第六章、查找

1、掌握静态查找表——顺序表、有序表、索引表的查找算法;理解算法复杂性的分析过程;熟悉算法特点。

2、掌握动态查找表——二叉排序树和平衡二叉树的概念、基本操作及其实现。

3、理解 B 树的概念和特点。

4、熟练掌握哈希查找思想、哈希冲突解决方法、哈希查找性能。

掌握静态查找表——顺序表、有序表、索引表的查找算法;理解算法复杂性的分析过程;熟悉算法特点

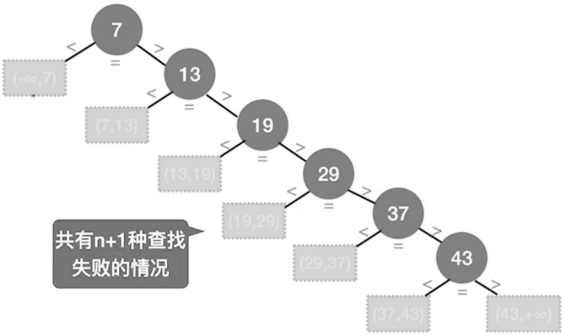
$$\text{动态查找表}\begin{cases}\text{静态查找表}\begin{cases}\text{①检索某个特定的数据元素是否在查找表中}\\\text{②检索某个特定的数据元素的各种属性}\end{cases}\\\text{③在查找表中插入/删除数据元素}\end{cases}$$

<table>
<tr><th></th><th></th><th>算法评价</th></tr>
<tr><td>顺序查找</td><td>typedef float KeyType;
typedef int KeyType;
type char *keytype;
#define EQ(a, b) ((a) == (b))
//--------------上面是定义
typedef struct
{
    ElemType elem; //数据元素存储空间基址,0号单元留空
    int length;   //表长度
} SSTable;
int Search.Seq(SStable ST, KeyType key)
{
    //若找到，函数值为该元素在表中的位置，否则为 0
    ST.elem[0].key = key;//哨兵
    for (int i = ST.length; ST.elem[i]!=key; --i)
    //后往前
    {
        return i;
    }
}</td><td>$ASL\begin{cases}\text{成功：}1+2+3+\cdots+n=\\\text{失败：}n+1\end{cases}$

缺点：平均查找长度大，当 n 很大时，查找效率低

优点：简单、适用面广，对表的结构没有任何要求</td></tr>
<tr><td></td><td>顺序查找的二叉判定树</td><td>仅针对有序表

失败 ASL:</td></tr>
</table>



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

折半查找

```
int Search.Bin(SSTable ST, KeyType key)
{
    low = 1, high = ST.length;
    while (low <= high)
    {
        mid = (low + high) / 2; // mid向下取整
        if (key == ST.elem[mid].key)
        {
            return mid;
        }
        if (key <= ST.elem[mid].key)
        {
            high = mid - 1; //比中值小，在前半部分继续查
        }
        else
        {
            low = mid + 1; //比中值大，在后半部分继续查
        }
    }
    return 0;
}
```

