Data structures and algorithms

Part 9

Searching and Search Trees II

Petr Felkel

10.12. 2007

Topics

Red-Black tree

- Insert
- Delete

B-Tree

- Motivation
- Search
- Insert
- Delete

Based on:
[Cormen, Leiserson, Rivest: Introduction to Algorithms, Chapter 14 and 19, McGraw Hill, 1990]
[Whitney: CS660 Combinatorial Algorithms, San Diego State University, 1996]
[Frederic Maire: An Introduction to Btrees, Queensland University of Technology,1998]

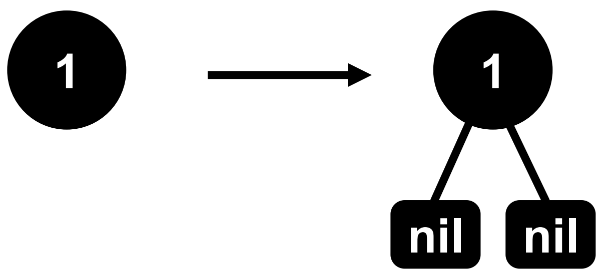
Red-Black tree

Approximately balanced BST

$h_{RB} \leq 2x\ h_{BST}$ (height ≤ 2x height of balanced tree)

Additional bit for COLOR = {red | black}

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



leaf ⟶ inner node

Red-Black tree

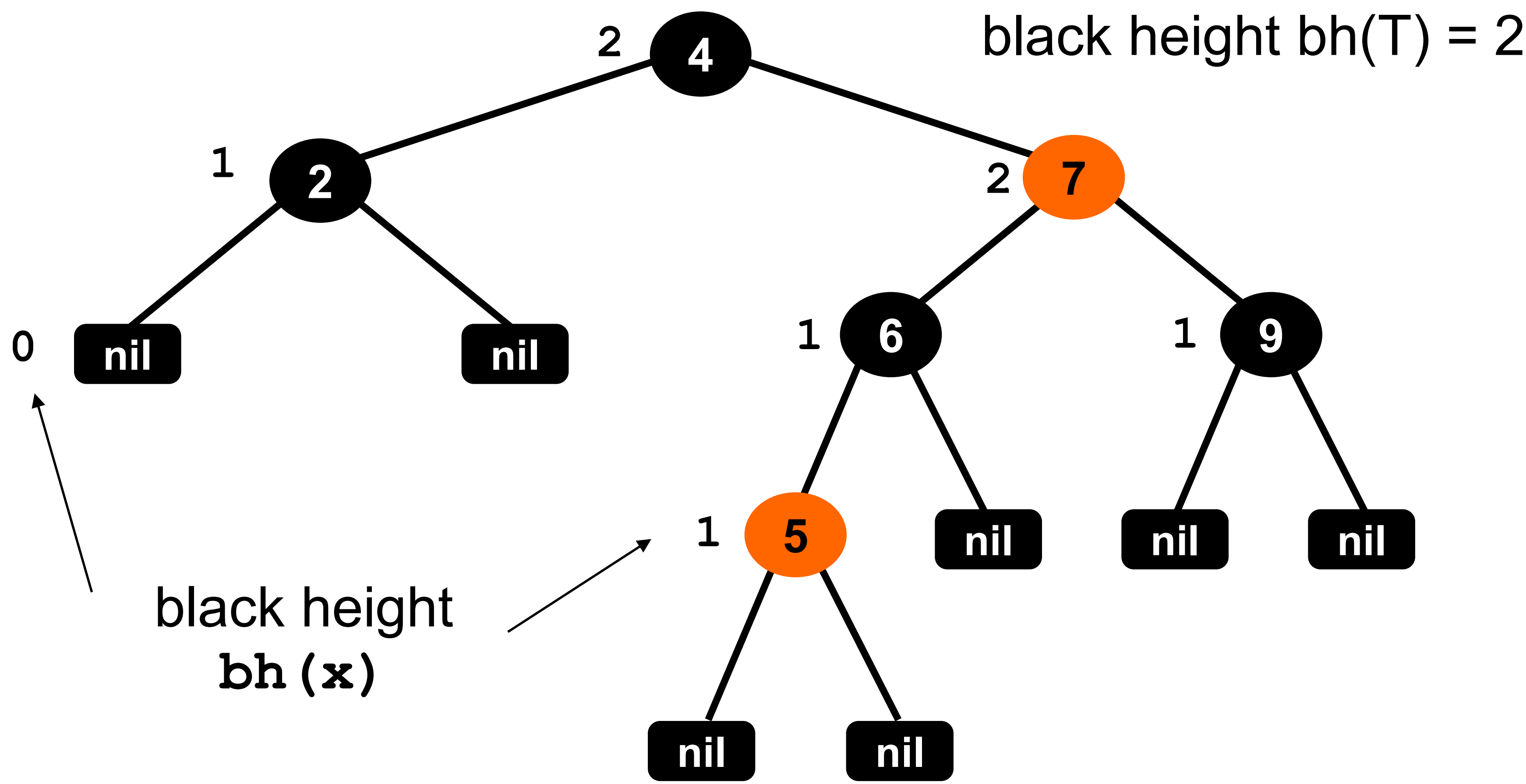
A binary search tree is a red-black tree if:

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

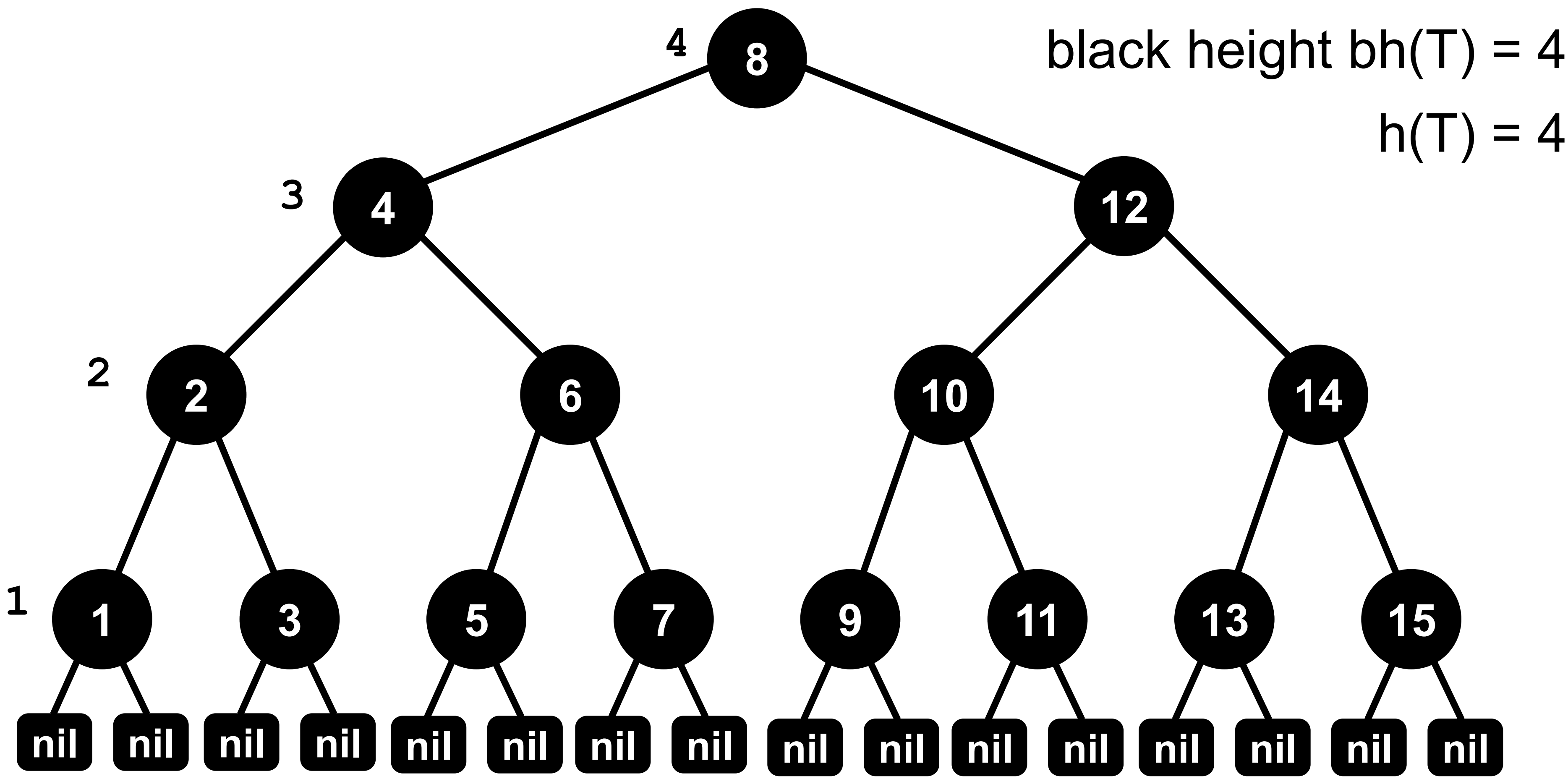
(5. Root is black)

