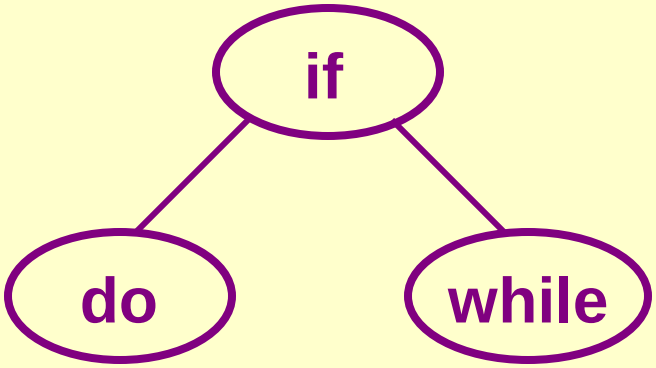
Chapter 10
Search Structures

10.1 Optimal Binary Search Trees

In this section, we consider the construction of binary search trees for a static set of identifiers. That is, we make neither additions nor deletions from the set. Only searches are performed.

Given n identifiers, we can construct a binary search tree equivalent to a **complete** binary search tree for them. Using the function *search* (Program 5.21), the worst search time is $\log_2 n$. If the identifiers are to be searched with equal probability, it is optimal.

For instance, the complete binary search tree for the file (do, if, while) is:

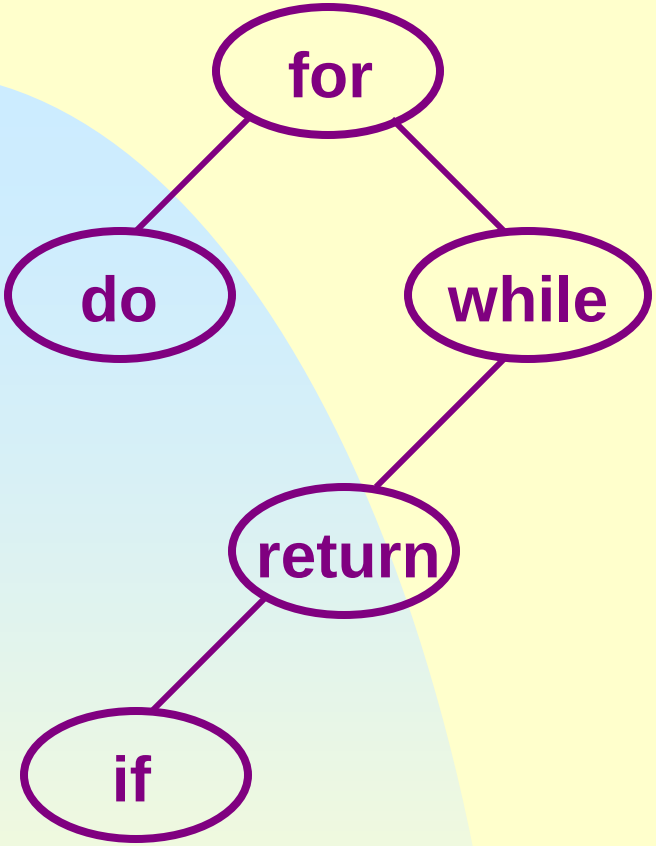


But it may not be the optimal binary search tree to use when different identifiers are to be searched with different probabilities.

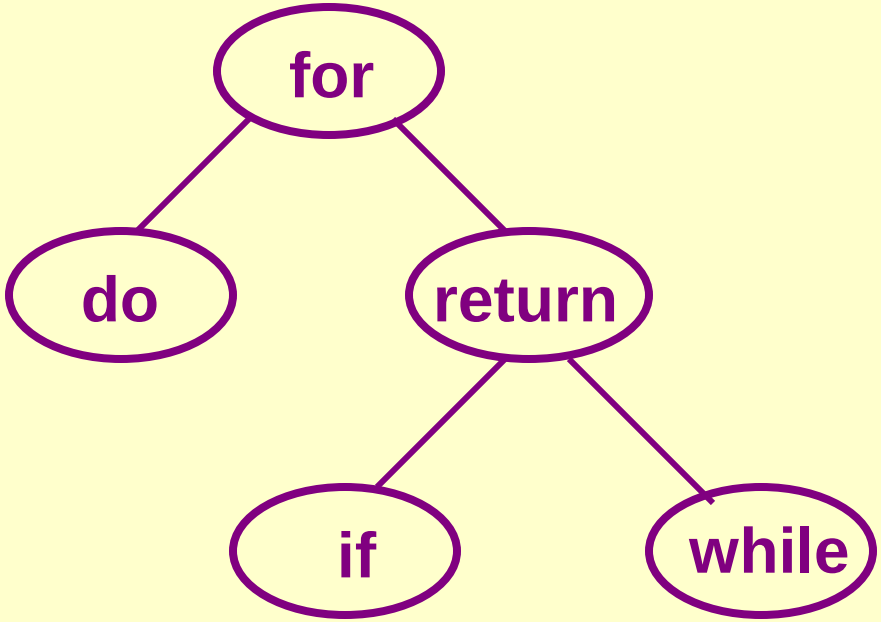
To find an optimal binary search tree for a given static file, we must first decide on a cost measure for search trees.

In the binary search tree *search algorithm*, an identifier at level k needs k iterations to find. Thus it is reasonable to use the level number of a node as its cost.

Consider 2 search trees of Fig. 10.2.



(a)



(b)

Fig. 10.2 Two binary search trees

- **worst case: tree(a) requires 4 comparisons, tree(b) requires 3 comparisons, (b) is better.**
- **average case:**
  - **Tree(a): 1 comparison---for, 2---do, 2---while, 3---return, 4---if. With equal probability, the average number of comparisons for a successful search is (1+2+2+3+4)/5=2.4 .**
  - **Tree(b): (1+2+2+3+3)/5=2.2**

(b) is better too.

To reflect the cost of unsuccessful search, it is useful to add a special "square" node at every null link.

Doing this to the trees of Fig. 10.2 yields the trees of Fig. 10.3.

Binary tree with n nodes has n+1 null links and therefore will have n+1 square nodes.

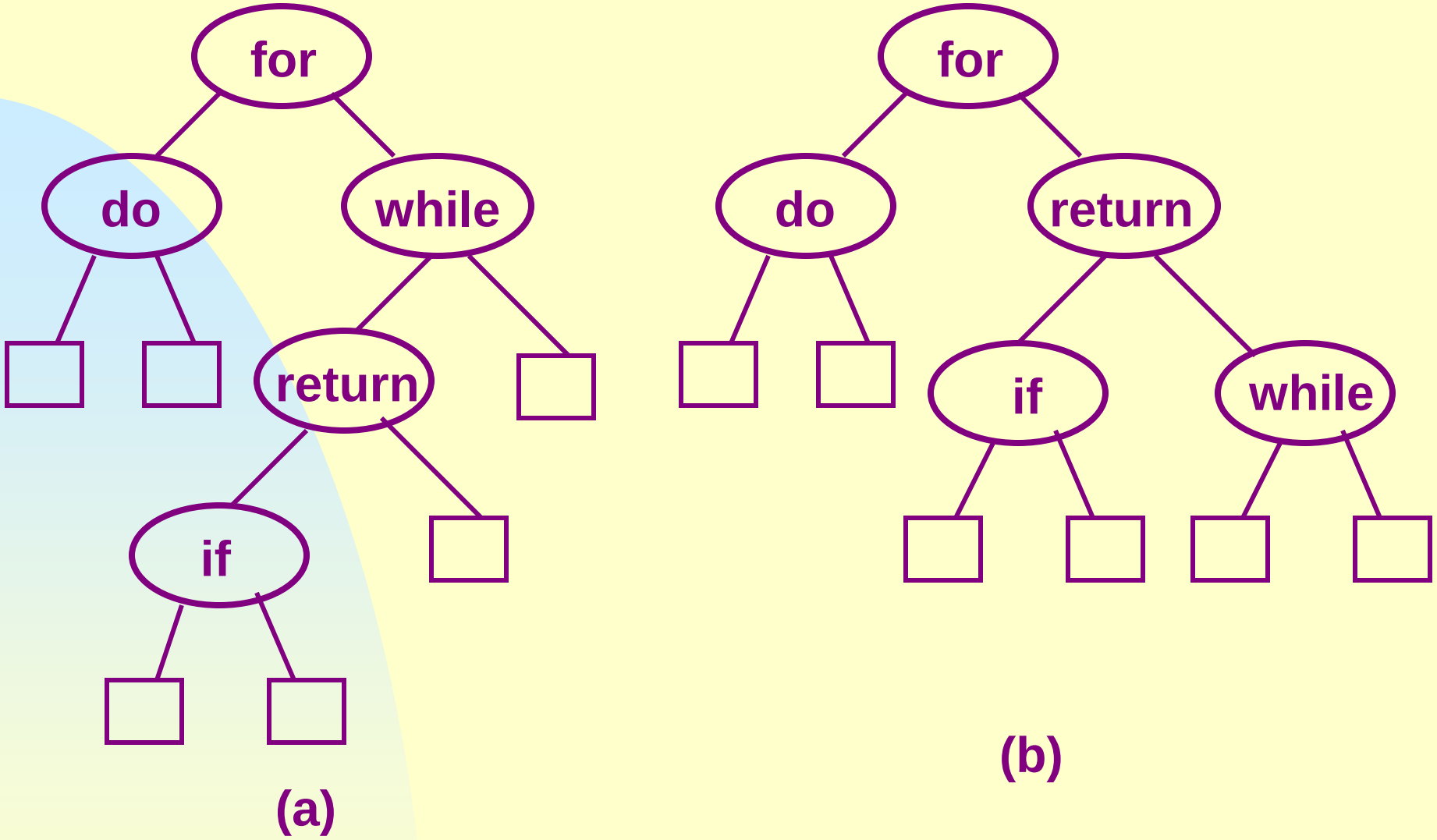


Fig. 10.3 Extended binary trees corresponding to search trees of Fig. 10.2

- **external (failure) nodes---square nodes.**
- **internal nodes---original nodes.**
- **extended binary tree---a binary tree with external nodes added.**
- **external path length, E, of a binary tree---the sum over all external nodes of the lengths of the paths from the root to those nodes.**
- **internal path length, I, of a binary tree---the sum over all internal nodes of the lengths of the paths from the root to those nodes.**

For Fig. 10.3(a):

I=0+1+1+2+3=7, E=2+2+4+4+3+2=17

In general, E=I+2n.

