# 面试题汇总

**基本的数据结构**：数组，链表，树，图，栈，队列，哈希表，字符串

**基本的算法：**

1. 几种排序算法的思想，快排，堆排和归并排序的代码要会写
2. 二分查找及其变形
3. BFS, DFS, DP, 贪心

## C/C++语言基础

### 一个C++程序从编译到运行都经历了哪些阶段？

答：预编译：宏替换，注释消除，找到相关的库文件。

编译: 在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或者汇编代码。

汇编：汇编过程是把汇编语言代码翻译成目标机器指令的过程。

链接：比如某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或函数调用）；在程序中可能调用了某个库文件中的函数等等。所有的问题都需要经过链接程序的处理才能得以解决。

g++执行的四个步骤：

1. 预处理：条件编译，头文件包含，宏替换的处理，生成.i文件。
2. 编译：将预处理之后的文件转换为汇编语言，生成.s文件
3. 汇编：汇编为目标代码(机器代码)，生成.o文件
4. 链接：链接目标代码，生成可执行程序。

### 动态链接与静态链接的区别与联系？

答：（1）静态链接：在生成可执行文件的时候(链接阶段)，把所有需要的函数的二进制代码都包含到可执行文件中去，因此，链接器需要知道参与链接的目标文件需要哪些函数，同时也要知道每个目标文件能够提供什么函数，这样链接器才能知道是不是每个目标文件所需要的函数都能正确地链接。如果某个目标文件需要的函数在参与链接的目标文件中找不到的话，链接器就报错了。目标文件中有两个重要的接口来提供这些信息：一个是符号表，另外一个是重定位表。在程序发布的时候不需要依赖库，也就是不需要带着库一起发布，程序可以独立执行。

缺点是1.程序体积相对较大 2.如果静态库有更新的话，所有的可执行文件都得重新链接才能连上新的静态库。

（2）动态链接：在编译的时候不直接拷贝可执行代码，而是通过记录一系列符号和参数，在程序运行或者加载时将这些信息传递给操作系统，操作系统负责将所需要的动态库加载到内存中，然后程序在执行到指定的代码时，去共享执行内存中已经加载的动态库可执行代码，最终达到运行时链接的目的。

优点是：多个程序可以共享一段代码，而不需要在磁盘上存储多个拷贝

缺点是由于是运行时加载，可能会影响程序前期的执行性能。

什么是库：库就是一些功能代码经过编译链接之后的可执行形式，windows下的dll和linux下的so都是库。

（3）1.静态库将导出声明和实现都放在库中，编译之后所有代码都嵌入到宿主程序。

2.动态库相当于一个h文件，是对实现部分的导出部分的声明。编译后只是将导出声明部分编译到宿主程序中，运行时需要相应的库支持。

简单来说，**静态链接**是指在链接阶段直接把静态库加入到可执行文件中去，这样可执行文件会比较大。

而**动态链接**则是指链接阶段仅仅只加入一些描述信息，而程序执行时再从系统中把相应动态库加载到内存中去。代码共享和延迟加载。

### C++的多态是如何实现的？虚函数是怎么发挥作用的？

答：（1）静态多态就是在系统编译期间就可以确定程序执行到这里将要执行哪个函数，例如函数重载，对象名加点操作符执行成员函数等称为静态多态，其中重载是在形成符号表的时候，对函数名做了区分，从而确定了函数执行到这里将要执行哪个函数，对象名加点操作符执行成员函数是通过this指针来调用的。

函数重载就是利用参数的不同确定调用关系的，而函数模板技术利用类型的不同生成不同的执行代码。都体现了静态多态性。

（2）动态多态就是利用虚函数实现运行时的多态，也就是说在系统编译的时候并不知道将要调用哪个函数，只有在运行到这里的时候才能确定接下来会跳转到哪一个函数的栈帧。

虚函数是在基类中声明了该函数是虚拟的，在函数前面加上一个virtual关键字，然后在子类中正式定义（子类的函数名，返回值，函数参数个数，参数类型等全部要与基类所声明的虚函数完全相同，此时才是重写，否则是重载），再定义一个指向基类对象的指针，然后将该指针指向由基类派生的子类对象，然后用这个指针来调用改写的虚函数，就能实现动态多态。

如果一个类定义了虚函数，该类和它的派生类就要生成一张虚函数表，存放了该类所有虚函数对应的函数指针，包括重写了基类虚函数的派生类虚函数和自己本身新增的虚函数。

派生类虚函数表的构造过程：

1. 拷贝基类的虚函数表；2.替换已经重写的虚函数指针；3.追加子类自己的虚函数指针

### 虚函数表在内存中存放的位置？

答：虚函数表vtable在linux/unix中存放在可执行文件的只读数据段中(rodata)。虚函数表一旦产生就具有不变性，所以编译器就经常把它放在稳定或者说是只读的内存区。

### 指针和引用的区别？

答：相同点是两者都是地址的概念；

不同点：（1）指针式一个实体（替身），引用只是个别名（本体的另一个名字）。

（2）引用只能在定义时被初始化一次，之后不可以改变，而指针可以修改。

（3）sizeof(引用)：所引用变量的大小；sizeof（指针）：指针变量本身的大小；

（4）指针自增是指变量地址自增，引用所指变量本身的值的自增。

（5）引用是类型安全的，引用会类型检查，指针不会进行类型检查。

（6）引用不能为空，但是指针可以为空；

（7）从内存分配上看，程序为指针变量分配内存区域，而引用不需要内存分配区域。

（8）引用没有 const，指针有 const；

### 什么时候将析构函数定义为虚函数，为什么？

答：析构函数经常被定义为虚函数，当父类指针指向派生类对象时，如果删除该指针，就会调用该指针指向的派生类的析构函数，而派生类的析构函数又会自动调用基类的析构函数，这样保证对派生类对象进行有必要的清理工作。

### 纯虚函数的作用是什么？

答：在基类不对虚函数进行有意义的实现，只是在派生类中有具体的意义。具有纯虚函数的类都是抽象类，抽象类不能定义对象，但是可以定义指向它的指针，在其派生类中重写该纯虚函数，否则会产生程序运行错误。抽象类的唯一的用途是为派生类提供基类。

### 在有派生类的情况下，构造函数和析构函数的调用顺序？

答：构造函数不能被继承，派生类的构造函数必须调用基类的构造函数来初始化基类成员的子对象。调用顺序：（1）基类构造函数；（2）子对象类的构造函数（基类子对象）；（3）派生类的构造函数；按照继承时说明的基类顺序调用基类的构造函数，再调用派生类的构造函数，调用析构函数的顺序刚好相反。

注意：基类成员构造用基类名，基类子对象构造用对象名。

### new/delete 和 malloc/free的区别？

答：（1）new/delete是c++的关键字，需要编译器支持，malloc/free是库函数，需要头文件支持。（2）new操作符申请内存分配时无需指定内存块的大小，编译器会根据类型信息自行计算，而malloc需要显示地之处所需内存的大小。

（3）new操作符分配内存成功时，返回对象类型指针，类型严格与对象匹配，无法进行类型转换。malloc 分配成功时返回void \*，需要强制类型转换将void \*指针转换为所需要的类型。（4）new分配失败时抛出bac\_alloc异常，malloc内存分配失败时返回NULL。

（5）new/delete可以调用构造函数/析构函数，对自定义类型对象做构造/析构；

（6）new/delete运算符可以重载，malloc不可以重载。

### 虚基类的作用？

答：虚基类的作用：如果一个派生类有多个直接基类，而这些直接基类又有一个共同的基类，则在最终的派生类中会保留该间接共同基类数据成员的多份同名成员。在引用这些同名的成员时，必须在派生类对象名之后增加直接基类名，以避免产生二义性，使其唯一地标识一个成员。比如c.A::display()。

我们不希望在一个类中保留间接基类的多份同名成员，这种现象是人们不希望看到的。因此引入了虚基类这个玩意，使得在继承间接共同基类时只保留一份成员。虚基类并不是在声明基类的时候声明的，而是在声明派生类时候指定继承方式时声明的。

1. **class**
2. A//定义基类A
3. {
4. A(**int** i){ } //基类构造函数，有一个参数};
5. **class** B :**virtual** **public** A
6. //A作为B的虚基类
7. {
8. B(**int** n):A(n){ } //B类构造函数，在初始化表中对虚基类初始化
9. };
10. **class** C
11. :**virtual** **public** A //A作为C的虚基类
12. {
13. C(**int** n):A(n){ }
14. //C类构造函数，在初始化表中对虚基类初始化
15. };
16. **class** D :**public** B,**public** C
17. //类D的构造函数，在初始化表中对所有基类初始化
18. {
19. D(**int** n):A(n),B(n),C(n){ }
20. };

在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。C++编译系统只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类B和类C)

### 友元函数的作用和实现？

答：友元函数尽量少用!

友元函数是指某些虽然不是类成员却能够访问类的所有成员的函数。类授予它的友元特别的访问权。1）必须在类的说明中说明友元函数，说明时以关键字friend开头，后跟友元函数的函数原型，友元函数的说明可以出现在类的任何地方，包括在private和public部分；

2）注意友元函数不是类的成员函数，所以友元函数的实现和普通函数一样，在实现时不用"::"指示属于哪个类，只有成员函数才使用"::"作用域符号；

3）友元函数不能直接访问类的成员，只能访问对象成员，

4）友元函数可以访问对象的私有成员，但普通函数不行；

5）调用友元函数时，在实际参数中需要指出要访问的对象，

6）类与类之间的友元关系不能继承。

7）一个类的成员函数也可以作为另一个类的友元，但必须先定义这个类。

### extern和extern ‘C’的作用？

答：（1）extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字：它告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。

（2）对于extern变量来说，仅仅是一个变量的声明，并不是在定义分配内存空间。如果该变量定义多次，会有连接错误。

（3）通常，在模块的头文件中对本模块提供给其他模块引用的函数和全局变量以关键字extern声明。也就是说c文件里面定义，如果该函数或者变量与开放给外面，则在h文件中用extern加以声明。所以外部文件只用include该h 文件就可以了。而且编译阶段，外面是找不到该函数的，但是不报错。link阶段会从定义模块生成的目标代码中找到此函数。

（4）与extern对应的关键字是static，被它修饰的全局变量和函数只能在本模块中使用。

extern ‘C’:

在C++环境下使用C函数的时候，常常会出现编译器无法找到obj模块中的C函数定义，从而导致链接失败的情况，应该如何解决这种情况呢？

（1）C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C语言则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。

（2）extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言的进行编译，而不是C++的。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般之包括函数名。

### 说一说RTTI的实现？

答：运行时类型识别，C++引入这个机制是为了让程序在运行时能根据基类指针或者引用来获得该指针或者引用所指的对象的实际类型。但是现在RTTI的类型识别已经不限于此了，它还能通过typeid操作符识别出所有的基本类型（int，指针等）的变量对应的类型。

C++通过以下两个操作符莱提供RTTI:

1. typeid运算符，该运算符返回其表达式或者类型名的实际类型。
2. dynamic\_cast运算符，该运算符将基类的指针或引用安全地转换为派生类类型的指针或引用。

### 结构体和类的内存对齐原则，求出一个结构体或者类的大小？

答：结构体和类存在内存对齐和补齐原则(按照数据类型定义的顺序一个一个地放到内存中去的)；

对齐原则是指：从变量大小的非负整数倍处开始存储变量。

补齐原则是指：计算并检查存储单元是否为所有元素中占据内存最大元素长度的整数倍，如果是，则结束，如果不是，将其补齐成为整数倍。

（1）类的大小与普通数据成员有关，与成员函数和静态成员无关，普通成员函数，静态成员函数，静态数据成员，静态常量数据成员均对类的大小无影响。因为类的成员函数是类的所有实例共享的，调用时通过隐藏的this指针和类的实例相关联，函数代码编译之后根本不在类的实例中，所以不占实例空间。静态成员之所以不在类的对象大小内，是因为类的静态成员被类的所有对象所共享，并不属于具体的哪个对象，静态数据成员定义在内存的全局区。

（2）空类的大小是1；虚函数对类的大小有影响，是因为虚函数表指针带来的影响。虚继承对类的大小有影响，是因为虚基类指针带来的影响。

（3）当派生类继承空类后，派生类如果有自己的数据成员，而空基类的一个字节并不会加到派生类中去。第二种情况是，一个类包含一个空类对象数据成员，这时候空类的一个字节会被算进去的。

1. #include<iostream>
2. **using** **namespace** std;
3. **class** A
4. {
5. };
6. **class** B
7. {
8. **char** ch;
9. **virtual** **void** func0()  {  }
10. };
12. **class** C
13. {
14. **char** ch1;
15. **char** ch2;
16. **virtual** **void** func()  {  }
17. **virtual** **void** func1()  {  }
18. };
20. **class** D: **public** A, **public** C
21. {
22. **int** d;
23. **virtual** **void** func()  {  }
24. **virtual** **void** func1()  {  }
25. };
26. **class** E: **public** B, **public** C
27. {
28. **int** e;
29. **virtual** **void** func0()  {  }
30. **virtual** **void** func1()  {  }
31. };
33. **int** main(**void**)
34. {
35. cout<<"A="<<**sizeof**(A)<<endl;    //result=1
36. cout<<"B="<<**sizeof**(B)<<endl;    //result=16
37. cout<<"C="<<**sizeof**(C)<<endl;    //result=16
38. cout<<"D="<<**sizeof**(D)<<endl;    //result=16
39. cout<<"E="<<**sizeof**(E)<<endl;    //result=32
40. **return** 0;
41. }

以上结果是在64位系统中的结果，A是空类，其大小是1个字节，B的大小是由char和一个虚函数指针组成的，虚函数指针的大小是8个字节，而char补齐之后就是16个字节。对于C，两个char类型，有虚函数，就有个虚函数指针，补齐之后也是16个字节。对于D，公有继承了B和C，里面包含了一个整型变量和两个虚函数，这俩虚函数只覆盖了C的对应虚函数，因此只有一个虚函数表，还有一个整型变量，补齐之后就是16个字节。对于E，公有继承了B和C两个类，分别覆盖了B和C的两个虚函数，因此有两个虚函数表指针，总共是8+8+8\*2=32个字节。

虚函数以及各个成员在内存中存放的位置如下图所示

类E的虚函数表以及成员的内存分配：

|  |  |  |
| --- | --- | --- |
| B vptr |  | E : func0( ) |
| B: char ch |  |
| C vptr | C : func( ) | E : func1( ) |
| C: char ch1 |  |
| C: char ch2 |  |
| E: int e |  |

类的总大小是32字节

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B vptr | B ch |  | 16 C:vptr 23 | C ch1 | C ch2 |  | 28 E: int e 31 | 32 |

### 共同体的内存分布和大小计算？

答: 共同体内的元素公用内存空间，从偏移量为0开始，其大小为所有数据类型中最大的那个数据类型的宽度，不同数据类型公用同一存储区域，所有成员在共同体变量中存放在同一地址，相互覆盖。

### C++容器有哪些？底层分别是如何实现的？

答：STL的六大组件

（1）容器（container)

list vector deque set map string

（2）迭代器(iterator)

如同一个指针

（3）算法(algorithm)

用来操作容器中的数据的模板函数，例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用；

（4）仿函数(function object)

（5）迭代适配器(adapter)

queue priority\_queue stack

（6）空间配置器(allocator)

1.vector底层数据结构是数组，支持快速的随机访问

2.list的底层数据结构是双向链表，支持快速增删

3.deque 底层数据结构是一个中央控制器和多个缓冲区，支持首尾快速增删，也支持随机访问。

4.stack 的底层一般用list或者deque实现，封闭头部即可，不用vector的原因是容量大小有限制，扩容耗时。

5.queue底层一般用list或者deque实现，封闭头部就好，不用vector的原因也是容量大小有限制，扩容耗时。

6.priority\_queue的底层数据结构一般是vector，堆heap是处理规则来管理底层容器的实现。

7.set底层数据结构是红黑树，有序不重复

8.multiset底层数据结构是红黑树，有序，可以重复

9.map的底层数据结构也是红黑树，有序不重复

10.multimap底层数据结构是红黑树，有序可重复

11.hash\_set的底层数据结构是hash表，无序不重复

12.hash\_multiset 底层数据结构是hash表，无序，可重复

13.hash\_map底层数据结构是hash表，无序，不重复

14.hash\_multimap的底层数据结构是hash表，无序，可重复。

### Vector的size和capacity的区别？

答：size是当前vector容器真实占用的大小，也就是容器当前拥有多少个容器。

capacity是指在发生realloc前能允许的最大元素数，即预分配的内存空间。

### Vector的push\_back方法是如何实现的？

答：push\_back的时候如果容量不够，会再增加50%的空间，重新分配内存，复制，释放原来的存储区域。reserve是预留存储空间的，

### Vector和map的erase方法实现有什么区别？

答：（1）vector循环删除的时候，erase(it)会返回下一个迭代器的地址，保险的做法是赋值给it 即 it= erase(it)，这是vector的内部机制所造成的，所以对vector进行erase的时候特别注意迭代器是否会失效！

（2）map则可以直接erase(it++)，vector和map都不能将it++写在for循环中，而在循环体内erase(it)!

1. **int** main()
2. {
3. vector<**int**> v;
4. v.push\_back(1);
5. v.push\_back(2);
6. v.push\_back(4);
7. v.push\_back(3);
8. v.push\_back(6);
9. v.push\_back(5);
10. cout << v.size() <<endl;
11. vector<**int**>::iterator it;
12. **for**(it = v.begin();it != v.end();)
13. {
14. **if**(\*it % 2 == 0)
15. //v.erase(it++);
16. //it = v.erase(it);
17. v.erase(it);
18. **else**
19. it++;
20. }
21. cout << v.size() <<endl;
22. **for**(it = v.begin();it != v.end();it++)
23. {
24. cout << \*it << " ";
25. }
26. // map的完美删除
27. map<**int**, **int**> m;
28. m[1] = 1;
29. m[2] = 2;
30. m[3] = 4;
31. m[4] = 3;
32. m[5] = 5;
33. m[6] = 6;
34. cout <<"m size = "<<m.size() <<endl;
35. map<**int**, **int**>::iterator it1;
36. **for**(it1 = m.begin(); it1!=m.end();)
37. {
38. **if**(it1->second % 2 == 0)
39. m.erase(it1++);
40. **else**
41. it1++;
42. }
43. cout <<"2的整数倍删除后应该剩下i3";
44. cout <<"m size = "<<m.size() <<endl;
46. }

### 为什么C语言中没有重载而C++中有？

答：比如一个函数的声明如下：

void function(int x,int y)；

在c语言中，编译器在编译后在库中的名字为\_function

在c++中，编译器在编译后在库中的名字为\_function\_int\_int

### static关键字作用？

1. static一般有2个作用,规定作用域和存储方式.
2. 对于局部变量,static规定其为静态存储方式,每次调用的初始值为上一次调用的值。调用结束后存储空间不释放;
3. 对于全局变量,如果以文件划分作用域的话,此变量只在当前文件可见; 对于static函数也是在当前模块内函数可见。
4. static,规定此全局变量只在当前模块(文件)中可见，static const,既是只读的,又是只在当前模块中可见的.
5. 成员变量：用static修饰类的数据成员实际是的其成为类的全局变量，会被类的所有对象共享，包括派生类的对象。static成员必须在类外进行初始化，不能在构造函数内进行初始化，不过也可以用const修饰static数据成员在类内进行初始化。
6. 成员函数：用static修饰成员函数，使这个类只存在这一份函数，所有对象共享该函数，不含this指针。
7. 静态成员是可以独立访问的，也就是说不需要创建任何的对象实例就可以访问，当static成员函数在类外定义时不需要加static修饰符。
8. 在静态成员函数的实现过程中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员，因为静态成员函数不含this指针。
9. 此时一定要注意：不可以同时用const和static修饰成员函数。c++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this\*，但是当一个成员是static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。static是为了说明该函数是只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态。

### const关键字的作用

1. const的作用是限定变量为不可修改，或者限定成员函数不可以修改任何数据成员。
2. const与指针：const char \*p表示指向的内容不能改变。
3. char \* const p, 就是将p声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

### C++的内存分配是什么机制？具体有哪些？怎么分配的？

答：首先，来谈谈C++的5个存储区：

1. 栈：是分配给函数局部变量的存储单元，函数结束后，该变量的存储单元自动释放，效率高，分配的空间有限。

2. 堆：由new创建，由delete释放的动态内存单元。如果用户不释放该内存，程序结束时，系统会自动回收。

3. 自由存储区：由malloc创建，由free释放的动态内存单元，与堆类似。

4. 全局（静态）存储去：全局变量和静态变量占一块内存空间。

5. 常量存储区：存储常量，内容不允许更改。

### 如何解决头文件重复包含的问题？

答：

（1）最常用的方法是用条件编译，我们在类定义的前后分别加上了#ifndef…#define…#endif语句。就保证了头文件在给定的源文件中只被处理了一次，首先检测#ifndef。第一次处理头文件时，测试会成功，因为\_SOME\_FILE\_H\_还未定义，下一条语句定义了\_SOME\_FILE\_H\_，那样的话，如果编译的文件恰好又一次包含了该头文件，#ifndef指示会发现\_SOME\_FILE\_H\_已经定义，并且忽略该头文件的剩余部分。

为了保证宏的唯一性，可以采用google提供的解决方案，在定义宏时，宏名基于其所在的项目源代码数的全路径命名，宏命名格式为：\_<PROJECT><PATH>\_<FILE>\_H\_

（2）#pragma once，这种方式一般由编译器提供，只要在头文件的最开始加入这条指令就能够保证头文件只被编译一次。#pragma once用来防止某个头文件被多次include,#ifndef方式用来防止某个宏被多次定义。#pragma once是编译相关，即在这个编译系统上使用，但在其它编译系统上不一定能用，也就是说移植性差，不过现在基本上已经是每个编译器都有这个定义了。针对#pragma once，GCC已经取消对其的支持了，而微软的VC++却依然支持如果写的程序需要跨平台，最好使用#ifndef方式，而避免使用#pragma once方式。

### C++的string类如何实现？

答：string.h文件

1. #ifndef \_STRING\_H\_
2. #define \_STRING\_H\_
3. #include <iostream>
4. **using** **namespace** std;
6. **class** String
7. {
8. **public**:
9. String();  //默认构造函数
10. String(**char** \*str);  //构造函数
11. String(**const** String& other);  //拷贝构造函数
12. ~String();  //析构函数
13. String& operator=(**char** \*str); //赋值运算符重载
14. String& operator=(**const** String& other);  //赋值运算符重载
15. **char**& operator[](unsigned **int** index);  //下标运算符重载
16. **const** **char**& operator[](unsigned **int** index) **const**;
17. **friend** String operator+(**const** String& s1, **const** String& s2);  //字符串连接
18. String& operator+=(**const** String& s);
19. **friend** ostream& operator<<(ostream& out, **const** String& s);
20. **friend** istream& operator>>(istream& in, String& s);
21. **void** Display();
22. **private**:
23. **char** \*str\_;
24. };
25. #endif

String.cpp文件：

1. #include <iostream>
2. #include <string.h>
3. #include "String.h"
4. **using** **namespace** std;
5. String::String()
6. {
7. str\_ = **new** **char**['\0'];
8. cout << "default construct String!" << endl;
9. }
10. String::String(**char** \*str)
11. {
12. cout << "construct String!" << endl;
13. **int** len = strlen(str) + 1;
14. str\_ = **new** **char**[len];
15. memset(str\_, 0, len);
16. strcpy(str\_, str)
17. }
18. String::String(**const** String& other)
19. {
20. **int** len = strlen(other.str\_) + 1;
21. str\_ = **new** **char**[len];
22. memset(str\_, 0, len);
23. strcpy(str\_, other.str\_);
24. }
25. String& String::operator=(**const** String& other)
26. {
27. **if**(**this** == &other)
28. {
29. **return** \***this**;
30. }
32. **int** len = strlen(other.str\_) + 1;
33. **delete** [] str\_;
34. str\_ = **new** **char**[len];
35. memset(str\_, 0, len);
36. strcpy(str\_, other.str\_);
37. **return** \***this**;
38. }
39. String& String::operator=(**char** \*str)
40. {
41. **delete** [] str\_;
42. **int** len = strlen(str) + 1;
43. str\_ = **new** **char**[len];
44. memset(str\_, 0, len);
45. strcpy(str\_, str);
46. **return** \***this**;
47. }
48. **char**& String::operator[](unsigned **int** index)
49. {
50. //return str\_[index];
51. **return** **const\_cast**<**char**&>(**static\_cast**<**const** String&>(\***this**)[index]);
52. }
53. **const** **char**& String::operator[](unsigned **int** index) **const**
54. {
55. **return** str\_[index];
56. }
57. String operator+(**const** String& s1, **const** String& s2)
58. {
59. #if 0
60. **int** len = strlen(s1.str\_) + strlen(s2.str\_) + 1;
61. **char** \*newptr = **new** **char**[len];
62. memset(newptr, 0, len);
63. strcpy(newptr, s1.str\_);
64. strcat(newptr, s2.str\_);
65. String tmp(newptr);
66. #endif
67. String tmp(s1);
68. tmp += s2;
69. **return** tmp;
70. }
71. String& String::operator+=(**const** String& s)
72. {
73. **int** len = strlen(s.str\_) + strlen(str\_) + 1;
74. **char** \*newptr = **new** **char**[len];
75. memset(newptr, 0, len);
76. strcpy(newptr, str\_);
77. strcat(newptr, s.str\_);
78. String tmp(newptr);
79. **delete** [] str\_;
80. str\_ = **new** **char**[len];
81. strcpy(str\_, newptr);
82. **return** \***this**;
83. }
84. ostream& operator<<(ostream& out, **const** String& s)
85. {
86. out << s.str\_;
87. **return** out;
88. }
89. istream& operator>>(istream& in, String& s)
90. {
91. **char** buffer[4096];
92. in >> buffer;
93. s.str\_ = buffer;
94. **return** in;
95. }
96. String::~String()
97. {
98. cout << "destroy String!" << endl;
99. }
100. **void** String::Display()
101. {
102. cout << "str = " << str\_ << endl;
103. }

### 如何避免拷贝构造函数和赋值运算符被随意调用？

### static\_cast和dynamic\_cast的区别？

答：层次类型转换的两个关键字，前者是支持编译时类型检查，dynamic\_cast支持运行时类型检查，对于下行转换时安全的，也就是将父类指针或者引用转换为派生类指针或者引用，一般来讲，父类指针指向子类对象，但是该指针只能访问父类定义的数据成员和函数。

### 智能指针的作用？

答：智能指针实际上是借鉴了java的内存回收机制（引用计数），智能指针将一个计数器与类指向的对象相关联，引用计数跟踪共有多少个类对象共享同一个指针。每次创建类的新对象的时候，初始化指针并将引用计数置为1。当对象作为另一个对象的副本创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数。对一个对象进行赋值的时候，赋值操作符减少左操作数所指对象的引用计数（如果引用计数减为0，则删除对象），并且增加右操作数所指对象的引用计数，这是因为左侧的指针指向了右侧指针所指向的对象，因此右指针所指向的对象的引用计数+1。调用析构函数时，构造函数减少引用计数（如果引用计数减为0，则删除基础对象）。实现智能指针有两种经典策略，一是引入辅助类，二是使用句柄类。

一个由c++编译的程序占用的内存分为以下几个部分：

栈区（stack）—— 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区（heap）—— 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

全局区（静态区）（static）——全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

文字常量区——常量字符串就是放在这里的，程序结束后由系统释放 。

程序代码区——存放函数体的二进制代码。

//main.cpp

int a = 0; 全局初始化区

char \*p1; 全局未初始化区

main() {

int b; 栈

char s[] = "abc"; 栈

char \*p2; 栈

char \*p3 = "123456"; 123456/0在常量区，p3在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char \*)malloc(10);

p2 = (char \*)malloc(20);

分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456"); 123456/0放在常量区，编译器可能会将它与p3所指向的"123456" 优化成一个地方。

}

分配在堆上的内存需要手动进行动态分配和释放，我们称之为动态内存，C++中，动态内存通过new和delete进行分配和释放的。

new：在动态内存中为对象分配空间并返回一个指向该对象的指针，我们可以选择对对象进行初始化。

delete：接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。

在自由空间分配的内存是无名的，因此new无法为其分配的对象命名，而是返回一个指向该对象的指针。

智能指针与常规指针的最大区别在于，智能指针负责主动释放所指向的对象，两种智能指针的区别在于管理底层指针的方式：

Shared\_ptr允许多个指针指向同一个对象，unigue\_ptr则独占所指向的对象。。标准库还定义了一个名为week\_ptr的伴随类，它是一种弱引用，指向shared\_ptr所管理的对象。这三种类型都定义在memory的头文件中。

Make\_shared函数，是最安全的分配和使用动态内存的方法，此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr。

shared\_ptr的拷贝和赋值

当进行拷贝或赋值操作时，每个shared\_ptr都会记录有多少个其他shared\_ptr指向相同的对象。

shared\_ptr自动销毁所管理的对象

shared\_ptr的析构函数会递减它所指向的对象的引用计数。如果引用计数变为0，shared\_ptr的析构函数就会销毁对象，并释放它占用的内存。

shared\_ptr还会自动释放相关联的内存

return会对shared\_ptr指针的引用次数进行递增操作。

shared\_ptr和new结合使用

接受指针参数的智能指针构造函数是explicit的，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式来初始化一个智能指针：

1. shared\_ptr<**int**> p1=**new** **int**(1024);       //错误
2. shared\_ptr<**int**> p2(**new** **int**(1024));      //正确
3. shared\_ptr<**int**> clone(**int** p){
4. **return** **new** **int**(p);                  //错误
5. }
6. shared\_ptr<**int**> clone(**int** p){
7. **return** shared\_ptr<**int**>(**new** **int**(p)); //正确
8. }

### STL如何进行内存管理？

答：容器的内存管理工作是通过STL提供的一个默认的allocator实现的，这个allocator是一个由两级分配器构成的内存管理器，当申请的内存大小大于128个字节时，就启动第一级分配器，通过malloc直接向系统的堆空间分配。如果申请的内存大小小于128个字节，就启动第二级分配器，从一个预先分配好的内存池中取出一块内存交付给用户，这个内存池由16个大小不同(8的倍数，8-128byte)的空闲列表组成，allocator会根据申请内存的大小（将这个大小round up成8的倍数），从对应的空闲列表中取表头块给用户。

### 用dynamic\_cast如何知道是调用的子类还是父类的虚函数？

答：虚函数对于dynamic\_cast的转换作用，也就是为什么使用dynamic\_cast转换类指针时，需要虚函数呢。

Dynamic\_cast转换是在运行时进行转换，运行时转换就需要知道类对象的信息（继承关系等）。如何在运行时获取到这个信息——虚函数表。C++对象模型中，对象实例最前面的就是虚函数表指针，通过这个指针可以获取到该类对象的所有虚函数，包括父类的。因为派生类会继承基类的虚函数表，所以通过这个虚函数表，我们就可以知道该类对象的父类，在转换的时候就可以用来判断对象有无继承关系。所以虚函数对于正确的基类指针转换为子类指针是非常重要的。

### 在c语言中fork()遇上printf函数会出现什么情况？

答：

### namespace 的作用

答：std命名空间是C+标准库类型对象的命名空间，如果加上using namespace std这句，程序就把文件内出现的用std命名的关键字都自动在前面加上std::。std命名空间是C++中标准库类型对象的命名空间。

namespace命名空间是在大型项目开发中，为了避免命名冲突而引入的一种机制。命名空间实际上是一个作用域。

### volatile的作用

答：volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如，操作系统，硬件或者其它线程等。由于访问寄存器的速度要快于RAM，所以编译器一般都会作减少存取外部RAM的优化。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

这个词的意思是易变，编译器在编译时期可能不能获取变量是否被多个线程或者进程修改的信息。这个时候一个变量是否在两次读操作之间发生改变，编译器肯定无法确定。然而编译优化技术针对一般的变量都会做出优化。

比如，

int a=0;

int b=a;

int c=a+1;

编译器极有可能把a放到寄存器中，供b,c的计算使用。更有甚者，编译器确定了a的值是0, 会直接计算出b=0,c=1；如果在实际运行中a的值被其他线程修改了，这么做就改变了代码的语义。

Volatile int a;//这里对a是否初始化已经不再重要了。为了消除这种问题，使用volatile关键字告诉编译器每次访问a的时候都需要读取内存，而不对其进行优化。

### explicit的作用

答：显式的，它和C++的隐式转换相关，禁止隐式类型转换就可以按照程序作者的需要限定构造函数的功能。

### static\_cast、const\_cast、dynamic\_cast、reinterpret\_cast

答：之所以把这四个关键字放在一起，是因为它们处理相似的问题——显式类型转换。C++延续了C风格的强制类型转换的语法。

转换关键字<类型>(表达式)

static\_cast和C风格类型转换功能完全相同，它属于在编译时期静态的类型转换。如果把一个double类型转换为整形，形式如下：

static\_cast<int>(0.1);

static\_cast功能有所限制，比如不能转化struct类型到int，不能转化指针到double等。另外，它不能在转换中消除const和volatile属性。

Const\_cast 用于消除引用或者指针的const或者volatile属性。

比如，const int &ci=100;

int &i=const\_cast<int&>(ci);

通过这种方式，ci引用的内存单元虽然无法通过修改ci改变，但是可以修改i改变内存的值。

Dynamic\_cast 一般出现在类到子类或者兄弟类的转换，并要求基类有虚函数，并且它能够提供转换后的结果和状态，一旦转换失败则返回空的指针。如果没有继承关系的转换一般使用static\_cast。

1. **class** Base
2. {
3. **virtual** **void** fun(){};//必须拥有虚函数
4. };
5. **class** A:**public** Base//必须是供有继承才能默认转换
6. {
7. };
8. Base b;
9. A \*a=**dynamic\_cast**<A\*>(&b);//基类到子类，显式转换
10. Base\*pb=a;//子类到基类，默认转换

