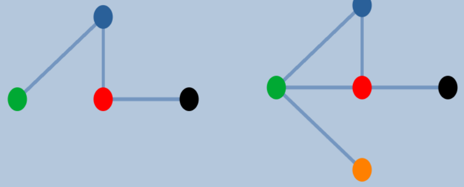
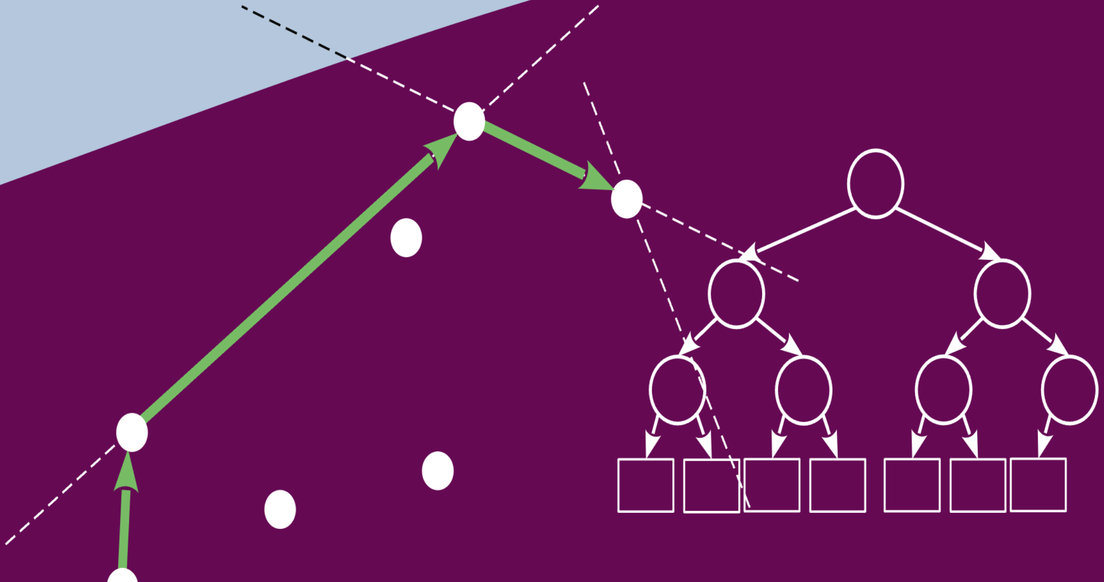
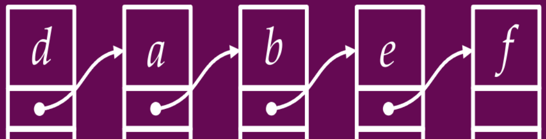
Advanced algorithms and data structures

Week 2: B-trees and Red-Black trees (RB)

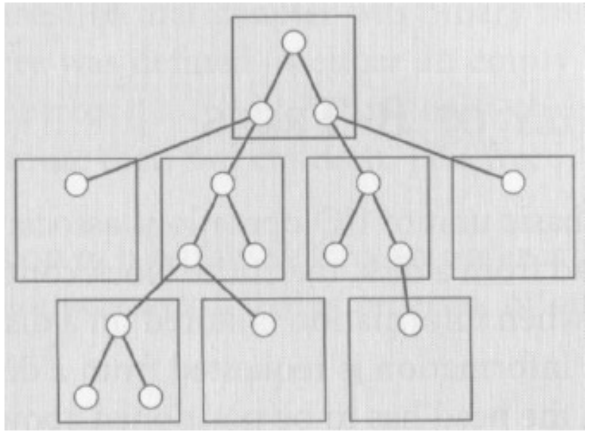






Motivation

- External memory
  - Sequential reading by blocks
  - Neighboring nodes in the tree can be scattered in distant blocks



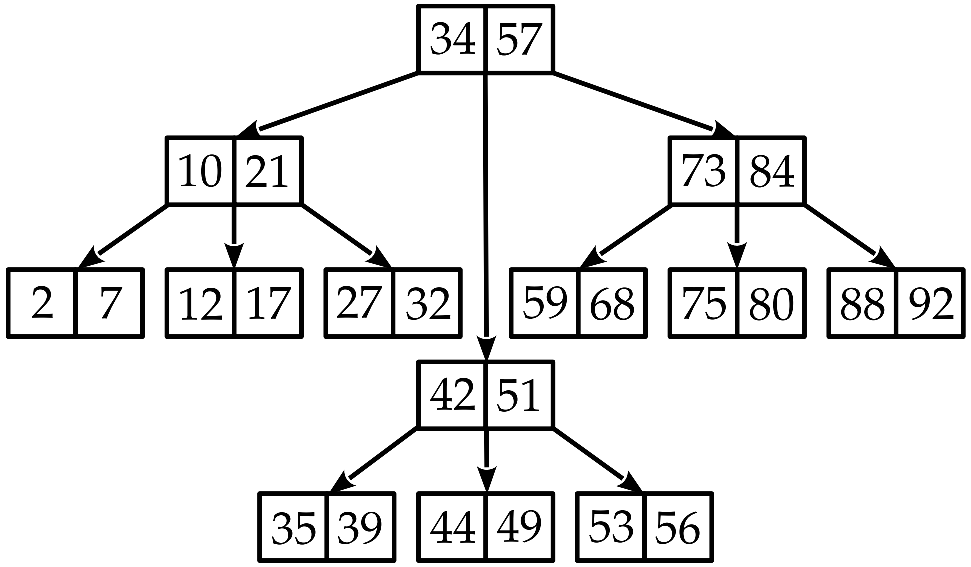
- B-trees alleviate the effects of the sequential block read limitation
  - The size of the node adjusts to the size of the block

Characteristics

- Complete balance
- Sort data by key value
- Storing a certain number of elements in one node

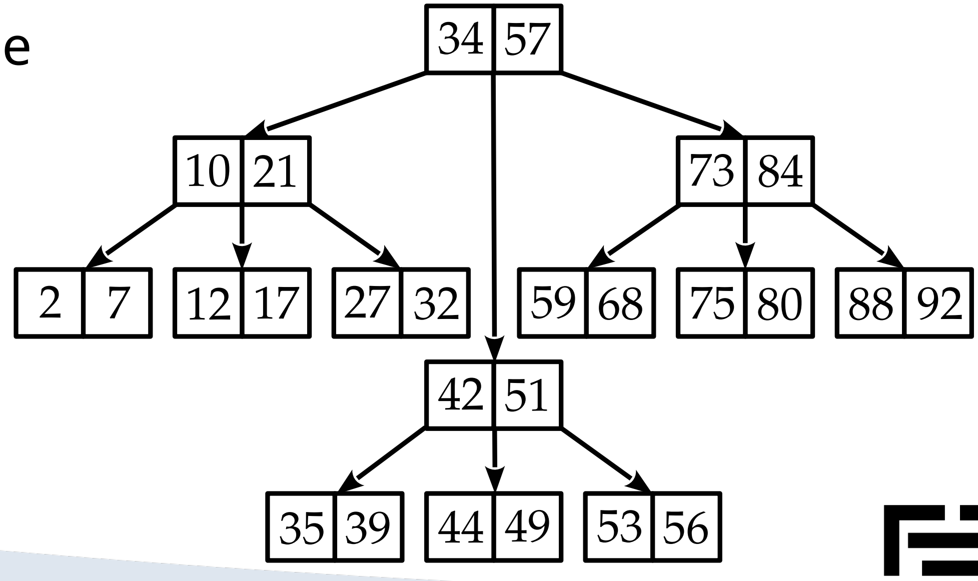
M trees

- M trees (*multiway tree*): trees in which nodes can have an arbitrary number of children
- M tree*m*-of order: M tree in which nodes can have at most *m* children.



M trees

- Properties of an M-tree*m*- of that order:
  1. Each node has a maximum*m*children and*m–1*data (keys)
  2. The keys in the nodes are sorted
  3. Keys in the first*and*children of a node are smaller than*and*of the key of the observed node
  4. Keys in the last*me*children of a node are greater than*and*of the key of the observed node



B-tree

- B tree*m*-of order is an M-tree with additional properties:
  1. The root has at least two children, unless it is also a leaf (the only node in the tree).

Utilization ≥50% - internal

  2. Each node, apart from the roots and leaves, contains**at least** $k-1$keys and $k$pointers to subtrees (has$k$children), whereby

$$\lceil m/2 \rceil \leq k \leq m$$

Utilization ≥50% - leaves

  3. All sheets contain**at least** $k-1$keys, where is it

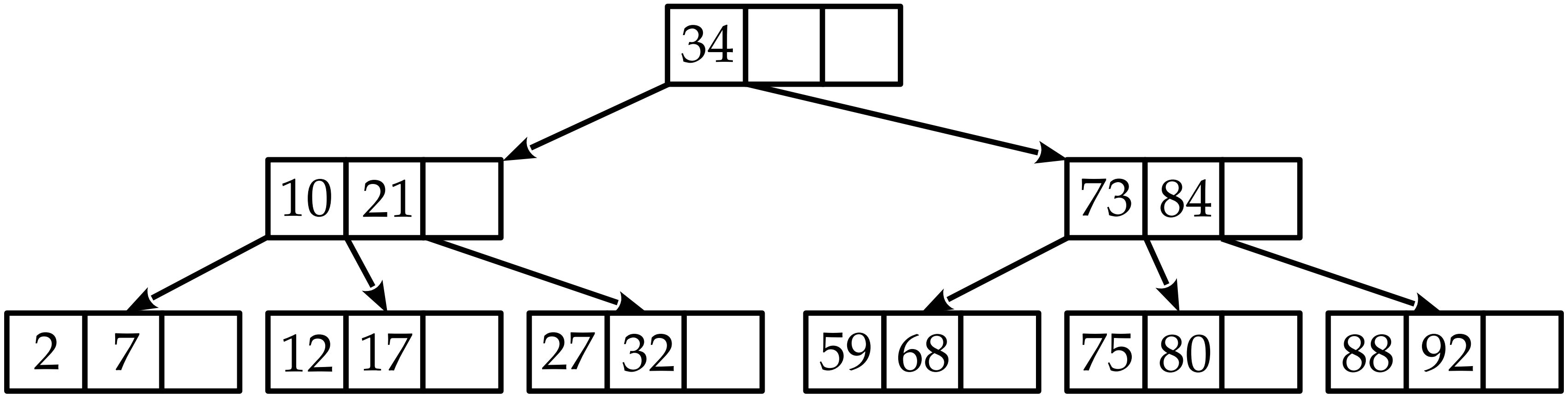
$$\lceil m/2 \rceil \leq k \leq m$$

  4. All sheets are on the same level

Perfect balance

B-tree

- Peculiarities
  - Occupancy at least 50%
  - Perfectly balanced



B-tree

- Implementation #1
  - A structure (class) with a field of *m-1* keys and field of *m* pointers – in Python, there can be a single field where pointers and keys are exchanged
  - It is possible to add data for easier maintenance (e.g. the number of entered data in the node)

- Implementation #2
  - Each node is a doubly linked list
  - Each key has pointers to children – only the last key uses both pointers

B-tree search algorithm

1. Enter the node and review the keys in turn as long as the current one is less than the requested one, and there are still unverified ones
   - The first node entered is the root

2. If the 1st step ended due to encountering a key larger than the required one or due to reaching the end of the node, go down to a lower level and repeat the first step
   - If there is no lower level, there is no required key

B-tree search - implementation

```
function BTreeSearch($n, v_s$)
    $n_v \leftarrow$ starting value of the node $n$
    while $value(n_v) < v_s$ and $next(n_v)$ is not $nil$ do
        $n_v \leftarrow next(n_v)$
    if $value(n_v) = v_s$ then
        return $n$
    else if $next(n_v)$ is $nil$ and $value(n_v) < v_s$ then
        if $rightChild(n_v)$ is not $nil$ then
            return BTreeSearch($rightChild(n_v), v_s$)
        else
            return no searched key
    else
        if $leftChild(n_v)$ is not $nil$ then
            return BTreeSearch($leftChild(n_v), v_s$)
        else
            return no searched key
```

- Remark
  - value(n_c) – value of key n_c– eg an integer
  - next(n_c) – the next key after $n_c$in the node key list – this depends on the node implementation
  - leftChild(n_c) and rightChild(n_c) – left and right child of key n_c

Adding data to the B-tree

- It is simpler to build a B-tree**from the bottom up**

- Algorithm:
  1. find the sheet in which the new data should be placed
  2. if there is space, enter new data
  3. if that sheet is full, "split" (*Split*) ga (create a new sheet, evenly distribute the elements between the two nodes, and write the central element in the parent)
  4. if the parent is also full, "split" the parent as well (repeat the procedure from step 3)
  5. if the root is also full, "split" it and make a new root

