

字符串类面试题的基础知识

前一节

教程

问答(4)

如果你使用 Java 语言，那么你首先要知道，Java 的 String 是一个类（Class），你需要知道如下的一些基本知识：

1. 如何判断两个字符串相等
2. 取出第 i 个字符以及字符串的遍历
3. null 和 "" 的区别

其他语言

- C++ 的 string 是一个类。
- Python 的字符串是一个类。

Java中判断字符串相等的方式

代码

先来看一段代码

```
public class StringTest {  
    public static void main(String[] args) {  
        String H = "hello";  
        String H_1 = H;  
        String H_2 = "hel";  
        String H_3 = H_2 + "lo";  
        String H_4 = H_2.concat("lo");  
  
        System.out.println(H);           // hello  
        System.out.println(H_1);         // hello  
        System.out.println(H_2);         // hel  
        System.out.println(H_3);         // hello  
        System.out.println(H_4);         // hello  
  
        //==等号测试  
        System.out.println(H == H_1);    // true  
        System.out.println(H == H_3);    // false  
        System.out.println(H == H_4);    // false  
        System.out.println(H_3 == H_4);  // false  
  
        //equals函数测试  
        System.out.println(H.equals(H_1)); // true  
        System.out.println(H.equals(H_3)); // true  
        System.out.println(H.equals(H_4)); // true  
        System.out.println(H_3.equals(H_4)); // true  
  
        //StringBuilder测试  
        StringBuilder helloBuilder = new StringBuilder("hel");  
        System.out.println(helloBuilder.equals(H_2)); // false  
    }  
}
```

代码中注释为对应的结果。

为什么Java中不能直接用 == 判等？

Java中String类型具有一个equals的方法可以用于判断两种字符串是否相等，但是这种相等又与运算符“==”所判断的“相等”有所不同。

- 使用“==”判断的相等时指相同的内存地址，也就是同一个对象实例。
- 使用equals方法判断的相等在不同的对象中实现不同，意义也不同。

Java中所有的对象都继承自Object类，在Object类中实现的equals()方法如下：

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

也就是等同于“==”，只有在内存一样的时候才返回true。

- String类重写了这个方法，重写后的方法首先判断内存地址是否一致，如果一致返回true，否则比较字符串的内容是否一致，如果内容一致也返回true。因此，使用String类的equals方法是比较内容是否一致，而使用“==”是比较实例是否是同一个实例。
- StringBuilder类并没有重写equals方法，因此使用equals比较时，需要时同一个实例才会返回true。否则返回false。

Java创建字符串的过程

在我们使用“=”赋值时，如果内存中已经有这个字符串，就会直接将其地址给这个变量，不会产生新的字符串。

如上面代码中的“H”与“H_1”，二者指向同一个实例。

当我们使用“+”或者“concat”方法拼接字符串的时候，会创建一个新的字符串，占用新的内存空间，因此使用“==”判断时返回false。

Java 中String的引用方式

```
public class Hello {  
    public static void main(String argv[]) {  
        String sa = "abc";  
        String sb = "abc";  
        if (sa == sb) {  
            System.out.println("Yes");  
        } else {  
            System.out.println("No");  
        }  
    }  
}
```

上面这段代码的结果是Yes。

程序运行的过程是这样，先在内存中创建字符串“abc”，然后将地址的引用给了变量sa，随后又把这个地址的引用给了sb。因此sa和sb引用的是同一段内存。

由于String类是一个不可更改的类。字符串不可被更改，所以这样的方式并不会产生问题。

Python中判断字符串相等的方式

Python可以直接使用 `==` 判断字符串是否相等:

```
s = "Hello"
s1 = s
s2 = "He"
s3 = "llo"
s4 = s2+s3

print(s)    # "Hello"
print(s1)   # "Hello"
print(s2)   # "He"
print(s3)   # "llo"
print(s4)   # "Hello"

print(s == s1) # True
print(s == s2) # False
print(s == s3) # False
print(s == s4) # True
```

代码中的注释为运行结果。

C++中判断字符串相等的方式

跟Python类似, C++也可以直接使用 `==` 比较字符串是否相等。

```
string s = "Hello";
string s1 = s;
string s2 = "He";
string s3 = "llo";
string s4 = s2+s3;

cout << s << endl; // "Hello"
cout << s1 << endl; // "Hello"
cout << s2 << endl; // "He"
cout << s3 << endl; // "llo"
cout << s4 << endl; // "Hello"

cout << (s == s1) << endl; // 1
cout << (s == s2) << endl; // 0
cout << (s == s3) << endl; // 0
cout << (s == s4) << endl; // 1
```

代码中的注释为运行结果。

如何遍历字符串

Java:

```
String s = new String("Hello");
for(int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    // ...
}
```

使用上述方式来遍历Java中的字符串。

其中`s.length()` 获取字符串的长度。

`String` 不支持下标索引的方式访问，所以需要使用`charAt(i)`的方式访问对应位置的字符。同时也就没有办法使用下标的方式对`String`进行修改。

`String`是一种不可变类，字符串一但生成就不能被改变。例如我们使用`**+`进行字符串连接，会产生新的字符串，原串不会发生任何变化；使用replace()** 进行替换某些字符的时候也是产生新的字符串，不会更改原有字符串。`

Python:

```
s = "Hello"
for i in range(len(s)):
    s[i].....
#另一种写法
for c in s:
    c.....
```

使用上述方式来遍历python中的字符串。

其中`len(s)` 获取字符串的长度, 使用`s[i]`可以访问对应位置的字符。

Python中的字符串是不可变的，字符串一但生成就不能被改变，因此不能直接用`s[i]=x`的方式改变字符串。例如我们使用`**+`进行字符串连接，会产生新的字符串，原串不会发生任何变化；使用replace()** 进行替换某些字符的时候也是产生新的字符串，不会更改原有字符串。`

C++:

```
string s = "Hello";
for (int i = 0; i < s.size(); ++i) {
    s[i] ...
}
// 或者
for (char c: s) {
    c...
}
// 跟上一种写法一样，但是此时改变c的值会同时改变原字符串
for (char& c: s) {
    c...
}
```

使用上述方式来遍历python中的字符串。

其中`s.size()` 获取字符串的长度, 使用`s[i]`可以访问对应位置的字符。`c++`中的字符串是可变的，可以直接用`s[i]=x`的方式改变字符串。

null 表示空对象

Java中一切皆对象的思想，null用来表示空对象。我们不能对空对象做任何操作，除了“=”和“==”。

```
String s = null;
```

说明我们只是定义了s这个变量，只是在栈内存中标记了这个变量的存在，但是并没有实际分配任何堆内存给这个变量，变量没有指向的地址，是个空对象。

Python 中类似的概念是None。

```
s = None
```

以上代码定义了一个空对象，我们不能对这个对象做任何操作。

C++不能直接定义一个空对象，但是可以将一个引用绑定在一个nullptr上。

```
string &p = *static_cast<string *>(nullptr);
```

此时也不能对p进行任何操作。

空串

Java:

```
String s = "";
```

Python:

```
s = ""
s = str() # 等价于 s= ''
```

C++:

```
string s;
```

这个声明中，s不是空对象，是指向实实在在的堆内存的。只是这段内存中没有数据而已，s此时是个空串。

我们可以对s做所有字符串的操作。例如取长度、拼接、替换、查找字符等。

其他还有很多常见的一些 String 的函数经常用到，如：

- `substring`, 取子字符串
- `startsWith`, 判断一个字符串是否以某个字符串开头
- `endsWith`, 判断一个字符串是否以某个字符串结尾
- `compareTo`, 比较两个字符串的大小，一般用于按照字典序排序字符串
- `indexOf`, 查询一个字符串里另外一个字符串第一次出现的位置
- `lastIndexOf`, 查询一个字符串里另外一个字符串出现的最后一个位置
- `format`, 格式化字符串

请前往参考资料获得更详细的用法描述

参考资料

Java: <http://www.runoob.com/java/java-string.html>

Python: <http://www.runoob.com/python/python-strings.html>

C++: <https://www.jianshu.com/p/90584f4404d2>

与面试官沟通的基本原则是：

不要把面试官当作你的 Interviewer，而要当作你的 Co-worker

所以你们俩之间：

1. **你可以问他索要提示，但是尽可能的不要问太多提示。** 正如工作中，你可以问你的同事寻求帮助，但是你问太多，问得事无巨细人家也很烦。
2. **沟通之后再动手。** 正如工作中，你的同事和你合作的时候，不会喜欢你一声不吭的先按照自己的想法把代码写了。
3. **意见不合别吵架，先认可对方的想法。** 正如工作中，你和同事讨论一个问题的不同解决方案的时候，最好先说，我觉得你的方法挺好的，然后再说，不过我觉得有个问题。而不是：我艹你这什么sb方案 / 这肯定不对呀 / blabla。何况面试过程中，面试官是开外挂作弊知道了正确答案的一方，多认可他提出的质疑。

面试中有两类极端的求职者：

第一类：一边写一边说，生怕面试官 for 循环看不懂。

第二类：一声不吭开始写。

这两类求职者都容易挂掉面试。第一类求职者 80%+ 会不够时间写完面试题。第二类求职者 60%+ 会理解错面试官想要他实现的内容或者使用了错误的方法进行实现。

更好的办法是：

1. 首先和面试官进行算法和实现方式上的沟通，从面试官那里得到确认你的方法是OK的，写出来是可以过的。
2. 开始写代码时，只对一些可能对方不太看得懂的做解释。如果他正在玩手机没看你，就不用理他赶紧写完。
3. 写完之后再一股脑给他解释代码

这样就算你没有足够的时间解释代码，但是代码只要能写完，挂的几率就减少了很多。

算法复杂度理论

教程

问答(1)

我们用复杂度来量化一个算法的时间，空间。在这一小节中，我们讲学习什么是复杂度，什么是时间复杂度，什么是空间复杂度。

在面试中，时间复杂度是问得比较多的，空间复杂度一般不会问。

时间复杂度

教程

问答(0)

时间复杂度是面试中必问的问题。学好时间复杂度，有如下的帮助：

1. 面试官会问你的算法时间复杂度是什么
2. 当面试官说，有没有更好的方法时，你知道朝什么样的复杂度优化
3. 利用时间复杂度倒推算法是面试常用技巧。如 $O(\log N)$ 的算法几乎可以确定是二分法。

一个算法的运行时间与其所要执行的语句的数量成正比，而所要执行的语句与问题规模正相关。因此算法的时间复杂度可以表示为一个与问题规模 N 相关的多项式。

例如下面的代码：

Java版本：

```
int sum = 0;
int n = 100;
for (int i = 0; i < n; i++) {
    sum += i;
}
for (int i = 0; i < n; i++) {
    sum += i;
}
```

Python版本：

```
sum = 0
n = 100
for i in range(n):
    sum += i
for i in range(n):
    sum += i
```

这段代码的运行时间与 n 的大小正相关，因此，我们可以将其复杂度写成多项式： $f(n) = 2n$;

在计算机科学中，时间复杂度使用标记 O (大写英文字母o)表示，只包含上述多项式中的最高次项，且忽略最高次项的系数上面这段代码的时间复杂度就是： $O(n)$ 。

时间复杂度定性的描述了一个算法的运行时间。（定义参照[WIKI](#)）

更为通俗的定义是：[程序执行了多少语句](#)，一条一句被多次执行，就要算多次。

时间复杂度计算的要点：

- **只包含多项式的最高次项。** 这是因为在复杂度计算中，最高次项对运行时间有决定性的作用，次高次项可以忽略不计。例如 $f(n) = n^2 + n$ ，此时 n 这一项对于多项式的值的影响远大于 n^2 可以忽略不计。在定性的描述中，我们只取最高次项。
- **不包含多项式最高次项的系数。** 对于最高次项，我们忽略它的系数，在算法中，我们称之为常数。上面代码中，常数是2，但是时间复杂度的计算，我们只取 $O(n)$ 。

在计算算法的时间复杂度时，我们关注其中最耗时的部分，对于相对不那么耗时的部分就忽略不去考虑。也就是上面讲的“只保留最高次项”。

例子：

Java版本：

```
int sum = 0;
int n = 100;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        sum++;
    }
}
for (int i = 0; i < n; i++) {
    sum++;
}
```

Python版本：

```
sum = 0
n = 100
for i in range(n):
    for j in range(n):
        sum += 1
for i in range(n):
    sum += 1
```

对于上面这段代码，一个两重循环和一个一重循环。

两重循环的时间复杂度是： $O(n^2)$ ，

一重循环的时间复杂度是： $O(n)$ 。

那么我们标记这段代码的时间复杂度为： $O(n^2)$ ，而不去考虑那个小的 $O(n)$ 。

我们只关心算法中最耗时的部分，并且，假如代码中做了好几次两重循环，我们也不计较这个几次，而是只保留复杂度的“最高次项”，即 $O(n^2)$ 。

面试中常见算法的时间复杂度

教程

问答(7)

算法中，常见的`时间复杂度`有：

复杂度	可能对应的算法	备注
$O(1)$	位运算	常数级复杂度，一般面试中不会有
$O(\log n)$	二分法，倍增法，快速幂算法，辗转相除法	
$O(n)$	枚举法，双指针算法，单调栈算法，KMP算法，Rabin Karp，Manacher's Algorithm	又称作线性时间复杂度
$O(n \log n)$	快速排序，归并排序，堆排序	
$O(n^2)$	枚举法，动态规划，Dijkstra	
$O(n^3)$	枚举法，动态规划，Floyd	
$O(2^n)$	与组合有关的搜索问题	
$O(n!)$	与排列有关的搜索问题	

在面试中，经常会涉及到时间复杂度的计算。当你在对于一个问题给出一种解法之后，面试官常会进一步询问，是否有更优的方法。此时就是在问你是否有时间复杂度更小的方法（有的时候也要考虑空间复杂度更小的方法），这个时候需要你对常用的数据结构操作和算法的时间复杂度有清晰的认识，从而分析出可优化的部分，给出更优的算法。

例如，给定一个已经排序的数组，现在有`多次`询问，每次询问一个数字是否在这个数组中，返回True or False.

- 方法1：每次扫描一遍数组，查看是否存在。
这个方法，每次查询的时间复杂度是: $O(n)$ 。
- 方法2：由于已经有序，可以使用二分查找的方法。
这个方法，每次查询的时间复杂度是: $O(\log n)$ 。
- 方法3：将数组中的数存入Hashset。
这个方法，每次查询的时间复杂度是: $O(1)$ 。

可以看到，上述的三种方法是递进的，时间复杂度越来越小。

在面试中还有很多常见常用的方法，他们的时间复杂度并不是固定的，都需要掌握其时间复杂度的分析，要能够根据算法过程自己推算出时间复杂度。

练习 1

Java:

```
sum = 0;
for (int i = 1; i < n; i *= 2) {
    sum++;
}
```

Python:

```
sum = 0
i = 0
while i < n:
    sum += 1
    i *= 2
```

C++:

```
int sum = 0;
for (int i = 1; i < n; i *= 2) {
    sum++;
}
```

答案 

时间复杂度为 $O(\log n)$

如果你将 sum 的值打印出来，就可以发现 $\text{sum} = \log n$ （以2为底），说明 for 循环共循环了 $\log n$ 次。

练习 2

Java

```
sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j *= 2) {
        sum++;
    }
}
```

Python

```
sum = 0
for i in range(n):
    j = 1
    while j <= n:
        sum += 1
        j *= 2
```

C++

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j *= 2) {
        sum++;
    }
}
```

答案 

时间复杂度为 $O(n \log n)$

练习 3

Java:

```
int Fibo(int n) {
    if (n == 0 || n == 1) return 1;
    return Fibo(n - 1) + Fibo(n - 2);
}
```

Python:

```
def Fibo(n):
    if n == 0 or n == 1:
        return 1
    return Fibo(n-1) + Fibo(n-2)
```

C++:

```
int Fibo(int n) {
    if (n == 0 || n == 1) return 1;
    return Fibo(n - 1) + Fibo(n - 2);
}
```

答案 

时间复杂度为 $O(2^{\frac{n}{2}}) \sim O(2^n)$

计算时间复杂度上界 : $Fibo(n) = Fibon(n - 1) + Fibon(n - 2) < 2 * Fibon(n - 1)$

也就是说, 递归版 Fibonacci 的时间复杂度 $< T(n) = 2 * T(n - 1) + O(1) = O(2^n)$

再来计算时间复杂度下界 : $Fibo(n) = Fibon(n - 1) + Fibon(n - 2) > 2 * Fibon(n - 2)$

也就是说, 递归版 Fibonacci 的时间复杂度 $> T(n) = 2 * T(n - 2) + O(1) = O(2^{\frac{n}{2}})$

练习 4

Java:

```
int[] F = new int[50];
F[0] = F[1] = 1;
...
int Fibo(int n) {
    if (F[n] != 0) return F[n];
    F[n] = Fibo(n-1) + Fibo(n-2);
    return F[n];
}
```

Python:

```
F = [0] * 50
F[0] = F[1] = 1;
...
def Fibo(n):
    if F[n] != 0:
        return F[n]
    F[n] = Fibo(n-1) + Fibo(n-2)
    return F[n]
...
```

C++:

```
int F[50];
F[0] = F[1] = 1;
...
int Fibo(int n) {
    if (F[n] != 0) return F[n];
    F[n] = Fibo(n-1) + Fibo(n-2);
    return F[n];
}
```

答案 

时间复杂度为 $O(n)$

练习 5

Java:

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum++;
    }
}
```

Python:

```
sum = 0
for i in range(n):
    for j in range(i, n):
        sum += 1
```

C++:

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum++;
    }
}
```

答案

时间复杂度为 $O(n^2)$

$i = 0$ 时, j 循环执行 n 次

$i = 1$ 时, j 循环执行 $n - 1$ 次

...

$i = n - 1$ 时, j 循环执行 1 次

累加所有的执行次数, 一共执行了 $(n + n - 1 + n - 2 \dots + 1) = (1 + n) * n / 2$ 次

$O((1 + n) * n / 2) = O(n^2)$

练习 6 (难)

Java:

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && nums[j] - nums[i] < window) {
        j++;
    }
}
```

Python:

```
j = 0
for i in range(0, n):
    while j < n and nums[j]-nums[i] < window:
        j += 1
```

C++:

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && nums[j] - nums[i] < window) {
        j++;
    }
}
```

答案

时间复杂度为 $O(n)$

这个程序是典型的双指针算法， $[i, j]$ 是一个滑动窗口的两端，滑动窗口之内的数，两两之差 $< \text{window}$ 。

你可能会不理解，为什么两重循环的结果不是 $O(n^2)$ ？难道不是双重循环就是几次方么？第二重循环最坏情况不就是会执行 n 次么？

大家一定要记住，数循环次数是一个偷懒的时间复杂度计算方法，却不是最准确的时间复杂度计算方法。时间复杂度的定义，是程序总共执行的语句数目的数量级。在这个代码中，执行次数最多的是 $j++$ 这个循环主体。而这个循环体不会被执行 $O(n^2)$ 次，因为 j 在每次 i 循环的时候，不会被重置到 i 或者 0 的位置开始重新计算。 j 一直是自顾自的单向递增，那么一旦某一次 while 循环使得 $j++$ 执行了 n 次以后，while 循环就再也进不去了。因此总共的执行次数是 $O(n + n) = O(n)$ 而不是 $O(n * n)$ 。

用 T 函数表示法计算时间复杂度

教程

问答(4)

T 函数推导法

我们介绍一种时间复杂度的推导方法：[T函数推导法](#)

比如二分法。二分法是每次通过 $O(1)$ 的时间将规模为 n 的问题降低为规模为 $n/2$ 的问题。

这里我们用 $T(n)$ 来表示规模为 n 的问题在该算法下的时间复杂度，那么我们得出推导公式：

$$T(n) = T(n/2) + O(1)$$

我们来逐个说明一下这个公式的意义。

首先 T 代表的是 Time Complexity, n 代表的是问题规模（二分法里就是数组的大小）。

那么 $T(n)$ 代表的就是：求处理问题规模为 n 的数据的时间复杂度是多少。注意这里是一个问句，不是一个答案。

$T(n)$ 根据算法的不同可以是 $O(n)$ 也可以是 $O(n \log n)$ 或任何值，而 $O(n)$ 就是 $O(n)$ 。

然后 O 代表的是时间复杂度。 $O(1)$ 就意味着，你大概用一个 if 语句，或者简单的加加减减，就可以完成。 O 在这里的意思是 数量级约等于。在 O 的世界里，我们只考虑最高项是什么，不考虑系数和常数项。比如：

- $O(100n) = O(n)$
- $O(n^2 + n) = O(n^2)$
- $O(2^n + n^2 + 10) = O(2^n)$

如何推导 T 函数

我们可以使用不断展开的方法进行推导：

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ &= T(n/4) + O(1) + O(1) \\ &= T(n/8) + O(1) * 3 \\ &= T(n/16) + O(1) * 4 \\ &\dots \\ &= T(1) + O(1) * \log n \\ &= O(\log n) \end{aligned}$$

在时间复杂度的领域里，有如下的一些性质：

1. $T(1) = O(1)$ // 解决规模为 1 的问题通常时间复杂度为 $O(1)$ 。这个不 100% 对，但是 99.9% 的情况下都是如此。
2. $k * O(n) = O(kn)$
3. $O(n) + O(m) = O(n + m)$

上面的方法，是采用 T 函数展开的方法，将二分法的时间复杂度最终用 $O(\dots)$ 来表示。

练习题 1

有一个算法大致结构如下：

```
while (n > 1) {  
    这里执行一个使用 O(n) 的算法，将 n 的规模缩小一半  
    n = n / 2  
}
```

问该算法的时间复杂度

答案 

根据算法，我们可以用 T 函数推导法 写出公式： $T(n) = T(n/2) + O(n)$

推导过程如下：

$$\begin{aligned}T(n) &= T(n/2) + O(n) \\&= T(n/4) + O(n/2) + O(n) \\&= T(n/8) + O(n/4) + O(n/2) + O(n) \\&= \dots \\&= O(1) + O(2) + \dots + O(n/2) + O(n) \\&= O(1 + 2 + 4 + \dots + n/2 + n) \\&= O(2n) = O(n)\end{aligned}$$

许多同学会拍脑袋认为这个式子的结果是 $O(n\log n)$ ，这是错误的。主要错在，当 $T(n/2)$ 往下继续展开的时候，很多同学直接写成 $T(n/4) + O(n)$ ，这是不对的。应该是 $T(n/4) + O(n/2)$ 。这里我们暂时不能约掉 $O(n/2)$ 里的 $/2$ 。因为会导致误差累积。

另外一个需要记住的结论就是： $O(1 + 2 + 4 + \dots + n/2 + n) = O(n)$ 。这个结论可以通过简单的将 $n = 1024$ 带入计算可以得到：

$$1 + 2 + 4 + \dots + 1024 = 2047 \approx 2 * 1024 = O(2n) = O(n)$$

这是一道小学数学题，给整个式子 + 1，然后两个 1 就变成了一个 2，两个 2 就变成了一个 4，以此类推。

练习题 2

请用 T 函数来推导归并排序算法的时间复杂度

答案 

归并排序算法的步骤为：

- 找出数组的中点
- 将数组分成两个部分递归执行该算法，分别排序两个部分
- 合并两个排序好的子数组到一个大数组

写成 T 函数为： $T(n) = 2 * T(n/2) + O(n)$ ，其中

- $2 * T(n/2)$ 代表将 n 的问题，拆分为两个 $n/2$ 的同类问题去进行处理。
- $O(n)$ 代表了，合并两个排好序的 $n/2$ 大小的数组的时间复杂度。

我们用展开的方式推导该算法下的 $T(n)$:

$$\begin{aligned}T(n) &= 2 * T(n/2) + O(n) \\&= 2 * (2 * T(n/4) + O(n/2)) + O(n) \\&= 4 * T(n/4) + 2 * O(n/2) + O(n) \\&= 4 * T(n/4) + 2 * O(n) \\&= 4 * (2 * T(n/8) + O(n/4)) + 2 * O(n) \\&= 8 * T(n/8) + 3 * O(n) \\&= 16 * T(n/16) + 4 * O(n) \\&\dots \\&= n * T(1) + \log n * O(n) \\&= O(n) + O(n\log n) \\&= O(n\log n)\end{aligned}$$

练习题 3

如果一个算法，每次通过 $O(1)$ 的时间将 n 的问题拆分为两个 $O(n/2)$ 的问题，求时间复杂度。

答案 

首先写出推导函数： $T(n) = 2 * T(n/2) + O(1)$ ，然后进行推导：

```
T(n) = 2 * T(n/2) + O(1)
= 2 * (2 * T(n/4) + O(1)) + O(1)
= 4 * T(n/4) + O(2 + 1)
= 8 * T(n/8) + O(4 + 2 + 1)
...
= n * T(1) + O(n/2 + n/4 + ... + 2 + 1)
= O(n) + O(n)
= O(n)
```

这里 $O(n/2 + n/4 + \dots + 2 + 1) = O(n)$ 的推导和练习 1 一样。

什么是空间复杂度

教程

问答(8)

类似于时间复杂度，空间复杂度就是衡量算法运行时所占用的临时存储空间的度量。也是一个与问题规模 n 有关的函数。
我们同样使用 $O(\text{大写字母}o)$ 来标记。

算法所占用的空间主要有三个方面：算法代码本身占用的空间、输入输出数据占用的空间、算法运行时临时占用的空间。
其中，代码本身和输入输出数据占用的空间不是算法空间复杂度考虑的范围内，空间复杂度只考虑运行时临时占用的空间，又称为算法的额外空间 (Extra space)。

临时占用的空间包括：

- 为参数列表中形参变量分配的空间
- 为函数体中局部变量分配的空间
(如果是递归函数，需要将上述两部分占用空间的和乘以递归的深度，这是堆栈空间，在下面小节中详细讲解这部分)

例如：

Java:

```
public void insertionSort(int[] A) {
    int n = A.length;
    for(int i = 1; i < n; i++){
        int t = A[i];
        int index = 0;
        for (int j = i - 1; j >= 0; j--){
            if (A[j] > t){
                A[j + 1] = A[j];
            } else {
                index = j + 1;
                break;
            }
        }
        A[index] = t;
    }
}
```

Python:

```
def insertionSort(A):
    n = len(A)
    for i in range(1, n):
        t = A[i]
        index = 0
        j = i - 1
        while j >= 0:
            if A[j] > t:
                A[j + 1] = A[j]
            else:
                index = j + 1
                break
            j -= 1
        A[index] = t
    return A
```

C++:

```
void insertionSort(vector<int> A) {
    int n = A.size();
    for(int i = 1; i < n; i++){
        int t = A[i];
        int index = 0;
        for (int j = i - 1; j >= 0; j--){
            if (A[j] > t){
                A[j + 1] = A[j];
            } else {
                index = j + 1;
                break;
            }
        }
        A[index] = t;
    }
}
```

上面这段代码是插入排序的一种实现，其中主要占用的临时空间包括：变量n、t、index、i、j。int[] A是我们的输入数据，不在我们的临时空间范围内。

因此这段代码的空间复杂性函数可以写成： $f(n) = 5$ ，因为无论数组多大，变量的数量就是这么多。

这段代码的额外空间复杂度是: $O(1)$ 。

与时间复杂度一样，我们只计算最高次项，且忽略最高次项的系数。此处最高次项是 $n^0 = 1$ 。

在面试中，很多时候面试官给出的问题会附带一个“不能使用多余的空间”这样的要求。很多时候这是在要求你的空间复杂度只能是 $O(1)$ 的，也就是你只能开几个辅助变量，而不能开大数组。

其他常见的还有，分析一下你的算法空间复杂度，寻找空间复杂度更优的解法等。

一般来说，算法占用的时间和空间会是两个互相平衡的元素，有的时候我们牺牲空间来换取时间，有的时候我们牺牲时间来换取空间。

在面试中常见算法的空间复杂度：

- 快速排序：最优： $O(\log n)$ ，最差： $O(n)$
- 二分查找： $O(1)$
- 最短路(Dijkstra)算法： $O(V^2)$ (V 表示点集大小)

在递归函数中，除了变量和数组所开辟的临时空间以外，还有一个空间我们需要纳入考虑，就是递归时占用的栈空间（Stack）。

递归函数需要保存当前的环境，以便在递归返回的时候能够还原之前的现场。因此递归的深度越深，所要占用的栈空间越大。当空间超出一定范围的时候就会出现程序**爆栈**（Stack Overflow）的情况。

很多博客文章中会写堆栈空间与递归调用的次数成正比，这个是不完全正确的，应该是与递归的深度成正比（此处只讨论单线程）。

因为递归在返回到上一层的时候，就会将本层的空间释放，因此占用的栈空间不会比最深的一次调用所占用的空间更多。

大部分的空间复杂度计算方法

累加下面两个部分的内容即是你代码的空间复杂度：

1. 你的代码里开辟了多少新的空间（new 了多少新的内容出来）
2. 你的递归深度 * 递归函数内部的参数和局部变量所占用的空间

以快速排序为例

快速排序的思路如下：

1. 选择一个基准元素，将原数组分为两部分，左边部分小于该元素，右边部分大于该元素。

2. 分别递归处理左边和右边。

- 最好情况：

每次都能恰好将数组分成左右相同长度的两部分，需要的递归深度是 $\lg n$ ，每次将数组分成两部分时，我们选择不使用辅助数组，在原数组上“就地”处理，所以每层的空间是 $O(1)$ 。

因此总的复杂度是： $O(\log n)$

- 最差情况：

每次都将数组分成长度差最大的两部分，即一边只有一个元素，其余的在另外一边，最大深度为 n ，因此空间复杂度为 $O(n)$ 。

有些递归算法的空间复杂度是稳定的，不会退化，快排的递归深度与其每次选择的“基准值”有很大关系，因此存在退化的情况。

练习 1

Java:

