数据结构

1. 常用查找算法？具体实现

顺序查找、二分查找、插值查找、斐波那契查找、树表查找、分块查找、哈希查找

2. 常用排序算法？具体实现，哪些是稳定的，时间复杂度、空间复杂度，快速排序非递归如何实现？快排的优势？

3. 图的常用算法？

1) 深度广度遍历；
2) 广度优先遍历；
3) 最短路径 Floyed 算法；
4) 最短路径 Dijktral 算法；
5) 最小生成树，Prime 算法；
6) 最小生成树 Kuraul 算法；

4. 哈夫曼编码？

1) 给定 n 个权值作为 n 个叶子结点，构造一棵二叉树，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。
2) 哈夫曼树的构造
   将节点权重进行升序排序；
   选择权重最小的两个节点，将两个节点的和作为新节点，将新节点加入数组中；
   重复以上步骤，最后数组中剩下一个值，该值就是树的带权路径长度，即哈夫曼树；

5. ***AVL 树、B+树、红黑树、B 树 B+树区别，B+树应用在哪里？

1) 一个 m 阶的 B+树具有如下特征：
   ① 有 k 个子树的中间节点包含有 k 个元素（B 树中是 k-1 个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。
   ② 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
   ③ 所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。

④ 在 B+树中，只有叶子节点带有数据，其余中间节点仅仅是索引，没有关联任何数据。

2) B+树的优势

① 单一节点存储更多的元素，使得查询的 IO 次数更少；

② 所有查询都要查询到叶子节点，查询性能稳定；

③ 所有叶子节点形成有序链表，便于范围查询；

3) B+树与 B-树的区别

6. 为什么使用红黑树，什么情况使用 AVL 树。红黑树比 AVL 树有什么优点。

1) 首先红黑树是不符合 AVL 树的平衡条件的，即每个节点的左子树和右子树的高度最多差 1 的二叉查找树。但是提出了为节点增加颜色，红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而 AVL 是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。所以红黑树的插入效率更高!!! 红黑树的查询性能略微逊色于 AVL 树，因为他比 avl 树会稍微不平衡最多一层，也就是说红黑树的查询性能只比相同内容的 avl 树最多多一次比较，但是，红黑树在插入和删除上完爆 avl 树，avl 树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于 avl 树为了维持平衡的开销要小得多。
2) 如果你的应用中，搜索的次数远远大于插入和删除，那么选择 AVL，如果搜索，插入删除次数几乎差不多，应该选择 RB。

7. 单链表如何判断有环?

1) 在链表头部设置两个指针，一个每次向后移动两个位置，一个每次向后移动一个位置。两个指针遍历链表过程中，如果快指针到达链表尾还没有和慢指针相遇，说明链表无环，反之有环；
2) 环的长度，从两个指针的交点开始，移动指针，当指针再次指向两个指针的交点的时候，就可以求出环的长度；
3) 环的入口，一个指针从头开始，一个指针指向(1)中的环中快慢指针的交点，开始遍历，直到这两个指针相遇，两个指针相遇的点就是环的入口点。

8. 如何判断一个图是否连同?

可以用 DFS（O（v^2））和 BFS(O(v+e))的思想都能实现，只要从一个点出发，然后判断是否能遍历完所有的点。

9. hash 用在什么地方，解决 hash 冲突的几种方法?负载因子?

1) 如何构造哈希函数
   a) 数字分析法；
   b) 平方取中法；
   c) 除留余数法；
   d) 伪随机数法；
2) 处理冲突
   e) 线性探测；
   f) 二次探测；
   g) 伪随机数探测；
   h) 拉链探测。
3) 如果负载因子是默认的 0.75，HashMap(16)的时候，占 16 个内存空间，实际上只用到了 12 个，超过 12 个就扩容。

   如果负载因子是 1 的话，HashMap(16)的时候，占 16 个内存空间，实际上会填满 16 个以后才会扩容。增大负载因子可以减少 hash 表的内存，如果负载因子是 0.75，hashmap(16)最多可以存储 12 个元素，想存第 16 个就得扩容成 32。如果负载因子是 1，hashmap(16)最多可以存储 16 个元素。同样存 16 个元素，一个占了 32 个空间，一个占了 16 个空间的内存。

10. n 个节点的二叉树的所有不同构的个数

11. 二叉树的公共祖先，排序二叉树的公共祖先

1) 搜索二叉树

   从树的根节点开始和两个节点进行比较，如果根节点大于两个节点值，则去根节点的左孩子去进行查找；如果根节点小于两个节点值，则去根节点的右孩子去进行查找；当根节点大于其中一个节点，小于其中一个节点，则该节点是最近的祖先节点。