Adding data to the B-tree

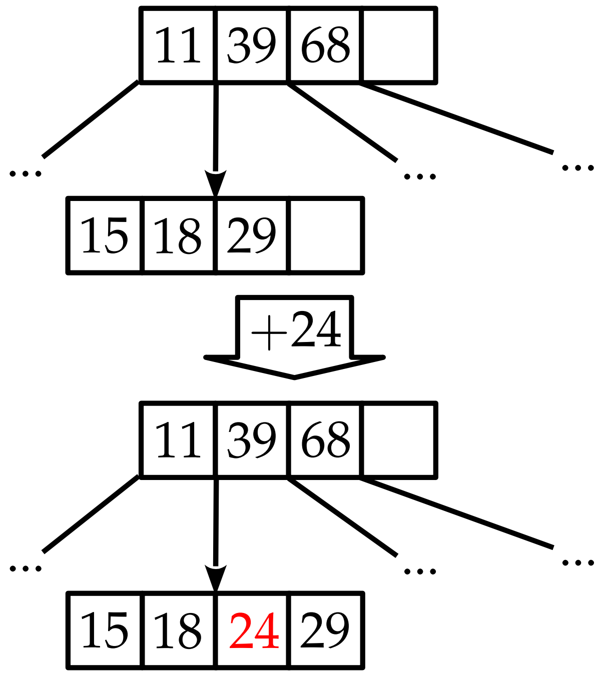
- When inserting new data, 3 situations are possible:

1. the sheet where the new element should go is not full
   - insert a new element in that sheet in the appropriate place, moving the previous content if necessary
2. the leaf where the new element should go is full, but the root of the tree is not
   - the leaf is split (a new node is created) and all elements are distributed evenly, with the central element being written to the parent
3. the leaf where the new element should go is full, and so is the root of the tree
   - when the root is divided, two B-trees are created that need to be united

Adding data to the B-tree

- The union in the third case is achieved by creating another node that will be the new root and writing the central element in it
    - It's the only case that ends up raising the tree
    - **The B-tree is always perfectly balanced**

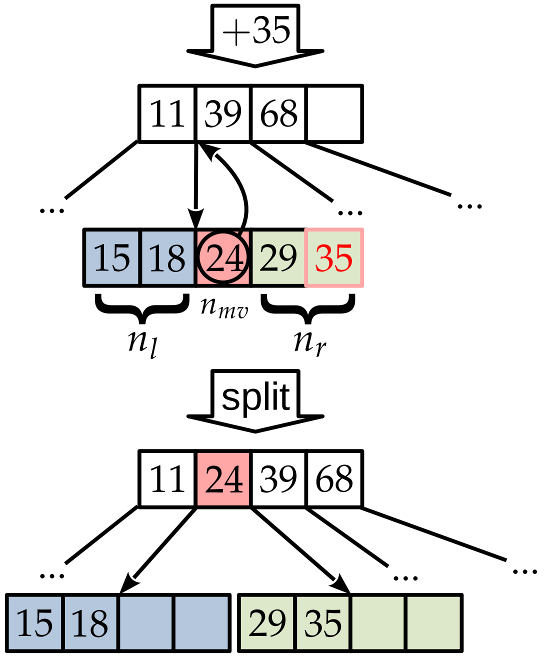
An example of adding data to a B-tree



- **Example 1:** we add 24 to the sheet in which there are less than − 1keys. There is no need to restructure the B-tree.

Max. degree = 5

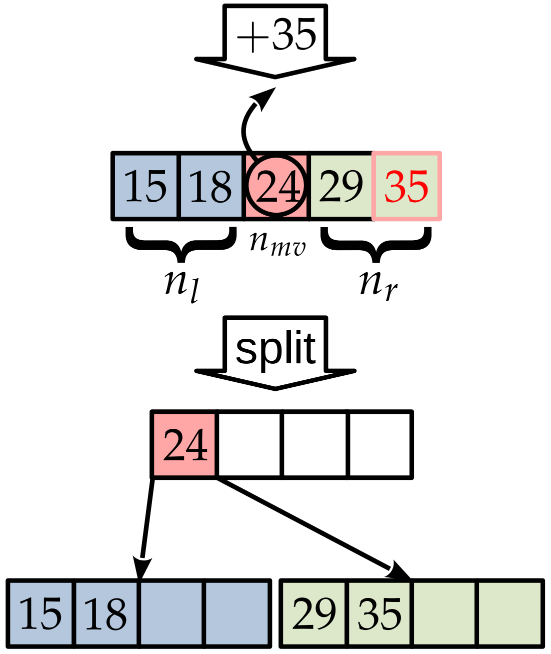
An example of adding data to a B-tree



- **Example 2**: we add 35 to the sheet where it is correct − 1keys and which has a parent node.
  - There is an overflow in the sheet.
  - The leaf is divided into a central key and two parts.
  - Insert the central key into the parent, and split the leaf into two nodes.

Max. degree = 5

An example of adding data to a B-tree



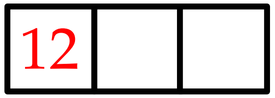
- **Example 3**: we add 35 to the node where there is exactly -1keys and which**there is none**the parent node (obviously the root node).
  - There is an overflow in the node.
  - The leaf is divided into a central key and two parts.
  - We use the central key to create a new root node, and the leaf we separate into two nodes.

Max. degree = 5

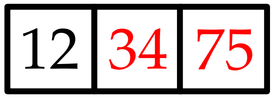
An example of adding data to a B-tree

- B-tree of row 4 – 4 hands, 3 keys
- We add the keys in order:

  12,75,34,62,19,25,66,30,33,71,47,21,15, 23,27

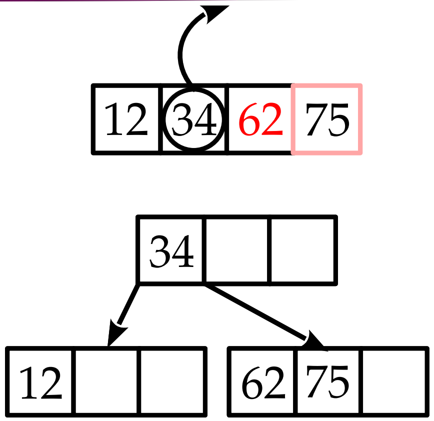


- **Step 1**: We form the root node with the first key**12**

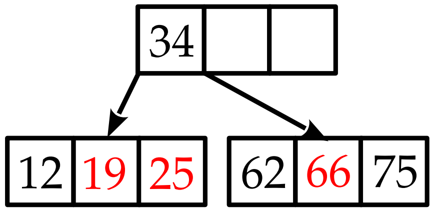


- **Step 2**: We add keys to the node until we can - we add**75**and**34**

An example of adding data to a B-tree



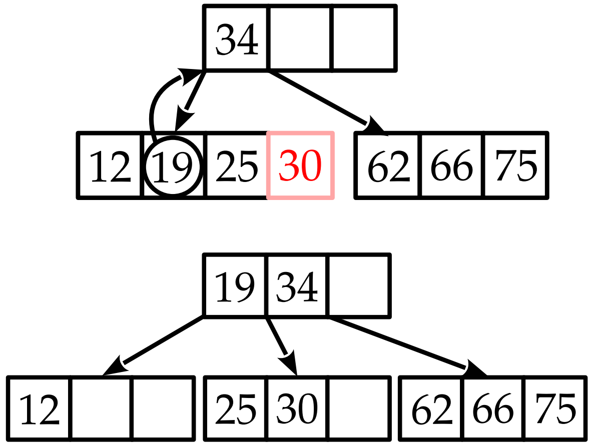
- **Step 3**: By adding a key**62**, there is an overflow in the root node, which we separate with the creation of a new one root node.



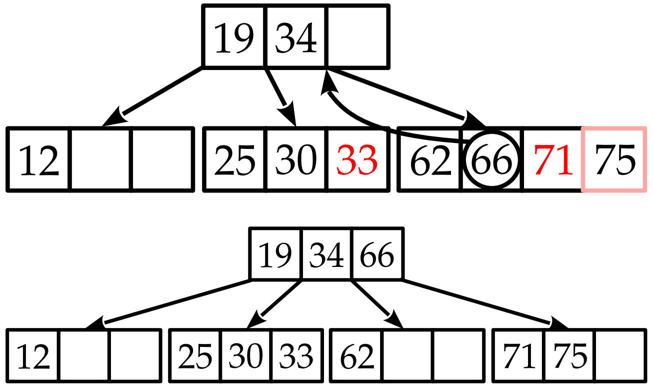
- **Step 4**: We are adding keys**19**,**25**and**66** directly into the leaves.

12,75,34,62,19,25,66,30,33,71,47,21,15,23,27

An example of adding data to a B-tree



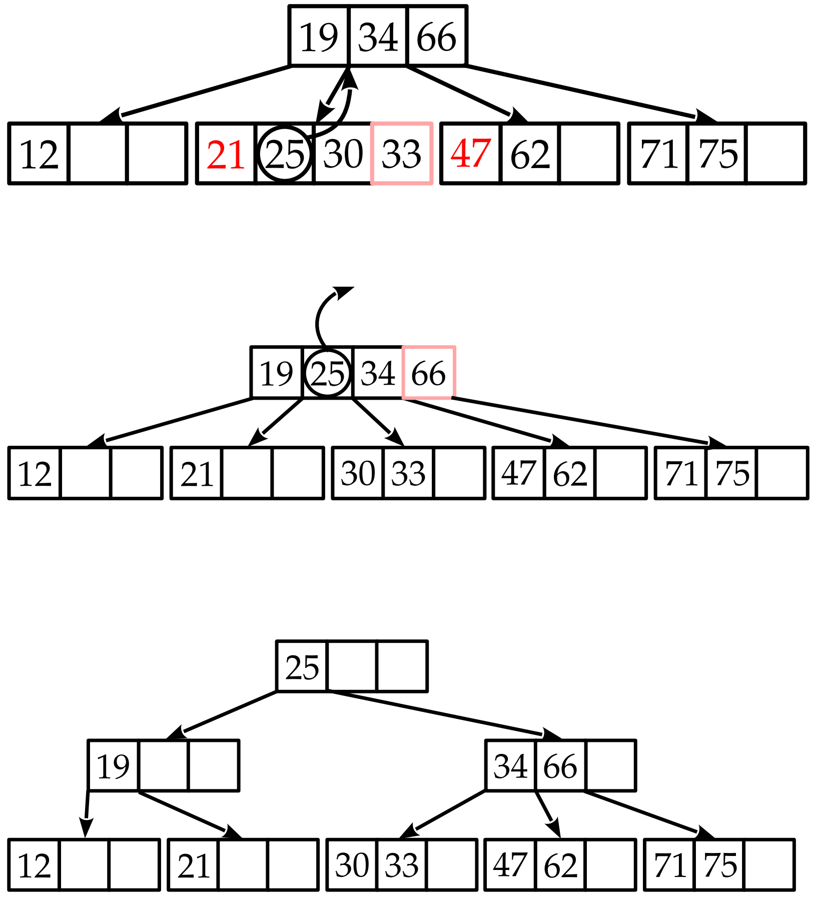
- **Step 5**: By adding a key**30**, there is an overflow in the left leaf, which we separate by inserting the middle key into the root node.



- **Step 6**: We add the key**33**, and then**71**. By adding**71**we cause an overflow in the right leaf, which we separate by inserting the middle key into the root node.

12,75,34,62,19,25,66,30,33,71,47,21,15,23,27

An example of adding data to a B-tree

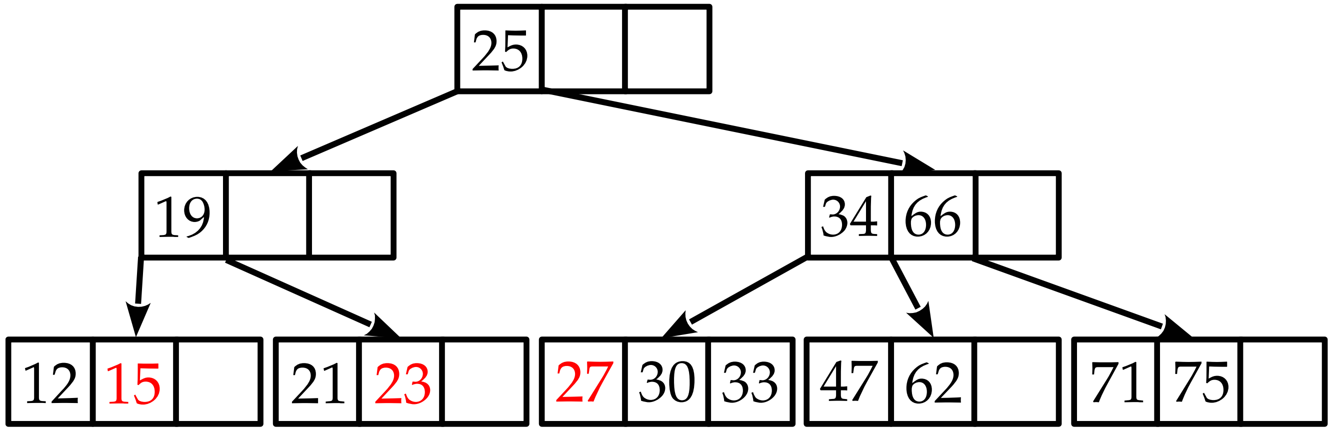


- **Step 7**: We add the key**47**, and then the key**21**.
  - By adding**21**we cause an overflow in the leaf, which we separate by inserting the middle key into the root node.
  - By inserting 25 into the root node, we cause an overflow of the root node, which we separate while creating a new root node.

