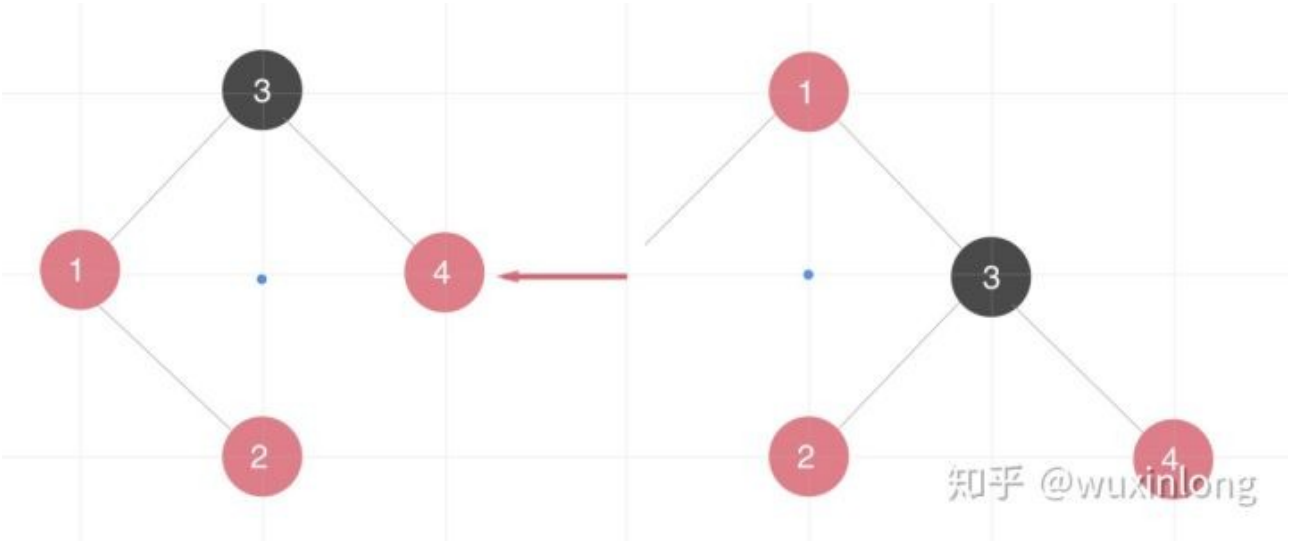
红黑树：二叉平衡树（同时也是二叉搜索树）

性质
1 根节点黑　2 不能有两个连着红　3　从根开始，任意路径黑节点数目相同：
保证树高永远是 O(log2n)