Hence, binary trees with the maximum E also have maximum I.

Over all binary trees with n internal nodes:

- **worst case: when the tree has a depth of n.**

$$I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

this is the maximum I.

• optimal case: as many internal nodes as close to the root as possible---complete binary tree.

I=0+2*1+4*2+8*3+……

If we number the nodes in a complete binary tree as before, the distance of node i from the root is

$$\lfloor \log_2 i \rfloor$$

so

$$I = \sum_{1 \le i \le n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

Now return to the problem of finding an optimal binary search tree for a set of identifiers to be searched with different probabilities.

If the binary search tree contains the identifiers $a_1, a_2, \ldots\ldots, a_n$ with $a_1 < a_2 < \ldots\ldots < a_n$, and the probability of searching for each a_i is p_i, then the total cost of any binary search tree is

$$\sum_{1 \le i \le n} p_i \,.\, \text{level}(a_i)$$

when only **successful** searches are made.

The cost of unsuccessful searches should also be included.

Unsuccessful searches terminate with algorithm *search* return a 0 pointer. Every node (*) with a null subtree defines a point at which such a termination can take place. Replace every null subtree by a failure node.

Identifiers not in the binary search tree may be partitioned into n+1 classes, E_i, $0 \le i \le n$.

$$E_0 = \{x \mid x < a_1\}$$

$$E_i = \{x \mid a_i < x < a_{i+1}\}, \quad 1 \le i < n.$$

$$E_n = \{x \mid x > a_n\}$$

For all x in E_i, $0 \le i \le n$, the search terminates at the same failure node i, and it terminates at different failure nodes for x in different classes.

If q_i is the probability for E_i, the cost of the failure nodes is

$$\sum_{0 \le i \le n} q_i.(\text{level(failure node i)}-1) \quad (*)$$

Therefore the total cost is

$$\sum_{1 \le i \le n} p_i.\text{level}(a_i) + \sum_{0 \le i \le n} q_i.(\text{level(failure node i)}-1) \quad (10.1)$$

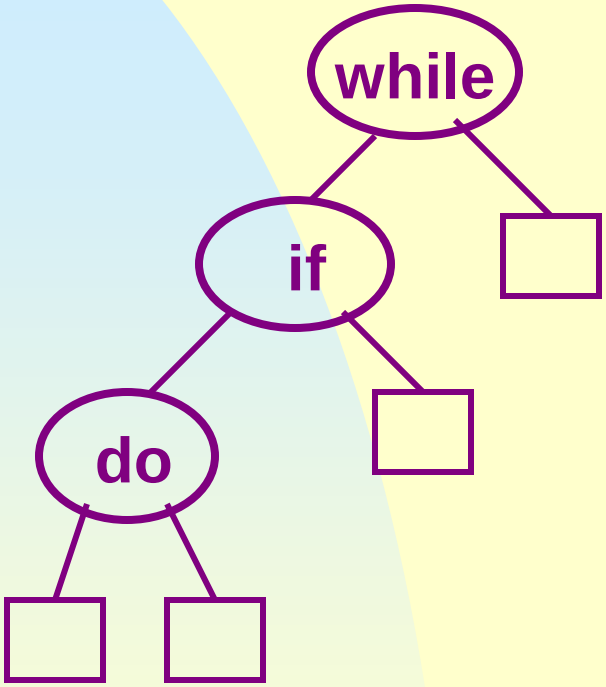
An optimal binary search tree for the identifier set $a_1, a_2, \ldots\ldots, a_n$ is the one that minimizes Eq. (10.1) over all possible binary search trees for this identifier set.

Note

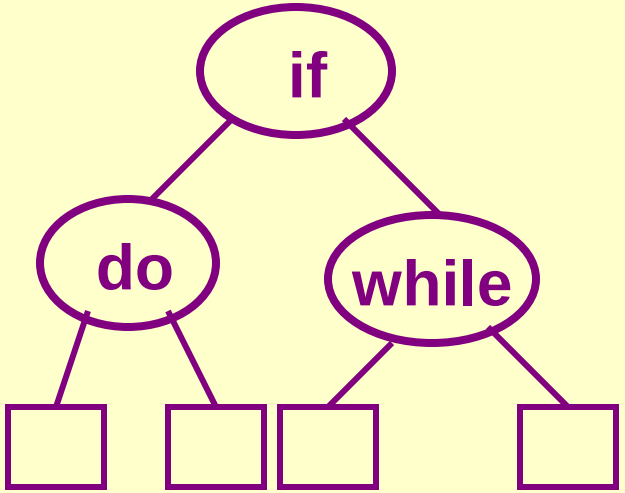
$$\sum_{1\le i\le n} p_i + \sum_{0\le i\le n} q_i = 1$$

Example 10.1:

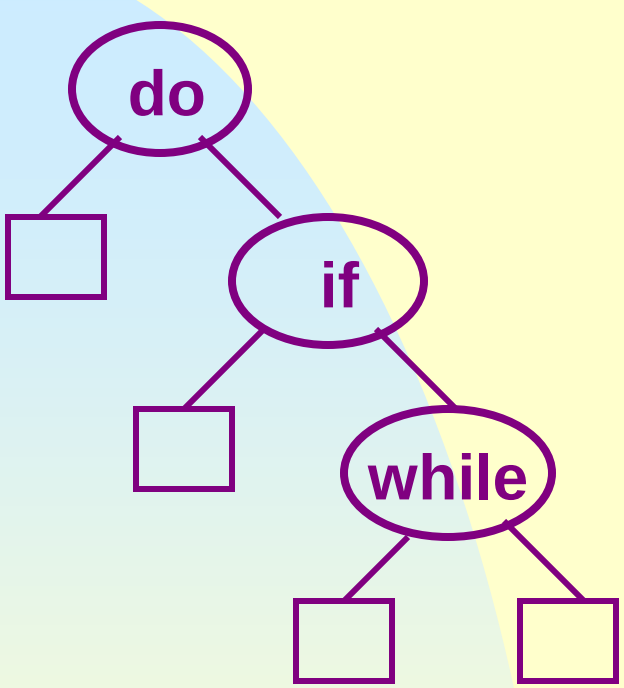
(a_1, a_2, a_3)=(do, if, while). The possible binary search tree for it are as follows:



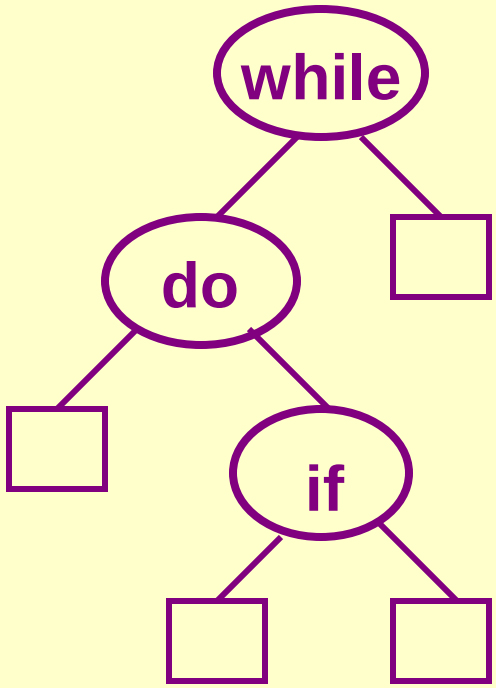
(a)



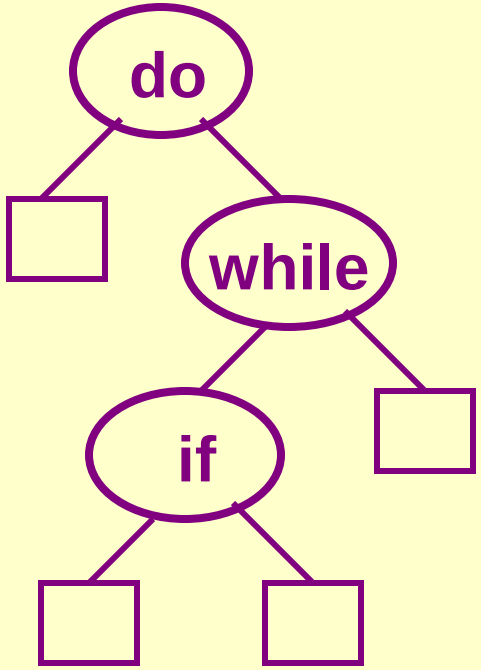
(b)



(c)



(d)



(e)

- with equal probabilities, $p_i=q_j$ for all i and j.

  cost(tree a)=15/7; cost(tree b)=13/7

  cost(tree c)=15/7; cost(tree d)=15/7

  cost(tree e)=15/7

  As expected, tree b is optimal.

- with $p_1=0.5$, $p_2=0.1$, $p_3=0.05$, $q_0=0.15$, $q_1=0.1$, $q_2=0.05$, $q_3=0.05$

  cost(tree a)=2.65; cost(tree b)=1.9

  cost(tree c)=1.5; cost(tree d)=2.15

  cost(tree e)=1.6

For instance,

cost(tree d) =0.05+2*0.5+3*0.1

+2*0.15+3*0.1+3*0.05+0.05

=2.15 (note the error in P546)

tree c is optimal.

How to determine the optimal tree?

