Index

References:

-- Section 14, Database System Concepts

-- Chapter 17, Fundamentals of Database Systems

-- Database course, CMU

Miao Qiao

The University of Auckland

Basic Concepts

- Why use indexes?
  - To speed up access to desired data.
- How to indicate the desired data?
  - **Search Key** - attribute used to look up records in a file.
- The basic structure of an **index file**
  - records (called **index entries**) of the form

search-key	pointer

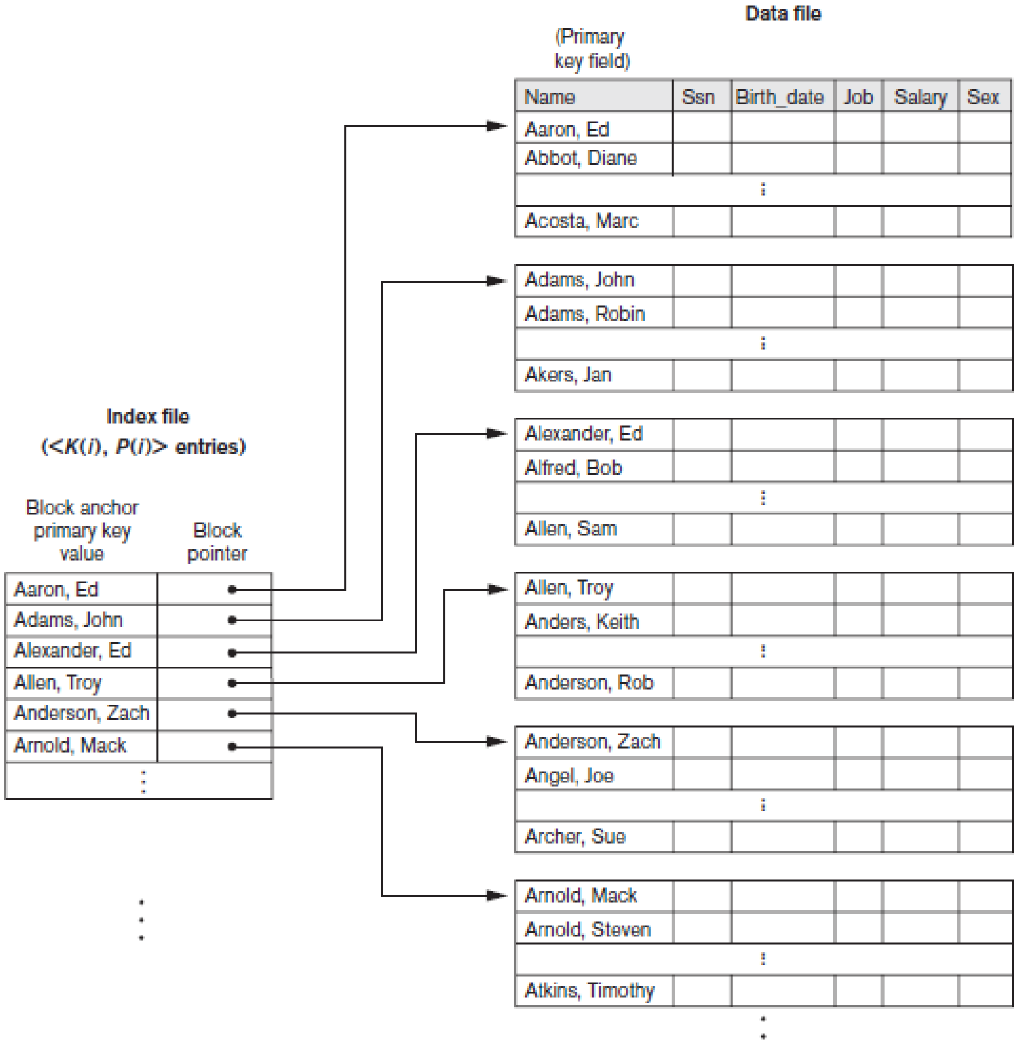
- Index Evaluation Metrics
  - Access types supported efficiently. E.g.,
    - Records with a specified value in the attribute
    - Records with an attribute value falling in a specified range of values.
  - Access time, Insertion time, Deletion time, Space overhead
- Types of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Ordered Indices

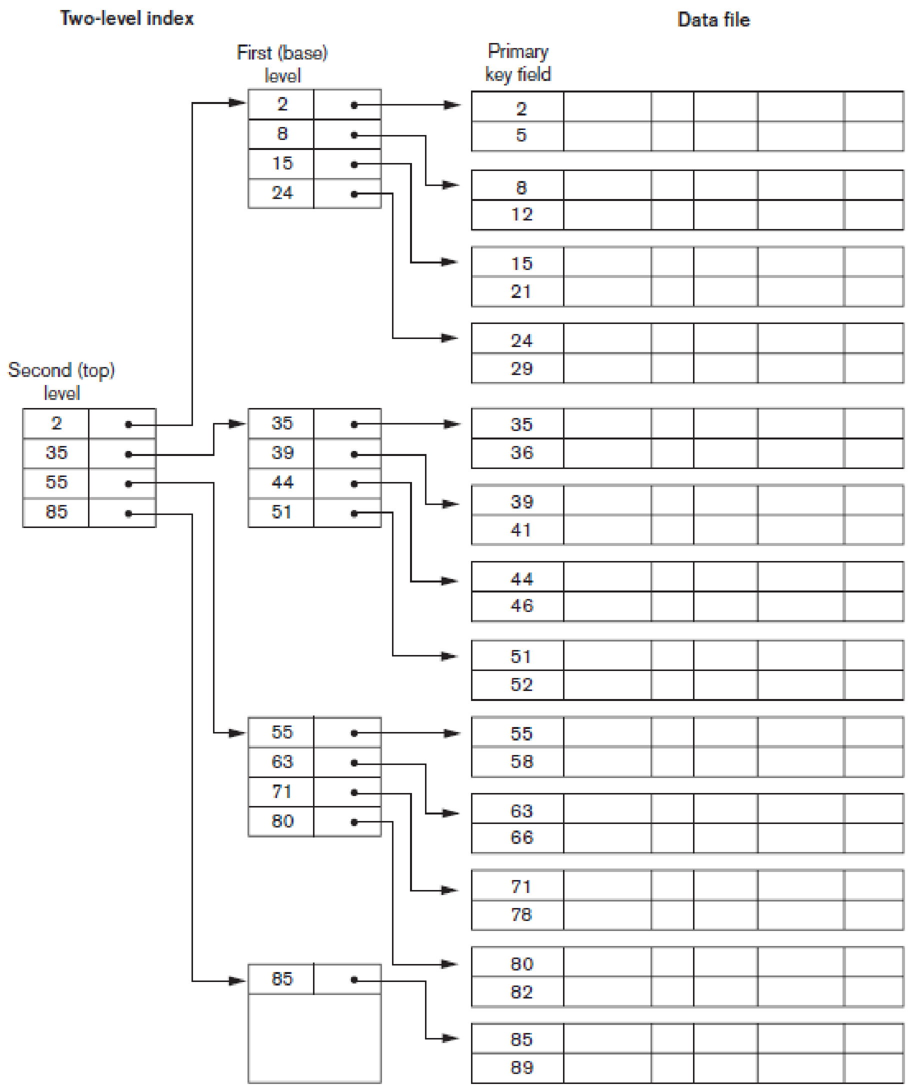
- In an **ordered index,** index entries are stored sorted on the search key value.
- Primary index
- Clustering index
- Secondary index
- Dense index
- Sparse index

Primary Indexes

- In a sequentially ordered file, the index whose search key specifies the sequential order of the file.

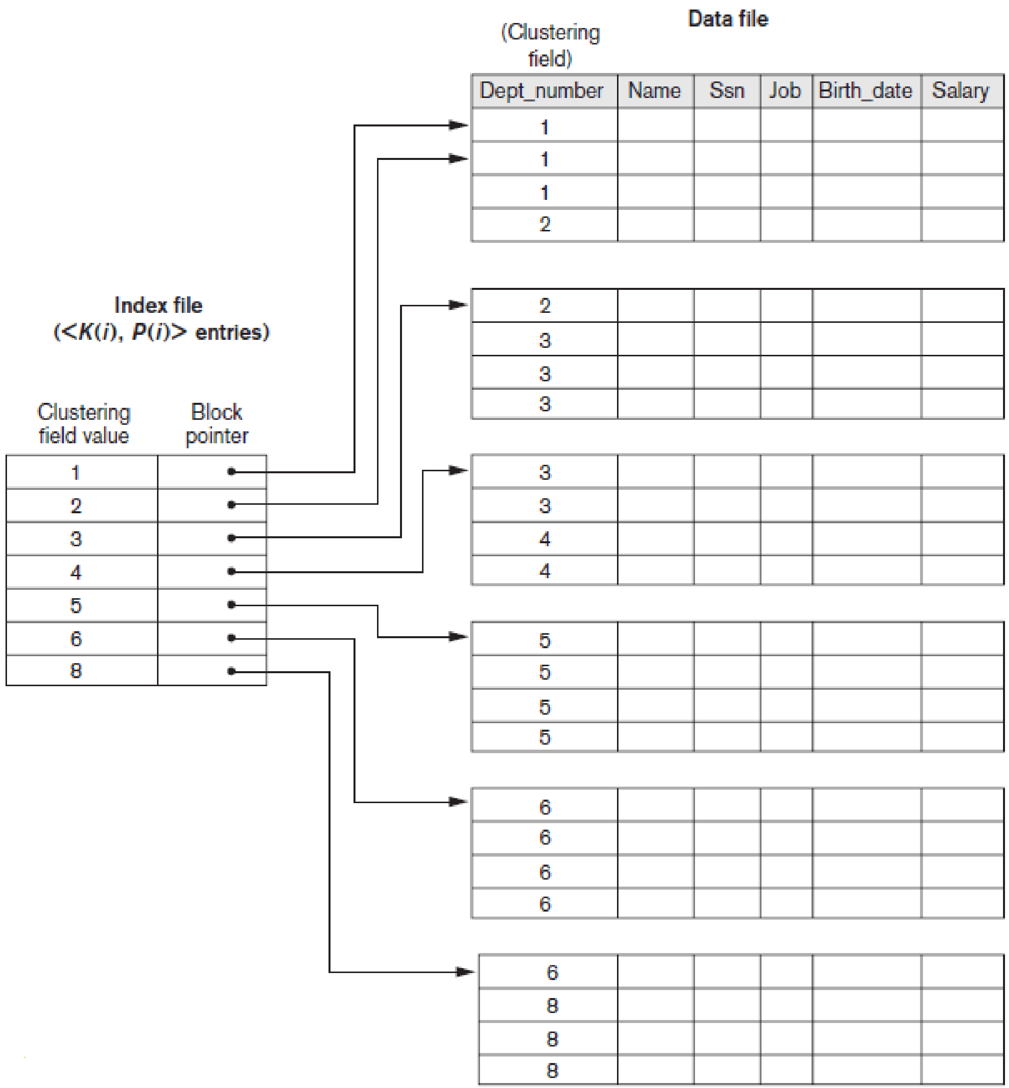


Primary Indexes (two-level)



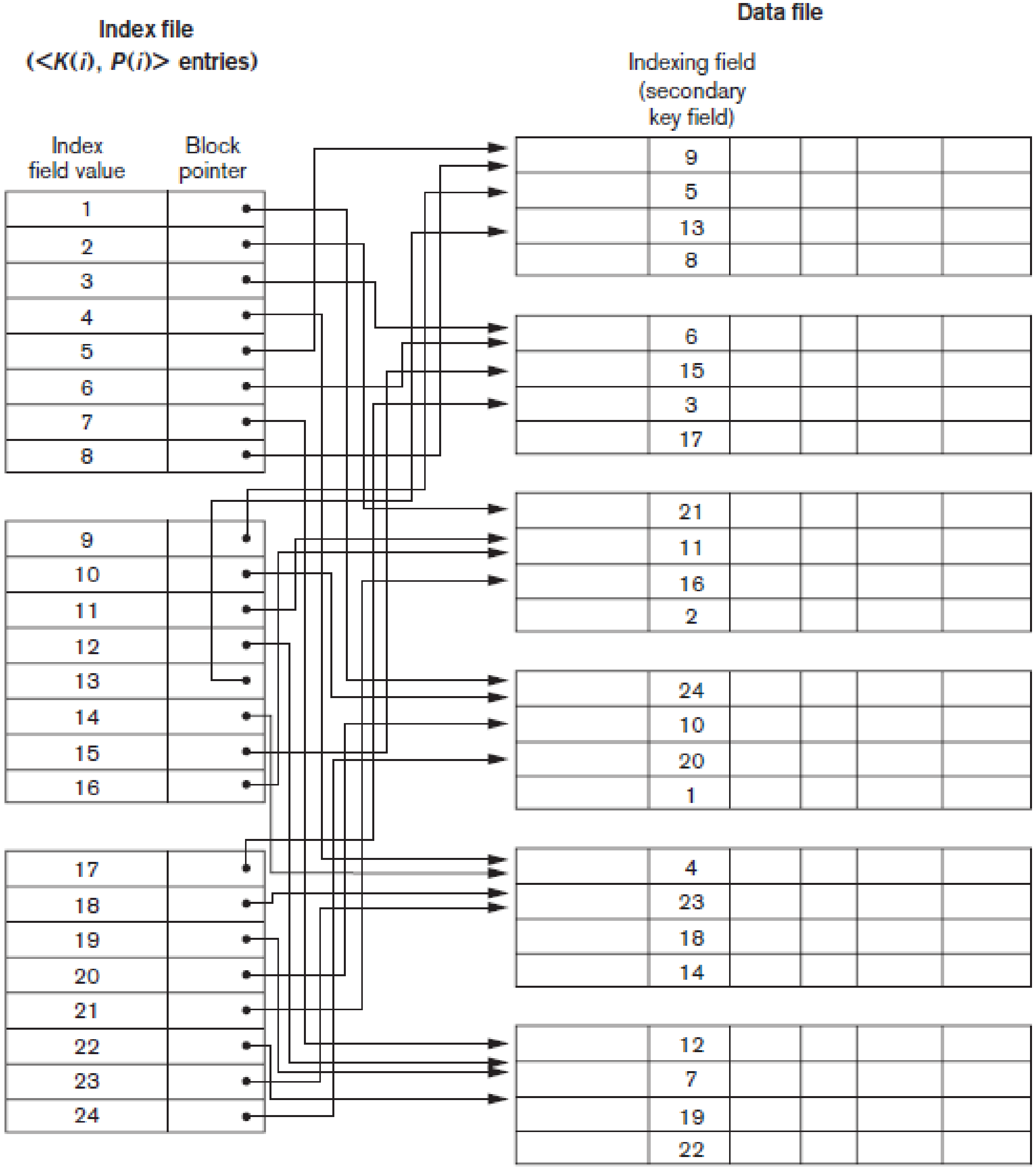
Clustering Index

- Sequential file ordered on a search key, with a clustering index on the search key.



Secondary Index

- An index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index**.



Dense index

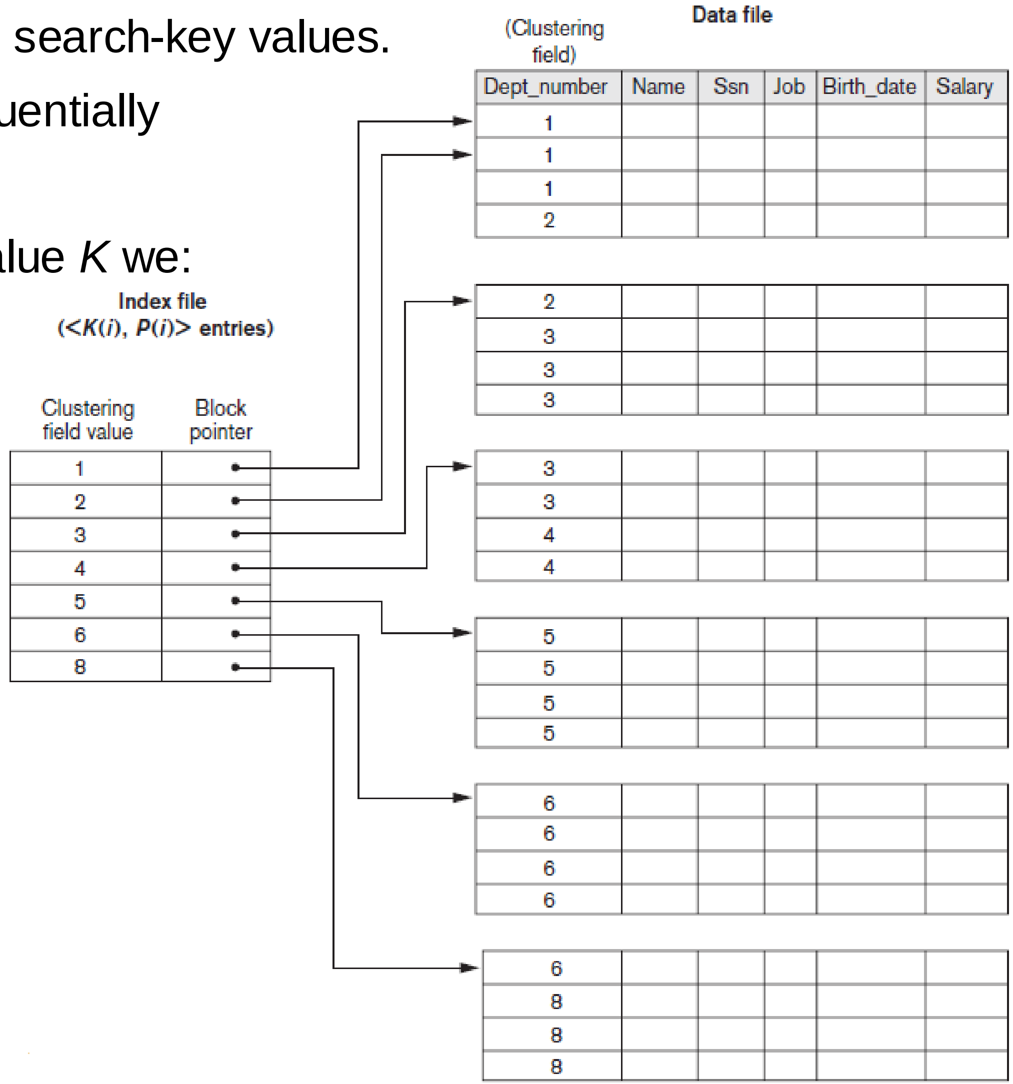
- Index record appears for every search-key value in the file. E.g. index on *ID* attribute of *instructor* relation
  - Secondary index must be dense

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sparse index

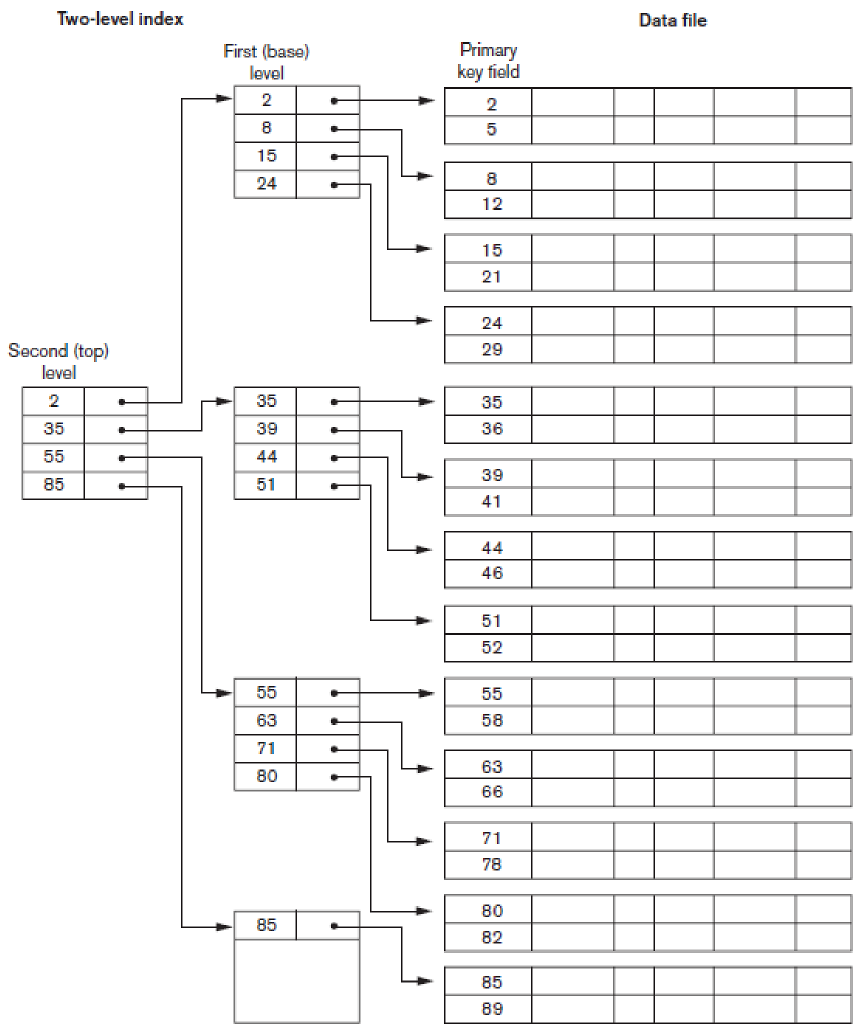
- Contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Get the pointer
  - Sequential read the data file
  - Example:
    - Find 2
    - Find 5



Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.
- **Dense index** — Index record appears for every search-key value in the file. E.g. index on *ID* attribute of *instructor* relation
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

Sparse index – How to update?



B-Tree Family

There is a specific data structure called a **B-Tree**.

People also use the term to generally refer to a class of balanced tree data structures:

→ **B-Tree** (1970)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B**^{link}**-Tree** (1981)
→ **Bε-Tree** (2003)
→ **Bw-Tree** (2013)

B+Tree

A **B+Tree** is a self-balancing, ordered ***m***-way tree for searches, sequential access, insertions, and deletions in **O(log$_m$ n) I/Os** where ***m*** is the tree fanout, n is the number of keys.

