**● 请你来说一说红黑树和AVL树的定义，特点，以及二者区别**

**参考回答：**

平衡二叉树（AVL树）：

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。一句话表述为：以树中所有结点为根的树的左右子树高度之差的绝对值不超过1。将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

红黑树：

红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;

区别：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

**● 请你说一下哈夫曼编码**

**参考回答：**

哈夫曼编码是哈夫曼树的一种应用，广泛用于数据文件压缩。哈夫曼编码算法用字符在文件中出现的频率来建立使用0，1表示个字符的最优表示方式，其具体算法如下：

(1)哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树T。

(2)算法以|C|个叶结点开始，执行|C|－1次的“合并”运算后产生最终所要求的树T。

(3)假设编码字符集中每一字符c的频率是f(c)。以f为键值的优先队列Q用在贪心选择时有效地确定算法当前要合并的2棵具有最小频率的树。一旦2棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列Q。经过n－1次的合并后，优先队列中只剩下一棵树，即所要求的树T。

**● 请你回答一下map底层为什么用红黑树实现**

**参考回答：**

1、红黑树：

红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;

2、平衡二叉树（AVL树）：

红黑树是在AVL树的基础上提出来的。

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。

AVL树中所有结点为根的树的左右子树高度之差的绝对值不超过1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

3、红黑树较AVL树的优点：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是O(logn)，且性能稳定，所以STL里面很多结构包括map底层实现都是使用的红黑树。

**● 请你介绍一下B+树**

**参考回答：**

B+是一种多路搜索树，主要为磁盘或其他直接存取辅助设备而设计的一种平衡查找树，在B+树中，每个节点的可以有多个孩子，并且按照关键字大小有序排列。所有记录节点都是按照键值的大小顺序存放在同一层的叶节点中。相比B树，其具有以下几个特点：

每个节点上的指针上限为2d而不是2d+1（d为节点的出度）

内节点不存储data,只存储key

叶子节点不存储指针

**● 请你说一说map和unordered\_map的底层实现**

**参考回答：**

map底层是基于红黑树实现的，因此map内部元素排列是有序的。而unordered\_map底层则是基于哈希表实现的，因此其元素的排列顺序是杂乱无序的。

**● 请你回答一下map和unordered\_map优点和缺点**

**参考回答：**

对于map，其底层是基于红黑树实现的，优点如下：

1)有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作

2)map的查找、删除、增加等一系列操作时间复杂度稳定，都为logn

缺点如下：

1）查找、删除、增加等操作平均时间复杂度较慢，与n相关

对于unordered\_map来说，其底层是一个哈希表，优点如下：

查找、删除、添加的速度快，时间复杂度为常数级O(c)

缺点如下：

因为unordered\_map内部基于哈希表，以（key,value）对的形式存储，因此空间占用率高

Unordered\_map的查找、删除、添加的时间复杂度不稳定，平均为O(c)，取决于哈希函数。极端情况下可能为O(n)

**● 请你回答一下epoll怎么实现的**

**参考回答：**

Linux epoll机制是通过红黑树和双向链表实现的。 首先通过epoll\_create()系统调用在内核中创建一个eventpoll类型的句柄，其中包括红黑树根节点和双向链表头节点。然后通过epoll\_ctl()系统调用，向epoll对象的红黑树结构中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。最后通过epoll\_wait()系统调用判断双向链表是否为空，如果为空则阻塞。当文件描述符状态改变，fd上的回调函数被调用，该函数将fd加入到双向链表中，此时epoll\_wait函数被唤醒，返回就绪好的事件。

**● 请你说一说Top(K)问题**

**参考回答：**

1、直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前K个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出top K个数据，所以该方法并不十分高效，不建议使用。

2、快速排序的变形 （只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前K个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号index刚好等于K，那么这个划分元以及它左边的数，刚好就是前K个最大的元素；如果index  > K，那么前K大的数据在index的左边，那么就继续递归的从index-1个数中进行一趟排序；如果index < K，那么再从划分元的右边继续进行排序，直到找到序号index刚好等于K为止。再将前K个数进行排序后，返回Top K个元素。这种方法就避免了对除了Top K个元素以外的数据进行排序所带来的不必要的开销。

3、最小堆法

这是一种局部淘汰法。先读取前K个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的K个数。

4、分治法

将全部数据分成N份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的K个数。此时剩下N\*K个数据，如果内存不能容纳N\*K个数据，则再继续分治处理，分成M份，找出每份数据中最大的K个数，如果M\*K个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5、Hash法

如果这些数据中有很多重复的数据，可以先通过hash法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

**● 请你说一说C++两种map**

**参考回答：**

unordered\_map（哈希表)和map（红黑树）

**● 请问红黑树了解吗**

**参考回答：**

参考博客<https://blog.csdn.net/tanrui519521/article/details/80980135>

**● 请你说一说红黑树的性质还有左右旋转**

**参考回答：**

考察点：算法

公司：京东，阿里巴巴

1）平衡二叉树（AVL树）：

红黑树是在AVL树的基础上提出来的。

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。

AVL树中所有结点为根的树的左右子树高度之差的绝对值不超过1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

