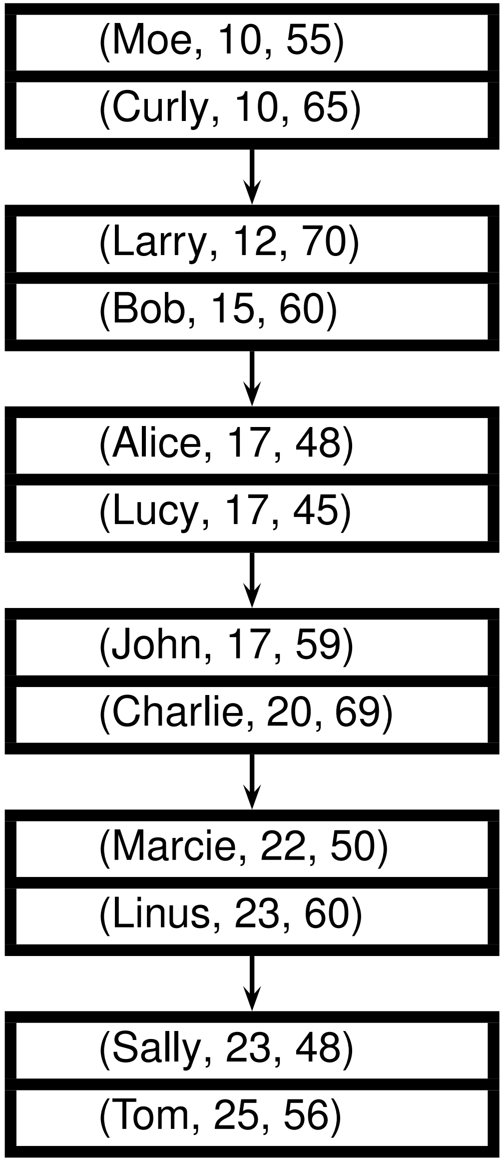
CS3223 Lecture 2
Indexing

Example

Relation R

name	age	weight
Moe	10	55
Curly	10	65
Larry	12	70
Bob	15	60
Alice	17	48
Lucy	17	45
John	17	59
Charlie	20	69
Marcie	22	50
Linus	23	60
Sally	23	48
Tom	25	56

Data File for R

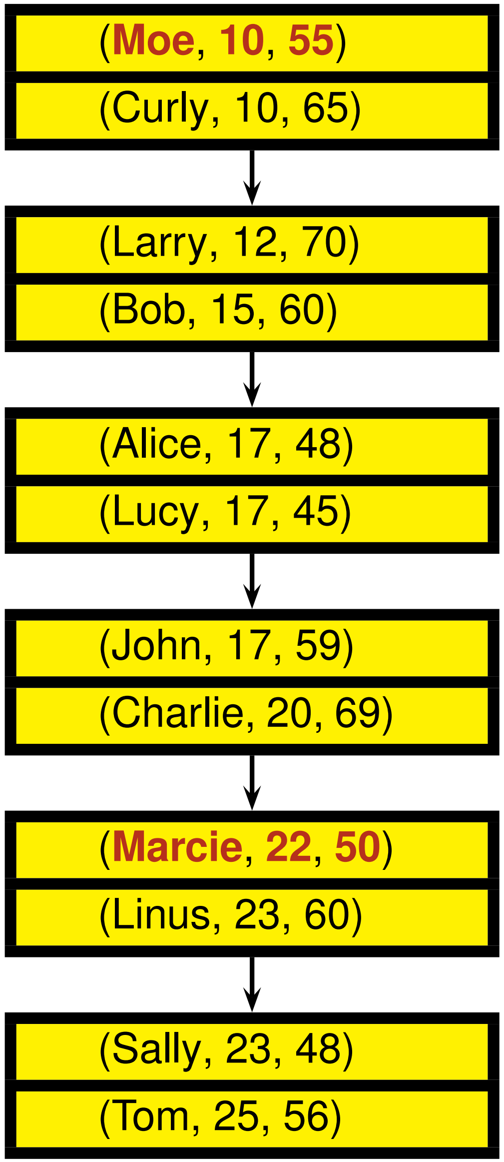


Example

Relation R

name	age	weight
Moe	10	55
Curly	10	65
Larry	12	70
Bob	15	60
Alice	17	48
Lucy	17	45
John	17	59
Charlie	20	69
Marcie	22	50
Linus	23	60
Sally	23	48
Tom	25	56

Data File for R

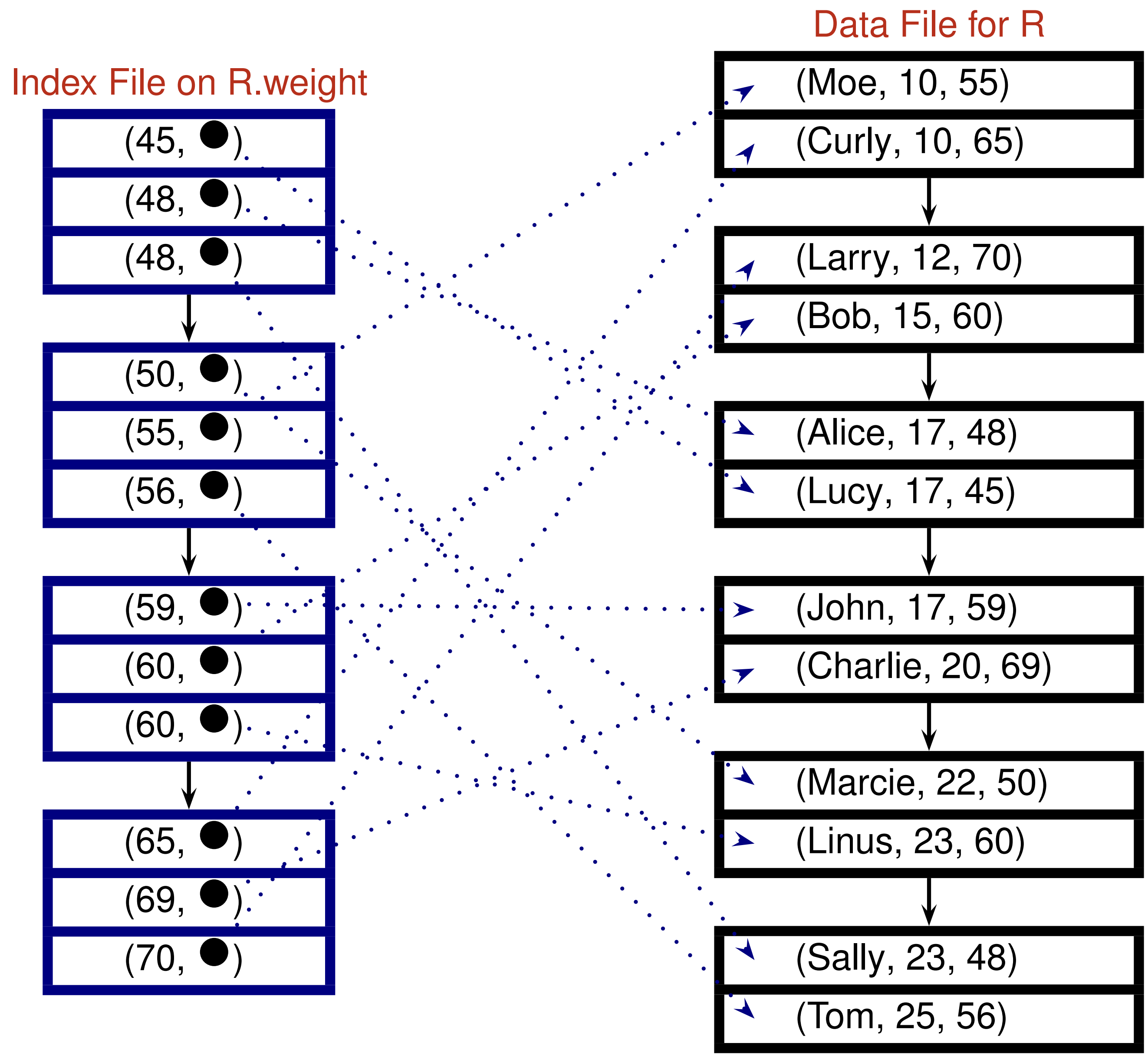


SELECT * FROM R WHERE weight BETWEEN 50 AND 55

Index

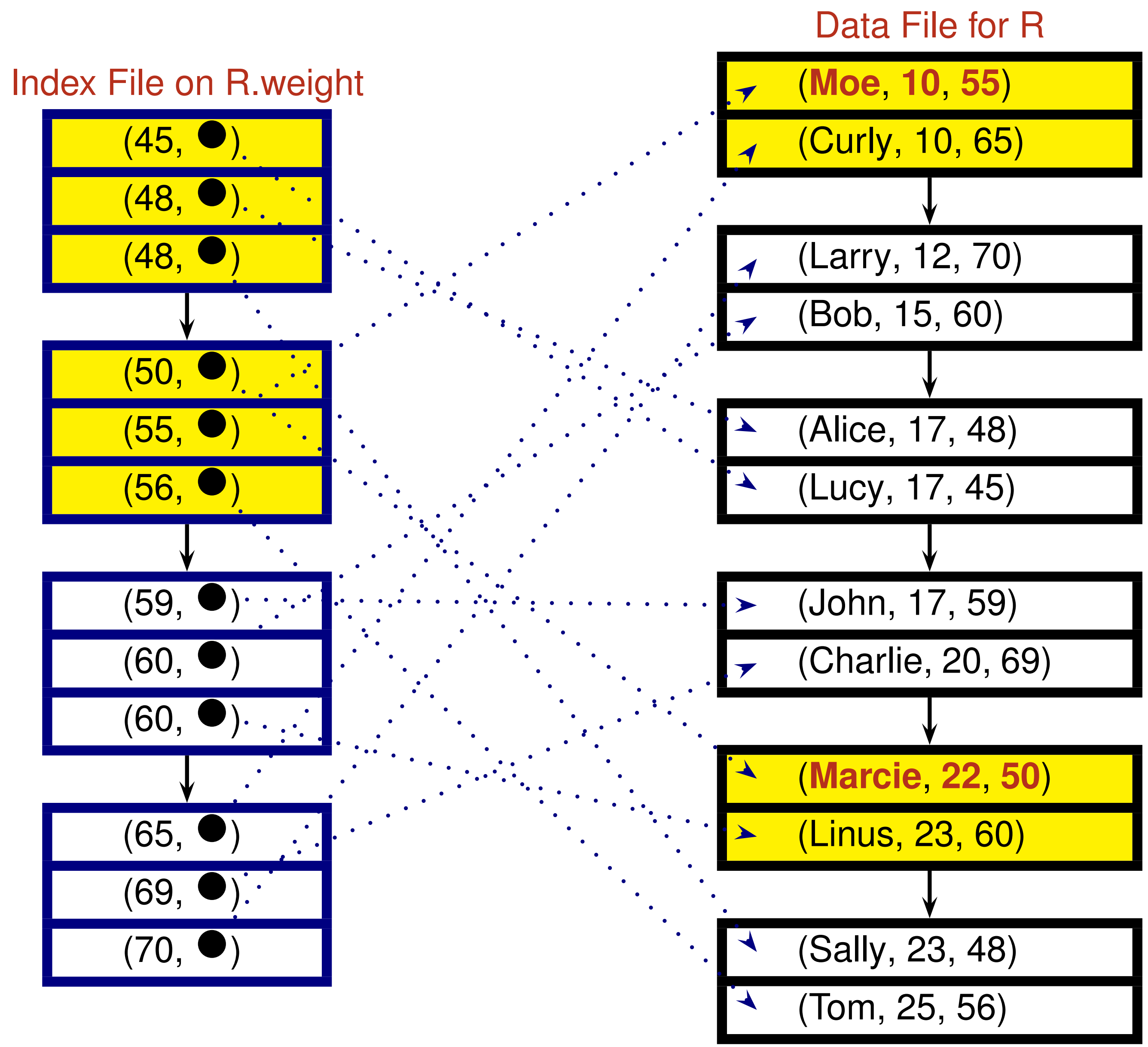
- An index is a data structure to speed up retrieval of data records based on some search key
- A search key is a sequence of k data attributes, $k \geq 1$
  - A search key is known as a composite search key if $k > 1$
  - Example of composite search key: (state, city)
- An index is a unique index if its search key is a candidate key; otherwise, it is a non-unique index
- An index is stored as a file
  - records in an index file are referred to as data entries

Simple Index Example



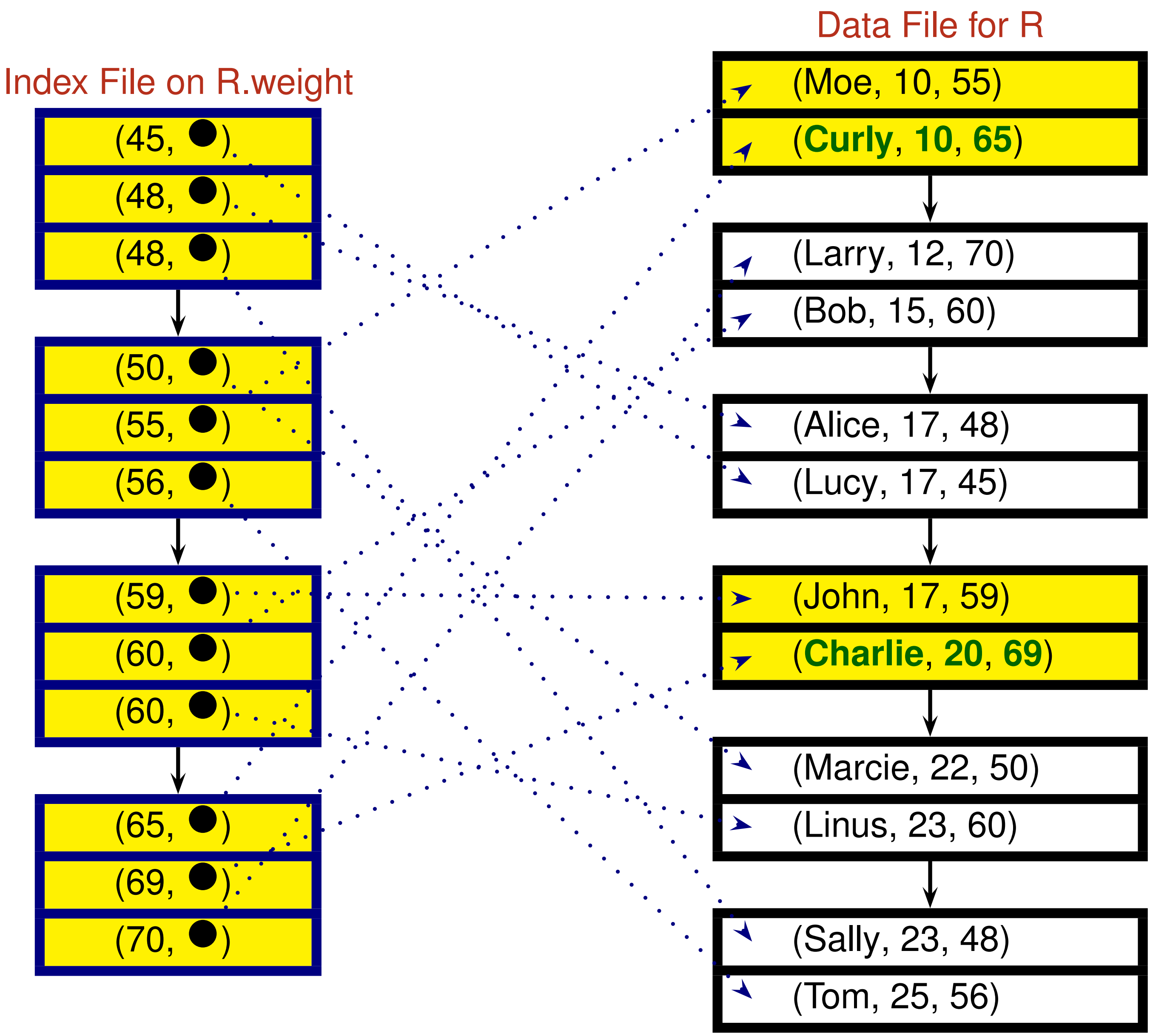
(50, •) is the data entry for the data record (Marcie, 22, 50); • denote a RID value

Simple Index Example



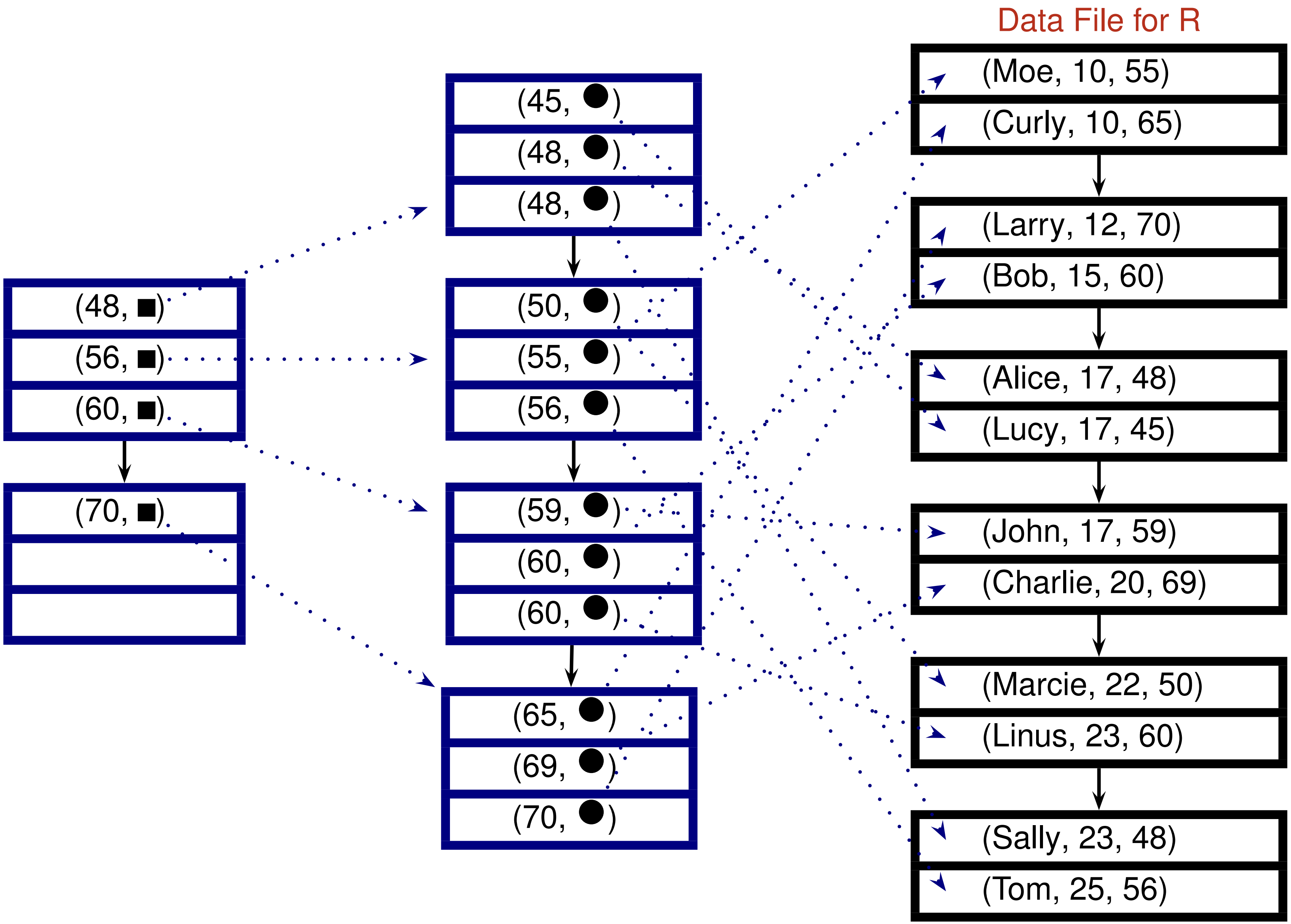
SELECT * FROM R WHERE weight BETWEEN 50 AND 55

Simple Index Example



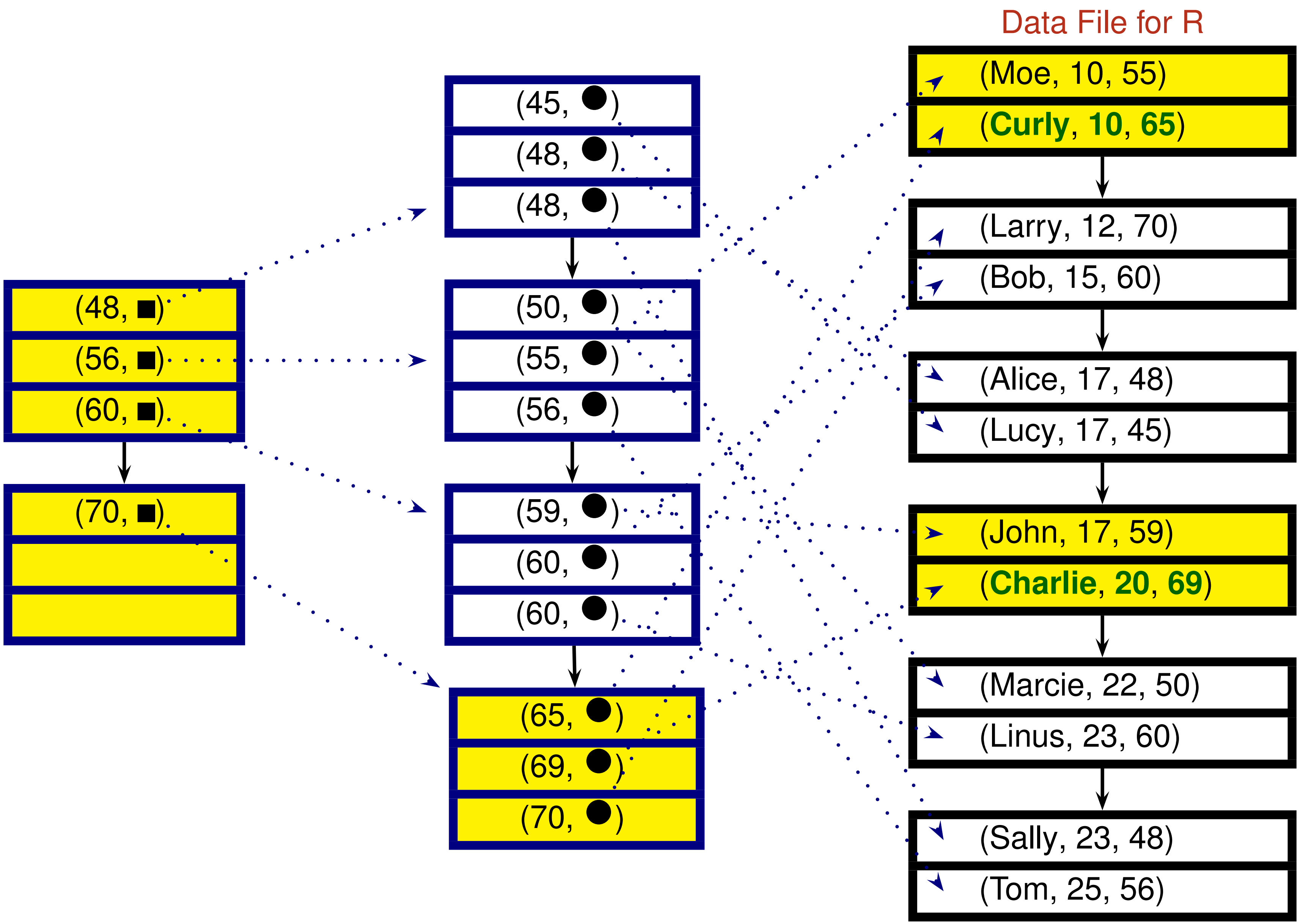
SELECT * FROM R WHERE weight BETWEEN 65 AND 69

Simple Index Example



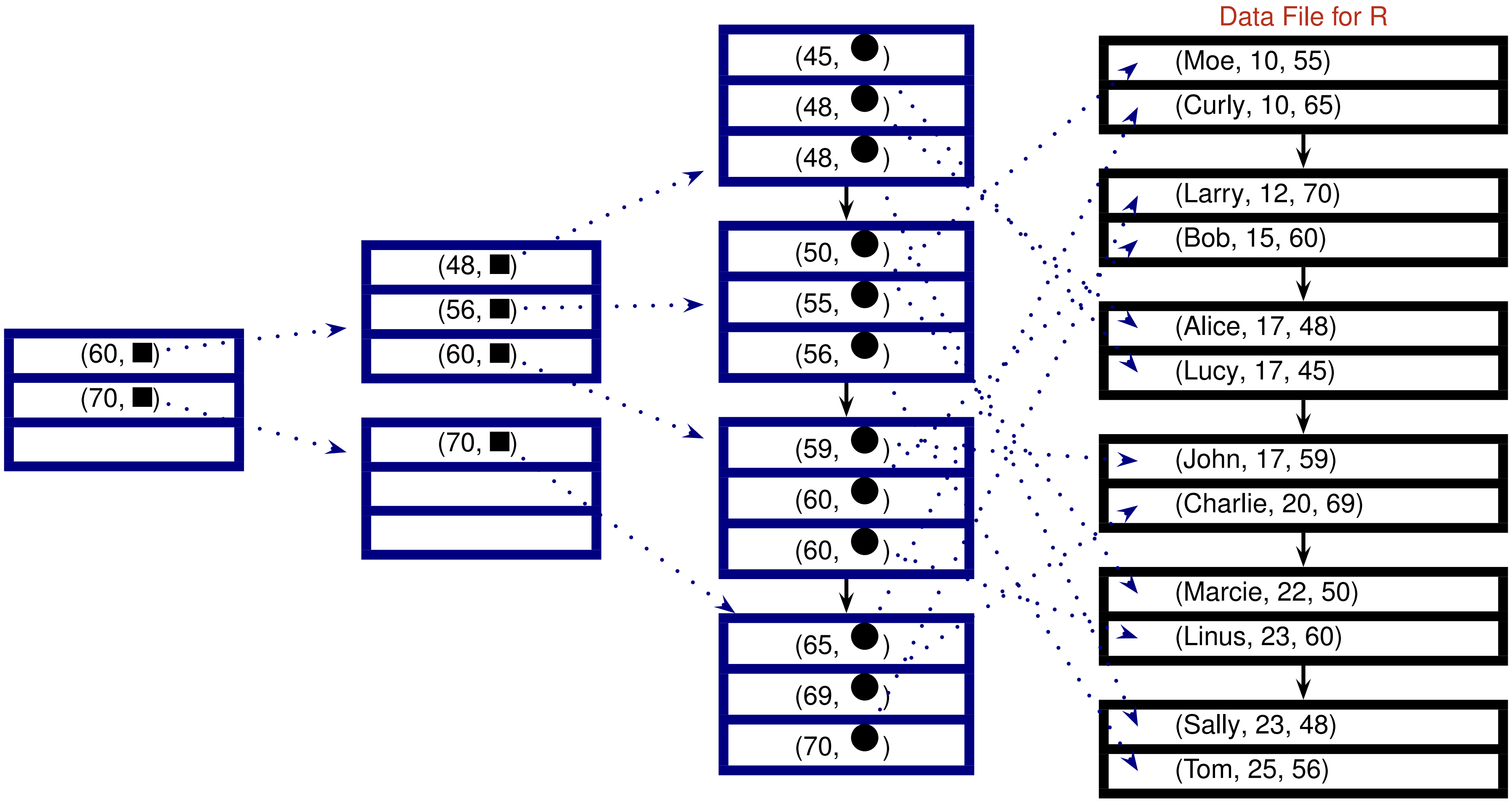
(48, ■) is the index entry for the first page of data entries; ■ denote a disk page address