→ It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)

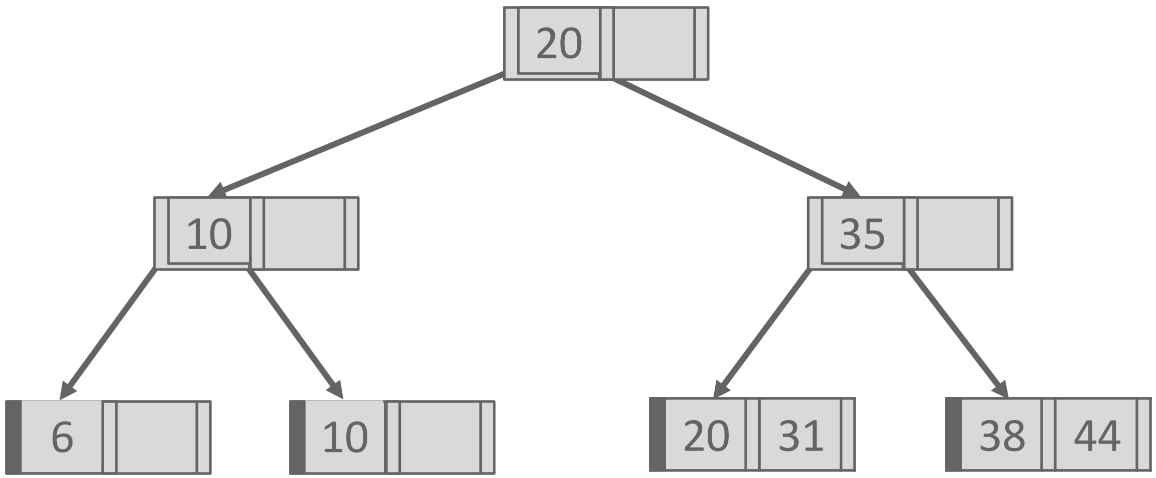
→ Every node other than the root is at least half-full
$m/2-1 \le k \le m-1$, k: # of keys in the node

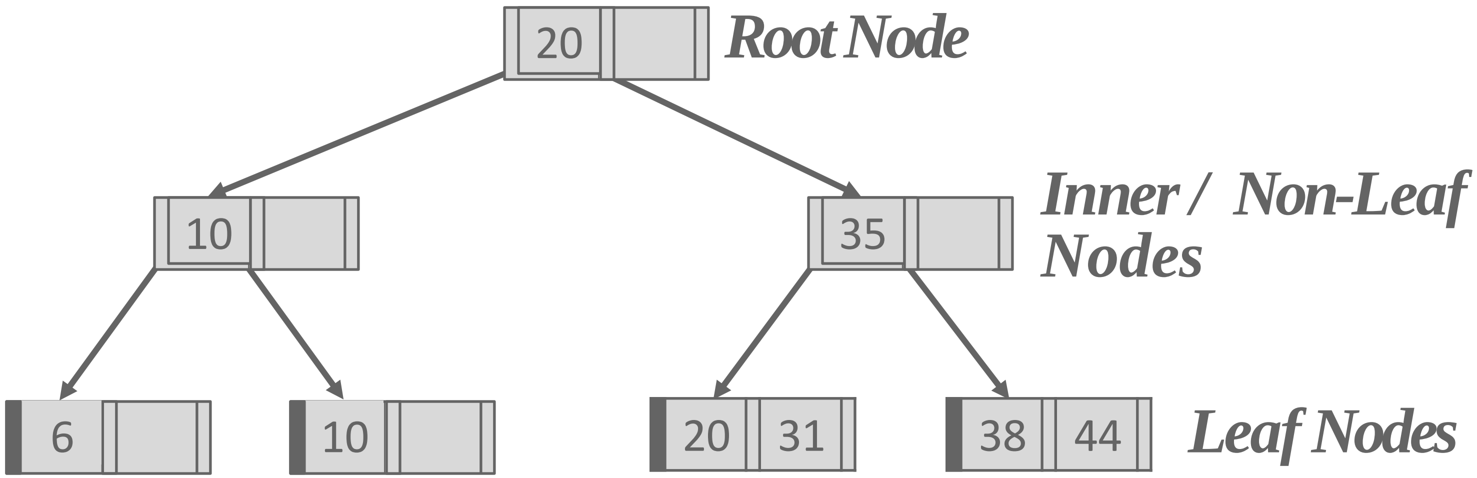
→ Every inner node with **k** keys has **k+1**non-null children.

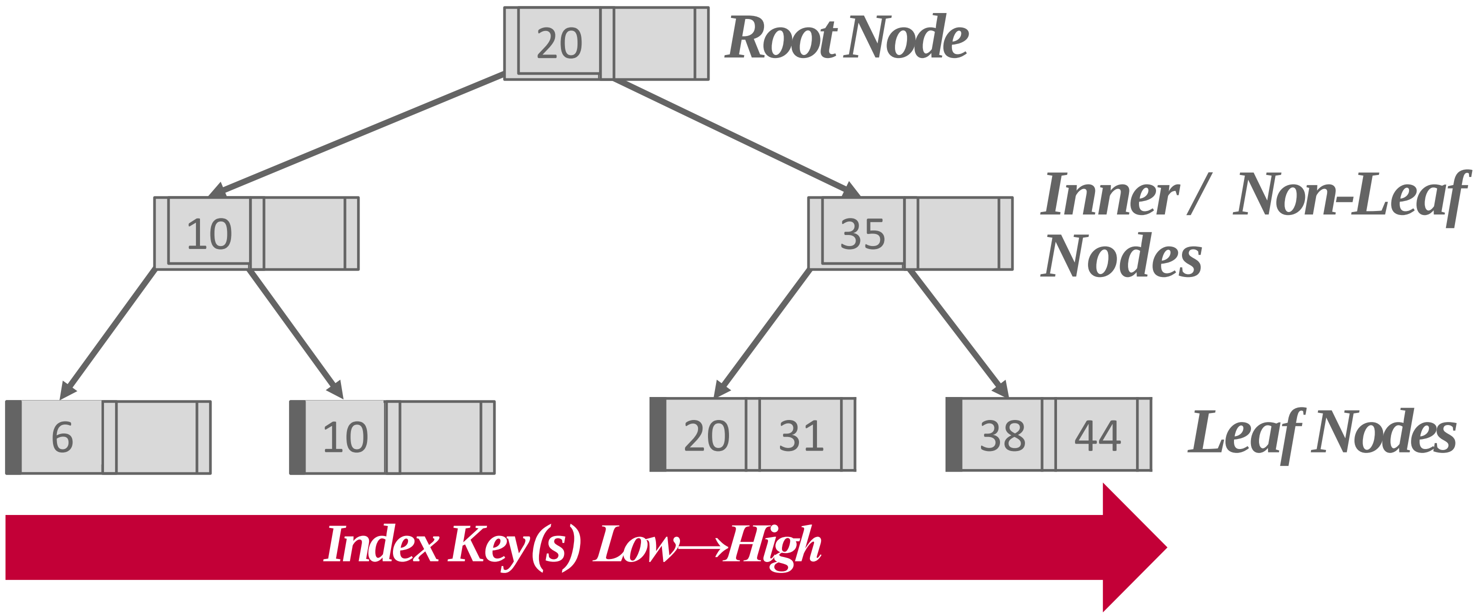
→ Optimized for reading/writing large data blocks.

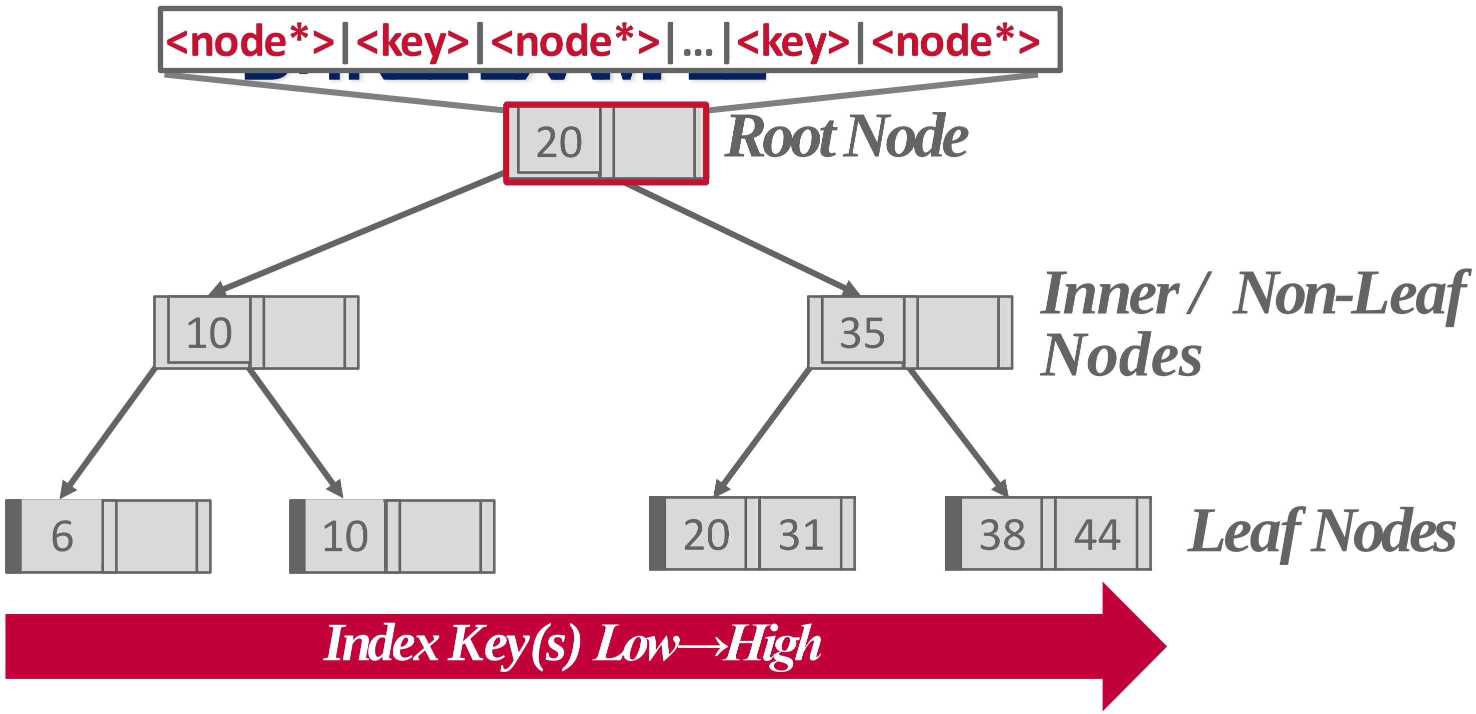
Some real-world implementations relax these properties, but we will ignore that for now…

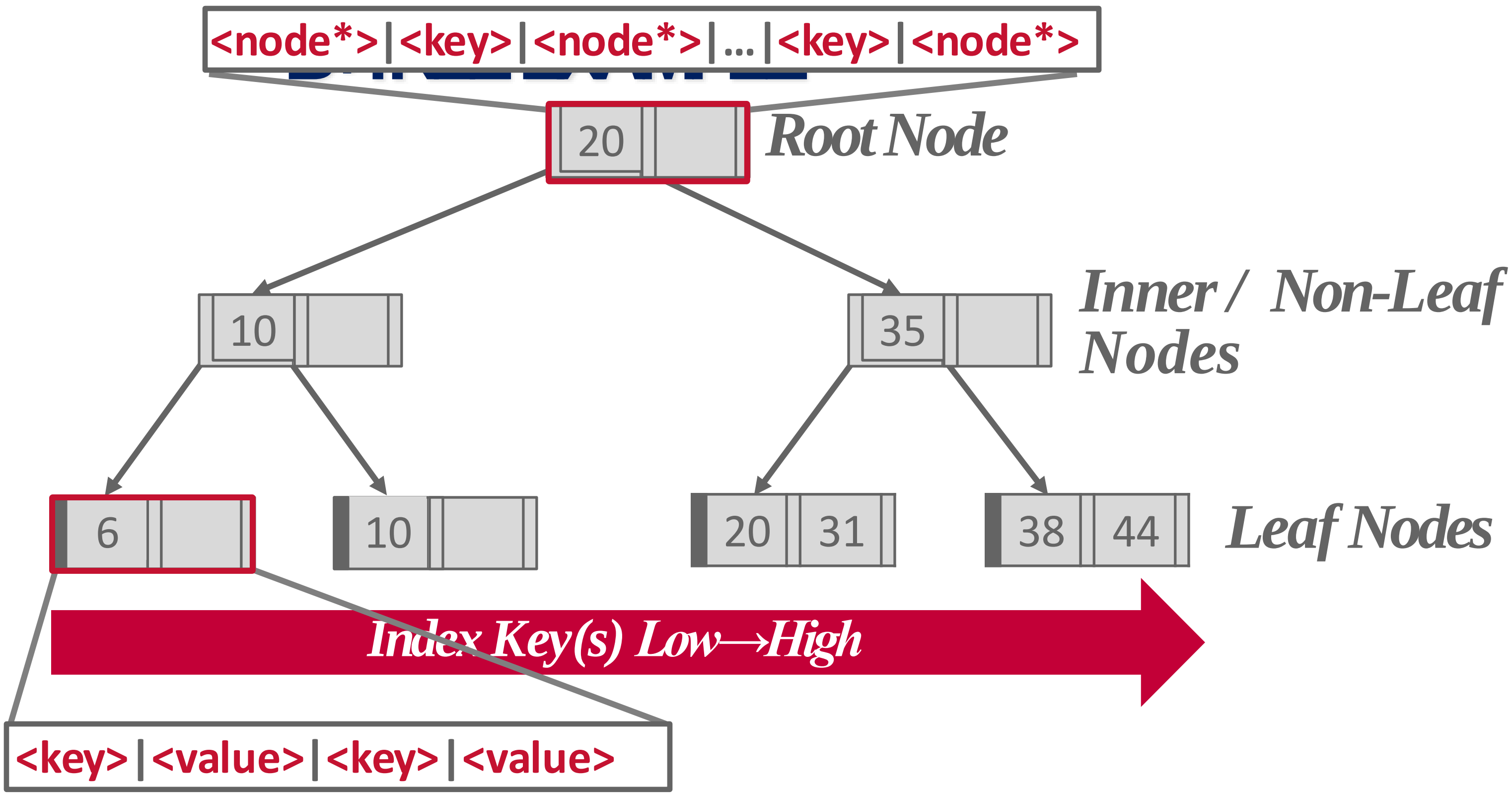
B+Tree



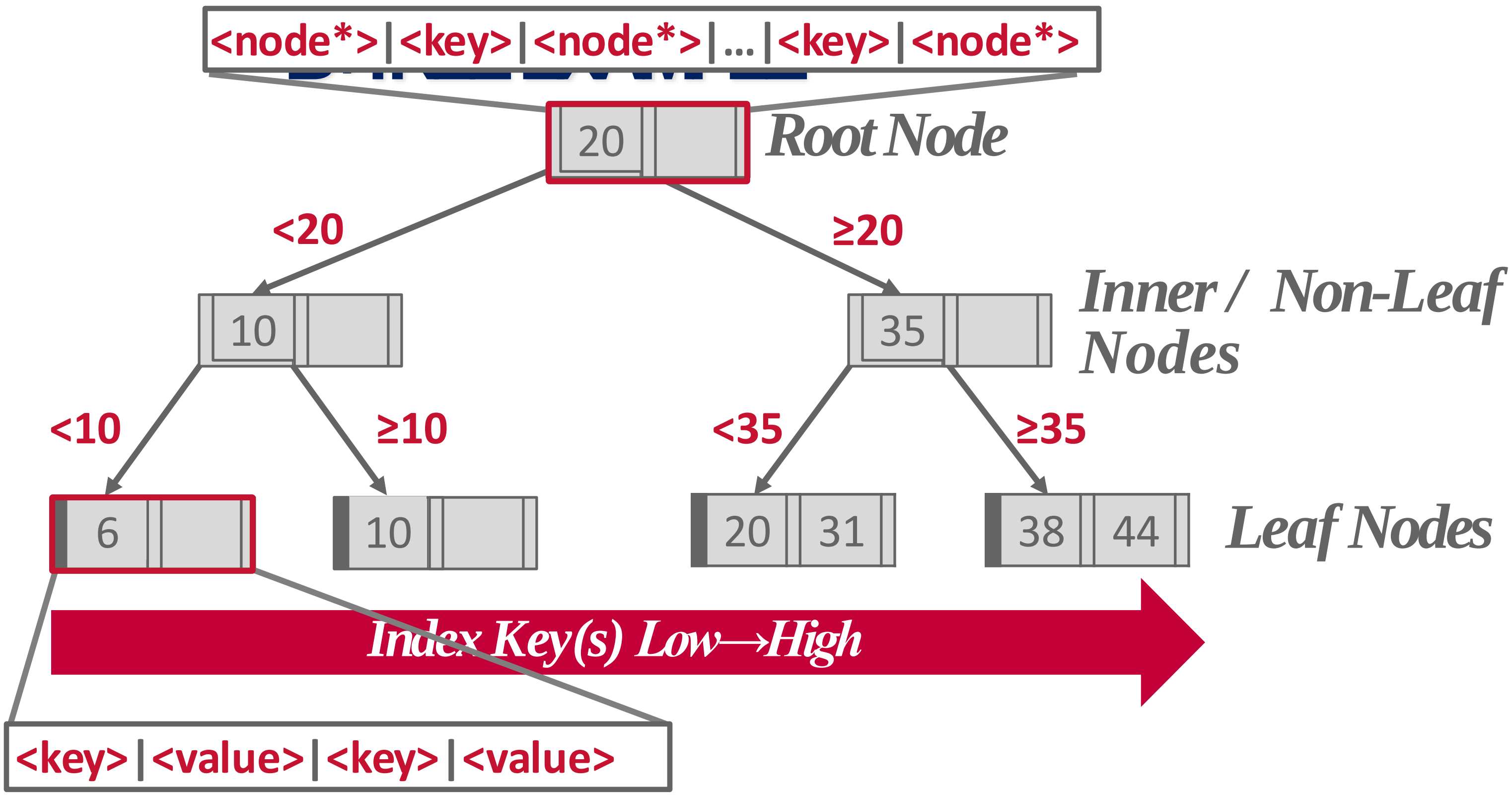

20
Root Node
10
35
Inner / Non-Leaf Nodes
6
10
20
31
38
44
Leaf Nodes


20
Root Node
10
35
Inner / Non-Leaf Nodes
6
10
20
31
38
44
Leaf Nodes
Index Key(s) Low→High

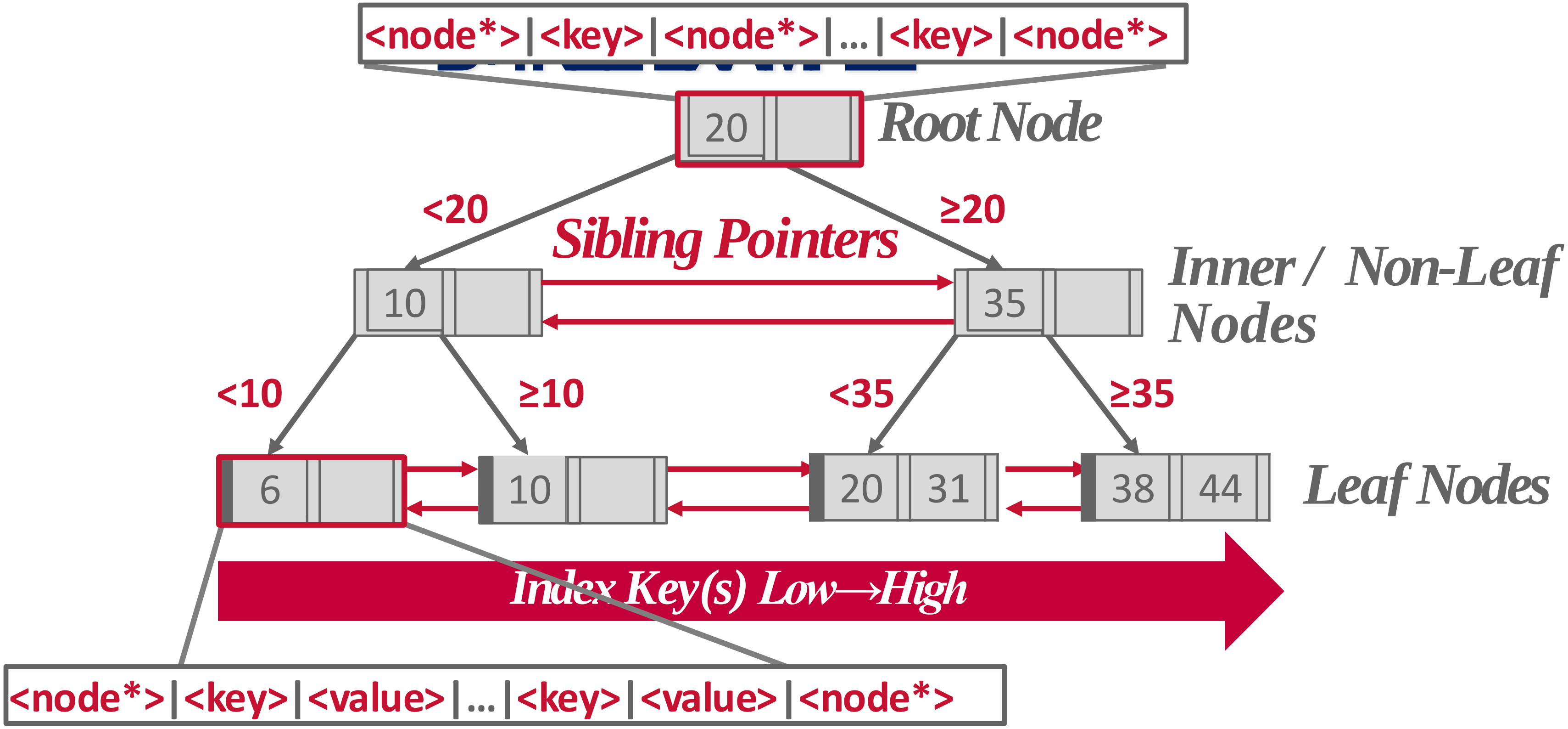

<node*>|<key>|<node*>|…|<key>|<node*>
20
Root Node
10
35
Inner / Non-Leaf Nodes
6
10
20 31
38 44
Leaf Nodes
Index Key(s) Low→High



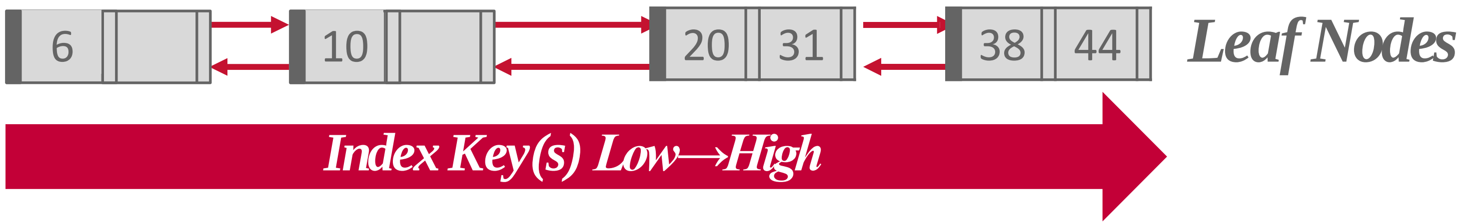
<node*>|<key>|<node*>|…|<key>|<node*>
20
Root Node
10
35
Inner / Non-Leaf Nodes
6
10
20
31
38
44
Leaf Nodes
Index Key(s) Low→High
<key>|<value>|<key>|<value>



<node*>|<key>|<node*>|...|<key>|<node*>
20
Root Node
<20
≥20
10
35
Inner / Non-Leaf Nodes
<10
≥10
<35
≥35
6
10
20
31
38
44
Leaf Nodes
Index Key(s) Low→High
<key>|<value>|<key>|<value>



<node*>|<key>|<node*>|…|<key>|<node*>
20
Root Node
<20
≥20
Sibling Pointers
10
35
Inner / Non-Leaf Nodes
<10
≥10
<35
≥35
6
10
20
31
38
44
Leaf Nodes
Index Key(s) Low→High
<node*>|<key>|<value>|…|<key>|<value>|<node*>


6
10
20
31
38
44
Leaf Nodes
Index Key(s) Low→High

Nodes

Every B+Tree node is comprised of an array of key/value pairs.

