

# 查找

- 查找

在数据集合中寻找满足某种条件的数据元素的过程称为查找

- 查找表（查找结构）

用于查找的数据集合称为查找表，它由同一类型的数据元素（或记录）组成

- 关键字

数据元素中唯一标识该元素的某个数据项的值，使用基于关键字的查找，查找结果应该是唯一的。

## 对查找表的常见操作

1. 查找符合条件的数据元素

仅关注查找速度即可

2. 插入、删除某个数据元素

除了查找速度，也要关注插/删操作是否方便实现

## 查找算法的评价指标

- 查找长度

在查找运算中，需要对比关键字的次数称为查找长度

- 平均查找长度 (ASL, Average Search Length)

所有查找过程中进行关键字的比较次数的平均值。评价一个查找算法的效率时，通常考虑查找成功/查找失败两种情况的ASL。ASL的数量级反映了查找算法时间复杂度。

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

$n$  : 数据元素个数

$P_i$  : 查找第 $i$ 个元素的概率

$C_i$  : 查找第 $i$ 个元素的查找长度



## 顺序查找和折半查找

### 顺序查找

顺序查找，又叫“线性查找”，通常用于线性表。

算法思想：从头到尾挨个找（或者反过来）

### 实现

```
typedef struct{//查找表的数据结构（顺序表）
    ELEMTYPE *elem;//动态数组基址
    int TableLen;//表的长度
}SSTable;
```

```

//顺序查找
int Search_Seq(SSTable ST, ElemenType key)
{
    int i;
    for(i=0; i<ST.TableLen && ST.elem[i]!=key; i++)
        //查找成功，则返回元素下标；查找失败，则返回-1
    return i==ST.TableLen?-1:i;
}

//顺序查找（哨兵）
int Search_Seq(SSTable ST, ElemenType key)
{
    ST.elem[0]=key; //哨兵
    int i;
    for(i=ST.TableLen; ST.elem[i]!=key; --i); //从后往前找
    return i;//查找成功，则返回元素下标；查找失败，则返回0
}

```

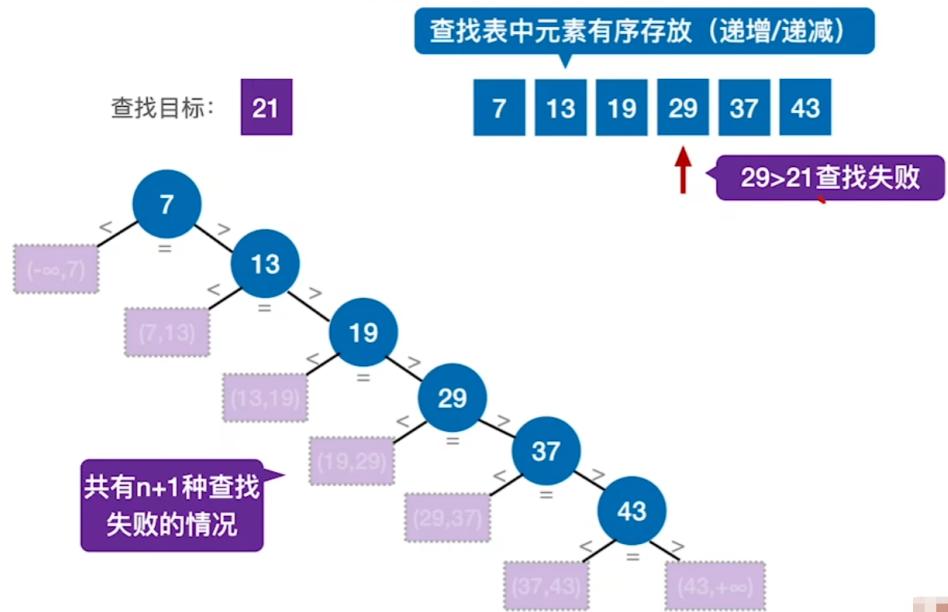
## 效率分析

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

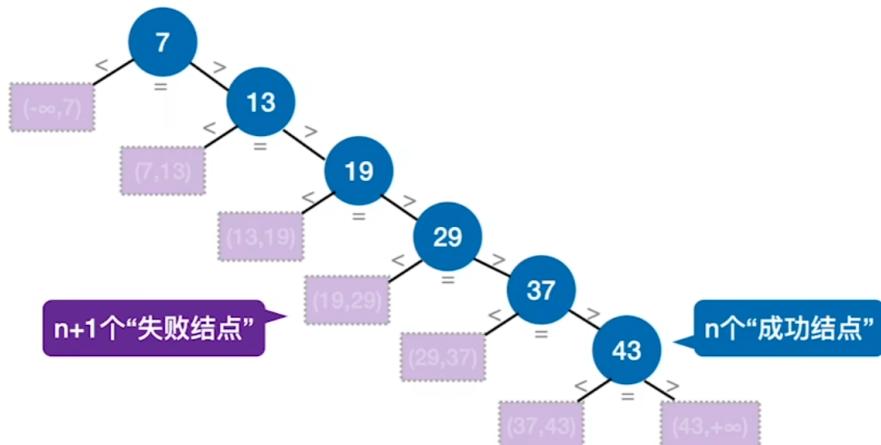
$$ASL_{\text{成功}} = \frac{1+2+3+\dots+n}{n} = \frac{n+1}{2}$$

$$ASL_{\text{失败}} = n+1$$

## 顺序查找的优化（对有序表）



## 用查找判定树分析ASL



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=其父结点所在层数

默认情况下，各种失败情况或成功情况都等概率发生

## 顺序查找的优化（被查概率不相等）

被查概率

7: 15%

13: 5%

19: 10%

29: 40%

37: 28%

43: 2%



$$ASL_{\text{成功}} = 1*0.15 + 2*0.05 + 3*0.1 + 4*0.4 + 5*0.28 + 6*0.02 = 3.67$$

被查概率大的放在靠前位置



$$ASL_{\text{成功}} = 1*0.4 + 2*0.28 + 3*0.15 + 4*0.1 + 5*0.05 + 6*0.02 = 2.18$$

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

## 顺序查找

算法实现

从头到尾（或者从尾到头）挨个找

适用于顺序表、链表，表中元素有序无序都OK  
可在0号位置存“哨兵”，从尾部向头部挨个查找  
优点：循环时无需判断下标是否越界

当前关键字大于（或小于）目标关键字时，查找失败

优化

若表中元素有序

优点：查找失败时 ASL 更少

成功结点的关键字对比次数=结点所在层数

失败结点的关键字对比次数=其父节点所在层数

若各个关键字被查概率不同

可按被查概率降序排列

优点：查找成功时 ASL 更少

时间复杂度  $O(n)$

## 折半查找

折半查找，又称“二分查找”，仅适用于有序的顺序表。顺序表拥有随机访问的特性，链表没有。

## 实现

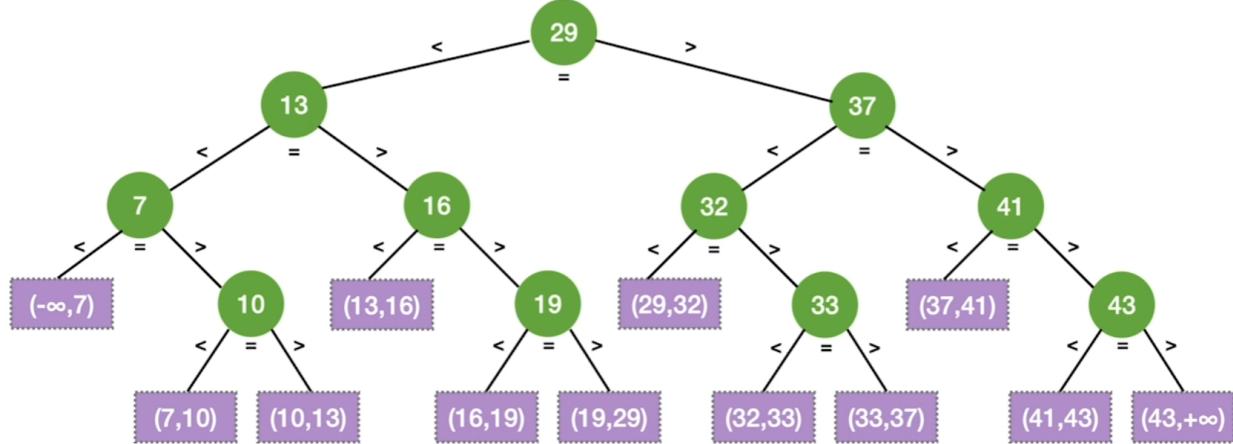
```
typedef struct{//查找表的数据结构（顺序表）
    ElemenType *elem; //动态数组基址
    int TableLen; //表的长度
}SSTable;
//折半查找
int Binary_Search(SSTable L, ElemenType key)
{
    int low=0,high=L.TableLen-1,mid;
    while(low<=high)
    {
        mid=(low+high)/2; //取中间位置
        if(L.elem[mid]==key)
```

```

        return mid;//查找成功则返回所在位置
    else if(L.elem[mid]>key)
        high = mid-1;//从前半部分继续查找
    else
        low = mid + 1;//从后半部分继续查找
    }
    return -1;//查找失败，返回-1
}

```

## 效率分析



$$ASL_{\text{成功}} = (1 * 1 + 2 * 2 + 3 * 4 + 4 * 4) / 11 = 3$$

$$ASL_{\text{失败}} = (3 * 4 + 4 * 8) / 12 = 11 / 3$$

## 折半查找判定树的构造

如果当前low和high之间有奇数个元素，则mid分隔后，左右两部分元素个数相等

如果当前low和high之间有偶数个元素，则mid分隔后，左半部分比右半部分少一个元素

折半查找的判定树中，若

$$mid = \lfloor (low + high) / 2 \rfloor$$

则对于任何一个结点，必有：

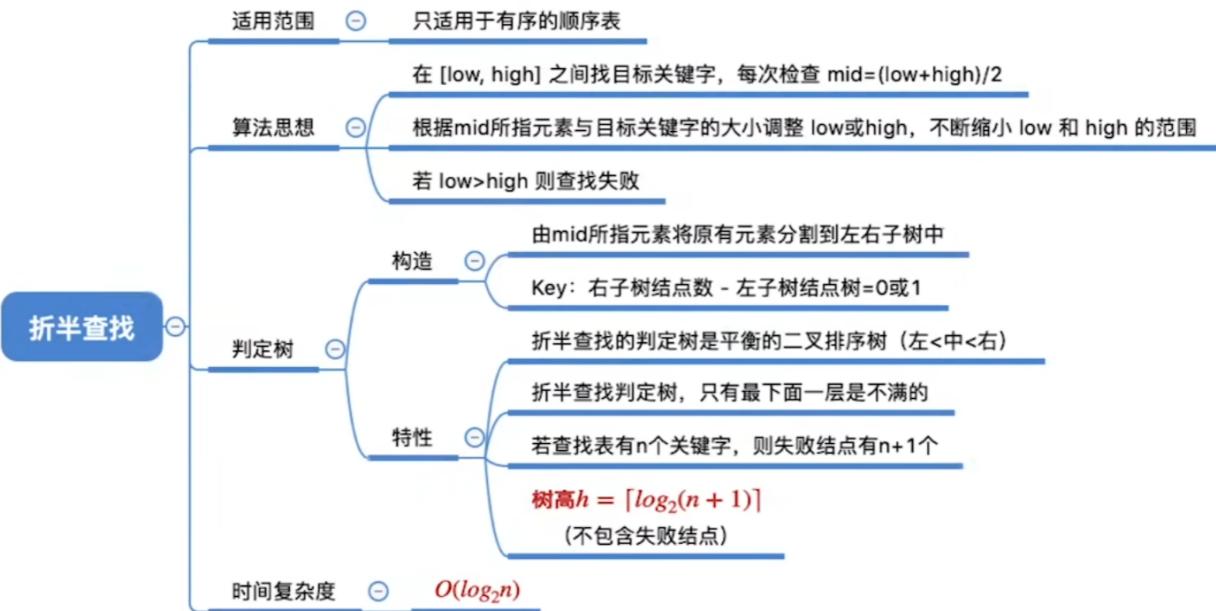
$$\text{右子树结点数} - \text{左子树结点数} = 0 \text{或} 1$$

