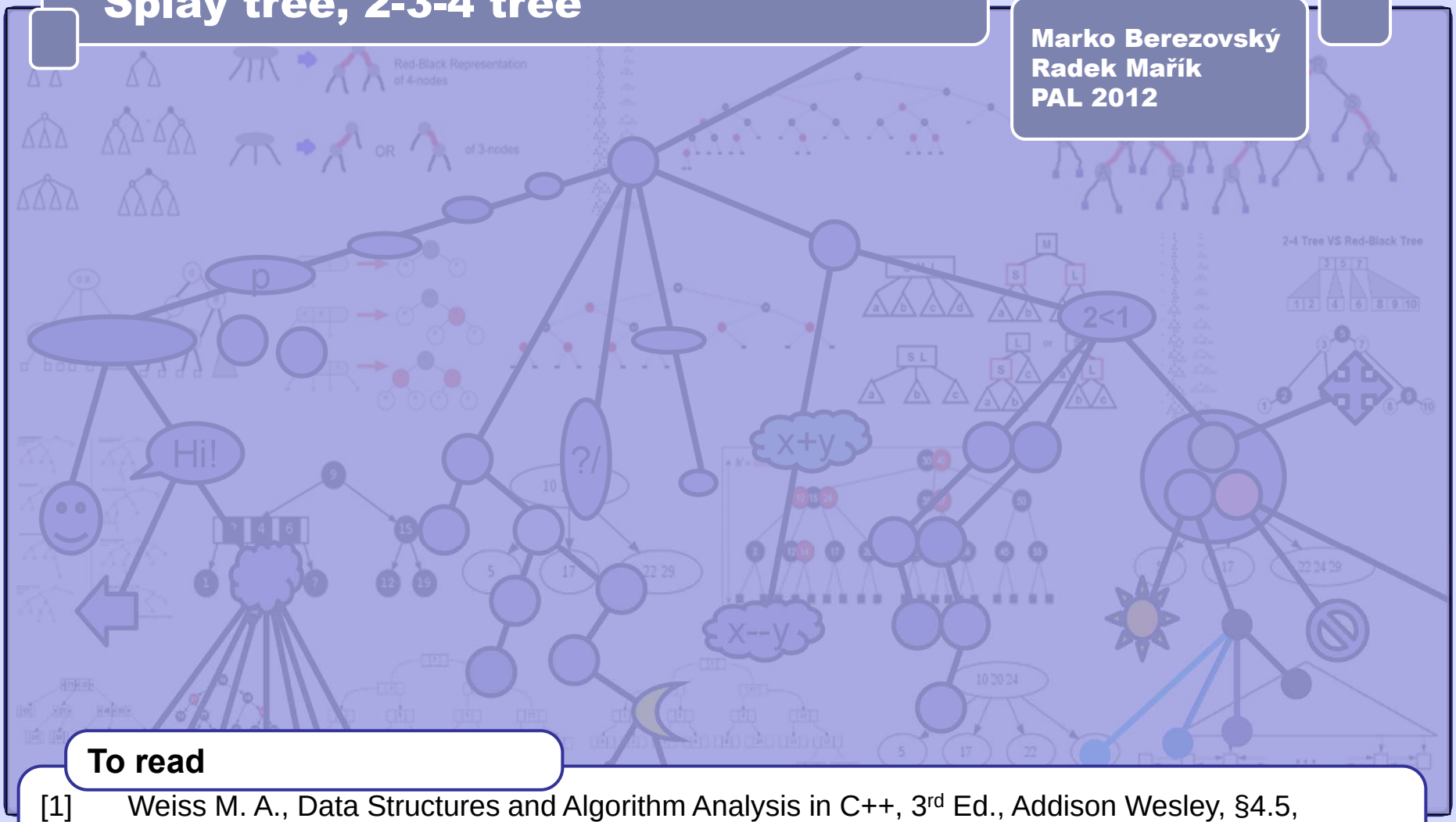
Splay tree, 2-3-4 tree

Marko Berezovský
Radek Mařík
PAL 2012



To read

[1] Weiss M. A., Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §4.5, pp.149-58.

[2] Daniel D. Sleator and Robert E. Tarjan, "Self-Adjusting Binary Search Trees", Journal of the ACM 32 (3), 1985, pp.652-86.

See also PAL webpage for references

Splay Tree - Description

AVL trees and red-black trees are binary search trees with logarithmic height This ensures all operations are $O(\ln(n))$

An alternative idea is to make use of an old maxim:

> Data that has been recently accessed is more likely to be accessed again in the near future.

Accessed nodes could be rotated or *splayed* to the root of the tree:

- Accessed nodes are splayed to the root during the count/find operation
- Inserted nodes are inserted normally and then splayed
- The parent of a removed node is splayed to the root

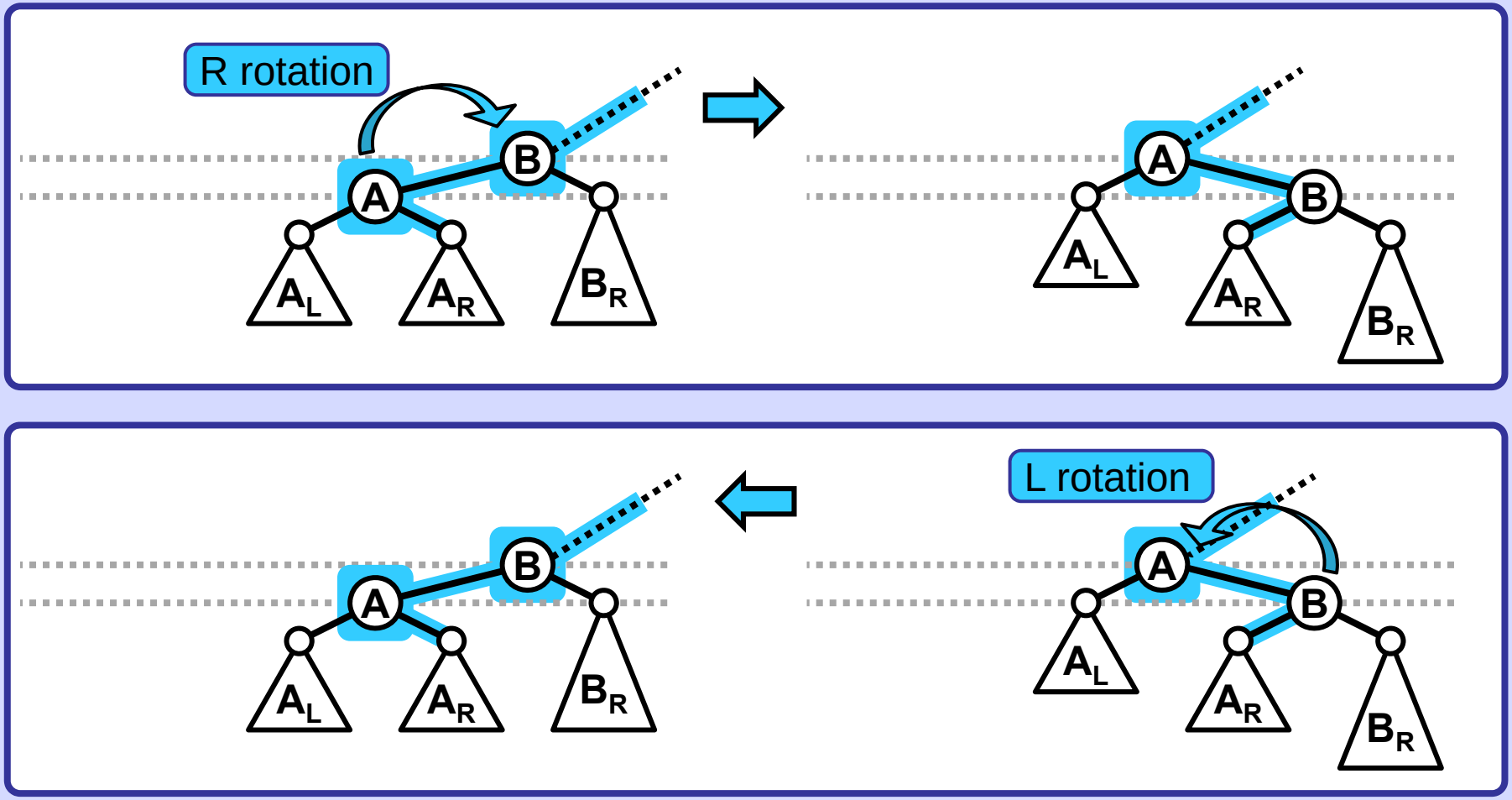
Invented in 1985 by Daniel Dominic Sleator and Robert Endre Tarjan.