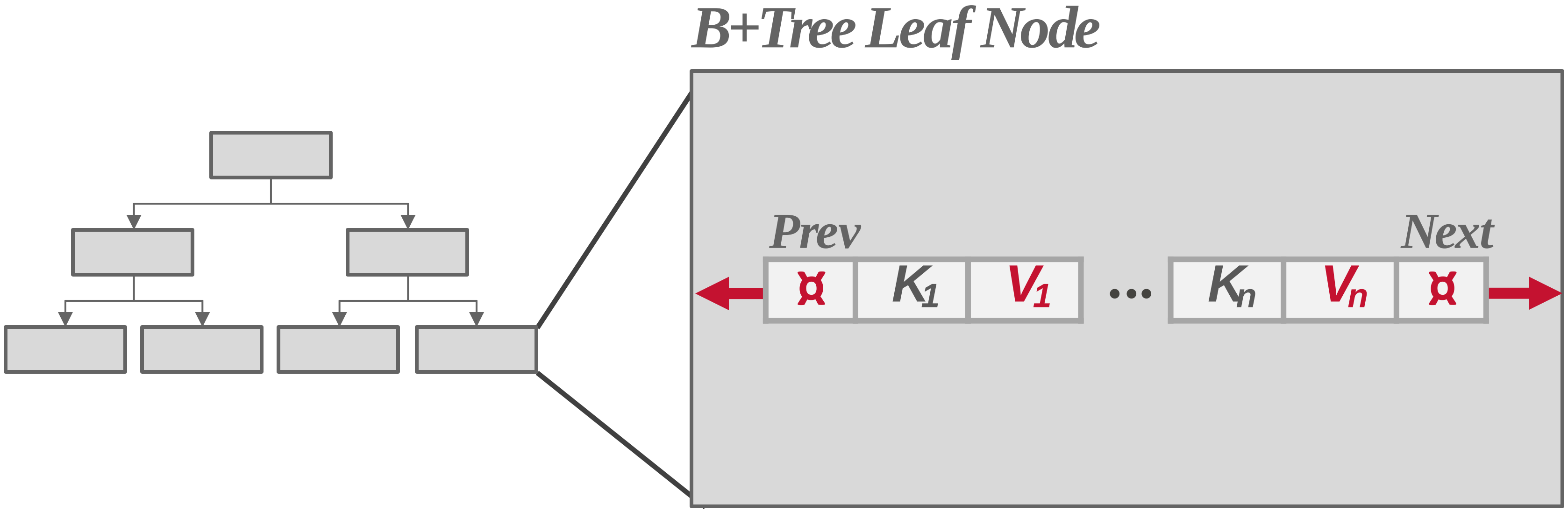
→ The keys are derived from the index's target attribute(s).

→ The values will differ based on whether the node is classified as an **inner node** or a **leaf node.**

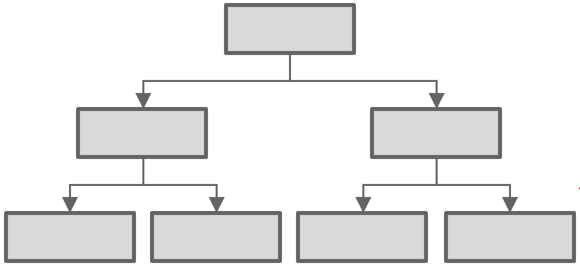
The arrays are (usually) kept in sorted key order.

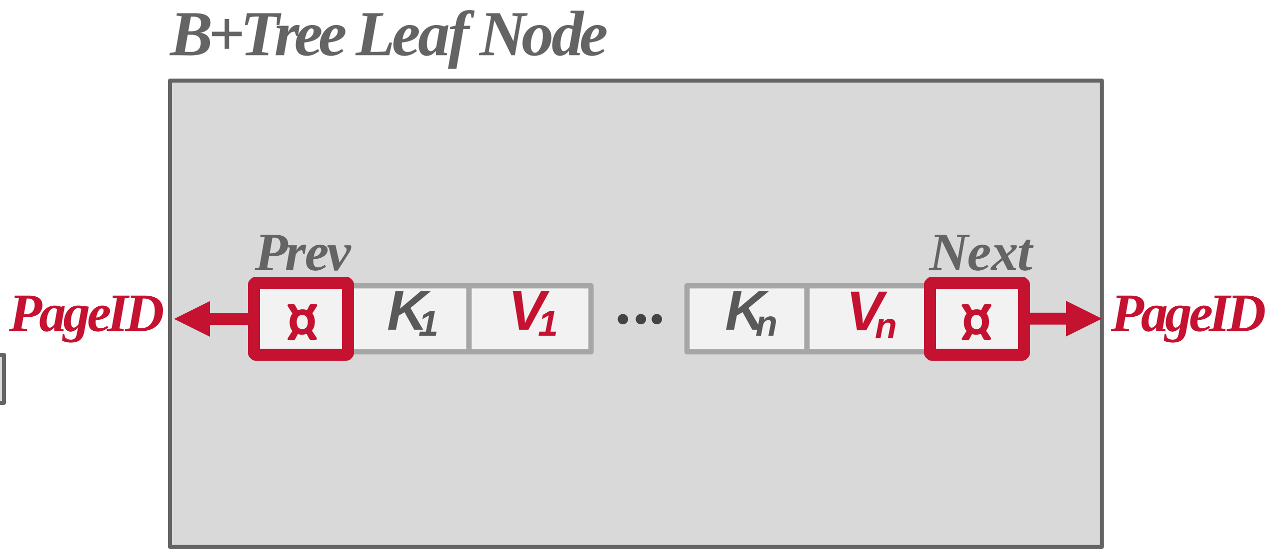
Store all **NULL** keys at either first or last leaf nodes.

B+Tree Leaf Nodes

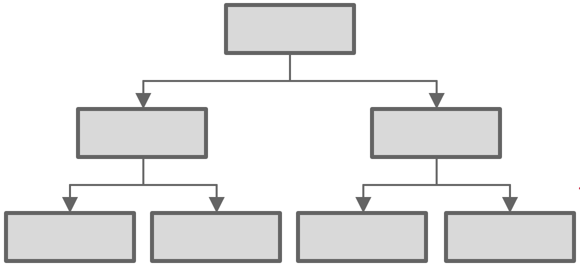


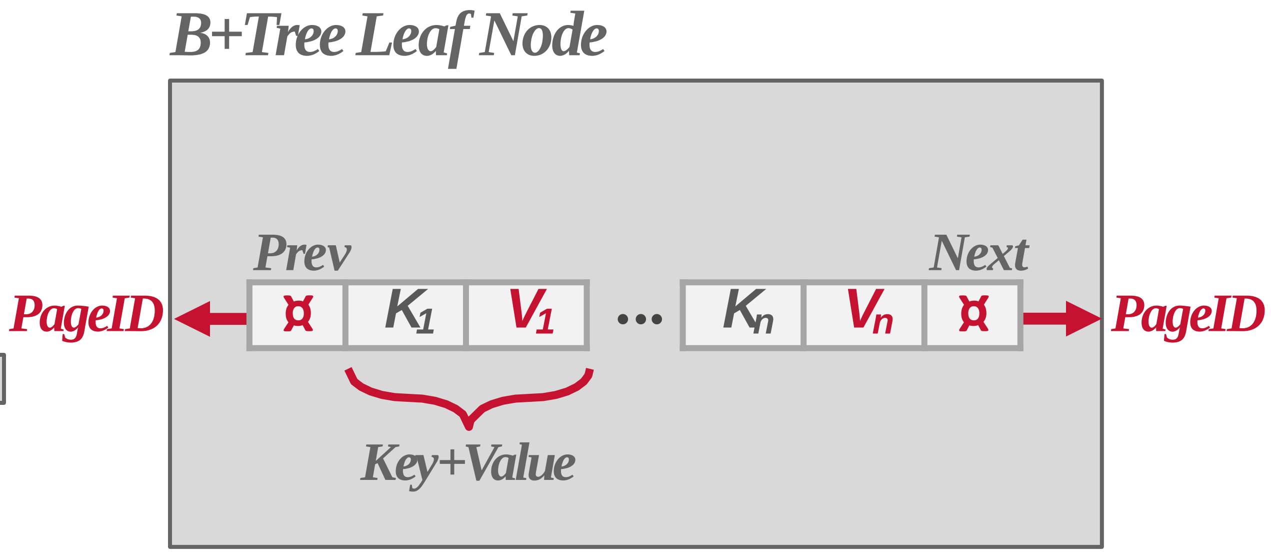
B+Tree Leaf Nodes



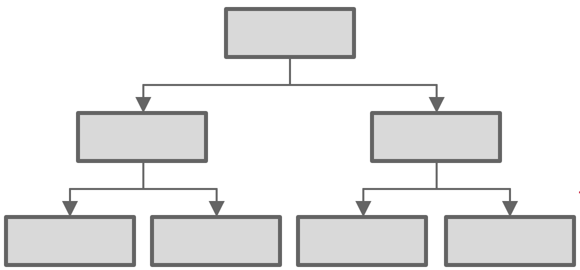


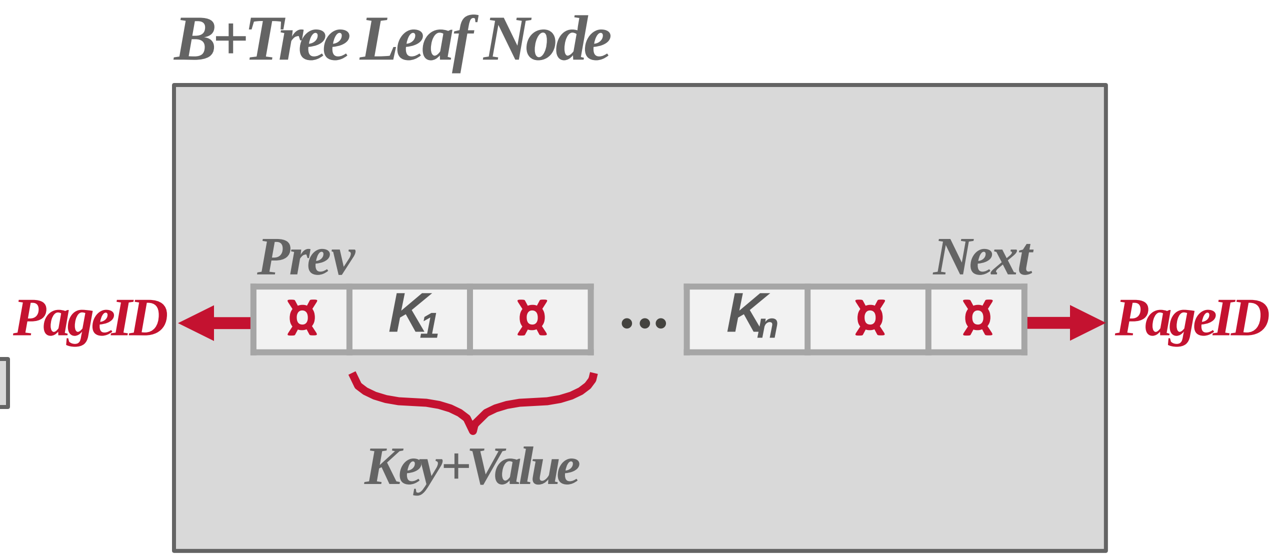
B+Tree Leaf Nodes



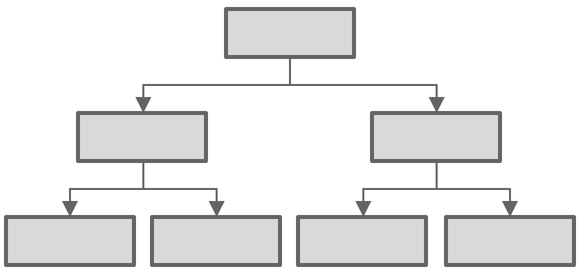


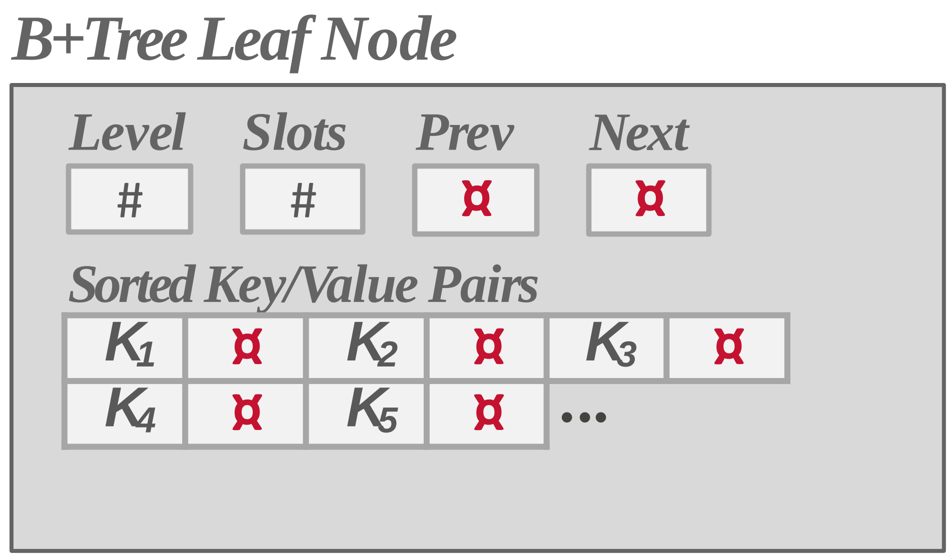
B+Tree Leaf Nodes



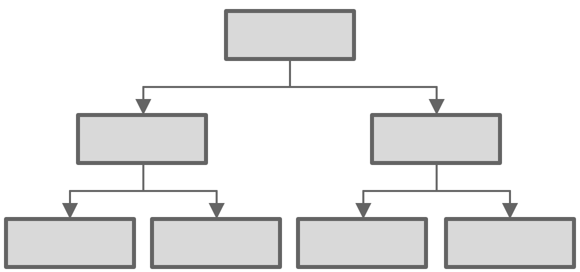


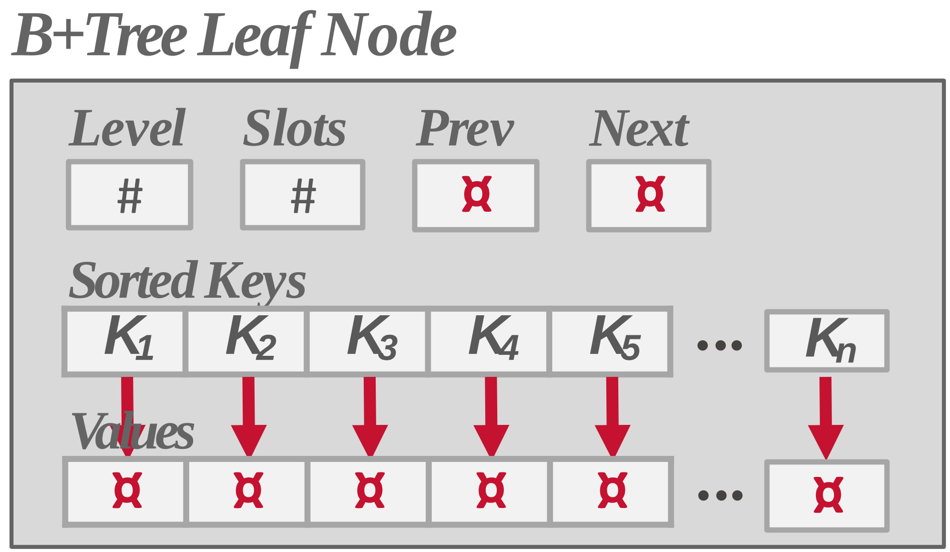
B+Tree Leaf Nodes





B+Tree Leaf Nodes





Leaf Node Values

Approach #1: Record IDs

→ A pointer to the location of the tuple to which the index entry corresponds.

→ Most common implementation.

Approach #2: Tuple Data

→ Index-Organized Storage

→ Primary Key Index: Leaf nodes store the contents of the tuple.

→ Secondary Indexes: Leaf nodes store tuples' primary key as their values.

B-Tree VS B+Tree

- The original B-Tree from 1971 stored keys and values in all nodes in the tree.
  - More space-efficient, since each key only appears once in the tree.

- A B+Tree only stores values in leaf nodes. Inner nodes only guide the search process.

B+Tree Insert

Find correct leaf node **L**.

Insert data entry into **L** in sorted order.

If **L** has enough space, done!

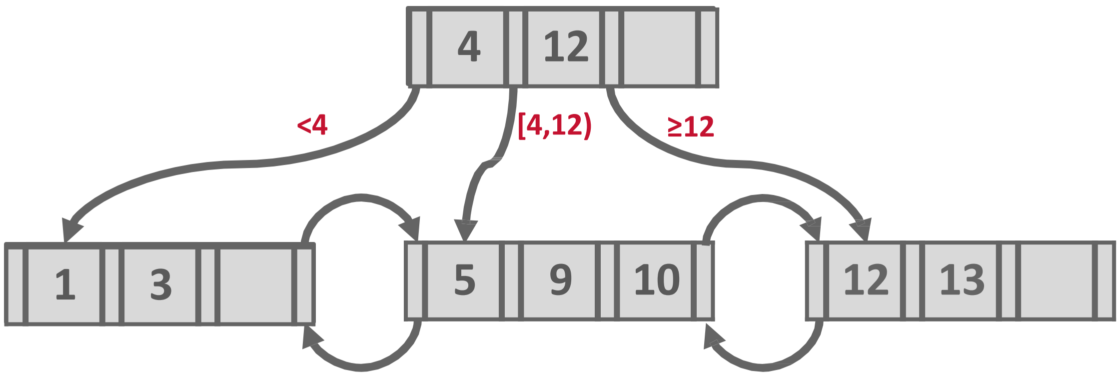
Otherwise, split **L** keys into **L** and a new node $\mathbf{L_2}$

→ Redistribute entries evenly, copy up middle key.

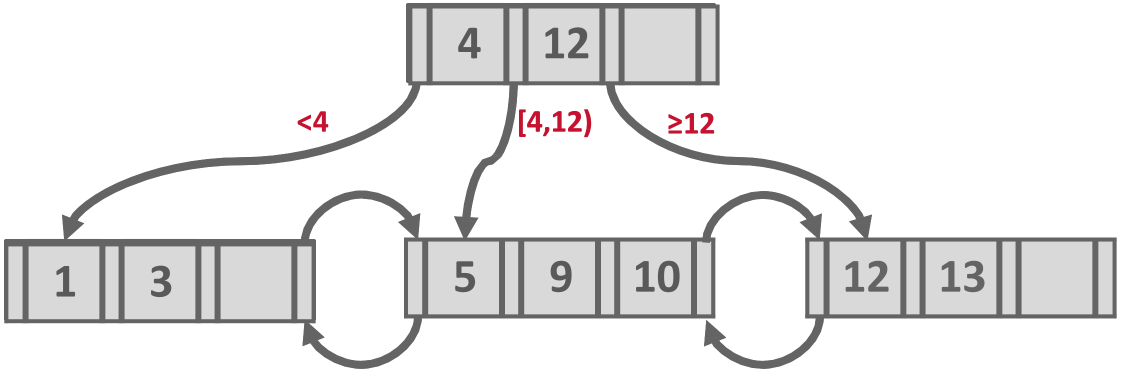
→ Insert index entry pointing to $\mathbf{L_2}$ into the parent of **L**.

To split inner node, redistribute entries evenly, but push up middle key.

