一、栈
新添加的元素被最后移除（LIFO），基于在集合中的时间来排序的方式。

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

    def isEmpty(self):   #判断栈是否为空
        return self.items==[]

    def push(self,item):   #压栈
        self.items.append(item)

    def pop(self):   #出栈，并返回栈顶元素
        return self.items.pop()

    def peek(self):   #只返回栈顶元素，不出栈
        return self.items[len(self.items)-1]

    def size(self):   #返回栈的长度
        return len(self.items)
```

二、队列
先进先出（FIFO）

```
class Queue:
    def __init__(self):
        self.items=[]

    def isEmpty(self): #判断队列是否为空
        return self.items==[]

    def enqueue(self,item):   #在队尾插入元素
        self.items.insert(0,items)

    def dequeue(self):   #在队首的元素出队，并返回元素值
        return self.items.pop()

    def size(self):   #判断大小
        return len(self.items)
```

三、双端队列
融合了栈和队列的特点

```
class Deque:
    def __init__(self):
        self.items=[]

    def isEmpty(self):    #判断双端队列是否为空
        return self.items==[]

    def addFront(self,item):    #在队首添加元素
        self.items.append(item)

    def addRear(self,item):    #在队尾添加元素
        self.items.insert(0,item)

    def removeFront(self):    #移除并返回队首元素
        return self.items.pop()

    def removeRear(self):    #移除并返回队尾元素
        return self.items.pop(0)

    def size(self):    #判断大小
        return len(self.items)
```

四、实现无序列表：链表

1.Node 类

```
class Node:
    def __init__(self,initdata):
        self.data=initdata
        self.next=None

    def getData(self):    #获取当前节点中存储的数据值
        return self.data

    def getNext(self):    #获取当前节点指向的“下一个节点”
        return self.next

    def setData(self,newdata):    #更新当前节点保存的数据
        self.data=newdata

    def setNext(self,newnext):    #设置当前节点的 next 指针指向另一个节点
        self.next=newnext
```

2.UnorderedList 类

```
class UnorderedList:
    def __init__(self):
        self.head=None    #初始构造为空

    def isEmpty(self):    #头为空说明链表为空
        return self.head==None

    def add(self,item):    #添加元素至表头
        temp=Node(item)
        temp.setNext(self.head)
        self.head=temp

    def length(self):    #判断长度
        current=self.head
        count=0
        while current!=None:
            count=count+1
            current=current.getNext()
        return count

    def search(self,item):    #查找元素
        current=self.head
        found=False
        while current!=None and not found:
            if current.getData()==item:
                found=True
            else:
                current=current.getNext()
        return found

    def remove(self,item):    #移除元素
        current=self.head
        previous=None
        found=False
        while not found:
            if current.getData==item:
                found=True
            else:
                previous=current
                current=current.getNext()
        if previous==None:    #移除的头节点
            self.head=current.getNext()
        else:
```

```
                previous.setNext(current.getNext())
```

五、有序表

```
class OrderedList:
    def __init__(self):
        self.head=None

    #有序表可以简化搜索过程（提前终止）
    def search(self,item):
        current=self.head
        found=False
        stop=False
        while current !=None and not found and not stop:
            if current.getData()==item:
                found=True
            else:
                if current.getData()>item:
                    stop=True
                else:
                    current=current.getNext()
        return found

    def add(self,item):
        current=self.head
        previous=None
        stop=False
        while current!=None and not stop:
            if current.getData()>item:
                stop=True
            else:
                previous=current
                current=current.getNext()

            temp=Node(item)
            if previous==None:   #添加到开头的特殊情况
                temp.setNext(self.head)
                self.head=temp
            else:
                temp.setNext(current)
                previous.setNext(temp)
```

六、递归

找零问题的递归解决方案

```
def recMC(coinValueList,change):
    minCoins=change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c<=change]:
            numCoins=1+recMC(coinValueList,change-i)
            if numCoins<minCoins:
                minCoins=numCoins
    return minCoins
```

添加查询表

```
def recDC(coinValueList,change,knownResults):
    minCoins=change
    if change in coinValueList:
        knownResults[change]=1
        return 1
    elif knowResults[change]>0:
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c<=change]:
            numCoins=1+recDC(coinValueList,change-i,knownResults)
            if numCoins<minCoins:
                minCoins=numCoins
                knownResults[change]=minCoins
    return minCoins
```

```
def dpMakeChange(coinValueList,change,minCoins):
    for cents in range (change+1):
        coinCount=cents
        for j in [c for c in coinValueList if c<=cent]:
            if minCoins[cents-j]+1<coinCount:
                coinCount=minCoins[cents-j]+1
        minCoins[cents]=coinCount
    return minCoins[change]
```

七、搜索

①顺序搜索

```
def sequentialSearch(alist,item):
    pos=0
    found=False
```

```
    while pos<len(alist) and not found:
        if alist[pos]==item:
            found=True
        else:
            pos=pos+1
    return found
```

有序表的顺序搜索

```
def orderedSequentialSearch(alist,item):
    pos=0
    found=False
    stop=False
    while pos<len(alist) and not found and not stop:
        if alist[pos]==item:
            found=True
        else:
            if alist[pos]>item:
                stop=True
            else:
                pos=pos+1
        return found
```

②有序表的二分搜索

