《数据结构与算法》课程组

重庆大学计算机学院

# Data Structures & Algorithms

10

# BINARY SEARCHING TREES

# Outline

10.1 Binary Search Trees (BST)

10.2 Balanced trees

- **AVL tree**

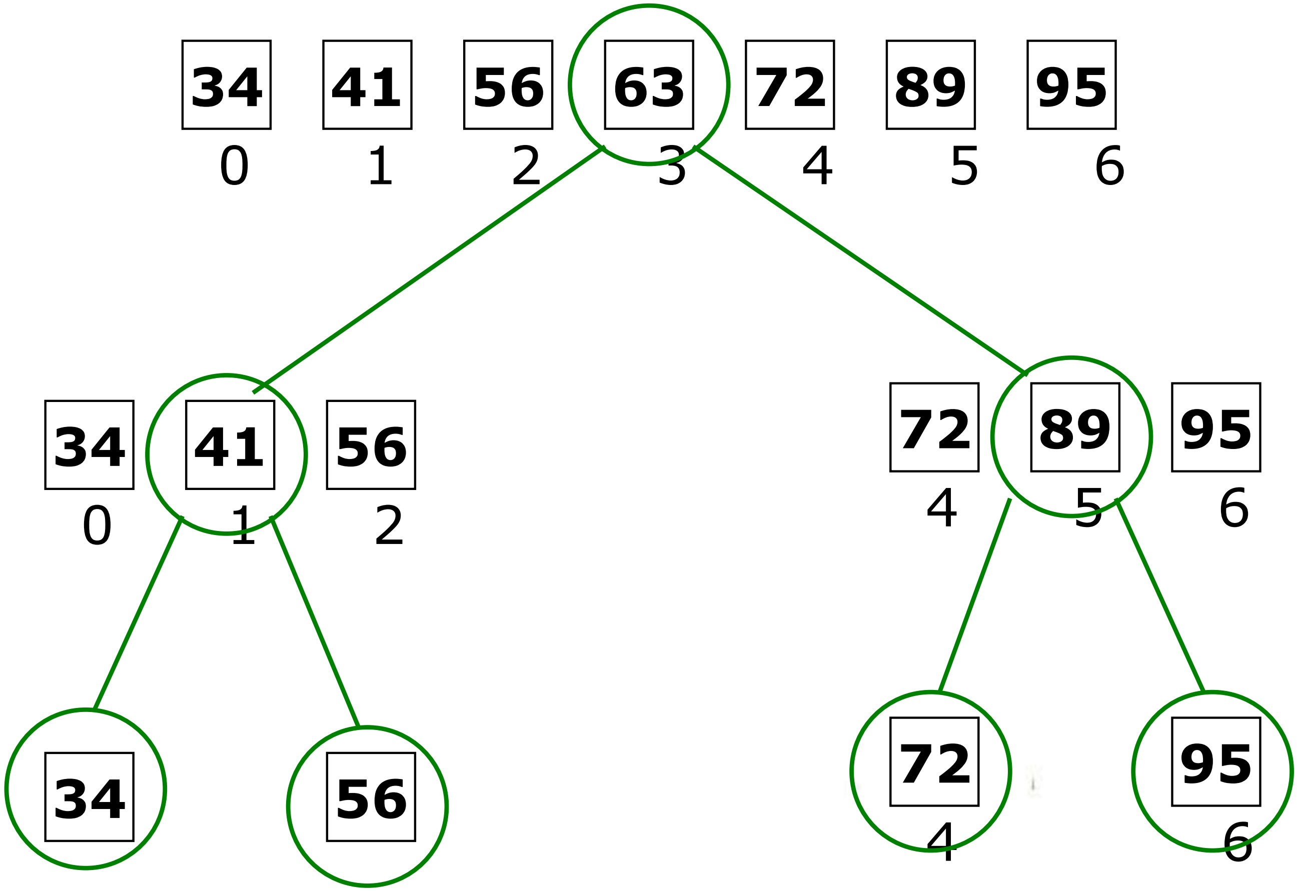
# 10.1 Binary Searching Tree

# A Taxonomy of Trees

- **General Trees – any number of children / node**
- **Binary Trees – max 2 children / node**
  - **Heaps – parent < (>) children**
  - **Binary Search Trees**

# Binary Search Algorithm

Binary Search algorithm of an array of *sorted* items reduces the search space by one half after each comparison



# BST： Motivation

- Binary search For sorted array search
  - search： $\Theta(\log n)$ fast
  - insertion ： $\Theta(n)$ on average， slow

    --once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.
- Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly?

# Binary Search Trees

- Binary search tree (BST)
  - Every element has a unique key (comparable)
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.
- if the BST nodes are printed using an inorder traversal, the resulting enumeration will be in sorted order from lowest to highest.

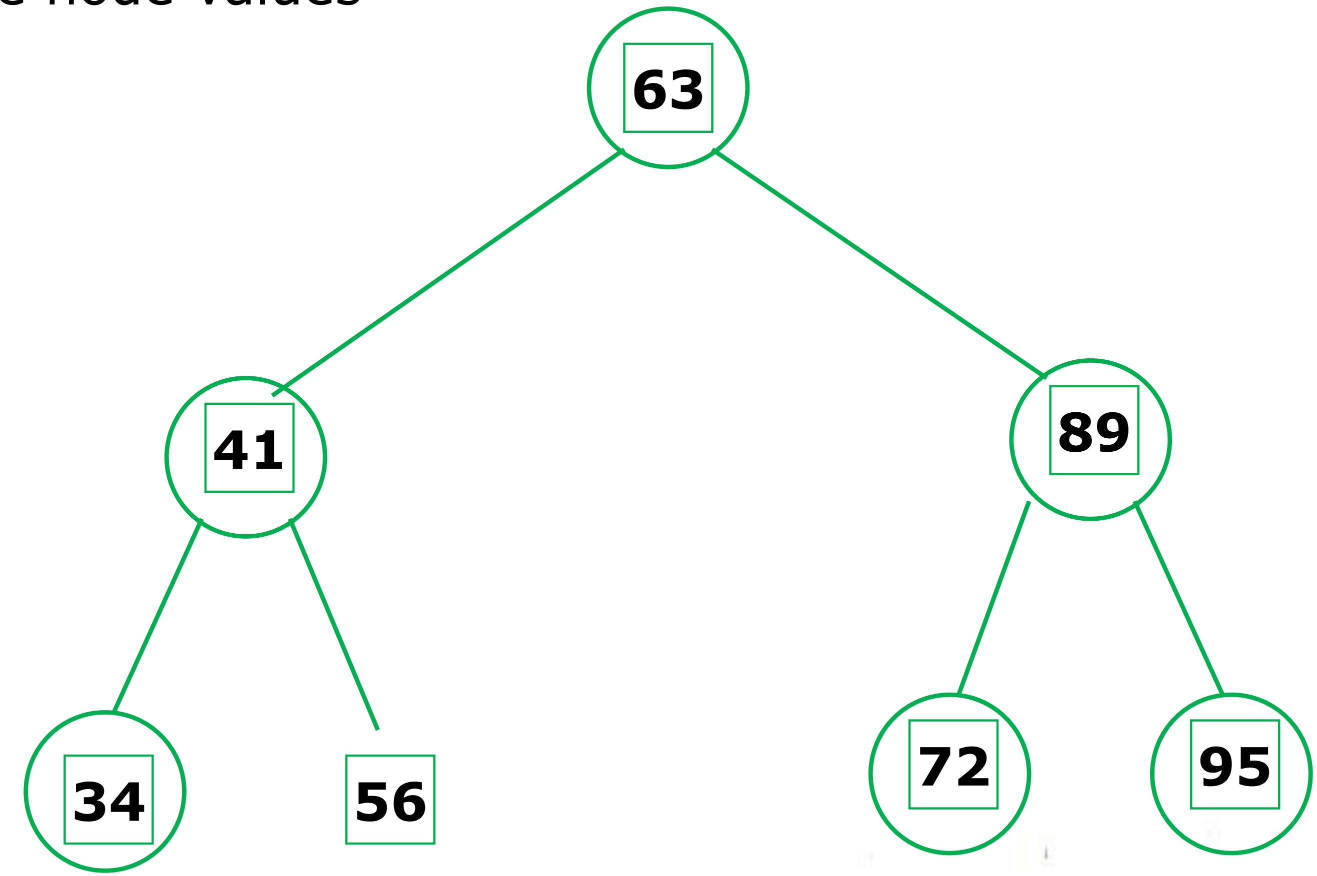
# Binary Search Trees

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.

- This data organization leads to $\Theta(\log(n))$ complexity for searches, insertions and deletions in certain types of the BST (**balanced trees**).
  - O(h) in general

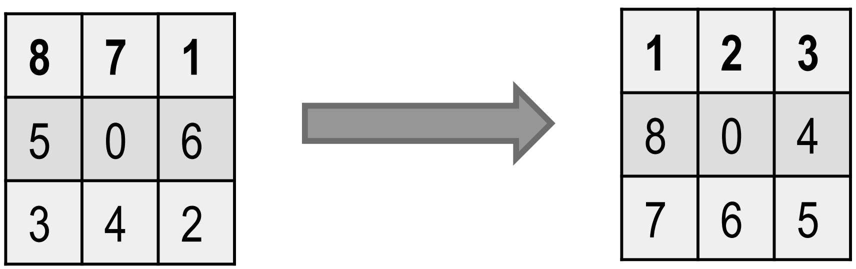
# Organization Rule for BST

- the values in all nodes in the left subtree of a node are less than the node value
- the values in all nodes in the right subtree of a node are greater than the node values



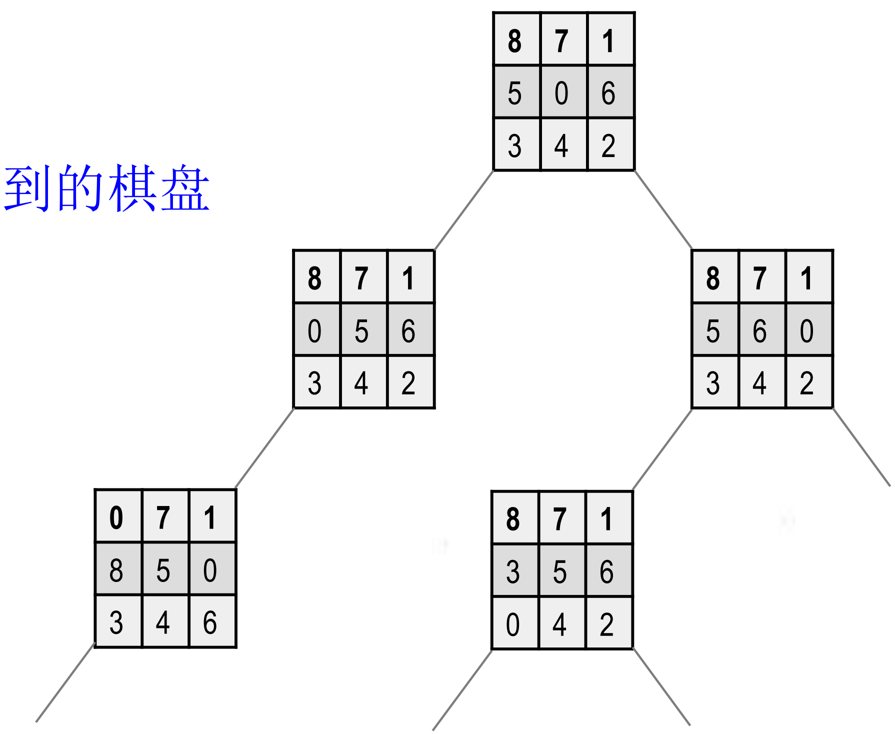
# Application of BST
# (八数码问题，POJ）

在3×3的棋盘上，摆有九个棋子，每个棋子上标有0至8的某一数字。每次只能将0与相邻数值交换。

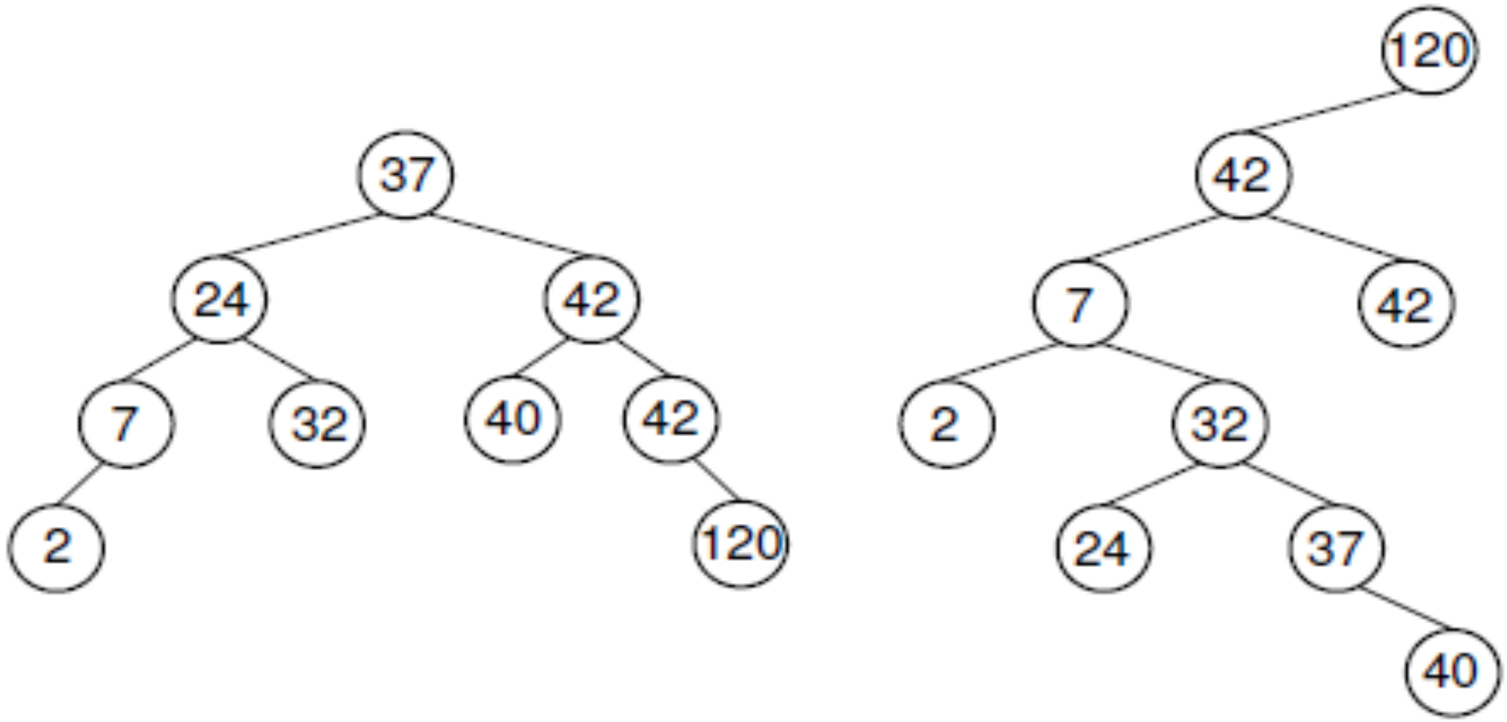



- 搜索棋盘状态空间：DFS，BFS，A*
- 状态数量：9！ = 362880
- 状态的动态存储与快速查重

用BST存储遍历到的棋盘状态并判重！



# BST Example




(a) (b)

