5，二叉树和树-3

- ❖ 树形结构和图示
- ❖ 树、树林和二叉树：定义，相关概念和性质
- ❖ 二叉树的 list 实现
- ❖ 二叉树应用：表达式和计算
- ❖ 二叉树应用：优先队列，离散事件模拟
- ❖ 二叉树遍历：先序、中序和后序，层次序
- ❖ 深度优先遍历算法：递归和非递归描述
- ❖ 二叉树应用：哈夫曼树和哈夫曼编码
- ❖ 树和二叉树

二叉树遍历算法：递归形式

- 上次课讨论了遍历二叉树的深度和宽度优先算法

  针对链接二叉树结点的定义，可以写出按这些序遍历二叉树的函数

  对深度优先变量，用递归形式定义的函数非常简单

  非递归定义的函数也有意义，后面讨论

- 按先根序遍历二叉树的递归函数：

```
def preorder(t, proc): # proc 用于提供遍历中的结点数据操作
    if t is None: return
    proc(t.data)
    preorder(t.left)
    preorder(t.right)
```

- 按中根序和后根序遍历二叉树的函数与此类似，只是函数里几个操作的排列顺序不同。自己看代码文件

  如需要，可以加断言语句 **assert(isinstence(t, BiTNode))** 检查参数

二叉树遍历算法

- 作为遍历的一个实例，定义一个输出二叉树的函数

  这里用带括号的前缀形式输出，空子树输出符号 "^"

```
def print_BiTNodes(t):
    if t is None:
        print("^", end="") # 不输出将无法区分左右子树
        return
    print("(" + str(t.data), end="")
    print_BiTNodes(t.left)
    print_BiTNodes(t.right)
    print(")", end="")
```

- 例如：

```
t = BiTNode(1, BiTNode(2, None, None), BiTNode(3, None, None))
print_BiTNodes(t)
```

  输出：(1(2^^)(3^^))

自我练习：写一个读入这种输出形式构造二叉树的函数，假设结点数据是字符串或数

二叉树遍历：宽度优先算法

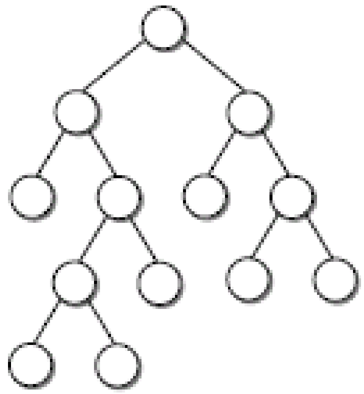
- 采用宽度优先方式遍历二叉树的函数，需要用一个队列：

```
from SQueue import *

def levelorder(t, proc):
    qu = SQueue()
    qu.enqueue(t)
    while not qu.is_empty():
        n = qu.dequeue()
        if t is None: continue # 弹出的空树直接跳过
        qu.enqueue(t.left)
        qu.enqueue(t.right)
        proc(t.data)
```

- 下面考虑非递归定义的深度优先遍历算法，基于两种考虑
  - ❑ 帮助看清递归与非递归的关系，以及遍历的具体过程和性质
  - ❑ 算法本身有用，还可以看到分析问题和设计算法的一些情况

非递归的先根序遍历

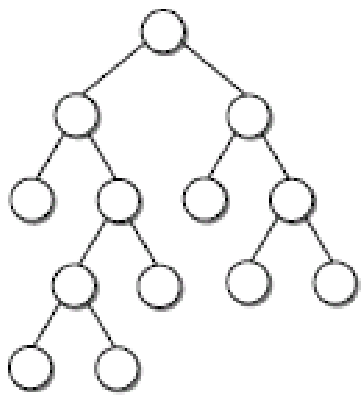


- 考虑先根序的非递归遍历算法

  需要用一个栈保存树尚未访问过部分的信息
- 考虑先根序遍历的一种方法。基本想法很简单
  - 先根序，访问遇到的结点并沿左枝下行

    尚未访问的右分支需要记录，将其入栈
  - 遇空树时回溯，取出栈中保存的右分支，像一棵树一样遍历它
- 算法还有一些细节，主要是循环的控制
  - 循环条件：“当前树非空（这棵树需要遍历）或者栈不空（还存在整个树的未遍历部分）”，这时就应该继续循环
  - 在向下检查左分支时把经过结点的右分支入栈（也要用一个循环）
  - 弹出栈中元素（一个右子树）回溯，要做的也是遍历一棵二叉树

非递归的先根序遍历



- 定义出来的函数很简单：

```
def preorder_nonrec(t, proc):
    s = SStack()
    while t is not None or not s.is_empty():
        while t is not None: # 沿左分支下行
            proc(t.data)     # 先根序先处理根数据
            s.push(t.right) # 右分支入栈
            t = t.left
        t = s.pop()  # 左分支到头了，回溯
```

- 如果 **tree** 的值是一棵结点保存着可打印数据的二叉树，下面语句将逐项输出其内容，空格分隔

  preorder_nonrec(tree, lambda x:print(x, end=" "))

