B-trees (Ubiquitous and otherwise)

Williams College :: CSCI 333
Spring 2019

Logistics

Deadlines

- Lab 2b
- Final Project

Project details

- Can choose partners
  - Status quo is to remain with your FUSE FAT teammates
- Must submit proposal
  - Must meet in person to discuss
- Final project includes a workshop-style write-up

Last Class

Hashing and Filters

- Bloom filters
- Cuckoo filters
- Quotient filters

Support "approximate membership" queries

- No false negatives
- Tunable false positive rate

Cache efficiency matters

- Quotient > Cuckoo > Bloom

API matters

- Deletes? Merges? Resizing?

This Class

DAM model

- How to analyze external memory algorithms

B-trees

- Operations
- Variants
- Discussion

How do you keep data organized?

An Analogy from [Comer 79 CSUR]

Filing Cabinet: folders of records, alpha-sorted by last name

- We think in terms of keys and values
  - Keys are the employee's last name
  - Values are the employee file (held in a folder, one per employee)

- A filing cabinet supports two types of searches
  - Sequential
    - read through every folder in every drawer in order
  - Random
    - use the labels on the drawers & folders to find the single record of interest

Indexes (yes, colloquially pluralized that way)

Indexes organize data

- Random searches utilize an ***index*** to:
  - Direct our search towards a small part of the total data
  - (Hopefully) speed up our search

Questions

- What operations does an index support?
- How do we quantify index performance?
- Is the data part of the index, or does the index "sit on top of" the data?

What operations does an index support?

Operations

- Insert(k,v): inserts key-value pair (k,v)
- Delete(k): deletes any pair (k,*)
- PointQuery(k): returns all pairs (k,*)
- RangeQuery(k_1,k_2): returns all pairs (k,*), $k_1 \leq k \leq k_2$

In short, indexes support the *dictionary* interface.

- Used when data is too big for memory.

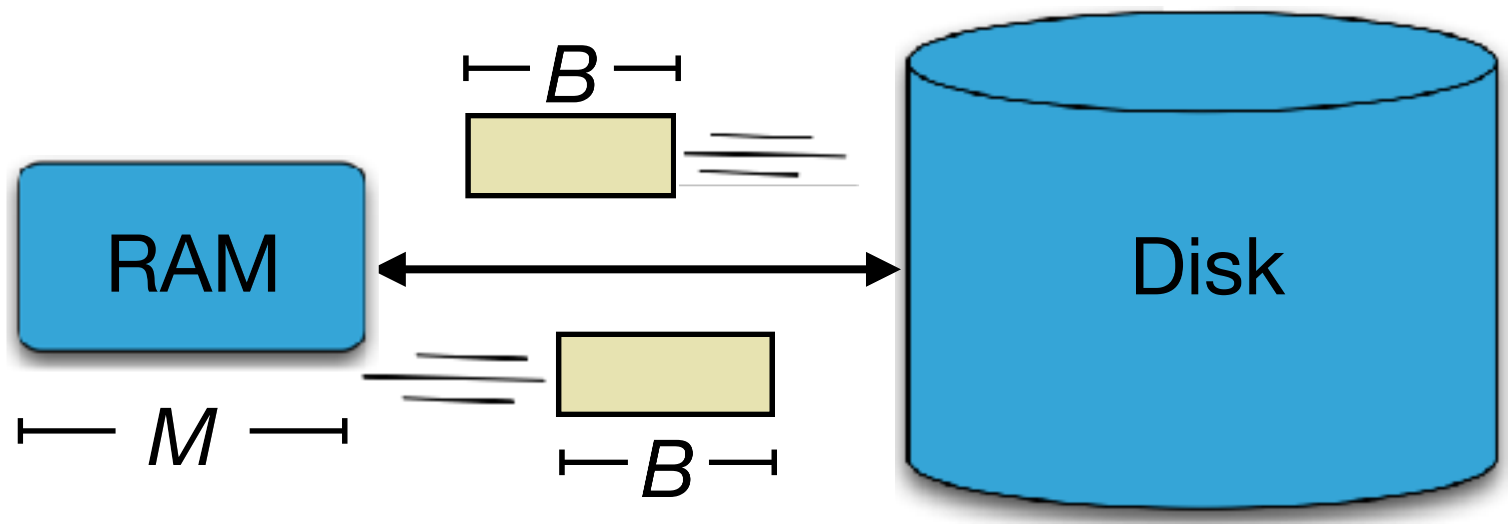
How to we quantify index performance?

DAM model:

- Useful when data is too big for memory
  - Data is transferred in **blocks** between RAM and disk.
- The number of block transfers dominates the running time.
  - Searching through a given block is “free” (once in-memory)

Goal: Minimize # of I/Os

- Performance bounds are parameterized by block size ***B***, memory size ***M***, data size ***N***.



[Aggarwal+Vitter ’88]

DAM Model an B-tree Analysis

Analyze worst-case costs by counting I/Os

- **B**: unit of transfer
  - B-tree node size
- **M**: amount of main memory
  - We can cache M/B nodes in memory at once
- **N**: size of our data
  - We're not worried about disk space, we use **N** to describe our tree

- We will think about the tree shape (height, fanout), then describe each operation's cost in terms of the DAM model

The B-tree

Terms and Conditions

B-trees store records

- Records are key-value pairs
- We assume that keys are
  - Unique (to simplify analysis)
  - Ordered

Terms and Conditions

Rules for our B-trees

- B-ary tree
  - Internal nodes have between d and 2d keys called pivots
    - Must be half full!
  - At least d+1 pointers to children (one more pointer than pivot key)
- If an operation would cause a violation of one of these invariants, must rebalance!

- Note: our B-tree's internal nodes ***do not*** store records
  - Option 1: Store (key, value) pairs in leaves
  - Option 2: Store (key, pointer to value) in leaves

Terms and Conditions

Several B-tree variants

- We will describe a "$B^{?!+-}$-tree" here, noting features of specific variants as they come up

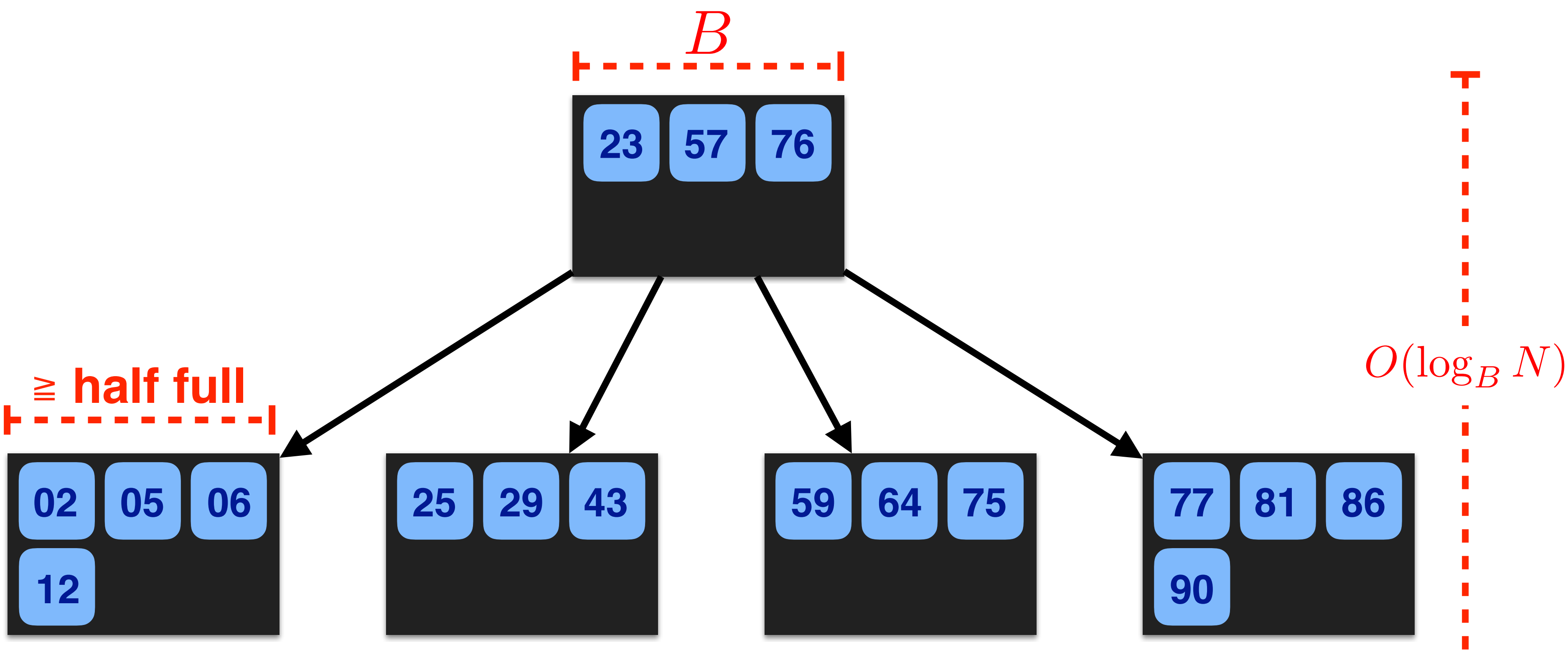
Popular Variants of B-trees

- B-tree: more-or-less what we'll describe here
- B^{+}-tree: B-tree where leaves form a linked list
- B^{*}-tree: B-tree where nodes always 2/3 full

B-tree: standard DAM dictionary

B-ary search tree

What does B Stand for?



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree Point Queries

B-tree Point Queries

Steps

- Starting at the root, find the first *pivot key* that is larger than your *search key*, and follow the pointer to its left
  - If there are no pivot keys larger than your search key, follow the last pointer
- Repeat until you arrive at a leaf node
- Search the leaf node (ordered list) for your target key
- Return the key-value pair (if found), or *NONE*

This work is done during an insert (need to find place where new key-value pair belongs), so we will walk through this then.

B-tree Point Queries

Cost

- How many nodes must be read/written in a search?
  - We read the root node to search the pivot keys
  - We recurse on the subtree
- Total cost of a search: O(h)
  - Recall **h = O($\log_B$N)**

B-tree Insertions

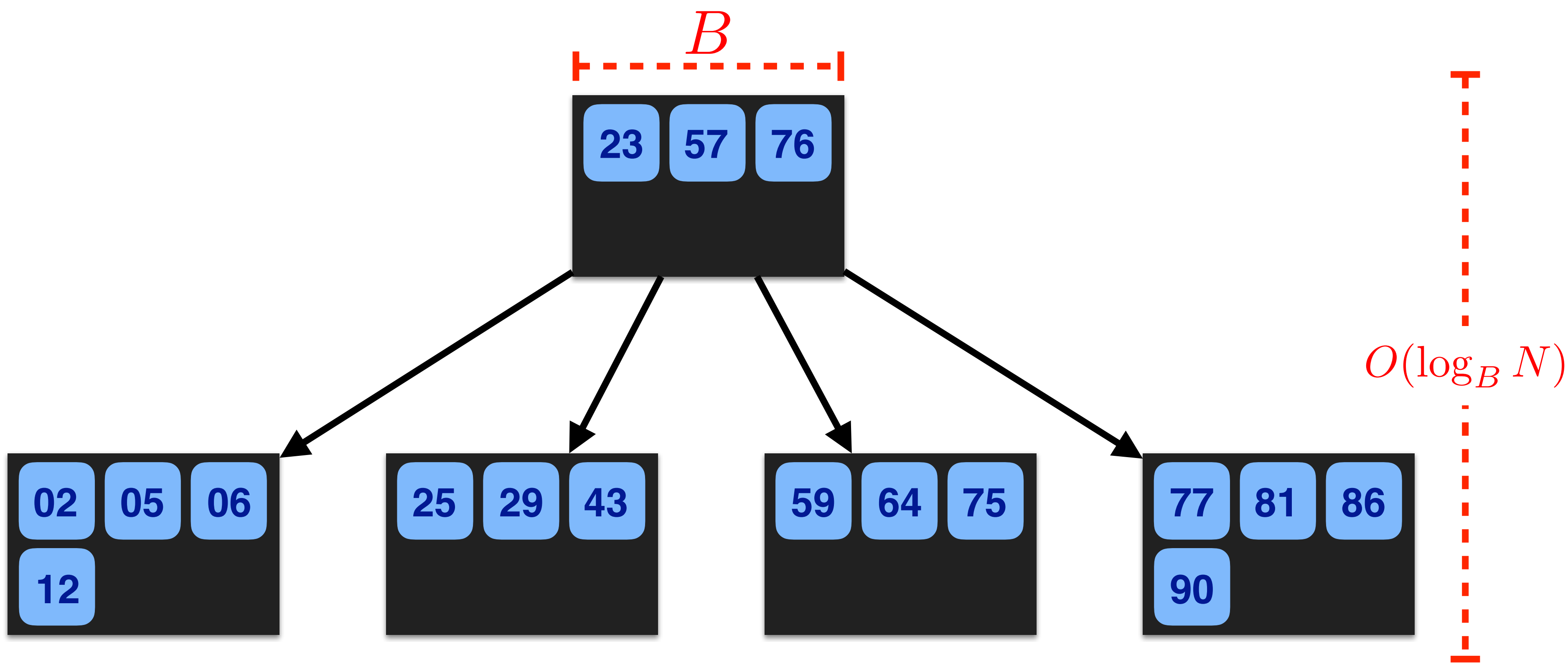
B-tree Insert

Steps

- Find the leaf node where your key-value pair belongs (point query)
- Insert your key-value pair into that leaf