**The shape of a BST depends on the order in which elements are inserted.**

# BST：Implementation

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;   // Root of the BST
  int nodecount;         // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                             const Key&, const E&);
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
  E findhelp(BSTNode<Key, E>*, const Key&) const;
  void printhelp(BSTNode<Key, E>*, int) const;

public:
  BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()   // Reinitialize tree
    { clearhelp(root); root = NULL; nodecount = 0; }
```

# BST Operations: Search

**Searching in the BST**

`method search(key)`

• implements the binary search based on comparison of the items in the tree

• the items in the BST must be comparable (e.g integers, string, etc.)

The search starts at the root. It probes down, comparing the values in each node with the target, till it finds the first item equal to the target. Returns this item or `null` if there is none.

# Search in BST - Pseudocode

if the tree is empty
        return NULL

else if  the item in the node equals the target
        return the node value

else if  the item in the node is greater than the target return the result of searching the left subtree

else if  the item in the node is smaller than the target return the result of searching the right subtree

# **Search in BST – implementation** (recursive)

```
// Return Record with key value k, NULL if none exist.
// k: The key value to find. */
// Return some record matching "k".
// Return true if such exists, false otherwise. If
// multiple records match "k", return an arbitrary one.
E find(const Key& k) const { return findhelp(root, k); }

// Re
int s

void
  if
  els
}
};
```

```
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                              const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);   // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

# Search in BST – implementation (non-recursive)

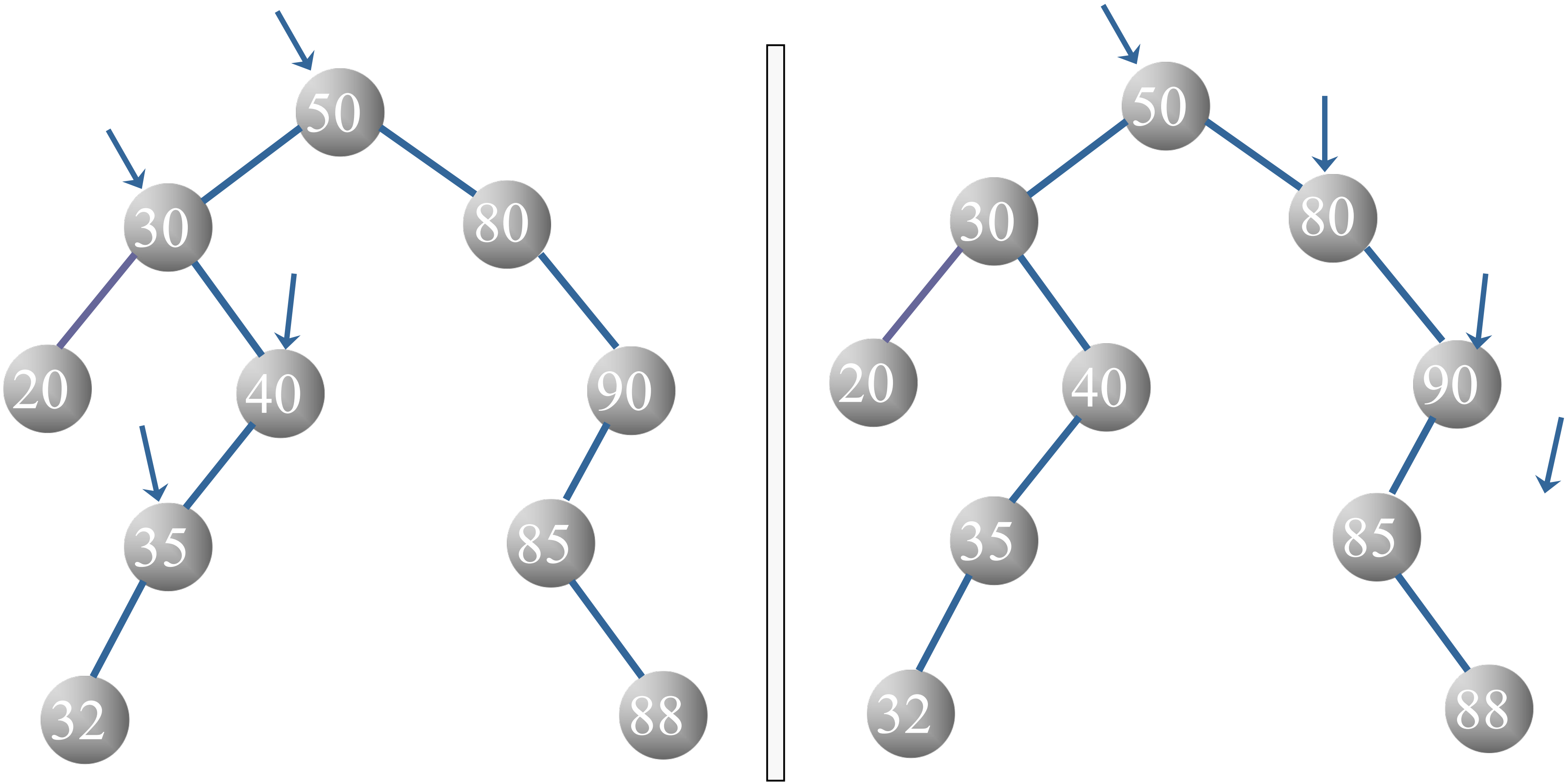
```
template <typename Key, typename E>
E BST<Key, E>::findhelp( BSTNode<Key, E>* root, const Key& k) const
{
    if(root == NULL)  return NULL;

    while(root && root->key() != k){     //二分查找
            if ( k  <  root->key() )
                root = root->left();
            else if( k > root->key() )
                root = root->right();
    }

    if(root)
          return root->element();
    else
          return NULL;
}
```

# Search in BST - Example

Search for 35 , 95



# BST Operations: Insertion

`method insert(key)`

- places a new item near the frontier of the BST while retaining its organization of data:
  - **starting at the root** it probes **down** the tree till it finds a node whose left or right pointer is empty that is a logical place for the new value
  - **using a binary search** to locate the insertion point is based on comparisons of the new item and values of nodes in the BST
    - *Elements in nodes must be comparable!*

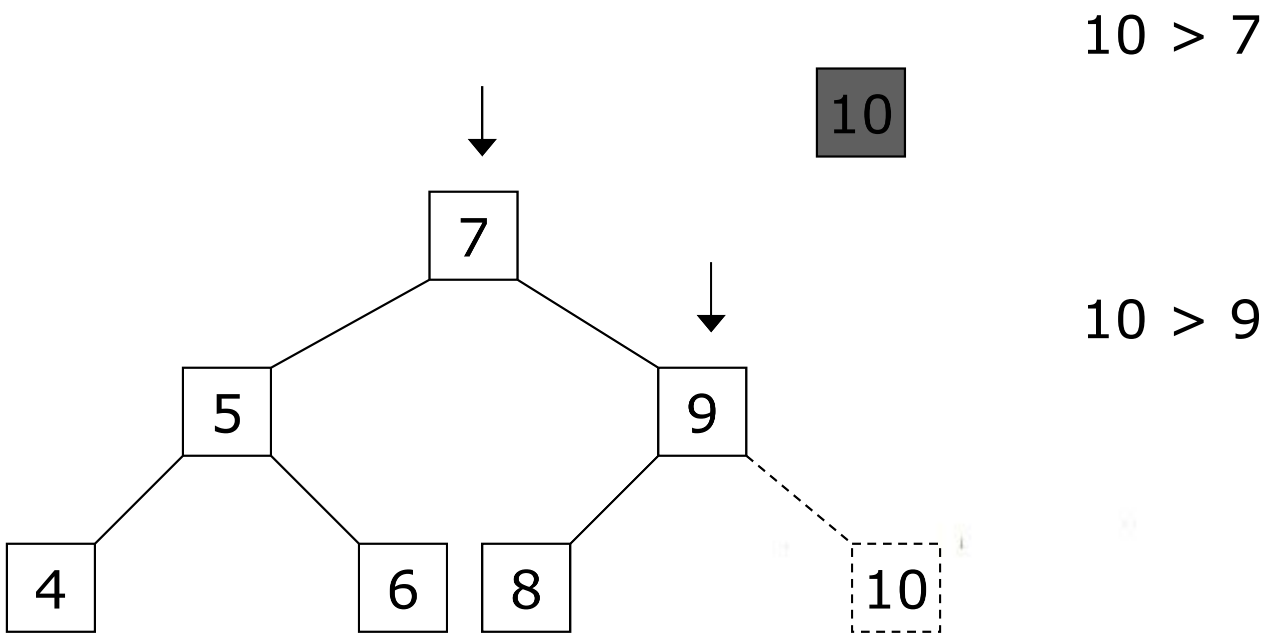
# Insertion in BST - Example

Case 1: The Tree is Empty

- Set the root to a new node containing the item

Case 2: The Tree is Not Empty

- Call a recursive helper method to insert the item



# Insertion in BST - Pseudocode

```
if  tree is empty
    create a root node with the new key
else
    compare key with the top node
    if key =  node key
        replace the node with the new value
    else if  key >  node key
        compare key with the right subtree:
          if  subtree is empty create a leaf node
          else add key  in right subtree
     else  key <  node key
        compare key with the left subtree:
          if the subtree is empty create a leaf node
          else add key to the left subtree
```

# BST: Insertion (recursive)

```
// Insert a record into the tree.
// k Key value of the record.
// e The record to insert.
void insert(const Key& k, const E& e) {
  root = inserthelp(root, k, e);
  nodecount++;
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted
}
```

# BST: Insertion (non-recursive)

```
template <typename Key, typename E>
BSTNode<Key, E>*  BST<Key, E>::
inserthelp( BSTNode<Key, E>* root, const Key& k, const E& it)
{
   if(root == NULL)  return new BSTNode<Key, E> (k,it,NULL,NULL);
   BSTNode<Key E> *father, *node = root;

   while(node && node->key() != k){     //二分查找插入节点的父节点
         father = node;
         node = ( k < node->key() ?  node->left() : node->right() );
   }

   if(node)
            node->setValue(it);
   else if ( k < father->key() )
            father->setLeft(new BSTNode<Key, E>(k,it,NULL,NULL));
   else
            father->setRight(new BSTNode<Key, E>(k,it,NULL,NULL));

   return root;
}
```

# BST Operations: Removal

- **removes** a specified item from the BST and **adjusts** the tree
- uses a binary search to locate the target item:
  - **starting at the root** it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)

- removal of a node must not leave a ‘gap’ in the tree,

# Removal in BST - Pseudocode

method remove (key)

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

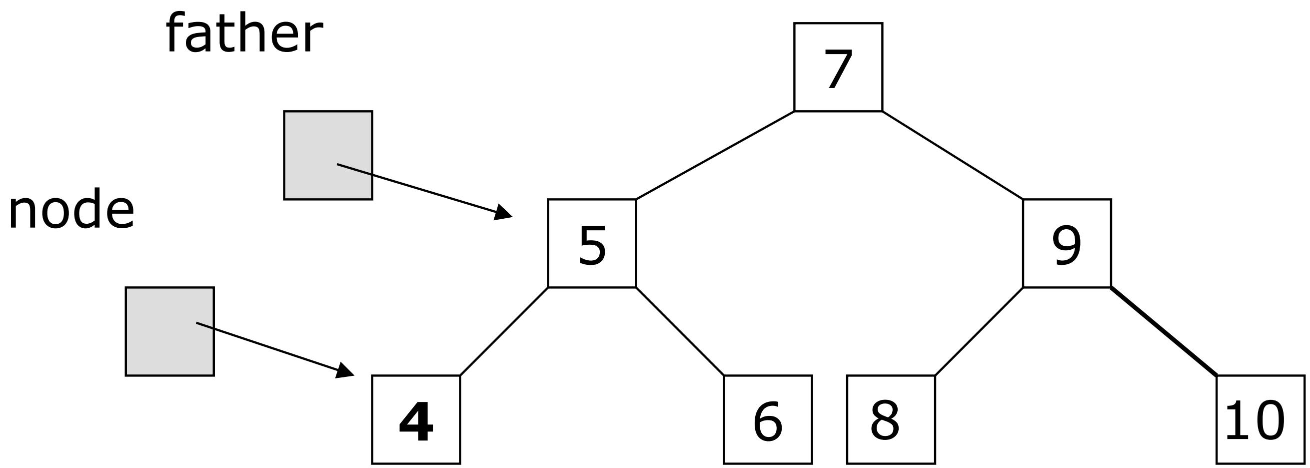
else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees

