B-trees

We have seen how to guarantee good behaviour in binary search trees with limited flexibility in balance.

A major problem remains: If the tree is too big, it will not fit in the main memory - each node access represents a disk access: this is costly.

We will introduce B-trees: *limited feasibility in the degree* of trees with complete balance.

Key issues for searching secondary store (disk)

- Very slow.
- Each access returns a "page" (several records).
- AVL tree is not suitable: For $n = 10^6$, need about 20 disk accessees.
- We would like $\log_k n$ access, $k = \#$ records/page

Definition. B-tree of order m:

1. All leaves are on the same level.

2. Root contains at least 1 key, at most $m-1$ keys. (2 to m subtrees).

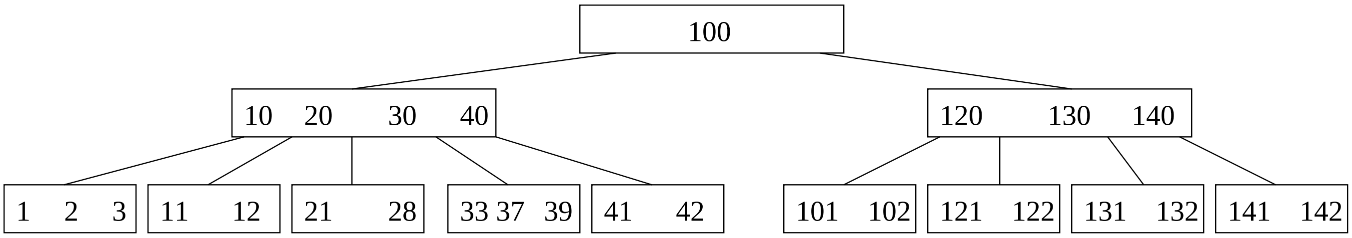
3. Each other internal node contains at least $\lceil \frac{m}{2} \rceil - 1$ kets and at most $\cdots m-1$ keys (or $\lceil \frac{m}{2} \rceil$ to m subtrees).

4. "Search tree order" preserved.

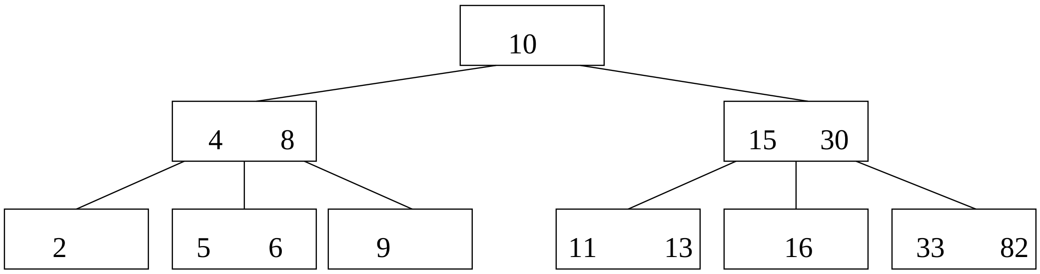
• This is used in most modern database systems. Typically, $m \approx 200$ to 300.

Examples.

- $m = 5$



- $m = 3$ (smallest possible value)



This is also called a **2-3 tree** (at most 2 keys and 3 subtrees).