### C++中深拷贝和浅拷贝的区别?

答：这个问题的由来是这样的，在C++中，对一个已知对象进行拷贝，编译器会自动调用拷贝构造函数，如果用户没有自己定义拷贝构造函数，则调用默认的拷贝构造函数。当对象成员中使用了动态的数据类型的时候（用new开辟空间），就不能直接相互赋值，否则会在程序执行过程中出现运行错误，主要是因为直接相互赋值的过程是调用了默认构造函数，实现了浅拷贝，两个指针指向了同一块内存，但是析构函数调用了两次，相当于对同一块内存析构两次，从而出现了内存泄漏现象。这个时候就需要重新定义拷贝构造函数，**使得拷贝之后的对象指针成员有自己的内存空间，也就是进行深拷贝**，这样就能避免内存泄漏的发生。

简单来说，浅拷贝只是对指针的拷贝，拷贝之后的指针指向了同一片内存空间，但是深拷贝不但对指针进行了拷贝，而且对指针指向的内容进行拷贝，经过深拷贝之后的指针是指向两个不同地址的指针。

一般来说，需要自定义拷贝构造函数和重载赋值运算符来实现深拷贝。

举例：

在对象中存在指针成员的时候，除了在复制对象时需要考虑自定义拷贝构造函数，还应该考虑以下两种情况：

1. 当函数的参数为对象时，实参传给形参的实际是实参的一个拷贝对象，系统通过拷贝构造函数实现。
2. 当函数的返回值是一个对象的时候，该对象实际上是函数内对象的一个拷贝，用于返回函数调用处。

### typedef和宏的区别?

答：宏处理就是简单替换，相当于word中的replace，发生在预编译阶段，与编译器无关。

Typedef 是C++的一个关键字，它定义了一种类型的别名，而不是简单的宏替换，可以用作同时声明指针型的多个对象。因为typedef是定义了一种类型的新别名，不是简单的字符串替换，所以它比宏来得稳健。

两者的不同主要体现在两个方面：

（1）可以用其他类型说明符对宏类型名进行扩展，但是对typedef所定义的类型名却不能这么做。

例如：#define peach int

unsigned peach i; //No problem

typedef int banana;

unsigned banana i;//error illegal

（2）在连续几个变量的声明中，用typedef定义的类型能够保证声明中所有的变量均为同一类型，而#define定义的类型则无法保证。

#define int\_ptr int\*

int\_ptr chalk, cheese;

其结果是，chalk 是int\*型，cheese是int型

而typedef char\* PCHAR;

PCHAR pa, pb; pa和pb都是字符型指针。

Typedef的其他用法总结：

1. 在旧的c语言的代码中，帮助struct。以前的代码中，声明struct新对象的时候，必须带上struct，即形式为struct结构名对象名，
2. **struct** tagPOINT1
3. {
4. **int** x;
5. **int** y;
6. };
7. **struct** tagPOINT1 p1;

而在C++中，可以直接写成结构名对象名： tagPOINT1 p1;

用typedef写成

1. **typedef** **struct** tagPOINT
2. {
3. **int** x;
4. **int** y;
5. }POINT; 这样声明的时候，POINT p1; //这样一来可以少些一个struct，比较省事。

2）平台无关性：typedef 的一个重要的用途就是它的平台无关性，例如定义一个REAL的浮点类型在目标机器上它可以获得最该的精度。

typedef long double REAL;

在不支持long double的机器上，该typedef看起来就是下面这样的：

typedef double REAL;

在连double也不支持的机器上，该typedef看起来是这样的：

typedef float REAL;

在跨平台的时候，只需要适当地改下typedef就可以了，不需要对其他源码进行任何修改。

3） 掩饰符合类型：typedef可以掩盖符合类型，比如指针和数组。

比如在要用到相同类型和大小的数组的时候，可以这样：typedef char Line[81];

这个时候Line表示具有81个元素的字符数组，使用的方法如下：

Line text, secondline; getline(text);

### 宏函数和内联函数的区别?

答：使用宏和内联函数都可以节省在函数调用方面所带来的时间和空间开销。二者都采用了空间换时间的方式，在其调用处进行展开：

（1）在预编译时期，宏定义在调用处执行字符串的原样替换。在编译时期，内联函数在调用处展开，同时进行参数类型检查。

（2）内联函数首先是函数，可以像调用普通函数一样调用内联函数。而宏定义往往需要添加很多括号防止歧义，编写更加复杂。

（3）内联函数可以作为某个类的成员函数，这样就可以使用类的保护成员和私有成员。而当一个表达式涉及到类的保护成员或者私有成员时，宏就无法实现了（无法将this指针放到合适的位置上）。

可以用内联函数完全替代宏。

在编写内联函数时，函数体应该短小而简洁，不应该包含循环等较复杂结构，否则编译器不会将其当作内联函数看待，而是把它决议成为一个静态函数。

有些编译器甚至会优化内联函数，通常为避免一些不必要拷贝和构造，提高工作效率。

频繁的调用内联函数和宏定义容易造成代码膨胀，消耗更大的内存而造成过多的换页操作。

### C语言中宏定义（#define）时do{}while(0)的作用？

答：因为宏定义是在预编译阶段的简单替换，为了增加代码的适应性，可扩展性和灵活性，用do{} while(0);

例如：没有使用do{}while(0)，

#define FOO(x) foo(x); bar(x);

进行单独调用的时候没有什么问题的，

FOO(100) 展开之后变成foo(100); bar(100);

但是放到控制语句中就错了。

if (condition)

FOO(x);

else

...;

将会变成

if (condition)

foo(x); bar(x);

else

...;

导致了语义错误。使用#define FOO(x) do { foo(x); bar(x); } while (0) 就会解决这个问题。

### 如何实现一个不能被继承的类？

答：从理论上来说，我们只要把类的构造函数设置成私有，就能解决这个问题。因为这样的话，子类就无法访问基类的构造函数，从而阻止了进行子类构造对象的任务，也就达到了不能继承的目的。但是这样的话给我们的利用也造成了一定的困难，我们还想利用这个类。

该问题的方法是：定义静态方法，在方法内部实现一个对象，然后返回它的指针。还要设计一个释放内存的函数，就可以了。

方法一：这种方法只能在堆上创建对象，而不能在栈上创建对象。

1. #include<iostream>
2. **using**  **namespace** std;
3. **class** A
4. {
5. **public**:
6. **static** A \* Construct(**int**  n)
7. {
8. A \*pa = **new** A;
9. pa->num = n;
10. cout << "num  is:" << pa->num << endl;
11. **return** pa;
12. }
13. **static** **void** Destruct(A \* pIntance)
14. {
15. **delete**  pIntance;
16. pIntance = NULL;
17. }
19. **private**:
20. A(){} //构造函数是私有的，不对其进行实现
21. ~A(){}//析构函数也是私有的
22. **public**:
23. **int** num;
24. };
25. **void**  main()
26. {
27. A \*f = A::Construct(9);//调用静态成员函数，堆上对象
28. cout << f->num << endl;
29. A::Destruct(f);//进行析构
30. }

方法二：

1. #include<iostream>
2. **using** **namespace** std;
3. **template** <**typename** T>
4. **class** Base
5. {
6. **friend** T;
7. **private**:
8. Base() {}
9. ~Base() {}
10. };
11. **class** Finalclass : **virtual** **public** Base<Finalclass>
12. {
13. **public**:
14. Finalclass() {}
15. ~Finalclass() {}
16. };
17. **class** TestClass : **public** Finalclass
18. {
19. };
20. **void**  main()
21. {
22. Finalclass\* p = **new** Finalclass;  // 堆上对象
23. Finalclass fs;               // 栈上对象
24. //  TestClass tc;  // 基类构造函数私有，不可以被继承。因此不可以创建栈上对象。
25. system("pause");
26. }

注意这种方式的两个重要的点，首先，Base是虚基类，就意味着其所有的派生类的构造函数都会直接去调用Base基类的构造函数，其次，我们考察的类Finalclass是Base基类的友元，所以它可以访问Base基类的私有构造函数和析构函数，编译运行时正确的。但是类Finalclass的派生类TestClass的构造函数也会直接去调用Base基类的构造函数，而Base的构造函数是私有的。编译错误！所以类Finalclass是一个真正的不能继承的类。

## 二．数据结构

### 数组

#### 数组的逆序对数目

这个题是归并排序的一个变种题目。

代码如下：

1. **class** Solution {
2. **public**:
3. /\*\*
4. \* @param A: an array
5. \* @return: total of reverse pairs
6. \*/
7. **long** **long** reversePairs(vector<**int**> &A) {
8. // write your code here
9. **int** n=A.size();
10. **return** process(A,0,n-1);
11. }
12. **int** process(vector<**int**> &A, **int** start, **int** end)
13. {
14. **if**(start>=end) **return** 0;
15. **int** mid=(start+end)/2;
16. **int** left=process(A,start, mid);
17. **int** right=process(A, mid+1,end);
18. vector<**int**> copy(A);
19. **int** p=mid;
20. **int** q=end;
21. **int** k=end;
22. **int** count=0;
23. **while**(p>=start && q>=mid+1)
24. {
25. **if**(A[p]>A[q])
26. {
27. copy[k--]=A[p--];
28. count+=q-mid;
29. }
30. **else**
31. {
32. copy[k--]=A[q--];
33. }
34. }
35. **while**(p>=start) copy[k--]=A[p--];
36. **while**(q>=mid+1) copy[k--]=A[q--];
37. **for**(**int** i=start;i<=end;i++)  //这里很容易出错，只需要将A数组变化的部分更新就行了，如果整个数组更新会超时，因为做了大量的无用功。
38. {
39. A[i]=copy[i];
40. }
41. **return** count+left+right;
42. }
43. };

#### 区间合并问题

这个问题用贪心来做，我的代码效率是97%。

思路：首先将区间按照start排序，注意排序函数一定是static的。先将排序之后的第一个interval放到结果数组中，然后进行贪心，当数组中的下一个Interval的start比结果数组中的最后一个Interval的end小时，合并这个区间到结果数组的最后一个区间，改变其end值即可。否则就将区间原封不动放进结果数组中。

1. /\*\*
2. \* Definition of Interval:
3. \* classs Interval {
4. \*     int start, end;
5. \*     Interval(int start, int end) {
6. \*         this->start = start;
7. \*         this->end = end;
8. \*     }
9. \* }
10. \*/
12. **class** Solution {
13. **public**:
14. /\*\*
15. \* @param intervals: interval list.
16. \* @return: A new interval list.
17. \*/
18. **static** **bool** compare(**const** Interval &p1, **const** Interval &p2)
19. {
20. **return** p1.start<p2.start;
21. }
22. vector<Interval> merge(vector<Interval> &intervals) {
23. // write your code here
24. **if**(intervals.size()<1) **return** intervals;
25. vector<Interval> res;
26. sort(intervals.begin(),intervals.end(),compare);
27. res.push\_back(intervals[0]);
28. **for**(**int** i=1;i<intervals.size();i++)
29. {
30. **if**(res.back().end>=intervals[i].start)
31. {
32. res.back().end=max(res.back().end,intervals[i].end);
33. }
34. **else**
35. res.push\_back(intervals[i]);
36. }
37. **return** res;
38. }
39. };

#### 旋转排序数组的查找

思路： 这个题是二分查找的变种，首先如果查到了目标，返回mid就好，但是如果没有一次找到，则需要判断哪边有序，将A[mid]和A[start]相比较，得出左边或者右边有序的结论，如果左边有序，则判断如果目标在左边区间时，end=mid-1,否则start=mid+1;

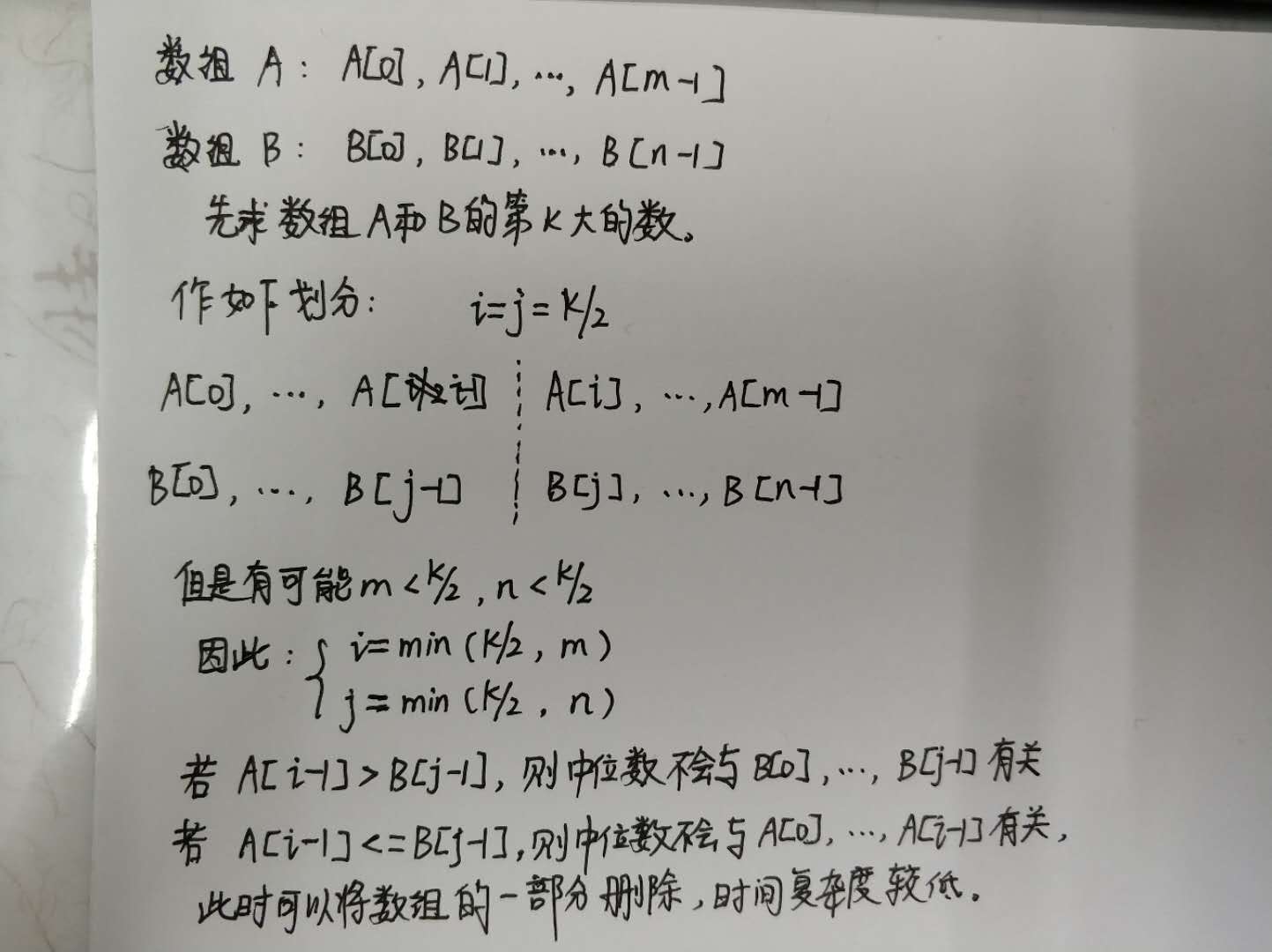
如果右边有序，则首先判断如果目标在右边区间时，start=mid+1, 否则在左边区间，end=mid-1;

如果A[mid]和A[start]相等，则无法判断哪边有序，这个时候将start向前走一步再进行判断。

1. **class** Solution {
2. **public**:
3. /\*\*
4. \* @param A: an integer rotated sorted array
5. \* @param target: an integer to be searched
6. \* @return: an integer
7. \*/
8. **int** search(vector<**int**> &A, **int** target) {
9. // write your code here
10. **int** n=A.size();
11. **int** start=0,end=n-1;
12. **while**(start<=end)
13. {
14. **int** mid=start+(end-start)/2;
15. **if**(A[mid]==target) **return** mid; //  找到了，返回下标
16. **else** **if**(A[mid]<A[start]) //右边有序
17. {
18. **if**(A[mid]<target && target<=A[end]) //目标在右边
19. start=mid+1;
20. **else** end=mid-1;
21. }
22. **else** **if**(A[mid]>A[start]) //左边有序
23. {
24. **if**(A[mid]>target && target>=A[start])//目标在左边
25. end=mid-1;
26. **else** start=mid+1;
27. }
28. **else** start++;  //无法判断，因为中间数字和开始数字相同，无法知道哪边有序
29. }
30. **return** -1;
31. }
32. };

#### 求两个排序数组的中位数的算法

要求时间复杂度是O(log(m+n))，时间复杂度是log级别的，那么一般是用到二分法来求，结合求解两个排序数组第k大的数来做，是最正确的选择。下面给出这道题的标准解法，也是最优化的解法：



1. **class** Solution {
2. **public**:
3. /\*
4. \* @param A: An integer array
5. \* @param B: An integer array
6. \* @return: a double whose format is \*.5 or \*.0
7. \*/
8. **double** findMedianSortedArrays(vector<**int**> &A, vector<**int**> &B) {
9. // write your code here
10. **int** m=A.size();
11. **int** n=B.size();
12. **int** l=(m+n+1)/2;
13. **int** r=(m+n+2)/2;
14. **int** \*l1=m>0? &A[0]: NULL;
15. **int** \*l2=n>0? &B[0]: NULL;
16. **return** (getKth(l1,m,l2,n,l)+getKth(l1,m,l2,n,r))/2.0;
17. }
18. **int** getKth(**int** \*l1, **int** m,**int** \*l2, **int** n, **int** k)
19. {
20. **if**(m>n)  //保证第一个数组是元素数目较少的
21. **return** getKth(l2,n,l1,m,k);
22. **if**(m==0) **return** l2[k-1]; //递归出口，当小数组被减少为0时，返回大数组的第k个元素
23. **if**(k==1) **return** min(l1[0],l2[0]); // 当k减少为1时，返回两个数组第一个元素的较小的那个
24. **int** i=min(k/2,m); // 划分数组
25. **int** j=min(k/2,n); // 划分数组
26. **if**(l1[i-1]>l2[j-1])
27. **return** getKth(l1,m,l2+j,n-j,k-j);
28. **else**
29. **return** getKth(l1+i,m-i,l2,n,k-i);
30. }
31. };

这道题的常规解法如下，将两个排序数组合并成为有序，然后求其中位数，但是这种方法的时间复杂度是O（m+n），空间复杂度也是O（m+n）。

1. **class** Solution
2. {
3. **public**:
4. **int** count=0;
5. priority\_queue<**int**, vector<**int**>,greater<**int**>> small;
6. priority\_queue<**int**, vector<**int**>,less<**int**>> big;
7. **double** findMedianSortedArrays(vector<**int**>& nums1, vector<**int**>& nums2)
8. {
9. **int** i=0,j=0;
10. vector<**int**> tmp;
11. tmp.reserve(nums1.size()+nums2.size());
12. **while**(i<nums1.size() && j<nums2.size())
13. {
14. **if**(nums1[i]<nums2[j])
15. {
16. tmp.push\_back(nums1[i]);
17. i++;
18. }
19. **else**
20. {
21. tmp.push\_back(nums2[j]);
22. j++;
23. }
24. }
25. **while**(i<nums1.size())
26. {
27. tmp.push\_back(nums1[i]);
28. i++;
29. }
30. **while**(j<nums2.size())
31. {
32. tmp.push\_back(nums2[j]);
33. j++;
34. }
35. **int** sum=nums1.size()+nums2.size();
36. **if**((nums1.size()+nums2.size())%2==0)
37. {
38. **return** (tmp[sum/2]+tmp[sum/2-1])/2.0;
39. }
40. **else** **return** tmp[sum/2];
41. }
42. };

#### 将一组数划分为两个部分，求两部分方差之和的最小值

思路，这个题首先先把数组进行排序，因为题目不是划分子数组，而只是把一组数字划分为两个部分，那么想到的是，相邻数字在一起的方差较小，因此排序，排好序之后，在将其划分，其实如果题目是划分子数组，方法也是一样的，只是不能排序而已。

#### 6. 求数据流的中位数，利用堆的性质。

1. **class** MedianFinder {
2. **public**:
3. /\*\* initialize your data structure here. \*/
4. priority\_queue<**int**, vector<**int**>, greater<**int**>> small;//小顶堆
5. priority\_queue<**int**, vector<**int**>, less<**int**>> big;//大顶堆
6. **int** count=0;
7. MedianFinder()
8. {
10. }
12. **void** addNum(**int** num)
13. {
14. count+=1;
15. //偶数，经小顶堆过滤进入大顶堆
16. **if**(count%2==0)
17. {
18. small.push(num);
19. big.push(small.top());
20. small.pop();
21. }
22. //奇数，经大顶堆过滤进入小顶堆
23. **else**
24. {
25. big.push(num);
26. small.push(big.top());
27. big.pop();
28. }
29. }
31. **double** findMedian()
32. {
33. **if**(count%2==0)
34. **return** **double**((small.top()+big.top())/2.0);
35. **else**
36. **return** small.top();
37. }
38. };

#### **7. Maximum Number in Mountain**

给定一个n个数的序列，数字先升后降，找出这个数组的峰值数字。

#### 8. Search in a Big Sorted Array

在一个很大的排序数组中，你无法直接获得整个数组的长度，你只能通过函数ArrayReader->get(k) for C++获得第k个数字。找出一个目标数字的Index,算法的时间复杂度是O（logk），其中k是目标数字的第一个index，如果目标数字不存在，则返回-1.

1. **int** searchBigSortedArray(ArrayReader \*reader, **int** target)
2. {
3. **int** index=1;
4. **while**(reader->get(index-1)<target)
5. index=index\*2;
6. **int** start=0, end=index-1;
7. **while**(start+1<end)  //当start和end相邻的时候退出
8. {
9. **int** mid=start+(end-start)/2;
10. **if**(reader->get(mid)>=target)  //target在左边
11. end=mid;
12. **else** start = mid;  //target在右边
13. }
14. **if**(reader->get(start)==target)  **return** start;
15. **if**(reader->get(end)==target) **return** end;
16. **return** -1;
17. }

#### 9. search range:

在一个排序数组中查找一个元素target的下标范围，以[start, end]的形式返回

1. **class** Solution {
2. **public**:
3. vector<**int**> searchRange(vector<**int**>& nums, **int** target)
4. {
5. **int** n=nums.size();
6. vector<**int**> res;
7. **if**(n==0)
8. {
9. res.push\_back(-1);
10. res.push\_back(-1);
11. **return** res;
12. }
13. **int** start=0,end=n-1;
14. //求下边界
15. **while**(start+1<end)  //两个相差1的时候退出
16. {
17. **int** mid=start+(end-start)/2;
18. **if**(nums[mid]>=target) end=mid;
19. **else** start=mid;
20. }
21. **if**(nums[start]==target) res.push\_back(start);
22. **else** **if**(nums[start]!=target && nums[end]==target) res.push\_back(end);
23. **else** res.push\_back(-1);
24. start=0,end=n-1;
25. //求上边界
26. **while**(start+1<end)
27. {
28. **int** mid=start+(end-start)/2;
29. **if**(nums[mid]>target) end=mid;
30. **else** start=mid;
31. }
32. **if**(nums[end]==target) res.push\_back(end);
33. **else** **if**(nums[end]!=target && nums[start]==target) res.push\_back(start);
34. **else** res.push\_back(-1);
35. **return** res;
36. }
37. };

#### 10. 第k大问题的各种变种题目

1. 用快速排序的思想求中位数，中位数是一个数组的中间一个数或者中间两个数， 时间复杂度平均是O(n), 最坏时间复杂度是O(n^2)。
2. **class** Solution {
3. **public**:
4. /\*\*
5. \* @param nums: A list of integers
6. \* @return: An integer denotes the middle number of the array
7. \*/
8. **int** median(vector<**int**> &nums)
9. {
10. // write your code here
11. **int** n=nums.size();
12. **int** start=0,end=n-1;
13. **if**(n%2==0)
14. **return** getKth(nums,start,end,n/2);
15. **else** **return** getKth(nums,start,end,n/2+1);
16. }
17. **int** getKth(vector<**int**> &nums, **int** start, **int** end,**int** k)
18. {
19. **int** p=partation(nums, start, end);
20. **if**(p==k-1) **return** nums[p];
21. **else** **if**(p>k-1) **return** getKth(nums,start, p-1,k);
22. **else** **return** getKth(nums,p+1,end,k);
23. }
24. **int** partation(vector<**int**> &nums, **int** start, **int** end)
25. {
26. **int** x=nums[end];
27. **int** i=start-1;
28. **for**(**int** j=start;j<end;j++)
29. {
30. **if**(nums[j]<x)
31. {
32. i++;
33. swap(nums[j],nums[i]);
34. }
35. }
36. nums[end]=nums[i+1];
37. nums[i+1]=x;
38. **return** i+1;
39. }
40. };

#### 11. 求数组中超过一半的数，保证那个数一定存在

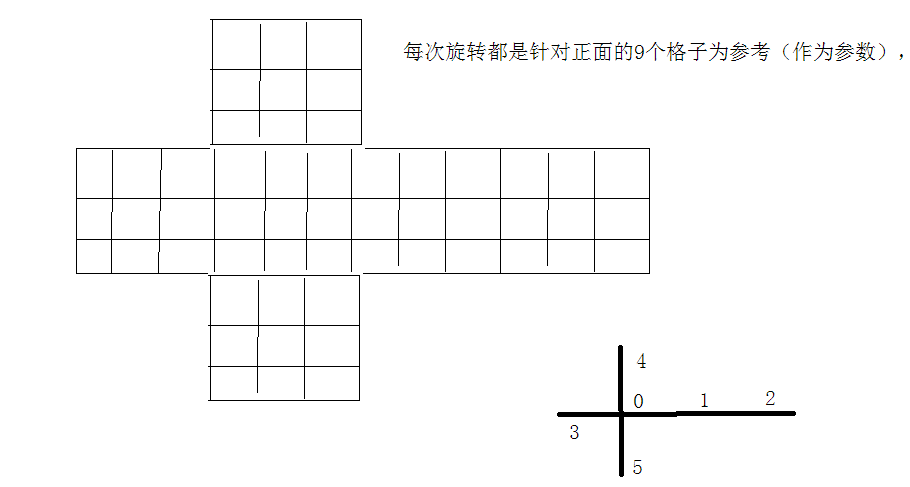
时间复杂度是O(n), 空间复杂度是O(1)。解法一的基本思路是，让result为第一个数字，然后定义一个出现次数变量times，如果后面的数字不等于result，那么times减1，times为0的时候，将result置为当前数字，times置为1.

最后还需要判断下最终的result是否是超过一半的数。

1. **class** Solution {
2. **public**:
3. **int** MoreThanHalfNum\_Solution(vector<**int**> numbers)
4. {
5. **int** m=numbers.size();
6. **int** result=numbers[0];
7. **int** times=1;
8. **for**(**int** i=1;i<m;i++)
9. {
10. **if**(times==0)
11. {
12. result=numbers[i];
13. times=1;
14. }
15. **else** **if**(result==numbers[i])
16. times++;
17. **else** times--;
18. }
19. **if**(!isMoreThanHalf(numbers,result)) **return** 0;
20. **else** **return** result;
21. }
22. **bool** isMoreThanHalf(vector<**int**> & nums, **int** result)
23. {
24. **int** n=nums.size();
25. **int** times=0;
26. **for**(**int** i=0;i<n;i++)
27. {
28. **if**(result==nums[i])
29. times++;
30. }
31. **if**(times\*2<=n) **return** **false**;
32. **else** **return** **true**;
33. }
34. };

解法二的基本思路是，利用快速排序的思想，利用求第k大的数求出数组的中位数，然后判断该中位数是不是超过一半的数，这个的平均时间复杂度也是O（n），空间复杂度也是O（1）。

#### 12. 写一个魔方的程序，六面，每一面3\*3.



1. #include <iostream>
2. **using** **namespace** std;
4. **class** MagicCube
5. {
6. **private**:
7. **enum**{LEN = 3,SPACE = 6};
8. **enum** color{red,yellow,black,blue,green,purple};
9. **enum** color Spacexy[SPACE][LEN][LEN];
10. **public**:
11. MagicCube();
12. ~MagicCube(){};
13. **void** LeftRotate(**int** y);
14. ////void RightRotate(int y);
15. **void** UpRotate(**int** x);
16. //void DownRotate(int y);
17. **void** PrintCube();
18. };
19. //构造函数
20. MagicCube::MagicCube()
21. {
22. **for**(**int** i = 0;i<6;i++)//每一面一个颜色
23. **for**(**int** j = 0;j<3;j++)
24. **for**(**int** k = 0;k<3;k++)
25. {
26. Spacexy[i][j][k] = (color)i;
27. }
28. }
29. **void** MagicCube::UpRotate(**int** y)
30. {
31. color tmp[3];
33. **for**(**int** i = 0;i<3;i++)
34. tmp[i] = Spacexy[0][i][y];
36. **for**(**int** i = 0;i<3;i++)
37. Spacexy[0][i][y] = Spacexy[5][i][y];
38. **for**(**int** i = 0;i<3;i++)
39. Spacexy[5][i][y] = Spacexy[2][i][2-y];
40. **for**(**int** i = 0;i<3;i++)
41. Spacexy[2][i][2-y] = Spacexy[4][i][y];
42. **for**(**int** i = 0;i<3;i++)
43. Spacexy[4][i][y] = tmp[i];
44. }
45. **void** MagicCube::PrintCube()
46. {
47. **for**(**int** s = 0;s<6;s++)
48. {
49. **switch**(s)
50. {
51. **case** 0:cout << " 正面:" << endl; **break**;
52. **case** 1:cout << " 右面:" << endl; **break**;
53. **case** 2:cout << " 后面:" << endl; **break**;
54. **case** 3:cout << " 左面:" << endl; **break**;
55. **case** 4:cout << " 上面:" << endl; **break**;
56. **case** 5:cout << " 下面:" << endl; **break**;
57. **default**:**break** ;
58. }
59. **for**(**int** i = 0;i<3;i++)
60. {
61. **for**(**int** j = 0;j<3;j++)
62. {
63. cout << Spacexy[s][i][j] << " ";
64. }
65. cout << endl;
66. }
67. cout << endl;
68. }
69. cout << "---------------------------------------" << endl;
70. }
71. **void** MagicCube::LeftRotate(**int** x)
72. {
73. color tmp[3];
74. **for**(**int** i = 0;i<3;i++)
75. tmp[i] = Spacexy[0][x][i];
76. **for**(**int** i = 0;i<3;i++)
77. Spacexy[0][x][i] = Spacexy[1][x][i];
78. **for**(**int** i = 0;i<3;i++)
79. Spacexy[1][x][i] = Spacexy[2][x][i];
80. **for**(**int** i = 0;i<3;i++)
81. Spacexy[2][x][i] = Spacexy[3][x][i];
82. **for**(**int** i = 0;i<3;i++)
83. Spacexy[3][x][i] = tmp[i];
84. }
85. **int** main(**void** )
86. {
87. MagicCube a;
88. a.PrintCube();
89. a.UpRotate(0,0);
90. a.PrintCube();
91. **return** 0;
92. }

### 链表

#### 1. 单向链表

单向链表的插入(注意记住其前驱结点)，反转（三个指针进行操作）,判断链表是不是带环，找出环的入口，对带环链表的深拷贝，对带random指针的链表进行深拷贝。

1. #include<iostream>
2. **using** **namespace** std;
3. **struct** ListNode
4. {
5. **int** val;
6. ListNode \*next;
7. ListNode(**int** x) :val(x), next(NULL){}
8. };
9. //单向链表的反转,三个指针进行操作，很简单
10. ListNode \*reverseList(ListNode \*head)
11. {
12. ListNode \*p1, \*p2, \*p3=NULL;
13. **if** (head == NULL || head->next == NULL) **return** head;
14. p1 = head;
15. p2 = p1->next;
16. **while** (p2)
17. {
18. p3 = p2->next;
19. p2->next = p1;
20. p1 = p2;
21. p2 = p3;
22. }
23. head->next = NULL;
24. head = p1;
25. **return** head;
26. }
27. //判断链表是不是带环，如果带环，找出环的入口节点
28. **bool** hasLoop(ListNode \*head, ListNode \*intrance)
29. {
30. ListNode \*p1, \*p2;
31. p1 = head, p2 = head;
32. **if** (p1 == NULL) **return** **false**;
33. **bool** tag = **false**;
34. **while** (p1 != NULL && p2 != NULL)
35. {
36. p1 = p1->next;
37. p2 = p2->next->next;
38. **if** (p1 == p2)
39. {
40. tag = **true**;
41. **break**;
42. }
43. }
44. **if** (tag)//说明存在环
45. {
46. p1 = head;
47. **while** (p1 != p2)
48. {
49. p1 = p1->next;
50. p2 = p2->next;
51. }
52. }
53. intrance = p1;
54. **return** tag;
55. }
56. //链表的深拷贝，链表可能有环
57. ListNode \*deepCopy(ListNode \*head)
58. {
59. **if** (!head) **return** NULL;
60. ListNode \*intrance = NULL;
61. **if** (hasLoop(head, intrance)) //如果链表有环
62. {
63. ListNode \*p = head;
64. ListNode \*copy = **new** ListNode(-1);
65. ListNode \*q = copy;
66. **while** (p != intrance)
67. {
68. q->next = **new** ListNode(p->val);
69. p = p->next;
70. q = q->next;
71. }
72. q->next = **new** ListNode(intrance->val);
73. ListNode \*cur = q->next;
74. p = intrance->next;
75. q = q->next;
76. **while** (p != intrance)
77. {
78. q->next = **new** ListNode(p->val);
79. p = p->next;
80. q = q->next;
81. }
82. q->next = cur;
83. **return** copy->next;
84. }
85. **else** //链表无环
86. {
87. ListNode \*copy = **new** ListNode(-1);
88. ListNode \*q = copy;
89. ListNode \*p = head;
90. **while** (p != NULL)
91. {
92. q->next = **new** ListNode(p->val);
93. p = p->next;
94. q = q->next;
95. }
96. q->next = NULL;
97. **return** copy->next;
98. }
99. }
100. //单链表节点的插入
101. ListNode \*insert(ListNode \*head, **int** num)
102. {
103. ListNode \*cur = **new** ListNode(num);
104. **if** (head == NULL)
105. {
106. **return** cur;
107. }
108. ListNode \*pre = head;
109. ListNode \*p=NULL; //记住前驱结点
110. **while** (num > pre->val && pre->next!=NULL)
111. {
112. p = pre; //记住前驱结点
113. pre = pre->next;
114. }
115. **if** (num <= pre->val)
116. {
117. **if** (pre == head)
118. {
119. cur->next = pre;
120. head = cur;
121. }
122. **else**
123. {
124. cur->next = pre;
125. p->next = cur;
126. }
127. }
128. **else**//该节点应该被插入到最后一个位置
129. {
130. pre->next = cur;
131. cur->next = NULL;
132. }
133. **return** head;
134. }
135. //带随机节点的链表的深复制
136. **struct** RandomListNode
137. {
138. **int** val;
139. RandomListNode \*next;
140. RandomListNode \*random;
141. RandomListNode(**int** x) :val(x), next(NULL), random(NULL){}
142. };
144. RandomListNode \*copyRandomListNode(RandomListNode \*head)
145. {
146. **if** (!head) **return** NULL;
147. RandomListNode \*pre = head;
148. //将节点复制放在原来节点的后面
149. **while** (pre)
150. {
151. RandomListNode \*cur = **new** RandomListNode(pre->val);
152. cur->next = pre->next;
153. pre->next = cur;
154. pre = cur->next;
155. }
156. pre = head;
157. RandomListNode \*cur = pre->next;
158. **while** (pre)
159. {
160. **if** (pre->random) cur->random = pre->random->next;
161. pre = cur->next;
162. **if** (pre) cur = pre->next;
163. }
164. pre = head;
165. cur = pre->next;
166. RandomListNode \*copy = cur;
167. **while** (cur->next)
168. {
169. pre->next = cur->next;
170. cur->next = cur->next->next;
171. pre = pre->next;
172. cur = cur->next;
173. }
174. pre->next = NULL;
175. **return** copy;
176. }
177. **void** print(ListNode \*head)
178. {
179. **while** (head)
180. {
181. cout << head->val << " ";
182. head = head->next;
183. }
184. }
185. **int** main()
186. {
187. ListNode \*head=NULL;
188. **for** (**int** i = 1; i < 10; i += 2)
189. {
190. head = insert(head, i);
191. }
192. head=reverseList(head);
193. print(head);
194. system("pause");
195. **return** 0;
196. }

