7，字典和集合-2

- 数据检索和字典，数据规模和检索，索引和字典
- 基于表和排序表实现字典和检索，二分检索
- 散列的概念，散列表和字典，散列函数和冲突消解，实现
- 集合数据结构，集合的实现，位向量实现
- Python 的字典和集合
- 二叉排序树的概念和实现
- 最佳二叉排序树，等概率和不等概率情况，构造算法
- 支持动态操作的排序树：AVL 树
- 支持字典实现的其他树形结构简介

字典和集合

- 任何字典实现方法都可以用于实现集合

  只需把集合的元素直接存储在保存字典项（关联）的位置

  集合的最基本操作是元素与集合的关系，对应于字典查询；集合数据结构需要的创建/空集检查/加入/删除等都有字典操作与之对应

- 集合实现需要考虑的新问题是常用集合运算的实现

  求并集和交集，求相对于某个集合的补集（集合差）都是从两个已有集合得到另一个集合。集合的实现需要考虑这些操作的效率

- 例，考虑用顺序表作为集合的实现结构，实现中保持元素的唯一性
  - 建立集合，就是建立一个无重复元素的顺序表
  - 判断元素关系，就是在表中检索元素的存在性，O(n) 时间
  - 求两集合的交集，需要逐个考虑一个集合的元素，如果它也属于另一集合就将其加入结果集合。设两个集合的元素分别为 m 和 n 个，求交集的复杂性就是 O(m*n)。求并集、差集的情况与此类似

基于排序表的集合实现

- 采用排序表能显著提高集合运算的效率，如求交集可采用下面算法：

  设求交集的集合 S 和 T 由表 s 和 t 表示，结果集合 r = []
  设置变量 i = 0; j = 0，成倍作为下次检查的 s 和 t 元素的下标

```
while i < len(s) and j < len(t):
    if s[i] < t[j] :
        i += 1
    elif t[j] < s[i] :
        j += 1
    else: # s[i] = t[j]
        r.append(s[i]); i += 1; j += 1
```

  r 就是得到的交集

- 显然，主循环每次迭代至少处理掉两个集合里的一个元素，时间复杂性是 O(m+n)，比 O(m*n) 是质的提高
- 并集/差集操作均可类似定义，复杂性也是O(m+n) 。但在另一方面，插入元素操作的复杂性变成了 O(n)（需要按序插入），这是代价

基于散列表实现集合

- 也可以考虑用散列表实现集合
  - 一个集合就是一个散列表
  - 加入/删除元素对应于加入/删除关键码
  - 集合元素判断对应于关键码检索
  - 集合运算都是基于已有散列表建立新散列表，不难实现
- 集合操作的效率：
  - 完全由散列表的性质确定，具有概率性。在最佳情况下都很高效
  - 加入/删除和元素判断，操作的代价接近常量
  - 求交集/并集/差集，大致为 O(m+n) 复杂性，其中 m 和 n 为参加运算的两个集合的大小
- 如果已经有了散列表结构，基于它实现一种集合数据类型不是很困难的工作。可作为简单的编程练习，这里不进一步讨论

集合的特殊实现技术：位向量实现

- 请注意，一个元素是否属于一个集合，是一种二值判断。基于这一认识，人们提出了一种专门的集合实现技术：位向量表示
- 如果所需要的集合对象有一个公共超集 U，也就是说，需要实现的集合都是 U 的子集，就可以采用位向量技术实现这些集合，方法是：
  - 假定 U 包含 n 个元素，给每个元素一个编号作为该元素的“下标”
  - 对任何要考虑的集合 S（注意 S⊆U），用一个 n 位的二进制序列（位向量）v_S 表示 S。对元素 e∈U，如果 e∈S，令 v_S 里对应于 e（的编号，下标）的那个位取值 1，否则令该位取值 0
- 例：假设 U 是 {a, b, c, d, e, f, g, h, i, j}，10 个元素，按字母序将其对应到 0, 1, 2, ..., 9，U 的子集都能用一个 10 位的位向量表示
  - {} 用 0000000000 表示，U 用 1111111111 表示
  - S1 = {a,b,d} 对应 1101000000，S2 = {a, e, i} 对应 1000100010

