10.1　查找的基本概念

10.2　线性表的查找

10.3　树表的查找

10.4　哈希表查找

本章小结

# 10.1　查找的基本概念

被查找的对象是由一组统计构成的表或文件,而每个统计则由若干个数据项构成,并假设每个统计都有一种能惟一标识该统计的关键字。

在这种条件下,查找的定义是：给定一种值k,在具有n个统计的表中找出关键字等于k的统计。若找到,则查找成功,返回该统计的信息或该统计在表中的位置；不然查找失败,返回有关的指示信息。

采用何种查找措施?

(1) 使用哪种数据构造来表达“表”,即表中统计是按何种方式组织的。

(2) 表中关键字的顺序。是对无序集合查找还是对有序集合查找?

若在查找的同步对表做修改运算(如插入和删除),则相应的表称之为动态查找表,不然称之为静态查找表。

因为查找运算的主要运算是关键字的比较,所以一般把查找过程中对关键字需要执行的平均比较次数(也称为平均查找长度)作为衡量一种查找算法效率优劣的原则。平均查找长度ASL(Average Search Length)定义为:

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中,n是查找表中统计的个数。$p_i$是查找第i个统计的概率,一般地,均以为每个统计的查找概率相等,即$p_i=1/n(1\le i\le n)$,$c_i$是找到第i个统计所需进行的比较次数。

## 10.2 线性表的查找

在表的组织方式中,线性表是最简朴的一种。三种在线性表上进行查找的措施:

(1) 顺序查找

(2) 二分查找

(3) 分块查找。

因为不考虑在查找的同步对表做修改,故上述三种查找操作都是在静态查找表上实现的。

查找与数据的存储构造有关,线性表有顺序和链式两种存储构造。本节只简介以顺序表作为存储构造时实现的顺序查找算法。定义被查找的顺序表类型定义如下：

```
#define MAXL <表中最多统计个数>
typedef struct
{      KeyType key;            /*KeyType为关键字的数据类型*/
       InfoType data;          /*其他数据*/
 } NodeType;
typedef NodeType SeqList[MAXL]; /*顺序表类型*/
```

## 10.2.1　顺序查找

顺序查找是一种最简朴的查找措施。

它的基本思绪是：从表的一端开始,顺序扫描线性表,依次将扫描到的关键字和给定值k相比较,若目前扫描到的关键字与k相等,则查找成功；若扫描结束后,仍未找到关键字等于k的统计,则查找失败。

例如,在关键字序列为{3,9,1,5,8,10,6,7,2,4}的线性表查找关键字为6的元素。

顺序查找过程如下:

开始: 3 9 1 5 8 10 6 7 2 4

第1次比较: **3** 9 1 5 8 10 6 7 2 4

i=0

第2次比较: 3 **9** 1 5 8 10 6 7 2 4

i=1

第3次比较: 3 9 **1** 5 8 10 6 7 2 4

i=2

第4次比较: 3 9 1 **5** 8 10 6 7 2 4

i=3

第5次比较: 3 9 1 5 **8** 10 6 7 2 4

i=4

第6次比较: 3 9 1 5 8 **10** 6 7 2 4

i=5

第7次比较: 3 9 1 5 8 10 6 7 2 4

i=6

查找成功,返回序号6

顺序查找的算法如下(在顺序表R[0..n-1]中查找关键字为k的统计,成功时返回找到的统计位置,失败时返回-1):

```
int SeqSearch(SeqList R,int n,KeyType k)
{   int i=0;
    while (i<n && R[i].key!=k) i++;   /*从表头往后找*/
    if (i>=n)
        return -1;
    else
        return i;
}
```

从顺序查找过程能够看到(不考虑越界比较 $i<n$),$c_i$取决于所查统计在表中的位置。如查找表中第1个统计R[0]时,仅需比较一次;而查找表中最终一种统计R[n-1]时,需比较n次,即$c_i=i$。所以,成功时的顺序查找的平均查找长度为:

$$ASL_{sq}=\sum_{i=1}^{n}p_i c_i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

查找成功时的平均比较次数约为表长的二分之一。

## 10.2.2 二分查找

二分查找也称为折半查找要求线性表中的结点必须已按关键字值的递增或递减顺序排列。它首先用要查找的关键字k与中间位置的结点的关键字相比较，这个中间结点把线性表提成了两个子表，若比较成果相等则查找完毕；若不相等，再根据k与该中间结点关键字的比较大小拟定下一步查找哪个子表，这么递归进行下去，直到找到满足条件的结点或者该线性表中没有这么的结点。

