数算学习随记——基本数据结构合集

compiled by zxy　since the start of the class.

前言：是在学习数算过程中的一些笔记和整理，有一些内容是从Python数据结构与算法分析（第2版）里面截取的，有一些是gpt给出的讲解，还有一些是自己学习后整理出来的，都是很基础的数据结构简介及经典算法原理及程序实现，记录了我学习这门课程的过程。

Tips：类

当我们定义一个类时，实际上是在创建一种新的对象类型。类（class）是面向对象编程的基本概念之一，它允许我们定义一组属性和方法，以便创建具有特定行为和功能的对象。

类的定义通常包含以下几个部分：

1. 类名：类名用于标识这个类的名称，通常采用大写字母开头的驼峰命名法。在这个例子中，类名是 `Deque`。
2. 构造函数（`__init__`）：构造函数是在创建类的实例时自动调用的特殊方法。它用于初始化类的属性和执行一些必要的设置操作。在这个例子中，构造函数 `__init__(self)` 创建了一个空列表 `self.items`，作为双端队列的内部存储结构。（见下的代码）
3. 方法：方法是在类中定义的函数，用于描述类的行为和功能。方法的第一个参数通常是 `self`，代表类的实例。通过 `self`，我们可以访问类的属性和调用其他方法。
    - `isEmpty(self)`：判断双端队列是否为空，通过比较 `self.items` 是否为空列表来实现。

零、列表

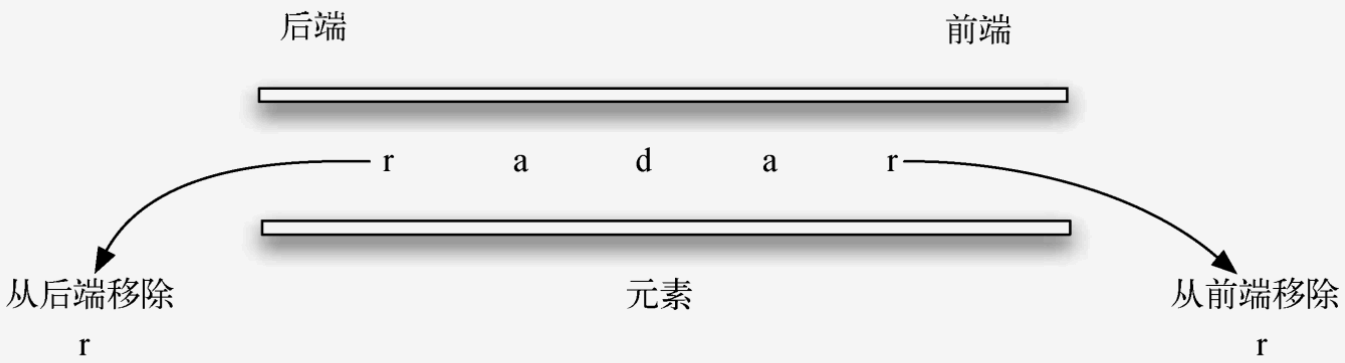
1. `insert()` 方法：
    - `insert(index, element)` 方法用于在指定位置 `index` 插入一个新元素 `element`。
    - 插入后，列表中原来在该位置及其后面的元素都会向右移动。
    - 如果 `index` 超过了列表的长度，则新元素将被插入到列表的末尾。

```
pythonmy_list = [1, 2, 3, 4]
my_list.insert(2, 5)  # 在索引为2的位置插入元素5
print(my_list)  # 输出：[1, 2, 5, 3, 4]
```

2. `pop()` 方法：
    - `pop(index)` 方法用于删除并返回指定位置 `index` 处的元素。
    - 如果不提供 `index` 参数，则默认删除并返回列表中的最后一个元素。
    - 删除后，列表中原来在该位置后面的元素都会向左移动。
    - 如果使用`pop()`方法而不提供索引参数，则默认删除并返回列表中的最后一个元素。

```
pythonmy_list = [1, 2, 3, 4]
popped_element = my_list.pop(2)  # 删除索引为2的元素，并返回删除的元素
print(popped_element)  # 输出：3
print(my_list)  # 输出：[1, 2, 4]
```

壹、双端队列



从前端和后端移除元素

图 3-17　双端队列示例

表 3-6　双端队列操作示例

双端队列操作	双端队列内容	返　回　值
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear('dog')	['dog', 4]	
d.addFront('cat')	['dog', 4, 'cat']	
d.addFront(True)	['dog', 4, 'cat', True]	
d.size()	['dog', 4, 'cat', True]	4
d.isEmpty()	['dog', 4, 'cat', True]	False
d.addRear(8.4)	[8.4, 'dog', 4, 'cat', True]	
d.removeRear()	['dog', 4, 'cat', True]	8.4
d.removeFront()	['dog', 4, 'cat']	True

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

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
```

```
        return len(self.items)

d = Deque()
print(d.isEmpty())
d.addRear(4)
d.addRear('dog')
d.addFront('cat')
d.addFront(True)
print(d.size())
print(d.isEmpty())
d.addRear(8.4)
print(d.removeRear())
print(d.removeFront())
```

当然python自己提供了双端队列：`from collections import deque

下面是 `deque` 提供的一些主要功能和方法：

```
from collections import deque
# 初始化一个空的deque
d = deque()
# 在右侧添加元素
d.append(1)
d.append(2)
# 在左侧添加元素
d.appendleft(0)
# 删除右侧元素
print(d.pop())
# 删除左侧元素
print(d.popleft())
# 打印deque中的元素
print(d)
```

贰、前中后序表达式转化

1. 从中序向前序和后序转换

到目前为止，我们使用了特定的方法来将中序表达式转换成对应的前序表达式和后序表达式。正如你所想，存在通用的算法，可用于正确转换任意复杂度的表达式。

首先使用完全括号表达式。如前所述，可以将 A + B * C 写作(A + (B * C))，以表示乘号的优先级高于加号。进一步观察后会发现，每一对括号其实对应着一个中序表达式（包含两个操作数以及其间的运算符）。

观察子表达式(B * C)的右括号。如果将乘号移到右括号所在的位置，并且去掉左括号，就会得到 B C *，这实际上是将该子表达式转换成了对应的后序表达式。如果把加号也移到对应的右括号所在的位置，并且去掉对应的左括号，就能得到完整的后序表达式，如图 3-6 所示。

图 3-6　向右移动运算符，以得到后序表达式

如果将运算符移到左括号所在的位置，并且去掉对应的右括号，就能得到前序表达式，如图 3-7 所示。实际上，括号对的位置就是其包含的运算符的最终位置。

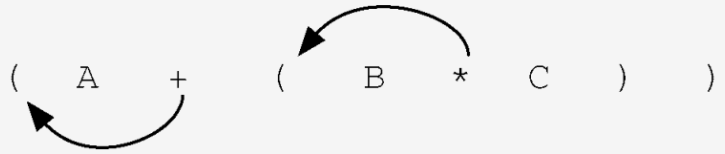


图 3-7　向左移动运算符，以得到前序表达式

再一次研究 A + B * C 这个例子。如前所示，其对应的后序表达式为 A B C * +。操作数 A、B 和 C 的相对位置保持不变，只有运算符改变了位置。再观察中序表达式中的运算符。从左往右看，第一个出现的运算符是+。但是在后序表达式中，由于*的优先级更高，因此*先于+出现。在本例中，中序表达式的运算符顺序与后序表达式的相反。

在转换过程中，由于运算符右边的操作数还未出现，因此需要将运算符保存在某处。同时，由于运算符有不同的优先级，因此可能需要反转它们的保存顺序。本例中的加号与乘号就是这种情况。由于中序表达式中的加号先于优先级更高的乘号出现，因此后序表达式需要反转它们的出现顺序。鉴于这种反转特性，使用栈来保存运算符就显得十分合理。

对于(A + B) * C，情况会如何呢？它对应的后序表达式为 A B + C *。从左往右看，首先出现的运算符是+。不过，由于括号改变了运算符的优先级，因此当处理到*时，+已经被放入结果表达式中了。现在可以来总结转换算法：当遇到左括号时，需要将其保存，以表示接下来会遇到高优先级的运算符；那个运算符需要等到对应的右括号出现才能确定其位置（回忆一下完全括号表达式的转换法）；当右括号出现时，便可以将运算符从栈中取出来。

在从左往右扫描中序表达式时，我们利用栈来保存运算符。这样做可以提供反转特性。栈的顶端永远是最新添加的运算符。每当遇到一个新的运算符时，都需要对比它与栈中运算符的优先级。

假设中序表达式是一个以空格分隔的标记串。其中，运算符标记有*、/、+和-，括号标记有(和)，操作数标记有 A、B、C 等。下面的步骤会生成一个后序标记串。

Shunting Yard 算法是一种用于将中缀表达式转换为后缀表达式的算法。它由荷兰计算机科学家 Edsger Dijkstra 在1960年代提出，用于解析和计算数学表达式。

Shunting Yard 算法的主要思想是使用两个栈（运算符栈和输出栈）来处理表达式的符号。算法按照运算符的优先级和结合性，将符号逐个处理并放置到正确的位置。最终，输出栈中的元素就是转换后的后缀表达式。

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：
        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

叁、栈

FILO，其实跟列表的append和pop一模一样。

肆、树！

代码清单 6-2　插入左子树