Simple Index Example

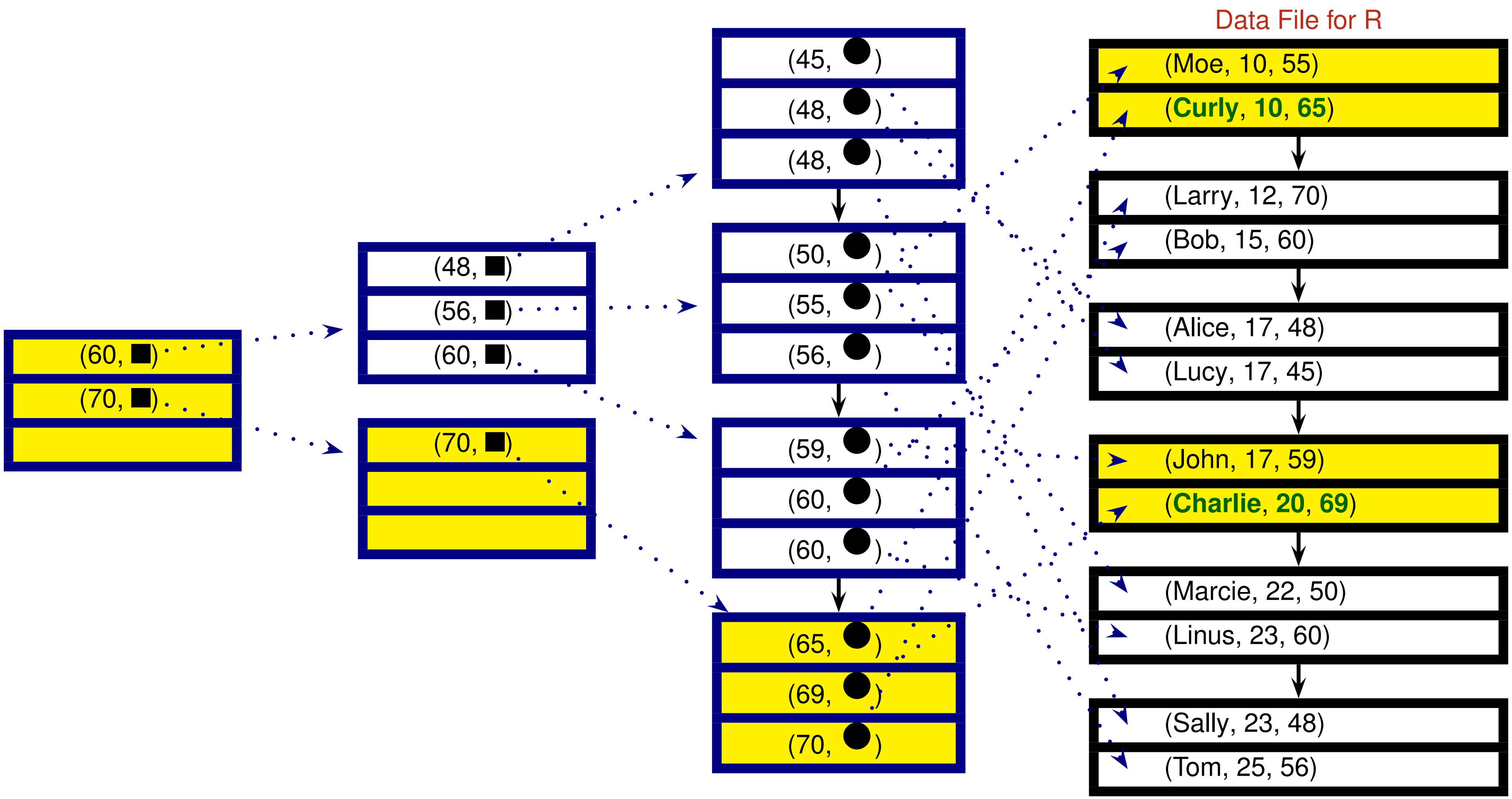


SELECT * FROM R WHERE weight BETWEEN 65 AND 69

Simple Index Example



Simple Index Example



SELECT * FROM R WHERE weight BETWEEN 65 AND 69

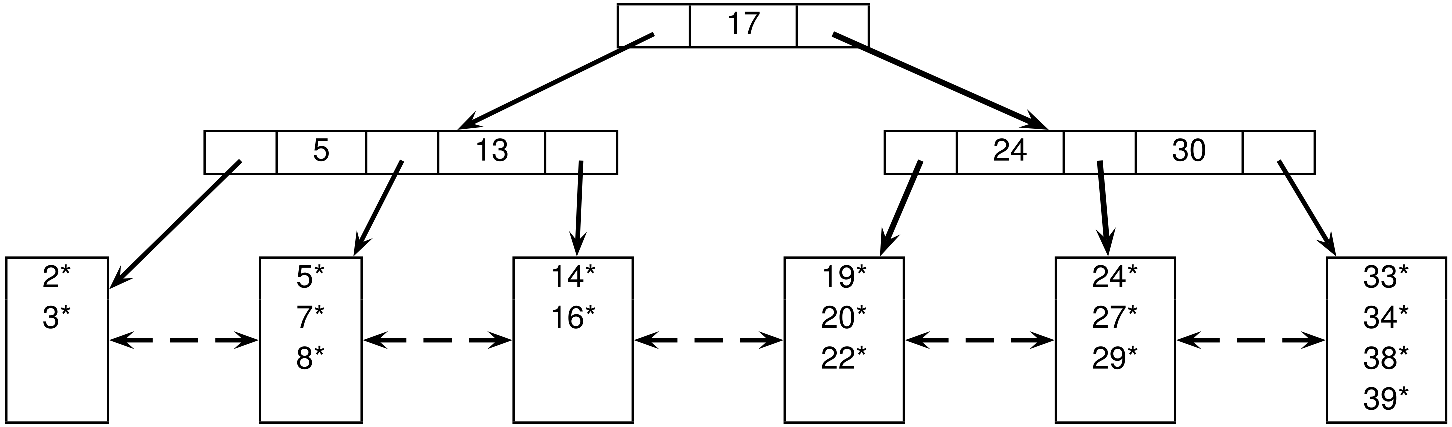
Index Types

- Two main types of indexes:
  - Tree-based index
    - ★ Based on sorting of search key values
    - ★ Examples: ISAM, B^+-tree
  - Hash-based index
    - ★ Data entries are accessed using hashing function
    - ★ Examples: static hashing, extendible hashing, linear hashing
- Things to consider when choosing an index:
  - Search performance
    - ★ equality search: $k = v$
    - ★ range search: $v_1 \leq k \leq v_2$
  - Storage overhead
  - Update performance

B^+-tree Index

Employee

name	deptNo	···
Alice	5	···
Bob	16	···
Charlie	19	···
Curly	39	···
Dave	38	···
Eve	14	···
Fred	33	···
Harry	2	···
John	34	···
Kate	8	···
Larry	27	···
Linus	24	···
Lucy	3	···
Marcie	22	···
Moe	29	···
Sally	20	···
Tom	7	···



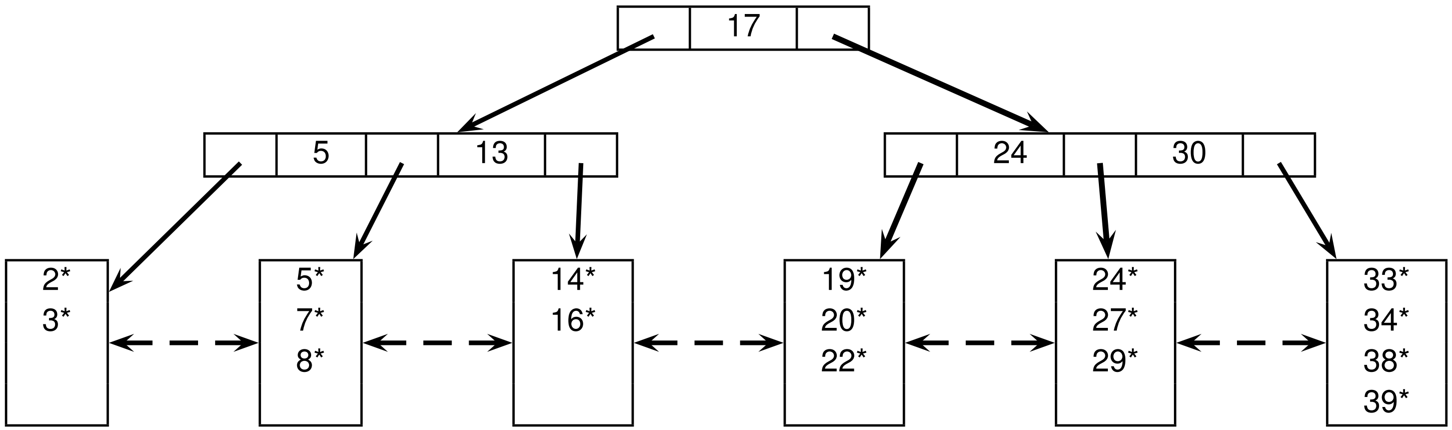
B^+-tree index on Employee.deptNo

- Leaf nodes store sorted data entries
  - k^* denote a data entry of the form (k, RID)
    - ★ k = search key value of corresponding data record
    - ★ RID = RID of corresponding data record
  - Leaf nodes are doubly-linked

B^+-tree Index

Employee

name	deptNo	···
Alice	5	···
Bob	16	···
Charlie	19	···
Curly	39	···
Dave	38	···
Eve	14	···
Fred	33	···
Harry	2	···
John	34	···
Kate	8	···
Larry	27	···
Linus	24	···
Lucy	3	···
Marcie	22	···
Moe	29	···
Sally	20	···
Tom	7	···

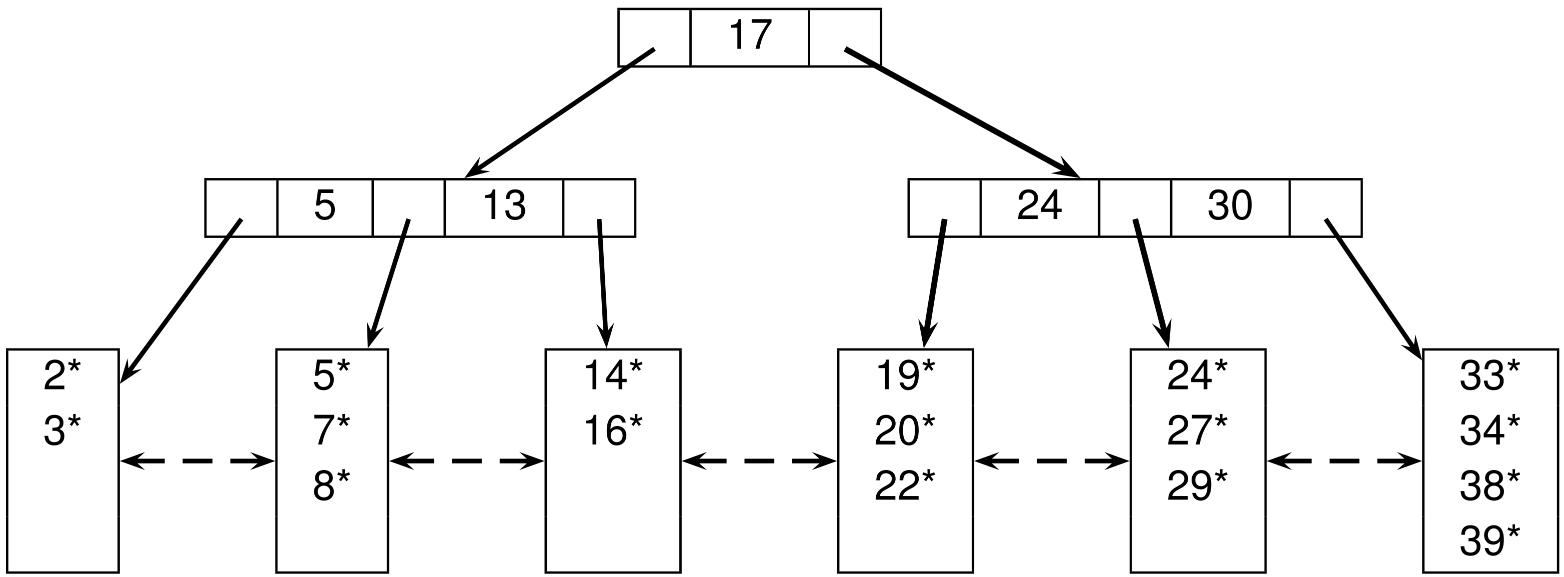


B^+-tree index on Employee.deptNo

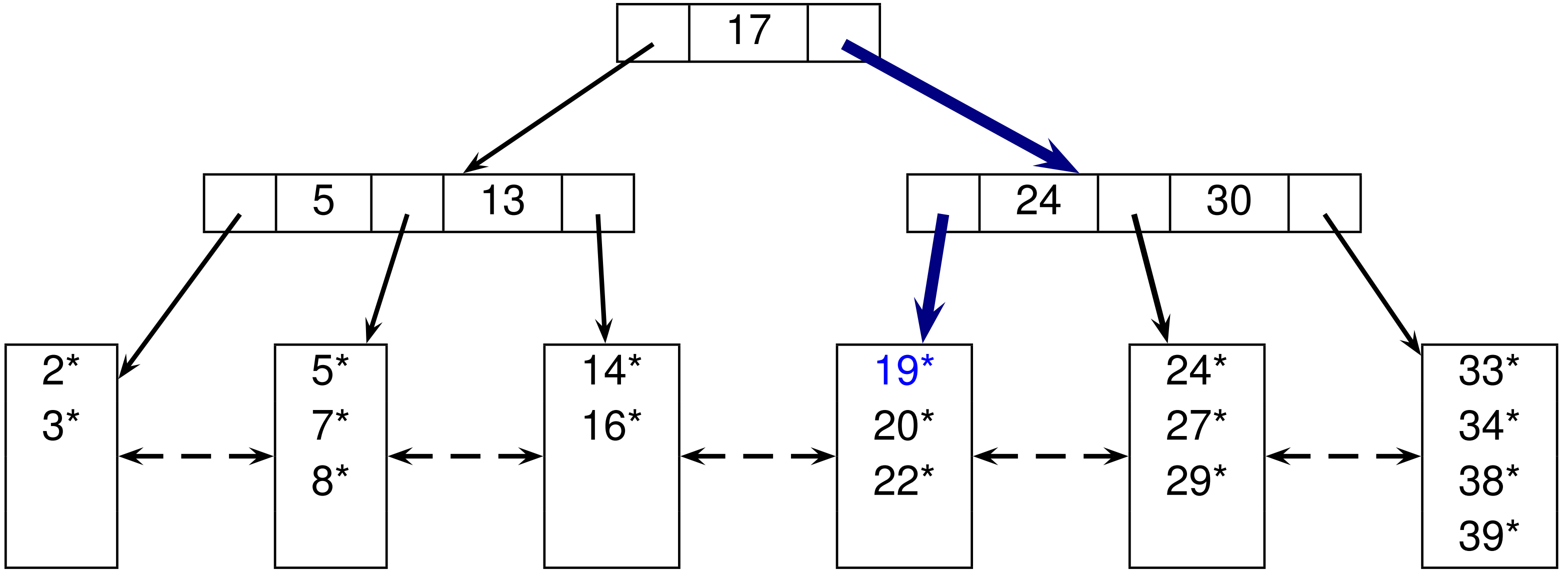
- Internal nodes store index entries of the form $(p_0, k_1, p_1, k_2, p_2, \cdots, p_n)$
  - $k_1 < k_2 < \cdots < k_n$
  - p_i = disk page address (root node of an index subtree T_i)
  - For each data entry k* in T_0, $k < k_1$
  - For each data entry k* in T_i ($i \in [1, n)$), $k \in [k_i, k_{i+1})$
  - For each data entry k* in T_n, $k \geq k_n$
- Each (k_i, p_i) is an index entry; k_i serves as a separator between the node contents pointed to by p_{i-1} & p_i

Properties of B^+-tree Index

- Dynamic index structure; adapts to data updates gracefully
- Height-balanced index structure
- Order of index tree, $d \in Z^+$
  1. Controls space utilization of index nodes
  2. Each non-root node contains m entries, where $m \in [d, 2d]$
  3. The root node contains m entries, where $m \in [1, 2d]$
- **Example:** B^+-tree with order = 2

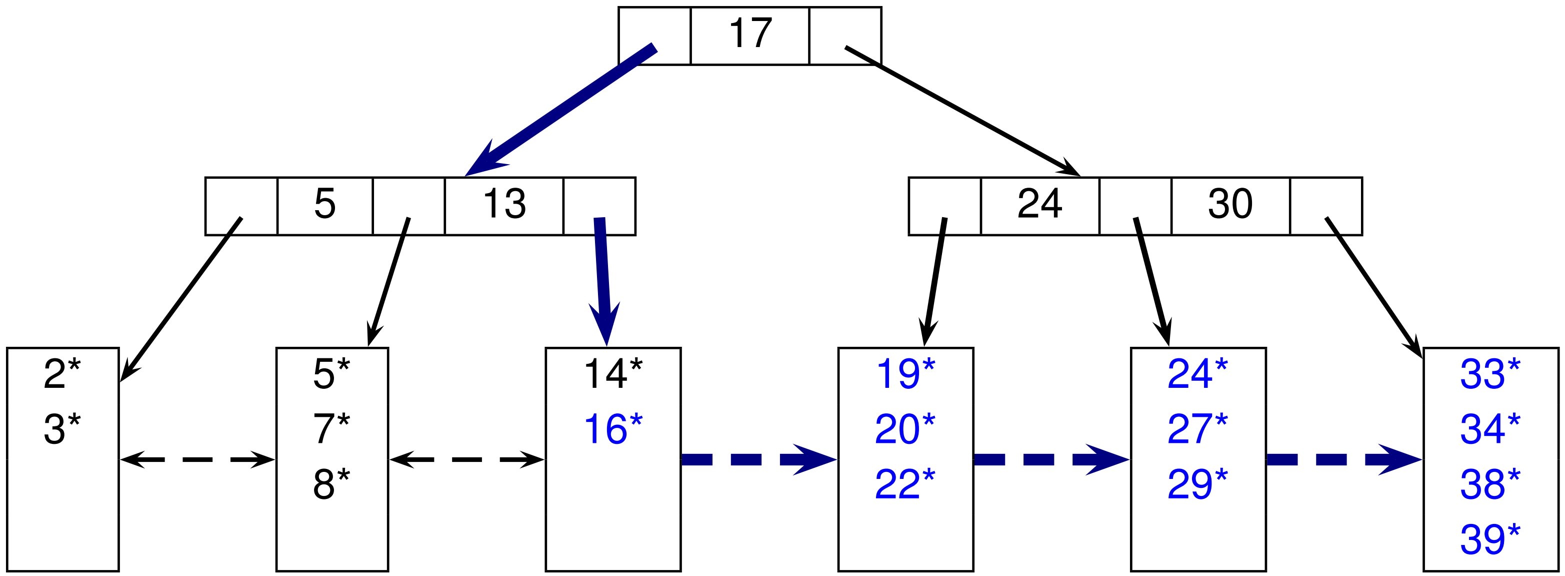


Equality Search ($k = 19$)

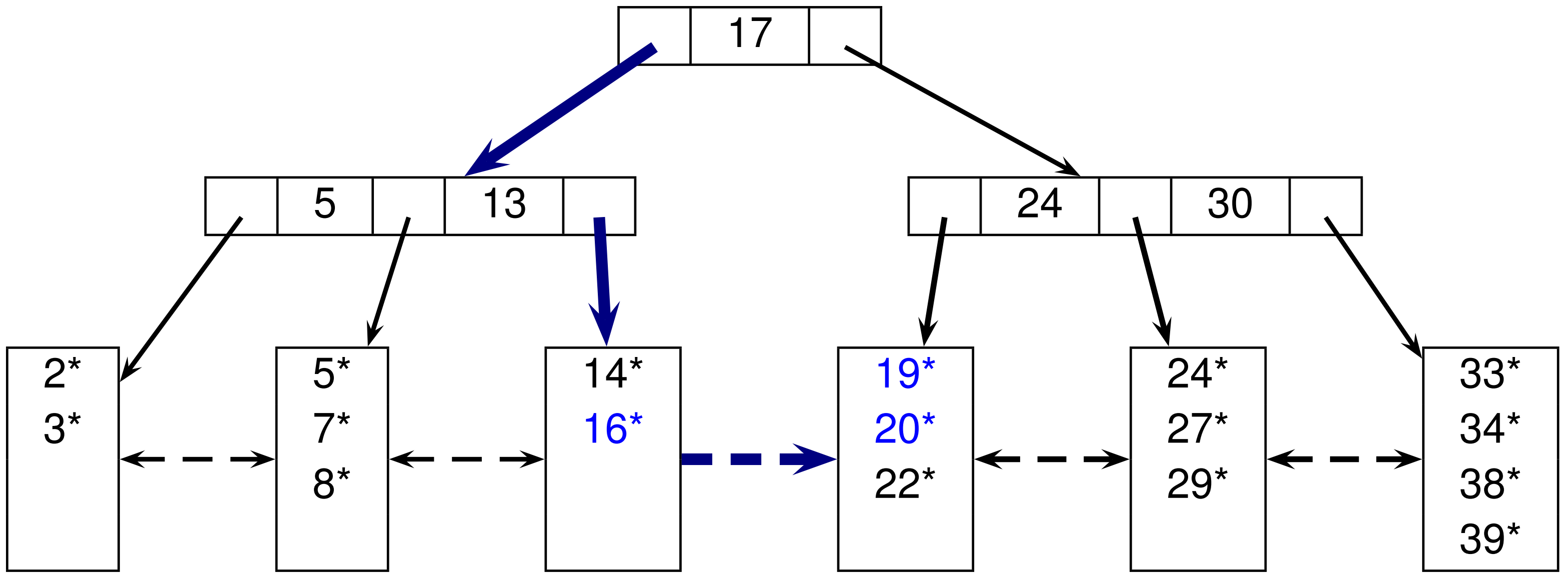


- At each internal node N, find the largest key k_i in N s.t. $k \geq k_i$
  - If k_i exists, then search subtree at p_i
  - Otherwise, search subtree at p_0

Range Search ($k \geq 15$)



Range Search ($15 \leq k \leq 21$)



Formats of Data Entries

- Three different formats for data entries:

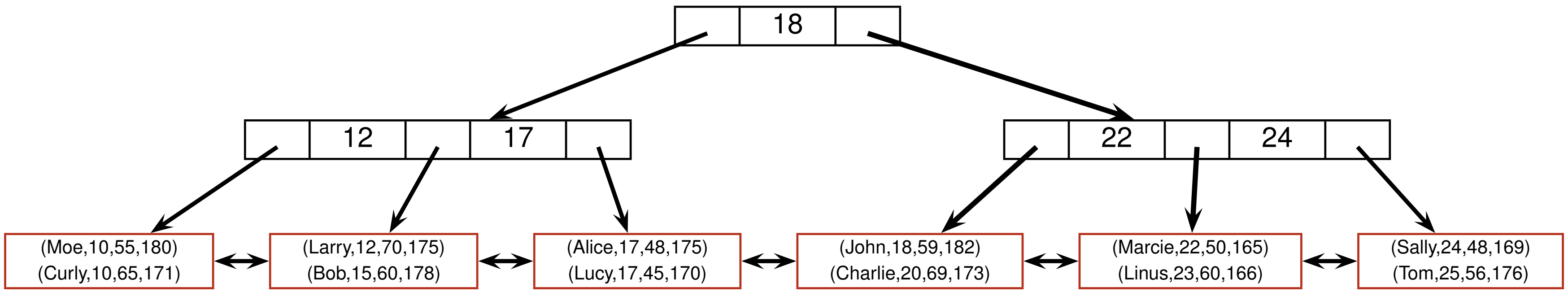
  Format 1: k* is an actual data record (with search key value k)

  Format 2: k* is of the form (k, rid), where *rid* is the record identifier of a data record with search key value *k*

  Format 3: k* is of the form (k, rid-list), where *rid-list* is a list of record identifiers of data records with search key value *k*

- So far, our examples assume Format 2.

