

CS 7270: Advanced Database Systems Fall 2025

Lecture 04

In-memory indexing (Trees, Skip Lists)

Prashant Pandey

p.pandey@northeastern.edu

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

In-memory indexing

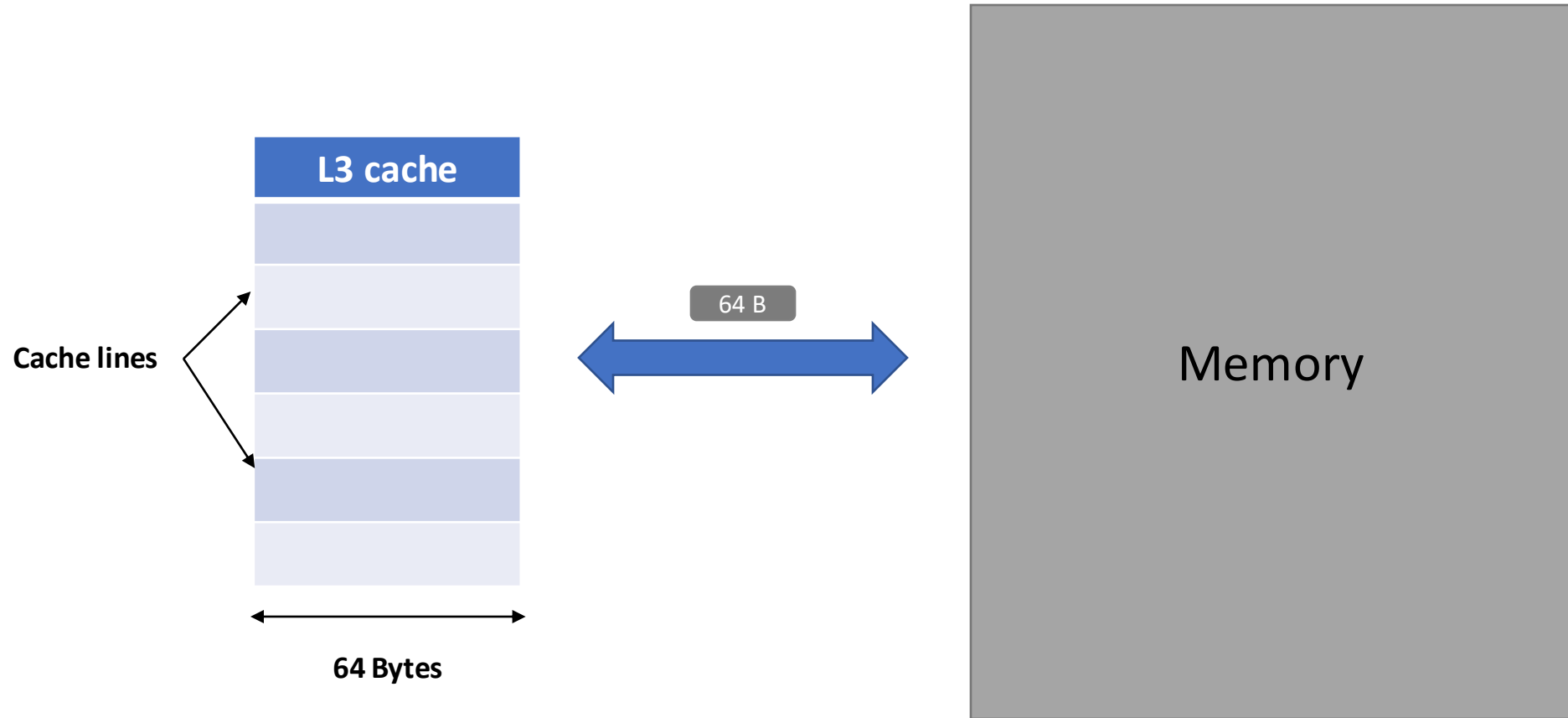
Bandwidth	Latency	Size	
1 TB/sec	1-4 n sec	256 KB	CPU
~400 GB/sec	~30 n sec	8 - 12 MB	on-chip cache
~50 GB/sec	80-100 n sec	4 GB - 1 TB	on-board cache
			main memory

In-memory indexing

Bandwidth	Latency	Size	
1 TB/sec	1-4 n sec	256 KB	CPU
			on-chip cache
~400 GB/sec	~30 n sec	8 - 12 MB	on-board cache
			↕
~50 GB/sec	80-100 n sec	4 GB - 1 TB	main memory

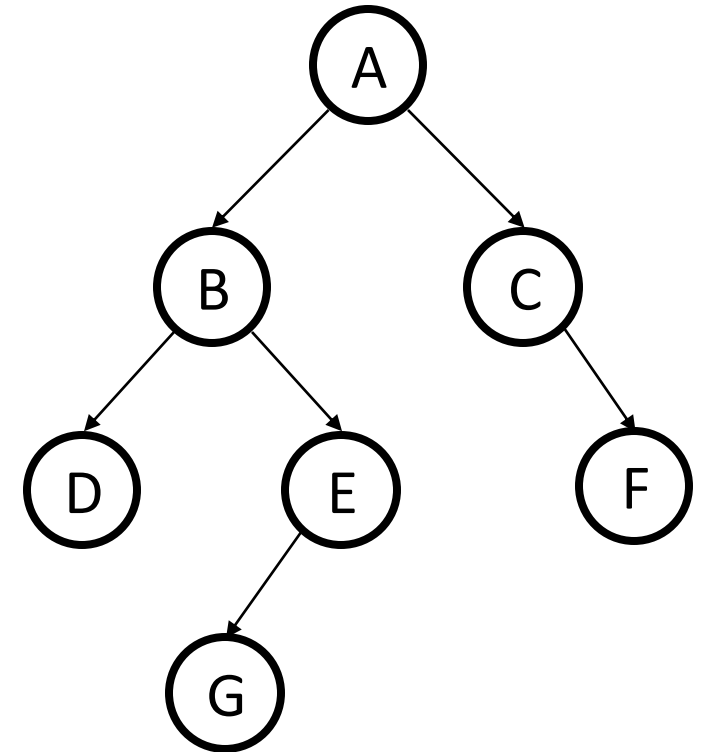
Data transfer between L3 and DRAM is the dominating factor.

