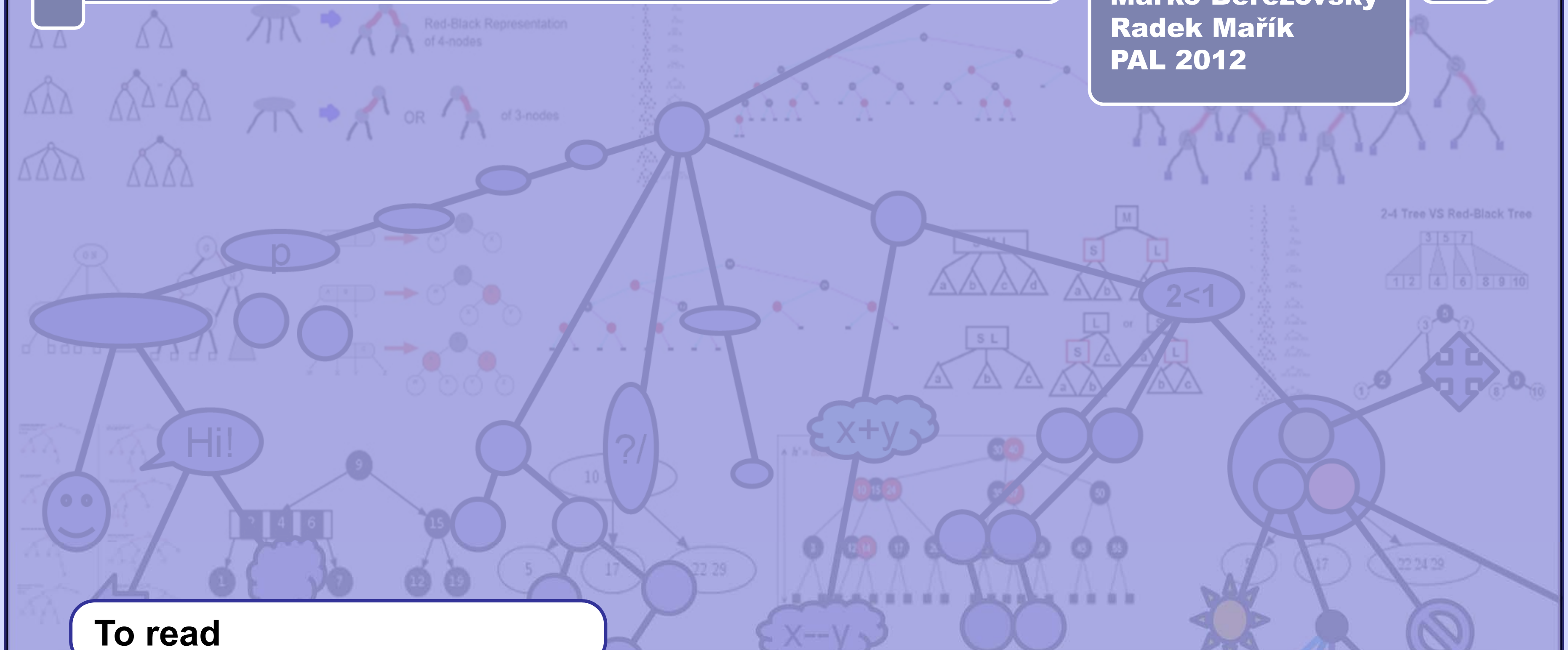
Splay tree, 2-3-4 tree

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.

[3] Ben Pfaff: Performance Analysis of BSTs in System Software, 2004, Stanford University, Department of Computer Science, http://benpfaff.org/papers/libavl.pdf

See also PAL webpage for references

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 are ***splayed*** (= moved by one or more rotations) to the root of the tree:

Find: Find the node like in a BST and then splay it to the root.

Insert: Insert the node like in a BST and then splay it to the root.

Delete: Splay the node to the root and then delete it like in a BST.

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

Splay tree

- A binary search tree.

- No additional tree shape description (no additional memory!) is used.

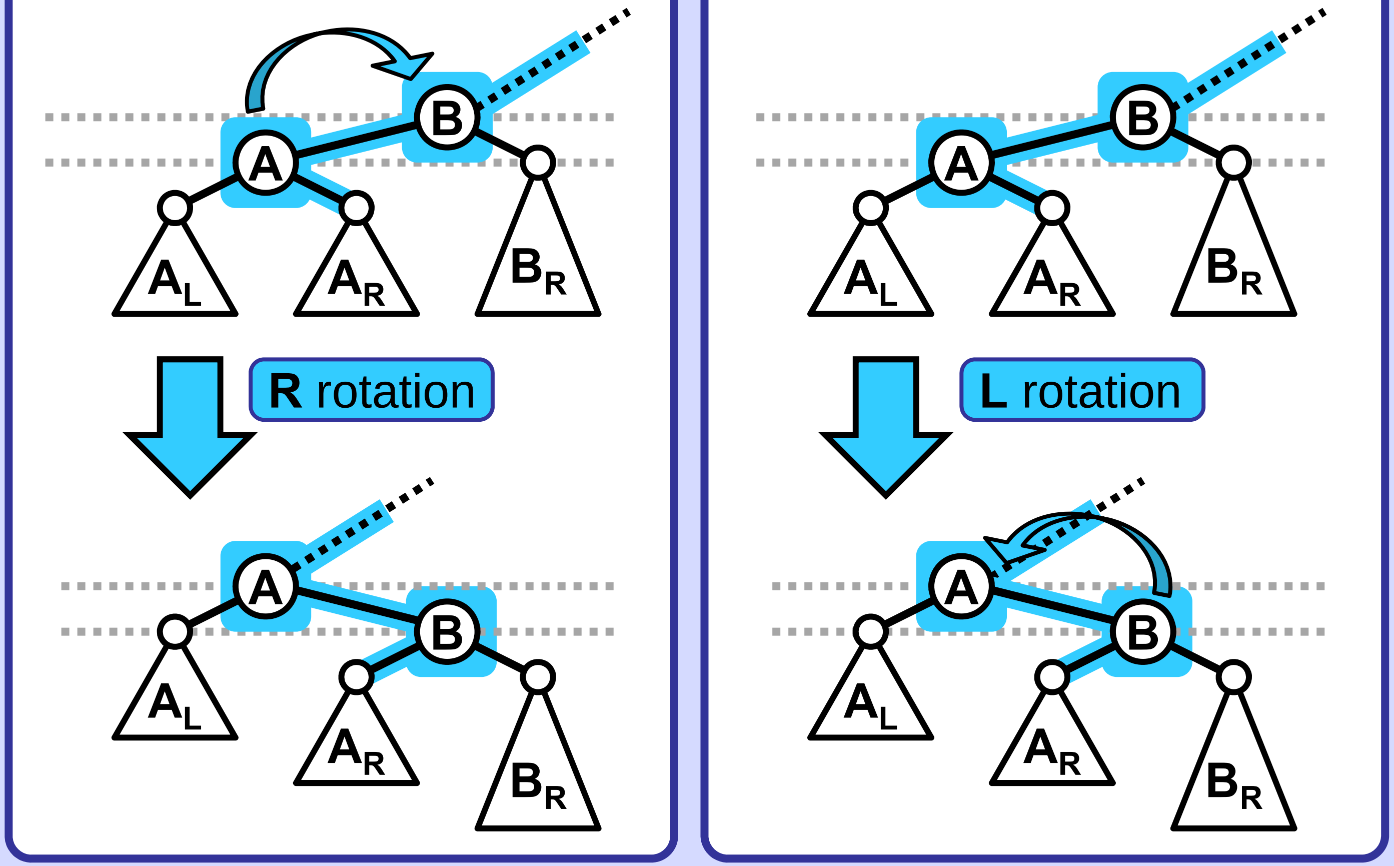
- Each node access or insertion *splays* that node to the root.

- Rotations are *zig, zig-zig* and *zig-zag,* based on BST single rotation.

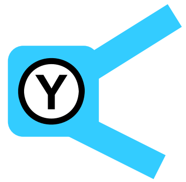
- All operations run times are $O(n)$, as the tree height can be $\Theta(n)$.

- Amortized run times of all operations are $O(\ln(n))$.

Zig rotation



Zig rotation is the same as a rotation (L or R) in AVL tree.