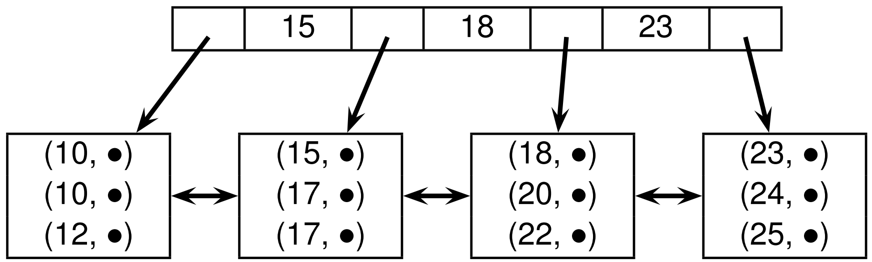
Formats of Data Entries: Example



B^+-tree index on R.age (Format 1)

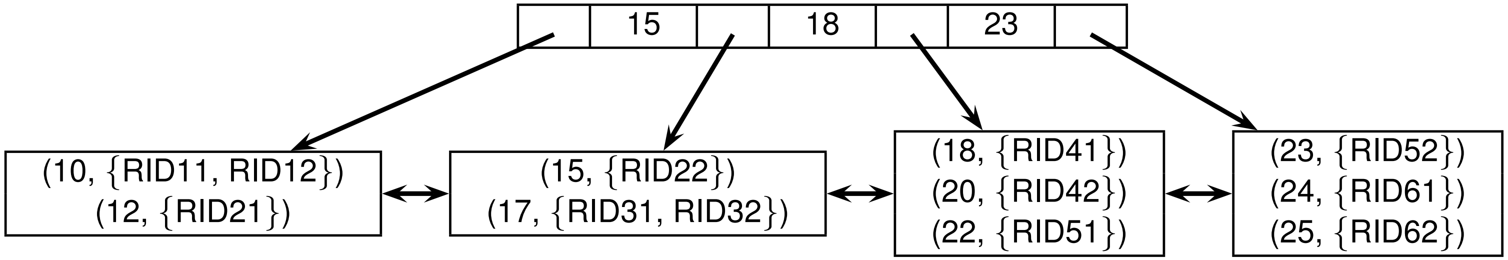
Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176



B^+-tree index on R.age (Format 2)

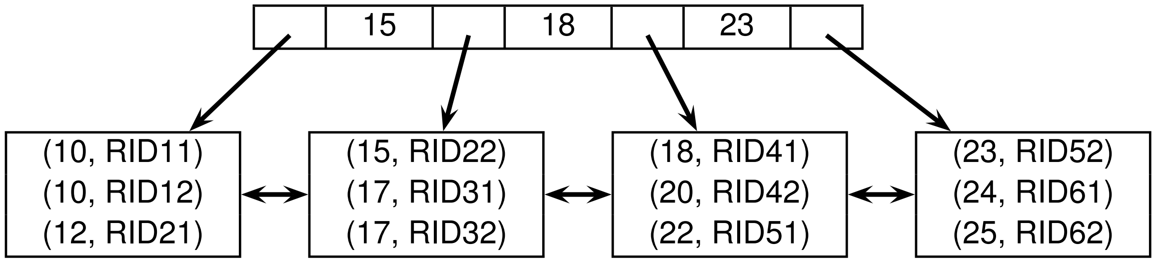
Format 2 vs Format 3 Index



Index on R.age (format 3)

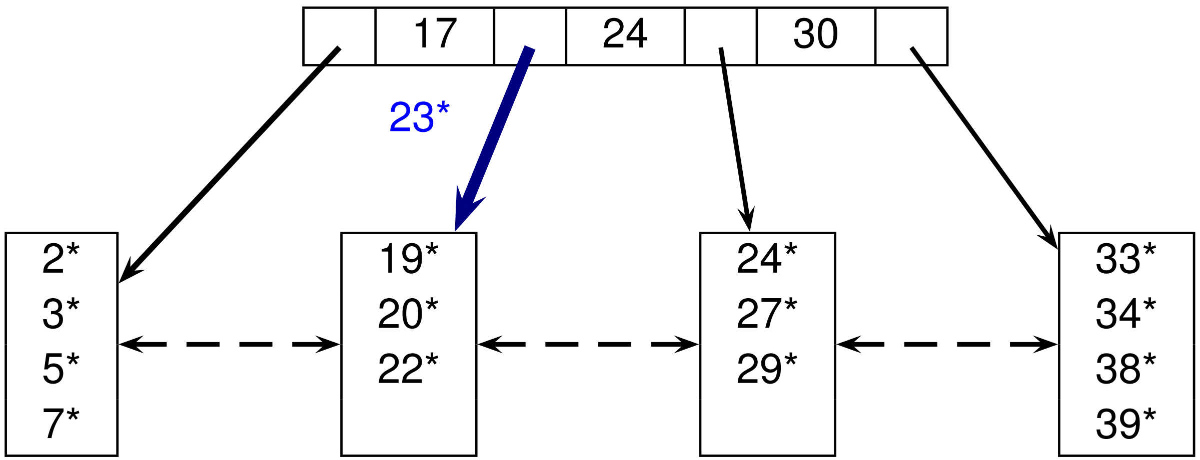
Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176

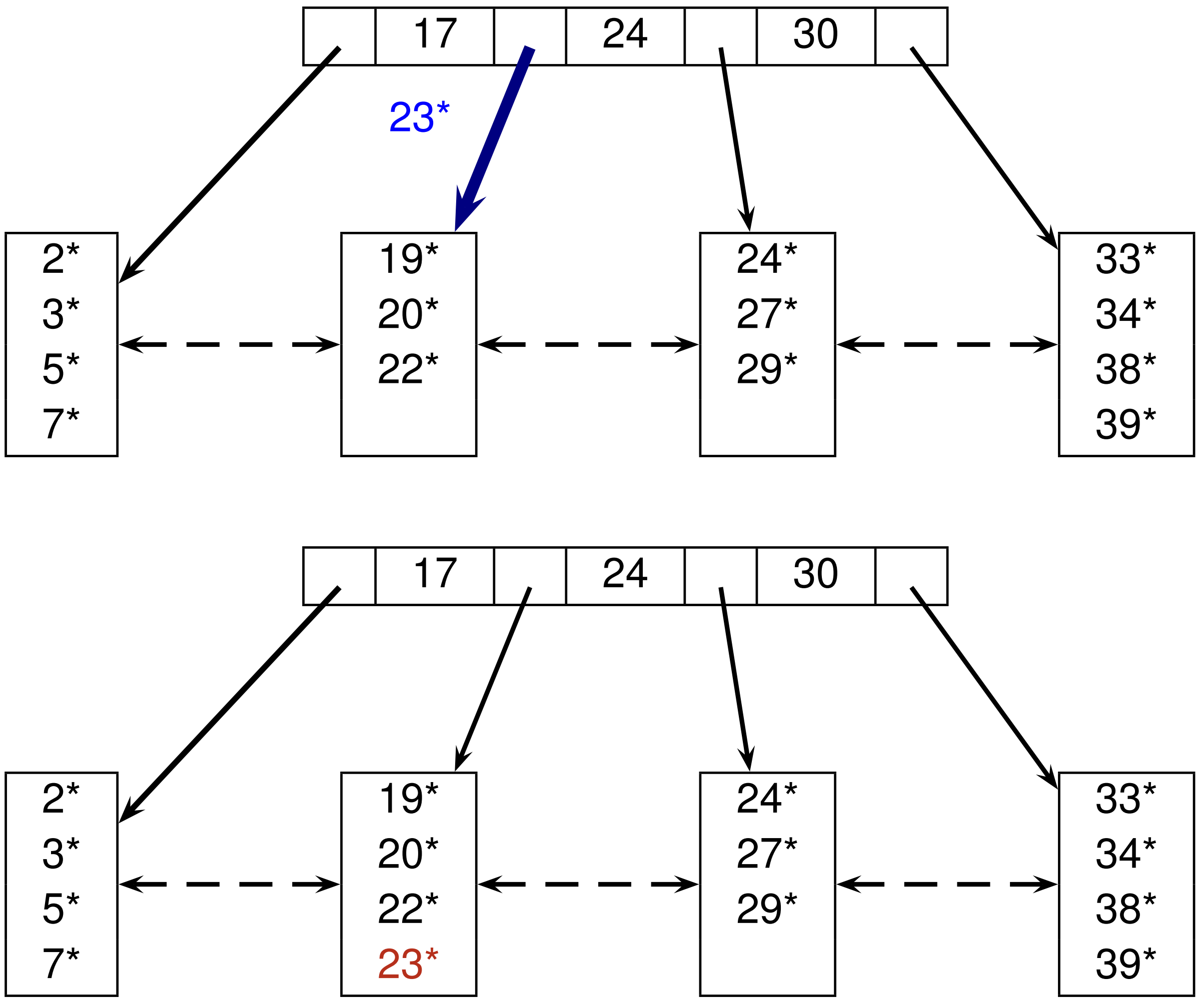


Index on R.age (format 2)

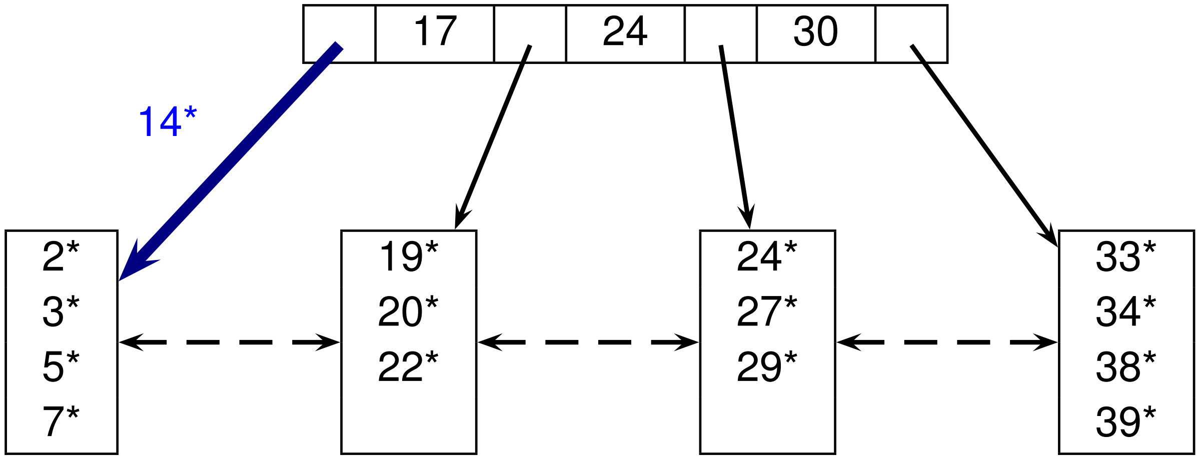
Inserting 23* (Simple Case)



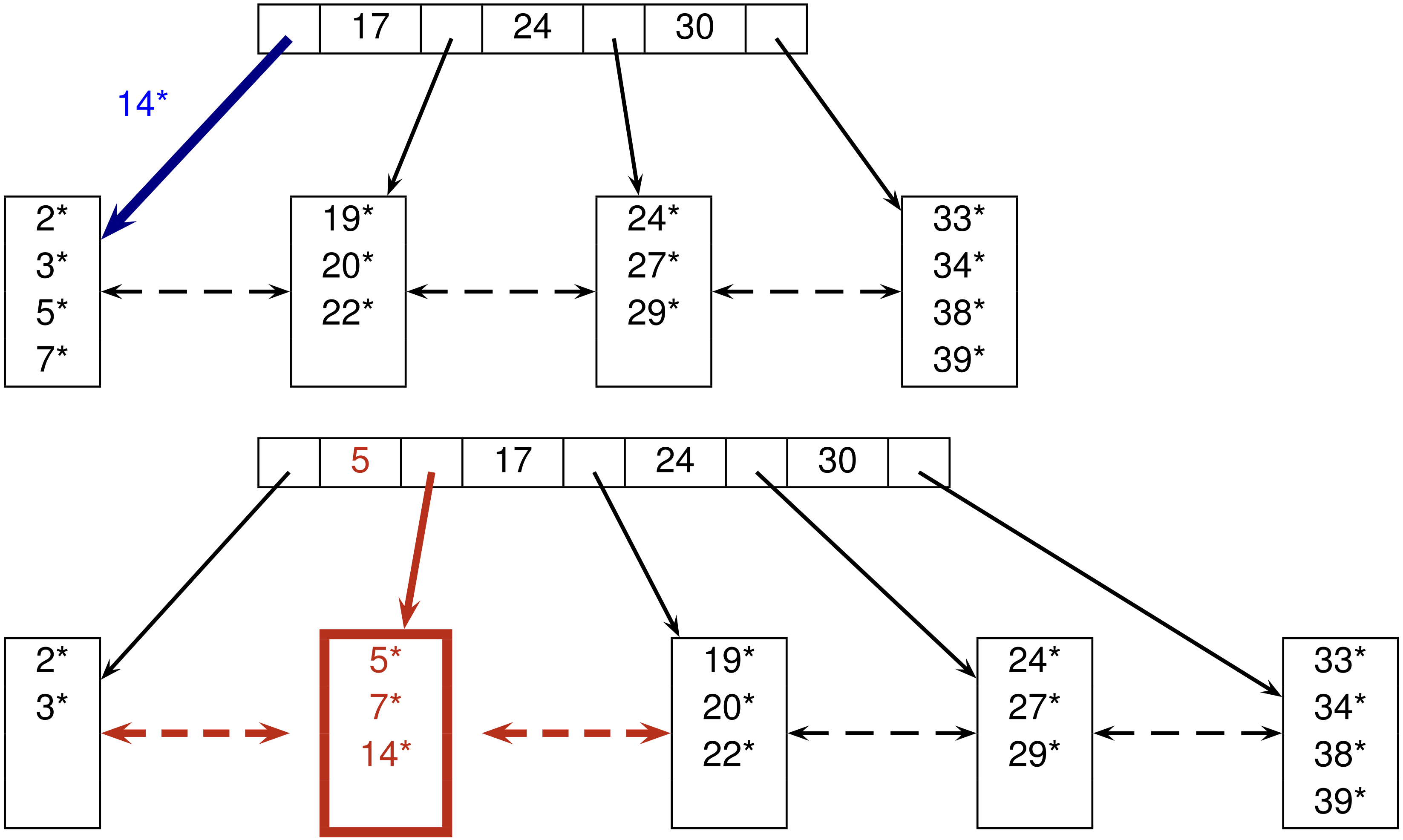
Inserting 23* (Simple Case)



Inserting 14* (Splitting of overflowed node)

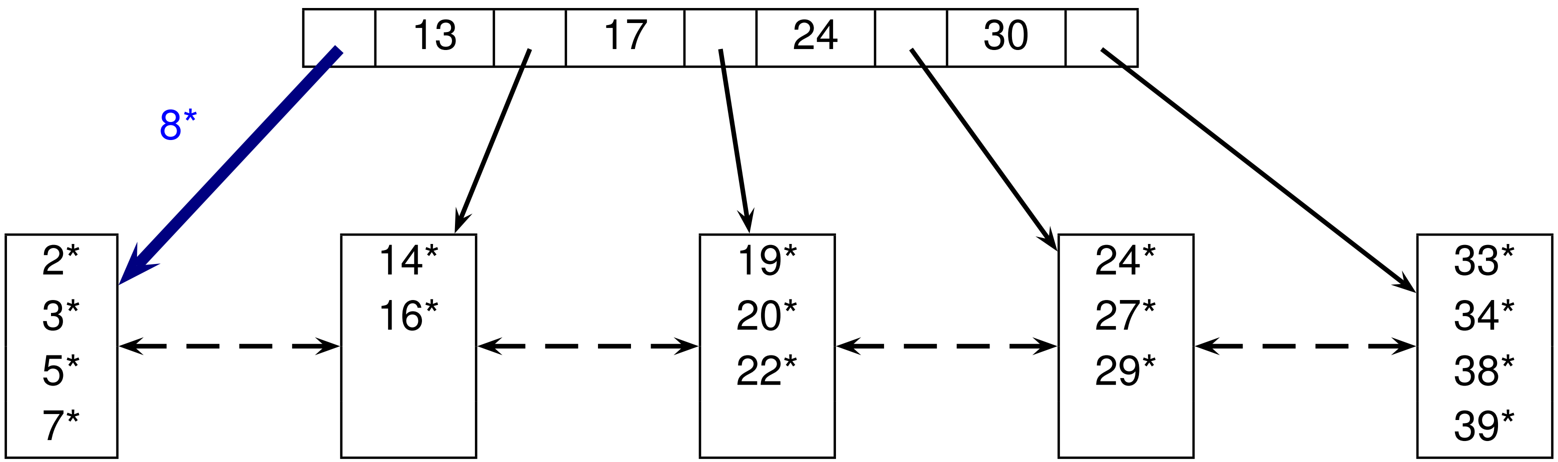


Inserting 14* (Splitting of overflowed node)

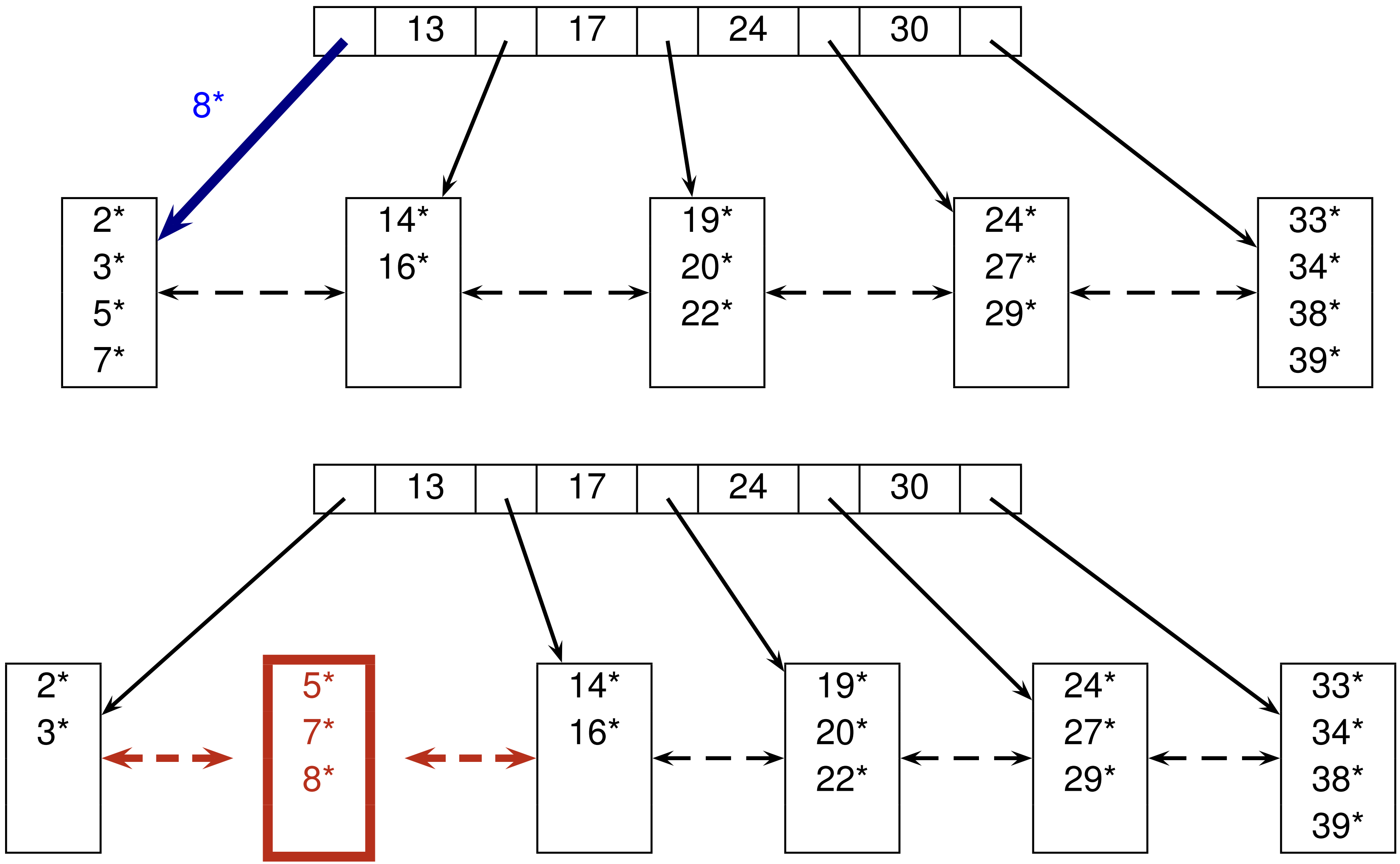


- Split overflowed leaf node by distributing **d+1** entries to new leaf node
- Create a new index entry using the smallest key in new leaf node
- Insert new index entry into parent node of overflowed node

Inserting 8* (Propagation of node splits)

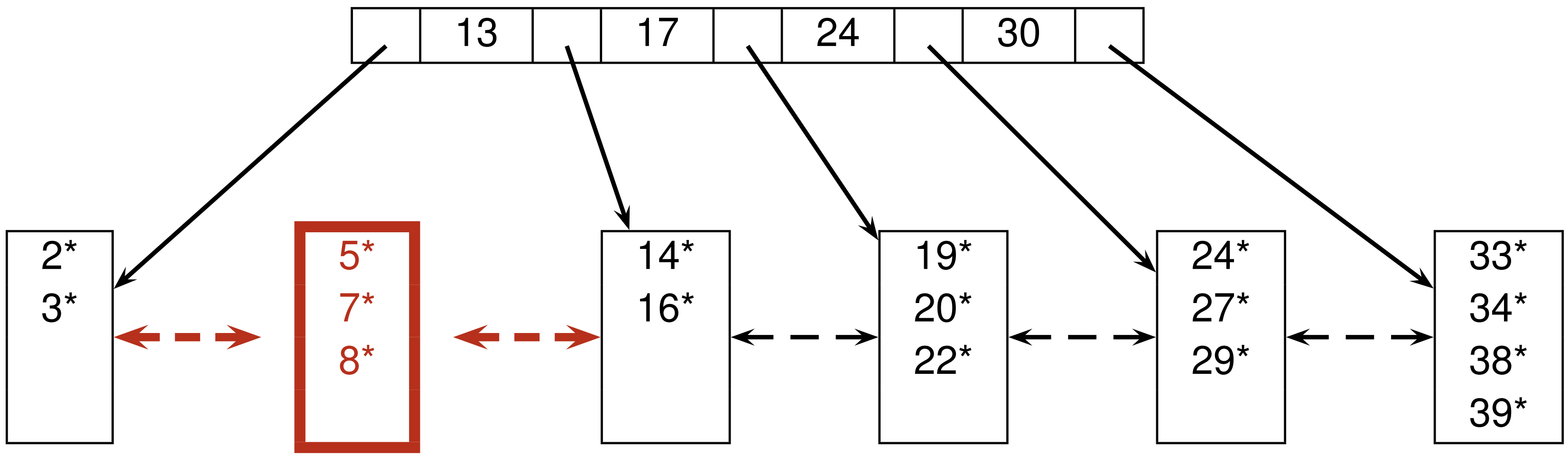


Inserting 8* (Propagation of node splits)

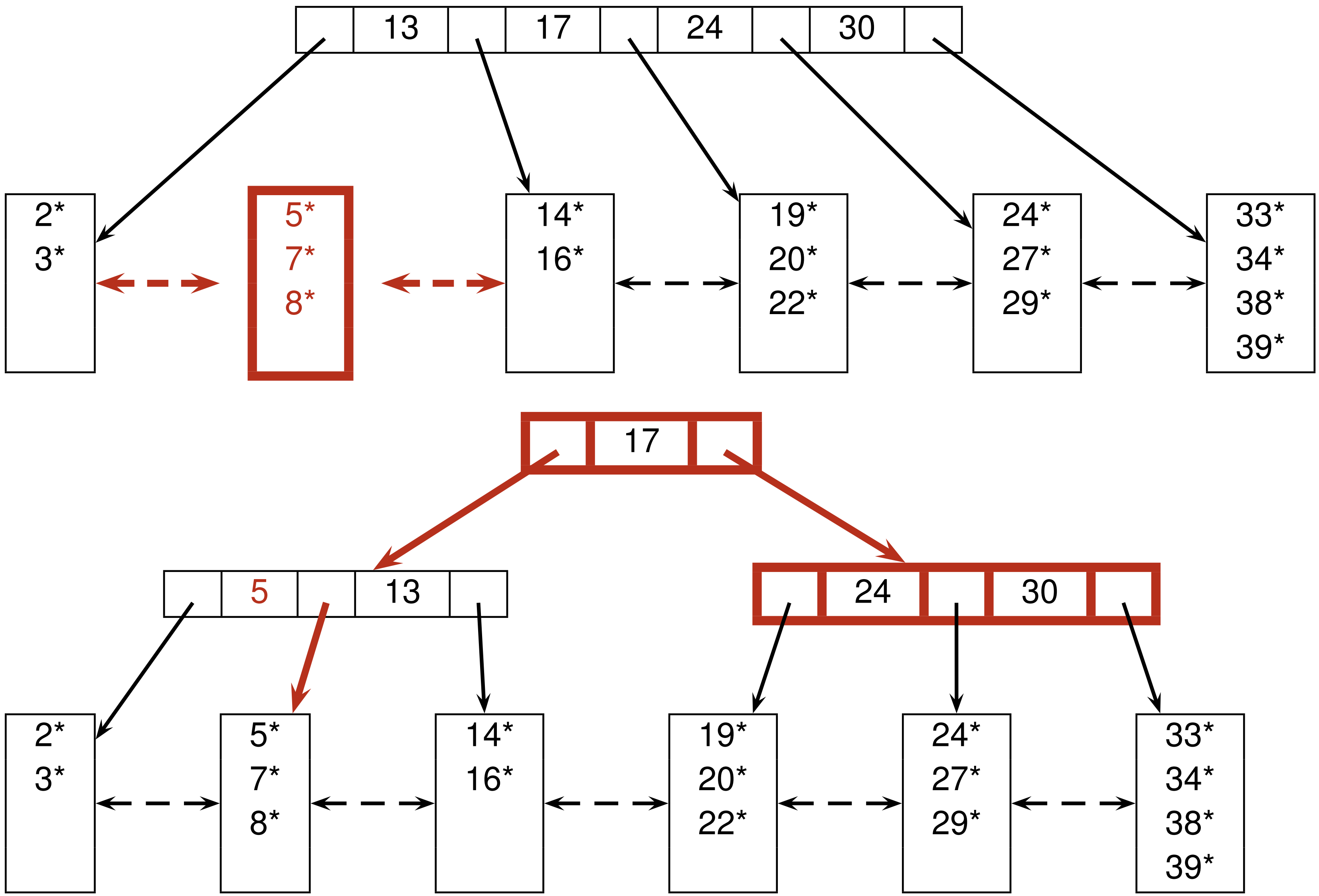


- Node splits can be propagated to ancestor internal nodes

Inserting 8* (Propagation of node splits)

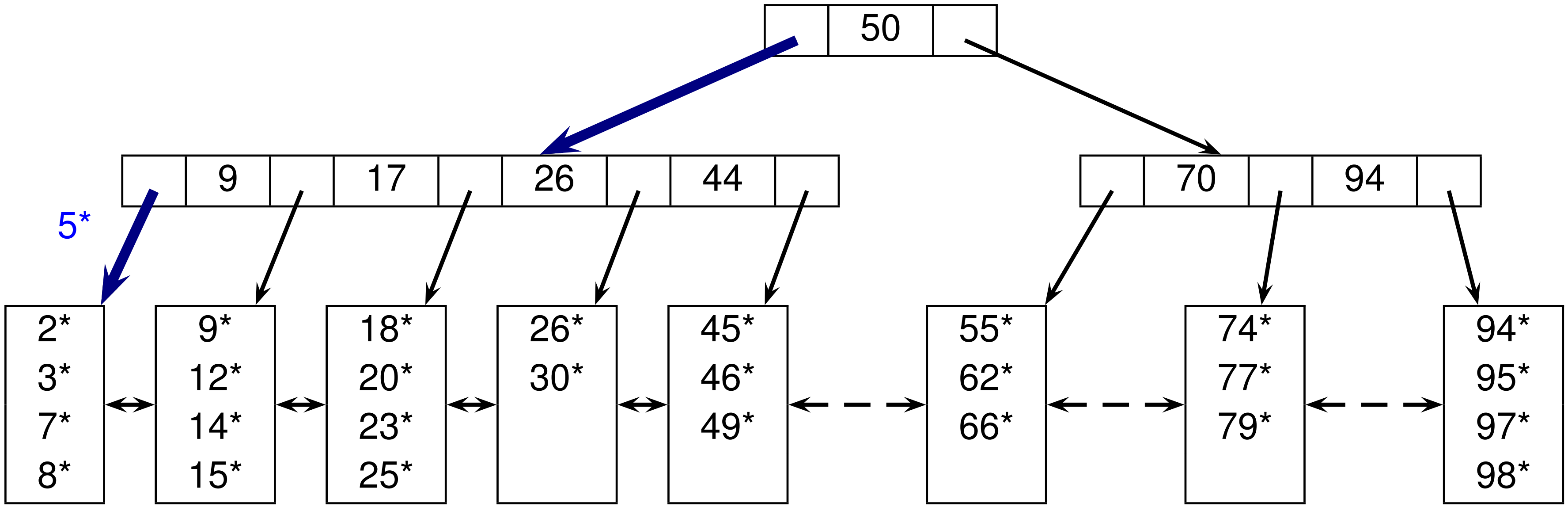


Inserting 8* (Propagation of node splits)

