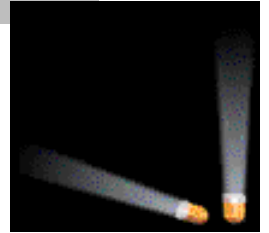
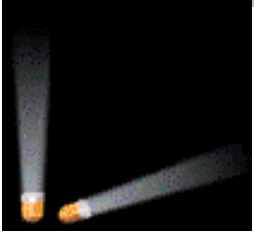


Efficient Binary Search Trees



- 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

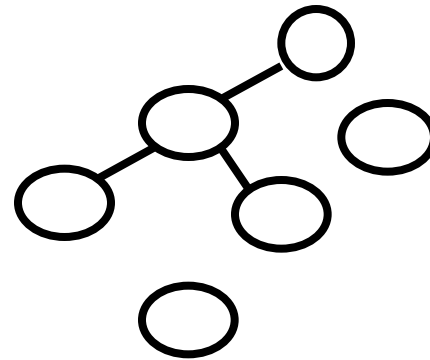
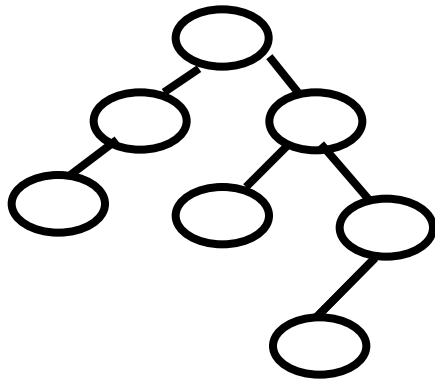
AVL Tree

- 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) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively

Balance Factor

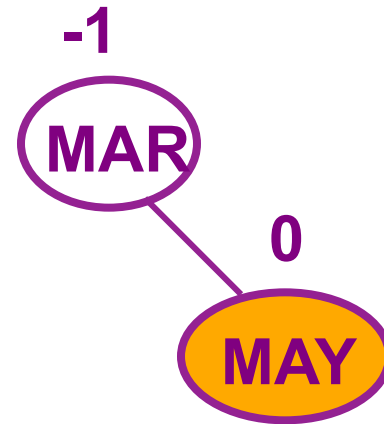
- The balance factor, $BF(T)$, of a node T in a binary tree
 - $h_L - h_R$
- For any node T in an AVL tree
 - $BF(T) = -1, 0, \text{ or } 1$.♪

AVL Tree?

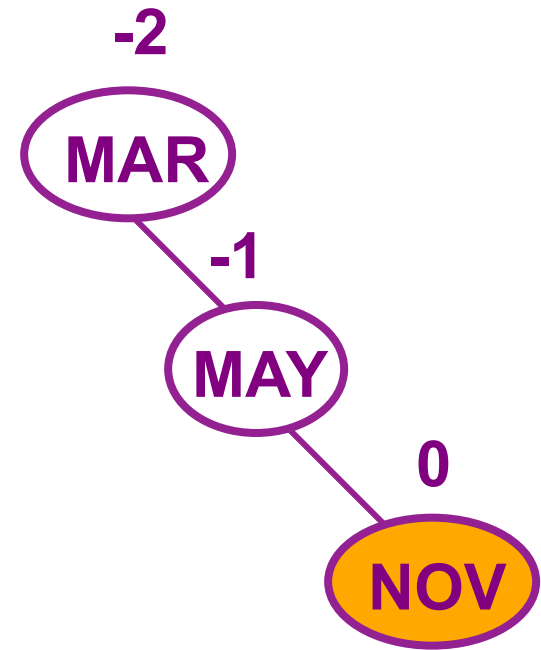
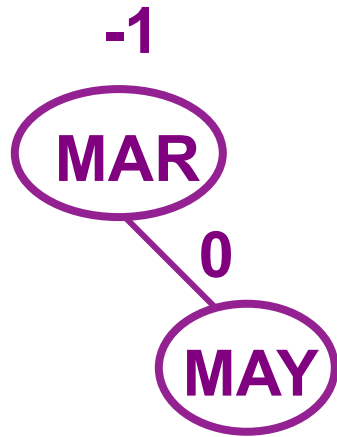




