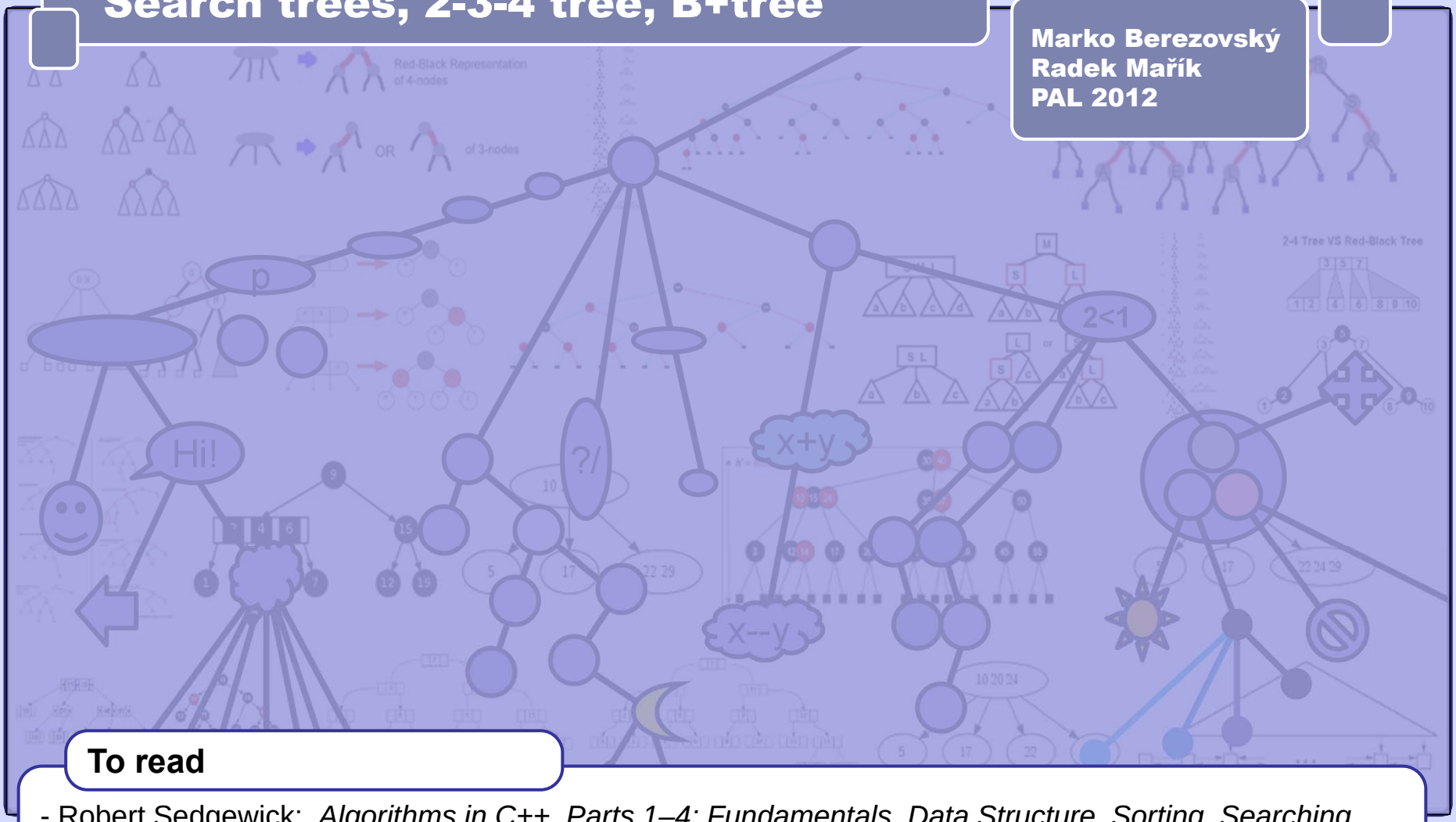
Search trees, 2-3-4 tree, B+tree

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 it contains three types of nodes:

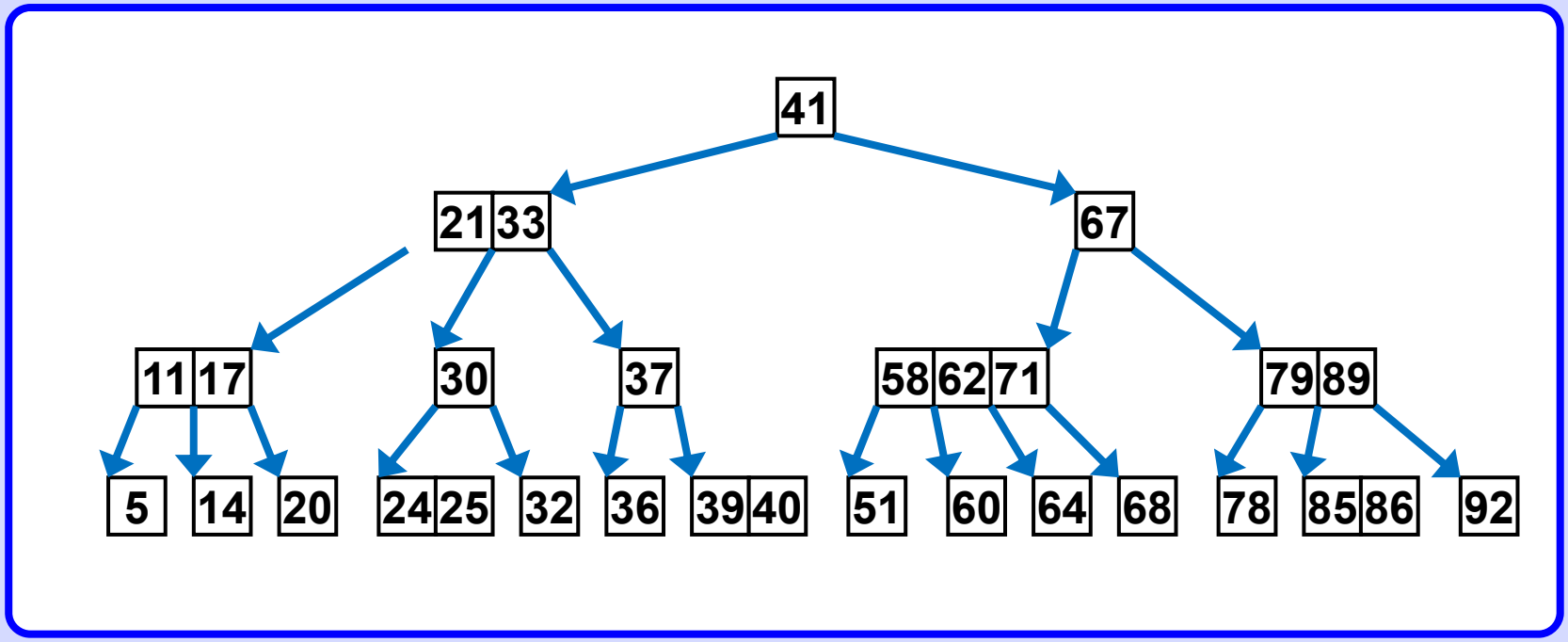
2-node, with one key, left link to a tree with smaller keys, and right link to a tree with larger keys;

3-node, 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-node, 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

