- 数据结构与算法：python语言描述
  - 第一章：绪论
    - 1.1 计算机问题求解
      - 区分问题和实例
        - 问题是需求，是总，实例是问题的具体体现，是分
        - 编写程序是为了解决问题，程序的每次执行能处理该问题的一个实例
      - 程序开发过程
        - 分析阶段：需求分析，弄清问题，将含糊的需求转化为详细的问题描述
        - 设计阶段：经过分析阶段得到的严格的问题描述，仅仅只是描述，不具有操作性，计算机不能实际操作。这一阶段就是设计出一个解决问题的抽象计算模型
        - 编码阶段：用具体语言来实现上阶段的抽象计算模型
        - 检查测试阶段：检查并测试程序有没有bug，能不能运行
        - 测试/调试阶段：测试这个程序是不是满足功能需要
    - 1.3 算法和算法分析
      - 三个基本概念
        - 问题：一个问题W是需要用计算求解的一个具体需求，是研究和实现算法的出发点
        - 问题实例：问题W的实例w是该问题的一个具体例子，通常可以通过一组具体的参数设定；一个问题是其所有实例的共性
        - 算法：解决问题W的算法A，是对一种计算过程的严格描述。对W的w，实施A描述的计算过程，就能得到w的解
          - 算法的性质：
            - 有穷性：算法必须在有限的长度内解决问题
            - 可行性：算法可以达到目的
            - 确定性：算法的每一步都要有清晰的定义
            - 输入：算法要有0个或多个输入
            - 输出：算法要有1个或多个输出
          - 常见算法设计模式：
            - 枚举法：根据具体问题枚举出各种可能，暴力破解，一般复杂度还蛮高
            - 贪心法：根据问题的信息尽可能作出部分解，基于部分解逐步扩充得到完整的解。未必能得到最好的解
            - 分治法：divide and conquer，将大问题分解成小问题，逐个击破，得到大问题的解
            - 回溯法（搜索法）：如果问题很复杂，没有清晰的求解思路，那么用试探的方式求解，选择一个方向求解下去，如果不行就回溯到前面的步骤，选择别的方向。就像Max的能力一样
            - 动态规划法：如果问题的求解需要靠前面的步骤求解积累的信息的话，在后面的步骤根据已知信息，动态选择已知的最好求解路径，并进一步积累信息
            - 分支界限法：回溯法的改良，如果能在试探过程中获得一些信息来排除掉一些路径的话，缩小求解空间，加速问题求解
            - 注意：
              - 以上模式只能借鉴，不应教条化
              - 这些模式并不相互隔绝和排斥，多种模式可以有机组合来解决问题
          - 评价算法：
            - 前提：
              - 每一次基本操作只耗费一个单位的时间
              - 计算设备存储数据用一组基本单元，每个单元只能保存固定的一点有限数据
            - 算法分析：针对一个具体的算法，用一种函数关系，以问题规模n为参数，反映该算法在处理规模n的问题实例时需要付出的开销
              - 目的：推导出算法的复杂度，最关键的是构造和求解递归方程
            - 算法开销
              - 三种考虑：
                - 最好情况，一般价值不大，最好情况出现的可能很小
                - 最坏情况：多数考虑这个
                - 平均情况：有时考虑这个
              - 开销度量：
                - 重要的是量级和趋势，在规模很大的情况下，常量因子可以忽略不计
                - 用渐近复杂度函数描述代价，常用的渐近复杂度函数（不同量级）：
                  - O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), O(2^n)
              - 时间代价
                - 基本计算规则：
                  - 基本操作，O(1)
                  - 加法规则，即顺序复合。如果一个算法是多个部分的复合，算法的复杂度是每个部分的复杂度之和
                  - 乘法规则，即循环结构。循环里面套循环，复杂度乘起来
                  - 取最大规则，即分支结构。取复杂度最大的那个分支的复杂度
              - 空间代价
    - 1.4 数据结构
      - 数据结构及其分类
        - 三个概念：
          - 信息：日常中的一切都是信息
          - 数据：经过编码的信息，即信息的编码表示，指计算机能够处理的符号形式的总和
          - 数据元素：最基本的数据单位
          - 数据结构：数据之间的关联和组合的形式
            - 抽象定义数据结构：D = (E, R)
              - E是数据结构D的元素集合，是某个数据集合的一个有穷子集
              - R是D中的E这些元素之间的某种关系
            - 结构性数据结构和功能性数据结构：
              - 结构性数据结构规定了元素之间的相互关系，功能性数据结构不会对元素之间的关系提出任何结构性要求，而是提出功能性要求，最基础的要求比如：支持元素的存储和访问
              - 典型结构性数据结构：
                - 集合：一堆元素的集合，元素之间没有关系
                - 线性结构：元素之间有一种明确的先后关系，即顺序关系
                - 层次结构：元素分属不同的层次，一个上层元素关联着一个或多个下层元素，关系R形成一种明确的层次性，只从上层到下层，通常允许跨层次

- 树形结构：像树一样，最上层的元素是根元素，除了根元素的每个元素有且仅有一个上层元素与之关联
- 图结构：元素之间可以有任意复杂的相互关系

功能性数据结构：

- 栈
- 队列
- 优先队列
- 字典等

第二章：抽象数据类型和python类

2.1 抽象数据类型

- 抽象数据类型（Abstract Data Type，ADT）是一种思想和方法，用于设计和实现程序模块，ADT的基本思想是抽象

抽象的思想：

计算层面的抽象：

- 包括接口和实现
- 设计者应该通过一套接口来给出程序的可用功能，接口包括函数名字和对参数的要求，功能可以随意实现；使用者使用时，只需要看接口是否满足需要，保证调用时符合函数头部的要求，不需要知道功能实现任何细节

数据层面的抽象

- 能围绕一类数据类型建立程序组件，将该类数据的具体表示和相关操作的实现包装成一个整体
- 同样包括接口和实现
- 模块接口提供使用它功能的所需信息，不涉及任何细节；模块的实现者要通过模块内部的一套数据定义和函数定义，实现接口的所有功能

ADT的实现：

- 将数据和操作隔离开

把数据定义为抽象对象的集合，只为它们定义可用的合法操作，不暴露内部的细节，基本操作包括：

- 构造操作：基于已知的信息，产生出这种类型的一个新对象
- 解析操作：从一个对象取得有用的信息，其结果反映了被操作对象某方面的特性，但结果不是本类型的对象
- 变动操作：修改被操作对象的内部状态

ADT的描述：

- 一个ADT描述由一个头部和按一定格式给出的一组操作描述构成
- ADT的头部给出类型名，最前面写关键词'ADT'
- 操作的形式描述给出操作的名字、参数的类型和参数名
- 各操作的实际功能用自然语言描述

2.2 Python类与对象

- python中利用类定义来实现ADT，类定义机制用于定义程序里需要的类型

类的定义说明：

- 类中的方法的第一个参数总是self，表示实际使用时调用的对象
- 类中定义的名字（数据属性或函数属性名），作用域不会延伸到类的内部嵌套作用域，所以想在类中的函数定义里应用这个类的属性，一定要采用基于类名的属性引用方式，即Class.name，而不能直接name
- 由下划线_开头的属性名和函数名都当做内部使用的名字，不能在类之外使用
- 由两个下划线开头，但不由两个下划线结束，无法在类之外采用属性访问这个名字的方式找到

由两个下划线开头，两个下划线结束的是特殊名字，比如python为所有运算符都定义了特殊方法名

- __str__：在类中定义一个把该类的对象转换到字符串的方法
- __doc__：该类的文档串
- - : __sub__
- + : __add__
- * : __mul__
- / : __truediv__
- % : __mod__

