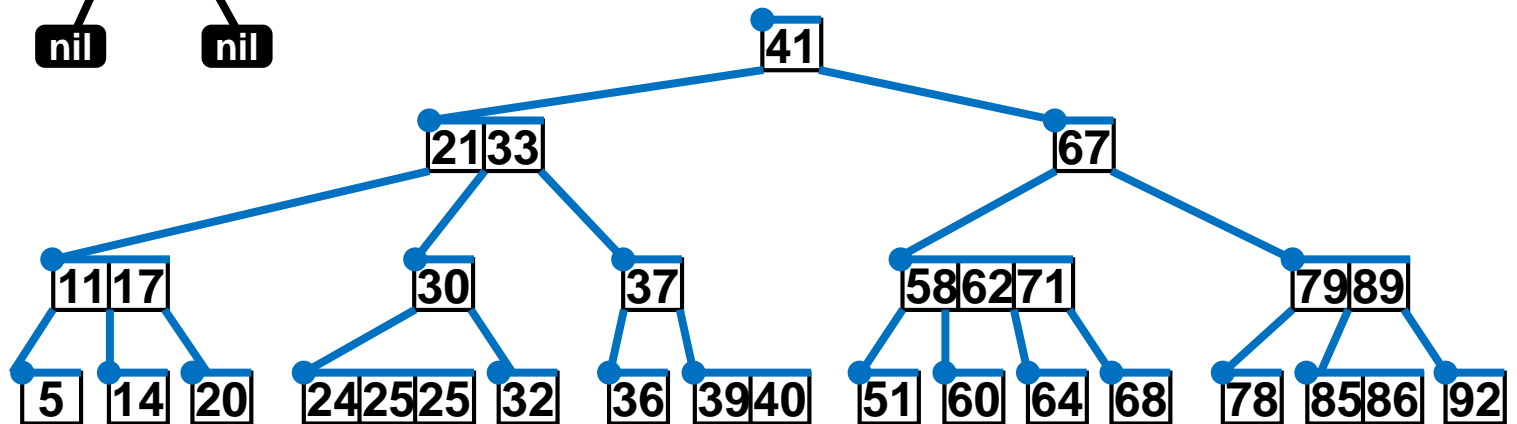
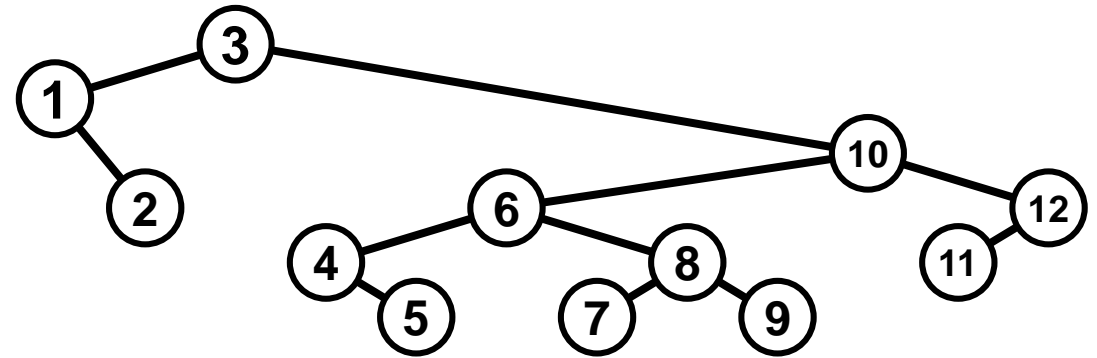
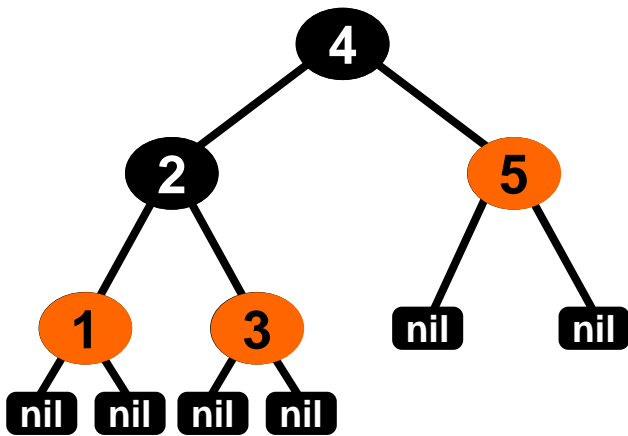


# Search trees

- Red-Black Tree
- Splay Tree
- 2-3-4 Tree



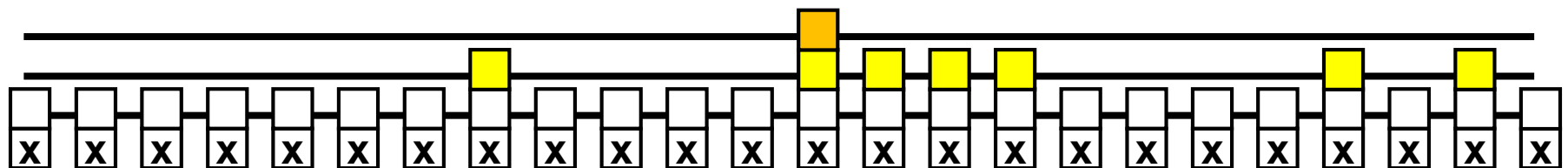
## More general randomness

Choose a fraction  $p$  between 0 and 1.

Rule: Fraction  $p$  of elements with level  $k$  pointers will have level  $k+1$  elements as well.

On average:  $(1-p)$  elements will be level 1 elements,  
 $(1-p) \cdot p$  elements will be level 2 elements,  
 $(1-p) \cdot p^2$  elements will be level 3 elements, etc.

This scheme corresponds to flipping a coin that has  $p$  chance of coming up heads,  $(1-p)$  chance of coming up tails.



Example of an experimental independent levels calculation with  $p = 0.33$ .

Element level (probability):

$$\begin{array}{ll}
 (1 - p) & \text{level 1} \\
 (1 - p) \cdot p & \text{level 2} \\
 \dots & \\
 (1 - p) \cdot p^{k-1} & \text{level k} \\
 \dots &
 \end{array}$$

---


$$\sum_{k=1}^{\infty} (1 - p) \cdot p^{k-1} = \frac{1-p}{1-p} = 1$$

Expected number of pointers per element:

$$\sum_{k=1}^{\infty} k \cdot (1 - p) \cdot p^{k-1} = (1 - p) \cdot \sum_{k=0}^{\infty} (k + 1) \cdot p^k = \frac{1 - p}{(1 - p)^2} = \frac{1}{1 - p}$$

This scheme corresponds to flipping a coin that has  $p$  chance of coming up heads,  $(1-p)$  chance of coming up tails.

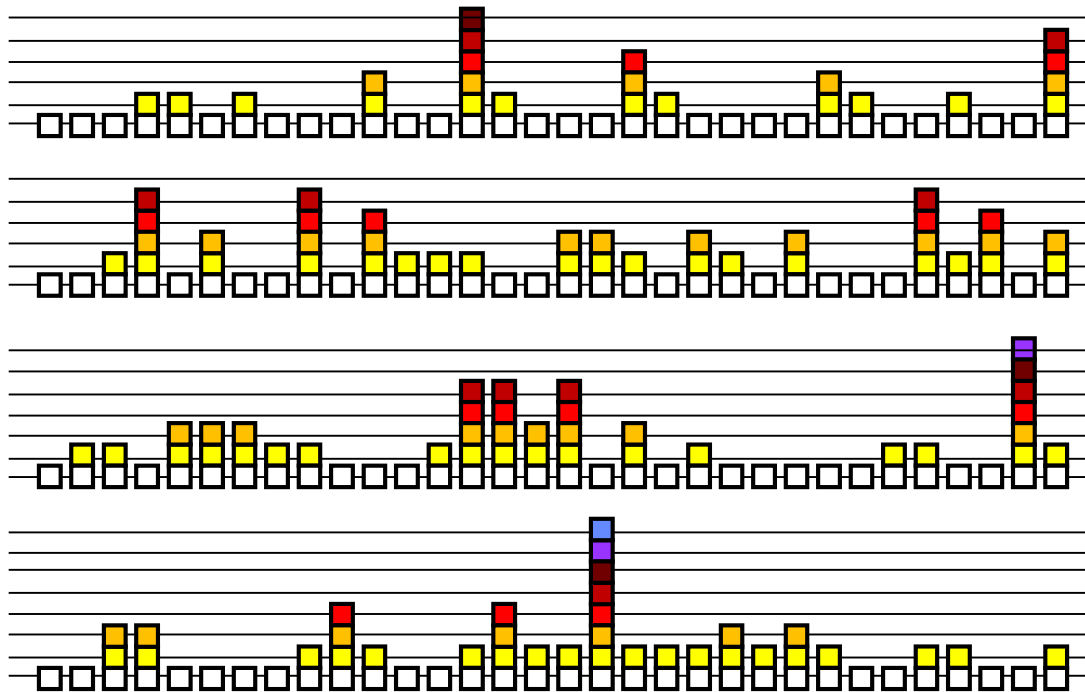
$$\sum_{k=0}^{\infty} (k + 1) \cdot x^k = \frac{1}{(1 - x)^2}$$

$$\frac{1}{1 - \frac{1}{2}} = 2, \quad \frac{1}{1 - \frac{1}{4}} = 1.33$$

## Experiment with Lehmer generator

$$X_{n+1} = 16807 X_n \bmod 2^{31}-1$$

seed = 23021905 // birth date of Derrick Henry Lehmer



Coin flipping:

$(X_n \gg 16) \& 1$

Head = 1

128 nodes

	Level	1	2	3	4	5	6	7	8	9	...
Number of nodes	Expected	64	32	16	8	4	2	1	1/2	1/4	...
	Actual	60	36	17	5	7	1	1	1	0	...

Data structures and algorithms

# Red-Black Trees

Petr Felkel

Exploited in Advanced Algorithms 2012-2020

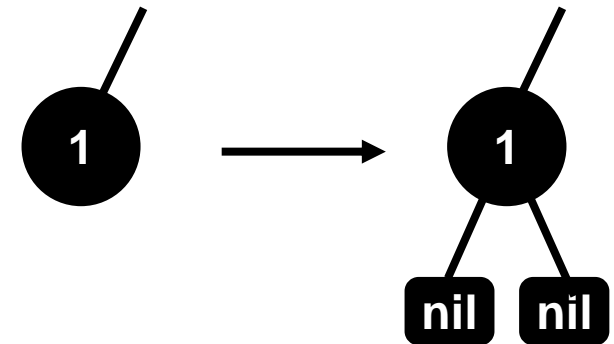
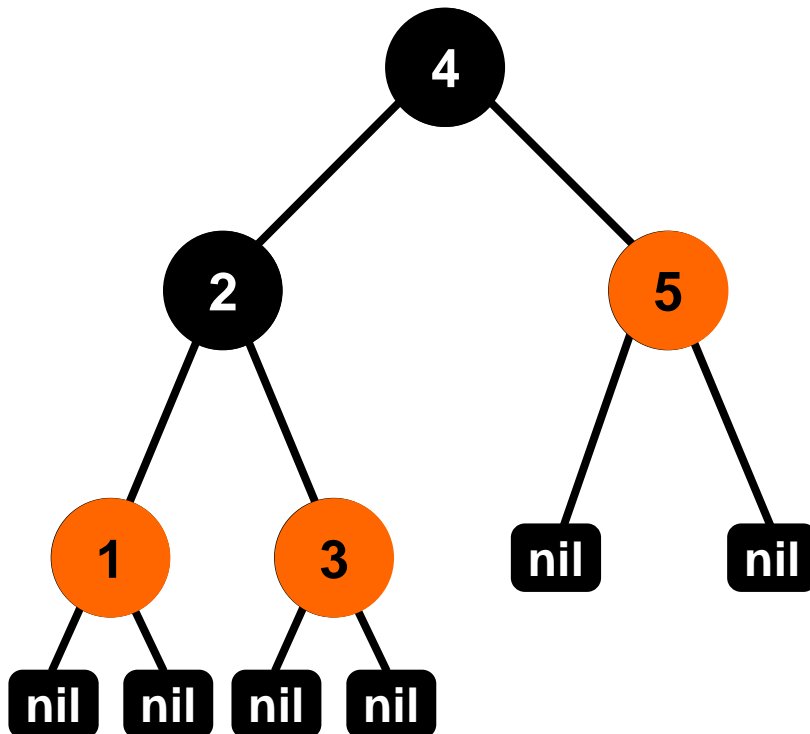
# Red-Black tree

Approximately balanced BST

$$h_{RB} \leq 2 \cdot h_{BST} \quad (\text{height} \leq 2 \times \text{height of a balanced tree})$$

Additional bit for COLOR = {red | black}

nil (non-existent child) = pointer to **nil** node



leaf → inner node

# Red-Black tree

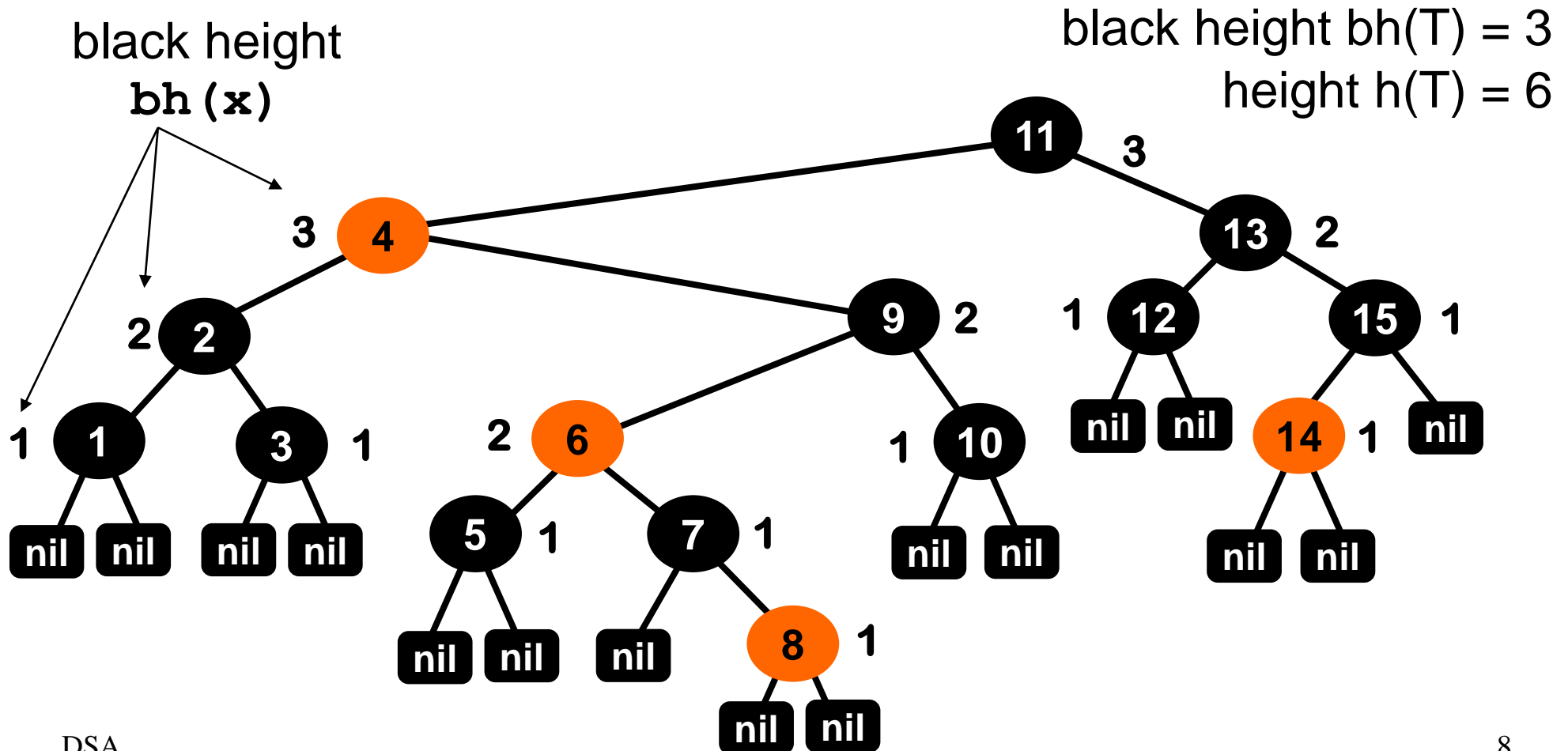
A binary search tree is a **red-black** tree if:

1. Every node is either **red** or **black**.
2. Every leaf (nil) is **black**.
3. If a node is **red**, then both its children are **black**.
4. Every simple path from a node to a descendant leaf contains the same number of **black** nodes.
5. Root is **black**.