折半查找的判定树一定是平衡二叉树

折半查找的判定树中，只有最下面一层是不满的

因此，元素个数为n时树高

$$h = \lceil \log_2(n + 1) \rceil$$



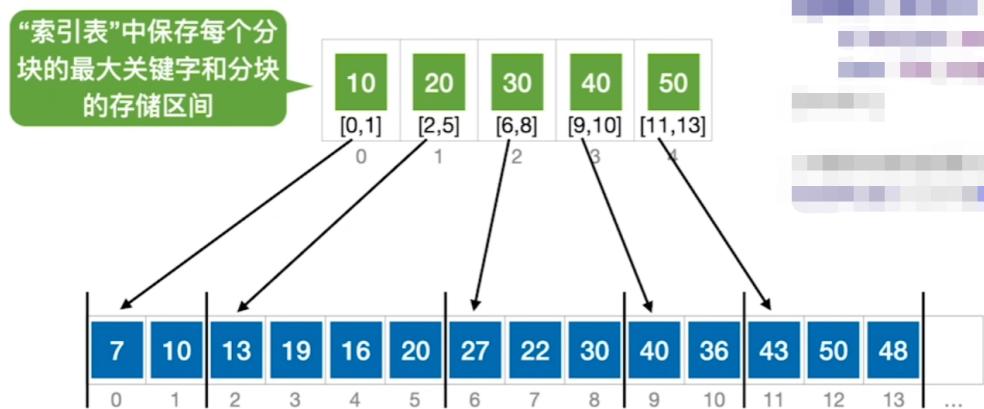
## 分块查找

分块查找，又称索引顺序查找，算法过程如下：

1. 在索引表中确定待查记录所属的分块（可顺序、可折半）
2. 在块内顺序查找

块内无序、块间有序

“索引表”中保存每个分块的最大关键字和分块的存储区间



```
//索引表
typedef struct{
    ElemenType maxValue;
    int low, high;
}Index;
//顺序表存储实际元素
ElemType List[100];
```

## 用折半查找索引

### 查找效率分析 (ASL)

4. 写出折半查找的递归算法。初始调用时，low为1，high为ST.length。

根据查找的起始位置和终止位置，将查找序列一分为二，判断所查找的关键字在哪一部分，然后用新的序列的起始位置和终止位置递归求解。

```

typedef struct{//查找表的数据结构
    ELEMType *elem;//存储空间基址, 建表时按实际长度分配, 0号留空
    int length;//表的长度
}SSTable;
int BinSearchRec(SSTable ST, ELEMType key, int low, int high)
{
    if(low>high)
        return 0;
    mid = (low+high)/2;//取中间位置
    if(key>ST.elem[mid])//向后半部分查找
        BinSearchRec(ST,key,mid+1,high);
    else if(key<ST.elem[mid])//向前半部分查找
        BinSearchRec(ST,key,low,mid-1);
    else//查找成功
        return mid;
}

```

5. 线性表中各结点的检索概率不等时, 可用如下策略提高顺序检索的效率: 若找到指定的结点, 则将该结点和其前驱结点(若存在)交换, 使得经常被检索的结点尽量位于表的前端。试设计在顺序结构和链式结构的线性表上实现上述策略的顺序检索算法。

检索时可先从表头开始向后顺序扫描, 若找到指定的结点, 则将该结点和其前驱结点(若存在)交换。采用顺序表存储结构的算法实现如下:

```

int SeqSrch(RcdType R[], ELEMType k)
{
    //顺序查找线性表, 找到后和其前面的元素交换
    int i=0;
    while((R[i].key!=k)&&(i<n))
        i++;//从前向后顺序查找指定结点
    if(i<n&&i>0)//若找到, 则交换
    {
        temp=R[i];R[i]=R[i-1];R[i-1]=temp;
        return --i;//交换成功, 返回交换后的位置
    }
    else return -1;//交换失败
}

```

链表方式实现的基本思想类似, 但在交换两个结点之前需要保存指向前一结点的指针。

6. 已知一个n阶矩阵A和一个目标值k。该矩阵无重复元素, 每行从左到右升序排列, 每列从上到下升序排列。请设计一个在时间上尽可能高效的算法, 判断矩阵中是否存在目标值k。例如, 矩阵为

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

目标值为8, 判断存在。要求:

- 给出算法的基本设计思想。
- 根据设计思想, 采用C或C++语言描述算法, 关键之处给出注释。
- 说明你的算法的时间复杂度和空间复杂度。

从矩阵A的右上角(最右列)开始比较, 若当前元素小于目标值, 则向下寻找下一个更大的元素; 若当前元素大于目标值, 则从右往左依次比较, 若目标值存在, 则只可能在该行中。

```

bool findkey(int A[][], int n, int k)
{
    int i=0, j=n-1;
    while(i<n&&j>=0)//离开边界时查找结束
    {
        if(A[i][j]==k) return true;//查找成功
        else if(A[i][j]>k) j--;//向左移动, 在该行内寻找目标值
        else i++;//向下移动, 查找下一个更大的元素
    }
    return false;//查找失败
}

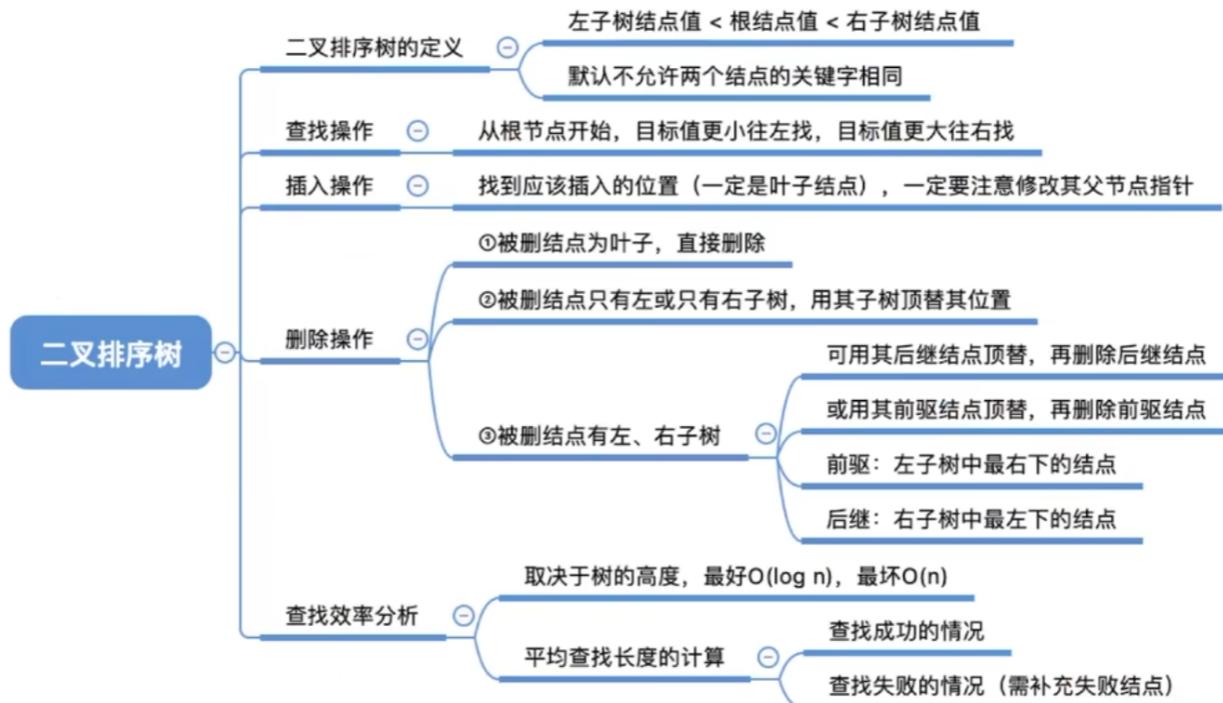
```

比较次数不超过 $2n$ 次，时间复杂度为 $O(n)$ ；空间复杂度为 $O(1)$ 。

## 树形查找

		BST	AVL Tree	Red-Black Tree
		1960	1962	1972
时间复杂度	Search	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$
	Insert	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$
	Delete	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$

## 二叉排序树 BST



## 二叉排序树的定义

二叉排序树，又称二叉查找树 (BST, Binary Search Tree)。

二叉排序树可用于元素的有序组织、搜索

一棵二叉树或者是空二叉树，或者是具有如下性质的二叉树：

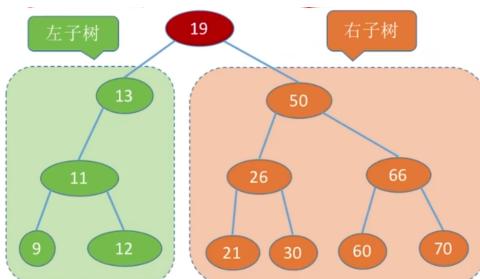
左子树上所有结点的关键字均小于根结点的关键字；

右子树上所有结点的关键字均大于根结点的关键字。

左子树和右子树又各是一棵二叉排序树。

左子树结点值<根节点值<右子树结点值

进行中序遍历，可以得到一个递增的有序序列



## 二叉排序树的查找

左子树结点值 < 根结点值 < 右子树结点值

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode *lchild, *rchild;
}BSTNode, *BSTree;
//在二叉排序树中查找值为key的结点 最坏空间复杂度O(1)
BSTNode *BST_Search(BSTree T, int key)
{
    while(T!=NULL&&key!=T->key)//若树空或等于根结点值，则结束循环
    {
        if(key<T->key)T=T->lchild;//小于，则在左子树上查找
        else T=T->rchild;//大于，则在右子树上查找
    }
    return T;
}
//在二叉排序树中查找值为key的结点（递归实现）最坏空间复杂度O(h)
BSTNode *BSTSearch(BSTree T, int key)
{
    if(T==NULL)
        return NULL;//查找失败
    if(key==T->key)
        return T;//查找成功
    else if(key<T->key)
        return BSTSearch(T->lchild, key); //在左子树中找
    else
        return BSTSearch(T->rchild, key); //在右子树中找
}
```

若树非空，目标值与根结点的值比较：

若相等，则查找成功；

若小于根结点，则在左子树上查找，否则在右子树上查找。

查找成功，返回结点指针；查找失败返回NULL

## 二叉排序树的插入

若原二叉排序树为空，则直接插入结点；否则，若关键字k小于根结点值，则插入到左子树，若关键字k大于根结点值，则插入到右子树

