9.3 树表的查找

以二叉树或树作为表的组织形式，称为**树表**，它是一类动态查找表，不仅适合于数据查找，也适合于表插入和删除操作。

常见的树表：

- 二叉排序树
- 平衡二叉树
- B-树
- B+树

9.3.1 二叉排序树

二叉排序树（简称BST）又称二叉查找（搜索）树，其定义为：二叉排序树或者是空树，或者是满足如下性质（BST性质）的二叉树：

❶ 若它的左子树非空，则左子树上所有节点值（指关键字值）均小于根节点值；

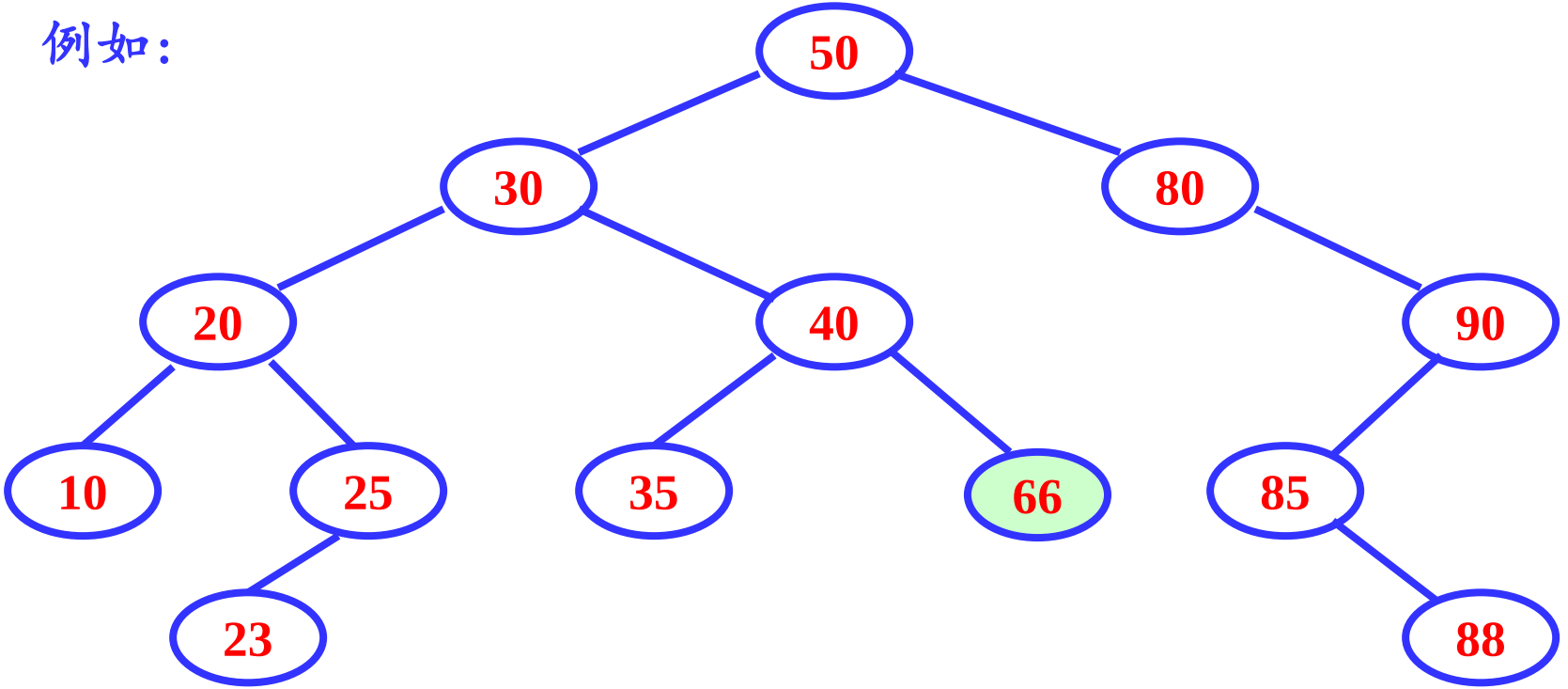
❷ 若它的右子树非空，则右子树上所有节点值均大于根节点值；

❸ 左、右子树本身又各是一棵二叉排序树。

注意：二叉排序树中没有相同关键字的节点。


二叉树结构
满足BST性质：节点值约束
二叉排序树

例如：



不是二叉排序树。

试一试

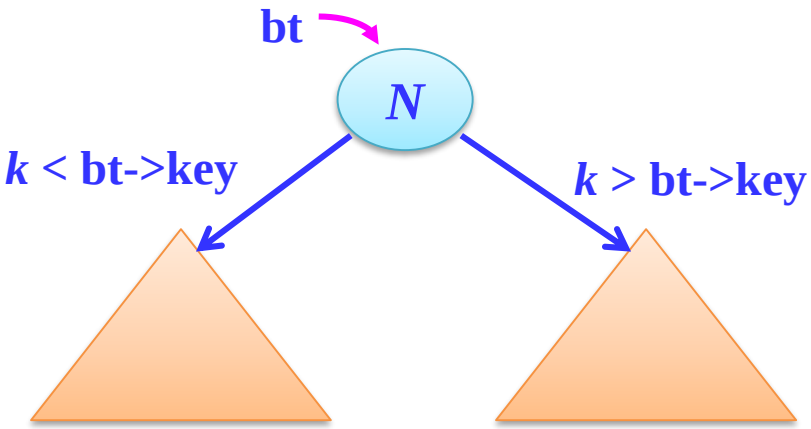
二叉排序树的中序遍历序列有什么特点？

二叉排序树的节点类型如下：

```
typedef struct node
{      KeyType key;                    //关键字项
       InfoType data;                  //其他数据域
       struct node *lchild,*rchild;    //左右孩子指针
} BSTNode;
```

1、二叉排序树上的查找

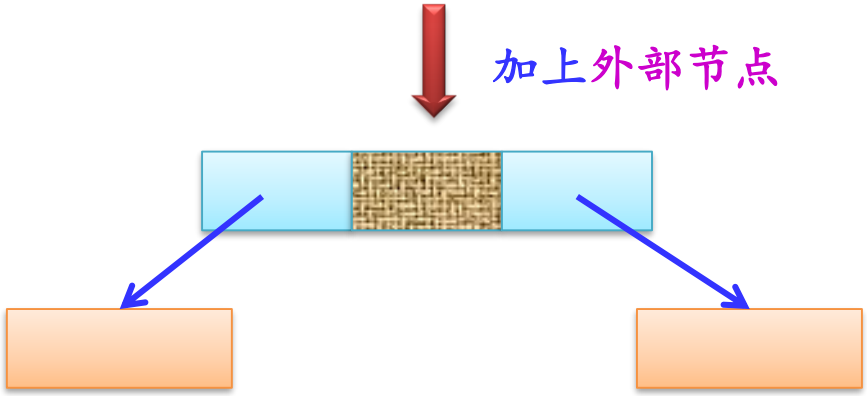
二叉排序树可看做是一个有序表，所以在二叉排序树上进行查找，和二分查找类似，也是一个逐步缩小查找范围的过程。



每一层只和一个节点进行关键字比较！

查找失败的情况

- 若*k*<p->data，并且p->lchild=NULL，查找失败。
- 若*k*>p->data，并且p->rchild=NULL，查找失败。



一个外部节点对应某内部节点的一个NULL指针

递归查找算法SearchBST()如下（在二叉排序树bt上查找关键字为k的记录，成功时返回该节点指针，否则返回NULL）：

```
BSTNode *SearchBST(BSTNode *bt,KeyType k)
