第九章 查找

1

查找表：由同一类型的数据元素（记录）组成的集合。

记作：ST={a1, a2, . . . , an}

学生成绩表

序号	学　号	姓　名	性别	数学	外语
1	200041	刘大海	男	80	75
2	200042	王　伟	男	90	83
3	200046	吴晓英	女	82	88
4	200048	王　伟	女	80	90
n	……	…..	…	….	…

数据项1（主关键字）　数据项2　　数据项5

- 关键字：　可以标识一个记录的数据项。
- 主关键字：　可以唯一地标识一个记录的数据项。
- 次关键字：　可以识别若干记录的数据项。

2

查找表的操作：

生成查找表

查找某个“特定的”数据元素(记录)x是否在表ST中

查询某个“特定的”数据元素(记录)x的各种属性

插入新元素(记录)x

删除元素(记录)x

……

3

查找：根据给定的某个关键字值，在查找表中确定一个其关键字等于给定值的记录或数据元素。

设k为给定的一个关键字值，R[1..n]为n个记录的表，

若存在R[i].key=k, 1≤i≤n, 称查找成功；

否则称查找失败。

静态查找：查询某个特定的元素，或检查某个特定的数据元素的属性，但不插入新元素或删除元素。

动态查找：在查找过程中，同时插入查找表中不存在的数据元素(记录)。

4

- 查找表的类型及其查找方法

(1) 静态查找表

- 顺序表，用顺序查找法
- 线性链表，用顺序查找法
- 有序的顺序表（有序表），用：折半查找法；

  **斐班那契查找法；插值查找法；
- 索引顺序表/分块表，用分块查找法。

(2) 动态查找表

- 二叉排序树,平衡二叉树(AVL树)
- **红黑树，B树，B+树，键树

(3) 哈希(Hash)表

5

- 平均查找长度-----查找一个记录时比较关键字次数的平均值。

$$ASL = \sum_{i=1}^{n} P_i C_i$$

P_i --- 查找r[i]的概率

C_i --- 查找r[i]所需比较关键字的次数

6

本章数据元素类型与比较运算的符号约定

（1）数据类型定义

```
typedef struct {
        keytype key; //关键字
        ......       //其他域
}ElemType
```

（2）关键字比较的符号约定

```
EQ(a,b) ((a)= =(b))
LT(a,b) ((a)<(b))
LQ(a,b) ((a)<=(b))
```

7

9.1 静态查找表

静态查找表的抽象数据类型参见教材。

静态查找表是线性表吗？

针对静态查找表的查找算法主要有：

一、顺序查找（线性查找）

二、折半查找（二分或对分查找）

三、静态树表的查找

四、分块查找（索引顺序查找）

8

9.1.1 顺序表与顺序查找法

以顺序表或线性链表表示静态查找表。

假设静态查找表的顺序存储结构为

```
typedef struct {
  ElemType *elem;
    // 数据元素存储空间基址，建表时
    // 按实际长度分配，0号单元留空
  int      length;     // 表的长度
} SSTable;
```

9

例1 元素类型为记录(结构)

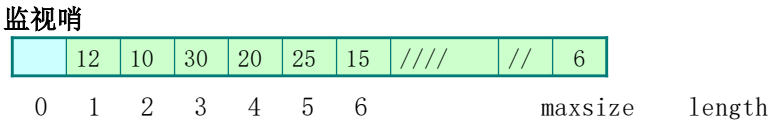
```
#define maxsize 100      //表长100
typedef struct node
  { keytype key ;        //关键字类型
    char name[6];        //姓名
    ......               //其它
  } ElemType;

typedef struct
  {
   ElemType elem[maxsize+1];
          //maxsize+1个记录的静态数组,elem[0]为监视哨
   int length;
  } SSTable;
SSTable ST1,ST2;
```

10

例2 元素类型为整型

```
#define maxsize 100 //表长100
typedef int ElemType;
typedef struct
 {
  ElemType elem[maxsize+1]; //maxsize+1个记录,
                            //elem[0]为监视哨
  int length;
  } SSTable;
SSTable ST1,ST2;
```



1

顺序查找法(sequential search)算法设计

算法1：假定不使用监视哨elem[0]

基本思想：

- ✓ 将关键字key依次与下列记录的关键字比较:

  elem [n].key, elem[n-1].key, ..., elem[1].key 。

- ✓ 如果找到一个记录elem[i],有:

  elem[i].key＝key(1≤i≤n),

  则查找成功，停止比较,返回记录的下标i;

- ✓ 否则,查找失败,返回0。

12

输入：查找表ST，查找条件（关键字）key
输出：成功时：记录序号，失败时：0

```
int search.seq(SSTable ST,keytype key)
{
  int i=ST.length;          //从第n个记录开始查找
  while (i>=1 && key!=ST.elem[i].key)
       i--;                          //继续扫描
  if (i)
     printf(" success\n" );          //查找成功
  else   printf(" fail\n" );         //查找失败
  return i;                      //返回记录的下标i
 }
```

13

算法2：假定使用监视哨elem[0]

基本思想：

- 先将关键字key存入elem[0].key,
- 再将key依次与
  elem[n].key,...,elem[1].key,elem[0].key
  进行比较。
- 如果找到一个记录,有：
  key=elem [i].key, (0≤i≤n)，则停止比较。
  - ✓ 如果 i>0,则查找成功;
  - ✓ 否则,查找失败。

14

```
int Search_Seq( SSTable  ST , KeyType  key ){
    //在顺序表ST中，查找关键字与key相同的元素；若成功，
    //返回其位置信息，否则返回0
  ST.elem[0].key =key;
   //设立哨兵，可免去查找过程中每一步都要检测是否查
   //完毕。当n>1000时，查找时间将减少一半。
  for( i=ST.length; ST.elem[ i ].key!=key;  - - i  );
     //不要用for(i=n; i>0; - -i) 或 for(i=1; i<=n; i++)
  return i;
   //若到达0号单元才结束循环，说明不成功，返回0值(i=0)。
   //成功时则返回找到的那个元素的位置i。
} // Search_Seq
```

15

查找算法性能分析：对n个记录的表，所需比较关键字的次数

$$ASL = \sum_{i=1}^{n} P_i C_i$$

分析：

查找第n个元素所需的比较次数为1；

查找第n-1个元素所需的比较次数为2；

……

查找第1个元素所需的比较次数为n；

对顺序表而言，

$C_i = n-i+1$

$$ASL = nP_1 + (n-1)P_2 + \ldots + 2P_{n-1} + P_n$$

在等概率查找的情况下，$P_i = \frac{1}{n}$

顺序表查找的平均查找长度为：

$$ASL_{ss} = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$$

16

在不等概率查找的情况下，ASL_{ss} 在

$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$

时取极小值。

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

顺序查找的特点：

优点：算法简单，且对顺序结构或链表结构均适用。

缺点： ASL 较长，时间效率较低。

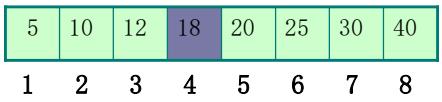
17

9.1.2 有序的顺序表的查找与折半查找法

1. 有序表

elem[1].key≤elem[2].key≤...≤elem[n].key

2. 折半查找(binary search,对半查找，二分查找)



low=1, high=8

mid=(low+hig)/2=4

先求位于查找区间中间对象的下标mid，用其关键码与给定值x比较:

Element[mid].key == x，查找成功；

Element[mid].key > x， 把查找区间缩小 到表的前半部分，继续折半查找；

Element[mid].key < x， 把查找区间缩小到表的后半部分，继续折半查找。

如果查找区间已缩小到一个对象，仍未找到想要查找的对象，则查找失败。

18

已知如下11个元素的有序表,请查找关键字为21和85的数据元素。

（05 13 19 21 37 56 64 75 80 88 92）

low mid high

① 先设定3个辅助标志: low, high, mid,

显然有: mid=⌊(low+high)/2⌋

② 运算步骤:

(1) low =1, high =11 , mid =6 , 待查范围是 [1, 11];

(2) 若 ST.elem[mid].key < key, 说明 key∈[mid+1, high] ,

则令: low =mid+1;重算 mid= ⌊(low+high)/2⌋; .

(3) 若 ST.elem[mid].key > key, 说明key∈[low , mid-1],

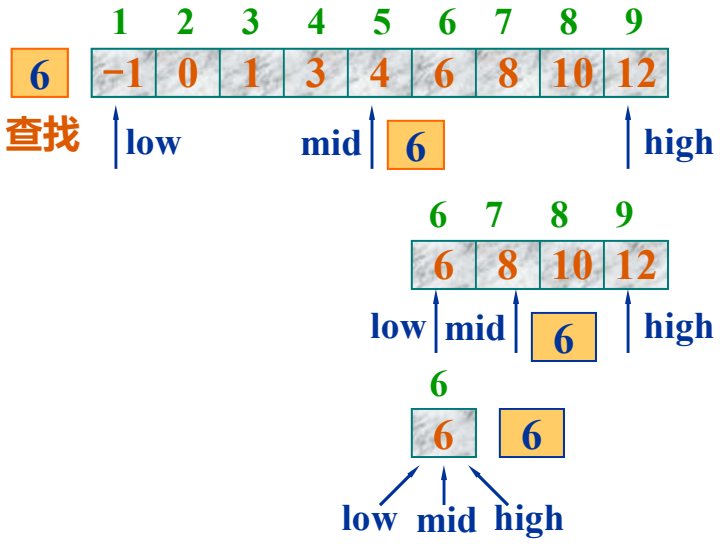
则令: high =mid-1;重算 mid ;

(4)若 ST.elem[mid].key = key, 说明查找成功, 元素序号=mid;

