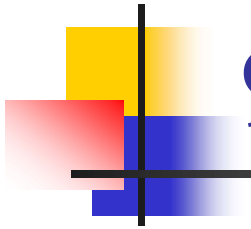


9.2 动态查找表

- 动态查找表的主要内容:
  - 二叉排序树和平衡二叉树
  - B-树和B+树

- 说明：动态查找表除了“查找”操作外，允许进行“插入”和“删除”

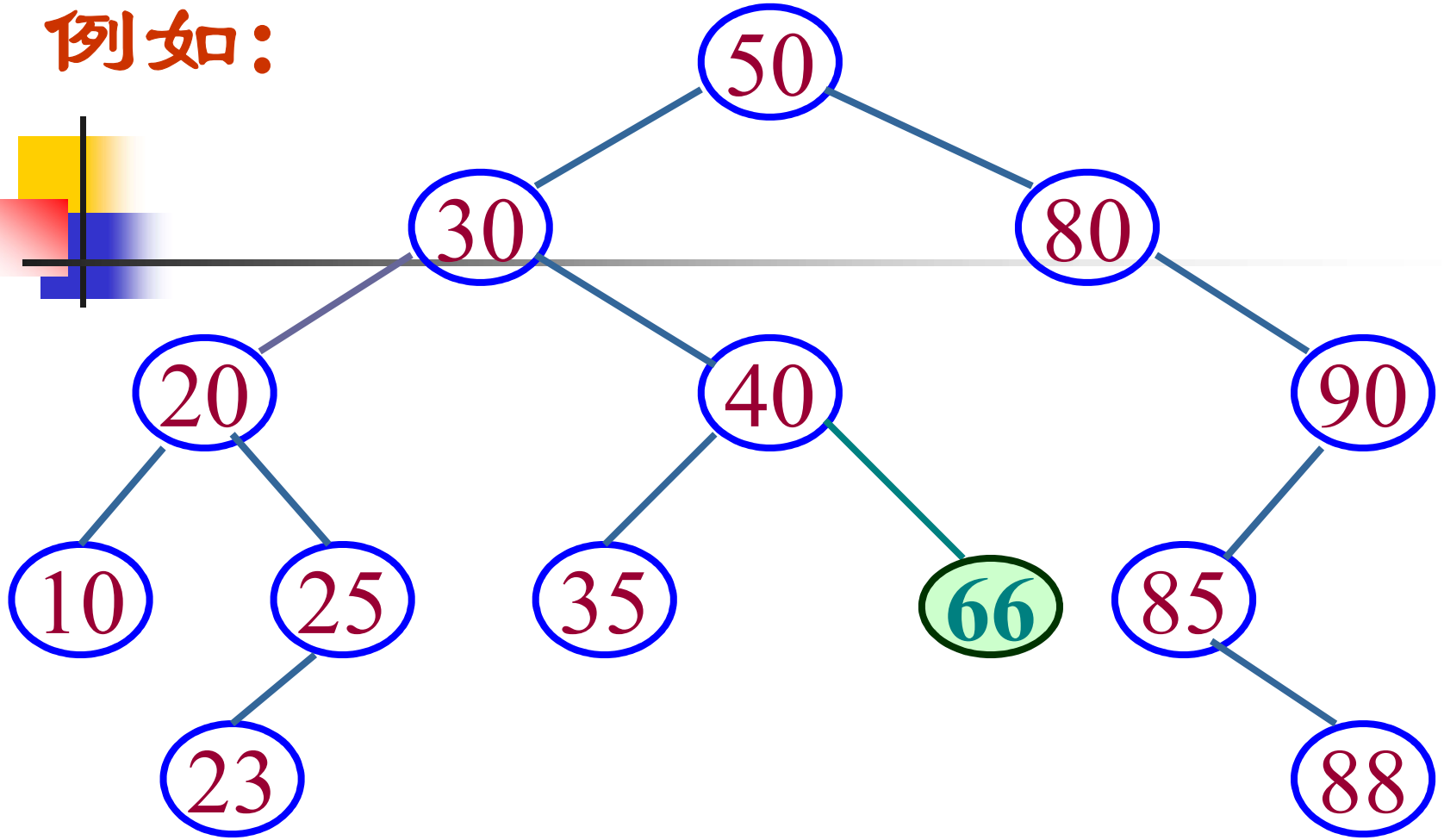
9.2.1 二叉排序树和平衡二叉树

- 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:

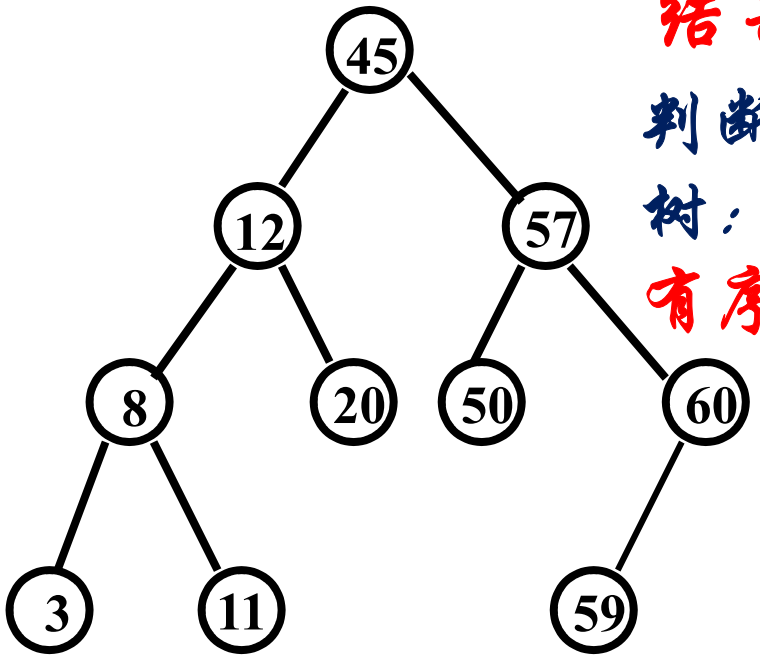
1. 若它的左子树不空，则左子树上所有结点的值均小于根结点的值;
2. 若它的右子树不空，则右子树上所有结点的值均大于根结点的值;
3. **它的左、右子树也都分别是二叉排序树**

例如：



不是二叉排序树

二叉排序树



二叉排序树的中序遍历结果递增有序

判断一棵二叉树是否为二叉排序树：看其中序遍历结果是否递增有序

设计非递归算法判断一棵二叉树是否为二叉排序树—调用二叉树的中序遍历非递归算法

- 中序遍历：3, 8, 11, 12, 20, 45, 50, 57, 59, 60

二叉排序树

- 对二叉排序树进行中序遍历，便可得到一个按关键字**有序**的序列.
- 因此，一个无序序列，可通过构建一棵二叉排序树而成为有序序列。
- 二叉排序树**主要作用：检索，排序**
- 二叉排序树主要操作：检索，插入（建立），删除