{
    if (bt==NULL || bt->key==k)            //递归出口
        return bt;
    if (k<bt->key)
        return SearchBST(bt->lchild,k);   //在左子树中递归查找
    else
        return SearchBST(bt->rchild,k);   //在右子树中递归查找
}
```

思考题

二叉排序树查找可以设计成非递归算法，如何实现？

2、二叉排序树的插入和生成

在二叉排序树中插入一个关键字为*k*的新节点，要保证插入后仍满足BST性质。

插入过程：

（1）若二叉排序树T为空，则创建一个key域为*k*的节点，将它作为根节点；

（2）否则将*k*和根节点的关键字比较，若两者相等，则说明树中已有此关键字*k*，无须插入，直接返回0；

（3）若*k*<T->key，则将*k*插入根节点的左子树中。

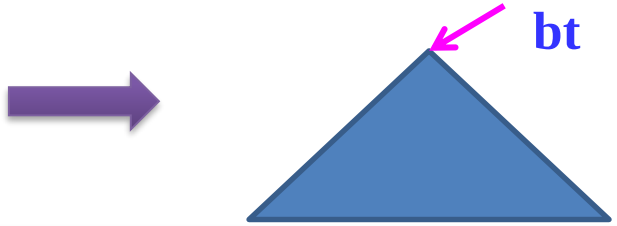
（4）否则将它插入右子树中。

对应的递归算法InsertBST()如下：

```
int InsertBST(BSTNode *&p,KeyType k)
{     if (p==NULL)                  //原树为空, 新插入的记录为根节点
      {    p=(BSTNode *)malloc(sizeof(BSTNode));
           p->key=k;p->lchild=p->rchild=NULL;
           return 1;
      }
      else if  (k==p->key)      //存在相同关键字的节点,返回0
          return 0;
      else if (k<p->key)
          return InsertBST(p->lchild,k);          //插入到左子树中
      else
          return InsertBST(p->rchild,k);          //插入到右子树中
}
```

