

§ 9. 查找

9.1. 基本概念

查找表：同一类型的数据元素构成的集合，数据元素间仅存在同属于一个集合的关系

查找的基本操作：

查找“特定”元素是否在表中	} } }	静态查找表
查找“特定”元素的某个属性		
查找后在表中插入元素		动态查找表
查找后在表中删除元素		

关键字：用于与其它数据元素相区分的某数据项的值

★ 能唯一区分的称为**主关键字**，否则称为**次关键字**

★ 若需要多个**次关键字**才能唯一区分，则按顺序分别称为第1、第2、...、第n关键字

=> 多关键字时，除第1关键字**整体有序**外，第i个关键字($i > 1$)只有在 $i-1$ 项相等时才**局部有序**

★ 如果数据元素需要全部数据项才能区分唯一，则称为**全关键字**

★ 数据元素又称为记录，由若干数据项组成，数据项也称为字段、属性

查找：根据给定值，在查找表中寻找**关键字值=给定值**的数据元素

{ 找到：成功，返回数据元素的信息/指针/位置
 找不到：不成功，空记录/空指针/0

平均查找长度(ASL)：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找长度

§ 9. 查找

9.1. 基本概念

关键字类型说明及比较约定 (P. 215 略有改动)

★ 常用关键字:

```
typedef float KeyType;  
typedef int   KeyType;  
typedef char *KeyType;
```

★ 数据元素的定义:

```
typedef struct {  
    KeyType key;  
    ... //其它数据项  
} SElemType;
```

● 大多数教材及参考书都直接用int型，要会转换

★ 两个关键字的比较约定 (int型):

	int型	float型	char *型
#define EQ(a, b)	((a)==(b))	(fabs((a)-(b))<1e-6)	(!strcmp((a), (b)))
#define LT(a, b)	((a)< (b))	((a)< (b))	(strcmp((a), (b))<0)
#define LQ(a, b)	((a)<=(b))	((a)<=(b))	(strcmp((a), (b))<=0)

● 另三个比较操作分别是 !EQ(a, b) / !LT(a, b) / !LQ(a, b)

● C++中char *可直接用string替代，且直接用比较运算符（已重载）即可比较

§ 9. 查找

9.2. 静态查找表

9.2.1. 静态查找表的形式化定义

P. 216 4种基本操作

9.2.2. 顺序表的查找

★ 适用范围：线性结构(顺序/链式存储)，关键字无序

★ 类型定义(顺序存储)

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

★ 算法实现(顺序存储)

```
/* 建立静态查找表 */
```

```
Status Create(SSTable &ST, int n)
{
    int i;
    /* 多申请一个空间, [0]暂不用, 仅用[1]-[n] */
    ST.elem = (ElemType *)malloc((n+1)*sizeof(ElemType));
    //C++方式: ST.elem = new ElemType[n+1];
    if (ST.elem==NULL)
        exit(LOVERFLOW);
    for (i=1; i<=n; i++) {
        cin >> ST.elem[i].key;
        cin >> ST.elem[i].其它项;
    }
    ST.length = n;

    return OK;
}
```

```
/* 销毁静态查找表 */  
Status Destroy(SSTable &ST)  
{  
    if (ST.elem)  
        free(ST.elem); //delete ST.elem;  
    ST.length = 0;  
  
    return OK;  
}
```

/* 遍历静态查找表 */

Status Traverse(SSTable ST, Status (*visit)(ElemType e))

```
{  int i;
    for (i=1; i<=ST.length; i++)
        if ((*visit)(ST.elem[i])!=FALSE)
            return ERROR;
    return OK;
}
```

遍历方法一

Status Traverse(SSTable ST, Status (*visit)(ElemType e))

```
{  ElemType *p = &(L.elem[1]);
    int      i = 1;
    while(i<=ST.length && (*visit)(*p++)!=TRUE)
        i++;
    if (i<=L.length)
        return ERROR;
    return OK;
}
```

遍历方法二

方法二效率高

具体已在第2章讨论过，不再重复
本章中为了直观，认为这两种方式
效率相同

/* 查找静态表(找到返回下标1-n, 否则返回0) */

/* 方法一 */

```
int Search(SSTable ST, KeyType key)
{
    int i;
    for(i=1; i<=ST.length; i++)
        if (EQ(ST.elem[i].key, key))
            return i;
    return 0;
}
```

n次循环, 每个循环比较两次
1次判断表是否结束
1次判断是否相等

/* 方法二 */

```
int Search(SSTable ST, KeyType key)
{
    int i=1;
    while(i<=ST.length&& !EQ(ST.elem[i].key, key))
        i++;

    return (i<=ST.length) ? i:0;
}
```

方法一与方法二性能相等

```
/* 查找静态表(找到返回下标1-n, 否则返回0) */
```

```
/* 方法三(P. 216-217 算法9.1) */
```

```
int Search(SSTable ST, KeyType key)
{
    int i=1;
    ST.elem[0].key = key; //[0]原来不用
    for(i=ST.length; !EQ(ST.elem[i].key, key); i--)
        ; //空语句

    return i;
}
```

[0]用做监视哨
每次只需比较key是否相等
不需要判断整个表是否结束
每次循环只比较1次, 效率
高于前两种方法

§ 9. 查找

9.2. 静态查找表

9.2.2. 顺序表的查找

★ 性能分析(方法三)

假设每次查找都成功:

n个元素, 每个元素被查找的概率为 $P_i = \frac{1}{n}$ (等概率)

第1个元素: 比较n次

...

第n个元素: 比较1次

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

假设查找不成功 (成功不成功各半):

n个元素, 每个元素被查找的概率 $P_i = \frac{1}{2n}$ (等概率)

查找成功 : $ASL = \frac{n+1}{4}$

查找不成功: 每次比较次数均为n+1

$$ASL = \frac{n+1}{2}$$

$$\text{总} ASL = \frac{3}{4}(n+1)$$

§ 9. 查找

9.2. 静态查找表

9.2.2. 顺序表的查找

★ 性能分析(方法三)

不等概率下的查找效率的提高:

- 若能实现预知每个数据元素的查找概率, 则可以按概率从小到大排列, 因为查找从尾部开始, 故概率大的元素比较次数少, 提高了效率
- 若无法预知每个数据元素的查找概率, 则可以在每个数据元素中附设一个访问频度域, 保证元素按访问频度从小到大排列; 或者最新查找的记录移动至表尾

(此方法虽然可以提高查找效率, 但数据的移动仍需要花较大的代价)

★ 顺序查找的优缺点

优点: 算法实现简单, 易于理解, 数据排列无要求

缺点: 平均查找长度较大, 不适合大数据量的查找

§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

9.2.3.1. 二分法查找(折半查找)

★ 适用范围：顺序存储的线性结构，关键字有序

★ 类型定义(顺序存储)

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




与顺序查找相同




★ 查找方法

- ① 初值 $low=1$, $high=n$ (上界和下界)
- ② 令 $mid = \lfloor (low+high)/2 \rfloor$ (中间位置)
- ③ 若 $key=ST.elem[mid].key$: 找到并结束查找
 $key < ST.elem[mid].key$: 上半区(令 $high=mid-1$)
 $key > ST.elem[mid].key$: 下半区(令 $low = mid+1$)
- ④ 若 $low \leq high$, 则重复步骤②③, 若 $low > high$, 则结束(未找到)

假设顺序表为(1, 3, 9, 18, 23, 73, 89, 91, 97)




查找3




1	3	9	18	23	73	89	91	97
								
low=1				mid=5				high=9




1	3	9	18	23	73	89	91	97
								
low=1	mid=2		high=4					

假设顺序表为(1, 3, 9, 18, 23, 73, 89, 91, 97)

