# B-Trees

---



# B-Trees

Generalizes multilevel index.

Number of levels varies with size of data file, but is often 3.

Different variants, we start with B+-trees.

Useful for primary, secondary indexes, primary keys, nonkeys.

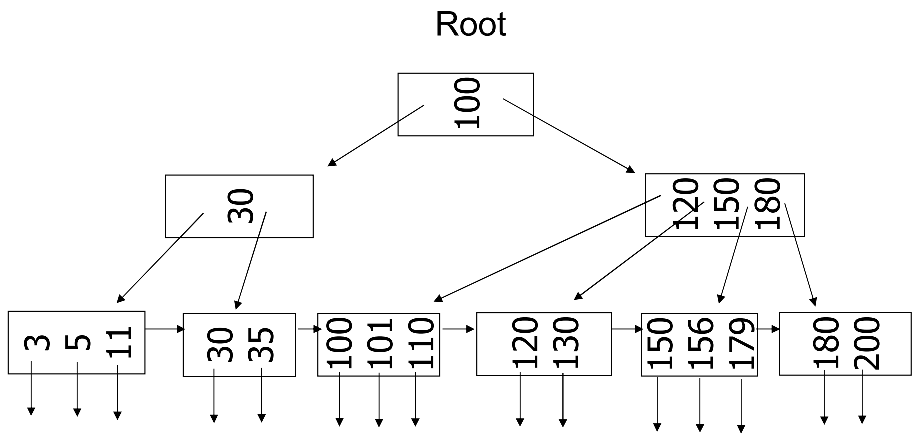
Each node in the tree represents a block.



---

Implementation of DBMS

# B+Tree Example






---

Implementation of DBMS

# Nodes of B+ Tree

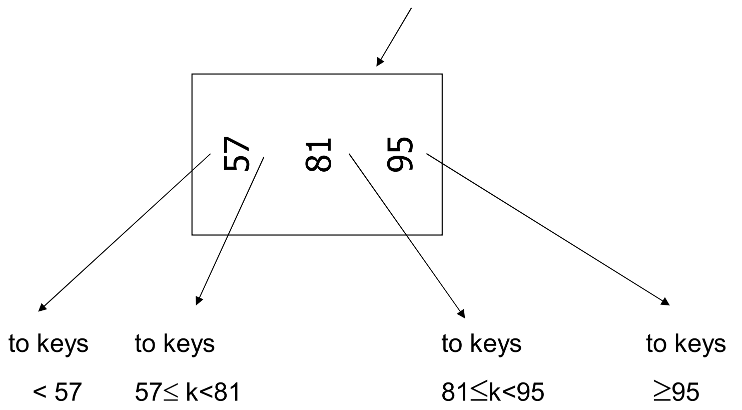
## Leaves

- One pointer to next leaf.
- keypointer pairs for records of data file.
- At least half of these (round up) occupied.

## Interior Nodes

- k keys form the divisions among k+1 subtrees.
- Key i is least key reachable from (i + 1)st child.

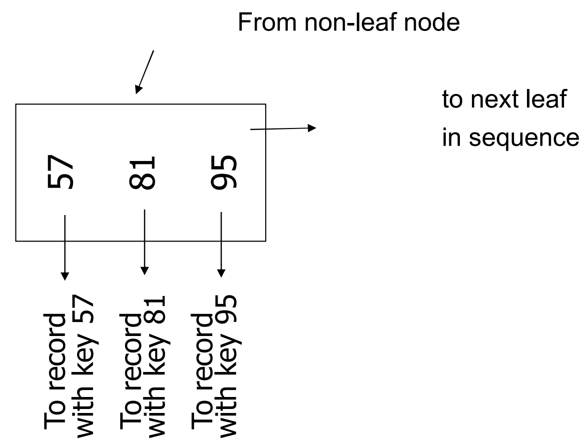
# Sample non-leaf








# Sample Leaf Node








# Don’t want nodes to be too empty

Trees have an order that determines the maximal number of keys in a node

Use in a tree of order n at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers to children