仅针对有序表和顺序存储结构

每个记录查找概率相同

设有序表长度

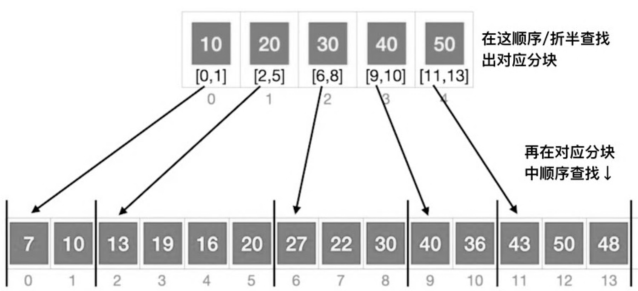
$n=2^{h-1}$, 成功 ASL

$$=\frac{1}{n}\sum_{i=1}^{h}j*2^{j-1}$$

$$=\frac{n+1}{n}log_2(n+1)-1$$

$$=log_2(n+1)-1(n>50)$$

分块/索引顺序查找



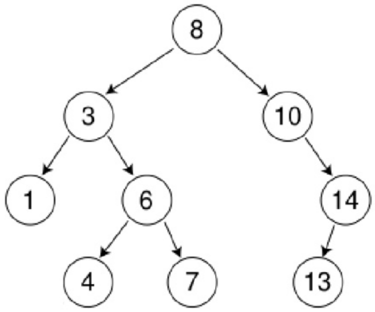
用于索引表表示的静态查找表

$$ASL\begin{cases}\text{成功：复杂，按情况计算}\\\text{失败：复杂，不考}\end{cases}$$

二叉排序树/二叉查找树/二叉搜索树 BST

<table>
<tr><th></th><th>代码</th><th>算法评价</th></tr>
<tr><td>定义</td><td>

递归表现为左子树<根<右子树



</td><td></td></tr>
<tr><td>查找</td><td>

法 1:非递归实现

```
//在二叉排序树中查找值为 key 的结点
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;
}
```

法二：递归实现

```
//在二叉排序树中查找值为 key 的结点（递归实现）
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
}
```

</td><td>

空间复杂度

$\begin{cases}\text{非递归：} O(1) \\ \text{递归：} O(h)\end{cases}$

成功的 ASL：

$\begin{cases}\text{最好：} O(log_2 n) \\ \text{最坏：} O(n)\end{cases}$

最好时为满二叉

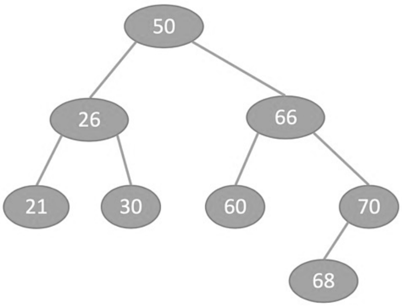
最坏时为单支树

</td></tr>
<tr><td>插入</td><td>

插入的结点一定是叶子结点，且是查找失败时的查找路径上的最后一个结点的左孩子 or 右孩子

```
//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);
}
```

</td><td></td></tr>
</table>

<table>
<tr>
<td>构造</td>
<td>按照序列从左到右的顺序，大的往右边放，小的往左边放。比如，{50, 66, 60, 26, 21, 30, 70, 68}的 BST


//按照 str[] 中的关键字序列建立二叉排序树
void Creat_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++;
    }
}</td>
<td>不同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</td>
</tr>
<tr>
<td>删除</td>
<td>step1. 搜索找到目标结点
step2.
叶子结点 → 直接删
只有左子树/右子树 → 子树替代掉它的位置
左右子树都有：用直接后继替代，后继原位执行删除；用直接前驱替代，前驱原位执行删除</td>
<td></td>
</tr>
</table>

平衡二叉树 AVL

- 相关定义

结点的平衡因子：该结点左子树的高-右子树的高

平衡二叉树：平衡二叉树结点的平衡因子只可能是-1、0 或 1

- 查找效率

AVL 高一层，就多分析一次，所以分析查找效率就是分析 AVL 的高。

平衡二叉树的最大深度$O(log_2n)$ →平均查找长度/时间复杂度$O(log_2n)$

- 高为 h 的平衡二叉树最少有几个结点

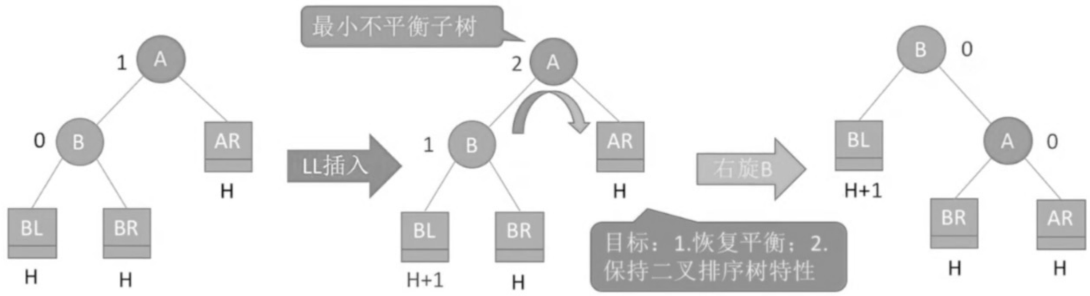
此时所有非叶子结点的平衡因子都为 1

设n_h为高度为h的 AVL 最少有的结点数，如$n_0 = 0$，n_1=1，$n_1 = 2$

$$n_h = n_{h-1} + n_{h-2} + 1$$

● *插入结点后如何构造一个新的AVL* $\begin{cases} LL \rightarrow A\text{的左孩子的左子树中插入导致不平衡} \\ RR \rightarrow A\text{的右}\qquad\text{右} \\ LR \rightarrow A\text{的左}\qquad\text{右} \\ RR \rightarrow A\text{的右}\qquad\text{左} \end{cases}$