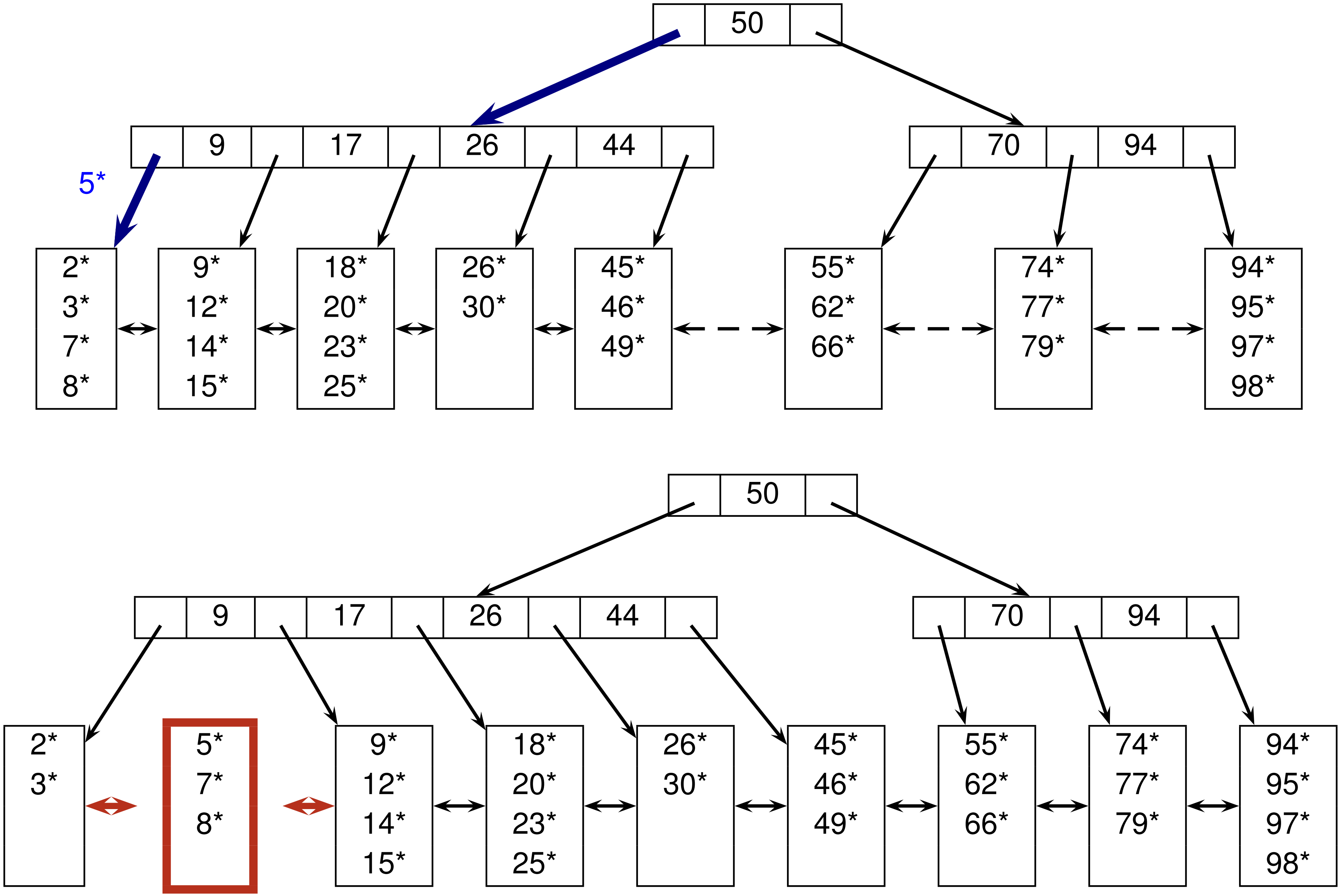


- When splitting an internal node, the middle key is pushed up to parent node

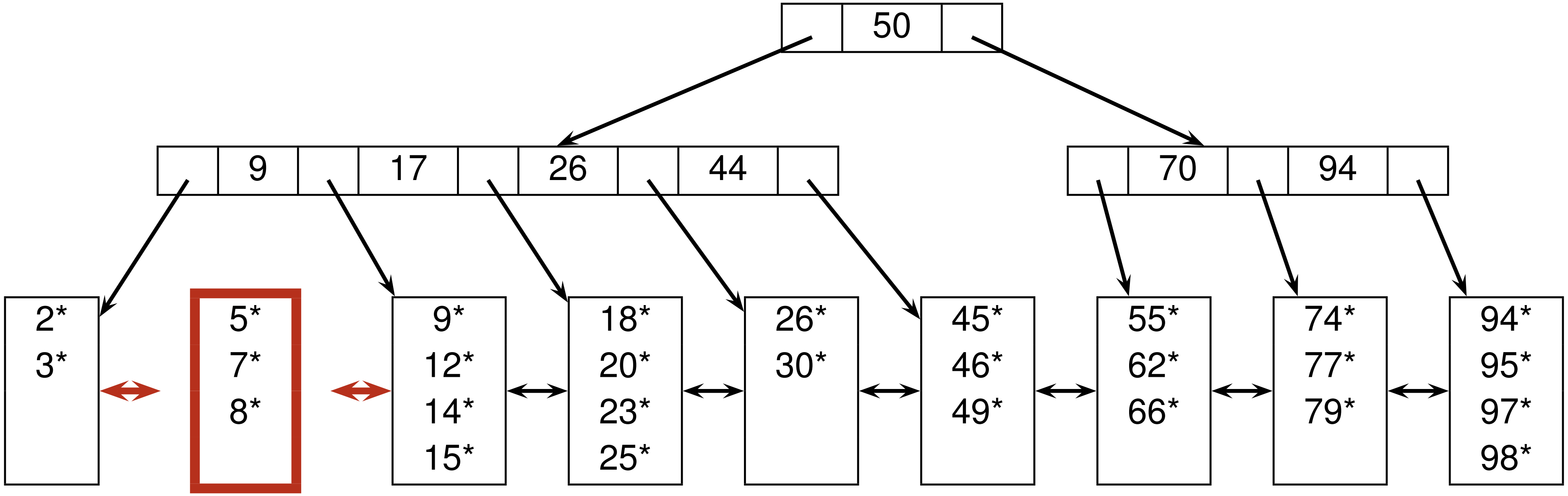
Inserting 5* (Propagation of node splits)



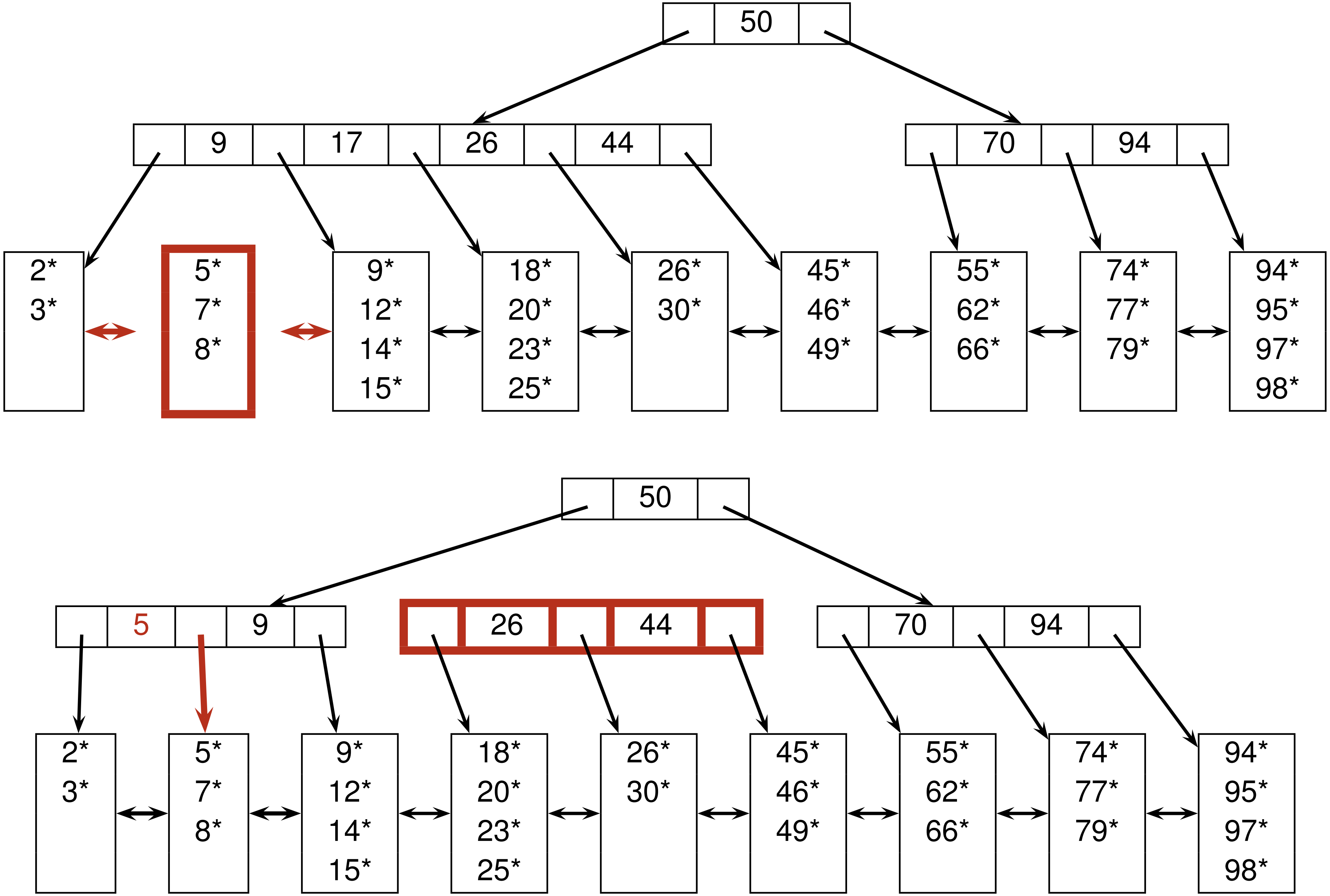
Inserting 5* (Propagation of node splits)



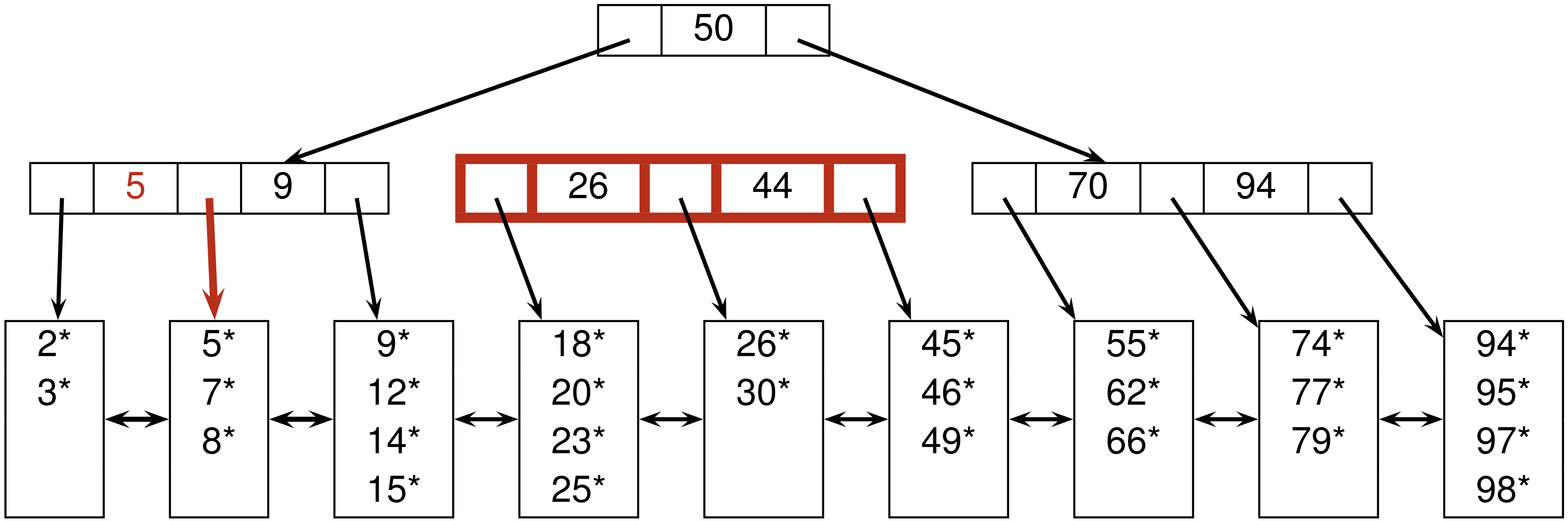
Inserting 5* (Propagation of node splits)



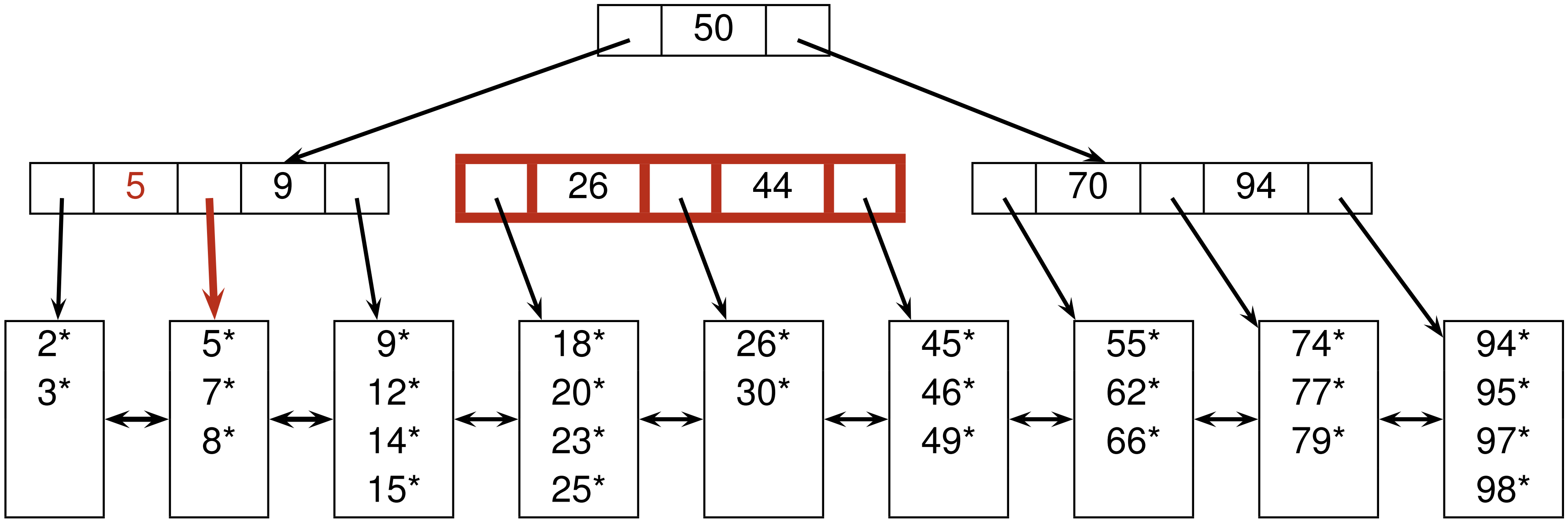
Inserting 5* (Propagation of node splits)

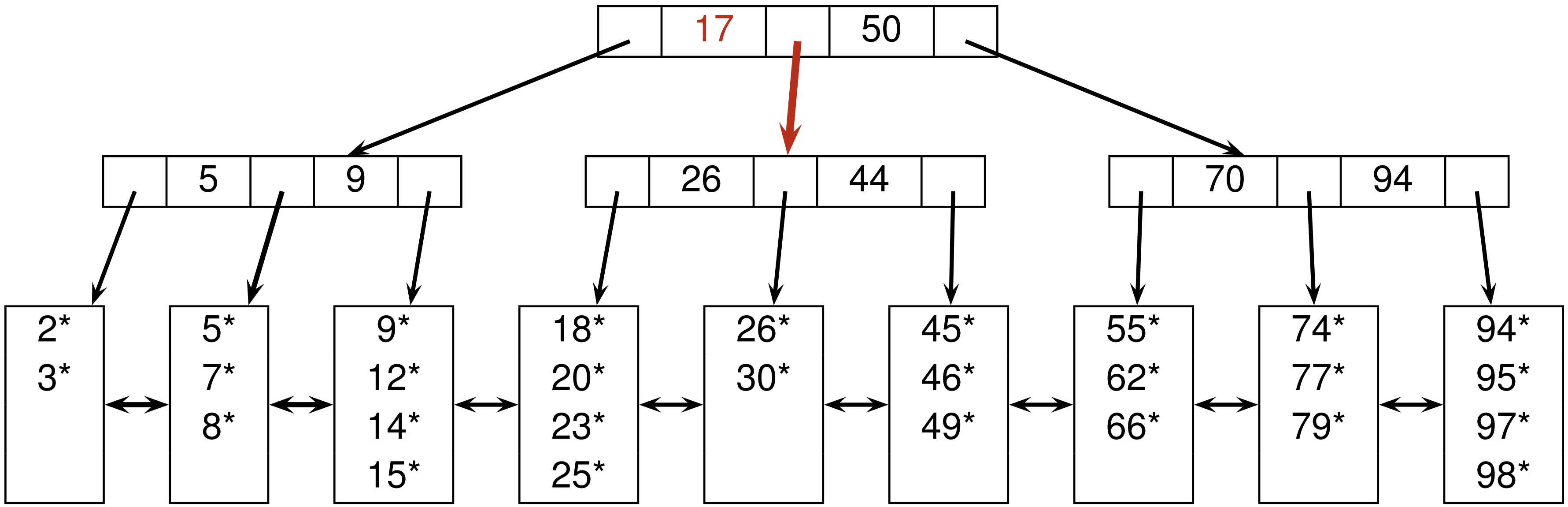


Inserting 5* (Propagation of node splits)



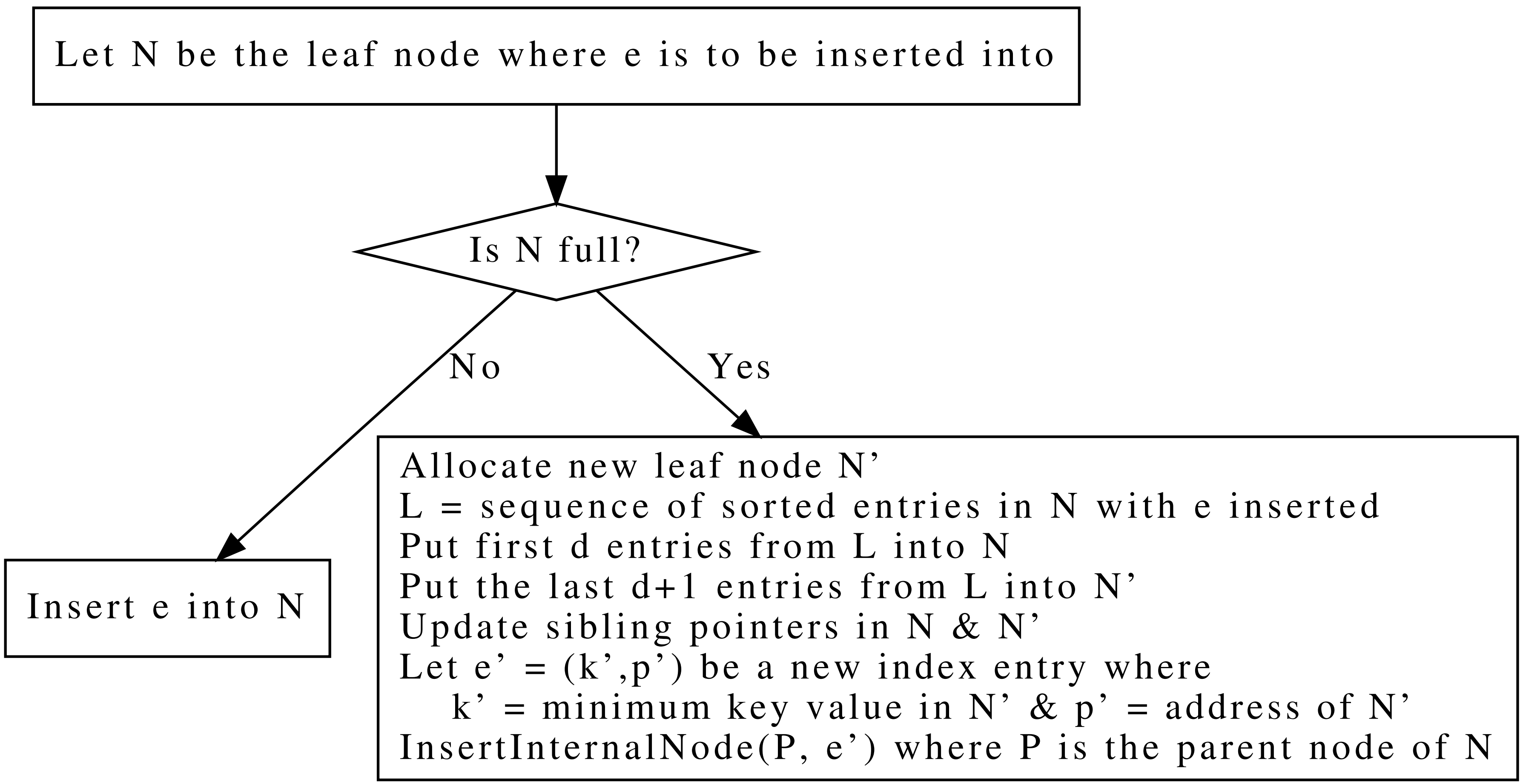
Inserting 5* (Propagation of node splits)





B^+-tree: Insertion Algorithm

d = order of index, **e** = new data entry to be inserted into index

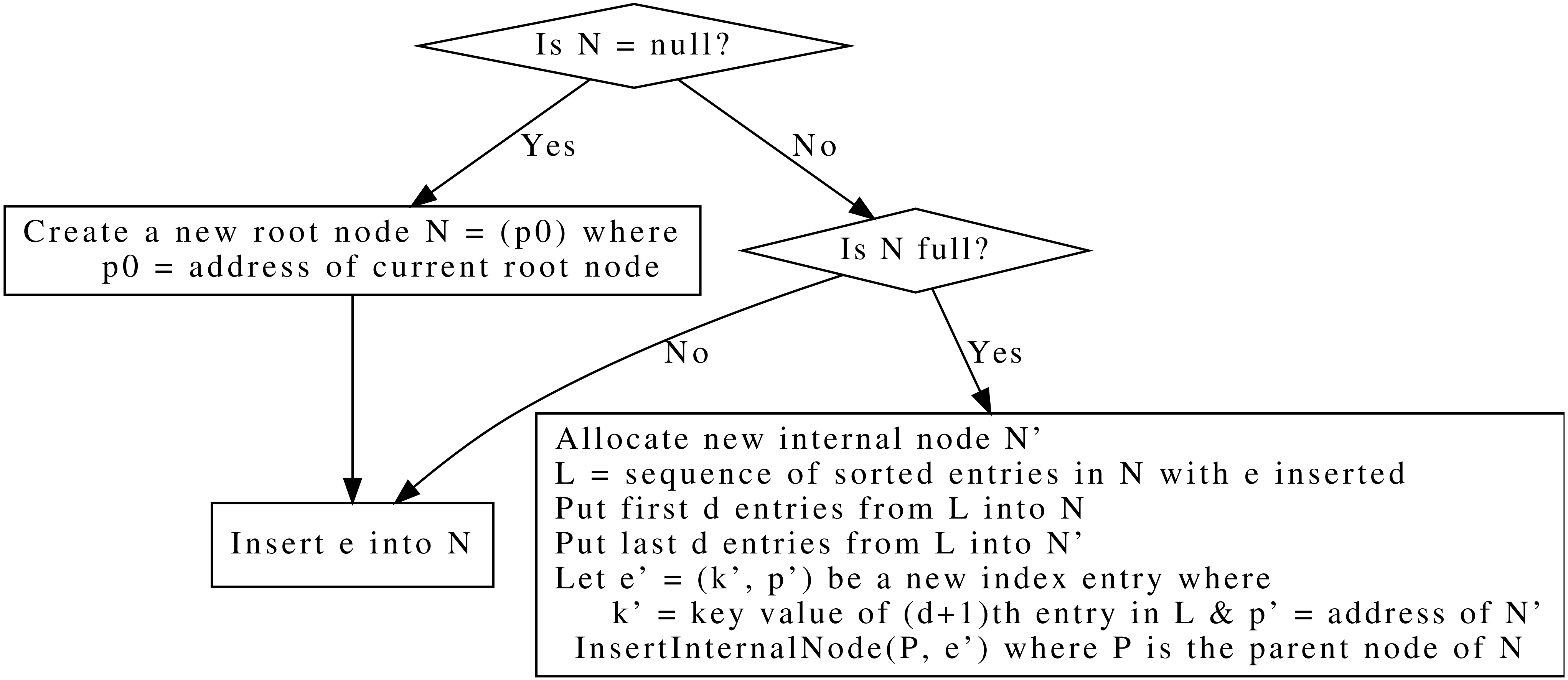


Note: The parent node of a root node is null

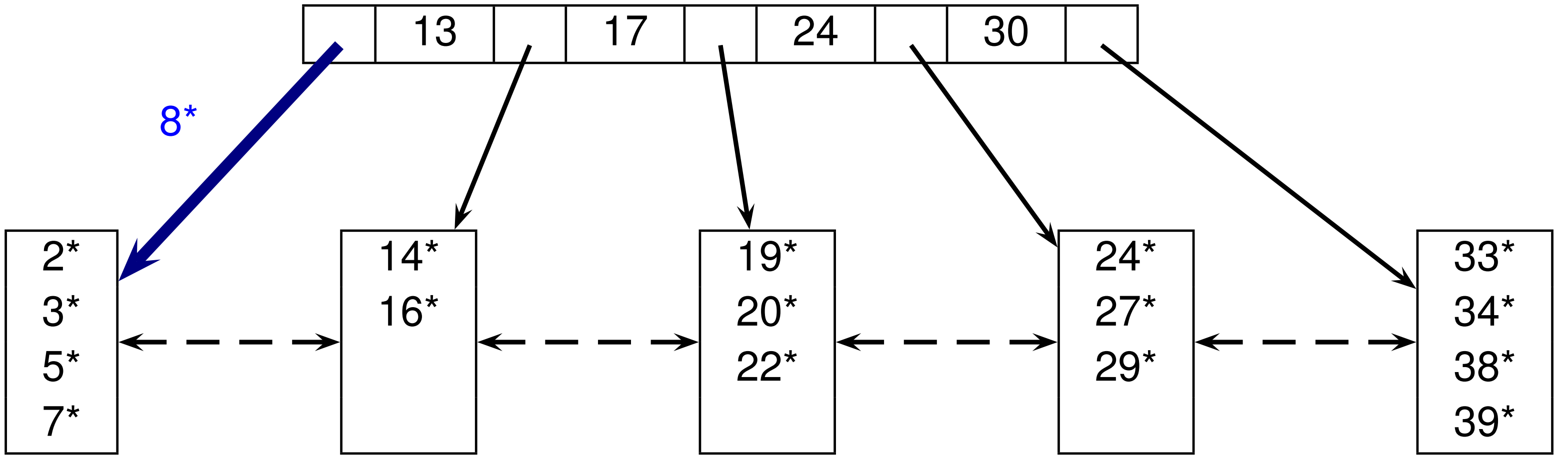
InsertInternalNode(N, e)