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**

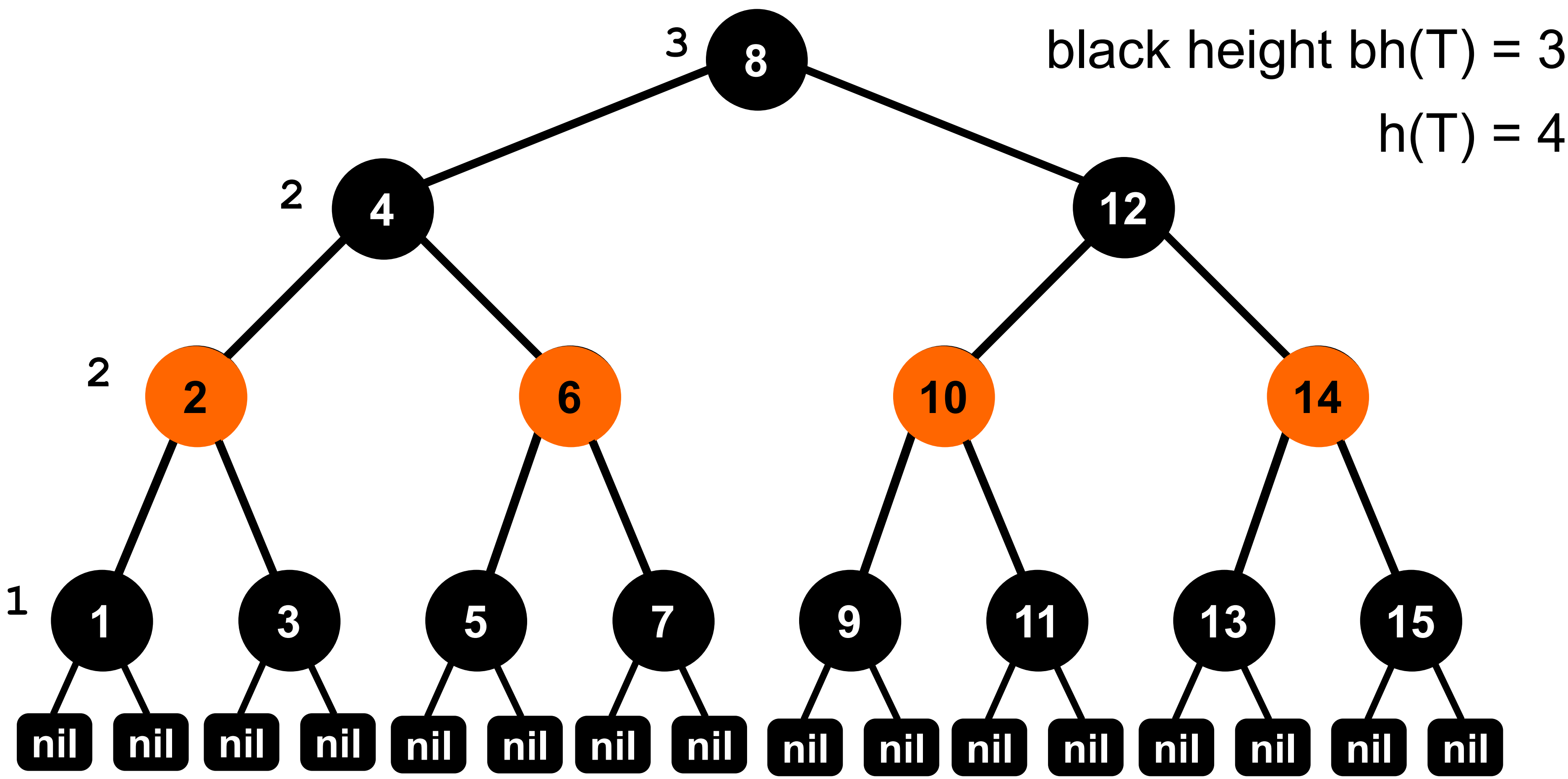
Red-Black tree



Binary Search Tree -> RB Tree



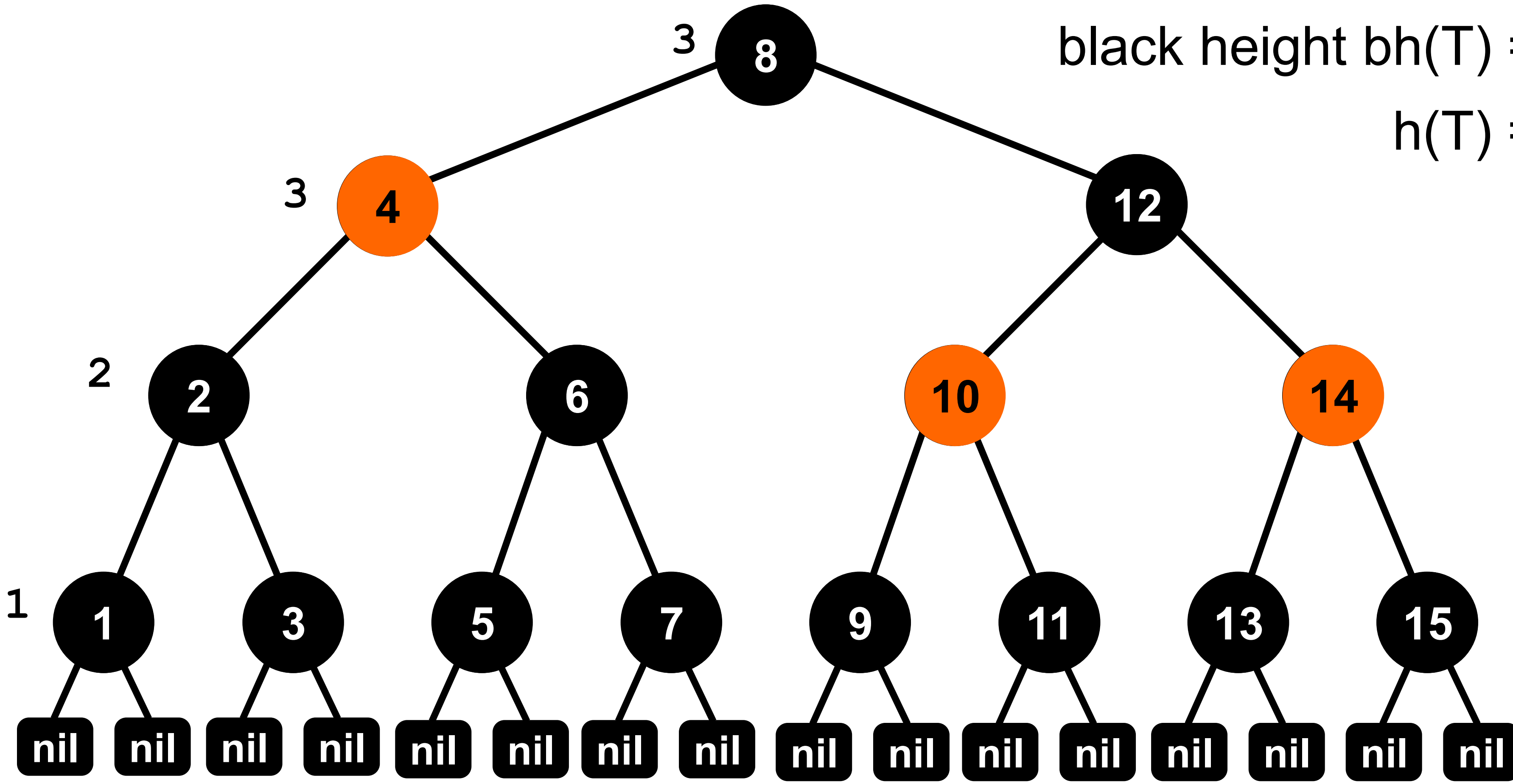
Binary Search Tree -> RB Tree



Binary Search Tree -> RB Tree

black height bh(T) = 3

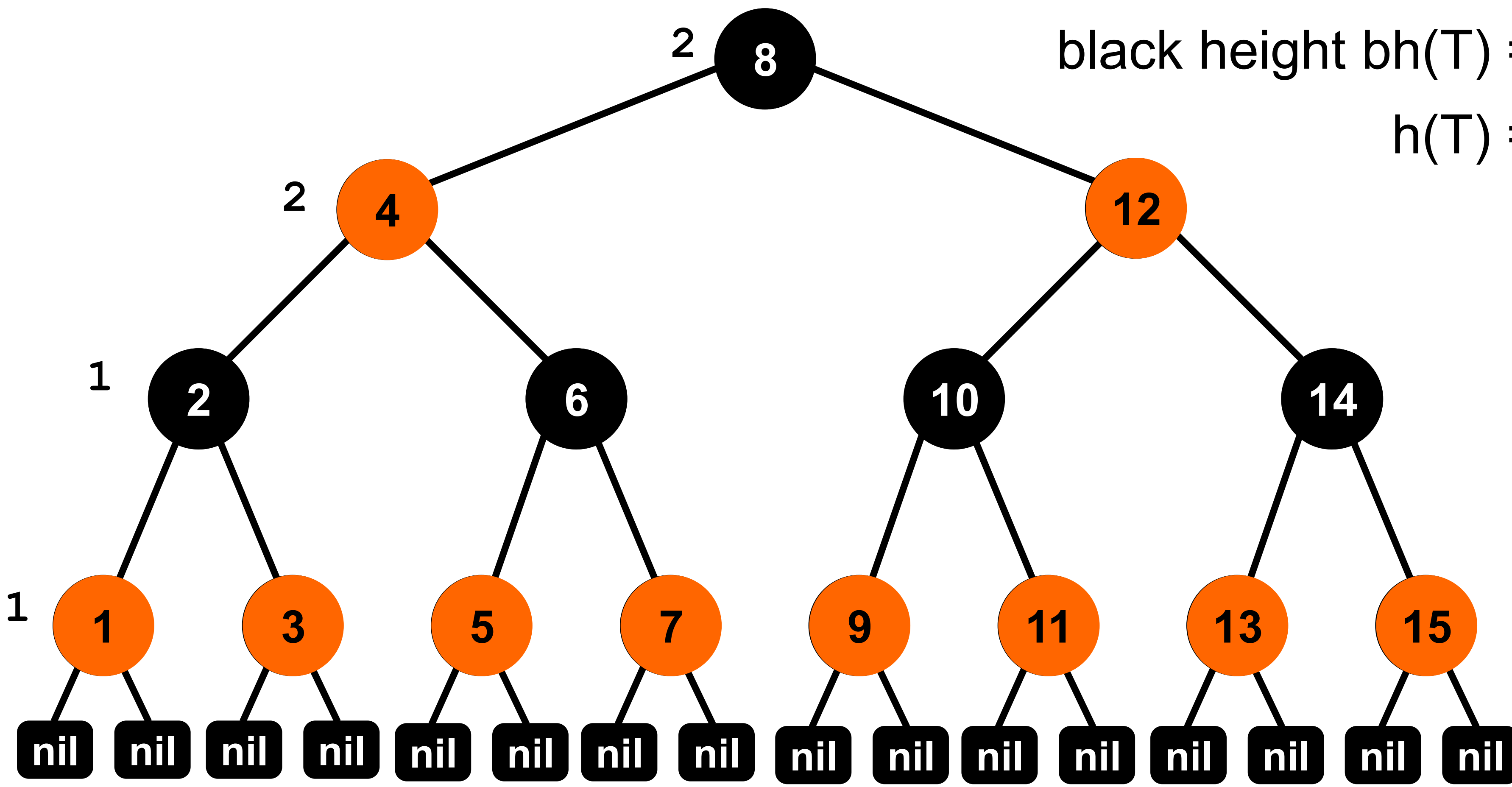
h(T) = 4



Binary Search Tree -> RB Tree

black height bh(T) = 2

h(T) = 4



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 ½ of nodes red => bh(x) ≈ ½ h(x), h(x) ≈ 2 lg(n+1)
  - if all nodes black => bh(x) = h(x) = lg(n+1) - 1

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

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

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 $2^{bh(x)}-1$ *internal nodes*.

By induction on height of x:

I. If x is a *leaf*, then $bh(x) = 0$, $2^{bh(x)}-1 = 0$ internal nodes //… nil node

II. Consider x with height h *and* two children (with height h -1)

- x's children black-height is either $bh(x)$ -1 or $bh(x)$ // black or red
- Ind. hypothesis: x's children subtree has at least $2^{bh(x)-1} -1$ internal nodes
- So subtree starting at x contains at least $(2^{bh(x)-1} -1) + (2^{bh(x)-1} -1) + 1 = 2^{bh(x)} - 1$ internal nodes => proved

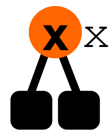
2. Let h = height of the tree rooted at x

- min ½ nodes are black on any path to leaf => $bh(x) \geq h / 2$
- Thus, $n \geq 2^{h/2} - 1 \iff n + 1 \geq 2^{h/2} \iff \lg(n+1) \geq h / 2$
- $h \leq 2\lg(n+1)$

