笔记内容：面向912考试，基于912真题和THU期末题以及之前做的重点笔记综合而成的更加凝练的笔记。

适合于最后冲刺阶段经常翻看。

**第1章 绪论**

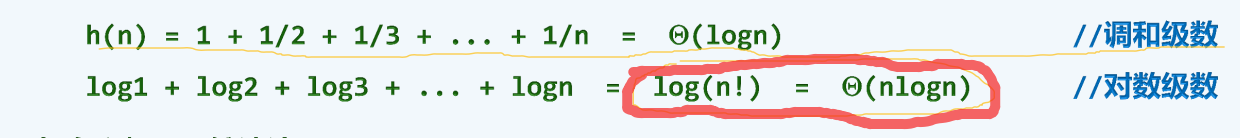
**c.大O记号**

  ，也即 logn < n^c

，也即多项式 n^c < 指数 2^n

调和级数 ：1+ 1/2+1/3+......+1/n=

对数级数: log1+log2+....+logn=

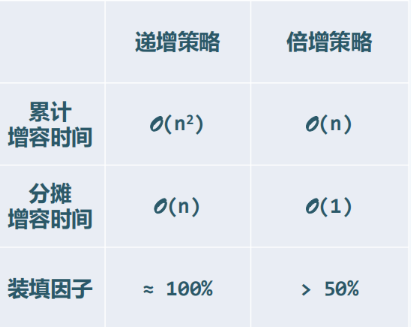


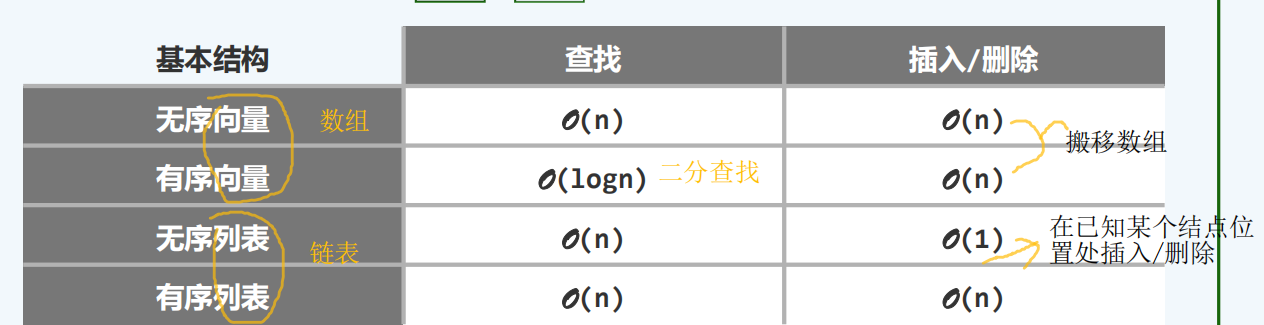
  ⭐

**向量**

**递增式扩容**：每次追加固定大小的容量

**加倍式扩容**：每次追加上一次容量的两倍





**二分查找和Fibnacci查找**

对有序向量做二分查找或Fibonacci 查找，就最坏情况而言，成功查找所需的比较次数与失败查找**相等。**

对二分查找和Fibonacci 查找，针对独立均匀分布二[0, 2n]内的整数目标，在固定的有序向量{ 1, 3, 5, ..., 2n - 1 }中查找或者要查找各元素的数值等概率独立均匀分布。若将平均的成功和失败查找长度分删记作 S 和 F，

则有 ：**(S + 1)∙n = F∙(n + 1)，或者记成 S×n + n = F×n + F ⭐ ⭐**

对n =Fib(k)-1有向序列做 Fibonacci 查找，比较操作的次数至多为**k-1**次。

**插值查找**

**原理**：元素均匀独立的随机分布，通过猜测轴点mi，可以极大提高收敛速度



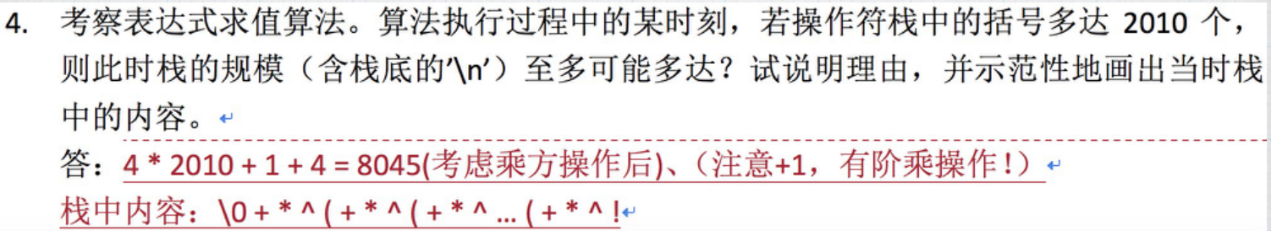
|  |  |  |
| --- | --- | --- |
| **性能**： | **缺点：** | **实际可行的方法：** |
| 最坏情况：O(n)  平均情况：每经一次比较，n缩至根号n。  整体为：O(loglogn) | 1.易受小扰动的干扰和“蒙骗”  2.须引入乘法、除法运算 | 大规模：插值查找  中规模：折半查找  小规模：顺序查找 |

**栈和队列**

**表达式求值**

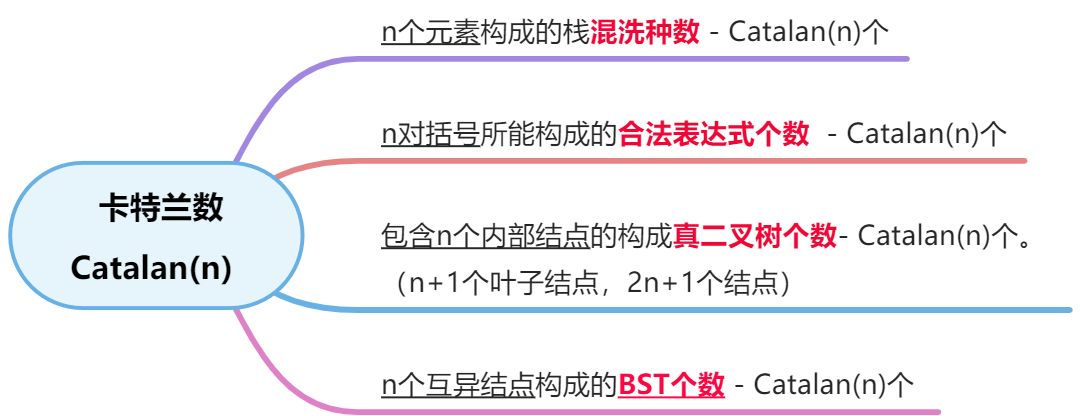
对不含括号的中缀表达式求值时，操作符栈的容量可以固定为某一常数。

RPN中各操作数的相对次序，与原中缀表达式完全一致。✔



**卡特兰数**

**卡特兰数**：



**二叉树**

**树的定义**

**树**： 连通无环图、极小连通图、极大无环图

树的边和结点数的关系： **e = n-1**

**结点深度：**从结点v到根结点的路径的边的个数 depth(v)

**结点高度**：max{ 所有子结点的高度 }+1

**树的高度**：树T中所有结点深度最大值成为该树的高度（注意树的高度是由某一叶子结点的深度决定的）

**约定：**根结点深度为0，仅含单个结点的树的高度为0，空树的高度为0

**问题**

1. **height( v ) + depth( v ) <= height( T ) 何时取等号 ?**

答：即height( v ) + depth( v ) <= height( T ) 取等号的时机应该是v节点处在这棵树的一条最长的由一个叶子到根的路径上所有节点构成的链上， 这条链上的所有节点都可以取等号，这条链的长度也等于树的高度

1. **在一棵树中,顶点p是顶点v的父亲,则它们的高度的关系是**

答：结点高度=max{ 所有子结点的高度 }+1； 所以h(p)>= h(v)+1。

通过【长子+兄弟】表示法，将这两个指针分别与二叉树的左、右孩子指针统一起来，可以进一步将原**有序多叉树**转换为**常规二叉树。**

**多叉树转二叉树步骤**：

1. 将同一结点的孩子结点之间用线两两连起来
2. 去掉除父结点与长子结点连线外父节点与其他孩子结点的连线 (步骤1,2即是用长子+兄弟表示法)
3. 将整棵树顺时针旋转45度即可得到二叉树

**二叉树**

**性质**

* 深度为k的结点，至多个 （即第k层，至多个个结点 ）
* 对于含n个结点，高度为h的二叉树来说
* 边数 e = n-1 = n1 + 2×n2
* 叶结点数 n0 = n2 + 1， 由此可知：n1 与 n0无关

有根有序的**多叉树**先序遍历与其所对应的**二叉树**的先序遍历序列完全相同。

在二叉树先序和层序遍历序列中，祖先节点一定位于其后代节点之前。

**真二叉树**

* **定义**：不含一度结点的二叉树称作真二叉树，即只含有度0或者度2结点的二叉树
* 真二叉树无n1节点，其内部节点数等于n2 = n0-1，即真二叉树内部节点数比叶子节点数少1。

**完全二叉树**

* 完全二叉树高度：
* 总节点数为n，叶子节点个数 =
* 当n为奇数，叶子节点个数 = 内部节点个数 + 1
* 当n为偶数，叶子节点个数 ＝ 内部节点个数

层次遍历中，【队列的最大规模】= **=** 叶子节点个数**，**且最大规模可能出现**两次**

即完全二叉树层次遍历【队列容量】 <= 叶子节点个数

* 当n为奇数，在 A[ n/2 ] 结点处于队首时出现一次
* 当n为偶数时，在 A[ n/2 -1 ] 和 A[ n/2 ] 结点处于队首时各出现一次，共计2次。

在整个遍历过程中，辅助队列的规模变化是单峰对称的，即

* { 0, 1, 2, ..., (n + 1)/2, ..., 2, 1, 0 } （n 为奇数时）
* { 0, 1, 2, ..., n/2, n/2, ..., 2, 1, 0 } （n 为偶数时）

**哈夫曼Huffman编码**

Huffman编码树是一颗最优带权编码树



* 不同频率数 = 叶子节点数
* 哈夫曼编码最大长度 = 不同频率数 - 1
* 互换深度不同但是频率相同的节点，平均编码长度不变，最优带权编码树性质不变。
* 哈夫曼树距离根节点深度更小的节点的权值 **>=** 深度更大的节点的权值。

**🌞重构**

**总结 - 任意给出树的两种遍历序列，能否确定唯一的一颗二叉树或二叉树的另一种遍历序列的一些结论如下：**

* **中序 + 先序|后序|层序 —> 唯一二叉树** （中序带谁都能组队打boss）

（中序 + 层序 -> 唯一二叉树）2016年已经考过证明

* **层序 + 先序|后序 —❌ —> 唯一二叉树**
* **先序+后序 —❌ —> 唯一二叉树**
* **(先序+后序)&&真二叉树 —> 唯一二叉树** （先序和后序在一起要加条件）
* **先序+后序 —> 层序**

（先序 + 后序 ->层序）2019年已经考过判断，2020年已经考过证明

**判断与证明 给定一棵二叉树的先序和后序遍历序列，通过先序和后序遍历序列能否确定唯一的一颗二叉树？若可以给出证明，不可以则说明理由。**

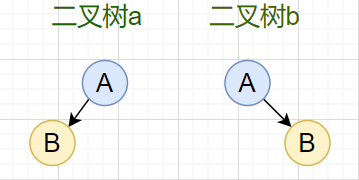
**解答**：**【先序 + 后序 】 不能唯一**确定一颗二叉树。

因为当结点的左子树或者右子树为空时，会产生歧义

|  |  |  |
| --- | --- | --- |
|  | 先序 | 后序 |
| 右子树为空 | r L | Lr |
| 左子树为空 | r R | LR |

