5，树与二叉树-I

- ❖ 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
- ❖ 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
- ❖ 二叉树的其他应用

复杂数据结构

- 从本章起，我们将研究一些复杂的数据结构
  - 非线性，元素之间的关系不是一对一的，存在更复杂的关系
  - n 个元素的数据结构，元素间的最远距离不是 n，可能小得多
- 这些给数据的组织和使用带来更多可能性，也带来更多问题
  - 可以表示数据之间更复杂的的关系（实际中有这种需要）
  - 数据的组织方式有更多选择
  - 可能存在更多不同的实现方法
- 处理结构中的元素的方法可能变得比较复杂

  常常需要借助于一些辅助数据结构，例如栈和队列
- 可能影响处理算法的设计和复杂性

  可能有较高的效率，也可能效率较低

树形结构

- 树形结构是一类非常重要的结构
  - 本章要讨论的树和二叉树都属于树形结构
  - 这类结构非常重要，在实际中使用广泛
- 树形结构也是由结点（树形结构中的逻辑单元，可用于保存数据）和结点间的联系构成，但其结构与线性结构（表）不同，最重要的特征是：
  - 每个结点都只有一个前驱（这一点与线性结构一样）
  - 另一方面，一个结点可以有多个后继（因此与线性结构不同）
- 另外的重要特征（使树形结构与其他更复杂的结构不同）：
  - 从一个树形结构里的任意两个结点出发，通过后继关系可达的结点集合，相互之间或者互不相交，或者有子集关系，
  - 树形结构可以表示许多常见的层次性关系

    树形结构形成了对其结点集合的一种层次性的划分

树的实例

- 生活中有许多树形结构的例子，例如
  - 家族关系，如家谱的简单情况是树形，长辈有子女，世代分层
  - 机构的结构关系，等
- 这类关系具有层次性，只有高层与低层可能有关，同层元素之间相互无关，也没有低层到高层的关系。与不同元素相关的元素互不相交
- 例：一个机构的分层结构
  - 假设机构 **A** 有两个分部 **B, C**；**B** 和 **C** 分别有下级机构 **D, E, F** 和 **G, H**；而且 **E** 有两个小组 **I , J**
  - 反映其关系的结构包含结点集合 **N** 和关系 **R**，就是一个树形结构：

