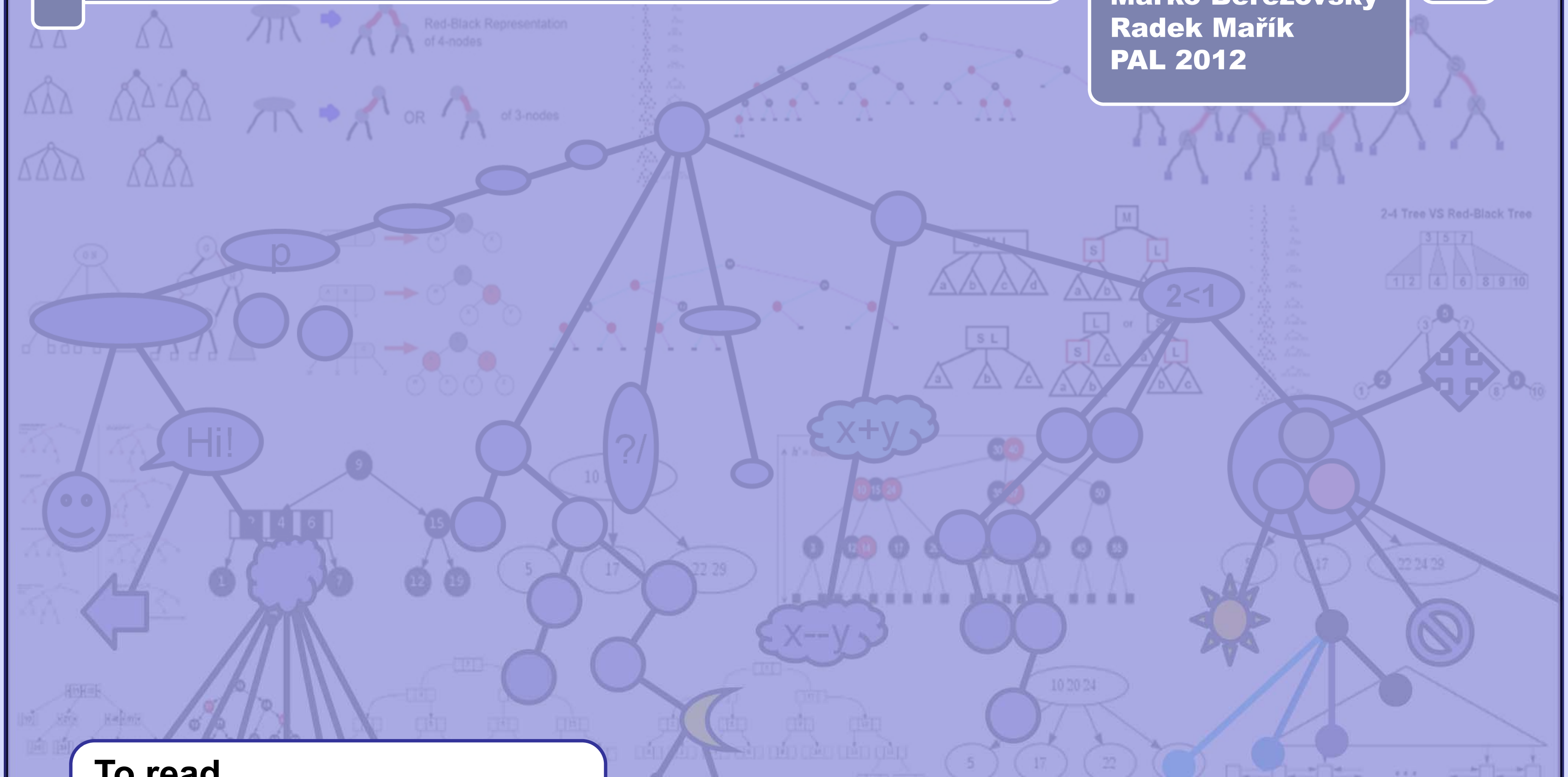
# Search trees, 2-3-4 tree, B+tree

**Marko Berezovský**
**Radek Mařík**
**PAL 2012**




## To read

- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998
- http://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf
- (CLRS) Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*, 3rd ed., MIT Press, 2009

See PAL webpage for references

A **2-3-4 search tree** is either empty or contains three types of nodes:

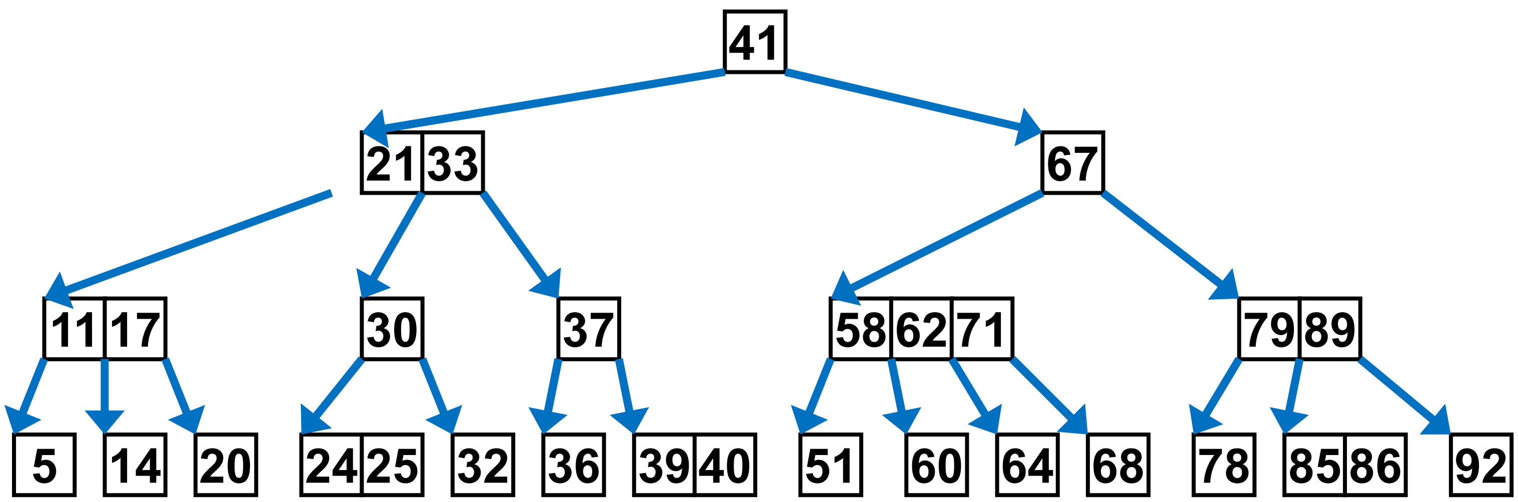
2-nodes, with one key, a left link to a tree with smaller keys, and a right link to a tree with larger keys;

3-nodes, with two keys, a left link to a tree with smaller keys, a middle link to a tree with key values between the node's keys and a right link to a tree with larger keys;

4-nodes, with three keys and four links to trees with key values defined by the ranges subtended by the node's keys.

AND: All links to empty trees, ie. all leaves, are at the same distance from the root, thus the tree is perfectly balanced.

A **2-3-4 search tree** is structurally a **B-tree of order 4**.




Note 2-nodes, 3-nodes, 4-node, same depth of all leaves.

