Height Balanced Trees

Martín Hernández

Juan Mendivelso

Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

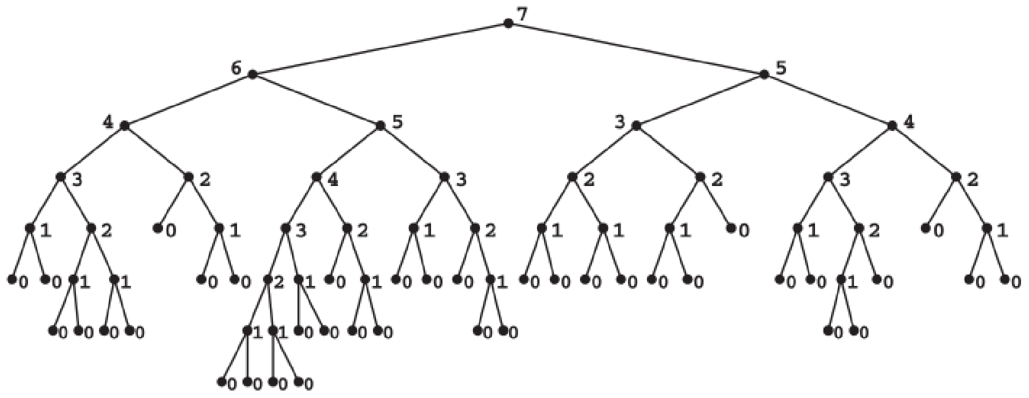
CONTENTS

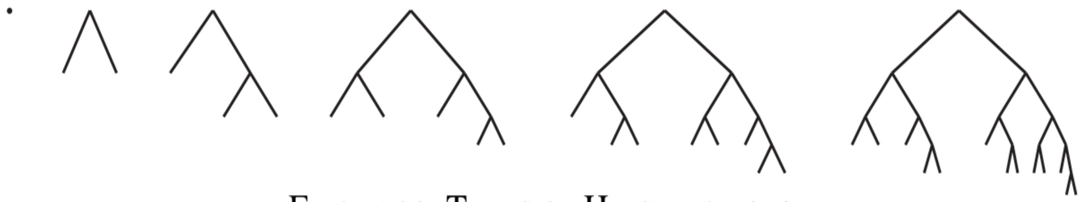
Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

1. Height-Balanced Trees

Height Balanced Trees

- Most common type of Balanced trees.
- Also the Oldest type of Balanced trees, introduced and analyzed by G.M. Adel'son-Vel'skiĭ and E.M. Landis (1962).
- At most has a difference of 1 in the height of the left and right subtrees.
- Most of the Search Trees implementation and operations have slight changes.





FIBONACCI TREES OF HEIGHT 0 TO 5

Structure and Properties

- Define an int for the height for each node
- Emphasis on the definition of the height of a node

```
typedef struct tr_n_t { key_t          key;
                        struct tr_n_t  *left;
                        struct tr_n_t *right;
                        int          height;
                     /* possibly other information */
                      } tree_node_t;
```

The height of a node `*n` is defined recursively by the following rules:

{ if `*n` is a leaf (`n->left` = `NULL`), then `n->height` = 0,

{ else `n->height` is one larger than the maximum of the height of the left and right subtrees:
`n->height` = 1 + max(`n->left->height`, `n->right->height`).

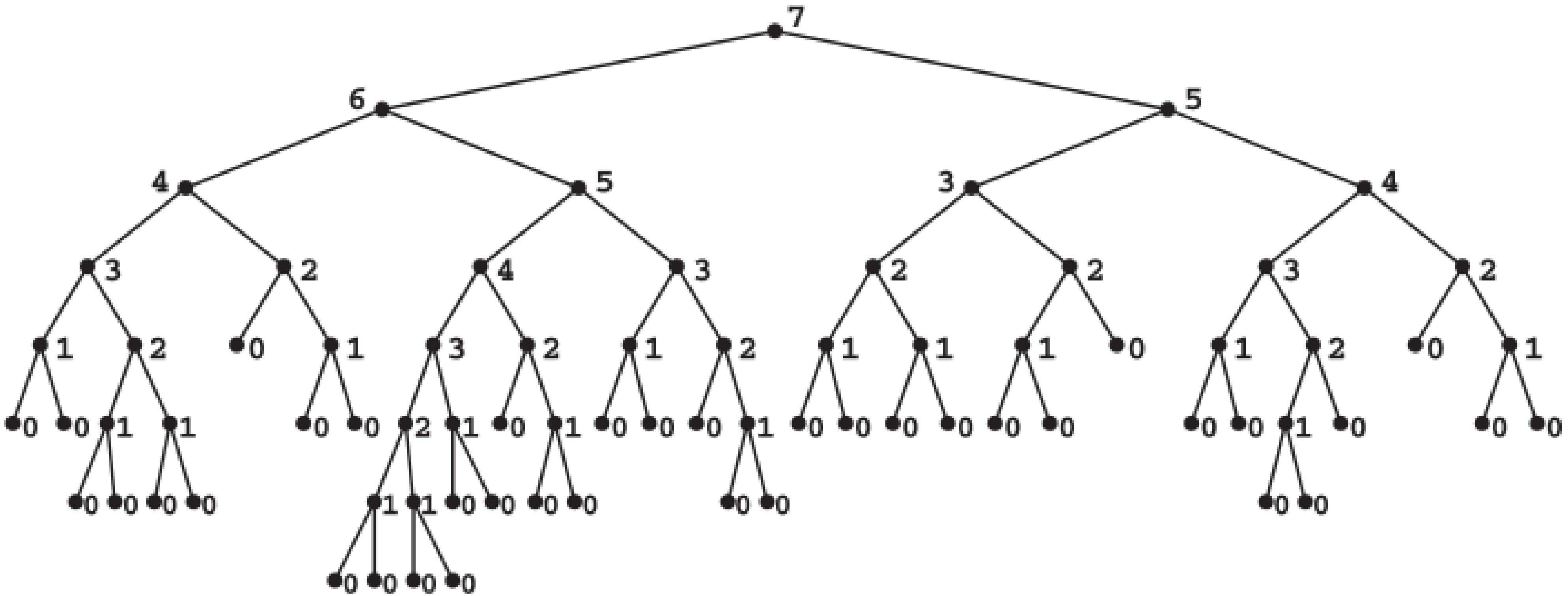
Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Structure and Properties

{ if `*n` is a leaf (`n->left` = `NULL`), then `n->height` = 0,

{ else `n->height` is one larger than the maximum of the height of the left and right subtrees:

`n->height` = 1 + max(`n->left->height`, `n->right->height`).



Height and Num. Leaves Theorem

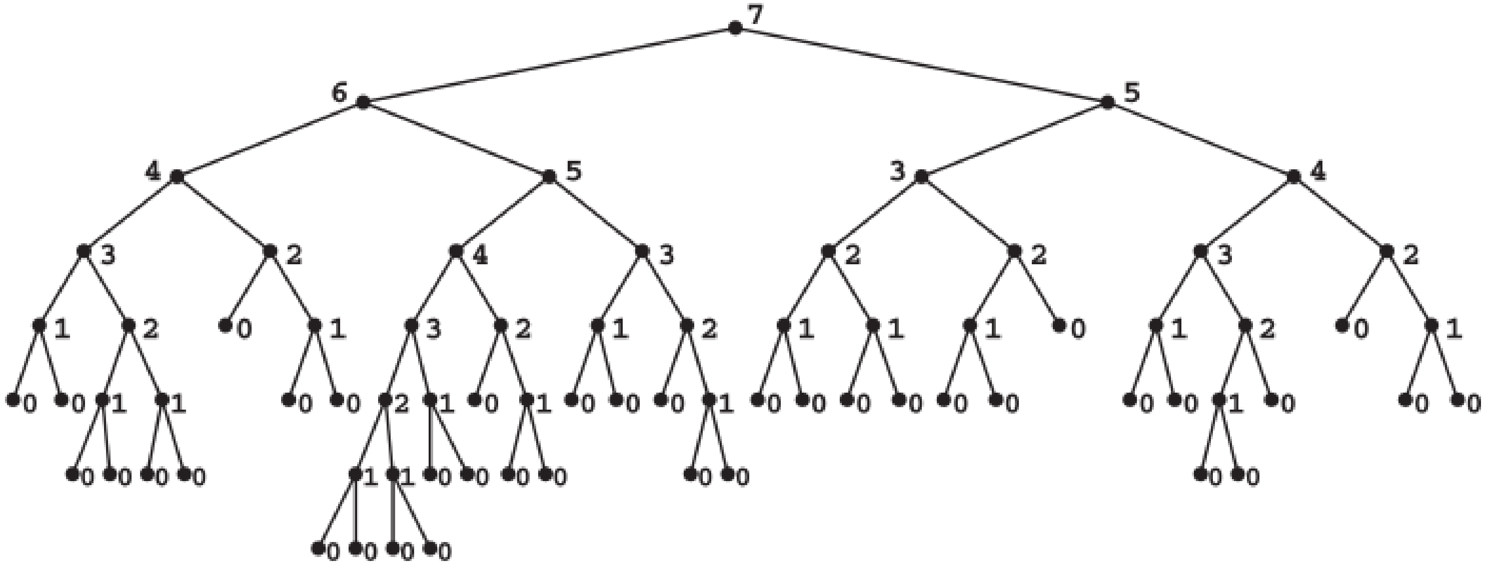
Theorem. A height-balanced tree of height h has at least

$$\left(\frac{3+\sqrt{5}}{2\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{3-\sqrt{5}}{2\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h \text{ leaves.}$$

A height-balanced tree with n leaves has height at most

$$\left\lceil \log_{\frac{1+\sqrt{5}}{2}} n \right\rceil = \lceil c_{Fib} \log_2 n \rceil \approx 1.44 \log_2 n,$$

where $c_{Fib} = (\log_2(\frac{1+\sqrt{5}}{2}))^{-1}$.



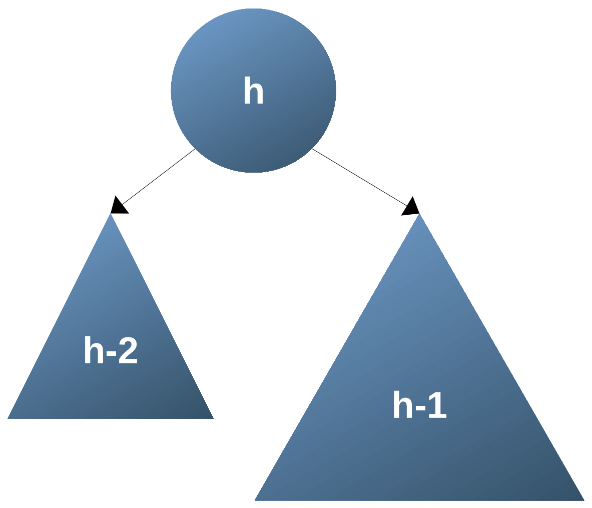
height = 7, then the tree has at least *leaves = 34*

leaves = 37, then the tree has at most *height ≈ 7.5*

Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

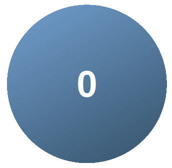
Height and Num. Leaves Theorem

- We have to prove part of this theorem.
- Suppose that we have a height balanced tree $\boldsymbol{F}$ with height h, then the left or right subtree **must** have at least a height of $h-1$, and the other subtree **must** also have at least a height of $h-2$.
- Then we can calculate the number of leaves on the tree $\boldsymbol{F}$ by using recursion with two cases:
  - leaves($\boldsymbol{F}_h$) = leaves($\boldsymbol{F}_{h-1}$) + leaves($\boldsymbol{F}_{h-2}$)
- With the cases:
  - leaves(0) = 1
  - leaves(1) = 2

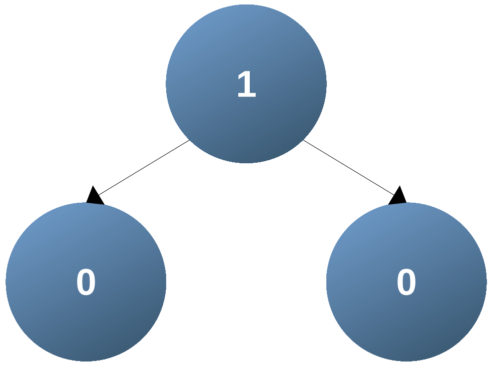


Height and Num. Leaves Theorem

- These cases come from the definition of Height Tree:
- Where if a node has a height of 0 then is a leaf

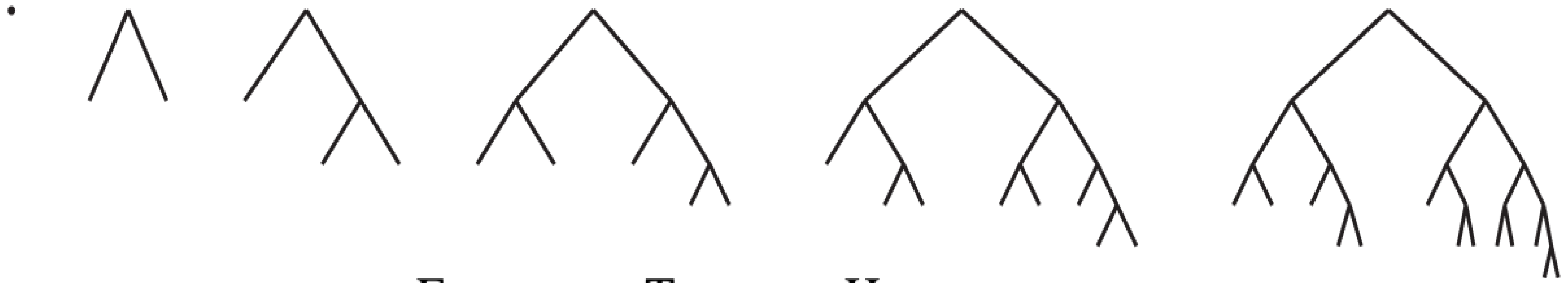


- And where a tree with height of 1 has at least 2 leaves.