```
1   def insertLeft(root, newBranch):
2       t = root.pop(1)
3       if len(t) > 1:
4           root.insert(1, [newBranch, t, []])
5       else:
6           root.insert(1, [newBranch, [], []])
7       return root
```

在插入左子树时，先获取当前的左子树所对应的列表（可能为空），然后加入新的左子树，将旧的左子树作为新节点的左子树。这样一来，就可以在树的任意位置插入新节点。`insertRight` 与 `insertLeft` 类似，如代码清单 6-3 所示。

代码清单 6-3　插入右子树

```
1   def insertRight(root, newBranch):
2       t = root.pop(2)
3       if len(t) > 1:
4           root.insert(2, [newBranch, [], t])
5       else:
6           root.insert(2, [newBranch, [], []])
7       return root
```

个叶子节点。还有一个很好的性质，那就是这种表示法可以推广到有很多子树的情况。如果树不是二叉树，则多一棵子树只是多一个列表。

```
>>> myTree = ['a', ['b', ['d', [], []], ['e', [], []]], \
                   ['c', ['f', [], []], []]]
>>> myTree
['a', ['b', ['d', [], []], ['e', [], []]],
      ['c', ['f', [], []], []]]
>>> myTree[1]
['b', ['d', [], []], ['e', [], []]]
>>> myTree[0]
'a'
>>> myTree[2]
['c', ['f', [], []], []]
```

代码清单 6-4　树的访问函数

```
1   def getRootVal(root):
2       return root[0]
3
4   def setRootVal(root, newVal):
5       root[0] = newVal
6
7   def getLeftChild(root):
8       return root[1]
9
10  def getRightChild(root):
11      return root[2]
```

括号表示树建树：

```
class TreeNode:
    def __init__(self, value):
```

```
        self.value = value
        self.children = []

# A(B(E),C(F,G),D(H(I)))
def parse_tree(s):
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点 m
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点
```

这段代码实现了一个解析器，用于将表达树结构的字符串转换为树形数据结构。下面我会通过一个例子来说明这个过程：

假设有输入字符串 `s = "A(B(E),C(F,G),D(H(I)))"`，表示如下的树结构：

```
   A
 / | \
B  C  D
| / \ |
E F  G H
       |
       I
```

1. 首先定义了一个 `TreeNode` 类，表示树的节点，每个节点包含一个值和子节点列表。
2. 在函数 `parse_tree` 中，依次遍历输入字符串中的每个字符。
3. 对于字符 'A'，创建节点 A，并将其赋给 `node`。
4. 对于字符 '('，将节点 A 推入栈中，然后将 `node` 置为 `None`。
5. 对于字符 'B'，创建节点 B，并将其作为节点 A 的子节点。
6. 对于字符 'E'，创建节点 E，并将其作为节点 B 的子节点。
7. 对于字符 ')'，弹出栈顶节点 B，将 `node` 设为 B。
8. 对于字符 'C'，创建节点 C，并将其作为节点 A 的子节点。
9. 对于字符 'F'，创建节点 F，并将其作为节点 C 的子节点。
10. 对于字符 'G'，创建节点 G，并将其作为节点 C 的子节点。
11. 对于字符 ')'，弹出栈顶节点 A，将 `node` 设为 A。
12. 对于字符 'D'，创建节点 D，并将其作为节点 A 的子节点。

13. 对于字符 'H'，创建节点 H，并将其作为节点 D 的子节点。
14. 对于字符 'I'，创建节点 I，并将其作为节点 H 的子节点。
15. 最后返回根节点 A。

这样就完成了对输入字符串的解析，构建了相应的树形数据结构。

树的前中后序遍历：

树的中序、后序和前序遍历是三种常见的树遍历方式，它们分别按照不同顺序访问树中的节点。这里简单介绍一下它们的遍历方式：

1. **前序遍历（Preorder Traversal）**：
    - 访问根节点
    - 前序遍历左子树
    - 前序遍历右子树
2. **中序遍历（Inorder Traversal）**：
    - 中序遍历左子树
    - 访问根节点
    - 中序遍历右子树
3. **后序遍历（Postorder Traversal）**：
    - 后序遍历左子树
    - 后序遍历右子树
    - 访问根节点

下面是对一棵二叉树进行前序、中序和后序遍历的示例：

```
    1
   / \
  2   3
 / \
4   5
```

- **前序遍历**：1 -> 2 -> 4 -> 5 -> 3
- **中序遍历**：4 -> 2 -> 5 -> 1 -> 3
- **后序遍历**：4 -> 5 -> 2 -> 3 -> 1

二叉搜索树：

二叉搜索树依赖于这样一个性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性，它会引导我们实现上述映射接口。图 6-20 描绘了二叉搜索树的这个性质，图中只展示了键，没有展示对应的值。注意，每一对父节点和子节点都具有这个性质。左子树的所有键都小于根节点的键，右子树的所有键则都大于根节点的键。

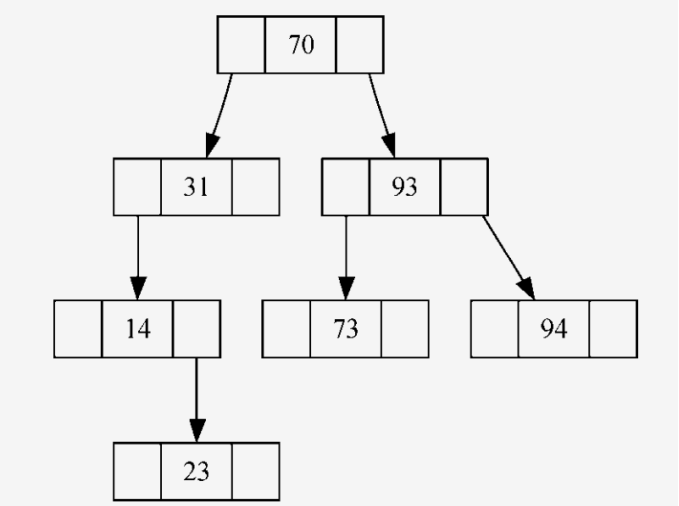


图 6-20　简单的二叉搜索树

接下来看看如何构造二叉搜索树。图 6-20 中的节点是按如下顺序插入键之后形成的：70、31、93、94、14、23、73。因为 70 是第一个插入的键，所以是根节点。31 小于 70，所以成为 70 的左子节点。93 大于 70，所以成为 70 的右子节点。现在树的两层已经满了，所以下一个键会成为 31 或 93 的子节点。94 比 70 和 93 都要大，所以它成了 93 的右子节点。同理，14比 70 和 31 都要小，所以它成了 31 的左子节点。23 也小于 31，所以它必定在 31 的左子树中。而它又大于 14，所以成了 14 的右子节点。

节点和引用&二叉搜索树的实现：