查找91

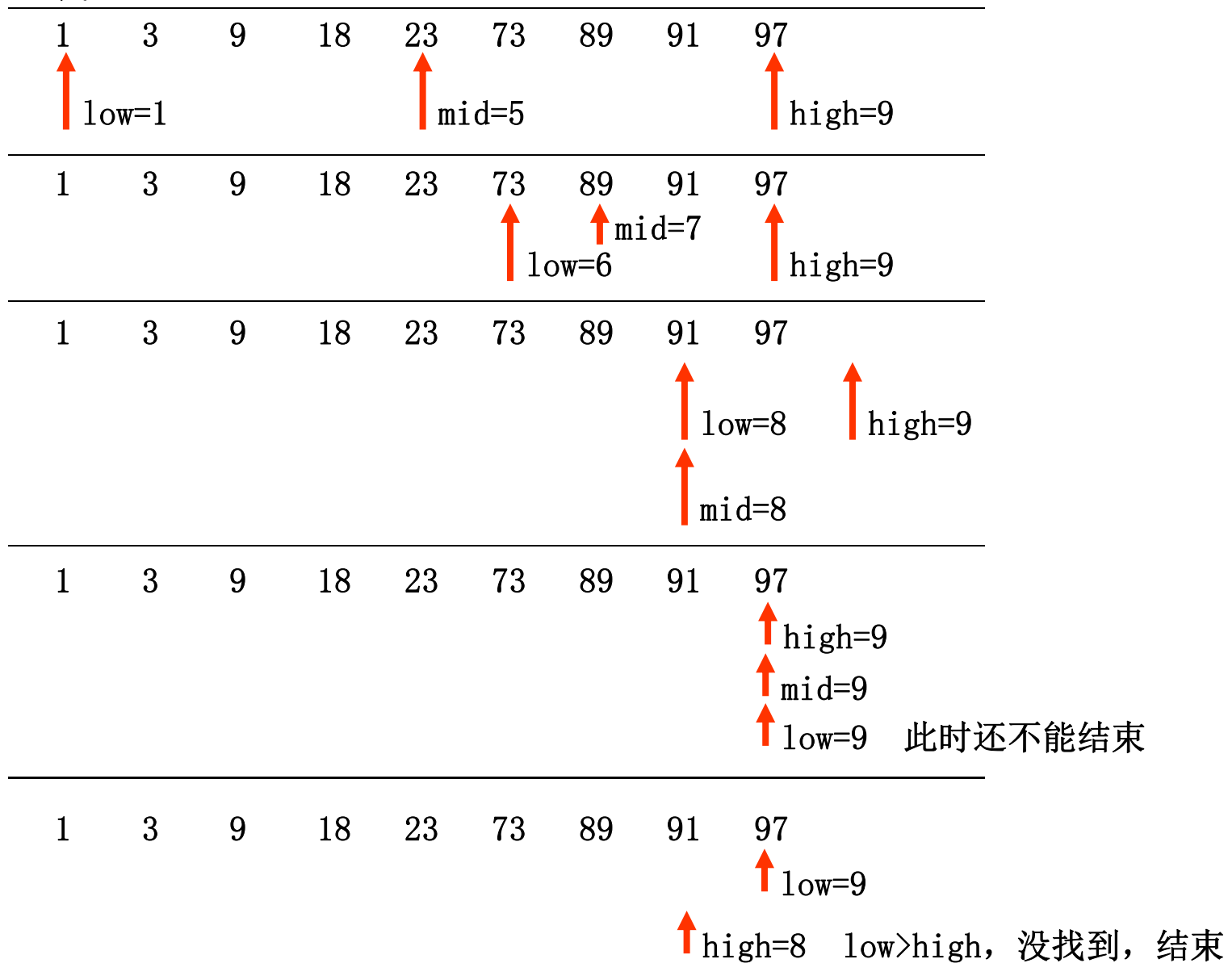
1	3	9	18	23	73	89	91	97
								
low=1				mid=5			high=9	

1	3	9	18	23	73	89	91	97
								
					low=6	mid=7	high=9	

1	3	9	18	23	73	89	91	97
								
							low=8	high=9
								mid=8

假设顺序表为(1, 3, 9, 18, 23, 73, 89, 91, 97)

查找94



§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

9.2.3.1. 二分法查找(折半查找)

★ 适用范围：顺序存储的线性结构，关键字有序

★ 类型定义(顺序存储)

★ 查找方法

★ 算法实现 (P. 220 算法9.2)

```
int Search_binary(SSTable ST, KeyType key)
{
    int low=1, high=ST.length, mid;

    while(low<=high) {
        mid=(low+high)/2;    //C++中自然向下取整
        if (EQ(key, ST.elem[mid].key))
            return mid;
        else if (LT(key, ST.elem[mid].key))
            high = mid-1;
        else    //只有key>ST.elem[mid].key这种情况了
            low = mid+1;
    }

    return 0; //找不到
}
```

§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

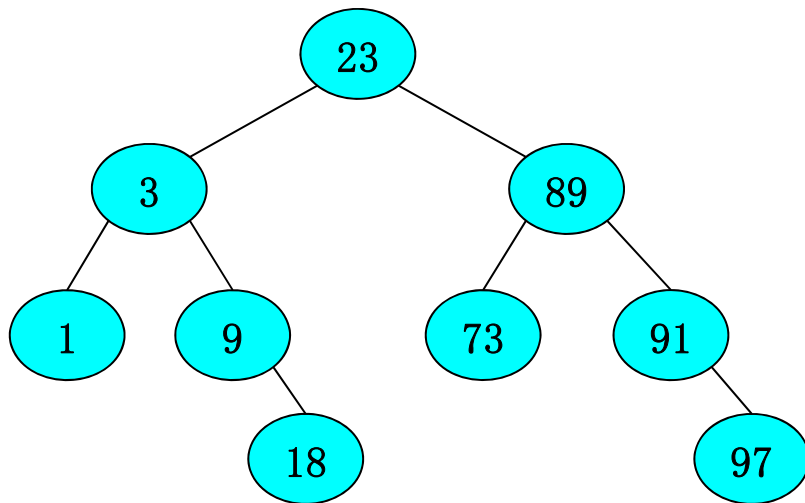
9.2.3.1. 二分法查找(折半查找)

★ 二叉判定树

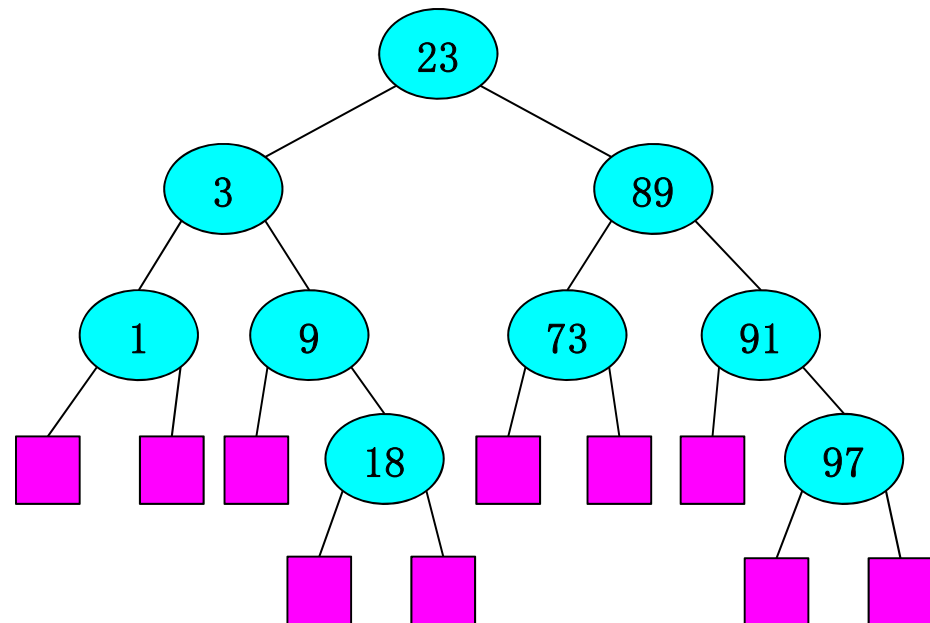
递归定义：根结点对应查找表的中点(mid)

左右子树分别对应上、下半区

1 3 9 18 23 73 89 91 97



仅9种找到的情况



9种找到+10种找不到的情况

§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

9.2.3.1. 二分法查找(折半查找)

★ 二叉判定树

递归定义：根结点对应查找表的中点(mid)，左右子树分别对应上、下半区

- 与二分法的比较顺序相对应的二叉树，中序有序
- 查找过程：从根结点到该结点的路径
- 比较次数：该结点所在的层次
- 特点：是相同数量的结点中具有最小层次的二叉树(与完全二叉树的层次相同)，
叶子结点层次最多差1，深度为 $\lfloor \log_2 n \rfloor + 1$

★ 性能分析

假设等概率查找 $P_i = \frac{1}{n}$ ，设 $n=2^h-1$ (满二叉树，高h)

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

当n足够大， $\frac{n+1}{n} \approx 1$ ， $ASL \approx \log_2(n+1) - 1$

★ 二分查找的优缺点

优点：查找时所用的比较次数少，效率高

缺点：只能是顺序存储的有序表，不适合动态查找表(维护有序的代价大)

§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

9.2.3.2. 斐波那契查找

斐波那契数列: $F_0=1$ $F_1=1$ $F_i=F_{i-2}+F_{i-1}$ ($i \geq 2$)

假设 n 个元素的有序序列, $F_{u-1} \leq n < F_u$

则key与ST.elem[F_{u-1}]进行比较

key==ST.elem[F_{u-1}]: 找到

key< ST.elem[F_{u-1}]: 在ST.elem[1]-[$F_{u-1}-1$]间找

key> ST.elem[F_{u-1}]: 在ST.elem[$F_{u-1}+1$]-[F_u-1]间找

★ 斐波那契数列足够大时, 前后项比值接近0.618, 因此也称为黄金分割法查找, 直接用 $mid = low + (high - low + 1) * 0.618$ 即可

§ 9. 查找

9.2. 静态查找表

9.2.3. 有序表的查找

9.2.3.3. 差值查找(精算点查找)

n个元素的有序序列，假设最小关键字为1，最大为h

则令

$$i = \frac{key - ST.elem[1].key}{ST.elem[h].key - ST.elem[1].key} (h - 1 + 1)$$

做为上下半区的分隔，比较key和ST.elem[i].key的值

★ 根据要查找的值计算出大致的位置

★ 要求关键字有序且分布均匀

§ 9. 查找

9. 2. 静态查找表

9. 2. 1. 静态查找表的形式化定义

9. 2. 2. 顺序表的查找

9. 2. 3. 有序表的查找

9. 2. 4. 静态树表的查找 (略)

§ 9. 查找

9.2. 静态查找表

9.2.5. 索引顺序表的查找

★ 含义

表中元素局部无序，整体无序，**但分块有序**

n个元素的表，分为b块，分块有序，块内无序

(第i块的最大值 < 第i+1块的最小值)

★ 查找算法

① 为表建立一个索引表，有b项，每项记录每个块中最大的关键字及块的起始位置

② 先折半查找索引表(索引为顺序表)，再顺序查找块

先顺序查找索引表(索引为链表)，再顺序查找块

1	2	3
22	48	86
1	7	13

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
22	12	13	8	9	20	33	42	44	38	24	48	60	58	74	49	86

§ 9. 查找

9.2. 静态查找表

9.2.5. 索引顺序表的查找

★ 性能分析

设n个元素, 均分为b块, 每块中s个元素, 则 $b = \lceil n/s \rceil$

假设查找概率相等, 块概率 $p=1/b$, 块内概率 $q=1/s$

① 先顺序查找索引表(索引为链表), 再顺序查找块

$$ASL = ASL_b + ASL_s = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2} \left(\frac{n}{s} + s \right) + 1$$