Height and Num. Leaves Theorem

- If we continue to calculate the cases where the height of the balanced tree is greater than 1, we will see that the number of leaves of the tree has the same behavior as the Fibonacci Sequence:



Fibonacci Trees of Height 0 to 5

Height and Num. Leaves Theorem

- Then by **solving linear recurrence** of this function we will reach the solution and what we have to prove:

$$leaves(h) = \left(\frac{3+\sqrt{5}}{2\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{3-\sqrt{5}}{2\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h$$

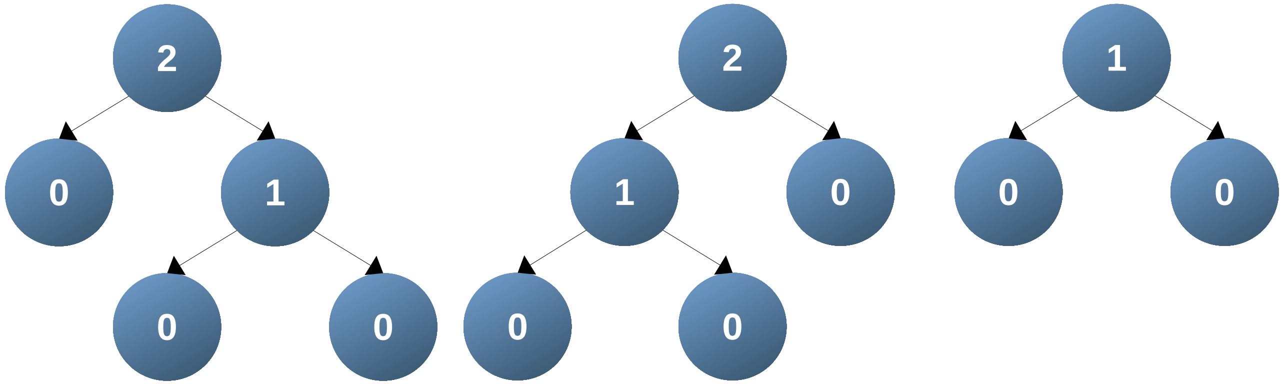
Cases of Balancing

*n is the current node

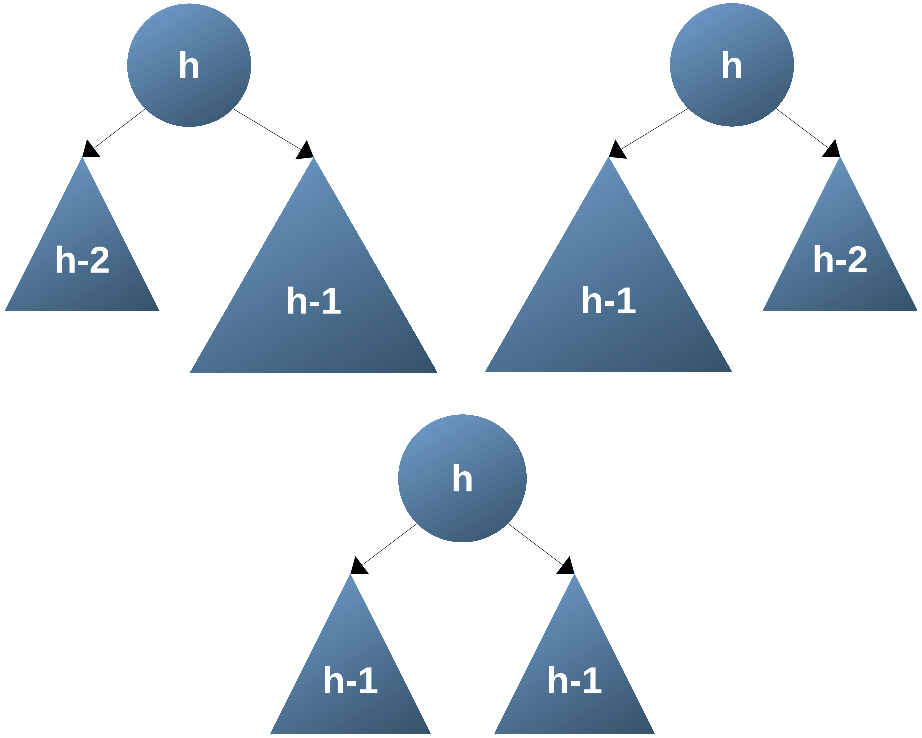
- Trivial case (doesn't balance anything):

$$|\texttt{n->left->height} - \texttt{n->right->height}| \leq 1.$$

- Examples:



- General cases:



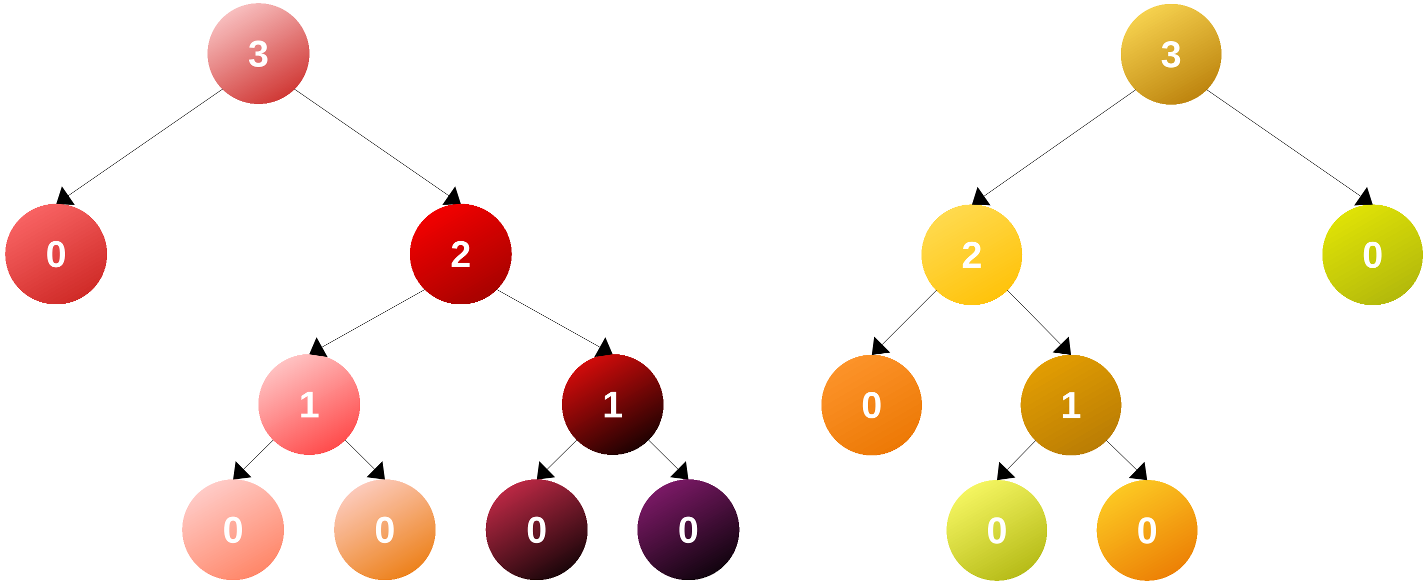
Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Cases of Balancing

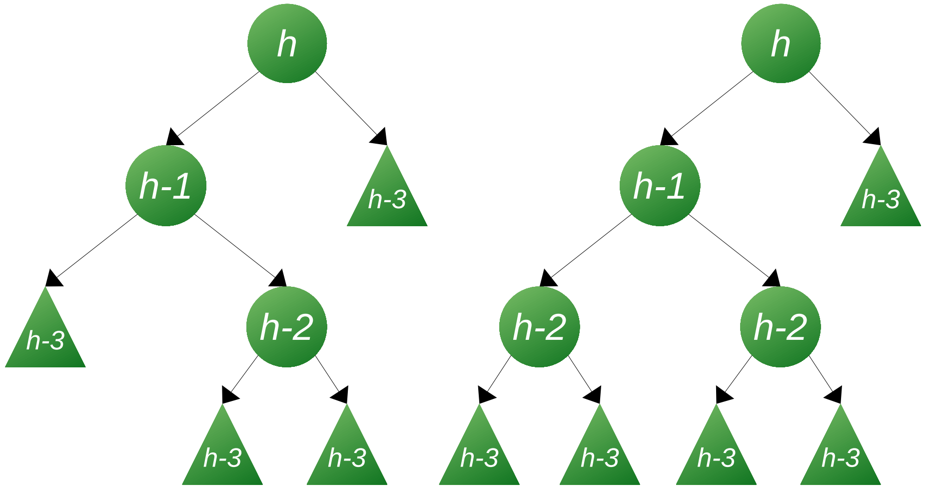
*n is the current node

- Balancing the tree is required:
  $|\texttt{n->left->height} - \texttt{n->right->height}| = 2.$
- From this case, there's 4 cases

- Examples:



- General cases:



Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

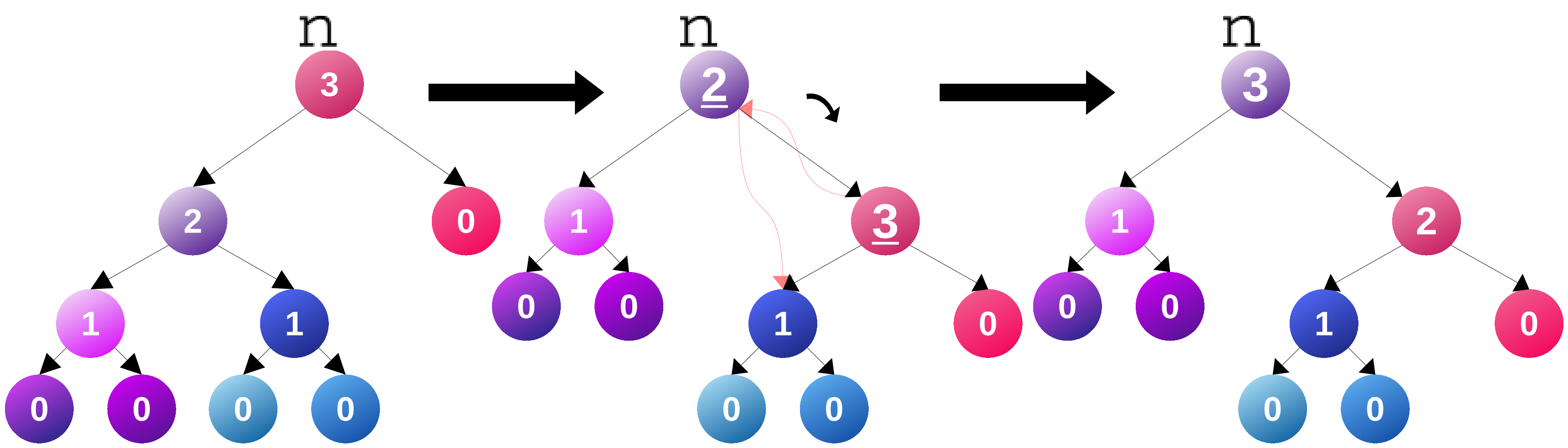
Cases of Balancing

*n is the current node

- Right Simple Rotation (type 2.1):

  If `n->left->height` = `n->right->height` + 2 and `n->left->left->height` = `n->right->height` + 1.

