>>

# B-trees

/\ >>

# ❖ B-Trees

B-trees are multi-way search trees with the properties:

- they are updated so as to remain balanced
- each node has at least *(n-1)/2* entries in it
- each tree node occupies an entire disk page

B-tree insertion and deletion methods

- are moderately complicated to describe
- can be implemented very efficiently

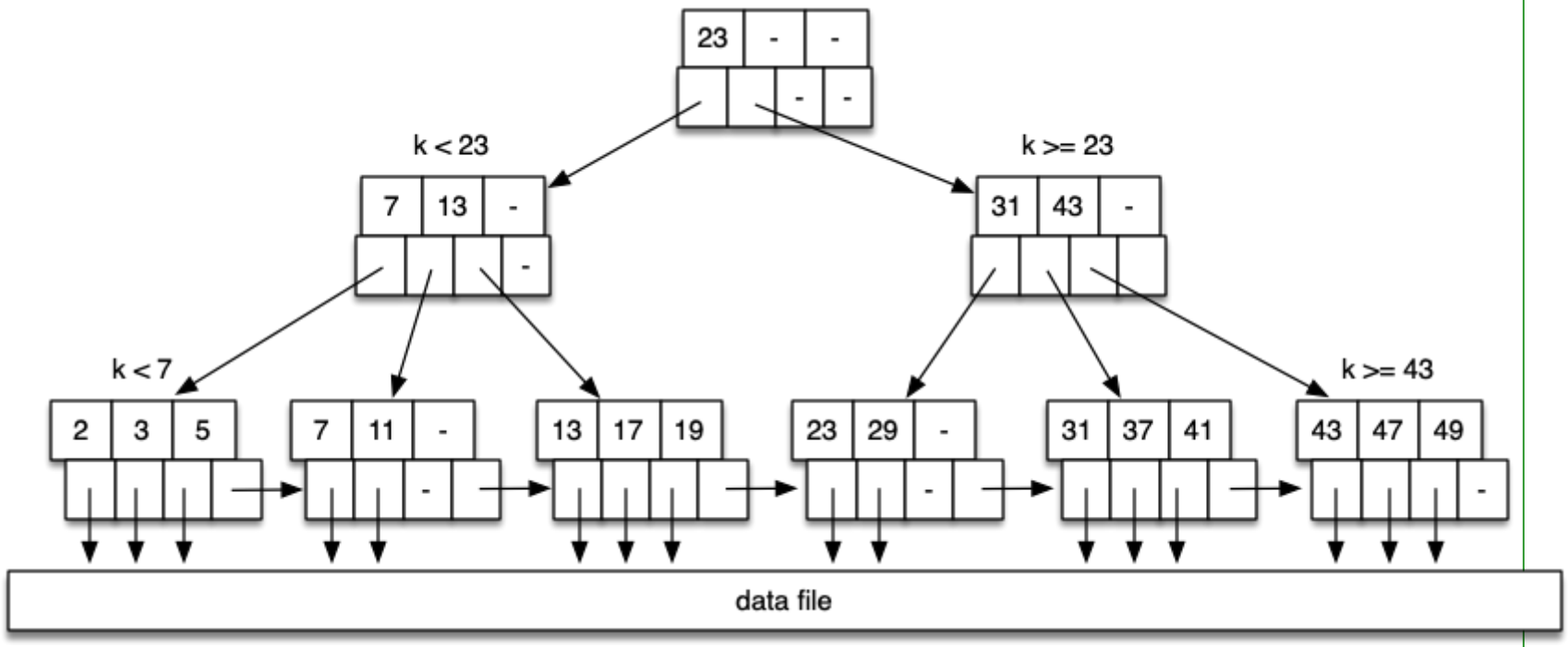
Advantages of B-trees over general multi-way search trees:

- better storage utilisation (around 2/3 full)
- better worst case performance (shallower)

<< ^ >>

# ❖ B-Trees (cont)

Example B-tree (depth=3, n=3) (actually B+ tree)




(Note: in DBs, nodes are pages ⇒ large branching factor, e.g. *n=500*)

<< ^ >>

# ❖ B-Tree Depth

Depth depends on effective branching factor (i.e. how full nodes are).