LL



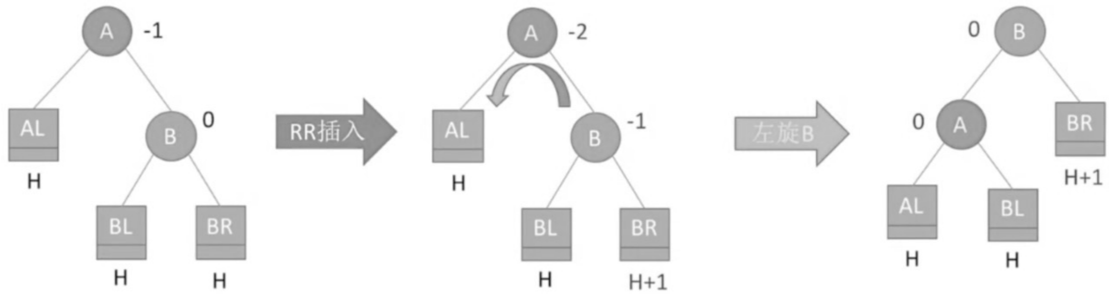
//代码实现

①A->lchild=B->rchild

②B->rchild=A

③A的父结点->lchild/rchild=B

RR

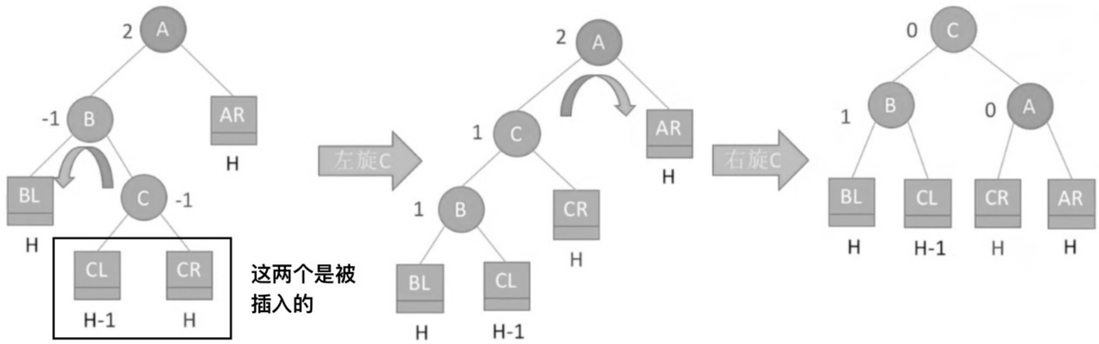


①A->rchild=B->lchild

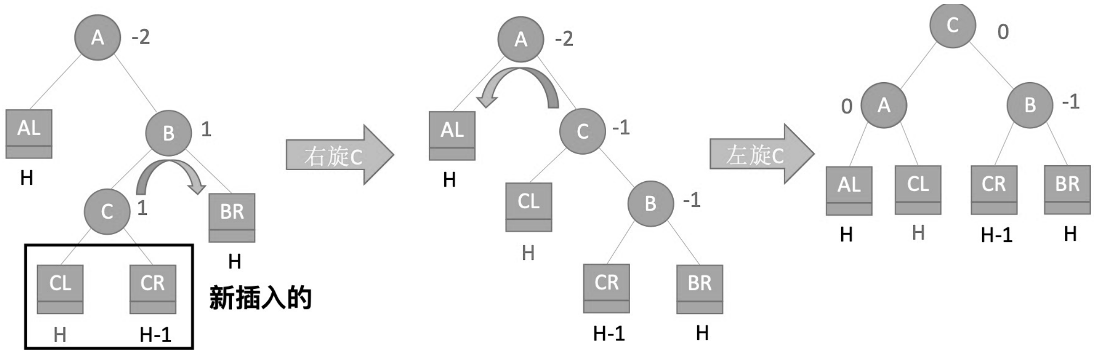
②B->lchild=A

③A的父结点->lchild/rchild=B

A 的 *LR*



A 的 RL



理解 B 树/多路平衡查找树的概念和特点（不要求代码）

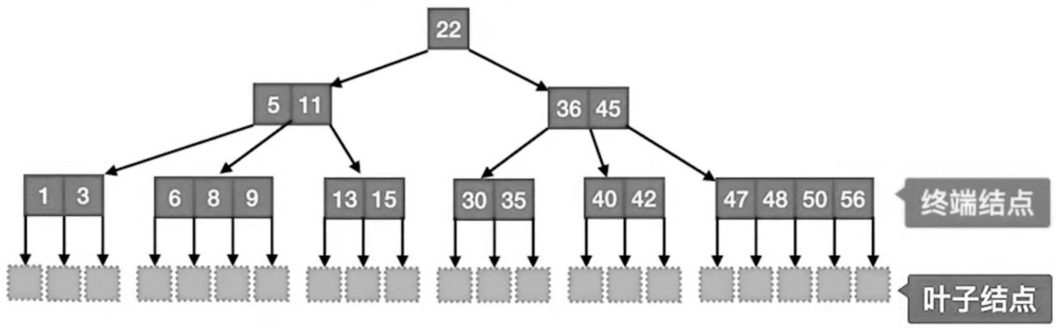
B 树的阶：B 树中所有结点的孩子个数的最大值，通常用 m 表示。

B 树概念

一棵 m 阶 B 树要么是空树，要么是满足如下特性的 m 叉树:

- 根节点的子树∈ $[2,m]$，关键字数∈ $[1,m-1]$。
- 其他结点的子树∈ $[\lceil m/2\rceil,m]$，关键字数∈ $(\lceil m/2\rceil-1,m-1]$)
- 对任一结点，其所有子树高度都相同
- 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)

likethis



B 树的特点

n 个关键字的 B 树必有 n+1 个叶子结点

含 n 个关键字的 m 阶 B 树的高$\begin{cases}最小：log_m(n+1)\\最大：log_{\lceil m/2\rceil}\frac{n+1}{2}+1\end{cases}$