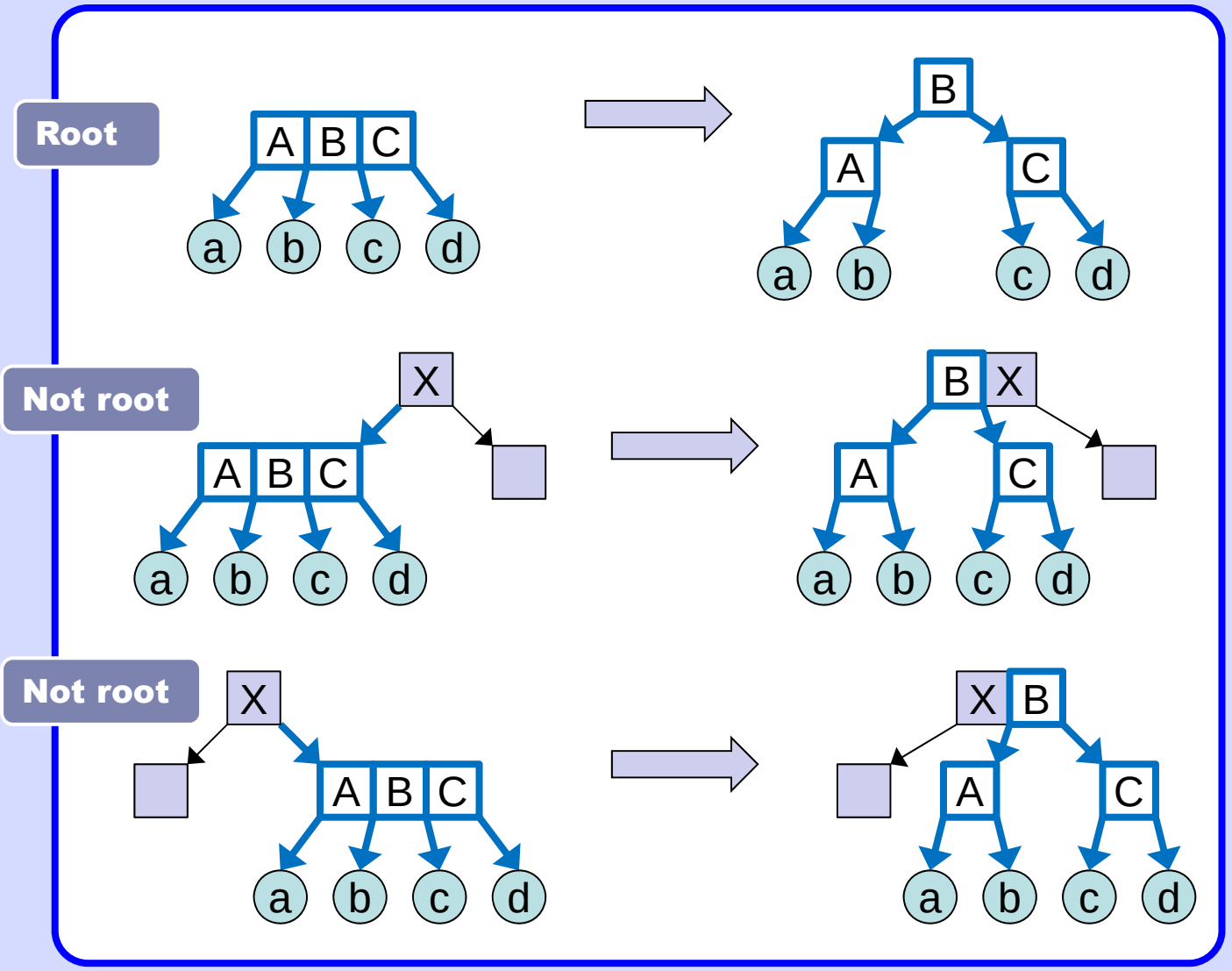
Insert: As in B-tree: Find the place for the inserted key x in a leaf and store it there. If necessary split the leaf.

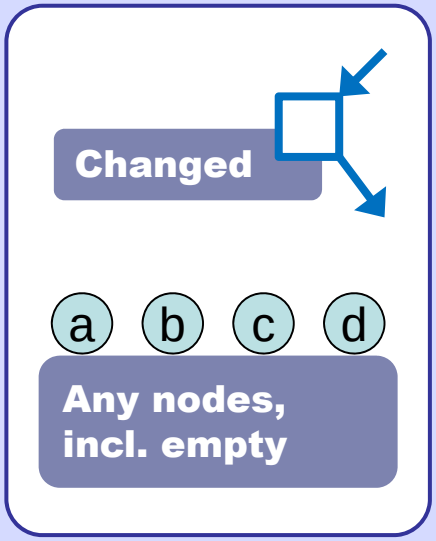
Additional **insert** rule:
In our way down the tree, whenever we reach a ***4-node***, we split it into two ***2- nodes***, and move the middle element up to the parent node.
This strategy prevents the following from happening:
After inserting a key it might happen in B tree that it is necessary to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