Simulation studies show typical B-tree nodes are 69% full.

Gives load $L_i = 0.69 \times c_i$ and depth of tree $\sim ceil( log_{L_i} r )$.

Example: $c_i=128$, $L_i=88$

| Level | #nodes | #keys |
|---|---|---|
| root | 1 | 88 |
| 1 | 89 | 7832 |
| 2 | 7921 | 697048 |
| 3 | 704969 | 62037272 |

Note: $c_i$ is generally larger than 128 for a real B-tree.

<< ^ >>

# ❖ Selection with B-Trees

For *one* queries:

```
Node find(k,tree) {
    return search(k, root_of(tree))
}
Node search(k, node) {
    // get the page of the node
    if (is_leaf(node)) return node
    keys = array of nk key values in node
    pages = array of nk+1 ptrs to child nodes
    if (k <= keys[0])
        return search(k, pages[0])
    else if (keys[i] < k <= keys[i+1])
        return search(k, pages[i+1])
    else if (k > keys[nk-1])
        return search(k, pages[nk])
}
```

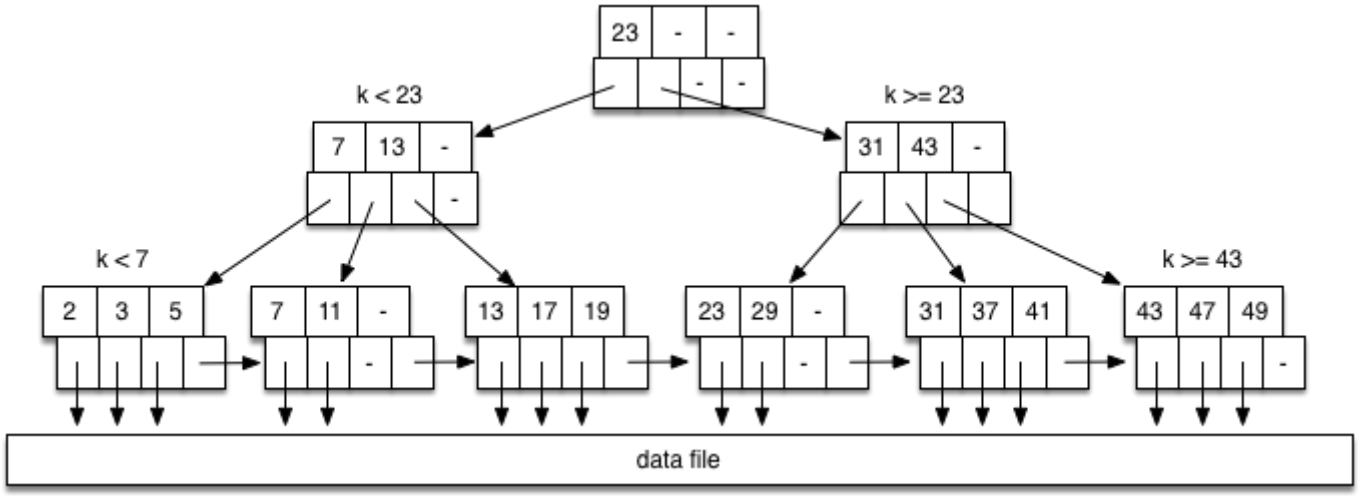
<< ^ >>

# ❖ Selection with B-Trees (cont)

Simplified description of search ...

```
N = B-tree root node
while (N is not a leaf node) N = scanToFindChild(N,K)
tid = scanToFindEntry(N,K)
access tuple T using tid
```