Splay Tree - Description

- A binary search tree.
- Similar to, but different from, AVL trees.
- No additional tree shape description (memory!) is used.
- An alternate idea to optimizing run times.
- Each node access or insertion moves that node to the root.
- A possible height of $\Theta(n)$ but amortized run times of $O(\ln(n))$.
- Operations are *zig, zig-zig* and *zig-zag.*

Splay Tree - rotation

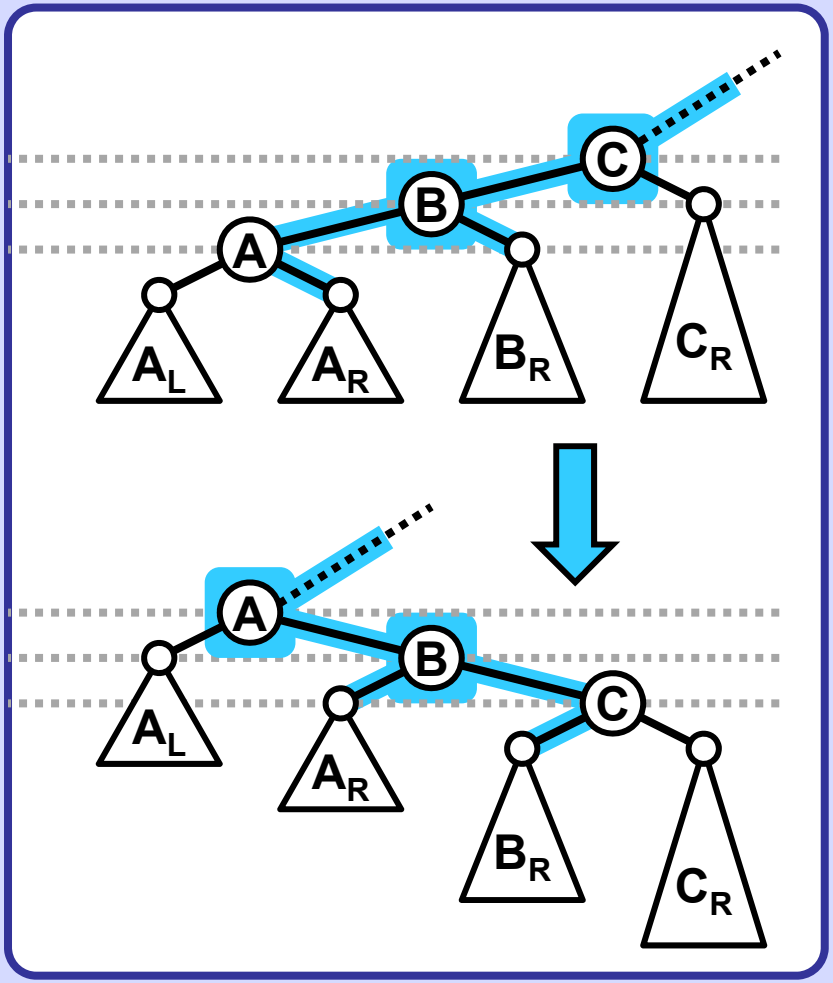
Zig



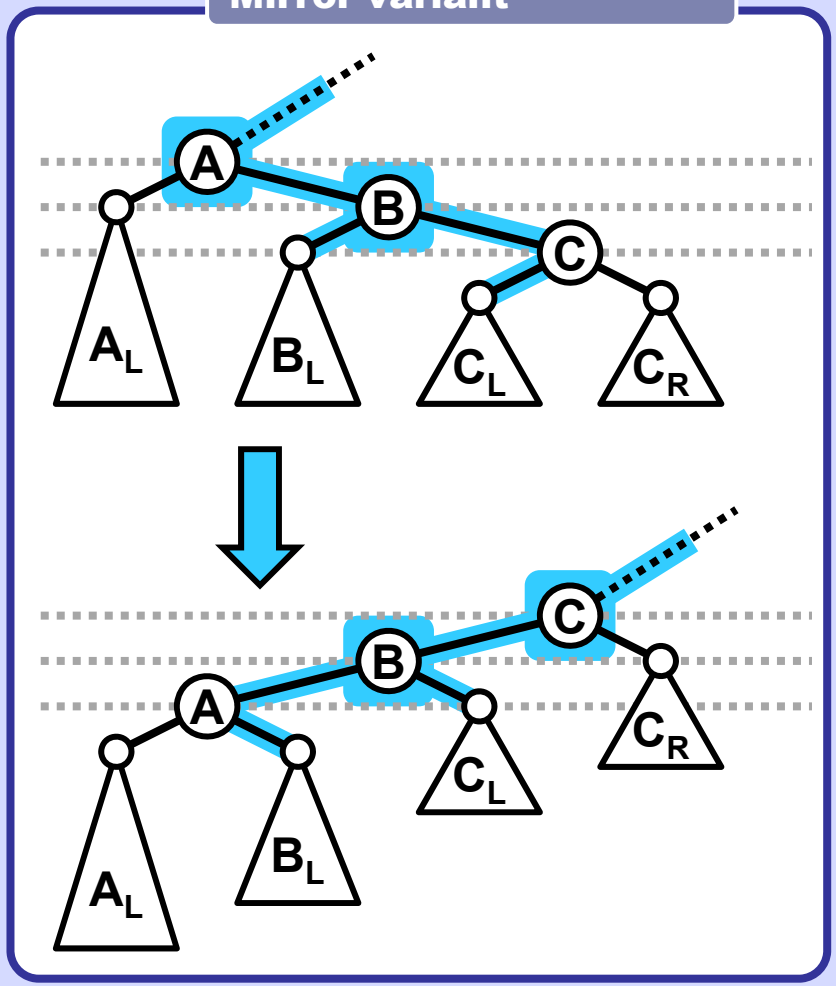
Note:

The terms "Zig" and "Zag" are not chiral, that is, they **do not** describe the direction (left or right) of the actual rotations.

Zig - Zig



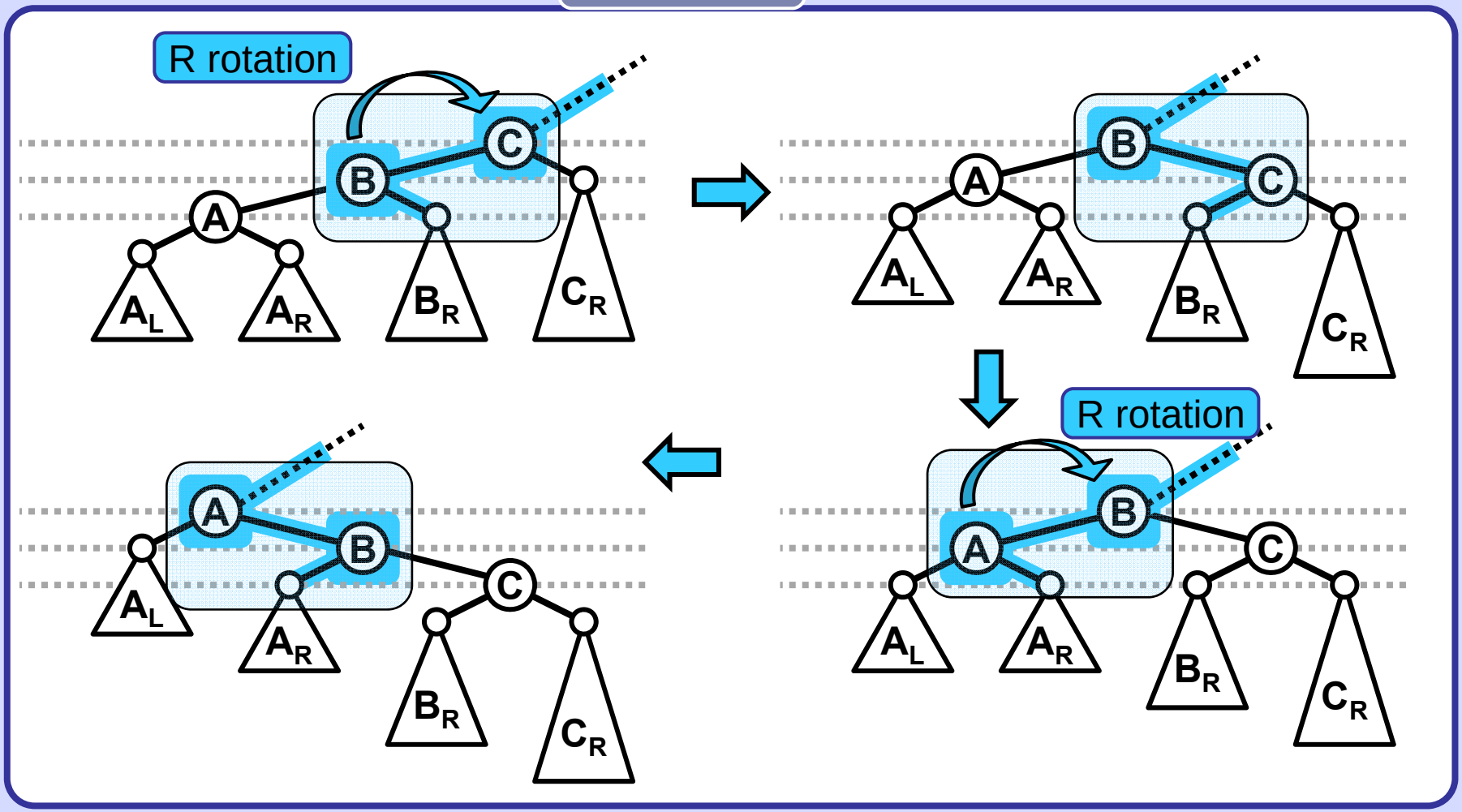
Mirror variant



Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.