  注意，这里用 **lambda** 表达式，以便定制一次输出后不换行
- 非递归的中根序算法与先根序类似，代码文件里有，请自己分析

非递归的先根序遍历

- 非递归算法的一个意义就是把算法过程完整地暴露出来，便于分析

  现在考虑非递归的先根序遍历算法的复杂性
- 时间复杂性：
  - 每个结点访问一次，一部分子树（所有右子树）压入弹出栈一次
  - **proc(t.data)** 的复杂性与树的大小无关，整个遍历是 **O(n)** 时间
- 空间复杂性：
  - 关键的因素是遍历中栈曾经达到的最大深度（栈中元素个数），而栈的最大深度由被遍历的二叉树的高度决定
  - 易见，在最坏情况下，算法的空间复杂性是 **O(n)**
  - **n** 个结点的二叉树有很多，可以枚举出所有可能的二叉树。算出它们的平均高度。结论：**n** 个结点的二叉树的平均高度是 **O(log n)**，所以，先根序遍历的平均空间复杂性是 **O(log n)**
  - 如果修改实现，只将非空右子树进栈，有可能减少空间开销

非递归的先根序遍历

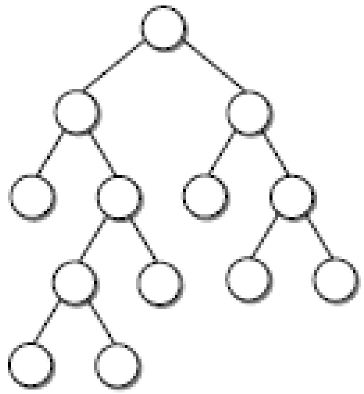
- 用 **Python** 写程序，考虑遍历数据汇集结构时，总应该想到迭代器
- 例如，下面是生成单链表中数据元素的生成器：

```
def list_iter(lst):
    p = lst.head
    when p in not None:
        yield p.elem
        p = p.next
```

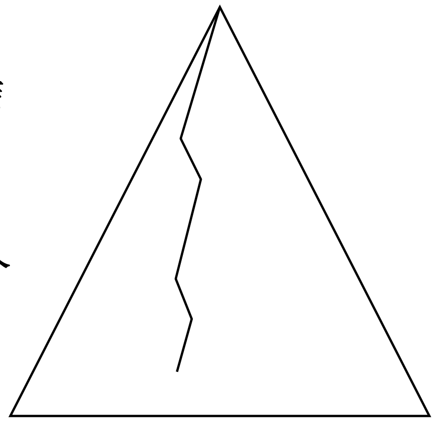
- 由前面非递归先根序遍历函数改造而得的二叉树迭代器：

```
def preorder_iter(t):
    s = SStack() # 迭代器用栈保存有用信息
    while t is not None or not s.is_empty():
        while t is not None: # 沿左分支下行
            yield t.data
            s.push(t.right); t = t.left
        t = s.pop()   # 回溯
```

非递归的后根序遍历算法



- 每个问题都可能有多种算法，下面介绍一种后根序遍历算法（是能找到的最简短的算法）
- 在实现遍历的循环中维持一种不变关系：
  - 栈中结点是对二叉树的划分，左边是已遍历过的部分，右边是尚未遍历的部分
  - 栈中每个结点的父结点就是位于它下面那个结点，当前结点的父结点是栈顶
  - 根据本结点是其父结点的左子结点或右子结点，可以决定下一步怎么做
  - 在需要从当前结点回溯之前访问这个结点
- 主要技术是一个“下行循环”，目标是找到下一个应访问结点，内层循环结束时该结点在栈顶
- 这个算法稍微复杂一点，大家下去再仔细看看



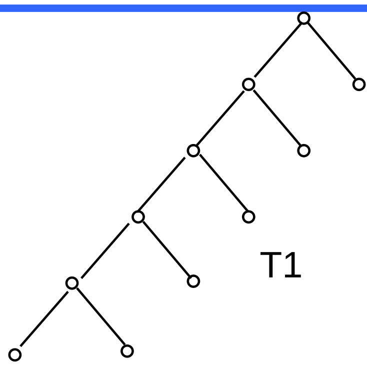
非递归的后根序遍历算法

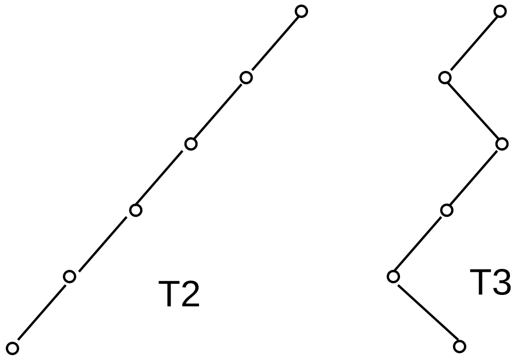
```
def postorder_nonrec(t, proc):
    s = SStack()
    while t is not None or not s.is_empty():
        while t is not None:  # 下行循环，直到栈顶的两子树空
            s.push(t)
            t = t.left if t.left is not None else t.right
            # 注意这个条件表达式的意思，能左就左/否则就右
        t = s.pop()  # 栈顶是应访问结点
        proc(t.data)
        if not s.is_empty() and s.top().left == t:
            t = s.top().right # 栈不空且当前结点是栈顶的左子结点
        else:
            t = None   # 没有右子树或右子树遍历完毕，强迫退栈
```