2）红黑树：

红黑树是在AVL树的基础上发展而来的。红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

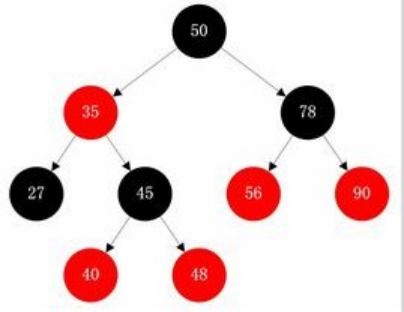
2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;

从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。恢复红黑属性需要少量(O(log n))的颜色变更(这在实践中是非常快速的)并且不超过三次树旋转(对于插入是两次)。这允许插入和删除保持为 O(log n) 次，



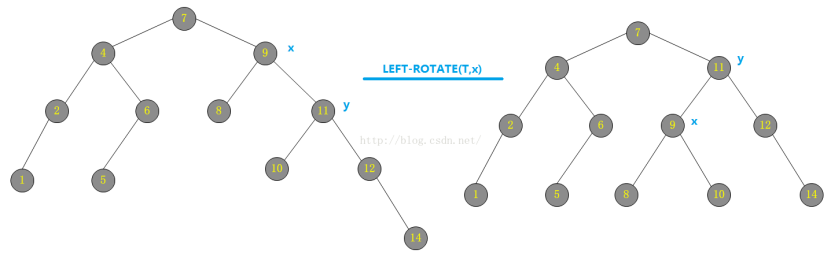
3）红黑树较AVL树的优点：

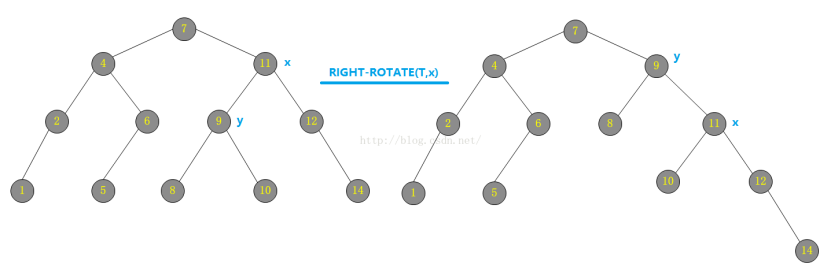
AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是O(logn)，且性能稳定，所以STL里面很多结构包括map底层实现都是使用的红黑树。

4）红黑树旋转：

旋转：红黑树的旋转是一种能保持二叉搜索树性质的搜索树局部操作。有左旋和右旋两种旋转，通过改变树中某些结点的颜色以及指针结构来保持对红黑树进行插入和删除操作后的红黑性质。

左旋：对某个结点x做左旋操作时，假设其右孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的左孩子，y的左孩子成为x的右孩子。

右旋：对某个结点x做右旋操作时，假设其左孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的右孩子，y的右孩子成为x的左孩子。

**● 请你实现二叉树的层序遍历并输出**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | void layerTrace(BTreeNode \*T)  {  if(T== nullptr)return;  BTreeNode \*p=T;  queue<BTreeNode\*>q;  q.push(p);  while(!q.empty())  {  p=q.front();  q.pop();  cout<<<<p->data;  if(p->left!= nullptr)q.push(p->left);  if(p->right!= nullptr)q.push(p->right);  }  }``` |

**● 手写代码：二叉树序列化反序列化**

**参考回答：**

> 序列化：必须保存一个中序遍历结果，然后外加一个前序或者后序遍历结果

>反序列化：根据两次遍历生成的结果恢复二叉树，代码如下(前序和中序)：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | ```TreeNode\* helper(vector<int>pre,int startPre,int endPre,vector<int>in,int startIn,int endIn)  {  if(startPre>endPre||startIn>endIn)  return nullptr;  TreeNode \* root=new TreeNode(pre[startPre]);  for(int i=startIn;i<=endIn;++i)  {  if(in[i]==pre[startPre])  {  root->left=helper(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);  root->right=helper(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);  break;  }  }  return root;  }  TreeNode\* reConstructBinaryTree(vector<int> pre,vector<int> vin)  {  TreeNode \*root=helper(pre,0,pre.size()-1,vin,0,vin.size()-1);  return root;  } |

**● 请说一说你理解的stack overflow，并举个简单例子导致栈溢出**

**参考回答：**

栈溢出概念：

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致栈中与其相邻的变量的值被改变。

栈溢出的原因：

1. 局部数组过大。当函数内部的数组过大时，有可能导致堆栈溢出。局部变量是存储在栈中的，因此这个很好理解。解决这类问题的办法有两个，一是增大栈空间,二是改用动态分配，使用堆（heap）而不是栈（stack）。

2. 递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。

3. 指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

栈溢出例子：

#include <stdio.h>

#include <string.h>

int main(int argc, char\* argv[]) {

char buf[256];

strcpy(buf,argv[1]);

printf("Input:%s\n",buf);

return 0;

}

上述代码中的strcpy(buf,argv[1]);这一行发生了缓冲区溢出错误，因为源缓冲区内容是用户输入的。

**● 请你回答一下栈和堆的区别，以及为什么栈要快**

**参考回答：**

堆和栈的区别：

堆是由低地址向高地址扩展；栈是由高地址向低地址扩展

堆中的内存需要手动申请和手动释放；栈中内存是由OS自动申请和自动释放，存放着参数、局部变量等内存

堆中频繁调用malloc和free,会产生内存碎片，降低程序效率；而栈由于其先进后出的特性，不会产生内存碎片

堆的分配效率较低，而栈的分配效率较高

栈的效率高的原因：

栈是操作系统提供的数据结构，计算机底层对栈提供了一系列支持：分配专门的寄存器存储栈的地址，压栈和入栈有专门的指令执行；而堆是由C/C++函数库提供的，机制复杂，需要一些列分配内存、合并内存和释放内存的算法，因此效率较低。