```
class TreeNode:#节点
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

class BinarySearchTree:#引用
    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 insert(self, key):  #插入新节点
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_recursively(self.root, key)
```

```
            #如果二叉搜索树是一个无根树（即空树），那么新插入的节点将作为根节点。在这种情况下，新插入的节点将成为树的根节点，并                且不需要考虑在哪里插入，因为树中只有一个节点，即新插入的节点。如果之前树是空的，而现在插入了一个节点，那么这个节点                就是新的根节点。这个节点没有父节点，它就是树的根，其他节点将会成为它的子节点。
        self.size += 1

    def _insert_recursively(self, node, key):  #递归实现有根树新节点的插入
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert_recursively(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert_recursively(node.right, key)

    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 __contains__(self, key): #检查树中是否有某个键
        if self._get(key, self.root):
            return True
        else:
            return False
    #你应该记得，__contains__方法重载了 in 运算符，因此我们可以写出这样的语句：
    #if 'Northfield' in myZipTree:   print("oom ya ya")

# 测试代码
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
```

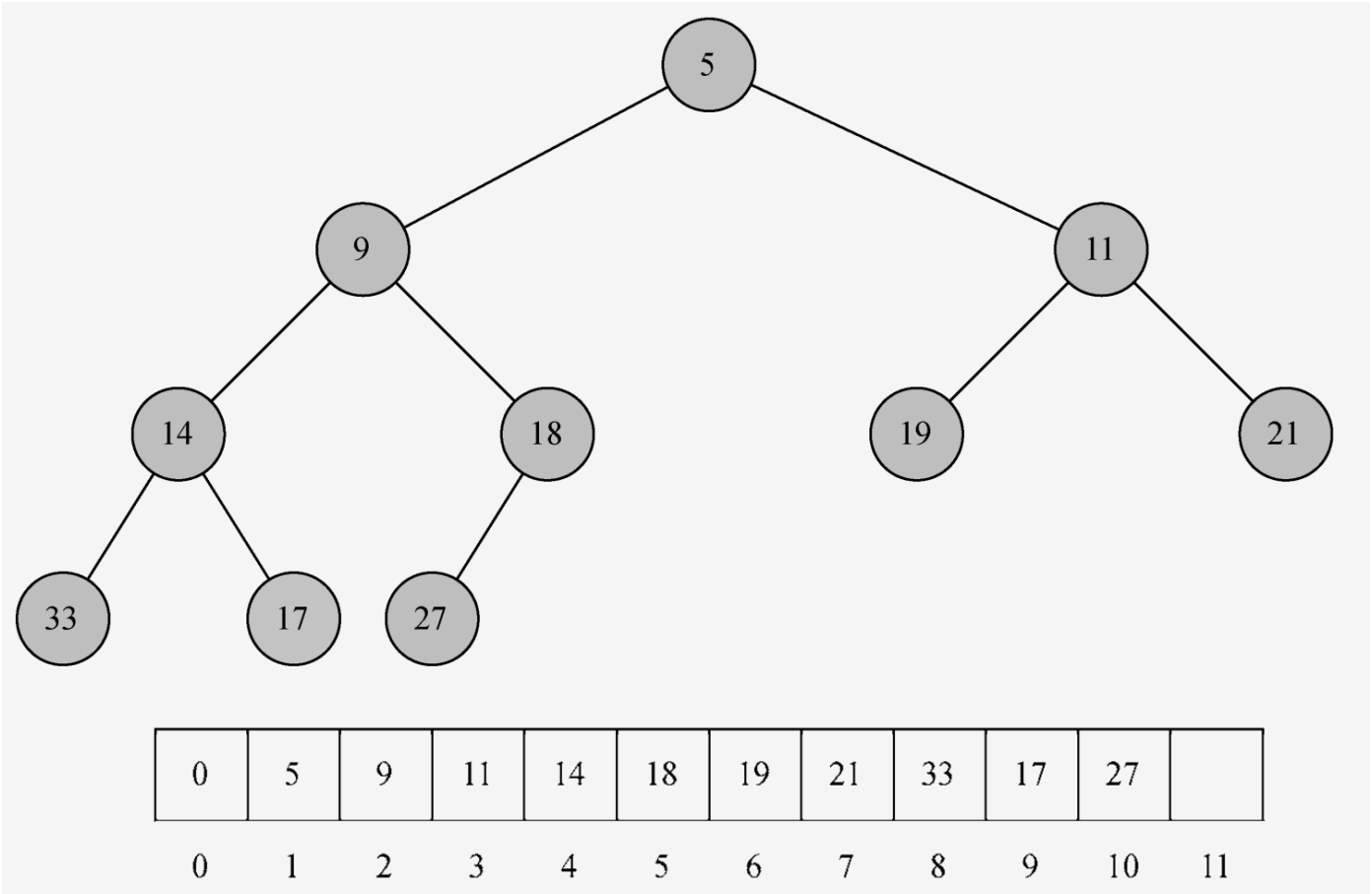
```
    bst.insert(20)
    bst.insert(40)

    print("Inorder traversal:", bst.inorder_traversal())
    print("Tree size:", len(bst))
```

上面的代码定义了 `TreeNode` 节点类和 `BinarySearchTree` 二叉搜索树类，包括了初始化方法、插入节点方法、中序遍历方法以及一些辅助方法。在测试代码中，创建了一个二叉搜索树并进行了节点的插入操作，然后进行了中序遍历并输出树的大小。

二叉堆：

为了使二叉堆能高效地工作，我们利用树的对数性质来表示它。你会在 6.7.3 节学到，为了保证对数性能，必须维持树的平衡。平衡的二叉树是指，其根节点的左右子树含有数量大致相等的节点。在实现二叉堆时，我们通过创建一棵完全二叉树来维持树的平衡。在完全二叉树中，除了最底层，其他每一层的节点都是满的。在最底层，我们从左往右填充节点。

完全二叉树的另一个有趣之处在于，可以用一个列表来表示它，而不需要采用“列表之列表”或“节点与引用”表示法。由于树是完全的，因此对于在列表中处于位置 p 的节点来说，它的左子节点正好处于位置 $2p$；同理，右子节点处于位置 $2p+1$。若要找到树中任意节点的父节点，只需使用 Python 的整数除法即可。给定列表中位置 n 处的节点，其父节点的位置就是 $n/2$。