类中的方法分为对象方法、静态方法、类方法

区别：

- 对象方法不能通过类名调用，其他两个可以

静态方法：

什么时候用：

- 该方法不依赖任何对象
- 该方法是类的实现时需要的一种辅助功能，即，它是一个局部的功能

- 好处：不需要实例即可使用该方法，即，这个方法不需要依赖对象本身就可以用。而且多个实例共享此静态方法
- 静态方法描述时需要最前面写一行@staticmethod
- 静态方法参数表中没有self参数，于是静态方法不会自动使用self参数，导致了静态方法不能访问类变量和实例变量，所以参数表里必须为每个形参提供实参

类方法：

- 类方法描述时需要最前面写一行@classmethod
- 类方法的第一个参数都是类对象cls而不是实例对象
- 类方法执行时，调用该方法的类，将自动约束到类方法的cls参数，可以通过这个参数访问类的其他属性
- 通常用类方法实现与本类的所有对象相关的操作

类对象支持的操作：

- 属性访问：.
- 实例化：创建这个类的实例对象

实例对象：

初始化：在创建类的实例对象时自动完成适当的初始化：__init__

- 类中的__init__方法用于构造本类的新对象
- __init__方法的self参数总表示当前正在创建的对象

- 用类中的除了__init___的其他函数查看或修改实例对象的状态

使用：

- 类C中有方法函数m和实例对象o，o要用C中的m，python就会创建一个方法对象，把o的m约束到这个方法对象中，后面用到这个方法对象时，o就会成为m的第一个实参，在m的定义中通过self访问，实现o的属性访问

继承

基于类和对象的程序设计（面向对象的程序设计）的基本工作：

- 定义程序需要的类

- 创建这些类的对象（实例）
- 调用对象的方法完成计算，包括对象间的信息交换

作用：

- 基于已有类定义新类，通过继承来复用已有类的功能
- 建立一组类之间的继承关系

- 如果一个类定义中美说明基类，那么该类就自动以object类作为基类；即任何类都是object的直接或间接派生类

派生类：通过继承定义出的新类

- 通过继承定义出的新类
- 通常需要重新定义__init__函数，完成该类实例的初始化

派生类的实例对象里包含基类实例的所有数据属性，在创建派生类的对象时，需要对基类对象的所有数据属性进行初始化

- 直接调用基类的__init__方法，利用它为正创建的实例中那些在基类实例中也有的数据属性设置初值，它完成派生类实例中基类的那部分属性的初始化工作

```
class DerivedClass(Baseclass):
        def __init__(self, ...):
                Baseclass.__init__(self, ...)
    ...
```

- super()：用在派生类的方法定义中，这个函数要求从这个类的基类开始做属性检索，而不是从这个类本身开始查找

第三章：线性表

3.1 线性表的概念和表的ADT

- 定义：一个线性表时某类元素的一个集合，还记录着元素之间的一种顺序关系

线性表的操作：

- 创建线性表，比如创建空表或非空表
- 动态改变表的内容，比如插入或删除
- 涉及一个或两个表的操作，比如表的组合
- 涉及对表中每个元素进行的操作

ADT：

- List(self)
- is_empty(self)
- len(self)
- prepend(self, elem)
- append(self, elem)
- insert(self, elem)
- del_first(self)
- del_last(self)
- def(self, i)
- search(self, elem)
- forall(self, op)

3.2 顺序表

基本操作：

创建和访问操作

- 创建空表
- 简单判断操作：判表空或表满，O(1)
- 访问给定下标操作：O(1)
- 遍历操作：O(n)
- 查找给定元素第一次出现的位置：O(n)
- 查找给定元素在位置k后第一次出现的位置：O(n)

变动操作：加入元素

- 尾端加入新数据项：O(1)
- 新数据项存入第i个单元

变动操作：删除元素

- 尾端删除：O(1)
- 非保序定位删除：O(1)
- 保序定位删除：O(n)，涉及到移动一些元素

结构：

一体式结构：顺序表存储表信息的单元和元素存储区是连续的，安排在一起的

- 优点：比较紧凑，整体性强，易于管理，访问元素O(1)
- 缺点：表对象的大小不统一，创建后元素存储区就固定了

分离式结构（Python的List）：表对象只保存整个表有关的信息，实际元素存放在另一个独立的元素存储区对象中，通过链接的方式和基本表关联

- 优点：表对象的大小统一，不同表对象可以关联不同大小的元素存储区
- 缺点：创建、管理工作复杂

3.3 链表

单链表

基本操作：

- 创建空链表：O(1)
- 删除链表：O(1)
- 判断表是否为空：O(1)
- 判断表是否为满

加入元素：

- 表首插入：O(1)
- 表尾插入：O(n)
- 插入指定位置：O(n)

删除元素：

- 删除表首元素：O(1)
- 删除表尾元素：O(n)
- 删除指定位置元素：O(n)

- 扫描、定位和遍历
  - 按位置定位：O(n)
  - 按元素定位：O(n)
- 求表的长度：O(n)
- 循环单链表
- 双链表
- 循环双链表

第四章：字符串

4.1 字符集、字符串和字符串操作

- 字符集：有穷的一组字符构成的集合
  - 字符序：字符集上的一种确定的序关系，即字符上定义的一种顺序
- 字符串：一类特殊的线性表，表中元素取自特定的字符集
- 相关概念和操作：
  - 长度和位置
  - 字符串相等
  - 字典序
  - 字符串拼接
  - 子串关系
  - 前缀和后缀等
- ADT
  - String(self, sseq)
  - is_empty(self)
  - len(self)
  - char(self, index)
  - substr(self, a, b)
  - match(self, string)
  - concat(self, string)
  - replace(self, str1, str2) # 将本字符中的子串str1都替换成str2

4.2 字符串实现

- 使用分离式顺序表形式
- python中的字符串
  - str是个不变类型，其对象创建后的内容和长度都不会变化，你对字符串做的变化的操作其实只是返回了新的字符串

4.3 查找子串

- 朴素的方式
  - 复杂度O(m x n)
- KMP算法
  - 思路：匹配中不回溯
  - 复杂度O(m + n)

4.4 字符串匹配

- 一种推广的串匹配，规定一种模式，这种模式可以匹配一类的串，普遍使用正则表达式
- python正则表达式
  - re包
    - 操作
      - pattern表示模式串，string表示被处理的字符串，repl表示替换串
        - 从pattern串生成正则表达式对象：re.compile(pattern, flag=0)
        - 检索：re.search(pattern, string, flag=0)
          - 在string里检索与pattern匹配的子串，找到就返回一个match对象，找不到返回None
        - 匹配：re.match(pattern, string, flag=0)
          - 在string中检查是否存在与pattern匹配的前缀，找到就返回match对象，找不到返回None
        - 分割：re.split(patter, string, maxsplit=0, flags=0)
          - 以pattern作为分割串讲string分段，参数maxsplit指明最大分割数，为0时表示要求处理完整个string
        - 找出所有匹配串：re.findall(pattern, string, flags=0)
          - 返回一个表，表中元素是按顺序给出的string里与pattern匹配的各个子串（从左到右，不重叠）
        - re.fullmatch(pattern, string, flags=0)
          - 如果整个string和pattern匹配成功，就返回记录匹配成功信息的match对象，不成功返回None
        - re.finditer(pattern, string, flags=0)
          - 与findall类似，但是不返回表，而是返回一个迭代器，与其他迭代器一样使用
        - re.sub(pattern, repl , string, count=0, flags=0)
          - 生成替换结果的串，string中与pattern匹配的各个非重叠子串按顺序用另一参数repl替换
          - repl是字符串时，直接替换
          - repl是match对象时，用该函数对匹配得到的match对象调用的返回值代换被匹配的子串
      - match对象，匹配成功时返回，记录了所完成的匹配中得到的信息，根据需要自取：
        - 取得被匹配的子串：mat.group()
        - 在目标串里的匹配位置：mat.start()
        - 目标串里被匹配子串的结束位置：mat.end()
        - 目标串里被匹配的区间：mat.span()
          - 得到匹配的开始和结束位置形成的二元组
        - mat.re：取得match对象所做匹配的正则表达式对象
        - mat.string：取得match对象所做匹配的正则表达式目标串
      - 正则表达式对象（complile生成）：方括号括起的部分可选
        - 检索：regex.search(string[, pos[, endpos]])
          - 检索给定的目标串string，可以指定开始和结束位置