- 注意：1）内层循环在找到一个最下最左结点后终止；2）如被处理结点是其父的左子结点，直接转到其右兄弟结点并继续；3）如被处理结点是其父的右子结点，设 t 为 None 迫使下次循环弹出并处理其父结点

递归和非递归的遍历

- 对递归遍历算法，无论用那种顺序（先根序，中根序或后根序），对深度为 **n** 的畸形树，栈深度都会到达 **n** 或 **n-1**
- 非递归遍历算法的情况可能不同
  - 后根序遍历中必须把未遍历的整条路径存入栈，对深度为 **n** 的树，无论其具体结构，栈深度也会达到 **n**
  - 对先根序遍历，如果只入栈非空右子树，遍历单枝树（如右图 **T2** 和 **T3**）时栈深度都不会深于 **1**，最坏情况是右面树 **T1** 的情况，栈深度可能达到大约 **n/2**
  - 中根序的情况与先根序类似





二叉树数据结构

- 直接用结点构造的二叉树具有递归的结构，可以很方便地递归处理。但这样的“二叉树结构”也有不统一的地方

  用 **None** 表示空树，但 **None** 不具有 **BiTNode** 类型
- 解决问题的方法是定义一个二叉树类型，以结点树作为其内部表示

  与链接表的情况类似，那里也是以一个结点表作为内部表示
- 定义二叉树类：

```
class BiTree:
    def __init__(self):
        self._root = None

    def is_empty(self):
        return self._root == None

    def set_root(self, rootnode):
        self._root = rootnode
```

二叉树数据结构

```
def set_left(self, leftchild):
    self._root.left = leftchild

def set_right(self, rightchild):
    self._root.right = rightchild

def root(self): return self._root
def leftchild(self): return self._root.left
def rightchild(self): return self._root.right
```

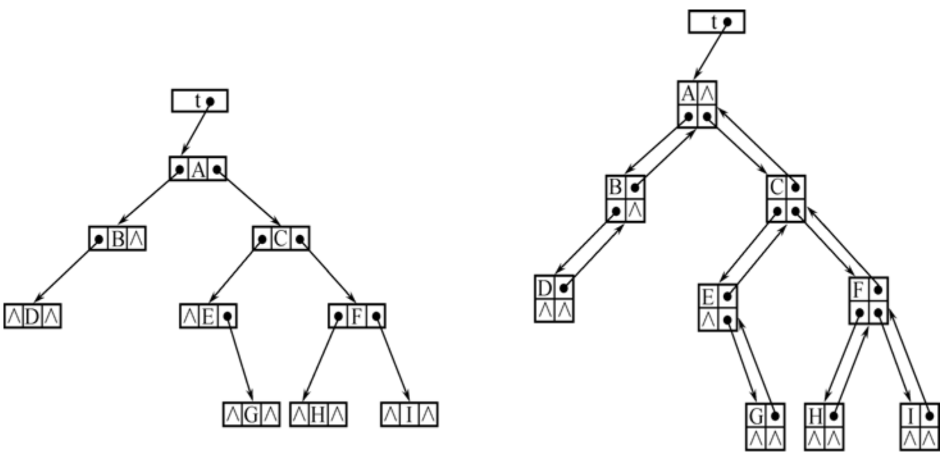
- 可以考虑定义一个或几个遍历迭代器，例如

```
def preorder_iter(self):
    t, s = self._root, SStack()
    while t is not None or not s.is_empty():
        while t is not None:
            s.push(t.right)
            yield t.data
            t = t.left
        t = s.pop()
```

二叉树数据结构

- 其他操作可以根据需要定义，也可以考虑以这个二叉树为基类定义派生的二叉树类。除遍历操作外，其他操作都是 **O(1)**
- 在这种表示上求父结点的操作较难实现，只能通过从根开始的遍历来完成，最坏时间代价是 **O(n)**

如果经常需要找父结点，可考虑增加一个父结点引用域



(a) 左－右指针表示　　(b) 三叉链表表示

思考题：

如何遍历包含父结点引用域的链接二叉树。能否不用栈？

哈夫曼树

- 哈夫曼树（**Huffman tree**）是一种二叉树，在信息领域有重要理论和实际价值。这里将其看作是二叉树的一种应用
- 考虑带权扩充二叉树的“外部路径长度”。这种二叉树的外部结点（叶结点）标有一个数值，称为该结点的权，表示与该叶有关的某种性质