Splay Tree - rotation

Zig - Zig

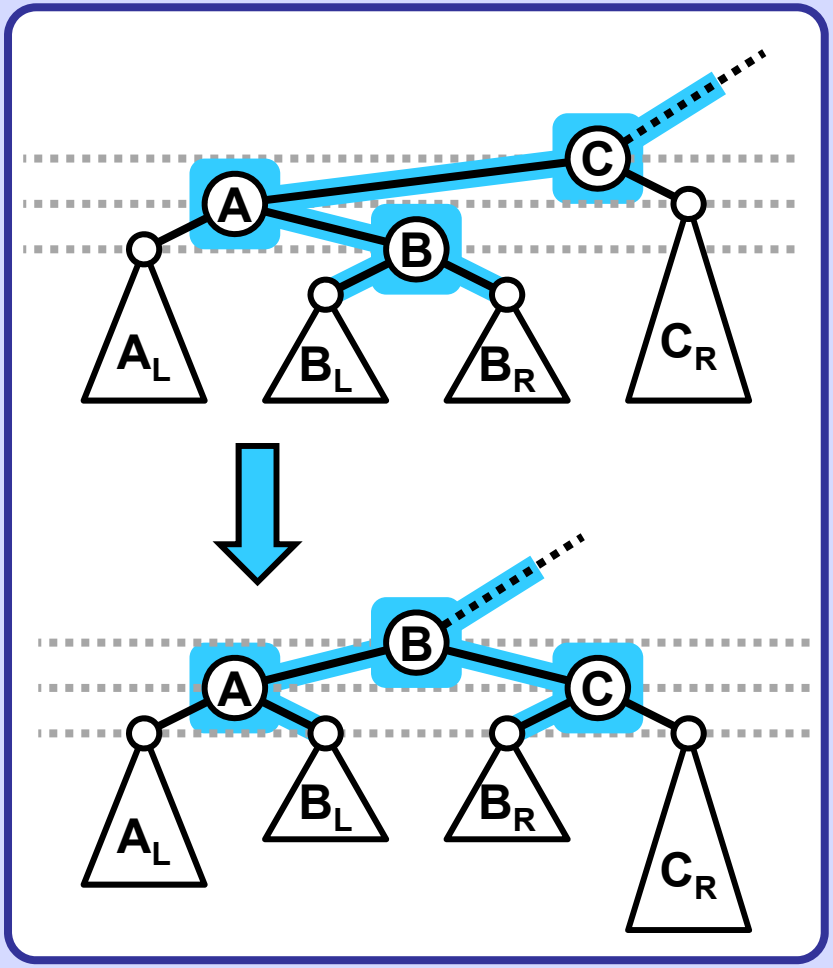


Note:

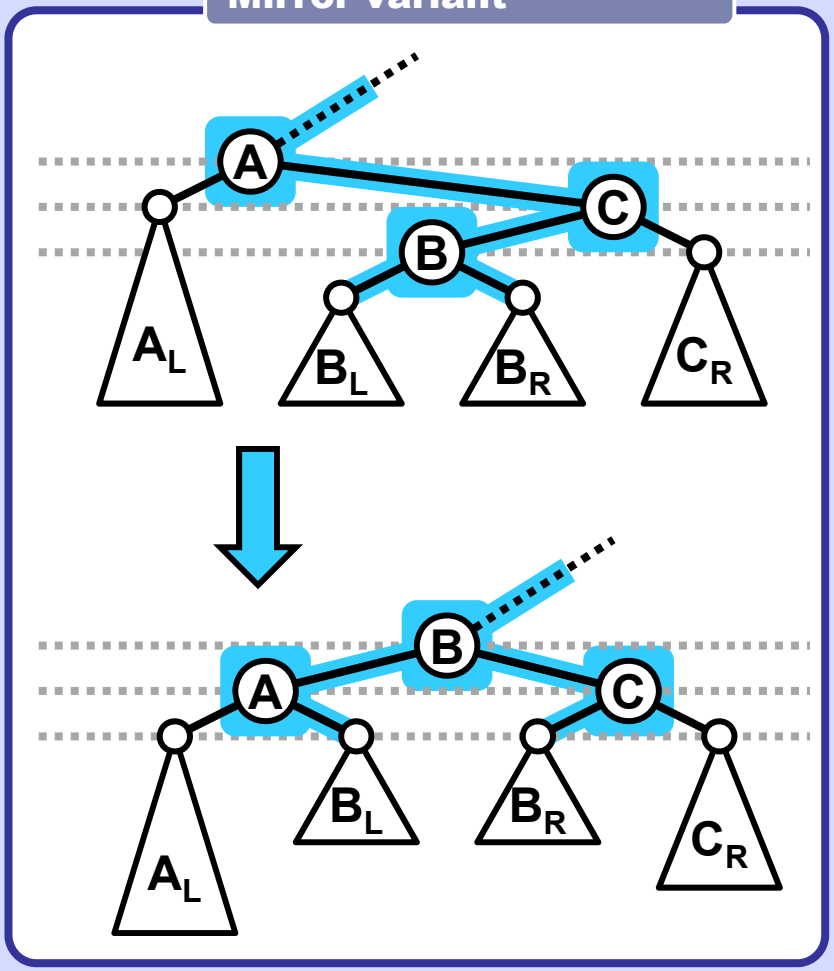
Both simple rotations are performed at the top of the current subtree therefore, the splayed node (with key A) **is not** involved in the first rotation.

Splay Tree - rotation

Zig - Zag



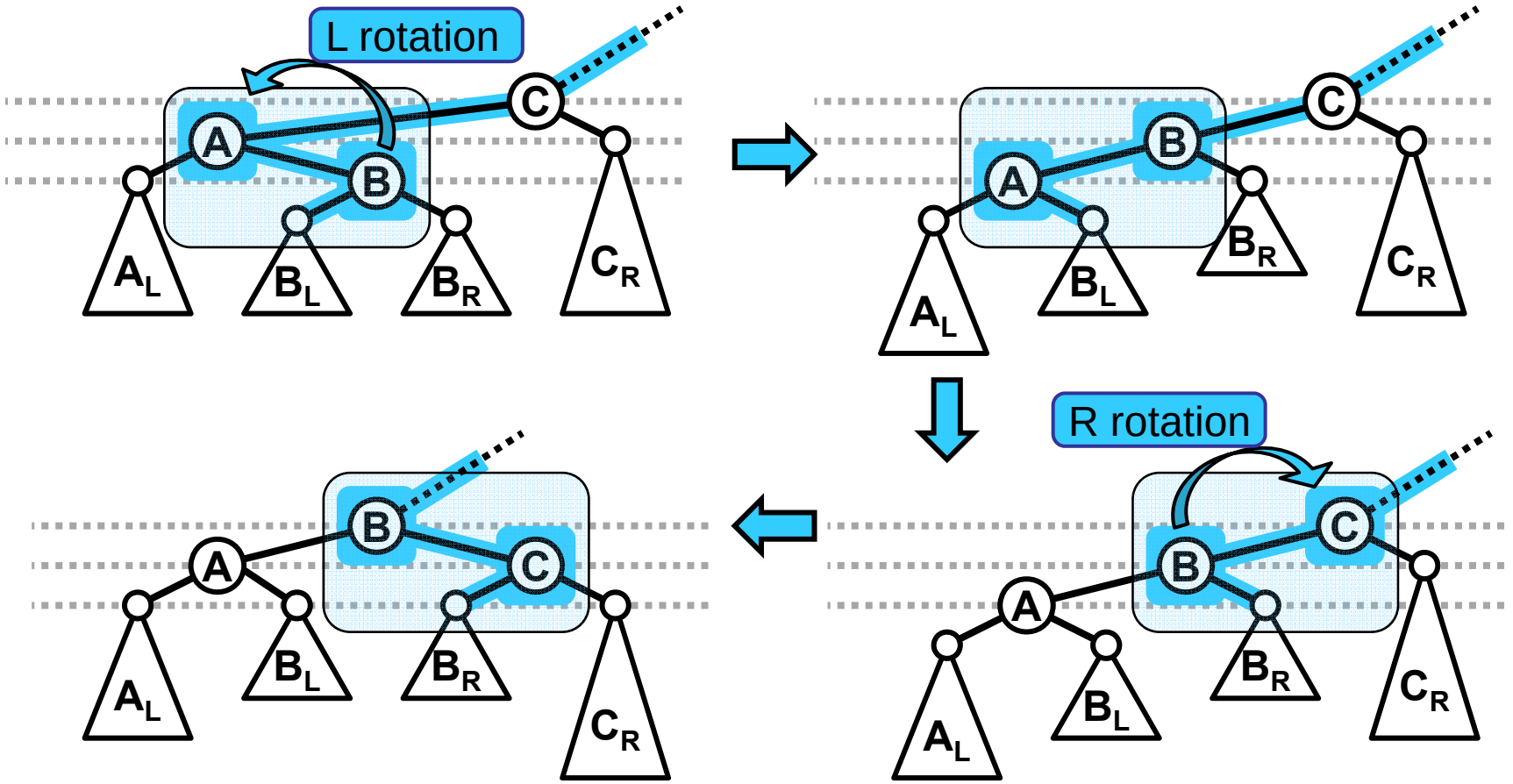
Mirror variant



Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.