分离根结点之后的部分究竟是左子树还是右子树是无法判断的，从而会得到局部不同的两个子树，二叉树的唯一性无法保证。

如：先序 AB , 后序 BA



所以如果整颗二叉树都没有度为1的结点，那么就不会产生歧义，所以 **(先序+后序)&&真二叉树 —> 唯一二叉树。**

**判断与证明 给定一棵二叉树的中序和后序遍历序列，通过中序和后序遍历序列能否确定唯一的一颗二叉树？若可以给出证明，不可以则说明理由。**

**解答：【中序+后序】**能够确定唯一的一颗二叉树，证明如下：

二叉树的中序遍历序列为：\*\*\*\*L\*\*\*\*，V，\*\*\*\*R\*\*\*\*

后序遍历序列为：\*\*\*\*L\*\*\*\*，\*\*\*\*R\*\*\*\*，V

对某一子树，根节点v0

① 若v0没有孩子，显然无歧义

② 若v0的左孩子为lc，右孩子为rc

其中序 lc\*\*\*\*pre(L)\*\*\*\*，v0，rc\*\*\*\*pre(R)\*\*\*\*

其后序 \*\*\*\*post(L)\*\*\*lc，\*\*\*\*post(R)\*\*\*\*rc，v0

从后往前扫描后序遍历，根据后序遍历的根节点可以在中序遍历找到对应根节点位置，并将中序遍历划分为左右子序列，然后再分别递归左右子序列。

③ 若v0的左孩子为lc，无右孩子

其中序 lc\*\*\*\*pre(L)\*\*\*\*，v0

其后序 \*\*\*\*post(L)\*\*\*lc，v0

从后往前扫描后序遍历，根据后序遍历的根节点可以在中序遍历找到对应根节点位置，其中序遍历只有左子序列，然后再递归左子序列。

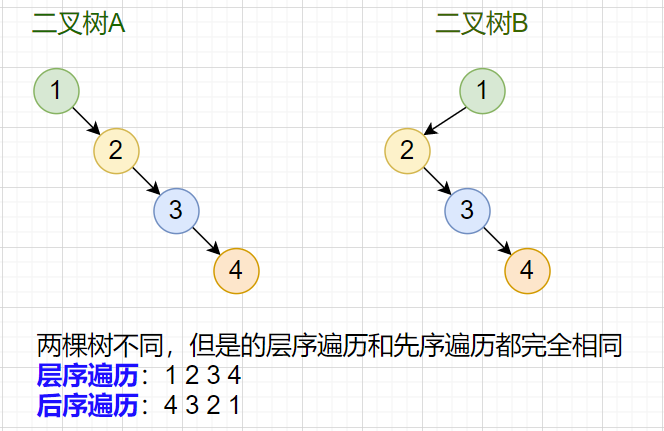
④若v0的右孩子为rc，无左孩子

与第③情况完全对称

综上，**【中序+后序】**能够唯一确定一颗二叉树。

**判断与证明 给定一棵二叉树的层序和后序遍历序列，通过层序和后序遍历序列能否确定唯一的一颗二叉树？若可以给出证明，不可以则说明理由。**

**解答：【层序+后序】**不能够确定唯一的一颗二叉树，举例如下：



**有关树的算法题**

* 先序遍历的第K个结点
* 中序遍历的第k个结点（2020年已考）
* 后序遍历的第K个结点（2019年已考）
* 中序遍历的第一个结点
* 后序遍历的第一个结点 （2017年已考）
* 结点先序遍历的直接后继
* 结点中序遍历的直接后继
* 结点后续遍历的直接后继（2017年已考）
* 逆波兰表达式建树（2016年已考）
* 逆波兰表达式求值 【[Leetcode 150. 逆波兰表达式求值](https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/" \t "_blank) 】
* spaly伸展树伸展算法（2020年已考）
* 前序和中序遍历序列输出后序
* 前中后序遍历非递归写法
* 归并排序
* 快速排序
* K-选取算法

**先遍历的第k个结点**

1.若 k == 1：直接返回根节点；

2.若 k <= leftSize +1 ：在左子树上寻找第 k-1 个节点；

3.若 k > leftSize +1： 在右子树上寻找第 k-1 - leftSize 个节点；

struct BinNode {//二叉树节点

BinNodePosi(T) lc;

BinNodePosi(T) rc;

int size; //当前节点和孩⼦总数

};

BinNodePosi(T) PreSearchK( BinNodePosi(T) root, int k){

int leftSize = root->lc->size;

if( k == 1)

return root;

else if(k <= leftSize+1)

return PreSearchK(root->lc,k-1);

else if(k > leftSize+1)

return PreSearchK(root->rc,k-1-leftSzie);

}

**中序遍历的第k个结点（2020年已考）**

1.若 k == leftSize + 1：直接返回根节点；

2.若 k < leftSize + 1：在左子树上寻找第 k 个节点；

3.若 k > leftSize + 1：在右子树上寻找第 k-leftSize -1个节点。

struct BinNode {//二叉树节点

BinNodePosi(T) lc;

BinNodePosi(T) rc;

int size; //当前节点和孩⼦总数

};

BinNodePosi(T) searchK( BinNodePosi(T) root, int k){

int leftSize = root->lc->size;

if(leftSize + 1 < k)

return searchK(root->rc, k-leftSize-1);

else if(leftSize + 1 == k)

return root;

else

return searchK(root->lc, k);

}

**后序遍历的第K个结点（2019年已考）**

1.若 root.size == k：直接返回根节点；

2.若 k <= leftSize ：在左子树上寻找第k个节点；

3.若 k > leftSize ： 在右子树上寻找第 k-leftSize 个节点；

struct BinNode{

int size; //当前节点和孩⼦总数

BinNode \*lchild,\*rchild;

};

BinNode \*rank(BinNode\* root, int k){

//检查参数是否合法

if(root == NULL || root->size < k) return NULL;

if(root->size == k) return root;

//如果没有左子树，则直接进入右子树搜索

if(root->lchild == NULL)

return rank(root->rchild, k);

int leftSize = root->lchild->size;

if(k <= leftSize) //左子树不为空，且根据左子树规模和目标比较，进入左子树搜索

return rank(root->lchild, k);

else

return rank(root->rchild, k-leftSize);

}

**总结**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **先序第K个结点** | **中序第K个结点** | **后序第K个结点** |
| **根结点** | k == 1 | k == leftSize +1 | k == root.size |
| **左子树** | k <= leftSize +1  递归：k = k-1 | k < leftSize +1  递归：k = k | k <= leftSize  递归：k = k |
| **右子树** | k > leftSize +1  递归：k = k-leftSize-1 | k > leftSize +1  递归：k = k-leftSize-1 | k > leftSize  递归：k = k-leftSize |

**中序遍历的第一个结点**

一直往左遍历，直到某结点没有左孩子，返回该结点。

struct binarytree{

struct binarytree \*parent；

struct binarytree \*lc；

struct binarytree \*rc；

};

//返回以当前结点为根结点的子树的中序遍历的第一个结点

binary\* binarytree::first() {

while(this->lc) {

this = this->lc;

}

return this;

}

**后序遍历的第一个结点 （2017年已考）**

优先遍历左孩子，没有左孩子就遍历右孩子，直到左右孩子都没有的结点，返回该结点。

struct binarytree{

struct binarytree \*parent；

struct binarytree \*lc；

struct binarytree \*rc；

struct binarytree\* first();

};

//返回以当前结点为根结点的子树的后序遍历的第一个结点

binary\* binarytree::first() {

while(this->lc || this->rc) {//优先遍历左子树，左子树为空时，才转到右子树

if(this->lc) this = this->lc;

else this = this->rc;

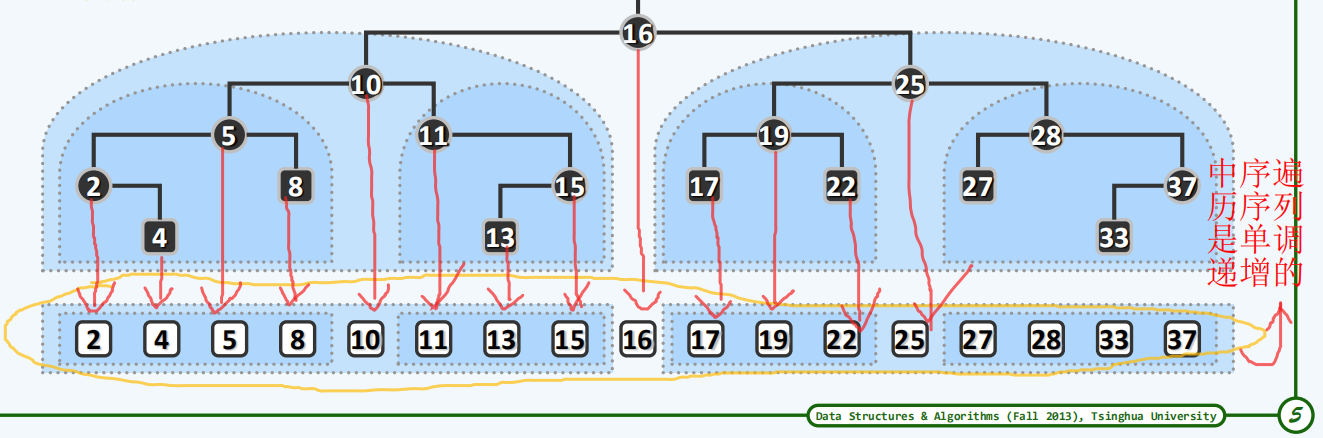
}

return this;

}

**结点先序遍历的直接后继**

* 先序遍历直接后继优先为其左孩子，无左孩子就是右孩子
* 无左右孩子
  + 先序遍历后继为把它包含在左子树中且右孩子不为空的最低祖先（如下图数值为8，13，17的结点）



struct BinNode { //二叉树节点

BinNodePosi(T) lc;

BinNodePosi(T) rc;

BinNodePosi(T) preTralsucc();

}

template <typename T>

BinNodePosi(T) BinNode<T>::preTralsucc()() { //定位节点v的先序遍历的直接后继

BinNodePosi(T) s = this; //记录后继的临时变量

if( lc ){ return lc;

}else if(rc){ return rc;

}else{//左右孩子均空，找到把它包含在左子树中且右孩子不为空的最低祖先

while(s){

while( IsRChild(s) ) s = s->parent;

s = s->parent;

if( !s && s->rc ) return s->rc;

}

return NULL;