```
class BinaryHeap:
    def __init__(self):
        """
        构造方法：初始化一个空的二叉堆。
        heapList：用于存储堆元素的列表，初始值为[0]。
```

```
        currentSize：当前堆的大小，初始值为0。
        """
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self, i):
        """
        辅助方法：上浮操作，用于维护堆的结构性质。
        参数i：当前节点的索引。
        """
        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):
        """
        插入方法：向堆中插入一个新的元素。
        参数k：待插入的元素。
        """
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self, i):
        """
        辅助方法：下沉操作，用于维护堆的结构性质。
        参数i：当前节点的索引。
        """
        while (i * 2) <= self.currentSize:
            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):
        """
        辅助方法：获取当前节点的较小子节点。
        参数i：当前节点的索引。
        返回值：较小子节点的索引。
        """
        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):
        """
        删除最小元素方法：删除并返回堆中的最小元素。
        返回值：被删除的最小元素。
        """
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self, alist):
        """
        构建堆方法：根据给定的元素列表构建一个堆。
        参数alist：待构建堆的元素列表。
        """
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

# Example usage:
bh = BinaryHeap()
bh.buildHeap([9, 6, 5, 2, 3])
print(bh.heapList)  # Output: [0, 2, 3, 5, 6, 9]
```

哈夫曼编码树：

哈夫曼编码树（Huffman Coding Tree）是一种用于数据压缩的编码方式，由大卫·哈夫曼于1952年提出。它通过使用变长编码，使得频率较高的字符使用较短的编码，频率较低的字符使用较长的编码，从而实现压缩数据的目的。哈夫曼编码是一种前缀编码，即任何编码的前缀都不是其他编码的前缀，这确保了编码的唯一性和解码的准确性。

算法思路

构建哈夫曼编码树的基本思路如下：

1. **统计频率**：
   - 对每个字符进行统计，计算其出现的频率。
2. **创建优先队列**：
   - 创建一个优先队列（通常使用最小堆），将所有字符及其频率作为叶节点插入队列。
3. **构建哈夫曼树**：
   - 从队列中取出两个频率最小的节点，创建一个新节点，频率为两个节点频率之和，新节点的左子节点为第一个取出的节点，右子节点为第二个取出的节点。
   - 将新节点插入队列。

- 重复以上步骤，直到队列中只剩下一个节点，即哈夫曼树的根节点。

4. **生成编码**：
   - 从哈夫曼树的根节点出发，通过遍历树，为每个字符生成对应的哈夫曼编码。通常，左子节点编码为“0”，右子节点编码为“1”。

具体步骤

以下是构建哈夫曼编码树的具体步骤：

1. 统计频率

假设有字符及其频率如下：

```
A: 5, B: 9, C: 12, D: 13, E: 16, F: 45
```

2. 创建优先队列

将所有字符及其频率作为节点插入优先队列：

```
[(5, 'A'), (9, 'B'), (12, 'C'), (13, 'D'), (16, 'E'), (45, 'F')]
```

3. 构建哈夫曼树

- 从队列中取出两个频率最小的节点 `(5, 'A')` 和 `(9, 'B')`，创建一个新节点 `14`，并插入队列：

  ```
  [(12, 'C'), (13, 'D'), (14, 'AB'), (16, 'E'), (45, 'F')]
  ```

- 继续取出两个频率最小的节点 `(12, 'C')` 和 `(13, 'D')`，创建一个新节点 `25`，并插入队列：

  ```
  [(14, 'AB'), (16, 'E'), (25, 'CD'), (45, 'F')]
  ```

- 重复以上步骤，直至队列中只剩下一个节点：

  ```
  [(29, 'ABE'), (45, 'F')]
  [(54, 'CDEF')]
  [(100, 'ABCDEFG')]
  ```

4. 生成编码

从根节点出发，遍历树生成编码：

- 对于叶节点 `'A'`，编码路径为 `00`
- 对于叶节点 `'B'`，编码路径为 `01`
- 对于叶节点 `'C'`，编码路径为 `100`
- 对于叶节点 `'D'`，编码路径为 `101`
- 对于叶节点 `'E'`，编码路径为 `110`
- 对于叶节点 `'F'`，编码路径为 `111`

最终编码为：

```
A: 00, B: 01, C: 100, D: 101, E: 110, F: 111
```

代码示例

以下是一个使用 Python 实现的哈夫曼编码的例子：

```python
import heapq
from collections import defaultdict

class Node:
    def __init__(self, freq, char=None, left=None, right=None):
        self.freq = freq
        self.char = char
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(frequencies):
    heap = [Node(freq, char) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.freq + right.freq, left=left, right=right)
        heapq.heappush(heap, merged)

    return heap[0]

def generate_huffman_codes(node, prefix="", codebook=None):
    if codebook is None:
        codebook = {}
    if node.char is not None:
        codebook[node.char] = prefix
    else:
        generate_huffman_codes(node.left, prefix + "0", codebook)
        generate_huffman_codes(node.right, prefix + "1", codebook)
    return codebook

# Example usage
frequencies = {
    'A': 5, 'B': 9, 'C': 12, 'D': 13, 'E': 16, 'F': 45
}
huffman_tree = build_huffman_tree(frequencies)
huffman_codes = generate_huffman_codes(huffman_tree)

print("Huffman Codes:", huffman_codes)
```

总结：哈夫曼编码树通过构建最优前缀编码树，使得频率较高的字符使用较短的编码，达到数据压缩的目的。构建过程主要涉及统计频率、创建优先队列、构建哈夫曼树以及生成编码四个步骤。最终生成的哈夫曼编码可以显著减少数据存储和传输的成本。

二叉平衡树：

下面是一个简单的二叉平衡树（AVL树）的实现，带有详细注释：

```
class TreeNode:
    def __init__(self, key):
        """
        树节点类
        :param key: 节点的键值
        """
        self.key = key  # 键值
        self.left = None  # 左子节点
        self.right = None  # 右子节点
        self.height = 1  # 节点的高度，默认为1

class AVLTree:
    def __init__(self):
        """AVL树类"""
        self.root = None  # 根节点

    def height(self, node):
        """
        获取节点的高度
        :param node: 节点
        :return: 节点的高度，如果节点为None则返回0
        """
        if node is None:
            return 0
        return node.height

    def balance_factor(self, node):
        """
        计算节点的平衡因子
        :param node: 节点
        :return: 平衡因子
        """
        if node is None:
            return 0
        return self.height(node.left) - self.height(node.right)

    def rotate_right(self, y):
        """
        右旋操作
        :param y: 不平衡节点
        :return: 新的子树根节点
        """
        x = y.left
        T2 = x.right
```

```
        # 执行旋转
        x.right = y
        y.left = T2

        # 更新节点的高度
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        x.height = 1 + max(self.height(x.left), self.height(x.right))

        return x

    def rotate_left(self, x):
        """
        左旋操作
        :param x: 不平衡节点
        :return: 新的子树根节点
        """
        y = x.right
        T2 = y.left

        # 执行旋转
        y.left = x
        x.right = T2

        # 更新节点的高度
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        y.height = 1 + max(self.height(y.left), self.height(y.right))

        return y

    def insert(self, root, key):
        """
        向树中插入节点
        :param root: 当前子树的根节点
        :param key: 待插入的键值
        :return: 插入后的根节点
        """
        # 执行标准BST插入
        if root is None:
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # 更新节点的高度
        root.height = 1 + max(self.height(root.left), self.height(root.right))

