CS525: Advanced Database Organization

Notes 4: Indexing and Hashing
Part II: B^+-Trees

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

September 11th, 2018

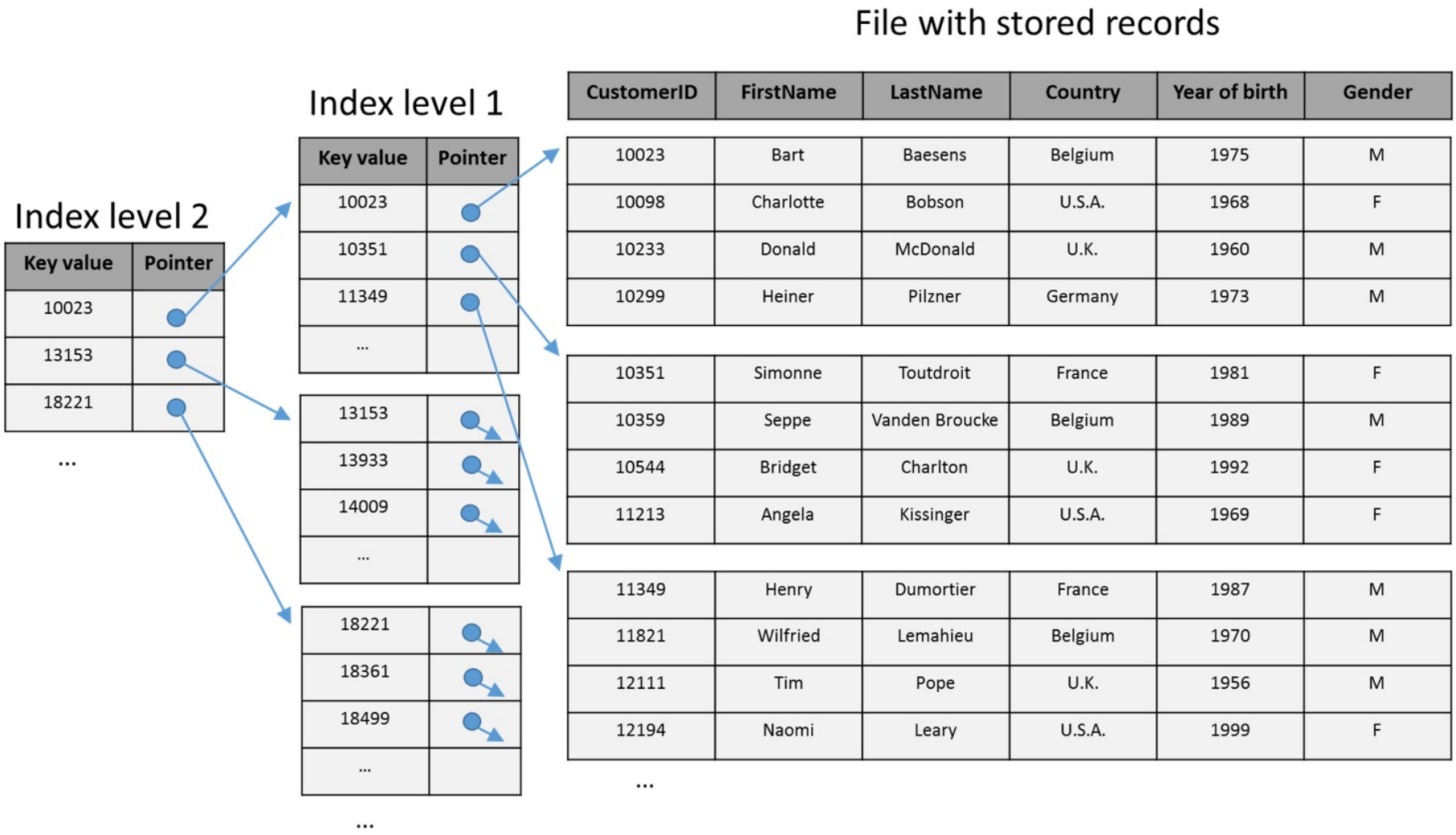
Slides: adapted from a course taught by Hector Garcia-Molina, Stanford, &Principles of Database Management

Outline

- Conventional indexes
  - Basic Ideas: sparse, dense, multi-level ...
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
- B^+-Trees
- Hashing schemes

Multilevel Indexes Revisited

- Creating index-to-an-index results in multilevel indexes



Multilevel Indexes Revisited

- Multilevel indexes useful for speeding up data access if lowest level index becomes too large
- Index can be considered as a sequential file and building an index-to-the-index improves access
- Higher-level index is, again, a sequential file to which index can be built and so on
- Lowest level index entries may contain pointers to disk blocks or records
- Higher-level index contains as many entries as there are blocks in the immediately lower level index
- Index entry consists of search key value and reference to corresponding block in lower level index
- Index levels can be added until highest-level index fits within single disk block
- First-level index, second-level index, third-level index etc.

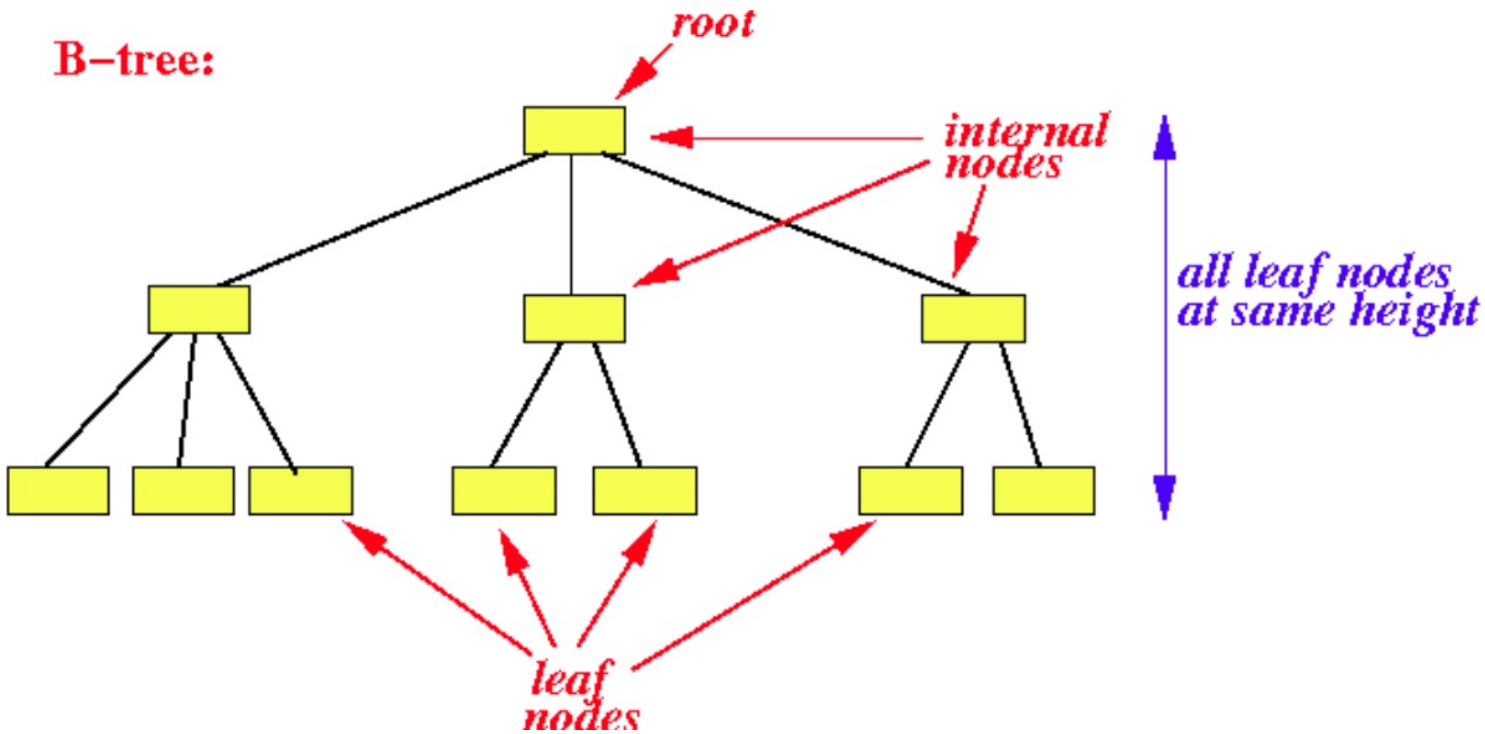
Multilevel Indexes Revisited

- With binary search on single index, search interval, consisting of disk blocks, is reduced by 2 with every iteration
- Approximately $\log_2(\texttt{NBLKI})$ random block access (rba) to search index consisting of NBLKI blocks
  - one additional rba needed to actual data file
- With multilevel index, search interval is reduced by BFI with every index level (BFI = blocking factor of index)
  - BFI denotes how many index entries fit within single disk block
  - also called fan-out of index

Multilevel Indexes Revisited

- Multilevel index can be considered as search tree, with each index level representing level in tree, each index block representing a node and each access to the index resulting in navigation towards a subtree in the tree
- Multilevel indexes may speed up data retrieval, but large multilevel indexes require a lot of maintenance in case of updates

Definitions related to a tree in general



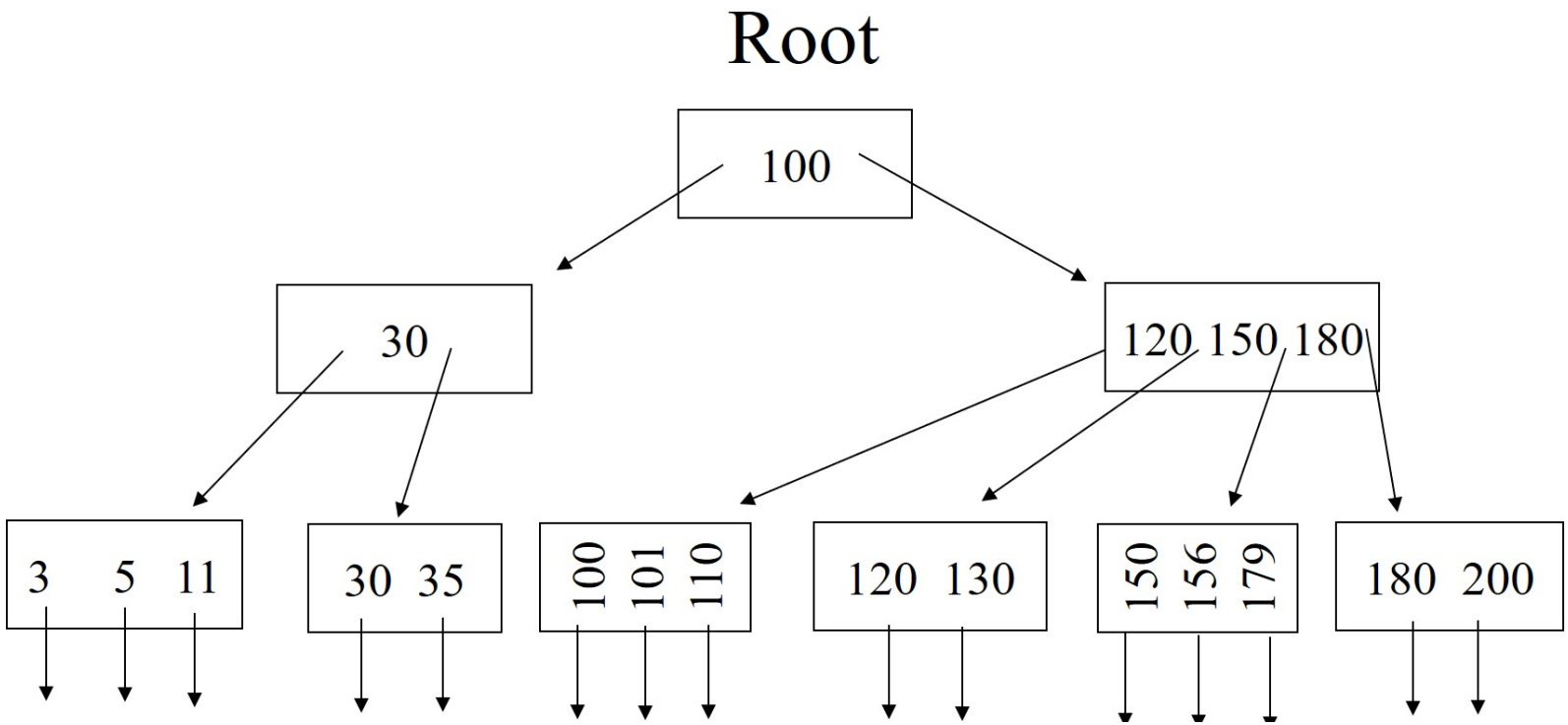
- Each node is stored in one block on disk
- `Root node`: the node at the "top" (or bottom depending how you draw the tree) of the tree
- `Internal/intermediate node`: a node that has one or more child node(s)
- `Leaf node`: a node that does not have any children nodes

Main Challenge in B$^+$Trees[1]

- To ensure good search cost, the tree should be kept of **minimum and uniform height**. Keep the tree:
  - full (packed), and
  - balanced (almost uniform height).
- This can be difficult – in face of insertions and deletions.
- **Solution**: Keep the tree "semi-full" (i.e., each tree node half-full)
  - Makes it easy to keep the tree balanced and semi-optimal.

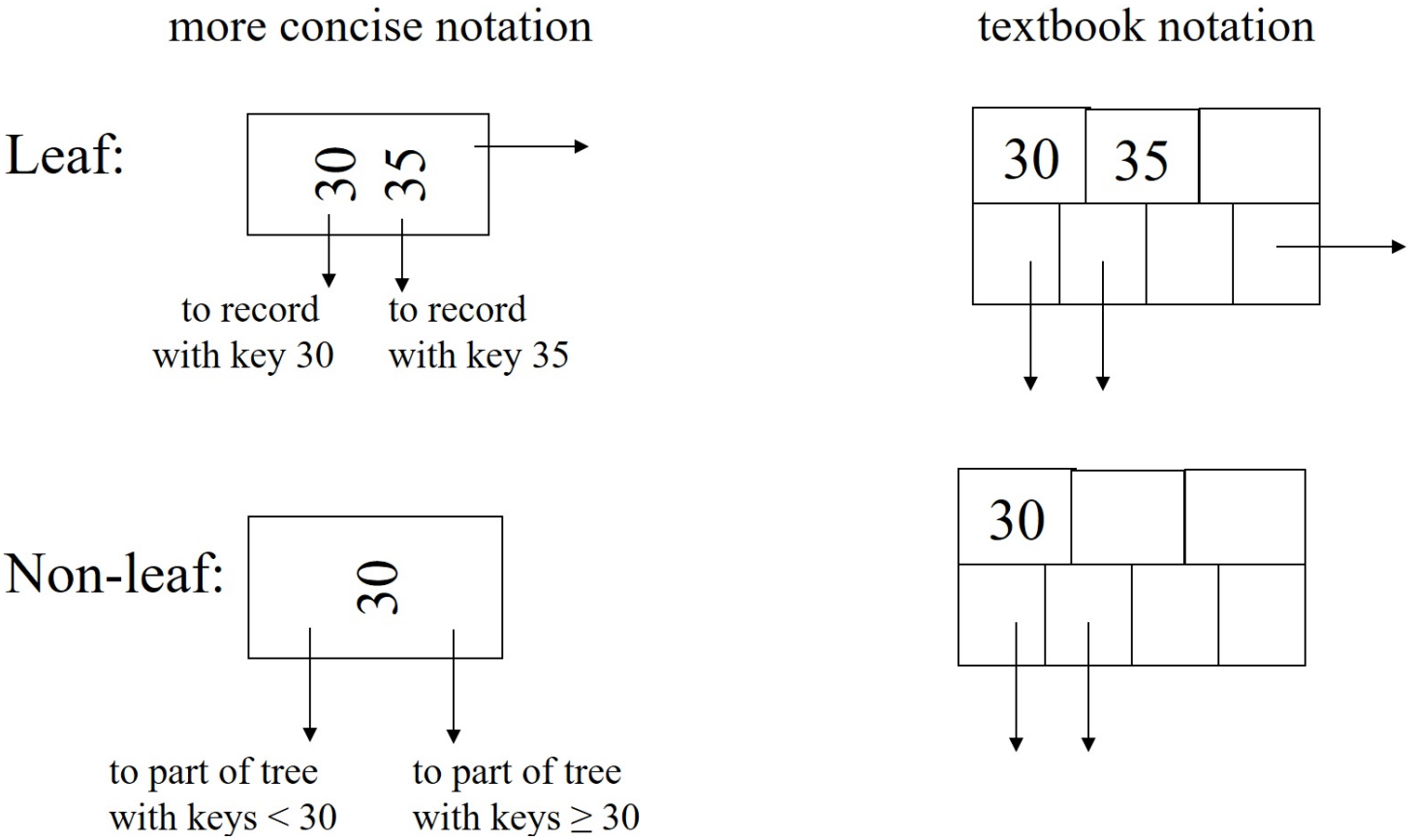
[1] Himanshu Gupta, Stony Brook University, CSE 532