扩充二叉树的外部路径长度：
（外部结点的路径长度之和）

$$E = \sum_{i=1}^{m} l_i$$

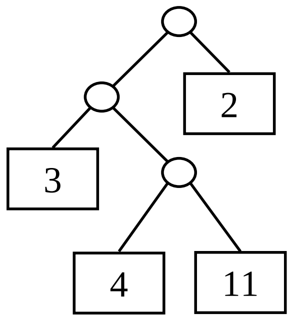
l_i：从根到外部结点 i 的路径长度

m：外部结点个数

带权外部路径长度：

$$WPL = \sum_{i=1}^{m} w_i l_i$$

w_i：外部结点 i 的权，



WPL = 53

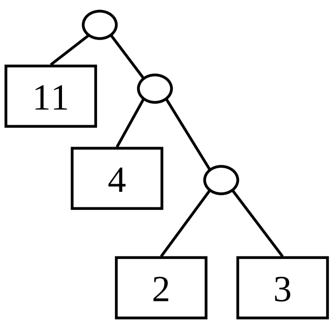
哈夫曼树

- 定义：有实数集 **W = {w_1, w_2, … , w_m}**，**T** 是一棵扩充二叉树

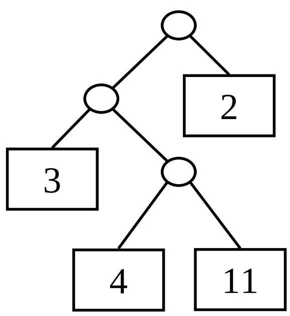
  包含 **m** 个分别以 w_i（**i = 1, 2, …, m**）为权的外部结点，且其其带权外部路径长度 **WPL** 达到最小

