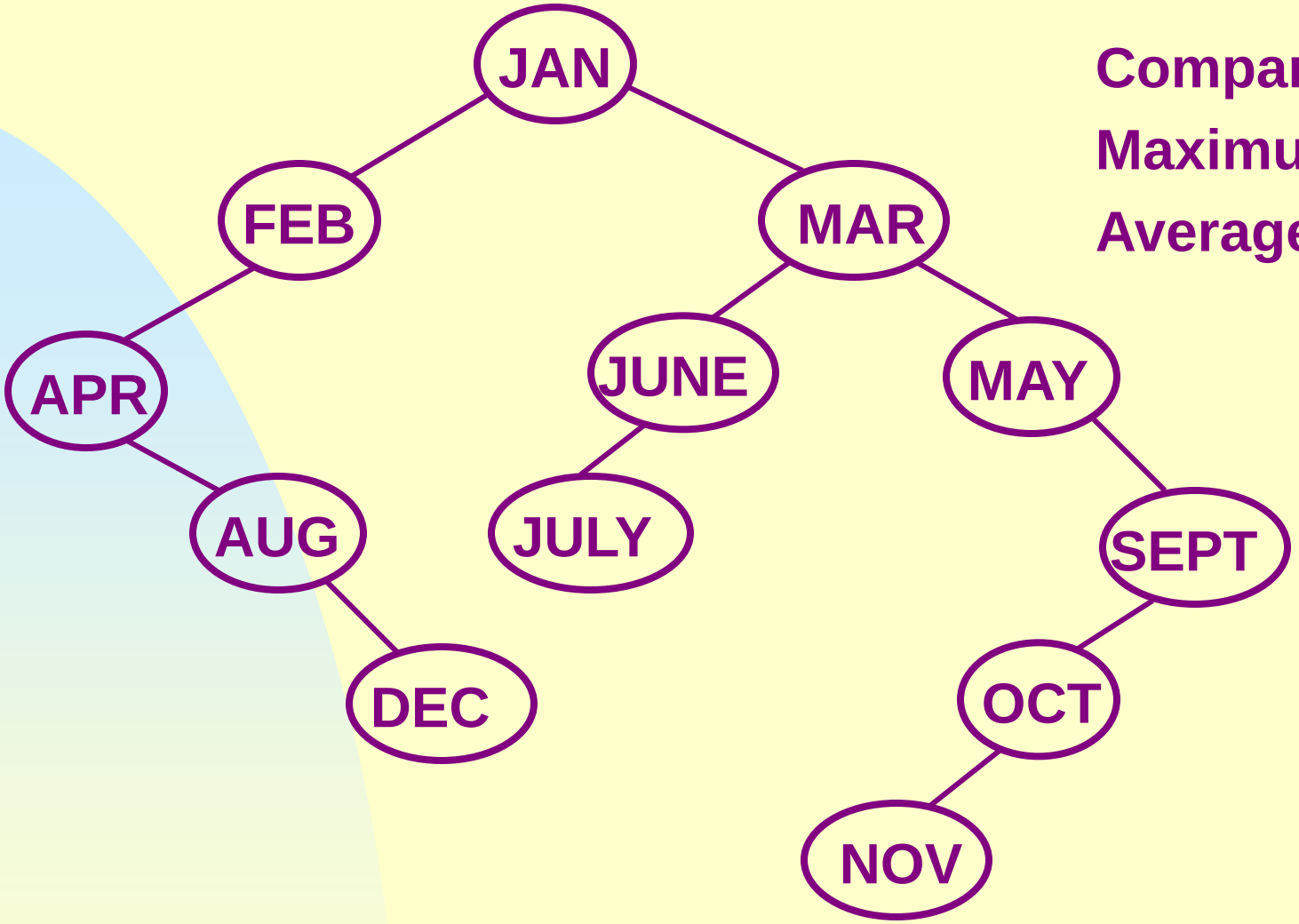
Chapter 10
Efficient Binary Search Trees

10.2 AVL Trees

Dynamic collections of elements may also be maintained as binary search trees.
Fig. 10.8 shows the binary search tree obtained by entering the months JAN to DEC into an initially empty binary search tree by using function Insert (Program 5.21).



Comparisons:

Maximum: 6

Average: 3.5

Fig. 10.8

Fig. 10.9 shows the binary search tree obtained by entering JULY, FEB, MAY, AUG, DEC, MAR, OCT, APR, JAN, JUNE, SEPT, NOV into an initially empty binary search tree.

Comparisons:

Maximum: 4

Average: 3.1

Balanced

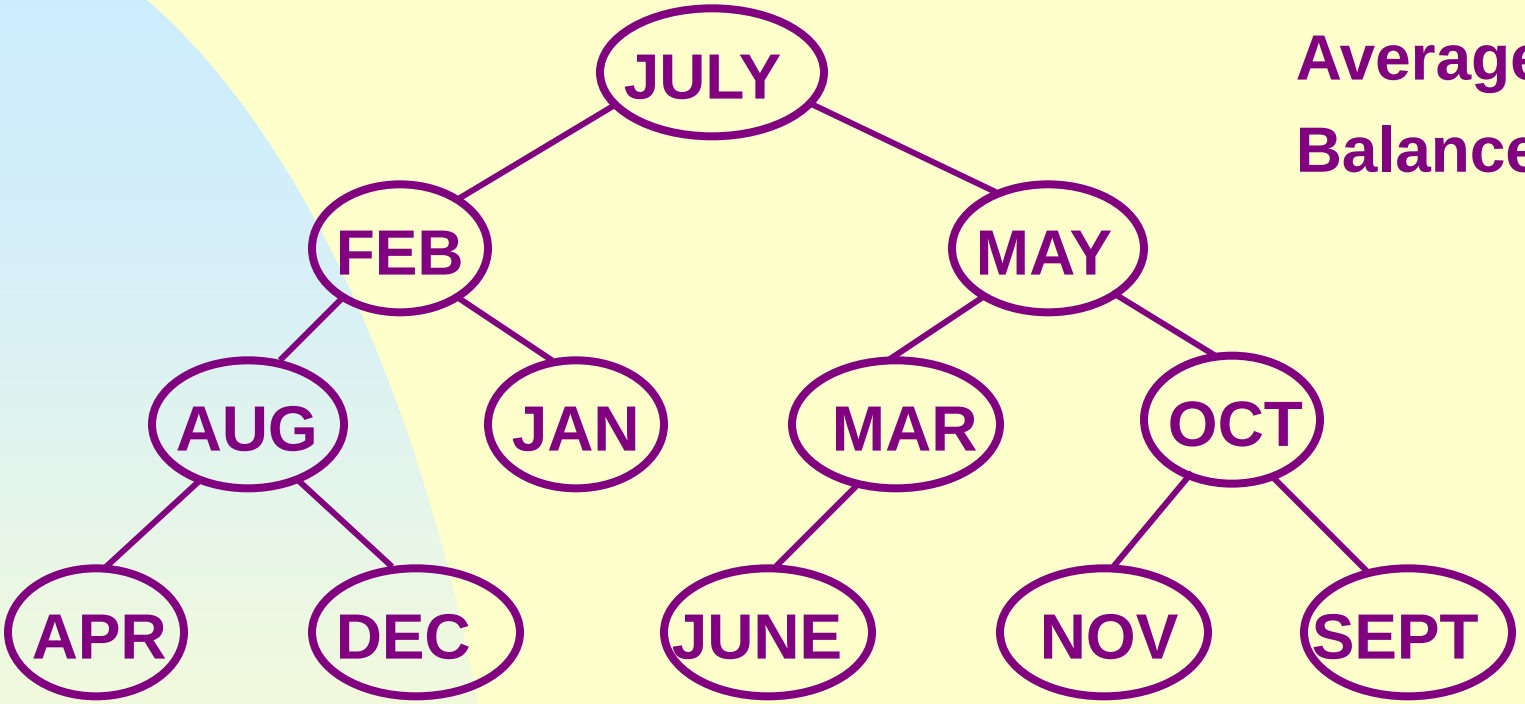
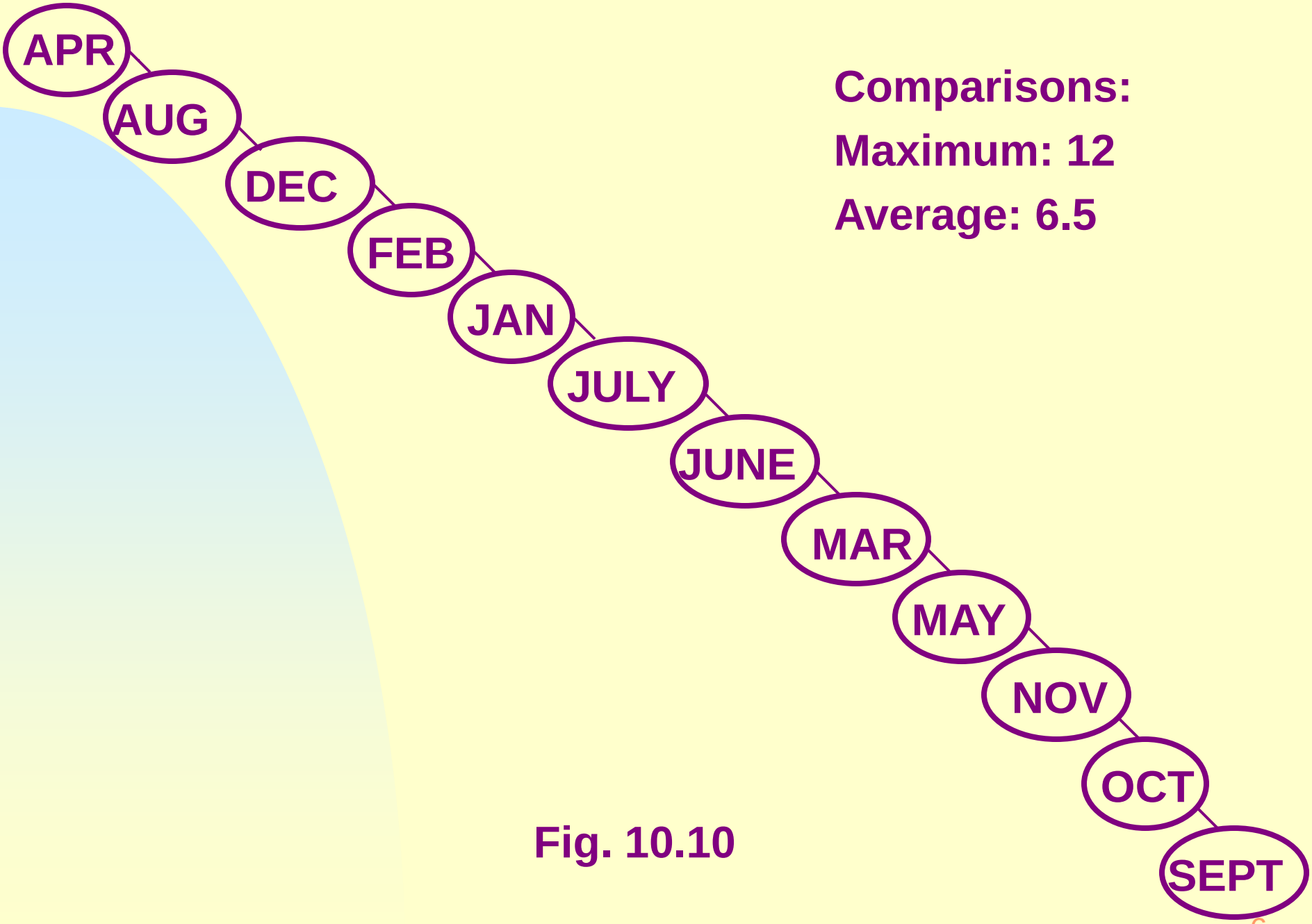


Fig. 10.9

Fig. 10.10 shows the binary search tree obtained by entering the months in lexicographic order into an initially empty binary search tree.



Comparisons:

Maximum: 12

Average: 6.5

Fig. 10.10

With equal search probabilities for keys, both the maximum and average search time will be minimized if the binary search tree is maintained as a complete binary tree at all times.

However, since the dynamic situation, it is difficult to achieve this without making the time required to insert a key very high.

It is, however, possible to keep the tree balanced to ensure both average and worst-case retrieval time of O(log n) for a tree with n nodes.

AVL tree (introduced by Adelson-Velskii and Landis) is a binary search tree that is balanced with respect to the heights of subtrees.

Definition: 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.

Fig. 10.8 is not height-balanced.

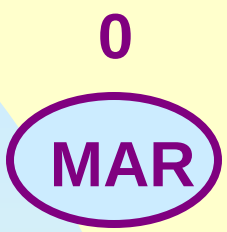
Fig. 10.9 is height-balanced.

Fig. 10.10 is not height-balanced.

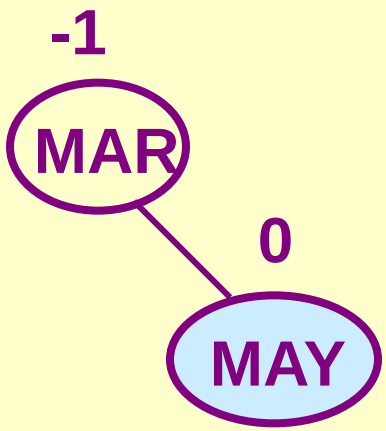
Definition: The balance factor, BF(T), of a node T in a binary tree is defined to be $h_L - h_R$. For any node T in an AVL tree BF(T)=-1, 0, or 1.

To illustrate the processes involved in maintaining an AVL tree, let's insert MAR, MAY, NOV, AUG, APR, JAN, DEC, JULY and FEB into an empty tree.

