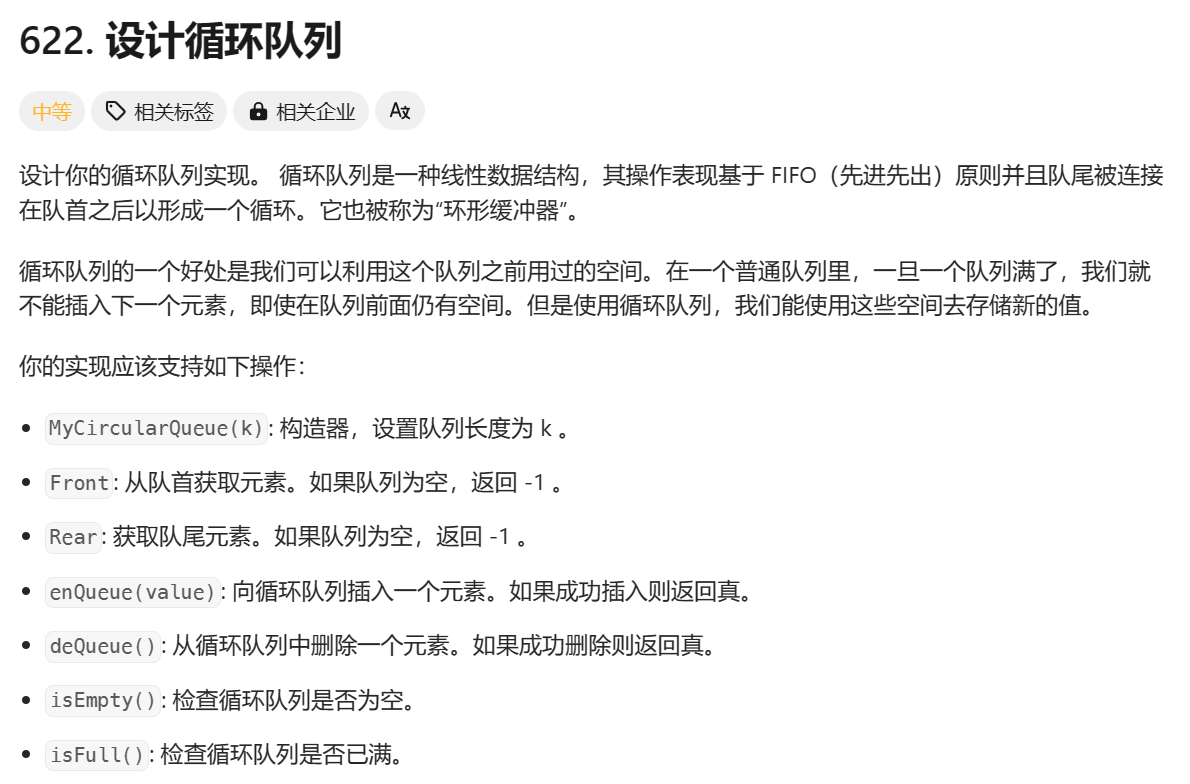
**栈和队列+树**

**在栈实现队列中：如果创建的两个栈是Stack\*类型的，就必须传二级指针进行初始化，如果相传一级指针，就必须先malloc一个空间，再进行初始化。**

****

**正常的队列是先进先出，循环队列也是先进先出，但是首位相连，队列满了之后，就不能插入数据了，只有删除了数据之后才能继续插入数据。**

**底层：单向循环，最好不要哨兵位，因为每次取完数据之后，要判断是否是哨兵位**

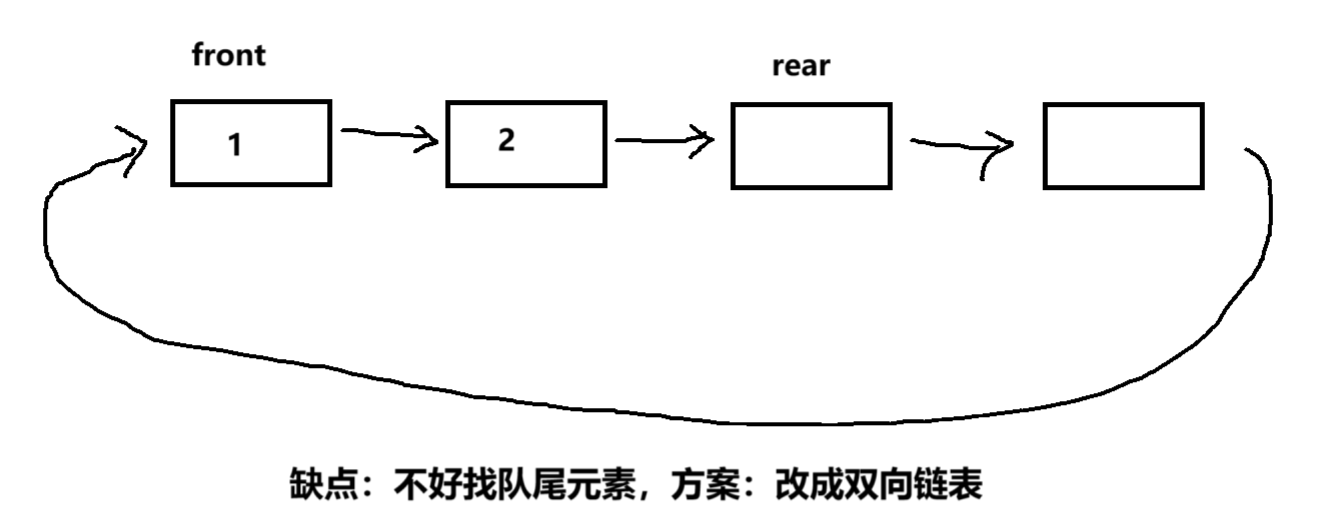
**类似于约瑟夫环**

**在循环队列中，在循环队列空的情况下一个头结点front和一个尾节点rear同时指向头结点**

**相当于，front==rear循环队列就为空，如果空front==rear那么满队列的时候也是front==rear，**

**所以加一个size，size==0是空，size==max，满，size==k就是未满。**

**Pop之中front往后走，插入数据就在rear位置处插入数据。**

****

**缺点：不好找队尾元素。**

**为什么链表的rear不指向尾。**

**空的时候front和rear指向同一个数据，有一个数据的时候front和rear也指向同一个数据。**

**用链表：**

**如果指向最后一个元素的下一个位置，我们就要用双向链表，才好控制rear取队列尾元素。**

**如果rear指向的是最后一个元素，取队尾元素方便了，但是pop的最后需要做特殊处理还有初始化的时候也需要做特殊处理。**

**更简单的方式就是用数组：**

**在这种题目中rear一般指向尾的下一个元素，在用数组的时候一般会多开一个空间，k==4的时候，就开了5个空间，防止假溢出问题，也就是空和满的问题，**

1. **空和满的问题可以用size解决**
2. **多开一个空间**