- 用Low、High和Mid表达待查找区间的下界、上界和中间位置指针，初值为Low=0，High=n-1。

(1)　取中间位置Mid：Mid=$\lfloor(Low+High)/2\rfloor$；

(2)　比较中间位置统计的关键字与给定的K值：

①　相等：　查找成功；

②　不小于：待查统计在区间的前半段，修改上界指针：　High=Mid-1，转(1)；

③　不不小于：待查统计在区间的后半段，修改下界指针：Low=Mid+1，转(1)；

直到越界(Low>High)，查找失败。

例如,在关键字有序序列{2,4,7,9,10,14,18,26,32,40}中采用二分查找法查找关键字为7的元素。

二分查找过程如下：

开始: 2 4 7 9 10 14 18 26 32 40

low=0 high=9 mid=(0+9)/2=4

第1次比较: 2 4 7 9 10 14 18 26 32 40

low=0 high=3 mid=(0+3)/2=1

第2次比较: 2 4 7 9 10 14 18 26 32 40

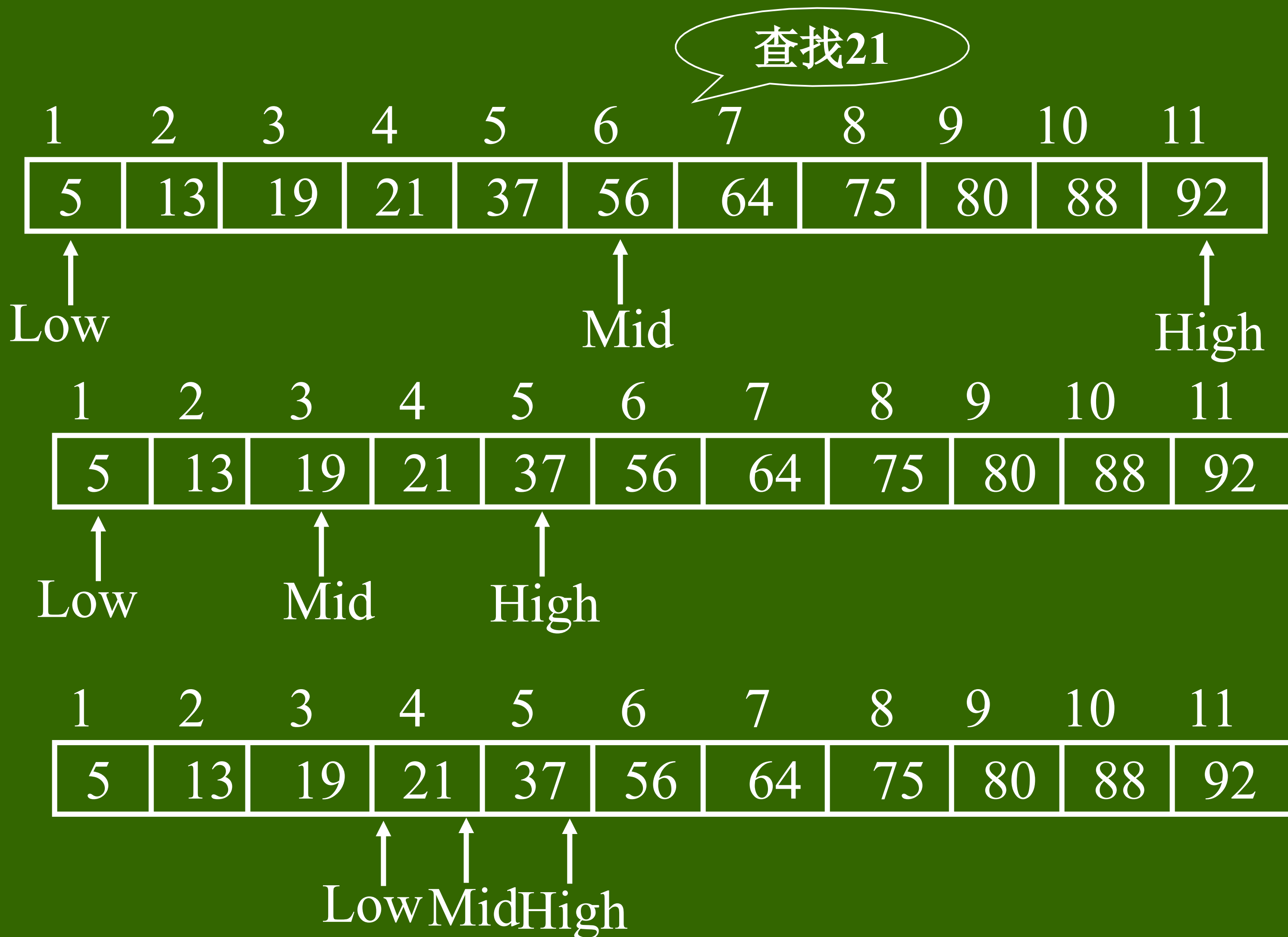
low=2 high=3 mid=(2+3)/2=2

第3次比较: 2 4 7 9 10 14 18 26 32 40