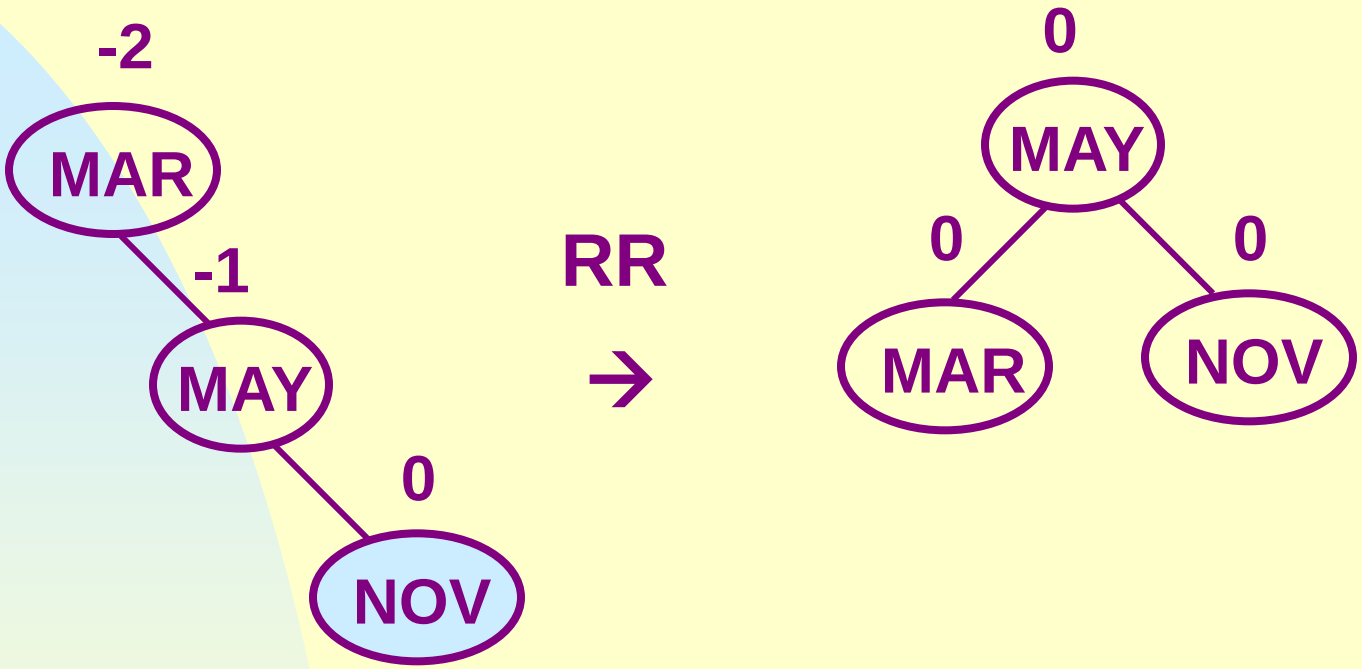
The following show the tree as it grows and restructuring involved in keeping it balanced. The numbers above each node represent the balance factor of that node.