**Front==rear就是空。**

**Rear指向的是最后一个数据的下一个数据，满了的话就表示rear+1==front**

**如何解决回绕的问题？(rear+1)%(k+1)==front**

**链表也可以多开一个空间，来解决循环回绕的问题**

**树的概念及结构**

**树的概念：**

1. **节点的度：一个节点含有的子树的个数称为该节点的度；（有几个孩子度就是多少） 如上图：A的为6**
2. **叶节点或终端节点：度为0的节点称为叶节点；（没有孩子的节点） 如上图：B、C、H、I...等节点为叶节点**
3. **非终端节点或分支节点：度不为0的节点； （中间的节点）如上图：D、E、F、G...等节点为分支节点**
4. **双亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点； 如上图：A是B的父节点**
5. **孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点； 如上图：B是A的孩子节点**
6. **兄弟节点：具有相同父节点的节点互称为兄弟节点； 如上图：B、C是兄弟节点**
7. **树的度：一棵树中，最大的节点的度称为树的度；（最大的度是多少树的度就是多少） 如上图：树的度为6 节点的层次：从根开始定义起，**
8. **根为第1层，根的子节点为第2层，以此类推； （如果从0开始空树如何区分，空树就是-1开始）**
9. **树的高度或深度：树中节点的最大层次； 如上图：树的高度为4**
10. **堂兄弟节点：双亲在同一层的节点互为堂兄弟；如上图：H、I互为兄弟节点**
11. **节点的祖先：从根到该节点所经分支上的所有节点；如上图：A是所有节点的祖先**
12. **子孙：以某节点为根的子树中任一节点都称为该节点的子孙。如上图：所有节点都是A的子孙**
13. **森林：由m（m>0）棵互不相交的树的集合称为森林；（并查集就是一个森林）**

**任何一棵树都被拆成根和子树，子树是递归定义的，子树也按一样的分时拆解，子树又会被拆成根和子树，……**

**直到走到叶子就终结了，最后一个叶子节点会被拆成根和零颗子树**

**只有没有关联的几颗树才是森林**

**树还有一个概念：**

1. **子树是不相交的**