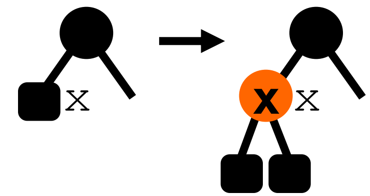
Inserting in Red-Black Tree

Color new node Red

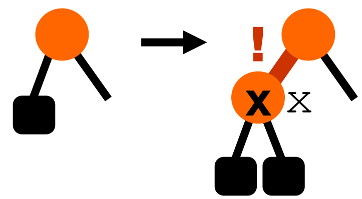
Insert it as in the standard BST



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



If parent 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

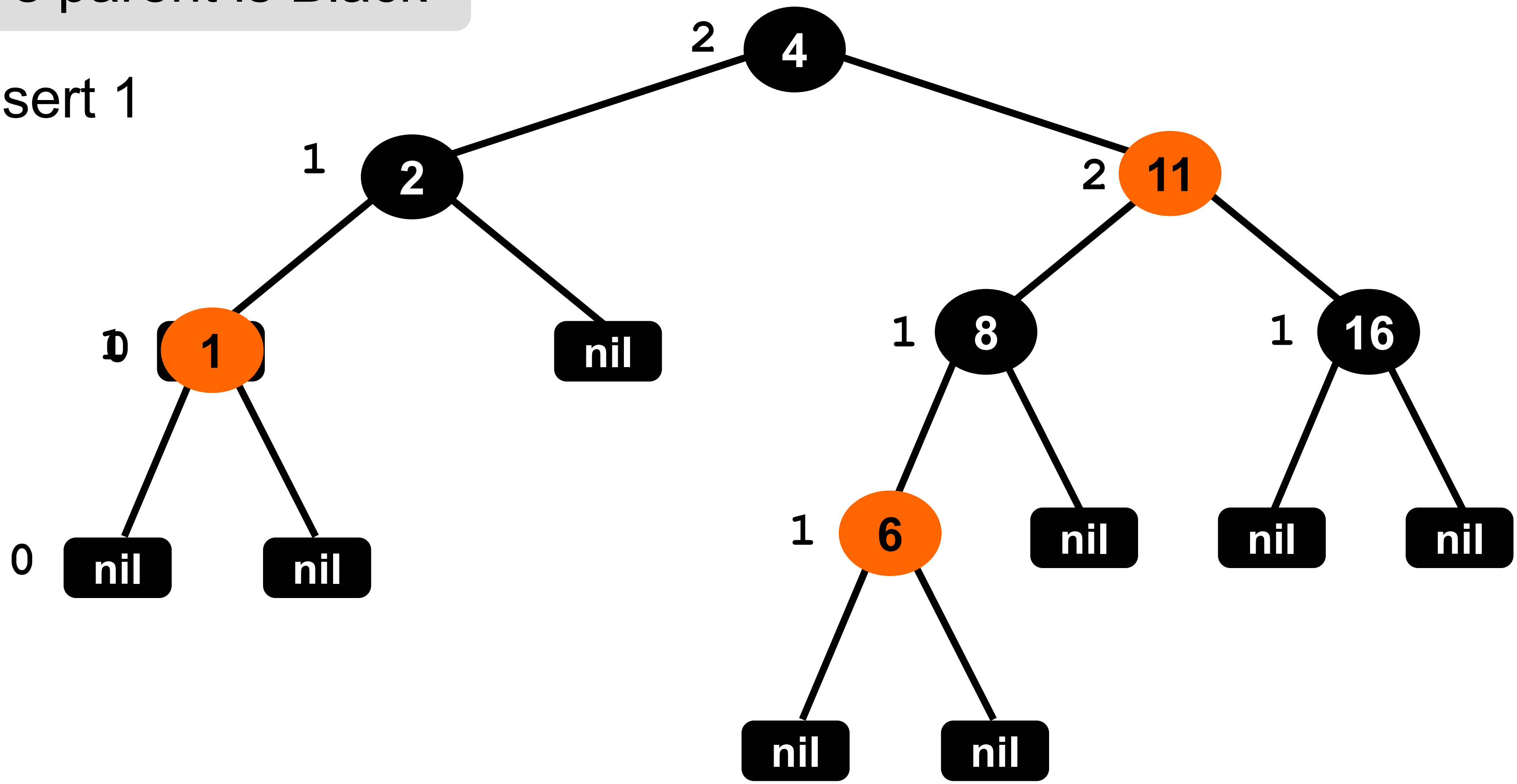
           case 3   // single rotation

Color root Black

Inserting in Red-Black Tree

x's parent is Black

Insert 1



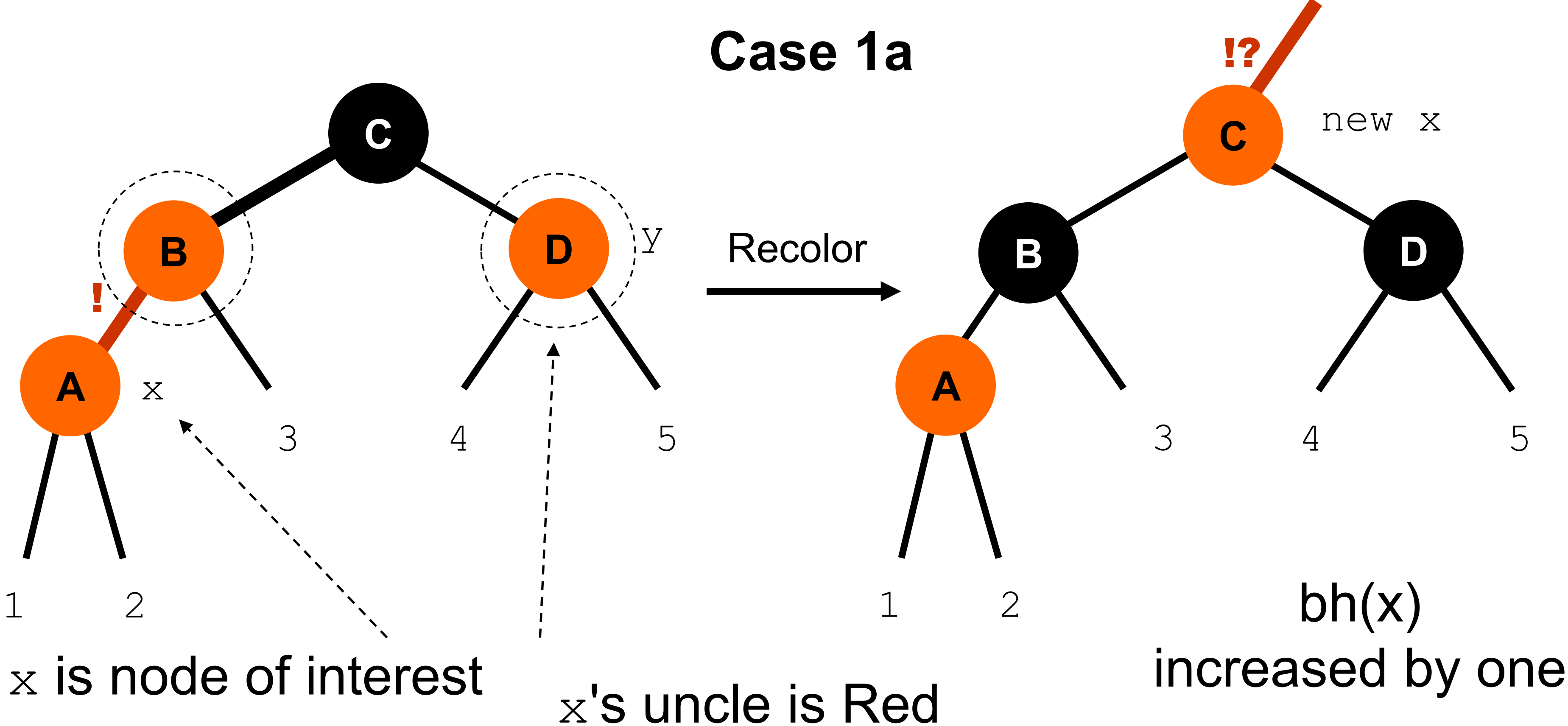
If parent is Black, stop. Tree is a Red-Black tree.