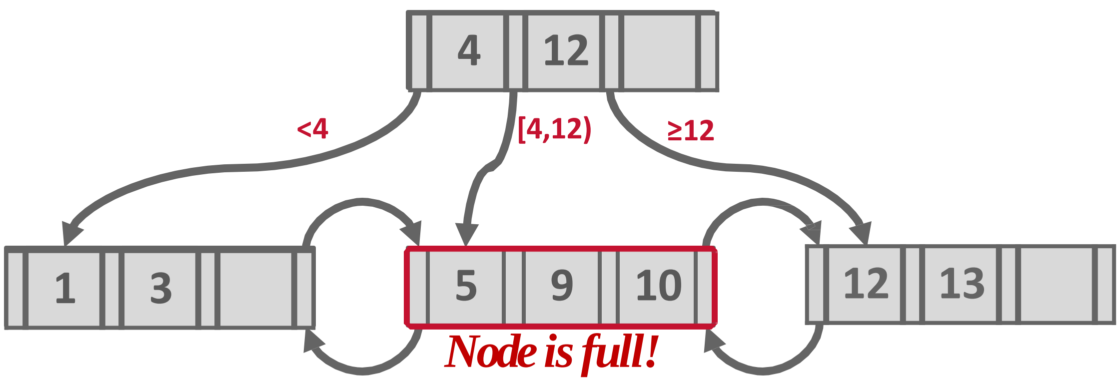
B+Tree Insert



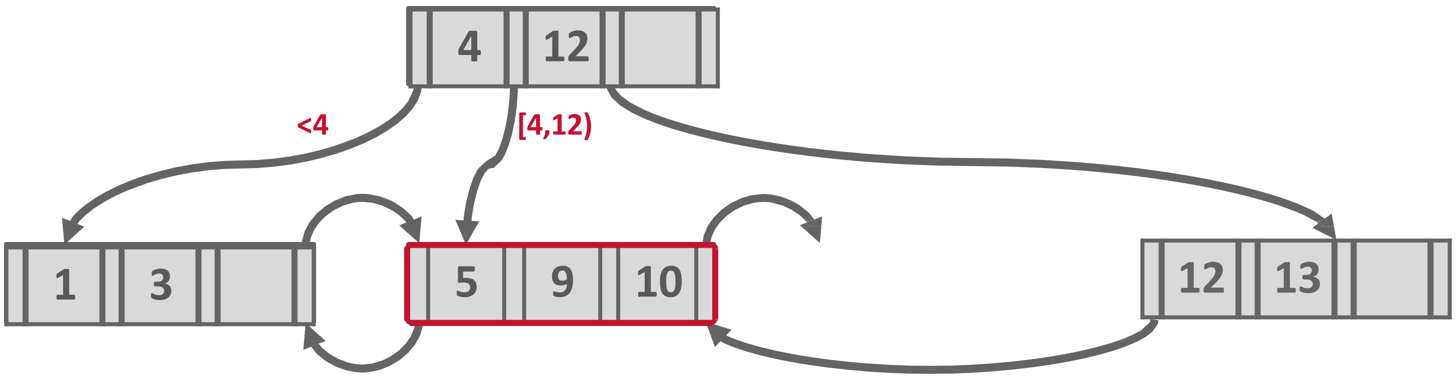
Insert 6



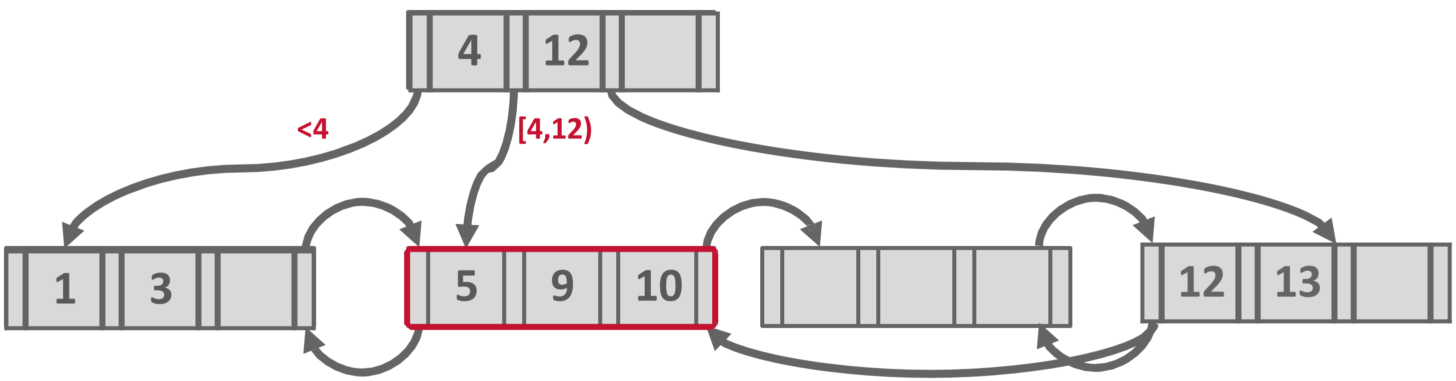
Insert 6



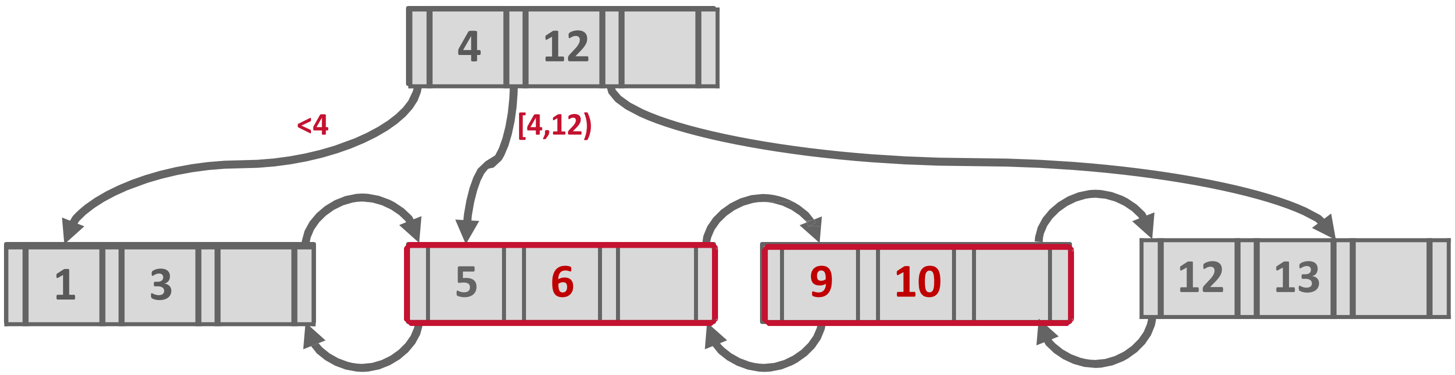
Insert 6



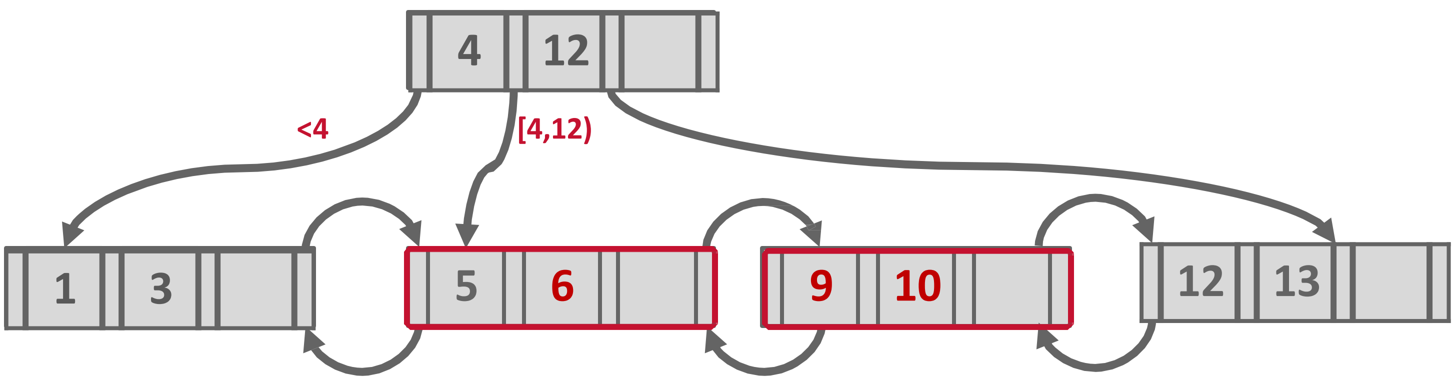
Insert 6



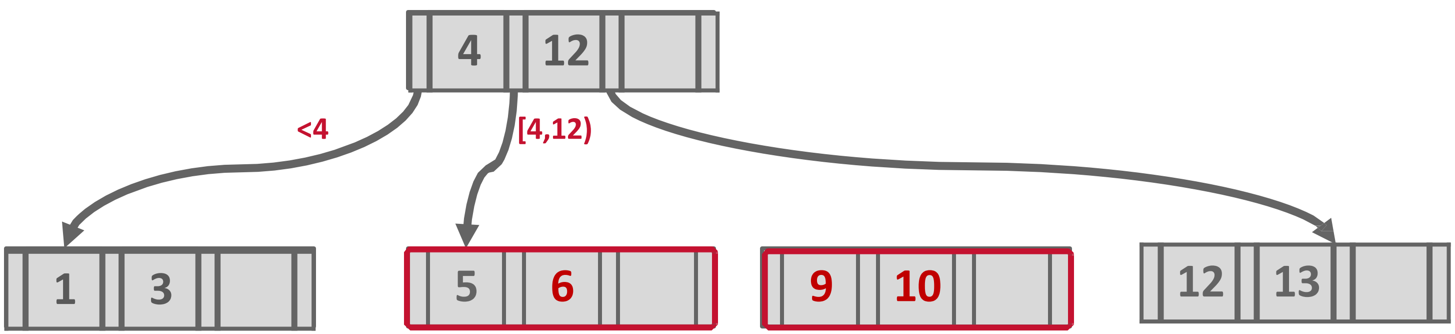
Insert 6



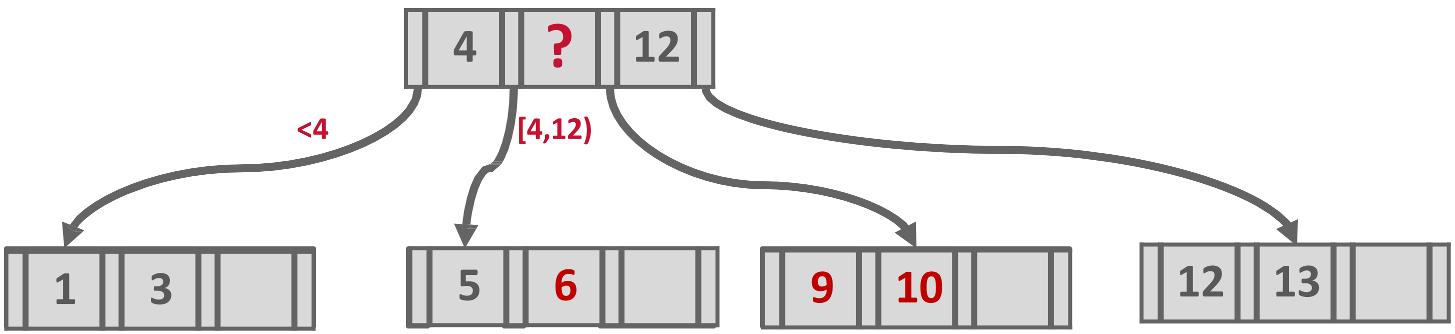
Insert 6



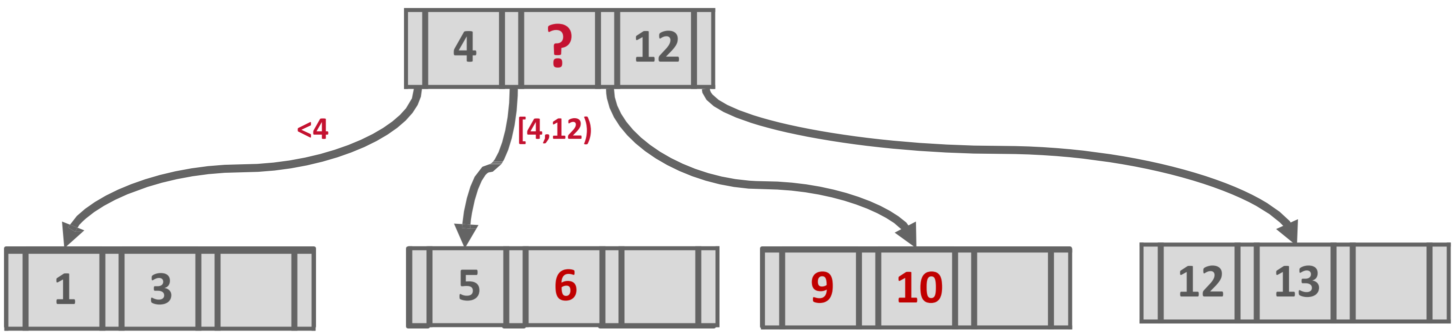
Insert 6



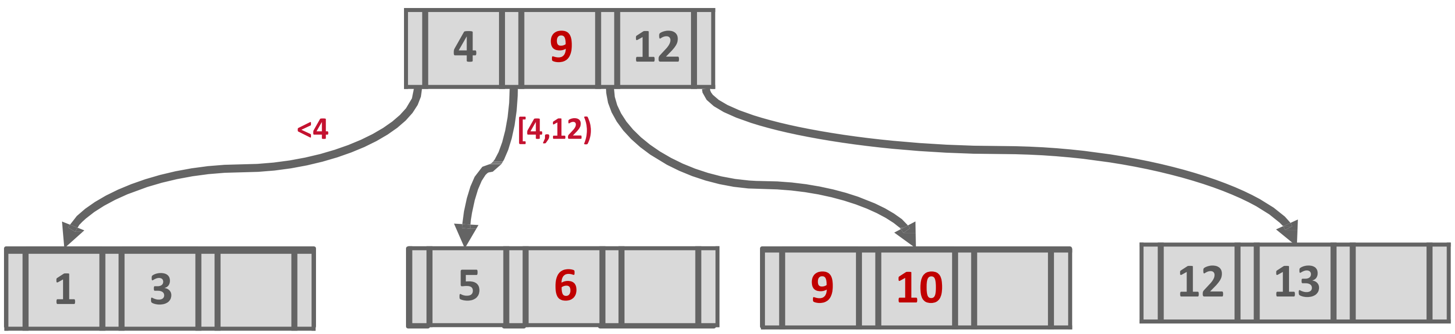
Insert 6



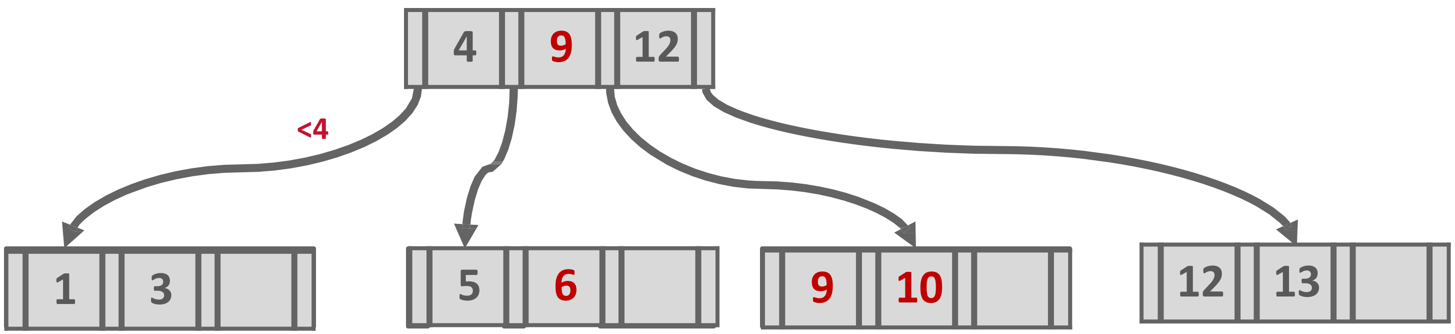
Insert 6



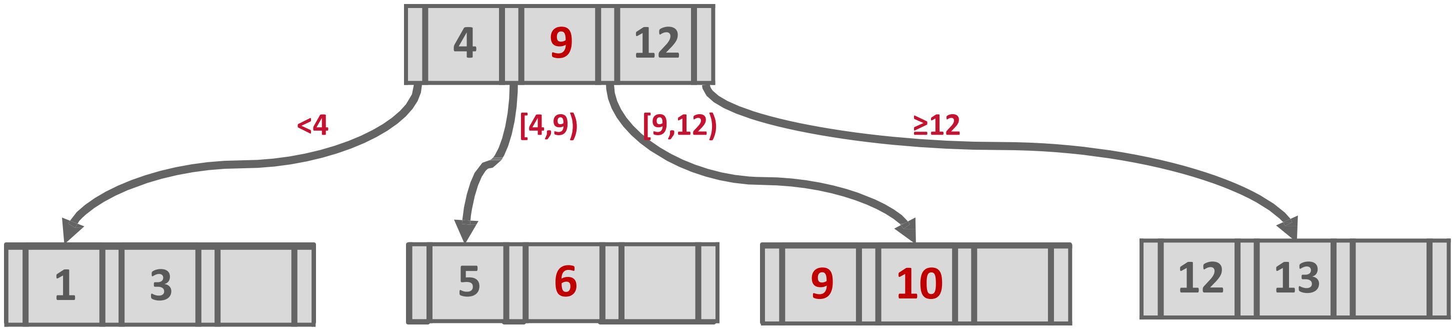
Insert 6



Insert 6

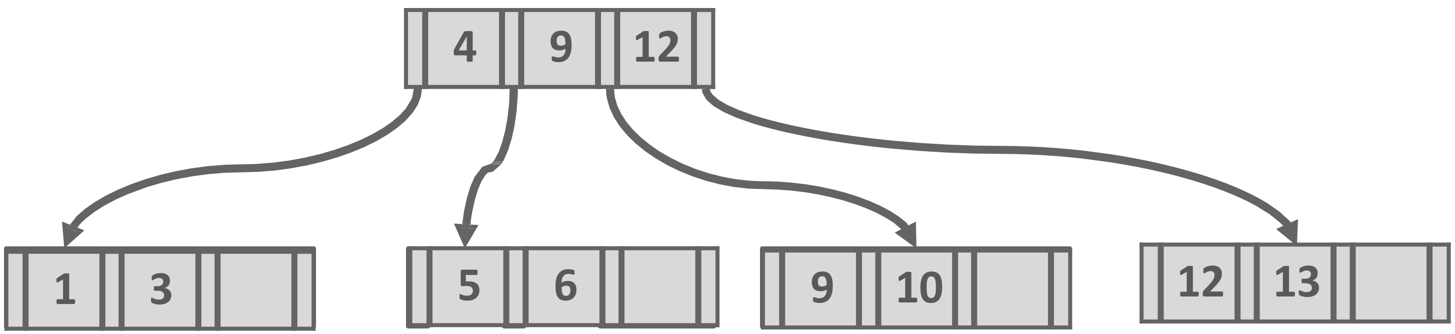


Insert 6



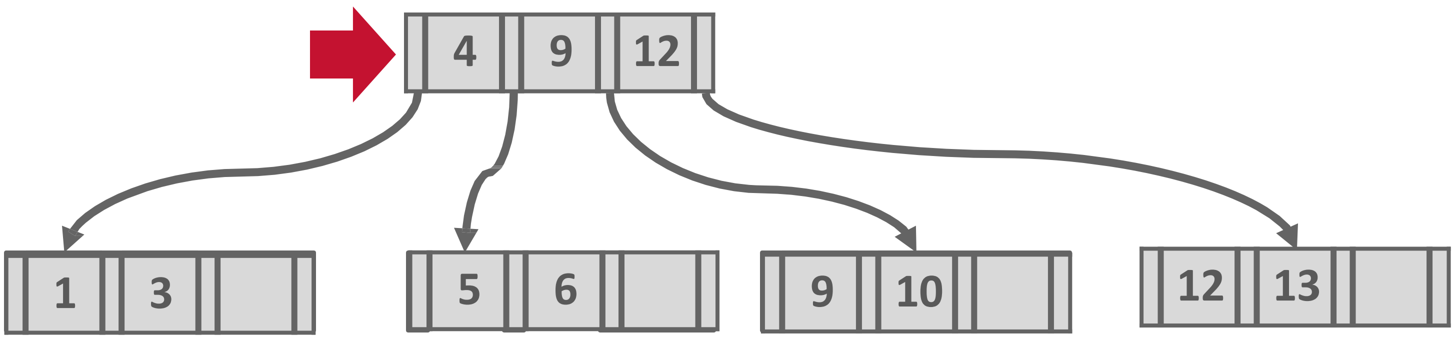
Insert 6

Insert 8



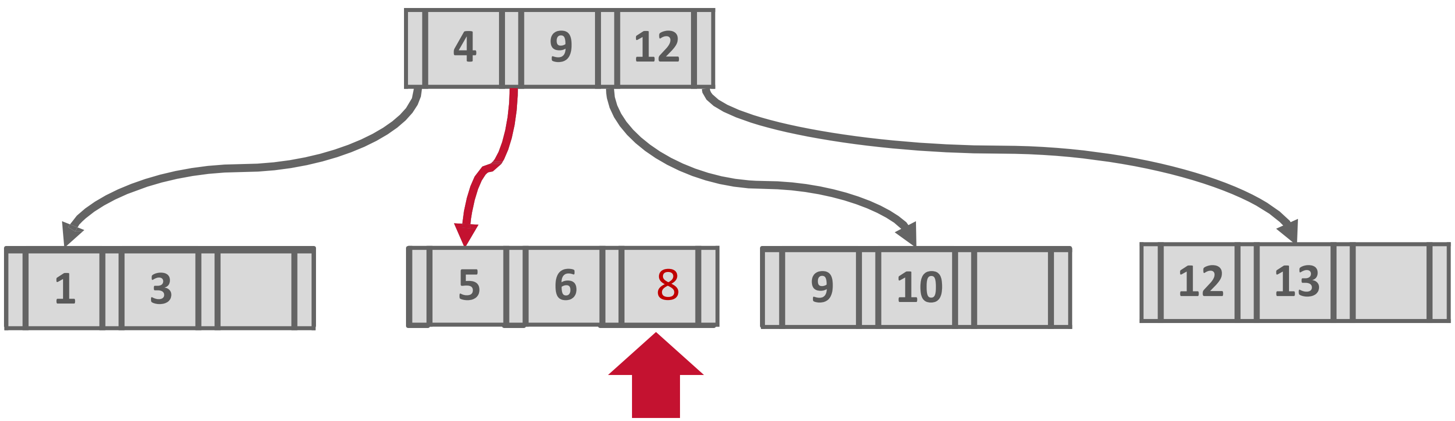
Insert 6

Insert 8

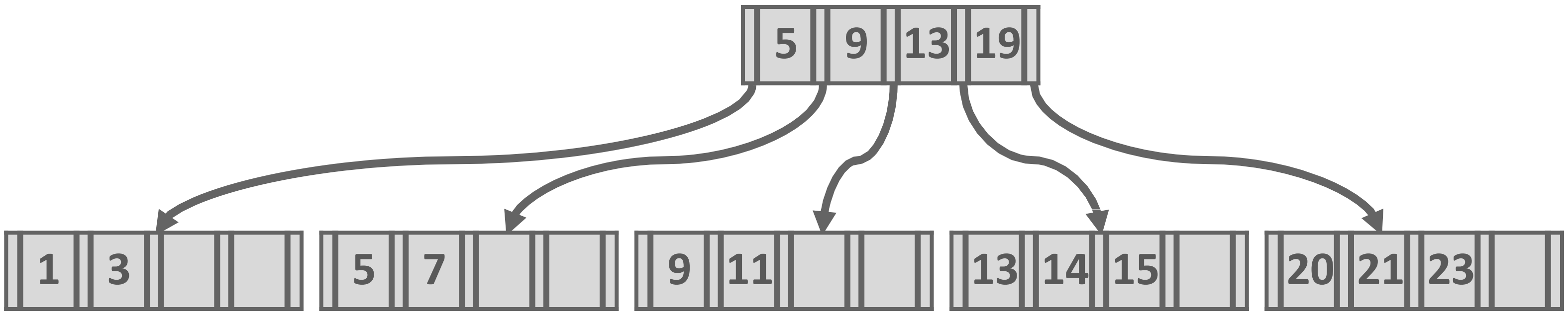


Insert 6

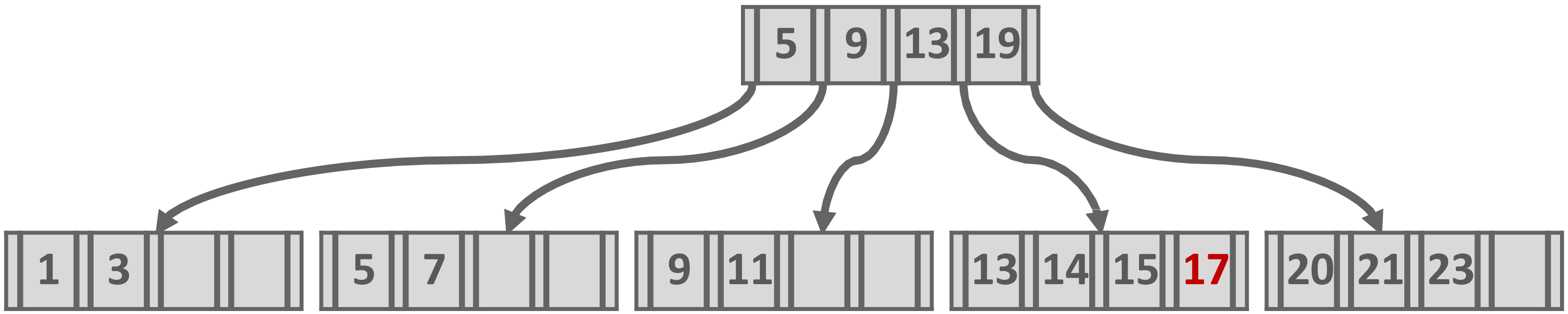
Insert 8



Insert 17

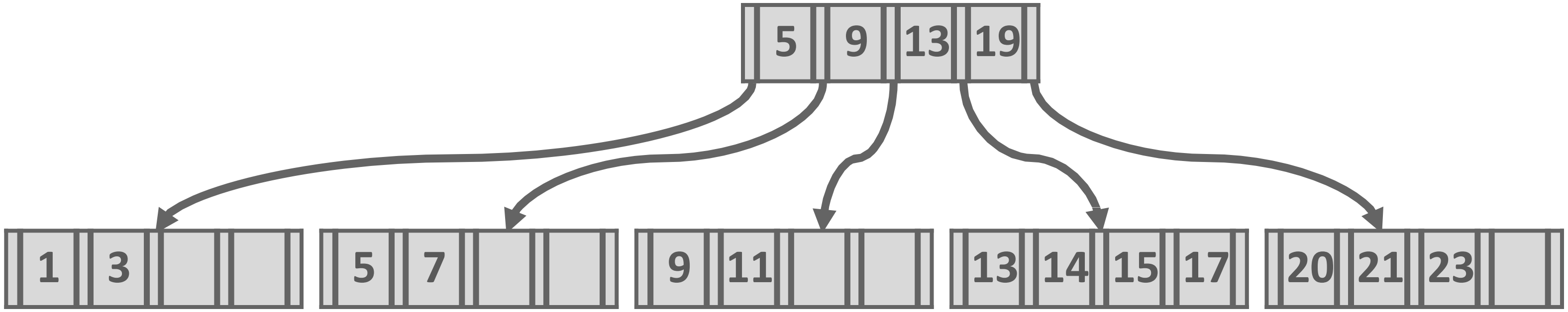


Insert 17



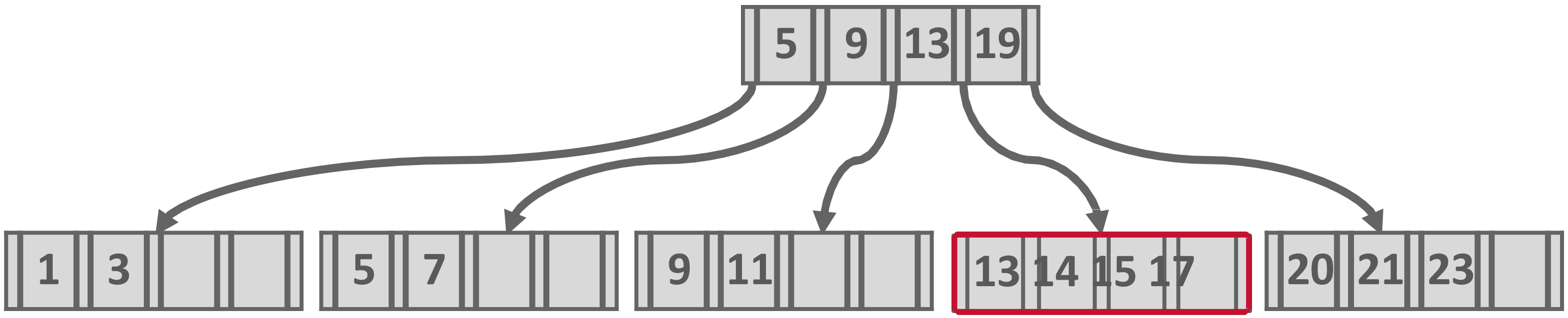
Insert 17

Insert 16



Insert 17

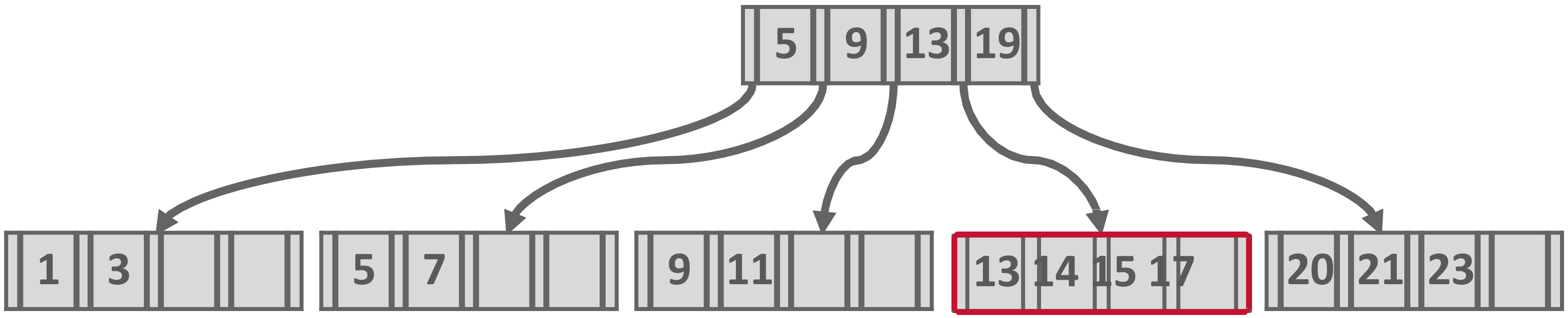
Insert 16



No space in the node where the new key "belongs".

Insert 17

Insert 16



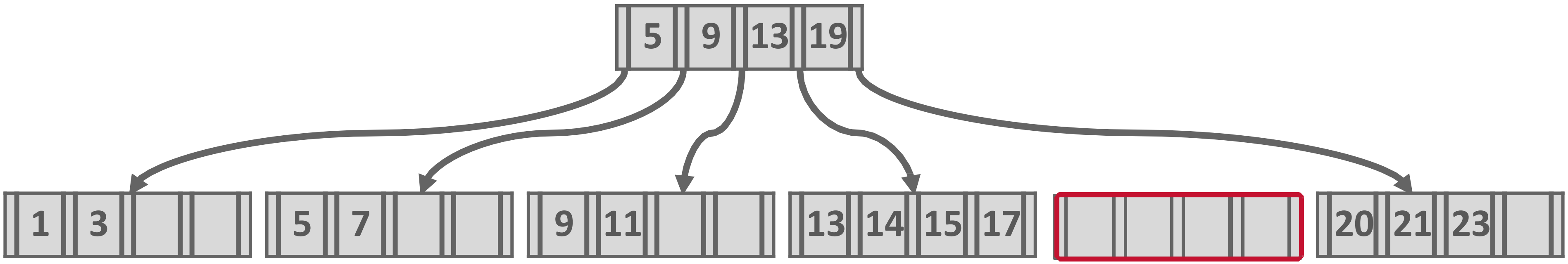
Split the node!