**● 手写代码：两个栈实现一个队列**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Solution  {  public:  void push(int node) {  stack1.push(node);  }  int pop() {  if(stack2.size()!=0){  int tmp = stack2.top();  stack2.pop();  return tmp;  }  else{  while(stack1.size()!=0){  int tmp = stack1.top();  stack1.pop();  stack2.push(tmp);  }  return pop();  }  }      private:  stack<int> stack1;  stack<int> stack2;   }； |

**● 请你来说一下堆和栈的区别**

**参考回答：**

1）申请方式：

栈由系统自动分配和管理，堆由程序员手动分配和管理。

2）效率：

栈由系统分配，速度快，不会有内存碎片。

堆由程序员分配，速度较慢，可能由于操作不当产生内存碎片。

3）扩展方向

栈从高地址向低地址进行扩展，堆由低地址向高地址进行扩展。

4）程序局部变量是使用的栈空间，new/malloc动态申请的内存是堆空间，函数调用时会进行形参和返回值的压栈出栈，也是用的栈空间。

**● 请你说一说小根堆特点**

**参考回答：**

堆是一棵完全二叉树（如果一共有h层，那么1~h-1层均满，在h层可能会连续缺失若干个右叶子）。

1）小根堆

若根节点存在左子女则根节点的值小于左子女的值；若根节点存在右子女则根节点的值小于右子女的值。

2）大根堆

若根节点存在左子女则根节点的值大于左子女的值；若根节点存在右子女则根节点的值大于右子女的值。

**● 请你回答一下Array&List， 数组和链表的区别**

**参考回答：**

数组的特点：

数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。数组的插入数据和删除数据效率低，插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动。但数组的随机读取效率很高。因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。数组需要预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间。并且数组不利于扩展，数组定义的空间不够时要重新定义数组。

链表的特点：

链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。不指定大小，扩展方便。链表大小不用定义，数据随意增删。

各自的优缺点

数组的优点：

1. 随机访问性强

2. 查找速度快

数组的缺点:

1. 插入和删除效率低

2. 可能浪费内存

3. 内存空间要求高，必须有足够的连续内存空间。

4. 数组大小固定，不能动态拓展

链表的优点:

1. 插入删除速度快

2. 内存利用率高，不会浪费内存

3. 大小没有固定，拓展很灵活。

链表的缺点:

不能随机查找，必须从第一个开始遍历，查找效率低

**● 一个长度为N的整形数组，数组中每个元素的取值范围是[0,n-1],判断该数组否有重复的数，请说一下你的思路并手写代码**

**参考回答：**

把每个数放到自己对应序号的位置上，如果其他位置上有和自己对应序号相同的数，那么即为有重复的数值。时间复杂度为O(N),同时为了节省空间复杂度，可以在原数组上进行操作，空间复杂度为O(1)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | bool IsDuplicateNumber(int \*array, int n)  {  if(array==NULL) return false;  int i,temp;  for(i=0;i<n;i++)  {  while(array[i]!=i)  {  if(array[array[i]]==array[i])  return true;  temp=array[array[i]];  array[array[i]]=array[i];  array[i]=temp;  }  }  return false;  } |

**● 请你来手写一下快排的代码**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | int once\_quick\_sort(vector<int> &data, int left, int right)  {  int key = data[left];  while (left < right)  {  while (left < right && key <= data[right])  {  right--;  }  if (left < right)  {  data[left++] = data[right];  }  while (left < right && key > data[left])  {  left++;  }  if (left < right)  {  data[right--] = data[left];  }  }  data[left] = key;  return left;  }  int quick\_sort(vector<int> & data, int left, int right)  {  if (left >= right )  {  return 1;  }  int middle = 0;  middle = once\_quick\_sort(data, left, right);  quick\_sort(data, left, middle-1);  quick\_sort(data, middle + 1, right);  }； |

**● 请你手写一下快排的代码**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | nt once\_quick\_sort(vector<int> &data, int left, int right)  {  int key = data[left];  while (left < right)  {  while (left < right && key <= data[right])  {  right--;  }  if (left < right)  {  data[left++] = data[right];  }  while (left < right && key > data[left])  {  left++;  }  if (left < right)  {  data[right--] = data[left];  }  }  data[left] = key;  return left;  }  int quick\_sort(vector<int> & data, int left, int right)  {  if (left >= right )  {  return 1;  }  int middle = 0;  middle = once\_quick\_sort(data, left, right);  quick\_sort(data, left, middle-1);  quick\_sort(data, middle + 1, right);  } |

**● 请问求第k大的数的方法以及各自的复杂度是怎样的，另外追问一下，当有相同元素时，还可以使用什么不同的方法求第k大的元素**

**参考回答：**

首先使用快速排序算法将数组按照从大到小排序，然后取第k个，其时间复杂度最快为O(nlogn)

使用堆排序，建立最大堆，然后调整堆，知道获得第k个元素，其时间复杂度为O(n+klogn)