**Find**: As in B-tree

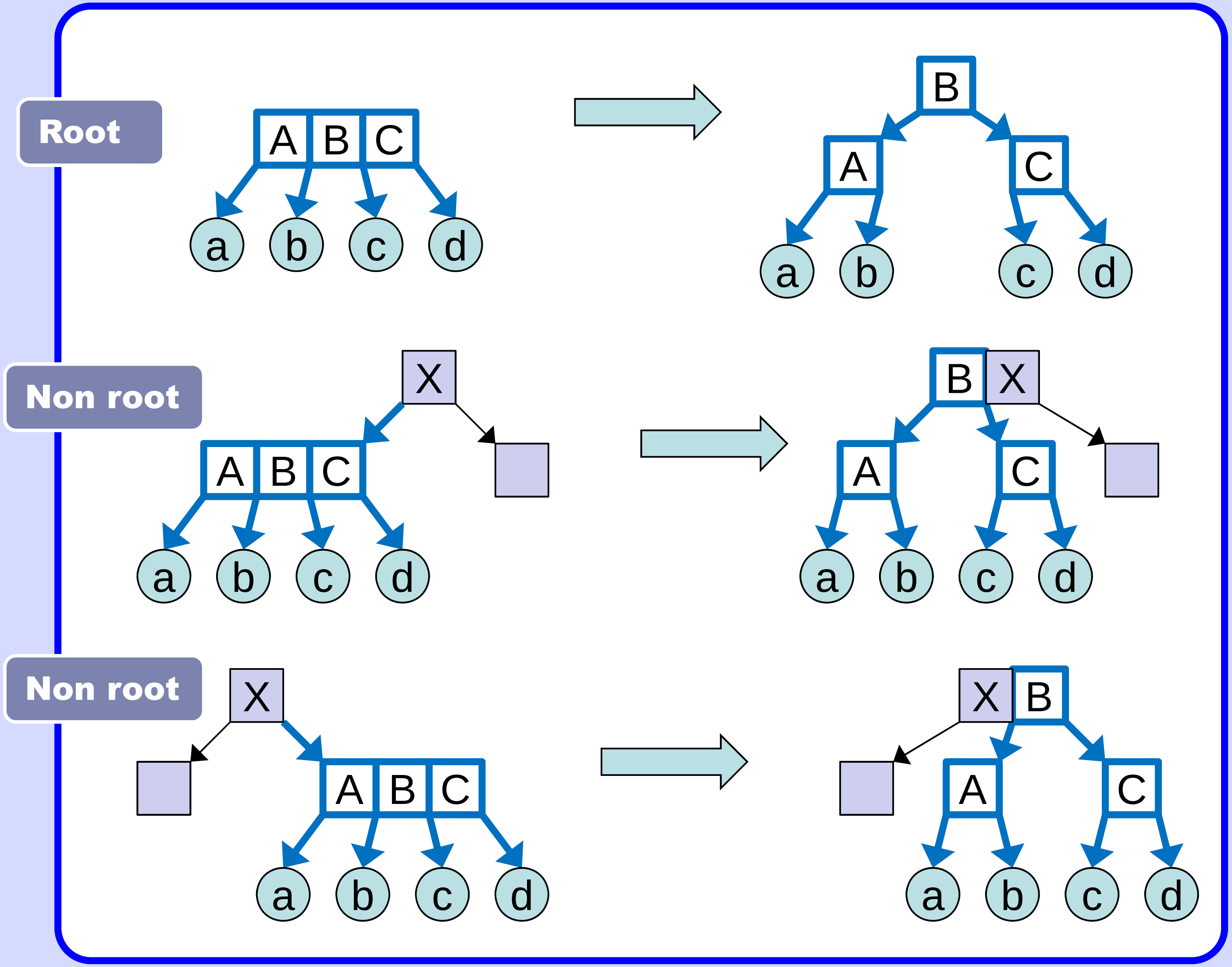
**Insert**: As in B-tree: Find the place for the inserted key x in a leaf and store it there. If necessary split the leaf.

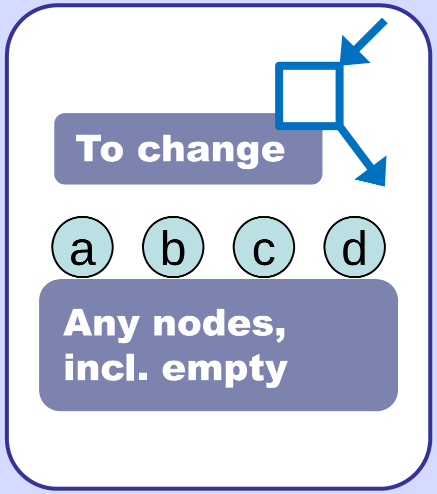
Aditional rule:
In our way down the tree, whenever we reach a ***4-node***, we split it into two ***2- nodes***, and move the middle element up
This strategy prevents the following from happening:
After inserting a key it might happen in B tree that it is necessary to split all the nodes going from inserted key back to the root. This is consider to be time consuming.

Splitting 4-nodes on the way down results in sparse occurence of 4-nodes in the tree, thus it never happens that we have to split nodes recursively bottom-up.