  则称 **T** 为数据集 **W** 的最优二叉树或哈夫曼树
- 以同一集实数为外部结点权的二叉树，**WPL** 可能不同。例：

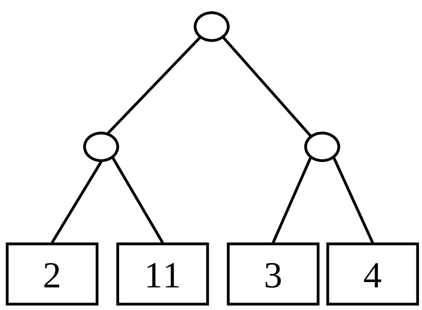
例：以 **{2, 3, 4, 11}** 为外部结点权的几棵扩充二叉树：



WPL = 34



WPL = 53



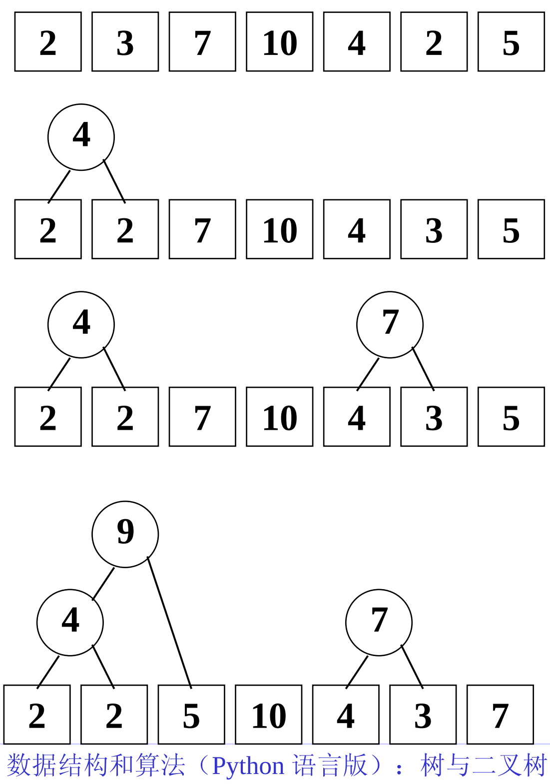
WPL = 40

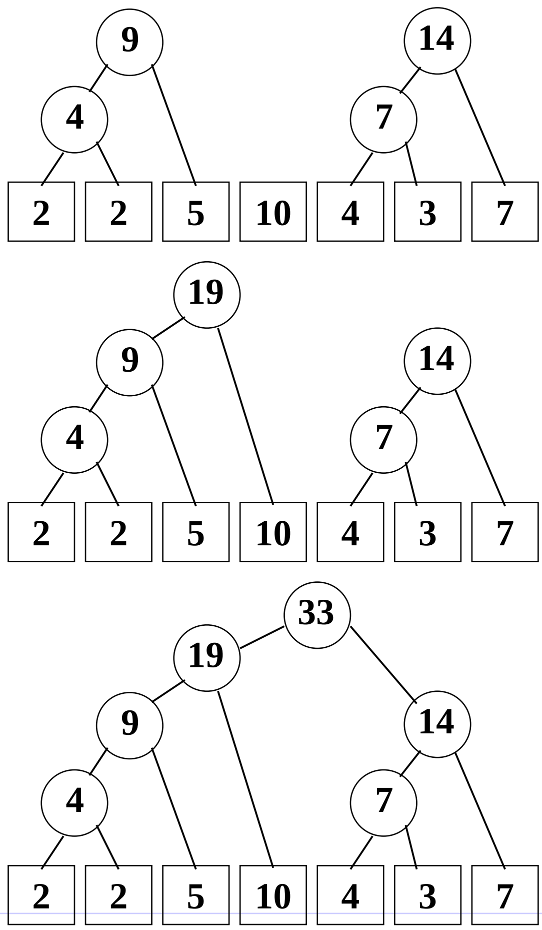
哈夫曼算法

- 给定一组实数，对应的哈夫曼树可以通过哈夫曼算法得到
- 哈夫曼算法：输入实数集 $W = \{w_1, w_2, w_3, \ldots, w_m\}$
  - 构造中维护包含 m 棵二叉树的集合 F，开始时 $F = \{T_1, T_2, \ldots, T_m\}$，其中 T_i 是只包含权为 w_i 的根结点的单点二叉树
  - 算法重复下面两个步骤，直到 F 中只含一棵树为止
    1. 从 F 中选取两棵权最小的树作为左右子树，构造一棵新二叉树，设置其根结点权值为两棵子树的根结点的权值之和
    2. 从 F 删除所选的两棵树，把新构造的二叉树加入F
  - 步骤 2 做一次 F 里的二叉树减少一棵，这保证了本算法必定结束
- 要证明这一算法做出的是哈夫曼树不太容易（请自己考虑）。只能用结构归纳法，关键是归纳证明一步怎么论述清楚
- 注意：集合 W 上的哈夫曼树不唯一。如果 T 是 W 上的哈夫曼树，交换其左右子树，得到的仍是 W 上的哈夫曼树

哈夫曼算法过程的示例

设有权组 {2, 3, 7, 10, 4, 2, 5}





哈夫曼算法：实现

- 实现哈夫曼算法，需要保存一组二叉树
  - 每棵树根记录的数据是它的权
  - 算法中需要不断使用权最小的两棵二叉树，构造新的二叉树
- 大家从这里看到了什么？

  应该用什么结构保存这组二叉树？
- 最佳选择是用一个优先队列存放这组二叉树
  - 队列里的二叉树按权值存入，要求先取出最小元素
  - 开始时建一组单结点二叉树，以权值作为优先码存入优先队列
- 反复做下面两件事，直至优先队列里只有一个元素：
  - 取出两个权最小的元素
  - 基于它们构造一棵新二叉树，权值取两棵子树的权值之和，并将构造的二叉树存入优先队列

哈夫曼算法：实现

- 为实现 Huffman 算法，基于已有类派生两个类：

```
class HTNode(BiTNode):
    def __lt__(self, othernode): return self.data < othernode.data

class HuffmanPrioQ(PrioQueue):
    def number(self): return self.num
```

- 算法的实现直截了当：

```
def HuffmanTree(weights):
    trees = HuffmanPrioQ()
    for w in weights:
        trees.enqueue(HTNode(w, None, None))
    while trees.number() > 1:
        t1 = trees.dequeue(); t2 = trees.dequeue()
        x = t1.data + t2.data
        trees.enqueue(HTNode(x, t1, t2))
    return trees.dequeue()
```

哈夫曼编码

- 哈夫曼编码在信息理论里有重要的理论意义，也有实际价值
- 问题：给定基本数据集合：

$$D = [d_1, d_2, \ldots , d_n] \qquad W = [w_1, w_2, \ldots , w_n]$$

其中：**D** 是需要编码的字符集合，**W** 为 **D** 中各个字符在实际信息传输（或者存储）中出现的频率

要求为 **D** 设计一套二进制编码，使得：**1)** 用此编码存储/传输时的平均开销最小；**2)** 对任意不同的 d_i 和 d_j，d_i 编码不是 d_j 编码的前缀

- 第二个条件使解码时容易判断是否已得到一个字符的编码

因为任一字符的编码不是另一字符编码的前缀，看到了对应于一个字符的一段编码，就可以确定原文应该是这个字符

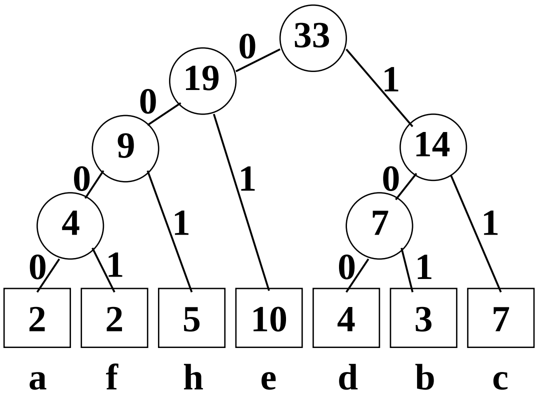
- 哈夫曼提出了一种解决这一问题的方法，即哈夫曼编码
- 注意：这里考虑的编码优化的问题，允许各字符的码长度不同。目前常用的 **ASCII** 码等采用等长编码，没考虑这种优化