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有k-1个数则为第k大的数

利用快排思想，从数组中随机选择一个数i，然后将数组分成两部分Dl,Dr，Dl的元素都小于i,Dr的元素都大于i。然后统计Dr元素个数，如果Dr元素个数等于k-1,那么第k大的数即为k，如果Dr元素个数小于k,那么继续求Dl中第k-Dr大的元素；如果Dr元素个数大于k,那么继续求Dr中第k大的元素。

当有相同元素的时候，

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有k-1个数则为第k大的数，平均情况下时间复杂度为O(n)

**● 请你来介绍一下各种排序算法及时间复杂度**

**参考回答：**

插入排序：对于一个带排序数组来说，其初始有序数组元素个数为1，然后从第二个元素，插入到有序数组中。对于每一次插入操作，从后往前遍历当前有序数组，如果当前元素大于要插入的元素，则后移一位；如果当前元素小于或等于要插入的元素，则将要插入的元素插入到当前元素的下一位中。

希尔排序：先将整个待排序记录分割成若干子序列，然后分别进行直接插入排序，待整个序列中的记录基本有序时，在对全体记录进行一次直接插入排序。其子序列的构成不是简单的逐段分割，而是将每隔某个增量的记录组成一个子序列。希尔排序时间复杂度与增量序列的选取有关，其最后一个值必须为1.

归并排序：该算法采用分治法；对于包含m个元素的待排序序列，将其看成m个长度为1的子序列。然后两两合归并，得到n/2个长度为2或者1的有序子序列；然后再两两归并，直到得到1个长度为m的有序序列。

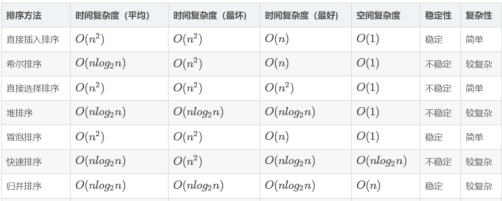
冒泡排序：对于包含n个元素的带排序数组，重复遍历数组，首先比较第一个和第二个元素，若为逆序，则交换元素位置；然后比较第二个和第三个元素，重复上述过程。每次遍历会把当前前n-i个元素中的最大的元素移到n-i位置。遍历n次，完成排序。

快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

选择排序：每次循环，选择当前无序数组中最小的那个元素，然后将其与无序数组的第一个元素交换位置，从而使有序数组元素加1，无序数组元素减1.初始时无序数组为空。

堆排序：堆排序是一种选择排序，利用堆这种数据结构来完成选择。其算法思想是将带排序数据构造一个最大堆（升序）/最小堆（降序），然后将堆顶元素与待排序数组的最后一个元素交换位置，此时末尾元素就是最大/最小的值。然后将剩余n-1个元素重新构造成最大堆/最小堆。

各个排序的时间复杂度、空间复杂度及稳定性如下：



**● 请你说一说你知道的排序算法及其复杂度**

**参考回答：**

1、冒泡排序：

从数组中第一个数开始，依次遍历数组中的每一个数，通过相邻比较交换，每一轮循环下来找出剩余未排序数的中的最大数并“冒泡”至数列的顶端。

稳定性：稳定

平均时间复杂度：O(n ^ 2)

2、插入排序：

从待排序的n个记录中的第二个记录开始，依次与前面的记录比较并寻找插入的位置，每次外循环结束后，将当前的数插入到合适的位置。

稳定性：稳定

平均时间复杂度：O(n ^ 2)

3、希尔排序（缩小增量排序）：

希尔排序法是对相邻指定距离(称为增量)的元素进行比较，并不断把增量缩小至1，完成排序。

希尔排序开始时增量较大，分组较多，每组的记录数目较少，故在各组内采用直接插入排序较快，后来增量di逐渐缩小，分组数减少，各组的记录数增多，但由于已经按di−1分组排序，文件叫接近于有序状态，所以新的一趟排序过程较快。因此希尔 排序在效率上比直接插入排序有较大的改进。

在直接插入排序的基础上，将直接插入排序中的1全部改变成增量d即可，因为希尔排序最后一轮的增量d就为1。

稳定性：不稳定

平均时间复杂度：希尔排序算法的时间复杂度分析比较复杂，实际所需的时间取决于各次排序时增量的个数和增量的取值。时间复杂度在O(n ^ 1.3)到O(n ^ 2)之间。

4、选择排序：

从所有记录中选出最小的一个数据元素与第一个位置的记录交换；然后在剩下的记录当中再找最小的与第二个位置的记录交换，循环到只剩下最后一个数据元素为止。

稳定性：不稳定

平均时间复杂度：O(n ^ 2)

5、快速排序

1）从待排序的n个记录中任意选取一个记录（通常选取第一个记录）为分区标准;

2）把所有小于该排序列的记录移动到左边，把所有大于该排序码的记录移动到右边，中间放所选记录，称之为第一趟排序；

3）然后对前后两个子序列分别重复上述过程，直到所有记录都排好序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

6、堆排序：

堆：

1、完全二叉树或者是近似完全二叉树。

2、大顶堆：父节点不小于子节点键值，小顶堆：父节点不大于子节点键值。左右孩子没有大小的顺序。

堆排序在选择排序的基础上提出的，步骤：

1、建立堆

2、删除堆顶元素，同时交换堆顶元素和最后一个元素，再重新调整堆结构，直至全部删除堆中元素。

稳定性：不稳定

平均时间复杂度：O(nlogn)

