9.2 动 态 查 找 表

抽象数据类型动态查找表的定义如下：

ADT DynamicSearchTable {

数据对象D： D是具有相同特性的数据元素的集合。

每个数据元素含有类型相同的关键字， 可唯一标识数据元素。

数据关系R： 数据元素同属一个集合。

基本操作P： InitDSTable(&DT)

DestroyDSTable(&DT) SearchDSTable(DT, key);

InsertDSTable(&DT, e); DeleteDSTable(&T, key);

TraverseDSTable(DT, Visit());

}ADT DynamicSearchTable

InitDSTable(&DT);

操作结果： 构造一个空的动态查找表DT。

DestroyDSTable(&DT);

初始条件： 动态查找表DT存在；

操作结果： 销毁动态查找表DT。

SearchDSTable(DT, key);

初始条件： 动态查找表DT存在，key为和关键字类型相同的给定值；

操作结果： 若DT中存在其关键字等于key的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。

InsertDSTable(&DT, e);

初始条件：动态查找表DT存在，e为待插入的数据元素；

操作结果：若DT中不存在其关键字等于e.key的数据元素，则插入e到DT。

DeleteDSTable(&DT, key);

初始条件：动态查找表DT存在，key为和关键字类型相同的给定值；

操作结果：若DT中存在其关键字等于key的数据元素，则删除之。

TraverseDSTable(DT, Visit());

初始条件： 动态查找表DT存在，**Visit**是对结点操作的应用函数；

操作结果： 按某种次序对DT的每个结点调用函数**Visit()** 一次且至多一次。一旦 **Visit()** 失败，则操作失败。

9.2.1 二叉排序树和平衡二叉树

一、二叉排序树及其查找过程

1．二叉排序树定义：

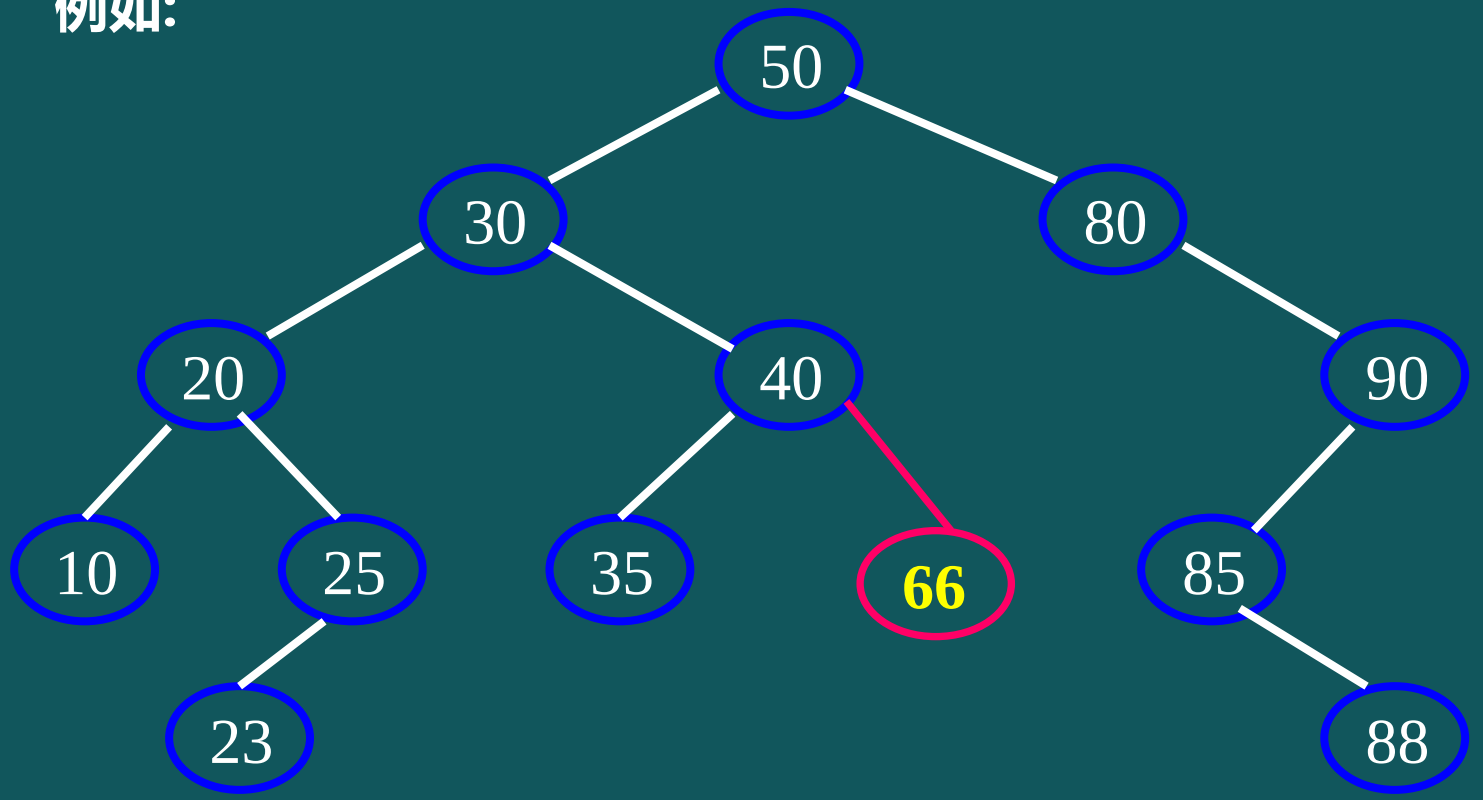
二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：

（1）若它的左子树不空，则左子树上所有结点的值均小于根结点的值；

（2）若它的右子树不空，则右子树上所有结点的值均大于根结点的值；

（3）它的左、右子树也都分别是二叉排序树。

例如:



不 是二叉排序树

通常，取二叉链表作为二叉排序树的存储结构。

```
typedef struct BiTNode { // 结点结构
    TElemType       data;
    struct BiTNode   *lchild, *rchild;
                                    // 左右孩子指针
} BiTNode, *BiTree;
```

2．二叉排序树的查找算法：

若二叉排序树**为空**，则**查找不成功**；

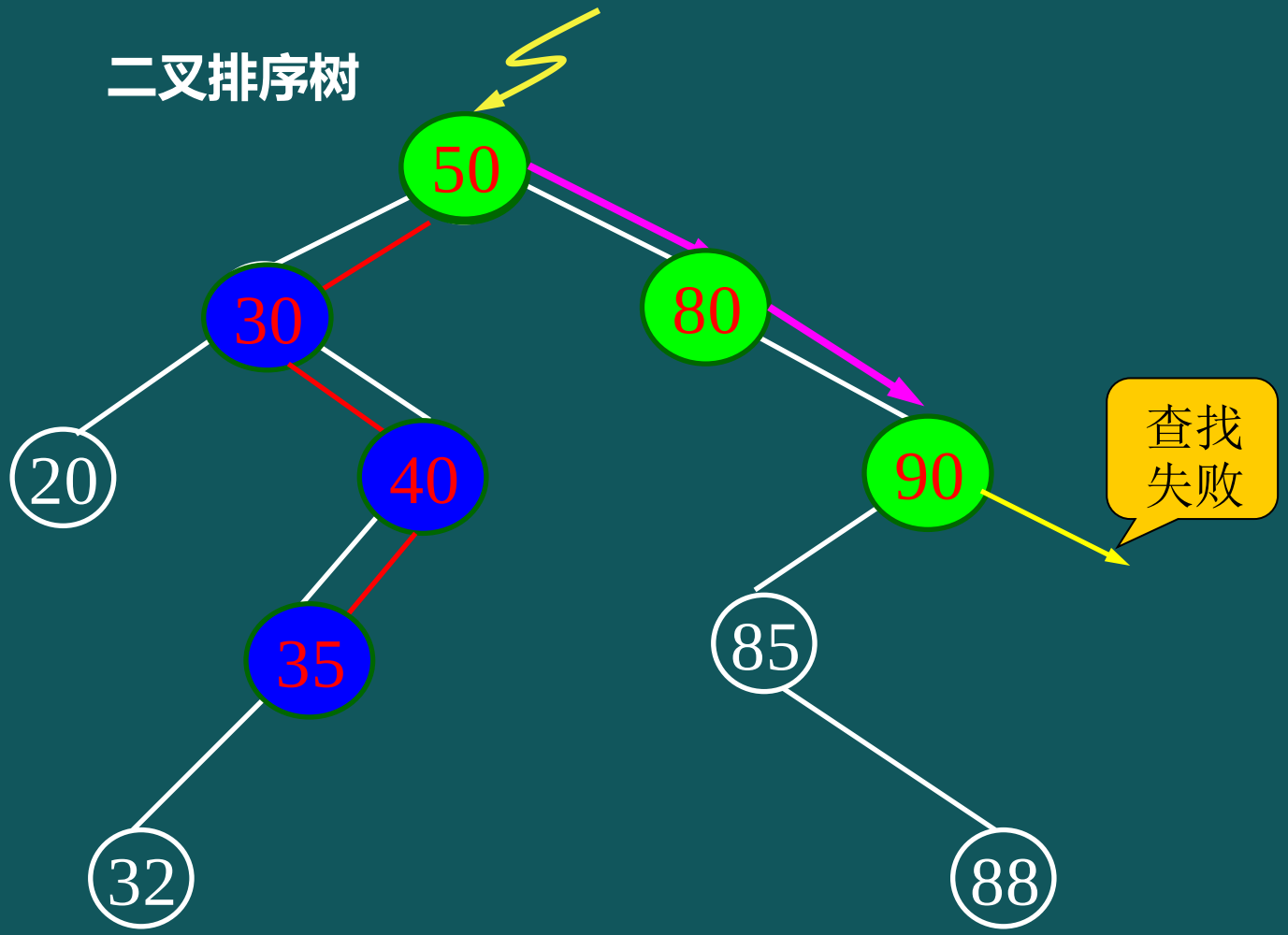
否则，

1）若给定值等于根结点的关键字，则查找成功；

2）若给定值小于根结点的关键字，则继续在左子树上进行查找；

3）若给定值大于根结点的关键字，则继续在右子树上进行查找。

例如:　　二叉排序树



查找关键字

= 50， 35 , 90 , 95 ,

从上述查找过程可见，

在查找过程中，生成了一条查找路径：

从根结点出发，沿着左分支或右分支逐层向下直至关键字等于给定值的结点;

——查找成功

或者

从根结点出发，沿着左分支或右分支逐层向下直至指针指向空树为止。

——查找不成功

算法描述如下：

```
Status SearchBST (BiTree T, KeyType key ) {
  // 在根指针 T 所指二叉排序树中递归地查找其关键
 //字等于 key 的数据元素，若查找成功，则返回指向
 //该数据元素的结点的指针，否则返回空指针
  if (!T)|| EQ(key, T->data.key) return TRUE //查找结束
  else  if ( LT(key, T->data.key) )
            return(SearchBST (T->lchild, key));
                                    // 在左子树中继续查找
        else return(SearchBST (T->rchild, key));
                                    // 在右子树中继续查找
} // SearchBST
```

算法9.5(a)

3．二叉排序树的插入算法

- 根据动态查找表的定义，**“插入”操作在查找不成功时才进行**；
- 若二叉排序树为空树，则新插入的结点为新的根结点；否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到。
- 为此，需将前面的二叉排序树的查找算法改写成算法9.5（b），以便在查找不成功时返回插入位置。插入算法如算法9.6所示。