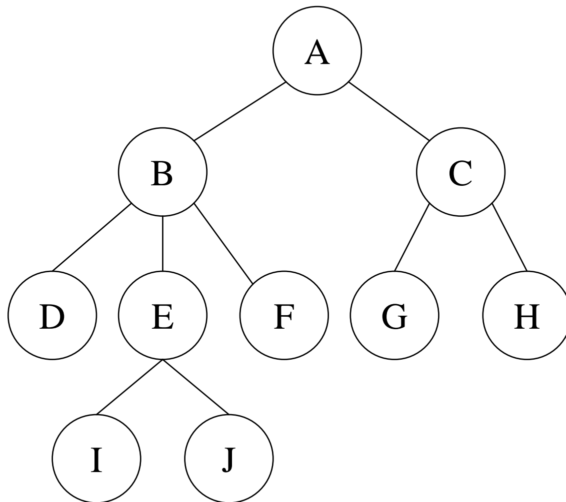
    N = {A, B, C, D, E, F, G, H, I, J}

    R = {<A, B>, <A, C>, <B, D> , <B, E>, <B, F>,
    <C, G>, <C, H>, <E, I>, <E, J> }

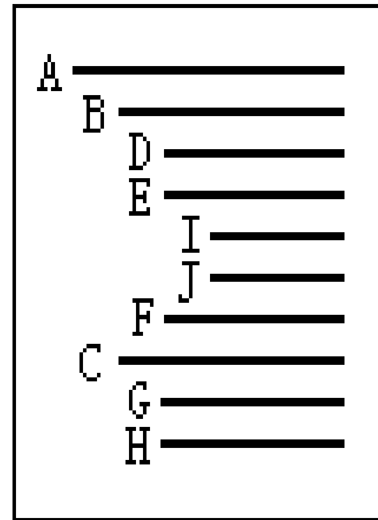
树的图示

- 树有多种直观的图示形式，可帮助直观理解树的结构

  当然，图示只能用于表示较小的树，但对学习和理解很有帮助

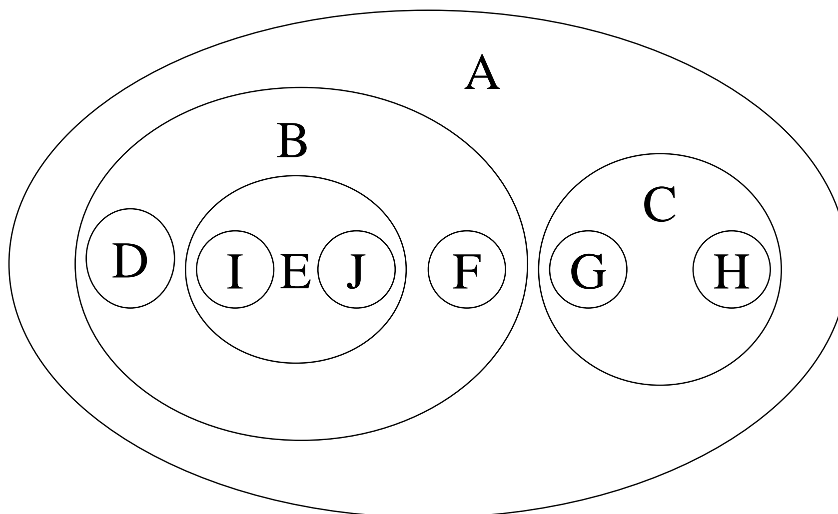


基本图示（结点连线图），树根总画在最上面



凹入表

树的图示



文氏图（Venn Diagram），也称韦恩图

John Venn 是十九世纪英国的哲学家和数学家，他在 1881年发明了文氏图

- 嵌套括号表示法（每个括号里的第一项是本层数据，随后内嵌的括号及内容表示树的子部分，其结构与外层一样。这种表示由 Lisp 语言引入）

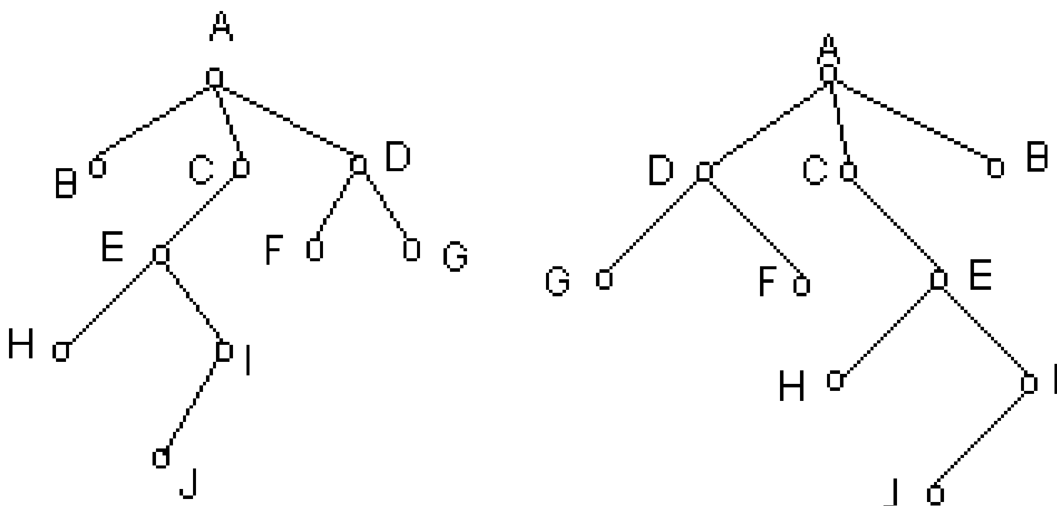
  (A (B (D) (E (I) (J)) F) (C (G) (H)))