Inserting in Red-Black Tree

x's parent is Red

x's <u>uncle</u> y is <u>Red</u>

x is a Left child

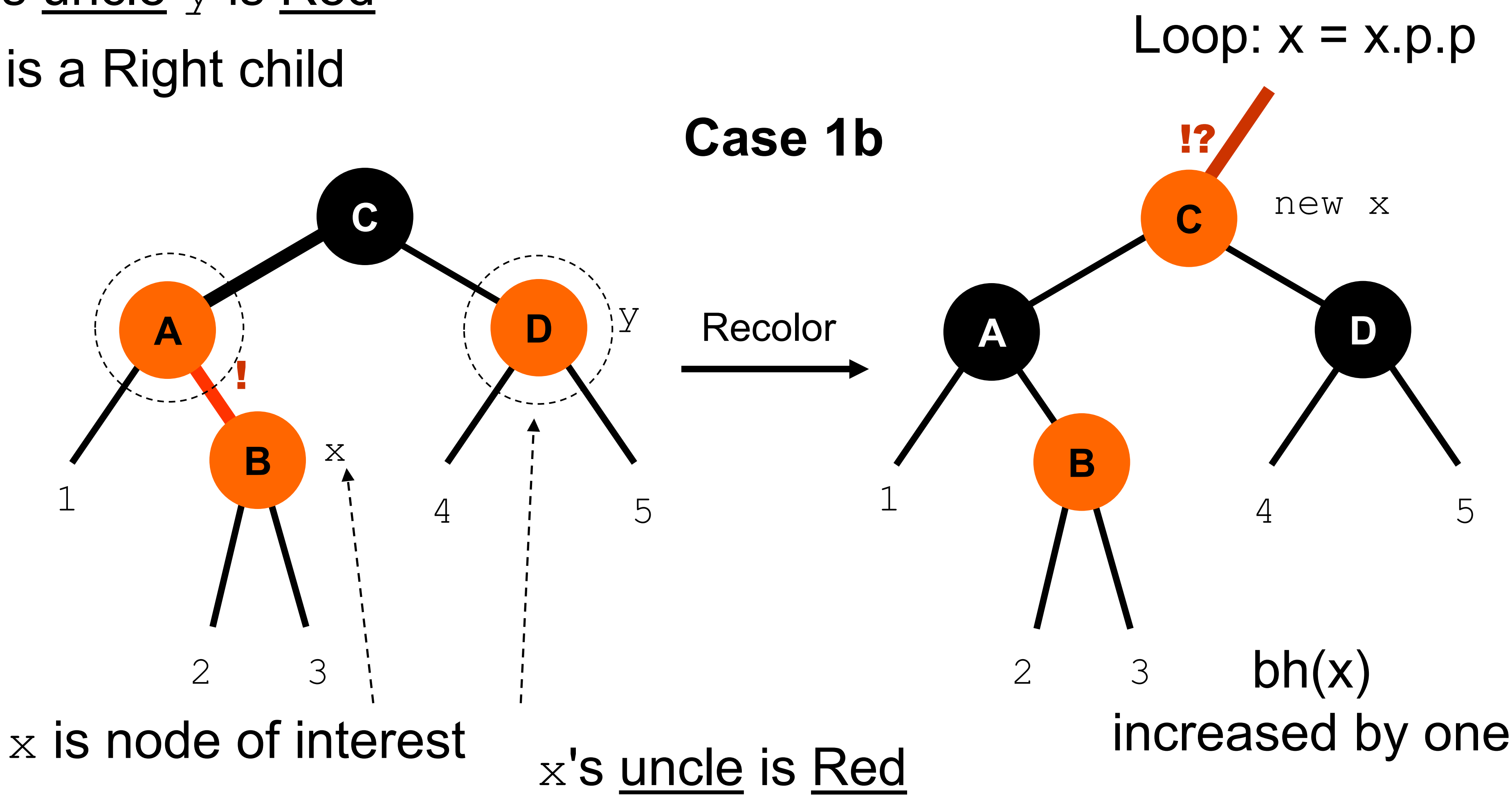


Inserting in Red-Black Tree

x's parent is Red

x's <u>uncle</u> y is <u>Red</u>

x is a Right child



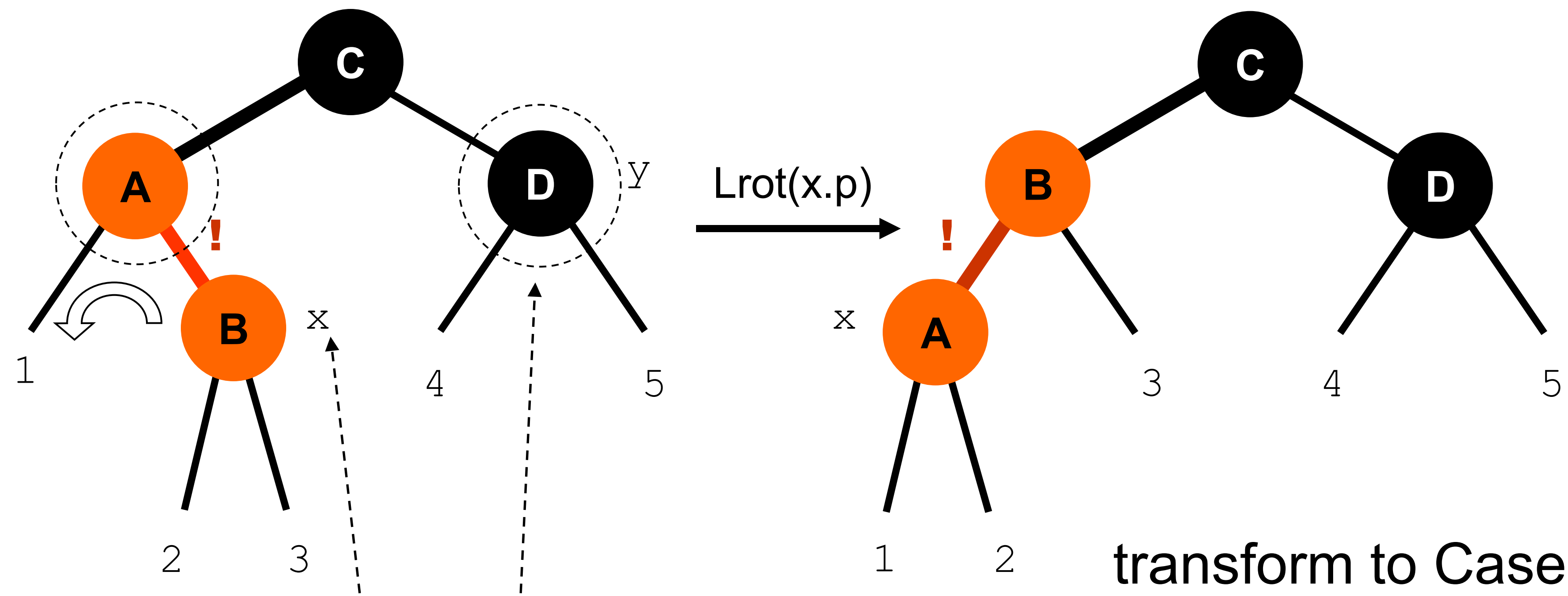
Inserting in Red-Black Tree

x's parent is Red

x's uncle y is Black

x is a Right child

Case 2



transform to Case 3

Inserting in Red-Black Tree

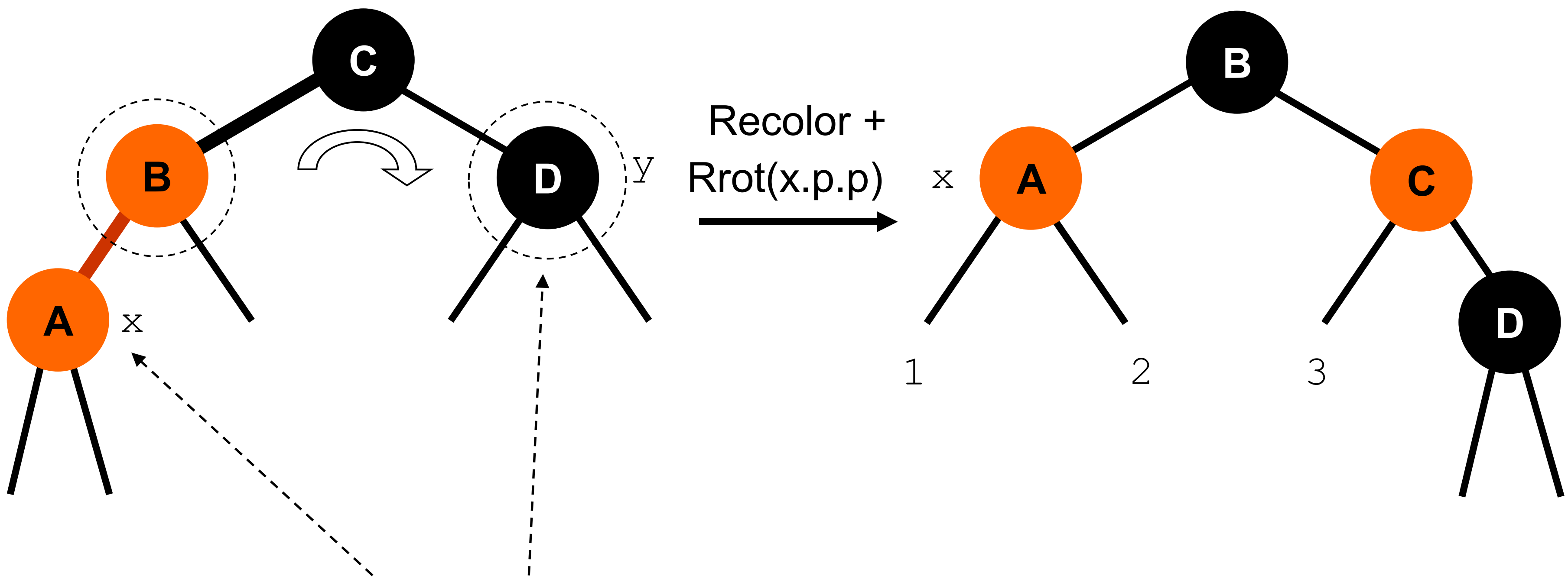
x's parent is Red

x's uncle y is Black

x is a Left child

Case 3

Terminal case, tree is a Red-Black tree

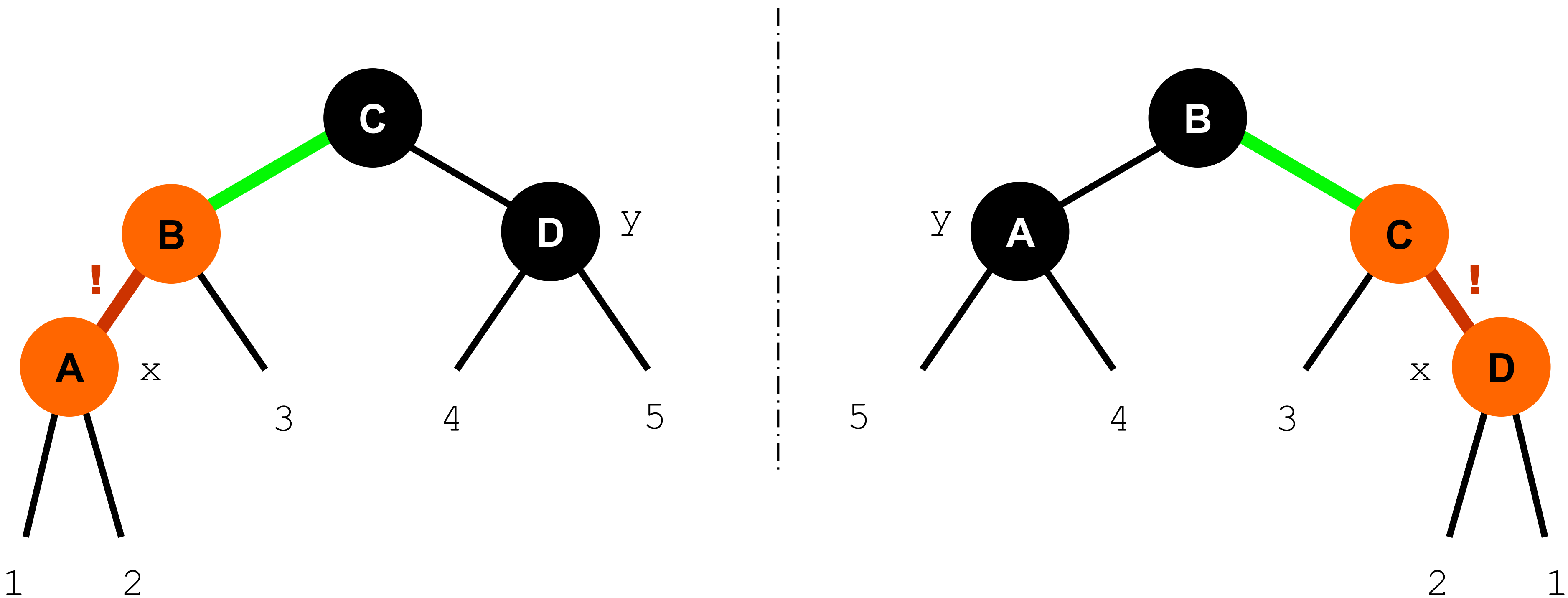


x is a Left child

x's uncle is Black

Inserting in Red-Black Tree

Cases Right from the grandparent
are symmetric



RB-INSERT(T, x)

```
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
      do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                 if color[y] = RED
                    then color[p[x]] ← BLACK            ▷ Case 1
                         color[y] ← BLACK               ▷ Case 1
                         color[p[p[x]]] ← RED           ▷ Case 1
                         x ← p[p[x]]                    ▷ Case 1
                    else if x = right[p[x]]
                            then x ← p[x]               ▷ Case 2
                                 LEFT-ROTATE(T, x)      ▷ Case 2
                         color[p[x]] ← BLACK            ▷ Case 3
                         color[p[p[x]]] ← RED           ▷ Case 3
                         RIGHT-ROTATE(T, p[p[x]])       ▷ Case 3
            else (same as then clause
                    with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

p[x] = parent of x
left[x] = left son of x
y = uncle of x

Red uncle y ->recolor up

[Cormen90]

Inserting in Red-Black Tree

Insertion in O(log(n)) time

Requires at most two rotations