        # 获取节点的平衡因子
        balance = self.balance_factor(root)

        # 如果节点不平衡，则进行相应的旋转操作
```

```
        if balance > 1:
            if key < root.left.key:
                return self.rotate_right(root)
            else:
                root.left = self.rotate_left(root.left)
                return self.rotate_right(root)
        if balance < -1:
            if key > root.right.key:
                return self.rotate_left(root)
            else:
                root.right = self.rotate_right(root.right)
                return self.rotate_left(root)

        return root

    def pre_order_traversal(self, root):
        """
        前序遍历
        :param root: 当前子树的根节点
        """
        if root:
            print("{0} ".format(root.key), end="")
            self.pre_order_traversal(root.left)
            self.pre_order_traversal(root.right)

# 示例用法
if __name__ == "__main__":
    avl_tree = AVLTree()
    keys = [9, 5, 10, 0, 6, 11, -1, 1, 2]
    for key in keys:
        avl_tree.root = avl_tree.insert(avl_tree.root, key)

    print("Preorder traversal of the constructed AVL tree is:")
    avl_tree.pre_order_traversal(avl_tree.root)
```

这个示例实现了一个简单的二叉平衡树（AVL树），包括节点的插入和前序遍历操作。

并查集（DSU）：

下面是一个简单的并查集（Disjoint Set）的 Python 实现：

```
class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
```

```
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return

        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        elif self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1

# 示例用法
if __name__ == "__main__":
    n = 5  # 假设有5个元素
    ds = DisjointSet(n)

    # 进行一系列合并操作
    ds.union(0, 1)
    ds.union(2, 3)
    ds.union(0, 4)

    # 打印每个元素的根节点
    for i in range(n):
        print(f"Element {i}: root = {ds.find(i)}")
```

这段代码创建了一个简单的并查集类 `DisjointSet`，其中 `parent` 数组存储了每个元素的父节点，`rank` 数组用于优化合并操作。`find` 方法用于查找元素所在集合的根节点，`union` 方法用于合并两个集合。

在示例用法中，首先创建了一个大小为 5 的并查集，然后进行了一系列的合并操作，并打印了每个元素所属集合的根节点。

当使用并查集时，通常会涉及两种基本操作：查找（Find）和合并（Union）。

1. **Find（查找）操作：**

   查找操作用于找到某个元素所属的集合（根节点）。通常情况下，我们希望通过查找操作来判断两个元素是否属于同一个集合。该操作的目的是找到元素所在集合的代表元素（根节点）。

   在并查集中，Find操作通常使用递归或迭代的方式实现。具体来说，当我们要查找某个节点的根节点时，我们会沿着该节点所在集合的父节点一直向上查找，直到找到根节点为止。在这个过程中，我们可以利用路径压缩技术，即在查找路径上的每个节点都直接指向根节点，从而优化查找操作的性能。

2. **Union（合并）操作：**

   合并操作用于将两个不相交的集合合并成一个集合。在并查集中，合并操作通常会选择两个集合的根节点进行合并。

   具体而言，合并操作的步骤如下：

   - 首先，我们通过Find操作找到两个元素所在集合的根节点。
   - 然后，我们将其中一个根节点的父节点指向另一个根节点，从而将两个集合合并成一个集合。

- 在合并时，我们可以根据集合的大小或高度等因素来优化合并策略，例如按秩合并。

综上所述，Find操作用于查找元素所属的集合，而Union操作用于将两个集合合并成一个集合。这两种操作是并查集的核心操作，通常用于解决连接性问题。

伍、链表

链表是一种常见的线性数据结构，用于存储一系列元素。它由一系列节点组成，每个节点包含数据以及指向下一个节点的引用（指针）。链表的特点是节点之间不必在内存中连续存储，它们通过指针相连，因此可以灵活地分配和释放内存。

链表通常分为两种主要类型：单向链表和双向链表。

1. **单向链表（Singly Linked List）**： 单向链表中，每个节点包含数据以及指向下一个节点的指针。链表的开始称为头节点，最后一个节点的指针指向一个特殊的值（如 None），表示链表的结束。单向链表只允许从头节点开始逐个访问节点。
2. **双向链表（Doubly Linked List）**： 双向链表中，每个节点除了包含数据和指向下一个节点的指针外，还包含一个指向前一个节点的指针。这样的结构使得在双向链表中，可以从头节点或尾节点开始遍历，也可以在常量时间内对节点进行前后移动。

链表的优点包括：

- 内存动态分配：相对于数组等静态数据结构，链表允许动态地分配内存。
- 插入和删除操作高效：在链表中，插入和删除节点的操作复杂度为 O(1)，只需调整节点之间的指针，不需要像数组那样移动元素。
- 大小可变：链表的长度可以动态增长或缩减，而不受固定容量的限制。

然而，链表也有一些缺点，包括：

- 随机访问性能差：链表中只能通过顺序访问或者从头/尾节点开始遍历，随机访问某个元素的效率较低。
- 额外的存储空间开销：每个节点都需要额外的指针空间，会增加存储开销。
- 不易逆向访问：单向链表只能从头到尾访问，而双向链表虽然可以逆向访问，但相比单向链表占用更多的存储空间。

链表在许多算法和数据结构中都有广泛的应用，例如堆栈、队列、哈希表的实现，以及在操作系统、数据库等领域中的应用。

陆、图

1、邻接矩阵

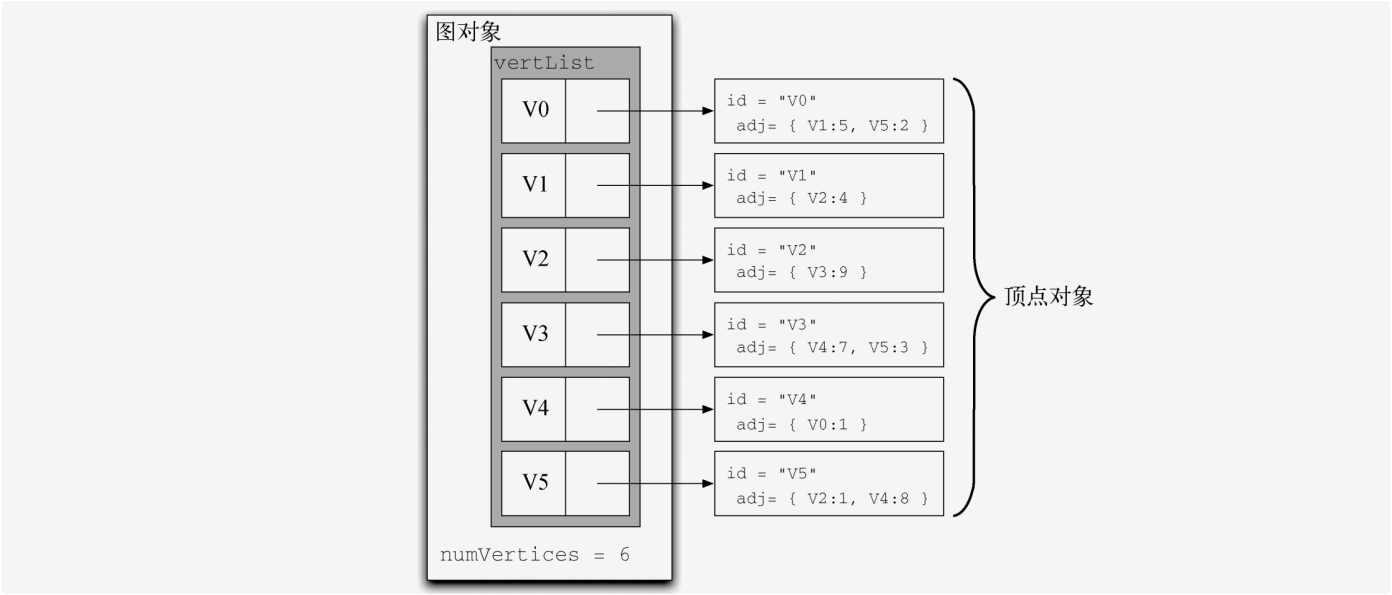
要实现图，最简单的方式就是使用二维矩阵。在矩阵实现中，每一行和每一列都表示图中的一个顶点。第 *v* 行和第 *w* 列交叉的格子中的值表示从顶点 *v* 到顶点 *w* 的边的权重。如果两个顶点被一条边连接起来，就称它们是相邻的。图 7-3 展示了图 7-2 对应的邻接矩阵。格子中的值表示从顶点 *v* 到顶点 *w* 的边的权重。

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

但是，图 7-3 中的绝大多数单元格是空的，我们称这种矩阵是“稀疏”的。对于存储稀疏数据来说，矩阵并不高效。事实上，要在 Python 中创建如图 7-3 所示的矩阵结构并不容易。

2、邻接表

为了实现稀疏连接的图，更高效的方式是使用邻接表。在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。在对 Vertex 类的实现中，我们使用字典（而不是列表），字典的键是顶点，值是权重。图 7-4展示了图 7-2 所对应的邻接表。



邻接表的优点是能够紧凑地表示稀疏图。此外，邻接表也有助于方便地找到与某一个顶点相连的其他所有顶点。

邻接表的实现：

Python 中，通过字典可以轻松地实现邻接表。我们要创建两个类：**Graph 类存储包含所有顶点的主列表，Vertex 类表示图中的每一个顶点。Vertex 使用字典 connectedTo 来记录与其相连的顶点，以及每一条边的权重。**代码清单7-1 展示了 Vertex 类的实现，其构造方法简单地初始化 id（通常是字符串），以及字典connectedTo。addNeighbor 方法添加从一个顶点到另一个的连接。getConnections 方法返回邻接表中的所有顶点，由 connectedTo 来表示。getWeight 方法返回从当前顶点到以参数传入的顶点之间的边的权重。