12,75,34,62,19,25,66,30,33,71,47,21,15,23,27

An example of adding data to a B-tree

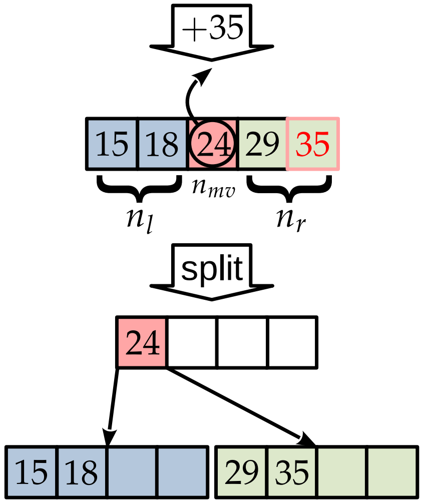
- **Step 8**: We are adding keys**15**,**23**and**27**directly into Bstable sheets without restructuring.

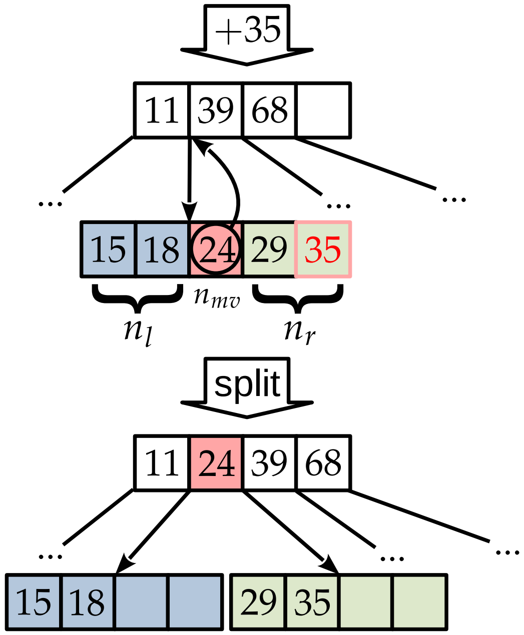


12,75,34,62,19,25,66,30,33,71,47,21,15,23,27

Implementation of B-tree addition

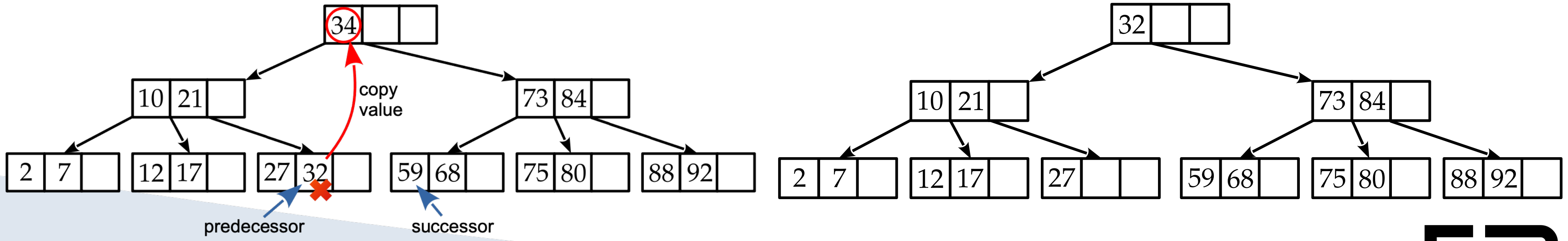
```
procedure BTreeSplit($btree, n$)
    if $|n| == m$ then
        $n_l \leftarrow$ new node having first $\lceil m/2 \rceil - 1$ values in the node $n$
        $n_{mv} \leftarrow$ the next value in the node $n$
        $n_r \leftarrow$ new node having the rest of values from the node $n$
        $n_{last} \leftarrow$ the last value in $n_l$
        $rightChild(n_{last}) \leftarrow leftChild(n_m)$
        if $parent(n)$ is $nil$ then
            $n_{root} \leftarrow$ new node
            $n_{mv}' \leftarrow$ insert $value(n_{mv})$ into the node $n_{root}$
            $root$ of $btree \leftarrow n_{root}$
            $leftChild(n_{mv}') \leftarrow n_l$
            $rightChild(n_{mv}') \leftarrow n_r$
        else
            $n_{mv}' \leftarrow$ insert $value(n_{mv})$ into the parent node of $n$
            $leftChild(n_{mv}') \leftarrow n_l$
            $rightChild(n_{mv}') \leftarrow n_r$
            BTreeSplit($btree$,parent of $n$)

procedure BTreeInsert($btree, v_n$)
    $(n, n_v) \leftarrow$BTreeSearch($root$ of $btree, v_n$)
    insert $v_n$ into the node $n$
    BTreeSplit($btree, n$)
```





Deleting data in the B-tree

- 2 cases:
  1. deleting an element in the tree leaf
  2. deleting an element in a node
     - it boils down to deleting an element from the list
       - in the place of the element to be deleted, its immediate predecessor (which can only be in the list) is written, then the overwritten element is deleted from the list using the standard procedure for deleting a list

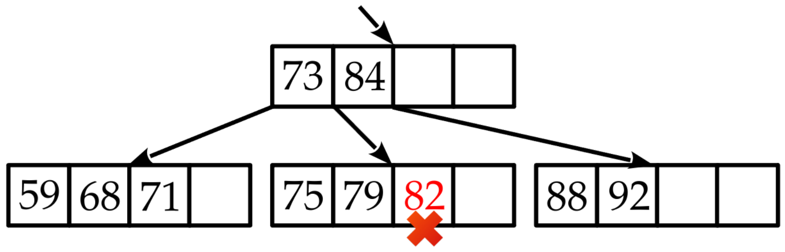


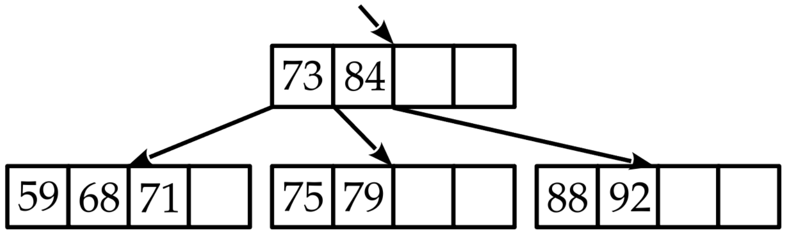
Deleting data in the B-tree

1. leaf even after deleting the element has ≥⌈*m/2*⌉-*1*keys;**END**
2. number of remaining elements (keys) <⌈*m/2*⌉-*1*
   1. if there is a neighbor with > on the left or right⌈*m/2*⌉-*1*keys
      - distribute leaf elements, neighbor elements, and the central element from the parent evenly into the leaf and neighbor, and write the central element of the united set (union) of elements as a new central element in the parent;**END**
   2. the leaf and neighbor are merged (all elements of the leaf and neighbor + the central element from the parent are written into the leaf, and the neighbor is deleted);**CONTINUE with parent**
   3. by procedure 2.2 we get to the root:
      - if root has more than 1 element: merge current node and neighbor as in 2.2;**END**
      - otherwise: all leaf, neighbor and root elements are written into 1 node which becomes the new root, and 2 nodes are deleted from the tree;**END**

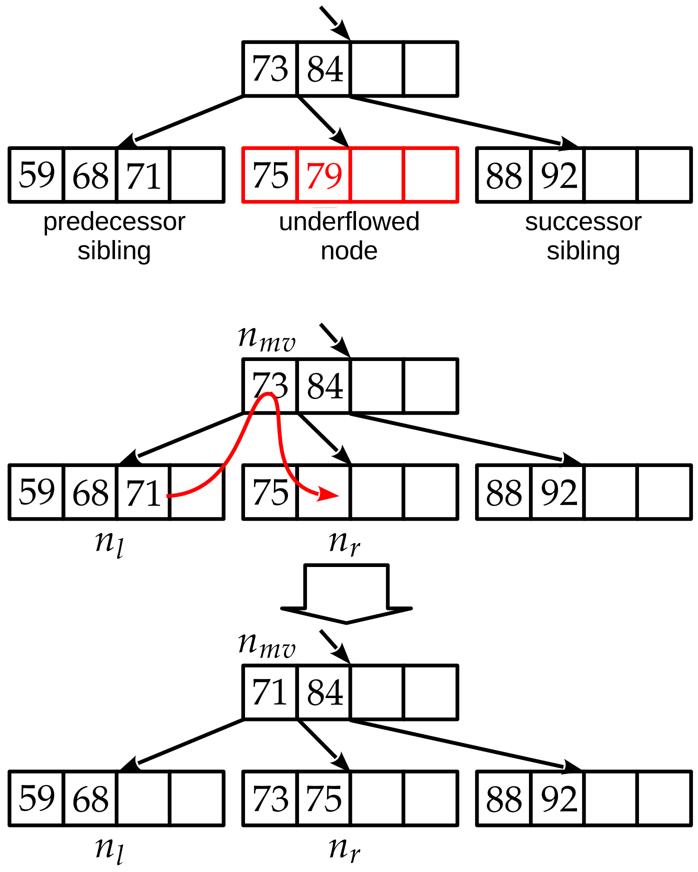
An example of deleting data from the B-tree

- **Example 1**: we delete the key**82**from the sheet. After deletion, the sheet is still filled≥50% because there is≥⌈m/2⌉-1keys. There is no need to restructure the B-tree.





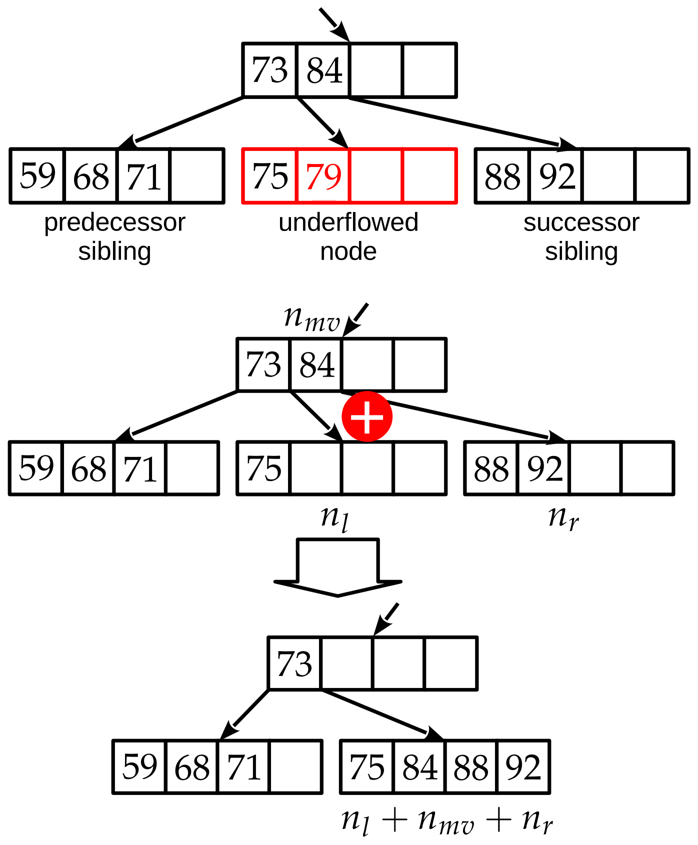
An example of deleting data from the B-tree



- **Example 2**: we delete the key**79**from the sheet.