7、归并排序：

采用分治思想，现将序列分为一个个子序列，对子序列进行排序合并，直至整个序列有序。

稳定性：稳定

平均时间复杂度：O(nlogn)

8、计数排序：

思想：如果比元素x小的元素个数有n个，则元素x排序后位置为n+1。

步骤：

1）找出待排序的数组中最大的元素；

2）统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

3）对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

4）反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

稳定性：稳定

时间复杂度：O(n+k)，k是待排序数的范围。

9、桶排序：

步骤：

1）设置一个定量的数组当作空桶子； 常见的排序算法及其复杂度：

2）寻访序列，并且把记录一个一个放到对应的桶子去；

3）对每个不是空的桶子进行排序。

4）从不是空的桶子里把项目再放回原来的序列中。

时间复杂度：O(n+C) ，C为桶内排序时间。

**● 请问海量数据如何去取最大的k个**

**参考回答：**

1.直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前K个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出top K个数据，所以该方法并不十分高效，不建议使用。

2.快速排序的变形 （只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前K个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号index刚好等于K，那么这个划分元以及它左边的数，刚好就是前K个最大的元素；如果index  > K，那么前K大的数据在index的左边，那么就继续递归的从index-1个数中进行一趟排序；如果index < K，那么再从划分元的右边继续进行排序，直到找到序号index刚好等于K为止。再将前K个数进行排序后，返回Top K个元素。这种方法就避免了对除了Top K个元素以外的数据进行排序所带来的不必要的开销。

3.最小堆法

这是一种局部淘汰法。先读取前K个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的K个数。

4.分治法

将全部数据分成N份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的K个数。此时剩下N\*K个数据，如果内存不能容纳N\*K个数据，则再继续分治处理，分成M份，找出每份数据中最大的K个数，如果M\*K个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5.Hash法

如果这些数据中有很多重复的数据，可以先通过hash法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

**● 请问快排的时间复杂度最差是多少？什么时候时间最差**

**参考回答：**

O（N2),元素本来倒序排列用时最多

**● 请问稳定排序哪几种？**

**参考回答：**

基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序

**● 请你介绍一下快排算法；以及什么是稳定性排序，快排是稳定性的吗；快排算法最差情况推导公式**

**参考回答：**

1、快排算法

根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素，直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

2、稳定排序

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

快排算法是不稳定的排序算法。例如：

待排序数组:int a[] ={1, 2, 2, 3, 4, 5, 6};

若选择a[2]（即数组中的第二个2）为枢轴，而把大于等于比较子的数均放置在大数数组中，则a[1]（即数组中的第一个2）会到pivot的右边， 那么数组中的两个2非原序。

若选择a[1]为比较子，而把小于等于比较子的数均放置在小数数组中，则数组中的两个2顺序也非原序。

3、快排最差情况推倒

在快速排序的早期版本中呢，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生啦：

1）数组已经是正序排过序的。 （每次最右边的那个元素被选为枢轴）

2）数组已经是倒序排过序的。 （每次最左边的那个元素被选为枢轴）

3）所有的元素都相同（1、2的特殊情况）

因为这些案例在用例中十分常见，所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了，但是如果输入的数组最大（或者最小元素）被选为枢轴，那最坏的情况就又来了。

快速排序，在最坏情况退化为冒泡排序，需要比较O(n2)次（n(n - 1)/2次）。

哈希表

**● 请你来说一说hash表的实现，包括STL中的哈希桶长度常数**

**参考回答：**

hash表的实现主要包括构造哈希和处理哈希冲突两个方面：

对于构造哈希来说，主要包括直接地址法、平方取中法、除留余数

法等。

对于处理哈希冲突来说，最常用的处理冲突的方法有开放定址法、再哈希法、链地址法、建立公共溢出区等方法。SGL版本使用链地址法，使用一个链表保持相同散列值的元素。

虽然链地址法并不要求哈希桶长度必须为质数，但SGI STL仍然以质数来设计哈希桶长度，并且将28个质数（逐渐呈现大约两倍的关系）计算好，以备随时访问，同时提供一个函数，用来查询在这28个质数之中，“最接近某数并大于某数”的质数。

**● 请你回答一下hash表如何rehash，以及怎么处理其中保存的资源**

**参考回答：**

C++的hash表中有一个负载因子loadFactor，当loadFactor<=1时，hash表查找的期望复杂度为O(1). 因此，每次往hash表中添加元素时，我们必须保证是在loadFactor <1的情况下，才能够添加。

因此，当Hash表中loadFactor==1时，Hash就需要进行rehash。rehash过程中，会模仿C++的vector扩容方式，Hash表中每次发现loadFactor ==1时，就开辟一个原来桶数组的两倍空间，称为新桶数组，然后把原来的桶数组中元素全部重新哈希到新的桶数组中。