rotation 操作：

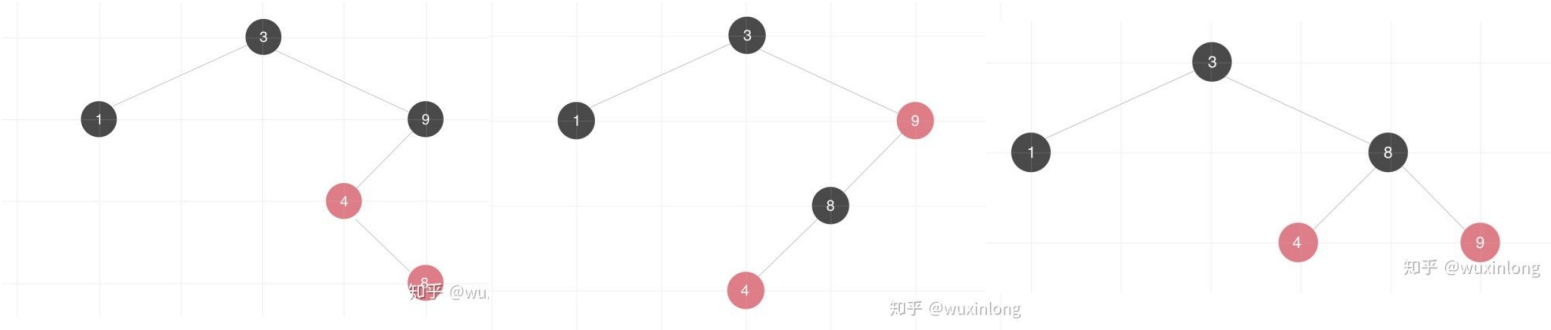
1 左旋



右孩子 Y 上去了，
X 成为 Y 新的左孩子
但 Y 旧的左孩子换到 X 处，成为 X 的右孩子

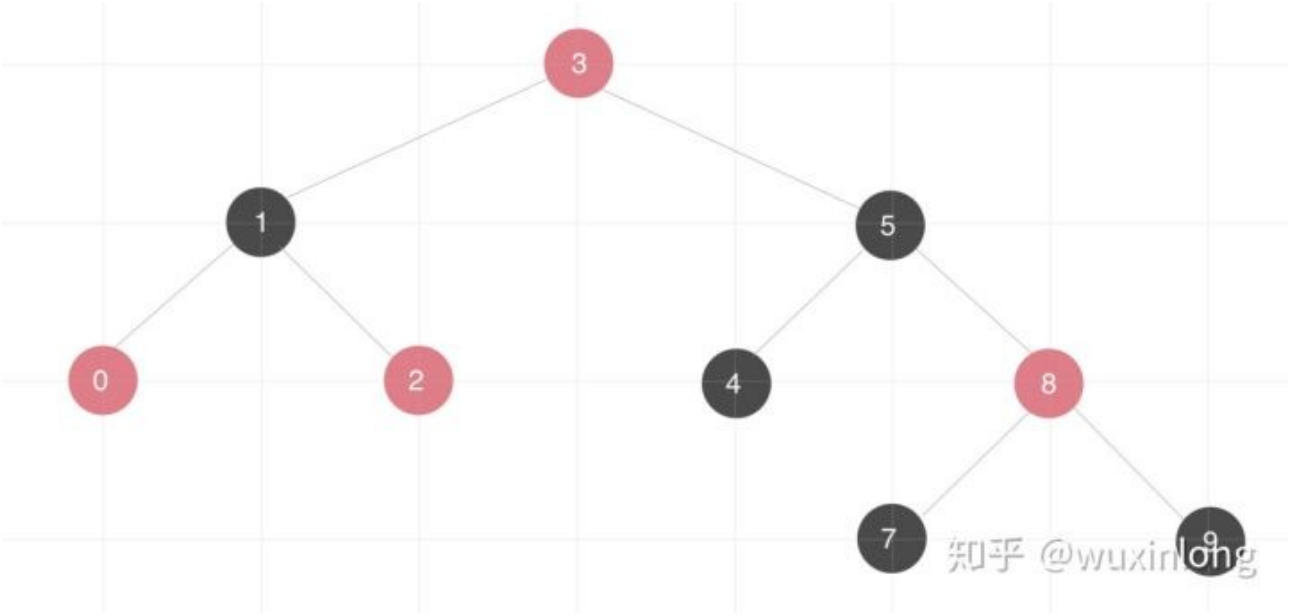
同理，对右旋：
左孩子 Y 上去了，
X 成为 Y 新的右孩子
但 Y 旧的右孩子换到 X 处，成为 X 的左孩子