}

}

**结点中序遍历的直接后继**

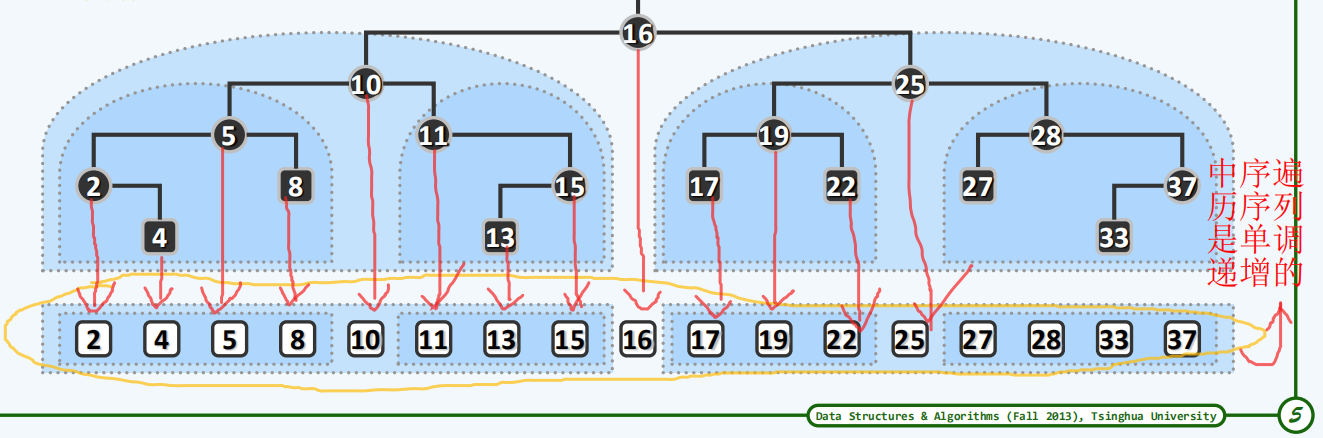
参考邓书代码5.16

① 当前结点的右子树不为空

* 直接后继为右子树中最靠左的结点 （如下图数值为25的结点）

② 当前结点的右子树为空

* 当前节点为左孩子 -> 其后继就是其父亲 （如下图数值为27的结点）
* 当前结点为右孩子 -> 其后继就是将当前结点纳入其左子树的那个最低的祖先结点 （如下图数值为15的结点）



struct BinNode { //二叉树节点

BinNodePosi(T) lc;

BinNodePosi(T) rc;

BinNodePosi(T) succ();

}

template <typename T>

BinNodePosi(T) BinNode<T>::succ() { //定位节点v的直接后继

BinNodePosi(T) s = this; //记录后继的临时变量

if ( rc ) { //若有右孩子，则直接后继必在右子树中，具体地就是

s = rc; //右子树中

while ( HasLChild ( \*s ) ) s = s->lc; //最靠左（最小）的节点

} else { //否则，直接后继应是“将当前节点包含于其左子树中的最低祖先”，具体地就是

while ( IsRChild ( \*s ) ) s = s->parent; //逆向地沿右向分支，不断朝左上方移动

s = s->parent; //最后再朝右上方移动一步，即抵达直接后继（如果存在）

}

return s;

}

**结点后续遍历的直接后继（2017年已考）**

1.如果当前结点是左孩子，其后序遍历后继为以其兄弟结点为根节点的子树的后序遍历的第一个结点

2.如果当前结点是右孩子，其后序遍历后继为其父亲

struct binarytree{

struct binarytree \*parent；

struct binarytree \*lc；

struct binarytree \*rc；

struct binarytree\* first();//二叉树后序遍历的第一个结点

};

struct realbinarytree{

struct binarytree p；

struct binarytree\* next();

};

//返回以当前结点在后序遍历当中的后继结点

binarytree\* realbinarytree::next(){

if(!this->parent) return NULL;//后继为空

if(this == this->parent->lc && !this->parent->rc){ //当前结点是左孩子并且其右兄弟不为空

return this->parent->rc->first();

}else{//【是右孩子】或者【是左孩子但右兄弟为空的情况】

return this->parent;

}

}

//利用first()和next()函数对二叉树进行后序遍历，时间复杂度为O(n)

//从结点x开始对此棵二叉树做后序遍历

//如果x是此棵二叉树后序遍历的第一个结点，下面的函数就能遍历完整颗二叉树

void PostTravel(realbinarytree\* x) {

while(!x){

visit(x);

x->next();

}

}

**逆波兰表达式建树（2016年已考）**

逆波兰表达式建树和逆波兰表达式求值的思想整体一致，只不过逆波兰表达式建树还需要把每一个值包装在一个结点上。

具体步骤：

* 扫描序列，操作数入栈。
* 当遇到符号，取出栈顶两个元素，作为此符号结点的左右子结点。
* 然后把这个结点入栈。
* 直到扫描完序列，栈变空

BinNode\* RPN2BinTree(char\* s) {

stack<BinNode\*> sta;

char\*p = s;

while(\*p != '\0'){

BinNode\* node = new BinNode(\*p);

if(\*p == 操作符 ) {

BinNode\* rc = sta.top(); sta.pop();

BinNode\* lc = sta.top(); sta.pop();

node->lc =lc;

node->rc =rc;

}

sta.push\_back(node);

}

return sta.top();

}

**spaly伸展树伸展算法（2020年已考）**

函数：splyto(a, x) ，a为x的祖先，通过zigzag操作将x调整变为a的孩子，若a为NULL，x调为根节点；

1.x已经使a的孩子了  —> 什么都不用做

2.x是a的孙子   —> 只需一次旋转

① x是左孩子   —> zig旋转，使x称为a的孩子

② x是右孩子 —> zag旋转，使x称为a的孩子

3.x不是a的直接孙子  —> splay算法的双层伸展 —>  直到x为a的孩子

  设x的父亲为p，p的父亲为g

① px为左左 —> zig，zig旋转

② px为右右 —> zag，zag旋转

③ px为左右 —> zag，zig旋转

④ px为右左 —> zig，zag旋转

void splyto(BinNodePosi(T) a, BinNodePosi(T) x){

if(a == x->parent) return;//x已经使a的孩子了，什么都不用做

else if(a == x->parent->parent){//x是a的孙子，只需一次旋转

if(isLchild(x)) zig(x->parent);

else zag(x->parent);

}else {//x不是a的直接孙子

BinNodePosi(T) p = x->parent;

BinNodePosi(T) g = p->parent;

if(isLchild(p)&&isLchild(x)) {zig(g);zig(p);}

else if(isRchild(p)&&isRchild(x)) {zag(g);zag(p);}

else if(isLchild(p)&&isRchild(x)) {zag(p);zig(g);}

else if(isRchild(p)&&isLchild(x)) {zig(p);zag(g);}

splyto(a,x);//继续检查a和x的关系,做递归的旋转

}

}

**二叉搜索树**

规模为 n 的任何两棵等价二叉搜索树，至多经过 **2n - 2** 次旋转调整，即可彼此转换。O(n)

**等价的BST**：就是拓扑结构不同，但是中序遍历序列一样的不同棵的BST

在 **kd-search** 中，查找区间 R 与任一节点的 4 个孙节点（假设存在）对应区域最多有 2 个相交。

以下数据结构中，空间复杂度不超过线性的有（ ）。（2014期末）

- A）2d-tree

- B）3d-tree

- C）2D range tree

- D）interval tree

- E）segment tree

- F）priority search tree

选A，B，D，E，F 除了C其他都选

2D range tree的空间复杂度时O(nlogn)

**AVL**

**重平衡**

平衡因子

对于AVL树来说

高为 **h** 的AVL树至少包含 **fib(h+3) - 1**个结点；反过来，包含 n = fib(h+3) - 1结点的AVL树最大高度为 h。 ⭐

在高度为 h 的AVL树中，叶子结点最小深度为

如 1596 <= n <2583，h = 14 ；故当n = 2020，最大高度h = 14，叶子结点最小深度 = 7

**插入和删除总结**

|  |  |  |
| --- | --- | --- |
|  | **插入** | **删除** |
| 失衡节点个数 | 至多Ω(logn)个 | 至多1个，O(1) |
| 全树恢复平衡旋转调整次数 | 至多两次旋转，O(1)，旋转后高度复原 | 至多**h/2**次旋转=Ω(logn)，旋转后高度可能减少 |
| 总体复杂度 | O(logn) | O(logn) |
| **注意：**插入并不一定高度不变，有可能不需要做旋转调整，高度就增加1。  删除也不一定要做旋转操作高度才会较少1，有可能不需要做旋转调整，高度就减少1 | | |

将n个结点按递增/递减次序插入AVL树，且n满足 ，则此AVL高度为 。⭐

当  时，此AVL为高为h的满树，且此时递增/递减次序构成的两颗AVL树拓扑结构相同。

**伸展树**

**逐层伸展**

最坏情况分摊时间复杂度 Ω(n)

低效率根源是全树的拓扑结构呈单链条结构，等价于一维列表；

**双层伸展**

折叠效果：一旦访问坏节点，对应路径长度将随折叠而**减半**

单趟伸展操作，分摊复杂度为

**伸展树综合评价**

伸展树所有基本操作接口的分摊时间复杂度(不论是否满足局部性原理的访问或是完全理想随机的访问)，均为 O(logn)。

最底部叶结点被访问并经过spaly调整后，伸展树的高度可能增加、减少或者不变。（注意：并不是只有减少这种情况）

将n个结点插入splay树，不管这n个结点是否按照单调次序插入的，能得到splay树是单链的情况有很多种，高度为n-1，且第一次访问最底层结点经过n-1次旋转。

将2014个数插入splay，第一次访问经过2013次旋转，则是单调插入的。（2014期末补充） ❌

将{0,1,2,.....,2018}插入一棵空的伸展树后若树高为2018 ，则上述词条必是按单调次序插入的。（2019.1期末）❌

**B树**

分支数x的范围 ： ，m阶B树也称作 树 ⭐

关键码数N的范围： （比分支数少1）

**搜索**

失败查找必然终止于外部结点，复杂度

N个内部结点，N种成功可能；N+1个外部结点，N+1种失败可能

**树高**

含有N个关键码m阶B树的树高h的范围 ：   ⭐

相比于BBST 降约至BBST的范围：

高度为 h 的 m 阶 B‐树所含关键码的范围：

**插入**

上溢可能持续发生，并逐层向上传播，最坏抵达树根 ，最坏时间复杂度

当抵达树根时，被提升的关键码将作为新的树根，导致B树树高增加，这也时B树树高增高的唯一可能。

将 N 个关键码按随机次序插入 B 树，则期望的分裂次数为 。 ⭐

分裂和合并在最坏情况下均需 的时间，然而在B-树任一足够长的生命期内，分裂与合并的总次数是O(n)级别的，分摊下来就是**O(1)**时间的。 ⭐

对于N个互异的关键码，插入一棵初始为空的m阶B树种，按照单调次序（单调增加或单调减少）插入，可以使得B-树高度最大。

**删除**

**下溢和合并**

* 刚发生下溢的结点V必恰好包含  个关键码和 个分支
* 左兄弟或右兄弟存在，且至少包含 个关键码
  + 从父结点那里借关键码过来，父结点再从兄弟结点那里借关键码过来
* 左兄弟或右兄弟不存在，或者其包含的关键码均不足 个
  + 从父结点借一个关键码，然后与其兄弟结点合并成一个结点
    - 下溢可能向上持续传播，最大为O(h)次，，最坏时间复杂度 。
    - 可能导致根节点被合并，从而导致全树高度减一

**其他结论**

B树的任一非叶节点内，每个关键码都存在直接后继，且必然来自某个叶节点。

**红黑树**

统一增设外部结点NULL，使之称为真二叉树。

红黑树等价于一颗 **（2，4）B树**，高度 h = O(logn)

**红黑树的定义：**

1. 树根：必为黑色
2. 外部结点：均为黑色
3. 其余结点：若为红，则只能是黑孩子  （红结点的孩子和父亲都必为黑色）
4. 外部结点到根：途中黑结点数目相等