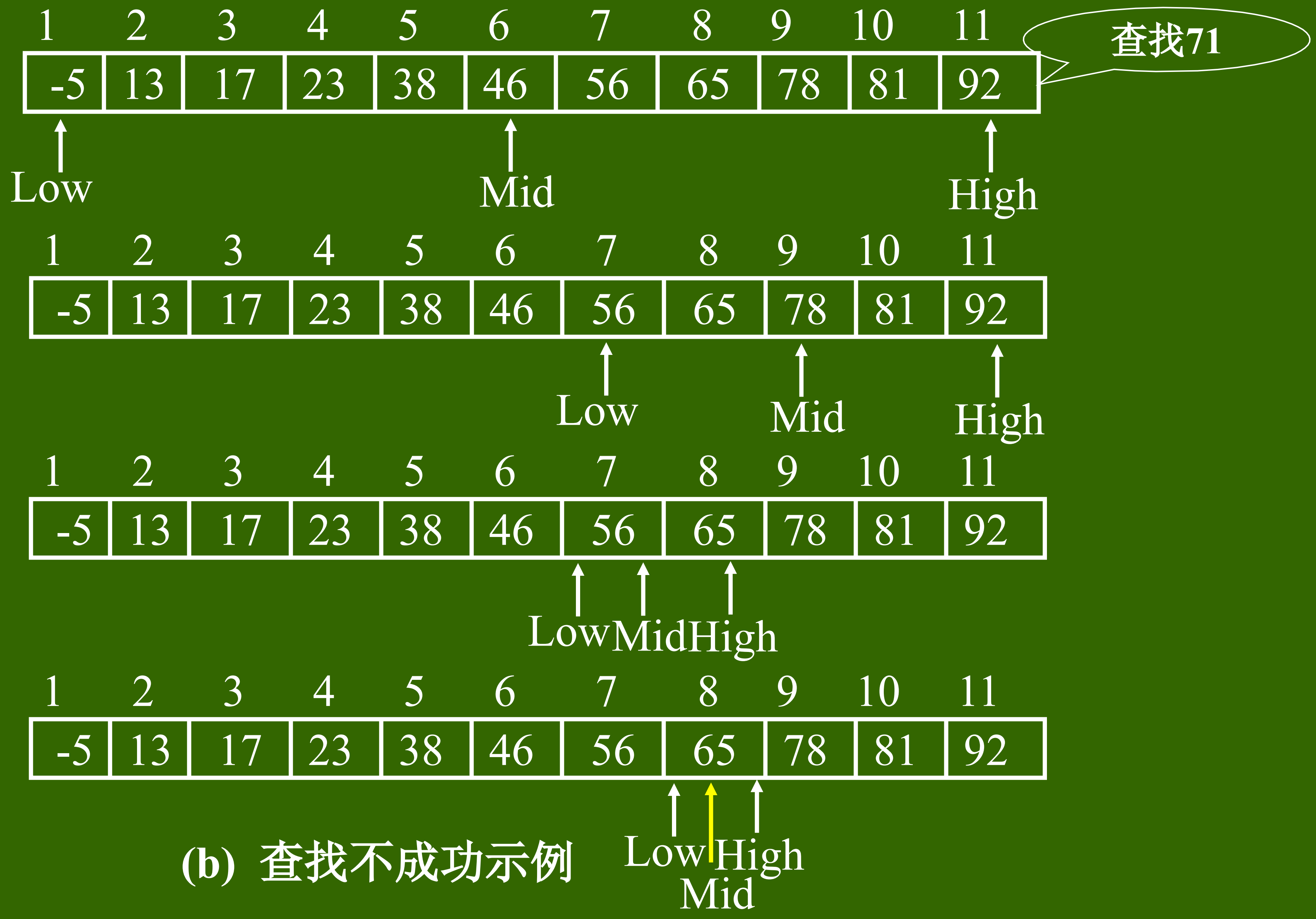
R[2].key=7

查找成功,返回序号2

**示例** 如图9-2(a)，(b)所示。




(a) 查找成功示例




**(b) 查找不成功示例**

**图9-2 折半查找示例**

其算法如下(在有序表R[0..n-1]中进行二分查找,成功时返回统计的位置,失败时返回-1):

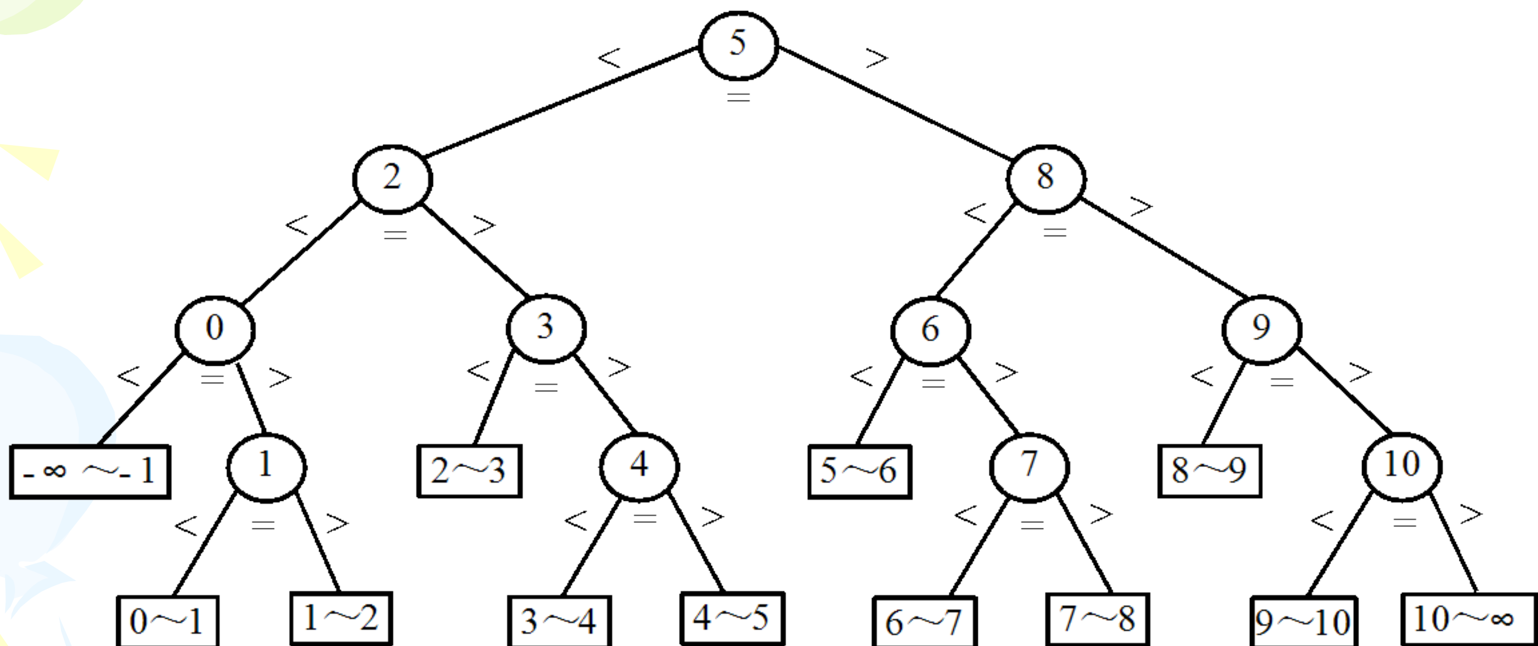
```
int BinSearch(SeqList R,int n,KeyType k)
{   int low=0,high=n-1,mid;
    while (low<=high)
    {   mid=(low+high)/2;
        if (R[mid].key==k)  /*查找成功返回*/
            return mid;
        if (R[mid].key>k)    /*继续在R[low..mid-1]中查找*/
            high=mid-1;
        else
            low=mid+1;    /*继续在R[mid+1..high]中查找*/
    }
    return -1;
}
```