One way: explicitly generate all possible binary search trees, compute the cost of each tree in O(n), then determine the optimal---need

$$O\left(\frac{n}{n+1}C_n^{2n}\right)=O\left(C_n^{2n}\right)$$ --- no good.

Instead, we can get an efficient algorithm by making use of the properties of optimal binary search trees(b.s.t.).

Let $a_1<a_2<,\ldots\ldots,<a_n$ be n identifiers in a b.s.t., denote:

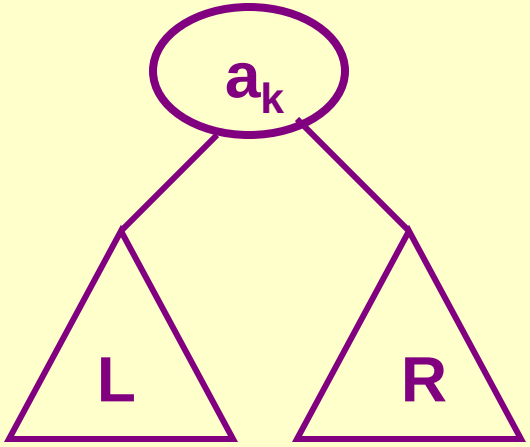
- T_{ij}---an optimal b.s.t. for $a_{i+1},\ldots\ldots, a_j$, $i<j$. T_{ii} is an empty tree, $0\le i\le n$. T_{ij} $(i>j)$ is not defined.
- c_{ij}---the cost of T_{ij}. $c_{ii}=0$.
- r_{ij}---the root of T_{ij}.
- w_{ij}---the weight of T_{ij}. $w_{ij}= q_i+ \sum_{k=i+1}^{j} (q_k+ p_k)$.

- $r_{ii}=0$, $w_{ii}=q_i$, $0 \le i \le n$.
- The optimal b.s.t. for $a_1,\ldots\ldots, a_n$ is T_{0n}. Its cost is c_{0n}, root r_{0n}.

If T_{ij} is an optimal b.s.t. for $a_{i+1},\ldots\ldots, a_j$ and $r_{ij}=k$, $i<k \le j$, then T_{ij} has 2 subtrees L and R.

L contains the identifiers $a_{i+1},\ldots\ldots, a_{k-1}$.

R contains the identifiers $a_{k+1},\ldots\ldots, a_j$.



The cost c_{ij} of T_{ij} is

$$c_{ij}=p_k+cost(L)+cost(R)+w_{i,k-1}+w_{kj} \qquad (10.2)$$

From (10.2), if c_{ij} is to be minimal, then cost(L)=$c_{i,k-1}$ and cost(R)=c_{kj} , as otherwise we could replace either L or R by a subtree with a lower cost, thus generating a b.s.t. for $a_{i+1},\ldots\ldots, a_j$ with a lower cost than c_{ij}. This violates the assumption that T_{ij} is optimal.

Hence, E.q.(10.2) becomes

$$c_{ij}=p_k+w_{i,k-1}+w_{kj}+c_{i,k-1}+c_{kj}$$

$$=w_{ij}+c_{i,k-1}+c_{kj} \qquad (10.3)$$

Since T_{ij} is optimal, from (10.3) we know $r_{ij}=k$ is such that

$$w_{ij}+c_{i,k-1}+c_{kj} = \min_{i<l\le j}\{w_{ij}+c_{i,l-1}+c_{lj}\}$$

or

$$c_{i,k-1}+c_{kj} = \min_{i<l\le j}\{c_{i,l-1}+c_{lj}\} \qquad (10.4)$$

Note also

$$w_{ij}=w_{i,j-1}+p_j+q_j .$$

(10.4) gives us a means of obtaining r_{0n}, T_{0n} and c_{0n}, starting from $r_{ii}=0$, $T_{ii}=\varnothing$ and $c_{ii}=0$.

Example 10.2:

Let $n=4$, $(a_1, a_2, a_3, a_4)=$(do, if, return,while).

Let $(p_1, p_2, p_3, p_4)=(3,3,1,1)$

and $(q_0, q_1, q_2, q_3, q_4)=(2,3,1,1,1)$.

The p's and q's are as integer for convenience.

Initially, $w_{ii} = q_i$, $r_{ii}=0$, $c_{ii}=0$, $0 \le i \le 4$.

Using (10.3) and (10.4), we get

$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8$

$c_{01} = w_{01} + \min \{c_{00} + c_{11}\} = 8$

$r_{01} = 1$

$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7$

$c_{12} = w_{12} + \min \{c_{11} + c_{22}\} = 7$

$r_{12} = 2$

$w_{23} = p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3$

$c_{23} = w_{23} + \min \{c_{22} + c_{33}\} = 3$

$r_{23} = 3$

$w_{34} = p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3$

$c_{34} = w_{34} + \min \{c_{33} + c_{44}\} = 3$

$r_{34} = 4$

Knowing $w_{i,i+1}$, $c_{i,i+1}$, we can use EQ.(10.3) and (10.4) to compute $w_{i,i+2}$, $c_{i,i+2}$, $r_{i,i+2}$:

$w_{02} = p_2 + q_2 + w_{01} = 3 + 1 + 8 = 12$

$c_{02} = w_{02} + \min \{c_{00} + c_{12} , c_{01} + c_{22}\} = 12+7=19$

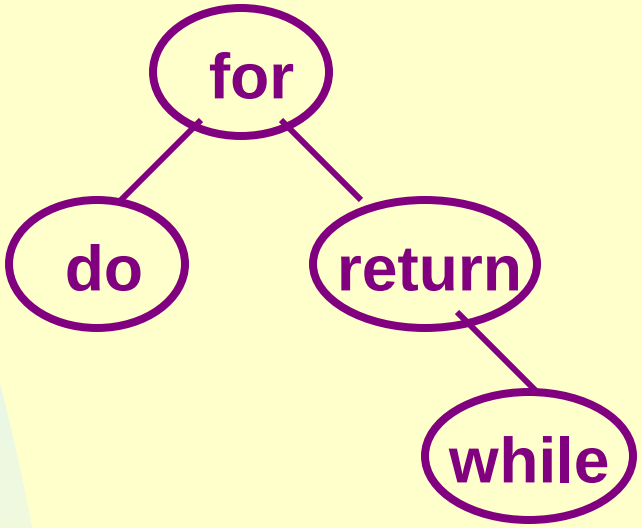
$r_{02} = 1$

…

until w_{04}, c_{04} and r_{04} are obtained. The next slide show the results of the computation.

j-i	i = 0	i = 1	i = 2	i = 3	i = 4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

With the data in the table, it is possible to construct T_{04} as shown below:



Optimal b.s.t. for Example 10.2

In general, we need to compute c_{ij} for j-i=1,2,…,n.

And for each j-i=m,

there are n-m+1 c_{ij} 's (c_{0m},…, $c_{n-m,n}$) to compute.

The computation of each c_{ij} need to find the minimum from m quantities (10.4)---need O(m).

The total time for all c_{ij}'s with j-i=m is O((n-m)m).

The total time is

$$\sum_{1\le m\le n}((n-m)m)=O(n^3)$$

In fact, Knuth proved that the optimal l in (10.4) may be found by limiting the search to the range $r_{i,j-1} \leq l \leq r_{i+1,j}$. In this case, the computing time becomes $O(n^2)$. Algorithm obst uses this result.

Class BST of Chapter 5 is augmented to include obst as a private member function and to have the private data members r, c, and w.

Now the algorithm obst.

```
template <class KeyType>
BST<KeyType>::Obst(int *p,int *q,Element<KeyType>*a,int n)
//given a1< a2<… <an, and pj, 1≤j≤n, and qi, 0≤i≤n, compute cij
//for Tij for ai+1,…, aj. Also compute rij and wij.
{
   for (int i=0; i<n; i++) {
      w[i][i]=q[i]; r[i][i]=c[i][i]=0;     //initialize
      w[i][i+1]=q[i]+q[i+1]+p[i+1]; //optimal trees with one node
      r[i][i+1]=i+1;
      c[i][i+1]=w[i][i+1];
    }
   w[n][n]=q[n]; r[n][n]=c[n][n]=0;
```

```
    for (int m=2; m<=n; m++)  //find optimal b.s.t. with m nodes
        for (i=0; i<=n-m; i++)  {
            int j=i+m;
            w[i][j]=w[i][j-1]+p[j]+q[j];
            int k=KnuthMin(i,j);
            //KnuthMin return k in the range [r[i,j-1], r[i+1, j]]
            //minimizing c[i,k-1]+c[k,j]
            c[i][j]=w[i][j]+c[i][k-1]+c[k][j]; //Eq. (10.3)
            r[i][j]=k;
        }
}   // end of Obst
```

