5，二叉树和树-2

- ❖ 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，堆，堆排序，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：哈夫曼算法和哈夫曼树
- ❖ 树和二叉树

优先队列

- 优先队列是一种常用的缓存结构
  - 与栈和队列类似，可以保存数据元素，访问和弹出元素
  - 特点是存入优先队列的每项数据附有一个优先级
  - 保证任何时候访问或弹出的总是当时所存元素中最优先的
- 抽象看，这里考虑的是存储一个有序集 **S = (D, ≤)** 的元素

  其中 ≤ 是集合 **D** 的一个全序（非严格的，有自反性）

  要求保证的是“最小元素先出”（先用）

  具体集合和序可以根据需要确定（同一集合上也可能有不同的序）
- 如果优先队列里可能存在多个最优先元素
  - 如果要求这些元素的先进先出，那就只能做出效率较低的实现
  - 如果只要求保证访问（弹出）的是最优先元素中的一个，不要求一定是最早进入优先队列的元素，就存在效率更高的实现

优先队列

- 优先关系代表了数据的某种性质，如用于描述
  - 各项工作的计划开始时间（实际中，模拟中都可能使用）
  - 一个大项目中各种工作任务的急迫程度（或截止期）
  - 银行客户的诚信评估，用于决定优先贷款，等等
- 优先队列的操作也很简单，应包括
  - 创建，判断空（还可以有清空内容、确定当前元素个数等）
  - 插入元素，访问和删除优先队列里（当时最优先）的元素
- 简单实现：基于线性表技术实现优先队列
- 考虑用连续表技术实现优先队列，存在着两种可能的方案：
  - 存入时保证元素按优先顺序排列，任何时候都可以直接取到最优先元素。采用有组织的元素存放方式，取用方便
  - 元素直接存到表的尾端，取用时检索最优先元素。用无组织方式存放元素，把选择最优元素的工作推迟到访问/取出时

优先队列

- 经过比较评价（请自己分析其合理性），我们采用第一种技术
  - 加入新元素时，确定正确插入位置保证表元素按优先顺序排序
  - 为保证 **O(1)** 的弹出元素操作，最优先元素应该出现在表尾端
- 注意：
  - **Python** 的 **list** 对象能根据元素个数的需要自动扩大元素存储区

    但也要注意，超下标范围的赋值是运行错误（**IndexError**）
  - 因此在加入新元素时不能用下面形式的语句做元素移位移位

    elems[i+1] = elems[i]

    只能用 **list** 的 **insert**（或其他类似操作）做定位插入

优先队列：连续表实现

- 把这个优先队列定义为一个类：

```
class PrioQue:
    def __init__(self, lst = [ ]):
        self.elems = sorted(lst)
```

  - 引进参数使人可以提供一组初始元素

- 插入元素时需检查队列元素的优先关系，确定正确插入位置
  - 假定存入队列的元素可以用 **<=** 比较。下面将一直把较小看作优先：

```
    def enqueue(self, e):
        i = len(self.elems) - 1
        while i >= 0:
            if self.elems[i] <= e:
                i -= 1
            else: break
        self.elems.insert(i+1, e)
```

  - **while** 的条件保证了优先度相同元素的正确排列顺序，这样这里就能保证它们的“先进先出”
  - **while** 结束时 **i** 或为 **–1**，或是第一个大于 **e** 的元素的下标

优先队列：连续表实现

- 其他函数都比较简单：

```
def is_empty(self):
    return len(self.elems) == 0

def peek(self):
    if self.is_empty():
        raise PrioQueueError("in top") # 给一点信息
    return self.elems[len(self.elems)-1]

def dequeue(self):
    if self.is_empty():
        raise PrioQueueError("in pop")
    return self.elems.pop()
```

- 还需要定义一个异常类（类体为 pass）：

```
class PrioQueueError(ValueError): pass
```

优先队列：连续表实现

- 分析各操作的效率（复杂性）：
  - 插入元素是 **O(n)** 操作，其他都是 **O(1)** 操作
  - 如果采用另一种实现方式
    - 插入元素是 **O(1)** 操作
    - 但检查和弹出队列首元素都是 **O(n)** 操作
- 注意：对前面实现，即使插入时存储区满需要换存储，也是 **O(n)**
- 下面考虑改善优先队列性能的可能性