**Delete**: As in B-tree

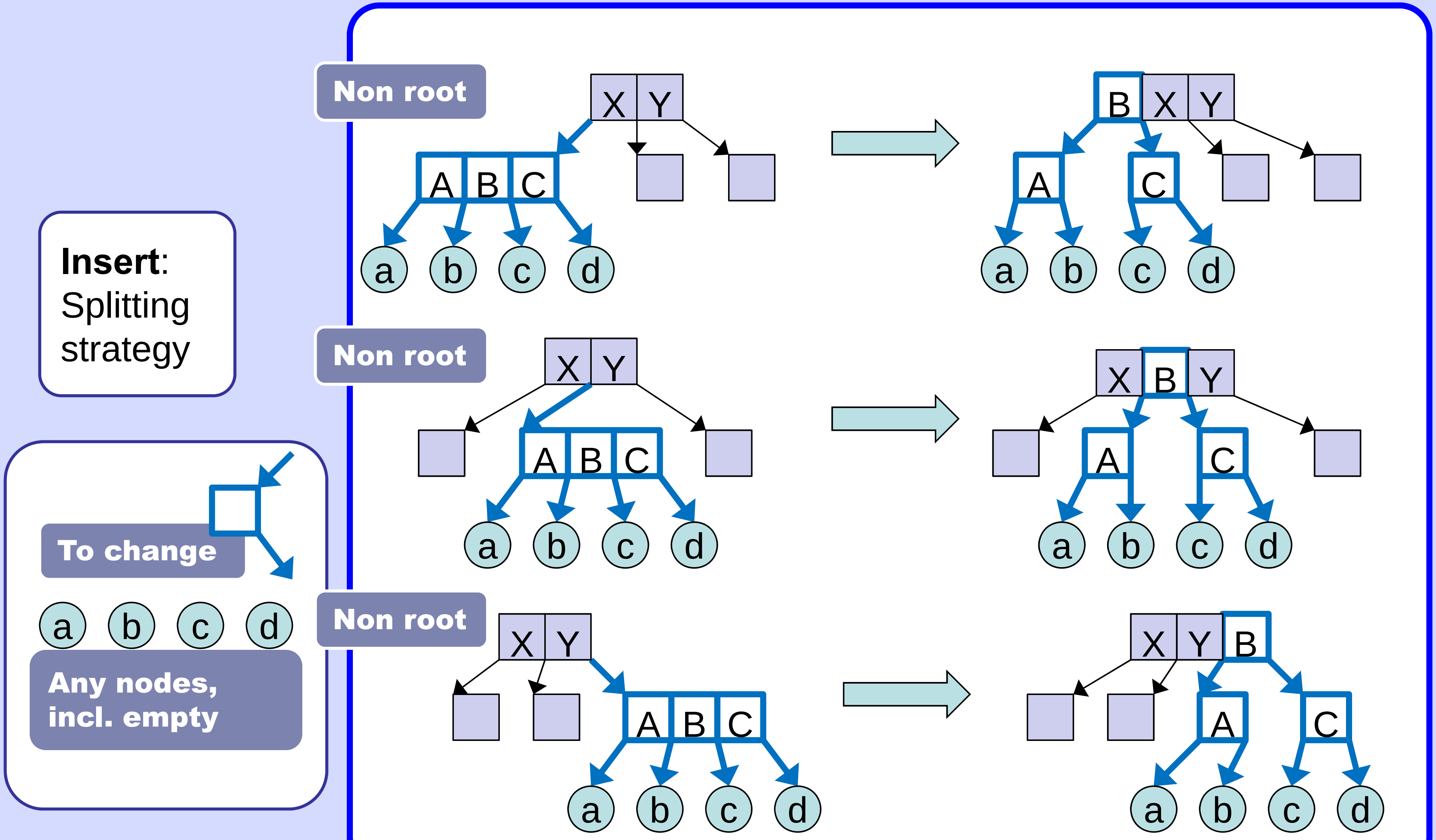
**Insert**:
Splitting
strategy



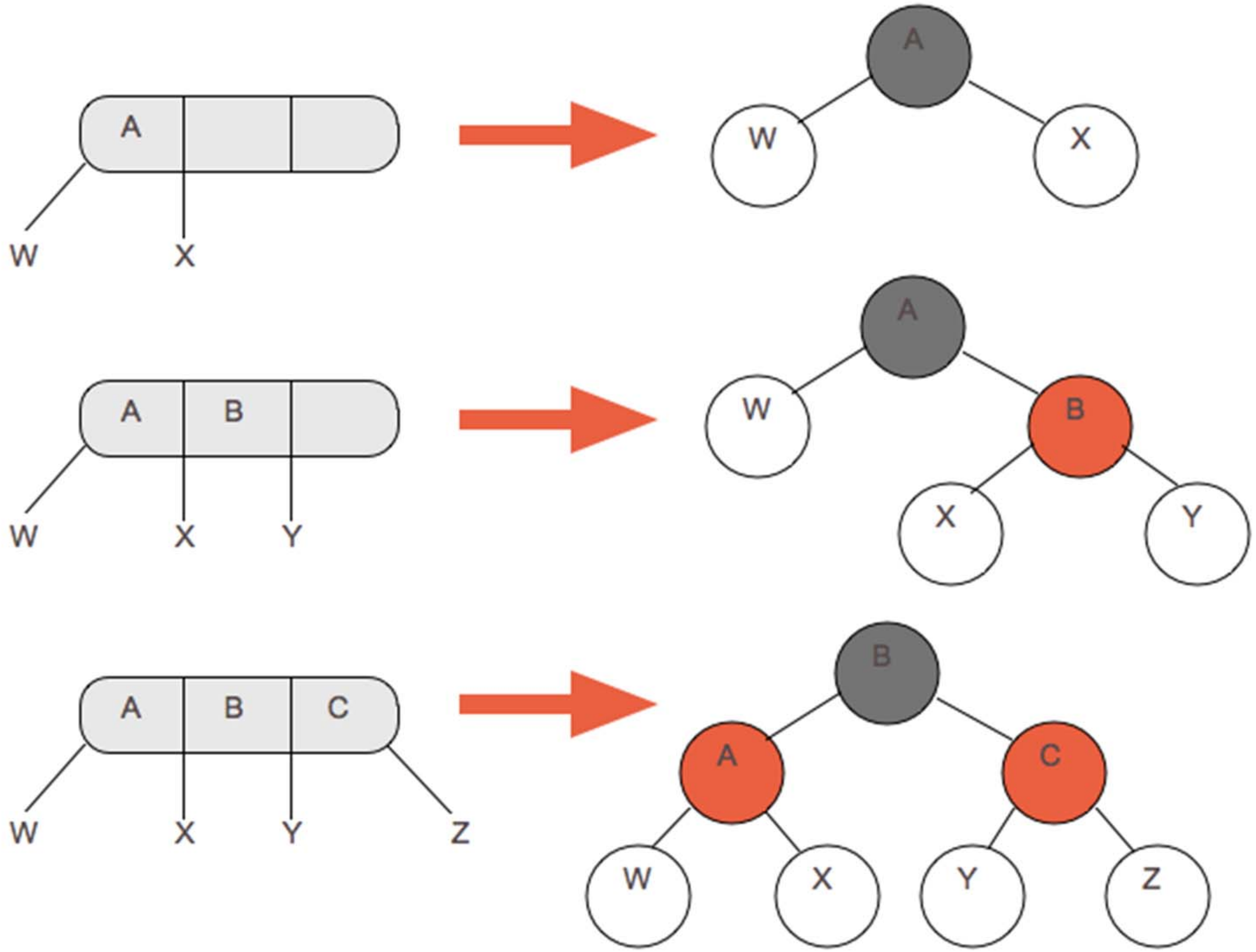

Note that splitting changes the height of a tree only when the root is splitted.

**Insert**: Splitting strategy

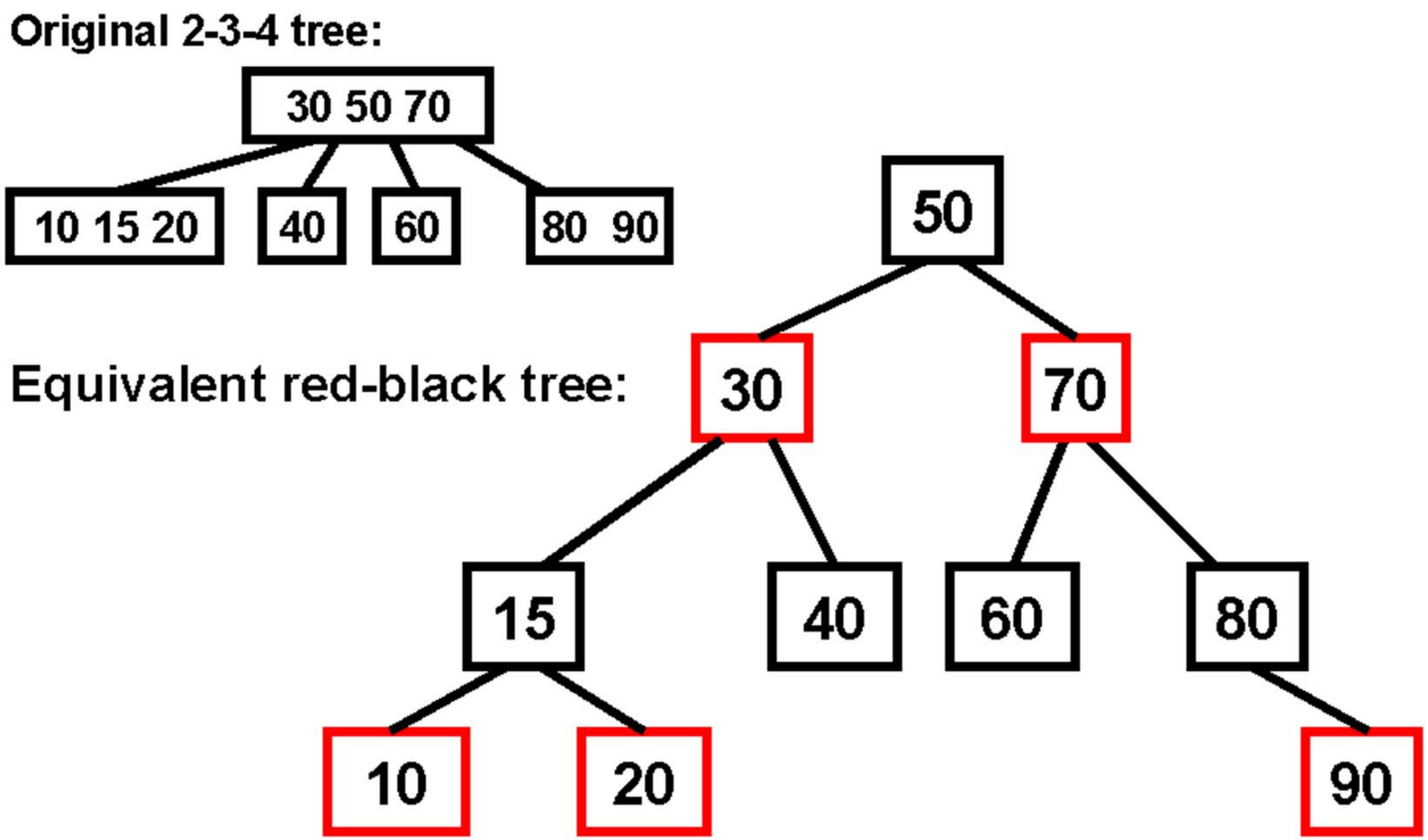



Note that splitting changes the height of a tree only when the root is splitted.