```
#Vertex类实现：
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):  ##nbr=Neighbor
        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]
```

Graph 类的实现如代码清单 7-2 所示，其中**包含一个将顶点名映射到顶点对象的字典**。在图7-4 中，该字典对象由灰色方块表示。Graph 类也提供了向图中添加顶点和连接不同顶点的方法。getVertices 方法返回图中所有顶点的名字。此外，我们还实现了_iter__方法，从而使遍历图中的所有顶点对象更加方便。总之，这两个方法使我们能够根据顶点名或者顶点对象本身遍历图中的所有顶点。

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

    def addVertex(self, key):
        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)

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

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

tips：`__iter__(self)` 是一个特殊方法（也称为魔术方法），用于定义对象的迭代行为。在这个方法中，通常会返回一个迭代器对象，使得该对象可以被 `for` 循环等迭代器工具使用。

在这段代码中，`__iter__(self)` 方法被定义在 `Graph` 类中，用于使 `Graph` 对象可以被迭代。具体来说，它返回了 `self.vertList.values()` 的迭代器，这样就可以迭代 `Graph` 对象中所有顶点的值（即 `Vertex` 对象）。

使用 `__iter__` 方法可以让你在对 `Graph` 对象进行迭代时直接使用 `for` 循环，例如：

```
graph = Graph()
# 添加顶点和边...

for vertex in graph:
    print(vertex)  # 这里的 vertex 是 Graph 中的每个顶点对象
```

这样做使得代码更加清晰和易读。

3、最小生成树（Prim，Kruskal）

（一）Prim算法：

Prim算法生成的最小生成树（MST）不一定是一条直线。实际上，最小生成树是覆盖图中所有顶点的无环子图，使得边的权重总和最小。

Prim算法的工作原理如下：

1. **初始化**：从任意一个顶点开始，将其标记为已访问，并将其所有相邻的边加入候选边集合。
2. **选择最小边**：从候选边集合中选择权重最小的边，并将该边的另一个端点标记为已访问。
3. **更新候选边集合**：将新访问顶点的所有未访问相邻边加入候选边集合。
4. **重复步骤2和3**，直到所有顶点都被访问。

Prim算法确保每次选择的边都是当前候选边中权重最小的，因此生成的最小生成树的边权重总和最小。但是，这并不意味着生成的树是一条直线，而是一个可能包含多个分支和连接的树形结构。

下面是一个Prim算法的Python实现示例：

```
import heapq

def prim(graph, start):
    mst = []  # 最小生成树的边
    visited = set([start])  # 已访问顶点集合
```

```
    edges = [(cost, start, to) for to, cost in graph[start]]  # 候选边集合
    heapq.heapify(edges)  # 构建最小堆

    while edges:
        cost, frm, to = heapq.heappop(edges)  # 选择权重最小的边
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost))
            for to_next, cost_next in graph[to]:
                if to_next not in visited:
                    heapq.heappush(edges, (cost_next, to, to_next))

    return mst

# 示例图，表示为邻接表
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('A', 1), ('C', 1), ('D', 3)],
    'C': [('A', 3), ('B', 1), ('D', 1)],
    'D': [('B', 3), ('C', 1)]
}

# 从顶点 'A' 开始构建最小生成树
mst = prim(graph, 'A')
print("最小生成树的边:", mst)
```

在这个例子中，图包含四个顶点（A, B, C, D）和一些带权重的边。运行Prim算法将生成一棵最小生成树，并输出其中的边。

综上所述，Prim算法生成的最小生成树并不一定是一条直线，而是一个树形结构，具体形状取决于输入图的结构和边的权重。

（二）Kruskal算法：

Kruskal算法是一种用于求解最小生成树（Minimum Spanning Tree, MST）的贪心算法。最小生成树是连通图中连接所有顶点且总权重最小的无环子图。Kruskal算法的核心思想是：按照边权重从小到大的顺序逐步将边添加到生成树中，但要确保不会形成环。

Kruskal算法的步骤

1. **排序**：将图中的所有边按照权重从小到大排序。
2. **初始化**：创建一个空的最小生成树（MST）。
3. **合并**：按照权重顺序依次检查每条边，如果边连接的两个顶点不在同一个连通分量中，则将该边加入到MST中，并合并这两个顶点所在的连通分量。
4. **重复**：重复步骤3，直到MST包含 `|V| - 1` 条边（其中 `|V|` 是图中的顶点数）。

使用并查集（Union-Find）

为了高效地管理和合并连通分量，Kruskal算法通常使用并查集（Union-Find）数据结构。

假设我们有一个无向图，图中有 `n` 个顶点和 `m` 条边。每条边用三个数表示：两个顶点和边的权重。可以通过以下步骤实现Kruskal算法。

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])  # 按照边权重从小到大排序
    uf = UnionFind(n)
    mst_cost = 0
    mst_edges = []

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst_cost += weight
            mst_edges.append((u, v, weight))

    return mst_cost, mst_edges

# 输入顶点数和边数
n, m = map(int, input().split())
edges = []

# 输入每条边的信息
for _ in range(m):
    u, v, weight = map(int, input().split())
    edges.append((u, v, weight))

mst_cost, mst_edges = kruskal(n, edges)

print(f"Minimum Spanning Tree cost: {mst_cost}")
print("Edges in the MST:")
```

```
for u, v, weight in mst_edges:
    print(f"{u} - {v}: {weight}")
```

