查找

▼ 基本概念

查找表：同一类型数据元素构成的集合

操作：

1. 查询是否在表中
2. 查询属性
3. 插入
4. 删除

静态查找表：进行1.2.

动态查找表：进行1.2.3.4.

关键字：标识一个元素

主关键字：唯一识别一个元素

次关键字：识别若干元素

平均查找长度

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

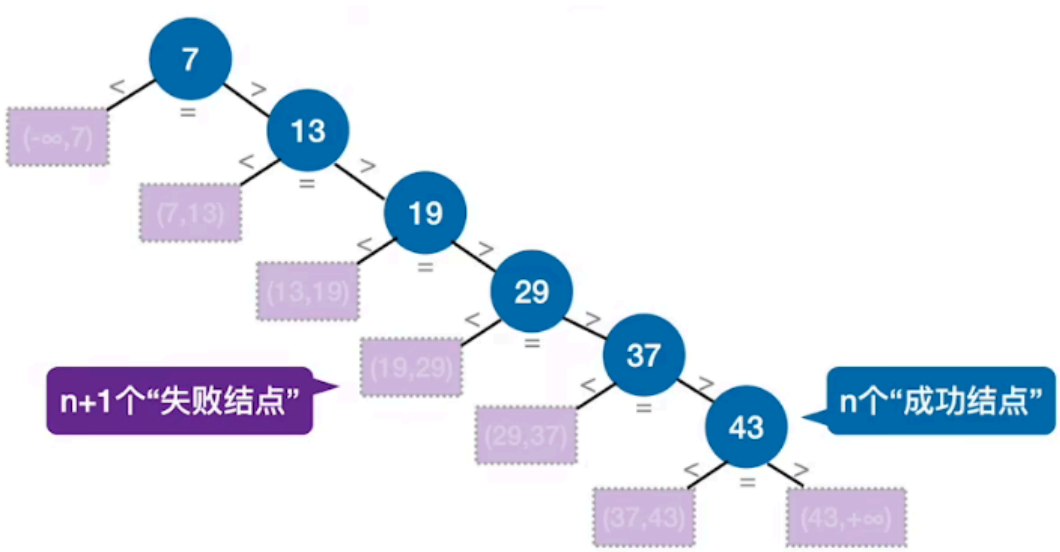
▼ 静态查找表

▼ 顺序表查找

```
typedef struct{
    ElemType *elem;
    int TableLen;
}SSTable;

//不加哨兵，从前往后
int Search_Seq(SSTable ST,ElemType key){
    int i;
    for(i=0;i<ST.TableLen && ST.elem[i]!=key;i++);
```

```
        return i==ST.TableLen?-1:i;
}

//加哨兵，从后往前
int Search_Seq(SSTable ST,ElemType key){
    ST.elem[0]=key;
    int i;
    for(i=ST.TableLen;ST.elem[i]!=key;i--);
    return i; //查找失败返回0
}
```

- 加入哨兵可以免去查找过程中判断越界。
- $ASL_{\text{成功}}=\frac{n+1}{2}$
- $ASL_{\text{失败}}=n+1$
- $O(n)$
- 若考虑查找不成功的情况， $ASL=P_{\text{成功}}ASL_{\text{成功}}+P_{\text{失败}}ASL_{\text{失败}}$
- 若成功和失败可能性相同， $ASL=\frac{3}{4}(n+1)$
- 优化1：若已知每个元素查找概率，应先按查找概率排序
- 优化2：对有序表（画判定树计算 $ASL_{\text{失败}}$）



一个成功结点的查找长度 = 自身所在层数
一个失败结点的查找长度 = 其父节点所在层数
默认情况下，各种失败情况或成功情况都等概率发生

优点：对表结构无要求，算法简单

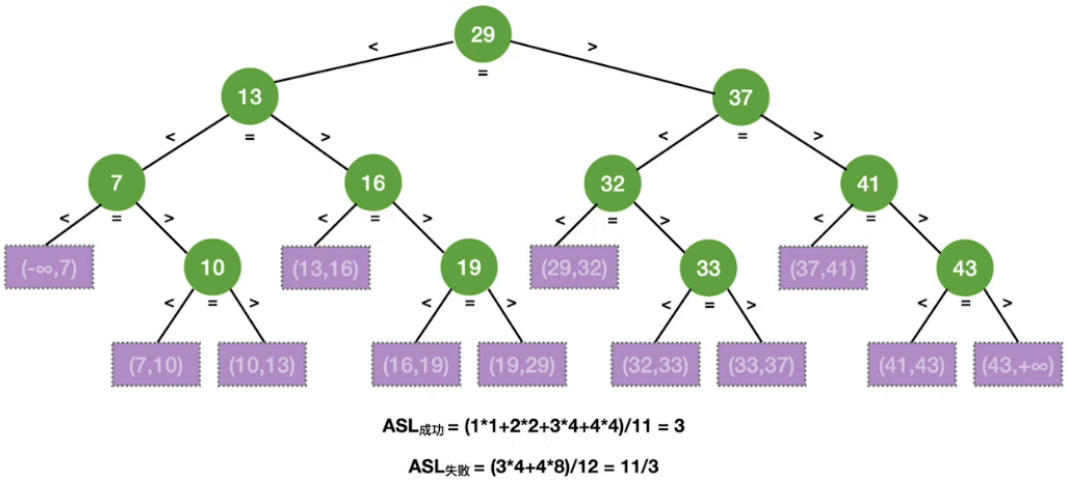
缺点：平均查找长度大

▼ 有序表查找（折半查找）

```
//对升序表
typedef struct{
   ElemType *elem;
   int TableLen;
}SSTable;

int Binary_Search(SSTable L,ElemType key){
   int low=0,high=L.TableLen-1,mid;
   while(low<=high){
      mid=(low+high)/2;
      if(L.elem[mid]==key) return mid;
      else if(L.elem[mid]>key) high=mid-1;
      else low=mid+1;
   }
```

```
    return -1;
}
```