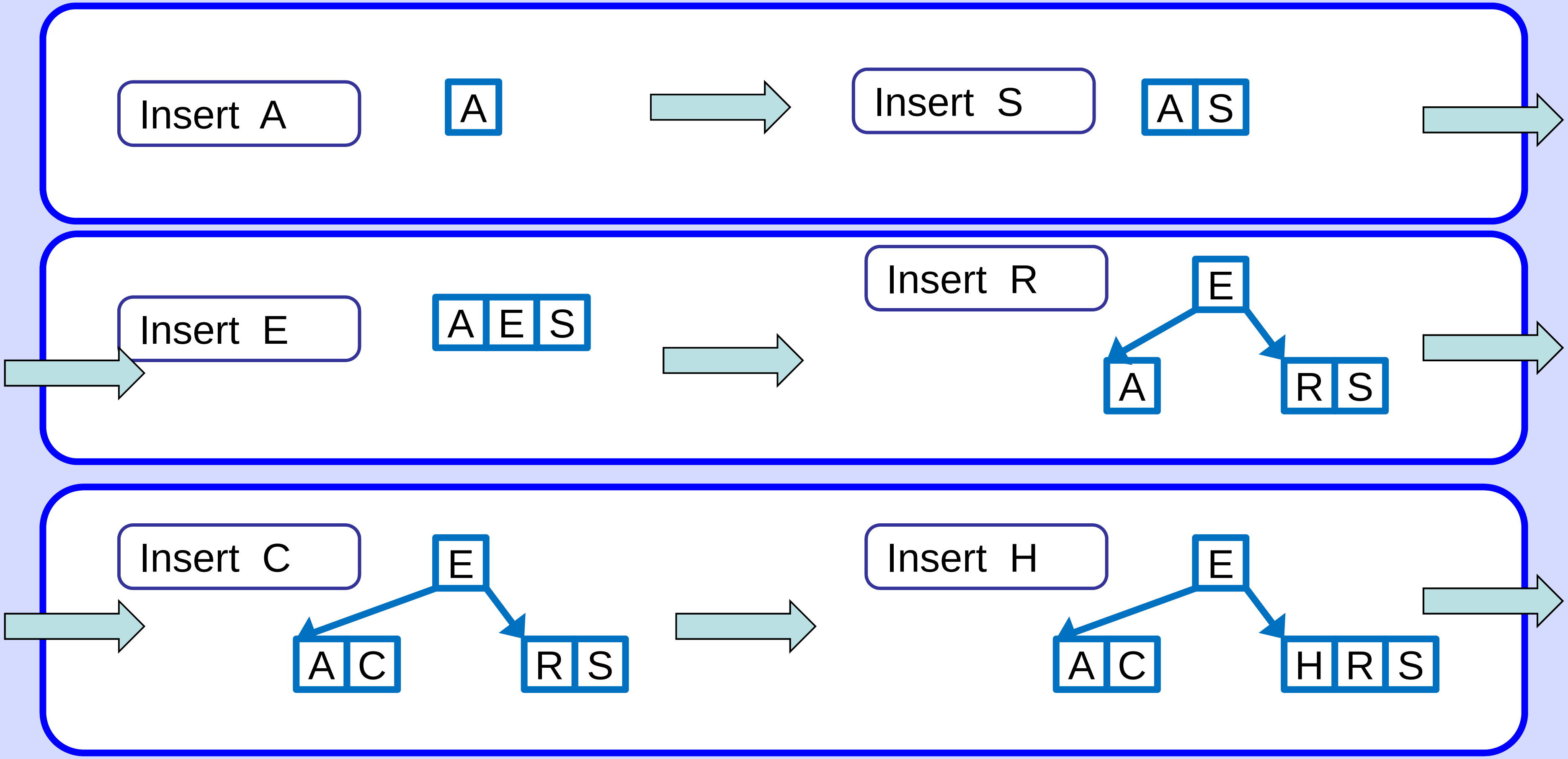
## Relation of 2-3-4 tree to Red-Black tree

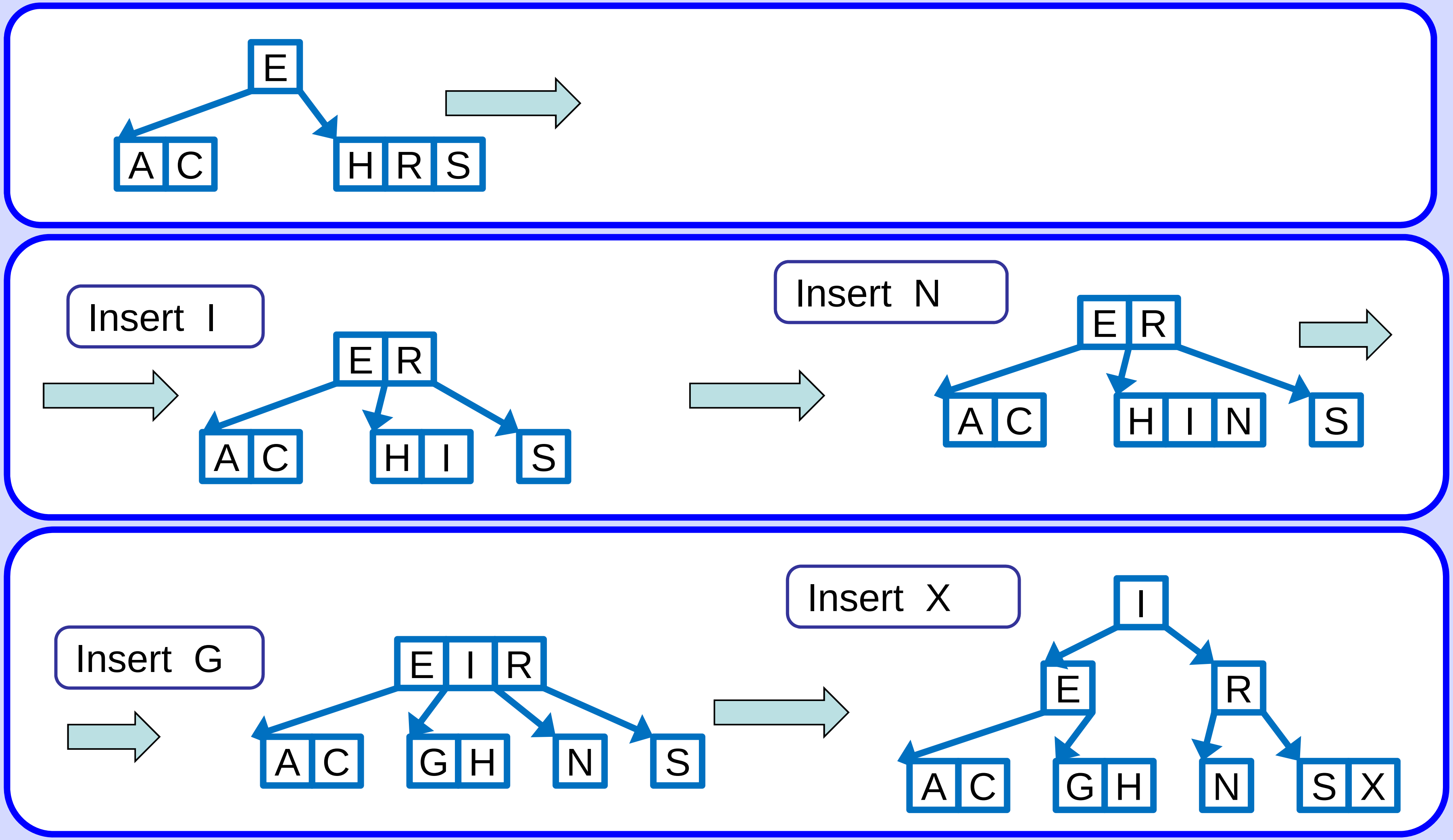


## Relation of 2-3-4 tree to Red-Black tree



Insert keys into initially empty 2-3-4 tree: A S E R C H I N G X






Note seemingly unnecessary split of EIR 4-node during insert of G.