B^+tree Example: $n = 3$

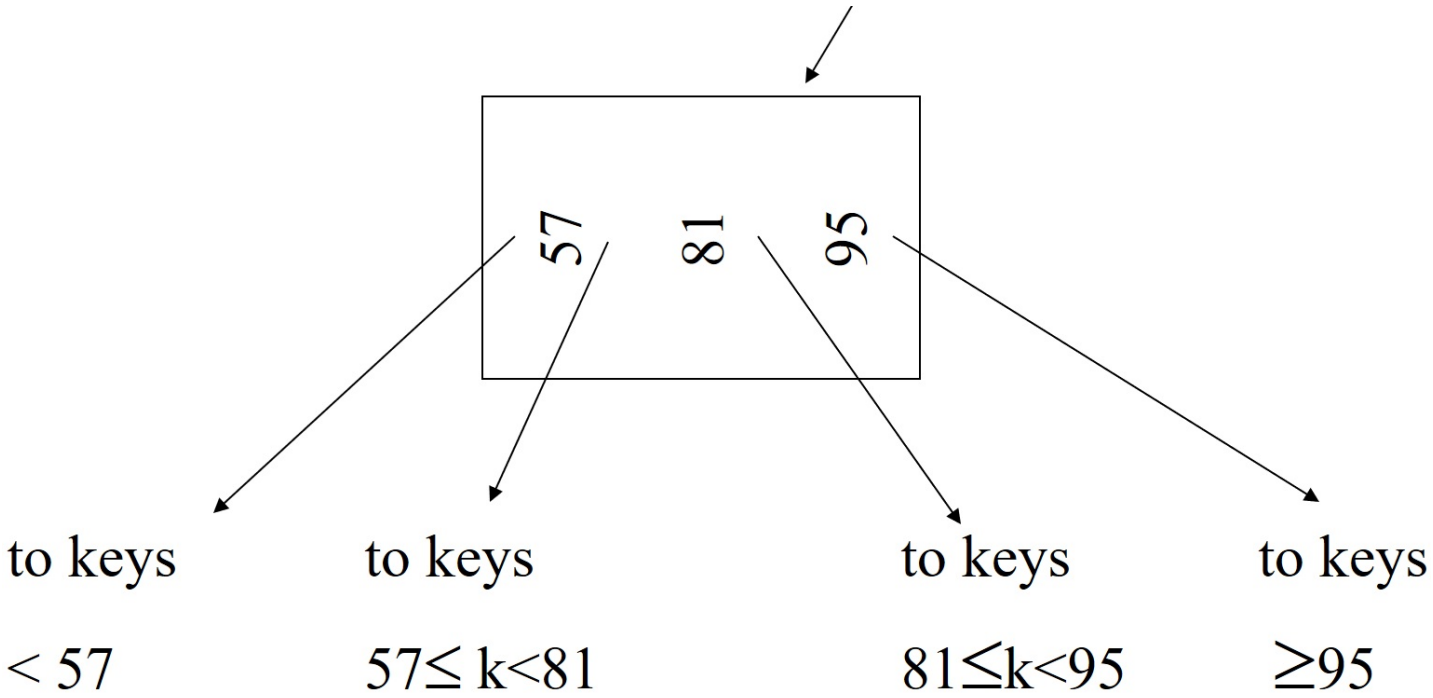


Example B^+tree nodes with $n = 3$

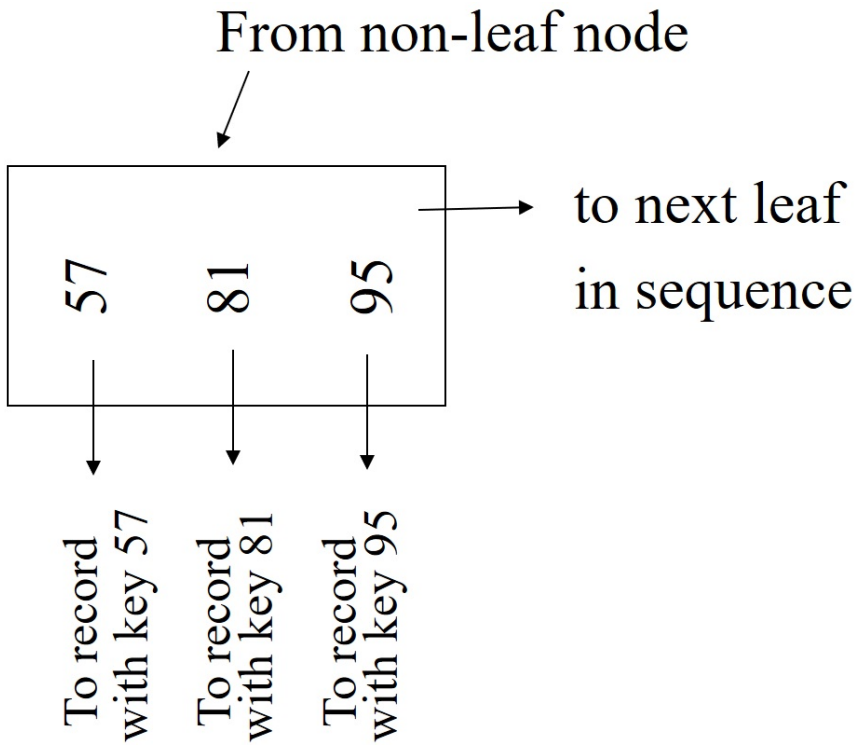
- Each internal node is stored in one block on disk
- and contains at most **n keys** and **(n+1) pointers**



Sample non-leaf

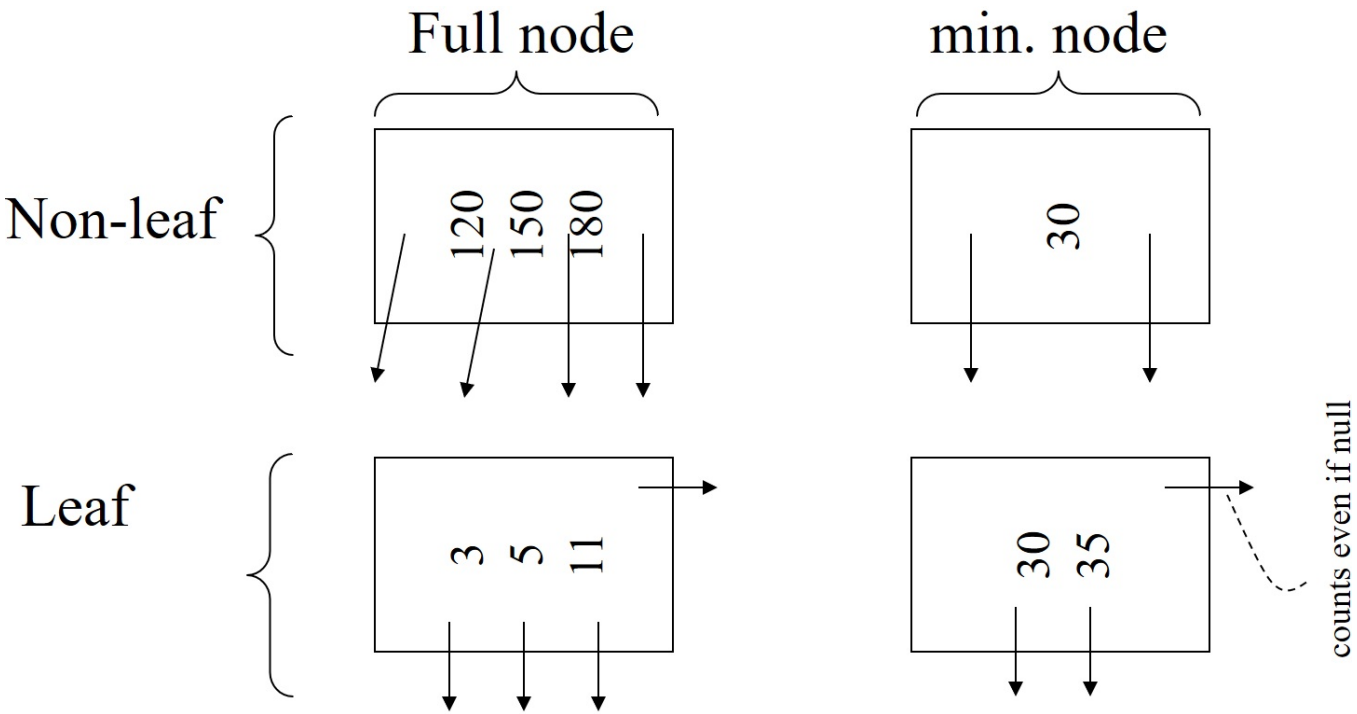


Sample leaf node



The last pointer points to the next leaf node (a disk block) in the B^+-tree

$n = 3$



- Each internal node is stored in one block on disk
- and contains at most **n keys** and **(n+1) pointers**

$$\textit{Size of nodes}: \begin{cases} n+1 \textit{ pointers} & \\ & (\textit{fixed}) \\ n \textit{ keys} & \end{cases}$$

Don't want nodes to be too empty

- Use a "Fill Factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for $\texttt{B}^{+}\texttt{tree}$
- Use at least (to ensure a balanced tree)
  - `Non-leaf`: $\left\lceil \frac{n+1}{2} \right\rceil$ pointers (to nodes)
    - Except `root`: required at least 2 be used
  - `Leaf`: $\left\lfloor \frac{n+1}{2} \right\rfloor$ pointers (to data)

Number of pointers/keys for B^+tree

	Max ptrs	Max keys	Min ptrs$\rightarrow$ data	Min keys
Non-leaf (non-root)	$n+1$	n	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil - 1$
Leaf (non-root)	$n+1$	n	$\lfloor \frac{n+1}{2} \rfloor$	$\lfloor \frac{n+1}{2} \rfloor$
Root	$n+1$	n	2*	1

*When there is only one record in the B^+tree, min pointers in the root is 1 (the other pointers are null)

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"

Insert into B$^+$tree

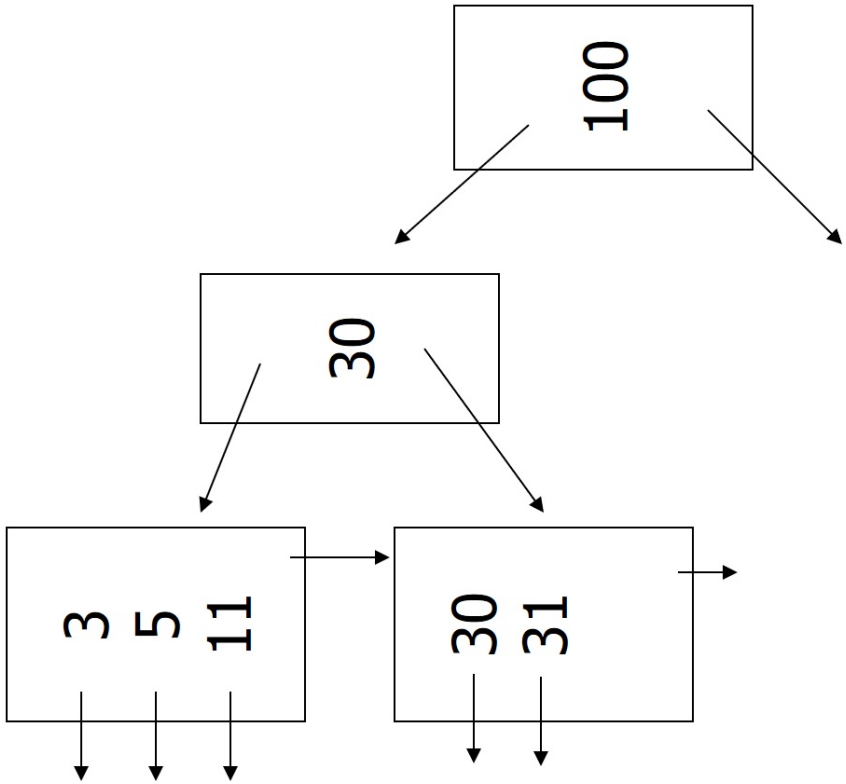
a) simple case
   - space available in `leaf`