```
def binarySearch(alist,item):
    first=0
    last=len(alist)-1
    found=False

    while first<=last and not found:
        midpoint=(first+last)//2
        if alist[midpoint]==item:
            found=True
        else:
            if item<alist[midpoint]:
                last=midpoint-1
            else:
                first=midpoint+1
    return found
```

递归版本

```
def binarySearch(alist,item):
```

```
    if len(alist)==0:
        return False
    else:
        midpoint=len(alist)//2
        if alist[midpoint]==item:
            return True
        else:
            if item<alist[midpoint]:
                return binarySearch(alist[:midpoint],item)
            else:
                return binarySearch(alist[midpoint+1:],item)
```

八、散列

HashTable 类的构造

```
class HashTable:
    def __init__(self):
        self.size=11
        self.slots=[None]*self.size
        self.data=[None]*self.size

    def put(self,key,data):   #往映射中加入新的键-值对
        hashvalue=self.hashfunction(key,len(self.slots)) #计算槽位

        if self.slots[hashvalue]==None:
            self.slots[hashvalue]=key
            self.data[hashvalue]=data
        else:
            if self.slots[hashvalue]==key:
                self.data[hashvalue]=data   #替换
            else:   #冲突
                nextslot=self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot]!=None and self.slots[nextslot]!=key:
                    nextslot=self.rehash(nextslot,len(self.slot))
                    #寻找可用的槽位（空或者已有该键值）
                if self.slots[nextslot]==None:
                    self.slots[nextslot]=key
                    self.data[nextslot]=data
                else:
                    self.data[nextslot]=data

        def hashfunction(self,key,size):
            return key%size   #计算槽位的方法
```

```
        def rehash(self,oldhash,size):
            return (oldhash+1)%size    #冲突时计算新槽位

    def get(self,key):
        startslot=self.hashfunction(key,len(self.slots))
        data=None
        stop=False
        found=False
        position=startslot
        while self.slots[position]!=None and not found and not stop:
            if self.slots[position]==key:
                found=True
                data=self.data[position]
            else:
                position=self.rehash(position,len(self.slots))
                if position==startslot:
                    stop=True    #查找一圈没有
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)
```

九、排序

①冒泡排序

```
def bubbleSort(alist):
    for passnum in range (len(alist)-1,0,-1):
        for i in range (passnum):
            if alist[i]>alist[i+1]:
                temp=alist[i]
                alist[i]=alist[i+1]
                alist[i+1]=temp
```

优化：提前退出循环

```
def shortbubbleSort(alist):
    exchanges=True
    passnum=len(alist)-1
    while passnum>0 and exchanges:
        exchanges=False
```

```
    for i in range (passnum):
        if alist[i]>alist[i+1]:
            exchanges=True
            temp=alist[i]
            alist[i]=alist[i+1]
            alist[i+1]=temp
        passnum=passnum-1
```

②选择排序　每次选取最大的元素

```
def selectionSort(alist):
    for fillslot in range (len(alist)-1,0,-1):
        positionOfMax=0
        for location in range (1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax=location
        temp=alist[fillslot]
        alist[fillslot]=alist[positionOfMax]
        alist[positionOfMax]=temp
```

③插入排序

```
def insertionSort(alist):
    for index in range (1,len(alist)):
        currentvalue=alist[index]
        position=index
        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]   #右移，提供空位
            position=position-1
        alist[position]=currentvalue
```

④希尔排序

```
def shellSort(alist):
    sublistcount=len(alist)//2
    while sublistcount>0:
        for startposition in range (sublistcount):
            gapInsertionSort(alist,startposition,sublistcount)
            print('After increments of size',sublistcount,
                  'The list is',alist)
            sublistcount=sublistcount//2

def gapInsertionSort(alist,start,gap):
    for i in range (start+gap,len(alist),gap):
```

```
            currentvalue=alist[i]
            position=i
            while position>=gap and alist[position-gap]>currentvalue:
                alist[position]=alist[position-gap]
                position=position-gap
            alist[position]=currentvalue
```

⑤归并排序

采取递归的方法，降低了时间复杂度，但是可能会占用额外的空间。

```
def mergeSort(alist):
    print('Splitting',alist)
    if len(alist)>1:
        mid=len(alist)//2
        lefthalf=alist[:mid]
        righthalf=alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)   #递归

        #合并
        i=0
        j=0
        k=0
        #左右两边已经排好序了
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
        print('Merging',alist)
```

⑥快速排序
先为基准值找到正确的位置，再对两边分别归并排序。

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:
        splitpoint=partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)

def partition(alist,first,last):
    pivotvalue=alist[first]    #基准值
    leftmark=first+1
    rightmark=last
    done=False
    while not done:
        while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
            leftmark=leftmark+1
        while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
            rightmark=rightmark-1
        if rightmark<leftmark:
            done=True
        else:
            alist[leftmark],alist[rightmark]=alist[rightmark],alist[leftmark]
        temp=alist[first]
        alist[first]=alist[rightmark]
        alist[rightmark]=temp        #把基准值放到合适的位置上
    return rightmark
```

