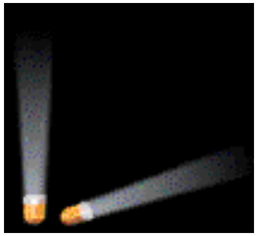
Efficient Binary Search Trees



- Binary Search Tree
  - height can be as large as N
  - Complexity: Search, Insert, Delete
    - O(n)
- We want a tree with small height
- A binary tree with N node has height at least
  - $\Theta(\log N)$
- Our goal
  - keep the height of a binary search tree O(log N)

balanced binary search trees

- AVL tree
  - **Adelson-Velskii and Landis**
- Red-black tree

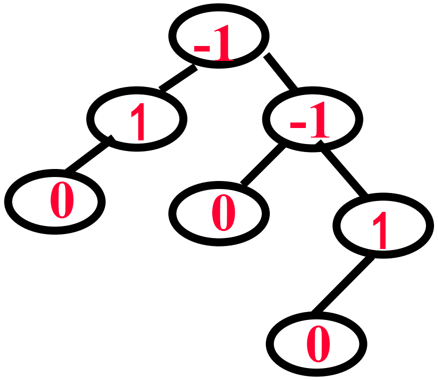
AVL Tree

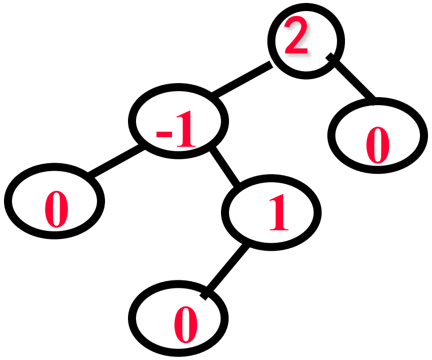
- an empty tree is height-balanced
- If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is height-balanced iff
- (1) T_L and T_R are height-balanced and
- (2) $| h_L - h_R | \le 1$ where h_L and h_R are the heights of T_L and T_R respectively

Balance Factor

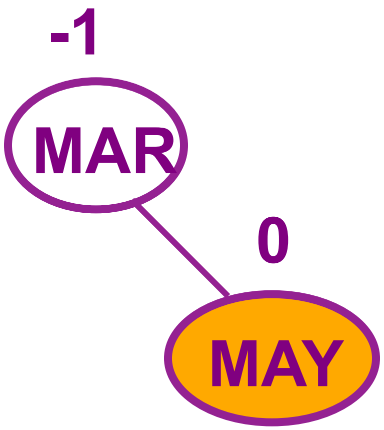
- The balance factor, BF(T), of a node T in a binary tree
  - $h_L - h_R$
- For any node T in an AVL tree
  - BF(T)= -1, 0, or 1.♪

AVL Tree?





(a) Insert MAR

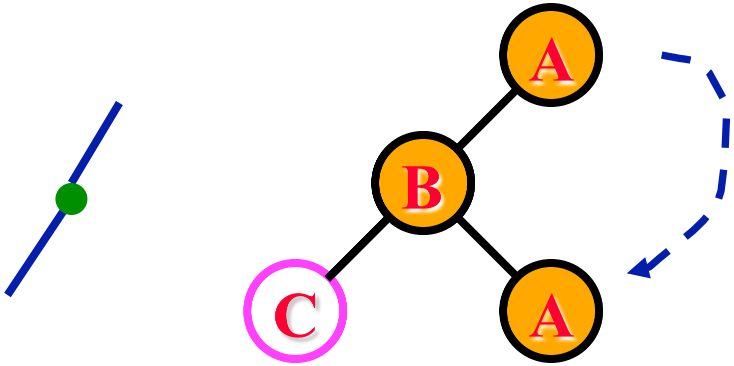


(b) Insert MAY

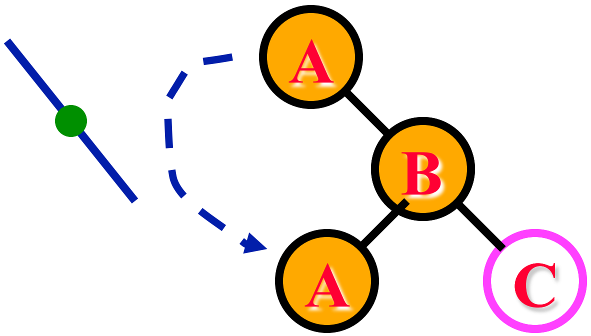
(c) Insert NOV

- Insertion may leads to unbalancing ！
- Rebalance it!

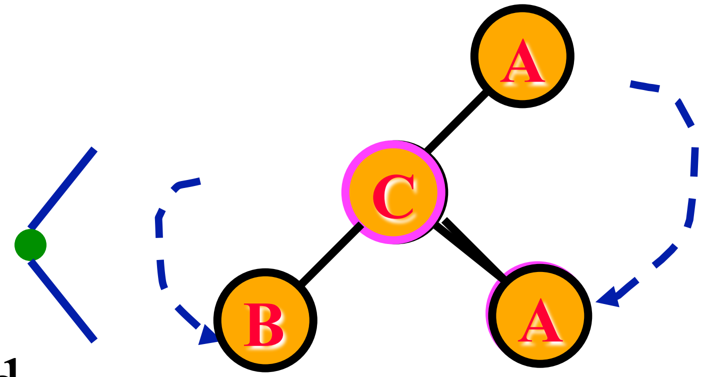
- LL
  - BF(A) = 2
  - Caused by insertion to the left-subtree of A's left-child



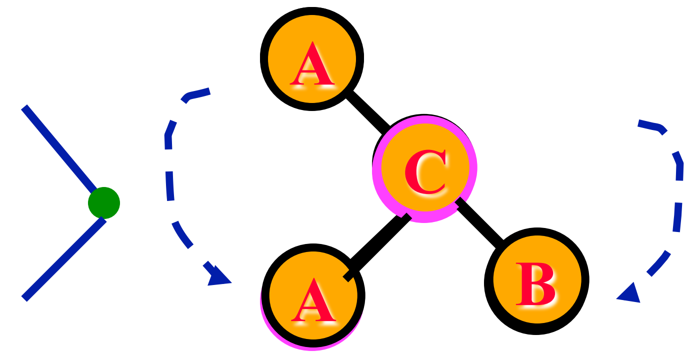
- RR
  - BF(A) = -2
  - Caused by insertion to the right-subtree of A's right -child



- LR
  - BF(A) = 2
  - Caused by insertion to the right-subtree of A’s left-child