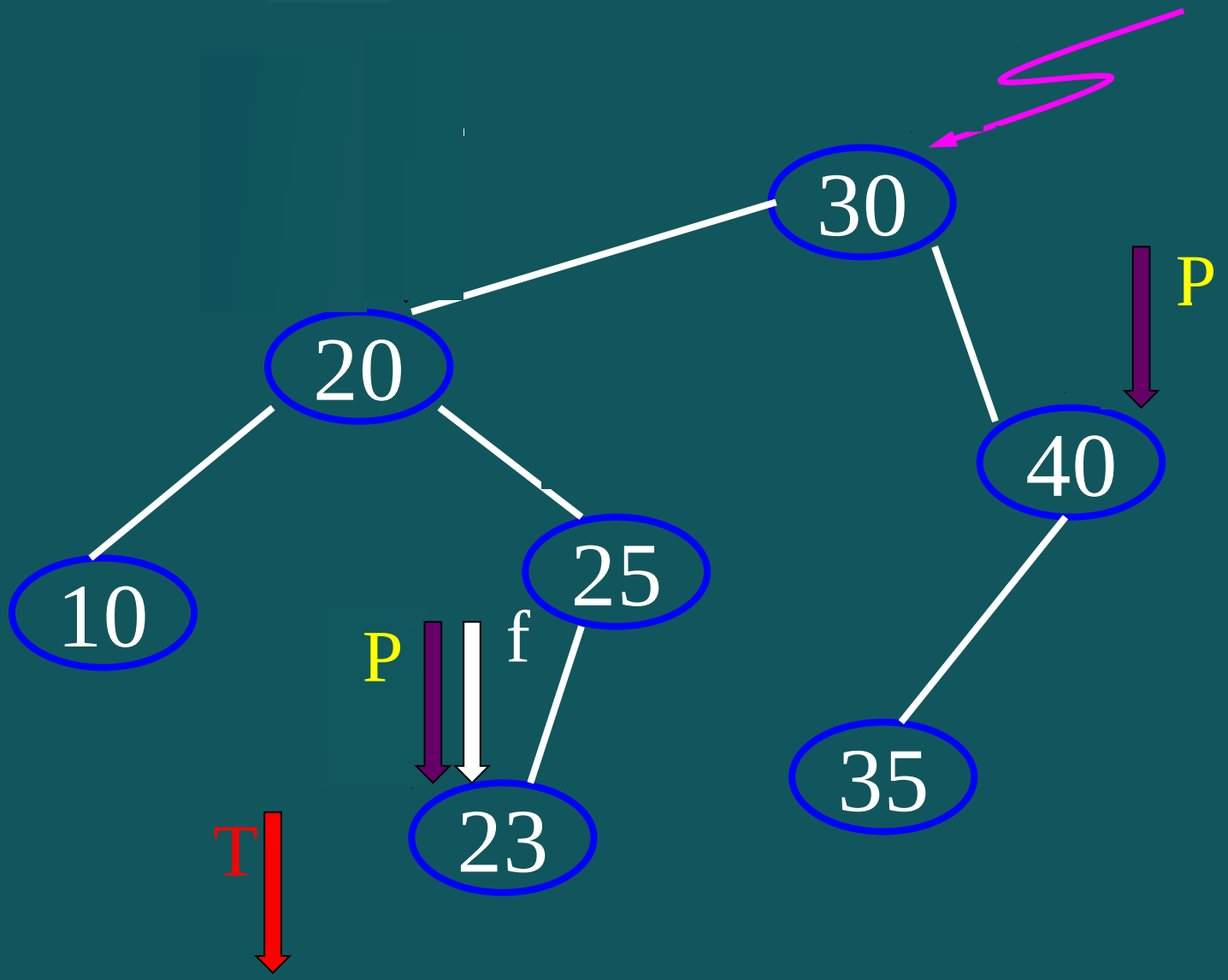
算法描述如下：

```
Status SearchBST (BiTree T, KeyType key,
                                  BiTree f, BiTree &p ) {
 // 在根指针 T 所指二叉排序树中递归地查找其关键字等
于 key 的数据元素，若查找成功， 则返回指针 p 指向该数
据元素的结点，并返回 函数值为 TRUE； 否则表明查找不
成功，返回 指针 p 指向查找路径上访问的最后一个结点，
并返回函数值为FALSE， 指针 f 指向当前访问 的结点的双
亲，其初始调用值为NULL
… … … …
} // SearchBST
```

算法9.5(b)

```
if (!T)
        { p = f;  return FALSE; }  // 查找不成功
else  if ( EQ(key, T->data.key) )
        { p = T;  return TRUE; }  // 查找成功
else  if ( LT(key, T->data.key) )
         SearchBST (T->lchild, key, T, p );
                                  // 在左子树中继续查找
else
         SearchBST (T->rchild, key, T, p );
                                  // 在右子树中继续查找
```

设 key = 22



```
Status Insert BST(BiTree &T, ElemType e )
{
  // 当二叉排序树中不存在关键字等于 e.key 的
  // 数据元素时，插入元素值为 e 的结点，并返
  // 回 TRUE；否则，不进行插入并返回FALSE
   if (!SearchBST ( T, e.key, NULL, p ))
     {      …    …              }
   else return FALSE;
} // Insert BST
```

算法 9.6

```
s = (BiTree) malloc (sizeof (BiTNode));
                    // 为新结点分配空间
s->data = e;
s->lchild = s->rchild = NULL;
  if （ !p ） T = s;     // 插入 s 为新的根结点
  else   if ( LT(e.key, p->data.key) )
        p->lchild = s;       // 插入 *s 为 *p 的左孩子
  else  p->rchild = s;     // 插入 *s 为 *p 的右孩子
   return TRUE;     // 插入成功
```