优先队列实现的效率

- 采用线性表实现优先队列，无论怎样实现，在插入与取出元素的操作中总有一种是线性复杂性操作，这一情况不令人满意

  前面讨论了采用连续表技术的实现，用链接表的情况也类似

- 例如，按序插入低效的原因是需要沿着表的顺序检索插入位置

  表长度是 **n**，检索（和插入）必然需要 **O(n)** 时间

  按优先级的线性顺序排列元素，这种复杂性不可避免

- 要想突破 **O(n)** 的复杂性约束，必须考虑其他数据结构组织方式

- 一般而言，确定最优先元素并不需要与其他所有元素比较
  - 例如，体育比赛中的淘汰赛，每个选手只需进行约 **log(n)** 场比赛
  - 基于树形结构的祖先/子孙序，可能得到更好的效率
  - 在优先队列实现中利用树形结构的优势，需要解决：在反复插入和删除元素的过程中，如何始终保持结构的有效性

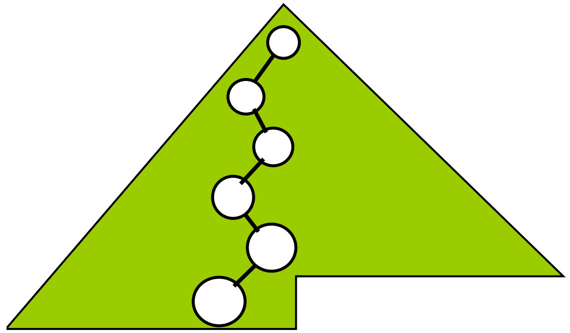
堆

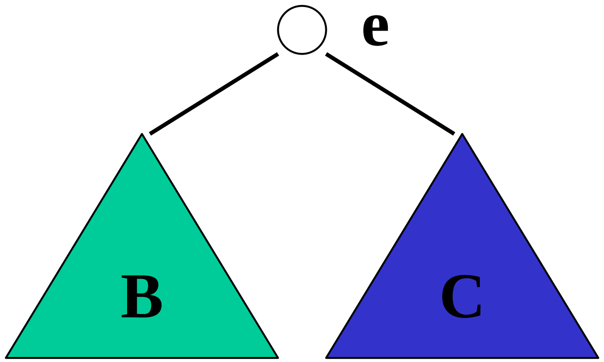
- 采用树形结构实现优先队列的一种有效技术称为堆
  - 一个堆在结构上是一棵在结点里存储数据的完全二叉树
  - 堆中元素按数据的优先关系排序，这种堆序保证任一结点所存数据（按所考虑的序）不大于其子结点（如果存在）的数据
  - 注意：堆序就是要求按结点子孙序，数据从小到大（非严格）递增
- 根据上面的定义可知：
  - 堆中最优先的元素位于堆顶，立即可得（**O(1)** 时间访问）
  - 位于树中不同路径上的元素，这里不关心其顺序关系
- 按上面要求的堆序构造的堆称为“小顶堆”（小元素在上）

  另一种堆称为“大顶堆”
    - 每个结点里的数据都大于等于其子结点的数据
    - 堆顶是堆中的“最大元素”

堆

- 前面介绍过：
  - 一棵完全二叉树可以自然而且信息完全地存入一个连续的线性结构（连续表）
  - 因此，一个堆也可以自然地存放在一个连续表里，仅使用下标就能方便地访问树中一个结点的父结点/子结点
- 几个重要事实：
  - 去掉一个堆的最后元素，剩下的仍是堆
  - 一个堆去掉堆顶，其余元素形成两个堆
  - 给按上面方式得到的两个堆加一个元素作为根，得到一棵完全二叉树但未必是堆
  - 在一个堆的最后加上一个元素，整个结构是一棵完全二叉树，但未必是堆





堆和优先队列

- 现在考虑如何基于堆的概念实现优先队列
- 下面讨论中说到堆时，经常指的是存储在连续表里的一个完全二叉树，其元素的存储情况形成了一个堆（一一对应）
- 用堆实现优先队列，可直接找到最优先元素，但还需要解决两个问题：
  - 加入一个元素，怎样将加入后得到的完全二叉树转变为堆
  - 弹出最小元素后，如何将剩下的元素重新做成堆