Splay Tree - rotation

Zig - Zag

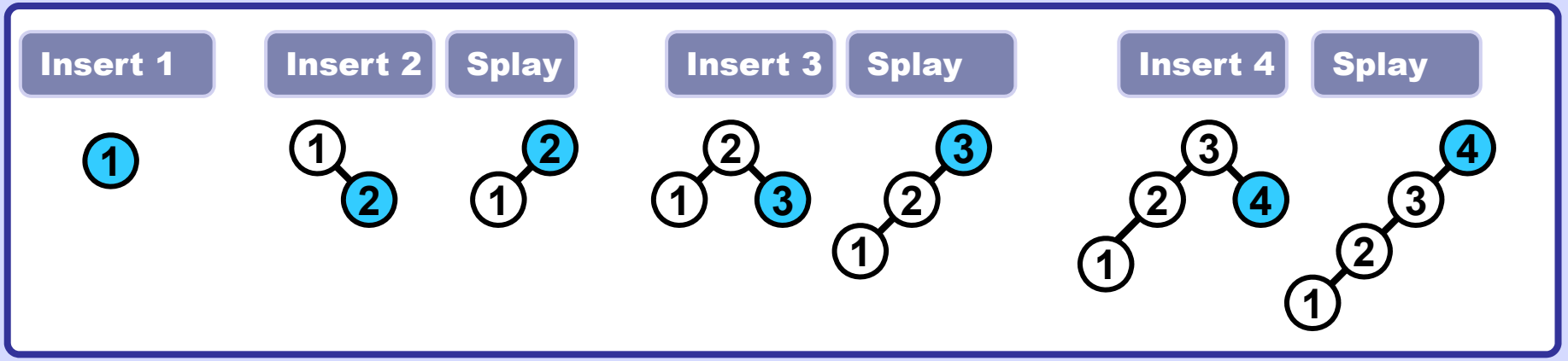


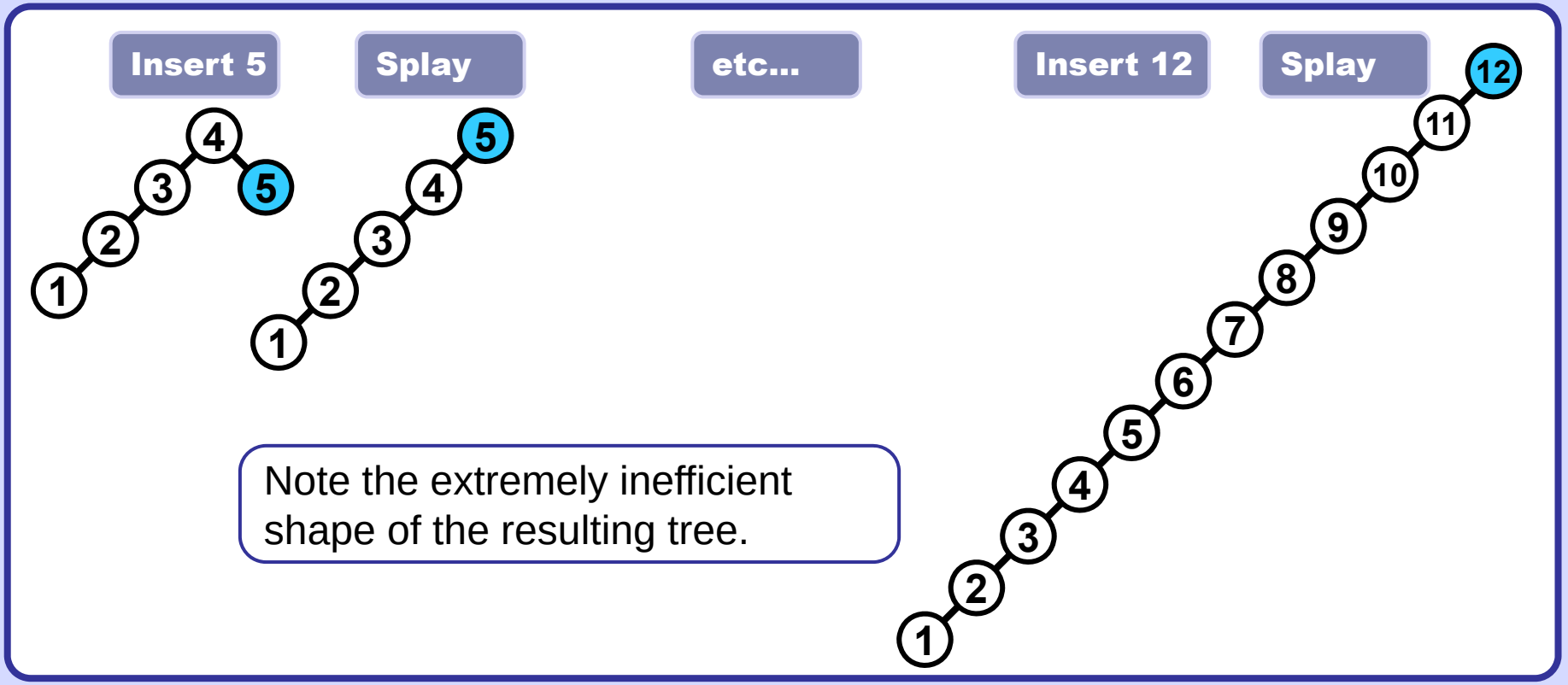
Note:

Zig-Zag rotation is identical to the double (LR or RL) rotation in AVL tree.

Splay Tree - Insert

Example



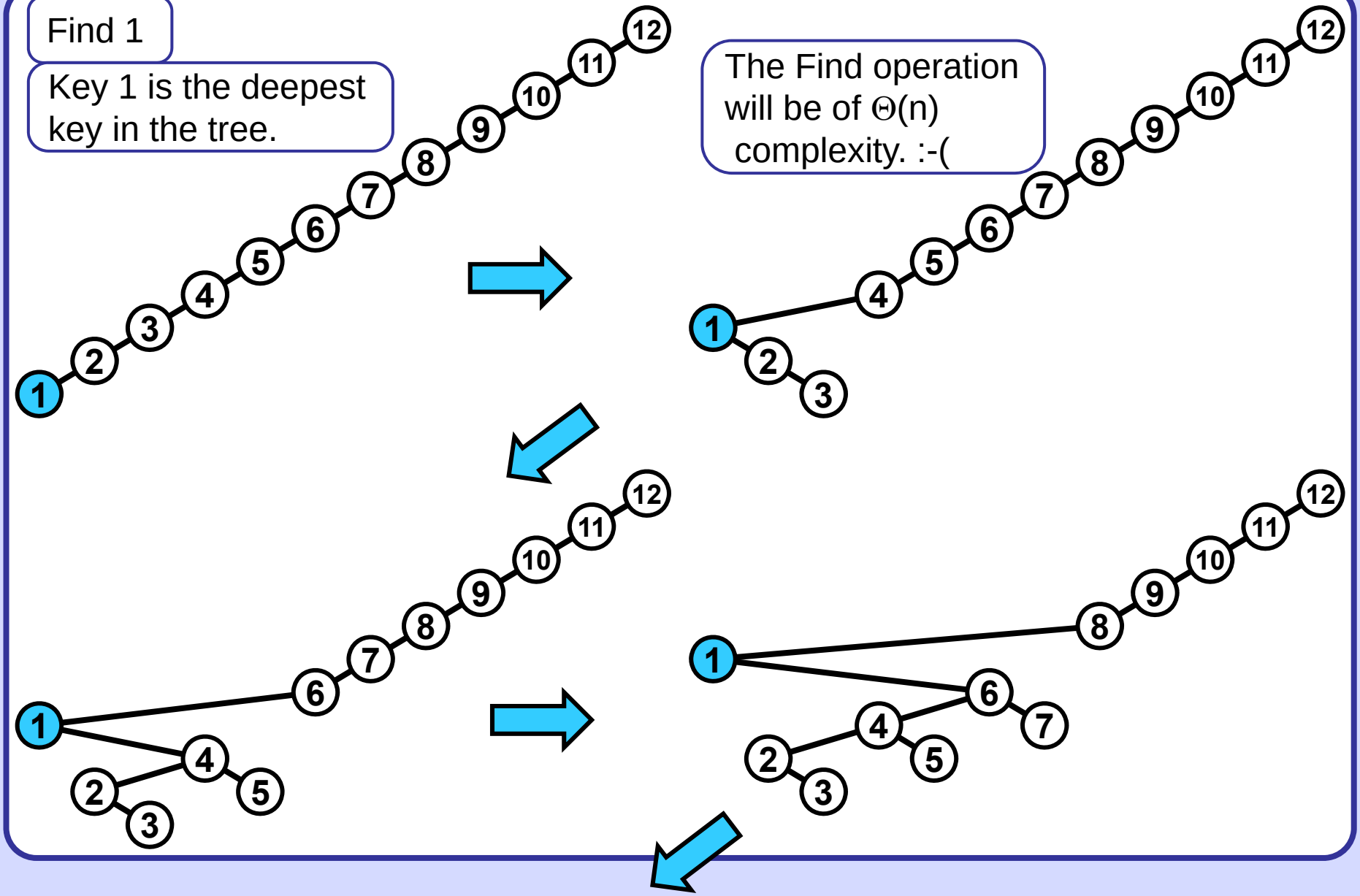


Note the extremely inefficient shape of the resulting tree.