  After deletion, the sheet is no longer filled ≥50% because it has <⌈m/2⌉-1 keys.
- If we look at the left neighbor (twin), we see that it is filled >⌈m/2⌉-1
  - We do the restructuring by transferring the keys from the left neighbor to the node that is in the subflow
  - In doing so, we pay attention to the shared key in the parent node

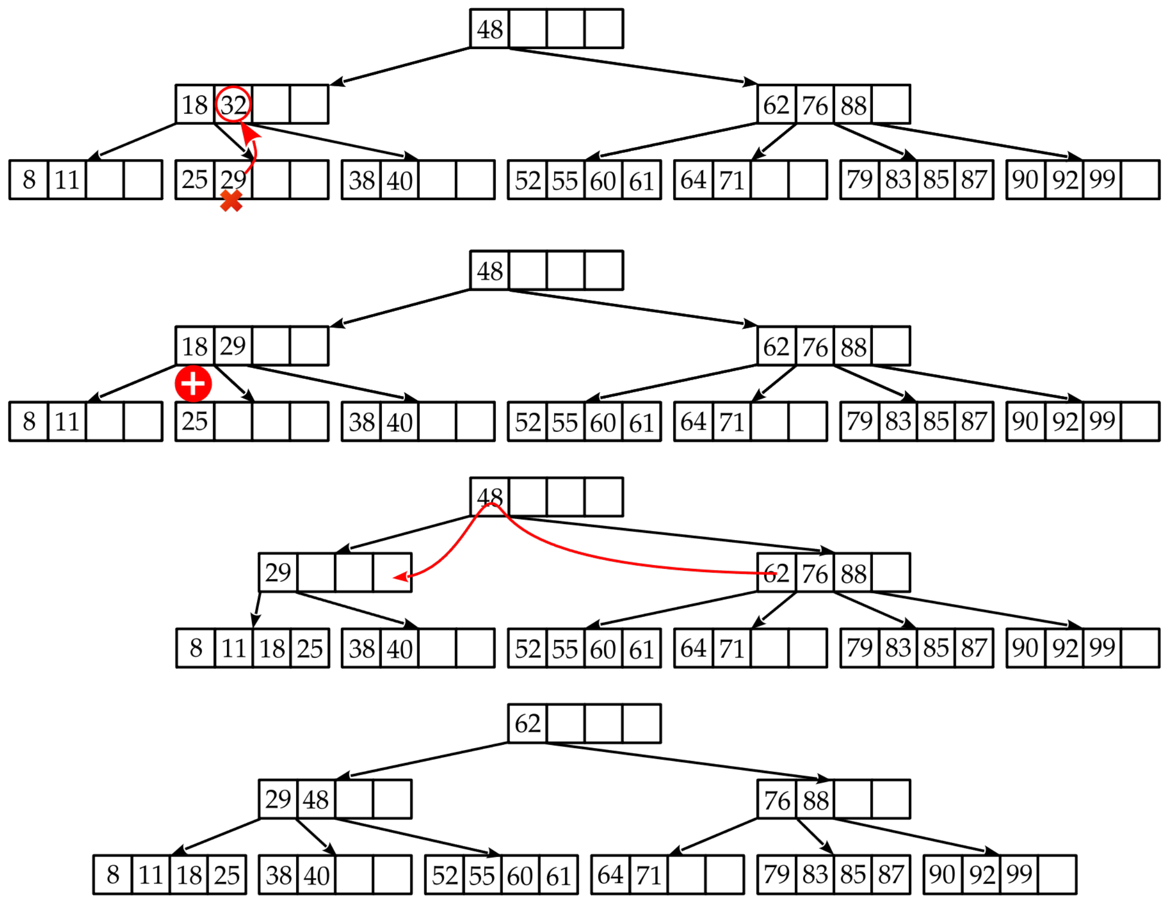
An example of deleting data from the B-tree



- **Example 3**: we delete the key**79**from the sheet.

  After deletion, the sheet is no longer filled ≥50% because it has <⌈m/2⌉-1 keys.
- If we look at the right neighbor (twin), we see that it is filled with =⌈m/2⌉-1
  - We connect the right neighbor and the node that is in the underflow (*underflow*)
- Notice that now the parent node is in the subflow, which we have to solve recursively

An example of deleting data from the B-tree

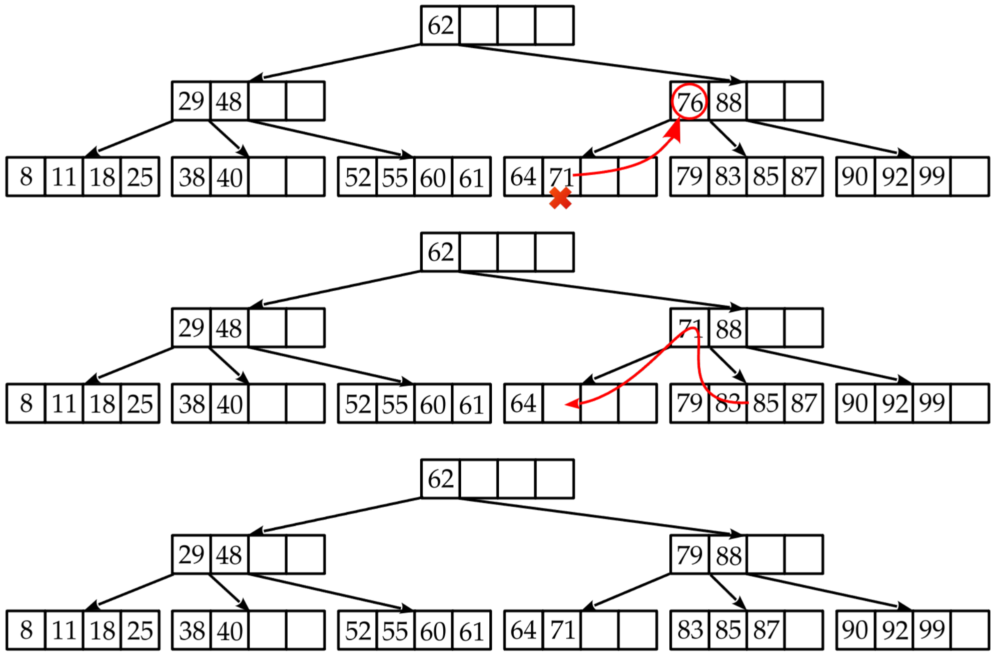


- We delete the keys from the initial B-tree

  32, 76, 48, 25 and 11

- **Step 1**: we delete the key **32** by copying process
  - We connect the leaf in the base with the left neighbor
  - This causes the underflow of the internal node
  - We restructure the right neighbor and the internal node in the subflow

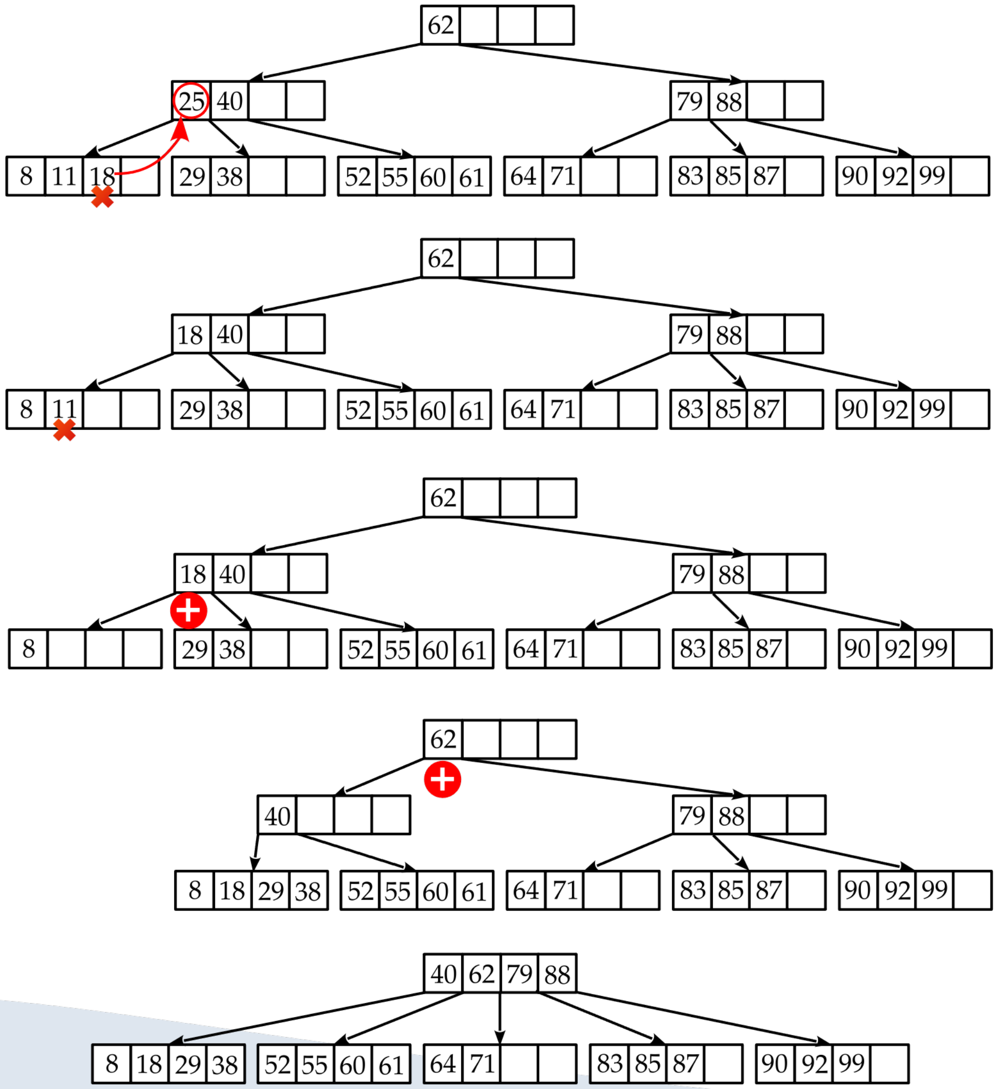
An example of deleting data from the B-tree



- **Step 2**: we delete the key**76**by copying process. The sheet where we deleted the replacement key is in the footer.
  - The right neighbor to the node in the underflow has 100% occupancy
  - We restructure the right neighbor and the node in the subflow

32, 76, 48, 25 and 11

An example of deleting data from the B-tree



- **Step 3**: we delete the key**25**by the copying process, and then we delete the key**11**. The sheet where we deleted 18 and 11 is now in the underfill.
    - The right neighbor to the node in the underflow has exactly 50% occupancy, so we connect the node in the underflow to the right neighbor
    - Now the internal node is in the underflow, and its right neighbor has exactly 50% occupancy, so we connect the node in the underflow with the right neighbor
    - With the previous merge, the old root node disappears, and the newly merged node becomes the root node. Tree depth is reduced by 1.

Implementation of element deletion in B-tree

```
procedure BTreeRemoval($btree, val$)
    $(n_{rem}, n_v) \leftarrow$ BTreeSearch(root of $btree, val$)
    if $n_v$ is not $nil$ then
        remove value $val$ from the node $n_{rem}$
        BTreeRemovalConsolidation($btree, n_{rem}$)
```

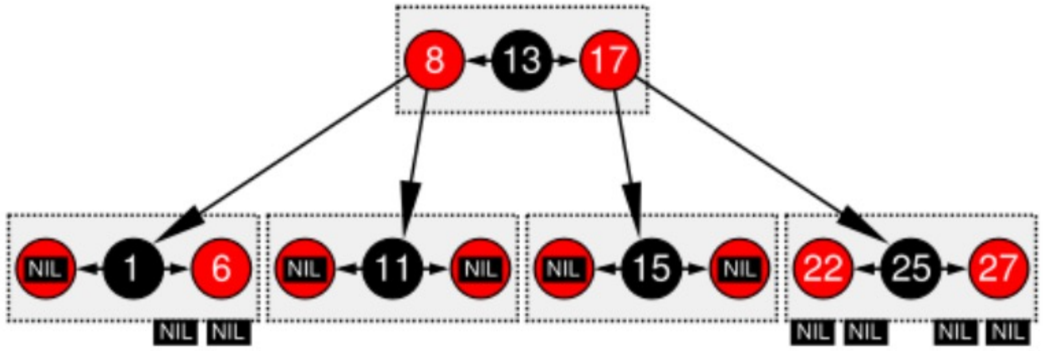
```
procedure BTreeRemovalConsolidation($btree, n$)
    if $|n| < \lceil$degree of $btree/2\rceil - 1$ then
        $ps \leftarrow$ the predecessor sibling
        $n_{root} \leftarrow nil$
        if $ps$ is not $nil$ then
            $n_{mv} \leftarrow$ the shared parent value between $ps$ and $n$
            if $|ps| > \lceil$degree of $btree/2\rceil - 1$ then
                BTreeRedistribute($btree, ps, n_{mv}, n$)
            else
                $n_{root} \leftarrow$ BTreeMerge($btree, ps, n_{mv}, n$)
        else
            $ss \leftarrow$ the successor sibling
            $n_{mv} \leftarrow$ the shared parent value between $ss$ and $n$
            if $|ss| > \lceil$degree of $btree/2\rceil - 1$ then
                BTreeRedistribute($btree, n, n_{mv}, ss$)
            else
                $n_{root} \leftarrow$ BTreeMerge($btree, n, n_{mv}, ss$)
        if $n_{root}$ is $nil$ and parent of $n$ exists then
            BTreeRemovalConsolidation($btree$, parent of $n$)
```