Copy the middle key.

Push the key up.

Insert 17

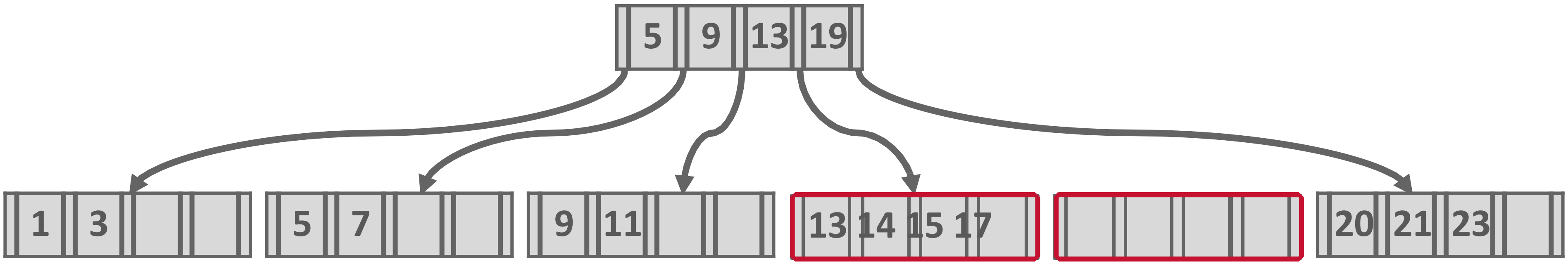
Insert 16



New Node!
Shuffle keys from the node that triggered the split.

Insert 17

Insert 16

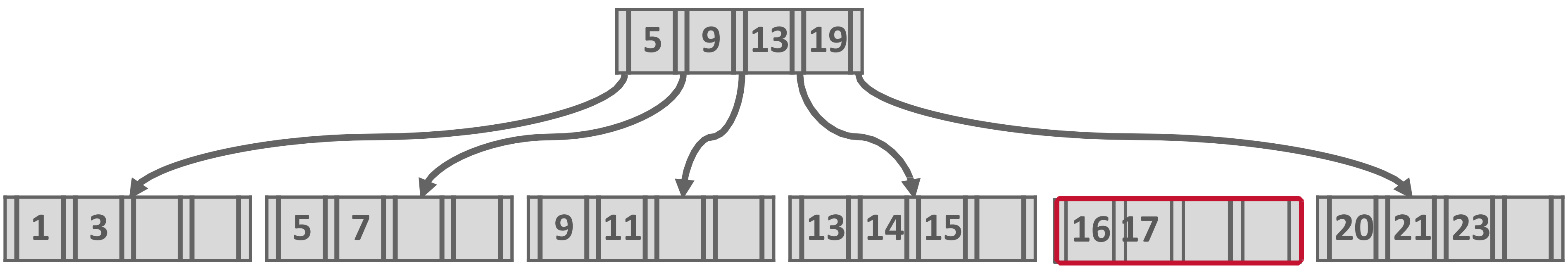


New Node!

Shuffle keys from the node that triggered the split.

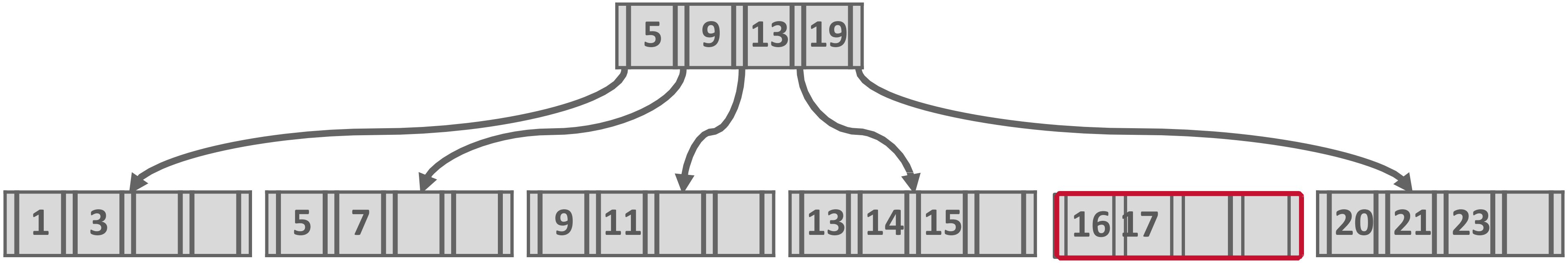
Insert 17

Insert 16



Insert 17

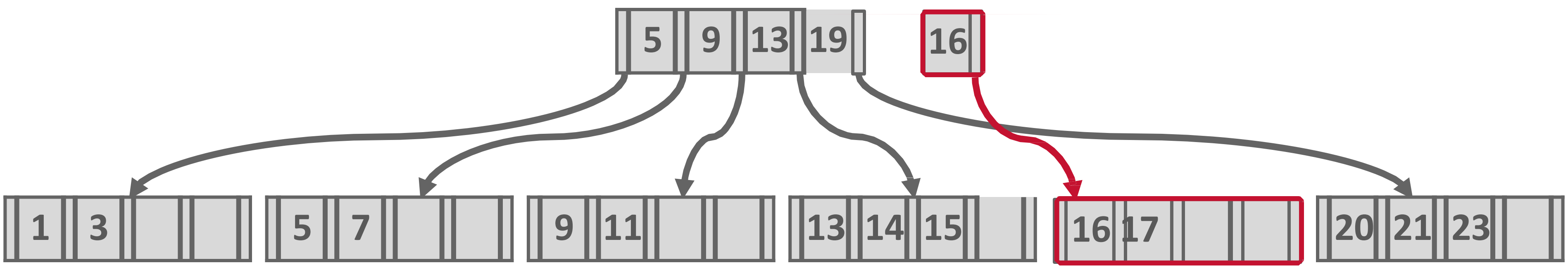
Insert 16



But this is an "orphan" node!
No parent node points to it.

Insert 17

Insert 16



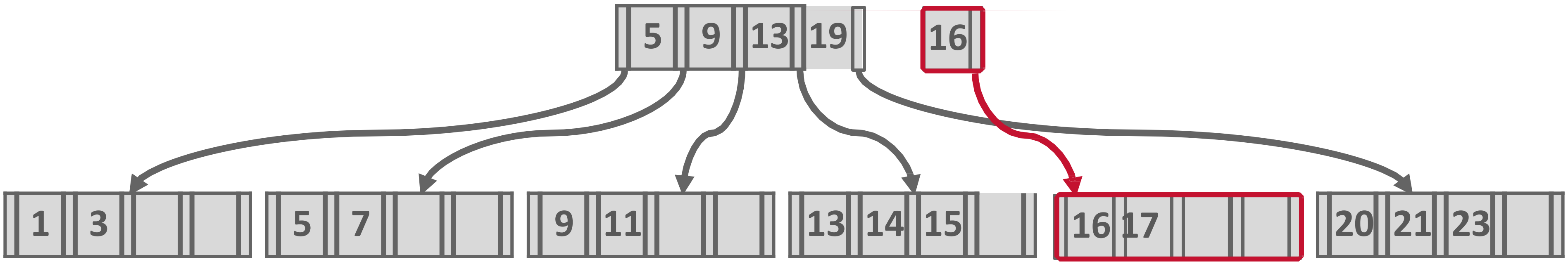
But this is an "orphan" node!

No parent node points to it.

Insert 17

Insert 16

Want to create a key, pointer pair like this. But cannot insert it in the root node, which is full.

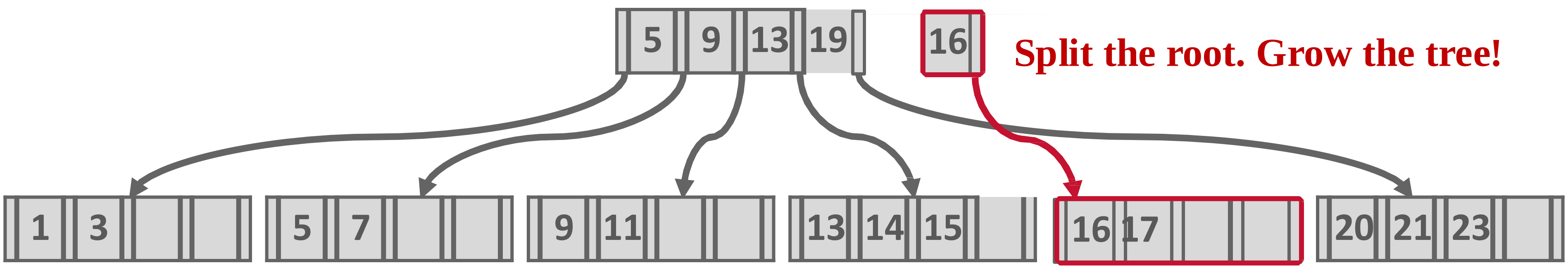


But this is an "orphan" node! No parent node points to it.

Insert 17

Insert 16

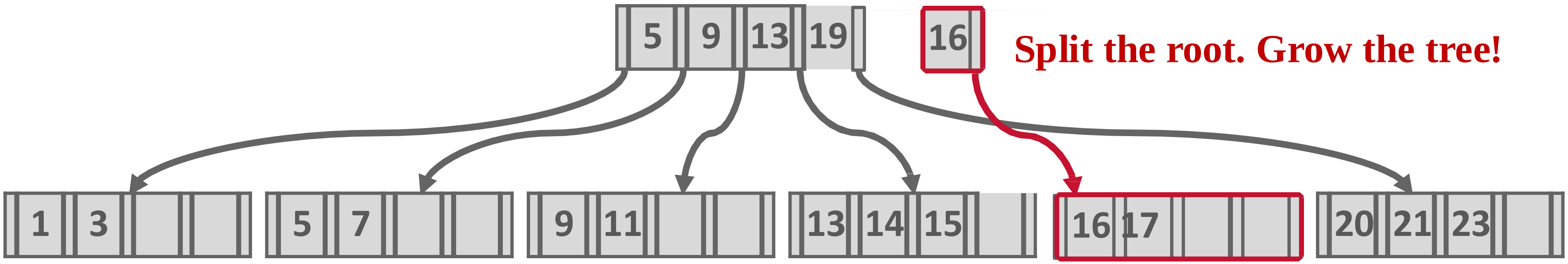
Want to create a key, pointer pair like this. But cannot insert it in the root node, which is full.



But this is an "orphan" node! No parent node points to it.

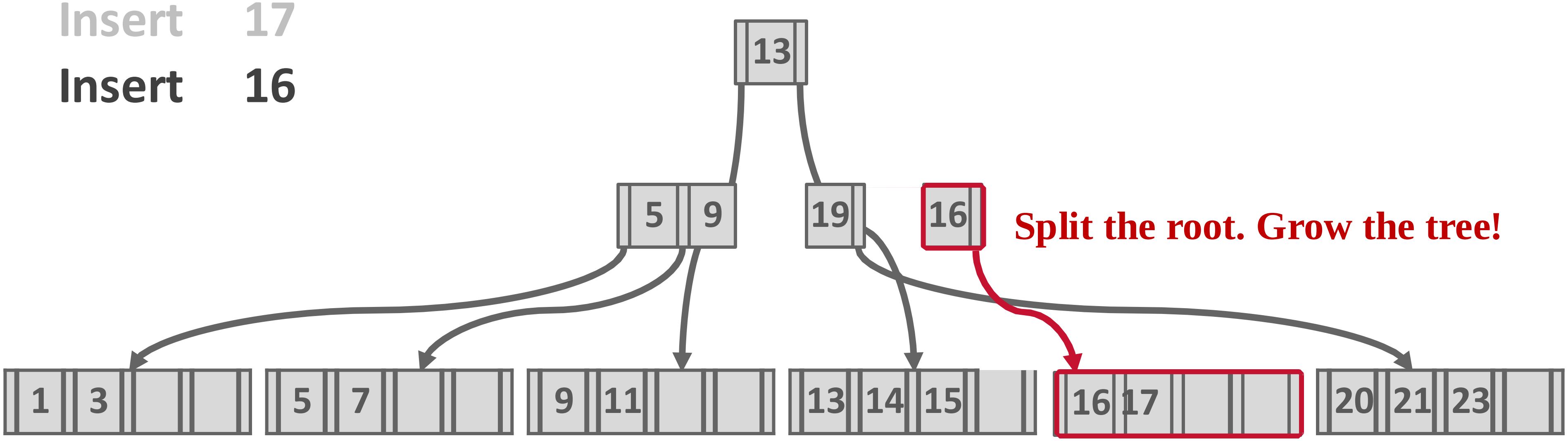
Insert 17

Insert 16



Insert 17

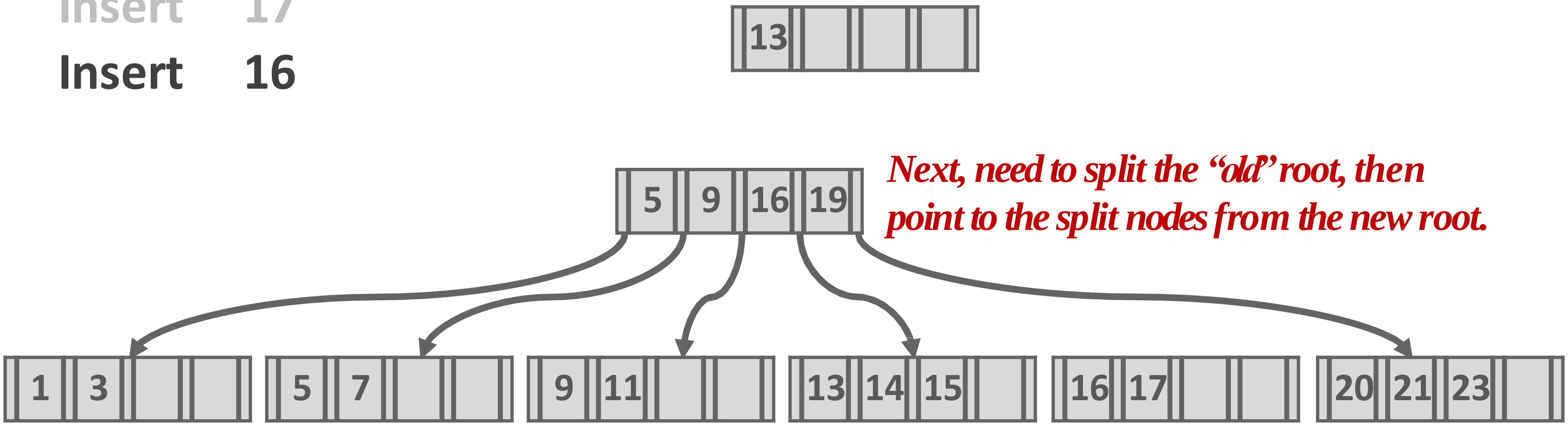
Insert 16



Split the root. Grow the tree!