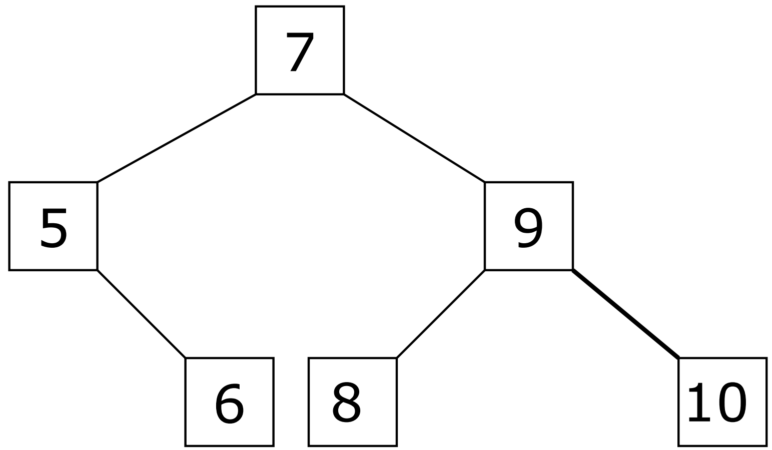
replace the link in the parent with null

# Removal in BST: Example

**Case 1:** removing a node with 2 EMPTY SUBTREES




***Removing 4***
replace the link in the parent with `null`



# Removal in BST - Pseudocode

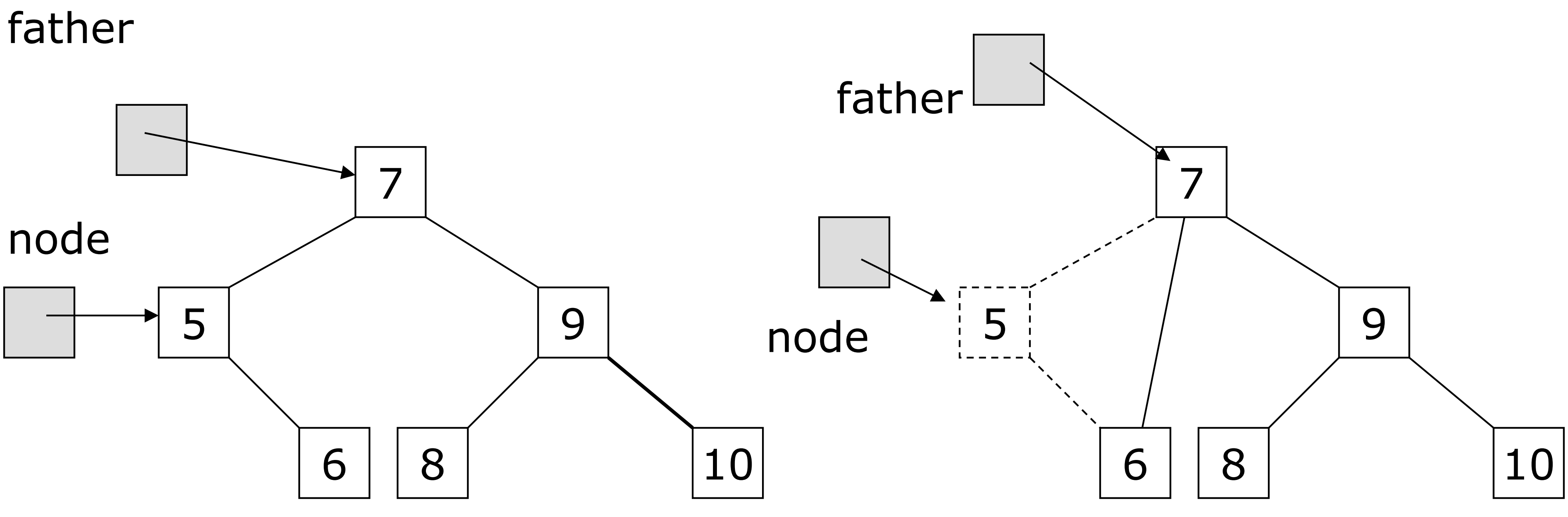
Case 2: if the node has no left child
- link the parent of the node
  to the right (non-empty) subtree

Case 3: if the node has no right child
- link the parent of the target
  to the left (non-empty) subtree

# Removal in BST: Example

**Case 2:** removing a node with 1 EMPTY SUBTREE

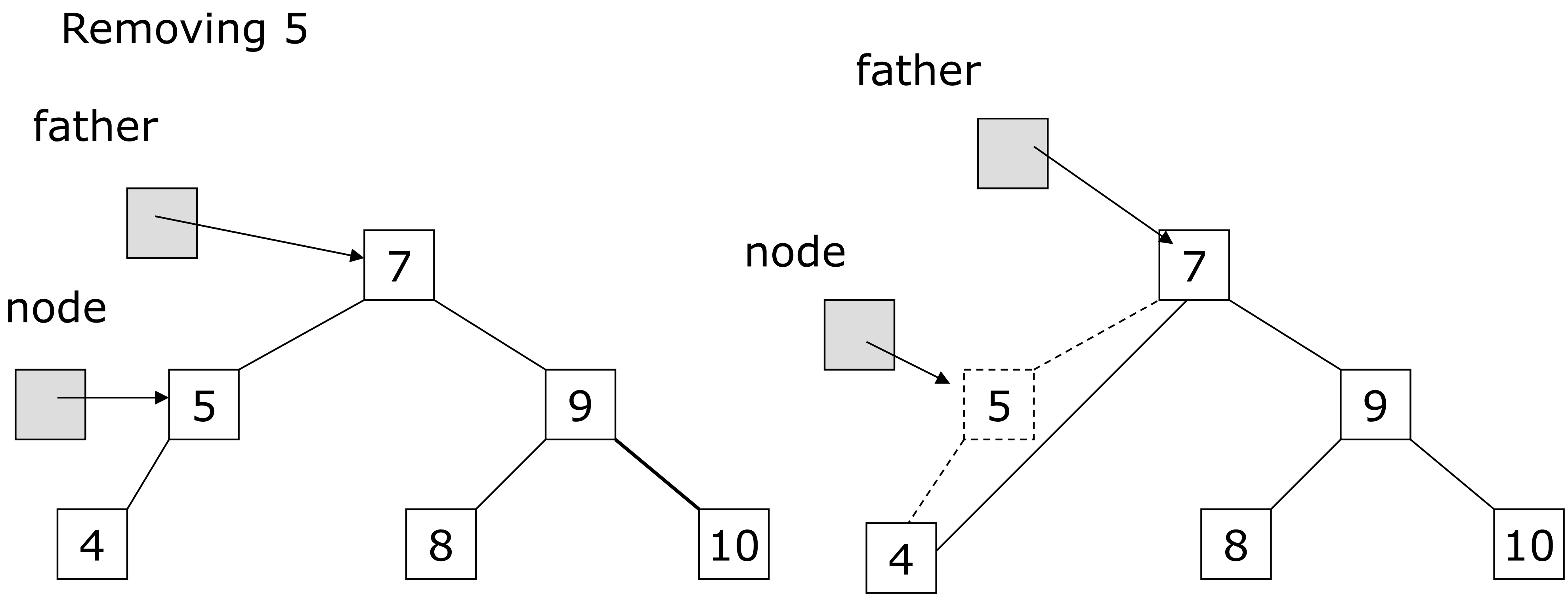
the node has no left child:
link the parent of the node to the right (non-empty) subtree



# Removal in BST: Example

**Case 3:** removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree



# Removal in BST - Pseudocode

Case 4: if the node has a left and a right subtree

(1) replace the node`s value with the min value in the right subtree

(2) delete the min node in the right subtree

# Removal in BST: Example

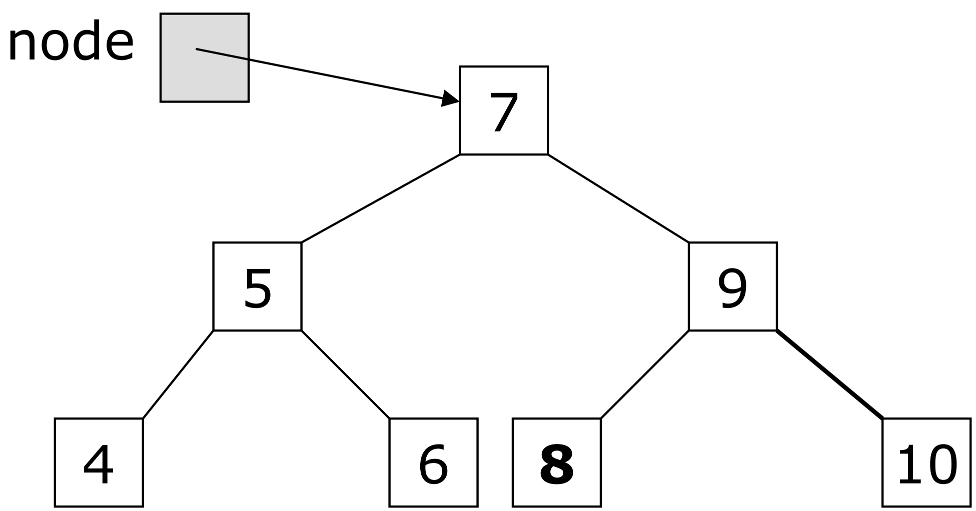
**Case 4:** removing a node with 2 SUBTREES

- replace the node's value with the min value in the right subtree
- delete the min node in the right subtree

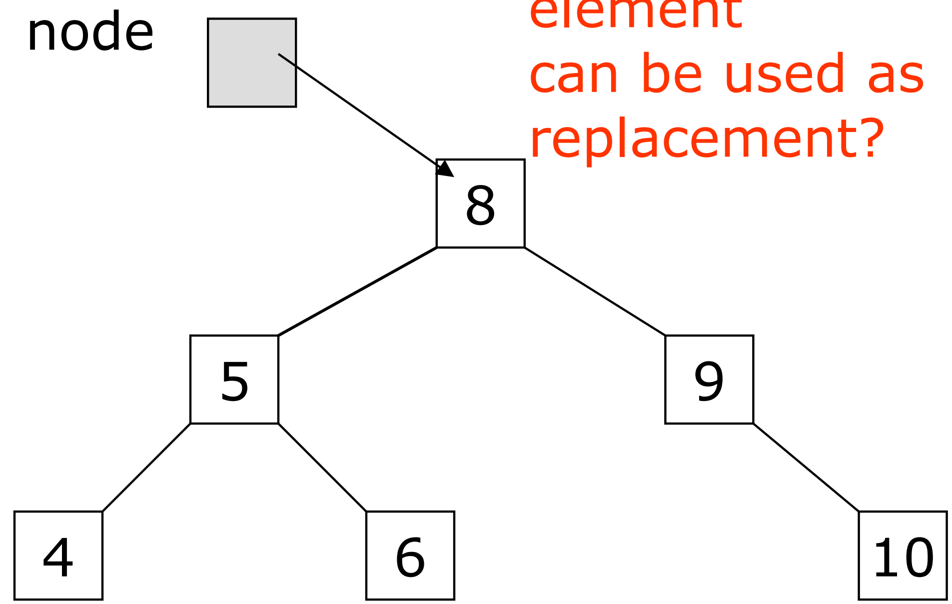
Q1: how to find the min node in the right subtree?

Q2: how many non-empty children a min node can have?

***Removing 7***




Q3: What other element
can be used as replacement?