  位向量表示很紧凑，空间利用率高，需要 U 的一批子集时比较适用

集合的位向量实现

- 位向量集合的元素操作
  - 加入删除元素，就是把位向量里相应的二进制位置 1 或置 0
  - 判断元素关系，对应于检查相应的二进制位是否为 1
- 位向量集合的集合运算都可以通过逐位操作实现：
  - S 和 T 的第 i 位都是 1 时 S∩T 第 i 位取值 1，否则取值 0
  - S 和 T 的第 i 位都是 0 时 S∪T 第 i 位取值 0，否则取值 1
  - S 第 i 位是 1 而 T 第 i 位是 0 时 S-T 第 i 位取值 1，否则取值 0
- 例：假设 U 是 {a, b, c, d, e, f, g, h, i, j}，其子集
  - S1 = {a,b,d}: 1101000000
  - S2 = {a,e,i}: 1000100010
  - S∩T = {a}: 1000000000
  - S∪T = {a,b,d,e,i}: 1101100010
  - S-T = {b,d}: 0101000000

位向量集合常用在操作效率要求高或存储受限的环境中。用较低级的语言实现，如用 C 语言

Python 字典和集合

- Python 语言内置类型包括字典（dict）和集合（set 和 frozenset），它们都是基于散列表实现的数据结构，采用内消解技术
  - dict 采用散列表技术实现，元素是 key-value（关键码-值）对，关键码可以是任何不变对象，值可以是任何对象
  - 建立空字典或小字典，初始创建的存储区可容纳 8 个元素
  - 负载因子超过 2/3 时换更大存储块，把字典已有内容重新散列到新存储块里。字典不太大时按当时字典中实际元素的 4 倍分配新块。元素超过 50000 时按实际元素个数的 2 倍分配新块

  上面以字典为例，集合的情况类似，许多实现代码完全一样
- 在官方 Python 系统，一些内部机制也基于字典实现，如全局/模块/类名字空间等。一个作用域里名字可能很多，用字典实现效率较高
- Python 中 dict 的关键码，set 和 frozenset 的元素都只能是不变对象。是为保证散列表的完整性（为保证数据项查询和删除的正确实现）

Python 的散列

- Python 标准函数中有一个 hash，它计算参数的散列值，hash
  - 是函数，对一个对象调用或返回一个整数或抛出异常表示无定义
  - 对数值类型有定义，保证当 a == b 时两个数的 hash 值相同
  - 对内置不变组合类型有定义，包括 str，tuple，frozenset
  - 对无定义的对象调用，例如对包含可变成分的序列，hash 将抛出异常 TypeError: unhashable type ...
- 调用时 hash 到参数所属的类里找名为 __hash__ 的方法
  - hash 有定义的内置类型都有自己的 __hash__ 方法

    类里没有 __hash__ 方法即是 hash 函数无定义
  - 自定义类里也可以定义这个方法
    - 定义该方法使这个类的对象可以存入集合或作为字典的关键码
    - 如果该类的对象可变，修改这种对象的值带来的后果自己负责

对字典的进一步考虑

- 从实现字典的角度看，前面研究的两种结构各有优点和缺点：
  - 基于线性表的字典结构简单，易于实现。但总存在低效操作，因此不适合用于实现大型字典
  - 散列字典操作效率高（概率的），对关键码类型无特殊要求，应用广泛。但没有确定性的效率保证，不适合效率要求严格的环境
- 两种结构都基于连续的大存储块实现，管理比较方便。但

  需要大块连续存储，动态变化不方便，也难以用于实现巨大的字典
- 要支持方便的动态变化，就应该考虑链接结构。另一方面，链接结构也能支持用大量较小存储块实现能存储大量信息的容器，包括字典
- 分析这些情况，很容易想到树形结构，它们
  - 可以用链接方式实现，容易处理动态插入/删除元素的操作
  - 树中平均路经的长度可以达到结点个数的对数，有可能实现高效操作（只要维持良好的树形结构）