② 先折半查找索引表(索引为顺序表), 再顺序查找块

$$\begin{aligned} ASL &= ASL_b + ASL_s = \log_2(b+1) - 1 + \frac{s+1}{2} \\ &= \log_2 \left(\frac{n}{s} + 1 \right) + \frac{s}{2} \end{aligned}$$

§ 9. 查找

9.3. 动态查找表

9.3.1. 基本概念

★ 动态查找表的特点

表结构在查找过程中会动态发生变化

推论：表结构本身在查找过程中动态生成

★ 动态查找表的形式定义

P. 226-227

相比静态查找表，增加了插入/删除两个操作

§ 9. 查找

9.3. 动态查找表

9.3.1. 基本概念

9.3.2. 二叉排序树

★ 二叉排序树的递归定义

若左子树不为空，则左子树所有结点均小于根结点

若右子树不为空，则右子树所有结点均大于根结点

其左右子树也是二叉排序树

- 二叉排序树也称为二叉检索树、二叉查找树

- 二叉排序树的中序遍历序列有序

★ 存储结构

同二叉树的二叉链表方式

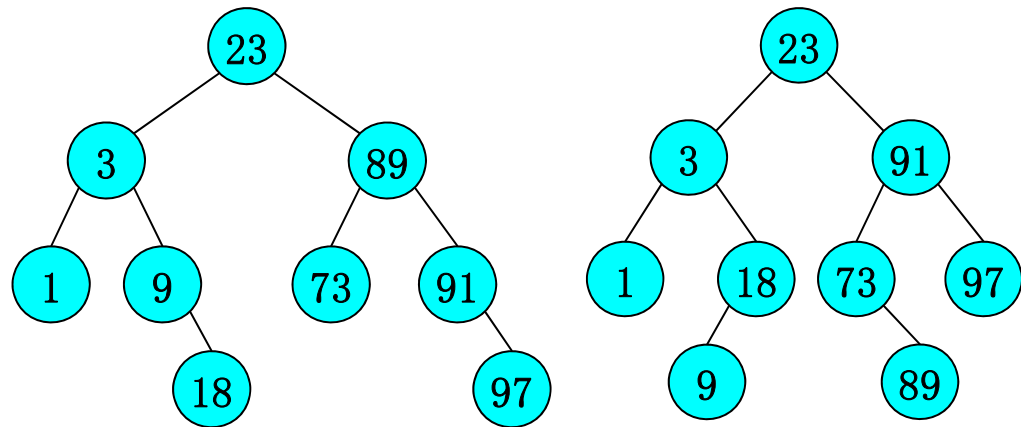
★ 基本操作

查找：成功 ：指向结点的指针

不成功：返回空指针

插入：保证插入后仍为二叉排序树

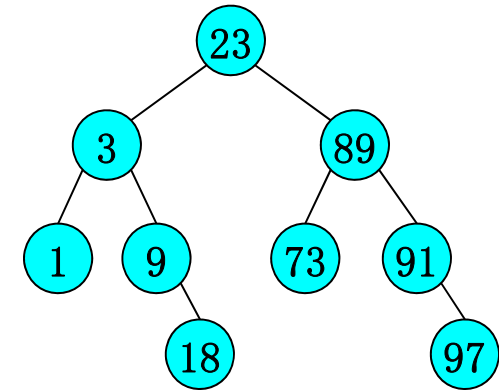
删除：保证删除后仍为二叉排序树



★ 二叉排序树的查找算法 (P. 228算法9.5(a))

```
BiTree SearchBST(BiTree T, KeyType key)
```

```
{  
    if ((!T) || (EQ(key, T->data.key))  
        return T;    //T为空则返回NULL  
    else if (LT(key, T->data.key)  
        return (SearchBST(T->lchild, key)); //递归左子树  
    else  
        return (SearchBST(T->rchild, key)); //递归右子树  
}
```



- 1、BiTree是第6章中二叉链表的定义，Data中单独列出key即可
- 2、找到返回指向结点的指针，找不到返回NULL

问：假设规定如果找不到则返回应该插入的位置，应该如何改进？

§ 9. 查找

9.3. 动态查找表

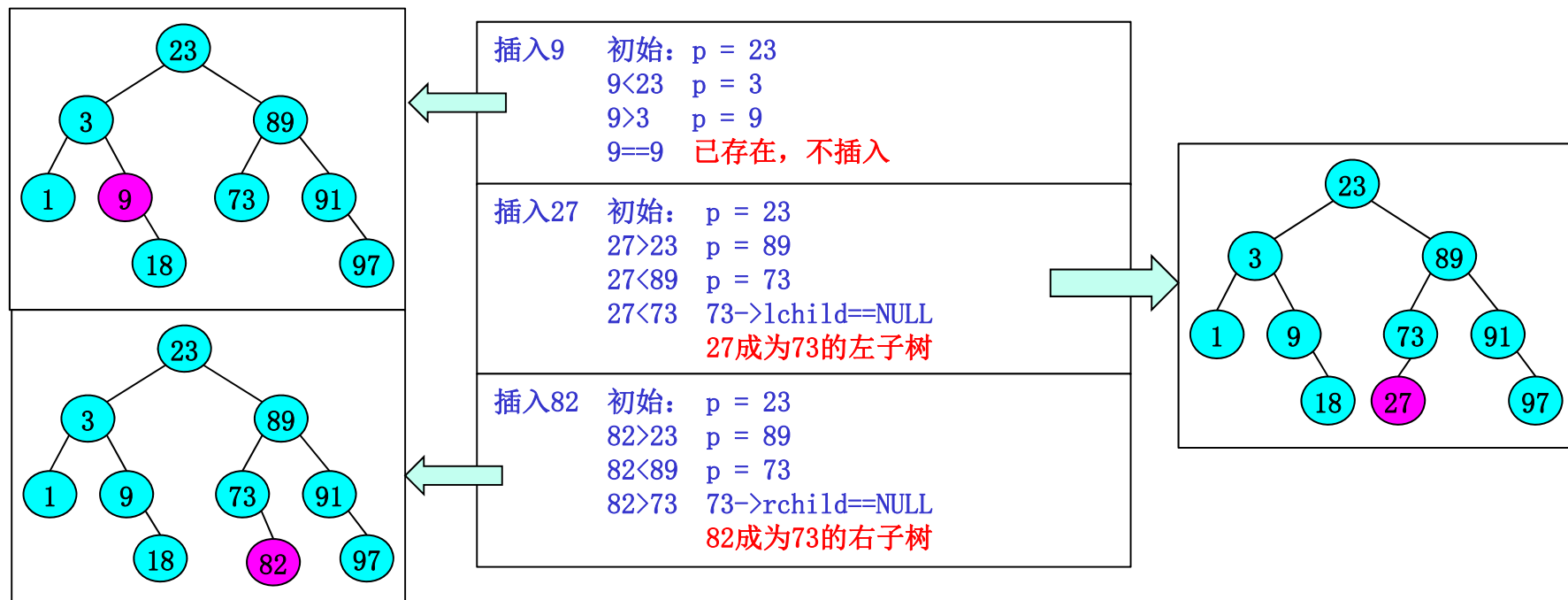
9.3.2. 二叉排序树

★ 二叉排序树的插入算法

设T为指向根结点的指针，p初始指向T，key为插入结点

- ① 若 $p == \text{NULL}$ ，则插入成为根结点
- ② 若 $\text{key} == p \rightarrow \text{key}$ ，则返回 (已存在，不插入)
- ③ 若 $\text{key} < p \rightarrow \text{key}$ ，则若 $p \rightarrow \text{lchild} == \text{NULL}$ ，将key插入为左子树，返回，否则令 $p = p \rightarrow \text{lchild}$ ，重复② ③至返回
若 $\text{key} > p \rightarrow \text{key}$ ，则若 $p \rightarrow \text{rchild} == \text{NULL}$ ，将key插入为右子树，返回，否则令 $p = p \rightarrow \text{rchild}$ ，重复② ③至返回

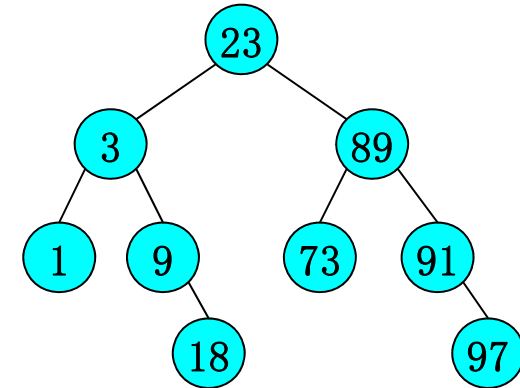
● 插入结点一定成为叶子



★ 二叉排序树的改进查找算法 (P. 228算法9.5(b))

Status SearchBST(BiTree T,KeyType key,BiTree f,BiTree &p)

```
{  if (!T) {
    p = f;
    return FALSE; //未找到,此时p指向插入位置的父结点
  }
  else if (EQ(key, T->data.key)) {
    p = T;
    return TRUE;  //找到
  }
  else if (LT(key, T->data.key))
    return SearchBST(T->lchild, key, T, p); //左子树
  else
    return SearchBST(T->rchild, key, T, p); //右子树
}
```



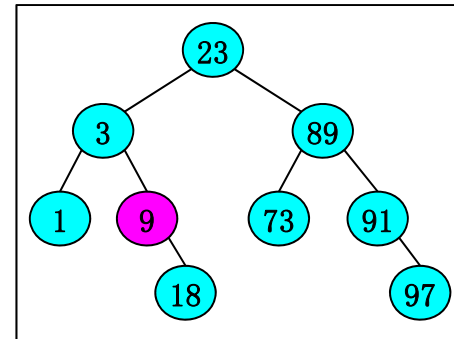
T : 要查找的二叉树
key: 查找的关键字
f : T的双亲, 初始为NULL
p : 找到则指向该元素, 找不到则指向即将插入位置的双亲
即要插入结点成为p的左子树(小)或右子树(大)