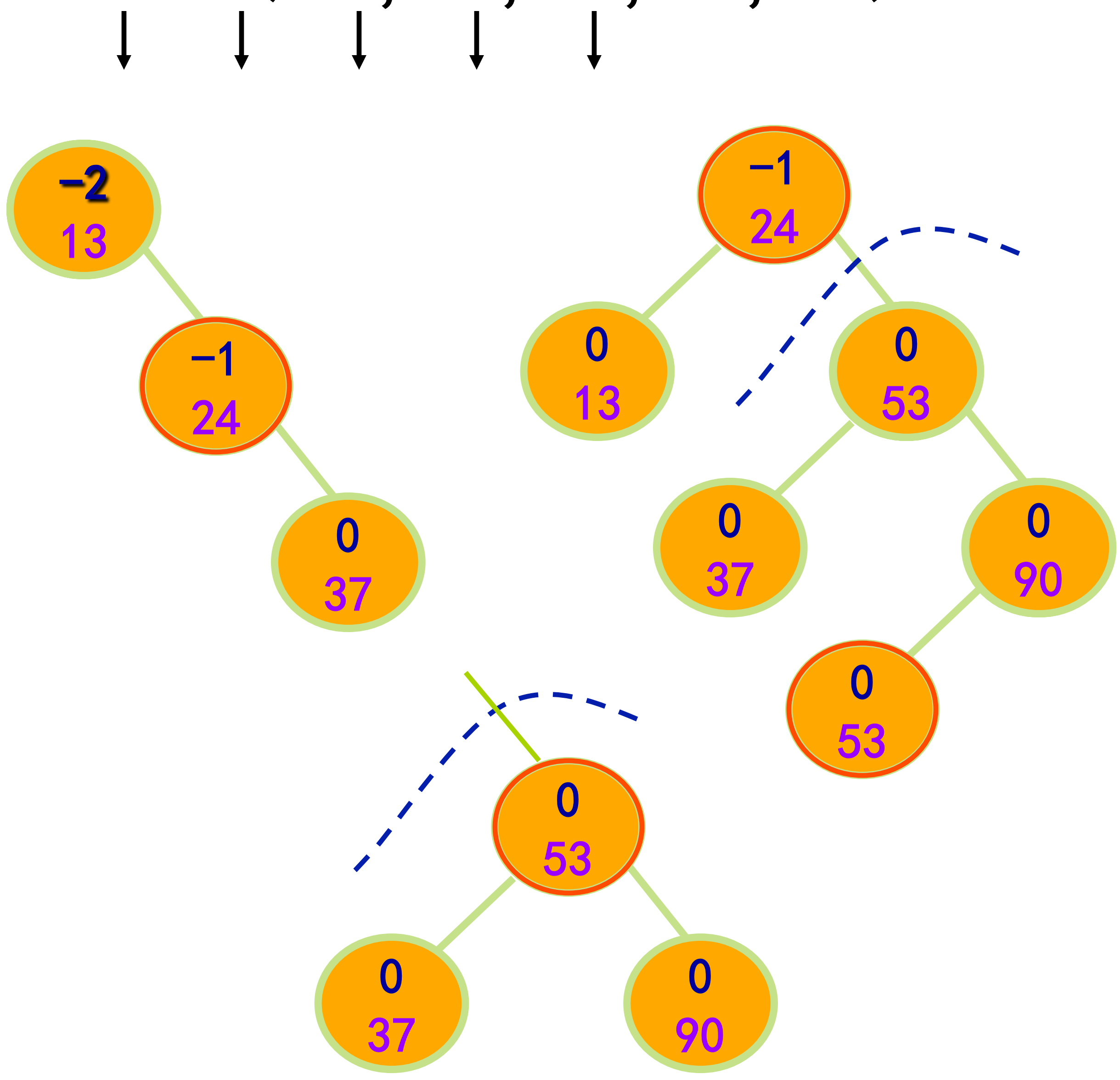


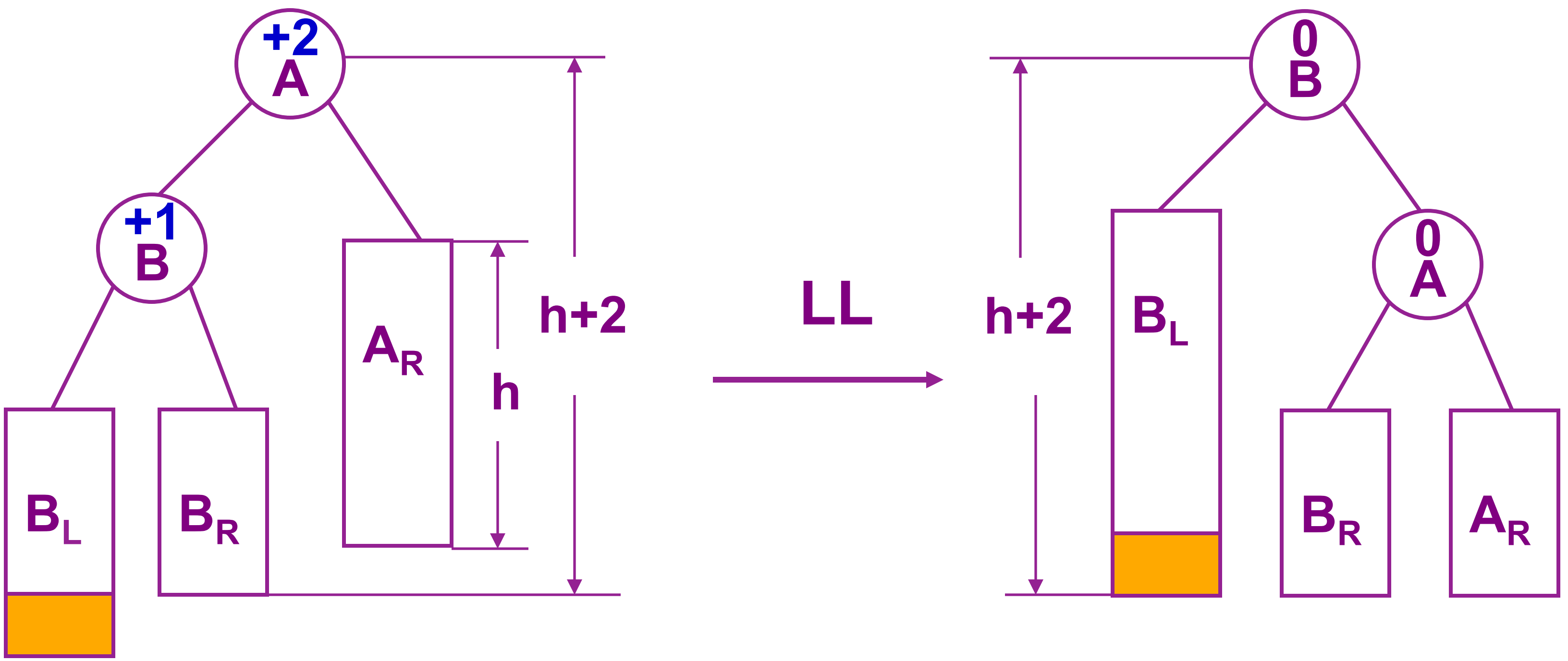
- RL
  - BF(A) = -2
  - Caused by insertion to the left-subtree of A’s right-child