⑦桶排序
将整个数据区间划分成若干个“桶”，把每个元素放入对应的桶中，再分别对每个桶中的数据排序，
最后把所有桶合并成一个有序序列。

```
def bucket_sort(arr):
    n=len(arr)
    if n<=1:
        return arr
    buckets=[[]for i in range (n)] #创建 n 个桶

    #把每个数放入对应的桶中，这里假设数据范围为 0 到 1
```

```
    for num in arr:
        index=int(num*n)
        if index==n:
            index=index-1
        buckets[index].append(num)

    for bucket in buckets:   #对每个桶排序
        bucket.sort()

    result=[]
    for bucket in buckets:
        result.extend(bucket)
        #把列表中所有元素加入列表末尾

    return result
```

⑧基数排序

基数排序是一种非比较性排序算法

从个位开始，对所有数进行稳定排序；然后按照十位、百位等依次重复上述过程直到最高位，排序结束后，原数组就是有序的。

```
def radixSort(s,m,d,getKey): #s 为列表，m 表示基数，十进制 m=10
    #key(x,k)表示取元素 x 的第 k 位
    for k in range (d):   #d 为最高位
        buckets=[[] for j in range (m)]
        for x in s:
            buckets[key(x,k)].append(x)
        i=0
        for bkt in buckets:
            for e in bkt:
                s[i]=e
                i=i+1   #桶排序

def getKey(x,i): #取非负整数的第 i 位（以个位为 0 位）
    tmp=None
    for k in range (i+1):
        tmp=x%10
        x=x//10
    return tmp
```

⑨堆排序

用 Python 自带的堆来排序

```
import heapq
def heapSorted(iterable):
```

```
    h=[]
    for value in iterable:
        h.append(value)
    heapq.heapify(h)
    return [heapq.heappop(h) for i in range (len(h))]
```

十、树

①列表之列表实现（并非二叉树类）

```
def BinaryTree(r):
    return [r,[],[]]

def insertLeft(root,newBranch):
    t=root.pop(1)
    if len(t)>1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch,[],[]])
    return root
    #把新元素插在了根节点的左子节点位置，
    #原来的左子树的父节点就是新插入的元素

def insertRight(root,newBranch):
    t=root.pop(2)
    if len(t)>1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
    #与插入左子树同理

def getRootVal(root):
    return root[0]   #访问根节点

def setRootVal(root,newVal):
    root[0]=newVal   #修改根节点

def getLeftChild(root):
    return root[1]   #访问左子树

def getRightChild(root):
    return root[2]   #访问右子树
```

②节点与引用实现（定义类）

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key=rootObj    #是对每一个节点的定义
        self.leftChild=None
        self.rightChild=None

    def insertLeft(self,newNode):
        if self.leftChild==None:
            self.leftChild=BinaryTree(newNode)
        else:
            t=BinaryTree(newNode)
            t.left=self.leftChild
            self.leftChild=t
        #同理是把新元素插在了根节点的左子节点位置，
        #原来的左子树的父节点就是新插入的元素

    def insertRight(self,newNode):
        if self.rightChild==None:
            self.rightChild=BinaryTree(newNode)
        else:
            t=BinaryTree(newNode)
            t.right=self.rightChild
            self.rightChild=t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):    #改变考察节点的值
        self.key=obj

    def getRootVal(self):
        return self.key
```

应用：解析树的构建

```
import Stack
import BinaryTree

def buildParseTree(fpexp):
    fplist=fpexp.split()
```

```
    pStack=Stack()   #需要栈来记录父节点
    eTree=BinaryTree('')
    pStack.push(eTree)
    currentTree=eTree
    for i in fplist:
        if i=='(':
            currentTree.insertLeft('')   #左沉
            pStack.push(currentTree)   #压栈，当前节点是父节点
            currentTree=currentTree.getLeftChild()
        elif i not in '+-*/)':
            currentTree.setRootVal(eval(i))
            parent=pStack.pop()   #回到父节点
            currentTree=parent
        elif i in '+-*/':
            currentTree.setRootVal(eval(i))
            currentTree.insertRight('')   #右沉
            pStack.push(currentTree)
            currentTree=currentTree.getRightChild()
        elif i==')':
            currentTree=pStack.pop()
    return eTree
```

计算二叉解析树

```
def evaluate(parseTree):
    opers={'+':operator.add,'-':operator.sub,
           '*':operator.mul,'/':operator.truediv}
    leftC=parseTree.getLeftChild()
    rightC=parseTree.getRightChild()

    if leftC and rightC:   #当前节点为操作符
        fn=opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:   #当前节点为操作数
        return parseTree.getRootVal()
```

③树的遍历

3.1 前序遍历：先访问根节点，然后递归前序遍历左子树，最后递归前序遍历右子树。

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()
```

3.2 后序遍历：先递归后序遍历右子树，再递归后序遍历左子树，最后访问根节点。

```
def postorder(tree):
    if tree!=None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

3.3 中序遍历：先递归中序遍历左子树，再访问根节点，最后递归中序遍历右子树。

```
def inorder(tree):
    if tree!=None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