Insert 17

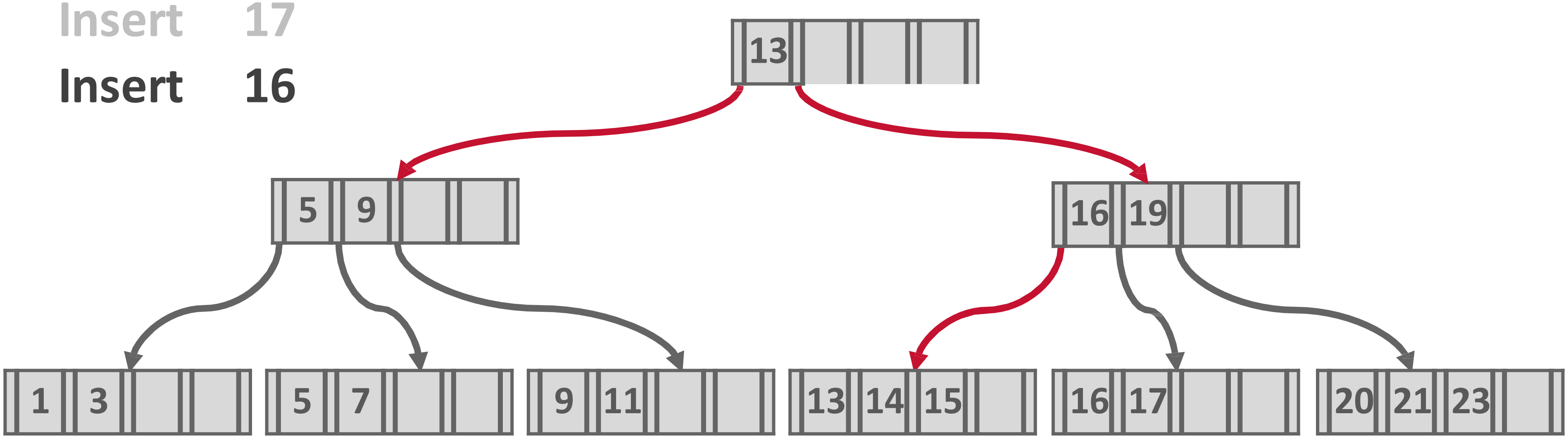
Insert 16



Next, need to split the "old" root, then point to the split nodes from the new root.

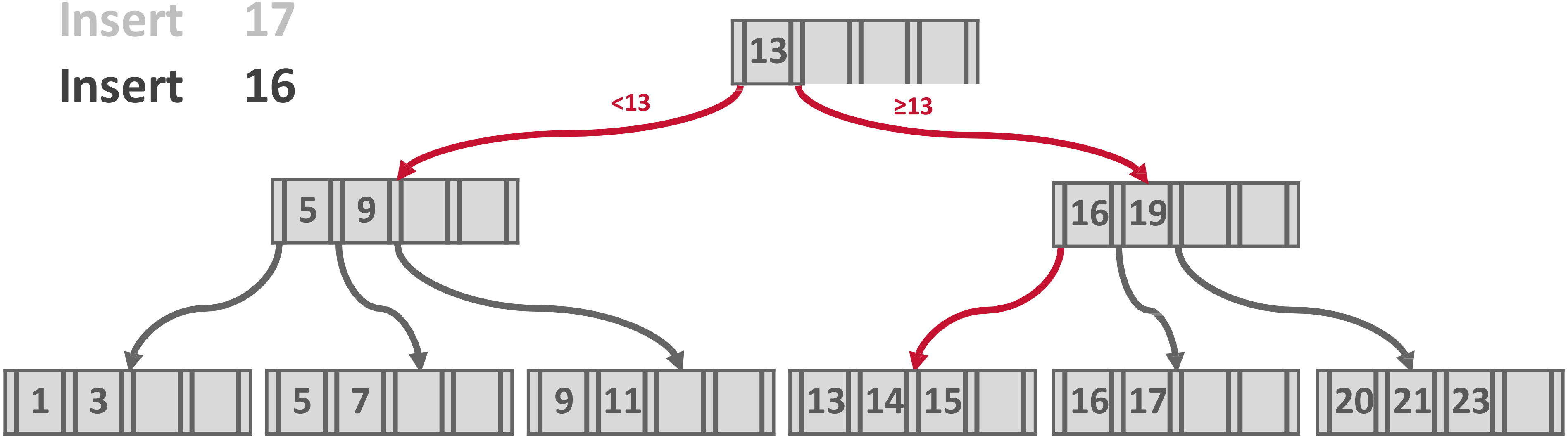
Insert 17

Insert 16



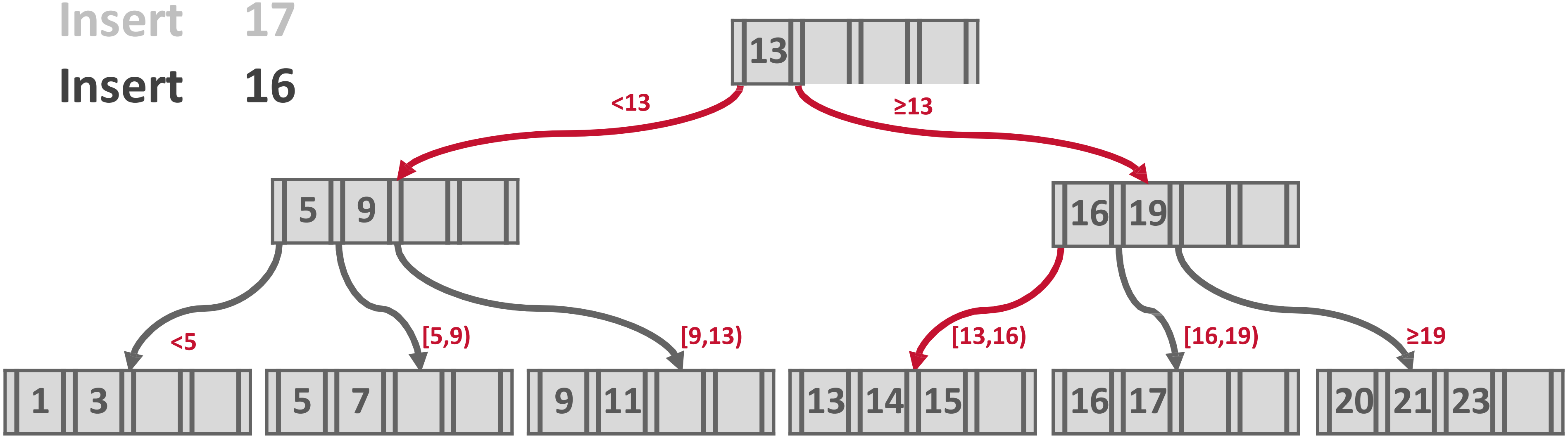
Insert 17

Insert 16



Insert 17

Insert 16



Exercises

1. If index entries are inserted in sorted order, what will be the occupancy of each leaf node in a B+-tree? Explain why.

2. If the fanout of the B+-tree is *m* and its height is 3, what are:
    1. The maximum number of leaf nodes?
    2. The minimum number of leaf nodes?

B+Tree Delete

Start at root, find leaf **L** where entry belongs. Remove the entry.

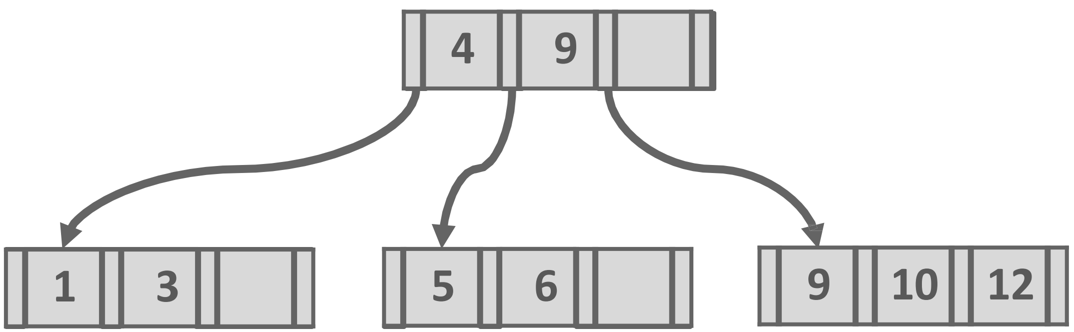
If **L** is at least half-full, done! If **L** has only **m/2-1** entries (recall that ***m*** is the tree fanout),

→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).

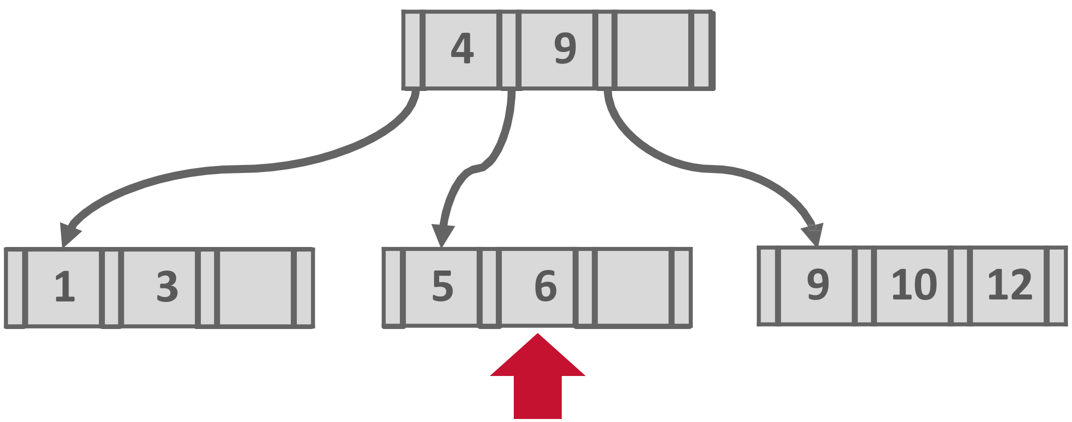
→ If re-distribution fails, merge **L** and sibling.

If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.

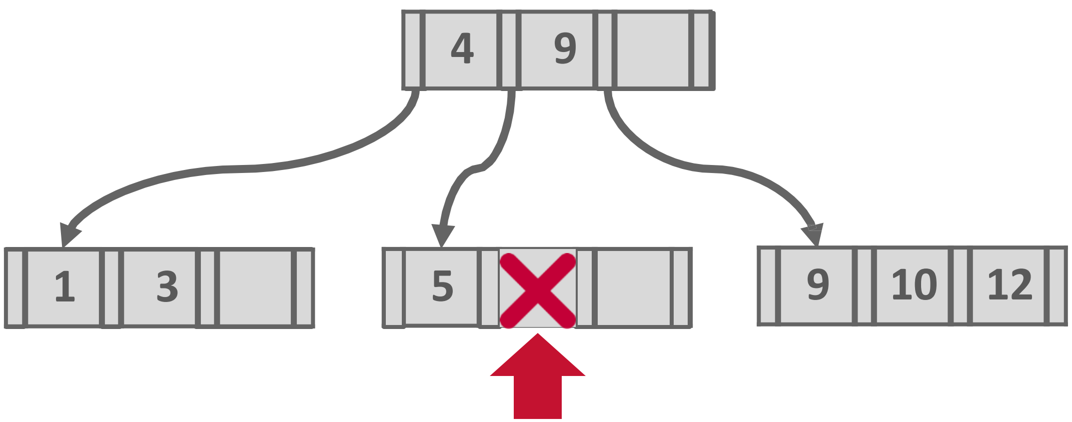
Delete 6



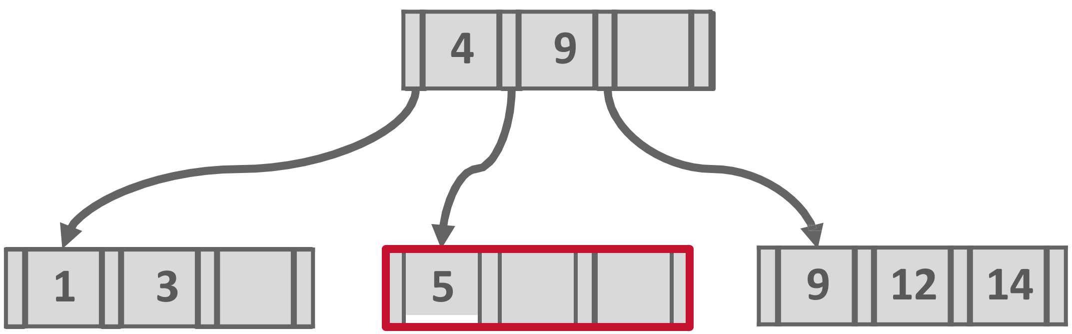
Delete 6



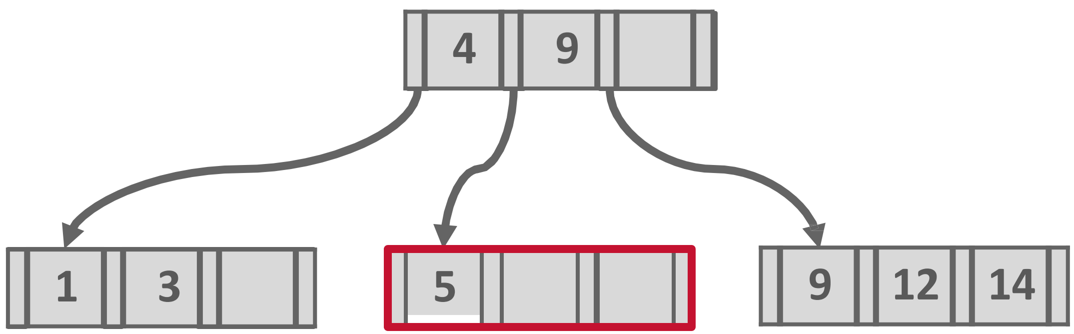
Delete 6



Delete 6



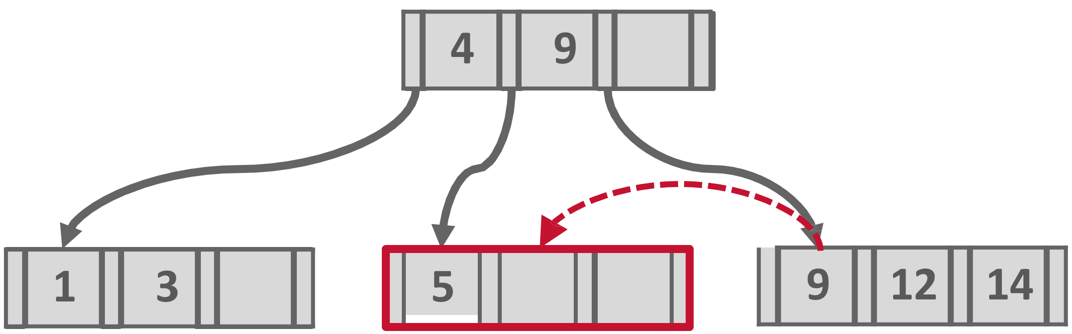
Delete 6



Borrow from a "rich" sibling node.

Could borrow from either sibling.

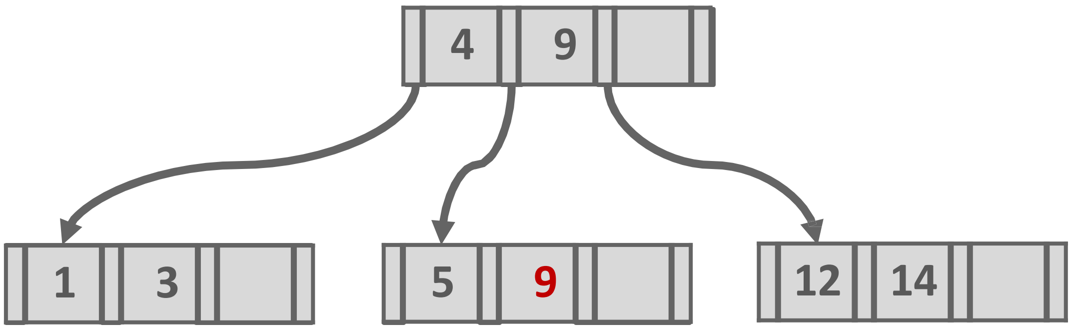
Delete 6



Borrow from a "rich" sibling node.

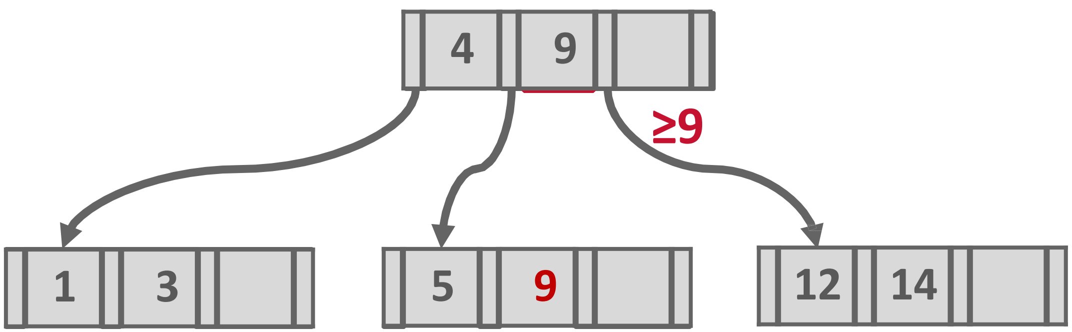
Could borrow from either sibling.

Delete 6



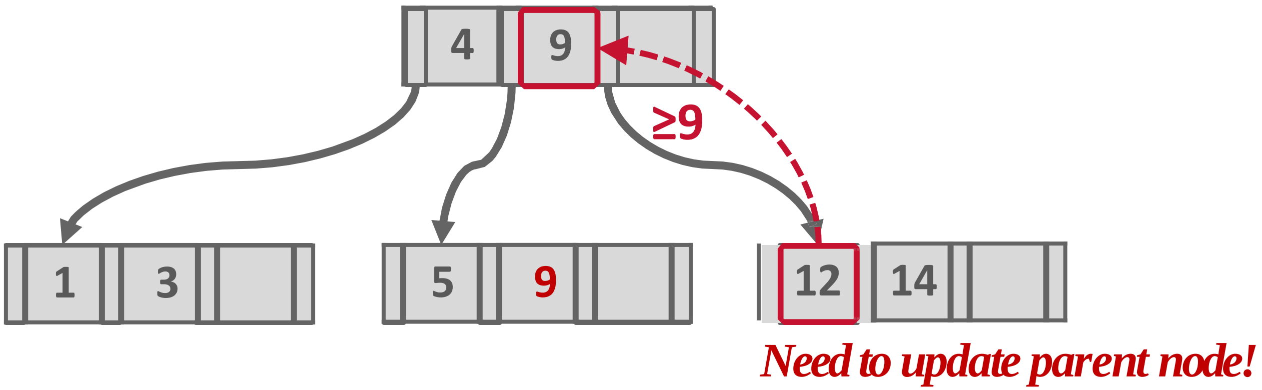
Need to update parent node!

Delete 6

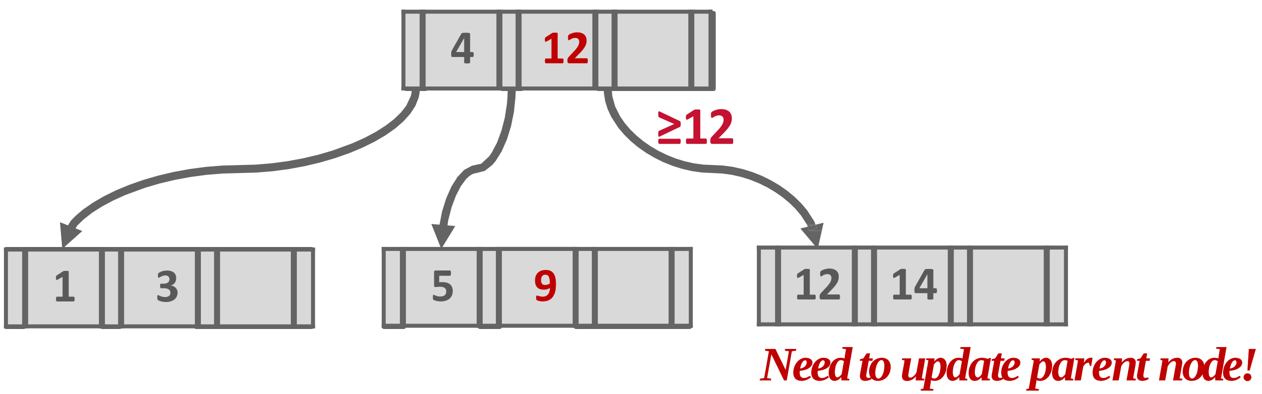


Need to update parent node!