#### 2. 双向链表的插入，删除和mergesort

1. #include<iostream>
2. **using** **namespace** std;
3. **struct** BiListNode
4. {
5. **int** val;
6. BiListNode \*pre;
7. BiListNode \*next;
8. BiListNode(**int** x) :val(x), pre(NULL), next(NULL){}
9. };
10. //双向链表的插入
11. BiListNode \*insert(BiListNode \*head, **int** num)
12. {
13. BiListNode \*node = **new** BiListNode(num);
14. **if** (head == NULL) **return** node;
15. BiListNode \*p = head;
16. **while** (node->val > p->val && p->next != NULL)
17. {
18. p = p->next;
19. }
20. **if** (node->val <= p->val)
21. {
22. //插入到头节点的位置
23. **if** (p == head)
24. {
25. node->next = head;
26. head->pre = node;
27. head = node;
28. }
29. //插入到中间位置
30. **else**
31. {
32. p->pre->next = node;
33. node->next = p;
34. node->pre = p->pre;
35. p->pre = node;
36. }
37. }
38. **else** //比哪个情况都大
39. {
40. p->next = node;
41. node->pre = p;
42. node->next = NULL;
43. }
44. **return** head;
45. }
46. //双向链表的删除
47. BiListNode \*del(BiListNode \*head, **int** num)
48. {
49. BiListNode \*p = head;
50. **while** (p->next != NULL && num != p->val)
51. p = p->next;
52. **if** (num == p->val)
53. {
54. **if** (p == head)
55. {
56. head = head->next;
57. head->pre = NULL;
58. **delete** p;
59. }
60. **else** **if** (p->next == NULL)
61. {
62. p->pre->next = NULL;
63. **delete** p;
64. }
65. **else**
66. {
67. p->pre->next = p->next;
68. p->next->pre = p->pre;
69. **delete** p;
70. }
71. }
72. **else**
73. {
74. cout << "not found" << endl;
75. }
76. **return** head;
77. }
78. //将两个排序链表合并成为一个有序链表，递归算法
79. BiListNode \*merge(BiListNode \*p1, BiListNode \*p2)
80. {
81. **if** (!p1) **return** p2;
82. **if** (!p2) **return** p1;
83. BiListNode \*head1 = p1, \*head2 = p2;
84. BiListNode \*head;
85. **if** (head1->val < head2->val)
86. {
87. head = head1;
88. BiListNode \*p= merge(head1->next, head2);
89. head->next = p;
90. p->pre = head;
91. }
92. **else**
93. {
94. head = head2;
95. BiListNode \*p= merge(head1, head2->next);
96. head->next = p;
97. p->pre = head;
98. }
99. **return** head;
100. }
101. BiListNode \*mergeNonrecur(BiListNode \*head1, BiListNode \*head2)
102. {
103. **if** (!head1) **return** head2;
104. **if** (!head2) **return** head1;
105. BiListNode \*head, \*res;
106. **if** (head1->val < head2->val)
107. {
108. head = head1;
109. res = head;
110. head1 = head1->next;
111. }
112. **else**
113. {
114. head = head2;
115. res = head;
116. head2 = head2->next;
117. }
118. **while** (head1 != NULL && head2 != NULL)
119. {
120. **if** (head1->val < head2->val)
121. {
122. head->next = head1;
123. head1->pre = head;
124. head1 = head1->next;
125. head = head->next;
126. }
127. **else**
128. {
129. head->next = head2;
130. head2->pre = head;
131. head2 = head2->next;
132. head = head->next;
133. }
134. }
135. **if** (head1)
136. {
137. head->next = head1;
138. head1->pre = head;
139. }
140. **else**
141. {
142. head->next = head2;
143. head2->pre = head;
144. }
145. **return** res;
146. }
147. **void** print(BiListNode \*head)
148. {
149. **while** (head != NULL)
150. {
151. cout << head->val << " ";
152. head = head->next;
153. }
154. cout << endl;
155. }
156. **int** main()
157. {
158. BiListNode \*head1=NULL, \*head2=NULL;
159. **for** (**int** i = 1; i <= 10; i += 2)
160. {
161. head1 = insert(head1, i);
162. }
163. head1 = insert(head1, 6);
164. **for** (**int** i = 2; i <= 10; i += 2)
165. {
166. head2 = insert(head2, i);
167. }
168. head2 = insert(head2,1);
169. head1 = del(head1, 3);
170. head2 = del(head2, 4);
172. BiListNode \*head3 = mergeNonrecur(head1, head2);
173. print(head3);
174. print(head1);
175. print(head2);
176. system("pause");
177. **return** 0;
178. }

#### 3. 循环链表

约瑟夫环问题：n个数，从1到n , 从第k个开始报数，报到第m个的出列，打印出出列的顺序。

方法一：用循环链表进行模拟

1. #include<iostream>
2. **using** **namespace** std;
3. **struct** ListNode
4. {
5. **int** val;
6. ListNode \*next;
7. ListNode(**int** value) :val(value), next(NULL){};
8. };
10. **void** YUESEFULOOP(**int** n, **int** m, **int** k)
11. {
12. //先建立环，设置当前节点位置和其前驱结点。
13. ListNode \*r, \*curr;
14. curr = **new** ListNode(1);
15. curr->next = curr;
16. **for** (**int** i = 2; i <= n; i++)
17. {
18. ListNode \*t = **new** ListNode(i);
19. t->next = curr->next;
20. curr->next = t;
21. curr = t;
22. }
23. //找出开始报数的节点
24. r = curr;
25. **while** (k--)
26. {
27. r = curr;
28. curr = curr->next;
29. }
30. //模拟报数出列的过程
31. **while** (n--)
32. {
33. //r是curr的前驱结点，找出出列的节点，curr指向它 ，走m-1步
34. **for** (**int** s = m - 1; s; r = curr, curr = curr->next, s--);
35. //r的next跳过curr指向其下一个
36. r->next = curr->next;
37. cout << curr->val << " ";
38. //删除掉curr
39. **delete** curr;
40. curr = r->next;
41. }
42. }
43. **int** main()
44. {
45. YUESEFULOOP(13, 4, 1);
46. system("pause");
47. **return** 0;
48. }

方法二：如果只是求出最后一个出列的人的位置，则可以用数学的方法来求解。

1. **void** yuesefu(**int** n, **int** m, **int** k)
2. {
3. **int** s = 0;
4. **for** (**int** i = 2; i <= n; i++)
5. {
6. s = (s + m) % i;
7. }
8. cout << (s + k)%n<< endl; //最后一个出列的人的编号
9. }

### 树

#### 1. 求从根节点到某个给定节点的路径

这个题的思路是，返回值是是否构建成功，构建成功返回true，构建失败返回false，路径数组首先将根节点放入，然后向左右树递归判断，最后pop\_back()

1. //二叉树的查找，注意，不是二叉查找树，因此没有大小比较的关系
2. **bool** getPathFromRoot(TreeNode \*root, **int** value, vector<**int**> &path)
3. {
4. **if** (root == NULL) **return** **false**;
5. path.push\_back(root->val);
6. **if** (root->val == value) **return** **true**;
7. **if** (getPathFromRoot(root->left, value, path)) //如果左子树走通了，返回true；
8. **return** **true**;
9. **if** (getPathFromRoot(root->right, value, path)) //如果右子树走通了，返回true；
10. **return** **true**;
11. path.pop\_back();
12. **return** **false**;  //左右子树都走完还没有找到，返回false.
13. }

#### 求出所有能构造出的BST

Leetcode 95题：Unique Binary Search Trees II，Given an integer n, generate all structurally unique BST's (binary search trees) that store values 1 ... n.

1. **class** Solution
2. {
3. **public**:
4. vector<TreeNode\*> generateTrees(**int** n)
5. {
6. vector<TreeNode\*> res;
7. **if**(n==0) **return** res;  //n是0的话，返回空的res；
8. **return** createTreeNode(1, n);
9. }
10. vector<TreeNode \*> createTreeNode(**int** start, **int** end)
11. {
12. vector<TreeNode\*> res;
13. **if**(start>end)//递归出口
14. {
15. res.push\_back(NULL);
16. }
17. **for**(**int** k=start;k<=end;k++)//从start到end分别作为根节点root
18. {
19. vector<TreeNode\*> left=createTreeNode(start,k-1);//左边继续进行递归
20. vector<TreeNode\*> right=createTreeNode(k+1,end);//右边也以相同的方式进行递归
21. **for**(**int** i=0;i<left.size();i++)
22. {
23. **for**(**int** j=0;j<right.size();j++)
24. {
25. TreeNode \*root=**new** TreeNode(k);//k作为跟节点
26. root->left=left[i];//左子树分别是left数组中的各个树，左树根节点分别是left数组中的每个子树的根节点
27. root->right=right[j];//右子树分别是right数组中的各个树，右树根节点分别是right数组中的每个子树的根节点
28. res.push\_back(root);//递归到跟节点了，将整个树压入到结果数组
29. }
30. }
31. }
32. **return** res;
33. }
34. };

#### 3. 二叉树的前序，中序和后序遍历的递归与非递归的写法，还有层序遍历的写法

1. **struct** TreeNode
2. {
3. TreeNode \*left, \*right;
4. **int** val;
5. TreeNode(**int** x) :val(x), left(NULL), right(NULL){}
6. };
7. //前序遍历的递归写法
8. **void** helper1(TreeNode \*root, vector<**int**> &res)
9. {
10. **if** (!root) **return**;
11. res.push\_back(root->val);
12. helper1(root->left, res);
13. helper1(root->right, res);
14. }
15. vector<**int**> preTraversalRecur(TreeNode \*root)
16. {
17. vector<**int**> res;
18. helper1(root, res);
19. **return** res;
20. }
21. //前序遍历的非递归写法
22. vector<**int**> preTraversalNonrecur(TreeNode \*root)
23. {
24. vector<**int**> res;
25. **if** (!root) **return** res;
26. stack<TreeNode \*> s;
27. TreeNode \*p = root;
28. **while** (!s.empty() || p != NULL)
29. {
30. **while** (p != NULL)
31. {
32. res.push\_back(p->val);
33. s.push(p);
34. p = p->left;
35. }
36. **if** (!s.empty())
37. {
38. p = s.top();
39. s.pop();
40. p = p->right;
41. }
42. }
43. **return** res;
44. }
46. //中序遍历的递归写法
47. **void** helper2(TreeNode \*root, vector<**int**> &res)
48. {
49. **if** (!root) **return**;
50. helper2(root->left, res);
51. res.push\_back(root->val);
52. helper2(root->right, res);
53. }
54. vector<**int**> inTraversalRecur(TreeNode \*root)
55. {
56. vector<**int**> res;
57. helper2(root, res);
58. **return** res;
59. }
60. //中序遍历非递归
61. vector<**int**> inTraversalNonrecur(TreeNode \*root)
62. {
63. vector<**int**> res;
64. **if** (!root) **return** res;
65. TreeNode \*p = root;
66. stack<TreeNode \*> s;
67. **while** (!s.empty() || p != NULL)
68. {
69. **while** (p != NULL)
70. {
71. s.push(p);
72. p = p->left;
73. }
74. **if** (!s.empty())
75. {
76. p = s.top();
77. s.pop();
78. res.push\_back(p->val);
79. p = p->right;
80. }
81. }
82. **return** res;
83. }
84. //后序遍历递归写法
85. **void** helper3(TreeNode \*root, vector<**int**> &res)
86. {
87. **if** (!root) **return**;
88. helper3(root->left, res);
89. helper3(root->right, res);
90. res.push\_back(root->val);
91. }
92. vector<**int**> postTraversalRecur(TreeNode \*root)
93. {
94. vector<**int**> res;
95. helper3(root, res);
96. **return** res;
97. }
98. //后序遍历非递归写法
99. vector<**int**> postTraversalNonrecur(TreeNode \*root)
100. {
101. vector<**int**> res;
102. **if** (!root) **return** res;
103. stack<TreeNode \*> s;
104. TreeNode \*pre=NULL, \*cur;
105. s.push(root);
106. **while** (!s.empty())
107. {
108. cur = s.top();
109. //如果当前节点是叶子节点或者当前节点的左右子节点都被访问过了，可以用前驱节点是其左右子节点来判断
110. **if** ((cur->left == NULL && cur->right == NULL) || (pre != NULL && (pre == cur->left || pre == cur->right)))
111. {
112. res.push\_back(cur->val); //访问该节点
113. pre = cur; // 使得该节点成为前驱结点
114. s.pop(); //将该节点从栈中弹出
115. }
116. //如果当前节点现在不能被访问，则将其右节点和左子节点依次放进栈中
117. **else**
118. {
119. **if**(cur->right!=NULL) s.push(cur->right);
120. **if**(cur->left!=NULL) s.push(cur->left);
121. }
122. }
123. **return** res;
124. }
125. //层序遍历，基本思想是队列，一层一层地遍历
126. vector<**int**> fromToptoBottom(TreeNode \*root)
127. {
128. vector<**int**> res;
129. queue<TreeNode \*> q;
130. **if** (!root) **return** res;
131. q.push(root);
132. **while** (!q.empty())
133. {
134. TreeNode \*p = q.front();
135. q.pop();
136. res.push\_back(p->val);
137. **if** (p->left) q.push(p->left);
138. **if** (p->right) q.push(p->right);
139. }
140. **return** res;
141. }

#### 4. 二叉查找树的查找，插入，构建和删除。

1. //二叉查找树的查找,保存路径到数组中，找到返回true，没找到就返回false
2. **bool** bstSearch(TreeNode \*root, **int** value, vector<**int**> &res)
3. {
4. **if** (root == NULL) **return** **false**; //如果根节点为空，则返回false
5. res.push\_back(root->val); //将根节点的值放到结果数组中
6. **if** (value == root->val) **return** **true**; //根节点的值等于该值，返回true；
7. **else** **if** (value < root->val)
8. {
9. **return** bstSearch(root->left, value, res);
10. }
11. **else**
12. {
13. **return** bstSearch(root->right, value, res);
14. }
15. }
16. //二叉树的查找，注意，不是二叉查找树，因此没有大小比较的关系
17. **bool** getPathFromRoot(TreeNode \*root, **int** value, vector<**int**> &path)
18. {
19. **if** (root == NULL) **return** **false**;
20. path.push\_back(root->val);
21. **if** (root->val == value) **return** **true**;
22. **if** (getPathFromRoot(root->left, value, path)) //如果左子树走通了，返回true；
23. **return** **true**;
24. **if** (getPathFromRoot(root->right, value, path)) //如果右子树走通了，返回true；
25. **return** **true**;
26. path.pop\_back();
27. **return** **false**;
28. }
29. //二叉查找树的插入
30. **bool** bstInsert(TreeNode \*&root, **int** value)
31. {
32. **if** (root == NULL)
33. {
34. root = **new** TreeNode(value);
35. **return** **true**;
36. }
37. **else** **if** (value < root->val)
38. {
39. **return** bstInsert(root->left, value);
40. }
41. **else** **if** (value > root->val)
42. {
43. **return** bstInsert(root->right, value);
44. }
45. **return** **false**;
46. }
47. //用数组构建二叉查找树
48. TreeNode \*createBST(vector<**int**> &list)
49. {
50. **int** n = list.size();
51. **if** (n == 0) **return** NULL;
52. TreeNode \*root = NULL;
53. **for** (**int** i = 0; i < n; i++)
54. {
55. **if** (!bstInsert(root, list[i]))
56. {
57. cout << list[i] << "已经存在在二叉查找树中了" << endl;
58. }
59. }
60. **return** root;
61. }
62. //二叉查找树的删除
63. /\*
64. 需要删除的节点下面没有子节点，则直接删除
65. 删除节点下面有一个左子节点/右子节点：删除该节点，并将左/右子树连接在其父节点上
66. 删除节点的左右子节点都存在：找出此节点右子树中的最小节点，替换该节点，然后删除该最小节点。
67. \*/
68. TreeNode \*removeBST(TreeNode \*root, **int** value)
69. {
70. **if** (value < root->val)
71. root->left = removeBST(root->left, value);
72. **else** **if** (value>root->val)
73. root->right = removeBST(root->right, value);
74. **else**
75. {
76. **if** (root->left == NULL || root->right == NULL)
77. {
78. root = root->left == NULL ? root->right : root->left;
79. }
80. **else** //左右子树都不为空
81. {
82. TreeNode \*p = root->right;
83. **while** (p->left != NULL)
84. {
85. p = p->left;
86. }
87. root->val = p->val;
88. root->right = removeBST(root->right, p->val);
89. }
90. }
91. **return** root;
92. }

#### 5. 二叉树的序列化和反序列化

我们介绍二叉树的先序序列化的方式，假设序列化的结果字符串是str，初始化时str等于空的字符串，先序遍历二叉树，如果遇到了空的节点，就在str的末尾加上“#!，‘#’表示节点为空，‘！’表示一个值的结束，如果遇到不是空的节点，假设节点值为3，就在str末尾加上3！。我们知道所谓的二叉树是一种由对象引用将多个节点关联起来的抽象数据结构，是存在于内存中的，不能进行持久化，如果需要将一棵二叉树的结构持久化保存，需要将其转换为字符串并保存到文件中去，于是关键是建立一套规则，使得二叉树可以与字符串一一对应，根据一个二叉树可以唯一地得到一个字符串，根据一个字符串也可以唯一地还原出一棵二叉树。所谓序列化就是持久化，显然序列化需要将每个节点按照一定的顺序转化为字符串，关键是这个顺序到底是什么顺序，这个顺序就是遍历树的顺序，按照遍历节点的顺序将节点转换为字符串就好了，因此先序遍历，中序遍历，后序遍历，按层遍历都可以进行序列化。

1. //二叉树序列化,将一棵树序列化为一个字符串，先序遍历，递归来做
2. string treetoString(TreeNode \*root)
3. {
4. string res = "";
5. **if** (root == NULL)
6. {
7. res += "#!";
8. **return** res;
9. }
10. res += to\_string(root->val);
11. res += '!';
12. res += treetoString(root->left);
13. res += treetoString(root->right);
14. **return** res;
15. }
16. //序列化之后返回的是char \* 该咋办？
17. **char** \*serialize(TreeNode \*root)
18. {
19. string s = treetoString(root);
20. **char** \*res = **new** **char**[s.size() + 1];
21. strcpy(res, s.c\_str()); // c\_str()的方法把这个string转换成 const char\*类型
22. **return** res;
23. }
25. //二叉树的反序列化
26. /\*
27. atoi()和stoi()的区别----数字字符串的处理
28. 相同点：
29. ①都是C++的字符处理函数，把数字字符串转换成int输出
30. ②头文件都是#include<cstring>
31. 不同点：
32. ①atoi()的参数是 const char\* ,因此对于一个字符串str我们必须调用 c\_str()的方法把这个string转换成 const char\*类型的,而stoi()的参数是const string\*,不需要转化为 const char\*；
33. \*/
34. TreeNode \*stringToTree(string s)
35. {
36. **if** (s.empty())
37. {
38. **return** NULL;
39. }
40. **if** (s[0] == '#') //根节点为NULL，则这棵树为空
41. {
42. s = s.substr(2); //删除两个字符#！
43. **return** NULL;
44. }
45. TreeNode \*node = **new** TreeNode(stoi(s));//将数字字符转换为数字，遇到！停下来
46. s = s.substr(s.find\_first\_of('!') + 1); //截取！之后的字符串部分
47. node->left = stringToTree(s);
48. node->right = stringToTree(s);
49. **return** node;
50. }
51. TreeNode \*Deserialize(**char** \*str)
52. {
53. string s(str);
54. **return** stringToTree(s);
55. }

#### 判断一棵二叉树的是否是镜像相等的。

1. //判断一棵树是不是镜像对称的
2. **bool** symmetric(TreeNode \*p, TreeNode \*q)
3. {
4. **if** (p == NULL && q == NULL) **return** **true**;
5. **if** (!p || !q) **return** **false**;
6. **return** (p->val == q->val && (symmetric(p->left, q->right) && symmetric(p->right, p->left)));
7. }
8. **bool** isSymmetric(TreeNode \*root)
9. {
10. **if** (root == NULL) **return** **true**;
11. **return** symmetric(root->left, root->right);

}

#### 字典树以及其实现

Trie是单词查找树，前缀树，是一种哈希树的变种。应用于字符串的统计和排序，经常被搜索引擎系统用于文本词频统计。

性质：

1. 根节点不包含字符，除了根节点之外的每一个节点都只包含一个字符。
2. 从根节点到某个节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

优点：

1. 查询快：对于长度为m的键值，最坏情况下只需要花费O(m)的时间，
2. 当存储大量字符串时，字典树耗费的空间小，因为键值并非是显式存储的，而是与其他键值共享子串。

字典树的基本操作代码

1. #include<queue>
2. #include<iostream>
3. #include<stack>
4. **using** **namespace** std;
6. **const** **int** size = 26; //26个字母
8. **struct** TrieTreeNode
9. {
10. **char** val;//字符
11. **bool** isEnd; //是否是单词结尾
12. **int** childCnt;//当前节点的孩子个数
13. **int** prefixCnt;//单词前缀个数
14. TrieTreeNode \*child[size]; //子节点的指针数组
15. //构造函数
16. TrieTreeNode(**char** \_val) :val(\_val), isEnd(**false**), childCnt(0), prefixCnt(0)
17. {
18. memset(child, NULL, **sizeof**(child)); //不是26
19. }
20. };
21. //将word插入到字典树中
22. **void** Insert(TrieTreeNode \*&root, **const** **char** \*word)
23. {
24. TrieTreeNode \*p = root;
25. **for** (**int** i = 0; i < strlen(word); i++)
26. {
27. //如果当前字母位置的孩子是NULL
28. **if** (p->child[word[i] - 'a'] == NULL)
29. {
30. p->child[word[i] - 'a'] = **new** TrieTreeNode(word[i]);
31. p->childCnt++; //p节点的孩子数目加1
32. }
33. //前缀中包含的word[i]的单词又多了一个
34. p->child[word[i] - 'a']->prefixCnt++;
35. p = p->child[word[i] - 'a'];
36. }
37. p->isEnd = **true**;//将一个单词插入到字典树之后，最后一个位置是一个单词的结尾。
38. }
39. //字典树的查找
40. **bool** find(TrieTreeNode \*root, **const** **char** \*word)
41. {
42. TrieTreeNode \*p = root;
43. **for** (**int** i = 0; i < strlen(word); i++)
44. {
45. **if** (p->child[word[i] - 'a'] == NULL)
46. **return** **false**;
47. p = p->child[word[i] - 'a'];
48. }
49. **return** p->isEnd;
50. }
51. //层序遍历字典树
52. **void** LevelOrderTraverse(TrieTreeNode \*root)
53. {
54. **if** (root==NULL)
55. **return**;
56. queue<TrieTreeNode \*> q;
57. q.push(root);
58. **while** (!q.empty())
59. {
60. TrieTreeNode \*p = q.front();
61. cout << p->val << "(" << p->childCnt << ")";
62. **for** (**int** i = 0; i < size; i++)
63. {
64. **if** (p->child[i] != NULL)
65. q.push(p->child[i]);
66. }
67. q.pop();
68. }
69. cout << "\n";
70. }
71. **void** PreOrderTraverse(TrieTreeNode \*treeNode)
72. {
73. **if** (treeNode != NULL)
74. {
75. cout << treeNode->val << "(" << treeNode->childCnt << ") ";
76. **for** (**int** i = 0; i < size; i++)
77. {
78. PreOrderTraverse(treeNode->child[i]);
79. }
80. }
81. }
83. **void** PostOrderTraverse(TrieTreeNode \*treeNode)
84. {
85. **if** (treeNode != NULL)
86. {
87. **for** (**int** i = 0; i < size; i++)
88. {
89. PostOrderTraverse(treeNode->child[i]);
90. }
91. cout << treeNode->val << "(" << treeNode->childCnt << ") ";
92. }
93. }
94. //清空整棵树
95. **void** MakeEmpty(TrieTreeNode \*&treeNode)
96. {
97. **if** (treeNode != NULL)
98. {
99. **for** (**int** i = 0; i < size; i++)
100. MakeEmpty(treeNode->child[i]);
101. **delete** treeNode;
102. }
103. treeNode = NULL;
104. }
105. //构建一棵字典树
106. **void** BuildTrieTree(TrieTreeNode \*&root, **const** **char** \*words[], **int** n)
107. {
108. **for** (**int** i = 0; i < n; i++)
109. Insert(root, words[i]);
110. }
111. //在字典树中删除一个单词
112. /\*
113. 1.如果途中某个字符找不到，则不删除
114. 2.若到达了字符串末尾的下一个位置，说明字符串的所有字符都已经找到，把包含字符串最后一个字符的那个节点设置为false,表明这个位置不是一个单词的结尾了。
115. 如果保存字符串中最后那个字符的节点是个叶子节点（childcnt=0），则删除这个节点。
116. 接下来便一层一层地向上返回，判断当卡节点是删除还是保留，若当前节点孩子为0，并且当前节点不是一个单词的结尾的时候（isend=false），则删除该节点，否则不删除。
117. \*/
118. **bool** remove(TrieTreeNode \*treeNode, **const** **char** \*word, **int** n)
119. {
120. stack<TrieTreeNode \*> s;  //用一个栈记住路径
121. TrieTreeNode \*p = treeNode;
122. **if** (p == NULL ) **return** **false**; //树为空，则没找到
123. s.push(p); //将根节点放到栈中
124. **int** i = 0; //初始位置为0
125. **while** (i<n)
126. {
127. **if** (p!=NULL && p->child[word[i] - 'a'] != NULL)  //路径正确，将路径节点依次放到栈中
128. {
129. s.push(p->child[word[i] - 'a']);
130. p = p->child[word[i] - 'a'];
131. i++;
132. }
133. **else** **return** **false**; //路径不正确，说明word不在字典树中，返回false。
134. }
135. **if** (i == n)
136. {
137. p->isEnd = **false**;//找到了单词，要删除它的第一步就是让它的尾节点的isend值为false
138. **int** count = 0; //统计出删除节点的个数，为上层节点的前缀prefixcnt的值变化做铺垫：prefixcount-=count;
139. **while** (!s.empty())
140. {
141. **if** (!p->isEnd && p->childCnt == 0) //可以删除该节点的条件
142. {
143. count++; //删除节点数加一
144. s.pop();
145. --i;
146. **if** (!s.empty())
147. {
148. p = s.top(); //被删除节点的上层父节点
149. --p->childCnt;   //被删除节点的父节点的子节点数减一
150. **delete** p->child[word[i] - 'a']; //删除该节点
151. p->child[word[i] - 'a'] = NULL; //将该节点位置置NULL；
152. p->prefixCnt -= count; //被删除节点的父节点的前缀值减去其下面已经删除节点的个数
153. }
154. }
155. **else** **break**;
156. }
157. }
158. **return** **true**;
159. }
160. **bool** Remove(TrieTreeNode \*&treeNode, **const** **char** \*word, **int** pos, **int** n)
161. {
162. **if** (pos == n)
163. {
164. treeNode->isEnd = **false**;//set the node not to be an end
165. //if the last node contains the last char is a leaf,return true to delete it
166. **return** treeNode->childCnt == 0;
167. }
169. //not found, not delete this node
170. **if** (treeNode->child[word[pos] - 'a'] == NULL)
171. **return** **false**;
173. //if true, the child is a leaf, delete the child
174. **if** (Remove(treeNode->child[word[pos] - 'a'], word, pos + 1, n))
175. {
176. **delete** treeNode->child[word[pos] - 'a'];
177. treeNode->child[word[pos] - 'a'] = NULL;
179. treeNode->prefixCnt--;
180. //if the node becomes a leaf and is not an end,return true to delete it
181. **if** (--treeNode->childCnt == 0 && treeNode->isEnd == **false**)
182. **return** **true**;
183. }
185. //other not delete
186. **return** **false**;
187. }
189. //统计某个前缀的单词的个数
190. **int** CountWordsWithPrefix(TrieTreeNode \*root, **const** **char** \*prefix)
191. {
192. TrieTreeNode \*p = root;
193. **for** (**int** i = 0; i < strlen(prefix); i++)
194. {
195. **if** (p->child[prefix[i] - 'a'] == NULL)
196. **return** 0;
197. p = p->child[prefix[i] - 'a'];
198. }
199. **return** p->prefixCnt;
200. }
201. **int** main()
202. {
203. TrieTreeNode \*root = **new** TrieTreeNode('\0');
204. **const** **char** \*words[] = { "abc", "ad", "ef" };//test remove
205. BuildTrieTree(root, words, **sizeof**(words) / **sizeof**(words[0]));
206. LevelOrderTraverse(root);
207. PreOrderTraverse(root);
208. cout << '\n';
209. PostOrderTraverse(root);
210. cout << "\n";
212. **if** (find(root, "ef"))
213. cout << "ef found" << endl;
214. **else** cout << "ef not found" << endl;
216. Insert(root, "e");
217. //after this insertion.the node 'e' becomes a end but it's not a leaf,
218. //so it can not be deleted unless its leaf(leaves) is deleted
219. LevelOrderTraverse(root);
221. Remove(root, "ef", 0,strlen("ef"));
222. LevelOrderTraverse(root);
224. Remove(root, "e",0,strlen("e"));
225. LevelOrderTraverse(root);
227. cout << CountWordsWithPrefix(root, "a") << endl;
228. Remove(root, "ad",0,strlen("ad"));
229. cout << CountWordsWithPrefix(root, "a") << endl;
231. MakeEmpty(root);
232. system("pause");
233. **return** 0;
234. }

#### 红黑树都有哪些特点？

答：红黑树，全称是Red-Black Tree，他是一种特殊的二叉查找树。红黑树的每个节点都有存储位表示节点的颜色，可以是红也可以是黑的。

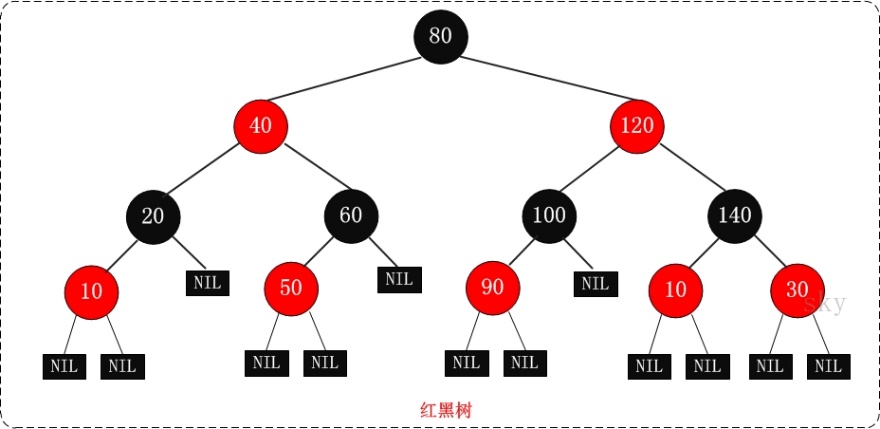
1）每个节点是黑色或者红色。

2）根节点是黑色的。

3）每个叶子节点是黑色的（这里的叶子节点是指空节点）。

4）如果一个节点是红色的，则它的子节点必须是黑色的。

5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。



注意：1）特性三中的叶子节点，是只为空的节点

1. 特性五，确保没有一条路径会比其他路径长出两倍，因而，红黑树是接近平衡的二叉树。
2. 定理：一棵含有n个节点的红黑树的高度至多为2log(n+1)。
3. AVL树和红黑树都是二叉搜索树的变体，都用来搜索。AVL树是严格平衡的，而红黑树是黑平衡的。但是维持平衡又需要额外的操作，这也加大了数据结构的时间复杂度，所以红黑树可以看作是二叉搜索树和AVL树的一个折衷，可以尽量维持树的平衡，又不花过多的时间来维持数据结构的性质。
4. AVL树的左右子树的树高差不超过1，只要插入和删除不满足平衡条件就通过旋转来保持平衡。旋转式非常耗费时间的。AVL树用于插入和删除次数较少，但是查找多的情况。应用较少，windows对于进程地址空间的管理就用到了AVL树。