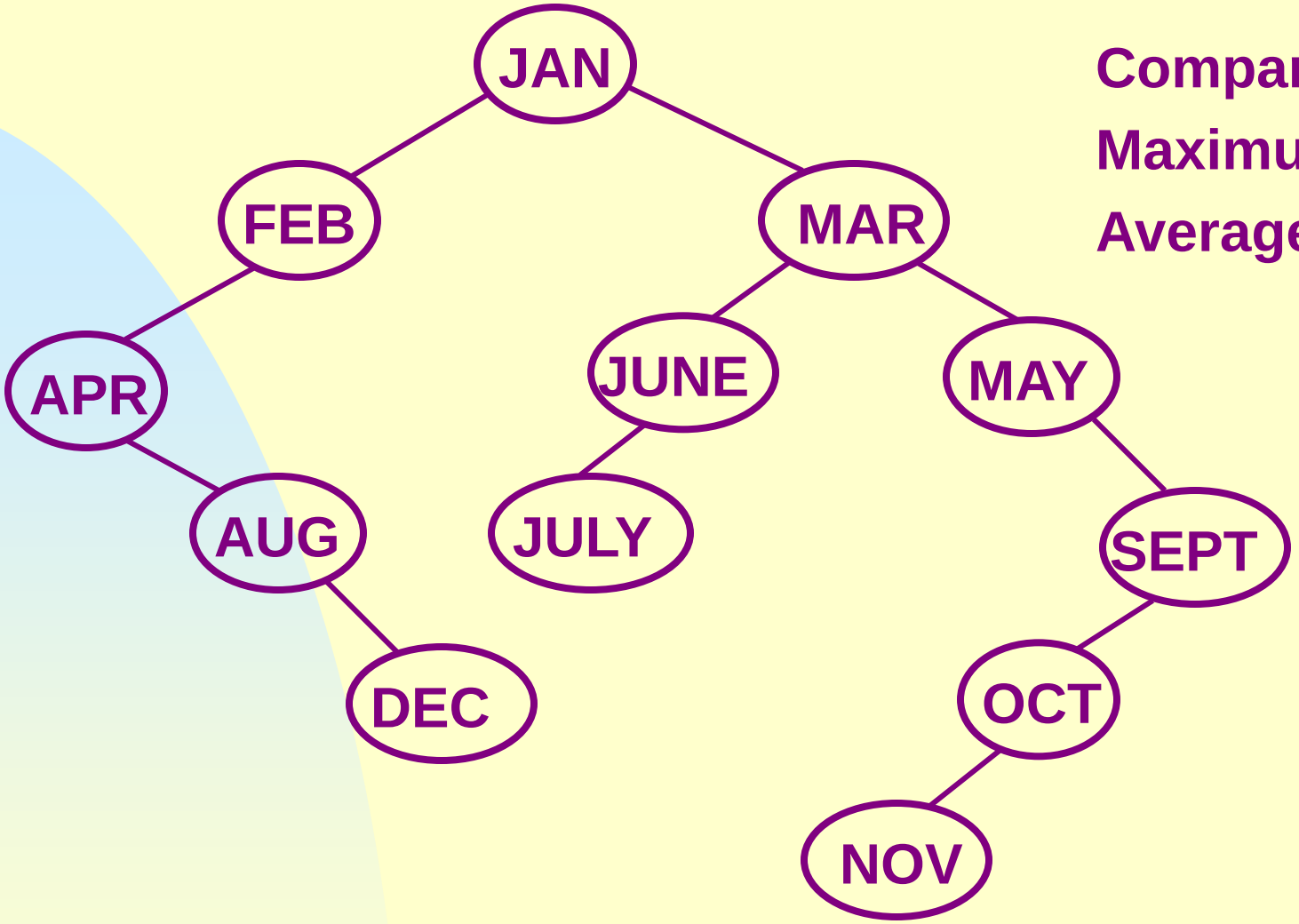
The actual T_{0n} may be constructed from r_{ij} in O(n).

Exercises: P562-3

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

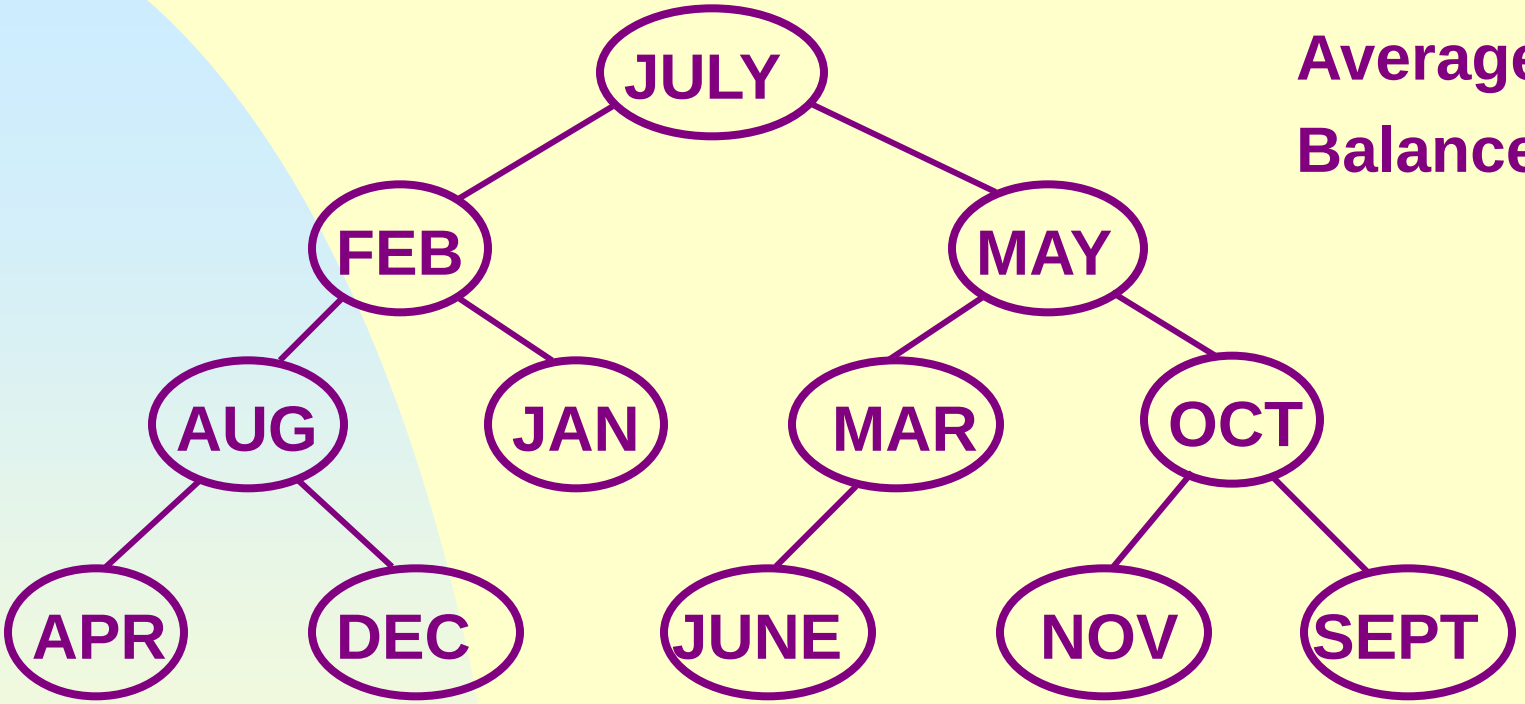
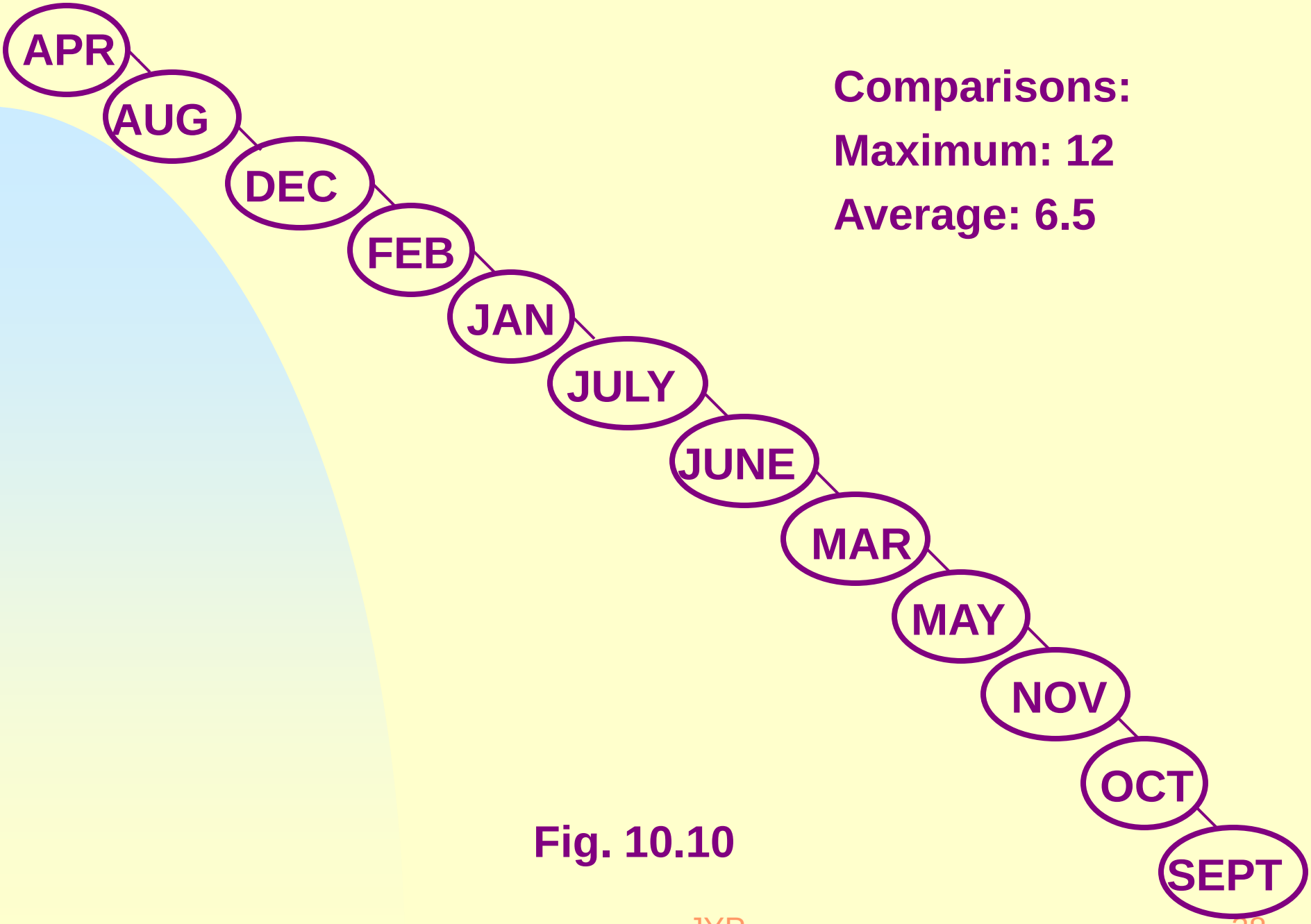


Fig. 10.9

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.