查找判定树

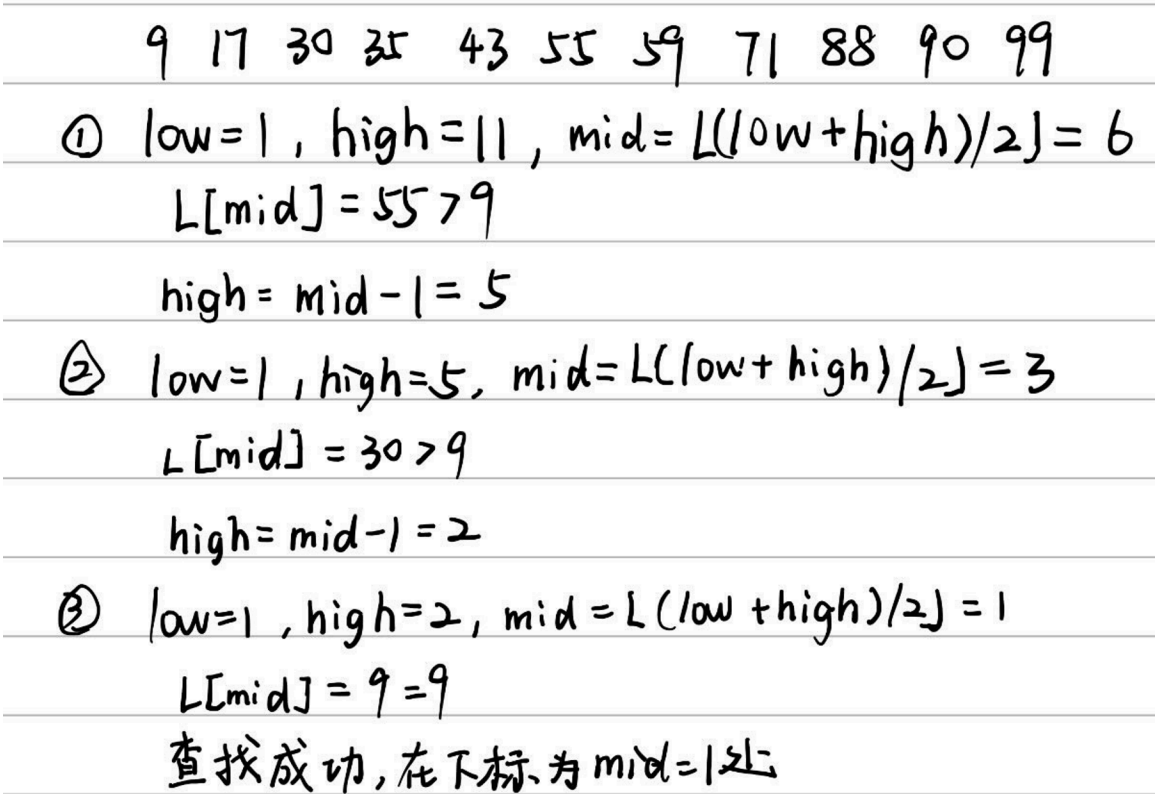


- 折半查找的判定树一定是平衡二叉树
- 元素个数为n时，树高（不包含失败结点） $h = \lfloor log_2(n) \rfloor + 1$
- 成功结点n个，失败结点n+1个
- 折半查找时间复杂度 $O(log_2 n)$
- 等概率折半查找时，查找成功的平均查找长度 $ASL_{bs} = \frac{n+1}{n}\log_2(n+1) - 1$
- n较大时， $ASL_{bs} = \log_2(n+1) - 1$
- ~~折半查找一定比顺序查找快~~

优点：效率高

缺点：不适用链表和无序表

1. 已知有序表为(9、17、30、35、43、55、59、71、88、90、99)，当用折半查找法查找9时，请写出具体的查找过程，包括low、high指针的位置变化及所做的比较操作。