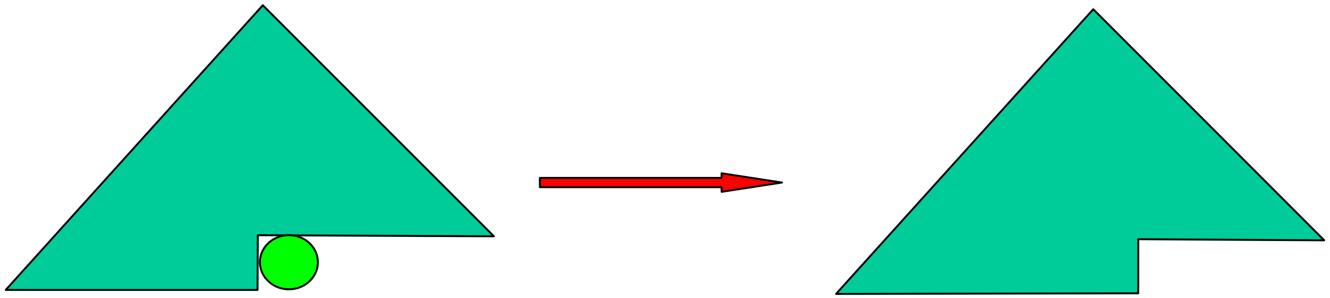
  下面将看到，这两个操作均可以在 **O(log n)** 时间内完成
- 其他操作都简单：
  - 建空堆、判空、访问最优先元素都是 **O(1)** 操作
  - 求堆大小和清空等也是 **O(1)** 操作
  - 显然在堆的表示中必须有元素个数记录

基本操作：元素加入和向上筛选

- 实现在堆中正确加入和弹出数据元素的基本操作是筛选

  筛选分为向上筛选和向下筛选两种

- 向上筛选：在堆 **H** 的最后加入一个元素 **e** 后，通过一次向上筛选，使整个完全二叉树重新恢复为一个堆

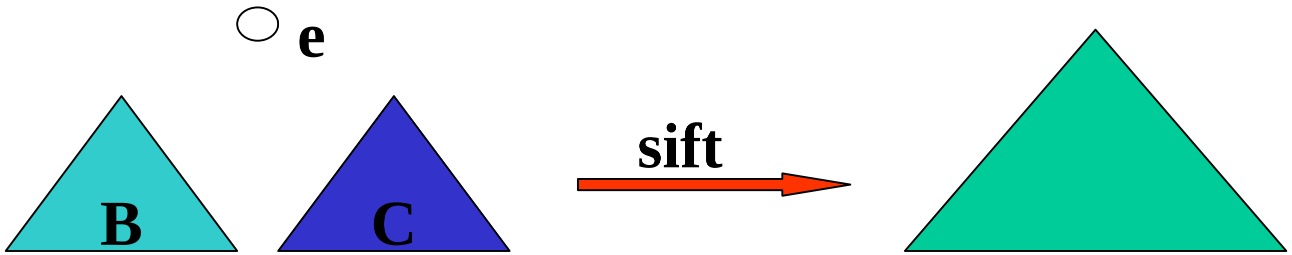


方法：不断用 **e** 与其父结点的数据比较，如果 **e** 较小就交换两个元素的位置。直至 **e** 的父结点的数据小于等于 **e** 时停止

- 在基于堆的优先队列中插入操作的实现：把新加入的元素放在（连续表里）已有的元素之后，执行一次向上筛选操作
- 结论：加入元素的操作可以在 **O(log n)** 时间完成（根据二叉树性质）

基本操作：向下筛选和元素删除

- 向下筛选：设两个堆 **B,C** 加根元素 **e** 构成一棵完全二叉树，现在需要把它们做成一个堆，这时做一次向下筛选：



- 方法：用 **e** 与 **B**、**C** 的顶元素（根）比，最小者作为整个堆的顶。若 **e** 不是最小，最小的必为 **B**（或 **C**）的顶元素。下面就考虑把 **e** 放入去掉堆顶的**B**（或 **C**）（同样问题但规模更小）。两种情况结束：
  - 某次比较中 **e** 最小，以它为顶的局部树成堆了，整个结构也成堆
  - **e** 落到底，这时它自身就是一个堆，整个结构也成为堆

  在这两种情况下，重新构造堆的工作都完成了

- 优先队列弹出操作的实现：弹出当时堆顶，从堆最后取一个元素放在堆顶，执行一次向下筛选。结论：删除元素是 **O(log n)** 时间操作

优先队列的堆实现

- 下面定义一个基于堆结构实现优先队列的类：

```
class PrioQueue:
    def __init__(self, elist = []):
        self.elems = elist
        if elist != []:
            self.buildheap()

    def is_empty(self):
        return len(self.elems) == 0

    def peek(self):
        if self.is_empty():
            raise PrioQueueError("in top")
        return self.elems[0]
```