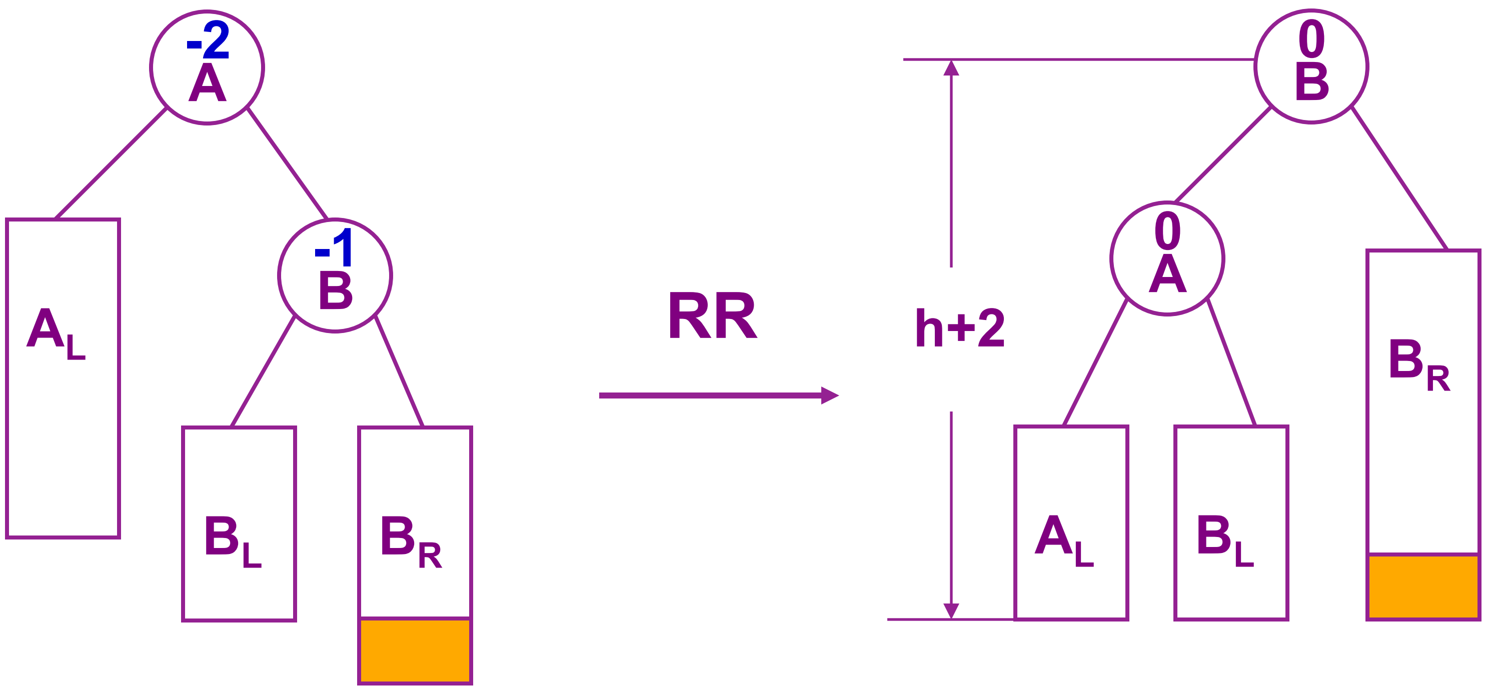


Build an AVL Tree

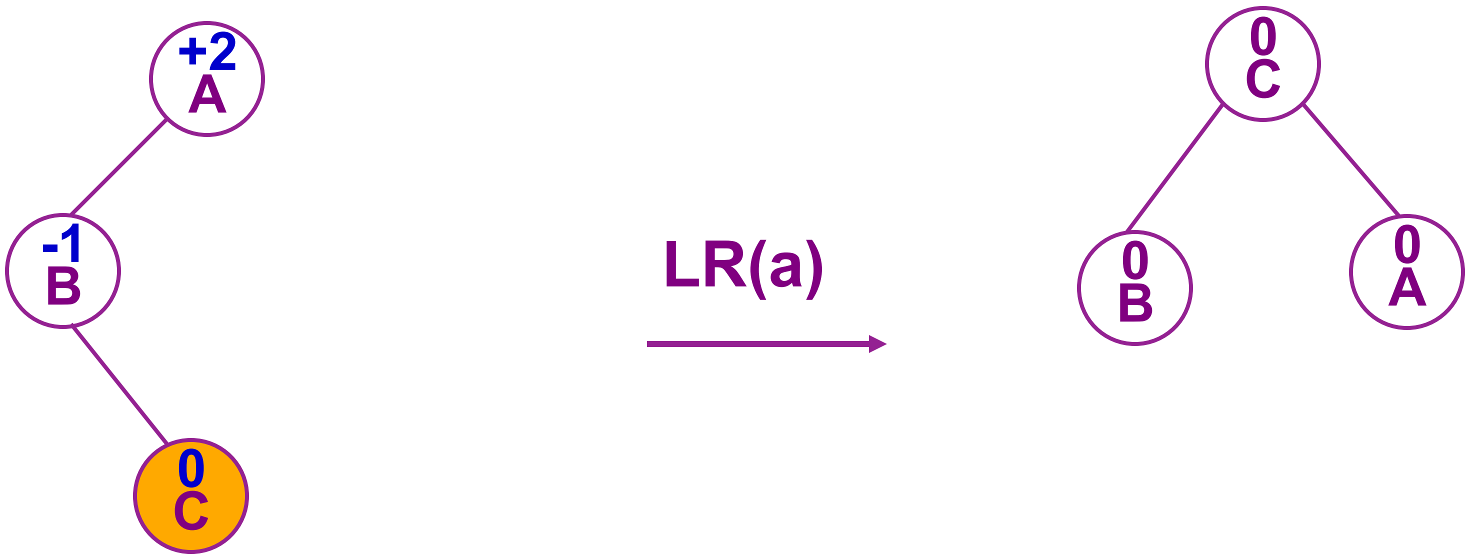
（ 13， 24， 37， 90， 53）

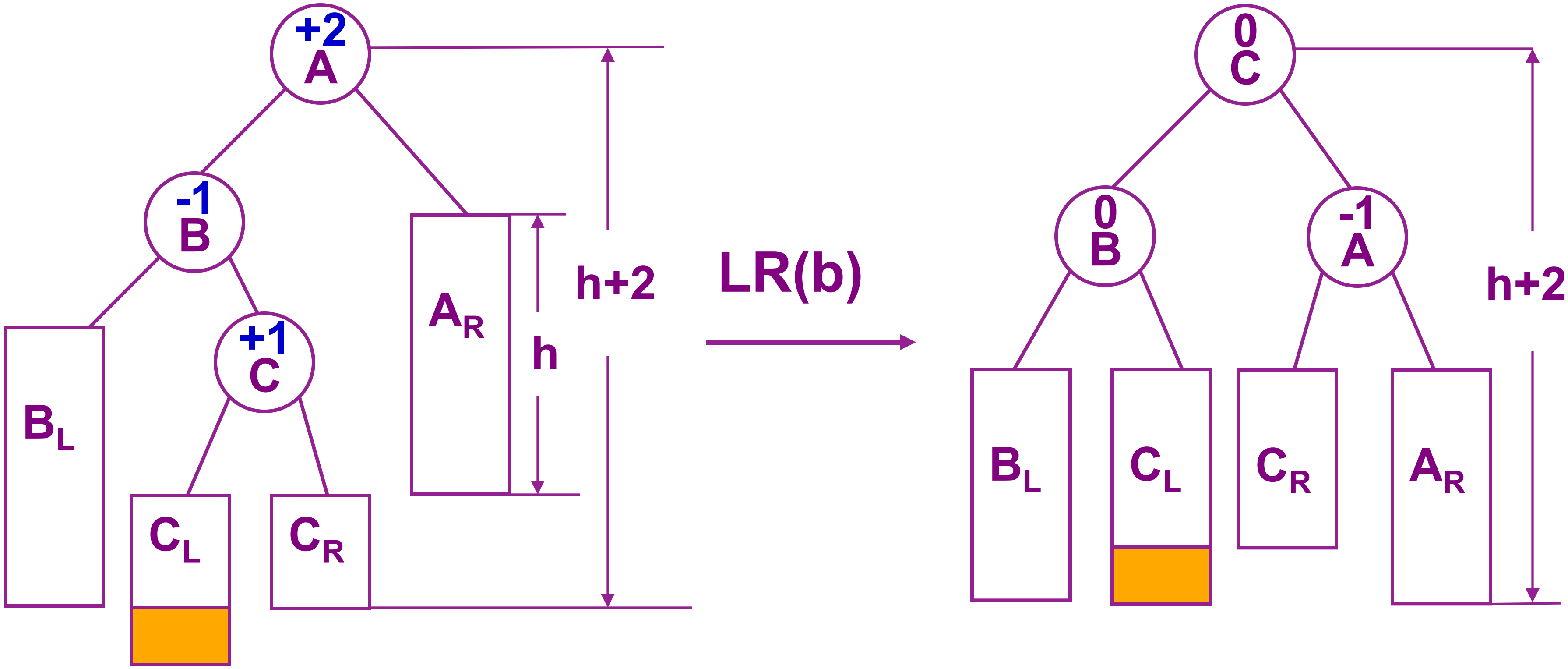


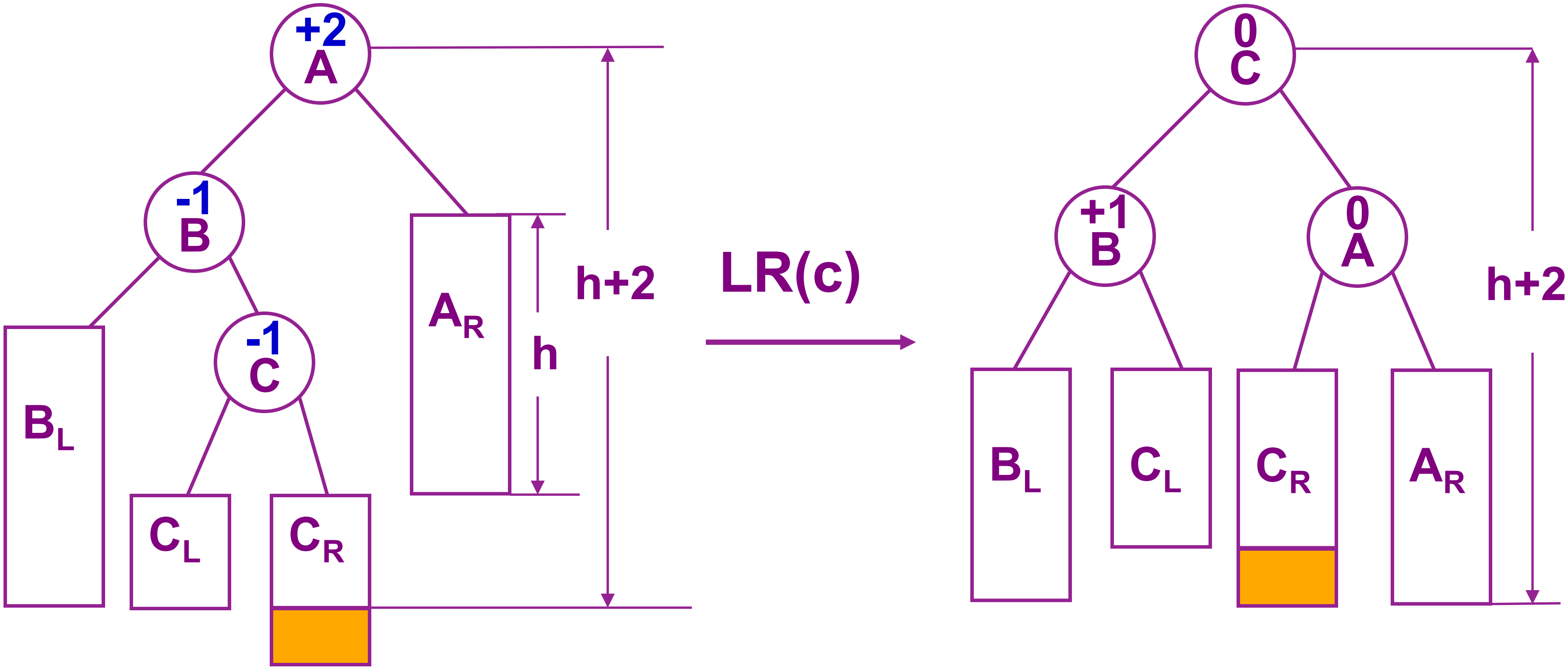

+2
A
+1
B
A_R
h
h+2
B_L
B_R
LL
0
B
0
A
h+2
B_L
B_R
A_R



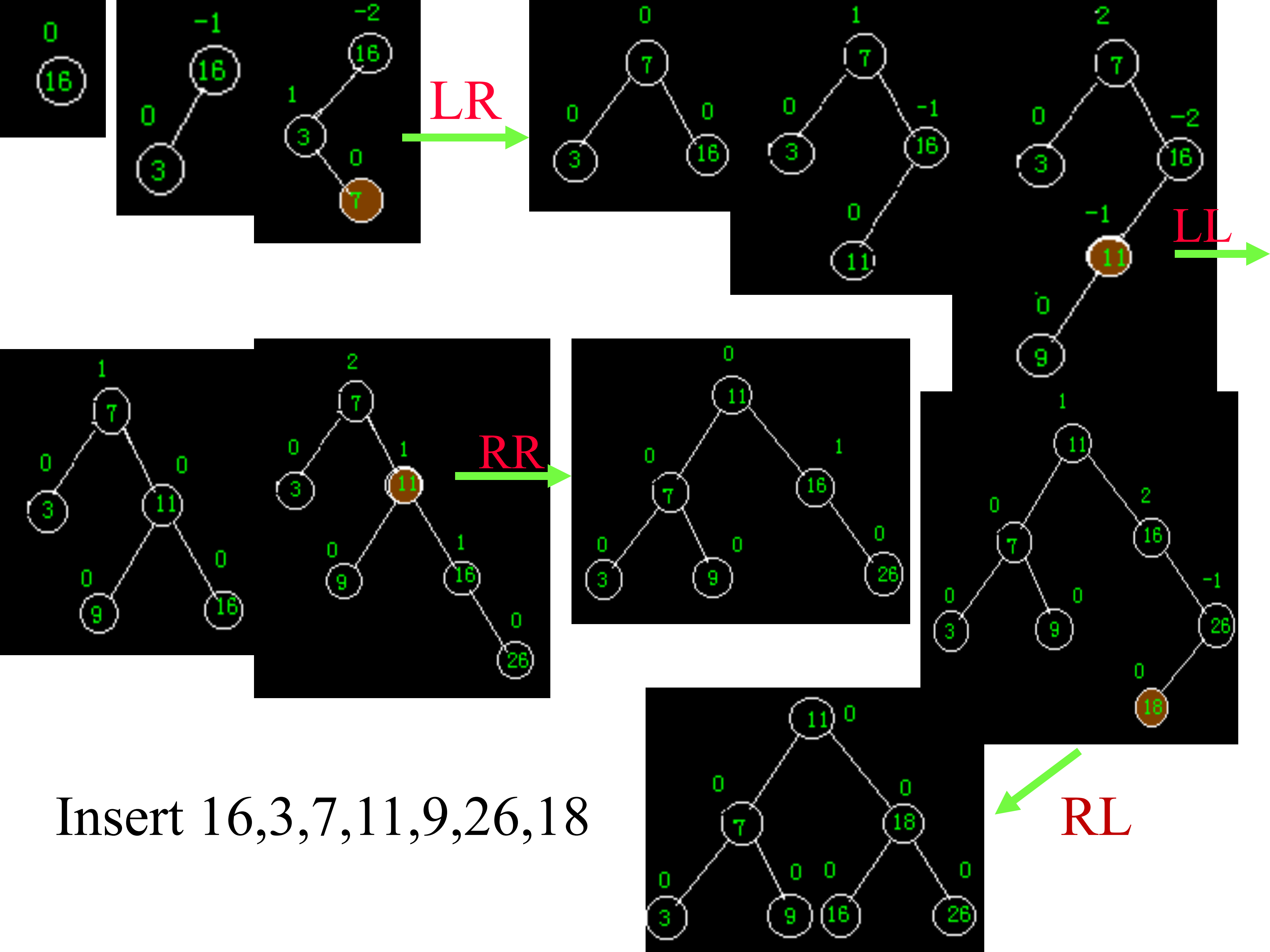
-2
A
-1
B
AL
BL
BR
RR
h+2
0
B
0
A
AL
BL
BR


+2
A
-1
B
0
C
LR(a)
0
C
0
B
0
A


+2
A
-1
B
+1
C
BL
CL
CR
AR
h
h+2
LR(b)
0
C
0
B
-1
A
BL
CL
CR
AR
h+2



+2
A
-1
B
-1
C
B_L
C_L
C_R
A_R
h
h+2
LR(c)
0
C
+1
B
0
A
B_L
C_L
C_R
A_R
h+2



Insert 16,3,7,11,9,26,18

Notations

- The height of the subtree involved in the rotation is the same after rebalancing as it was before
- The only nodes whose BF can change are those in the subtree that is rotated.

Notations

- Node A
  - the nearest ancestor of Y, whose BF becomes ± 2
  - the nearest ancestor with BF= ± 1 before insertion.
- Before the insertion, the BF's of all nodes on the path from A to the new insertion point must have been 0
- To complete the rotation, the parent of A, F is also needed (Why?)