先序遍历的思想

关键字数组
$A[0..n-1]$




```
BSTNode *CreatBST(KeyType A[],int n) //返回树根指针
{     BSTNode *bt=NULL;    //初始时bt为空树
      int i=0;
      while (i<n)
      {    InsertBST(bt,A[i]);  //将A[i]插入二叉排序树T中
           i++;
      }
      return bt;                    //返回建立的二叉排序树的根指针
}
```

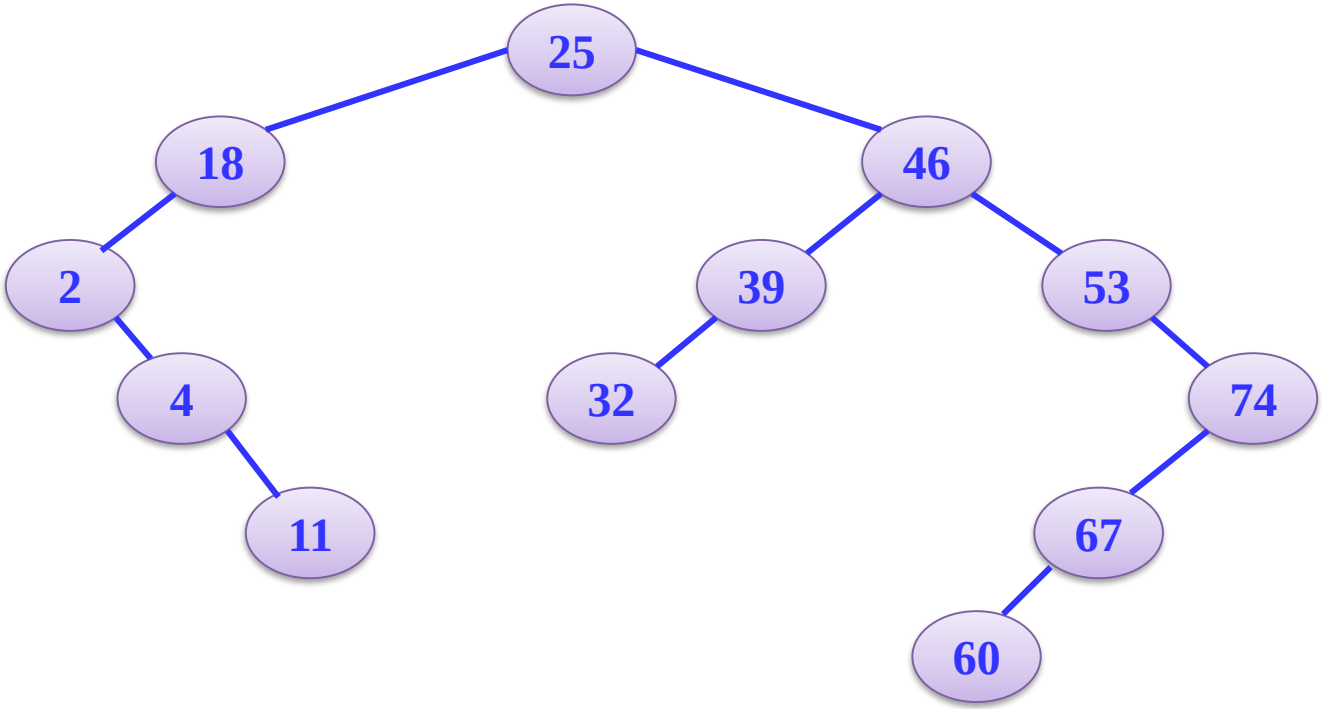
注意：任何节点插入到二叉排序树时，都是以叶节点插入的。

【例9-2】已知一组关键字为{25，18，46，2，53，39，32，4，74，67，60，11}。

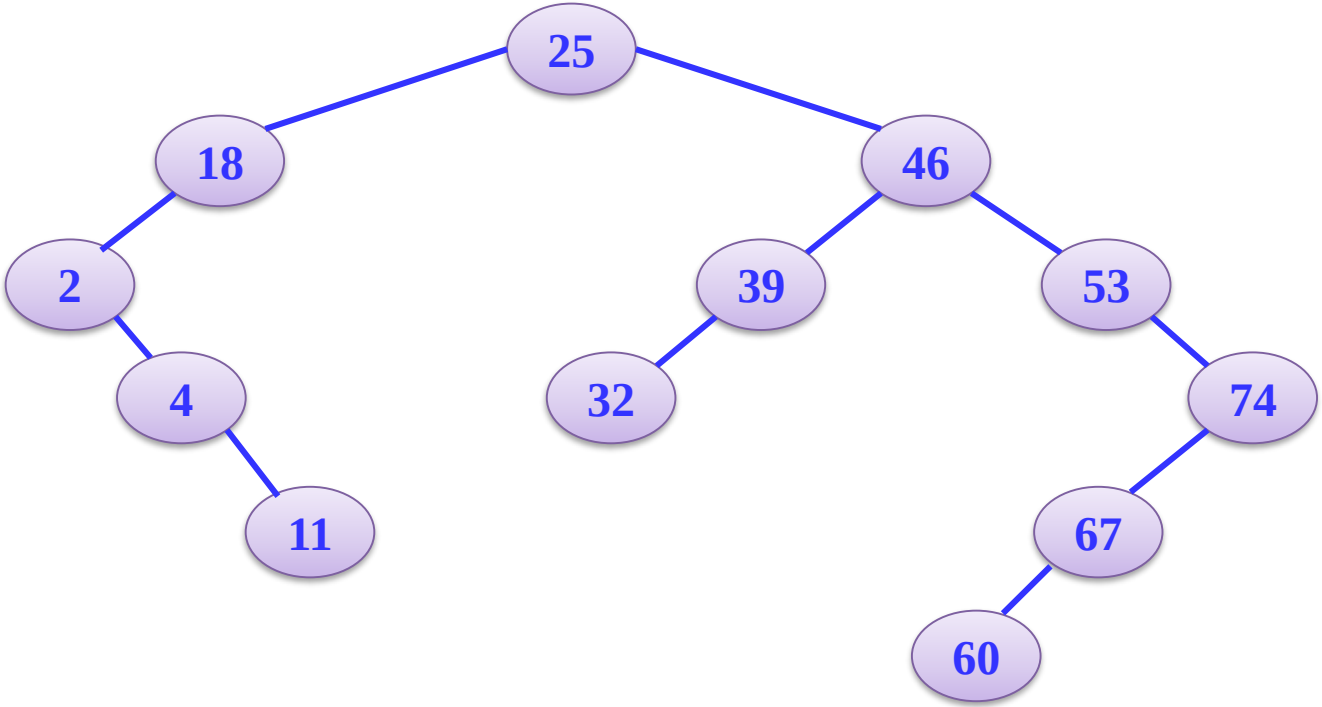
按表中的元素顺序依次插入到一棵初始为空的二叉排序树中，画出该二叉排序树。