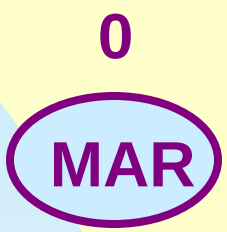
Fig. 10.9 is height-balanced.

Fig. 10.10 is not height-balanced.

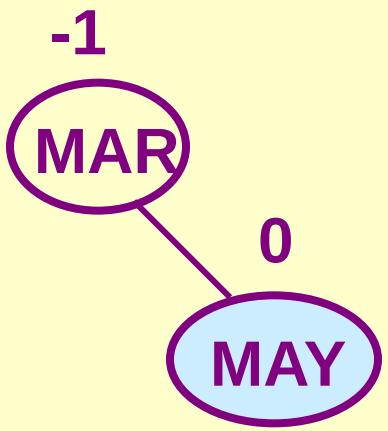
Definition: The balance factor, BF(T), of a node T in a binary tree is defined to be $h_L - h_R$. For any node T in an AVL tree BF(T)=-1, 0, or 1.

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.

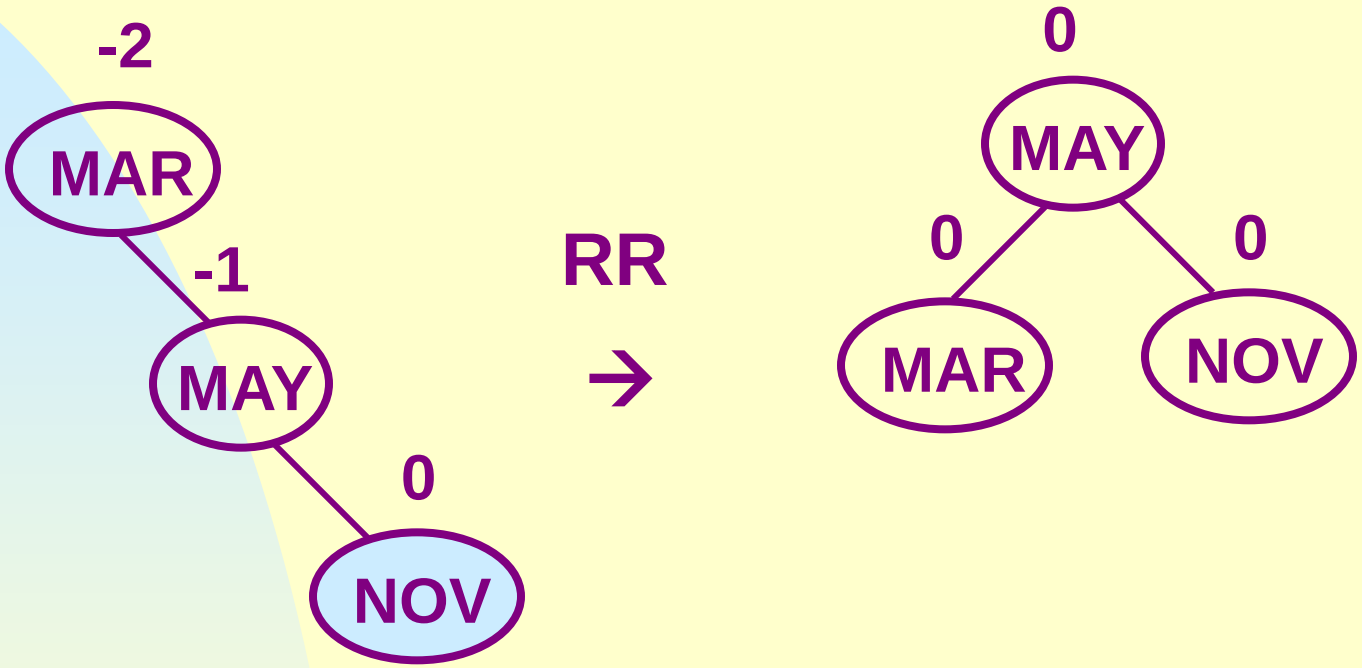
The following show the tree as it grows and restructuring involved in keeping it balanced. The numbers above each node represent the balance factor of that node.