**Black-height  $bh(x)$**  of a node  $x$  is the number of **black** nodes on any path from  $x$  to a leaf, not counting  $x$

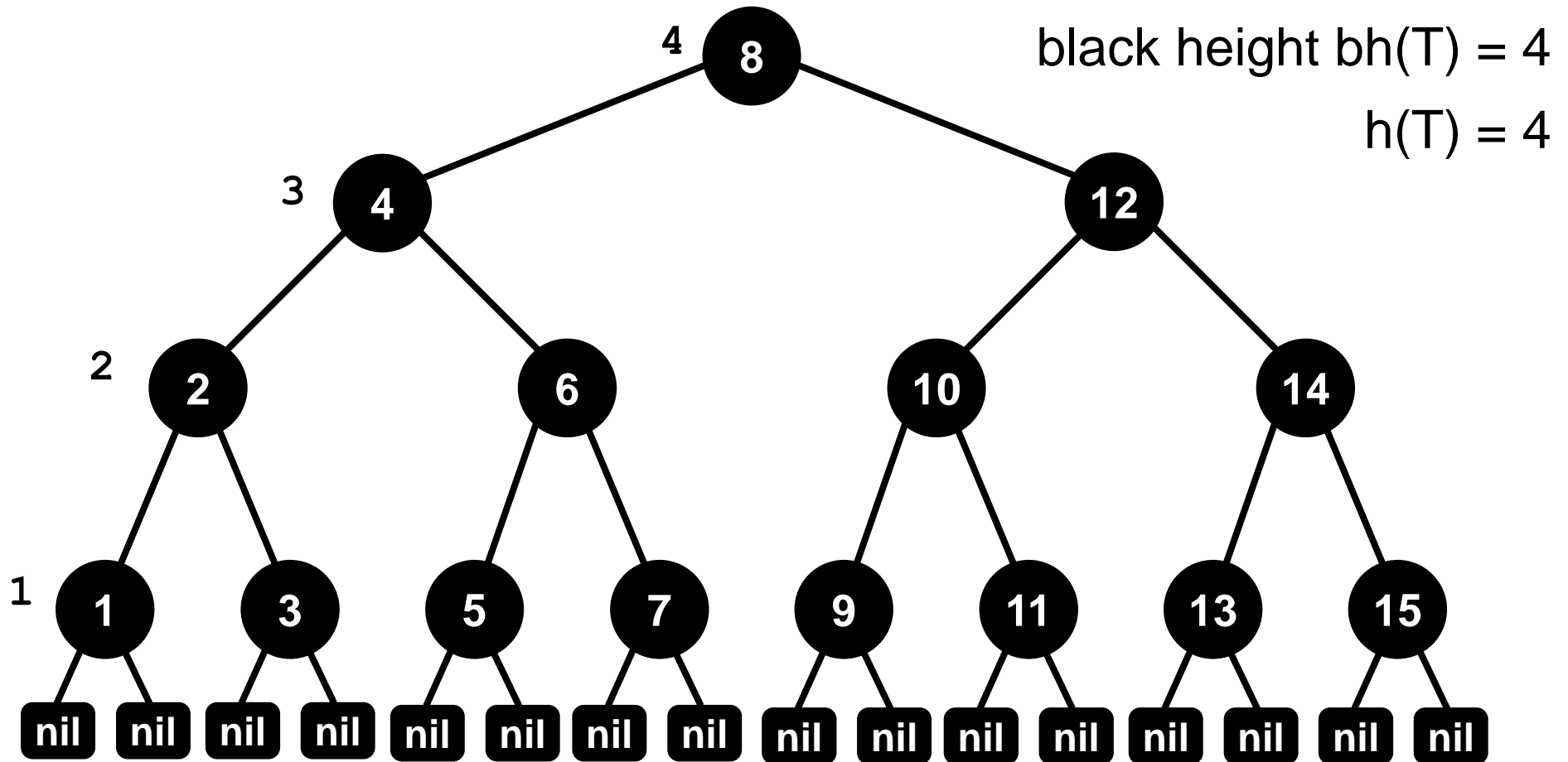
# Red-Black tree

Black-height  $bh(x)$  of a node  $x$  is the number of **black** nodes on any path from  $x$  to a leaf, not counting  $x$ .

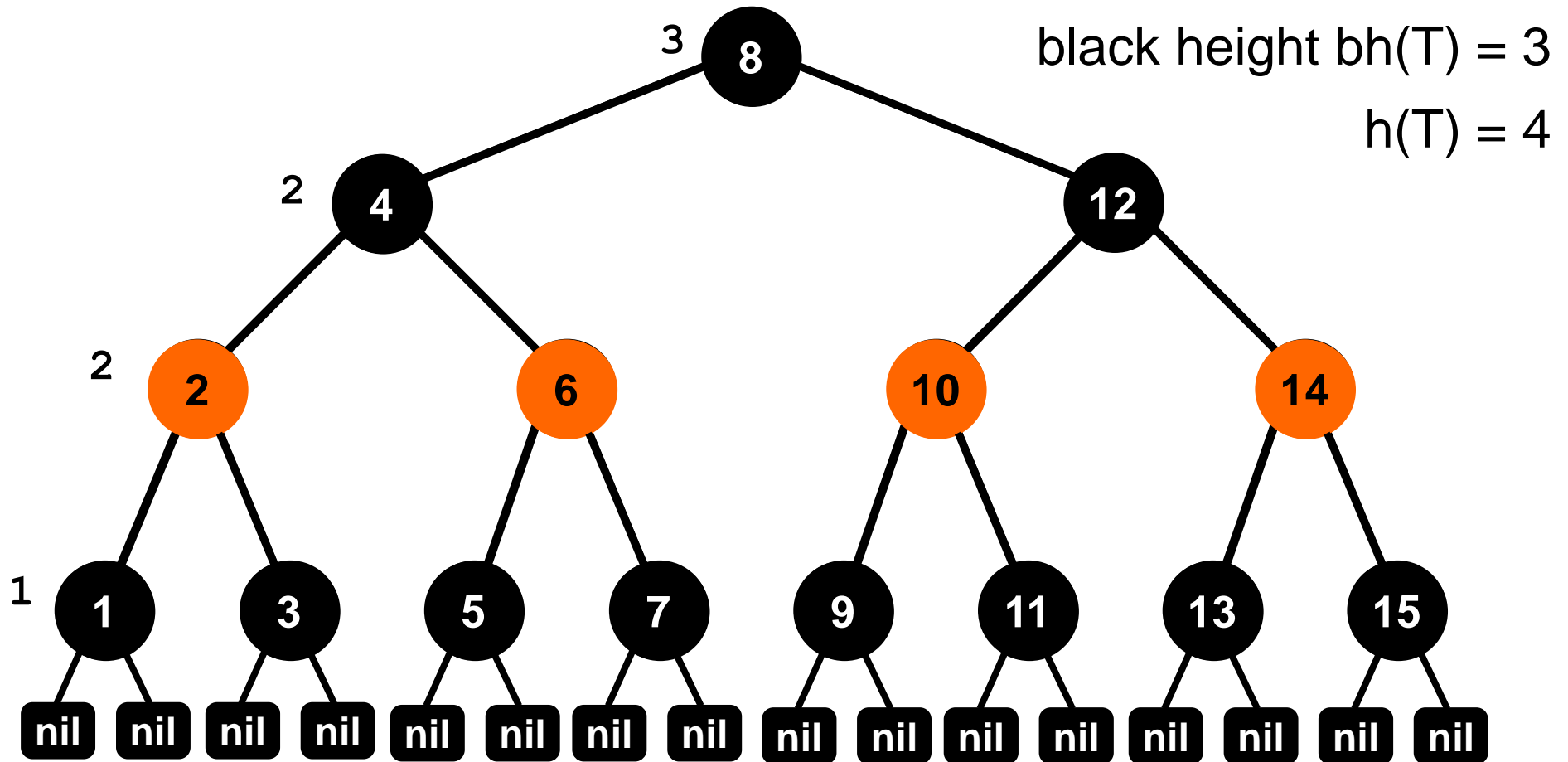




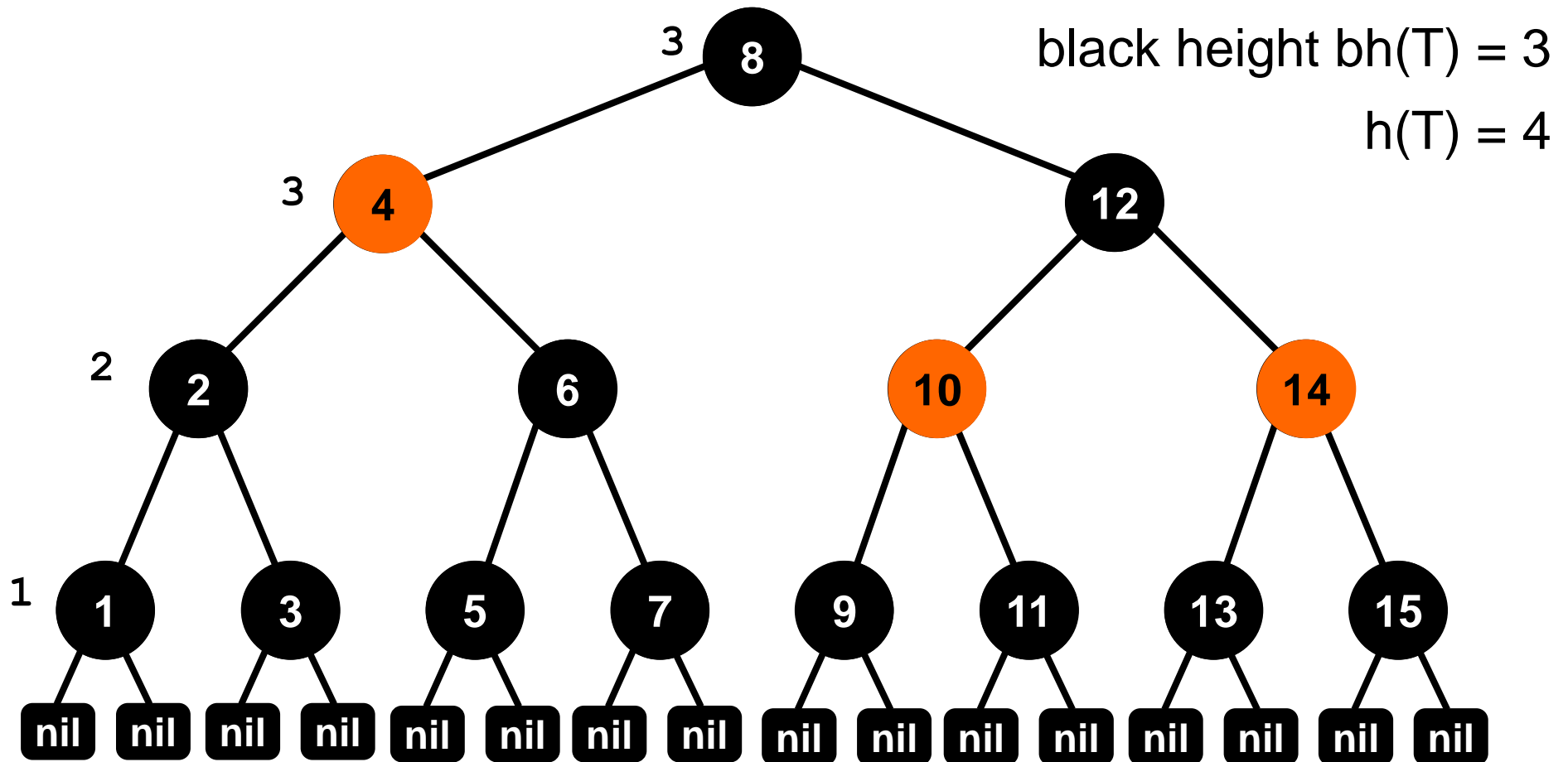
# Binary Search Tree -> RB Tree



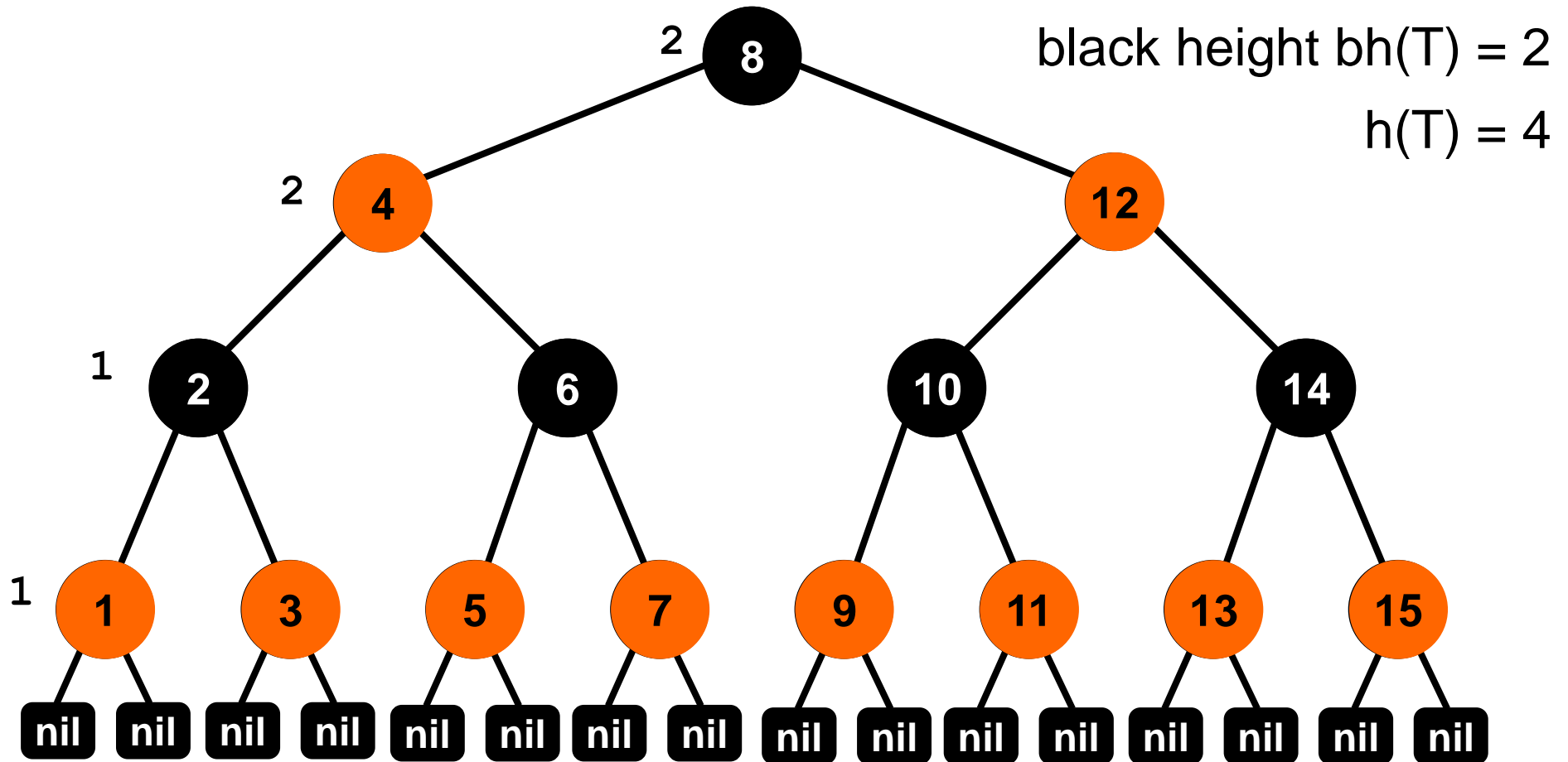
# Binary Search Tree -> RB Tree



# Binary Search Tree -> RB Tree



# Binary Search Tree -> RB Tree



# Red-Black tree

**Black-height  $bh(x)$**  of a node  $x$

- is the number of black nodes on any path from  $x$  to a leaf, not counting  $x$
- is equal for all paths from  $x$  to a leaf
- For given  $h$  is  $bh(x)$  in the range from  $h/2$  to  $h$ 
  - if  $1/2$  of nodes red  $\Rightarrow bh(x) \approx 1/2 h(x), h(x) \approx 2 \lg(n+1)$
  - if all nodes black  $\Rightarrow bh(x) = h(x) = \lg(n+1)$

**Height  $h(x)$**  of a RB-tree rooted in node  $x$

- is at maximum twice of the optimal height of a balanced tree
- $h \leq 2\lg(n+1)$  ....  $h \in \Theta(\lg(n))$

# RB-tree height proof [Cormen, p.264]

A red-black tree with  $n$  internal nodes has height  $h$  at most  $2\lg(n+1)$

Proof 1. Show that **subtree starting at  $x$  contains at least  $2^{\text{bh}(x)}-1$  internal nodes.**

By induction on height of  $x$ :

- I. If  $x$  is a *leaf*, then  $\text{bh}(x) = 0$ ,  $2^{\text{bh}(x)}-1 = 0$  internal nodes //... nil node
- II. Consider  $x$  with height  $h$  and two children (with height at most  $h-1$ )
  - $x$ 's children black-height is either  $\text{bh}(x) - 1$  or  $\text{bh}(x)$  //  $x$  is black or red
  - Ind. hypothesis:  $x$ 's children subtree has at least  $2^{\text{bh}(x)-1} - 1$  internal nodes
  - So subtree rooted at  $x$  contains at least  $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$  internal nodes  $\Rightarrow$  proved

Proof 2. Let  $h =$  height of the tree rooted at  $x$

- min  $\frac{1}{2}$  nodes are black on any path to leaf  $\Rightarrow \text{bh}(x) \geq h / 2$
- Thus,  $n \geq 2^{h/2} - 1 \Leftrightarrow n + 1 \geq 2^{h/2} \Leftrightarrow \lg(n+1) \geq h / 2$
- $h \leq 2\lg(n+1)$

# RB-tree Search

Search is performed as in simple BST, node colors do not influence the search.

Search in R-B tree with  $N$  nodes takes

1. In general -- at most  $2 \cdot \lg(N+1)$  key comparisons.
2. In best case when keys are generated randomly and uniformly -- cca  $1.002 \cdot \lg(N)$  key comparisons,  
very close to the theoretical minimum.