# Removal in BST: implementation (recursive)

```
// Remove a node with key value k
// Return: The tree with the node removed
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
removehelp(BSTNode<Key, E>* rt, const Key& k) {
  if (rt == NULL) return NULL;    // k is not in tree
  else if (k < rt->key())
    rt->setLeft(removehelp(rt->left(), k));
  else if (k > rt->key())
    rt->setRight(removehelp(rt->right(), k));
  else {
    BSTNode<Key, E>* tem
    if (rt->left() == NU
      rt = rt->right();
      delete temp;
    }
    else if (rt->right()
      rt = rt->left();
      delete temp;
    }
    else {
      BSTNode<Key, E>* te
      rt->setElement(temp->element());
      rt->setKey(temp->key());
      rt->setRight(deletemin(rt->right()));
      delete temp;
    }
  }
  return rt;
}
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();
  else {                          // Continue left
    rt->setLeft(deletemin(rt->left()));
    return rt;
  }
}
```

```
template <typename Key, typename E>
BSTNode<Key, E>*  BST<Key, E>::removehelp( BSTNode<Key, E>* root, const Key& k)
{
    BSTNode<Key, E> *father, *node = root;
    while(node && node->key() != k){ //二分查找
        father = node;
        node = ( k < node->key() ?  node->left() : node->right() );
    }
    if(node == NULL) return root;

    if(node->left() && node->right()){ //左右子树不为空
        BSTNode<Key, E> tmp = father = node;
        node = node->right();
        while(node->left())  { //找右边子树中值最小节点
            father = node;
            node = node->left();
        }
        tmp->setElement(node->element());
    }
    if(node == root)            //删除节点最多只有一个非空子树
        root = (node->left()? node->left() : node->right());
    else if (node == father->left())
            father->setLeft((node->left()? node->left() : node->right()));
        else
            father->setRight((node->left()? node->left() : node->right()));
    delete node;  //删除
    return root;
}
```

# Analysis of BST Operations

- The complexity of operations **`get`**, **`insert`** and **`remove`** in BST is $\Theta(h)$ , where h is the height.
- $\Theta(\log(n))$ when the tree is balanced.
- The updating operations cause the tree to become unbalanced. So the tree can degenerate to a linear shape and the operations will become

# 二叉查找树常见面试题

1. 给定一个整数数组A[1..n]，按要求返回一个新数组 *counts[1..n]*。数组 *counts* 有该性质： counts[i] 的值是 A[i] 右侧小于 A[i] 的元素的数量。

示例:

输入: [5,2,6,1]

输出: [2,1,1,0]　hint： 从后往前

2.给定一个二叉查找树, 找到该树中两个指定节点的最近公共祖先。

3.给定一个二叉树，判断其是否是一个有效的二叉查找树。

4. 查找二叉查找树的第k小元素

# 10.2 AVL Tree

# Balanced Trees

- BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations.
- Solutions :

1. to adopt another search tree structure such as the 2-3 tree or the red-black tree.
2. to modify the BST access functions in some way to guarantee that the tree performs well.
   - requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update

- If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure, e.g., the AVL tree.

# The AVL tree

- The AVL tree (named for its inventors *Adelson-Velskii* and *Landis*) : a BST with the following additional property:
  - **For every node, the heights of its left and right subtrees differ by at most 1.**
- if a AVL tree contains n nodes, then it has a depth of at most $\Theta(\log(n))$. As a result, search for any node will cost $\Theta(\log(n))$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $\Theta(\log(n))$, even in the worst case.
- The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property.
  - **implement the revised update routines in $\Theta(\log(n))$ time.**

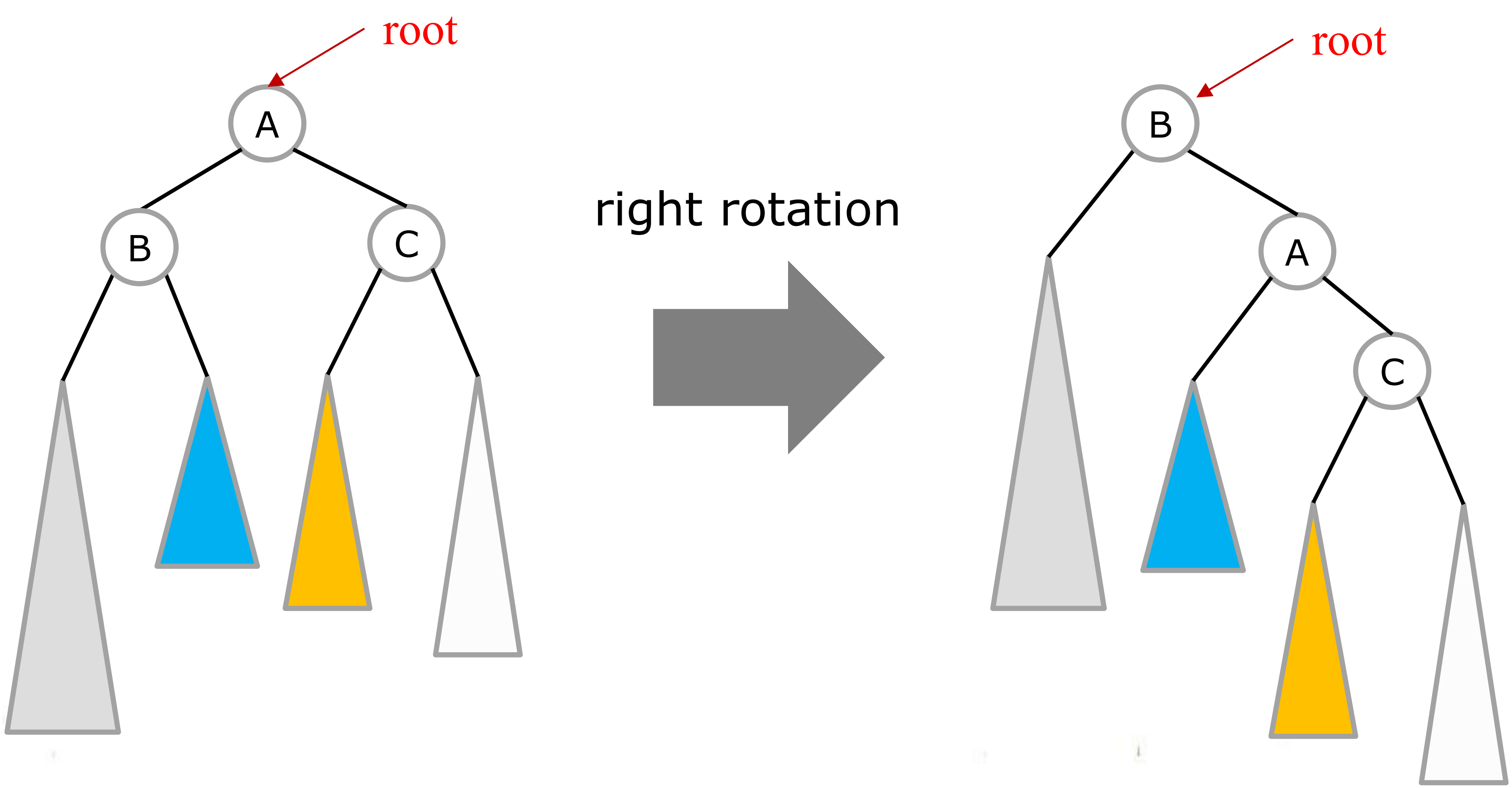
# AVL Tree Node

```
template <typename E>
class AVLNode{
   public:
        E element;
        int ht;    //以节点为根的子树高度
        AVLNode* left;
        AVLNode* right;

};
```

# How to balance the tree in O(log n) time?

- **using a series of local operations known as rotations**

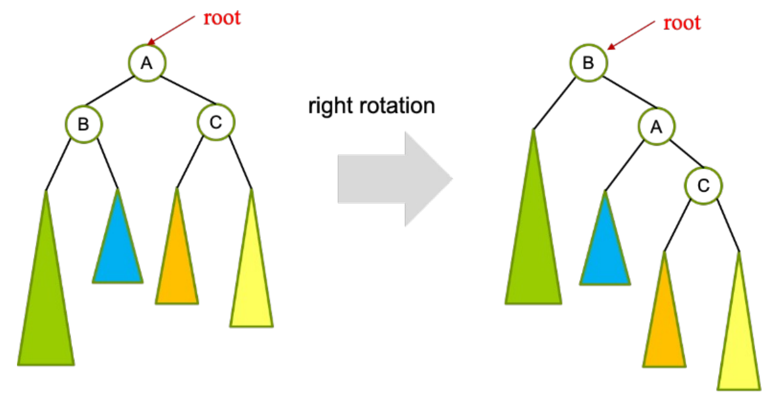


# How to balance the tree in O(log n) time?

- using a series of local operations known as **rotations**

**height(root)**

**if** root = NIL
  **then** **return** 0
**else**
  **return** root $\rightarrow$ *ht*




**getHeight(root)**

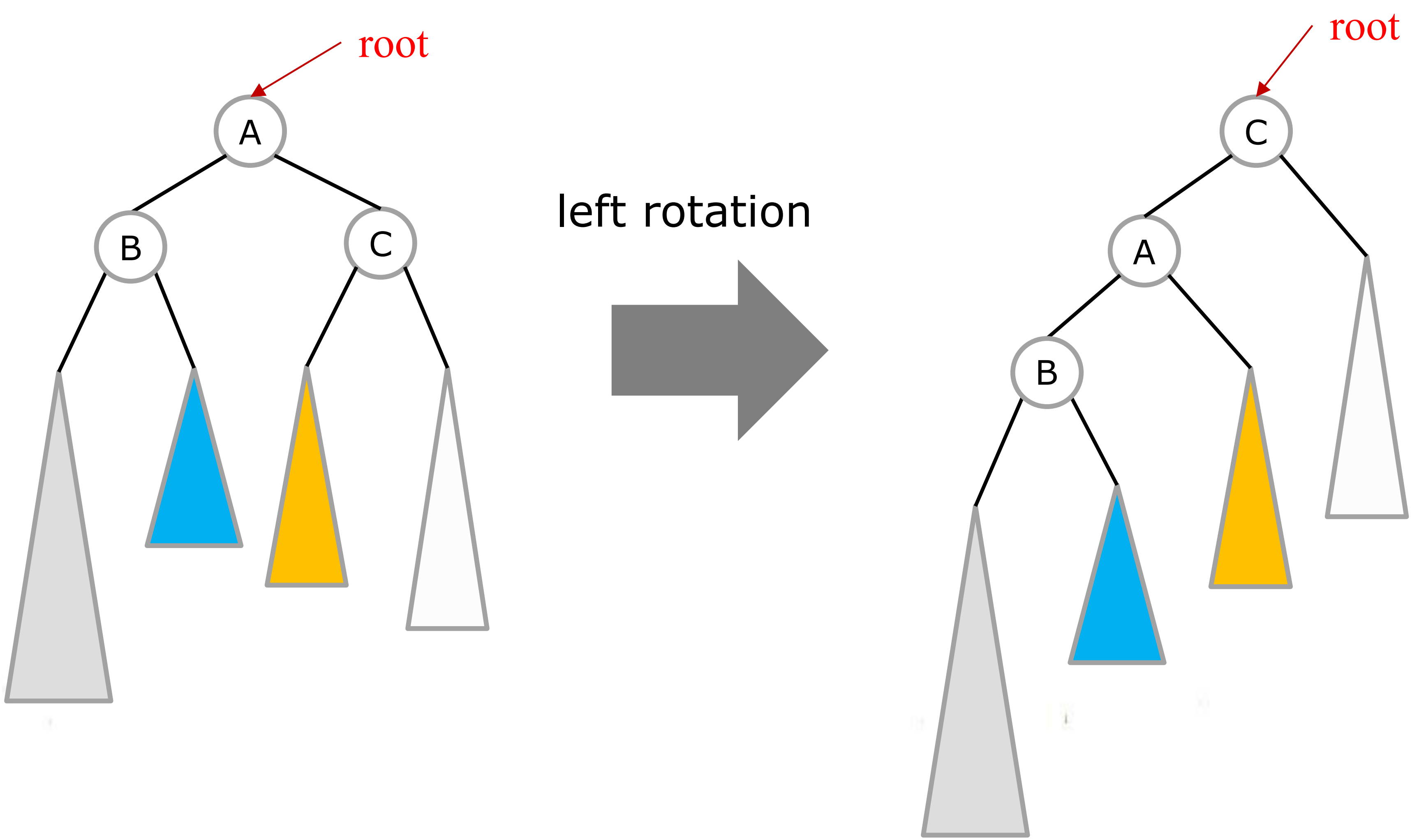
**if** root = NIL
  **then** **return** 0
**else**
  $L \Leftarrow$ **height**$(root \rightarrow left)$
  $R \Leftarrow$ **height**(root $\rightarrow$ *right*)
  **return** max($L$, $R$) + 1

**rightRotate(root)**

t $\Leftarrow$ root $\rightarrow$ *left*
root $\rightarrow$ *left* $\Leftarrow$ t $\rightarrow$ *right*
root $\rightarrow$ *ht* $\Leftarrow$ **getHeight**(root)
t $\rightarrow$ *right* $\Leftarrow$ root
t $\rightarrow$ *ht* $\Leftarrow$ **getHeight**(t)
**return** t

# How to balance the tree in O(log n) time?

- using a series of local operations known as **rotations**



# How to balance the tree in O(log n) time?

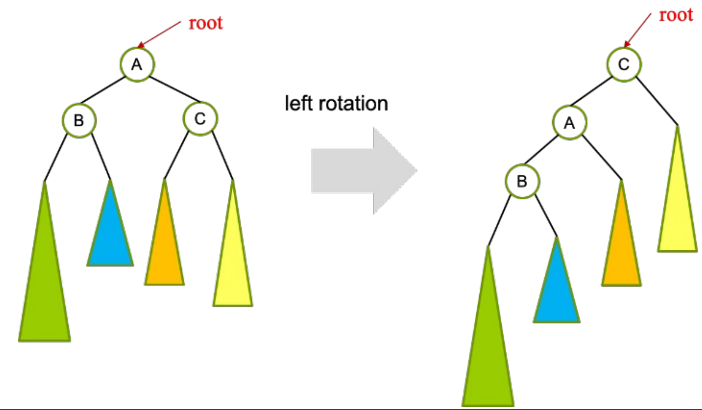
- using a series of local operations known as **rotations**

**height(root)**

1. **if** root = NIL
2.     **then return** 0
3. **else**
4.     **return** root $\rightarrow$ *ht*