树的定义

- 树（tree）是具有递归性质的结构，所以树的定义也是递归的
- 一棵树是 n (n≥0) 个结点的有限集 T（可为空），T 非空时满足：
  - 有且仅有一个特殊的称为根的结点 r
  - 根结点外的其余结点划分为m（m ≥ 0）个互不相交的非空有限集 $T_1, T_2, \ldots, T_m$，每个集合各为一棵非空树，称为 r 的子树 (subtree)
- 要求子树都非空，使子树的个数（和树的结构）能有明确定义
  - 结点个数为 0 的树称为空树
  - 一棵树可以只有根但没有子树（m = 0），这是一棵单结点的树，只包含一个根结点
- 树是一种层次性结构
  - 子树的根看作是树根的下一层元素
  - 一棵树里的元素可以根据这种关系分为一层层的元素

树：相关概念

- 一棵树（除树根外）可能有多棵子树，根据是否认为子树的排列顺序有意义，可以把树分为有序树和无序树两种概念
- 按有序树的观点，下面两个图表示的是两棵不同的树，按无序树的观点它们表示同一棵树



树：相关概念

- 父结点和子结点（是相对定义的）
  - ❑ 一棵树的根结点称为该树的子树的根结点的父结点
  - ❑ 子树的根是树根的子结点
- 边：从父结点到子结点的连线（注意，边有方向）
- 兄弟结点：父结点相同的结点互为兄弟结点
- 树叶、分支结点：没有子结点的结点称为树叶，树中的其余结点称为分支结点（注意：分支结点可以只有一个分支）
- 祖先和子孙：基于父结点/子结点关系和传递性，可以确定相应的传递关系，称为祖先关系或子孙关系

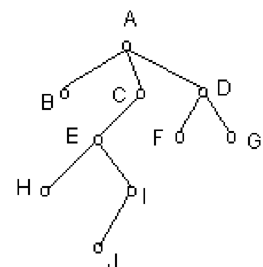
  由这两个关系决定了一个结点的祖先结点，或子孙结点
- 度数：一个结点的子结点个数称为该结点的度数，显然树叶的度数为 **0**

  一棵树的度数就是它里面度数最大的结点的度数

树：相关概念

- 路径，路径长度
  - ❑ 从一个祖先结点到其子孙结点的一系列边称为树中一条路径

    显然，从一棵树的根到树中任一个结点都有路径，且路径唯一
  - ❑ 路径中边的条数称为路径长度，认为每个结点到自身有长 **0** 的路径
- 结点的层数
  - ❑ 树根到结点的路径长度是该结点的层数
  - ❑ 结点都有层数，根所在的层为 **0**，子结点的层数比⋯



- 高度（或深度）
  - ❑ 树的高度或深度是树中结点的最大层数（最长路径的长度）加 **1**
  - ❑ 是树的整体性质，空树高度为 **0**，只有根结点的树高度为 **1**
- 结点的顺序（最左，…，仅在考虑有序树时有意义）

树林：定义，与树的关系

- 树林：m（m ≥ 0）棵互不相交的树的集合
- 一棵非空树是一个二元组 Tree = (r, F)，其中
  - r 是树的根结点
  - F 是 m（m ≥ 0）棵子树构成的树林
  - F = (T_1, T_2,…, T_m)。其中 T_i 称为根 r 的第 i 棵子树（这是有序树的情况。对于无序树 F 是一个集合，F = {T_1, T_2,…, T_m} ）
- 注意树与树林的关系：
  - 树由根和子树树林构成
  - 树林由一集树组成
  - 树和树林可以相互递归定义
- 有关树和树林的进一步讨论放到后面进行。我们先转到一种相对简单的树形结构：二叉树