b) `leaf` overflow

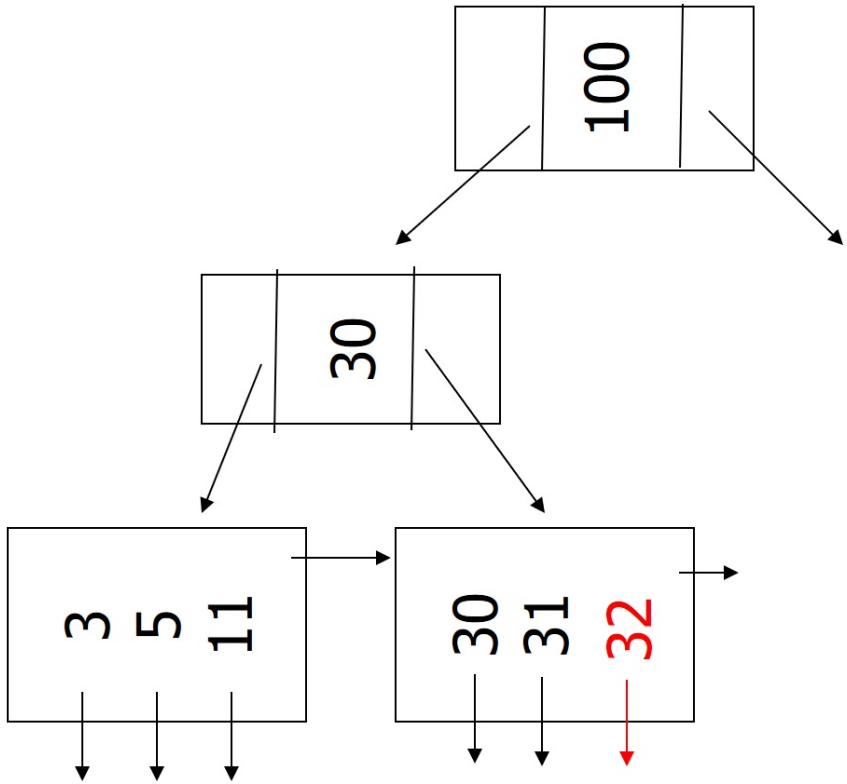
c) `non-leaf` overflow

d) new `root`

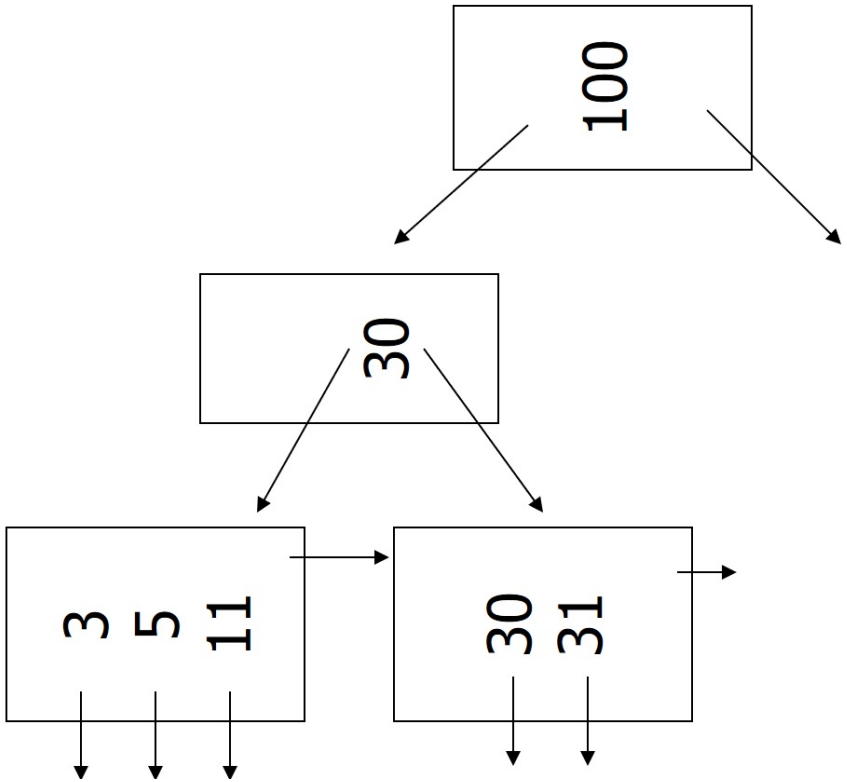
a) Insert key $= 32$, $n = 3$



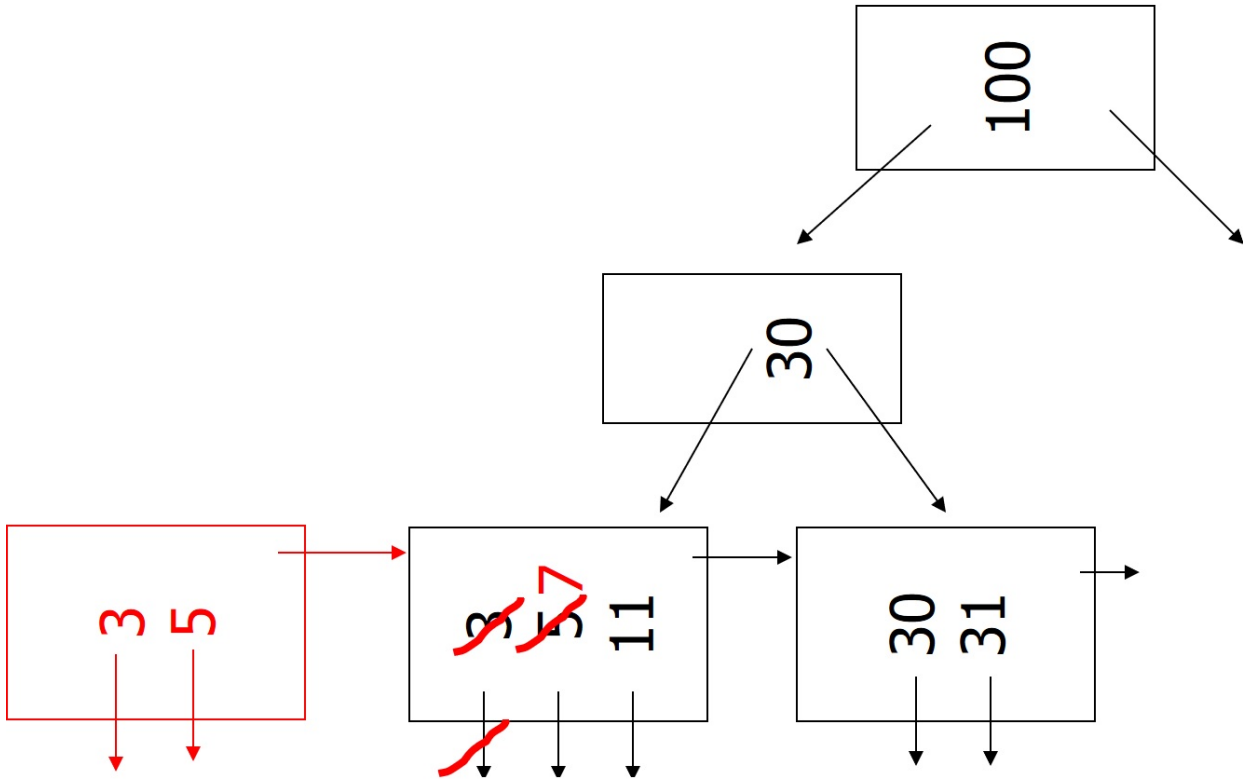
a) Insert key $= 32$, $n = 3$



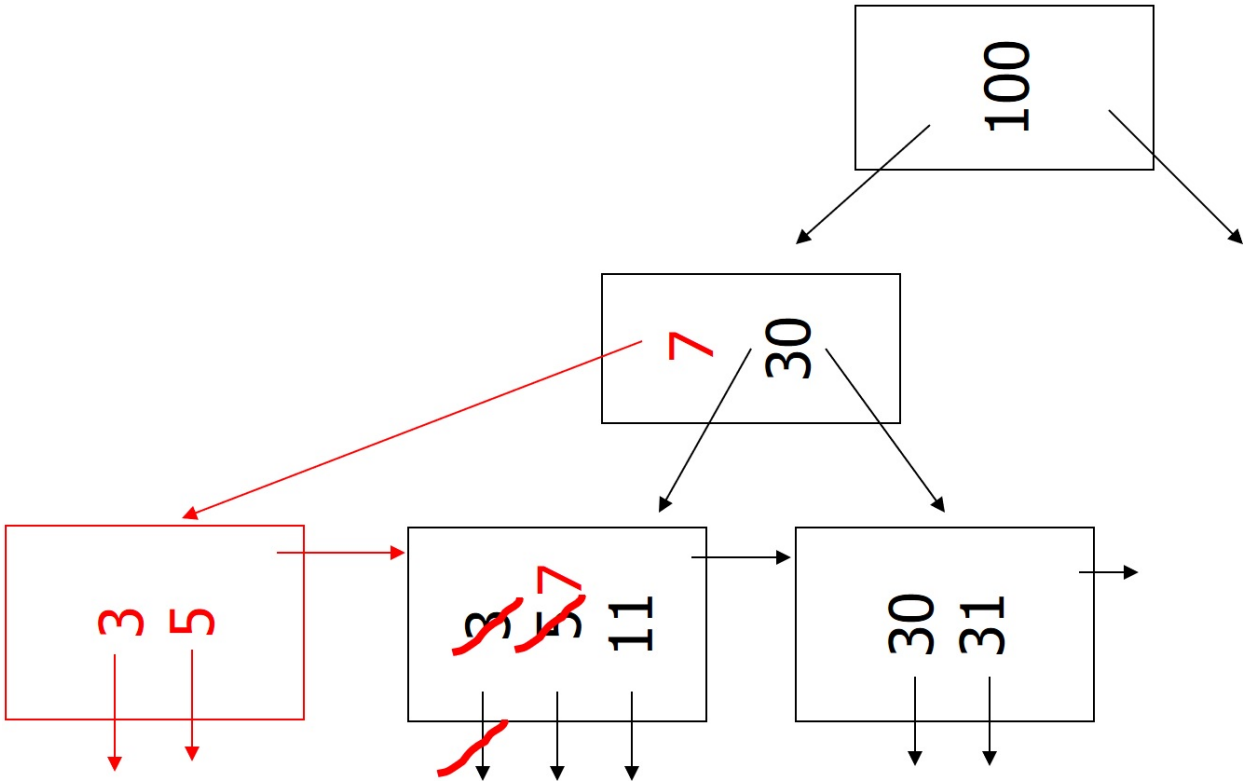
b) Insert key $= 7$, $n = 3$



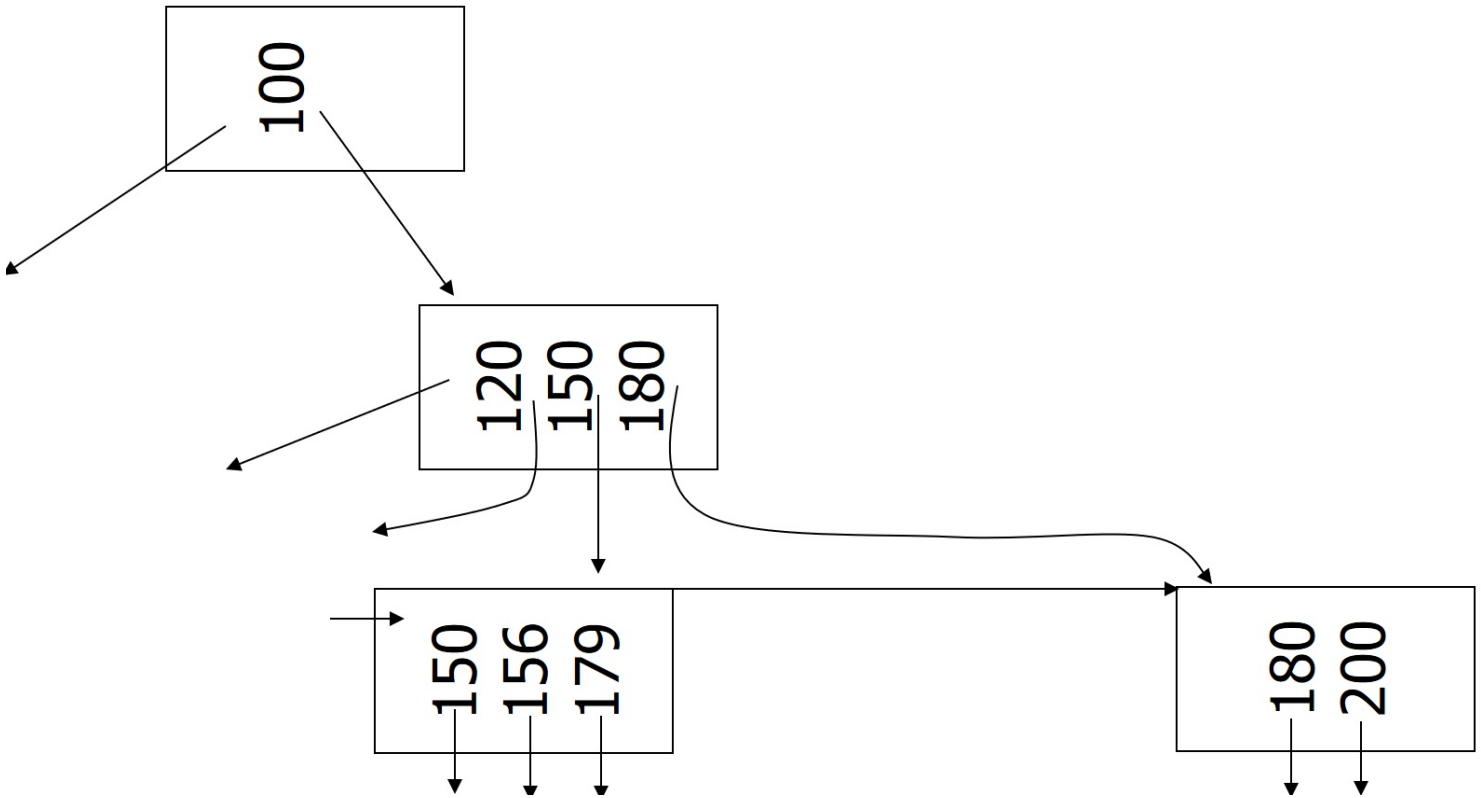
b) Insert key $= 7$, $n = 3$



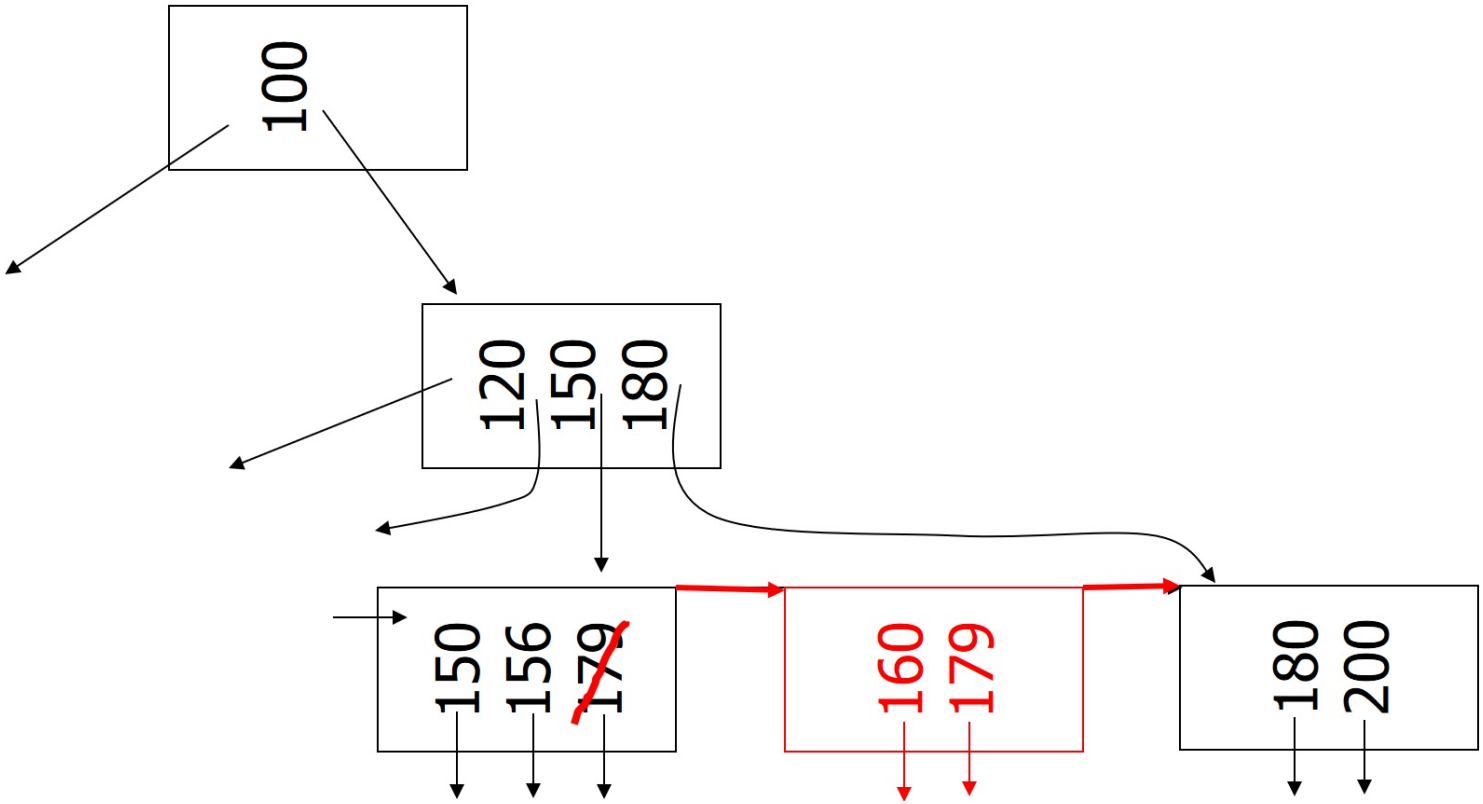
b) Insert key $= 7$, $n = 3$



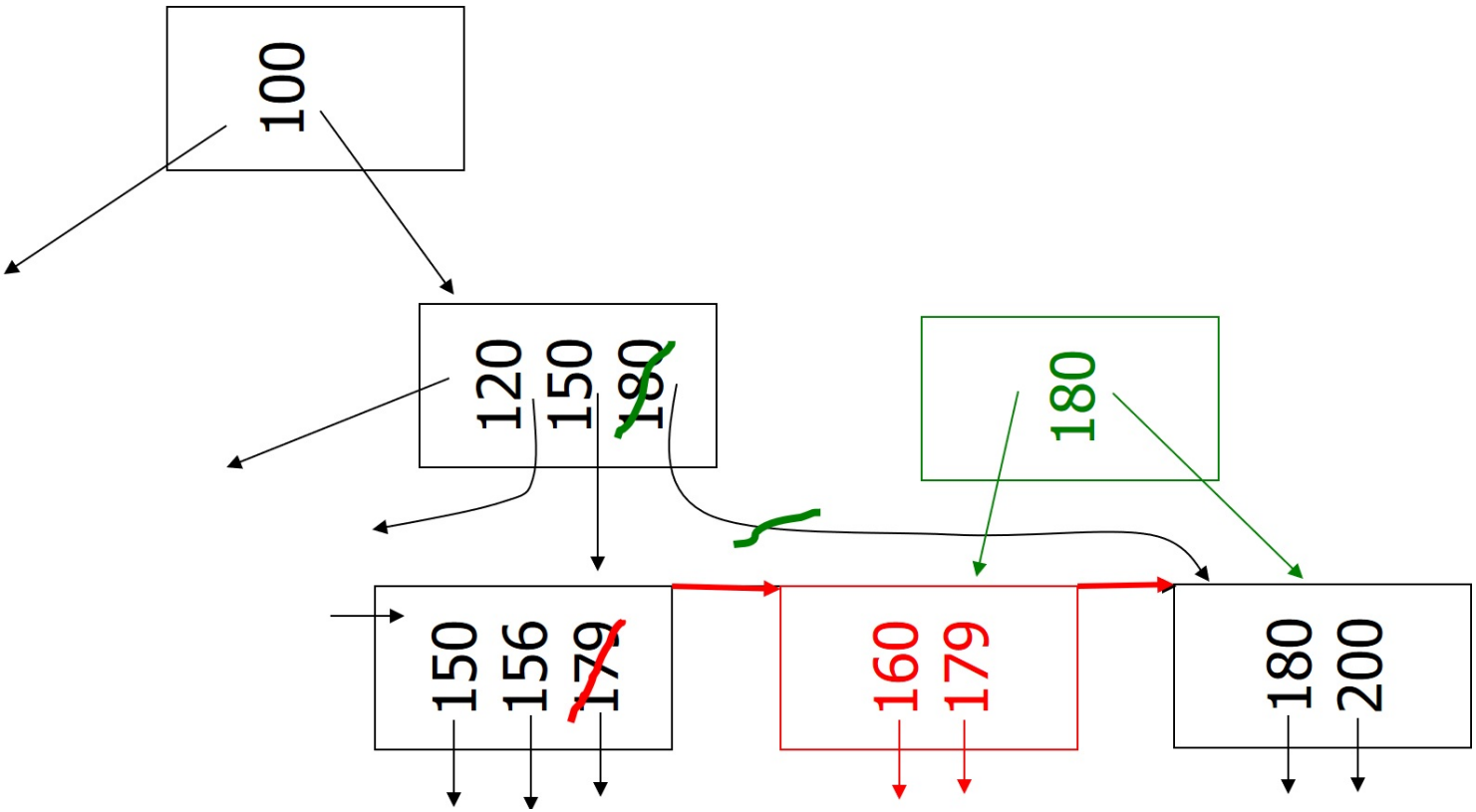
c) Insert key = 160, $n = 3$



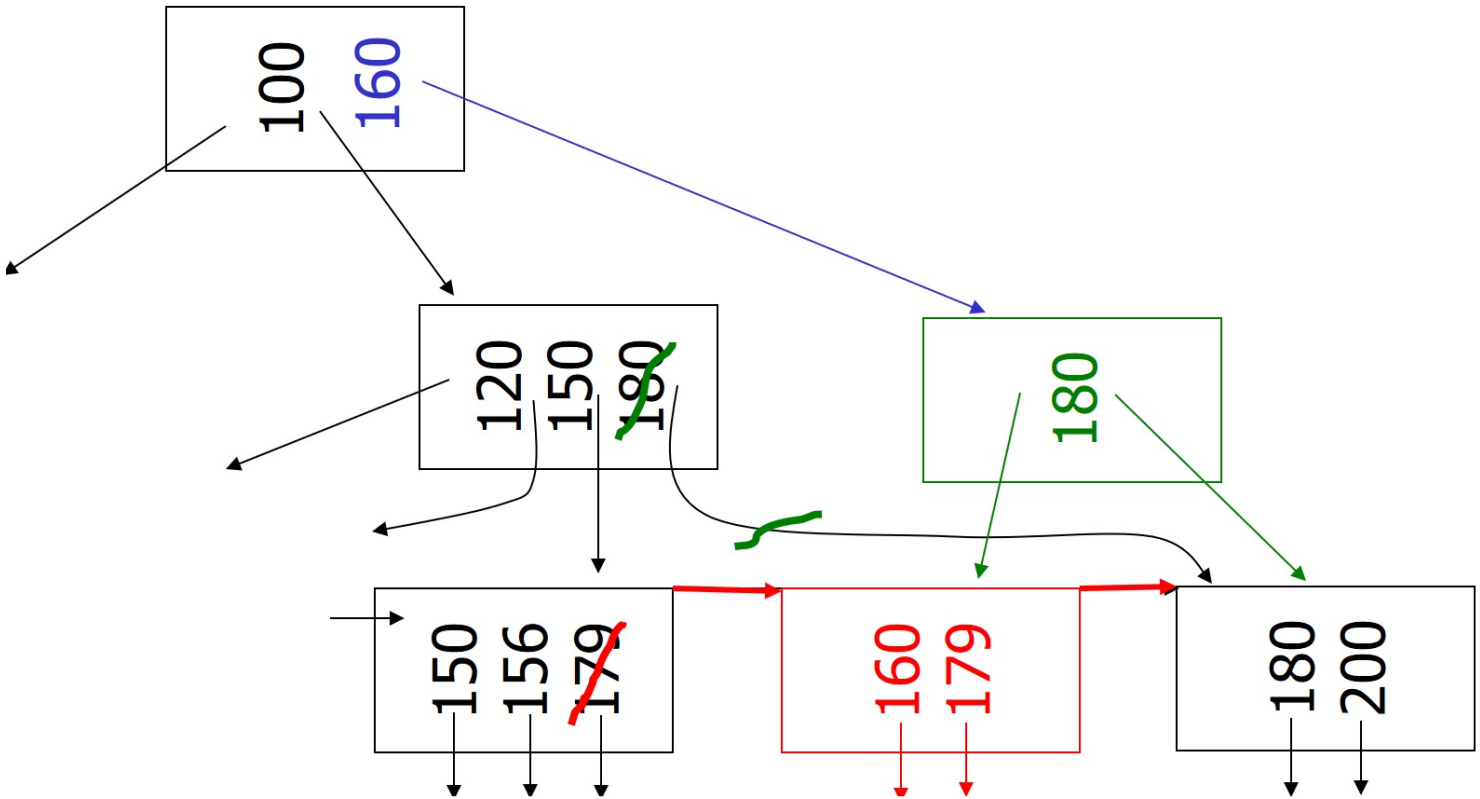
c) Insert key = 160, $n = 3$



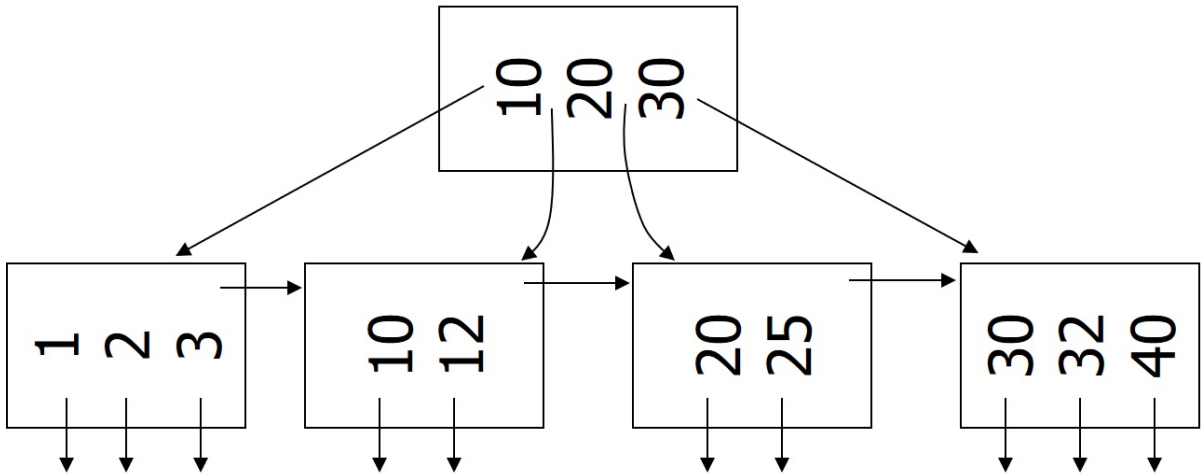