```
int Fibo(int n) {
    if(n == 0 || n == 1) return 1;
    return Fibo(n-1) + Fibo(n-2);
}
```

Python:

```
def Fibo(n):
    if n == 0 or n == 1:
        return 1
    return Fibo(n-1) + Fibo(n-2)
```

C++:

```
int Fibo(int n) {
    if(n == 0 || n == 1) return 1;
    return Fibo(n-1) + Fibo(n-2);
}
```

答案 

O(n)

练习 2

Java:

```
int Fibo(int n) {  
    int x1 = 1, x2 = 1, ret = 1;  
    for(int i = 2; i <= n; i++){  
        ret = x1 + x2;  
        x1 = x2;  
        x2 = ret;  
    }  
    return ret;  
}
```

Python:

```
def Fibo(n):  
    x1, x2, ret = 1, 1, 1  
    for i in range(2, n+1):  
        ret = x1 + x2  
        x1, x2 = x2, ret  
    return ret
```

C++:

```
int Fibo(int n) {  
    int x1 = 1, x2 = 1, ret = 1;  
    for(int i = 2; i <= n; i++){  
        ret = x1 + x2;  
        x1 = x2;  
        x2 = ret;  
    }  
    return ret;  
}
```

答案 

O(1)

以 \sqrt{n} 为时间复杂度的算法并不多见，最具代表性的就是分解质因数了。

题目描述

<http://www.lintcode.com/problem/prime-factorization/>

具体步骤

1. 记 $up = [\sqrt{n}]$, 作为质因数k的上界, 初始化 $k = 2$ 。
2. 当 $k \leq up$ 且 n 不为1时, 执行步骤3, 否则执行步骤4。
3. 当 n 被 k 整除时, 不断整除并覆盖 n , 同时结果中记录 k , 直到 n 不能整出 k 为止。之后 k 自增, 执行步骤2。
4. 当 n 不为1时, 把 n 也加入结果当中, 算法结束。

几点解释

- 不需要判定 k 是否为质数, 如果 k 不为质数, 且能整出 n 时, n 早被 k 的因数所除。故能整除 n 的 k 必是质数。
- 为何引入 up ? 为了优化性能。当 k 大于 up 时, k 已不可能整除 n , 除非 k 是 n 自身。也即为何步骤4判断 n 是否为1, n 不为1时必是比 up 大的质数。
- 步骤2中, 也判定 n 是否为1, 这也是为了性能, 当 n 已为1时, 可早停。

代码

Java:

```
public List<Integer> primeFactorization(int n) {
    List<Integer> result = new ArrayList<>();
    int up = (int) Math.sqrt(n);

    for (int k = 2; k <= up && n > 1; ++k) {
        while (n % k == 0) {
            n /= k;
            result.add(k);
        }
    }

    if (n > 1) {
        result.add(n);
    }

    return result;
}
```

Python:

```
def primeFactorization(n):
    result = []
    up = int(math.sqrt(n));

    k = 2
    while k <= up and n > 1:
        while n % k == 0:
            n //= k
            result.append(k)
        k += 1

    if n > 1:
        result.append(n)

    return result
```

C++:

```
vector<int> primeFactorization(int n) {
    vector<int> result;
    int up = (int)sqrt(n);

    for (int k = 2; k <= up && n > 1; ++k) {
        while (n % k == 0) {
            n /= k;
            result.push_back(k);
        }
    }

    if (n > 1) {
        result.push_back(n);
    }

    return result;
}
```

复杂度分析

- 最坏时间复杂度 $O(\sqrt{n})$ 。当n为质数时，取到其最坏时间复杂度。
- 空间复杂度 $O(\log(n))$ ，当n质因数很多时，需要空间大，但总不会多于 $O(\log(n))$ 个。

延伸

质因数分解有一种更快的算法，叫做Pollard Rho快速因数分解。该算法时间复杂度为 $O(n^{1/4})$ ，其理解起来稍有难度，有兴趣的同学可以进行自学，[参考链接](#)。

我们通常所说的内存空间，包含了两个部分：栈空间（Stack space）和堆空间（Heap space）

当一个程序在执行的时候，操作系统为了让进程可以使用一些固定的不被其他进程侵占的空间用于进行函数调用，递归等操作，会开辟一个固定大小的空间（比如 8M）给一个进程使用。这个空间不会太大，否则内存的利用率就很低。这个空间就是我们说的栈空间，Stack space。

我们通常所说的栈溢出（Stack Overflow）是指在函数调用，或者递归调用的时候，开辟了过多的内存，超过了操作系统余留的那个很小的固定空间导致的。那么哪些部分的空间会被纳入栈空间呢？栈空间主要包含如下几个部分：

1. 函数的参数与返回值
2. 函数的局部变量

我们来看下面的这段代码：

Java:

```
public int f(int n) {  
    int[] nums = new int[n];  
    int sum = 0;  
    for (int i = 0; i < n; i++) {  
        nums[i] = i;  
        sum += i;  
    }  
    return sum;  
}
```

Python:

```
def f(n):  
    nums = [0]*n # 相当于Java中的new int[n]  
    sum = 0  
    for i in range(n):  
        nums[i] = i  
        sum += i  
    return sum
```

C++:

```
int f(int n) {  
    int *nums = new int[n];  
    int sum = 0;  
    for (int i = 0; i < n; i++) {  
        nums[i] = i;  
        sum += i;  
    }  
    return sum;  
}
```

根据我们的定义，参数 n，最后的函数返回值 f，局部变量 sum 都很容易的可以确认是放在栈空间里的。那么主要的难点在 nums。

这里 nums 可以理解为两个部分：

1. 一个名字叫做 nums 的局部变量，他存储了指向内存空间的一个地址（Reference），这个地址也就是 4 个字节（32位地址总线的计算机，地址大小为 4 字节）
2. new 出来的，一共有 n 个位置的整数数组，int[n]。一共有 $4 * n$ 个字节。

这里 nums 这个变量本身，是存储在栈空间的，因为他是一个局部变量。但是 nums 里存储的 n 个整数，是存储在 堆空间 里的，Heap space。他并不占用栈空间，并不会导致栈溢出。

在大多数的编程语言中，特别是 Java, Python 这样的语言中，万物皆对象，基本上每个变量都包含了变量自己和变量所指向的内存空间两个部分的逻辑含义。

来看这个例子：

Java:

```
public int[] copy(int[] nums) {
    int[] arr = new int[nums.length];
    for (int i = 0; i < nums.length; i++) {
        arr[i] = nums[i];
    }
    return arr;
}

public void main() {
    int[] nums = new int[10];
    nums[0] = 1;
    int[] new_nums = copy(nums);
}
```

Python:

```
def copy(nums):
    arr = [0]*len(nums) # 相当于Java中的new int[nums.length]
    for i in range(len(nums)):
        arr[i] = nums[i]
    return arr

# 用list comprehension实现同样功能
def copy(nums):
    arr = [x for x in nums]
    return arr

# 以下相当于Java中的main函数
if __name__ == "__main__":
    nums = [0]*10
    nums[0] = 1
    new_nums = copy(nums)
```

C++:

```
int* copy(int nums[], int length) {
    int *arr = new int[length];
    for (int i = 0; i < length; i++) {
        arr[i] = nums[i];
    }
    return arr;
}

int main() {
    int *nums = new int[10];
    nums[0] = 1;
    int *new_nums = copy(nums, 10);
    return 0;
}
```

在 copy 这个函数中，arr 是一个局部变量，他在 copy 函数执行结束之后就会被销毁。但是里面 new 出来的新数组并不会被销毁。这样，在 main 函数里，new_nums 里才会有被复制后的数组。所以可以发现一个特点：

栈空间里存储的内容，会在函数执行结束的时候被撤回

简而言之可以这么区别栈空间和堆空间：

new 出来的就放在堆空间，其他都是栈空间

什么是递归深度

递归深度就是递归函数在内存中，同时存在的最大次数。

例如下面这段求阶乘的代码：

Java:

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    }  
    return factorial(n - 1) * n;  
}
```

Python:

```
def factorial(n):  
    if n == 1:  
        return 1  
    return factorial(n-1) * n
```

C++:

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    }  
    return factorial(n - 1) * n;  
}
```

当 $n=100$ 时，递归深度就是100。一般来说，我们更关心递归深度的数量级，在该阶乘函数中递归深度是 $O(n)$ ，而在二分查找中，递归深度是 $O(\log(n))$ 。在后面的教程中，我们还会学到基于递归的快速排序、归并排序、以及平衡二叉树的遍历，这些的递归深度都是 $(O(\log(n)))$ 。注意，此处说的是递归深度，而非时间复杂度。

太深的递归会内存溢出

首先，函数本身也是在内存中占空间的，主要用于存储传递的参数，以及调用代码的返回地址。

函数的调用，会在内存的栈空间中开辟新空间，来存放子函数。递归函数更是会不断占用栈空间，例如该阶乘函数，展开到最后 $n=1$ 时，内存中会存在 `factorial(100), factorial(99), factorial(98) ... factorial(1)` 这些函数，它们从栈底向栈顶方向不断扩展。

当递归过深时，栈空间会被耗尽，这时就无法开辟新的函数，会报出 `stack overflow` 这样的错误。

所以，在考虑空间复杂度时，递归函数的深度也是要考虑进去的。

Follow up :

尾递归：若递归函数中，递归调用是整个函数体中最后的语句，且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。（上例 `factorial` 函数满足前者，但不满足后者，故不是尾递归函数）

尾递归函数的特点是：在递归展开后该函数不再做任何操作，这意味着该函数可以不等子函数执行完，自己直接销毁，这样就不再占用内存。一个递归深度 $O(n)$ 的尾递归函数，可以做到只占用 $O(1)$ 空间。这极大的优化了栈空间的利用。

但要注意，这种内存优化是由编译器决定是否要采取的，不过大多数现代的编译器会利用这种特点自动生成优化的代码。在实际工作当中，尽量写尾递归函数，是很好的习惯。

而在算法题当中，计算空间复杂度时，建议还是老老实实地算空间复杂度了，尾递归这种优化提一下也是可以，但别太在意。

Java 版本

```
public class Solution {
    /**
     * @param A an integer array sorted in ascending order
     * @param target an integer
     * @return an integer
     */
    public int findPosition(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1;
        // 要点1: start + 1 < end
        while (start + 1 < end) {
            // 要点2: start + (end - start) / 2
            int mid = start + (end - start) / 2;
            // 要点3: =, <, > 分开讨论, mid 不+1也不-1
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

        // 要点4: 循环结束后, 单独处理start和end
        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        }
        return -1;
    }
}
```

其他语言的参考代码请见：

<http://www.jiuzhang.com/solutions/binary-search/>

常见问题

Q: 为什么要用 `start + 1 < end` 而不是 `start < end` 或者 `start <= end` ?

A: 为了避免死循环。二分法的模板中，整个程序架构分为两个部分：

1. 通过 `while` 循环，将区间范围从 n 缩小到 2（只有 `start` 和 `end` 两个点）。
2. 在 `start` 和 `end` 中判断是否有解。

`start < end` 或者 `start <= end` 在寻找目标最后一次出现的位置的时候，出现死循环。

Q: 为什么明明可以 `start = mid + 1` 偏偏要写成 `start = mid`?

A: 大部分时候，`mid` 是可以 +1 和 -1 的。在一些特殊情况下，比如寻找目标的最后一次出现的位置时，当 `target` 与 `nums[mid]` 相等的时候，是不能够使用 `mid + 1` 或者 `mid - 1` 的。因为会导致漏掉解。那么为了节省 **脑力**，统一写成 `start = mid / end = mid` 并不会造成任何解的丢失，并且也不会损失效率—— $\log(n)$ 和 $\log(n+1)$ 没有区别。

许多同学在写二分法的时候，都比较习惯性的写 `while (start < end)` 这样的循环条件。这样的写法及其容易出现死循环，导致 LintCode 上的测试“超时”（Time Limit Exceeded）。

什么情况下会出现死循环？

在做 `last position of target` 这种模型下的二分法时，使用 `while (start < end)` 就容易出现超时。

在线练习：

<http://www.lintcode.com/problem/last-position-of-target/>

我们来看看会超时的代码：

Java版本：

```
int start = 0, end = nums.length - 1;
while (start < end) {
    int mid = start + (end - start) / 2;
    if (nums[mid] == target) {
        start = mid;
    } else if (nums[mid] < target) {
        start = mid + 1;
    } else {
        end = mid - 1;
    }
}
```

Python版本：

```
start, end = 0, len(nums) - 1
while start < end:
    mid = start + (end - start) // 2
    if nums[mid] == target:
        start = mid
    else if nums[mid] < target:
        start = mid + 1
    else:
        end = mid - 1
```

C++版本：

```
int start = 0, end = nums.size() - 1; // nums 是 vector<int>
while (start < end) {
    int mid = start + (end - start) / 2;
    if (nums[mid] == target) {
        start = mid;
    } else if (nums[mid] < target) {
        start = mid + 1;
    } else {
        end = mid - 1;
    }
}
```

上面这份代码是大部分同学的实现方式。看上去似乎没有太大问题。我们来注意一下 `nums[mid] == target` 时候的处理。这个时候，因为 mid

nums = [1,1], target = 1

将数据带入过一下代码：

```
start = 0, end = 1
while (0 < 1) {
    mid = 0 + (1 - 0) / 2 = 0
    if (nums[0] == 1) {
        start = 0;
    }
    ...
}
```

我们发现，start 将始终是 `0`。

出现这个问题的主要原因是， $mid = start + (end - start) / 2$ 这种写法是偏向start的。也就是说 mid 是中间偏左的位置。这样导致如果或许你会说，那么我改成 $mid = (start + end + 1) / 2$ 是否能解决问题呢？没错，确实可以解决 last position of target 的问题，但

快速幂算法

基本原理

计算 x 的 n 次方，即计算 x^n 。

由公式可知： $x^n = x^{n/2} * x^{n/2}$ 。

如果我们求得 $x^{n/2}$ ，则可以 $O(1)$ 求出 x^n ，而不需要再去循环剩下的 $n/2$ 次。

以此类推，若求得 $x^{n/4}$ ，则可以 $O(1)$ 求出 $x^{n/2}$ 。

...

因此一个原本 $O(n)$ 的问题，我们可以用 $O(\log n)$ 复杂度的算法来解决。

递归版本的快速幂算法

Java:

```
int power(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int tmp = power(x, n / 2);
        return tmp * tmp;
    } else {
        int tmp = power(x, n / 2);
        return tmp * tmp * x;
    }
}
```

Python:

```
def power(x, n):
    if n == 0:
        return 1

    if n % 2 == 0:
        tmp = power(x, n // 2)
        return tmp * tmp
    else:
        tmp = power(x, n // 2)
        return tmp * tmp * x
```

C++:

```
int power(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int tmp = power(x, n / 2);
        return tmp * tmp;
    } else {
        int tmp = power(x, n / 2);
        return tmp * tmp * x;
    }
}
```

注意:

- 不要重复计算，在计算 $x^{n/2} * x^{n/2}$ 的时候，先计算出 $x^{n/2}$ ，存下来然后返回 $tmp * tmp$;
- n为奇数的时候要记得再乘上一个x。

非递归版本

Java:

```
int power(int x, int n) {
    int ans = 1, base = x;
    while (n != 0) {
        if (n % 2 == 1) {
            ans *= base;
        }
        base *= base;
        n = n / 2;
    }
    return ans;
}
```

Python:

```
def power(x, n):
    ans = 1
    base = x
    while n > 0:
        if n % 2 == 1:
            ans *= base
        base *= base
        n = n // 2
    return ans
```

C++:

```
int power(int x, int n) {
    int ans = 1, base = x;
    while (n != 0) {
        if (n % 2 == 1) {
            ans *= base;
        }
        base *= base;
        n = n / 2;
    }
    return ans;
}
```

非递归版本与递归版本原理相同，计算顺序略有不同。

因为递归是从大问题进入，划分子问题然后层层返回再求解大问题。这里要从小问题开始，直接求解大问题。

你可以打印出每次循环中 base 和 ans 的值，来理清楚其中的算法思路。

递归版本和非递归版本都应该熟练掌握，虽然递归版本更容易掌握和理解，且 log_n 的计算深度也不会导致 Stack Overflow。但是面试官是很有可能为了加大难度让你在用非递归的版本写一遍的。

相关练习

<http://www.lintcode.com/problem/fast-power/>

<http://www.lintcode.com/problem/powx-n/>

辗转相除法

算法介绍

辗转相除法，又名欧几里德算法，是求最大公约数的一种方法。它的具体做法是：用较大的数除以较小的数，再用除数除以出现的余数（第一余数），再用第一余数除以出现的余数（第二余数），如此反复，直到最后余数是0为止。如果是求两个数的最大公约数，那么最后的除数就是这两个数的最大公约数。

代码

Java:

```
public int gcd(int big, int small) {
    if (small != 0) {
        return gcd(small, big % small);
    } else {
        return big;
    }
}
```

Python:

```
def gcd(big, small):
    if small != 0:
        return gcd(small, big % small)
    else:
        return big
```

C++:

```
int gcd(int big, int small) {
    if (small != 0) {
        return gcd(small, big % small);
    } else {
        return big;
    }
}
```

LintCode 相关练习

<http://www.lintcode.com/problem/greatest-common-divisor/>

第三章 双指针算法

本章节的先修内容：

- 什么是同向双指针，有哪些基本的同向双指针面试题？
- 什么是相向双指针，有哪些基本的相向双指针面试题？
- 最基本的两数之和问题
- 快速排序算法
- 归并排序算法
- 快速选择算法

课后补充内容：

- 三指针算法
- 烙饼排序 Pancake Sort

相向双指针

相向双指针，指的是在算法的一开始，两根指针分别位于数组/字符串的两端，并相向行走。如我们在小学的时候经常遇到的问题：

小明和小红分别在铁轨A站和B站相向而行，小红的速度为 1m/s, 小明的速度为 2m/s, A站和B站相距 1km。
请问 ... 他们什么时候被火车撞死？（逃

一个典型的相向双指针问题就是翻转字符串的问题。在第二节课中我们学到的三步翻转法，就是一个典型的例子。

用 while 循环的写法：

Java:

```
public void reverse(char[] s) {
    int left = 0, right = s.length - 1;
    while (left < right) {
        char temp = s[left];
        s[left] = s[right];
        s[right] = temp;
        left++;
        right--;
    }
}
```

Python:

```
"""
@param s: a list of characters
"""
def reverse(s):
    left, right = 0, len(s)-1
    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1
```

用 for 循环的写法：

Java:

```
public void reverse(char[] s) {
    for (int i = 0, j = s.length - 1; i < j; i++, j--) {
        char temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }
}
```

Python: 无

没有掌握三步翻转法的同学请回到第二章节的课后补充内容中进行学习。

另外一个双指针的经典练习题，就是回文串的判断问题。给一个字符串，判断这个字符串是不是回文串。

我们可以用双指针的算法轻易的解决：

Java:

```
boolean isPalindrome(String s) {
    for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
        if (s.charAt(i) != s.charAt(j)) {
            return false;
        }
    }
    return true;
}
```

Python:

```
def isPalindrome(s):
    i, j = 0, len(s)-1
    while i < j:
        if s[i] != s[j]:
            return False
        i += 1
        j -= 1
    return True
```

Follow up 1: 不区分大小写，忽略非英文字母

完整的题目描述请见：

<http://www.lintcode.com/problem/valid-palindrome/>

这个问题本身没有太大难度，只是为了给过于简单的 isPalindrome 函数增加一些实现技巧罢了。

代码上和上面的 isPalindrome 函数主要有2个区别：

1. 在 `i++` 和 `j--` 的时候，要用 `while` 循环不断的跳过非英文字母
2. 比较的时候要都变成小写之后再比较

```

class Solution:
    # @param {string} s A string
    # @return {boolean} Whether the string is a valid palindrome
    def isPalindrome(self, s):
        start, end = 0, len(s) - 1
        while start < end:
            while start < end and not s[start].isalpha() and not s[start].isdigit():
                start += 1
            while start < end and not s[end].isalpha() and not s[end].isdigit():
                end -= 1
            if start < end and s[start].lower() != s[end].lower():
                return False
            start += 1
            end -= 1
        return True

```

Follow up 2: 允许删掉一个字母（类似的，允许插入一个字母）

完整的题目描述请见：

<http://www.lintcode.com/problem/valid-palindrome-ii/>

FLAG 的面试中出现过此题。一个简单直观的粗暴想法是，既然要删除一个字母，那么我们就 for 循环枚举（Enumerate）每个字母，试试看删掉这个字母之后，该字符串是否为一个回文串。

上述粗暴算法的时间复杂度是 $O(n^2)$ ，因为 for 循环枚举被删除字母的复杂度为 $O(n)$ ，判断剩余字符构成的字符串是否为回文串的复杂度为 $O(n)$ ，总共花费 $O(n^2)$ 。这显然一猜就应该不符合面试官的要求。

正确的算法如下：

1. 依然用相向双指针的方式从两头出发，两根指针设为 L 和 R。
2. 如果 $s[L]$ 和 $s[R]$ 相同的话， $L++$, $R--$
3. 如果 $s[L]$ 和 $s[R]$ 不同的话，停下来，此时可以证明，如果能够通过删除一个字符使得整个字符串变成回文串的话，那么一定要么是 $s[L]$ ，要么是 $s[R]$ 。

简单的来说，这个算法就是依然按照原来的算法走一遍，然后碰到不一样的字符的时候，从总选一个删除，如果删除之后的字符换可以是 Palindrome 那就可以，都不行的话，那就不行。

这个需要一点数学证明来证明为什么是对的，大家可以先尝试自己证明一下，再来看下面的答案：

证明

假设从两边往中间比较的过程中，找到了第一对 $s[L] \neq s[R]$ ， L 的左边和 R 的右边都一样：

```
xyz...?...?...zyx  
  ^  ^  
  L  R
```

我们总共需要证明两件事情：

1. L 和 R 中间不存在任何字符，删除之后可以使得字符串变为回文串。
2. L 左侧（ R 右侧同理）不存在任何字符，删除之后可以使得字符串变为回文串。

先证明 1

假如被删除的字符在中间，我们用 $\$$ 来表示（ $\$$ 可以是任何字符）：

```
xyz...?.$.?.zyx  
  ^  ^  
  L  R
```

既然 $\$$ 删除之后，整个字符串是回文串，那么这个字符串左右两边必然包含 $xyz..L$ 和 $R...zyx$ 的部分（ xyz 只是一个例子，可以是任何其他的对称字符串），那又因为 $s[L] \neq s[R]$ ，所以可以知道这个字符串并不是轴对称的，也就是并不是回文串。

再证明 2

假如 L 左侧存在一个字符（是个变量，可以是任何字符），删除之后，使得整个字符串为回文串：

```
xyz.$$'?.?.$.zyx  
  ^  ^  
  L  R
```

我们将其对称的右边的位置也标记出来。如果 $\$$ 被删除之后，那么他后面紧随而来的字符 $\$'$ 就有义务和 $\$$ 的对称字符，也就是 $\$$ 相等。也就是说，'=，那么此时，我们删除 $\$$ 和 删除 $\$'$ 的效果应该是一样的。那么我们就认为这次删除相当于删除了 $\$'$ ，那么同理我们可以证明，如果 $\$$ 后面的字符分别是 $\$, \$'', \$'''$ 。可以得到 $\$ == \$' == \$'' == \$''' ...$ 一直到 $\$ == L$ 。那么此时也就是说，删除 $\$$ 的效果和删除 L 的效果是一样的。那么就证明了，删除任何 L 左侧的字符，和删除 L 没有区别，那么就证明了仍然是在 L 和 R 中去选一个删除就行了。

双指针的鼻祖：两数之和

题目描述

给一个整数数组，找到两个数使得他们的和等于一个给定的数 target。
返回这两个数。

使用哈希表来解决

Java:

```
public int[] twoSum(int[] numbers, int target) {
    HashSet<Integer> set = new HashSet<>();

    for (int i = 0; i < numbers.length; i++) {
        if (set.contains(target - numbers[i])) {
            int[] pair = new int[2];
            pair[0] = numbers[i];
            pair[1] = target - numbers[i];
            return pair;
        }
        set.add(numbers[i]);
    }

    return null;
}
```

Python:

```
def twoSum(numbers, target):
    hash_set = set()

    for i in range(len(numbers)):
        if target-numbers[i] in hash_set:
            return (numbers[i], target-numbers[i])
        hash_set.add(numbers[i])

    return None
```

我们使用一个HashSet，来记录每个值是否存在。

每次查找 $target - numbers[i]$ 是否存在，存在即说明找到了，返回两个数即可。

使用双指针算法来解决

Java:

```
public class Solution {
    public int[] twoSum(int[] numbers, int target) {
        Arrays.sort(numbers);

        int L = 0, R = numbers.length - 1;
        while (L < R) {
            if (numbers[L] + numbers[R] == target) {
                int[] pair = new int[2];
                pair[0] = numbers[L];
                pair[1] = numbers[R];
                return pair;
            }
            if (numbers[L] + numbers[R] < target) {
                L++;
            } else {
                R--;
            }
        }
        return null;
    }
}
```

Python:

```
class Solution:
    def twoSum(self, numbers, target):
        numbers.sort()

        L, R = 0, len(numbers)-1
        while L < R:
            if numbers[L]+numbers[R] == target:
                return (numbers[L], numbers[R])
            if numbers[L]+numbers[R] < target:
                L += 1
            else:
                R -= 1
        return None
```

1. 首先我们对数组进行排序。
2. 用两个指针(L, R)从左右开始：
 - 如果 $\text{numbers}[L] + \text{numbers}[R] == \text{target}$, 说明找到, 返回对应的数。
 - 如果 $\text{numbers}[L] + \text{numbers}[R] < \text{target}$, 此时L指针右移, 只有这样才可能让和更大。
 - 反之使R左移。
3. L和R相遇还没有找到就说明没有解。

两个算法的对比

1. Hash方法使用一个Hashmap结构来记录对应的数字是否出现, 以及其下标。时间复杂度为 $O(n)$ 。空间上需要开辟Hashmap来存储, 空间复杂度是 $O(n)$ 。
2. Two pointers方法, 基于有序数组的特性, 不断移动左右指针, 减少不必要的遍历, 时间复杂度为 $O(n\log n)$, 主要是排序的复杂度。但是在空间上, 不需要额外空间, 因此额外空间复杂度是 $O(1)$

同向双指针

同向双指针的问题，是指两根指针都从头出发，朝着同一个方向前进。我们通过下面 5 个题目来初步认识同向双指针：

1. 数组去重问题 Remove duplicates in an array
2. 滑动窗口问题 Window Sum
3. 两数之差问题 Two Difference
4. 链表中点问题 Middle of Linked List
5. 带环链表问题 Linked List Cycle

数组去重问题

问题描述

给你一个数组，要求去除重复的元素后，将不重复的元素挪到数组前段，并返回不重复的元素个数。

LintCode 练习地址：<http://www.lintcode.com/problem/remove-duplicate-numbers-in-array/>

问题分析

这个问题有两种做法，第一种做法比较容易想到的是，把所有的数扔到 hash 表里，然后就能找到不同的整数有哪些。但是这种做法会耗费额外空间 $O(n)$ 。面试官会追问，如何不耗费额外空间。

此时我们需要用到双指针算法，首先将数组排序，这样那些重复的整数就会被挤在一起。然后用两根指针，一根指针走得快一些遍历整个数组，另外一根指针，一直指向当前不重复部分的最后一个数。快指针发现一个和慢指针指向的数不同的数之后，就可以把这个数丢到慢指针的后面一个位置，并把慢指针++。

给一个整数数组，去除重复的元素。

你应该做这些事

1. 在原数组上操作
2. 将去除重复之后的元素放在数组的开头
3. 返回去除重复元素之后的元素个数