- 匹配：regex.match (string[, pos[, endpos]])
  - 检查给定的串string是否有与regex匹配的前缀，可以指定开始和结束位置
- 完全匹配：regex
  - 检查string里由指定范围构成的子串是否与regex匹配
- regex.findall(string[, pos[, endpos]])
- regex.finditer(string[, pos[, endpos]])

- 基本概念：
  - 原始字符串：r'xxx'
  - 元字符：re包规定的特殊字符，包括
    - 通配符：.
      - 能匹配任意一个字符
    - 行首描述符：^
      - 以这个符号开头的模式，只能与一行的前缀子串匹配
      - 串首描述符：\A，开头的模式只与整个被匹配串的前缀匹配
    - 行尾描述符：$
      - 以这个符号结尾的模式，只能与一行的后缀子串匹配
      - 串尾描述符：\Z，结束的模式只与整个被匹配串的后缀匹配
    - 重复描述符：*
      - 要求与模式匹配的字符串0次或多次重复
      - 该运算符做贪婪匹配，即模式与字符串里有可能匹配的最长子串匹配
    - +
      - 要求与模式匹配的字符串1次或多次重复
      - 也是贪婪匹配
    - 可选描述符：？
      - 要求与模式匹配的字符串0次或1次重复匹配
    - \
      - 转义
    - 选择描述符：|
      - 描述与两种或多种情况之一的匹配
    - 重复次数描述符：{}
      - 确定次数多重复可以用{n}描述，表示与模式匹配的字符串n次重复
      - 如果用{m,n}，表示重复m到n次，包括m和n
    - 字符组描述符：[…]
      - 与方括号中列出的字符序列中的任一字符匹配
      - [^…]：与所有没有写在方括号中的字符匹配
    - ()
  - 常用字符组：
    - \d：与十进制数字匹配，等价于[0-9]
    - \D：与非十进制数字的所有字符匹配，等价于[^0-9]
    - \s：与所有空白字符匹配，等价于[\t\v\n\f\r]
    - \S：与所有非空白字符匹配，等价于[^ \t\v\n\f\r]
    - \w：与所有字母数字字符匹配，等价于[0-9a-zA-Z]
    - \W：与所有非字母数字字符匹配，等价于[^0-9a-zA-Z]

第五章：栈和队列

5.1 概述

- 这是一章讲被称为容器的数据结构
- 最常用的是栈和队列
- 它们主要用于在计算过程中临时保存计算过程中产生的，后面可能需要使用的数据
- 使用情况：
  - 计算过程分为一些顺序进行的步骤
  - 计算过程中有某些步骤会不断产生后面可能需要的中间数据
  - 产生的数据有些不能立即使用，但又需要在将来使用
  - 需要保存的数据项数不能事先确定

5.2 栈

- 概念：是一种容器，可以存入数据，访问数据，删除数据
- 特点：
  - 存入栈中的元素相互之间没有任何具体关系，只有到来的先后顺序
  - 确认了一种默认元素访问顺序，后进先出，访问时不需其他信息
- ADT：
  - Stack(self)
  - is_empty(self)
  - push(self, elem)
  - pop(self)
  - top(self)
- 实现：
  - 顺序栈：
    - 复杂度：后端插入和删除是O(1)，栈顶端
    - 基于python的list实现，但是因为list提供了太多栈不应该支持的操作，所以需要定义一个类，将list作为实现基础，隐藏list
  - 链栈
    - 复杂度：前端插入和删除是O(1)，栈顶端
- 应用：
  - 括号匹配

- 中缀、后缀表达式的计算
- 中缀表达式转后缀表达式
- 用来将递归算法转成非递归算法：
  - 任何一个递归函数都可以通过引入一个栈来保存中间结果的方式，翻译成一个非递归函数
    - 与此对应，任何一个包含循环的程序都可以翻译为一个不包含循环的递归函数
  - 其实递归算法内部使用的是函数帧，其中保存每次调用的相关信息，栈可以模拟这个帧，来达到递归转非递归
  - 函数调用的内部动作：
    - 前序动作：进入新函数之前保存当前函数的一些状态和信息，按顺序包括：
      - 为被调用函数的局部变量和形式参数分配存储区（称为函数帧 / 活动记录 / 数据区）
      - 将所有实参和函数的返回地址存入函数帧（实参形参的结合 / 传值）
      - 将控制转到被调用函数入口
    - 后序动作：退出一次函数调用的时候恢复调用前的状态和信息（函数返回时完成）：
      - 将被调用函数的计算结果存入指定位置
      - 释放被调用函数的存储区（帧）
      - 按以前保存的返回地址将控制转回调用函数
  - 应用：背包问题的递归和非递归算法

5.3 队列

- 概念：是一种容器，可以存入数据，访问数据，删除数据
- 特点：
  - 也没位置的概念，只支持默认方式的元素存入和取出
  - 先进先出
- ADT：
  - Queue(self)
  - is_empty(self)
  - enqueue(self, elem)
  - dequeue(self)
  - peek(self) # 查看最早进入队列的元素，不删除
- 实现：
  - 顺序表实现队列
    - 存在的问题：根据队列的先进先出的性质，队列中的元素整体会越来越往队尾移，造成前面都是空的，但是无法再插入新元素了，即“假溢出”
    - 解决办法：
      - 不那么好的解决办法：每次取出队首元素后，将整个表中的队列元素全部往前移
        - 这样的话，将得到O(n)的时间复杂度
      - 循环队列：引入两个指针，q.head指向队首，q.rear指向队尾最后一个元素后面的第一个空位
        - 入队：q.rear = (q.rear + 1) % q.len
        - 出队：q.head = (q.head + 1) % q.len
        - 同时为了区别队空和队满的情况，在队列中空了一个元素
          - 队空：q.head == q.rear
          - 队满：(q.rear + 1) % q.len == q.head
    - 基于list实现一个可以实现一个自动扩充存储的队列
      - 不能使用list自动存储扩充机制的原因：
        - 因为队列和list的默认存储方式不一样，队列是存在表里的任意一段或头尾两段，list总是存在存储区的最前面一段
        - list不提供检查元素存储区容量的机制，队列无法判断何时扩容
      - 数据不变式：
        - 有些变动操作会改变一些对象属性的取值，所以最基本的问题是这些操作需要维护对象属性之间的正确关系，这样的关系叫做数据结构的不变式
        - 数据不变式说明对象的不同属性的性质，描述它们应该满足的逻辑约束关系
        - 数据不变式队相关操作提出的要求包括两方面：
          - 所有构造对象的操作，都必须把对象成分设置为满足数据不变式的状态，即，对象的初始状态应该满足数据不变式
          - 对象的每个操作都应保证不破坏数据不变式，即，对象执行该操作前的状态是完好，执行完该操作后的状态应该还是完好
  - 链表实现队列
    - 可以使用前面带头尾指针的改进链表
    - 复杂度：尾端插入、首端访问和删除都是O(1)
- 应用：
  - 文件打印：打印机中缓存打印任务的队列
  - 万维网服务器
  - Windows系统和消息队列
  - 离散事件系统模拟