B-tree: standard DAM dictionary

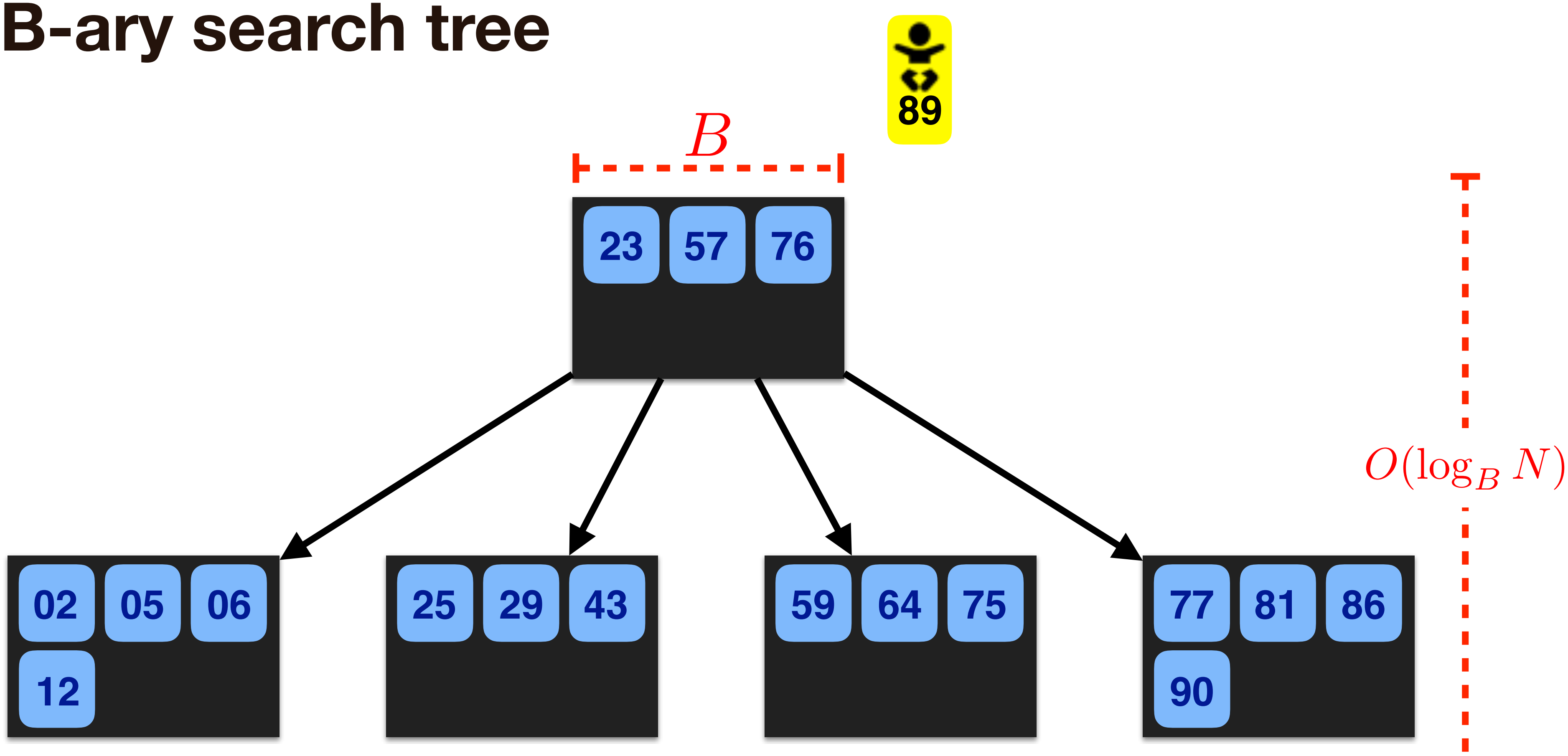
B-ary search tree



<u>Summary</u>	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

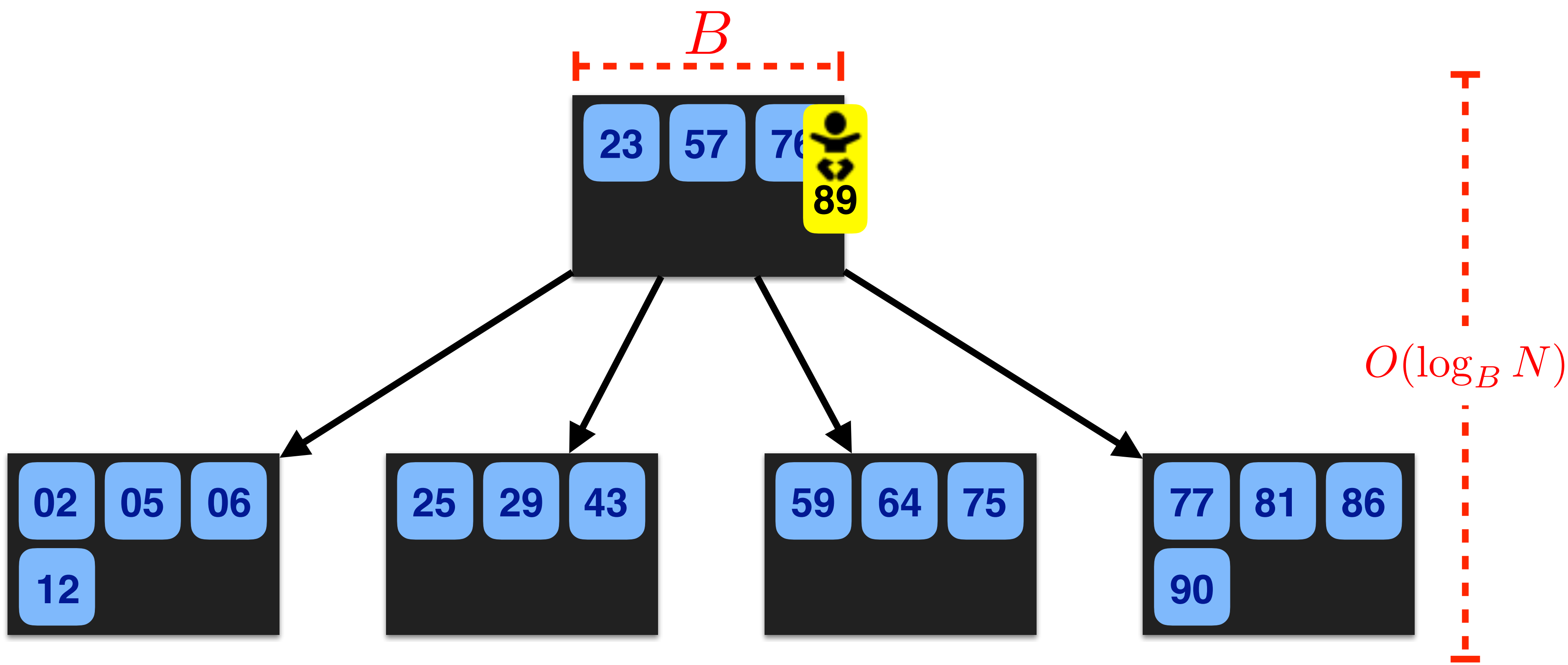
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

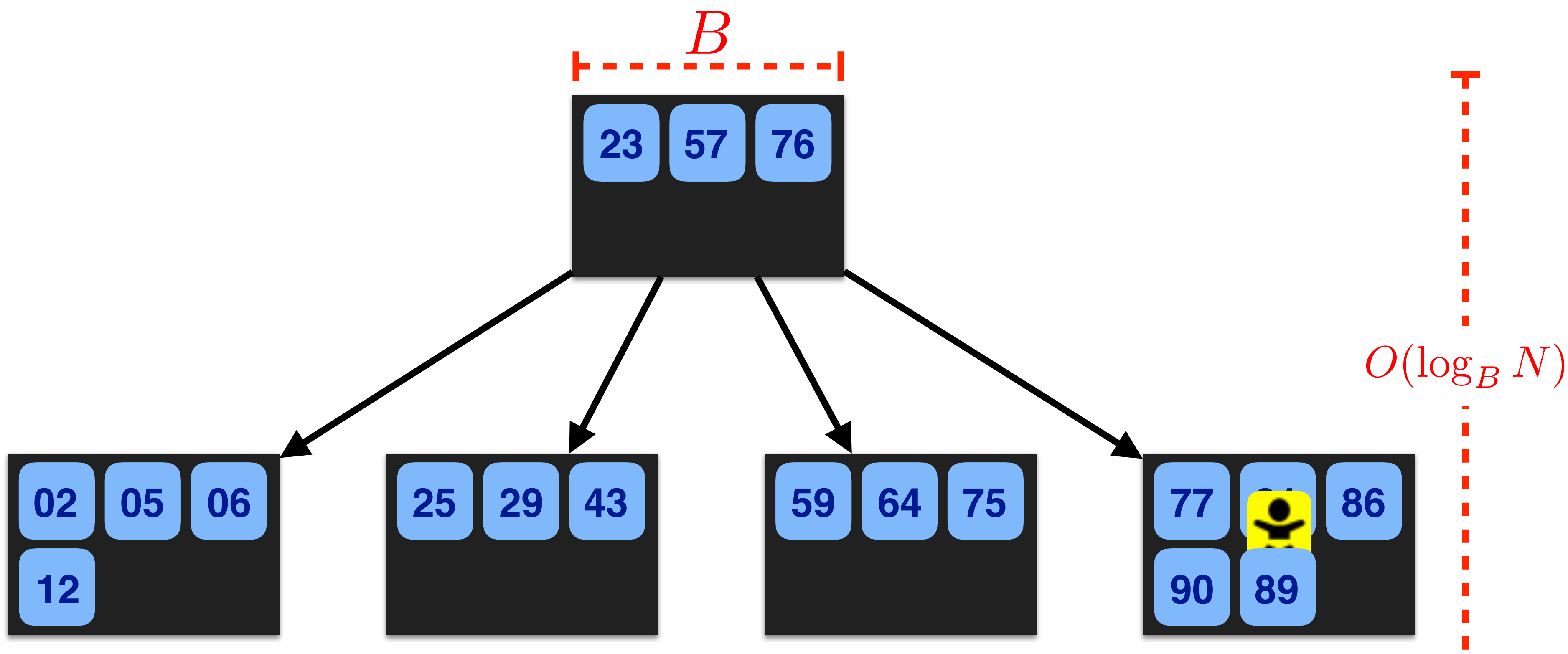
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