# Inserting in Red-Black Tree

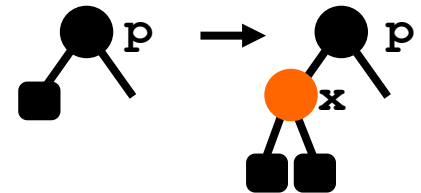
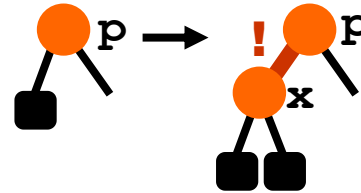
Color new node **x** **Red**

Insert it as in the standard BST



If parent **p** is **Black**, stop. Tree is a Red-Black tree.

If parent **p** is **Red** (3+3 cases)...



resp.

While **x** is not root and parent is Red

if **x**'s *uncle* is **Red** then case 1

// propagate red up

else { if **x** is *Right child* then case 2

// double rotation

case 3 }

// single rotation

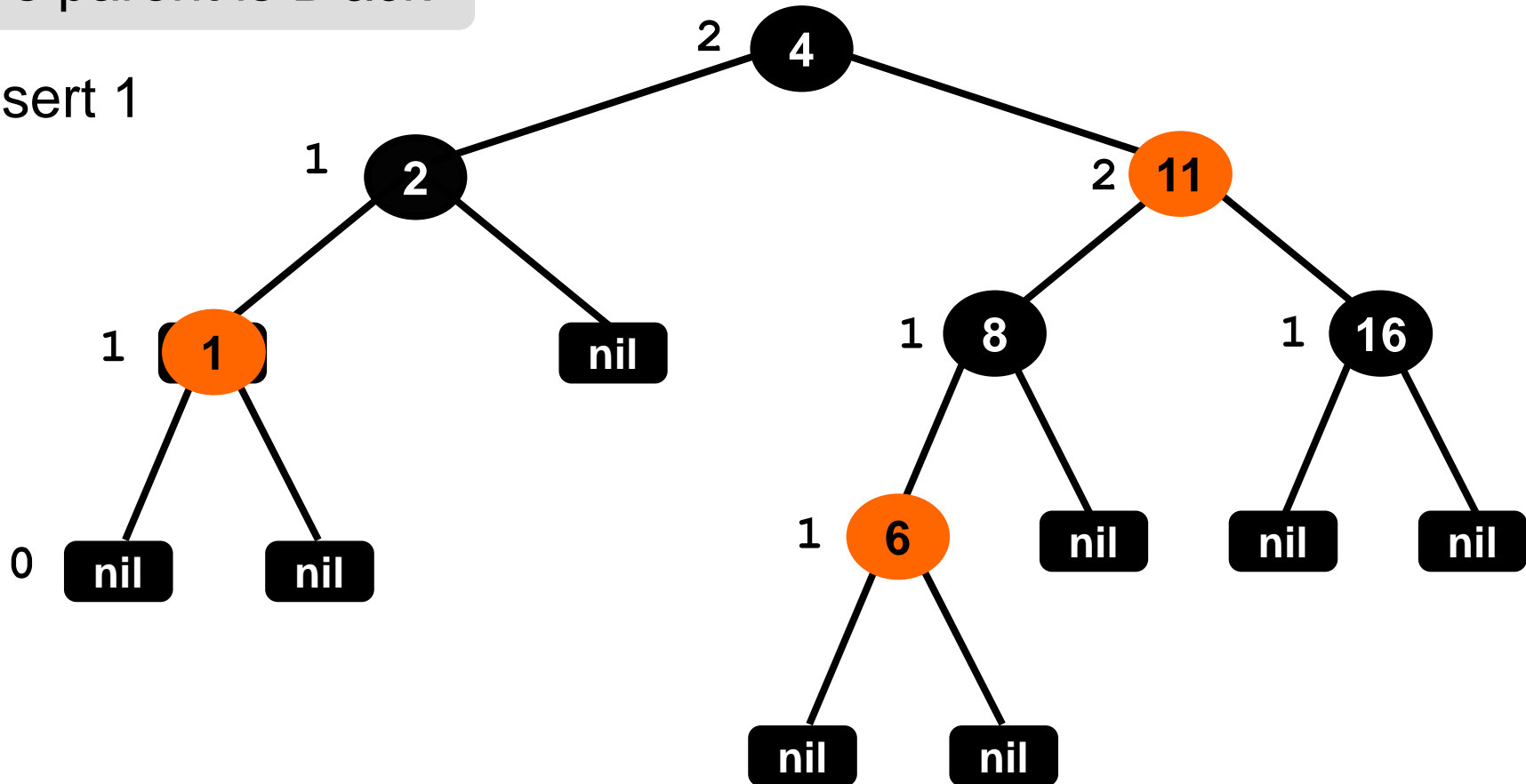
Color root Black



# Inserting in Red-Black Tree

x's parent is Black

Insert 1



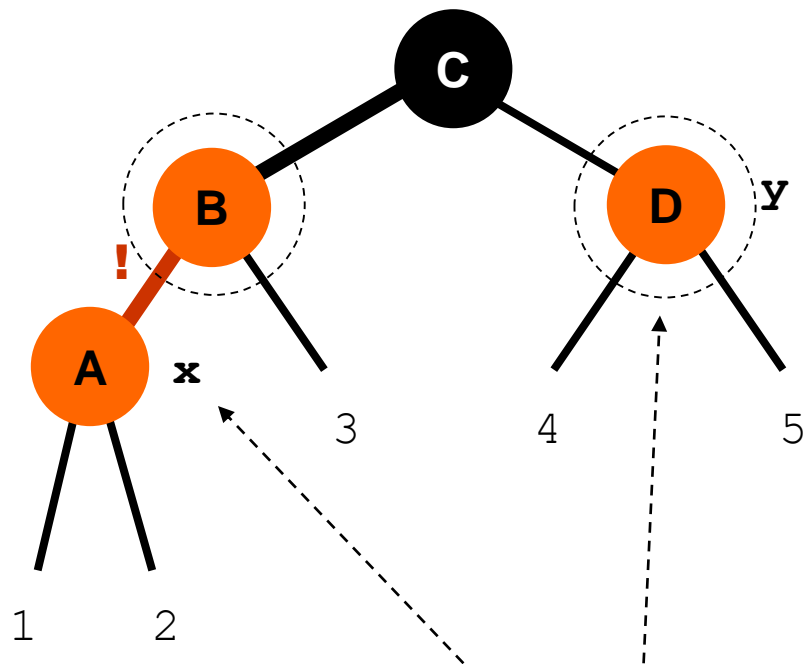
If parent is Black, stop. Tree is a Red-Black tree.