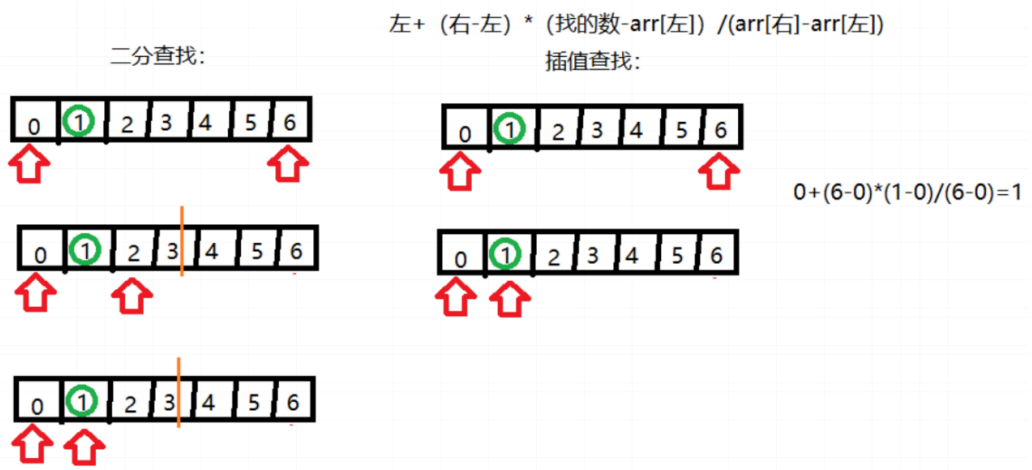
插值查找

$mid = 左 + (右 - 左) \times \frac{key - arr[左]}{arr[右] - arr[左]}$

- 适用于数据量较大，关键字分布比较均匀的查找表

```
int InsertionSearch(int a[], int value, int low, int high){
    int mid = low+(value-a[low])/(a[high]-a[low])*(high-low);
  if(a[mid] == value)
    return mid;
  if(a[mid]>value)
    return InsertionSearch(a, value, low, mid-1);
    if(a[mid]<value)
```

```
        return InsertionSearch(a, value, mid+1, high);
}
```



▼ 索引顺序表（分块查找）

```
//索引表
typedef struct{
    ElemType maxValue;
    int low,high; //可存起始地址，也可存下标范围
}Index;
//顺序表存储实际元素
ElemType List[100];
```

- 索引项：关键字项（子表最大关键字）、指针项（起始地址）
- 表长n，均匀分成b块，每块长度s，$b=\lceil n/s \rceil$，索引查找 L_b，块中查找 L_w
- 等概率、顺序查找成功：$ASL_{bs}=L_b+L_w=\frac{b+1}{2}+\frac{s+1}{2}=\frac{1}{2}(\frac{n}{s}+s)+1$
- s取$\sqrt{n}$时，$(ASL_{bs})_{min}=\sqrt{n}+1$
- 索引折半，块中顺序 $ASL_{bs}=\log_2(\frac{n}{s}+1)+\frac{s}{2}$
- 效率低于折半查找

例题：对2500个记录的表进行分块查找，理想的块长为：50。

▼ 动态查找表

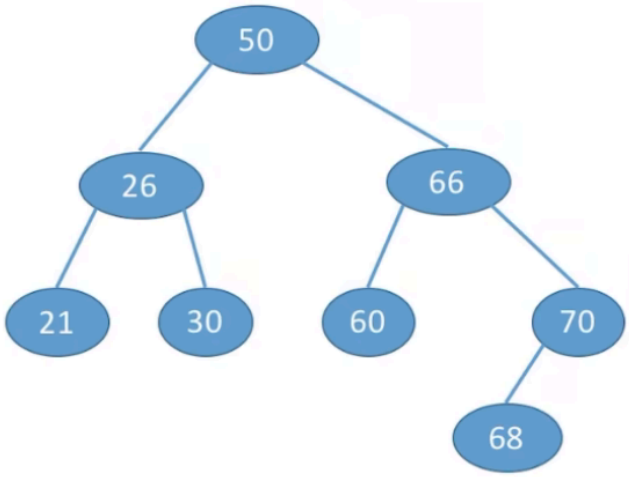
▼ 二叉排序树（BST）

左<根<右

```
//非递归查找
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL && key!=T→key){
        if(key<T→key) T=T→lchild;
        else T=T→rchild;
    }
    return T;
}

//递归查找
BSTNode *BSTSearch(BSTree T,int key){
    if(T=NULL) return NULL;
    if(key==T→key) return T;
    else if(key<T→key) return BSTSearch(T→lchild,key);
    else return BSTSearch(T→rchild,key);
}

//插入
int BST_Insert(BSTree &T,int k){
    if(T==NULL){
        T=(BSTree)malloc(sizeof(BSTNode));
        T→key=k;
        T→lchild=T→rchild=NULL;
        return 1;
    }
    else if(k==T→key) return 0;
    else if(k<T→key) return BST_Insert(T→lchild,k);
    else return BST_Insert(T→rchild,k);
}
```

```
//构造
void Create_BST(BSTree &T,int str[],int n){
    T=NULL;
    int i=0;
    while(i<n){
        BST_Insert(T,str[i]);
        i++;
    }
}

//删除结点
BSTNode *Delete(KeyType X,BSTNode *T){
    BSTNode *TmpCell;
    if(T==NULL) return NULL;
    else if(X<T→key) T→lchild=Delete(X,T→lchild);
    else if(X>T→key) T→rchild=Delete(X,T→rchild);
    else{
        if(T→lchild && T→rchild){ //用右子树最小代替p
            TmpCell=T→rchild;
            while(TmpCell→lchild){
                TmpCell=TmpCell→lchild;
            }
            T→key=TmpCell→key;
            T→rchild=Delete(T→key,T→rchild);
        }
        else{
            TmpCell=T;
            if(T→lchild==NULL) T=T→rchild;
            else if(T→rchild==NULL) T=T→lchild;
            free(TmpCell);
        }
    }
    return T;
}
```

- 空间复杂度 非递归$O(1)$ 递归最坏$O(n)$
- 构造（考）



例1：按照序列str={50, 66, 60, 26, 21, 30, 70, 68}建立BST

- 删除结点p
  1. p为叶子结点→直接删
  2. p只有左/右子树→孩子取代p
  3. p有左子树和右子树→左子树最大（最右下）或右子树最小（最左下）替代p，转换成1or2情况

性能分析

- 最好情况（二叉排序树各结点分布均匀） $ASL = O(\log_2 n)$
- 最坏情况 $ASL = O(n)$
- 平均情况 $ASL = O(\log_2 n)$
- 二叉排序树效率介于顺序查找和二分查找之间