推理：

最大高度——=求：多高的 B 树至少有 n 个结点。让各层的分叉尽可能的少，即根节点只有 2 个分叉，其他结点只有$\lceil m/2\rceil$个分叉，各层结点至少有：第一层 1、第二层 2、第三层 $2\lceil m/2\rceil$ ... 第h层 $2(\lceil m/2\rceil)^{h-2}$、第$h+1$层共有叶子结点(失败结点) $2(\lceil m/2\rceil)^{h-1}$ 个

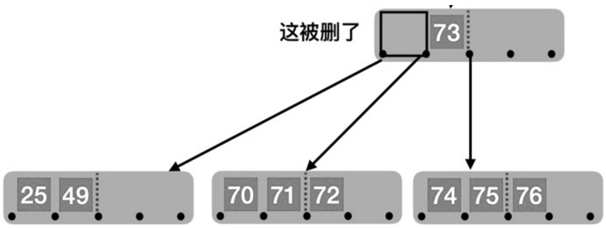
最小高度——让每个结点尽可能的满，有$m-1$个关键字，m 个分叉，则有 $n \le (m-1)(1+m+m^2+m^3+\ldots+m^h-1)=m^h-1$，

B 树的删除

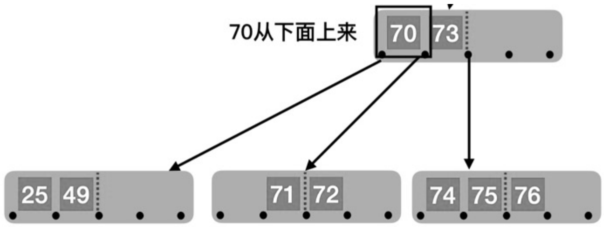
$$\begin{cases}\text{终端结点}\to\text{删除后低于下限？}\begin{cases}\text{是}\to\text{兄弟结点够挪用？}\begin{cases}\text{是}\begin{cases}\text{挪用右兄弟}\\\text{挪用左兄弟}\end{cases}\\\text{否}\to\text{把父结点抠下来合并}\end{cases}\\\text{否}\to\text{直接删}\end{cases}\\\text{非终端结点}\to\text{删除后用直接前驱 or 直接后继顶替}\end{cases}$$

