1. 什么是时间复杂度？O（n）的O代表了什么？

答：时间复杂度是指执行算法所需要的计算工作量，可以用于描述一个程序的规模。O（n）中的O表示的是最坏情况下的时间复杂度。

2. 时间复杂度的量级排序是什么？

答：O（logn）< O（n）< O（nlogn）< O（n^2）< O（2^n）< O（n！）

3. 顺序存储与链式存储是什么？他们各自的优缺点是什么？

答： 顺序存储结构 是一种地址空间连续、支持随机访问的线性表结构，优点是支持随机访问，缺点是插入或删除某个元素需要移动大量的元素；链式存储是不要求连续地址空间，只需要在逻辑上通过指针进行相连的线性表结构，优点是插入与删除元素简单，缺点是不支持随机访问。

4. 为什么循环队列要牺牲一个空间？

答：在循环队列中通常有队头指针和队尾指针，其中队头指针指向队列中的第一个元素，队尾指针指向最后一个元素的下一个位置。如此一来在队列执行判空和判满的操作时，判定条件都是头指针等于尾指针。此时通过牺牲一个空间，可以将两种操作分开：判空时条件是头指针等于尾指针，判满时条件是（rear+1）% size == front。

用于区分判满和判空的类似操作还可以是维护一个size变量来存储队列元素的个数，或者设定一个flag变量，用于表示最后一次操作是插入还是删除。当front==rear时，若最后一次操作是插入，则代表队满；如果是删除，则代表队空。

5. 有一个循环队列Q，编号为0到n-1，头尾指针分别是f与r，求当前队列中的元素个数？

答： （f-r+n）% n。

6. 介绍一下KMP算法？

答：KMP算法是一种字符串匹配算法，通常用于查找文本串S中模式串P出现的位置。相比于朴素的字符串匹配算法中，匹配失败需要主串和模式串同时回溯的做法，KMP只需要回溯模式串而不需要回溯主串，提高查找效率。KMP算法的实现需要借助于next数组，该数组记录了当匹配失败情况发生时，模式串需要回溯到什么位置开始重新匹配，其本质是模式串子串的最长相同前后缀。

Next数组的计算流程如下：初始化next数组为模式串的大小，初始化i与j指针为0与-1，当j==-1或者next[i]==next[j]时，i与j同时加1，此时j代表的是已经匹配了多长的前后缀，即next[i] = j；如果不满足该条件，则j回溯，j = next[j]，并循环知道将next数组填充完毕。

匹配流程如下：初始化i与j指针为-1，i指针指向主串，j指针指向模式串，进行与朴素匹配类似的操作。唯一的不同是，当主串的第i个字符不等于模式串的第j个字符时，i不用回溯，而只需要将j=next[j]便可继续进行匹配，直到j指向模式串最后一个字符之后则代表匹配成功，否则匹配失败。

7. 满二叉树与完全二叉树的区别？

答：满二叉树即对于一颗高为h的二叉树，结点个数为2^h-1的二叉树，表现为除了最后一层叶子结点之外，根节点以及分支结点都有两个孩子，即每一层都是满的。

而完全二叉树则是在满二叉树的基础上，在最后一层从右往左依次删除一定数量的叶子结点所形成的二叉树。完全二叉树的特点是叶子结点只出现在倒数第一和第二层，且如果有分支结点仅有一个孩子，那只能是左孩子。

8. 二叉排序树（查找树）与二叉平衡树的区别?

答：二叉排序树即BST，其每一个结点都有用于排序的关键码，结点左子树的所有结点的关键码都小于当前结点的关键码，而所有右子树的关键码都大于该结点的关键码。对二叉排序树进行中序遍历，即可以得到从小到大有序的关键码序列。它利用了二分的思想，可以快速查找到关键码，查找效率为O（nlog2n）。
二叉平衡树是一种特殊的二叉排序树，它满足对于树上的任意一个结点，其左子树的深度与右子树的深度之差的绝对值不超过1。平衡因子可以用于描述二叉平衡树，平衡因子是某个结点的左子树深度与右子树深度之差，对于一棵二叉平衡树，其任意结点的平衡因子只能是-1，0或1。

9. 二叉排序树的查找、插入与删除操作过程是什么?