求在等概率的情况下查找成功的平均查找长度和查找不成功的平均查找长度。

序列： 25 18 46 2 53 39 32 4 74 67 60 11

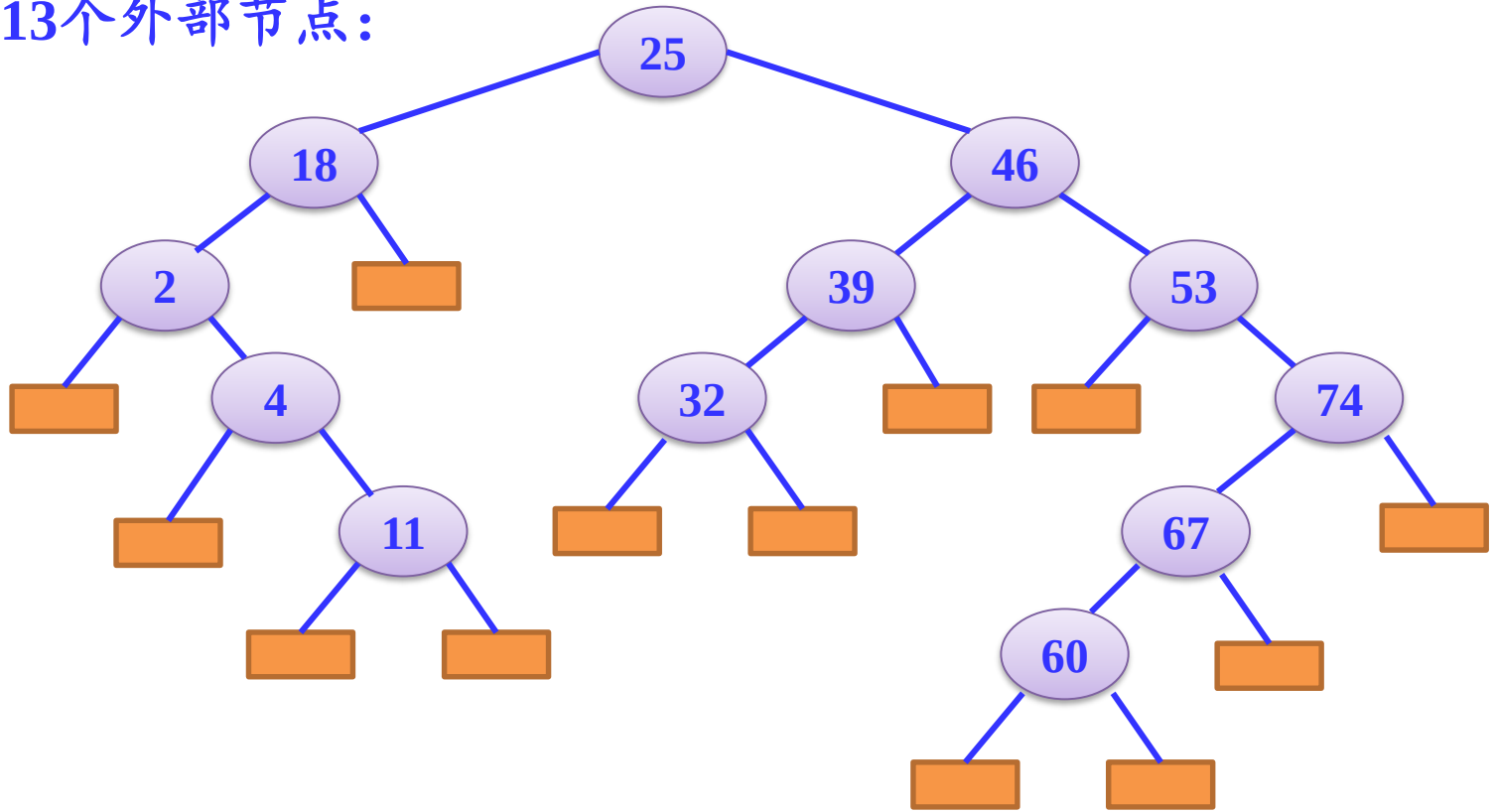


二叉排序树创建完毕



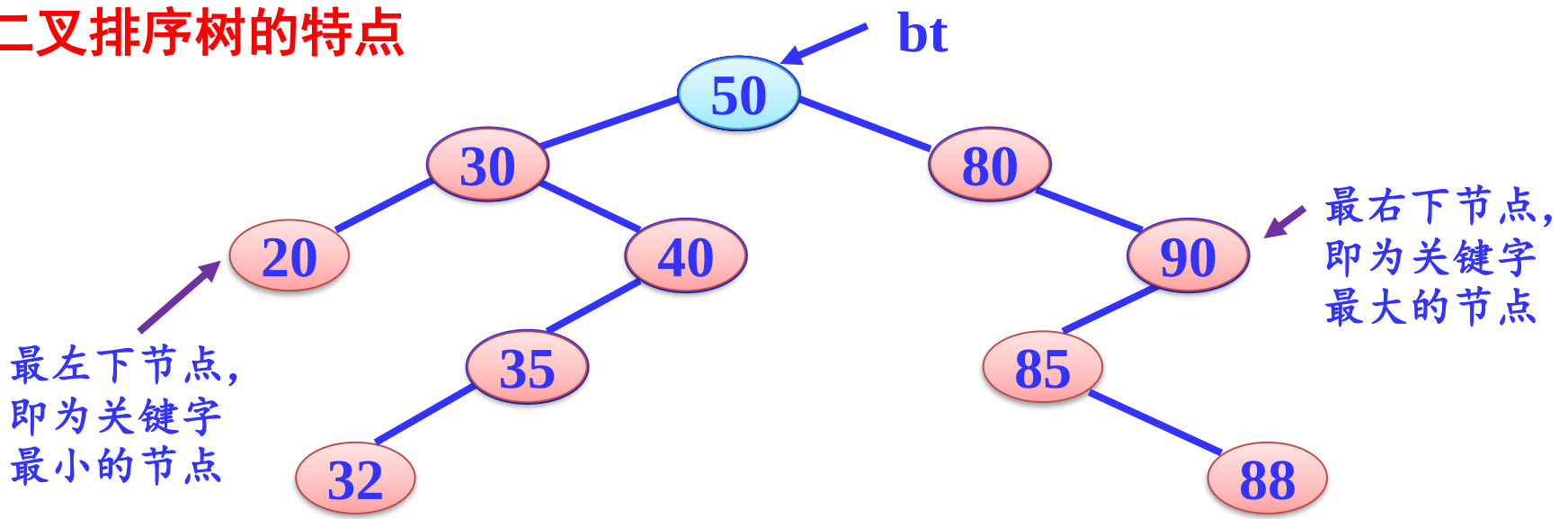
$$ASL_{成功} = \frac{1\times1+2\times2+3\times3+3\times4+2\times5+1\times6}{12} = 3.5$$