$Cost_{one} = (D + 1)_r$

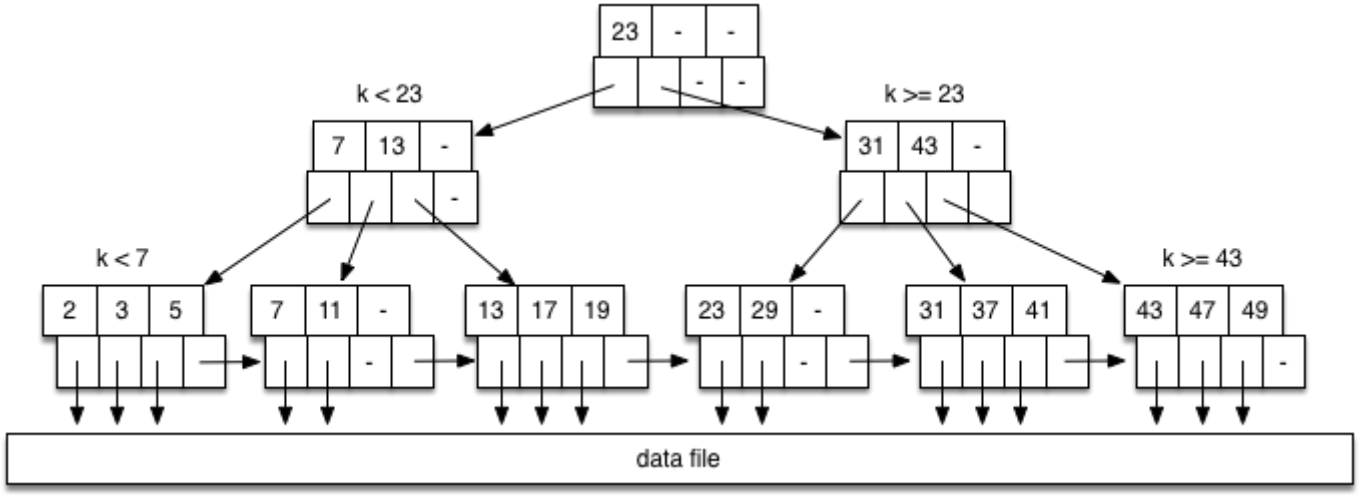


# ❖ Selection with B-Trees (cont)

For *range* queries (assume sorted on index attribute):

```
search index to find leaf node for Lo
for each leaf node entry until Hi found
    add pageOf(tid) to Pages to be scanned
scan Pages looking for matching tuples
```

$Cost_{range} = (D + b_i + b_q)_r$



<< ^ >>

# ❖ Insertion into B-Trees

Overview of the method:

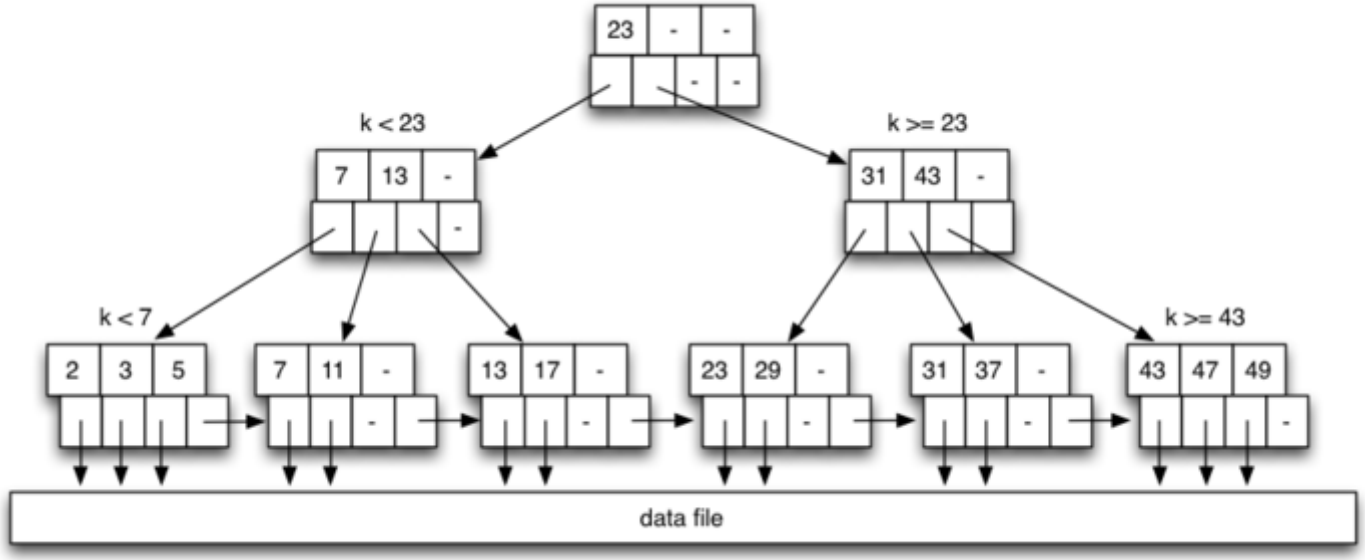
1. find leaf node and position in node where new key belongs
2. if node is not full, insert entry into appropriate position
3. if node is full ...
   - promote middle element to parent
   - split node into two half-full nodes (< middle, > middle)
   - insert new key into appropriate half-full node
4. if parent full, split and promote upwards
5. if reach root, and root is full, make new root upwards