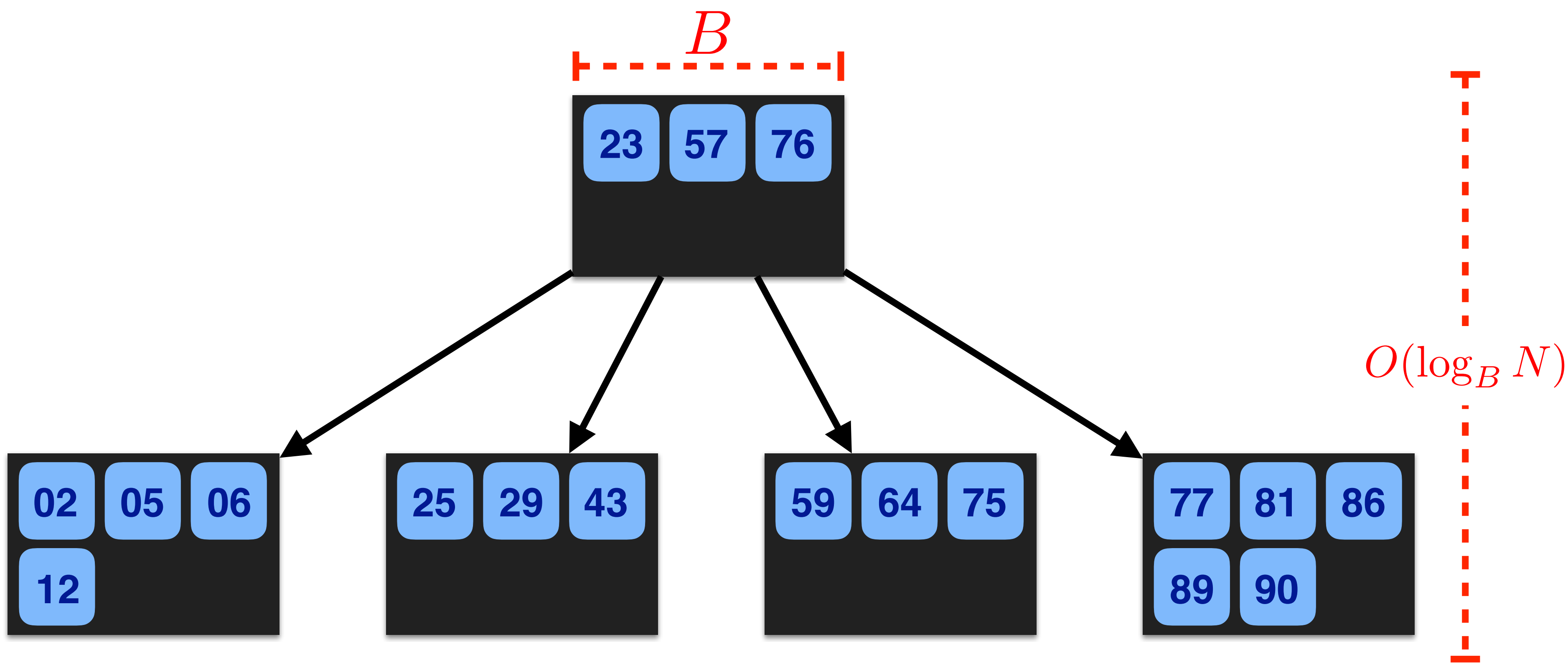
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

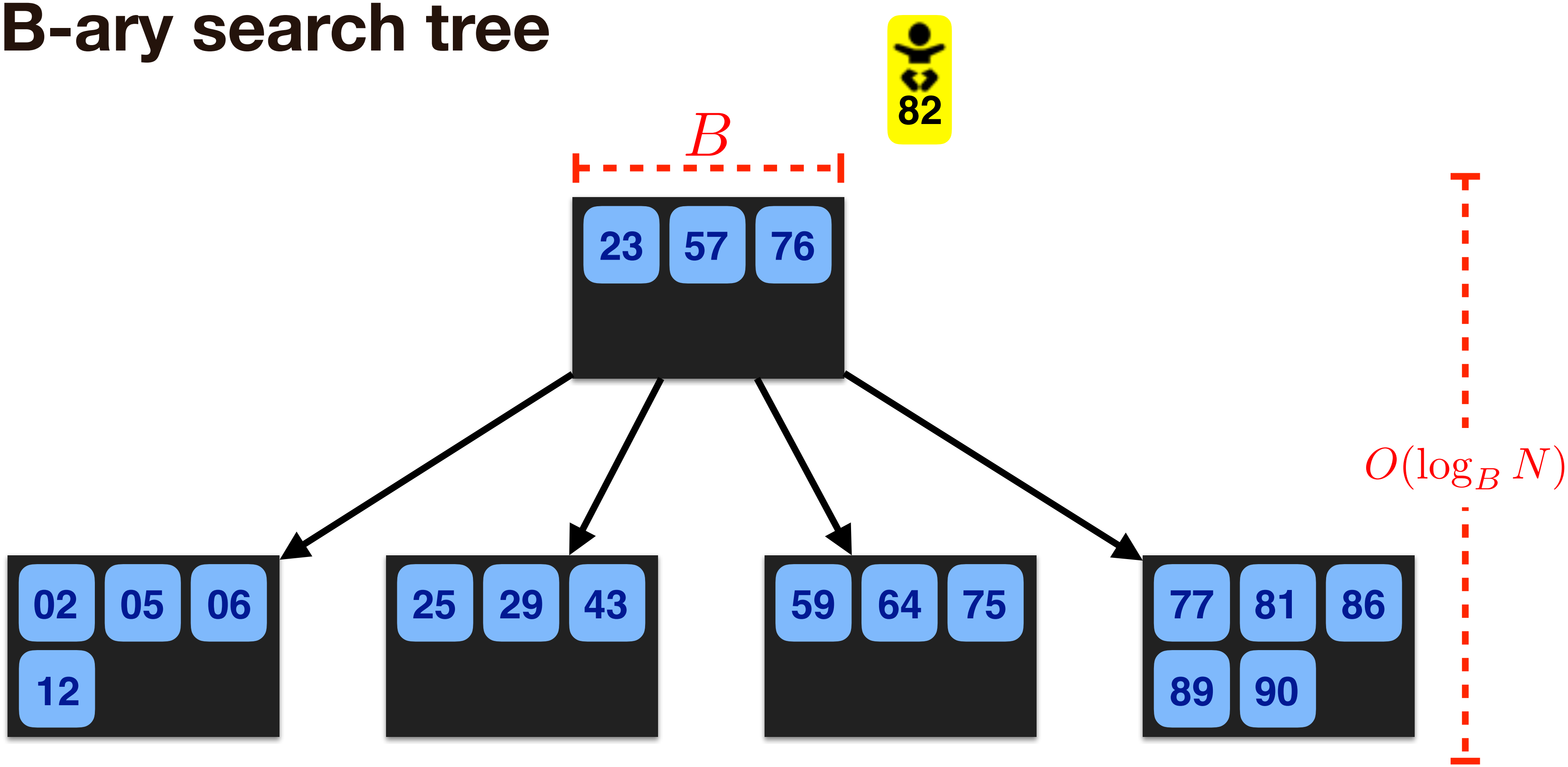
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

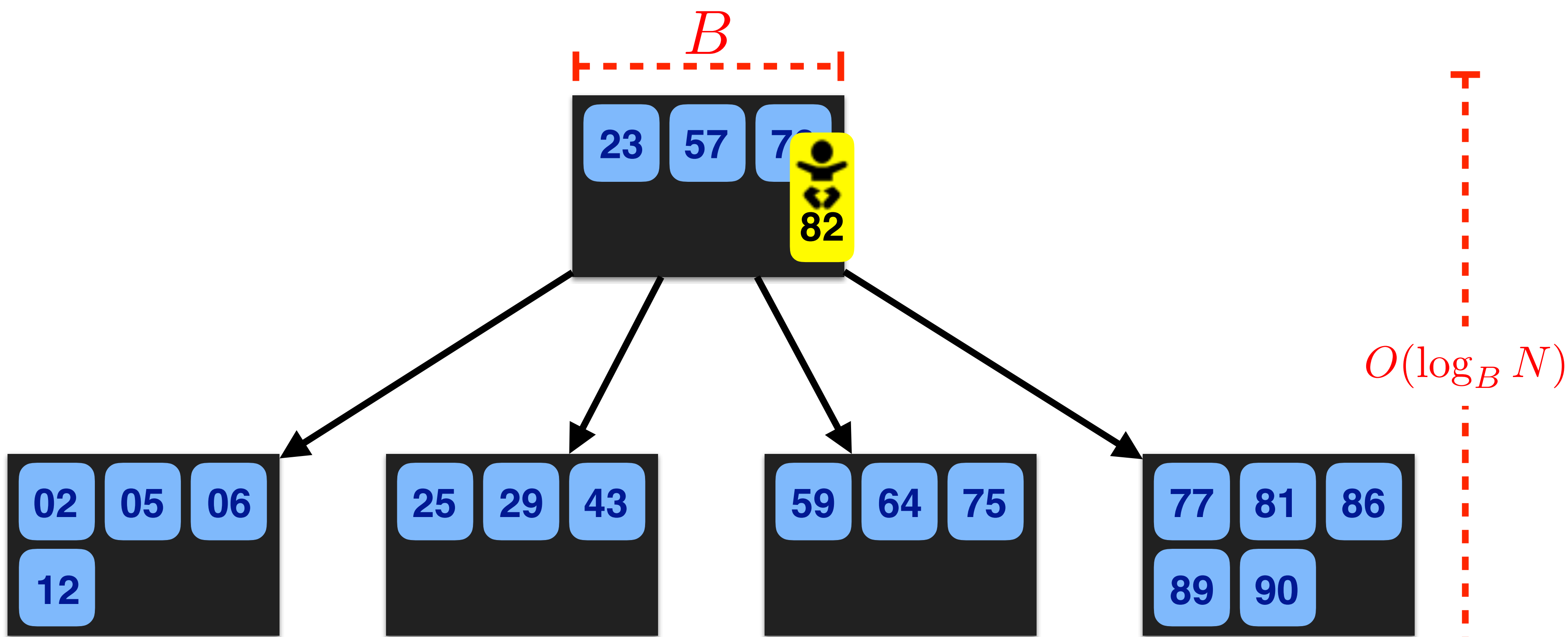
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

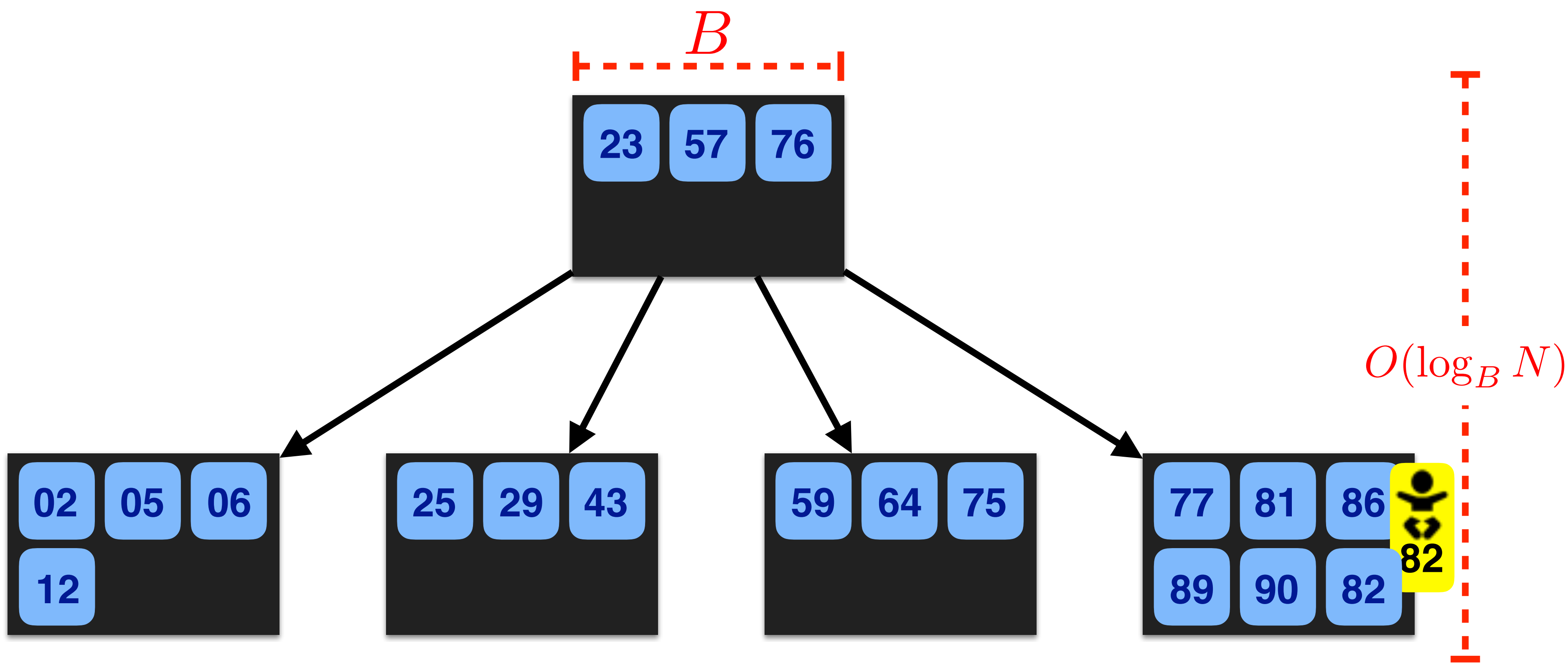
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