**红黑树是一种半平衡二叉搜索树，也就是其最大高度不超过最小高度的两倍。⭐**

当某条路径最短时，这条路径必然都是由黑色节点构成。

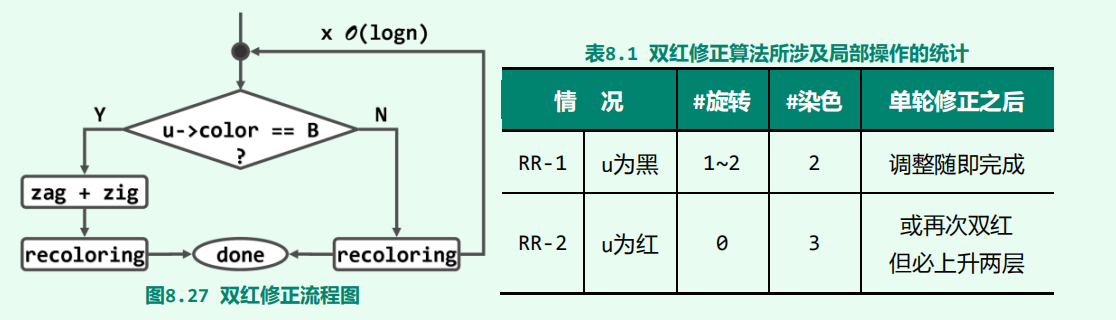
当某条路径长度最长时，路径上红色节点数量 = 黑色节点数量。该路径长度为两倍黑色节点数量，也就是最短路径长度的2倍。

对于**黑节点**来说，其 黑高度+黑深度=全树的黑高度；

对于**红节点**来说，其 黑高度+黑深度=全树的黑高度 - 1。

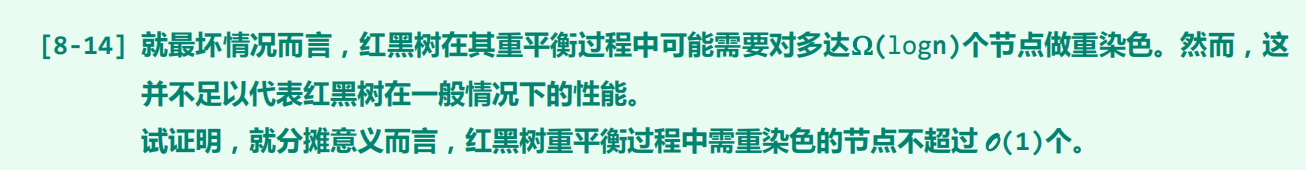
所以，红黑树上所有节点的黑深度和黑高度之和并不**都**相等，红黑结点要区分。

**节点插入算法与双红现象**

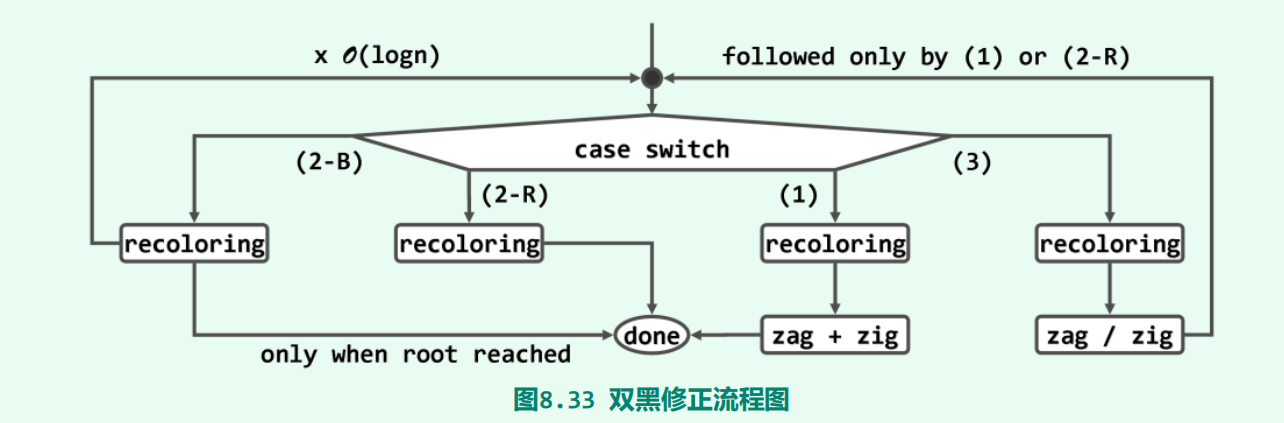


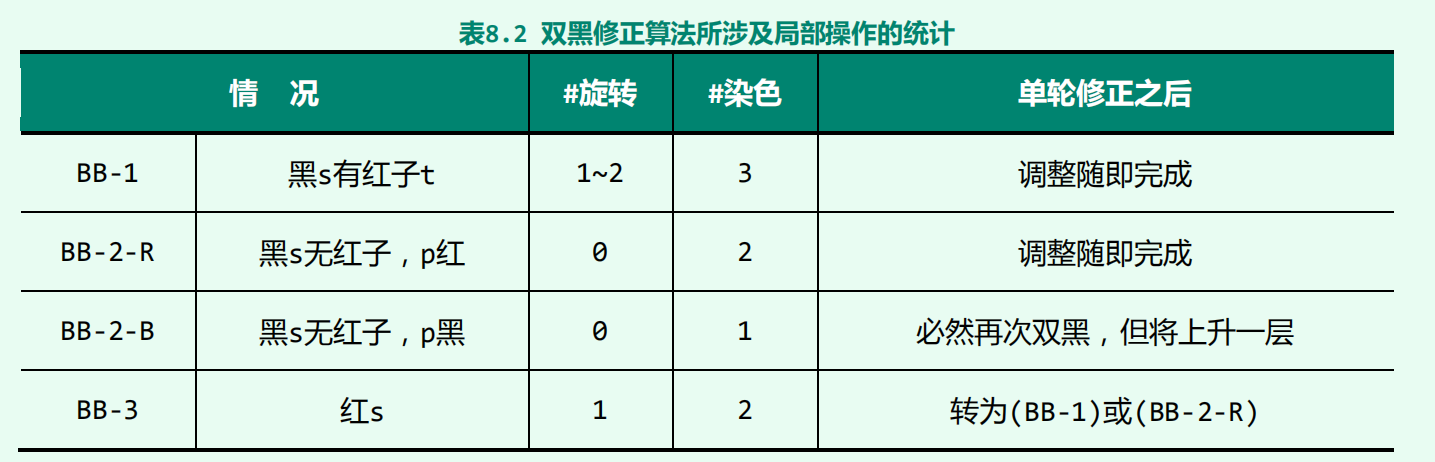
红黑树的黑高度增加，只能是插入过程中一直发生双红修正RR-2的情况，此情况下不会发生“3+4”重构（即拓扑结构变换），无任何结构调整，只会发生重染色（每次染3个，最多染O(logn)个）。

红黑树的插入或删除操作，都有可能导致Ω(logn) 个节点的颜色反转。

但是，就分摊意义而言，红黑树重平衡过程中需重染色的节点不超过 O(1)个。

**节点删除算法与双黑现象**





红黑树结构，如果不显式记录颜色，通过隐式记录应该如何操作？（2020.1期末）

红黑树是一种半平衡二叉搜索树，也就是其最大高度不超过最小高度的两倍。

引入红色其实也是为了让其左右子树的高度保持相对的平衡。

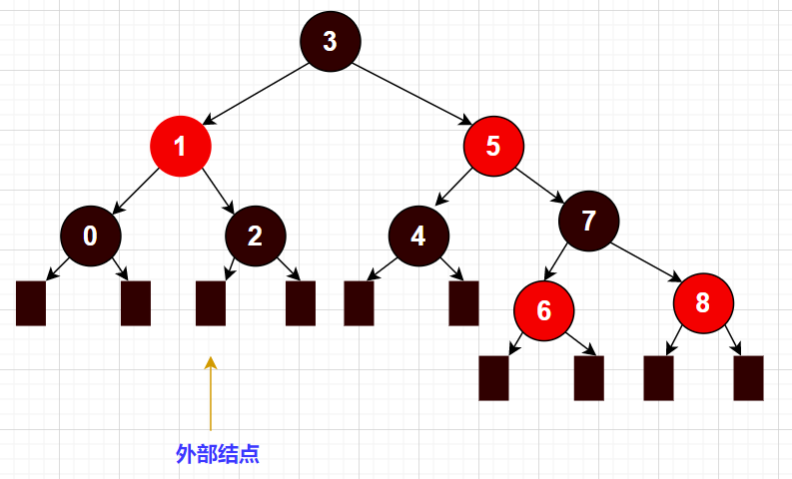
现在不显示的记录颜色，也就是不引入红色了，那么就需要引入类似于AVL树的平衡因子，对于红黑树来说这里的平衡因子可以定义为以每个结点为根的子树其最大高度不超过最小高度的两倍。