- 应选择尾端端作为加入元素一端，选择首端作为取元素端，与基于线性表的情况相反。请根据前面的分析，考虑交换两端的排会遇到的情况

优先队列的堆实现

- 加入新数据元素的操作，关键是完成向上筛选的辅助函数

```
def enqueue(self, e):
    self.elems.append(None) # add a dummy element
    self.siftup(e, len(self.elems)-1)

def siftup(self, e, last):
    elems, i, j = self.elems, last, (last-1)//2
    while i > 0 and e < elems[j]:
        elems[i] = elems[j]
        i, j = j, (j-1)//2
    elems[i] = e
```

- 注意：这里没有先存入元素，而是直接去查找元素的正确存入位置
  - 循环条件保证跳过的都是优先度较低的元素，将它们下移
  - 循环结束时 i 就是应该存入元素的位置
  - 也可以先把 e 存入最后，需要上移时交换元素

优先队列的堆实现

- 弹出最优先元素的操作，关键也是向下筛选

```
def dequeue(self):
    if self.is_empty():
        raise PrioQueueError("in pop")
    elems = self.elems
    e0 = elems[0]; e = elems.pop()
    if len(elems) > 0:
        self.siftdown(e, 0, len(elems))
    return e0

def siftdown(self, e, begin, end):
    elems, i, j = self.elems, begin, begin*2+1
    while j < end: # invariant: j == 2*i+1
        if j+1 < end and elems[j+1] < elems[j]:
            j += 1 # elems[j] <= its brother
        if e < elems[j]: # e is the smallest of the three
            break
        elems[i] = elems[j] # elems[j] is the smallest, move it up
        i, j = j, 2*j+1
    elems[i] = e
```

堆的应用：堆排序

- 如果一个连续表里存储了一个堆，按优先队列的的操作方式反复弹出堆顶元，将得到一个递增序列。这显然是一种可行的排序方式
- 基于这种技术实现排序，还需要解决两个问题：
  - 连续表里的初始元素序列通常不满足堆序，怎样整理？
  - 选出的元素存放在哪里？能不能不用其他空间？
- 第二个问题很好解决：随着元素弹出，堆中元素也越来越少。每弹出一个元素，后面就会空出一个位置，正好存放弹出的元素

  但直接存放，用小顶堆时，最后排出的序列是左边最大且越来越小

  如果需要元素从小到大排序，可用大顶堆或定义合适的序谓词
- 第一个问题也可以解决：
  - 从堆最下最右的分支结点出发，向前一个个建堆
  - 每次要做的都是把两个堆加一个元素形成的完全二叉树调整为堆

堆排序

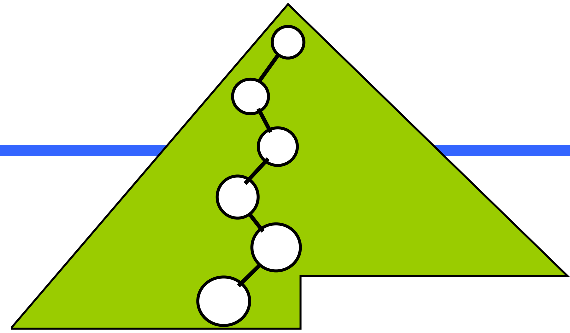
- 函数定义不复杂（有了向下筛选操作）：

```
def heap_sort(elems):
    end = len(elems)
    for i in range(end//2, -1, -1):
        siftdown(elems, elems[i], i, end) # 定义类似
    for i in range((end-1), 0, -1):
        e = elems[i]
        elems[i] = elems[0]
        PrioQueue.siftdown(elems, e, 0, i)
```

- 函数主要是两个循环：
  - 第一个循环建堆，从位置 **i** 开始，以 **end** 为范围边界
  - 第二个循环逐个取出最小元素，将其在表的最后后退着积累

    这个循环做 **n** 次，每次新的顶元素下行距离不超过 **log n**。显然第二个循环的总开销为 **O(n log n)**

堆排序



- 考虑第一个循环的复杂性：

```
for i in range(end//2, -1, -1):
    siftdown(elems, elems[i], i, end)
```

- 元素个数为 **n** 的完全二叉树高度为 **h**，易见，在这棵树里
  - 高度为 **2** 的子树约为 **n/4**（**n/2²**）棵，把这样的一棵树调整为堆，元素的移动距离不超过 **1**
  - 高度为 **3** 的子树约为 **n/8**（**n/2³**）棵，把这样的一棵树调整为堆，其中元素移动距离不超过 **2**，**... ...**。总移动次数：

$$
\begin{aligned}
C_1(n) &\le \textstyle\sum_{i=0}^{h-1}(h-i)2^{i+1} = \sum_{j=1}^{h} j \cdot 2^{h-j+1} \qquad (\text{let } j = h-i) \\
&= \textstyle\sum_{j=1}^{h} 2 \cdot j \cdot 2^{-j} \cdot 2^{h} \le 2n \sum_{j=1}^{h} j/2^{j} \qquad (\sum_{j=1}^{h} j/2^{j} \le 2) \\
&\le 4n = O(n)
\end{aligned}
$$