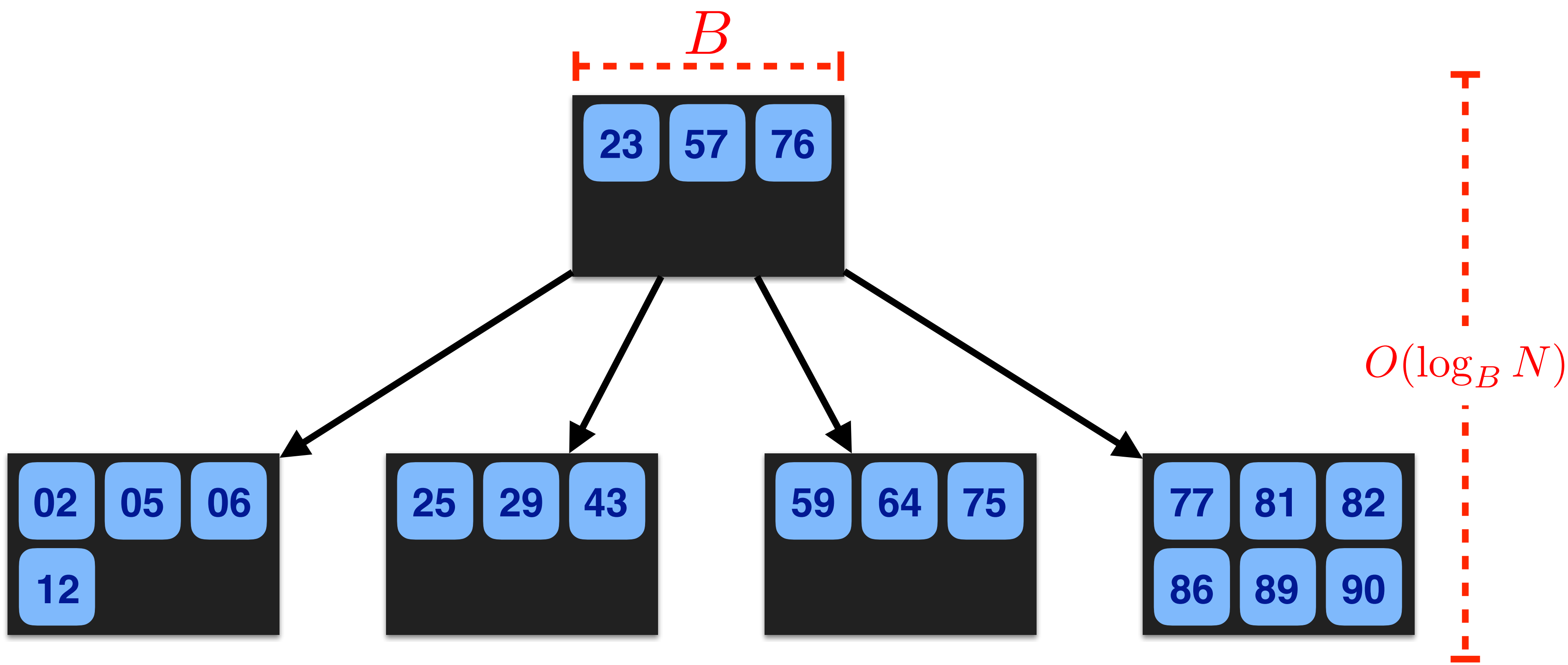
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

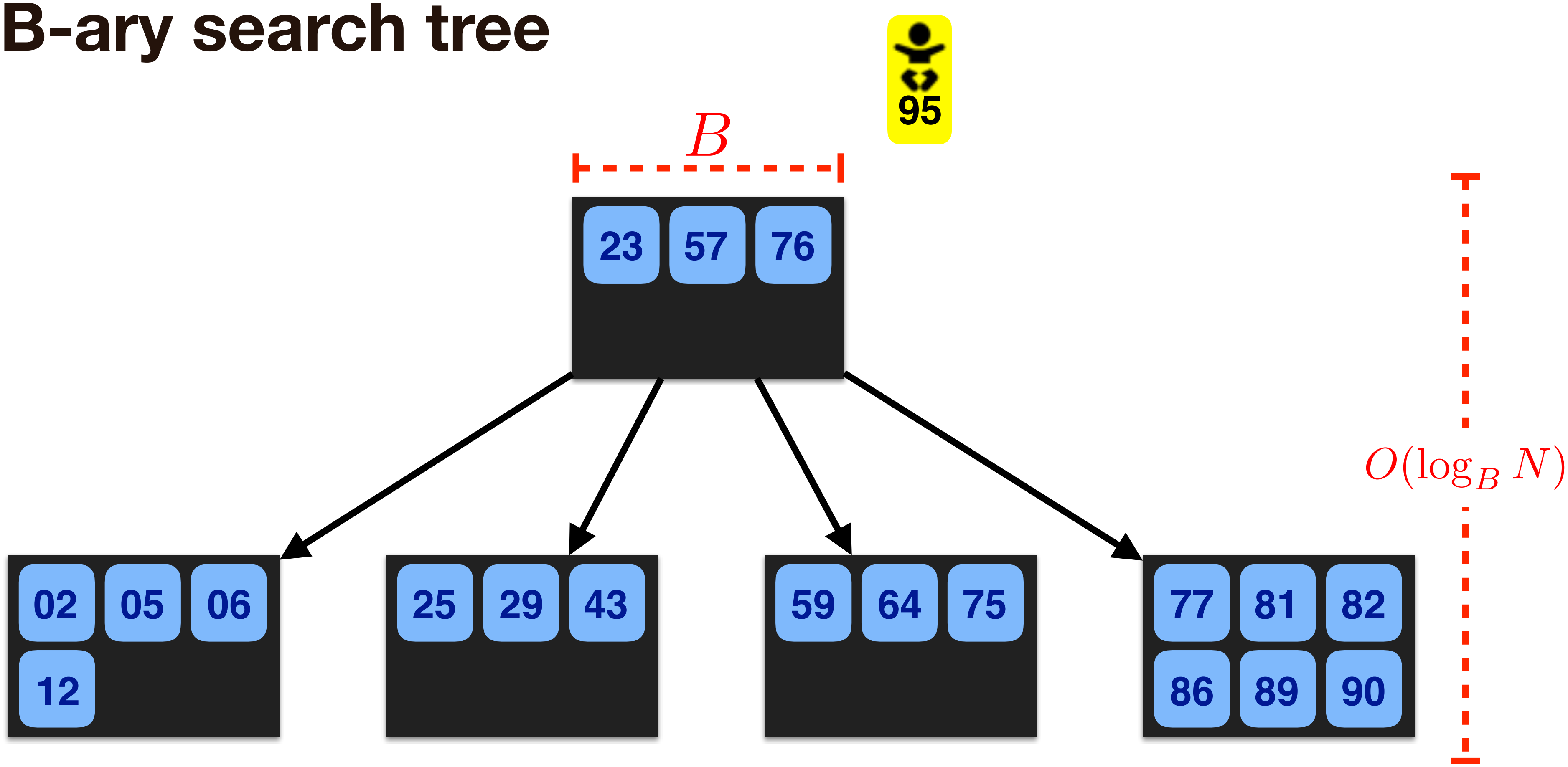
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

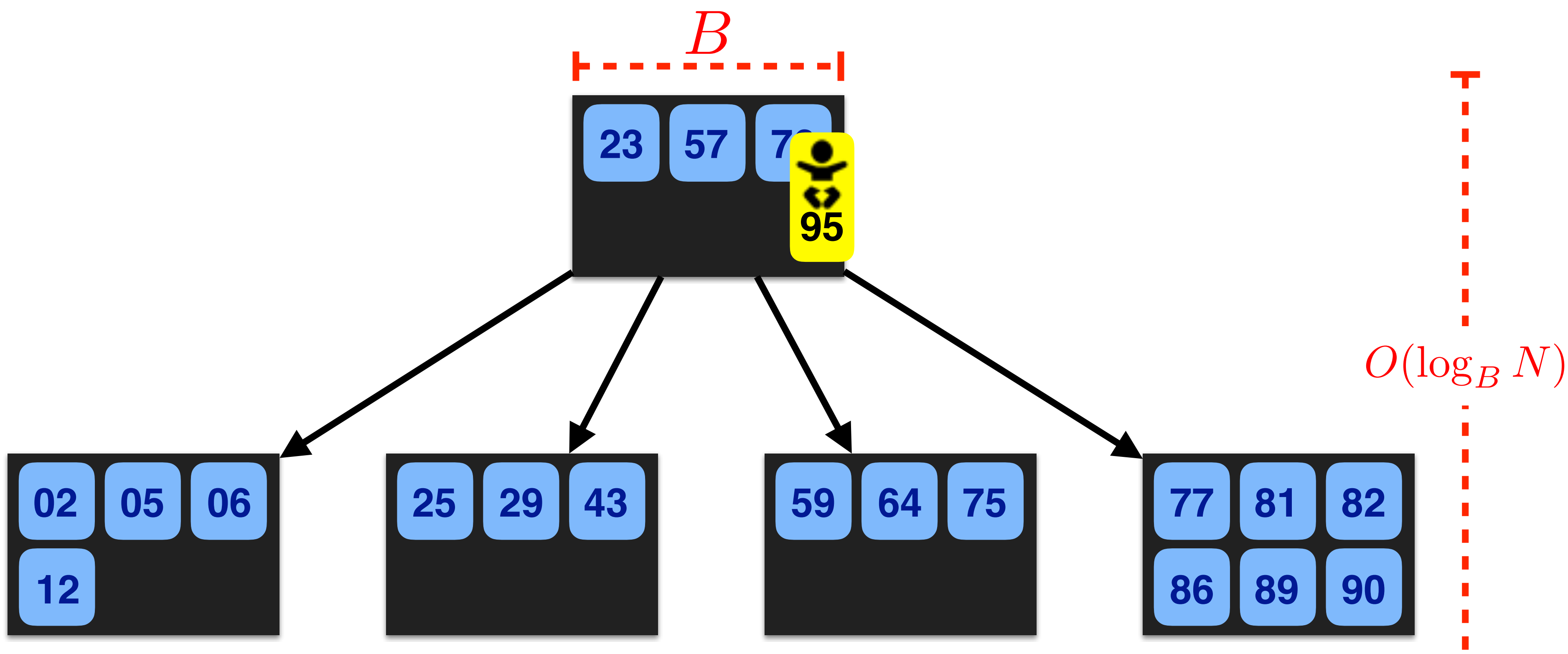
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