二分查找过程可用二叉树来描述,我们把目前查找区间的中间位置上的统计作为根,左子表和右子表中的统计分别作为根的左子树和右子树,由此得到的二叉树,称为描述二分查找的**鉴定树**或**比较树**。

查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{n}\sum_{j=1}^{h} j \times 2^{j-1} = \frac{n+1}{n}\log_2(n+1)-1$$

当n很大 (n>50)时， $ASL \approx \log_2(n+1)-1$。




**R[0..10]的二分查找的鉴定树(n=11)**

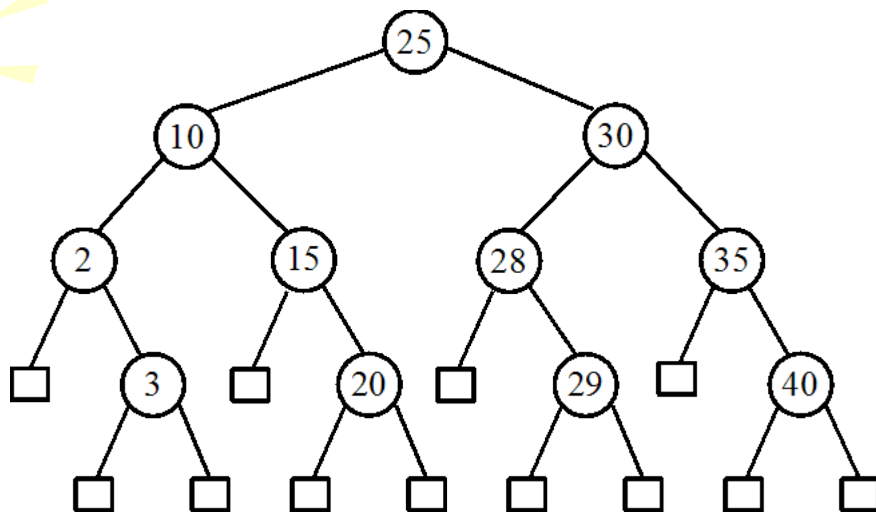
例 10.1对于给定11个数据元素的有序表{2,3,10,15,20,25,28,29,30,35,40},采用二分查找,试问:

(1)若查找给定值为20的元素,将依次与表中哪些元素比较?

(2)若查找给定值为26的元素,将依次与哪些元素比较?

(3)假设查找表中每个元素的概率相同,求查找成功时的平均查找长度和查找不成功时的平均查找长度。

二分查找鉴定树




(1)若查找给定值为20的元素,依次与表中25,10,15,20元素比较,共比较4次。

(2)若查找给定值为26的元素,依次与25,30,28元素比较,共比较3次。

(3)在查找成功时,会找到图中某个圆形结点,则成功时的平均查找长度:

$$ASLsucc = \frac{1\times1+2\times2+4\times3+4\times4}{11} = 3$$

## 10.2.3 分块查找

分块查找又称索引顺序查找,它是一种性能介于顺序查找和二分查找之间的查找措施。

**1 查找表的组织**

① 将查找表提成几块。块间有序，即第i+1块的全部统计关键字均不小于(或不不小于)第i块统计关键字；块内无序。

② 在查找表的基础上附加一种索引表，索引表是按关键字有序的，索引表中统计的构成是：

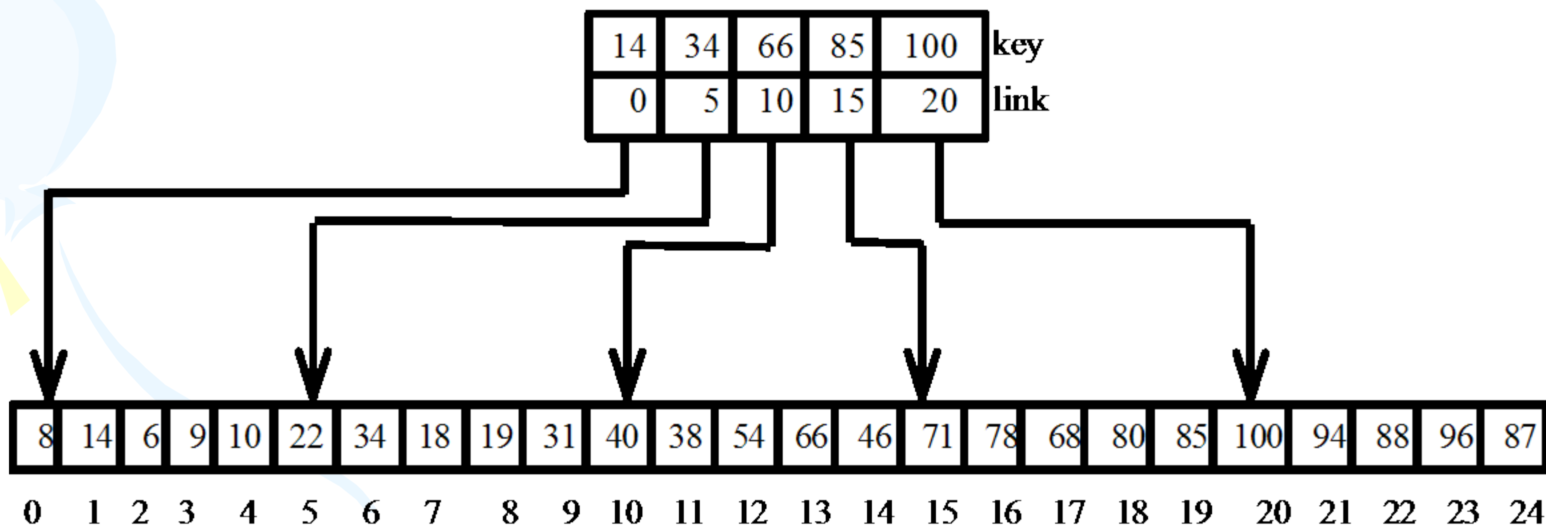
| 最大关键字 |
|---|
| 起始指针 |

**2 查找思想**

先拟定待查统计所在块，再在块内查找(顺序查找)。

```
#define MAXI <索引表的最大长度>
typedef struct
{    KeyType key;          /*KeyType为关键字的类型*/
     int link;             /*指向相应块的起始下标*/
} IdxType;
typedef IdxType IDX[MAXI];        /*索引表类型*/
```