```
# # O(n) time, O(n) space
class Solution:
    # @param {int[]} nums an array of integers
    # @return {int} the number of unique integers
    def deduplication(self, nums):
        # Write your code here|
        d, result = {}, 0
        for num in nums:
            if num not in d:
                d[num] = True
                nums[result] = num
                result += 1

    return result
```

```
# O(nlogn) time, O(1) extra space
class Solution:
    # @param {int[]} nums an array of integers
    # @return {int} the number of unique integers
    def deduplication(self, nums):
        # Write your code here
        n = len(nums)
        if n == 0:
            return 0

        nums.sort()
        result = 1
        for i in xrange(1, n):
            if nums[i - 1] != nums[i]:
                nums[result] = nums[i]
                result += 1

    return result
```

滑动窗口问题

问题描述

求出一个数组每 k 个连续整数的和的数组。如 $\text{nums} = [1, 2, 3, 4]$, $k = 2$ 的话, window sum 数组为 $[3, 5, 7]$ 。

<http://www.lintcode.com/problem/window-sum/>

问题分析

这个问题并没有什么难度, 但是如果你过于暴力的用户 $O(n * k)$ 的算法去做是并不合适的。比如当前的 window 是 $[1, 2, 3, 4]$ 。那么当 window 从左往右移动到 $[1, 2, 3], 4$ 的时候, 整个 window 内的整数和是增加了3, 减少了1。因此只需要模拟整个窗口在滑动的过程中, 整数一进一出的变化即可。这就是滑动窗口问题。

```
class Solution:
    # @param nums {int[]} a list of integers
    # @param k {int} size of window
    # @return {int[]} the sum of element inside the window at each moving
    def winSum(self, nums, k):
        # Write your code here
        n = len(nums)
        if n < k or k <= 0:
            return []
        sums = [0] * (n - k + 1)
        for i in range(k):
            sums[0] += nums[i];

        for i in range(1, n - k + 1):
            sums[i] = sums[i - 1] - nums[i - 1] + nums[i + k - 1]

        return sums
```

两数之差问题

问题描述

在一个数组中，求出满足两个数之差等于 target 的那一对数。返回他们的下标。

LintCode 练习地址：

<http://www.lintcode.com/problem/two-sum-difference-equals-to-target/>

问题分析

作为两数之和的一个 Follow up 问题，在两数之和被问烂了以后，两数之差是经常出现的一个面试问题。

我们可以先尝试一下两数之和的方法，发现并不奏效，因为即便在数组已经排好序的前提下， $\text{nums}[i] - \text{nums}[j]$ 与 target 之间的关系并不能决定我们淘汰掉 $\text{nums}[i]$ 或者 $\text{nums}[j]$ 。

那么我们尝试一下将两根指针同向前进而不是相向而行，在 i 指针指向 $\text{nums}[i]$ 的时候，j 指针指向第一个使得 $\text{nums}[j] - \text{nums}[i] \geq |\text{target}|$ 的下标 j：

1. 如果 $\text{nums}[j] - \text{nums}[i] == |\text{target}|$ ，那么就找到答案
2. 否则的话，我们就尝试挪动 i，让 i 向右挪动一位 $\Rightarrow i++$
3. 此时我们也同时将 j 向右挪动，直到 $\text{nums}[j] - \text{nums}[i] \geq |\text{target}|$

可以知道，由于 j 的挪动不会从头开始，而是一直递增的往下挪动，那么这个时候，i 和 j 之间的两个循环的关系不是累乘关系而是叠加关系。

核心代码

Java:

```
Arrays.sort(nums);
target = Math.abs(target)

// 下面这个部分的代码是 O(n) 的
int j = 1;
for (int i = 0; i < nums.length; i++) {
    while (j < nums.length && nums[j] - nums[i] < target) {
        j++;
    }
    if (nums[j] - nums[i] == target) {
        // 找到答案!
    }
}
```

Python:

```
nums.sort()
target = abs(target)

j = 1
for i in range(len(nums)):
    while j < len(nums) and nums[j]-nums[i] < target:
        j += 1
    if nums[j]-nums[i] == target:
        # 找到答案!
```

[完整参考代码](#)

相似问题

G家的一个相似问题：找到一个数组中有多少对二元组，他们的平方差 $< \text{target}$ (target 为正整数)。

我们可以用类似放的方法来解决，首先将数组的每个数进行平方，那么问题就变成了有多少对两数之差 $< \text{target}$ 。

然后走一遍上面的这个流程，当找到一对 $\text{nums}[j] - \text{nums}[i] \geq \text{target}$ 的时候，就相当于一口气发现了：

```
nums[i + 1] - nums[i]
nums[i + 2] - nums[i]
...
nums[j - 1] - nums[i]
```

一共 $j - i - 1$ 对满足要求的二元组。累加这个计数，然后挪动 i 的位置 +1 即可。

链表中点问题

问题描述

求一个链表的中点

LintCode 练习地址：<http://www.lintcode.com/problem/middle-of-linked-list/>

问题分析

这个问题可能大家会觉得，WTF 这么简单有什么好做的？你可能的想法是：

先遍历一下整个链表，求出长度 L ，然后再遍历一下链表找到第 $L/2$ 的那个位置的节点。

但是在你抛出这个想法之后，面试官会追问你：[如果只允许遍历链表一次怎么办？](#)

可以看到这种 Follow up 并不是让你优化算法的时间复杂度，而是严格的限制了你遍历整个链表的次数。你可能会认为，这种优化有意义么？事实上是很有意义的。因为 [遍历一次](#) 这种场景，在真实的工程环境中会经常遇到，也就是我们常说的 [数据流问题](#)（Data Stream Problem）。

数据流问题 Data Stream Problem

所谓的数据流问题，就是说，你需要设计一个在线系统，这个系统不断的接受一些数据，并维护这些数据的一些信息。比如这个问题就是在数据流中维护中点在哪儿。（维护中点的意思就是提供一个接口，来获取中点）

类似的一些数据流问题还有：

1. 数据流中位数 <http://www.lintcode.com/problem/data-stream-median/>
2. 数据流最大 K 项 <http://www.lintcode.com/problem/top-k-largest-numbers-ii/>
3. 数据流高频 K 项 <http://www.lintcode.com/problem/top-k-frequent-words-ii/>

这类问题的特点都是，你 [没有机会第二次遍历所有数据](#)。上述问题部分将在《九章算法强化班》中讲解。

用双指针算法解决链表中点问题

我们可以使用双指针算法来解决链表中点的问题，更具体的，我们可以称之为 [快慢指针](#) 算法。该算法如下：

Java:

```
ListNode slow = head, fast = head.next;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
```

Python:

```
slow, fast = head, head.next
while fast != None and fast.next != None:
    slow = slow.next
    fast = fast.next.next

return slow
```

完整参考程序

在上面的程序中，我们将快指针放在第二个节点上，慢指针放在第一个节点上，while 循环中每一次快指针走两步，慢指针走一步。这样当快指针走到头的时候，慢指针就在中点了。

快慢指针的算法，在下一小节的“带环链表”中，也用到了。

一个小练习

将上述代码改为提供接口的模式，即设计一个 class，支持两个函数，一个是 [add\(node\)](#) 加入一个节点，一个是 [getMiddle\(\)](#) 求中间的那个节点。

两大经典排序算法

快速排序（Quick Sort）和归并排序（Merge Sort）是算法面试必修的两个基础知识点。很多的算法面试题，要么是直接问这两个算法，要么是这两个算法的变化，要么是用到了这两个算法中同样的思想或者实现方式，要么是挑出这两个算法中的某个步骤来考察。

本小节将从算法原理，实现，以及时间复杂度，空间复杂度、排序稳定性等方面的对比，让大家对这两个经典算法有一个更深入的理解和认识。

三指针算法

题目描述

将包含0, 1, 2三种颜色代码的数组按照颜色代码的大小排序。如 `[1, 0, 1, 0, 2] => [0, 0, 1, 1, 2]`。

LintCode 提交地址：<http://www.lintcode.com/problem/sort-colors/>

解法分析

在颜色排序（Sort Color）这个问题中，传统的双指针算法可以这么做：

1. 先用 partition 的方式区分开 0 和 1, 2
2. 再在右半部分区分开 1 和 2

这个算法不可避免的要使用两次 Partition，写两个循环。许多面试官会要求你，能否只 partition 一次，也就是只用一个循环。

用一个循环的方法如下：

<http://www.jiuzhang.com/solution/sort-colors>

分析一下核心代码部分：

Java:

```
public void sortColors(int[] a) {  
    if (a == null || a.length <= 1) {  
        return;  
    }  
  
    int pl = 0;  
    int pr = a.length - 1;  
    int i = 0;  
    while (i <= pr) {  
        if (a[i] == 0) {  
            swap(a, pl, i);  
            pl++;  
            i++;  
        } else if (a[i] == 1) {  
            i++;  
        } else {  
            swap(a, pr, i);  
            pr--;  
        }  
    }  
}
```

Python:

```
def sortColors(a):
    if not a:
        return

    pl, pr = 0, len(a)-1
    i = 0
    while i <= pr:
        if a[i] == 0:
            a[pl], a[i] = a[i], a[pl]
            pl += 1
            i += 1
        elif a[i] == 1:
            i += 1
        else:
            a[pr], a[i] = a[i], a[pr]
            pr -= 1
```

pl 和 pr 是传统的双指针，分别代表 0~pl-1 都已经是 0 了，pr+1~a.length - 1 都已经是 2 了。

另一个角度说就是，如果你发现了一个 0，就可以和 pl 上的数交换，pl 就可以 ++；如果你发现了一个 2 就可以和 pr 上的数交换 pr 就可以 --。

这样，我们用第三根指针 i 来循环整个数组。如果发现 0，就丢到左边（和 pl 交换，pl++），如果发现 2，就丢到右边（和 pr 交换，pr--），如果发现 1，就不管 (i++)

这就是三根指针的算法，两根指针在两边，一根指针扫描所有的数。

这里有一个实现上的小细节，当发现一个 0 丢到左边的时候，i 需要 ++，但是发现一个 2 丢到右边的时候，i 不用 ++。原因是，从 pr 换过来的数有可能是 0 或者 2，需要继续判断丢到左边还是右边。而从 pl 换过来的数，要么是 0 要么是 1，不需要再往右边丢了。因此这里 i 指针还有一个角度可以理解为，i 指针的左侧，都是 0 和 1。

类似的题

G 家问过一个类似的题：给出 low, high 和一个数组，将数组分为三个部分， $< \text{low}$, $\geq \text{low} \& \leq \text{high}$, $> \text{high}$ 。解法和本题一模一样

烙饼排序是说，如果有一个操作 flip(arr, i)，能够在 O(1) 的时间内将数组 arr 的前 i 个数进行翻转。你能否用这个操作来实现一个排序算法？因为 flip 操作就像是一沓烙饼伸一个铲子进去然后把最上面的 i 个烙饼翻转了一下，因此叫做烙饼排序 (Pancake Sorting)。

GeeksforGeeks 上已经有比较好的讲解和代码：

<https://www.geeksforgeeks.org/pancake-sorting/>

LintCode 练习地址：

<http://www.lintcode.com/problem/pancake-sorting/>

第四章 BFS与拓扑排序

在这一章节中，我们将学习一个一天就能够学会的，面试中又极高频会考到的知识点——宽度优先搜索。英文全称 Breadth First Search，简称 BFS。

BFS是一个你必须掌握的算法，而且也很容易掌握，每个 BFS 的题目，程序写起来都差不多。因此是一个学习起来性价比极高的算法。

本章节的先修内容有

- 什么是队列，如何自己实现一个队列
- 什么是 Interface，LinkedList 和 Queue 之间的关系是什么？
- 什么是拓扑排序
- 如何定义一个图的数据结构

课后补充内容有：

- 宽度优先搜索（BFS）的另外两种实现方式
- 双向宽度优先搜索算法（Bidirectional BFS）

什么是队列（Queue）？

队列（queue）是一种采用先进先出（FIFO, first in first out）策略的抽象数据结构。比如生活中排队，总是按照先来的先服务，后来的后服务。队列在数据结构中举足轻重，其在算法中应用广泛，最常用的就是在宽度优先搜索(BFS) 中，记录待扩展的节点。

队列内部存储元素的方式，一般有两种，数组（array）和链表（linked list）。两者的最主要区别是：

- 数组对随机访问有较好性能。
- 链表对插入和删除元素有较好性能。

在各语言的标准库中：

- Java常用的队列包括如下几种：
`ArrayDeque`：数组存储。实现Deque接口，而Deque是Queue接口的子接口，代表双端队列（double-ended queue）。
`LinkedList`：链表存储。实现List接口和Deque接口，不仅可做队列，还可以作为双端队列，或栈（stack）来使用。
- C++中，使用<queue>中的`queue`模板类，模板需两个参数，元素类型和容器类型，元素类型必要，而容器类型可选，默认`deque`，可改用`list`（链表）类型。
- Python中，使用`collections.deque`，双端队列。

如何自己用数组实现一个队列？

队列的主要操作有：

- `add()` 队尾追加元素
- `poll()` 弹出队首元素
- `size()` 返回队列长度
- `empty()` 判断队列为空

下面利用Java的`ArrayList`和一个头指针实现一个简单的队列。（注意：为了将重点放在实现队列上，做了适当简化。该队列仅支持整数类型，若想实现泛型，可用反射机制和object对象传参；此外，可多做安全检查并抛出异常）

Java:

```
class MyQueue {
    private ArrayList<Integer> elements; // 用ArrayList存储队列内部元素
    private int pointer; // 表示队头的位置

    // 队列初始化
    public MyQueue() {
        this.elements = new ArrayList<>();
        pointer = 0;
    }

    // 获取队列中元素个数
    public int size() {
        return this.elements.size() - pointer;
    }

    // 判断队列是否为空
    public boolean empty() {
        return this.size() == 0;
    }

    // 在队尾添加一个元素
    public void add(Integer e) {
        this.elements.add(e);
    }

    // 弹出队首元素，如果为空则返回null
    public Integer poll() {
        if (this.empty()) {
            return null;
        }
        return this.elements.get(pointer++);
    }
}
```

Python:

```
class MyQueue:
    # 队列初始化
    def __init__(self):
        self.elements = [] # 用list存储队列元素
        self.pointer = 0 # 队头位置

    # 获取队列中元素个数
    def size(self):
        return len(self.elements)-pointer

    # 判断队列是否为空
    def empty(self):
        return self.size() == 0

    # 在队尾添加一个元素
    def add(self, e):
        self.elements.append(e)

    # 弹出队首元素，如果为空则返回None
    def poll(self):
        if self.empty():
            return None
        pointer += 1
        return self.elements[pointer-1]
```

队列在工业界的应用

队列可用于实现消息队列（message queue），以完成异步（asynchronous）任务。

“消息”是计算机间传送的数据，可以只包含文本；也可复杂到包含嵌入对象。当消息“生产”和“消费”的速度不一致时，就需要消息队列，临时保存那些已经发送而并未接收的消息。例如集体打包调度，服务器繁忙时的任务处理，事件驱动等等。

常用的消息队列实现包括RabbitMQ，ZeroMQ等等。

更多消息队列的参考资料：

[为什么需要消息队列，及使用消息队列的好处？](#)

[RabbitMQ的应用场景以及基本原理介绍](#)

Java 中的 Interface

什么是 Interface

Java接口(Interface)是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为。打一个比方，接口好比一个戏中的角色，这个角色有一些特定的属性和操作，然后实现接口的类就好比扮演这个角色的人，一个角色可以由不同的人来扮演，而不同的演员之间除了扮演一个共同的角色之外，并不要求其它的共同之处。

有哪些面试常用的 Interface

Set

注重独一无二,该体系集合可以知道某物是否已经存在于集合中,不会存储重复的元素。Set的实现类在面试中常用的是：**HashSet** 与 **TreeSet**

- **HashSet**

- 无重复数据
- 可以有空数据
- 数据无序

```
Set<String> set = new HashSet<>();
for (int i = 1; i < 6; i++) {
    set.add(i + "");
}
set.add("1"); //不会重复写入数据
set.add(null); //可以写入空数据
Iterator<String> iter = set.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " ");
} // 输出(无序)为 3 4 1 5 null 2
```

- **TreeSet**

- 无重复数据
- 不能有空数据
- 数据有序

```
Set<String> set = new TreeSet<>();
for (int i = 1; i < 6; i++) {
    set.add(i + "");
}
set.add("1"); //不会重复写入数据
//set.add(null); //不可以写入空数据
Iterator<String> iter = set.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " ");
} // 输出(有序)为 1 2 3 4 5
```

Map

Map用于存储具有映射关系的数据。Map中存了两组数据(`key`与`value`)，它们都可以是任何引用类型的数据，`key`不能重复，我们可以通过`key`取到对应的`value`。Map的实现类在面试中常用是：**HashMap** 和 **TreeMap**.

• HashMap

- `key` 无重复，`value` 允许重复
- 允许 `key` 和 `value` 为空
- 数据无序

```
public class Solution {  
    public static void main(String[] args){  
        Map<String, String> map = new HashMap<>();  
        for (int i = 5; i > 0; i --) {  
            map.put(i + "", i + "");  
        }  
        map.put("1","1");//key无重复  
        map.put("11","1");//value可以重复  
        map.put(null, null);//可以为空  
        for (Iterator i = map.keySet().iterator(); i.hasNext(); ) {  
            String key = (String)i.next();  
            String value = map.get(key);  
            System.out.println("key = " + key + ", value = " + value);  
        }  
    }  
}  
/*  
key = 11, value = 1  
key = null, value = null  
key = 1, value = 1  
key = 2, value = 2  
key = 3, value = 3  
key = 4, value = 4  
key = 5, value = 5  
*/  
//输出顺序与输入顺序无关
```

• TreeMap

- `key` 无重复，`value` 允许重复
- 不允许有null
- 有序(存入元素的时候对元素进行自动排序，迭代输出的时候就按排序顺序输出)

```

public static void main(String[] args){
    Map<String, String> map = new TreeMap<>();
    for (int i = 5; i > 0; i --) {
        map.put(i + "", i + "");
    }
    map.put("1","1");//key无重复
    map.put("11","1");//value可以重复
    //map.put(null, null);//不可以为空
    for (Iterator i = map.keySet().iterator(); i.hasNext(); ) {
        String key = (String)i.next();
        String value = map.get(key);
        System.out.println("key = " + key + ", value = " + value);
    }
}
//输出
/*
key = 1, value = 1
key = 11, value = 1
key = 2, value = 2
key = 3, value = 3
key = 4, value = 4
key = 5, value = 5
*/
//输出顺序位String排序后的顺序

```

List

一个 List 是一个元素有序的、可以重复(这一点与Set和Map不同)、可以为 null 的集合，List的实现类在面试中常用是：**LinkedList** 和 **ArrayList**

- **LinkedList**
 - 基于链表实现
- **ArrayList**
 - 基于动态数组实现
- **LinkedList** 与 **ArrayList** 对比：
 - 对于随机访问 `get` 和 `set`，ArrayList绝对优于LinkedList，因为LinkedList要移动指针
 - 对于新增和删除操作 `add` 和 `remove`，LinkedList比较占优势，因为ArrayList要移动数据

Queue

队列是一种比较重要的数据结构，它支持FIFO(First in First out)，即尾部添加、头部删除（先进队列的元素先出队列），跟我们生活中的排队类似。

- **PriorityQueue**
 - 基于堆(heap)实现
 - 非FIFO(最先出队列的是优先级最高的元素)
- **LinkedList**
 - 基于链表实现
 - FIFO

什么时候使用宽搜

如下的一些场景是使用宽度优先搜索的常见场景：

图的遍历 Traversal in Graph

图的遍历，比如给出无向连通图(Undirected Connected Graph)中的一个点，找到这个图里的所有点。这就是一个常见的场景。

LintCode 上的 [Clone Graph](#) 就是一个典型的练习题。

更细一点的划分的话，这一类的问题还可以分为：

- 层级遍历 Level Order Traversal
- 由点及面 Connected Component
- 拓扑排序 Topological Sorting

层级遍历，也就是说我不仅仅需要知道从一个点出发可以到达哪些点，还需要知道这些点，分别离出发点是第几层遇到的，比如 [Binary Tree Level Order Traversal](#) 就是一个典型的练习题。

由点及面，前面已经提到。

拓扑排序，让我们在后一节中展开描述。

最短路径 Shortest Path in Simple Graph

最短路径算法有很多种，BFS 是其中一种，但是他有特殊的使用场景，即必须是在简单图中求最短路径。

大部分简单图中使用 BFS 算法时，都是无向图。当然也有可能是有向图，但是在面试中极少会出现。

什么是简单图（Simple Graph）？

即，图中每条边长度都是1（或边长都相同）。

图上的宽度优先搜索

BFS 大部分的时候是在图上进行的。

什么是图 (Graph)

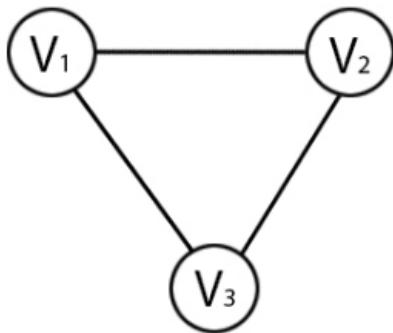
图在离线数据中的表示方法为 `<E, V>`，E 表示 Edge，V 表示 Vertex。也就是说，图是顶点 (Vertex) 和边 (Edge) 的集合。

图分为：

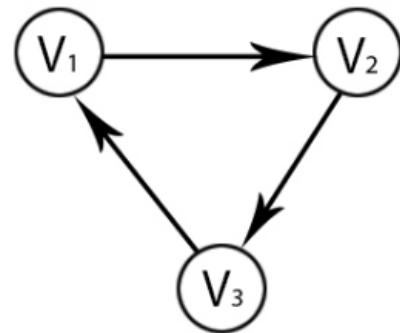
- 有向图 (Directed Graph)
- 无向图 (Undirected Graph)

见下图

Undirected Graph



Directed Graph



BFS 在两种图上都适用。另外，树 (Tree) 也是一种特殊的图。

二叉树的BFS vs 图的BFS

二叉树中进行 BFS 和图中进行 BFS 最大的区别就是二叉树中无需使用 HashSet (C++: unordered_map, Python: dict) 来存储访问过的节点 (丢进过 queue 里的节点)

因为二叉树这种数据结构，上下层关系分明，没有环 (circle)，所以不可能出现一个节点的儿子的儿子是自己的情况。
但是在图中，一个节点的邻居的邻居就可能是自己了。

如何定义一个图的数据结构？

有很多种方法可以存储一个图，最常用的莫过于：

1. 邻接矩阵
2. 邻接表

而邻接矩阵因为耗费空间过大，我们通常在工程中都是使用邻接表作为图的存储结构。

邻接矩阵 Adjacent Matrix

```
[  
    [1, 0, 0, 1],  
    [0, 1, 1, 0],  
    [0, 1, 1, 0],  
    [1, 0, 0, 1]  
]
```

例如上图表示0号点和3号点有连边。1号点和2号店有连边。

当然，每个点和自己也是默认有连边的。

图中的 `0` 表示不连通，`1` 表示连通。

我们也可以用一个更具体的整数值来表示连边的长度。

邻接矩阵我们可以直接用一个二维数组表示，如 `int[][] matrix;`。这种数据结构因为耗费 $O(n^2)$ 的空间，所以在稀疏图上浪费很大，因此并不常用。

邻接表 (Adjacent List)

```
[  
    [1],  
    [0, 2, 3],  
    [1],  
    [1]  
]
```

这个图表示 0 和 1 之间有连边，1 和 2 之间有连边，1 和 3 之间有连边。即每个点上存储自己有哪些邻居（有哪些连通的点）。

这种方式下，空间耗费和边数成正比，可以记做 $O(m)$ ， m 代表边数。 m 最坏情况下虽然也是 $O(n^2)$ ，但是邻接表的存储方式大部分情况下会比邻接矩阵更省空间。

自定义邻接表

可以用自定义的类来实现邻接表

Java:

```
class DirectedGraphNode {  
    int label;  
    List<DirectedGraphNode> neighbors;  
    ...  
}
```

Python:

```
def DirectedGraphNode:  
    def __init__(self, label):  
        self.label = label  
        self.neighbors = [] # a list of DirectedGraphNode's  
        ...
```

使用 Map 和 Set (面试时)

也可以使用 HashMap 和 HashSet 搭配的方式来存储邻接表

Java:

```
Map<T, Set<T>> = new HashMap<Integer, HashSet<Integer>>();
```

Python:

```
# 假设nodes为节点标签的列表:  
  
# 使用了Python中的dictionary comprehension语法  
adjacency_list = {x:set() for x in nodes}  
  
# 另一种写法  
adjacency_list = {}  
for x in nodes:  
    adjacency_list[x] = set()
```

其中 T 代表节点类型。通常可能是整数(Integer)。

这种方式虽然没有上面的方式更加直观和容易理解，但是在面试中比较节约代码量。

而自定义的方法，更加工程化，所以在面试中如果时间不紧张题目不难的情况下，推荐使用自定义邻接表的方式。

拓扑排序

定义

在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序（英语：Topological sorting）。

- 每个顶点出现且只出现一次；
- 若A在序列中排在B的前面，则在图中不存在从B到A的路径。

也可以定义为：拓扑排序是对有向无环图的顶点的一种排序，它使得如果存在一条从顶点A到顶点B的路径，那么在排序中B出现在A的后面。

(来自 [Wiki](#))

实际运用

拓扑排序 Topological Sorting 是一个经典的图论问题。在实际的运用中，拓扑排序可以做如下的一些事情：

- 检测编译时的循环依赖
- 制定有依赖关系的任务的执行顺序

拓扑排序不是一种排序算法

虽然名字里有 Sorting，但是相比起我们熟知的 Bubble Sort, Quick Sort 等算法，Topological Sorting 并不是一种严格意义上的 Sorting Algorithm。

确切的说，一张图的拓扑序列可以有很多个，也可能没有。拓扑排序只需要找到其中 **一个** 序列，无需找到 **所有** 序列。

入度与出度

在介绍算法之前，我们先介绍图论中的一个基本概念，**入度** 和 **出度**，英文为 in-degree & out-degree。

在有向图中，如果存在一条有向边 A->B，那么我们认为这条边为 A 增加了一个出度，为 B 增加了一个入度。

算法流程

拓扑排序的算法是典型的宽度优先搜索算法，其大致流程如下：

1. 统计所有点的入度，并初始化拓扑序列为**空**。
2. 将所有入度为 0 的点，也就是那些没有任何 **依赖** 的点，放到宽度优先搜索的队列中
3. 将队列中的点一个一个的释放出来，放到拓扑序列中，每次释放出某个点 A 的时候，就访问 A 的相邻点（所有 A 指向的点），并把这些点的入度减去 1。
4. 如果发现某个点的入度被减去 1 之后变成了 0，则放入队列中。
5. 直到队列为空时，算法结束，

深度优先搜索的拓扑排序

深度优先搜索也可以做拓扑排序，不过因为不容易理解，也并不推荐作为拓扑排序的主流算法。

这是 LintCode 上的题目：Topological Sorting

参考程序

```
public class Solution {
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {
        // map 用来存储所有节点的入度，这里主要统计各个点的入度
        HashMap<DirectedGraphNode, Integer> map = new HashMap();
        for (DirectedGraphNode node : graph) {
            for (DirectedGraphNode neighbor : node.neighbors) {
                if (map.containsKey(neighbor)) {
                    map.put(neighbor, map.get(neighbor) + 1);
                } else {
                    map.put(neighbor, 1);
                }
            }
        }