(a) Insert MAR

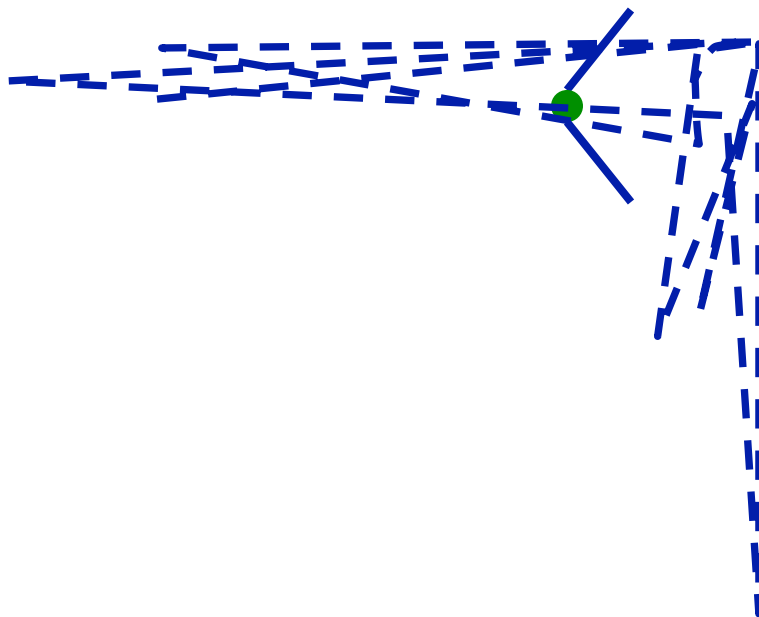


(b) Insert MAY



(c) Insert NOV

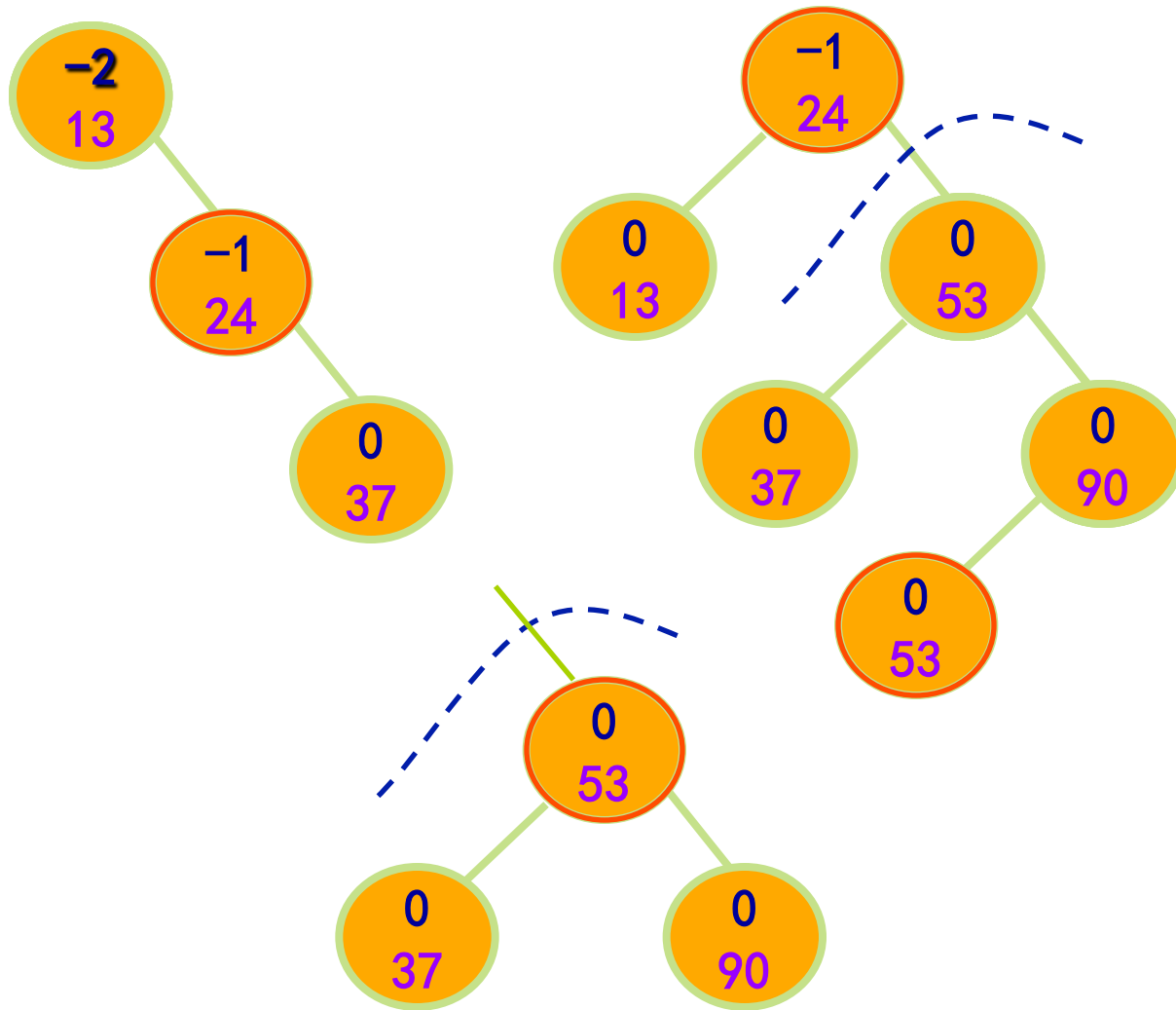
- Insertion may leads to unbalancing !
- Rebalance it!