5.4 状态空间搜索问题

- 基本特征描述：
  - 存在一些可能状态
  - 有一个初始状态s0，一个或多个结束状态，或者有判断成功结束的方法
  - 每个状态s，都有neighbor(s)表示与s相邻的一组状态（一部可达的状态）
  - 有一个判断函数valid(s)判断s是否为可行状态
  - 问题：找出从s0出发到达某个（或全部）结束状态的路径；或从s0出发，设法找到一个或者全部解
- 解决办法：
  - 递归求解
  - 非递归求解：
    - 深度优先搜索（回溯法）：基于栈的搜索
      - 栈保存中间发现的回溯点
      - 表现为：尽可能向前检查，尽可能向远探索
      - 时间开销：正比于找到解之前访问的状态个数
      - 空间开销：取决于找到一个解（或所有解）之前遇到过最长的那一条搜索路径决定

- 缺点：
  - 如果一个不包含解的子区域很大，就会浪费很多时间
  - 如果包含无穷多状态的无解子区域，可能永远找不到解
  - 找到的解不一定是最优解，如果想找最优解，要遍历完整个状态空间找到所有解后，再从所有解中找出最优解
- 广度优先搜索：基于队列的搜索
  - 队列中保存的是搜索过程中较早遇到的状态
  - 表现为：齐头并进式的搜索，搜索过程中没有回溯，是一种逐步扩张的过程
  - 时间开销：正比于找到解之前访问的状态个数
  - 空间开销：
    - 由搜索过程中可能路径分支最多的那一层决定
    - 如果需要得到路径的话，还需要额外的存储，存储量与搜索区域的大小线性相关
  - 优点：
    - 一定可以找到解，而且最先找到的一定是最短路径（最近的解）
  - 缺点：
    - 如果状态空间巨大，记录前一状态的空间可能会成为很大的负担，尤其是其中很多记录可能完全没用
    - 如果没有额外的存储，通常只能返回结果，不能返回路径
- 经典实例：
  - 迷宫问题
  - 八皇后问题
  - 骑士周游问题

5.5 补充

- 双端队列deque：
  - 两端都可以插入和删除元素，而且都是O(1)的时间复杂度
  - python标准库的collections包中定义了deque类型
- 能用顺序表就不要用链表
  - 因为计算机访问数据的时候，总是把磁盘上的数据一批一批放到高速缓存中访问
  - 如果连续进行的一批内存访问时局部的，操作速度会快得多
  - 考虑程序的效率时，一个重要的线索就是尽可能使对计算机内存的使用局部化
  - 在python语言层面上，顺序表时局部化的典型代表，在可能的情况下要尽可能使用
  - 链表由于在内存中任意分配，所以访问过程中是在许多位置随机的单元之间跳来跳去，效率上有损耗

第六章：二叉树和树

6.1 二叉树

- 定义：二叉树是结点的有穷集合。这个集合或者是空集，或者其中有一个称为根结点的特殊结点，其余结点分属两棵不相交的二叉树，即左子树和右子树
- 注意：二叉树明确区分左右，而树不
- 概念：
  - 空树
  - 父结点，子结点，分支结点，叶子结点，（对于扩充二叉树：内部结点，外部结点）
  - 单点树
  - 度数
  - 路径
  - 高度、深度
  - 满二叉树：所有分支结点的度数都是2
  - 扩充二叉树：加入足够多的新叶结点，使得原本的结点都变成度为2的分支结点
    - 外部路径长度E：从树根到树中各外部结点的路径长度之和
    - 内部路径长度I：从树根到树中各内部结点的路径长度之和
  - 完全二叉树：如果从第一层到倒数第二层到结点都是满的，最后一层结点不满的话，那么结点的排列顺序是从左往右连续排列的树
    - 一棵完全二叉树可以存入一个连续的线性结构中
- 性质：
  - 1. 非空二叉树第i层中至多有2^i个结点
  - 2. 高度为h的二叉树至多有2^(h+1) - 1个结点
  - 3. 对于任何非空二叉树T，如果其叶结点（度为0）的个数为n0，度为2的结点个数为n2，那么n0 = n2 + 1
  - 4. 满二叉树里的叶结点比分支结点多一个（第3条的变形）
  - 5. 扩充二叉树的外部结点个数比内部结点个数多一个（同上）
  - 6. 扩充二叉树如果有n个内部结点，那么E = I + 2 * n
  - 7. n个结点的完全二叉树高度为h为不大于log2 n的最大整数
  - 8. 对完全二叉树进行编号的话，对于任一结点i：
    - 序号为0的结点是根
    - 对于i > 0，其父结点的编号是(i - 1)/2
    - 若2i + 1 < n，则其左孩子序号为2i + 1，否则无左孩子
    - 若2i + 2 < n，则其右孩子序号为2i + 2，否则无右孩子
  - 9. n个结点的二叉树的平均高度是O(log n)
- 抽象数据类型：
  - BinTree(self, data, left, right)
  - is_empty(self)
  - num_nodes(self)
  - data(self)
  - left(self)
  - right(self)
  - set_left(self)
  - set_right(self)
  - traversal(self)
  - forall(self, op)

- 遍历二叉树：
  - 深度优先：前序遍历、中序遍历、后续遍历
    - 如何从遍历结果恢复成二叉树?
      - 必须知道中序遍历结果和其他两个结果中的一个，这样可以唯一确定一个二叉树
  - 广度优先：层序遍历
- 实现：
  - 方案1：基于list或tuple实现，采用嵌套括号的方式表示树
  - 方案2：链接实现
    - 遍历时，可以递归定义遍历函数，也可以非递归定义遍历函数，这样的话需要一个栈
      - 非递归遍历的复杂度：
        - 时间：结点个数的线性函数，O(n)
        - 空间：最坏O(n)，平均O(log n)
    - 对于层序遍历，还需要一个队列
- 应用：
  - 表达式树
  - 哈夫曼树：
    - 定义：带权路径最短的二叉树
    - 构造：每次选取所有结点里面最小的两个结点作为子树，根是两个结点之和
    - 实现：选用优先队列存放，每次都能弹出最小的结点
      - 复杂度：算法执行时构造出一棵包含2m-1个结点的树，时间复杂度O(m log m)，空间复杂度是O(m)
    - 应用：哈夫曼编码，要求：
      - 用这种编码的存储/传输时平均开销最小
      - 对任意两个字符的编码，其中一个的编码不是另一个编码的前缀