▼ 例题

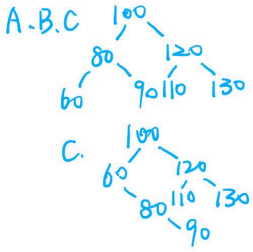
8. 分别用以下序列构造二叉排序树，与其他三个序列构造结果不同的是[C]

A 100 80 90 60 120 110 130

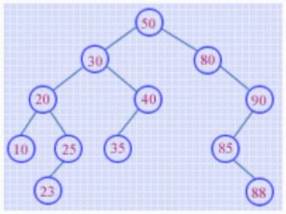
B 100 120 110 130 80 60 90

C 100 60 80 90 120 110 130

D 100 80 60 90 120 130 110



2. 如下所示二叉排序树，请先画出插入结点24以后的二叉排序树，在此基础上，请再画出删除结点30以后的二叉排序树，并说明插入和删除的步骤。



插入24：

① 50>24，进入左子树

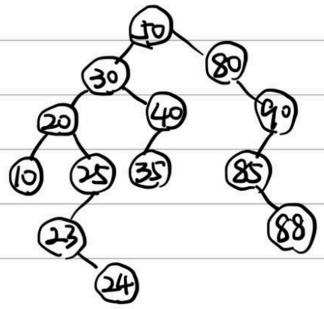
② 30>24，进入左子树

③ 20<24，进入右子树

④ 25>24，进入左子树

⑤ 23<24，进入右子树

⑥ 结点为空，插入24，如右图



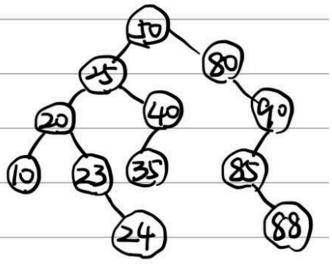
删除结点30

情况1① 删30，30有左子树和右子树 → 取左子树中最大的元素替代，进入左子树

② 20有右孩子 → 进入20右子树

③ 25无右孩子 → 25为30左子树中最大的，25替代30

④ 删25，25只有左子树 → 左孩子替代25，如右图

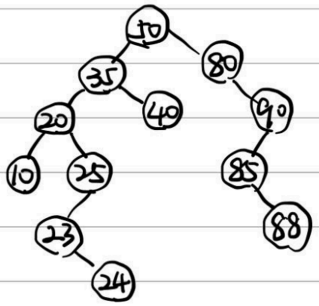


情况2 ① 删30，取右子树最小代替 → 进入右子树

② 40有左孩子 → 进入40左子树

③ 35无左孩子 → 35为右子树中最小，35替换30

④ 删35，无孩子 → 直接删



▼ 平衡二叉树（AVL）

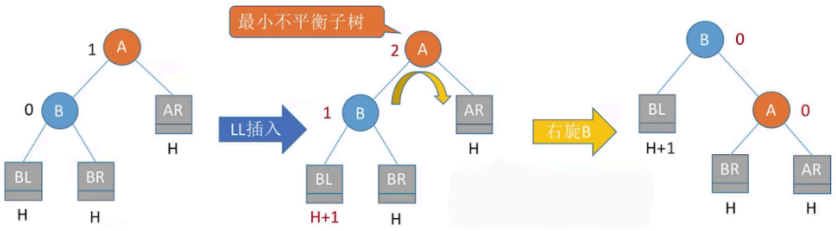
- 左子树和右子树都是AVL树且左子树和右子树高度之差不超过1
- 平衡因子=左子树高度-右子树高度　AVL树只能取-1，0，1
- 平均查找长度为 $O(\log_2 n)$，插入、删除时间复杂度 $O(\log_2 n)$

```
typedef struct AVLNode{
    int key;
    int balance;
    struct AVLNode *lchild,*rchild;
}AVLNode,*AVLTree;
```

▼ 插入

插入新结点导致不平衡，只需调整最小不平衡子树

- LL（左孩子的左子树）

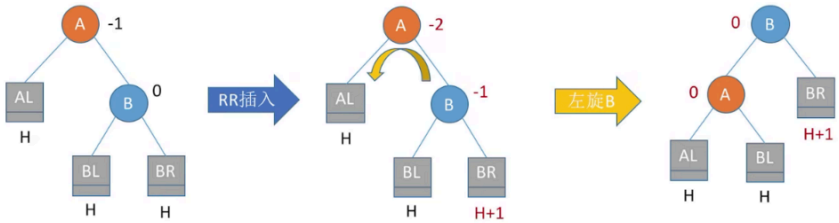


二叉排序树的特性：左子树结点值 < 根结点值 < 右子树结点值

BL<B<HR<A<AR

```
//右旋
f→lchild=p→rchild; //A的左孩子改为BR
p→rchild=f;  //B的右孩子改为A
gf→lchild/rchild=p;  //将B改为根结点
```