④二叉堆：时间复杂度为 O(logn)的排序方法。
父节点都小于等于其子节点

```
class BinaryHeap:
    def __init__(self):
        self.heapList=[0]
        self.currentSize=0

    #插入新元素
    def percUp(self,i):
        while i//2>0:
            if self.heapList[i]<self.heapList[i//2]:
                tmp=self.heapList[i//2]
                self.heapList[i//2]=self.heapList[i]
                self.heapList[i]=tmp
            i=i=//2

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize=self.currentSize+1
```

```
            self.percUp(self.currentSize)

    #删除最小元素
    def percDown(self,i):
        while i*2<=self.cuurrentSize:
            mc=self.minChild(i)
            if self.heapList[i]>self.heapList[mc]:
                tmp=self.heapList[i]
                self.heapList[i]=self.heapList[mc]
                self.heapList[mc]=tmp
            i=mc

    def minChild(self,i):
        if i*2+1>self.currentSize:
            return i*2
        else:
            if self.heapList[i*2]<self.heapList[i*2+1]:
                return i*2
            else:
                return i*2+1

    def delMin(self):    #删除并返回最小元素，但不改变堆的性质。
        retval=self.heapList[i]
        self.heapList[1]=self.heapList[self.currentSize]
        self.currentSize=self.currentSize-1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self,alist):    #通过列表构建堆
        i=len(alist)//2    #大于它的都是叶节点了，不用下沉
        self.currentSize=len(alist)
        self.heapList=[0]+alist[:]
        while i>0:
            self.percDown(i)
            i=i-1
```

⑤二叉搜索树

小于父节点的键都在左子树中，大于父节点的键都在右子树中。

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

```
        self.rightchild=right
        self.parent=parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild==self

    def is RightChild(self):
        return self.parent and self.parent.rightChild==self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key=key
        self.payload=value
        self.leftChild=lc
        self.rightChild=rc
        if self.hasLeftChild():
            self.leftChild.parent=self
        if self.hasRightChild():
            self.rightChild.parent=self

    def findSuccessor(self):   #寻找后继节点，本质上就是找比它大的最小值。也就是其
        #中序遍历的下一位，分三种情况。
        succ=None
        if self.hasRightChild():   #①该节点有右子节点，后继节点就是右子树的最小值。
            succ=self.rightChild.findMin()
        else:
            if self.parent:
```

```
            if self.isLeftChild():   #②节点没有右节点，本身是父节点的左子节点，
                #后继节点就是父节点。
                succ=self.parent
            else:       #③后继节点是除他以外父节点的后继节点
                self.parent.rightChild=None #除此之外
                succ=self.parent.findSuccessor()
                self.parent.rightaChild=self
    return succ

def findMin(self):
    current=self
    while current.hasLeftChild():
        current=current.leftChild
        return current

def spliceOut(self):   #将节点从树中摘除的过程
    if self.isLeaf():   #叶子结点
        if self.isLeftChild():
            self.parent.leftChild=None
        else:
            self.parent.rightChild=None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
            if self.isLeftChild():
                self.parent.leftChild=self.leftChild   #子节点上移
            else:
                self.parent.rightChild=self.leftChild
            self.leftChild.parent=self.parent
        else:     #完全对称
            if self.isLeftChild():
                self.parent.leftChild=self.rightChild
            else:
                self.parent.rightChild=self.rightChild
            self.rightChild.parent=self.parent

    def __iter__(self):   #迭代方式：中序遍历
        if self:
            if self.hasLeftChild():
                for elem in self.leftChild:
                    yield elem
            yield self.key
            if self.hasRightChild():
                for elem in self.rightChild:
                    yield elem
```

```
class BinarySearchTree:
    def __init__(self):
        self.root=None
        self.size=0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)    #有根节点递归插入
        else:
            self.root=TreeNode(key,val)    #无根节点创建根节点
        self.size=self.size+1

    def _put(self,key,val,currentNode):
        if key<currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
            else:
                currentNode.leftChild=TreeNode(key,val,parent=currentNode)
        else:
            if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
            else:
                currentNode.rightChild=TreeNode(key,val,parent=currentNode)

    def get(self,key):
        if self.root:
            res=self._get(key,self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None
```

```
def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key==key:
        return currentNode
    elif key<currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)

def delete(self,key):
    if self.size>1:
        nodeToRemove=self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size=self.size-1
        else:
            raise KeyError('key not in tree')
    elif self.size==1 and self.root.key==key:
        self.root=None
        self.size=self.size-1
    else:
        raise KeyError('key not in tree')

def remove(self,currentNode):
    #一旦找到待删除键对应的节点，必须考虑三种情况
    #①待删除节点没有子节点
    if currentNode.isLeaf():
        if currentNode==currentNode.parent.leftChild:
            current.parent.leftChild=None
        else:
            current.parent.rightChild=None

    #②待删除节点有两个子节点
    elif currentNode.hasBothChildren():
        succ=currentNode.findSuccessor()
        succ.spliceOut()
        currentNode=succ.key
        currentNode.payload=succ.payload

    #③待删除节点只有一个子节点
    else:
        if currentNode.hasLeftChild():
            if currentNode.isLeftChild():
```

```
                currentNode.leftChild.parent=currentNode.parent
                currentNode.parent.leftChild=currentNode.leftChild
            elif currentNode.isRightChild():
                currentNode.leftChild.parent=currentNode.parent
                currentNode.parent.rightChild=currentNode.leftChild
            else:
                currentNode.replaceNodeData(currentNode.leftChild.key,
                                            currentNode.leftChild.payload,
                                            currentNode.leftChild.leftChild,
                                            currentNode.leftChild.rightChild)
        else:
            if currentNode.isLeftChild():
                currentNode.rightChild.parent=currentNode.parent
                currentNode.parent.leftChild=currentNode.rightChild
            elif currentNode.isRightChild():
                currentNode.rightChild.parent=currentNode.parent
                currentNode.parent.rightChild=currentNode.rightChild
            else:
                currentNode.replaceNodeData(currentNode.rightChild.key,
                                            currentNode.rightChild.payload,
                                            currentNode.rightChild.leftChild,
                                            currentNode.rightChild.rightChild)
```