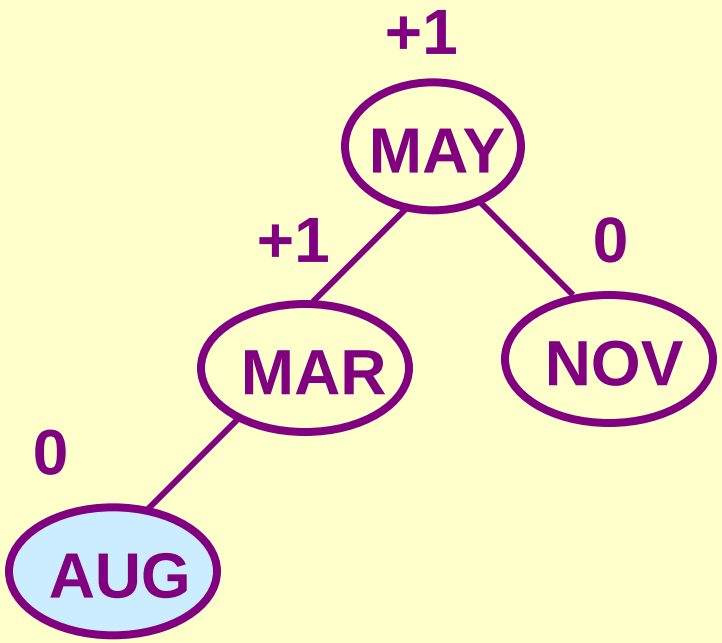
(a) Insert MAR



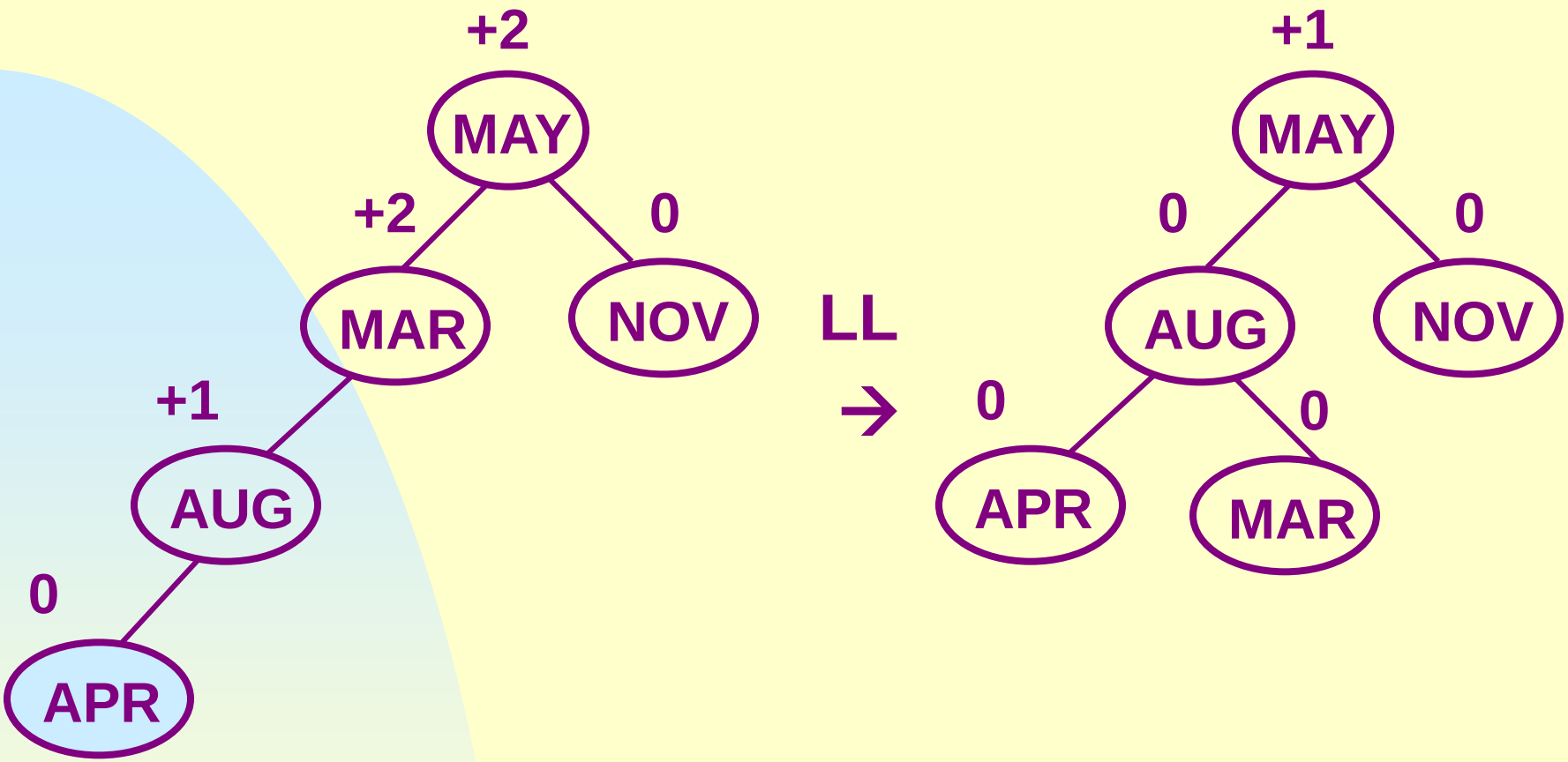
(b) Insert MAY



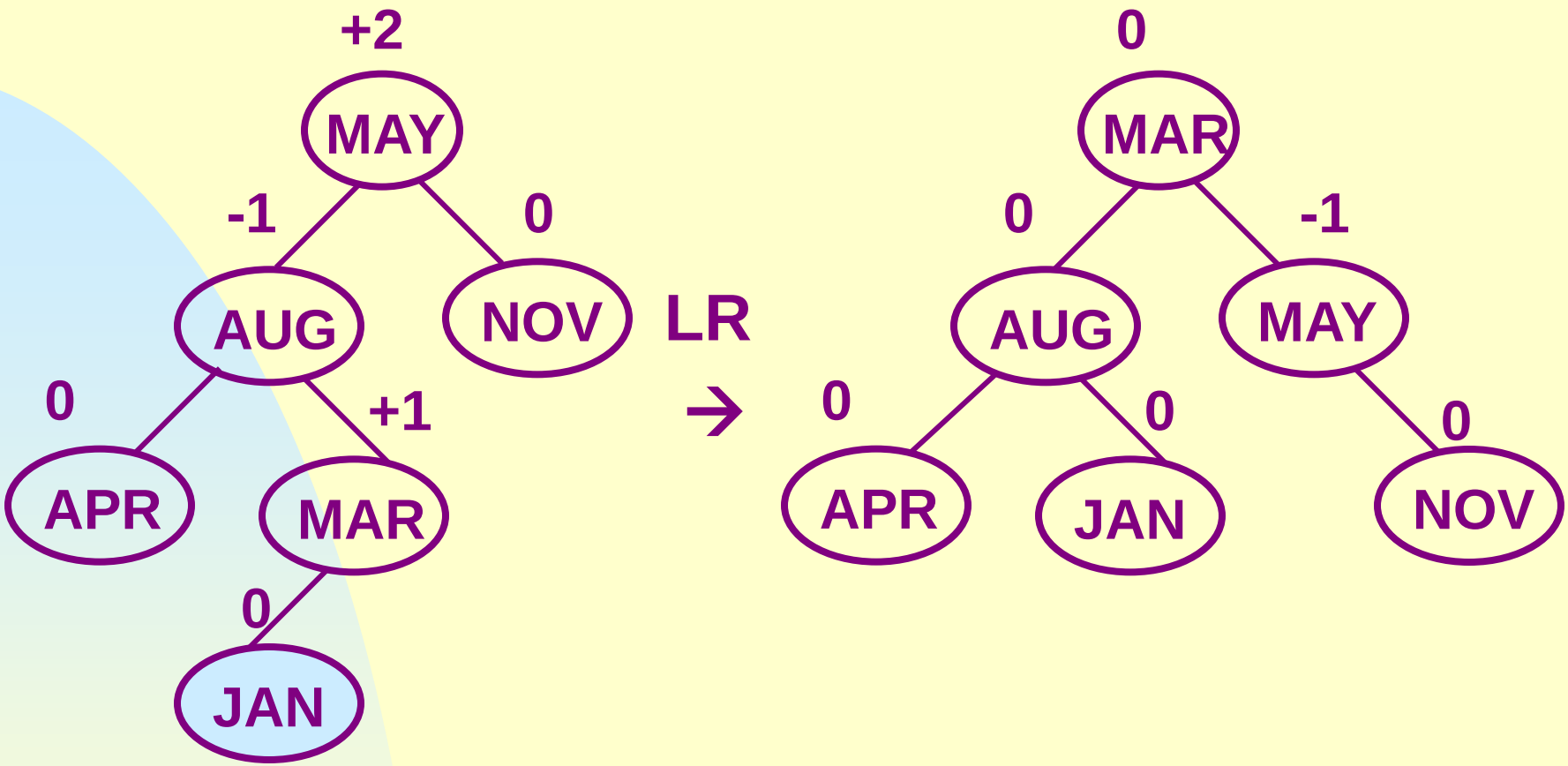
(c) Insert NOV



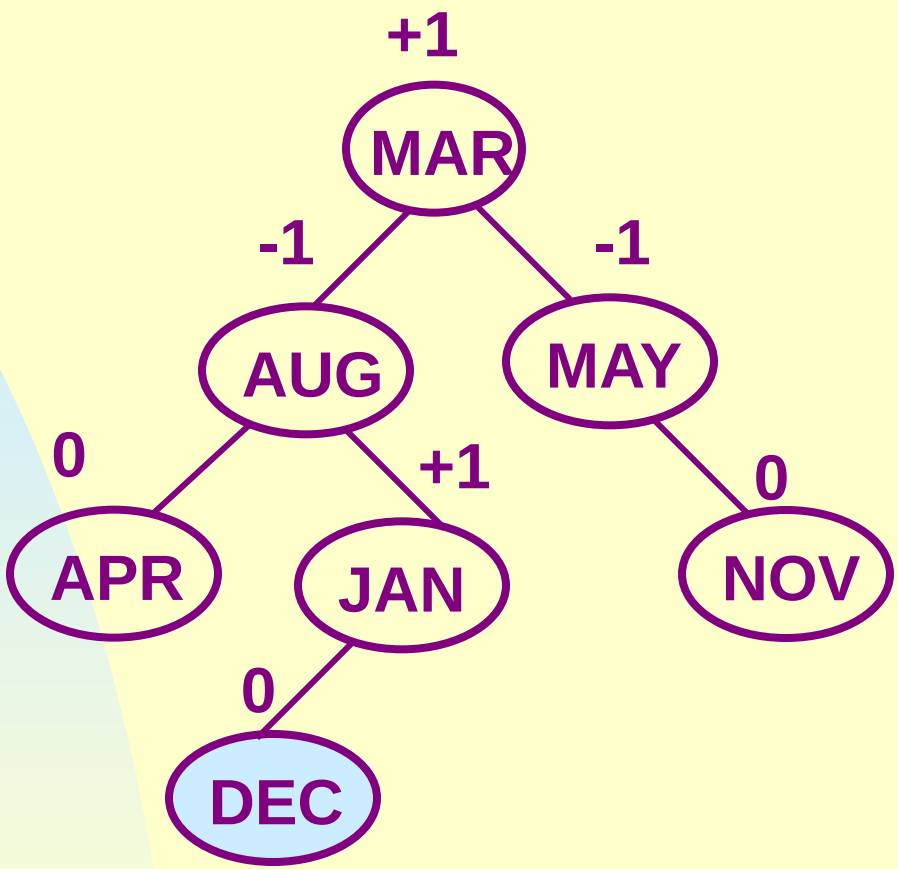
(d) Insert AUG



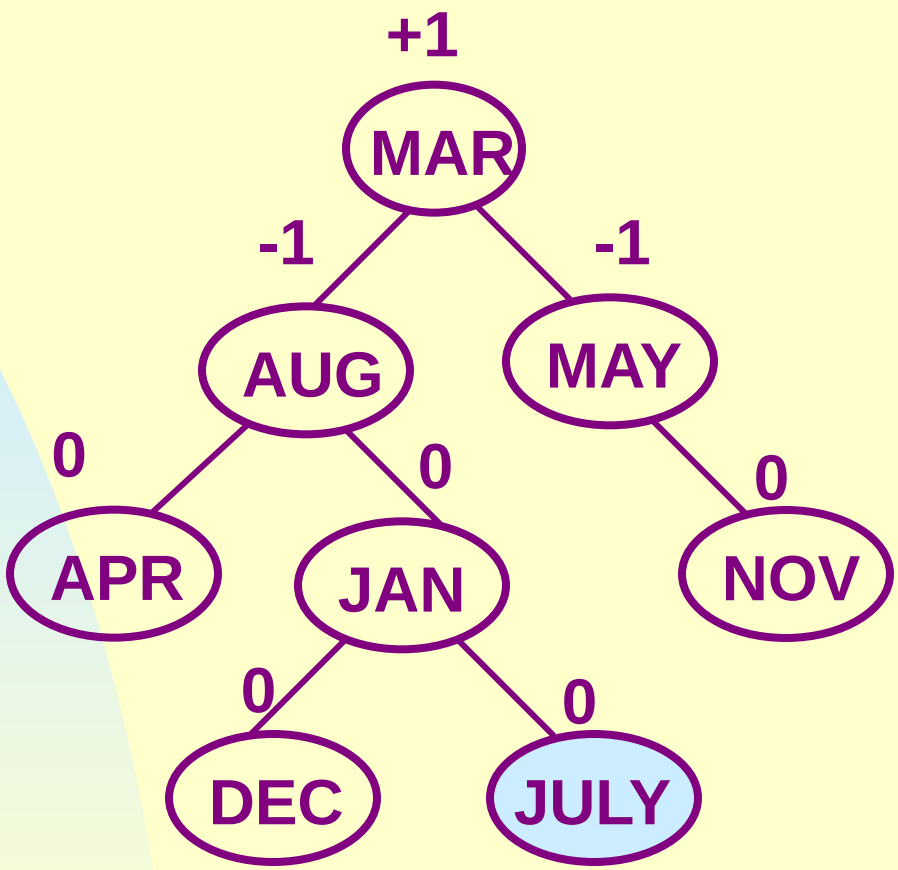
(e) Insert APR


+2
MAY
-1
0
AUG
NOV
0
+1
APR
MAR
0
JAN
LR
0
MAR
0
-1
AUG
MAY
0
0
0
APR
JAN
NOV

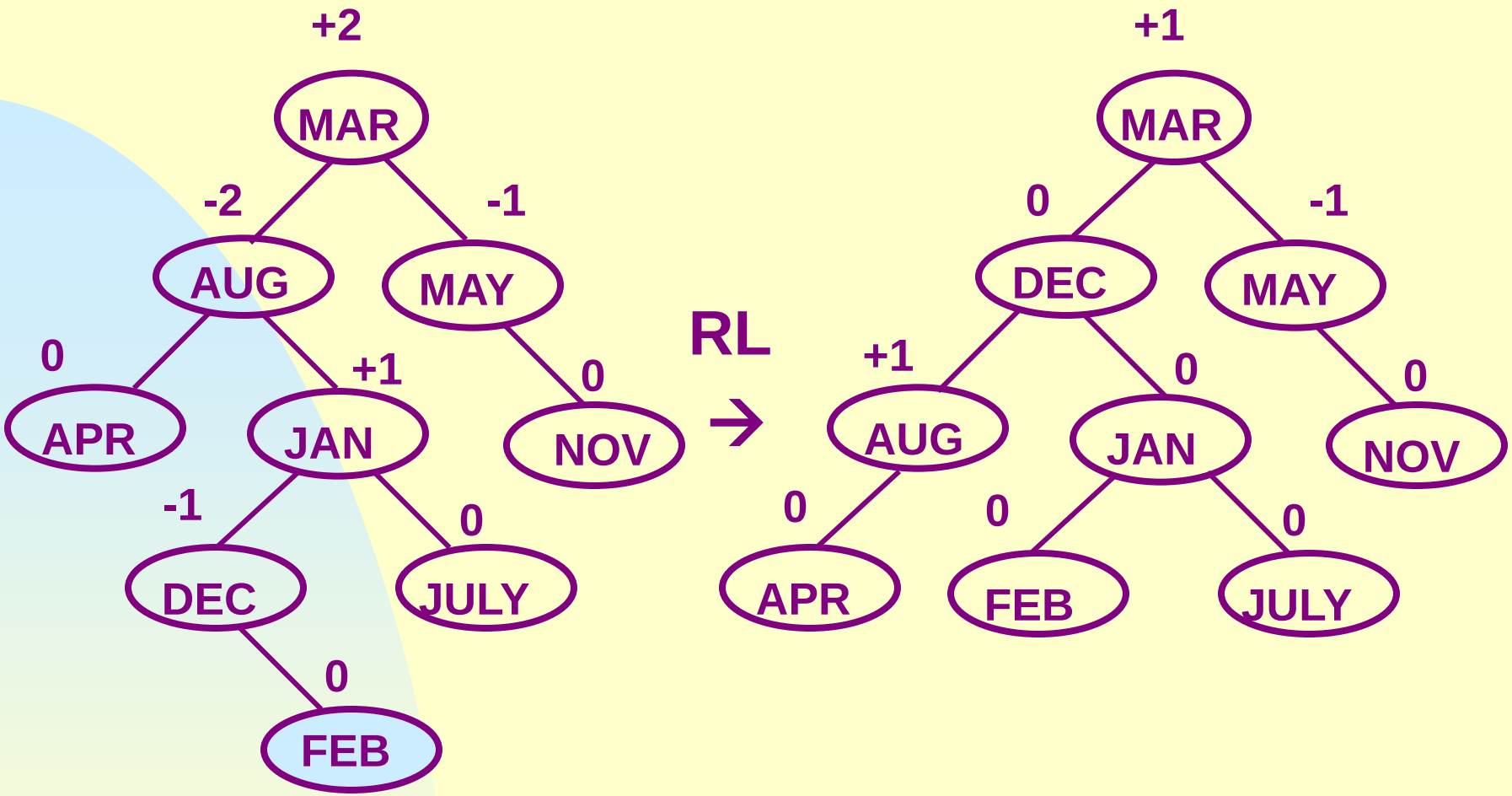
(f) Insert JAN