  Rotates right the current *n node, and recomputes the height on *n and the rotated side.




```
if(n->left->left->height -
    n->right->height == 1 )
{   right_rotation(n);
    n->right->height =
    n->right->left->height + 1;
    n->height =
       n->right->height + 1;
}
```

Cases of Balancing

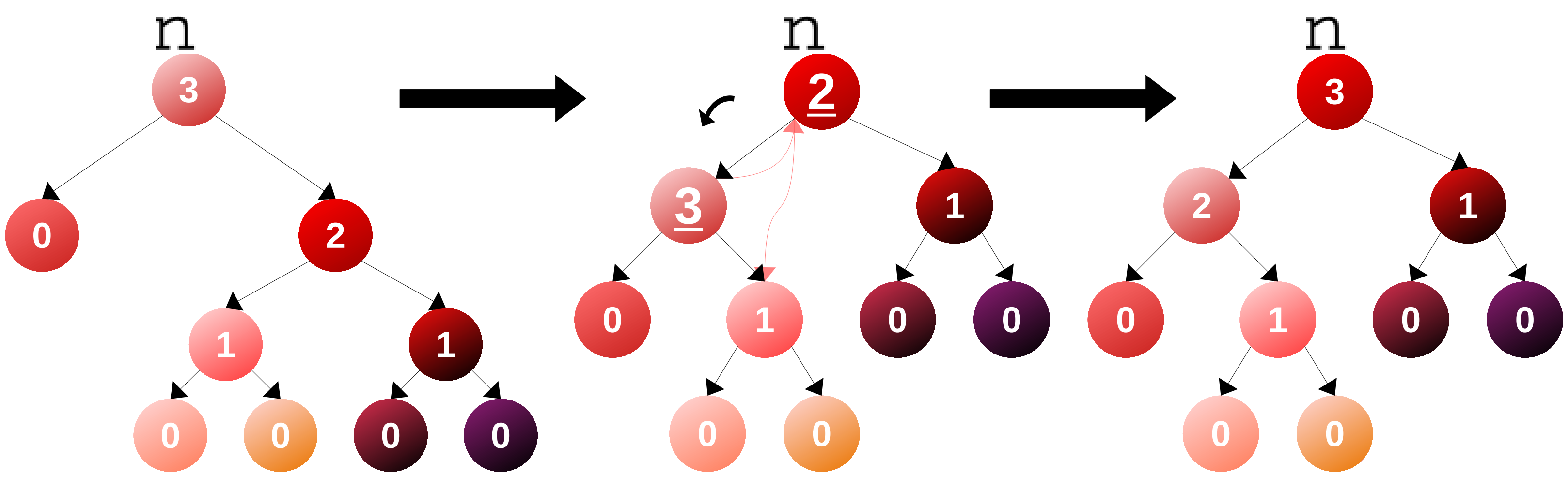
*n is the current node

- Left Simple Rotation (type 2.3):

  If n->right->height = n->left->height + 2 and

  n->right->right->height = n->left->height + 1.

  Rotates left the current *n node, and recomputes the height on *n and the rotated side.




```
if( n->right->right->height -
    n->left->height == 1 )
{   left_rotation(n);
    n->left->height =
    n->left->right->height + 1;
    n->height =
       n->left->height + 1;
}
```


Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

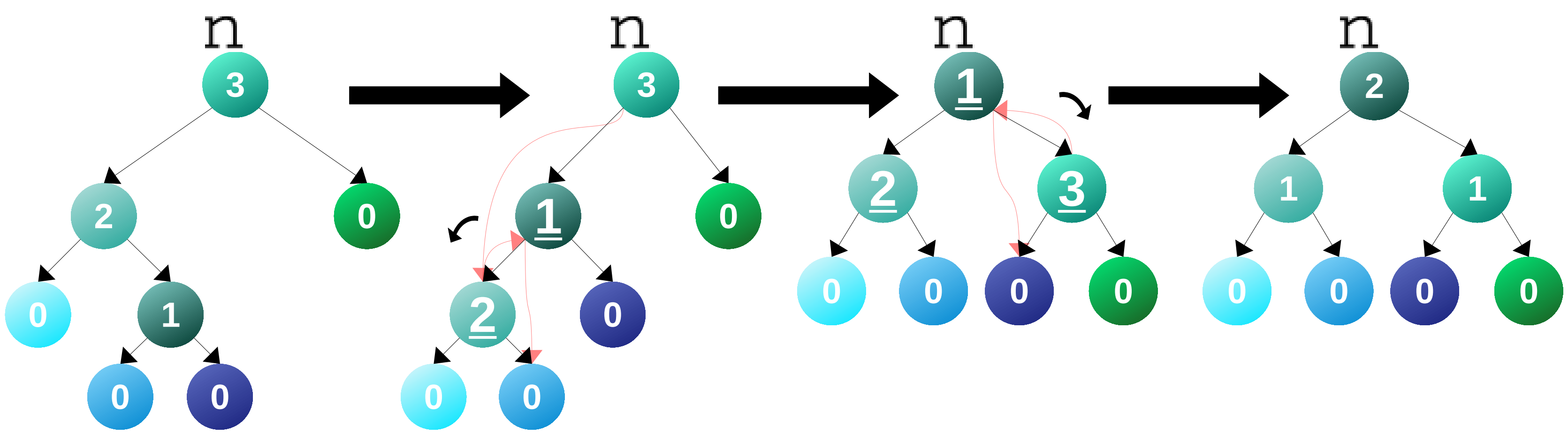
Cases of Balancing

*n is the current node

- Left Double Rotation (types 2.2):

If `n->left->height` = `n->right->height` + 2 and `n->left->left->height` = `n->right->height`.

First, rotates left **on** the left node, then rotates right **on** *n. Finally, recomputing the height in the rotated nodes.




```
left_rotation( n->left );
right_rotation( n );
tmp_height =
    n->left->left->height;
n->left->height  =
   tmp_height + 1;
n->right->height  =
   tmp_height + 1;
n->height = tmp_height + 2;
```

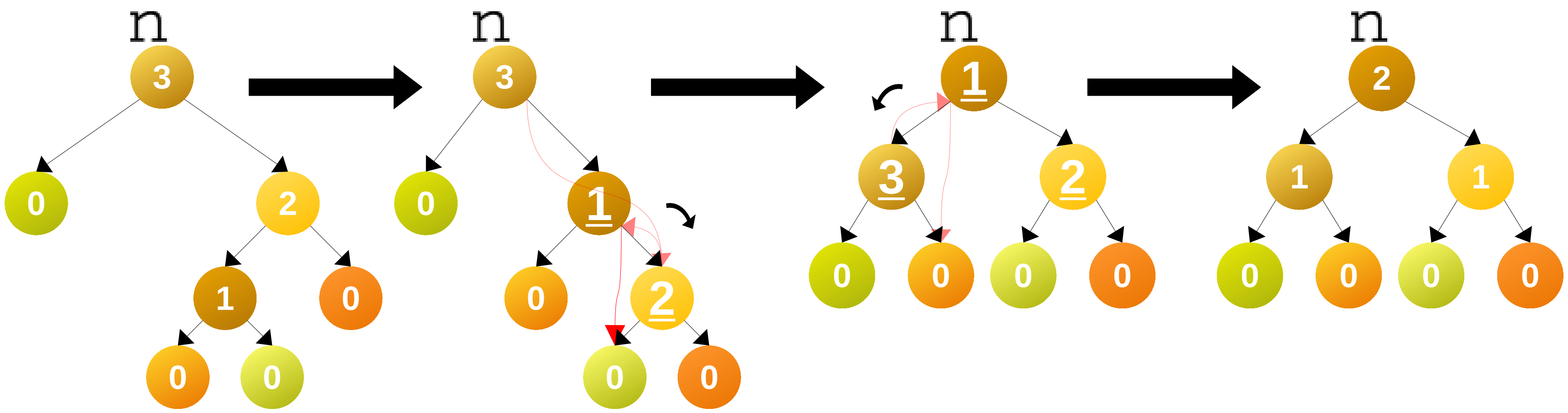
Cases of Balancing

*n is the current node

- Right Double Rotation (types 2.4):

If `n->right->height` $=$ `n->left->height` $+ 2$ and `n->right->right->height` $=$ `n->left->height`.

First, rotates right **on** the right node, then rotates left **on** *n. Finally, recomputing the height in the rotated nodes.




```
right_rotation(n->right);
left_rotation(n);
tmp_height =
    n->right->right->height;
n->left->height  =
    tmp_height + 1;
n->right->height =
    tmp_height + 1;
n->height = tmp_height + 2;
```

Cases of Balancing

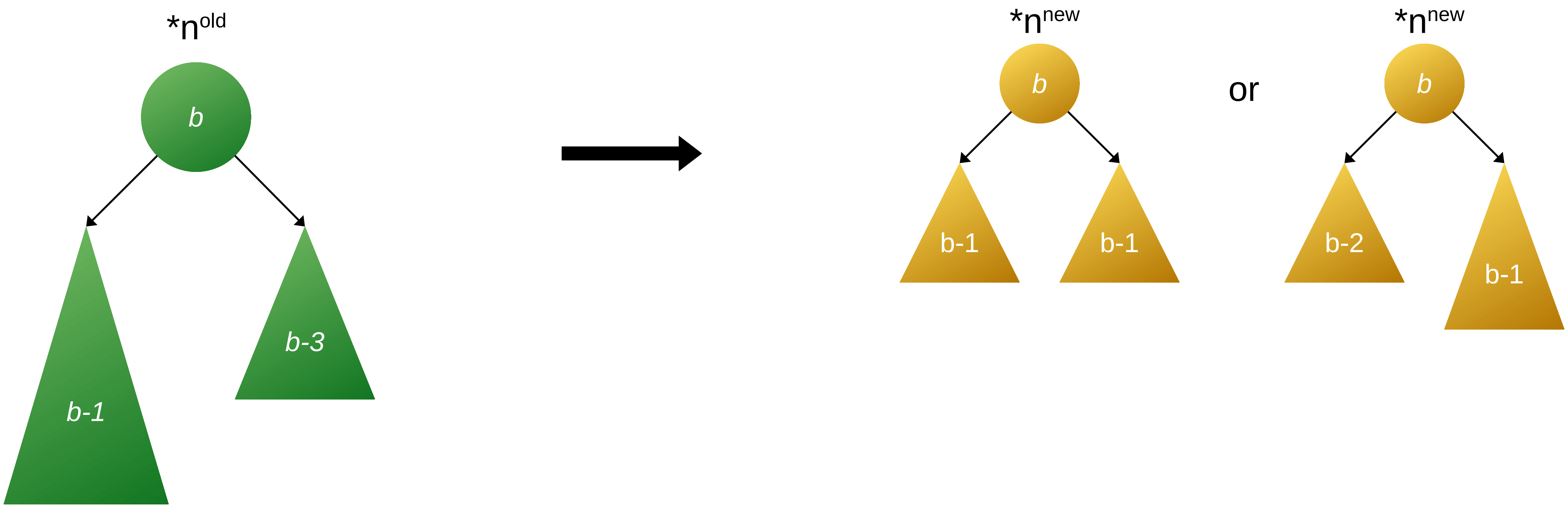
- The cases 2.1 and 2.3, 2.2 and 2.4 have the same logic, but the inverse execution.

- At most in the balancing we have to do two rotations and three recomputations of the height on a node, making up to $O(logn)$ time.

- Yet, we have to prove that this process balances the tree.

Cases of Balancing

- Let $*n^{old}$ a node which is height balanced but the subtrees' height differs by 2.
- Let $*n^{new}$ the same node after the rebalancing step.