例如,设有一种线性表,其中包括25个统计,其关键字序列为{8,14,6,9,10,22,34,18,19,31,40,38,54,66,46,71,78,68,80,85,100,94,88,96,87}。假设将25个统计分为5块,每块中有5个统计,该线性表的索引存储构造如下图所示。



采用二分查找索引表的分块查找算法如下(索引表I的长度为m)：

```
int IdxSearch(IDX I,int m,SeqList R,int n,KeyType k)
{     int low=0,high=m-1,mid,i;
      int b=n/m;                    /*b为每块的统计个数*/
      while (low<=high)             /*在索引中二分查找*/
      {     mid=(low+high)/2;
            if  (I[mid].key>=k)   high=mid-1;
            else  low=mid+1;
      }
```

```
if (low<m)   /*在块中顺序查找*/
{      i=I[low].link;
       while (i<=I[low].link+b-1 && R[i].key!=k) i++;
       if (i<=I[low].link+b-1)
              return i;
       else
            return -1;
}
return -1;
}
```

**设表长为n个统计，均分为b块，每块统计数为s，则b=[n/s]。设统计的查找概率相等，每块的查找概率为1/b，块中统计的查找概率为1/s，则平均查找长度ASL：**

$$ASL=L_b+L_w=\sum_{j=1}^{b} j+\frac{1}{s}\sum_{i=1}^{s} i=\frac{b+1}{2}+\frac{s+1}{2}$$

# 查找措施比较

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

## 10.3　树表的查找

当表的插入或删除操作频繁时,为维护表的有序性,需要移动表中诸多统计。这种由移动统计引起的额外时间开销,就会抵消二分查找的优点。也就是说,二分查找只合用于静态查找表。若要对动态查找表进行高效率的查找,可采用下面简介的几种特殊的二叉树或树作为表的组织形式,在这里将它们统称为树表。下面将分别讨论在这些树表上进行查找和修改操作的措施。

## 10.3.1　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的二叉树：

(1) 若它的左子树非空,则左子树上全部统计的值均不不小于根统计的值；

(2) 若它的右子树非空,则右子树上全部统计的值均不小于根统计的值；

(3) 左、右子树本身又各是一棵二叉排序树。

结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一种递增序列。

BST依然能够用二叉链表来存储

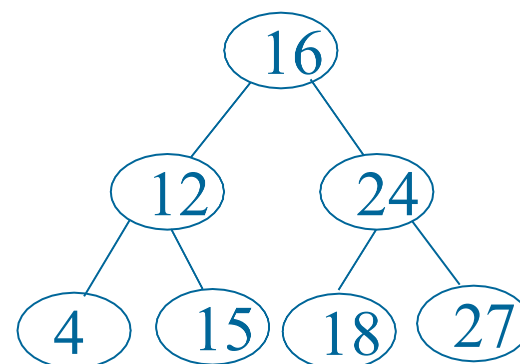



图9-4 二叉排序树

在讨论二叉排序树上的运算之前,定义其结点的类型如下：

```
typedef struct node                              /*统计类型*/
{    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

因为二叉排序树可看做是一种有序表,所以在二叉排序树上进行查找,和二分查找类似,也是一种逐渐缩小查找范围的过程。

**查找思想:**

首先将给定的K值与二叉排序树的根结点的关键字进行比较：若相等： 则查找成功;

① 给定的K值不不小于BST的根结点的关键字：继续在该结点的左子树上进行查找;

② 给定的K值不小于BST的根结点的关键字：继续在该结点的右子树上进行查找。

- 递归查找算法SearchBST()如下(在二叉排序树bt上查找关键字为k的统计,成功时返回该结点指针,不然返回NULL):
- BSTNode *SearchBST(BSTNode *bt,KeyType k)
- { if (bt==NULL || bt->key==k) /*递归终止条件*/
- 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*/
- }

## 2. 二叉排序树的插入和生成

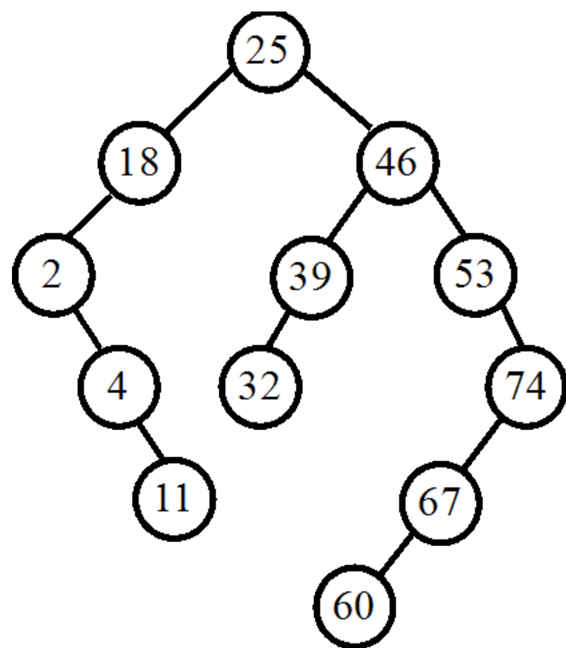
在二叉排序树中插入一种新统计,要确保插入后仍满足BST性质。其插入过程是：若二叉排序树T为空,则创建一种key域为k的结点,将它作为根结点；不然将k和根结点的关键字比较,若两者相等,则阐明树中已经有此关键字k,不必插入,直接返回0；若k<T->key,则将k插入根结点的左子树中,不然将它插入右子树中。相应的递归算法InsertBST()如下：