# Inserting in Red-Black Tree

$x$ 's parent is Red

$x$ 's uncle  $y$  is Red

$x$  is a Left child

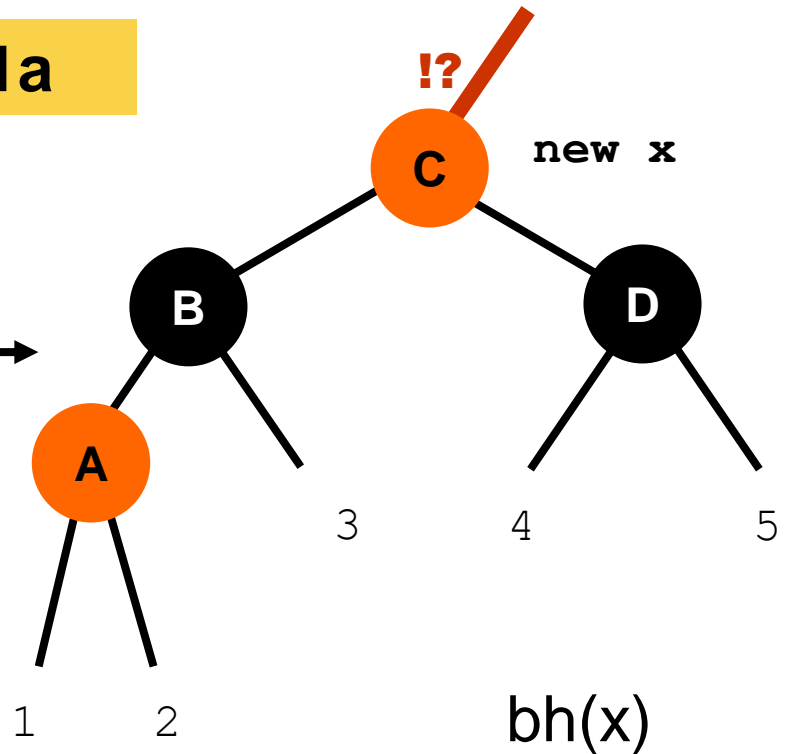


$x$  is node of interest

$x$ 's uncle is Red

Case 1a

Recolor



$bh(x)$   
increased by one

Loop:  $x = x.p.p$

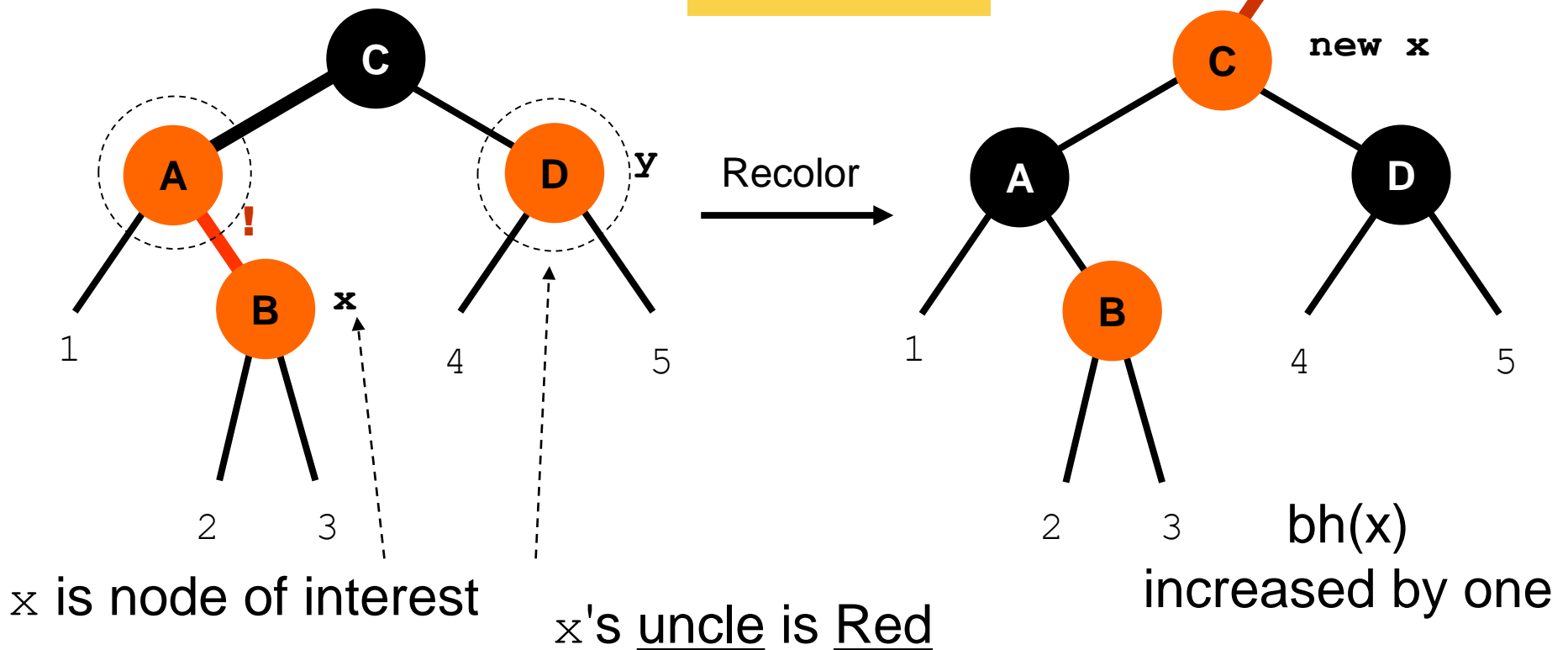
# Inserting in Red-Black Tree

$x$ 's parent is Red

$x$ 's uncle  $y$  is Red

$x$  is a Right child

## Case 1b



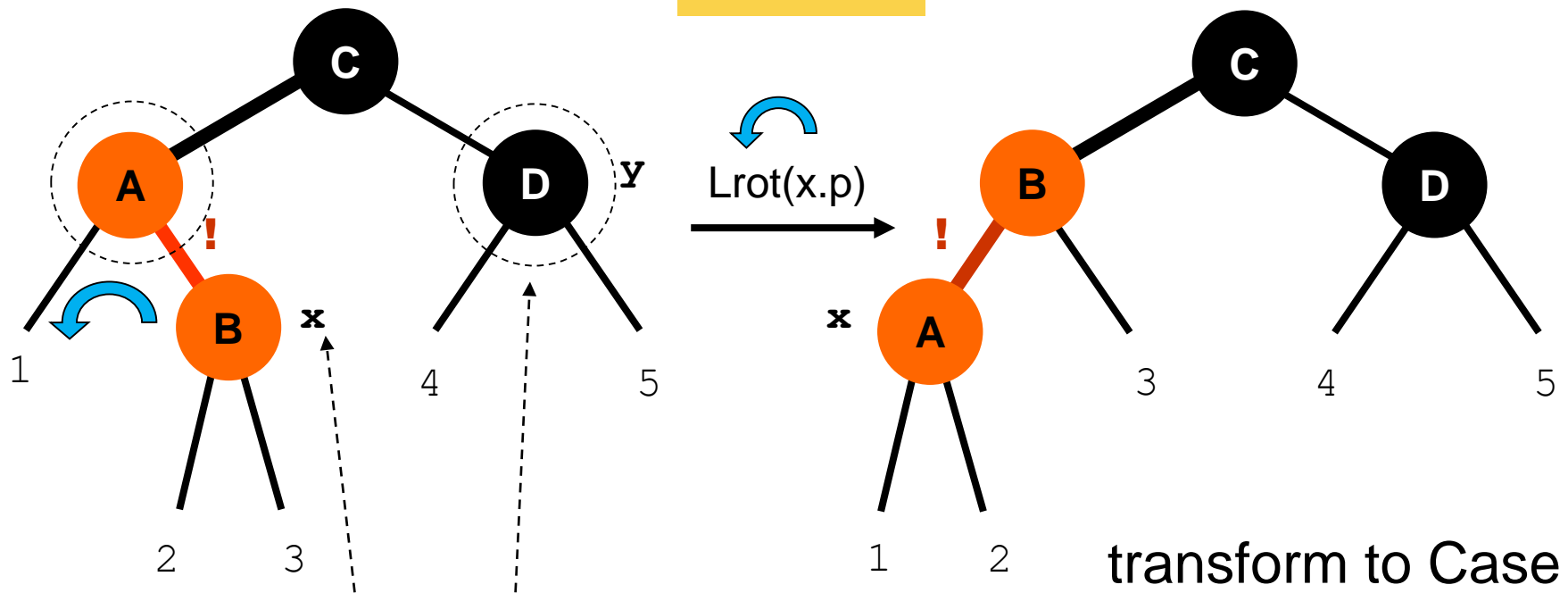
# Inserting in Red-Black Tree

$x$ 's parent is Red

$x$ 's uncle  $y$  is Black

$x$  is a Right child

## Case 2



# Inserting in Red-Black Tree

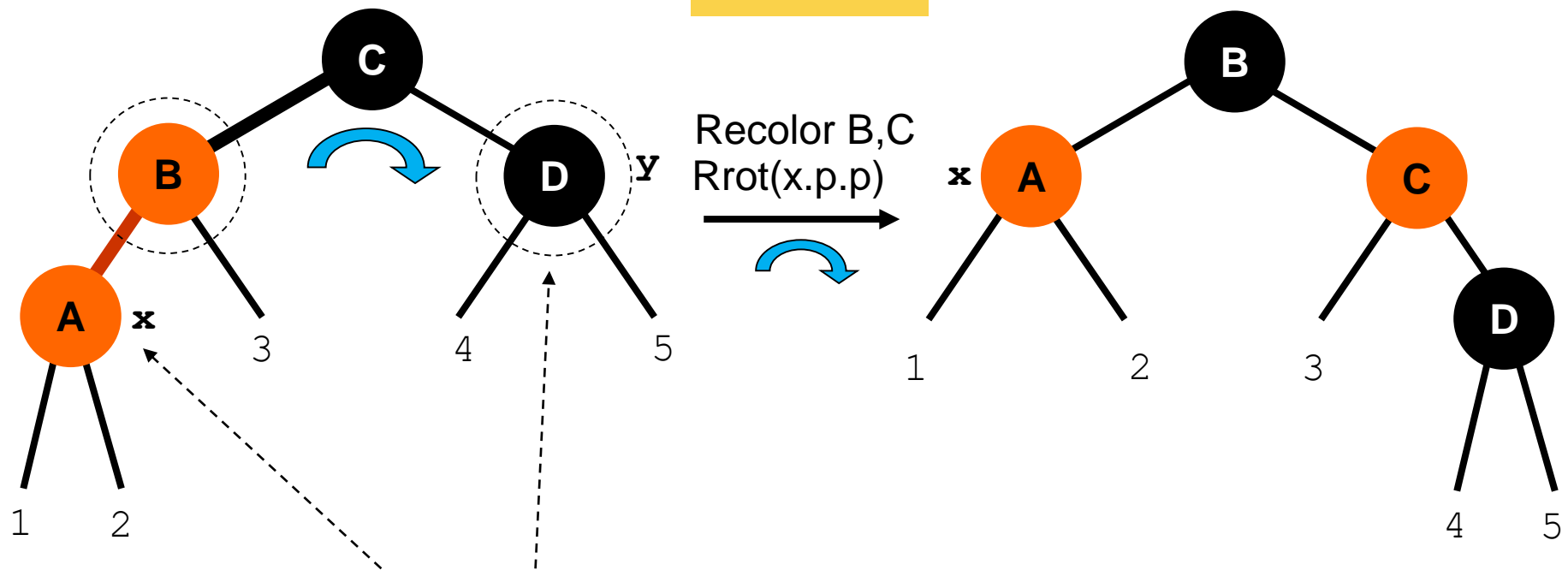
$x$ 's parent is Red

$x$ 's uncle  $y$  is Black

$x$  is a Left child

Terminal case, tree is a Red-Black tree

## Case 3

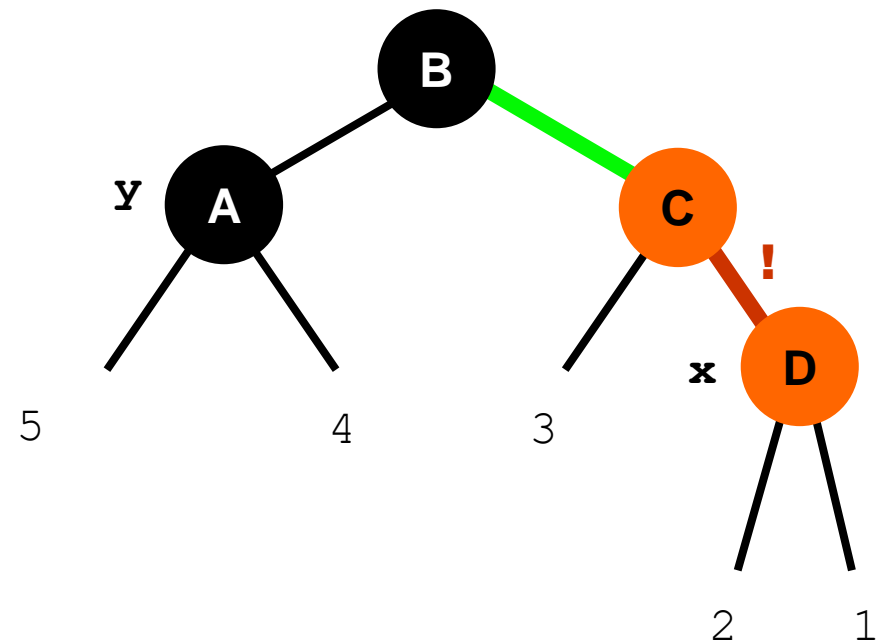
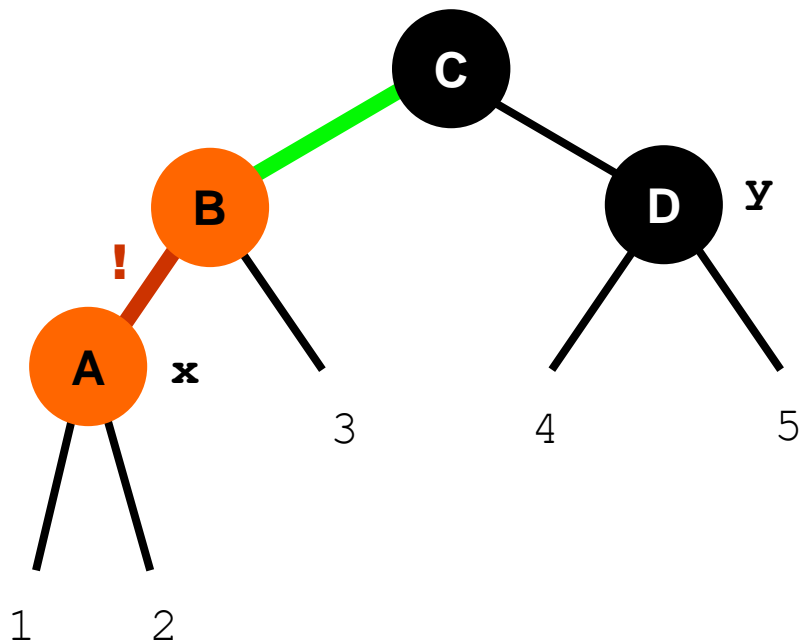


$x$  is a Left child

$x$ 's uncle is Black

# Inserting in Red-Black Tree

Cases Right from the grandparent  
are symmetric



## RB-INSERT( $T, x$ )

```
1  TREE-INSERT( $T, x$ )
2   $color[x] \leftarrow RED$ 
3  while  $x \neq root[T]$  and  $color[p[x]] = RED$ 
4      do if  $p[x] = left[p[p[x]]]$ 
5          then  $y \leftarrow right[p[p[x]]]$            Red uncle  $y \rightarrow$  recolor up
6              if  $color[y] = RED$ 
7                  then  $color[p[x]] \leftarrow BLACK$            ▷ Case 1
8                       $color[y] \leftarrow BLACK$              ▷ Case 1
9                       $color[p[p[x]]] \leftarrow RED$          ▷ Case 1
10                      $x \leftarrow p[p[x]]$                  ▷ Case 1
11              else if  $x = right[p[x]]$ 
12                  then  $x \leftarrow p[x]$                    ▷ Case 2
13                     LEFT-ROTATE( $T, x$ )                   ▷ Case 2
14               $color[p[x]] \leftarrow BLACK$                  ▷ Case 3
15               $color[p[p[x]]] \leftarrow RED$                  ▷ Case 3
16              RIGHT-ROTATE( $T, p[p[x]]$ )                   ▷ Case 3
17          else (same as then clause
                with “right” and “left” exchanged)
18   $color[root[T]] \leftarrow BLACK$ 
```

$p[x]$  = parent of  $x$   
 $left[x]$  = left child of  $x$   
 $y$  = uncle of  $x$

[Cormen90]

# Inserting in Red-Black Tree

Insertion in  $\Theta(\log(n))$  time

Requires at most two rotations



# Deleting in Red-Black Tree

Find node to delete

Delete node as in a regular BST

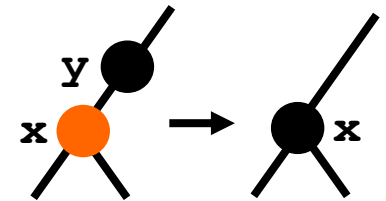
Node  $y$  to be physically deleted will have at most one child  $x$ !!!

If we **delete a Red node**, tree still is a Red-Black tree, **stop**

Assume we **delete a black node**

Let  $x$  be the **child of deleted (black) node  $y$**

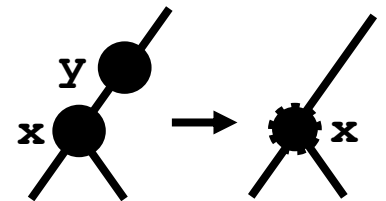
If  $x$  is **red**, color it black and **stop**



while( $x$  is not root) AND (  $x$  is black)


move  $x$  with virtual black mark through the tree

(If  $x$  is black, mark it virtually double black **A**)



//note that the whole  $x$  's subtree lost 1 unit of black height

# Deleting in Red-Black Tree

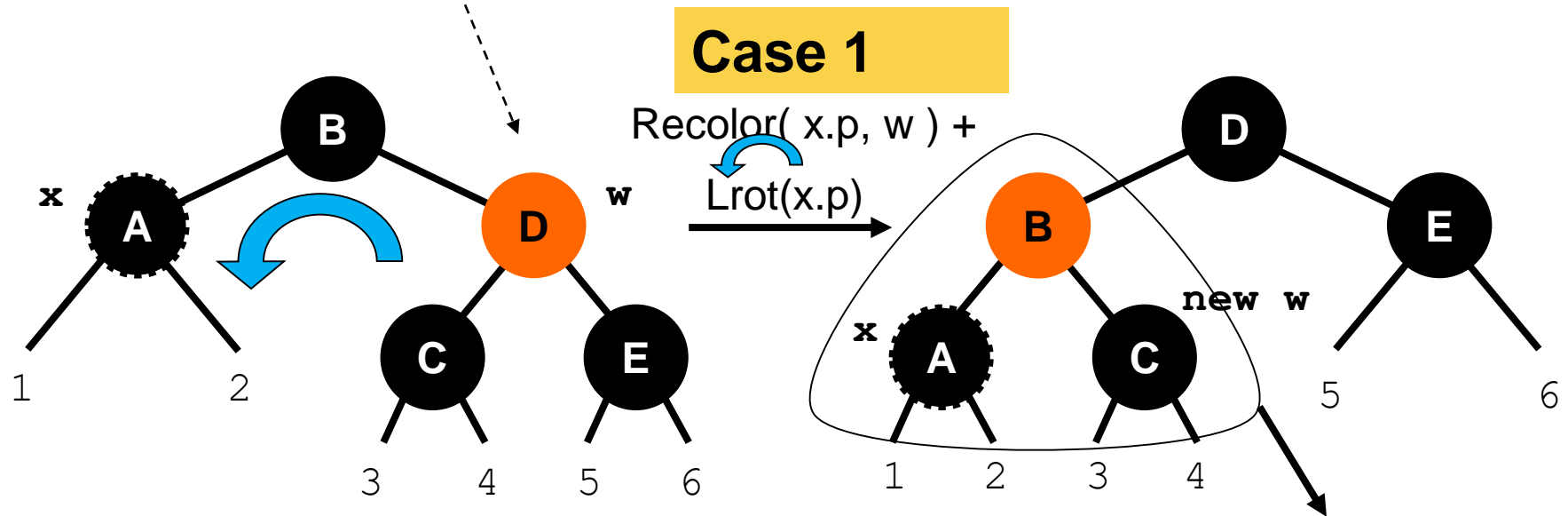
```
while(x is not root) AND ( x is black) {  
    // move x with virtual black mark  through the tree  
    // just recolor or rotate other subtree up (decrease bh in R subtree)  
    if (sibling is red)  
        -> Case 1: Rotate right subtree up, color sibling black, and  
                continue in left subtree with the new sibling  
    if (sibling is black with both black children)  
        -> Case 2: Color sibling red and go up  
    else // black sibling with one or two red children  
        if(red left child) -> Case 3: rotate to surface  
        Case 4: Rotate right subtree up  
}
```

# Deleting in R-B Tree - Case 1

$x$  is the child of the physically deleted black node  $\Rightarrow$  double black  
 $x$ 's sibling  $w$  is red



( $x$ 's parent *must* be black)



$x$  stays at the same black height

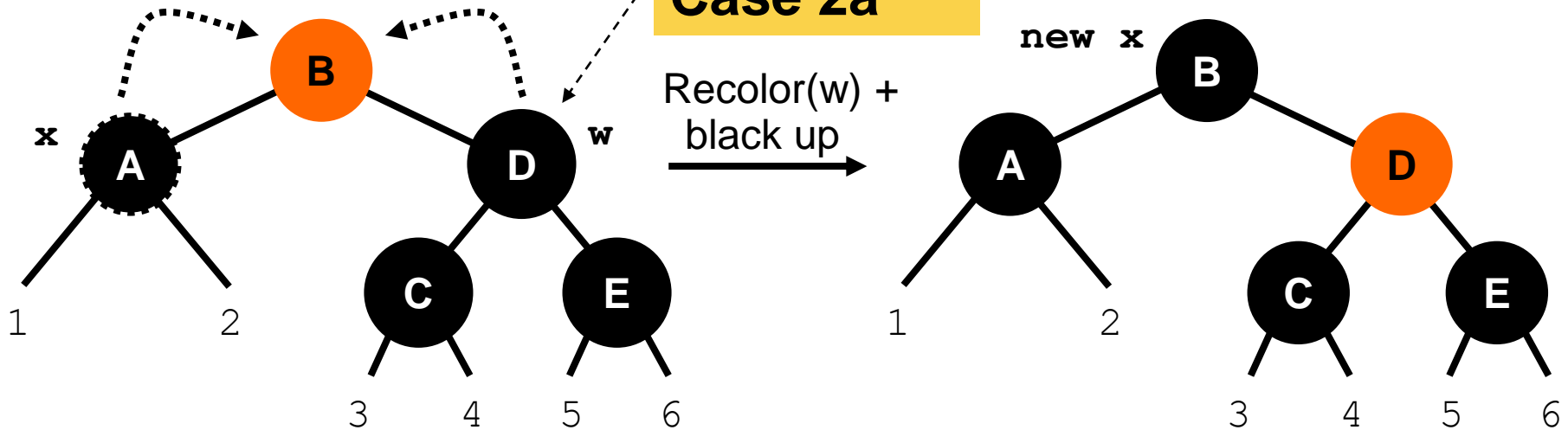
**continue**

[Possibly transforms to case 2a and terminates – depends on 3,4]

# Deleting in R-B Tree - Case 2a

- x's sibling w is black
- x's parent is red
- x's sibling left child is black
- x's sibling right child is black

## Case 2a



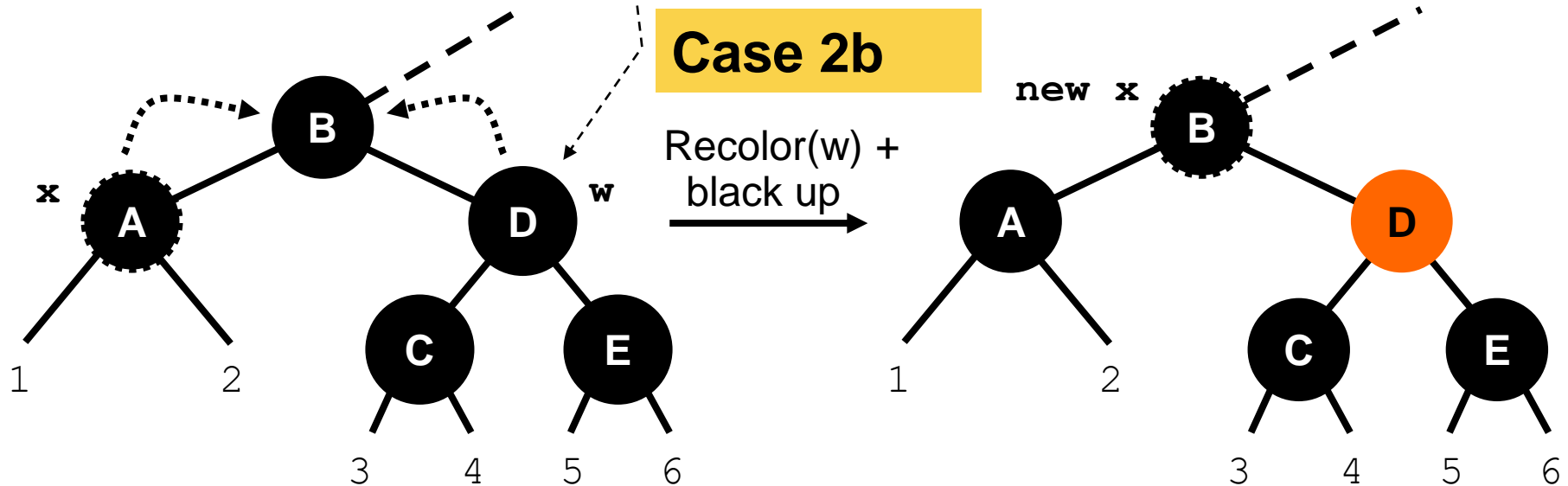
Terminal case, tree is Red-Black tree

**STOP**

Note that A's subtree *had* less by 1 black height than D's subtree

# Deleting in R-B Tree - Case 2b

- $x$ 's sibling  $w$  is black
- $x$ 's parent is black
- $x$ 's sibling left child is black
- $x$ 's sibling right child is black