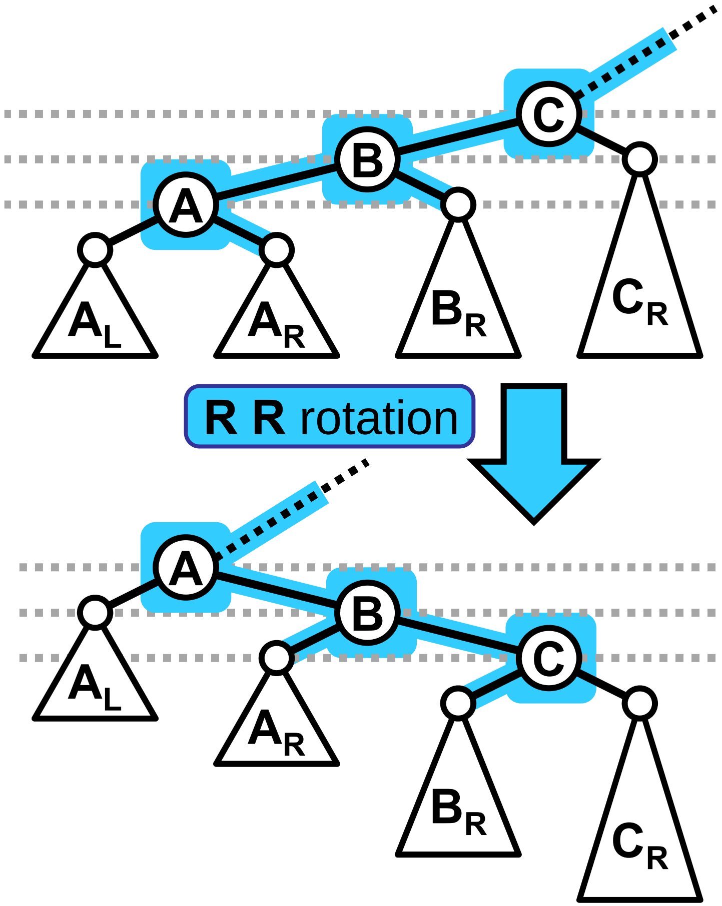


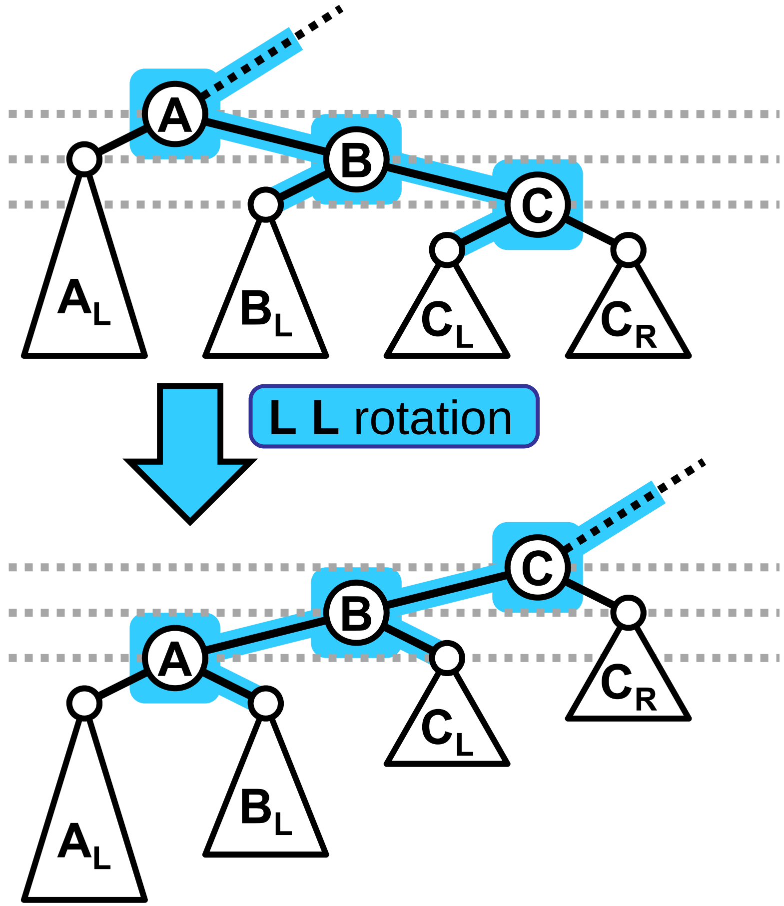
Afected nodes and edges

Note

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

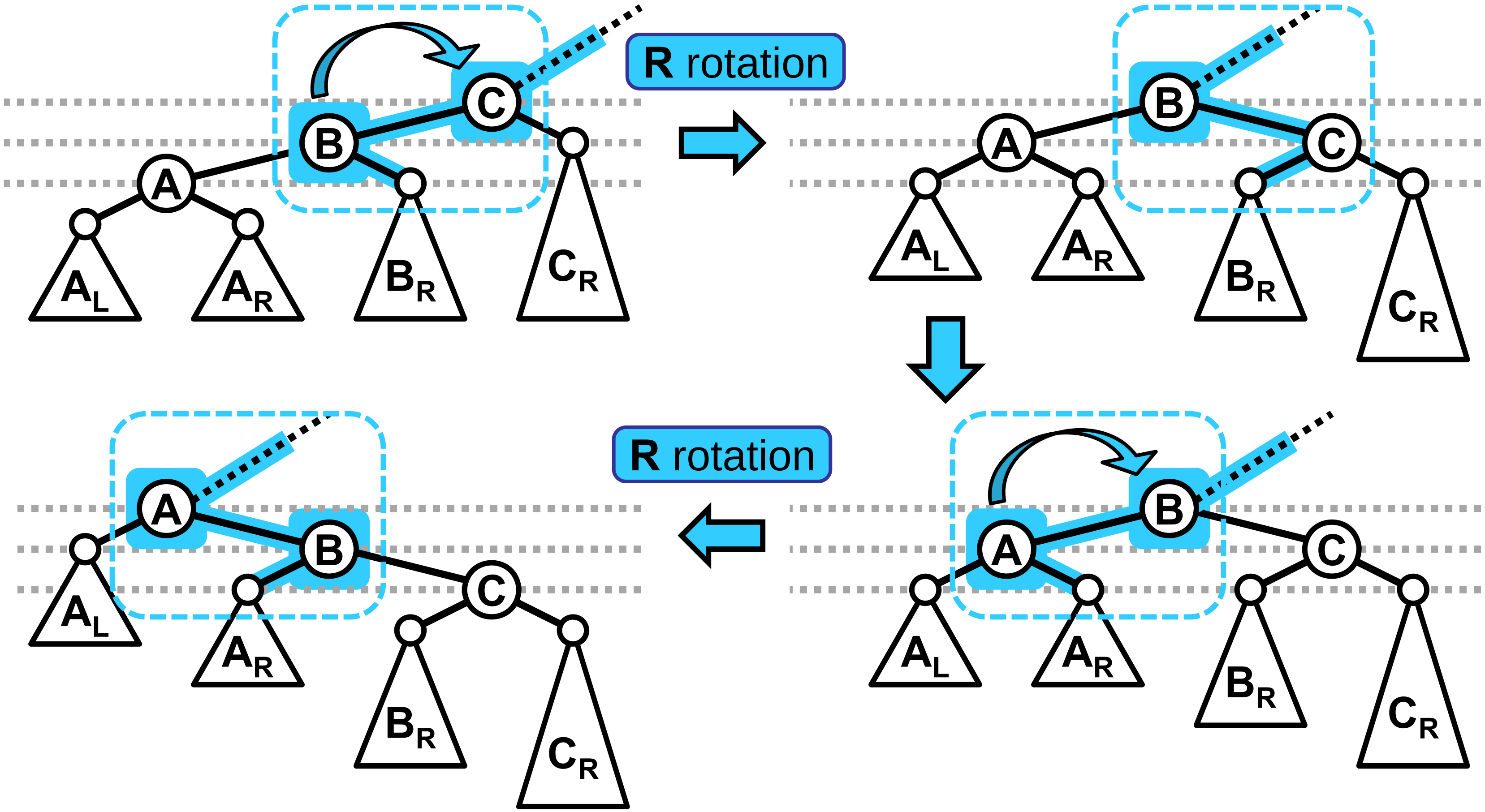
Zig - zig rotation





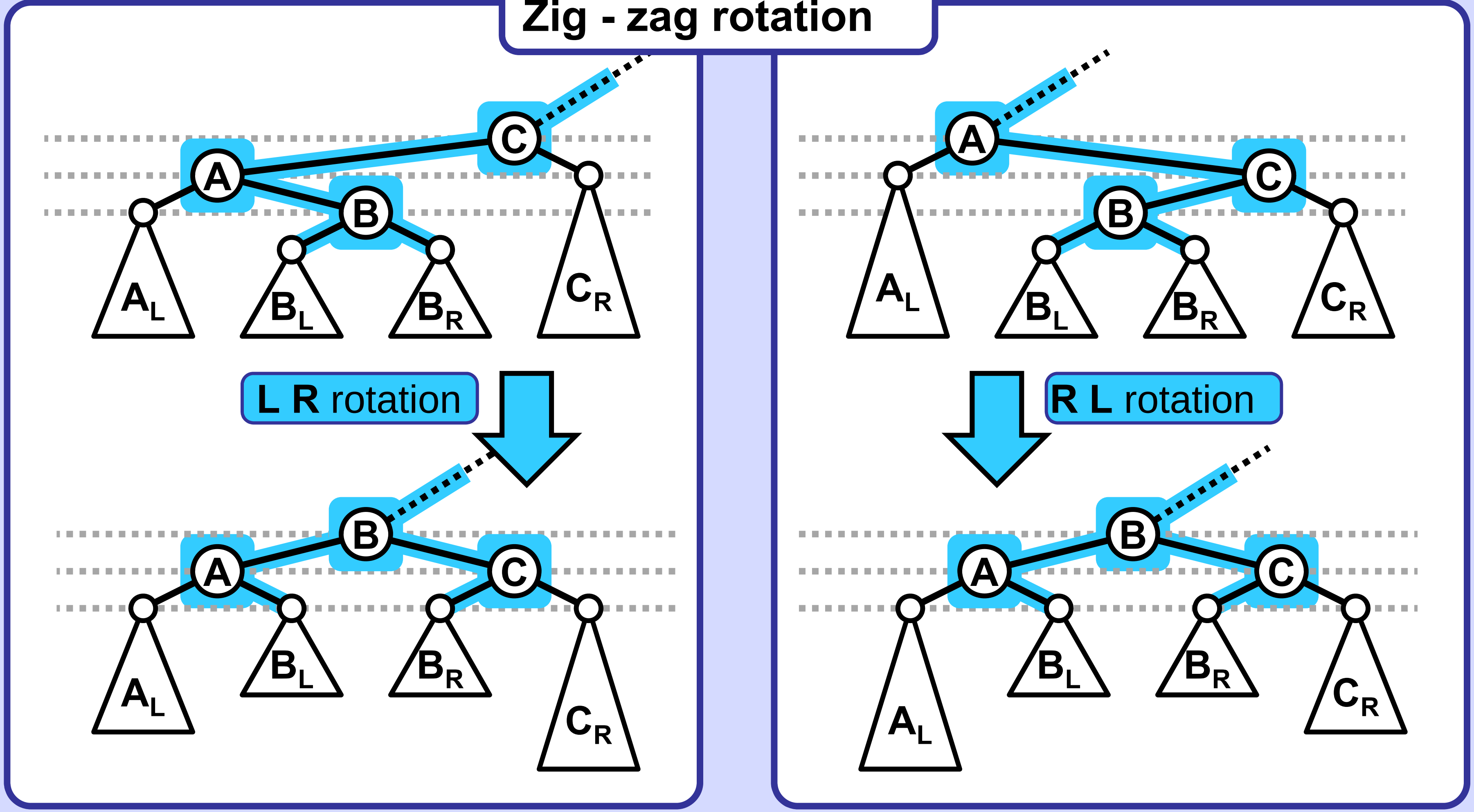
Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.