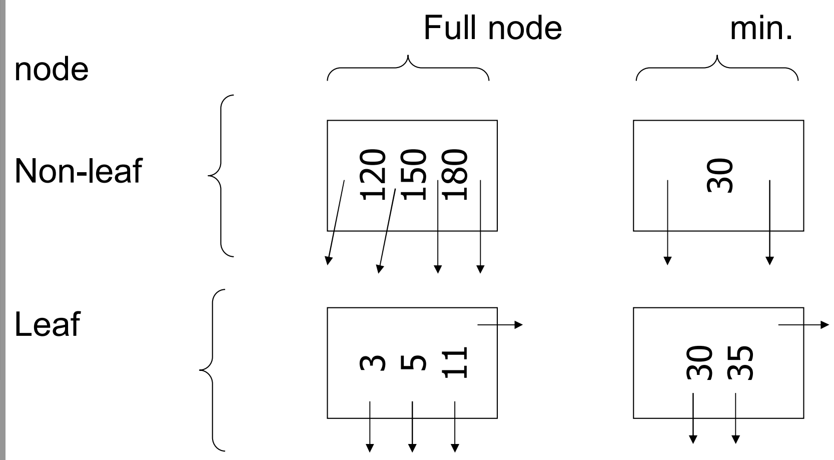
Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to records

Root is a special Case





n=3



# B+ Tree rules (Tree of order n)

(1) All leaves at same lowest level
(balanced tree)

(2) Pointers in leaves point to records
except for “sequence pointer”

(3) Number of pointers/keys for B+ tree (except for sequence pointers)

| | Max ptrs | Max keys | Min ptrs→data | Min keys |
|---|---|---|---|---|
| Non-leaf (non-root) | n+1 | n | $\lceil(n+1)/2\rceil$ | $\lceil(n+1)/2\rceil - 1$ |
| Leaf (non-root) | n | n | $\lfloor(n+1)/2\rfloor$ | $\lfloor(n+1)/2\rfloor$ |
| Root | n+1 | n | 1 (if leaf) | 1 |



# Lookup

Lookup in B+ Tree

- Start at root.
- Until you reach a leaf, follow the pointer that could lead to the key you want.
- Search that leaf (and leaves to the right if duplicates are possible).



# B+ Tree Insertion

Search for the key being inserted.

If there is room for another key-pointer pair at that leaf, insert there.

If no room, split leaf.

- Split of leaf results in insert of key-pointer pair at level above.
  - key is **copied** to level above
- Thus, recursive splitting all the way up the tree is possible.
  - split of non-leaf results in **moving** one key to level above
- Convention: If the number of keys in the two nodes resulting from the split is uneven, put one more key in the left node.
  Otherwise: both nodes get the same number of keys