二叉排序树的检索操作

- 在二叉排序树中**查找**是否存在值为*x*的结点。
- 查找过程为：

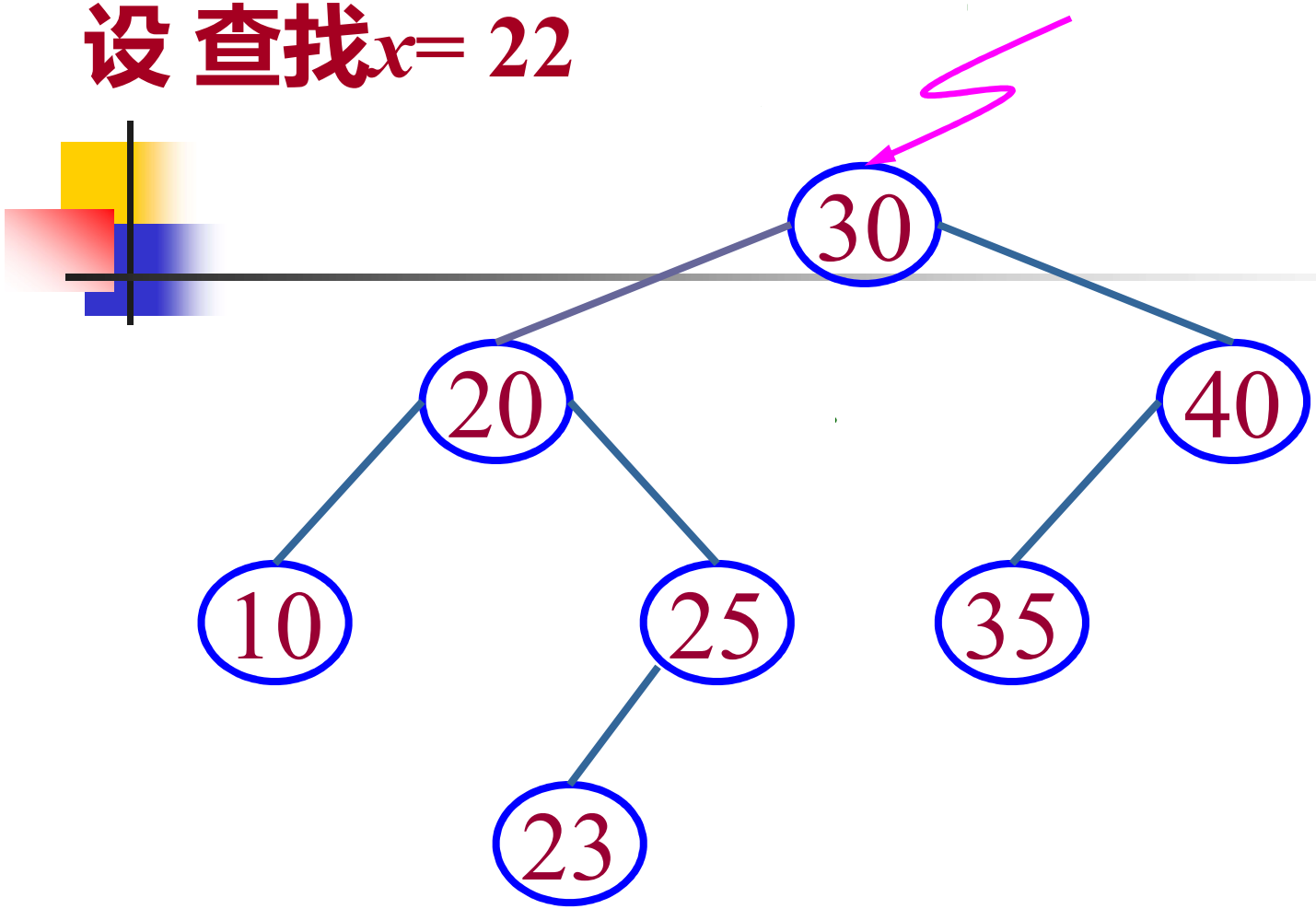
① 若二叉排序树为**空**，查找**失败**。

② 否则，将*x*与二叉排序树的**根**结点关键字值比较：若**相等**，查找**成功**，结束；否则，

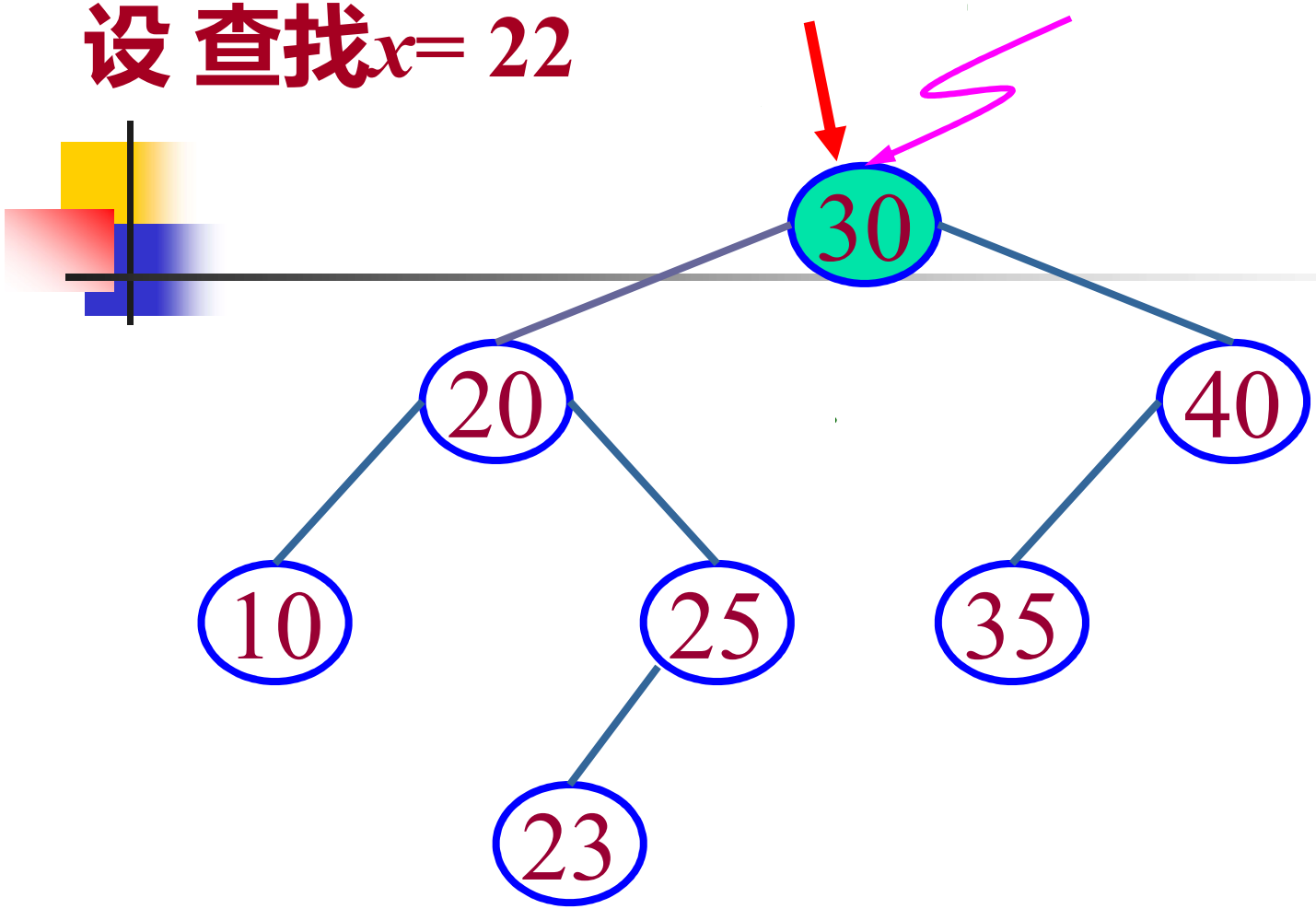
a．若*x*小于根结点关键字，在**左子树**上继续进行，转①

b．若*x*大于根结点关键字，在**右子树**上继续进行，转①

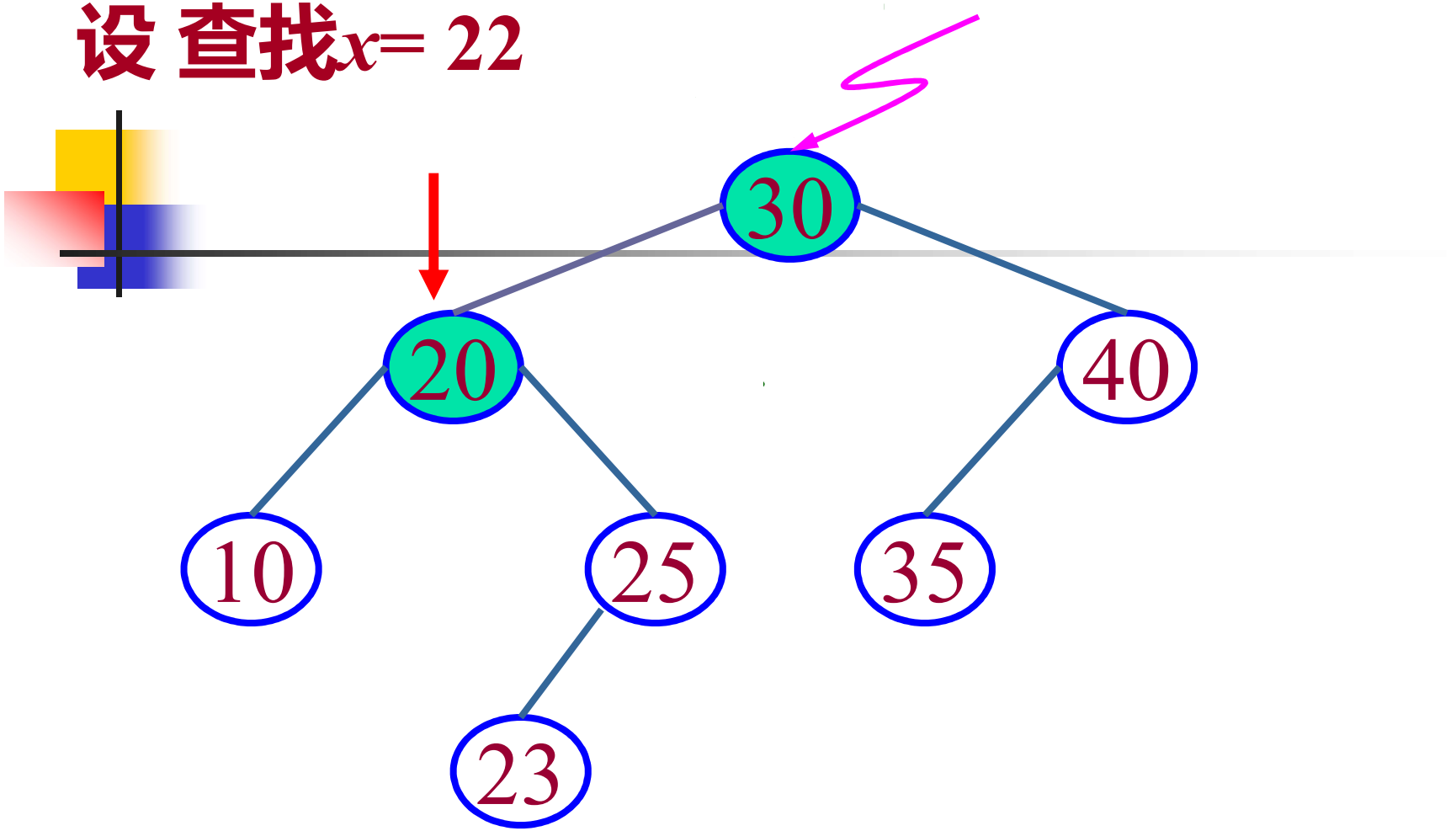
设 查找x= 22



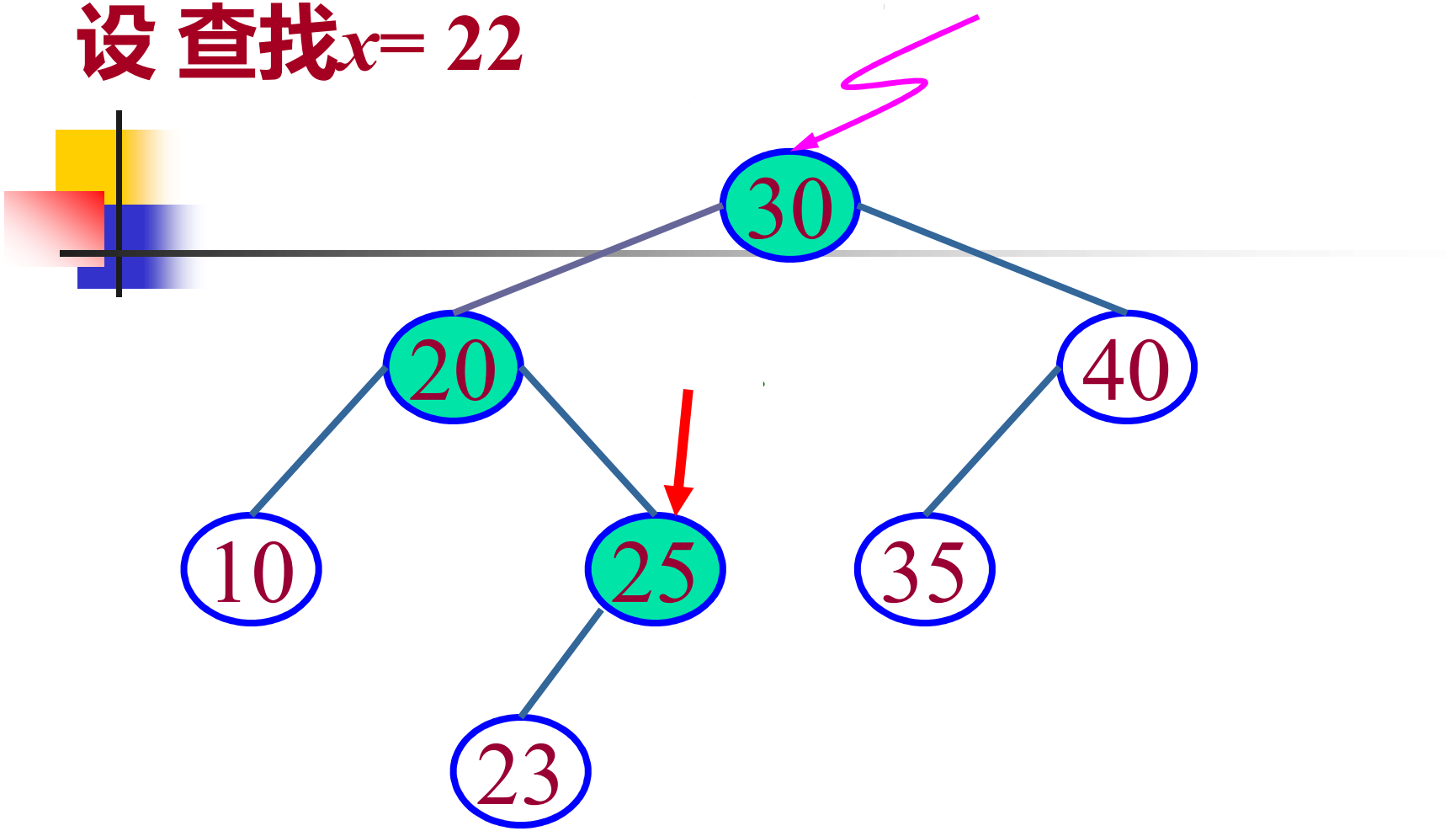
设 查找x= 22



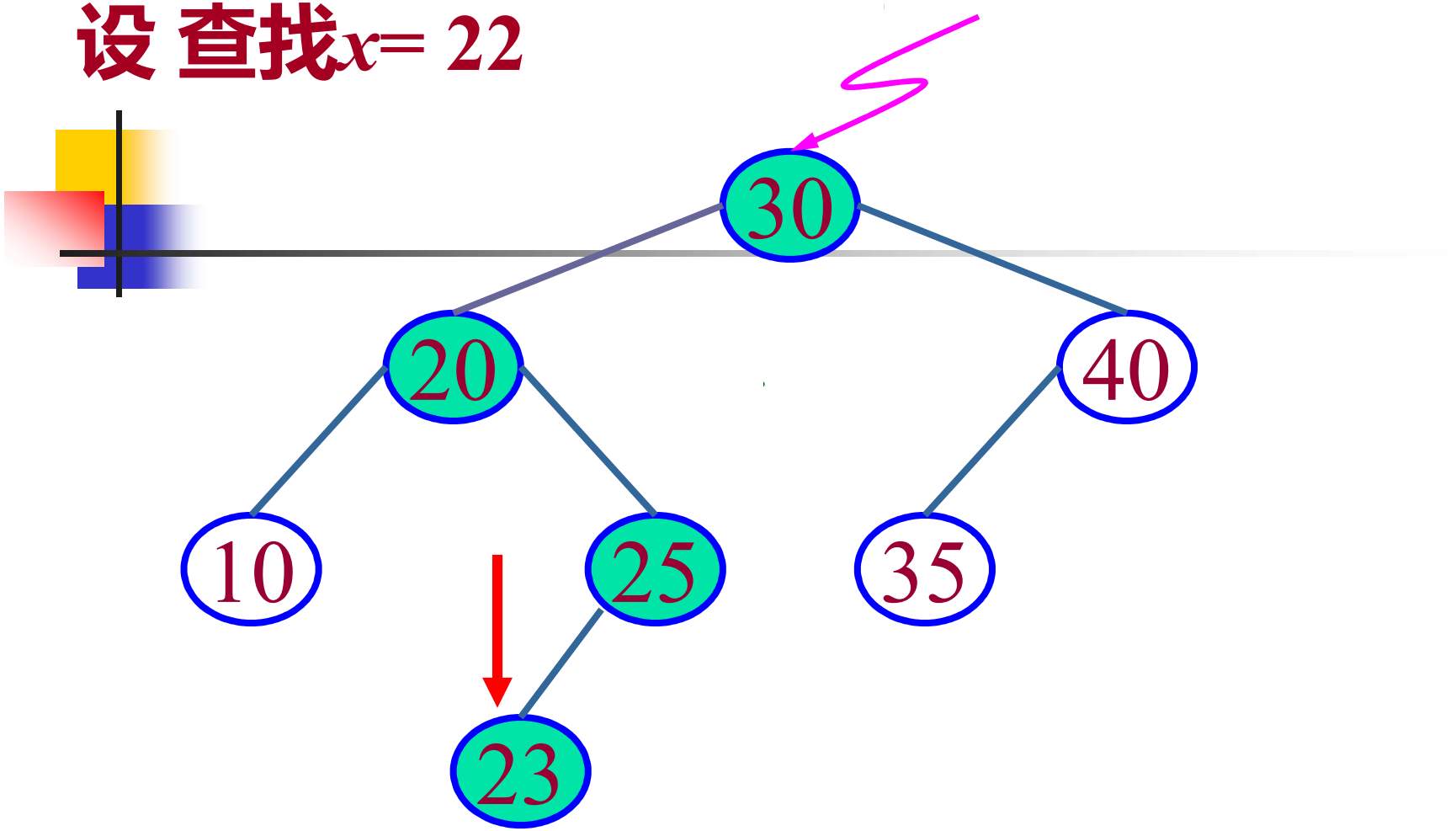