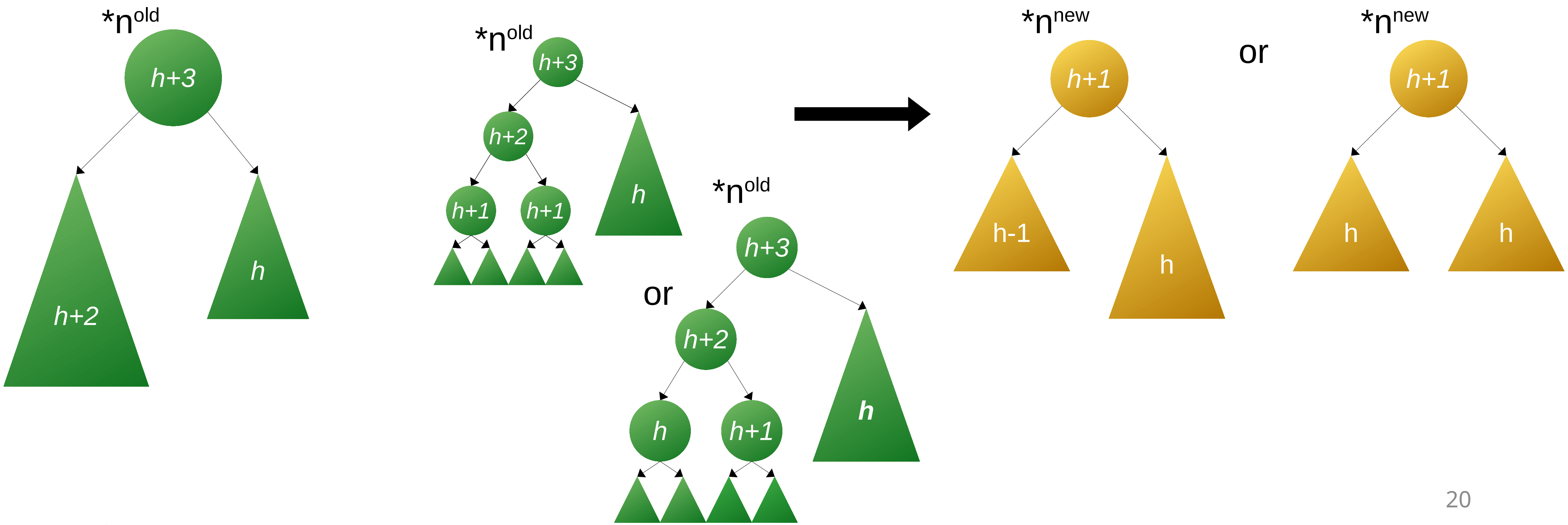
Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Cases of Balancing

- Then we can assume that:

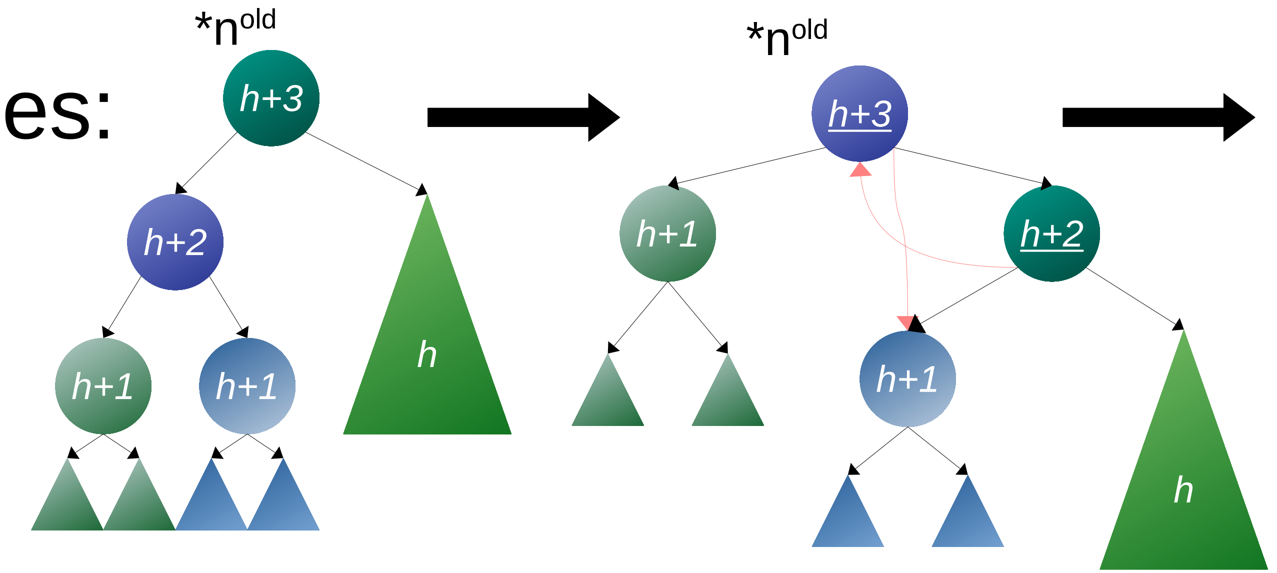
$$\texttt{n}^{\text{old}}\texttt{->left->height} = \texttt{n}^{\text{old}}\texttt{->right->height} + 2.$$

- Let $h = \texttt{n}^{\text{old}}\texttt{->right->height}.$
- Then $\max(\texttt{n}^{\text{old}}\texttt{->left->left->height}, \texttt{n}^{\text{old}}\texttt{->left->right->height}) = h + 1$

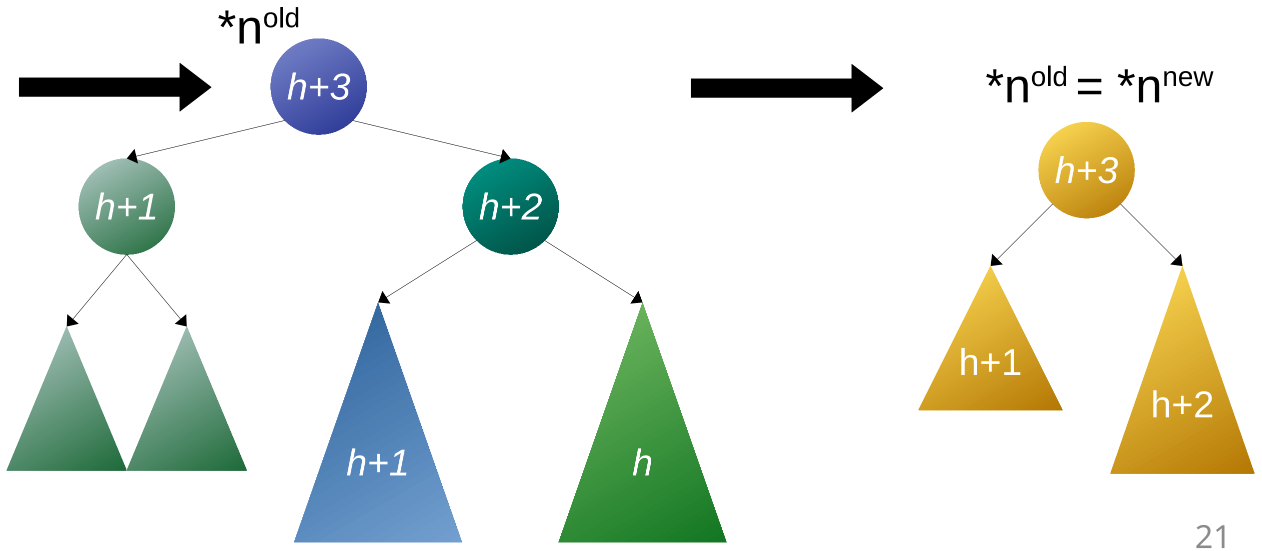


Cases of Balancing

- Then we have the next cases:

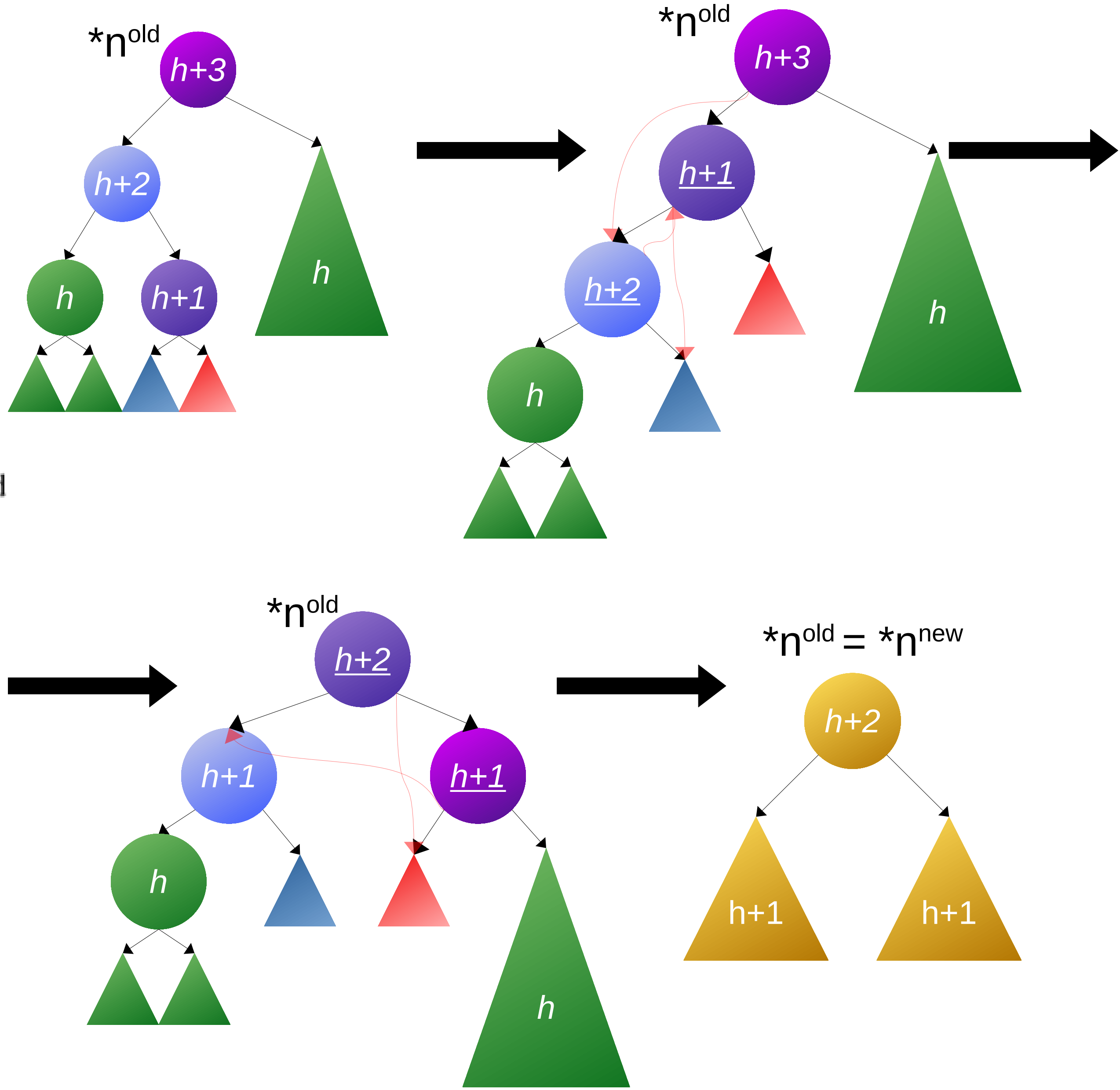


(a) $\texttt{n}^{\text{old}}\texttt{->left->left->height} = h+1$ and
$\texttt{n}^{\text{old}}\texttt{->left->right->height} \in \{h, h+1\}$.
By rule 2.1 we perform a right rotation around $\texttt{n}^{\text{old}}$.
By this $\texttt{n}^{\text{old}}\texttt{->left->left}$ becomes $\texttt{n}^{\text{new}}\texttt{->left}$,
$\texttt{n}^{\text{old}}\texttt{->left->right}$ becomes $\texttt{n}^{\text{new}}\texttt{->right->left}$, and
$\texttt{n}^{\text{old}}\texttt{->right}$ becomes $\texttt{n}^{\text{new}}\texttt{->right->right}$.
So $\texttt{n}^{\text{new}}\texttt{->left->height} = h+1$,
$\texttt{n}^{\text{new}}\texttt{->right->left->height} \in \{h, h+1\}$,
$\texttt{n}^{\text{new}}\texttt{->right->right->height} = h$.
Thus, the node $\texttt{n}^{\text{new}}\texttt{->right}$ is height-balanced, with
$\texttt{n}^{\text{new}}\texttt{->right->height} \in \{h+1, h+2\}$.
Therefore, the node $\texttt{n}^{\text{new}}$ is height-balanced.