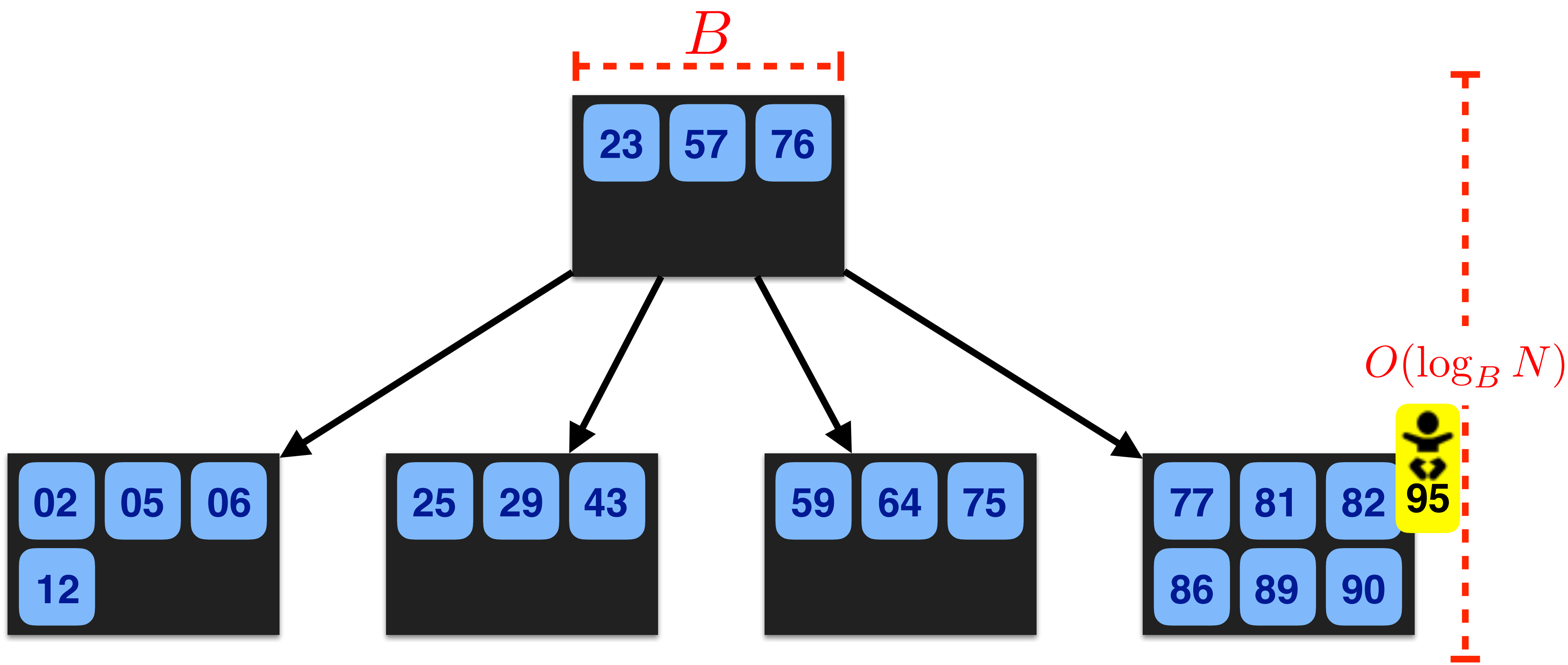
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

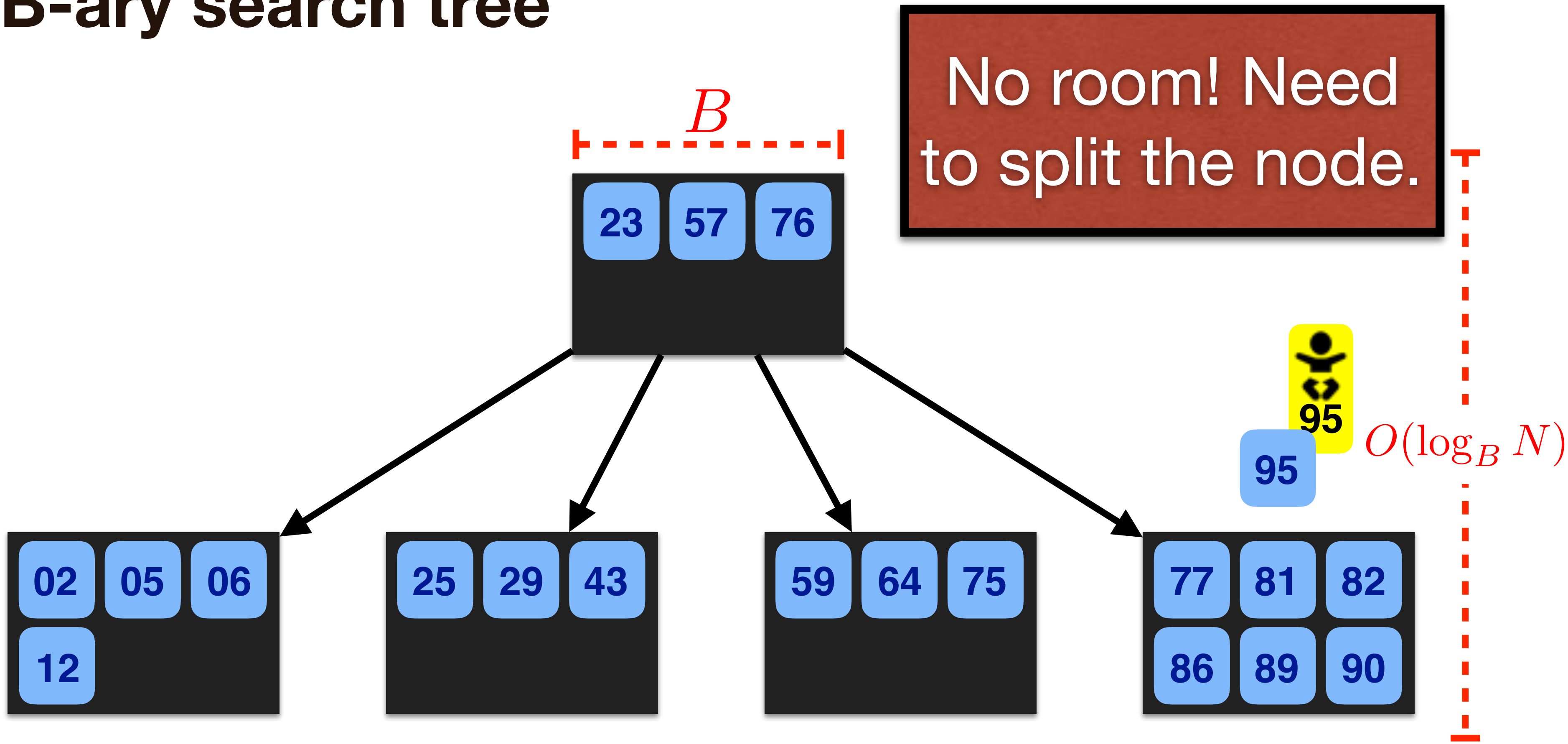
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

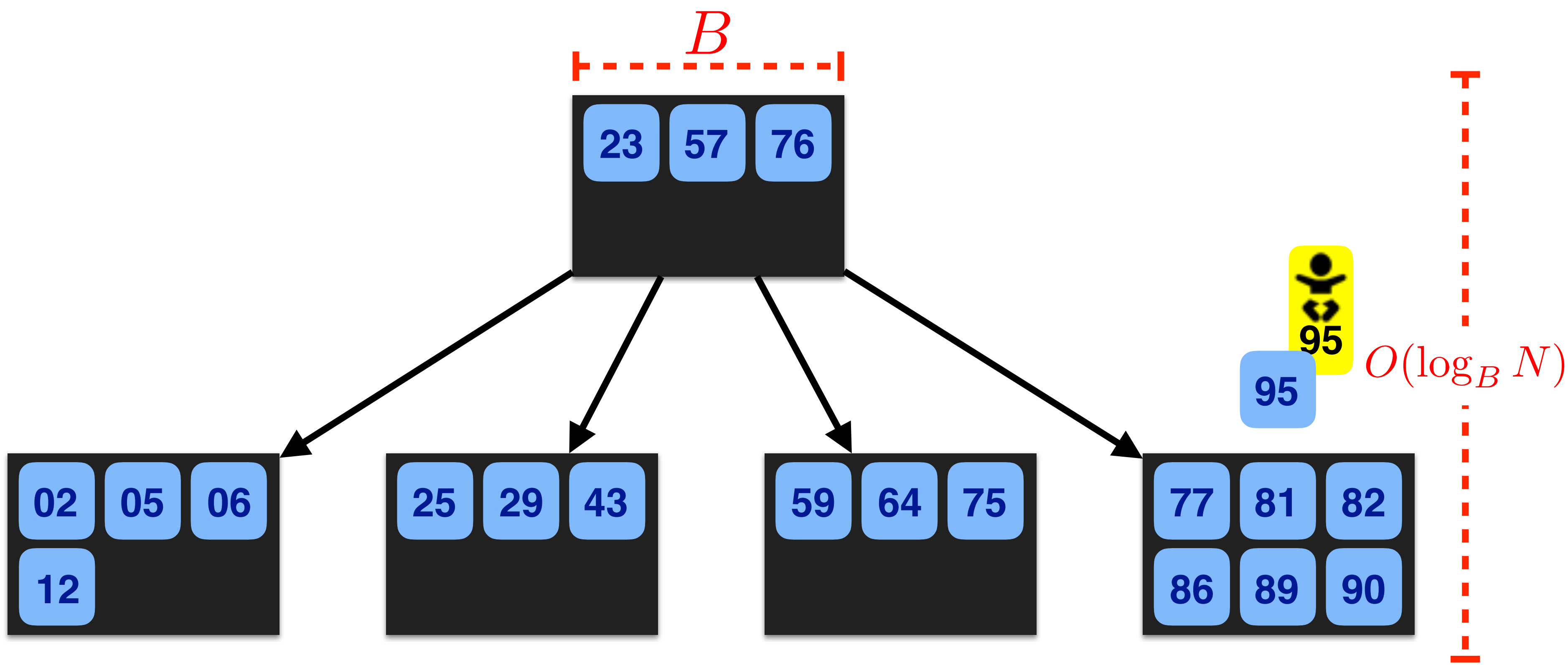
Splitting a B-tree node

Steps

- Sort all **2d+1** keys (**2d** + new key that causes overflow)
- Make new node with first **d** keys
- Make new node with last **d** keys
- Move middle key as a pivot of the parent
- Add pointers to new children
- Recurse up the tree if necessary (rare)