红黑树的应用：主要用红黑树来存储有序的数据，时间复杂度是O（logn），效率很高。在C++STL中的set、map以及Linux虚拟内存管理，都用的是红黑树来实现的。总结其应用：

1. 广泛用于c++的STL中, map和set都是红黑树来实现的。
2. 著名的Linux进程调度Completely Fair Scheduler，用红黑树管理进程控制块。
3. Epoll在内核中的实现，用红黑树管理事件块。
4. Nginx中，用红黑树来管理timer等。
5. Java的TreeMap的实现。

#### 9. B树，B-树，B+树，B\*树分别是什么，特点是什么？

答：(1) B树就是二叉搜索树，这个就不过多地解释了。如果B树的所有非叶子节点的左右子树的节点数目保持差不多（平衡），那么B树的搜索性能逼近二分查找；但是它比连续内存空间的二分查找的优点是，改变B树的结构不需要移动大段的内存数据，甚至通常是常数开销。实际使用的B树都是在原B树的基础上加上平衡算法，即“平衡二叉树”；如何保持B树结点分布均匀的平衡算法是平衡二叉树的关键；平衡算法是一种在B树中插入和删除结点的策略；

（2）B-树是一种多路搜索树，不是二叉的：

1. 定义任意非叶子节点最多只有M个儿子，且M>2；

2. 根节点的儿子个数是[2，M]。

3. 除了根节点以外的非叶子节点的儿子数是[M/2, M]；

4. 每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

5. 非叶子结点的关键字个数=指向儿子的指针个数-1；

6. 非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

7. 非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

8. 所有叶子结点位于同一层；

(3) B+树是B-树的变体，也是一种多路搜索树：

1. 其定义基本与B-树同，除了：

2. 非叶子结点的子树指针与关键字个数相同；

3. 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

4. 为所有叶子结点增加一个链指针；

5. 所有关键字都在叶子结点出现；

小节：

B树：二叉树，每个节点只存储一个关键字，等于则命中，小于则走左节点，大于则走右节点。

B-树：多路搜索树，每个节点存储M/2到M个关键字，非叶子节点存储指向关键字范围的子节点，所有关键字在整棵树中出现，且只出现一次，非叶子节点可以命中；

B+树：在B-树的基础之上，为叶子节点增加链表指针，所有关键字都在叶子节点中出现，非叶子节点作为叶子节点的索引；B+树总是到叶子节点才命中。

应用：为了将大型数据库文件存储在硬盘上，以减少访问硬盘次数为目的，在此提出了一种平衡多路查找树——B-树结构。由其性能分析可知它的检索效率是相当高的 为了提高 B-树性能’还有很多种B-树的变型，力图对B-树进行改进，比如B+树。

### 队:先进先出

#### 1. 编程实现队列的入队/出队操作

利用链表来做，注意queue要定义构造函数。需要考虑队列为空的情况，队列不为空的情况。在出队操作中还需要考虑队列中只有一个元素的情况。

1. #include<iostream>
2. **using** **namespace** std;
4. **struct** node
5. {
6. **int** val;
7. node \*next;
8. node(**int** x) :val(x), next(NULL){};
9. };
10. **struct** queue
11. {
12. node \*front, \*rear;
13. queue() :front(NULL), rear(NULL){};
14. };
15. //队列的入队
16. queue \*insert(queue \*HQ, **int** x)
17. {
18. node \*s = **new** node(x);
19. //如果队列为空，队列的头和尾都指向这个节点
20. **if** (HQ->rear == NULL)
21. {
22. HQ->front = s;
23. HQ->rear = s;
24. }
25. //队列不为空，直接在其后面加上这个链表节点就好了
26. **else**
27. {
28. HQ->rear->next = s;
29. HQ->rear = s;
30. }
31. **return** HQ;
32. }
34. //队列的出队
35. queue \*pop(queue \*HQ)
36. {
37. node \*p;
38. **if** (HQ->rear == NULL)
39. {
40. cout << "队列为空" << endl;
41. }
42. **else**
43. {
44. p = HQ->front;
45. //队列中只有一个元素
46. **if** (HQ->front == HQ->rear)
47. {
48. HQ->front = NULL;
49. HQ->rear = NULL;
50. }
51. **else**
52. {
53. HQ->front = HQ->front->next;
54. **delete**(p);//删除掉头节点；
55. }
57. }
58. **return** HQ;
59. }
61. **int** main()
62. {
63. queue \*q=**new** queue();
65. insert(q, 2);
66. insert(q, 3);
67. insert(q, 4);
68. pop(q);
69. **while** (q->front != q->rear)
70. {
71. cout << q->front->val << " ";
72. q->front = q->front->next;
73. }
74. cout << q->front->val << endl;
75. system("pause");
76. **return** 0;
77. }

#### 2. 利用数组实现循环队列

思路：这是一道腾讯面试题，考察基本的数据结构。

1. #include<iostream>
2. **using** **namespace** std;
3. //必须注意的是，队列一定需要包含其头和尾。
4. **class** queue
5. {
6. **int** size;//数组的大小，队列中元素最多存size-1个，有一个位置来区分循环队列是满还是空
7. **int** head; //队列的头
8. **int** end; //队列的尾
9. **int** array1[1000]; //定义数组
10. **public**:
11. queue(){}
12. queue(**int** n, **int** front, **int** tail)
13. {
14. size = n;
15. head = front;
16. end = tail;
17. }
18. **void** insert(**int**); //向循环队列中添加元素
19. **int** top(); //取出循环队列的头元素的值
20. **void** pop(); //出队
21. **bool** isempty(); //判断队列是不是空
22. **bool** isfull(); //判断队列是不是满
23. **int** getlength(); //获得当前队列的大小
24. };
25. **bool** queue::isfull()
26. {
27. **return** (end + 1) % size == head; //规定的是两个差1则队列是满的
28. }
29. **bool** queue::isempty()
30. {
31. **return** head == end; //规定的是两个指向同一个位置则是空的
32. }
33. **void** queue::insert(**int** r)
34. {
35. **if** (queue::isfull())
36. {
37. cout << "The queue is full" << endl;
38. **return**;
39. }
40. array1[end] = r;
41. end = (end + 1) % size;//找到下一个正确的索引，注意end的位置始终是空的
42. }
43. **void** queue::pop()
44. {
45. **if** (queue::isempty())
46. {
47. cout << "The queue is empty" << endl;
48. **return**;
49. }
50. head = (head + 1) % size;//找到下一个正确的索引
51. }
52. **int** queue::getlength()
53. {
54. **return** (end - head + size) % size;
55. }
56. **int** queue::top()
57. {
58. **return** array1[head]; //返回头节点的值
59. }
60. **int** main()
61. {
62. **int** n;
63. cin >> n;
64. queue q(n, 1,1);//初始时，
65. q.insert(7);
66. q.insert(8);
67. q.insert(9);
68. //q.pop();
69. cout << q.getlength() << endl;
70. **int** t = q.getlength();
71. **for** (**int** i = 0; i < t; i++)
72. {
73. cout << q.top() << endl;
74. q.pop();
75. }
76. system("pause");
77. **return** 0;
78. }

### 栈:先进后出

#### 1, 首先用链表实现一个栈，包括入栈和出栈的操作

1. #include<iostream>
2. **using** **namespace** std;
3. //链表节点的结构体
4. **struct** node
5. {
6. **int** val;
7. node \*next;
8. node(**int** x) :val(x), next(NULL){}
9. };
10. //栈的数据结构，需要栈顶和栈底的指针，并将其初始化为NULL
11. **struct** stack
12. {
13. node \*bottom, \*top;
14. stack():bottom(NULL), top(NULL){}
15. };
16. //入栈
17. stack \*push(stack \*s, **int** x)
18. {
19. node \*temp = **new** node(x);
20. //如果栈是空的，top和bottom都指向该节点，也就是将该节点入栈了，作为栈底元素
21. **if** (s->bottom == NULL)
22. {
23. s->bottom = temp;
24. s->top = temp;
25. }
26. **else**//如果栈不是空的，栈顶增长即可，top->next指向temp。
27. {
28. s->top->next = temp;
29. s->top = s->top->next;
30. }
31. **return** s;
32. }
33. //出栈
34. stack \*pop(stack \*s)
35. {
36. node \*p, \*q;
37. **if** (s->bottom == NULL)
38. {
39. cout << "The stack is empty" << endl;
40. }
41. **else**//栈不为空
42. {
43. **if** (s->bottom == s->top) //栈中只有一个元素，栈顶和栈底相等，将这两个指针都置为空就实现了出栈
44. {
45. s->bottom = NULL;
46. s->top = NULL;
47. }
48. **else**
49. {
50. p = s->bottom;
51. **while** (p->next != s->top) //首先找出top的前一个节点
52. {
53. p = p->next;
54. }
55. q = s->top; //q指向栈顶元素
56. s->top = p; //top指向它的前一个节点
57. **delete** q;  //将栈顶元素析构
58. }
59. }
60. **return** s;
61. }
62. **int** main()
63. {
64. stack \*s=**new** stack();
65. push(s, 1);
66. push(s, 2);
67. push(s, 3);
68. push(s, 4);
69. pop(s);  //4 出栈
70. **while** (s->top!= s->bottom)
71. {
72. cout << s->top->val << endl;
73. pop(s);
74. }
75. cout << s->bottom->val << endl; //依次输出3，2，1，和入栈方向相反。
76. system("pause");
77. **return** 0;
78. }

#### **2. 用两个栈来实现一个队列**

用两个栈实现队列的push和pop操作，队列中的元素是整型的。对于队列，push是加到队尾，pop是从队头弹出。

1. **class** queue
2. {
3. stack<**int**> s1;
4. stack<**int**> s2;
5. **public**:
6. **void** push(**int** x)
7. {
8. **if** (s1.empty())
9. {
10. **while** (!s2.empty())
11. {
12. **int** temp = s2.top();
13. s1.push(temp);
14. s2.pop();
15. }
16. }
17. s1.push(x);
18. }
19. **int** pop()
20. {
21. **if** (!s2.empty()) //连续两次出队，提高效率，s2不为空的时候，直接返回s2的栈顶元素，这时s1一定是空的。
22. {
23. **int** temp = s2.top();
24. s2.pop();
25. **return** temp;
26. }
27. **else** //s2为空的时候，将s1中的元素倒入到s2中，之后再将s2的栈顶元素出栈并返回。
28. {
29. **if** (!s1.empty())
30. {
31. **int** temp = s1.top();
32. s2.push(temp);
33. s1.pop();
34. }
35. **int** temp = s2.top();
36. s2.pop();
37. **return** temp;
38. }
39. }
40. };

#### **定义一个栈的数据结构，要求能够返回栈中最小的元素**

思路就是两个栈s1和s2，s2的栈顶维护栈中的最小值。

1. **class** minstack
2. {
3. stack<**int**> s1, s2; // s2栈顶始终维护栈中的最小值
4. **public**:
5. **void** push(**int** value)
6. {
7. s1.push(value);
8. **if** (s2.empty())
9. {
10. s2.push(value);
11. }
12. **else**
13. {
14. **if** (s2.top() > value)
15. s2.push(value);
16. }
17. }
18. **void** pop()
19. {
20. **if** (s1.top() == s2.top())
21. {
22. s2.pop(); //当前出栈元素是最小元素，则s2也出栈
23. }
24. s1.pop();
25. }
26. **int** top()
27. {
28. **return** s1.top();
29. }
30. **int** getmin()
31. {
32. **return** s2.top();
33. }
34. };

#### 4．判断入栈和出栈的顺序

两个整数序列，一个是栈的压入顺序，请判断第二个序列是不是这个栈的合法的弹出顺序，我们假设压入栈中的元素并无重复的。思路是：借助一个辅助栈来模拟入栈和出栈的顺序，最后判断该栈是不是空的。

1. **bool** isPopOrder(vector<**int**> pushV, vector<**int**> popV)
2. {
3. //有一个序列为空或者两个序列的大小不一样都不符合
4. **if** (pushV.empty() || popV.empty() || pushV.size() != popV.size())
5. **return** **false**;
6. stack<**int**> s;
7. **int** i, j;
8. **for** (i = 0, j = 0; i < pushV.size(); i++)
9. {
10. s.push(pushV[i]);
11. **while** (!s.empty() && s.top() == popV[j])
12. {
13. j++;
14. s.pop();
15. }
16. }
17. **if** (s.empty()) **return** **true**;
18. **else** **return** **false**;
19. }

### 6. 图以及图论

#### 1. 最短路径问题的Dijkstra算法：贪心算法的思想

该算法的实现部分比较复杂，但是实现过程需要熟记。

Dijkstra算法是图论中典型的带权最短路径优先算法，其基本思想是广度优先策略，以起始点为中心向外层层扩展，直到找到终点为止。

基本思想：

1. 首先需要指定起始点s（从起始点s开始计算）。
2. 此外，引进两个集合S和U。S的作用是记录已经求出的最短路径的顶点（以及相应的最短路径的长度），而U是记录还未能求出最短路径的顶点（以及该定点到起点s的距离）。
3. 初始时，S中只有s;U中是除了s之外的顶点，并且U中顶点的路径是“起点s到该定点的路径”。然后，从U中找出路径最短的顶点，并将其加入到S中去；接着更新U中的顶点和顶点对应的路径。然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中顶点和顶点对应的路径。重复该操作，直到遍历完所有的顶点。

操作步骤：

1）初始时，S只包含起点s，U包含除了s之外的所有节点，并且U中顶点的路径是s到该顶点的距离[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为∞]。

2）从U中选出“距离最短的顶点k”，并将顶点k加入到S中；同时，从U中移除顶点k。

3）更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是因为上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离，比如，(s,v)的距离可能大于(s,k)+(k,v)的距离。

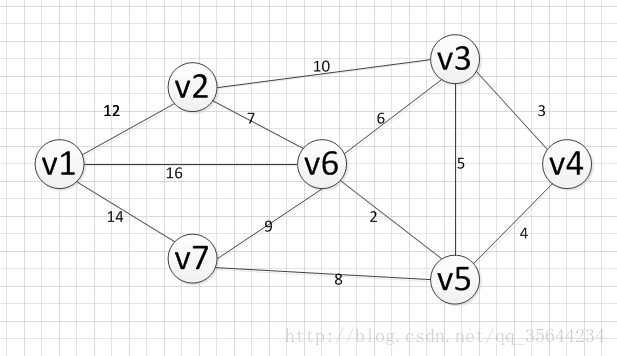
4）重复2）和3），直到遍历完所有的顶点。

上面提到Dijkstra算法，该算法是单源最短路径算法，只能计算从指定节点到其它节点的最短路径。而且权重不得出现负的值。该算法的原始时间复杂度是O(n^2)，经过堆优化之后的时间复杂度是O(nlogn)，其中n是顶点的个数。

设起点为s，dis[v]表示从s到v的最短路径，pre[v]为v的前驱节点，用来输出路径。   
a)初始化：dis[v]=∞(v≠s); dis[s]=0; pre[s]=0;   
b)For (i = 1; i <= n ; i++)   
1.在没有被访问过的点中找一个顶点u使得dis[u]是最小的。   
2.u标记为已确定最短路径   
3.For 与u相连的每个未确定最短路径的顶点v   
if (dis[u]+w[u][v] < dis[v])   
{   
dis[v] = dis[u] + w[u][v];   
pre[v] = u;   
}   
c)算法结束：dis[v]为s到v的最短距离；pre[v]为v的前驱节点，用来输出路径。

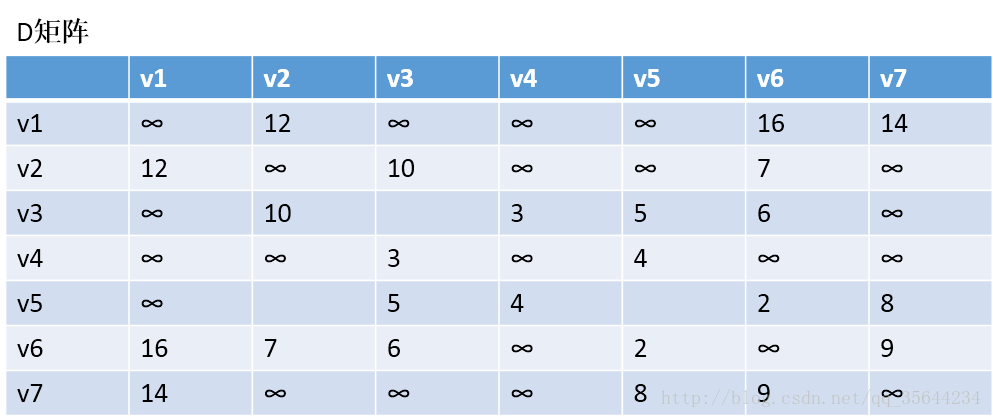
#### 2. 最短路径问题的Floyd算法

弗洛伊德算法（Floyd）是计算图中任意两个顶点之间的加权最短路径算法，可以正确处理有向图或者负权(不可以出现负权路径)的最短路径问题，同时被用于计算有向图的传递闭包。

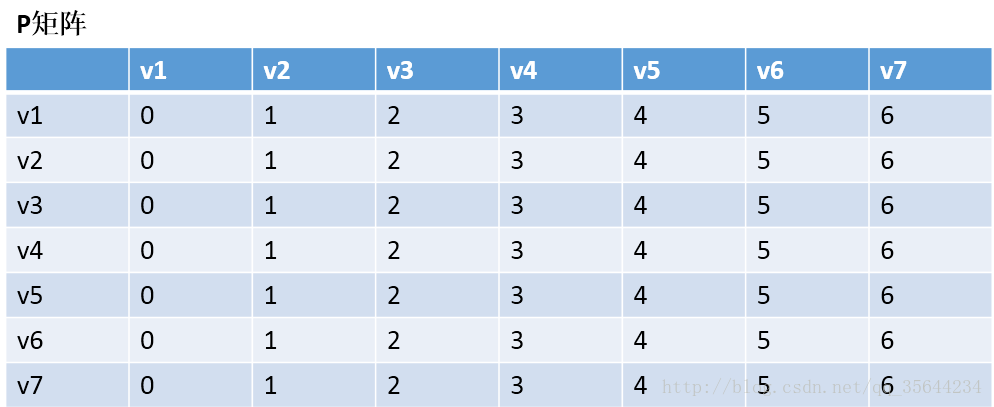


该算法的思路是：首先需要引入两个矩阵，一个矩阵S中的元素a[i][j]表示从顶点i到顶点j的距离。矩阵P中的元素b[i][j]表示从顶点i到顶点j经过了b[i][j]记录的值表示的顶点。

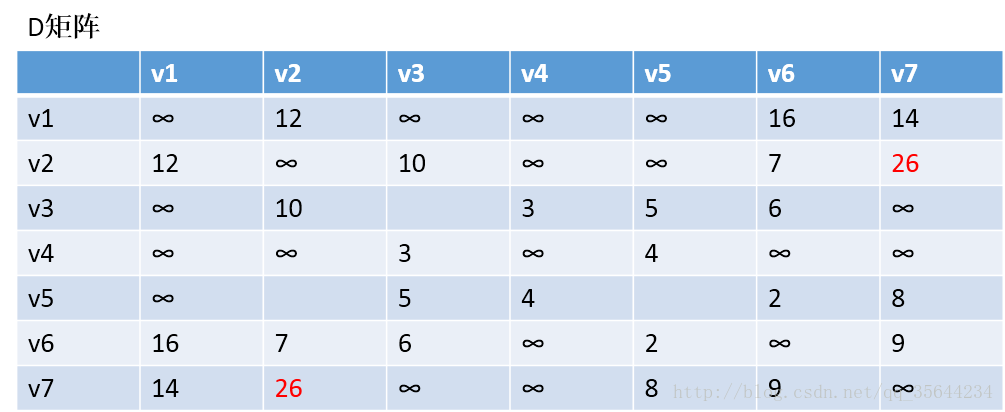
S矩阵初始化：S初始化为各个边的权重

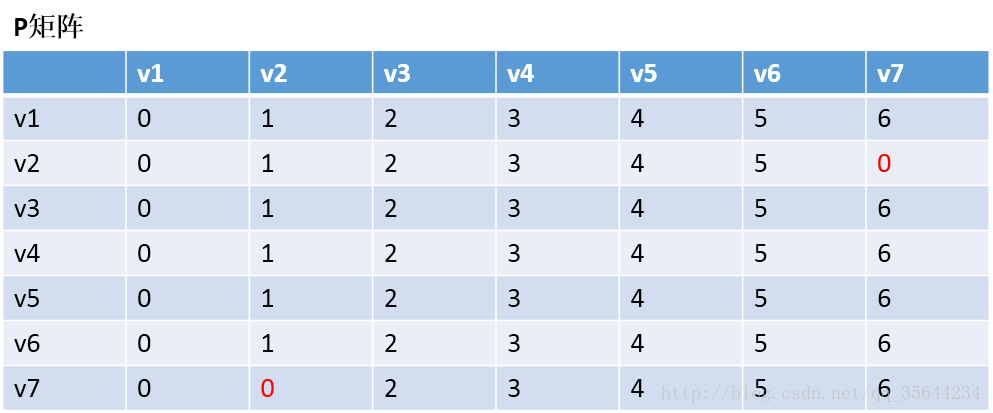


P矩阵初始化：P初始化为终节点，表示初始时刻就是从源节点到终节点的连接。



第二步是分别以每个节点为中介，更新两个矩阵，以v1为中介，更新两个矩阵：发现，a[1][0]+a[0][6] < a[1][6] 和a[6][0]+a[0][1] < a[6][1]，所以我们只需要矩阵D和矩阵P，结果如下：





#### 3. 最短路径问题的SPFA算法

适用于权值有负值，且没有负圈的图的**单源**最短路径，论文中的复杂度O(kE)，k为每个节点进入Queue的次数，且k一般<=2，但此处的复杂度证明是有问题的，其实SPFA的最坏情况应该是O(VE)。

#### 最短路径问题的Bellman－Ford算法

也是求**单源**最短路径的算法，给定一个源，找到该源到其它所有顶点的最短路径，但是图中可能存在负权重的边，其和dijkstra不同的地方在于，dijkstra算法不能用于有负权的网络中，但是该算法可以。**当权值有负值，而且可能存在负值，则用BellmanFord，并且能够检测并输出负圈。时间复杂度是O(VE)。**

以上四种带权的图的最短路径算法的比较：

1. 当权值为非负时候，用Dijkstra.
2. 当权值有负且没有负圈，则用SPFA，SPFA能够检测负圈，但是不能输出负圈。
3. 当权值有负值，而且可能存在负圈，则用BellmanFord，能够检测并且输出负圈。
4. SPFA检测负圈：当存在一个点入队次数大于等于V次，则有负环。
5. 单源算法：Dijkstra，SPFA，BellmanFord；
6. 求出每对节点之间的最短路径：Floyd，该算法的时间复杂度是O(n^3)。

输入：图和源顶点src

输出：从src到所有顶点的最短距离。如果有负权回路（不是负权值的边），则不计算最短距离，因为没有意义，因为可以穿越负权回路任意次，则最终为负无穷。

**算法步骤：**

Bellman－Ford算法可以大致分为三个部分

第一，初始化所有点。每一个点保存一个值，表示从原点到达这个点的距离，将原点的值设为0，其它的点的值设为无穷大（表示不可达）。

第二，进行循环，循环下标为从1到n－1（n等于图中点的个数）。在循环内部，遍历所有的边，进行松弛计算。

第三，遍历途中所有的边（edge（u，v）），判断是否存在这样情况：

d（v） > d (u) + w(u,v)

则返回false，表示途中存在从源点可达的权为负的回路。

之所以需要第三部分的原因，是因为，如果存在从源点可达的权为负的回路。则 应为无法收敛而导致不能求出最短路径。

**该算法与Dijkstra算法的区别：**

（1）Dijkstra算法在求解的过程中，源点到集合S内各个顶点的最短路径一旦求出，之后就不再变了，修改的仅仅是源点到T集合中各个顶点的路径长度。

（2）Bellman算法在求解的过程中，每次循环都要修改所有顶点的dist[]，也就是说源点到各个顶点的最短路径长度要等到算法结束才能确定下来。

如果用邻接表表示存储图，内层的两个for循环可以改成while循环，可以将算法的复杂度降为O(n\*e)。邻接表里直接存储的是边的信息，浏览完所有的边，时间复杂度是O（e）。而邻接矩阵是间接存储边，浏览完所有的边，时间复杂度是O（n^2）。

具体描述是：对图中的每条有向边<u,v>，权值是w，如果dist[u]+w<dist[v]，即有向边<u,v>的引入，会缩短v0到顶点v的最短路径长度，那么应该修改dist[v]，修改成dist[u]+w。

1. #define MAX 999999
2. #define EDGE\_MAX 100
3. #define VER\_MAX 50
4. **struct** Edge
5. {
6. **int** u,v,w;//边，起点，终点和权值
7. }；
8. Edge edge[EDGE\_MAX]; //存储所有的边
9. **int** m;  //实际边的个数
10. **int** n;   //顶点个数
11. /\* dist是源点v0到各个顶点的最短距离，如果初始为v0到各个顶点的直接边的长度，则算法中的循环要执行n-2次，如果初始化为MAX,则要循环执行n-1次，第一次求得的dist就是v0到各个顶点直接边的长度。\*/
12. **int** dist[VER\_MAX]={INT\_MAX};
13. //假定边的数组、边的个数这些信息已经读进来了
14. **bool** bellman\_ford()
15. {
16. **int** i,k,t;
17. **for**(i=1;i<n;i++)
18. {
19. /\*假定第k条边的起点是u，终点是v，以下循环考虑第k条边是否会使得源点v0到v最短距离缩短，也就是判断dist[edge[k].u]+edge[k].w<dist[edge[k].v]是不是成立\*/
20. **for**(k=0;k<m;k++)
21. {
22. t=dist[edge[k].u]+edge[k].w;
23. **if**(dist[edge[k].u]!=INT\_MAX && t< dist[edge[k].v])
24. {
25. dist[edge[k].v]=t;
26. }
27. }
28. }
29. }
30. /\*以下算法是检查的过程，如果还有更新，则说明存在无限循环的负环\*/
31. **for**(k=0;k<m;k++)
32. {
33. **if**(dist[edge[k].u]!=INT\_MAX && dist[edge[k].u]+edge[k].w<dist[edge[k].v])
34. **return** **false**;
35. }
36. **return** **true**;
37. }

#### 5. 最小生成树概念

图论中几个概念的定义：

连通图：在无向图中，如果任意两个顶点都有路径联通，则该无向图是连通图。

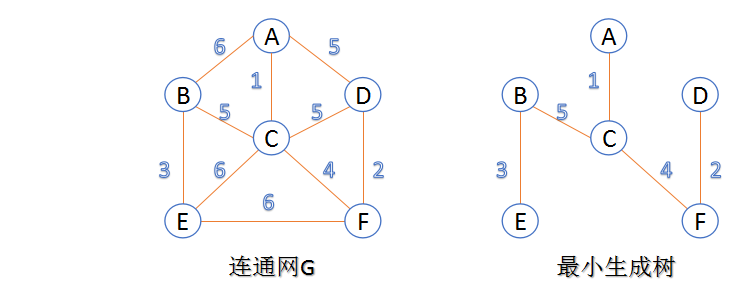
强连通图：在有向图中，若任意两个顶点之间都有路径相通，则称该有向图为强连通图。

连通网：在连通图中，若图的边具有一定的意义，每一条边都对应一个数，称为权，权代表这条边的代价，称为连通图叫做连通网。

生成树：一个连通图的生成树是指一个连通子图，它含有图中全部的n个顶点，但只有足以构成一棵树的n-1条边。一棵有n个顶点的生成树只有n-1条边，如果生成树再增加一条边，就一定会成环。

最小生成树：在连通网的所有生成树中，所有的边的代价和最小的生成树成为最小生成树。

可见，连通图，连通网和生成树的概念都是在带权的无向图的基础上建立的。



#### 6. 最小生成树之Kruskal算法

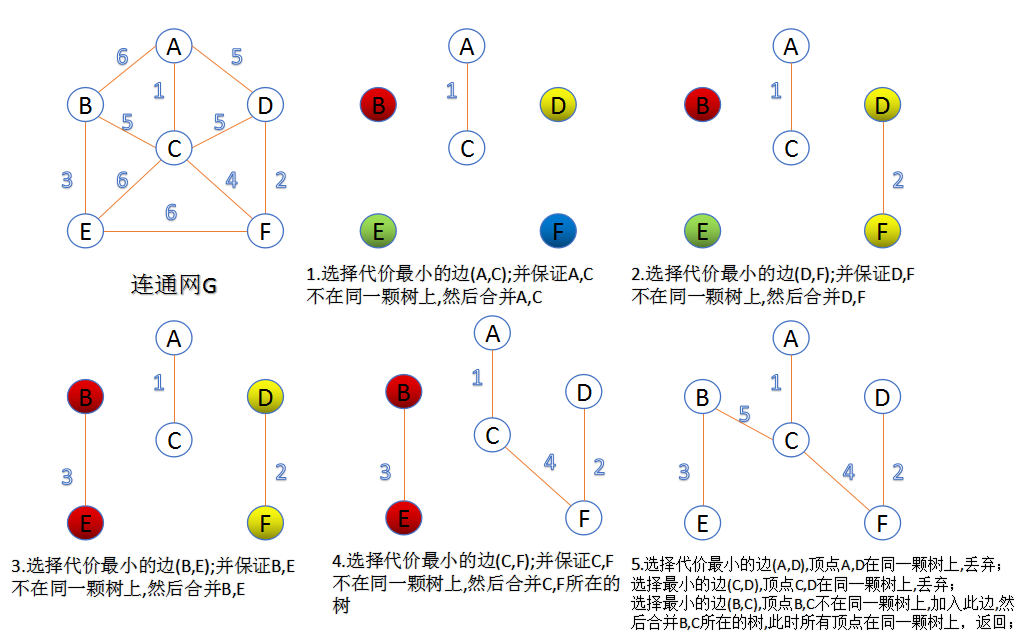
这个算法称为加边法，初始最小生成树的边数为0，每迭代一次就选择一条满足条件的最小代价边，加入到最小生成树的边的集合中。

1. 把图中的所有边按照代价从小到大排序；

2. 把图中的n个顶点看成独立的棵树组成的森林；

3. 按照权值从小到大选择边，所选的边连接的两个顶点ui和vi应该属于两棵不同的树，则成为最小生成树的一条边，并将这两棵树合并作为一棵树。

4．重复3，直到所有的顶点都在一棵树内或者有n-1条边为止。



时间复杂度主要是在对边进行排序的过程，O(ElgE)。

#### 7. 最小生成树之Prim算法

该算法可以称为加点法，每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个节点开始，逐渐长大覆盖整个连通网的所有顶点。

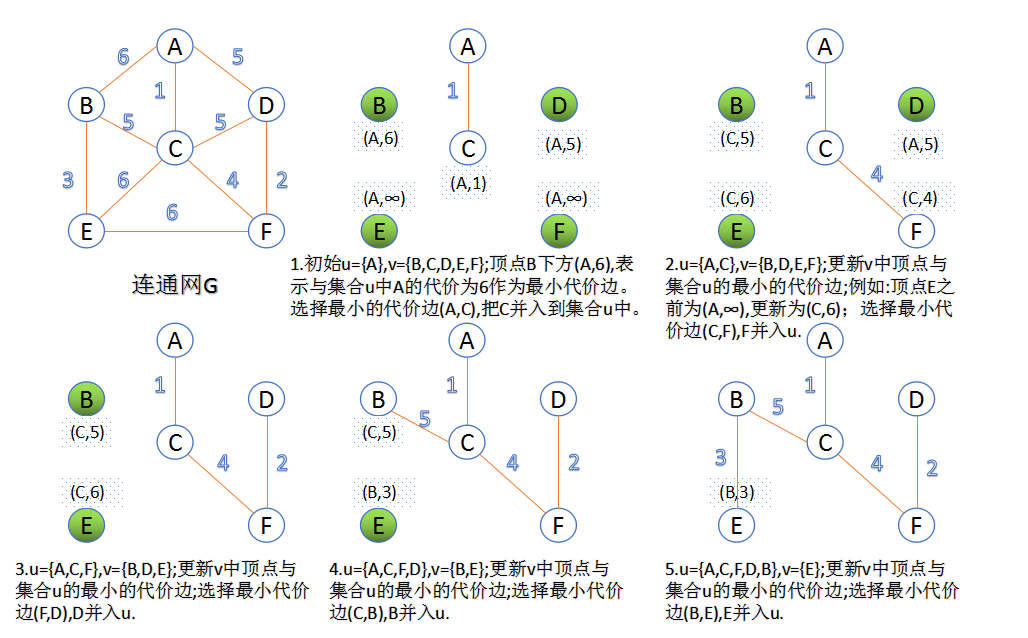
1. 图中的所有顶点集合为V; 初始令集合u={s},v=V-u;