在任何重构操作（删除、插入）时需要保证每个结点都保持平衡，当结点不平衡时，就需要通过旋转使其恢复平衡。

依次插入[0,N)，写出N=9的红黑树 ⭐ 要自己手动画一下！！！



最后所得👇



考查含有N 个内部节点的红黑树 （习题解析[8-13]） ⭐

红黑树等价于一颗 **（2，4）B树**

**最小黑高度**

**最大黑高度**

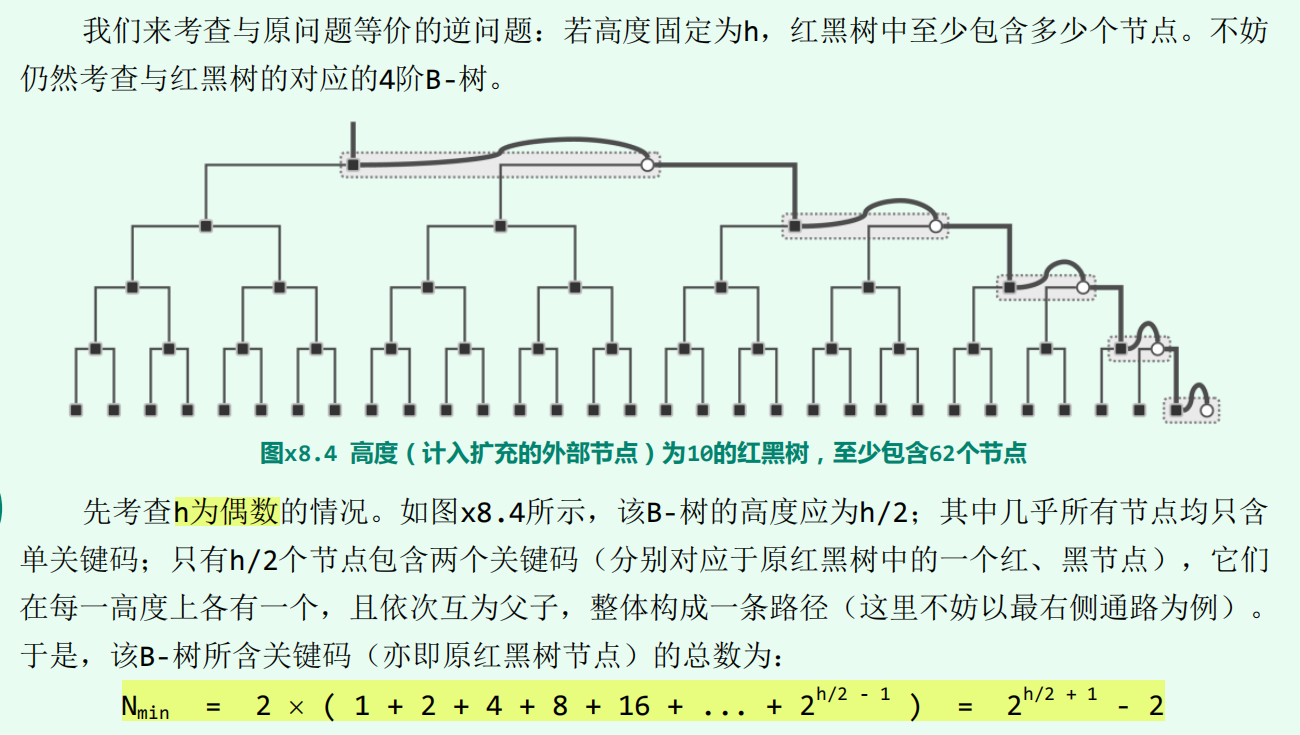
与之相对应的4阶B-树，则该B-树中存放的关键码恰有N个

**最小高度 hmin**

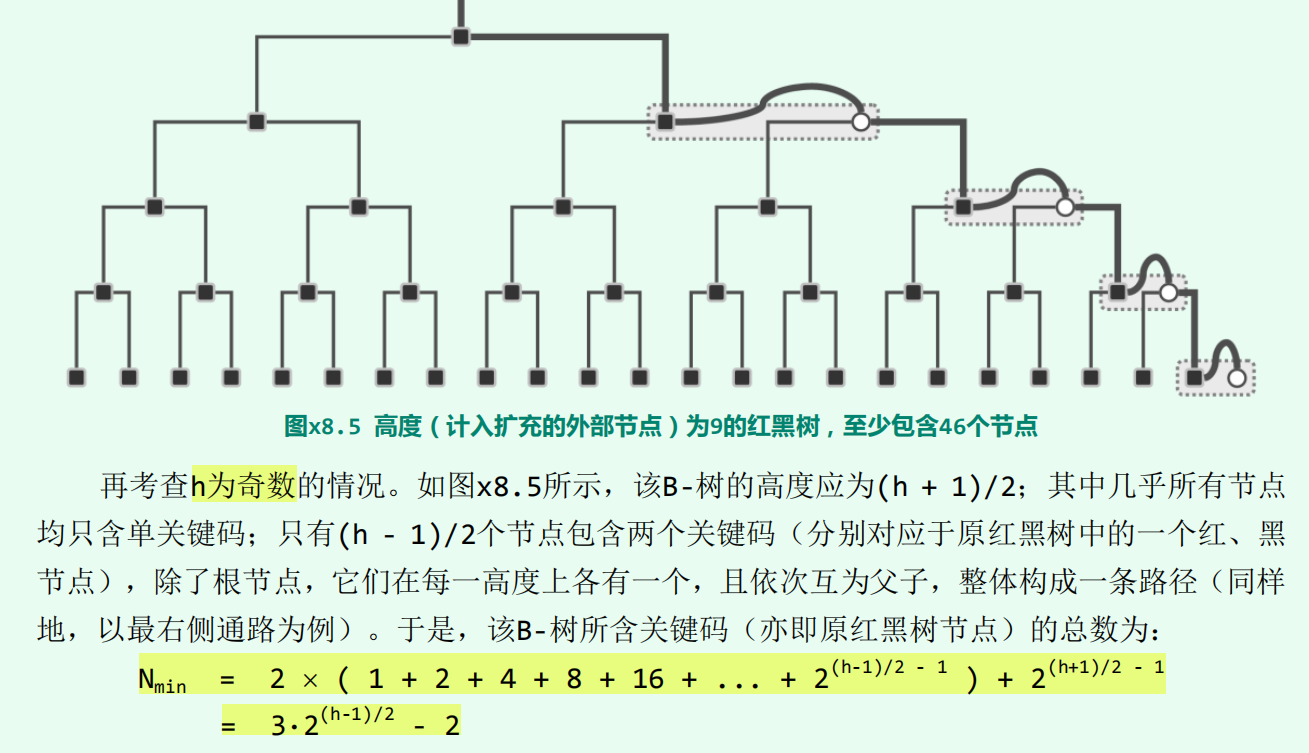
**最大高度 hmax**

考查与原问题等价的逆问题：若高度固定为h，红黑树中至少包含多少个节点

* 当h为偶数



* 当h为奇数



**各种搜索树大比拼**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **AVL树** | **splay伸展树** | **红黑树** | **m阶B树** |
| **插入** | 整体为O(logn)。  失衡结点可能多达Ω(logn)，但旋转操作仅为O(1)。 | 均摊复杂度O(logn) | O(logn)  至多O(logn)次重染色+2次旋转 |  |
| **删除** | O(logn)。  失衡结点仅为O(1)，但旋转操作可能多达Ω(logn)。 | 均摊复杂度O(logn) | O(logn)  至多O(logn)次重染色+3次旋转 |  |
| **优点** | 无论查找、插入或者删除，最坏情况下时间复杂度为O(logn)，空间复杂度O(n)。 | 1.无需记录平衡因子，相比AVL树编程实现简单易行。而其分摊复杂度O(logn)也与AVL树相当。  2.利用局部性，缓存命中率高时，效率甚至会更高。 | 单次操作树拓扑结构变化小，仅为O(1) | 针对外部查找，大大减少I/O次数  特别适用于对外存的批量访问 |
| **缺点** | 1.借助高度或平衡因子，需要额外封装  2.删除后的旋转，成本不菲  3.单次删除操作调整后，全树的拓扑结构的变化量可能达到Ω(logn)。 | 不能杜绝单次最坏情况的出现，不适合对效率敏感的场合。 |  |  |

**图**

采用**邻接表**实现的BFS和DFS时间复杂度是O(N+E)

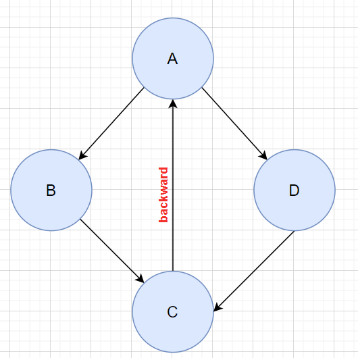
而采用**邻接矩阵**实现的BFS和DFS时间复杂度是O(N^2)

**广度优先搜索**

BFS树中从s到v的路径，即是s到v所有边中的最短路径

**深度优先搜索**

有向图DFS后有k条边被标记为后向边，图中一定有环路，但是不一定恰好有k各环路，环路数应该大于等于k。⭐



有向图的DFS不仅在起点任意，而且每⼀步迭代往往都会有多个顶点可供选择，

* 故所生成的DFS森林并不唯⼀确定；
* 且其中所含 树边，前向边，后向边，跨越边的数量都有可能不同；
* 只要起始于G中某顶点s的某次DFS所生成的是⼀棵树，则起始于s的任何⼀次DFS都将生成⼀棵树；
* s在树中的度数必然固定。

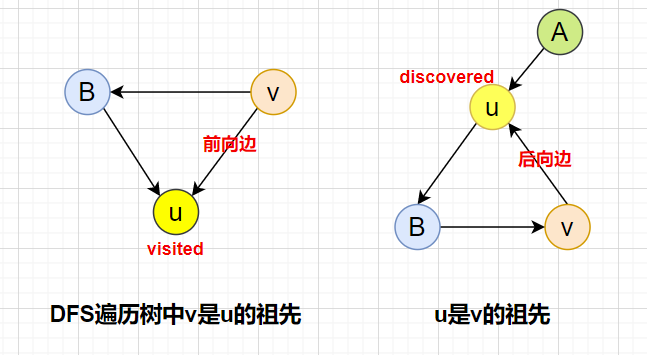
在图DFS() 算法中的default分支，将dTime(v)<dTime(u) 改为dTime(v)<fTime(u) 同样可行。✔

**DFS过程中何时标记前向边？何时标记后向边？**

设两个顶点分别为u和v

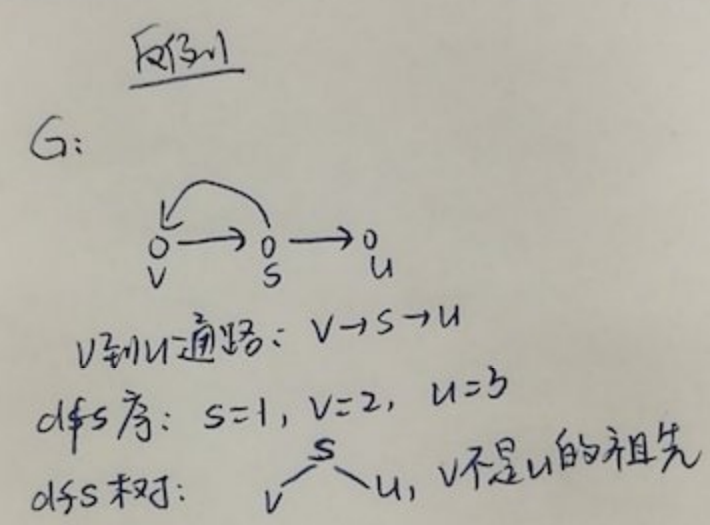
发现当前顶点v，枚举其邻居u的时候

* 发现u被标记为VISITED状态，且v的发现时间早于u的，则此条边v->u被标记为前向边，v为u的祖先
* 发现u被标记为DISCOVERD状态，则此条边u->v被标记为后向边，u为v的祖先



在有向图 G 中，存在一条自顶点 V 通向 u 的路径，且在某次 DFS 中有 **dTime[v]<dTime[u]**，则在这次 DFS 所生成的 DFS 森林中，**v 是否一定是 u 的祖先**？若是，请给出证明；若不是，请举出反例。

虽然有dTime[v]<dTime[u]，但生成的DFS森林中，v 不一定u 的祖先。反例如下👇



**拓扑排序**

* 可以拓扑排序的有向图，必定无环
* 任何DAG(有向无环图)，都至少存在一种拓扑排序

**构造拓扑排序**

DFS搜索过程中各顶点被标记为VISITED（回溯）的次序，恰好（按逆序）给出了原图的一个拓扑排序。⭐

如果此算法用栈实现，则拓扑排序就是栈内各顶点自顶向底排序。

**优先级搜索 PFS**

PFS过程中，尽管每一步迭代都可能多次调用prioUdpater() ，但累计不过O(e)次。

**最小支撑树算法Prim算法**

算法策略：基于贪心的迭代式算法。

过程如下：

1. 任选一个顶点A作为初始的子树T，T{A}
2. 从A的边中选出一条最短的边AB加入树T中 ，T{A，B}
3. 从与结点A，结点B相连的还未纳入T中的边中选一条最短的，如AD纳入T中，T{A,B,D}
4. 仿照前面迭代...........