**getHeight(root)**

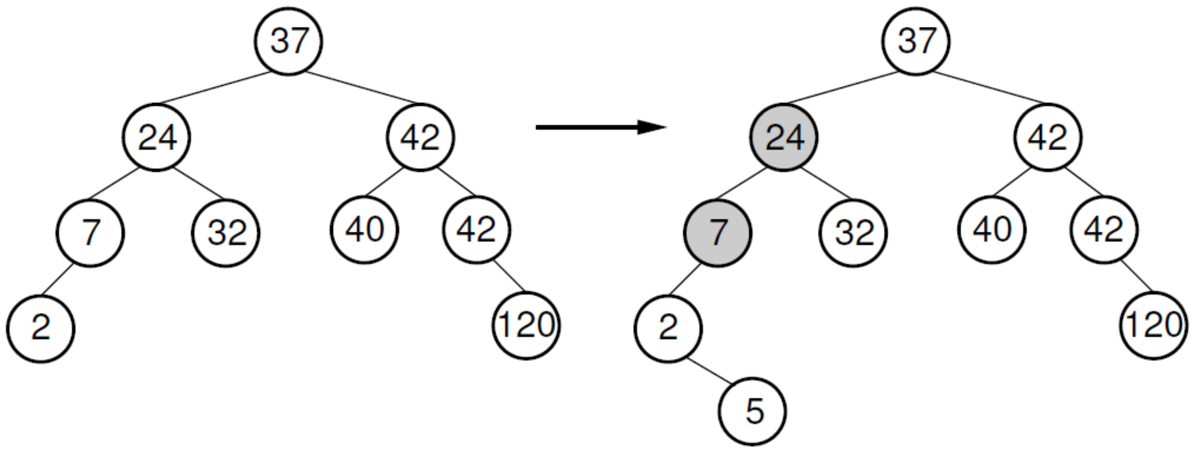
1. **if** root = NIL
2.     **then return** 0
3. else
4.     $L \Leftarrow$ **height**(root $\rightarrow$ *left*)
5.     $R \Leftarrow$ **height**(root $\rightarrow$ *right*)
6.     **return** $\max(L, R) + 1$




**leftRotate(root)**

1. t $\Leftarrow$ root $\rightarrow$ *right*
2. root $\rightarrow$ *right* $\Leftarrow$ t $\rightarrow$ *left*
3. root $\rightarrow$ *ht* $\Leftarrow$ **getHeight**(root)
4. t $\rightarrow$ *left* $\Leftarrow$ root
5. t $\rightarrow$ *ht* $\Leftarrow$ **getHeight**(t)
6. **return** t

# Insertion in AVL tree: Example




After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

For the bottommost unbalanced node, call it S, there are 4 cases:

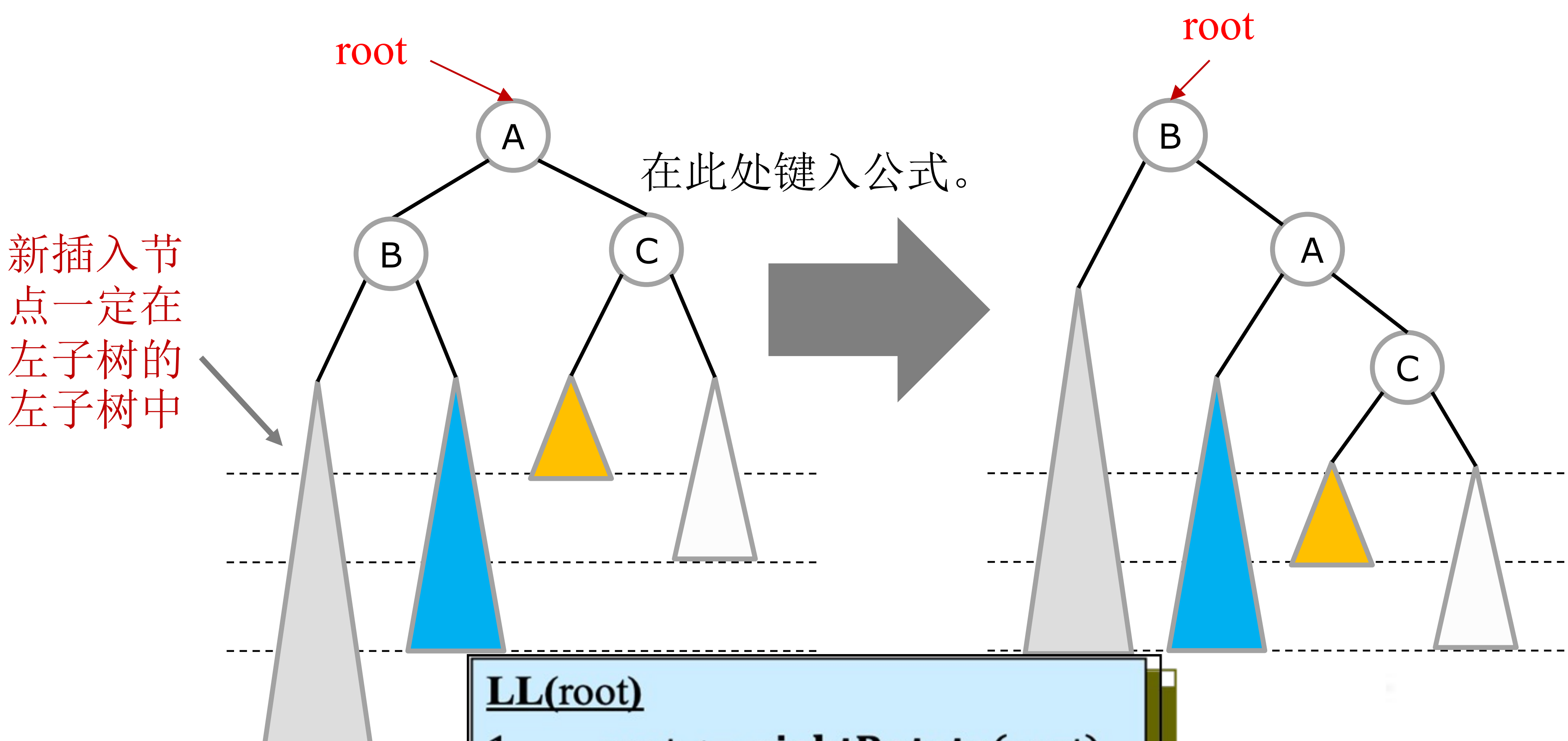
1. LL : the extra node is in the left child of the left child of S.
2. LR : the extra node is in the right child of the left child of S.
3. RL: the extra node is in the left child of the right child of S.
4. RR: the extra node is in the right child of the right child of S.

- LL and RR are symmetrical, as are cases LR and RL.
- Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

# How to balance the tree in O(log n) time?

- **LL:**

$$\textbf{height}(\text{root} \rightarrow left) - \textbf{height}(\text{root} \rightarrow right) = 2$$
$$\&\&\ \textbf{height}(\text{root} \rightarrow left \rightarrow left) > \textbf{height}(\text{root} \rightarrow left \rightarrow right)$$




**LL(root)**

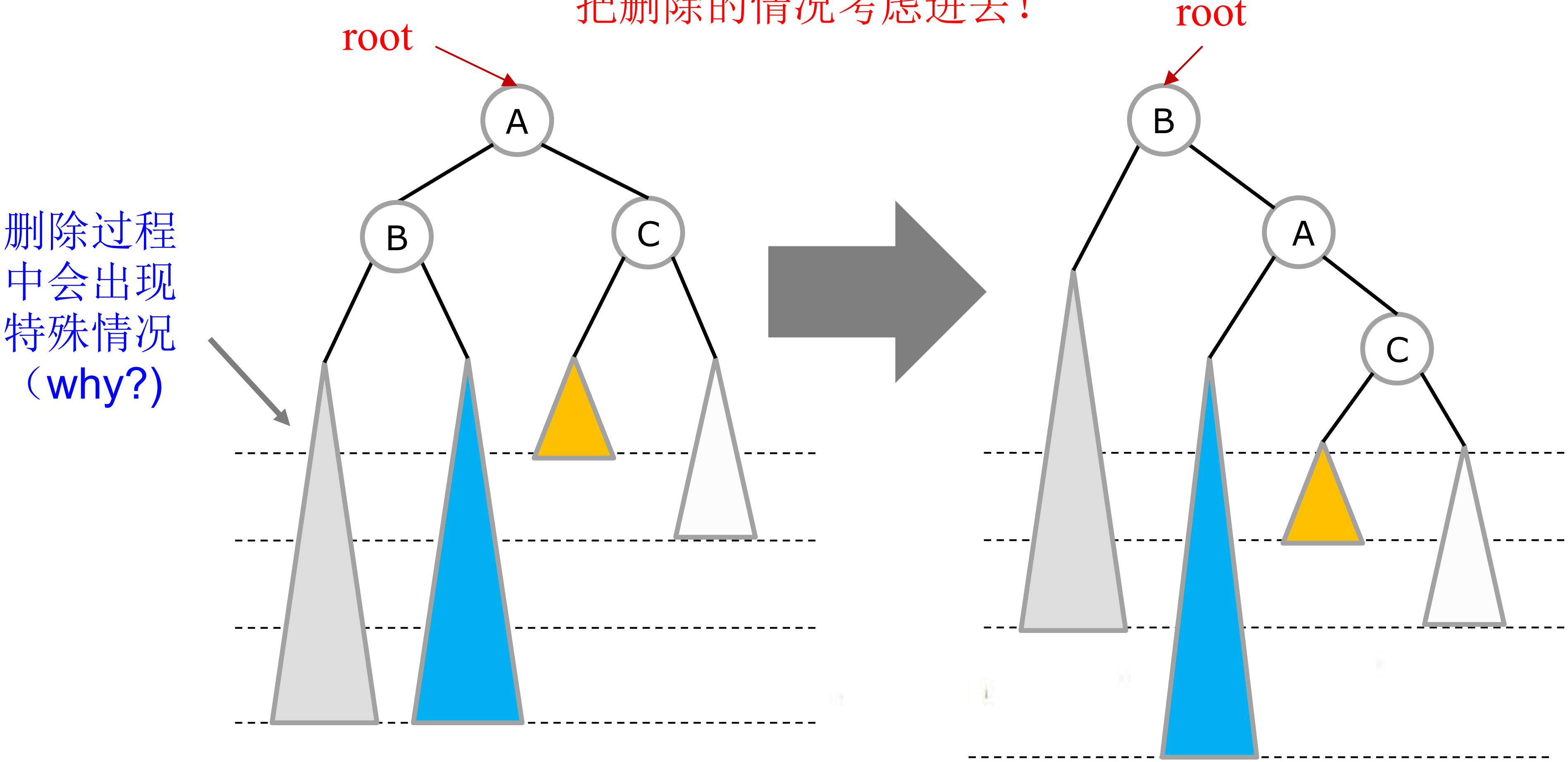
1. root $\Leftarrow$ **rightRotate**(root)
2. **return** root

# How to balance the tree in O(log n) time?

- **LL:**

$$\mathbf{height}(\text{root} \to left) - \mathbf{height}(\text{root} \to right) = 2$$
$$\&\&\ \mathbf{height}(\text{root} \to left \to left) \geq \mathbf{height}(\text{root} \to left \to right)$$

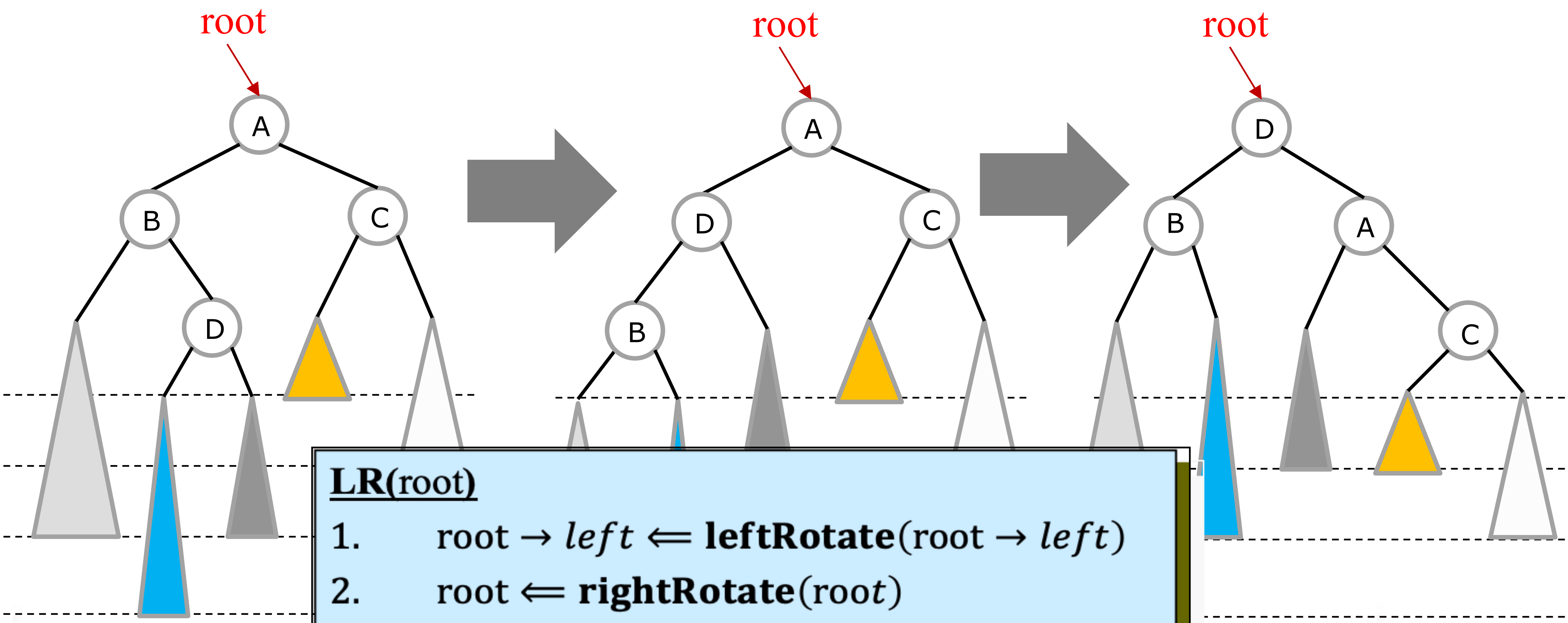
把删除的情况考虑进去！



# How to balance the tree in O(log n) time?

- **LR:**

$$\textbf{height}(\text{root} \to \mathit{left}) - \textbf{height}(\text{root} \to \mathit{right}) = 2$$
$$\&\&\ \textbf{height}(\text{root} \to \mathit{left} \to \mathit{left}) < \textbf{height}(\text{root} \to \mathit{left} \to \mathit{right})$$