**● 请你说一下哈希表的桶个数为什么是质数，合数有何不妥？**

**参考回答：**

哈希表的桶个数使用质数，可以最大程度减少冲突概率，使哈希后的数据分布的更加均匀。如果使用合数，可能会造成很多数据分布会集中在某些点上，从而影响哈希表效率。

算法：

给定一个数字数组，返回哈夫曼树的头指针

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | struct BTreeNode\* CreateHuffman(ElemType a[], int n)  {  int i, j;  struct BTreeNode \*\*b, \*q;  b = malloc(n\*sizeof(struct BTreeNode));  for (i = 0; i < n; i++)  {  b[i] = malloc(sizeof(struct BTreeNode));  b[i]->data = a[i];  b[i]->left = b[i]->right = NULL;  }  for (i = 1; i < n; i++)  {  int k1 = -1, k2;  for (j = 0; j < n; j++)  {  if (b[j] != NULL && k1 == -1)  {  k1 = j;  continue;  }  if (b[j] != NULL)  {  k2 = j;  break;  }  }  for (j = k2; j < n; j++)  {  if (b[j] != NULL)  {  if (b[j]->data < b[k1]->data)  {  k2 = k1;  k1 = j;  }  else if (b[j]->data < b[k2]->data)  k2 = j;  }  }  q = malloc(sizeof(struct BTreeNode));  q->data = b[k1]->data + b[k2]->data;  q->left = b[k1];  q->right = b[k2];  b[k1] = q;  b[k2] = NULL;  }  free(b);  return q;  } |

**● 请你说一下解决hash冲突的方法**

**参考回答：**

当哈希表关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，这样的现象称为哈希冲突。目前常用的解决哈希冲突的方法如下：

开放定址法: 当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。

再哈希法：当发生哈希冲突时使用另一个哈希函数计算地址值，直到冲突不再发生。这种方法不易产生聚集，但是增加计算时间，同时需要准备许多哈希函数。

链地址法：将所有哈希值相同的Key通过链表存储。key按顺序插入到链表中

建立公共溢出区：采用一个溢出表存储产生冲突的关键字。如果公共溢出区还产生冲突，再采用处理冲突方法处理。

**● 请你说一说哈希冲突的解决方法**

**参考回答：**

考察点：hash冲突，数据结构

公司：腾讯

1、开放定址

开放地址法有个非常关键的特征，就是所有输入的元素全部存放在哈希表里，也就是说，位桶的实现是不需要任何的链表来实现的，换句话说，也就是这个哈希表的装载因子不会超过1。它的实现是在插入一个元素的时候，先通过哈希函数进行判断，若是发生哈希冲突，就以当前地址为基准，根据再寻址的方法（探查序列），去寻找下一个地址，若发生冲突再去寻找，直至找到一个为空的地址为止。所以这种方法又称为再散列法。

有几种常用的探查序列的方法：

①线性探查

dii=1，2，3，…，m-1；这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

②二次探查

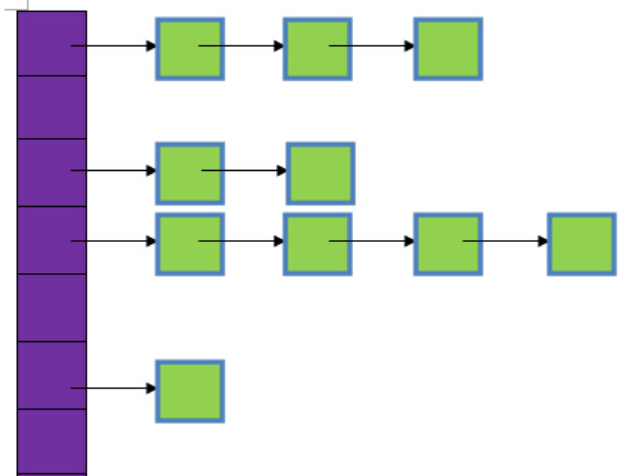
di=12，-12，22，-22，…，k2，-k2    ( k<=m/2 )；这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

③ 伪随机探测

di=伪随机数序列；具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），生成一个位随机序列，并给定一个随机数做起点，每次去加上这个伪随机数++就可以了。

2、链地址

每个位桶实现的时候，采用链表或者树的数据结构来去存取发生哈希冲突的输入域的关键字，也就是被哈希函数映射到同一个位桶上的关键字。



紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中，即链接在桶后。

3、公共溢出区

建立一个公共溢出区域，把hash冲突的元素都放在该溢出区里。查找时，如果发现hash表中对应桶里存在其他元素，还需要在公共溢出区里再次进行查找。

4、再hash

再散列法其实很简单，就是再使用哈希函数去散列一个输入的时候，输出是同一个位置就再次散列，直至不发生冲突位置。

缺点：每次冲突都要重新散列，计算时间增加。

动态规划

**● 请你手写代码：最长公共连续子序列**

**参考回答：**

int substr(string & str1, string &str2)

{

int len1 = str1.length();

int len2 = str2.length();

vector<vector<int>>dp(len1,vector<int>(len2,0));

for (int i = 0; i < len1; i++)

{

dp[i][0] = str1[i]==str1[0]?1:0;

}

for (int j = 0; j <= len2; j++)

{

dp[0][j] = str1[0]==str2[j]?1:0;

}

for (int i = 1; i < len1; i++)

{

for (int j = 1; j < len2; j++)

{

if (str1[i] == str2[j])

{

dp[i][j] = dp[i - 1][j - 1]+1;

}

}

}

int longest = 0;

int longest\_index = 0;

for (int i = 0; i < len1; i++)

{

for (int j = 0; j < len2; j++)

