查找

概念

查找表：是由同一类型的数据元素构成的集合
查找：根据某个特定的值，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，若表中存在这样的一个记录，则查找成功，否则查找不成功
关键字：用来标识一个数据元素的数据项，有主关键字（唯一）、次关键字（不一定唯一）

静态/动态查找表：查找过程中表本身发不发生变化

- 静态查找表：顺序、折半、分块
- 动态查找表：各种树表
- 哈希表

查找方法的评价

- 查找速度：占用存储空间的多少，算法本身复杂度
- 平均查找长度ASL（Average Search Length）：为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望对含有n个记录的查找表，查找成功时的平均查找长度

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

其中：p_i为查找表中第i个记录的概率，且$\sum_{i=1}^{n} p_i = 1$
c_i为找到表中第i个元素所需的比较数

FxMarkBrown's Blog

静态查找表

顺序查找表

```
typedef struct {
    KeyType key;
    ...
} ElemType;

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

int Search_Seq(SSTable ST, keyType key) {
    ST.elem[0].key = key; // elem[0] 作为哨兵位置
    for(i = ST.length; !EQ(ST.elem[i].key, key); --i);
    return i;
}
```

性能分析：

1. key在表中，顺序查找时比较次数取决于所查找记录在表中的位置，一般情况下$_i = n\ i\ 1$，在等概率的情况下查找的次数为

$$\sum_{i=1}^{n} n\ i\ 1 = n\ 1$$

2. 若key在表中不存在，则需要进行$n\ 1$次比较（n个元素+哨兵位置自己）
3. 考虑查找成功的概率为p，失败的概率为q=1-p，求ASL

$$ASL = p\ n\ 1\quad n\ 1 = n\ 11\ p$$

讨论：

4. 若记录的查找概率不相等时如果提高查找效率：按查找概率高低存储，概率高的放在后面，概率低的放在前面
5. 若记录的查概率无法测定，则按概率动态调整记录顺序，在每个记录中设一个访问频度域，记录始终按访问频度非递减有次序排列

顺序查找表的特点：

1. 优点：算法简单，对表中元素排列没有要求，实际应用较多，特别是记录少时较有效
2. 缺点：ASL太长，时间效率太低

FxMarkBrown's Blog

有序表的二分查找

设表长为n，low、high和mid分别指向待查元素所在区间的下界、上界和中点，k为给定值

算法：

- 初始时，令low=1，high=n，mid=i
- 让k与mid指向的记录比较
  - 若 $== i$则查找成功
  - 若 i则$i = i\ 1$
  - 若 i则 $= i\ 1$
- 重复上述操作，直至 i时查找失败

```C++
int Search_Bin(SSTable ST, keyType key) {
    low = 1; high = ST.length;
    while(low <= high) {
        mid = (low + high) / 2;
        if (EQ(ST.elem[mid].key, key)) return mid;
        else if (LT(key, ST.elem[mid].key)) high = mid -1;
        else low = mid + 1;
    }

    return 0;
}
```

查找成功时：

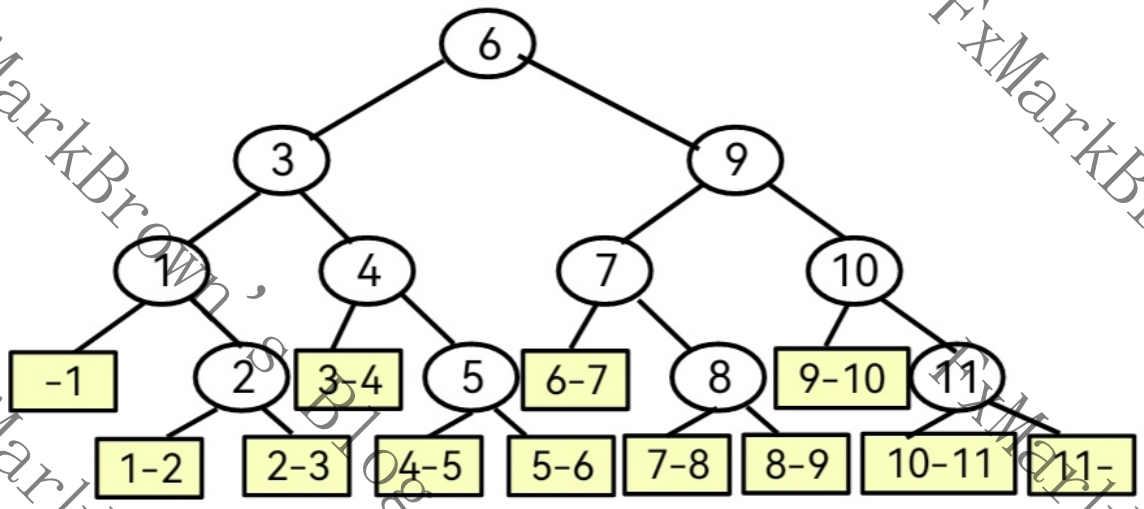
- 比较次数=路径上的结点数=结点的层数
- 比较次数树的深度$=\ n\ 1$