Note: if duplicates not allowed and key exists, may stop after step 1.

<< ^ >>

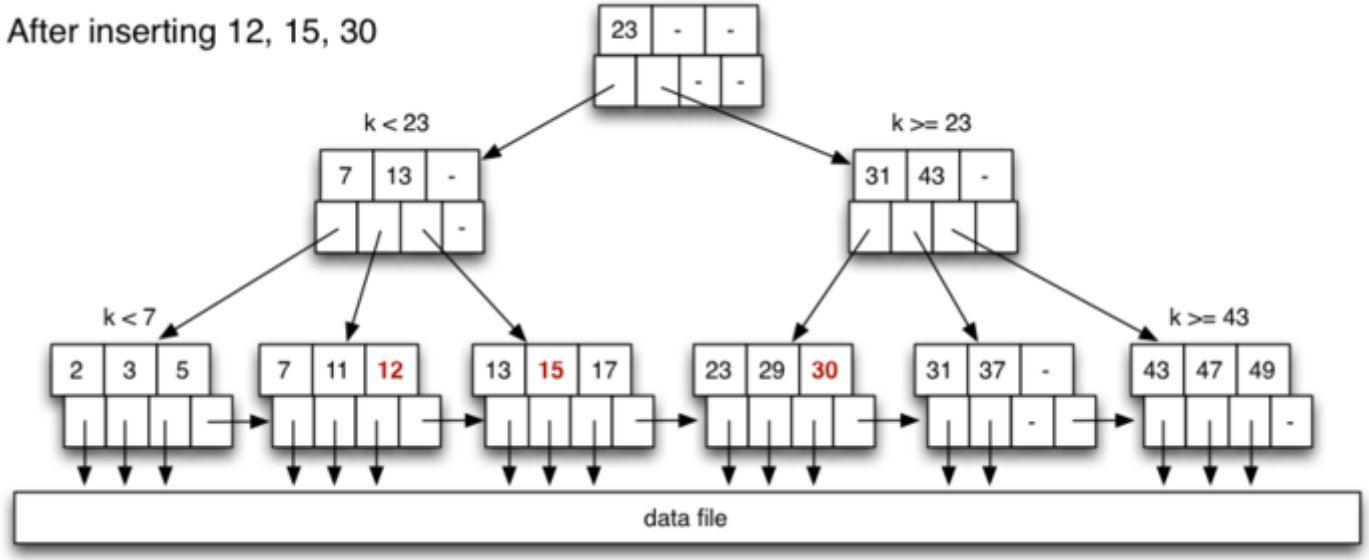
# ❖ Example: B-tree Insertion

Starting from this tree:



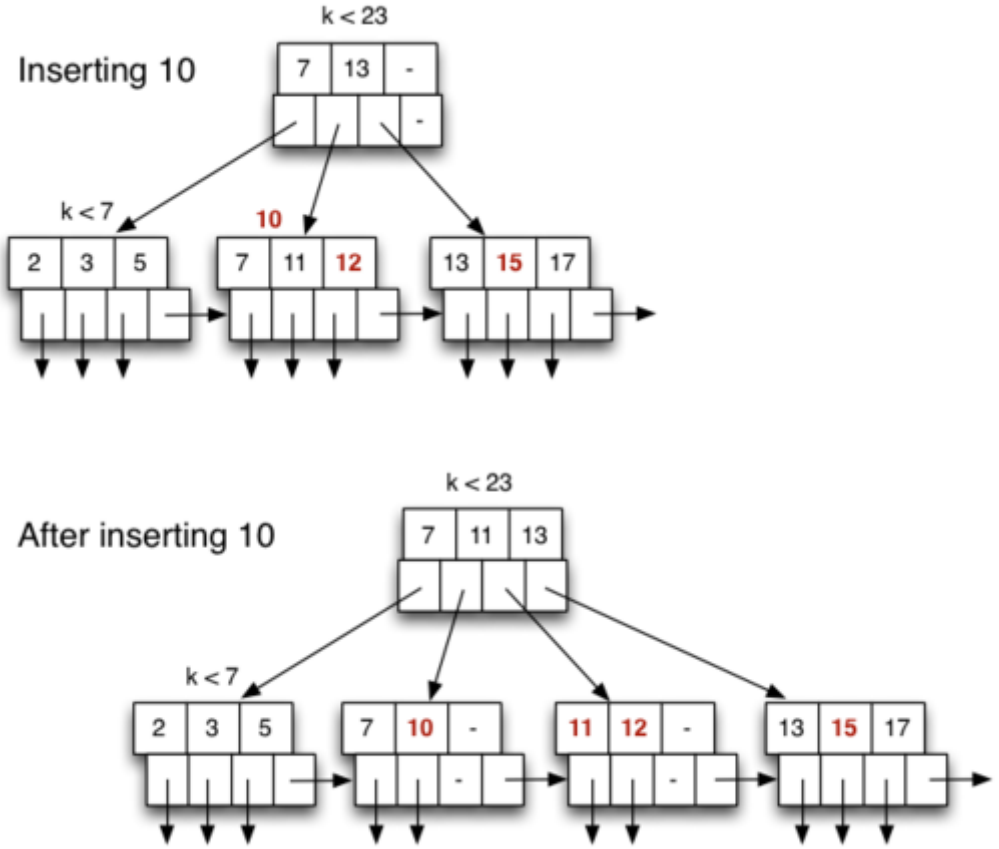

insert the following keys in the given order **12 15 30 10**

# ❖ Example: B-tree Insertion (cont)



<< ^ >>

# ❖ Example: B-tree Insertion (cont)



<< ^ >>

# ❖ B-Tree Insertion Cost

Insertion cost = $Cost_{treeSearch}$ + $Cost_{treeInsert}$ + $Cost_{dataInsert}$

Best case: write one page (most of time)

- traverse from root to leaf
- read/write data page, write updated leaf

$Cost_{insert} = D_r + 1_w + 1_r + 1_w$

Common case: *3* node writes (rearrange 2 leaves + parent)

- traverse from root to leaf, holding nodes in buffer
- read/write data page
- update/write leaf, parent and sibling