Find 1

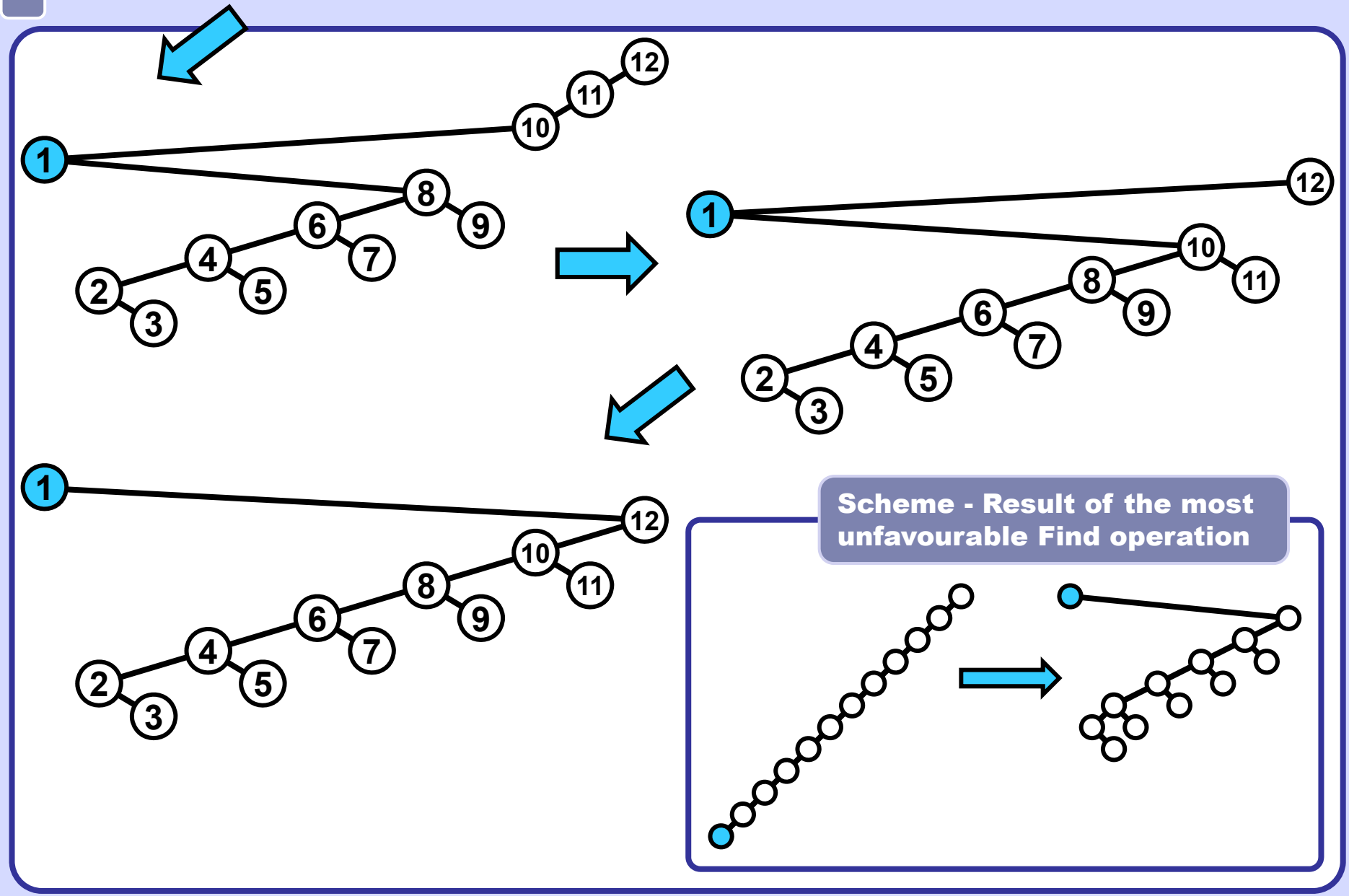
Key 1 is the deepest key in the tree.

The Find operation will be of Θ(n) complexity. :-(



Splay Tree - Insert

Example

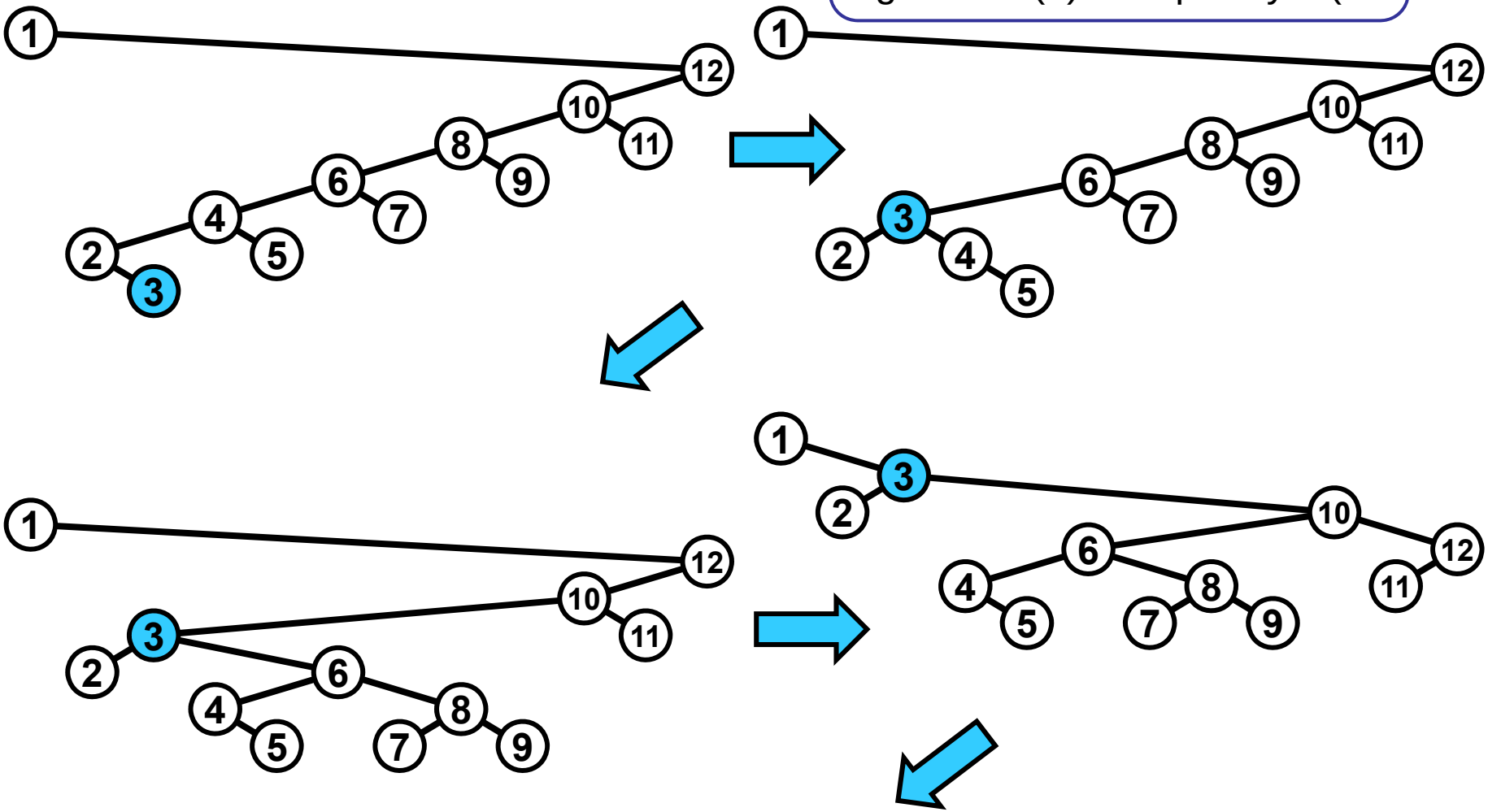


Scheme - Result of the most unfavourable Find operation

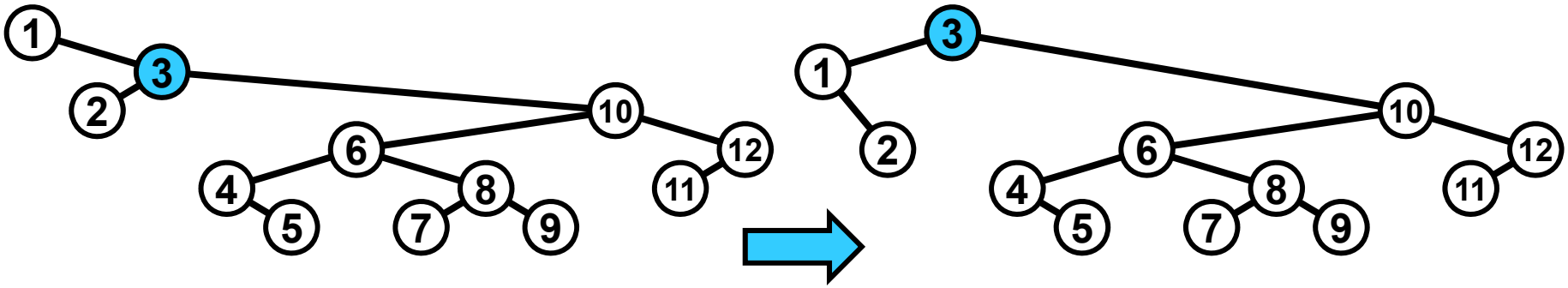
Find 1

Key 3 is the deepest key in the tree.

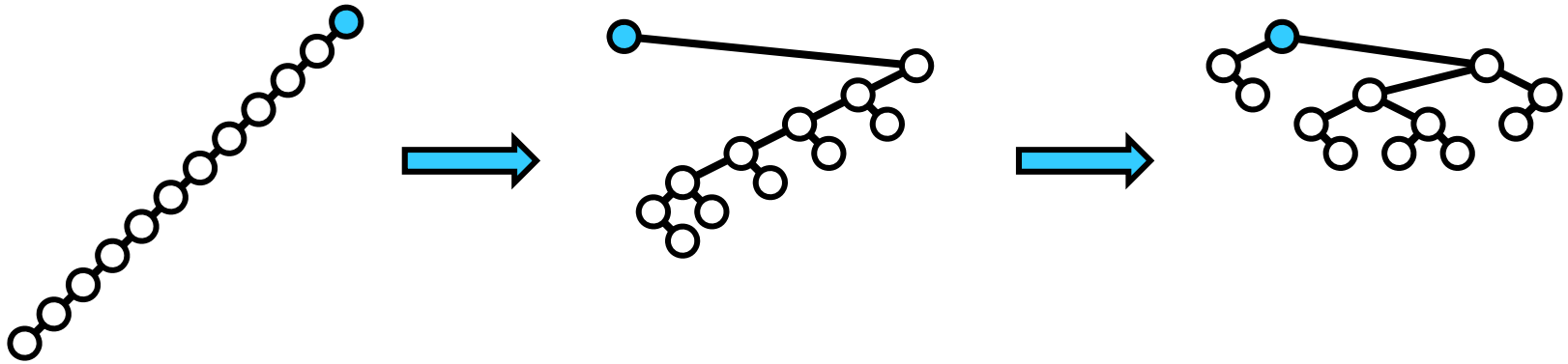
The Find operation will be again of $\Theta(n)$ complexity. :-(