找9	T	key	f	p
	23	9	NULL	NULL
	3	9	23	NULL
	9	9	3	NULL
	9	9	3	9 //TRUE
找27	T	key	f	p
	23	27	NULL	NULL
	89	27	23	NULL
	73	27	89	NULL
	NULL	27	73	NULL
		27	73	73 //FALSE
找82	T	key	f	p
	23	82	NULL	NULL
	89	82	23	NULL
	73	82	89	NULL
	NULL	82	73	NULL
		82	73	73 //FALSE

★ 二叉排序树的插入算法 (P. 228-229算法9.6)

Status InsertBST(BiTree &T, ElemType e)

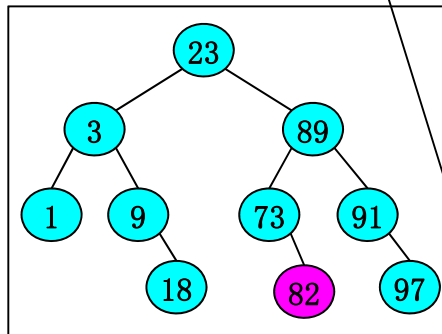
```
{  if (!SearchBST(T, e.key, NULL, p)) {
    if (!(s = (BiTree)malloc(sizeof(BiTreeNode)) ) )
        exit(LOVERFLOW);
    s->data = e;
    s->lchild = s->rchild = NULL;
    if (!p)
        T = s; //新结点成为根结点
    else if (LT(e.key, p->data.key))
        p->lchild = s; //新结点成为p的左孩子(p肯定无左孩子)
    else
        p->rchild = s; //新结点成为p的右孩子(p肯定无右孩子)
    return TRUE;
}
else
    return FALSE; //结点已存在, 不需要插入
}
```



找9

T	key	f	p
23	9	NULL	NULL
3	9	23	NULL
9	9	3	NULL
9	9	3	9

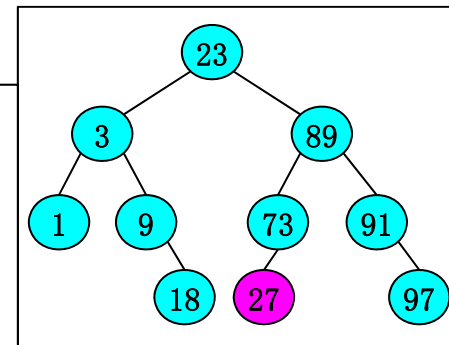
//TRUE



找27

T	key	f	p
23	27	NULL	NULL
89	27	23	NULL
73	27	89	NULL
NULL	27	73	NULL
27	27	73	73

//FALSE



找82

T	key	f	p
23	82	NULL	NULL
89	82	23	NULL
73	82	89	NULL
NULL	82	73	NULL
82	82	73	73

//FALSE

§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

★ 二叉排序树的删除算法

设p指向要删除的结点，f指向p的双亲(不失一般性，假设p为f的左子树)

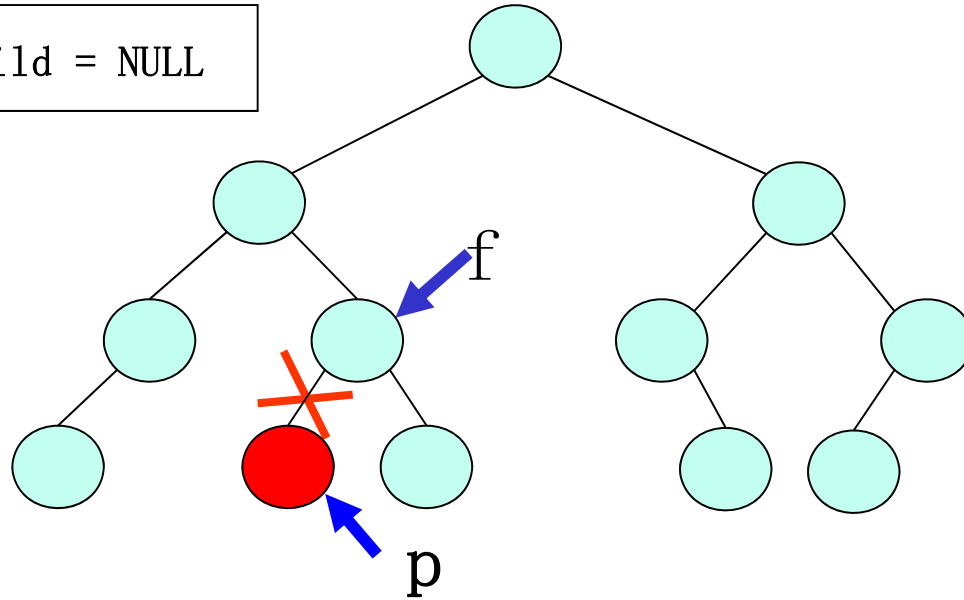
- ① 若p->lchild==NULL 且 p->rchild==NULL (p是叶子)，则f->lchild=NULL即可
- ② 若p->lchild==NULL 或 p->rchild==NULL (p是单枝)，令p的左/右子树(非空的)直接成为f的左子树(取代p的位置)，即：f->lchild = p->lchild/p->rchild
- ③ 若p的左右子树均不为空 (p是双枝)
方法1：p的左子树成为f的左子树，p的右子树成为f的左子树的中序遍历的最后一个结点的右子树(该结点一定无右子树)
方法2：以中序遍历中p的直接前驱s为f的左子树p的左子树(除s外)成为s的左子树，s原来的左子树成为s原来父的右子树，p的右子树成为s的右子树

§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

① p是叶子，令f->lchild = NULL

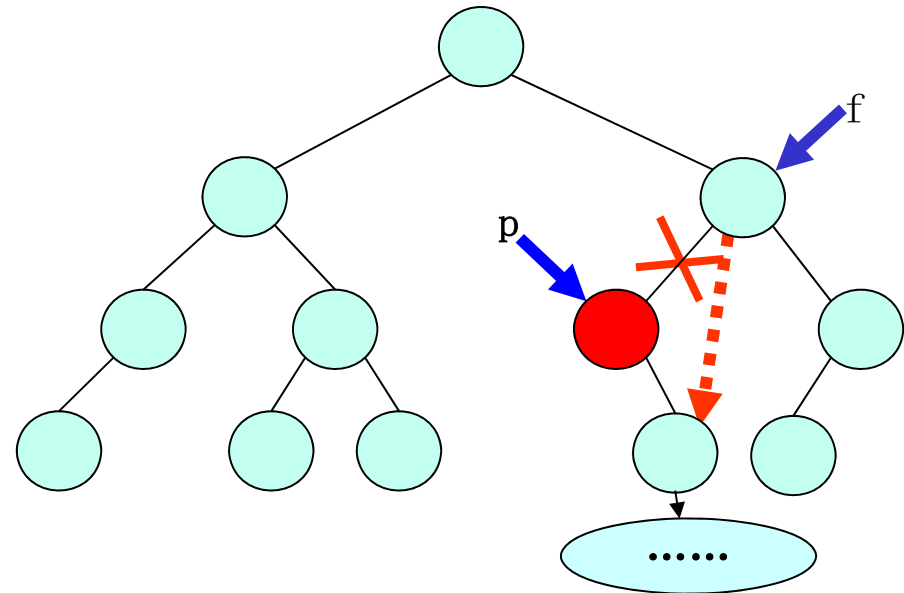
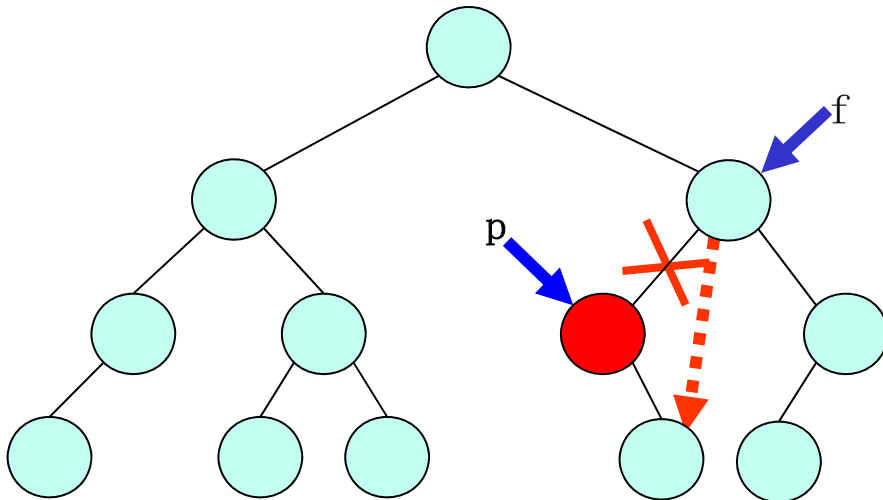
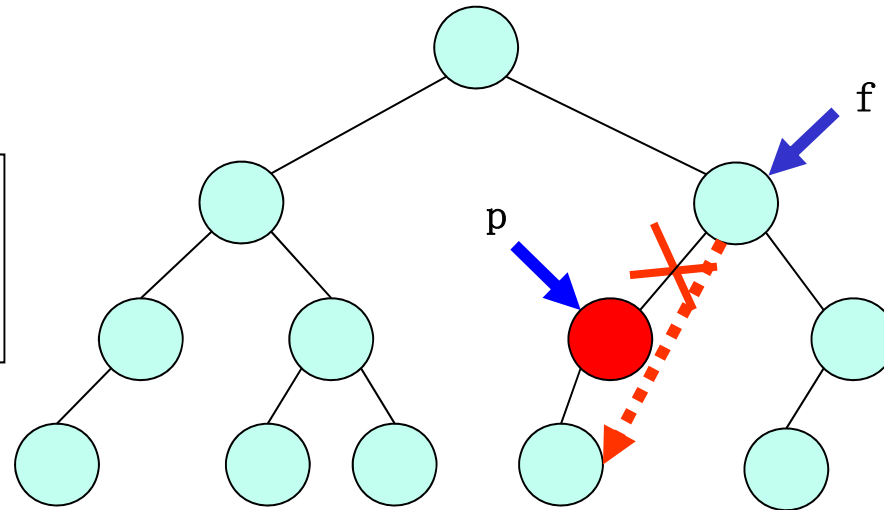