- 6.2 优先队列
  - 定义：它是一种缓存结构，主要用于保存元素来访问和弹出。特点是存入其中的每项数据都另外附有一个数值，表示该项的优先程度，称做优先级。优先队列会保证每次访问或者弹出的数据都是这个结构里优先级最高的元素
  - 实现：
    - 基于线性表：
      - 方案1：存入数据时，保证表中的元素始终按优先顺序排列，这样的话存入麻烦，访问和弹出简单
        - 采用连续表实现方案1的话，最优先元素应该定在表尾，这样的话访问和弹出操作的复杂度是O(1)，插入是O(n)（无论换不换更大多存储区）
      - 方案2：无组织存放数据，每次都需要检索才能找到最优先的元素，这样的话存入简单，访问和弹出麻烦
        - 插入操作的复杂度是O(1)（队列满，换更大的存储区时需要O(n)），访问和弹出都是O(n)
      - 总的来说，基于线性表的总要有复杂度是O(n)的操作，不能满意；终极原因在于要沿着表顺序检索，只要元素按优先级顺序线性排列，就无法避免线性复杂度，所以要考虑其他数据结构组织方式
    - 基于堆：
      - 堆：
        - 采用树形结构实现优先队列的一种方式；堆就是结点中按堆序存储了数据的完全二叉树；所以跟完全二叉树一样可以存在线性表中
        - 堆序：任一结点里所存的数据先于或者等于其子结点里的数据，即优先关系非严格递减
          - 小元素优先的堆序就是小顶堆
          - 大元素优先的就是大顶堆
        - 特点：
          - 优先关系从上到下非严格递减
          - 最优先的元素必定是根结点，O(1)时间就能得到
          - 位于树中不同路径上的元素，不关心其顺序关系，即，不管兄弟结点之间的大小
        - 性质：
          - 1. 在堆的最后（即相应连续表的最后）加一个元素，整个结构还是完全二叉树，但不一定是堆
          - 2. 去掉堆顶元素，其余元素形成两个子堆，堆和完全二叉树的一切还都适用
          - 3. 给从2中得到的两个子堆增加一个根元素，整个结构还是完全二叉树，但不一定是堆
          - 4. 去掉堆的最后元素，还是堆
        - 应用：堆排序
          - 即先构建一个堆，然后不断弹出元素，弹出的元素就是按顺序排列的了
          - 节省空间的做法：弹出的元素放到堆后面空出来的位置，所以空间复杂度是O(1)
          - 时间复杂度：O(nlog n)
      - 使用堆作为优先队列：
        - 访问复杂度O(1)
        - 插入复杂度O(log n)，如果因list对象满，而替换存储区的话，是最坏情况O(n)
          - 过程：在堆的最后插入新元素，并且做一次向上筛选
          - 向上筛选：不断用新加入的元素（设为e）与其父结点的数据进行比较，如果e较小（小顶堆，若是大顶堆就是较大）就交换两者位置，直到根结点
          - 向上筛选中比较和交换的次数不会超过二叉树中最长路径的长度，即O(log n)
        - 弹出复杂度O(log n)
          - 过程：弹出根结点后，取下堆中最后一个元素作为新的根结点，并做一次向下筛选
          - 向下筛选：把根e和左右元素进行比较，最小者（小顶堆）作为整个堆的顶，某次比较中e最小，或者e已经到底，就停止
          - 弹出堆顶和取下最后一个元素作为根的复杂度是O(1)，向下筛选中的操作次数不会比最长路径多，所以需要O(log n)
        - 构建堆的复杂度O(n)
        - 其他：O(1)
  - 应用：离散事件模拟

- 6.3 树和树林
  - 树：
    - 定义：有个根结点，除根没有前驱以外，其他结点只有一个前驱，不管有几个后继
    - 注意：二叉树和度为2的有序树的区别，如果某个分支结点只有一个子结点，二叉树必须指明时左孩子还是右孩子，而度为2的有序树就没有这个概念
    - 概念：
      - 有序树、无序树
      - 其他同二叉树

- 性质：
  - 1. 度为k的树中，第i层至多有k^i个结点
  - 2. 度为k，高为h的树中至多有[k^(h+1) - 1] / (k - 1)个结点
  - 3. n个结点的k度完全树，高度为不大于logk n的最大整数
  - 4. n个结点的树里有n-1条边
- 抽象数据类型：
  - Tree(self, data, forest)
  - is_empty(self)
  - num_nodes(self)
  - data(self)
  - first_child(self)
  - children(self)
  - set_first(self, tree)
  - insert_child(self, i, tree)
  - traversal(self)
  - forall(self, op)
- 树的遍历：
  - 深度优先：先序遍历、后序遍历
  - 广度优先：层序遍历
- 树的表示：
  - 双亲表示法：每个结点，除了存储自身的信息，还存储着双亲结点的位置
    - 优点：已知孩子找双亲O(1)
    - 缺点：已知双亲找孩子，需要遍历整棵树
  - 孩子表示法：每个结点上存放指向头一个孩子位置的指针，头一个孩子再指向下一个
    - 优点：找孩子方便
    - 缺点：找双亲麻烦
  - 双亲孩子表示法：既存双亲位置，也存孩子指针
  - 孩子兄弟表示法：每个结点设两个指针域，一个指向长子，一个指向兄弟
- 实现：基于list实现树

- 树林：
  - 定义：0棵或多棵互不相交的树的集合
  - 注意：树和树林可以相互递归定义：非空树由树根和子树树林构成，树林由一组树组成

- 树、树林和二叉树的关系：存在一种一一对应的关系，可以把任何一个（有序）树林映射到一颗二叉树，而其逆映射可以把二叉树恢复成树林
  - 即二叉树和树、树林之间可以相互转换

第七章：图

7.1 基础

- 定义：图G由顶点V和边E组成
- 分类：
  - 有向图：边有方向，是顶点的有序对，用<vi，vj>表示，vi始点，vj终点
  - 无向图：边无方向，是顶点的无序对，用(vi，vj)表示
- 概念：
  - 邻接点、邻接关系：两个顶点之间存在边（有向图邻接关系是单向，无向图是双向），就说这两个顶点是邻接点，存在邻接关系
  - 度：顶点邻接的边的条数，对于有向图还分入度和出度
    - 入度：以该顶点为终点的边的个数
    - 出度：以该顶点为始点的边的个数
  - 完全图：任意两个顶点之间都有边的图
    - n个顶点的无向完全图有n * (n-1) / 2条边
    - n个顶点的有向完全图有n * (n-1)条边
    - |E| <= |V^2|，即|E| = O(|V^2|)，边的条数总是小于或等于顶点个数的平方
  - 路径：从一个顶点可以走到另一个顶点
    - 长度：路径上边的条数
    - 回路（环）：起点和终点相同的路径
      - 简单回路：一个环除了起点和终点外其他顶点均不相同
    - 简单路径：内部不包含回路的路径，即起点和终点有可能相同，简单回路是简单路径
  - 有根图：有向图里的一个顶点到其他顶点都有路径，根可以不唯一
    - 最小有根图：去掉任何一个顶点不再是有根图的有根图
  - 连通图：
    - 连通：从一个顶点到另一个顶点有路径，那就叫连通；注意，有向图的连通可以不是双向的
    - 连通无向图：无向图中任意两个顶点之间都连通
    - 强连通有向图：有向图中任意两个顶点之间互相连通，即，两个方向的路径都存在
      所以完全无向图和完全有向图都是连通图，但反过来不一定对
    - 最小连通图：去掉任意一条边后不再连通的连通图
  - 子图：原图的一部分，这一部分还要满足图的定义，即子图的任意一条边都要有两个顶点
  - 连通子图：原图可能不连通，但是它的子图连通，这样的子图就是原图的连通子图/强连通子图
  - 极大连通子图(连通分量)：加一个顶点就要不连通的连通子图
  - 极大强连通子图(强连通分量)：同上，注意，有向图的强连通分量之间可能单向连通，即所有的强连通分量只对顶点进行划分，边可以少几条，它们合起来不一定等于原图
  - 带权(有向/无向)图：每条边上都赋予了权值
  - 网络：带权的连通无向图
- 性质：
  - 1. 顶点数、边数、度数的关系：边数等于所有顶点的度数求和除以2
  - 2. 最小连通无向图的边数：包含n个顶点的最小连通无向图有n-1条边
  - 3. 最小有根图的边数：包含n个顶底的最小有根图有n-1条边
  - 4. 图中的路径和路径上的边数：n个顶点的图，可以找到一个包含n-1条边的边集合，这个集合包含从v0到其他所有顶点的路径（即可以构造生成树）

- 5. 生成树边数：n个顶点的连通图的生成树包含n-1条边，无向图的生成树就是该图的最小连通子图
- 6. 生成树林的边数：n个顶点，m个连通分量的无向图的生成树林包含 n - m 条边

抽象数据类型：

- Graph(self)
- is_empty(self)
- vertex_num(self)
- edge_num(self)
- vertices(self)：获得顶点的集合
- edges(self)：获得边的集合
- add_vertex(self, vertex)
- add_edge(self, v1, v2)
- get_edge(self, v1, v2)：得到v1到v2边有关的信息
- out_edge(self, v)：取得从v出发的所有边
- degree(self, v)：获取v的度