B-tree: standard DAM dictionary

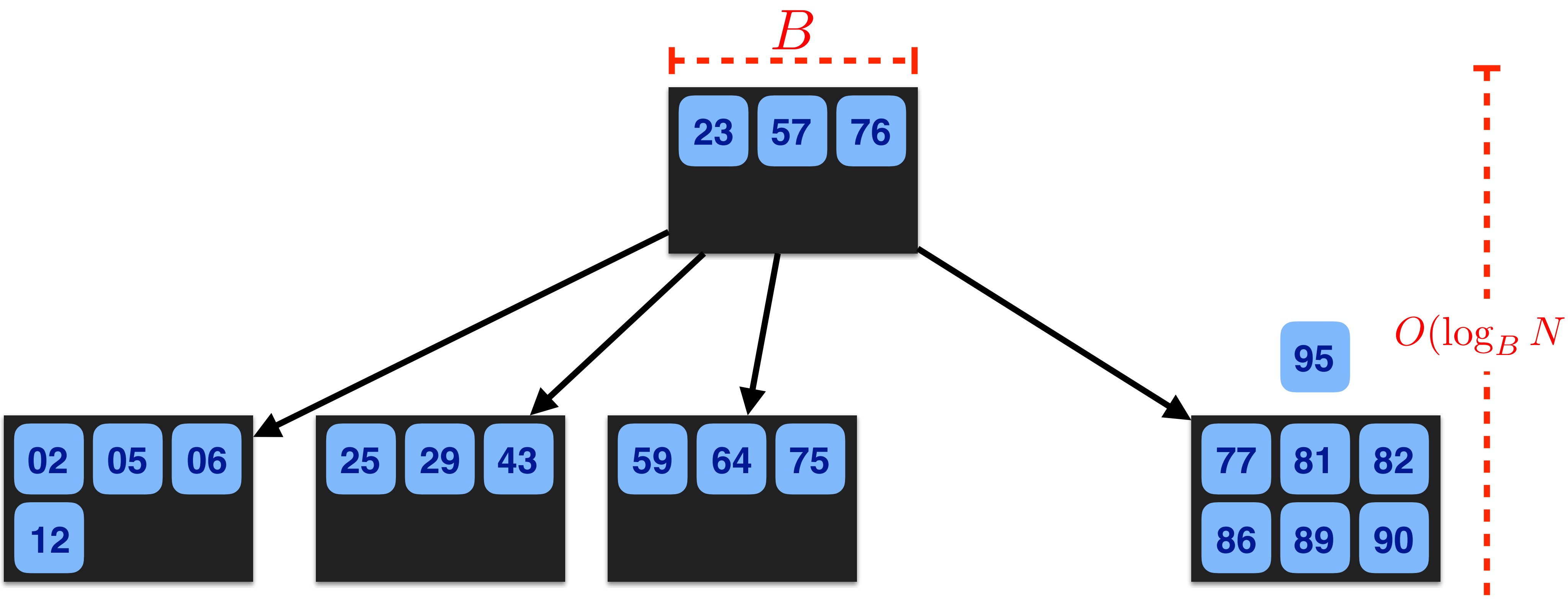
B-ary search tree



<u>**Summary**</u>	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

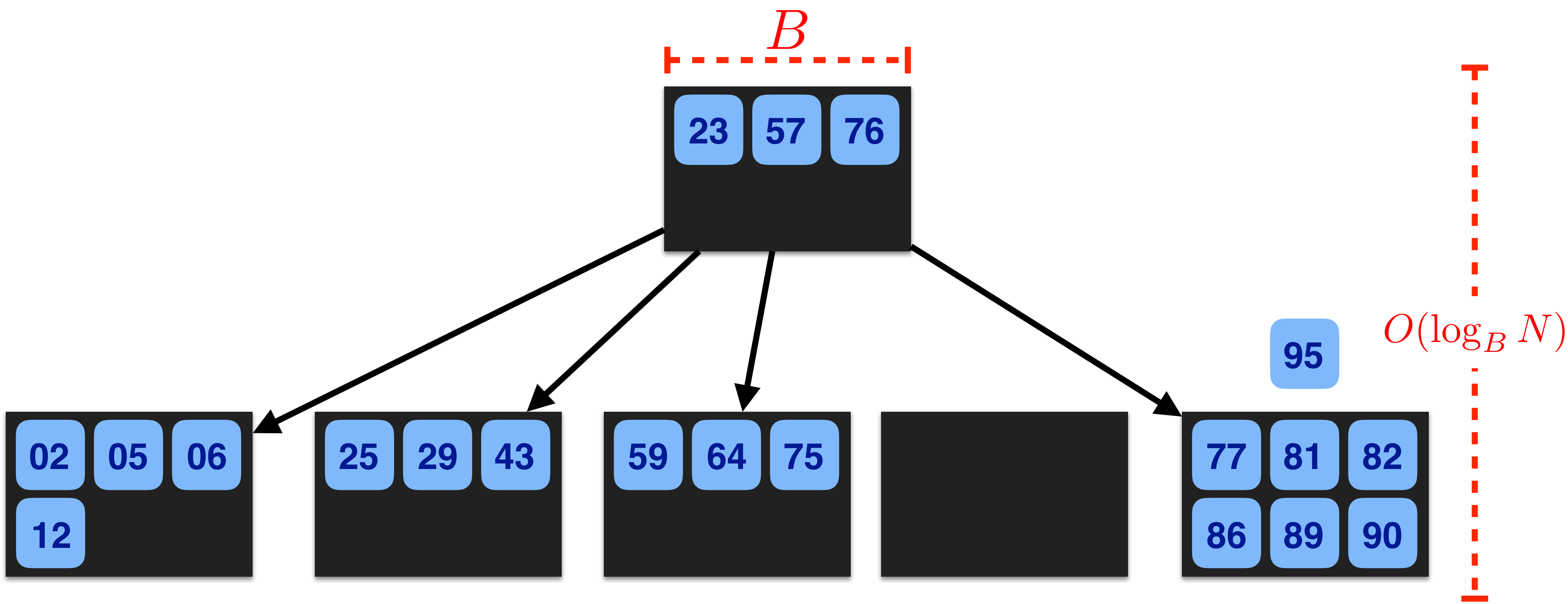
B-ary search tree



<u>Summary</u>	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

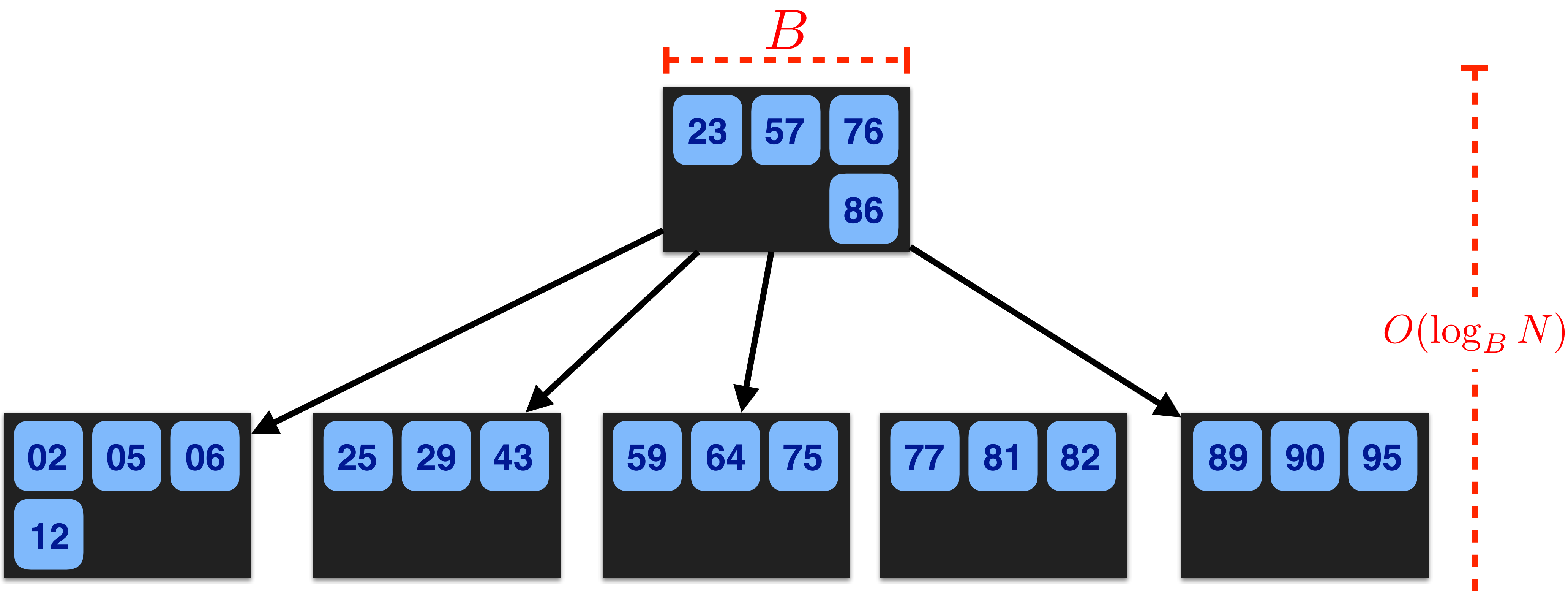
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

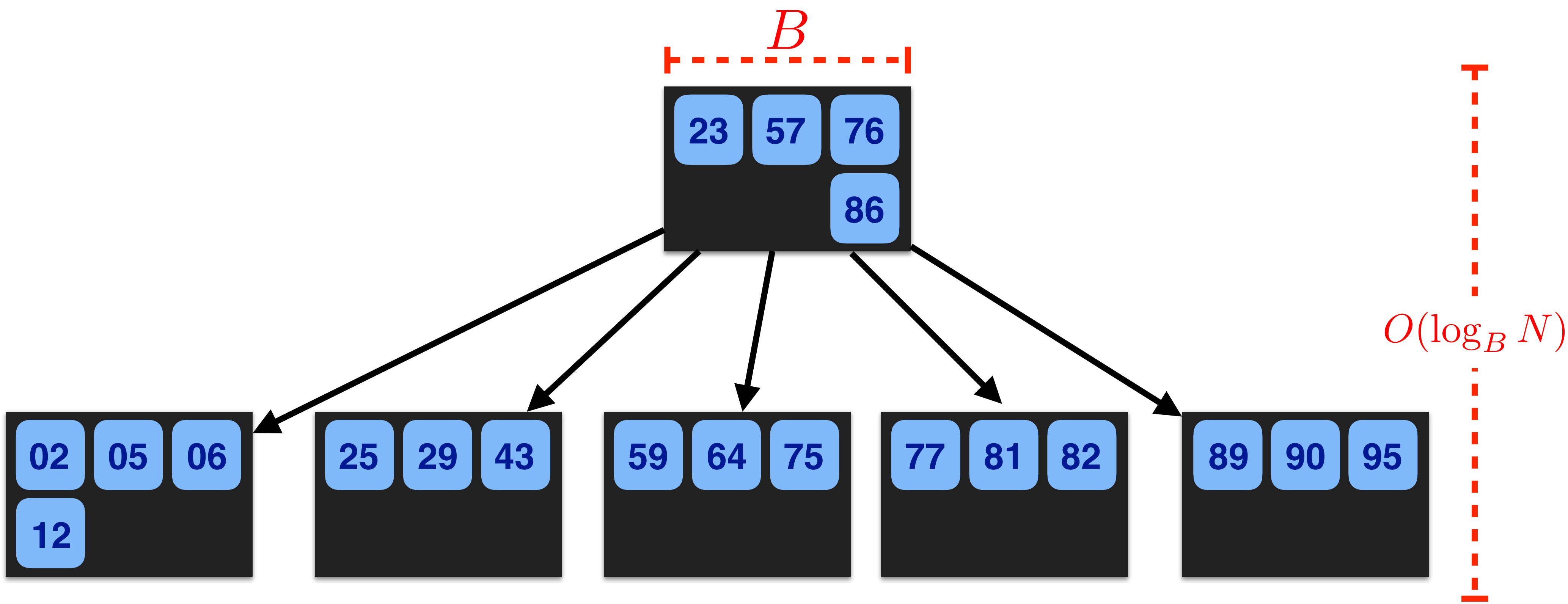
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

B-ary search tree



<u>Summary</u>	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

Splitting a B-tree node

Cost

- How many nodes must be read/written in a local split?
  - We read the node being split
  - We write the old node and the new node (first **d** keys, last **d** keys)
  - We read/write the parent node
- What if we overflow the parent?
  - If we recurse, we already read the parent, so we repeat the same steps one level above
- Total cost of an insert: O(h)
  - Reads: O(h)
  - Writes: O(2h)