§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

- ② **p是单枝**，令p的左/右子树直接成为f的左子树
 $f \rightarrow lchild = p \rightarrow lchild / p \rightarrow rchild$

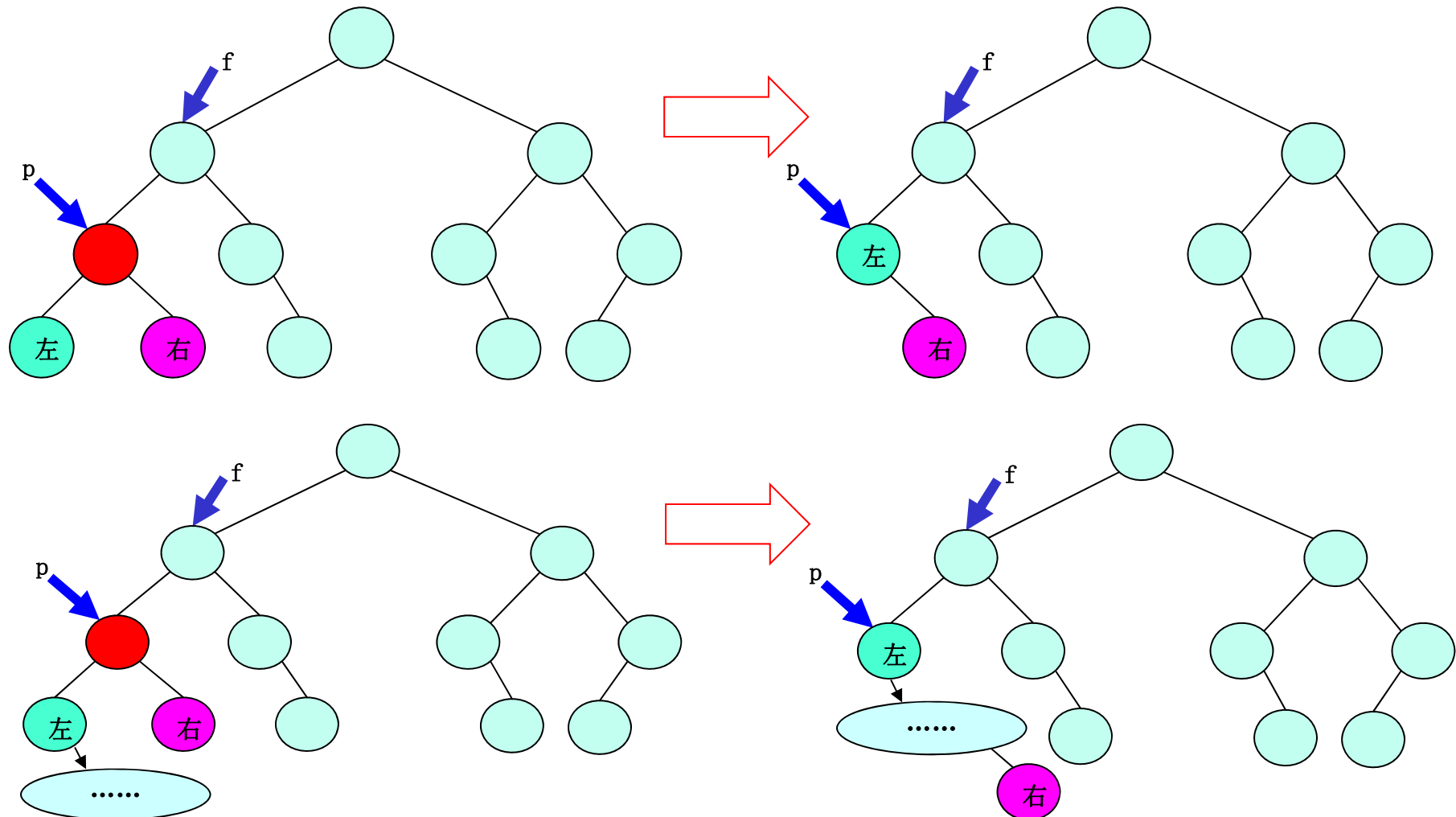


§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

③ **p是双枝** 方法1: p的左子树成为f的左子树, p的右子树成为f的左子树的中序遍历的最后一个结点的右子树

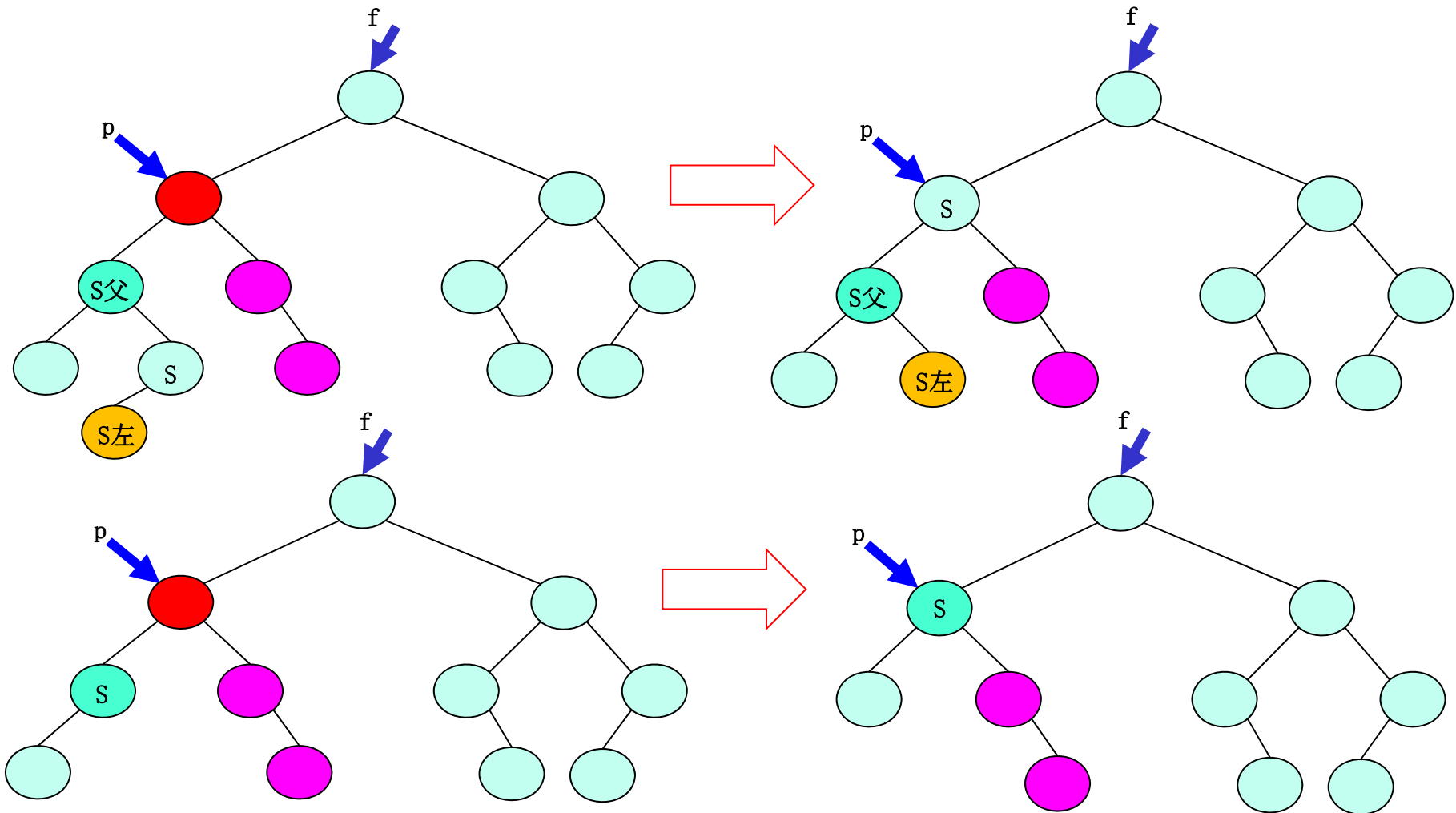


§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

③ **p是双枝** 方法2: 以中序遍历中p的直接前驱s为f的左子树, p的左子树(除s外)成为s的左子树, s原来的左子树成为s原来父的右子树p的右子树成为s的右子树



★ 二叉排序树的删除算法 (P. 230算法9.7)

```

Status DeleteBST(BiTree &T, KeyType key)
{
    if (!T)
        return FALSE; //不存在关键字为key的结点
    else {
        if (EQ(key, T->data.key)) {
            return Delete(T); //在树中删除T结点
        }
        else if (LT(key, T->data.key))
            return DeleteBST(T->lchild, key); //左子树
        else
            return DeleteBST(T->rchild, key); //右子树
    }
}