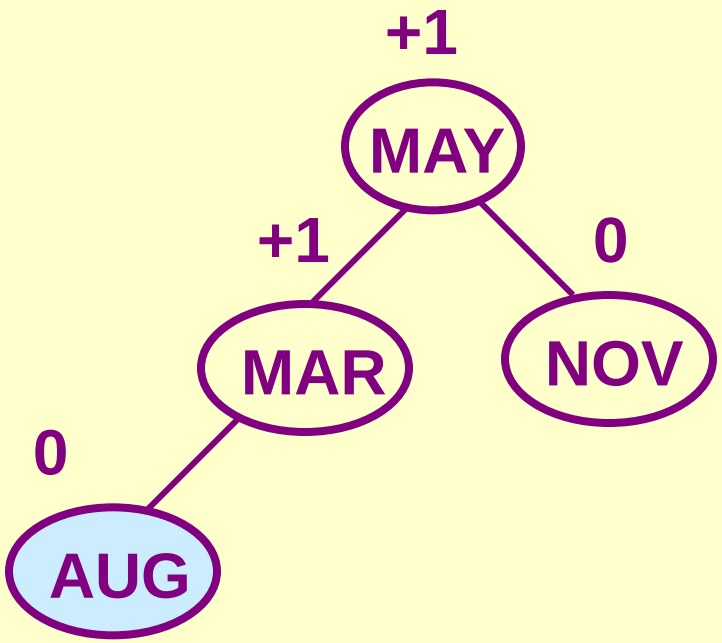
(a) Insert MAR



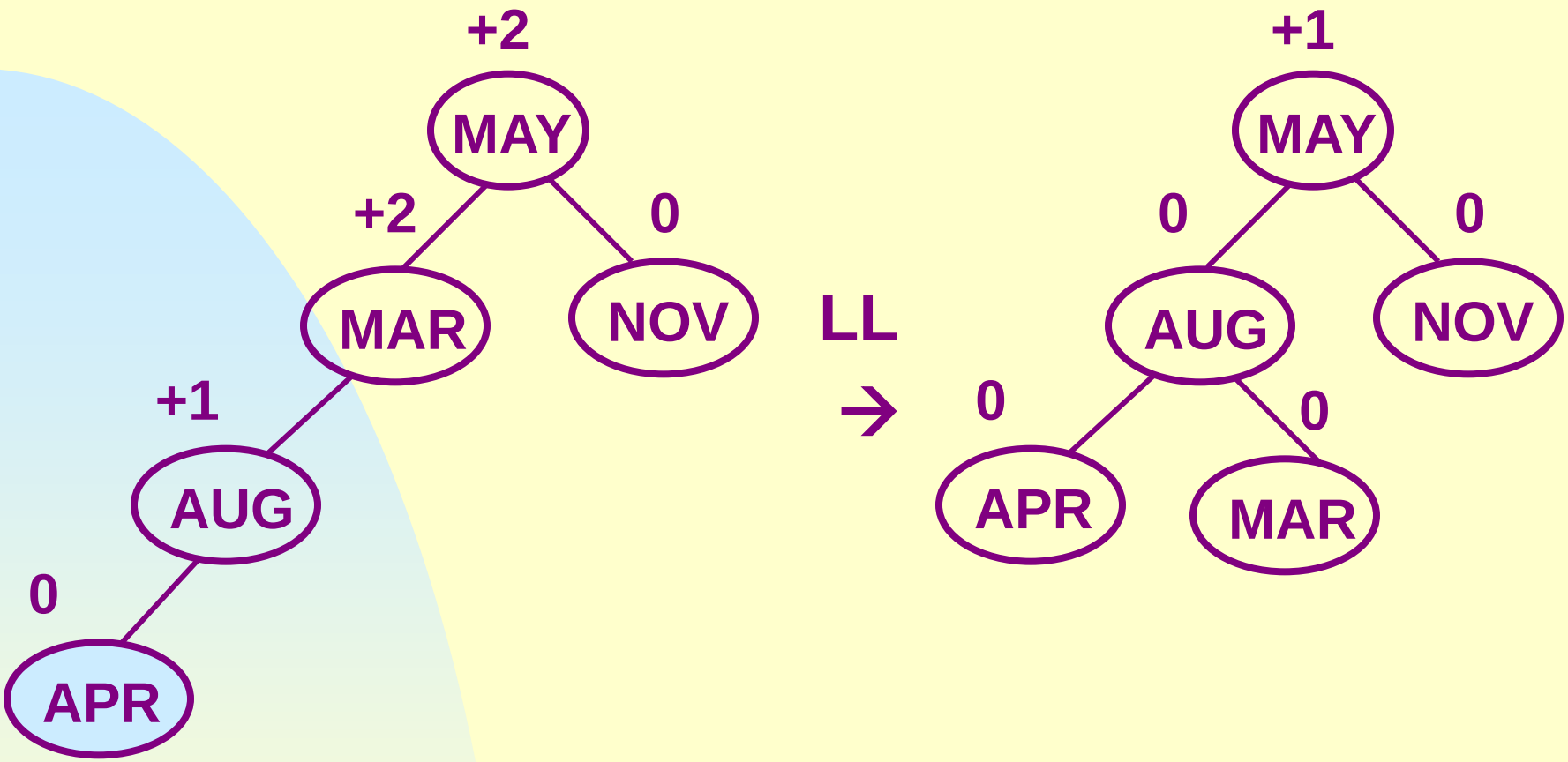
(b) Insert MAY



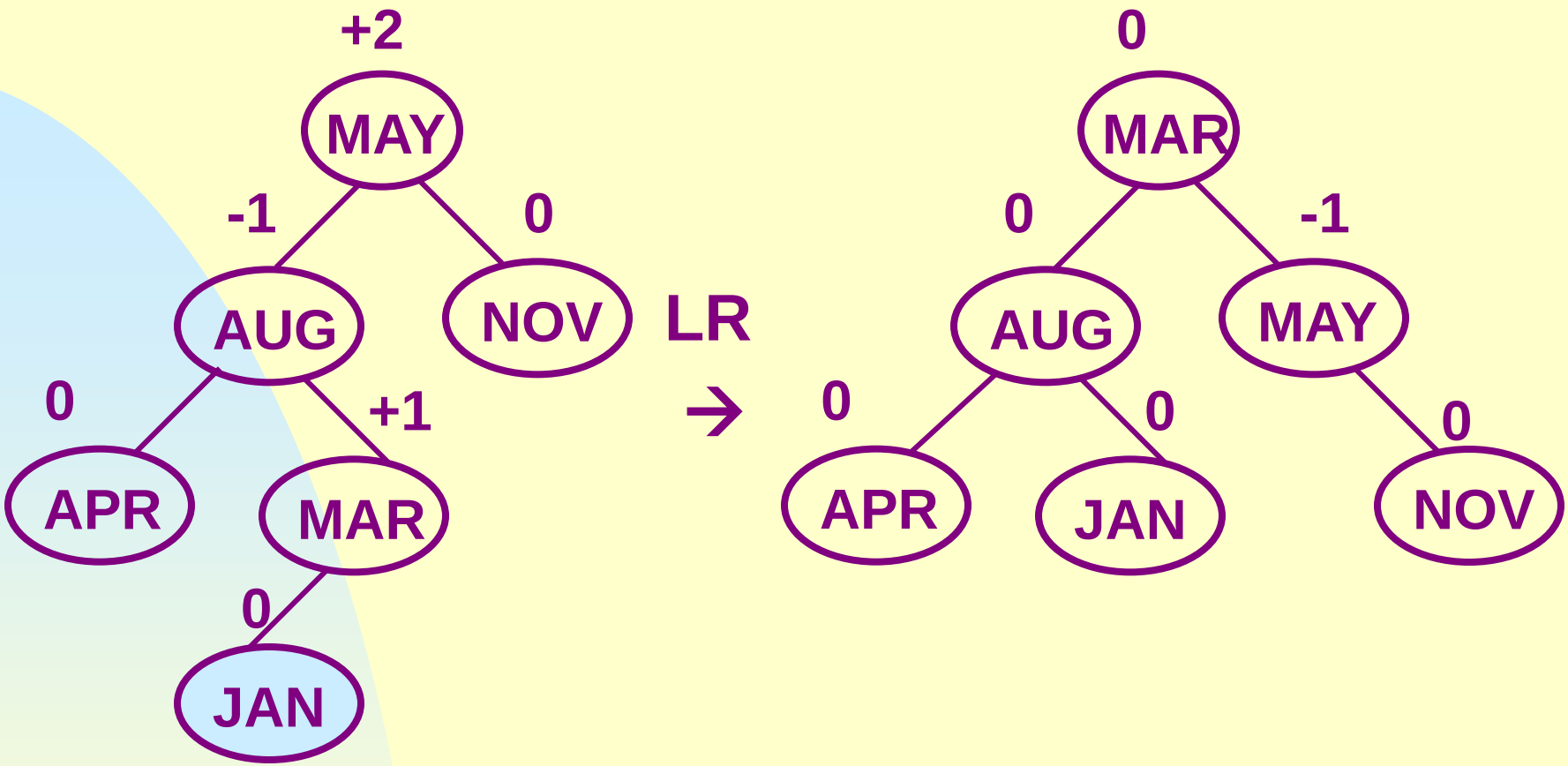
(c) Insert NOV



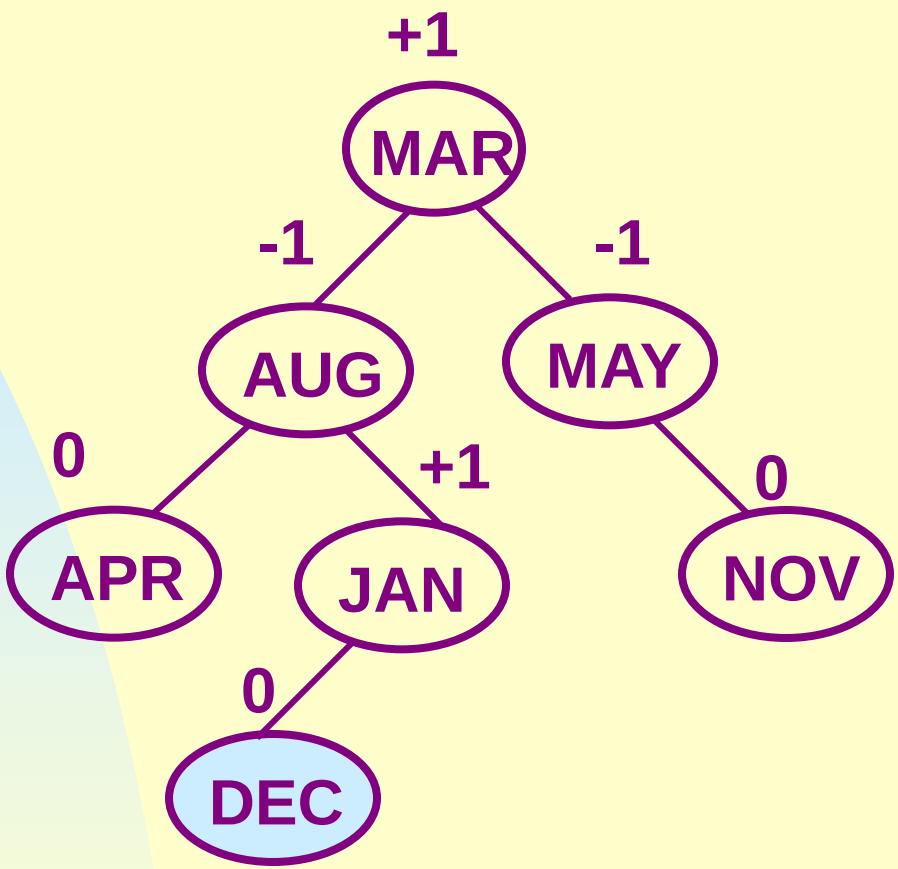
(d) Insert AUG



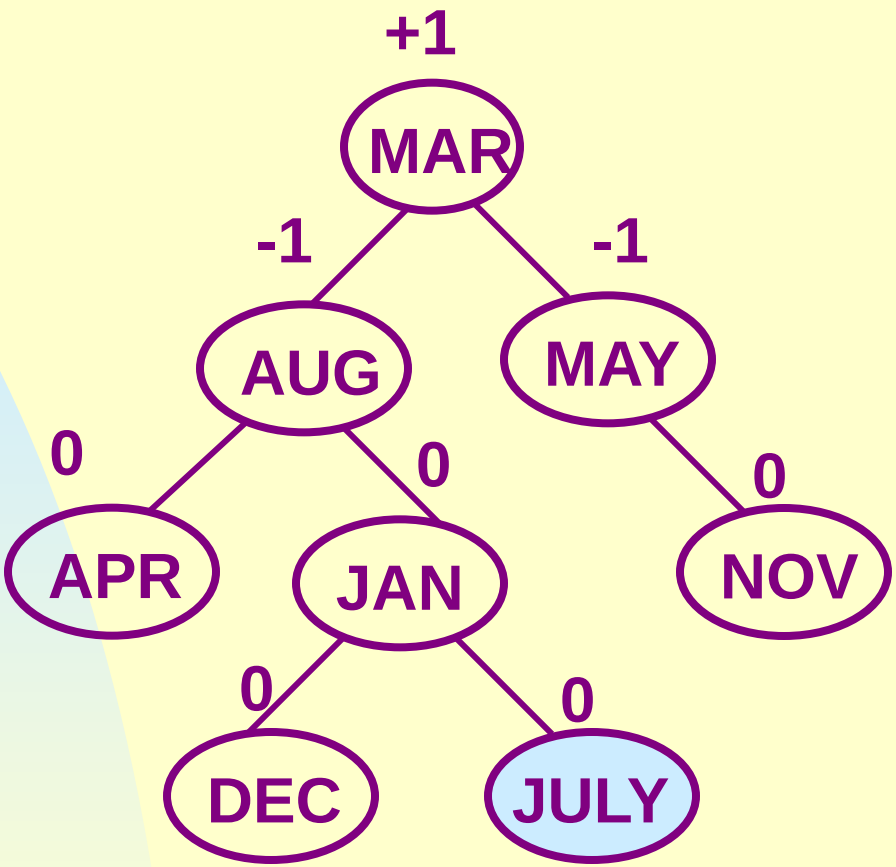
(e) Insert APR