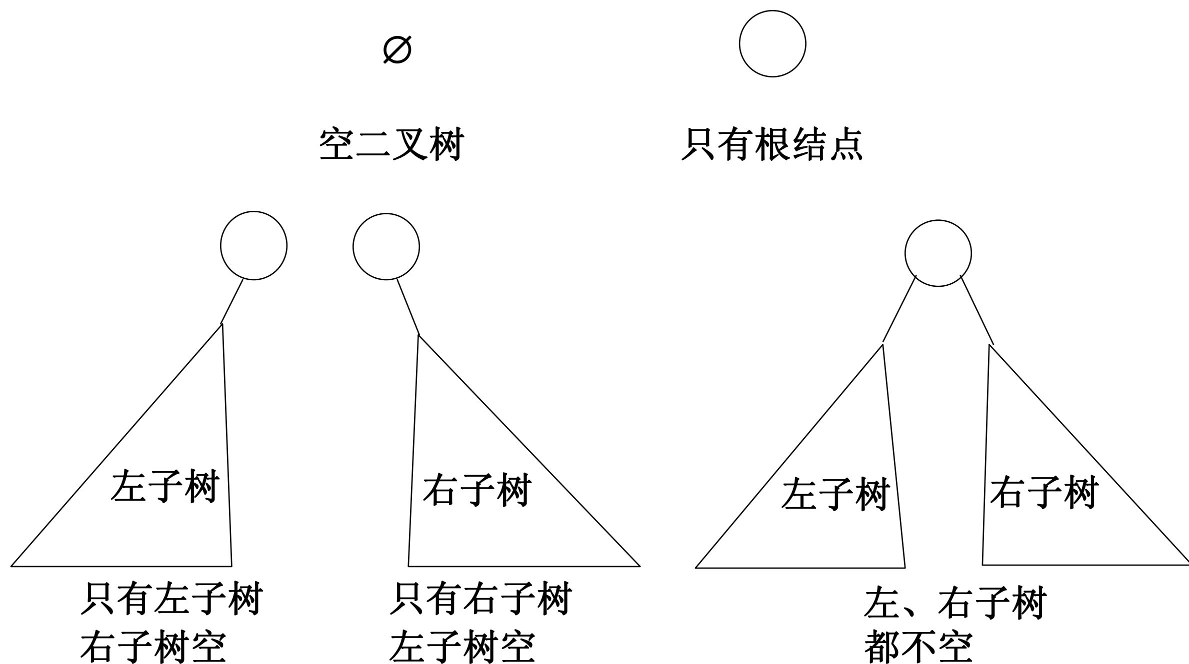
二叉树：定义

- 二叉树是一种树形结构
  - 特点是与每个结点关联的子结点至多有两个（可为 0, 1, 2）
  - 每个结点的子结点关联有位置关系
- 定义（二叉树）：
  - 二叉树是结点的有限集合，该集合或为空集，或由一个根元素和两棵不相交的二叉树组成（递归定义）
  - 二叉树的两棵子树分别称作它的（其根的）左子树和右子树
- 注意二叉树的特点：
  - 一个结点至多有两棵子树
  - 子树有左右之分