基于树实现字典

- 人们研究用于实现字典的树形结构，主要的考虑还有
  - 支持大型字典的需要，例如数据库系统系统
  - 支持高效的结构调整，保证长期工作的系统仍能维持良好性能
  - 有可能较好地利用计算机系统的存储结构，例如，很好利用内存和外存（磁盘、磁带等），以及缓存等多层次存储结构
  - 用于为大型数据集合建立索引，提高复杂（复合）查询的效率
- 下面主要讨论最简单的树形结构：二叉树

  二叉树也有多种不同的使用方式

  其他树形结构将在后面做简单介绍
- 在这里，应特别注意二叉树（和其他树形结构）的特点（正反两面）
  - 如果树结构“良好”，最长路径长度与结点个数成对数关系
  - 如果树结构“畸形”，最长路径长度可能与结点个数成线性关系

二叉排序树

- 采用（链接实现的）二叉树作为字典的存储结构
  - 可能得到较高的检索效率
  - 采用链接式的实现方式，数据项的插入、删除操作比较灵活方便
- 基本想法：
  - 在二叉树结点里存储信息，设法组织好存储方式
  - 利用树的平均高度（通常）远小于树中结点个数的性质
  - 使检索能沿着树中路径（父子关系）进行，以获得高检索效率
- 下面介绍二叉排序树 **(Binary Sort Tree)**，是一种存储数据的二叉树
  - 可用于保存关键码有序（存在明确定义的序关系）的数据
  - 树中数据的存储和使用都利用了数据（或关键码）的序
  - 可以作为一种基于二叉树结构实现字典的方法

二叉排序树：概念

- 定义：二叉排序树或者为空；或者是具有下列性质的二叉树：
  - 其根结点保存着一个数据项（及其关键码）
  - 如果其左子树不空，则左子树里所有结点保存的（关键码）值均小于它的根结点保存的（关键码）值；
  - 若其右子树不空，则右子树上所有结点保存的（关键码）值均大于它的根结点保存的（关键码）值；
  - 左右子树（如果存在）也是二叉排序树
- 二叉排序树是一种递归结构
  - 对二叉排序树做中序遍历，得到的是按关键码排序的“上升”序列
  - 如果存在重复关键码，关键码一样的不同数据项的前后顺序不确定
  - 另一方面，同样关键码的数据必定位于按中序遍历的相邻位置
  - 二分法检索的判定树就是一棵二叉排序树

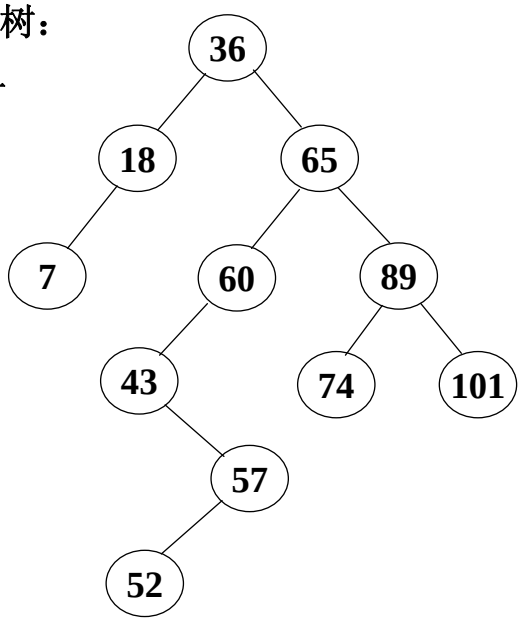
二叉排序树：例

考虑关键码的序列：

K= [36, 65, 18, 7, 60, 89, 43, 57, 101, 52, 74]

右边是保存这些数据的一棵二叉排序树：

显然，一集数据对应的二叉排序树不唯一



- 下面讨论中将集中关注二叉排序树本身的结构，忽略关键码以外的数据项部分（它们与结构无关）
- 二叉排序树也可作为索引结构，实际数据另行存储，从树中可以找到数据存储的位置信息
- 下面讨论中将不区分二叉排序树是作为字典的索引还是字典本身