例如，设查找的关键字序列为{ 45,24,53,45,12,24,90 } ，则生成的二叉排序树如图9.8所示。

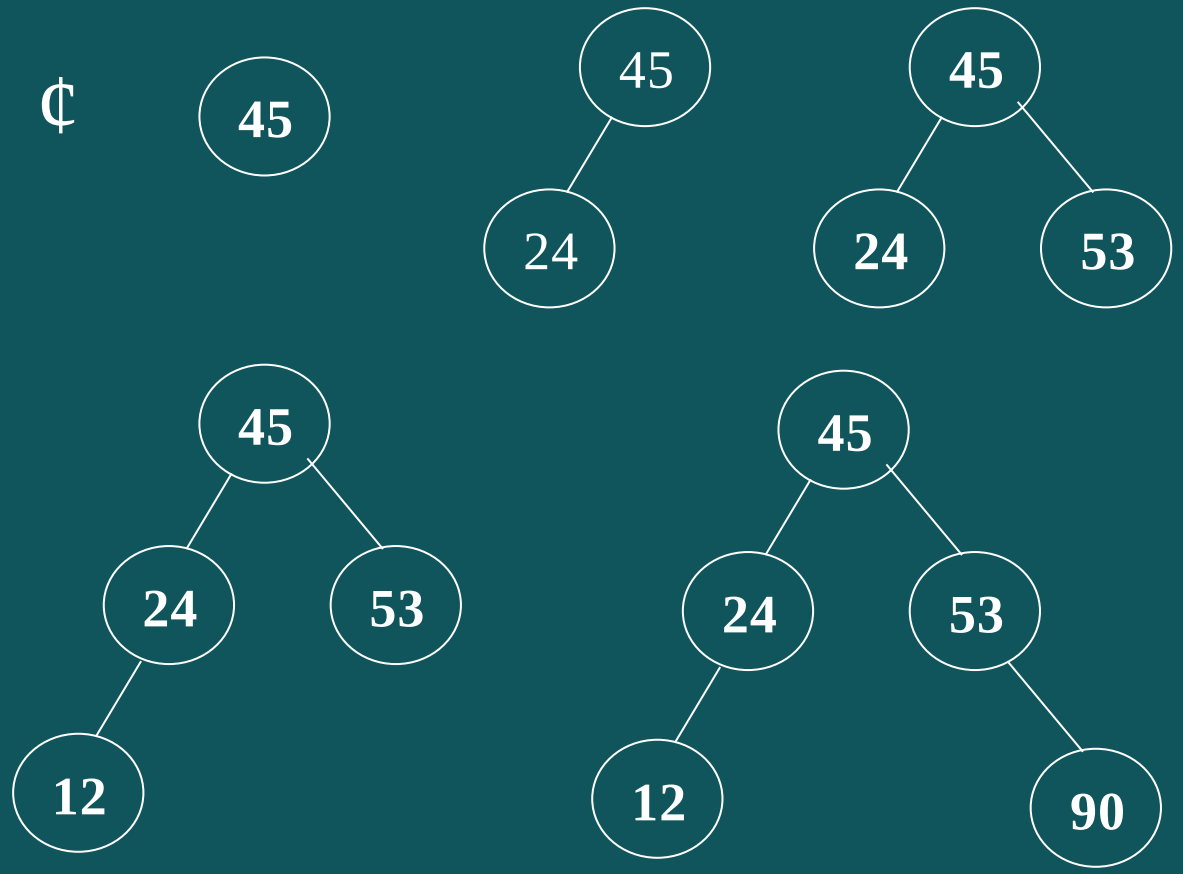


图9.8 二叉排序树的构造过程

4．二叉排序树的删除算法

和插入相反，删除在**查找成功**之后进行，并且要求在删除二叉排序树上某个结点之后，**仍然保持二叉排序树的特性。**

可分三种情况讨论：

（1）被删除的结点是叶子；

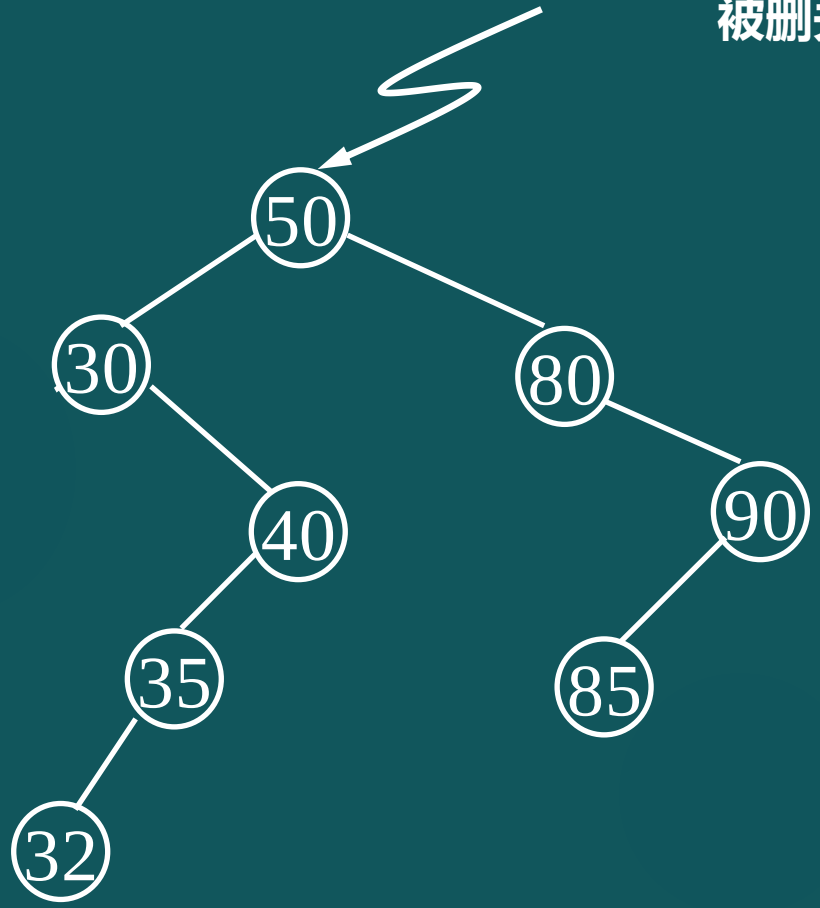
（2）被删除的结点只有左子树或者只有右子树；

（3）被删除的结点既有左子树，也有右子树。

（1）被删除的结点是**叶子结点**

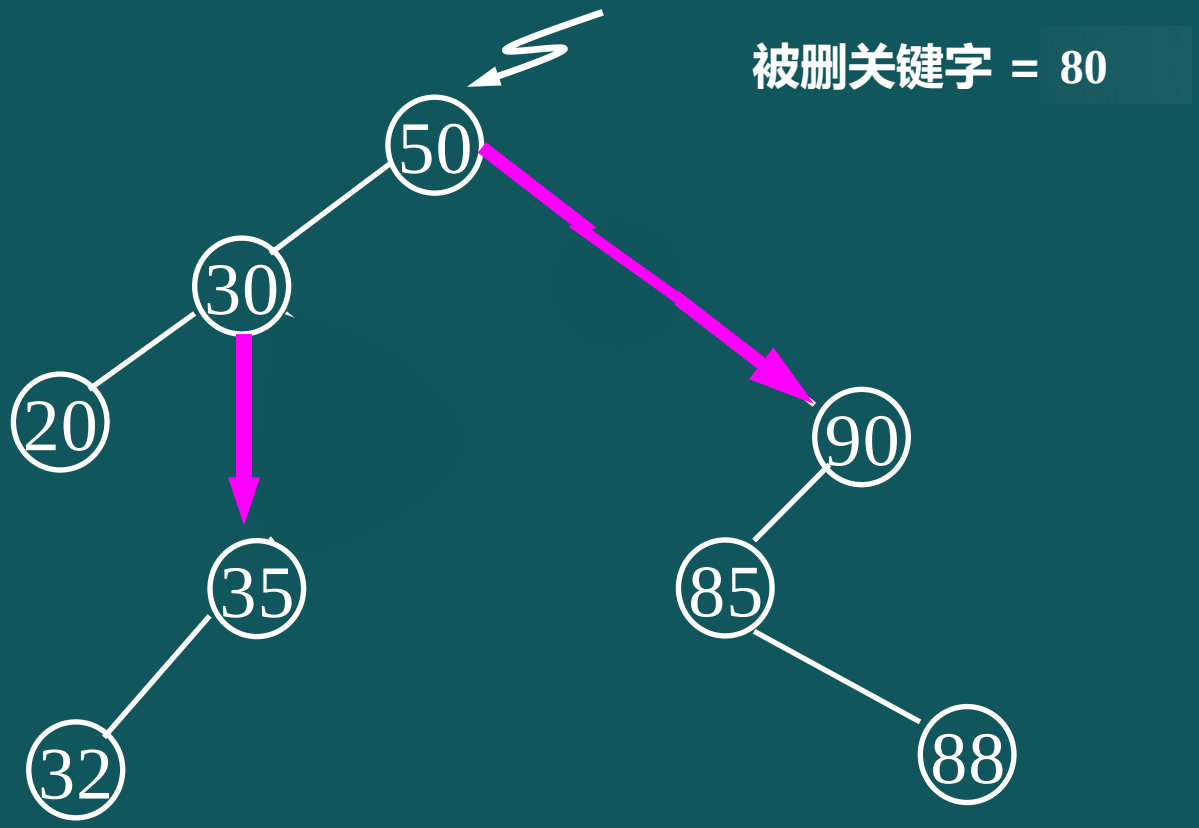
例如:

被删关键字 = 88



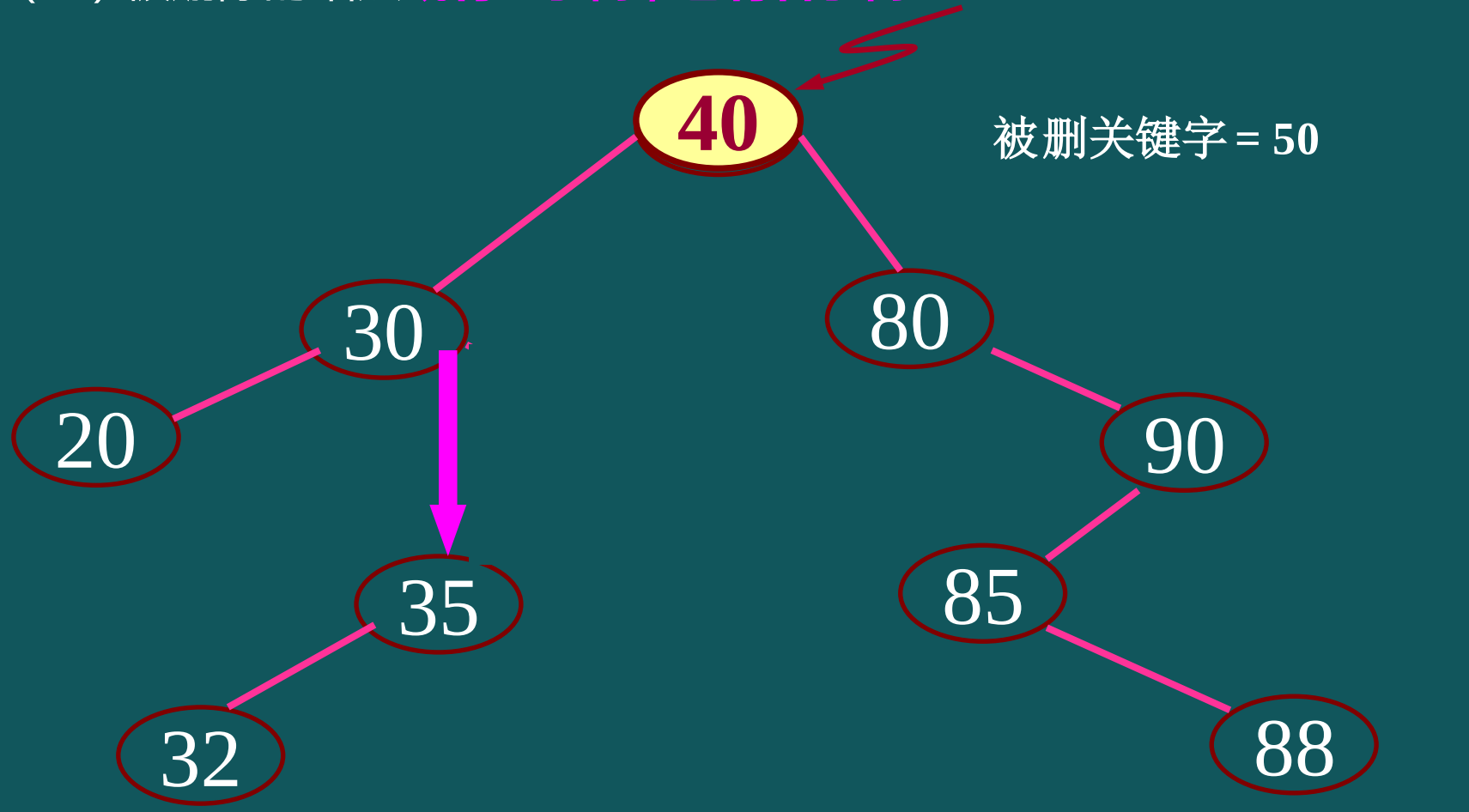
其双亲结点中相应指针域的值改为“空”

（2）被删除的结点**只有左子树**，或者**只有右子树**

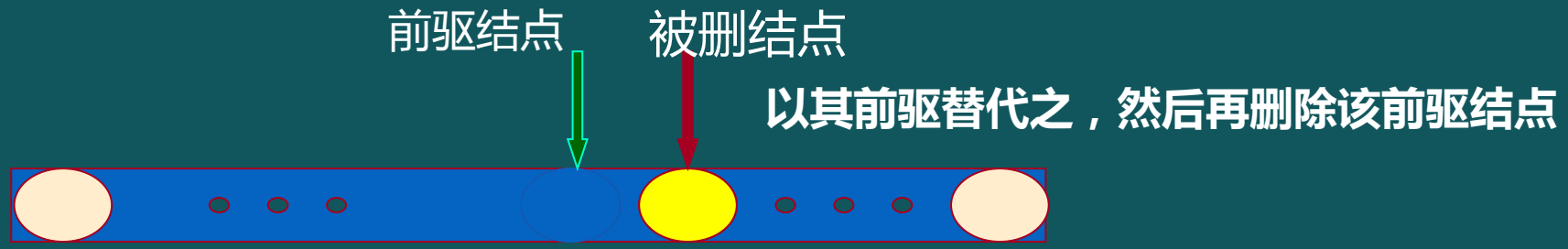


其双亲结点的相应指针域的值改为 “指向被删除结点的左子树或右子树”。

（3）被删除的结点**既有左子树，也有右子树**



被删关键字 = 50



以其前驱替代之，然后再删除该前驱结点

算法描述如下：

```
Status DeleteBST (BiTree &T,  KeyType key ) {
  // 若二叉排序树 T 中存在其关键字等于 key 的数据元素，则
 //删除该数据元素结点，并返回函数值 TRUE，否则返回函
 //数值 FALSE
  if (!T)  return FALSE;  // 不存在关键字等于key的数据元素
  else {
        if ( EQ (key, T->data.key) ) return Delete (T);
            //找到关键字等于key的数据元素
        else if ( LT (key, T->data.key) ) return DeleteBST ( T->lchild,
key );  //继续在左子树中进行查找
       else return DeleteBST ( T->rchild, key );
                    //继续在右子树中进行查找
} // DeleteBST
```

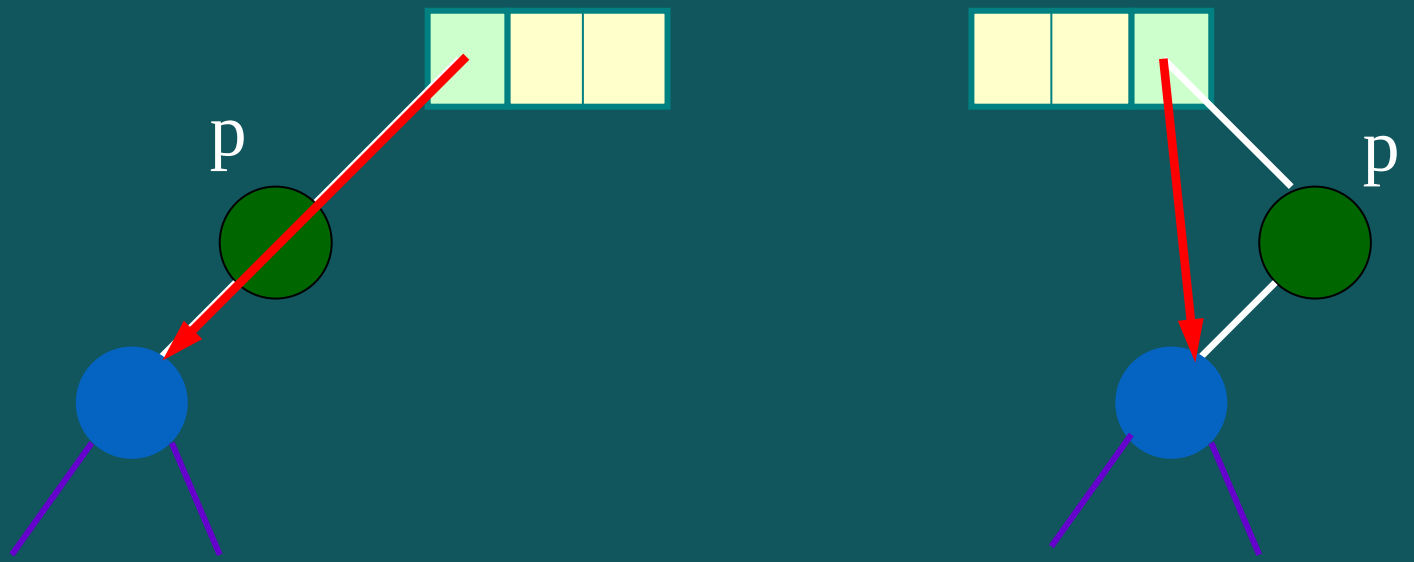
算法 9.7

其中**删除操作**过程如下所描述：

```
void Delete ( BiTree &p ){
  // 从二叉排序树中删除结点 p，并重接它的左
  //子树或右子树    ...
  if (!p->rchild)  {        } //右子树空则只需重
                          ...
                              //接它的左子树
            ...
  else if (!p->lchild) {       } //只需重接右子树
  else {          } //左右子树均不空
} // Delete
```