  因此，二叉树是与树不同的结构（不是树的特殊情况）

二叉树：形态

- 二叉树共有 **5** 种不同的基本形态：



二叉树

- 二叉树不是树的特殊情形，主要是其子树分左子树和右子树，即使只有一棵子树也要明确说明是左还是右

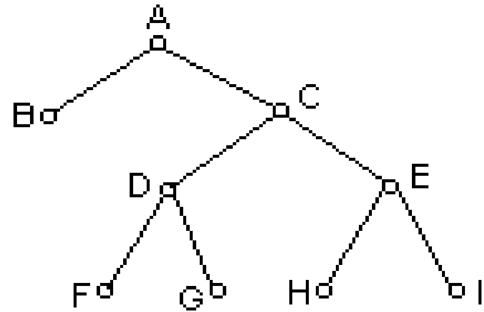
  画二叉树的图示时，需要明确地把子树画在左边或右边

  下面是两棵不同的二叉树，而作为树它们是相同的：

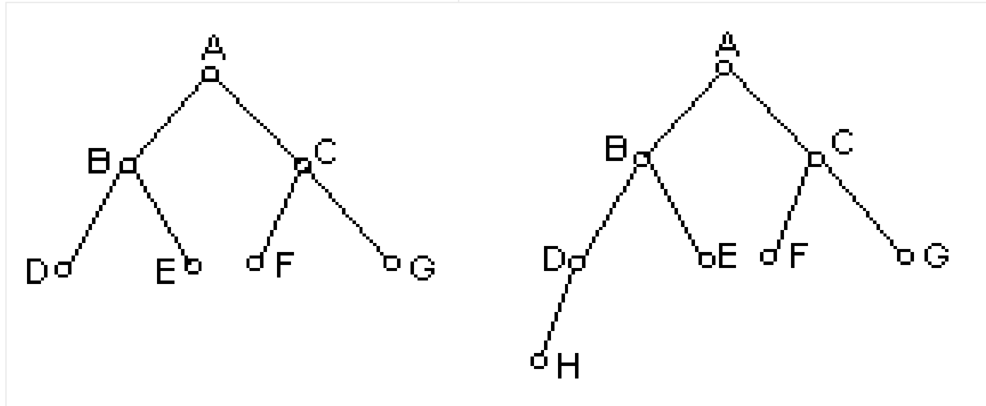
- 对二叉树，树的许多概念都可沿用，包括：父/子结点，树叶和分支结点，结点的度数，祖先/子孙，路径及其长度，结点的层数，二叉树的高度等。有关定义类似，不再重复
- 下面介绍另外几个概念

满的和完全的二叉树

- 满二叉树：树中每个分支结点（非叶结点）都有两棵非空子树



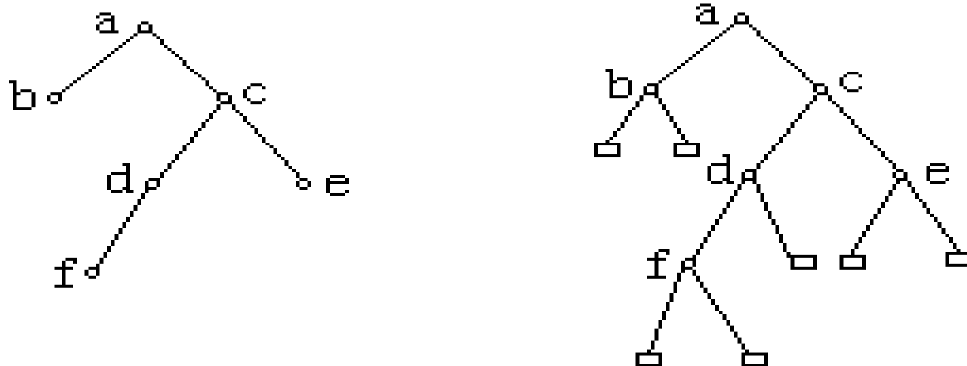
- 完全二叉树：除最下两层外，其余结点度数都是 **2**（显然都不是叶结点），如果最下一层的结点不满，则所有空位都在在右边，左边没有空位



扩充二叉树

- 扩充二叉树（由已有非空二叉树生成的一种二叉树）：

  是原二叉树的最小结点扩充，使原树中所有结点的度数都变成 **2**



- 扩充二叉树新增结点（称为其外部结点）的个数比原树结点（称为其内部结点）的个数多 **1**。这一性质可以通过树上的结构归纳法证明：
  - 如果原树只有一个结点，结论显然成立
  - 如果原树包含根结点 **r** 和左右子树 T_1, T_2，… …

路径

- 扩充二叉树有下面性质（其中 n 是内部结点的个数），且

$$E = I + 2n$$

  E 是扩充二叉树的外部路径长度：

  扩充二叉树里从根到各外部结点的路径长度之和

  I 是扩充二叉树的内部路径长度：

  扩充二叉树里从根到各内部结点的路径长度之和

  这一性质也可以用结构归纳法证明。请自己考虑

- 二叉树有很多有用的性质，下面讨论

  对于作为数据结构，最重要的性质是树和高度和树中可以容纳的最多结点个数之间的关系

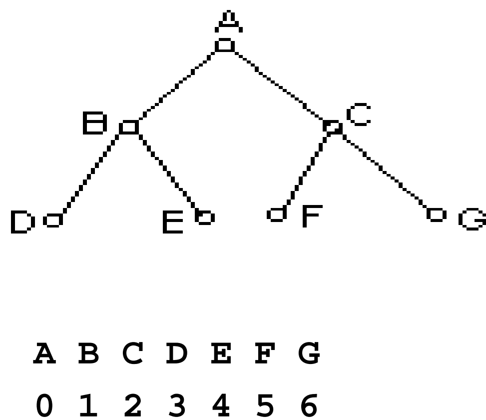
  树高度类似于表长，是从根结点（首结点）到其他结点的最大距离。长 n 的表只有 n 个结点，而高 n 的二叉树中的结点可能多得多