Data movement b/w L3 cache and memory



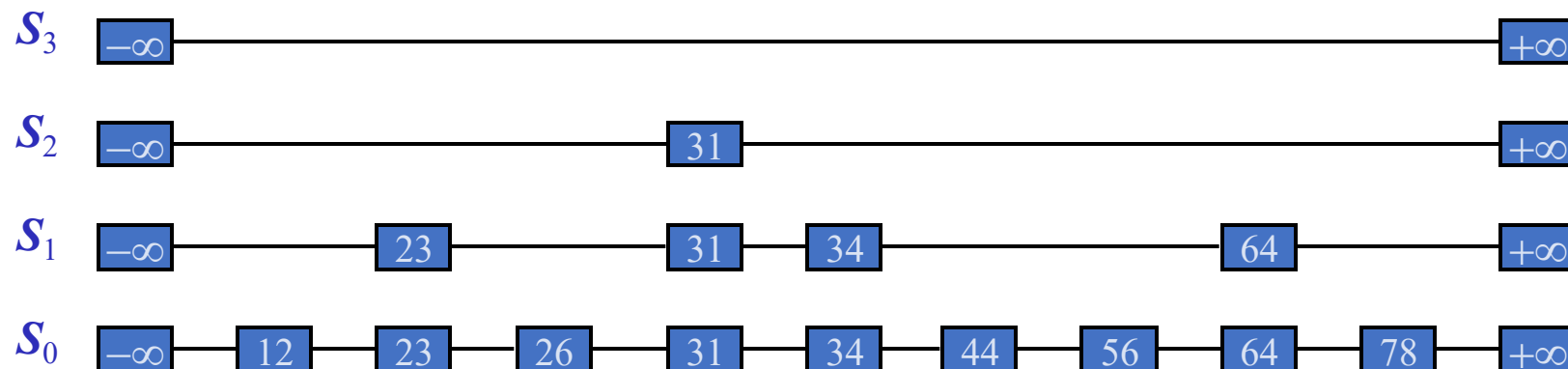
Balanced binary search tree

- **Binary search tree**: Each node has at most 2 children (branching factor 2)
- Binary tree is
 - A root (*with data*)
 - A left subtree (*may be empty*)
 - A right subtree (*may be empty*)
- Search/Insert/Delete $O(\log(N))$



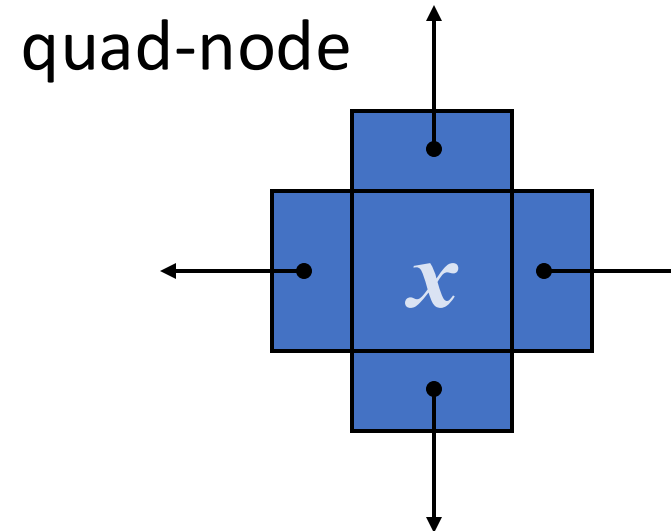
What is a Skip List

- A **skip list** for a set \mathcal{S} of distinct (key, element) items is a series of lists $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_h$ such that
 - Each list \mathcal{S}_i contains the special keys $+\infty$ and $-\infty$
 - List \mathcal{S}_0 contains the keys of \mathcal{S} in non-decreasing order
 - Each list is a subsequence of the previous one, i.e.,
$$\mathcal{S}_0 \supseteq \mathcal{S}_1 \supseteq \dots \supseteq \mathcal{S}_h$$
 - List \mathcal{S}_h contains only the two special keys
- Skip lists are one way to implement the dictionary