   方法一首先给出 node1 的父节点 node1->_parent，然后将 node1 的所有父节点依次和 node2->parent 作比较，如果发现两个节点相等，则该节点就是最近公共

祖先，直接将其返回。如果没找到相等节点，则将 node2 的所有父节点依次和 node1->_parent->_parent 作比较.....直到 node1->_parent==NULL。

方法二给定的两个节点都含有父节点，因此，可将这两个节点看做是两个链表的头结点，将求两个节点的最近公共祖先节点转化为求两链表的交点，这两个链表的尾节点都是根节点。

2) 一般二叉树

方法一，将从根节点到 node1 的路径保存在数组中；将从根节点到 node2 的路径保存在数组中；两个数组从头开始遍历，直到找到不相同的两个值，该值前一个就是最近祖先；

方法二，从根节点开始遍历，如果 node1 和 node2 中的任一个和 root 匹配，那么 root 就是最低公共祖先。 如果都不匹配，则分别递归左、右子树，如果有一个 节点出现在左子树，并且另一个节点出现在右子树，则 root 就是最低公共祖先. 如果两个节点都出现在左子树，则说明最低公共祖先在左子树中，否则在右子树。

12. 节点的最大距离

1) 如果具有最远距离的两个节点经过根节点，那么最远的距离就是左边最深的深度加上右边最深的深度之和；如果具有最远距离的两个节点之间的路径不经过根节点，那么最远的距离就在根节点的其中一个子树上的两个叶子节点。

```
int _Height(root, distance)
{
    if(root 为空)
        return 0;
    left = _Height(左子树);
    right = _Height(右子树);
    if(left+right>distance)
        distance = left+right;
    return  左子树、右子树较大加 1;
}
```

13. 把一颗二叉树原地变成一个双向链表

递归中序遍历；

14. 二叉树的所有路径

```
vector<string> binaryTreePaths(TreeNode* root) {
    // Write your code here
    vector<string> res;
    if(root==NULL) return res;
    binaryTreePathsCore(root,res,to_string(root->val));
    return res;
}

void binaryTreePathsCore(TreeNode* root,vector<string> &str,string strpath){

    if(root->left==NULL&&root->right==NULL){
        //叶子结点
        str.push_back(strpath);
        return;
    }
    if(root->left!=NULL){
        binaryTreePathsCore(root->left,str,strpath+"->"+to_string(root->left->val));
    }
    if(root->right!=NULL){
        binaryTreePathsCore(root->right,str,strpath+"->"+to_string(root->right->val));
    }
}
```

15. 二叉树中寻找每一层中最大值?

1) 利用队列来分别将每层的树添加进队列进行分析，先记录下第一个值作为最大值并弹出，然后往后边比较边弹出，如果当前值比前面的最大值还要大则替换当前最大值。在弹出每次的节点时将孩子节点加进队列。直到整棵树被遍历完。
2) 伪代码

```
if(根节点为空)
    返回
申请队列 Q，将根节点入队列
While(队列不空)
```

```
{
    s=队列长度
    for(i=0;i < s;i++)
    {
        比较队首元素，并将队首元素出栈
        队首左孩子入栈，右孩子入栈
    }
}
```

16. 最大深度、最小深度、会否是平衡树

1) 二叉树的深度等于二叉树的高度，也就等于根节点的高度。根节点的高度为左右子树的高度较大者+1。

```
int deepth(root)
{
    if(root 为空)
        return 0;
    else
    {
        left = deepth(左孩子);
        right = deepth(右孩子);
        return left>right? left+1:right+1;
    }
}
```

2) 求最大深度的时候，只需要比较左右子树的深度，取较大者+1 就行了；但是求最小深度的时候，需要区分双子树与单子树，双子树时，深度较小者+1，单子树时（即左右子树有一颗为空时）为深度较大者+1。

```
int deepth(root)
{
    if(root 为空)
        return 0;
    left = deepth(左孩子);
    right = deepth(右孩子);
    if(左孩子或有孩子为空)
        return  不为空的深度+1;
```

```
        return  左右较小者+1;
    }
```

3) 判断平衡二叉树，只需要判断平衡因子小于 1 即可，递归判断左右节点是否是平衡的即可；

```
    bool isBalance(root)
    {
        left =  根节点左孩子高度;
        right =  根节点右孩子高度;
        if(right 与 left 不满足平衡因子)
            return false;
        return isBalance(左孩子)&&isBalance(右孩子);
    }
```