复杂度O(n^2)，作为PFS搜索的特例，Prim算法的效率可借助优先级队列进一步提高。

**最短路径算法Dijkstra算法**

选定一个顶点s，计算图中各顶点ui到顶点s的最短距离

Dijkstra算法与Prim算法类似，不同的是每一步迭代时考虑的式ui到顶点s的距离，而不再是其到树Tk的距离。

复杂度O(n^2)，作为PFS搜索的特例，Dijkstra算法的效率可借助优先级队列进一步提高。

权值都为正整数的图能用Dijkstra算法构造出最短路径。

当分叉数 m = e/n + 2 时 ⭐

基于d叉堆实现的PFS优先级搜索 - 如：Prim算法、Dijkstra算法性能能达到最优

* 对于稀疏图，接近于 O(nlogn)
* 对于稠密图，接近于 O(e) —— 效率大大增加

**词典-散列-哈希**

**跳转表**

跳转表的一些结论 ⭐⭐

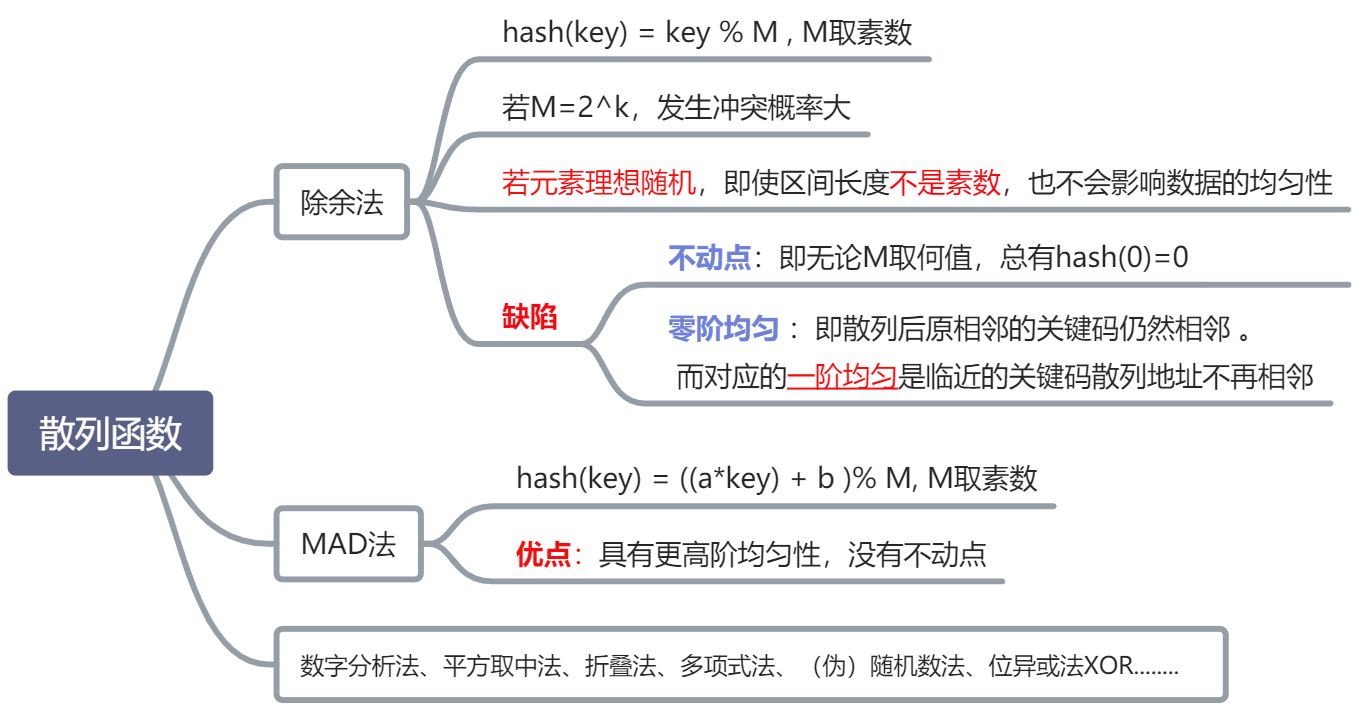
**空间复杂度**

* 期望的塔高 E(h) = 2
* 空间性能 O(n)

**时间复杂度**

* 跳转表期望高度O(logn)
* 同一高度上，彼此紧邻的塔顶结点数目期望值为2，整个查找过程中所作横向跳转期望次数O(logn)
* 查找，添加，删除 O(logn)

**散列函数**



**排解冲突**

**封闭定址**：桶单元只能存放与这个桶单元地址相同的词条，也即词条属于哪个地址，事先已经由散列函数注定。

**开放定址**：散列地址空间对所有词条开放。

**闭散列**：可用的散列地址仅限于散列表所覆盖的范围之内。

**开散列：**可用的散列地址除了散列表，还引入了别的附加空间，如多槽位，独立链，公共溢出区。

**相比开散列，闭散列的优势 ⭐**

* 闭散列结构本身保持简洁，在散列内部解决冲突，无需附加额外空间
* 闭散列可以更有效地利用局部缓存

**开发定址 —— 线性试探、平放试探、再散列、重散列等**

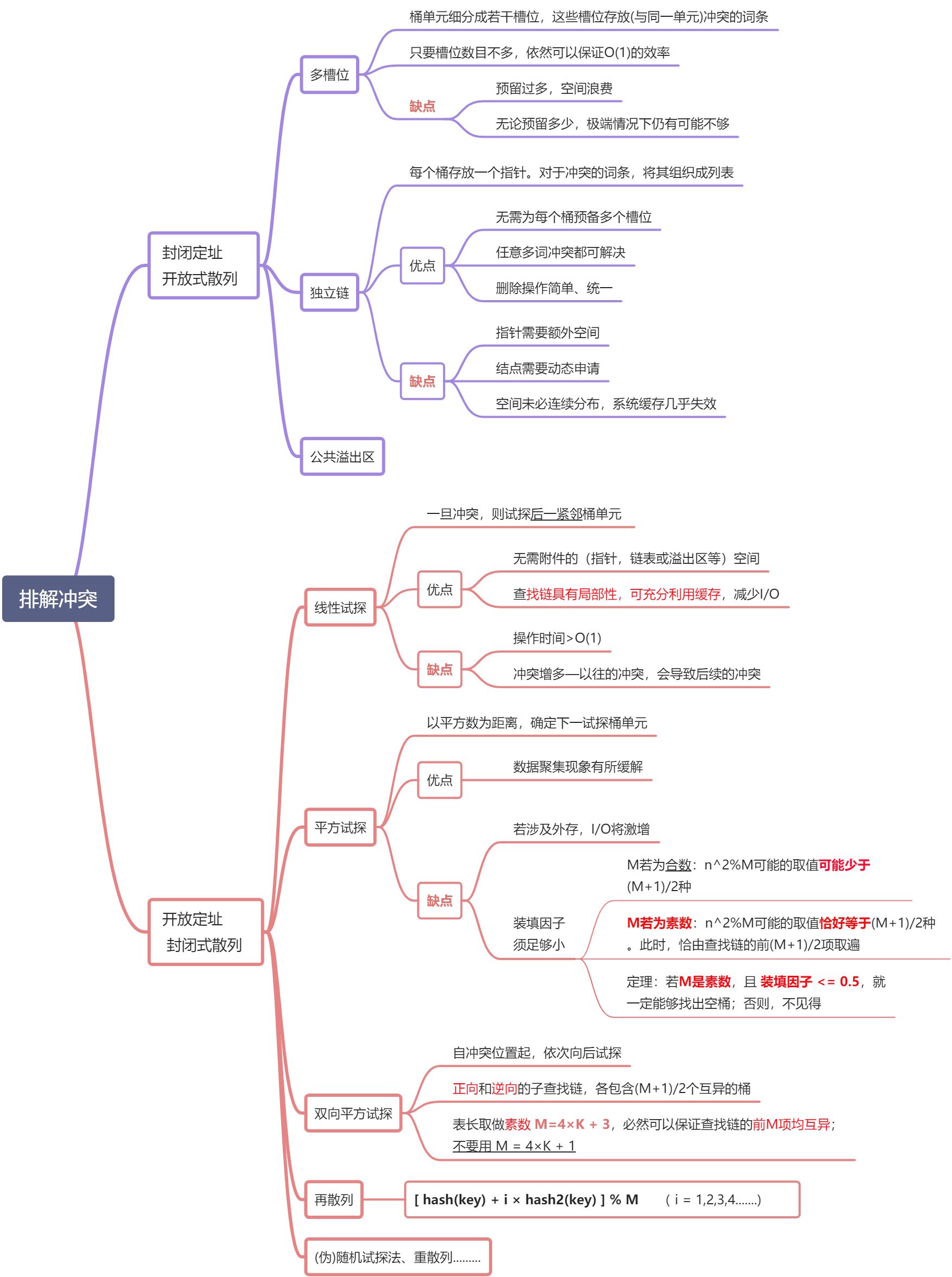
* 为每个桶都事先约定若干备用桶，它们构成一个查找链
* 查找时：沿查找链，逐个转向下一桶单元，直到命中，或者抵达一个空桶标志失败

优点：

* 结构本身保持简洁
* 在散列表内部解决冲突，无需附加空间

缺点：

* 冲突之后，可能引发本可避免的新冲突



**平放试探**

* M若为合数：可能的取值可能少于 (M+1)/2
* M若为素数：可能的取值恰有(M+1)/2，任一关键码对应的查找链中，前 (M+1)/2个桶必然互异；

注意：2011，2017是素数，2019，2021，2023都是合数

若长度M是素数，且装填因子λ <= 0.5，则只要有空桶，就一定能找出，否则不见得。

* 在装填因子尚未增至50%以前，插入操作必然成功；
* 在装填因子超过50%以后，只要适当的调整各桶的位置，下一插入操作必然因无法抵达空桶而失败（即插入操作不成功）；
* 散列表长度M为合数时，即便装填因子低于50%，平方试探仍有可能无法终止（即插入操作不成功）。

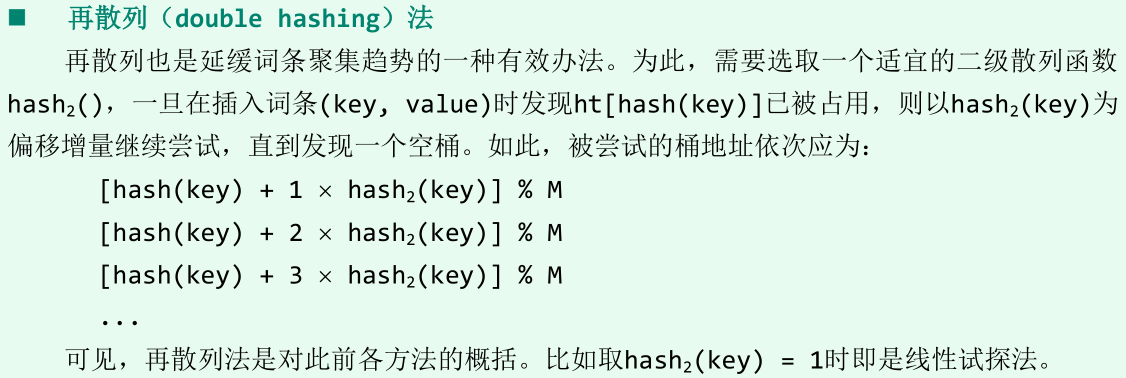
将n个词条逐个插入一个容量为M 、采用线性试探策略、初始为空的散列表，n<M，则无论它们的插入次序如何，最终的平均成功查找长度都必然一样。 ✔