Notations

- Whether or not the restructuring is needed, the BF's of several nodes change
- Let A be the nearest ancestor of the new node with BF=± 1 before the insertion
  - If no such an A, let A be the root.
  - The BF's of nodes from A to the parent of the new node will change to ± 1

- template <class K, class E> class AvlNode {
- friend class AVL<K, E>;
- public:
- AvlNode(const K& k, const E& e)
- {key=k; element=e; bf=0;
- leftChild=rightChild=0;}
- private:
- K key;
- E element
- int bf;;
- AvlNode<K, E> *leftChild, *rightChild;
- };

```
• template <class K, class E>
• class AVL {
• public:
•       AVL(): root(0) { };
•       E& Search(const K&) const;
•       void Insert(const  K&, const E&);
•       void Delete(const  K&);
• private:
•       AvlNode<K, E> *root;
• };
```

- template <class K, class E>
- void AVL<K, E>::Insert(const K& k, const E& e){
-     if (!root) {   // empty tree
-         root=new AvlNode<K, E>(k, e);
-         return;
-     }
-    // phase 1: Locate insertion point for e.
-    AvlNode<K, E> *a=root, // most recent node with BF±1
-             *pa,   // parent of a
-             *p=root, // p move through the tree
-             *pp=0;   // parent of p

```
while (p)  {  // search for insertion point for x
        if (p→bf)
               {a=p; pa=pp;}
        if (k<p→key)
               {pp=p; p=p→leftChild;}
        else if (k>p→key)
               {pp=p; p=p→rightChild;}
                else
               {p→element=e; return;} // k in the tree
 } // end of while
```

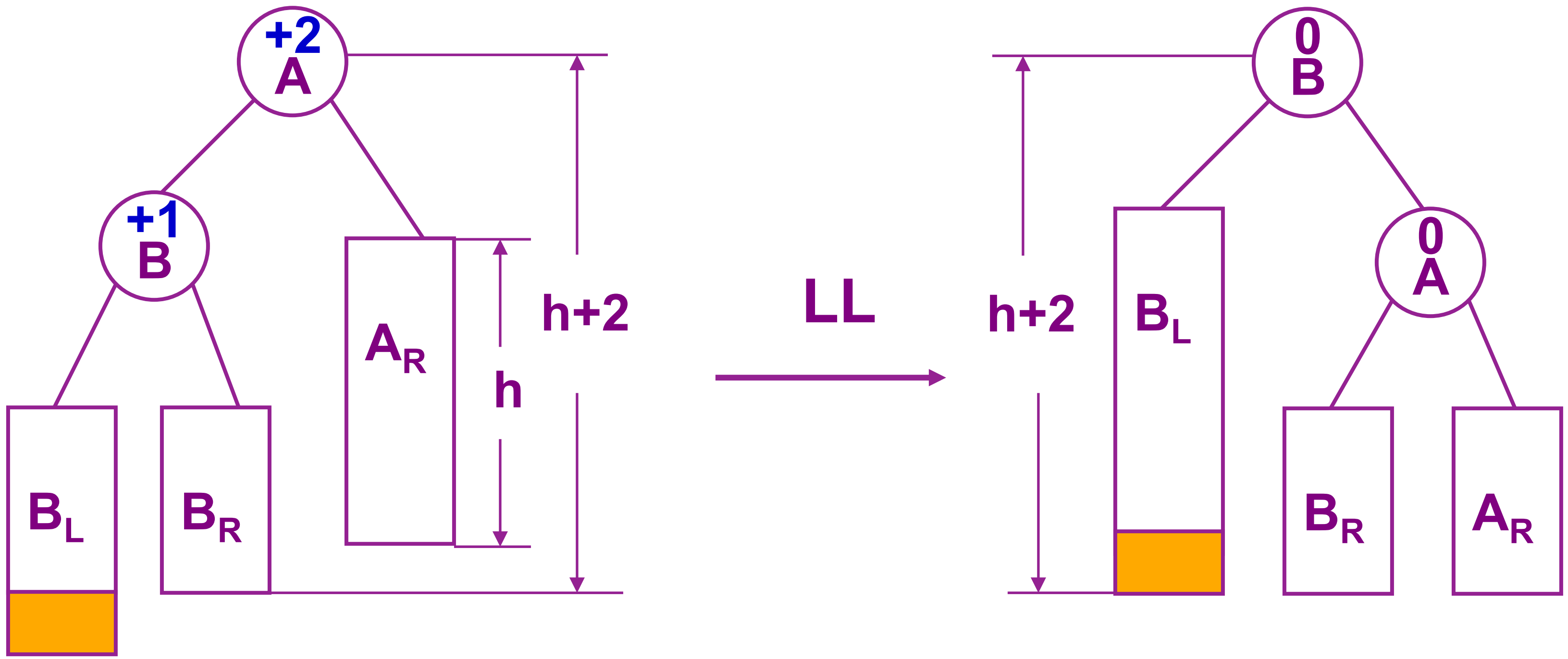
- // phase 2: Insert and rebalance. k is not in the tree
- // will be inserted as the appropriate child of pp.
- AvlNode<K, E> *y=new AvlNode<K, E>(k, e);
- if (k<pp→key)
- pp→leftChild=y; // as left child
- else
- pp→rightChild=y; // as right child

- // Adjust BF's of nodes on path from a to pp.
- // d=+1 implies k is inserted in the left subtree of
- // a and d=-1 in the right.
- // The BF of a will be changed later.
- **int** d;
- AvlNode<k, E> *b, // child of a
- *c; // child of b
- **if** (k>a→key)
- { b=p=a→rightChild; d=-1;}
- **else**
- { b=p=a→leftChild; d=1;}