{

if (longest < dp[i][j])

{

longest = dp[i][j];

longest\_index = i;

}

}

}

//字符串为从第i个开始往前数longest个

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int i = longest\_index-longest+1; i <=longest\_index; i++)  {  cout << str1[i] << endl;  }  return longest;  } |

**● 手写代码：求一个字符串最长回文子串**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | int LongestPalindromicSubstring(string & a)  {  int len = a.length();  vector<vector<int>>dp(len, vector<int>(len, 0));  for (int i = 0; i < len; i++)  {  dp[i][i] = 1;  }  int max\_len = 1;  int start\_index = 0;  for (int i= len - 2; i >= 0; i--)  {  for (int j = i + 1; j < len; j++)  {  if (a[i] == a[j])  {  if (j - i == 1)  {  dp[i][j] = 2;  }  else  {  if (j - i > 1)  {  dp[i][j] = dp[i + 1][j - 1] + 2;  }  }  if (max\_len < dp[i][j])  {  max\_len =  dp[i][j];  start\_index = i;  }  }  else  {  dp[i][j] = 0;  }  }  }  cout << "max len is " << max\_len << endl;  cout << "star index is" << start\_index << endl;  return max\_len;  } |

**● 手写代码：查找最长回文子串**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | int LongestPalindromicSubstring(string & a)  {  int len = a.length();  vector<vector<int>>dp(len, vector<int>(len, 0));  for (int i = 0; i < len; i++)  {  dp[i][i] = 1;  }  int max\_len = 1;  int start\_index = 0;  for (int i= len - 2; i >= 0; i--)  {  for (int j = i + 1; j < len; j++)  {  if (a[i] == a[j])  {  if (j - i == 1)  {  dp[i][j] = 2;  }  else  {  if (j - i > 1)  {  dp[i][j] = dp[i + 1][j - 1] + 2;  }  }  if (max\_len < dp[i][j])  {  max\_len =  dp[i][j];  start\_index = i;  }  }  else  {  dp[i][j] = 0;  }  }  }  cout << "max len is " << max\_len << endl;  cout << "star index is" << start\_index << endl;  return max\_len;  } |

链表

**● 请你手写代码，如何合并两个有序链表**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Solution {  public:  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  if(l1 == NULL)  {  return l2;  }  if(l2 == NULL)  {  return l1;  }  if(l1->val < l2->val)  {  l1->next=mergeTwoLists(l1->next,l2);  return l1;  }  else  {  l2->next=mergeTwoLists(l1,l2->next);  return l2;  }  }  }; |

**● 手写代码：反转链表**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Void reversal\_list(mylist \* a\_list)  {  mylist \* forward\_node = nullptr;  mylist \* cur\_node = a\_list->next;  mylist\* next\_node = cur\_node->next;  if(cur\_node == nullptr)  {  return ;  }  while(1)  {  cur\_node->next = forward\_node;  forward\_node = cur\_node;  cur\_node = next\_node;  if(cur\_node == nullptr)  {  break;  }  next\_node = cur\_node->next;  }  a\_list->next = forward\_node;  } |

**● 判断一个链表是否为回文链表，说出你的思路并手写代码**

**参考回答：**

思路：使用栈存储链表前半部分，然后一个个出栈，与后半部分元素比较，如果链表长度未知，可以使用快慢指针的方法，将慢指针指向的元素入栈，然后如果快指针指向了链表尾部，此时慢指针指向了链表中间

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | bool is\_palindromic\_list2(mylist \*a\_list)  {  if(a\_list == nullptr)  {  return false;  }  stack<int>list\_value;  mylist \* fast =a\_list;  mylist \*slow =a\_list;  while(fast->next!=nullptr && fast->next->next!=nullptr)  {  list\_value.push(slow->next->value);  slow = slow->next;  fast = fast->next->next;  }  cout<<"middle elem value is "<<slow->next->value<<endl;  if(fast->next != nullptr)  {  cout<<"the list has odd num of node"<<endl;  slow =slow->next;  }  int cur\_value;  while(!list\_value.empty())  {  cur\_value = list\_value.top();  cout<<"stack top value is"<<cur\_value<<endl;  cout<<"list value is "<<slow->next->value<<endl;  if(cur\_value != slow->next->value)  {  return false;  }  list\_value.pop();  slow = slow->next;  }  return true;  } |

**● 请你手写链表反转**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | struct ListNode {  int val;  struct ListNode \*next;  ListNode(int x) :val(x), next(NULL) {}  }  ListNode\* ReverseList(ListNode\* pHead)  {  if(!pHead||!pHead->next)return pHead;  ListNode \*pre=nullptr;  ListNode \*p=pHead;  ListNode \*next=pHead->next;  while(p)  {  p->next=pre;  pre=p;  p=next;  if(next)  next=next->next;  }  return pre;  } |

**● 请问什么是单向链表，如何判断两个单向链表是否相交**

**参考回答：**

考察点：数据结构，算法

公司：百度

1、单向链表

单向链表（单链表）是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的；其中每个结点都有指针成员变量指向列表中的下一个结点。