**双向平放试探**

表长取做素数 M=4×K + 3 ⭐

* 必然可以保证查找链的前M项均互异（即取遍整个散列表，访问其全部元素）
* 在装填因子未增至100%之前，插入操作必然成功（而不致因无法抵达空桶而失败）

**再散列法**



**优先级队列**

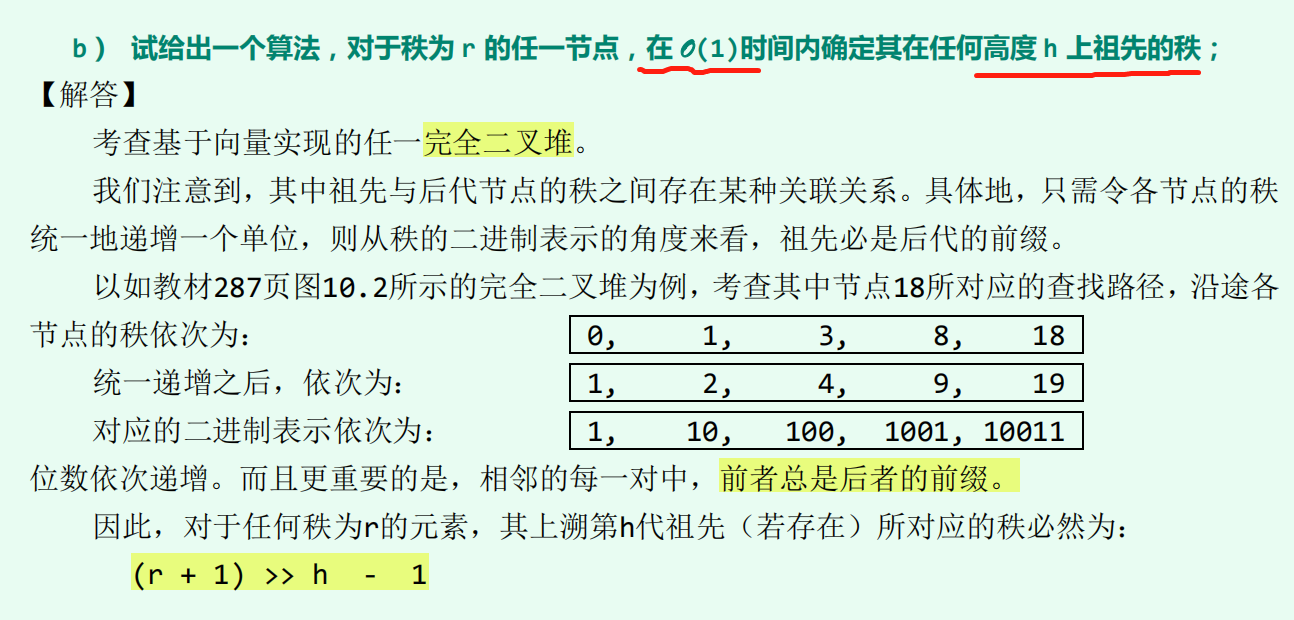
任何CBA式Huffman树构造算法，在最坏情况下都需要Ω(nlogn)的时间。

**完全二叉堆**

完全二叉树最后一个内部节点的下标为：  =

|  |  |  |  |
| --- | --- | --- | --- |
| **完全二叉堆** | | | |
| **插入元素** | **删除元素delMax** | **建堆** | |
| 最坏O(logn) | O(logn) | 自上而下的上滤 | 自**下**而上的**下**滤(Floyd建堆算法) |
| 若关键码均匀独立分布时为O(1) | O(nlogn) | O(n) |

**Flyod建堆算法**中，同层内部结点下滤的次序，对建堆结果没有影响，对建堆所需的时间也没有影响。



**多叉堆**

将二叉堆改成多叉堆(d-heap)，则高度降至

* 上滤成本改进至，插入操作时间成本降低，即插入更快。
* 下滤成本增加，delMax()接口效率降至

注意：三叉堆是反例，三叉堆比二叉堆的删除下滤更快，成本更低

其他的四五六七......叉堆都比二叉堆删除慢，成本更高

对于稠密图，插入操作次数远远多于删除操作次数，故利大于弊，使用多叉堆替换二叉堆，效率增加。

当分叉数 m = e/n + 2 时 ⭐

基于d叉堆实现的PFS优先级搜索 - 如：Prim算法、Dijkstra算法性能能达到最优

* 对于稀疏图，接近于 O(nlogn)
* 对于稠密图，接近于 O(e) —— 效率大大增加

**锦标赛排序**

锦标赛树 - **胜者树** - 优先级队列 —— 是一种完全二叉树

建堆 O(n)；删除，插入操作O(logn)

锦标赛排序空间复杂度 O(n)，时间复杂度O(nlogn)。

借助锦标赛树，从n个元素中找出最小的 k个，就常系数而言，比小顶堆更优。

**胜者树**的根节点是冠军；而**败者树**的根节点记录的是败者（不一定是亚军），败者树需要加一个结点来记录整个比赛的胜利者。

要记住的结论： ⭐⭐

重赛过程中，**败者树**只与其父结点比较，不用交替访问沿途结点及其兄弟，减少了访存时间。

而插入删除操作都要重构，重新选择胜者；故败者树插入，删除操作的时间复杂度在常系数上优于胜者树。

与败者树相比，**胜者树**在重赛过程中需反复将节点与其兄弟进行比较。

败者树效率 > 胜者树 > 堆

**左式堆**

引入左式堆是为了高效的完成堆合并，需参与调整的结点不超过O(logn)个

左式堆首先是堆，即具有堆序性，其次是其具有高效的合并操作。

**空节点路径长度npl**

* 若x为外部节点，则npl(x) = npl(null) = 0
* 若x为内部节点，则 npl = 1+ min(npl(lc(x)),npl(rc(x)))

即节点x的npl值取决于其左、右孩子npl值中的小者。

npl(x)即等于x到外部节点的最近距离，同时也等于以x为根的最大满子树的高度。

**左式堆与左倾性**

左式堆是处处满足“左倾性”的二叉堆，即任意内部节点x都满足

* 任一内部节点的左孩子的npl值都不小于其右孩子的npl 值
* 各节点npl值均取决于其右孩子

x的左孩子到外部节点的最近距离 **>=** 右孩子到外部节点的最近距离

**注意**：“左孩子的npl值不小于其右孩子” 并不意味着 “左孩子的高度必不小于其右孩子”

这两者没有必然联系，“左孩子的npl值不小于其右孩子”   “左孩子的高度必不小于其右孩子”

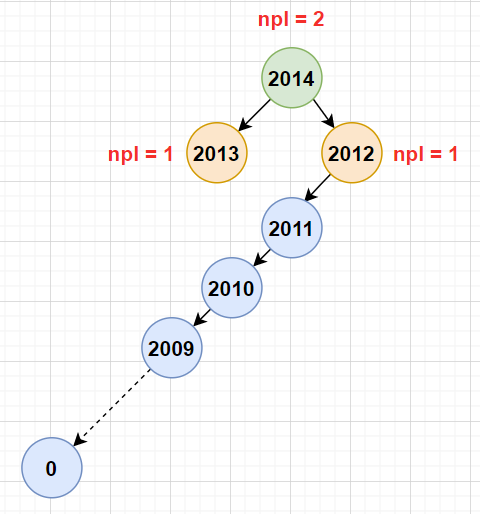
**最右侧通路rPath(x)**

从x出发沿右侧分支一直前行直至空节点，经过的通路称作其最右侧通路rPath(x)。

每个节点的npl值等于其最右侧通路的长度，也即npl(r) = |rPath(x)| = d

* d <= log(n+1) - 1 = O(logn) ⭐
* 总结点数：n >= 2^(d+1) -1
* 内部节点数 >= 2^d -1

有 n 个节点的左式堆，左子堆最小规模为 1。 ⭐



**合并算法**

设待合并规模分别为n和m，则其两条最右侧通路的长度分别不会超过O(logn)和O(logm)

合并复杂度**：**O(logn) + O(logm) = O(logn+logm) = O(log(max(n,m))) ⭐

Crane算法合并左式堆A和B为H，H右侧链节点未必都来自A或B右侧链。 ⭐

也就是说H的右侧链节点可能既包含了A也B的右侧链，等。

**插入与删除**

**插入即是合并**，把新插入的结点封装为以其为根节点的子树，将其与原有树合并，就可以完成插入操作了。

而本次插入操作调用的就是merge()合并接口

故复杂度为**O(logn)**

**删除亦是合并，**删除根节点后，将左右两棵子树合并

而本次删除操作调用的就是merge()合并接口

故复杂度为**O(logn)**

**串**

**蛮力匹配**

随着字符表规模的扩大，以上最坏情况出现的概率将急剧下降

**就期望意义而言，蛮力算法的时间复杂度可以接近甚至达到线性 expected-O(n)** 重要！！

**KMP算法**

**构造next[]表**

* 约定next[0] = -1
* next[ i ] 等于 [ 0，i-1 ] 的模式串中最长真前缀和真后缀匹配的长度

**kmp算法时间复杂度**

KMP复杂度为O(n+m)，即使是在最坏情况下也为O(n+m)重要！

**KMP算法：再改进**

记忆 = 教训 = 预知力

每次在P[0,j)中发现长度为t的真前缀和真后缀相互匹配以后，还需进一步检查 p[ j ] 是否等于p[ t ]。

* 唯有在p[j] ≠ p[t]时，才能将t赋予next[j]
* 否则，需转而代之以next[t]

要会计算串的KMP的nex表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **rank** | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| **P[ ]** | \* | **H** | **H** | **F** | **B** | **H** | **H** | **F** | **H** | **H** | **F** | **B** | **S** | **H** | **F** |
| **未改进Next表** | N/A | -1 | 0 | 1 | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 0 | 1 |
| **改进Next表** | N/A | -1 | -1 | 1 | 0 | -1 | -1 | 1 | 3 | -1 | 1 | 0 | 4 | -1 | 1 |

**在何种情形下KMP优于蛮力算法，为什么？**

KMP算法利用以往成功比对所提供的信息（记忆），不仅可以避免文本串字符指针的回退，而且使模式串尽可能大跨步地右移（经验）。

**BM\_BC算法 - 坏字符策略**

模式串中越靠后的字符对算法性能的优化作用更大

BM算法：模式串的字符从右向左与文本串的字符进行比对

**BM\_GS算法：好后缀**

**gs[]表**：gs[ j ]表示在BM\_GS算法中，当P[ j ] 与目标串某某发生失配时，此时应该将模式串右移 gs[ j ] 个字符。

对于字符集大小为T，长度为 m 的串进行串匹配时好后缀数组中 gs[0]=1 的概率为

**GS表手工计算方法**：设表长为n，其中GS[n-1] = 1

GS[ i ]的计算：根据 i 位置的最长后缀，即模式串 P[ i+1 -> n-1 ]，然后用这段后缀去模式串 P[ 0 -> i-1 ]（ i 前面）找是否有匹配的

* 若有，假设 P[x -> y] = P[ i+1 -> n-1 ]，则还需比较 P[ i ] 和 P[x-1] 是否相等
  + P[ i ] == P[x-1] ，则GS[ i ] = i + 1 - x，即把 x-1 与 i 对齐需要的位移。
  + P[ i ] != P[x-1]，还得继续往前找（若有多个匹配的，选择最靠后的那个）
* 若无，则GS[ i ] = n = 表长，即直接跳过这段

写ladygaga的GS表。（2014期末补充）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **rank** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **P[ ]** | **l** | **a** | **d** | **y** | **g** | **a** | **g** | **a** |
| **GS表** | 8 | 8 | 8 | 8 | 8 | 2 | 6 | 1 |

**串匹配算法对比**

字符表规模越大，单次匹配概率越小

字符表规模越小，单次匹配概率越大

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **最好情况** | **最差情况** | **平均情况** |  |
| **蛮力算法** | | 单次匹配概率越小时，  O(n+m) | O(n×m) | O(n+m) |
| **KMP** | | O(n+m) | O(n+m) | O(n+m) |
| **BM** | **BC** | 单次匹配概率越小时，效率越高： O(n/m) | 单次匹配概率越大时，越接近蛮力算法 O(n×m) |  |
| **GS** | O(n/m) | O(n×m) |  |
| **BC+GS** | O(n/m) | O(n+m) |  |