Zig - zig rotation



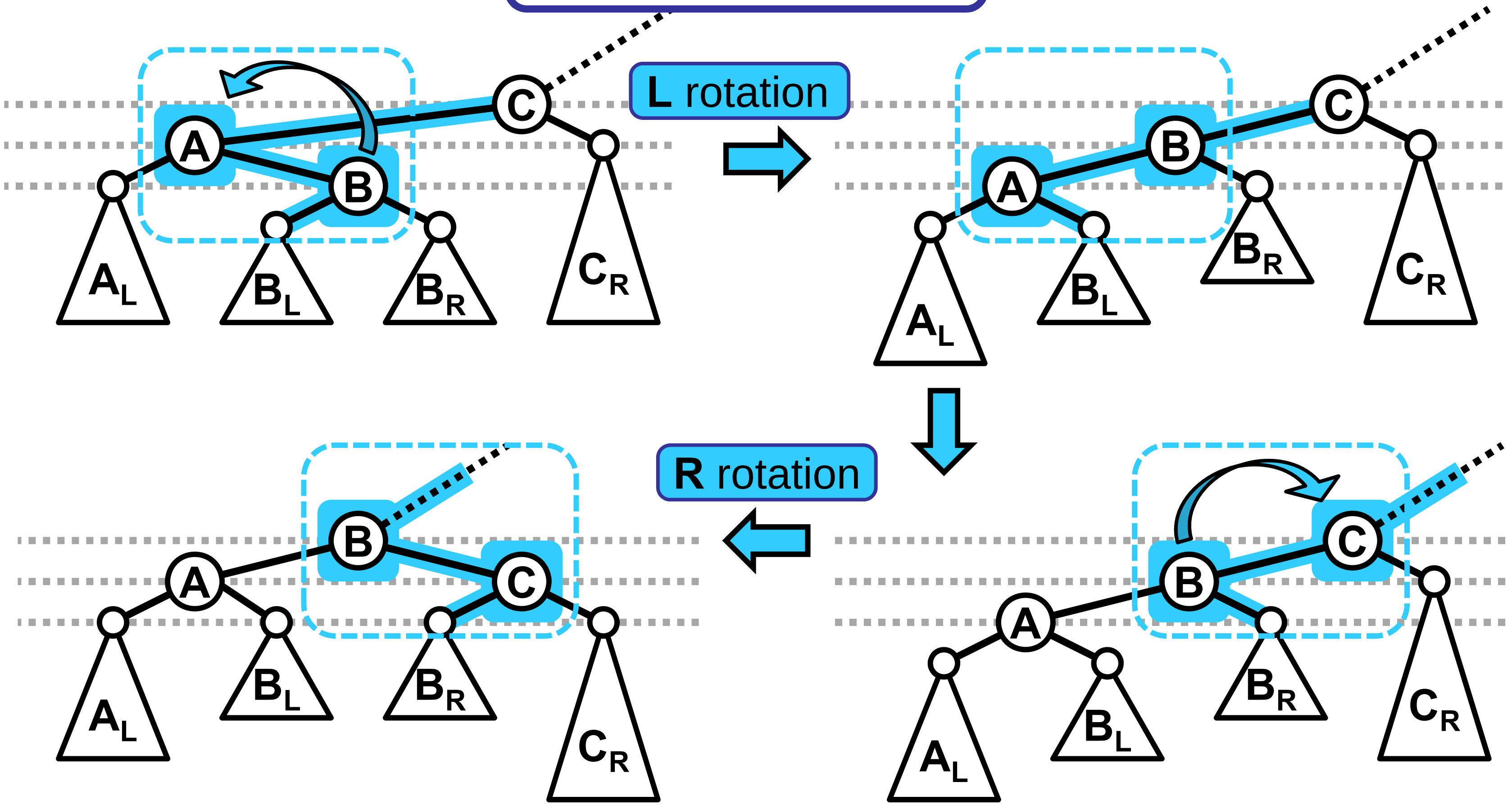
Both simple rotations are performed at the top of the current subtree, the splayed node (with key A) is not involved in the first rotation.

Zig - zag rotation



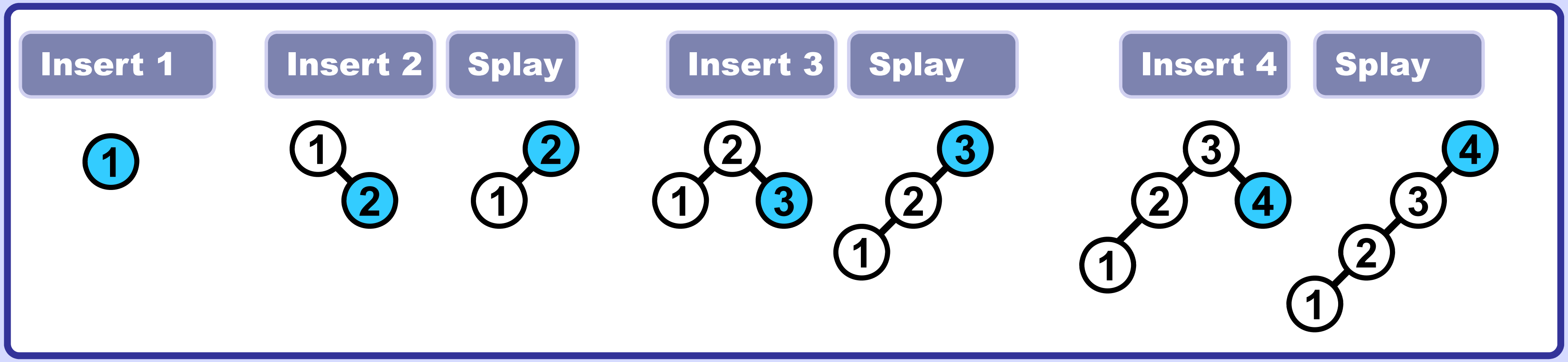
Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.