结束条件: (1) 查找成功 : ST.elem[mid].key = key

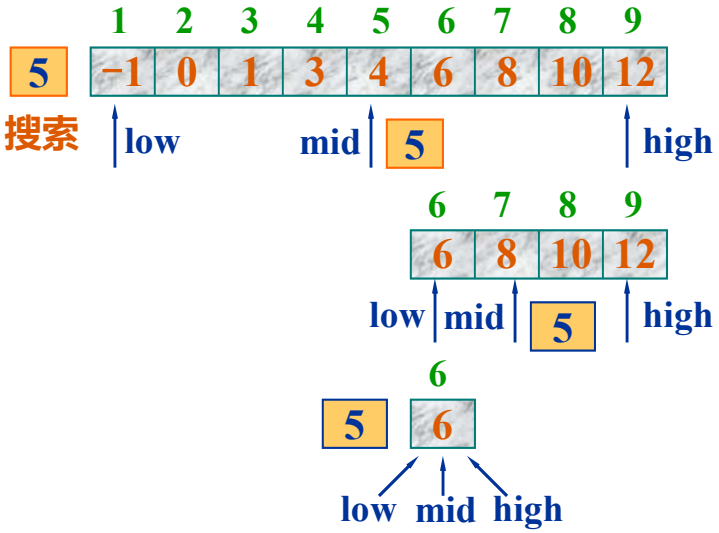
(2) 查找不成功 : high≤low （意即区间长度小于0）

19



查找成功的例子

20



搜索失败的例子

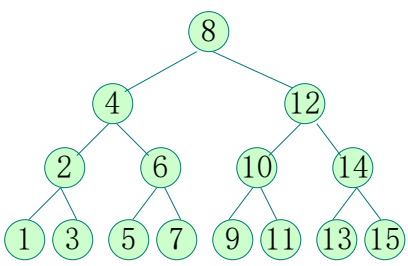
搜索失败的标志：high< low，此时停止查找。

21

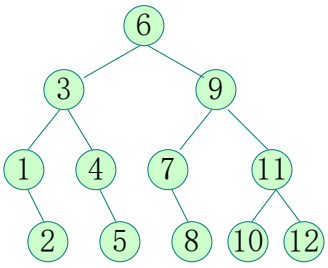
```
int Search_Bin ( SSTable ST, KeyType key ) {
  low = 1;  high = ST.length;        // 置区间初值
  while (low <= high) {
    mid = (low + high) / 2;
    if  ( EQ (key , ST.elem[mid].key) )
      return  mid;                   // 找到待查元素
    else  if ( LT (key , ST.elem[mid].key) )
      high = mid - 1;          // 继续在前半区间进行查找
    else  low = mid + 1;   // 继续在后半区间进行查找
  }
  return 0;                  // 顺序表中不存在待查元素
} // Search_Bin
```

22

4. 判定树（描述折半查找过程的二叉树）



n=15, 满二叉树



n=12, 非满二叉树

- 结点内的数据表示数据元素的序号(左图：1～ 15)
- 判定树由表中元素个数决定。
- 找到有序表中任一记录的过程：走了一条从根结点到与该记录相应的结点的路径。
- 比较的关键字个数：为该结点在判定树上的层次数。

23

一般情况下，表长为 n 的折半查找的判定树和含有n个结点的完全二叉树的深度相同：$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$

假设 $n=2^h-1$ 并且查找概率相等（深度为h），则

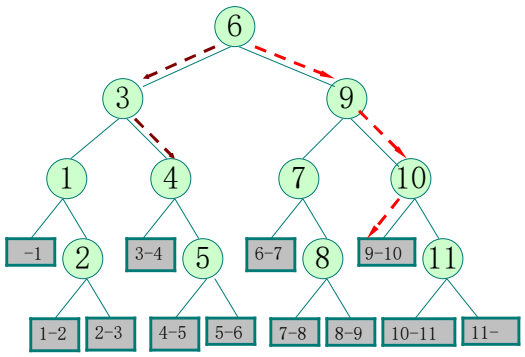
$$ASL_{bs} = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left[\sum_{j=1}^{h} j \times 2^{j-1}\right] = \frac{n+1}{n}\log_2(n+1) - 1$$

在n>50时，可得近似结果

$$ASL_{bs} \approx \log_2(n+1) - 1$$

当n=12，$ASL_{bs} \approx 3.1$.

24



n=11，加外部结点的判定树

11个元素的有序表：（05 13 19 21 37 56 64 75 80 88 92），查找关键字为21和85的数据元素。

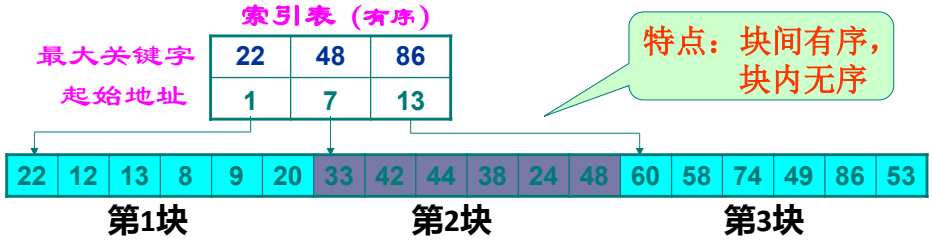
- 查找成功时检测指针停留在树中某个结点。
- 查找不成功时检测指针停留在某个外结点（失败结点）。
- 针对此表，具体估计查找失败的平均查找长度。

25

9.1.4 索引顺序表(分块表)与分块查找法

❑先让数据分块有序，即分成若干子表，要求每个子表中的数值（关键字）都比后一块中数值小（但子表内部未必有序）。

❑然后将各子表中的最大关键字构成一个索引表，表中含每个子表的起始地址（即头指针）。



查找分为两步(例如: k=38):

1. 确定待查记录所在块; (可以用顺序或折半查找)
2. 在块内顺序查找. (只能用顺序查找)

26

查找效率： $ASL_{bs}=L_b+L_w$

对索引表查找的ASL　　对块内查找的ASL

- 设查找表中n个元素可均匀地分为b块，每块含有s个记录
- 若在索引表和块内都进行顺序查找，则：

$$ASL_{bs}=L_b+L_w=\frac{1}{b}\sum_{j=1}^{b}j+\frac{1}{s}\sum_{i=1}^{s}i=\frac{1}{2}(\frac{n}{s}+s)+1$$

- 实际上，在索引表中可进行折半查找，则

$$ASL_{bs}\approx\log_2(\frac{n}{s}+1)+\frac{s}{2}\qquad(\log_2 n\le ASL_{bs}\le\frac{n+1}{2})$$

27

对比顺序表和有序表的查找性能

	顺序表	有序表
表的特性	无序	有序
存储结构	顺序 或 链式	顺序
插删操作	易于进行	需移动元素
ASL的值	大	小

28

9.2 动态查找表

特点：表结构在查找过程中动态生成。

要求：对于给定值key

✓若表中存在其关键字等于key的记录，则查找成功返回；

✓否则插入关键字等于key 的记录。

一、二叉排序树

二、平衡二叉树

三、B-树和B+树

29

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。
各个数据元素均含有类型相同，
可唯一标识数据元素的关键字。

数据关系**R**：数据元素同属一个集合。

基本操作**P**：

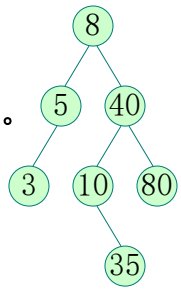
```
InitDSTable(&DT);
                    DestroyDSTable(&DT);
SearchDSTable(DT, key);
                    InsertDSTable(&DT, e);
DeleteDSTable(&DT, key);
                    TraverseDSTable(DT, Visit());
```

} ADT DynamicSearchTable

30

1. 二叉排序树(二叉查找树Binary Search Tree (BST))

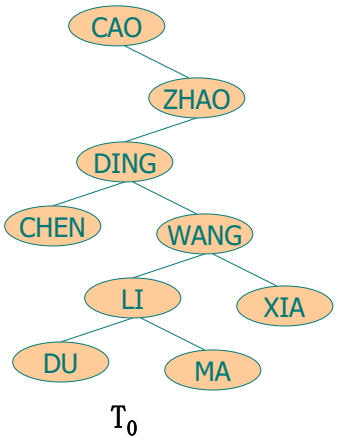
- 二叉排序树或是一棵空树，或是具有下列性质的二叉树：
  1) 若其左子树不空，则左子树上所有结点的值均小于它的根结点的值；
  2) 若其右子树不空，则右子树上所有结点的值均大于它的根结点的值
  3) 其左、右子树也分别为二叉排序树
- 二叉排序树的中序遍历序列一定是递增有序的。
- **The most common use of BST is to implement sets and tables.**



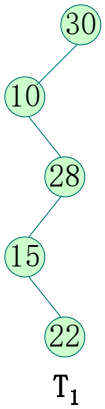
LDR：3, 5, 8, 10, 35, 40, 80

31

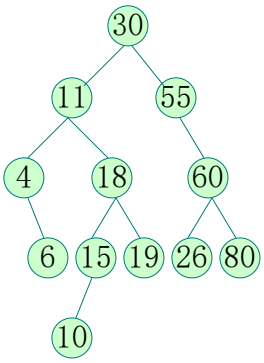
下列二叉树是否为二叉排序树？



T_0



T_1



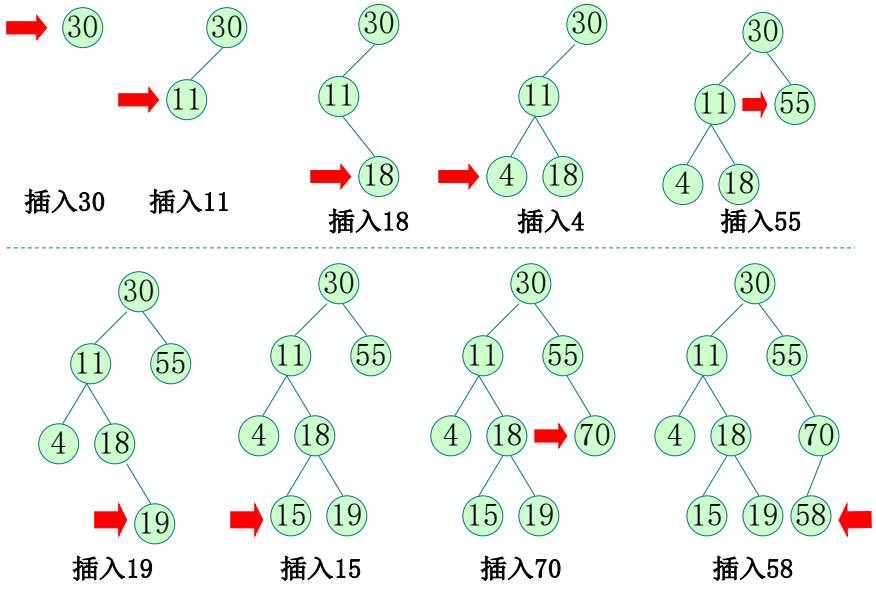
T_2

T_3

32

（2）二叉排序树的生成

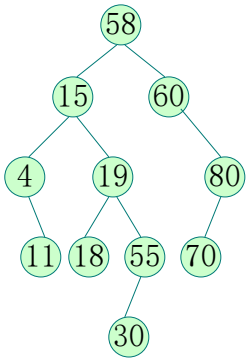
设输入序列为：30, 11, 18, 4, 55, 19, 15, 70, 58