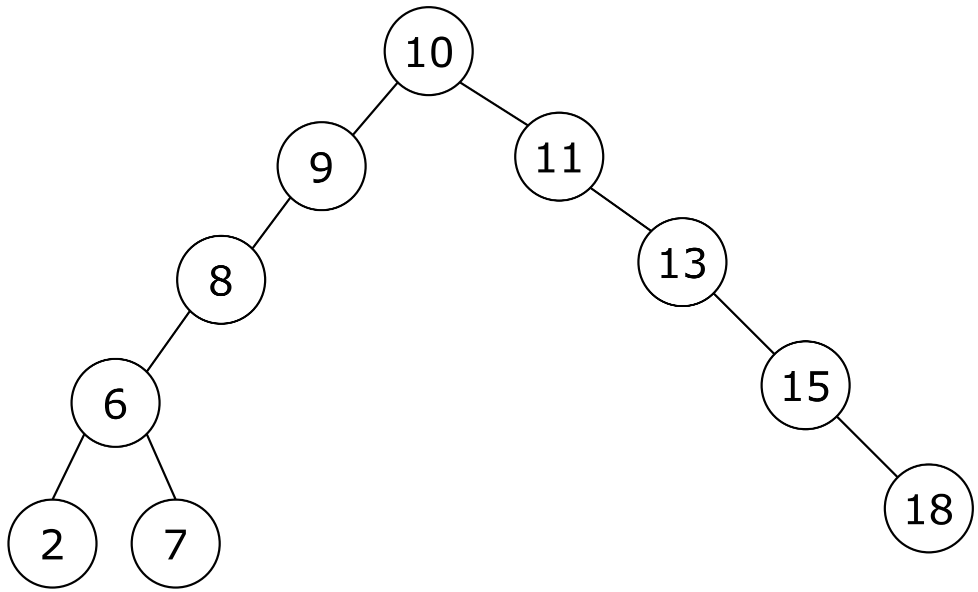



**<u>LR(root)</u>**

1. $\text{root} \to \mathit{left} \Leftarrow \textbf{leftRotate}(\text{root} \to \mathit{left})$
2. $\text{root} \Leftarrow \textbf{rightRotate}(\text{root})$
3. **return** root

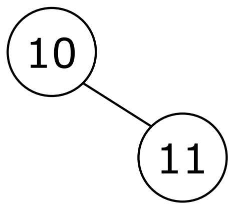
# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，7，2 to BST