终端结点
↓
删除后低于下限
↓
右兄弟够用
↓
挪右兄弟

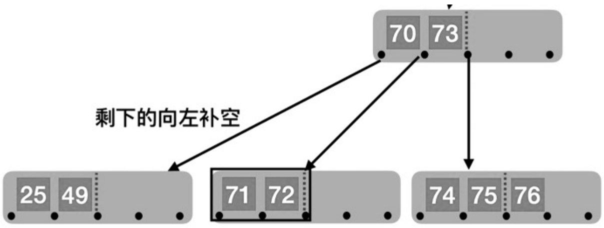
***step*1.**



***step*2.**

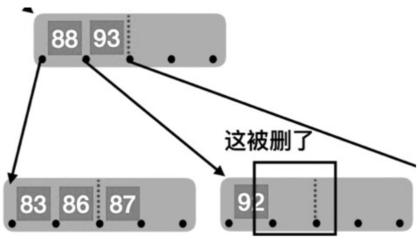


***step*3.**

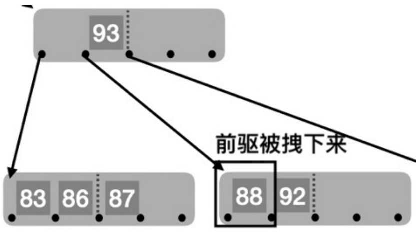


终端结点
↓
删除后低于下限
↓
左兄弟够用
↓
挪左兄弟

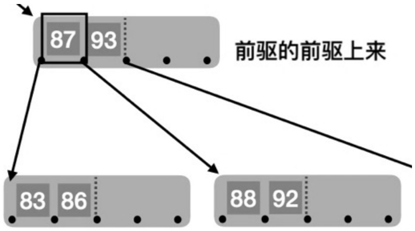
***step*1.**



***step*2.用前驱（88）和前驱的前驱（87）补空。先把前驱拽下来**



***step*3.再把前驱的前驱薅上去**



终端结点

↓

删除后低于下限

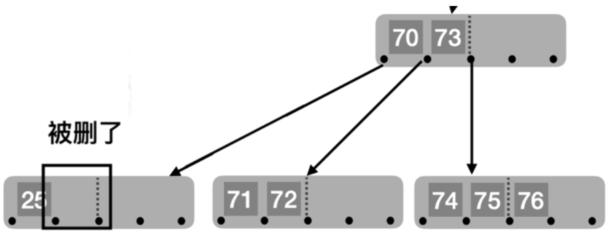
↓

兄弟不够用

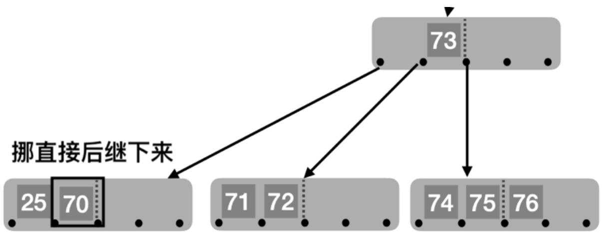
↓

抠父结点然后合并

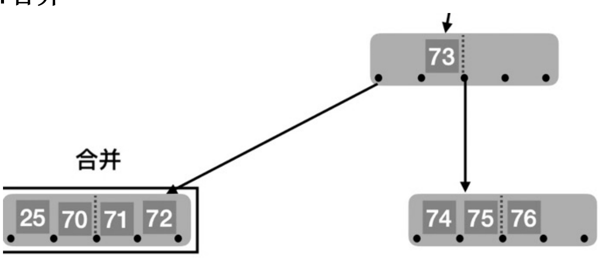
***step*1.**



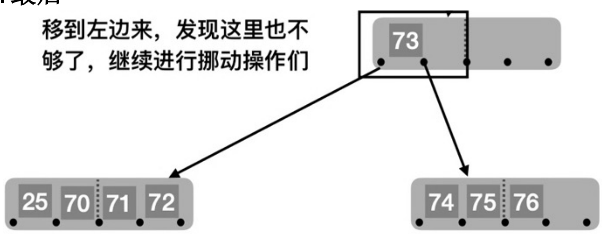
***step*2.** 挪后继下来



***step*3.** 合并



***step*4.** 最后

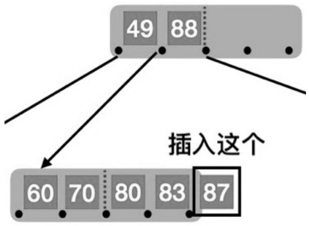


B 树的插入