```
int InsertBST(BSTNode *&p,KeyType k)
/*在以*p为根结点的BST中插入一种关键字为k的结点。插入成功返回1,不然返回0*/
{   if (p==NULL)     /*原树为空, 新插入的统计为根结点*/
    {  p=(BSTNode *)malloc(sizeof(BSTNode));
       p->key=k;p->lchild=p->rchild=NULL;
       return 1;
    }
    else if  (k==p->key) /*存在相同关键字的结点,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*/
}
```

二叉排序树的生成,是从一种空树开始,每插入一种关键字,就调用一次插入算法将它插入到目前已生成的二叉排序树中。从关键字数组A[0..n-1]生成二叉排序树的算法CreatBST()如下：

```
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*/
}
```

例10.2 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,并求在等概率的情况下查找成功的平均查找长度。



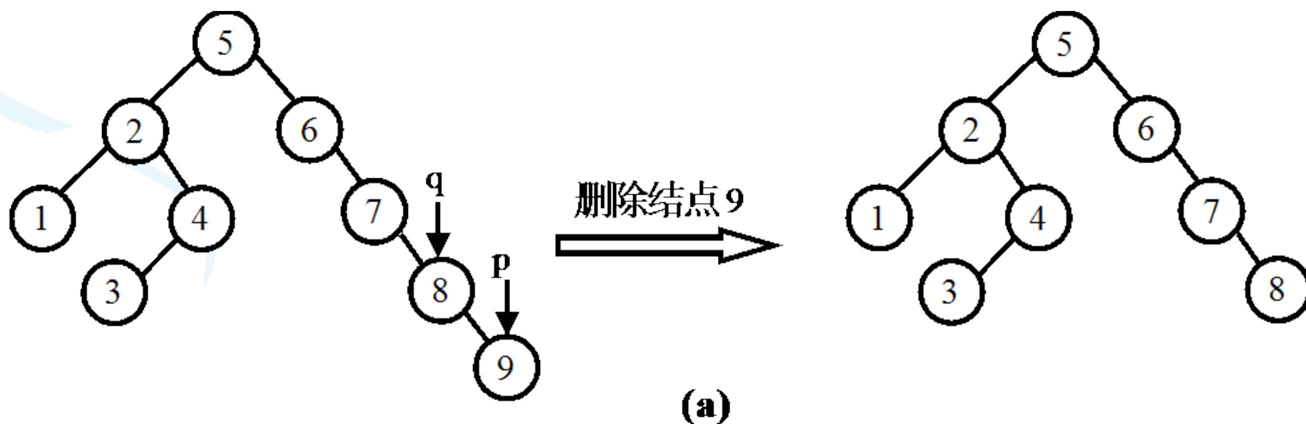

解：生成的二叉排序树如右图所示。

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

## 3. 二叉排序树的删除

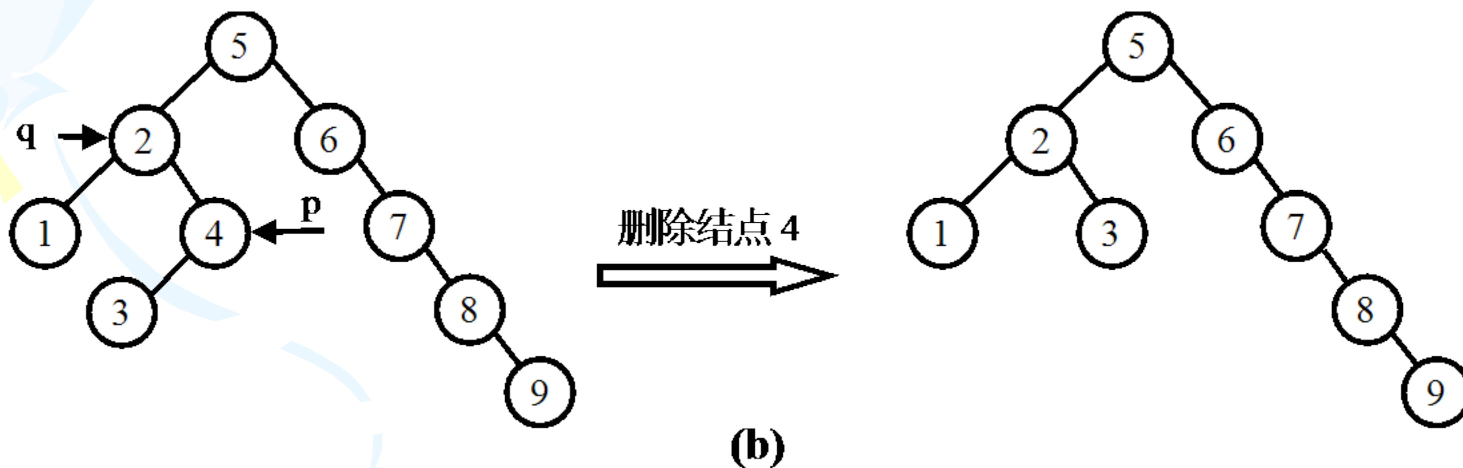
删除操作必须首先进行查找,假设在查找过程结束时,已经保存了待删除结点及其双亲结点的地址。指针变量p指向待删除的结点,指针变量q指向待删除结点p的双亲结点。删除过程如下:

(1) 若待删除的结点是叶子结点,直接删去该结点。如图(a)所示,直接删除结点9。这是最简朴的删除结点的情况。



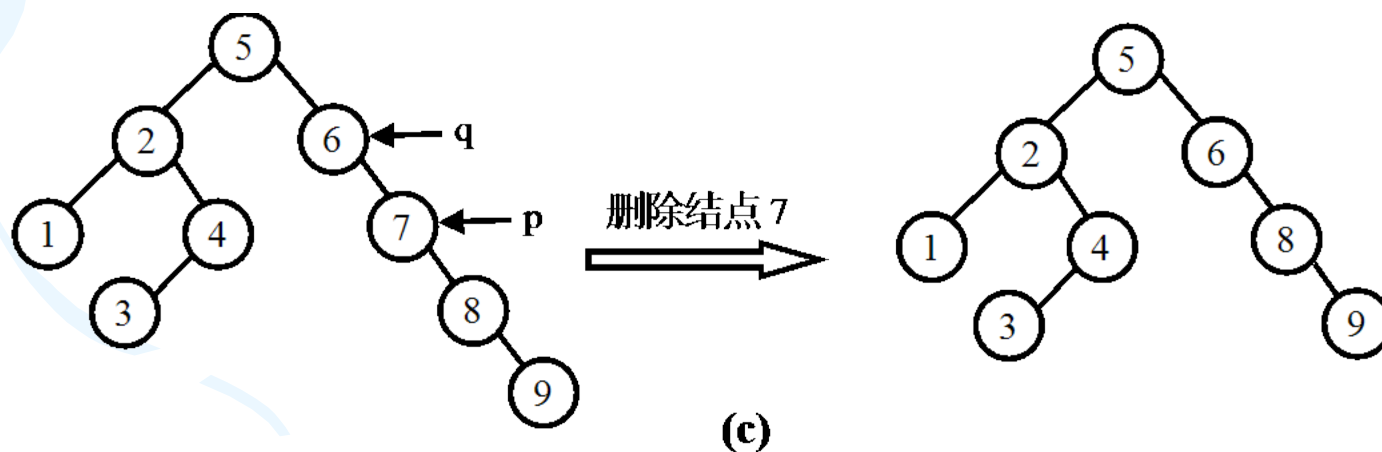
(a)

(2) 若待删除的结点只有左子树而无右子树。根据二叉排序树的特点,能够直接将其左子树的根结点放在被删结点的位置。如图(b)所示,*p作为*q的右子树根结点,要删除*p结点,只需将*p的左子树(其根结点为3)作为*q结点的右子树。