- 结论：堆排序的时间复杂性是 **O(n log n)**，空间是 **O(1)**

优先队列应用：海关检查站的模拟

- 现在考虑优先队列的一类应用：离散事件模拟。这种工作需要模拟一个系统在一段时间里的行为，系统行为表现为：
  - 系统运行中可能（带一些随机性）发生事件
  - 一个事件在某个时刻发生，其发生有可能导致其他事件在未来发生
- 假设有一个海关检查站，负责检查过境车辆：
  - 车辆按一定间隔到达，间隔时间有随机性，设范围为 **[a, b]** 分钟
  - 由于车辆的不同情况，每辆车的检查时间为 **[m, n]** 分钟
  - 海关可以开 **k** 个通道，我们希望模拟 **N** 分钟时长的过关情况
- 这里的基本想法是按事件发生的顺序处理，用一个优先队列保存已知在在将来会并需要处理的事件，按事件发生的时间顺序
  - 系统的模拟过程就是不断处理已经存储的事件，直至模拟结束
  - 随着模拟的进行，系统维护着模拟的时间

海关检查站的模拟

■ 根据上面考虑，可以实现一个通用的模拟框架：

```
class Simulation:
    def __init__(self, duration):
        self._eventq = PrioQueue()
        self._time = 0
        self._duration = duration

    def run(self): # 模拟的运行就是处理一个个事件
        while not self._eventq.is_empty():
            event = self._eventq.dequeue()
            self._time = event.time()
            if self._time > self._duration: break
            event.run() # 一个事件的运行可能把新事件加入 eventq

    def add_event(self, event): self._eventq.enqueue(event)

    def cur_time(self): return self._time
```

海关检查站的模拟

- 一个具体模拟，需要实现一组特殊的事件（类），考虑定义一个公共基类，实现各种事件的公共操作。这是离散模拟系统的一种实现方法：

```
class Event:
    def __init__(self, event_time, customs):
        self.ctime = event_time
        self.customs = customs

    def __lt__(self, other_event): # 为事件定义 < 运算符表示优先序
        return self.ctime < other_event.ctime

    def time(self): return self.ctime

    def run(self, time, event_name): pass # 需要派生类的具体定义
```

- 采用"面向对象"的技术，把具体事件类定义为 **Event** 的派生类
  - 不同派生类根据自己的情况，定义具体的 **run** 方法
  - 这些方法完成所需操作，还可能生成新的事件加入事件队列

海关检查站的模拟

- 海关检查站系统有几种事件：
  - 汽车到达事件，定义类 **Arrive**
  - 汽车开始检查事件，因为太简单，在下面实现中没定义专门的类
  - 汽车检查完毕的离开事件，定义类 **Leave**
- 这两个类都定义为 **Event** 的派生类，它们的对象将放入事件队列
  - 它们的构造函数完成必要的设置
  - 它们的 **run** 函数描述这种事件发生时应该做的事情
- **Arrive** 类及其构造函数

```
class Arrive(Event):
    def __init__(self, arrive_time, customs):
        Event.__init__(self, arrive_time, customs) # 调用 Event的...
        self.customs.add_event(self)
        # 这里包含了将新生成事件加入事件队列的操作
```

海关检查站的模拟

- **Arrive** 类的 **run** 函数

```
def run(self):
    time, customs = self.ctime, self.customs
    event_log(time, "car arrive") # 输出“日志”信息，有利检查

    # genarate the next Arrive event
    Arrive(time + randint(*car_arrive_interval), customs)

    car = Car(time) # dealing with current Arrive car
    i = customs.find_gate() # 有空闲gate时返回编号,否则 None
    if i is not None:
        event_log(time, "car check") # 输出“日志”信息
        Leave(time + randint(*car_check_time), i, car, customs)
    else:
        customs.enqueue(car)
```

- 如果遇到没有空闲边检通道时，车辆进入排队。注意，**Arrive** 事件可能生成一个具有特定时间和通道号的 **Leave** 事件

海关检查站的模拟

- **Leave** 类的定义类似，但需要记录更多信息：

```
class Leave(Event):
    def __init__(self, leave_time, gate_num, car, customs):
        Event.__init__(self, leave_time, customs)
        self.car = car; self.gate_num = gate_num
        self.customs.add_event(self)

    def run(self):
        time, customs = self.ctime, self.customs
        event_log(time, "car leave")
        customs.free_gate(self.gate_num)
        customs.car_count_1()
        customs.total_time_acc(time - self.car.arrive_time())
        if customs.has_queued_car():
            car = customs.next_car()
            i = customs.find_gate()
            event_log(time, "car check")
            customs.wait_time_acc(time - car.arrive_time())
            Leave(time + randint(*car_check_time), i, car, customs)
```