```

③ p是双枝

方法2: 以中序遍历中p的直接前驱s为f的左子树, p的左子树(除s外)成为s的左子树, s原来的左子树成为s原来父的右子树, p的右子树成为s的右子树

```

Status Delete(BiTree &p)
{
    if (!p->rchild) { //右子树为空 (含左右均为空)
        q = p;
        p = p->lchild; //p的左子树直接取代p (含p=NULL)
        free(q);
    }
    else if (!p->lchild) { //左子树为空 (右子树必非空)
        q = p;
        p = p->rchild; //p的右子树直接取代p
        free(q);
    }
    else { //左右均不为空
        q = p;
        s = p->lchild; //s指向p的左子树,
        while(s->rchild) { //循环向右到底
            q = s; //s是p中序遍历的直接前驱
            s = s->rchild; //q是s的父
        }
        p->data = s->data; //s取代p, p原来左右成为
        //s的左右 (位置未变)

        if (q != p)
            q->rchild = s->lchild; //s原来的左成为
        else //s原来父的右子树
            q->lchild = s->lchild; //q==p, s就是原p的左
        free(s);
    }
    return TRUE;
}

```

§ 9. 查找

9.3. 动态查找表

9.3.2. 二叉排序树

★ 二叉排序树的查找性能分析

n个结点的二叉排序树，最大层次n，最小 $\lfloor \log_2 n \rfloor + 1$

假设等概率查找：

最大层次时： $ASL = \frac{n+1}{2}$ 顺序

最小层次时： $ASL = \log_2(n+1) - 1$ 折半

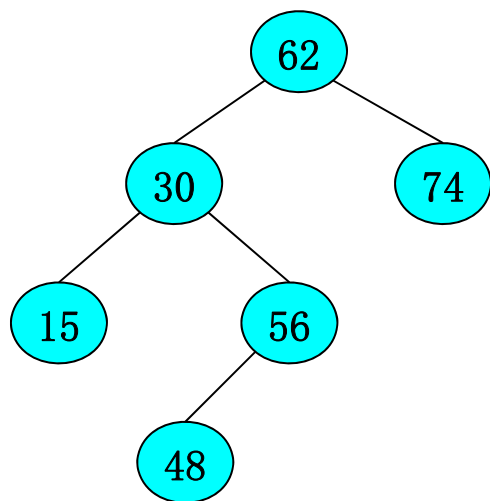
平均： $ASL = 1 + \log_2 n$

P. 232 公式9-18： $P(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i P(i)$

由数学归纳法可以证明 $P(n) < 1 + \log_2 n$ ，具体过程略

例：如图所示二叉排序树

1、求查找成功情况下的平均查找长度



不同元素的查找次数

62 : 1

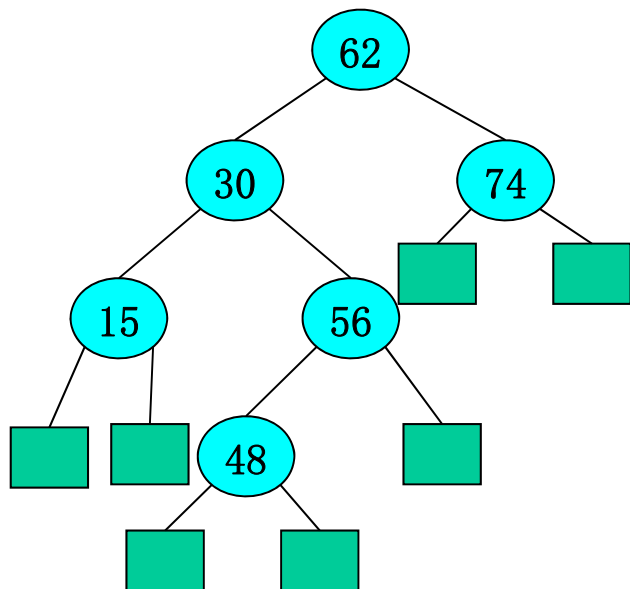
30、74 : 2

15、56 : 3

48 : 4

$$ASL_{\text{succ}} = (1*1 + 2*2 + 3*2 + 4*1) / 6 = 15 / 6 = 2.5$$

2、求查找不成功情况下的平均查找长度



不同范围的查找次数

<15 : 3

16-29 : 3

31-47 : 4

49-55 : 4

57-61 : 3

63-73 : 2

>74 : 2

$$ASL_{\text{unsucc}} = (3 + 3 + 4 + 4 + 3 + 2 + 2) / 7 = 21 / 7 = 3$$

§ 9. 查找

9.3. 动态查找表

9.3.3. 二叉平衡树

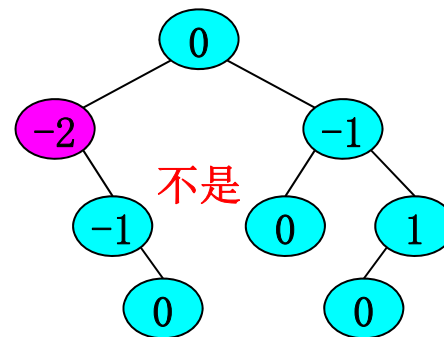
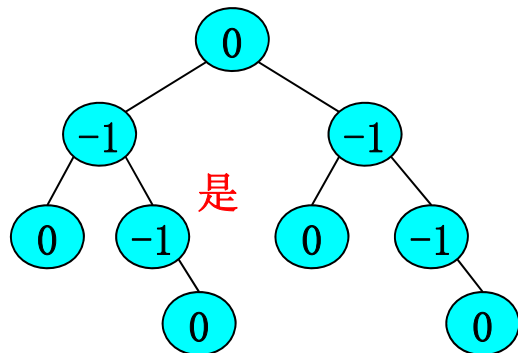
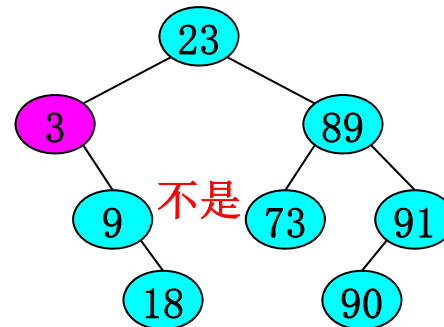
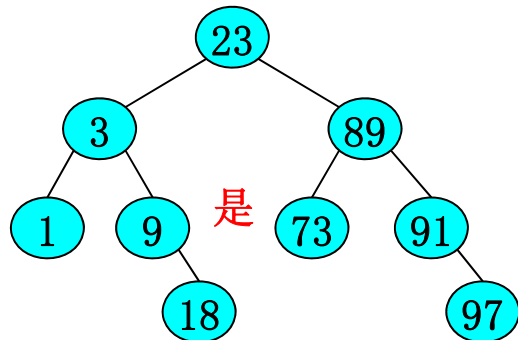
★ 二叉平衡树（AVL树）的递归定义

左右子树深度差的绝对值 ≤ 1

左右子树都是二叉平衡树

● 结点的左子树深度-右子树深度称为结点的平衡因子

=> 二叉平衡树只可能是-1/0/1



§ 9. 查找

9.3. 动态查找表

9.3.3. 二叉平衡树

★ 二叉平衡树 (AVL树) 最小结点数与高度的关系

在一棵高度为h的AVL树中的最少结点数为

$$n_h = F_{h+2} - 1 \quad (h \geq 0) \quad (\text{其中 } F_i \text{ 为 Fibonacci 数列中的第 } i \text{ 项})$$

证明：用数学归纳法，当h=1时，有 $n_1 = F_3 - 1 = 2 - 1 = 1$ 成立

当h=2时，有 $n_2 = F_4 - 1 = 3 - 1 = 2$ 成立

假设h≤k时成立，即 $n_h = F_{h+2} - 1$

假设h=k+1, 对于一个k+1高度的平衡二叉树，其左右子树都是平衡的。在结点数最少的情况下，左右子树的结点个数是不相符的，设其中一个是h=k的平衡二叉树，

另一个是h=k-1的平衡二叉树，由前面假设可知：

$$n_k = F_{k+2} - 1 \quad n_{k-1} = F_{k+1} - 1$$

而一棵平衡二叉树的结点个数是左右子树的结点个数加1，则

$$n_{k+1} = n_k + n_{k-1} + 1 = F_{k+2} - 1 + F_{k+1} - 1 + 1 = F_{k+2} + F_{k+1} - 1$$

由Fibonacci数列特性可知： $F_{k+3} = F_{k+2} + F_{k+1}$

所以： $n_{k+1} = F_{(k+1)+2} - 1$

§ 9. 查找

9.3. 动态查找表

9.3.3. 二叉平衡树

★ 二叉平衡树（AVL树）最小结点数与高度的关系

在一棵高度为h的AVL树中的最少结点数为

$$n_h = F_{h+2} - 1 \quad (h \geq 0) \quad (\text{其中} F_i \text{为Fibonacci数列中的第} i \text{项})$$