如果按照顺序插入，可能会构造出高度为 n 的搜索树，这样函数的时间复杂度均为 O(n)。
因此需要构建：平衡二叉搜索树（AVL 树）

⑥AVL 树
定义平衡因子为左右子树的高度之差：
balance Factor=height(leftSubTree)-height(rightSubTree)
定义平衡因子为-1，0，1 时树是平衡的，时间复杂度为 O(logn)

重新实现 BinarySearchTree 的子类：

```
def _put(self,key,val,currentNode):
    if key<currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key,val,currentNode.leftChild)
        else:
            currentNode.leftChild=TreeNode(key,val,parent=currentNode)
            self.updateBalance(currenntNode.leftChild)    #更新平衡因子
    else:
        if currentNode.hasRightChild():
            self._put(key,val,currentNode.rightChild)
```

```
            else:
                currentNode.rightChild=TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)

    def updateBalance(self,node):   #更新平衡因子
        if node.balanceFactor>1 or node.balanceFactor<-1:
            self.rebalance(node)     #发现失衡调用 rebalanced()进行旋转
            return
        if node.parent!=None:
            if node.isLeftChild():
                node.parent.balanceFactor=node.parent.balanceFactor+1
            elif node.isRightChild():
                node.parent.balanceFactor=node.parent.balanceFactor-1
            if node.parent.balanceFactor!=0:
                self.updateBalance(node.parent)

    def rotateLeft(self,rotRoot):   #左旋
        newRoot=rotRoot.rightChild
        rotRoot.rightChild=newRoot.leftChild
        if newRoot.leftChild!=None:
            newRoot.leftChild.parent=rotRoot
        newRoot.parent=rotRoot.parent
        if rotRoot.isRoot():
            self.root=newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild=newRoot
            else:
                rotRoot.parent.rightChild=newRoot
            newRoot.leftChild=rotRoot
            rotRoot.parent=newRoot
            rotRoot.balanceFactor=rotRoot.balanceFactor+1\
                                    -min(newRoot.balanceFactor,0)
            newRoot.balanceFactor=newRoot.balanceFactor+1\
                                    +max(rotRoot.balanceFactor,0)

    def rotateRight(self,rotRoot):   #右旋是完全对称的
        newRoot=rotRoot.leftChild
        rotRoot.leftChild=newRoot.rightChild
        if newRoot.rightChild!=None:
            newRoot.rightChild.parent=rotRoot
        newRoot.parent=rotRoot.parent
        if rotRoot.isRoot():
            self.root=newRoot
```

```
    else:
        if rotRoot.isLeftChild():
            rotRoot.parent.leftChild=newRoot
        else:
            rotRoot.parent.rightChild=newRoot
        newRoot.rightChild=rotRoot
        rotRoot.parent=newRoot
        rotRoot.balanceFactor=rotRoot.balanceFactor-1+\
                                    max(newRoot.balanceFactor,0)
        newRoot.balanceFactor=newRoot.balanceFactor-1-\
                                    min(rotRoot.balanceFactor,0)

def rebalance(self,node):
    if node.balanceFactor<0:
        if node.rightChild.balanceFactor>0: #RL 型，要先右旋右子节点，再左旋当前节点
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
        else:
            self.rotateLeft(node)    #直接左旋就行
    elif node.balanceFactor>0:
        if node.leftChild.balanceFactor<0:#LR 型，先左旋左子节点，再右旋当前节点
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
        else:
            self.rotateRight(node)    #直接右旋
```