        // 初始化拓扑序列为空
        ArrayList<DirectedGraphNode> result = new ArrayList<DirectedGraphNode>();

        // 把所有入度为0的点，放到BFS专用的队列中
        Queue<DirectedGraphNode> q = new LinkedList<DirectedGraphNode>();
        for (DirectedGraphNode node : graph) {
            if (!map.containsKey(node)) {
                q.offer(node);
                result.add(node);
            }
        }

        // 每次从队列中拿出一个点放到拓扑序列里，并将该点指向的所有点的入度减1
        while (!q.isEmpty()) {
            DirectedGraphNode node = q.poll();
            for (DirectedGraphNode n : node.neighbors) {
                map.put(n, map.get(n) - 1);
                // 减去1之后入度变为0的点，也放入队列
                if (map.get(n) == 0) {
                    result.add(n);
                    q.offer(n);
                }
            }
        }

        return result;
    }
}
```

宽度优先搜索的模板

宽度优先搜索有很多种实现方法，这里为了大家记忆方便和教学的方便，我们只介绍最实用的一种方法，即使用一个队列的方法。这种方法也根据 BFS 时的需求不同，有两个版本，即需要分层遍历的版本和不需要分层遍历的版本。

什么时候需要分层遍历？

1. 如果问题需要你区分开不同层级的结果信息，如 [二叉树的分层遍历 Binary Tree Level Order Traversal](#)
2. 简单图最短路径问题，如 [单词接龙 Word Ladder](#)

无需分层遍历的宽度优先搜索

Java:

```
// T 指代任何你希望存储的类型
Queue<T> queue = new LinkedList<>();
Set<T> set = new HashSet<>();

set.add(start);
queue.offer(start);
while (!queue.isEmpty()) {
    T head = queue.poll();
    for (T neighbor : head.neighbors) {
        if (!set.contains(neighbor)) {
            set.add(neighbor);
            queue.offer(neighbor);
        }
    }
}
```

Python:

```
from collections import deque

queue = deque()
seen = set() #等价于Java版本中的set

seen.add(start)
queue.append(start)
while len(queue):
    head = queue.popleft()
    for neighbor in head.neighbors:
        if neighbor not in seen:
            seen.add(neighbor)
            queue.append(neighbor)
```

上述代码中：

- neighbor 表示从某个点 head 出发，可以走到的下一层的节点。
- set 存储已经访问过的节点（已经丢到 queue 里去过的节点）
- queue 存储等待被拓展到下一层的节点
- set 与 queue 是一对好基友，无时无刻都一起出现，往 queue 里新增一个节点，就要同时丢到 set 里。

需要分层遍历的宽度搜先搜索

Java:

```
// T 指代任何你希望存储的类型
Queue<T> queue = new LinkedList<>();
Set<T> set = new HashSet<>();

set.add(start);
queue.offer(start);
while (!queue.isEmpty()) {
    int size = queue.size();
    for (int i = 0; i < size; i++) {
        T head = queue.poll();
        for (T neighbor : head.neighbors) {
            if (!set.contains(neighbor)) {
                set.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }
}
```

Python :

```
from collections import deque

queue = deque()
seen = set()

seen.add(start)
queue.append(start)
while len(queue):
    size = len(queue)
    for _ in range(size):
        head = queue.popleft()
        for neighbor in head.neighbors:
            if neighbor not in seen:
                seen.add(neighbor)
                queue.append(neighbor)
```

上述代码中：

- `size = queue.size()` 是一个必须的步骤。如果在 `for` 循环中使用 `for (int i = 0; i < queue.size(); i++)` 会出错，因为 `queue.size()` 是一个动态变化的值。所以必须先把当前层一共有多少个节点存在局部变量 `size` 中，才不会把下一层的节点也在当前层进行扩展。

使用两个队列的BFS实现

我们可以将当前层的所有节点存在第一个队列中，然后拓展（Extend）出的下一层节点存在另外一个队列中。来回迭代，逐层展开。

参考代码如下：

Java:

```
// T 表示任意你想存储的类型
Queue<T> queue1 = new LinkedList<>();
Queue<T> queue2 = new LinkedList<>();
queue1.offer(startNode);
int currentLevel = 0;

while (!queue1.isEmpty()) {
    int size = queue1.size();
    for (int i = 0; i < size; i++) {
        T head = queue1.poll();
        for (all neighbors of head) {
            queue2.offer(neighbor);
        }
    }
    Queue<T> temp = queue1;
    queue1 = queue2;
    queue2 = temp;

    queue2.clear();
    currentLevel++;
}
```

Python:

```
from collections import deque

queue1, queue2 = deque(), deque()
seen = set()

seen.add(start)
queue1.append(start)
currentLevel = 0
while len(queue1):
    size = len(queue1)
    for _ in range(size):
        head = queue1.popleft()
        for neighbor in head.neighbors:
            if neighbor not in seen:
                seen.add(neighbor)
                queue2.append(neighbor)
    queue1, queue2 = queue2, queue1
    queue2.clear()
    currentLevel += 1
```

使用 Dummy Node 进行 BFS

什么是 Dummy Node

Dummy Node，翻译为哨兵节点。Dummy Node 一般本身不存储任何实际有意义的值，通常用作“占位”，或者链表的“虚拟头”。

如很多的链表问题中，我们会在原来的头head的前面新增一个节点，这个节点没有任何值，但是 next 指向 head。这样就会方便对 head 进行删除或者在前面插入等操作。

```
head->node->node->node ...
=>
dummy->head->node->node->node...
```

Dummy Node 在 BFS 中如何使用

在 BFS 中，我们主要用 dummy node 来做占位符。即，在队列中每一层节点的结尾，都放一个 `null` (or `None` in Python, `nil` in Ruby)，来表示这一层的遍历结束了。这里 dummy node 就是一个 `null`。

Java:

```
// T 可以是任何你想存储的节点的类型
Queue<T> queue = new LinkedList<>();
queue.offer(startNode);
queue.offer(null);
currentLevel = 0;
// 因为有 dummy node 的存在，不能再用 isEmpty 了来判断是否还有没有拓展的节点了
while (queue.size() > 1) {
    T head = queue.poll();
    if (head == null) {
        currentLevel++;
        queue.offer(null);
        continue;
    }
    for (all neighbors of head) {
        queue.offer(neighbor);
    }
}
```

Python:

```
from collections import deque

queue = deque()
seen = set()

seen.add(start)
queue.append(start)
queue.append(None)
currentLevel = 0
while len(queue) > 1:
    head = queue.popleft()
    if head == None:
        currentLevel += 1
        queue.append(None)
        continue
    for neighbor in head.neighbors:
        if neighbor not in seen:
            seen.add(neighbor)
            queue.append(neighbor)
```

双向宽度优先搜索算法

双向宽度优先搜索 (Bidirectional BFS) 算法适用于如下的场景：

1. 无向图
2. 所有边的长度都为 1 或者长度都一样
3. 同时给出了起点和终点

以上 3 个条件都满足的时候，可以使用双向宽度优先搜索来求出起点和终点的最短距离。

算法描述

双向宽度优先搜索本质上还是BFS，只不过变成了起点向终点和终点向起点同时进行扩展，直至两个方向上出现同一个子节点，搜索结束。我们还是可以利用队列来实现：一个队列保存从起点开始搜索的状态，另一个保存从终点开始的状态，两边如果相交了，那么搜索结束。起点到终点的最短距离即为起点到相交节点的距离与终点到相交节点的距离之和。

Q. 双向BFS是否真的能提高效率？

假设单向BFS需要搜索 N 层才能到达终点，每层的判断量为 X ，那么总的运算量为 X^N 。如果换成是双向BFS，前后各自需要搜索 $N/2$ 层，总运算量为 $2 * X^{N/2}$ 。如果 N 比较大且 X 不为 1，则运算量相较于单向BFS可以大大减少，差不多可以减少到原来规模的根号的量级。

参考代码

Java:

```
/*
 * Definition for graph node.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) {
 *         label = x; neighbors = new ArrayList<UndirectedGraphNode>();
 *     }
 * }
 */
public int doubleBFS(UndirectedGraphNode start, UndirectedGraphNode end) {
    if (start.equals(end)) {
        return 1;
    }
    // 起点开始的BFS队列
    Queue<UndirectedGraphNode> startQueue = new LinkedList<>();
    // 终点开始的BFS队列
    Queue<UndirectedGraphNode> endQueue = new LinkedList<>();
    startQueue.add(start);
    endQueue.add(end);
    int step = 0;
    // 记录从起点开始访问到的节点
    Set<UndirectedGraphNode> startVisited = new HashSet<>();
    // 记录从终点开始访问到的节点
    Set<UndirectedGraphNode> endVisited = new HashSet<>();
    startVisited.add(start);
    endVisited.add(end);
    while (!startQueue.isEmpty() || !endQueue.isEmpty()) {
        int startSize = startQueue.size();
        int endSize = endQueue.size();
        // 按层遍历
        step++;
        for (int i = 0; i < startSize; i++) {
            UndirectedGraphNode cur = startQueue.poll();
            for (UndirectedGraphNode neighbor : cur.neighbors) {
                if (startVisited.contains(neighbor)) { //重复节点
                    continue;
                } else if (endVisited.contains(neighbor)) { //相交
                    return step;
                } else {
                    startVisited.add(neighbor);
                    startQueue.add(neighbor);
                }
            }
        }
        step++;
        for (int i = 0; i < endSize; i++) {
            UndirectedGraphNode cur = endQueue.poll();
            for (UndirectedGraphNode neighbor : cur.neighbors) {
                if (endVisited.contains(neighbor)) {
                    continue;
                } else if (startVisited.contains(neighbor)) {
                    return step;
                } else {
                    endVisited.add(neighbor);
                    endQueue.add(neighbor);
                }
            }
        }
    }
    return -1; // 不连通
}
```

Python:

```
from collections import deque

def doubleBFS(start, end):
    if start == end:
        return 1

    # 分别从起点和终点开始的两个BFS队列
    startQueue, endQueue = deque(), deque()
    startQueue.append(start)
    endQueue.append(end)
    step = 0

    # 从起点开始和从终点开始分别访问过的节点集合
    startVisited, endVisited = set(), set()
    startVisited.add(start)
    endVisited.add(end)
    while len(startQueue) and len(endQueue):
        startSize, endSize = len(startQueue), len(endQueue)
        # 按层遍历
        step += 1
        for _ in range(startSize):
            cur = startQueue.popleft()
            for neighbor in cur.neighbors:
                if neighbor in startVisited: # 重复节点
                    continue
                elif neighbor in endVisited: # 相交
                    return step
                else:
                    startVisited.add(neighbor)
                    startQueue.append(neighbor)
        step += 1
        for _ in range(endSize):
            cur = endQueue.popleft()
            for neighbor in cur.neighbors:
                if neighbor in endVisited:
                    continue
                elif neighbor in startVisited:
                    return step
                else:
                    endVisited.add(neighbor)
                    endQueue.append(neighbor)

    return -1
```

学习建议

Bidirectional BFS 掌握起来并不是很难，算法实现上稍微复杂了一点（代码量相对单向 BFS 翻倍），掌握这个算法一方面加深对普通 BFS 的熟练程度，另外一方面，基本上写一次就能记住，如果在面试中被问到了如何优化 BFS 的问题，Bidirectional BFS 几乎就是标准答案了。

第五章 二叉树和基于树的DFS

在这一章节的学习中，我们将要学习一个数据结构——[二叉树](#)（Binary Tree），和基于二叉树上的搜索算法。

在二叉树的搜索中，我们主要使用了分治法（Divide Conquer）来解决大部分的问题。之所以大部分二叉树的问题可以使用分治法，是因为二叉树这种数据结构，是一个天然就帮你做好了分治法中“分”这个步骤的结构。

本章节的先修内容有：

- 什么是递归（Recursion）——请回到第二章节中复习
- 递归（Recursion）、回溯（Backtracking）和搜索（Search）的联系和区别
- 分治法（Divide and Conquer）和遍历法（Traverse）的联系和区别
- 什么是结果类 ResultType，什么时候使用 ResultType
- 什么是二叉查找树（Binary Search Tree）
- 什么是平衡二叉树（Balanced Binary Tree）

本章节的补充内容有：

- Morris 算法：使用 $O(1)$ 的额外空间复杂度对二叉树进行先序遍历（Preorder Traversal）
- 用非递归的方法实现先序遍历，中序遍历和后序遍历
- 二叉查找树（Binary Search Tree）的增删查改
- Java 自带的平衡排序二叉树 TreeMap / TreeSet 的介绍和面试中的应用

二叉树上的遍历法

定义

遍历（Traversal），顾名思义，就是[通过某种顺序，一个一个访问一个数据结构中的元素](#)。比如我们如果需要遍历一个数组，无非就是要从前往后，要么从后往前遍历。但是对于一棵二叉树来说，他就有很多种方式进行遍历：

- 层序遍历（Level order）
- 先序遍历（Pre order）
- 中序遍历（In order）
- 后序遍历（Post order）

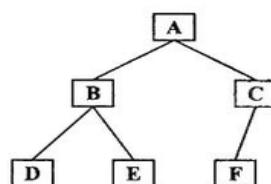
我们在之前的课程中，已经学习过了二叉树的层序遍历，也就是使用 BFS 算法来获得二叉树的分层信息。通过 BFS 获得的顺序我们也可以称之为 BFS Order。而剩下的三种遍历，都需要通过深度优先搜索的方式来获得。而这一小节中，我们将讲一下通过深度优先搜索（DFS）来获得的节点顺序，

先序遍历 / 中序遍历 / 后序遍历

先序遍历（又叫先根遍历、前序遍历）

首先访问根结点，然后遍历左子树，最后遍历右子树。[遍历左、右子树时，仍按先序遍历](#)。若二叉树为空则返回。

该过程可简记为**根左右**，注意该过程是**递归的**。如图先序遍历结果是：**ABDECF**。



核心代码：

Java:

```
// 将根作为root，空ArrayList作为result传入，即可得到整棵树的遍历结果
private void traverse(TreeNode root, ArrayList<Integer> result) {
    if (root == null) {
        return;
    }
    result.add(root.val);
    traverse(root.left, result);
    traverse(root.right, result);
}
```

Python:

```
# 将根作为root，空list作为result传入，即可得到整棵树的遍历结果
def traverse(root, result):
    if not root:
        return
    result.append(root.val)
    traverse(root.left, result)
    traverse(root.right, result)
```

[相关练习及完整答案。](#)

中序遍历（又叫中根遍历）

首先遍历左子树，然后访问根结点，最后遍历右子树。**遍历左、右子树时，仍按中序遍历。**若二叉树为空则返回。简记为**左根右**。

上图中序遍历结果是：**DBEAFC**。

核心代码：

Java:

```
private void traverse(TreeNode root, ArrayList<Integer> result) {
    if (root == null) {
        return;
    }
    traverse(root.left, result);
    result.add(root.val); // 注意访问根节点放到了遍历左子树的后面
    traverse(root.right, result);
}
```

Python:

```
def traverse(root, result):
    if not root:
        return
    traverse(root.left, result)
    result.append(root.val) # 注意访问根节点放到了遍历左子树的后面
    traverse(root.right, result)
```

相关练习及完整答案。

后序遍历（又叫后根遍历）

首先遍历左子树，然后遍历右子树，最后访问根结点。遍历左、右子树时，仍按后序遍历。若二叉树为空则返回。简记为左右根。

上图后序遍历结果是：**DEBFCA**。

核心代码：

Java:

```
private void traverse(TreeNode root, ArrayList<Integer> result) {
    if (root == null) {
        return;
    }
    traverse(root.left, result);
    traverse(root.right, result);
    result.add(root.val); // 注意访问根节点放到了最后
}
```

Python:

```
def traverse(root, result):
    if not root:
        return
    traverse(root.left, result)
    traverse(root.right, result)
    result.append(root.val) # 注意访问根节点放到了最后
```

相关练习及完整答案。

一些有趣的题目：

<http://www.lintcode.com/problem/construct-binary-tree-from-inorder-and-postorder-traversal/>

<http://www.lintcode.com/problem/construct-binary-tree-from-preorder-and-inorder-traversal/>

二叉树上的分治法

定义

分治法 (Divide & Conquer Algorithm) 是说将一个大问题，拆分为2个或者多个小问题，当小问题得到结果之后，合并他们的结果来得到大问题的结果。

举一个例子，比如中国要进行人口统计。那么如果使用遍历 (Traversal) 的办法，做法如下：

人口普查员小张自己一个人带着一个本子，跑遍全中国挨家挨户的敲门查户口

而如果使用分治法，做法如下：

1. 国家统计局的老板小李想要知道全国人口的总数，于是他找来全国各个省的统计局领导，下派人口普查任务给他们，让他们各自去统计自己省的人口总数。在小李这儿，他只需要最后将各个省汇报的人口总数结果累加起来，就得到了全国人口的数目。
2. 然后每个省的领导，又找来省里各个市的领导，让各个市去做人口统计。
3. 市找县，县找镇，镇找乡。最后乡里的干部小王挨家挨户敲门去查户口。

在这里，把全国的任务拆分为省级的任务的过程，就是分治法中 **分** 的这个步骤。把各个小任务派发给别人去完成的过程，就是分治法中 **治** 的这个步骤。但是事实上我们还有第三个步骤，就是将小任务的结果合并到一起的过程，**合** 这个步骤。因此如果我来取名字的话，我会叫这个算法：**分治合算法**。

为什么二叉树的问题适合使用分治法？

在一棵二叉树 (Binary Tree) 中，如果将整棵二叉树看做一个大问题的话，那么根节点 (Root) 的左子树 (Left subtree) 就是一个小问题，右子树 (Right subtree) 是另外一个小问题。这是一个天然就帮你完成了“分”这个步骤的数据结构。

这一小节中，我们通过如下的一些较为简单的练习题，来学习和对比遍历法和分治法：

- 二叉树最大深度
- 判断平衡二叉树
- 判断排序二叉树

递归，分治法，遍历法的联系与区别

联系

分治法（Divide & Conquer）与遍历法（Traverse）是两种常见的递归（Recursion）方法。

分治法解决问题的思路

先让左右子树去解决同样的问题，然后得到结果之后，再整合为整棵树的结果。

遍历法解决问题的思路

通过前序/中序/后序的某种遍历，游走整棵树，通过一个全局变量或者传递的参数来记录这个过程中所遇到的点和需要计算的结果。

两种方法的区别

从程序实现角度分治法的递归函数，通常有一个 **返回值**，遍历法通常没有。

递归、回溯和搜索

什么是递归（Recursion）？

很多书上会把递归（Recursion）当作一种算法。事实上，递归是包含两个层面的意思：

1. 一种由大化小，由小化无的解决问题的算法。类似的算法还有动态规划（Dynamic Programming）。
2. 一种程序的实现方式。这种方式就是一个函数（Function / Method / Procedure）自己调用自己。

与之对应的，有非递归（Non-Recursion）和迭代法（Iteration），你可以认为这两个概念是一样的概念（番茄和西红柿的区别）。不需要做区分。

什么是搜索（Search）？

搜索分为深度优先搜索（Depth First Search）和宽度优先搜索（Breadth First Search），通常分别简写为 DFS 和 BFS。搜索是一种类似于枚举（Enumerate）的算法。比如我们需要找到一个数组里的最大值，我们可以采用枚举法，因为我们知道数组的范围和大小，比如经典的打擂台算法：Java:

```
int max = nums[0];
for (int i = 1; i < nums.length; i++) {
    max = Math.max(max, nums[i]);
}
```

Python:

```
max_num = nums[0]
for i in range(1, len(nums)):
    max_num = max(max_num, nums[i])
```

枚举法通常是你知道循环的范围，然后可以用几重循环就搞定的算法。比如我需要找到所有 $x^2 + y^2 = K$ 的整数组合，可以用两重循环的枚举法：

Java:

```
// 不要在意这个算法的时间复杂度
for (int x = 1; x <= k; x++) {
    for (int y = 1; y <= k; y++) {
        if (x * x + y * y == k) {
            // print x and y
        }
    }
}
```

Python:

```
for x in range(1, k+1):
    for y in range(1, k+1):
        if x*x + y*y == k:
            # print x and y
```

而有的问题，比如求 N 个数的全排列，你可能需要用 N 重循环才能解决。这个时候，我们就倾向于采用递归的方式去实现这个变化的 N 重循环。这个时候，我们把这个算法称之为 [搜索](#)。因为你已经不能明确的写出一个不依赖于输入数据的多重循环了。

通常来说 DFS 我们会采用递归的方式实现（当然你强行写一个非递归的版本也是可以的），而 BFS 则无需递归（使用队列 Queue + 哈希表 HashMap 就可以）。[所以我们在面试中，如果一个问题既可以使用 DFS，又可以使用 BFS 的情况下，一定要优先使用 BFS。](#) 因为他是非递归的，而且更容易实现。

什么是回溯(Backtracking)？

有的时候，深度优先搜索算法（DFS），又被称为回溯法，所以你可以完全认为回溯法，就是深度优先搜索算法。在我的理解中，回溯实际上是深度优先搜索过程中的一个步骤。比如我们在进行全子集问题的搜索时，假如当前的集合是 {1,2} 代表我正在寻找以 {1,2} 开头的所有集合。那么他的下一步，会去寻找 {1,2,3} 开头的所有集合，然后当我们找完所有以 {1,2,3} 开头的集合时，我们需要把 3 从集合中删掉，回到 {1,2}。然后再把 4 放进去，寻找以 {1,2,4} 开头的所有集合。这个把 3 删掉回到 {1,2} 的过程，就是回溯。

Java:

```
subset.add(nums[i]);
subsetsHelper(result, subset, nums, i + 1);
subset.remove(list.size() - 1) // 这一步就是回溯
```

Python:

```
subset.add(nums[i])
subsetsHelper(result, subset, nums, i + 1)
subset.remove(len(list) - 1)
```

递归三要素

我们以《二叉树的最大深度》和《二叉树的前序遍历》两个题目为例子，来分析一下递归的三要素。

相关题目链接：

<http://www.lintcode.com/problem/maximum-depth-of-binary-tree/>

<http://www.lintcode.com/problem/binary-tree-preorder-traversal/>

1. 递归的定义

每一个递归函数，都需要有明确的定义，有了正确的定义以后，才能够对递归进行拆解。

例子：

Java:

```
int maxDepth(TreeNode root)
```

Python:

```
def maxDepth(root):
```

代表 以 `root` 开头的子树的最大深度是多少。

Java:

```
void preorder(TreeNode root, List<TreeNode> result)
```

Python:

```
def preorder(root, result):
```

代表 将 `root` 开头的子树的前序遍历放到 `result` 里面

Java:

```
void preorder(TreeNode root, List<TreeNode> result)
```

Python:

```
def preorder(root, result):
```

代表 将 `root` 开头的子树的前序遍历放到 `result` 里面

2. 递归的拆解

一个 大问题 如何拆解为若干个 小问题 去解决。

例子：

Java:

```
int leftDepth = maxDepth(root.left);
int rightDepth = maxDepth(root.right);
return Math.max(leftDepth, rightDepth) + 1;
```

Python:

```
leftDepth = maxDepth(root.left)
rightDepth = maxDepth(root.right)
return max(leftDepth, rightDepth) + 1
```

整棵树的最大深度，可以拆解为先计算左右子树深度，然后在左右子树深度中找到最大值+1来解决。

Java:

```
result.add(root);
preorder(root.left, result);
preorder(root.right, result);
```

Python:

```
result.append(root)
preorder(root.left, result)
preorder(root.right, result)
```

一棵树的前序遍历可以拆解为3个部分：

1. 根节点自己 (root)
2. 左子树的前序遍历
3. 右子树的前序遍历

所以对应的，我们把这个递归问题也拆分为三个部分来解决：

1. 先把 root 放到 result 里 --> result.add(root);
2. 再把左子树的前序遍历放到 result 里 --> preorder(root.left, result)。回想一下递归的定义，是不是正是如此？
3. 再把右子树的前序遍历放到 result 里 --> preorder(root.right, result)。

3. 递归的出口

什么时候可以直接知道答案，不用再拆解，直接 return

例子：

Java:

```
// 二叉树的最大深度
if (root == null) {
    return 0;
}
```

Python:

```
# 二叉树的最大深度
if not root:
    return 0
```

一棵空的二叉树，可以认为是一个高度为 0 的二叉树。

Java:

```
// 二叉树的前序遍历
if (root == null) {
    return;
}
```

Python:

```
if not root:
    return
```

一棵空的二叉树，自然不用往 result 里放任何的东西。

每一个递归函数，都需要有明确的定义，有了正确的定义以后，才能够对递归进行拆解。

例子：

Java:

```
int maxDepth(TreeNode root)
```

Python:

```
def maxDepth(root):
```

代表 以 `root` 开头的子树的最大深度是多少。

Java:

```
void preorder(TreeNode root, List<TreeNode> result)
```

Python:

```
def preorder(root, result):
```

代表 将 `root` 开头的子树的前序遍历放到 `result` 里面

一个 **大问题** 如何拆解为若干个 **小问题** 去解决。

例子：

Java:

```
int leftDepth = maxDepth(root.left);
int rightDepth = maxDepth(root.right);
return Math.max(leftDepth, rightDepth) + 1;
```

Python:

```
leftDepth = maxDepth(root.left)
rightDepth = maxDepth(root.right)
return max(leftDepth, rightDepth) + 1
```

整棵树的最大深度，可以拆解为先计算左右子树深度，然后在左右子树深度中找到最大值+1来解决。

Java:

```
result.add(root);
preorder(root.left, result);
preorder(root.right, result);
```

Python:

```
result.append(root)
preorder(root.left, result)
preorder(root.right, result)
```

一棵树的前序遍历可以拆解为3个部分：

1. 根节点自己 (root)
2. 左子树的前序遍历
3. 右子树的前序遍历

所以对应的，我们把这个递归问题也拆分为三个部分来解决：

1. 先把 root 放到 result 里 --> result.add(root);
2. 再把左子树的前序遍历放到 result 里 --> preorder(root.left, result)。回想一下递归的定义，是不是正是如此？
3. 再把右子树的前序遍历放到 result 里 --> preorder(root.right, result)。

什么时候可以直接知道答案，不用再拆解，直接 return

例子：

Java:

```
// 二叉树的最大深度
if (root == null) {
    return 0;
}
```

Python:

```
# 二叉树的最大深度
if not root:
    return 0
```

一棵空的二叉树，可以认为是一个高度为 0 的二叉树。

Java:

```
// 二叉树的前序遍历
if (root == null) {
    return;
}
```

Python:

```
if not root:
    return
```

一棵空的二叉树，自然不用往 result 里放任何的东西。

使用 ResultType 返回多个值

什么是 ResultType

通常是我们定义在某个文件内部使用的一个类。比如：

Java:

```
class ResultType {
    int maxValue, minValue;
    public ResultType(int maxValue, int minValue) {
        this.maxValue = maxValue;
        this.minValue = minValue;
    }
}
```

什么时候需要 ResultType

当我们定义的函数需要返回多个值供调用者计算时，就需要使用 ResultType 了。

所以如果你只是返回一个值就够用的话，就不需要。

其他语言需要 ResultType 么？

不是所有的语言都需要自定义 ResultType。

像 Python 这样的语言，天生支持你返回多个值作为函数的 return value，所以是不需要的。

什么是二叉搜索树

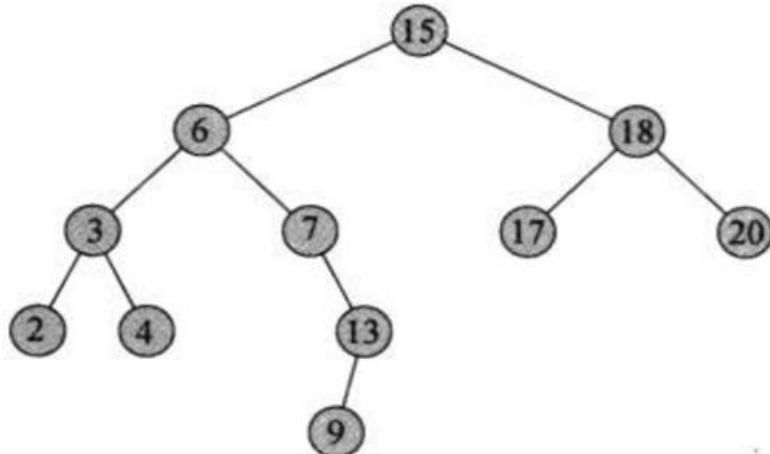
定义

二叉搜索树 (Binary Search Tree, 又名排序二叉树, 二叉查找树, 通常简写为BST) 定义如下：

空树或是具有下列性质的二叉树：

- (1) 若左子树不空，则左子树上所有节点值均小于或等于它的根节点值；
- (2) 若右子树不空，则右子树上所有节点值均大于或等于它的根节点值；
- (3) 左、右子树也为二叉搜索树；

如图即为BST：



BST 的特性

- 按照[中序遍历](#) (inorder traversal) 打印各节点，会得到由小到大的顺序。
- 在BST中搜索某值的平均情况下复杂度为 $O(\log N)$ ，最坏情况下复杂度为 $O(N)$ ，其中N为节点个数。将待寻值与节点值比较，若不相等，则通过是小于还是大于，可断定该值只可能在左子树还是右子树，继续向该子树搜索。
- 在balanced BST中查找某值的时间复杂度为 $O(\log N)$ 。

BST 的作用

- 通过中序遍历，可快速得到升序节点列表。
- 在BST中查找元素，平均情况下时间复杂度是 $O(\log N)$ ；插入新节点，保持BST特性平均情况下要耗时 $O(\log N)$ 。（[参考链接](#)）。
- 和有序数组的对比：有序数组查找某元素可以用二分法，时间复杂度是 $O(\log N)$ ；但是插入新元素，维护数组有序性要耗时 $O(N)$ 。

常见的BST面试题

<http://www.lintcode.com/en/tag/binary-search-tree/>

BST是一种重要且基本的结构，其相关题目也十分经典，并延伸出很多算法。

在BST之上，有许多高级且有趣的变种，以解决各式各样的问题，例如：

- 用于数据库或各语言标准库中索引的[红黑树](#)
- 提升二叉树性能底线的[伸展树](#)
- 优化红黑树的[AA树](#)
- 随机插入的[树堆](#)
- 机器学习kNN算法的高维快速搜索[k-d树](#)

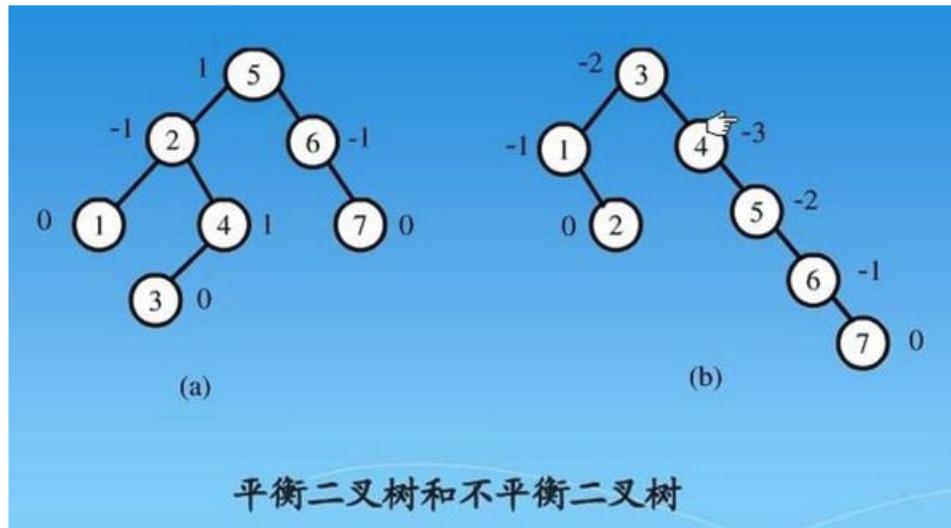
.....

什么是平衡二叉搜索树

定义

平衡二叉搜索树 (Balanced Binary Search Tree, 又称为AVL树, 有别于AVL算法) 是二叉树中的一种特殊的形态。二叉树当且仅当满足如下两个条件之一, 是平衡二叉树:

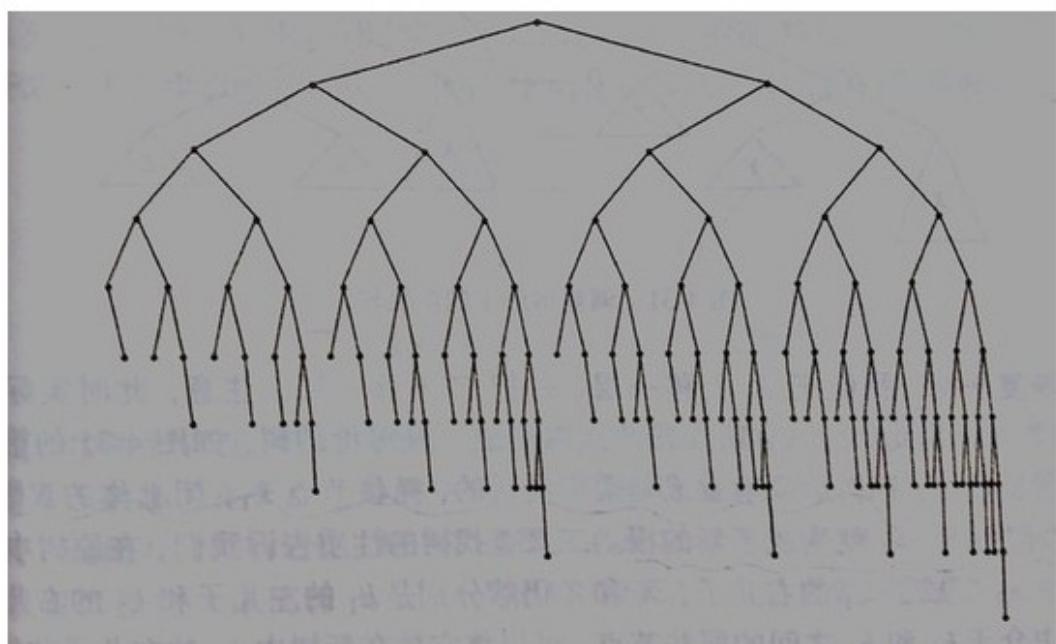
- 空树。
- 左右子树高度差绝对值不超过1且左右子树都是平衡二叉树。



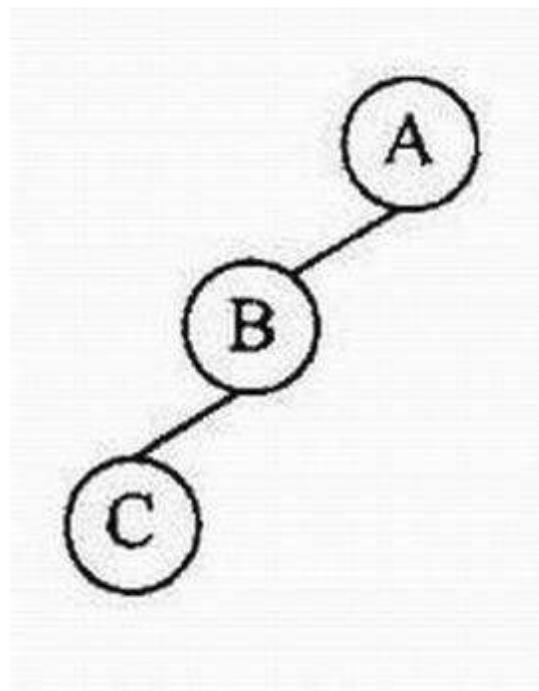
AVL树的高度为 $O(\log N)$

当AVL树有N个节点时, 高度为 $O(\log N)$ 。为何?

试想一棵满二叉树, 每个节点左右子树高度相同, 随着树高的增加, 叶子容量指数暴增, 故树高一定是 $O(\log N)$ 。而相比于满二叉树, AVL树仅放宽一个条件, 允许左右两子树高度差1, 当树高足够大时, 可以把1忽略。如图是高度为9的最小AVL树, 若节点更少, 树高绝不会超过8, 也即为何AVL树高会被限制到 $O(\log N)$, 因为树不可能太稀疏。严格的数学证明复杂, 略去。



为何普通二叉树不是 $O(\log N)$? 这里给出最坏的单枝树, 若单枝扩展, 则树高为 $O(N)$:



AVL树有什么用?

最大作用是保证查找的最坏时间复杂度为 $O(\log N)$ 。而且较浅的树对插入和删除等操作也更快。

AVL树的相关练习题

判断一棵树是否为平衡树

<http://www.lintcode.com/problem/balanced-binary-tree/>

提示：可以自下而上递归判断每个节点是否平衡。若平衡将当前节点高度返回，供父节点判断;否则该树一定不平衡。

二叉树相关有一些内容，如果学有余力可以掌握一下，可以提升自信心（因为知道了别人不知道的东西），更有底气去面试（虽然考到的概率很低）：

1. 用 Morris 算法实现 $O(1)$ 额外空间对二叉树进行先序遍历
2. 用非递归（Non-recursion / Iteration）的方式实现二叉树的前序遍历，中序遍历和后序遍历
3. BST 的增删查改
4. 平衡排序二叉树（Balanced Binary Search Tree）及 TreeSet / TreeMap 的使用

用 Morris 算法实现 O(1) 额外空间遍历二叉树

什么是 Morris 算法

与递归和使用栈空间遍历的思想不同，Morris 算法使用二叉树中的叶节点的right指针来保存后面将要访问的节点的信息，当这个right指针使用完成之后，再将它置为null，但是在访问过程中有些节点会访问两次，所以与递归的空间换时间的思路不同，Morris则是使用时间换空间的思想。

节点定义

Java:

```
class TreeNode{  
    int val;  
    TreeNode left;  
    TreeNode right;  
    public TreeNode(int val) {  
        this.val = val;  
        this.left = this.right = null;  
    }  
}
```

Python:

```
class TreeNode:  
    def __init__(self, val):  
        self.val = val  
        self.left, self.right = None, None
```

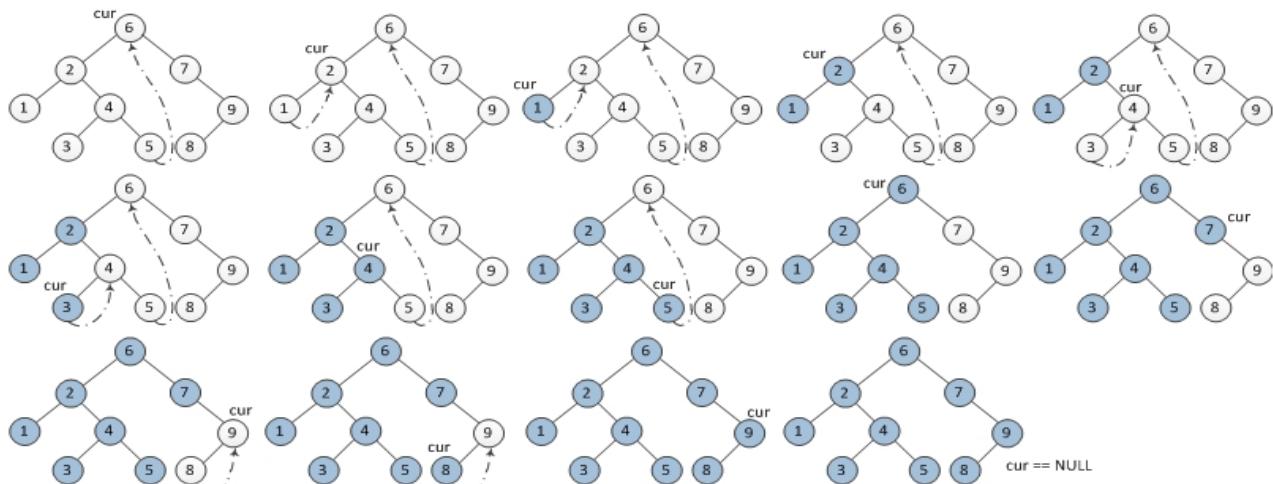
用 Morris 算法进行中序遍历(Inorder Traversal)

思路

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。
 1. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。
 2. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状）。输出当前节点。当前节点更新为当前节点的右孩子。
3. 重复1、2两步直到当前节点为空。

图示

下图为每一步迭代的结果（从左至右，从上到下），cur代表当前节点，深色节点表示该节点已输出。



示例代码

Java:

```
public class Solution {
    /**
     * @param root: A Tree
     * @return: Inorder in ArrayList which contains node values.
     */
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> nums = new ArrayList<>();
        TreeNode cur = null;

        while (root != null) {
            if (root.left != null) {
                cur = root.left;
                while (cur.right != null && cur.right != root) {
                    cur = cur.right;
                }
            }

            if (cur.right == root) {
                nums.add(root.val);
                cur.right = null;
                root = root.right;
            } else {
                cur.right = root;
                root = root.left;
            }
        } else {
            nums.add(root.val);
            root = root.right;
        }
    }
}
```

Python:

```
class Solution:
    """
    @param root: A Tree
    @return: Inorder in ArrayList which contains node values.
    """
    def inorderTraversal(self, root):
        nums = []
        cur = None

        while root:
            if root.left:
                cur = root.left
                while cur.right and cur.right != root:
                    cur = cur.right

                if cur.right == root:
                    nums.append(root.val)
                    cur.right = None
                    root = root.right
                else:
                    cur.right = root
                    root = root.left
            else:
                nums.append(root.val)
                root = root.right

        return nums
```

LintCode 练习

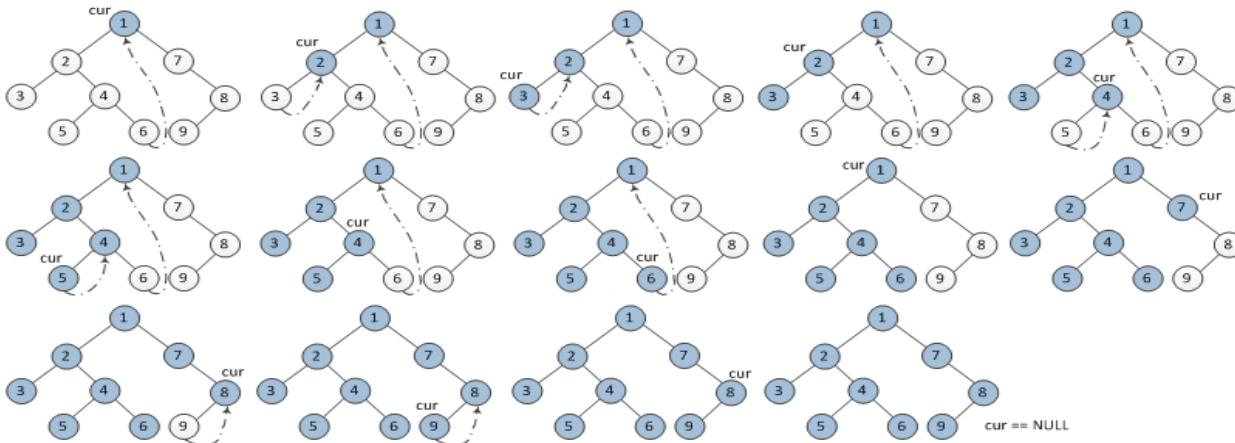
<http://www.lintcode.com/problem/binary-tree-inorder-traversal/>

用 Morris 算法实现先序遍历(Preorder Traversal)

思路

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。
 1. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。**输出当前节点**（与中序遍历唯一一点不同）。当前节点更新为当前节点的左
 2. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。当前节点更新为当前节点的右孩子。
3. 重复1、2两步直到当前节点为空。

图示



示例代码

Java:

```
public class Solution {  
    /**  
     * @param root: A Tree  
     * @return: Preorder in ArrayList which contains node values.  
     */  
    public List<Integer> preorderTraversal(TreeNode root) {  
        // morris traversal  
        List<Integer> nums = new ArrayList<>();  
        TreeNode cur = null;  
        while (root != null) {  
            if (root.left != null) {  
                cur = root.left;  
                // find the predecessor of root node  
                while (cur.right != null && cur.right != root) {  
                    cur = cur.right;  
                }  
                if (cur.right == root) {  
                    cur.right = null;  
                    root = root.right;  
                } else {  
                    nums.add(root.val);  
                    cur.right = root;  
                    root = root.left;  
                }  
            } else {  
                nums.add(root.val);  
                root = root.right;  
            }  
        }  
        return nums;  
    }  
}
```

Python:

```
class Solution:
    """
    @param root: A Tree
    @return: Preorder in ArrayList which contains node values.
    """
    def preorderTraversal(self, root):
        nums = []
        cur = None

        while root:
            if root.left:
                cur = root.left
                while cur.right and cur.right != root:
                    cur = cur.right
                if cur.right == root:
                    cur.right = None
                    root = root.right
                else:
                    nums.append(root.val)
                    cur.right = root
                    root = root.left
            else:
                nums.append(root.val)
                root = root.right

        return nums
```

LintCode 练习

<http://www.lintcode.com/problem/binary-tree-preorder-traversal/>

用 Morris 算法实现后序遍历(Postorder Traversal)

思路

* 后序遍历其实可以看作是和前序遍历左右对称的，此处，我们同样可以利用这个性质，基于前序遍历的算法，可以很快得到后序遍历的结果。我们只：

示例代码

Java:

```
public class Solution {  
    /**  
     * @param root: A Tree  
     * @return: Postorder in ArrayList which contains node values.  
     */  
    public List<Integer> postorderTraversal(TreeNode root) {  
        List<Integer> nums = new ArrayList<>();  
        TreeNode cur = null;  
        while (root != null) {  
            if (root.right != null) {  
                cur = root.right;  
                while (cur.left != null && cur.left != root) {  
                    cur = cur.left;  
                }  
                if (cur.left == root) {  
                    cur.left = null;  
                    root = root.left;  
                } else {  
                    nums.add(root.val);  
                    cur.left = root;  
                    root = root.right;  
                }  
            } else {  
                nums.add(root.val);  
                root = root.left;  
            }  
        }  
        Collections.reverse(nums);  
        return nums;  
    }  
}
```

Python:

```
class Solution:
    """
    @param root: A Tree
    @return: Postorder in ArrayList which contains node values.
    """
    def postorderTraversal(self, root):
        nums = []
        cur = None

        while root:
            if root.right != None:
                cur = root.right
                while cur.left and cur.left != root:
                    cur = cur.left
                if cur.left == root:
                    cur.left = None
                    root = root.left
                else:
                    nums.append(root.val)
                    cur.left = root
                    root = root.right
            else:
                nums.append(root.val)
                root = root.left

        nums.reverse()
        return nums
```

LintCode 练习

<http://www.lintcode.com/problem/binary-tree-postorder-traversal/>

非递归的方式实现二叉树遍历

先序遍历

思路

遍历顺序为根、左、右

1. 如果根节点非空，将根节点加入到栈中。
2. 如果栈不空，弹出出栈顶节点，将其值加加入到数组中。
 - i. 如果该节点的右子树不为空，将右子节点加入栈中。
 - ii. 如果左子节点不为空，将左子节点加入栈中。
3. 重复第二步，直到栈空。

代码实现

Java:

```
public class Solution {  
    public List<Integer> preorderTraversal(TreeNode root) {  
        Stack<TreeNode> stack = new Stack<TreeNode>();  
        List<Integer> preorder = new ArrayList<Integer>();  
  
        if (root == null) {  
            return preorder;  
        }  
  
        stack.push(root);  
        while (!stack.empty()) {  
            TreeNode node = stack.pop();  
            preorder.add(node.val);  
            if (node.right != null) {  
                stack.push(node.right);  
            }  
            if (node.left != null) {  
                stack.push(node.left);  
            }  
        }  
  
        return preorder;  
    }  
}
```

Python:

```
class Solution:  
    """  
    @param root: A Tree  
    @return: Preorder in ArrayList which contains node values.  
    """  
    def preorderTraversal(self, root):  
        stack = []  
        preorder = []  
  
        if not root:  
            return preorder  
  
        stack.append(root)  
        while len(stack) > 0:  
            node = stack.pop()  
            preorder.append(node.val)  
            if node.right:  
                stack.append(node.right)  
            if node.left:  
                stack.append(node.left)  
  
        return preorder
```

练习

<http://www.lintcode.com/problem/binary-tree-preorder-traversal/>

中序遍历

思路

遍历顺序为左、根、右

1. 如果根节点非空，将根节点加入到栈中。
2. 如果栈不空，取栈顶元素（暂时不弹出），
 - i. 如果左子树已访问过，或者左子树为空，则弹出栈顶节点，将其值加入数组，如有右子树，将右子节点加入栈中。
 - ii. 如果左子树不为空，则将左子节点加入栈中。
3. 重复第二步，直到栈空。

代码实现

Java:

```
public class Solution {  
    /**  
     * @param root: The root of binary tree.  
     * @return: Inorder in ArrayList which contains node values.  
     */  
    public ArrayList<Integer> inorderTraversal(TreeNode root) {  
        Stack<TreeNode> stack = new Stack<>();  
        ArrayList<Integer> result = new ArrayList<>();  
  
        while (root != null) {  
            stack.push(root);  
            root = root.left;  
        }  
  
        while (!stack.isEmpty()) {  
            TreeNode node = stack.peek();  
            result.add(node.val);  
  
            if (node.right == null) {  
                node = stack.pop();  
                while (!stack.isEmpty() && stack.peek().right == node) {  
                    node = stack.pop();  
                }  
            } else {  
                node = node.right;  
                while (node != null) {  
                    stack.push(node);  
                    node = node.left;  
                }  
            }  
        }  
        return result;  
    }  
}
```

Python:

```
class Solution:  
    """  
    @param root: A Tree  
    @return: Inorder in ArrayList which contains node values.  
    """  
    def inorderTraversal(self, root):  
        stack = []  
        result = []  
  
        while root:  
            stack.append(root)  
            root = root.left  
  
        while len(stack) > 0:  
            node = stack[-1]  
            result.append(node.val)  
  
            if not node.right:  
                node = stack.pop()  
                while len(stack) > 0 and stack[-1].right == node:  
                    node = stack.pop()  
            else:  
                node = node.right  
                while node:  
                    stack.append(node)  
                    node = node.left  
  
        return result
```

后序遍历

思路

遍历顺序为左、右、根

1. 如果根节点非空，将根节点加入到栈中。
2. 如果栈不空，取栈顶元素（暂时不弹出），
 - i. 如果（左子树已访问过或者左子树为空），且（右子树已访问过或右子树为空），则弹出栈顶节点，将其值加入数组，
 - ii. 如果左子树不为空，且未访问过，则将左子节点加入栈中，并标左子树已访问过。
 - iii. 如果右子树不为空，且未访问过，则将右子节点加入栈中，并标右子树已访问过。
3. 重复第二步，直到栈空。

代码实现

Java:

```
public ArrayList<Integer> postorderTraversal(TreeNode root) {  
    ArrayList<Integer> result = new ArrayList<Integer>();  
    Stack<TreeNode> stack = new Stack<TreeNode>();  
    TreeNode prev = null; // previously traversed node  
    TreeNode curr = root;  
  
    if (root == null) {  
        return result;  
    }  
  
    stack.push(root);  
    while (!stack.empty()) {  
        curr = stack.peek();  
        if (prev == null || prev.left == curr || prev.right == curr) { // traverse down the tree  
            if (curr.left != null) {  
                stack.push(curr.left);  
            } else if (curr.right != null) {  
                stack.push(curr.right);  
            }  
        } else if (curr.left == prev) { // traverse up the tree from the left  
            if (curr.right != null) {  
                stack.push(curr.right);  
            }  
        } else { // traverse up the tree from the right  
            result.add(curr.val);  
            stack.pop();  
        }  
        prev = curr;  
    }  
  
    return result;  
}
```

Python

```
class Solution:
    """
    @param root: A Tree
    @return: Postorder in ArrayList which contains node values.
    """
    def postorderTraversal(self, root):
        result = []
        stack = []
        prev, curr = None, root

        if not root:
            return result

        stack.append(root)
        while len(stack) > 0:
            curr = stack[-1]
            if not prev or prev.left == curr or prev.right == curr: # traverse down the tree
                if curr.left:
                    stack.append(curr.left)
                elif curr.right:
                    stack.append(curr.right)
                elif curr.left == prev: # traverse up the tree from the left
                    if curr.right:
                        stack.append(curr.right)
                else: # traverse up the tree from the right
                    result.append(curr.val)
                    stack.pop()
            prev = curr

        return result
```

练习

<http://www.lintcode.com/problem/binary-tree-postorder-traversal/>

BST 的增删查改

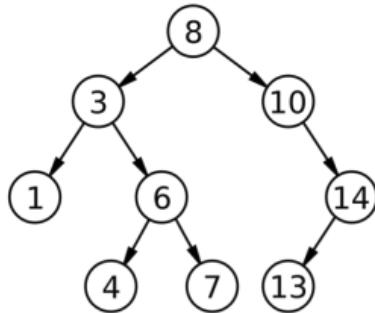
什么是二叉搜索树(Binary Search Tree)

二叉搜索树可以是一棵空树或者是一棵满足下列条件的二叉树：

- 如果它的左子树不空，则左子树上所有节点值 **均小于** 它的根节点值。
- 如果它的右子树不空，则右子树上所有节点值 **均大于** 它的根节点值。
- 它的左右子树均为二叉搜索树(BST)。
- 严格定义下BST中是没有值相等的节点的(No duplicate nodes)。

根据上述特性，我们可以得到一个结论：BST中序遍历得到的序列是升序的。如下述BST的中序序列为：

[1,3,4,6,7,8,10,13,14]



BST基本操作——增删改查(CRUD)

对于树节点的定义如下：

Java:

```
class TreeNode{  
    int val;  
    TreeNode left;  
    TreeNode right;  
    public TreeNode(int val) {  
        this.val = val;  
        this.left = this.right = null;  
    }  
}
```

Python:

```
class TreeNode:  
    def __init__(self, val):  
        self.val = val  
        self.left, self.right = None, None
```

基本操作之查找(Retrieve)

- 思路
 - 查找值为**val**的节点，如果**val**小于根节点则在左子树中查找，反之在右子树中查找
- 代码实现

Java:

```
public TreeNode searchBST(TreeNode root, int val) {  
    if (root == null) {  
        return null;  
    } // 未找到值为val的节点  
    if (val < root.val) {  
        return searchBST(root.left, val); // val小于根节点值，在左子树中查找  
    } else if (val > root.val) {  
        return searchBST(root.right, val); // val大于根节点值，在右子树中查找  
    } else {  
        return root; // 找到了  
    }  
}
```

Python:

```
def searchBST(root, val):  
    if not root:  
        return None # 未找到值为val的节点  
    if val < root.val:  
        return searchBST(root.left, val) # val小于根节点值，在左子树中查找哦  
    elif val > root.val:  
        return searchBST(root.right, val) # val大于根节点值，在右子树中查找  
    else:  
        return root
```

- 实战

- <http://www.lintcode.com/en/problem/search-range-in-binary-search-tree/>
- <http://www.lintcode.com/en/problem/two-sum-bst-edition/>
- <http://www.lintcode.com/en/problem/closest-binary-search-tree-value/>
- <http://www.lintcode.com/en/problem/closest-binary-search-tree-value-ii/>
- <http://www.lintcode.com/en/problem/trim-binary-search-tree/>
- <http://www.lintcode.com/en/problem/bst-swapped-nodes/>

基本操作之修改(Update)

- 思路

- 修改仅仅需要在查找到需要修改的节点之后，更新这个节点的值就可以了

- 代码实现

Java:

```
public void updateBST(TreeNode root, int target, int val) {  
    if (root == null) {  
        return;  
    } // 未找到target节点  
    if (target < root.val) {  
        updateBST(root.left, target, val); // target小于根节点值，在左子树中查找  
    } else if (target > root.val) {  
        updateBST(root.right, target, val); // target大于根节点值，在右子树中查找  
    } else { // 找到了  
        root.val = val;  
    }  
}
```

Python:

```
def updateBSTBST(root, target, val):
    if not root:
        return # 未找到target节点
    if target < root.val:
        updateBST(root.left, target, val) # target小于根节点值，在左子树中查找哦
    elif target > root.val:
        updateBST(root.right, target, val) # target大于根节点值，在右子树中查找
    else: # 找到了
        root.val = val
```

- 实战

- <http://www.lintcode.com/en/problem/bst-swapped-nodes/>

基本操作之增加(Create)

- 思路

- 根节点为空，则待添加的节点为根节点
- 如果待添加的节点值小于根节点，则在左子树中添加
- 如果待添加的节点值大于根节点，则在右子树中添加
- 我们统一在树的叶子节点(Leaf Node)后添加

- 代码实现

Java:

```
public TreeNode insertNode(TreeNode root, TreeNode node) {
    if (root == null) {
        return node;
    }
    if (root.val > node.val) {
        root.left = insertNode(root.left, node);
    } else {
        root.right = insertNode(root.right, node);
    }
    return root;
}
```

Python:

```
def insertNode(root, node):
    if not root:
        return node
    if root.val > node.val:
        root.left = insertNode(root.left, node)
    else:
        root.right = insertNode(root.right, node)
    return root
```

- 实战
 - <http://www.lintcode.com/en/problem/insert-node-in-a-binary-search-tree/>

基本操作之删除(Delete)

- 思路(最为复杂)
 - 考虑待删除的节点为叶子节点，可以直接删除并修改父亲节点(Parent Node)的指针，需要区分待删节点是否为根节点
 - 考虑待删除的节点为单支节点(只有一棵子树——左子树 or 右子树)，与删除链表节点操作类似，同样的需要区分待删节点是否为根节点
 - 考虑待删节点有两棵子树，可以将待删节点与左子树中的最大节点进行交换，由于左子树中的最大节点一定为叶子节点，所以这时再删除待删的节点可以参考第一条
 - 详细的解释可以看 http://www.algolist.net/Data_structures/Binary_search_tree/Removal
- 代码实现

Java:

```
public TreeNode removeNode(TreeNode root, int value) {  
    TreeNode dummy = new TreeNode(0);  
    dummy.left = root;  
    TreeNode parent = findNode(dummy, root, value);  
    TreeNode node;  
    if (parent.left != null && parent.left.val == value) {  
        node = parent.left;  
    } else if (parent.right != null && parent.right.val == value) {  
        node = parent.right;  
    } else {  
        return dummy.left;  
    }  
    deleteNode(parent, node);  
    return dummy.left;  
}  
  
private TreeNode findNode(TreeNode parent, TreeNode node, int value) {  
    if (node == null) {  
        return parent;  
    }  
    if (node.val == value) {  
        return parent;  
    }  
    if (value < node.val) {  
        return findNode(node, node.left, value);  
    } else {  
        return findNode(node, node.right, value);  
    }  
}  
  
private void deleteNode(TreeNode parent, TreeNode node) {  
    if (node.right == null) {  
        if (parent.left == node) {  
            parent.left = node.left;  
        } else {  
            parent.right = node.left;  
        }  
    } else {  
        TreeNode temp = node.right;  
        TreeNode father = node;  
        while (temp.left != null) {  
            father = temp;  
            temp = temp.left;  
        }  
        if (father.left == temp) {  
            father.left = temp.right;  
        } else {  
            father.right = temp.right;  
        }  
        if (parent.left == node) {  
            parent.left = temp;  
        } else {  
            parent.right = temp;  
        }  
        temp.left = node.left;  
        temp.right = node.right;  
    }  
}
```

Python:

```
def removeNode(root, value):
    dummy = TreeNode(0)
    dummy.left = root
    parent = findNode(dummy, root, value)
    node = None
    if parent.left and parent.left.val == value:
        node = parent.left
    elif parent.right and parent.right.val == value:
        node = parent.right
    else:
        return dummy.left
    deleteNode(parent, node)
    return dummy.left

def findNode(parent, node, value):
    if not node:
        return parent
    if node.val == value:
        return parent
    if value < node.val:
        return findNode(node, node.left, value)
    else:
        return findNode(node, node.right, value)

def deleteNode(parent, node):
    if not node.right:
        if parent.left == node:
            parent.left = node.left
        else:
            parent.right = node.left
    else:
        temp = node.right
        father = node
        while temp.left:
            father = temp
            temp = temp.left
        if father.left == temp:
            father.left = temp.right
        else:
            father.right = temp.right
        if parent.left == node:
            parent.left = temp
        else:
            parent.right = temp
        temp.left = node.left
        temp.right = node.right
```

- 实战

- <http://www.lintcode.com/en/problem/remove-node-in-binary-search-tree/>
- <http://www.lintcode.com/en/problem/trim-binary-search-tree/>

平衡排序二叉树

平衡排序二叉树(Self-balancing Binary Search Tree)

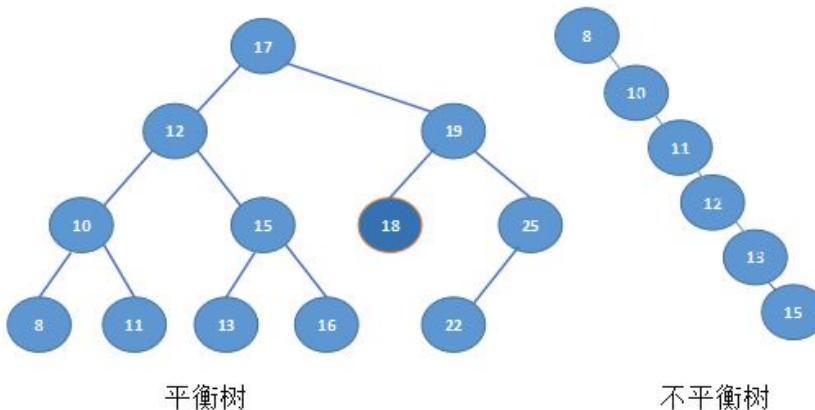
定义

平衡二叉搜索树又被称为AVL树（有别于AVL算法），且具有以下性质：

- 它是一棵空树或它的左右两个子树的高度差的绝对值不超过1
- 左右两棵子树都是一棵平衡二叉搜索树
- 平衡二叉搜索树必定是二叉搜索树，反之则不一定。

平衡排序二叉树 与 二叉搜索树 对比

也许因为输入值不够随机，也许因为输入顺序的原因，还或许一些插入、删除操作，会使得二叉搜索树失去平衡，造成搜索效率低落的情况。



比如上面两个树，在平衡树上寻找15就只要2次查找，在非平衡树上却要5次查找方能找到，效率明显下降。

平衡排序二叉树节点定义

Java:

```
class TreeNode{  
    int val;  
    TreeNode left;  
    TreeNode right;  
    public TreeNode(int val) {  
        this.val = val;  
        this.left = this.right = null;  
    }  
}
```

Python:

```
class TreeNode:  
    def __init__(self, val):  
        self.val = val  
        self.left, self.right = None, None
```

常用的实现办法

- AVL树 --> https://en.wikipedia.org/wiki/AVL_tree
- 红黑树(Red Black Tree) --> http://blog.csdn.net/v_july_v/article/details/6105630

Java中的 TreeSet / TreeMap

TreeSet / TreeMap 是底层运用了[红黑树](#)的数据结构

对比 HashSet / HashMap

- HashSet / HashMap 存取的时间复杂度为 **O(1)**, 而 TreeSet / TreeMap 存取的时间复杂度为 **O(logn)** 所以在存取上并不占优。
- HashSet / HashMap 内元素是无序的，而 TreeSet / TreeMap 内部是有序的(可以是按自然顺序排列也可以自定义排序)。
- TreeSet / TreeMap 还提供了类似 `lowerBound` 和 `upperBound` 这两个其他数据结构没有的方法
 - 对于 TreeSet, 实现上述两个方法的方法为：
 - `lowerBound`
 - `public E lower(E e)` --> 返回set中严格小于给出元素的最大元素, 如果没有满足条件的元素则返回 `null`.
 - `public E floor(E e)` --> 返回set中不大于给出元素的最大元素, 如果没有满足条件的元素则返回 `null`.
 - `upperBound`
 - `public E higher(E e)` --> 返回set中严格大于给出元素的最小元素, 如果没有满足条件的元素则返回 `null`.
 - `public E ceiling(E e)` --> 返回set中不小于给出元素的最小元素, 如果没有满足条件的元素则返回 `null`.
 - 对于 TreeMap, 实现上述两个方法的方法为：
 - `lowerBound`
 - `public Map.Entry<K,V> lowerEntry(K key)` --> 返回map中严格小于给出的key值的最大key对应的key-value对, 如果没有满足条件的key则返回 `null`.
 - `public K lowerKey(K key)` --> 返回map中严格小于给出的key值的最大key, 如果没有满足条件的key则返回 `null`.
 - `public Map.Entry<K,V> floorEntry(K key)` --> 返回map中不大于给出的key值的最大key对应的key-value对, 如果没有满足条件的key则返回 `null`.
 - `public K floorKey(K key)` --> 返回map中不大于给出的key值的最大key, 如果没有满足条件的key则返回 `null`.
 - `upperBound`
 - `public Map.Entry<K,V> higherEntry(K key)` --> 返回map中严格大于给出的key值的最小key对应的key-value对, 如果没有满足条件的key则返回 `null`.
 - `public K higherKey(K key)` --> 返回map中严格大于给出的key值的最小key, 如果没有满足条件的key则返回 `null`.
 - `public Map.Entry<K,V> ceilingEntry(K key)` --> 返回map中不小于给出的key值的最小key对应的key-value对, 如果没有满足条件的key则返回 `null`.
 - `public K ceilingKey(K key)` --> 返回map中不小于给出的key值的最小key, 如果没有满足条件的key则返回 `null`.

对比 PriorityQueue(Heap)

PriorityQueue是基于Heap实现的，它可以保证队头元素是优先级最高的元素，但其余元素是不保证有序的。

- 方法时间复杂度对比：
 - 添加元素 add() / offer()
 - TreeSet: O(logn)
 - PriorityQueue: O(logn)
 - 删除元素 poll() / remove()
 - TreeSet: O(logn)
 - PriorityQueue: O(n)
 - 查找 contains()
 - TreeSet: O(logn)
 - PriorityQueue: O(n)
 - 取最小值 first() / peek()
 - TreeSet: O(logn)
 - PriorityQueue: O(1)

常见用法

比如滑动窗口需要保证有序，那么这时可以用到TreeSet,因为TreeSet是有序的，并且不需要每次移动窗口都重新排序，只需要插入和删除(O(logn))就可以了。

注：在 C++ 中类似的结构为 set / map。在Python中没有内置的TreeSet、 TreeMap，需要使用第三方库或者自己实现。

练习

<http://www.lintcode.com/problem/consistent-hashing-ii/>

练习：链表转平衡排序二叉树

题目描述

将有序链表转换为平衡的排序二叉树。

LintCode 练习地址：

<http://www.lintcode.com/en/problem/convert-sorted-list-to-balanced-bst/>

粗暴的算法

可以十分容易想到一个一个 $O(n \log n)$ 的分治算法，以链表作为参数，二叉树作为返回值：

Java:

```
TreeNode convert(ListNode head) {
    if (head == null) {
        return null;
    }

    // find the following three nodes in the linked list
    // .... prev -> mid -> next ...
    // prev.next = null; // break the connect between prev & mid

    TreeNode root = new TreeNode(mid.val);
    root.left = convertListBBT(head);
    root.right = convertListBBT(next);
    return root;
}
```

Python:

```
def convert(head):
    if not head:
        return null

    # find the following three nodes in the linked list
    # .... prev -> mid -> next ...
    # prev.next = null; // break the connect between prev & mid

    root = TreeNode(mid.val)
    root.left = convertListBBT(head)
    root.right = convertListBBT(next)
    return root
```

算法的大致思路就是，找到链表中点和他前后的点，然后左边的部分递归生成一棵左子树，右边的部分递归生成一棵右子树，再和中间的点拼接起来就好了。

这个算法我们不难发现他的时间复杂度是 $O(n \log n)$ 的，因为找到中点的时间复杂度是 $O(n)$ ，因此可以用 T 函数推算法来进行推算：

$$T(n) = 2 * T(n/2) + O(n) = O(n \log n)$$

优化的算法

为了优化这个算法，我们给分治函数带上了一个参数 n 代表目前打算去转换 head 开始，长度为 n 那么多个节点，让其变为 Balanced Binary Tree。
递归函数接口如下：

Java:

```
TreeNode convert(ListNode head, int n)
```

Python

```
"""
Returns a TreeNode.
"""
def convert(head, n):
```

这样，我们不用真正把链表从 prev 和 mid 之间断开。可以利用对第二个参数的大小控制来让处理规模缩小。

但是虽然我们可以很快的调用 `convert(head, n / 2)`，让链表的一半变成二叉树。但是如何很快知道链表的中点呢？这里的办法是，如果我们把 head 放在参数里，那么就无法利用 convert 函数对 head 进行挪动了，所以我们把 head 挪出来，放到全局，作为一个全局变量。这样之后函数的接口改为：

Java:

```
public class Solution {
    private ListNode current;

    private TreeNode convert(int n) {
        // ...
    }

    // the entry point to public
    public TreeNode sortedListToBST(ListNode head) {
        current = head;
        convert(getLength(head));
    }
}
```

Python:

```
class Solution:
    def __init__(self):
        self.current = None

    def convert(self, n):
        # ...

    def sortedListToBST(self, head):
        self.current = head
        self.convert(getLength(head));
```

这里我们在全局放了一个 current 指针，这个指针会指向当前还没有被变成 Tree 的下一个 List 上的节点。因此如果我们把左子树变成 Tree 以后，current 就要让他指向 List 上的下一个点，也就是中间的这个点了。

算法有一些绕，建议使用几个小数据模拟整个算法的执行过程。

完整参考程序见：

<http://www.jiuzhang.com/solution/convert-sorted-list-to-balanced-bst/>

完整算法描述如下：

1. 首先求得整个list的长度 $O(n)$
2. 利用 helper 函数进行递归，`helper(head, len)` 表示把从 head 开始的，长度为 len 的链表，转换为一个bst并且return。与此同时，把global variable的指针挪到head开始的第 $len + 1$ 个listnode上。

那么 `convert(head, len)` 就可以分为，三个步骤：

1. 把head开头的长度为 $len/2$ 的先变成bst，也就是我们的左子树，`convert(head, len / 2)`。这个时候他顺便会把global variable 挪到第 $len / 2 + 1$ 的那个node，这个就是我们的root。
2. 然后得到了root之后，把global variable 往下挪一个挪到 第 $len/2 + 2$ 个点，也就是右子树开头的那个点，然后调用 `convert(global variable, len - len/2 -1)`，构造出右子树。
3. 然后把root，左子树，右子树，拼接在一起，return

这个题算法框架就是这样，如果不是很明白的话，建议模拟一个小数据，比如 5个节点的情况。模拟几个数据结合算法的思路来分析，就应该可以明白。这个题的这种解法背下来就好了。

第六章 基于组合的DFS

在非二叉树上的深度优先搜索（Depth-first Search）中，90%的问题，不是求组合（Combination）就是求排列（Permutation）。特别是组合类的深度优先搜索的问题特别的多。

本章节的先修内容有：

- 通过全子集问题 Subsets 了解组合类搜索的两种形式
- 通过全子集问题 II 了解如何在搜索中去重

课后补充内容有：

- 使用非递归的方法实现全子集问题

全子集问题

为了开始学习组合类的深度优先搜索，让我们先来做一个入门练习题：全子集问题。

LintCode 练习地址：

<http://www.lintcode.com/problem/subsets/>

题目的意思就是求出一个集合的所有子集。假设这个集合中是没有重复元素的。你可能已经会做这个问题，但是你知道么，这个问题存在 4 种解法么？

我们将从下面的 3 个方面来讲解这个问题：

1. 如何用最简单的递归方式来实现？
2. 如何用可以推广到排列类搜索问题的递归方式来实现？
3. 如果集合中有重复元素如何处理？

完整的 4 种解法的参考代码

<http://www.jiuzhang.com/solution/subsets/>

非递归的实现方法，见课后补充内容。

最简单的递归方式

可以拓展到排列类搜索的递归方式

全子集 Follow up I: 如何去重

什么是 Deep Copy ?

在 Subsets 的 Java 实现中，我们用到了如下的代码记录每一个找到的集合

Java:

```
results.add(new ArrayList<Integer>(subset));
```

事实上，这句话是调用了 `ArrayList` 的一个构造函数（Constructor），这个构造函数可以接受另外一个 `ArrayList` 作为其初始化的状态。

这种方式，我们叫它 **深度拷贝**（Deep Copy），又叫做硬拷贝（Hard Copy）或者克隆（Clone，名字多得老忘记不住啊）。与之对应的就有 **软拷贝**（Soft copy），又名引用拷贝（Reference Copy）。

在 Python 中，也有如下类似代码：

Python:

```
results.append(list(s)) # S is a set()
```

这新建了一个 `list`，`list` 的构造接受一个 `Iterable` 对象作为参数，并将该对象内的元素按顺序添加到新建的 `list` 中。这也是一次 Deep Copy。

不使用 Deep copy 会怎样呢？

我们来看看不使用 Deep copy 会怎样：

Java:

```
List<Integer> subset = new ArrayList<>();
subset.add(1); // 此时 subset 是 [1]

List<List<Integer>> results = new ArrayList<>();
results.add(subset); // 此时 results 是 [[1]]

subset.add(2); // 此时 subset 是 [1, 2]
results.add(subset); // 此时你以为 results 是 [[1], [1, 2]] 而事实上他是 [[1, 2], [1, 2]]

subset.add(3); // 此时 results 里是 [[1, 2, 3], [1, 2, 3]]
```

Python:

```
subset = []
subset.append(1) # 此时subset是[1]

results = []
results.append(subset) # 此时results是[[1]]

subset.append(2) # 此时subset是[1, 2]
results.append(subset) # 此时你以为results是[[1], [1, 2]]而事实上他是[[1, 2], [1, 2]]

subset.append(3) # 此时results里是[[1, 2, 3], [1, 2, 3]]
```

我们看到由于每一次 results.add 都是加入了相同的变量 subset，因此如果 subset 有变化，那么 result 里的记录就会同步的发生变化。原因是 results.add(subset) 加入的是 subset 的 reference，也就是 subset 在内存中的地址。那么事实上，当 results 里有两个 subset 的时候，相当于存储的是两个内存地址，而这两个内存地址又是一样的，才会导致如果这个内存地址里存的东西发生了变化，results 看起来就每个元素都发生了变化。

参数中引用传递

来看这段代码

Java:

```
public void func(List<Integer> subset) {
    subset.add(1);
}
public void main() {
    List<Integer> subset = new ArrayList<>();
    // 此时 subset 是 []
    func(subset);
    // 此时 subset 就是 [1] 了
}
```

Python:

```
def func(subset): # subset is a list
    subset.append(1)

def main():
    subset = []
    # 此时subset是[]
    func(subset)
    # 此时subset就是[1]了
```

可能你会奇怪，不是说修改参数不会影响到函数之外的参数么？也就是：

Java:

```
public void func(int x) {
    x = x + 1;
}
public void main() {
    int x = 0;
    func(x);
    // 此时 x 仍然是 0
}
```

Python:

```
def func(x):
    x = x+1

def main():
    int x = 0
    func(x)
    # 此时x仍然是0
```

上面两者的区别在于，人们习惯性的认为 `subset.add` 和 `x = x + 1` 都是对参数进行了修改。而事实上，`x = x + 1` 确实是对参数进行了修改，这个修改只在函数func的局部有效，出了func回到main就失效了。而 `subset.add` 并没有修改 `subset` 这个参数本身，而只是在 `subset` 所指向的内存空间中增加了一个新的元素，这个操作是永久性的，不是临时的，是全局有效的，不是局部有效的。那么怎么样才是对 `subset` 这个参数进行了修改呢？比如：

Java:

```
public void func(List<Integer> subset) {
    subset = new ArrayList<Integer>();
    subset.add(1);
}
public void main() {
    List<Integer> subset = new ArrayList<>();
    // 此时 subset 是 []
    func(subset);
    // 此时 subset 还是 []
}
```

Python:

```
def func(subset):
    subset = list(subset)
    subset.append(1)

def main():
    subset = []
    # 此时 subset 是 []
    func(subset)
    # 此时 subset 还是 []
}
```

我们可以看到如果你的修改操作是 `参数x = ...` 那么这才是对参数x的修改，而 `参数x.call_method()` 并不是对参数 x 本身的修改。

全子集 Follow up II: 如何非递归？

用非递归（Non-recursion / Iteration）的方式实现全子集问题，有两种方式：

1. 进制转换（binary）
2. 宽度优先搜索（Breadth-first Search）

进制转换的方法

九章微课堂 - 《位运算入门》 中有此方法的详细讲解：

<http://www.jiuzhang.com/tutorial/bit-manipulation/83>

参考代码如下：

Java:

```
class Solution {  
    /**  
     * @param S: A set of numbers.  
     * @return: A list of lists. All valid subsets.  
     */  
    public List<List<Integer>> subsets(int[] nums) {  
        List<List<Integer>> result = new ArrayList<List<Integer>>();  
        int n = nums.length;  
        Arrays.sort(nums);  
  
        // 1 << n is 2^n  
        // each subset equals to an binary integer between 0 .. 2^n - 1  
        // 0 -> 000 -> []  
        // 1 -> 001 -> [1]  
        // 2 -> 010 -> [2]  
        // ..  
        // 7 -> 111 -> [1, 2, 3]  
        for (int i = 0; i < (1 << n); i++) {  
            List<Integer> subset = new ArrayList<Integer>();  
            for (int j = 0; j < n; j++) {  
                // check whether the jth digit in i's binary representation is 1  
                if ((i & (1 << j)) != 0) {  
                    subset.add(nums[j]);  
                }  
            }  
            result.add(subset);  
        }  
        return result;  
    }  
}
```

Python:

```
class Solution:
    def subsets(self, nums):
        result = []
        n = len(nums)
        nums.sort()

        # 1 << n is 2^n
        # each subset equals to an binary integer between 0 .. 2^n - 1
        # 0 -> 000 -> []
        # 1 -> 001 -> [1]
        # 2 -> 010 -> [2]
        # ...
        # 7 -> 111 -> [1, 2, 3]
        for i in range(1 << n):
            subset = []
            for j in range(n):
                if (i & (1 << j)) != 0:
                    subset.append(nums[j])
            result.append(subset)
        return result
```

基于 BFS 的方法

在 BFS 那节课的讲解中，我们很少提到用 BFS 来解决找所有的方案的问题。事实上 BFS 也是可以用来做这件事情的。

用 BFS 来解决该问题时，层级关系如下：

```
第一层: []
第二层: [1] [2] [3]
第三层: [1, 2] [1, 3], [2, 3]
第四层: [1, 2, 3]
```

每一层的节点都是上一层的节点拓展而来。

参考代码如下：

Java:

```
public class Solution {

    /*
     * @param nums: A set of numbers
     * @return: A list of lists
     */
    public List<List<Integer>> subsets(int[] nums) {
        // List vs ArrayList (google)
        List<List<Integer>> results = new LinkedList<>();

        if (nums == null) {
            return results; // 空列表
        }

        Arrays.sort(nums);

        // BFS
        Queue<List<Integer>> queue = new LinkedList<>();
        queue.offer(new ArrayList<Integer>());

        while (!queue.isEmpty()) {
            List<Integer> subset = queue.poll();
            results.add(subset);

            for (int i = 0; i < nums.length; i++) {
                if (subset.size() == 0 || subset.get(subset.size() - 1) < nums[i]) {
                    List<Integer> nextSubset = new ArrayList<Integer>(subset);
                    nextSubset.add(nums[i]);
                    queue.offer(nextSubset);
                }
            }
        }

        return results;
    }
}
```

Python:

```
class Solution:
    def subsets(self, nums):
        results = []

        if not nums:
            return results

        nums.sort()

        # BFS
        queue = deque()
        queue.append([])

        while queue:
            subset = queue.popleft()
            results.append(subset)

            for i in range(len(nums)):
                if not subset or subset[-1] < nums[i]:
                    nextSubset = list(subset)
                    nextSubset.append(nums[i])
                    queue.append(nextSubset)

        return results
```

第七章 基于排列、图的DFS

本章节的先修知识有：

1. 全排列问题如何使用深度优先搜索来实现？和全子集问题的异同在哪儿？
2. 全排列问题的 Follow up: Permutation II。如何去重？
3. 如何求一个排列的下一个排列？

课后补充内容有：

1. 如何求一个排列是第几个排列？

如何求下一个排列

问题描述

给定一个若干整数的排列，给出按整数大小进行字典序从小到大排序后的下一个排列。若没有下一个排列，则输出字典序最小的序列。

例如 $1, 2, 3 \rightarrow 1, 3, 2, 3, 2, 1 \rightarrow 1, 2, 3, 1, 1, 5 \rightarrow 1, 5, 1$

原题链接：

<http://www.lintcode.com/problem/next-permutation-ii/>

<http://www.lintcode.com/problem/next-permutation/>

(两题类似，一个要求原地修改，一个要求返回新的排列)

算法描述

如果上来想不出方法，可以试着找找规律，我们关注的重点应是原数组末尾。

从未尾往左走，如果一直递增，例如 $\dots, 9, 7, 5$ ，那么下一个排列一定会牵扯到左边更多的数，直到一个非递增数为止，例如 $\dots, 6, 9, 7, 5$ 。对于原数组的变化就只到 6 这里，和左侧其他数再无关系。 6 这个位置会变成 6 右侧所有数中比 6 大的最小的数，而 6 会进入最后3个数中，且后3个数必是升序数组。

所以算法步骤如下：

- 从右往左遍历数组 nums ，直到找到一个位置 i ，满足 $\text{nums}[i] > \text{nums}[i - 1]$ 或者 i 为 0。
- i 不为0时，用 j 再次从右到左遍历 nums ，寻找第一个 $\text{nums}[j] > \text{nums}[i - 1]$ 。而后交换 $\text{nums}[j]$ 和 $\text{nums}[i - 1]$ 。注意，满足要求的 j 一定存在！且交换后 $\text{nums}[i]$ 及右侧数组仍为降序数组。
- 将 $\text{nums}[i]$ 及右侧的数组翻转，使其升序。

Q： i 为 0 怎么办？

A： i 为 0 说明整个数组是降序的，直接翻转整个数组即可。

Q：有重复元素怎么办？

A：在遍历时只要严格满足 $\text{nums}[i] > \text{nums}[i - 1]$ 和 $\text{nums}[j] > \text{nums}[i - 1]$ 就不会有问题是。

Q：元素过少是否要单独考虑？

A：当元素个数小于等于1个时，可以直接返回。

参考代码

Java:

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers that's next permutation
     */
    // 用于交换nums[i]和nums[j]
    public void swapItem(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
    // 用于翻转nums[i]到nums[j]，包含两端的这一小段数组
    public void swapList(int[] nums, int i, int j) {
        while (i < j) {
            swapItem(nums, i, j);
            i++;
            j--;
        }
    }
    public void nextPermutation(int[] nums) {
        int len = nums.length;
        if (len <= 1) {
            return;
        }
        int i = len - 1;
        while (i > 0 && nums[i] <= nums[i - 1]) {
            i--;
        }
        if (i != 0) {
            int j = len - 1;
            while (nums[j] <= nums[i - 1]) {
                j--;
            }
            swapItem(nums, j, i-1);
        }
        swapList(nums, i, len - 1);
    }
}
```

Python:

```
class Solution:
    # 用于翻转nums[i]到nums[j]，包含两端的这一小段数组
    def swapList(self, nums, i, j):
        while i < j:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
            j -= 1

    """
    @param nums: An array of integers
    @return: nothing
    """

    def nextPermutation(self, nums):
        n = len(nums)
        if n <= 1:
            return

        i = n-1
        while i > 0 and nums[i] <= nums[i-1]:
            i -= 1

        if i != 0:
            j = n-1
            while nums[j] <= nums[i-1]:
                j -= 1
            nums[i], nums[j] = nums[j], nums[i]
        self.swapList(nums, i, n-1)
```

全排列问题

全排列问题是“排列式”深度优先搜索问题的鼻祖。很多搜索的问题都可以用类似全排列的代码来完成。包括我们前面学过的全子集问题的一种做法。

这一小节中我们需要掌握：

1. 普通的全排列问题怎么做
2. 有重复的全排列问题怎么做？如何在搜索类问题中去重？
3. 如何实现一个非递归的全排列算法？

普通的全排列问题

有重复数的全排列问题

如何实现非递归版本的全排列问题？

基本思路

非递归的全排列，采用的是迭代方式，在[如何求下一个排列](#)中，我们讲过如何求下一个排列，那么我们只需要不断调用这个 `nextPermutation` 方法即可。