设 查找x= 22



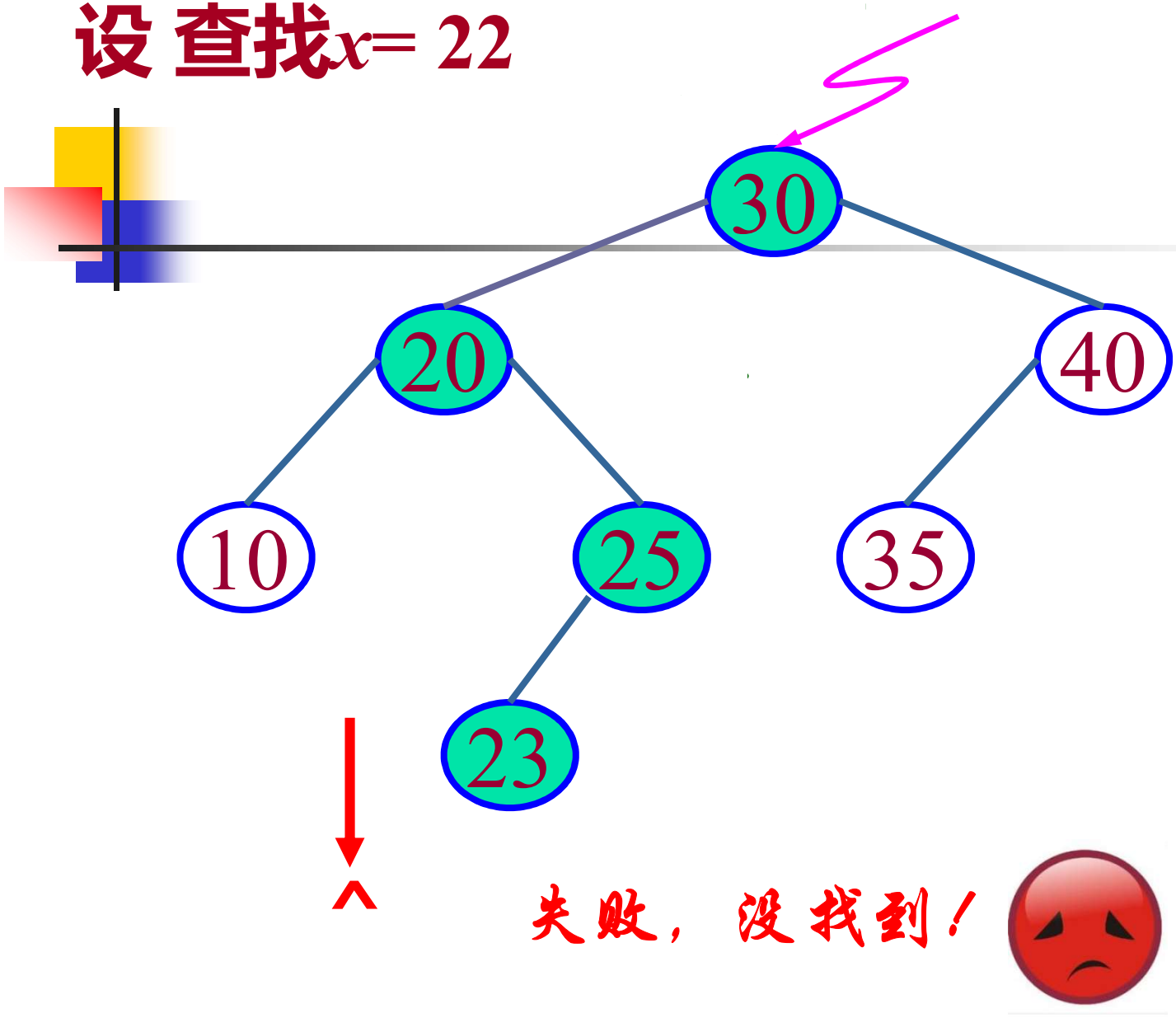
设 查找x= 22



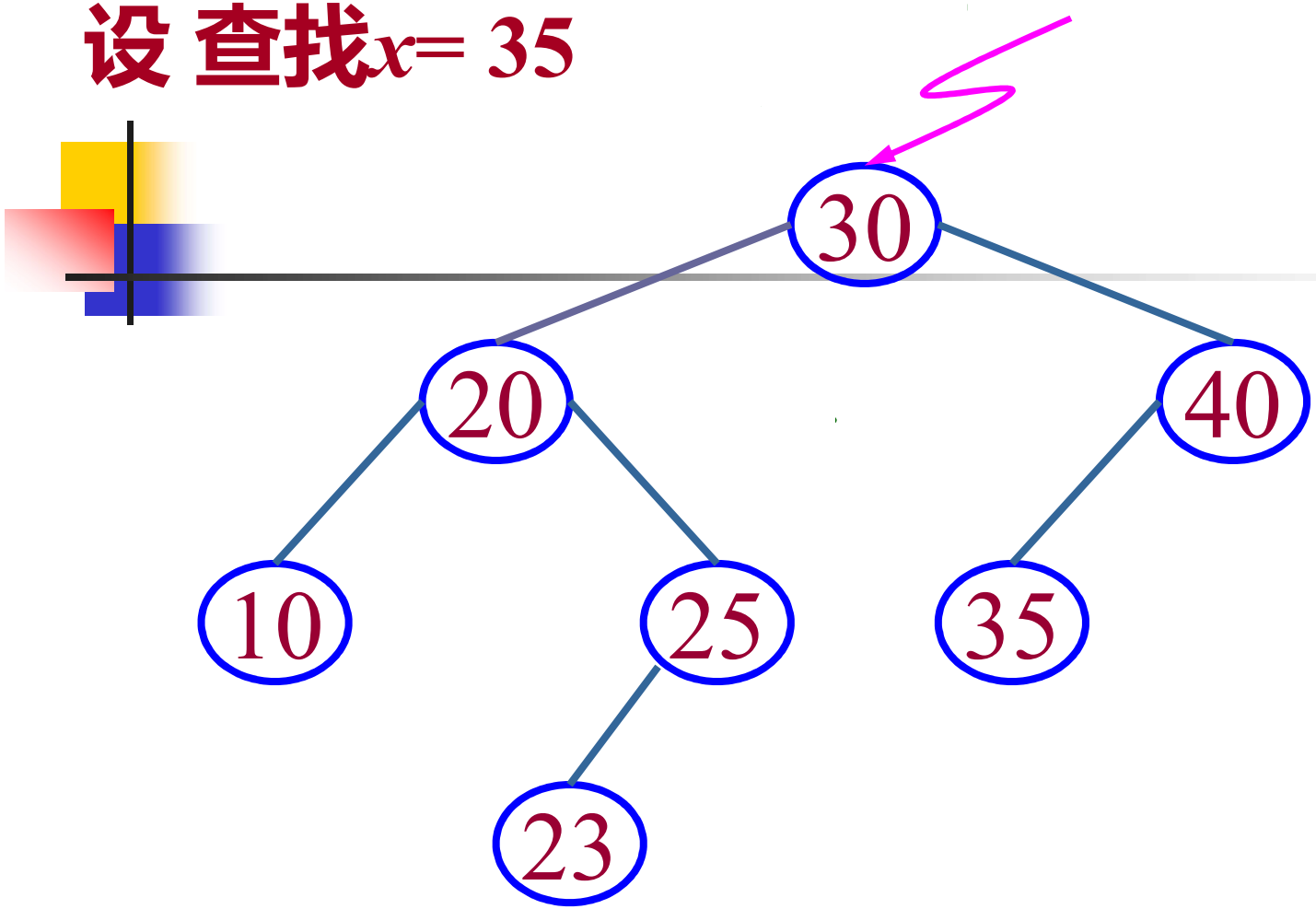
设 查找x= 22



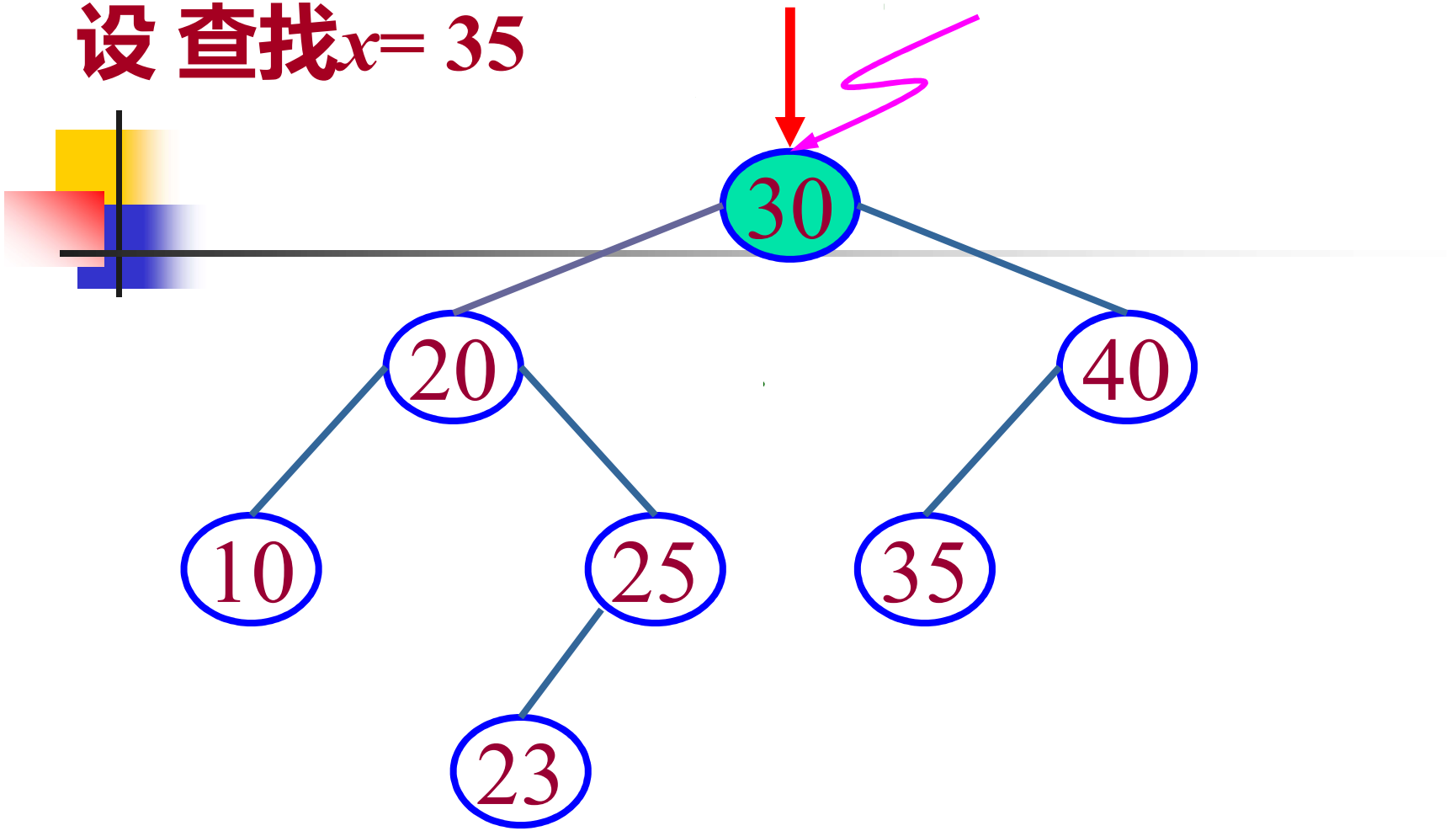
设 查找x= 22



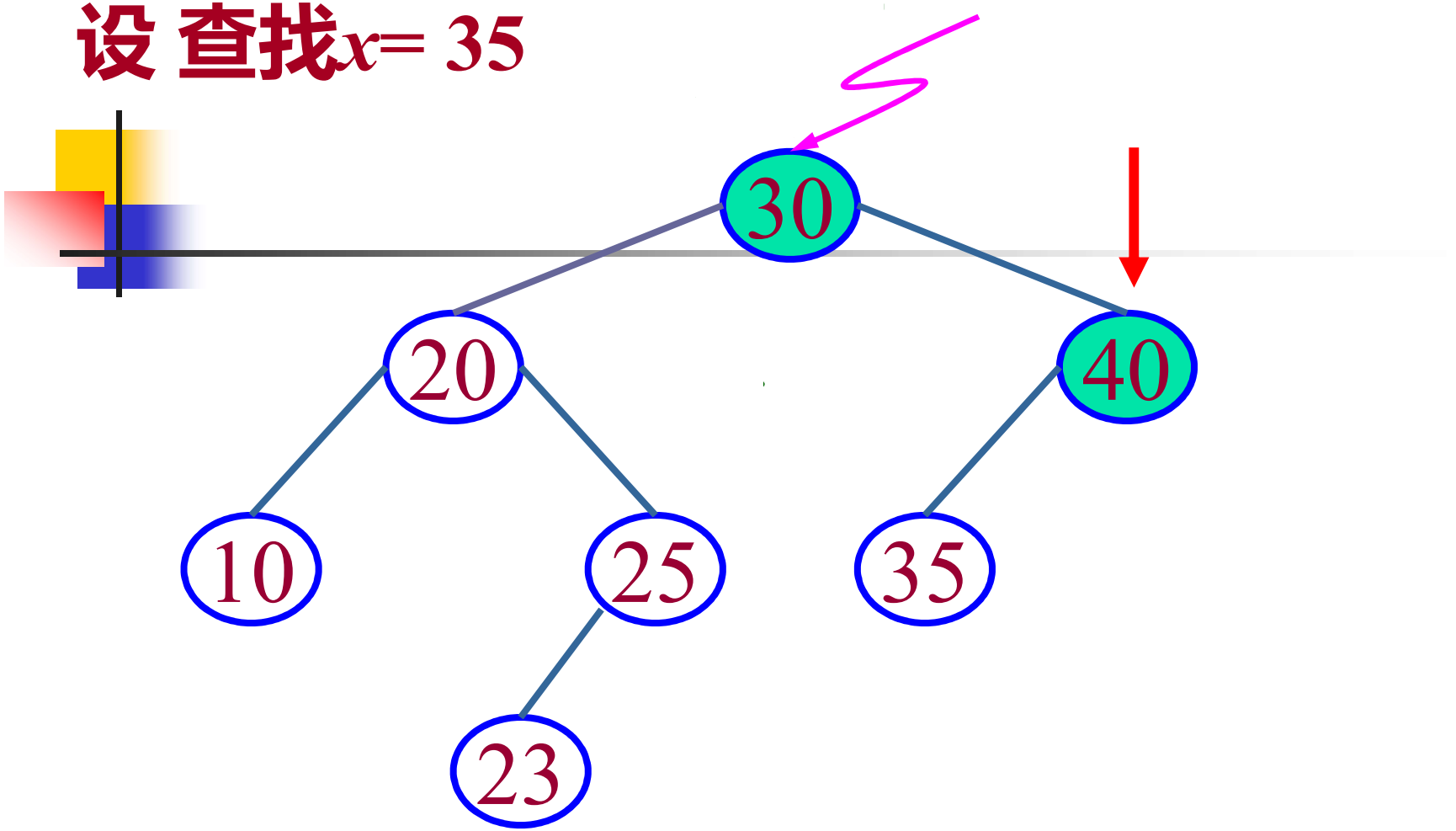
设 查找x= 35



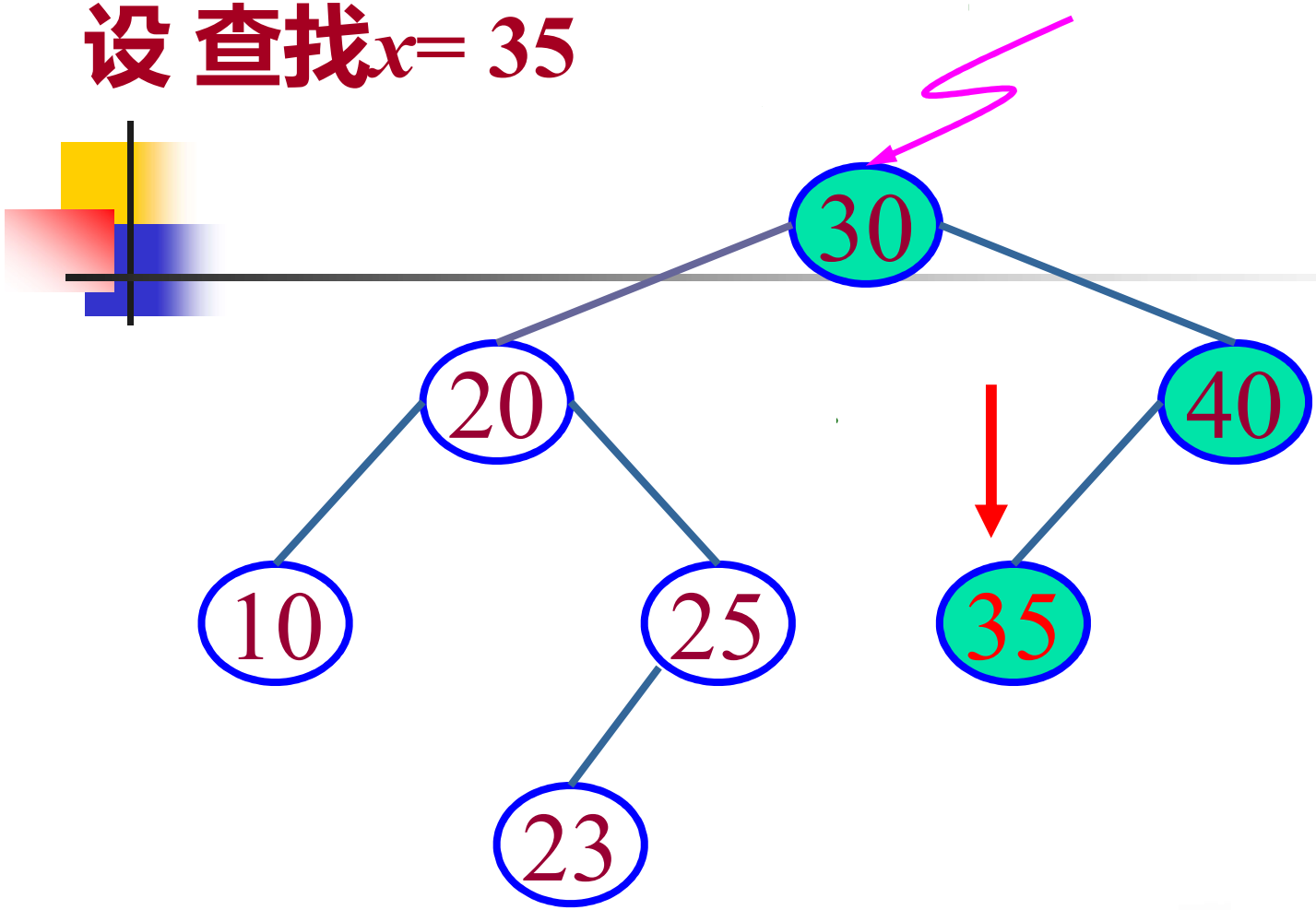
设 查找x= 35



设 查找x= 35

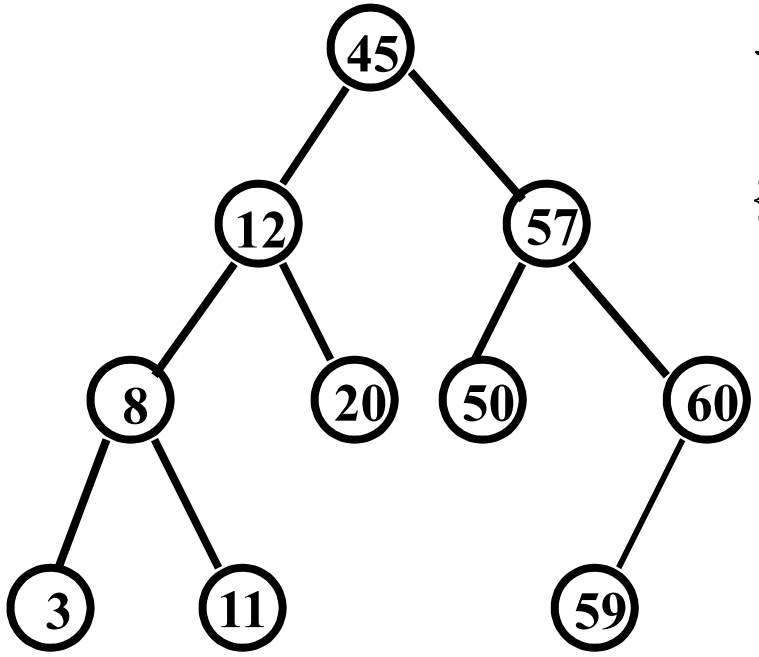


设 查找x= 35



哈哈，找到了！

二叉排序树的存储结构



二叉链表作为二叉排序树的存储结构

```
typedef   struct  NODE