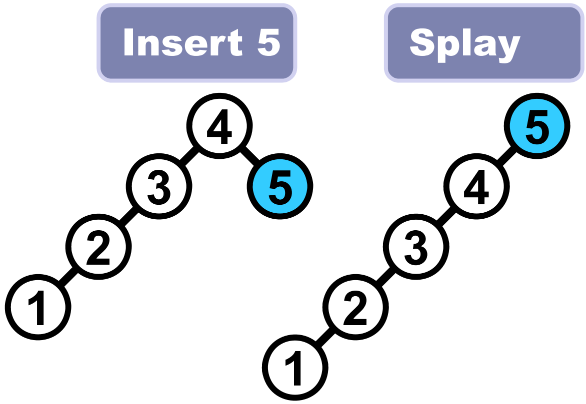
Zig - zag rotation



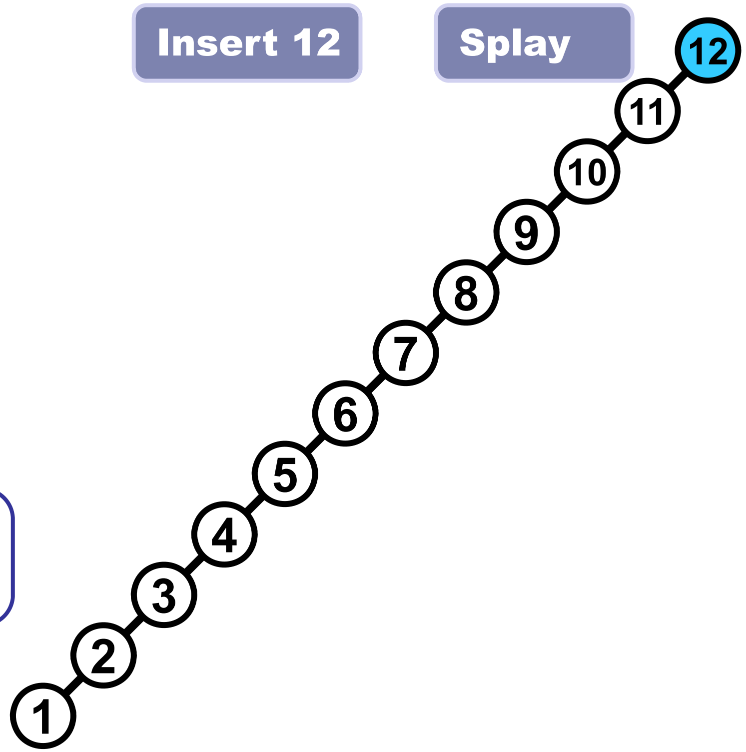
Note:

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





etc...



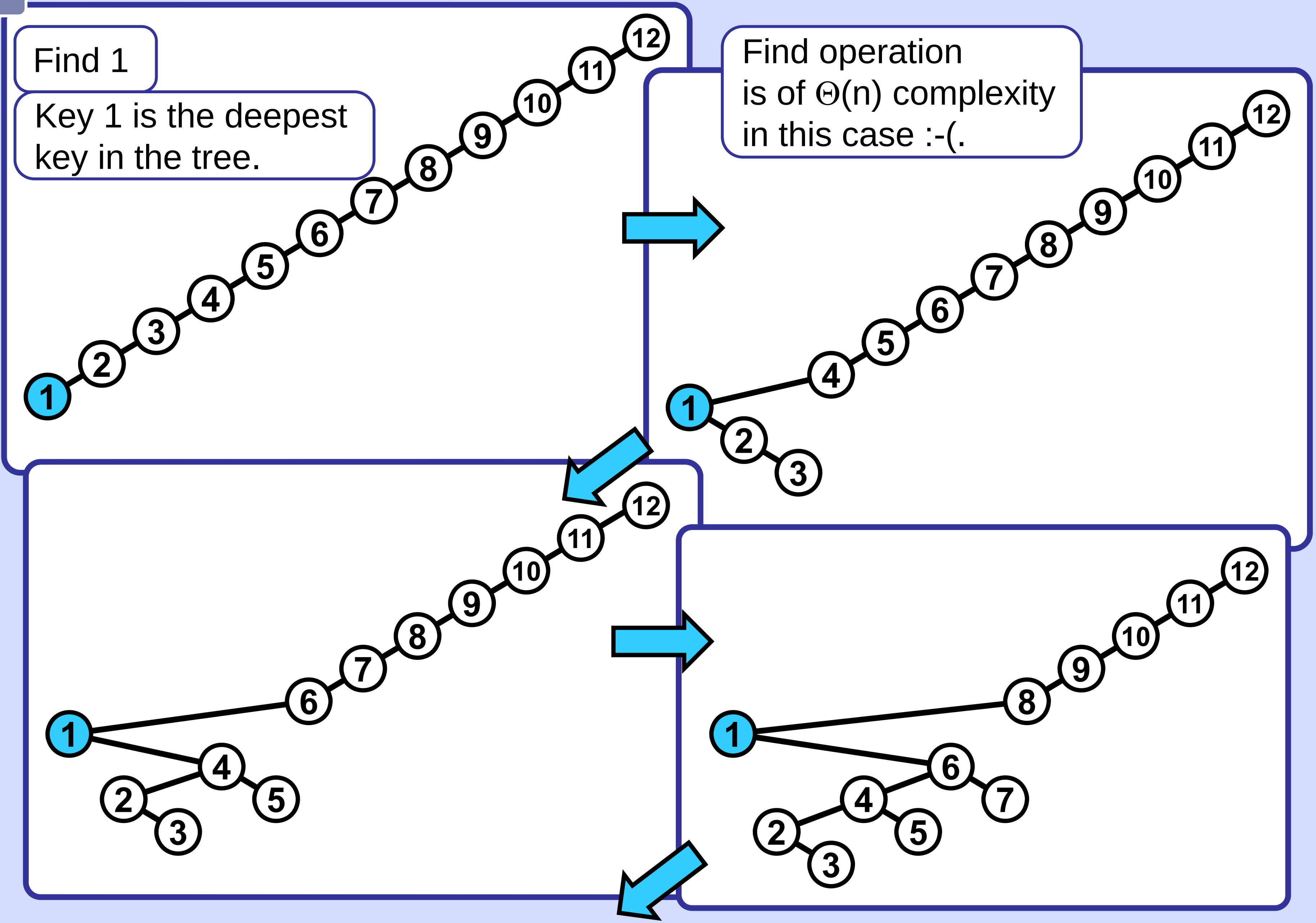
Note the extremely inefficient shape of the resulting tree.