- RR（右孩子的右子树）

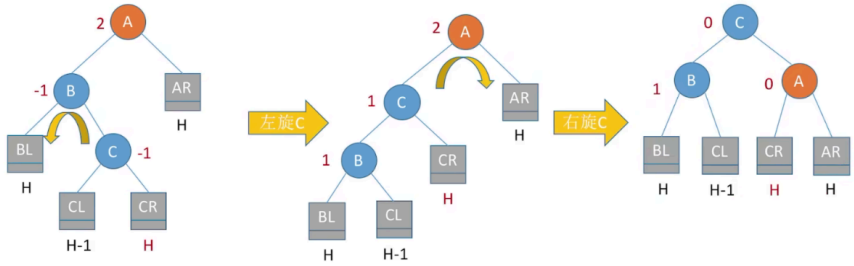


二叉排序树的特性：左子树结点值 < 根结点值 < 右子树结点值

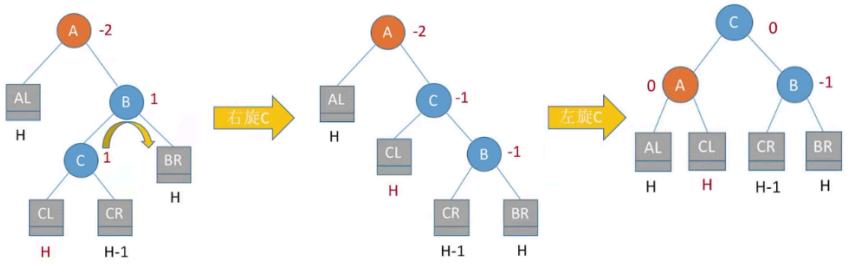
AL<A<BL<B<BR

```
//左旋
f→rchild=p→lchild; //A的右孩子改为BL
p→lchil=f;  //B的左孩子改为A
gf→lchild/rchild=p;  //将B改为根结点
```