Insert index entry **e** into internal index node **N**

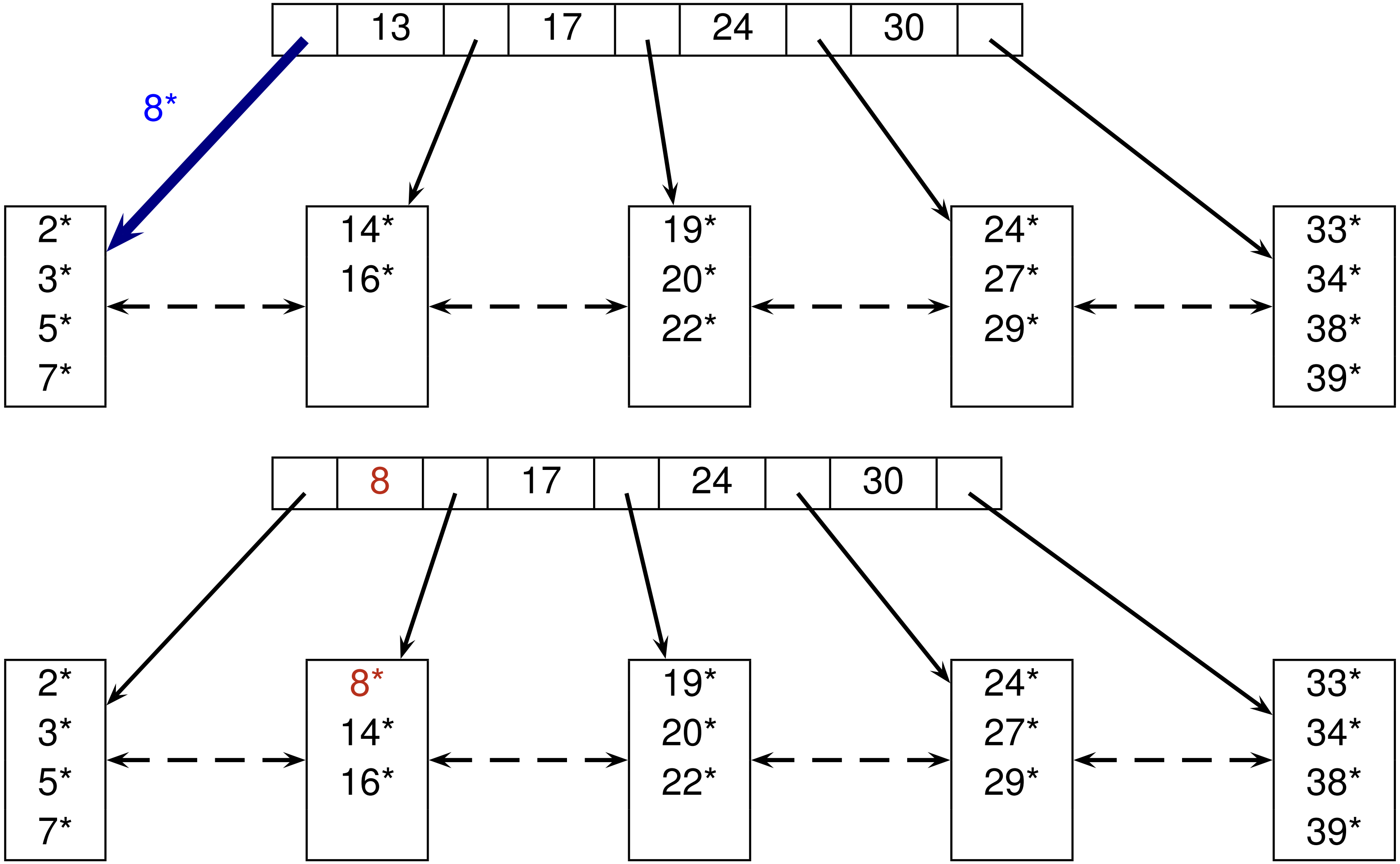
d = order of index



Inserting 8* (Redistribution of data entries)



Inserting 8* (Redistribution of data entries)

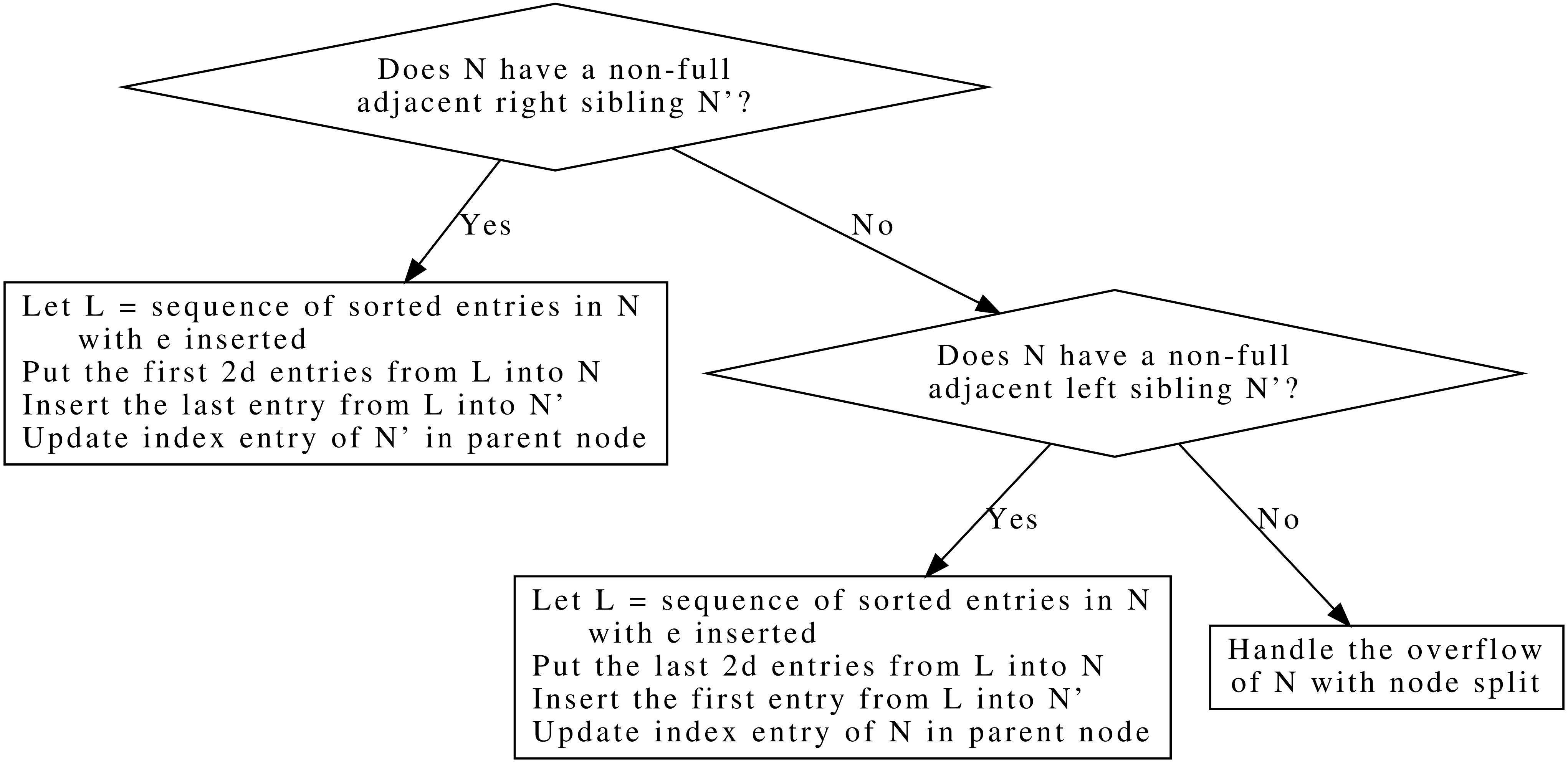


- A node split can sometimes be avoided by distributing entries from overflowed node to a non-full adjacent sibling node
- Two nodes at the same level are sibling nodes if they have the same parent node

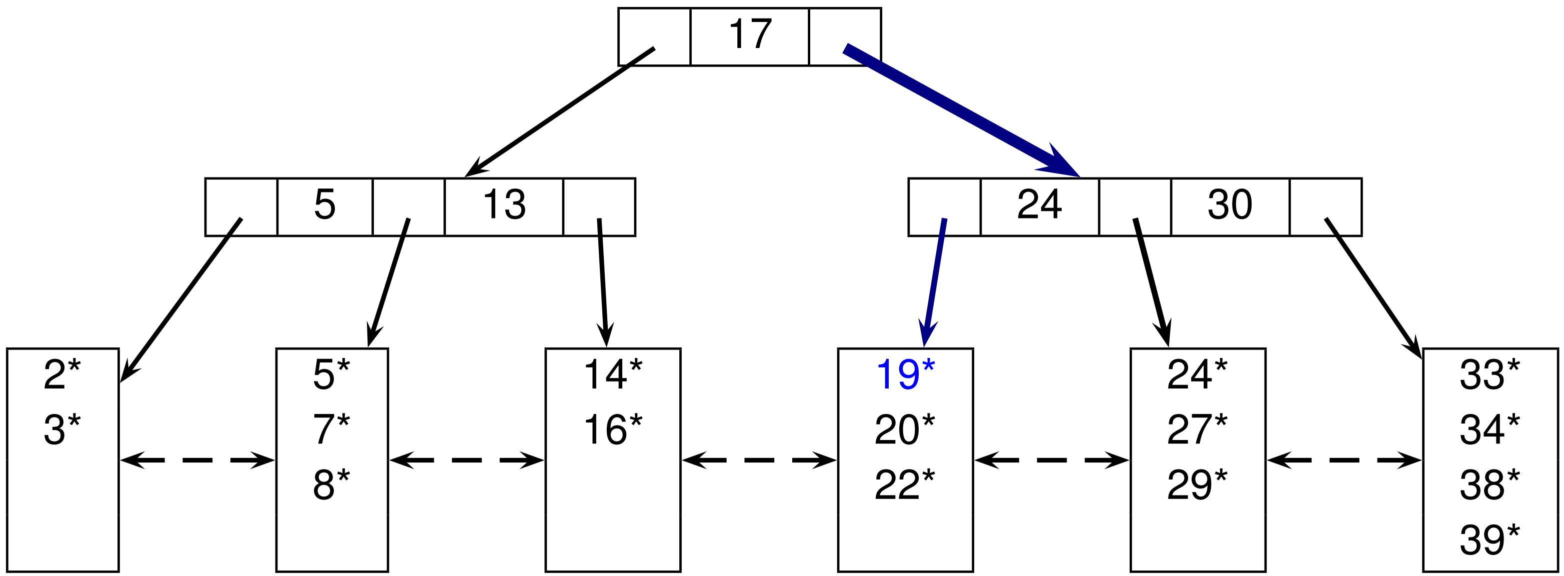
Redistribution of data entries (in leaf nodes)

e = new data entry to be inserted into a full leaf node **N**

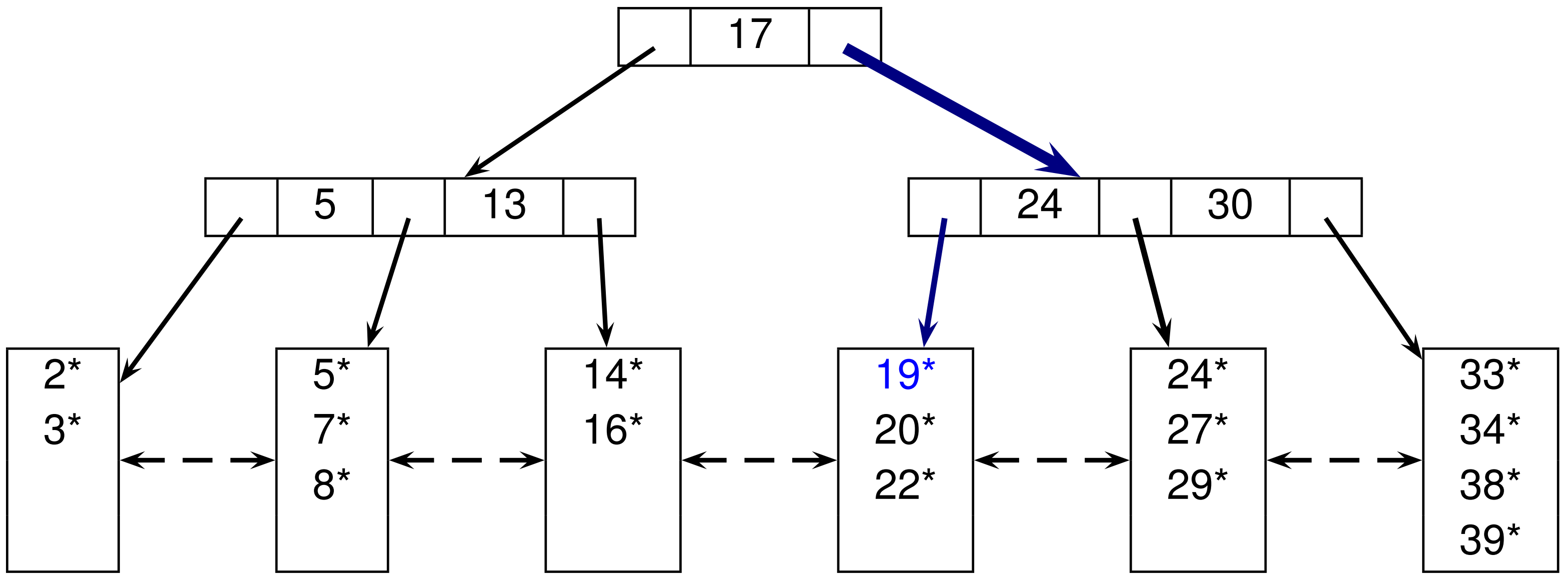
d = order of index

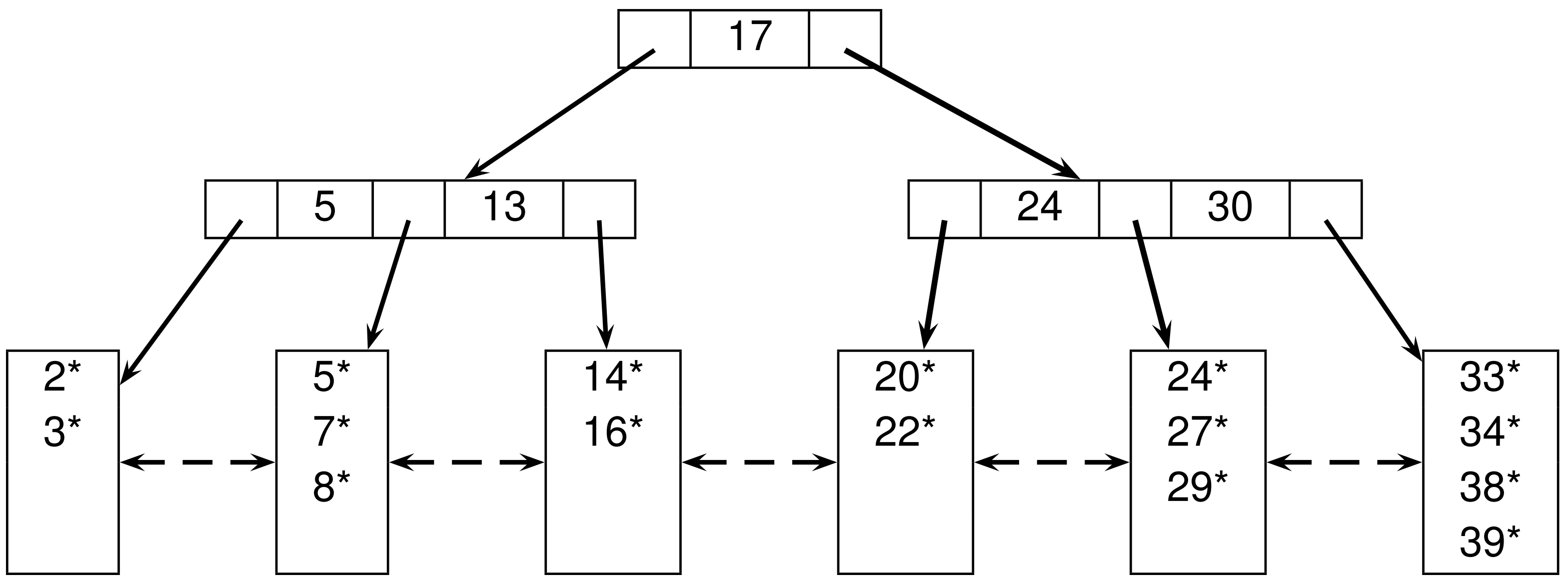


Deleting 19* (Simple Case)

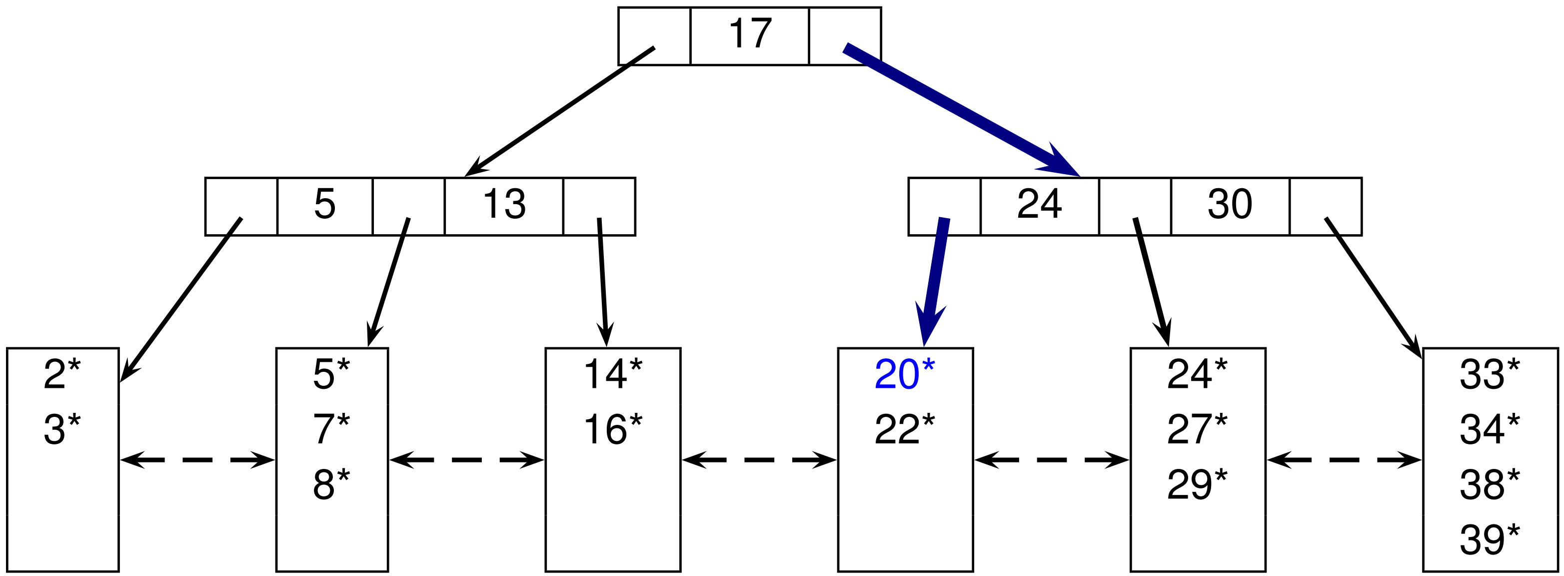


Deleting 19* (Simple Case)

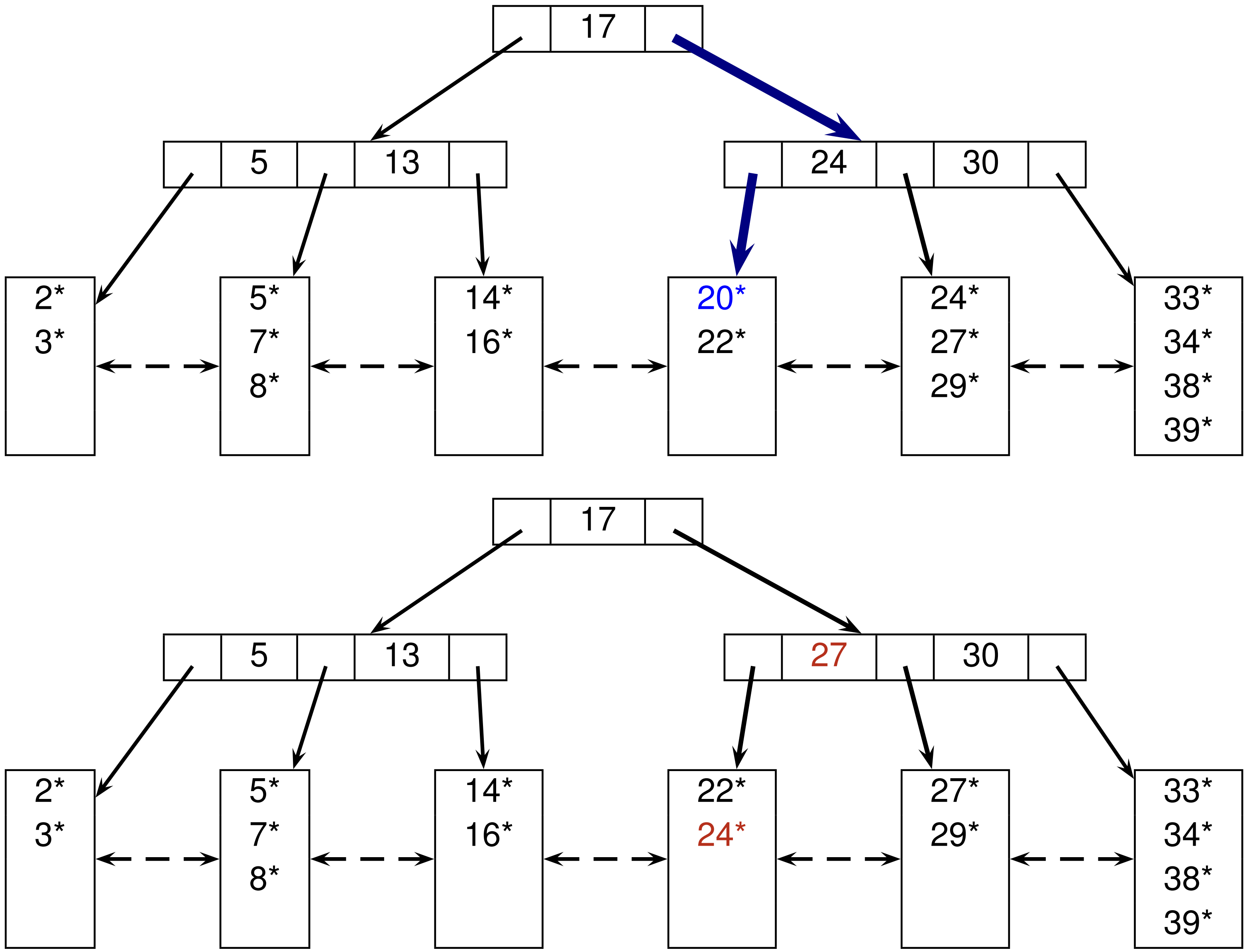




Deleting 20* (Redistribution of leaf entries)

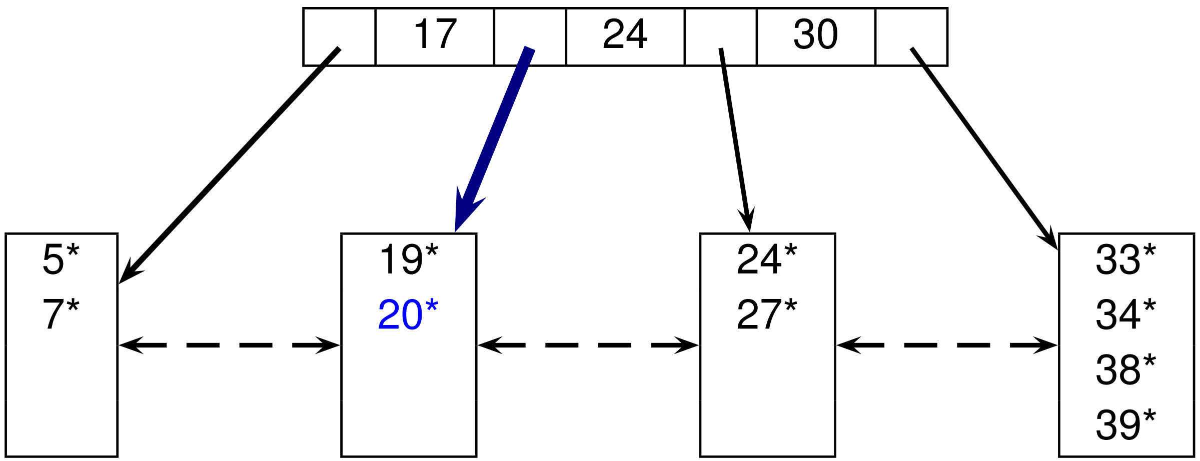


Deleting 20* (Redistribution of leaf entries)