17. 二叉树中叶子节点的数量

1) 叶子节点就是左右孩子都是空的节点，在进行遍历的过程中，判断是否为叶子节点，如果是叶子节点，将计数器加 1；

2) 伪代码

```
    void leafNodeNum(root,k)
    {
        if(树为空)
            return;
        if(root 不为空)
        {
            if(叶子节点)
                k++;
            leafNodeNum(root->左孩子,k);
            leafNodeNum(root->右孩子,k);
        }
    }
```

18. 交换左右孩子、二叉树镜像

1) 递归交换左右孩子，从跟节点开始交换，节点的左右孩子不都为空，则进行交换操作；否则返回；

```
    void exchangeChild(root)
```

```
{
    if(root->left 空&&root->right 空)
        return;
    swap(root->left,root->right);
    exchangeChild(root->left);
    exchangeChild(root->right);
}
```

19. 两个二叉树是否相等

1) 判断两颗树的根节点是否相同，如果不相同返回 false，如果相同则递归判断根节点的左右子节点；如果两颗树中有一个树没有遍历完则说明不相等；两棵树都为空则两棵树相等；两棵树一颗为空一颗不为空则不相等；

```
bool treesEqual(root1,root2)
{
    if(root1 空 && root2 空)
        return true;
    if(root1 空 || root2 空)
        return false;
    if(root1->data == root2->data)
        return
treesEqual(root1->left,root2->left)&&treesEqual(root1->right,root2->right);
    else
        return false;
}
```

20. 是否为完全二叉树

1) 如果一个结点有右孩子而没有左孩子，那么这棵树一定不是完全二叉树。

如果一个结点有左孩子，而没有右孩子，那么按照层序遍历的结果，这个结点之后的所有结点都是叶子结点这棵树才是完全二叉树。

如果一个结点是叶子结点，那么按照层序遍历的结果，这个结点之后的所有结点都必须是叶子结点这棵树才是完全二叉树。

用一个标记变量 leaf 标记，当一个节点有左孩子无右孩子，leaf=true。之后所有的节点必须为叶子节点。

21. 是否为对称二叉树

```
bool isSymmetrical(root1,root2)
{
    if(root1 空 && root2 空)
        return true;
    if(root1 空 || root2 空)
        return false;
    if(root1->data != root2->data)
        return false;
    else
        return
isSymmetrical(root1->left,root2->right)&&isSymmetrical(root1->right,root2->left);
}
```

22. 判断 B 是否为 A 的子树

1) 找值相同的根结点（遍历解决）

判断两结点是否包含（递归：值、左孩子、右孩子分别相同）

```
bool isPart(root1,root2)
{
    if(root1 空&&root2 空)
        return true;
    if(root1 空 || root2 空)
        return false;
    if(root1->data != root2->data)
        return false;
    else
        return isPart(root1->left,root2->left)&&isPart(root1->right,root2->right);
}
bool isPartTree(root1,root2)
{
    if (root1 不空 && root2 不空)
    {
        if (root1->data == root2->data)
```

```
            result = IsPart(root1, root2);
        if (!result)
            result = IsPartTree(root1->left, root2);
        if (!result)
            result = IsPartTree(root1->right, root2);
    }
    return result;
}
```

23. 构建哈夫曼树

24. 手写单链表反转？删除指定的单链表的一个节点

1. 单链表反转：尾插法转头查法；设置三个指针，当前节点指针，下一个节点指针，上一个节点指针，一开始上一个节点指针置为空；最后当下一个节点指针为空时说明到达最后节点，则返回该节点指针。
2. 删除指定节点：如果我们把要删除节点的下一个节点的内容复制到需要删除的节点上，然后把删除节点的下一个节点删除，就可以完成删除该节点，同时时间复杂度为 O(1)。如果是尾节点，只能遍历删除，如果只有一个节点，还要删除头节点。

25. 实现一个循环队列

循环中 front 与 rear 的求值。

队首指针进 1：front = (front+1)%MaxSize;

队尾指针进 1：rear = (rear+1)%MaxSize;

队空：rear == front;

队满：(rear+1)%MaxSize == front

26. Top K 问题

1) 如果要找前 K 个最大的数，我们用最小堆，每次用堆顶元素和遍历的数比，如果堆顶元素小，则让堆顶元素的值等于它，然后向下调整
2) 如果要找前 K 个最小的数，我们用最大堆，每次用堆顶元素和遍历的数比，如果堆顶元素大，则让堆顶元素的值等于它，然后向下调整
3) 用快速排序将数组进行排序，然后将数组输出