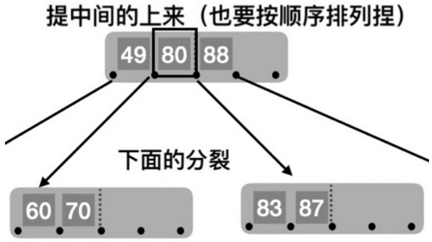
只有新元素一定是插入底层的终端结点，用查找来确定插入位置

插入后超过最大限制？$\begin{cases}\text{否} \rightarrow \text{直接插} \\ \text{是} \rightarrow \text{提中间关键字到父结点}\end{cases}$

$\boldsymbol{step1}$



$\boldsymbol{step2}$

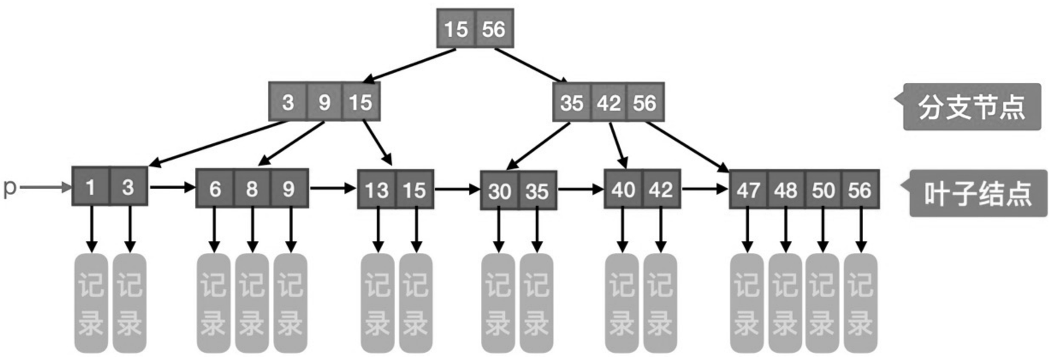


B+树概念

一颗 m 阶的 B+树满足：

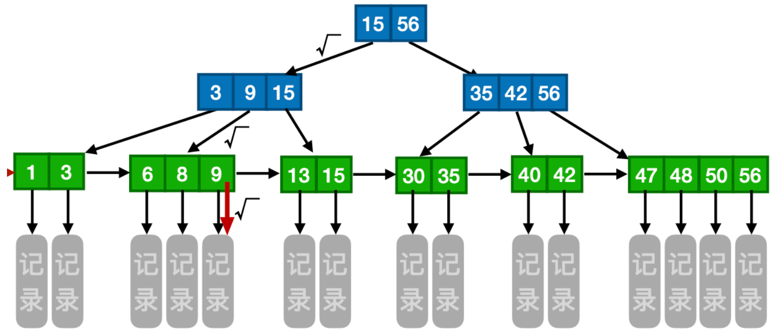
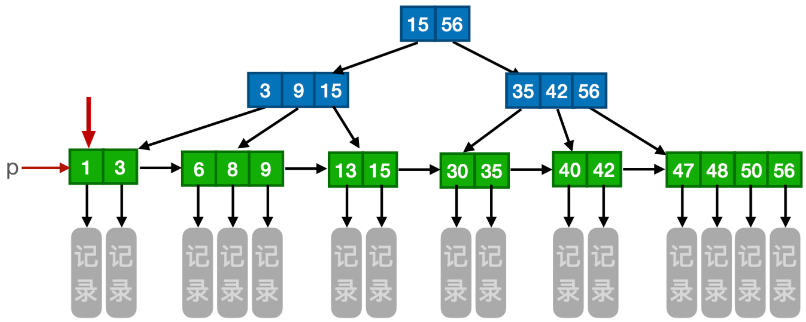
- 每个分支结点最多有 m 棵子树(孩子结点)。
- 非叶根结点至少有两棵子树，其他每个分支结点至少有$\lceil m/2\rceil$棵子树。
- 结点的子树个数与关键字个数相等。
- 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。
- 所有分支结点中仅包含它的各个子结点中关键字的最大值及指向其子结点的指针。

likethis



注：以上是一棵4阶B+树

B+树查找

查找方式		**特点**
多路查找	查找目标：9  	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	p 指针顺序查找  	