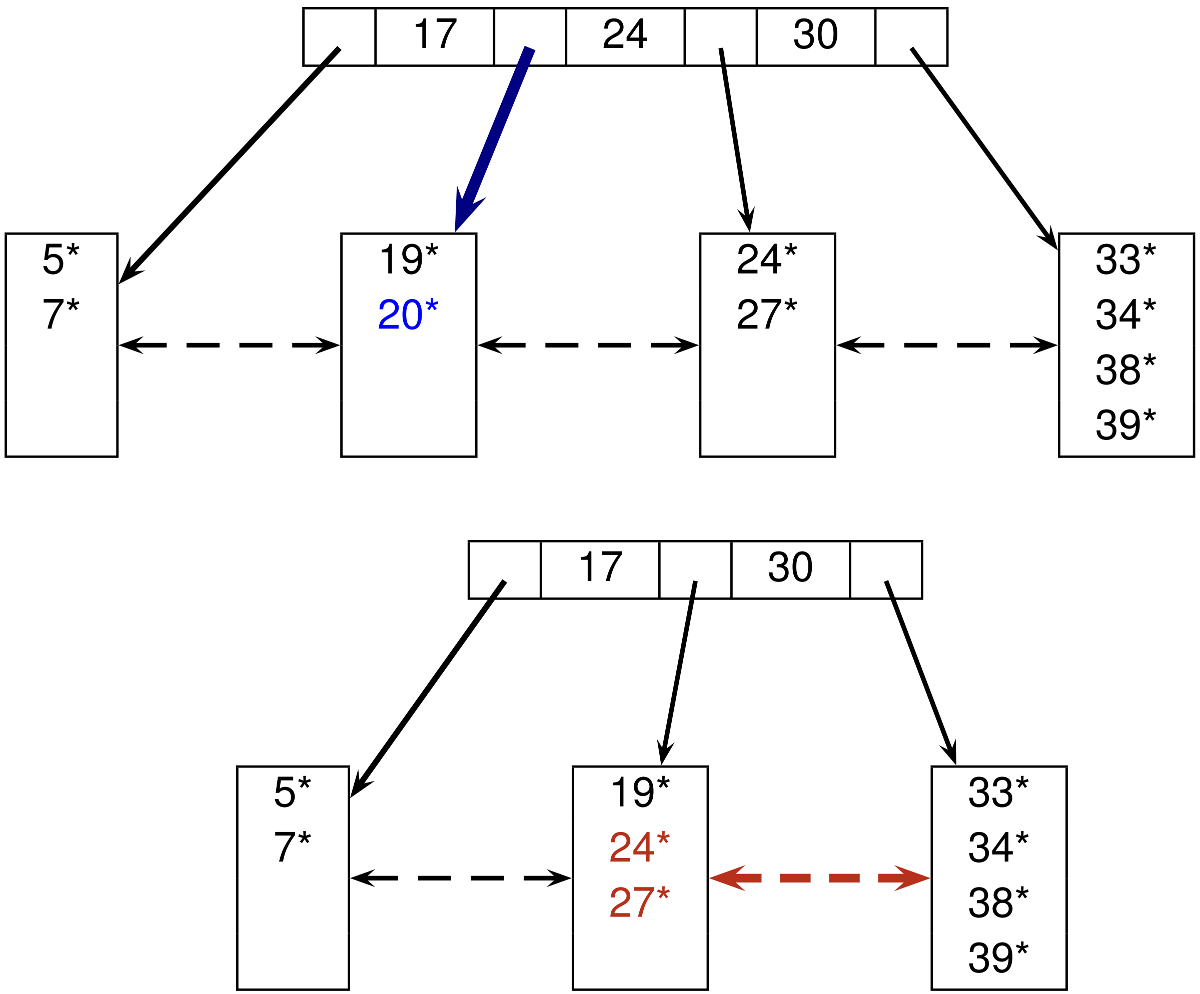


- An underflowed node could be balanced using an adjacent sibling's entry

Deleting 20* (Merging of nodes)

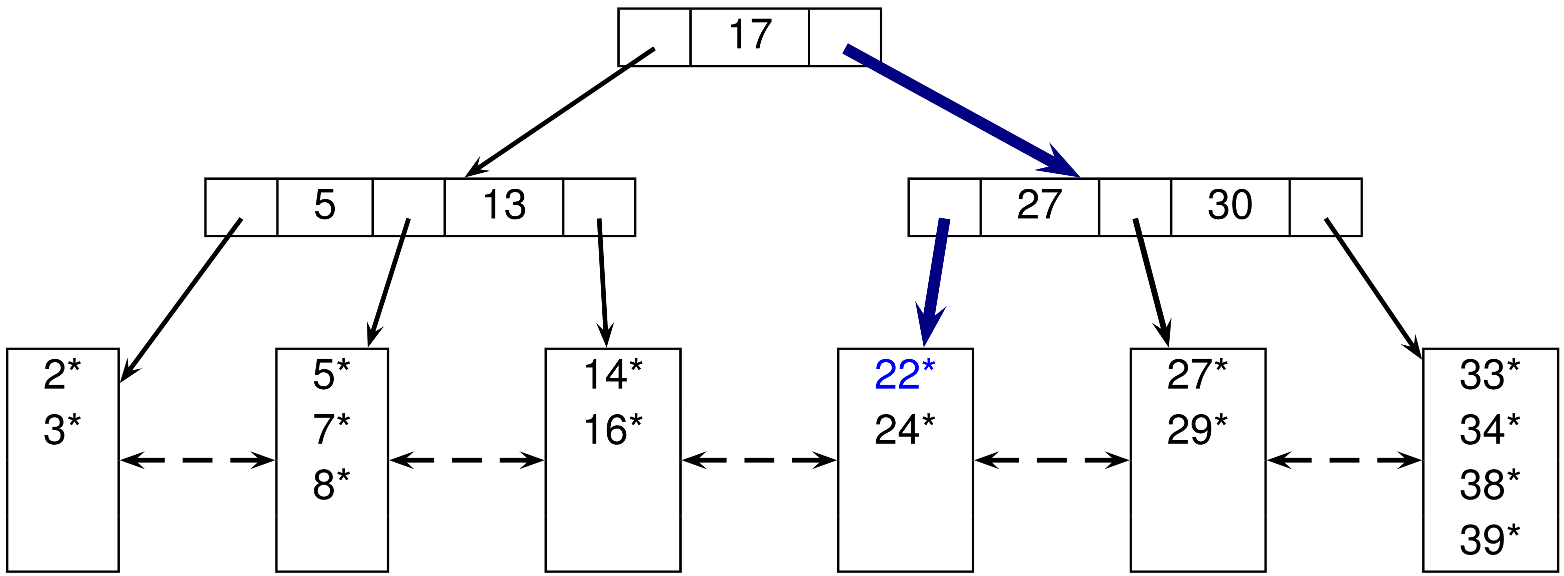


Deleting 20* (Merging of nodes)

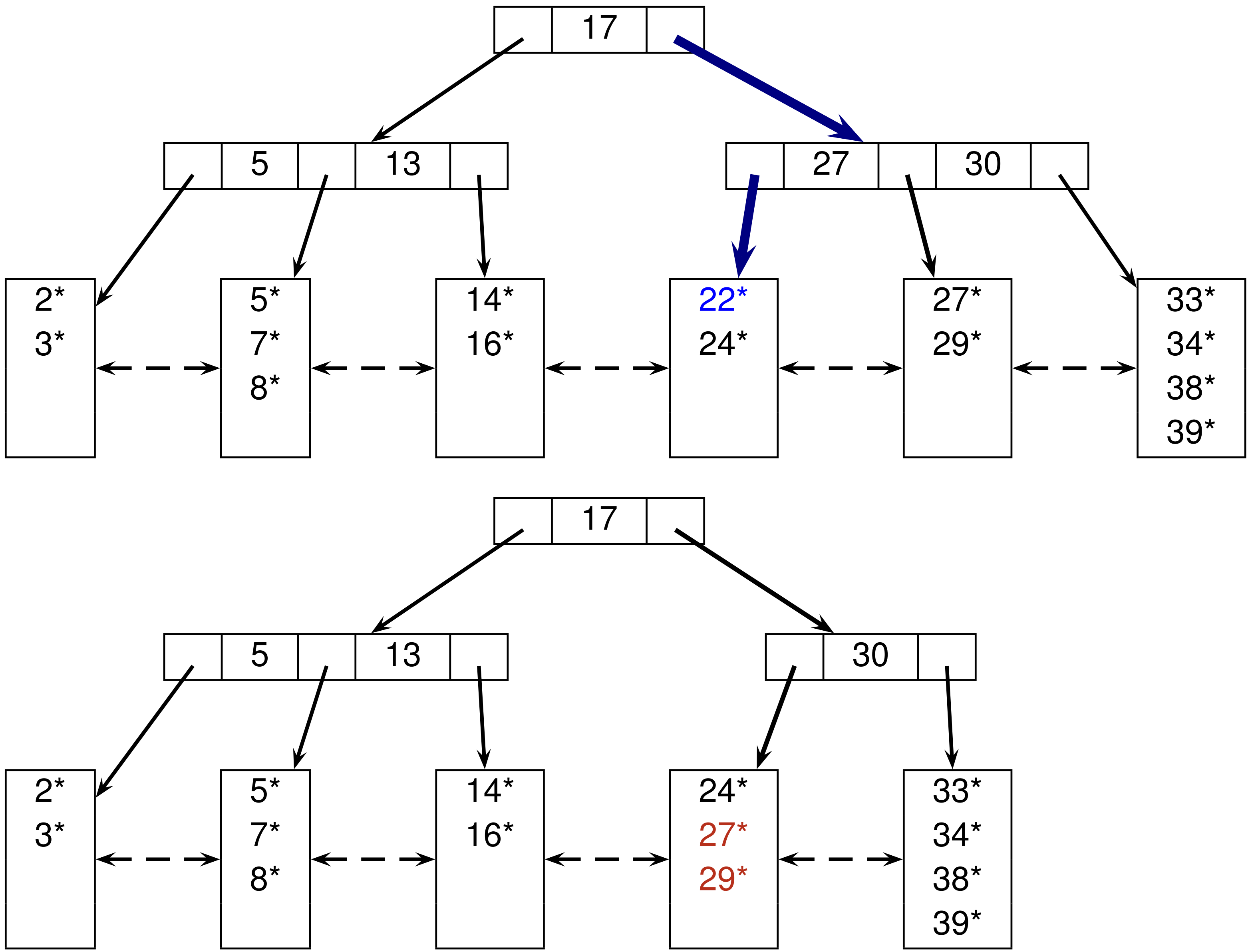


- An underflowed node needs to be merged if each of its adjacent sibling nodes has exactly *d* entries (d = order)

Deleting 22* (Propagation of node merges)

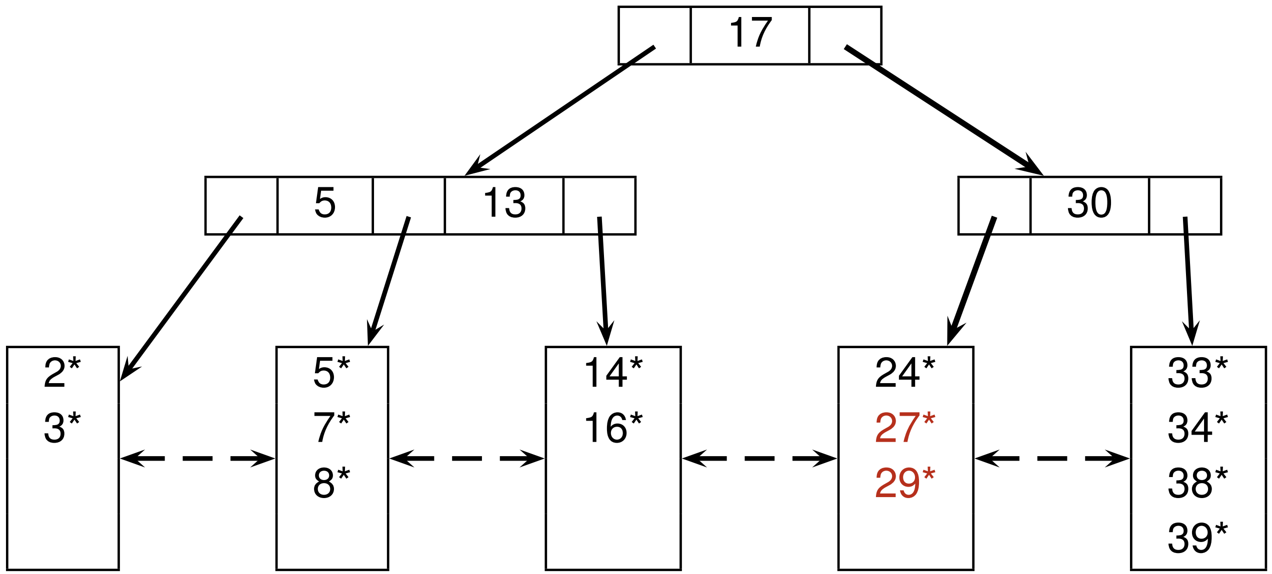


Deleting 22* (Propagation of node merges)

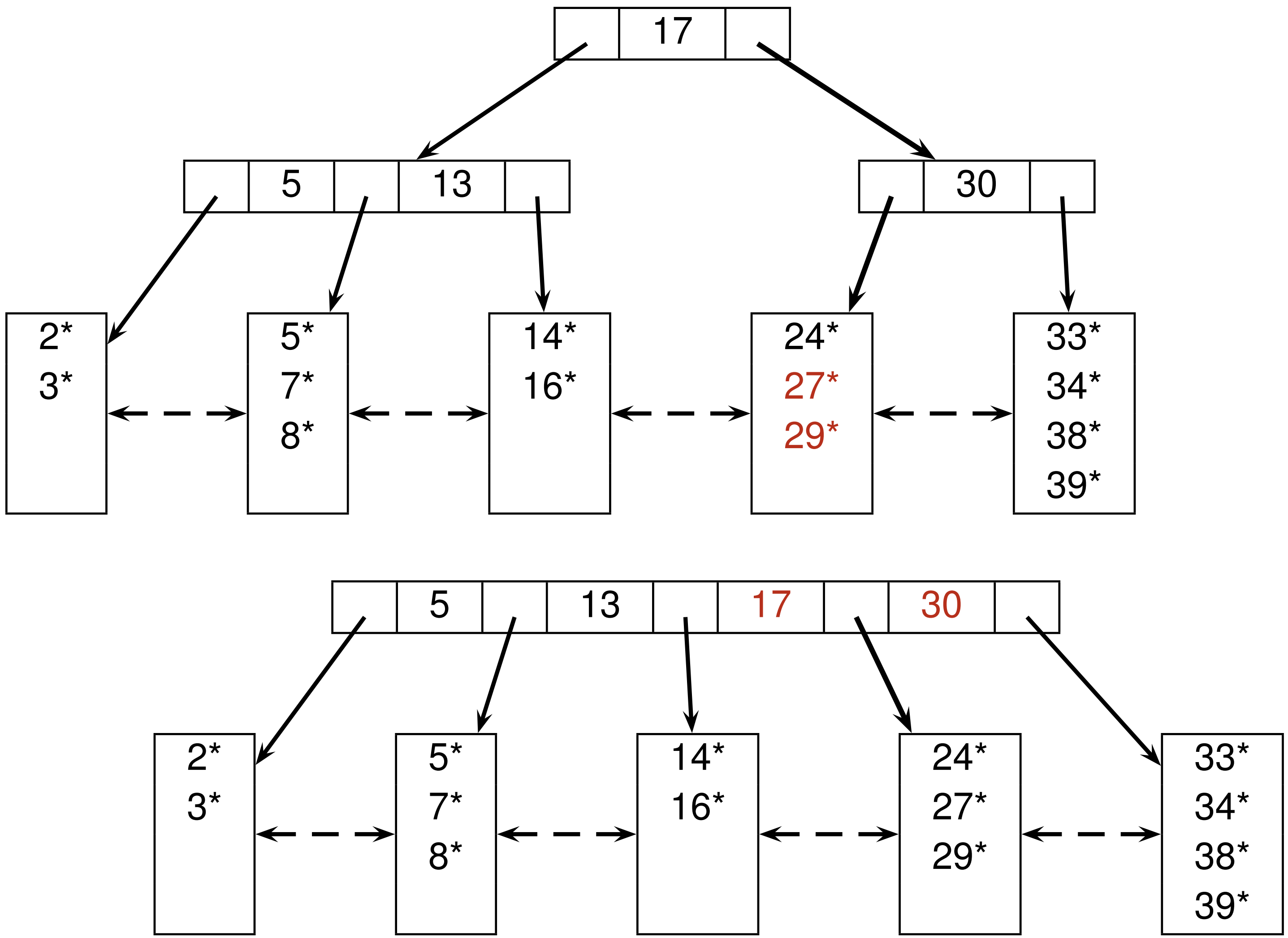


- Node merges can be propagated to ancestor nodes

Deleting 22* (Propagation of node merges)

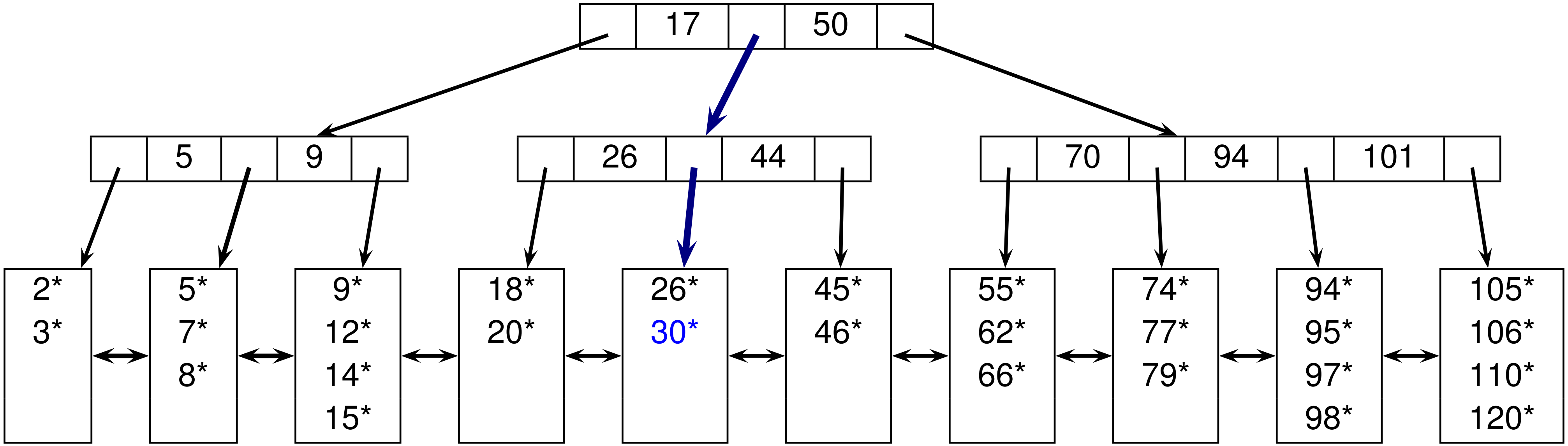


Deleting 22* (Propagation of node merges)

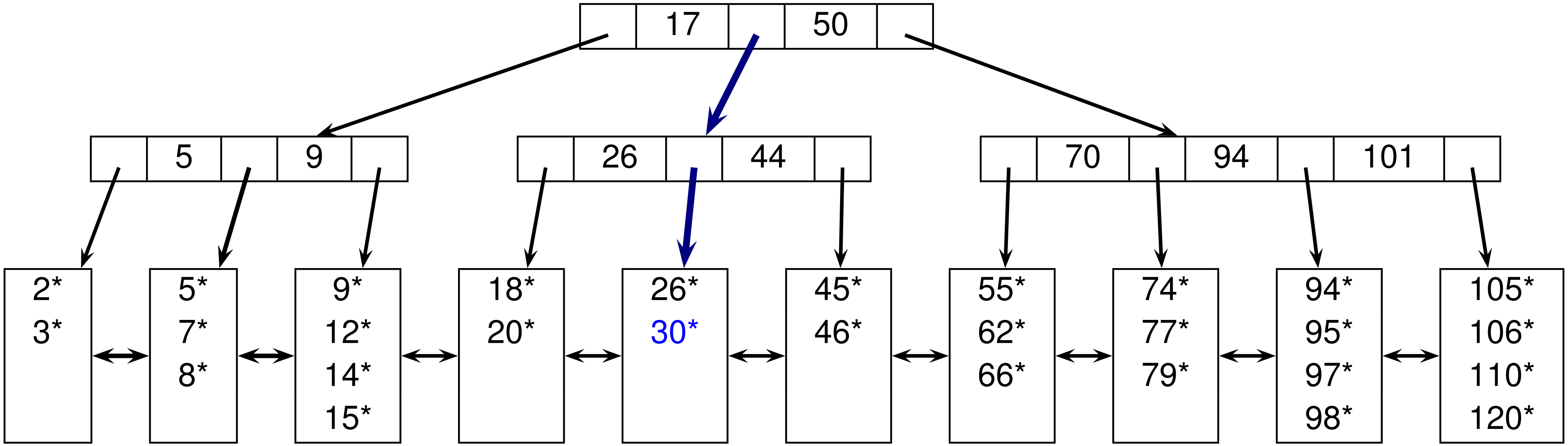


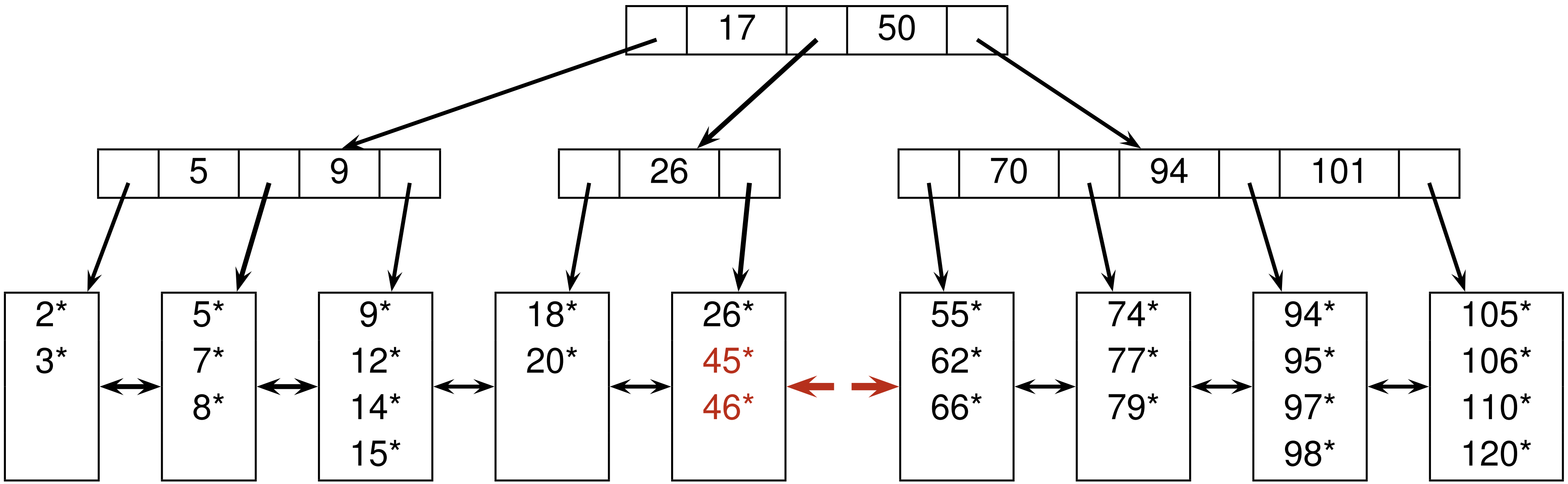
- Pull down appropriate key from parent node to form merged node

Deleting 30* (Redistribution of internal entries)

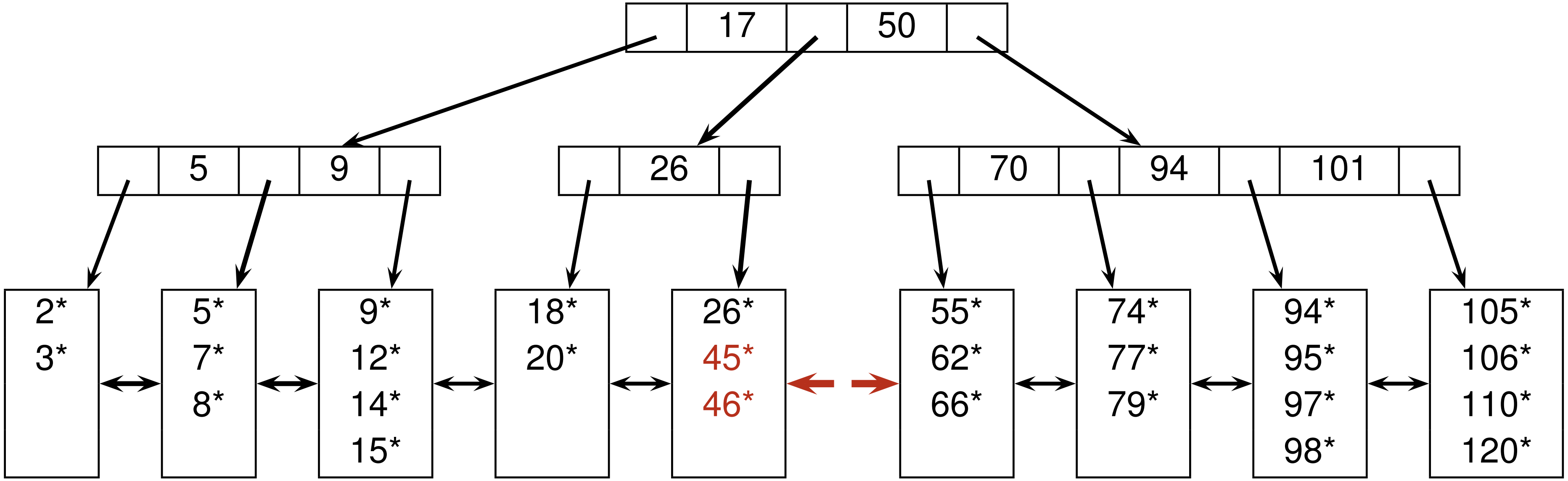


Deleting 30* (Redistribution of internal entries)