图的表示：

邻接矩阵

- 无向图的邻接矩阵都是对称阵
- 缺点：比较稀疏，浪费空间

- 邻接表：图中每个点关联一个边表
- 邻接多重表
- 十字链表法

7.2 图的算法

图的遍历：

- 深度优先遍历(DFS: Depth-First Search)
- 广度优先遍历(BFS: Breadth-First Search)

生成树：

- 生成树可能不唯一
- 遍历的时候记录一下经过的边，加上所有的顶点就是一棵生成树

分类：

- 深度优先生成树
- 广度优先生成树

最小生成树：

- 基于带权连通无向图，求解权值最小的生成树(Minimum Spanning Tree, MST)

Kruskal算法

- 不断选择最小的，可以减少连通分量的边

Prim算法

- 从顶点出发，选点，后选择相邻的最小边
- 基于MST性质：如果与顶点相连的候选边里有一个权值最小，那么这个边一定包含在一个最小生成树里

最短路径：

求解单源点最短路径：Dijkstra算法

- 每次都选择当前已知距离里面最短的那一个
- 最短路径中的前段也是最短路径
- 限制：权值不小于0

求解到所有点的最短路径：Floyd算法

- 权值可以小于0，但是不能有包含负权值的环（负权回路）
- 最开始是形成一个邻接矩阵，然后只允许1号顶点中转，看看能不能更新矩阵，然后允许1号和2号顶点，到允许1～n号所有顶点进行中转
- 从i号顶点到j号顶点只经过前k号点的最短路程

AOV/AOE网及算法(基于有向图)：

概念：

- AOV(activity on vertex network)：顶点活动网，用顶点表示活动，用边表示活动之间的先后顺序关系
- AOE(activity on edge network)：顶点表示事件，它的入边所表示的活动都要已完成才能开始该顶点的活动，它的出边表示

拓扑排序：

- 如果存在顶点1到顶点2的路径，那么1就排在2前面，将AOV网里的所有排成满足这样顺序
- 一个图存在拓扑排序的充要条件：图中不包含回路
- 拓扑排序得到的拓扑序列不一定唯一
- 将AOV网的拓扑序列反向得到的序列，都是图中每条边反转得到的AOV网(逆网)的一个拓扑序列

关键路径算法：

- 关键路径可能不止一条
- 通过拓扑排序算出最早开工时间(时间取最大的那个)得到的数组，和最晚开工时间(逆向，时间取最小的那个)得到的数组比较，值相同的就是关键路径上的点
- 时间复杂度：采用邻接表：O(|V| + |E|)，采用邻接矩阵：O(|V|^2)
- 空间复杂度：O(|V|)

第八章：字典和集合

8.1 字典

- 定义：支持基于关键码的数据存储与检索的数据结构

分类：

- 静态字典：建立字典后，字典内容和结构不会再变化，主要操作只有检索。所以最重要的检索效率
- 动态字典：建立字典后，字典的内容和结构一直处于变动之中。不仅需要考虑检索效率，也要考虑插入和删除效率。

评价字典：

- 设计良好的字典要保证其性能不会随着操作的进行逐渐恶化

评价标准：平均检索长度(Average Search Length)

- 将所有数据元素的检索长度和检索概率的乘积求和
- 但是这样只考虑了检索关键码在字典中存在的情况，还要考虑不存在的情况

- 实现：
  - ADT：
    - Dict(self)
    - is_empty(self)
    - num(self)
    - search(self, key)
    - insert(self, key, value)
    - delete(self, key)
    - values(self) # 以迭代方式取得字典里保存的所有value
    - entries(self) # 以迭代方式取得字典里所有的key和value二元组
  - 线性表实现
    - 将key和value组成数据项存入线性表中
    - 无序线性表
      - 检索时只能按照要找的关键码顺序查找，ASL=O(n)
      - 优点：简单明了，适用于任意关键码，插入复杂度低
      - 缺点：检索和删除复杂度高
    - 有序线性表
      - 插入时考虑按key的顺序插入关键码，这样可以利用二分法快速检索数据
      - 时间复杂度：
        - 插入、删除：O(n)，虽然检索要插入和删除的位置可以使用二分法，为O(log n)，但是完成增删的操作需要移动元素
        - 检索：O(log n)
      - 优点：检索效率提高
      - 缺点：
        - 只适用于关键码可以排序的字典
        - 只适用于顺序存储结构，需要连续的存储块，不适合实现很大的动态字典
        - 插入删除复杂度高
    - 总的来说，线性表实现方法的缺点：
      - 不能很好地支持数据的变化
      - 必须采用连续的方式存储整个数据集合，如果数据集很大，那就无法接受这种方式
      - 用顺序表实现字典总存在一些低效操作，所以线性表只适用于小型字典的实现

8.2 散列和字典

- 基本思想：把关键码当作存储数据的地址或下标，需要找数据的时候，直接去搜是关键码的那个地址，这样只要O(1)时间就可找到数据。如果一种关键码不能当成地址或下标，就通过一个变换，把关键码映射成一个地址或下标。总的来说，就是把基于关键码的检索转变为基于整数下标的直接元素访问
- 散列表实现字典：
  - 选取一个整数的下标范围（通常从0或1开始），建立一个包括相应元素位置范围的顺序表
  - 选定一个从实际关键码集合到上面的下标范围的散列函数h：
    - 在遇到关键码为key的数据时，将其存入表中第h(key)个位置
    - 遇到以key为关键码的检索数据时，直接去表中第h(key)个位置找
  - 优点：基于概率考虑，操作效率高；对关键码类型无特殊要求，应用广泛
  - 缺点：没有确定性的效率保证，不适合用于对效率有严格要求的环境；散列表中不存在遍历元素的明确顺序
- 评价散列表的性能：
  - 负载因子a = 散列表中当时的实际数据项数 / 散列表基本存储区能容纳的元素个数
  - a越大，出现冲突的可能性就越大；a越小，散列表中空闲空间的比例就越大
  - 负载因子达到一定大小之后，需要扩大基本存储区，同时对散列函数进行调整，尽可能利用新增加的空间，提高操作效率，用空间换时间
  - 一般来说a应该在0.7以下
- 因为下标集合通常远远小于关键码的集合，所以冲突(碰撞)不可避免
- 要解决两个大问题：
  - 散列函数的设计
    - 原则：尽可能减少冲突的可能性，散列函数的映射关系越乱越好，越不清晰越好（尽可能消除关键码和映射值之间明显的规律性）
    - 设计考虑：
      - 函数应该能把关键码映射到值域INDEX中尽可能大的部分，这样扩大空间，可以把数据映射到新增加的存储区，尽可能利用新增加的空间，出现冲突的可能性就会下降
      - 不同关键码在值域INDEX里分布均匀，有可能减少冲突
      - 函数的计算比较简单。本意是提高效率，函数复杂的话，得不偿失
    - 用于整数关键码的散列方法：
      - 数字分析法：
        - 对于给定的关键码集合，分析所有关键码中各（个十百千）位数字的出现频率，从而选出分布情况比较好的若干数字作为散列函数的值
        - 缺点：关键码和其分布要事先已知，不能用于未知的情况
      - 折叠法：将较长的关键码切分为几段，通过某种运算将切分的几段合并。
      - 中平方法：先求出关键码的平方，然后取中间的几位作为散列值
      - 除余法：
        - 关键码是整数key，用key除以某个不大于散列表长度m的整数（一般取小于m的最大素数），用得到的余数（或余数加s，由下标开始值确定）作为散列地址
        - 缺点：
          - 除偶数时，就会使偶数关键码映射到偶数散列值，奇数关键码映射到奇数关键值，违背了散列函数的设计原则
          - 相近的关键码会映射到相近的位置
      - 基数转换法：
        - 不仅适用整数关键码还适用于字符串关键码
        - 对于整数关键码：取一个正整数r，把关键码看作基数为r的数（r进制数），将其转换为十进制或二进制数。通常r取素数以减少规律性。
        - 对于字符串关键码：把字符串里的每个字符看做一个整数（比如适用ascii码），这样把整个字符串就可以看成以某个整数（一般是29或31）为基数的整数
    - 对于非整数的关键码：一般设计一种方法将其转换成整数，再用整数散列的方法
  - 冲突消解机制
    - 因为散列函数是从大集合到小集合的全函数，所以冲突不可避免
    - 消解方法分类：