哈夫曼编码

- 哈夫曼提出的方法就是通过构造哈夫曼树实现哈夫曼编码：
  - 以字符 $d_1, d_2, \ldots, d_n$ 作为外部结点的标注，把 $w_1, w_2, \ldots, w_n$ 作为这 **n** 个外部结点的权，基于它们构造一棵哈夫曼树
  - 在得到的哈夫曼树中，给所有从一个结点到其左子结点的边标上二进制数字 **0**；引向其右子结点的边标上 **1**
  - 以从根结点到一个叶结点的路径上的二进制数字序列，作为这个叶结点的字符的编码。这就是哈夫曼编码
- 可以证明：对任意的数据集合对 **(D, W)**，按上述方式得到的哈夫曼编码是其最优（最短）编码
- 哈夫曼编码在编码理论里有重要意义，是可能字符集（在确定的概率分布情况下）的最优编码
  - 信息专业后续课程“信息科学基础”和“数字信号处理”还会涉及
  - 不难利用前面的代码，实现相关功能

哈夫曼编码

- 假设有“字符:权值”组 **{a:2, b:3, c:7, d:4, e:10, f:2, h:5}**
  - 利用哈夫曼算法做出下面的哈夫曼树
  - 按前述方法标出相应的编码，就得到了哈夫曼编码



总结可得各字符的编码

a: 0000
b: 101
c: 11
d: 100
e: 01
f: 0001
h: 001

编码：
0010000100000010101100

对应字符：　h　a　d　a　b　e　d

树

- 现在转到树和树林，首先讨论树的基本操作。树的操作也没有统一的说法，可根据实际需要考虑。这里列出一些基本操作
  - 基于子树创建树（**k = 0** 时是树叶）：**Tree(data, t_1, t_2, …, t_k)**
  - 创建空树：**create_empty_tree()**
  - 判断空树：**is_empty()**
  - 取树根：**root()**
- 其他操作：
  - 取得树的父结点：**parent()**
  - 求树的度数：**degree()**
  - 树的最左子树：**first_child()**，第 **i** 棵子树：**ichild(i)**
  - 已知一棵子树找其下一兄弟结点：**next_sibling(t, c)**
  - 树遍历。也有多种不同方式，可基于上述操作实现。下面讨论

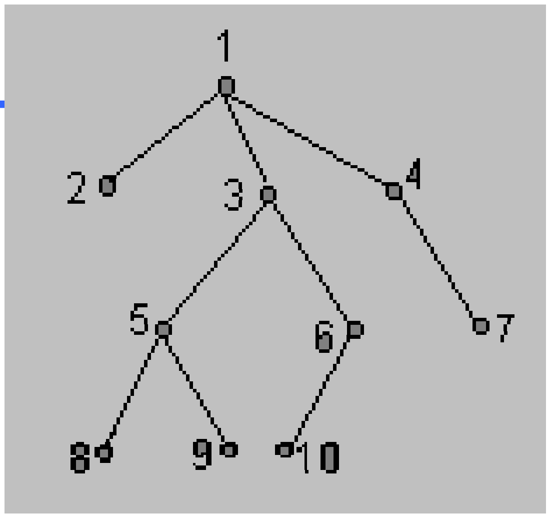
树的遍历

- 与二叉树一样，遍历就是访问树中所有结点且访问每个结点恰好一次的过程。遍历算法需要采用某种系统化的方式
  - 最基本的系统化遍历方法：按深度遍历和按宽度遍历
  - 按深度优先或宽度优先遍历方式访问结点，访问的顺序也就是深度优先搜索或宽度优先搜索中访问结点的顺序
- 在一般的状态空间搜索问题里，搜索过程中经历的状态（看作结点）和状态之间的联系（看作边）就形成了一棵“树”，称为“搜索树”。搜索过程就是按某种顺序“遍历”这棵树（虽然并没有显式地表示出来）
- 由于非空的树总有一个根结点，掌握了根结点也就掌握了以它为根的树，人们常把树 t 与其根结点统一看待
- 深度优先遍历也有多种方式，差别在于遍历中访问根结点信息的时刻