对其他结构的讨论也如此

二叉排序树：实现

- 二叉排序树可以用任何能实现二叉树的技术实现
  - 为了支持树结构的动态变化，最常见的是采用链接结构
  - 可以基于前面的 **BiTNode** 类实现二叉排序树
- 首先考虑二叉排序树的检索算法。由于树（子树）根数据总把树中数据划分为两组，用检索关键码与之比较，就知道下步应该到哪棵子树去检索（递归的）。下面函数用一个循环实现这个过程：

```
def bt_search(btree, key):
    bt = btree
    while bt is not None:
        entry = bt.data
        if key < entry.key: bt = bt.left
        elif key > entry.key: bt = bt.right
        else: # key == entry.key
            return entry.value
    return None
```

二叉排序树字典类

- 现在基于二叉排序树的思想定义一个字典类，并分析和实现各种重要算法。类的基础结构采用前面定义的二叉树结点类和关联类：

```
from BiTree1 import BiTNode, print_BiTNodes
from SStack import *
from Assoc import Assoc

class DictBiTree:
    def __init__(self): self._root = None

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

    def inorder(self): # 定义一个中序遍历的生成器方法
        t, s = self._root, SStack()
        while t is not None or not s.is_empty():
            while t is not None:
                s.push(t); t = t.left
            t = s.pop(); yield(t.data); t = t.right

    def search(self, key): ... # 前面检索函数的简单修改
```


二叉排序树：插入操作

- 插入操作的基本要求是加入新数据项并维持二叉树的完整性

  需要找到加入新结点的正确位置并将结点正确连接到树上

  如果插入操作中遇到与检索关键码相同的关键码，下面用替换关联值的方式处理（这样就不会出现重复关键码）

- 查找位置就是用关键码检索，基于检索插入数据的基本算法：
  - 如果树空，直接建立一个包括新关键码和关联值的树根结点
  - 否则搜索新结点的插入位置，沿子结点关系向下
    - 遇到应该向左子树而左子树为空，或者应该向右子树而右子树为空，就是找到了新字典项的插入位置
    - 遇到结点里的关键码等于被检索关键码时，直接替换关联值
- 为算法描述的方便，修改 BiTNode 的 __init__ （定义时就该这样做）

```
def __init__(self, data, left = None, right = None): ... ...
```

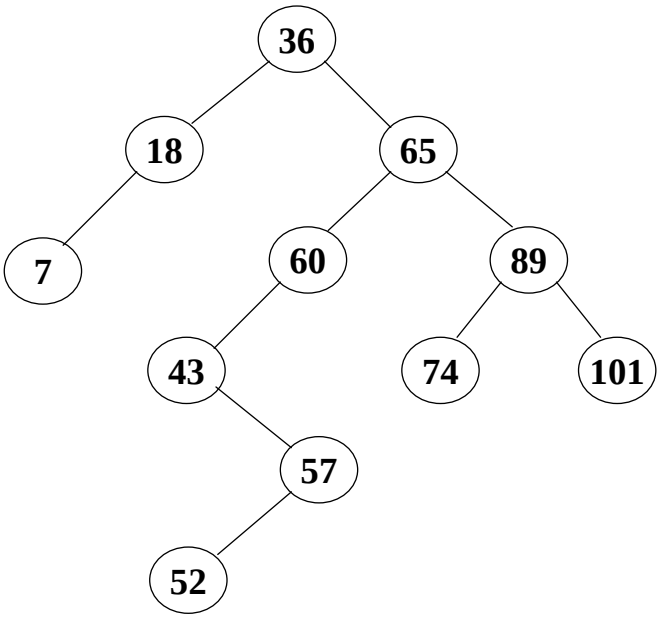
二叉排序树：插入操作

```
def insert(self, key, value): # DictBiTree 类的插入方法
    bt = self._root
    if bt == None:
        self._root = BiTNode(Assoc(key,value)); return
    while True:
        entry = bt.data
        if key < entry.key: # 考虑左子树
            if bt.left == None:
                bt.left = BiTNode(Assoc(key,value)); return
            bt = bt.left
        elif key > entry.key: # 考虑右子树
            if bt.right == None:
                bt.right = BiTNode(Assoc(key,value)); return
            bt = bt.right
        else: # 找到相同关键码
            bt.data.value = value; return
```


