS in 2013.
- 256-way radix tree that supports different node types based on its population.
  - Stores meta-data about each node in its header.
- Concurrency support was added in 2015.

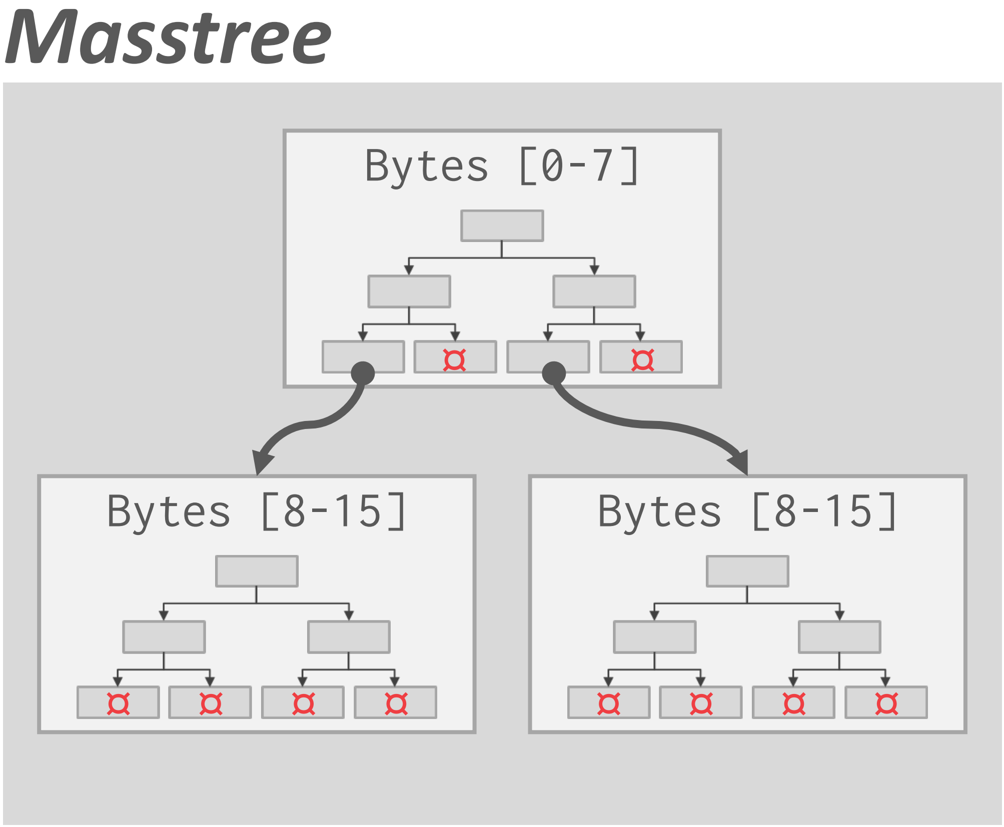


THE ADAPTIVE RADIX TREE: ARTFUL INDEXING FOR MAIN-MEMORY DATABASES
ICDE 2013

ART vs. JUDY

- **Difference #1: Node Types**
  - Judy has three node types with different organizations.
  - ART has four nodes types that (mostly) vary in the maximum number of children.
- **Difference #2: Purpose**
  - Judy is a general-purpose associative array. It "owns" the keys and values.
  - ART is a table index and does not need to cover the full keys. Values are pointers to tuples.