  先根序和后根序，是在访问所有子树之前或者之后访问根

  “中根序”的意义不明确，例如在访问第一棵子树之后访问根

树的遍历



- 先根序

  访问根结点，然后从左到右按先根序遍历各子树

  1, 2, 3, 5, 8, 9, 6, 10, 4, 7
- 后根序

  从左到右按后根序遍历各子树，然后访问根结点

  2, 8, 9, 5, 10, 6, 3, 7, 4, 1
- 层次序

  宽度优先，按树的层次，逐层访问树中结点

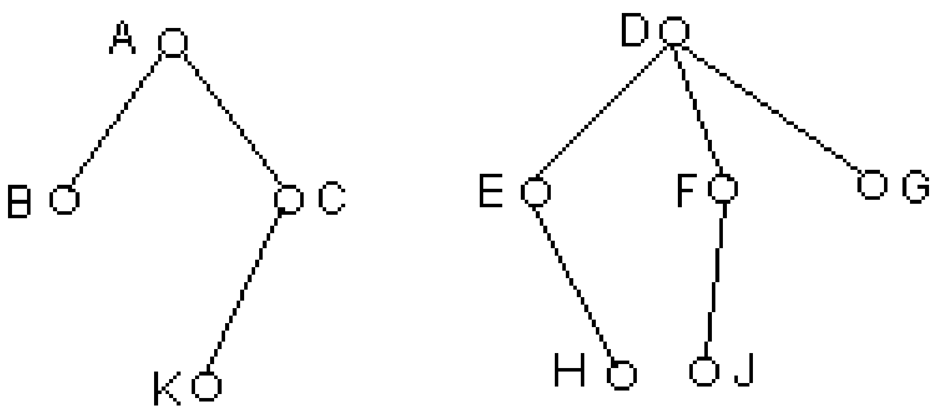
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10

抽象的树遍历算法很简单，这里不专门讨论了

在考虑树的具体实现时，需要具体考虑

树林的遍历

- 树林的遍历，就是逐个遍历其中各棵子树
- 示例
  1. 先根（**A, B, C, K, D, E, H, F, J, G)**
  2. 后根 **(B, K, C, A, H, E, J, F, G, D)**



树的表示

- 树结构比较复杂，可以考虑的表示方法很多。不同表示方法有各自不同的优点和缺点，应该根据实际情况和需要选择
- 常用的树表示方法有：
  - 子指针表示法，父指针表示法
  - 子表表示法
  - 长子-兄弟表示法
- 实际上，树林和二叉树之间有一种 **1-1** 对应关系。在这种对应下，
  - 树林可以用二叉树表示
  - 树可以用二叉树的一个子集表示
- 由于二叉树结构比较规范，容易实现，人们经常用二叉树来实现树

  下面先讨论这个问题，应该看作树的二叉树表示法

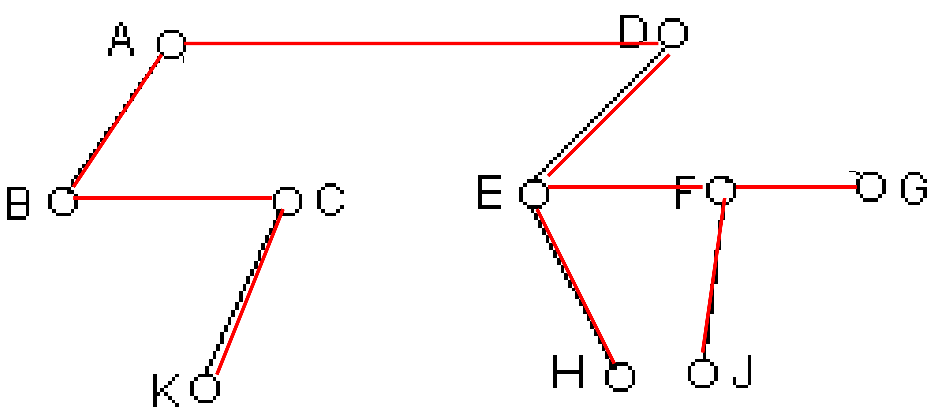
  所谓“长子-兄弟表示法”，实际上就是树的二叉树表示法

树林（树）与二叉树的对应（转换）

- 由于存在一一对应关系，树林（树）与二叉树之间可以相互转换

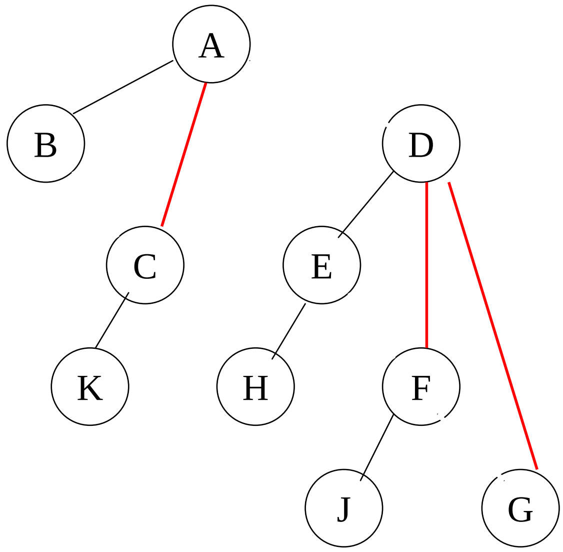
  二叉树的所有实现技术都可以作为树和树林的实现技术
- 从树、树林转换为二叉树的步骤：
  - 对于相邻的兄弟结点，从左到右，从前一子结点和后一子结点连一条边（对树林，相邻树的根之间也同样连一条边）；
  - 对每个非叶结点，只保留从它到其最左子结点的边，删去它到它的其它子女的边
- 这样，（现在的树里）每个非叶结点至多发出两条边：
  - 一条到它（在原树中）的第一个子结点，此边看作在二叉树里这个结点到其左子树根结点的边
  - 另一条到它在原来树中的下一个兄弟结点，此边看作在二叉树里该结点到其右子树根结点的边