```
//在二叉排序树插入关键字为k的新结点（递归实现）
int BST_Insert(BSTree &T, int k)
{
    if(T==NULL)//原树为空，新插入的结点为根结点
    {
        T=(BSTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
        return 1;//返回1，插入成功
    }
    else if(k==T->key)//树中存在相同关键字的结点，插入失败
        return 0;
    else if(k<T->key)//插入到T的左子树
        return BST_Insert(T->lchild,k);
    else//插入到T的右子树
        return BST_Insert(T->rchild,k);
}
```

## 二叉排序树的构造

```
//按照str[]中的关键字序列建立二叉排序树
void Create_BST(BSTree &T, int str[], int n)
{
    T=NULL;//初始时T为空树
    int i=0;
    while(i<n)//依次将每个关键字插入到二叉排序树中
    {
        BST_Insert(T,str[i]);
        i++;
    }
}
```

不同的关键字序列可能得到同款二叉排序树

也可能得到不同款二叉排序树

## 二叉排序树的删除

左子树结点值 < 根结点值 < 右子树结点值

先搜索找到目标结点：

1. 若被删除结点z是叶结点，则直接删除，不会破坏二叉排序树的性质。
2. 若结点z只有一棵左子树或右子树，则让z的子树成为在父结点的子树，替代z的位置。
3. 若结点z有左、右两棵子树，则令z的直接后继（或直接前驱）替代z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。

进行中序遍历，可以得到一个递增的有序序列

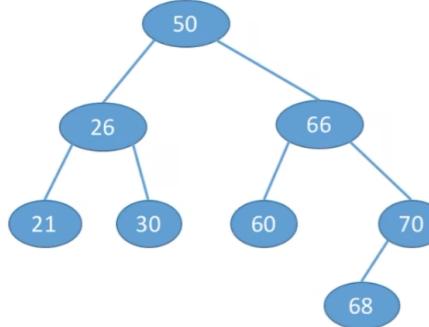
z的后继：z的右子树中最左下结点（该结点一定没有左子树）

z的前驱：z的左子树中最右下结点（该结点一定没有右子树）

## 查找效率分析

查找长度，在查找运算中，需要对比关键字的次数称为查找长度，反映了查找操作时间复杂度。

查找成功的平均查找长度ASL (Average Search Length)



$$ASL = (1 * 1 + 2 * 2 + 3 * 4 + 4 * 1) / 8 = 2.625$$

若树高为h，找到最下层的一个结点需要对比h次

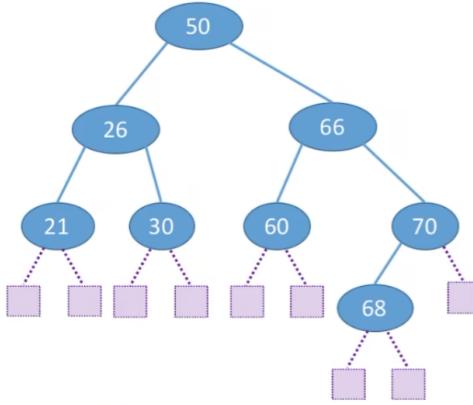
最好情况：n个结点的二叉树最小高度为

$$\lfloor \log_2 n \rfloor + 1$$

平均查找长度=O(log<sub>2</sub> n)

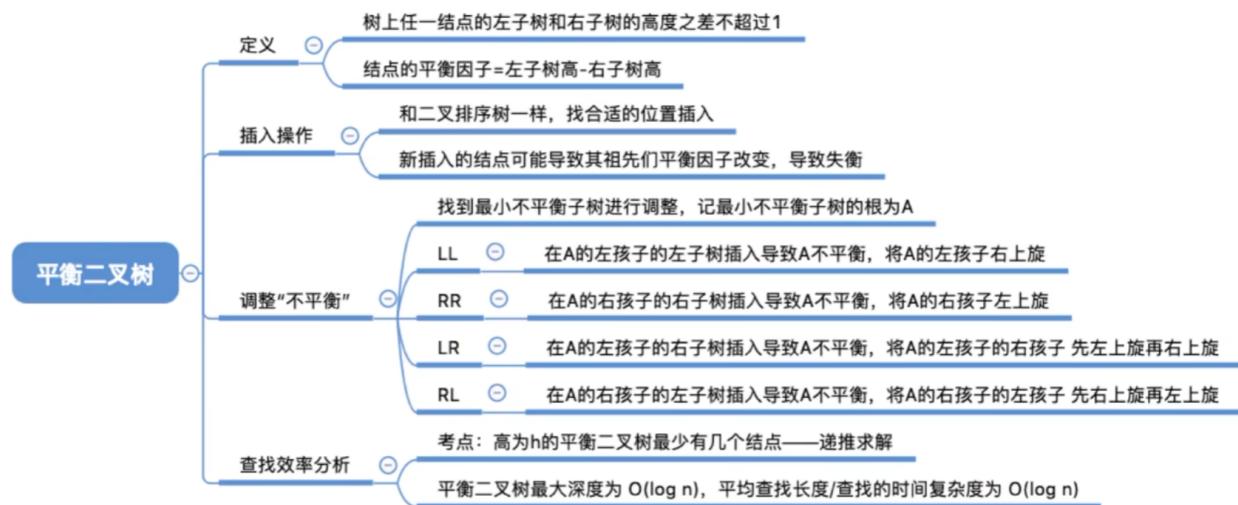
最坏情况：每个结点只有一个分支，树高h=结点数n。平均查找长度=O(n)

查找失败的平均查找长度ASL (Average Search Length)



$$ASL = (3 * 7 + 4 * 2) / 9 = 3.22$$

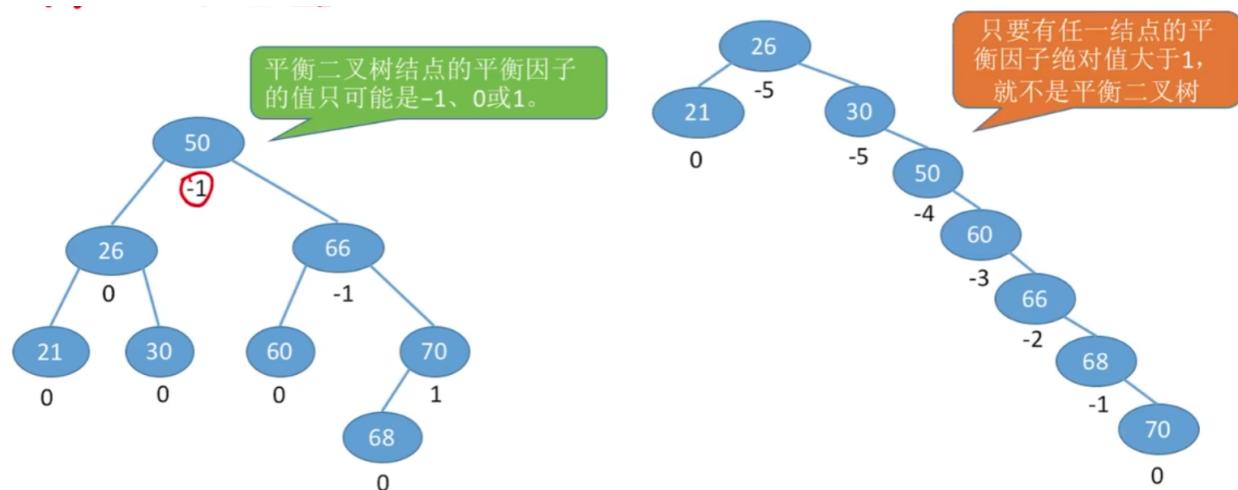
## 平衡二叉树 AVL



## 平衡二叉树的定义

平衡二叉树 (Balanced Binary Tree), 简称平衡树 (AVL树) ——树上任一结点的左子树和右子树的高度之差不超过1。

结点的平衡因子=左子树高-右子树高。



```

//平衡二叉树结点
typedef struct AVLNode{
    int key; //数据域
    int balance; //平衡因子
    struct AVLNode *lchild, *rchild;
}AVLNode, *AVLTree;