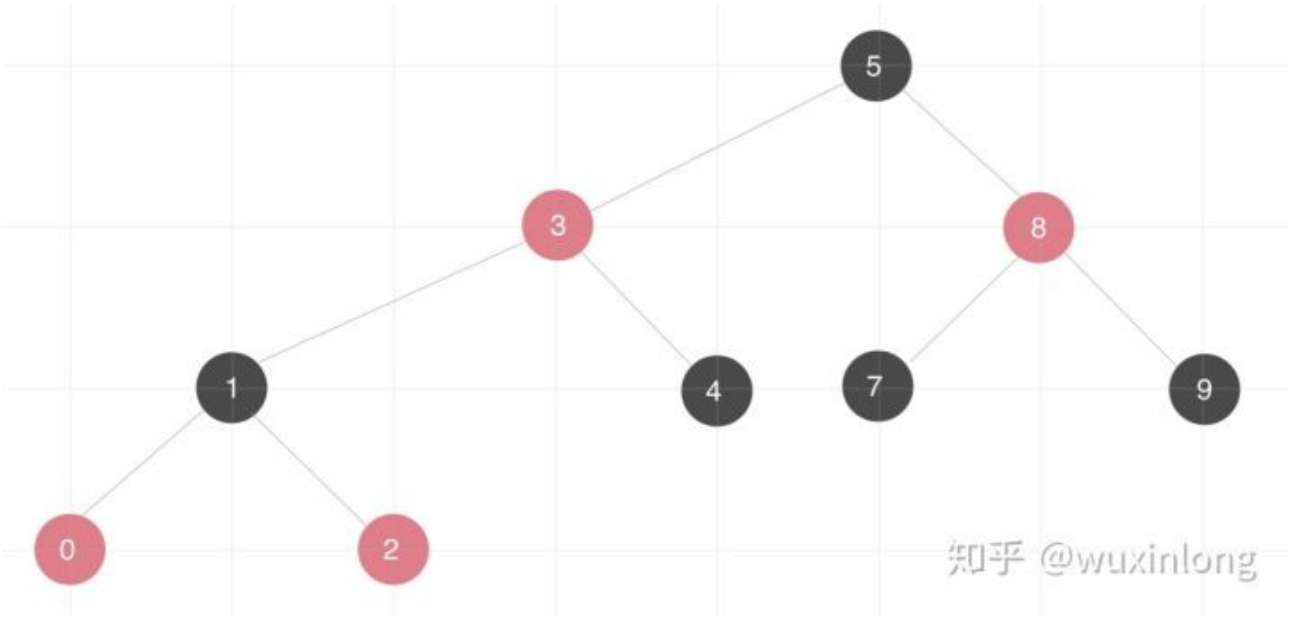
仍然保持二叉搜索树的性质 （左< 中<右）
左旋 4 （Y 没有左孩子　recolor 4,8 ,9）+ 右旋 9（Y 没有右孩子）



左旋 3 （Y 有左孩子 4）

左旋之前，右边黑色 node 多 。
原来 root 红。 左旋了一个黑色成为新 root,，左边黑色 node 增加到同右边





代码：
https://zhuanlan.zhihu.com/p/56890954?utm_source=wechat_session&utm_medium=social&utm_oi=53635367043072

插入+删除：（recoloer+rotation）
https://zhuanlan.zhihu.com/p/57041683?utm_source=wechat_session&utm_medium=social&utm_oi=53635367043072

（一般二叉搜索树，插入都是二分查找，插到第一个空位置。 删除是找到要删除的位置，根据有无孩子分情况。 无孩子直接删除或者只有一边有孩子，直接顶上。 两边都有孩子，找他的前驱（左子树里）到该位置，删掉他前驱）

2 单链表的快排：

原始快排：都是先选择第一个元素做 pivot.

从最后一个元素 j 开始向前找，找到严格小的扔到前边。（和 i 交换）。

从首元素 i 元素开始向前找，找到严格大的，扔到后边。（和 j 交换）

直到最后，i==j。这个时候 i 不比 pivot 小，是分界线,另 a[i]=pivot

返回分界线 i

链表中没法从后向前找，只能用**两个指针**。对于｛A｝pivot{B}

一个指针 i 指向 A 的最后一个元素。。i 和 i 之前的元素都比 pivot 小,或者等于。i 之后的元素比 pivot 大

另一个指针 j 用来遍历。找到比 pivot 大的元素，跳过继续找，所以所有 j 之前,i 之后的，都比 pivot 大。

当 j 碰上比 pivot 小的元素，可以把该元素和 i 之后的元素交换，i++. 使得该元素成为 i 之前的元素。而原来 i 之后，j 之前的第一个元素(比 pivot 大)，被交换到位置 j。保证 i-j 之间仍然比 j 大。

初始：

3 4 1 2 5

pivot

i j

a[j]<pivot,交换 a[i+1]=4,a[j]=1,i++,j++ (先执行 i++,直接交换 a[i],a[j]即可)

3 4 1 2 5 → 3 1 4 2 5 → 3 1 4 2 5

pivot pivot pivot

i j i j i j

a[j]<pivot,交换 a[i+1]=4,a[j]=2,i++,j++

3 1 4 2 5 → 3 1 2 4 5

pivot pivot

i j i j

最后交换 a[i],a[0]. 返回分隔位置 i

3 1 2 4 5 → 2 1 3 4 5

pivot pivot

i j i j

codes:

```
i=0
j=1
pivot=a[0]
while j<len(a):
    if a[j]<pivot:
        i+=1
        a[i],a[j]=a[j],a[i]
    j+=1
a[0],a[i]=a[i],a[0]
print(a)
print(i)
```