Complexities

Searches in N-node 2-3-4 trees visit at most lg N + 1nodes

Insertions into N-node 2-3-4 trees require fewer than lg N + 1 node splits in the worst case, and seem to require less than one node split on the average

Precise analytic results on the average-case performance of 2-3-4 trees have so far eluded the experts**, but it is clear from empirical studies that very few splits are used to balance the trees. The worst case is only lg N, and that is not approached in practical situations.

** Now, finally there is an appropriate challenge for you!

Results of an example experiment with N uniformly distributed random keys from renge {1, ..., $10^9$ } being inserted into initially empty 2-3-4 tree:

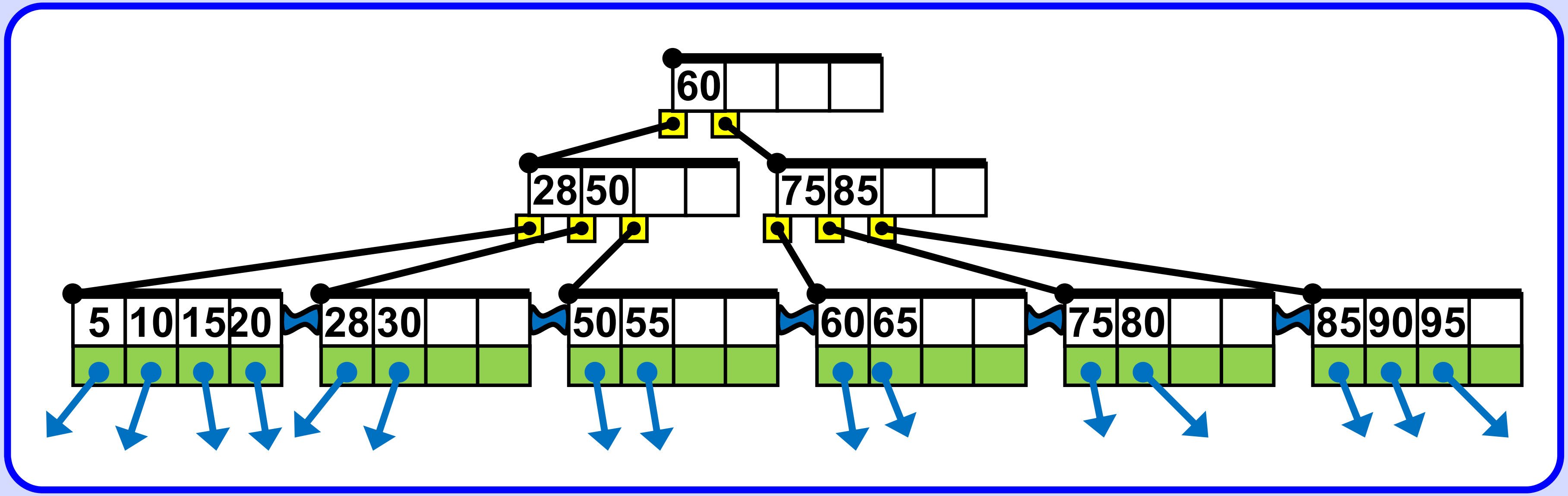
| N | Tree depth | 2-nodes | 3-nodes | 4-nodes |
|---:|---:|---:|---:|---:|
| 10 | 2 | 6 | 2 | 0 |
| 100 | 4 | 39 | 29 | 1 |
| 1000 | 7 | 414 | 257 | 24 |
| 10 000 | 10 | 4 451 | 2 425 | 233 |
| 100 000 | 13 | 43 583 | 24 871 | 2 225 |
| 1 000 000 | 15 | 434 671 | 248 757 | 22 605 |
| 10 000 000 | 18 | 4 356 849 | 2 485 094 | 224 321 |

B+ tree

B+ tree is analogous to the B-tree, namely in
being perfectly balanced all the time
nodes cannot be less than half full
operational complexity.

The differences are
Records (or pointers to actual records) are stored only at the leaf nodes.
Internal nodes store only search key values, and are used only as placeholders to guide the search.

The leaf nodes of a $B^+$-tree are linked together to form a linked list. This is done so that the records can be retrieved sequentially without accessing the $B^+$-tree index. This also supports fast processing of range-search queries.




Routers or keys 75

Data records
or pointers to them

Values in internal nodes are routers, originally each of them was a key when a record was inserted later they migrated in the tree and may stay there even after the record and its key was deleted. Insert and Delete operations by splitting and Joining the nodes move routers around.

Values in the leaves are actual keys associated with the records and must be deleted when a record is deleted (their router copies may live on).

Inserting key K (and its associated record ) into B+ tree

Find, as in B tree, correct leaf to insert K,

**Case 1**

Free slot in a leaf? YES

Place the key and its associated record in the leaf

**Case 2**

Free slot in a leaf? NO. Free slot in parent node? YES.

1. Split the leaf, consider all its keys including K sorted.
2. insert middle (median) key M in the parent node in an appropriate slot Y.
3. Left leaf from Y contains records with keys smaller than M.
4. Right leaf from Y contains records with keys equal to or greater than M.

Note: Splitting leaves / inner nodes works same way as in B-trees.

Inserting key K (and its associated record ) into B+ tree

Find, as in B tree, correct leaf to insert K,

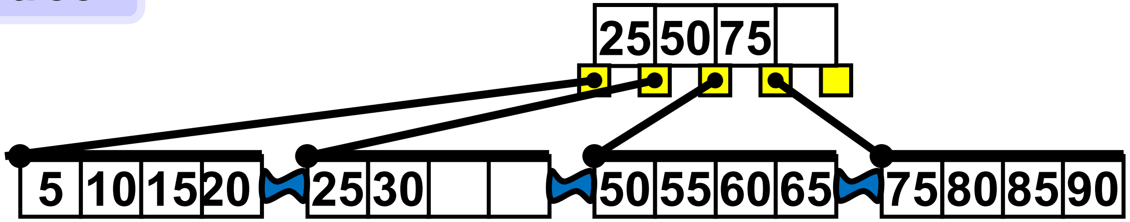
**Case 3**

Free slot in a leaf?  NO. Free slot in parent node?  NO.

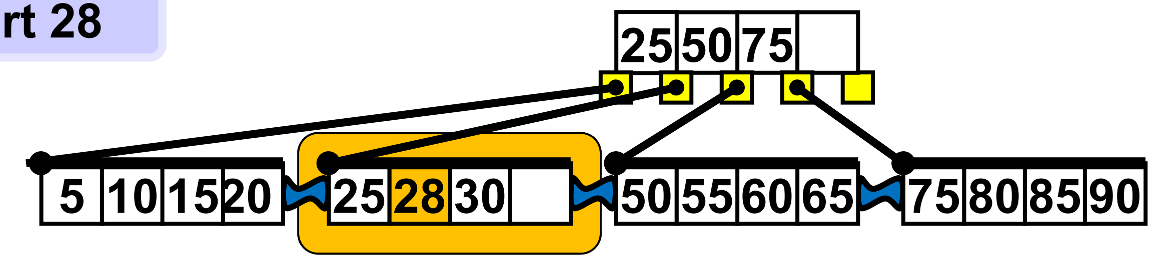
1. Split the leaf, consider all its keys including K sorted, denote M median of these keys
2. Records with keys < M go to the left leaf.
3. Records with keys >= M go to the right leaf.

4. Split the parent node P to nodes P1 and P2, consider all its keys including M sorted, denote M1 median of these keys.
5. Keys < M1 key go to P1.
6. Keys > M1 key go to P2.
7. If parent PP of P  is not full, insert M1 to PP and stop.
   Else  set M := M1, P := PP and continue splitting  parent nodes recursively up the tree, repeating from step 4.