Operations on B-Trees

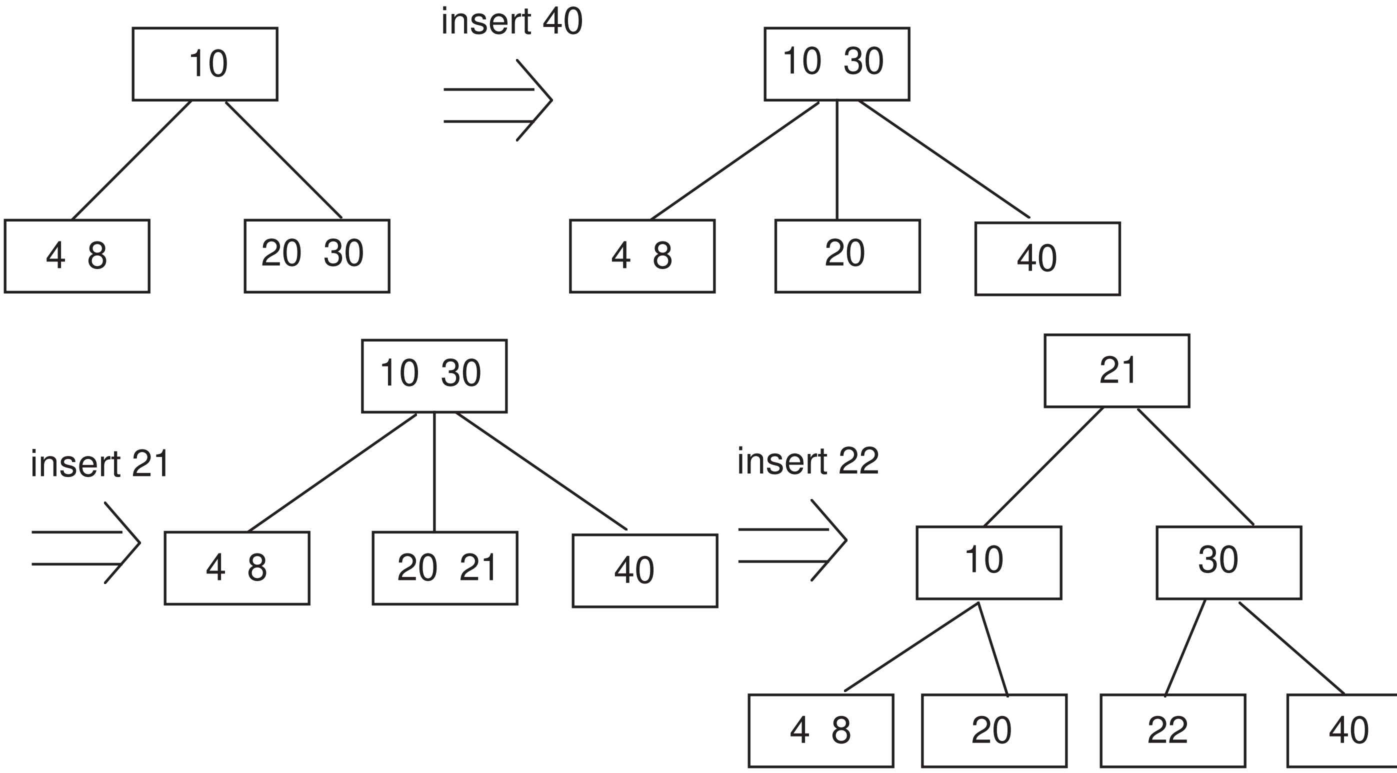
- Search: obvious.

- **Insertion**:

  1. Do search;

  2. Add to bottom level. If this node **overflows** ($\geq m$ keys), split it into two nodes, moving middle key to parent. If parent overflows also, repeat same process, going up the tree until it ends. (It will end, eventually at root).

Example: $m = 3$,



10
4 8
20 30
insert 40
10 30
4 8
20
40
insert 21
10 30
4 8
20 21
40
insert 22
21
10
30
4 8
20
22
40

- **Deletions**

If key not in a leaf, swap with successor or predecessor as in a binary tree.

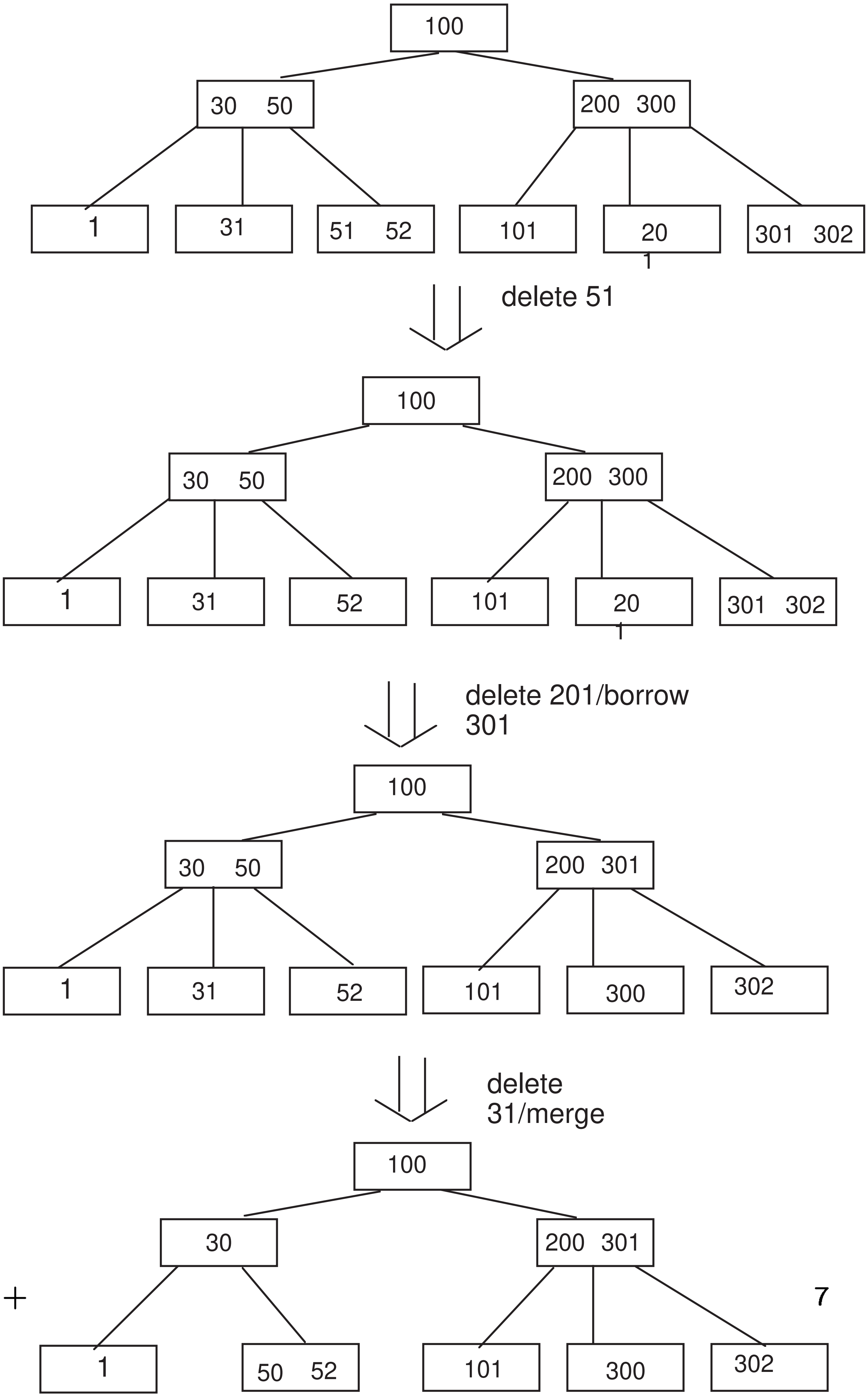
Remove key from leaf, call this "current node".