(g) Insert DEC



(h) Insert JULY


+2
MAR
-2
-1
AUG
MAY
0
+1
0
APR
JAN
NOV
-1
0
DEC
JULY
0
FEB
RL
+1
MAR
0
-1
DEC
MAY
+1
0
0
AUG
JAN
NOV
0
0
0
APR
FEB
JULY

(i) Insert FEB

In the proceeding example we saw that the addition of a node to a balanced binary search tree could unbalance it.

The rebalancing uses essentially 4 kinds of rotations:

LL, RR, LR, and RL.

LL and RR are symmetric, as are LR and RL.

The rotations are characterized by the nearest ancestor, A, whose BF becomes ± 2, of the inserted node Y.

LL: Y is inserted in the left subtree of the left subtree of A.

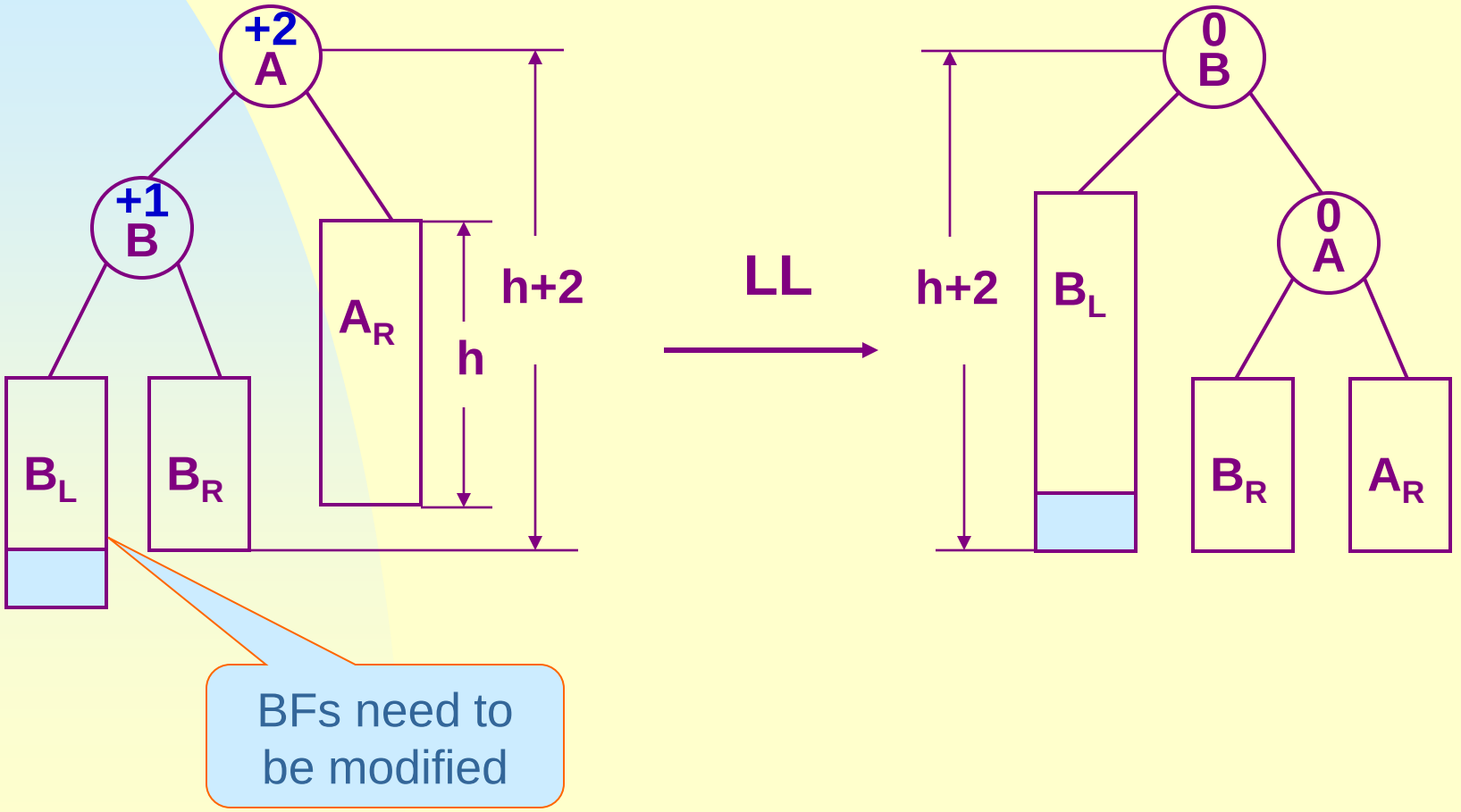
LR: Y is inserted in the right subtree of the left subtree of A.

RR: Y is inserted in the right subtree of the right subtree of A.

RL: Y is inserted in the left subtree of the right subtree of A.

As shown in the following:

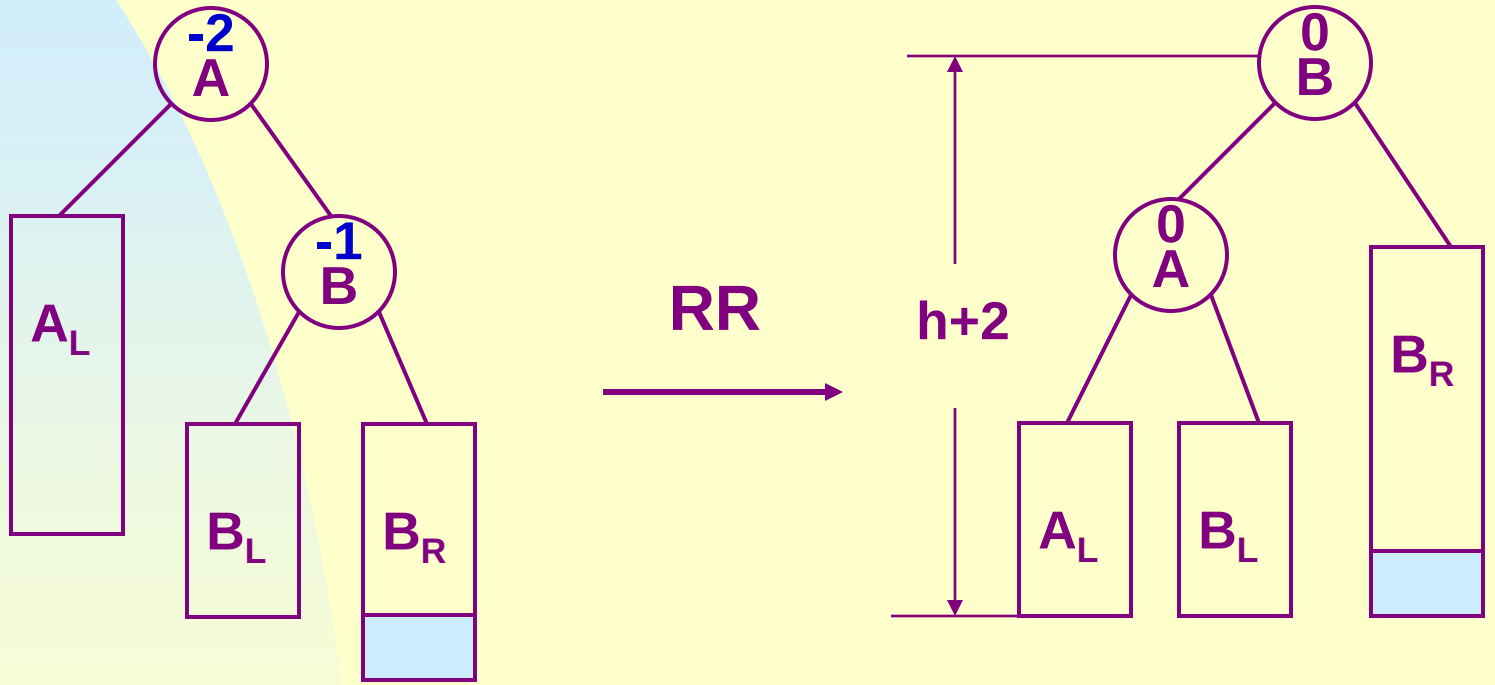
Unbalanced following insertion **rotation type** **Rebalanced**



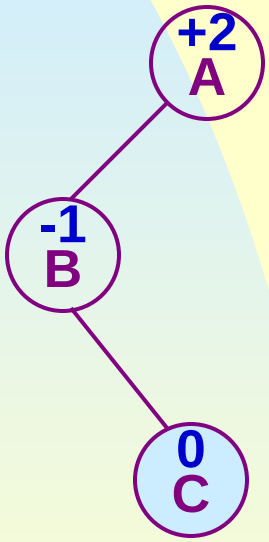
Unbalanced following insertion

rotation type

Rebalanced



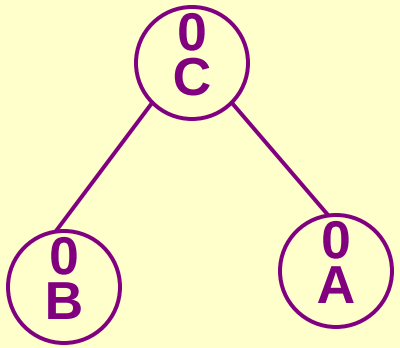
Unbalanced following insertion



rotation type

LR(a)