- LR（左孩子的右子树）

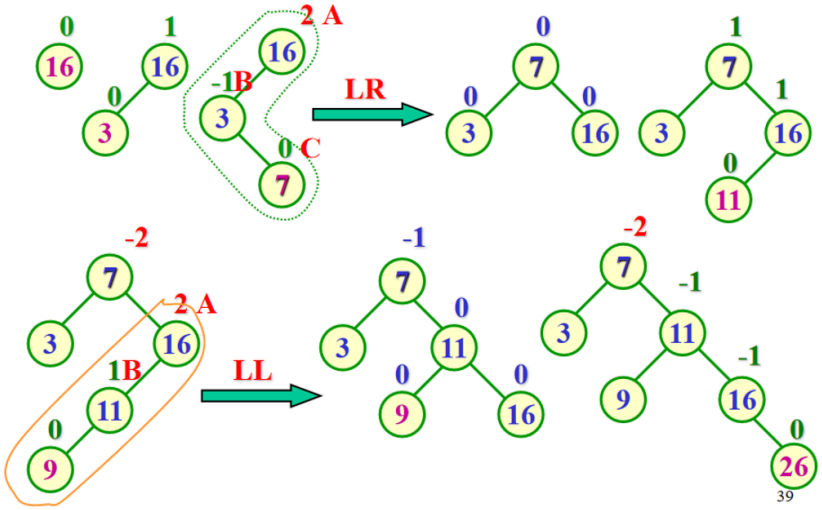


- RL（右孩子的左子树）

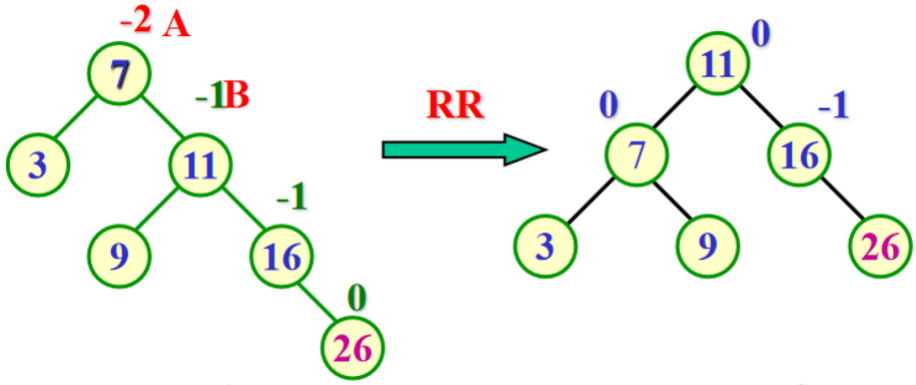


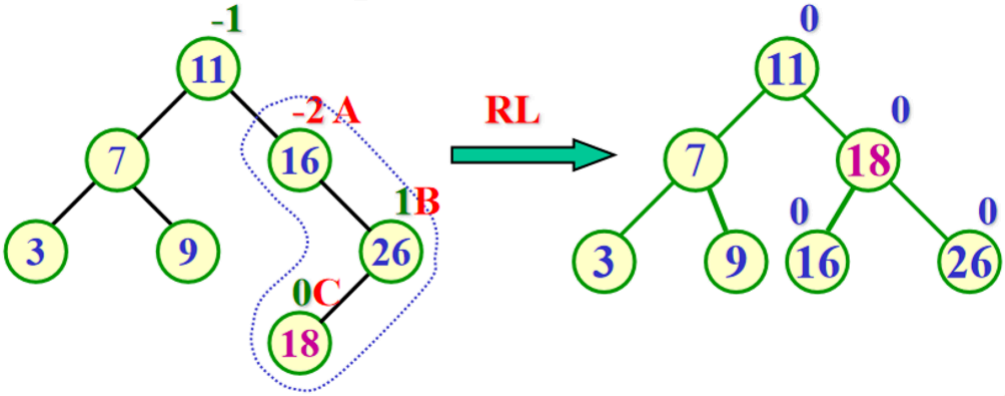
▼ 例题

例，key={ 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。

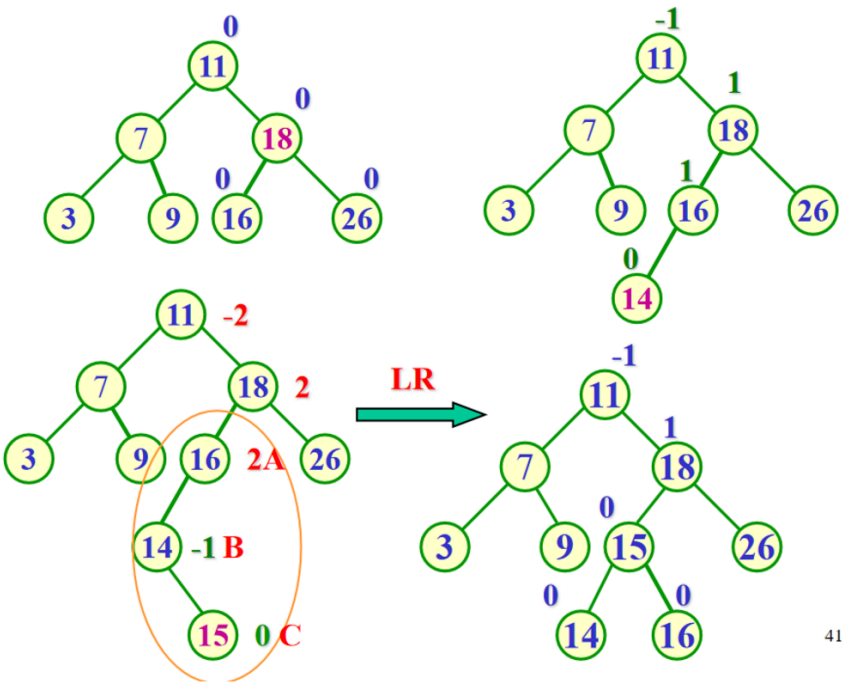


39



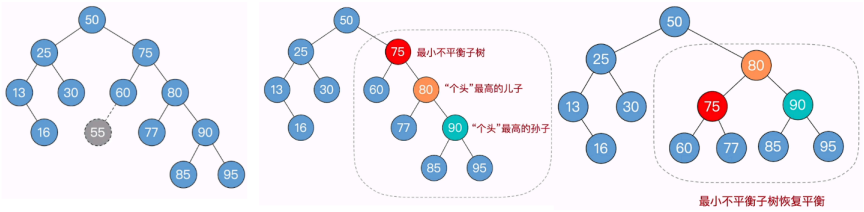


4



▼ 删除

1. 删除结点（遵循二叉排序树规则）
2. 找到最小不平衡子树
3. 最小不平衡子树下，找到高度最大的儿子和最大的孙子
4. 根据孙子位置，调整（LL，RR，LR，RL）
5. 不平衡向上传导，继续2.



▼ 例题

4．现有一棵无重复关键字的平衡二叉树（AVL 树），对其进行中序遍历可得到一个降序序列。下列关于该平衡二叉树的叙述中，正确的是 D 。

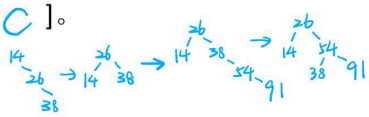
A．根结点的度一定为 2　　B．树中最小元素一定是叶结点

C．最后插入的元素一定是叶结点　　D．树中最大元素一定是无左子树

9. 对于关键字序列(14,26,38,54,91)，按序列次序创建一颗平衡二叉排序树，在等概率情况下查找成功时，其平均查找长度是[C]。

A. 7/5　　B. 9/5　　C. 11/5　　D. 13/5

$\frac{1+2\times2+3\times2}{5}=\frac{11}{5}$



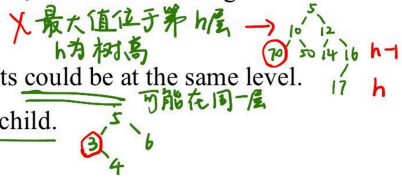
13. Assume the root is at level 1. Which of the following statements is **FALSE**? (B)

A. In a MinHeap, the second smallest key entry is always at level 2.

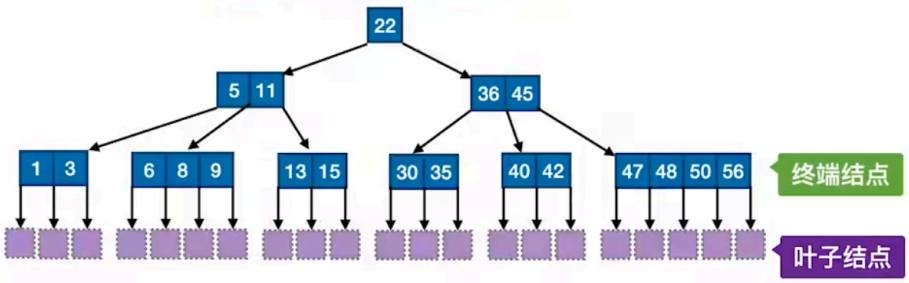
B. In a MinHeap, the largest key element is always at level *h*, where *h* is the height of the heap.