Red and black trees

Red-black tree (*Red-black tree*)

- A binary tree that**conceptually**arises from a B-tree of order 4 if its node elements are considered colored according to strict rules

- Comparison with B-tree:
  - lower memory consumption
  - maintains balance
  - the complexity is the same



Definition rules

1.each node is**red**or**black**

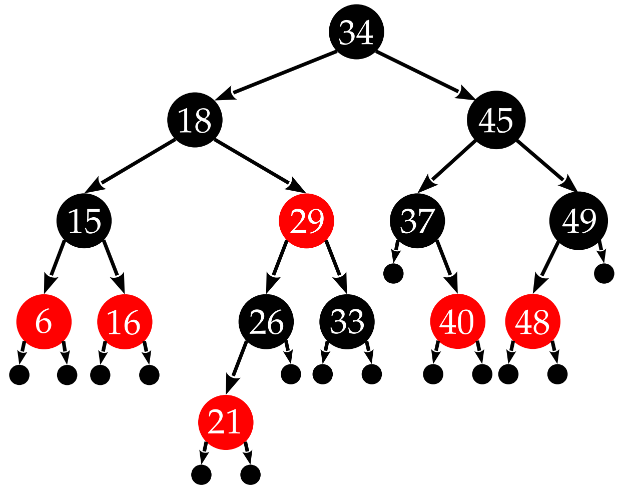
2.is the root**black**(optional but common)

3.each sheet* is**black**

4.both descendantsrednodes are**black**

5.every path from a node to (any) leaf that is its descendant passes through the same one

by the number of black nodes



- * Leaves in a red-black (RB) tree do not contain information, so they do not have to exist, but parents can have NULL pointers or all point to the same special node, the sentinel

Red and black tree height

- We differentiatered and **black**tree height:
  - rh(x), bh(x)
    - the number of nodes of a certain color on the path from node x to the leaf that is its descendant (x is not counted).

- Key property for RB tree balance:
  - the longest path from the root to a leaf is at most twice as long as the shortest path from the root to a (other) leaf
  - i.e. the longest path is at most twice as long as the shortest.

Theorem

- The height of the RB-tree s of internal nodes is $h \leq 2$!(+ 1) **Evidence:**

  Binary height tree *h* has the most = 2"- 1of nodes Due to the 4th rule, at least half the height is black height so it is *h* ≥ *h*/2.Since n is greater than or equal to the number of black nodes on the path from the root to the lowest leaf, it follows:

  $$\geq 2^{"\#} - 1 \geq 2^{"} - 1,\overset{!}{\text{and}} \text{ from that directly}$$

  $$h \leq 2 \;\; !(\;\; + 1)$$

- Searching a binary tree is of complexity O(h), so the complexity of searching an (RB-tree) !

Adding a node to the RB-tree

- For easier analysis, terms are introduced
  - node-uncle (uncle) which is denoted by**IN**, and means the twin of the parent of the observed node (parent's brother/sister)
  - the grandfather node denoted by s**MR**(grandparent), meaning the parent of a parent

1. insert a new node as in any other binary search tree and assign itredcolor
2. restructure the tree (by applying rotations and coloring nodes) to satisfy the definition rules

Adding a node to the RB-tree

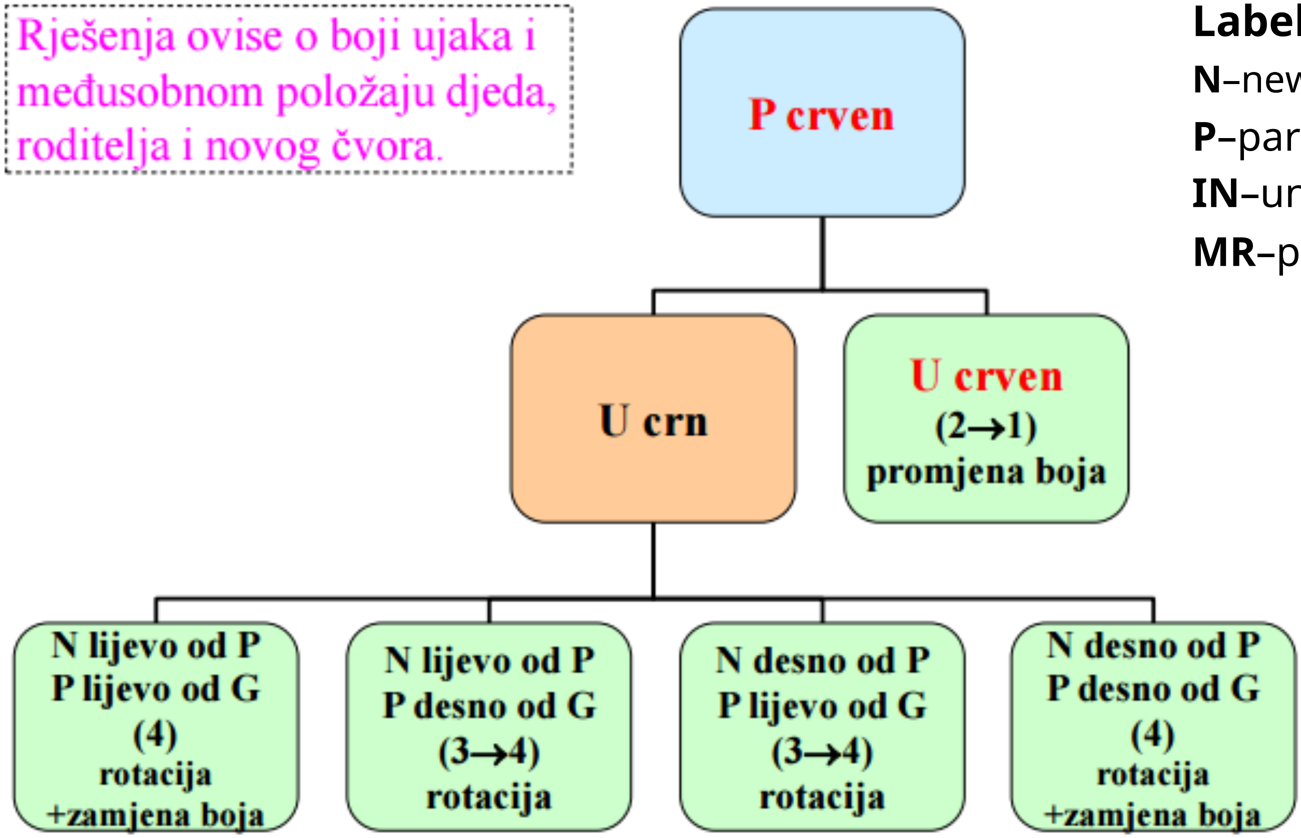
- Definition rules 1, 3 and 5 are always satisfied when adding a new node, and 2 and 4 can be compromised (not simultaneously) in the following ways:
  - rule 2 if the new node is the root
  - rule 4 if it is the parent of the new nodered
    - In both cases, restructuring is necessary
- Restructuring:

1.the new node is the root:

  - recolor it in**black**(Rule 5 remains satisfied because it is an additional black node in all paths in the tree)

Adding a node to the RB-tree

- Restructuring:
  2. the parent of the new node is red (step 1):


Rješenja ovise o boji ujaka i međusobnom položaju djeda, roditelja i novog čvora.
P crven
U crn
U crven (2→1) promjena boja
N lijevo od P P lijevo od G (4) rotacija +zamjena boja
N lijevo od P P desno od G (3→4) rotacija
N desno od P P lijevo od G (3→4) rotacija
N desno od P P desno od G (4) rotacija +zamjena boja

Labels:

N–new node

P–parent of N

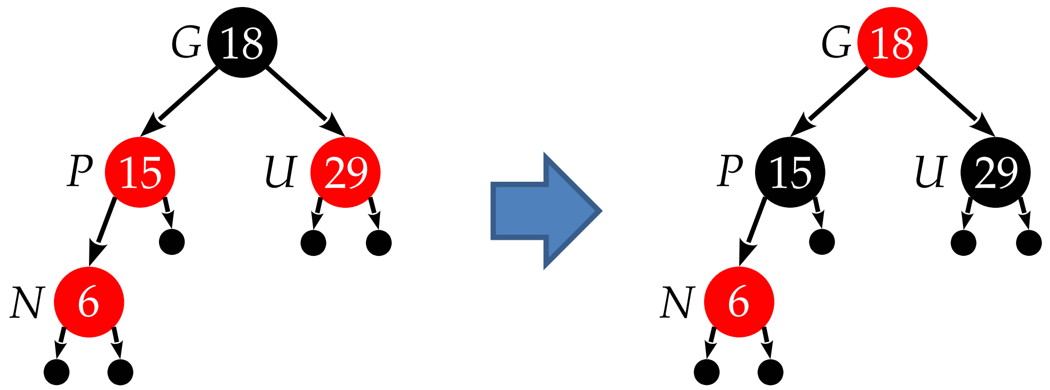
IN–uncle (twin of P)

MR–parent of P

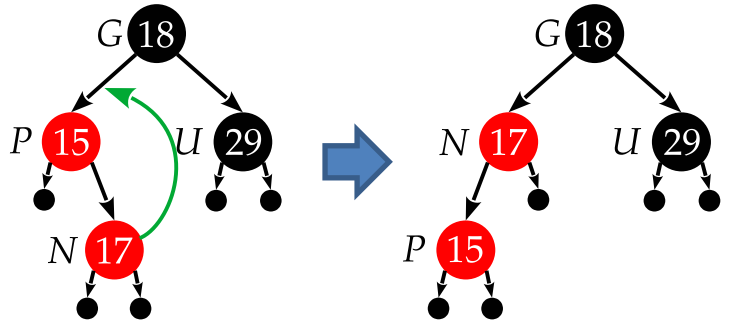
Adding a node to the RB-tree

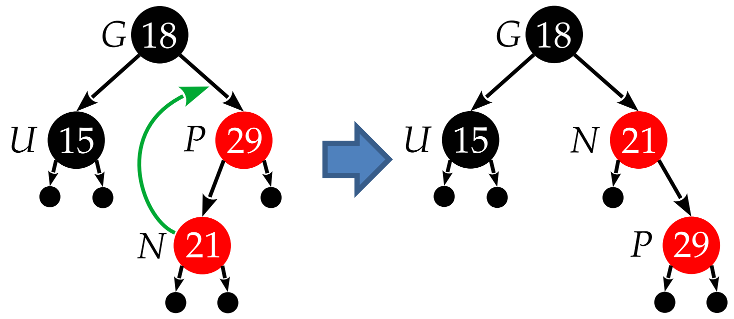
2.They are a parent and an uncleRed

- Rule 4 violated (two reds strung together; P and N)
  - repaint P and U in black (solves the 4th rule), and G inred (preservation of rule 5) - now G can violate rule 4 if it has redof the parent or the 2nd rule if it is the root
  - **Continue**with a check considering G as a new node (N)



Adding a node to the RB-tree

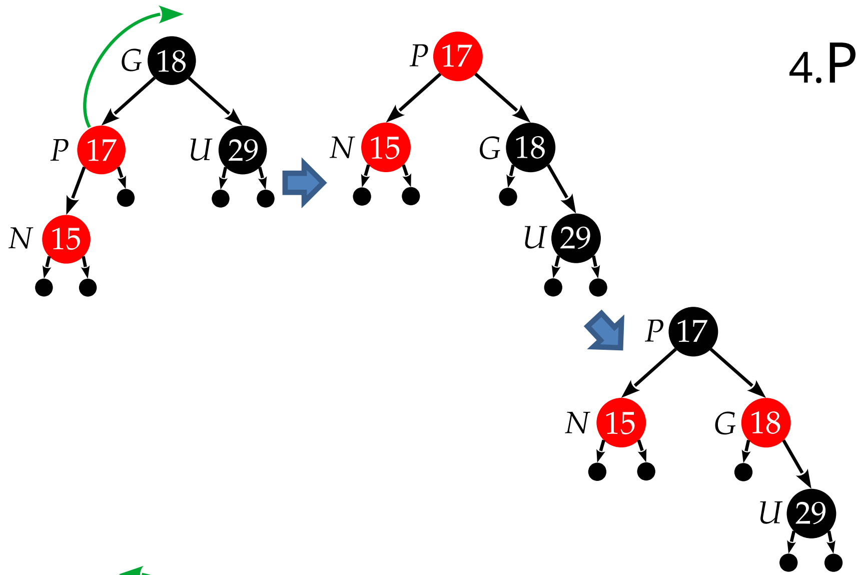


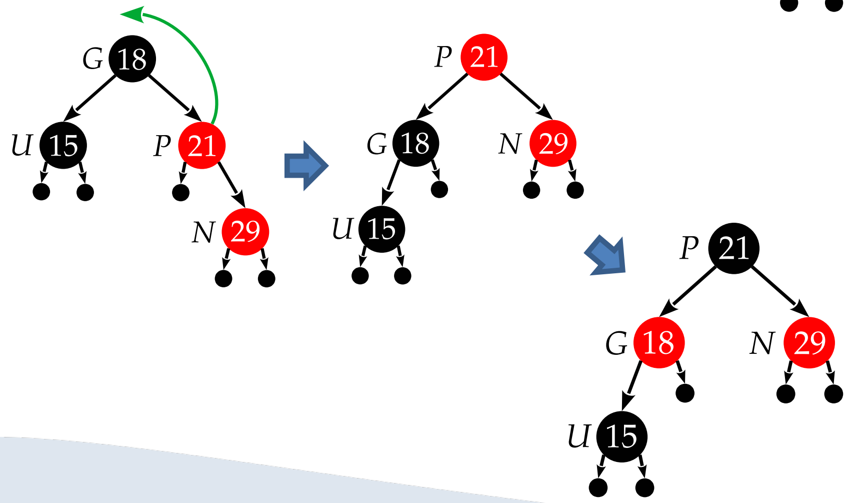


3.Parentredand Uncle Black ("broken" order of N, P and G)