(a) If current node has sufficient number of keys ($\geq \lceil \frac{m}{2} \rceil - 1$), we are done.

(b) If not, try to "borrow" from left or right sibling, if either has $>$ minimum number of keys.

(c) If both siblings are "poor", combine current node with one sibling and the key in parent that separates them. But this reduces the number of keys/subtrees of the parent, parent may not have enough keys, parent becomes "current node". Repeat above process, going up the tree.

Example (for $m = 3$) is in next page.


100
30 50
200 300
1
31
51 52
101
201
301 302
delete 51
100
30 50
200 300
1
31
52
101
201
301 302
delete 201/borrow
301
100
30 50
200 301
1
31
52
101
300
302
delete
31/merge
100
30
200 301
1
50 52
101
300
302

What is maximum height of B-tree of order m with n data values? Or given height k, what is minimum number of values in the tree?

- Root $\geq$ 2 children.

- Other nodes have $\geq \lceil \frac{m}{2} \rceil$ children. # values at level k is at least $2\lceil \frac{m}{2} \rceil^{k-1} - 1$ ($\leq m^k - 1$).

Example: $m = 256$, $n = 16$ million. This requires a B-tree of height 3. Storing the root in the main memory, we need at most 3 disk accesses to retrieve any data item.

Modification: (1) deletion may involve locking nodes as we back up. On deletion, we can just remove element, leave space, re-use for later insert. (2) It is possible to do B-tree operations 1 pass, without the back up merge/split process. See Cormen, Leiserson, Rivest, *Introduction to Algorithms*, MIT Press.

B^+ trees: All data on leaves. Intermediate nodes as indices.

This is used when actual data records are much larger than "key".

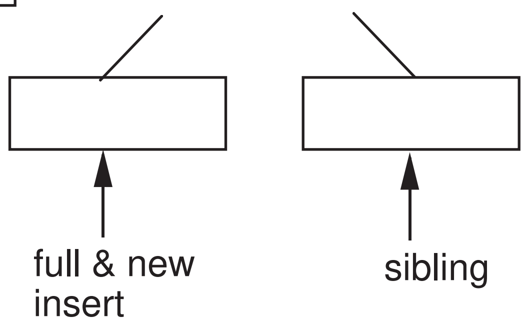
B-tree space utilization

- Could be as low as 50%.
- On average 69%.

Example. OED project at Waterloo uses a variation of B^+ tree. (The indexing is improved to guarantee fast access.)

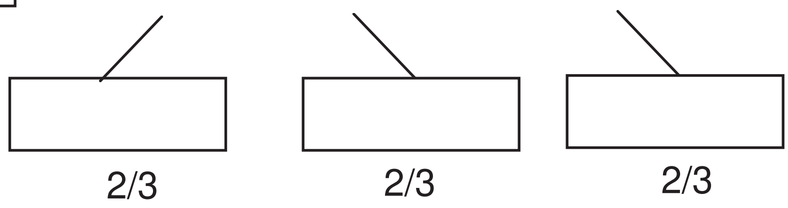
We can improve by:

Insert



split only if node and sibling full, o.w. move keys. (utilization 2/3)

Delete



If $< 2(m-1)$ values in node and its 2 siblings, then compress these 3 nodes into 2.

This guaratees 66% usage.

And average $>$ 90% usage.