海关检查站的模拟

- 现在考虑这个模拟系统的主类，它定义有关的基础结构

```
car_arrive_interval = (1, 2) # 定义两个模拟用的全局变量
car_check_time = (3, 5)      # 完全可以把它们做成可设置的形式

class Customs:
    def __init__(self, gate_num, duration):
        self.simulation = Simulation(duration)
        self.waitline = SQueue() # 汽车等待队列
        self.duration = duration
        self.gates = [0]*gate_num
        self.total_wait_time = 0
        self.total_used_time = 0
        self.car_num = 0
# 下面继续
```

- 这里定义了一个 **Simulation** 对象和一个 **Squeue** 对象，检查通道用一个 **list** 表示，还有一些用于统计模拟情况的数据成分

海关检查站的模拟

- **Customs 类的一些简单方法**

```
def wait_time_acc(self, n): self.total_wait_time += n
def total_time_acc(self, n): self.total_used_time += n
def car_count_1(self): self.car_num += 1
def cur_time(self): return self.simulation.cur_time()
def add_event(self, event): self.simulation.add_event(event)
def enqueue(self, car): self.waitline.enqueue(car)
def has_queued_car(self): return not self.waitline.is_empty()
def next_car(self) : return self.waitline.dequeue()

def find_gate(self):
    for i in range(len(self.gates)):
        if self.gates[i] == 0:
            self.gates[i] = 1; return i # 找到就设为 1 表示占用
    return None
def free_gate(self, i):
    if self.gates[i] == 1: self.gates[i] = 0 # 清除
    else: raise ValueError("Clear gate error.")
```

海关检查站的模拟

- **Customs 类的主要函数和统计数据输出：**

```
def simulate(self):
    Arrive(0, self) # initially generate one car
    self.simulation.run()
    self.statistics()

def statistics(self):
    print("Simulate " + str(self.duration) + " minutes, for "
          + str(len(self.gates)) + " gates")
    print(self.car_num, "cars pass the customs")
    print("Avarage waiting time:",
          self.total_wait_time/self.car_num)
    print("Avarage passing time:",
          self.total_used_time/self.car_num)
    i = 0
    while not self.waitline.is_empty():
        self.waitline.dequeue(); i += 1
    print(i, "cars are in waiting line.")
```

海关检查站的模拟

- 最后是 **Car** 类和输出“日志”的辅助函数

```
class Car:
    def __init__(self, arrive_time):
        self.time = arrive_time

    def arrive_time(self):
        return self.time

def event_log(time, name): pass
#    print("Event: " + name + ", happens at " + str(time))
```

- 总结一下：
  - **Simulation** 类实现了一个通用的支持离散事件模拟的框架
  - 几个事件类的 **run** 方法通过生成新事件完成模拟过程的实际控制
  - **Customs** 类实现了海关检查站模拟系统的支撑功能和主控函数

    其中用一个队列作为缓冲，保存已经到来但还不能检查汽车

二叉树的链接实现

- 前面介绍了二叉树的一些应用，主要是借助于二叉树的结构

  特别是二叉树的结点个数与高度之间的关系
- 现在考虑二叉树结构本身在 **Python** 里的实现

  定义一个二叉树的结点类和一个二叉树类

  一个结点通过链接引用到其子结点，没有子结点时可以用 **None** 值
- 下面是实现二叉树结点的类定义：

```
class BiTNode:
    def __init__(self, dat, left, right):
        self.data = dat
        self.left = left
        self.right = right
```

- 构造一棵包含三个结点的二叉树，变量 **t** 引着树根结点

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

二叉树的链接实现

- 基于 **BiTNode** 对象构造的二叉树具有递归结构，递归处理非常方便：

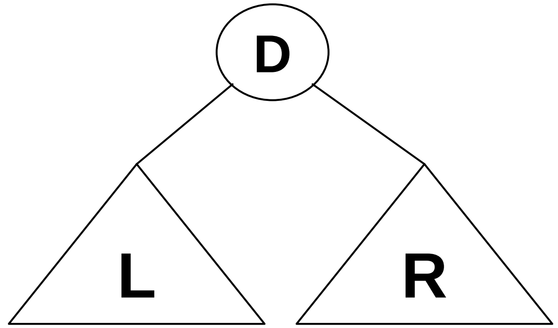
```
# 统计树中结点个数：
def count_BiTNodes(t):
    if t is None:
        return 0
    else:
        return 1 + count_BiTNodes(t.left) + count_BiTNode(t.right)

# 对结点中中保存着数值的二叉树里的所有数值求和：

def sum_BiTNodes(t):
    if t is None:
        return 0
    else:
        return t.dat + sum_BiTNodes(t.left) + sum_BiTNodes(t.right)
```