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

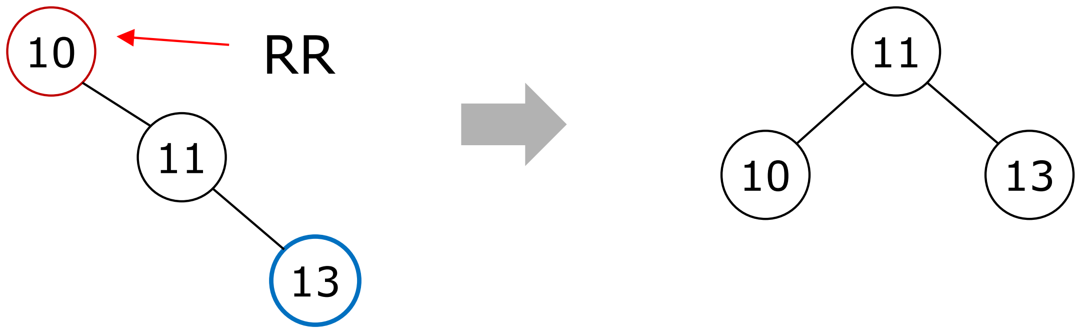



insert 10,11

# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

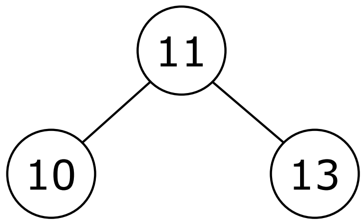
insert 13



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

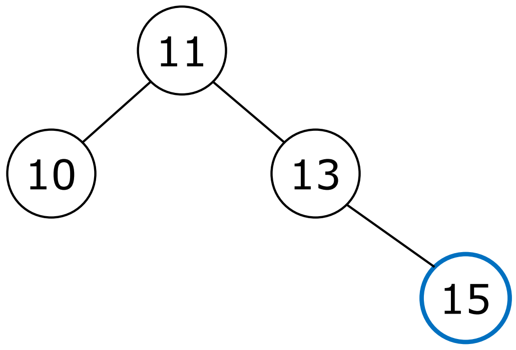
insert 13



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

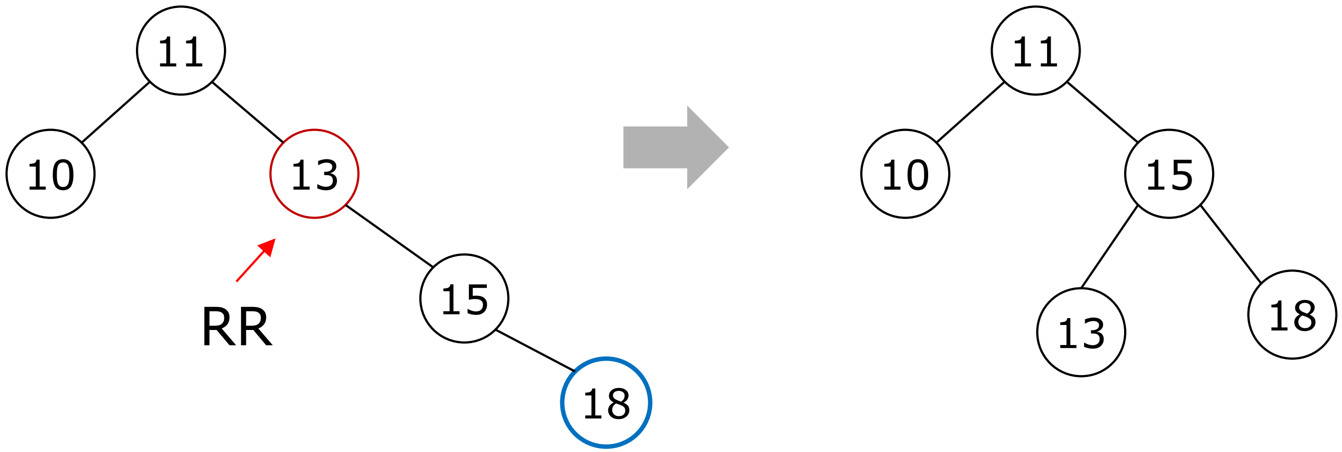
insert 15



# Insertion in AVL tree: Example

Insert 10， 11， 13， 15， 18， 9， 8， 6， 5， 2 to AVL

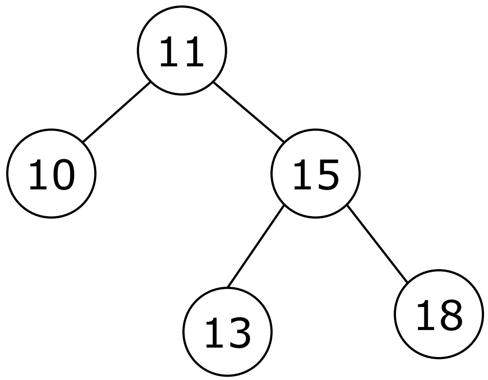
insert 18



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

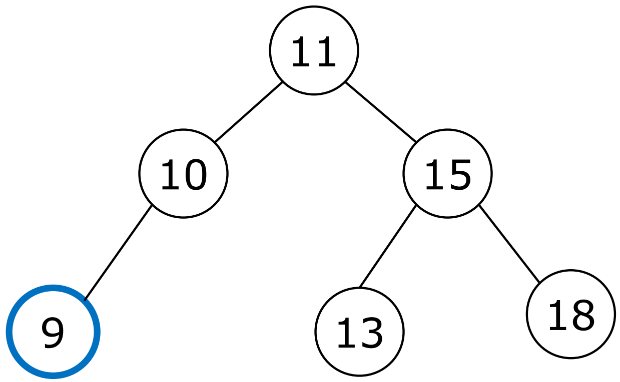
insert 18



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

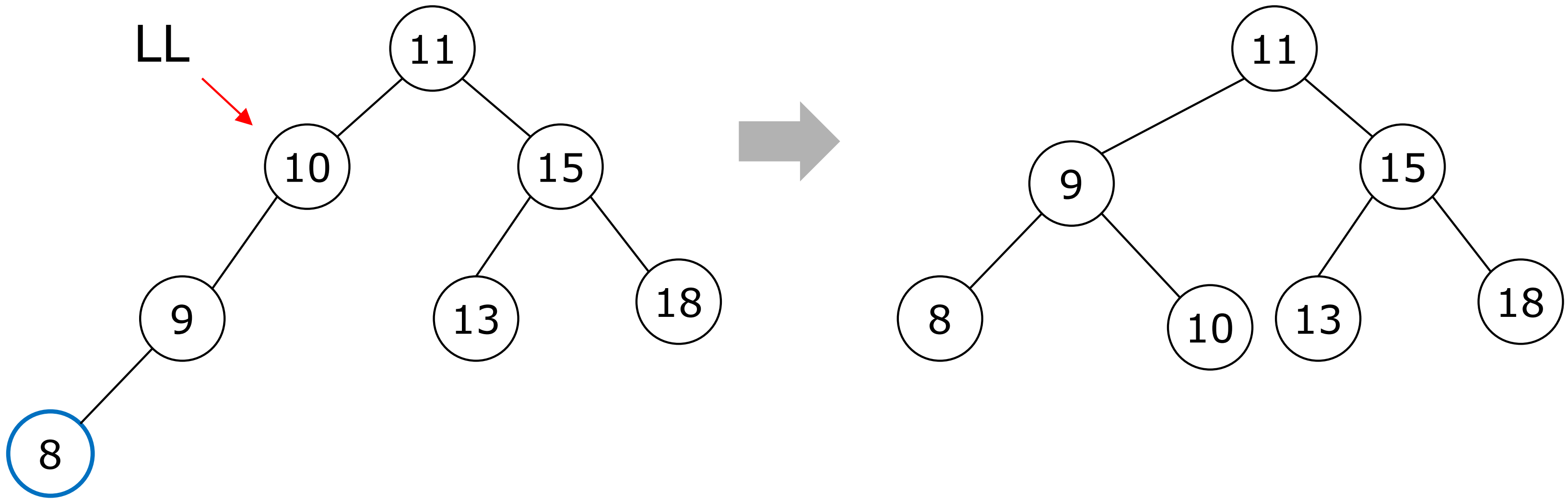
insert 9



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

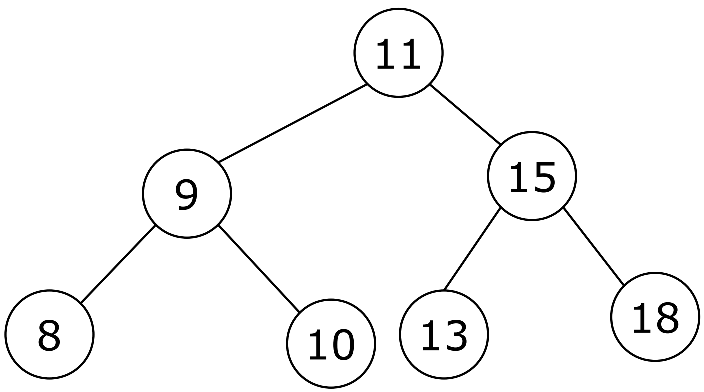
insert 8



# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，5，2 to AVL

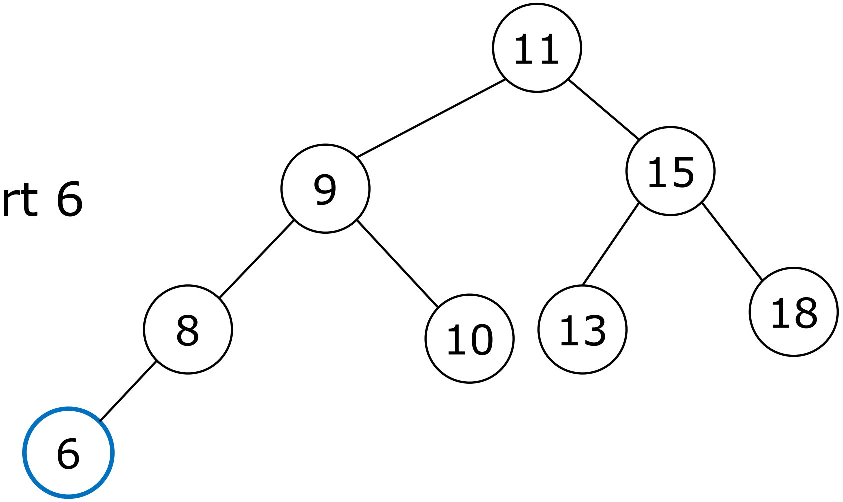
insert 8



# Insertion in AVL tree: Example

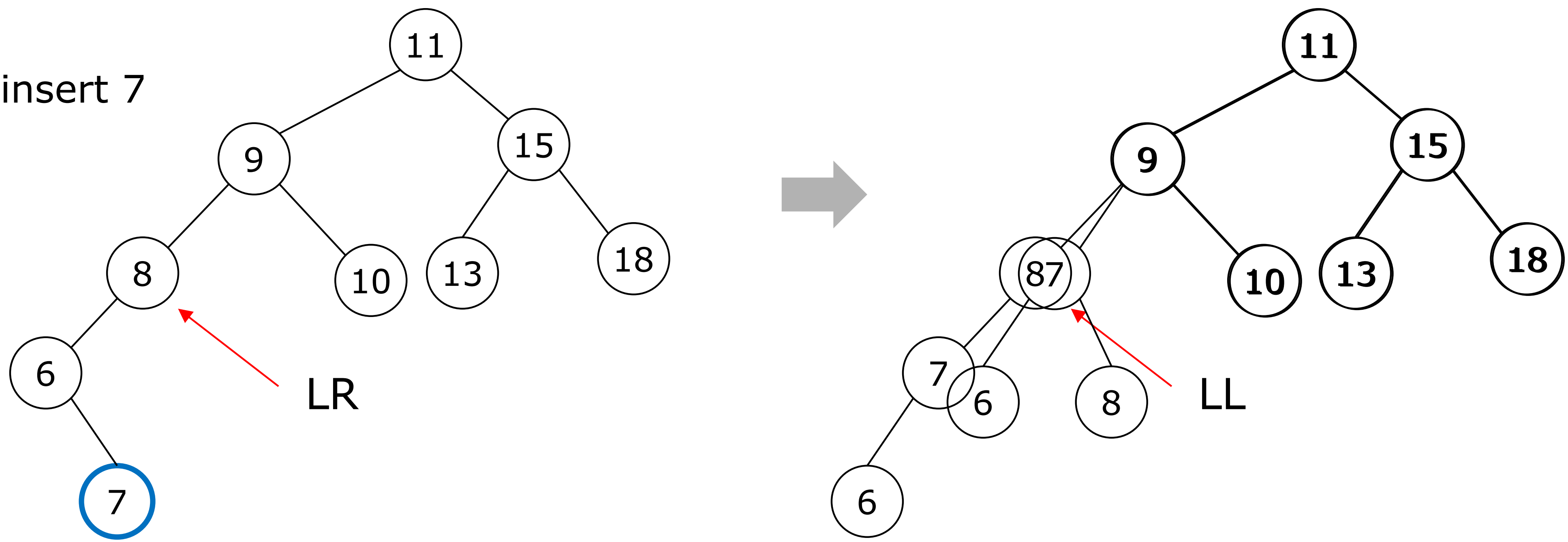
Insert 10， 11， 13， 15， 18， 9， 8， 6， 5， 2 to AVL

insert 6



# Insertion in AVL tree: Example

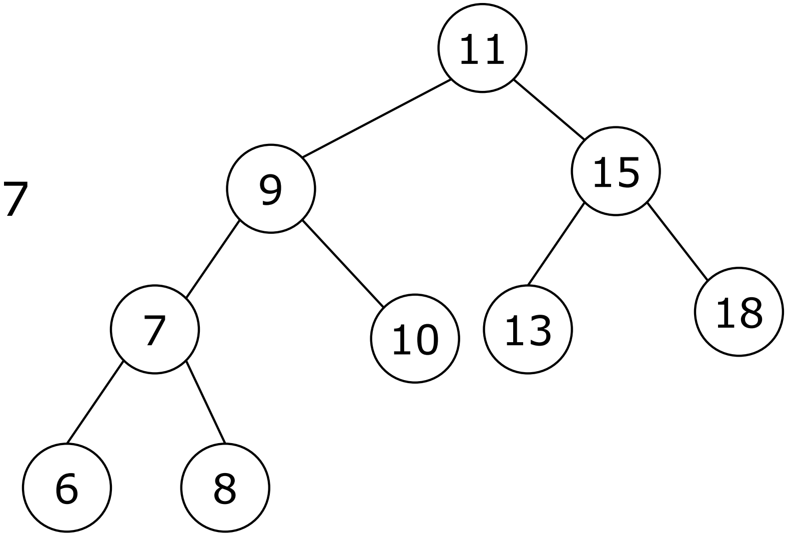
Insert 10，11，13，15，18，9，8，6，7，2 to AVL



# Insertion in AVL tree: Example

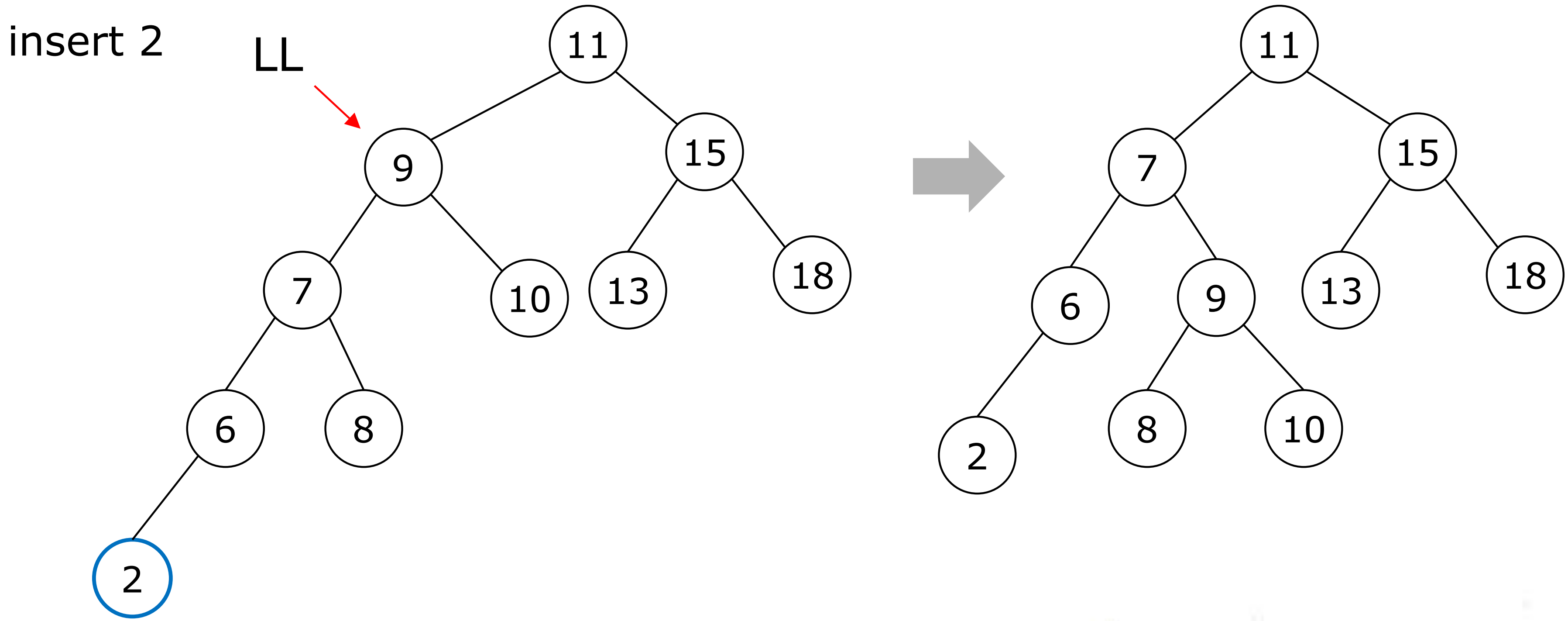
Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 7



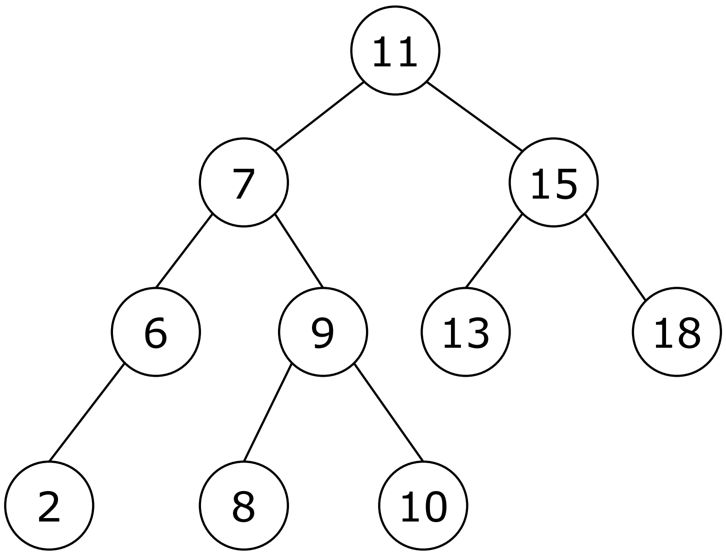
# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