// 右子树为空树则只需重接它的左子树

q = p; p = p->lchild; free(q);



// 左子树为空树只需重接它的右子树

q = p; p = p->rchild; free(q);

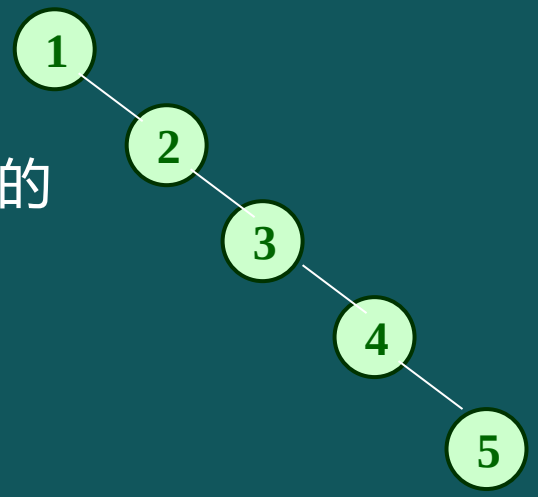
```
        // 左右子树均不空
q = p;  s = p->lchild;
while (!s->rchild) { q = s;  s = s->rchild; }
                        // s 指向被删结点的前驱
p->data = s->data;
if (q != p )  q->rchild = s->lchild;
else  q->lchild = s->lchild;
                        // 重接*q的左子树
free(s);
```

5．二叉排序树的查找分析

对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的 ***ASL*** 值，显然，由值相同的 ***n*** 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长 度的值不同，甚至可能差别很大。

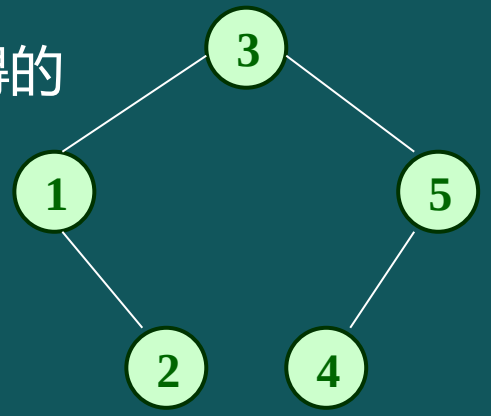
例如：

由关键字序列 **1，2，3，4，5**构造而得的二叉排序树，



ASL =（1+2+3+4+5）/ 5 = 3

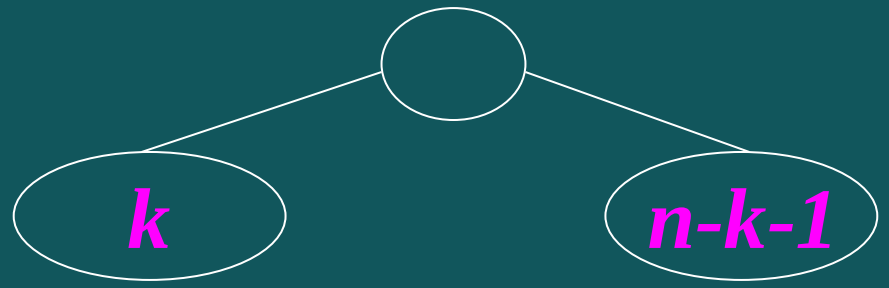
由关键字序列 **3，1，2，5，4**构造而得的二叉排序树，



ASL =（1+2+3+2+3）/ 5 = 2.2

一般情况下:

不失一般性，假设长度为 ***n*** 的序列中有 ***k*** 个关键字**小于**第一个关键字，则必有 ***n-k-1*** 个关键字**大于**第一个关键字,由它构造的二叉排序树：



平均查找长度是 *n* 和 *k* 的函数

$$P(n) = 2\frac{n+1}{n}\log n + C$$

二、平衡二叉树

平衡二叉树又称AVL树，是二叉排序树的另一种形式。它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。

结点的平衡因子：

该结点的左子树高度减去它的右子树高度。即：

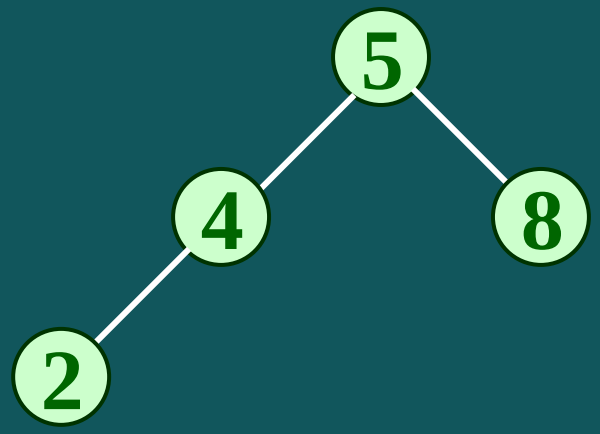
$$h_L - h_R$$

平衡二叉树的特点为：

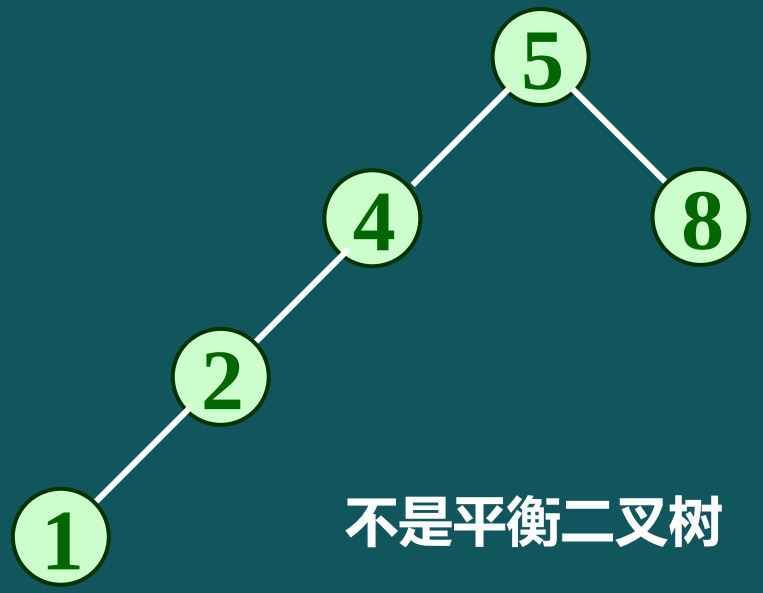
树中每个结点的左、右子树深度之差的绝对值不大于1。即

$$|h_L - h_R| \leq 1$$

例如:



是平衡二叉树

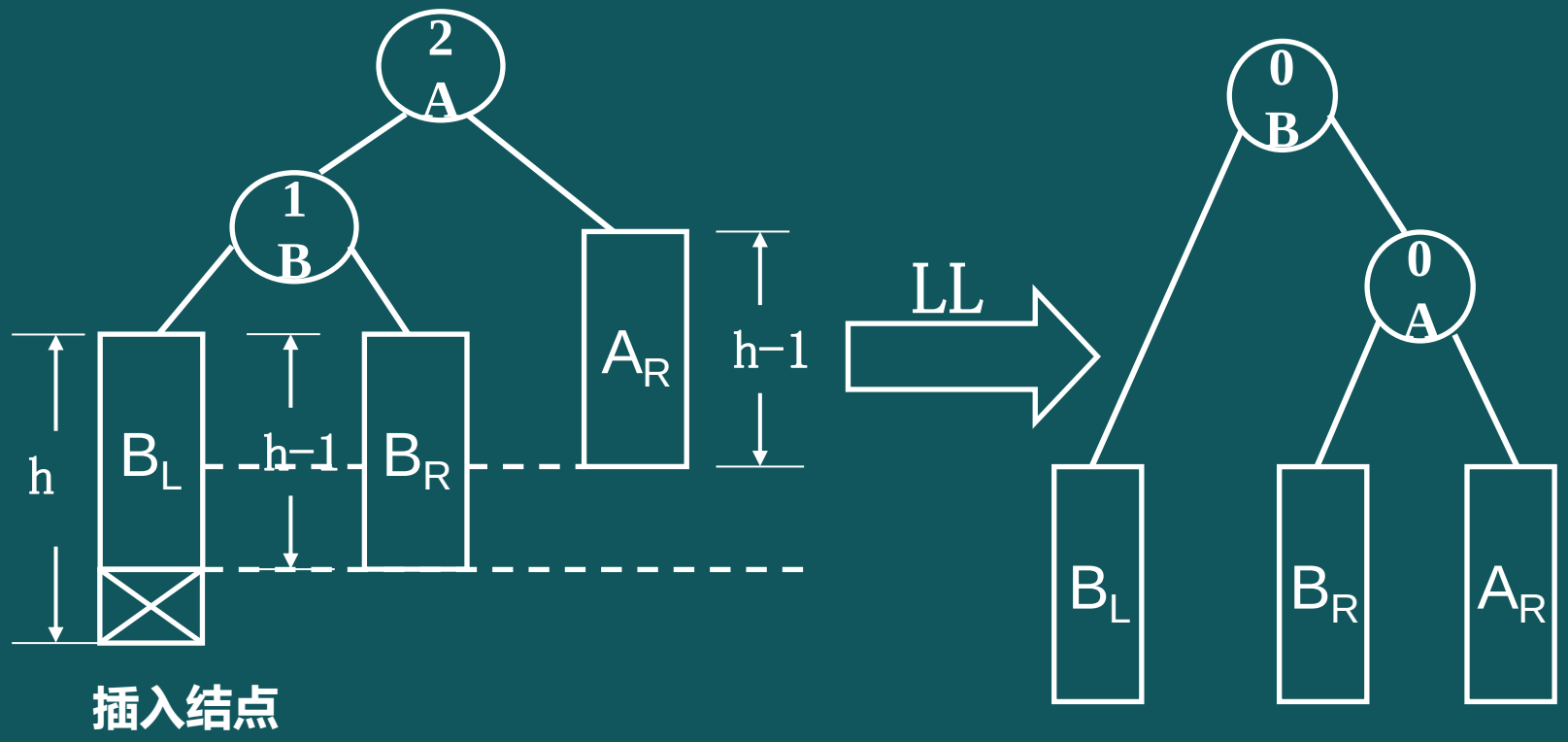


不是平衡二叉树

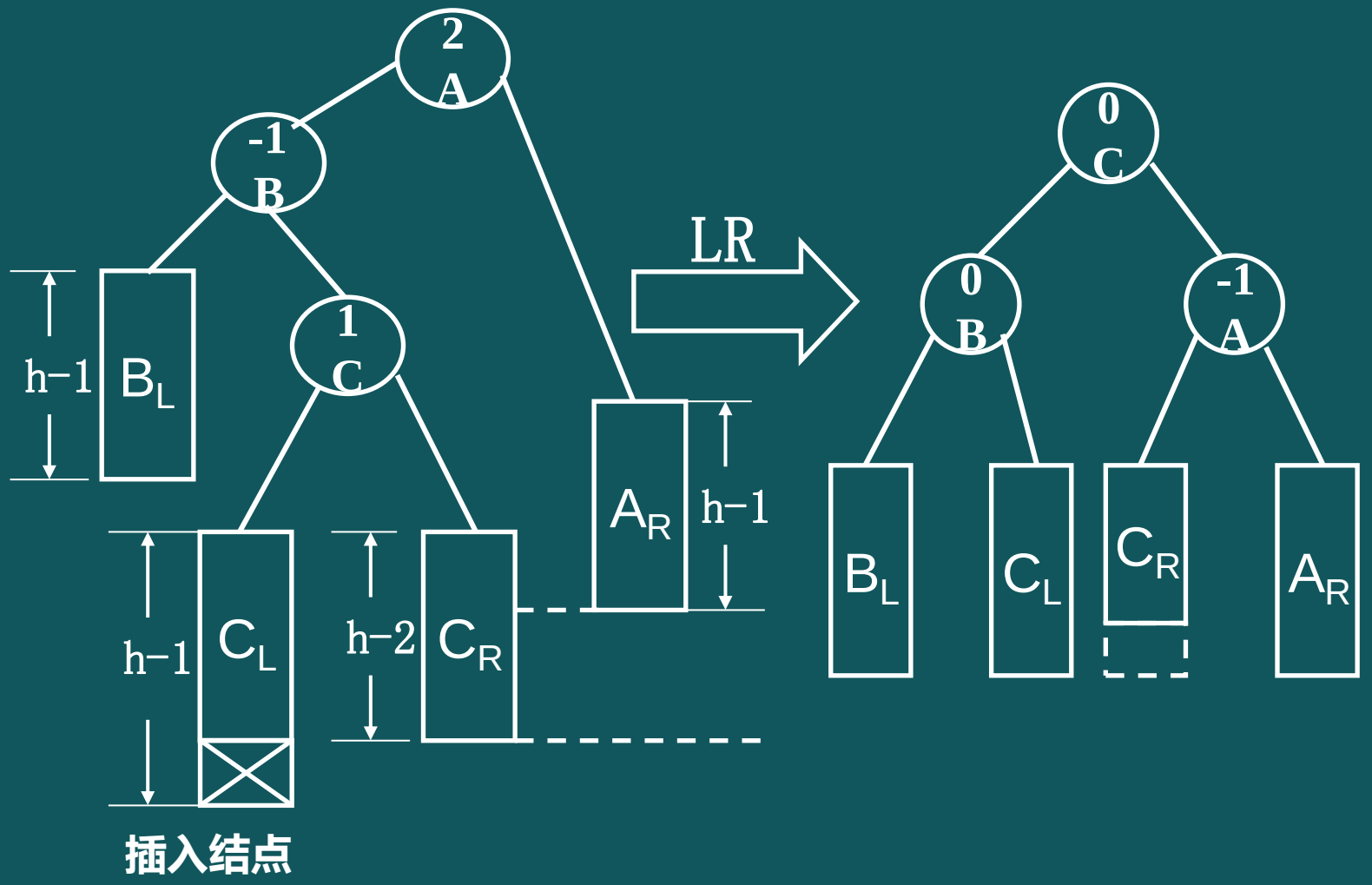
构造二叉平衡（查找）树的方法是：**在插入过程中，采用平衡旋转技术。**

一般情况下，假设由于在二叉排序树上插入结点而失去平衡的最小子树根结点的指针为A（即A是离新插入结点最近，且平衡因子绝对值超过1的祖先结点），则失去平衡后进行调整的规律可归纳为下列四种情况：LL型、LR型、RR型、LR型。

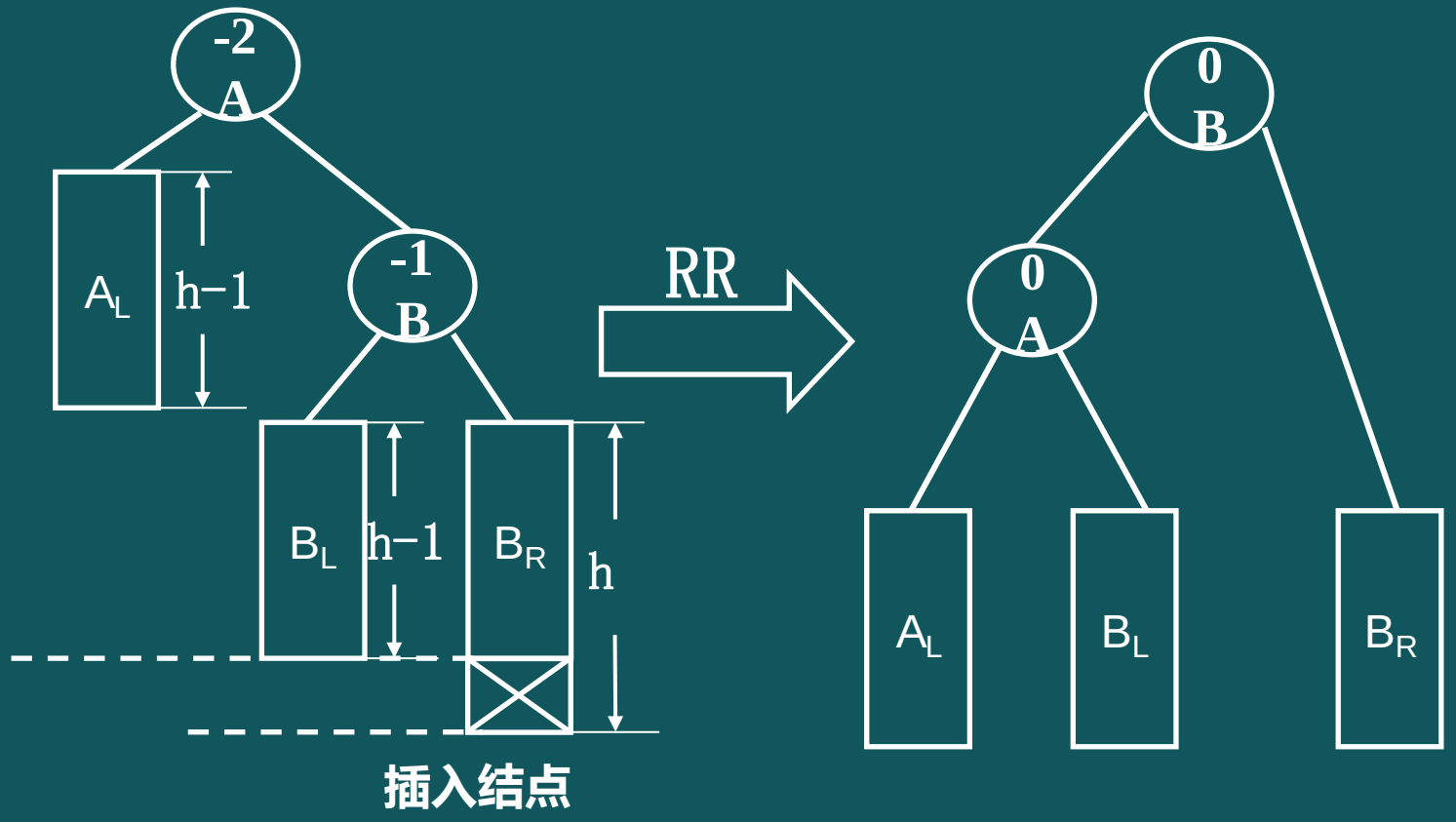
• LL型：即新插入的结点在 A 的左子树的左子树上。此时应选择 A 的左子树的根结点 B，然后进行调整；



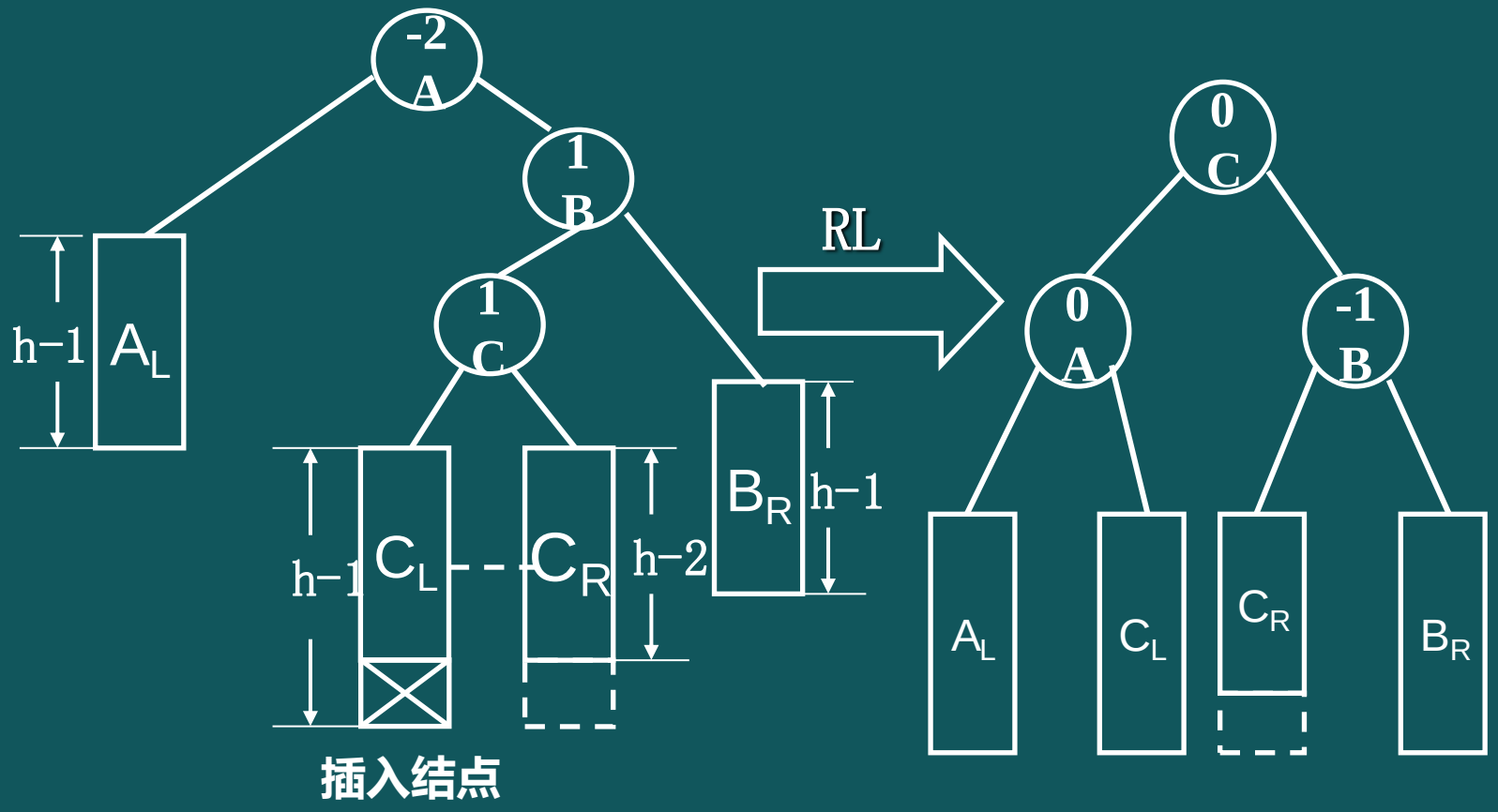
• LR型：即新插入的结点在A的左子树的右子树上。此时应选择 A 的左子树的根结点B ，和 B 的右子树的根结点 C ，然后进行调整；