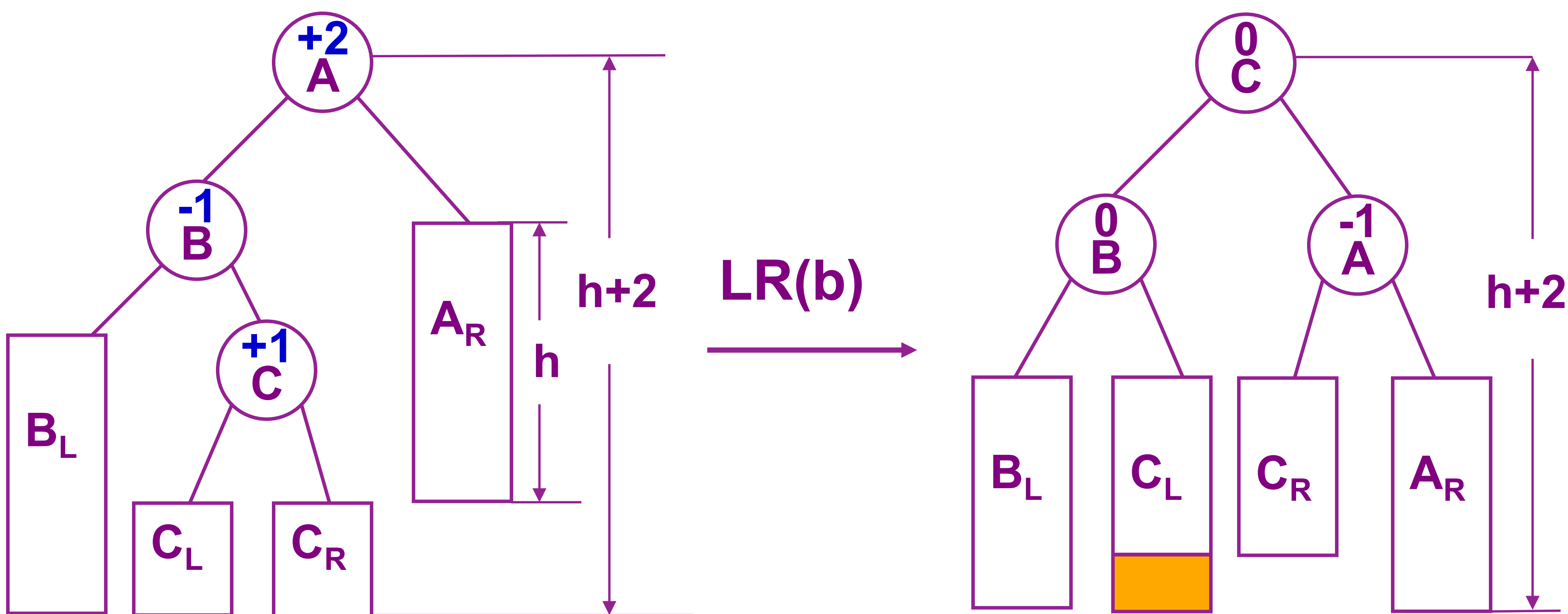
- **while** (p!=y)
- **if** (k>p→key) **{** // height of right increases by 1
- p→bf= -1**;**
- p=p→rightChild**;**
- **}**
- **else {** // height of left increases by 1
- p→bf= 1**;**
- p=p→leftChild**;**
- **}**

- // Is tree unbalanced?
- **if** (!(a→bf) || !(a→bf +d)) {
- // tree still balanced
- a→bf +=d**; return;**
- }
- //tree unbalanced, determine rotation type
- **if** (d==1) { // left imbalance

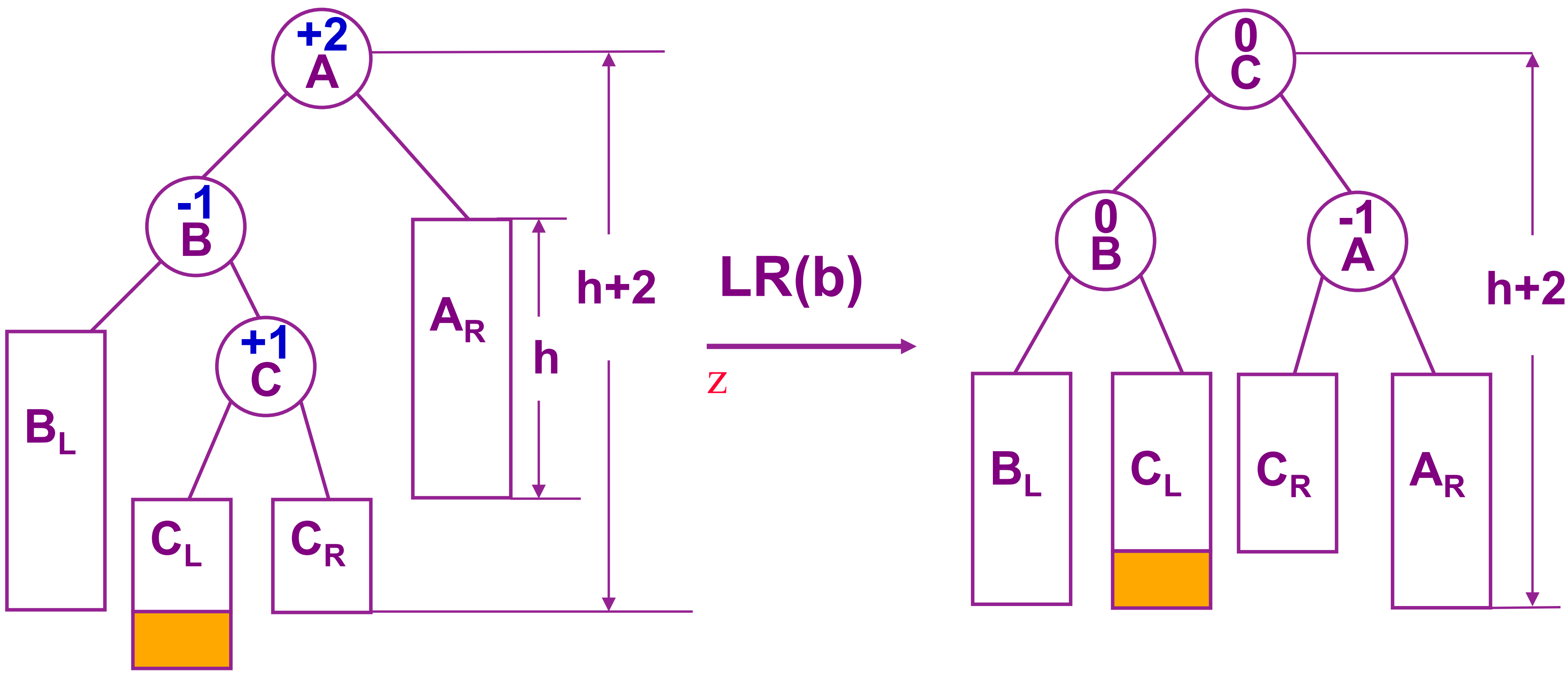
- **if** (b→bf==1) { // type LL
- a→leftChild=b→rightChild;
- b→rightChild=a;
- a→bf=0; b→bf=0;
- }