c) Insert key = 160, $n = 3$



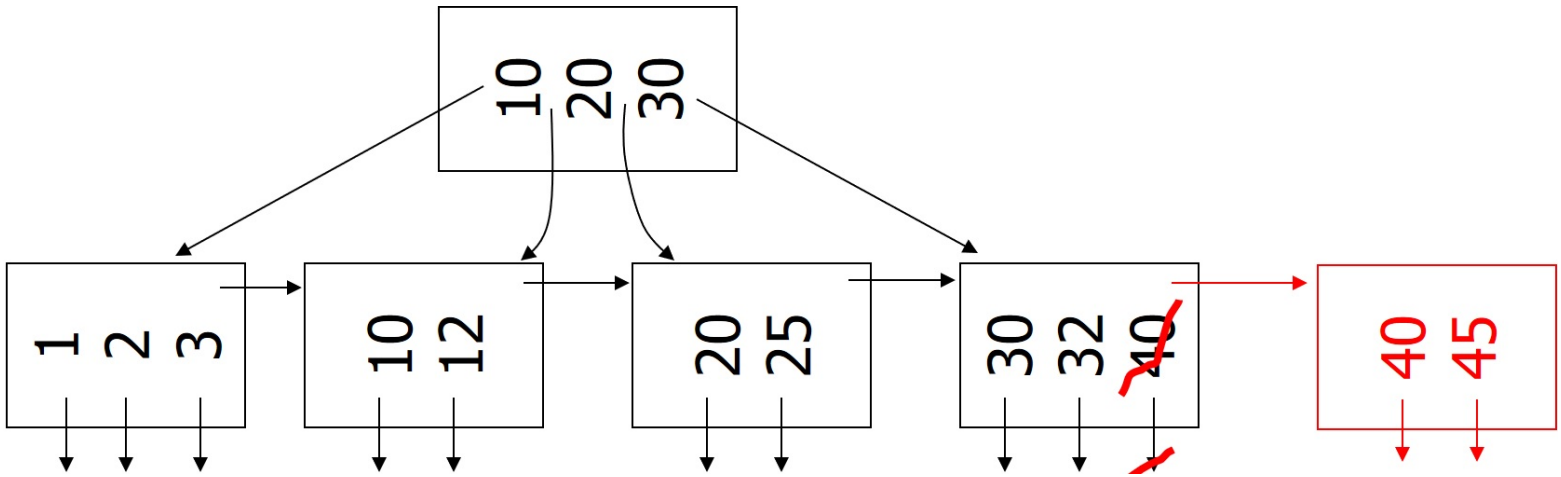
c) Insert key = 160, $n = 3$



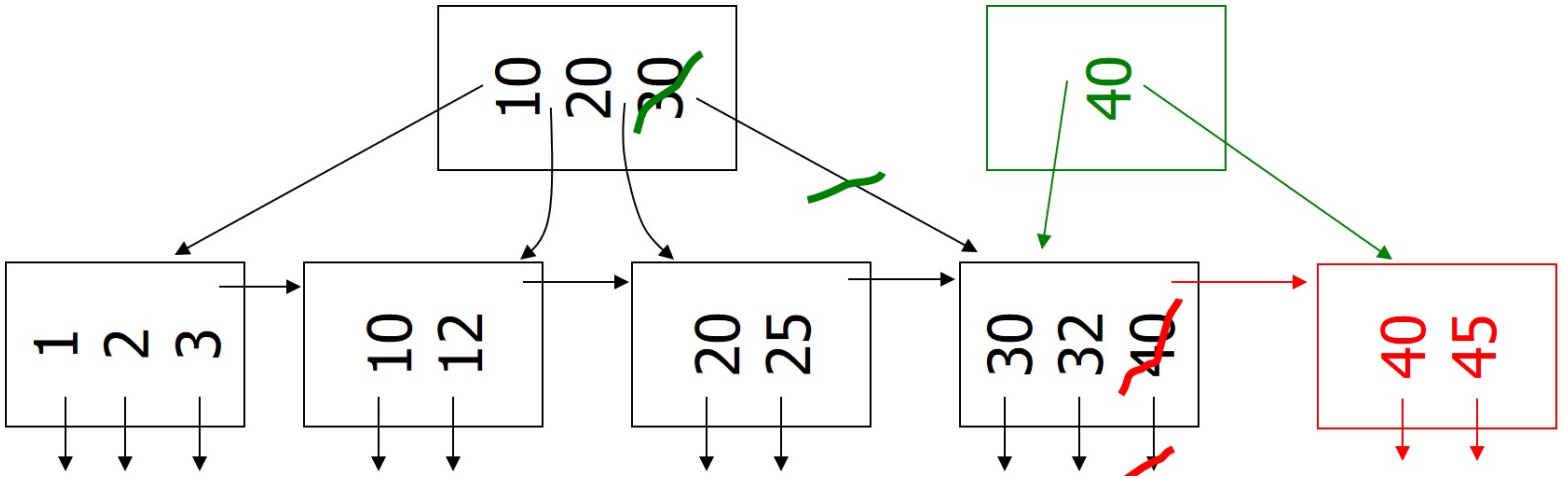
d) New `root`, insert 45, $n = 3$



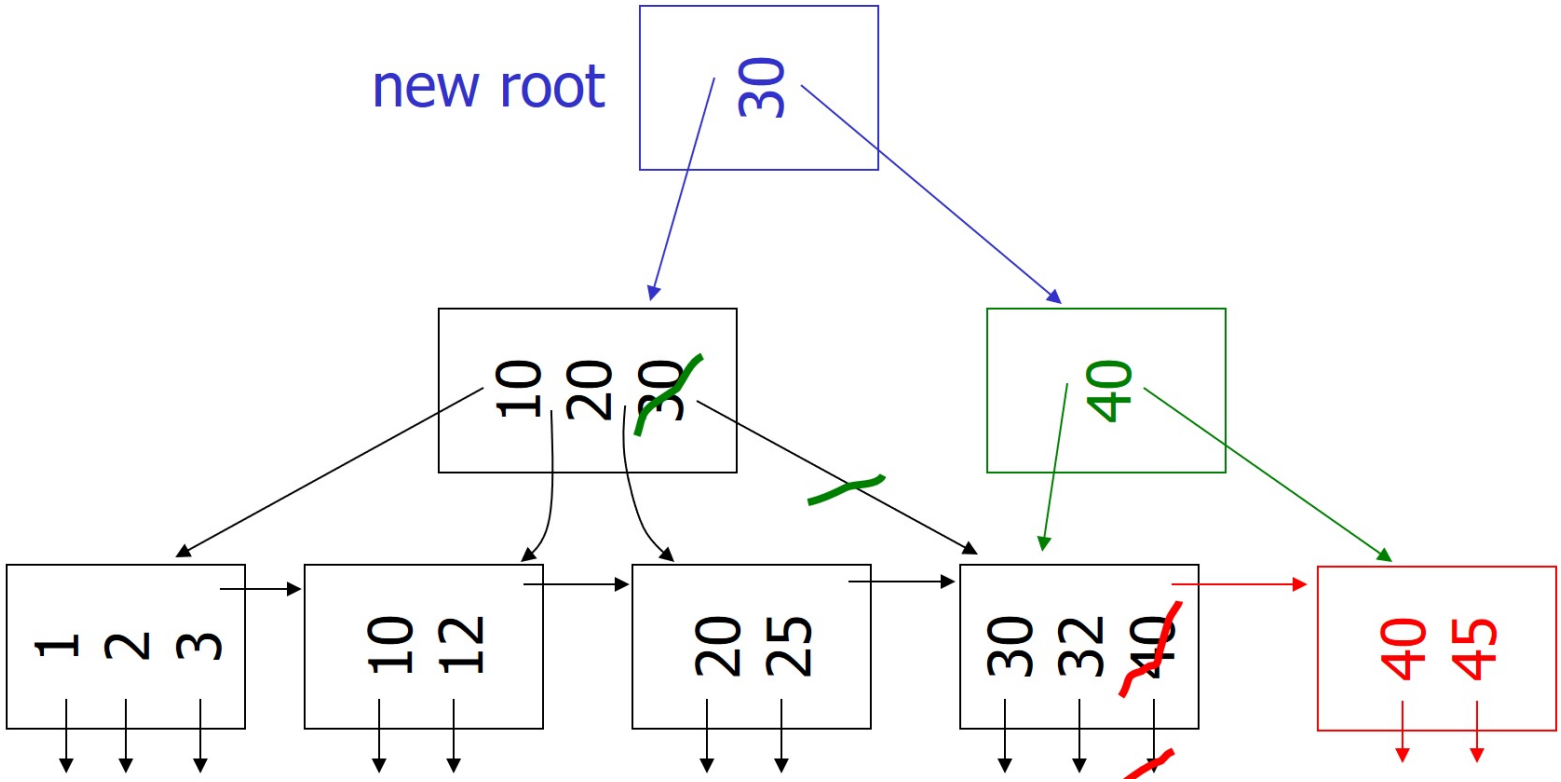
d) New `root`, insert 45, $n = 3$



d) New `root`, insert 45, $n = 3$



d) New `root`, insert 45, $n = 3$



Insertion Algorithm

- Insert Record with `key k`
- Search `leaf node` for `k`
  - `Leaf node` has at least one space
    - Insert into `leaf`
  - `Leaf` is full
    - `Split` leaf into two nodes (new leaf)
    - `Insert` new leaf's smallest key into parent

Insertion Algorithm (cont.)