Splay Tree - Insert

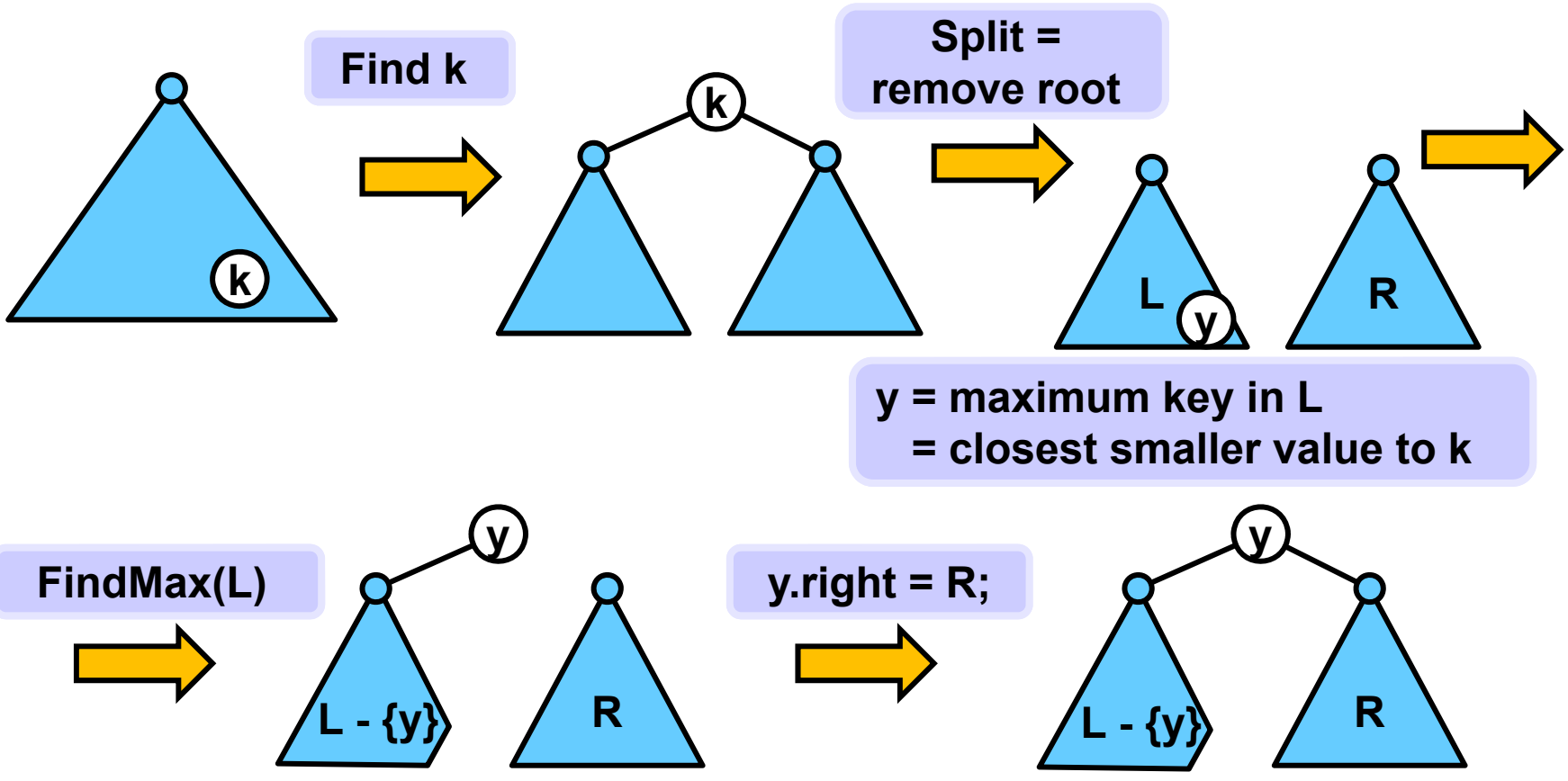


Scheme - Progress of the two most unfavourable Find operations.



Note the relatively favourable sgape of the resulting tree.

1. Find(k); // This splays k to the root
2. Remove the root; // Splits the tree into L and R subtree of the root.
3. y = Find max in L subtree; // This splays y to the root of L subtree
4. y.right = R subtree;



It is very difficult with small trees to demonstrate the amortized logarithmic behaviour of splay trees

The original ACM article [2] proves the *balance theorem*:

> The run time of performing a sequence of m operations on a splay tree with n nodes is $O(\ m(1 + \ln(n)) + n \ln(n)\ )$.

Therefore the run time for a splay tree is comparable to any balanced tree assuming at least n operations.

From the time of introducing splay trees (1985) up till today the following conjecture (among others) remains unproven.

Dynamic optimality conjecture[2]

Consider any sequence of successful accesses on an *n*-node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than O(*n*) plus a constant times the time required by algorithm A.

Advantages:

- The amortized run times are similar to that of AVL trees and red-black trees
- The implementation is easier
- No additional information (height/colour) is required

Disadvantages:

- The tree will change with read-only operations

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

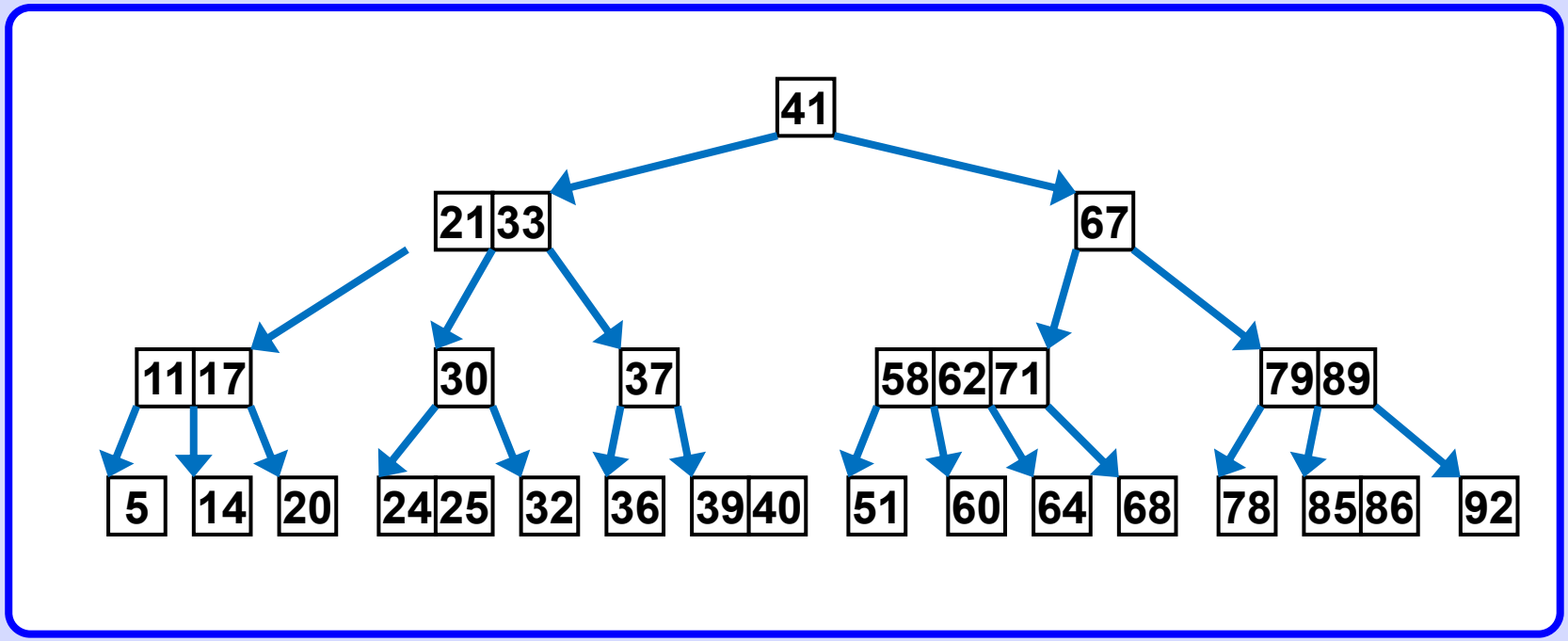
2-node, with one key, left link to a tree with smaller keys, and right link to a tree with larger keys;

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 maximum degree 4**.