{         int       key;//数据元素的关键字
                ... ;//数据元素的其他数据项
          struct NODE   *lc,*rc;
                        }BiNode,*BiTree;
```

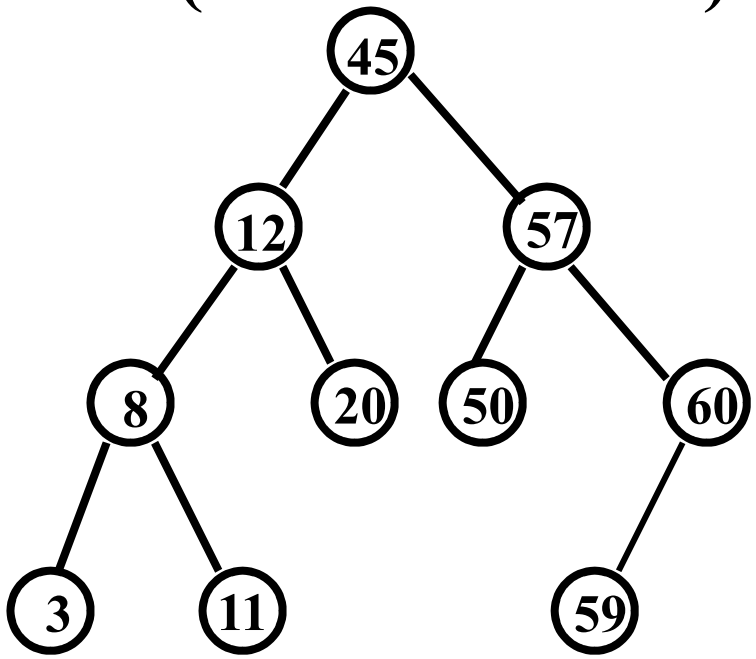
二叉排序树的检索算法

```
Bitree Search(BiNode *t, int x)
{  BiTree p;
   p=t;
   while(p!=NULL)
  {if (x==p->key)   return p;
        if (x<p->key) p=p->lc;
        else p=p->rc;
        }
   return p;
}//函数返回查找结果，没找到为空指针!
```

平均查找长度计算

- 该二叉树查找成功的平均比较次数:

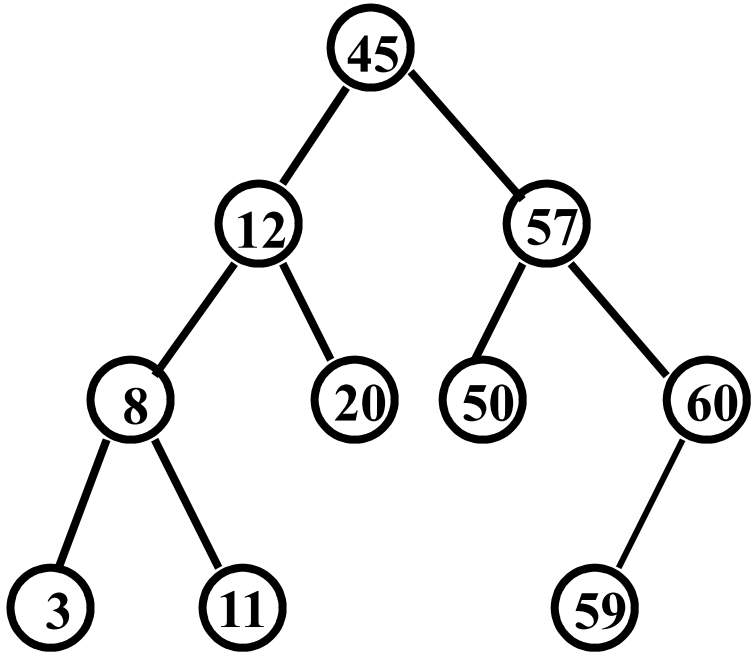
ASL=(1+2*2+3*4+4*3)/10=2.9



二叉排序树的插入算法

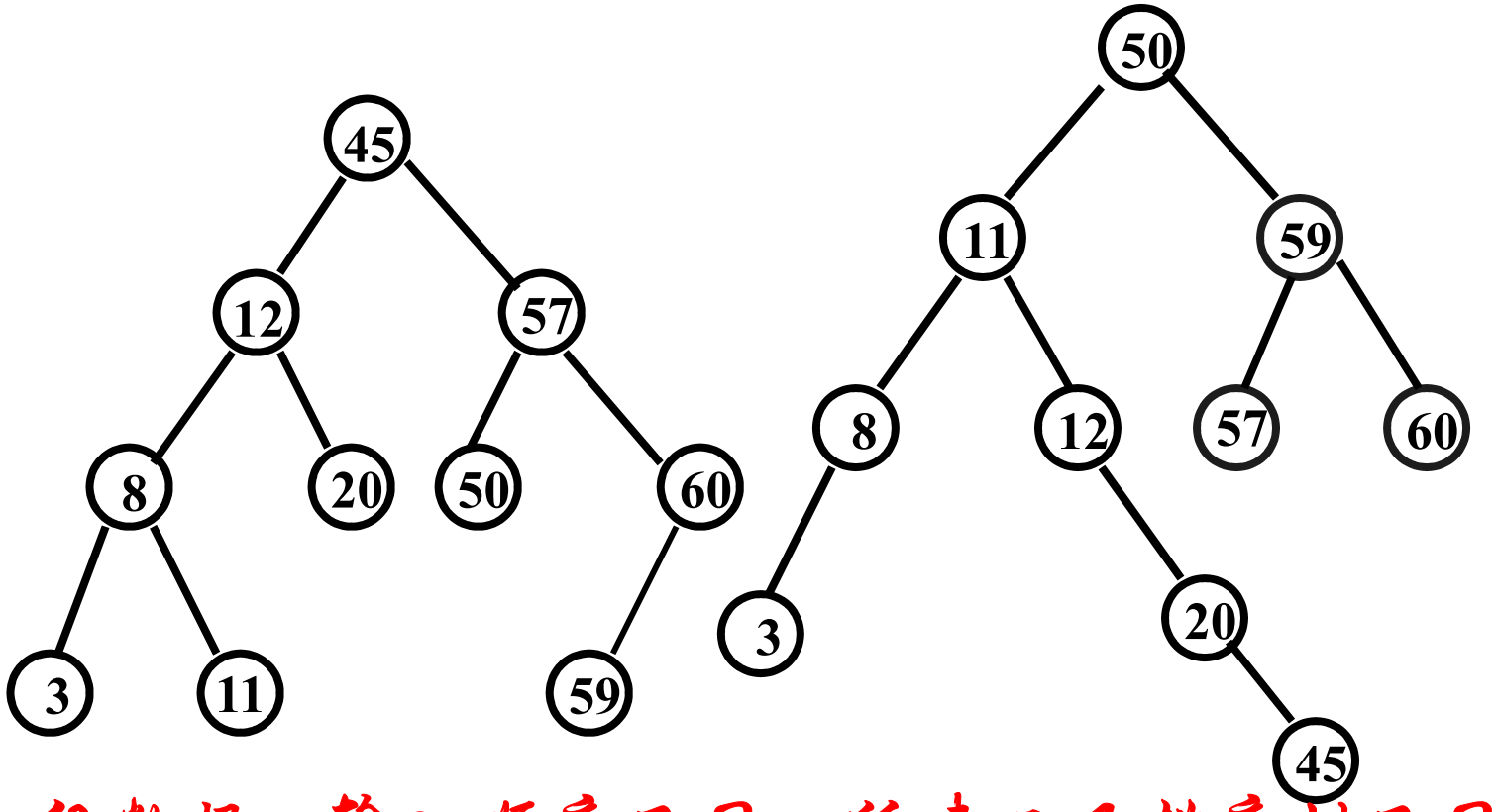
- 向二叉排序树中插入*x*：
  - 先要在二叉排序树中进行查找，若查找成功，按二叉排序树定义，待插入数据已存在，不用插入；查找不成功时，则插入之。
  - 新插入结点一定是作为叶子结点添加上去的。
- 建立一棵二叉排序树则是逐个插入结点的过程。

45，12，8，57，60，20，11，50，59，3

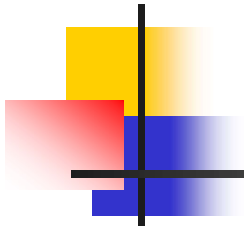


45，12，8，57，60，20，11，50，59，3

50，11，8，12，20，45，3，59，57，60



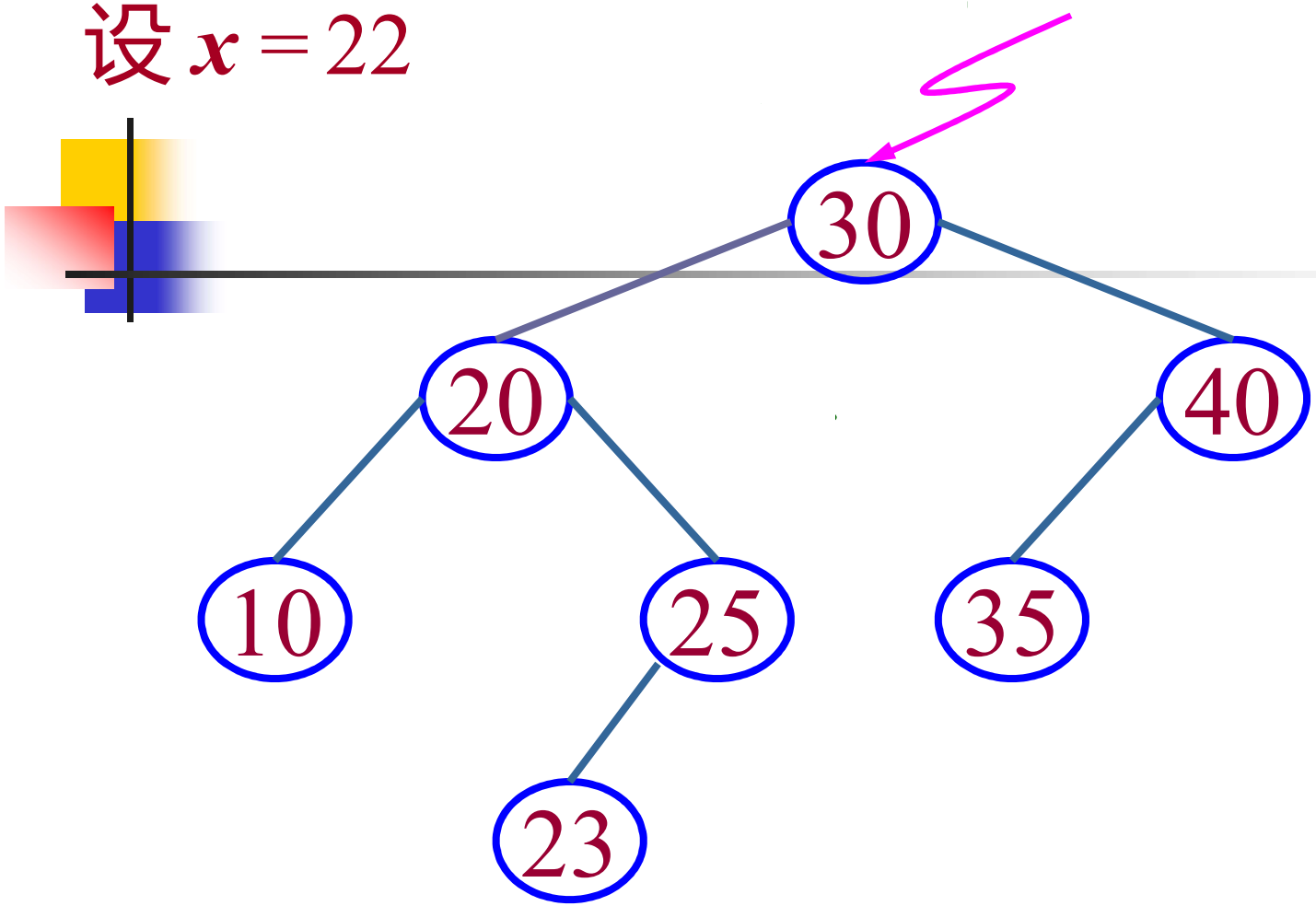
同一组数据，输入顺序不同，所建二叉排序树不同！



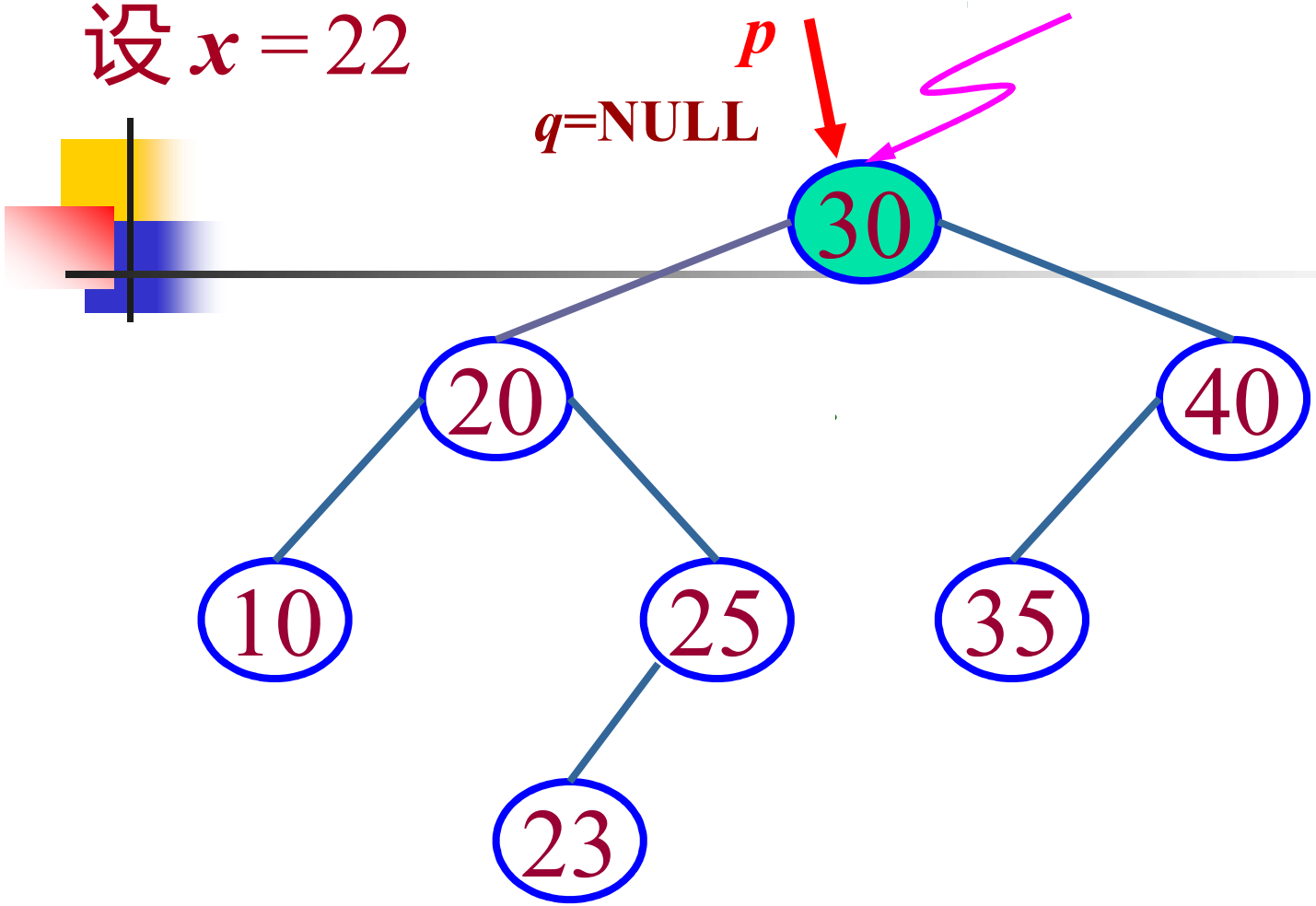
二叉排序树的插入算法

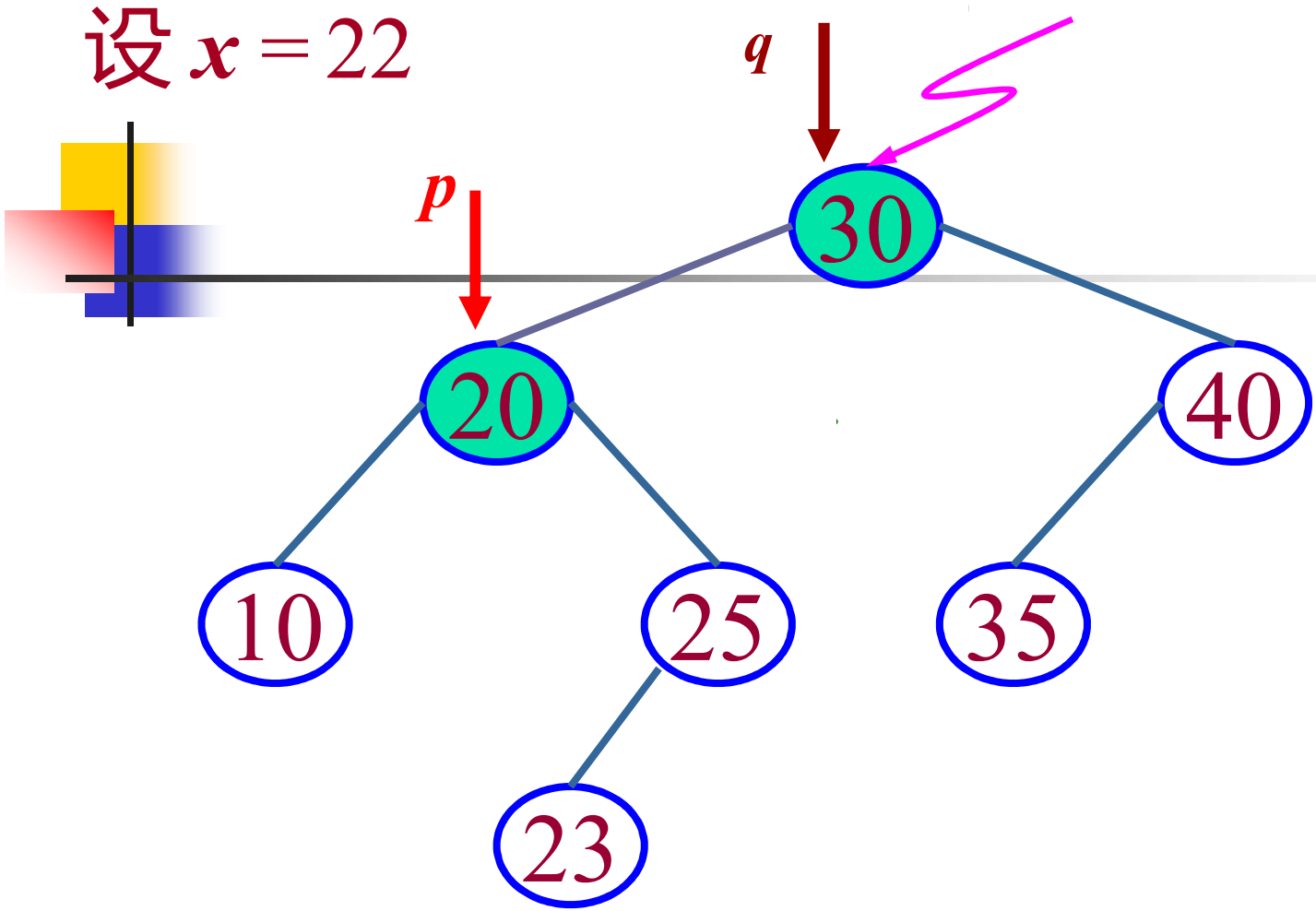
- 问题：按照不同的顺序输入1，2，3，可以建立多少棵不同的二叉排序树?