- `Non-leaf` node is full
  - `Split` parent
  - `Insert` median key into parent
- `Root` is full
  - `Split` root
  - `Create` new root with two pointers and single key
- B^+-trees grow at the root

Deletion from $\texttt{B}^{+}\texttt{tree}$

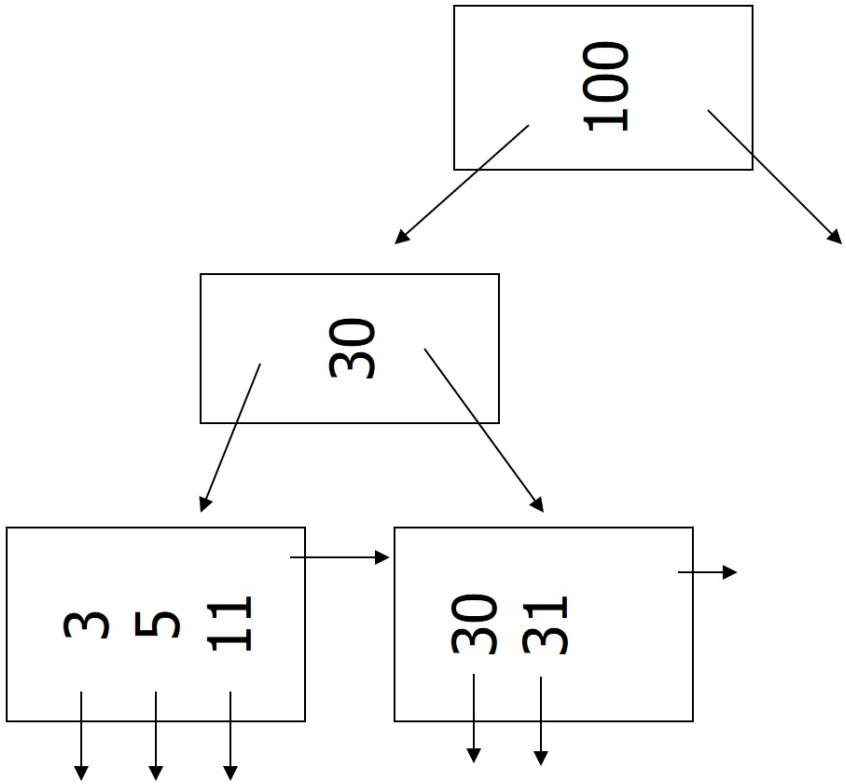
a) Simple case

b) Coalesce with neighbor (sibling)

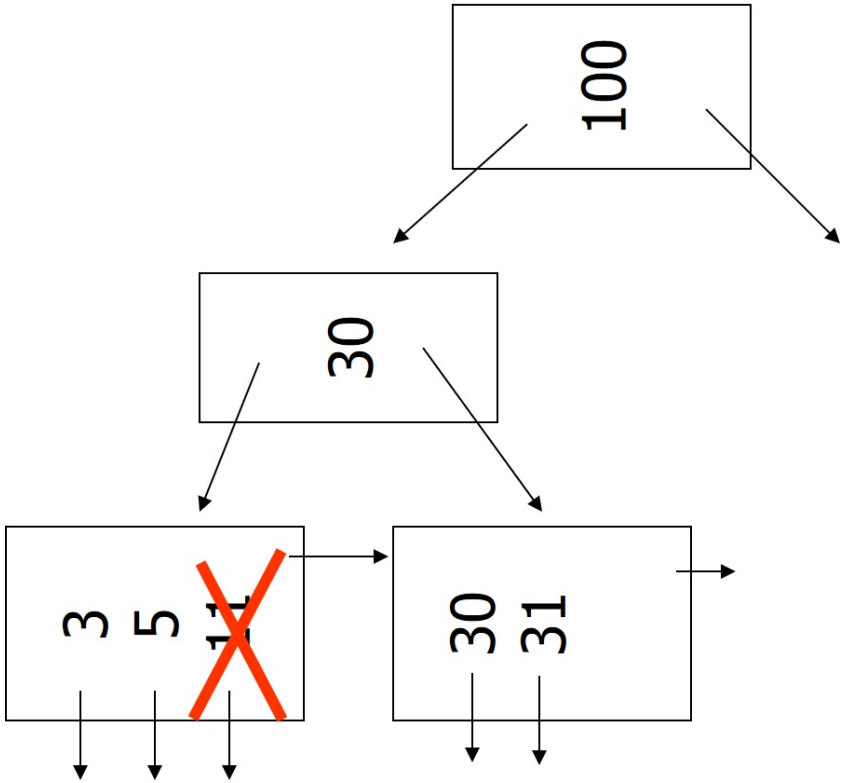
c) Re-distribute keys

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

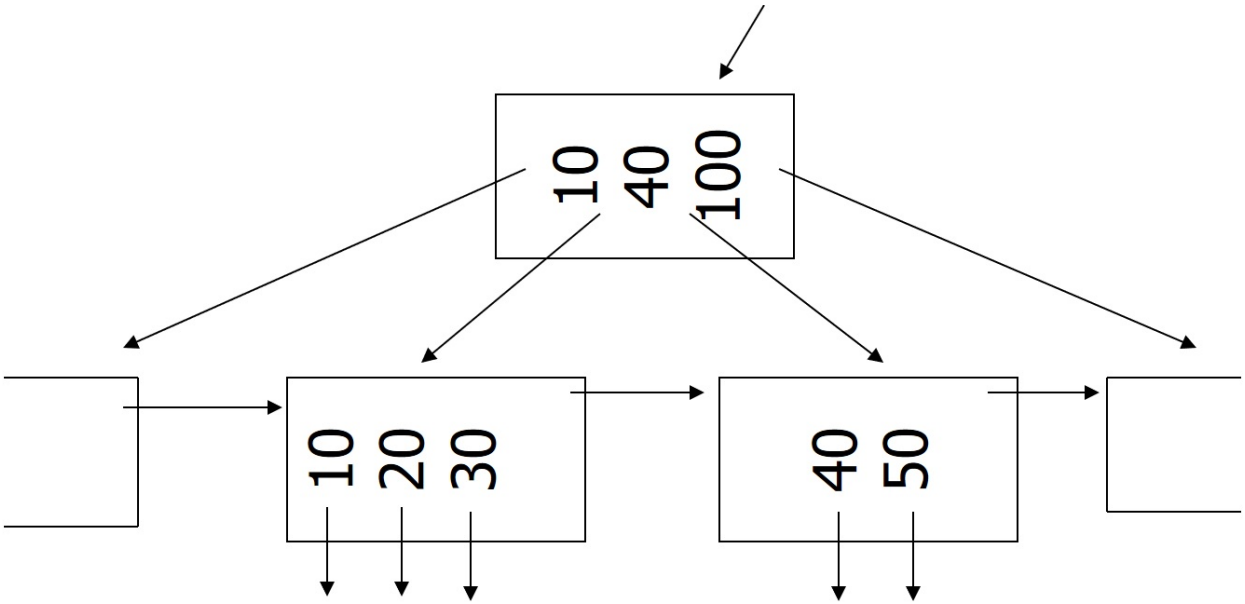
a) Delete key = 11, $n = 3$



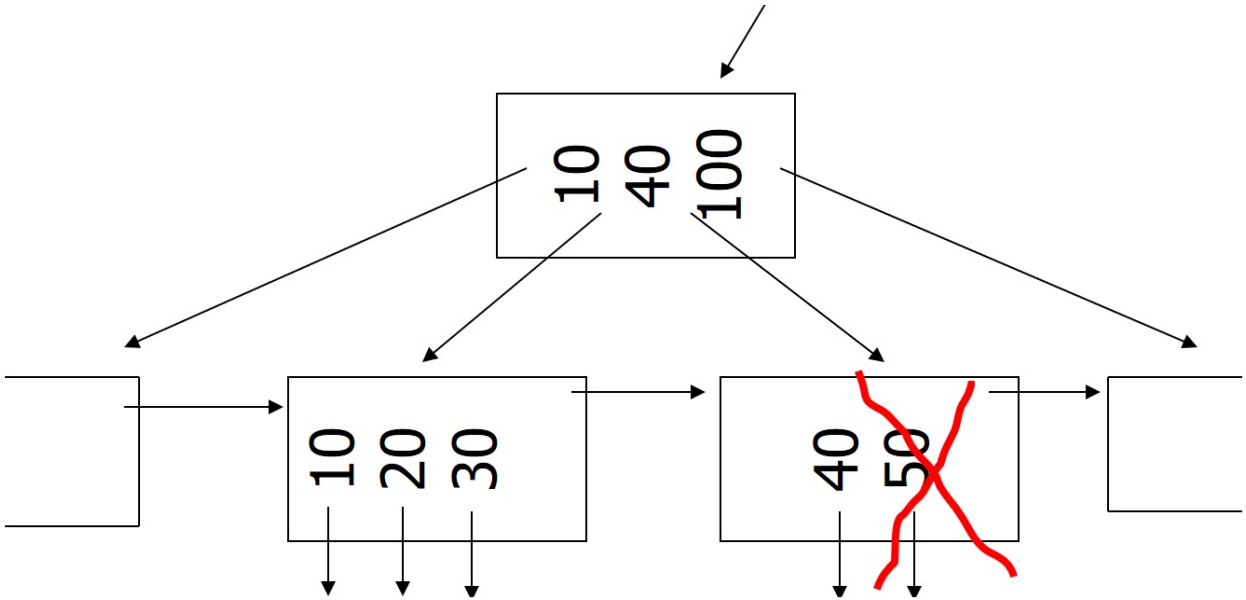
a) Delete key = 11, $n = 3$



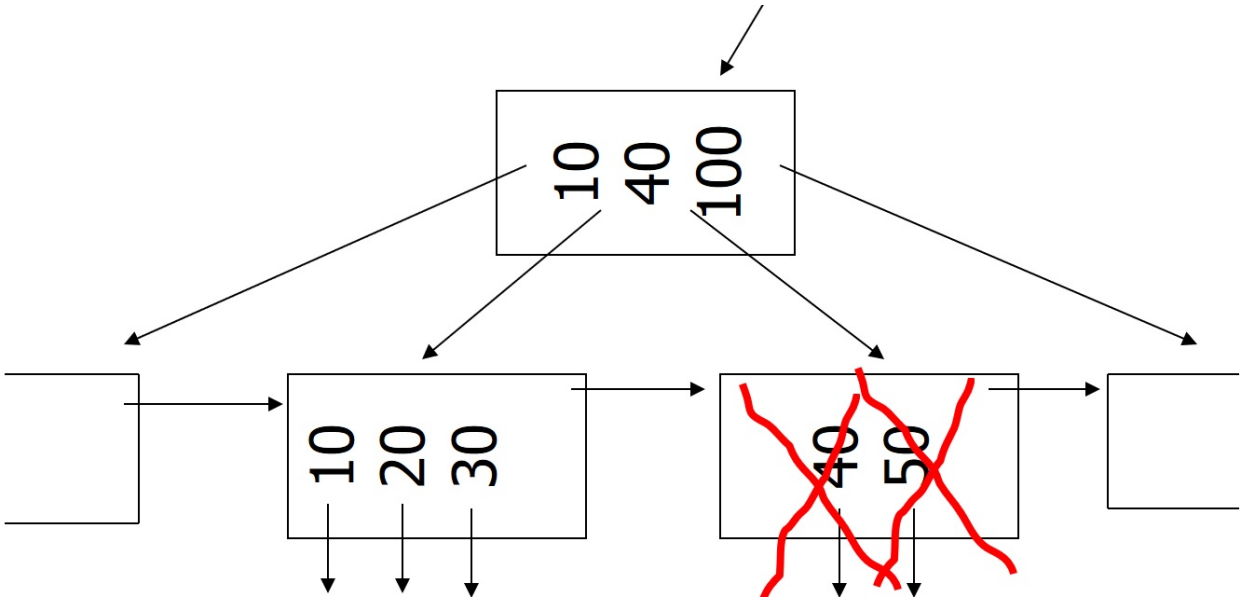
b) Coalesce with sibling: Delete 50, $n = 4$



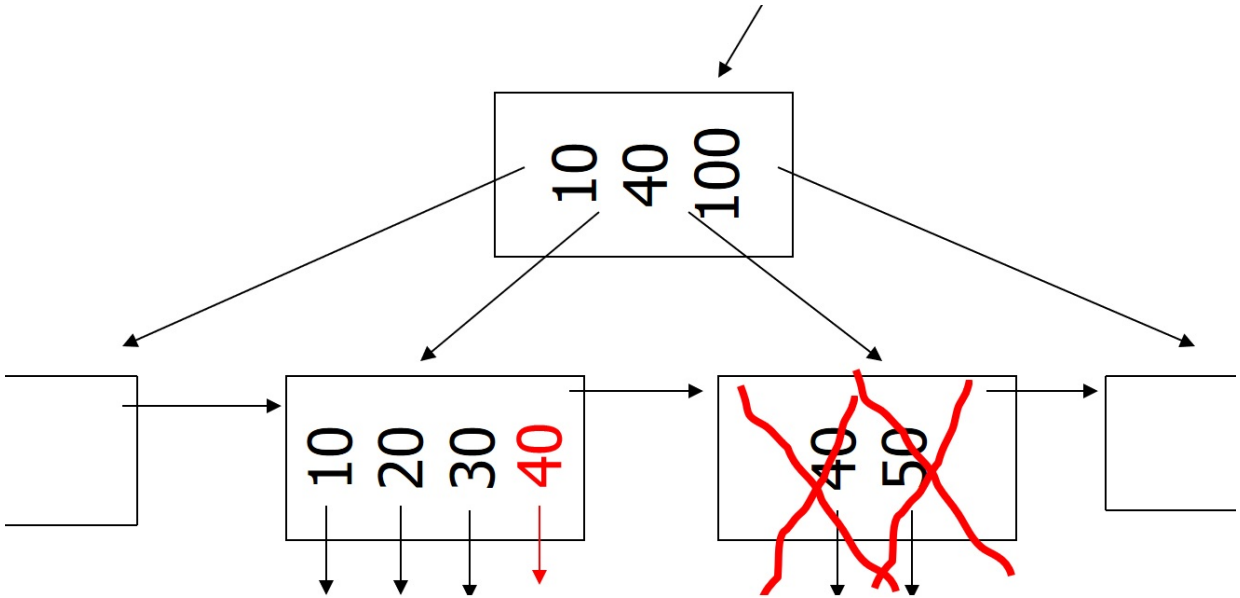
b) Coalesce with sibling: Delete 50, $n = 4$



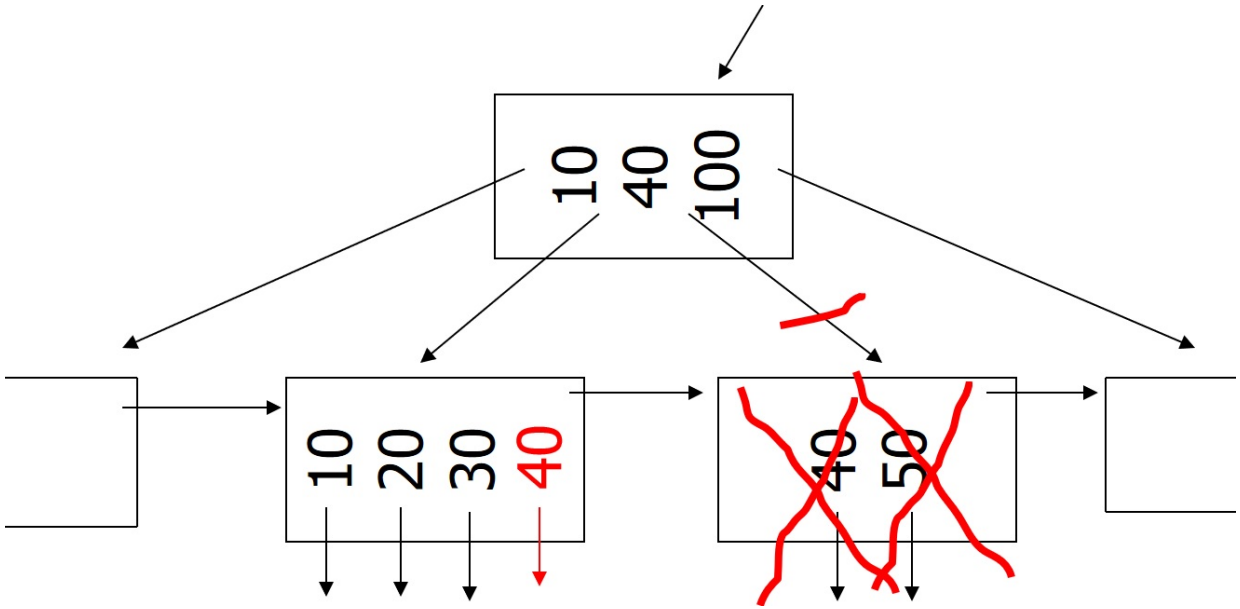
b) Coalesce with sibling: Delete 50, $n = 4$