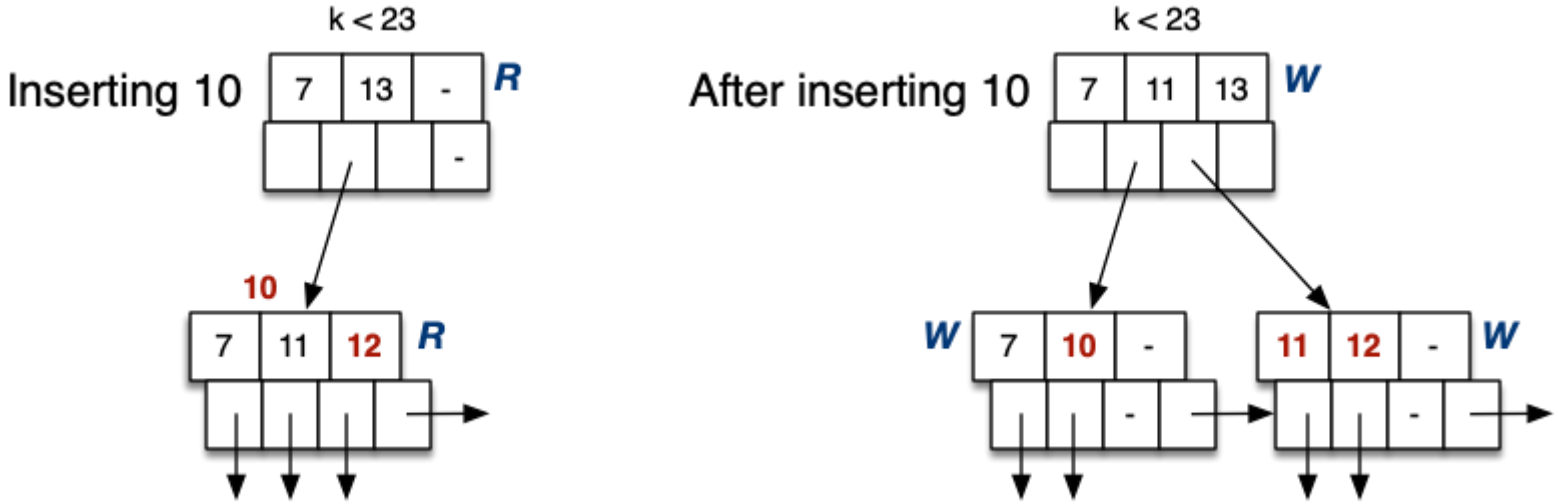
$Cost_{insert} = D_r + 3_w + 1_r + 1_w$

<< ^ >>

# ❖ B-Tree Insertion Cost (cont)

Worst case: propagate to root $Cost_{insert} = D_r + D.3_w + 1_r + 1_w$

- traverse from root to leaf
- read/write data page
- update/write leaf, parent and sibling
- repeat previous step *D-1* times



<< ^ >>

# ❖ B-trees in PostgreSQL

PostgreSQL implements ≅ Lehman/Yao-style B-trees

- variant that works effectively in high-concurrency environments.

B-tree implementation: **`backend/access/nbtree`**

- **`README`** ... comprehensive description of methods
- **`nbtree.c`** ... interface functions (for iterators)
- **`nbtsearch.c`** ... traverse index to find key value
- **`nbtinsert.c`** ... add new entry to B-tree index

Notes:

- stores all instances of equal keys (dense index)
- avoids splitting by scanning right if key = max(key) in page
- common insert case: new key is max(key) overall; handled efficiently

## ❖ B-trees in PostgreSQL (cont)

Changes for PostgreSQL v12

- indexes smaller
  - for composite keys, only store first attribute
  - index entries are smaller, so $c_i$ larger, so tree shallower
- include TID in index key
  - duplicate index entries are stored in "table order"
  - makes scanning table files to collect results more efficient

To explore indexes in more detail:

- `\di+ IndexName`
- `select * from bt_page_items(IndexName,BlockNo)`

<< ^

# ❖ B-trees in PostgreSQL (cont)

Interface functions for B-trees

```
// build Btree index on relation
Datum btbuild(rel,index,...)
// insert index entry into Btree
Datum btinsert(rel,key,tupleid,index,...)
// start scan on Btree index
Datum btbeginscan(rel,key,scandesc,...)
// get next tuple in a scan
Datum btgettuple(scandesc,scandir,...)
// close down a scan
Datum btendscan(scandesc)
```

Produced: 24 Mar 2021