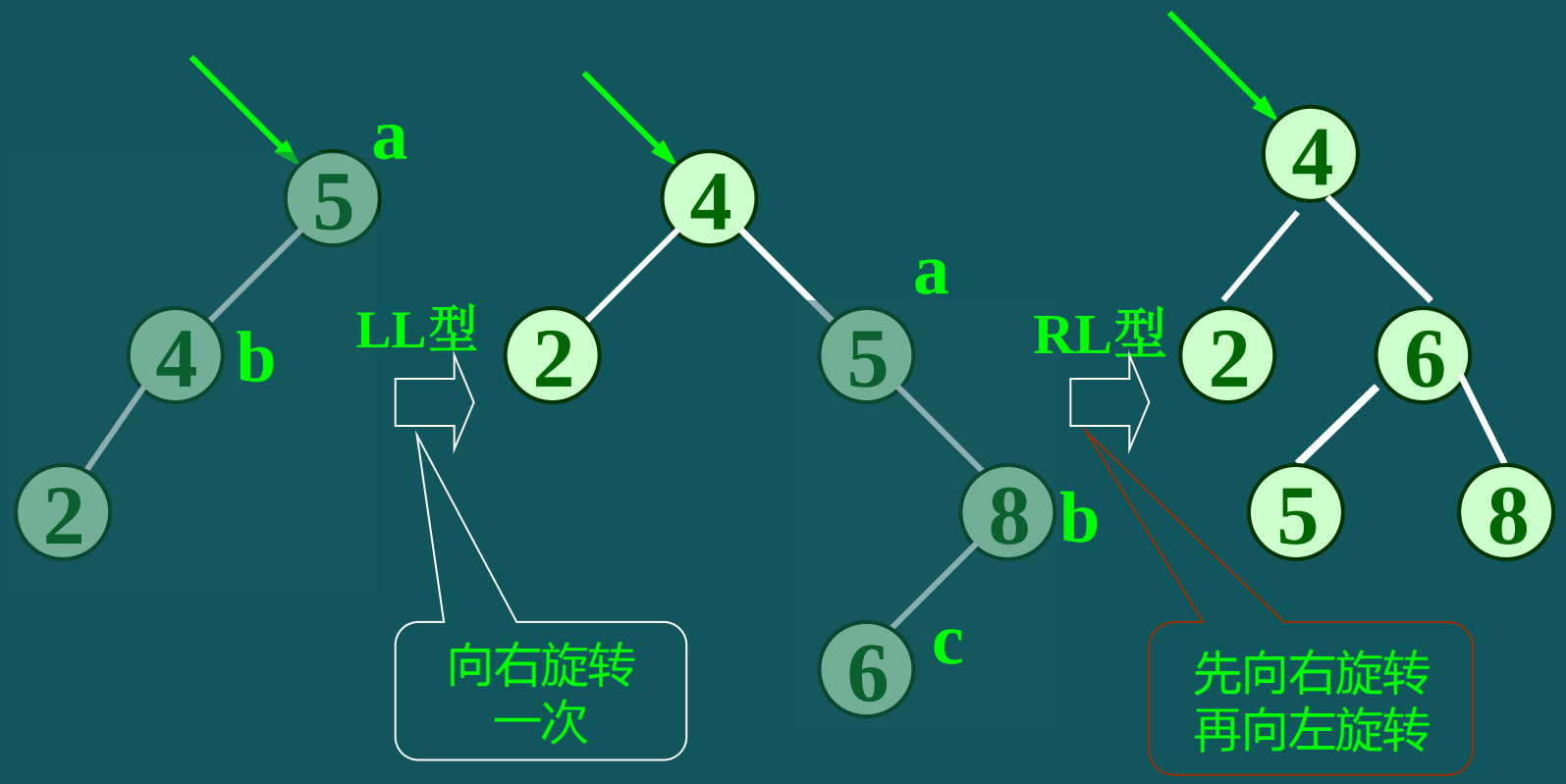
• RR型：即新插入的结点在 A 的右子树的右子树上。此时应选择A的右子树的根结点 B，然后进行调整；



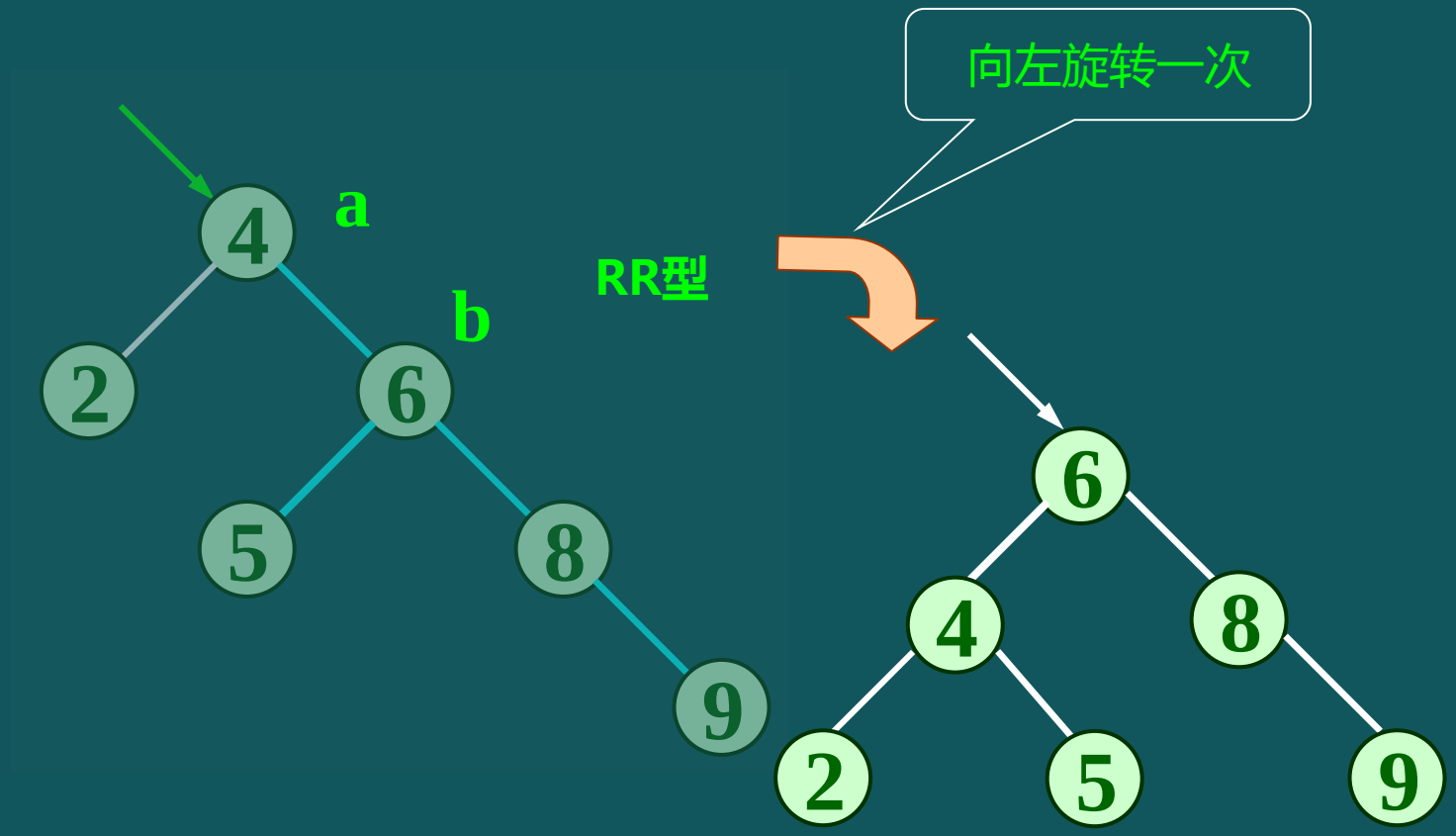
• RL型：即新插入的结点在 A 的右子树的左子树上。此时应选择A的右子树的根结点 B ，和 B 的左子树的根结点 C ，然后进行调整；



例如:依次插入的关键字为5, 4, 2, 8, 6, 9



继续插入关键字 9



二叉排序树的类型定义为：

```
typedef struct BSTNode {
    ElemType     date;
     int                 bf;    //结点的平衡因子
     struct BSTNode * lchild, * rchild;
                                 //左、右孩子指针
 } BSTNode, *BSTree;
```

```
void R_Rotate ( BSTree &p) {
    //对以*p为根的二叉排序树作右旋处理，处理之后p指向新的根
结点，即旋转处理之前的左子树的根结点
    lc = p->lchild;   //lc指向的*p的左子树根结点
    p->lchild= lc->rchild;
    //lc的右子树挂接为*p的左子树
    lc->rchild=p; p=lc;            //p指向新的根结点
}// R_Rotate
```

算法 9.9

```
void L_Rotate ( BSTree &p) {
    //对以*p为根的二叉排序树作左旋处理，处理之//后p指向新的
根结点，即旋转处理之前的右子树//的根结点
    rc = p->rchild;
    //rc指向的*p的右子树根结点
    p->rchild= rc->lchild;
    //rc的左子树挂接为*p的右子树
    rc->lchild=p; p=rc;            //p指向新的根结点
}// L_Rotate
```

算法 9.10

```
#define LH+1      //左高
#define EH  0     //等高
#define RH-1      //右高
 Status InsertAVL (BSTree &T, ElemType e, Boolean &taller)
{ //若在平衡的二叉排序树T中不存在和e有相同关键字的结点，则
插入一个数据元素为e的新结点，并返回1，否则返回0。若因插入
而使二叉排序树失去平衡，则作平衡旋转处理，布尔变量taller反
映T长高与否
   if (!T) {//插入新结点，树“长高”，置taller为TRUE
    T = (BSTree) malloc (sizeof (BSTNode)); T->data = e;
    T->lchild=T->rchild=NULL; T->bf=EH; taller=TRUE;
 }
```

算法 9.11

```
  else {
    if (EQ (e.key, T->data.key))  //树中已存在和e有相同关键字的结点
      { taller = FALSE;  return 0;  }                    //则不再输入
    if (LT (e.key, T->data.key)) {  //应继续在*T的左子树中进行搜索
      if (!InsertAVL (T->lchild, e, taller))  return 0;          //未插入
      if (taller)                  //已插入到*T的左子树中且左子树“长高”
        switch (T->bf) {                                  //检查*T的平衡
度
          case LH;        //原本左子树比右子树高，需要做左平衡处理
            LeftBalance (T);  taller = FALSE; break;
          case EH;    //原本左、右子树等高，现因左子树增高而使树增高
            T->bf = LH; taller = TRUE; break;
          case RH;        //原本右子树比左子树高，现左、右子树等高
            T->bf = EH; taller = FALSE; break;
      }//switch (T->bf)
```

```
}//if
else {        //应继续在*T的右子树中进行搜索
   if (!InsertAVL (T->rchild, e, taller)) return 0;  //未插入
   if (taller)       //已插入到*T的右子树中且右子树“长高”
        switch (T->bf) {          //检查*T的平衡度
          case LH; //原本左子树比右子树高，现左、右子树等高
            T->bf = EH; taller = FALSE; break;
          case EH;     //原本左、右子树等高，现因右子树增高而使树
增高
            T->bf = RH; taller = TRUE; break;
          case RH; //原本右子树比左子树高，需要做右平衡处理
            RightBalance (T);  taller = FALSE; break;
        }//switch (T->bf)
      }//else
    }//else
    return 1;
}//InsertAVL
```

```
void LeftBalance ( BSTree &T ) {
    //对以指针T所指结点为根的二叉树作左平衡旋转处理，本算法结束时，指针T指向新的根结点
    lc = T->lchild;              //lc指向*T的左子树根结点
    switch (lc->bf) {            //检查*T的左子树的平衡度，并作//相应平衡处理
        case LH;                 //新结点插入在*T的左孩子的左子//树，要作单右旋处理
            T->bf = lc->bf = EH;
            R_Rotate(T);   break;
        case RH;                 //新结点插入在*T的左孩子的右子//树，要作双旋处理
            rd = lc->rchild;  //rd指向*T的左孩子的右子树根
```