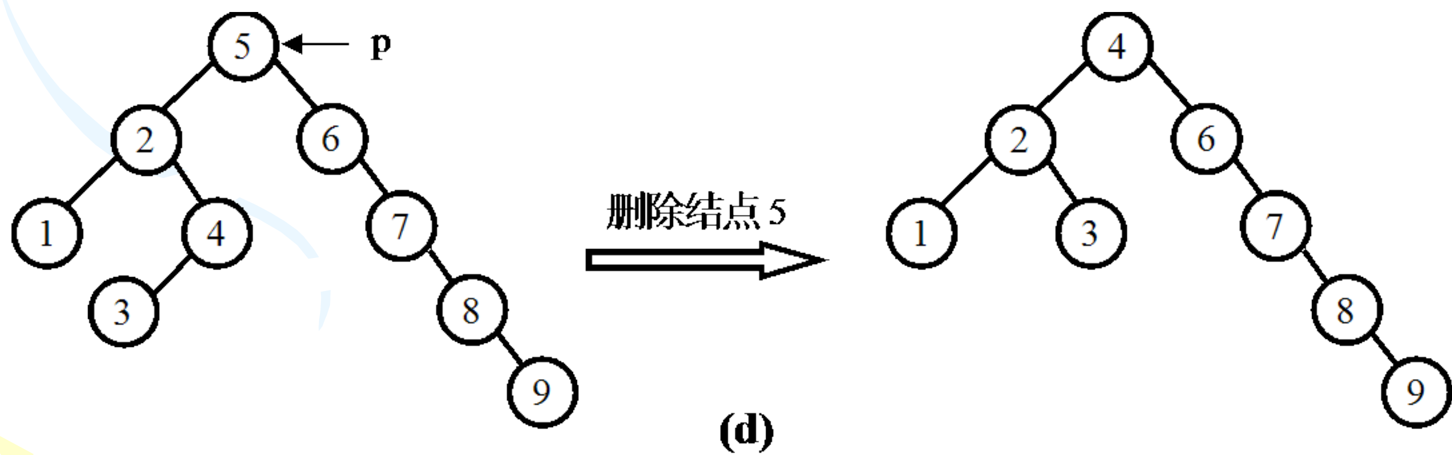
(b)

(3) 若待删除的结点只有右子树而无左子树。与(2)情况类似,能够直接将其右子树的根结点放在被删结点的位置。如图(c)所示,*p作为*q的左子树根结点,要删除*p结点,只需将*p的右子树(其根结点为8)作为*q结点的右子树。



(c)

(4) 若待删除的结点同步有左子树和右子树。根据二叉排序树的特点,能够从其左子树中选择关键字最大的结点或从其右子树中选择关键字最小的结点放在被删去结点的位置上。假如选用左子树上关键字最大的结点,那么该结点一定是左子树的最右下结点。如图(d)所示,若要删除*p(其关键字为5)结点,找到其左子树最右下结点4,它的双亲结点为2,用它替代*p结点,并将其原来的左子树(其根结点为3)作为原来的双亲结点2的右子树。



(d)

# BST树的删除

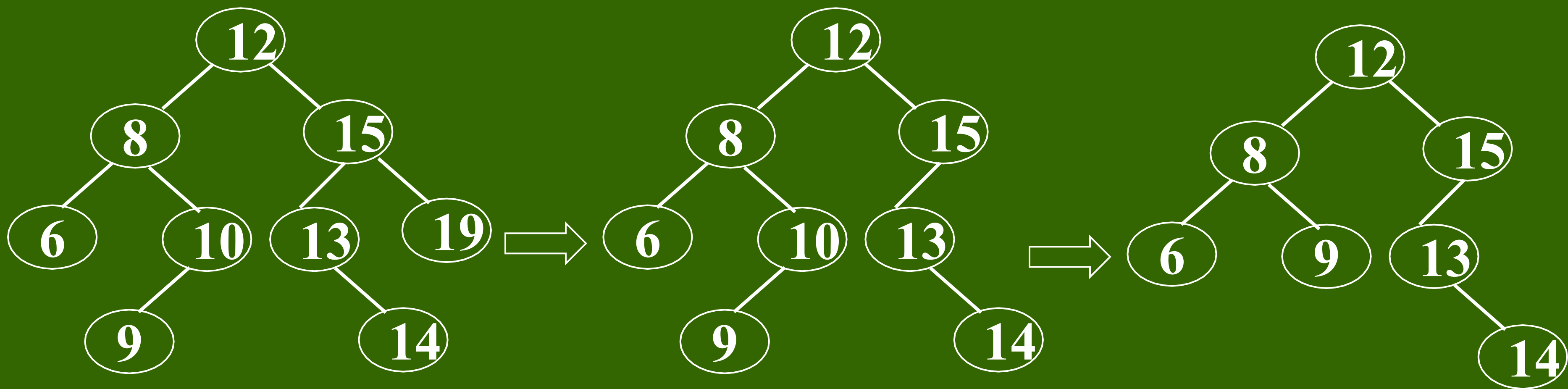
① 若p是叶子结点： 直接删除p，如图9-5(b)所示。

② 若p只有一棵子树(左子树或右子树)：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树。即原来p是f的左子树，则p的子树成为f的左子树；原来p是f的右子树，则p的子树成为f的右子树

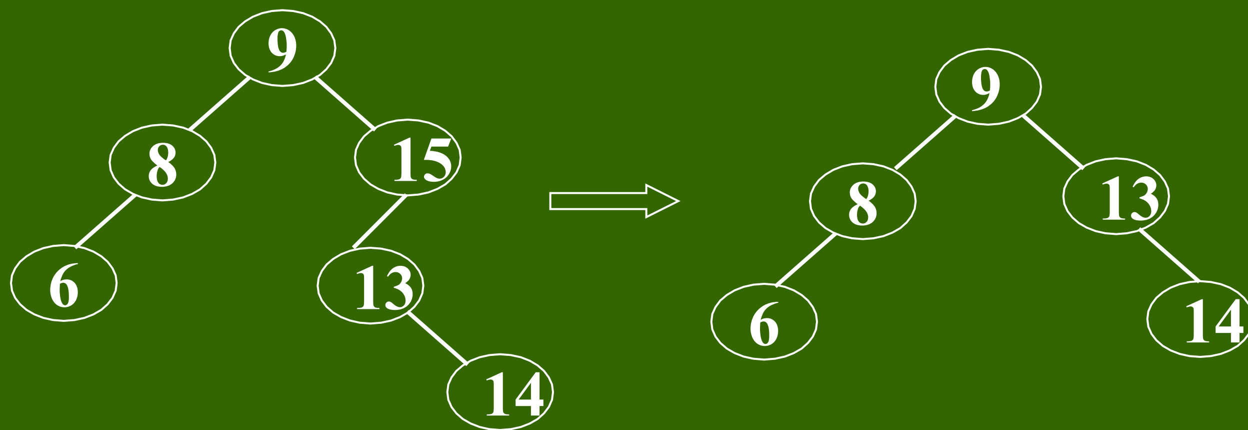
③ 若p既有左子树又有右子树：处理措施有下列两种，能够任选其中一种。

- ◆ 用p的直接前驱结点替代p。即从p的左子树中选择值最大的结点s放在p的位置(用结点s的内容替代结点p内容)，然后删除结点s。s是p的左子树中的最右边的结点且没有右子树，对s的删除同②

- ◆ 用p的直接后继结点替代p。即从p的右子树中选择值最小的结点s放在p的位置(用结点s的内容替代结点p内容)，然后删除结点s。s是p的右子树中的最左边的结点且没有左子树，对s的删除同②




(a) BST树　　(b) 删除结点19　　(c) 删除结点10




(e) 删除结点12　　(d) 删除结点15

图9-5 BST树的结点删除情况

删除二叉排序树结点的算法DeleteBST()如下(指针变量p指向待删除的结点,指针变量q指向待删除结点*p的双亲结点):