通过插入建立二叉排序树

- 从空树出发经过一系列插入，可以生成一棵二叉排序树。

  例，K = [36, 65, 18, 7, 60, 89, 43, 57, 101, 52, 74]

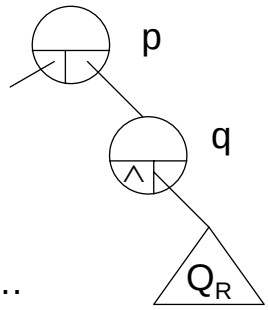


二叉排序树：删除操作

- 从二叉排序树中删除结点的操作比较复杂，操作分两个阶段：
  1. 首先检索，找到需要删除的结点（假设找到后用 **q** 指向）
  2. 删除结点并做必要的处理（维护二叉排序树的性质）
- 关键问题：工作中既要删除结点，又要维护二叉排序树的完整性（包括：树结构完整；非删除结点仍存在；整个树仍然是二叉排序树）
  - 被删结点可能出现在树中任何位置
  - 保持树中有序 **iff** 二叉树的中序周游序列不变！（少了结点 **q**）
  - 下面分 **3** 种情况讨论（设 **q** 是被删除结点）
- 情况 **1**：**q** 是叶结点
  - 删除叶结点不会破坏整棵树的结构和结点之间的序关系
  - 这种情况可以直接删除，只需修改其父结点的引用关系

二叉排序树：删除操作

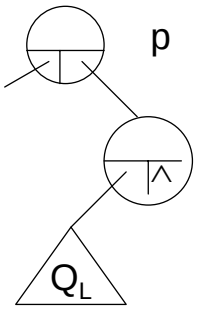
- 情况 **2**：**q** 只有左子树 Q_L 或只有右子树 Q_R
  - 只要用 **q** 的唯一子结点 Q_L 或 Q_R 的根结点代替 **q** 即可
  - 假设 **p** 指向 **q** 的父结点（**p, parent**）



中序序列：

… p q Q_R …

删除后：**… p Q_R** …



中序序列

… p Q_L q …

删除后：**… p Q_L** …

- 显然删除 **q** 后中序周游序列没有变（只是是少了结点 **q**）
- 情况 **1/2** 可统一处理：**q** 无左子树时让其父到 **q** 指针改到 **q** 的右子树...