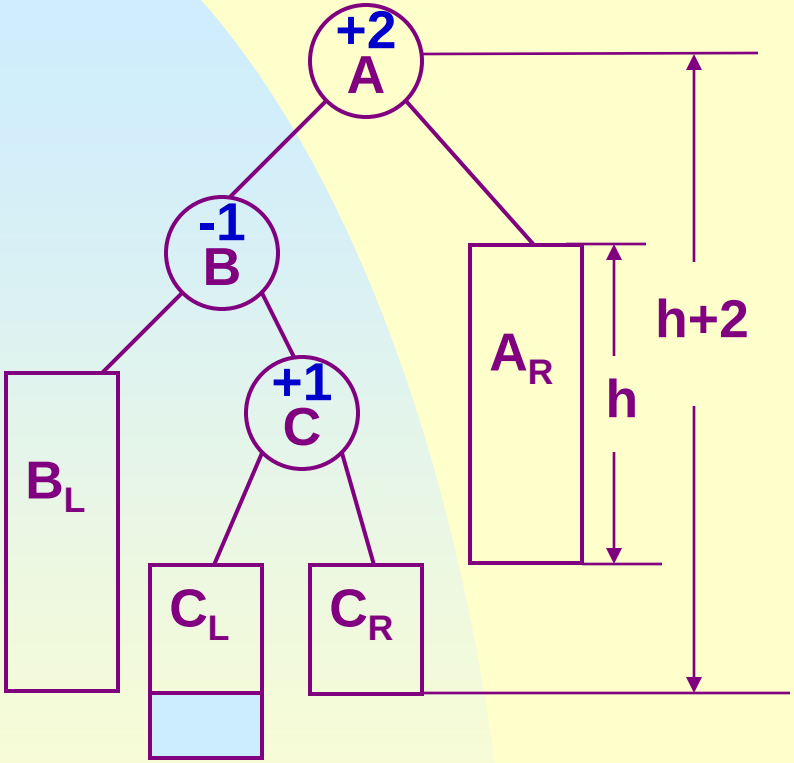
Rebalanced



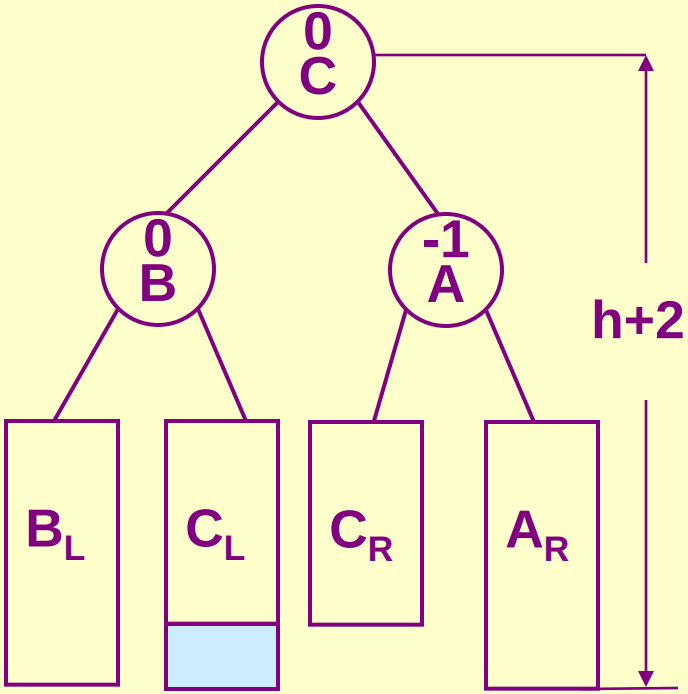
Unbalanced following insertion

rotation type

Rebalanced



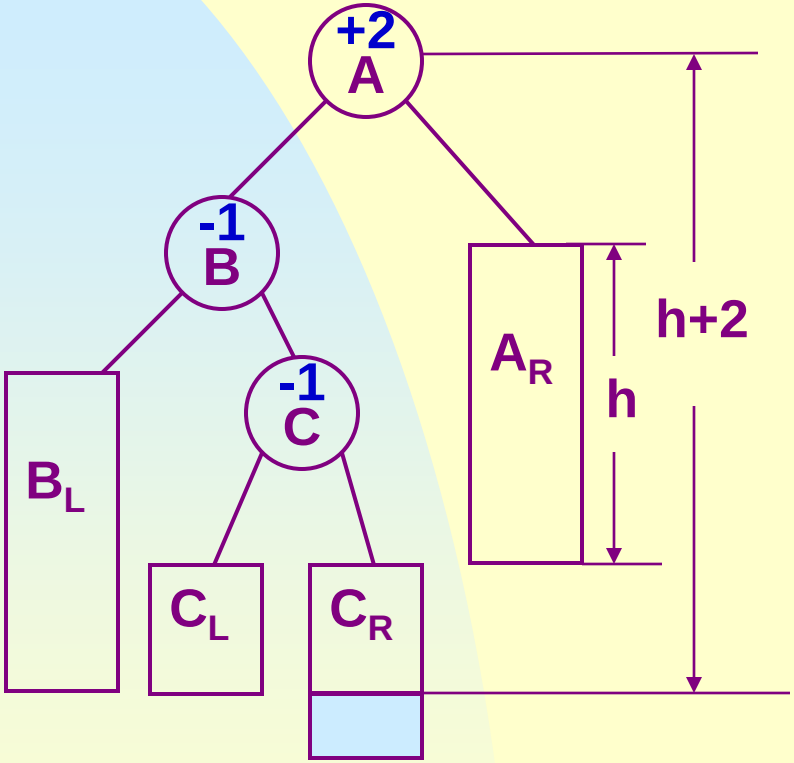
LR(b)



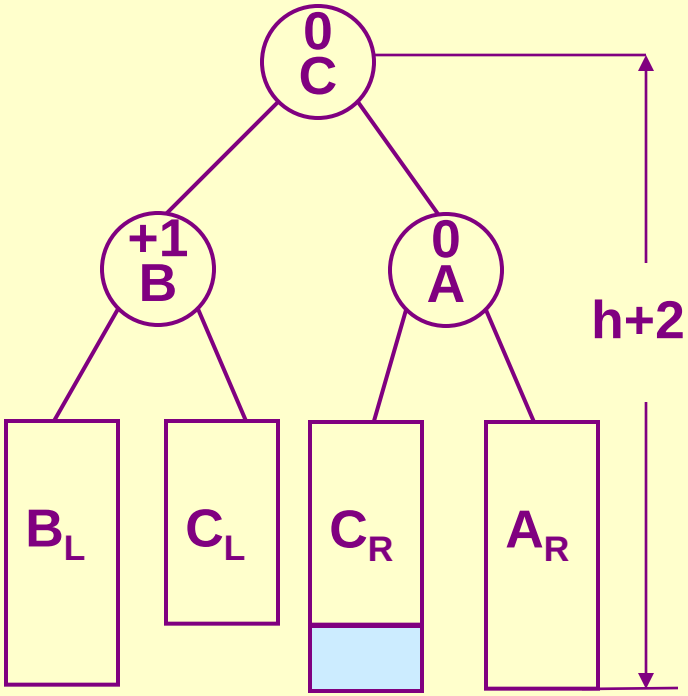
Unbalanced following insertion

rotation type

Rebalanced



LR(c)



LL, RR, LR, and RL are the only 4 cases possible for rebalancing.

LL and RR --- single rotations,

LR and RL --- double rotations.

For example, LR can be viewed as an RR followed by an LL rotation.

Note that the height of the subtree involved in the rotation is the same after rebalancing as it was before the insertion.

This means that once the rebalancing has been done on the subtree in question, examining the remaining tree is unnecessary.

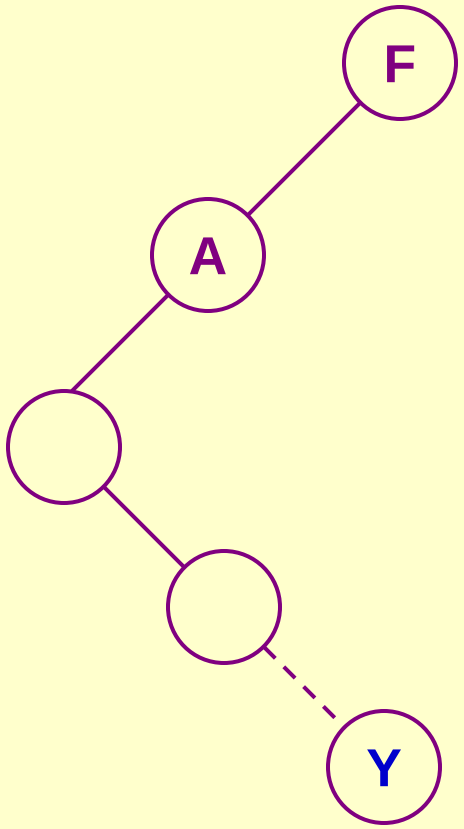
The only nodes whose BF can change are those in the subtree that is rotated.

To locate A, note it is the nearest ancestor of Y, whose BF becomes ±2, and it is also the nearest ancestor with BF= ±1 before the insertion.

Therefore, before the insertion, the BF's of all nodes on the path from A to the new insertion point must have been 0.

Thus, A is the nearest ancestor of the new node having a BF=±1 before insertion.

To complete the rotation, the parent of A, F is also needed.


F
A
Y

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.

AVL::Insert gives the details:

```
template <class K, class E> class AVL;

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 already in the tree
 } // end of while

// phase 2: Insert and rebalance. k is not in the tree and
// may 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
```

Computing time:

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).

To prove h=O(log n), let N_h be the minimum number of nodes in an AVL tree of height h, the heights of its subtrees are h-1 and h-2, and both are AVL trees. Hence

$N_h = N_{h-1} + N_{h-2} + 1$ (the root) and

$N_0 = 0$, $N_1 = 1$, $N_2 = 2$.

By induction on h, we can show

$N_h = F_{h+2} - 1$ **for** $h \geq 0$.

As the Fibonacci number

$F_{h+2} \geq \phi^h$, $\phi = (1+\sqrt{5})/2$

If n nodes in the tree, its height h is at most

$\log_\phi (n+1) = O(\log n)$

The exercises show that it is possible to find and delete a node with key k from an AVL tree in O(log n).

Exercises: P578-3, 5, 9

The next slide compares the worst-case times of certain operations on sorted sequential lists, sorted linked lists, and AVL trees.

Operation	Sequential list	Linked list	AVL tree
Search for k	O(log n)	O(n)	O(log n)
Search for jth item	O(1)	O(j)	O(log n)
Delete k	O(n)	O(1)[1]	O(log n)
Delete jth item	O(n-j)	O(j)	O(log n)
Insert	O(n)	O(1)[2]	O(log n)
Output in order	O(n)	O(n)	O(n)

1. Doubly linked list and position of k known
2. Position for insertion known