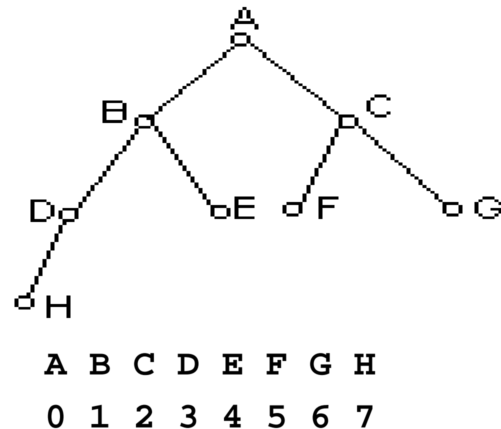
二叉树的性质

- 性质1. 非空二叉树第 i 层上至多有 2^i 个结点（$i \geqslant 0$）
- 性质2. 高度为 k 的二叉树至多有 2^k-1 个结点（$k \geqslant 0$）
- 性质3. 对任何非空二叉树 T，若其叶结点个数为 n_0，度数为 2 的结点个数为 n_2，则$n_0 = n_2 + 1$
- 性质4. n 个结点的完全二叉树的高度 $k = \lceil \log_2(n+1) \rceil$
- 性质5. 满二叉树里的叶结点比分支结点多一个
- 上面这些性质都可以基于空树或一个结点的树，用结构归纳法证明
- 例：证明性质 5
  - 基础：一个结点的树只有一个叶结点，无分支结点，结论成立
  - 归纳：设满二叉树 T 有根 r 和两棵子树 T_1 和 T_2，且 T_1 中叶结点比分支结点多一个，T_2 中叶结点比分支结点多一个。立即可以算出 T 中叶结点比分支结点多一个

性质：完全二叉树

- 性质6.（完全二叉树）如果 n 个结点的完全二叉树按层次按从左到右的顺序从 0 开始编号，对任一结点 i（0 ≤ i ≤ n-1) 都有：
  - 序号 0 的结点是根；对于 i > 0，其父结点是 ⌊(i - 1)/2⌋
  - 若 2 * i + 1 ≤ n，其左子结点序号为2 * i + 1；否则它无左子结点
  - 若 2 * i + 2 ≤ n，其右子结点序号为2 * i + 2；否则它无右子结点





完全二叉树和一般二叉树

- 性质 6 是完全二叉树的一个重要性质：
  - 使其可以方便地存入一系列连续位置（存入一个表或数组）
  - 不需要另外保存其他信息，直接根据元素的下标就可以找到一个结点的子结点或者父结点（并确定二叉树的结构）
- 一般二叉树不能方便地映射到线性结构（完全二叉树到线性结构有定义非常自然的双向映射，可以方便地从其线性结构恢复完全二叉树）

  下面很快就会看到这一性质的价值和用途
- 一般而言，n 个结点的二叉树有如下情况：
  - 如果它足够“丰满整齐”，树中最长路径的长度将为 O(log n)

    例如，完全二叉树就是这样情况
  - 如果它比较“畸形”，最长路径的长度可能到达 O(n)

    也就是说，一般而言，n 个结点的二叉树的最长路径为 O(n)

二叉树数据结构

- 考虑二叉树作为一种数据结构，其构造和基本操作等问题
- 二叉树数据结构的基本操作应该包括
  - 创建二叉树
  - 一棵二叉树或为空（用 **None** 表示），或是两棵已有二叉树和要存在树根结点的一项数据，构造起的根结点代表构造出的二叉树：

    BiTree(dat, left, right)
  - 判断树空：**is_empty(bitree)**
  - 访问操作，访问二叉树的组成成分：
    - 访问二叉树的根结点数据元素：**data()**
    - 取得一棵二叉树的左右子树：**right()**，**left()**
  - 其他操作，后面考虑