- 内消解方法：在基本存储区内部解决冲突问题，又叫开地址法
  - 在准备插入数据并发现冲突时，设法在基本存储区里为要插入到数据另行安排一个位置
  - 需要设计一个探查方式来安排位置，抽象的方法是为散列表定义一种易于计算的探查位置序列
    - 探查序列是简单整数序列的话，就是线性探查
    - 探查序列是另一个散列函数的话，就是双散列探查
  - 要插入数据的时候，有空位就直接插入，要是里面有数据了，就按照探查序列挨个看有没有空位
  - 负载因子a 在0.7到0.75的时候，散列表的平均检索长度就接近于常熟
- 外消解方法：在基本存储区外解决冲突问题
  - 溢出区方法：另外设置一个溢出区，一旦发生冲突，就把冲突的数据放到溢出区里
    - 缺点：随着溢出区中数据的增长，散列表的性能将趋向线性
  - 桶散列：
    - 散列表中不放数据项，而放的是对数据项的引用，引用着一个保存实际数据的存储桶
    - 最简单的设计：拉链法
      - 拉链法中一个存储桶就是一个链接的结点表，数据项存入相应结点表中的结点里，具有同样散列值的数据项都存在这个散列值对应的链表里
      - 优点：所有数据项可以统一处理，不区分冲突项，而且允许任意的负载因子，但随着负载因子的变大，检索时间也趋于线性

8.3 二叉排序树和字典

- 线性实现和散列表实现都需要把字典存储在一个连续存储块里，方便管理；但是如果字典很大，就需要很大块的连续存储，动态修改不太方便，而且无法实现巨型字典
- 为了更好地支持存储内容的动态变化，考虑采用链接结构，即树
- 二叉排序树：
  - 定义：一种结点里存储数据的二叉树。一棵二叉排序树或者为空，或者具有以下性质：
    - 其根结点保存着一个数据项（及其关键码）
    - 如果左子树不空，那么左子树所有结点的值都小于（或不大于）根结点保存的值
    - 如果右子树不空，那么右子树所有结点的值都大于根结点保存的值
    - 非空的左子树或右子树也是二叉排序树，二叉排序树是一种递归结构
  - 性质：
    - 中序遍历二叉排序树，得到按关键码上升的递增序列
    - 同一数据集对应的二叉排序树不唯一
    - 如果树的结构良好，检索的时间开销是O(log n)；如果树的结构畸形，检索大时间开销可能达到O(n)；n个结点的所有可能二叉树的平均高度是O(log n)，所以可以认为二叉排序树检索时的平均时间复杂度是O(log n)
    - 二叉排序树的插入删除检索的空间复杂度是O(1)
- 最佳二叉排序树：
  - 定义：使检索的平均比较次数达到最少的二叉排序树
  - 性质：最佳二叉排序树的任何子树也是最佳二叉排序树
  - 构造方法：动态规划，从最小的最佳二叉排序树做起，逐步做出所需的最佳二叉排序树；构造每棵树时，都要考虑所有可能的组合构造方式，从中选择出最佳的一棵树
  - 优点：可以保证最佳的检索效率
  - 缺点：
    - 需要掌握所有元素的分布情况，构造成本高
    - 只适合作为静态字典的表示，不能很好的支持动态变化

8.4 平衡二叉排序树（AVL树）

- 定义：平衡二叉树是一类特殊的二叉排序树，或为空树，或者其左右子树都是平衡二叉排序树（递归结构），而且其左右子树的高度之差的绝对值不超过1
- 概念：
  - 平衡因子(Balance Factor)：左子树高度减右子树高度的差，其可能取值只有0，1，-1
- 性质：
  - 如果能维持平衡二叉树的结构，检索操作就能在O(log n)时间内完成
  - 插入删除需要恢复树的平衡，所以时间代价为O(log n)
- 操作：
  - 插入：插入后可能会出现失衡，需要经过局部的旋转调整来恢复树的平衡。具体做法是先插入即诶单，发现失衡后就在最小不平衡子树的根结点附近做局部调整，就可以把该子树根的平衡因子变为0，恢复操作分为：
    - LL型调整：根结点的左子树较高，新结点插在根结点的左子树的左子树
    - LR型调整：根结点的左子树较高，新结点插在根结点的左子树的右子树
    - RR型调整：根结点的右子树较高，新结点插在根结点的右子树的右子树
    - RL型调整：根结点的右子树较高，新结点插在根结点的右子树的左子树
  - 删除：与插入类似，也是确定结点并删除，而后调整结构恢复
- 优点：
  - 失衡后的恢复可以在局部完成，插入删除的复杂度不超过O(log n)
  - 可以适应长期运行和反复动态修改，保证O(log n)的操作复杂度
- 缺点：操作的实现比较复杂

8.5 动态多分支排序树

- B树
  - 定义：一棵m阶B树或者为空，或者具有以下特征：
    - 树中分支结点至多有m-1个排序存放的关键码。根结点至少有一个关键码，其他结点至少有 [(m-1)/2]下 个关键码。所有叶结点位于同一层，仅用于表示检索失败，实际上不需要表示（例如，可以用空引用表示）
    - 如果一个分支结点有j个关键码，它就有j+1棵子树，这一结点中保存的是一个序列<p0, k0, p1, k1, … ,p j-1, k j-1, pj>，其中ki为关键码，pi为子结点的引用，而且ki大于pi所引用子树里所有的关键码，小于pi+1所引子树里的所有关键码
  - 设计B树的原则：
    - 保持树形结构和结点中关键码有序，用分支结点的关键码作为相应子树关键码的区分关键码，保证检索能正确进行
    - 保证树中从根到所有叶结点的路径等长，并保证分支结点中关键码的个数在确定的最小最大范围内变化，并因此保证树结构良好
  - 特点：
    - 结点内检索和结点间检索的结合
    - 采用结点分裂和合并的技术控制树高，保证到所有叶结点的路径长度相同
  - 应用：实现大型数据库的索引
- B+树
  - 定义：一棵m阶B+树或者为空，或者具有以下特征：
    - 树中每个分支结点至多有m棵子树，除根结点外的分支结点至少有 [m/2]下 棵子树。如果根结点不是叶结点，至少有两棵子树。
    - 关键码在结点里按顺序存放。分支结点里的每个关键码关联着一棵子树，这个关键码等于其所关联子树的根结点里的最大关键码。叶结点里的每个关键码关联着一个数据项的存储位置，数据项另行存储

- 与B树的不同：
  - 分支结点的关键码不是子树的区分关键码，而是子树的索引关键码
  - 分支结点的关键码并不关联数据项，只有叶结点的关键码关联数据项；即，一个叶结点可以看作一个基本索引块，其中每个关键码对应一项数据的索引，而分支结点可以看作索引的索引，整个B+树形成一个分层索引结构