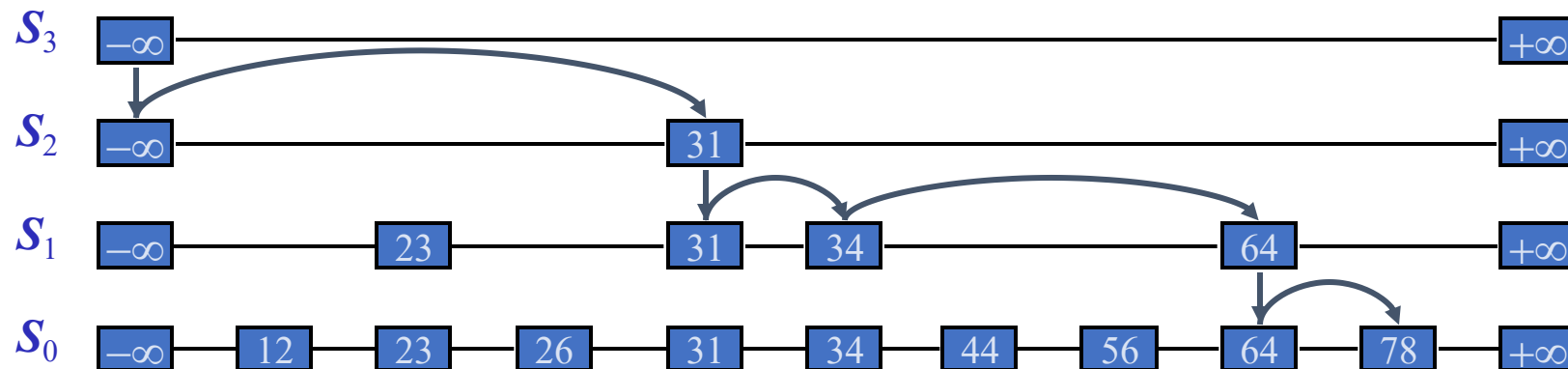
Implementation

- 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



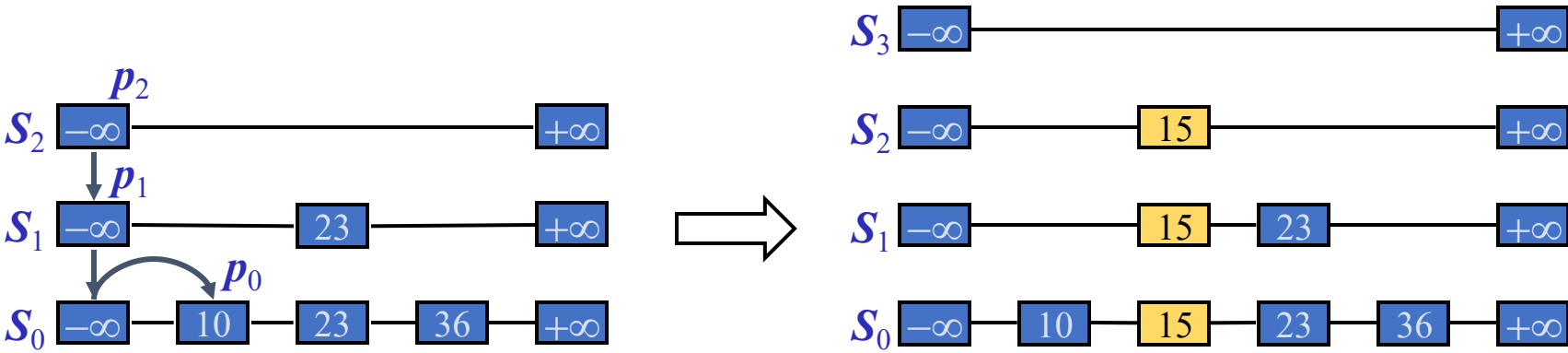
Search

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



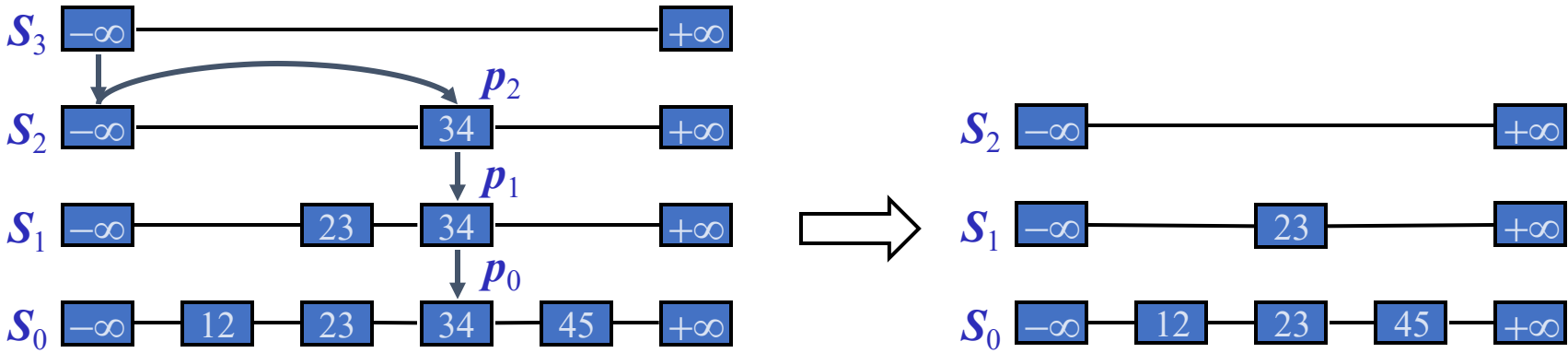
Insertion

- 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}, \dots, 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, \dots, p_i of the items with largest key less than x in each list S_0, S_1, \dots, S_i
 - For $j \leftarrow 0, \dots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



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, \dots, p_i of the items with key x , where position p_j is in list S_j
 - We remove positions p_0, p_1, \dots, p_i from the lists S_0, S_1, \dots, S_i
 - We remove all but one list containing only the two special keys
- Example: remove key 34



Randomized Algorithms

- A **randomized algorithm** controls its execution through random selection (e.g., coin tosses)
- It contains statements like:
$$b \leftarrow \text{randomBit}()$$
$$\text{if } b = 0$$
$$\quad \text{do A ...}$$
$$\text{else } \{ b = 1 \}$$
$$\quad \text{do B ...}$$
- 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**

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 i consecutive heads when flipping a coin is $1/2^i$

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

- Consider a skip list with n items
 - By Fact 1, we insert an item in list \mathcal{S}_i with probability $1/2^i$
 - By Fact 2, the expected size of list \mathcal{S}_i is $n/2^i$
- 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 n items is $O(n)$

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 $1 - 1/n^2$

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)

Height

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

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.

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)

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

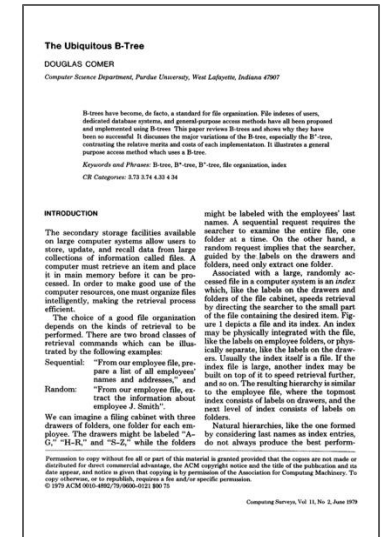
Question?



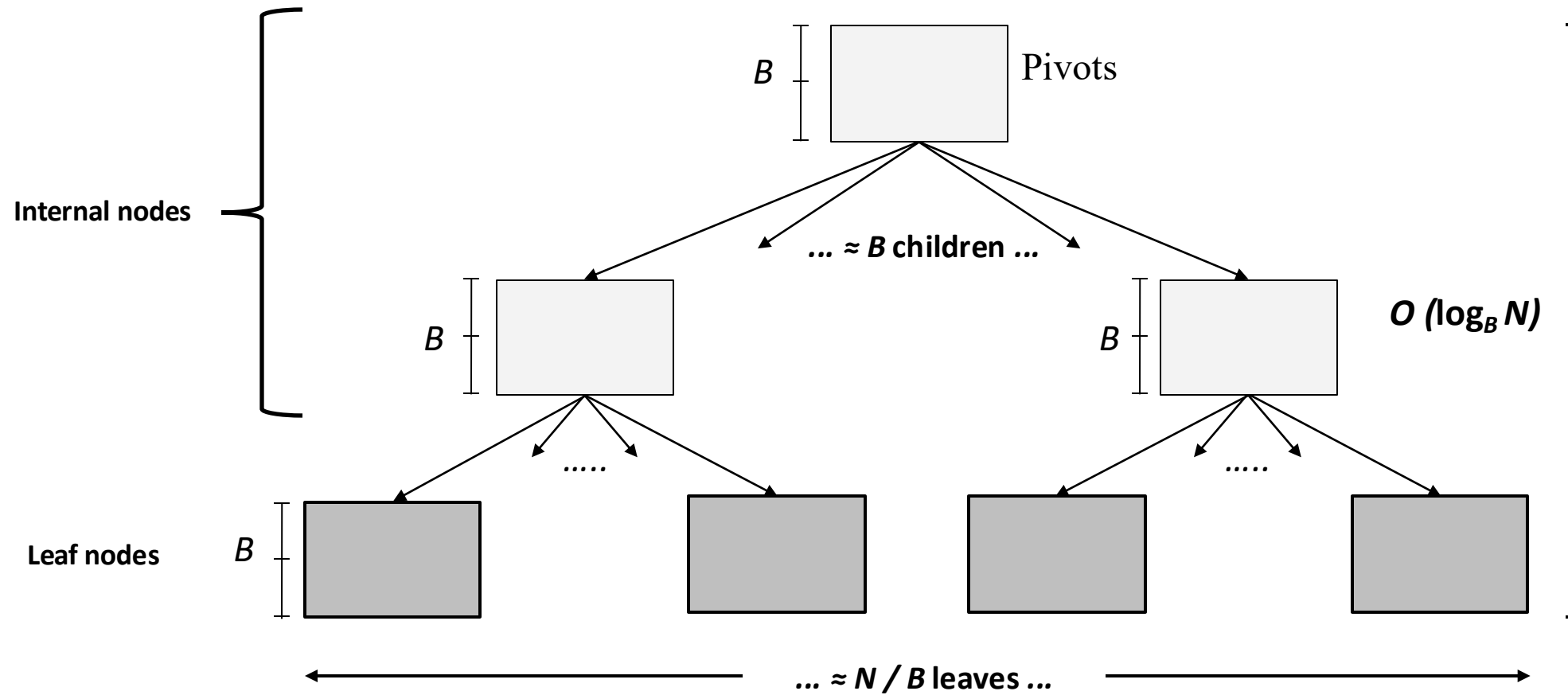
Are Binary trees and skip lists optimal for in-memory indexing?