DEMO: http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html

(Intuitive, good for understanding)

http://reptar.uta.edu/NOTES5311/REDBLACK/RedBlack.html

(little different order of re-coloring and 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**
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)

Deleting in Red-Black Tree

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

// move x with virtual black mark A through the tree

// just recolor or rotate other subtree up (decrease bh in R subtree)

if(red sibling)

-> Case 1: Rotate right subtree up, color sibling black, and continue in left subtree with new sibling

if(black sibling 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

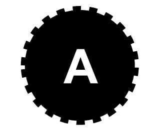
Case 4: Rotate right subtree up

}

Deleting in R-B Tree - Case 1

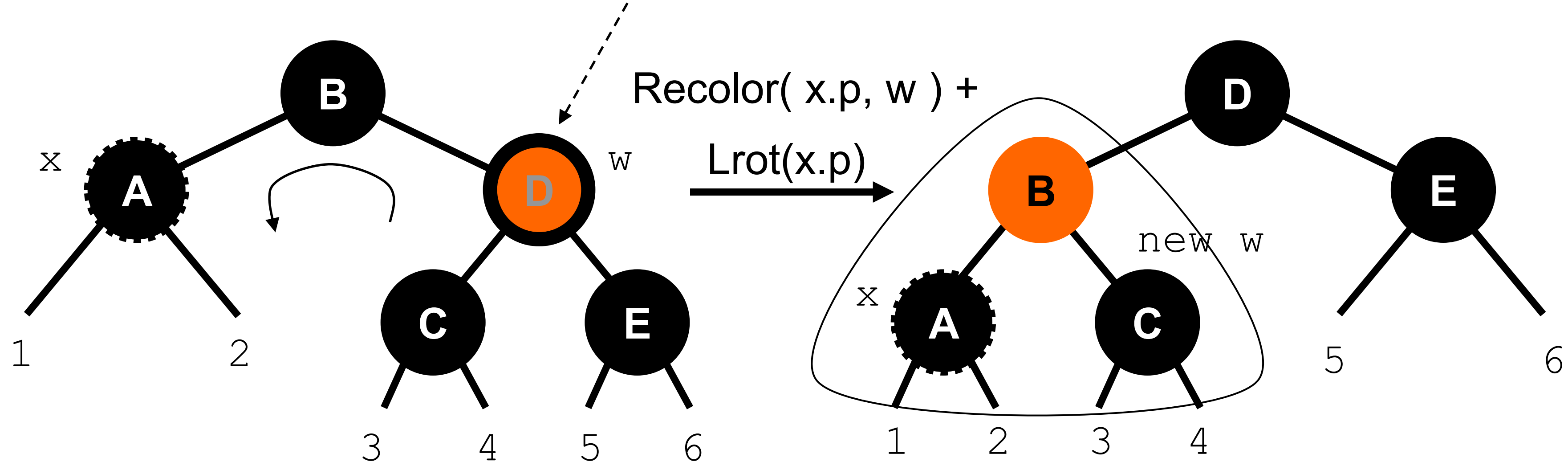
x is the child of the physically deleted black node => double black

x's <u>sibling</u> w (*sourozenec*) is <u>red</u>



(x's parent MUST be black)

Case 1

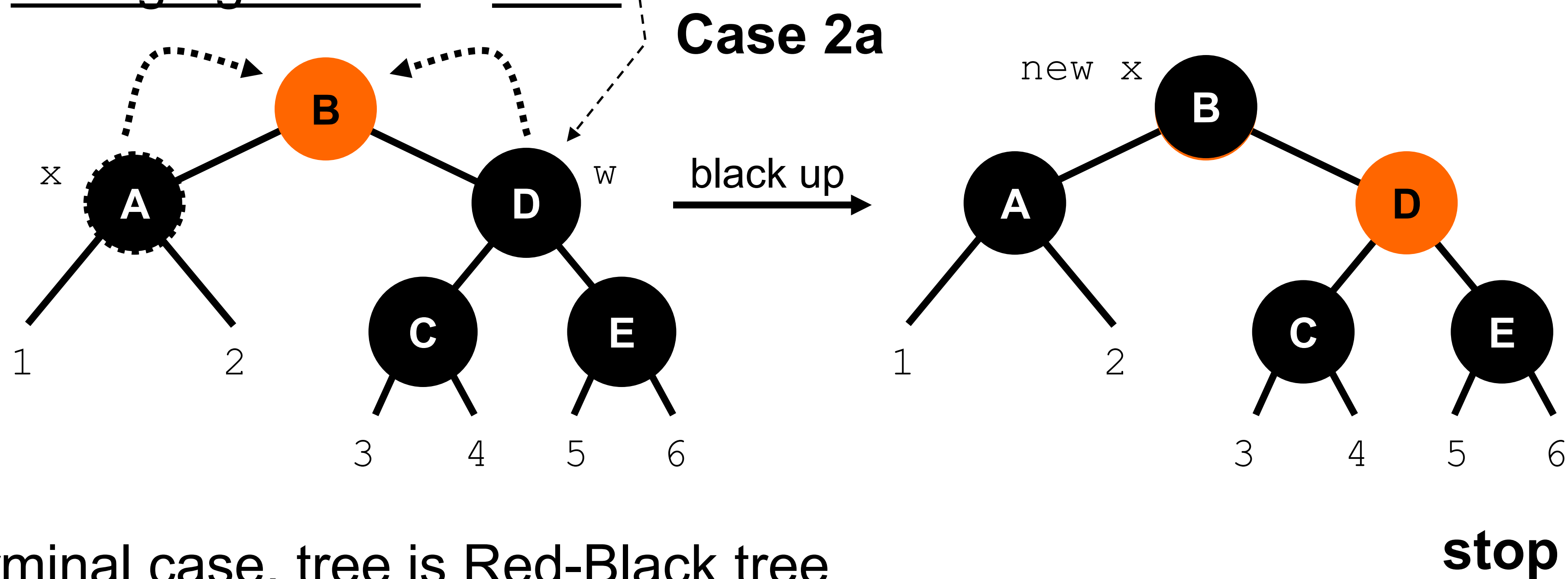


x stays at the same black height

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

Deleting in R-B Tree - Case 2a

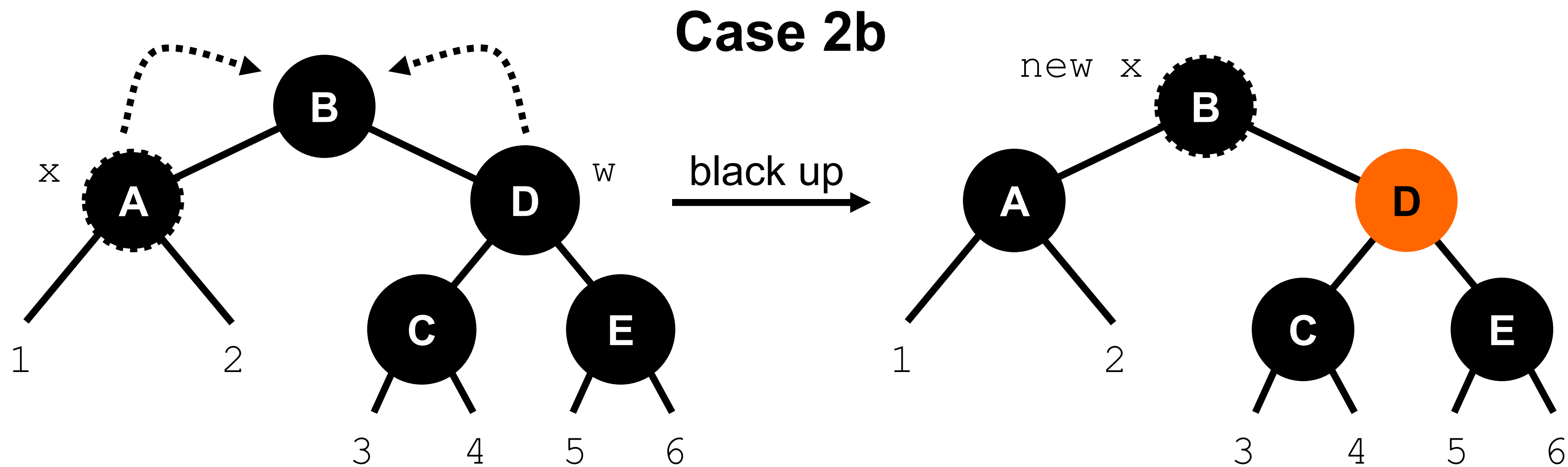
x's sibling w is black
x's parent is red
x's sibling left child is black
x's sibling right child is black



Terminal case, tree is Red-Black tree

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

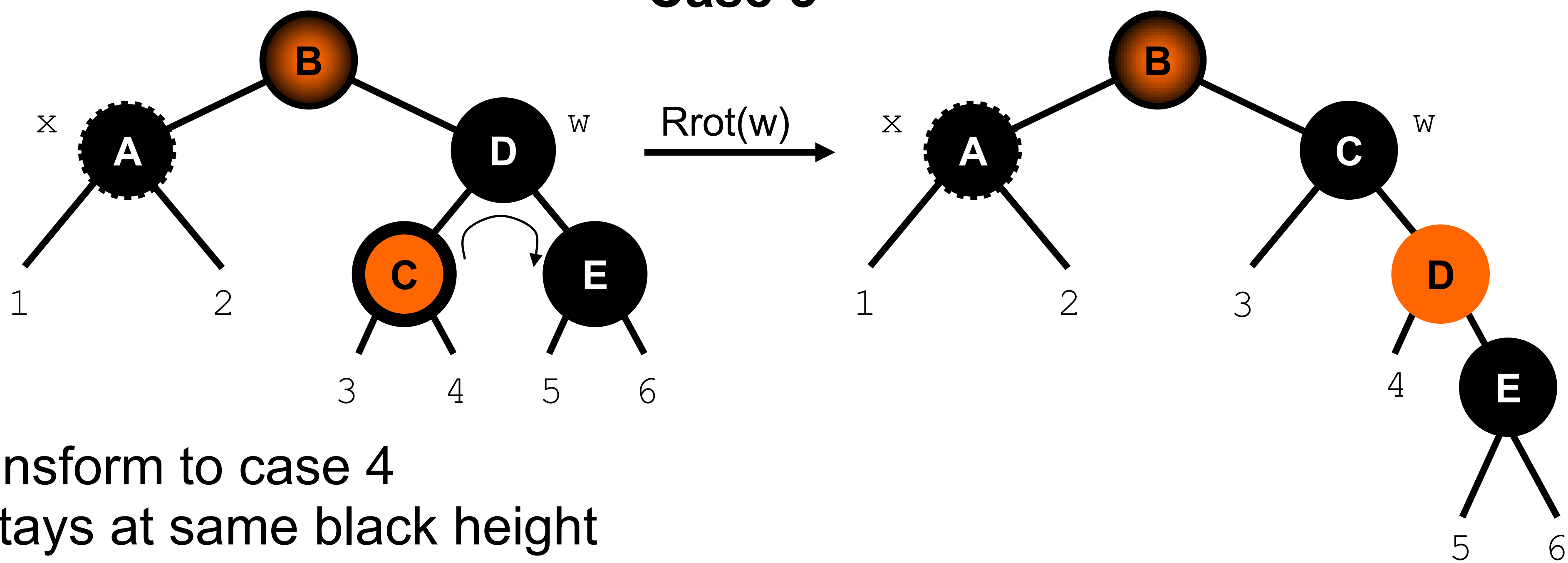


Decreases x black height by one

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 // blocks coloring w red
x's sibling right child is black

Case 3

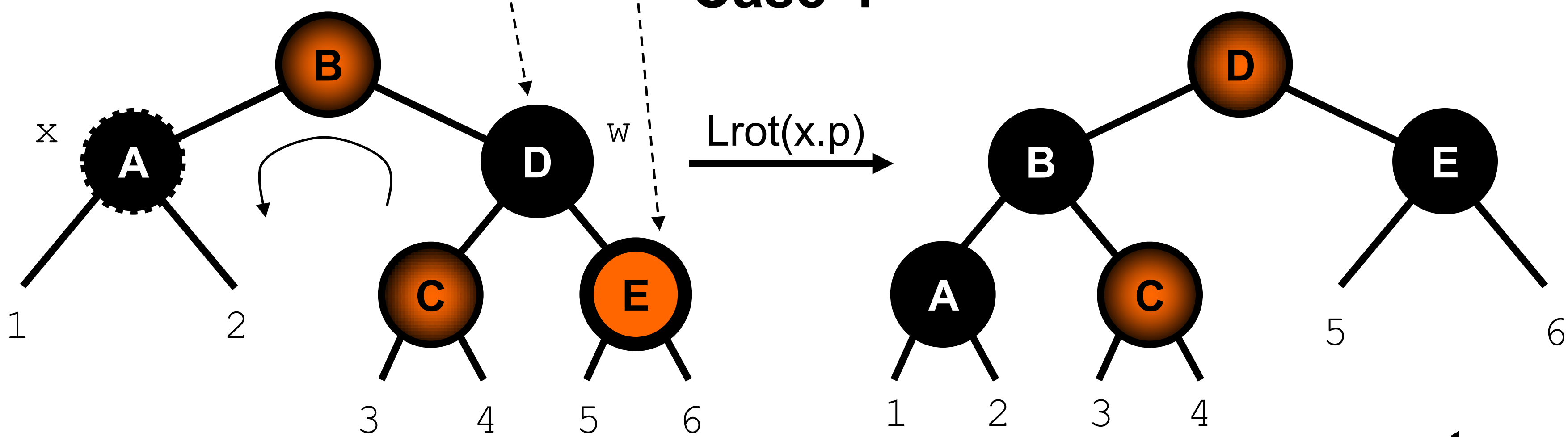


Transform to case 4
x stays at same black height

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   // blocks coloring w red

Case 4



stop

Terminal case, tree is Red-Black tree

Deleting in Red-Black Tree

RB-DELETE(*T*, *z*)