2. 在两个集合u,v能够组成的边中，选择一条代价最小的边（u0,v0），加入到最小生成树中，并把v0并入到集合u中。

3．重复以上步骤，直到最小生成树有n-1条边或者n个顶点为止。

由于不断向集合u中加入顶点，所以最小代价边必须同步更新；需要建立一个辅助数组closedge，用来维护v中每个顶点与集合u中顶点的最小代价边的信息。

1. **struct**
2. {
3. **char** vertexData   //表示u中顶点信息
4. **UINT** lowestcost   //最小代价
5. }closedge[vexCounts]



**Prim算法和Kruskal算法的比较：**

1. prim在稠密图中比Kruskal优，在稀疏图中比Kruskal劣。

2. prim+heap在任何时候都有令人满意的时间复杂度，但是空间代价极大（以及代码很复杂）。

3. 时间复杂度并不能反映一个算法的实际优劣

4.平时用到的稀疏图比较多，所以尽可能用prim+heap,在稀疏图中这是无敌的。如果一定要在朴素prim和kruskal中选择一个的话就用kruskal。但是prim代码简单一些，只要不是极其稀疏的图，两者的性能相差不是很大。

**应用**：利用最小生成树能够解决旅游线路的问题，将旅游线路中旅游景点看作是图中的顶点，各个景点之间的路线看作图中顶点之间的边，景点之间路线的长度看作图中各个边的权值。这样我们就把旅游线路问题转换成为一个有向连通网的最小生成树的问题。

#### 8. 两种求最小生成树的代码：

**(1) Prim算法**

1. //返回最小代价边的函数
2. **int** Graph\_DG::Minmum(**struct** node\* closedge)
3. {
4. unsigned **int** min = INT\_MAX;
5. **int** index = -1;
6. **for** (**int** i = 0; i < **this**->vexnum; i++)
7. {
8. //在u中的节点的最小代价边都置为0了不再重复将其加入到u集合中，只考虑v中节点为起点的最小代价边，求其与u中节点之间的最小边代价
9. **if** (closedge[i].lowestcost != 0 && closedge[i].lowestcost < min)
10. {
11. min = closedge[i].lowestcost;
12. index = i;
13. }
14. }
15. **return** index; //返回的一定是v中顶点
16. }
17. **void** Graph\_DG::MiniSpanTree\_Prim(**int** s)
18. {
20. **for** (**int** i = 0; i < **this**->vexnum; i++)
21. {
22. closedge[i].lowestcost = INT\_MAX;
23. }
24. closedge[s].data = s; //从顶点s开始
25. closedge[s].lowestcost = 0;
26. //初始化辅助数组
27. **for** (**int** i = 0; i < **this**->vexnum; i++)
28. {
29. **if** (i != s)
30. {
31. closedge[i].data = s;
32. closedge[i].lowestcost = arc[s][i];
33. }
34. }
35. //n-1条边时退出
36. **for** (**int** e = 1; e <= **this**->vexnum-1; e++)
37. {
38. //选择最小代价边，k是在v集合中选中的节点，其与u中的节点拥有最小代价边，因此将k加入到u集合中。
39. **int** k = Minmum(closedge);
40. //该点加入到最小生成树中，k进入u集合中。
41. cout << closedge[k].data+1 << "---"<< k+1<< endl;
42. //将k的代价设置为0，k在u中了，它不再作为最小代价边的起点，但是它依然可以作为最小代价边的终点，用这种方式区分u和v集合
43. closedge[k].lowestcost = 0;
44. //更新v中顶点的最小代价边信息
45. **for** (**int** i = 0; i <**this**-> vexnum; i++)
46. {
47. //这里的i都是v集合里的，不可能是u集合里的，因为u集合里面的点的最小边代价都是0，不可能大于原始边权。
48. //因为u每加入一个节点k，就更新一次最小代价边信息，每次更新的都是u中顶点和v中顶点之间的最小代价，不存在v中顶点之间的最小代价
49. **if** (arc[k][i]<closedge[i].lowestcost)
50. {
51. closedge[i].data = k; //v中顶点的最小代价边都是只面向u中顶点的
52. closedge[i].lowestcost = arc[k][i]; //v中顶点的最小代价是面向与u中顶点的
53. }
54. }
55. }
56. }

**(2) kruskal算法**

1. //该函数保存图的边信息
2. **void** Graph\_DG::ReadArc(vector<Arc> &vertexArc)
3. {
4. Arc \*temp = NULL;
5. **for** (unsigned **int** j = 0; j < vexnum; j++)
6. {
7. **for** (unsigned **int** i = 0; i < vexnum; i++)
8. {
9. **if** (arc[j][i] != INT\_MAX)
10. {
11. temp = **new** Arc;
12. temp->u = j;
13. temp->v = i;
14. temp->cost = arc[j][i];
15. vertexArc.push\_back(\*temp);
16. }
17. }
18. }
19. }
20. //比较函数，用于对边进行排序
21. **bool** Graph\_DG::compare(Arc a, Arc b)
22. {
23. **return** a.cost < b.cost ? **true** : **false**;
24. }
25. //该函数的作用是，首先检查u,v是否在同一棵树上，如果在，则返回false，如果不在则将两棵树合并，并返回true
26. **bool** Graph\_DG::FindTree(unsigned **int** u, unsigned **int** v, vector<vector<unsigned **int**>> &Tree)
27. {
28. unsigned **int** index\_u = INT\_MAX;
29. unsigned **int** index\_v = INT\_MAX;
30. //检查u,v分别属于哪棵树
31. **for** (unsigned **int** i = 0; i < Tree.size(); i++)
32. {
33. **if** (find(Tree[i].begin(), Tree[i].end(), u) != Tree[i].end())
34. index\_u = i;
35. **if** (find(Tree[i].begin(), Tree[i].end(), v) != Tree[i].end())
36. index\_v = i;
37. }
38. **if** (index\_u != index\_v)//u,v不在一棵树上，合并两棵树
39. {
40. **for** (unsigned **int** i = 0; i < Tree[index\_v].size(); i++)
41. {
42. Tree[index\_u].push\_back(Tree[index\_v][i]);
43. }
44. Tree[index\_v].clear();
45. **return** **true**;
46. }
47. **return** **false**;//u，v在一棵树上
48. }
49. **void** Graph\_DG::MiniSpanTree\_Kruskal()
50. {
51. vector<Arc> vertexArc;
52. ReadArc(vertexArc); //将其转换为边的信息
53. sort(vertexArc.begin(), vertexArc.end(),Graph\_DG::compare);
54. vector<vector<unsigned **int**>> Tree(vexnum); //开始的时候，每个顶点就是一棵树
55. **for** (unsigned **int** i = 0; i < vexnum; i++)
56. {
57. Tree[i].push\_back(i);//初始化森林
58. }
59. //按照从小到大的顺序取出最小代价边
60. **for** (unsigned **int** i = 0; i < vertexArc.size(); i++)
61. {
62. unsigned **int** u = vertexArc[i].u;
63. unsigned **int** v = vertexArc[i].v;
64. **if** (FindTree(u, v, Tree))//检查这个边的两个顶点是不是在一棵树内
65. {
66. cout << u+1 << "---" << v+1<< endl; // 把这个边加入到最小生成树中，在这里直接输出来就好
67. }
68. }

}

#### A\*寻路

代码的Github地址：<https://github.com/lingyun3730/Astar_cpp>

A\*寻路算法是结合了Dijkstra算法与最佳优先搜索的一种启发式算法。

Dijkstra算法从物体所在的初始点开始，访问图中的结点。它迭代检查待检查结点集中的结点，并把和该结点最靠近的尚未检查的结点加入待检查结点集。该结点集从初始结点向外扩展，直到到达目标结点。Dijkstra算法保证能找到一条从初始点到目标点的最短路径，只要所有的边都有一个非负的代价值。

　 最佳优先搜索（BFS）算法按照类似的流程运行，不同的是它能够评估（称为启发式的）任意结点到目标点的代价。与选择离初始结点最近的结点不同的是，它选择离目标最近的结点。BFS不能保证找到一条最短路径。然而，它比Dijkstra算法快的多，因为它用了一个启发式函数（heuristic function）快速地导向目标结点。与Dijkstra 算法相比，BFS运行得更快。

和Dijkstra一样，A\*能用于搜索最短路径。和BFS一样，A\*能用启发式函数（注：原文为heuristic）引导它自己。在简单的情况中，它和BFS一样快。成功的秘决在于，它把Dijkstra算法（靠近初始点的结点）和BFS算法（靠近目标点的结点）的信息块结合起来。在讨论A\*的标准术语中，g(n)表示从初始结点到任意结点n的代价，h(n)表示从结点n到目标点的启发式评估代价（heuristic estimated cost）。

一种极端情况，如果h(n)是0，则只有g(n)起作用，此时A\*演变成Dijkstra算法，这保证能找到最短路径。

如果h(n)经常都比从n移动到目标的实际代价小（或者相等），则A\*保证能找到一条最短路径。h(n)越小，A\*扩展的结点越多，运行就得越慢。

如果h(n)精确地等于从n移动到目标的代价，则A\*将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等（译者：指让h(n)精确地等于实际值）。只要提供完美的信息，A\*会运行得很完美，认识这一点很好。

如果h(n)有时比从n移动到目标的实际代价高，则A\*不能保证找到一条最短路径，但它运行得更快。

另一种极端情况，如果h(n)比g(n)大很多，则只有h(n)起作用，A\*演变成BFS算法。

#### 10. 并查集

#### 11. 图的基本数据结构，克隆图

1. /\*\*
2. \* Definition for undirected graph.
3. \* struct UndirectedGraphNode {
4. \*     int label;
5. \*     vector<UndirectedGraphNode \*> neighbors;
6. \*     UndirectedGraphNode(int x) : label(x) {};
7. \* };
8. \*/

克隆图：克隆一个无向图，每个节点包含一个标签和一个他的邻居列表，我们如何序列化这个有向图：每一个节点被唯一地标记。

思路：用BFS，首先定义两个map，一个map表示该节点是否已经被标记，另一个表示新图和旧图之间的映射，如果节点为空，返回空，

1. **class** Solution {
2. **public**:
3. UndirectedGraphNode\* cloneGraph(UndirectedGraphNode\* node) {
4. // write your code here
5. **if**(node==NULL)
6. **return** NULL;
7. queue<UndirectedGraphNode\*> q;
9. map<UndirectedGraphNode \*, **bool**> isvisited;
10. map<UndirectedGraphNode \*, UndirectedGraphNode \*> mp;
11. UndirectedGraphNode\* newnode=**new** UndirectedGraphNode(node->label);
12. isvisited[node]=**true**;
13. mp[node]=newnode;
14. q.push(node);
15. **while**(!q.empty())
16. {
17. node=q.front();
18. q.pop();
19. **for**(**int** i=0;i<node->neighbors.size();i++)
20. {
21. **if**(!isvisited[node->neighbors[i]])
22. {
23. UndirectedGraphNode \*temp=**new** UndirectedGraphNode(node->neighbors[i]->label);
24. isvisited[node->neighbors[i]]=**true**;
25. mp[node->neighbors[i]]=temp;
26. q.push(node->neighbors[i]);
27. mp[node]->neighbors.push\_back(temp);
28. }
29. **else**
30. {
31. mp[node]->neighbors.push\_back(mp[node->neighbors[i]]);
32. }
33. }
34. }
35. **return** newnode;
36. }
37. };

#### 12. 图的拓扑排序

1. /\*\*
2. \* Definition for Directed graph.
3. \* struct DirectedGraphNode {
4. \*     int label;
5. \*     vector<DirectedGraphNode \*> neighbors;
6. \*     DirectedGraphNode(int x) : label(x) {};
7. \* };
8. \*/

拓扑排序：给定一个有向图，图节点的拓扑排序被定义为：对于每条有向边A->B，则A必须排在B的前面，拓扑排序的第一个节点可以是任何在图中没有其他节点指向他的节点。

找到给定图的任一拓扑排序。

思路：首先用getIndegree求出每个节点的入度，利用BFS来求解，队列里存放的是入度为0的节点，从队列的头节点开始，将其子节点的入度分别减一，如果其子节点的入度为0，将该子节点入队，返回结果数组。

1. **class** Solution {
2. **public**:
3. /\*
4. \* @param graph: A list of Directed graph node
5. \* @return: Any topological order for the given graph.
6. \*/
7. vector<DirectedGraphNode\*> topSort(vector<DirectedGraphNode\*>& graph) {
8. // write your code here
9. vector<DirectedGraphNode\*> res;
10. map<DirectedGraphNode\*,**int**> mp;
11. queue<DirectedGraphNode\*> q;
12. getIndegree(graph,mp);
13. **for**(auto i:graph)
14. {
15. **if**(mp[i]==0)
16. q.push(i);
17. }
18. **while**(!q.empty())
19. {
20. DirectedGraphNode\* tmp=q.front();
21. q.pop();
22. res.push\_back(tmp);
23. **for**(**int** i=0;i<tmp->neighbors.size();i++)
24. {
25. --mp[tmp->neighbors[i]];
26. **if**(mp[tmp->neighbors[i]]==0)
27. q.push(tmp->neighbors[i]);
28. }
29. }
30. **return** res;
31. }
32. **void** getIndegree(vector<DirectedGraphNode\*>& g, map<DirectedGraphNode\*,**int**> &mp)
33. {
34. **for**(**int** i=0;i<g.size();i++)
35. {
36. **for**( **int** j=0;j<g[i]->neighbors.size();j++)
37. {
38. ++mp[g[i]->neighbors[j]];
39. }
40. }
41. }
42. };

#### 13. 判断一个无向图是不是树

判断无向图是否是树：给定的是节点和边的信息

思路：首先是将节点和边的信息转换成为邻接矩阵的形式，判断图是否是树，首先判断这个图有没有环，如果有环则不是，如果无环，则判断是否有孤立的节点，如果没有，则是树，如果有则不是。判断是否存在环的函数中包含参数邻接矩阵，访问状态矩阵，父节点和当前的子节点。利用DFS求解，因为父节点被访问了，如果当前的子节点也是被访问的状态，那么存在环，这是basis的逻辑，针对初始节点的邻接节点，进行深度优先遍历，如果存在环则返回true结束。

1. **class** Solution {
2. **public**:
3. /\*\*
4. \* @param n: An integer
5. \* @param edges: a list of undirected edges
6. \* @return: true if it's a valid tree, or false
7. \*/
8. **bool** validTree(**int** n, vector<vector<**int**>> &edges) {
9. // write your code here
10. //首先建立邻接表
11. vector<vector<**int**>> neighbors(n);
12. **for**(auto i: edges)
13. {
14. neighbors[i[0]].push\_back(i[1]);
15. neighbors[i[1]].push\_back(i[0]);
16. }
17. vector<**bool**> visited(n,0);
18. **if**(hasCycle(neighbors, visited, -1,0)) **return** **false**;
19. **for**(**int** i=0;i<n;i++)
20. {
21. **if**(!visited[i]) **return** **false**;
22. }
23. **return** **true**;
24. }
25. **bool** hasCycle(vector<vector<**int**>> &neighbors, vector<**bool**> &visited, **int** par,**int** kid)
26. {
27. **if**(visited[kid]) **return** **true**;
28. visited[kid]=1;
29. **for**(auto i:neighbors[kid])
30. {
31. **if**(i!=par)
32. {
33. **if**(hasCycle(neighbors,visited,kid,i))
34. **return** **true**;
35. }
36. }
37. **return** **false**;
38. }
39. };

### 7. 字符串

#### 1. 字符串循环移动

用三个反转就可以了

1. **class** solution
2. {
3. **public**:
4. **void** MyReverse(string &s, **int** i, **int** j)
5. {
6. **while**(i<j)
7. {
8. swap(s[i++],s[j--]);
9. }
10. }
11. //循环左移
12. **void** LeftRotate(string &s, **int** m)
13. {
14. **int** n=s.size();
15. m%=n;
16. MyReverse(s,0,m-1);
17. MyReverse(s,m,n-1);
18. MyReverse(s,0,n-1);
19. }
20. };

#### 2. 求出最长的无重复字符的子串

1. **class** Solution
2. {
3. **public**:
4. **int** lengthOfLongestSubstring(string s)
5. {
6. map<**char**,**int**> mp;
7. **int** pre=0;
8. **int** maxlen=0;
9. **for**(**int** i=1;i<=s.size();i++)
10. {
11. **if**(mp[s[i-1]]>=pre)
12. pre=mp[s[i-1]];   //pre指向上一个s[i-1]出现的位置
13. mp[s[i-1]]=i; //s[i-1]位置更新
14. maxlen=max(maxlen,i-pre);//最大值更新
15. }
16. **return** maxlen;
17. }
18. };

#### **同构字符串** Isomorphic Strings (leetcode 205)

给定两个字符串s和t，如果s能经过字符替换得到t，那么返回true，否则返回false; 要注意的是这是个一一映射的过程，所以要正反交叉验证，用两个map来做。

1. **bool** isIsomorphic(string s, string t)
2. {
3. **int** n=s.size();
4. **int** m=t.size();
5. **if**(m!=n) **return** **false**;
6. unordered\_map<**char**, **char**> mps;
7. unordered\_map<**char**, **char**> mpt;
8. **for**(**int** i=0;i<m;i++)
9. {
10. mps.insert(make\_pair(s[i],t[i]));
11. mpt.insert(make\_pair(t[i],s[i]));
12. }
13. string sret,tret;
14. **for**(**int** i=0;i<m;i++)
15. {
16. sret+=mps[s[i]];
17. tret+=mpt[t[i]];
18. }
19. **return** sret==t && tret==s;
20. }

### 8. 哈希表

#### 1. 概念和基本思想

哈希法又称为散列法、杂凑法以及关键字地址计算法等，相应的表称为哈希表。

基本思想是，首先在元素的关键字K和元素的位置P之间建立起一个对应关系f，使得P=f(K)，其中f称为哈希函数。创建哈希表时，把关键字K的元素直接存入地址为f(K)的单元，查找关键字K的元素时利用哈希函数计算出该元素的存储位置P=f(K).

当关键字集合很大的时候，关键字值得不同元素可能会映像到哈希表的同一地址上，即K1!=K2，但是f(K1)=f(K2),这种现象称为hash冲突，实际中冲突是不可避免的，只能通过改进哈希函数的性能来减少冲突。

#### 2. 哈希函数的构造原则

函数本身容易计算，计算出来的地址分布均匀。也就是说，对于任意K，f(K)对应于不同地址的概率应该相等。有以下方法：1）数字分析法；2）平方取中法；3）分段叠加法；4）除留余数法；

#### 3. 冲突处理的方式：

1)开放定址法，也就是再散列的方法；2）再哈希法：这个方法是同时构造多个不同的哈希函数，当发生冲突的时候，再计算另一个哈希函数，直到冲突不再发生，这个方法不容易产生聚集，但是增加了计算时间。3）拉链法(HashMap的冲突处理方式)链地址法：将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。4）建立公共溢出区：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

## 三．基本常见算法题

### 1. 排序类算法

#### 1. 稳定排序的概念和常见排序算法的比较

**排序稳定性：**队列中存在两个相等的数字排序过程中，这两个数字的先后顺序如果不会发生变化，就叫做稳定排序，否则不稳定。

几种排序的情况

1. 排序方法   平均情况          最好情况    最坏情况    辅助空间     稳定性
2. 冒泡排序   O(n^2)             O(n)        O(n^2)       O(1)        稳定
3. 简单选择   O(n^2)             O(n^2)      O(n^2)       O(1)        稳定
4. 直接插入   O(n^2)             O(n)        O(n^2)       O(1)        稳定
5. 希尔排序   O(nlogn)~O(n^2)    O(n^1.3)    O(n^2)       O(1)        不稳定
6. 堆排序     O(nlogn)           O(nlogn)    O(nlogn)     O(1)        不稳定
7. 归并排序   O(nlogn)           O(nlogn)    O(nlogn)     O(n)        稳定
8. 快速排序   O(nlogn)           O(nlogn)    O(n^2)       O(logn)~O(n)不稳定

**插入排序：**是一种简单的排序方法，基本思想是通过构建有序序列，对于未排序的数据，在已排序序列中从后往前扫描，找到相应位置并插入。思考抓牌的过程。

**希尔排序**：对直接插入排序的一种优化，就是把直接插入排序改成了分组插入排序。基本思想是将整个待排序元素序列按照gap分割成为N个组，对每个组进行直接插入排序，然后再减小gap进行直接插入排序，直到gap达到最小时，即数组基本达到有序时，再对数组进行直接插入排序，此时直接插入排序就可以达到最高效率。

**选择排序：**每次从无序数组中找到最小的放到最前面，再从剩下的元素中找出次小的放在无序区的最前面，直到数组有序为止。

**堆排序：**使用大顶堆升序，基本实现原理也是一种选择排序，它同样是确定了为止选择符合位置的元素，但是堆排序是更加优化的选择排序的版本，它利用了堆的特性，父节点的值大于子节点，且满足完全二叉树，大大提高了选择排序的效率。

**冒泡排序：**交换式的排序方式，基本思想是相近的两个数字比较，小的放在前面，大的放在后面，按照这个规则从头向后比较，最大的数就沉在了数组的最后。

**快速排序：**应用场景是大规模的数据排序，并且实际性能要好于归并排序。它的基本原理是从数组中选取一个元素，把所有大于它的元素都放在它的后面，所有小于它的元素都放在它的前面，然后这个元素就把原数组分成了两个部分，再分别进行同样的操作，直到数组不能再切分此时数组有序。

**归并排序：**归并是指将两个或者若干个有序表组合成一个新的有序表，归并排序和快速排序一样也是采用分治的思想，它的基本原理是通过对若干个有序节点序列的合并为一个有序序列来实现排序的。

#### 2. 排序分类方式

**稳定排序和不稳定排序**

队列中存在两个相等的数字排序过程中，这两个数字的先后顺序如果不会发生变化，就叫做稳定排序，否则不稳定

**插入类**：直接插入； 希尔排序

**选择类排序**：简单选择，堆排序

**交换类排序**：冒泡排序，快速排序

**归并类排序:** 归并排序

#### 3. 快速排序代码以及快速排序的应用

1. //quick sort
2. **class** Solution {
3. **public**:
4. **void** sortColors(vector<**int**>& nums)
5. {
6. **int** n=nums.size();
7. **if**(n<=1) **return**;
8. sort2(nums,0,n-1);
9. }
10. **void** sort2(vector<**int**>& nums,**int** start, **int** end)
11. {
12. **if**(start<end)
13. {
14. **int** p=partition(nums,start,end);
15. sort2(nums,start,p-1);
16. sort2(nums,p+1,end);
17. }
18. }
19. **int** partition(vector<**int**> &nums,**int** start,**int** end)
20. {
21. **int** x=nums[end];
22. **int** i=start-1;  //i始终指向最后一个小于等于x的元素的下标
23. **int** j;
24. **for**(j=start;j<end;j++)  //不判断end位置的值，因为end位置的值就是x
25. {
26. **if**(nums[j]<=x)
27. {
28. i++;
29. swap(nums[i],nums[j]);
30. }
31. }
32. nums[end]=nums[i+1];  //将x和nums[i+1]进行位置交换
33. nums[i+1]=x;  //从而实现x左边都小于等于它，右边都大于它。
34. **return** i+1;   //i+1是第一个大于x的元素的下标
35. }
36. }

快速排序的Partition的思想可以用来处理很多问题：比如将一个数组左边全部是奇数，右边全部是偶数，这样进行划分就是partition的过程。代码如下：

1. **class** Solution {
2. **public**:
3. /\*
4. \* @param nums: an array of integers
5. \* @return: nothing
6. \*/
7. **void** partitionArray(vector<**int**> &nums) {
8. // write your code here
9. **int** i=-1;
10. **int** start=0,end=nums.size()-1;
11. **for**(**int** j=start;j<=end;j++)
12. {
13. **if**(nums[j]%2)
14. {
15. i++;
16. swap(nums[j],nums[i]);
17. }
18. }
19. }
20. };

利用快速排序求解数组的第k大的数：

1. **class** Solution {
2. **public**:
3. /\*
4. \* @param n: An integer
5. \* @param nums: An array
6. \* @return: the Kth largest element
7. \*/
8. **int** kthLargestElement(**int** n, vector<**int**> &nums) {
9. // write your code here
10. **int** m=nums.size();
11. **return** kthLargestHelper(nums,0,m-1,n);
13. }
14. **int** kthLargestHelper(vector<**int**> &nums, **int** start, **int** end, **int** k)
15. {
16. **int** p=partation(nums,start,end);
17. **if**(k-1==p) **return** nums[p];
18. **else** **if**(k-1>p)//说明k在右半边
19. **return** kthLargestHelper(nums, p+1,end,k);
20. **else** **return** kthLargestHelper(nums,start, p-1, k);
21. }
22. **int** partation(vector<**int**> &nums, **int** start, **int** end)
23. {
24. **int** i=start-1;
25. **int** x=nums[end];
26. **for**(**int** j=start;j<end;j++)
27. {
28. **if**(nums[j]>x)
29. {
30. i++;
31. swap(nums[j],nums[i]);
32. }
33. }
34. nums[end]=nums[i+1];
35. nums[i+1]=x;
36. **return** i+1;
37. }
38. };

#### 堆排序代码：

堆排序的基本思想：将待排序的序列构造成为一个大顶堆，此时整个序列的最大值就是堆顶元素，然后将其和末尾元素进行交换，末尾元素就是最大值了，然后将剩下的n-1个元素重新构造成一个堆，这样会得到n个元素的次小值，如此反复执行就能得到一个有序序列。

主要是建立堆的过程比较麻烦，建立堆的过程可以分为对每个非叶子节点的调整过程，先找到第一个非叶子节点，进行堆调整，依次向二叉树上面走，不断进行堆调整。直到堆顶元素，建堆结束得到的堆顶元素就是数组的最大元素。

1. //堆调整过程
2. **void** adjust(**int** a[], **int** start, **int** end)
3. {
4. **int** temp=a[start]; //记住当前节点
5. **int** i=2\*start+1;  //当前节点的第一个左子树节点
6. **while**(i<=end)
7. {
8. //找出start的左右子节点中最大的那个
9. **while** (i+1<=end && a[i+1]>a[i]) i++;
10. //满足堆的定义，不用调整，break
11. **if**(a[i]<=temp) **break**;
12. //最大子节点向上调整, 替换掉其父节点
13. a[start]=a[i];
14. start=i;
15. i=2\*start+1;
16. }
17. a[start]=temp;
18. }
19. //堆排序过程
20. **void** heapsort(**int** a[], **int** len)
21. {
22. //第一次建立最大堆的过程
23. **for**(**int** i=len/2-1; i>=0; i--)
24. adjust(a, i, len-1);
25. //进行堆排序, 调整len-1次，交换第一个元素和当前数组最后一个元素
26. **for** (**int** i=len-1; i>0; i--)
27. {
28. swap(a[i], a[0]);
29. adjust(a, 0, i-1);
30. }
31. }

#### 归并排序代码：

1. **class** Solution
2. {
3. **public**:
4. **void** sortColors(vector<**int**>& nums)
5. {
6. **int** n=nums.size();
7. **if**(n<=1) **return**;
8. process(nums,0,n-1);
9. }
10. **void** process(vector<**int**> &data,**int** start,**int** end)
11. {
12. **if**(start>=end)  **return**;
13. vector<**int**> copy(data);
14. **int** mid=(start+end)/2;
15. process(data,start,mid); //左边有序
16. process(data,mid+1,end); //右边有序
17. **int** p=mid; //指向左边的最后一个位置
18. **int** q=end; //指向右边的最后一个位置
19. **int** k=end; //拷贝数组的最后一个位置
20. **while**(p>=start && q>=mid+1)  //归并排序的主要逻辑，将两边有序的数组合并为一个数组
21. {
22. **if**(data[p]>data[q])
23. {
24. copy[k--]=data[p--];
25. }
26. **else**
27. {
28. copy[k--]=data[q--];
29. }
30. }
31. **while**(p>=start) copy[k--]=data[p--];
32. **while**(q>=mid+1) copy[k--]=data[q--];
33. **for**(**int** i=start;i<=end;i++) //copy是有序的，将它拷贝给data，这样data也是有序的。
34. {
35. data[i]=copy[i];
36. }
37. }

};

### 二分查找及其变形题目

### DFS深度优先遍历类

#### 1. DFS问题的四个母题：

**1）求一个字符串的所有子串**

1. Given a set of characters represented by a String, return a list containing all subsets of the characters.

Assumptions

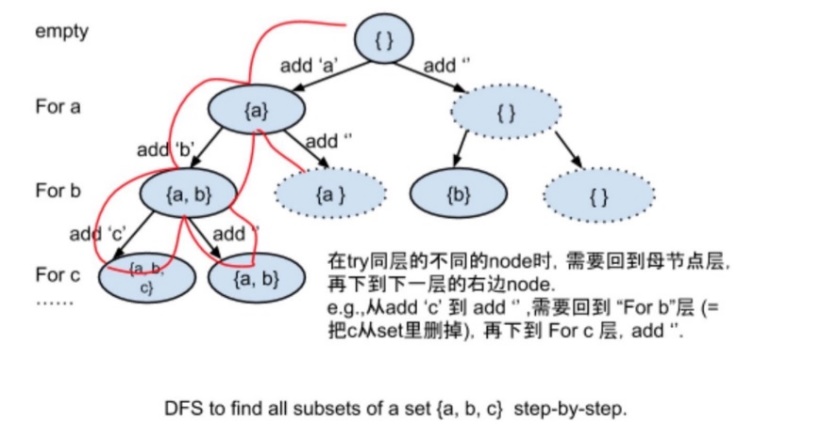
There are no duplicate characters in the original set.

​Examples

Set = "abc", all the subsets are [“”, “a”, “ab”, “abc”, “ac”, “b”, “bc”, “c”]

Set = "", all the subsets are [""]

Set = null, all the subsets are []



答案：

1. **class** Solution
2. {
3. **public**:
4. vector<string> subsets(string &s)
5. {
6. vector<string> res;
7. string tmp;
8. helper(s,0,tmp,res);
9. **return** res;
10. }
11. //index表示层号
12. **void** helper(string &s, **int** index, string & tmp,vector<string>& res)
13. {
14. **if**(index==s.size())
15. {
16. res.push\_back(tmp);
17. **return**;
18. }
19. tmp.push\_back(s[index]);
20. helper(s,index+1,tmp,res);
21. tmp.pop\_back();
22. helper(s,index+1,tmp,res);
23. }
24. };

**2. Given N pairs of parentheses “()”, return a list with all the valid permutations.**

Assumptions

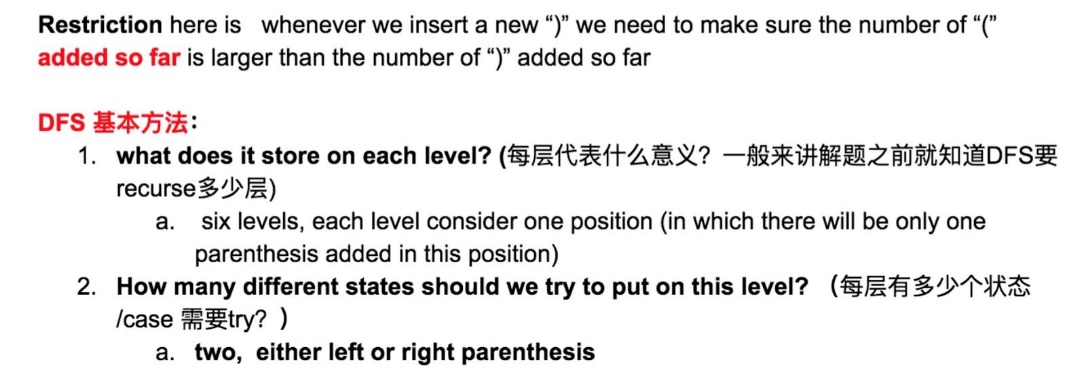
N >= 0

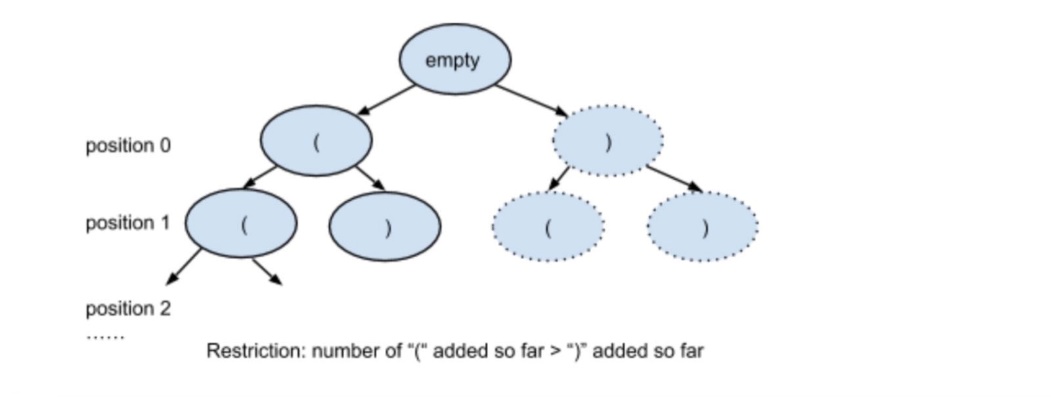
Examples

N = 1, all valid permutations are ["()"]

N = 3, all valid permutations are ["((()))", "(()())", "(())()", "()(())", "()()()"]

N = 0, all valid permutations are [""]





答案：2\*n层，每层两个分支，带剪枝

1. **class** Solution {
2. **public**:
3. vector<string> generateParenthesis(**int** n)
4. {
5. vector<string> res;
6. string s="";
7. helper(s,res,0,0,n);
8. **return** res;
9. }
10. **void** helper(string s, vector<string>& res, **int** left, **int** right,**int** n)
11. {
12. **if**(left==n && right==n) //左右括号都已经用完，则将s放到结果数组中，返回
13. {
14. res.push\_back(s);
15. **return**;
16. }
17. **if**(left<n)//左括号还有，添加一个左边括号，想象成二叉树的左边分支
18. helper(s+'(',res,left+1,right,n);
19. **if**(left>right) //右边括号的个数少于左边括号，则可以添加右边括号进去
20. helper(s+')',res,left,right+1,n);
21. }
22. };

**3. Given a number of different denominations of coins (e.g., 1 cent, 5 cents, 10 cents, 25 cents), get all the possible ways to pay a target number of cents.**

Arguments

coins - an array of positive integers representing the different denominations of coins, there are no duplicate numbers and the numbers are sorted by descending order, eg. {25, 10, 5, 2, 1}

target - a non-negative integer representing the target number of cents, eg. 99

Assumptions

coins is not null and is not empty, all the numbers in coins are positive

target >= 0

You have infinite number of coins for each of the denominations, you can pick any number of the coins.