加上13个外部节点：



$$ASL_{\text{不成功}} = \frac{1\times2+3\times3+4\times4+3\times5+2\times6}{13} = 4.15$$

二叉排序树的特点



中序序列： 20，30，32，35，40，50，80，85，88，90

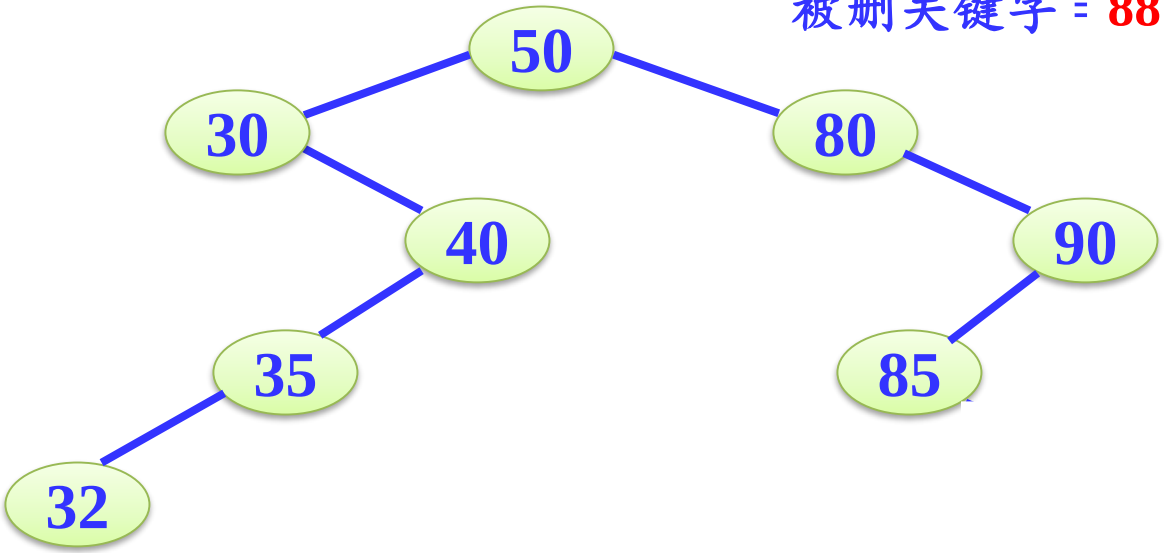
- 二叉排序树的中序序列是一个递增有序序列
- 根节点的最左下节点是关键字最小的节点
- 根节点的最右下节点是关键字最大的节点