一些可以做得更细致的地方：

- 为了确定何时结束，建议在迭代前，先对输入 `nums` 数组进行升序排序，迭代到降序时，就都找完了。有心的同学可能还记得在 `nextPermutation` 当中，当且仅当数组完全降序，那么从右往左遍历的指针 `i` 最终会指向0。所以可以为 `nextPermutation` 带上布尔返回值，当 `i` 为0时，返回 `false`，表示找完了。要注意，排序操作在这样一个NP问题中，消耗的时间几乎可以忽略。
- 当数组长度为1时，`nextPermutation` 会直接返回 `false`；当数组长度为0时，`nextPermutation` 中 `i` 会成为-1，所以返回 `false` 的条件可以再加上 `i` 为 `-1`。
- Java中，如果输入类型是 `int[]`，而输出类型是 `List<List<Integer>>`，要注意，并没有太好的方法进行类型转换，这是由于 `int` 是基本类型。建议还是自行手动复制，实际工作中还可使用 `guava` 库。

核心代码

Java:

```
class Solution {
    public boolean nextPermutation(int[] nums) {
        int len = nums.length;
        int i = len - 1;
        while (i > 0 && nums[i] <= nums[i - 1]) {
            i--;
        }
        if (i <= 0) {
            return false;
        }
        /*
         * 所有剩余代码参见 http://www.jiuzhang.com/tutorial/algorithm/439
        */
        return true;
    }

    public List<List<Integer>> permute(int[] A) {
        Arrays.sort(A);
        List<List<Integer>> result = new ArrayList<>();

        boolean next = true; // next 为 true 时，表示可以继续迭代
        while (next) {
            List<Integer> current = new ArrayList<>(); // 进行数组复制
            for (int a : A) {
                current.add(a);
            }

            result.add(current);
            next = nextPermutation(A);
        }
        return result;
    }
}
```

Python:

```
class Solution:
    def nextPermutation(self, nums):
        n = len(nums)
        if n <= 1:
            return

        i = n-1
        while i > 0 and nums[i] <= nums[i-1]:
            i -= 1

        if i <= 0:
            return False

        # 所有剩余代码参见http://www.jiuzhang.com/tutorial/algorithm/439
        return True

    def permute(self, A):
        A.sort()
        result = []

        hasNext = True # hasNext 为 true 时，表示可以继续迭代
        while hasNext:
            current = list(A) # 进行数组复制
            result.append(current)
            hasNext = self.nextPermutation(A)

        return result
```

如何求一个排列是第几个排列？

题目描述

给出一个不含重复数字的排列，求这些数字的所有排列按字典序排序后该排列的编号，编号从1开始。

例如排列 [1, 2, 4] 是第 1 个排列。

<http://www.lintcode.com/zh-cn/problem/permutation-index/>

算法描述

只需计算有多少个排列在当前排列 A 的前面即可。如何算呢？举个例子，[3, 7, 4, 9, 1]，在它前面的必然是某位置 i 对应元素比原数组小，而 i 左侧和原数组一样。也即 [3, 7, 4, 1, X]，[3, 7, 1, X, X]，[3, 1或4, X, X, X]，[1, X, X, X, X]。

而第 i 个元素，比原数组小的情况有多少种，其实就是 A[i] 右侧有多少元素比 A[i] 小，乘上 A[i] 右侧元素全排列数，即 A[i] 右侧元素数量的阶乘。i 从右往左看，比当前 A[i] 小的右侧元素数量分别为 1, 1, 2, 1，所以最终字典序在当前 A 之前的数据为 $1 \times 1! + 1 \times 2! + 2 \times 3! + 1 \times 4! = 39$ ，故当前 A 的字典序为 40。

具体步骤：

- 用 permutation 表示当前阶乘，初始化为 1，result 表示最终结果，初始化为 0。由于最终结果可能巨大，所以用 long 类型。
- i 从右往左遍历 A，循环中计算 A[i] 右侧有多少元素比 A[i] 小，计为 smaller，result += smaller * permutation。之后 permutation *= A.length - i，为下次循环 i 左移一位后的排列数。
- 已算出多少字典序在 A 之前，返回 result+1。

参考代码

Java:

```
public class Solution {  
    /**  
     * @param A: An array of integers  
     * @return: A long integer  
     */  
    public long permutationIndex(int[] A) {  
        // write your code here  
        long permutation = 1;  
        long result = 0;  
        for (int i = A.length - 2; i >= 0; --i) {  
            int smaller = 0;  
            for (int j = i + 1; j < A.length; ++j) {  
                if (A[j] < A[i]) {  
                    smaller++;  
                }  
            }  
            result += smaller * permutation;  
            permutation *= A.length - i;  
        }  
        return result + 1;  
    }  
}
```

Python:

```
class Solution:  
    """  
    @param A: An array of integers  
    @return: A long integer  
    """  
  
    def permutationIndex(self, A):  
        permutation = 1  
        result = 0  
        for i in range(len(A) - 2, -1, -1):  
            smaller = 0  
            for j in range(i + 1, len(A)):  
                if A[j] < A[i]:  
                    smaller += 1  
            result += smaller * permutation  
            permutation *= len(A) - i  
        return result + 1
```

Q：为了找寻每个元素右侧有多少元素比自己小，用了 $O(n^2)$ 的时间，能不能更快些？

A：可以做到 $O(n \log n)$ ！但是很复杂，这是另外一个问题了，可以使用BST，归并排序或者线段树，详见<http://www.lintcode.com/zh-cn/problem/count-of-smaller-number-before-itself/>

Q：元素有重复怎么办？

A：好问题！元素有重复，情况会复杂的多。因为这会影响 $A[i]$ 右侧元素的排列数，此时的排列数计算方法为总元素数的阶乘，除以各元素值个数的阶乘，例如 $[1, 1, 1, 2, 2, 3]$ ，排列数为 $6! \div (3! \times 2! \times 1!)$ 。

为了正确计算阶乘数，需要用哈希表记录 $A[i]$ 及右侧的元素值个数，并考虑到 $A[i]$ 与右侧比其小的元素 $A[k]$ 交换后，要把 $A[k]$ 的计数减一。用该哈希表计算正确的阶乘数。

而且要注意，右侧比 $A[i]$ 小的重复元素值只能计算一次，不要重复计算！

第八章 数据结构：栈，队列，哈希表，堆

栈 Stack

我们在前面的二叉树的学习中，已经学习了如何使用 Stack 来进行非递归的二叉树遍历。

这里我们来看看栈在面试中的其他一些考点和考题：

1. 如果自己实现一个栈？
2. 如何用两个队列实现一个栈？
3. 用一个数组如何实现三个栈？

什么是栈 (Stack) ?

栈 (stack) 是一种采用后进先出 (LIFO, last in first out) 策略的抽象数据结构。比如物流装车，后装的货物先卸，先装的货物后卸。栈在数据结构中的地位很重要，在算法中的应用也很多，比如用于非递归的遍历二叉树，计算逆波兰表达式，等等。

栈一般用一个存储结构（常用数组，偶见链表），存储元素。并用一个指针记录栈顶位置。栈底位置则是指栈中元素数量为0时的栈顶位置，也即栈开始的位置。

栈的主要操作：

- `push()`，将新的元素压入栈顶，同时栈顶上升。
- `pop()`，将新的元素弹出栈顶，同时栈顶下降。
- `empty()`，栈是否为空。
- `peek()`，返回栈顶元素。

在各语言的标准库中：

- Java，使用 `java.util.Stack`，它是扩展自 `Vector` 类，支持 `push()`，`pop()`，`peek()`，`empty()`，`search()` 等操作。
- C++，使用 `<stack>` 中的 `stack` 即可，方法类似Java，只不过C++中 `peek()` 叫做 `top()`，而且 `pop()` 时，返回值为空。
- Python，直接使用 `list`，查看栈顶用 `[-1]` 这样的切片操作，弹出栈顶时用 `list.pop()`，压栈时用 `list.append()`。

如何自己实现一个栈？

参见问题：<http://www.lintcode.com/en/problem/implement-stack/>

这里给出一种用 `ArrayList` 的通用实现方法。（注意：为了将重点放在栈结构上，做了适当简化。该栈仅支持整数类型，若想实现泛型，可用反射机制和object对象传参；此外，可多做安全检查并抛出异常）

Java:

```
public class Stack {  
  
    List<Integer> array = new ArrayList<Integer>();  
  
    // 压入新元素  
    public void push(int x) {  
        array.add(x);  
    }  
  
    // 栈顶元素弹出  
    public void pop() {  
        if (!isEmpty()) {  
            array.remove(array.size() - 1);  
        }  
    }  
  
    // 返回栈顶元素  
    public int top() {  
        return array.get(array.size() - 1);  
    }  
  
    // 判断是否是空栈  
    public boolean isEmpty() {  
        return array.size() == 0;  
    }  
}
```

```

class Stack:
    def __init__(self):
        self.array = []

    # 压入新元素
    def push(self, x):
        self.array.append(x)

    # 栈顶元素出栈
    def pop(self):
        if not self.isEmpty():
            self.array.pop()

    # 返回栈顶元素
    def top(self):
        return self.array[-1]

    # 判断是否是空栈
    def isEmpty(self):
        return len(self.array) == 0

```

栈在计算机内存当中的应用

我们在程序运行时，常说的内存中的堆栈，其实就是栈空间。这一段空间存放着程序运行时，产生的各种临时变量、函数调用，一旦这些内容失去其作用域，就会被自动销毁。

函数调用其实是栈的很好的例子，后调用的函数先结束，所以为了调用函数，所需要的内存结构，栈是再合适不过了。在内存当中，**栈从高地址不断向低地址扩展**，随着程序运行的层层深入，栈顶指针不断指向内存中更低的地址。

相关参考资料：

https://blog.csdn.net/liu_yude/article/details/45058687

如何用两个队列实现一个栈？

算法步骤

队列的知识请看：<http://www.jiuzhang.com/tutorial/algorithms/391>

两个队列实现一个栈，其实并没有什么优雅的办法。就看大家怎么去写这个东西了。

- 构造的时候，初始化两个队列，`queue1`，`queue2`。`queue1` 主要用来存储，`queue2` 则主要用来帮助 `queue1` 弹出元素以及访问栈顶。
- `push`：将元素推入 `queue1` 当中。
- `pop`：注意要弹出的元素在 `queue1` 末端，故将 `queue1` 中元素弹出，并直接推入 `queue2`，当 `queue1` 只剩一个元素时，把它 `pop` 出来，并作为结果。而后交换两个队列。
- `top`：类似 `pop`，不过不扔掉 `queue1` 中最后一个元素，而是把它也推入 `queue2` 当中。
- `isEmpty`：判断 `queue1` 是否为空即可。

参考代码

Java:

```
public class Stack {
    public Queue<Integer> queue1 = new LinkedList<Integer>();
    public Queue<Integer> queue2 = new LinkedList<Integer>();

    // 将queue1中元素移入queue2,留下最后一个。
    public void moveItems() {
        while (queue1.size() != 1) {
            queue2.offer(queue1.poll());
        }
    }

    // 交换两个队列
    public void swapQueues() {
        Queue<Integer> temp = queue1;
        queue1 = queue2;
        queue2 = temp;
    }

    public void push(int x) {
        queue1.offer(x);
    }

    public void pop() {
        moveItems();
        queue1.poll();
        swapQueues();
    }

    public int top() {
        moveItems();
        int item = queue1.poll();
        swapQueues();
        queue1.offer(item);
        return item;
    }

    public boolean isEmpty() {
        return queue1.isEmpty();
    }
}
```

Python:

```
from collections import deque

class Stack:
    def __init__(self):
        self.queue1 = deque()
        self.queue2 = deque()

    # 将queue1中元素移入queue2, 留下最后一个。
    def moveItems(self):
        while len(self.queue1) != 1:
            self.queue2.append(self.queue1.popleft())

    def swapQueues(self):
        self.queue1, self.queue2 = self.queue2, self.queue1

    def push(self, x):
        self.queue1.append(x)

    def pop(self):
        self.moveItems()
        self.queue1.popleft()
        self.swapQueues()

    def top(self):
        self.moveItems()
        item = self.queue1.popleft()
        self.swapQueues()
        self.queue1.append(item)
        return item

    def isEmpty(self):
        return len(self.queue1) == 0
```

相关题目

<http://www.lintcode.com/zh-cn/problem/implement-stack-by-two-queues/>

如何用一个数组实现三个栈？

题目描述

用一个数组实现三个栈。你可以假设这三个栈都一样大并且足够大。你不需要担心如果一个栈满了之后怎么办。

详见：<http://www.lintcode.com/zh-cn/problem/implement-three-stacks-by-single-array/>

算法描述

这道题的本质是把数组索引当作地址，用链表来实现栈。数组 `buffer` 中的每一个元素，并不能单单是简单的int类型，而是一个链表中的节点，它包含值 `value`，栈中向栈底方向的之前元素索引 `prev`，向栈顶方向的后来元素索引 `next`。

在该三栈数据结构中，要记录三个栈顶指针 `stackPointer`，也就是三个栈顶所在的数组索引，通过这三个栈顶节点，能够用 `prev` 找到整串栈。

此外还要用 `indexUsed` 记录整个数组中的所用的索引数。其实也就是下一次 `push` 的时候，向数组的 `indexUsed` 位置存储。

具体操作：

- 构造：要初始化 `stackPointer` 为 `3个-1`，表示没有；`indexUsed=0`；`buffer` 为一个长度为三倍栈大小的数组。
- `push`：要把新结点new在 `buffer[indexUsed]`，同时修改该栈的 `stackPointer`，`indexUsed` 自增。注意修改当前栈顶结点 `prev` 和之前栈顶结点的 `next` 索引。
- `peek`：只需要返回 `buffer` 中对应的 `stackPointer` 即可。
- `isEmpty`：只需判断 `stackPointer` 是否为-1。
- `pop`：`pop`的操作较为复杂，因为有三个栈，所以栈顶不一定在数组尾端，`pop`掉栈顶之后，**数组中可能存在空洞**。而这个空洞又很难 `push` 入元素。所以，解决方法是，当要 `pop` 的元素不在数组尾端（即 `indexUsed-1`）时，交换这两个元素。**不过一定要注意，交换的时候，要注意修改这两个元素之前、之后结点的 `prev` 和 `next` 指针，使得链表仍然是正确的**，事实上这就是结点中 `next` 的作用——为了找到之后结点并修改它的 `prev`。在交换时，一种很特殊的情况是栈顶节点刚好是数组尾端元素的后继节点，这时需要做特殊处理。在交换完成后，就可以删掉数组尾端元素，并修改相应的 `stackPointer`、`indexUsed` 和新栈顶的 `next`。

参考代码

`pop`操作确实非常复杂，编写时如履薄冰。这里给出一种比较好的写法。

Java:

```
public class ThreeStacks {
    public int stackSize;
    public int indexUsed;
    public int[] stackPointer;
    public StackNode[] buffer;

    public ThreeStacks(int size) {
        // do intialization if necessary
        stackSize = size;
        stackPointer = new int[3];
        for (int i = 0; i < 3; ++i)
            stackPointer[i] = -1;
        indexUsed = 0;
        buffer = new StackNode[stackSize * 3];
    }

    public void push(int stackNum, int value) {
        // Write your code here
        // Push value into stackNum stack
        int lastIndex = stackPointer[stackNum];
        stackPointer[stackNum] = indexUsed;
        indexUsed++;
        buffer[stackPointer[stackNum]] = new StackNode(lastIndex, value, -1);
        if (lastIndex != -1) {
            buffer[lastIndex].next = stackPointer[stackNum];
        }
    }

    public int pop(int stackNum) {
        // Write your code here
        // Pop and return the top element from stackNum stack
        int value = buffer[stackPointer[stackNum]].value;
        int lastIndex = stackPointer[stackNum];
        if (lastIndex != indexUsed - 1)
            swap(lastIndex, indexUsed - 1, stackNum);

        stackPointer[stackNum] = buffer[stackPointer[stackNum]].prev;
        if (stackPointer[stackNum] != -1)
            buffer[stackPointer[stackNum]].next = -1;

        buffer[indexUsed-1] = null;
        indexUsed--;
        return value;
    }

    public int peek(int stackNum) {
        // Write your code here
        // Return the top element
        return buffer[stackPointer[stackNum]].value;
    }
}
```

```

public boolean isEmpty(int stackNum) {
    // Write your code here
    return stackPointer[stackNum] == -1;
}

public void swap(int lastIndex, int topIndex, int stackNum) {
    if (buffer[lastIndex].prev == topIndex) {
        int tmp = buffer[lastIndex].value;
        buffer[lastIndex].value = buffer[topIndex].value;
        buffer[topIndex].value = tmp;
        int tp = buffer[topIndex].prev;
        if (tp != -1) {
            buffer[tp].next = lastIndex;
        }
        buffer[lastIndex].prev = tp;
        buffer[lastIndex].next = topIndex;
        buffer[topIndex].prev = lastIndex;
        buffer[topIndex].next = -1;
        stackPointer[stackNum] = topIndex;
        return;
    }

    int lp = buffer[lastIndex].prev;
    if (lp != -1)
        buffer[lp].next = topIndex;

    int tp = buffer[topIndex].prev;
    if (tp != -1)
        buffer[tp].next = lastIndex;

    int tn = buffer[topIndex].next;
    if (tn != -1)
        buffer[tn].prev = lastIndex;
    else {
        for (int i = 0; i < 3; ++i)
            if (stackPointer[i] == topIndex)
                stackPointer[i] = lastIndex;
    }

    StackNode tmp = buffer[lastIndex];
    buffer[lastIndex] = buffer[topIndex];
    buffer[topIndex] = tmp;
    stackPointer[stackNum] = topIndex;
}
}

class StackNode {
    public int prev, next;
    public int value;
    public StackNode(int p, int v, int n) {
        value = v;
        prev = p;
        next = n;
    }
}

```

Python:

```
class ThreeStacks:
    def __init__(self, size):
        self.stackSize = size
        self.stackPointer = [-1, -1, -1]
        self.indexUsed = 0
        self.buffer = [StackNode(-1, -1, -1) for _ in range(size*3)]

    def push(self, stackNum, value):
        lastIndex = self.stackPointer[stackNum]
        self.stackPointer[stackNum] = self.indexUsed
        self.indexUsed += 1
        self.buffer[self.stackPointer[stackNum]] = StackNode(lastIndex, value, -1)
        if lastIndex != -1:
            self.buffer[lastIndex].next = self.stackPointer[stackNum]

    def pop(self, stackNum):
        value = self.buffer[self.stackPointer[stackNum]].value
        lastIndex = self.stackPointer[stackNum]
        if lastIndex != self.indexUsed - 1:
            self.swap(lastIndex, self.indexUsed-1, stackNum)

        self.stackPointer[stackNum] = self.buffer[self.stackPointer[stackNum]].prev
        if self.stackPointer[stackNum] != -1:
            self.buffer[self.stackPointer[stackNum]].next = -1

        self.buffer[self.indexUsed-1] = None
        self.indexUsed -= 1
        return value

    def peek(self, stackNum):
        return self.buffer[self.stackPointer[stackNum]].value

    def isEmpty(self, stackNum):
        return self.stackPointer[stackNum] == -1

    def swap(self, lastIndex, topIndex, stackNum):
        if self.buffer[lastIndex].prev == topIndex:
            self.buffer[lastIndex].value, self.buffer[topIndex].value = self.buffer[topIndex].value, self.buffer[lastIndex].value
            tp = self.buffer[topIndex].prev
            if tp != -1:
                self.buffer[tp].next = lastIndex
            self.buffer[lastIndex].prev = tp
            self.buffer[lastIndex].next = topIndex
            self.buffer[topIndex].prev = lastIndex
            self.buffer[topIndex].next = -1
            self.stackPointer[stackNum] = topIndex
        return

        lp = self.buffer[lastIndex].prev
        if lp != -1:
            self.buffer[lp].next = topIndex

        tp = self.buffer[topIndex].prev
        if tp != -1:
            self.buffer[tp].next = lastIndex

        tn = self.buffer[topIndex].next
        if tn != -1:
            self.buffer[tn].prev = lastIndex
        else:
            for i in range(3):
                if self.stackPointer[i] == topIndex:
                    self.stackPointer[i] = lastIndex

        self.buffer[lastIndex], self.buffer[topIndex] = self.buffer[topIndex], self.buffer[lastIndex]
        self.stackPointer[stackNum] = topIndex

class StackNode:
    def __init__(self, p, v, n):
        self.value = v
        self.prev = p
        self.next = n
```

队列 Queue

队列通常被用作 BFS 算法的主要数据结构。除此之外面试中其他可能考察队列这个数据结构的地方并不多。我们这里把非 BFS 的队列考题考点做一个汇总：

1. 如何用链表实现队列？
2. 如何用两个栈实现一个队列？
3. 什么是循环数组，如何用循环数组实现队列？

如何用链表实现队列

定义

1. 队列为一种先进先出的线性表
2. 只允许在表的一端进行入队，在另一端进行出队操作。在队列中，允许插入的一端叫队尾，允许删除的一端叫队头，即入队只能从队尾入，出队只能从队头出

思路

1. 需要两个节点，一个头部节点，也就是dummy节点，它是在加入的第一个元素的前面，也就是`dummy.next=第一个元素`，这样做是为了方便我们删除元素，还有一个尾部节点，也就是tail节点，表示的是最后一个元素的节点
2. 初始时，tail节点跟dummy节点重合
3. 当我们要加入一个元素时，也就是从队尾中加入一个元素，只需要新建一个值为val的node节点，然后`tail.next=node`，再移动tail节点到`tail.next`
4. 当我们需要删除队头元素时，只需要将`dummy.next`变为`dummy.next.next`，这样就删掉了第一个元素，这里需要注意的是，如果删掉的是队列中唯一的一个元素，那么需要将tail重新与dummy节点重合
5. 当我们需要得到队头元素而不删除这个元素时，只需要获得`dummy.next.val`就可以了

示例代码

Java:

```
class QueueNode {
    public int val;
    public QueueNode next;
    public QueueNode(int value) {
        val = value;
    }
}

public class Queue {

    private QueueNode dummy, tail;

    public Queue() {
        dummy = new QueueNode(-1);
        tail = dummy;
    }

    public void enqueue(int val) {
        QueueNode node = new QueueNode(val);
        tail.next = node;
        tail = node;
    }

    public int dequeue() {
        int ele = dummy.next.val;
        dummy.next = dummy.next.next;

        if (dummy.next == null) {
            tail = dummy;// reset
        }
        return ele;
    }

    public int peek() {
        int ele = dummy.next.val;
        return ele;
    }

    public boolean isEmpty() {
        return dummy.next == null;
    }
}
```

Python:

```
class QueueNode:
    def __init__(self, value):
        self.val = value
        self.next = None

class Queue:
    def __init__(self):
        self.dummy = QueueNode(-1)
        self.tail = self.dummy

    def enqueue(self, val):
        node = QueueNode(val)
        self.tail.next = node
        self.tail = node

    def dequeue(self):
        ele = self.dummy.next.val
        self.dummy.next = self.dummy.next.next

        if not self.dummy.next:
            self.tail = self.dummy
        return ele

    def peek(self):
        return self.dummy.next.val

    def isEmpty(self):
        return self.dummy.next == None
```

如何用两个栈实现队列

我们已经知道，栈是一个先进后出的数据结构，而队列是一个先进先出的数据结构，那么如何用栈来实现队列呢？这里我们可以用两个栈来实现队列

思路：

1. 现在我们已经有了两个栈stack1和stack2，最暴力的做法，当需要往队列中加入元素时，可以往其中一个栈stack2中加入元素，当需要得到队头元素时，只需要将stack2中的元素倒入到stack1中，再取stack1的头元素就可以了，如果是需要删掉队头元素，那么直接pop stack1的栈顶元素就可以了，再将stack1中的元素再倒入到stack2中，以便下一次的加入元素。
2. 上面的实现中，我们每取一次队头元素或者删掉队头元素，都需要将stack2中的元素先倒入到stack1中，再从stack1中倒回去，每次需要倒两边十分麻烦，那么是否有更加简便一些的方法呢？答案当然是有的，其实当我们把stack2中的元素倒入到stack1中的时候，我们发现stack1中的元素的顺序就是按照队列的先进先出顺序，那么我们不再将stack1中的元素倒入到stack2中，在获取队头元素或者删除队头元素的时候，我们先判断stack1是否为空，如果不为空，从stack1中取即可，如果为空，那么将stack2中的元素倒入到stack1中，每次加入元素的时候都是往stack2中加入元素。

示例代码

Java:

```
public class MyQueue {  
    private Stack<Integer> stack1;  
    private Stack<Integer> stack2;  
  
    public MyQueue() {  
        stack1 = new Stack<Integer>();  
        stack2 = new Stack<Integer>();  
    }  
  
    private void stack2ToStack1() {  
        while (!stack2.empty()) {  
            stack1.push(stack2.peek());  
            stack2.pop();  
        }  
    }  
  
    public void push(int number) {  
        stack2.push(number);  
    }  
  
    public int pop() {  
        if (stack1.empty() == true) {  
            this.stack2ToStack1();  
        }  
        return stack1.pop();  
    }  
  
    public int top() {  
        if (stack1.empty() == true) {  
            this.stack2ToStack1();  
        }  
        return stack1.peek();  
    }  
}
```

Python:

```
class MyQueue:  
    def __init__(self):  
        self.stack1 = []  
        self.stack2 = []  
  
    def stack2ToStack1(self):  
        while self.stack2:  
            self.stack1.append(self.stack2.pop())  
  
    def push(self, number):  
        self.stack2.append(number)  
  
    def pop(self):  
        if not self.stack1:  
            self.stack2ToStack1()  
        return self.stack1.pop()  
  
    def top(self):  
        if not self.stack1:  
            self.stack2ToStack1()  
        return self.stack1[-1]
```

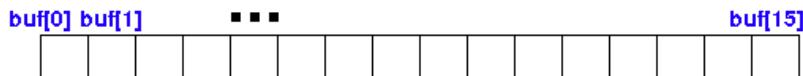
什么是循环数组，如何用循环数组实现队列？

什么是循环数组

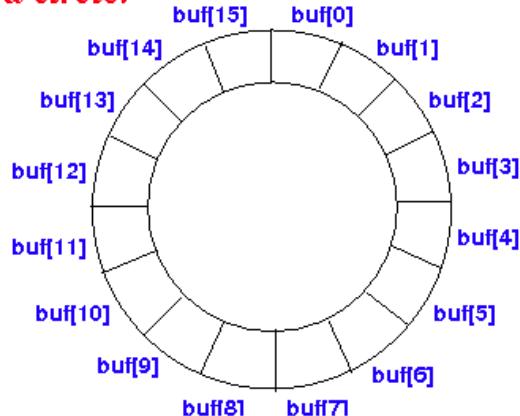
Circular array = a data structure that used a **array** as if it were **connected end-to-end**

可以图示为：

Array:



Pretend array is a circle:



如何实现队列

1. 我们需要知道队列的入队操作是只在队尾进行的，相对的出队操作是只在队头进行的，所以需要两个变量front与rear分别来指向队头与队尾
2. 由于是循环队列，我们在增加元素时，如果此时 $rear = array.length - 1$ ， $rear$ 需要更新为 0；同理，在元素出队时，如果 $front = array.length - 1$ ， $front$ 需要更新为 0。对此，我们可以通过对数组容量取模来更新。

示例代码

Java:

```
public class CircularQueue {  
  
    int[] circularArray;  
    int front;  
    int rear;  
    int size;  
    public CircularQueue(int n) {  
        // initialize your data structure here  
  
        this.circularArray = new int[n];  
        front = 0;  
        rear = 0;  
        size = 0;  
    }  
    /**  
     * @return: return true if the array is full  
     */  
    public boolean isFull() {  
        // write your code here  
        return size == circularArray.length;  
    }  
  
    /**  
     * @return: return true if there is no element in the array  
     */  
    public boolean isEmpty() {  
        // write your code here  
        return size == 0;  
    }  
  
    /**  
     * @param element: the element given to be added  
     * @return: nothing  
     */  
    public void enqueue(int element) {  
        // write your code here  
        if (isFull()) {  
            throw new RuntimeException("Queue is already full");  
        }  
        rear = (front + size) % circularArray.length;  
        circularArray[rear] = element;  
        size += 1;  
    }  
  
    /**  
     * @return: pop an element from the queue  
     */  
    public int dequeue() {  
        // write your code here  
        if (isEmpty()) {  
            throw new RuntimeException("Queue is already empty");  
        }  
        int ele = circularArray[front];  
        front = (front + 1) % circularArray.length;  
        size -= 1;  
        return ele;  
    }  
}
```