Initial tree




Insert 28

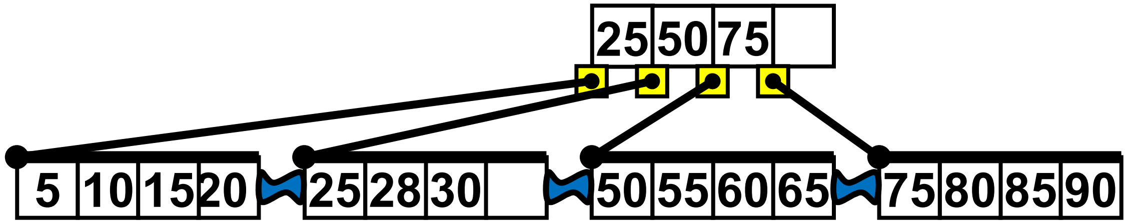



Changes

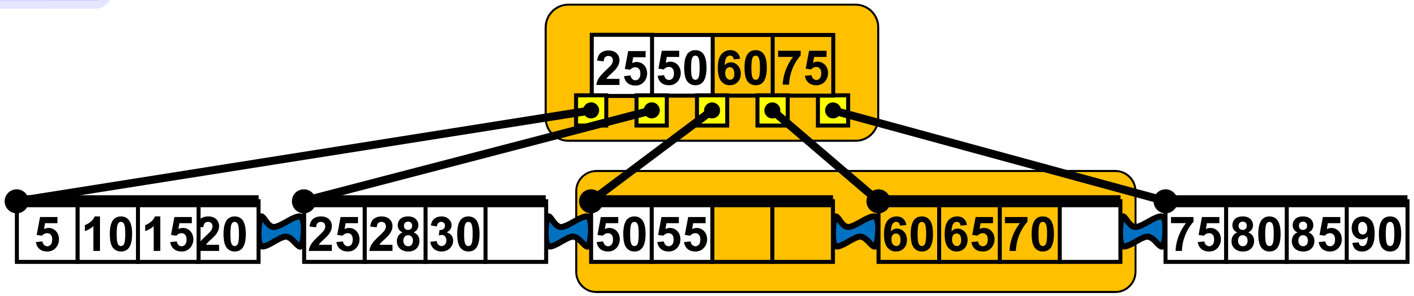
Leaf links

Records or records pointers are not drawn here for sake of simplicity.

Initial tree



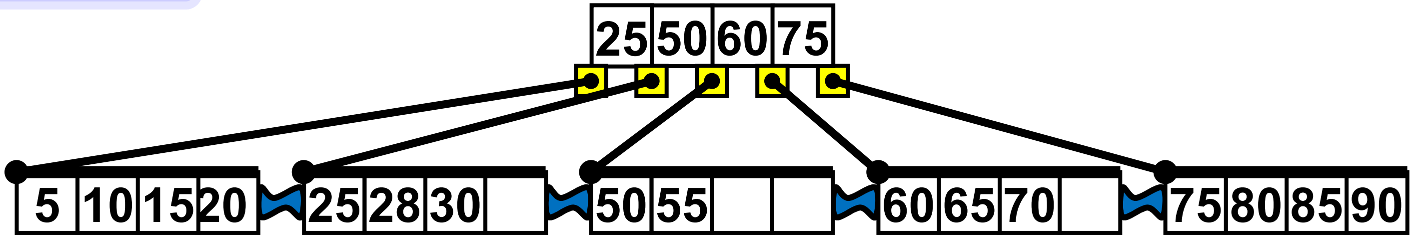

Insert 70



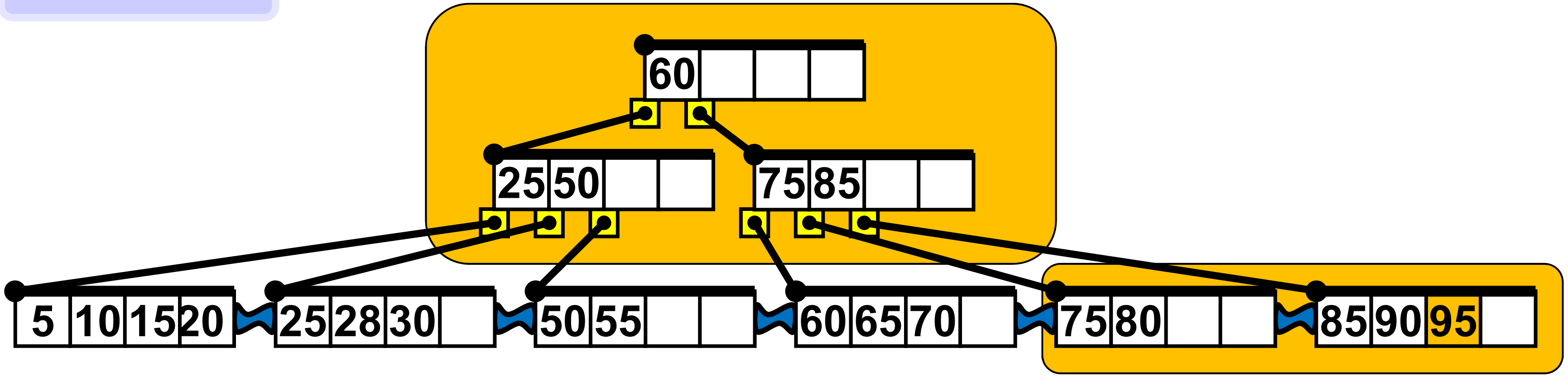

Changes

Leaf links

### Initial tree




### Insert 95




Changes

Leaf links

Deleting key K (and its associated record ) into B+ tree

Find, as in B tree, key K in a leaf,

**Case 1**

Leaf more than half full?   YES.

Delete the key and its record from the leaf. Arrange keys in the leaf in ascending order to fill void. If the deleted key K appears in the parent node P too, replace it by next bigger key K1 from L (it must always exist) and leave K1 in L as well.

**Case 2**

Leaf more than half full?  NO.   Left or right sibling more than half fulll?  YES.

Move one (or more if you wish and rules permit)  key from sibling S to the leaf L, reflect the changes in the parent P of leaf and parent P2 of sibling S (if P2 != P).

Note: Joining leaves/inner nodes works same way as in B-trees.

Deleting key K (and its associated record ) into B+ tree

Find, as in B tree, key K in a leaf,

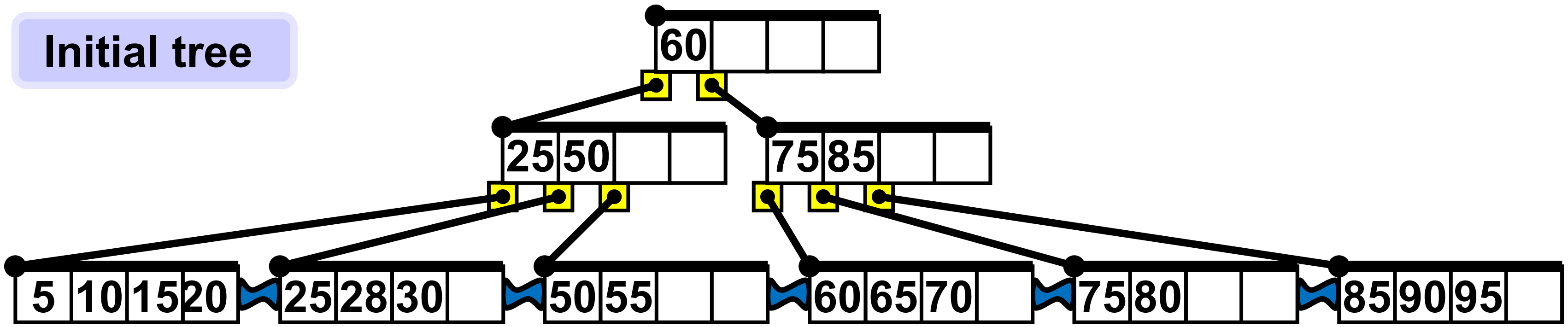
**Case 3**

Leaf more than half full? NO. Left or right sibling more than half fulll? NO.