算法 9.12

```
        switch (rd->bf)     { //修改*T及其左孩子的平衡//因子
            case LH: T->bf = RH;   lc->bf = EH; break;
            case EH: T->bf = lc->bf = EH; break;
            case RH: T->bf = EH;   lc->bf = LH; break;
        }// switch (rd->bf)
        rd->bf = EH;
        L_Rotate(T->lchild);  //对*T的左子树做左旋平//衡处理
        R_Rotate(T);   //对*T的做右旋平衡处理
    }// switch (lc->bf)
}// LeftBalance
```

9.2.2 B -树和B+树

1．B-树的定义

B-树是一种**平衡**的**多路查找**树：

一棵 ***m*** 阶的B-树，或为空树，或为满足下列特性的m叉树：

（1）所有非叶结点均至少含有$\lceil m/2 \rceil$棵子树，至多含有 ***m*** 棵子树；

（2）根结点或为叶子结点，或至少含有两棵子树；

（3）所有非终端结点含有下列信息数据：

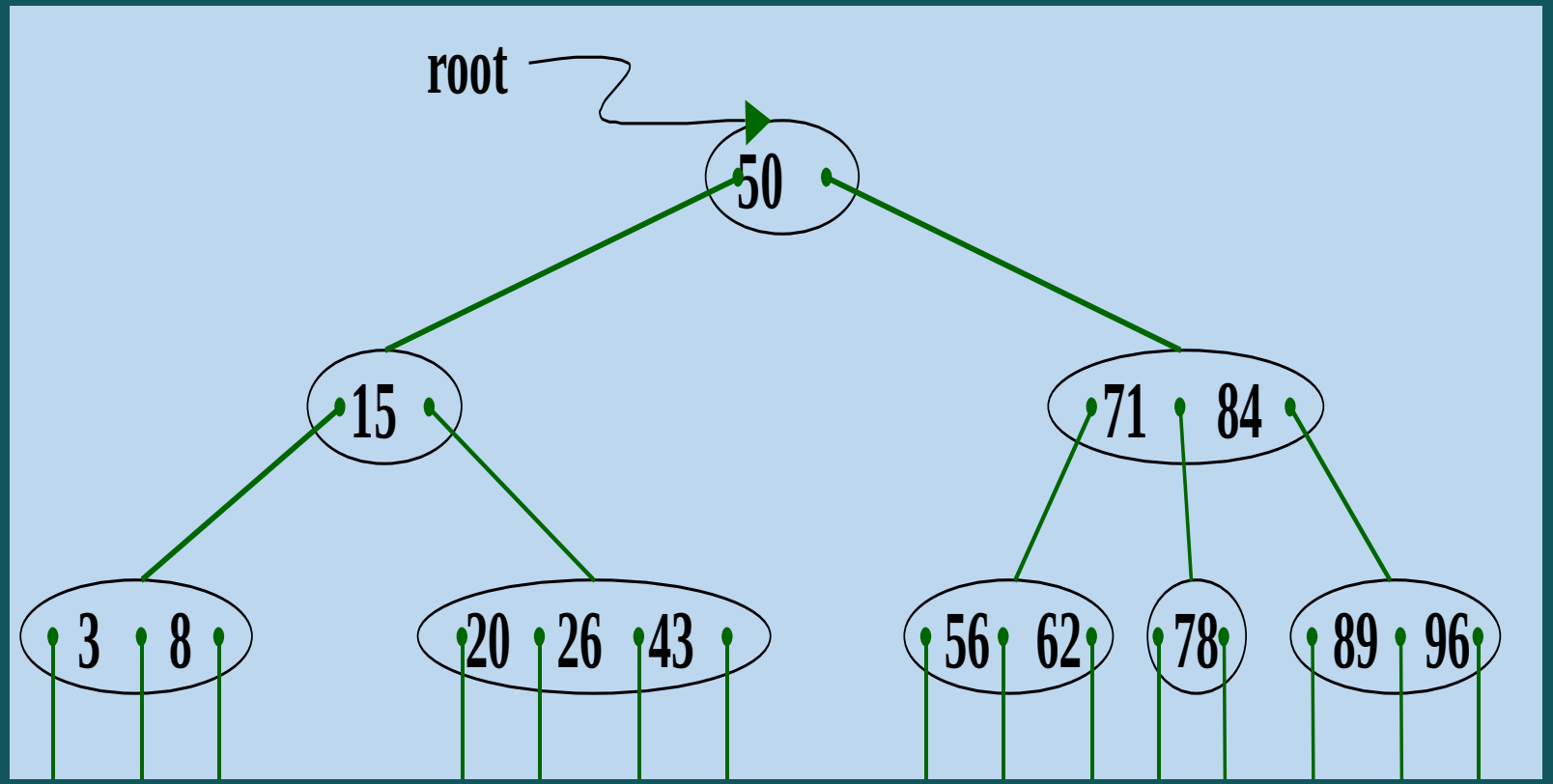
（n，A_0，K_1，A_1，K_2，A_2，…K_n，A_n）

其中：K_i为**关键字**，且均自小至大有序排列，即：$K_1 < K_2 < \ldots < K_n$

A_i**为**指向子树根结点的指针，且指针A_{i-1}所指子树上所有关键字均**小于**K_i ； A_n 所指子树上所有关键字均**大于**K_n ；

（4）树中所有叶子结点均不带信息，且在树中的同一层次上；

例如，下图是一棵4阶的B-树



B-树结构的C语言描述如下:

```
#define   m   3              //B-树的阶，暂设为3
typedef struct BTNode {
  int  keynum;        // 结点中关键字个数，即结点大小
  struct BTNode  *parent;
                      // 指向双亲结点的指针
  KeyType   key[m+1]; // 关键字向量（0号单元不用）
  struct BTNode  *ptr[m+1]; // 子树指针向量
  Record       *recptr[m+1];  // 记录指针向量（0号单元不用）
} BTNode, *BTree; // B-树结点和B-树的类型
```

2.查找过程：

从根结点出发，沿指针**搜索结点**和**在结点内进行**顺序（或折半）**查找** 两个过程交叉进行。

若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；

若查找不成功，则返回插入位置。

假设返回的是如下所述结构的记录：

```
typedef struct {
  BTNode  *pt;     // 指向找到的结点的指针
  int  i;       // 1..m，在结点中的关键字序号
  int  tag;   // 标志查找成功(=1)或失败(=0)
} Result;              // 在B-树的查找结果类型
```

```
Result SearchBTree(BTree T, KeyType K) {
  // 在m 阶的B-树 T 中查找关键字 K, 返回
  // 查找结果 (pt, i, tag)。若查找成功，则
  // 特征值 tag=1, 指针 pt 所指结点中第 i 个
  // 关键字等于 K； 否则特征值 tag=0, 等于
  //  K 的关键字应插入在指针 pt 所指结点
  // 中第 i 个关键字和第 i+1个关键字之间
{
          … …
} // SearchBTree1
```

算法 9.13

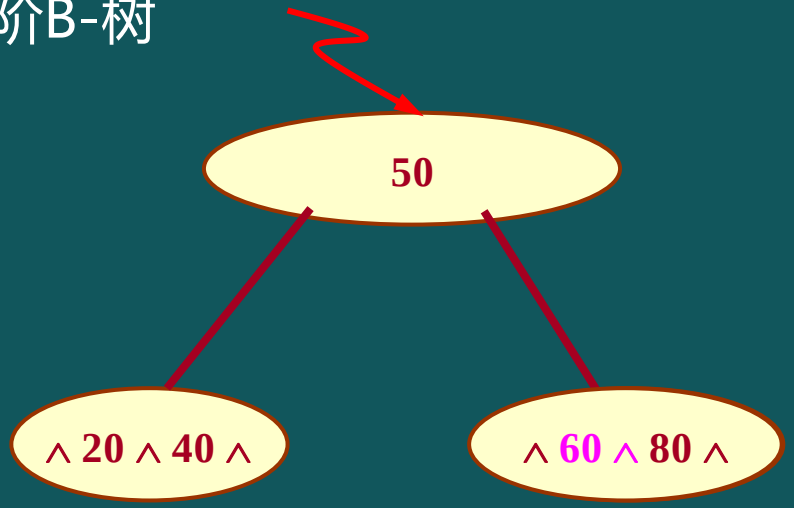
```
p=T;  q=NULL;  found=FALSE;  i=0;
//初始化，p指向待查接点，q指向p的双亲
while (p && !found) {
   i=Search(p, K);     // 在p->key[1..keynum]中查找i ，
                       //使得p->key[i]<=K<p->key[i+1]
   if (i>0 && p->key[i]==K)  found=TRUE; //找到待查
                                         // 关键字
   else { q=p;   p=p->ptr[i]; }
}
if (found)  return (p,i,1);       // 查找成功
else return (q,i,0);              // 查找不成功，返回k的插
                                  //入位置信息
```

3．插入

在查找不成功之后，需进行插入。显然，关键字**插入**的**位置**必定在**最下层的非叶结点**，有下列几种情况：

- 1）插入后，该结点的关键字个数n<m，不修改指针; 例如

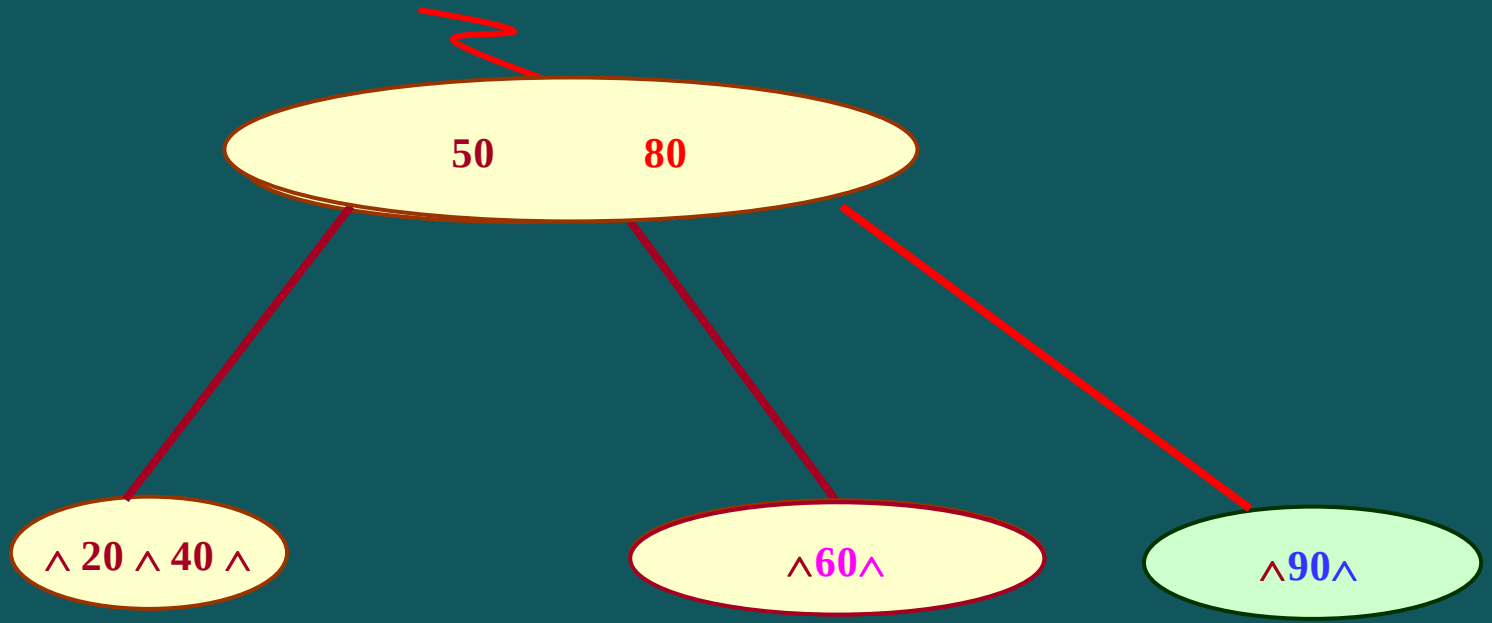
例如:下列为 3 阶B-树



插入关键字 = **60**,

- 2）插入后，该结点的关键字个数 n=m，则需进行“结点分裂”，令 s =⌈m/2⌉，在原结点中保留（A_0，K_1，……，K_{s-1}，A_{s-1}）；建新结点（A_s，K_{s+1}，……，K_n，A_n）；将（K_s，p）插入双亲结点；

