查找

查找(Search)的基本概念

有关阅读：数据结构：查找(Search)【详解】

- 查找表——由同一类型的数据元素或记录构成的集合。由于“集合”中的数据元素之间存在着**完全松散**的关系，因此查找表是一种非常**灵便的数据结构**。
- 静态查找表——查找时**不对查找表做任何修改**。
- 动态查找表——查找时对查找表具有修改操作。
- 关键字——记录中某个数据项的值，可用来识别一个记录。
- 主关键字——唯一标识数据元素。
- 次关键字——可以标识若干个数据元素。
- 平均查找长度——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(Average Search Length, ASL)。是**查找算法的评价指标**。

关键字的平均比较次数，也称平均查找长度*ASL(Average Search Length)*

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

n：记录的个数
pi：查找第i个记录的概率（通常认为pi =1/n）
ci：找到第i个记录所需的比较次数

线性表的查找

顺序查找（线性查找）

应用范围：顺序表或线性链表表示的静态查找表；表内元素之间无序。

顺序表的表示：

```
typedef struct{
    ElemType *R;//表基址
    int length;//表长
}SSTable;
```

查找顺序表L中值为e的数据元素。

```
int LocateElem(SqList L.ElemType e){
    for(i=0;i<L.length;i++){
        if(L.elem[i]==e)
            return i+1;
    }
    return 0;
}
```

把待查关键字key存入表头（"哨兵"），从后向前逐个比较，可免去查找过程中每一步都要检查是否查找完毕，加快速度。

```
int Search_Seq(SSTable ST,KeyType key){
    //若成功返回其位置信息，否则返回0
    ST.R[0].key=key;
    for(i=ST.length;ST.R[i].key≠key;--i);//若表中元素不等于关键字则持续循环，等于时跳出循环，返回i的值。
    return i;
}
```

```
//另一种形式
/*有哨兵顺序查找*/
int Sequential_Search(int *a, int n, int key){
    int i;
    a[0] = key; //设置a[0]为关键字，称之为“哨兵”
    i = n;  //循环从数组尾部开始
    while(a[i] != key){
        i--;
    }
    return i;    //返回0则说明查找失败
}
```

- 空间复杂度：一个辅助空间；
- 时间复杂度：
  - 查找成功：ASLs(n)=(1+2+...+n)/n=**(n+1)/2;**
  - 查找不成功：`ASLf` =n+1;

顺序查找算法特点

- 算法简单，对表结构无任何要求。
- n很大时，查找效率低，不宜采用顺序查找。

▶▶ 课堂练习:判断对错

例1： n个数存在一维数组A[1..n]中，在进行顺序查找时，这n个数的排列有序或无序其平均查找长度ASL不同。（✗）

查找概率相等时，ASL相同；
查找概率不等时，如果从前向后查找，则按查找概率由大到小排列的有序表其ASL要比无序表ASL小。

折半查找(二分或对分查找)

> 查找表为从小到大排列的有序表时，进行如下操作折半查找。

设表长为n，low=1，high=n，mid=（low+high）/2

若k==R[mid].key , return true;

若k<R.[mid].key , 则high = mid-1;

若k>R.[mid].key , 则low = mid+1;

若low>high ， 则查找失败，表中没有关键字。

折半查找算法

```
//循环折半查找算法
int Search_Bin(SSTable ST,KeyType key){
    low=1;
    high=ST.length;
    while(low≤high){
        mid=(low+high)/2;
```