# Examples for Insert into B+ Tree

(a) simple case
- space available in leaf

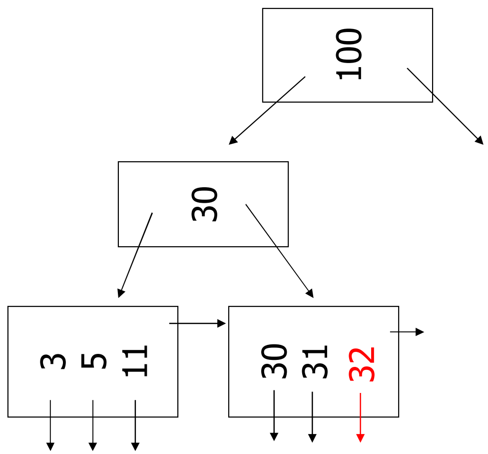
(b) leaf overflow

(c) non-leaf overflow

(d) new root

## (a) Insert key = 32

n=3

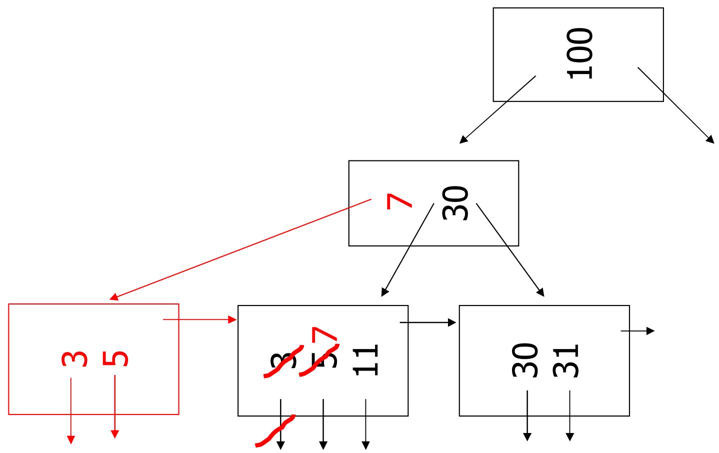





Implementation of DBMS

## (b) Insert key = 7

n=3

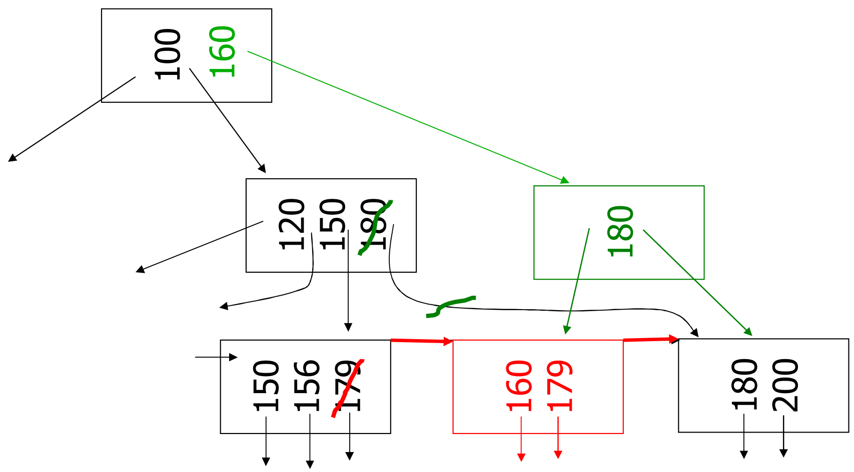





Implementation of DBMS

## (c) Insert key = 160

n=3

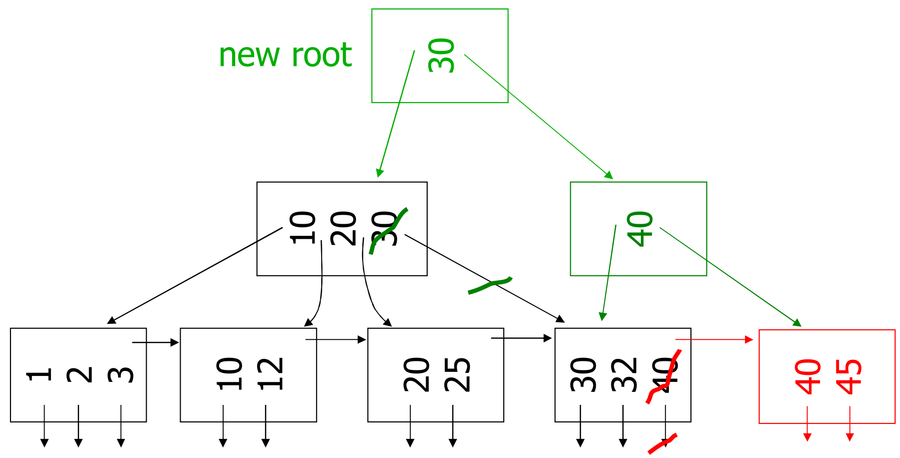





Implementation of DBMS

## (d) New root, insert 45

n=3



# B+ Tree Deletion

Search for key being deleted; If found, delete from the leaf.

If the lower limit on occupancy is violated:

- First look for an adjacent sibling that is above lower limit; transfer a key-pointer pair from that node (and update parent).
  - Convention: If you have the choice, use left sibling
  - A transfer between non-leaves involves a key in the parent and also results in the transfer of a child
- If none, then there must be two adjacent leaves, one at minimum, one below minimum. Just enough to merge nodes.
  - Convention: If you have the choice, use left sibling
  - A merge is the opposite of a split: delete key in parent when merging leaves; move key from parent into merged node for non-leaves
- Merger looks like delete above, so recursive deletion possible.
- Again, make sure keys are adjusted above.

Sometimes, it is OK to allow a B+ Tree leaf to become subminimum. But we handle underflows!!!