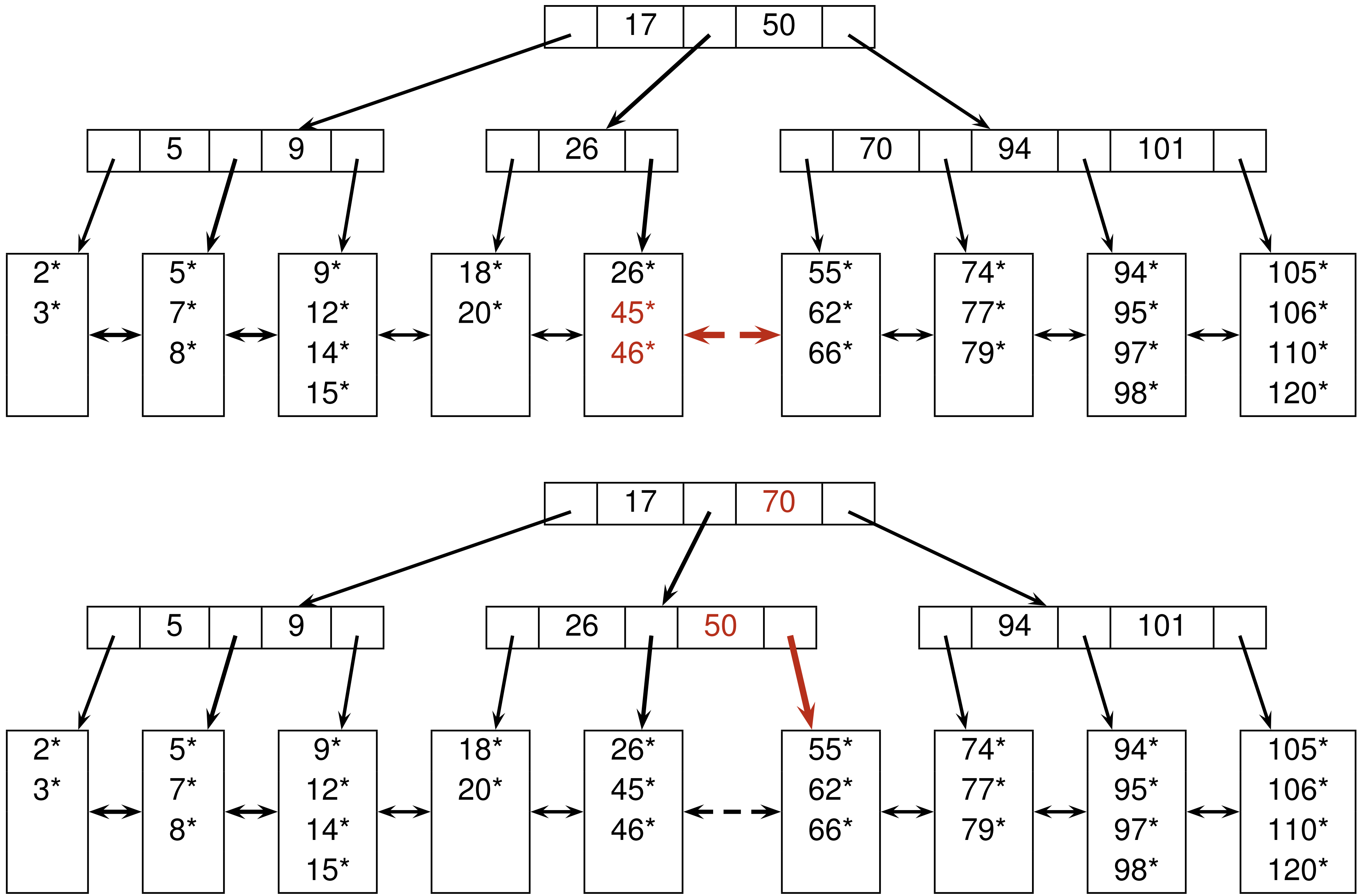




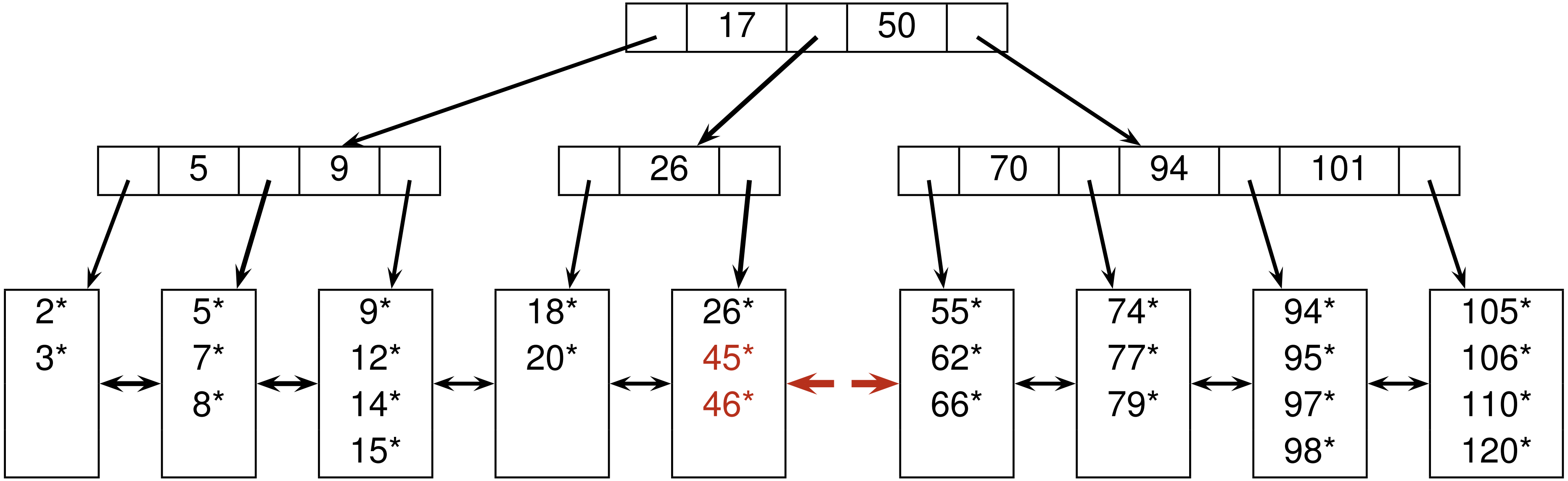
Deleting 30* (Redistribution of data entries)

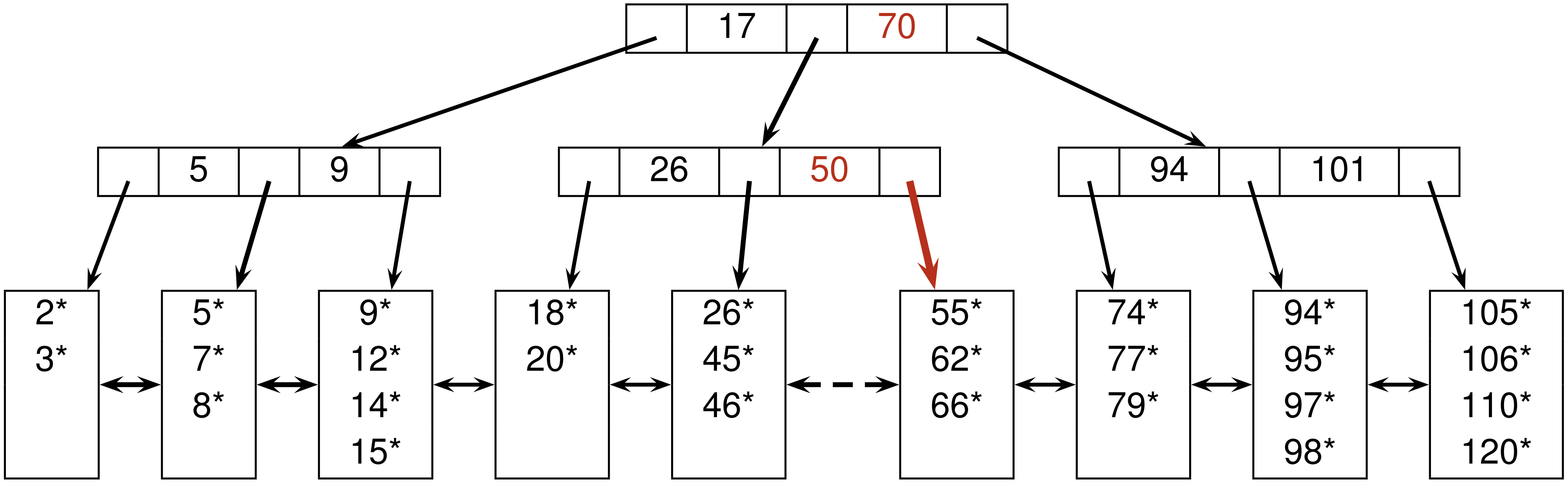


Deleting 30* (Redistribution of data entries)



Deleting 30* (Redistribution of data entries)

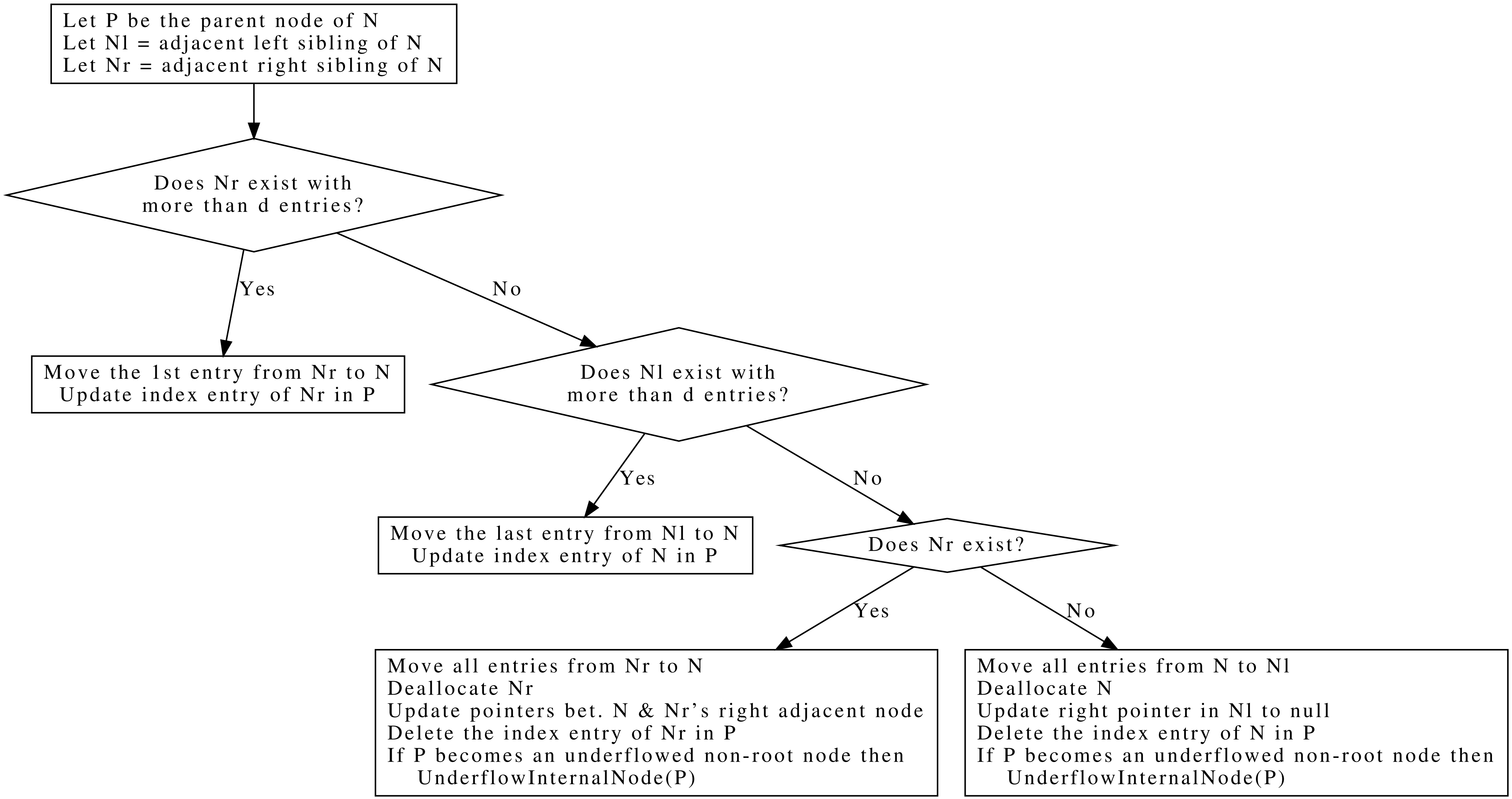




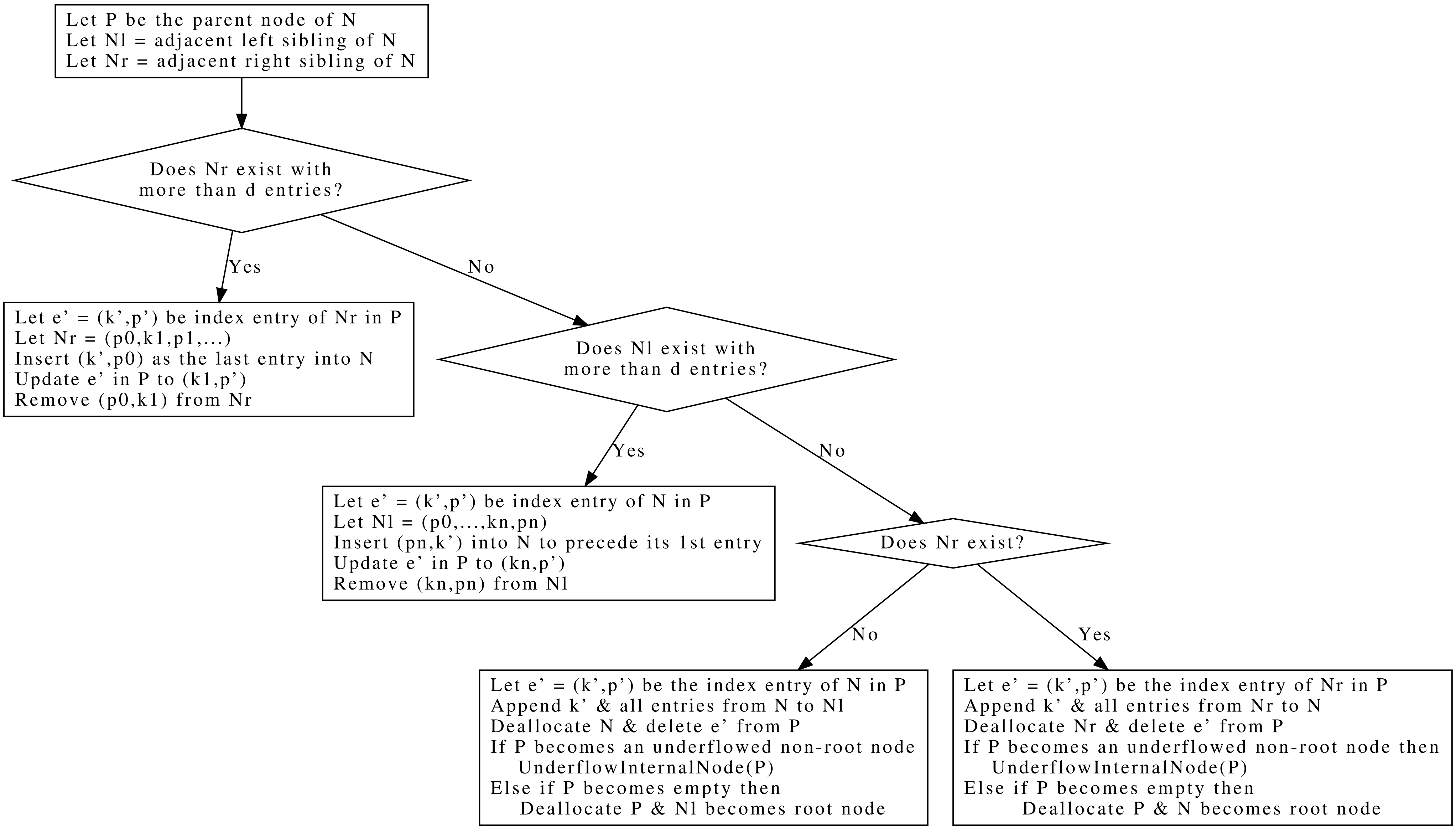
B^+-tree: Deletion Algorithm

N = non-root leaf node that underflows after deletion of a data entry

Assume redistribution is attempted whenever possible



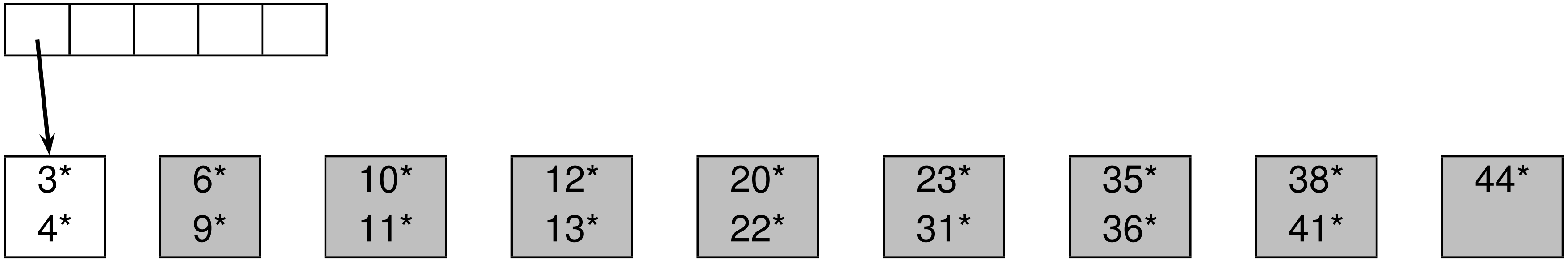
UnderflowInternalNode(N)



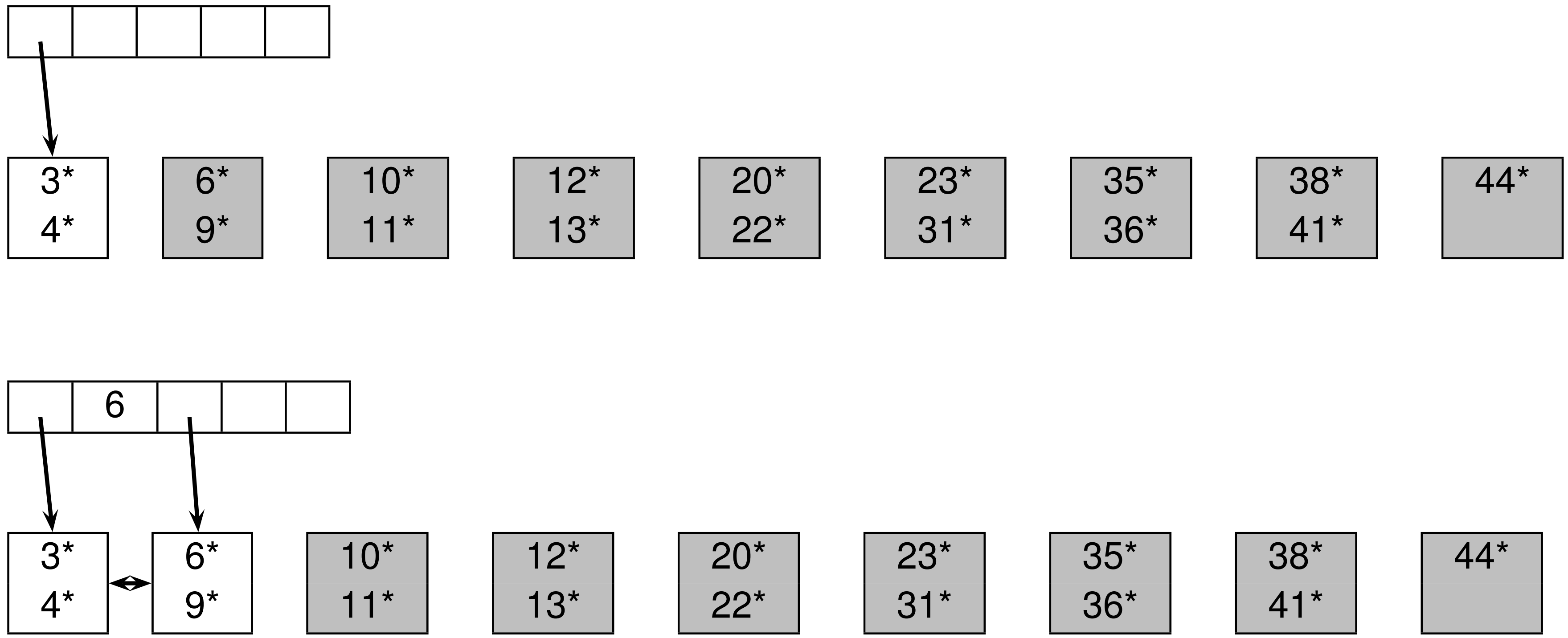
Bulk Loading a B$^+$-tree

- How to build a B$^+$-tree index on a collection of records?
  1. Simple approach: insert records into B$^+$-tree one at a time
  2. Alternative approach: bulk load B$^+$-tree
- Steps to bulk load a B$^+$-tree :
  1. Sort the data entries to be inserted by search key
  2. Load the leaf pages of B$^+$-tree with sorted entries
  3. Initialize B$^+$-tree with an empty root page
  4. For each leaf page (in sequential order), insert its index entry into the rightmost parent-of-leaf level page of B$^+$-tree
- Advantages of bulk loading:
  - Efficient construction algorithm
  - Leaf pages are allocated sequentially

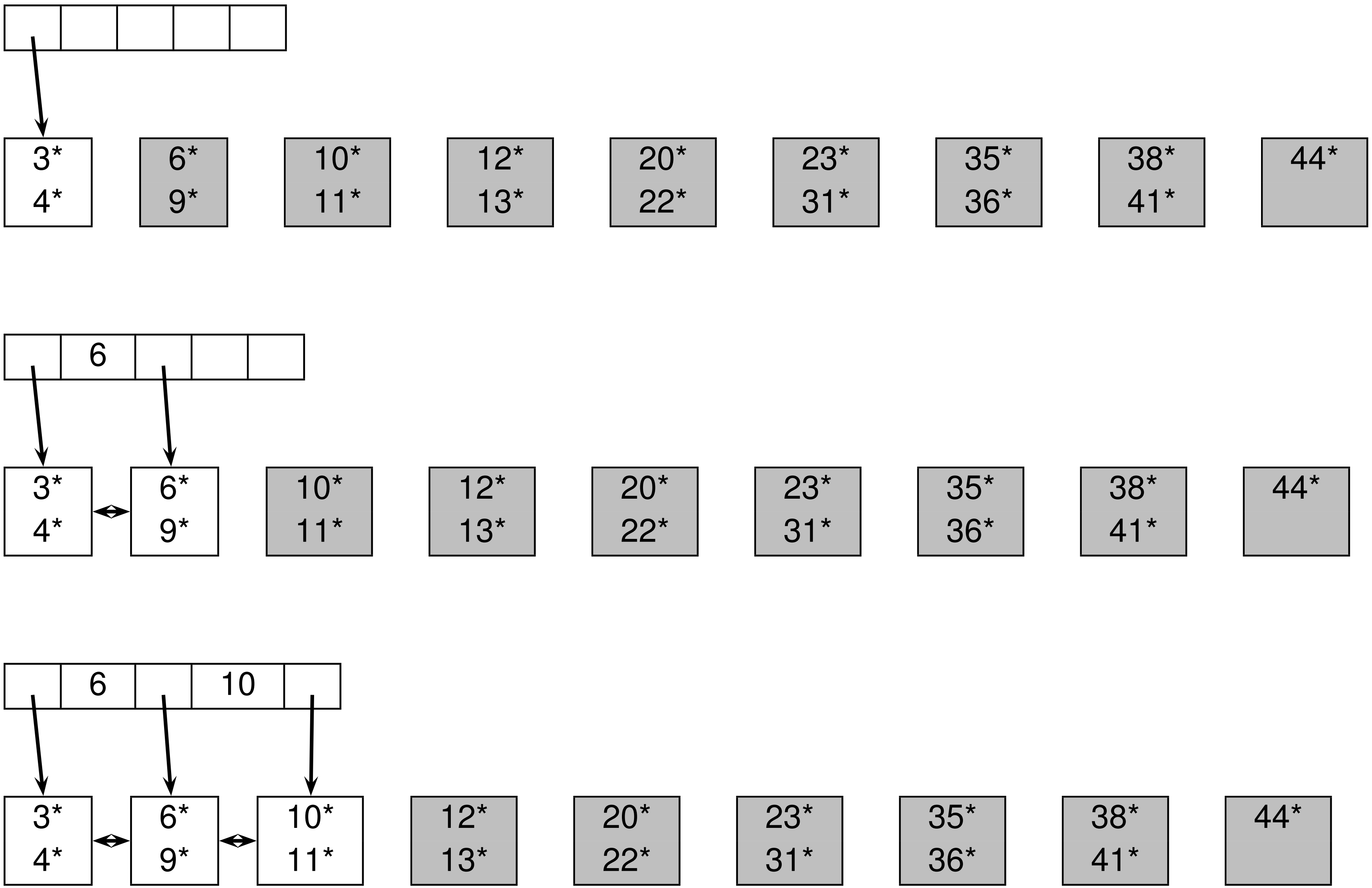
Bulk Loading a B^+-tree : Example (d = 1)



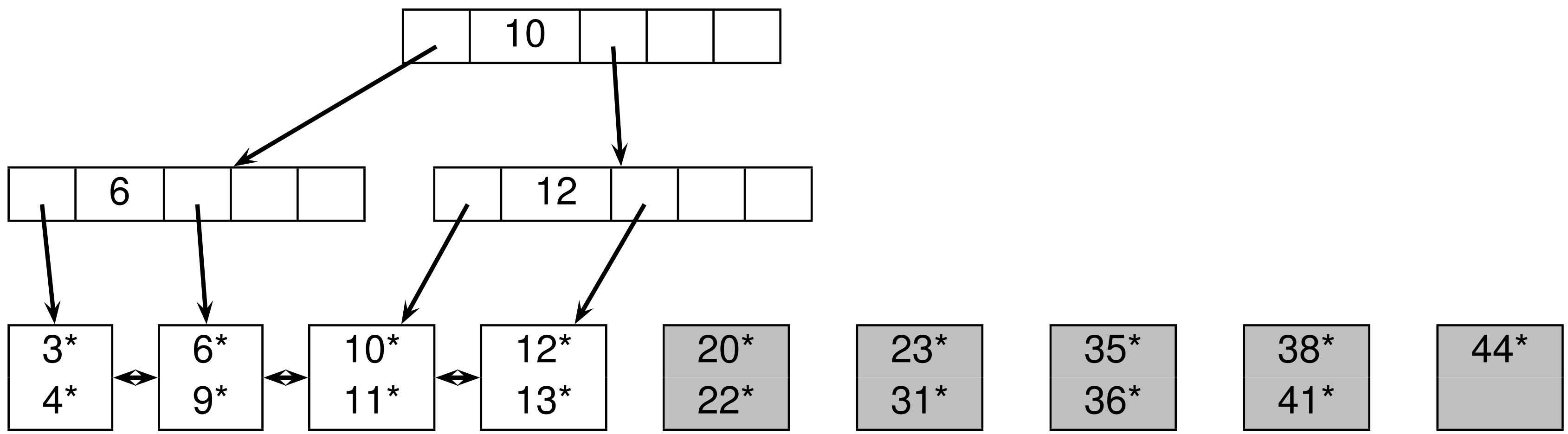
Bulk Loading a B$^+$-tree : Example (d = 1)



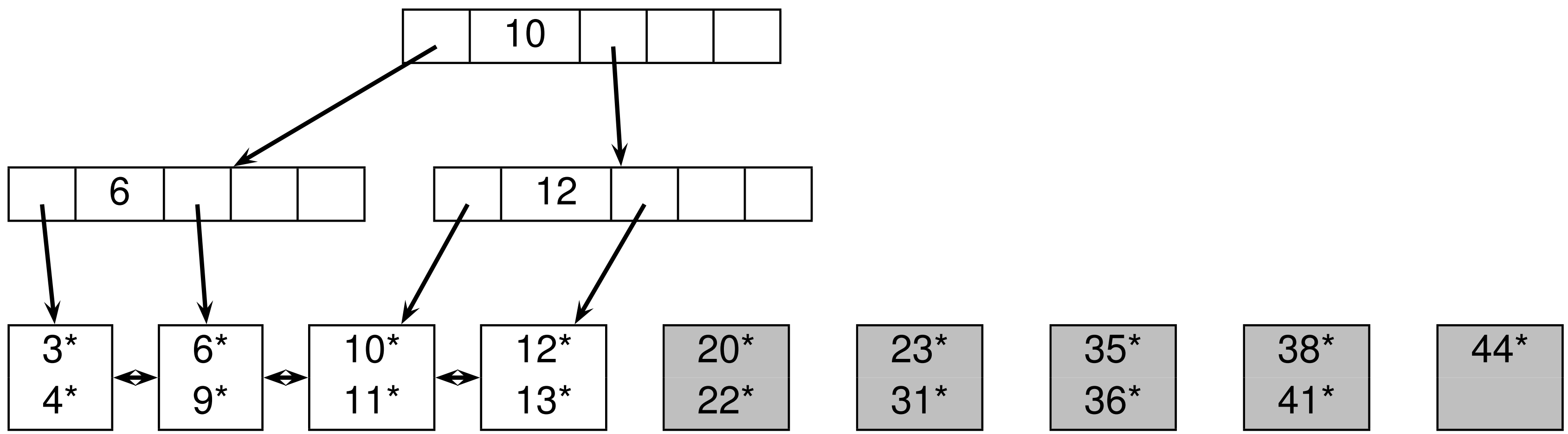
Bulk Loading a B^+-tree : Example (d = 1)