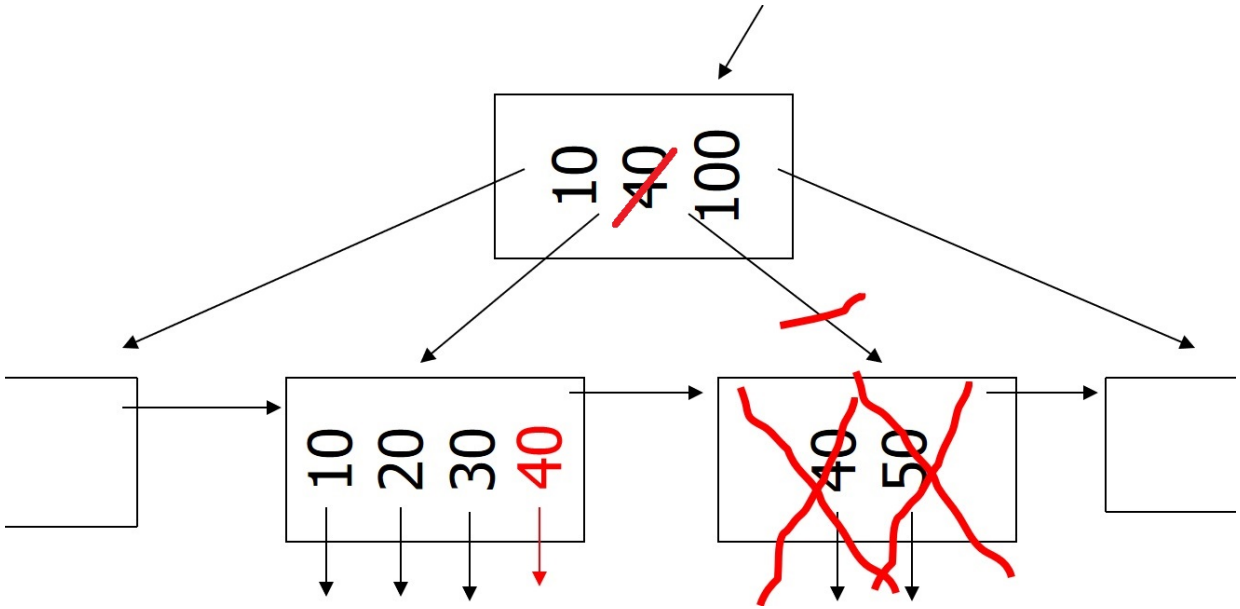
b) Coalesce with sibling: Delete 50, $n = 4$



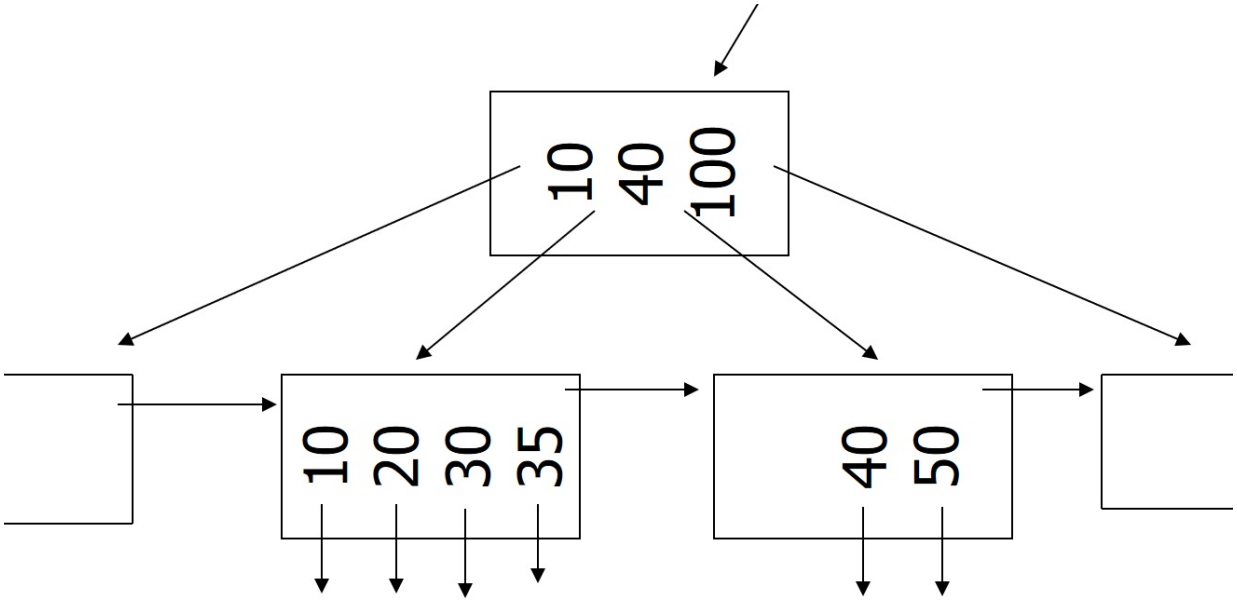
b) Coalesce with sibling: Delete 50, $n = 4$



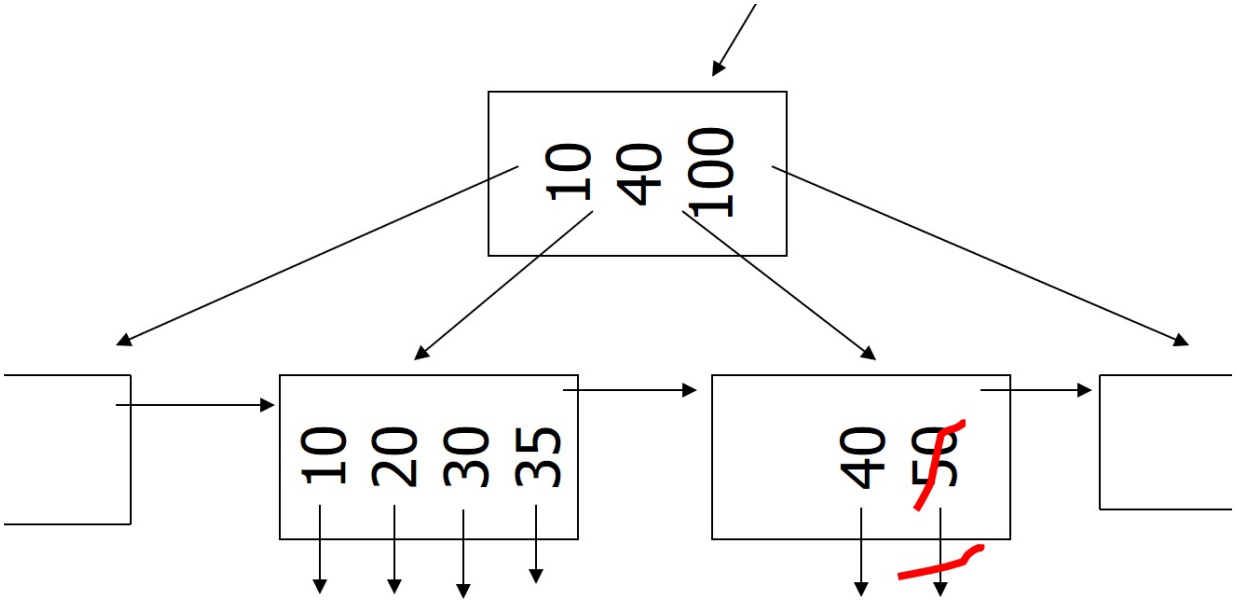
b) Coalesce with sibling: Delete 50, $n = 4$



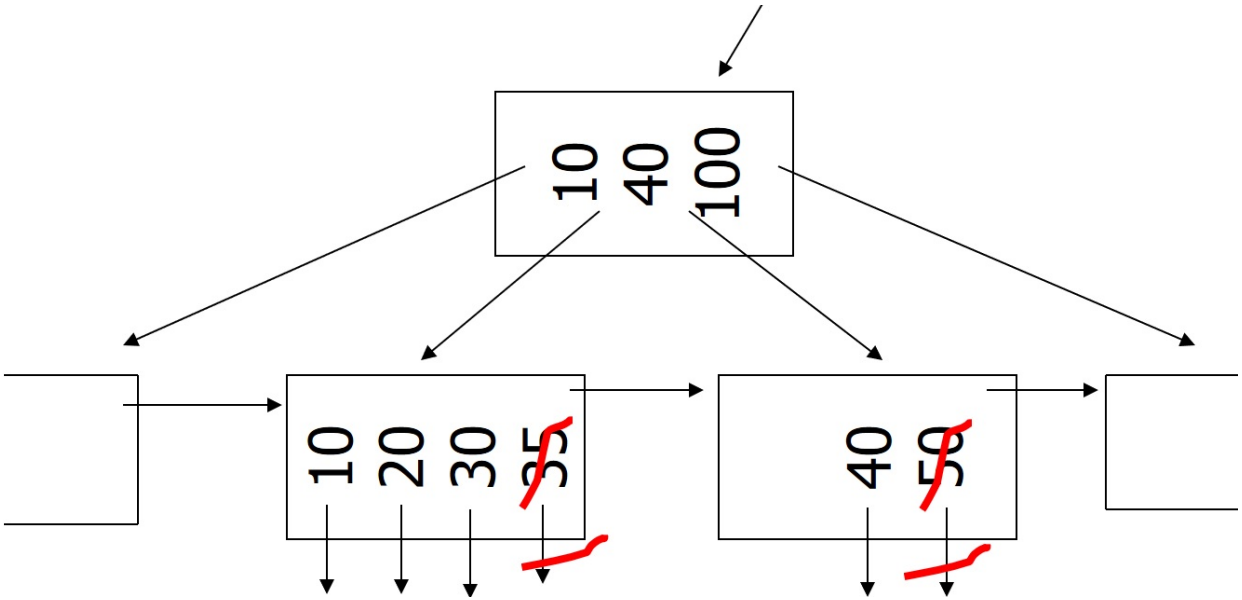
c) Redistribute keys: Delete 50, $n = 4$



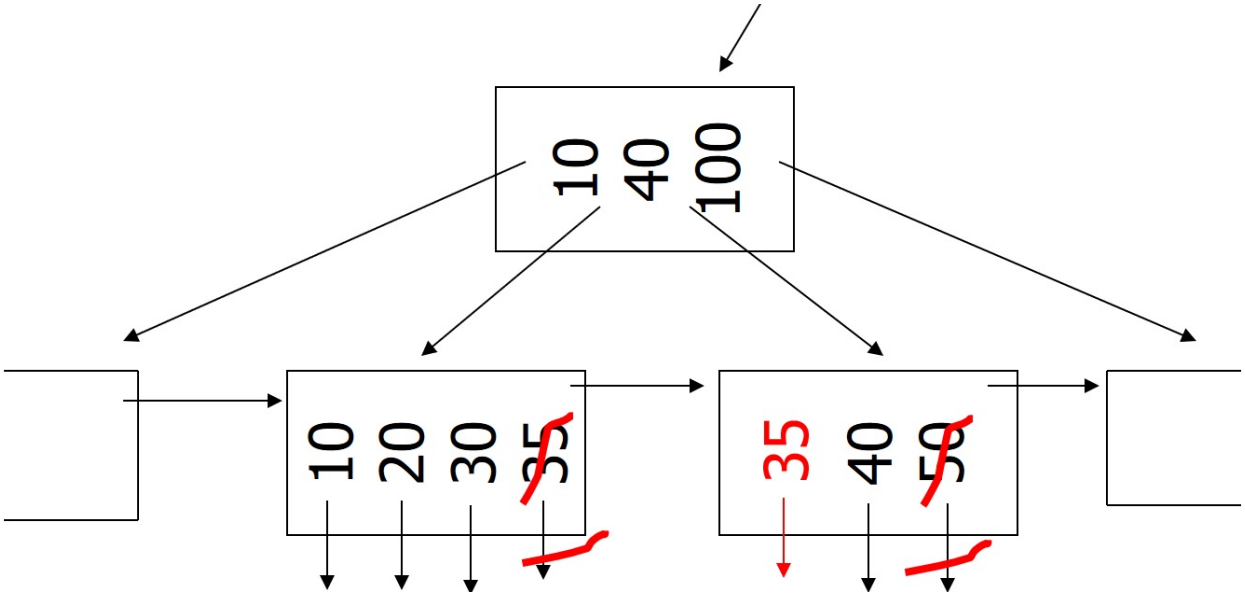
c) Redistribute keys: Delete 50, $n = 4$



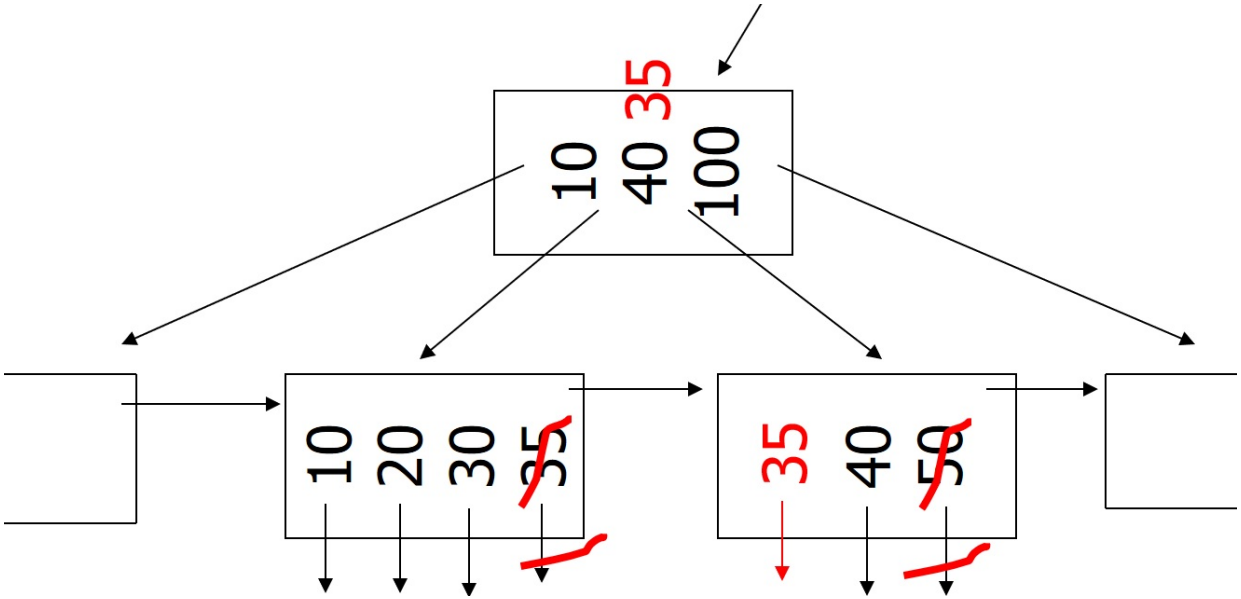
c) Redistribute keys: Delete 50, $n = 4$



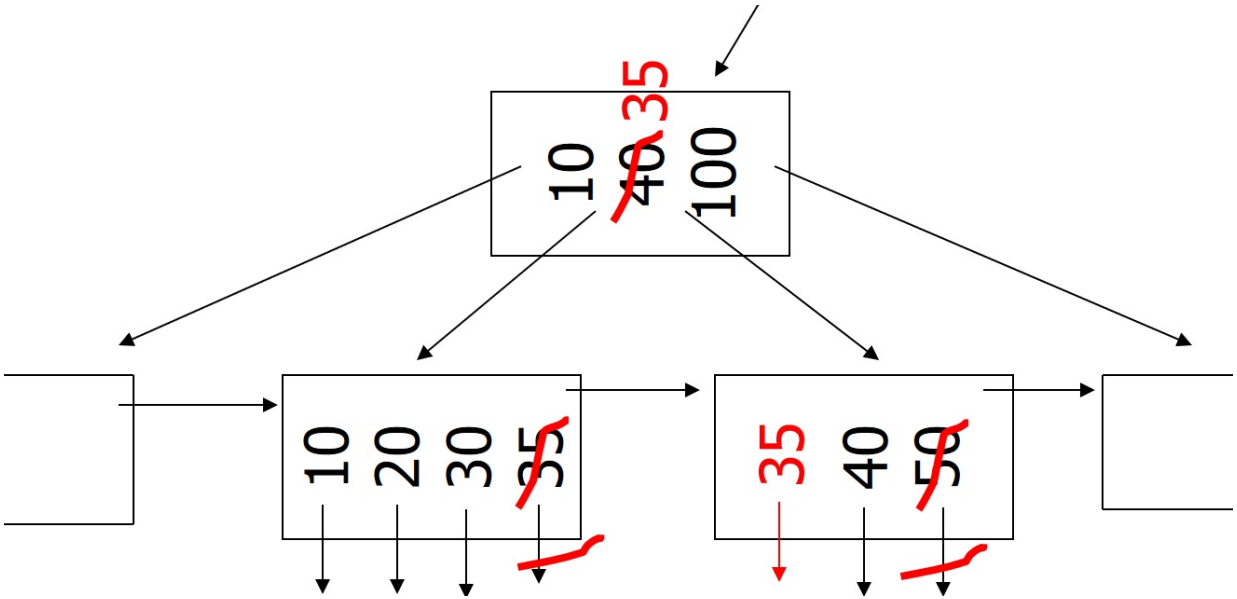
c) Redistribute keys: Delete 50, $n = 4$