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 $O(\log_B(N))$.
 - 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.



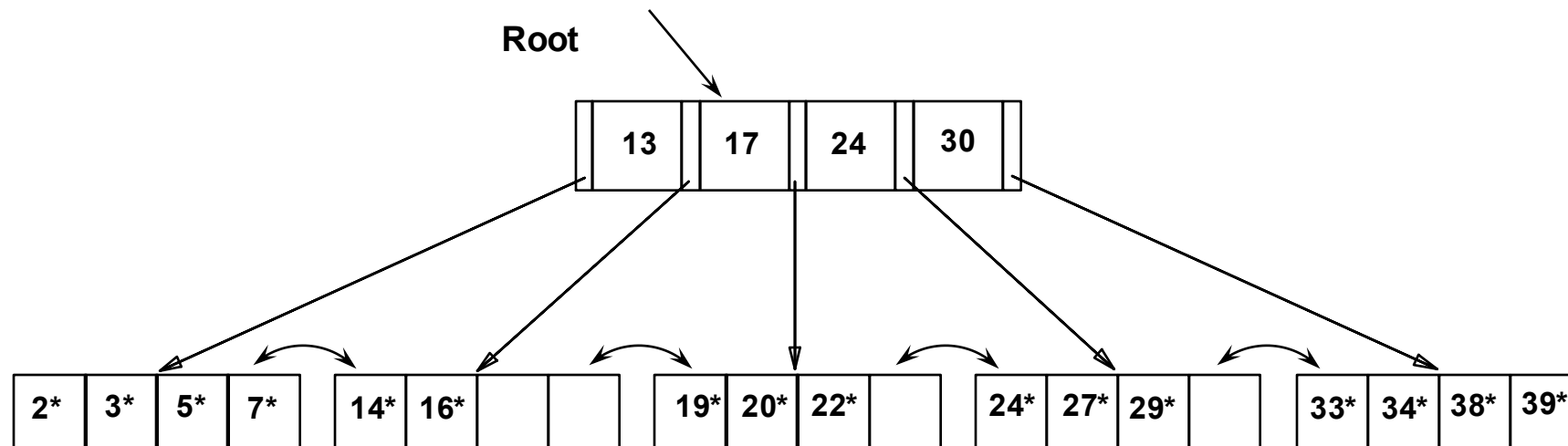
B+ Trees



B+ Trees

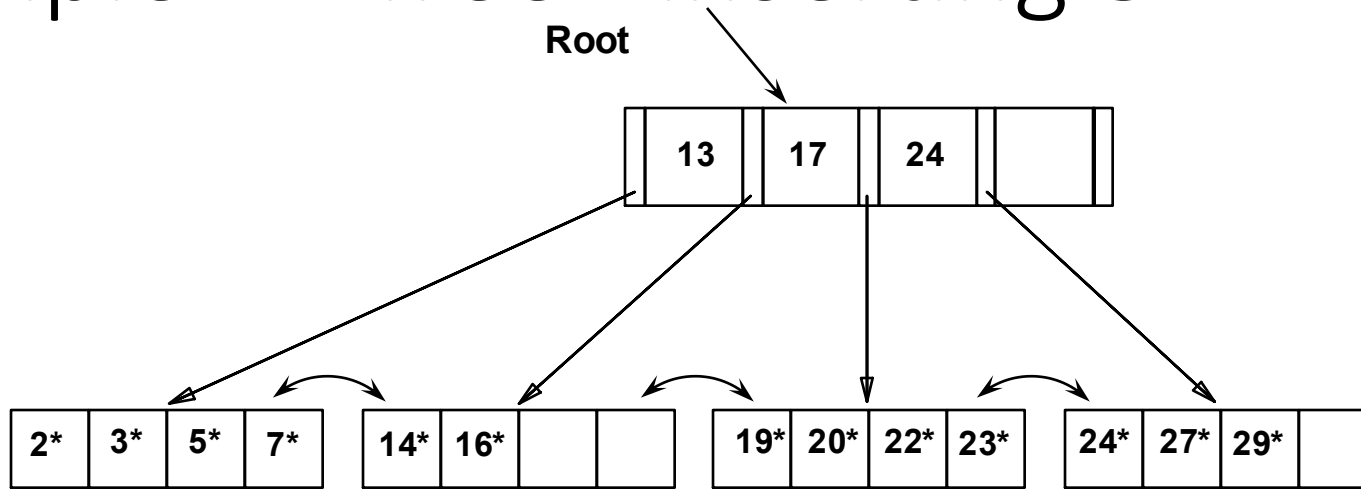
Search begins at root, and key comparisons direct it to a leaf.

Search for 5*, 15*, all data entries $\geq 24^*$...