- Two symmetrical cases:
  - N right child of P and P left child of G
  - N is the left child of P and P is the right child of G
- Solution:
  - the rotation of N around P, which translates the state into an “aligned order” of N, P and G which is solved in the 4th check
  - **Continue**with check (4), assigning Pu role to Na

Adding a node to the RB-tree



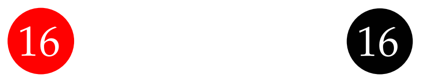


4.Parentredand uncle black ("linear" order N, P and G)

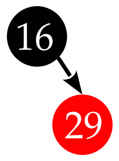
- Two symmetrical cases:
  - N is the left child of P and P is the left child of G
  - N right child of P and P right child of G
- Solution:
  - *rotation*Around G
  - *color swap*P and G (we know that G is black because otherwise P would not be could bered);**END**

Example of adding data to the RB-tree

- We add the keys in order:**16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40**

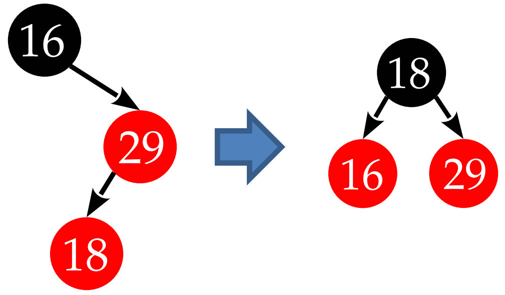


- **Step 1**: We form the root node with the first key**16**. After adding, the node is red, so let's turn it into black (rule 2).

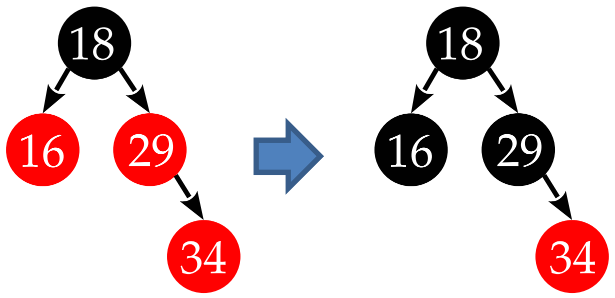


- **Step 2**: We add a key to the tree **29**. No RB-tree restructuring.

Example of adding data to the RB-tree



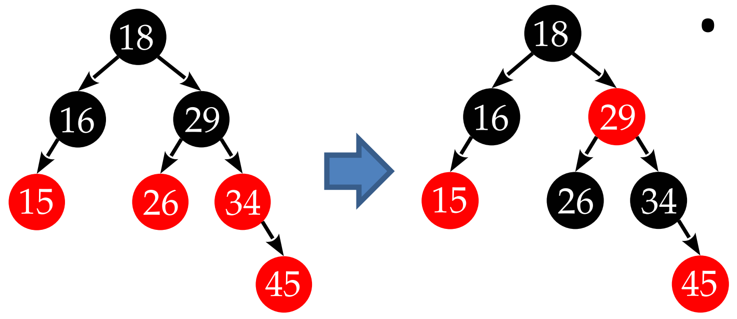
- **Step 3**: We add a key to the tree**18**.
  - **Case 3**: Right rotation**18**eye**29**
  - **Case 4**: Left rotation**18**eye**16**+ color swap.



- **Step 4**: We add a key to the tree**34**.
  - **Case 2**: Set**18**in red, a**16**and **29**in black
  - **Case 1**: Set the root**18**in black

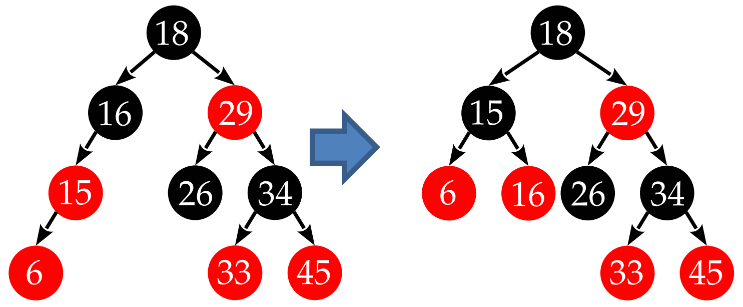
16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40

Example of adding data to the RB-tree



- **Step 5**: We add keys to the tree**26**, **15** and**45**.
  - After adding**45**we have**case 2**: Set**29**in red, a**26**and**34**in black.

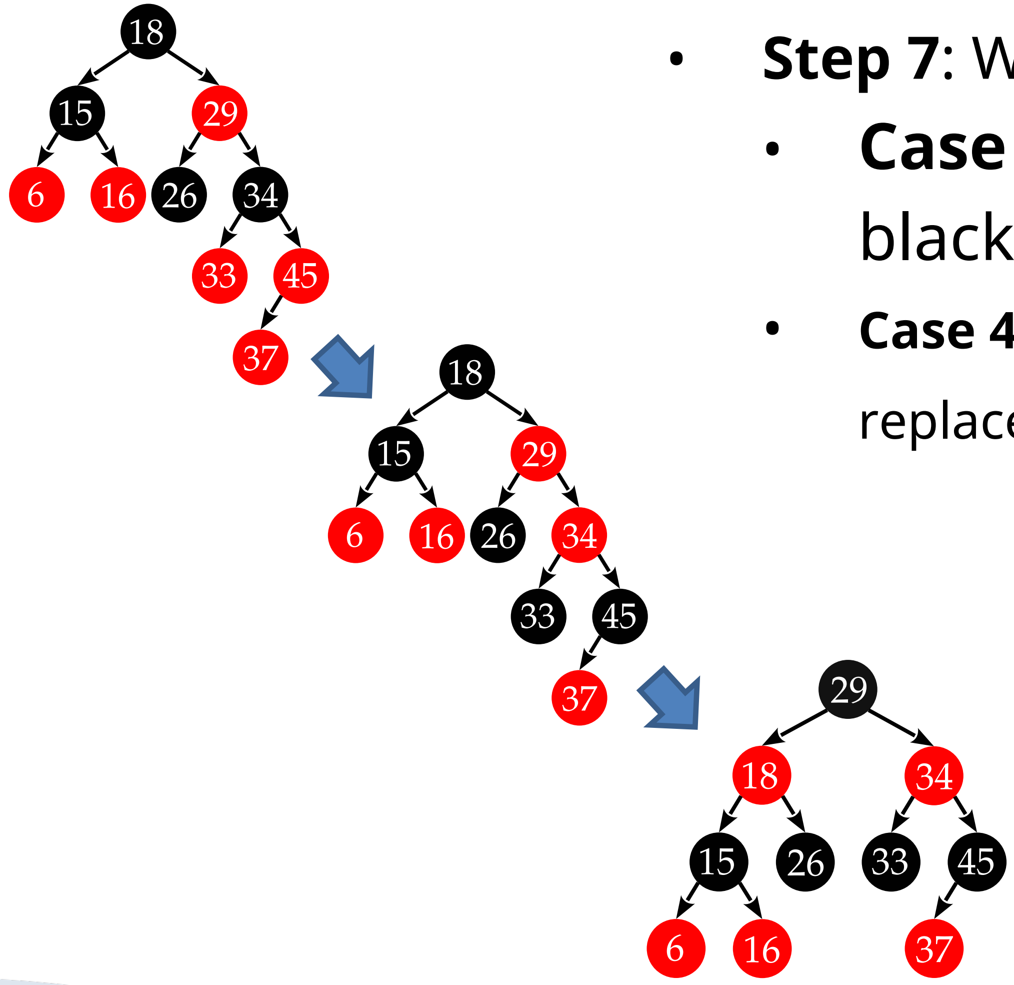
- **Step 6**: We add keys to the tree**33** and**6**.



  - After adding**6**we have**case 4**: right rotation**15**eye**16**and swapping colors in between**15**and**16**

16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40

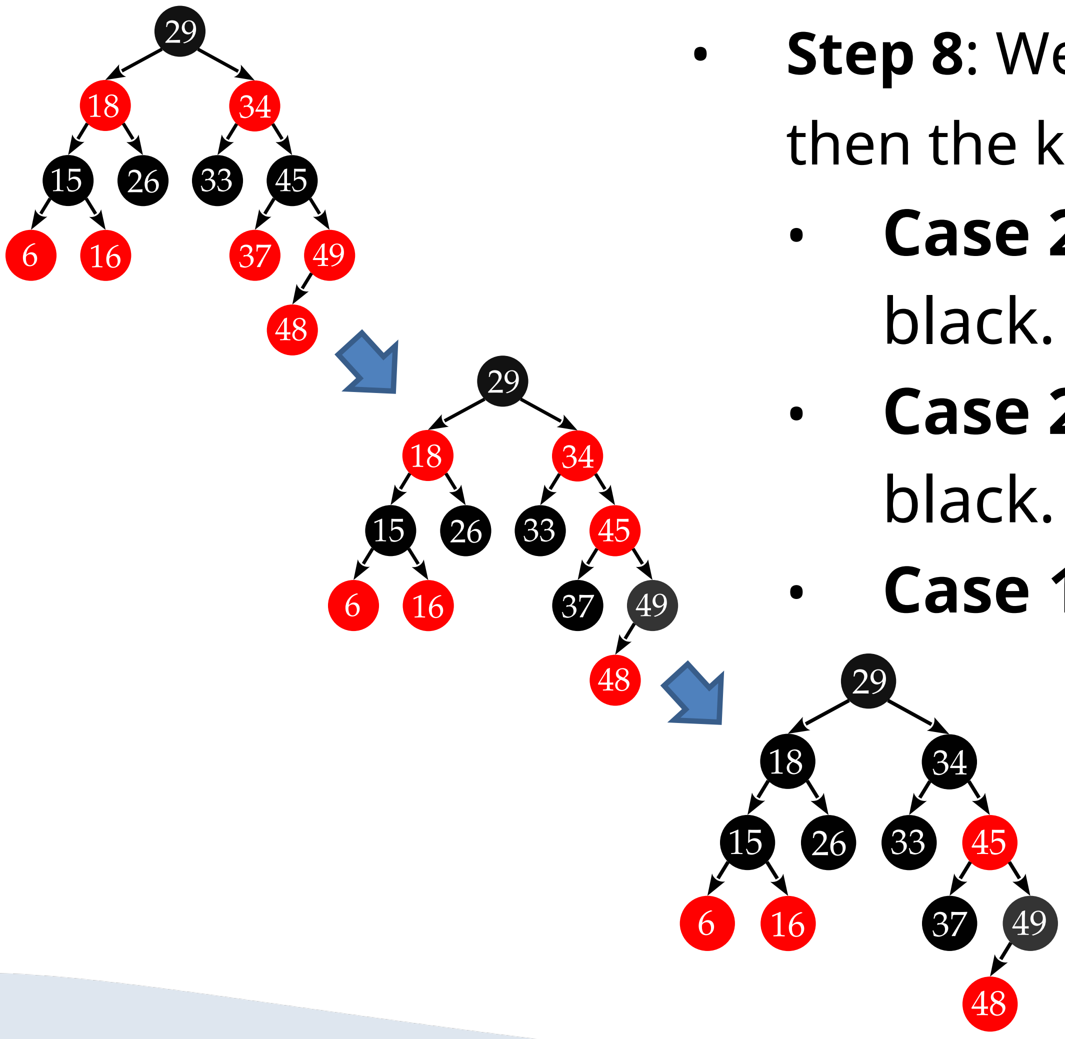
Example of adding data to the RB-tree



- **Step 7**: We add a key to the tree**37**.
  - **Case 2**: Set**34**in red, a**33**and **45**in black.
  - **Case 4**: Left rotation**29**eye**18**and color replacement**18**and**29**.