33

课堂练习：

设输入关键字序列为:58, 60, 15, 80, 19, 55, 4, 18, 70, 11, 30, 生成二叉排序树,试画出二叉排序树；假定查找每个结点(关键字)的概率相同，计算查找成功时的平均查找长度ASL。



$$ASL=\frac{1+2+2+3+3+3+4+4+4+4+5}{11}=\frac{35}{11}\approx 3.18$$

34

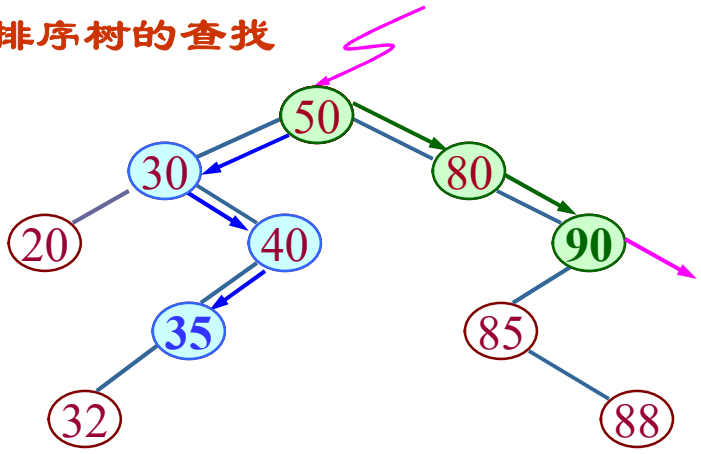
(3) 二叉排序树的存储结构

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;              // 包含key
  struct BiTNode  *lchild, *rchild;  // 左右孩子指针
} BiTNode, *BiTree;
```

35

二叉排序树的查找



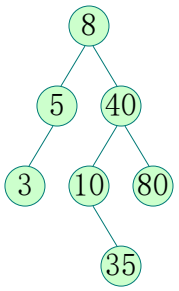
查找关键字

== 50 , 35 , 90 , 95 ,

36

(4) 二叉排序树的查找算法

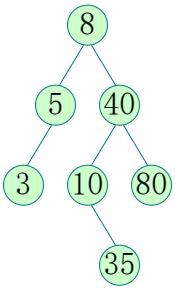
```
BiTree SearchBST (BiTree T，KeyType key){
  //在根指针T所指二叉排序树中递归地查找某关键字等于key的数据元素
  //若查找成功，则返回指向该数据元素结点的指针，否则返回空指针
    if(  (!T)||EQ(key，T->data.key)) return (T)；  //查找结束
    else if LT(key，T->data.key)
        return(SearchBST(T->lchild，key))；
                    //在左子树中继续查找
    else return(SearchBST(T->rchild，key))；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



37

非递归算法（无需栈：查找路径自根往下，不必回溯）

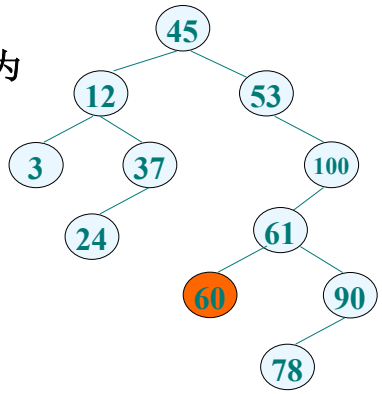
```
struct BiTNode *search_tree(struct BiTNode *T,keytype key)
//返回值  失败：NULL  成功：非NULL，结点指针
  { while (T!=NULL)
      if (key==T->data.key)
          return T;        //查找成功
      else
         if (key<T->data.key)
             T=T->lchild;  //查左子树
         else
             T=T->rchild;  //查右子树
    return T;              //查找失败
   }
```



38

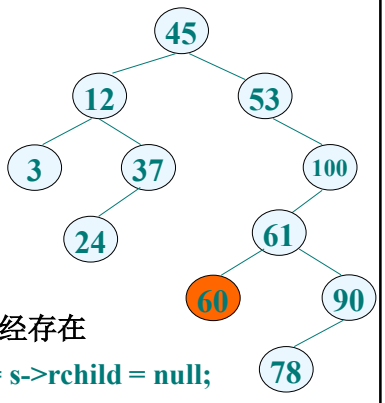
(5) 插入1个元素到二叉排序树的算法

- 如何把60插入到二叉排序树中？
- 先查找，成功则返回；
- 查找不成功，则生成新结点作为叶子插入。



39

```
Status InsertBST(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e并返回true，否则返回false
 p = T; father = NULL;
  while (p && p->data.key!=e.key) {
       father = p;
       if (e.key>p->data.key) p = p->rchild;
       else  p = p->lchild;
 }//while
 if (p) return false;  //键值为e.key的结点已经存在
  s = new BiTnode; s->data = e;   s->lchild = s->rchild = null;
 if (father==NULL)  T = s;   //空树插入
 else if (e.key>father->data.key) father->rchild = s;
       else father->lchild = s;
}//InsertBST
```



40

二叉排序树的特点

- 将一个无序序列的元素依次插入到一棵二叉排序树上，然后进行中序遍历，可得到一个有序序列。
- 在二叉排序树上插入元素时，总是将新元素作为叶子结点插入，插入过程中无须移动其他元素，仅需将一个空指针修改为非空。
- 相同记录的不同二叉排序树中序遍历的结果相同吗？
- 二叉排序树与二分查找的判定树有何区别？

41

（6）讨论：二叉排序树的删除操作如何实现？

对于二叉排序树，删除一个结点相当于删除有序序列中的一个记录，要求删除后仍需保持二叉排序树的特性。

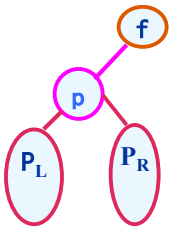
如何删除一个结点？

假设：*p表示被删结点的指针；

P_L和P_R分别表示*P的左、右孩子指针；

*f表示*p的双亲结点指针；

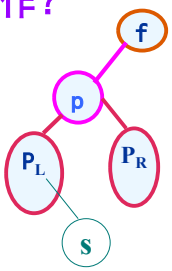
并假定*p是*f的左孩子；



则可能有三种情况：

- *p为叶子：　删除此结点时，直接修改*f指针域即可；
- *p只有一棵子树（或左或右）：令P_L或P_R为*f的左孩子即可；
- *p有两棵子树：如何处理？

难点：*p有两棵子树时，如何进行删除操作？



分析：

设删除前的中序遍历序列为：

…. P_L s p P_R f ….　　//p的直接前驱是s

//s是*p左子树最右下方的结点

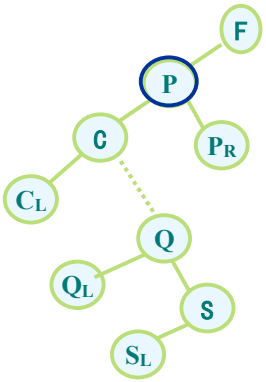
希望删除p后，其它元素的相对位置不变。有两种解决方法：

法1：令*p的左子树为 *f的左子树，*p的右子树接为*s的右子树；　//即 $f_L=P_L$； $S_R=P_R$；

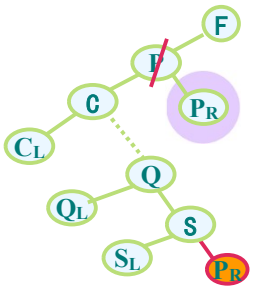
法2：直接令*s代替*p　// *s为*p左子树最右下方的结点

43

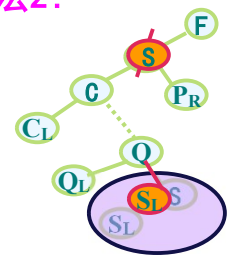
例：请从下面的二叉排序树中删除结点P。



法1：



法2：



44

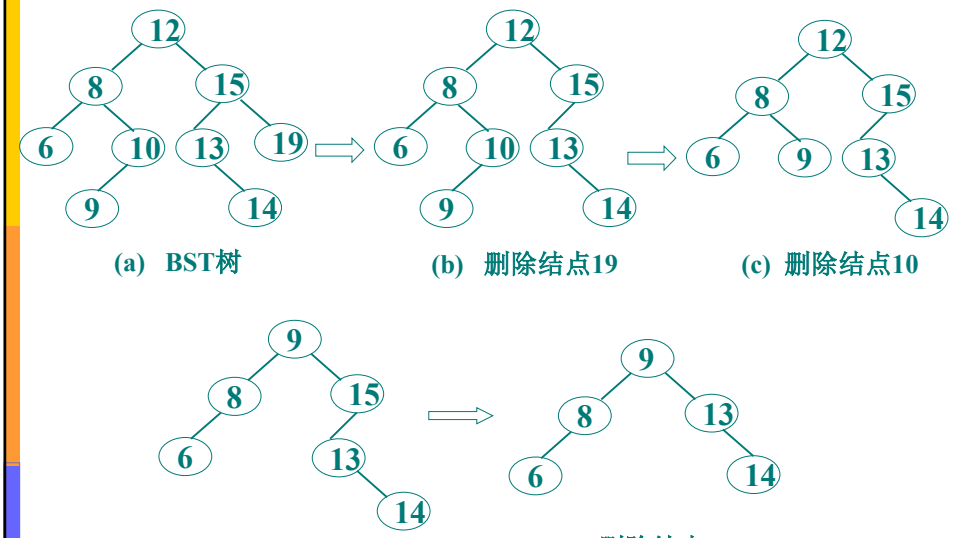
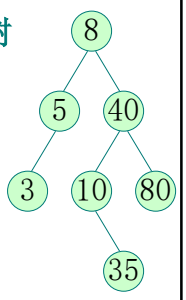
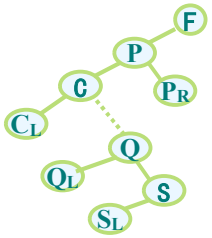


图 BST树的结点删除情况

```
void Delete_BST (BSTNode *T , KeyType  key )
  /*  在以T为根结点的BST树中删除关键字为key的结点  */
{  BSTNode *p=T , *f=NULL , *q , *s ;       //q指向删除结点的孩子
  while ( p!=NULL&&!EQ(p->key, key) )
     {  f=p ;
        if (LT(key, p->key) ) p=p->Lchild ;  // 搜索左子树
       else p=p->Rchild ;  //搜索右子树
     }
  if  (p==NULL)  return ;     //没有要删除的结点
```



```
    s=p ;   /* 找到了要删除的结点为p，先找其替代结点s */
    if (p->Lchild!=NULL&& p->Rchild!=NULL) // 左、右子树都不空
      { f=p ; s=p->Lchild ;                    // 从左子树开始找
        while (s->Rchild!=NULL)
          { f=s ; s=s->Rchild ; }   // 找左子树中最右边的结点
        p->key=s->key ; p->otherinfo=s->otherinfo ;  //用s替换p
      } //第3种情况用方案2处理
    if (s->Lchild!=NULL) q=s->Lchild ; //若s,即p只有左子树
    else q=s->Rchild ;                  //第2,3种情况归一处理
    if (f==NULL) T=q ;                  //p为根结点
    else if (f->Lchild==s) f->Lchild=q ; //p为左孩子
         else f->Rchild=q ;    //p为右孩子
    free(s) ;                  //删除p
}
```

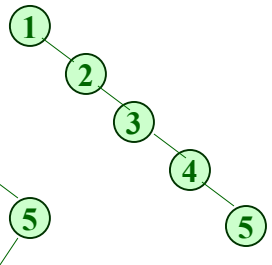


(7) 二叉排序树的查找分析

- 若查找成功，则走了一条从根结点到某结点的路径，
  若查找失败，则走到一棵空的子树时为止。
- 最坏情况下，其平均查找长度不会超过树的高度。
- 具有n个结点的二叉树的高度取决于其形态。