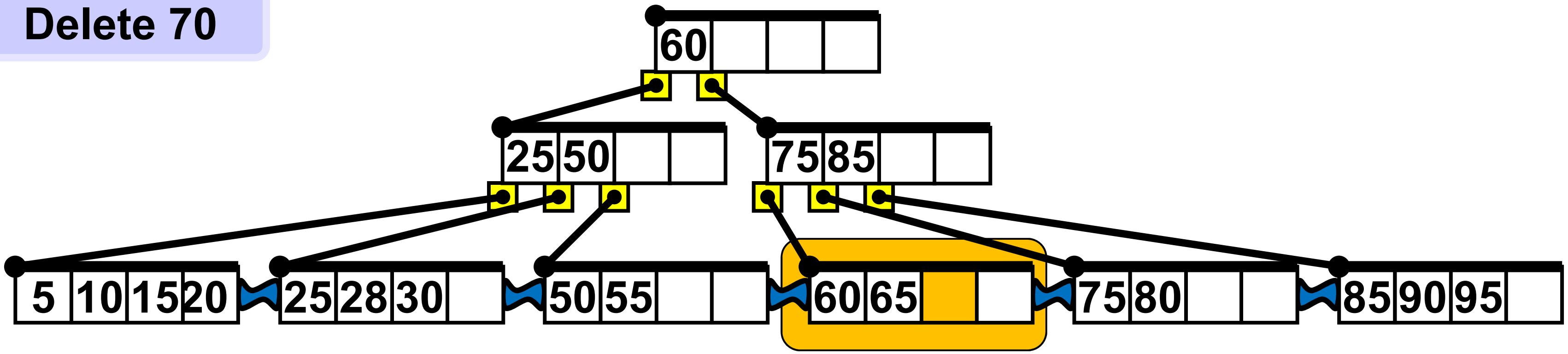
1. Consider sibling S of L which has same parent P as L.
2. Consider set M of ordered keys of L and S without K but together with key K1 in P which separates L and S.
3. Joining. Store M in L, connect L to the other sibling of S (if exists), destroy S.
4. The reference left to K1 points to L. Adjust P. If P contains K delete it from P. Delete K1 from S as well. If P is still at least half full stop, else continue with 5.
5. If any sibling SP of P is more then half full, move necessary number of keys from SP to P and adjust links in P, SP and their parents accordingly and stop. Else join P with sibling SP which parent PP is parent of P too and continue recursively as in B-tree up to the root if necessary.

Note: Joining leaves/inner nodes works same way as in B-trees.