```
if left[z] = nil[T] or right[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
    then x ← left[y]
    else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
         ▷ If y has other fields, copy them, too.
if color[y] = BLACK
    then RB-DELETE-FIXUP(T, x)
return y
```

Notation similar to AVL

z = *logically* removed

y = *physically* removed

x = y's only son

[Cormen90]

RB-DELETE-FIXUP(T, x)

```
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                  then color[w] ← BLACK                          ▷ Case 1
                       color[p[x]] ← RED                         ▷ Case 1
                       LEFT-ROTATE(T, p[x])                      ▷ Case 1
                       w ← right[p[x]]                           ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                  then color[w] ← RED                            ▷ Case 2
                       x ← p[x]                                  ▷ Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] ← BLACK            ▷ Case 3
                               color[w] ← RED                    ▷ Case 3
                               RIGHT-ROTATE(T, w)                ▷ Case 3
                               w ← right[p[x]]                   ▷ Case 3
                       color[w] ← color[p[x]]                    ▷ Case 4
                       color[p[x]] ← BLACK                       ▷ Case 4
                       color[right[w]] ← BLACK                   ▷ Case 4
                       LEFT-ROTATE(T, p[x])                      ▷ Case 4
                       x ← root[T]                               ▷ Case 4
          else (same as then clause
                  with "right" and "left" exchanged)
color[x] ← BLACK
```

x = *son* of removed node
p[*x*] = parent of *x*
w = sibling (brother) of *x*

Case 1: R subtree up, Check L

Case 2: Recolor, Black up, Go up

Case 3: inner R-subtree up

Case 4: R subtree up, stop

[Cormen90]

Deleting in R-B Tree

Delete time is O(log(n))
At most three rotations are done

Which BS tree is the best? [Pfaff 2004]

It is data dependent

- For random sequences

  => use *unsorted tree*, no waste time for rebalancing

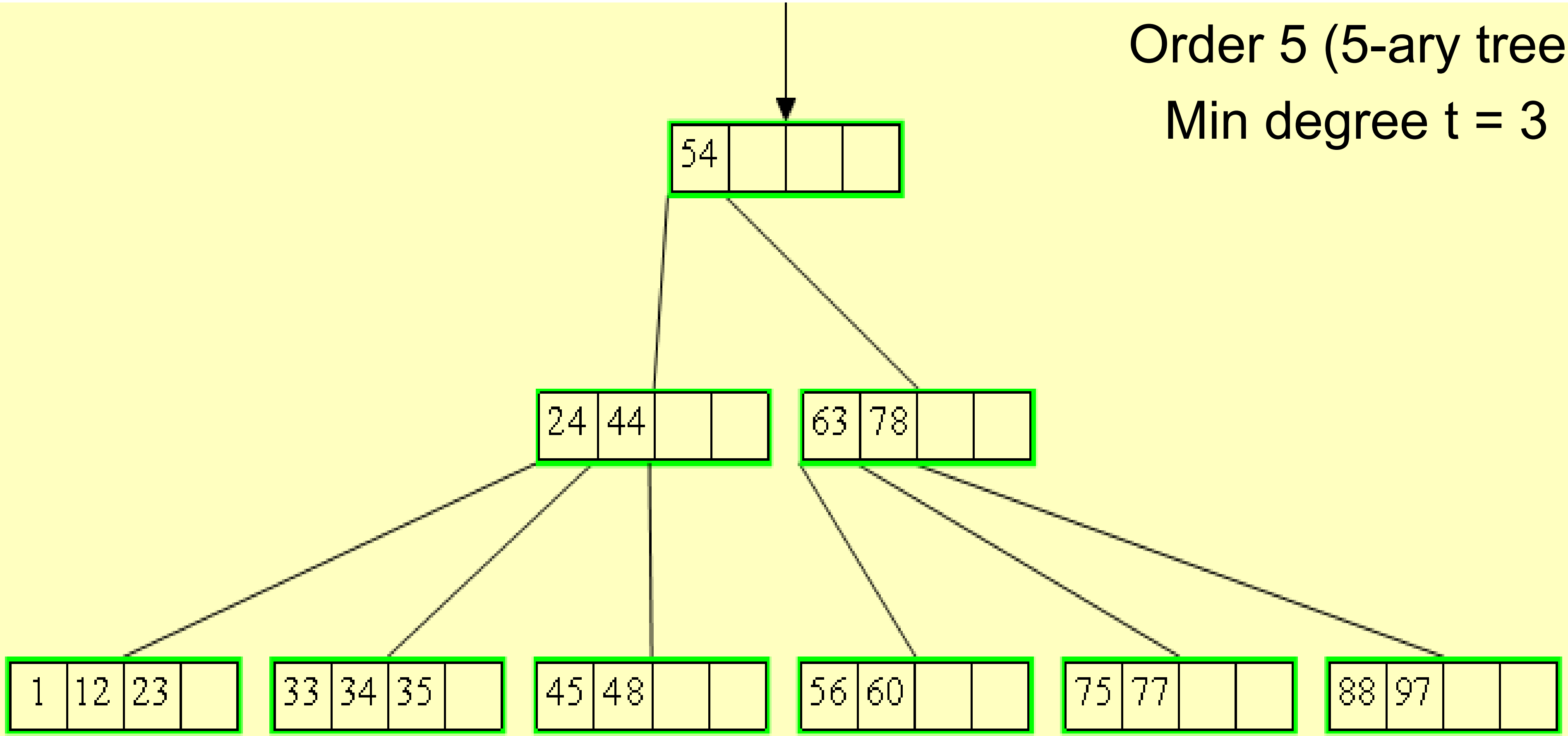
- For mostly random ordering with occasional runs of sorted order

  => use *red-black trees*

- For insertions often in a sorted order and
  - later accesses tend to be random => AVL trees
  - later accesses are sequential or clustered => splay trees
    - self adjusting trees,
    - update each search by moving searched element to the root

B-tree as BST on disk

B-tree



Order 5 (5-ary tree)

Min degree t = 3

Based on [Cormen] and [Maire]

B-tree

1. Motivation
2. Multiway search tree
3. B-tree
4. Search
5. Insert
6. Delete

B-tree

Motivation

- Large data do not fit into operational memory -> disk
- Time for disk access is limited by HW (Disk access = Disk-Read, Disk-Write)

DISK : 16 ms
Seek 8ms + rotational delay 7200rpm 8ms

Instruction:
800 MHz 1,25ns

- Disk access is MUCH slower compared to instruction
  - 1 disk access ~ 13 000 000 instructions!!!!
  - Number of disk accesses dominates the computational time

B-tree

Motivation

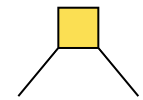
Disk access = Disk-Read, Disk-Write

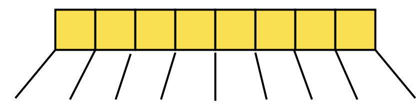
- Disk divided into blocks
  (512, 2048, 4096, 8192 bytes)
- Whole block transferred

- Design a *multiway search tree*
- Each node fits to one disk block

B-tree

Multiway search tree

= a generalization of Binary search tree  (*m*=2)

Each node has at most *m* children  (*m*>2)

Internal node with *n* keys has *n*+1 successors, $n < m$

(except root)

Leaf nodes with no successors

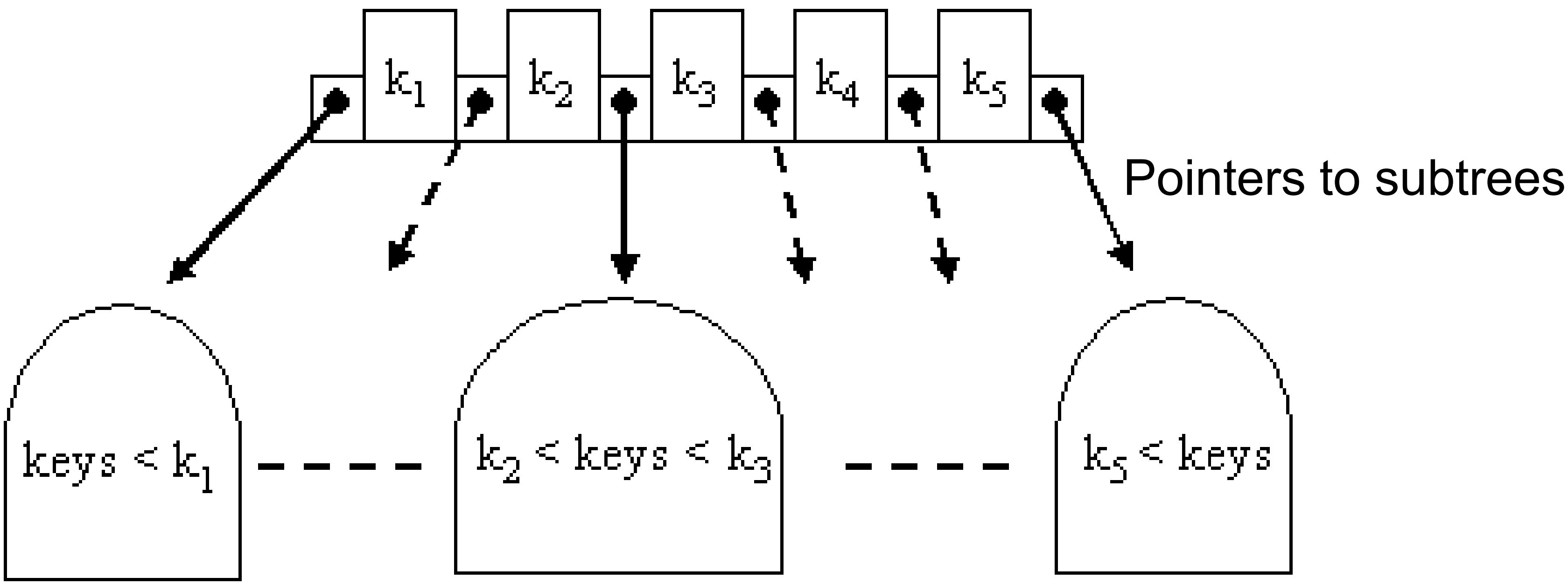
Tree is ordered %

Keys in nodes separates the ranges in subtrees %

B-tree

Multiway search tree – internal node

Keys in internal node separate the ranges of keys in subtrees

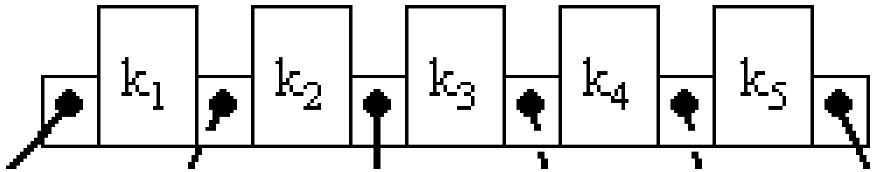


$$k_1 < k_2 < \ldots < k_5$$

B-tree

Multiway search tree – leaf node

Leaves have no subtrees and do not use pointers



Leaves have no pointers to subtrees

$$k_1 < k_2 < \dots < k_5$$

B-tree

B-tree

= of order *m* is an *m*-way search tree, such that

- All leaves have the same height (B-tree is balanced)
- All internal nodes are constrained to have
  - <u>at least</u> *m*/2 non-empty children and (precisely later)
  - <u>at most</u> *m* non-empty children
- The root can have 0 or between 2 to *m* children
  - 0 - leaf
  - *m* - a **full node**

B-tree

B-tree – problems with notation

Different authors use different names

- Order *m* B-tree
  - Maximal number of children
  - Maximal number of keys (No. of children - 1)
  - Minimal number of keys
- Minimum degree *t*
  - Minimal number of children [Cormen]

B-tree

B-tree – problems with notation

Relation between minimal and maximal number of children also differs

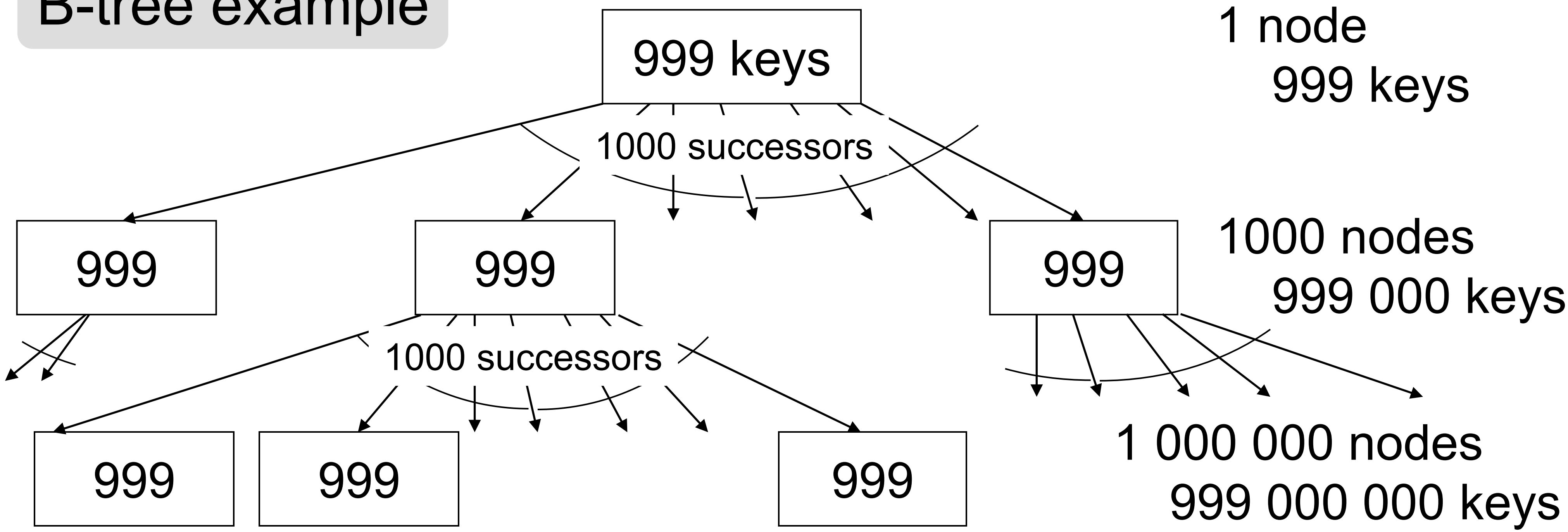
For minimal number *t* of children

Maximal number *m* of children is

- $m = 2t - 1$ simple B-tree, multiphase update strategy
- $m = 2t$ optimized B-tree, singlephase update strategy

B-tree

B-tree example



B-tree of order *m*=1000 of height 2 contains

1 001 001 nodes (1+1000 + 1 000 000)

999 999 999 keys ~ one billion keys (1 miliarda klíčů)

B-tree

B-tree node fields

n … number of keys k_i stored in the node $n < m$.

Node with $n = m$-1 is a **full-node**

k_i … n keys, stored in non-decreasing order

$k_1 \leq k_2 \leq \ldots \leq k_n$

leaf … boolean value, true for leaf, false for internal node

c_i … n+1=m pointers to successors (undefined for leaves)

Keys k_i separate the keys in subtree:

For $keys_i$ in the subtree with root k_i holds

$$keys_1 \leq k_1 \leq keys_2 \leq k_2 \leq \ldots \leq k_n \leq keys_{n+1}$$

B-tree

B-tree algorithms

- Search
- Insert
- Delete

B-tree search

Similar to BST tree search

Keys in nodes sequentially or binary search

Input: pointer to tree root and a key k

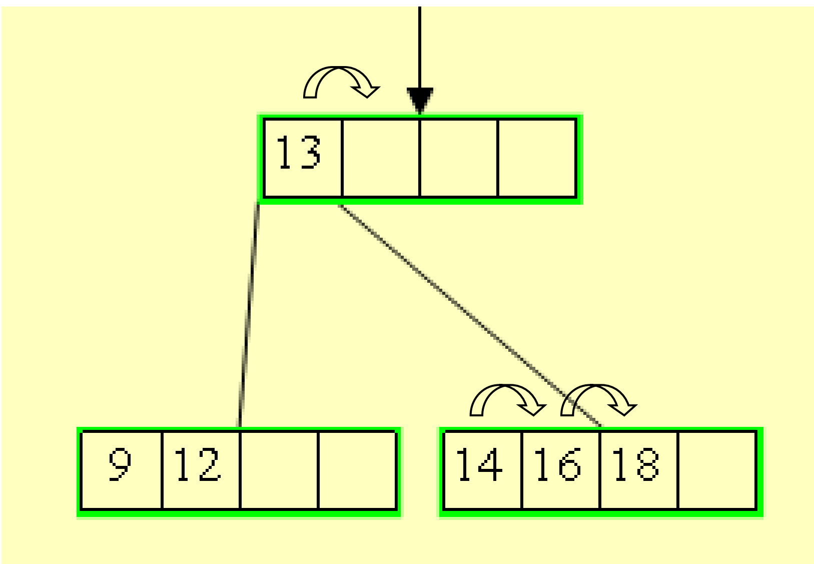
Output: an ordered pair (y, i), node y and index i

such that $y.k[i] = k$

or NIL, if k not found

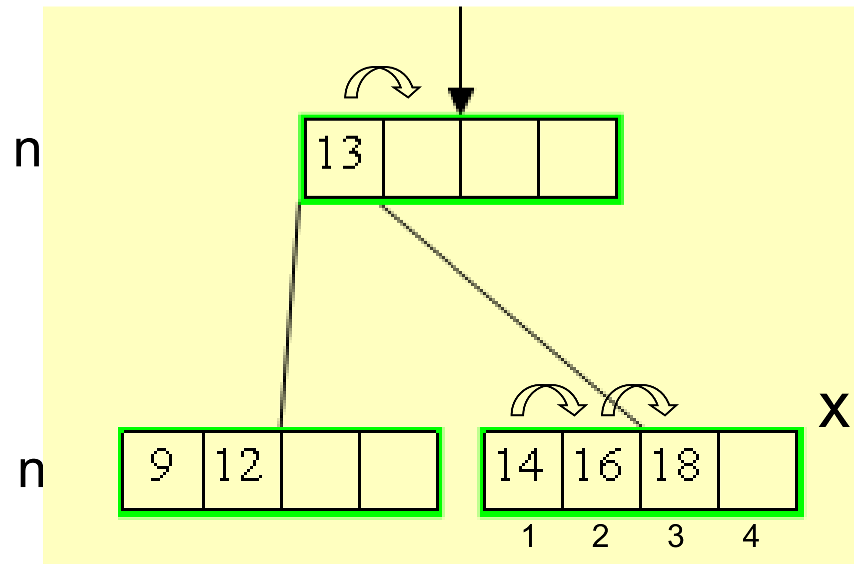
B-tree search

Search 17 (17)



17 not found => return NIL

Search 18 (18)



18 found => return (x, 3)

B-tree search