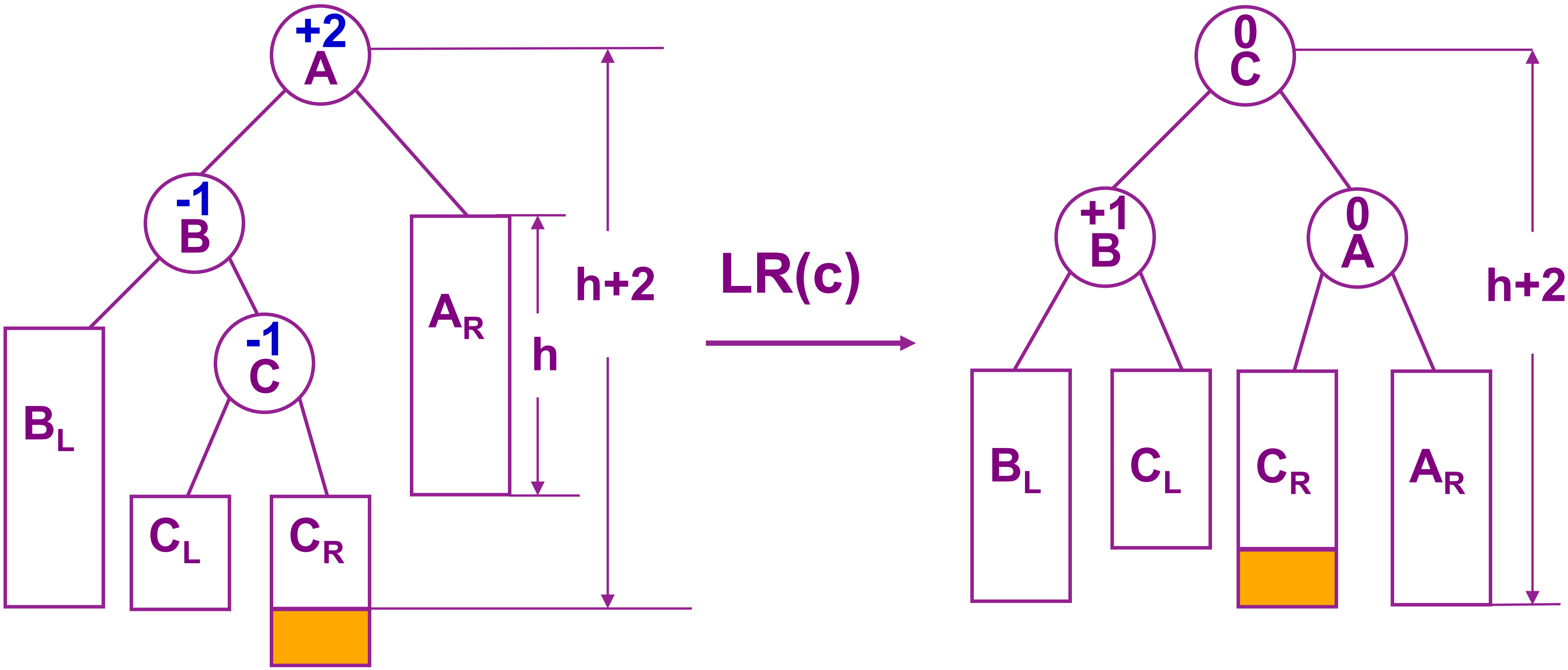
- **else** {   // type LR
-         c=b→rightChild;
-         b→rightChild=c→leftChild;
-         a→leftChild=c→rightChild;
-         c→leftChild=b;
-         c→rightChild=a;



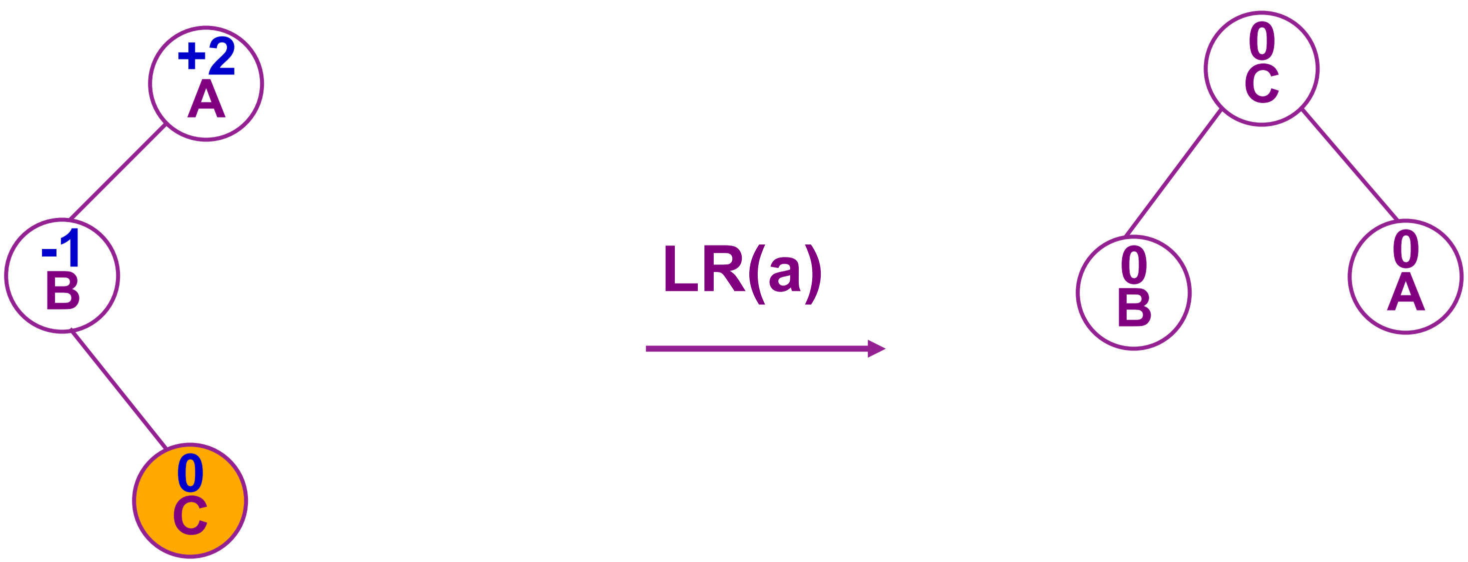
- **switch** (c→bf) {
- **case** 1: // LR(b)
- a→bf=-1; b→bf=0;
- **break;**



- **case** -1: // LR(c)
- b→bf=1; a→bf=0;
- **break;**



- **case** 0: // LR(a)
- b→bf=0; a→bf=0;
- **break;**
- }
- c→bf=0; b=c; // b is the new root
- } // end of LR
- } // end of left imbalance



- **else** { // right imbalance
- // symmetric to left imbalance
- }
- // Subtree with root b has been rebalanced.
- **if** (!pa)
- root=b**;** // A has no parent and a is the root
- **else if** (a==pa→leftChild)
- pa→leftChild=b**;**
- **else** pa→rightChild=b**;**
- **return;**
- } // end of AVL::Insert

Analysis

- If h is the height of the tree before insertion, the time to insert a new key is O(h).
- In case of AVL tree, h can be at most O(log n), so the insertion time is O(log n).

Exercises: P578-3, 5, 9