完整的 AVL 树代码：

```
class AVLTree:
    def __init__(self):
        self.root=None
        self.size=0

    def put(self,key,val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key, val)
        self.size += 1

    def _put(self, key, val, currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key, val, currentNode.leftChild)
```

```
            else:
                currentNode.leftChild = TreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.leftChild)
        else:
            if currentNode.hasRightChild():
                self._put(key, val, currentNode.rightChild)
            else:
                currentNode.rightChild = TreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.rightChild)

    def updateBalance(self, node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            self.rebalance(node)
            return
        if node.parent:
            if node.isLeftChild():
                node.parent.balanceFactor += 1
            elif node.isRightChild():
                node.parent.balanceFactor -= 1
            if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)

    def rotateLeft(self, rotRoot):
        newRoot = rotRoot.rightChild
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.leftChild:
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)

    def rotateRight(self, rotRoot):
        newRoot = rotRoot.leftChild
        rotRoot.leftChild = newRoot.rightChild
        if newRoot.rightChild:
```

```
            newRoot.rightChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isRightChild():
                rotRoot.parent.rightChild = newRoot
            else:
                rotRoot.parent.leftChild = newRoot
        newRoot.rightChild = rotRoot
        rotRoot.parent = newRoot
        rotRoot.balanceFactor = rotRoot.balanceFactor - 1 - max(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor - 1 + min(rotRoot.balanceFactor, 0)

    def rebalance(self, node):
        if node.balanceFactor < 0:
            if node.rightChild.balanceFactor > 0:
                self.rotateRight(node.rightChild)
                self.rotateLeft(node)
            else:
                self.rotateLeft(node)
        elif node.balanceFactor > 0:
            if node.leftChild.balanceFactor < 0:
                self.rotateLeft(node.leftChild)
                self.rotateRight(node)
            else:
                self.rotateRight(node)

    def get(self, key):
        if self.root:
            res = self._get(key, self.root)
            return res.payload if res else None
        return None

    def _get(self, key, currentNode):
        if not currentNode:
            return None
        elif key == currentNode.key:
            return currentNode
        elif key < currentNode.key:
            return self._get(key, currentNode.leftChild)
        else:
            return self._get(key, currentNode.rightChild)
```

```
def remove(self, key):
    if self.size > 1:
        nodeToRemove = self._get(key, self.root)
        if nodeToRemove:
            self._remove(nodeToRemove)
            self.size -= 1
        else:
            raise KeyError("Key not found in tree.")
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size -= 1
    else:
        raise KeyError("Key not found in tree.")

def _remove(self, currentNode):
    parent = currentNode.parent

    if currentNode.isLeaf():
        if currentNode.isLeftChild():
            parent.leftChild = None
        elif currentNode.isRightChild():
            parent.rightChild = None
        self.updateBalanceAfterRemove(parent)

    elif currentNode.hasBothChildren():
        succ = currentNode.findSuccessor()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
        self._remove(succ)

    else:
        if currentNode.hasLeftChild():
            child = currentNode.leftChild
        else:
            child = currentNode.rightChild

        if currentNode.isLeftChild():
            parent.leftChild = child
            child.parent = parent
        elif currentNode.isRightChild():
            parent.rightChild = child
            child.parent = parent
        else:
            self.root = child
```

```
                child.parent = None
            self.updateBalanceAfterRemove(parent)

    def updateBalanceAfterRemove(self, node):
        while node:
            self.updateNodeBalance(node)
            if node.balanceFactor > 1 or node.balanceFactor < -1:
                self.rebalance(node)
            if node.balanceFactor == 0:
                node = node.parent
            else:
                break

    def updateNodeBalance(self, node):
        leftHeight = self.getHeight(node.leftChild)
        rightHeight = self.getHeight(node.rightChild)
        node.balanceFactor = leftHeight - rightHeight

    def getHeight(self, node):
        if not node:
            return 0
        return 1 + max(self.getHeight(node.leftChild), self.getHeight(node.rightChild))
```

⑦并查集

将集合合并

```
def GetRoot(a):
    if parent[a]!=a:
        parent[a]=GetRoot(parent[a])
    return parent[a]   #路径压缩，把树的结构扁平化
parent=[i for i in range (N)]

def Merge(a,b):       #把 b 的树根挂在 a 树根下
    parent[GerRoot(b)]=GetRoot(a)

def Query(a,b):       #判断 a,b 是否位于同一棵树下
    return GetRoot(a)==GetRoot(b)
```

例题：已知二叉树的前序遍历序列和后序遍历序列，一共有多少种树的形式？

不能唯一确定的原因：只有前序遍历和后序遍历时，若一个节点只有一个子节点，无法确定该节点是左子节点还是右子节点。

所以该题的思路就是找到所有只有一个子节点的节点个数。

```
preorder=str(input())
postorder=str(input())
```

```
def count_inorders(preorder,postorder):
    def dfs(pre,post):   #深搜
        global total
        n=len(pre)
        if n==0:
            return 1
        if n==1:
            return 1

        for i in range (1,n):   #尝试所有可能的左子树大小
        #在这样的定义下，pre[1,i+1]是左子树，pre[i+1:]是右子树
        #post[0:L]是左子树，post[L:-1]是右子树,post[-1]是根节点
            if pre[1]==post[i-1]:   #左子树的根节点相等，说明这样划分是合理的
                root=pre[0]
                left_size=i
                right_size=n-1-i
                if left_size==0 or right_size==0:
                    total=total+1
                left_count=dfs(pre[1:i+1],post[:i]) #左右子树分别递归
                right_count=dfs(pre[i+1:],post[i:-1])
        return total
    return dfs(preorder,postorder)
total=0
print(2**count_inorders(preorder,postorder))
```

十一、图

名词：顶点、边、权重、路径、环。

①邻接表实现

```
class Vertex:   #定义顶点的的所有信息
    def __init__(self,key):
        self.id=key
        self.connectedTo={}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr]=weight

    def __str__(self):
        return str(self.id)+'connectedTo:'+str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()
```

```
    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]

class Graph:
    def __init__(self):
        self.vertList={}
        self.numVertices=0

    def addVertex(self,key):
        self.numVertices=self.numVertices+1
        newVertex=Vertex(key)
        self.vertList[key]=newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv=self.addVertex(f)
        if t not in self.vertList:
            nv=self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t],cost)
        #双向表再加上    self.vertList[t].addNeighbor(self.vertList[f],cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

②宽度优先搜索（BFS）

词梯问题