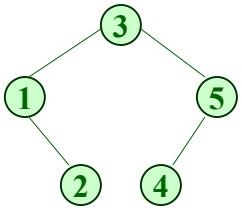
由关键字序列 1，2，3，4，5构造而得的二叉排序树，

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



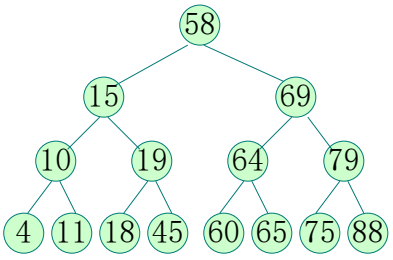
48

最好情况(为满二叉树)

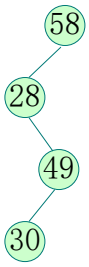
$$ASL=\frac{n+1}{n}\log_2(n+1)-1 = O(\log_2 n)$$

最坏情况(为单枝树)： ASL=(1+2+...+n)/n=(n+1)/2

平均值： $ASL\approx O(\log_2 n)$



满二叉树



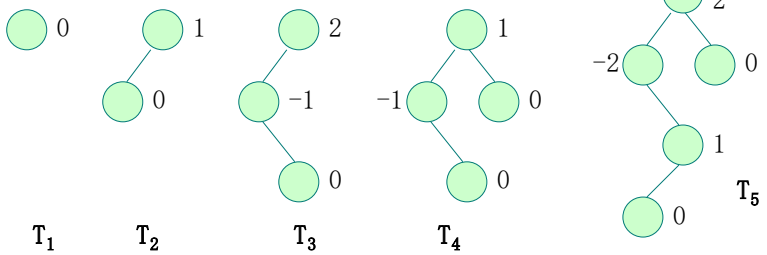
单枝树

ASL=(15+1)/15*$\log_2$(15+1)-1≈3.3　　ASL=(1+2+3+4)/4=2.5

49

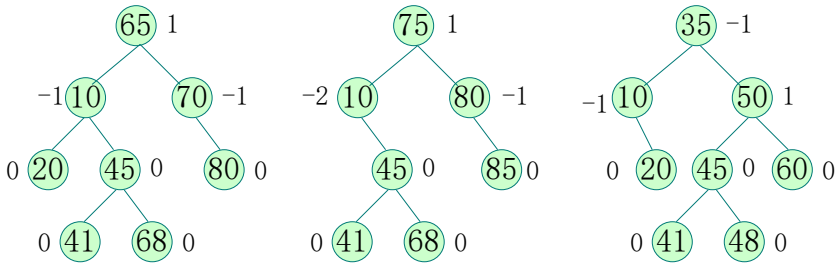
2. 平衡二叉树(高度平衡二叉树)

- AVL树：由G.M.Adelson-Velskii和E.M.Landis提出。
- 结点的平衡因子：结点的左右子树的深度之差，表示为bf。
- 平衡二叉树：任意结点平衡因子的绝对值小于等于1的二叉树。
- 递归定义：平衡二叉树或者是一棵空树，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、右子树的深度之差的绝对值不超过1。
- 平衡二叉树中结点的平衡因子为0、1或-1。



50

平衡二叉树、二叉排序树、平衡二叉排序树的区别：



平衡二叉树　　二叉排序树　　平衡二叉排序树

51

结点类型定义如下：

```
typedef  struct  BNode
  {  KeyType  key ;    /*  关键字域  */
     int  Bfactor ;    /*  平衡因子域  */
     …      /*  其它数据域  */
     struct  BNode  *Lchild , *Rchild ;
  }BSTNode ;
```

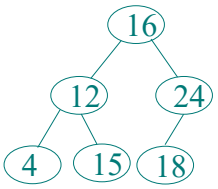


图 平衡二叉树

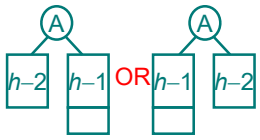
平衡二叉排序树的查找：

与二叉排序树的查找相似，与给定的K值比较的次数不超过AVL树的深度。

设深度为h的平衡二叉排序树所具有的最少结点数为N_h，则由平衡二叉排序树的性质知：

$N_0=0$，$N_1=1$，$N_2=2$，…，

$N_h= N_{h-1}+N_{h-2}+1$



该关系和Fibonacci数列相似。

由归纳法可证，当h≥0时，$N_h=F_{h+2}-1$，…，而

$$F_h\approx\frac{\Phi^h}{\sqrt{5}}\ \text{其中}\ \Phi=\frac{1+\sqrt{5}}{2}\qquad \text{则}N_h\approx\frac{\Phi^{h+2}}{\sqrt{5}}-1$$

含有n个结点的平衡二叉排序树的最大深度为

$$h\approx\log_{\Phi}(\sqrt{5}\times(n+1))-2$$

则平衡二叉排序树上查找的平均查找长度O(log *n*)。

如果在一棵AVL树中插入一个新结点，就有可能造成失衡，此时须重新调整树的结构，使之恢复平衡。我们称调整平衡过程为平衡旋转。

只需对最小不平衡子树进行旋转处理

(重点处理离插入点最近的失衡结点)。

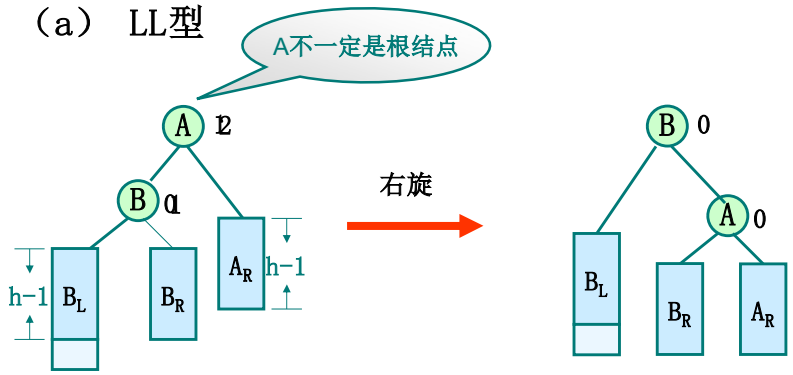
平衡旋转可以归纳为四类：

- LL平衡旋转
- RR平衡旋转
- LR平衡旋转
- RL平衡旋转

54

平衡二叉排序树的平衡旋转方法：

（a） LL型

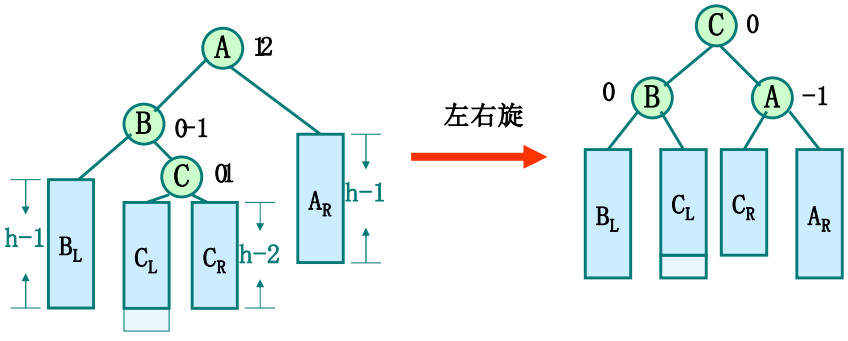


保持中序遍历次序不变：B_L，B，B_R，A，A_R

55

平衡二叉排序树的平衡旋转方法：

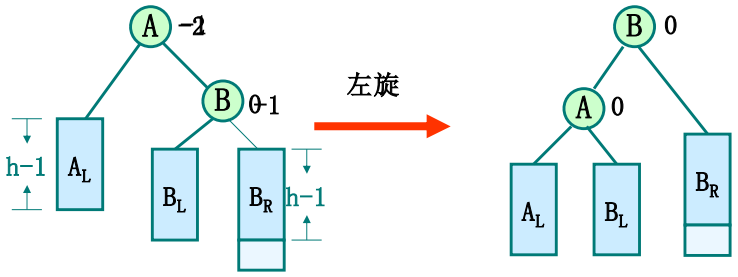
（b） LR型



56

平衡二叉排序树的平衡旋转方法：

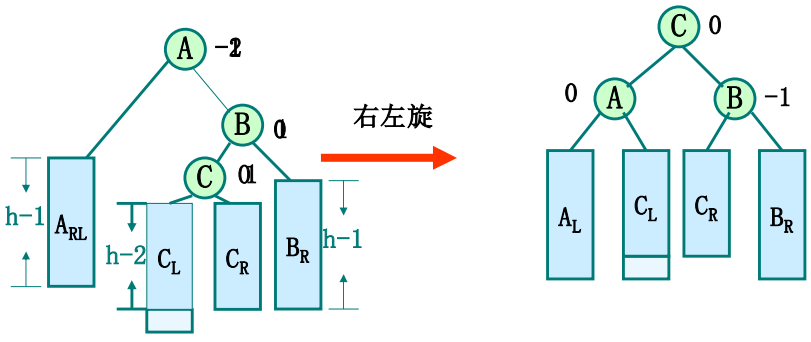
（c） RR型



57

平衡二叉排序树的平衡旋转方法：

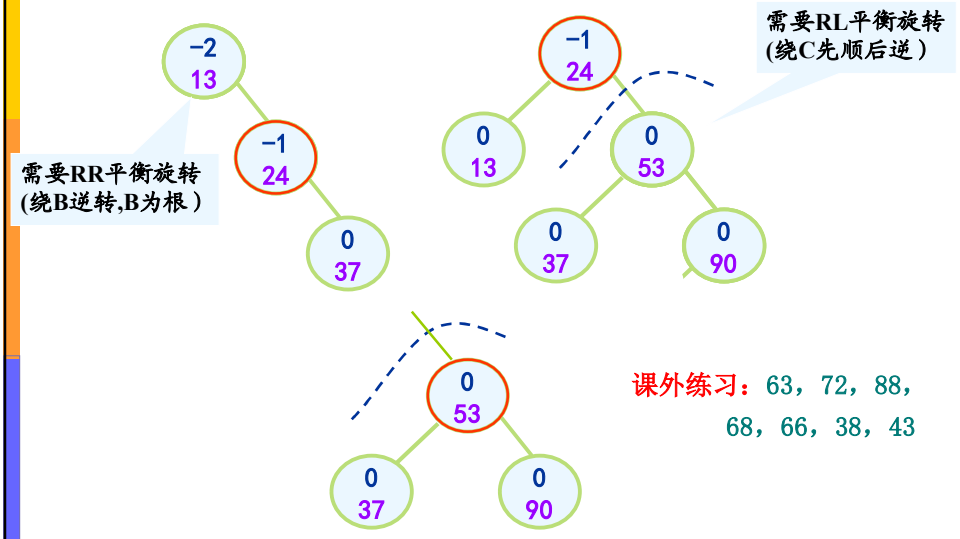
（d） RL型



58

例：请将下面序列构成一棵平衡二叉排序树：

（13，24，37，90，53）



课外练习：63，72，88，68，66，38，43

59

9.2.2 B-树和B+树

- 前面的查找称为内部查找，适合规模比较小的文件。
- 若以平衡二叉树作为磁盘文件索引组织时，如以结点为内、外存交换单位，则在索引文件中查找需要的关键字，存取与查找索引平均对磁盘需要进行$\log_2 n$次访问。
- 如何降低磁盘访问的代价？

  索引（如 B-树）文件一般在磁盘，根结点可驻内存。
- **B-树**：B-树是一种平衡的多路查找树，对于相同大小的文件，B-树的深度比AVL树的深度小，存取与查找**B-树**索引时磁盘访问次数降低 。

60

B-树的概念

❑ 定义：一棵m阶的B-树，或为空树，或满足下列特性

(1) 树中每个结点至多有m棵子树；

(2) 若根结点不是叶子结点，则至少有两棵子树；

(3) 除根之外的所有非终端结点至少有ceil(m/2)棵子树；

(4) 所有的非终端结点中包含下列信息数据

$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

其中：K_i为关键字，且$K_i<K_{i+1}$；A_i为指向子树根结点的指针，且指针A_{i-1}所指子树中所有结点的关键字均小于K_i，A_i所指子树中所有结点的关键字均大于K_i，$(i=1,2,\cdots,n)$

$\lceil m/2\rceil-1\leqslant n\leqslant m-1$.