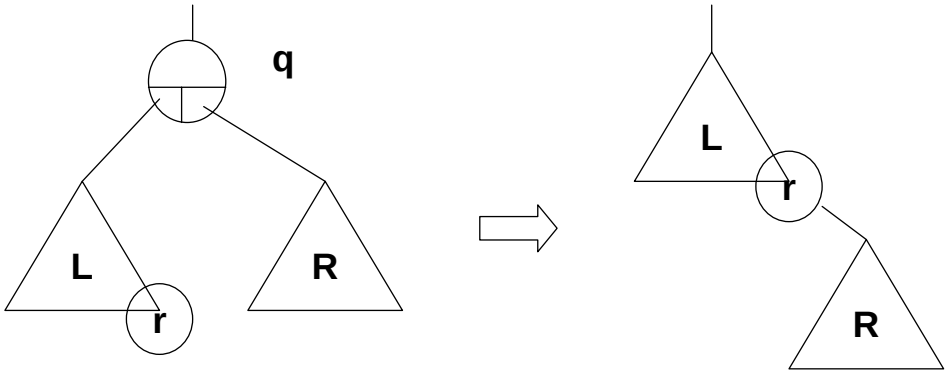
二叉排序树：删除操作

- 情况 3，若 q 的左右子树均不空，删除 q 前中序周游序列为

  … L (r) q R …

  删除 q 后要保持其它元素的中序顺序不变，下面介绍一种做法

- 删除方法：用 q 的左子树的根结点代替 q ，q的右子树作为 (r) (q 的左子树中的最大结点) 的右子树



- 貌似删除后的结果树形态“可能”不好，“可能”使树变高，检索效率“可能”恶化。但也未必，依赖于实际树的情况

二叉排序树：删除算法实现概要

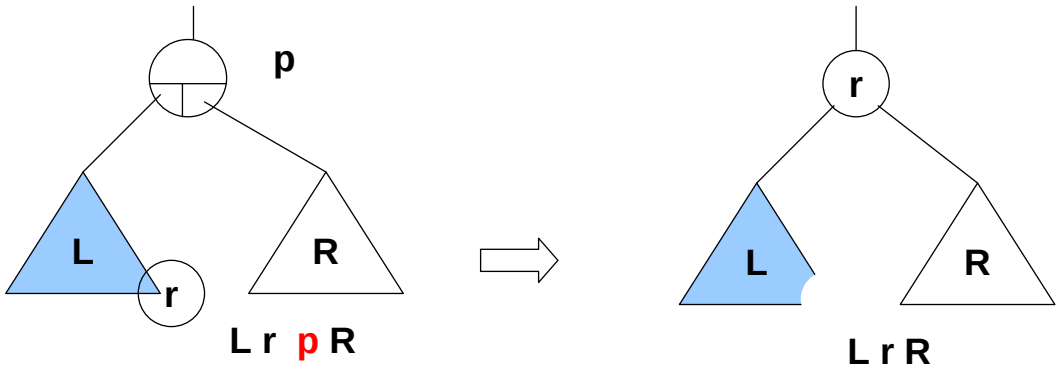
- 根据上面三种情况适当调整，实现删除操作。概要如下：
- 首先检索，找到要删除的结点 q（其关键码等于要删除的 key）

  维持 p 为 q 的父结点，如果 q 是根结点，p 值为 None
- 如果这样的 q 不存在，删除完成，操作结束
- 如果 q 没有左子树，用其右子树取代 q 原来的位置。根据 p 和 q 的情况决定修改的方式，分别处理后结束：
  - 如果 q 是根结点，将其右子结点作为新的根结点
  - q 是 p 的左（右）子树时，将 q 的右子树作为 p 的左（右）子树
- 找到 q 的左子树的最右结点 r
  - 把 q 的右子树作为 r 的右子树
  - 用 q 的左子树取代 q 的位置，操作与上面相同

```
def delete(self, key): # DictBiTree 类的删除方法，用方法前面
    p, q = None, self._root # 维持 p 为 q 的父结点
    while q is not None and q.data.key != key:
        p = q
        if key < q.data.key: q = q.left
        else: q = q.right
    if q == None: return # key 不在树中
    # q 是关键码为 key 的结点且 p 是其父，或者 q 是 self._root
    if q.left == None: # q 没有左子树
        if p == None: self._root = q.right # q 是 self._root 时修改它
        elif q == p.left: p.left = q.right # 修改 p 的相应子结点引用
        else: p.right = q.right
        return
    r = q.left
    while r.right is not None: # 找到 q 子树的最右结点
        r = r.right
    r.right = q.right
    if p == None: self._root = q.left # q == self._root
    elif p.left == q: p.left = q.left
    else: p.right = q.left
```


二叉排序树：删除的另一方法

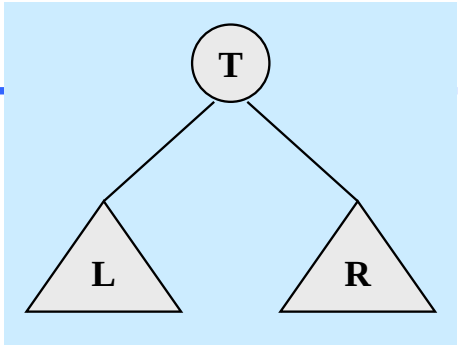
- 删除方法 2：
  - 按对称序周游 p 的左子树，找到关键码最大的结点 r，把 r 从树中取下（叶结点简单，非叶用其左子女代替）
  - 用结点 r 代替被删除结点p（可以把结点 r 里的信息复制到 p 去）



这种方法的实现作为课下自己练习，实现可能复杂一点

注：这张幻灯片原来就有，因为只是另一方法，课堂没讲。下课后一位同学通过自己思考也提出了这种做法（值得表扬）。现给出供大家参考。
这种方法的良好性质：保证删除操作不会使二叉树增高