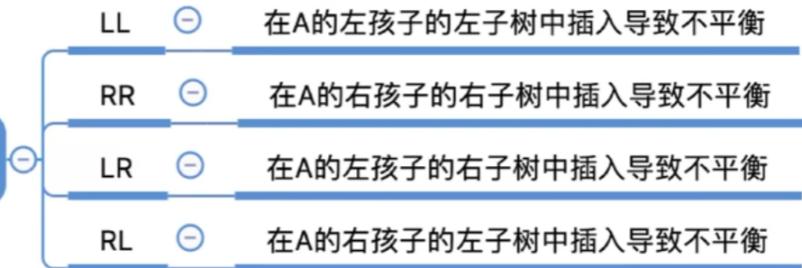
```

## 平衡二叉树的插入

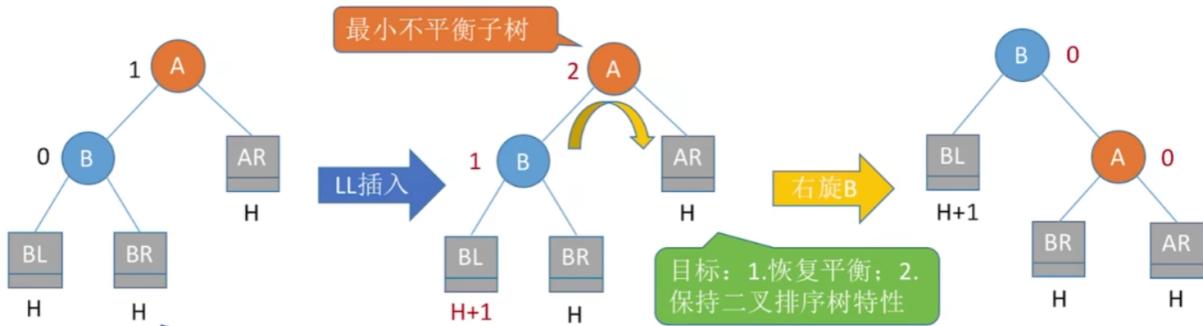
- 从插入点往回找到第一个不平衡结点，调整以该结点为根的子树
- 每次调整的对象都是“最小不平衡子树”
- 查找路径上的所有结点都有可能受到影响
- 在插入操作中，只要将最小不平衡子树调整平衡，则其他祖先结点都会恢复平衡

### 调整最小不平衡子树

#### 调整最小不平衡子树 A



#### 调整最小不平衡子树 (LL)



思考：为什么要假定所有子树的高度都是H？

二叉排序树的特性：左子树结点值 < 根结点值 < 右子树结点值

$BL < B < BR < A < AR$

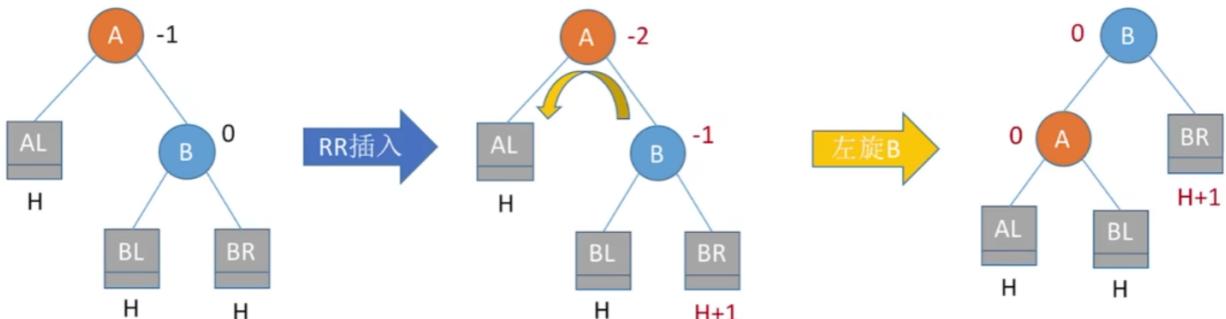
二叉排序树的特性：

左子树结点值 < 根结点值 < 右子树结点值

$BL < B < BR < A < AR$

LL平衡旋转（右单旋转）。由于在结点A的左孩子（L）的左子树（L）上插入了新结点，A的平衡因子由1增至2，导致以A为根的子树失去平衡，需要一次向右的旋转操作。将A的左孩子B向右上旋转代替A成为根结点，将A结点向右下旋转成为B的右子树的根结点，而B的原右子树则作为A结点的左子树。

#### 调整最小不平衡子树 (RR)



二叉排序树的特性：左子树结点值 < 根结点值 < 右子树结点值

$AL < A < BL < B < BR$

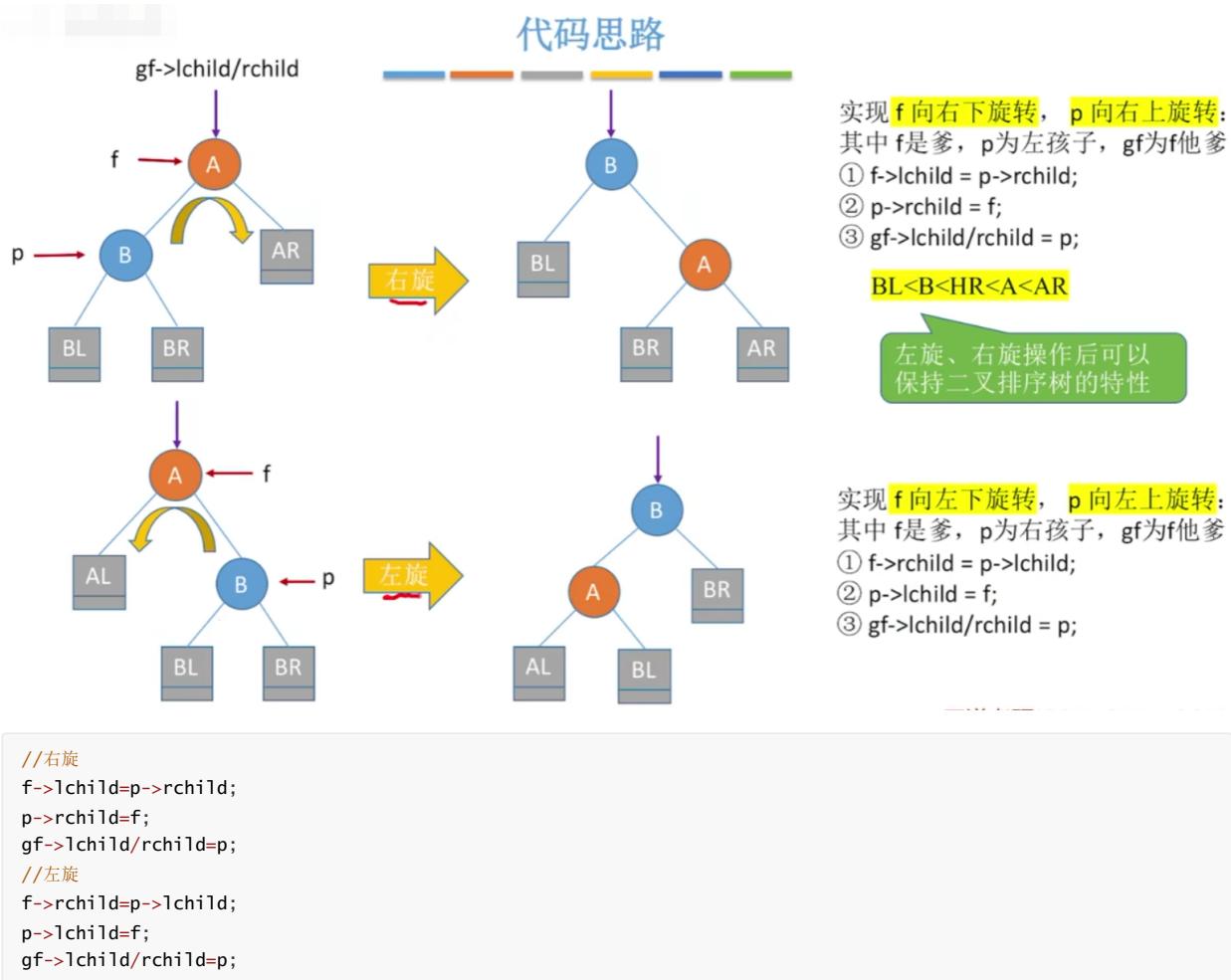
二叉排序树的特性：

左子树结点值 < 根结点值 < 右子树结点值

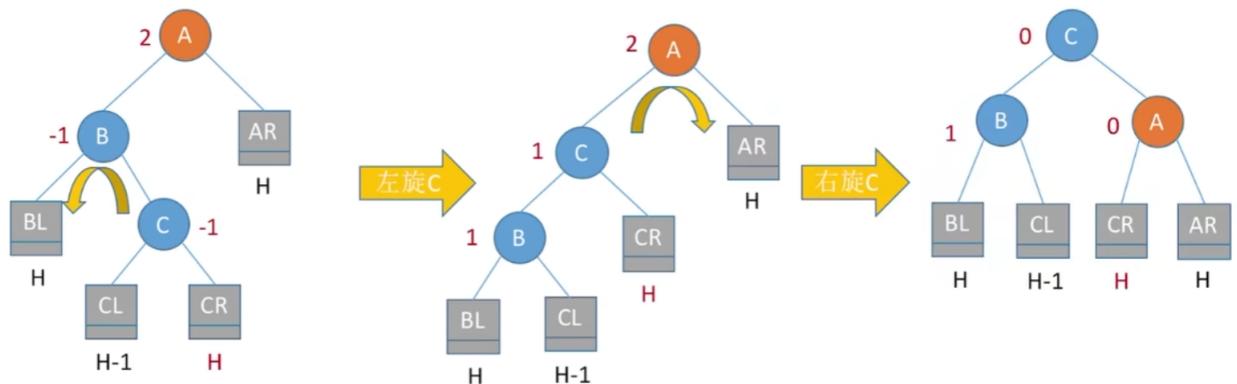
$AL < A < BL < B < BR$

RR平衡旋转（左单旋转）。由于在结点A的右孩子（R）的右子树（R）上插入了新结点，A的平衡因子由-1减至-2，导致以A为根的子树失去平衡，需要一次向左的旋转操作。将A的右孩子B向左上旋转代替A成为根结点，将A结点向左下旋转成为B的左子树的根结点，而B的原左子树则成为A结点的右子树。

## 代码思路



## 调整最小不平衡子树 (LR)

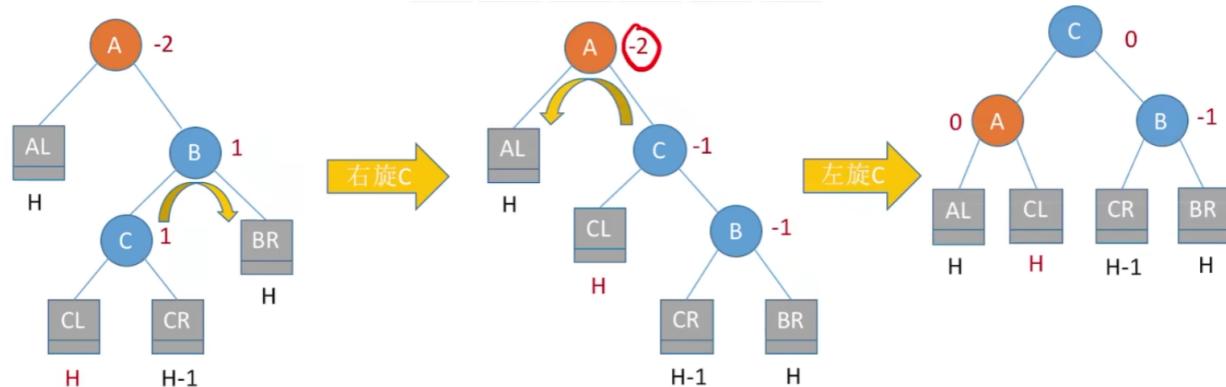




$$BL < B < CL < C < CR < A < AR$$

LR平衡旋转（先左后右双旋转）。由于在A的左孩子（L）的右子树（R）上插入新结点，A的平衡因子由1增至2，导致以A为根的子树失去平衡，需要进行两次旋转操作，先左旋转后右旋转。先将A结点的左孩子B的右子树的根结点C向左上旋转提升到B结点的位置，然后再把该C结点向右上旋转提升到A结点的位置。

### 调整最小不平衡子树 (RL)



$$AL < A < CL < C < CR < B < BR$$

RL平衡旋转（先右后左双旋转）。由于在A的右孩子（R）的左子树（L）上插入新结点，A的平衡因子由-1减至-2，导致以A为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转。先将A结点的右孩子B的左子树的根结点C向右上旋转提升到B结点的位置，然后再把该C结点向左上旋转提升到A结点的位置。

## 查找效率分析

若树高为h，则最坏情况下，查找一个关键字最多需要对比h次，即查找操作的时间复杂度不可能超过O(h)

平衡二叉树——树上任一结点的左子树和右子树的高度之差不超过1。

假设以 $n_h$ 表示深度为h的平衡树中含有的最少结点数

则有 $n_0 = 0, n_1 = 1, n_2 = 2$ , 并且有 $n_h = n_{h-1} + n_{h-2} + 1$

可以证明含有n个结点的平衡二叉树的最大深度为 $O(\log_2 n)$ , 平衡二叉树的平均查找长度为 $O(\log_2 n)$

## 平衡二叉树的删除

### 平衡二叉树的插入&删除

平衡二叉树的插入操作：

- 插入新结点后，要保持二叉排序树的特性不变（左<中<右）
- 若插入新结点导致不平衡，则需要调整平衡

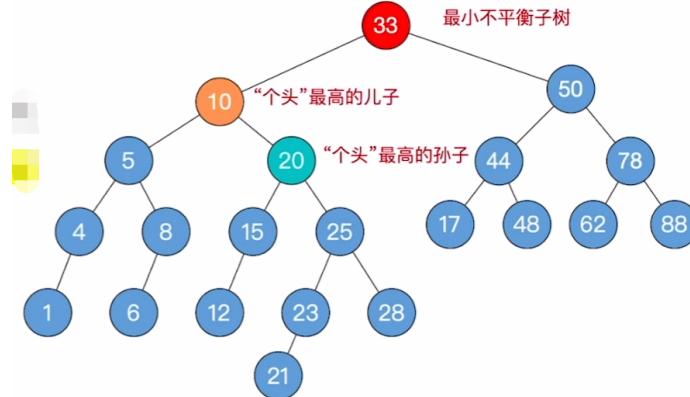
平衡二叉树的删除操作：

- 删除结点后，要保持二叉排序树的特性不变（左<中<右）
- 若删除结点导致不平衡，则需要调整平衡

## 平衡二叉树的删除

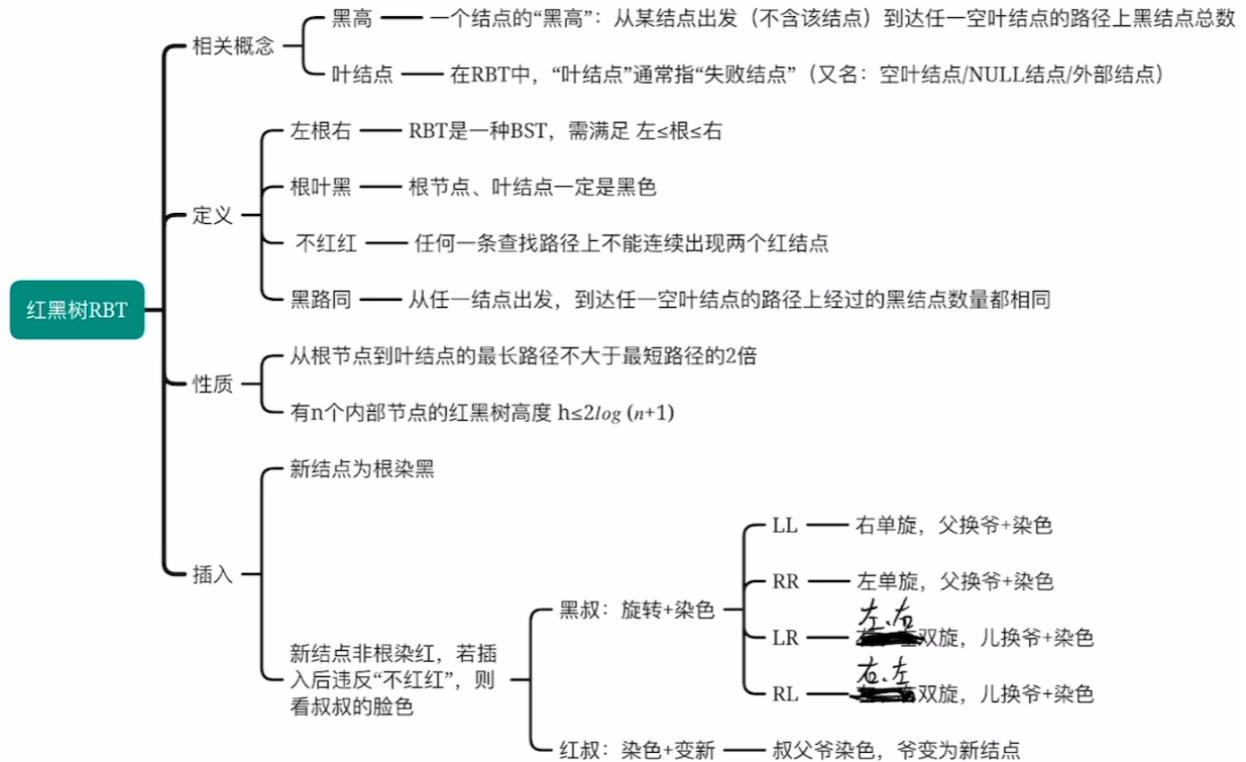
平衡二叉树删除操作时间复杂度=O(log<sub>2</sub> n)

1. 删除结点 (方法同“二叉排序树”)
  - 若删除的结点是叶子, 直接删
  - 若删除的结点只有一个子树, 用子树顶替删除位置
  - 若删除的结点有两棵子树, 用前驱 (或后继) 结点顶替, 并转换为对前驱 (或后继) 结点的删除
2. 一路向北找到最小不平衡子树, 找不到就return
3. 找到最小不平衡子树下, “个头”最高的儿子、孙子
4. 根据孙子的位置, 调整平衡 (LL/RR/LR/RL)
  - 孙子在LL: 儿子右单旋
  - 孙子在RR: 儿子左单旋
  - 孙子在LR: 孙子先左旋, 再右旋
  - 孙子在RL: 孙子先右旋, 再左旋



5. 如果不平衡向上传递, 继续 2
  - 对最小不平衡子树的旋转可能导致树变矮, 从而导致上层祖先不平衡 (不平衡向上传递)

## 红黑树 Red-Black Tree



左根右 根叶黑

不红红 黑路同

## 为什么要发明红黑树？

平衡二叉树AVL：插入/删除很容易破坏“平衡”特性，需要频繁调整树的形态。如：插入操作导致不平衡，则需要先计算平衡因子，找到最小不平衡子树（时间开销大），再进行LL/RR/LR/RL调整。

红黑树RBT：插入/删除很多时候不会破坏“红黑”特性，无需频繁调整树的形态。即便需要调整，一般都可以在常数级时间内完成。

平衡二叉树：适用于以查为主、很少插入/删除的场景

红黑树：适用于频繁插入、删除的场景，实用性更强

## 红黑树大概会怎么考？

红黑树的定义、性质——选择题

红黑树的插入/删除——要能手绘从插入过程（不太可能考代码，略复杂），删除操作也比较麻烦，也许不考。

## 红黑树的定义

红黑树是二叉排序树

左子树结点值  $\leq$  根结点值  $\leq$  右子树结点值

与普通BST相比，有什么要求

1. 每个结点或是红色，或是黑色的
2. 根结点是黑色的
3. 叶结点（外部结点、NULL结点、失败结点）均是黑色的
4. 不存在两个相邻的红结点（即红结点的父结点和孩子结点均是黑色）
5. 对每个结点，从该结点到任一叶结点的简单路径上，所含黑结点的数目相同