8.6 集合

- 定义：个体的汇集称为集合
- 描述方法：
  - 集合的外延表示：明确列出集合中所有元素，只能描述有穷集合
  - 给出集合中的元素应满足的性质
- 概念：
  - 基数：集合中元素的个数
  - 空集：不包含任何元素
  - 子集和真子集：真子集不会相等
  - 并集、交集、差集：交集求相交的地方，并集把集合并到一起
- ADT：
  - Set(self)
  - is_empty(self)
  - member(self, elem) # 检查elem是否为集合中的元素
  - insert(self, elem)
  - delete(self, elem)
  - intersection(self, oset) # 求交集
  - union(self, oset) # 求并集
  - different(self, oset) # 求差集
  - subset(self, oset) # 判断本集合是否为oset的子集
- 实现：
  - 线性表实现：
    - 时间复杂度：
      - member：O(n)
      - 集合操作：(m * n)，集合元素分别为m、n
    - 优点：简单
    - 缺点：元素判断和集合运算的操作效率低
  - 排序顺序表实现：
    - 集合操作时间复杂度：O(m + n)
  - 散列表实现：
    - 一个集合就是一个散列表
    - 插入删除元素对应散列表插入删除关键码
    - 集合元素判断对应关键码检索
    - 集合运算可以采用建立新散列表的方式实现
    - 具有概率性，最佳情况下，插入删除的代价是常量，集合操作的复杂度是O(m + n)
  - 位向量实现——集合的位向量表示：
    - 前提：集合对象有一个不太大的公共超集U，要使用的集合对象S们都是U的子集，就可以考虑用位向量技术实现集合
    - 实现：
      - 假定U中包含n个元素，每个元素确定一个编号作为该元素的下标
      - 对要使用的集合对象S，用一个n位的二进制序列（位向量）来表示，对任何U中的元素，如果它在S中存在，那么就把位向量中对应的二进制位取1，否则取0
      - 这样一来，检查是否数据是否存在，就看对应二进制位是否为1即可
      - 插入就是把二进制位置1，删除就是把二进制位置0
      - 集合操作都能通过逐位操作实现：
        - 求交集时，两个都为1，交集的位向量对应位置才为1，否则为0
        - 求并集时，两个都为0，并集的位向量对应位置才为0，否则为1
        - 求差集时，一个为1，另一个为0，差集的位向量对应位置才位1，否则为0

第九章：排序

- 定义：整理数据的序列，使其中的元素按照特定的顺序排列的操作
- 分类（按存放记录位置）：
  - 外排序：针对外存（磁盘等）数据的排序工作
    - 适合的排序算法：归并排序
  - 内排序：待排序的记录全部保存在内存
- 基本操作（基于比较的排序）：
  - 比较关键码的操作，确定顺序关系
  - 移动数据记录的操作，调整记录的位置或顺序
- 评价：
  - 时间复杂度：
    - 基于比较的排序时间复杂度最优是O(n log n)，比如堆排序，不存在更优
  - 空间复杂度：
    - 常量的空间复杂度的算法是原地排序算法
  - 稳定性：排序后，关键码相同的记录相对位置不变；任何不稳定的算法都可以改成稳定的，只要把相同的关键码再加一个相对位置，排序算法发现相同元素时，转而进一步比较后面的相对位置，就可以保证稳定性
  - 适应性：对更有序的序列，算法排序的越快，就是有适应性；如果不管序列是不是更有序，算法都要耗费一样的时间的话，就是没有适应性；也就是说，最好的时间复杂度和最坏的时间复杂度不一样
- 分类（按排序的特点）：
  - 简单排序算法：
    - 插入排序：
      - 简单插入排序：
        - 思想：维持所构造好的序列的排序性质，将未排序元素一个个插入到已排序的序列中
        - 时间复杂度：
          - 最好：O(n)

        - 最坏：O(n^2)
        - 平均：O(n^2)
      - 空间复杂度：O(1)
      - 稳定性：是
      - 适应性：是
    - 二分插入排序：
      - 思想：在简单插入排序的基础上，用二分法才已排序序列中寻找未排序元素应该插入的位置
      - 时间复杂度：
        - 最好：O(n log n)
        - 最坏：O(n^2)
        - 平均：O(n^2)
      - 空间复杂度：O(1)
      - 稳定性：是
      - 适应性：是
  - 选择排序：
    - 直接选择排序：
      - 思想：每次排序都选出整个序列中最小的或最大的元素，放到已排序的序列中（可以通过简单交换来实现）
      - 时间复杂度：
        - 最好：O(n^2)
        - 最坏：O(n^2)
        - 平均：O(n^2)
      - 空间复杂度：O(1)
      - 稳定性：否
      - 适应性：否
    - 堆排序：
      - 思想：把要排序的元素建立一个堆（大顶或小顶），每次从堆顶取元素，取完就是排序完
      - 时间复杂度：
        - 最好：O(n log n)
        - 最坏：O(n log n)
        - 平均：O(n log n)
      - 空间复杂度：O(1)
      - 稳定性：否
      - 适应性：否
  - 交换排序
    - 冒泡排序：
      - 思想：通过比较相邻记录，发现相邻逆序对时就交换它们，消除逆序对，得到排序序列（如果一次扫描中没遇到逆序，就说明已经排好序，可以提前结束）
      - 时间复杂度：
        - 最好：O(n)
        - 最坏：O(n^2)
        - 平均：O(n^2)
      - 空间复杂度：O(1)
      - 稳定性：是
      - 适应性：是
- 快速排序：
  - 思想：划分，按照某种标准把要考虑的记录划分成小记录和大记录，并通过递归不断划分，最终得到一个排序的序列
  - 做法：随机在序列中选一个标准（可以是第一个元素），设置两个指针i指向序列的头和j指向序列的尾，从右端开始，发现第一个小于标准的元素，j停下来，开始向右移动i，发现一个大于标准的元素就停下来，交换i和j指向的元素，再开始新一轮移动j，直到i和j相遇，这个位置就是标准在已排序的序列中应该放置的位置。交换标准和这个位置的元素，然后从标准的两边分别开始递归进行新一轮的排序，直到最终每个记录组中只包含一个记录时，排序完成
  - 时间复杂度：
    - 最好：O(n log n)
    - 最坏：O(n^2)
    - 平均：O(n log n)
  - 空间复杂度：
    - 最好：O(log n)
    - 最坏：O(n)
  - 稳定性：否
  - 适应性：否
- 归并排序
  - 思想（二路归并）：divide and conquer，和快速排序的不断划分直到只有一个元素不同，归并排序一开始把n个待排序的记录看作n个有序子序列，然后将n个有序子序列两两归并，完成一遍后，序列组里的排序序列个数减半，每个子序列的长度加倍，对加长的有序子序列重复上面的操作，最终得到一个长度为n的有序序列
  - 时间复杂度：
    - 最好：O(n log n)
    - 最坏：O(n log n)
    - 平均：O(n log n)
  - 空间复杂度：O(n)
  - 稳定性：是
  - 适应性：否
- 分配排序
  - 思想：不基于关键码的比较，基于一种固定位置的分配和收集
  - 做法：为每个关键码设置一个桶（能容纳任意多个记录的容器），排序时简单地根据关键码把记录放入桶中，存入所有记录以后，顺序收集各个桶里的记录，就得到排序完的序列
  - 时间复杂度：O(n)
  - 缺点：要建立大量的桶；而且关键码取值比较宽的话，可能很多桶中都是空的
- 基数排序（多轮分配排序）：

- 思想：排序过程中从低到高位逐位进行分配和收集，这样可以保证在高位分配收集时，低位是递增排序的

- Timsort：python内置排序函数sort采用的排序方法，是一种混成式排序算法，结合了归并排序和插入排序
  - 时间复杂度：
    - 最好：O(n)
    - 最坏：O(n log n)
    - 平均：O(n log n)
  - 空间复杂度：O(n)
  - 稳定性：是
  - 适应性：是