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

Find: As in B-tree

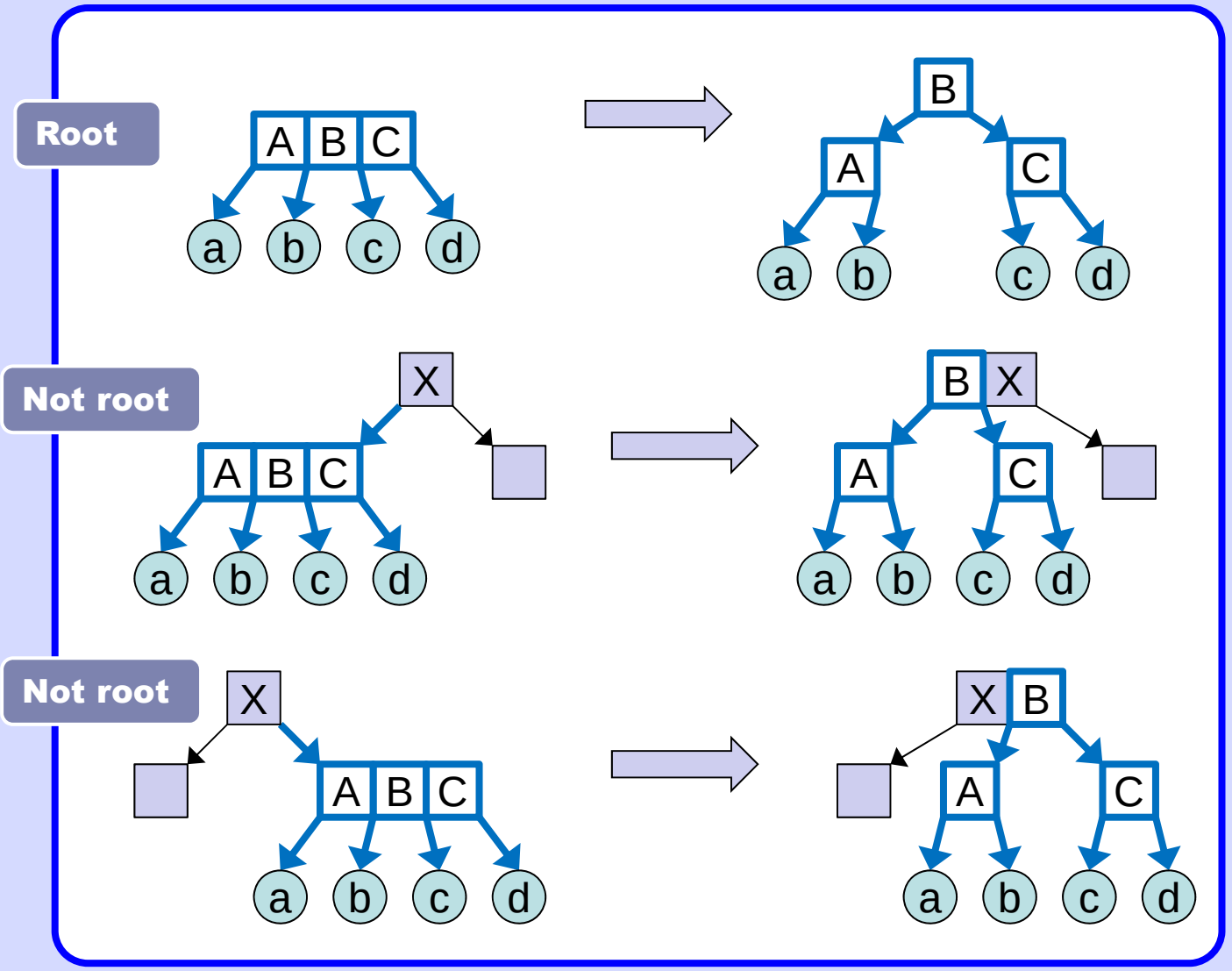
Insert: As in B-tree: Find the place for the inserted key x in a leaf and store it there. If necessary split the leaf.

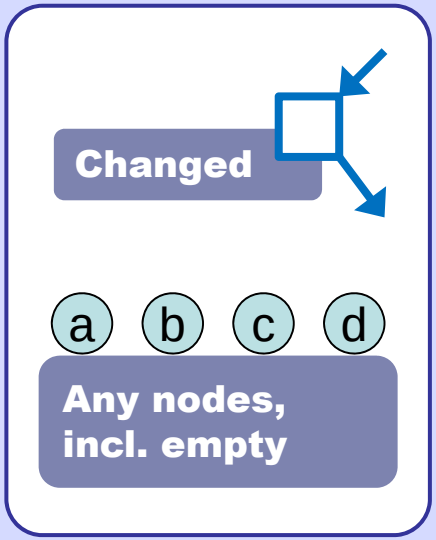
Additional **insert** rule:
In our way down the tree, whenever we reach a ***4-node***, we split it into two ***2- nodes***, and move the middle element up to the parent node.
This strategy prevents the following from happening:
After inserting a key it might happen in B tree that it is necessary to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

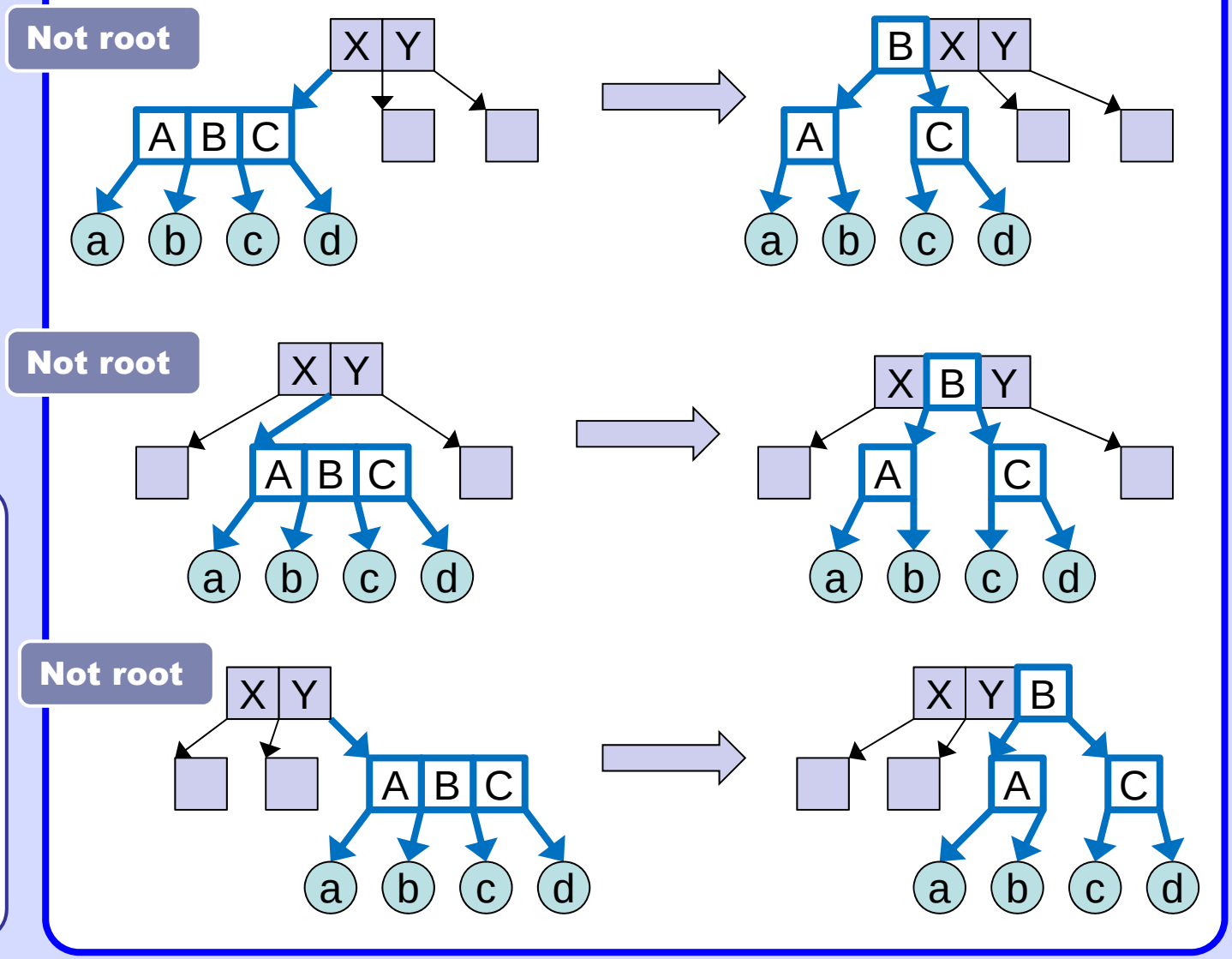
Insert: Splitting strategy

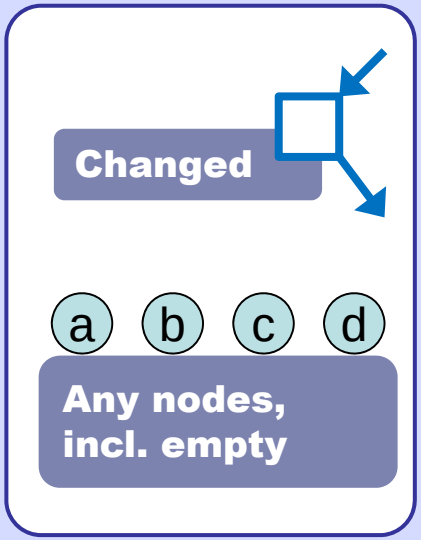




Note that splitting changes the height of the 2-3-4 tree only when the root is splitted.