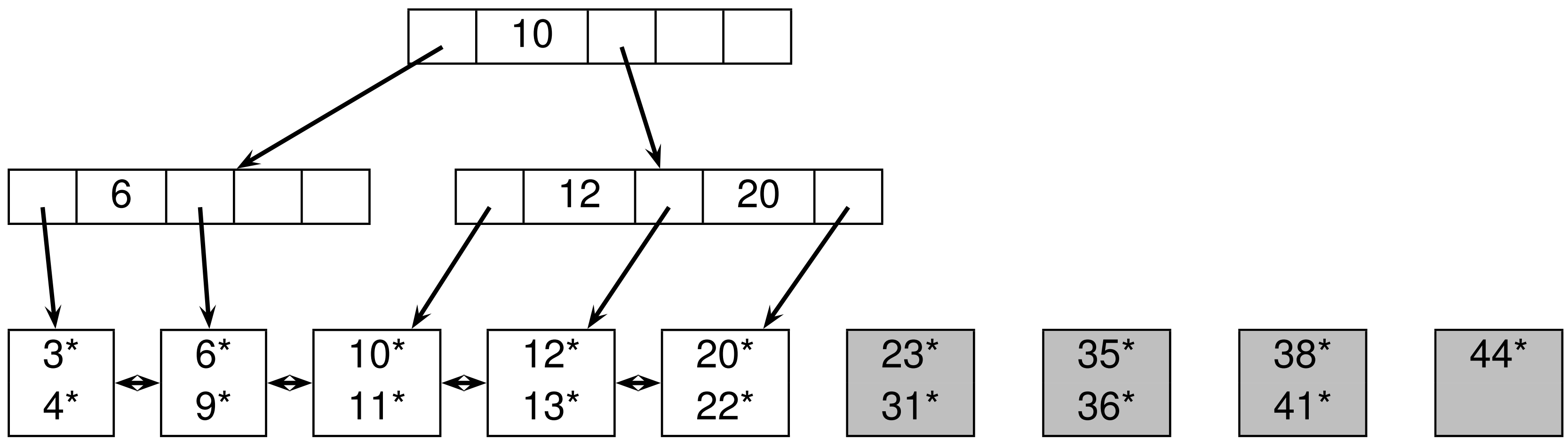


Bulk Loading a B^+-tree : Example (d = 1)

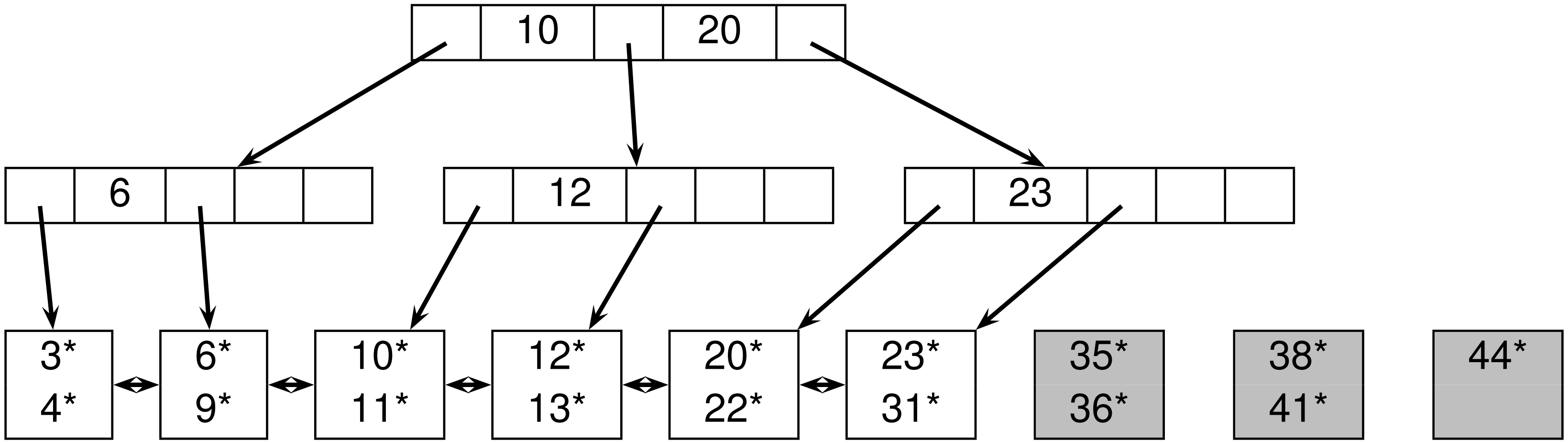


Bulk Loading a B$^+$-tree : Example (d = 1)

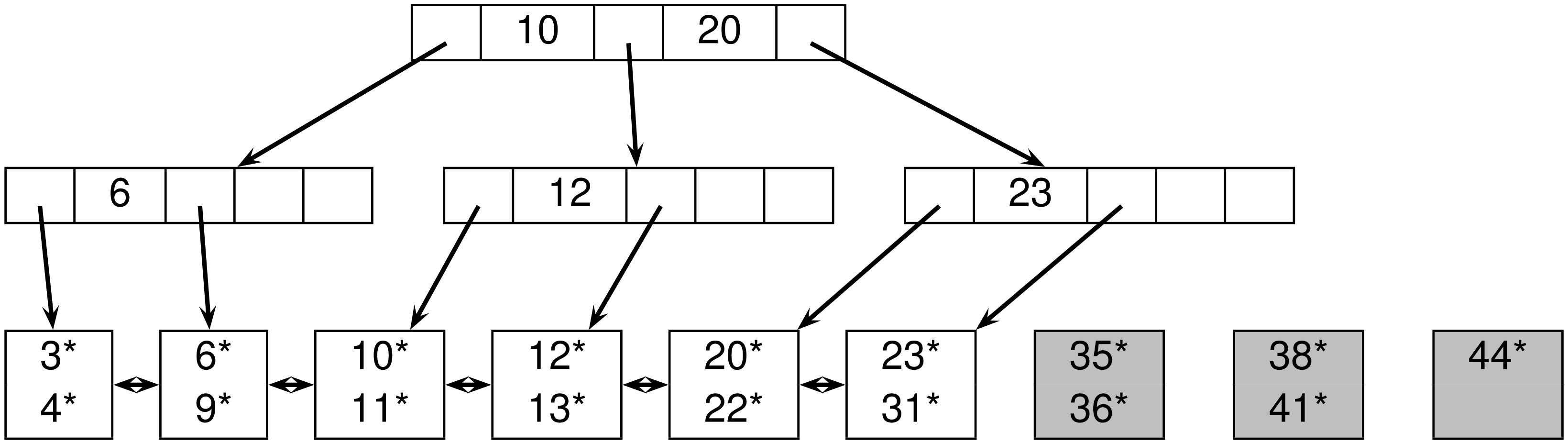


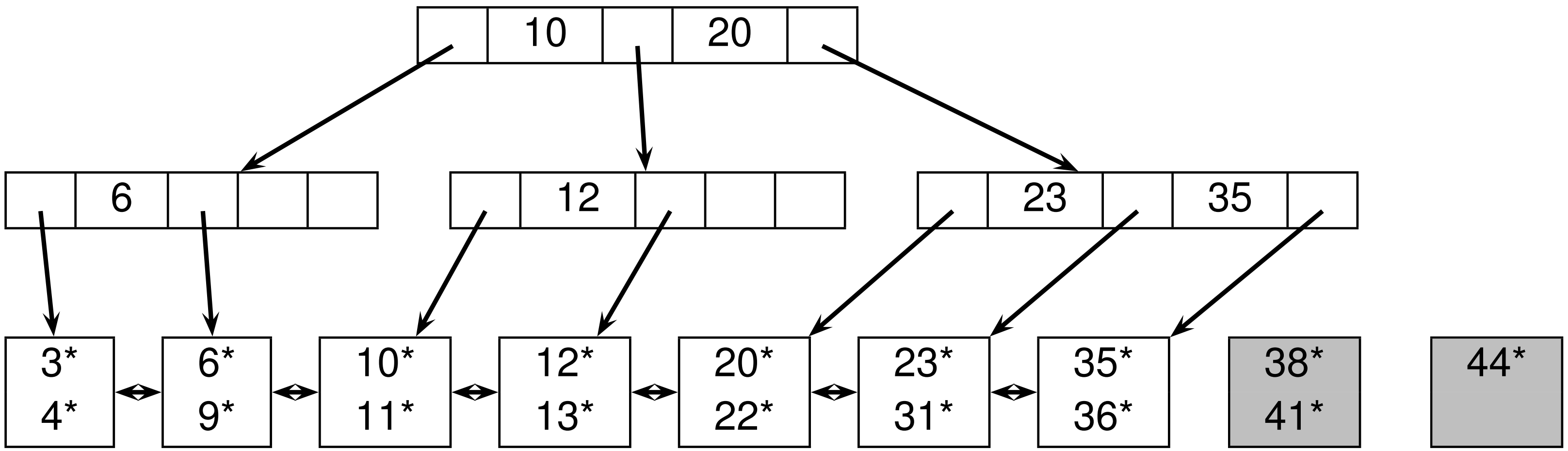


Bulk Loading a B^+-tree : Example (d = 1)

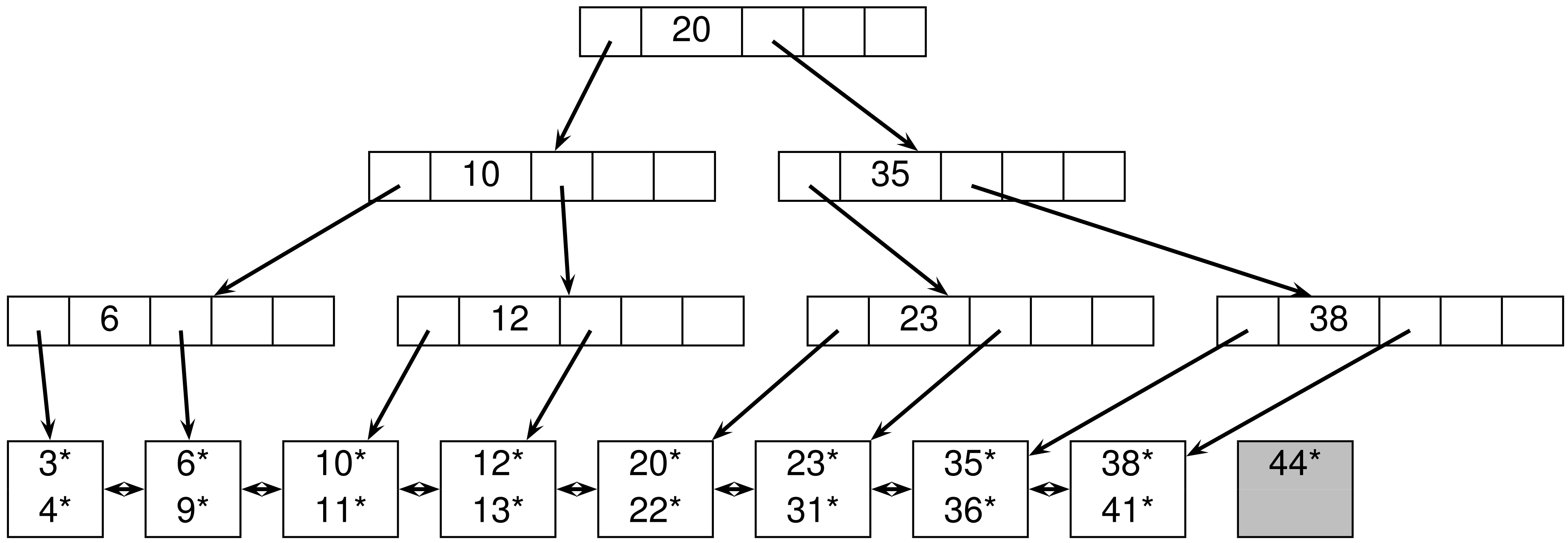


Bulk Loading a B$^+$-tree : Example (d = 1)

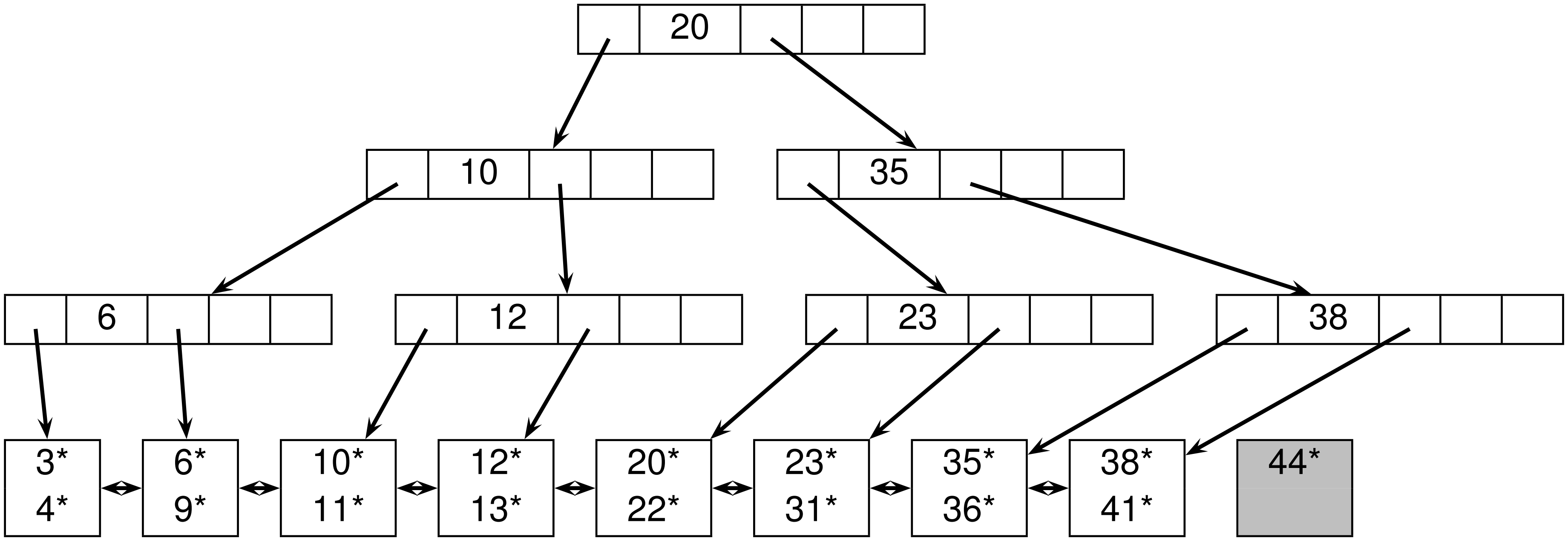


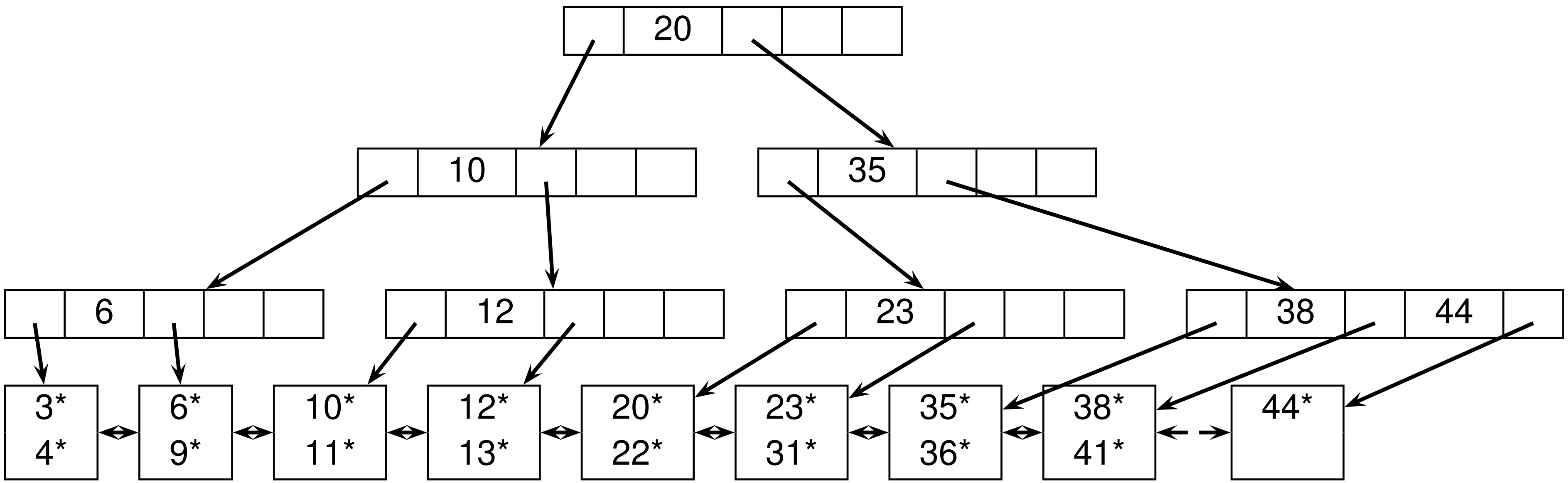


Bulk Loading a B$^+$-tree : Example (d = 1)



Bulk Loading a B$^+$-tree : Example (d = 1)

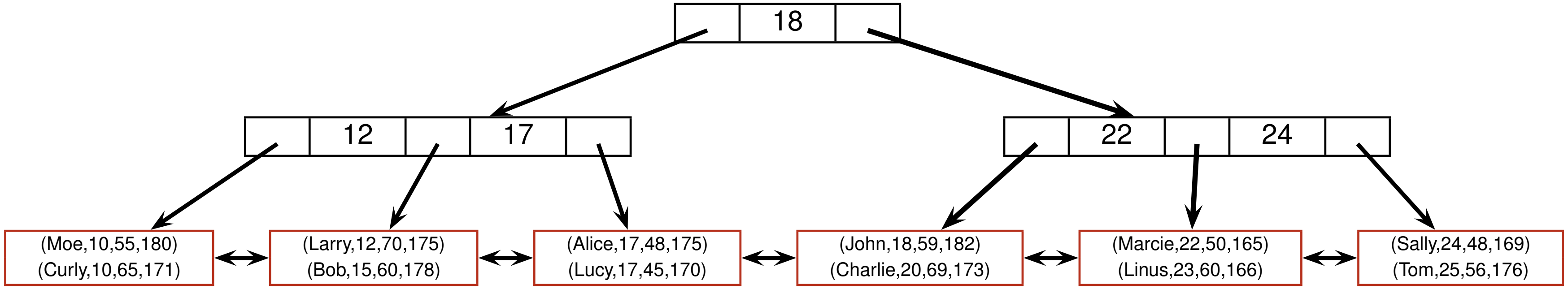




Clustered vs Unclustered Index

- An index is a clustered index if the order of its data entries is the same as or ‘close to’ the order of the data records; otherwise, it is an unclustered index
- An index using Format 1 for its data entries is a clustered index
- There is at most one clustered index for each relation

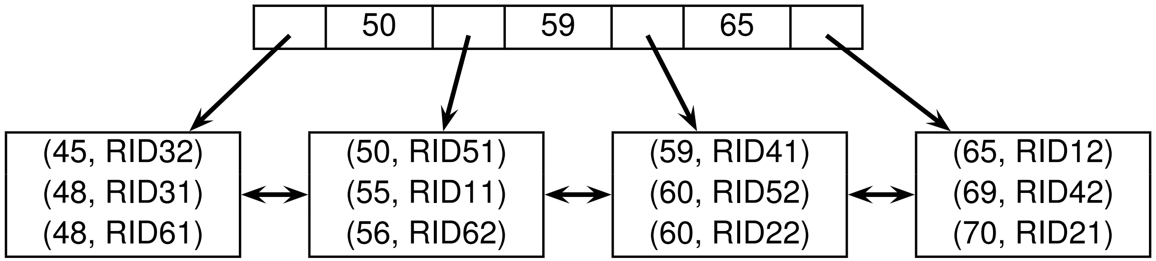
Clustered vs Unclustered Index: Example



Clustered index on R.age

Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176



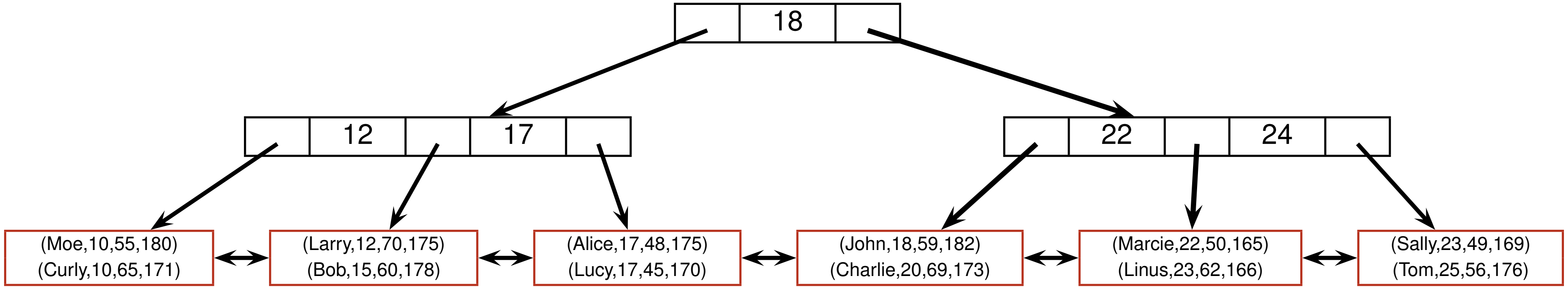
Unclustered index on R.weight

(RIDij = slot j on data page i)

Dense vs Sparse Index

- An index is a dense index if there is an index record for every search key value in the data; otherwise, it is a sparse index
  - Unclustered index must be dense

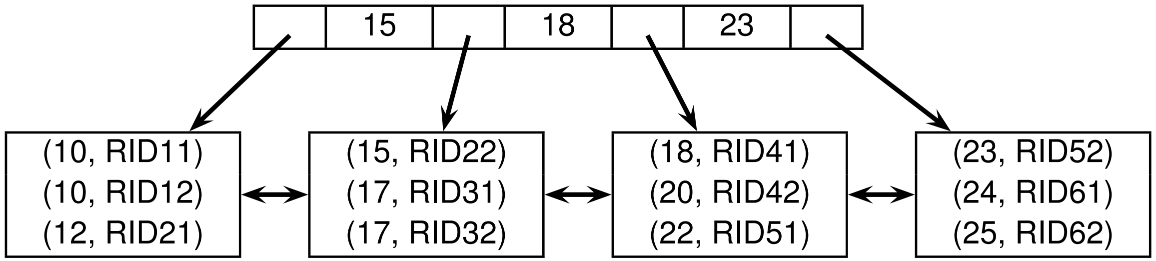
Dense vs Sparse Index: Example



Sparse clustered index on R.age

Relation R

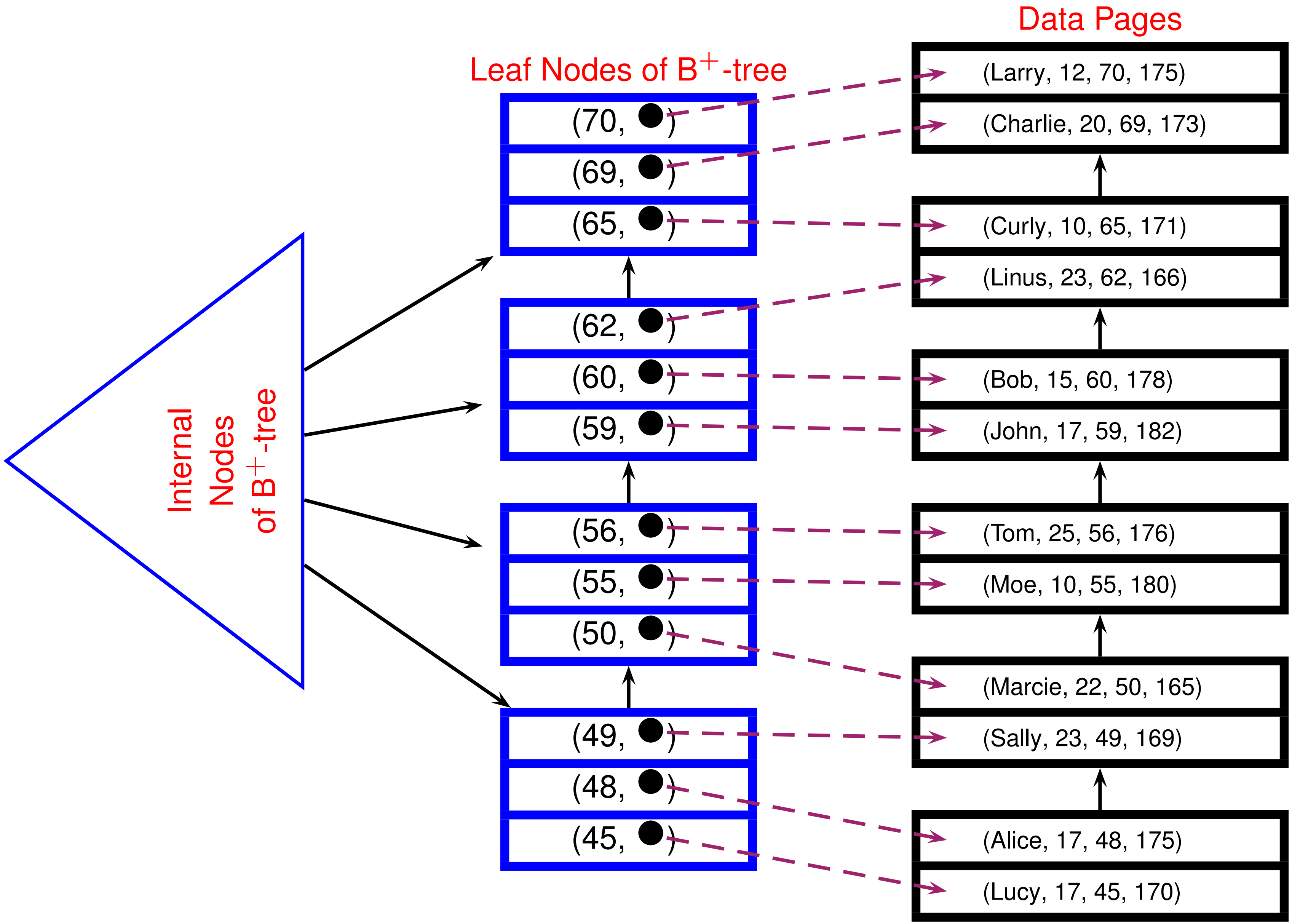
name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	62	166
Sally	24	49	169
Tom	25	56	176



Dense clustered index on R.age

(RIDij = slot j on data page i)

Clustered & Dense B^+-tree on R.weight



Unclustered & Dense B$^+$-tree on R.weight