二叉树的 list 实现

- 首先考虑二叉树的一种 **list** 实现

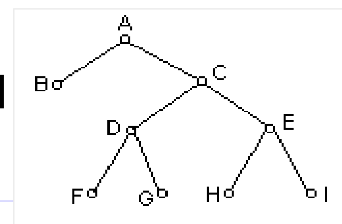
  实际上，这种技术不仅可用于实现二叉树，也可用于实现一般的树
- 采用下面的设计：
  - 空树用空表 **[ ]** 表示
  - 非空二叉树用包含三个元素的表 **[d, l, r]** 表示

    其中 **d** 表示存在根结点的元素，

    l 和 **r** 是两棵子树，按同样方式用两个 **list** 表示
- 显然，这样设计做出的是一种递归结构，每棵二叉树都有一个与之对应的（递归结构的）**list**。例如下面是一棵二叉树及其 **list** 表示

  ['A', ['B', [], []], ['C', ['D', ['F', [], []], ['G', [], []]],
  ['E', ['H', [], []], ['I', [], []]]]]

  其实这就是前面提过的嵌套括号表示（**Lisp**）



二叉树的 list 实现

- 相关的二叉树操作很容易实现，下面是一组基本操作

```
def BiTree(data, left, right): return [data, left, right]

def is_empty_BiTree(bitree): return bitree == []

def root(bitree): return bitree[0]
def leftch(bitree): return bitree[1]
def rightch(bitree): return bitree[2]

def set_root(bitree, data): betree[0] = data
def set_leftch(bitree, left): betree[1] = left
def set_rightch(bitree, right): betree[2] = right
```

- 简单使用：

  t1 = BiTree(2, BiTree(4, [], []), BiTree(8, [], []))

  这相当于：**t1 = [2, [4, [], []], [8, [], []]]**

二叉树的 list 实现

- 可以修改二叉树中的任何部分，例如

  set_leftch(leftch(t1), BiTree(5, [], []))

  其中把 **t1** 的左子树的左子树用 **BiTree(5, [], [])** 取代，**t1** 变成了：

  [2, [4, [5, [], []], []], [8, [], []]]

  这是一棵高度为 **3** 的树。**list** 的嵌套深度就是树的高度，不计空表
- 下面考虑这种二叉树实现的一个应用：表达式树
  - 这一应用主要是利用二叉树的结构
  - 二叉树的层次结构可以用来表示表达式的层次结构
  - 二叉树中结点和子树的关系，用于表示运算符对运算对象的作用
- 下面考虑只包含二元运算符的表达式

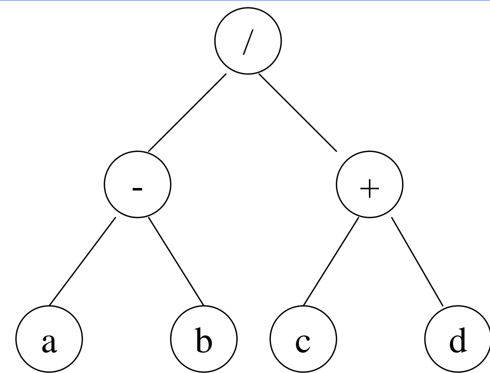
  实际上，这种表示还可以扩充，完全可以用于一般的表达式

二叉树应用：表达式树

- 算术（和代数）表达式可以自然地映射到二叉树（运算符看作二元运算符）
  - 基本运算对象（数和变量）作为叶结点的数据
  - 运算符作为分支结点的数据

    其两棵子树是它的运算对象

    可以是基本运算对象，也可以是作为运算对象的两个子表达式
  - 实际上，一元和二元函数也可以采用与运算符同样的表示方式
  - 多元函数的问题后面有讨论
- 各种数学软件都采用此类表示方式



(a – b) / (c + d)的二叉树表示

表达式树

- 为了表示的更简洁，对上面的二叉树的 list 表示做一点修改

  把基本运算对象（数或变量）直接放在二叉树中空表的位置

  例如，表达式 3 * (2 + 5) 直接映射到二叉树是

  ['*', [3, [], []], ['+', [2, [], []], [5, [], []]]]