二叉排序树：操作性能

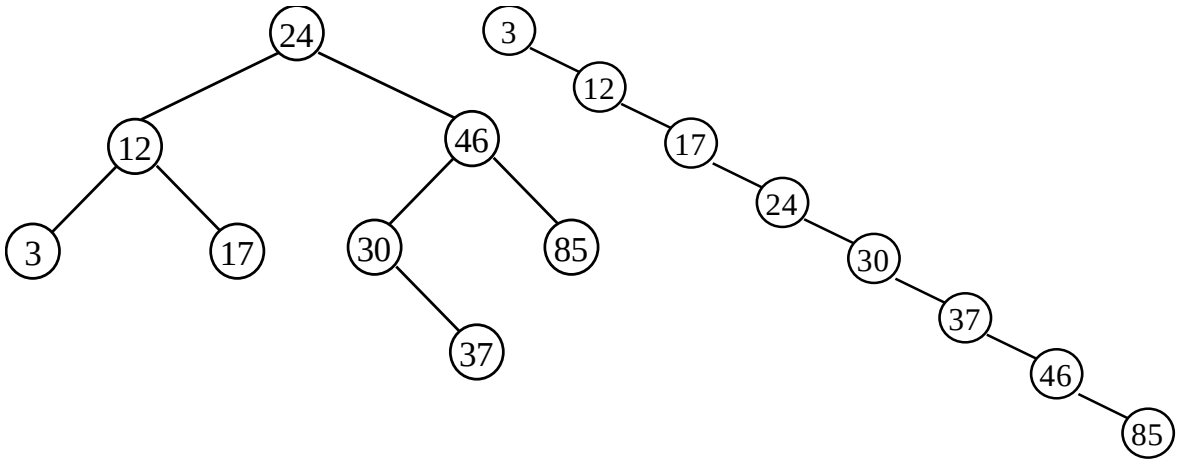


- 在二叉排序树中检索
  - 操作的效率依赖于树结构，每做一次比较后进入树的下一层
  - 如果树结构良好，高度与树结点数成对数关系，检索时间开销是 **O(log n)**
  - 如果树结构畸形，检索的时间开销可能达到 **O(n)**（最坏时间复杂度）
- 在随机的情况下，平均性能也是 **O(log n)**

  随机假设：**n** 个结点关键码不同；**n** 个结点的任何序列可生成二叉排序树。计算每棵二叉排序树上的所有检索的平均长度，求出平均值
- 算法分析细节可参考：**“Introduction to Algorithms”**，高教出版社影印本，**265**页。其他许多关于算法的书籍上也有
- 二叉排序树检索操作的空间复杂度是 **O(1)**，不需要复杂的辅助结构

最佳二叉排序树

- 显然，同样关键码可能构成出很多不同二叉排序树。例如



- **n** 个不同的关键码，按不同顺序插入一棵空二叉排序树，形成的二叉排序树不同，总共可以形成 **n!** 棵二叉排序树
- 最佳二叉排序树是以检索效率为追求目标定义的概念。“最佳二叉排序树” 就是平均检索效率最高（平均检索路径最短）的二叉排序树

最佳二叉排序树

- 基于平均检索长度评价二叉排序树的优或劣。需要参考
  - 各个关键码出现的概率
  - 成功和失败（关键码不存在）检索
- 概念：扩充二叉排序树的对称序列（中序遍历序列）：