Return

a list of ways of combinations of coins to sum up to be target.

each way of combinations is represented by list of integer, the number at each index means the number of coins used for the denomination at corresponding index.

Examples

coins = {2, 1}, target = 4, the return should be

[

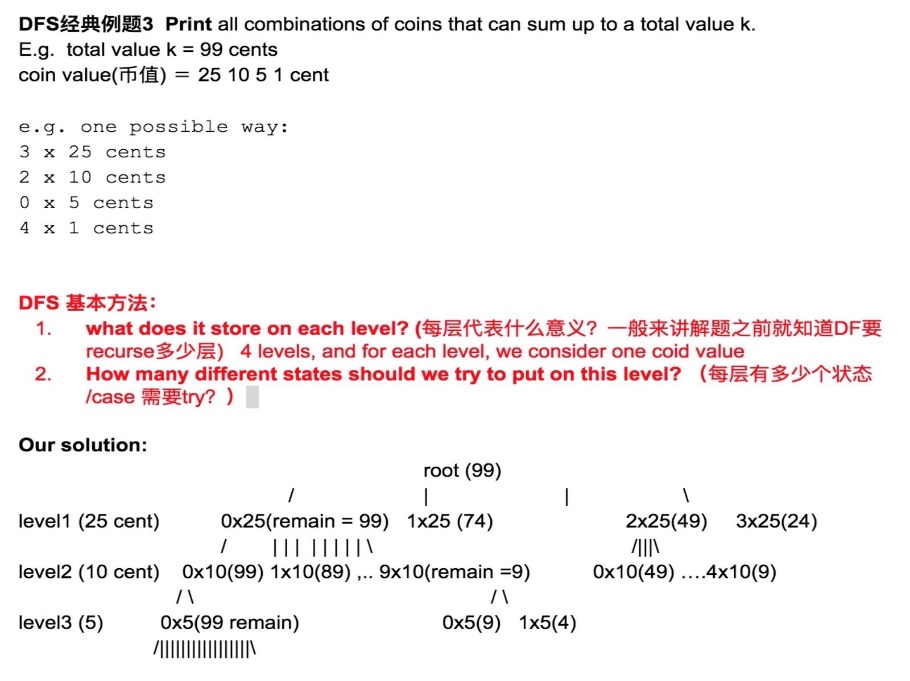
[0, 4], (4 cents can be conducted by 0 2 cents + 4 1 cents)

[1, 2], (4 cents can be conducted by 1 2 cents + 2 1 cents)

[2, 0] (4 cents can be conducted by 2 2 cents + 0 1 cents)

]

答案：这道题的难度在于，两种不同的建立树的方式，时间复杂度不一样。



1. **void** helper(vector<vector<**int**>> &res, vector<**int**> &tmp, **int** target, vector<**int**> & coins, **int** layer)
2. {
3. **if** (layer == coins.size())
4. {
5. **if** (target == 0)
6. res.push\_back(tmp);
7. **return**;
8. }
9. **for** (**int** i = 0; i <= target / coins[layer]; i++)
10. {
11. tmp.push\_back(i);
12. helper(res, tmp, target - i\*coins[layer], coins, layer + 1);
13. tmp.pop\_back();
14. }
15. }
16. vector<vector<**int**>> arrengeCoins(vector<**int**>& coins, **int** target)
17. {
18. vector<vector<**int**>> res;
19. vector<**int**> tmp;
20. helper(res, tmp, target, coins,0);
21. **return** res;
22. }

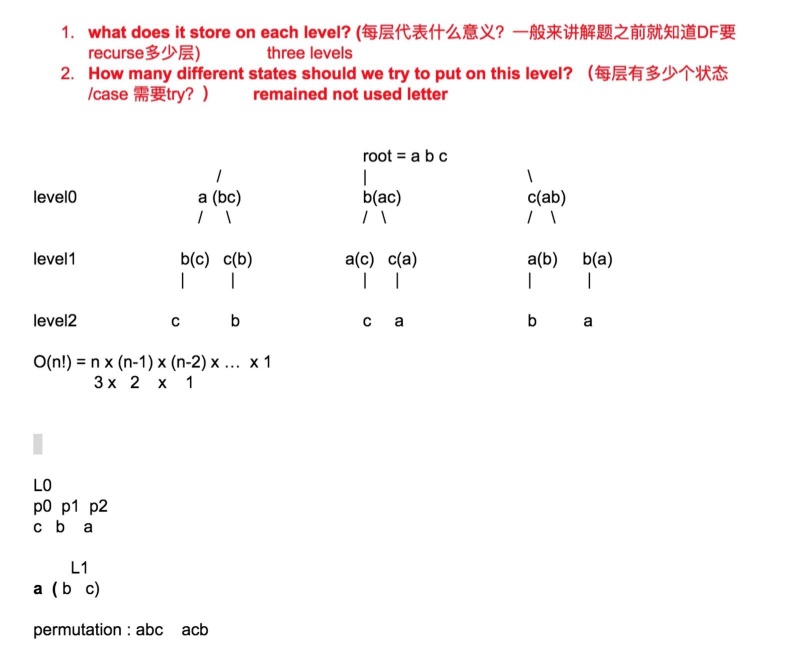
**4. Given a string with no duplicate characters, return a list with all permutations of the characters.**

Examples

Set = “abc”, all permutations are [“abc”, “acb”, “bac”, “bca”, “cab”, “cba”]

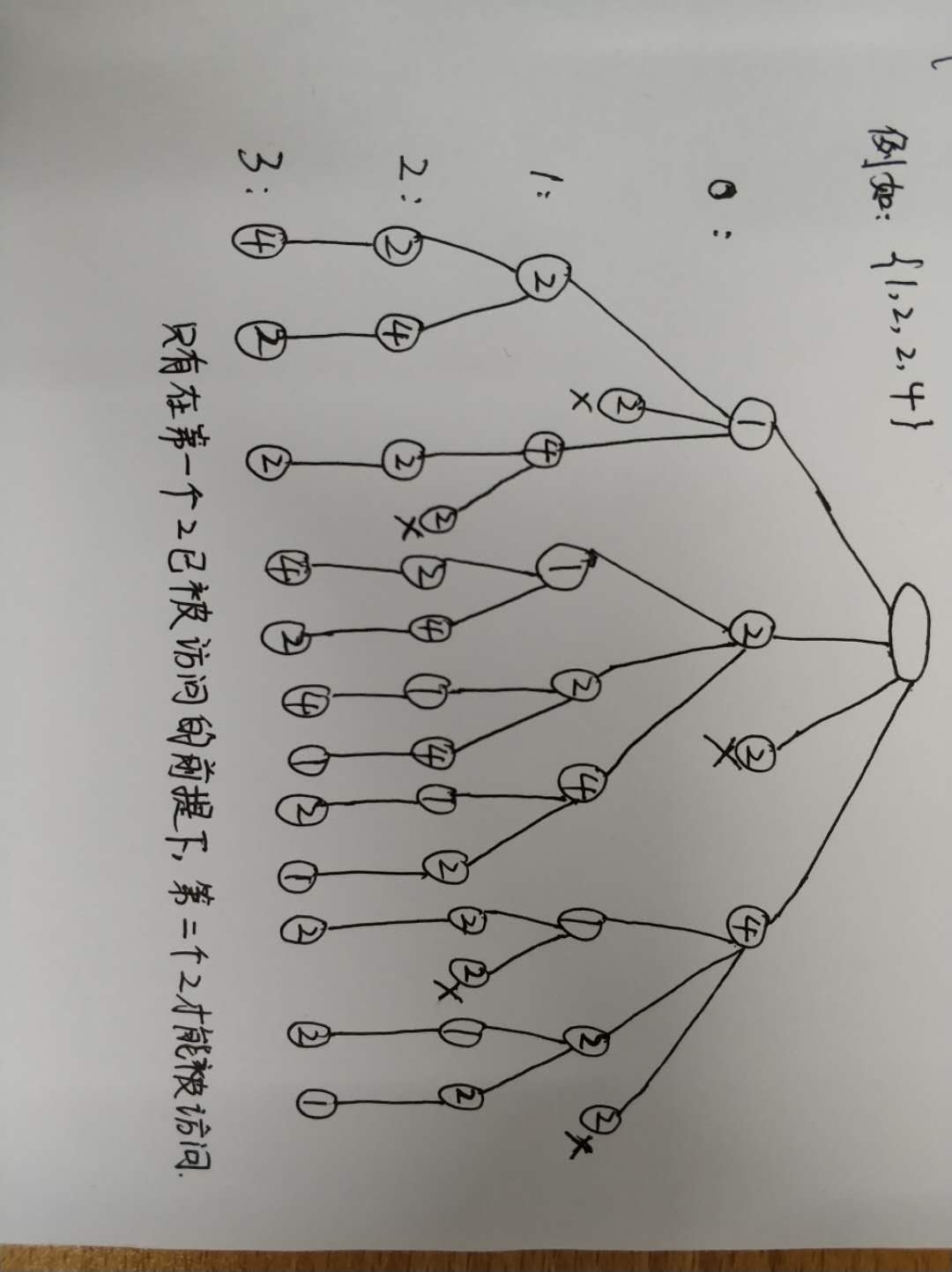
Set = "", all permutations are [""]

Set = null, all permutations are []



1. **class** Solution
2. {
3. **public**:
4. vector<vector<**int**>> permute(vector<**int**>& nums)
5. {
6. vector<vector<**int**>> res;
7. vector<**int**> tmp;
8. helper(nums,res,tmp,0);
9. **return** res;
10. }
11. **void** helper(vector<**int**> &nums,vector<vector<**int**>> &res, vector<**int**> &tmp, **int** layer)
12. {
13. **if**(layer==nums.size())
14. {
15. res.push\_back(tmp);
16. **return**;
17. }
18. **for**(**int** i=layer;i<nums.size();i++)
19. {
20. swap(nums[i],nums[layer]);
21. tmp.push\_back(nums[layer]);
22. helper(nums,res,tmp,layer+1);
23. tmp.pop\_back();
24. swap(nums[i],nums[layer]);
25. }
26. }
27. };

这道题要是有重复，比如{1，2，2}这种情况的全排列，需要去重，那么该怎么做呢？为了避免出现重复，在递归函数中判断前面一个数和当前的数是否相等，如果相等，前面的数必须已经使用了，即对应的visited的值为1，当前的数字才能使用，否则需要跳过，这样就不会产生重复的排列了。这个答案的精髓在于用了一个visited数组来进行剪树枝，既然用了visited来标识该元素是否被访问，其实就不需要进行元素交换了。



代码如下：

1. **class** Solution {
2. **public**:
3. /\*
4. \* @param :  A list of integers
5. \* @return: A list of unique permutations
6. \*/
7. vector<vector<**int**>> permuteUnique(vector<**int**> &nums) {
8. // write your code here
9. vector<vector<**int**>> res;
10. vector<**int**> tmp;
11. vector<**bool**> visited(nums.size(),0);
12. sort(nums.begin(),nums.end());
13. helper(nums,res,tmp,0,visited);
14. **return** res;
15. }
16. **void** helper(vector<**int**> &nums, vector<vector<**int**>> &res, vector<**int**> tmp,**int** layer,vector<**bool**> &visited)
17. {
18. **if**(layer==nums.size())
19. {
20. res.push\_back(tmp);
21. **return**;
22. }
23. **else**
24. {
25. **for**(**int** i=0;i<nums.size();i++)
26. {
28. **if**(visited[i]==0)
29. {
30. **if**(i>0 && nums[i]==nums[i-1] && visited[i-1]==0) **continue**;
31. visited[i]=1;
32. tmp.push\_back(nums[i]);
33. helper(nums,res,tmp,layer+1,visited);
34. tmp.pop\_back();
35. visited[i]=0;
36. }
38. }
39. }
40. }
41. };

#### 2. 其他典型的用dfs做的题目：

1. 头条笔试字符匹配的问题，目标字符串的相似度问题，从100开始减，按照顺序，减去dest字符串两个字母在字典字符串中的位置差，计算得出匹配度最高的分数，比如

Dest: mbh

Source: amhbgyhdc

m和b相隔1，b和h相隔2，那么相似度得分就是100-1-2=97

用dfs做，层数是dest字符串的长度，每层的分支数是指char字符出现在source的不同位置。

1. #include<iostream>
2. #include<vector>
3. #include<string>
4. #include<map>
5. **using** **namespace** std;
6. **void** dfs(**int** &max\_i, map<**char**, vector<**int**>> &mp, **int** layer, string dest, **int** pre,**int** cur)
7. {
8. **if** (layer == dest.size())
9. {
10. **if** (max\_i < cur)
11. max\_i = cur;
12. **return**;
13. }
14. **else**
15. {
16. **for** (auto i : mp[dest[layer]])
17. {
18. **if** (pre == -1)
19. dfs(max\_i, mp, layer + 1, dest, i, cur);
20. **else** **if** (pre != -1 && pre < i)
21. dfs(max\_i, mp, layer + 1, dest, i, cur - (i - pre - 1));
22. **else** **continue**;
23. }
24. }
25. }
26. **int** main()
27. {
28. string dest;
29. string source;
30. cin >> dest;
31. cin >> source;
32. map<**char**, vector<**int**>> mp;
33. **for** (**int** i = 0; i < source.size(); i++)
34. {
35. mp[source[i]].push\_back(i);
36. }
37. **int** max\_i = 0;
38. dfs(max\_i, mp, 0, dest, -1, 100);
39. cout << max\_i << endl;
40. system("pause");
41. **return** 0;
42. }
43. 头条笔试第五题：

描述：一个矩阵，元素有正有负，从最左边一列开始，到最右边为止，每次只能向右上，右边和右下的位置走，走到一个位置，加上该位置的值。但是可以将其中的一值变为它的相反数，目标是使得路径和最大，求出最大的路径和。

分析：dfs来做，参数为矩阵，矩阵的行数和列数，当前行号和列号，当前打分的值和路径中是否有值已经变号了。变号发生在当前位置是负数，正数和0没有必要变号。

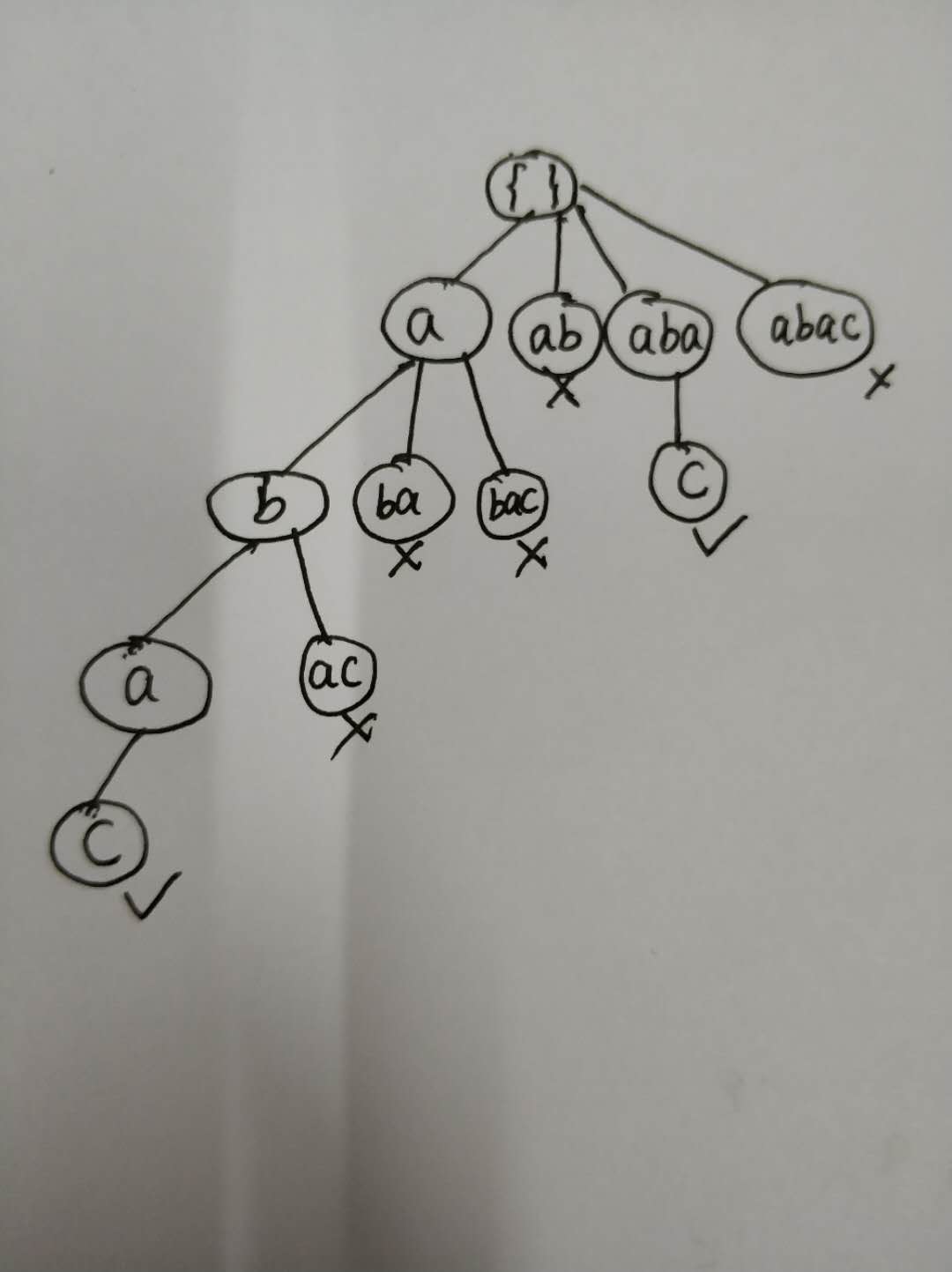
1. #include <iostream>
2. **using** **namespace** std;
3. **int** max\_i = 0;  //最大值设置为全局变量
4. **void** dfs(**int** A[100][100], **int** m, **int** n, **int** i, **int** j, **int** cur, **bool** tag)
5. {
6. //当前位置不合法，结束
7. **if** (j > n - 1 || i > m - 1 || j < 0 || i < 0)
8. {
9. **return**;
10. }
11. //该位置为负数，且之前路径没有数变号
12. **if** (tag==**false** && A[i][j]<0)
13. {
14. cur = cur - A[i][j]; //变号加入到当前值中
15. **if** (j == n - 1) //到了最右边，判断路径和和当前最大值比较，更新路径最大值
16. {
17. **if** (max\_i < cur)
18. max\_i = cur;
19. **return**;
20. }
21. **if** (cur < 0) **return**; //当前路径和小于0，结束，该分支不再进行递归
22. dfs(A, m, n, i + 1, j + 1, cur, **true**); //向右下dfs
23. dfs(A, m, n, i - 1, j + 1, cur, **true**);  //向右上dfs
24. dfs(A, m, n, i, j + 1, cur, **true**);  //向右dfs
25. }
26. //该位置不变号，相同的执行过程，继续递归
27. cur = cur + A[i][j];
28. **if** (j == n - 1)
29. {
30. **if** (max\_i < cur)
31. max\_i = cur;
32. **return**;
33. }
34. **if** (cur < 0) **return**;
35. dfs(A, m, n, i + 1, j + 1, cur, tag);
36. dfs(A, m, n, i - 1, j + 1, cur, tag);
37. dfs(A, m, n, i, j + 1, cur, tag);
38. }
39. **int** main()
40. {
41. **int** m, n;
42. cin >> m >> n;
43. **int** A[100][100];
44. **for** (**int** i = 0; i < m; i++)
45. {
46. **for** (**int** j = 0; j < n; j++)
47. cin >> A[i][j];
48. }
49. **for** (**int** i = 0; i < m; i++)
50. {
51. dfs(A, m, n, i, 0, 0,**false**);
52. }
53. cout << max\_i << endl;
54. system("pause");
55. **return** 0;
56. }

3. Leetcode 上的 126 word ladder II，这个题如果用dfs会出现递归深度很大，然后超时的情况，代码如下，显然会超时，所以不能用dfs做，但是为了体现思想，我们还是写出了dfs的代码。

1. **class** Solution {
2. **public**:
3. vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList)
4. {
5. **int** temp = wordList.size();
6. unordered\_set<string> word(wordList.begin(), wordList.end());
7. word.erase(beginWord);
8. **int** max\_i = wordList.size() + 3; //路径长度最多为把所有的单词遍历一遍
9. vector<string> tmp;
10. vector<vector<string>> res;
11. tmp.push\_back(beginWord);
12. helper(beginWord, endWord, word, max\_i, 1,tmp,res);
13. //cout << max\_i << endl;
14. **for**(auto it=res.begin();it!=res.end();)
15. {
16. **if**((\*it).size()!=max\_i)
17. res.erase(it);
18. **else** it++;
19. }
20. **return** res;
21. }
22. **void** helper(string beginWord, string endWord, unordered\_set<string> &word, **int** &max\_i, **int** cur\_i,vector<string> tmp,vector<vector<string>> &res)
23. {
24. **if** (beginWord == endWord)
25. {
26. **if** (cur\_i < max\_i)
27. max\_i = cur\_i;
28. res.push\_back(tmp);
29. **return**;
30. }
31. **if** (word.size() == 0)
32. {
33. **return**;
34. }
35. **for** (**int** j = 0; j < beginWord.size(); j++)
36. {
37. **for** (**char** i = 'a'; i <= 'z'; i++)
38. {
39. **if** (beginWord[j] == i) **continue**;
40. **char** temp = beginWord[j];
41. beginWord[j] = i;
42. **if** (word.find(beginWord) != word.end())
43. {
44. tmp.push\_back(beginWord);
45. word.erase(beginWord);      //已经访问的单词不再遍历
46. helper(beginWord, endWord, word, max\_i, cur\_i + 1,tmp,res);
47. word.insert(beginWord);    //恢复现场
48. tmp.pop\_back();   //恢复现场
49. }
50. beginWord[j] = temp;
51. }
52. }
53. }
54. };

4. 求一个字符串的所有的回文串组合。

首先明确这个树层数是字符串的长度，因为无论如何单个字符总归是回文串，思路是，从字符串的每个字符出发，向后截取子串，判断其是否是回文串，因此每层的分支数是字符串长度减去当前层数。以字符串abac为例，其DFS图解表示为：



1. **class** Solution {
2. **public**:
3. vector<vector<string>> partition(string &s) {
4. // write your code here
5. vector<vector<string>> res;
6. vector<string> tmp;
7. helper(s,res,tmp,0);
8. **return** res;
9. }
10. **void** helper(string &s,vector<vector<string>> &res,vector<string> tmp,**int** layer)
11. {
12. **if**(layer==s.size())
13. {
14. res.push\_back(tmp);
15. **return**;
16. }
17. **for**(**int** i=layer;i<s.size();i++)
18. {
19. string temp=s.substr(layer,i-layer+1);
20. **if**(!isPalindrome(temp)) **continue**;
21. tmp.push\_back(temp);
22. helper(s,res,tmp,i+1);
23. tmp.pop\_back();
24. }
25. }
26. **bool** isPalindrome(string & str)
27. {
28. **int** n=str.size();
29. **int** start=0,end=n-1;
30. **while**(start<end)
31. {
32. **if**(str[start]!=str[end])
33. **return** **false**;
34. **else**
35. {
36. start++;
37. end--;
38. }
39. }
40. **return** **true**;
41. }
42. };
43. N皇后问题

思路：经典解法是回溯递归，一层一层的向下扫描，需要用到一个pos数组，其中pos[i]表示第i行皇后的位置，初始化为-1.然后从第0开始递归，每一行都依次遍历各个列，判断如果在该位置放置皇后会不会出现冲突，依次类推，当最后一行的皇后放好之后，一种解法就生成了，将其存入结果res中，然后再继续完成搜索所有的情况。

1. **class** Solution {
2. **public**:
3. /\*
4. \* @param n: The number of queens
5. \* @return: All distinct solutions
6. \*/
7. vector<vector<string>> solveNQueens(**int** n) {
8. // write your code here
9. vector<vector<string>> res;
10. vector<**int**> pos(n,-1);  //pos数组保存每一行放置皇后的位置
11. helper(pos,0,res);
12. **return** res;
13. }
14. //放置皇后的辅助函数，DFS
15. **void** helper(vector<**int**> &pos,**int** row,vector<vector<string>> &res)
16. {
17. **int** n=pos.size();
18. **if**(row==n)  //row其实相当于DFS中的当前层数，row==n，得到一个结果
19. {
20. vector<string> tmp(n,string(n,'.'));
21. **for**(**int** i=0;i<n;i++)
22. {
23. tmp[i][pos[i]]='Q';
24. }
25. res.push\_back(tmp);
26. }
27. **else**
28. {
29. //对当前行来说，尝试每一列看能否放置皇后
30. **for**(**int** col=0;col<n;col++)
31. {
32. **if**(isValid(pos,row,col))   //当前位置合法，可以放置皇后
33. {
34. pos[row]=col;  //放置皇后
35. helper(pos,row+1,res);  //继续DFS
36. pos[row]=-1;  //恢复现场
37. }
38. }
39. }
40. }
41. //判断当前位置（row,col）是否合法的函数
42. **bool** isValid(vector<**int**> &pos,**int** row,**int** col)
43. {
44. **for**(**int** i=0;i<row;i++)
45. {
46. **if**(col==pos[i] || abs(col-pos[i])==abs(row-i))
47. **return** **false**;
48. }
49. **return** **true**;
50. }
51. };

### 5. BFS广度优先遍历类

1. DFS是暴力解法，但是其往往会出现递归过深的现象，导致超时，BFS用于搜索最短路径的解，比如求最少步数的解，最少交换次数的解，因为BFS搜索过程中遇到的解一定是离根最近的，所以遇到一个解，一定就是最优解，此时搜索算法可以终止。这个时候不宜用DFS，因为DFS搜索的解不一定是离根最近的，只有全局搜索完毕才能找出所有解中离根最近的解。（当然这个DFS的不足，可以使用迭代加深搜索ID-DFS去弥补）
2. 空间优劣上，DFS是有优势的，DFS不需要保存搜索过程中的状态，而BFS在搜索过程中需要保存搜索过的状态，而且一般情况需要一个队列来记录。
3. DFS适合搜索全部的解，因为要搜索全部的解，那么BFS搜索过程中，遇到离根最近的解，并没有什么用，也必须遍历完整棵搜索树，DFS搜索也会搜索全部，但是相比DFS不用记录过多信息，所以搜素全部解的问题，DFS显然更加合适。

现在用BFS来解决word ladder的问题。

#### 1. Leetcode 127 word ladder I

1. **class** Solution {
2. **public**:
3. **int** ladderLength(string beginWord, string endWord, vector<string>& wordList)
4. {
5. **int** n=beginWord.size();
6. unordered\_set<string> word(wordList.begin(),wordList.end());
7. **if**(beginWord.size()!=endWord.size())  **return** 0;
8. **int** step=1;
9. **bool** exist=**false**;
10. queue<string> q;
11. q.push(beginWord);
12. q.push("1");  //用”1”来区分层
13. **while**(!q.empty())
14. {
15. string temp=q.front();
16. q.pop();
17. **if**(temp==endWord)
18. {
19. exist=**true**;
20. **break**;
21. }
22. **if**(temp=="1")  //遇到一层结束，则加一
23. {
24. step++;
25. **continue**;
26. }
27. **for**(**int** i=0;i<n;i++)
28. {
29. **for**(**char** j='a';j<='z';j++)
30. {
31. string t=temp;
32. **if**(t[i]!=j) t[i]=j;
33. **if**(word.find(t)!=word.end())
34. {
35. word.erase(t);
36. q.push(t);
37. }
38. }
39. }
40. **if**(!q.empty() && q.front()=="1")
41. q.push("1");
42. }
43. **if**(exist) **return** step;
44. **return** 0;
45. }
46. };

#### 2. Leetcode 126题word ladder II

这个题的难点在于，如何记住路径，记住一个路径也很简单，但是要记住所有的最短路径，这个需要动点脑筋。

BFS+DFS:BFS用于搜索最短路径，在搜索的过程中，记住每个节点的前驱结点，前驱结点会有多个，但是访问过的点已经被删除了，也只能记住一个前驱结点，这也就是为什么输出一条最优路径简单而输出所有的最优路径很难的原因，用BFS层序遍历。

DFS用于输出所有的最优路径。

1. **class** Solution {
2. **public**:
3. vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList)
4. {
5. unordered\_set<string> word(wordList.begin(), wordList.end());
6. unordered\_set<string> copyword(word);
7. **int** n = beginWord.size();
8. vector<vector<string>> res;
9. vector<string> tmp;
10. queue<string> q;
11. q.push(beginWord);
12. q.push("1");
13. **bool** exist = **false**;
14. **int** step = 1;
15. unordered\_map<string, vector<string>> mp;
16. **while** (!q.empty())
17. {
18. string temp = q.front();
19. q.pop();
20. **if** (temp == endWord)
21. {
22. exist = **true**;
23. **break**;
24. }
25. **if** (temp == "1")  //遇到一层结束，则加一
26. {
27. step++;
28. **continue**;
29. }
30. string t = temp;
31. **for** (**int** i = 0; i<n; i++)
32. {
33. **char** x = t[i];
34. **for** (**char** j = 'a'; j <= 'z'; j++)
35. {
37. **if** (t[i] != j) t[i] = j;
38. **if** (word.find(t) != word.end())
39. {
40. word.erase(t);
41. q.push(t);
42. queue<string> cop(q);
43. mp[t].push\_back(temp);
44. string tmp = cop.front();
45. **while** (tmp != "1")
46. {
47. cop.pop();
48. **if** (difference(t, tmp) == 1)
49. mp[t].push\_back(tmp);
50. tmp = cop.front();
51. }
52. }
53. t[i] = x;
54. }
55. }
56. **if** (!q.empty() && q.front() == "1")
57. q.push("1");
58. }
59. **if** (exist)
60. {
61. tmp.push\_back(endWord);
62. helper(tmp, res, mp, endWord, beginWord);
63. }
64. **return** res;
65. }
66. **int** difference(string s1, string s2)
67. {
68. **int** n1 = s1.size();
69. **int** n2 = s2.size();
70. **int** res = 0;
71. **if** (n1 != n2) **return** -1;
72. **for** (**int** i = 0; i < n1; i++)
73. {
74. **if** (s1[i] != s2[i]) res++;
75. }
76. **return** res;
77. }
78. **void** helper(vector<string> tmp, vector<vector<string>> &res, unordered\_map<string, vector<string>> mp,string endWord, string beginWord)
79. {
80. **if**(mp[endWord][0]==beginWord)
81. {
82. tmp.push\_back(beginWord);
83. reverse(tmp.begin(),tmp.end());
84. res.push\_back(tmp);
85. **return**;
86. }
87. **for**(auto i:mp[endWord])
88. {
89. tmp.push\_back(i);
90. helper(tmp,res,mp,i,beginWord);
91. tmp.pop\_back();
92. }
93. }
94. };

### DP动态规划

#### 1. 动态规划的基本思路

首先明确无法用动态规划的情况有哪些？

第一种，无法提前知道拓扑排序，也就是无法知道填充顺序。

第二种，没有重复计算，dependency graph里node的入度都是1，没必要用dp。

第三种，依赖关系构成环。

知道拓扑关系，就知道填充数组或者矩阵的顺序是什么样的，到底是从上到下还是从右向左还是斜着对角线。

动态规划问题的步骤：

1. definition: dp[i] 表示是f(i) 是多少 =**>** 存的是value
2. induction rule: dp[i] = dp[i - 1] + dp[i - 2] =**>** cose: O(1)
3. base case: dp[0]  dp[1]
4. filling order: 从左往右
5. result: max(dp[i])
6. optimized on space or time:

#### 2. Leetcode 97题：给定s1,s2，求s3是否是由s1和s2形成的，组合交错形成也是可以的。

思路是，二维动态规划，dp[i][j]表示s1取前i位，s2取前j位，是否能构成s3的前i+j位。

递推公式如下：

1. **if**(i==0 && j==0) dp[i][j]=**true**;
2. **else** **if**(i>0 && s1[i-1]==s3[i+j-1] && dp[i-1][j])
3. dp[i][j]=**true**;
4. **else** **if**(j>0 && s2[j-1]==s3[i+j-1] && dp[i][j-1])
5. dp[i][j]=**true**;

#### 3. Leetcode 72题：编辑距离：给定两个单词w1和w2，求最少需要操作多少次，才能将w1变成w2。操作方式包括插入，删除，替换。

也是二维动态规划解决的问题，递推公式是

1. 如果w1[i-1]==w2[j-1]: dp[i][j]=dp[i-1][j-1]；
2. 如果w1[i-1]!=w2[j-1]：dp[i][j]= min(dp[i-1][j-1]+1,min(dp[i-1][j]+1,dp[i][j-1]+1));

分别对应的是，替换，删除和插入。

#### **4. Leetcode** 120题：三角形

给定一个三角形，找出路径和的最小值，从顶到底部，每一步只能找与它相邻位置的路径。

这个题从底部到顶走比较好，最后一行的最短路径等于它所在位置本身。递推公式是dp[i][j]=min(dp[i+1][j+1],dp[i+1][j])+t[i][j]; 返回的是dp[0][0];