Cases of Balancing

(b) n$^{\text{old}}$`->left->left->height` $= h$ and
n$^{\text{old}}$`->left->right->height` $= h + 1$.
By rule 2.2 we perform left rotation around n$^{\text{old}}$`->left`, followed by a right rotation around n$^{\text{old}}$. By this
n$^{\text{old}}$`->left->left` becomes n$^{\text{new}}$`->left->left`,
n$^{\text{old}}$`->left->right->left` becomes n$^{\text{new}}$`->left->right`,
n$^{\text{old}}$`->left->right->right` becomes n$^{\text{new}}$`->right->left`, and
n$^{\text{old}}$`->right` becomes n$^{\text{new}}$`->right->right`.
So n$^{\text{new}}$`->left->left->height` $= h$,
n$^{\text{new}}$`->left->right->height` $\in \{h - 1, h\}$,
n$^{\text{new}}$`->right->left->height` $\in \{h - 1, h\}$,
n$^{\text{new}}$`->right->right->height` $= h$.
Thus, the nodes n$^{\text{new}}$`->left` and n$^{\text{new}}$`->right` are height-balanced, with n$^{\text{new}}$`->left->height` $= h + 1$ and
n$^{\text{new}}$`->right->height` $= h + 1$.
Therefore, the node n$^{\text{new}}$ is height balanced.



Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Implementation of the Insert Method

- The insert method has a lot of similarities with the insert on search trees.
- The main addition to the method is a stack based rebalancing process.
- In this rebalancing process we make use of the previously mentioned cases of rebalancing.

Implementation of the Insert Method

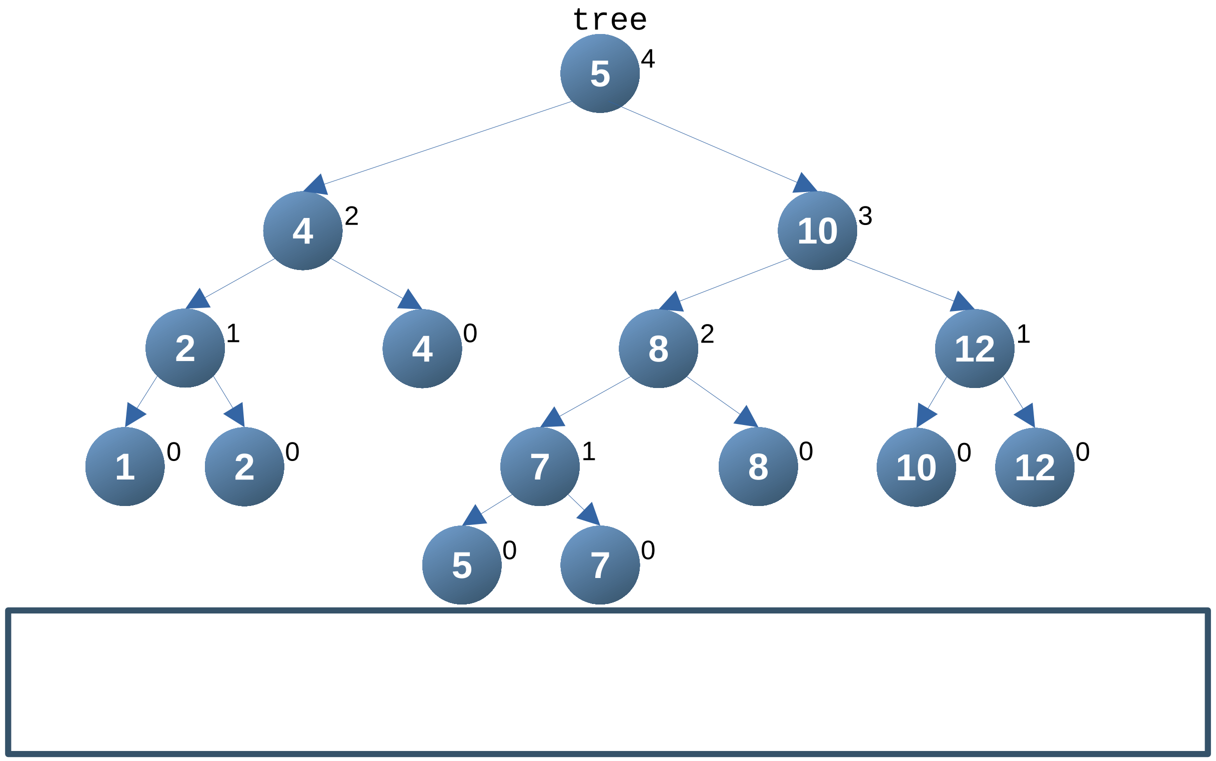
- First thing is setting up local variables and checking the case where the tree is empty.

- Iter through the tree until we reach a leaf.
  Pushing each visited node in a stack.

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
    else
    {   create_stack();
        tmp_node = tree;
        while( tmp_node->right != NULL )
        {   push( tmp_node );
            if( new_key < tmp_node->key )
                 tmp_node = tmp_node->left;
            else
                 tmp_node = tmp_node->right;
        }
```

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                     tmp_node = tmp_node->left;
              else
                     tmp_node = tmp_node->right;
         }
```

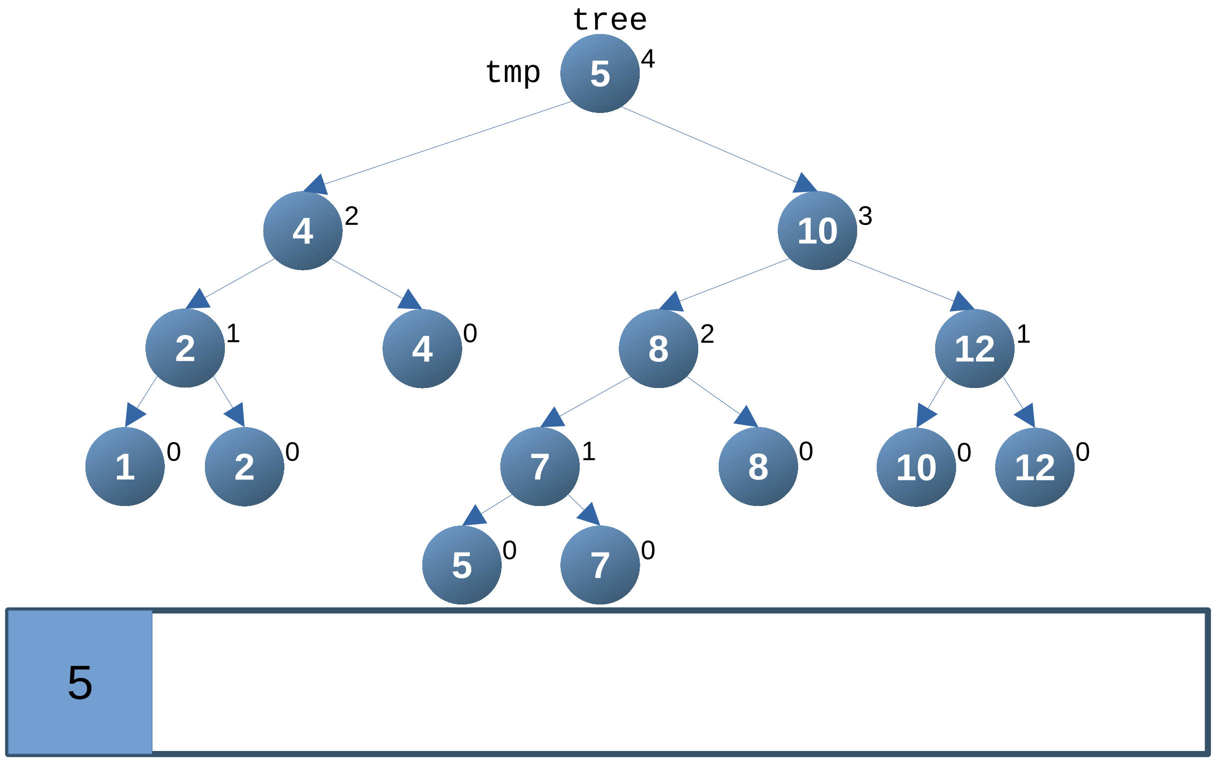



```
new_object = 6

tmp_node = NULL
finished = NULL
```

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                     tmp_node = tmp_node->left;
              else
                     tmp_node = tmp_node->right;
         }
```

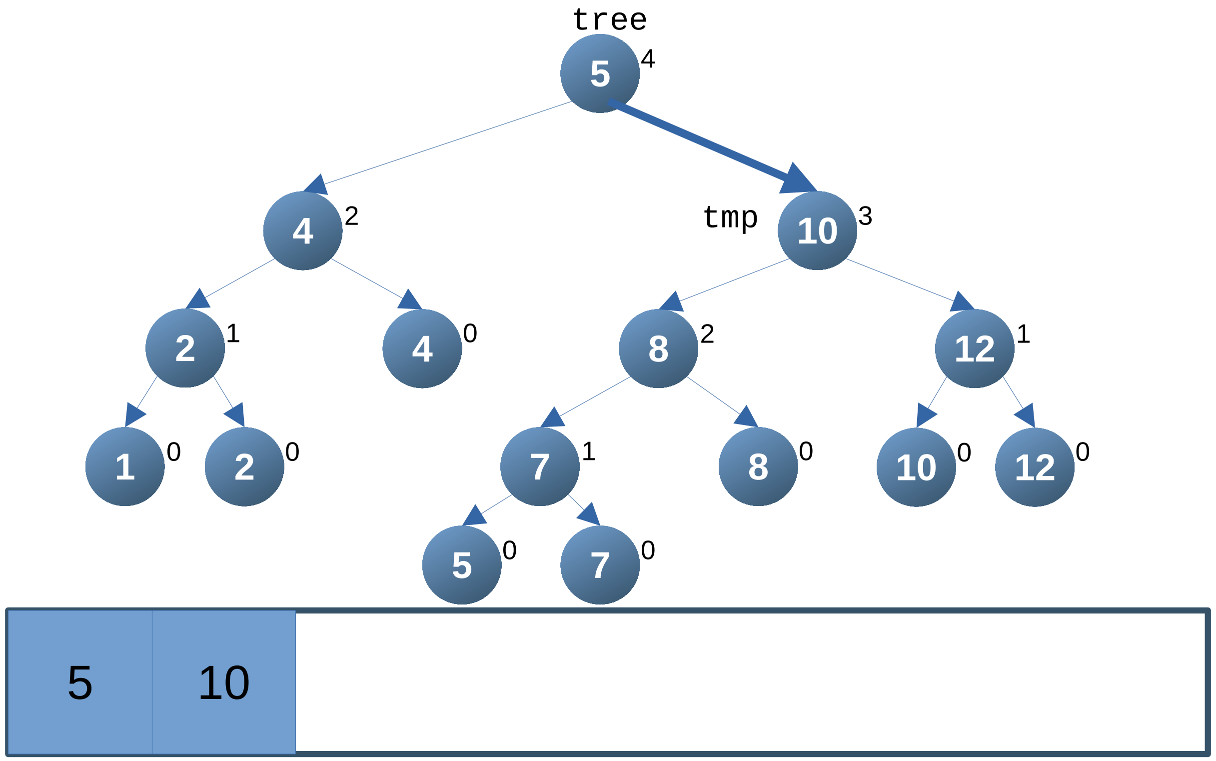



```
new_object = 6

tmp_node = 5
finished = NULL
```

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                     tmp_node = tmp_node->left;
              else
                     tmp_node = tmp_node->right;
         }
```

