查找

查找表可分为两类:

- 静态查找表

  仅作查询和检索操作的查找表。

- 动态查找表
  有时在查询之后，还需要将“查询”结果为“不在查找表中”的数据元素插入到查找表中;

  或者，从查找表中删除其“查询结果为“在查找表中”的数据元素

一、静态查找表

数据结构定义:

```c
typedef struct {
    KeyType key;     // 关键字（用于标识元素）
    // ... 其他数据域（根据需求添加）
} ElemType;

typedef struct {
    ElemType *elem; // 数据存储基址（0号单元留空）
    int length;     // 表长度
} SSTable;
```

---

1. 顺序查找

算法思想:
从**表尾向表头**逐个比较元素关键字，利用`elem[0]`作为哨兵简化边界判断。

时间复杂度:

- 成功：$O(n)$
- 失败：$O(n)$

C语言实现:

```c
int Search_Seq(SSTable ST, KeyType key) {
    ST.elem[0].key = key; // 哨兵（简化越界判断）
    int i = ST.length;    // 从表尾开始查找
    // 从后往前扫描，直到找到关键字或遇到哨兵
    while (ST.elem[i].key != key) i--;
    return i; // 返回位置（0表示未找到）
}
```

---

2. 有序查找（折半查找）

算法思想：
仅适用于**有序表**。每次将待查区间缩小一半：

- 若 `key == mid.key`，找到元素
- 若 `key < mid.key`，在左半区继续查找
- 若 `key > mid.key`，在右半区继续查找

时间复杂度：$O(\log n)$

C语言实现：

```c
int Search_Bin(SSTable ST, KeyType key) {
    int low = 1, high = ST.length; // 初始化查找区间
    while (low <= high) {
        int mid = (low + high) / 2;  // 计算中间位置
        if (key == ST.elem[mid].key)
            return mid;              // 找到元素
        else if (key < ST.elem[mid].key)
            high = mid - 1;          // 在左半区继续查找
        else
            low = mid + 1;           // 在右半区继续查找
    }
    return 0; // 未找到
}
```

例题：查找 `key=64`

```
初始：low=1, high=11 → mid=6 (ST.elem[6]=60)
64>60 → low=7, high=11 → mid=9 (ST.elem[9]=88)
64<88 → high=8, low=7 → mid=7 (ST.elem[7]=64) 找到!
```

---

3. 静态查找树（次优查找树）

算法思想：
针对**查找概率不等**的有序表，构造一棵二叉查找树，使得查找代价最小化。选择根结点使得左右子树权值累计差最小。

```
关键字:          A      B        C        D        E
  Pi:          0.2    0.3      0.05     0.3      0.15
  Ci:           2      3        1        2        3
此时 ASL=2+0.2+3+0.3+1+0.05+2+0.3+3+0.15=2.4
 若改变Ci的值     2      1        3        2        3
则 ASL=2+0.2+1+0.3+3+0.05+2+0.3+3+0.15=1.9
```

数据结构：

```c
typedef struct BiTNode {
    ElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

构造步骤：

1. 计算累计权值和 `sw`
2. 选择 $\Delta P_i = |sw_h + sw_{l-1} - sw_{i-1} - sw_i|$ 最小的结点作为根
3. 递归构建左右子树

C语言实现：

```c
void SecondOptimal(BiTree *T, ElemType R[], float sw[], int low, int high) {
    if (low > high) return;
    int min_delta = INT_MAX, i_min = low; // 初始化最小权值差和对应的根节点下标
    // min_delta 用来记录找到的最小的左右子树权值差,初始设为最大的整数值，以便后续比较
    // i_min 记录找到的使 delta 最小的关键字的下标