输入示例

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

输出示例

```
Minimum Spanning Tree cost: 19
Edges in the MST:
2 - 3: 4
0 - 3: 5
0 - 1: 10
```

4、并查集

路径压缩（Path Compression）是并查集（Union-Find）算法中的一种优化技术，用于加快查找操作的效率。在并查集中，每个元素都有一个指向其父节点的指针，通过路径压缩，可以有效地降低查找操作的时间复杂度，使其几乎接近常数时间。

并查集的基本操作

- **Find**：查找元素所属的集合（找到该集合的代表元素或根节点）。
- **Union**：合并两个集合（将其中一个集合的根节点指向另一个集合的根节点）。

路径压缩的原理

路径压缩的基本思想是：在执行查找操作时，将沿途所有节点直接连接到根节点上。这样可以大大减少树的高度，从而加快以后的查找操作。

路径压缩的实现

在路径压缩的实现中，每次调用 `find` 方法时，不仅返回根节点，还会把沿途的所有节点直接连接到根节点。这可以通过递归或者迭代的方法实现。

以下是带有路径压缩的并查集实现代码：

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [1] * size

    def find(self, p):
        if self.parent[p] != p:
```

```
            self.parent[p] = self.find(self.parent[p])  # 路径压缩
        return self.parent[p]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP != rootQ:
            # 按秩合并
            if self.rank[rootP] > self.rank[rootQ]:
                self.parent[rootQ] = rootP
            elif self.rank[rootP] < self.rank[rootQ]:
                self.parent[rootP] = rootQ
            else:
                self.parent[rootQ] = rootP
                self.rank[rootP] += 1

# 示例
uf = UnionFind(10)
uf.union(1, 2)
uf.union(2, 3)
print(uf.find(3))  # 输出1，经过路径压缩后，节点3直接连接到根节点1
print(uf.parent)   # 查看路径压缩后的父节点数组
```

详细解释

1. **初始化**：
   - `self.parent` 初始化为每个元素指向自己。
   - `self.rank` 用于记录每个集合树的高度。
2. **Find 操作**：
   - 通过递归调用 `find`，最终找到根节点。
   - 在递归返回时，将当前节点的父节点直接更新为根节点，实现路径压缩。
3. **Union 操作**：
   - 首先找到两个元素的根节点。
   - 根据 `rank`（树的高度）决定哪个根节点应该作为另一个根节点的父节点。
   - 如果两个根节点的 `rank` 相同，则选择其中一个作为根节点，并将其 `rank` 加1。

路径压缩的优点

路径压缩显著减少了查找操作的时间复杂度。在最坏情况下，原始并查集的查找操作时间复杂度为 O(n)，但经过路径压缩后，时间复杂度可以降低到接近于 O(1) 的摊还复杂度。这意味着在大量操作下，平均每次操作的时间非常接近于常数时间，使得并查集在实际应用中非常高效。

结论

路径压缩是并查集中的一种重要优化技术，通过在查找操作中将沿途的节点直接连接到根节点，有效减少了树的高度，极大地提升了查找操作的效率。这种优化技术在各种需要处理动态连通性问题的算法和数据结构中都有广泛的应用。

通过图示来帮助你理解按秩合并（Union by Rank）的过程。按秩合并是一种优化策略，用于减少并查集的树的高度，从而提高效率。秩（rank）表示树的高度或者近似高度。

初始状态

假设我们有以下元素和它们各自的集合（每个元素开始时都是单独的集合）：

```
0   1   2   3   4   5   6
```

每个集合的秩（初始为 0）如下：

```
秩: 0 0 0 0 0 0 0
```

过程图示

第一步：合并 (0, 1)

```
0 - 1   2   3   4   5   6
```

0 和 1 合并后，由于它们的秩相同，所以任选一个作为根节点，并将秩加 1。这里我们选择 0 作为根节点：

```
0
|
1

秩: 1 0 0 0 0 0 0
```

第二步：合并 (2, 3)

```
0 - 1   2 - 3   4   5   6
```

同样，2 和 3 的秩相同，任选一个作为根节点，并将秩加 1。这里我们选择 2 作为根节点：

```
2
|
3

秩: 1 0 1 0 0 0 0
```

第三步：合并 (0, 2)

```
    0         2
   / \       / \
  1   2     3   0 - 1
```

此时，0 和 2 的秩相同，任选一个作为根节点，并将秩加 1。这里我们选择 0 作为根节点：

```
   0
  / \
 1   2
     |
     3

秩: 2 0 1 0 0 0 0
```

第四步：合并 (4, 5)

```
   0         4 - 5
  / \       / \
 1   2     4   5
     |
     3

秩: 2 0 1 0 1 0 0
```

同样，4 和 5 的秩相同，任选一个作为根节点，并将秩加 1。这里我们选择 4 作为根节点：

```
   0
  / \
 1   2
     |
     3

4
|
5

秩: 2 0 1 0 1 0 0
```

第五步：合并 (0, 4)

```
   0         4
  / \       / \
```

```
 1   2    5   0 - 1
     |
     3
         2
         |
         3

秩: 2 0 1 0 1 0 0
```

0 的秩为 2，4 的秩为 1，由于 0 的秩大，所以将 4 挂到 0 上：

```
    0
   /|\
  1 2 4
     | |
     3 5

秩: 2 0 1 0 1 0 0
```

最终状态

```
    0
   /|\
  1 2 4
     | |
     3 5

秩: 2 0 1 0 1 0 0
```

解释

- **初始状态**：每个元素都是一个单独的集合，秩都为 0。
- **合并相同秩的集合**：任选一个作为根节点，并将秩加 1。
- **合并不同秩的集合**：将秩小的集合挂到秩大的集合上，根节点的秩保持不变。

通过按秩合并，树的高度被尽量保持在较低水平，从而减少查找操作的时间复杂度。

下面是一个通过路径压缩优化并查集查找操作的示意图：

初始状态

假设我们有一个并查集，其中元素的初始状态如下：

```
0 - 1 - 2 - 3 - 4
```

这表示 0 是 1 的父节点，1 是 2 的父节点，2 是 3 的父节点，3 是 4 的父节点。

执行 Find 操作前

我们需要查找元素 4 的根节点：

```
Find(4)
```

在没有路径压缩的情况下，我们会按照以下步骤查找：

1. 找 4 的父节点 3
2. 找 3 的父节点 2
3. 找 2 的父节点 1
4. 找 1 的父节点 0

所以，4 的根节点是 0。

执行 Find 操作并进行路径压缩

在执行 `find` 操作时，我们会在查找的过程中，将沿途的节点直接连接到根节点上。这可以通过递归实现：

1. 找 4 的父节点 3，然后将 4 的父节点更新为 0
2. 找 3 的父节点 2，然后将 3 的父节点更新为 0
3. 找 2 的父节点 1，然后将 2 的父节点更新为 0
4. 找 1 的父节点 0

最终所有节点都会直接连接到根节点 0 上。

路径压缩后的状态