(5) 所有的叶结点都出现在同一层上，且不带信息。

(可看作是外部结点或查找失败的结点，实际上这些结点不存在)

61

For a non-empty B-tree of order m:

This may be zero, if the node is a leaf as well

	Root node	Non-root node
Minimum number of keys	1	$\lceil m/2\rceil - 1$
Minimum number of non-empty subtrees	2	$\lceil m/2\rceil$
Maximum number of keys	m - 1	m – 1
Maximum number of non-empty subtrees	m	m

These will be zero if the node is a leaf as well

例如图所示为一棵4阶的B-树，其深度为4。

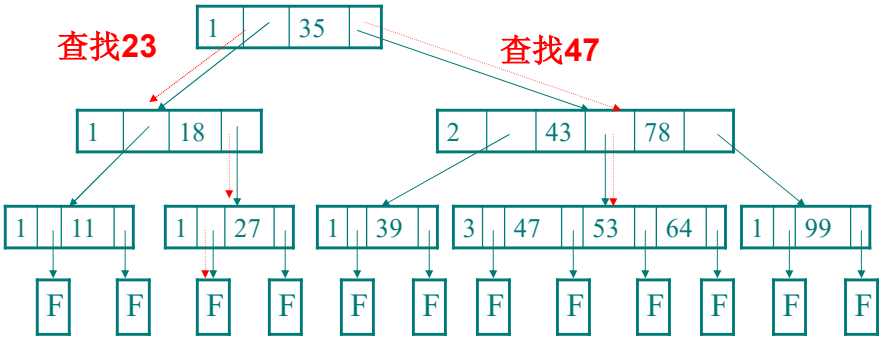


图9.14　一棵4阶的B-树

由此可见，在B-树上进行查找的过程是一个顺指针查找结点和在结点中查找关键字交叉进行的过程。

63

Comparing B-Trees with AVL Trees

- The height h of a B-tree of order m, with a total of n keys, satisfies the inequality:

$$h <= 1 + \log_{\lceil m/2 \rceil} ((n + 1) / 2)$$

- If m = 300 and n = 16,000,000 then $h \approx 4$.
- Thus, in the worst case finding a key in such a B-tree requires 3 disk accesses (assuming the root node is always in main memory).
- The average number of comparisons for an AVL tree with n keys is $\log_2 n + 0.25$ where n is large.
- If n = 16,000,000, the average number of comparisons is 24.
- Thus, in the average case, finding a key in such an AVL tree requires 24 disk accesses.

B-树结构的C语言描述如下：

```
typedef struct BTNode {
  int  keynum;        // 结点中关键字个数，结点大小
  struct BTNode  *parent;
                      // 指向双亲结点的指针
  KeyType  key[m+1];  // 关键字（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
```

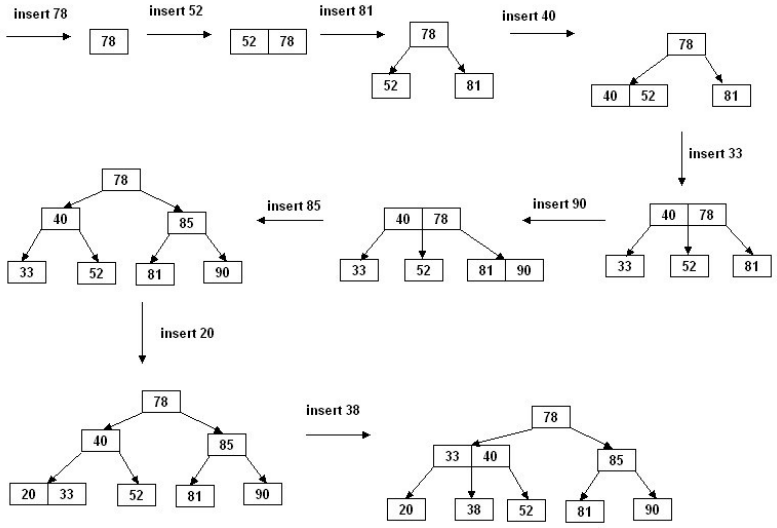
65

Insertion in B-Trees

- **Overflow Condition:**
  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.
- **Insertion algorithm:**
  If a node overflows, split it into two, propagate the "middle" key to the parent of the node.
  If the parent overflows the process propagates upward.
  If the node has no parent, create a new root node.
- **Note:** Insertion of a key always starts at a leaf node.

Insertion in B-Trees

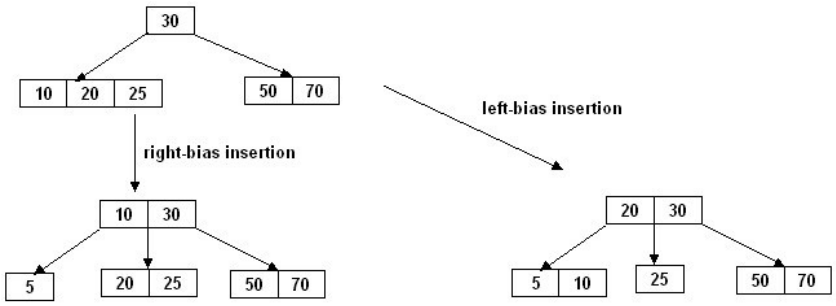
- **Insertion in a B-tree of odd order**
- **Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3**



- **Insertion in a B-tree of even order**

At each node the insertion can be done in 2 different ways:

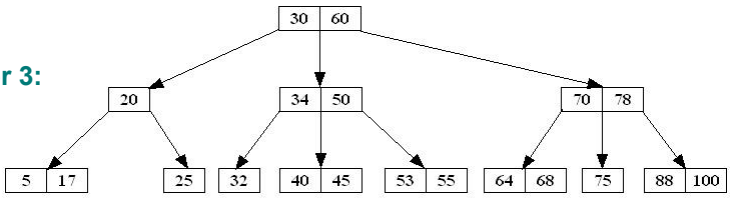
- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.
- **Example**: Insert the key **5** in the following B-tree of order **4**:



Deletion in B-Tree

- Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).
- The successor of a key **k** is the smallest key greater than **k**.
- The predecessor of a key **k** is the largest key smaller than **k**.

Example:
B-tree of order 3:



key	predecessor	successor
20	17	25
30	25	32
34	32	40
50	45	53
60	55	64
70	68	75
78	75	88

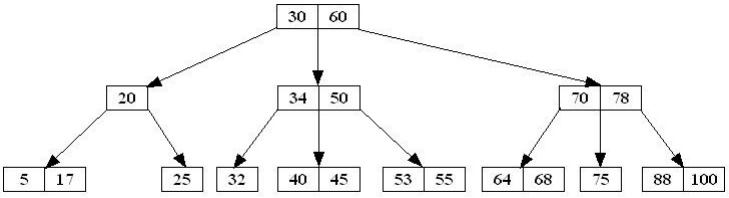
Deletion in B-Tree

- **Underflow Condition:**

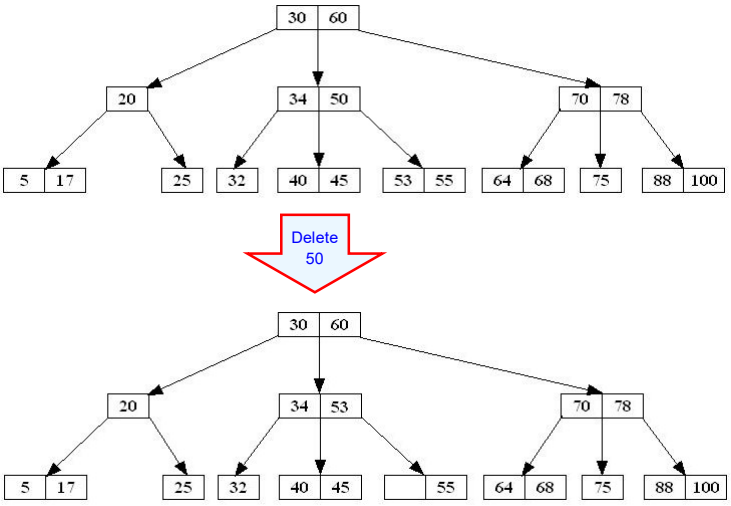
  A non-root node of a B-tree of order m underflows if, after a key deletion, it contains $\lceil m / 2 \rceil - 2$ keys

- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.
- **Deletion algorithm:**

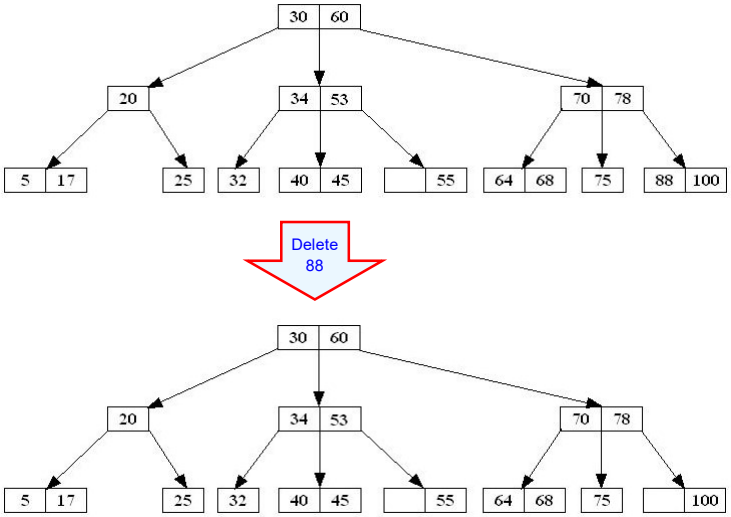
  If a node underflows, **rotate** the appropriate key from the **adjacent** right- or left-sibling if the sibling contains at least $\lceil \mathbf{m / 2} \rceil$ keys; otherwise perform a **merging**.