3 n-sum

1 第一种（剑指 offer）

任意给定 s, 求 1,...s-1 中和为 s 的全部连续序列。 如给定 3,输出[[1,2]],给定 15,输出[1,2,3,4,5],[7,8]

双指针（相当于排好序）。i 是序列头，j 是序列尾。i-j 小，j++; i-j 大，i++ 直到 i 为 s//2

2 任意正数数组，求满足 sum>=s 的长度最小**连续子数组**（的长度） O(n) (leetcode 209)

同样双指针。[i,j]是子数组的 start,end

[i,j]<s, j++,更新 sum

[i,j]>=s 得到长度。 ~:看新区间 i++,更新 sum [i+1,j]如果大于 s, 重复~,直到 i==j 或者 sum<s

或者新区间[i+1,j]小于 s.

([i+1,j]大于 s 时不 j--,因为[i,j-1]<s　i 固定的已经算过)

此时新区间[i,j]<s 继续 j++

i,j 分别只遍历 n 次　->O(n)

code:

```
i=0
j=1
current=nums[i]
min_l=1 if current>=s else 9999999999
while j<len(nums):
   current+=nums[j]                # 起始是 i. 每次算新的[i,j]
   while current>=s and i<=j:      # 只要区间>=s
      min_l=min(min_l,j-i+1)       # 算新区间 i+=1，新区间要是还大，继续算新区间。
                                   #直到区间 sum 小于 s /或者 i<=j
      current-=nums[i]             #新区间 sum
      i+=1                         #  如果 i==j, 意味着 a[j]>s,  新区间从 i=j+1 开始  current==0
   # 直到新区间[i,j]<s. 算下一个 j
   j+=1
if min_l==9999999999:
   return 0
else:
   return min_l
```

3　任意正数数组，求 sum=s 的所有**连续**子数组(或者最大长度) O(n)　(leetcode 560)

首先将 s[i],s[j]存起来。其中 s[i]是 s[0]-s[i]的和　s[0]:a[0]　s[1]:a[0]+a[1]

连续子数组[i,j]的和是 s[j]-s[i]+a[i]

3.1 只判断有没有：

i=0.j=n-1

同样双指针[i,j] 对于每个 i　如果[i,j]<s, j--

[i,j]<s,i++

==s,return True

3.2 输出所有子数组　s[j]-s[i]=s → s[j]-s=s[i]　　（有负数也成立）

对于当前的每一个 j，如果其累积和 s[j]-s 在之前的累积和中。sum　[0, a[0],a[0]+a[1], …]

i=　-1　0

1　…

那么(i,j]就一定和为 s.　s[2]=a[0]+a[1]+a[2] 如果 s=a[1]+a[2]　i=0　区间为：[i+1,j] → [1,2]

如果到 0-j 包含和为 s 的连续子数组，s[j]-s 就一定存在. 数组长为 j-i

可以用 dict():　s[i]:i 保存所有的累积和，和对应的 idx. 初始为 sum=0:pos=-1

注意：**有可能多个 sum 值相同。都写到 sum 里 sum: (i1,i2,i3)**

code:

```
sum2pos=dict()
sum2pos[0]=[-1]      # 初始累积和为 0,位置为 pos==-1
count=0
```

```
    summ=0
    for j in range(len(nums)):
        summ+=nums[j]
        find=summ-k
        if find in sum2pos:
            pre_pos=sum2pos[find]          # i  -->实际区间[i+1,j]
            count+=len(sum2pos[find])   # 可能有多个 sum[i]值相同
        if summ not in sum2pos:
            sum2pos[summ]=[j]
        else:
            sum2pos[summ].append(j)
    return count
```

4　任意数组**不连续** n-sum combinational sum　NP-hard,指数级，用回溯

(leetcode 39,40,216,377)