Python:

```
class CircularQueue:
    def __init__(self, n):
        self.circularArray = [0]*n
        self.front = 0
        self.rear = 0
        self.size = 0

    """
    @return: return true if the array is full
    """
    def isFull(self):
        return self.size == len(self.circularArray)

    """
    @return: return true if there is no element in the array
    """
    def isEmpty(self):
        return self.size == 0

    """
    @param element: the element given to be added
    @return: nothing
    """
    def enqueue(self, element):
        if self.isFull():
            raise RuntimeError("Queue is already full")
        self.rear = (self.front+self.size) % len(self.circularArray)
        self.circularArray[self.rear] = element
        self.size += 1

    """
    @return: pop an element from the queue
    """
    def dequeue(self):
        if self.isEmpty():
            raise RuntimeError("Queue is already empty")
        ele = self.circularArray[self.front]
        self.front = (self.front+1) % len(self.circularArray)
        self.size -= 1
        return ele
```

在线练习

<http://www.lintcode.com/zh-cn/problem/implement-queue-by-circular-array/#>

哈希表 Hash

哈希表（Java 中的 HashSet / HashMap，C++ 中的 unordered_map，Python 中的 dict）是面试中非常常见的数据结构。它的主要考点有两个：

1. 是否会灵活的使用哈希表解决问题
2. 是否熟练掌握哈希表的基本原理

这一小节中，我们将介绍一下哈希表原理中的几个重要的知识点：

1. 哈希表的工作原理
2. 为什么 hash 上各种操作的时间复杂度不能单纯的认为是 O(1) 的
3. 哈希函数（Hash Function）该如何实现
4. 哈希冲突（Collision）该如何解决
5. 如何让哈希表可以不断扩容？

哈希表的基本原理

哈希函数 Hash Function

冲突的解决办法

冲突（Collision），是说两个不同的 key 经过哈希函数的计算后，得到了两个相同的值。解决冲突的方法，主要有两种：

1. 开散列法（Open Hashing）。是指哈希表所基于的数组中，每个位置是一个 Linked List 的头结点。这样冲突的 `<key, value>` 二元组，就都放在同一个链表中。
2. 闭散列法（Closed Hashing）。是指在发生冲突的时候，后来的元素，往下一个位置去找空位。

重哈希 Rehashing

堆 Heap

Heap 的结构和原理

堆化 Heapify

基于 Siftup 的版本 O(nlogn)

Java版本：

```
public class Solution {  
    /**  
     * @param A: Given an integer array  
     * @return: void  
     */  
    private void siftup(int[] A, int k) {  
        while (k != 0) {  
            int father = (k - 1) / 2;  
            if (A[k] > A[father]) {  
                break;  
            }  
            int temp = A[k];  
            A[k] = A[father];  
            A[father] = temp;  
  
            k = father;  
        }  
    }  
  
    public void heapify(int[] A) {  
        for (int i = 0; i < A.length; i++) {  
            siftup(A, i);  
        }  
    }  
}
```

Python版本：

```
import sys  
import collections  
class Solution:  
    # @param A: Given an integer array  
    # @return: void  
    def siftup(self, A, k):  
        while k != 0:  
            father = (k - 1) // 2  
            if A[k] > A[father]:  
                break  
            temp = A[k]  
            A[k] = A[father]  
            A[father] = temp  
  
            k = father  
    def heapify(self, A):  
        for i in range(len(A)):  
            self.siftup(A, i)
```

Python版本：

```
import sys
import collections
class Solution:
    # @param A: Given an integer array
    # @return: void
    def siftup(self, A, k):
        while k != 0:
            father = (k - 1) // 2
            if A[k] > A[father]:
                break
            temp = A[k]
            A[k] = A[father]
            A[father] = temp

            k = father
    def heapify(self, A):
        for i in range(len(A)):
            self.siftup(A, i)
```

算法思路：

1. 对于每个元素 $A[i]$ ，比较 $A[i]$ 和它的父亲结点的大小，如果小于父亲结点，则与父亲结点交换。
2. 交换后再和新的父亲比较，重复上述操作，直至该点的值大于父亲。

时间复杂度分析

1. 对于每个元素都要遍历一遍，这部分是 $O(n)$ 。
2. 每处理一个元素时，最多需要向根部方向交换 $\log n$ 次。

因此总的时间复杂度是 $O(n \log n)$

基于 Siftdown 的版本 $O(n)$

Java版本：

```
public class Solution {
    /**
     * @param A: Given an integer array
     * @return: void
     */
    private void siftdown(int[] A, int k) {
        while (k * 2 + 1 < A.length) {
            int son = k * 2 + 1; // A[i] 的左儿子下标。
            if (k * 2 + 2 < A.length && A[son] > A[k * 2 + 2])
                son = k * 2 + 2; // 选择两个儿子中较小的。
            if (A[son] >= A[k])
                break;

            int temp = A[son];
            A[son] = A[k];
            A[k] = temp;
            k = son;
        }
    }

    public void heapify(int[] A) {
        for (int i = (A.length - 1) / 2; i >= 0; i--) {
            siftdown(A, i);
        }
    }
}
```

Python版本：

```
import sys
import collections
class Solution:
    # @param A: Given an integer array
    # @return: void
    def siftdown(self, A, k):
        while k * 2 + 1 < len(A):
            son = k * 2 + 1      #A[i]左儿子的下标
            if k * 2 + 2 < len(A) and A[son] > A[k * 2 + 2]:
                son = k * 2 + 2    #选择两个儿子中较小的一个
            if A[son] >= A[k]:
                break

            temp = A[son]
            A[son] = A[k]
            A[k] = temp
            k = son

    def heapify(self, A):
        for i in range(len(A) - 1, -1, -1):
            self.siftdown(A, i)
```

算法思路：

1. 初始选择最接近叶子的一个父结点，与其两个儿子中较小的一个比较，若大于儿子，则与儿子交换。
2. 交换后再与新的儿子比较并交换，直至没有儿子。
3. 再选择较浅深度的父亲结点，重复上述步骤。

时间复杂度分析

这个版本的算法，乍一看也是 $O(n \log n)$ ，但是我们仔细分析一下，算法从第 $n/2$ 个数开始，倒过来进行 `siftdown`。也就是说，相当于从 `heap` 的倒数第二层开始进行 `siftdown` 操作，倒数第二层的节点大约有 $n/4$ 个，这 $n/4$ 个数，最多 `siftdown` 1次就到底了，所以这一层的时间复杂度耗费是 $O(n/4)$ ，然后倒数第三层差不多 $n/8$ 个点，最多 `siftdown` 2次就到底了。所以这里的耗费是 $O(n/8 * 2)$ ，倒数第4层是 $O(n/16 * 3)$ ，倒数第5层是 $O(n/32 * 4)$... 因此累加所有的时间复杂度耗费为：

$$T(n) = O(n/4) + O(n/8 * 2) + O(n/16 * 3) \dots$$

然后我们用 $2T - T$ 得到：

$$\begin{aligned} 2 * T(n) &= O(n/2) + O(n/4 * 2) + O(n/8 * 3) + O(n/16 * 4) \dots \\ T(n) &= O(n/4) + O(n/8 * 2) + O(n/16 * 3) \dots \\ 2 * T(n) - T(n) &= O(n/2) + O(n/4) + O(n/8) + \dots \\ &= O(n/2 + n/4 + n/8 + \dots) \\ &= O(n) \end{aligned}$$

因此得到 $T(n) = 2 * T(n) - T(n) = O(n)$

红黑树 Red-black Tree

红黑树（Red-black Tree）是一种平衡排序二叉树（Balanced Binary Search Tree），在它上面进行增删查改的平均时间复杂度都是 $O(\log n)$ ，是我们在日常生活中旅行的常备数据结构。

Q: 在面试中考不考呢？

A: 很少考……

Q: 需不需要了解呢？

A: 需要！

Q: 了解到什么程度呢？

A: 知道它是 Balanced Binary Search Tree，知道它支持什么样的操作，会用就行。不需要知道具体的实现原理。

红黑树的几个常用操作

Java当中，红黑树主要是 `TreeSet`，位于 `java.util.TreeSet`，继承自 `java.util.AbstractSet`，它的主要方法有：

- `add`，插入一个元素。
- `remove`，删除一个元素。
- `clear`，删除所有元素。
- `contains`，查找是否包含某元素。
- `isEmpty`，是否空树。
- `size`，返回元素个数。
- `iterator`，返回迭代器。
- `clone`，对整棵树进行浅拷贝，即不拷贝元素本身。
- `first`，返回最前元素。
- `last`，返回最末元素。
- `floor`，返回不大于给定元素的最大元素。
- `ceiling`，返回不小于给定元素的最小元素。
- `pollFirst`，删除并返回首元素。
- `pollLast`，删除并返回末元素。

更具体的细节，请参考 [Java Reference](#)。

此外，在Java当中，有一种map，用红黑树实现key查找，这种结构叫做 `TreeMap`。如果你需要一种map，并且它的key是有序的，那么强烈推荐 `TreeMap`。

在C++当中，红黑树即是默认的 `set` 和 `map`，其元素也是有序的。

而通过哈希表实现的则是分别是 `unordered_set` 和 `unordered_map`，注意这两种结构是在 `C++11` 才有的。

在Python当中，默认的set和dict是用哈希表实现，没有默认的红黑树。如果你想使用红黑树的话，可以使用 `rbtree` 这个模块，下载地址：<https://pypi.python.org/pypi/rbtree/0.9.0>

Merge K Sorted Lists 多路归并算法的三种实现方式

第九章 数据结构：区间、数组、矩阵和树状数组

本章节我们会讲一些关联度比较大的，和区间（Interval），数组（Array），矩阵（Matrix）有关的面试算法题。

请先修如下知识点：

- 如何用 Comparator 对区间进行排序
- 在排好序的区间序列中插入一个新区间
- 快速选择算法 Quick Select（返回第三章复习）
- K 路归并算法（K-way Merge Algorithm）
- 子数组与前缀和（Subarray & Prefix Sum）
- 位运算操作

子数组与前缀和

两个排序数组的中位数

题目描述

在两个排序数组中，求他们合并到一起之后的中位数

时间复杂度要求： $O(\log(n + m))$ ，其中 n, m 分别为两个数组的长度

LintCode 练习地址：

<http://www.lintcode.com/problem/median-of-two-sorted-arrays/>

解法

这个题有三种做法：

1. 基于 FindKth 的算法。整体思想类似于 median of unsorted array 可以用 find kth from unsorted array 的解题思路。
2. 基于中点比较的算法。一头一尾各自丢掉一些，去掉一半的时候，整个问题的形式不变。可以推广到 median of k sorted arrays.
3. 基于二分的方法。二分 median 的值，然后再用二分法看一下两个数组里有多少个数小于这个二分出来的值。

■ 基于中点比较的算法

两个排序的数组A和B分别含有m和n个数，找到两个排序数组的中位数，要求时间复杂度应为O(log (m+n))。

在线评测地址: <http://www.lintcode.com/problem/median-of-two-sorted-arrays/>

<https://www.jiuzhang.com/solution/median-of-two-sorted-arrays/>

■ 基于二分的算法

问题描述

求两个排好序的数组合并在一起之后，他们最中间的那个数是谁。如果总共有偶数个数，那么返回中间的两个数的平均值。

LintCode 练习地址：<http://www.lintcode.com/problem/median-of-two-sorted-arrays/>

算法描述

1. 我们需要先确定二分的上下界限，由于两个数组 `A, B` 均有序，所以下界为 `min(A[0], B[0])`，上界为 `max(A[A.length - 1], B[B.length - 1])`。
2. 判断当前上下界限下的 `mid(mid = (start + end) / 2)` 是否为我们需要的答案；这里我们可以分别对两个数组进行二分来找到两个数组中小于等于当前 `mid` 的数的个数 `cnt1` 与 `cnt2`，`sum = cnt1 + cnt2` 即为 `A` 跟 `B` 合并后小于等于当前 `mid` 的数的个数。
3. 如果 `sum < k`，即中位数肯定不是 `mid`，应该大于 `mid`，更新 `start` 为 `mid`，否则更新 `end` 为 `mid`，之后再重复第二步
4. 当不满足 `start + 1 < end` 这个条件退出二分循环时，再分别判断一下 `start` 跟 `end`，最终返回符合要求的那个数即可

算法详解

如果对该算法有点疑问，我们下面来详细讲解一下：

- 这一题如果用二分法来做，其实就是一个二分答案的过程
- 首先我们已经得到了上下界限，那么答案必定是在这个上下界限中的，需要实现的就是从这个歌上下界限中找出答案
- 我们每次取的 `mid`，其实就是我们每次在假设答案为 `mid`，二分的过程就是不断的推翻这个假设，然后再假设新的答案
- 需要满足的条件为：
 - 上面算法描述中的 `sum` 需要等于 `k`，这里的 `k = (A.length + B.length) / 2`。如果 `sum < k`，很明显当前的 `mid` 偏小，需要增大，否则就说明当前的 `mid` 偏大，需要缩小。
- 最终在 `start` 与 `end` 相邻的时候退出循环，判断 `start` 跟 `end` 哪个符合条件即可得到最终结果

■ 基于 FindKth 的算法

如何写 Comparator 来对区间进行排序？

对于数组、链表、堆等结构，标准库中排序方法，往往是对于基本类型的升序排序，有的时候，不一定能满足我们的要求。例如我们有一些特殊的顺序要求，或待排序的对象类型不是基本类型。此时，就需用到自定义排序。自定义排序可以用在很多地方，比如数组排序，堆的排序规则等。

一、Java实现自定义排序

Java实现自定义排序，主要有两种方法：

1.实现Comparable接口：

以 Interval 区间为例，在定义该类时，让其实现 Comparable，并重写其中的 compareTo 方法，使得 Interval 类可以进行大小比较，这样也可实现自定义的排序：

```
class Interval implements Comparable<Interval> {
    int left, right;
    Interval(int left, int right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int compareTo(Interval o) {
        return this.left - o.left;
    }
}
```

这样，在其他地方就可以直接对 Interval 对象的大小进行比较。完整的测试方法如下：

```
import java.util.ArrayList;
import java.util.List;
import static java.util.Collections.sort;

// Interval类如上

public class Main {
    public static void main(String[] args) {
        List<Interval> A = new ArrayList<>();
        A.add(new Interval(1, 7));
        A.add(new Interval(5, 6));
        A.add(new Interval(3, 4));
        System.out.println("Before sort:");
        for (Interval i : A)
            System.out.println("(" + i.left + ", " + i.right + ")");

        sort(A);
        System.out.println("After sort:");
        for (Interval i : A)
            System.out.println("(" + i.left + ", " + i.right + ")");
    }
}
```

2. 定义比较类：

该方法定义一个新的比较类，使其继承自Comparator，并完善其中的compare()方法。在调用时，使用该比较类进行比较。仍以 Interval 为例：

```
class Interval { // 注意，这里没有继承自Comparable
    int left, right;
    Interval(int left, int right) {
        this.left = left;
        this.right = right;
    }
}

class MyCmp implements Comparator<Interval> {
    @Override
    public int compare(Interval o1, Interval o2) {
        return o1.left - o2.left;
    }
}
```

要对 `List<Interval> A` 进行排序时，调用如下方法，填入一个比较类的对象即可：

```
A.sort(new MyCmp());
```

大家可以对以上各种方法，进行输出调试，也可以修改比较方法，做各种尝试。

二、Python实现自定义排序

Python中也有类似的两种实现方法：

1. 实现 `__lt__` 方法：

以 Interval 区间为例，在定义该类时，重写其中的 `__lt__` 方法，使得Interval类可以进行大小比较，这样也可实现自定义的排序：

```
class Interval:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    # 以下为重写的__lt__方法
    def __lt__(self, other):
        # 当两个Interval比较大小时，直接比较它们的left属性
        return self.left < other.left
```

这样，在其他地方就可以直接对Interval对象的大小进行比较。完整的测试方法如下：

```
# Interval类如上

if __name__ == "__main__":
    A = []
    A.append(Interval(1, 7))
    A.append(Interval(5, 6))
    A.append(Interval(3, 4))
    print("Before sort:")
    for i in A:
        print("({}, {})".format(i.left, i.right))

    # 由于定义了__lt__，方法，此处可以直接调用sort方法进行升序排序
    A.sort()

    print("After sort:")
    for i in A:
        print("({}, {})".format(i.left, i.right))
```

输出可以自行观察，不再赘述。

2. 定义key函数

可以给sort方法传入一个key函数，表示按照什么标准来对元素进行排序，仍以上面的例子为例：

```
# 要传给sort函数的key方法，表示按照interval.left进行排序
def IntervalKey(interval):
    return interval.left

A = []
A.append(Interval(1, 7))
A.append(Interval(5, 6))
A.append(Interval(3, 4))
A.sort(key=IntervalKey)
```

如果要实现多关键字排序怎么办（比如left小的排前面，如果left相等的话，那么right小的排前面）？

答案是返回一个tuple作为key。在Python中，两个tuple比较大小时会先比较第一个元素，返回第一个元素较小的那个，如果第一个元素相等，再比较第二个元素，返回较小的那个，以此类推……

我们可以利用这一点来实现多关键字排序，具体可以参考下面的例子：

```
class Interval:
    def __init__(self, left, right):
        self.left, self.right = left, right

    # 打印函数，用于直接print一个Interval对象
    def __repr__(self):
        return "Interval(%d, %d)" % (self.left, self.right)

data = [(3, 2), (3, 1), (2, 7), (1, 5), (2, 6), (1, 7)]
intervals = [Interval(left, right) for left, right in data]

print(sorted(intervals, key=lambda i: (i.left, i.right))) # 先按x从小到大排，再按y从小到大排
# 结果: [Interval(1, 5), Interval(1, 7), Interval(2, 6), Interval(2, 7), Interval(3, 1), Interval(3, 2)]

print(sorted(intervals, key=lambda i: (-i.left, i.right))) # 先按x从大到小排，再按y从小到大排
# 结果: [Interval(3, 1), Interval(3, 2), Interval(2, 6), Interval(2, 7), Interval(1, 5), Interval(1, 7)]

print(sorted(intervals, key=lambda i: (i.right, i.left))) # 先按y从小到大排，再按x从小到大排
# 结果: [Interval(3, 1), Interval(3, 2), Interval(1, 5), Interval(2, 6), Interval(1, 7), Interval(2, 7)]

print(sorted(intervals, key=lambda i: (-i.right, i.left))) # 先按y从大到小排，再按x从小到大排
# 结果: [Interval(1, 7), Interval(2, 7), Interval(2, 6), Interval(1, 5), Interval(3, 2), Interval(3, 1)]
```

在排好序的区间序列中插入新区间

问题描述

给一个排好序的区间序列，插入一段新区间。求插入之后的区间序列。要求输出的区间序列是没有重叠的。

LintCode 练习地址：<http://www.lintcode.com/problem/insert-interval/>

算法描述

1. 将该新区间按照**左端值**插入原区间中，使得原区间**左端值**是有序的。
2. 遍历原区间列表，并把它复制到一个新的 `answer` 区间列表当中，`answer` 是最后要返回的结果。
3. 遍历时，要记录上一次访问的区间 `last`。若当前区间**左端值**小于等于 `last` 区间的**右端值**，说明这两区间有重叠，此时仅更新 `last` 的**右端值**为这两区间**右端值**较大者；若当前区间**左端值**大于 `last` 的**右端值**，则可以直接加入 `answer`。
4. 返回 `answer`。

F.A.Q

Q：第三步有什么意义？

A：插入新区间后的原区间列表，仅能保证左端是有序的。而区间中是否存在重叠，右端是否有序，这些都是未知的。

Q：时空复杂度多少？

A：都是 $O(N)$ 。

Q：有没有更高效的做法？

A：有！在查找左端新区见待插位置时，可以采用二分查找。原算法的的第三步，实际上是在查找右端的位置，也可以用二分查找，这样两次查找的复杂度都降为了 $O(\log N)$ 。但是，完全没必要，因为这个算法涉及到数组中间位置的移动，所以 $O(N)$ 的时间复杂度是逃不开的，二分查找的改进对效率提升不明显，而且会增大编码难度。有兴趣的同学可以自己尝试~

参考代码

Java版本：

```
public class Solution {  
    /*  
     * @param intervals: Sorted interval list.  
     * @param newInterval: new interval.  
     * @return: A new interval list.  
     */  
    public List<Interval> insert(List<Interval> intervals, Interval newInterval) {  
        List<Interval> answer = new ArrayList<>();  
  
        int index = 0;  
        while (index < intervals.size() && intervals.get(index).start < newInterval.start) {  
            index++;  
        }  
        intervals.add(index, newInterval);  
  
        Interval last = null;  
        for (Interval item : intervals) {  
            if (last == null || last.end < item.start) {  
                answer.add(item);  
                last = item;  
            } else {  
                last.end = Math.max(last.end, item.end); // Modify the element already in list  
            }  
        }  
        return answer;  
    }  
}
```

Python版本：

```
class Solution:
    # @param intervals: Sorted interval list.
    # @param newInterval: new interval.
    # @return: A new interval list.
    def insert(self, intervals, new_interval):
        answer = []

        index = 0
        while index < len(intervals) and intervals[index].start < new_interval.start:
            index += 1
            intervals.insert(index, new_interval)

        last = None
        for item in intervals:
            if last == None or last.end < item.start:
                answer.append(item)
                last = item
            else:
                last.end = max(last.end, item.end)
        return answer
```

相关练习

<http://www.lintcode.com/problem/intersection-of-arrays/>

<http://www.lintcode.com/problem/merge-intervals/>

外排序与K路归并算法

介绍

外排序算法（External Sorting），是指在内存不够的情况下，如何对存储在一个或者多个大文件中的数据进行排序的算法。外排序算法通常是解决一些大数据处理问题的第一个步骤，或者是面试官所会考察的算法基本功。外排序算法是海量数据处理算法中十分重要的一块。

在学习这类大数据算法时，经常要考虑到内存、缓存、准确度等因素，这和我们之前见到的算法都略有差别。

基本步骤

外排序算法分为两个基本步骤：

1. 将大文件切分为若干个个小文件，并分别使用内存排好序
2. 使用K路归并算法（k-way merge）将若干个排好序的小文件合并到一个大文件中

第一步：文件拆分

根据内存的大小，尽可能多的分批次的将数据 Load 到内存中，并使用系统自带的内存排序函数（或者自己写个快速排序算法），将其排好序，并输出到一个个小文件中。比如一个文件有1T，内存有1G，那么我们就这个大文件中的内容按照 1G 的大小，分批次的导入内存，排序之后输出得到 1024 个 1G 的小文件。

第二步：K路归并算法

K路归并算法使用的是数据结构堆（Heap）来完成的，使用 Java 或者 C++ 的同学可以直接用语言自带的 PriorityQueue（C++中叫priority_queue）来代替。

我们将 K 个文件中的第一个元素加入到堆里，假设数据是从小到大排序的话，那么这个堆是一个最小堆（Min Heap）。每次从堆中选出最小的元素，输出到目标结果文件中，然后如果这个元素来自第 x 个文件，则从第 x 个文件中继续读入一个新的数进来放到堆里，并重复上述操作，直到所有元素都被输出到目标结果文件中。

Follow up: 一个个从文件中读入数据，一个个输出到目标文件中操作很慢，如何优化？

如果我们每个文件只读入1个元素并放入堆里的话，总共只用到了 1024 个元素，这很小，没有充分的利用好内存。另外，单个读入和单个输出的方式也不是磁盘的高效使用方式。因此我们可以为输入和输出都分别加入一个缓冲（Buffer）。假如一个元素有10个字节大小的话，1024 个元素一共 10K，1G 的内存可以支持约 100K 组这样的数据，那么我们就为每个文件设置一个 100K 大小的 Buffer，每次需要从某个文件中读数据，都将这个 Buffer 装满。当然 Buffer 中的数据都用完的时候，再批量的从文件中读入。输出同理，设置一个 Buffer 来避免单个输出带来的效率缓慢。

相关练习

[Lintcode相关练习](#)

[合并K个有序数组](#)

[合并K个有序链表](#)

面试相关问题

[面试题：合并 K 个排好序的大文件](#)

[面试题：求两个超大文件中 URLs 的交集](#)

[更多海量数据算法相关知识可参见](#)

[九章算法——海量数据处理算法与面试题全集](#)

简单位运算操作

什么是位运算

程序中所有数在内存中都以二进制形式储存。位运算（bit operation）就是直接对整数在内存中的二进制位进行操作。使用的主要目的是节约内存，加速运行，以及对内存要求苛刻时使用。

位运算在面试中的“初衷”是考察面试者的基本功，但不幸，位运算所考察的，大部分是知道就知道，不知道不知道。

按位与操作

主要讲解“按位与”（and）操作，操作符为 `&`。

将A和B的二进制表示的每一位进行与操作，只有两个对应的二进制位都为1时，结果位才为1，否则为0。

```
1 & 1 = 1  
1 & 0 = 0  
0 & 1 = 0  
0 & 0 = 0
```

例如：

下面 $(x)_y$ 表示 x 是 y 进制。

$A = (10)_{10} = (001010)_2$ （注意高位全是0）

$B = (44)_{10} = (101100)_2$

$A \& B = 10 \& 44 = 001010 \& 101100 = (001000)_2 = (8)_{10}$

具体的Java程序如下：

```
int a = 10 & 44; // a的值是8
```

Python版本类似：

```
a = 10 & 44
```

按位与相关题目

计算一个32位整数的二进制表示中有多少个 1 (<http://www.lintcode.com/zh-cn/problem/count-1-in-binary/>)。

例如 32 (1000000)，返回 1 ； 5 (101)，返回 2 ； 1023 (1111111111)，返回 10 。

算法思路：

用1不断左移（左移操作可参见下文其他操作，每次和num做按位与看是否为0，不是0的话说明这一位是1。左移32次后停止。代码如下：

Java:

```
public class Solution {  
    public int countOnes(int num) {  
        int count = 0;  
        for(int i = 0 ; i < 32; i++) {  
            if((num & (1<<i)) != 0) {  
                count++;  
            }  
        }  
        return count;  
    }  
}
```

Python:

```
class Solution:  
    def countOnes(self, num):  
        count = 0  
        for i in range(32):  
            if (num & (1 << i)) != 0:  
                count += 1  
        return count
```

Q：这方法有啥问题没有？

A：几乎没有，但是不管这个数是几，总要循环32次，稍微有点费时，而且看上去很蠢笨。

更精妙的算法

不断用num和num-1做按位与，结果直接赋给num。只要num不为0，就重复该过程。最后返回以上过程的次数即可。代码如下：

Java:

```
public class Solution {  
    public int countOnes(int num) {  
        int count = 0;  
        while (num != 0) {  
            num &= num - 1;  
            count++;  
        }  
        return count;  
    }  
}
```

Python:

```
class Solution:
    def countOnes(self, num):
        if num < 0:
            # Python的整数是无限长的, -1在Java/C++的32位整数中为: 11...11111 (32个1)
            # 但是在Python中为: ...1111111111111111 (无限个1)
            # 因此在遇到负数时要先截断为32位
            num &= (1 << 32)-1
        count = 0
        while num != 0:
            num &= num - 1
            count += 1
        return count
```

Q：这为啥可以？

A：其实原理很简单，先说结论：**每一次 `num &= num - 1` 会使得 `num` 最低位 1 变为 0。**

例如12，二进制表示为 `1100`，减1后的二进制表示为 `1011`。注意到了吗，减1后，最低位1变成了0，而最低位1后面的0全变成了1，高位不变。这样和原数按位与后，就只有最低位1发生了变化。所以该过程循环了多少次，就说明抹掉了多少个1。这对于其余正整数也是适用的。

但是要注意的是，Python中的整数是无限长的，负数的二进制表示中会有无限个前导1，因此要先将负数截断至32位。

位运算其他操作

其他的位运算操作同样重要，可参考[九章算法——位运算入门教程](#)自行学习，这里附上各自链接：

- 左移操作 `A << B`
- 右移操作 `A >> B`, `A >>> B`
- 按位或操作 `A | B`
- 按位非操作 `~A`
- 按位异或操作 `A^B`

■ 线段树 Segment Tree