+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

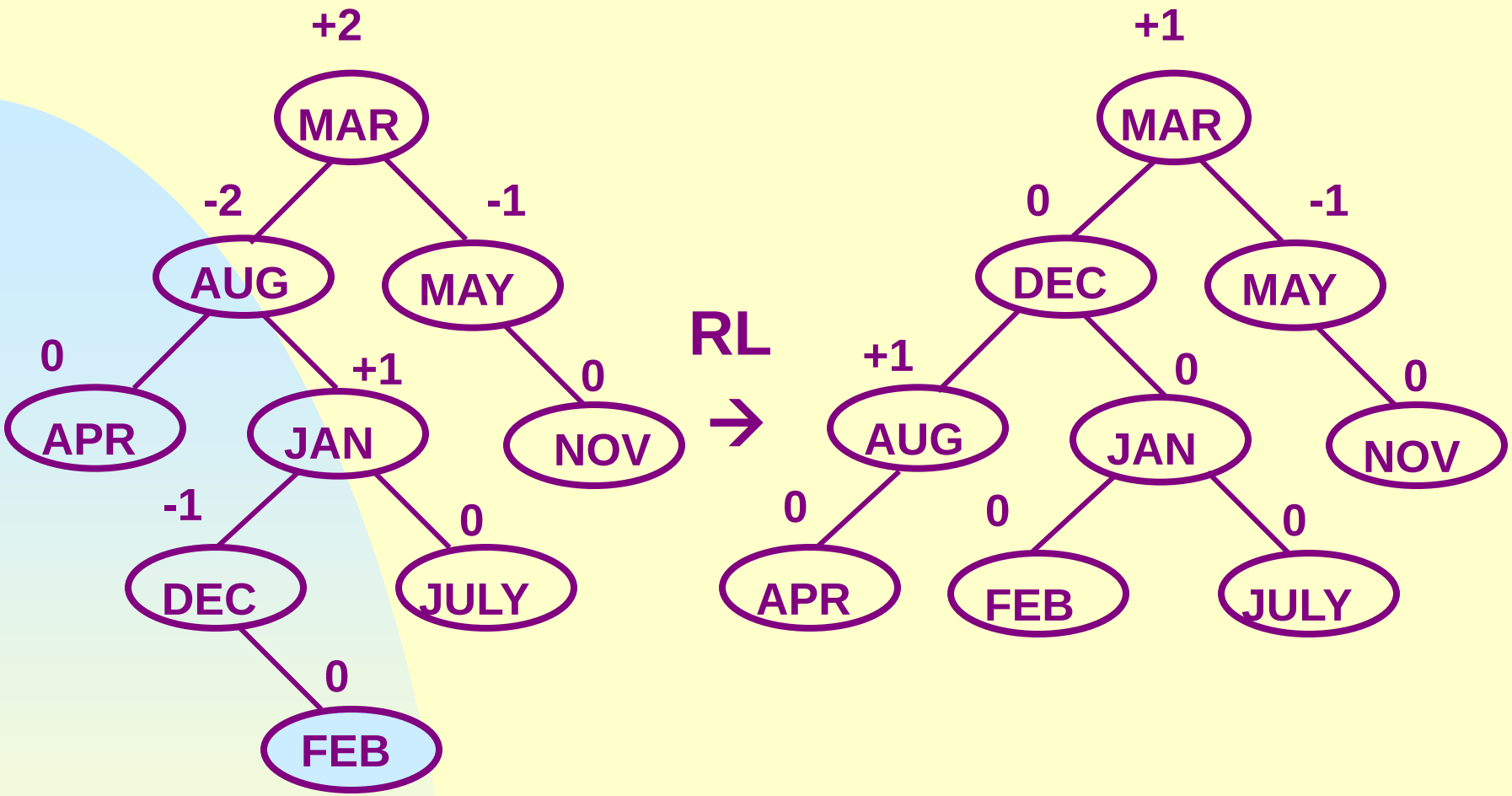
(f) Insert JAN



(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.

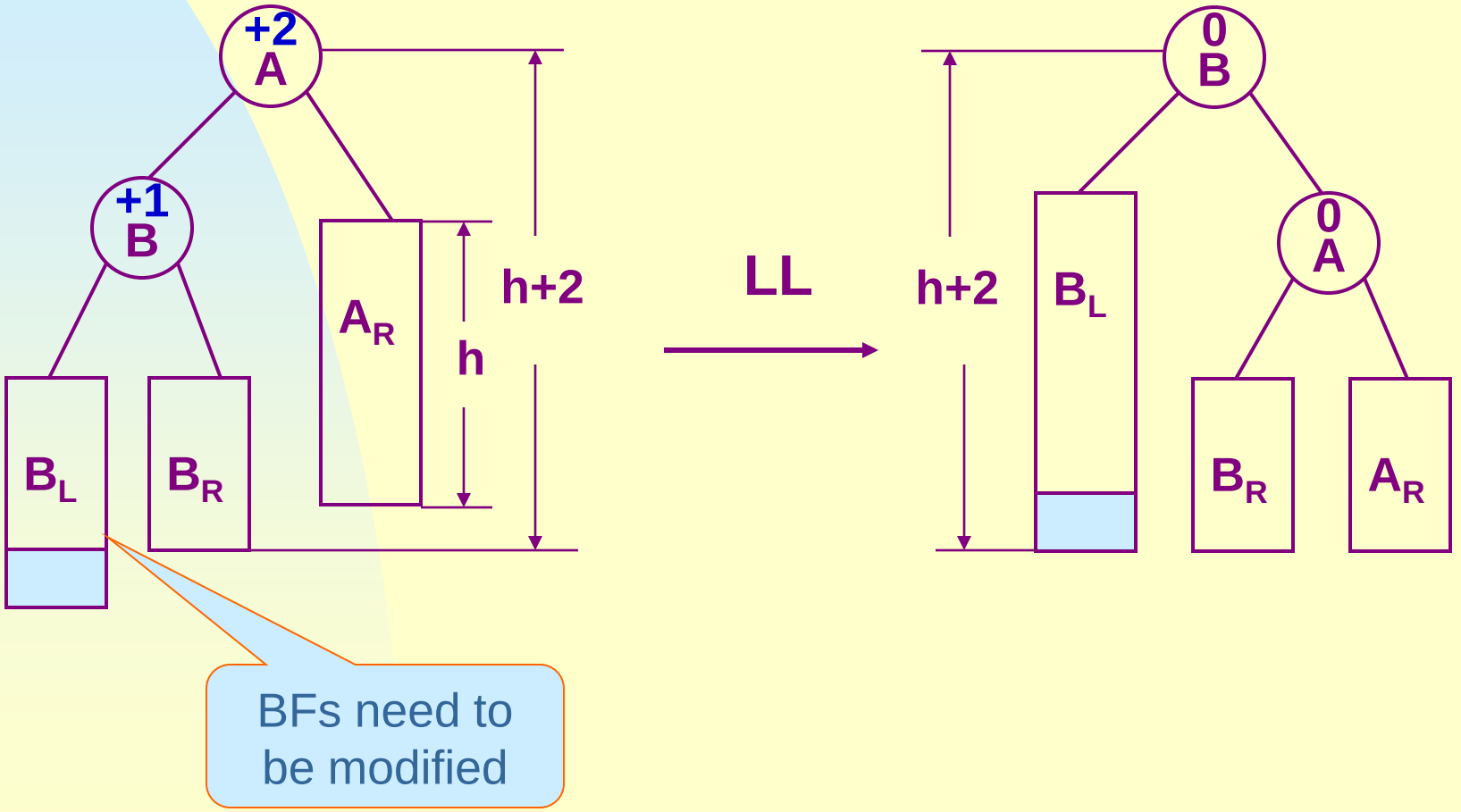
RL: Y is inserted in the left subtree of the right subtree of A.

As shown in the following:

Unbalanced following insertion

rotation type

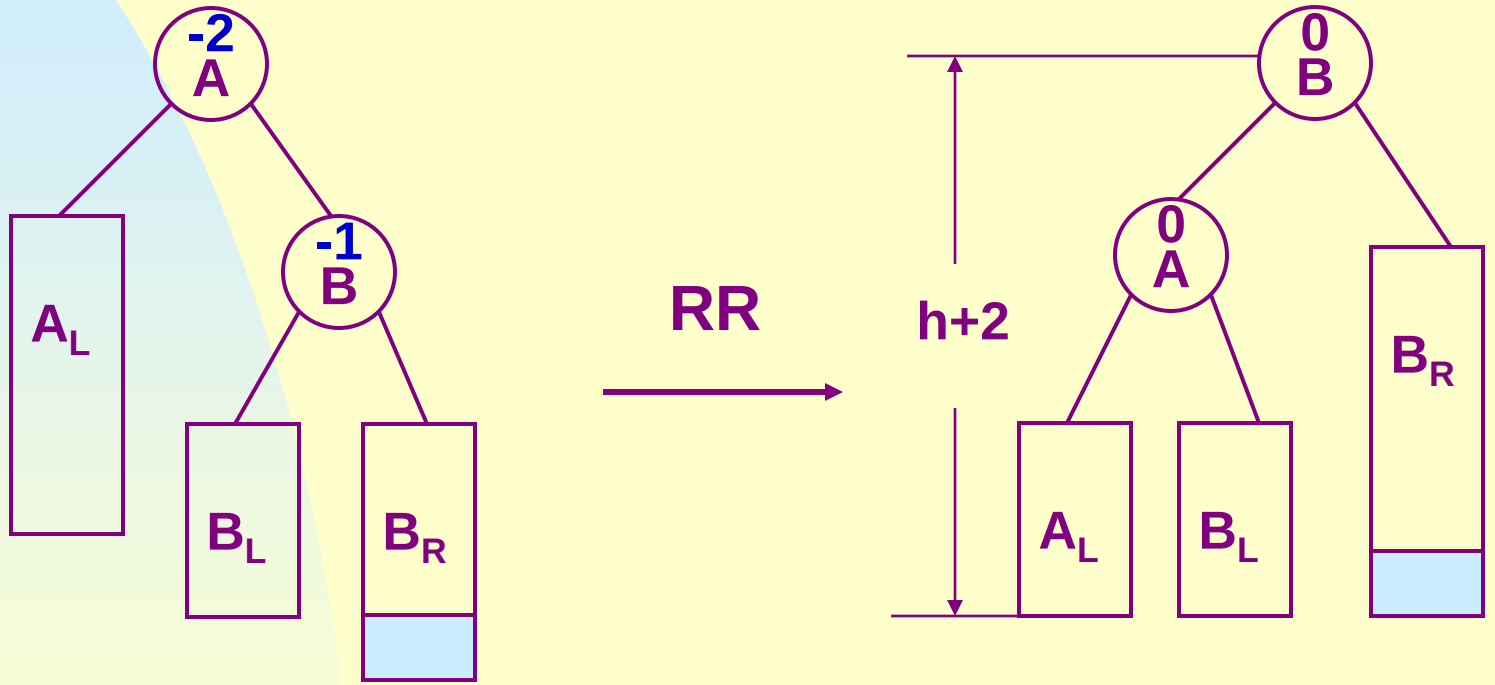
Rebalanced



Unbalanced following insertion

rotation type

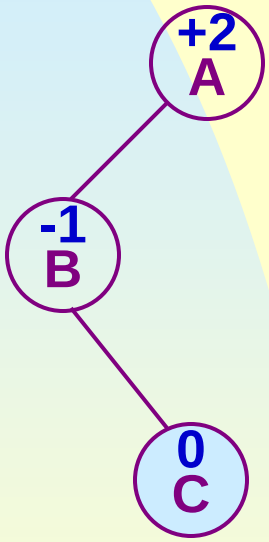
Rebalanced



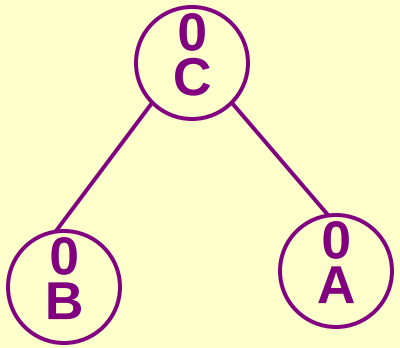
Unbalanced following insertion

rotation type

Rebalanced



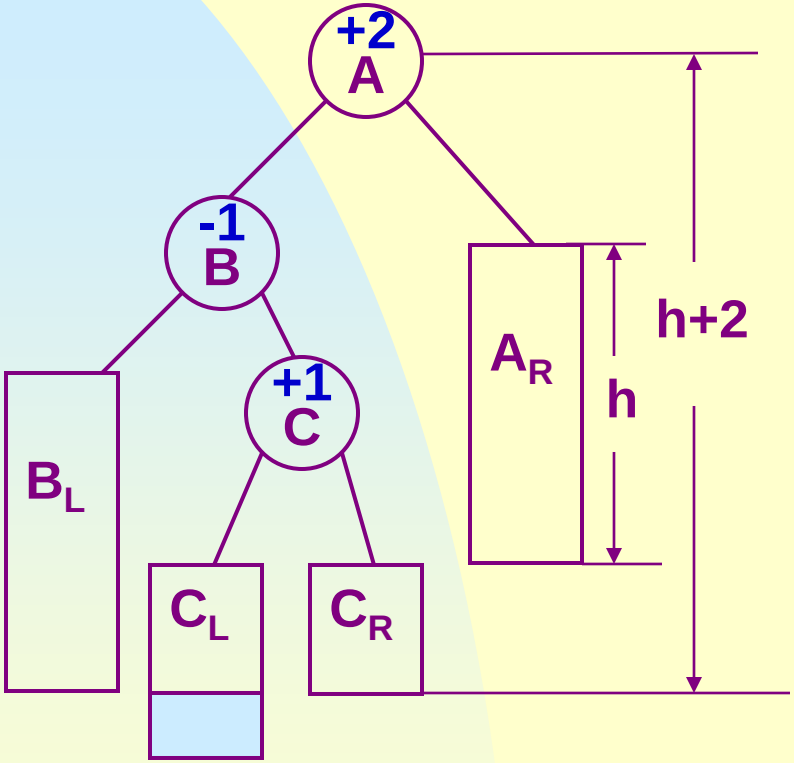
LR(a)



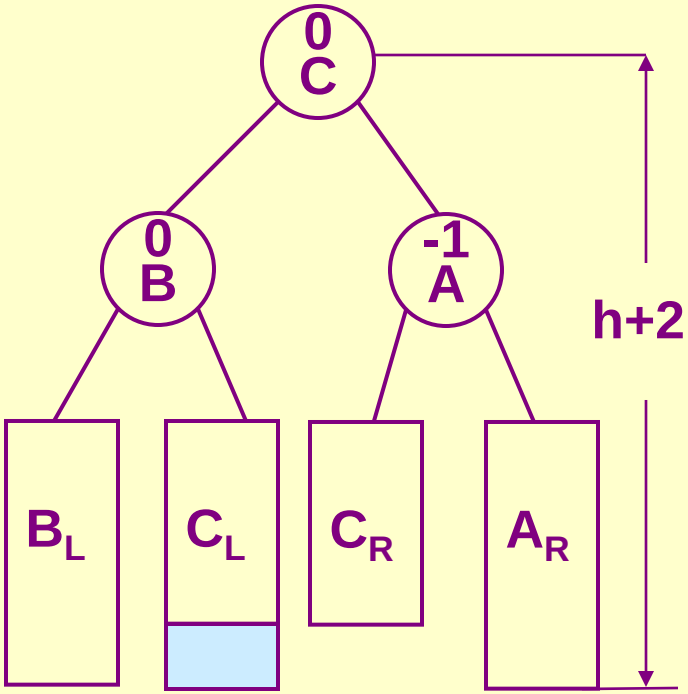
Unbalanced following insertion

rotation type

Rebalanced



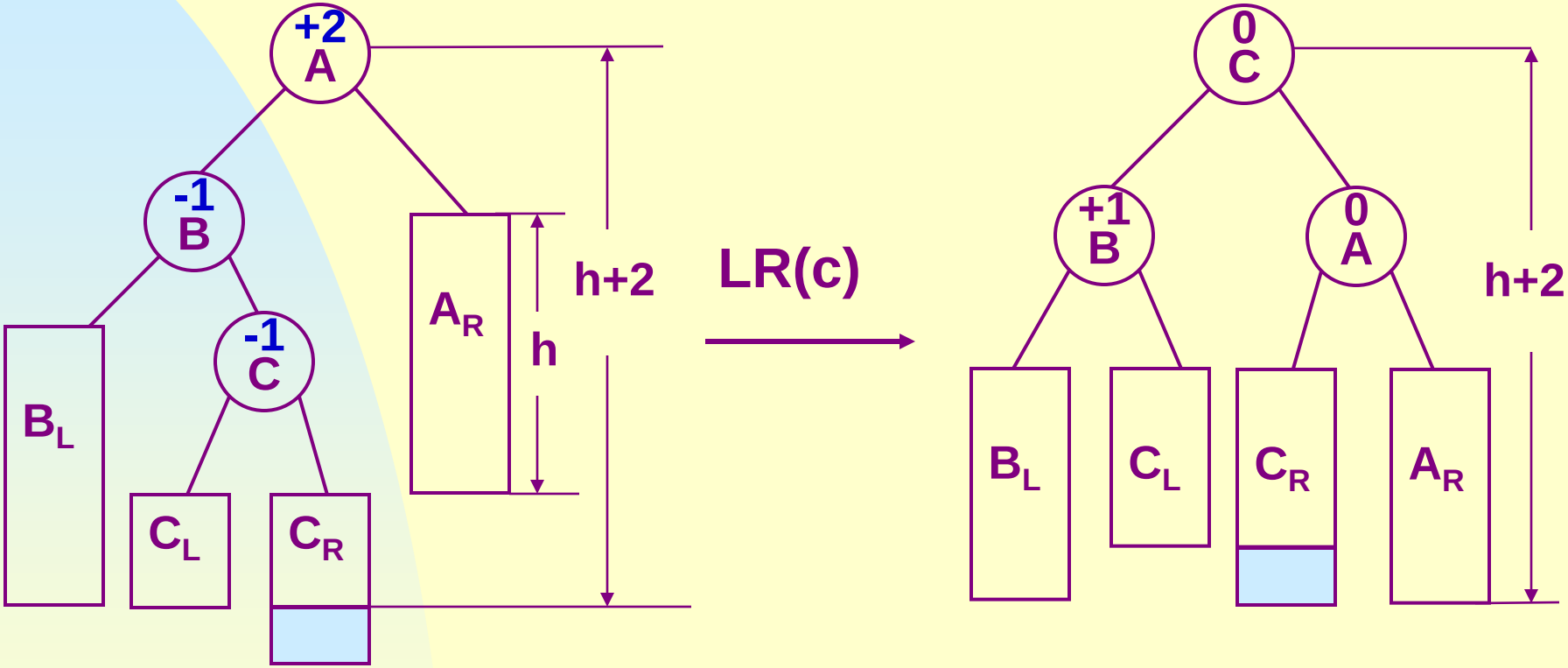
LR(b)



Unbalanced following insertion

rotation type

Rebalanced



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.

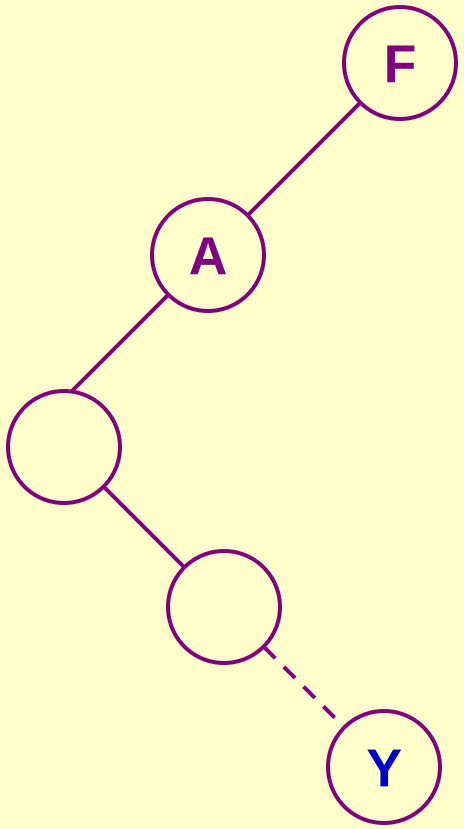
The only nodes whose BF can change are those in the subtree that is rotated.

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=0,       // 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

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

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