- 递归处理二叉树的操作具有统一的模式：如何处理空树情况，如何分别处理二叉树的左右子树和根，并基于这些得到对整个树的处理结果

二叉树的遍历

- 每棵二叉树有唯一的根结点，它可看作这棵二叉树的唯一标识
  - 从根出发可以找到树里所有信息（从父结点找子结点是实现的基础）
  - 因此，实际中常用根结点代表一棵二叉树，其左右子树由它们的根结点代表。这种看法在实际表示和算法设计方面都很有用
- 任何存储数据的汇集结构，都有遍历（周游，**traverse**）元素的问题
  - 遍历二叉树，就是按某种系统方式访问二叉树里的每个结点一次
  - 可基于二叉树的基本操作实现，遍历中可能操作结点里的数据
  - 很多复杂的二叉树操作需要基于遍历实现，例如找父结点
- 系统化遍历有多种可能方式，下面讨论不同的算法。二叉树（树的情况类似）遍历就像前面讨论的状态搜索，基本方式有两种：
  - 深度优先遍历，尽可能沿一条路径向前，必要时回溯
  - 宽度优先遍历，在所有路径上齐头并进

二叉树的深度优先遍历



- 深度优先遍历中要做三件事情：遍历左子树，遍历右子树和访问根结点（操作其中数据）。下面用 **L**、**R**、**D** 表示这三项工作
- 根据三项工作的不同排列方式，有**3**种常见遍历顺序（假定总先处理左子树，否则就是**6**种）：
  - 先根序遍历（**DLR**）
  - 中根序遍历（**LDR**），也称对称序
  - 后根序遍历（**LRD**）
- 由于子树也是二叉树，将各种遍历方式继续用到子树遍历中，就形成了遍历二叉树的统一方法
  - 遍历过程遇到子树为空时就结束对它的处理，转去继续做下一步工作
  - 如，先根序中遇到左子树空就去遍历右子树

二叉树深度优先遍历示例

- 按不同的深度优先方式遍历右边的树
- 先根序遍历（先访问根结点，而后以同样方式顺序地遍历左子树和右子树）得到的结点访问序列：

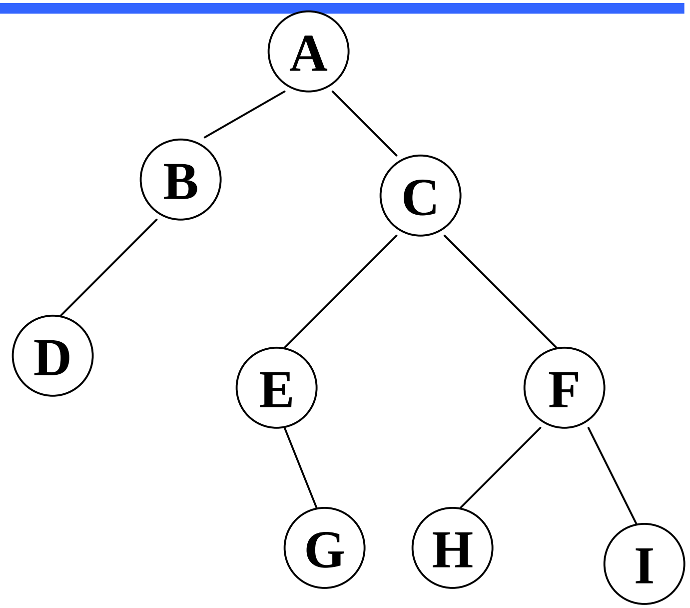
  A B D C E G F H I

- 后根序（先以同样方式遍历左右子树，而后访问根结点）

  D B G E H I F C A

- 对称序（中根序）（先以同样方式遍历左子树，而后访问根结点，最后再以同样方式遍历右子树）

  D B A E G C H F I



二叉树实例

二叉树的深度优先遍历

- 如果树中数据（或标识）唯一，通过它可以反射到遍历中经过的结点
  - 这时考虑（二叉树结点的）遍历序列就有意义
  - 按先根序遍历二叉树得到的结点序列称为其先根序列
  - 按后根序遍历二叉树得到的结点序列称为其后根序列
  - 按对称序遍历二叉树得到的结点序列称为其对称（中根）序列
- 显然对给定的一棵二叉树