B-tree Range Queries

B-tree Range Query

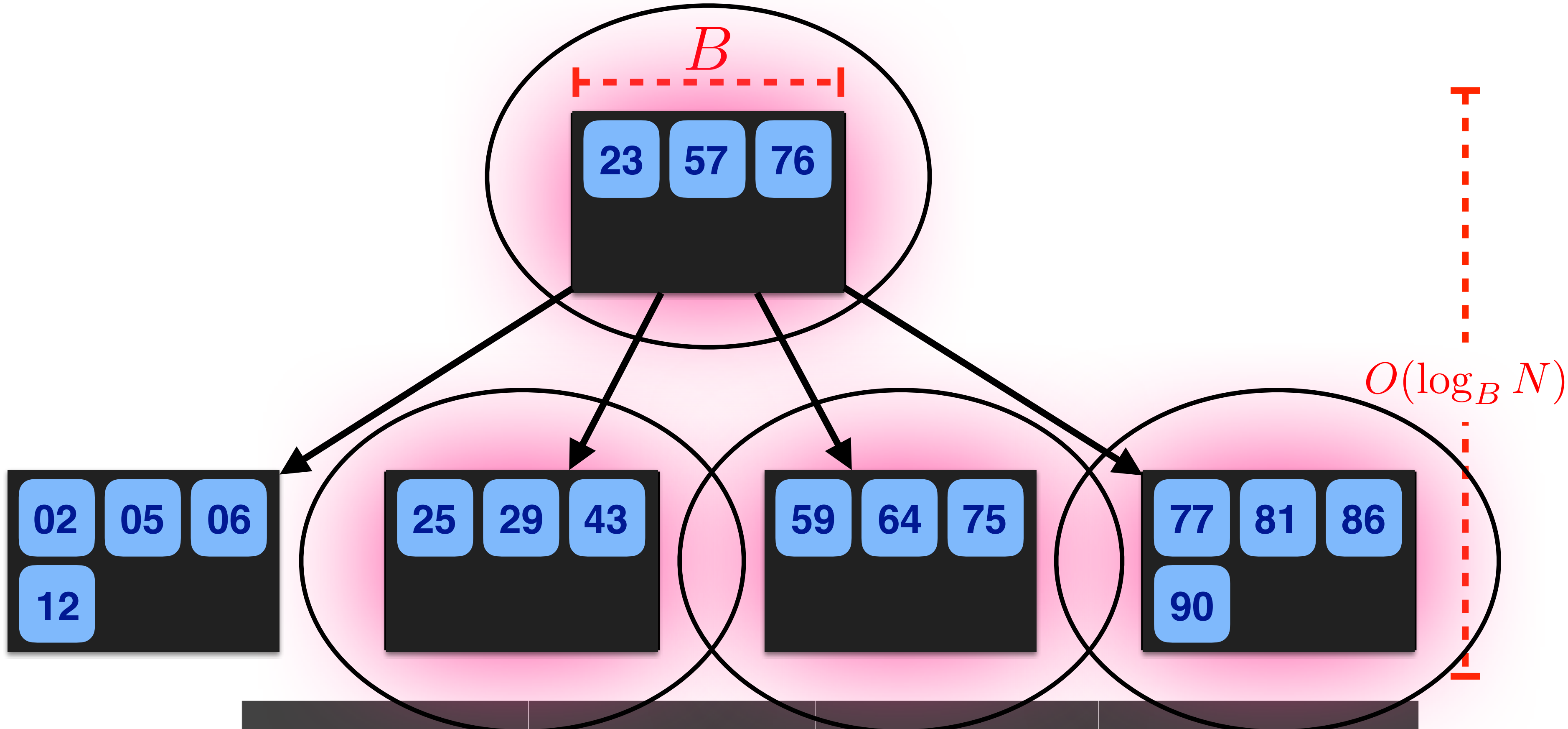
(Range query: point query + successor^k)

Steps

- Find the leaf node where the first key-value pair belongs (point query)
- Read all key-value pairs from that node that are part of your range
- Consult your parent to find its next child pointer
- Read all key-value pairs from that node that are part of your range
- Loop

B-tree: standard DAM dictionary

B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree Deletes

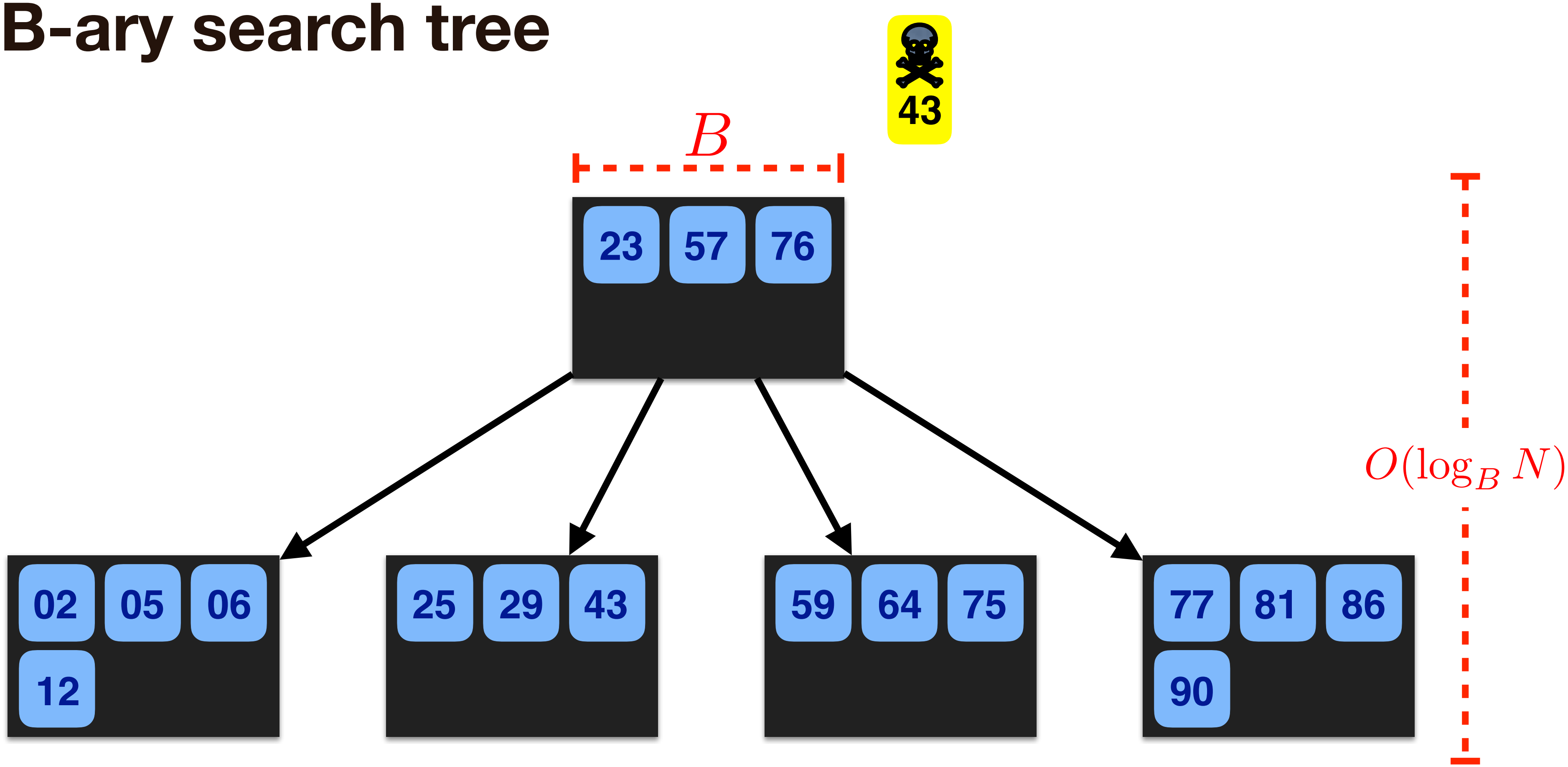
B-tree Deletions

Steps

- Search for the leaf containing the target key-value pair (point query)
- Remove the element from the leaf (if present)
- If the size of the node drops below **d**, merge with a neighbor
  - Remove extra pivot key and pointer from parent (the pointer to the node that is being deleted as part of the merge)
  - Merge contents of nodes
  - Write parent and merged node
  - If the parent size dropped below **d**, recurse upwards

B-tree: standard DAM dictionary

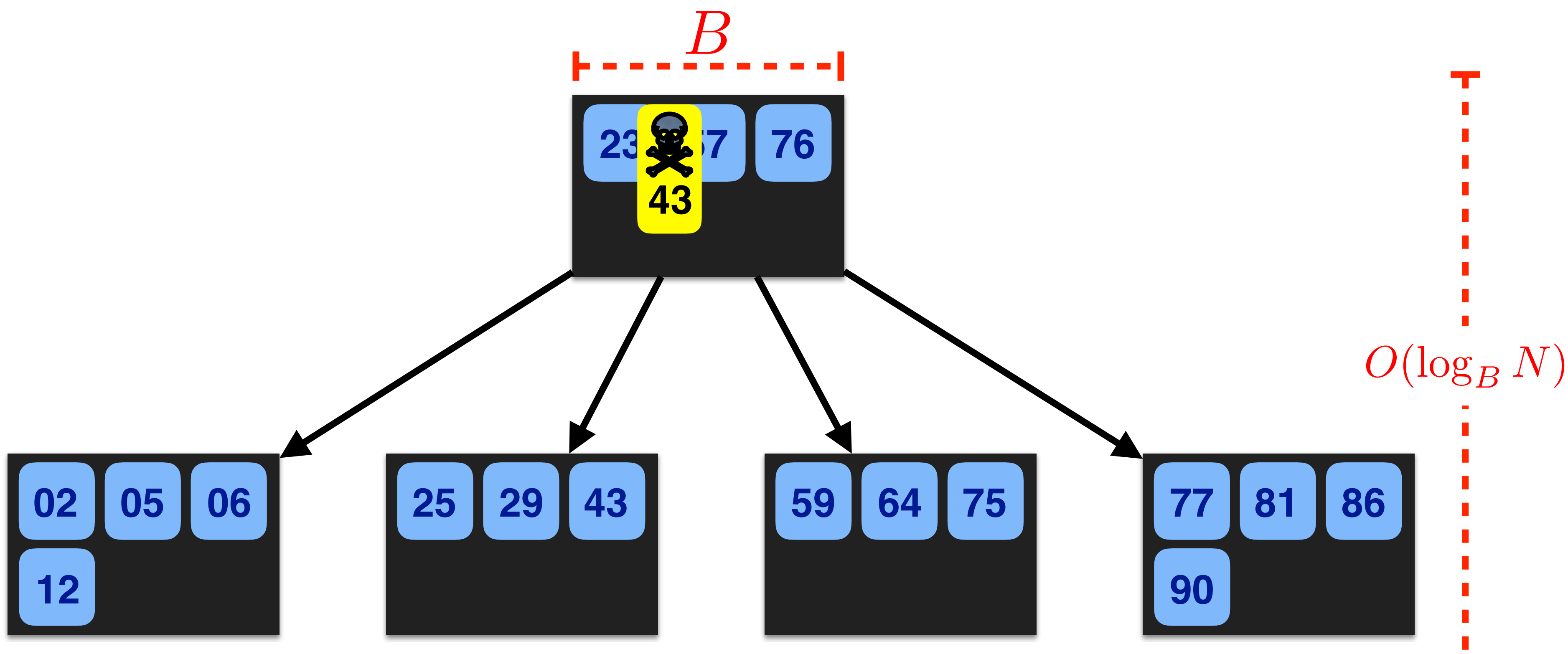
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

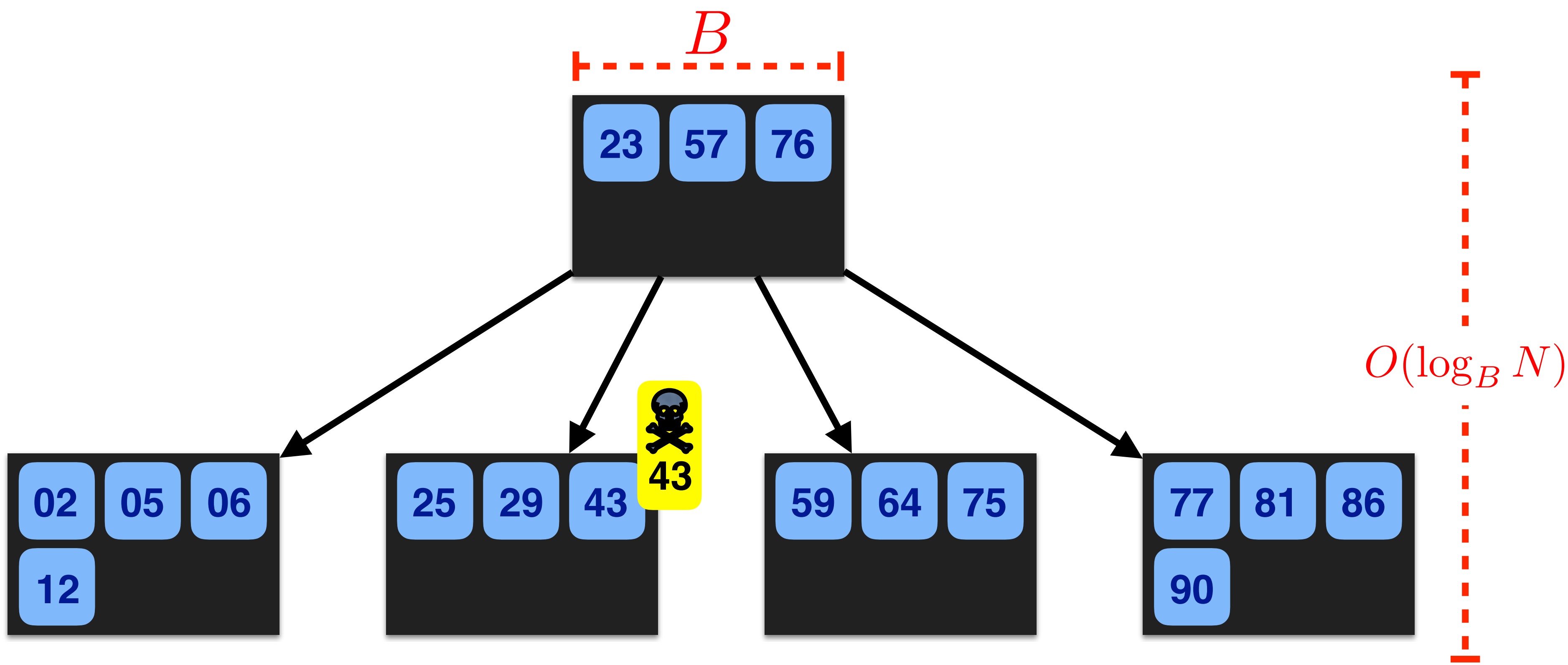
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