Decreases  $x$  black height by one

**continue with new  $x$**

Note that A's subtree *had less by 1* black height than D's subtree

# Deleting in R-B Tree - Case 3

$x$ 's sibling  $w$  is black

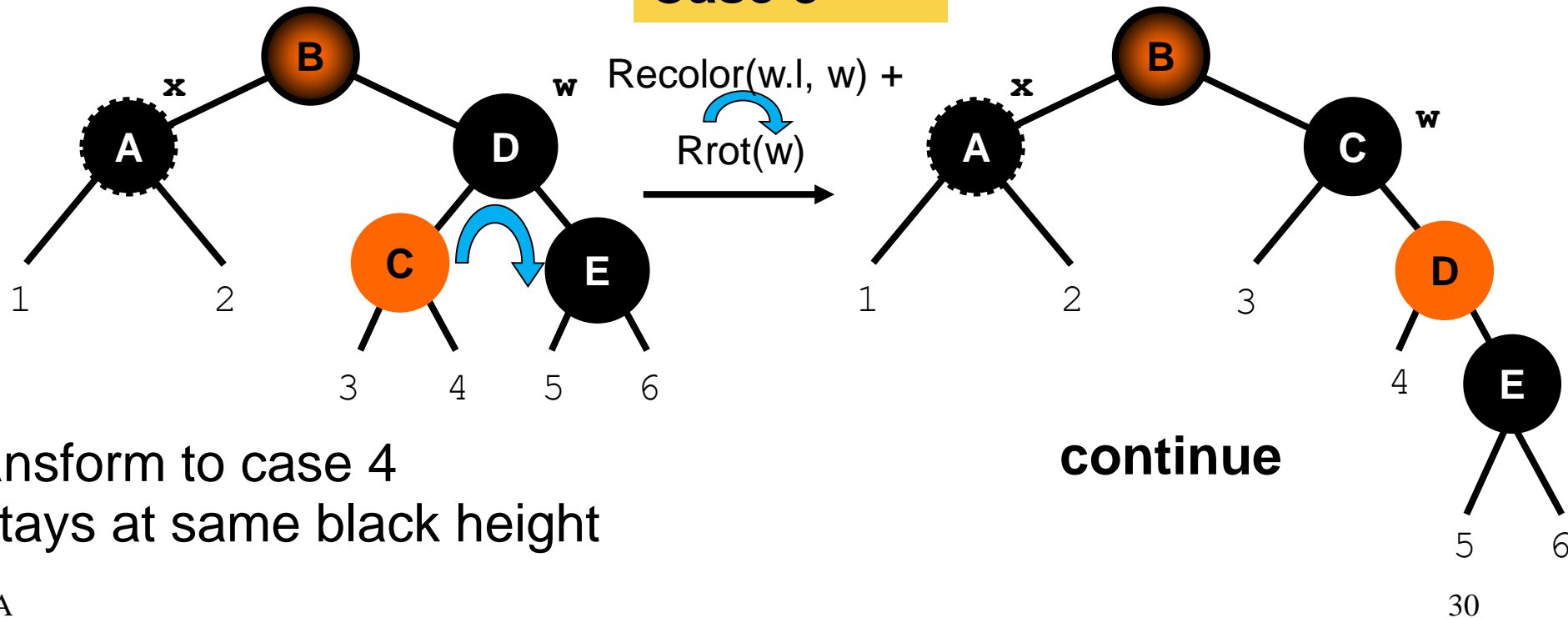
$x$ 's parent is either

$x$ 's sibling left child is red

// impossible to color  $w$  red

$x$ 's sibling right child is black

## Case 3



Transform to case 4

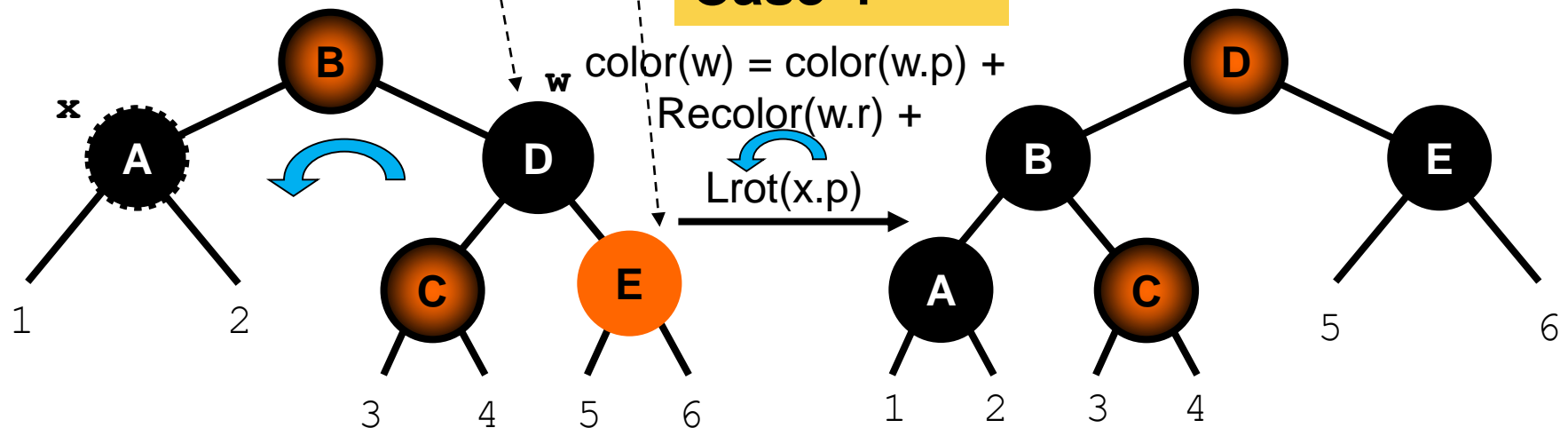
$x$  stays at same black height

# Deleting in R-B Tree - Case 4

- $x$ 's sibling  $w$  is black
- $x$ 's parent is either
- $x$ 's sibling left child is either
- $x$ 's sibling right child is red

// impossible to color  $w$  red

## Case 4



Terminal case, tree is Red-Black tree  
( D inherits the color of B)

**STOP**

# Deleting in Red-Black Tree

RB-DELETE( $T, z$ )

```
1  if left[z] = nil[T] or right[z] = nil[T]
2    then y ← z
3    else y ← TREE-SUCCESSOR(z)
4  if left[y] ≠ nil[T]
5    then x ← left[y]
6    else x ← right[y]
7  p[x] ← p[y]
8  if p[y] = nil[T]
9    then root[T] ← x
10   else if y = left[p[y]]
11         then left[p[y]] ← x
12         else right[p[y]] ← x
13  if y ≠ z
14    then key[z] ← key[y]
15         ▷ If y has other fields, copy them, too.
16  if color[y] = BLACK
17    then RB-DELETE-FIXUP(T, x)
18  return y
```

Notation similar to AVL  
 $z$  = *logically* removed  
 $y$  = *physically* removed  
 $x$  =  $y$ 's only child

[Cormen90]



# RB-DELETE-FIXUP( $T, x$ )

$x$  = *child* of removed node  
 $p[x]$  = parent of  $x$   
 $w$  = sibling of  $x$

```

1  while  $x \neq \text{root}[T]$  and  $\text{color}[x] = \text{BLACK}$ 
2      do if  $x = \text{left}[p[x]]$ 
3          then  $w \leftarrow \text{right}[p[x]]$ 

```

```

4      if  $\text{color}[w] = \text{RED}$ 
5          then  $\text{color}[w] \leftarrow \text{BLACK}$            ▷ Case 1
6               $\text{color}[p[x]] \leftarrow \text{RED}$        ▷ Case 1
7              LEFT-ROTATE( $T, p[x]$ )             ▷ Case 1
8               $w \leftarrow \text{right}[p[x]]$          ▷ Case 1

```

R subtree up  
Check L

```

9      if  $\text{color}[\text{left}[w]] = \text{BLACK}$  and  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
10         then  $\text{color}[w] \leftarrow \text{RED}$            ▷ Case 2
11              $x \leftarrow p[x]$                  ▷ Case 2

```

Recolor  
Black up  
Go up

```

12     else if  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
13         then  $\text{color}[\text{left}[w]] \leftarrow \text{BLACK}$    ▷ Case 3
14              $\text{color}[w] \leftarrow \text{RED}$            ▷ Case 3
15             RIGHT-ROTATE( $T, w$ )                ▷ Case 3
16              $w \leftarrow \text{right}[p[x]]$          ▷ Case 3

```

inner R-  
subtree up

```

17          $\text{color}[w] \leftarrow \text{color}[p[x]]$        ▷ Case 4
18          $\text{color}[p[x]] \leftarrow \text{BLACK}$          ▷ Case 4
19          $\text{color}[\text{right}[w]] \leftarrow \text{BLACK}$    ▷ Case 4
20         LEFT-ROTATE( $T, p[x]$ )                 ▷ Case 4
21          $x \leftarrow \text{root}[T]$                  ▷ Case 4

```

R subtree up  
stop

else (same as then clause  
 with “right” and “left” exchanged)

```

23   $\text{color}[x] \leftarrow \text{BLACK}$ 

```

[Cormen90]

# Deleting in R-B Tree

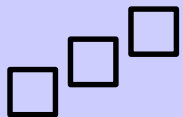
Delete time is  $\Theta(\log(n))$

At most three rotations are done

# R-B Tree vs. AVL Tree

- Faster insertion and deletion operations (fewer rotations are done due to relatively relaxed balancing).
- Requires only 1 bit of information per node.
- AVL trees provide faster lookups.

=> Red-Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in databases where faster retrievals are required.

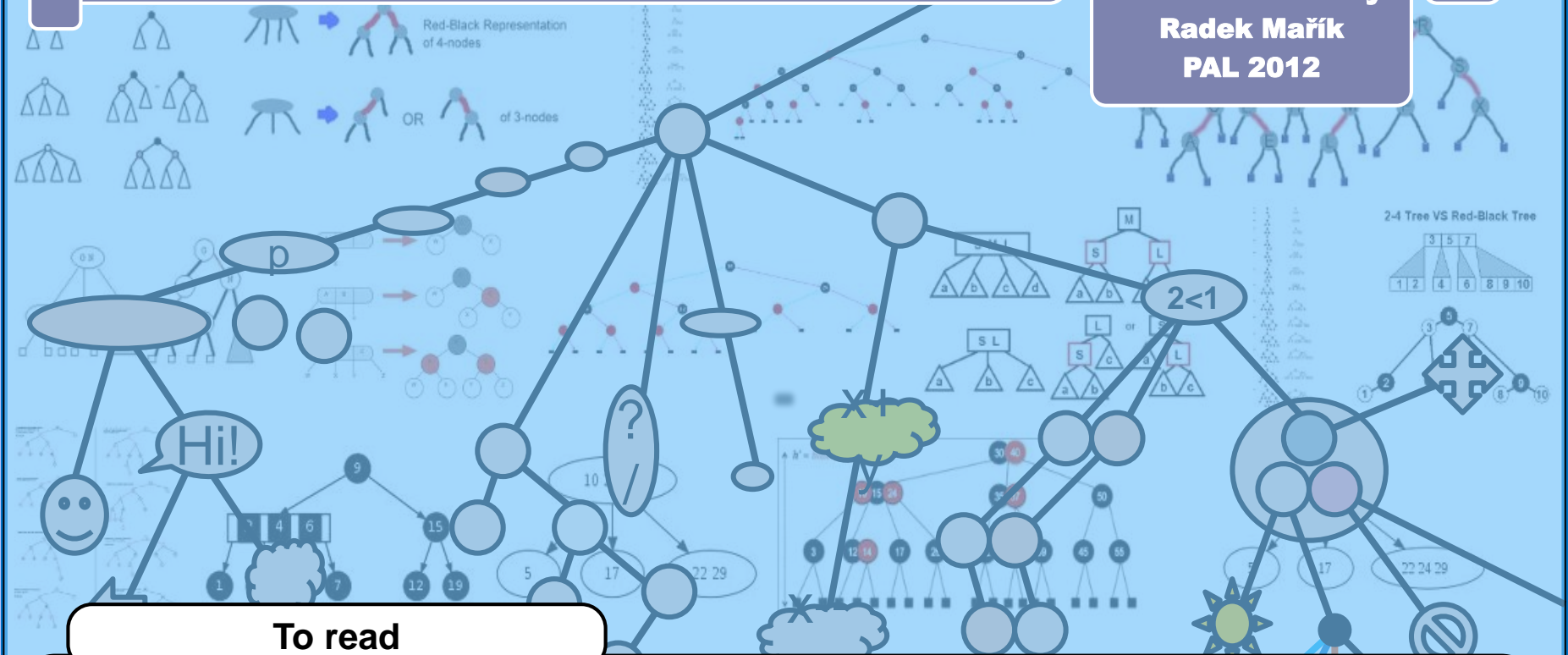


# Křížovková pauza

	puška	červeno	české město	unavím	měkký kov	iniciály českého zpěváka a baviče	atmosférický světelný jev	obejda	mezinárodní značka Rumunska
doprovod manželky šlechtice									
1. část tajenky									
identi- fikátor			staré zájmeno			Roosveltovy iniciály			ušlechtilý kov
			programo- vací jazyk			tlak krve (zkr.)			
bývalý francouzský tenista					slovenská rocková skupina popěvek				
značka Tesly		2. část tajenky							
		těžký vodík							
patřící obyvateli ráje						buď nápomocný			

# 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++, 3<sup>rd</sup> 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.

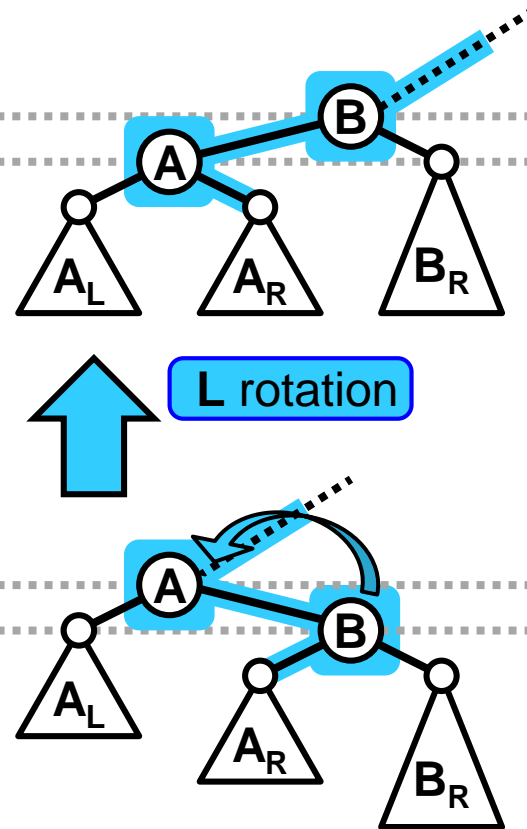
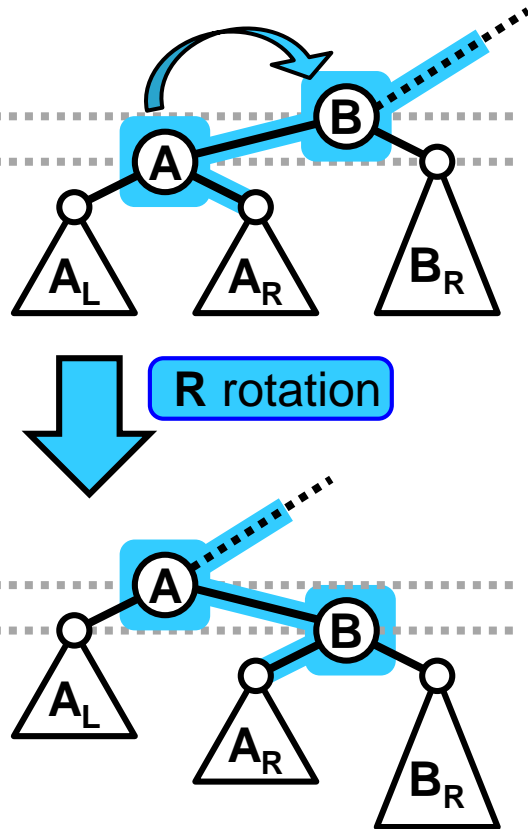
**Delete:** Splay the node to the root and then delete it like in a BST.

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

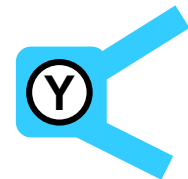
## Splay tree

- 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.



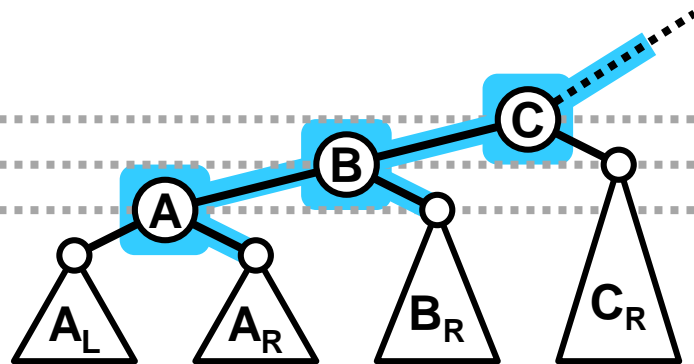
Afected nodes and edges

#### Note

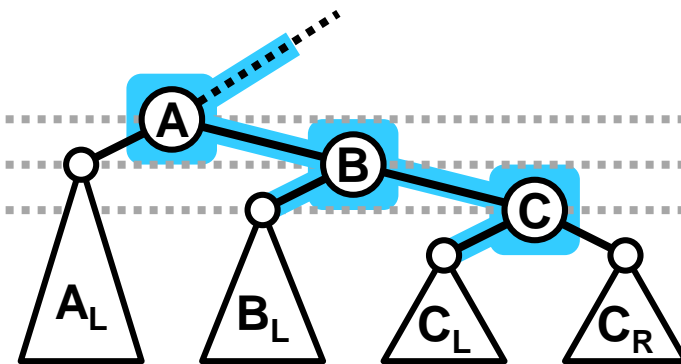
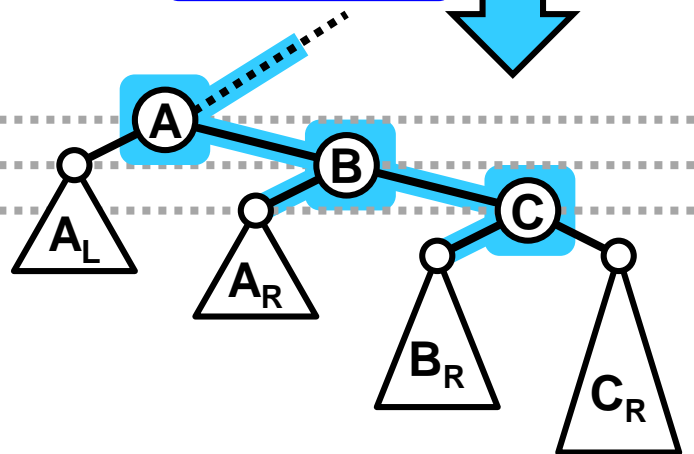
The terms "Zig" and "Zag" are not chiral, that is, they do not describe the direction (left or right) of the actual rotations.