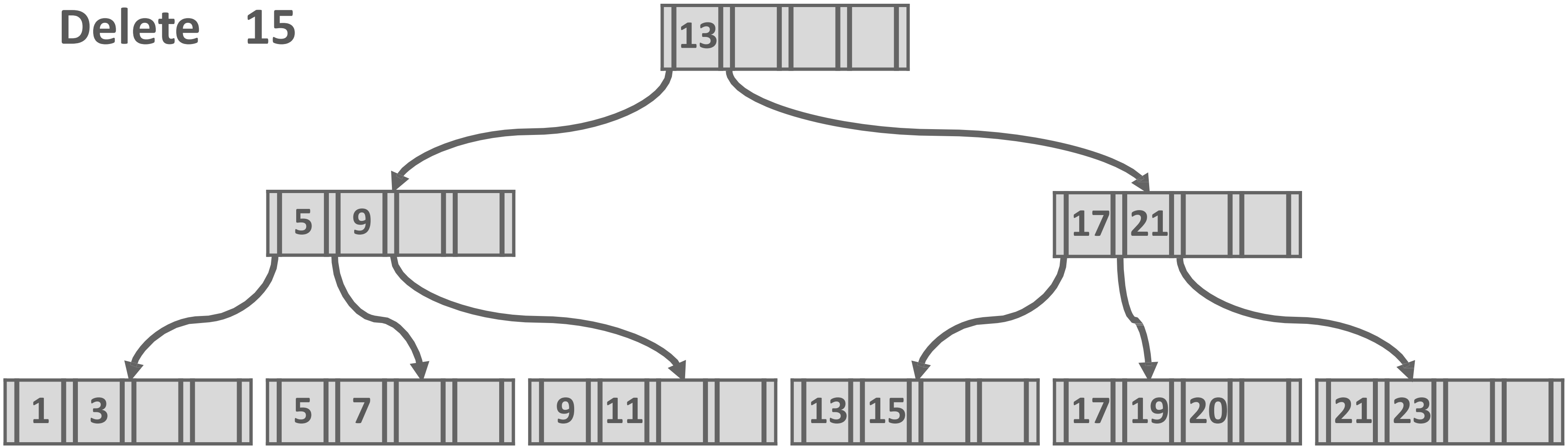
Delete 6



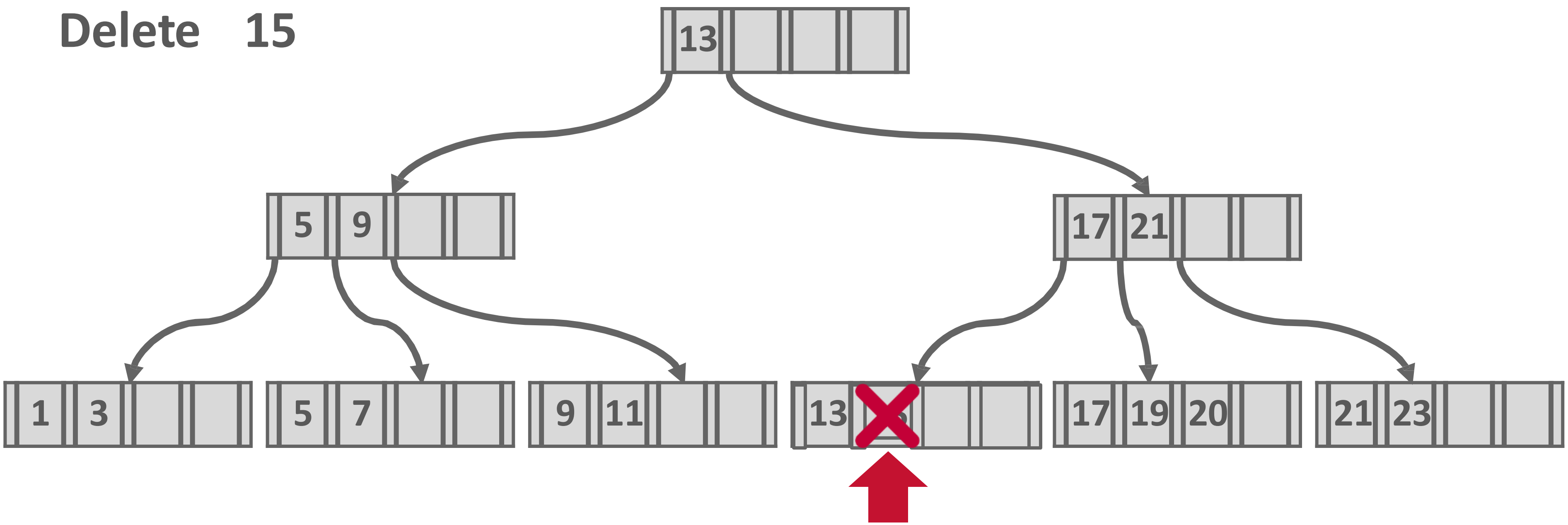
Delete 6



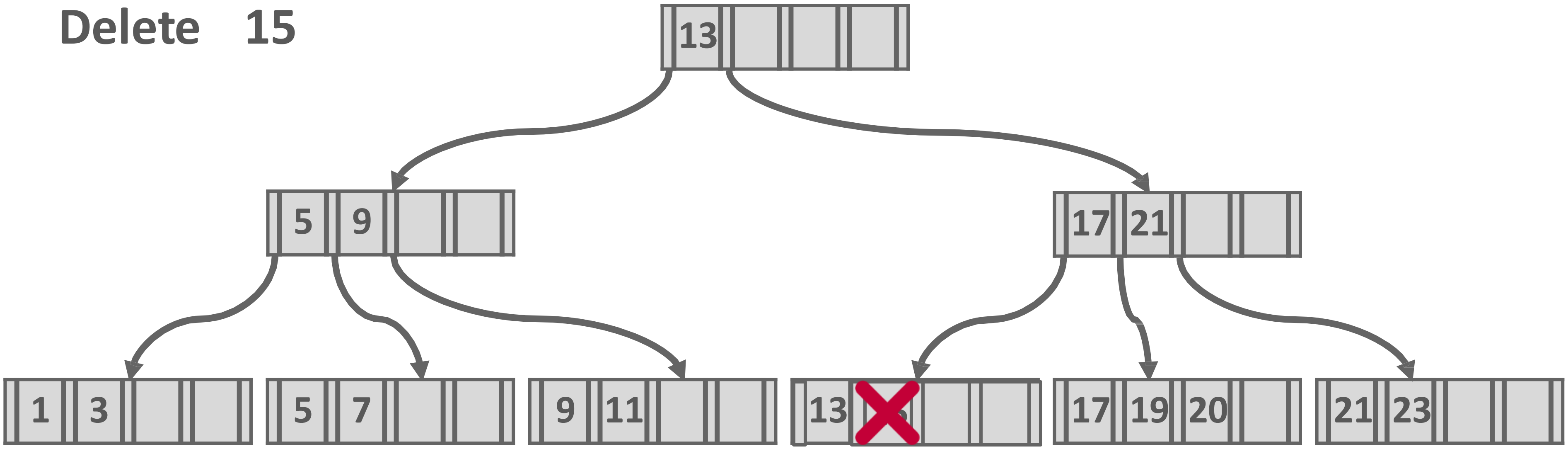
Delete 15



Delete 15

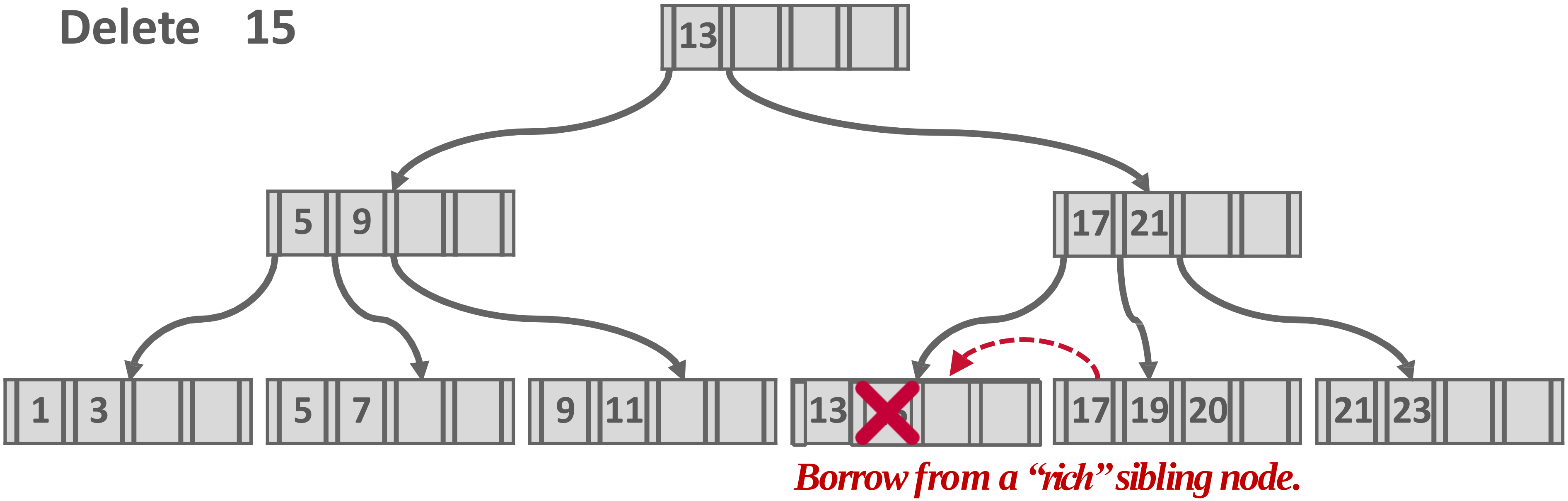


Delete 15

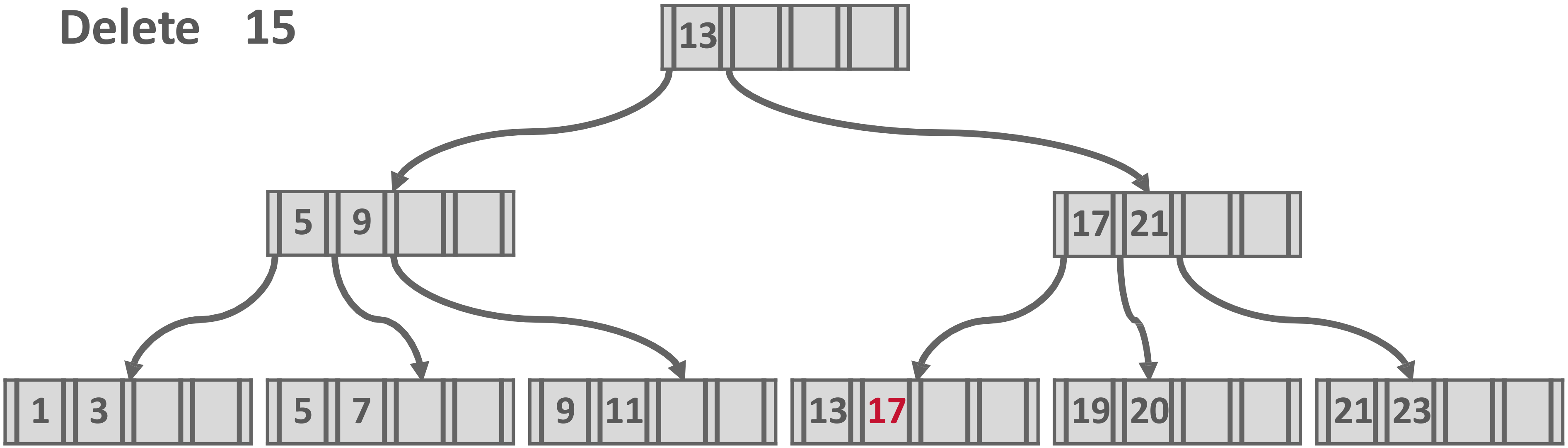


Borrow from a "rich" sibling node.

Delete 15

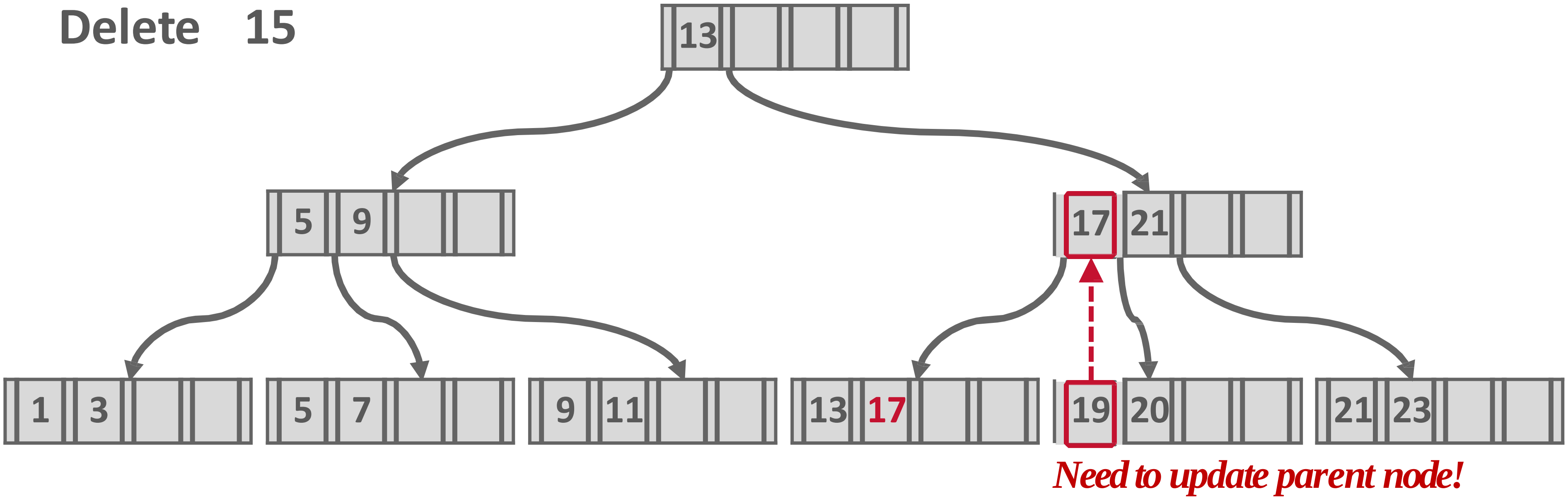

13
5 9
17 21
1 3
5 7
9 11
13
17 19 20
21 23
Borrow from a "rich" sibling node.

Delete 15

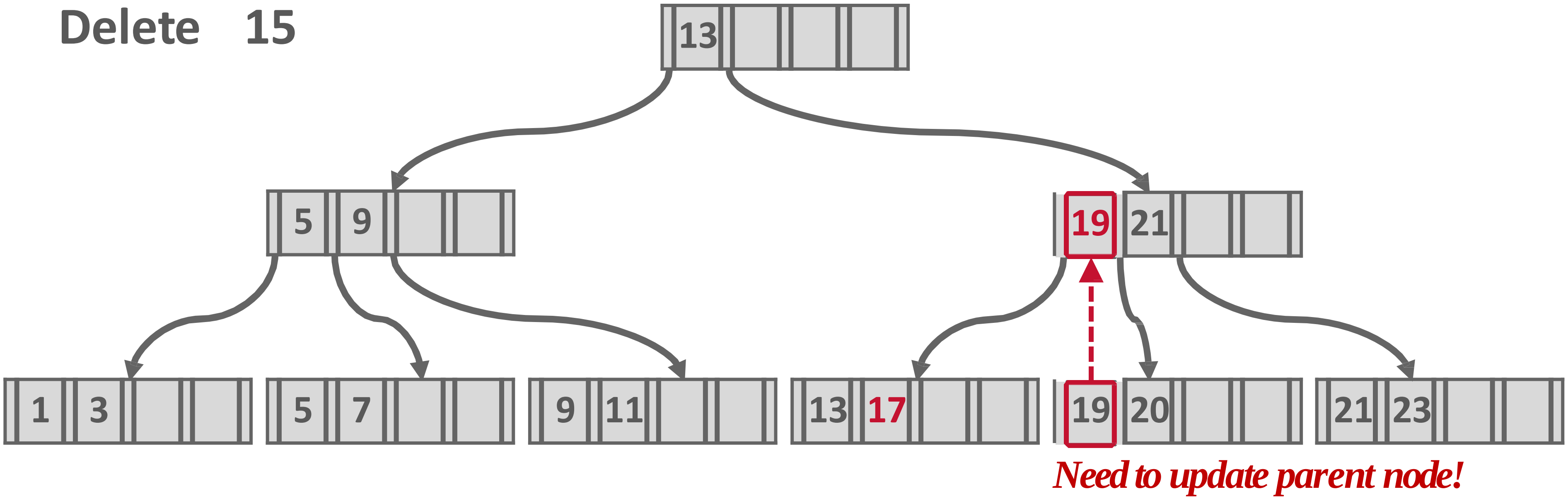


Need to update parent node!