16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40

Example of adding data to the RB-tree

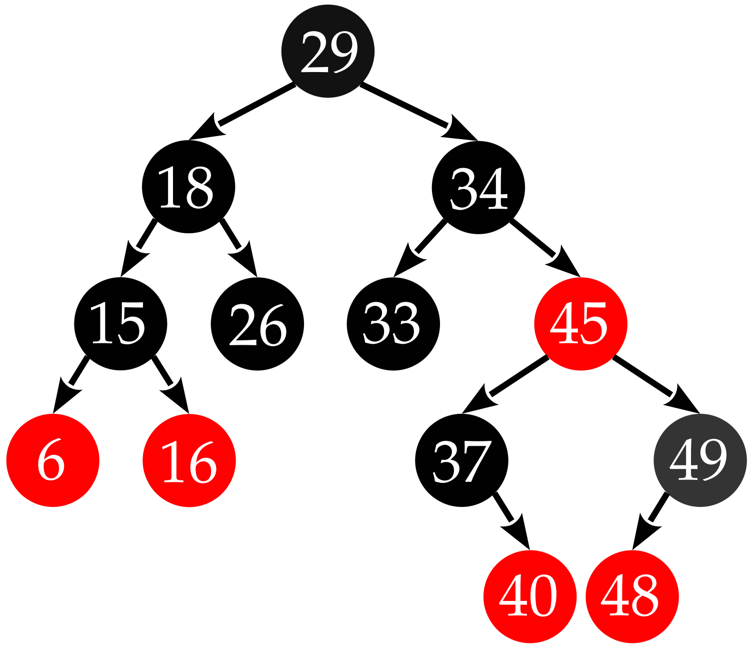


- **Step 8**: We add a key to the tree**49**, and then the key**48**.
  - **Case 2**: Set**45**in red, a**37**and **49**in black.
  - **Case 2**: Set**29**in red, a**18**and **34**in black.
  - **Case 1**: Set the root**29**in black.

16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40

Example of adding data to the RB-tree

- **Step 9**: We add a key to the tree**40**



16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40

Implementation of adding a node to the RB-tree

```
procedure RBTreeInsert($rbtree, N$)
    $P \leftarrow parent(N)$
    $G \leftarrow parent(P)$
    while $P$ is 🔴 do
        if $P$ is the left child then
            $case \leftarrow LL$
            if $N$ is the right child then
                $case \leftarrow LR$
            $U \leftarrow$ the right child of $G$
        else
            $case \leftarrow RR$
            if $N$ is the left child then
                $case \leftarrow RL$
            $U \leftarrow$ the left child of $G$
        if $U$ and $P$ are 🔴 then
            $P \leftarrow U \leftarrow$ ●
            $G \leftarrow$ 🔴
            $N \leftarrow G$
        else if $P$ is 🔴 and $U$ is ● then
            if $case \in \{LL, RR\}$ then                          ▷ straight cases
                if $case$ is $LL$ then
                    right rotate $P$ around $G$
                else
                    left rotate $P$ around $G$
                switch $P$ and $G$ colors
                break
            else                                                   ▷ broken cases
                if $case$ is $LR$ then
                    right rotate $N$ around $P$
                else
                    left rotate $N$ around $P$
                $N \leftarrow P$
        $P \leftarrow parent(N)$
        $G \leftarrow parent(P)$
    $root(rbtree) \leftarrow$ ●                                    ▷ Rule 2
```

Deleting a node in the RB-tree

- Algorithm:
  1. Delete by copying (substitute node; hereafter labeled X)
  2. Remove replacement node; he can have at most one child, so the problem is simplified

- If the replacement node is:
  - red: the properties of the RB-tree are not violated, the procedure is finished
  - **black**: a more complex procedure

Blackhead removal

- There are 3 possible problems after removing a black node:
  1. if the root is removed, it could have only one child (N) which becomes the new root, and that can be ired
     - violation of the 2nd rule (the root is black)
  2. after removing X, its child N and parent P are in a child-parent relationship and if bothRed
     - violation of the 4th rule (childrenredare black)
  3. removing black X means reducing the black height of all its predecessors (ancestors)
     - violation of the 5th rule
- For the first case, it is enough to recolor N in black and everything is solved, because by changing the color of the root, the black height of all nodes of the tree changes equally. The other two cases depend on the color of node N.

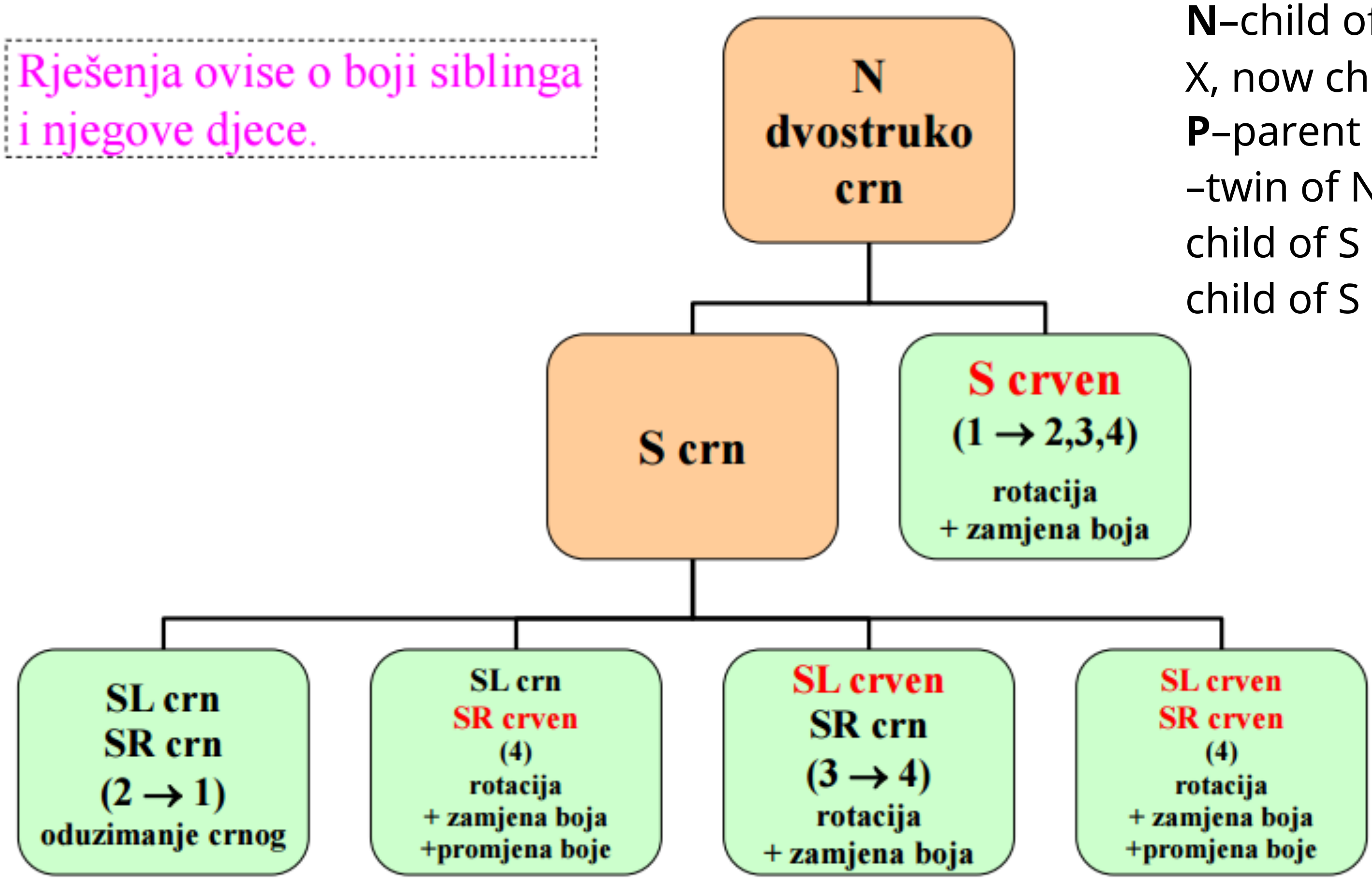
Blackhead removal

- Let's imagine that we can somehow transfer the blackness of X to N. Then by removing X we would not lose it and the RB rules would not be violated:
  - If N was previouslyred, will become red-black and contribute 1 to the black height.
  - If N was previously black, it will become double black and contribute 2 to the black height.

- Solution:
  - If it is red-black, it is enough to repaint it in pure black.
  - If it is double black, the idea is to pass the excess black to the predecessor and thus raise that excess until it reaches a place where we can permanently incorporate it into the tree or until it reaches the root where we can ignore it.

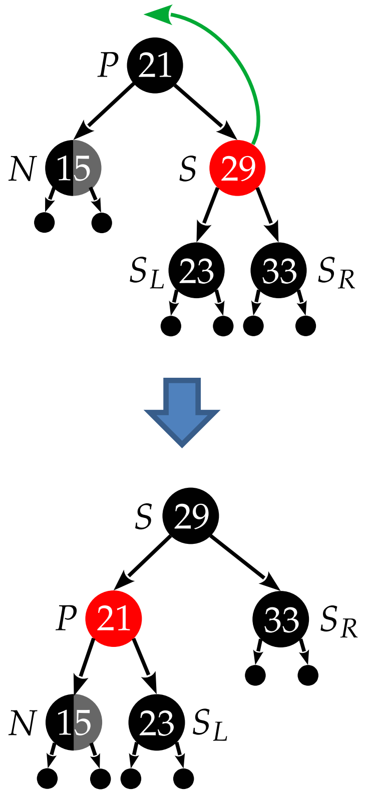
Black node removal (double black)

- 4 (+4 symmetrical) are possible cases, and they depend on the color of the twin node (child of the same parent as N) and its children


Rješenja ovise o boji siblinga i njegove djece.
N dvostruko crn
S crn
S crven (1 → 2,3,4) rotacija + zamjena boja
SL crn SR crn (2 → 1) oduzimanje crnog
SL crn SR crven (4) rotacija + zamjena boja +promjena boje
SL crven SR crn (3 → 4) rotacija + zamjena boja
SL crven SR crven (4) rotacija + zamjena boja +promjena boje

Labels:
N–child of removed X, now child of P **P**–parent of N **WITH**–twin of N **FIG**–left child of S **SR**–right child of S

Black node removal (double black)



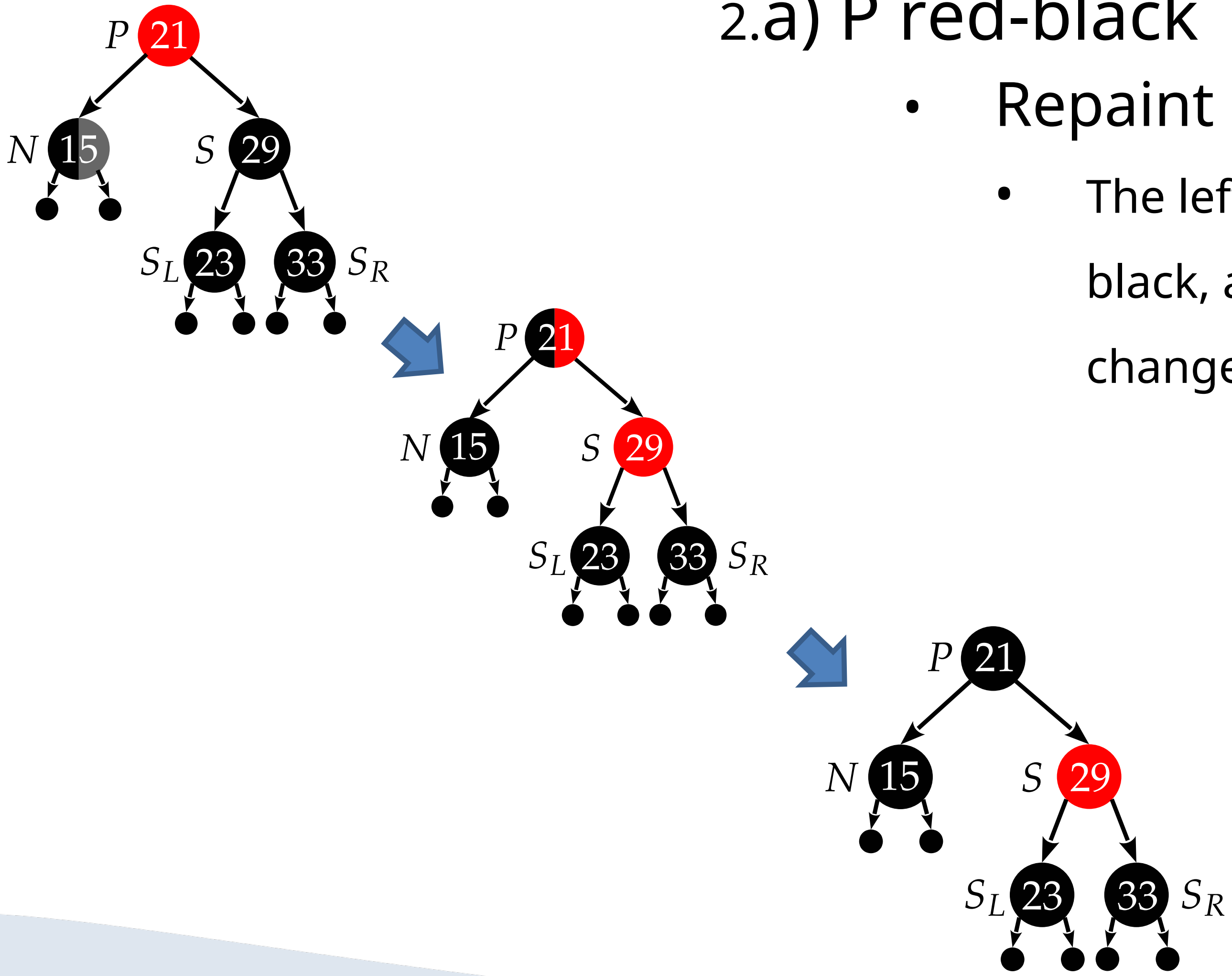
1. Twin S is red
    - P must be black because it has red child
    - After deleting X the black height of the left subtree of P is one less than the black height of the right subtree (ie N double black)
    - Solution: rotate S around P (symmetry) and swap the colors of P and S
    - **CONTINUATION** balancing from N