答：查找从根节点开始，如果要查找的关键码大于当前关键码，则下一个查找的结点为根节点的右子树，反之则是左子树。再以新节点为根，重复以上的查找步骤，直到查到得到匹配的关键码为止。
插入操作则是基于查找操作进行，查找合适的位置进行插入。该合适的位置指的是按照查找步骤进行到的叶子节点处，若欲插入的关键码大于该叶子结点，则插入为右孩子，反之为左孩子。插入的结点必须是叶子结点。若开始树空，则直接成为根节点；若欲插入的关键码已存在，则插入失败。二叉树的构造过程也是不断插入的过程。
删除操作同样是基于查找操作，首先查找到欲删除的结点。此时，删除结点通常包括三种情况：① 若删除的结点是叶子结点，则可以直接将结点删去；② 若删除的结点只有左孩子或者右孩子，则用它的孩子代替它；③ 若删除的结点有左右孩子，则可以寻找其中序遍历的直接前驱或者直接后继代替它，再删去该直接前驱或直接后继。具体做法是：寻找删去结点的左子树的右右右孩子直到右孩子为空，即为直接前驱，或者寻找右子树的左左左孩子直到左孩子为空，即为直接后继。替代之后，再按照①②情况对直接前驱或后继进行删除。

10. 二叉平衡树的插入与删除过程（对平衡二叉树的调整）?

答：平衡二叉树的插入与删除操作与二叉搜索树相同，若出现不平衡现象，则需要对其进行调整。出现不平衡现象则意味着出现某结点的平衡因子大于1或者小于-1的情况，此时需要查找最小不平衡点，即距离插入结点最近的平衡因子大于1或者小于-1的结点，对其进行调整，调整思路分为以下四种情况：
① 插入结点位于最小不平衡点的左孩子的左子树中（LL型）：利用扁担原理，以最小不平衡点的左孩子为支撑进行右旋。代码操作为：最小不平衡点的左指针指向其左孩子的右孩子，其左孩子的右指针指向最小不平衡点，再将其左孩子的父节点更换为最小不平衡点的父节点。
② 插入结点位于最小不平衡点的右孩子的右子树中（RR型）：与LL型类似进行左旋。
③ 插入结点位于最小不平衡点的左孩子的右子树中（LR型）：先进行左旋，再进行右旋。按照RR型的调整方式对最小不平衡点的左孩子进行调整使其变成LL型，再对最小不平衡点进行LL型的调整方式。
④ 插入结点位于最小不平衡点的右孩子的左子树中（RL型）：与LR型类似先进行右旋再进行左旋。

11. 什么是哈夫曼树?

答：哈夫曼树又称为最优二叉树，其特点是，给定一组带权的叶子结点，若构造所得到的二叉树拥有最小的带权路径长度WPL，则称该二叉树为一棵哈夫曼树。
构造哈夫曼树的具体过程是：将带权叶子结点并入一个集合，首先在集合中挑选出两个权值最小的叶子结点进行合并（值相加）得到新的结点加入集合，再将两

个被选中的结点剔除出集合。在树的构造上，将这两个结点作为叶子结点衔接到合并而成的新结点上。重复以上过程直到集合中只有一个元素，哈夫曼树则完成构造。
哈夫曼树的应用是哈夫曼编码，其特点是消除了编码前缀相同的二义性。特别的，在哈夫曼编码中，只有哈夫曼树的叶子结点可以进行编码。

12. 树有什么存储结构?

答：对于普通的树来说，存储结构通常分顺序存储与链式存储，并且再细分为以下三种存储结构：① 双亲表示法：这是一种顺序存储结构，以顺序表为载体，每个结点有数据域和双亲位。其中，双亲位存储的数据是该结点的父节点所存储的下标；② 孩子表示法：这是一种顺序存储结构与 链式存储结构 的结合，每个结点有数据域和第一个孩子指针域，以链表的形式存储该结点所有孩子的下标数据；③ 孩子兄弟表示法：这是一种链式存储结构，每个结点有两个指针域，一个是指向自己的第一个孩子结点，另一个则指向自己的第一个右兄弟。
而对于二叉树来说，也可以分为顺序存储与链式存储结构。① 顺序存储结构中，按照二叉树层序遍历的顺序将结点存储于顺序表中，特别注意空节点也需要占有位置。若某结点下标为i，则其左孩子下标为2i，右孩子下标为2i+1，父节点下下标为i/2向下取整。该存储结构适合存储完全二叉树；② 链式存储结构中，每个结点通常一个数据域与两个指针域，分别指向自己的左孩子和右孩子。而为了充分利用左右孩子指针，可以将左孩子指向自己的前序、中序或后序遍历的直接前序，右孩子指向直接后继，从而形成二叉线索树，方便查找。