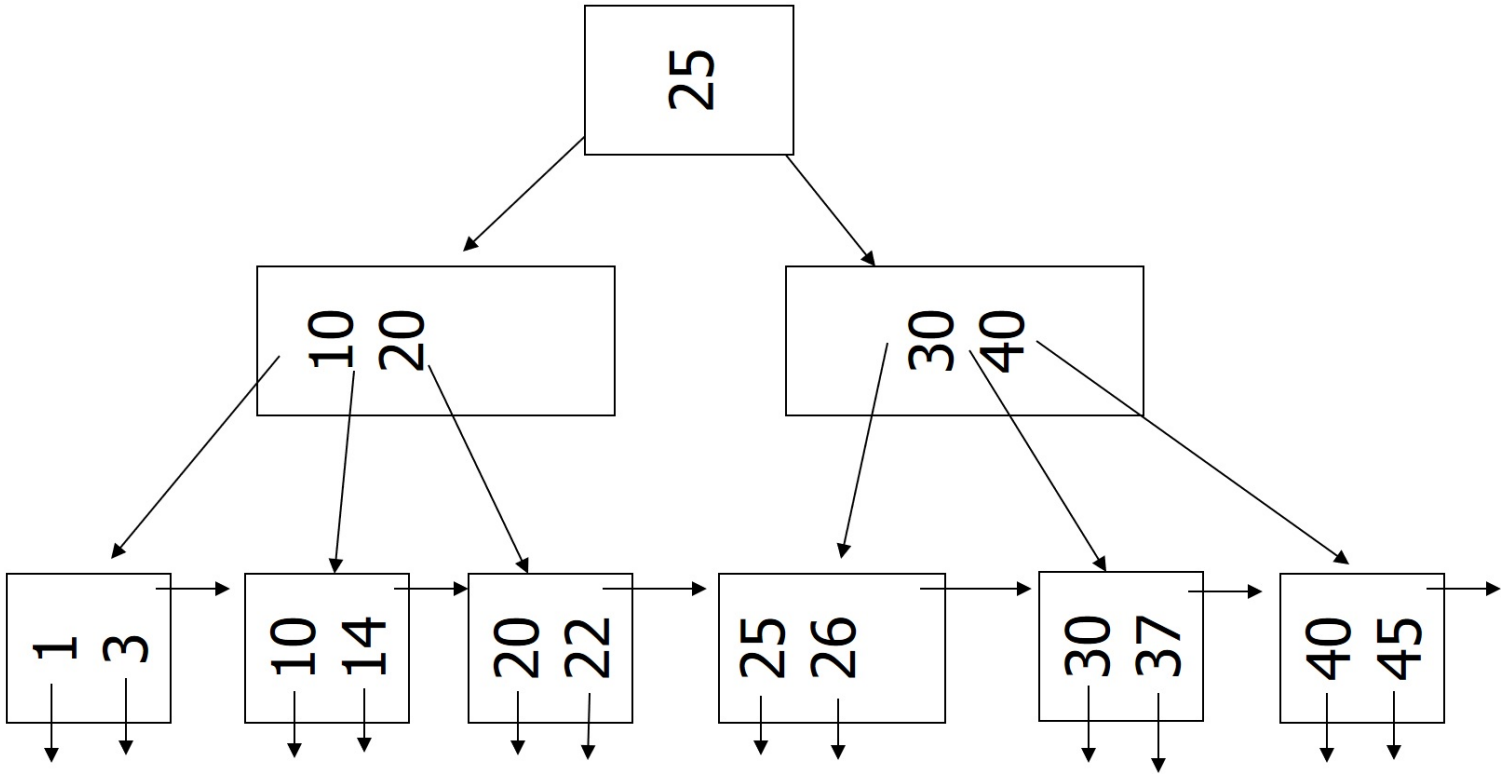
c) Redistribute keys: Delete 50, $n = 4$



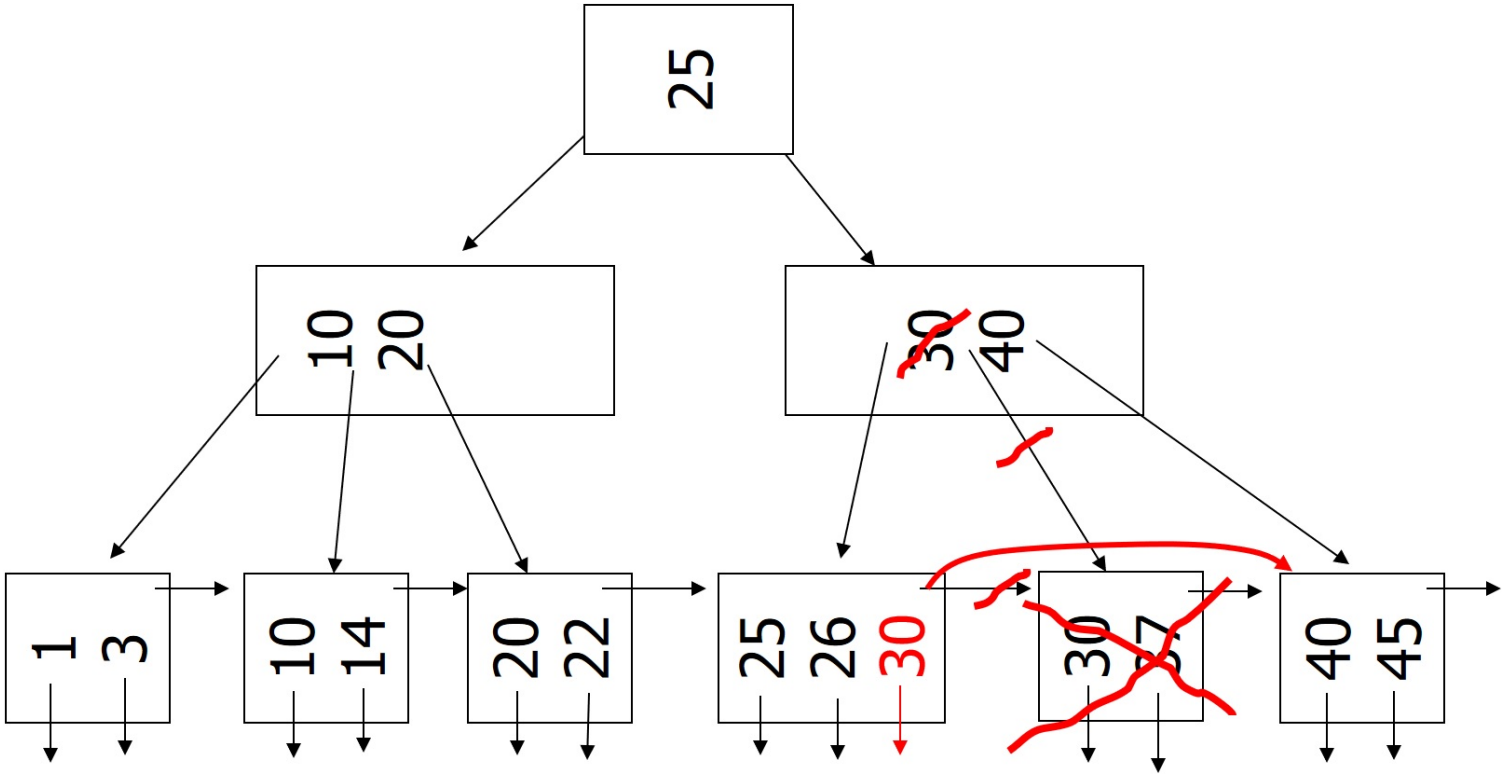
c) Redistribute keys: Delete 50, $n = 4$



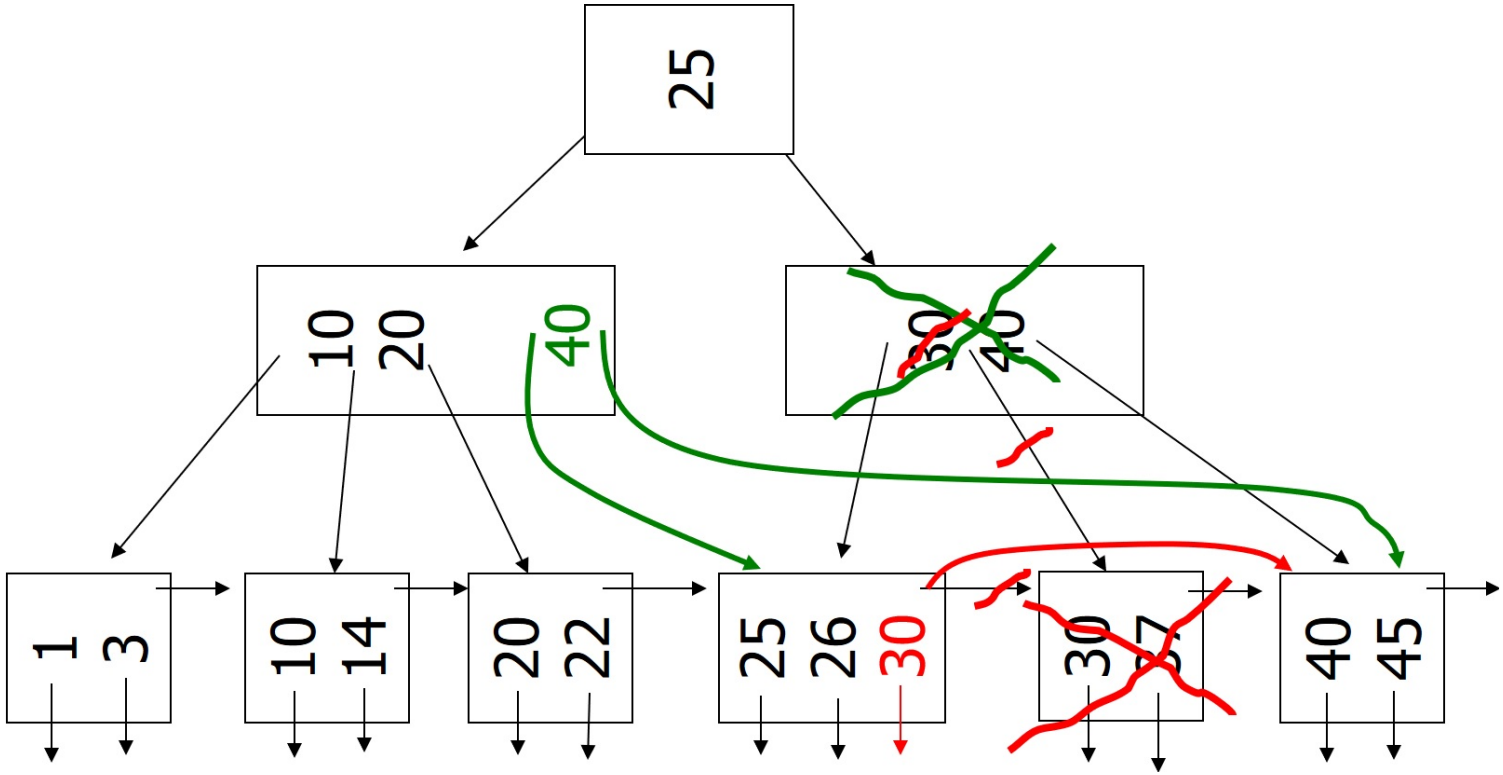
d) Non-leaf Coalesce: Delete 37, $n = 4$



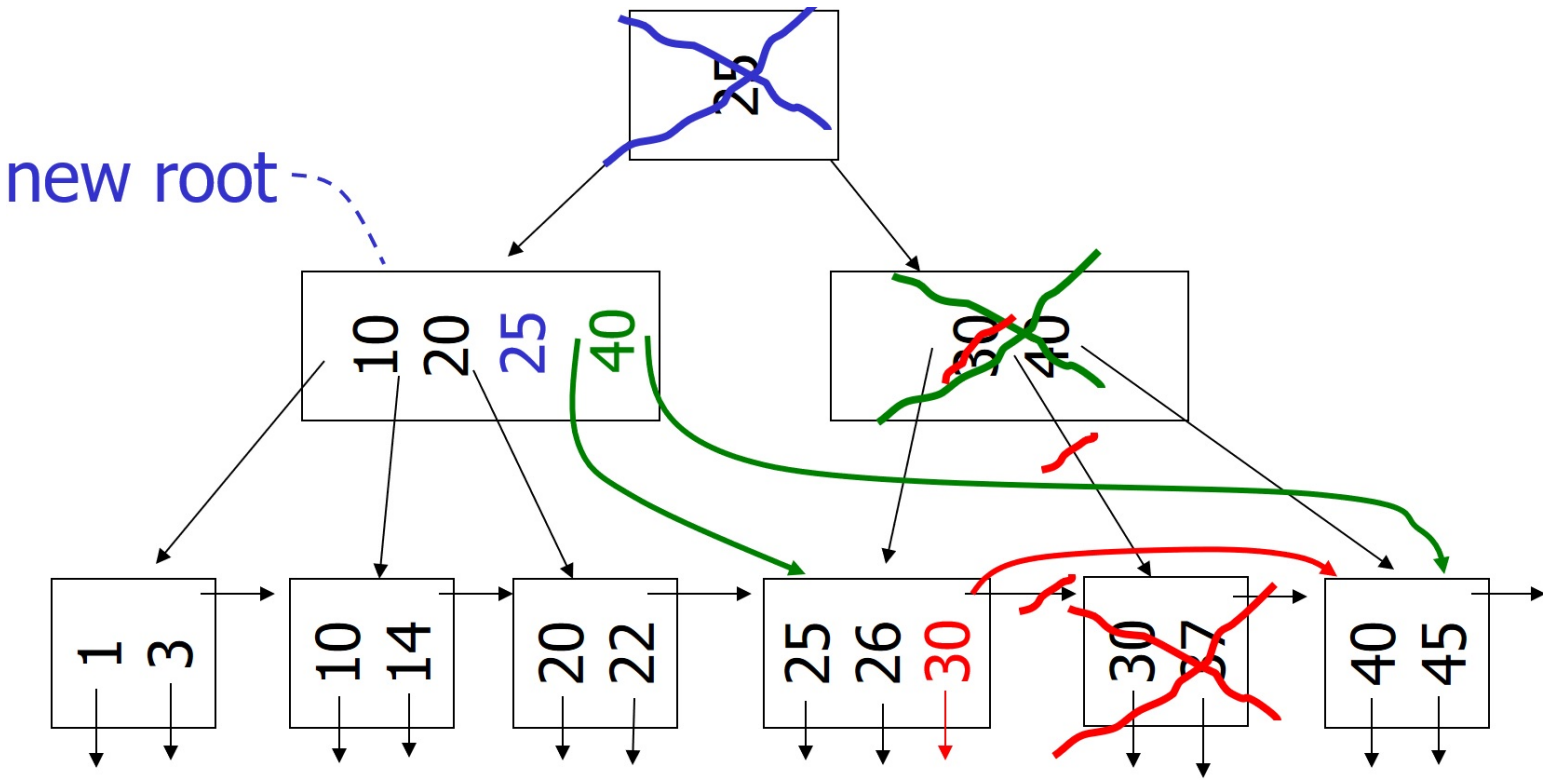
d) Non-leaf Coalesce: Delete 37, $n = 4$



d) Non-leaf Coalesce: Delete 37, $n = 4$



d) Non-leaf Coalesce: Delete 37, $n = 4$



Deletion Algorithm

- Delete record with key k
- Search leaf node for k
  - Leaf has more than min entries
    - Remove from leaf
  - Leaf has min entries
    - Try to borrow from sibling
  - One direct sibling has more min entries
    - Move entry from sibling and adapt key in parent

Deletion Algorithm (cont.)