Splay Tree - Find

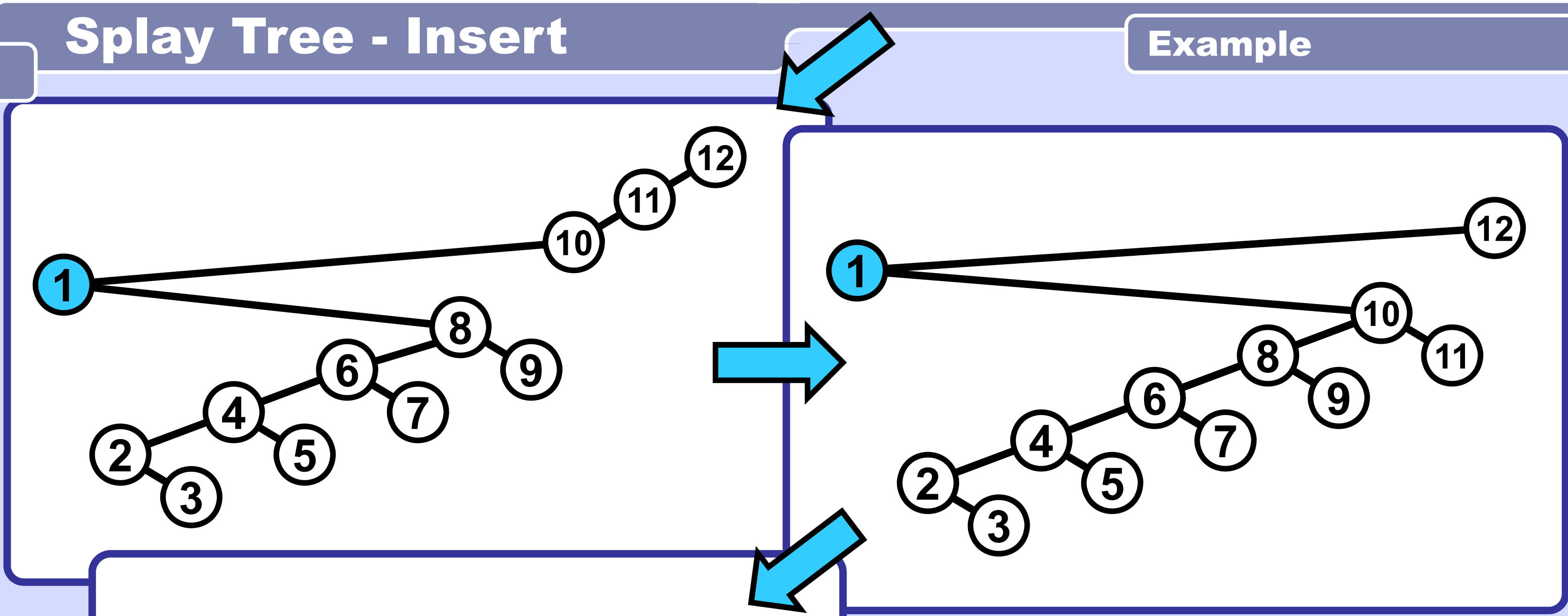
Find 1

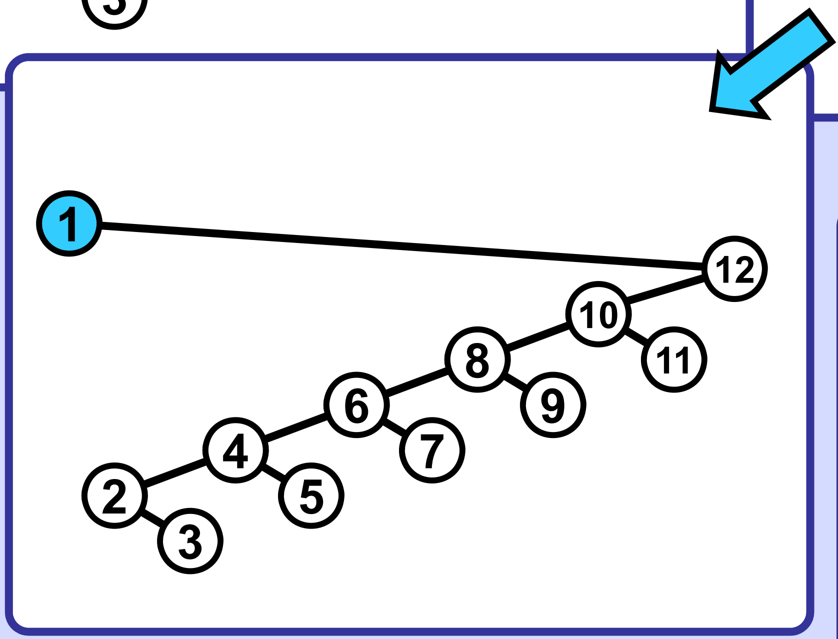
Key 1 is the deepest key in the tree.

Find operation is of Θ(n) complexity in this case :-(.

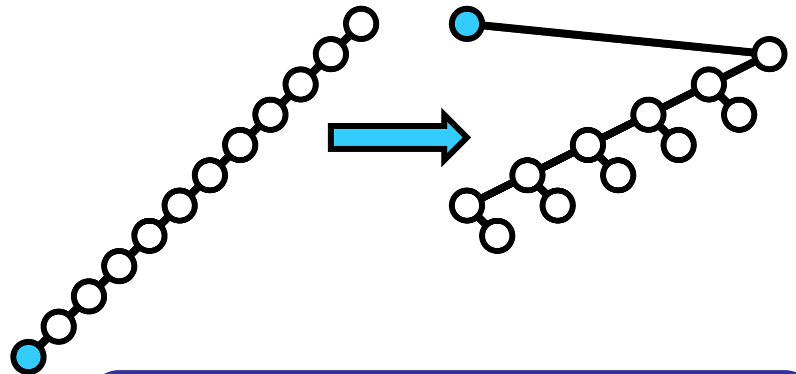


Splay Tree - Insert





Scheme - Result of the most unfavourable Find operation

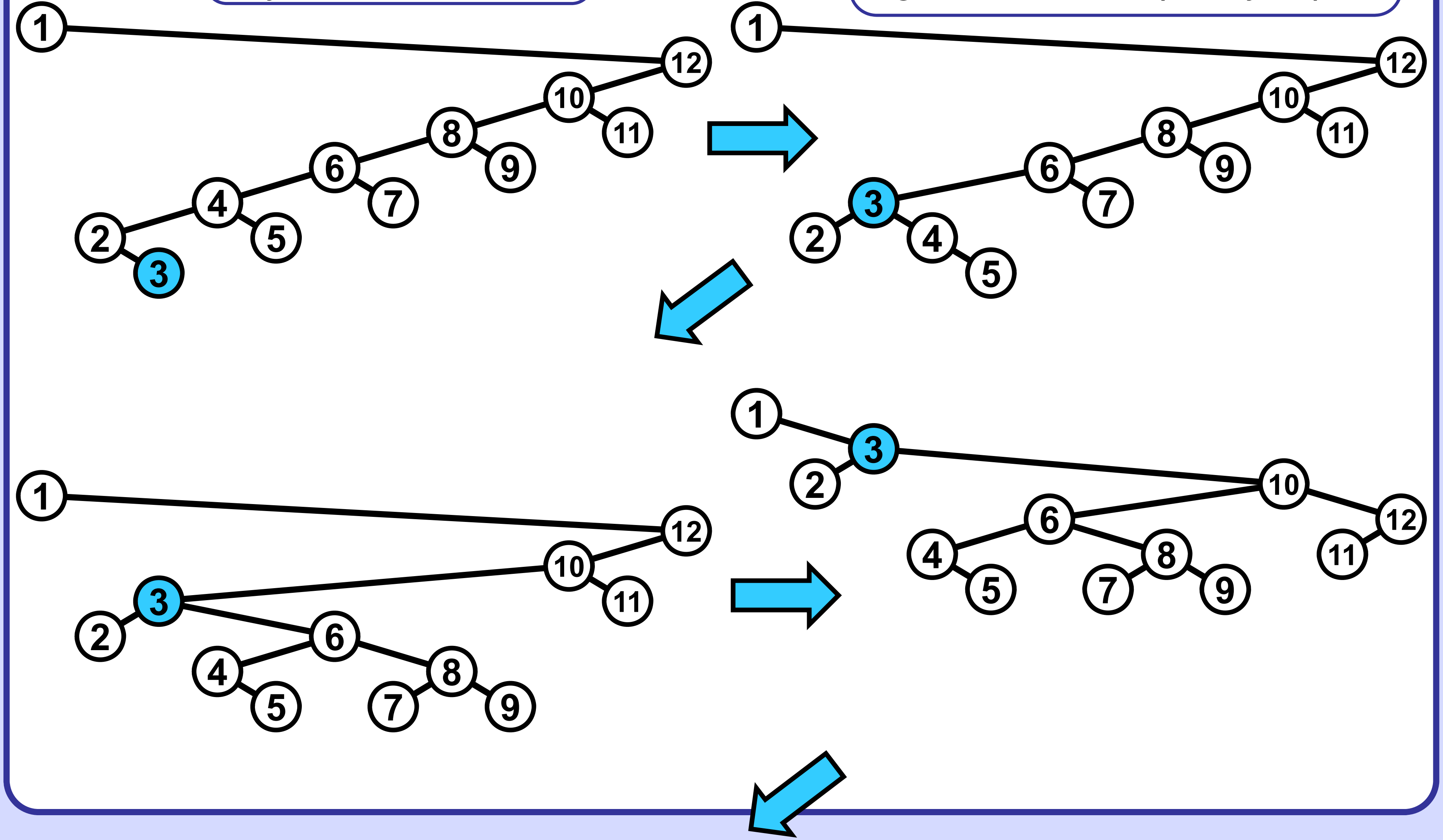


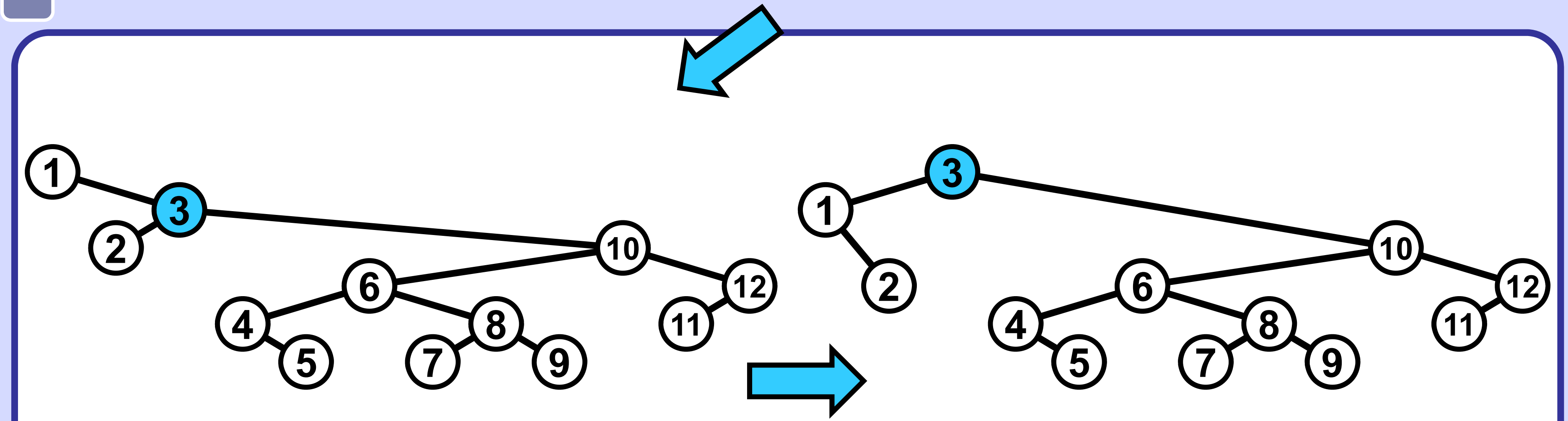
Note that the tree heigth is roughly halved. H $\rightarrow$ (H + 3) / 2