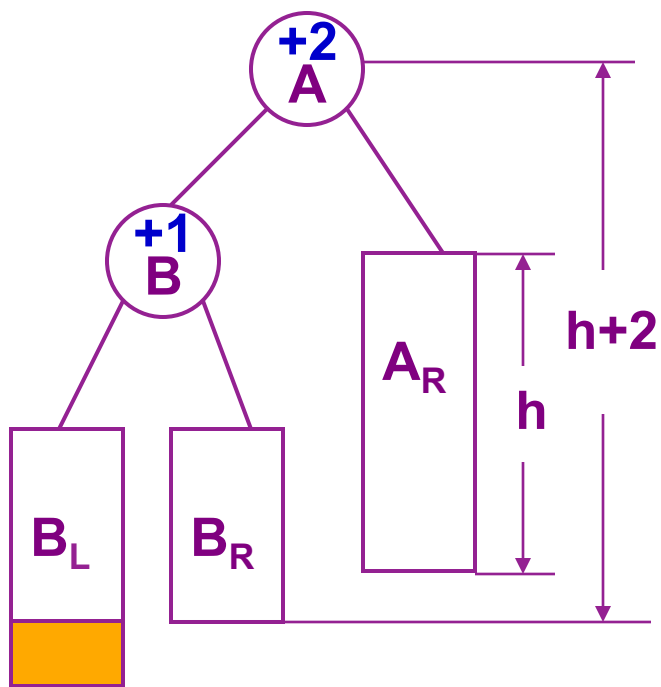


Build an AVL Tree

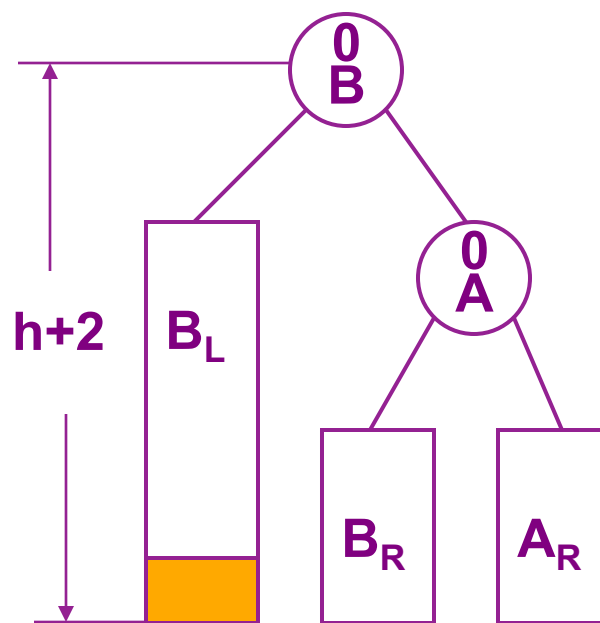
(13, 24, 37, 90, 53)