2. **除了根节点外，每个节点有且仅有一个父节点**
3. **一颗N个节点的树有N-1条边**

**二叉树的表示法：**

**双亲表示法**

**左孩子右兄弟表示法**

**如何判断一个森林，可以判断两个节点的根节点是否相同。**

**树在实际当中的应用**

**表示文件系统的目录树结构**

**Windows的文件系统可以认为是一个森林，C盘和d盘还有其他盘构建了这个森林，在windows的文件存储系统的底层就是用的做孩子右兄弟表示法，在一个文件夹下创建一个文件只需要链接一个兄弟就可以了。**

**思维导图也可以理解成树形结构**

**二叉树概念及结构**

1. **概念**

**一颗二叉树是结点的一个有限集合，该集合：**

1. **或者为空**
2. **由一个节点价加上两颗别称为左子树和右子树的二叉树组成**

**二叉树不存在度大于2的节点**

**二叉树比树形结构简单，二叉树用来存储数据由什么意义，二叉树单纯用来存储数据，没有什么价值，不如顺序表\链表**

**真正有意义的二叉树是搜索二叉树**

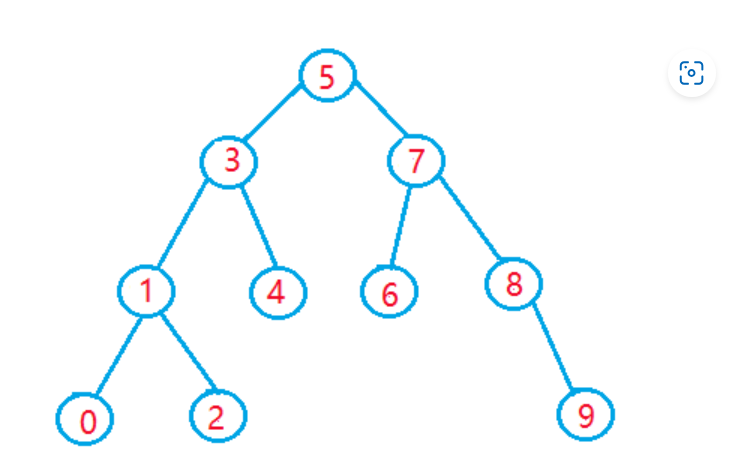
**搜索二叉树的特点：**

1. **存储数据**
2. **搜索数据**

**每棵节点都必须满足左子树小于根，右子树大于根**

**这样查找非常的快。**

**在普通二叉树中**

****

**搜索二叉树最多查找高度次**

**搜索二叉树也有很多不足，**

**搜索二叉树扩展形成AVL树和红黑树**

**M阶的b树（多叉），m是多少就是多少叉树，多用于数据库的引擎**

**在数据库中也要学到M阶的b树**

**哈希表**

**二叉线索化**

**树中还有一些好用的，比如哈夫曼树，哈夫曼主要用于文件压缩。**

**学习二叉树主要有两个**

**特殊的二叉树**

1. **满二叉树：一个二叉树在每一个层的节点数都达到了最大值，则这个二叉树就是满二叉树，也就是说，如果一个二叉树的层数为k，且节点总数是2k-1，则它就是满二叉树**
2. **完全二叉树**