**排序**

**插入排序**

序列的插入排序平均有expected-O(logn)个元素无需移动。

插入排序中，若所有逆序对的间距均不超过k，则运行时间为O(kn)

* 特别的，当k为常数时，插入排序可在线性时间内完成

若共有I个逆序对，则关键码比较的次数不超过O(I)，运行时间为O(n+I)

**插入排序**：就地算法、在线算法、具有输入敏感性、有最好/最坏情况

二分查找？改用向量、用“比较”换“交换”？总体不变

插入排序、选择排序、冒泡排序排序过程中，**逆序对的个数**一直在减少，也就是单调不增；

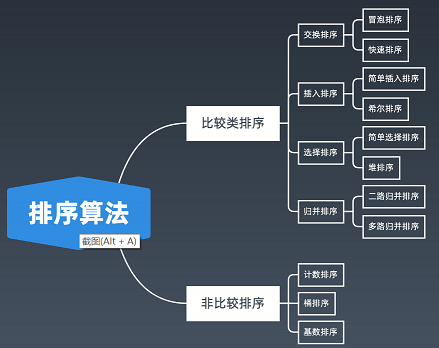
而对于**循环节**个数来说：

* 插入排序过程中，循环节个数可能不变，可能减少，可能增加
* 选择排序的循环节个数一直在增加，也就是单调不减

**快速排序**

基于比较式的算法可以在**O(n)**时间内在任意n个无序整数中找出居于前10%的数（其实换成任意大的数都可以，不只是10%）。

—根据【k-选取算法】可以在O(n)时间内完成k-选取。



**排序算法大比拼**

要点：比较次数、交换次数

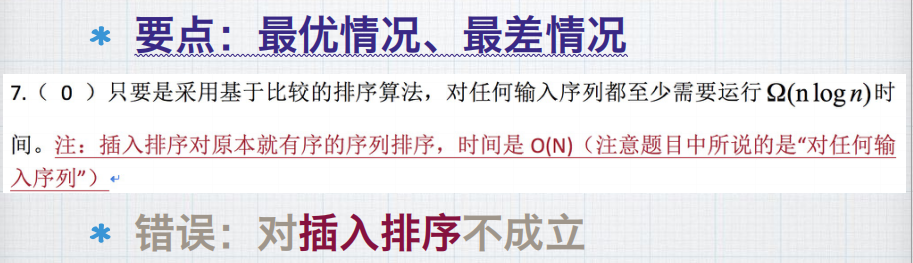
**选择排序**相对于**起泡排序**的优点：比较次数不变、交换次数减少

选择排序、起泡排序：比较次数均为O(n^2)，交换次数不同

要点：最优情况、最差情况

**插入排序**相对于**选择排序**的优点：在最优情况下，比选择排序要好（选择时间是固定的、插入是根据输入规模的）

冒泡排序在最优情况下，也比选择排序要快

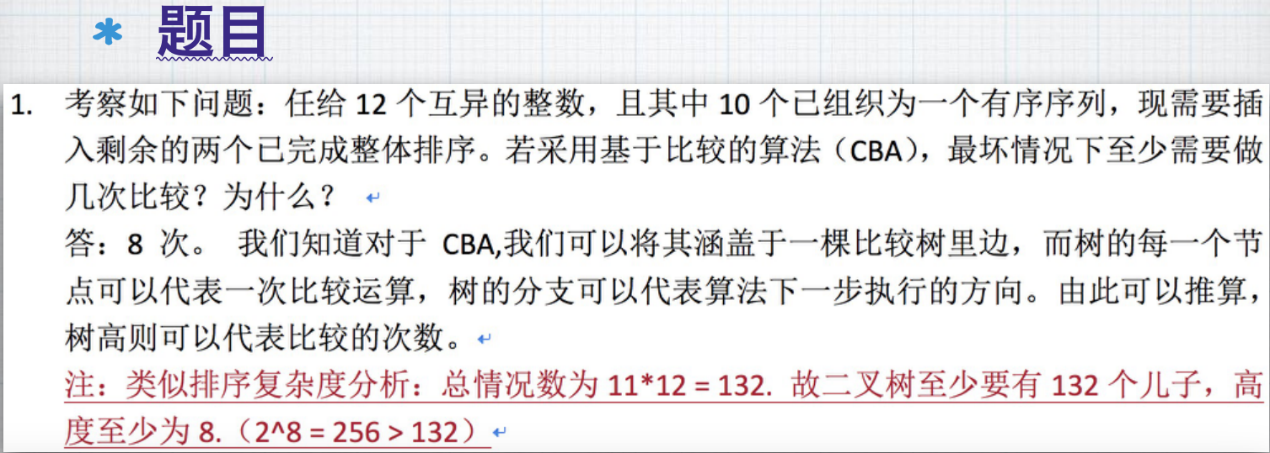


要点：时间复杂度分析

**选择排序**：排List，与循环节长度有关

每次会减少循环节的总长度1

实际上是没什么影响的（lnn/n->0）



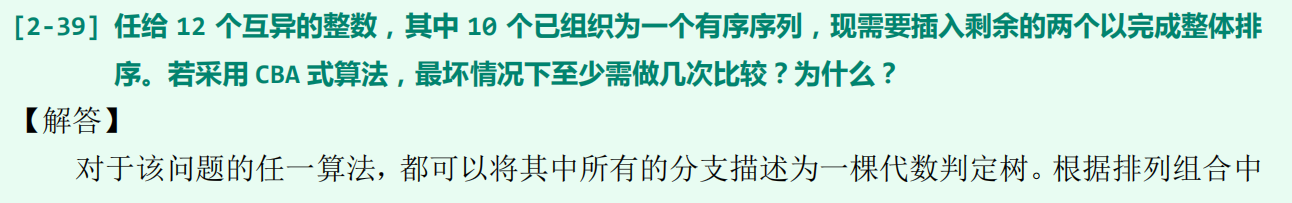
第一个数插入10个有序中，有11个可能的位置

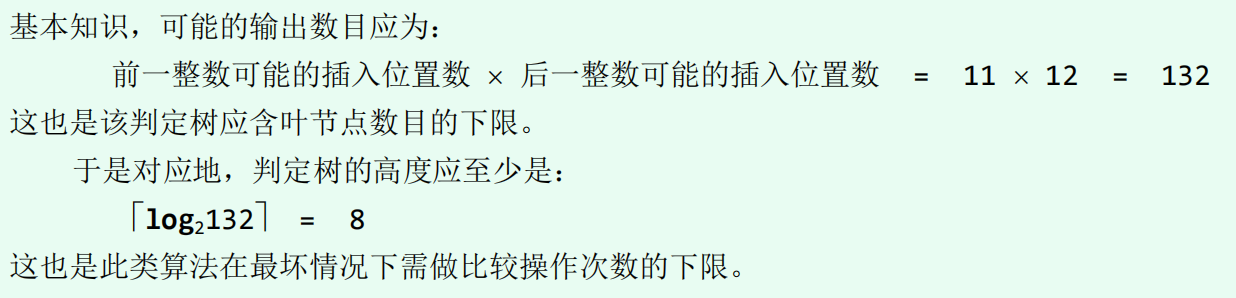
第一个数插入11个有序中，有12个可能的位置

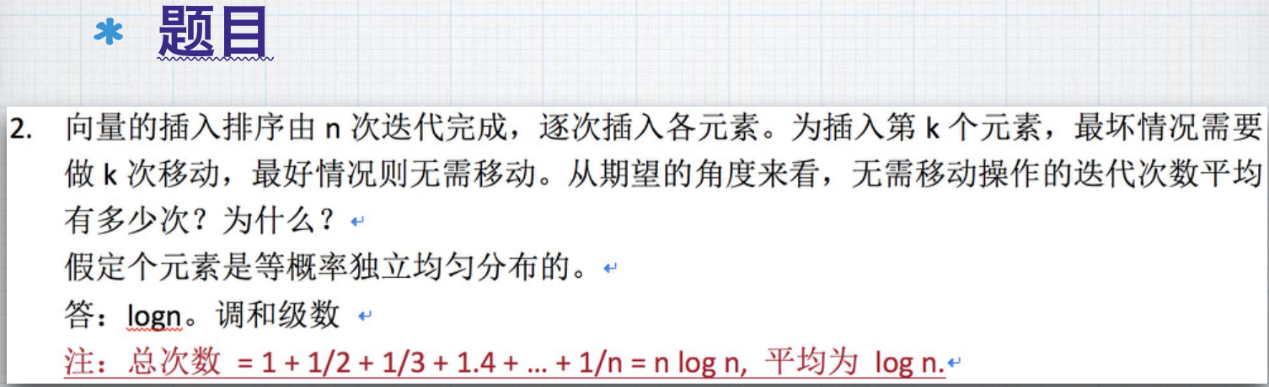
故总情况数为11×12 = 132

判定树的树高至少为：

【习题解析 2-39】



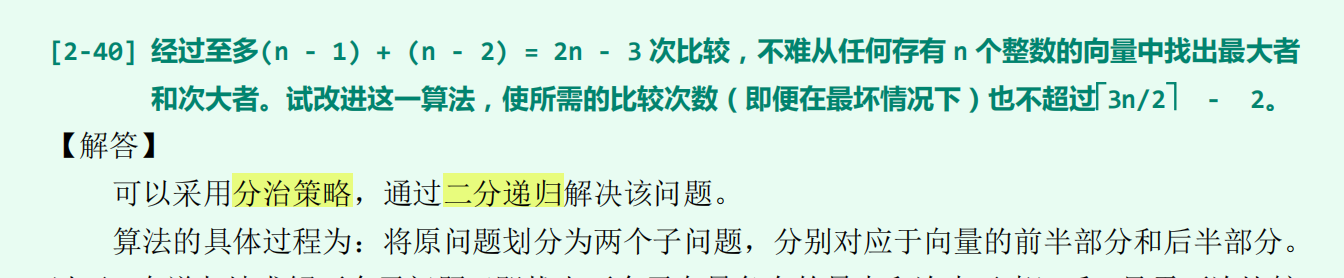




不存在CBA式算法，能够经过少于2n-3次比较操作，即从n个整数中找出最大和次大者。（2014期中）

错误 ❌

存在，可不超过 (3n + 1) /2 - 2次



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **最好性能** | **平均性能** | **最坏性能** | **稳定性** | **空间复杂度** | **其他特点** |
| **冒泡排序** | O(n) | O(n2) | O(n2) | 稳定 | O(1)就地 |  |
| **选择排序** | O(n^2) | O(n2) | O(n2) | 可以实现稳定 | O(1)就地 | 每次迭代移动次数为O(1) |
| **插入排序** | O(n) | O(n2) | O(n2) | 稳定 | O(1)就地 | 输入敏感，与逆序数相关 |
| **归并排序** | O(nlogn) | O(nlogn) | O(nlogn) | 稳定 | O(n)归并时需要额外数组 | 二分思想 |
| **堆排序** | O(nlogn) | O(nlogn) | O(nlogn) | 不稳定 | O(1)就地 |  |
| **快速排序** | O(nlogn) | O(nlogn) | O(n2) | 不稳定 | 最好的情况 O(logn) 最坏是 O(n) | 分而治之，pivot划分左右两个序列，左 <= 右 |
| **希尔排序** | O(n) | 取决于步长 | O(n2) | 不稳定 | O(1)就地 |  |