    // 1. 在此区内寻找ΔPi最小的位置i_min（递归后依旧可见）
    for (int i = low; i <= high; i++) {
        float delta = fabs( (sw[high] - sw[i]) - (sw[i-1] - sw[low-1]) );
        if (delta < min_delta) { // 如果当前的 delta 比已知的最小 delta 还要小
            min_delta = delta; // 更新最小 delta
            i_min = i; // 记录当前使 delta 最小的根节点下标
        }
    }
    // 2. 创建根结点
    *T = (BiTree)malloc(sizeof(BiTNode));
    (*T)->data = R[i_min];//将关键字 R[i_min] 赋值给新创建的根节点的数据域
    // 3. 递归构建左右子树
    SecondOptimal(&((*T)->lchild), R, sw, low, i_min-1);
    SecondOptimal(&((*T)->rchild), R, sw, i_min+1, high);
}
```

二、动态查找表

1. 二叉排序树（BST）

定义：

- 左子树所有结点值 < 根结点值
- 右子树所有结点值 > 根结点值
- 左右子树也是二叉排序树

核心操作：

(1) 查找

```c
/**
 * @brief 在二叉查找树中查找指定关键字。
 * @param T 当前子树的根节点指针。在递归调用中，T 会逐级指向左子树或右子树。
 * @param f 指向 T 的父节点的指针。在递归查找未成功时，它将指向查找路径上最后一个被访问的节点。
 * @param p 指向指针的指针，用于返回查找结果。
 * - 如果找到关键字，*p 将指向包含该关键字的节点。
 * - 如果未找到关键字，*p 将指向查找路径上最后一个被访问的节点 (即最终可能插入新节点的位置)。
 */

```

```c
//递归查找方法
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree *p) {
    // 递归终止条件 / 查找失败情况：当前子树 T 为空（即到达了叶子节点的下方，或者初始传入的就是空树）
    if (!T) {
        // 未找到目标关键字。
        // 此时，*p 应该指向查找路径上最后访问的那个非空节点 f。
        // 这个 f 就是新节点如果需要插入的话，它的父节点。
        *p = f;
        return FALSE; // 返回 FALSE，表示查找未成功
    }

    // 查找成功情况：
    // 如果当前节点 T 的关键字等于要查找的目标关键字 key
    if (key == T->data.key) {
        // 找到了目标关键字。
        *p = T; // 将 *p 指向当前找到的节点 T。
        return TRUE; // 返回 TRUE，表示查找成功
    }

    // 递归查找情况：
    // 如果要查找的关键字 key 小于当前节点 T 的关键字
    if (key < T->data.key)
        // 按照二叉查找树的性质，目标关键字一定在左子树中。
        // 当前节点 T 成为下一次递归调用中的父节点 f。
        return SearchBST(T->lchild, key, T, p);
    else
        return SearchBST(T->rchild, key, T, p);
}
```

(2) 插入

```c
Status InsertBST(BiTree *T, ElemType e) {
    BiTree p;// 声明一个 BiTree 指针 p，用于接收 SearchBST 的查找结果。
             // p 将指向要插入新节点位置的父节点，或者已存在的节点。
    if ( SearchBST(*T, e.key, NULL, &p) )
        return FALSE; // 已存在，不插入
    BiTree s = (BiTree)malloc(sizeof(BiTNode));
    s->data = e;
    s->lchild = s->rchild = NULL;//对新节点赋值初始化

    if (!p)
        *T = s;           // a. 如果 p 为 NULL，说明原始树是空的 (*T 为 NULL)，新节点 s 成为树的根
    else if (e.key < p->data.key)
        p->lchild = s;        // 插入为左孩子
    else
        p->rchild = s;        // 插入为右孩子
    return TRUE;
    //一定是插入新的节点中，可以自己出几个题目
}
```

(3) 删除（三种情况）