```
B-treeSearch(x,k)
   i ← 1
   while i ≤ x.n  and  k > x.k[i]      //sequential search
      do i ← i+1
   if i ≤ x.n and  k = x.k[i]
      return (x, i)                   // pair: node & index
   if x.leaf
      then return NIL
      else
           Disk-Read(x.c[i]) // tree traversal
           return B-treeSearch(x.c[i],k)
```

B-tree search

B-treeSearch complexity

Using tree order *m*

Number of disk pages read is

$O(h) = O(\log_m n)$

Where *h* is tree *height* and

m is the tree order

n is number of tree nodes

Since num. of keys $x.n < m$, the while loop takes $O(m)$ and

total time is $\boldsymbol{O(m \log_m n)}$

B-tree search

B-treeSearch complexity   Using minimum degree *t*

Number of disk pages read is

$$O(h) = O(\log_t n)$$

Where h is tree *height* and

t is the minimum degree of B-tree

n is number of tree nodes

Since num. of keys $x.n < 2t$, the while loop takes $O(t)$ and

total time is $\boldsymbol{O(t \log_t n)}$

B-tree update strategies

Two principal strategies

1. Multiphase strategy
   "solve the problem, when appears"   m=2t-1 children

2. Single phase strategy [Cormen]
   "avoid the future problems"   m =2t children

Actions:

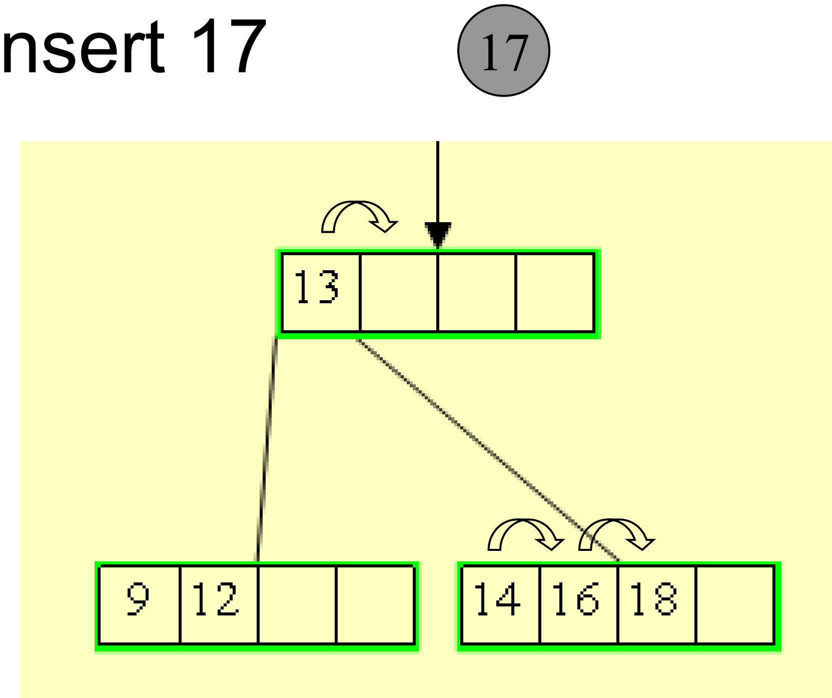
Split full nodes

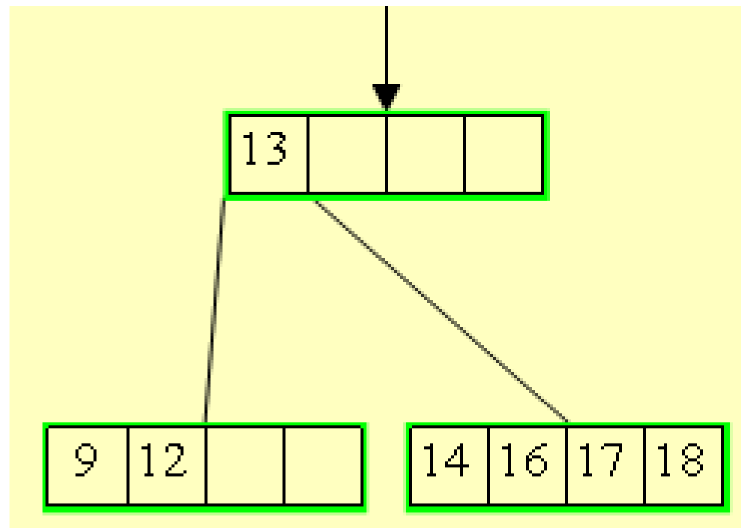
Merge nodes with less than minimum entries

B-tree insert - 1.Multiphase strategy

Insert to a **non-full** node

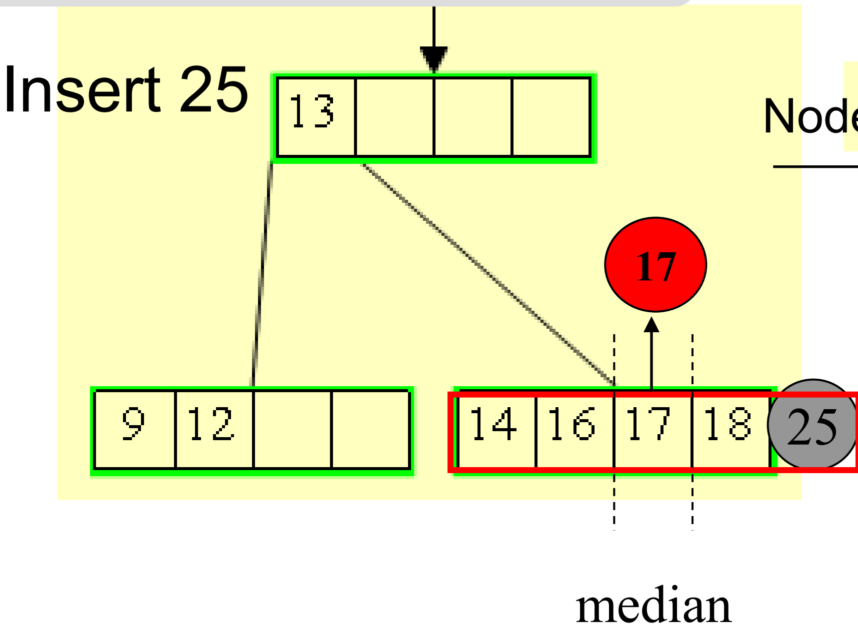
Insert 17

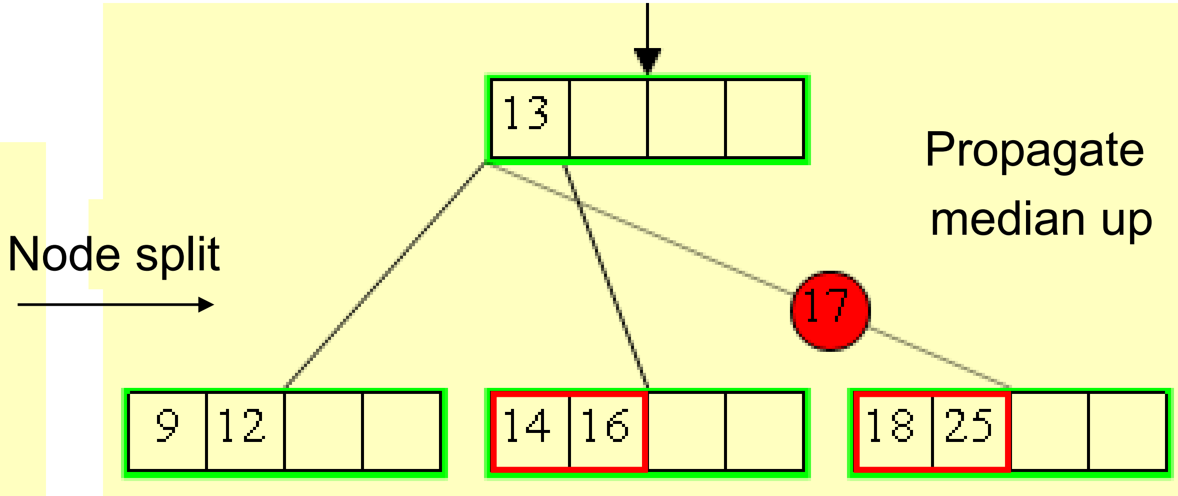


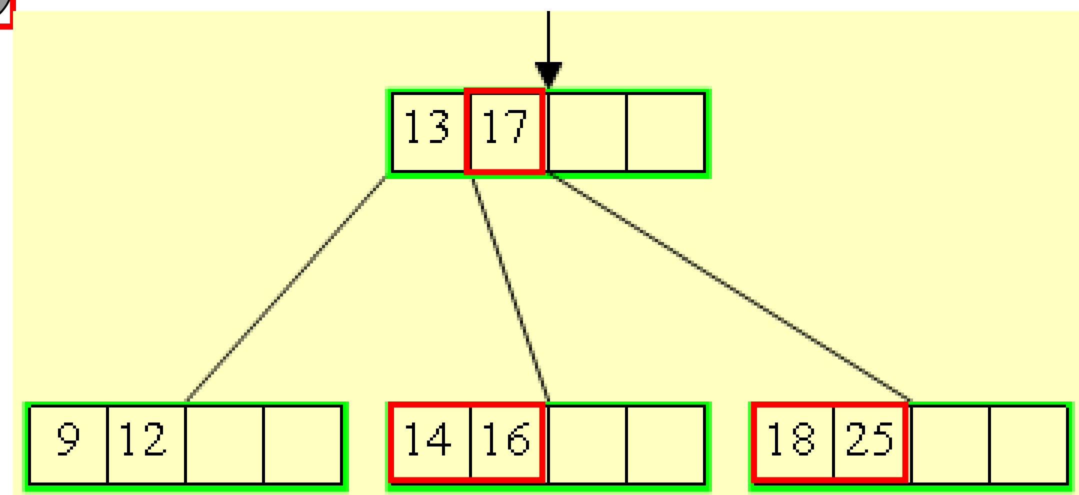


B-tree insert - 1.Multiphase strategy

Insert to a **full node**







1.Multiphase strategy
“solve the problem, when appears”

B-tree insert - 1.Multiphase strategy

Insert (x, T) - pseudocode x…key, T…tree

Find the leaf for *x* — Top down phase

If not full, insert *x* and stop

while (current_node full) (node overflow)

- find median (in keys in the node after insertion of x)
- split node into two — Bottom-up phase
- promote median up as new *x*
- current_node = parent of current_node or new root

Insert *x* and stop

B-tree insert - 2.Singlephase strategy

Principle: “avoid the future problems”

Top down phase only

- Split the full node with 2t-1 keys when enter
- It creates space for future medians from the children
- No need to go bottom-up

- Splitting of
  - Root => tree grows by one
  - Inner node or leaf => parent gets median key

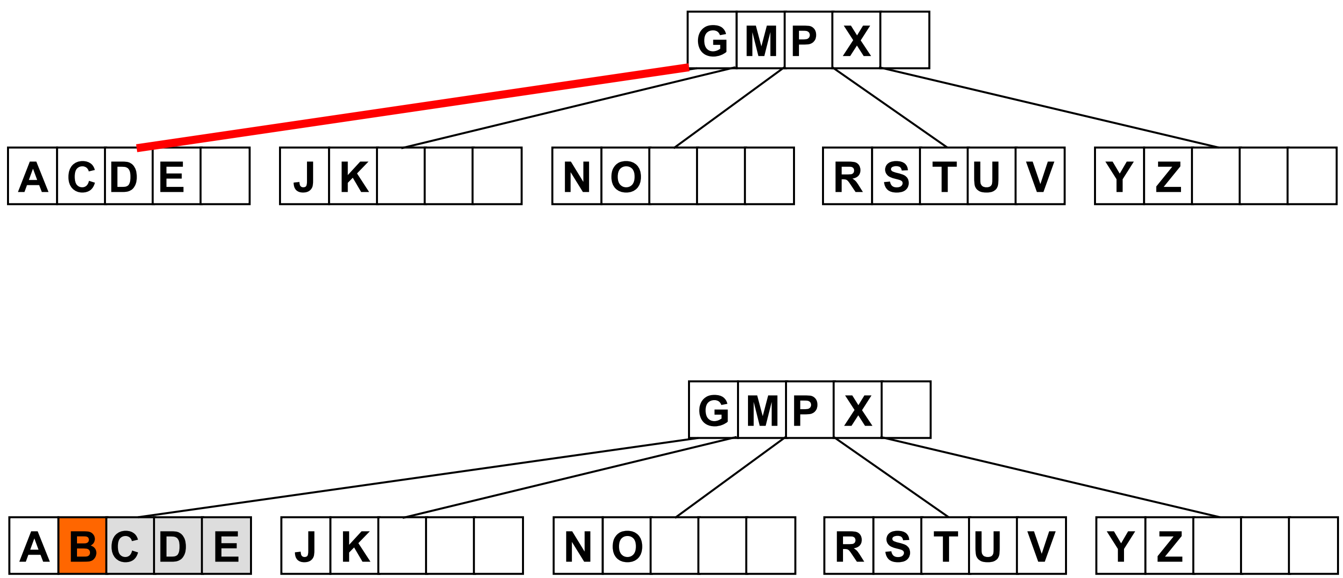
B-tree insert - 2.Singlephase strategy

Insert to a **non-full** node

m = 2*t* = 6 *children*

m-1 keys = odd max number

Insert B

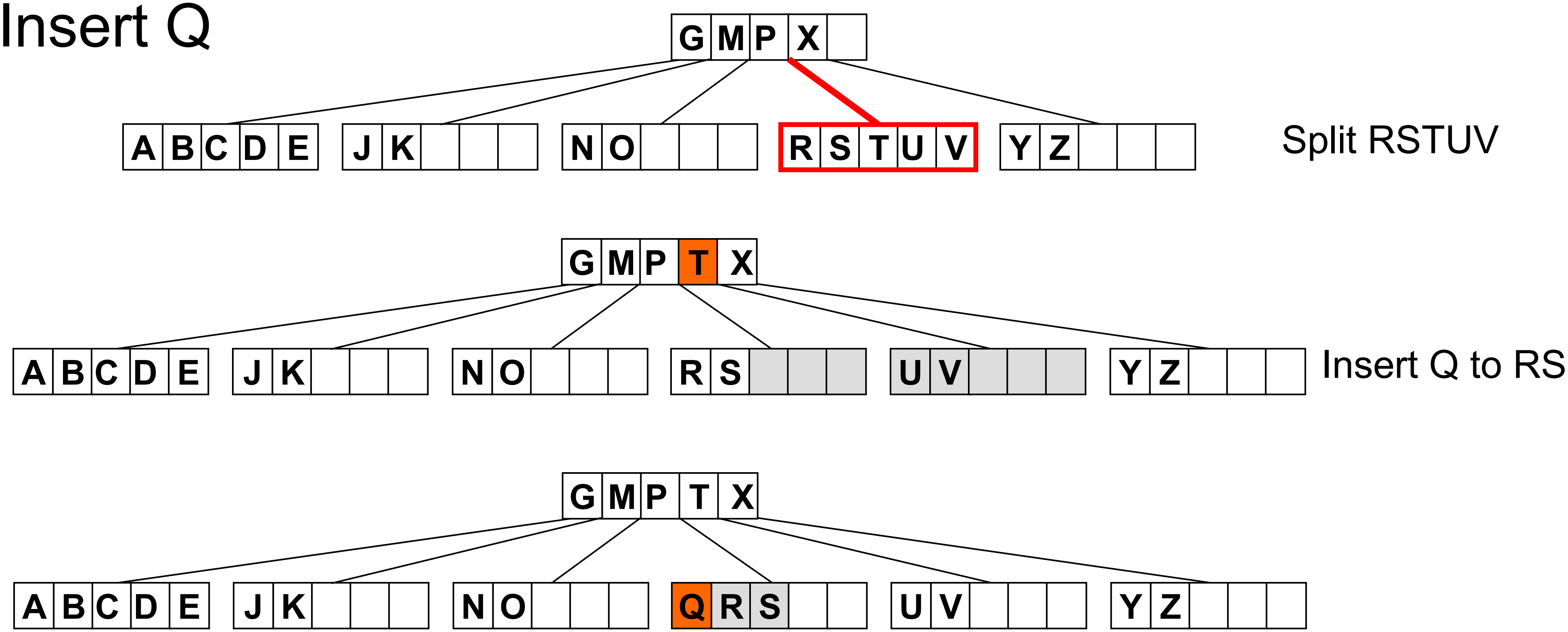


B-tree insert - 2.Singlephase strategy

1 new node

Splitting a passed **full node** and insert to a **not full** node

Insert Q

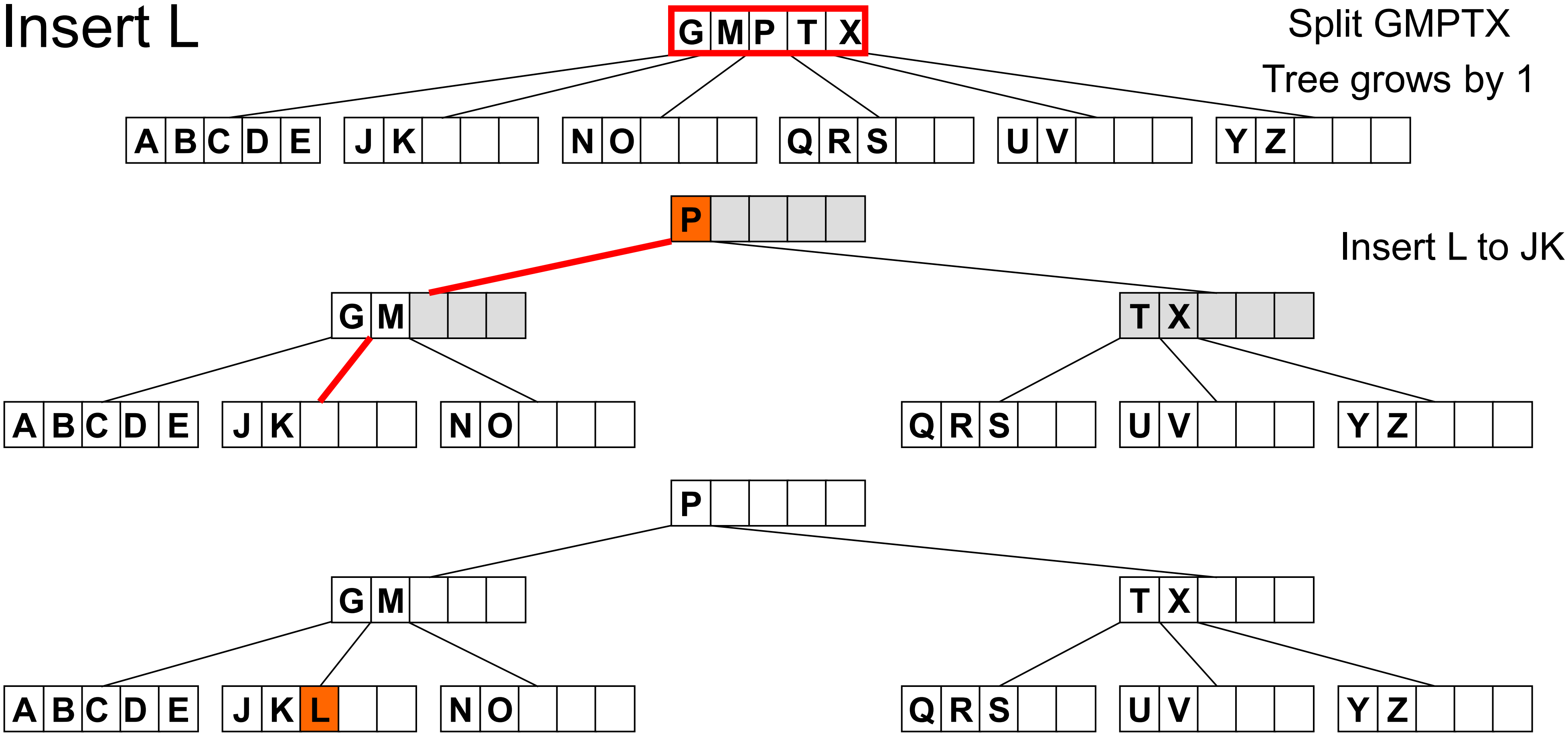


B-tree insert - 2.Singlephase strategy

2 new nodes

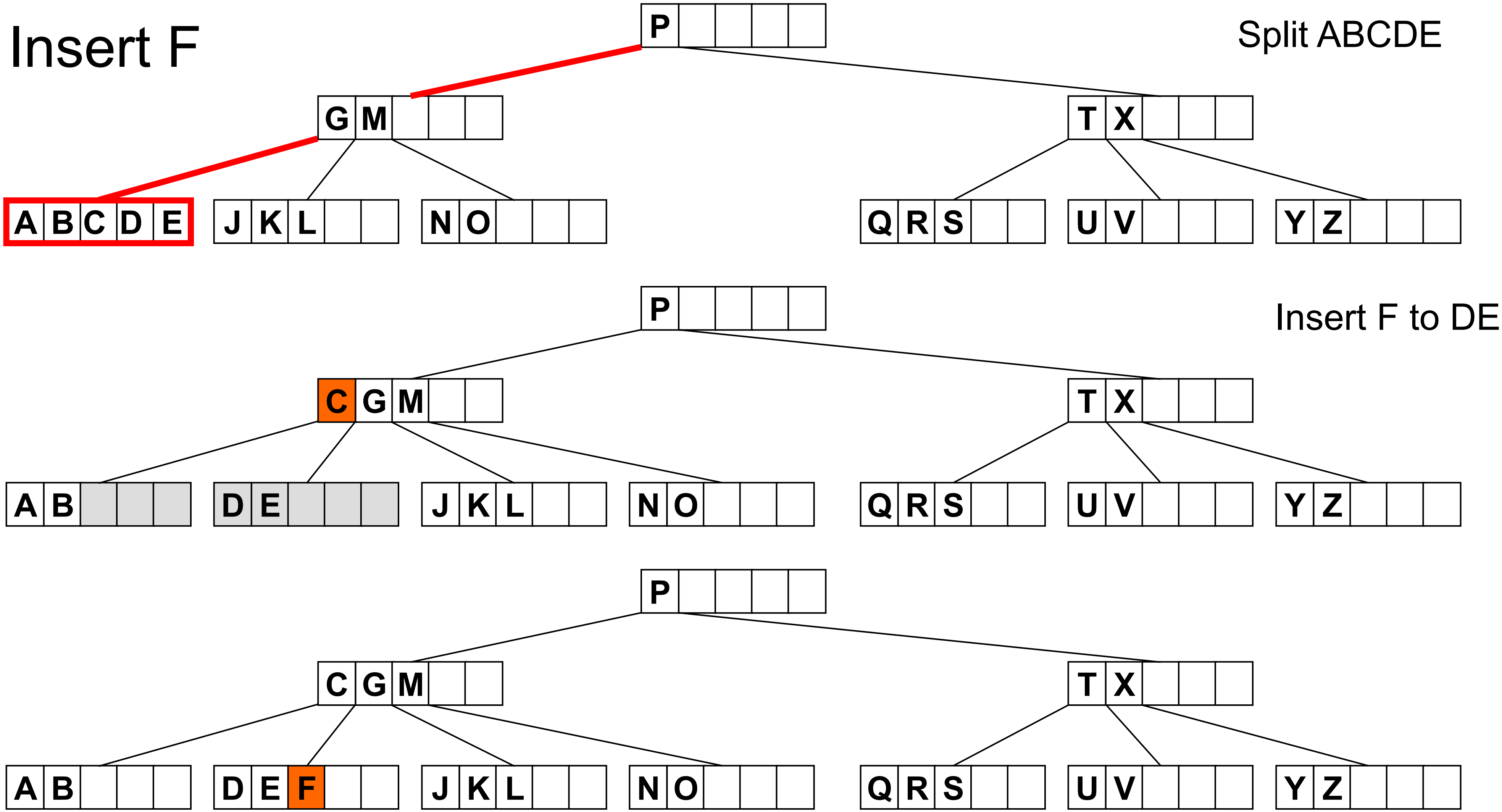
Splitting a passed **full root** and insert to a **not full** node

Insert L



B-tree insert - 2.Singlephase strategy

Insert F



B-tree insert - 2.Singlephase strategy

Insert (x, T) - pseudocode

Top down phase only

x …key, *T*… tree

While searching the leaf *x*

  if (node full)

    find median (in keys in the full node only)

    split node into two

    insert median to parent (there is space)

Insert *x* and stop

B-tree delete

Delete (x, btree) - principles

Multipass strategy only

- Search for value to delete
- Entry is in leaf
  is simple to delete. Do it. Corrections of number of elements later...
- Entry is in Inner node
  - It serves as separator for two subtrees
  - swap it with predecessor(x) or successor(x)
  - and delete in leaf

Leaf in detail

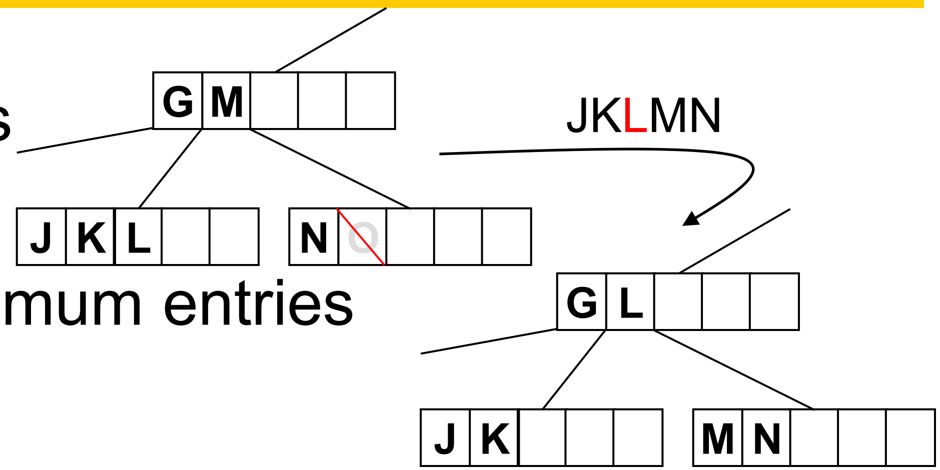
if leaf had more than minimum number of entries
    delete x from the leaf and STOP
else
    redistribute the values to correct and delete x in leaf
    (may move the problem up to the parent,
    problem stops by root, as it has no minimum number of entries)

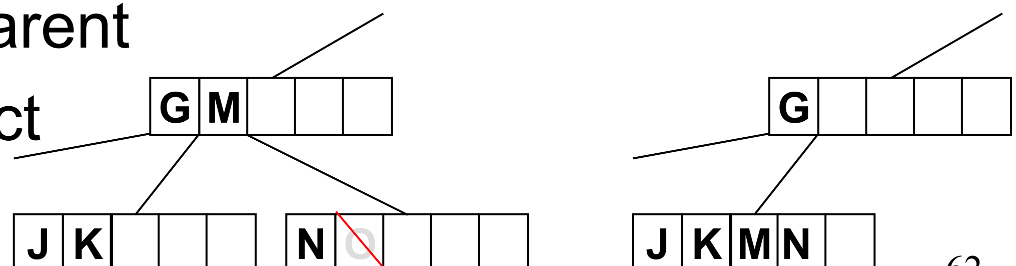
B-tree delete

Node has less than minimum entries

- Look to siblings left and right
- If one of them has more than minimum entries
  - Take some values from it
  - Find new median in the sequence:

    (sibling values – separator- node values)
  - Make new median a separator (store in parent)
- Both siblings are on minimum
  - Collapse node – separator – sibbling to one node
  - Remove separator from parent
  - Go up to parent and correct





B-tree delete

Delete (x, btree) - pseudocode

Multipass strategy only