```

struct RBnode{//红黑树的结点定义
    int key;//关键字的值
    RBnode* parent;//父结点指针
    RBnode* lchild;//左孩子指针
    RBnode* rchild;//右孩子指针
    int color;//结点颜色, 如: 可用0/1表示黑/红, 也可使用枚举型enum表示颜色
};


```

左根右, 根叶黑

不红红, 黑路同

结点的“黑高”

结点的黑高bh——从某结点出发 (不含该结点) 到达任一空叶结点的路径上黑结点总数

## 红黑树的性质

1. 从根结点到叶结点的最长路径不大于最短路径的2倍
2. 有n个内部结点的红黑树高度

$$h \leq 2 \log_2(n + 1)$$

红黑树查找操作时间复杂度=O(log<sub>2</sub> n), 查找效率与AVL树同等数量级

## 红黑树的查找

与BST、AVL相同, 从根出发, 左小右大, 若查找到一个空叶结点, 则查找失败

## 红黑树的插入

- 先查找, 确定插入位置 (原理同二叉排序树), 插入新结点
- 新结点是根——染为黑色
- 新结点非根——染为红色
  - 若插入新结点后依然满足红黑树定义, 则插入结束
  - 若插入新结点后不满足红黑树定义, 需要调整, 使其重新满足红黑树定义

如何调整: 看新结点叔叔的脸色

- 黑叔: 旋转+染色
  - LL型: 右单旋, 父换爷+染色
  - RR型: 左单旋, 父换爷+染色
  - LR型: 左、右双旋, 儿换爷+染色
  - RL型: 右、左双旋, 儿换爷+染色
- 红叔: 染色+变新
  - 叔父爷染色, 爷变为新结点

## 与“黑高”相关的推论

结点的黑高bh——从某结点出发 (不含该结点) 到达任一叶结点的路径上黑结点总数。

思考: 根结点黑高为h的红黑树, 内部结点数 (关键字) 至少有多少个?

回答: 内部结点数最少的情况——总共h层黑结点的满树形态

结论: 若根结点黑高为h, 内部结点数 (关键字) 最少有2<sup>h</sup>-1个

## 红黑树的删除操作

重要考点

1. 红黑树删除操作的时间复杂度=O(log<sub>2</sub> n)
2. 在红黑树中删除结点的处理方式和“二叉排序树的删除”一样
3. 按2删除结点后, 可能破坏“红黑树特性”, 此时需要调整结点颜色、位置, 使其再次满足“红黑树特性”。

5. 试编写一个算法，判断给定的二叉树是否是二叉排序树。

对二叉排序树来说，其中序遍历序列为一个递增有序序列。因此，对给定的二叉树进行中序遍历，若始终能保持前一个值比后一个值小，则说明该二叉树是一棵二叉排序树。