树林到二叉树：例



二叉树转换到树林（树）

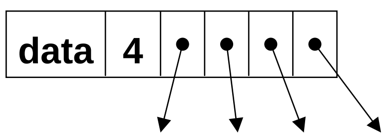
- 转换方法：
  - 如果某结点是其父结点的左子树，则将其向右的路径上的各个结点作为其父结点的顺序的各个子结点
  - 去掉原二叉树中各结点到其右子结点的连线



- 由于二叉树结点的形式比较规范和统一。树的最常见表示方式就是先转换到对应的二叉树，而后采用二叉树的表示技术
- 下面介绍其他常用的表示技术

树的表示：结点链接表示

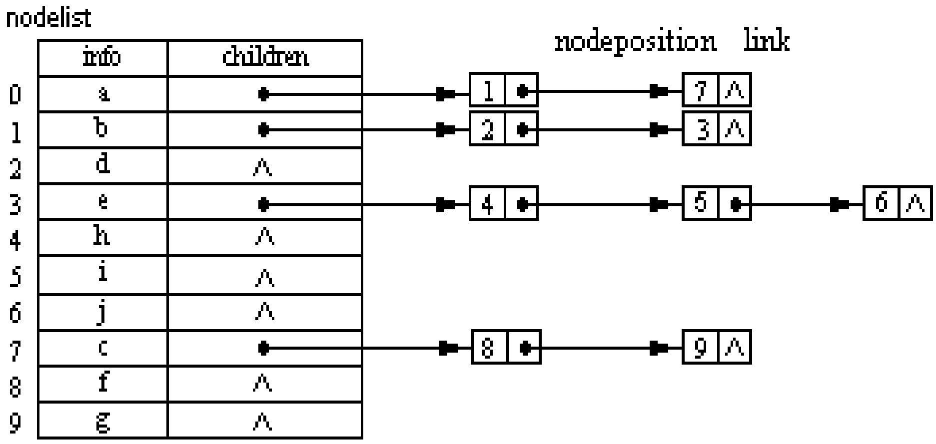
- 树的一种基本表示方法是结点链接表示
  - 一个树结点用一个存储块表示
  - 在这个存储块里记录树结点本身的信息，还要记录其子树个数
  - 在结点的存储块里保存到它的各子结点的引用
  - 根结点存储块代表整棵树
  - 如果需要，还可以在每个结点里增加一个父结点链接
- 这种表示方法很直接，缺点就是结点存储块的大小由结点的子结点个数决定，大小不一



- 为寻求比较统一的树实现方式，人们提出了一些办法

树的表示：子表表示法

- 树（树林）的一种表示是用一个结点数组（或者线性表），每个结点在数组中有一个位置（下标）
  - 每个结点关联一个子结点表，其中记录子结点的（数组）位置
  - 结点本身的数据存储在数组的相应位置
- 结点表可以用链接表（结点的大小相同）



树的表示

- 在实际算法和程序里，树的使用不如二叉树广泛，主要原因是
  - 树的结构不规整，结点的子结点个数可能不同，而且没有限制。在计算机里表示和处理都比较麻烦
  - 表示树结点的存储块大小可能不同，而且可能相差很大，存储管理工作比较麻烦，时间和空间开销较大（常量因子）
  - 需要在树结点里记录子结点的个数，维护这种记录
  - 如果树结构可以变动，动态地增减子结点（改变树的结构），也会带来更复杂的管理问题（包括存储管理）
- 在 **Python** 里可以用 **list** 或 **tuple** 表示树结点，

  让系统的底层支撑模块（包括存储管理模块）处理上述问题
- 实际中需要使用树结构时，人们常常利用树与二叉树的关系，把树转换为二叉树后存储和处理

  有些数学软件采用二叉树方式实现数学表达式（树结构）

总结

- 树形结构在计算中使用广泛，许多数据可以用树结构表示
  - 树形结构是典型的递归结构，用递归方式描述算法非常自然。也可以根据情况借助于辅助结构（栈或队列）用非递归方式处理
  - 状态空间搜索，探索路径的整体也形成一种树形，称为搜索树
- 对包含许多数据元素的结构，逐个访问其中各元素的过程称为遍历。对树这类比较复杂的结构，存在多种不同的遍历方法
  - 系统化的遍历方法可分为深度优先和宽度优先两类
  - 根据访问根结点信息的时机，深度优先遍历又有多种不同方式
- 二叉树和树是最重要的树形结构，两者之间可以相互转换
- 二叉树和树有许多重要应用，如表达式树，堆和优先队列，哈夫曼树和哈夫曼算法等，也有许多实际应用，如数学表达式，离散事件模拟等
- 二叉树和树都有多种不同的表示方式，使用最多的是结点链接表示。另外，人们也经常用二叉树的方式表示树