```c
void Delete(BiTree *p) {
    BiTree q, s; // 声明辅助指针 q 和 s

    // 情况一：被删除节点 *p 没有右子树 (包括 *p 是叶子节点的情况，此时左右子树都为空)
    // 这种情况下，直接用 *p 的左子树来替代 *p 的位置
    if (!(*p)->rchild) {      // 右子树空 → 重接左子树
        q = *p; *p = (*p)->lchild; free(q);
    // 情况二：被删除节点 *p 没有左子树 (但有右子树)
    // 这种情况下，直接用 *p 的右子树来替代 *p 的位置
    } else if (!(*p)->lchild) { // 左子树空 → 重接右子树
        q = *p; *p = (*p)->rchild; free(q);
    } else {                  // 左右均不空
        // 情况三：被删除节点 *p 左右子树均不为空
    // 这是最复杂的情况。通常有两种策略：
    //   1. 找到 *p 的直接前驱 (左子树中最大的节点)。
    //   2. 找到 *p 的直接后继 (右子树中最小的节点)。
        // 这里采用的是找直接前驱的策略。
        q = *p; // q 最初指向被删除节点 *p，用于追踪 s 的父节点
        s = (*p)->lchild;   // s 最初指向被删除节点 *p 的左孩子 (开始寻找直接前驱)
        while (s->rchild) {
            q = s; // q 总是 s 的父节点 (或者最初是被删除节点本身)
            s = s->rchild;
        } // 找直接前驱s-左子树中最大的节点
        (*p)->data = s->data; // s的值 替换被删结点
                //  重接 s 的父节点的子树 (删除 s 节点本身)
        // 被删除节点 *p 的数据已经被 s 的数据替换，现在我们只需要删除 s 节点。
        // s 节点位于被删除节点的左子树中。
        // 因为 s 是左子树中最大的，所以它不会有右孩子，但可能有左孩子。如果有，需要被重接到 s 的父节点 q 的右孩子位置。

        // 判断 q 和 *p 是否是同一个节点
        // 如果 q == *p，说明 s 是 *p 的直接左孩子 (即 *p 的左子树没有右孩子，s就是其左孩子)
        // 例如：被删除节点 A，左孩子 B，B没有右孩子，那么B就是A的直接前驱。
        // 此时 q=A，s=B。
        if (q != *p)
            q->rchild = s->lchild; // 重接q的右子树
        else
            q->lchild = s->lchild; // 重接q的左子树
        free(s);
    }
}
```

性能分析：

- 最好：树平衡 $\rightarrow O(\log n)$
- 最坏：树退化成链 $\rightarrow O(n)$

---

2. 平衡二叉树（AVL树）

定义：任意结点左右子树高度差绝对值 ≤1。

平衡调整（四种旋转）：

失衡类型	旋转操作
LL型（左左）	右旋
RR型（右右）	左旋
LR型（左右）	先左旋再右旋
RL型（右左）	先右旋再左旋

右旋操作代码：

```
void R_Rotate(BiTree *p) {
    BiTree lc = (*p)->lchild;   // lc指向p的左孩子
    (*p)->lchild = lc->rchild; // lc的右子树挂到p的左子树
    lc->rchild = *p;            // p成为lc的右孩子
    *p = lc;                    // lc成为新根
}
```

左旋操作代码：

```
void L_Rotate(BiTree *p) {
    BiTree rc = (*p)->rchild;   // rc指向p的右孩子
    (*p)->rchild = rc->lchild;  // rc的左子树挂到p的右子树
    rc->lchild = *p;            // p成为rc的左孩子
    *p = rc;                    // rc成为新根
}
```

性能分析：
含 n 个结点的AVL树最大深度 $h \approx 1.44\log_2(n+1)$，查找时间复杂度 $O(\log n)$。

三、关键概念总结

1. **平均查找长度（ASL）**：
   $ASL = \sum_{i=1}^{n} P_i \times C_i$
   P_i：查找第 i 个元素的概率，C_i：找到需比较的次数。
2. **静态 vs 动态查找表**：
   - 静态：仅查询（顺序表、有序表、静态树）
   - 动态：支持增删查（BST、AVL树、哈希表）
3. **核心算法选择**：

场景	推荐算法
无序静态表	顺序查找
有序静态表	折半查找

场景	推荐算法
频繁动态增删	二叉平衡树
查找概率差异大且静态	次优查找树

> 代码均用标准C语言实现，可直接运行。建议结合绘图理解树结构操作（如旋转），变量命名与注释已优化可读性。