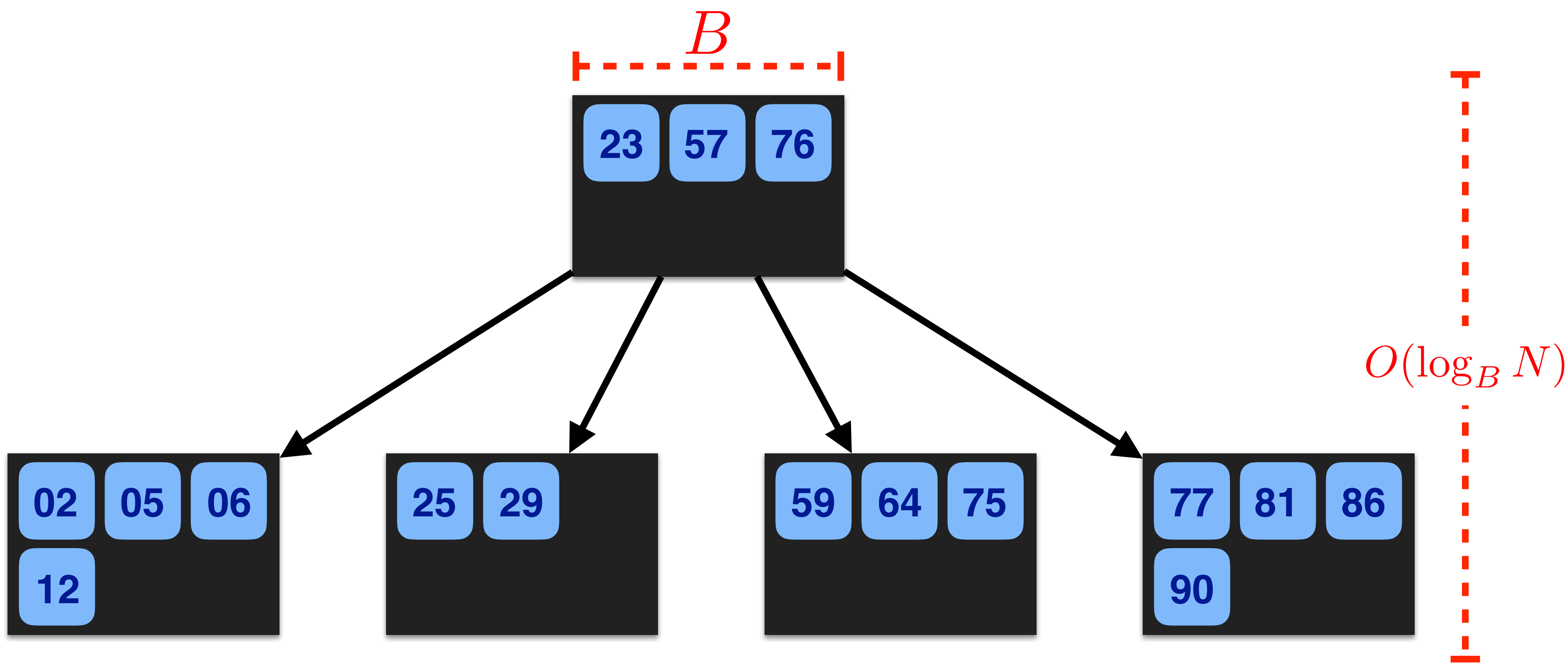
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

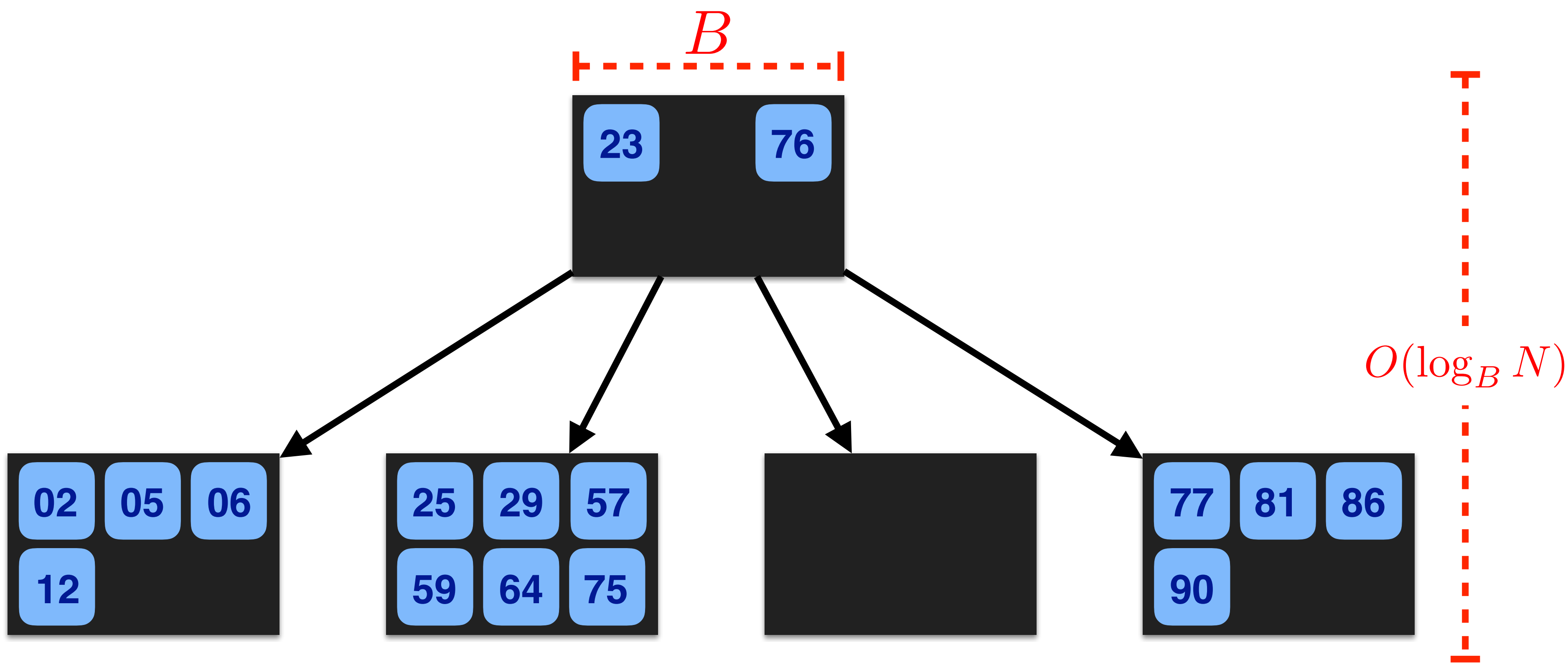
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

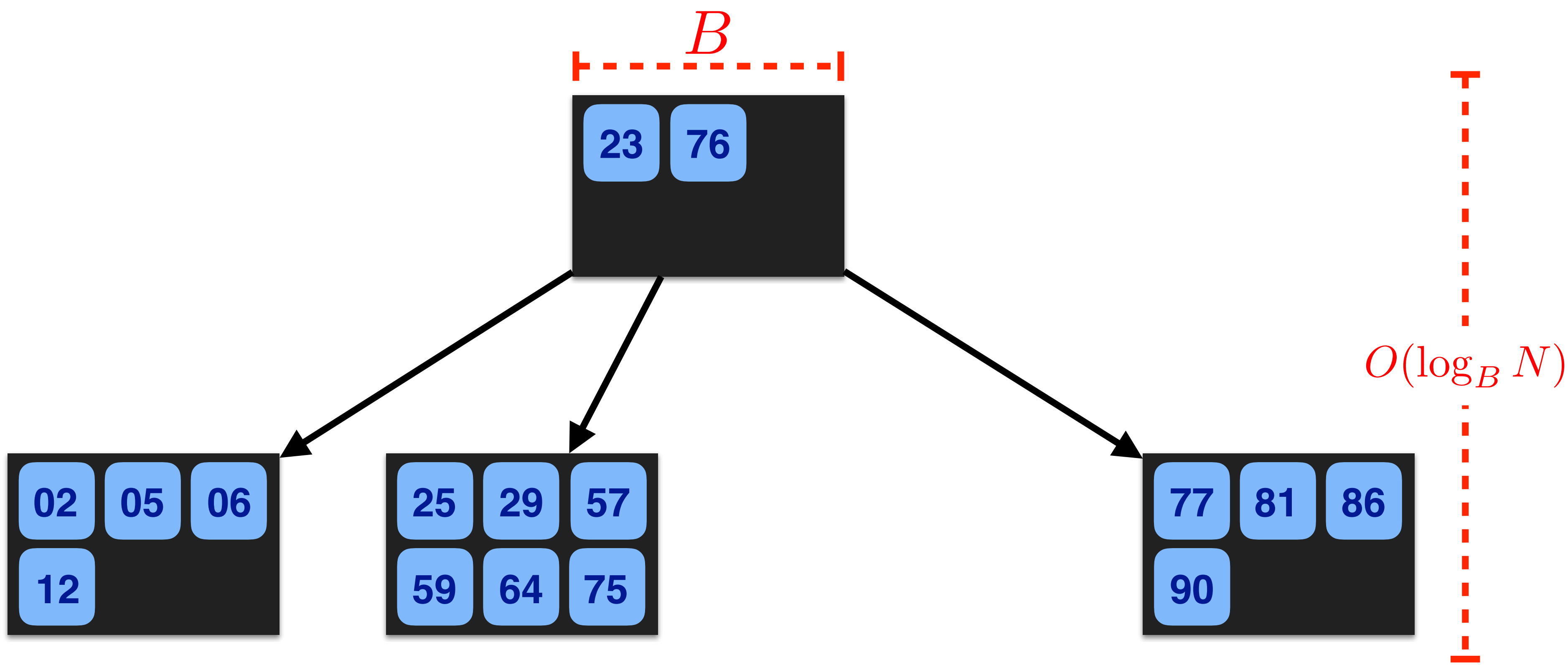
B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

B-tree: standard DAM dictionary

B-ary search tree



Summary	Point Query	Insert	Delete	Range Query
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$	$O\left(\log_B N + \frac{K}{B}\right)$

Summary

- B-trees are the de-facto search structure for external memory applications
- Variants exist to tune utilization and range scan performance, but the idea is the same
- We can analyze performance using the DAM model

Other discussions

- Concurrent access - how to lock the tree?
  - Hand-over-hand locking for queries
  - Reservations or top-down splitting
- How to choose the node size (B)?
  - Must balance competing goals:
    - Small B minimizes write amplification (each update requires writing whole node)
    - Large B minimizes fragmentation (more data read per seek)

Looking Ahead

More trees

- Log structured merge trees (next class)
- B^{e}-trees Monday

Write optimization

- Making our trees lazy!
  - Better I/O performance for writes
  - Not worse off for reads