1. **int** minimumTotal(vector<vector<**int**>>& t)
2. {
3. **int** n=t.size();
4. vector<vector<**int**>> dp(n,vector<**int**>(n,0));
5. dp[n-1]=t[n-1];//最后一行的最短路径等于它所在位置本身。
6. **for**(**int** i=n-2;i>=0;i--)
7. {
8. **for**(**int** j=0;j<t[i].size();j++)
9. {
10. dp[i][j]=min(dp[i+1][j+1],dp[i+1][j])+t[i][j];
11. }
12. }
13. **return** dp[0][0];

#### 5. Maximum Product Subarray

这个题的解法：需要维护的当前最大值和当前最小值，都是在dp\_min[i-1] \* A[i]，dp\_max[i] \* A[i]，和A[i]这三者里面取一即可。状态的跳转是需要前一个位置的最大值，最小值和当前位置的值。

1. **class** Solution
2. {
3. **public**:
4. **int** maxProduct(vector<**int**>& nums)
5. {
6. **int** pMax=1,nMax=1,m=INT\_MIN;
7. **int** n=nums.size();
8. **for**(**int** i=0;i<n;i++)
9. {
10. **if**(nums[i]<0) swap(pMax,nMax);
11. pMax=max(pMax\*nums[i],nums[i]);
12. nMax=min(nMax\*nums[i],nums[i]);
13. m=max(m,pMax);
14. }
15. **return** m;
16. }
17. };

### 5. 贪心类题目

贪心类题目的特征：做出每一步贪心决策都无法改变，因为贪心策略是**由上一步的最优解推倒下一步的最优解，而上一步之前的最优解不做保留**。并且，每一步的最优解一定包含上一步的最优解。

动态规划算法中，全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有最优解。动态规划的关键是状态转移方程，即如何由已经求出的局部最优解来推倒全局最优解，也就是说，把一个复杂的问题分解成一块一块的小问题，每一个问题得到最优解，再从这些最优解中获得更优的答案。

#### 1. 买卖股票的最佳时机

#### 2. 最大子数组差

给定一个整数数组，找出两个不重叠的子数组A和B，使两个子数组和的差的绝对值|SUM(A) - SUM(B)|最大。

返回这个最大的差值。

样例

给出数组[1, 2, -3, 1]，返回 6

挑战: 时间复杂度为O(n)，空间复杂度为O(n)

思路：这个题采用的是贪心算法，求出数组中每个元素（含）以左的最大子数组和和每个元素（含）以右的最小子数组和，遍历求出最大子数组差，再将数组反序，进行相同的过程，求出正序和反序得到的最大子数组差的最大值。

1. **class** Solution {
2. **public**:
3. /\*\*
4. \* @param nums: A list of integers
5. \* @return: An integer indicate the value of maximum difference between two substrings
6. \*/
7. **int** maxDiffSubArrays(vector<**int**> &nums) {
8. // write your code here
9. **int** l1=helper(nums);
10. reverse(nums.begin(),nums.end());
11. **int** l2=helper(nums);
12. **return** max(l1,l2);
13. }
14. **int** helper(vector<**int**> &nums)
15. {
16. **int** n= nums.size();
17. vector<**int**> leftmax(n,0);
18. vector<**int**> rightmin(n,0);
19. **int** mx=INT\_MIN;
20. **int** sum=0;
21. **for**(**int** i=0;i<n;i++)
22. {
23. sum+=nums[i];
24. mx=max(mx,sum);
25. **if**(sum<0)
26. sum=0;
27. leftmax[i]=mx;
28. }
29. mx=INT\_MAX;
30. sum=0;
31. **for**(**int** i=n-1;i>=0;i--)
32. {
33. sum+=nums[i];
34. mx=min(mx,sum);
35. **if**(sum>0)
36. sum=0;
37. rightmin[i]=mx;
38. }
39. **int** res=0;
40. **for**(**int** i=1;i<n;i++)
41. {
42. res=max(res,abs(leftmax[i-1]-rightmin[i]));
43. }
44. **return** res;
45. }
46. };

#### **3. 删除数字：**

给出一个字符串A，表示一个n位正整数，删除其中的k位数字，使得剩余的数字仍然按照原来的顺序排列产生一个新的正整数，找到删除k个数字之后的最小正整数。N<=240，k<=N；

样例: 给出一个字符串代表的正整数 A 和一个整数 k, 其中 A = 178542, k = 4

返回一个字符串 "12"。

思路是：每次删除最先出现的最大数字，也就是每次删除一个局部最大值，删除k次，剩下的数字按照顺序就构成了删除k个数字之后的最小正整数。

1. **class** Solution {
2. **public**:
3. /\*\*
4. \* @param A: A positive integer which has N digits, A is a string
5. \* @param k: Remove k digits
6. \* @return: A string
7. \*/
8. string DeleteDigits(string &A, **int** k) {
9. // write your code here
10. **int** n=A.size();
11. **while**(k--)
12. {
13. **int** i=0;
14. **while**(i<n-1 && A[i]<=A[i+1])
15. {
16. i++;
17. }
18. A=A.substr(0,i)+A.substr(i+1);
19. }
20. **int** i=0;
21. **while**(1)
22. {
23. **if**(A[i]=='0') i++;
24. **else** **break**;
25. }
26. **return** A.substr(i);
27. }
28. };

## 四．操作系统

### 1. 堆和栈的区别与联系

答：栈区：由编译器自动分配释放，存放函数参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆区：一般由程序员分配释放，若程序员不释放，程序结束之后可能会由OS回收。

联系与区别：

（1）申请方式上不同，堆是由程序员自己申请并指明大小，在c中malloc函数，如p1 = (char \*)malloc(10);栈由系统自动分配，比如在声明一个局部变量的时候 int b；

（2）申请之后系统的响应：栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。堆：首先，操作系统有一个记录空闲内存地址的链表，当系统收到程序申请的时候，会遍历这个链表，寻找第一个空间大于所申请空间的堆节点，然后将该结点从空闲节点链表中删除，并将该结点的内存分配给程序，对于大多数系统来说，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete才能正确地释放本内存空间。另外，由于找到的堆节点大小不一定正好等于申请的大小，系统会自动地把多余的那部分重新放回空闲链表。

（3）申请大小的限制：栈：在windows系统中，栈是向低地址扩展的数据结构。是一片连续存储的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先确定好的，大小大约是2M。编译时是个常数，如果申请的空间超过剩余空间会提示overflow。堆是向高地址扩展的数据结构，是不连续的内存区域，系统用链表存储空闲的内存地址，自然是不连续的，而链表的遍历方向也是从低地址到高地址。从堆中获得的空间比较大，也比较灵活。

（4）申请效率：栈是由系统自动分配的，速度较快，但是程序是无法控制的。堆是由new分配的内存，一般速度慢，而且容易产生内存碎片。

### 2. 线程与进程的区别？

答：（1）进程是程序在某一个数据集合上的一次运行活动。是系统进行资源调度和分配的独立单位。

（2）线程是进程的实体，是CPU调度和分配的基本单位，它是比进程更小的可以独立运行的基本单位。线程基本上不拥有资源，它只拥有一些在运行时必不可少的资源（比如程序计数器、堆栈、一组寄存器），但是它可以与通属于一个进程的其它线程共享进程所拥有的全部资源。

（3）一个线程可以创建和撤销另一个线程，同一个进程的所有线程可以并发执行。

（4）主要区别在于他们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃之后，在保护模式下其它进程不受影响；而线程没有独立的地址空间，一个线程死掉整个进程死掉，所以多进程的程序比多线程的程序更加健壮，但是在切换进程时，耗费的资源比较大，效率更差。但是对于一些要求同时进行并且又要共享某些变量的并发操作，只能用多线程而不能用多进程。

### 3. 线程同步的方式有哪些？

答：线程同步是指多个线程同时访问某个资源的时候，采用一系列机制保证同时最多只能有一个线程访问该资源。线程同步是多线程中必须考虑和解决的问题，因为很有可能会发生多个线程同时访问（主要是写操作）同一个资源。如果不进行线程同步，很可能会引起数据混乱，造成线程死锁等。

（1）临界区：通过对多线程的串行化来访问公共资源或者一段代码，速度快，适合控制数据访问。

（2）互斥量：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以可以保证公共资源不会同时被多个线程访问。

（3）信号量：它允许多个线程同一时刻访问同一资源，但是会限制同一时刻访问此资源的最大线程数目。信号量对象对线程的同步方式与前面几种方法不同，它允许多个线程同时使用共享资源，与操作系统的PV操作类似。

（4）事件（信号）：通过通知操作的方式保持多线程的同步，还可以方便地实现多线程的优先级比较操作。

同步与互斥的关系与区别？

同步体现一种协助性，互斥体现一种排他性。同步是在互斥的基础上实现访问者对资源的有序访问。互斥是指某一资源某个时刻允许一个访问者对其访问，具有唯一性和排他性。互斥无法限制访问者对资源的访问顺序，访问时无序的。

线程同步一般有三种常见的机制：

**互斥量、读写锁和条件变量**

读写锁：读写锁是因为有3种状态, 所以可以有更高的并行性，一次只有一个线程可以占有写模式的读写锁, 但是可以有多个线程同时占有读模式的读写锁。正是因为这个特性，当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。当读写锁在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是如果线程希望以写模式对此锁进行加锁，它必须阻塞直到所有线程释放锁。

读写锁适合于对数据结构的读次数比写次数多得多的情况. 因为, 读模式锁定时可以共享, 以写模式锁住时意味着独占, 所以读写锁又叫共享-独占锁。

条件变量

### 4. 进程间通信的方式有哪些？特点是什么？ 管道的实现方式，是否是全双工？

答：（1）管道：半双工通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间调用。进程的亲缘关系是指父子进程关系。

（2）有名管道：半双工的通信方式，允许无亲缘关系的进程通信。

（3）信号量：信号量是个计数器，可以用来控制多个进程对共享资源的访问。它经常作为一种锁机制，防止某个进程访问一个正被其他进程访问的资源，主要作为进程间或者同一进程中不同线程之间的同步手段。

（4）消息队列：消息队列是消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受到限制等缺点。

（5）信号：是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

（6）共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但是多个进程都可以访问。共享内存是最快的IPC方式，它往往与其他通信机制，比如信号量等配合使用，来实现进程间的同步和通信。

（7）套接字：套接字也是一种进程间通信的机制，与其他通信机制不同的是，它可以用于不同主机的进程间通信。

### 5. 什么是缓冲区溢出？有哪些危害？原因是什么？

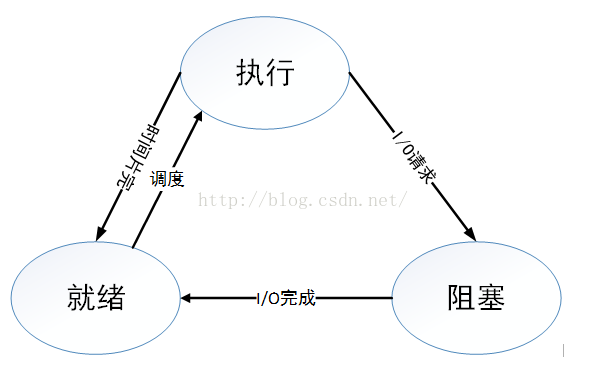
答：（1）缓冲区溢出是指当计算机向缓冲区填充数据的时候超过了缓冲区本身的容量，溢出的数据覆盖在了原有的合法数据上。

（2）最大的危害是造成堆栈溢出，入侵者利用堆栈溢出在函数返回时改变返回程序的地址，让其跳转到任意地址导致程序崩溃而拒绝服务，或者让我跳转到一段恶意代码上。通常往程序的缓冲区写超过其长度的内容，造成缓冲区溢出，从而破坏程序的堆栈使其转而执行其他指令，以达到攻击的目的。

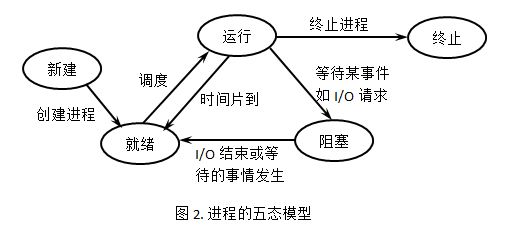
（3）造成缓冲区溢出的主要原因是没有仔细检查用户输入的参数。

### 6. 进程有哪些状态？如何在linux下查询一个进程的状态？

答：（1）三种状态



（2）五态模型



（3）linux下查询一个进程的状态是 ps –ef | grep 进程名

ps aux

### 7. 什么是死锁？死锁产生的条件是什么？死锁处理的基本策略和基本方法？

答：（1）死锁就是两个或者多个并发进程中，如果每个进程持有某种资源而又等待别的进程释放它或者它们现在保持着的资源，在没有改变这种状态之前都不能向前推进，就称这组进程产生了死锁。两个或多个进程无限期阻塞、相互等待的一种状态。

（2）死锁的原因：系统资源不足 或者 进程推进顺序非法

（3）产生死锁必要条件：互斥、不可抢占、占有等待、环形等待

（4）死锁避免算法：银行家算法，该方法允许进程动态地申请资源，系统在进行资源分配之前，先计算资源分配的安全性。若此次分配不会导致系统从安全状态向不安全状态转化，那么可以将资源分配给进程，否则不分配资源，进程必须阻塞等待。从而避免死锁的发生。

（5）死锁的处理机制：鸵鸟策略、预防策略、避免策略、检测与恢复策略

（6）死锁解除：1. 强制性从系统中撤销一个或者多个死锁进程以断开循环等待链，并回收分配给终止进程的全部资源供剩下的进程使用。

2.使用一个有效的挂起和解除机构来挂起一些死锁进程，其实质是从被挂起进程那里抢占资源以解除死锁。

### 8. 分页和分段有什么区别？

答：（1）页是信息的物理单位，分页是实现离散分配的方式，以消减内存的外零头，提高内存利用率。分页只是出于系统管理的需要，而不是用户的需要。

（2）段是信息的逻辑单位，它含有一组意义相对完整的信息。分段的作用是为了更好地满足用户的需要。

（3）页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面。段的长度却不固定，决定于用户所编写的程序，通常由编辑程序在对源代码进行编辑时，根据信息的性质来划分的。

（4）分页的作业地址空间是一维的，即单一的线性空间，程序员只需利用一个记忆符，即可表示这一地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需要给出段名，又需要给出段内地址。

段页式系统中，作业的地址结构包含三部分的内容：段号 页号 页内位移量。

程序员按照分段系统的地址结构将地址分为段号与段内位移量，地址变换机构将段内位移量分解为页号和页内位移量。

为实现段页式存储管理，系统应为每个进程设置一个段表，包括每段的段号，该段的页表始址和页表长度。每个段有自己的页表，记录段中的每一页的页号和存放在主存中的物理块号。

### 9. 说一说进程同步的机制有哪些？

答：进程同步的主要任务是：对多个相关进程在执行次序上进行协调，以使得并发执行的诸多进程之间能够有效地共享资源和相互合作，以使程序的执行具有可再现性。同步机制遵循的原则：（1）空闲让进；（2）忙则等待（保证对临界区的互斥访问）；（3）有限等待（有限代表有限的时间，避免死等）；（4）让权等待（当进程不能进入自己的临界区时，应当释放处理机，以免陷入忙等状态）。

临界区、互斥区、事件、信号量四种方式

### 10. linux常用的命令

（1）grep的用法，找出某个文件中某个关键字的位置

grep match\_pattern file\_name

grep "match\_pattern" file\_name

统计文件或者文本中包含匹配字符串的行数 -c 选项：

grep -c "text" file\_name

输出包含匹配字符串的行数 -n 选项：

grep "text" -n file\_1 file\_2

多级目录中对文本进行递归搜索

grep "text" . -r -n

# .表示当前目录。

（2）查看当前进程：ps

执行退出：exit ; 查看当前路径： pwd ;

（3）清屏 : clear 退出当前命令： ctrl+C 执行睡眠： ctrl+z 挂起当前进程 fg恢复后台；

（4）linux下的进程有哪些状态？在ps显示出来的信息中，分别用什么符号表示的？

答：查看进程状态的命令是 ps -aux

linux是一个多用户，多任务的系统，可以同时运行多个用户的多个程序，就必然产生很多进程，每个进程会有不同的状态。

1. R (TASK\_RUNNNING)，可执行状态。只有在该状态的进程才可能在CPU上运行。而同一时刻会有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。
2. S(TASK\_INTERRUPTIBLE),可中断的睡眠状态。处于这个状态的进程因为等待某个事件的发生（比如等待socket连接、等待信号量），而被挂起，这些进程的进程控制块被放进对应事件的等待队列中。当这些事件发生时（由外部中断触发或者由其他进程触发）,对应的等待队列中的一个或者多个进程将被唤醒。
3. D(TASK\_UNINTERRUPTIBLE)，不可中断的睡眠状态。该进程不能响应异步信号，存在的意义在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程。
4. T(TASK\_STOPPED OR TASK\_TRACED),暂停状态或者跟踪状态。向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK\_STOPPED状态（SIGSTOP和SIGKILL信号一样，是非常强制的。不允许用户进程通过signal系列的系统调用重新设置对应的信号处理函数）。而TASK\_TRACED状态相当于在TASK\_STOPPED之上多了一层保护，处于TASK\_TRACED状态的进程不能响应SIGCONT信号而被唤醒。
5. Z(TASK\_DEAD-EXIT\_ZOOBIE),退出状态，进程成为僵尸进程。进程在退出状态中处于TASK\_DEAD状态。在这个退出过程中，进程占有的所有资源将被回收，除了task\_struct结构（以及少数资源）以外。于是进程就只剩下task\_struct这么个空壳，故称为僵尸。之所以保留task\_struct，是因为task\_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。
6. X (TASK\_DEAD - EXIT\_DEAD)，退出状态，进程即将被销毁。

进程自从被创建以来，状态可能发生一系列变化，直到进程退出，而尽管进程状态有好几种，但是进程状态的变迁却只有两个方向—从TASK\_RUNNING状态变为非TASK\_RUNNING状态、或者从非TASK\_RUNNING状态变为TASK\_RUNNING状态。

### 11. 进程调度算法有哪些？具体是如何实现的？

答：（1）先来先服务（FCFS）在所有调度算法中，最简单的是非抢占式的FCFS算法。

算法原理：进程按照它们请求CPU的顺序使用CPU。就想你买东西去排队，谁第一个排，谁就先被执行，在它执行的过程中，不会中断它。

算法优点：易于理解且实现简单，只需要一个队列，且相当公平。

算法缺点：比较有利于长进程，不利于段进程，有利于CPU繁忙的进程，不利于I/0繁忙进程。

(2) 最短作业优先（SJF, shortest job first）该算法的目标是减少平均周转时间。

算法原理：对预计执行时间短的进程优先分派处理机，通常后来的段进程不抢先正在执行的进程。

算法优点：相比于先来先服务，该算法可以改善平均周转时间和平均带权周转时间，缩短进程的等待时间，提高系统的吞吐量。

算法缺点：对长进程非常不利，可能长时间得不到执行，且未能依据进程的紧迫程度来划分执行的优先级，以及难以准确估计进程的执行时间，从而影响调度性能。

(3)最高响应比优先（HRRN，Highest Response Ratio Next)

FCFS方式只考虑每个作业的等待时间而未考虑执行时间的长短，而SJF方式只考虑执行时间而未考虑等待时间的长短。该方法作为两者的折中。

算法原理：响应比R定义如下： R =(W+T)/T = 1+W/T

其中T为该作业估计需要的执行时间，W为作业在后备状态队列中的等待时间。每当要进行作业调度时，系统计算每个作业的响应比，选择其中R最大者投入执行。

算法优点：由于长作业也有机会投入运行，在同一时间内处理的作业数显然要少于SJF，从而采用HRRN方式时其吞吐量将小于SJF法时的吞吐量。

算法缺点：由于每次调度前要计算响应比，系统开销也要相应地增加。

(4)时间片轮转算法（RR, Round-Robin）该算法采用剥夺策略。

算法原理：让就绪进程以FCFS的方式按照时间片轮流使用CPU的调度方式，每次调度时将CPU 分派给队首进程，让其执行一个时间片，时间片的长度从几个ms 到几百ms。在一个时间片结束时，发生时钟中断，调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程，进程可以未使用完一个时间片，就出让CPU（如阻塞）。

算法优点: 简单易行，平均响应时间短。

缺点：不利于处理紧急作业，时间片的大小对系统性能的影响很大，因此时间片的大小应选择恰当。

(5)多级反馈队列（Multilevel Feedback Queue) 多级反馈队列调度算法是一种CPU处理机调度算法，UNIX就采用这种调度算法。

描述：1.进程进入待调度的队列等待时，首先进入优先级最高的Q1等待。

2.首先调度优先级高的队列中的进程，若优先级中队列中已经没有调度的进程，则调度次优先级队列中的进程，例如：Q1,Q2,Q3三个队列，只有在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。

3.对于同一个队列中的各个进程，按照时间片轮转调度，比如Q1队列的时间片是N，那么Q1中的作业在经历了N个时间片后还没有完成则进入Q2队列中等待，若Q2的时间片用完之后作业还没完成，一直进入下一级队列，直至完成。

4.在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片之后，CPU马上分配给新到达的作业(抢占式)。

在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需的处理时间时，便能更好地满足各种类型用户的需要。

### 12. 并发编程与生产者消费者模型

答：在并发编程中使用生产者消费者模式将能解决绝大多数并发问题，该模式通过平衡生产进程和消费进程的工作能力来提高程序的整体处理数据的速度。

**为什么要使用生产者和消费者模式？**

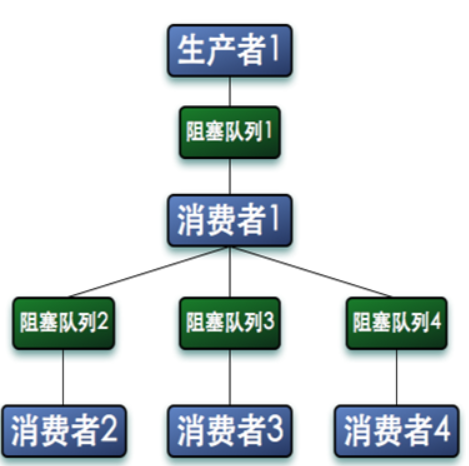
在线程的世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者必须等待消费者处理结束，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这种生产消费能力不均衡的问题，所以便有了生产者和消费者的模式。

**什么是生产者消费者模式？**

生产者和消费者之间通过一个阻塞队列进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不着生产者要数据，而是直接从阻塞队列中取得，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者进行解耦的。其实大多数的设计模式都会找一个第三者进行解耦，工厂模式的第三者是工厂类，模板模式的第三者是模板类。

**多生产者和多消费的情况**

多核时代，多线程并发处理速度比单线程处理速度更快，所以我们可以使用多个线程来生产数据，同样可以使用多个消费线程来消费数据。更加复杂的情况是，消费者消费的数据，有可能需要进一步处理，于是消费者处理完数据之后，它又要作为生产者把数据放在新的队列中，交给其他消费者继续处理。



我们在一个长链接服务器中使用了这种模式，生产者1负责将所有的客户端发送的消息存放在阻塞队列1中，消费者1从队列里读消息，然后通过消息ID进行哈希得到N个队列中的1个，然后根据编号将消息存放到不同的队列，每个阻塞队列会分配一个线程来消费阻塞队列里的数据。如果消费者2无法消费消息，就将消息再抛回到阻塞队列1中，交给其他的消费者处理。

### 13. 信号机制，常见的有哪些信号？信号量是什么？

答：信号（软中断信号）用来通知进程发生了异步事件。

原理：一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是进程间通信机制中唯一的异步通信机制，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号什么时候到达。进程之间可以互相通过系统调用哪个kill发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。信号机制除了基本通知功能之外，还可以传递附加信息。

分类：从两个不同的分类角度对信号进行：

可靠性方面：可靠信号与不可靠信号；

与时间的关系上：实时信号与非实时信号。

信号量机制：

Linux内核的信号量用来操作系统进程间同步访问共享资源。原理：信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初始值为1就变成了互斥锁，即同时只能有一个任务可以访问信号量保护的共享资源。

一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减掉1，若当前信号量的值为负数，表示无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可以用；若当前该信号量的值为非负数，表示可以获得该信号量，因而可以立即访问被该信号量保护的共享资源。

当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

## 五．计算机网络

### 1. TCP和UDP的区别？

答：（1）TCP是有连接的，两台主机在进行数据交互之前必须通过三次握手建立连接；而UDP是无连接的，没有建立连接的过程。

（2）TCP是可靠的传输，TCP协议通过确认和重传机制来保证数据传输的可靠性；而UDP是不可靠的传输。

（3）TCP提供了拥塞控制、滑动窗口等机制来保证传输的质量，而UDP没有。

（4）TCP是面向字节流的服务，将数据看做是无结构的字节流传输，当应用程序交给TCP的数据长度过长，超过MSS的时候，TCP就会对数据进行分段，因此TCP的数据是没有边界的。UDP是面向报文的，无论应用程序交给UDP层多长的报文，UDP都不会对数据包进行任何拆分处理，因此UDP保存了应用层数据的边界。

**这里需要注意，**MSS是TCP数据包每次能够传输的最大数据分段，不包括TCP首部，而TCP窗口是为了控制传输过程中的速度，传输速率会根据窗口大小进行调整。窗口大小是和MSS没有任何关系的。规定MSS的大小是求得一个平衡，就是在网络层传输过程中网络利用率和IP数据包分片开销之间的平衡。

### 2. TCP发送方发送了3个包，接收方可能收到几个包？

答：TCP是面向字节流的服务，TCP不保证接收方应用程序收到的数据块和发送方应用程序发出的数据块具有对应的大小关系。也就是说，发送方发送了3个包，接收方可能只用了两个数据块就把收到的字节流交付给上层的应用程序。

### 3. TCP可靠传输的机制是如何实现的？

答：TCP可靠传输是依靠以字节位单位的滑动窗口和超时重传来实现的。

要点：（1）发送端和接收端分别维护一个窗口。

（2）采用累积确认的方式，在收到几个分组之后，对按序到达的最后一个分组发送确认，表示到这个分组为止的所有分组已经收到了。

（3）发送窗口包括两个部分，一部分是已经发送但是没有收到确认，另一部分是允许发送但是尚未发送字节的可用窗口。接收窗口包括未能按序到达的字节和允许接收的字节部分。  
另一种答案：

（1）应用数据被分割成TCP认为最适合发送的数据块。

（2）超时重传：当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

（3）TCP给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

（4）差错检测：校验和：TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。

（5）TCP的接收端会丢弃重复的数据。

（6）流量控制：TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP使用的流量控制协议是可变大小的滑动窗口协议。

（7）拥塞控制：当网络拥塞时，减少数据的发送。

### 4. TCP的流量控制和拥塞控制分别是什么？有什么区别？

答：（1）TCP流量控制就是要求发送方的发送速率不要太快，让接收方来得及接收，发送方发送窗口不能超过接收方给出的接收窗口的数值。

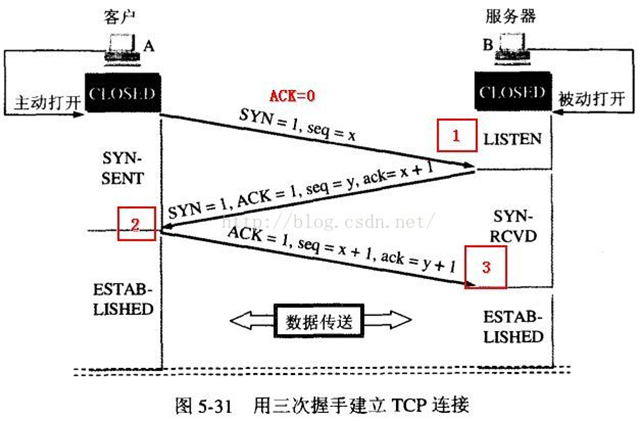
（2）拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器/链路不至于过载，拥塞控制的条件是：对资源的总需求>总需求。TCP端点只要迟迟收不到对方的确认信息，就猜想此时网络某处很有可能发生了拥塞。

（3）流量控制是点对点的通信量控制，而拥塞控制是全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。

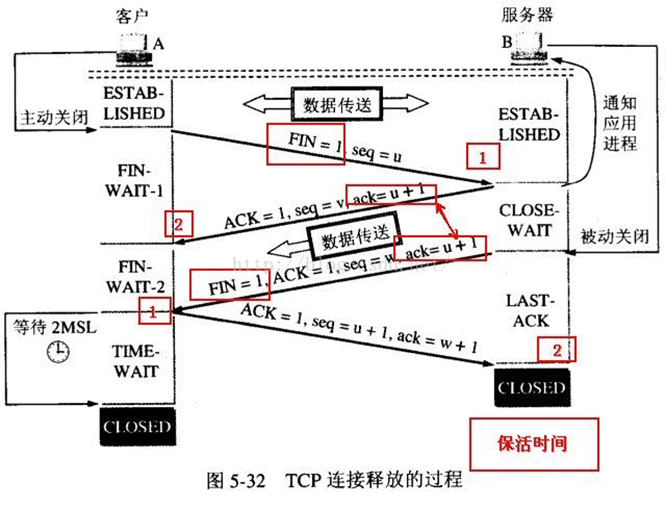
（4）拥塞控制的方法：慢开始；拥塞避免；快重传；快恢复。

### 5.TCP三次握手机制和四次断开机制？

答：（1）三次握手的过程



（2）四次断开的过程



### 6. 为什么建立连接需要三次握手，而断开连接需要四次握手？

答：在建立连接的过程中，服务器可以把确认ACK和同步SYN放在一个包中传输。但是在断开连接的时候，如果一端收到FIN包，此时仍有数据尚未发送完毕，此时首先向对端发送ACK，等到剩下的包都发送完毕，再向对端发送FIN和ACK，断开这个方向的连接。

简单来说，每个方向需要一个FIN和ACK，当一端发送了FIN包之后，处于半关闭状态，此时仍然可以接收数据包。

### 7. TIME\_WAIT状态持续时间及原因？

答：TIME\_WAIT发生在主动请求断开连接的一端，这个状态必须等待2MSL（最长报文段寿命）时间，有两个原因：

（1）保证A发送的最后一个ACK确认包能够到达B。因为如果丢失，B会重传自己的FIN+ACK报文段，A就能在2MSL时间内收到这个重传的报文段。接着A重传一次确认，重新启动2MSL计时器。如果不等待这个时间，那么B就一直处于LAST\_ACK状态无法关闭连接。

（2）FIN和ACK在网络中最长生存时间就是2MSL，这样就可以可靠地断开TCP双向连接。还有就是防止“已经失效的请求连接报文段”出现在本连接中，在2MSL时间内，本连接持续时间内的所有报文段都从网络中消失。下一个连接不会出现旧的连接请求报文段。

### 8. 超时重传和快速重传

答：超时重传：当超时时间到达时，发送方还未收到对端的ACK确认，就重传该数据包。

快速重传：当后面的序号先到达，如果接收方收到了1、3、4，而没有收到2，就会立即向发送方重复发送ACK=2的确认请求重传。如果发送方连续收到三个相同序号的ACK，就重传这个数据包，不用等待超时。

### 9. 有哪些应用层协议是基于TCP的，哪些是基于UDP的？

答：TCP： FTP、HTTP、Telnet、SMTP、POP3、HTTPS

UDP：DNS、SNMP、NFS

### 10. 三次握手过程中有哪些不安全性？

答：a. 伪装的IP地址向服务器发送SYN请求建立连接，服务器向该IP回复SYN和ACK，但是找不到该IP对应的主机，超时时，服务器重传ACK，当大量攻击者请求建立连接的时候，服务器存在大量的未完成的三次握手连接，而不能响应其他正常的连接，就是SYN泛洪攻击，是DoS攻击的一种。

防范措施：

（1）降低SYN timeout时间，使得主机尽快地释放半连接占用。

（2）采用SYN cookie设置，短时间内收到连续的某个IP重复SYN请求，就认为该IP受到了攻击，丢弃该IP后续的请求报文。

（3）在网关处设置过滤，拒绝将一个源IP地址不属于其来源子网的包进行更远的路由。

b. 当一个主机向服务器发送SYN请求连接，服务器回复ACK和SYN后，攻击者截获ACK和SYN。然后伪装成原始主机继续与服务器进行通信。

### 11. accept发生在三次握手的哪一步？

答：（1）TCP三次握手发生在connect step, accept只是把内核中的“已完成连接队列（FIFO with established state)”取出，另一个队列是“未完成连接队列( FIFO with syn\_rcvd state)”，并且这两个队列总和的最大值是backlog。

（2）accept会监听已完成队列是否非空，当队列为空时，accept就会阻塞。当队列非空时，就从已完成队列中取出一项并返回。而已完成队列中的都是三次握手过程已经完成的，因此accept发生在三次握手之后。

### 12. TCP在listen时的参数backlog的意义？

答：linux内核中会维护两个队列：（1）未完成队列：接收到一个SYN建立连接请求，处于SYN\_RCVD状态；（2）已完成队列：已完成TCP三次握手过程，处于ESTABLISHED状态

　　当有一个SYN到来请求建立连接时，就在未完成队列中新建一项。当三次握手过程完成后，就将套接口从未完成队列移动到已完成队列。

　　backlog曾被定义为两个队列的总和的最大值，也曾将backlog的1.5倍作为未完成队列的最大长度，一般将backlog指定为5。

### 13. 什么是http 协议？http协议的请求方法有哪些？

答：（1）http是客户端和服务器之间进行数据传输的格式规范，格式简称为超文本传输协议。

（2）http请求方法包括一些几种：

GET 用于请求访问已经被URI标识的资源，通过URL传参给服务器

POST 用于传输信息给服务器，主要功能和GET相似，一般推荐用POST方式传参

PUT 传输文件，报文主体中包含文件内容，保存到对应的URI位置

HEAD 获得报文头部，与GET相似，只是不返回报文主体，一般用于检验URI是否有效

DELETE 删除文件，与PUT方法相反，删除对应URI位置的文件

OPTIONS 查询响应URI支持的HTTP方法

### 14.说出http的状态返回码？

答：1xx: 这些状态代码表示临时的响应。客户端在收到常规响应之前，应准备接收一个或多个1xx响应。100 继续 101 切换协议

2xx：成功 这类状态代码表明服务器成功地接受了客户端请求。

200-确定。客户端请求已成功；201-已创建；202-已接受；

3xx： 重定向 客户端浏览器必须采取更多操作来实现请求。例如，浏览器可能不得不请求服务器上的不同的页面，或通过代理服务器重复该请求。301-对象已永久移走，即永久重定向，302-对象已临时移动，比如未登陆的用户访问用户中心重定向到登录页面，访问404页面会重新定向到首页。