13. B树和B+树的区别?

答：B树是一种多路平衡查找树。首先，多路是因为对于一棵m叉B树，每个结点最多有m棵子树，并且结点有m-1个关键字。关键字的左子树的关键字都小于当前，右子树的关键字都大于当前。而为了保持较高的查找效率，B树的高度不能太高，因此限定除了根节点以外（根节点至少要有一个关键码，即要有两个分支），所有结点的关键码个数不能少于m除以2向上取整再减1个。而平衡是因为B树是一棵高平衡树，所有结点的平衡因子都为0，叶子结点均在最后一层，倒数第二层称为终端结点。
B+树与B树有所不同，主要有以下几点：① B+树的结点关键码个数与结点的子树个数相同，而B树中关键码个数比子树个数少1； ② B+树的关键码存储的是其对应子结点中关键码的最大值，利用了分块查找的思想，而B树则是利用了二分查找的思想； ③ B+树的倒数第二层称为索引结点，通过指针彼此连接，而B树的倒数第二层是终端结点，没有相互连接； ④ B+树的结点不存储记录，记录只存储在最后一层的叶子结点中，因此查找结束通常发生在叶子结点，而B树的每个结点都存储记录，因此查找结束可以发生在任何结点。 ⑤ 由于B+树的特性，其可以实现范围查找（通过索引结点的索引遍历），并且查找速度稳定，相反B树不能实现范围查找（每次都需要从头开始查找），速度不稳定。

14. B树和B+树在数据库中的应用?

答：在数据库查询中，通常以树形结构来存储数据。B树或者B+树中的每一个关键字，代表一个相应的磁盘块。系统首先从根节点读取磁盘块到内存，再按照内存中读取的分块信息继续读取子树中关键字所代表的磁盘块，知道寻找得到关键字，再读出记录。因此，树的高度意味着需要进行多少次I/O操作，高度越少，需要进行的I/O次数相应越少。
如果使用AVL树存储数据，当数据量大的时候，AVL树的高度会变得非常大。而B树与B+树一个结点可以存储多个值，读取磁盘时是将整个结点读取到内存再进行处理，因此可以提升I/O效率。而B树的缺点是不利于范围查找，每次查找需要多次从根节点开始，B+树由于有索引结点的存在，并且按照索引值从小到大进行排序，只在叶子结点存储记录，因此可以通过链表的遍历实现范围查找。

15. 你了解红黑树吗？能不能简单聊聊红黑树?

16. 介绍一下最小生成树算法?

答：首先讲一下我对于最小生成树的理解。生成树是一种特殊的图，它通过普通的图结构删减边得到，没有环路，但每个结点相互连通。而最小生成树就是找到路径权重之和最小的那个生成树。最小生成树算法有普里姆和科鲁兹卡尔算法。

普里姆算法：首先在图中取一个结点加入空集中形成一个集合，并且挑选与该结点相连接权值最小的结点加入集合，再继续寻找一个与集合中结点相连接权值最短的结点加入集合。循环以上步骤直到图中所有结点都加入集合，则完成最小生成树的构建。时间复杂度为O(n^2)。

科鲁兹卡克算法：首先将图中所有的变从小打到进行排序，然后从小边开始寻找，找到一条合适的边加入图中。合适的边指的是在该边加入图中之前，该边两端的结点集合并不连通。重复以上过程直到图连通，则完成最小生成树的构建。判断是否属于同一个连通集通常使用并查集算法，时间复杂度为O(log2n)，由于算法需要进行n次搜索，因此总时间复杂度为O(nlog2n)。

17. 介绍一下最短路径算法?

答：最短路径算法即求图中两点之间的最短路径问题，通常有BFS算法、迪杰斯特拉算法和弗洛伊德算法，其中BFS只能处理无权图，迪杰斯特拉算法可以进一步解决带权图问题，弗洛伊德可以进一步解决带负权边图问题，但是他们都不能解决带负权回路图问题。

BFS即广度优先搜索，通过队列来实现，首先将单原点加入队列。每次循环将队列中队头元素弹出，并且将与该元素所代表的结点相邻的结点加入队列，直到队列为空。每次循环代表距离加一，BFS可以找出单源点到其他结点的路径长度，取最小即为最短路。

迪杰斯特拉应用了贪心的思想，通常解决单源点问题。首先将源点直接相连的结点作为候选结点，不直接相连的结点将距离标注为无穷。此时在所有候选节点中选出距离源点最短的结点，作为下一次迭代的中间结点，并对于中间结点直接相连的结点的路径长度进行更新，即“若源点到中间结点的距离加中间结点到该结点的距离<源点到该节点的距离”则进行更新。重复以上过程，直到所有结点完成更新。其时间复杂度为O(n^2)。

弗洛伊德则是应用了动态规划的思想，通常解决多源点问题。首先对图进行初始化，遍历图中所有结点，将与结点直接相邻的结点之间的边标注上原有的权值，而到不相邻结点的权值标注为无穷。算法将遍历图中所有结点，每次将遍历到的结点作为中间结点，并且对所有结点之间的权值进行更新，即“若结点1到结点2的距离>结点1到中间结点再到结点2的距离”则进行更新。由于需要进行三层遍历，其实间复杂度为O(n^3)。

18. 最小生成树算法以及最短路径算法的优化?