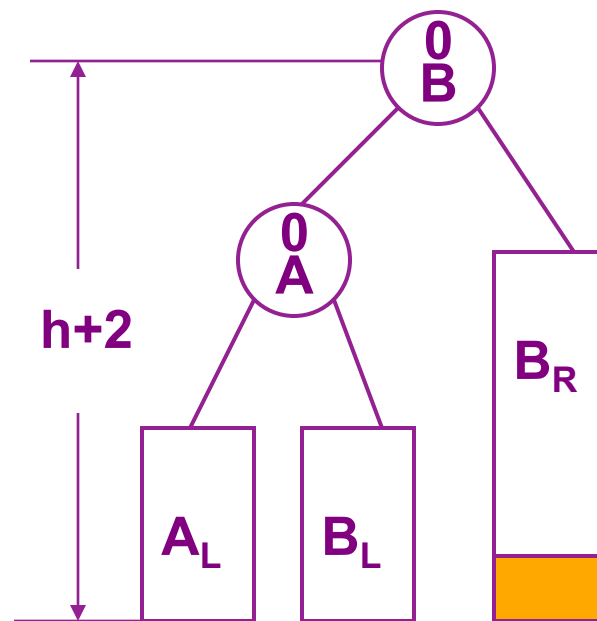
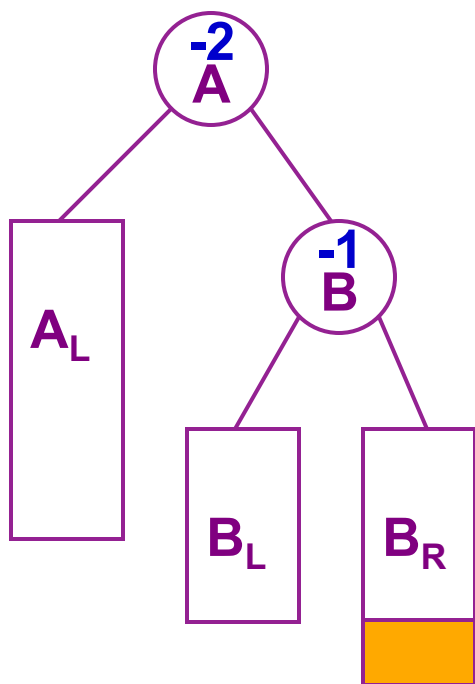
↓ ↓ ↓ ↓ ↓