3、 二叉排序树的节点删除

（1）被删除的节点是叶子节点：直接删去该节点。

例如：

被删关键字 = 88

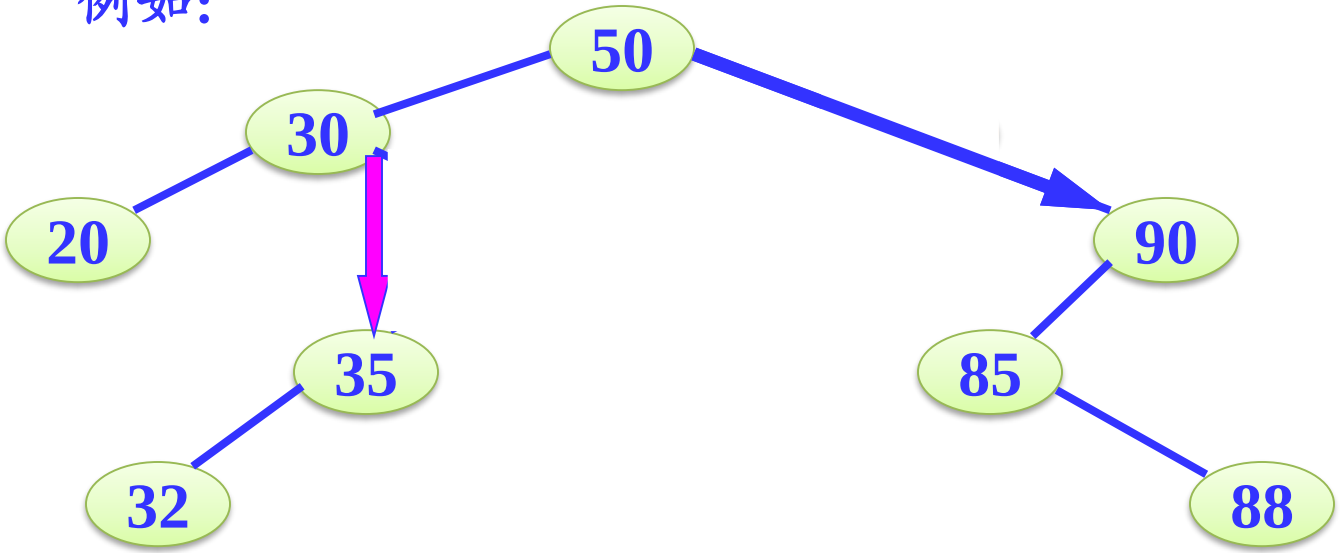


其双亲节点中相应指针域的值改为“空”

（2）被删除的节点只有左子树或者只有右子树，用其左子树或者右子树替换它（节点替换）。

被删关键字 = 80

例如：

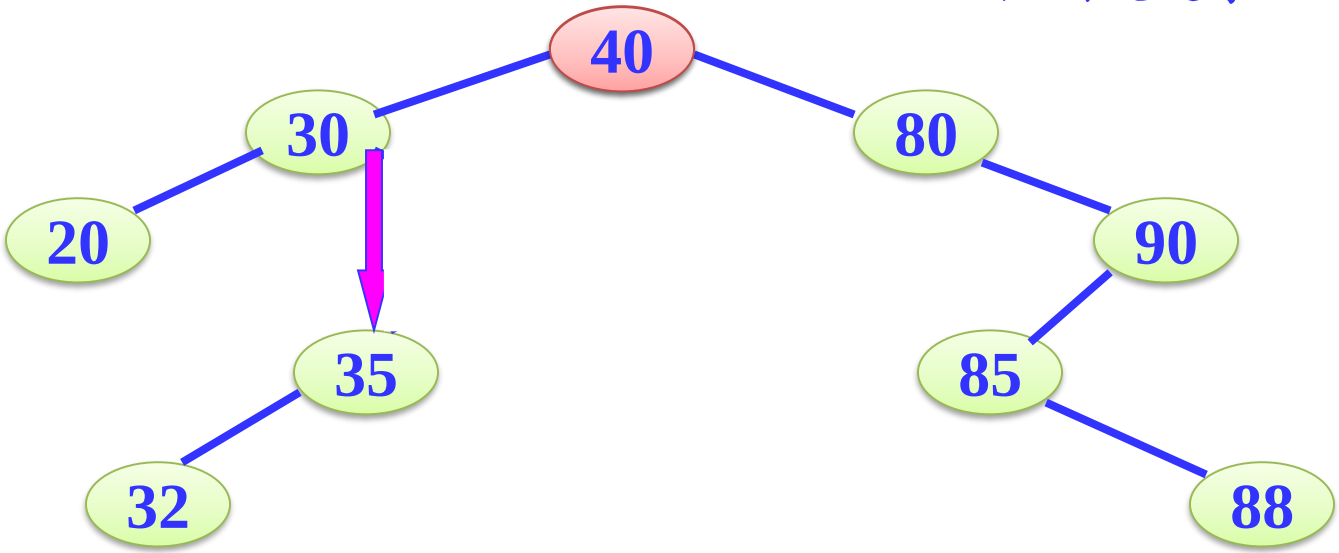


其双亲节点的相应指针域的值改为 “指向被删除节点的左子树或右子树”。

（3）被删除的节点既有左子树，也有右子树

例如：

被删关键字 = 50



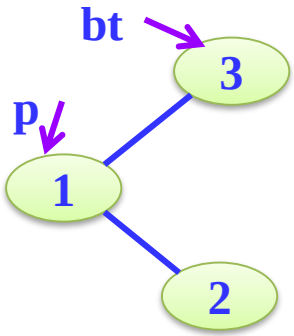
以其中序前趋值替换之（值替换），然后再删除该前趋节点。前趋是左子树中最大的节点。