设 $x = 22$

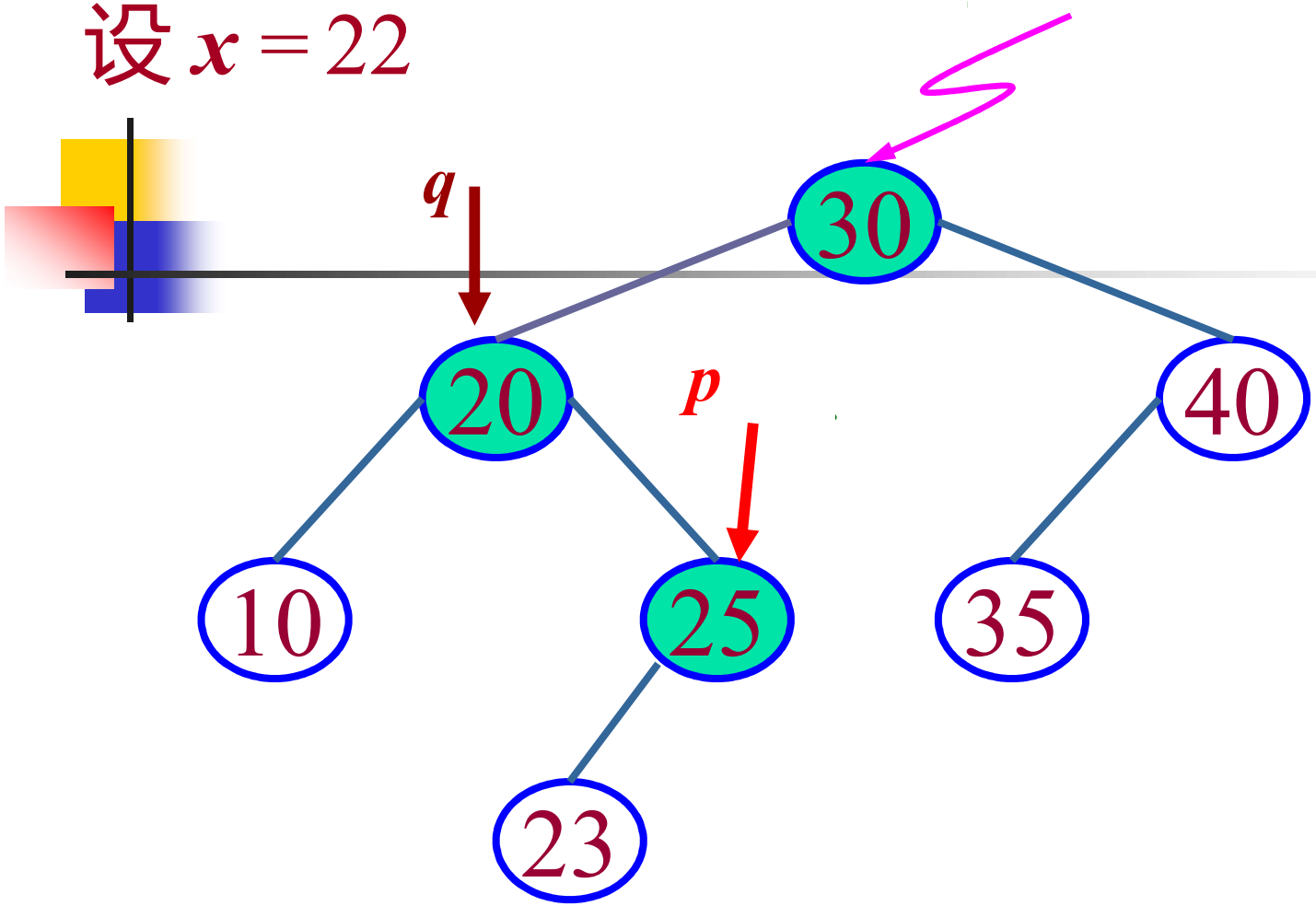


设 $x = 22$

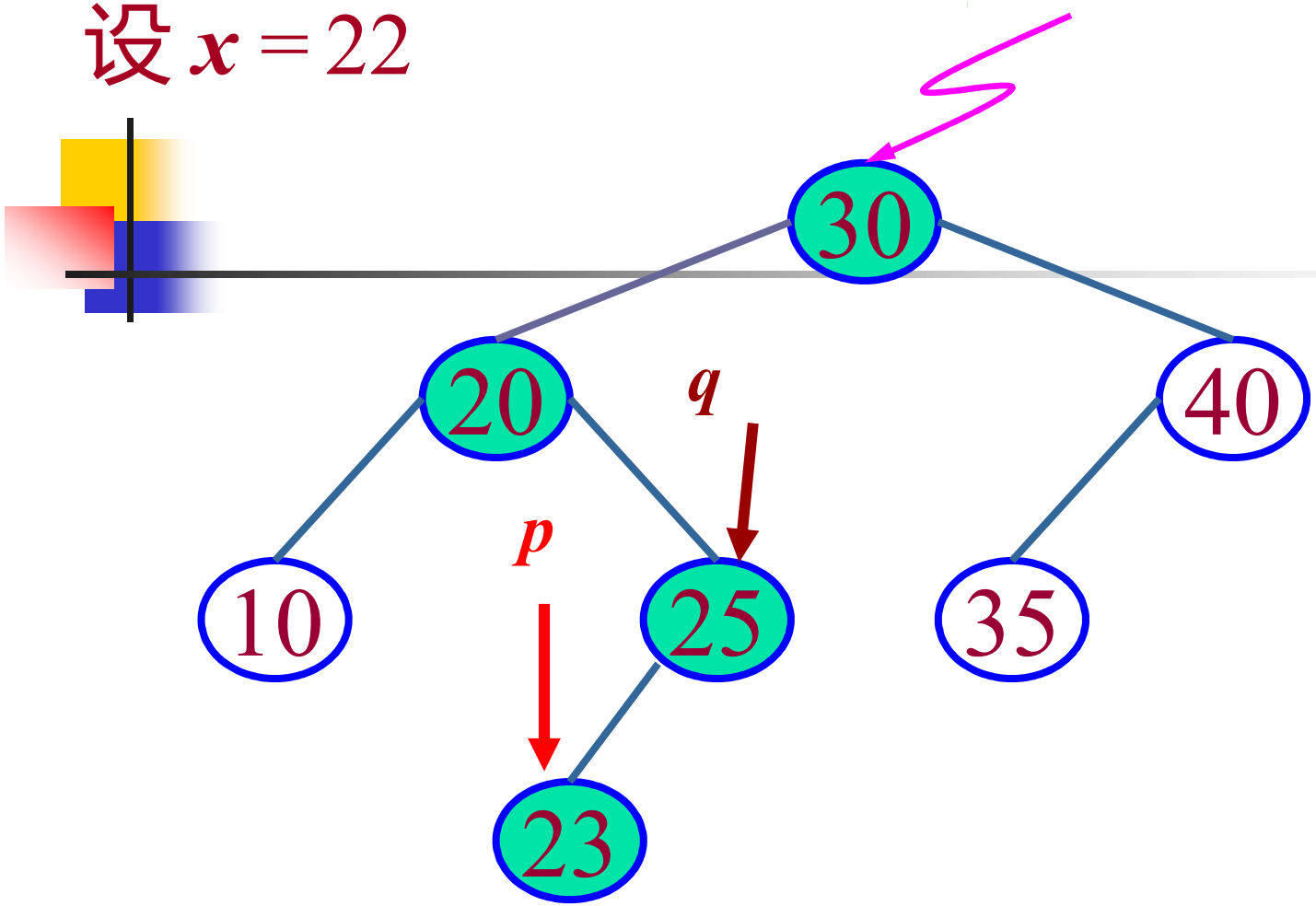



设 x = 22
q
p
30
20
40
10
25
35
23

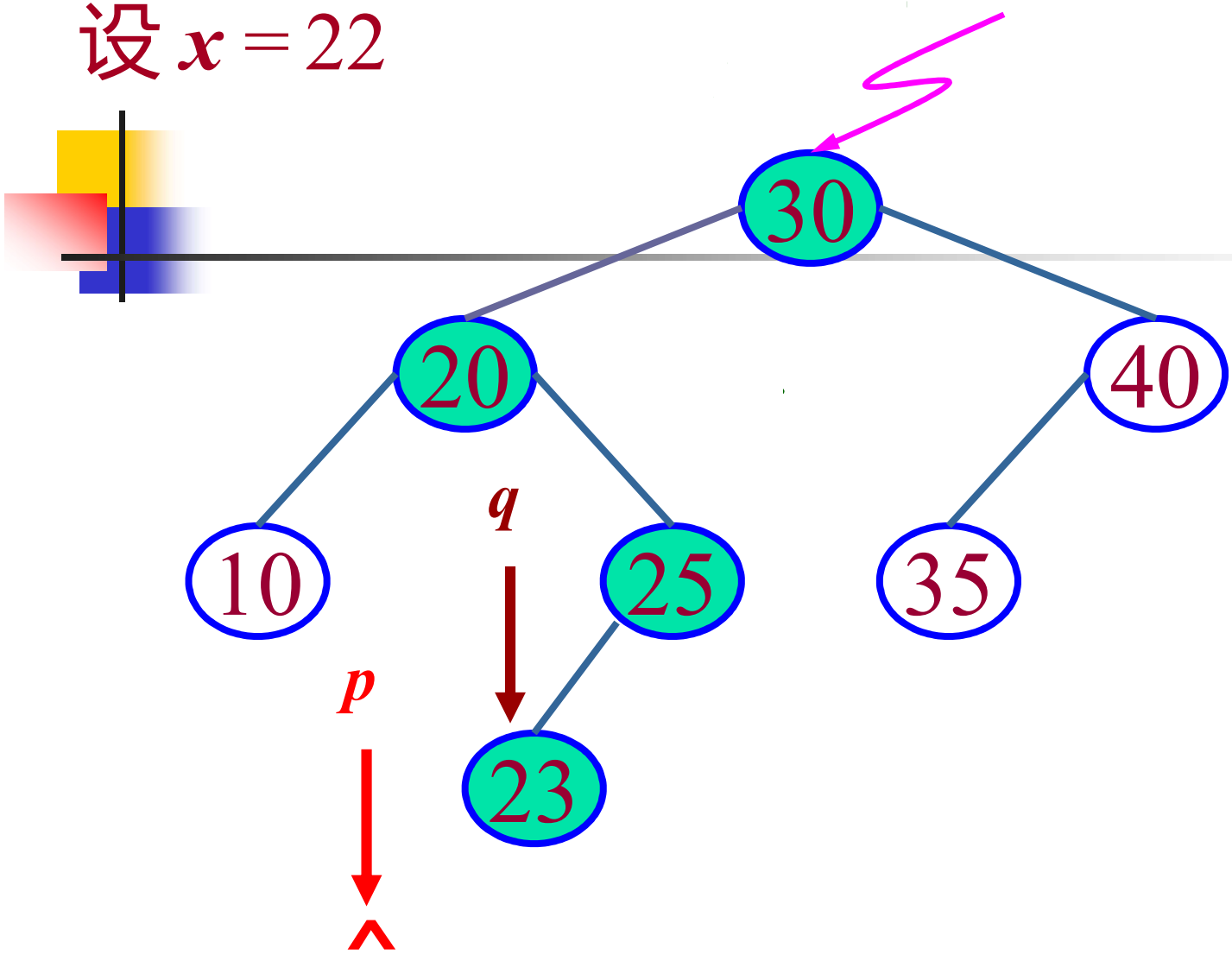
设 $x = 22$



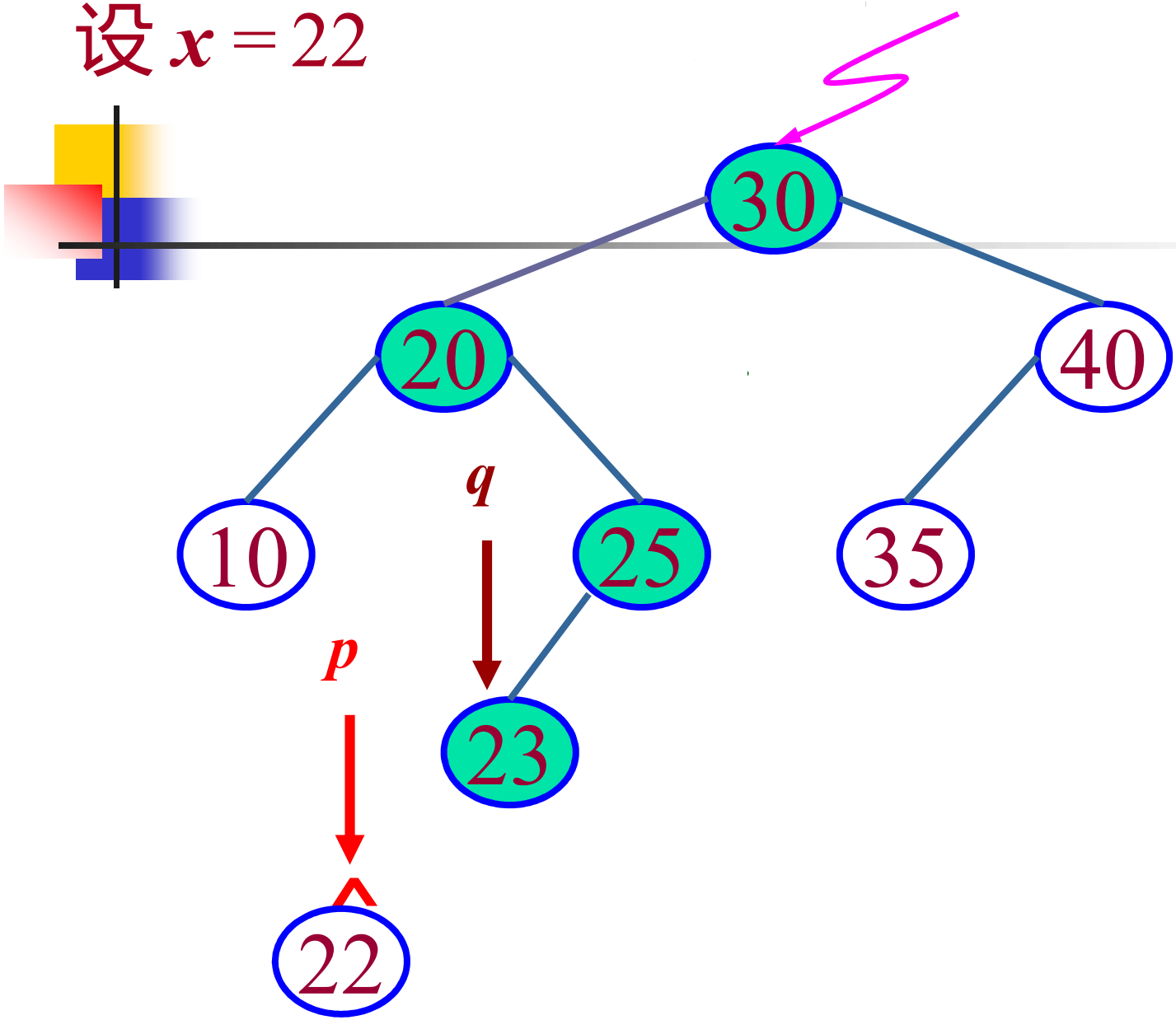
设 $x = 22$



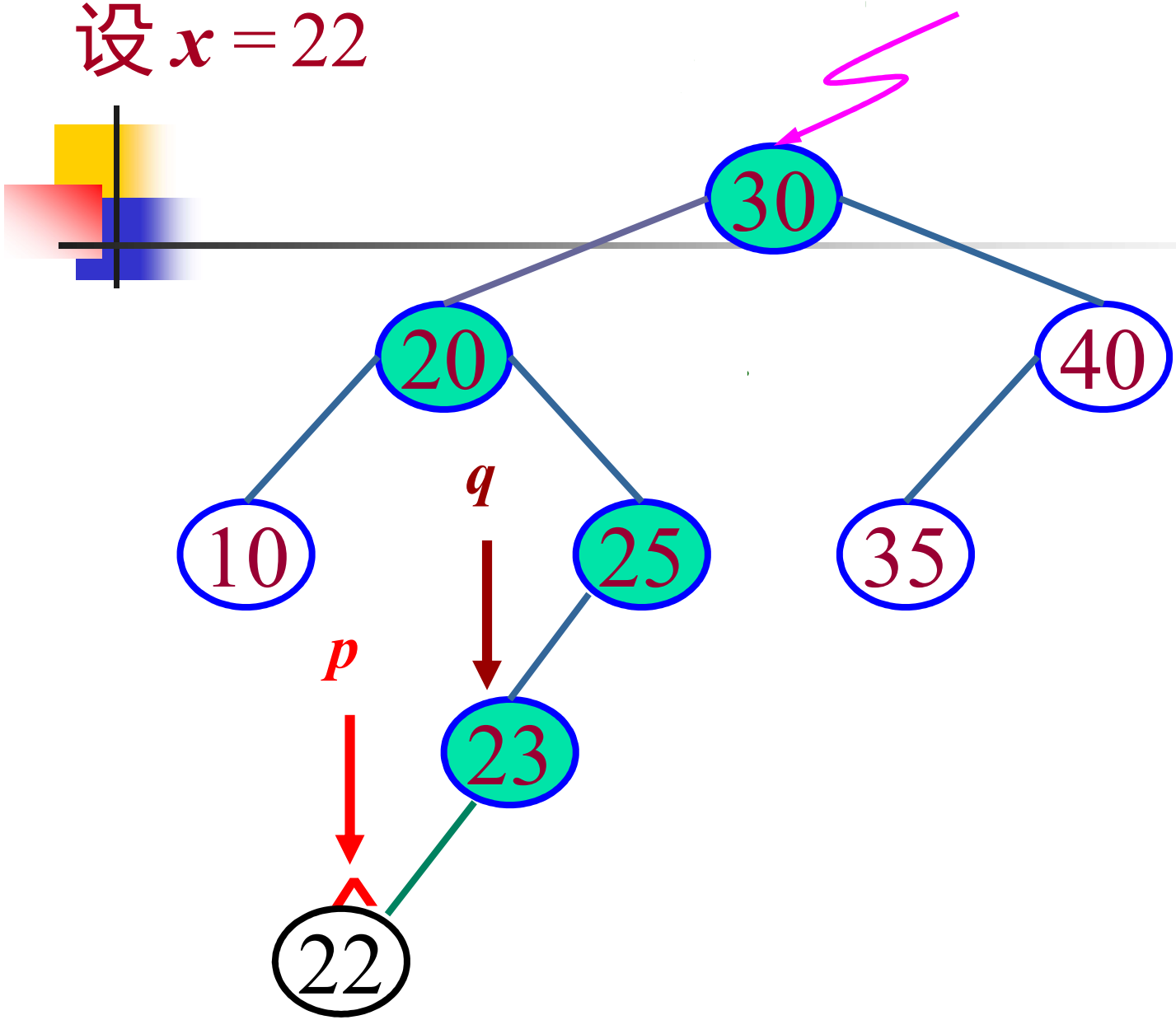
设 $x = 22$



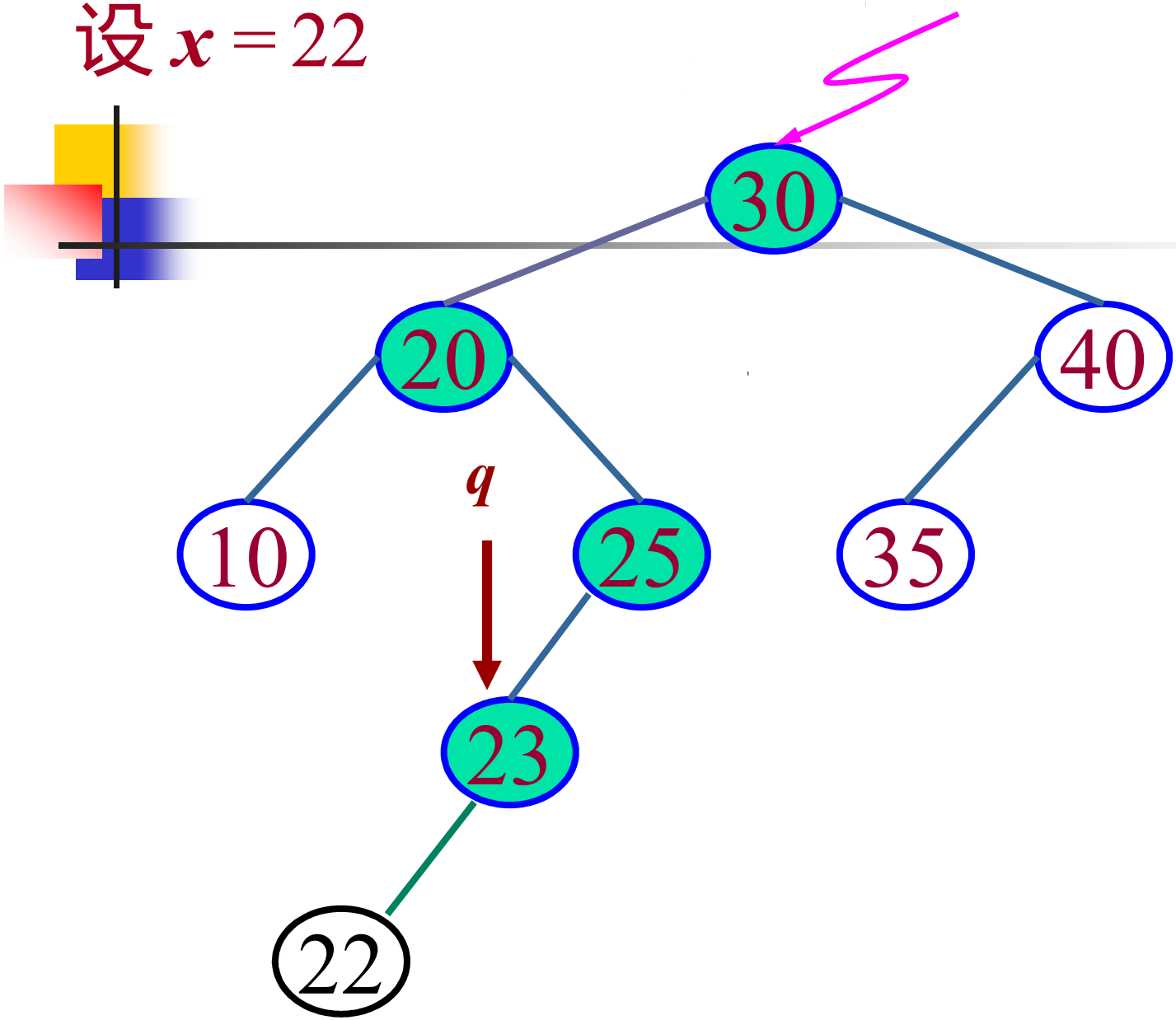
设 $x = 22$



设 $x = 22$



设 $x = 22$



```
int Insert(BiTree &t,int x)//二叉排序树的插入算法
{   BiTree q, p, s;
   q=NULL;   p=t; ;//p为正在查看的节点，初始从根节点开始；q为p的双亲节点，根节点无双亲节点
  while(p!=NULL)
  { if (x==p->key)          return 0;//在当前的二叉排序树中找到x，直接返回，不再做插入