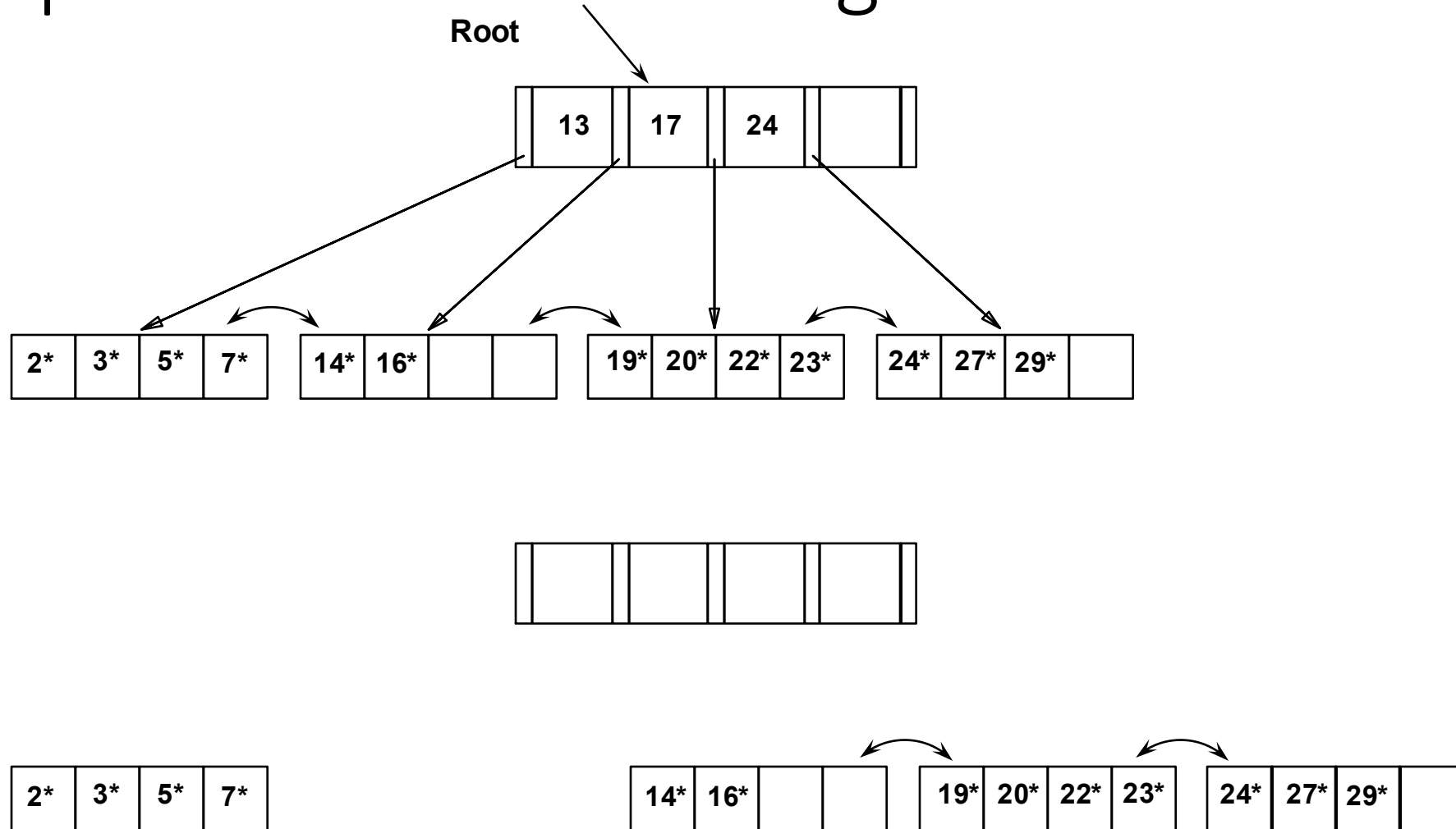


Based on the search for 15, we know it is not in the tree!*

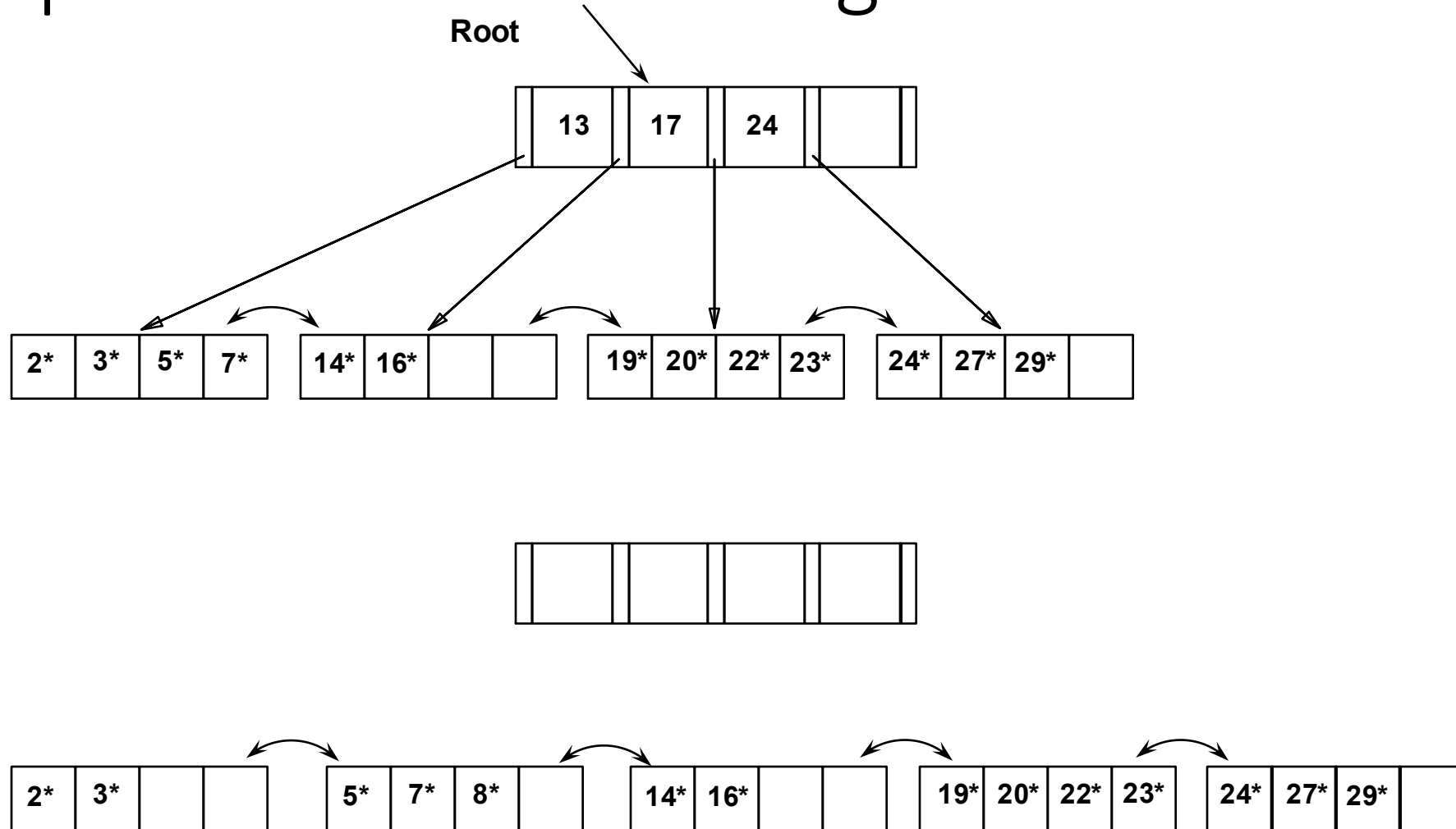
Example B+ Tree - Inserting 8*