# Deletion from B+ Tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

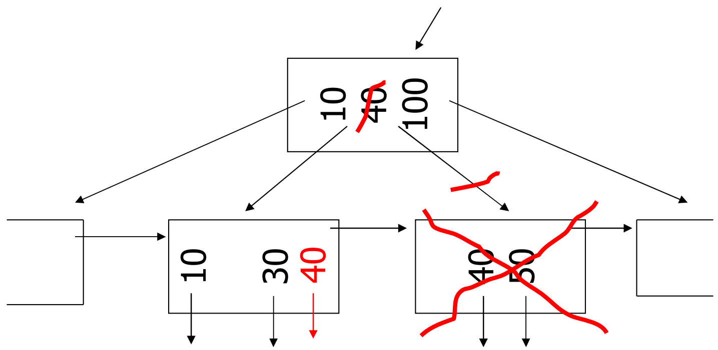




## (b) Coalesce with sibling

- Delete 50

n=4



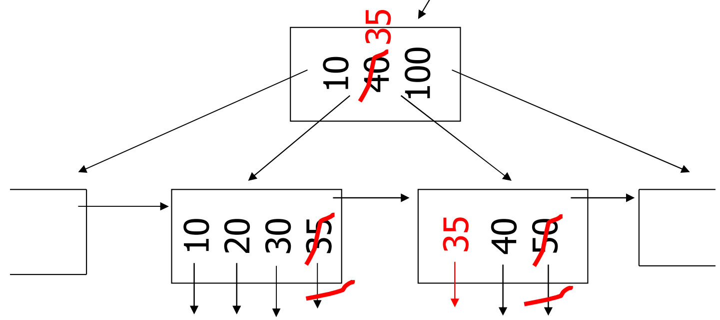





## (c) Redistribute keys

- Delete 50

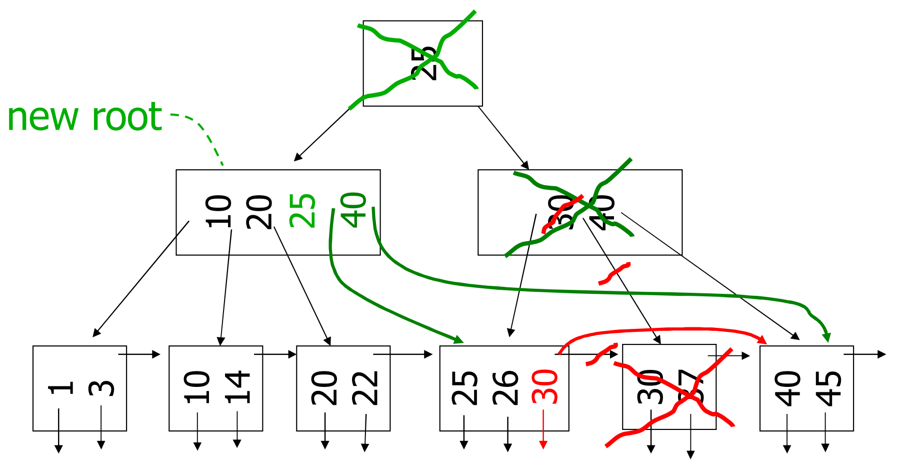
n=4



## (d) Non-leaf coalese

- Delete 37

n=4

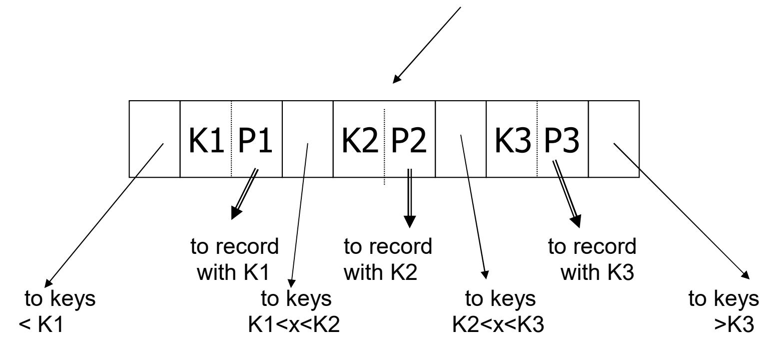





# Variation on B+ Tree: B Tree (no +)

Idea:

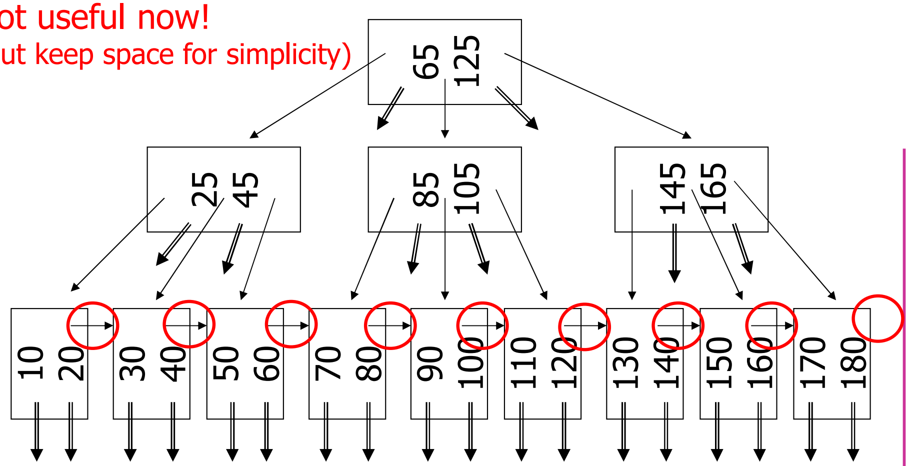
- Avoid duplicate keys
- Have record pointers in non-leaf nodes






# B Tree example (n=2)

sequence pointers
not useful now!
(but keep space for simplicity)






# B Tree operations

Ideas for search, insertion and deletion are similar to B+-trees

Main difference: We do not have to keep all keys in leaves

Consequence: When splitting a leaf, we **move** one key to the level above (also **move** when merging leaves)

Results:

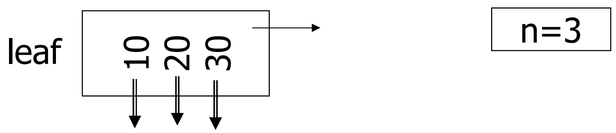
- split / merge for leaves are performed like the corresponding operations for non-leaves
- the minimum number of keys (and pointers) in a leaf is the same as for a non-leaf: $\lceil (n+1)/2 \rceil$ - 1 keys

Deletion:

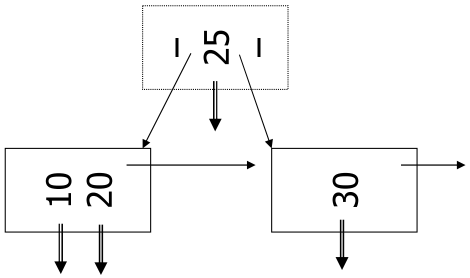
- when deleting a key that is in a non-leaf: replace the key with the next larger key in the tree
- handling of underflows always starts from a leaf

# Example: Insert

Insert record with key = 25




Afterwards:






# Comparison

☺ B-trees have faster lookup for keys in internal nodes than B+-trees

☹ in real implementations of a B-trees, non-leaf nodes can store a smaller number of keys compared to B+-trees due to the additional pointers

☹ Therefore, in B-trees the height of a tree for a particular number of keys can be larger compared to a B+-tree

➨ B+-trees are usually preferred!

Lookup for B+-tree is actually better!!



# Example

- Pointers 4 bytes
- Keys 4 bytes
- Blocks 100 bytes
- Look at full 2 level tree



# B tree

100/(4key byte+4 byte point to recor+4 byte pointer to node)
100/12 = 8 keys

Root has 8 keys + 8 record pointers + 9 child pointers

= 8x4 + 8x4 + 9x4 = 100 bytes

Each of 9 childs: 12 rec. pointers (+12 keys)

= 12x(4+4) = 96 bytes

2-level B-tree, Max # records =

12x9 + 8 = 116

# B+tree

Root has 12 keys + 13 child pointers

= 12x4 + 13x4 = 100 bytes

Each of 13 childs: 12 rec. ptrs (+12 keys)

= 12x(4 +4) + 4 = 100 bytes

2-level B+tree, Max # records

= 13x12 = 156

Conclusion:

- For fixed block size a B+ tree is better
- each node can store more keys and pointers to child nodes