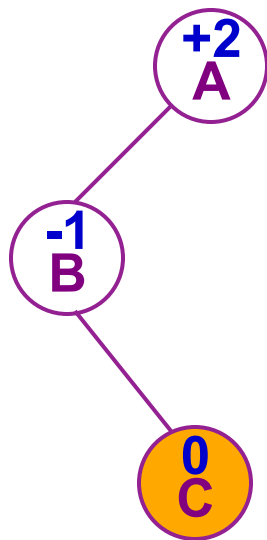




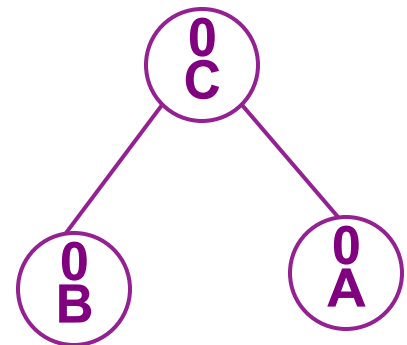
LL

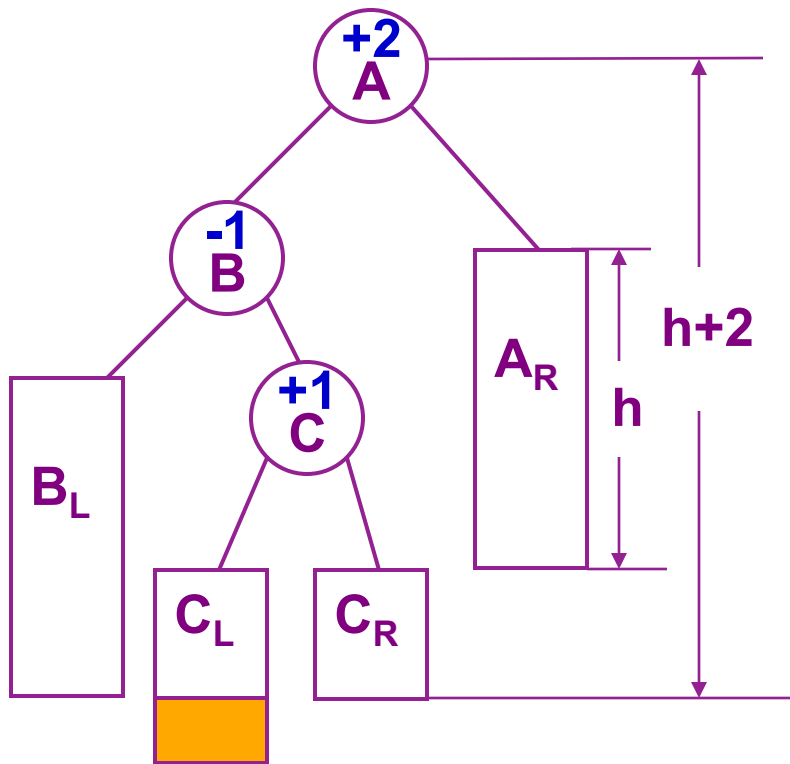




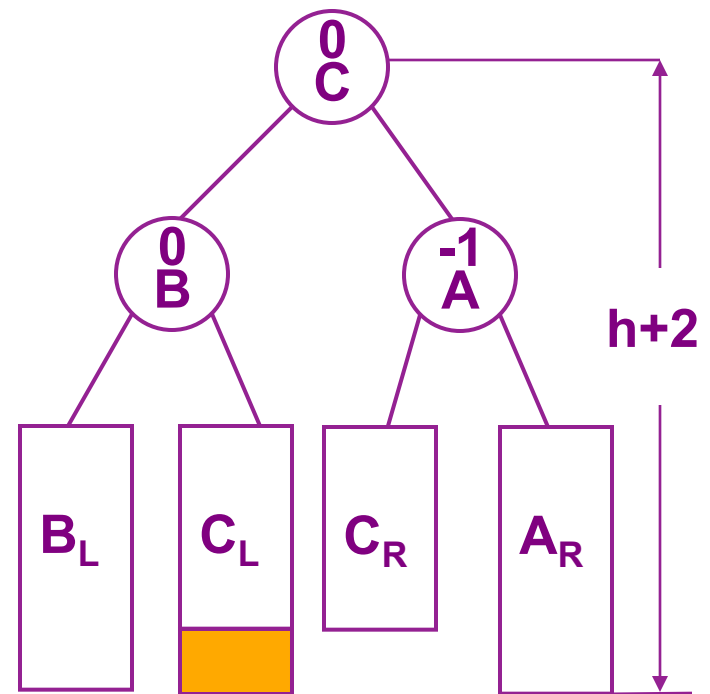


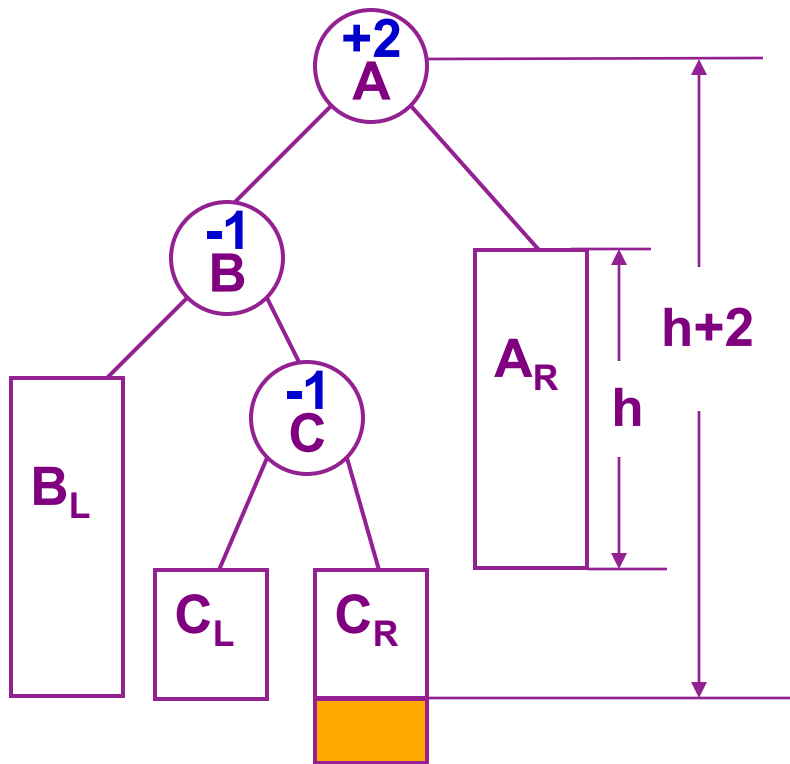
LR(a)