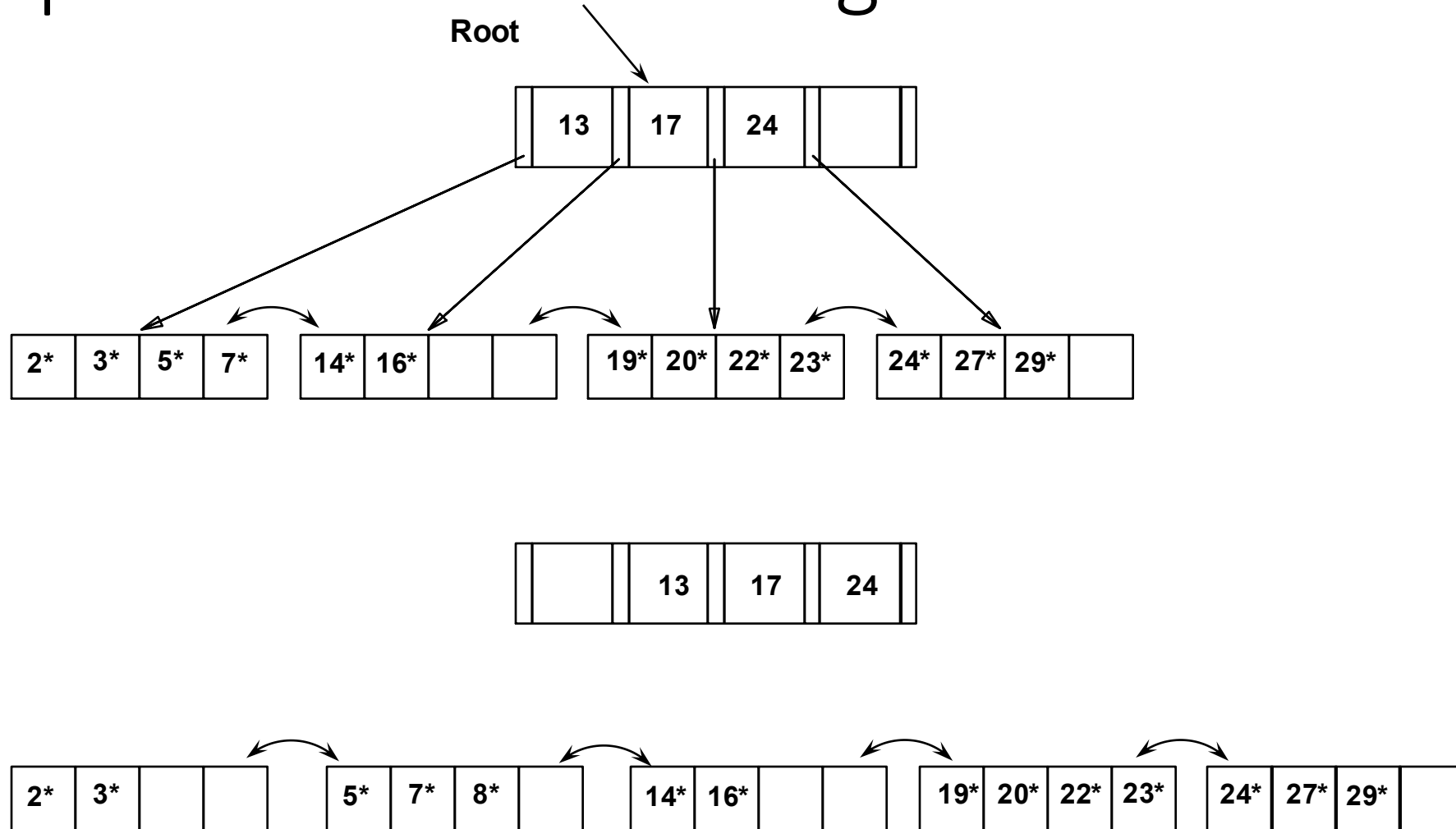
Example B+ Tree - Inserting 8*



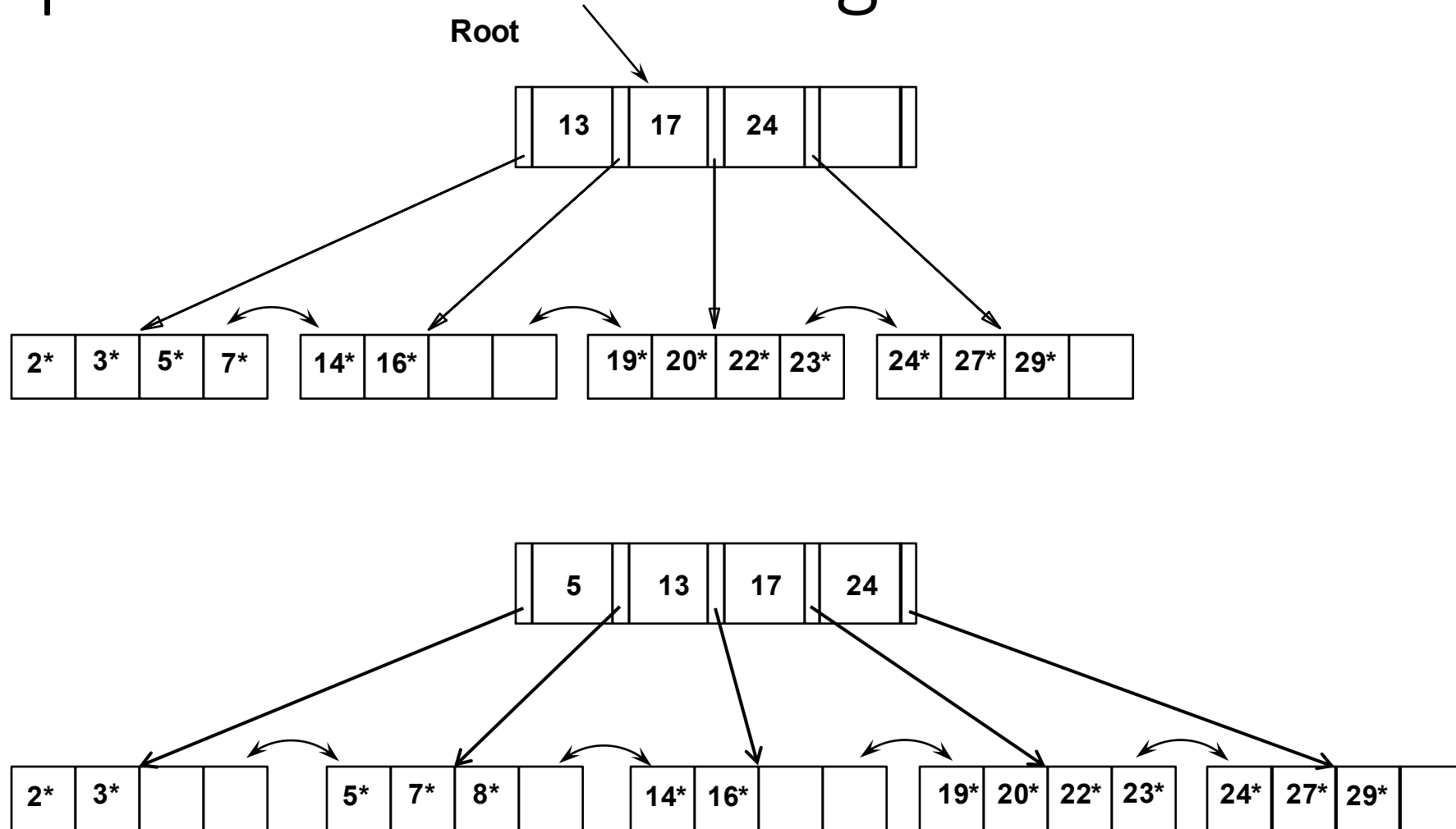
Example B+ Tree - Inserting 8*



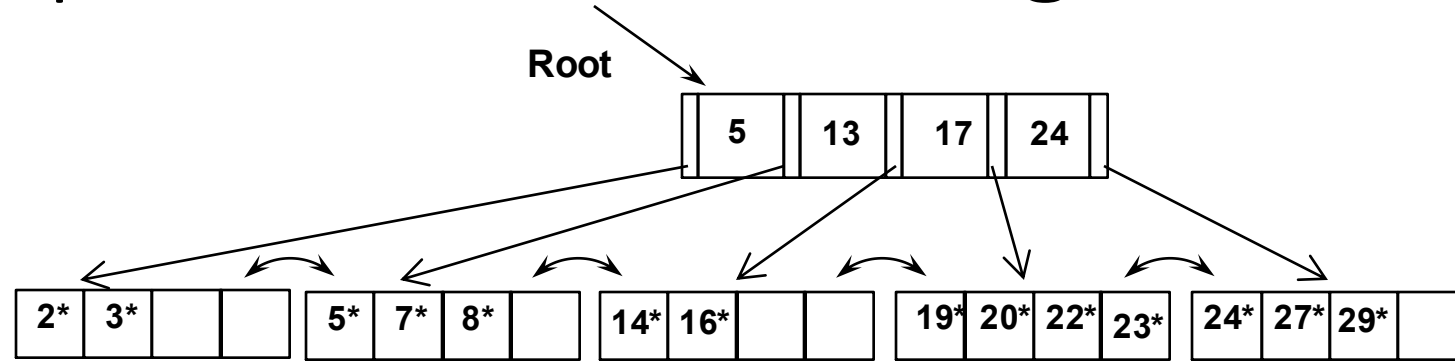
Example B+ Tree - Inserting 8*