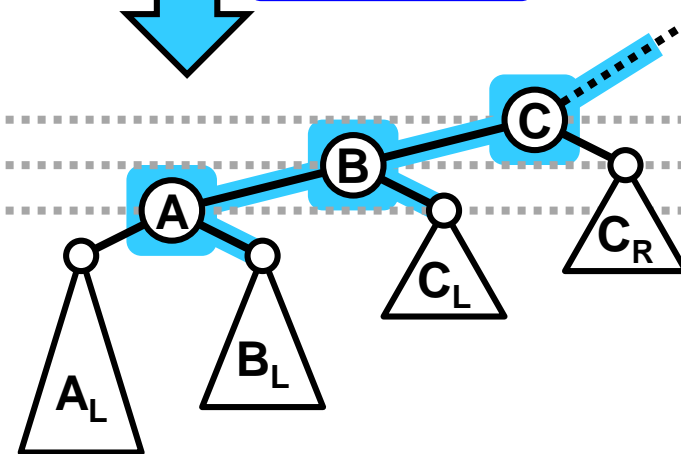
### Zig - zig rotation



RR rotation

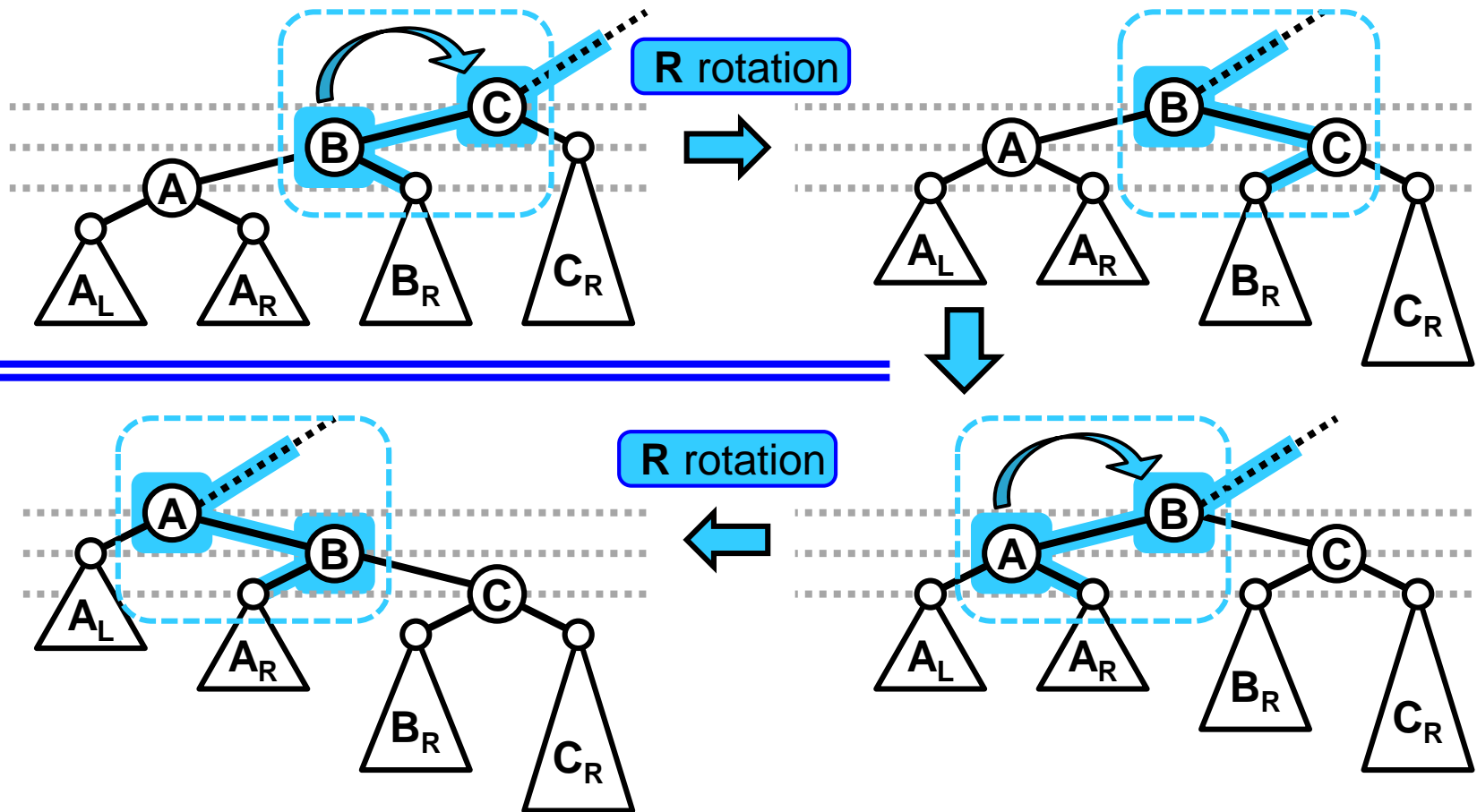


LL rotation



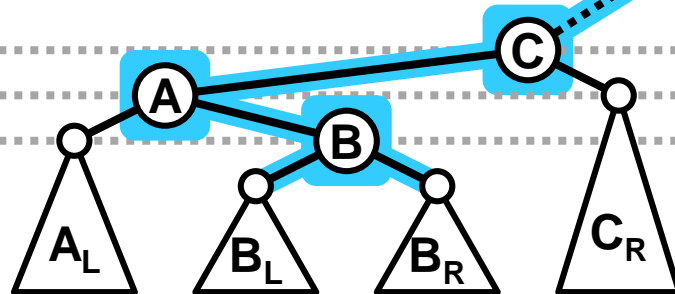
Note that the topmost node might be either the tree root or the left or the right child of its parent. Only the left child case is shown. The other cases are analogous.

### Zig - zig rotation

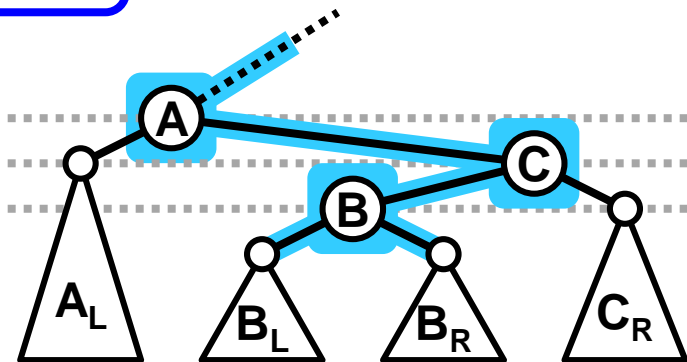
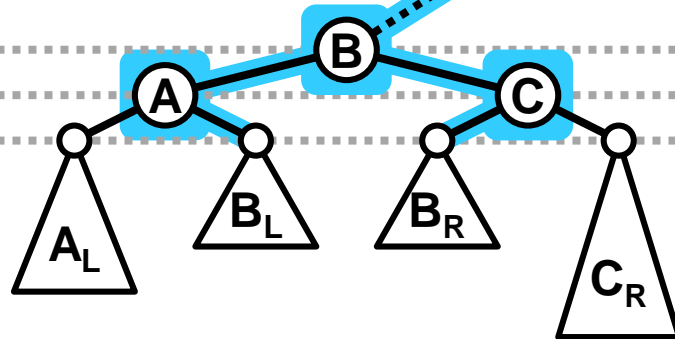


Both simple rotations are performed at the top of the current subtree, the splayed node (with key A) is not involved in the first rotation.

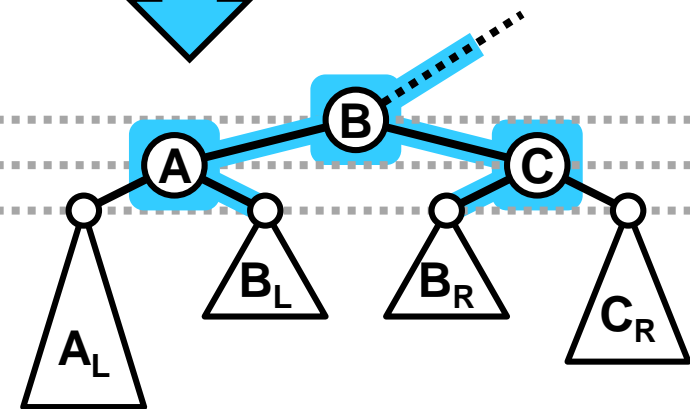
### Zig - zag rotation



LR rotation

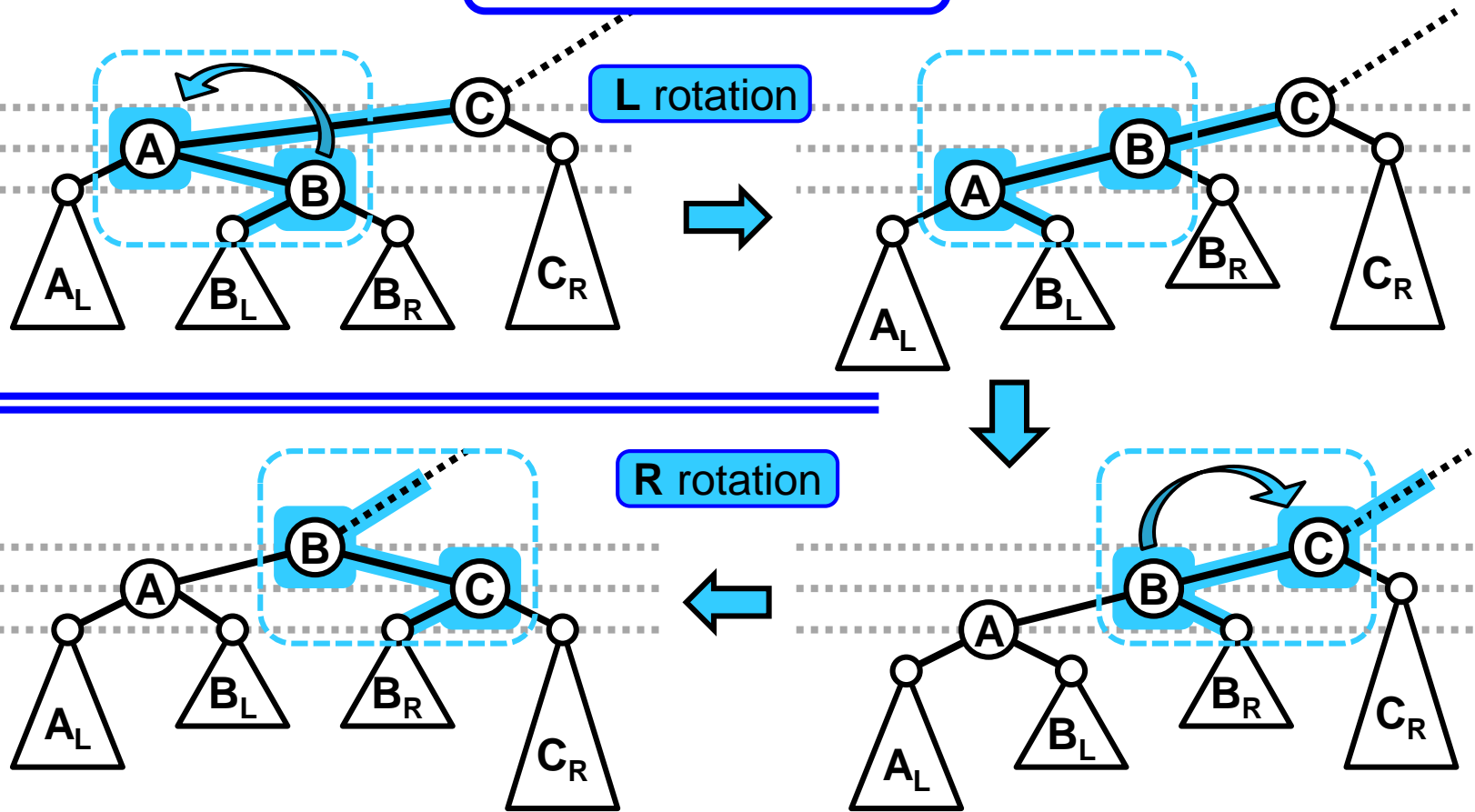


RL rotation



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



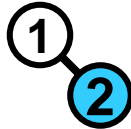
#### Note:

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

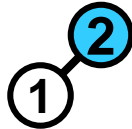
Insert 1



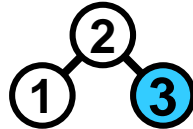
Insert 2



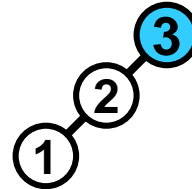
Splay



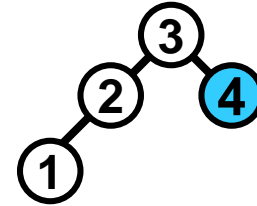
Insert 3



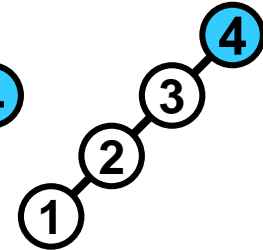
Splay



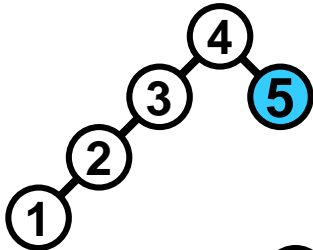
Insert 4



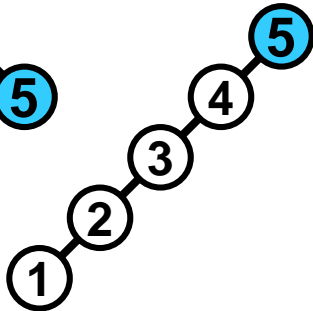
Splay



Insert 5

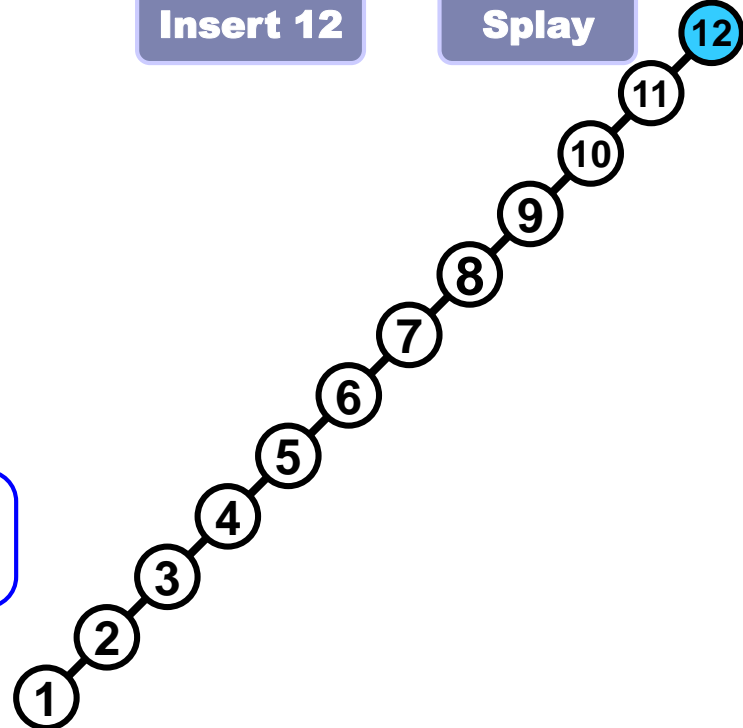


Splay



etc...