       q=p;
      if (x<p->key) p=p->lc;
       else p=p->rc;
    }
    s=(BiTree)malloc(sizeof(Binode)) ;//没找到x，做插入：先申请节点空间
   s->key=x;s->lc=NULL; s->rc=NULL ;//存放x，设为叶节点
  if (q==NULL)t=s; ;//若原先的二叉树是一棵空树，新插入的x对应的节点s为插入后二叉树的根节点
  else if (x<q->key) q->lc=s; ;//插入s为q的孩子
       else q->rc=s; return 1;
}
```

二叉排序树的删除算法

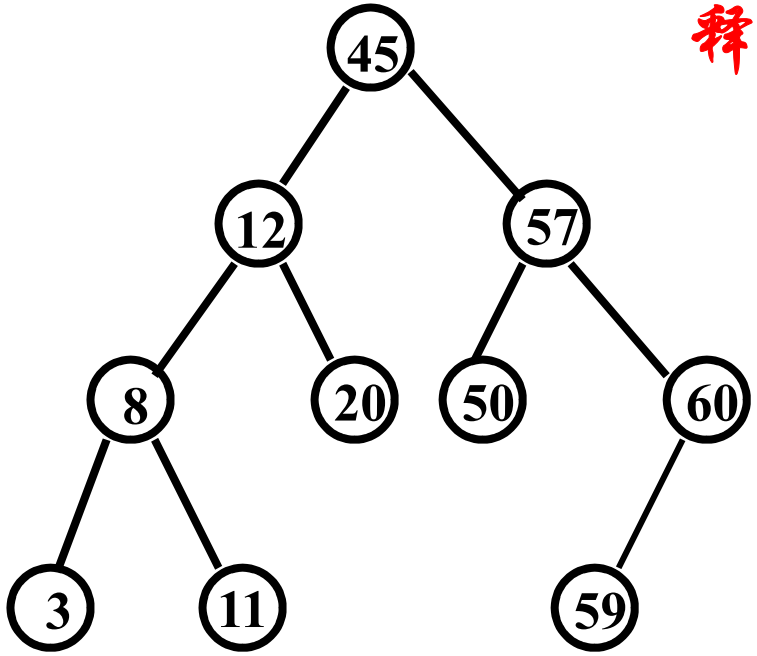
- 从二叉排序树中删除一个结点之后，使其仍能保持二叉排序树的特性。
- 被删结点为叶结点，由于删去叶结点后不影响整棵树的特性，所以只需将被删结点的双亲结点相应指针域改为空指针。
- 若被删结点只有右子树pr　或只有左子树pl，此时，只需将pr或pl替换被删结点即可。
- 若被删结点既有左子树pl又有右子树pr，可按中序遍历保持有序进行调整。