4) 两者的时间复杂度都是 O（$n\log_2 n$）,快排的空间复杂度为 O($\log_2 n$)，堆排序的空间复杂度为 O(1)

27. 求一颗树的最大距离

对于二叉树，若要两个节点 U，V 相距最远，有两种情况：

1，从 U 节点到 V 节点之间的路径经过根节点

2，从 U 节点到 V 节点之间的路径不经过根节点，这种情况下，U，V 节点必定在根节点的左子树或者右子树上，这样就转化为求以根节点的孩子节点为根节点的二叉树中最远的两个节点间的距离

28. KMP

核心是 next（）函数的书写；

29. 数组和链表的区别?

1) 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况，即数组的大小一旦定义就不能改变。当数据增加时，可能超出原先 定义的元素个数；当数据减少时，造成内存浪费；链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删 除数据项时，需要移动其它数据项）。
2) （静态)数组从栈中分配空间（用 NEW 创建的在堆中），对于程序员方便快速,但是自由度小；链表从堆中分配空间，自由度大但是申请管理比较麻烦.
3) 数组在内存中是连续存储的，因此，可以利用下标索引进行随机访问；链表是链式存储结构，在访问元素的时候只能通过线性的方式由前到后顺序访问，所以访问效率比数组要低。

30. 逆序对思路

31. 100 个有序数组合并

归并排序，两个数组合并生成 50 个数组，生成 25 个数组，13 个数组，6 个数组，3 个数组，2 个数组，1 个数组

32. 使用递归和非递归求二叉树的深度

33. 索引、链表的优缺点?

34. 找一个点为中心的圆里包含的所有的点。

35. 字典树的理解

1) 字典树，又称单词查找树，是一种树形结构，是一种哈希树的变种；
2) 根节点不包含字符，除根节点外的每一个子节点都包含一个字符；
   从根节点到叶子节点，路径上经过的字符链接起来，就是该节点对应的字符串；
   每个节点的所有子节点包含的字符都不同；
3) 典型应用是用于统计，排序和保存大量的字符串(不仅限于字符串)，经常被搜索引擎系统用于文本词频统计。

36. 快速排序的优化

1) 当我们每次划分的时候选择的基准数接近于整组数据的最大值或者最小值时，快速排序就会发生最坏的情况，但是每次选择的基准数都接近于最大数或者最小数的概率随着排序元素的增多就会越来越小，我们完全可以忽略这种情况。但是在数组有序的情况下，它也会发生最坏的情况，为了避免这种情况，我们在选择基准数的时候可以采用三数取中法来选择基准数。三数取中法：选择这组数据的第一个元素、中间的元素、最后一个元素，这三个元素里面值居中的元素作为基准数。
2) 当划分的子序列很小的时候(一般认为小于 13 个元素左右时)，我们在使用快速排序对这些小序列排序反而不如直接插入排序高效。因为快速排序对数组进行划分最后就像一颗二叉树一样，当序列小于 13 个元素时我们再使用快排的话就相当于增加了二叉树的最后几层的结点数目，增加了递归的次数。所以我们在当子序列小于 13 个元素的时候就改用直接插入排序来对这些子序列进行排序。

37. 海量数据的 bitmap 使用原理

1) BitMap 解决海量数据寻找重复、判断个别元素是否在海量数据当中等问题；
2) 40 亿个 int 占（40 亿*4）/1024/1024/1024 大概为 14.9G 左右，很明显内存只有 2G，放不下，因此不可能将这 40 亿数据放到内存中计算；40 亿个 int 需要的内存空间为 40 亿/8/1024/1024 大概为 476.83MB;

算法

1. 动态规划，最长公共子序列

2. 分治与递归

3. 贪心算法，背包问题

4. BFS,DFS,地杰斯特拉算法,佛洛依德算法

5. 动态规划的回文字符串?

6. 排序算法?时间复杂度？稳定性算法?

7. 查找算法

8. 字符串匹配?

9. 求一个数开根号（二分）

10. 万个数找到第 20 个大小?

11. 设计抢红包算法

12. 字符串中最长不重复子串

13. 动态规划与分支界限的差异，背包问题和分支界限的差异

14. 出 1-n 的子集。比如 123，有 1，2,3,12，13,123,23.