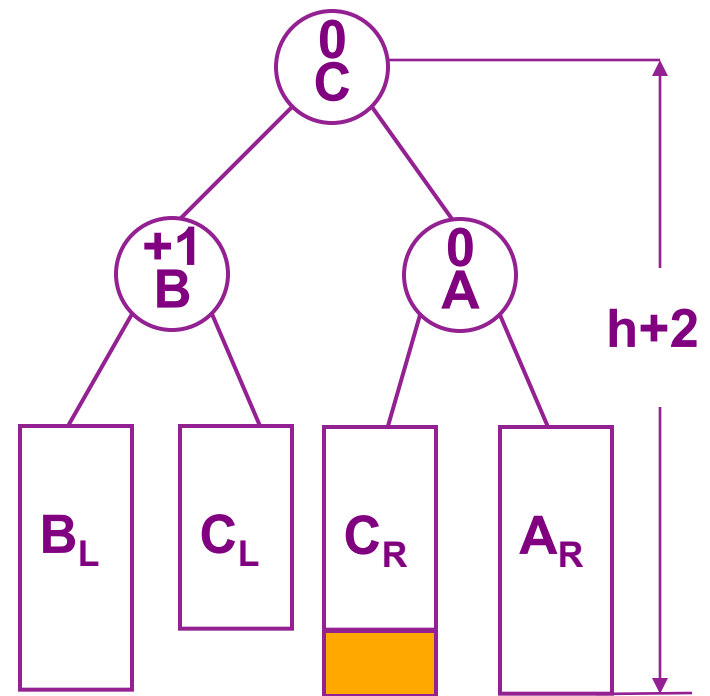


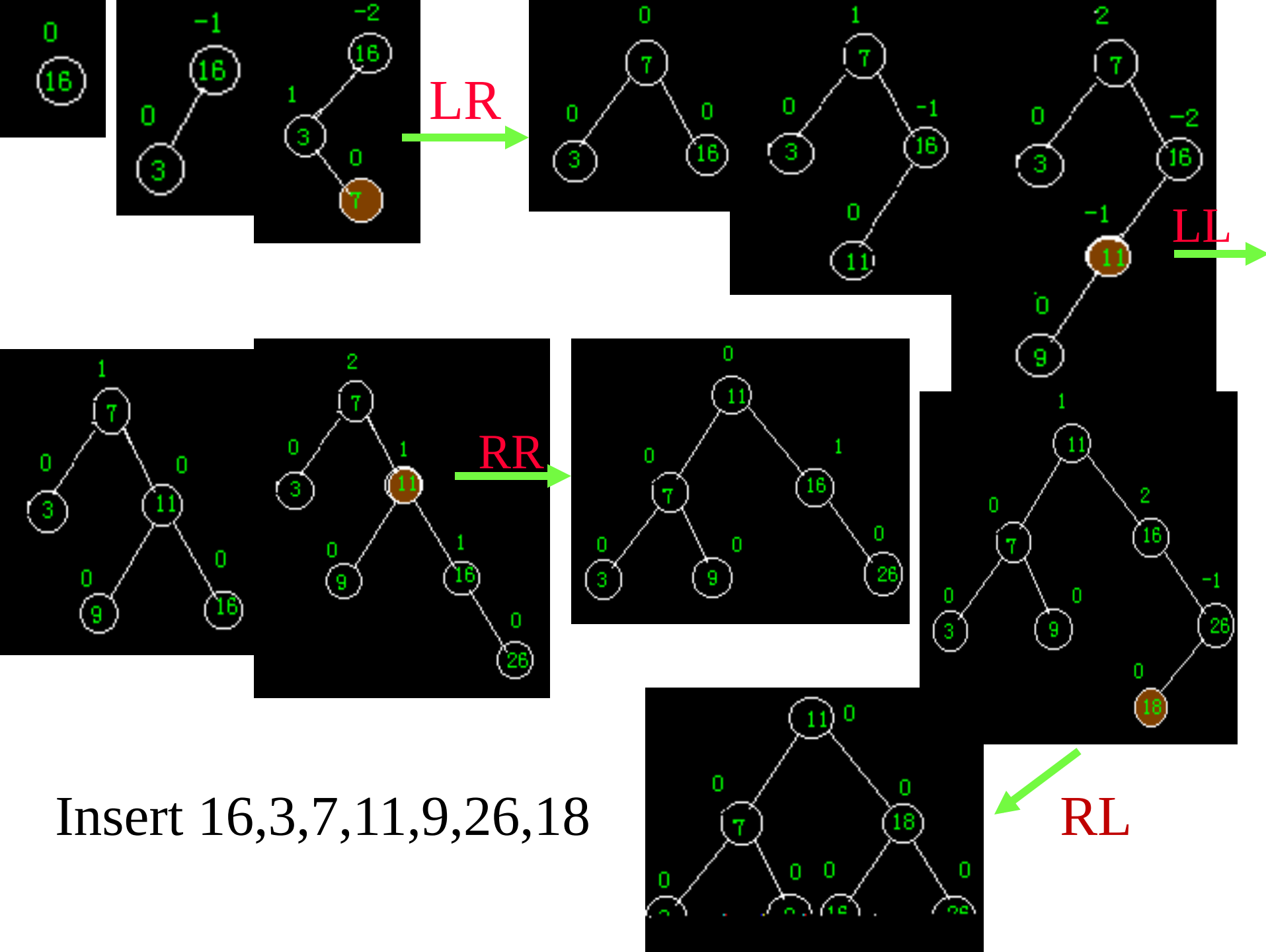
LR(b)





LR(c)





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

- 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 $BF = \pm 1$ 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