#39 给的元素无重复，但元素可以用任意次　回溯

每次递归，还从当前元素开始，为了重复用　但也需要用 start 控制已经遍历过的节点，之后 2 2 3 不再遍历 2,所有的 2 已经遍历过。最多遍历 2 2 3 3, 2 2 3 4

直到当前元素结束了 2 2 2 2，num>remain,后边更不可能，返回上一节点 2 2 2,接着遍历 2 2 3

或者当前有一次满足了，也结束，2 2 3,后边的 2 2 4 同样不用看了，直接返回更上层节点。用 flag 控制

codes:

```
    candidates=sorted(candidates)
    remain=target
    path=[]
    result=[]
    def mydfs(remain,path,result,start):
        if remain==0:
            result.append(path)
            return False          #2 2 3 满足了，2 2 4 不试了

        # 回退到 2 2 3
        # 对于每一个元素，如果没有用完，可以一直用，看能不能拼出 remian
        flag=True
        for i in range(start,len(candidates)):
            num=candidates[i]
            # 2 2 2 2 不行　不试 2 2 2 3, 返回　直接回退到上一节点 2 2 2/ 2 2 3
            if num>remain:
                return True
            flag=mydfs(remain-num,path+[num],result,i)
            # start:保证 i 之后，2 2 3 只找 3,4, 不再回去找 2
            # 此时本来是 2 2 2. num==2  那就试 2 2 3
            # 如果可以，也不用试 2 2 4 了，直接再回退到上一个节点　2 2
            # 上一节点 2 2 相当于都试完了，之后试 2 3
            # 直接加到满足 2 2 3，不试 2 2 4, 返回上一节点 2 2/ 只有这个节点满足了，后边的才不试
            # 不满足的情况，前边已经返回了
            if flag==False:
                return True
            #都是 return True. 因为返回了，这个轮次不试后边的了，但是上一个轮次可以继续试
    mydfs(remain,[],result,0)
```

#40 给的元素有重复，但只能用 1 次

1 在所有元素中找 s, 相当于加上 nums[i]后，在剩下没被访问的元素中找 remian=s-num[i]

2 每个 idx,访问过后就不重复访问，剩余元素都是 i 之后的元素。递归 start 从 i+1 开始

3 事先排好序。 有序数组中先用小的凑。当碰到 remian 小于 num[i]时，剩下的更凑不出来

(排序后相邻的也挨着，更容易找到重复元素)

对[1,1,2,5,6]

4 允许[1,1,6], 避免[1,2,5],[1,2,5]: 如果是同一个 for 循环里的，都>=start,才跳过

code: 每次递归时，通过控制 start=i+1,控制剩余元素，避免访问之前的重复元素

```
# 避免重复用，用 idx 记录
# 回溯
candidates=sorted(candidates)  # 先用小的试
visited=set()
result=[]
remain=target
path=[]
def mydfs(start,path,remain,result):
    if remain==0:
        result.append(path)
        return
    # 每次递归，从 start 开始。之前的元素用过了，不再用
    for i in range(start,len(candidates)):
        num=candidates[i]
        # 如果有重复，比如[1,1,2,5], 算完了第一个 1 的所有序列[1,2,5]. 不再算第二个 1 的[1,2,5]
        # (同一批次 i>start)(但不在同一次递归里，如[1,1,6],允许)
        if i-1>=0 and i>start :
            if num==candidates[i-1]:
                continue
        # 在剩下的元素里找 remina-num
        if remain<num: # 如果剩下最小的元素也比 remian 大，break. 之后所有元素都无法组合出 s
            return
        # 在之后的元素 i+1 里找      如果此时元素 num 使得剩下的满足，该 path 加入 result/
start=i+1
        mydfs(i+1,path+[num],remain-num,result)
mydfs(0,path,remain,result)
return result
```

#216 固定正数数目 组成 n

377 给的元素无重复，但元素可以用任意次 求所有可能的组合总数。只求总数目

类似于 39,但不同的是次数所有排列都算，[1 1 2] ,[1,2,1] 各记一次。复杂度高

codes:

```
# 对任意一个 target i, 组合数目是用掉一个 c 后，剩下的 target-c 对应的组合数目
# 7   [2,3,5]
#   用一个 2 以后，所有 5 的组合数目   +   用一个 3 以后，所有 4 的组合数目   +   用一个 5 以后，
所有 2 的组合数目
#  [2,2,3],[2,5]                [3,2,2]                    [5,2]
# dp[i]= all  sum dp[i-c]  for all c
dp=[0]*(target+1)
# target 是 0 时，可能的组合数目是 1   比如对 i==2,dp[2]=dp[0]==1
dp[0]=1
for i in range(1,target+1):
```

```
            dp[i]=sum ([dp[i-c] for c in nums if i>=c])
        return dp[-1]
```