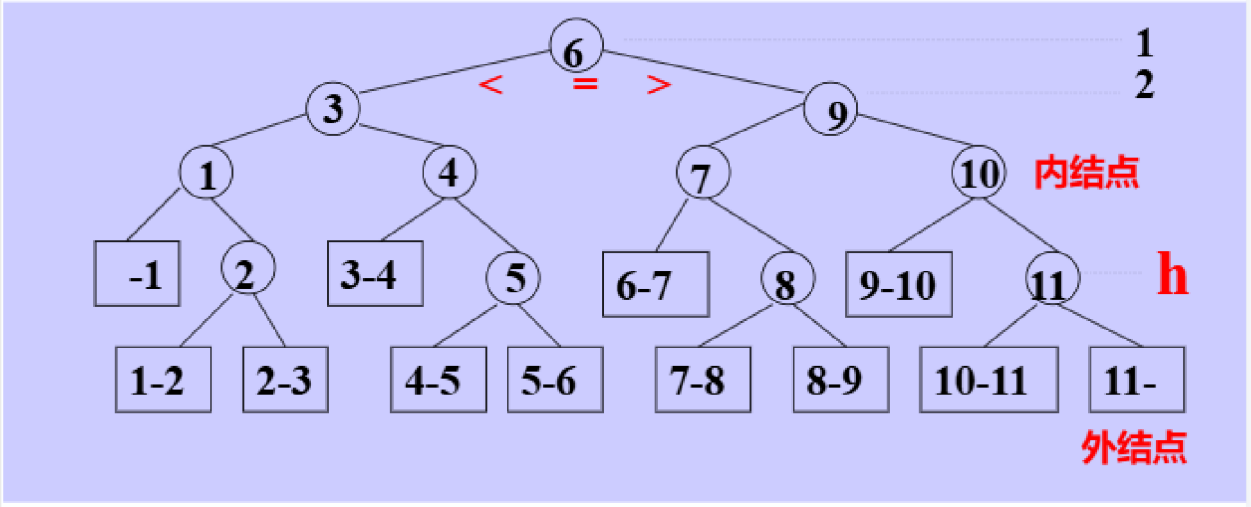
```
        if(key==ST.R[mid].key)
            return mid;
        else if(key<ST.R[mid].key)
            high = mid -1;
        else
            low = mid+1;
    }
    return 0;
}
```

```
//递归折半查找算法
//调用函数时输入初始的最大值和最小值
int Search_Bin(SSTable ST,KeyType key,int low,int high){
    if(low>high)
        return 0;
    mid = (low+high)/2;
    if(key==ST.R[mid].key)
        return mid;
    else if(key<ST.R[mid].key)
        return Search_Bin(ST,key,low,high);
    else
        return Search_Bin(ST,key,low,high);
}
```

> 查找时间复杂度：ASL= O（$log_2 n$）
>
> 折半查找适用条件：采用顺序存储结构的有序表。

折半查找的性能分析 - 判定树

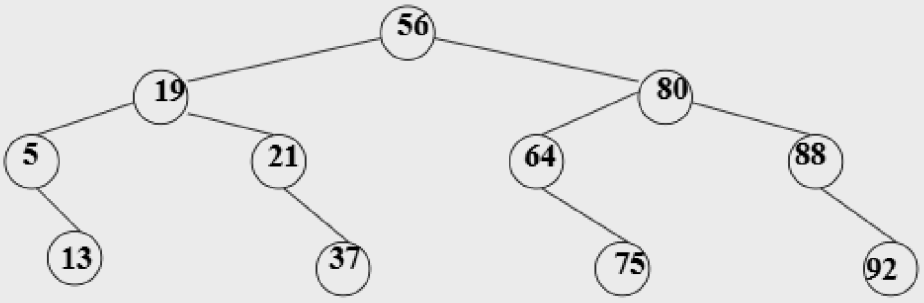
1	2	3	4	5	6	7	8	9	10	11
5	13	19	21	37	56	64	75	80	88	92



若所有结点的空指针域设置为一个指向一个方形结点的指针，称方形结点为判定树的外部结点；对应的，圆形结点为内部结点。

例2： 假定有序表（5,13,19,21,37,56,64,75,80,88,92）每个元素的查找概率相等，求查找成功时的平均查找长度。

解析：折半查找判定树如下：



则查找成功时的平均查找长度为：

$ASL = 1/11*(1*1 + 2\times2 + 4\times3+4*4) = 33/11=3$

分块查找（块间有序，块内无序）

> 即分为若干子表，每一子表的数值都比后一块中的数值小（子表块内部未必有序）。

> 将各子表中的**最大关键字**构成一个**索引表**，表中包含每个**子表的起始地址**。
> 对索引表**折半查找**。
> 确定关键字所在子表后，在子表内**顺序查找**。