```
if(x to be removed is not in a leaf)
    swap it with successor(x)
currentNode = leaf
while(currentNode underflow)
    try to redistribute entries from an immediate
        sibling into currentNode via its parent
    if(impossible) then merge currentNode with a
        sibling and one entry from the parent
    currentNode = parrent of CurrentNode
```

Maximum height of B-tree

$h \leq \log_{\lceil m/2 \rceil}((n+1)/2)$

half node used for k,
half of children

Gives the upper bound to number of disk accesses

See [Maire] or [Cormen] for details

References

[Cormen] Cormen, Leiserson, Rivest: Introduction to Algorithms, Chapter 14 and 19, McGraw Hill, 1990

Red Black Tree

[Whitney]: CS660 Combinatorial Algorithms, San Diego State University, 1996], RedBlack, B-trees http://www.eli.sdsu.edu/courses/fall96/cs660/notes/redBlack/redBlack.html#RTFToC5

[RB tree] John Franco - java applet

http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html

[RB tree] Robert Lafore. Applets accompanying the book "Data Structures and Algorithms in Java,“ Second Edition. Robert Lafore, 2002
(applet, v němž si lze vyzkoušet vkládání a mazání u Red-Black Tree)
http://cs.brynmawr.edu/cs206/WorkshopApplets/Chap09/RBTree/RBTree.html

B-tree

[Maire] Frederic Maire: An Introduction to Btrees, Queensland University of Technology,1998]
http://sky.fit.qut.edu.au/~maire/baobab/lecture/

References

[Wiki] B-tree. *Wikipedia, The Free Encyclopedia*. (2006, November 24). Retrieved December 12, 2006, from http://en.wikipedia.org/w/index.php?title=B-tree&oldid=89805120

[Jones] Jeremy Jones: B-Tree animation - java applet https://www.cs.tcd.ie/Jeremy.Jones/vivio/trees/B-tree.htm

Splay tree

[Wiki] Splay tree. *Wikipedia, The Free Encyclopedia*. (2007, Oct 29) Retrieved November 27, 2007 from <http://en.wikipedia.org/w/index.php?title=Splay_tree&oldid=167855497>.

Tree comparison

[Pfaff 2004] Ben Pfaff. Performance Analysis of BSTs in System Software, extended abstract of this paper appeared in the proceedings of SIGMETRICS/Performance 2004. http://www.stanford.edu/~blp/papers/libavl.pdf