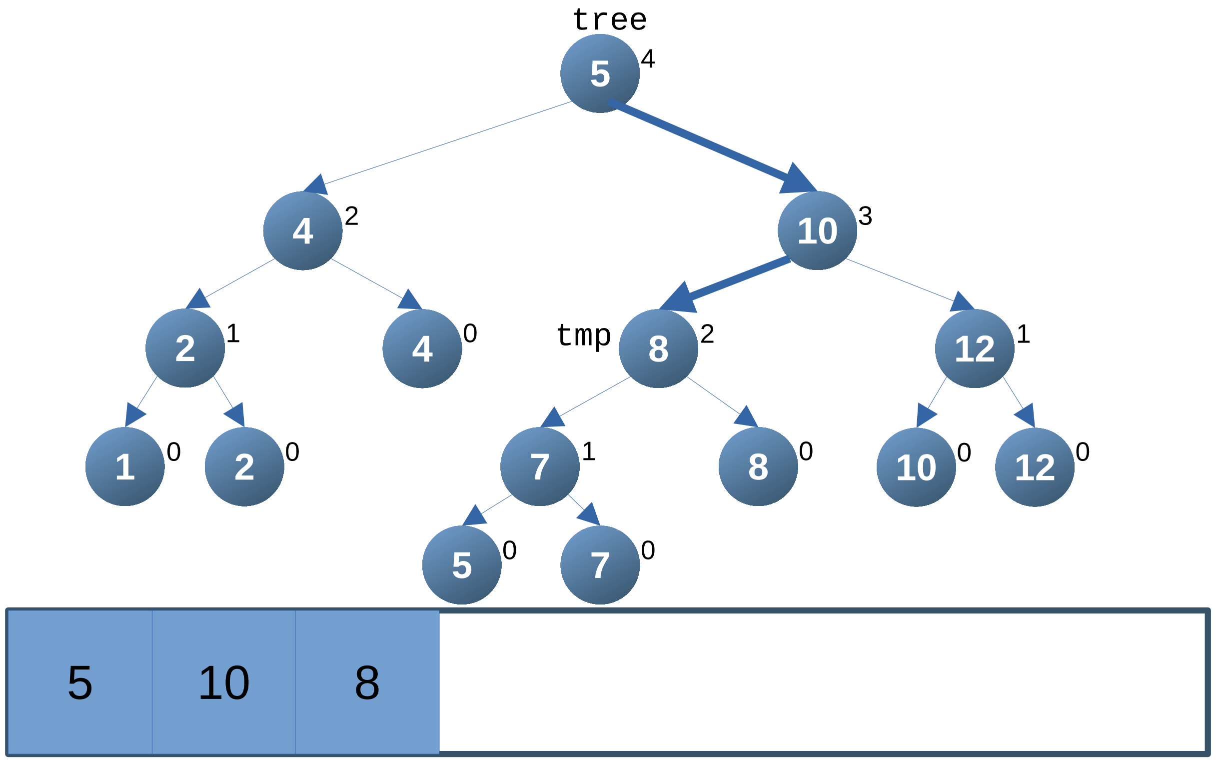



```
new_object = 6

tmp_node = 10
finished = NULL
```

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                    tmp_node = tmp_node->left;
              else
                    tmp_node = tmp_node->right;
         }
```



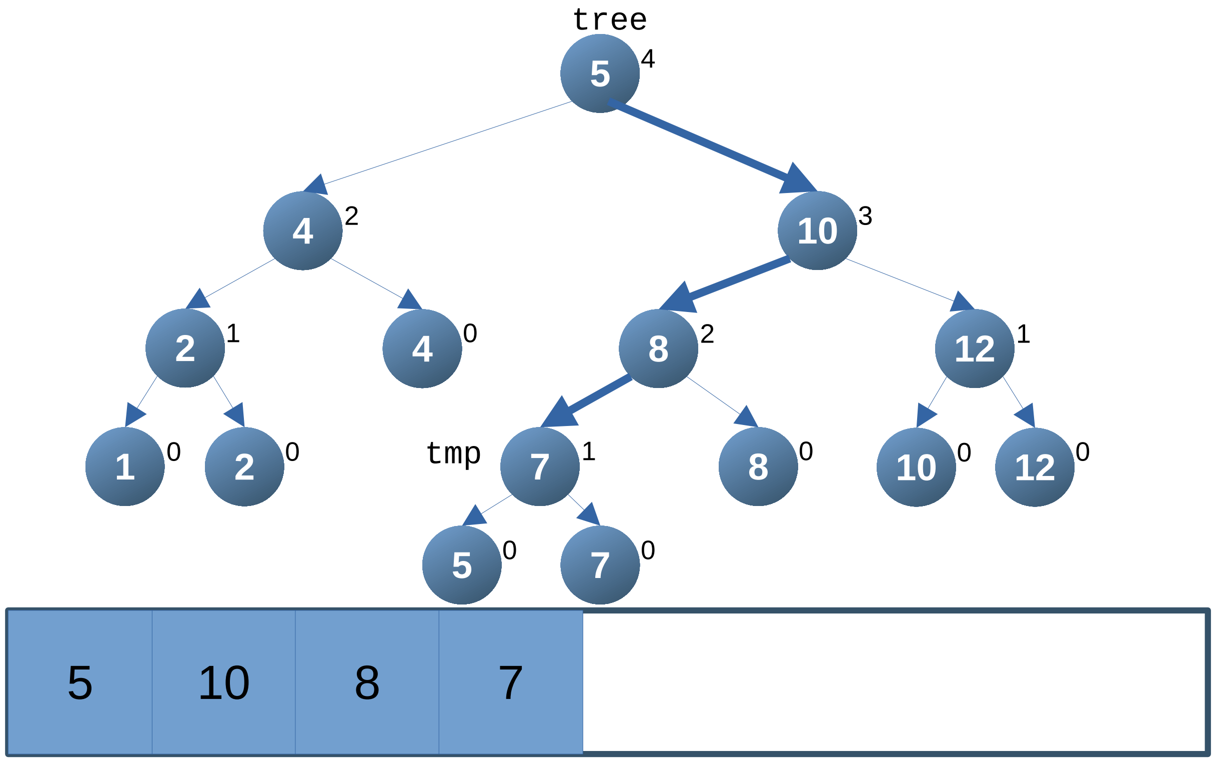

```
new_object = 6

tmp_node = 8
finished = NULL
```


Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                  tmp_node = tmp_node->left;
              else
                  tmp_node = tmp_node->right;
         }
```

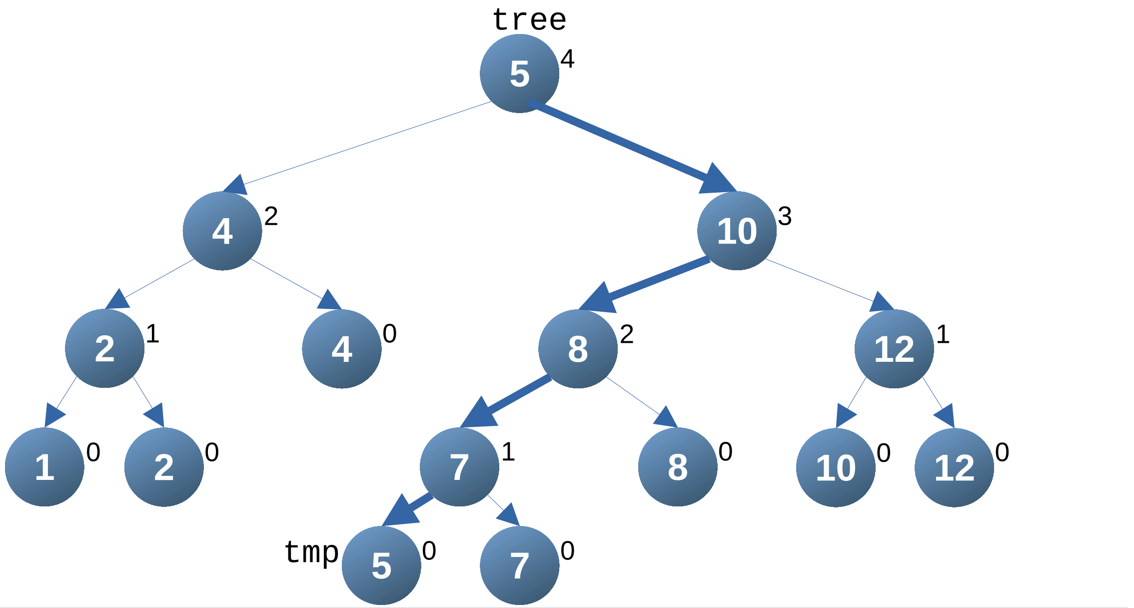



```
new_object = 6

tmp_node = 7
finished = NULL
```

Implementation of the Insert Method

```
int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{   tree_node_t *tmp_node;
    int finished;
    if( tree->left == NULL )
    {   tree->left = (tree_node_t *) new_object;
        tree->key  = new_key;
        tree->height = 0;
        tree->right  = NULL;
    }
     else
     {   create_stack();
         tmp_node = tree;
         while( tmp_node->right != NULL )
         {    push( tmp_node );
              if( new_key < tmp_node->key )
                    tmp_node = tmp_node->left;
              else
                    tmp_node = tmp_node->right;
         }
```



5	10	8	7	

```
new_object = 6

tmp_node = 5
finished = NULL
```

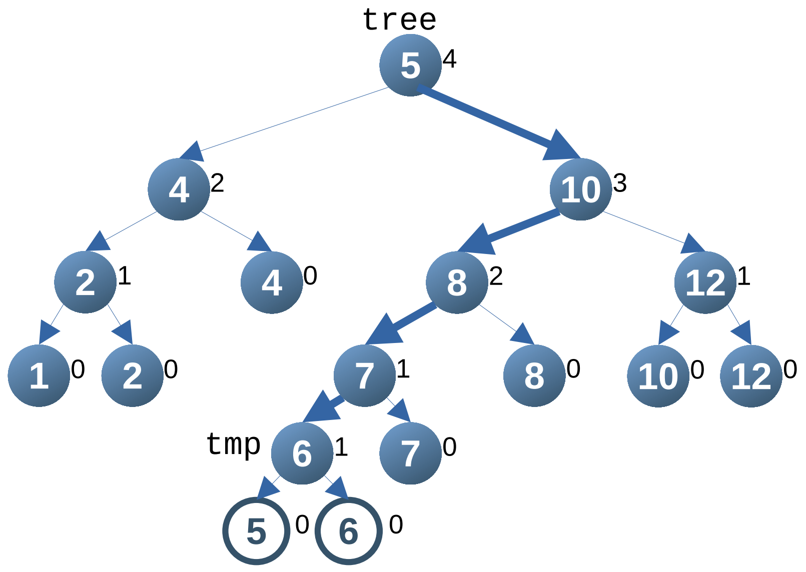
Implementation of the Insert Method

- Check if the key already exists, if it does, return error.
- Otherwise, insert the new leaf node to the tree.
- Setting the height of the new leaf and the previous node.

```
/* found the candidate leaf. Test whether
   key distinct */
if( tmp_node->key == new_key )
   return( -1 );
/* key is distinct, now perform
   the insert */
{  tree_node_t *old_leaf, *new_leaf;
   old_leaf = get_node();
   old_leaf->left = tmp_node->left;
   old_leaf->key = tmp_node->key;
   old_leaf->right  = NULL;
   old_leaf->height = 0;
   new_leaf = get_node();
   new_leaf->left = (tree_node_t *)
   new_object;
   new_leaf->key = new_key;
   new_leaf->right  = NULL;
   new_leaf->height = 0;
   if( tmp_node->key < new_key )
   {   tmp_node->left  = old_leaf;
       tmp_node->right = new_leaf;
       tmp_node->key = new_key;
   }
   else
   {   tmp_node->left  = new_leaf;
       tmp_node->right = old_leaf;
   }
   tmp_node->height = 1;
}
```

Implementation of the Insert Method

```
/* found the candidate leaf. Test whether
   key distinct */
if( tmp_node->key == new_key )
   return( -1 );
/* key is distinct, now perform
   the insert */
{  tree_node_t *old_leaf, *new_leaf;
   old_leaf = get_node();
   old_leaf->left = tmp_node->left;
   old_leaf->key = tmp_node->key;
   old_leaf->right  = NULL;
   old_leaf->height = 0;
   new_leaf = get_node();
   new_leaf->left = (tree_node_t *)
   new_object;
   new_leaf->key = new_key;
   new_leaf->right  = NULL;
   new_leaf->height = 0;
   if( tmp_node->key < new_key )
   {    tmp_node->left  = old_leaf;
        tmp_node->right = new_leaf;
        tmp_node->key = new_key;
   }
   else
   {    tmp_node->left  = new_leaf;
        tmp_node->right = old_leaf;
   }
   tmp_node->height = 1;
}
```