查找效率：ASL=对索引表查找的ASL + 对块内查找的ASL

$ASL_{bs} = log_2(n/s+1) + s/2 \ \ (log_2 n =< ASL_{bs} =< (n+1)/2)$

s为块内部的记录个数，n/s即块的数目。

分块查找的特点

优点：**插入和删除**比较容易，**无需进行大量移动**。

缺点：要增加一个**索引表的存储空间**，并对**初始索引表进行排序**运算。

适用情况：线性表既要快速查找又要经常动态变化。

树表的查找

> 表结构在查找过程中动态生成。
> 对于给定值key，若表中存在则成功返回；
> 否则，插入关键字等于key的记录。

二叉排序树

二叉排序树或是空树，或者：

- 若其**左子树非空**，则**左子树上所有结点**的值均**小于根结点**的值。
- 若其**右子树非空**，则**右子树上所有结点**的值均**大于根结点**的值。
- 其左右子树本身又各是一颗二叉树。

> 中序遍历二叉排序树，得到一个关键字的增减有序序列。

二叉排序树查找

类似折半查找，二叉排序树为空，查找失败，返回0。

查找的关键字等于根结点，则返回根结点；小于根结点，则查找其左子树；大于根结点，则查找其右子树。

```
BBTree SearchBST(BSTree T,KeyType key){
    if(!T||Key == T->data.key)
        return T;
    else if(key < T->data.key)
        return SearchBST(T->lchild.key);
    else
        return SearchBST(T->rchild.key);
}
```

时间复杂度：$O(log_2n)$

二叉排序树的插入

> 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，不再插入。
> - 树中没有，才插入。新插入节点一定是一个新添加的叶子结点。并且是查找不成功时查找路径上访问的最后一个左孩子或右孩子结点。

```
void InsertBST(BSTree &T,ElemType e){
    if(!T){
        S= new BSTNode;
        S→data=e;
        S→lchild = s→rchild =NULL;
        T=S;
    }
    else if(e.key < T→data.key)
        InsertBST(T→lchild,e);
    else if(e.key > T→data.key)
        InsertBST(T→rchild,e);
}
```

时间复杂度：$O(log_2n)$

二叉排序树的创建

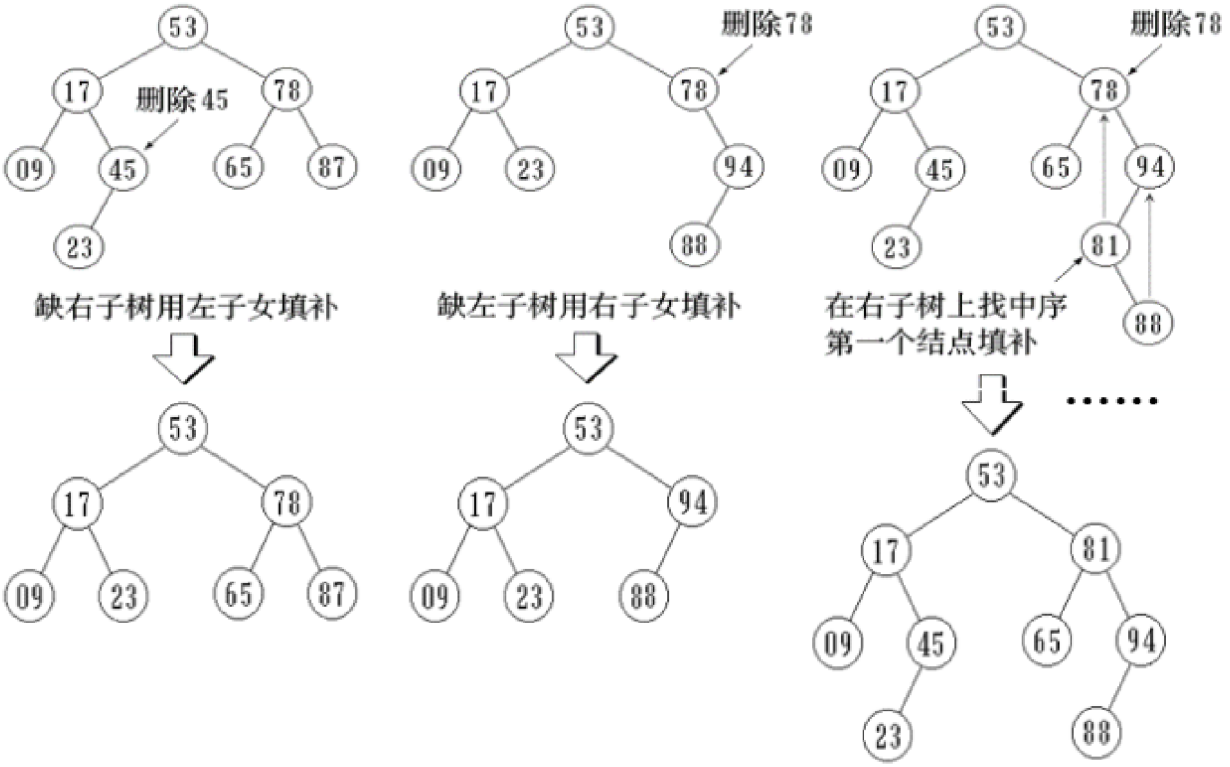
```
//二叉排序树的创建算法描述
void CreatBST(BSTree &T){
    //依次读入一个关键字为key的系欵但，将此节点插入二叉树。
    T=NULL;//初始化二叉树为空树
    cin>>e;
    while(e.key≠ENDFLAG){//ENDFLAG为自定义常量作为输入结束标志。
        InsertBST(T,e);//将此结点插入二叉排序树中
        cin>>e;
    }
}
```