Insert 12

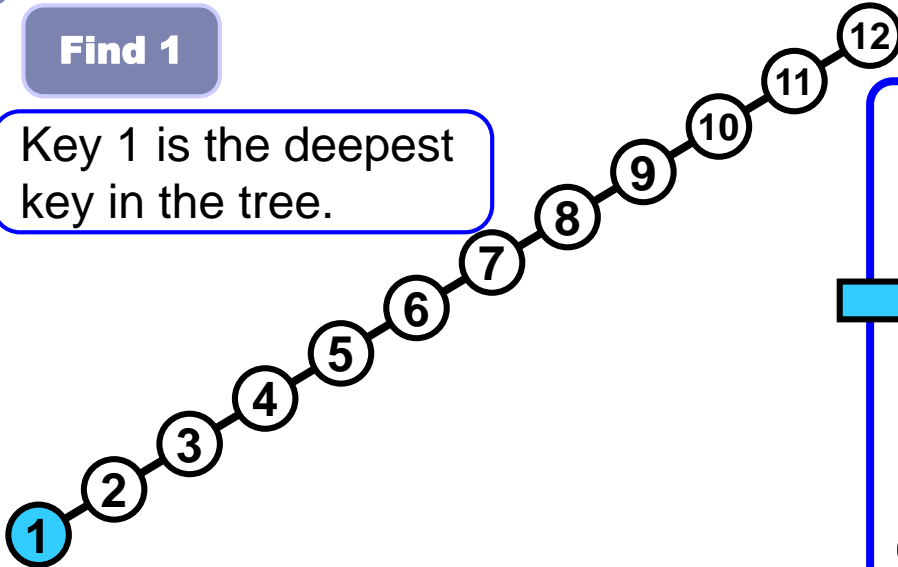


Splay

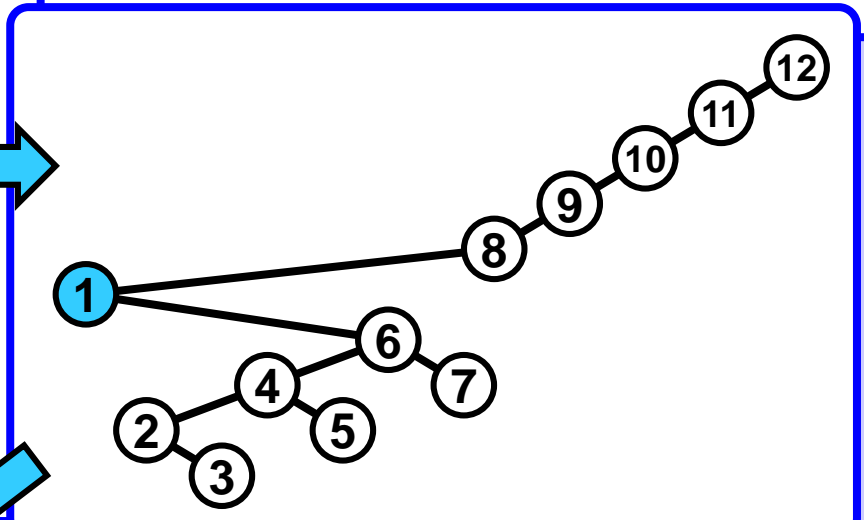
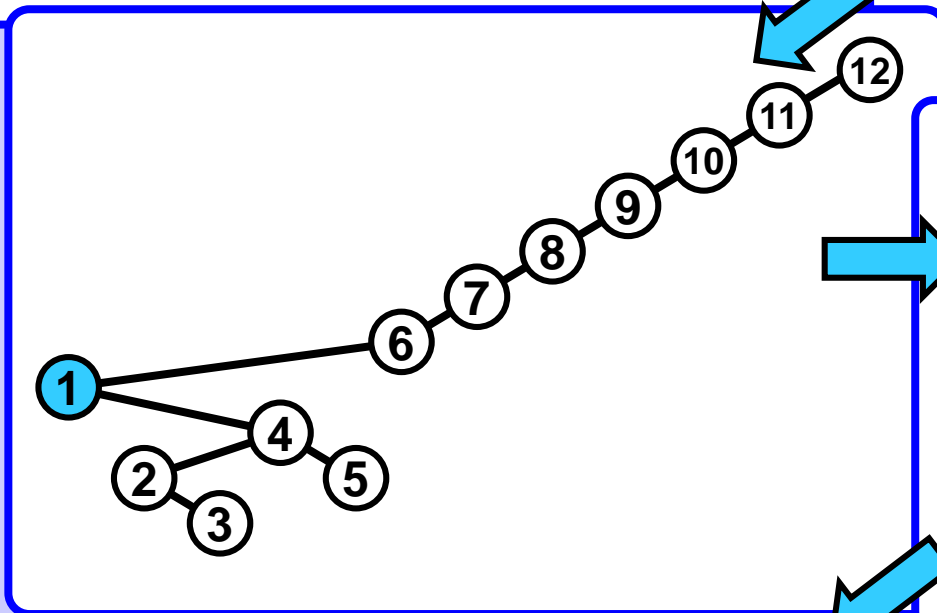
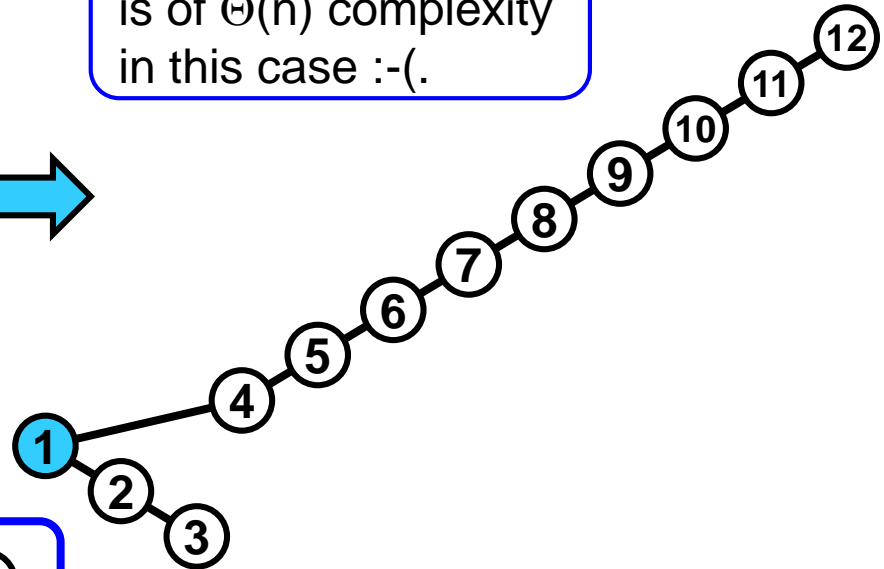
Note the extremely inefficient shape of the resulting tree.

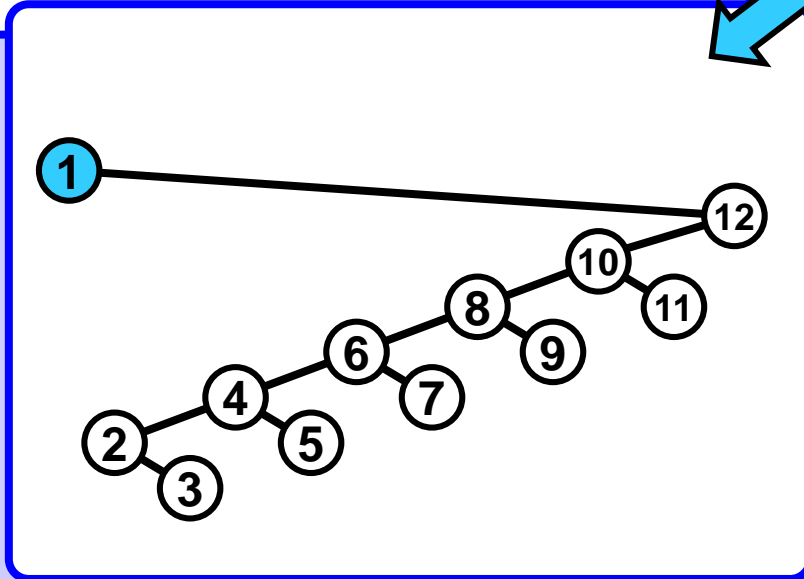
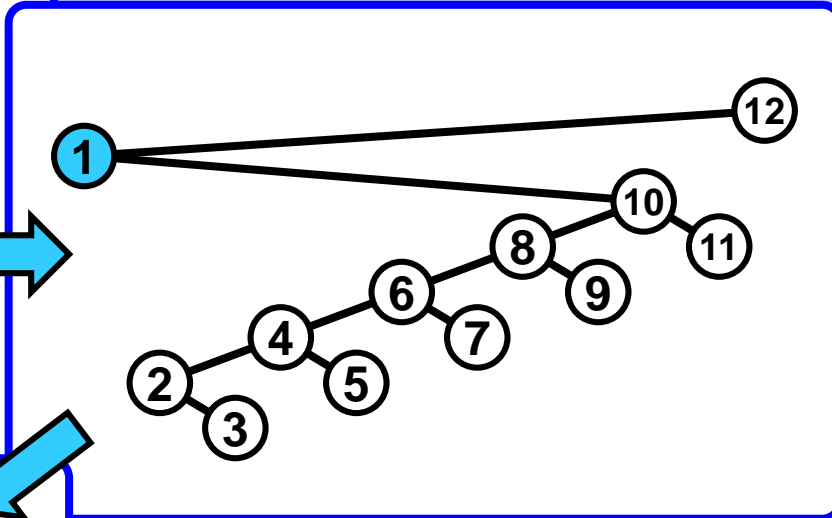
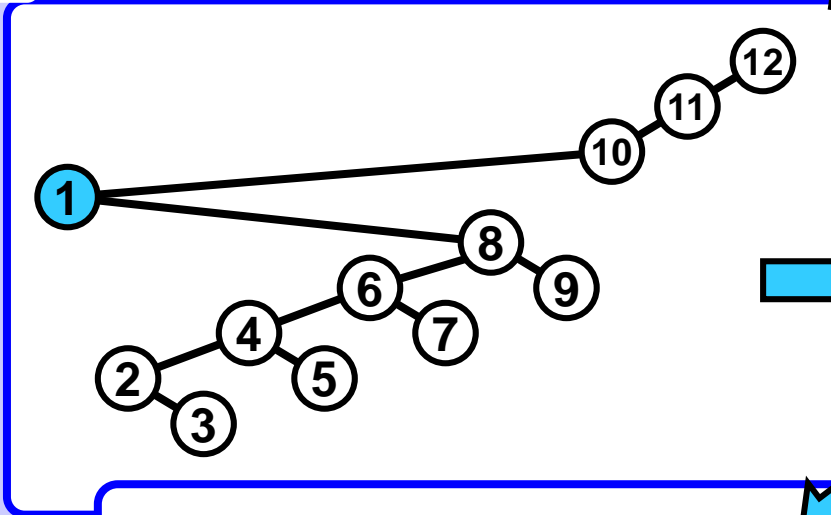
### Find 1

Key 1 is the deepest key in the tree.

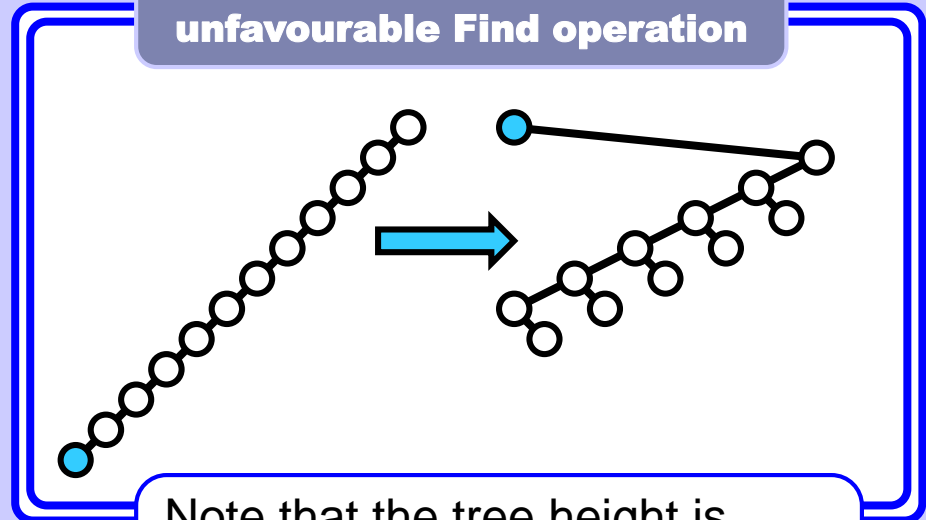


Find operation is of  $\Theta(n)$  complexity in this case :-(.





**Scheme - Result of the most unfavourable Find operation**

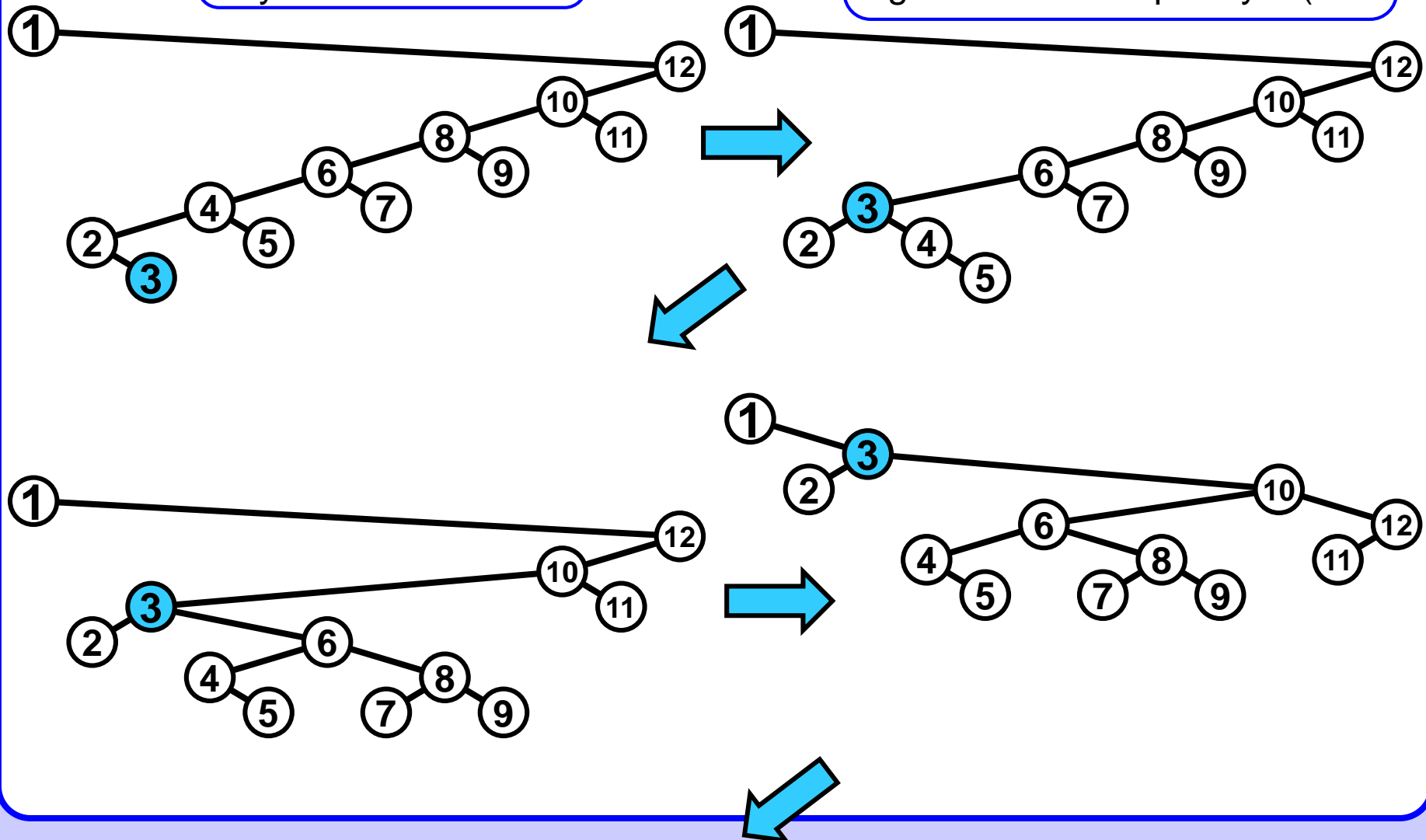


Note that the tree height is roughly halved.  $H \rightarrow (H + 3) / 2$

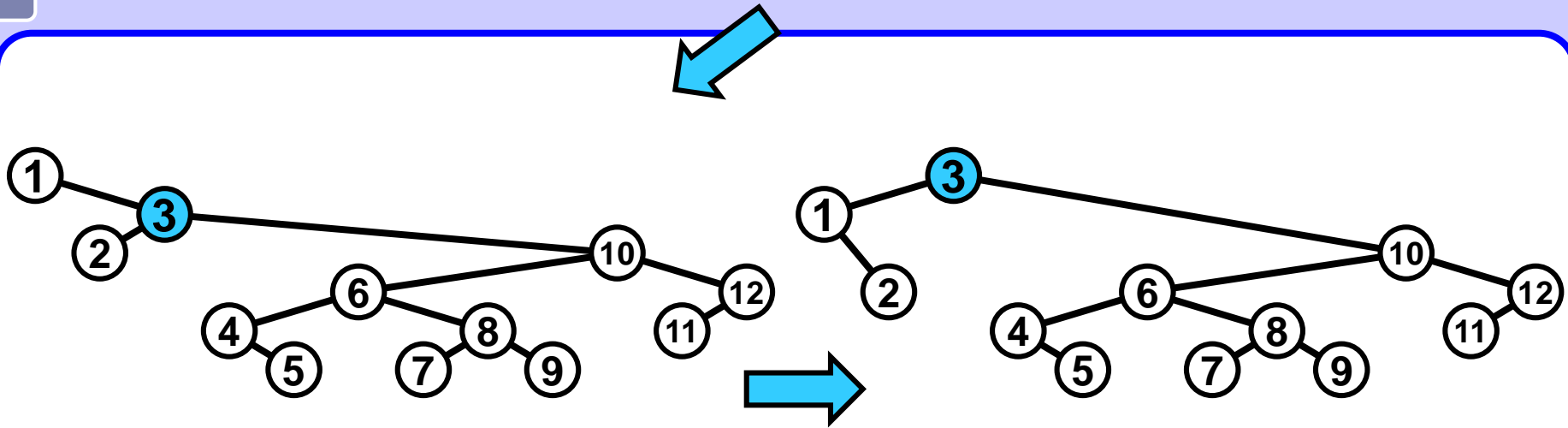
**Find 3**

Key 3 is the deepest key in the tree.