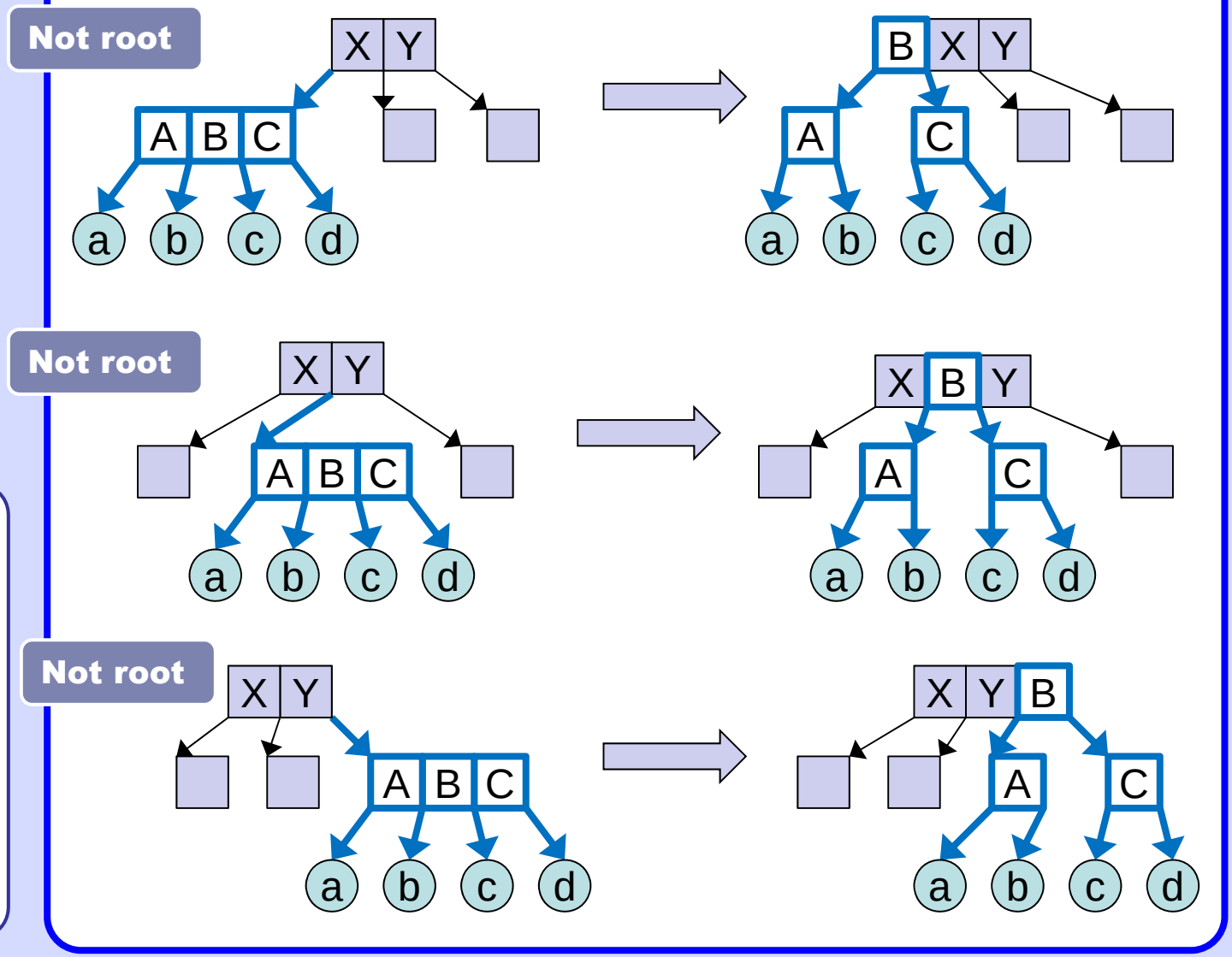
Insert: Splitting strategy

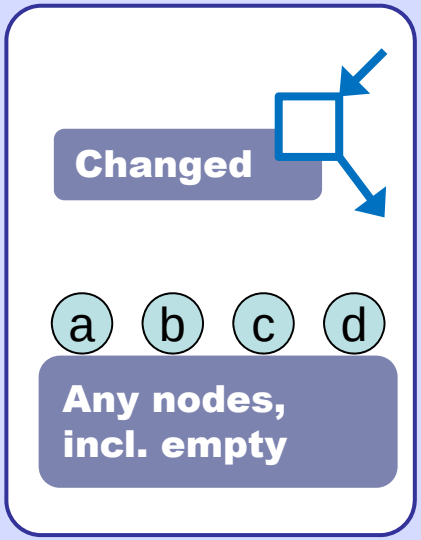




Note that splitting changes the height of the 2-3-4 tree only when the root is splitted.

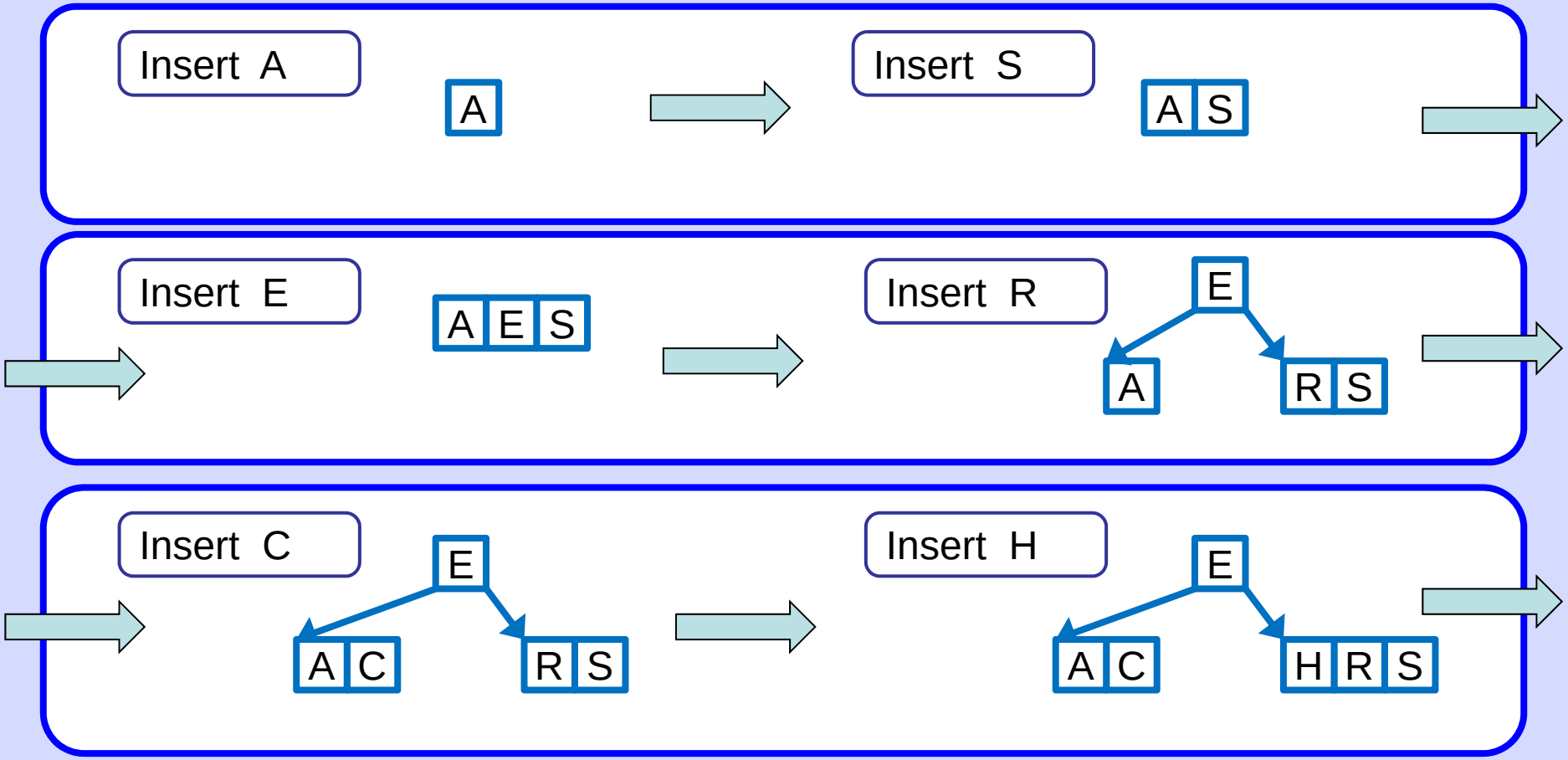
Insert: Splitting strategy



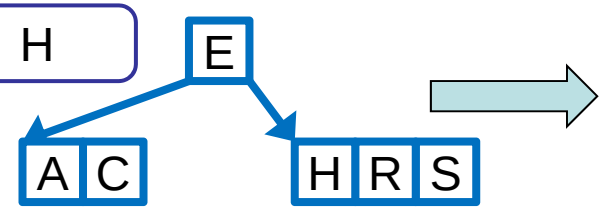


Note that splitting changes the height of the 2-3-4 tree only when the root is splitted.