```
void Delete1(BSTNode *p,BSTNode *&r)
/*当被删*p结点有左右子树时的删除过程*/
{     BSTNode *q;
      if (r->rchild!=NULL)
            Delete1(p,r->rchild);    /*递归找最右下结点*/
      else     /*找到了最右下结点*r*/
      {     p->key=r->key;           /*将*r的关键字值赋给*p*/
            q=r;    r=r->lchild;
             /*将左子树的根结点放在被删结点的位置上*/
            free(q);                 /*释放原*r的空间*/
      }
}
```

```
void Delete(BSTNode *&p)   /*从二叉排序树中删除*p结点*/
{   BSTNode *q;
    if (p->rchild==NULL)     /**p结点没有右子树的情况*/
    {  q=p;  p=p->lchild;
           /*其右子树的根结点放在被删结点的位置上*/
       free(q);
    }
    else if (p->lchild==NULL)   /**p结点没有左子树*/
    {  q=p;   p=p->rchild;
           /*将*p结点的右子树作为双亲结点的相应子树/
       free(q);
    }
    else Delete1(p,p->lchild);
         /**p结点既没有左子树又没有右子树的情况*/
}
```

```
int DeleteBST(BSTNode *&bt,KeyType k)
/*在bt中删除关键字为k的结点*/
{   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的结点*/
        else   if (k>bt->key)   return DeleteBST(bt->rchild,k);
                /*递归在右子树中删除为k的结点*/
        else
        {     Delete(bt);   /*调用Delete(bt)函数删除*bt结点*/
              return 1;
        }
    }
}
```

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/188006111015006137