```
#先构建单词关系图
def buildGraph(wordFile):
```

```
    d={}
    g=Graph()
    wfile=open(wordFile,'r')
    #创建词桶
    for line in wfile:
        word=line[:-1]   #去掉每行末尾的换行符，得到单词。
        for i in range (len(word)):
            bucket=word[:i]+'_'+word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket]=[word]
    #为同一个桶中的单词添加顶点和边
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
    return g

#宽度优先搜索
def bfs(g,start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue=Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size()>0):
        currentVert=vertQueue.dequeue()   #访问当前节点，出队
        for nbr in currentVert,getConnections():
            if (nbr.getColor()=='white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance())
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)   #子节点入队（下一层）
        currentVert.setColor('black')

#打印词梯
def traverse(y):
    x=y
    while (x.getPred()):   #有前驱就把前驱打印下来
        print(x.getId())
        x=x.getPred()
    print(x.getId())
```

这样写出来的 BFS 问题是只能找到一条最短路径，无法找到全部的最短路径。
所以需要修改代码

```
def bfs_all_preds(graph,start):
    dist={}
    preds=defaultdict(list)   #实现一个字典，记录所有的前驱列表
    queue=deque()
    queue.append(start)
    dist[start]=0

    while queue:
        current=queue.popleft()
        for nbr in graph[current]:
            if nbr not in dist:   #第一次访问
                dist[nbr]=dis[current]+1
                preds[nbr].append(current)
                queue.append(nbr)
            elif dist[nbr]=dis[current]+1:
                preds[nbr].append(current)
    return preds
```

③深度优先搜索（DFS）
骑士周游问题

```
def knightTour(n,path,u,limit):
    #搜索当前深度，已被访问过的顶点列表，希望访问的顶点，路径的顶点总数
    u.setColor('gray')
    path.append(u)
    if n<limit:
        nbrList=list(u.getConnections())
        i=0
        done=False
        while i<len(nbrlist) and not done:
            if nbrList[i].getColor()=='white':
                done=knightTour(n+1,path,nbrList[i],limit)
            i=i+1
        if not done:
            path.pop()
            u.setColor('white')
    else:
        done=True
    return done
```

先选择最少合理走法的格子可以优化查找效率：

```
def orderByAvail(n):
    resList=[]
    for v in n.getConnections():
        if v.getColor()=='white':
            c=0
            for w in v.getConnections():
                if w.getColor()=='white':
                    c=c+1
                resList.append((c,v))
    resList.sort(key=lambda x:x[0])
    return [y[1] for y in resList]
```

实现通用深度优先搜索：

```
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()   #调用 grpah 类的初始化部分
        self.time=0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)   #初始化所有节点的状态
        for aVertex in self:
            if aVertex.getColor()=='white':
                self.dfsvisit(aVertex)   #实现深度优先搜索

    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time=self.time+1   #被发现的时间
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor()=='white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
            startVertex.setColor('black')
            self.time=self.time+1   #搜索完成的时间
            startVertex.setFinish(self.time)
```

④图的算法

1、Dijkstra 算法：从一个点出发找最短路线

思路核心：从源点出发，不断扩展“当前最短路径确定的点”，

直到所有点都确定最短路径。

思路：①从源点开始，初始化所有距离为 float('inf')，源点距离为 0；

②维护一个优先队列（heapq），每次取当前距离最小的点 u；
③对于每个与 u 相邻的点 v，若通过 u 到 v 的距离更短，更新 dist[v]；
④重复上述过程。

```
import PriorityQueue
def dijkstra(aGraph,start):
    pq=PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert=pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist=currentVert.getDistance()\
                    +currentVert.getWeight(nextVert)
        if newDist<nextVert.getDistance():
            nextVert.setDistance(newDist)
            nextVert.setPred(currentVert)
            pq.decreaseKey(nextVert,newDist)
```

用 heapq 实现：本质上就是使用了 heapq 的 BFS

```
import heapq
def dijkstra(graph,start):
    dist={node:float('inf') for node in graph}
    dist[start]=0
    visited=set()
    heap=[(0,start)]   #距离、节点的顺序

    while heap:
        dis,cur_u=heapq.heappop(heap)
        if cur_u in visited:
            continue
        visited.add(cur_u)

        for v,w in graph[cur_u]:
            if dist[v]>dist[u]+w:
                dist[v]=dist[u]+w
                heapq.heappush(heap,(dist[v],v))
    return dist
```

2、Prim 算法：全部连接如何成本最小（从点拓展边）
思路：从任意一个点开始，用最小堆 heapq 维护“所有可以连接到外部的边”
每次取最小边(w,u,v)，如果 v 没访问过，就把它加入生成树，并把邻边加入堆
重复直到所有点都加入生成树。

```
def prim(G,start):
    pq=PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v)for v in G])
    while not pq.isEmpty():
        currentVert=pq.delMin()
        for nextVert in currentVert.getConnections():
            newCost=currentVert.getWeight(nextVert)\
                     +currentVert.getDistance()
            if v in pq and newCost<nextVert.getDistance():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(nextVert,newCost)
```

heapq 实现：