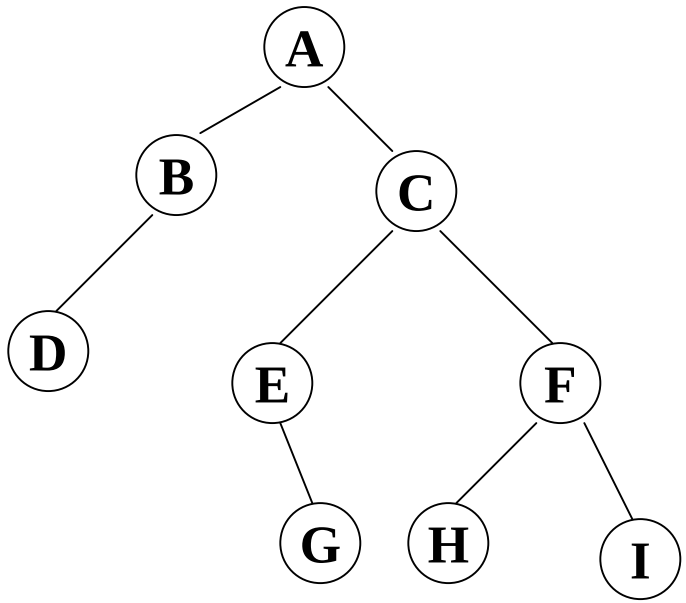
  可唯一确定其先根序列、后根序列和对称序列
- 但给定二叉树的任一种遍历序列，无法唯一确定相应的二叉树
- 命题：如果知道了一棵二叉树的对称序列，又知道另一遍历序列（无论是先根还是后根序列），就可以唯一确定这个二叉树（请自己想想）

  如果知道一棵二叉树的先根序列和后根序列，能够确定原二叉树吗？
  如果能请证明之，不能请举出反例

二叉树的宽度优先遍历

- 宽度优先遍历二叉树，就是逐层访问二叉树里的各结点
  与状态空间搜索的情况类似，这种遍历不能写成一个递归过程
- 宽度优先遍历的做法是
  - 从树根（位于第 **0** 层）开始逐层访问树结点
  - 每层都从左到右逐个访问
  - 实现算法是需要用一个队列缓存
- 宽度优先遍历又称为按层次顺序遍历
  - 这样遍历产生的结点序列称为二叉树的层次序列
  - 右边二叉树的层次序列是

    A B C D E F G H I



二叉树实例

例：算术表达式

- 前面介绍过二元算术表达式的二叉树表示：以运算符作为根，以两个运算对象作为其左、右子树。对子表达式同样处理
- 右边是一个表达式的二叉树表示
- 对这一表达式树进行先根、后根和中根序遍历得到的结点序列：

先根：*** + a b - c d**

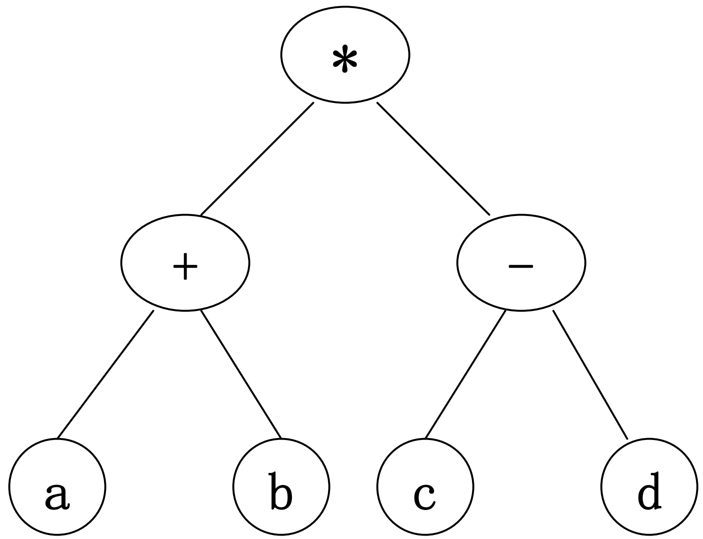
前缀表示

后根：**a b + c d - ***

后缀表示**(**波兰表示法**)**

对称序：**a + b * c - d**

中缀表示（但得到的序列里缺少表示计算顺序的括号）



表达式（a + b）*（c − d）的二叉树表示