也可以用其后继替换之，然后再删除该后继节点。后继是右子树中最小的节点。

算法实现：如何删除仅仅有右子树的节点*p：

❶ 查找被删节点

```
int deletek(BSTNode *&bt,KeyType k)
{    if (bt!=NULL)
     {    if (k==bt->key)
          {    deletep(bt)
               return 1;
          }
          else if (k<bt->key)
               deletek(bt->lchild,k);
          else
               deletek(bt->rchild,k);
     }
     else
          return 0;
}
```



deletek(bt,1)

⬇ 1<3

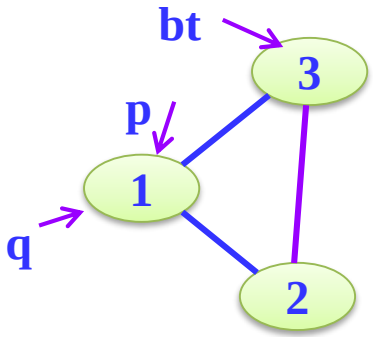
deletek(bt->lchild,1)

⬇ 1=1

deletep(p)

❷ 删除节点*p

```
void deletep(BSTNode *&p)
{    BSTNode *q;
     q=p;
     p=p->rchild;
        //用其右孩子节点替换它
     free(q);
}
```



达到用*p的右孩子节点替换它的目的

deletek(bt,1)

↓

deletek(bt−>lchild,1)

↓ ↑ bt−>lchild=p

deletep(p)

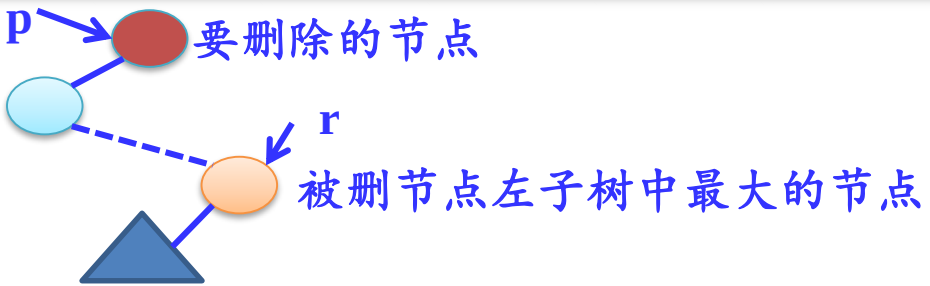
在二叉排序树bt中删除节点的算法

```
int DeleteBST(BSTNode *&bt,KeyType k)  //在bt中删除关键字为k的节点
{       if (bt==NULL) return 0;          //空树删除失败
        else
        {     if (k<bt->key) return DeleteBST(bt->lchild,k);
                                          //递归在左子树中删除为k的节点
              else if (k>bt->key) return DeleteBST(bt->rchild,k);
                                          //递归在右子树中删除为k的节点
              else   //bt->key=k
              {       Delete(bt);         //调用Delete(bt)函数删除*bt节点
                      return 1;
              }
      }
}
```

```
void Delete(BSTNode *&p)                    //从二叉排序树中删除*p节点
{    BSTNode *q;
     if (p->rchild==NULL)                   //*p节点没有右子树的情况
     {    q=p; p=p->lchild;                 //用其左孩子节点替换它
          free(q);
     }
     else if (p->lchild==NULL)              //*p节点没有左子树的情况
     {    q=p; p=p->rchild;                 //用其右孩子节点替换它
          free(q);
     }
     else Delete1(p,p->lchild);
          //*p节点既没有左子树又没有右子树的情况
}
```

p指向待删除的节点　　r指向其左孩子节点

```
void Delete1(BSTNode *p,  BSTNode *&r)
 //当被删*p节点有左右子树时的删除过程
 {    BSTNode *q;
      if (r->rchild!=NULL)
            Delete1(p,r->rchild);                  //递归找*r的最右下节点
     else                                          //r指向最右下节点
     {      p->key=r->key;  p->data=r->data        //值替换
            q=r; r=r->lchild;                      //删除原*r节点
            free(q);                               //释放原*r的空间
     }
 }
```



——本讲完——