```
import heapq
def prim(n,graph):
    visited=set()
    heap=[(0,0)]
    total_weight=0

    while heap and len(visited)<n:
        weight,u=heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        total_weight=total_weight+weight

        for v,w in graph[u]:
            if v not in visited:
                heapq.heappush(heap,(w,v))

    return total_weight
```

3、拓扑排序

对有向无环图调用 DFS 算法，得到每个顶点的“结束时间”。

按照每个顶点的“结束时间”从大到小排序，输出这个次序下的顶点列表。

算法：①从图中任选一个没有前驱（入度为零）的顶点 x 输出；

②从图中删除 x 和所有以它为起点的边，再找没有前驱的顶点输出；

③以此类推，直到不存在无前驱的顶点为止。

例：给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。

输入
若干行整数，第一行有 2 个数，分别为顶点数 v 和弧数 a，接下来有 a 行，
每一行有 2 个数，分别是该条弧所关联的两个顶点编号。v<=100, a<=500

输出
若干个空格隔开的顶点构成的序列(用小写字母)。

样例输入
6 8
1 2
1 3
1 4
3 2
3 5
4 5
6 4
6 5

样例输出
v1 v3 v2 v6 v4 v5

```
from collections import defaultdict
import heapq

v,a=map(int,input().split( ))
d=defaultdict(list)
degree=[0]*(v+1)

for i in range (a):
    x,y=map(int,input().split( ))
    d[x].append(y)
    degree[y]=degree[y]+1

heap=[]

for i in range (1,v+1):
    if degree[i]==0:
        heapq.heappush(heap,i)

result=[]
```

```
while heap:
    u=heapq.heappop(heap)
    result.append(f'v{u}')
    for nbr in d[u]:
        degree[nbr]=degree[nbr]-1
        if degree[nbr]==0:
            heap.heappush(heap,nbr)

print(' '.join(result))
```

在拓扑排序的基础上，可以求解每个事件的最早发生时间和最晚发生时间。

```
from collections import defaultdict
import heapq

v,a=map(int,input().split( ))
d=defaultdict(list)
rev_d=defaltdict(list)  #反向图
degree=[0]*(v+1)  #顶点从 1 开始
weight=dict()  #记录每条边的权重

for i in range (a):
    x,y,w=map(int,input().split( ))
    d[x].append(y)
    rev_d[y].append(x)
    weight[(x,y)]=w
    degree[y]=degree[y]+1  #有一条边到 y，y 的入度加 1

heap=[]
for i in range (1,v+1):
    if degree[i]==0:
        heapq.heappush(heap,i)

result=[]  #记录拓扑排序结果
earliest=[0]*(v+1) #记录最早发生时间

while heap:
    u=heap.heappop(heap)
    result.append(u)
    for nbr in d[u]:
        earliest[nbr]=max(earliest[nbr],earliest[u]+weight[(u,nbr)])
        degree[nbr]=degree[nbr]-1
        if degree[nbr]==0:
```

```
                    heap.heappush(heap,nbr)
print(earliest)

project_time=max(earliest[i] for i in range (1,v+1))    #完成任务的最短时间

lastest=[project_time]*(v+1)    #初始化，事件 i 最晚不影响工期的时间

for u in reversed(result):
    for pre in rev_d[u]:
        lastest[pre]=min(lastest[pre],lastest[u]-weight[(u,pre)])

print(lastest)
```

4、Kosaraju 算法

在有向图中寻找所有的强连通分量（u 可到达 v，v 也可到达 u）

算法思路：

①对原图进行 DFS，得到完成时间栈

②构建反图

③栈顶出栈，在反图中 DFS，即可找到强连通分量

④重复过程直到栈空

```
def kosaraju(graph):
    n=len(graph)
    visited=[False]*n
    stack=[]

    def dfs1(u): #正向图 dfs
        visited[u]=True
        for v in graph[u]:
            if not visited[v]:
                dfs1(v)
        stack.append(u)    #按照时间顺序入栈

    for u in range (n):
        if not visited[u]:
            dfs1(u)

    rev_graph=[[] for i in range (n)]    #构建反图
    for u in range (n):
        for v in graph[u]:
            rev_graph[v].append(u)

    visited=[False]*n
```

```python
    sccs=[]    #强连通分量

    def dfs2(u,component):
        visited[u]=True
        component.append(u)
        for v in rev_graph[u]:
            if not visited[v]:
                dfs2(v,component)

    while stack:
        u=stack.pop()
        if not visited[u]:
            component=[]
            dfs2(u,component)
            sccs.append(component)

    return sccs
```

5、Kruskal 算法（从边合并点）
用来求最小生成树

核心思想：贪心策略
从权值最小的边开始，逐步加边，只要不形成环，就加入最小生成树。

算法思路：
①将所有边按权值升序排序
②若选取的边使生成的图无环，则并入 TE；若有环则舍弃。直到有 N-1 条边。
用并查集提高效率

```python
def GetRoot(a):
    if parent[a]==a:
        return a
    parent[a]=GetRoot(parent[a])
    return parent[a]

def Merge(a,b):
    p1=GetRoot(a)
    p2=GetRoot(b)
    if p1==p2:
        return
    parent[p2]=p1

def Union(a,b):
    return GetRoot(a)==GetRoot(b)
```

```
edges.sort()    #按权值排序
parent=[i for i in range (n)]
weight=0
count=0

for w,u,v in edges:
    if not Union(u,v):    #如果是 True，连接就会形成环，跳过
        Merge(u,v)
        weight=weight+w
        count=count+1
        if count=n-1:
            break
print(weight)
```