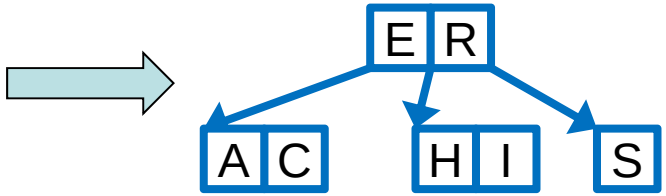
Insert keys into initially empty 2-3-4 tree: A S E R C H I N G X



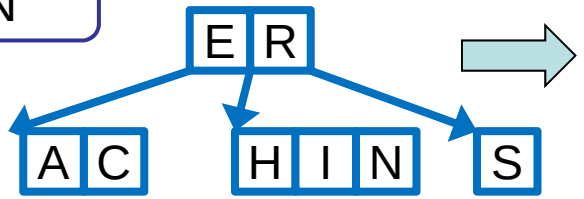
... Insert H



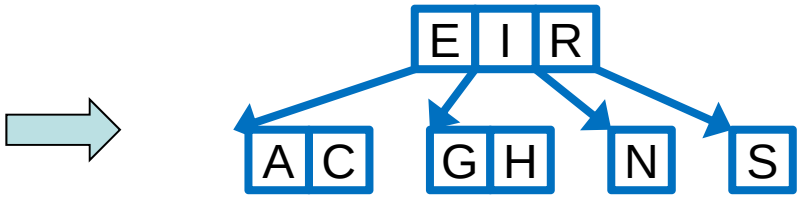
Insert I



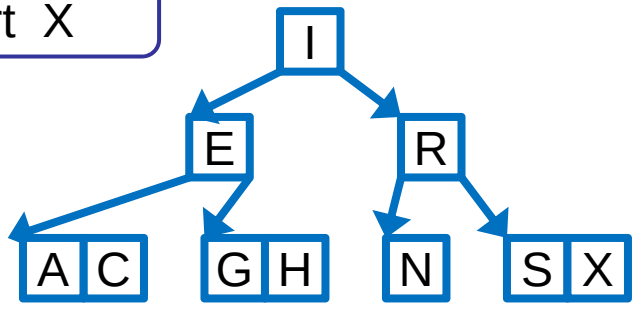
Insert N



Insert G



Insert X

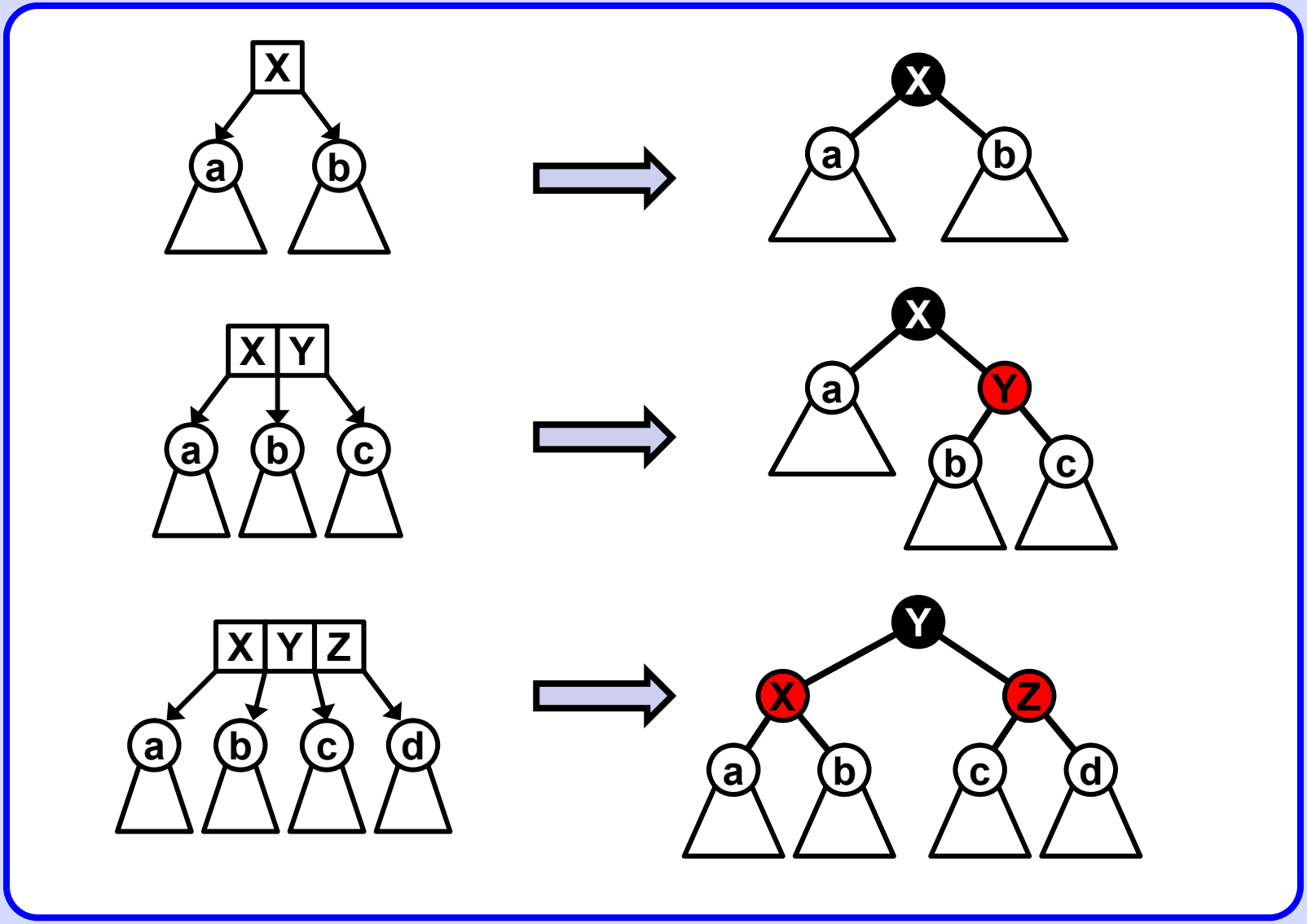


Note seemingly unnecessary split of EIR 4-node during insert of G.

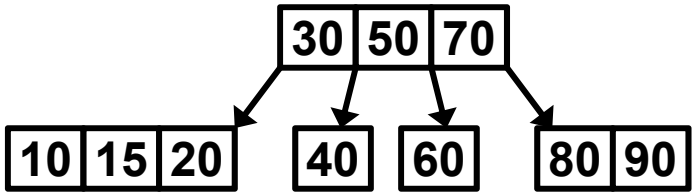
Results of an experiment with N uniformly distributed random keys from range {1, ..., 10^9 } inserted into initially empty 2-3-4 tree:

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

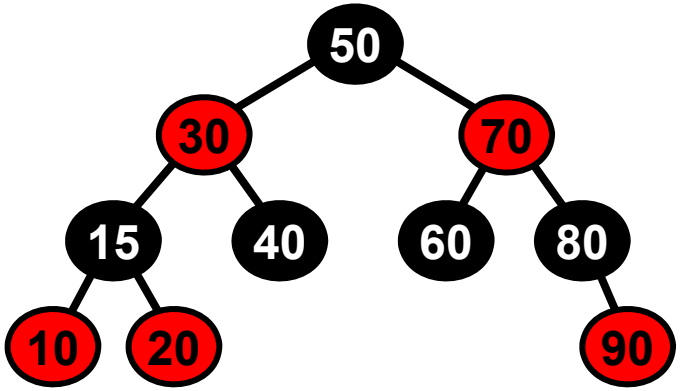
Relation of a 2-3-4 tree to a red-black tree



Relation of a 2-3-4 tree to a red-black tree



Original 2-3-4 tree



Equivalent r-b tree

B+ tree

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

The differences are:
-- Records (or pointers to actual records) are stored only in the leaf nodes,
-- internal nodes store only search key values which are used only as routers 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.