类似于零钱兑换的几种方式，,dp： 给定固定 target, candidate,看不同的硬币组合方式.
(相当于可重复用元素的 n-sum,求满足条件的 n-sum 对应的最少元素个数)
回溯可以做，但只求个数没必要，太耗时。
用 dp, targte 需要的最少元素，是 target-c 需要的最少元素+1

code:

```
    def coinChange(self, coins: List[int], amount: int) -> int:
        # 如果硬币中有 1 存在，那么就简单很多。依次贪心找，最后的都给 1
        # 但如果 1 不存在，直接找的话，如果最后剩下的余数>1,比如剩下 2 元，但最小硬币是 3。就需要调整前边的
        # 所以用动态规划。如果钱数 i 对应的最小硬币数是 dp[i],那么对任意一枚硬币 c  dp[i]=min(dp[i-c]+1) 所有 c
        # 金额是 0 时，不需要硬币   dp[0]=0
        # 初始化其他 dp[i]==99999999
        INT_MAX=999999999
        dp=[INT_MAX]*(amount+1)
        dp[0]=0
        for i in range(1,amount+1): # 计算每个 dp[i],直到 dp[amount],得到需要的最少硬币
            # 每个 i 用掉的最少硬币是去掉一个硬币后，剩下钱对应的最少硬币+1
            # 如果去掉任意一个硬币都兑换不了。那这个钱也兑不了,保持 INT_MAX
            # 不论是 dp[i-c]==INT_MAX,还是
            dpi=[dp[i-c]+1 for c in coins if c<=i and dp[i-c]!=INT_MAX]
            if len(dpi)!=0: # 没有一个硬币可兑换，保持
                dp[i]=min(dpi)

        if dp[-1]==INT_MAX:
            return -1
        else:
            return dp[amount]
```

连续子数组最大和，也可以 dp:
dp[i]:以元素 a[i]结尾的连续子数组最大和。（即必须包含 a[i]）
然后在所有 dp[i]里选最大的。
递推式：（不论 a[i]本身正负，必须包含 a[i]）
如果之前的连续子数组最大和 dp[i-1]是小于 0 的。不论 a[i]本身正负，最大的一定是 a[i]
如果 dp[i-1]大于 0, 那么到 i 的连续子数组最大和一定是 dp[i-1]+a[i]

code:

```
        max_num=-99999999   #存以第一时刻结尾...以第 i 时刻结尾的连续子数组，最大和
        cur=array[0]        #以第 i 时刻结尾的连续子数组的最大和
        for i in range(1,len(array)):
            if cur<0:
                cur=array[i]
            else:
                cur+=array[i]
            if cur>max_num:
                max_num=cur
        return max_num
```

４ 圆盘涂色问题　n 个扇形，m 种颜色

５ 10G 文件，2G 内存。　中位数/topK

6 强连通分量

7 全排列　　dfs.　　n!个全排列
每次有一个元素和首元素交换，求 a[0]固定时剩下元素的全排列。递归。直到 path=len(a).
该首元素的全排列结束，交换回来，换下一个首元素

```
l=len(ss)
result=[]
def dfs(s,path,result):
    if len(path)==l:
        result.append(path)
        return
    have_change=set()
    for i in range(len(s)):
        if s[i] not in have_change: #重复的，不交换．　前边的全排列已经有了
            # 以该次 path+s[0]为首的全排列
            s=s[i]+s[0:i]+s[i+1:]

            path+=s[0]
            have_change.add(s[0])
            if len(path)==l:
                result.append(path)
            else:
                dfs(s[1:],path,result) # s[0]固定，　之后的全排列
            path=path[:-1]
            s=s[i]+s[0:i]+s[i+1:]
dfs(ss,"",result)
```