Find 3

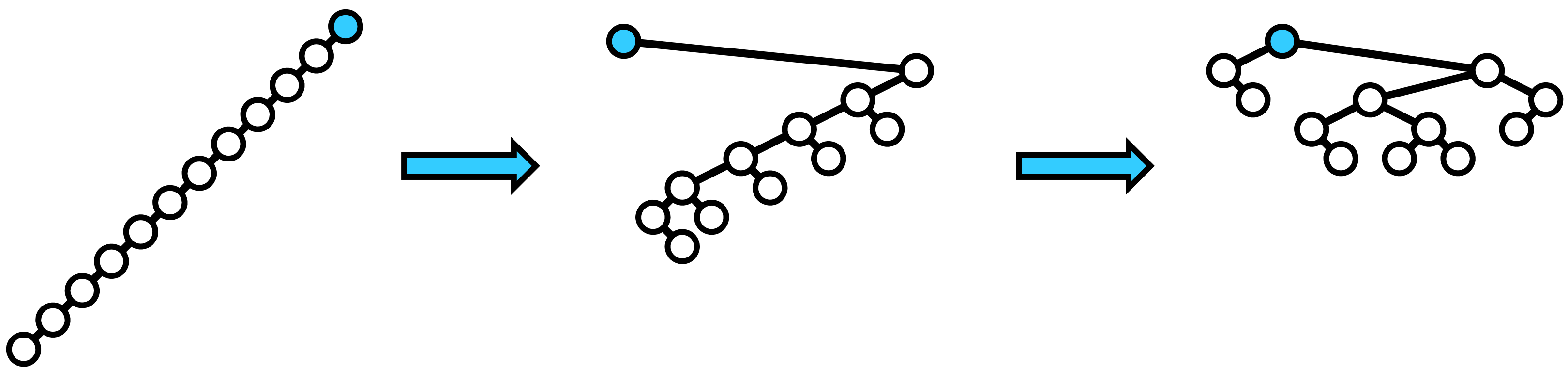
Key 3 is the deepest key in the tree.

The Find operation would be again of ~n complexity. :-(





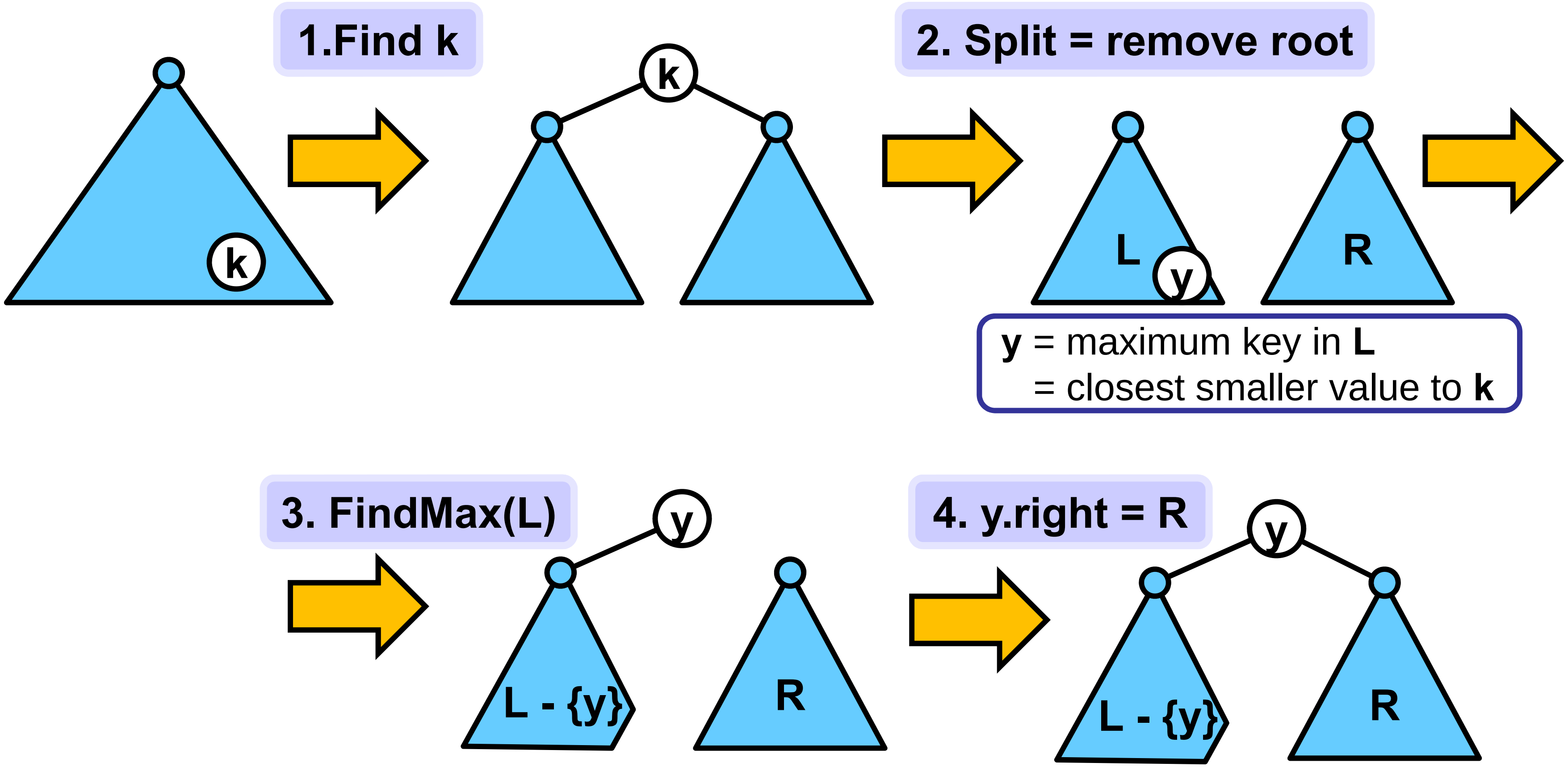
Scheme - Progress of the two most unfavourable Find operations.



Note the relatively favourable shape of the resulting tree.

Delete(k)

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 difficult to demonstrate the amortized logarithmic behaviour of splay trees using only small trees with few nodes.

The original ACM article [2] proves the *balance theorem*:
The run time of performing a sequence of $\boldsymbol{m}$ operations on a splay tree with $\boldsymbol{n}$ nodes is $\mathrm{O}(\ \boldsymbol{m}(1 + \ln(\boldsymbol{n})) + \boldsymbol{n} \ln(\boldsymbol{n})\ )$.

Therefore, the run time for a splay tree is comparable to any balanced tree assuming at least $\boldsymbol{n}$ 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 A 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.

Splay Tree - Performance

Advantages:

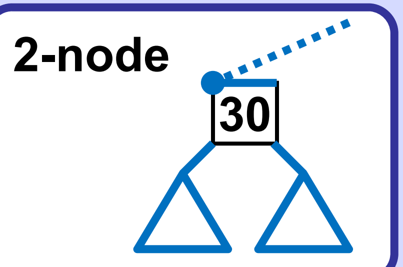
- The amortized run times are similar to that of AVL trees and red-black trees
- The implementation is easier
- No additional information (height/colour) is required

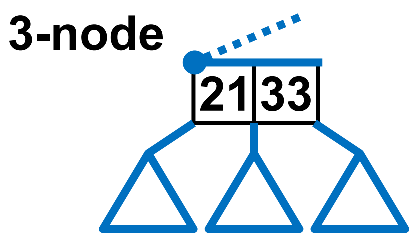
Disadvantages:

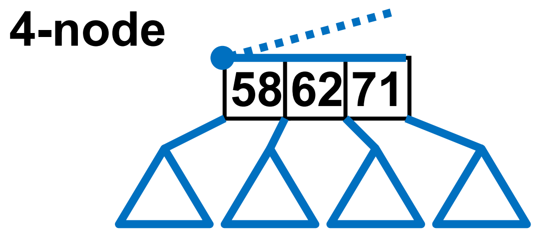
- The tree will change with read-only operations

A **2-3-4 search tree** is structurally a **B-tree of min degree 2 and max degree 4.**