  将简化表示为（带括号的前缀表达式，括号表示运算符的作用范围）

  ['*', 3, ['+', 2, 5]]
- 采用这种修改的表达方式，表达式由两种结构组成：
  - 如果是表（list），就是运算符作用于运算对象的复合表达式
  - 否则就是基本表达式，即，数或者变量
  - 上述两条可用于解析表达式的结构，实现对表达式的处理
  - 算术表达式是特殊情况，其中的基本表达式都是数

表达式树

- 下面定义几个表达式构造函数：

```
def make_sum(a, b):
    return ['+', a, b]

def make_prod(a, b):
    return ['*', a, b]
def make_diff(a, b):
    return ['-', a, b]

def make_div(a, b):
    return ['/', a, b]

# 其他构造函数与此类似，略
```

- 下面语句构造出一个算术表达式：

```
e1 = make_prod(3, make_sum(2, 5))
```


表达式树

- 显然，采用这种结构可以构造出具有任意复杂结构的表达式

  用字符串表示变量，就能构造出各种代数表达式，例如

```
make_sum(make_prod('a', 2), make_prod('b', 7))
```

- 定义表达式处理函数，经常需要区分基本表达式（直接处理）和复合表达式（递归处理）。定义一个函数判别是否为基本表达式：

```
def is_basic_exp(a):
    return not isinstance(a, list)
```

- 有时需要判断表达式是否为数。在 **Python** 程序里判别是否为数对象，需要用标准库包 **numbers** 里定义的 **Number** 类。使用前需要导入

```
from numbers import Number

# 表达式 isinstance(x, Number) 返回 True 表示是数对象

# 可以是整数 int，浮点数 float 或复数 complex
```

表达式“求值”

- 现在考虑一个求表达式值的函数
  - 对表达式里的数和变量，其值就是它们自身
  - 其他表达式，要根据运算符的情况处理，可以定义专门处理函数
  - 如一个运算符的两个运算对象都是数，它就可以求出一个数值
  - 还有些情况，如加数是 **0**，乘数是 **0** 或 **1**，可部分求值化简
- 求值函数的基本部分：

```
def eval_exp(e):
    if is_basic_exp(e):
        return e
    op, a, b = e[0], eval_exp(e[1]), eval_exp(e[2])
    if op == '+':
        return eval_sum(a, b)
# 下页继续
```


表达式“求值”

```
    elif op == '-':
        return eval_diff(a, b)
    elif op == '*':
        return eval_prod(a, b)
    elif op == '/':
        return eval_div(a, b)
    else:
        raise ValueError("Unknown operator:", op)
```

- 注意这里的实现方法：
  - 根据运算符的情况，把不同计算分发给具体实现函数处理
  - 可以送给异常任意多个实际参数
    - 如果不处理，在显示异常信息时就会输出实际参数的值，这里会输出实际的“不能识别的运算符”
    - 捕捉并处理时，可通过异常对象的 **args** 取得这些实参值使用

表达式树

- “求值”和式和除式的专门函数（其他类似）：

```
def eval_sum(a, b):
    if isinstance(a, Number) and isinstance(b, Number):
        return a + b
    if isinstance(a, Number) and a == 0: return b
    if isinstance(b, Number) and b == 0: return a
    return make_sum(a, b)

def eval_div(a, b):
    if isinstance(a, Number) and isinstance(b, Number):
        return a / b
    if isinstance(a, Number) and a == 0: return 0
    if isinstance(b, Number) and b == 1: return a
    if isinstance(b, Number) and b == 0:
        raise ZeroDivisionError
    return make_div(a, b)
```


表达式树

- 可以定义更多的表达式操作，例如
  - 以比较好读的形式输出表达式（请自己想想，后面有讨论）
  - 代数表达式的各种运算（化简是很困难的工作，除简单化简外）
  - 对某个变量，求导代数表达式的函数（得到另一代数表达式）
  - 求不定积分（不容易做好）
  - 等等
- 显然，上面的表达式定义形式也可以扩充
  - **list** 可以有任意多的成员，可以表示一元或多元运算符的作用
  - 例如，允许写 **['+', 2, 4, ['*', 15, ['sin', 2.3], 'x']]**

    如何修改前面的定义，请自己考虑
- 可以在 **Python** 基础上做出一个完成复杂表达式计算的系统，功能类似于 **Maple** 或 **Mathematica**