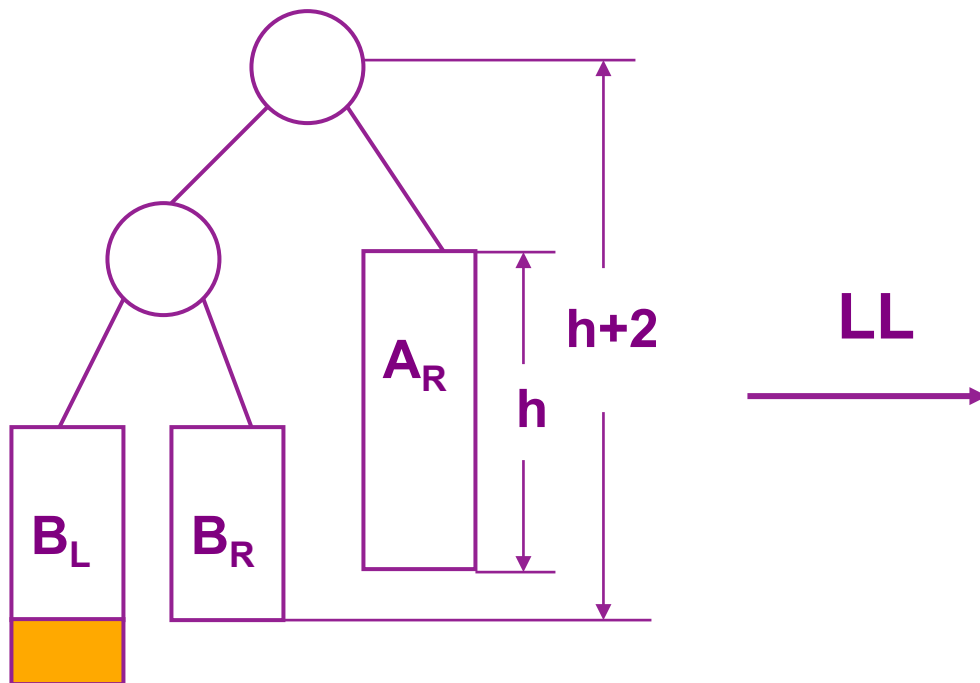
- // phase 2: Insert and rebalance. k is not in the tree
- // will be inserted as the appropriate child of pp.
- `AvlNode<K, E> *y=new AvlNode<K, E>(k,`
 `e);`
- `if (k<pp→key)`
- `pp→leftChild=y;     // as left child`
- `else`
- `pp→rightChild=y;    // as right child`

- // 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

- **while** ($p \neq y$)
- **if** ($k > p \rightarrow \text{key}$) { // height of right increases by 1
- $p \rightarrow \text{bf} = -1;$
- $p = p \rightarrow \text{rightChild};$
- }
- **else** { // height of left increases by 1
- $p \rightarrow \text{bf} = 1;$
- $p = p \rightarrow \text{leftChild};$
- }

- // Is tree unbalanced?
- **if** ($!(a \rightarrow bf) \parallel !(a \rightarrow bf + d)$)

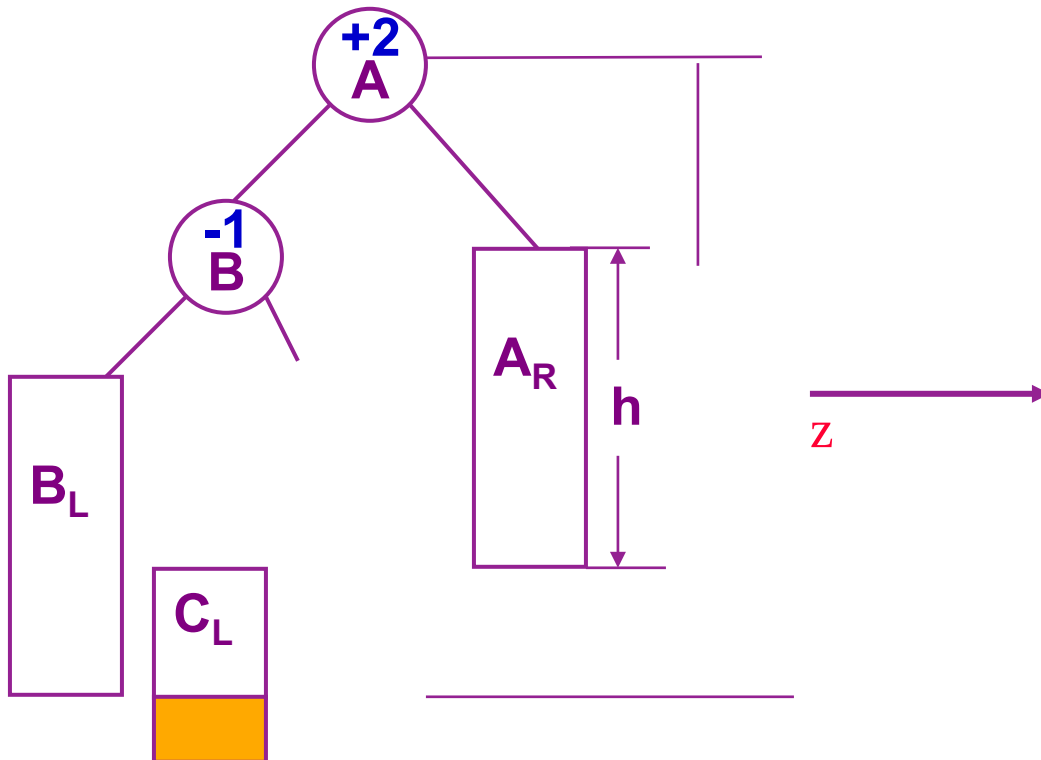
- **if** ($b \rightarrow bf == 1$) { // type LL
- $a \rightarrow \text{leftChild} = b \rightarrow \text{rightChild};$
- $b \rightarrow \text{rightChild} = a;$
- $a \rightarrow bf = 0; b \rightarrow bf = 0;$
- }