时间复杂度：$O(nlog_2n)$

二叉排序树的删除

> 将因删除结点而断开的二叉链表重新链接起来，防止重新链接后树的高度增加，并且保证仍然是二叉排序树。
>
> $O(log_2n)$

二叉排序树的删除



二叉排序树的删除

删除叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。

被删结点缺右子树，可以拿它的左子女结点顶替它的位置，再释放它。

被删结点缺左子树，可以拿它的右子女结点顶替它的位置，再释放它。

被删结点左、右子树都存在，可以在它的右子树中寻找中序下的第一个结点(关键码最小),用它的值填补到被删结点中，再来处理这个结点的删除问题。

二叉排序树查找的性能分析

计算平均查找长度ASL。

最好：log_2n（形态均称，与二分查找的判定树相似）

最坏：(n+1)/2（单支树）

平均查找长度：O(log_2n)

> 如何提高二叉排序树的查找效率?
>
> 尽可能使二叉排序树的形状均衡。（平衡二叉树）

平衡二叉树（AVL）：

- 空树。
- 或者，左、右子树的深度只差的绝对值=<1。且左、右子树是平衡二叉树。

平衡因子：

该结点左、右子树的高度差。只能取-1、0、1。绝对值大于1则失去平衡，非AVL。

一棵有n各结点的AVL树，其高度保持在O(log_2n)数量级，ASL也保持在O(log_2n)量级。

平衡旋转：

一棵因插入新结点而失去平衡的AVL树，重新调整树的结构，使之恢复平衡。

1. LL平衡旋转 ： A的左子树的左子树插入新结点，顺时针旋转。
2. RR平衡旋转：A的右子树的右子树插入新结点，逆时针旋转。
3. LR平衡旋转：A的左子树的右子树插入新结点，先逆时针旋转，再逆时针旋转。
4. RL平衡旋转：A的右子树的左子树插入新结点，先顺时针旋转，再逆时针旋转。

```
//RL示意图
A              A              A
  \              \              \
    B              B              C                C
                  /              \              /   \
                 C                B            A     B
```

小结

01 OPTION 熟练掌握顺序表（顺序查找）和有序表（折半查找）的查找算法及其性能分析；

02 OPTION 熟练掌握二叉排序树的创建和查找算法及其性能分析；

03 OPTION 掌握二叉排序树的插入算法及其性能分析；

04 OPTION 了解二叉排序树的删除方法；

05 OPTION 了解平衡二叉树概念及二叉排序树平衡化过程。