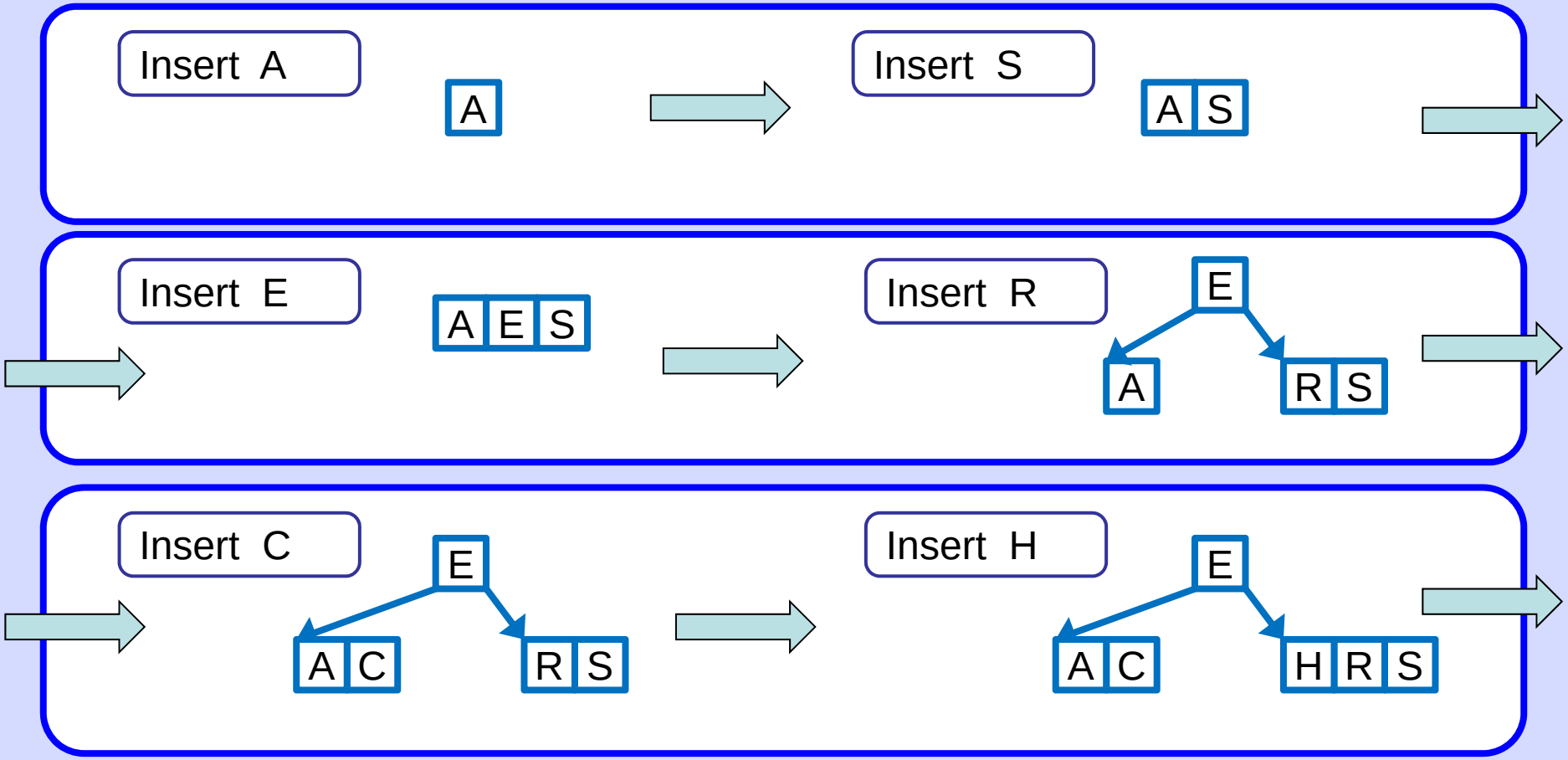
Insert: Splitting strategy



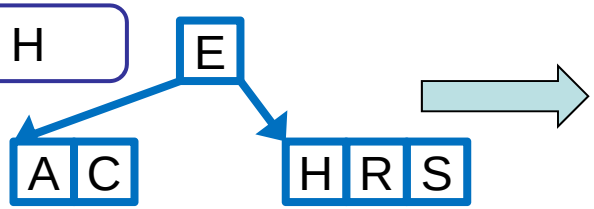


Note that splitting changes the height of the 2-3-4 tree only when the root is splitted.

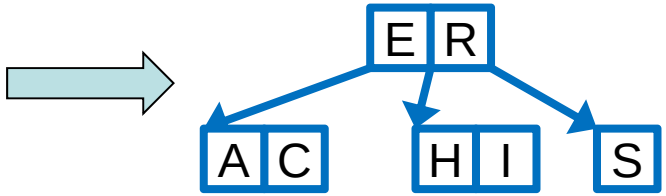
Insert keys into initially empty 2-3-4 tree: A S E R C H I N G X



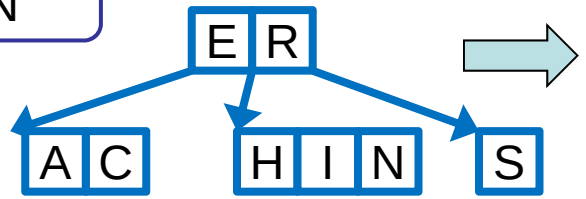
... Insert H



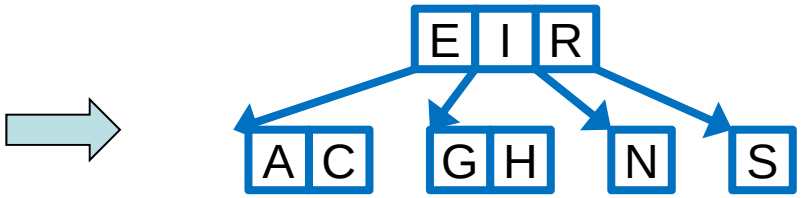
Insert I



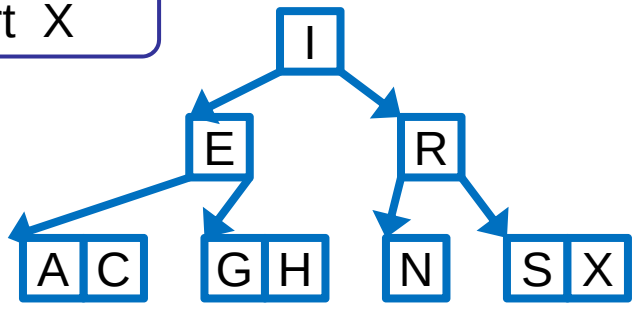
Insert N



Insert G



Insert X

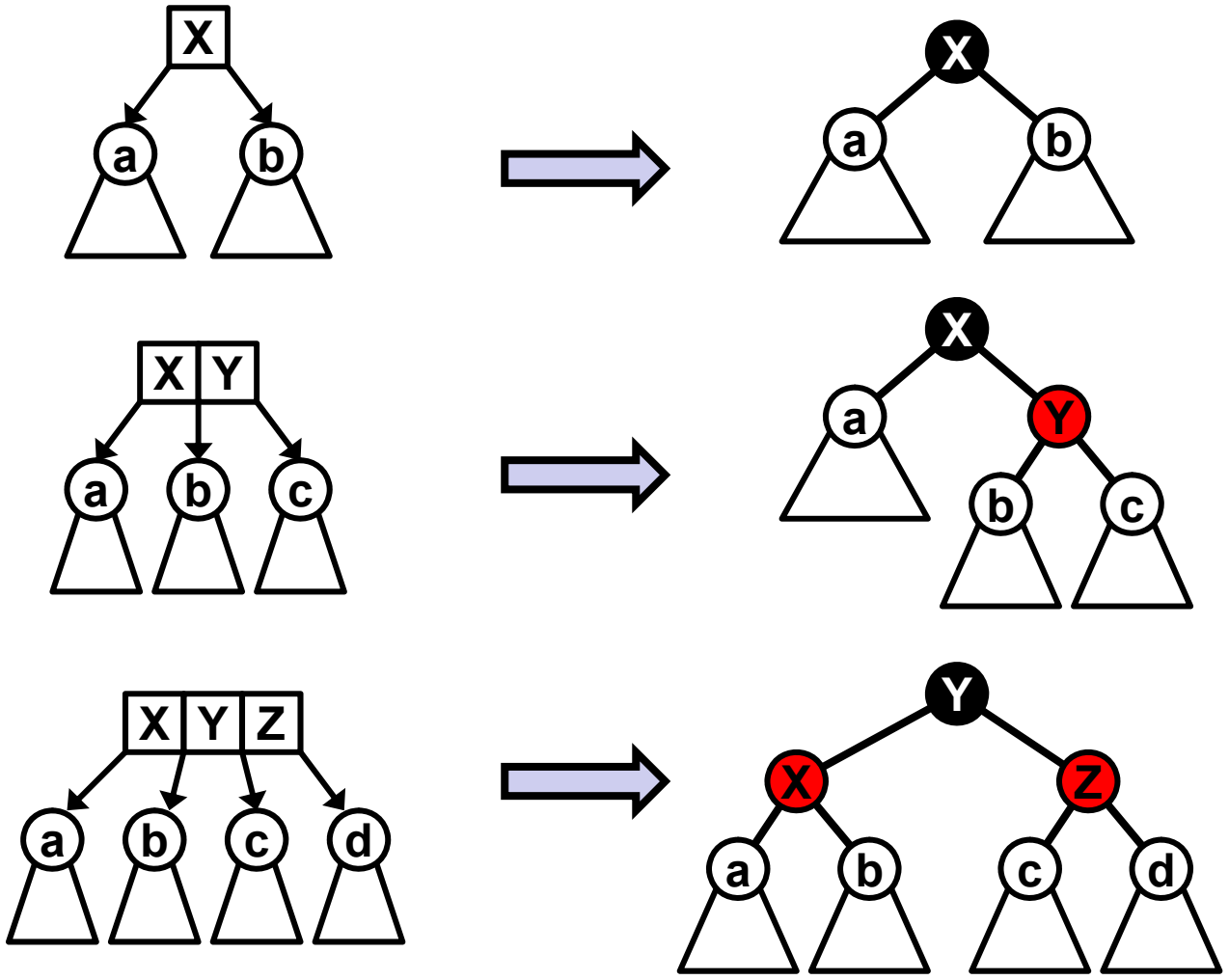


Note seemingly unnecessary split of EIR 4-node during insert of G.

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



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