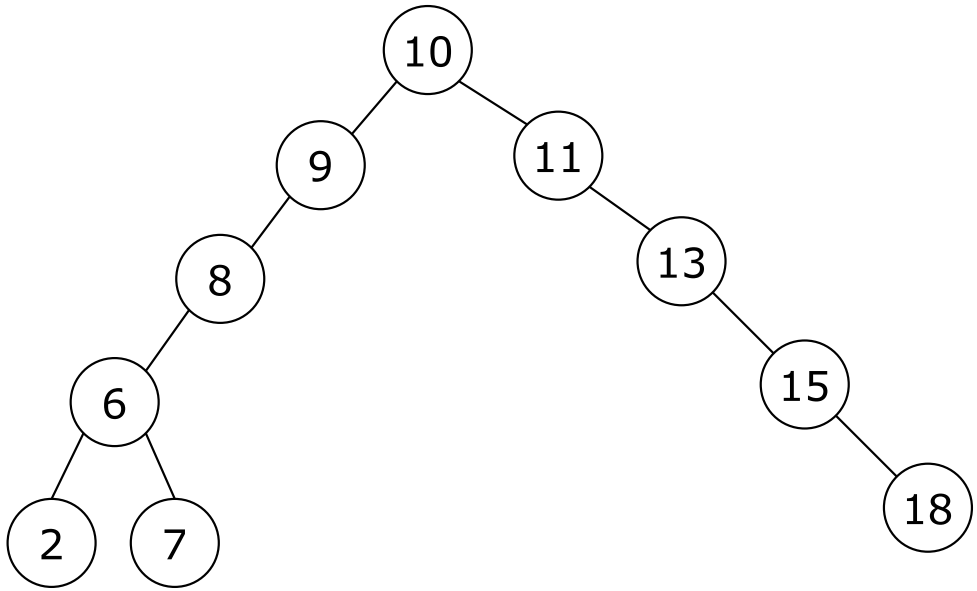


# Insertion in AVL tree: Example

Insert 10，11，13，15，18，9，8，6，7，2 to AVL



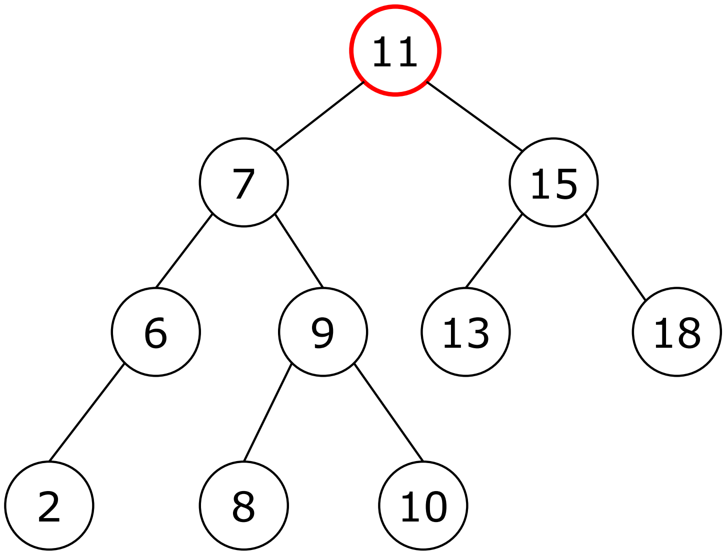

AVL



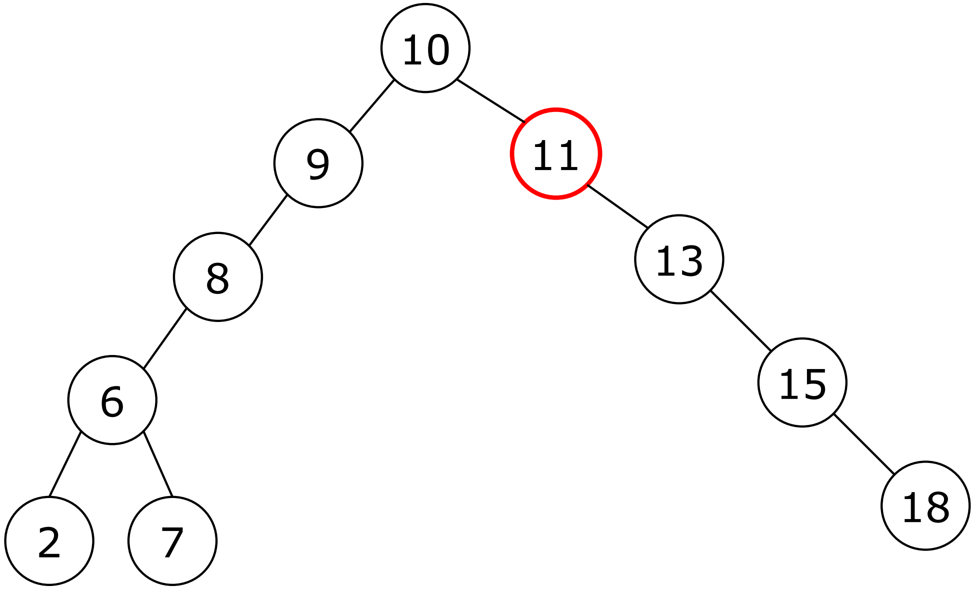

BST

# Deletion in AVL tree: Example

Delete 11



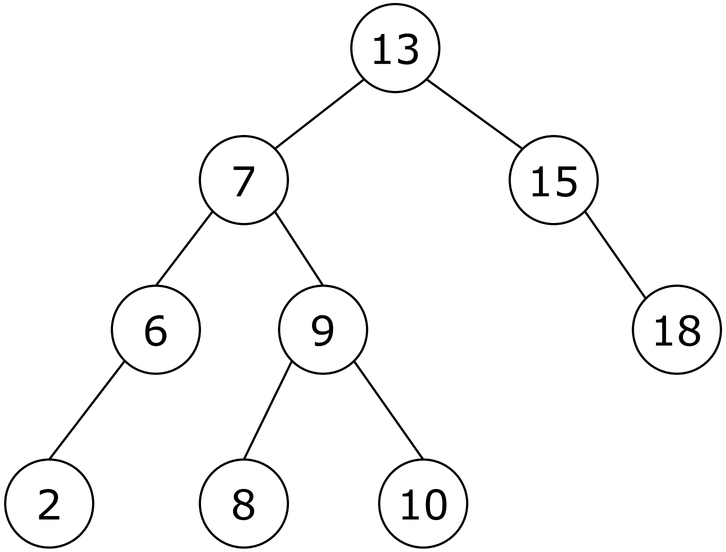

AVL



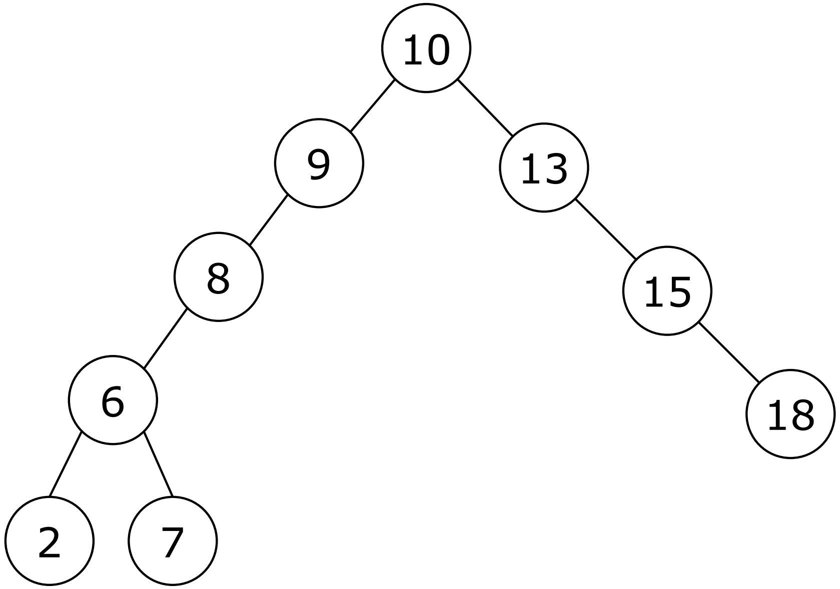

BST

# Deletion in AVL tree: Example

Delete 11



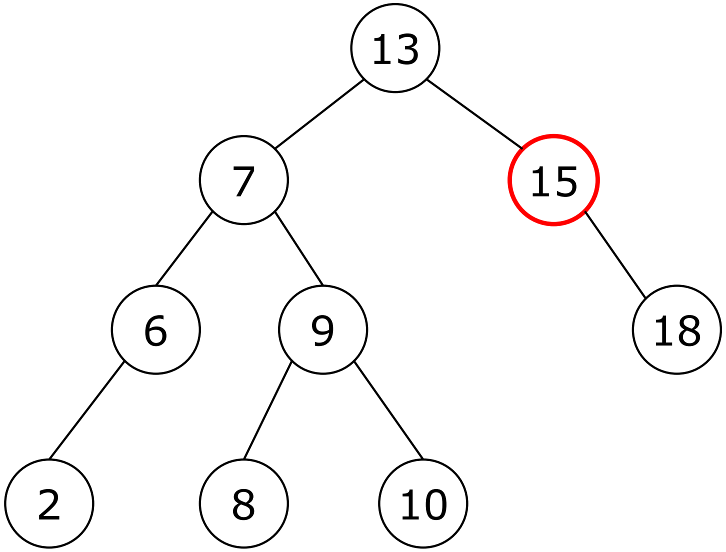

AVL



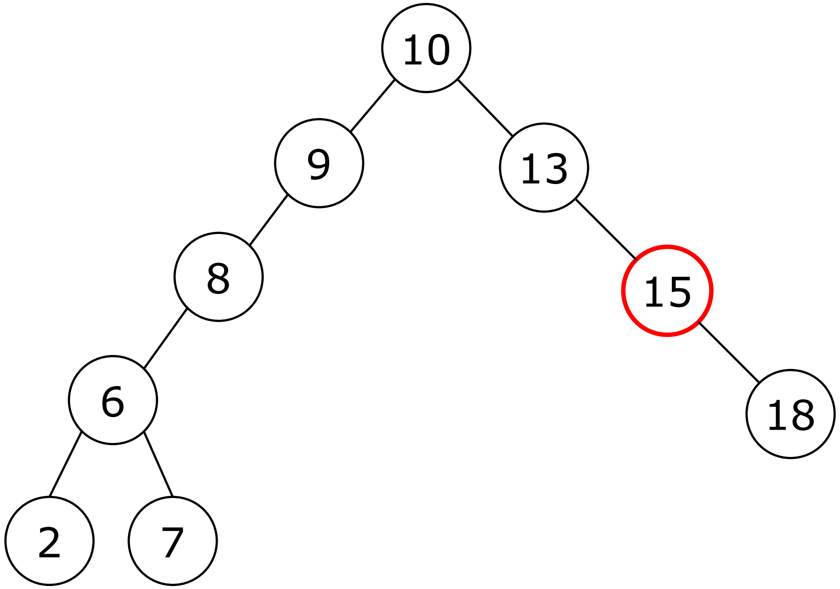

BST

# Deletion in AVL tree: Example

Delete 15



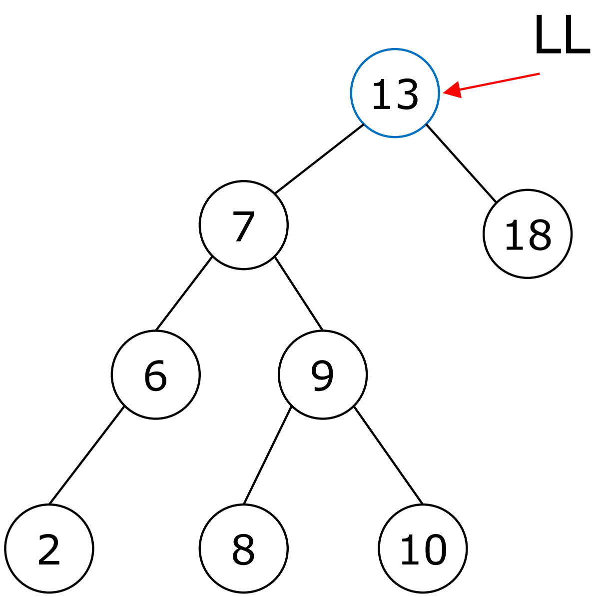

AVL



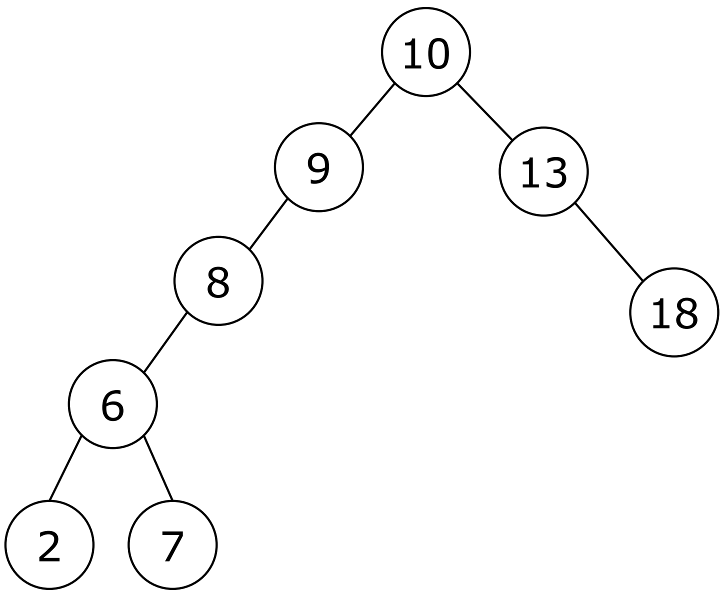

BST

# Deletion in AVL tree: Example

Delete 15

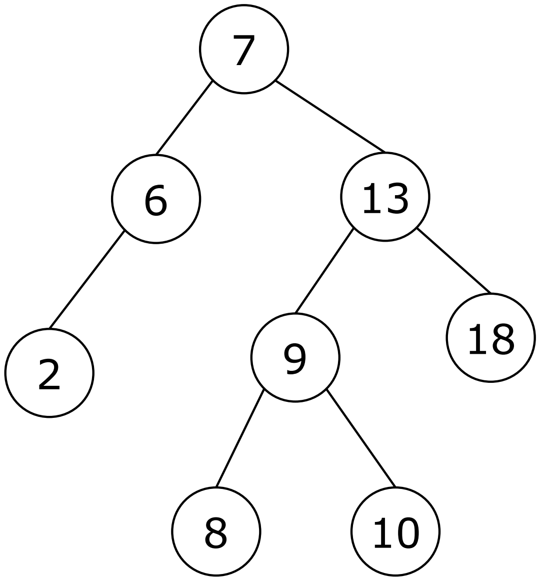



AVL

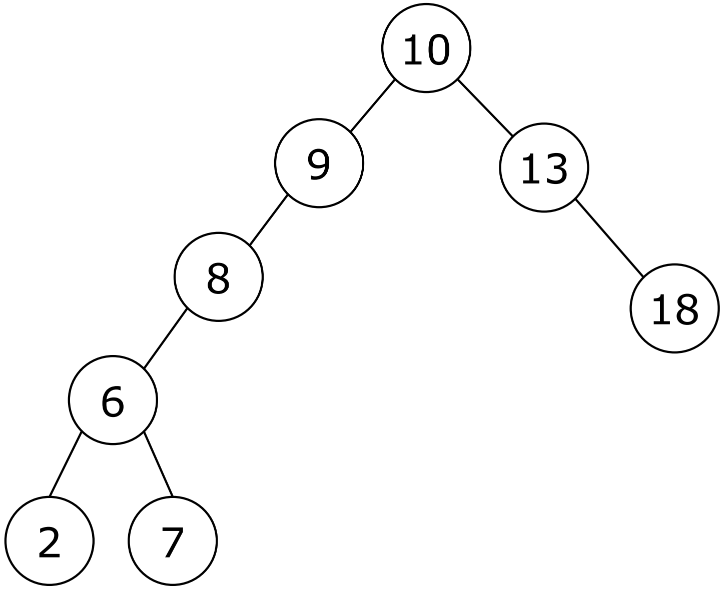



BST

# Deletion in AVL tree: Example




AVL




BST

# Operations in AVL tree

- Insertion algorithm:

1. begin with a normal BST insert
2. Then as the recursion unwinds up the tree, perform the appropriate rotation on any node that is found to be unbalanced.

- Deletion is similar
  - consideration for unbalanced nodes must begin at the level of the deletemin operation.

# Balancing AVL tree

**Balancing**(root, node) //node为新插入的节点或删除节点的父节点

**if** $node \to val < root \to val$
    **then** $root \to left \Leftarrow \boldsymbol{Balancing}(root \to left, node)$
**else if** $node \neq root$
    **then** $root \to right \Leftarrow \boldsymbol{Balancing}(root \to right, node)$

二分递归

$root \to ht \Leftarrow \boldsymbol{getHeight}(root)$
**if** $\boldsymbol{height}(root \to left) - \boldsymbol{height}(root \to right) = 2$
    **then if** $\boldsymbol{height}(root \to left \to left) < \boldsymbol{height}(root \to left \to right)$
        **then** $root \to left \Leftarrow \boldsymbol{leftRotate}(root \to left)$ //LR → LL
    $root \Leftarrow \boldsymbol{rightRotate}(root)$ //LL

**if** $\boldsymbol{height}(root \to right) - \boldsymbol{height}(root \to left) = 2$
    **then if** $\boldsymbol{height}(root \to right \to right) < \boldsymbol{height}(root \to right \to left)$
        **then** $root \to right \Leftarrow \boldsymbol{rightRotate}(root \to right)$ //RL→RR
    $root \Leftarrow \boldsymbol{leftRotate}(root)$ //RR

平衡

**return** root

《数据结构与算法》课程组

重庆大学计算机学院

# End of Chapter