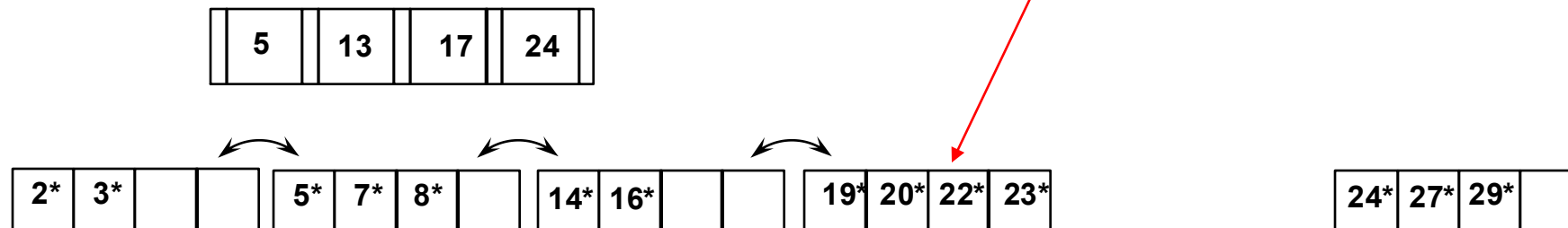
Example B+ Tree - Inserting 8*



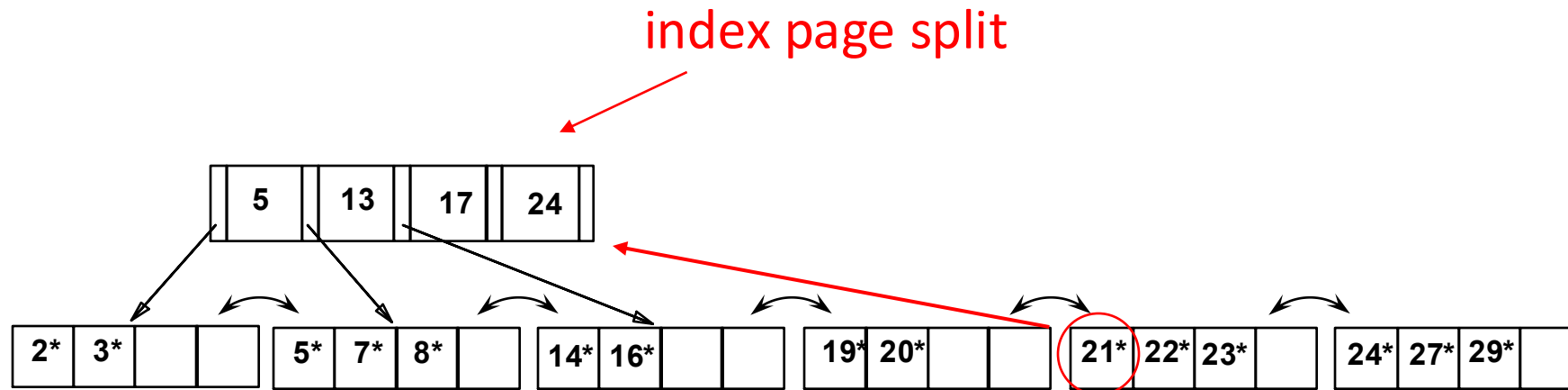
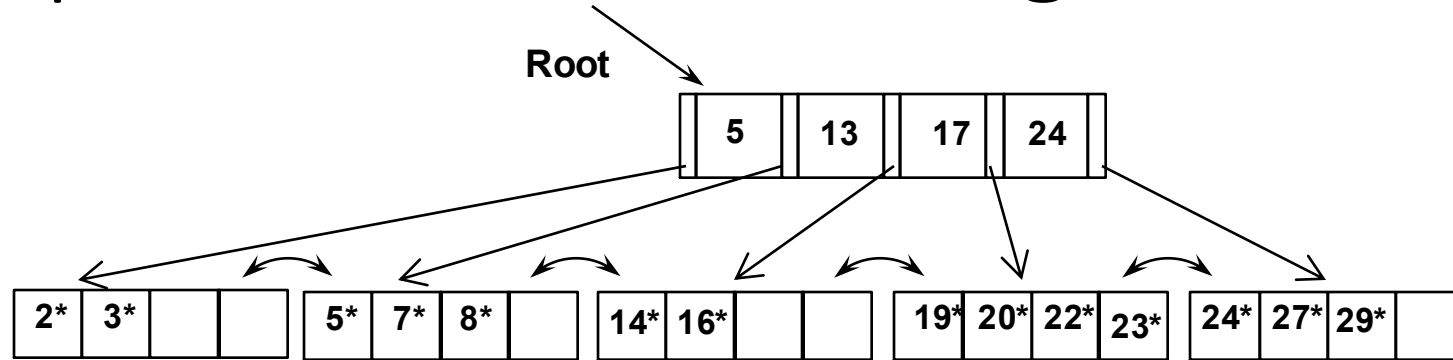
Example B+ Tree - Inserting 21*