Delete 15

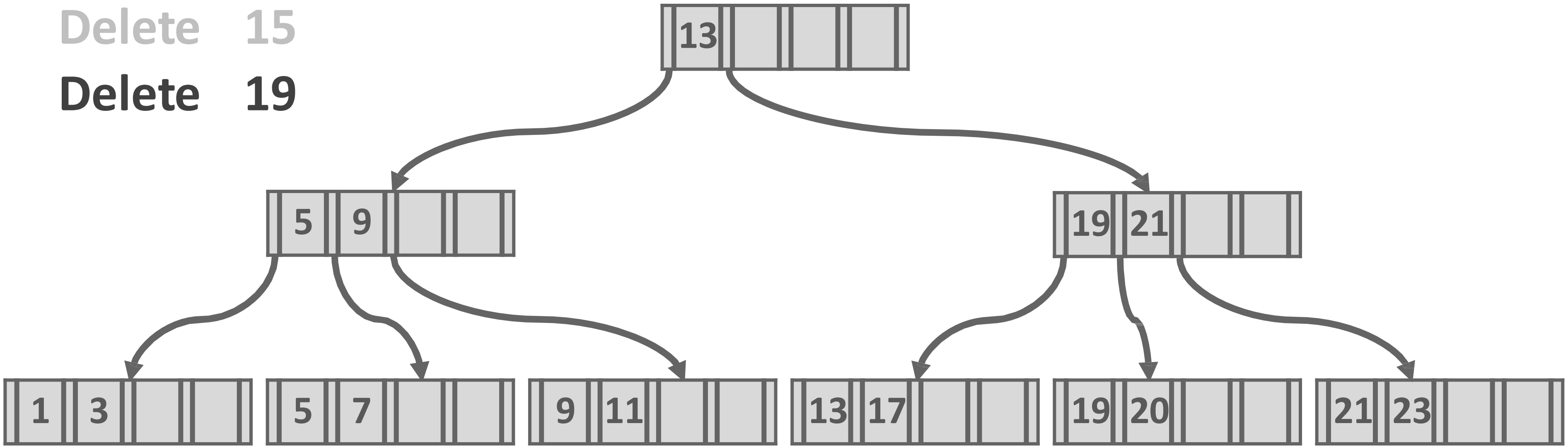


Delete 15



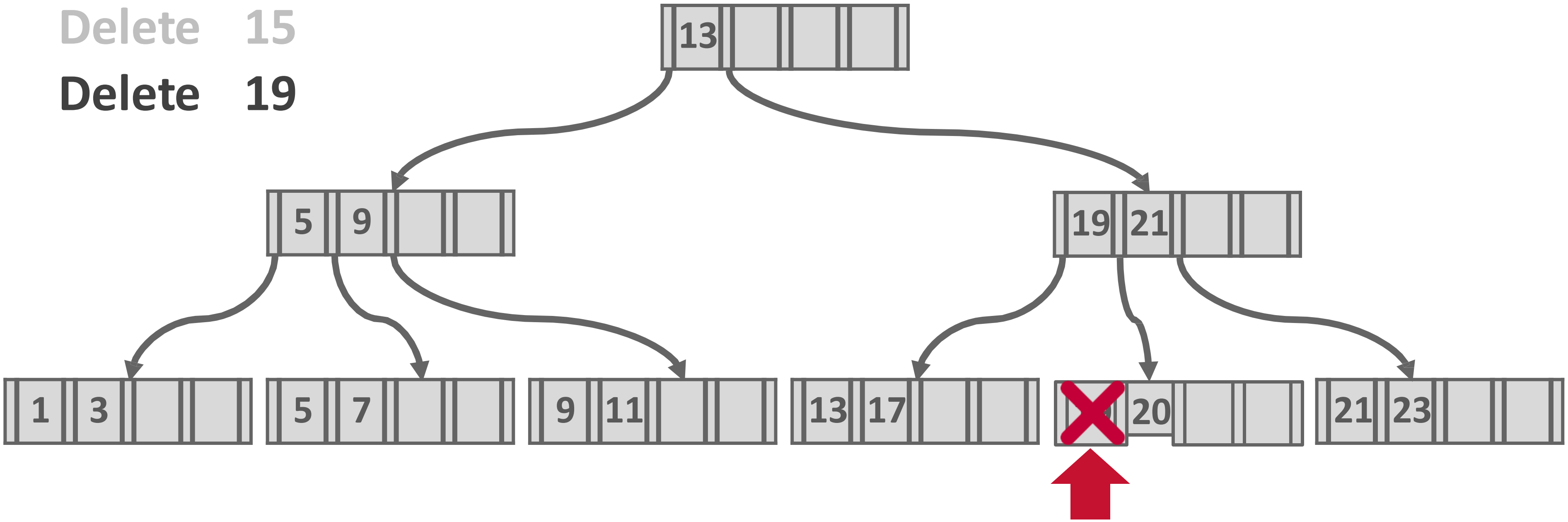
Delete 15

Delete 19



Delete 15

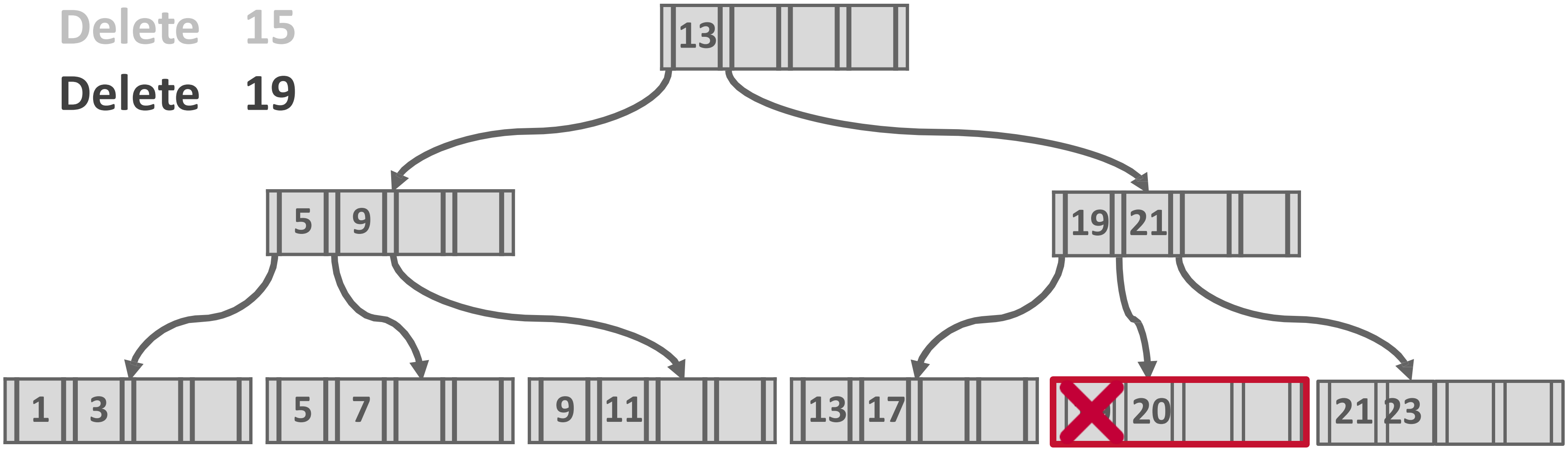
Delete 19



THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND
SCIENCE DEPARTMENT OF COMPUTER SCIENCE

Delete 15

Delete 19



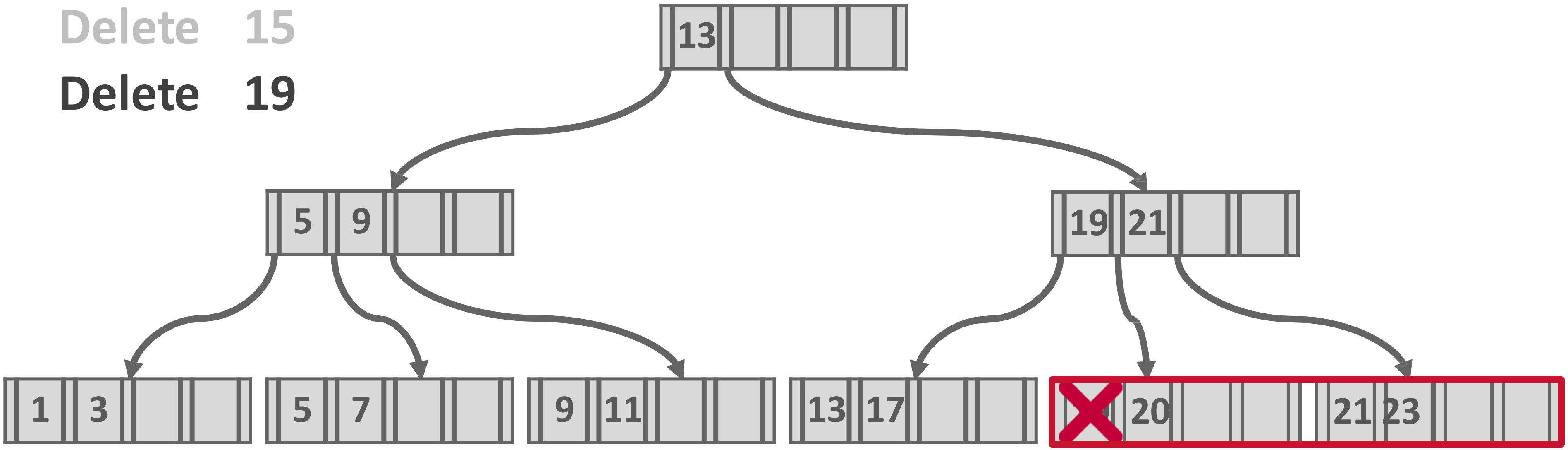
Under-filled!

No "rich" sibling nodes to borrow.

Merge with a sibling

Delete 15

Delete 19



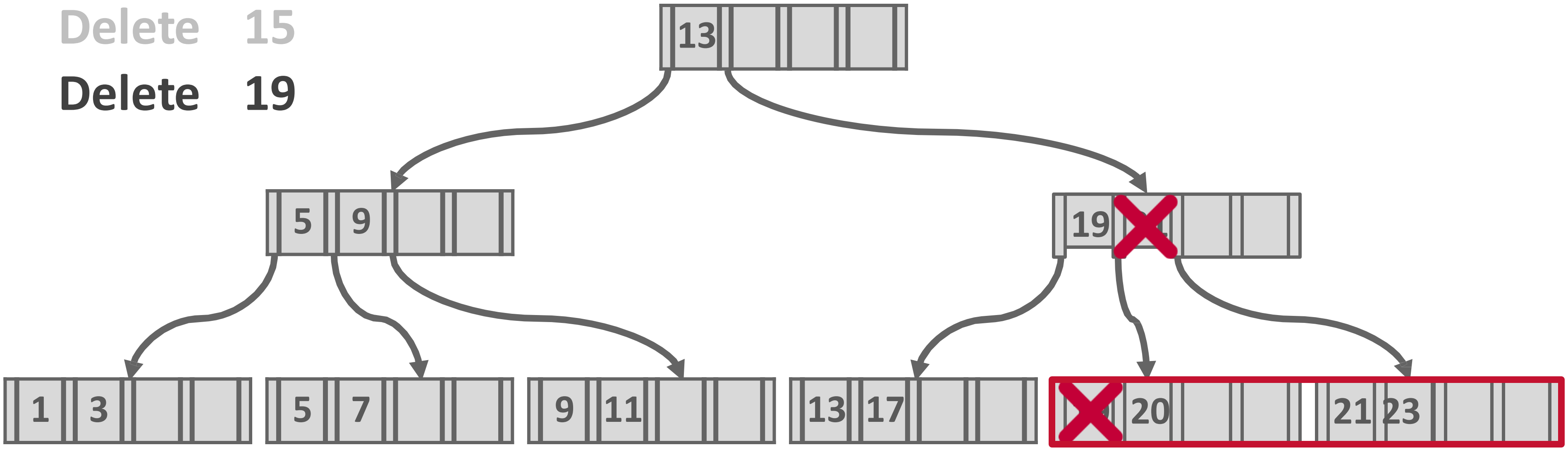
Under-filled!

No "rich" sibling nodes to borrow.

Merge with a sibling

Delete 15

Delete 19



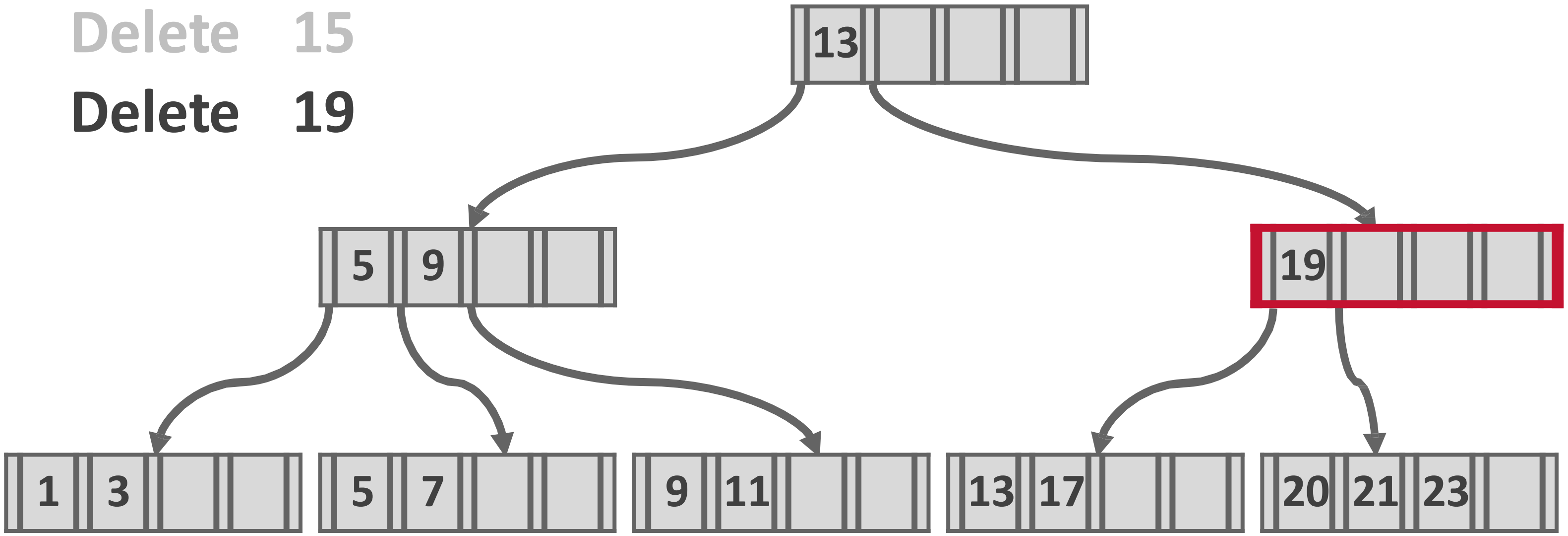
Under-filled!

No "rich" sibling nodes to borrow.

Merge with a sibling

Delete 15

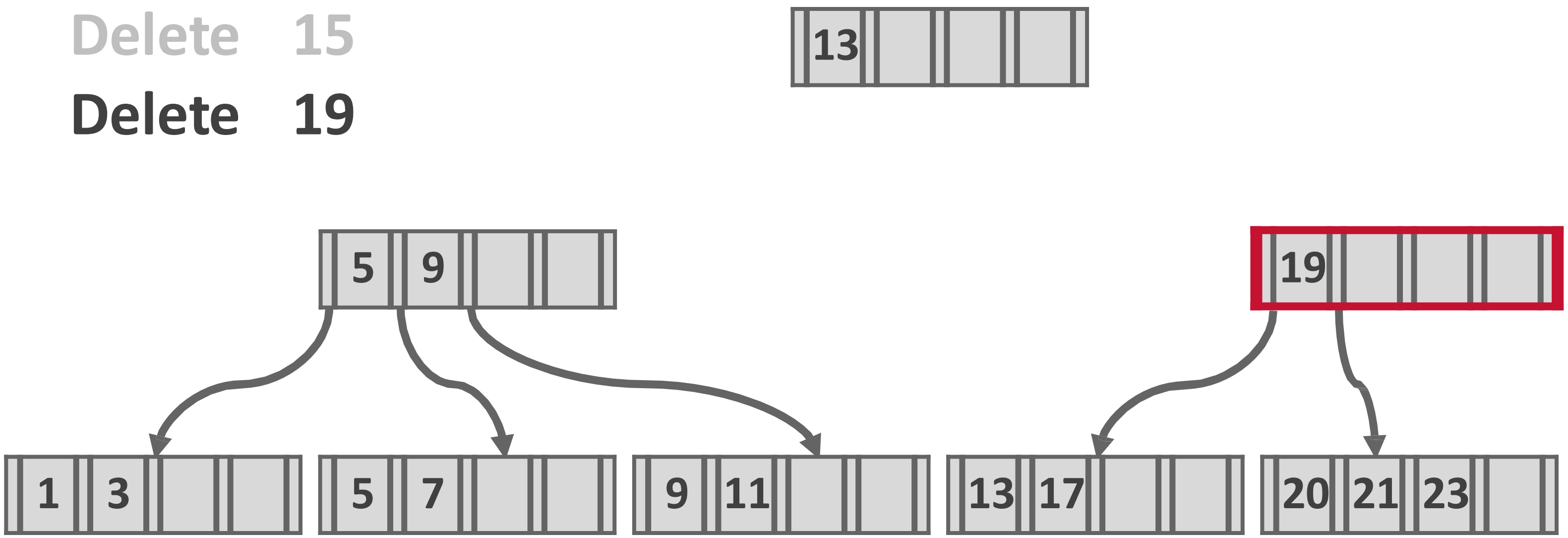
Delete 19



This node is under-filled! Pull-down.

Delete 15

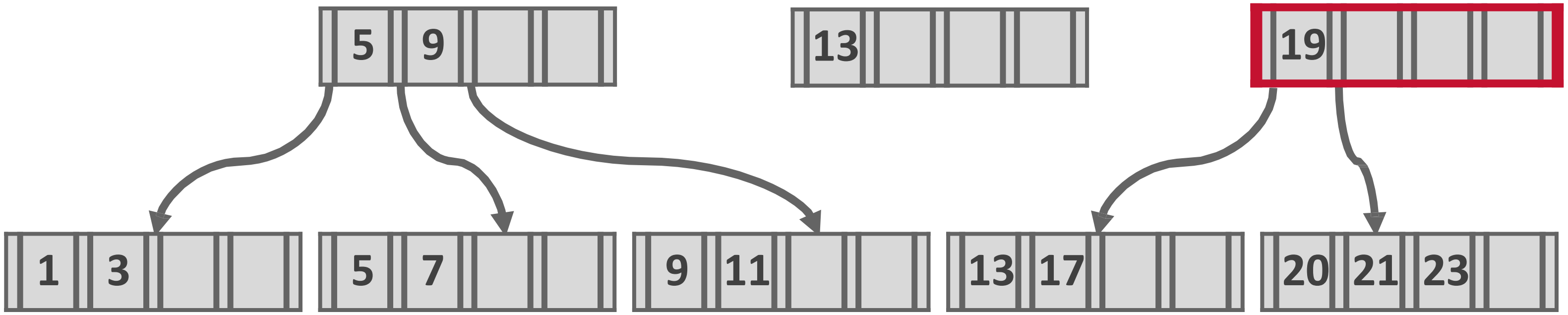
Delete 19



This node is under-filled! Pull-down.

Delete 15

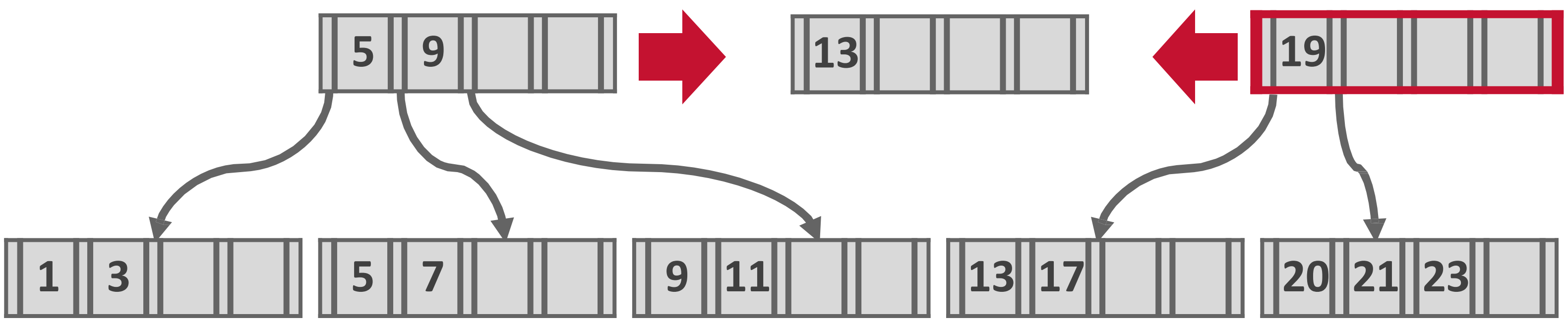
Delete 19



This node is under-filled! Pull-down.

Delete 15

Delete 19

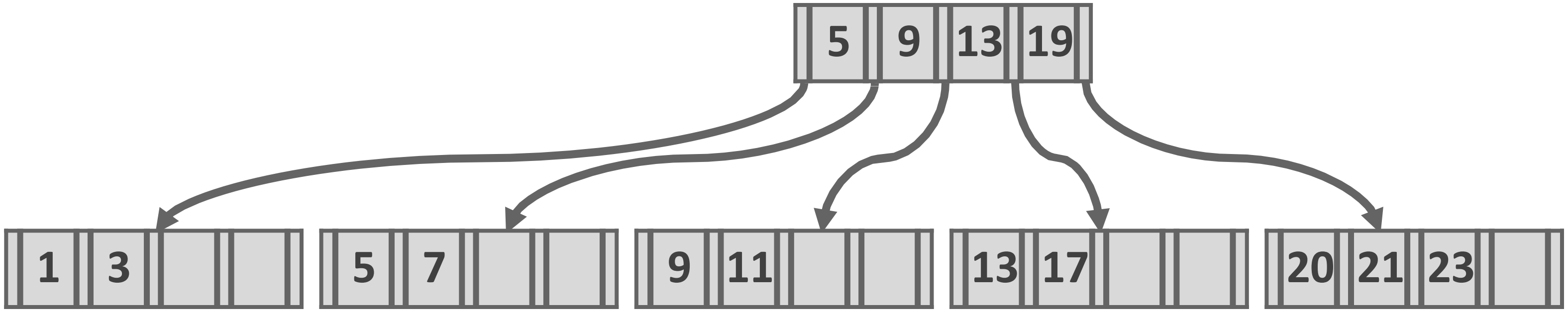


This node is under-filled! Pull-down.

Delete 15

Delete 19

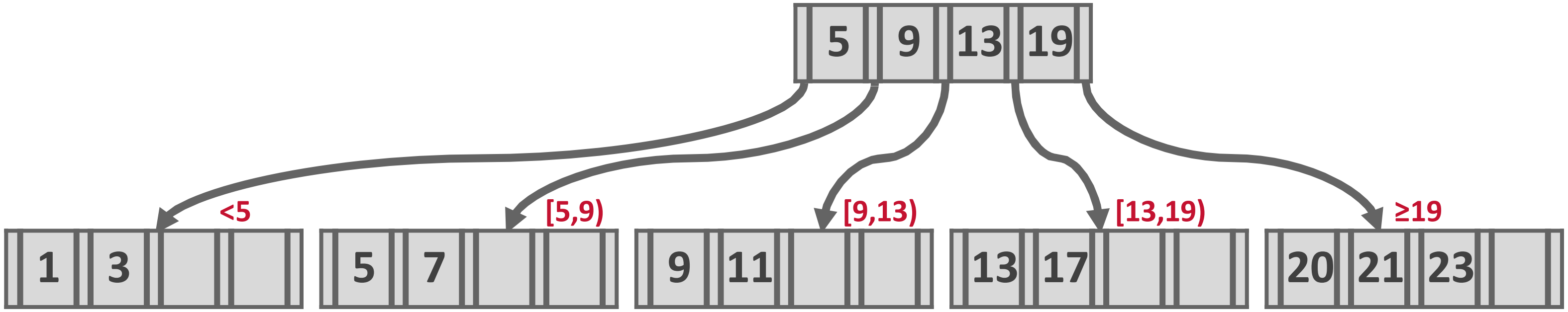
The tree has shrunk in height.



Delete 15

Delete 19

The tree has shrunk in height.



Queries on B⁺-Trees

- If there are *n* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil m/2 \rceil}(n) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and *m* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *m* = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

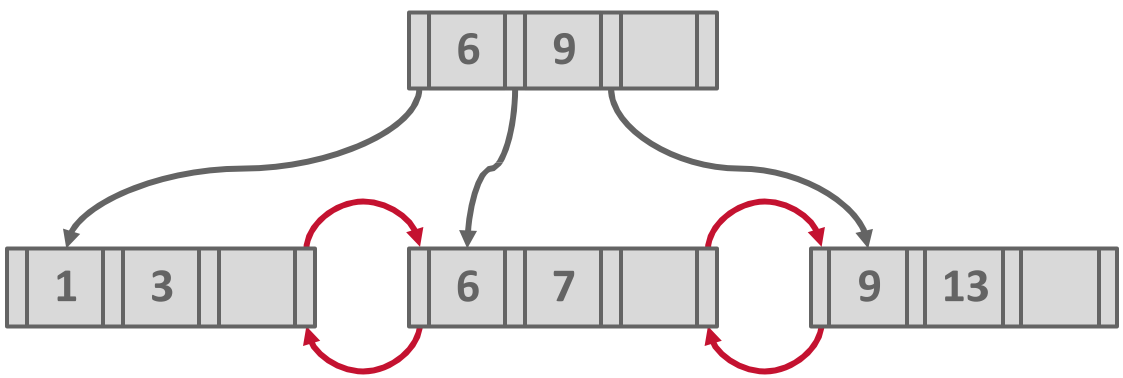
Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With n entries and maximum fanout of m, worst case complexity of insert/delete of an entry is $O(\log_{\lceil m/2 \rceil}(n))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

Bulk Loading and Bottom-Up Build

- The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

- Keys: 3, 7, 9, 13, 6, 1.
- Sorted Keys: 1, 3, 6, 7, 9, 13



Why bulk loading is better than insertion-based B+Tree construction?

Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires $\geq$ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms)
  - insert in sorted order
    - a leaf needs to be written out only once
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., "Silas" and "Silberschatz" can be separated by "Silb"
  - Keys in leaf node can be compressed by sharing common prefixes

FIN

Any questions?

Queries on B$^+$-Trees

function *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i then*
   3. Set *C = last non-null pointer in C*
   4. **else if** (v = C.K_i) Set C = P_{i+1}
   5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */

Insert

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L'
            Copy L.P_1 … L.K_{n−1} to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L'.P_n = L.P_n; Set L.P_n = L'
            Erase L.P_1 through L.K_{n−1} from L
            Copy T.P_1 through T.K_{⌈n/2⌉} from T into L starting at L.P_1
            Copy T.P_{⌈n/2⌉+1} through T.K_n from T into L' starting at L'.P_1
            Let K' be the smallest key-value in L'
            insert_in_parent(L, K', L')
        end
```

```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than or equal to K
            Insert P, K into L just after L.K_i
        end
procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P_1 ... T.P_⌈(n+1)/2⌉ into P
            Let K'' = T.K_⌈(n+1)/2⌉
            Copy T.P_⌈(n+1)/2⌉+1 ... T.P_n+1 into P'
            insert_in_parent(P, K'', P')
        end
```