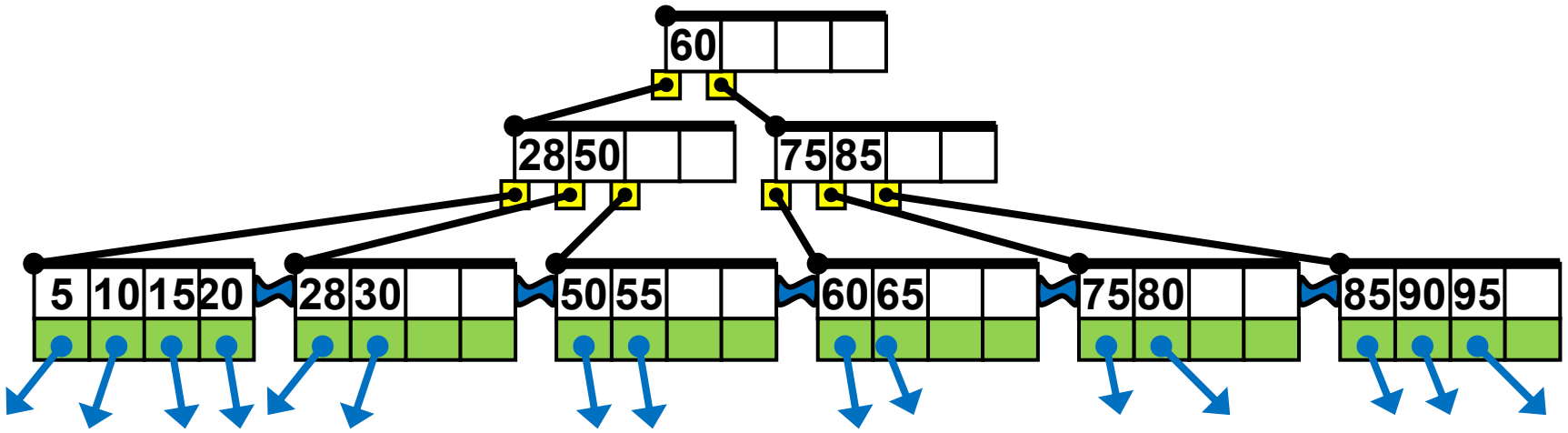
Complexities

Searches in N-node 2-3-4 tree visit at most lg(N) + 1 nodes

Insertion into N-node 2-3-4 tree requires 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 lg(N), and that is not even approached in practical situations.

** Now, finally, there is some challenge for you!



Routers and keys 75

Data records or pointers to them

Leaves links

Values in internal nodes are routers, originally each of them was a key when a record was inserted. Insert and Delete operations split and merge the nodes and thus move the keys and routers around. A router may remain in the tree even after the corresponding record and its key was deleted.

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 data record) into B+ tree

Find, as in B tree, correct leaf to insert K. Then there are 3 cases:

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 the parent node? YES.

1. Consider all keys in the leaf, including K, to be sorted.
2. Insert middle (median) key M in the parent node in the appropriate slot Y.
   (If parent does not exist, first create an empty one = new root.)
3. Split the leaf to two new leaves L1 and L2.
4. Left leaf (L1) from Y contains records with keys smaller than M.
5. Right leaf (L2) from Y contains records with keys equal to or greater than M.

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

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

Find, as in B tree, correct leaf to insert K. Then there are 3 cases:

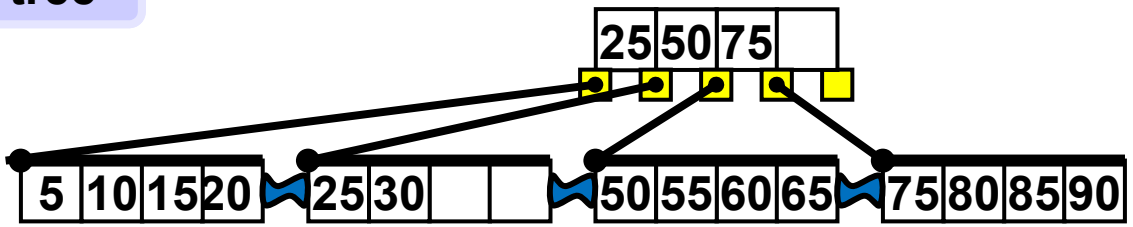
Case 3

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

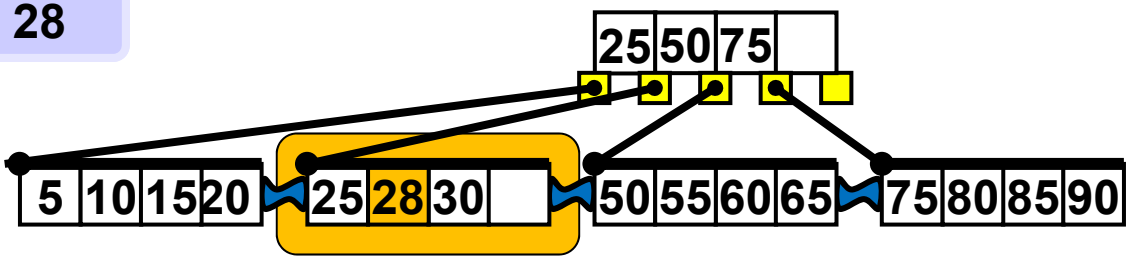
1. Split the leaf to two leaves L1 and L2, consider all its keys including K sorted, denote M median of these keys.
2. Records with keys < M go to the left leaf L1.
3. Records with keys >= M go to the right leaf L2.

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.
   (If PP does not exist, first create an empty one = new root.)
   Else set M := M1, P := PP and continue splitting parent nodes recursively up the tree, repeating from step 4.