4xx: 客户端错误，发生错误，客户端似乎有问题。例如，客户端请求不存在的页面，客户端未提供有效的身份验证信息。400-错误的请求。401-访问被拒绝，403-禁止访问，404-未找到。

5xx: 服务器错误，服务器由于遇到错误而不能完成该请求。500-内部服务器错误，503-服务不可用，504-网关超时。

### 15. 什么是Http协议无状态协议?怎么解决Http协议无状态协议?

答：（1）无状态协议对于事务的处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息。（2）无状态协议解决办法：通过coolie或者session会话保存。

### 16. Http协议由什么组成?

答：请求报文包括三部分：

1. 请求行：包含请求方法，URI，HTTP版本协议
2. 请求首部字段
3. 请求内容实体

响应报文包含三部分：

1. 状态行：包含HTTP版本，状态码，状态码原因短语
2. 响应首部字段
3. 响应内容实体

### 17. Http协议中Http1.0和1.1区别?

答：在1.0版本中，当建立连接之后，客户端发送一个请求，服务器返回一个信息之后就关闭连接，当浏览器下次请求的时候又要建立连接，显然这种不断建立连接的方式会造成很多问题。

（1） HTTP/1.0协议使用非持久连接,即在非持久连接下,一个tcp连接只传输一个Web对象，每次请求都需要与服务器建立TCP连接，服务器完成请求之后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。

（2）HTTP/1.1默认使用持久连接(然而,HTTP/1.1协议的客户机和服务器可以配置成使用非持久连接)。在持久连接下,不必为每个Web对象的传送建立一个新的连接,一个连接中可以传输多个对象。支持长连接和请求的流水线处理。同一个TCP可以传送多个HTTP请求和响应。

### 18. Http协议实现的原理机制



### 19. Get和Post的区别是什么？

答：区别一：GET重点从服务器上获取资源，POST重点在向服务器发送数据。

区别二：GET传输数据通过URL请求，以field=value的形式，放置在URL后面，并用?连接，多个请求用&连接，http://127.0.0.1/Test/login.action?name=admin&password=admin

区别三：GET将数据直接放在URL后面，而POST将数据放在request\_body里面，GET传输量较小，所以上传文件的时候只能通过POST方式。

区别四：GET是不安全的，因为URL可见，可能会泄漏隐私信息，比如密码等，POST比GET安全。

区别五：对参数的数据类型，GET只接受ASCII字符，而POST没有限制；GET请求在URL中传送的参数是有长度限制的，而POST么有。

区别六：对于GET请求，浏览器会把HTTP header和data一起发出去，服务器响应200（返回数据）。而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

### 20. Http与Https优缺点?

答：(1).通信使用明文不加密,内容可能被窃听,也就是被抓包分析

(2).不验证通信方身份,可能遭到伪装

(3).无法验证报文完整性,可能被篡改

Https就是Http加上加密处理(一般是SSL安全通信线路)+认证+完整性保护

HTTP的默认端口是80，HTTPS的端口是443

### 21.Restful API的特点？

答：*面向资源，统一接口，无状态，URI*

（1）Representational State Transfer，翻译是“表现层状态转化”，URL定位资源，用HTTP动词（GET,POST,PUT,DELETE)描述操作。REST是所有Web应用都应该遵守的架构设计指导原则。

（2）面向资源是REST的最明显的特征，对于同一个资源的一组不同的操作。资源是服务器上一个可命名的抽象概念，资源是以名词为核心来组织的，首先关注的是名词。REST要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操作。（7个HTTP方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS）。

（3）HTTP协议就是属于REST架构的设计模式。比如，无状态，请求-响应。

### 22.cookie是什么？它的作用是干嘛的？session是什么？作用是什么？

答：背景：HTTP是一种无状态的协议，为了分辨链接时谁发起的，就必须自己去解决这个问题，不然有些情况下即使是同一个网站我们每打开一个页面就要登录一下。Cookie和session就是来解决这个问题的。

（1）cookie: 场景是这样的，日常登录一个网站，今天输入用户名和密码登录了，第二天再打开很多情况下就直接打开了，用到了cookie。通俗地讲，cookie是访问某些网站以后在本地存储的一些网站相关信息，下次再访问的时候减少一些步骤。准确地讲，cookie是服务器在本地机器上存储的一小段文本并随着每一个请求发送到同一个服务器，是客户端保持状态的一种方案。Cookie的主要内容包括：名字，值，过期时间，路径和域。

（2）session: 场景是这样的，一个购物车，添加了商品之后客户端处可以看到添加了哪些商品，而服务器如何判断呢，所以也需要存储一些信息，这里就用到了session。Session是存在在服务器中的一种用来存放用户数据的类HashTable结构。当浏览器第一次发送请求的时候，服务器自动生成了一个HashTable和一个session ID来唯一标识这个HashTable，并将其通过响应发送到浏览器。当浏览器第二次发送请求的时候，会将前一次服务器响应中的session ID放在请求中一起发送到服务器上，服务器从请求中提取出session ID，和保存的session ID做比较，找到这个用户对应的HashTable。

两者的联系：

Session的实现方式与Cookie有一定的联系。试想一下，建立一个连接就有一个session ID，那么打开几个页面就好几个了，这显然不是我们想要的，我们为了区分，将session ID放在Cookie中，每次访问的时候将Session ID带过去就可识别了。

两者区别：

一个在客户端一个在服务器。因为cookie存在客户端所以用户可以看到，所以可以编辑伪造，十分不安全。还有就是session过多的时候回消耗服务器的资源，所以大型网站会有session的服务器，cookie存在在客户端当然没啥问题。

### 23. 访问一个网站，从敲入url回车之后，大概经过了哪些步骤

答：简单来说，

1. 首先是浏览器程序开始解析地址，把地址分为域名和路径两个部分，然后连接DNS服务器，查询这个域名的IP地址。
2. 获得DNS返回的IP地址之后，浏览器开始按照HTTP协议的格式向该IP地址和路径请求内容。
3. 服务器受到某个HTTP请求之后，就会把内容按照HTTP协议的格式返回这个请求的客户端。
4. 浏览器受到服务器返回的内容之后，开始渲染并显示出来。
5. 浏览器和服务器结束对话。

计算机网络课本上的答案：

1. 浏览器分析超链接指向的页面的URL。
2. 浏览器向DNS请求解析[www.baidu.com的IP](http://www.baidu.com的IP)地址。
3. 域名系统DNS解析出百度服务器的IP地址。
4. 浏览器与服务器建立TCP连接
5. 浏览器发出取文件指令：GET/chn/yxsz/index.html
6. 服务器给出响应，把文件index.html发给浏览器。
7. TCP连接释放
8. 浏览器显示“文件的index.html中的所有文本。

## 六．数据库

### 1. 什么是存储过程？有哪些优缺点？

答：存储过程是在大型数据库系统中，一组为了完成特定功能的SQL语句集，存储在数据库中，经过第一次编译之后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。

一般所说的存储过程就是本地存储过程。

### 2. 索引是什么？有什么作用以及优缺点？索引无论哪种情况都能加快效率吗？

答：索引是存储的表中一个特定列的值数据结构（最常见的是B-tree）。索引是在表的列上创建。所以要记住的关键点是索引包含一个表中列的值，并且这些值存储在一个数据结构中。请记住这一点，索引是一种数据结构。

B-Tree 是最常用的用于索引的数据结构。因为它们是时间复杂度低， 查找、删除、插入操作都可以可以在对数时间内完成。另外一个重要原因存储在B-Tree中的数据是有序的。数据库管理系统（RDBMS）通常决定索引应该用哪些数据结构。但是，在某些情况下，你在创建索引时可以指定索引要使用的数据结构。

哈希表索引是怎么工作的？

哈希表是另一种你可能看到用作索引的数据结构。这些索引通常被称为哈希索引。使用哈希索引的原因是，在寻找值时哈希表效率极高。所以，如果使用哈希索引，对于比较字符串是否相等的查询能够极快的检索出的值。例如之前我们讨论过的这个查询(SELECT \* FROM Employee WHERE Employee\_Name = ‘Jesus’) 就可以受益于创建在Employee\_Name 列上的哈希索引。哈系索引的工作方式是将列的值作为索引的键值（key），和键值相对应实际的值（value）是指向该表中相应行的指针。因为哈希表基本上可以看作是关联数组，一个典型的数据项就像“Jesus => 0x28939″，而0x28939是对内存中表中包含Jesus这一行的引用。在哈系索引的中查询一个像“Jesus”这样的值，并得到对应行的在内存中的引用，明显要比扫描全表获得值为“Jesus”的行的方式快很多。

### 3. 什么是事务？数据库事物的四大特征？

答：

### 4. 说一说drop、delete与truncate的区别

### 5. 超键、候选键、主键、外键分别是什么？

### 6. 什么是视图？以及视图的使用场景有哪些？

### 7. 说一说三个范式。

### 8. 如何加快查询语句的效率

## 七．设计模式

### 1. 单例模式是什么，如何实现？

答：这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码：构造函数是私有的。

应用实例： 1、一个党只能有一个主席。 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景： 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

### 2.单例模式的线程安全版本代码：

### 3.什么是工厂模式？

## 八．大数据

### 1. 10亿个QQ号，找出一个QQ号是否在里面，用O(1)的代价，QQ号的范围是int型的。

答：这个题用bitmap来做，bitmap是位图，位图的原理是用一个bit来标识一个数字是否存在，采用一个bit来存储一个数据，这样可以节省存储空间。可以用一个int的位存储0-31这些数据，这些数据的每个数据仅用一个bit存储，这样节省了空间。

因为QQ号是整型，开一个2^32位的bit数组，大概是5亿个字节，大约500M，然后遍历10亿个QQ号，找到对应QQ号的下标位置置1.这样就实现了对10亿个QQ号的存储。给定一个QQ号，找到这个QQ号对应的bit位，如果这个bit位是1的话，则存在，若不是1则不存在。

### 2. 给定100亿个整数，设计算法找出只出现一次的整数。

答：使用hash将所有的整数映射到1000个文件中去，每个文件使用bitmap表示，用两个bit表示出现的次数，00表示没有出现过，01表示只出现过一次，10表示出现过多次，11表示舍弃。最后归并每个文件中只有一次的数。

无符号整数一共42亿个，一个整数用两位表示，大约占用1G内存，00表示没有出现过，01表示只出现过一次，10表示出现过多次，11表示舍弃。

3. 给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件的交集。

答：(1)哈希桶分组和组内bitmap。如果这里的整数是32bit的话，直接使用bitmap的方法就能实现了。所有整数一共2^32种可能，每个数用2bit来表示。共需要（2^32）\*2/8=1GB内存，遍历两个文件中所有的整数，然后寻找bitmap中‘11’对应的整数就是两个文件的交集。这样就在线性复杂度完成。“00”表示两个文件均没出现，“10”表示文件1出现过，“01”表示文件2出现过，“11”表示两个文件均出现过。

(2)用哈希函数将第一个文件所有整数映射到1000个文件中，每个文件大约1000万个整数，大约40M内存，内存可以放下，把1000个文件记为a1,a2,..,a1000。用同样的哈希函数将第二个文件映射到1000个文件中去，将这1000个文件记为b1,b2,…,b1000。由于使用的是相同的哈希函数，所以两个文件中一样的数字将会被分到文件下标一致的文件中，那么分别对a1和b1求交集，a2和b2求交集….，最后将结果汇总，就求出了两个文件的交集。Linux下：cat file1,file2 | sort | uniq –d

### 3. 给一个超过100G的日志文件，设计算法找到出现次数最多的IP地址。

答：利用Hash切分，可以划分为100份，重点是使得相同的IP地址进入编号相同的小文件中去。这就避免了相同的IP地址分散在不同的小文件的问题。把100G大文件中的IP读出来,将每个IP地址映射到相应的文件中去，file\_id=hash(ip)%1000;每个文件分别求出最高频率的IP地址，再合并hash分桶法。

### 4. 100亿个数据中找到最大的1000个数字

答：（1）用小顶堆做，小顶堆是一种特殊的完全二叉树，假设二叉树一共N个节点，那么二叉树的深度就是logN，对于小顶堆来说，移动根元素到底部或者移动底部元素到根部的时间复杂度都是logN。基本思路是，先从问价中取出1000个元素构成一个小顶堆数组k，然后依次对剩下的100亿个数组进行遍历m，如果数字m大于小顶堆的顶元素，那么就用m取代对顶元素，继续对新的数组进行重新构建组成一个新的小顶堆。时间复杂度是（100亿-1000）log(1000)),也就是O((N-M)log(M)),这个算法的性能尚可，空间复杂度比较低，IO读取比较频繁，对系统的压力大。

（2）采用分区法：将100亿个数据分成1000个分区，每个分区上1000万个数据，在每个分区上再细分成100个分区，一共1000\*100个分区，然后启动多线程进行处理，各个分区上采用小顶堆算法计算出最大的1000个数据，分层进行合并然后重新计算不同层上的最大1000个数，最终递归到上层。Linux系统中一个进程能启动的默认线程数是1024，所以要么调整最大线程数，要么在线程调用上做一些处理。比如一个线程完成一个分区之后再去处理相邻的分区，或者在分区的时候将所有分区的数目控制在1024内。这个算法契合了map-reduce的思想，利用了多线程和多处理器的优势，减少了多余的比较和IO读取操作，性能比第一种好，但是算法复杂，要考虑的情况也更多。

（3）分块查找，比如，100万为一块，一共可以分为10000块，从100万中找出1000大的，剩下就是从一千万中找出1000个。100万中找出最大的1000个，可以采用快速排序的方法：分为两堆，如果大的那个堆的个数N大于1000个，继续对大堆进行快速排序分为两堆，如果大堆个数N小于1万个，就从小堆中快速排序一次，找出1000-N大的数字，不断地递归上述的步骤，就可以找到相关的结果。

## 九．网络编程/多线程

### 1. 分布式日志题目

分布式日志是一个常见的设计模式，基本的假设如下：

1. 你收到一个消息序列: m1, m2, m3, …

2. (可选)这些消息流是无界的（如果候选者不能理解一个无限消息序列，请他们去解决一个大的确定的消息序列）。

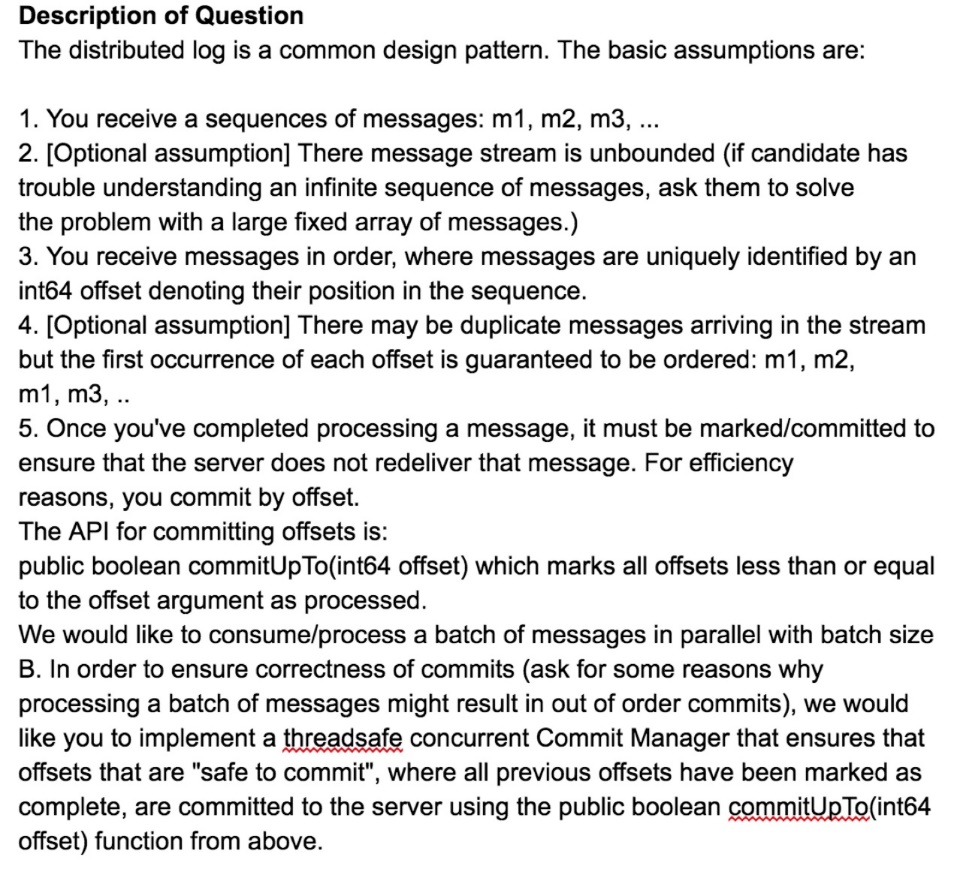
3. 你按序接收消息，消息被唯一地用64位整型偏置表示，表示他们在序列中的位置。

4.(可选)数据流可能存在消息重复，但是消息第一次出现是按序的：m1,m2,m1,m3….

5.一旦你完成了一个消息的处理，它必须被标记/提交以确保服务器不会重新传递该消息。出于效率考虑，你用offset提交。

用于提交offset的API是：public boolean commitUpTo(int64 offset)

一个标记所有偏移量小于或等于所处理的偏移量参数的函数。我们将并行消费/处理一批消息，批大小是B。为了保证提交的正确性（为什么处理一批消息可能会导致不按序提交呢），我们请你写一个线程安全的并发提交管理器，保证offsets的安全提交，所有之前的offsets已经标识为完成。



### 2. 常见的多线程的问答题：

#### 1. 线程的基本概念、线程的基本状态和状态之间的关系？

概念：线程是进程的一个执行控制单元，执行路径。一个线程是进程的一个顺序执行流。同类的多个线程共享一块内存空间和一组系统资源，线程本身有一个供程序执行的堆栈。线程在切换时符合小，因此线程也被称为轻负荷进程，一个进程可以包含多个线程。

基本状态：

1）新建：new语句创建的线程对象处于新建状态，仅仅被分配了内存。

2）等待：当线程在new之后，并且在调用start方法之前，线程处于等待状态。

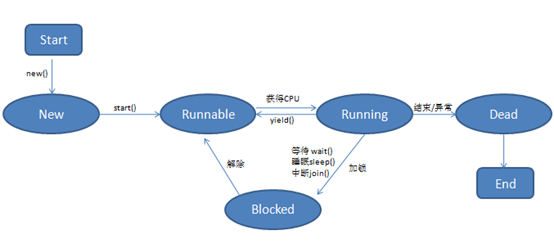
3）就绪：当线程对象创建之后，其他线程调用它的 start()方法，该线程就进入就绪状态。

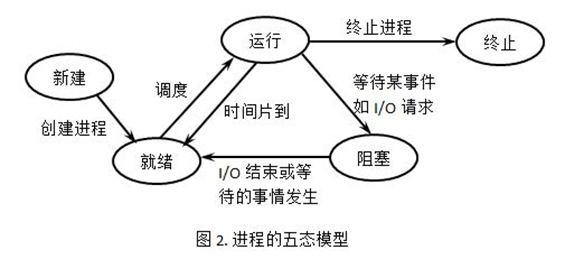
4）运行: 处于这个状态的线程占用CPU，执行程序代码。在并发环境中，如果计算机只有一个CPU，那么任何时刻只会有一个线程处于这个状态。只有就绪的线程才能转到运行状态。

5）阻塞：阻塞状态是指线程因为某些原因放弃了CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU，直到线程重新进入就绪状态，它才会有机会获得运行状态。

阻塞分为三种状态：等待阻塞，同步阻塞，其他阻塞。

6）死亡：当线程执行外run()方法的代码时，或者遇到了未捕获的异常，就会推出run()方法，此时进入死亡状态，该线程结束生命周期。





#### 2. 进程与线程的区别?这个题在操作系统那块有介绍。

#### 3. 多线程有几种实现方法，分别是什么？

答：多线程有两种实现方法，一是继承Thread类，重写方法run()，二是实现Runnable接口，实现方法run()；

同步有两种实现方法，分别是synchronized、wait与notify。

#### 4. 线程同步和互斥有几种实现方法，都是什么？

答：线程同步是指多个线程同时访问某个资源的时候，采用一系列机制保证同时最多只能有一个线程访问该资源。线程同步是多线程中必须考虑和解决的问题，因为很有可能会发生多个线程同时访问（主要是写操作）同一个资源。如果不进行线程同步，很可能会引起数据混乱，造成线程死锁等。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

（1）临界区：通过对多线程的串行化来访问公共资源或者一段代码，速度快，适合控制数据访问。

（2）互斥量：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以可以保证公共资源不会同时被多个线程访问。

（3）信号量：它允许多个线程同一时刻访问同一资源，但是会限制同一时刻访问此资源的最大线程数目。信号量对象对线程的同步方式与前面几种方法不同，它允许多个线程同时使用共享资源，与操作系统的PV操作类似。

（4）事件（信号）：通过通知操作的方式保持多线程的同步，还可以方便地实现多线程的优先级比较操作。

#### 5. 多线程同步和互斥有何异同，分别在什么情况下使用它们请举例说明。

答： 线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步

#### 6. 在Windows编程中互斥量与临界区比较类似，请分析一下二者的主要区别。

答：（1）临界区只能用于对象在同一进程里线程之间的互斥访问；互斥体可以用于对象在进程间或者线程间的互斥访问。

(2) 临界区是非内核对象，只能在用户态进行锁操作，速度快；互斥体是内核对象，在内核态进行锁操作，速度慢。

（3）临界区和互斥体都可以在windows下使用，linux只有在互斥体可用。

互斥量可以很好的解决由于线程意外终止资源无法释放的问题。

### 3. 编写一个程序，开启3个线程，这三个线程的ID分别是A,B,C，每个线程将自己的ID在屏幕上打印10遍，要求输出结果必须按照ABC的顺序显示：ABCABC…依次递推。

思路：用信号量进行各个子线程之间的互斥，创建三个信号量ABC。初始时A的资源数是1，B和C的资源数是0，访问A之后，将B的资源数加1，访问B之后将C的资源数加1，访问C之后将A的资源数加1，创建三个子线程顺序访问ABC。

首先我们来看一个条件变量的api:

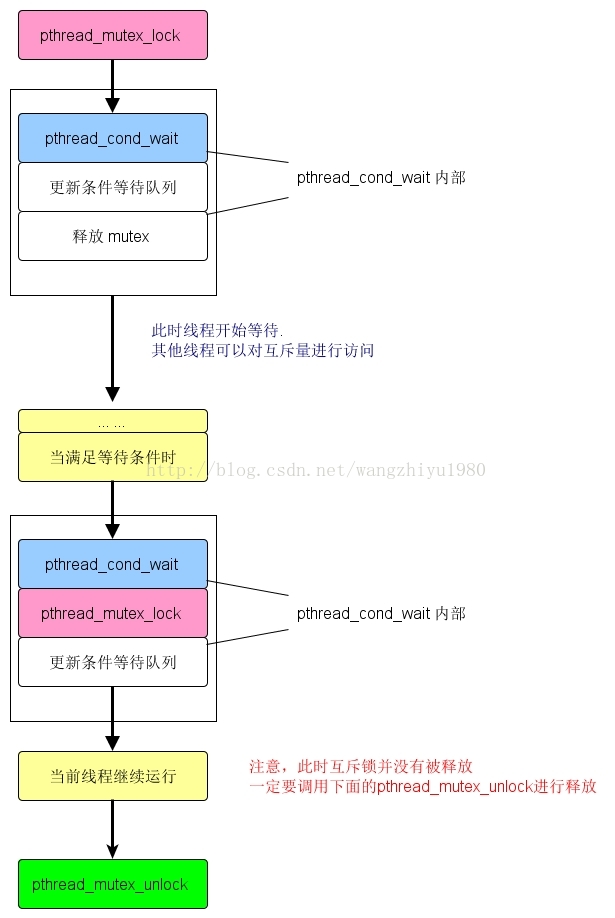
pthread\_cond\_wait() :线程间同步有一种情况：线程A需要等待某个条件成立才能继续向下进行，现在这个条件不成立，线程A就阻塞等待，而线程B在执行过程中如果使得这个条件成立了，就会唤醒A继续执行。 在pthread库中通过条件变量来阻塞等待一个条件，或者唤醒等待这个条件的线程。

Pthread\_ create() :创建一个线程。

Pthread\_join() : 用来等待一个线程的结束。原型：int pthread\_join(pthread\_t thread, void \*\*retval); 这个函数以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么函数会立即返回，并且thread指定的线程必须是joinable的。**在Linux中，默认情况下是在一个线程被创建后，必须使用此函数对创建的线程进行资源回收**，但是可以设置Threads attributes来设置当一个线程结束时，直接回收此线程所占用的系统资源，详细资料查看Threads attributes。

代码中如果没有pthread\_join主线程会很快结束从而使整个进程结束，从而使创建的线程没有机会开始执行就结束了。加入pthread\_join后，主线程会一直等待直到等待的线程结束自己才结束，使得创建的线程有机会执行。

代码：



1. #include <iostream>
2. #include <stdlib.h>
3. #include <pthread.h>
4. **using** **namespace** std;
5. //初始化互斥量和条件变量
6. pthread\_mutex\_t myloack=PTHREAD\_MUTEX\_INITIALIZER;
7. pthread\_cond\_t mycond=PTHREAD\_COND\_INITIALIZER;
8. **int** n=0;
9. **void** \*ThreadFunc(**void** \*arg)
10. {
11. **int** num=(**int** )arg;
12. **for** (**int** i = 0; i < 10; ++i)
13. {
14. pthread\_mutex\_lock(&myloack);  //锁住互斥量
15. **while** (n!=num)
16. pthread\_cond\_wait(&mycond,&myloack);  //阻塞等待条件
18. **if** (num==0)
19. cout<<"A";
20. **else** **if**(num==1)
21. cout<<"B";
22. **else**
23. cout<<"C"<<endl;
24. n=(n+1)%3;
25. pthread\_mutex\_unlock(&myloack);  //释放锁
26. pthread\_cond\_broadcast(&mycond);  //广播，根据加入等待队列的先后顺序依次唤醒他们。
27. }
28. **return** (**void** \*)0;
29. }
30. **int**  main(**int** argc, **char** **const** \*argv[])
31. {
32. pthread\_t id[3];
33. **for** (**int** i = 0; i < 3; ++i)
34. {
35. **int** err=pthread\_create(&id[i],NULL,ThreadFunc,(**void** \*)i);
36. **if** (err!=0)
37. {
38. cout<<"create err:"<<endl;
39. exit(-1);
40. }
42. }
44. **for** (**int** i = 0; i < 3; ++i)
45. {
46. **int** ret=pthread\_join(id[i],NULL);  //释放资源
47. **if** (ret!=0)
48. {
49. cout<<"join err:"<<endl;
50. exit(-1);
51. }
52. }
53. **return** 0;
54. }

### **4. 一个生产者-消费者的例子**

消费者A等待生产B生产出产品之后才能打印，否则消费者阻塞等待生产者生产。

## 十．其它常识题归总

### 分布式存储的原理

### 一致性哈希

### 负载均衡

### SSL基本原理

## 十．项目经历：

项目一：密态数据处理分析技术原型验证系统设计

(1) 学习并理解Cryptdb原理以及源码，测试并使用使用Cryptdb 加密数据库；

Cryptdb是在数据库上实现了同态加密技术，cryptdb希望在数据库系统上实现加密运算，达到的效果是：存在数据库中的数据全部是加密的，但是数据库依然可以对加密的数据执行用户的SQL语句，返回加密的数据给用户，然后用户对返回的结果进行解密，获得明文数据。但对于数据库来讲，只要求几种常见的运算，不需要任意的运算，比如，select中where需要相等比较的操作，order by需要大小比较的操作，对于一些函数比如sum，需要加法操作。如果只支持这些常见的操作，不涉及更为复杂的操作时，就可以在吞吐量下降20%的开销的情况下, 满足大部分的SQL查询。

（2）编写部分密态处理程序，包括同态加，同态乘，排序，求最大值最小值，求线性回归，平均值等运算。设计演示系统功能，并编写前端和后端的系统接口，最后调试系统，演示和讲解。Web应用这块用的是简单的python flask restful api，来写的接口，接口涉及到前端所需和后端能够提供的格式的匹配，数据传输的过程中是以json的形式传输，在后端的服务程序处理的时候是以密态的字符串进行处理。在用C++写密态处理程序的时候，用到的开源密码库是Openssl和HElib库，前者提供了基础的加密算法和基本协议，后者是针对同态加密的开源密码库。

## 十一．HR面试

### 1. 自我介绍

我叫党林涛，来自上海交通大学，现在是电院研究生二年级的一名硕士研究生，专业是电子与通信工程。研究方向是下一代网络与安全，研究生期间除了做理论研究并且发表两篇高水平论文之外，还参与了一个有工程锻炼价值的加密数据库的开源项目，用做研究过程中设计的算法为加密数据库添加了一些功能，另一方面还独立实现相关的同态加密算法，编译成动态链接库供他人使用。

### 2. 3年之后的规划，要达到什么样的状态？

感谢你提出这么深刻的话题，我的兴趣是技术，优势是善于思考并解决问题，因此我选择阿里的研发岗，这个工作符合我的预期，我也很喜欢，如果有可能，我会认真对待在阿里做研发的工作，因为这是个难得的机会。

谈到职业规划，近三到五年，我打算通过做技术研发积累起代码经验和实际的解决工程问题的经验，希望能够稳定提升，学到更多的实践知识，后续争取能够独当一面，成为一名优秀的研发工程师。至于长期的职业规划，我会根据环境的变化，工作内容的变化，以及自身能力的变化不断做调整，这是我暂时的考虑，谢谢！

### 3. 自己的缺点和优点？

缺点: 作为一名在校学生，目前最大的缺点就是工程经验相对较少，研究生期间虽然做过一些项目，但主要还是做研究，设计算法，实践项目少了一些。我意识到这个问题，已经开始着手让自己多接触一些开源项目练手，努力提高工程能力。当然，最大的锻炼还是在实习中才能获得，所以我非常期待去阿里实习的机会。

优点：最大的优点是做事踏实，善于思考并解决问题。我也认为这是一名优秀工程师必不可少的特质。

### 4. 有什么品质能够推动你前进？

### 谈谈你对互联网的见解

### 为什么选择ali?

1. 阿里巴巴具有国内最高水平的互联网技术，是做技术的人都非常向往的公司。
2. 阿里巴巴内部的技术氛围浓厚，技术达人众多，能帮助技术人员快速成长。

### 7. 你了解阿里文化和开源吗？

（1）2011年阿里就公布了第一个开源项目，阿里巴巴是唯一一家入围 GitHub 顶尖贡献名单的中国公司。到2017年底已经有150+个开源项目了。阿里巴巴并没有开源项目的 KPI 考核，各个团队都是发自内心地将踩过的坑和总结的经验融入到开源项目中，供业界所有人使用，希望帮助他人解决问题。这正是社区一致的开源精神，大家都可以站在巨人的肩膀上，每个人都可以即是老师又是学生，这或许可以解释为什么IT技术的发展能如此充满活力。不过现在最大的问题在于开源项目创建需要长期维护，在这方面还是饱受诟病的。

（2）阿里的“六脉神剑”：客户第一，客户是衣食父母；团队合作：共享共担，平凡人做非凡事；拥抱变化：迎接变化，勇于创新；诚信：诚实正直，言行坦荡；激情：乐观向上，永不放弃；敬业:专业执着，精益求精。

### 8. 你能说说你在项目中的收获吗？

（1）首先是工程能力得到很快提升，在做项目的时候需要用到的很多东西没有学过，就通过快速学习然后迅速解决问题，坚信方法总比困难多。

（2）其次是与人沟通和合作的能力，一个大的项目需要多个人共同完成，比如在做加密数据库那个项目的时候，我负责写后台服务器部分，需要和做前端的同学协商好接口，但是做前端的同学比较忙，我就需要多和他沟通，实在赶不上就先自己把接口格式写好然后写成文档发给他，等他有空的时候将接口写完然后联调。总之不能耽误项目进度。

### 9. 你还有什么问题问我吗？

我想了解下前三个技术面试官以及您对我面试的总体评价，谢谢！

我想了解下实习留用率的情况！

## 十二．摩根笔试面试面经

### 1. Introduce yourself

My name is Dang Lintao and I come from Shanghai Jiao Tong University. Now I am a graduate student in second grade. My major is Electronics and Communication Engineering. My research focus on the next-generation network and information security. I study how to use cryptography to achieve the security of information flow in various next-generation networks like software-defined network and Information-centric network. At present, two papers have been accepted by well-known international conferences. As for amateur(ˈæmətər) life, I'd like to play badminton, watching movies and watching sports games. I will graduate in March next year and plan to work on software development or R&D after graduation. That's all.

### 2.Why morgen?

As an investment bank, the internal trading system certainly has high quality requirements and this will be a challenge.

As a global company, there are more opportunities to communicate with people from different regions and teams.

I like the working environment and atmosphere of foreign companies like morgen.

### 3.Morgen?

Morgan Stanley is an international financial service company. Stock Research Department, Investment Banking Department, Private Wealth Management Department, Foreign Exchange/Debench Department, Commodity Trading Department, Fixed Income Research Department, Investment Management Department, Direct Investment Department, and Institutional Stocks Department.