0
B

- **else** { // type LR
- $c = b \rightarrow \text{rightChild};$
- $b \rightarrow \text{rightChild} = c \rightarrow \text{leftChild};$
- $a \rightarrow \text{leftChild} = c \rightarrow \text{rightChild};$
-

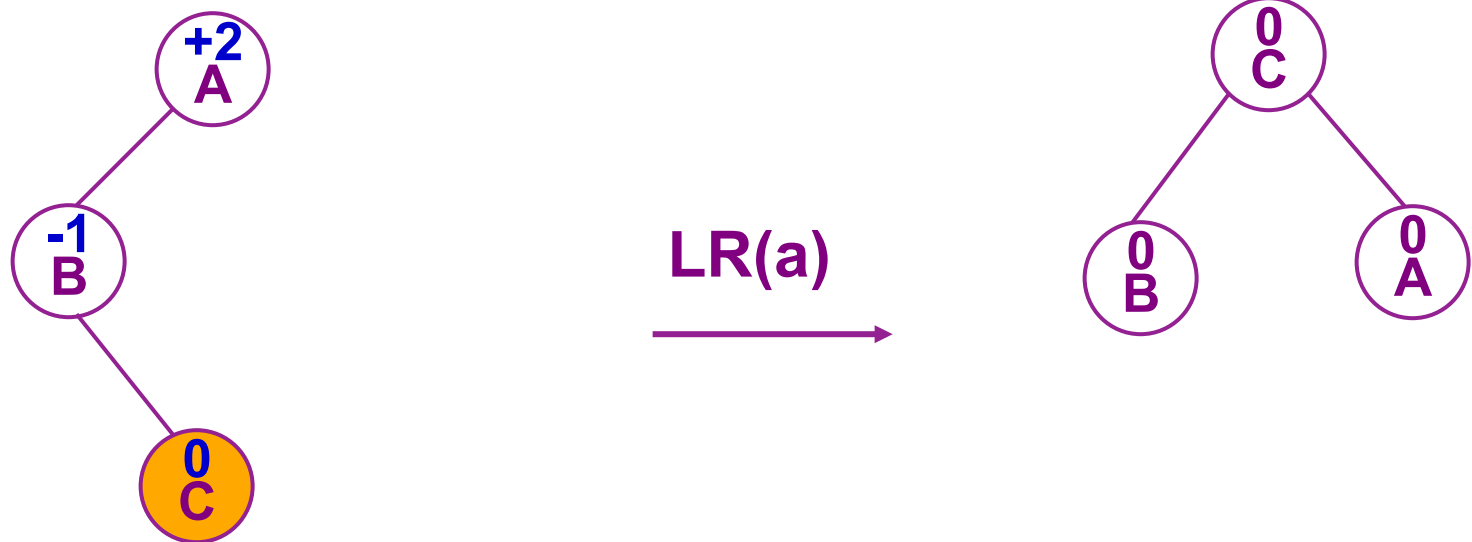
- **switch** ($c \rightarrow bf$) {
- **case** 1: // LR(b)
- $a \rightarrow bf = -1$; $b \rightarrow bf = 0$;
- **break**;



-

case

- **case 0: // LR(a)**
- $b \rightarrow bf=0; a \rightarrow bf=0;$
- **break;**
- **}**
- $c \rightarrow bf=0; b=c; // b \text{ is the new root}$
- **} // 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