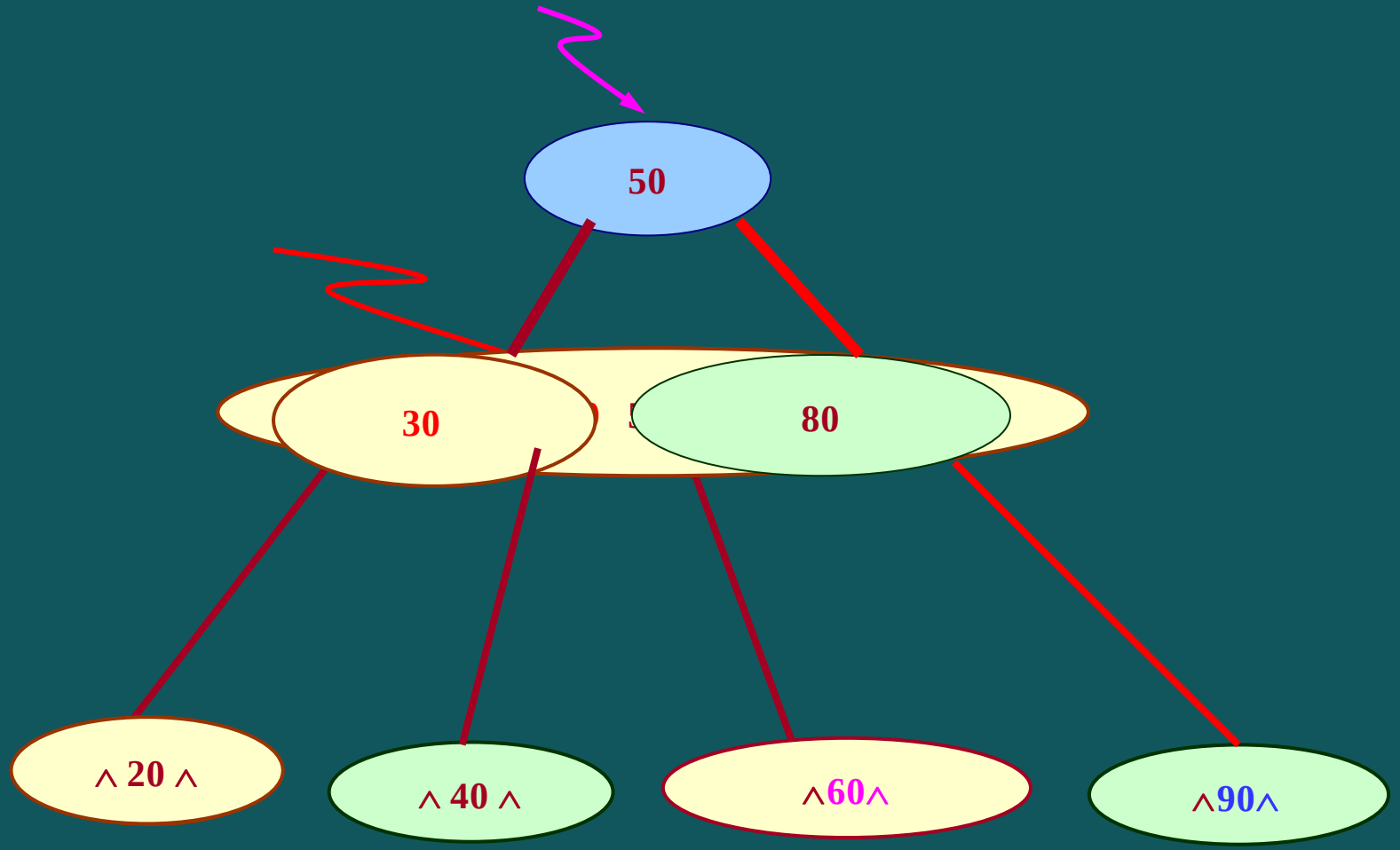
例如:下列 3 阶B-树



插入关键字 = **90**,

3）若双亲为空，则建新的根结点。

例如:下列 3 阶B-树



插入关键字 = **30**,

详细过程，见教材 p242—p245 图9.16

```
Status InsertBTree (BTree &T, KeyType K, BTree q, int i ) {
    //在m阶B-树T上结点*q的key[i]和key[i+1]之间插入关键字K
若引起结点过大，则沿双亲链进行必要的结点分裂调整，使T仍是
m阶B-树。
    x = K;  ap = NULL;  finished = FALSE;
    while (q && !finished) {
        Insert (q, i, x, ap) ;
        //将x和ap分别插入到q->key[i+1]和q->ptr[i+1]
        if (q->keynum<m)  finished = TRUE;
        //插入完成
```

算法 9.14

```
else {
            s=[m/2];  split (q, s, ap);  x = q->key[s];
 //将q->key[s+1..m], q->ptr[s..m]和q>recptr[s+1..m]
//移入新结点*ap
            q = q->parent;
            if (q)  i = Search (q, x);
            //在双亲结点*q中查找x的插入位置
          }//else
       }//while
       if (!finished)       //T是空树(参数q初值为NULL)或者根
            //结点已分裂为结点*q和*ap
          NewRoot (T, q, x, ap);       //生成含信息(T, x, ap)
          //的新的根结点*T，原T和ap为子树指针
       return OK;
  }// InsertBTree
```

4．删除

和插入的考虑相反，首先必须找到待删关键字所在结点，并且要求删除之后，结点中关键字的个数不能小于⌈m/2⌉-1，否则，要从其左(或右)兄弟结点“借调”关键字，若其左和右兄弟结点均无关键字可借(结点中只有最少量的关键字)，则必须进行结点的“合”

假若所删关键字为非终端结点中的Ki,则可以指针Ai所指子树中的最小关键字Y替代Ki，然后在相应的结点中删去Y。

详细过程，见教材 p245 图9.17

6. B^+树

是B-树的一种变型。

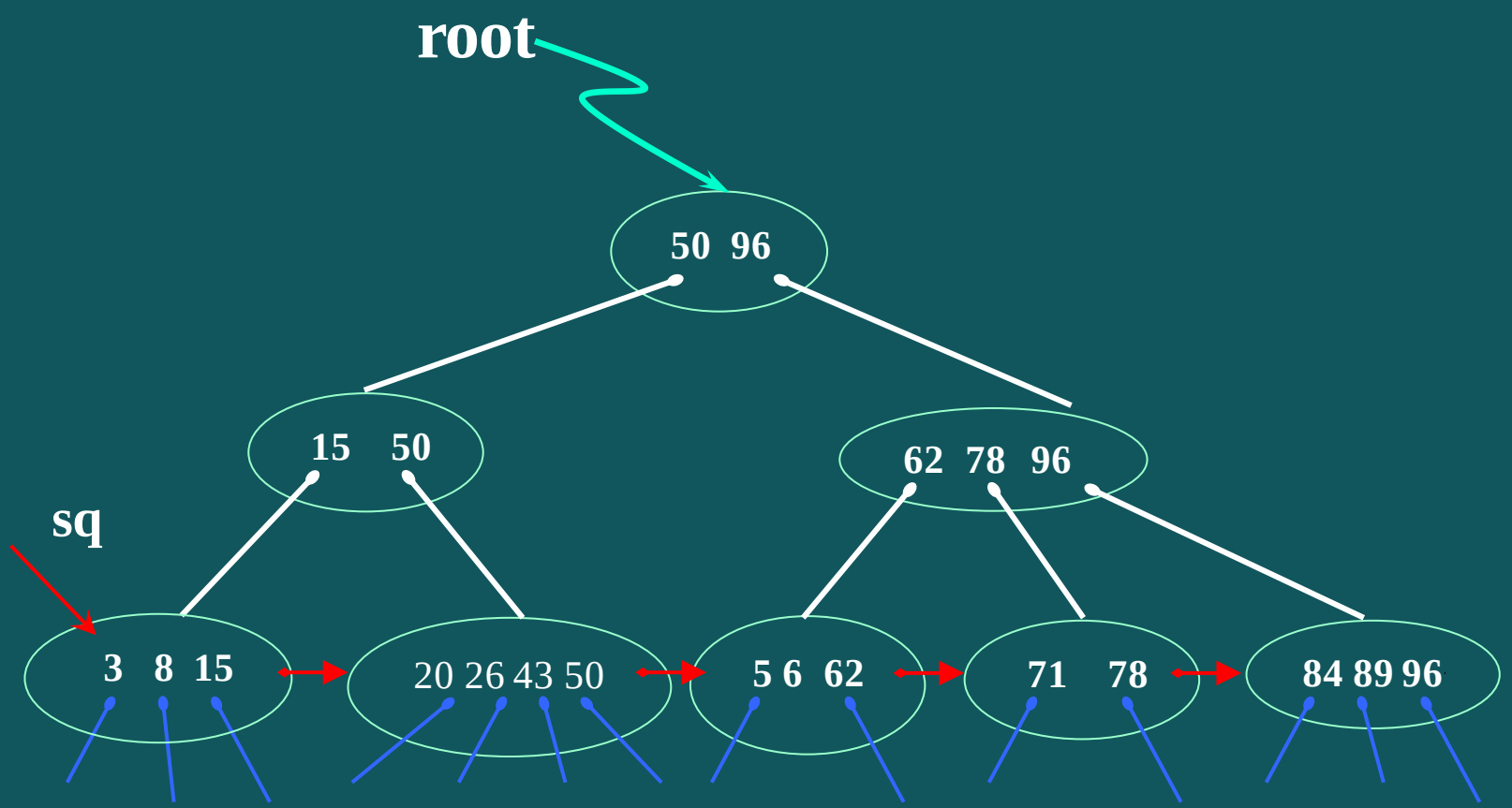
（1）B^+树的结构特点：

※ 每个叶子结点中含有 n 个关键字和 n 个指向记录的指针；并且，所有**叶子**结点彼此相**链接**构成一个有序链表，其头指针指向含最小关键字的结点；

※ 每个非叶结点中的关键字 K_i即为其相应指针 A_i所指子树中关键字的最大值；

※ 所有叶子结点都处在同一层次上，每个叶子结点中关键字的个数均介于 $\lceil m/2 \rceil$和 m 之间。

例如，下图是一棵4阶的B+树



（2）查找过程

※ 在 B^+ 树上，既可以进行缩小范围的查找（自顶向下），也可以进行顺序查找（在最底层，自左向右）；

※ 在进行缩小范围的查找时，不管成功与否，都必须查到叶子结点才能结束；

※ 若在结点内查找时，给定值$\leqslant K_i$，则应继续在 Ai 所指子树中进行查找。

（3）插入和删除的操作

类似于B-树进行，即必要时，也需要进行结点的“分裂”或“归并”。