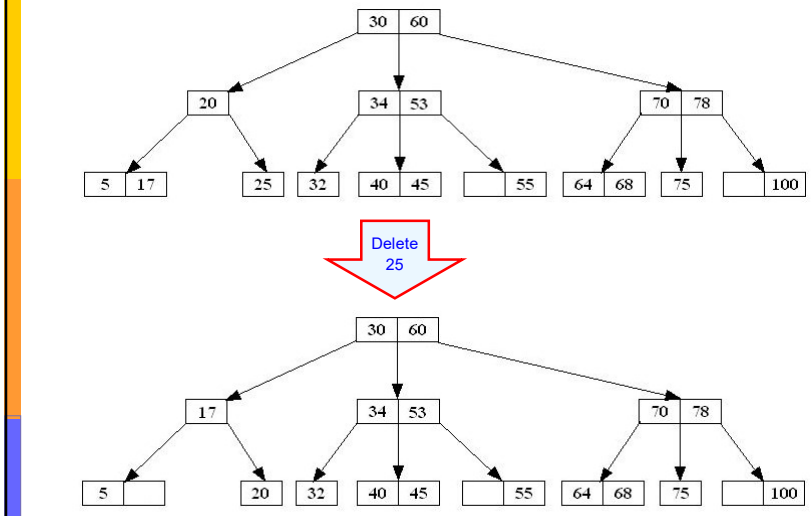
Deletion in B-Tree

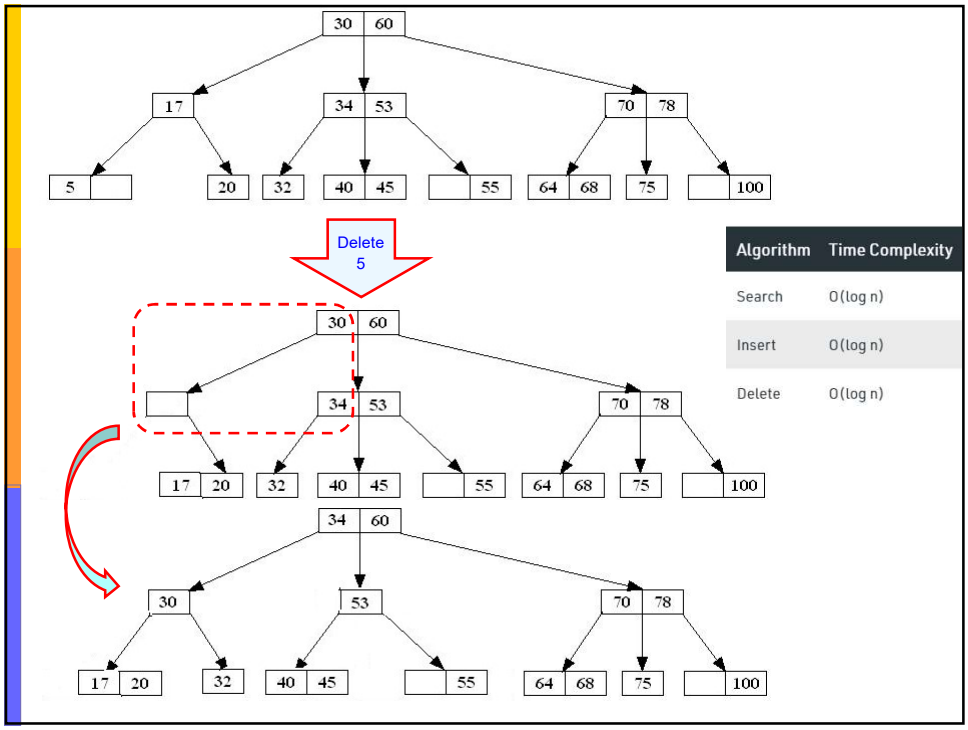


Deletion in B-Tree



Deletion in B-Tree



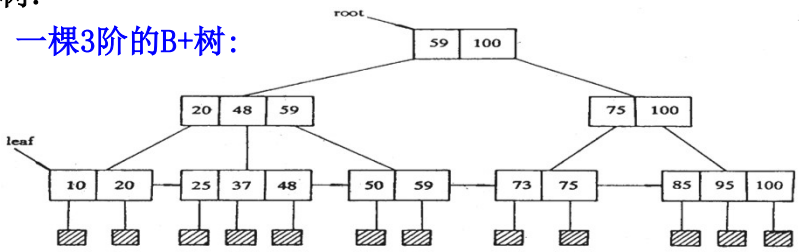


Algorithm	Time Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

B+树

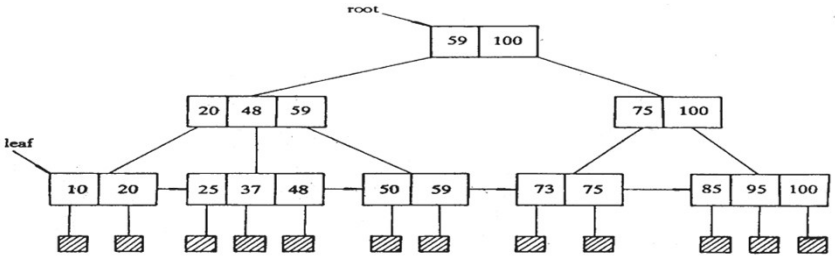
考虑B-树的一种变形，该B-树中所有的信息仅存于B树的叶页(结点)上，而非叶页(结点)仅含有便于查找的辅助信息，如本子树中结点关键码最小（或最大）值，这样的B-树被称B+树.

例 一棵3阶的B+树:



- 通常在B+树上有两个头指针，一个指向根页，另一个指向关键码最小的叶页。
- 因此可以对树进行两种查找运算。一种是从最小关键码起顺序查找，另一种是从根结点开始进行查找。

m阶B+树与B-树的区别



- 结点中关键字个数等于子树棵数，关键字与子树对应关系不同
- 所有叶子结点含查找表全部关键字及指向对应记录指针，内部结点作为索引只含关键字信息。
- 叶子结点构成有序链表，可以进行顺序查找。
- B+树进行随机查找，从根查到含关键字结点时需继续搜至叶子。

静态和动态查找表查找方法小结

静态查找表和动态查找表通过比较关键字进行查找。

顺序表，对数据元素的存储一般有两种形式：

(a) 按到达次序连续存放，查找时顺序比较查找；

(b) 按关键字的相对关系整理后以递增或递减形式连续存放，则查找时使用顺序法或二分法比较查找。

二叉排序树，从根开始进行比较查找。

不足：查找时无法根据关键字的值估计数据元素可能的位置。

77

9.3 哈希表

- 前面介绍的所有查找都是基于待查关键字与表中元素进行比较而实现的查找方法，查找的效率依赖于查找过程中所进行的比较次数。
- 能否不经过任何比较，一次便能得到所查记录？

——hash表

一、哈希表的概念

二、哈希函数的构造方法

三、冲突处理方法

四、哈希表的查找及分析

78

- **基本思想：** 存储数据元素时，用一个Hash函数根据数据元素的关键字计算出该数据元素的存储位置。

  Hash：关键字→ 地址

- **查找：** 根据给定的数据元素的关键字计算出该数据元素可能存储位置。
- 查找速度快（O(1)），查找效率与元素个数n无关！
- 哈希表亦称为杂凑表，散列表。
  - ✓ 其数据元素的存储一般不连续。
  - ✓ 可用于表示sets, tables等抽象数据类型
- 通过Hash函数计算出的地址称为哈希地址或散列地址。

79

例： 有数据元素序列(14，23，39，9，25，11)，若规定每个元素k的存储地址H（k）＝k，请画出存储结构图。

解：根据散列函数H（k）＝k ，可知元素14应当存入地址为14的单元，元素23应当存入地址为23的单元，……，

对应散列存储表（哈希表）如下：

地址	…	9	…	11	…	14	…	23	24	25	…	39	…
内容		9		11		14		23		25		39	

查找： key=9

访问H(9)=9号地址，若内容为9则成功；

若查不到，返回一个特殊值，例如空指针或空记录。

缺点：空间效率低！

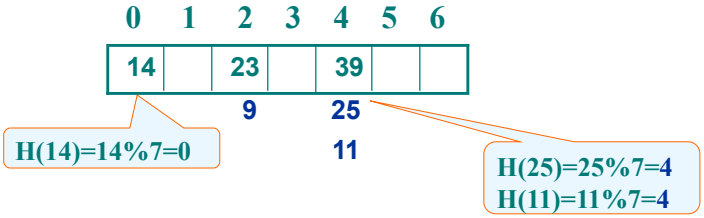
80

有6个元素的关键码分别为：（14，23，39，9，25，11）。

选取关键字与元素位置间的函数为H(k)=k mod 7

6个元素用7个地址应该足够!

通过哈希函数对6个元素建立哈希表：



有冲突！

81

散列技术中的主要问题

(1)冲突

Hash函数是将一组关键字映象到一个有限的地址区间上。

一般情况下，设计出的散列函数很难是单射的，即不同的关键字对应到同一个存储位置，这样就造成了冲突(碰撞)。

此时，发生冲突的关键字互为同义词。

(2)散列函数的设计目标

设计一个简单、均匀、存储空间利用率高、冲突少的散列函数是关键。

82

哈希表的定义：

根据设定的哈希函数 H(key) 和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集（区间）上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

9.3.2 构造哈希函数的方法与相应的冲突解决技术

要求一：空间利用率高

要求二：

尽量均匀地存放元素，以避免冲突。

1. 直接定址法
2. 除留余数法
3. 乘余取整法
4. 数字分析法
5. 平方取中法
6. 折叠法
7. 随机数法

1. 直接定址法

取关键字或关键字的某个线性函数值为哈希地址

H(key)=key

H(key)=a.key+b

例1 人口统计表

序号(地址)	年 龄	人 数(万)
1	1	10.5
2	2	12.6
3	3	11.0
4	4	20.8
	...	...
150	150	...

key

H(key)=key = 地址

H(年龄) = 年龄

85

例2 学生成绩表

序号(地址)	学 号	姓 名	性别	数学	外语
1	200041	刘大海	男	80	75
2	200042	王 伟	男	90	83
3	200043	吴晓英	女	82	88
4	200044	王 伟	女	80	90
	.....	.....	...	..	..
n	.....	.....	...	..	..

key

H(key) = key - 200040 = 地址

H(学号)= 学号- 200040

86

例3 标识符表

序号	标识符(key)
1	ABC
2	
3	CAD
4	DAT
5	
6	FOX
25	YAB
26	ZOO

H(key)=key的第一个字母在字母表中的序号

key =ABC CAD DAT FOX YAB ZOO

H(key)=1 3 4 6 25 26

直接定址法分析

优点：以关键字key的某个线性函数值为哈希地址，不会产生冲突.