删除3

叶结点

修改3的父结点8的左孩子为空

释放3结点所占空间

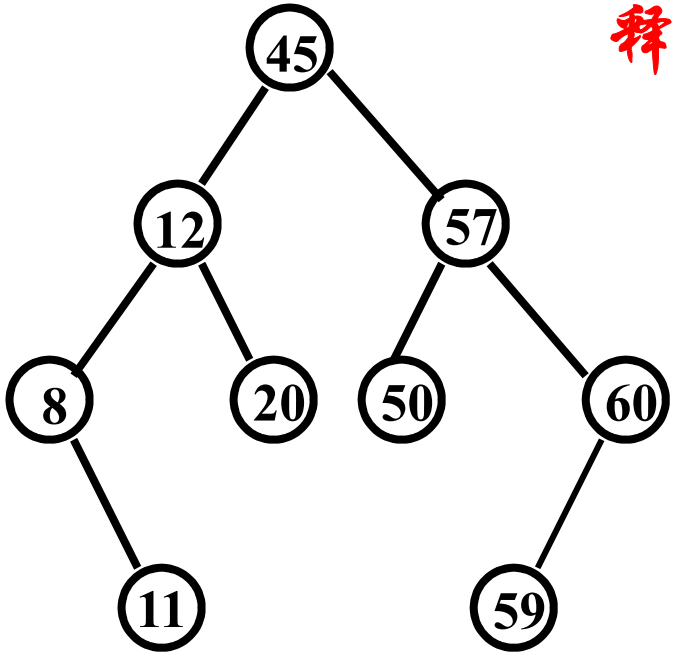


删除3

叶结点

修改3的父结点8的左孩子为空

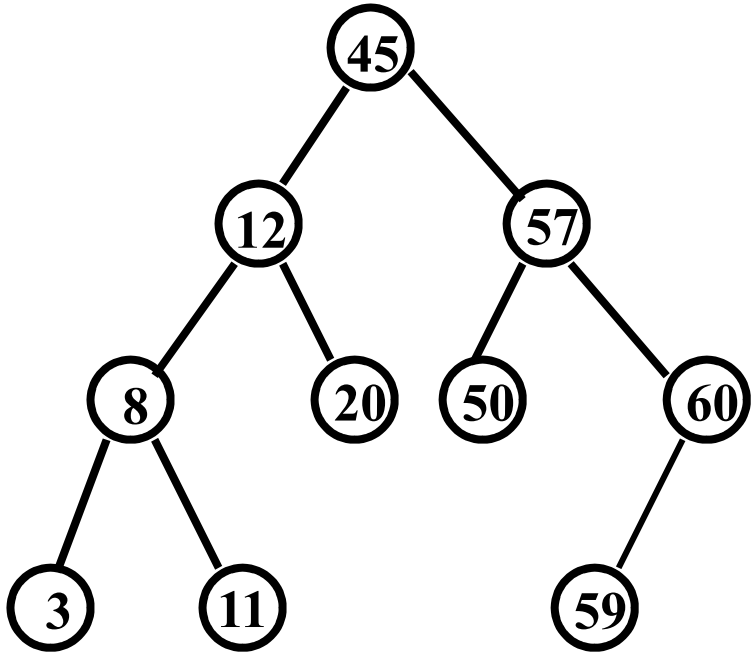
释放3结点所占空间



删除60

只有左孩子结点

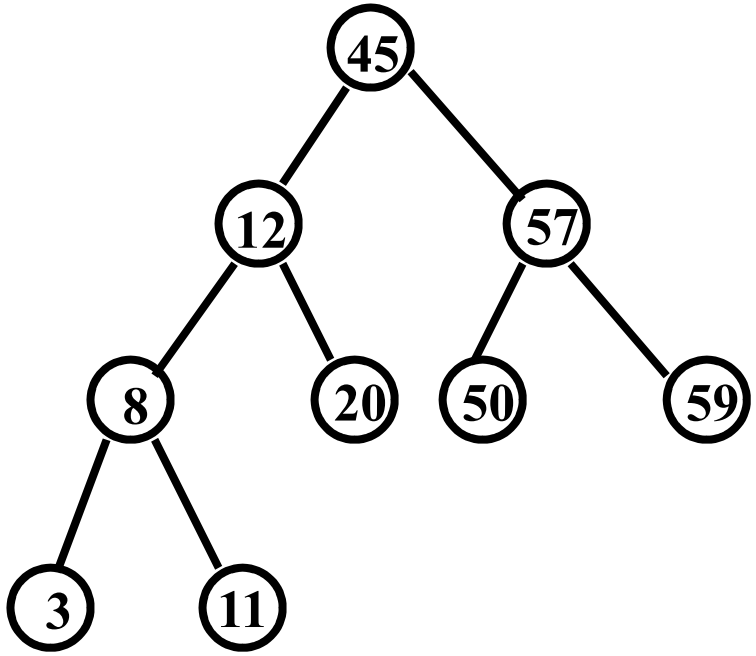
用其左孩子59替代它



删除60

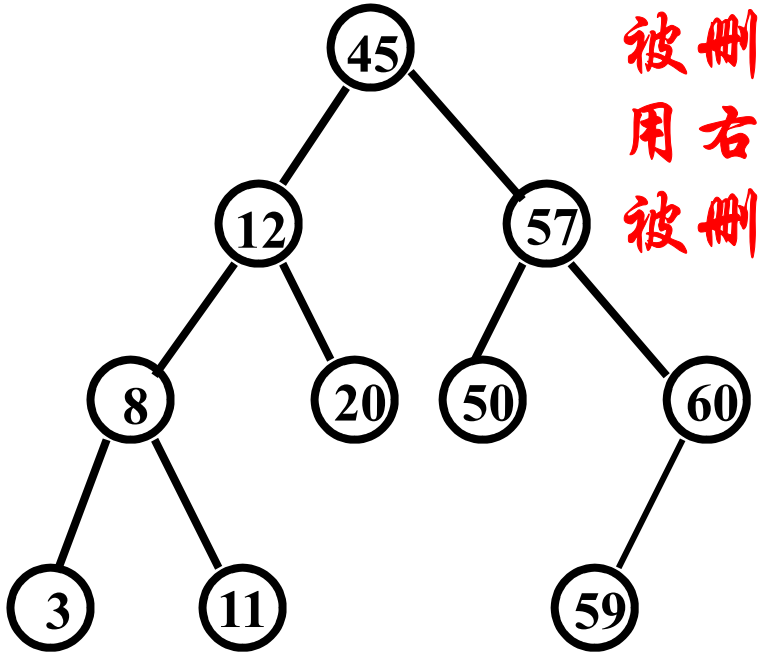
只有左孩子结点

用其左孩子59替代它



删除12

左右孩子均存在



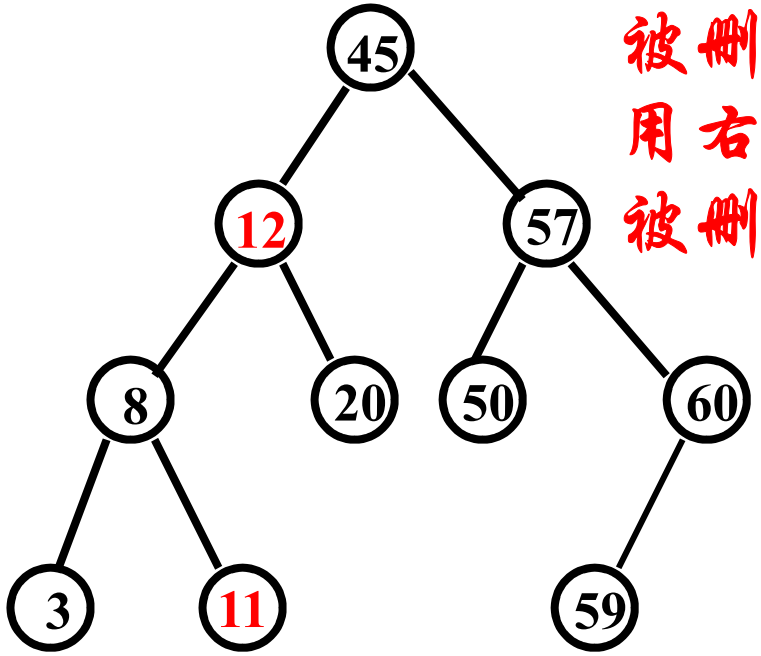
用左子树中最大的结点g的值替换被删结点，再删“g”或

用右子树中最小的结点w的值替换被删结点，再删“w”

结点g(和w)的特点是最多只有一个孩子

删除12

左右孩子均存在

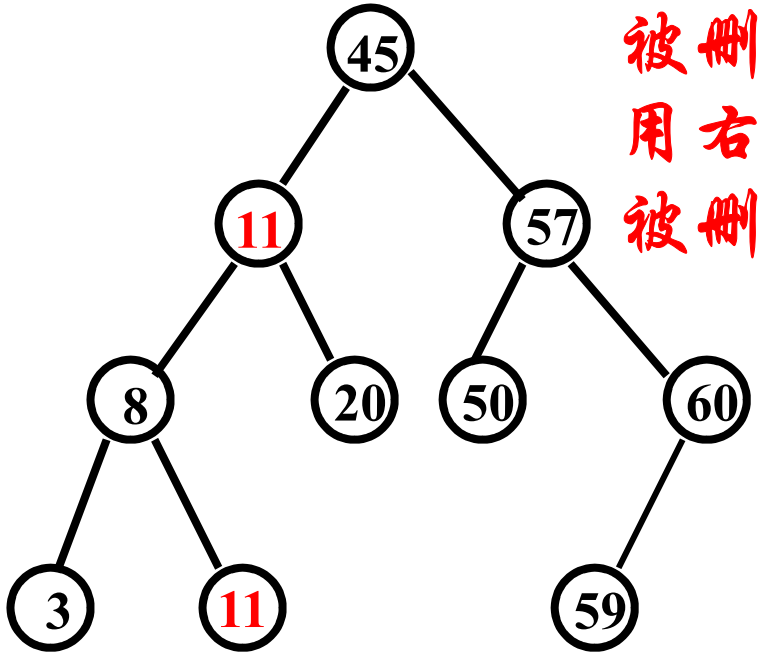


用左子树中最大的结点g的值替换被删结点，再删“g”或

用右子树中最小的结点w的值替换被删结点，再删“w”

结点g(和w)的特点是最多只有一个孩子

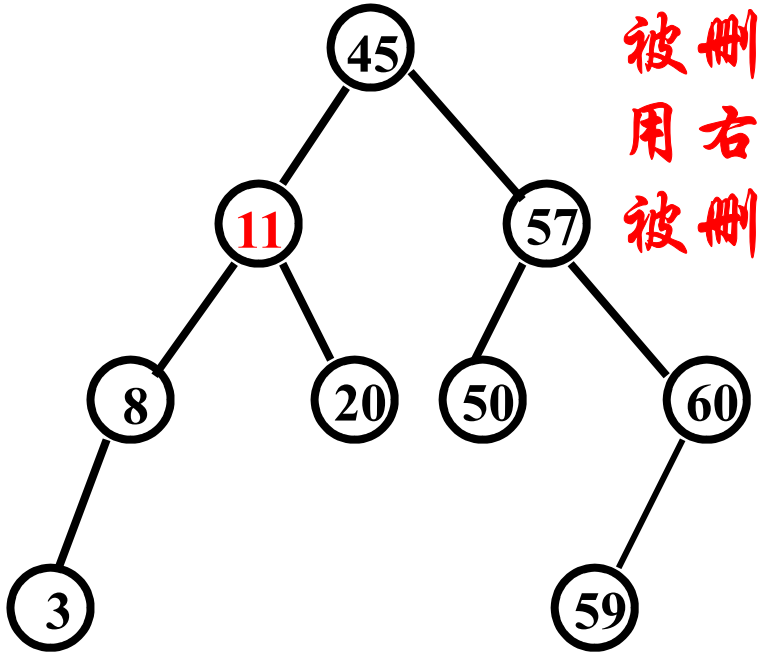
删除12　　左右孩子均存在



用左子树中最大的结点g的值替换被删结点，再删“g”或
用右子树中最小的结点w的值替换被删结点，再删“w”

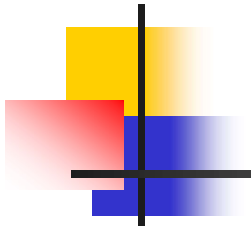
结点g(和w)的特点是最多只有一个孩子

删除12　　左右孩子均存在



用左子树中最大的结点g的值替换被删结点，再删“g”或
用右子树中最小的结点w的值替换被删结点，再删“w”

结点g(和w)的特点是最多只有一个孩子



最佳二叉排序树

- 二叉排序树查找过程与二分查找判定树相似
- n个数据元素按照不同的输入顺序构造的二叉排序树不同，其中平均查找性能最高的为**最佳二叉排序树**
- 按照二分查找判定树的建立过程建立的二叉排序树为最佳二叉排序树。
- 二叉排序树的查找速度一般比顺序查找快，但如果是单枝树则一样快。