5	10	8	7	

```
new_object = 6

tmp_node = 6
finished = NULL
```

Implementation of the Insert Method

- Start the rebalancing of the tree using the path of previous nodes stored in a stack.
- Check, and if needed apply, the balancing cases of a height tree.
- (Case 2.1)
- (Case 2.2)

```
/* rebalance */
finished = 0;
while( !stack_empty() && !finished )
{   int tmp_height, old_height;
    tmp_node = pop();
    old_height= tmp_node->height;
    if( tmp_node->left->height -
        tmp_node->right->height == 2 )
    {   if( tmp_node->left->left->height -
            tmp_node->right->height == 1 )
        {   right_rotation( tmp_node );
            tmp_node->right->height =
            tmp_node->right->left->height + 1;
            tmp_node->height =
               tmp_node->right->height + 1;
        }
        else
        {   left_rotation( tmp_node->left );
            right_rotation( tmp_node );
            tmp_height =
               tmp_node->left->left->height;
            tmp_node->left->height  =
               tmp_height + 1;
            tmp_node->right->height =
               tmp_height + 1;
            tmp_node->height = tmp_height + 2;
        }
    }
```

Implementation of the Insert Method

- (Case 2.3)
- (Case 2.4)

```
else if( tmp_node->left->height -
         tmp_node->right->height == -2 )
{  if( tmp_node->right->right->height -
        tmp_node->left->height == 1 )
   {  left_rotation( tmp_node );
      tmp_node->left->height =
      tmp_node->left->right->height + 1;
      tmp_node->height =
         tmp_node->left->height + 1;
   }
   else
   {  right_rotation( tmp_node->right );
      left_rotation( tmp_node );
      tmp_height =
         tmp_node->right->right->height;
      tmp_node->left->height  =
         tmp_height + 1;
      tmp_node->right->height =
         tmp_height + 1;
      tmp_node->height = tmp_height + 2;
   }
}
```

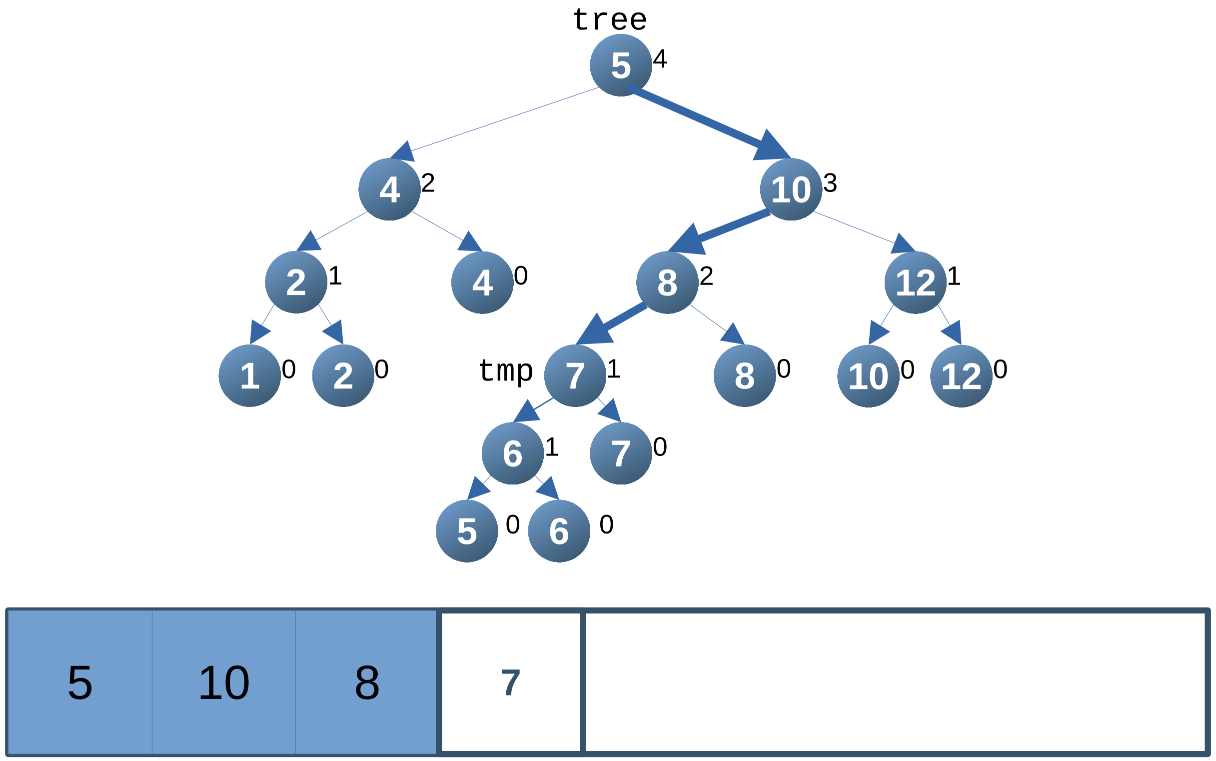
Implementation of the Insert Method

- (Trivial case) In this case we still recompute the height of the node.
- Ensure that the loop will stop after the balancing and not only after emptying the stack.
- Remove stack from memory and return.

```
            else /* update height even if there
                    was no rotation */
            {   if( tmp_node->left->height >
                    tmp_node->right->height )
                   tmp_node->height =
                   tmp_node->left->height + 1;
                else
                   tmp_node->height =
                   tmp_node->right->height + 1;
            }
            if( tmp_node->height == old_height )
                finished = 1;
         }
         remove_stack();
      }
      return( 0 );
   }
```

Implementation of the Insert Method

```
/* rebalance */
finished = 0;
while( !stack_empty() && !finished )
{  int tmp_height, old_height;
   tmp_node = pop();
   old_height= tmp_node->height;
   if( tmp_node->left->height -
       tmp_node->right->height == 2 )
   {  if( tmp_node->left->left->height -
          tmp_node->right->height == 1 )
      {  right_rotation( tmp_node );
         tmp_node->right->height =
         tmp_node->right->left->height + 1;
         tmp_node->height =
            tmp_node->right->height + 1;
      }
      else
      {  left_rotation( tmp_node->left );
         right_rotation( tmp_node );
         tmp_height =
            tmp_node->left->left->height;
         tmp_node->left->height  =
            tmp_height + 1;
         tmp_node->right->height =
            tmp_height + 1;
         tmp_node->height = tmp_height + 2;
      }
   }
```



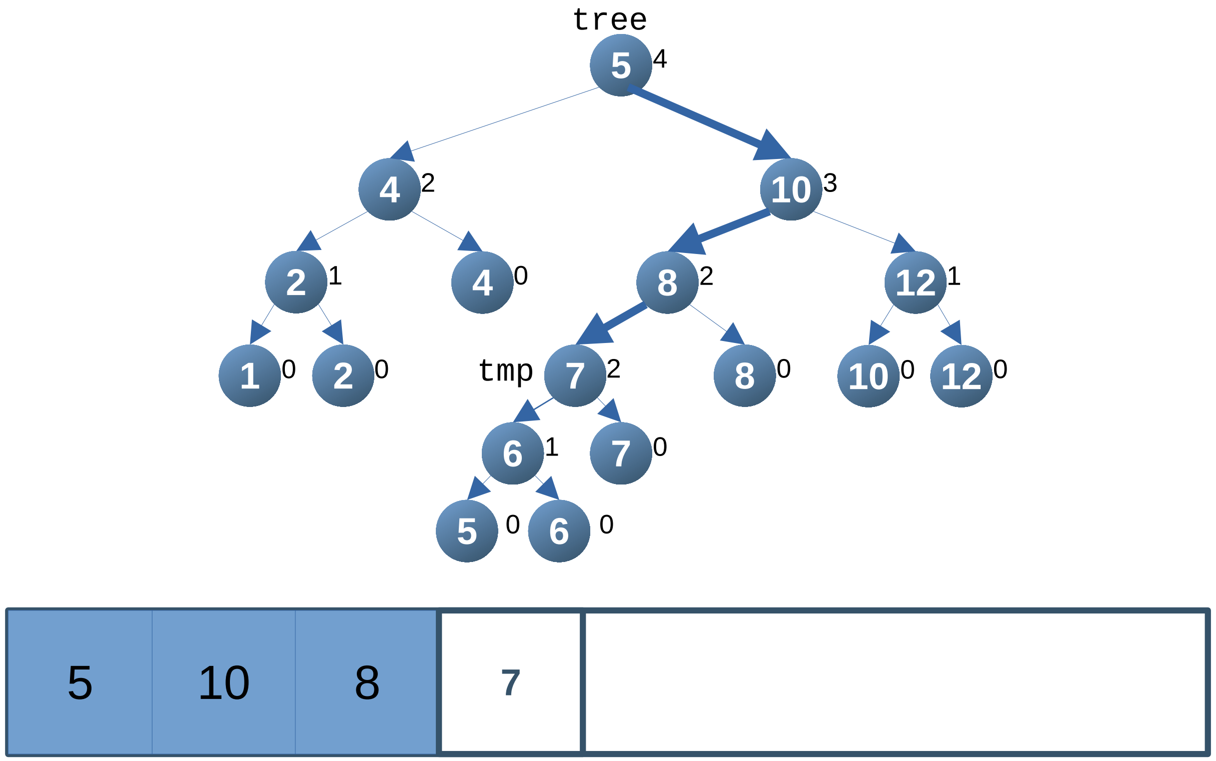

```
new_object = 6

tmp_node = 7
finished = 0

old_height = 1
```

Implementation of the Insert Method

```
        else /* update height even if there
                was no rotation */
        {   if( tmp_node->left->height >
                tmp_node->right->height )
                tmp_node->height =
                tmp_node->left->height + 1;
            else
                tmp_node->height =
                tmp_node->right->height + 1;
        }
        if( tmp_node->height == old_height )
            finished = 1;
      }
      remove_stack();
    }
    return( 0 );
}
```



new_object = 6

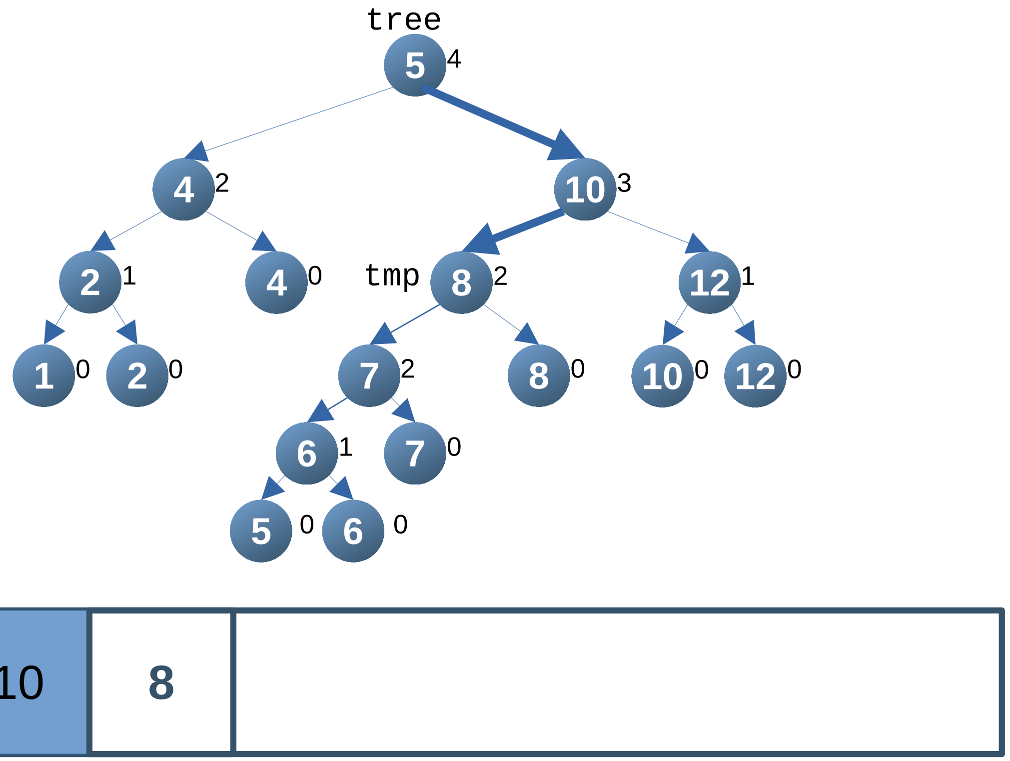
tmp_node = 7
finished = 0

old_height = 1

Presentation made by Hernández & Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Implementation of the Insert Method

```
/* rebalance */
finished = 0;
while( !stack_empty() && !finished )
{  int tmp_height, old_height;
   tmp_node = pop();
   old_height= tmp_node->height;
   if( tmp_node->left->height -
       tmp_node->right->height == 2 )
   {  if( tmp_node->left->left->height -
          tmp_node->right->height == 1 )
      {  right_rotation( tmp_node );
         tmp_node->right->height =
         tmp_node->right->left->height + 1;
         tmp_node->height =
            tmp_node->right->height + 1;
      }
      else
      {  left_rotation( tmp_node->left );
         right_rotation( tmp_node );
         tmp_height =
            tmp_node->left->left->height;
         tmp_node->left->height  =
            tmp_height + 1;
         tmp_node->right->height =
            tmp_height + 1;
         tmp_node->height = tmp_height + 2;
      }
   }
```