- Both direct siblings have min entries
  - Merge with one sibling
  - Remove node or sibling from parent
  - -¿recursive deletion
- Root has two children that get merged
  - Merged node becomes new root

B^+tree deletions in practice

- Often, coalescing is not implemented
  - Too hard and not worth it!
  - Assumption: nodes will fill up in time again

Splitting or Merging nodes

- When splitting or merging nodes follow these conventions:
  - **Leaf Split**: In case a leaf node needs to be split during insertion and n is even, the left node should get the extra key. E.g, if $n = 2$ and we insert a key 4 into a node [1,5], then the resulting nodes should be [1,4] and [5]. For odd values of n we can always evenly split the keys between the two nodes. In both cases the value inserted into the parent is the smallest value of the right node.
  - **Non-Leaf Split**: In case a non-leaf node needs to be split and n is odd, we cannot split the node evenly (one of the new nodes will have one more key). In this case the "middle" value inserted into the parent should be taken from the right node. E.g., if $n = 3$ and we have to split a non-leaf node [1,3,4,5], the resulting nodes would be [1,3] and [5]. The value inserted into the parent would be 4
  - **Node Underflow**: In case of a node underflow you should first try to redistribute values from a sibling and only if this fails merge the node with one of its siblings. Both approaches should prefer the left sibling. E.g., if we can borrow values from both the left and right sibling, you should borrow from the left one.

Comparison: B^+tree vs. static indexed sequential file

- Ref #1: Held & Stonebraker, "B-Trees Re-examined", CACM, Feb. 1978
- Ref #2: M. Stonebraker, "Retrospection on a database system", TODS, June 1980

Comparison: B^+tree vs. static indexed sequential file

B^+tree

- Consumes more space, so lookup slower
- Each insert/delete potentially restructures
- Build-in restructuring
- Predictable performance

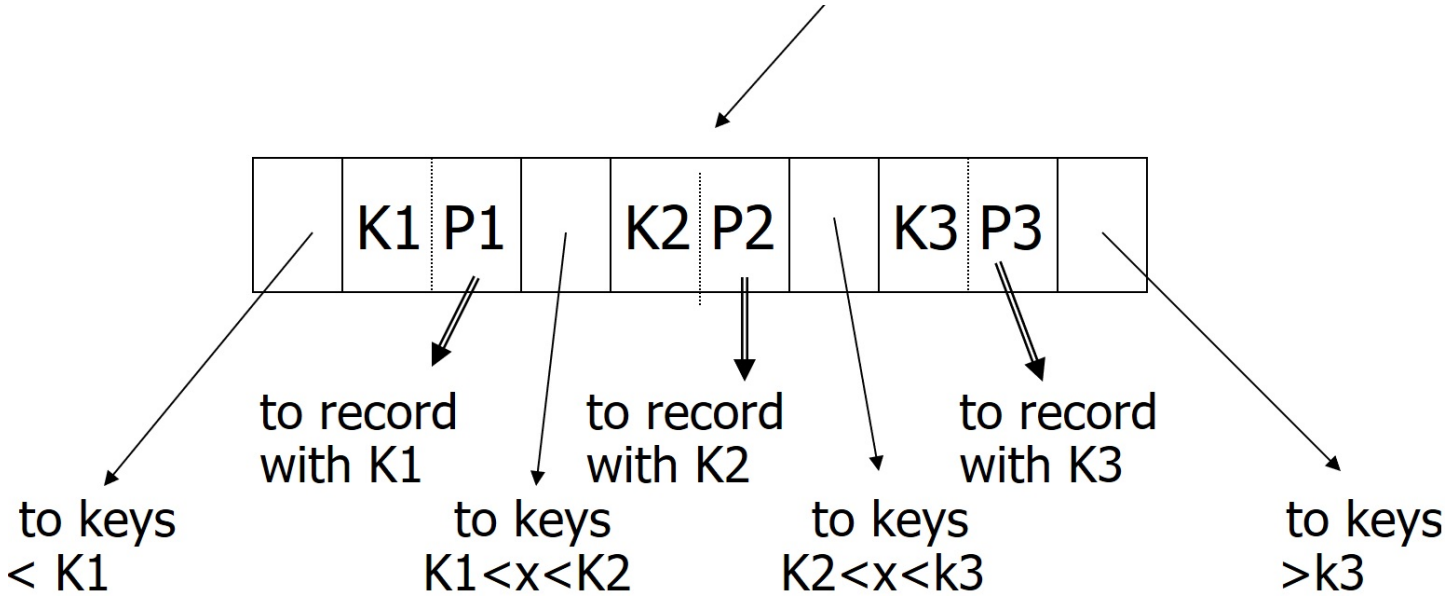
indexed seq. file

- Less space, so lookup faster
- Inserts managed by overflow area
- Requires temporary restructuring
- Unpredictable performance

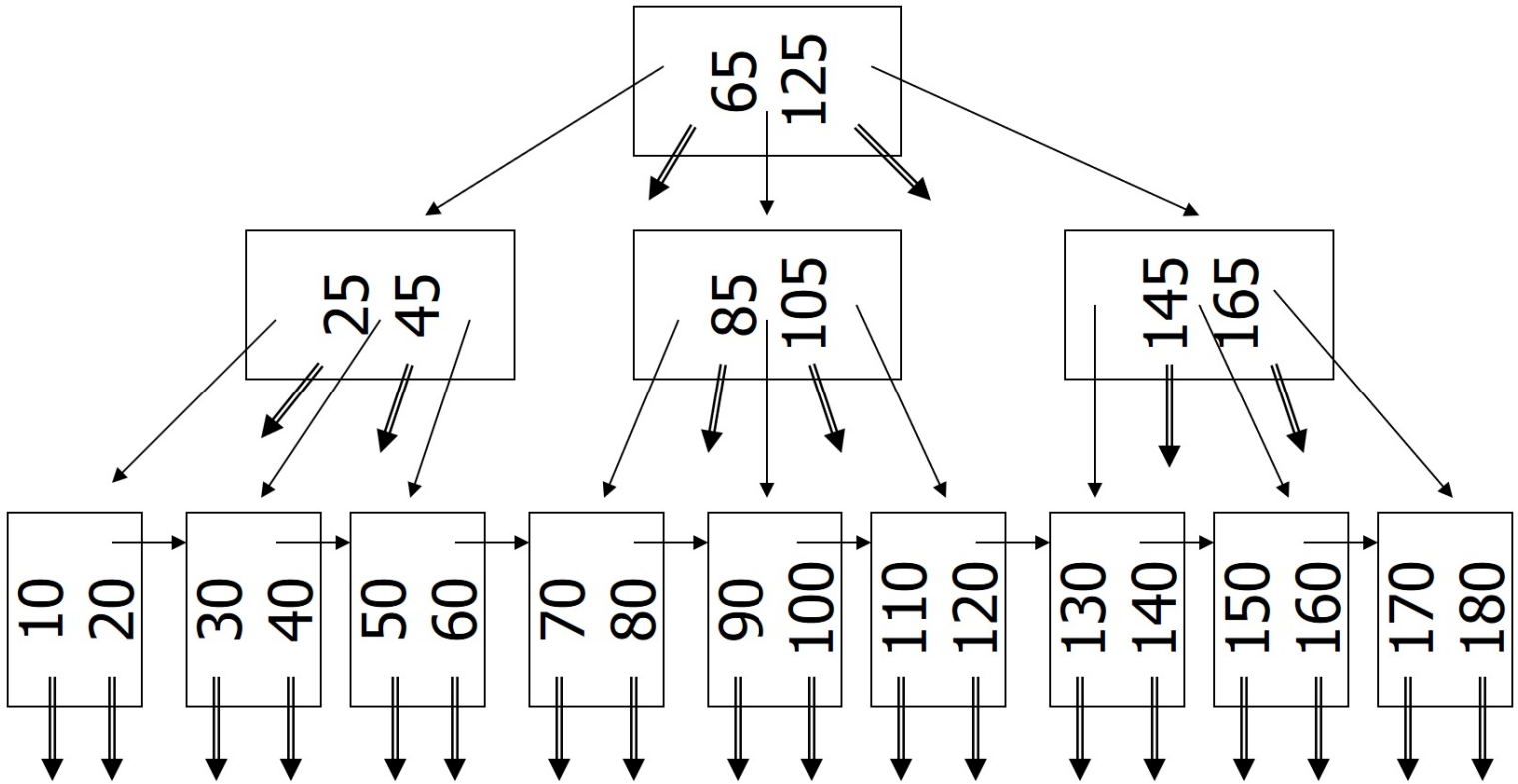
Variation on B^+tree: B-tree (no +)

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

Variation on B^+tree: B-tree (no +)



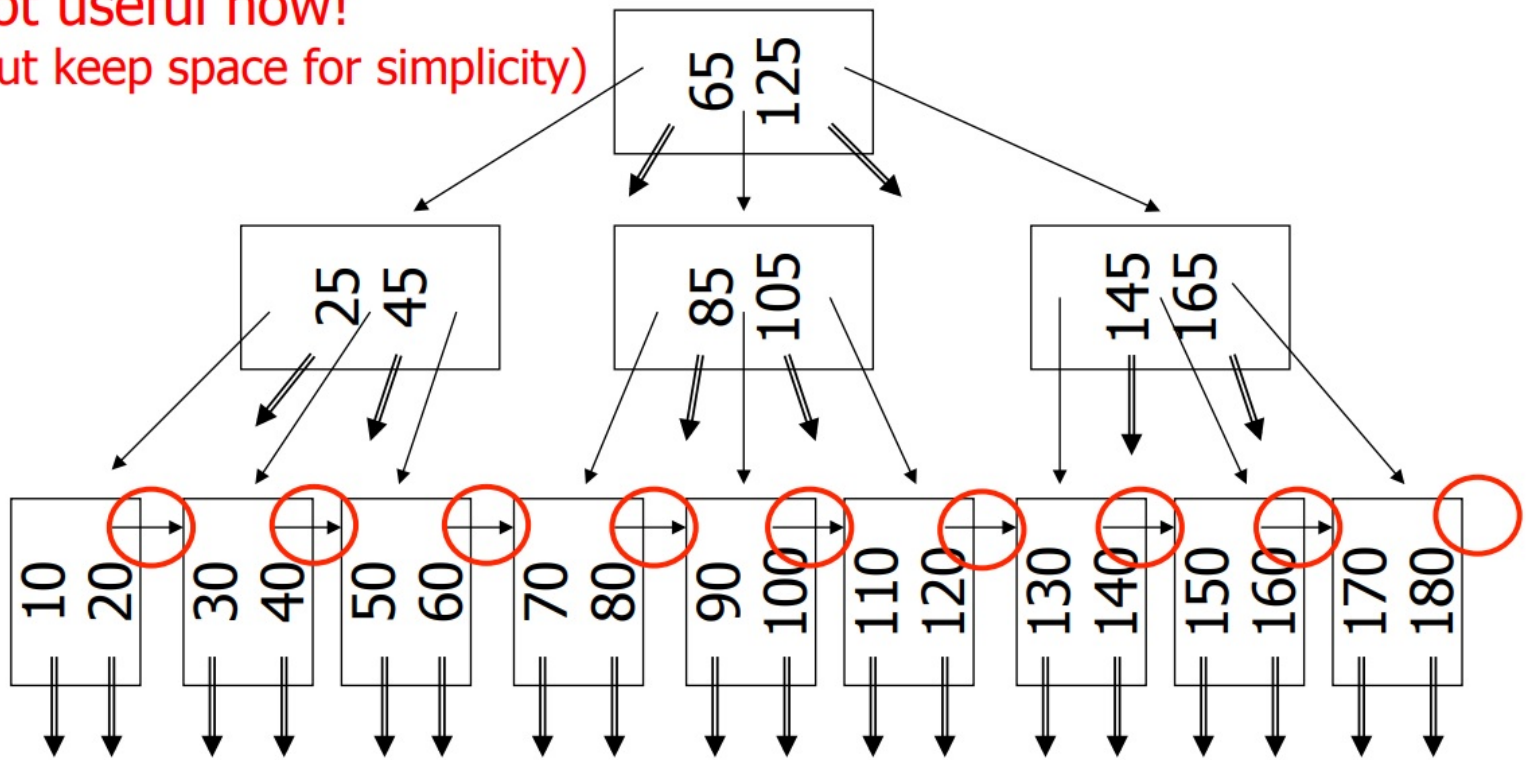
B-tree example: $n = 2$



- The leaf nodes in a B-tree is not linked
- i.e., the last record pointer in a leaf node is not used

B-tree example: $n = 2$

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



B-tree example: $n = 2$

- B-trees have faster lookup than B^+trees
- in B-tree, non-leaf & leaf different sizes
- in B-tree, deletion more complicated

  $\Rightarrow$ B^+trees preferred

But, note:

- If `blocks` are fixed size (due to disk and buffering restrictions)
- Then `lookup` for $\texttt{B}^{+}\texttt{tree}$ is **actually better**

Example

- Consider a DBMS that has the following characteristics:
  - Pointers 4 bytes
  - Keys 4 bytes
  - Blocks 100 bytes
- Compute the maximum number of records we can index with:
  a) 2-level B-tree
  b) 2-level B^+tree

B-tree

- Find largest integer value of n such that $4n + 4(2n + 1) < 100$

  $n = 8$
- `Root` has 8 keys + 8 record pointers +9 children pointers

  $= 8 \times 4 + 8 \times 4 + 9 \times 4 = 100 bytes$
- Each of 9 children: 12 records pointers (+12 keys)

  $= 12 \times (4 + 4) + 4 = 100$ bytes
- 2-level `B-tree`, maximum # of records $= 12 \times 9 + 8 = 116$

B^+tree

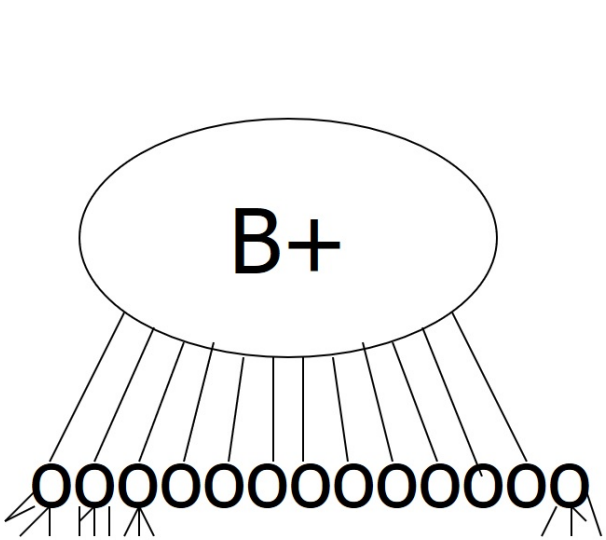
- Find largest integer value of n such that $4n + 4(n + 1) < 100$

  $n = 12$
- `Root` has 12 keys + 13 children pointers

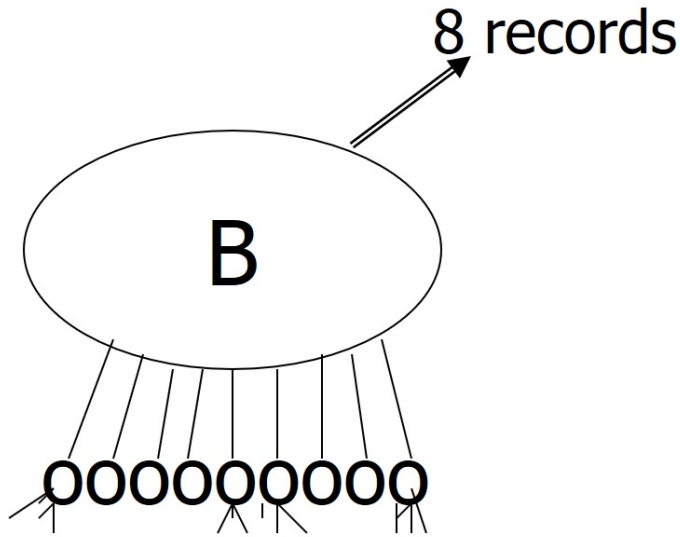
  $= 12 \times 4 + 13 \times 4 = 100 bytes$
- Each of 13 children: 12 records pointers (+12 keys)

  $= 12 \times (4 + 4) + 4 = 100$ bytes
- 2-level B^+`tree`, maximum # of records $= 13 \times 12 = 156$

So,



156 records



108 records

Total = 116

Reading

- Chapter 4: Index Structures (uploaded on course Blackboard)

Next

- Hashing schemes