Initial tree



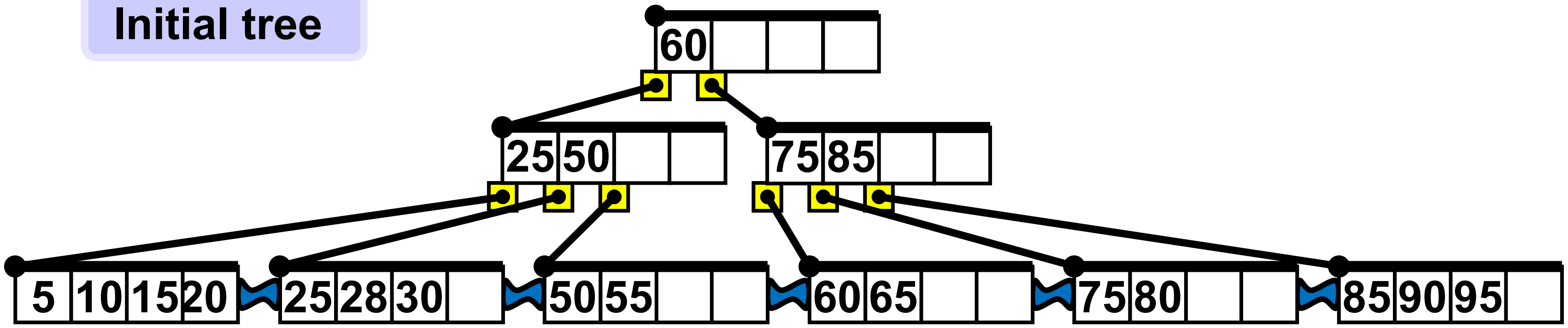

Delete 70



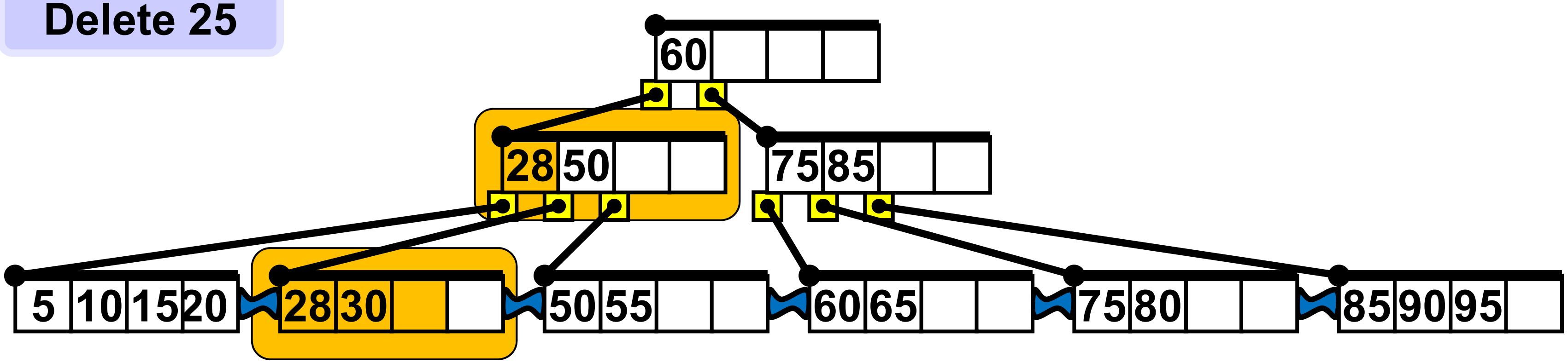

Changes

Leaf links

Initial tree




Delete 25

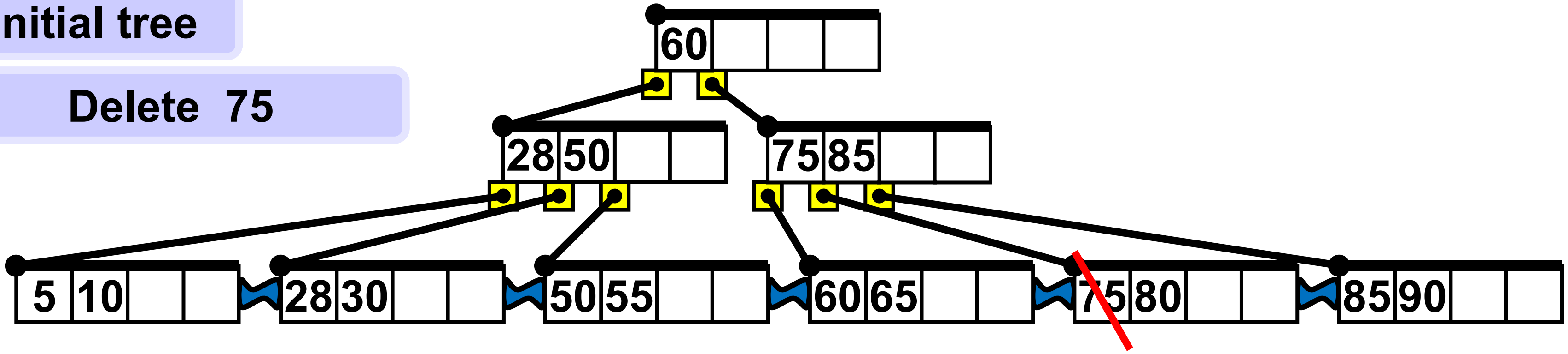



Changes

Leaf links

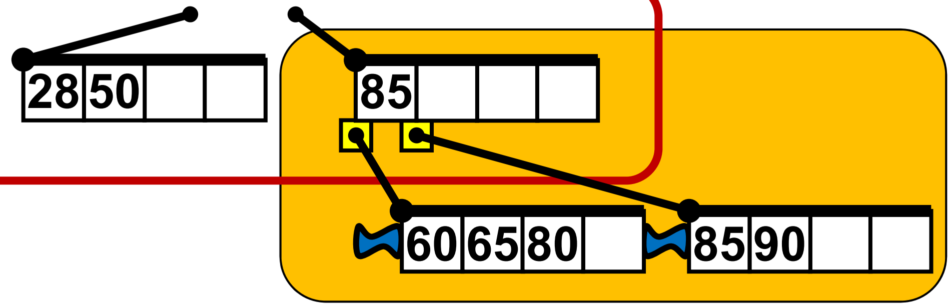
**Initial tree**

**Delete 75**

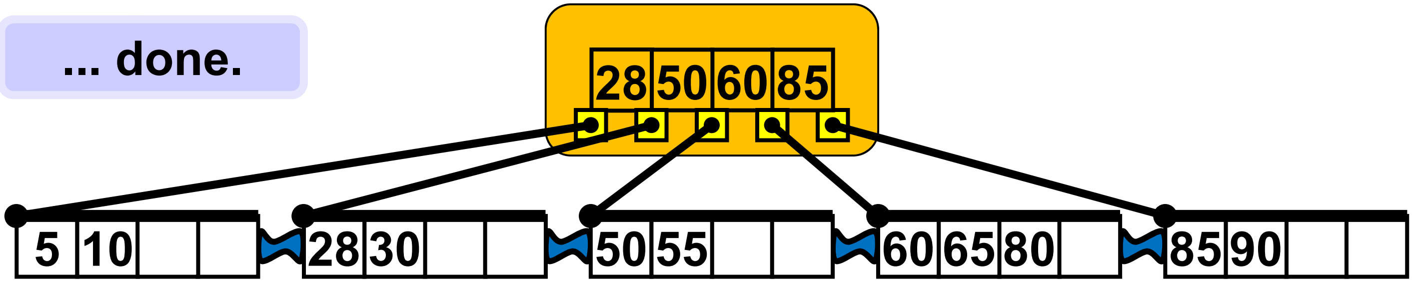


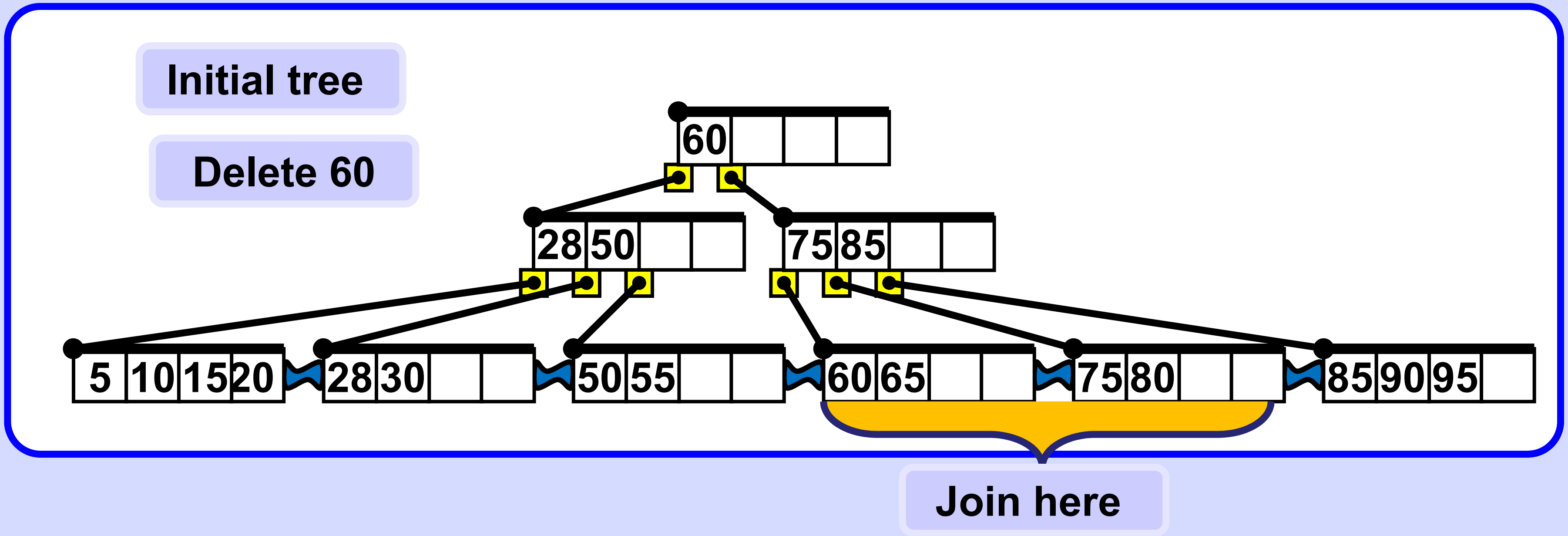

Too few keys, join these two nodes and bring key from parent (recursively)

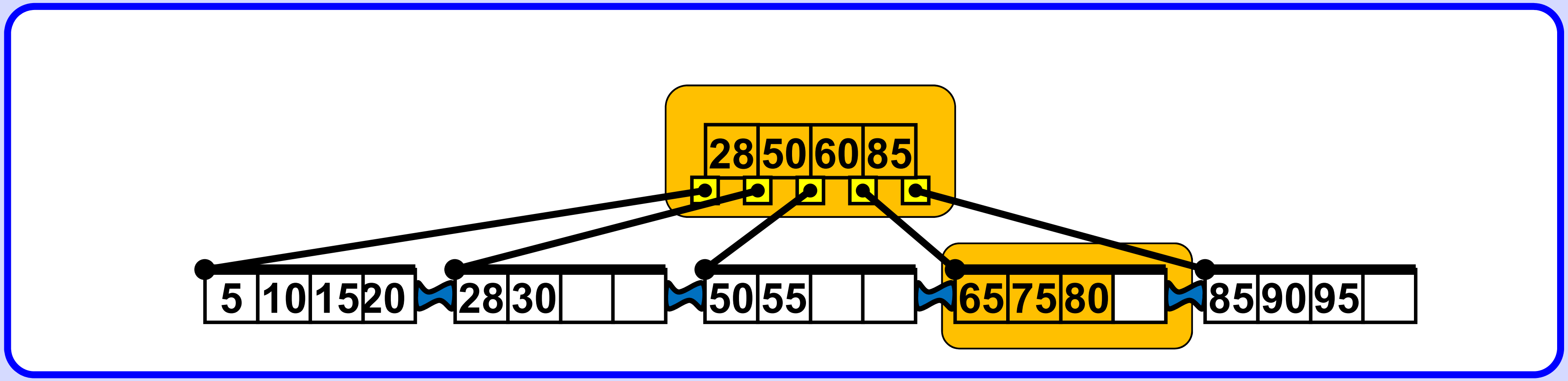
**Progress...**




**... done.**






Changes

Leaf links

Complexities

Find, Insert, Delete,
all need $\Theta(\log_b n)$ operations, where n is number of records in the tree,
and b is the branching factor or, as it is often understood, the order of the tree.

Note: Be careful, some authors (e.g CLRS) define degree/order of B-tree as [b/2], there is no unified precise common terminology.

Range search thanks to the linked leaves is performed in time
$\Theta(\log_b(n) + k)$
where k is the range (number of elements) of the query.