  按中序遍历扩充二叉树得到的结点标记序列。其中内外部结点交叉排列，第 i 个内部结点位于第 i-1 个外部结点和第 i 个外部结点之间

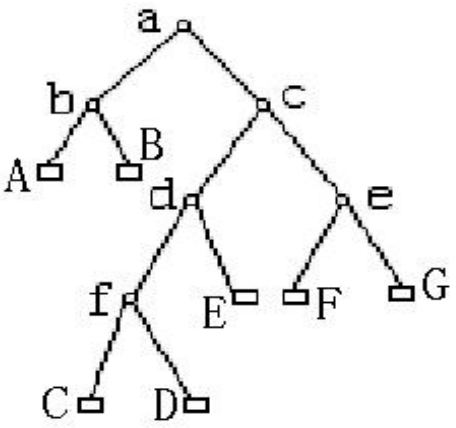
扩充的二叉排序树表示了一个字典的可能关键码集合：

内部结点代表已有元素的关键码

外部结点代表介于两个相邻内部结点的关键码之间的那些关键码

右图的中序周游序列：

AbBaCfDdEcFeG



最佳二叉排序树：平均检索长度

- 在扩充二叉排序树里，关键码的平均检索长度由下面公式给出

$$E(n) = \frac{1}{w}\left[\sum_{i=0}^{n-1} p_i(l_i+1) + \sum_{i=0}^{n} q_i l'_i\right]$$

其中 l_i 是内部结点 i 的层数，l'_i 是外部结点 i 的层数

p_i：检索内部结点 i 的关键码的频率，确定内部结点需比较层数加一次

q_i：被检索关键码属于外部结点 i 代表的关键码集合的频率

p_i，q_i 看作相应结点的权

其中

$$w = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i$$

这样，p_i/w 就是检索内部结点 i 的关键码的概率

q_i/w 是被检索的关键码属于外部结点 i 的关键码集合的概率

最佳二叉排序树

- 最佳二叉排序树使检索的平均比较次数达到最少

  也即，它是使 $E(n)$ 的值达到最小的二叉排序树

- 问题：给定了一组数据及其分布，如何构造最佳二叉排序树？

先考虑简单情况：所有结点的检索概率相等

$$\frac{p_0}{w}=\frac{p_1}{w}=\cdots=\frac{p_{n-1}}{w}=\frac{q_0}{w}=\frac{q_1}{w}=\frac{q_2}{w}=\cdots=\frac{q_n}{w}=\frac{1}{2n+1}$$

$$\begin{aligned} E(n) &= \frac{1}{2n+1}\left[\sum_{i=0}^{n-1} p_i(l_i+1)+\sum_{i=0}^{n} q_i l'_i\right] \\ &= (IPL+n+EPL)/(2n+1) \\ &= (2\cdot IPL+3n)/(2n+1) \end{aligned}$$

$$\text{其中 } IPL=\sum_{k=1}^{n}\lfloor\log_2 k\rfloor=(n+1)\lfloor\log_2 n\rfloor-2^{\lfloor\log_2 n\rfloor+1}+2$$

内部路径长度 IPL 最小时这棵树达到最佳，也就是说，最低的树最好

最佳二叉排序树：简单情况的构造

- 一种构造方法：递归构造（左右子树结点均分的方法）

  设表 a 里是按关键码排序的一组字典项

  0，令 low = 0, high = len(a)-1

  1，m = (high + low)/2;

  2，把 a[m] 存入被构造的二叉排序树的根结点 t，递归地：

  - 将基于 a[low:m-1] 构造的二叉排序树作为 t 的左子树
  - 将基于 a[m+1:high] 构造的二叉排序树作为 t 的右子树
  - 切片为空时直接返回 NULL，表示空树

- 构造的时间代价：
  - 构造最佳二叉排序树的算法的构造部分的复杂性为 O(n)
  - 从有关排序的研究可知，n 个关键码的排序，复杂性为 O(n log n)
  - 因此整个构造过程的复杂性也为 O(n log n)

最佳二叉排序树：简单情况的算法

- 下面类的初始化方法从参数构造最佳二叉排序树，采用第二种方法：

  其参数是任何的序列，要求元素可以比较大小（< 运算符有定义）

```
class DictOptBTree(DictBTree):
    def __init__(self, seq):
        DictBTree.__init__(self)
        data = sorted(seq)
        self._root = DictOptBTree.buildOBT(data, 0, len(data)-1)

    @staticmethod
    def buildOBT(data, start, end): # 注意，这里没有真做切片
        if start > end: return None
        mid = (end + start)//2
        left = DictOptBTree.buildOBT(data, start, mid-1)
        right = DictOptBTree.buildOBT(data, mid+1, end)
        return BiTNode(Assoc(*data[mid]), left, right)
```

- 其他操作继承自 DicBTree。简单插入新数据项不能保证最佳性质