```
KeyType predt=32767; //predt为全局变量，保存当前结点中序前驱

int JudgeBST(BiTTree bt)
{
    int b1,b2;
    if(bt==NULL) //空树
        return 1;
    else
    {
        b1=JudgeBST(bt->lchild); //判断左子树是否是二叉排序树
        if(b1==0 || predt>=bt->data) //若左子树返回值为0或前驱大于或等于当前结点
            return 0;
        predt=bt->data; //保存当前结点的关键字
        b2=JudgeBST(bt->rchild); //判断右子树
        return b2; //返回右子树的结果
    }
}
```

6. 设计一个算法，求出指定结点在给定二叉排序树中的层次。

设二叉树采用二叉链表存储结构，在二叉排序树中，查找一次就下降一层。因此，查找该结点所用的次数就是该结点在二叉排序树中的层次。采用二叉排序树非递归查找算法，用n保存查找层次，每查找一次，n就加1，直到找到相应的结点。

```
int level(BiTTree bt, BSTNode *p)
{
    int n=0; //统计查找次数
    BiTree t=bt;
    if(bt!=NULL)
    {
        n++;
        while(t->data!=p->data)
        {
            if(p->data<t->data) //在左子树中查找
                t=t->lchild;
            else //在右子树中查找
                t=t->rchild;
            n++; //层次加1
        }
    }
    return n;
}
```

7. 利用二叉树遍历的思想编写一个判断二叉树是否是平衡二叉树的算法。

设置二叉树的平衡标记balance，以标记返回二叉树bt是否为平衡二叉树，若为平衡二叉树，则返回1，否则返回0；h为二叉树bt的高度。采用后序遍历的递归算法：

- 若bt为空，则高度为0，balance=1。
- 若bt仅有根结点，则高度为1，balance=1。
- 否则，对bt的左、右子树执行递归运算，返回左、右子树的高度和平衡标记，bt的高度为最高子树的高度加1。若左、右子树的高度差大于1，则balance=0；若左、右子树的高度差小于或等于-1，且左、右子树都平衡时，balance=1，否则balance=0。

```
void Judge_AVL(BiTTree bt, int &balance, int &h)
{
    int b1=0, br=0, h1=0, hr=0; //左、右子树的平衡标记和高度
    if(bt==NULL) //空树，高度为0
    {
        h=0;
        balance=1;
    }
    else if(bt->lchild==NULL&&bt->rchild==NULL) //仅有根结点，则高度为1
    {
```

```

    h=1;
    balance=1;
}
else
{
    Judge_AVL(bt->lchild,b1,h1); //递归判断左子树
    Judge_AVL(bt->rchild,br,hr); //递归判断右子树
    h=(h1>hr?h1:hr)+1;
    if(abs(h1-hr)<2)//若子树高度差的绝对值<2，则看左、右子树是否都平衡
        balance=b1&&br;//&&为逻辑与，即左、右子树都平衡时，二叉树平衡
    else
        balance=0;
}
}

```

8. 设计一个算法，求出给定二叉排序树中最小和最大的关键字。

在一棵二叉排序树中，最左下结点即为关键字最小的结点，最右下结点即为关键字最大的结点，本算法只要找出这两个结点即可，而不需要比较关键字。

```

KeyType MinKey(BSTNode *bt)
{
    while(bt->lchild!=NULL)
        bt=bt->lchild;
    return bt->data;
}

KeyType MaxKey(BSTNode *bt)
{
    //求出二叉排序树最大关键字结点
    while(bt->rchild!=NULL)
        bt=bt->rchild;
    return bt->data;
}

```

9. 设计一个算法，从大到小输出二叉排序树中所有值不小于k的关键字。

由二叉排序树的性质可知，右子树中所有的结点值均大于根结点值，左子树中所有的结点值均小于根结点值。为了从大到小输出，先遍历右子树，再访问根结点，后遍历左子树。

```

void OutPut(BSTNode *bt, KeyType k)
{
    //本算法从大到小输出二叉排序树中所有值不小于k的关键字
    if(bt==NULL)
        return;
    if(bt->rchild!=NULL)
        OutPut(bt->rchild,k); //递归输出右子树结点
    if(bt->data>=k)
        printf("%d",bt->data); //只输出大于或等于k的结点值
    if(bt->lchild!=NULL)
        OutPut(bt->lchild,k); //递归输出左子树的结点
}

```

也可采用中序遍历加辅助栈的方法实现。

10. 编写一个递归算法，在一棵有n个结点的、随机建立起来的二叉排序树上查找第k( $1 \leq k \leq n$ )小的元素，并返回指向该结点的指针。要求算法的平均时间复杂度为  $O(\log_2 n)$ 。二叉排序树的每个结点中除data、lchild、rchild 等数据成员外，增加一个count成员，保存以该结点为根的子树上的结点个数。

设二叉排序树的根节点为\*t，根据结点存储的信息，有以下几种情况：

当t->lchild为空时，情况如下：

- 若t->rchild非空且k==1，则\*t即为第k小的元素，查找成功。
- 若t->rchild非空且k!=1，则第k小的元素必在\*t的右子树。

当t->lchild非空时，情况如下：

- t->lchild->count==k-1，\*t即为第k小的元素，查找成功
- t->lchild->count>k-1，第k小的元素必在\*t的左子树，继续到\*t的左子树中查找。

- o  $t \rightarrow lchild \rightarrow count < k - 1$ , 第k小的元素必在右子树, 继续搜索右子树, 寻找第 $k - (t \rightarrow lchild \rightarrow count + 1)$ 小的元素  
对左右子树的搜索采用相同的规则, 递归实现的算法描述如下:

```

BSTNode *Search_Small(BSTNode *t, int k)
{
    //以t为根的子树上寻找第k小的元素, 返回其所在结点的指针。k从1开始计算
    //在树结点中增加一个count数据成员, 存储以该结点为根的子树的结点个数
    if(k<1 || k>t->count) return NULL;
    if(t->lchild==NULL)
    {
        if(k==1) return t;
        else return Search_Small(t->rchild,k-1);
    }
    else
    {
        if(t->lchild->count==k-1) return t;
        if(t->lchild->count>k-1) return Search_Small(t->lchild,k);
        if(t->lchild->count<k-1)
            return Search_Small(t->rchild, k-(t->lchild->count+1));
    }
}

```

最大查找长度取决于树的高度。由于二叉排序树是随机生成的, 其高度应是 $O(\log_2 n)$ , 算法的时间复杂度为 $O(\log_2 n)$ .

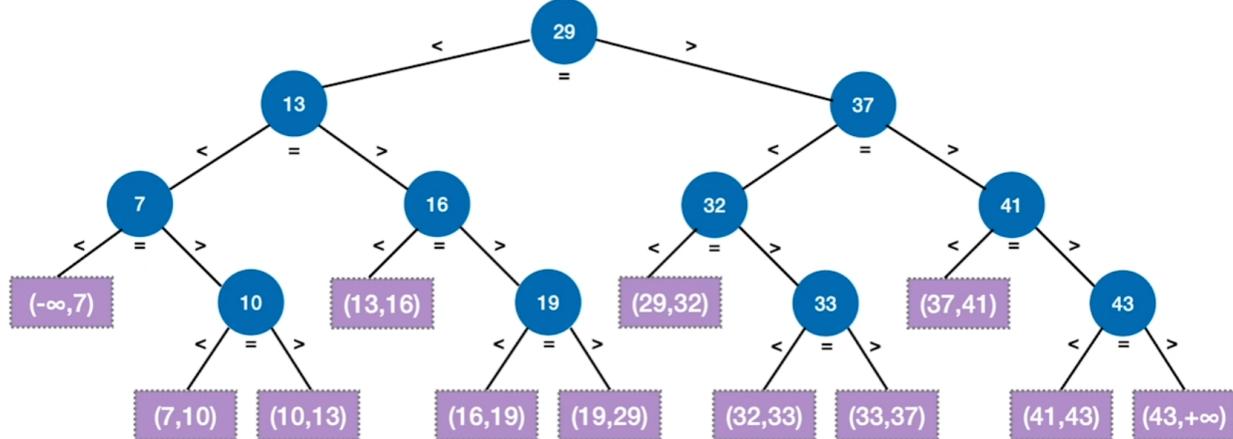
$$\text{卡特兰数} = \begin{cases} \text{栈, } n \text{ 个元素入栈, 求出栈序列个数} \\ \text{二叉树, } n \text{ 个结点能构成多少种不同形状的二叉树} \\ \text{括号匹配, } n \text{ 对括号, 有多少种括号匹配序列} \end{cases}$$

$$\frac{1}{n+1} C_{2n}^n$$

## B树和B+树

### B树

#### 二叉查找树 (BST)



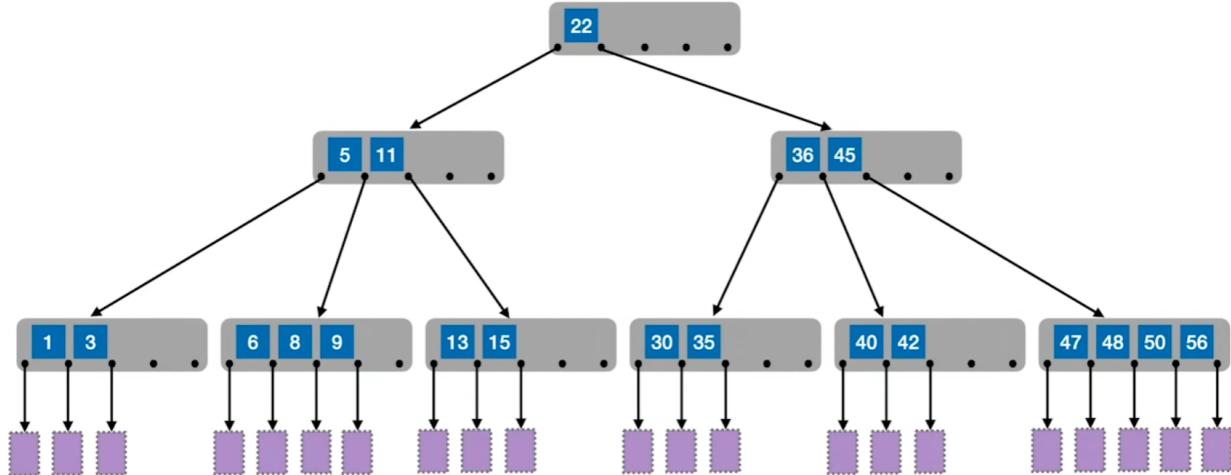
```

//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode *lchild, *rchild;
} BSTNode, *BSTree;

```

能不能变成m叉查找树?

## 5叉查找树



```
//5叉排序树的结点定义
struct Node{
    ELEM_TYPE keys[4];//最多4个关键字
    struct Node *child[5];//最多5个孩子
    int num;//结点中有几个关键字
};