缺点：要占用连续地址空间，空间效率低。

87

2. 除留余数法

思想：设哈希表HT[0..m-1]的表长为m，

哈希地址为key除以p所得的余数

H(key)=key % p //C语言

其中：%为“取模”或“求余”

p<=m ，p为接近m的质数(素数)，如：

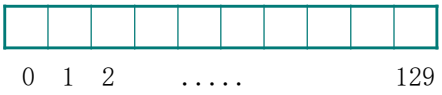
3,5,7,11,13,17,19,23,29,31,37.....

或不包含小于20的质因子的合数，如：

713(=23*31)

88

例1　设 m=130, 取p=127,
H(key)=key % 127



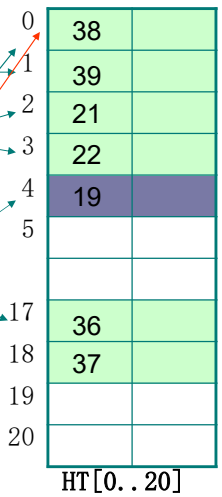
例2　设 m=256　取 p=251
H(key)=key % 251

89

例 设哈希表的地址范围为0～20,哈希函数为
H(Key)=Key % 19

输入关键字序列：39,22,21,37,36,38,19,解决冲突的方法为线性探测再散列(哈希),构造哈希表HT[0..20]。

关键字K	H(K)=K%19
39	39%19=1
22	22%19=3
21	21%19=2
37	37%19=18
36	36%19=17
38	38%19=0
19	19%19=0



19与38冲突,再与39,21,22冲突,存入HT[4]

$H_i = (H(key)+d_i)$ MOD m　i=1,2,…K(K<=m-1)

$d_i = i$,其中i=1,2,…K

90

$H_i = (H(key)+d_i)$ MOD m　i=1,2,…K(K<=m-1)

再输入17, 56, 55

17%19=17

17与36冲突, 再与37冲突, 存入HT[19]。

56%19=18

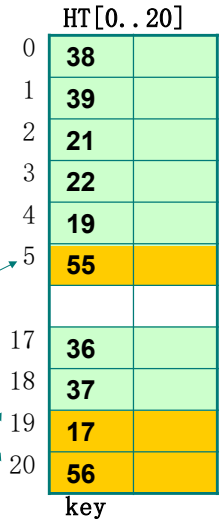
56与37冲突, 再与17冲突, 存入HT[20]。

55%19=17

55与36冲突, 再与37, 17, 56冲突, 再与38, 39, 21, 22, 19冲突, 存入HT[5]。

对于H(k)=k % 19，所有的19a+b为同义词, 0≤b≤19

如：5, 24, 43, 62, 81, ...



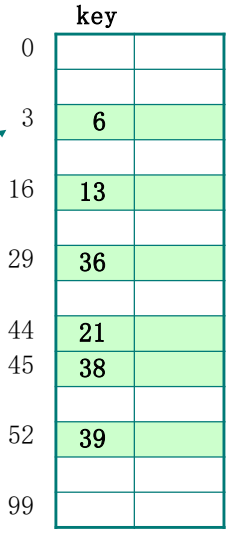
91

3. 平方取中法：取关键字平方后的中间某几位为哈希地址, 即：

H(k)=取k^2的中间某几位数字

例. 设哈希表为HT[0..99], 哈希函数为：H(K)=取k^2的中间2位数, 输入关键字序列：39, 21, 6, 36, 38, 13, 用线性探测再散列法解决冲突, 构造HT[0..99]。

K	k^2	H(K)
39	1521	52
21	0441	44
6	0036	03
36	1296	29
38	1444	44
13	0169	16



92

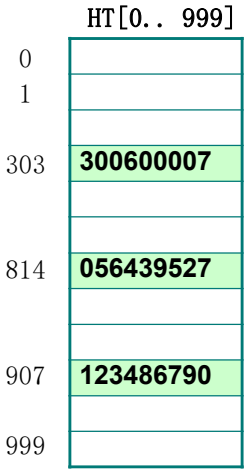
4.折叠法

将关键字分割成位数相同的几部分,然后取这几部分的叠加和作为哈希地址。

(1)边界折叠法（沿边界来回折叠）

设表地址范围为0～999

- k1=056439527
  650 + 439 + 725 =1814
  H(k1)=814
- k2=123486790
  321 + 486 + 097 =907
  H(k2)=907
- k3=300600007
  003 + 600 + 700 =1303
  H(k2)=303

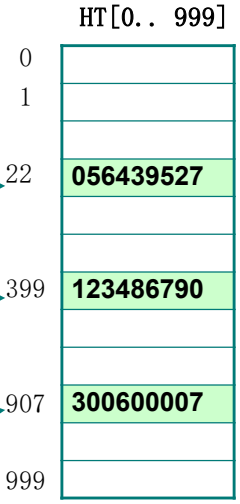


93

(2)移位折叠法(最低位对齐)

设表地址范围为0～999

- k1=056439527
  056 + 439 + 527 =1022
  H(k1)=022
- k2=123486790
  123 + 486 + 790 =1399
  H(k2)=399
- k3=300600007
  300 + 600 + 007 =907
  H(k2)=907

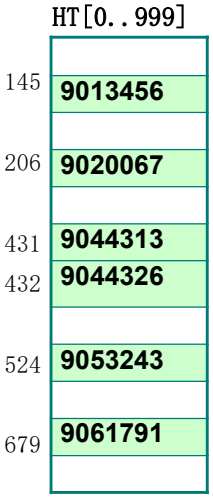


94

5. 数字分析法

设哈希表中可能出现的关键字都是事先知道的，则可取关键字的若干分布均匀的位组成哈希地址。

k	H(k)
9020067	206
9044313	431
9013456	145
9044326	432
9053243	524
9061791	679



95

6. 随机数法

H(key)=random(key)

random(key)为产生伪随机数的函数

7. 灵活构造哈希函数

例. 设哈希表为HT[0..40]，哈希函数为：

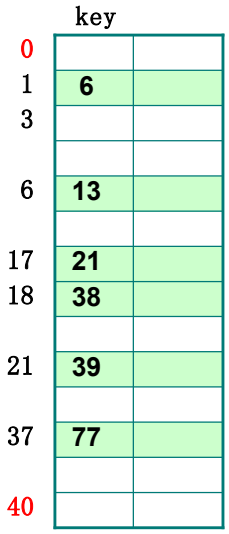
H(K)=取k^2的中间2位数*40/99

其中40/99将其00～99压缩到00～40之内，

输入关键字序列：39, 21, 6, 36, 38, 13,

用线性探测再散列法解决冲突。

K	k^2	H(K)
39	1521	52*40/99=21
21	0441	44*40/99=17
6	0036	03*40/99=1
77	5929	92*40/99=37
38	1444	44*40/99=17
13	0169	16*40/99=6



96

小结：构造哈希函数的考虑的因素：

① 执行速度（即计算哈希函数所需时间）；
② 关键字的长度；
③ 哈希表的大小；
④ 关键字的分布情况；
⑤ 查找频率。

97

9.3.3 如何解决冲突

1. 开放定址法（开地址法）
2. 链地址法（拉链法）
3. 再哈希法（双哈希函数法）
4. 建立一个公共溢出区

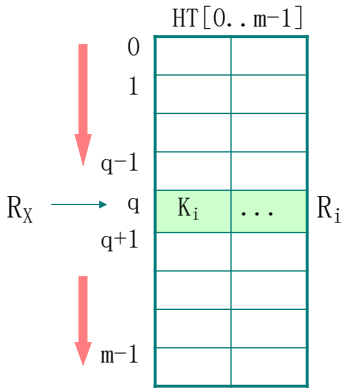
98

1. 开放地址法(开式寻址法)

假定记录R_i, R_X的关键字K_i, K_X为同义词，散列地址都为q，R_i已存入HT[0..m-1]中的HT[q]中，则R_X 存入HT中的某个空位上。依次在地址：

q+1, q+2, . . . , m-1, 0, 1, . . . , q-1

中寻找一个空位，叫做线性探测再散列。



(1) 线性探测再散列

$H_i=(H(key)+d_i) \bmod m$　　($1 \le i < m$)

其中：H (key)为哈希函数

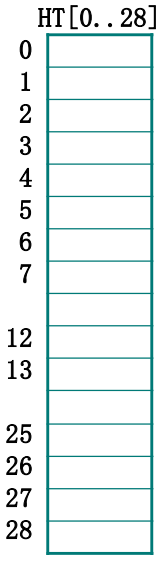
m为哈希表长度

d_i 为增量序列 1，2，…m-1，且$d_i=i$

99

练习：设 H(k)=k的首字母在字母表中的序号,用线性探测再散列法解决冲突，依次用下列关键字，造哈希表HT[0..28]。

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4



100

练习：设 H(k)=k的首字母在字母表中的序号,用线性探测再散列法解决冲突，依次用下列关键字，造哈希表HT[0..28]。

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4

HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
25	YY
26	ZMN
27	ZE
28	ZOO

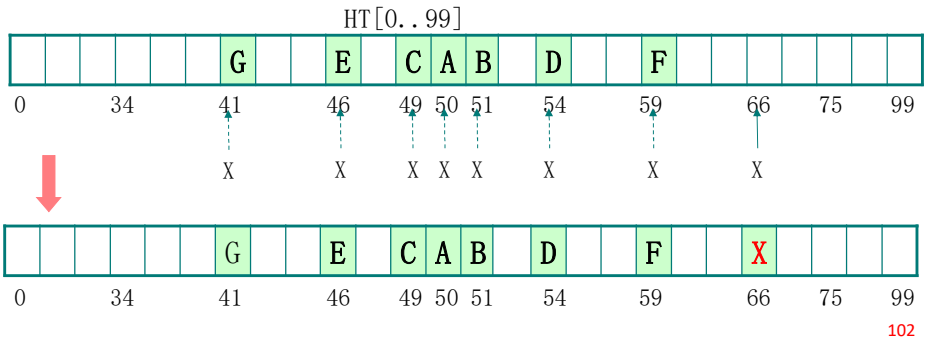
101

(2) 二次探测再散列

假定记录R_i和R_j的关键字K_i和K_j为同义词，散列地址为q，R_i已存入HT[0..m-1]中的HT[q]中。若依次在地址

$q+1^2, q-1^2, q+2^2, q-2^2, \ldots, q+i^2, q-i^2, \ldots$ 中寻找一个空位，叫做二次探测再散列。

例：设记录X和A为同义词，散列地址为50，二次探测再散列的地址序列为：51, 49, 54, 46, 59, 41, 66, 34, 75, ...