★ 含有n个结点的二叉平衡树（AVL树）的最大高度

$$F_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right]$$

$$\text{令 } \phi = \frac{1+\sqrt{5}}{2}, \text{ 则 } F_h \approx \frac{\phi^h}{\sqrt{5}}, \quad n_h \approx \frac{\phi^{h+2}}{\sqrt{5}} - 1$$

$$h = \log_{\phi} (\sqrt{5}(n+1)) - 2 \Rightarrow \text{二叉平衡树查找效率为 } O(\log n)$$

§ 9. 查找

9.3. 动态查找表

9.3.3. 二叉平衡树

★ 二叉平衡树的应用

n个结点，按顺序依次插入，形成二叉排序树，因为插入顺序的不同，形成的二叉排序树形状各异，为提高查找效率，希望二叉排序树高度尽可能小

=> 保持二叉排序树特性的同时，希望是二叉平衡树

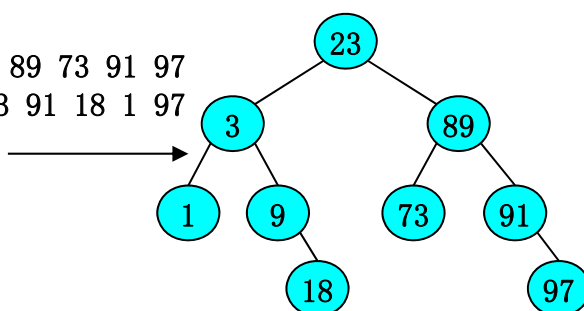
=> 二叉排序树插入过程中，为保持平衡，需要调整

插入序列：

23 3 1 9 18 89 73 91 97

23 3 89 9 73 91 18 1 97

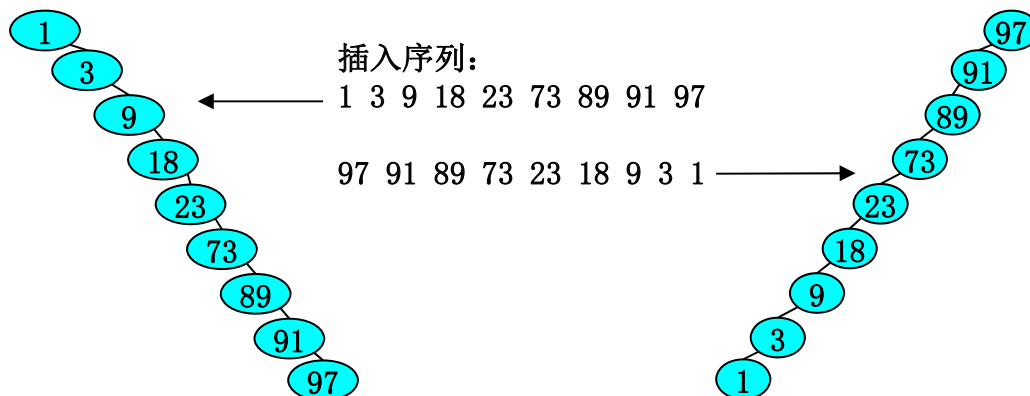
(还有多种)



插入序列：

1 3 9 18 23 73 89 91 97

97 91 89 73 23 18 9 3 1



★ 二叉平衡树的调整规则

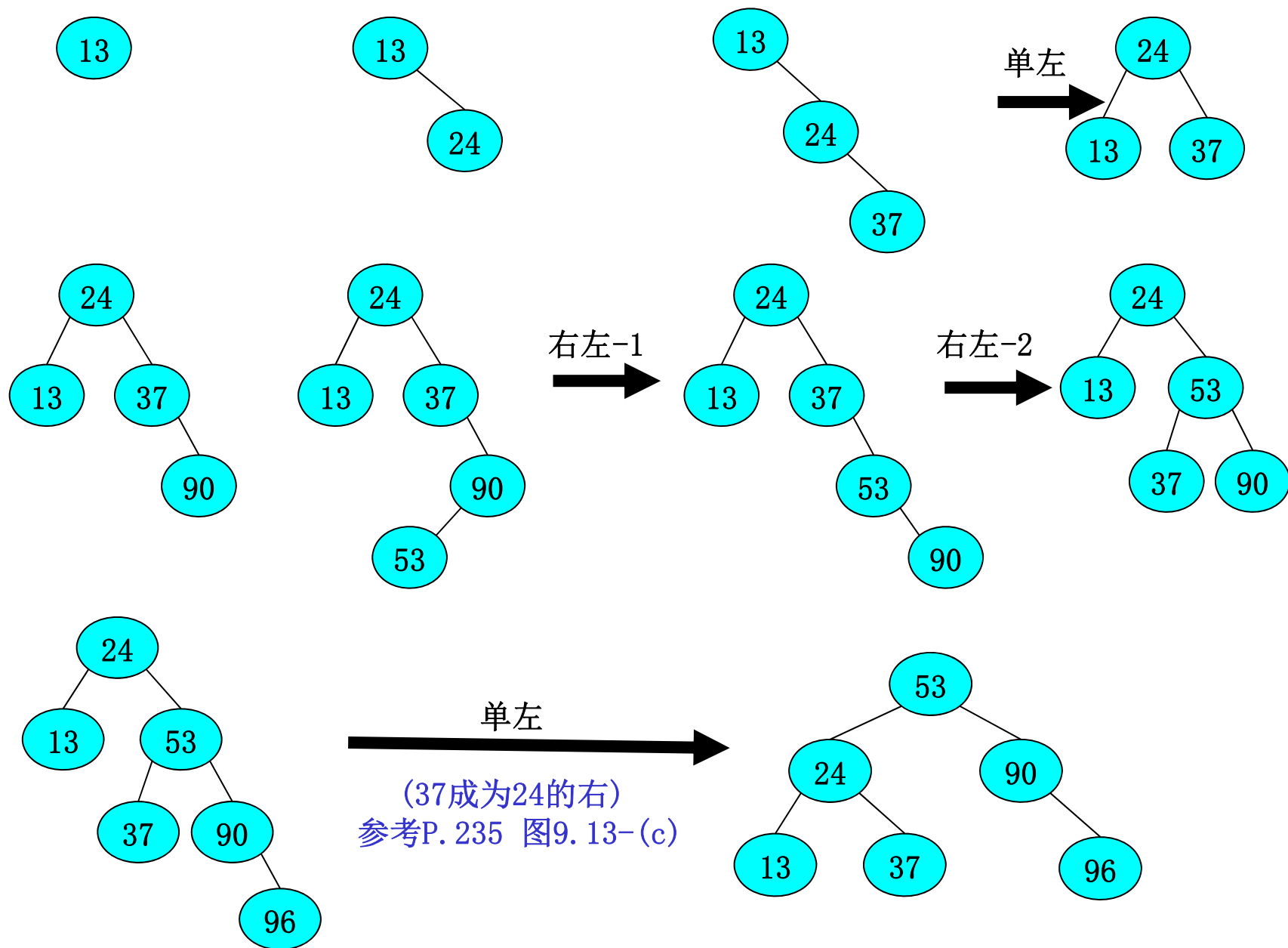
单向右旋：在左子树的左子树插入，平衡因子 $1 \Rightarrow 2$

单向左旋：在右子树的右子树插入，平衡因子 $-1 \Rightarrow -2$

先左后右双向旋转：在左子树的右子树插入，平衡因子 $1 \Rightarrow 2$

先右后左双向旋转：在右子树的左子树插入，平衡因子 $-1 \Rightarrow -2$

例：按照13、24、37、90、53、96的次序形成平衡二叉树



§ 9. 查找

9. 3. 动态查找表

9. 3. 3. 二叉平衡树

★ 二叉平衡树的调整规则

★ 二叉平衡树的插入算法 (略)

9. 3. 4. B树及B+树 (略)

第二学年暑期课程设计时需要自学

9. 3. 5. 键树 (略)

§ 9. 查找

9. 4. 哈希表

9. 4. 1. 基本概念

若以线性表中每个元素的关键字 k 为自变量，通过某个函数 $h(k)$ 计算出函数值，将该值解释为一块连续存储空间(数组)的单元地址(下标)，并将该元素存储到该单元中，则：

- ★ $h(k)$ 称为散列函数(哈希函数)，用于实现关键字与存储地址的映射
- ★ $h(k)$ 的值称为散列地址(哈希地址)
- ★ 使用的数组空间称为散列表(哈希表)

例：关键字 5, 8, 13, 21

哈希函数 $h(k) = k \bmod 9$ 3, 4, 5, 8称为哈希地址

数组为 $a[0-8]$ → 哈希表

则：存储为

0	1	2	3	4	5	6	7	8
			21	13	5			8

- ★ 查找时，由 k 得 $h(k)$ 的值，从数组的 $[h(k)]$ 中取对应元素，不需比较，一次完成(前面的顺序查找、折半查找、二叉排序树、B树等均基于比较)

§ 9. 查找

9. 4. 哈希表

9. 4. 1. 基本概念

★ 冲突：若 $k_1 \neq k_2$, $h(k_1) = h(k_2)$, 则称为冲突

★ 同义词：具有相同函数值的关键字 (k_1, k_2)

例：关键字 5, 8, 17, 23, 哈希函数 $h(k) = k \bmod 9$, 数组为 $a[0-8]$

则：5和23冲突 8和17冲突, 5和23 / 8和17称为同义词

★ 均匀哈希函数：若任一关键字 k , $h(k)$ 的值为存储地址中的任何一个的概率相同, 则称为均匀哈希函数

★ 装填因子：哈希表中已存入元素的个数 n 与表空间大小 m 的比值 $\alpha = n/m$

例：关键字 5, 8, 13, 21, 哈希函数 $h(k) = k \bmod 9$, 数组为 $a[0-8]$

则装填因子 $\alpha = 0.44$ ($4/9$)

α 越小：空闲单元比例越大, 发生冲突的可能性越小

(空间浪费多, 利用率不高)

α 越大：冲突可能性越大(浪费少)

一般 0.6-0.9

§ 9. 查找

9. 4. 哈希表

9. 4. 1. 基本概念

9. 4. 2. 哈希函数的构造方法

★ 直接定址法:

★ 数字分析法:

★ 平方取中法:

★ 折叠法:

★ 除留余数法

$$H(\text{key}) = \text{key} \bmod p \quad (p \leq m, m \text{ 为表长})$$

● p 一般选奇数, 不选偶数(偶数只有一半可用)

● p 一般选素数或不包含小于20的质因数的合数

★ 随机数法

其它方法了解即可

选取哈希函数的原则:

- 1、计算时间
- 2、关键字长度
- 3、哈希表大小
- 4、关键字的分布
- 5、记录的查找频率

§ 9. 查找

9.4. 哈希表

9.4.1. 基本概念

9.4.2. 哈希函数的构造方法

9.4.3. 处理冲突的方法

★ 开放定址法 (常用)

$$H_i = (H(\text{key}) + d_i) \bmod m; \quad (i=1 \sim m-1, m \text{ 为表长})$$

$$d_i \text{ 取值为 } \begin{cases} 1, 2, \dots, m-1 & \text{线性探测法} \\ 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2 & \text{平方探查法} \\ \text{伪随机数序列} & \text{随机探查法} \end{cases}$$

● 处理原则：反复查找至不产生冲突为止

● 冲突过程中，两个哈希值不同的记录争夺同一个后继哈希地址的现象称为二次聚集

★ 再哈希法 (了解即可)

★ 链地址法 (常用)

将哈希地址相同的记录存储在同一线性链表中

★ 建立一个公共溢出区 (了解即可)

例：关键字 5, 8, 17, 23 $h(k)=k \bmod 9$ 数组为 $a[0-8]$

则： $h(5)=5$

$h(8)=8$

$h(17)=8$ 冲突

线性探测法： $h(17)=(8+1) \bmod 9 = 0$

平方探测法： $h(17)=(8+1^2) \bmod 9 = 0$

随机探测法：假设伪随机数序列为 3, 7, 2, ...