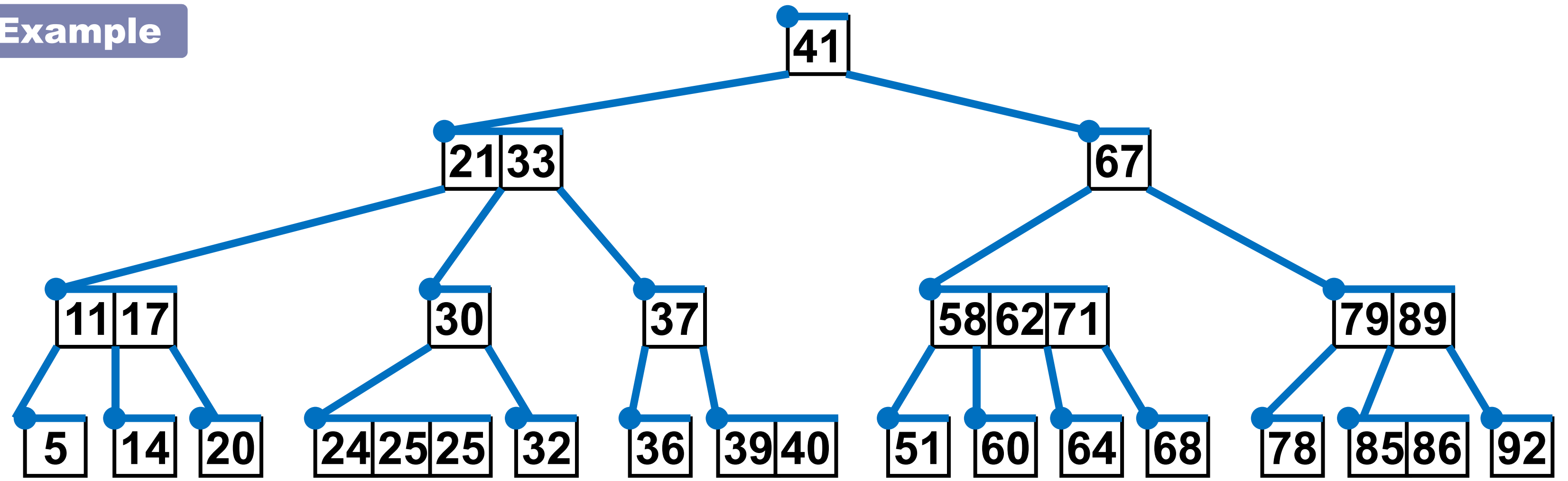
A node is a **2-node** or a **3-node** or a **4-node**.
If a node is not a leaf it has the corresponding number (2, 3, 4) of children.
All leaves are at the same distance from the root, the tree is **perfectly balanced.**







Example



Find: As in B-tree

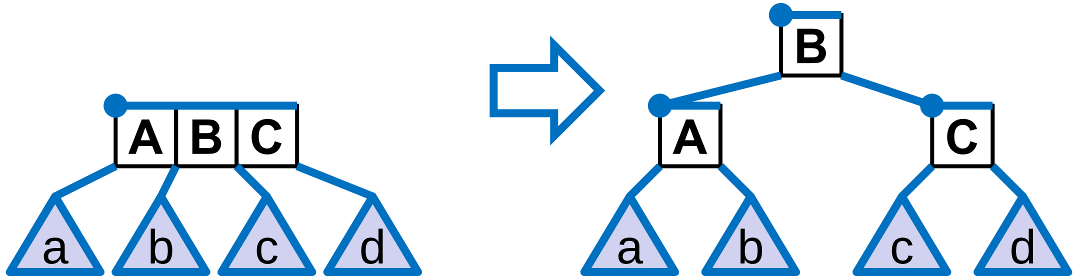
Insert: As in B-tree: Find the place for the inserted key x in a leaf and store it there. If necessary, split the leaf and store the median in the parent.

Splitting strategy
Additional insert rule (like single phase strategy in B-trees):
In our way down the tree, whenever we reach a **4-node** (including a leaf),
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 be necessary to split all the nodes going from inserted key back to the root. Such outcome is considered to be time consuming.

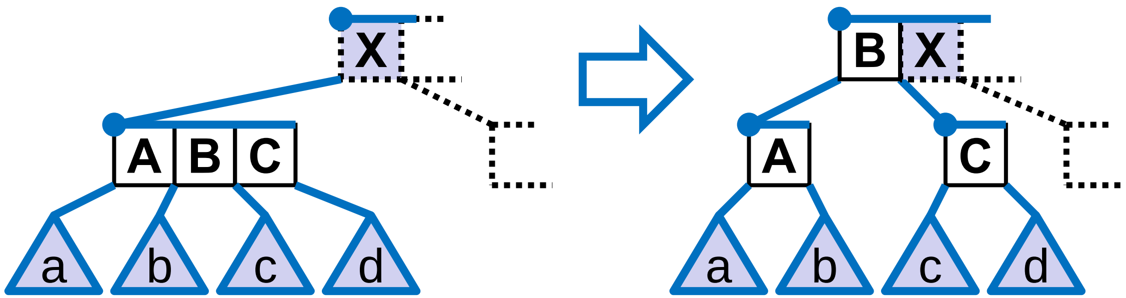
Splitting 4-nodes on the way down results in sparse occurence of 4-nodes in the tree, thus the nodes never have to be split recursively bottom-up.

Delete: As in B-tree

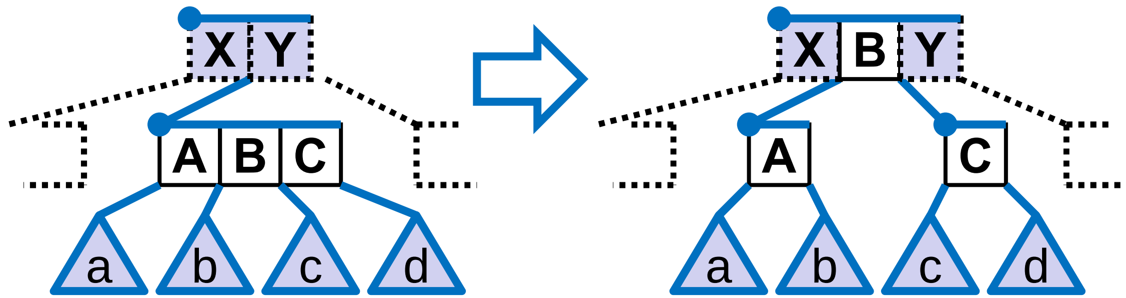
Split node is the root.
Only the root splitting
increases the tree height



Split node is the leftmost or the rightmost child of either a 2-node or a 3-node. (Only the leftmost case is shown, the righmost case is analogous)