列表是由结点构成，head指针指向第一个成为表头结点，而终止于最后一个指向nuLL的指针。



2、判断两个链表是否相交

1）方法1：

链表相交之后，后面的部分节点全部共用，可以用2个指针分别从这两个链表头部走到尾部，最后判断尾部指针的地址信息是否一样，若一样则代表链表相交！

2）方法2：

可以把其中一个链表的所有节点地址信息存到数组中，然后把另一个链表的每一个节点地址信息遍历数组，若相等，则跳出循环，说明链表相交。进一步优化则是进行hash排序，建立hash表。

高级算法

**● 请问加密方法都有哪些**

**参考回答：**

考察点：密码学

公司：腾讯

1、单向加密

单向加密又称为不可逆加密算法，其密钥是由加密散列函数生成的。单向散列函数一般用于产生消息摘要，密钥加密等，常见的有：

MD5（Message Digest Algorithm 5）：是RSA数据安全公司开发的一种单向散列算法，非可逆，相同的明文产生相同的密文；

SHA（Secure Hash Algorithm）：可以对任意长度的数据运算生成一个160位的数值。其变种由SHA192，SHA256，SHA384等；

CRC-32，主要用于提供校验功能；

算法特征：

输入一样，输出必然相同；

雪崩效应，输入的微小改变，将会引起结果的巨大变化；

定长输出，无论原始数据多大，结果大小都是相同的；

不可逆，无法根据特征码还原原来的数据；

2、对称加密

采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。

特点：

1、加密方和解密方使用同一个密钥；

2、加密解密的速度比较快，适合数据比较长时的使用；

3、密钥传输的过程不安全，且容易被破解，密钥管理也比较麻烦；

优点：对称加密算法的优点是算法公开、计算量小、加密速度快、加密效率高。

缺点：对称加密算法的缺点是在数据传送前，发送方和接收方必须商定好秘钥，然后使双方都能保存好秘钥。其次如果一方的秘钥被泄露，那么加密信息也就不安全了。另外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一秘钥，这会使得收、发双方所拥有的钥匙数量巨大，密钥管理成为双方的负担。

3、非对称加密

非对称密钥加密也称为公钥加密，由一对公钥和私钥组成。公钥是从私钥提取出来的。可以用公钥加密，再用私钥解密，这种情形一般用于公钥加密，当然也可以用私钥加密，用公钥解密。常用于数字签名，因此非对称加密的主要功能就是加密和数字签名。

特征：

1）秘钥对，公钥(public key)和私钥(secret key)

2）主要功能：加密和签名

发送方用对方的公钥加密，可以保证数据的机密性（公钥加密）。

发送方用自己的私钥加密，可以实现身份验证（数字签名）。

3）公钥加密算法很少用来加密数据，速度太慢，通常用来实现身份验证。

常用的非对称加密算法

RSA：由 RSA公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的；既可以实现加密，又可以实现签名。

DSA（Digital Signature Algorithm）：数字签名算法，是一种标准的 DSS（数字签名标准）。

ECC（Elliptic Curves Cryptography）：椭圆曲线密码编码。

**● 什么是LRU缓存**

**参考回答：**

LRU(最近最少使用)算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高

实现：使用一个链表保存缓存数据，将新数据插入到头部，每当缓存命中时，则将命中的数据移动到链表头部，当链表满的时候，将链表尾部的数据丢弃。

**● 请你说一说洗牌算法**

**参考回答：**

考察点：

公司：腾讯

1、Fisher-Yates Shuffle算法

最早提出这个洗牌方法的是 Ronald A. Fisher 和 Frank Yates，即 Fisher–Yates Shuffle，其基本思想就是从原始数组中随机取一个之前没取过的数字到新的数组中，具体如下：

1）初始化原始数组和新数组，原始数组长度为n(已知)。

2）从还没处理的数组（假如还剩k个）中，随机产生一个[0, k)之间的数字p（假设数组从0开始）。

3）从剩下的k个数中把第p个数取出。

4）重复步骤2和3直到数字全部取完。

5）从步骤3取出的数字序列便是一个打乱了的数列。

时间复杂度为O(n\*n)，空间复杂度为O(n)。

2）Knuth-Durstenfeld Shuffle

Knuth 和 Durstenfeld  在Fisher 等人的基础上对算法进行了改进，在原始数组上对数字进行交互，省去了额外O(n)的空间。该算法的基本思想和 Fisher 类似，每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字。

算法步骤为：

1. 建立一个数组大小为 n 的数组 arr，分别存放 1 到 n 的数值；

2. 生成一个从 0 到 n - 1 的随机数 x；

3. 输出 arr 下标为 x 的数值，即为第一个随机数；

4. 将 arr 的尾元素和下标为 x 的元素互换；

5. 同2，生成一个从 0 到 n - 2 的随机数 x；

6. 输出 arr 下标为 x 的数值，为第二个随机数；

7. 将 arr 的倒数第二个元素和下标为 x 的元素互换；

……

如上，直到输出m 个数为止

时间复杂度为O(n)，空间复杂度为O(1)，缺点必须知道数组长度n。

字符串

**● 给你一个字符串，找出第一个不重复的字符，如“abbbabcd”，则第一个不重复就是c**

**参考回答：**

使用哈希的思想，建立256个bool数组array，初始都为false,从头开始扫描字符串，扫到一个，将以其ascii码为下标的元素置true。例如扫描到A的时候，执行：array['A']=true。第二边扫描，扫到一个字母就以其ascii码为下标，去array数组中看其值，如果是true,返回改字母，如果是false，继续扫描下一个字母。