$h(17)=(8+3) \bmod 9 = 2$

例：关键字 5, 6, 14, 23 $h(k)=k \bmod 9$ 数组为 $a[0-8]$

则： $h(5)=5$

$h(6)=6$

$h(14)=5$ 冲突

平方探测法： $h(14)=(5+1^2) \bmod 9 = 6$ 冲突

$h(14)=(5-1^2) \bmod 9 = 4$

随机探测法：假设伪随机数序列为 3, 7, 2, ...

$h(14)=(5+3) \bmod 9 = 8$

线性探测法： $h(14)=(5+1) \bmod 9 = 6$ 冲突

$h(14)=(5+2) \bmod 9 = 7$

$h(23)=5$ 冲突

线性探测法： $h(23)=(5+1) \bmod 9 = 6$ 冲突

该冲突称为二次聚集

§ 9. 查找

9. 4. 哈希表

9. 4. 4. 哈希表的查找及其分析

★ 查找的基本过程：计算+比较

找 到：哈希表[H(key)].key = key

找不到：某位置为空/链表为空

★ 影响平均查找长度的因素

哈希函数

冲突策略

装填因子

★ 平均查找长度(随机探查法处理冲突)

$$ASL = -\frac{1}{a}(\ln(1-a)) \quad (\text{P. 262 推导过程})$$

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$$H_i(K) = (H(k) + d_i) \bmod 11 \quad (i=1, 2, \dots) \quad \text{其中} \quad d_1=1, d_{i+1}=(7d_i+3) \bmod 11 \quad (i \geq 1)$$

试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表, 并求等概率情况下查找成功的平均查找长度

$$\{6, 8, 10, 17, 20, 23, 53, 41, 54, 52\}$$
[illegible]

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$H_i(K)=(H(k)+d_i) \bmod 11$ ($i=1, 2, \dots$) 其中 $d_1=1, d_{i+1}=(7d_i+3) \bmod 11$ ($i \geq 1$)

试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表, 并求等概率情况下查找成功的平均查找长度

{6, 8, 10, 17, 20, 23, 53, 41, 54, 52}

下标	0	1	2	3	4	5	6	7	8	9	10
值		6									
次数		1									

$$H(6)=2*6 \bmod 11=1$$

值	下标	比较次数
6	1	1次

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$H_i(K)=(H(k)+d_i) \bmod 11$ ($i=1, 2, \dots$) 其中 $d_1=1, d_{i+1}=(7d_i+3) \bmod 11$ ($i \geq 1$)

试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表, 并求等概率情况下查找成功的平均查找长度

{6, 8, 10, 17, 20, 23, 53, 41, 54, 52}

下标	0	1	2	3	4	5	6	7	8	9	10
值		6				8					
次数		1				1					

$$H(6)=2*6 \bmod 11=1$$

$$H(8)=2*8 \bmod 11=5$$

值 下标 比较次数

6 1 1次

8 5 1次

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$H_i(K)=(H(k)+d_i) \bmod 11$ ($i=1, 2, \dots$) 其中 $d_1=1, d_{i+1}=(7d_i+3) \bmod 11$ ($i \geq 1$)

试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表,
并求等概率情况下查找成功的平均查找长度

{6, 8, 10, 17, 20, 23, 53, 41, 54, 52}

下标	0	1	2	3	4	5	6	7	8	9	10
值		6				8				10	
次数		1				1				1	

$$H(6)=2*6 \bmod 11=1$$

$$H(8)=2*8 \bmod 11=5$$

$$H(10)=2*10 \bmod 11=9$$

值 下标 比较次数

6 1 1次

8 5 1次

10 9 1次

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$H_i(K)=(H(k)+d_i) \bmod 11$ ($i=1, 2, \dots$) 其中 $d_1=1, d_{i+1}=(7d_i+3) \bmod 11$ ($i \geq 1$)

试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表, 并求等概率情况下查找成功的平均查找长度

{6, 8, 10, 17, 20, 23, 53, 41, 54, 52}

下标	0	1	2	3	4	5	6	7	8	9	10
值		6	17			8				10	
次数		1	2			1				1	

$$H(6)=2*6 \bmod 11=1$$

$$H(8)=2*8 \bmod 11=5$$

$$H(10)=2*10 \bmod 11=9$$

$$H(17)=2*17 \bmod 11=1 \text{ (冲突)}$$

$$H_1(17)=(1+d_1) \bmod 11=(1+1) \bmod 11=2$$

值 下标 比较次数

6 1 1次

8 5 1次

10 9 1次

17 2 2次

后续略, 请自行完成

例1: 已知哈希函数 $H(k)=2*k \bmod 11$, 用开放定址法处理冲突:

$H_i(K)=(H(k)+d_i) \bmod 11$ ($i=1, 2, \dots$) 其中 $d_1=1, d_{i+1}=(7d_i+3) \bmod 11$ ($i \geq 1$)

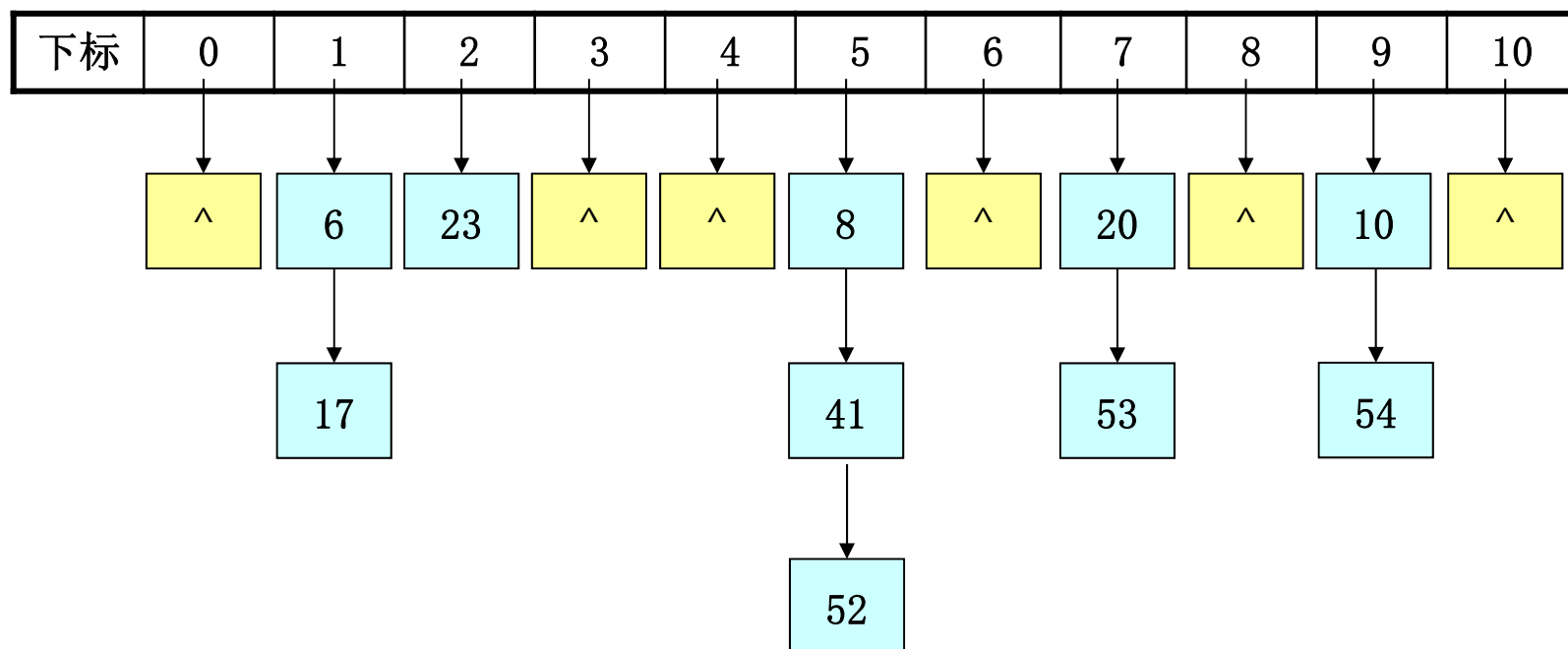
试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表, 并求等概率情况下查找成功的平均查找长度

{6, 8, 10, 17, 20, 23, 53, 41, 54, 52}

下标	0	1	2	3	4	5	6	7	8	9	10
值	/	6	17	23	52	8	41	20	53	10	54
次数	0	1	2	2	3	1	2	1	2	1	2

$$ASL = (4*1 + 5*2 + 1*3) / 10 = 1.7$$

例2：已知哈希函数 $H(k)=2*k \bmod 11$ ，用链地址法处理冲突，试在0~10的哈希地址空间对关键序列 {6, 8, 10, 17, 20, 23, 53, 41, 54, 52} 构造哈希表，并求等概率情况下查找成功的平均查找长度



$$ASL = (5*1 + 4*2 + 1*3) / 10 = 1.6$$