查找不成功时：

- 比较次数=路径上的内部结点数
- 比较次数 $n\ 1$

要会计算查找成功时的ASL，比如

FxMarkBrown's Blog



ASL=1/11*（1*1+2*2+4*3+4*4）=33/11=3

（结点的比较次数=其所在的层数）

若要计算平均查找长度：设表长为$n=1$，则$=n\ 1$（此时，判定树为深度=h的满二叉树），且表中每个记录的查找概率相等：$p_i = 1n$

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

二分查找的特点

- 优点：效率比顺序查找高n
- 缺点：只适用于有序表，且限于顺序存储结构（对线性链表无效）

分块查找

将顺序表分成多块，块内元素任意存放，但块间有序
索引表有序，可以折半查找也可以顺序查找，每一块的查找只能顺序查找

性能分析：

- 分块查找的平均长度：$ASL = L\ L$
  - L为查找索引表确定所在块的平均查找长度
  - L为在块中查找元素的平均查找长度
- 设将长度为n的表均匀分成，每块含有个记录，则$= ciin_n$
- 假定记录的查找概率相等，则每块查找概率为1，即n

FxMarkBrown's Blog

- 若全用顺序查找确定所在块：

$$ASL = L\ \ L = 1 \sum_{=1}\ \ 1 \sum_{i=1} i = 1n\ \ \ 1$$

在给定n的前提下，当 $= n$时，ASL取最小值$n\ \ 1\ \ n$

	顺序查找	二分查找	分块查找
ASL	最大	最小	两者之间
表结构	无要求	有序表	分块有序
存储结构	顺序表、线性链表	顺序表	顺序表、线性链表

动态查找表

表结构本身是在查找过程中动态生成的。即对给定的关键字，若在表中存在则查找成功，否则插入该关键字的记录

排序二叉树

排序二叉树是一棵空树，或是具有下列性质的二叉树

- 若它的左子树不为空，则左子树上所有结点的值均小于根结点的值
- 若它的右子树不为空，则右子树上所有结点的值均大于或等于它的根结点的值
- 左右子树也分别为排序二叉树

性质：排序二叉树中的中序序列为递增有序序列

在二叉排序树中查找关键字key，若根结点的关键字等于给定关键字key，则返回指向根的指针。若根结点的关键字大于key，则在其左子树上查找，否则在其右子树上查找

将元素插入二叉排序树的原则为：

- 若二叉排序树为空，则插入结点应为根结点
- 否则，继续在其左、右子树上查找
  - 树中已有，不在插入
  - 树中没有，查找直至某个结点的左子树或右子树为空为止，则插入结点应为该结点的左孩子或右孩子
- 注意：插入的元素一定在叶子结点上

FxMarkBrown's Blog

```C++
Status SearchBST(BiTree T, keyType key, BiTreee f, BiTree &p) {
    if(!T) { p = f; return FALSE; }
    else if EQ(key, T->data.key) { p=T; return TRUE; }
    else if LT(key, T->data.key) SearchBST(T->lchild, key, T, p);
    else SearchBST(T->rchild, key, T, p);
}

Status Insert_BST(BiTree &T, ElemType e) {
    if(!SearchBST(T, e.key, NULL, p)) {
        s = (BiTree) malloc(sizeof(BiTNode));
        s->data = e; s->lchild = s->rchild = NULL;
        if(!p) T = s;
        else if (LT(e.key, p->data.key)) p->lchild = s;
        else p->rchild = s;
        return TRUE;
    }
    return FALSE;
}
```

对于二叉排序树的删除：
从二叉树中删除一个结点，不能把以该结点为根的子树都删去，只能删掉该结点，并且还应保证删除后所得的二叉树仍然满足二叉排序树的性质不变。
由于中序遍历二叉排序树可以得到一个递增有序的序列。那么，在二叉排序树中删去一个结点相当于删去有序序列中的一个结点
设要删除的结点为P：

1. 若P为叶子结点，直接删除
2. 若P只有左子树或只有右子树，则直接用其左或右子树的根结点代替P
3. 若P结点的左右子树均不为空，则沿P的左子树C的右子树向下找到S，S的右子树为空，将S的左子树变为S的双亲Q的右子树，用S取代P。如果C没有右子树，用C取代P

FxMarkBrown's Blog

```C++
Status DeleteBST(BiTree &T, KeyType key) {
    if(!T) return FALSE;
    else {
        if EQ(key, T->data) Delete(T);
        else if LT(key, T->data.key) DeleteBST(T->lchild, key);
        else DeleteBST(T->rhcild, key);
        return TRUE;
    }
}

void Delete(BiTree &p) {
    if(!p->rchild) { q = p; p = p->lchild; free(q); }
    else if(!p->lchild) { q = p; p -> rchild; free(q); }
    else {
        q = p; s = p->lchild;
        while(s->rchild) { q = s; s = s->rchild; }
        p->data = s->data;
        if(q != p) q->rchild = s->lchild;
        else p->lchild = s->lchild;
        free(s);

    return TRUE;
}
```