//最少1个关键字，2个分叉
//最多4个关键字，5个分叉
//结点内关键字有序
```

## 如何查找

每个结点内也可以折半查找

## 如何保证查找效率

- 若每个结点内关键字太少，导致树变高，要查更多层结点，效率低

策略： $m$ 叉查找树中，规定除了根结点外，任何结点至少有 $\lceil m/2 \rceil$ 个分叉，即至少含有 $\lceil m/2 \rceil - 1$ 关键字

- 不够“平衡”，树会很高，要查很多层结点

策略： $m$ 叉查找树中，规定对于任何一个结点，其所有子树的高度都要相同

对于5叉排序树，规定除了根结点外，任何结点都至少有3个分叉，2个关键字

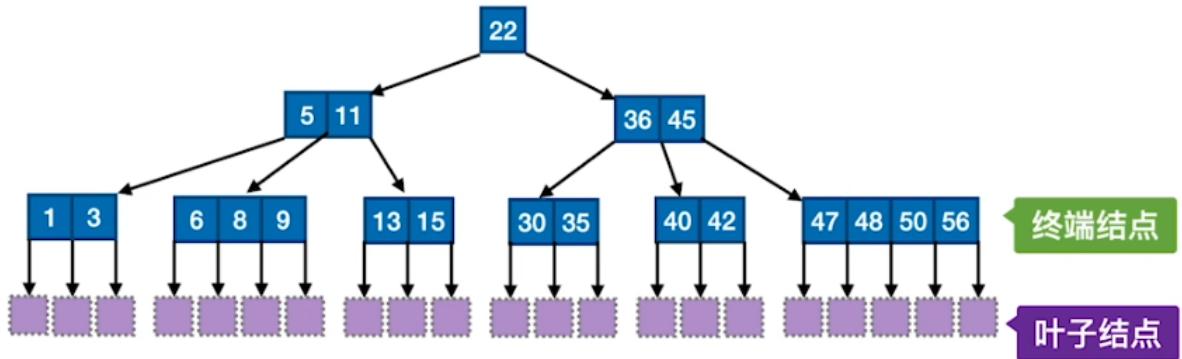
## B树

B树，又称多路平衡查找树，B树中所有结点的孩子个数的最大值称为B树的阶，通常用 $m$ 表示。一棵 $m$ 阶B树或为空树，或为满足如下特性的 $m$ 叉树：

- 树中每个结点至多有 $m$ 棵子树，即至多含有 $m-1$ 个关键字。
- 若根结点不是终端结点，则至少有两棵子树。
- 除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树，即至少含有 $\lceil m/2 \rceil - 1$ 个关键字。
- 所有的叶结点都出现在同一层次上，并且不带信息（可以视为外部结点或类似于折半查找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。
- 所有非叶结点的结构如下：

其中， $K_i (i = 1, 2, \dots, n)$ 为结点的关键字，且满足 $K_1 < K_2 < \dots < K_n$ ； $P_i (i = 0, 1, \dots, n)$ 为指向子树根结点的指针，且指针 $p_{i-1}$ 所指子树中所有结点的关键字均小于 $K_i$ ， $P_i$ 所指子树中所有结点的关键字均大于 $K_i$ ， $n (\lceil m/2 \rceil - 1 \leq n \leq m - 1)$ 为结点中关键字的个数。

$n$	$P_0$	$K_1$	$P_1$	$K_2$	$P_2$	$\dots$	$K_n$	$P_n$
-----	-------	-------	-------	-------	-------	---------	-------	-------



m阶B树的核心特性：

1. 根结点的子树数  $\in [2, m]$ , 关键字数  $\in [1, m - 1]$   
其他结点的子树数  $\in [\lceil m/2 \rceil, m]$ ; 关键字数  $\in [\lceil m/2 \rceil - 1, m - 1]$
2. 对任一结点, 其所有子树高度都相同
3. 关键字的值: 子树0<关键字1<子树1<关键字2<子树2<... (类比二叉查找树 左<中<右)

## B树的高度

问题：含n个关键字的m阶B树，最小高度、最大高度是多少？

最小高度——让每个结点尽可能的满，有m-1个关键字，m个分叉，则有

$$n \leq (m-1)(1+m+m^2+m^3+\dots+m^{h-1}) = m^h - 1, \text{因此 } h \geq \log_m(n+1)$$

最大高度——让各层的分叉尽可能的少，即根结点只有2个分叉，其他结点只有m/2向上取整个分叉各层结点至少有：第一层1、第二层2、第三层2(m/2)向上取整...第h层2(m/2向上取整)^{h-2}，第h+1层共有叶子结点(失败结点) 2(m/2向上取整)^{h-1}个

$$n \text{个关键字的 } B \text{树必有 } n+1 \text{ 个叶子结点, 则 } n+1 \geq 2(\lceil m/2 \rceil)^{h-1}, \text{ 即 } h \leq \log_{\lceil m/2 \rceil} \frac{n+1}{2} + 1$$

n个关键字将数域切分为n+1个区间

问题：含n个关键字的m叉B树，最小高度、最大高度是多少？

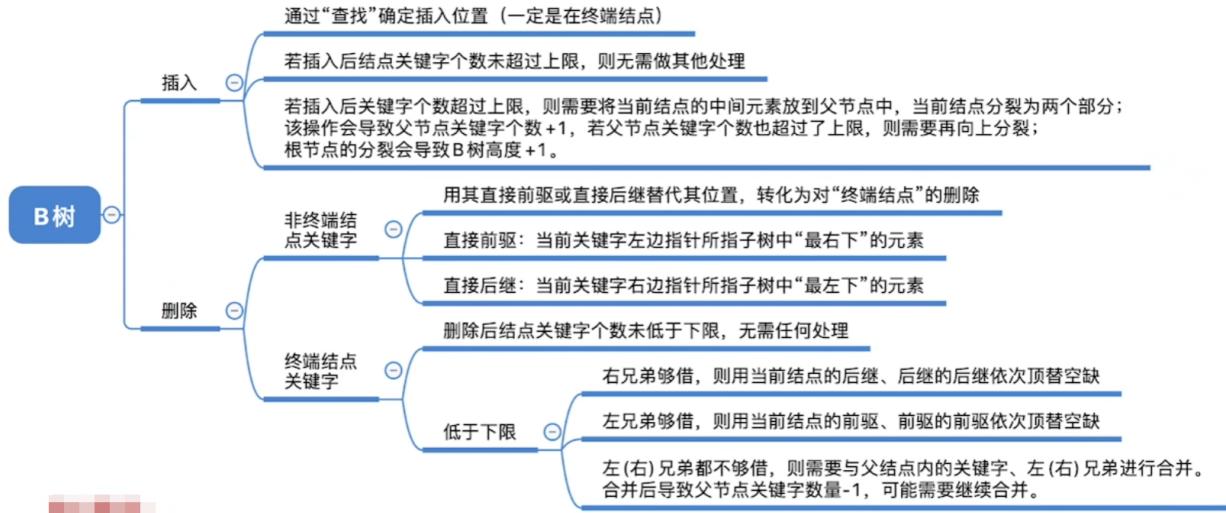
最大高度——让每个结点包含的关键字、分叉尽可能的少。记  $k=[m/2]$

	最少结点数	最少关键字数
第一层	1	1
第二层	2	$2(k-1)$
第三层	$2k$	$2k(k-1)$
第四层	$2k^2$	$2k^2(k-1)$
...	...	....
第h层	$2k^{h-2}$	$2k^{h-2}(k-1)$

h层的m阶B树至少包含关键字总数  $1+2(k-1)(k^0+k^1+k^2+\dots+k^{h-2}) = 1+2(k^{h-1}-1)$

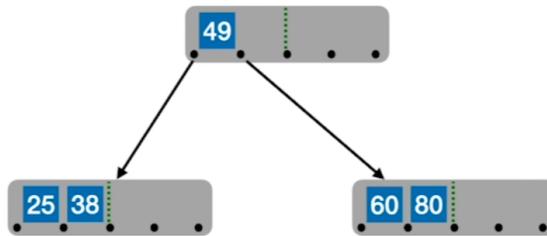
若关键字总数少于这个值，则高度一定小于h，因此  $n \geq 1+2(k^{h-1}-1)$

$$\text{得, } h \leq \log_k \frac{n+1}{2} + 1 = \log_{\lceil m/2 \rceil} \frac{n+1}{2} + 1$$



## B树的插入

5阶B树  
结点关键字个数 $\lceil m/2 \rceil$ ,  $-1 \leq n \leq m - 1$   
即： $2 \leq n \leq 4$ (省略失败结点)



在插入key后, 若导致原结点关键字数超过上限, 则从中间位置( $\lceil m/2 \rceil$ )将其中的关键字分为两部分,  
左部分包含的关键字放在原结点中,  
右部分包含的关键字放到新结点中,  
中间位置( $\lceil m/2 \rceil$ )的结点插入原结点的父结点

新元素一定是插入到最底层“终端结点”，用“查找来确定插入位置”

若此时导致其父结点的关键字个数也超过了上限, 则继续进行这种分裂操作,  
直至这个过程传到根结点为止, 进而导致B树高度增1

## B树的删除

若被删除关键字在终端结点, 则直接删除该关键字  
(要注意结点关键字个数是否低于下限 $\lceil m/2 \rceil - 1$ )

若被删除关键字在非终端结点, 则用直接前驱或直接后继来替代被删除的关键字

直接前驱：当前关键字左侧指针所指子树中最右下的元素  
直接后继：当前关键字右侧指针所指子树中最左下的元素

对非终端结点关键字的删除, 必然可以转化为对终端结点的删除操作

当前结点关键字不够, 兄弟够借

若被删除关键字所在结点删除前的关键字个数低于下限, 且与此结点右(或左)兄弟结点的关键字个数还很宽裕,

则需要调整该结点右(或左)兄弟结点及其双亲结点(父子换位法)

当右兄弟宽裕时, 用当前结点的后继, 后继的后继来填补空缺

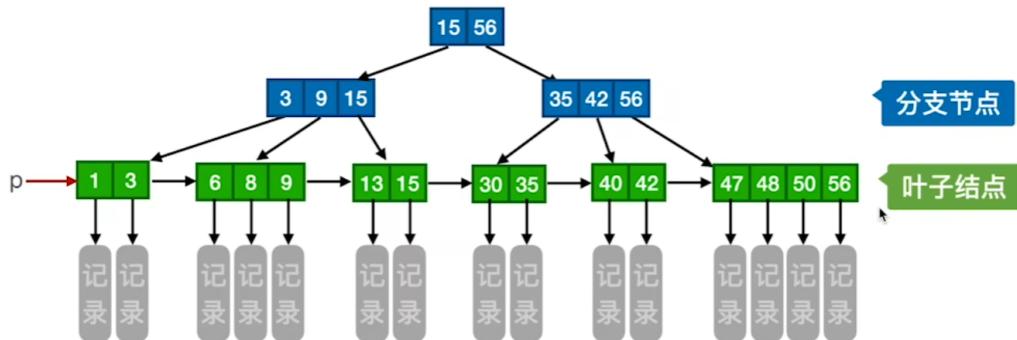
当左兄弟宽裕时, 用当前结点的前驱, 前驱的前驱来填补空缺

兄弟不够借

若被删除关键字所在结点删除前的关键字个数低于下限,  
且此时与该结点相邻的左, 右兄弟结点的关键字个数均 =  $\lceil m/2 \rceil - 1$ ,  