**完全二叉树就是前n-1层姐姐都是满的，但是最后一层不一定是满的要求从左到右的节点都是连续的。**

**满二叉树也还是完全二叉树（就像正方形是一个特殊的长方形）**

**满二叉树一定是完全二叉树，但是完全二叉树不一定是满二叉树。**

**二叉树的逻辑结构：二叉树**

**二叉树的物理结构：数组**

**用数组表示的话，数组的下标之间有一个规律**

**Leftchild = parent\*2+1**

**Rightchild = parent\*2+2**

**Parent = (child-1)/2**

**结论：数组存储只适合完全二叉树和满二叉树。（减少空间浪费啊）**

**堆的概念及结构结构**

**堆的要求：**

**堆必须满足完全二叉树**

**大堆：任何一个父亲都大于等于孩子**

**小堆：任何一个父亲都小于等于孩子**

**逻辑结构：二叉树**

**物理结构：数组**

**堆的作用：堆排序，topk**

**大堆可以找最大值，小堆可以找最小值。**

**下堆中插入数据可能会影响祖先，所以当插入之后，就会用到向上调整算法。**

**如何将数组变成堆**

**HPPop对于这个函数**

1. **如果采用删除首元素然后挪动覆盖时间复杂度是ON**
2. **堆结构被破坏，父子变兄弟兄弟变父子**
3. **就会导致重新建堆。**

**回顾**

**堆是一个完全二叉树**

**在底层是一个数组，在逻辑上是一个二叉树**

**堆只能保证堆顶是一个最大的值或者是一个最小的值**

**在pop中，如果直接pop堆顶的值，那么会使整个堆全部变乱**

**堆的插入和删除的时间复杂度是多少，要分析时间复杂度，就要分析数的高度的关系。**

**假设数中的节点个数是N，树的高度是h，那么，高度和树的节点就可以写出一个关系，这个关系是一个等比数列和，**

**在满二叉树中高度和高度和节点的个数如下图所示：**

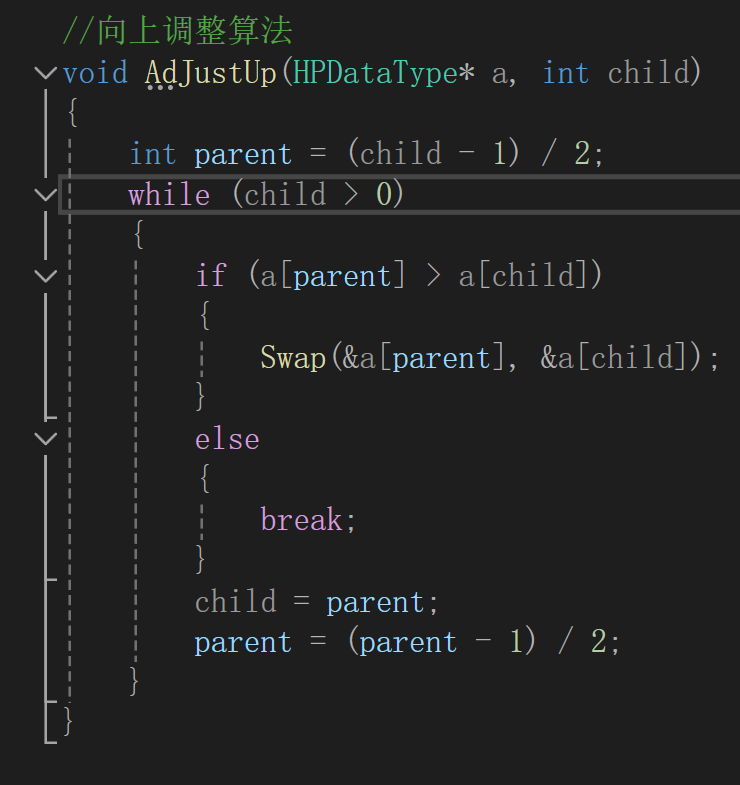