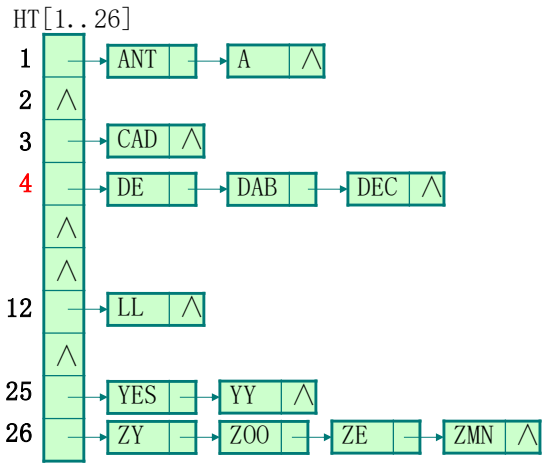


102

2. 链地址法：将关键字为同义词的所有记录存入同一链表中。

例 设H(k)=k的首字母在字母表中的序号，用下列关键字造哈希表 HT[1..26]

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4



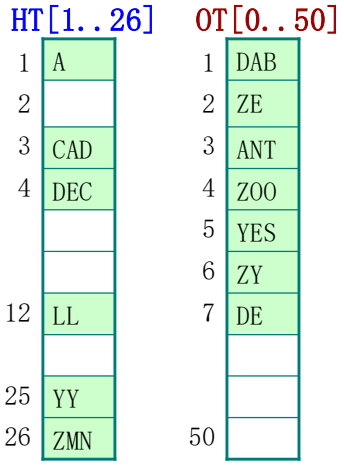
103

3. 建立公共溢出区

将发生冲突的所有记录存入一个公共溢出表OT[0..v]中。

例 设H(k)=k的首字母在字母表中的序号，用下列关键字生成基本表HT[1..26]和溢出表OT[0..50]

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4



104

4. 再哈希法

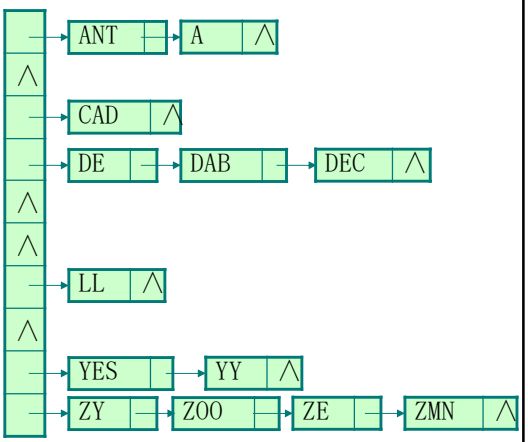
发生冲突时，使用下一个哈希函数计算哈希地址：

j1=H1(K)； j2=H2(K)； j3=H3(K)；.....

105

9.3.4 哈希表的查找及其分析

例1 (链地址法)假定每个记录的查找概率相等，查找成功时的平均查找长度：



ASL=(1+2+1+1+2+3+1+1+2+1+2+3+4)/13

= 24/13

≈1.85

106

例2（线性探测再散列）假定每个记录的查找概率相等，查找成功时的平均查找长度.

关键字K	H(K)	比较次数
YES	25	5
A	1	1
ANT	1	2
CAD	3	1
DEC	4	1
DAB	4	2
ZY	26	10
DE	4	4
LL	12	1
YY	25	1
ZMN	26	1
ZE	26	2
ZOO	26	3
	合计	34

ASL= 34/13

≈2.62

HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
13	
25	YY
26	ZMN
27	ZE
28	ZOO

107

讨论：如何估计查找失败时的平均查找长度？

- 查找成功：查找表中的记录（目标明确），假定每个记录的查找机会均等。
- 查找失败：查找不在表中的记录（目标记录与个数不明确），但查找地址数目明确。
  - ✓ Hash函数可能地址：0，1，…,p-1;机会均等，易于统计；
  - ✓ Hash表可能地址：0，1，…,m-1; 机会不均等，不便统计。

108

例2(续) (线性探测再散列)查找不成功时的平均查找长度.需统计不成功时比较次数

H(K)	比较次数	H(K)	比较次数
1	7	14	0
2	6	15	0
3	5	16	0
4	4	17	0
5	3	18	0
6	2	19	0
7	1	20	0
8	0	21	0
9	0	22	0
10	0	23	0
11	0	24	0
12	1	25	12
13	0	26	11

ASL=52/26=2 （只计入关键字比较次数）

HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
13	
25	YY
26	ZMN
27	ZE
28	ZOO

109

一般情况：平均查找长度依赖于哈希表的装填因子：

α=(表中填入记录数)/(哈希表的长度)

解决冲突的方法	平均查找长度	
	查找成功	查找失败
线性探测再散列	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}[1+\frac{1}{(1-\alpha)^2}]$
二次探测再散列	$\frac{\ln(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$
链地址	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

110

讨论：如何在哈希表中删除一个记录？

对于非链式地址冲突处理的哈希表，删除一个记录时需填充一个特殊符号，以便能找到后续的“同义词”。

对填充删除标记之后的HASH表， 其查找与插入操作算法需要作相应调整。

111

例9.4 给定关键字序列11，78，10，1，3，2，4，21，试分别用顺序查找、二分查找、二叉排序树查找、散列查找(用线性探查法和拉链法)来实现查找，试画出它们的对应存储形式(顺序查找的顺序表，二分查找的判定树，二叉排序树查找的二叉排序树及两种散列查找的散列表)，并求出每一种查找的成功平均查找长度。散列函数H(k)=k%11。

顺序查找的顺序表（一维数组）如图所示，

0	1	2	3	4	5	6	7	8	9	10
11	78	10	1	3	2	4	21			

图 顺序存储的顺序表

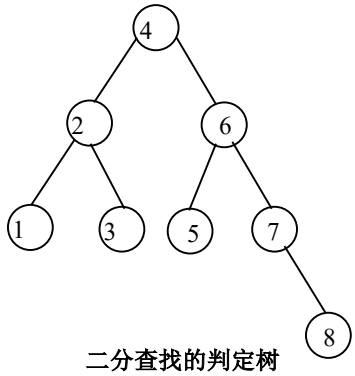
从图可以得到顺序查找的成功平均查找长度为:

ASL=(1+2+3+4+5+6+7+8)/8=4.5;

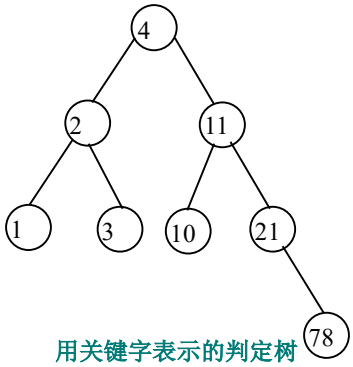
112

11，78，10，1，3，2，4，21

二分查找基于有序序列：1，2，3，4，10，11，21，78，
其判定树如图示，（中序序列为从小到大排列的）



二分查找的判定树



用关键字表示的判定树

从图可以得到二分查找的成功平均查找长度为：

ASL=(1+2*2+3*4+4)/8=2.625;

113

11，78，10，1，3，2，4，21

二叉排序树如图所示：

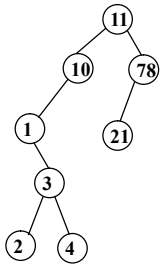


图 二叉排序树

从图可以得到二叉排序树查找的成功平均查找长度为：

ASL=(1+2*2+3*2+4+5*2)/8=3.125;

114

H(k)=k%11　　11，78，10，1，3，2，4，21

H(11)=11%11=0；　　H(3)=3%11=3；
H(78)=78%11=1；　　H(2)=2%11=2，3，4；
H(10)=10%11=10；　　H(4)=4%11=4，5；
H(1)=1%11=1，2；　　H(21)=21%11=10，0，1，2，3，4，5，6

线性探查法解决冲突的散列表如图所示，

0	1	2	3	4	5	6	7	8	9	10
11	78	1	3	2	4	21				10

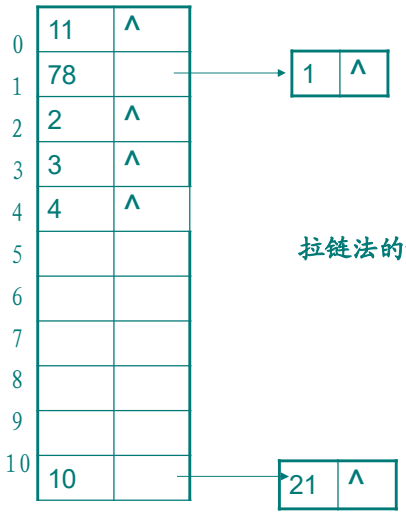
图 线性探查的散列表

从图可以得到线性探查法的成功平均查找长度为：

ASL=（1*4+2*2+3+8）/8=2.375；

115

拉链法解决冲突的散列表如图所示。



拉链法的散列表图

从图可以得到拉链法的成功平均查找长度为：

ASL=(1*6+2*2)/8=1.25。

116

例9.5 给定关键字序列7，8，30，11，18，9，14存储到散列表中，散列存储空间为下标为从0开始的一维数组，散列函数H(key)=(key*3)MOD7,冲突采用线性探测再散列法，装填因子为0.7。试画出所构造的散列表；分别计算在等概率情况下查找成功与查找不成功的平均查找长度。

构造的散列表为：

0	1	2	3	4	5	6	7	8	9
7	14		8		11	30	18	9	

H(7)=(7*3)MOD7=0　H(8)=(8*3)MOD7=3　H(30)=(30*3)MOD7=6

H(11)=(11*3)MOD7=5　　H(18)=(18*3)MOD7=5，→ 6，→ 7

H(9)=(9*3)MOD7=6，→ 7，→ 8

H(14)=(14*3)MOD7=0，→ 1

$ASL_{成功}$ =(1次×4+2次×1+3次×2)/7=12/7;

$ASL_{不成功}$ =(2+1+0+1+0+4+3)/7=11/7

117

例10

二叉树结点的结构如下：

```
typedef struct{
    ElemType Sqlistdata[MaxSize];
    int num;
}Sqlist;
```

用数组保存二叉树，每个结点保存正整数，空节点的值为-1，设计一个高效算法，判断二叉树是否为二叉搜索树。

（1） 给出算法的基本设计思想

（2） 根据设计思想采用C或者C++语言描述算法，关键之处给出注释

答案：(1)判断二叉搜索树，只需要判断该树的中序遍历是否是升序序列即可，可采用递归方式中序遍历。(注：1.根结点在数组0下标处，所以i结点的左儿子是2*i+1，右儿子是2*i+2

118

例10

（2）解法一，中序递归（推荐）：

```
bool T=true;
int k=0;
void inorder(Sqlist a,int i)
{
if (i>a.num || a.Sqlistdata[i]=-1 || !T) return;
                                        //左子树不存在或已经确定不是二叉搜索树了
inorder(a,i*2+1);                                    //访问左子树
if (k>a.Sqlistdata[i*2+1]) {T=false; return;}
k=a.Sqlistdata[i*2+1];
inorder(a,i*2+2);                                    //访问右子树
}
bool ans(Sqlist a)
{
inorder(a,0);
return T;
}
```

119

本章小结

- 静态查找表，顺序查找、折半查找和平均查找长度
- 动态查找表
  - 二叉排序树的查找、插入和删除
  - 平衡二叉树的概念，平衡处理的基本方法
  - B树的基本概念及其查找过程
- 散列（哈希）查找
  - 哈希造表，hash函数
  - 冲突及消解
  - 装填因子与查找效率

120

课外思考：严9.19　9.31

9.25(求查找成功时的平均查找长度）

121