Black node removal (double black)

2.S black, children of S black

- Take away one black Nu and Su; N remains solid black and S becomesred
- Pass that excess black to a higher level (convergence!), i.e. Pu, which thereby becomes either red-black or double black
- The procedure after the intervention depends on Pu (cases 2a and 2b):

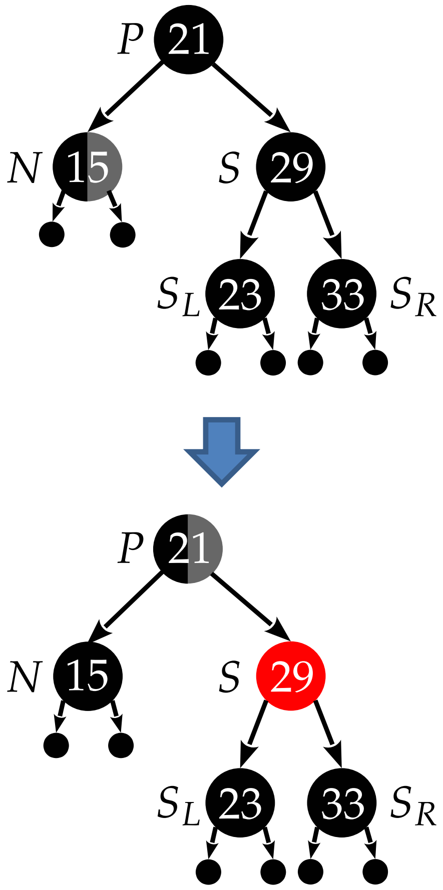
Black node removal (double black)



2.a) P red-black

- Repaint P black
  - The left subtree thus gets the lost black, and the right one does not change anything because S red; **END**

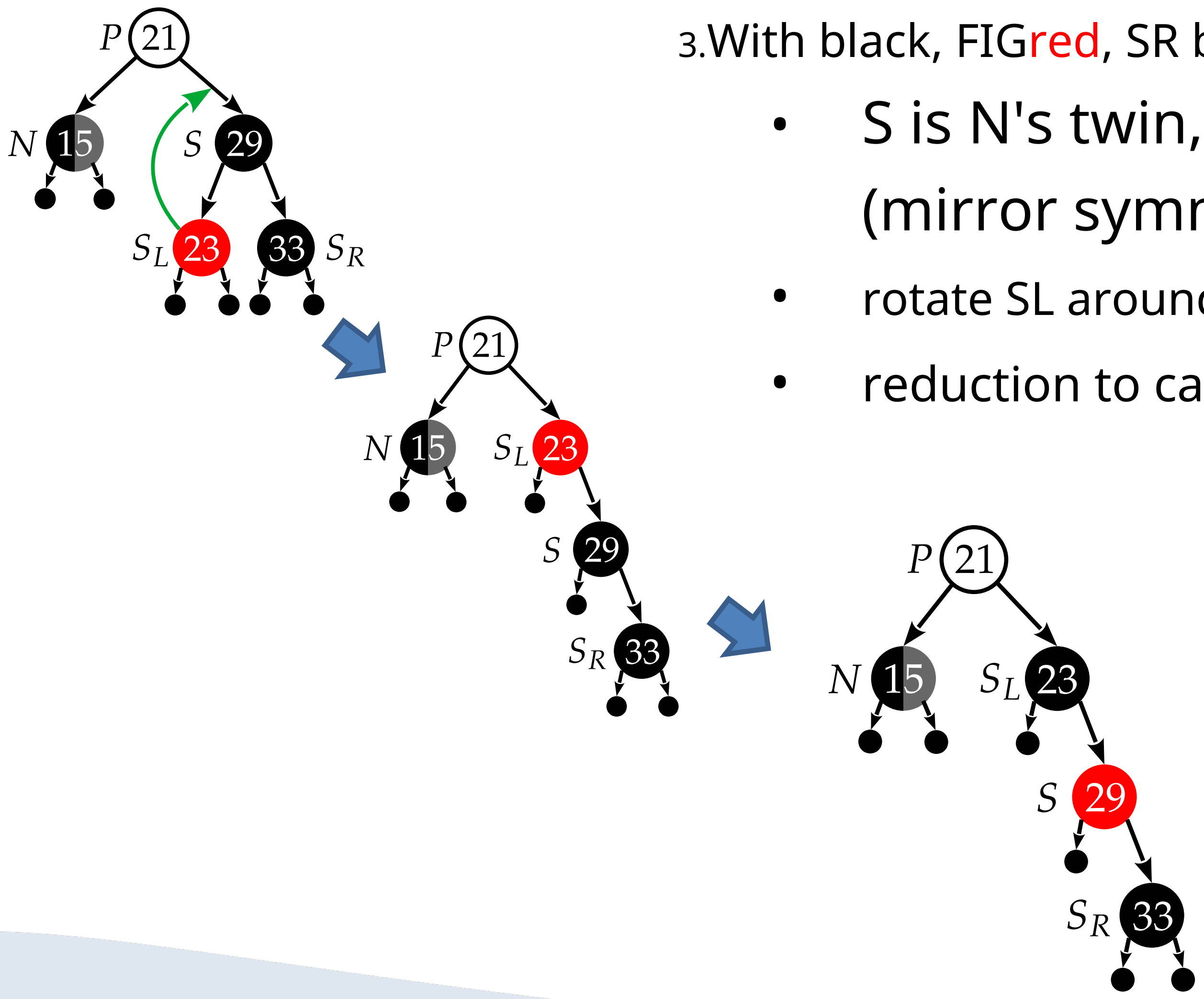
Black node removal (double black)



2.b) P double black

- P is the root
  - excess black is discarded;**END**
- P is not a root
  - back to case 1 viewing P as N; **CONTINUATION**
  - the problem is a level higher (convergence!)

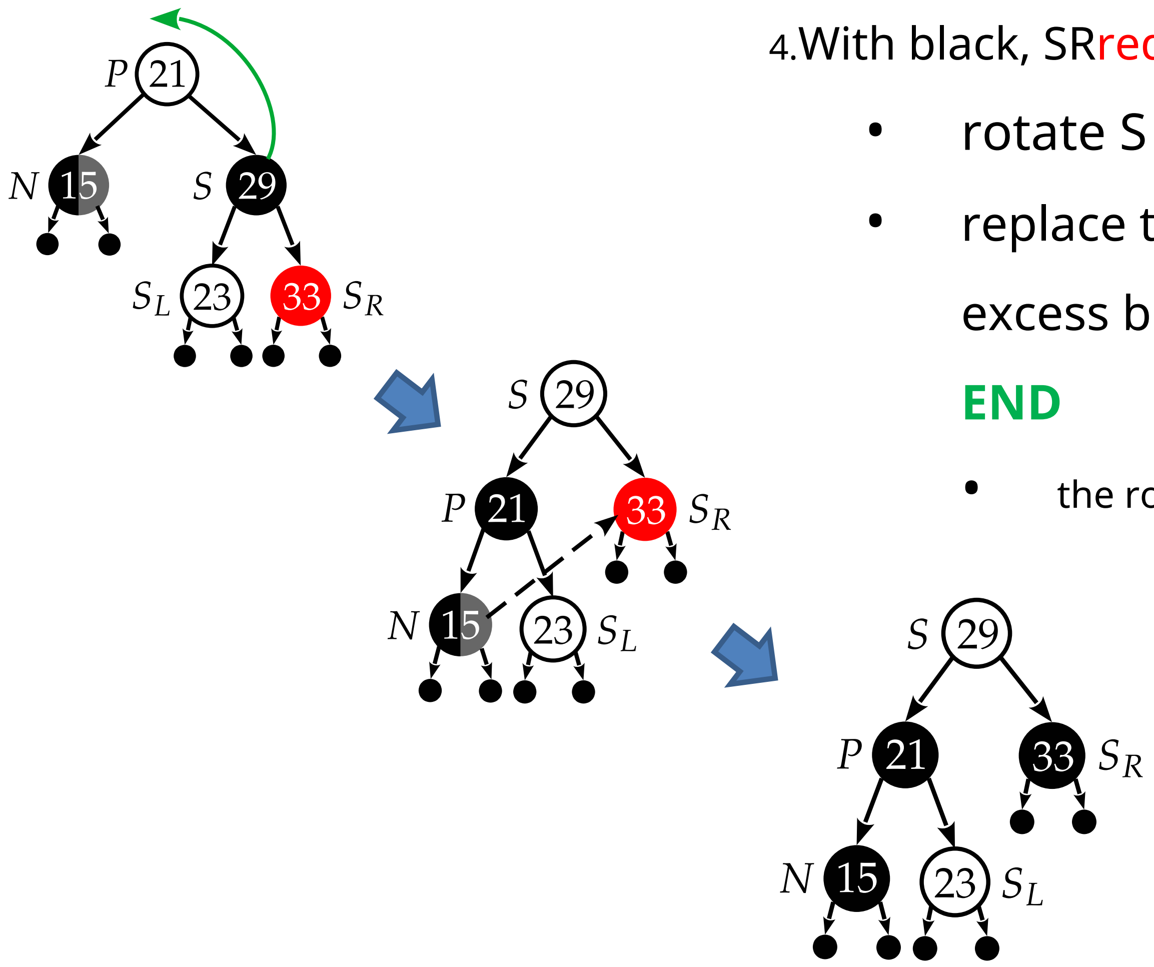
Black node removal (double black)



3.With black, FIGred, SR black, P unimportant

- S is N's twin, and N isleft child of P (mirror symmetry!)
- rotate SL around S and swap their colors
- reduction to case 4;**CONTINUATION**

Black node removal (double black)



4. With black, SRred, P and SL unimportant

- rotate S around P (mirror symmetry!)
- replace the colors S and P, and transfer the excess black from N to SR (repaint in black);

END

- the root of the subtree remains the same color

Implementation of node deletion in the RB-tree

```
procedure RBTreeRemove($rbtree, N$)
    while $N$ is not $root(rbtree)$ and $N$ is ● do
        $P \leftarrow$ the parent of $N$
        if $N$ is the left child of $P$ then
            $S \leftarrow$ the right child of $P$
            $S_L, S_R \leftarrow$ children of $S$
            if $S$ is 🔴 then
                $S \leftarrow$ ●
                $P \leftarrow$ 🔴
                left rotate $S$ around $P$
            if $S_L$ is ● and $S_R$ is ● then
                $S \leftarrow$ 🔴
                $N \leftarrow$ the parent of $N$
            else
                if $S_R$ is ● then
                    $S_L \leftarrow$ ●
                    $S \leftarrow$ 🔴
                    right rotate $S_L$ around $S$
                color of $S \leftarrow$ color of $P$
                $P \leftarrow S_R \leftarrow$ ●
                left rotate $S$ around $P$
                $N \leftarrow root(rbtree)$
        else                                ▷ implement the symmetrical cases
    $N \leftarrow$ ●
```