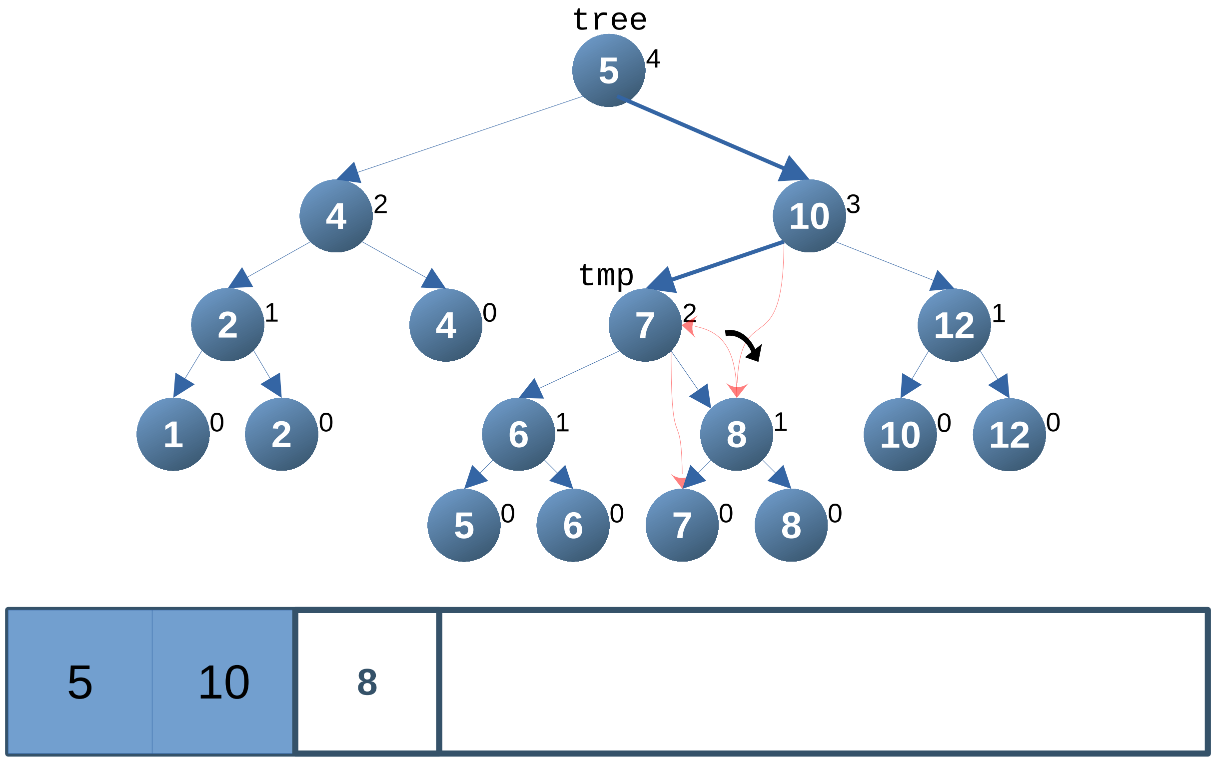

```
new_object = 6

tmp_node = 8
finished = 0

old_height = 2
```

Implementation of the Insert Method

```
/* rebalance */
finished = 0;
while( !stack_empty() && !finished )
{   int tmp_height, old_height;
    tmp_node = pop();
    old_height= tmp_node->height;
    if( tmp_node->left->height -
        tmp_node->right->height == 2 )
    {   if( tmp_node->left->left->height -
            tmp_node->right->height == 1 )
        {   right_rotation( tmp_node );
            tmp_node->right->height =
            tmp_node->right->left->height + 1;
            tmp_node->height =
               tmp_node->right->height + 1;
        }
        else
        {   left_rotation( tmp_node->left );
            right_rotation( tmp_node );
            tmp_height =
               tmp_node->left->left->height;
            tmp_node->left->height  =
               tmp_height + 1;
            tmp_node->right->height =
               tmp_height + 1;
            tmp_node->height = tmp_height + 2;
        }
    }
```



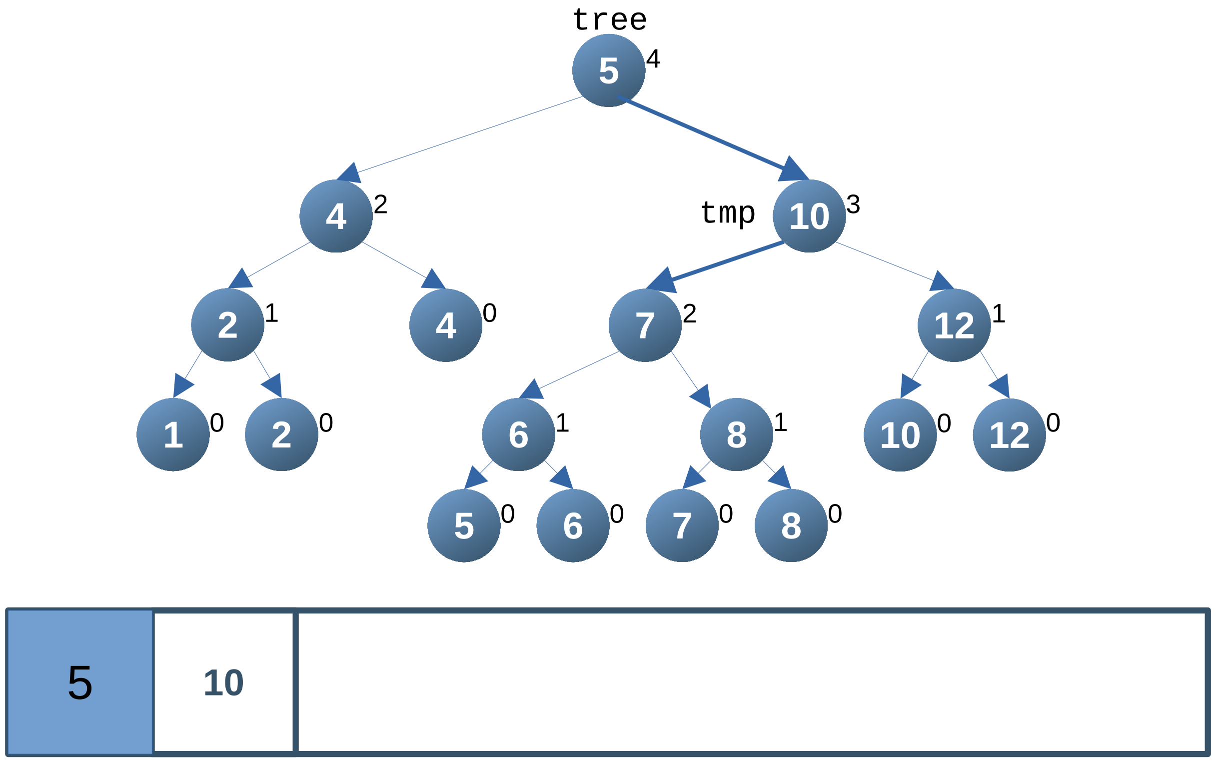

```
new_object = 6

tmp_node = 8
finished = 0

old_height = 2
```

Implementation of the Insert Method

```
    /* rebalance */
    finished = 0;
    while( !stack_empty() && !finished )
    {  int tmp_height, old_height;
       tmp_node = pop();
       old_height= tmp_node->height;
       ⋮
       else /* update height even if there
               was no rotation */
       {  if( tmp_node->left->height >
              tmp_node->right->height )
             tmp_node->height =
             tmp_node->left->height + 1;
          else
             tmp_node->height =
             tmp_node->right->height + 1;
       }
       if( tmp_node->height == old_height )
          finished = 1;
    }
    remove_stack();
  }
  return( 0 );
}
```



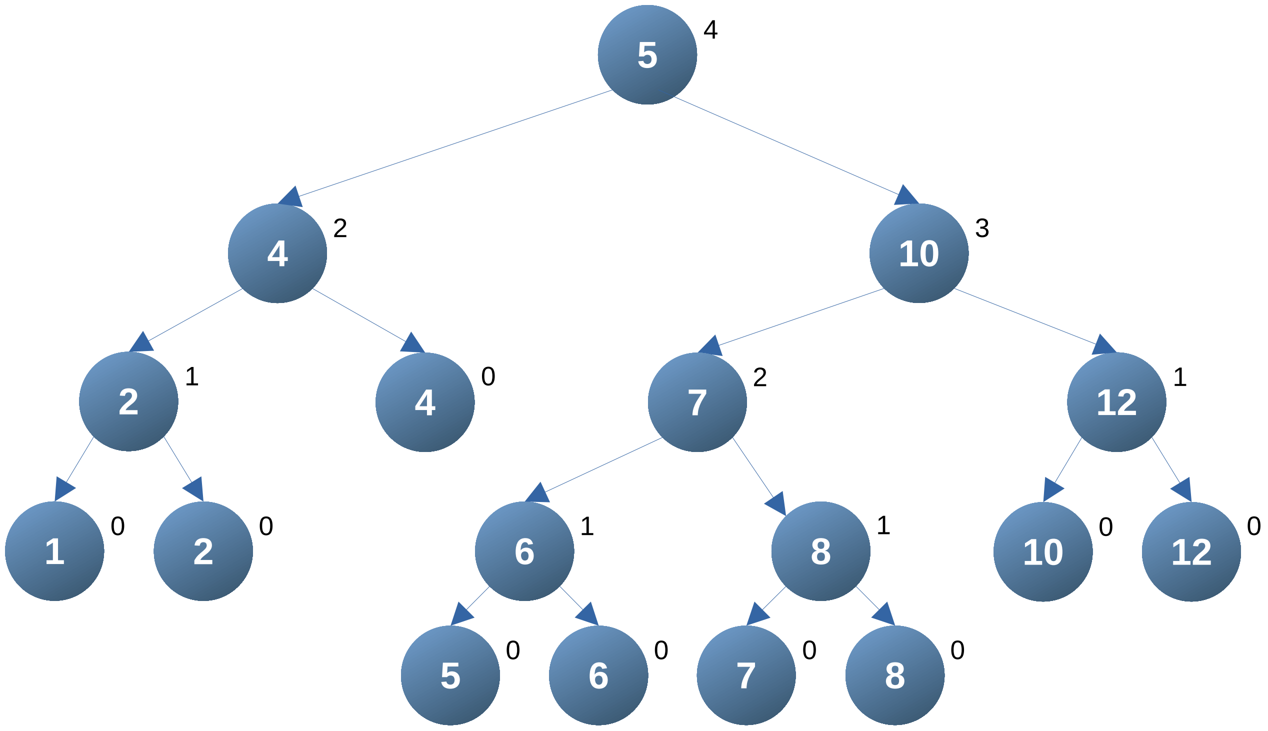
new_object = 6

tmp_node = 10
finished = 1

old_height = 3

Implementation of the Insert Method

```
/* rebalance */
finished = 0;
while( !stack_empty() && !finished )
{   int tmp_height, old_height;
    tmp_node = pop();
    old_height= tmp_node->height;
                    ⋮
    else /* update height even if there
              was no rotation */
    {   if( tmp_node->left->height >
              tmp_node->right->height )
              tmp_node->height =
              tmp_node->left->height + 1;
        else
              tmp_node->height =
              tmp_node->right->height + 1;
    }
    if( tmp_node->height == old_height )
        finished = 1;
}
remove_stack();
  }
  return( 0 );
}
```



Other Implementations

- In the end, height-balanced trees are a good type of balanced tree by it’s relative simplicity.
- Although a little slower than a good search tree, it’s structure makes them better than the average search tree.
- Other implementation that use the height-balanced trees are often slower and use more memory.
- These other implementations change mainly on the restriction of difference between the heights of the subtrees of a node.

BIBLIOGRAPHY

- Brass, P. (2008). Advanced Data Structures. Cambridge University Press.