C. In an AVL tree, the largest and the smallest key elements could be at the same level.

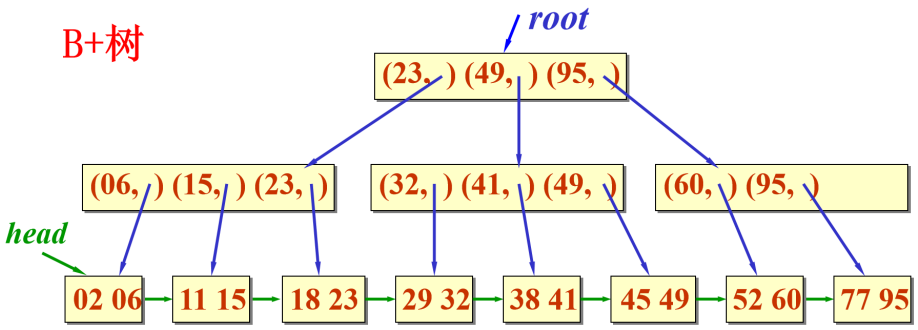
D. In an AVL tree, the smallest key entry may have a leaf child.



▼ B树



- 所有结点的孩子个数最大值称为B树的阶m（图中m=5）
- 一颗m阶B-树满足：
  - 每个结点最多m个孩子
  - 除根结点和叶子结点，其他结点至少有 $\lceil m/2 \rceil$ 个孩子
  - 根节点至少有两个孩子
  - 所有叶子结点在同一层
  - 有k个孩子的分支结点包含k-1个关键字



- 一颗m阶B+树满足
  - B树前四条
  - 有k个孩子的分支结点包含k个关键字
  - 内部结点都是作为索引，不存数据

▼ 哈希表

哈希方法：依哈希函数按关键字计算存储位置，查找时用同一个函数计算地址进行查找

哈希函数：转换函数

冲突：将不同的关键字映射到同一个地址上

▼ 哈希函数的构造方法

主要考虑的因素

- 计算哈希函数所需时间
- 关键字长度
- 哈希表大小
- 关键字分布
- 查找频率

1. **直接定址法**

$$Hash(key) = a * key + b$$

例：关键码集合为{100，300，500，700，800，900}，
选取哈希函数为Hash(key)=key/100，
则存储结构（哈希表）如下：

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

优点：不会产生冲突

缺点：占用连续地址空间，空间效率低

2. **数字分析法**

关键字的某几位组合成哈希地址。

例：有一组（例如80个）关键字，其样式如下：

3	4	7	0	5	2	4
3	4	9	1	4	8	7
3	4	8	2	6	9	6
3	4	8	5	2	7	0
3	4	8	6	3	0	5
3	4	9	8	0	5	8
3	4	7	9	6	7	1
3	4	7	3	9	1	9
位号：	①	②	③	④	⑤	⑥	⑦

讨论：

① 第1、2位均是“3和4”，第3位也只有“ 7、8、9”，因此，这几位不能用，余下四位分布较均匀，可作为哈希地址选用。

② 若哈希地址取两位（因元素仅80个），则可取这四位中的任意两位组合成哈希地址，也可以取其中两位与其它两位叠加求和后，取低两位作哈希地址。

3. **平方取中法**

平方后取中间的若干位作为哈希地址

例：2589的平方值为6702921，可以取中间的029为地址。

4. **折叠法**

法1：移位法 —— 将各部分的最后一位对齐相加。
法2：间界叠加法——从一端向另一端沿分割界来回折叠后，最后一位对齐相加。
例：元素42751896，用法1： 427＋518＋96=1041
用法2： 427 518 96—> 427+815+96 =1338

5. **除留余数法**

$$Hash(key) = key \mod p$$

若设计的哈希表长为m，则一般取p≤m且为质数（也可以是不包含小于20质因子的合数）

6. **随机数法**

$$Hash(key) = random(key)$$

适用于关键字长度不等的情况

▼ 冲突处理方法

同义词：通过Hash函数计算地址相同的元素

1. **开放定址法**

找下一个空位

- **线性探测法** $H_i = (Hash(key) + d_i) \mod m$

m为哈希表长度， d_i为增量序列

就近找空地址存

例：关键字集合 {47，7，29，11，16，92，22，8，3}.
哈希表: HT[11]； 哈希函数: Hash(key)=key mod 11;
冲突处理方法: 线性探测
H(47)=3
H(7)=7
H(29)=7，冲突，找下一个地址;
H_1=(Hash(29)+1) mod 11=8， 找到空闲地址，插入29.
H(11)=0; H(16)=5; H(92)=4;
H(22)=0, 冲突，找下一个地址;
H_1=(Hash(22)+1) mod 11=1，找到空闲地址，插入22.
H(8)=8, 冲突，找下一个地址;
H_1=(Hash(8)+1) mod 11=9，找到空闲地址，插入8.
Hash(3)=3，冲突，找下一个地址;
H_1=(Hash(3)+1) mod 11=4 冲突;
H_2=(Hash(3)+2) mod 11=5 冲突 ;
H_3=(Hash(3)+3) mod 11=6 找到空闲地址。

0	1	2	3	4	5	6	7	8	9	10
11	22		47	92	16	3	7	29	8	

△ ▲ △ △

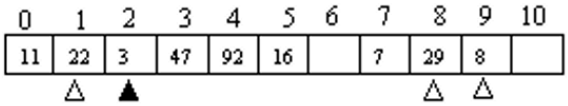
缺点：容易堆积

- **二次探测法（平方探测法）**

$$H_i = (Hash(key) \pm d_i) \mod m$$

d_i增量序列为 $1^2, -1^2, 2^2, -2^2, ..., q^2$

例：关键字集合 {47，7，29，11，16，92，22，8，3}.
哈希表: HT[11]; 哈希函数: Hash(key)=key mod 11;
冲突处理方法: 二次探测