B/B- VS B+

	B/B-	B+
来源	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	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”)	

哈希查找思想、哈希查找性能

定义们

散列表/哈希表：一种数据结构，数据元素的关键字与其存储地址直接相关

同义词：若不同的关键字通过散列函数映射到同一个值，则称为同义词

冲突：通过散列函数确定的位置已经存放了其他元素，称为冲突

查找长度：需要对比关键字的次数称为查找长度

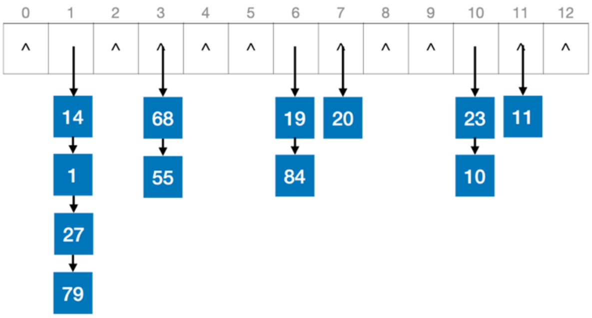
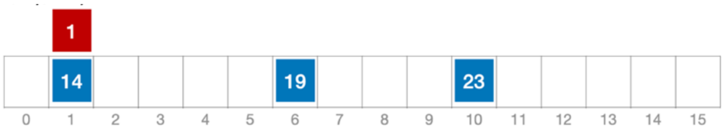
装填因子 α=表中记录数/散列表长度

常见散列函数

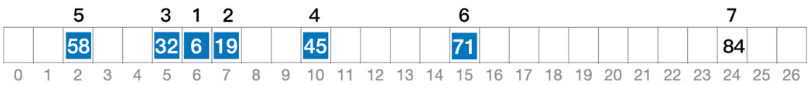
名称	定义	适用情况	举例
除留余数	散列表表长为 m，取一个不大于 m 但最接近或等于 m 的质数 p H(key) = key % p		散列表表长15，散列函数 H(key)=key%13
直接定址	H(key) = key 或 H(key) = a*key + b 其中，a 和 b 是常数。这种方法计算最简单，且不会产生冲突。	关键字的分布基本连续。 （若不连续，则空位多浪费存储空间）	存储同一个班级的学生信息，班内学生学号为 (1120112176~1120112205) H(key) = key - 1120112176
数字分析	选取数码分布较为均匀的若干位作为散列地址。	关键字的每一位出现的频率不同	手机号码，前三位大多一样。 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	取关键字的平方值的中间几位作为散列地址。 （取多少位要视实际情况而定）	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。	以身份证号为关键字存储学校的学生信息，设计散列函数。学生的生日、地址等大多集中在某几个数字

哈希冲突解决方法

- 开放定址法
  - 线性探测再散列
  - 二次/平方探测再散列
  - 伪随机探测再散列
- 链地址法
- 再哈希法

<table>
<tr><th colspan="2"></th><th></th><th>效率</th></tr>
<tr><td colspan="2">再哈希法</td><td>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，直到不冲突为止</td><td></td></tr>
<tr><td colspan="2">链地址法</td><td>如图，关键字为 {19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 H(key)=key%13
%13 后为{6, 1, 10, 1, 3, 7, 6, 1, 3, 11, 10, 1}

</td><td></td></tr>
<tr><td rowspan="4">开放定址法</td><td>定义</td><td>指可存放新表项的空闲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。
采用开放定址法时，删除结点不能只将结点置为空，需要做一个删除标记。</td><td></td></tr>
<tr><td>线性探测</td><td>发生冲突时，每次往后探测相邻的下一个单元是否为空
如{19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 key%13
$step1$.插入 1 时，1 位置已有关键字，因此向后搜索


$step2$. 顺序往后一个就找到空了

</td><td></td></tr>
<tr><td>二次/平方探测</td><td>发生冲突，向右一个，向左一个，向右四个，向左四个...到头了就从头开始
$d_i = 0^2, 1^2, -1^2, 2^2, -2^2, ..., k^2, -k^2$
如图，插入 84。上面的黑字是比较的次数

</td><td></td></tr>
<tr><td>伪随机探测</td><td>$d_i = 0, 5, 24, 11, ...$</td><td></td></tr>
</table>