则将关键字删除后与左(或右)兄弟结点及双亲结点中的关键字进行合并

# B+树

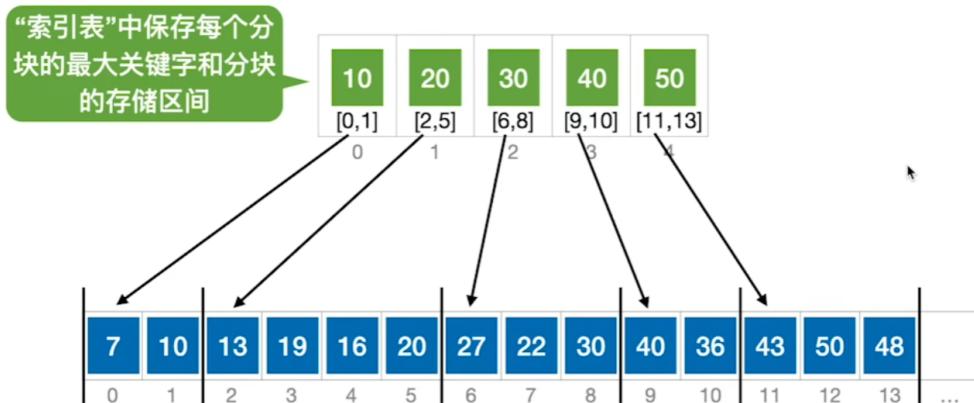
## 4阶B+树



一棵m阶的B+树需满足下列条件：

1. 每个分支结点最多有m棵子树（孩子结点）。
2. 非叶根结点至少有两棵子树，其他每个分支结点至少有 $m/2$ 向上取整棵子树。  
可以理解为：要追求“绝对平衡”，即所有子树高度要相同
3. 结点的子树个数与关键字个数相等。
4. 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。  
支持顺序查找
5. 所有分支结点中仅包含它的各个子结点中关键字的最大值及指向其子结点的指针。

## 对比：分块查找



## B+树的查找

B+树种，无论查找成功与否，最终一定都要走到最下面一层结点

顺序查找

## 对比：B树的查找

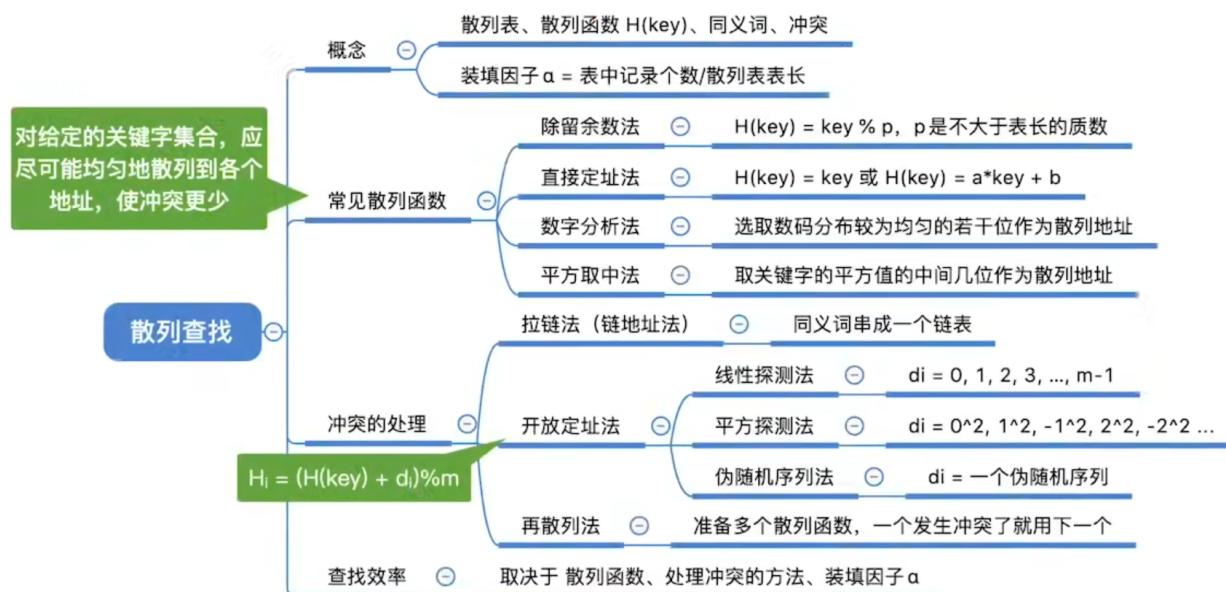
查找成功，可能停在任何一层

在B+树中，非叶结点不含有该关键字对应记录的存储地址。

可以使一个磁盘块可以包含更多个关键字，使得B+树的阶更大，树高更矮，读磁盘次数更少，查找更快

	m阶B树	m阶B+树
类比	二叉查找树的进化——>m叉查找树	分块查找的进化——>多级分块查找
关键字与分叉	n个关键字对应n+1个分叉（子树）	n个关键字对应n个分叉
结点包含的信息	所有结点中都包含记录的信息	只有最下层叶子结点才包含记录的信息 (可使树更矮)
查找方式	不支持顺序查找。查找成功时，可能停在任何一层结点，查找速度“不稳定”	支持顺序查找。查找成功或失败都会到达最下一层结点，查找速度“稳定”
相同点	除根节点外，最少 $\lceil m/2 \rceil$ 个分叉（确保结点不要太“空”） 任何一个结点的子树都要一样高（确保“绝对平衡”）	

## 散列查找



## 散列表 (Hash Table)

散列表 (Hash Table)，又称哈希表，是一种数据结构，特点是：数据元素的关键字与其存储地址直接相关

如何建立“关键字”与“存储地址”的联系？

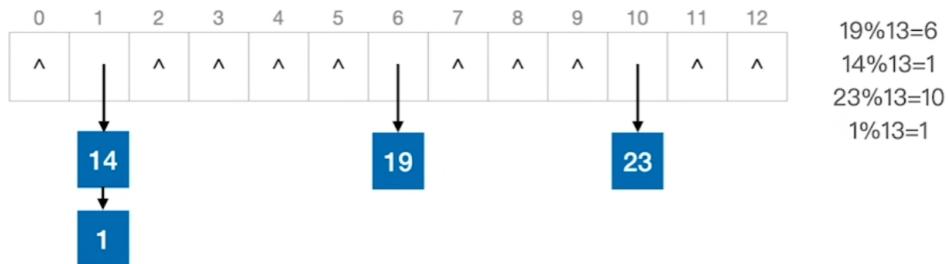
通过“散列函数 (哈希函数)”:  $\text{Addr} = H(\text{key})$

若不同的关键字通过散列函数映射到同一个值，则称它们为“同义词”

通过散列函数确定的位置已经存放了其他元素，则称这种情况为“冲突”

## 处理冲突的方法——拉链法

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  
 $H(key) = key \% 13$



拉链法（又称链接法、链地址法）处理“冲突”：把所有“同义词”存储在一个链表中

## 散列查找

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

装填因子会直接影响散列表的查找效率

## 常见的散列函数

设计目标——让不同关键字的冲突尽可能地少

### 除留余数法

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

散列表表长为m，取一个不大于m但最接近或等于m的质数p

质数又称素数。指除了1和此整数自身外，不能被其他自然数整除的数

用质数取模，分布更均匀，冲突更少。

### 直接定址法

$$H(key) = key$$

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

其中，a和b是常数。这种方法计算最简单，且不会产生冲突。它适合关键字的分布基本连续的情况，若关键字分布不连续，空位较多，则会造成存储空间的浪费。

### 数字分析法

选取数码分布较均匀的若干位作为散列地址

设关键字是r进制数（如十进制数），而r个数码在各位上出现的频率不一定相同，可能在某些位上分布均匀一些，每种数码出现的机会均等；而在某些位上分布不均匀，只有某几种数码经常出现，此时可选取数码分布较为均匀的若干位作为散列地址。这种方法适合于已知的关键字集合，若更换了关键字，则需要重新构造新的散列函数。

例：以“手机号码”作为关键字设计散列函数

138XXXX2875  
 138XXXX1682  
 138XXXX9125  
 ....  
 199XXXX1684  
 199XXXX1236

设计长度为10000的散列表，以  
 手机号后四位作为散列地址



### 平方取中法

取关键字的平方值的中间几位作为散列地址。

具体取多少位要视实际情况而定。这种方法得到的散列地址与关键字的每位都有关系，因此使得散列地址分布比较均匀，适用于关键字的每位取值都不够均匀或均小于散列地址所需的位数。

例：要存储整个学校的学生信息，以“身份证号”作为关键字设计散列函数

身份证号码规则：

- 前1、2位数字表示：所在省份的代码；
- 第3、4位数字表示：所在城市的代码；
- 第5、6位数字表示：所在区县的代码；
- 第7-14位数字表示：出生年、月、日；
- 第15、16位数字表示：所在地的派出所的代码；
- 第17位数字表示性别：奇数表示男性，偶数表示女性；
- 第18位数字是校检码。

---

散列查找是典型的“用空间换时间”的算法，只要散列函数设计的合理，则散列表越长，冲突的概率越低。

---

## 处理冲突的方法——开放定址法

所谓开放定址法，是指可存放新表项的空间地址既向它的同义词表项开放，又向它的非同义词表项开放。其数学递推公式为：

$$H_i = (H(key) + d_i) \mod m$$

$i = 0, 1, 2, \dots, k$  ( $k \leq m - 1$ )， $m$  表示散列表表长； $d_i$  为增量序列； $i$  可理解为“第*i*次发生冲突”

### 1. 线性探测法

$d_i = 0, 1, 2, 3, \dots, m - 1$ ；即发生冲突时，每次往后探测相邻的下一个单元是否为空

### 2. 平方探测法

$d_i = 0^2, 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$  时，称为平方探测法，又称二次探测法其中  $k \leq m/2$

### 3. 伪随机序列法

$d_i = \text{某个伪随机序列}$

注意：采用“开放定址法”时，删除结点不能简单地将被删结点的空间置为空，否则将截断在它之后填入散列表的同义词结点的查找路径，可以做一个“删除标记”，进行逻辑删除

## 查找效率分析 (ASL)

线性探测法很容易造成同义词、非同义词的“聚集（堆积）”现象，严重影响查找效率

产生原因——冲突后再探测一定是放在某个连续的位置

## 处理冲突的方法——再散列法

再散列法（再哈希法）：除了原始的散列函数  $H(key)$  之外，多准备几个散列函数，当散列函数冲突时，用下一个散列函数计算一个新地址，直到不冲突位置：

$$H_i = RH_i(Key), i = 1, 2, 3, \dots, k$$