Initial tree



Insert 28

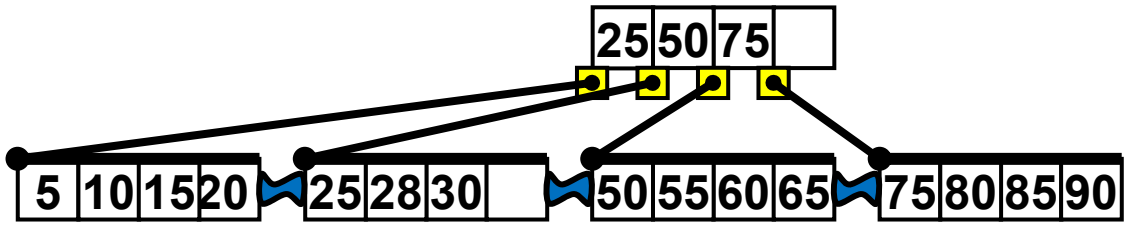


Changes

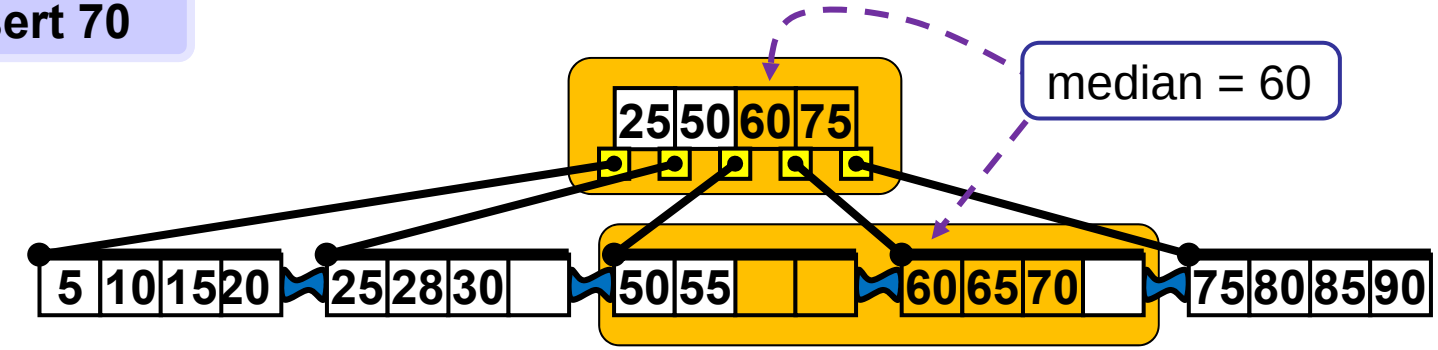
Leaves links

Data records and pointers to them are not drawn here for simplicity's sake.

Initial tree



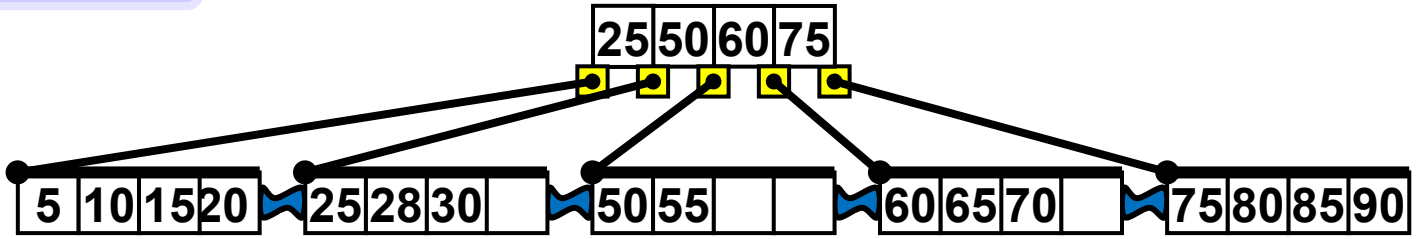
Insert 70



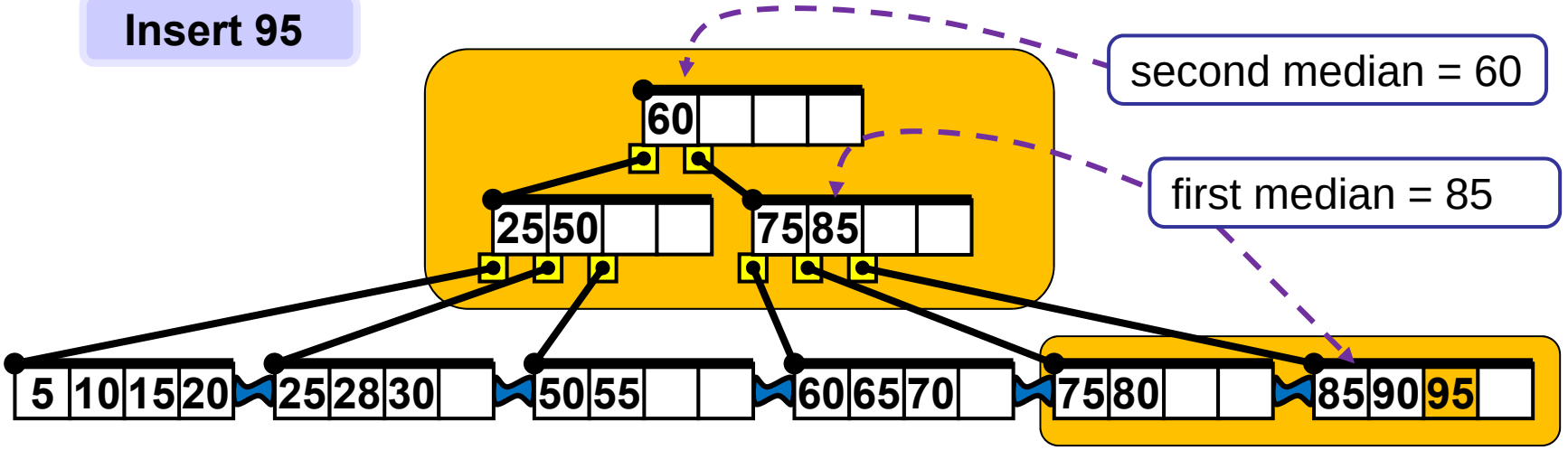
Changes

Leaves links

Initial tree



Insert 95



Changes

Leaves links

Note the router 60 in the root, detached from its original position in the leaf.

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

Find, as in B tree, key K in a leaf. Then there are 3 cases:

Case 1

Leaf more than half full or leaf == root? YES.

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

Case 2

Leaf more than half full? NO. Left or right sibling more than half full? 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 L and parent P2 of sibling S.
(If S does not exist then L is the root, which may contain any number of keys).

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

Find, as in B tree, key K in a leaf. Then there are 3 cases:

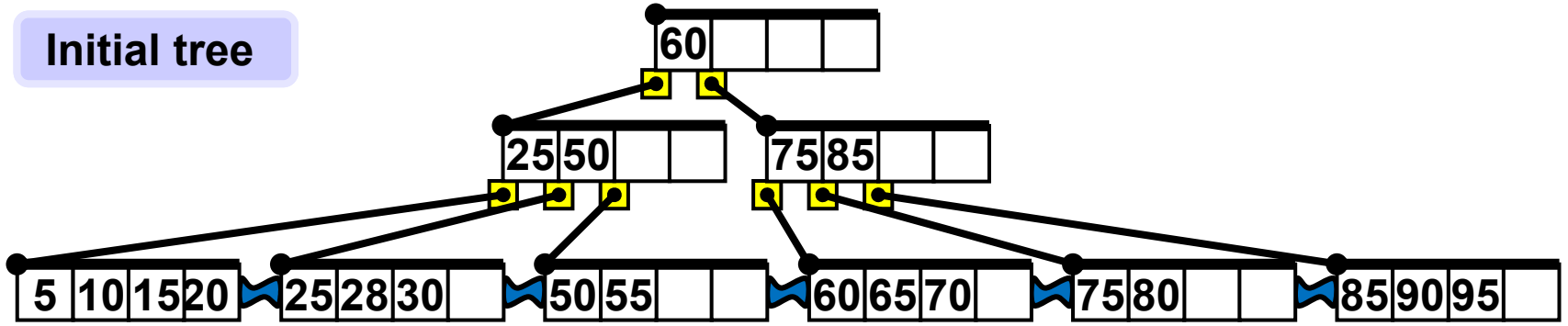
Case 3

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

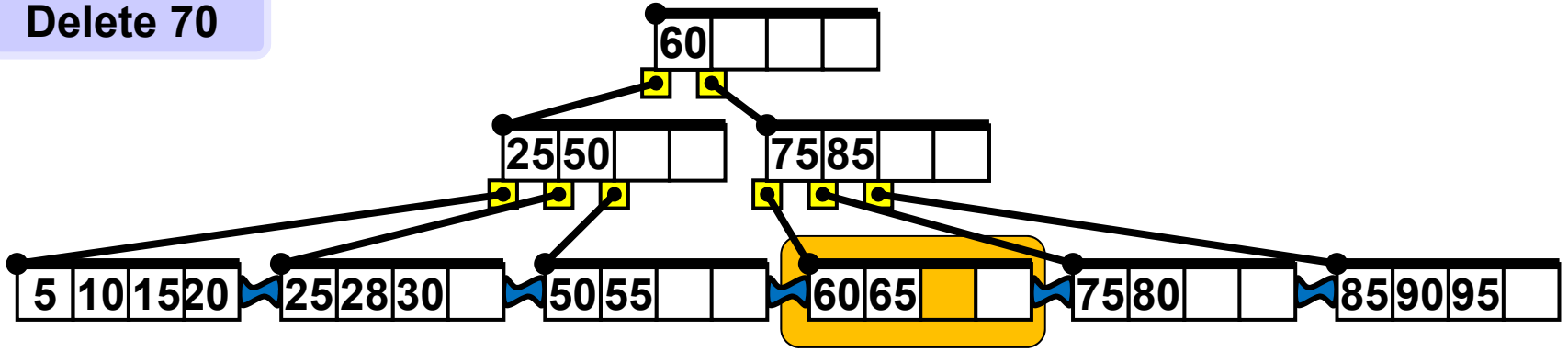
1. Consider sibling S of L which has same parent P as L.
2. Consider set M of ordered keys of L and S without K but together with key K1 in P which separates L and S.
3. Merge: Store M in L, connect L to the other sibling of S (if exists), destroy S.
4. Set the reference left to K1 to point to L. Delete K1 from P. If P contains K delete it also from P. 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 set L := P and continue recursively up the tree (like in B-tree), repeating from step 1.

Note: Merging leaves and inner nodes works same way as in B-trees.