**N=2^0+2^1+2^2+……+2^(h-1) 等比数列求和**

**N=2^h-1**

**完全二叉树的节点个数和高度的关系：**

**高度为h，节点个数是N**

**N=2^(h-1)**

****

**向上调整算法最多执行logN次就是高度次，删除**

**Pop在向下调整中，最多也走高度次，也就是最多一个logN次**

**在效率上ON和OlogN效率是非常大的**

**假设有1000个值，logN的效率只需要执行10次，但是N效率只能执行1000次**

**冒泡排序的时间复杂度是N^2**

**快速排序的时间复杂度是N\*logN**

**Topk问题**

**假设有100亿个数据，找出前10个最大的数据**

**方法：**

1. **用前十个数据建小堆**
2. **后续跟这个堆顶数据比较，如果比这个堆顶数据大，就替代堆顶，，然后向下调整**

**堆里面最后剩的就是最大的前K个**

**完全二叉树用数组存储来表示堆，就能解决堆排序和topk问题**

**二叉树链式结构的实现**

**二叉树部分：**

**遍历**

**分为：前序，中序，后续**

**在链式二叉树中，分为三部分，一个是根一个左子树和右子树**

**左子树继续分：分为左子树和右子树和一个节点，右子树也要分**

**直到分为空树就停止了**

**前序：根 左子树 右子树**

**中序：左子树 右子树 根**

**后续：左子树 右子树 根**

**层序遍历：**

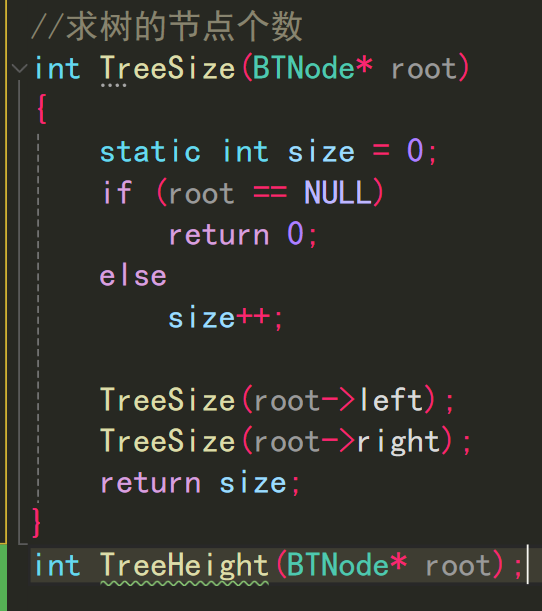
**普通二叉树的增删查改没有价值，**

**所以二叉树学他的结构操作，前中后序**

**Static是存在静态区中的，当函数栈帧销毁的时候static修饰的变量是不会销毁的**

**静态成员变量只会被初始化一次**

**同一个变量不会初始化两次，**

****

**对于上面的函数中的静态变量size，每一次调用这个函数都会使size叠加，因为静态变量是存储在静态区中的，每次调用这个函数时，只会初始化一次，所以这种方法是不行的，如果用这个方法的话，必须每次调用的时候初始化一次，就可以解决上面算法产生的size每次叠加的问题**

**全局变量的线程是不安全的，局部变量的线程是安全的。**

**可以把一个局部变量的size的指针，对指针进行操作，然后返回这个指针**