```
   0
 / | | \
1  2  3  4
```

此时，所有节点都直接连接到根节点 0。

图示过程

下面是图示的详细过程：

1. **初始状态**：

```
0 - 1 - 2 - 3 - 4
```

2. **Find(4) 后路径压缩的中间状态**：

```
0 - 1 - 2 - 3 - 4
            ^
            |
            |
            0
```

3. **Find(3) 后路径压缩的中间状态**:

```
0 - 1 - 2 - 3 - 4
        ^
        |
        |
        0
            0
            |
            3
```

4. **Find(2) 后路径压缩的中间状态**:

```
0 - 1 - 2 - 3 - 4
    ^
    |
    |
    0
        0
        |
        2
            0
            |
            3
                0
                |
                4
```

5. **Find(1) 后路径压缩的最终状态**:

```
   0
 / | | \
1  2  3  4
```

解释

通过路径压缩，所有节点都直接连接到根节点 `0` 上。这大大减少了树的高度，未来的查找操作将更加高效。每次 `find` 操作都会压缩路径，使得所有经过的节点直接连接到根节点，从而降低后续查找的时间复杂度。

5、有向图 拓扑排序

拓扑排序是一种将有向无环图（DAG，Directed Acyclic Graph）的顶点进行线性排序的方法，使得对于每一条有向边 (u \rightarrow v)，顶点 (u) 都排在顶点 (v) 之前。拓扑排序在许多应用中都很有用，例如任务调度、课程安排等。

拓扑排序的基本概念

- **有向无环图（DAG）**：一个有向图，其中不存在从某个顶点出发经过若干有向边又回到该顶点的路径。
- **拓扑排序**：对有向无环图的一种线性排序，使得对于图中的每一条有向边 (u \rightarrow v)，顶点 (u) 都排在顶点 (v) 之前。

拓扑排序的算法

有两种常用的拓扑排序算法：Kahn's 算法和基于深度优先搜索（DFS）的算法。

1. Kahn's 算法

Kahn's 算法基于入度的概念，通过反复删除入度为0的顶点来实现拓扑排序。

步骤：

1. 计算每个顶点的入度。
2. 初始化一个队列，将所有入度为0的顶点入队。
3. 反复执行以下操作，直到队列为空：
    - 从队列中取出一个顶点，将其加入拓扑排序结果。
    - 对其所有邻接顶点，入度减1。如果某个邻接顶点的入度变为0，将其加入队列。
4. 如果所有顶点都被处理，则拓扑排序成功；否则，有环存在。

代码实现：

```
from collections import deque

def kahn_topological_sort(graph, V):
    in_degree = [0] * V
    for u in range(V):
        for v in graph[u]:
            in_degree[v] += 1

    queue = deque()
    for i in range(V):
        if in_degree[i] == 0:
            queue.append(i)

    topo_sort = []
    count = 0
```

```
    while queue:
        u = queue.popleft()
        topo_sort.append(u)
        count += 1
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    if count != V:
        print("Graph has a cycle")
    else:
        print("Topological Sort: ", topo_sort)

# Example usage:
graph = [[1, 2], [3], [3], []]
V = 4
kahn_topological_sort(graph, V)
```

```
def isCyclicKahn(graph,V):
    in_degree=[0]*V
    for u in range(V):
        for v in graph[u]:
            in_degree[v]+=1
    queue=[]
    for i in range(V):
        if in_degree[i]==0:
            queue.append(i)
    count=0
    while queue:
        u=queue.pop(0)
        count+=1
        for v in graph[u]:
            in_degree[v]-=1
            if in_degree[v]==0:
                queue.append(v)
    return count!=V

T=int(input())
ans=[]
for _ in range(T):
    N, M=map(int,input().split())
    graph=[[] for _ in range(N)]
    for _ in range(M):
        u,v=map(int,input().split())
        graph[u-1].append(v-1)
    ans.append("Yes" if isCyclicKahn(graph,N) else "No")

for i in ans:
    print(i)
```

2. 基于深度优先搜索（DFS）的算法

DFS 算法基于递归的方式，通过回溯实现拓扑排序。

步骤：

1. 初始化一个布尔数组 `visited`，标记每个顶点是否已被访问。
2. 初始化一个栈，用于存储顶点的拓扑排序结果。
3. 对每个未访问的顶点，调用递归函数 `dfs`：
    - 标记当前顶点为已访问。
    - 递归访问所有邻接顶点。
    - 递归结束后，将当前顶点压入栈中。
4. 最终从栈中依次弹出顶点，即为拓扑排序结果。

代码实现：

```
def dfs(v, visited, stack, graph):
    visited[v] = True
    for neighbor in graph[v]:
        if not visited[neighbor]:
            dfs(neighbor, visited, stack, graph)
    stack.append(v)

def dfs_topological_sort(graph, V):
    visited = [False] * V
    stack = []

    for i in range(V):
        if not visited[i]:
            dfs(i, visited, stack, graph)

    topo_sort = []
    while stack:
        topo_sort.append(stack.pop())

    print("Topological Sort: ", topo_sort)

# Example usage:
graph = [[1, 2], [3], [3], []]
V = 4
dfs_topological_sort(graph, V)
```

如何判断有向图是否有环

如前所述，如果图中存在环，则无法生成拓扑排序。具体步骤如下：

- **Kahn's 算法**：如果最终处理的顶点数不等于图中的顶点数，则图中存在环。
- **DFS 算法**：在递归过程中，如果遇到一个已经在递归调用栈中的顶点，则图中存在环。

柒、其他

1、单调栈

单调栈，就是一个栈，不过栈内元素保证单调性。即，栈内元素要么从小到大，要么从大到小。
而单调栈维护的就是一个数前/后第一个大于/小于他的数。
例题就是一个求每个数后第一个大于他的数。

那么重点来了：怎么做！面对这样的数据，不好下手。那么我们把她转化一下：有n*n*个人，每个人向右看，求她看到的第一个人。
看图：

通过观察，我们会发现，在她后面的，比她矮的，一定会被她遮住。那么，这个点就没用了。而根据现实生活和刚才的推断，我们看到的人肯定是一个比一个高的，而没看到的，留着也没用，直接扔了QwQ。那么，这就是符合单调性的。再看，那些没用的人什么时候扔掉？当然是遇到比她高的人了。那么就可以一个一个地走掉，而且肯定是在已经判断过的人的前面（中间和后面的在之前就走掉了），所以就直接从前面弹出。咦？这不就像一个栈吗？没错，这就是单调栈的实现方法。

再归纳一下：

- 从后往前扫
- 对于每个点：
  - 弹出栈顶比她小的元素
  - 此时栈顶就是答案
  - 加入这个元素

由于是从前往后输出，还要把答案放到一个数组里。

代码：

```
#include<cstdio>
#include<stack>
using namespace std;
int n,a[3000005],f[3000005];//a是需要判断的数组（即输入的数组），f是存储答案的数组
stack<int>s;//模拟用的栈
int main()
{
  scanf("%d",&n);
  for(int i=1;i<=n;i++) scanf("%d",&a[i]);
  for(int i=n;i>=1;i--)
  {
    while(!s.empty()&&a[s.top()]<=a[i]) s.pop();//弹出栈顶比当前数小的
    f[i]=s.empty()?0:s.top();//存储答案，由于没有比她大的要输出0，所以加了个三目运算
    s.push(i);//压入当前元素
  }
  for(int i=1;i<=n;i++) printf("%d ",f[i]);//输出
  return 0;
}
```