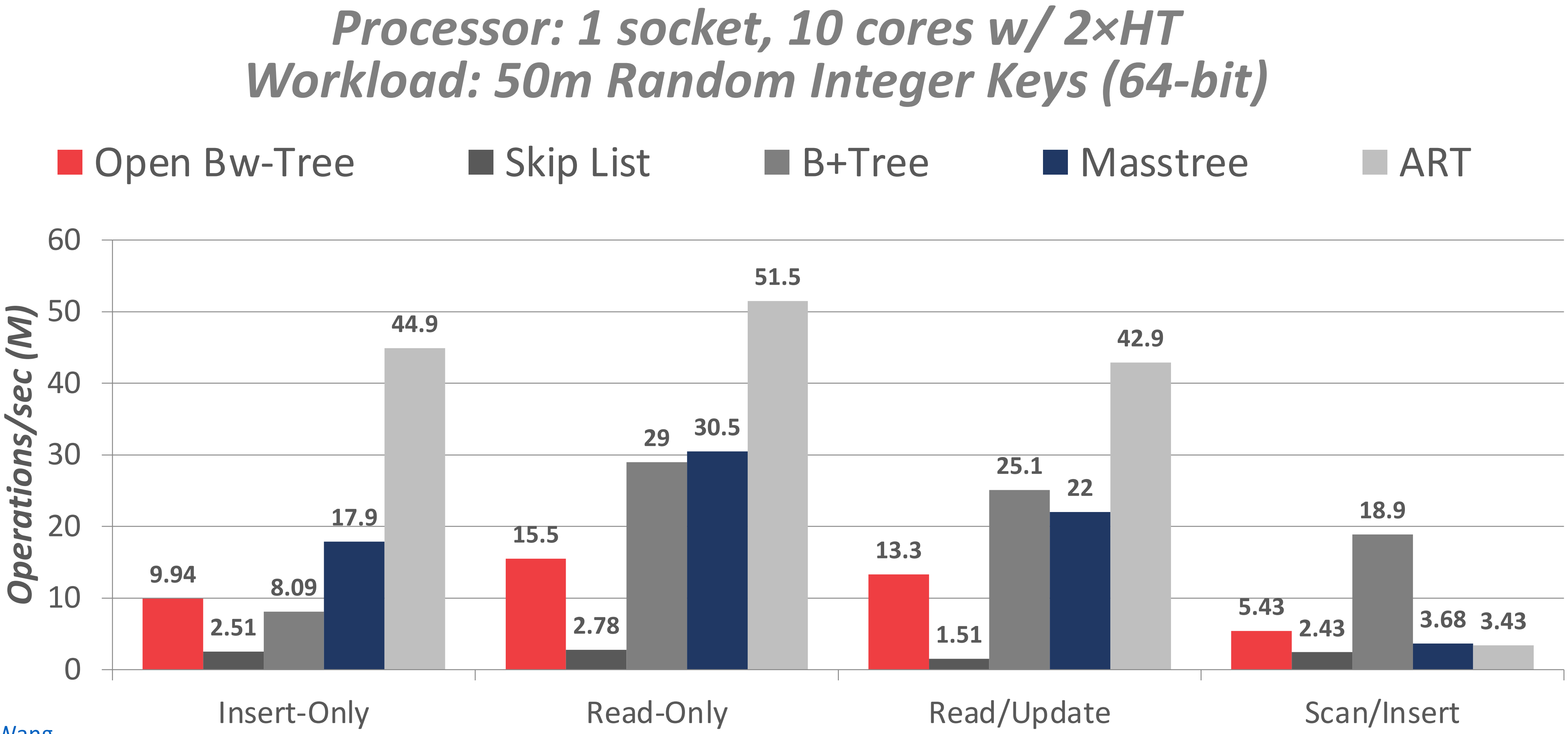
MASSTREE



- Instead of using different layouts for each trie node based on its size, use an entire B+Tree.
  - Each B+tree represents 8-byte span.
  - Optimized for long keys.
  - Uses a latching protocol that is similar to versioned latches.
- Part of the Harvard Silo project.

CACHE CRAFTINESS FOR FAST MULTICORE KEY-VALUE STORAGE
EUROSYS 2012

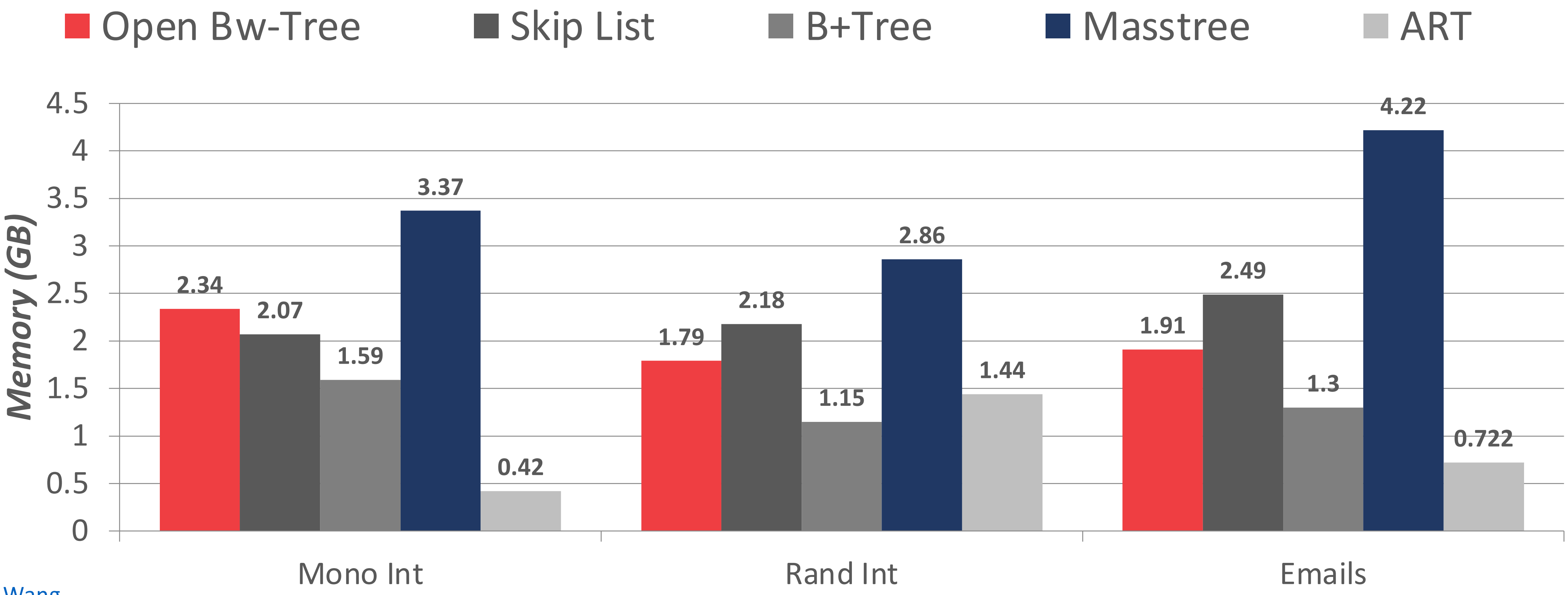
IN-MEMORY INDEXES



Source: [Ziqi Wang]

IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Keys



Source: [Ziqi Wang]()

PARTING THOUGHTS

- B+ trees are the go to in-memory indexing data structures.
- Radix trees have interesting properties, but a well-written B+tree is still a solid design choice.
- Skip lists are amazing if you don’t want to implement self balancing binary trees

Next class

- Concurrency control

Make sure to read the related papers from the reading list