Hash(3)=3，冲突
H_1=(Hash(3)+1^2) mod 11=4 冲突；
H_2=(Hash(3)-1^2) mod 11=2 找到空闲地址，插入

避免连续冲突，探测范围更广。

- **伪随机数探测法**

  d_i为一个伪随机序列

2. **再哈希法**

   $H_i = RH_i(key)$

   RH_i是不同的哈希函数，产生冲突就用另一个

   优点：不易产生聚集

   缺点：增加计算时间

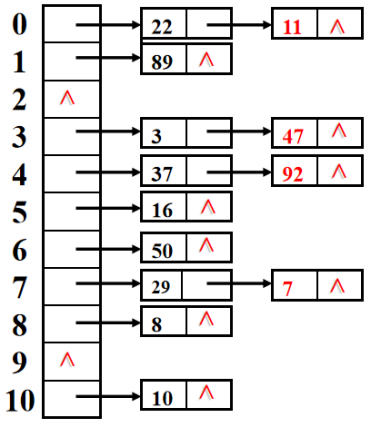
3. **链地址法**

   具有相同哈希地址（同义词）记录成一个链表

例：设{ 47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89 }的哈希函数为：

Hash(key)=key mod 11，用链地址法处理冲突，则建表如右图所示。

注：有冲突的元素可以插在表尾，也可以插在表头



4. **建立一个公共溢出区**

除哈希基本表外，设立一个溢出向量表

▼ 哈希表的查找及分析

Example： A set of keys {47, 7, 29, 11, 16, 92, 22, 8, 3}

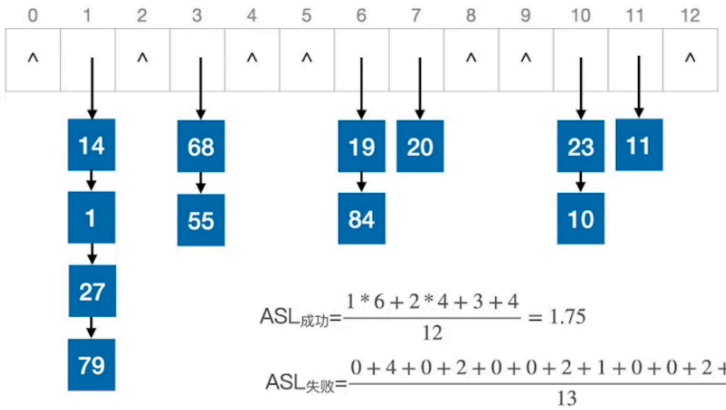
Hash table: HT[11];

Hash function: Hash(key)=key mod 11;

Collision resolution policy: Linear Probing.

$$ASL = \frac{1}{9}\sum_{i=1}^{9} Ci = \frac{1}{9}(1+2+1+1+1+4+1+2+2) = \frac{15}{9}$$

例：有一堆数据元素，关键字分别为 {19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 H(key)=key%13



$$ASL_{成功}=\frac{1*6+2*4+3+4}{12}=1.75$$

$$ASL_{失败}=\frac{0+4+0+2+0+0+2+1+0+0+2+1+0}{13}=0.92$$

估算法：

装填因子$\alpha = \frac{\text{表中记录数}}{\text{散列表长度}}$

α越大，冲突越多，查找次数越多

拉链法：$ASL \approx 1 + \frac{\alpha}{2}$

线性探测法：$ASL \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$

随机探测法：$ASL \approx -\frac{1}{\alpha}\ln(1-\alpha)$

哈希函数不可逆，因为有冲突

3. 已知待散列的线性表为（22，16，43，52，62），散列用的一维地址空间为[0..6]，假定选用的哈希函数是H（K）= K mod 7，若发生冲突采用线性探测法处理，请构造出这个哈希表，写出构建过程，并计算平均查找长度。

① $H(22)=22 \bmod 7=1$ 有空闲地址，插入22

② $H(16)=16 \bmod 7=2$ 有空闲地址，插入16

③ $H(43)=43 \bmod 7=1$ 冲突

$H_1=(H(43)+1) \bmod 7=2$ 冲突

$H_2=(H(43)+2) \bmod 7=3$ 有空闲地址，插入43

④ $H(52)=52 \bmod 7=3$ 冲突

$H_1=(H(52)+1) \bmod 7=4$ 有空闲地址，插入52

⑤ $H(62)=62 \bmod 7=6$ 有空闲地址，插入62

0	1	2	3	4	5	6
22	16	43	52			62

$ASL=\frac{1}{5}\times(1+1+3+2+1)=1.6$

五、 已知散列表的地址空间为 A[0..10]，散列函数 H（k）=（3k+5） mod 11，采用线性探测再散列法处理冲突。**（10 分）**

（1）请将关键字序列{25, 17, 92, 51, 33, 29, 83, 123, 42, 105}依次插入到下面的散列表中，给出下表中各空的值；

（2）并计算出在等概率情况下查找成功和不成功时的平均查找长度。

关键字	25	17	92	51	33	29	83	123	42	105
H(k)	3	1	6	4	5	4	1	0	10	1

散列地址		0	1	2	3	4	5	6	7	8	9	10
关键字		123	17	83	25	51	33	92	29	105		42
比较次数	查找成功	1	1	2	1	1	1	1	4	8		1
	查找失败	10	9	8	7	6	5	4	3	2	1	11

该位置到空位置有几个

$ASL_{成功} = \frac{1+1+2+1+1+1+1+4+8+1}{10} = 2.1$

$ASL_{不成功} = \frac{10+9+8+7+6+5+4+3+2+1+11}{11} = 6$