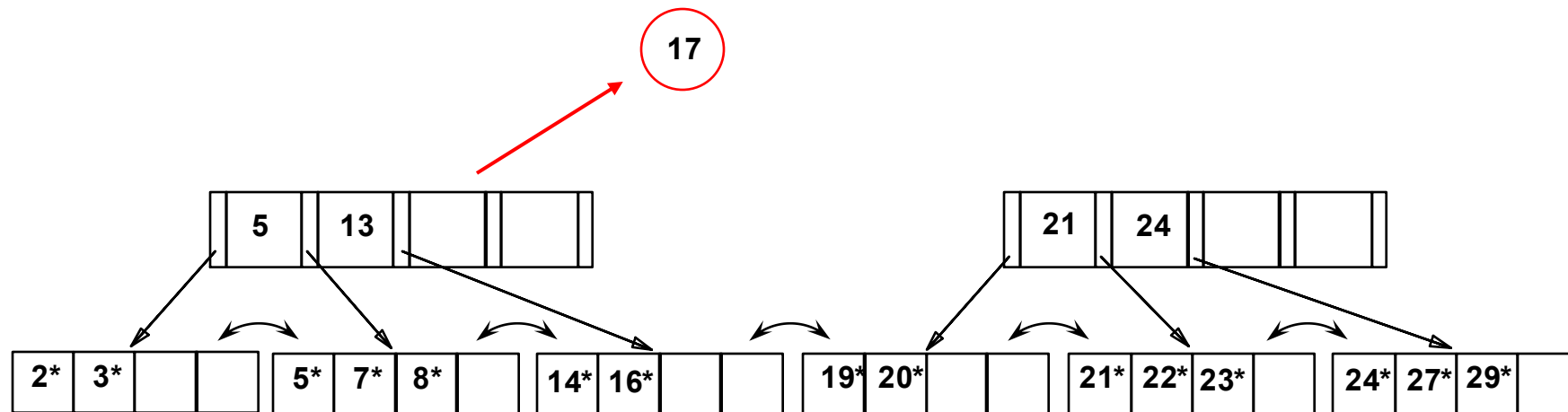
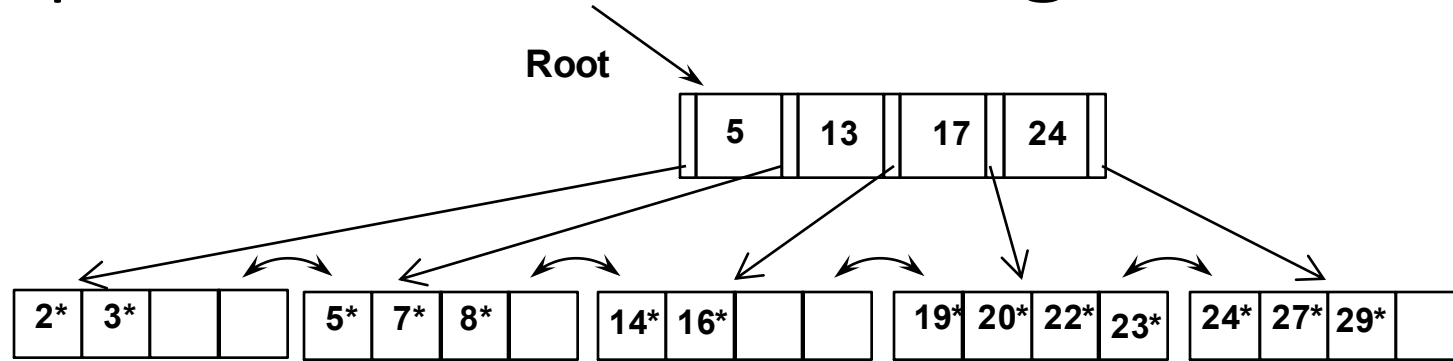
data page split



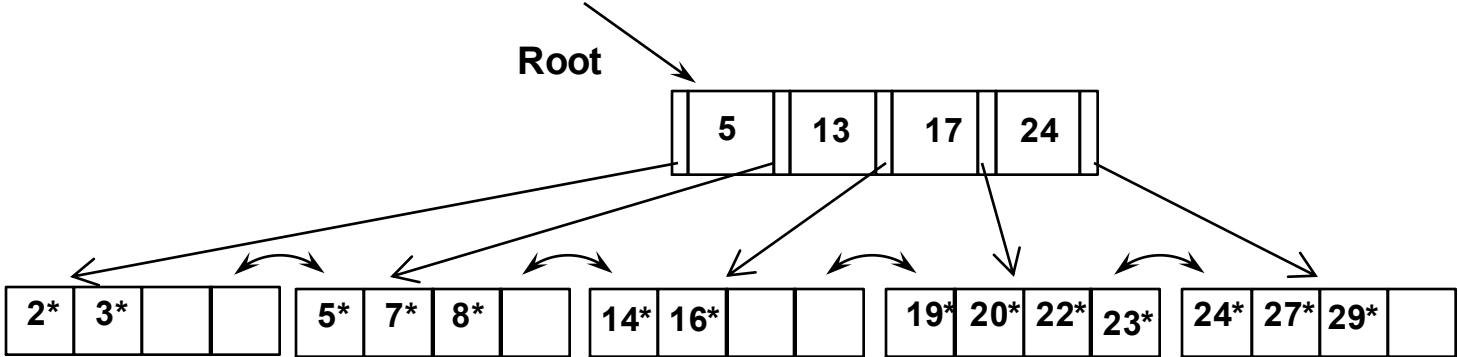
Example B+ Tree - Inserting 21*



Example B+ Tree - Inserting 21*

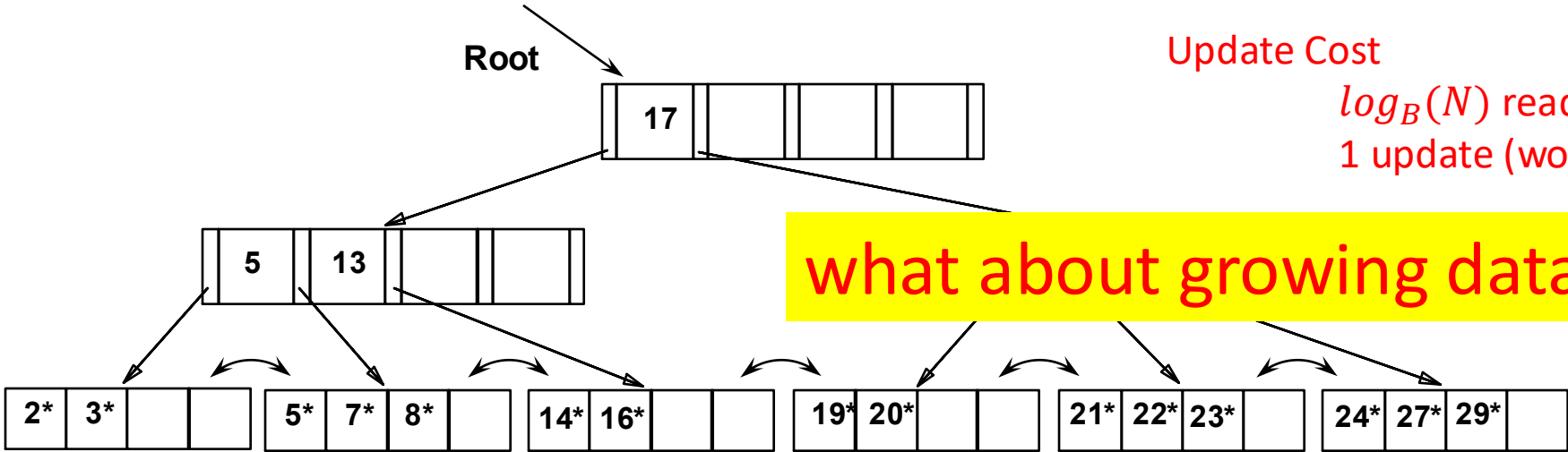


Example B+ Tree - Inserting 21*



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.

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.

How to size the B+-tree nodes?