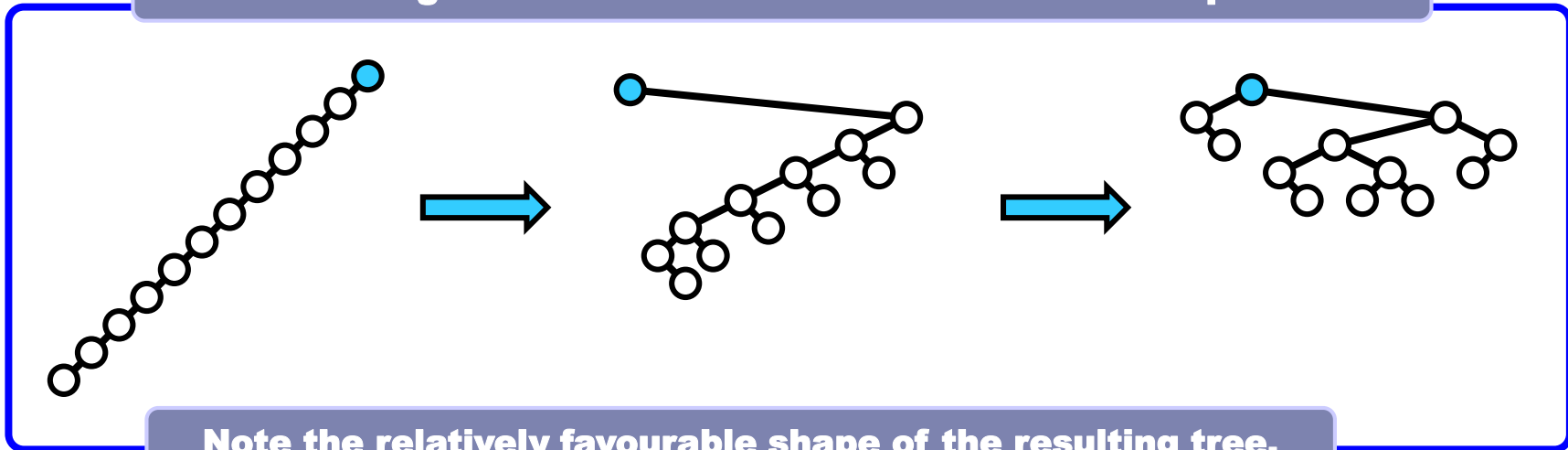
The Find operation would be again of  $\sim n$  complexity. :-)







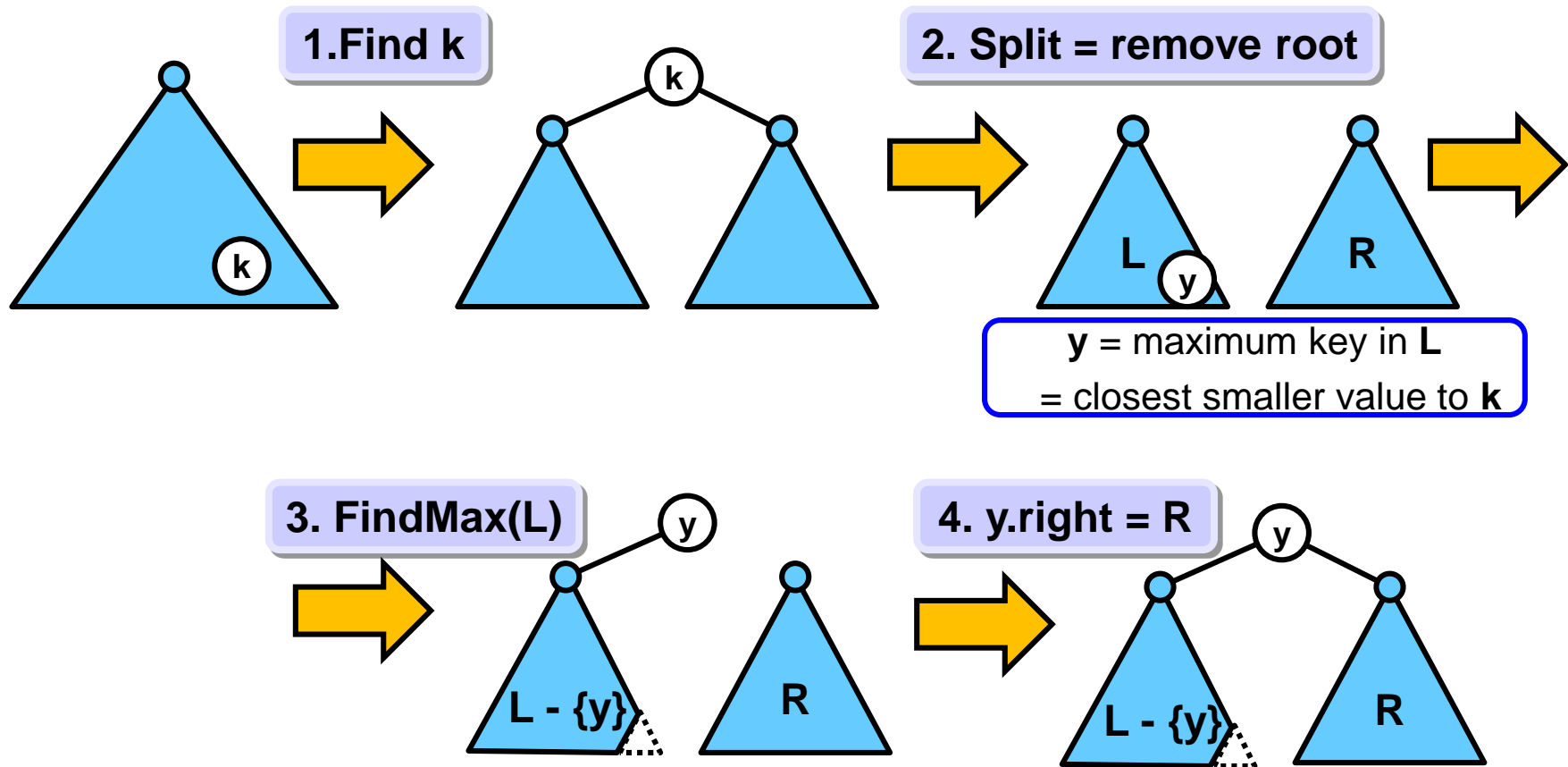
**Scheme - Progress of the two most unfavourable Find operations.**



**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;



### 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

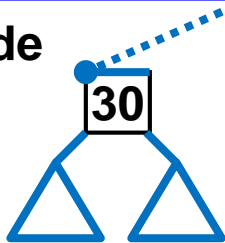
A **2-3-4 search tree** is structurally a **B-tree** of min degree 2 and max degree 4.

A node is a **2-node** or a **3-node** or a **4-node**.

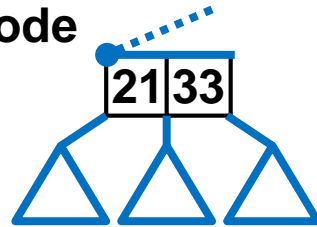
If a node is not a leaf it has the corresponding number (2, 3, 4) of children.

All leaves are at the same distance from the root, the tree is **perfectly balanced**.

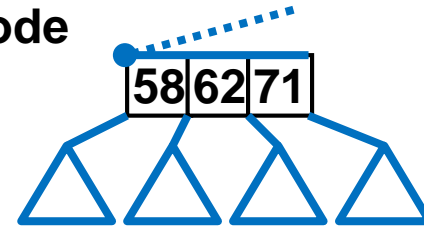
2-node



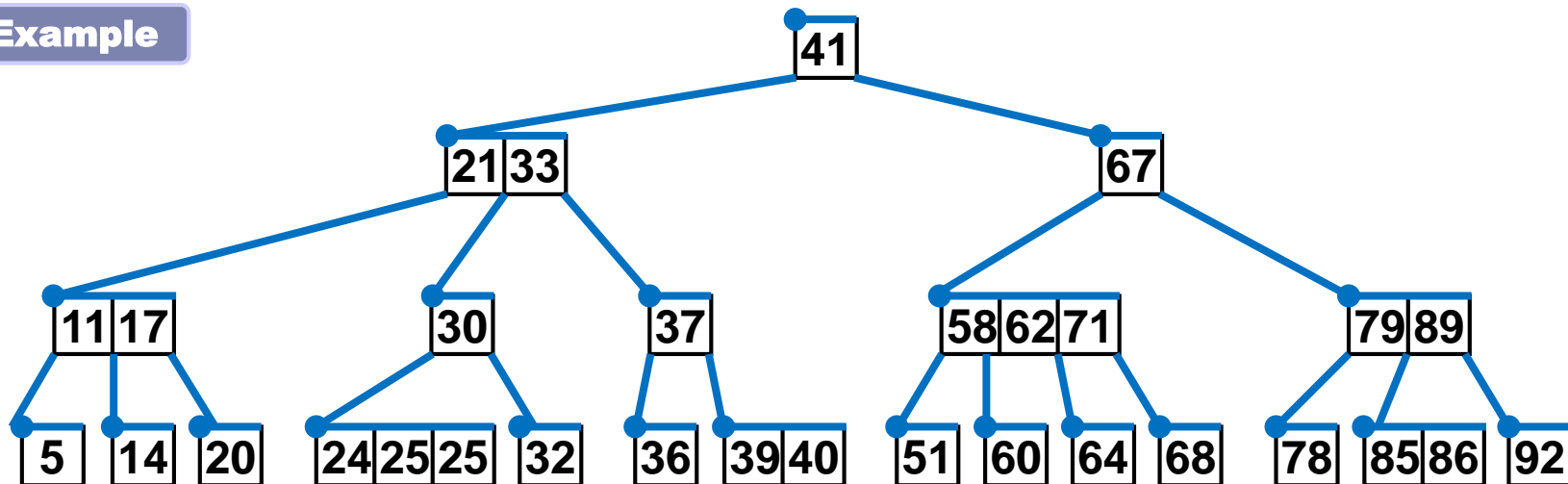
3-node



4-node



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.

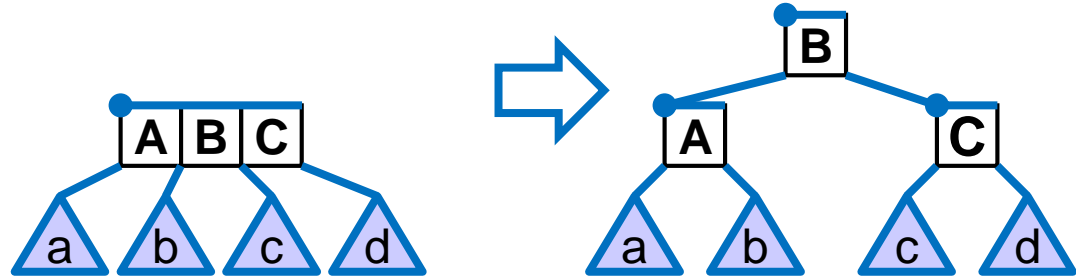
This strategy prevents the following from happening:

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.

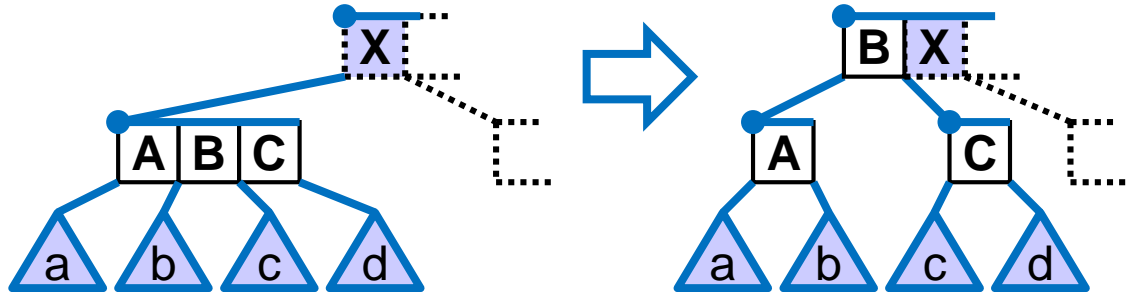
Splitting 4-nodes on the way down results in sparse occurrence of 4-nodes in the tree, thus the nodes never have to be split recursively bottom-up.

## Delete: As in B-tree

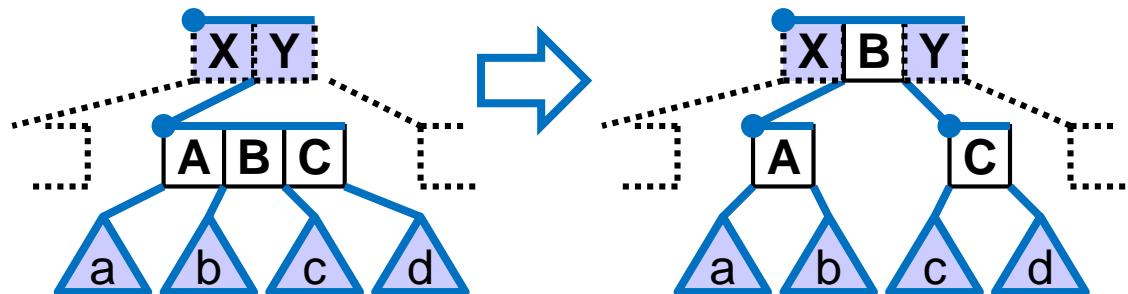
Split node is the root.  
Only the root splitting  
increases the tree height



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 rightmost 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: SEARCHINGKLM

Insert S



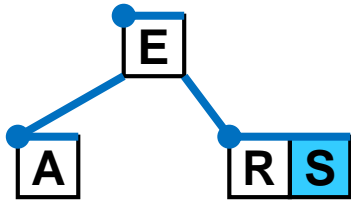
Insert E



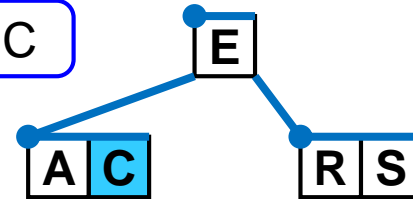
Insert A



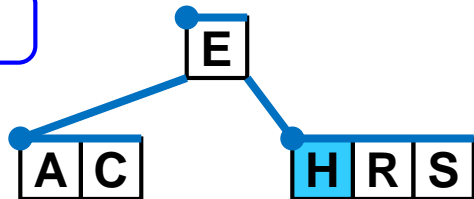
Insert R



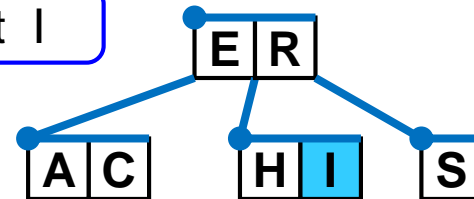
Insert C

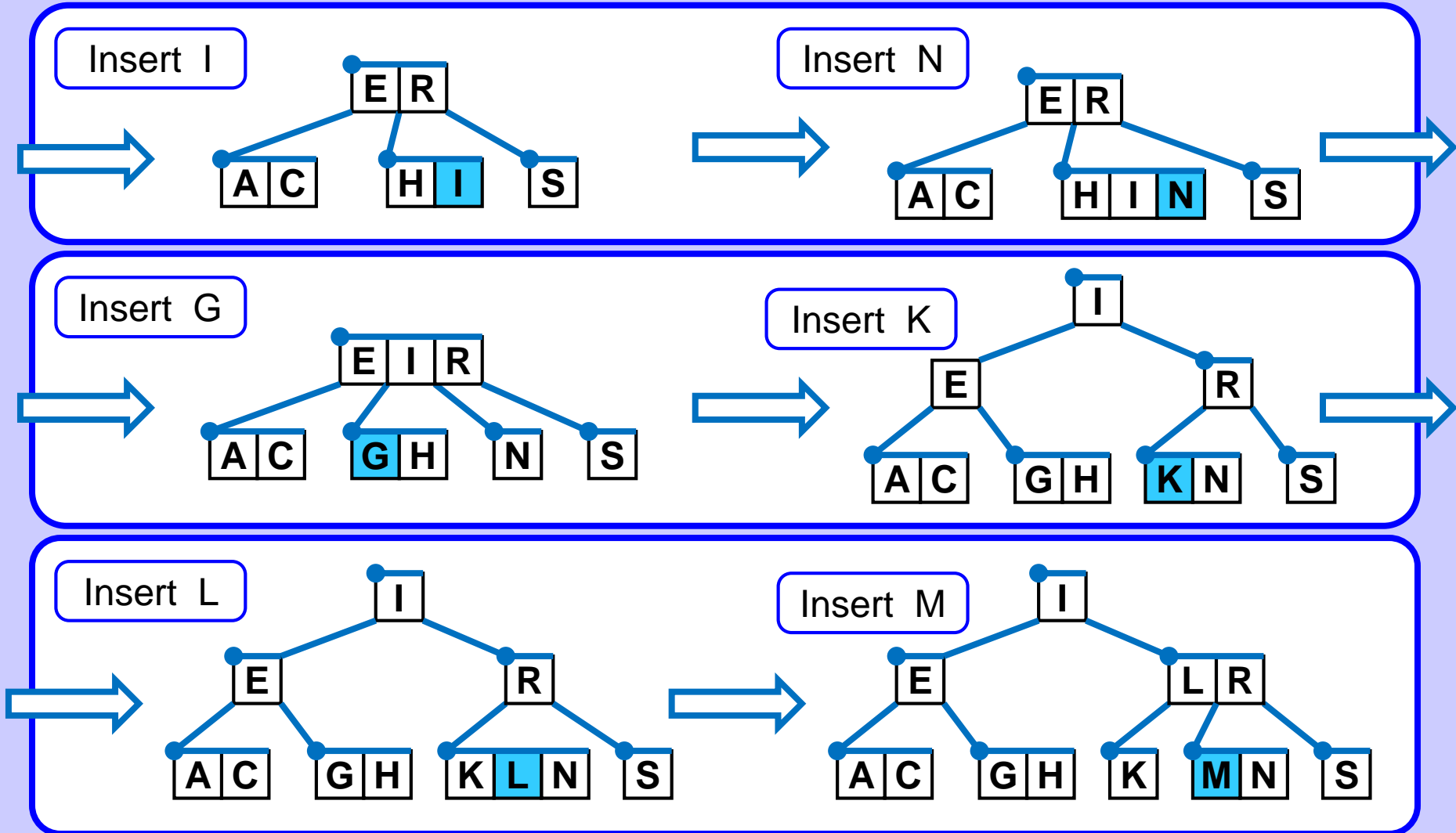


Insert H



Insert I





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  $\{1, \dots, 10^9\}$  inserted into initially empty 2-3-4 tree:

<b>N</b>	<b>Tree depth</b>	<b>2-nodes</b>	<b>3-nodes</b>	<b>4-nodes</b>
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