Initial tree



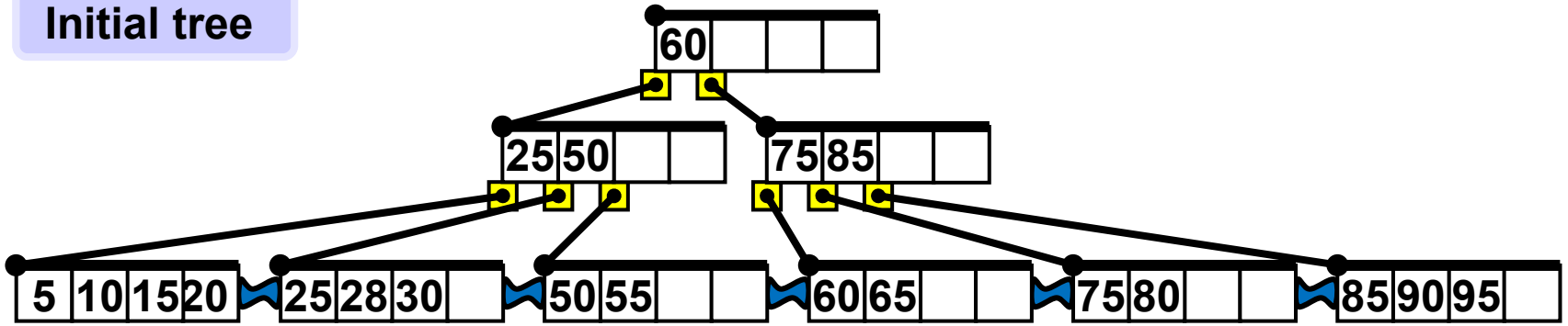
Delete 70



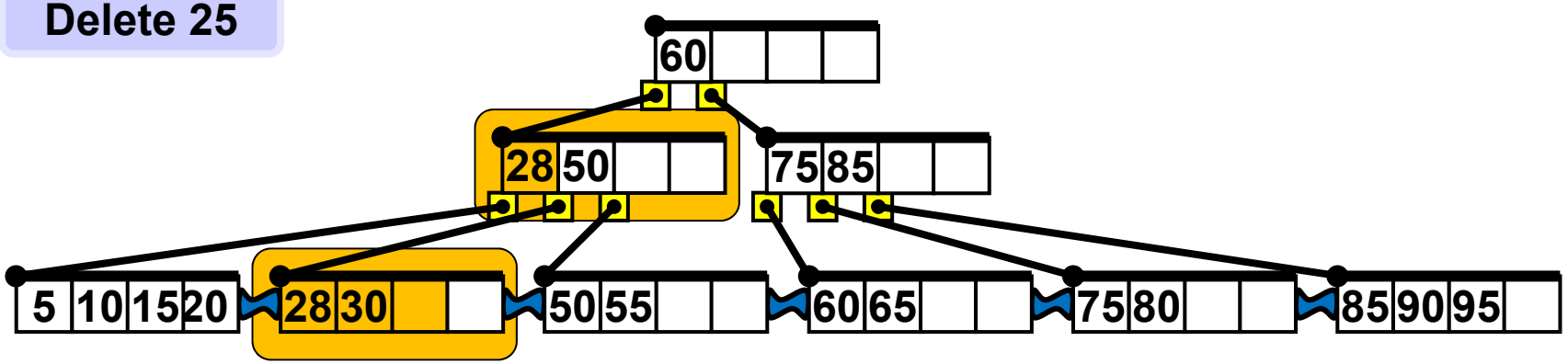
Changes

Leaves links

Initial tree



Delete 25

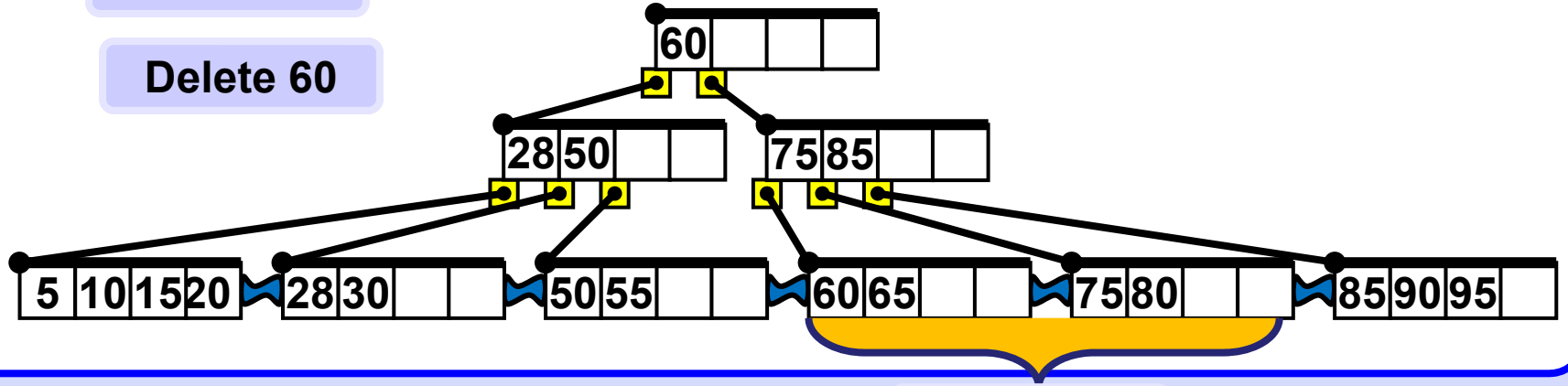


Changes

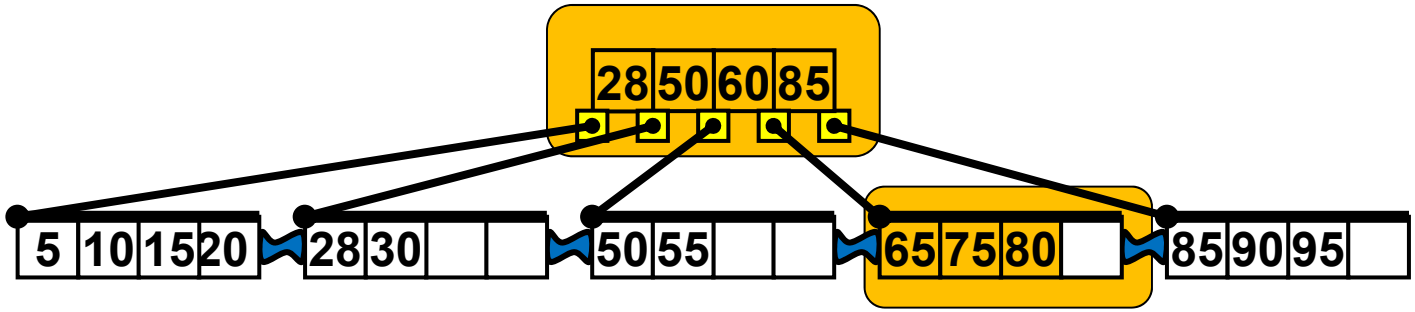
Leaves links

Initial tree

Delete 60



Join here

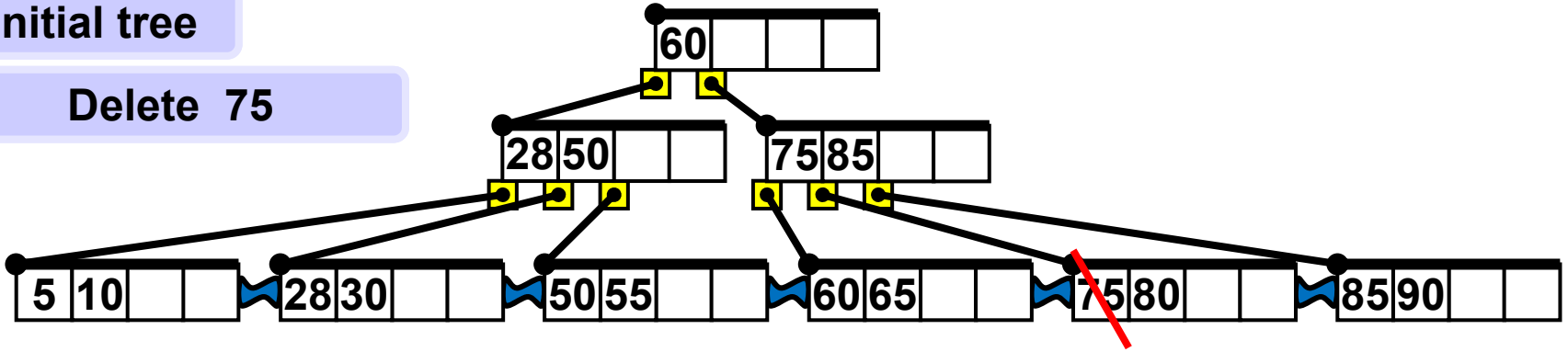


Changes

Leaves links

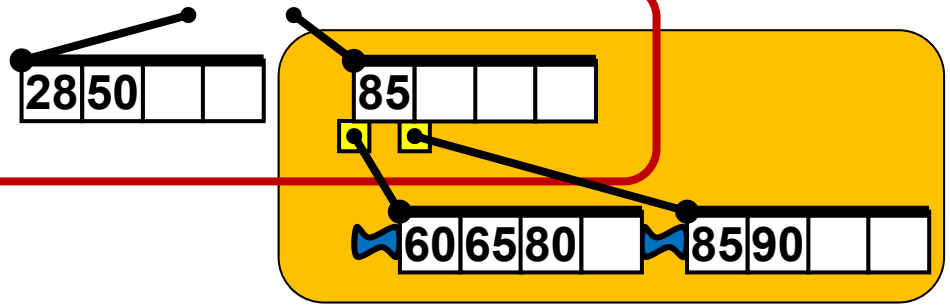
Initial tree

Delete 75

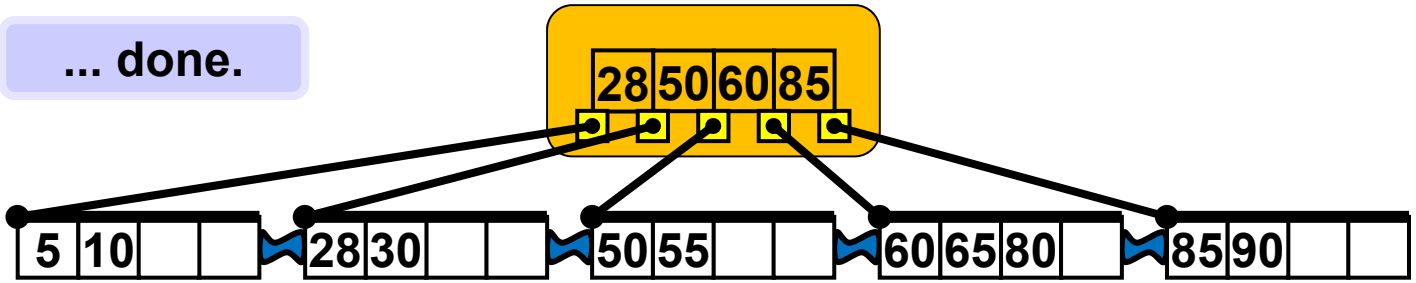


Too few keys, join these two nodes and bring key from parent (recursively).

Progress...



... done.



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.

Ben Pfaff: **Performance Analysis of BSTs in System Software**

Stanford University, Department of Computer Science

Conclusions:

- ...Unbalanced BSTs are best when randomly ordered input can be relied upon;
- if random ordering is the norm but occasional runs of sorted order are expected, then red-black trees should be chosen.
- On the other hand, if insertions often occur in a sorted order, AVL trees excel when later accesses tend to be random,
- and splay trees perform best when later accesses are sequential or clustered.

Some consequences:

Managing virtual memory areas in OS kernel:
... Many kernels use BSTs for keeping track of VMAs:
Linux before 2.4.10 used AVL trees, OpenBSD and later versions of Linux use red-black trees, FreeBSD uses splay trees, and so does Windows NT for its VMA equivalents...

tree / time in msec / order

Memory management supporting web browser

BST	AVL	RB	splay
15.67	3.65	3.78	2.63
4	2	3	1

Artificial uniformly random data

BST	AVL	RB	splay
1.63	1.67	1.64	1.94
1	3	2	4

Secondary peer cache tree

BST	AVL	RB	splay
3.94	4.07	3.78	7.19
2	3	1	4

Processing identifiers cross-references

BST	AVL	RB	splay
4.97	4.47	4.33	4.00
4	3	2	1