答：最小生成树算法中的普里姆算法与科鲁兹卡尔算法都需要排序，前者需要对点到集合的路径进行排序，而后者需要对边进行排序，因此二者都可以利用堆进行优化，通过访问堆顶元素以及堆调整来实现快速访问最短边。与此类似，最短路径中的迪杰斯特拉也可以利用堆进行优化，时间复杂度可以下降为O(nlog2n)。（在更新最短路径的时候会破坏原有的堆结构，此时应该对被破坏的结点进行上浮到合适的位置）

19. 介绍一下拓扑排序?

答：拓扑排序即一种生成图的拓扑序列的方式，用于检查图中是否存在环，或者应用于找出事件发生的先后顺序。拓扑排序的具体步骤是：利用栈或者队列，将图中入度为0的结点入栈，同时在图中将该结点进行删除，再将删除该结点之后入度变为0的结点进行入栈，重复以上顺序即可获得拓扑序列。若算法结束时，序列长度不等于图中结点的数量，则说明图中存在环，即无法描述先后顺序。

20. 介绍一下AOE网?

答：AOE网是在带权有向图中，以顶点代表时间，以有向边代表活动，以边上的权值代表完成该活动的开销或所需时间的网络。从源点到汇点的有向路径可能有多条，在所有路径中，具有最大路径长度的路径称为关键路径，关键路径上的活动称为关键活动。完成整个工程的最短时间就是关键路径的长度，若关键活动不能按时完成，则整个工程的完成时间就会延长。

21. 哈希表的概念？构造方法？解决冲突的方式？装填因子?

答：哈希表的概念：散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度；
哈希表的构造方法：哈希表的构造方法一般有直接定址法、数字分析法和平方取中法（取关键字平均值的中间几位作为散列地址）；
解决冲突的方法：开放定址法（线性探测、平方探测）、拉链法、再哈希法；
装填因子：哈希表中记录长度 / 哈希表长度。

22. 你了解的排序算法有什么?

答：首先是插入排序方法，包括插入排序与希尔排序。其次是选择排序方法，包括简单选择排序和堆排序。而后是交换排序方法，包括冒泡排序和快速排序。最后是其他类排序，例如归并排序、基数排序、败者树、外部排序等。
不同排序方法的稳定性与时间复杂度不同：

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

CSDN @别码了W哥

23. 快速排序的复杂度分析?

答：快速排序的时间复杂度应该为O(n*递归深度)，而递归深度则是由递归树的高度决定。快速排序中的递归树是一棵二叉树，因此其最大的高度是n（即所有序列都有序时），最低高度是log2n向下取整+1，因此快速排序的最好时间复杂度为O(nlog2n)，最坏时间复杂度为O(n^2)。

24. 大、小根堆的建堆过程？堆排序的过程?

答：以大根堆为例，首先以数组的形式存储堆，因为堆是一颗完全二叉树，因此对于下标为i的结点，其左孩子的下标为2*i*+1，*右孩子的小标为*2i+2。首先将初始堆（待排序序列）按照顺序存储在数组中。初建堆的过程将从最后一个分支节点开始，若结点的值小于其左孩子与右孩子中较大的一个，则与该孩子结点进行交换，直到无法再交换为止。然后再对倒数第二个分支结点进行调整，直到完成对根节点的调整即完成大根堆的建堆过程。

若需要进行堆排序，则将根节点与最后一个叶子结点进行交换，并将最后一个叶子结点移除序列（该结点即为序列中最大的结点），而后对新的根节点进行调整形成堆结构，重复上述过程直到所有结点都移除堆结构即可得到已排序的序列。

25. 归并排序的最坏时间复杂度优于快排，为什么我们还是选择快排?

答：快排是一个in-place排序算法，而归并排序是一个out-place排序算法，期间存在O（n）的空间复杂度，这就意味着堆内存的拷贝释放等底层操作，当排序数量大时，这些时间开销将会增大。另一个原因是，快排通常情况下多为 平均时间复杂度 ，最坏的情况出现的概率往往很小。