查找分析：

- 含有n个结点的二叉排序树的平均查找长度和树的形态有关
- 最好情况：平均查找长度和n成正比，即 n
- 最坏情况：平均查找长度为$n1$，即n
- 平均情况：n

平衡二叉树（AVL树）

定义：平衡二叉树或是一棵空树，或是具有下列性质的二叉排序树

- 左子树与右子树的高度之差的绝对值小于等于
- 它的左右子树也分别称为平衡二叉树
  平衡因子BF：

$$= 结点左子树的高度\ 结点右子树的高度$$

  根据平衡二叉树的定义，平衡二叉树上所有结点的平衡因子只能是-1，0，1

一般情况下，若插入一个叶子结点，相应子树的根结点变化大致有三种情况

1. 结点原来平衡，现在变成左重或右重

FxMarkBrown's Blog

2. 结点原来某一边重，而现在成为平衡的
3. 结点原来就是左重或右重，而新结点加在重的一边

检查调整是否正确的重要判据：中序序列不变

对于调整的情况：

1. 右重插入到右边的右子树中，进行单向左旋平衡处理，左重插入到左边的左子树中，进行单向右旋平衡处理
2. 右重插入到右边的左子树中，右侧左子树的根右旋上去后再左旋。左重插入到左边的右子树中...

B-树和B+树

一棵m阶的B-树，或为空树，或为满足下列特性的m叉树：

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，则至少有两颗子树
3. 除根之外的所有非终端结点至少有ci棵子树
4. 所有非终端结点中包含下列信息数据
   $n\,A_{\,1}\,A_1\;\;A_{\;n}\,A_n$
   其中$_i i=1\;\;n$为关键字，且$_i\;\;_{i1} i=1\;\;n\;\;1\,A_i i=\;\;$为指向子树根结点的指针，且指针$A_{i1}$所指子树中所有结点的关键字均小于$_i i=1\;\;n\,A_n$所指子树中所有结点的关键字均小于$_i i=1\;\;n$，$A_n$所指子树中所有结点的关键字均大于$_n$，$nci\;\;1\;\;n\;\;\;1$为关键字的个数（或$n\;\;1$为子树个数）

哈希表

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法

哈希函数可写成：$_i=\;_i$，$_i$是表中的一个元素，$_i$是$_i$的存储地址

根据设定的哈希函数和处理冲突的方法将一组关键字映像到一个有限的连续地址集（区间）上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表就叫做**哈希表**，这一映像过程称为**哈希造表或散列**，所得存储位置称**哈希地址或散列地址**，使用哈希表要解决好两个问题：

1. 构造好的哈希函数：所选函数要尽可能简单，以便提高转换速度；所选函数对关键字计算出的地址，应在哈希地址集中均匀分布，以减少空间浪费
2. 制定好的解决冲突的方案：查找时，如果从哈希函数计算出的地址中查不到关键字，应依据解决冲突的规则，有规律地查询其他相关单元

第一个问题的解决方案有：

1. 直接定址法：取关键字或关键字的某个线性函数作为哈西地址

2. 数字分析法：对关键字进行分析，取关键字的若干位或其组合作为哈希地址
3. 平方取中法：取关键字平方后中间几位作哈希地址，适于不知道全部关键字情况
4. 折叠法：将关键字分割为位数相同的几部分，然后取这几部分的叠加和作哈希地址
5. 除留余数法：取关键字被某个不大于哈希表表长的数p除后所得余数作哈希地址，即 $=$ pp

第二个问题的解决方案有：

1. 开放地址法：有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。常用方法有：线性探测法、二次探测法、伪随机探测法
    1. 线性探测法：例如在除留余数法时$_i = \ _i \ 1 \ i$
    $_i$为1 $\ 1$
    2. 二次探测法：$_i$为1 1 的二次序列
    3. 伪随机探测法：$_i$为伪随机数序列
2. $_i = i = 1 \ _{ri}$是不同的哈希函数，在同义词产生地址冲突时计算另一个函数地址，直到冲突不再发生
3. 链地址法：相同哈希地址的记录链接成以单链表

哈希表的查找：
给定查找值：

- 根据造表时的哈希函数求哈希地址
- 若表中这个位置没有记录，则查找不成功
- 否则比较关键字，若和给定值相等，则查找成功
- 否则根据设定的处理冲突的方法找“下一地址”，直到哈希表中某个位置为空或者表中关键字等于给定值为止

仍需以ASL作为衡量哈希表查找效率的量度，ASL取决于

- 哈希函数
- 处理冲突的方法
- 哈希表的装填因子

$$\Rightarrow \text{表中填入的记录数哈希表的长度}$$

越大，表中记录越多，说明表装的越满，发生冲突的可能性就越大，查找时比较次数就越多。实际常取0.65-0.9
ASL与有关，既不是严格的1，也不是n
链地址法：$ASL \ 1$
线性探测法：$ASL \ _{11}1 \ _{11}$
随机探测法：$ASL \ _{11}1$

FxMarkBrown's Blog