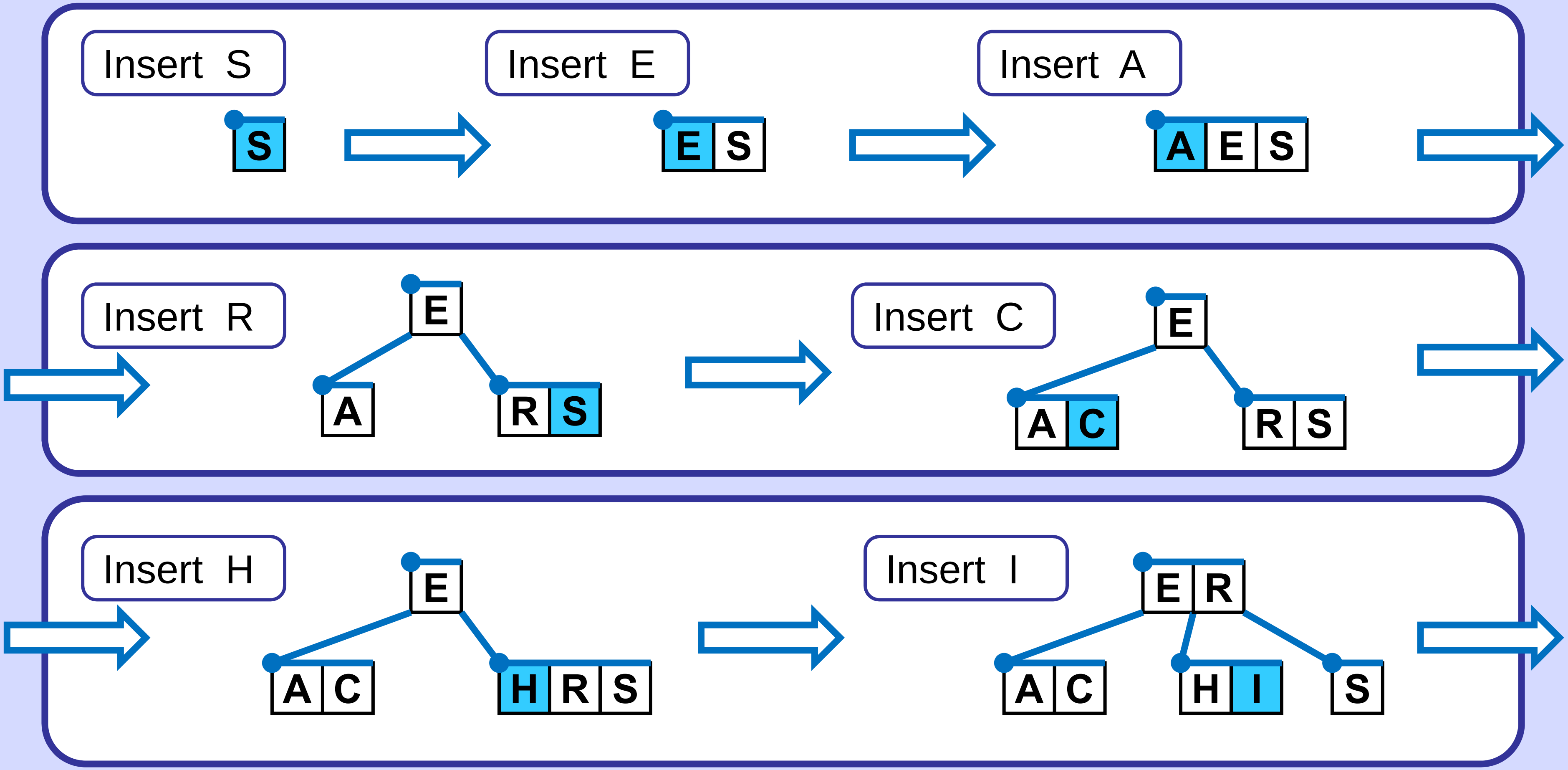


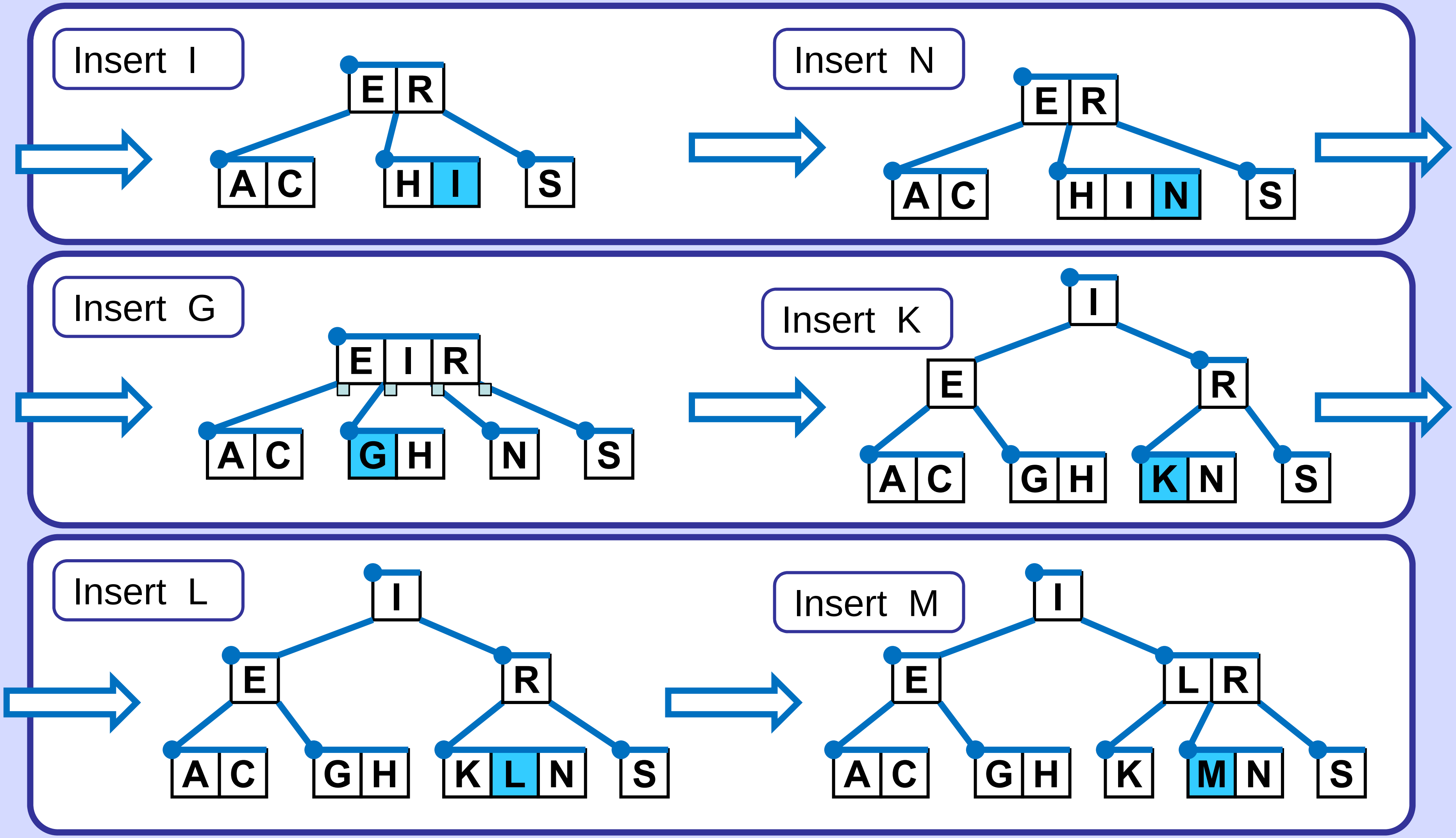
Split node is the middle child of a 3-node.



The node being split cannot be a child of a 4-node, due to the splitting strategy.

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



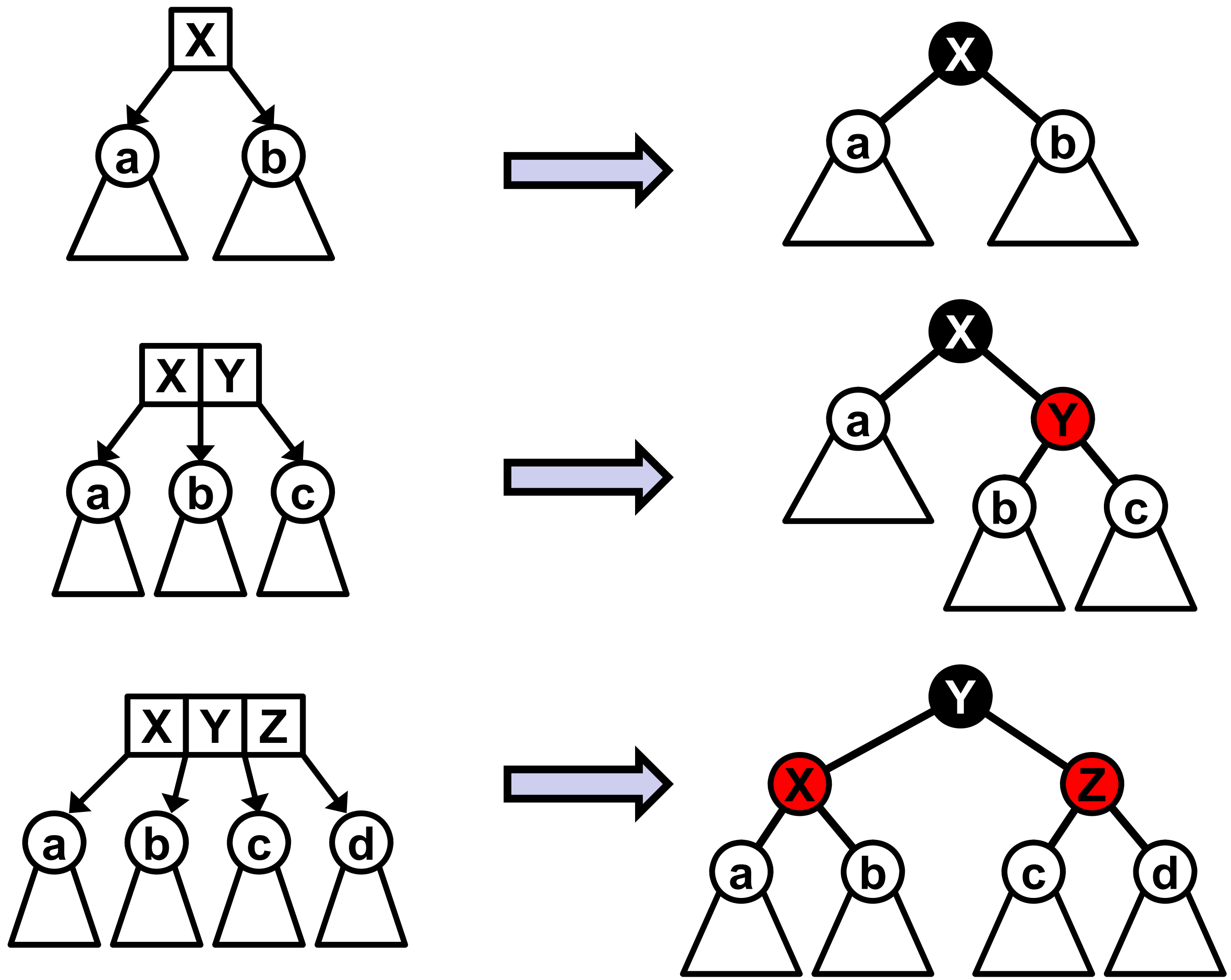


Note the seemingly unnecessary split of E,I,R 4-node during insertion of K.

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, 2004, [3]**

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 virtual memory areas (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

Compilation identifiers cross-references

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