# 集合概览

## 集合类型

### 线性集合

除了第一项之外，每项有唯一的前驱

除了最后一项之外，每项有唯一的后继动

### 层级集合

除了最顶端的第一个项，每个项都有唯一的前驱

每个项可能有多个后继

### 图集合

每个项可以有多个前驱，也可以有多个后继

与某项分别相连的两项可以成为邻居

### 无序集合

集合中的元素是没有顺序的，估没有必要讨论去是否存在前驱与后继

### 有序集合

为每项加上了一个自然排序。比如电话号码本，允许客户按照排好的顺序进行访问

## 集合的操作

确定大小

测试项的成员关系

遍历集合

获取一个字符串标识

测试相等性

链接两个集合

转换为另一种类型的集合

插入一项

删除一项

替换一项

访问或获取一项

## 小结

* 集合是保存0个或多个其他对象的对象
* 集合拥有访问对象、插入对象、删除对象、确定集合的大小以及遍历或访问集合的对象的操作
* 集合的5个主要类型是：线性集合、层级集合、图集合、无需集合、有序集合
* 集合是可迭代的，可以用一个for循环访问包含在集合中的每一项
* 抽象的数据类型是一组对象，以及这些对象上的操作，因此，集合是抽象数据类型
* 数据结构是表示集合中包含的数据的一个对象

# 搜索、排序和复杂度分析

## 评估算法的性能

### 度量算法的运行时间

#### 基准评价（探查）描述

使用计算机的时钟获取一个世纪的运行时间，针对不同大小的样本，多测试几次之后取平均值

#### 基准评价（探查）问题

不同硬件平台的处理速度不同

对于很大的数据集合来说，确定某些算大的运行时间是不切实际的

### 统计指令

#### 描述

统计对不同的问题规模说要执行的指令数目。不管算法找什么平台上运行，这个统计数字对于算法所需要执行的抽象工作量给出了很好的预计

#### 指令类别

不管问题规模多大，执行相同次数的指令

根据问题规模的大小，执行不同次数的指令

#### 度量算法所使用的的内存

和算法所需要消耗的内存大小

## 复杂度分析

### 复杂度的阶

算法复杂度表达式的阶乘的数值

#### 常数阶 c

对与任何规模的问题，都需要相同的操作次数，里：列表索引

#### 线性阶 n

随着问题规模的大小变化，而等比例变化操作次数

#### 对数阶log2n

和问题规模的log2成比例，也就是说问题规模翻倍，工作量只加一

#### 平方阶 n2

和问题规模的平方成比例

#### 字数阶 2n

工作量的常数的n规模大小次方

### 大O表示法

用于标识一个算法的效率或者计算复杂度的一种表示法，忽略项和系数，只关注复杂的n，O表示为 on zhe order of (在……阶)

### 常量比例的作用

大O表示经常会忽略常量，但是当敞亮比较大时候，他们可能影响到算法

## 搜索算法

### 最好情况、最坏情况和平均情况的性能

#### 以判断元素是否存在在列表中为例

顺序搜索是目标所在的位置决定了算法的性能 ，所以要综合考虑其性能

最坏情况：元素在列表的末尾，或者没有该元素，需要执行n次迭代

最好情况：元素在列表的开头，需要执行1次迭代

平均情况：计算元素在每个位置的迭代次数求出平均值 即（n+1）/2

#### 有序的二叉搜索

即二分法查找，其最坏的情况的复杂度为O(log2n)

## 基本排序算法

### 选择排序 O(n2)

#### 描述

从第一位开始到底len-1位，找到最小项的索引，如果最小项的索引不是起始位置，则交换最小项与其实位置的项

#### 代码

1. def selectionSort(tlist):
2. i = 0
3. while i < len(tlist) - 1:
4. minIndex = i
5. j = i + 1
6. while j < len(tlist):
7. if tlist[j] < tlist[minIndex]:
8. minIndex = j
9. j = j + 1
10. if minIndex != i:
11. tlist[minIndex], tlist[i] = tlist[i], tlist[minIndex]
12. i = i + 1
13. return tlist
14. if \_\_name\_\_ == '\_\_main\_\_':
15. import random
16. lyst = [random.randint(1, 100) for i in range(20)]
17. print(lyst)
18. selectionSort(lyst)
19. print(lyst)

### 冒泡排序 O(n2)

#### 描述

从第一位开始，搜索到最后一位，如果前面（开头方向）的比后面的大，则交换两个元素的位置，知道最后一样，然后再次从第一位开始，判断到len-1位，直至结束

#### 代码

1. def bubbleSort(tlist):
2. n = len(tlist)
3. while n > 1:
4. i = 1
5. flag = False
6. while i < n:
7. if tlist[i] < tlist[i - 1]:
8. tlist[i], tlist[i - 1] = tlist[i - 1], tlist[i]
9. flag = True
10. i = i + 1
11. n = n - 1
12. if not flag:
13. return tlist
14. return tlist
15. if \_\_name\_\_ == '\_\_main\_\_':
16. import random
17. lyst = [random.randint(1, 100) for i in range(20)]
18. print(lyst)
19. bubbleSort(lyst)
20. print(lyst)

### 插入排序O(n2)

#### 描述

（index从0开始）i = range（1，len-1），将第i个项插入到前i个项中的正确位置，直至结束，类似于给扑克牌排序

#### 代码

1. def insertionSort(tlist):
2. i = 1
3. while i < len(tlist):
4. waitInsert = tlist[i]
5. j = i - 1
6. while j >= 0:
7. if waitInsert < tlist[j]:
8. tlist[j + 1] = tlist[j]
9. j = j - 1
10. else:
11. break
12. tlist[j + 1] = waitInsert
13. i = i + 1
14. return tlist
15. if \_\_name\_\_ == '\_\_main\_\_':
16. import random
17. lyst = [random.randint(1, 100) for i in range(20)]
18. print(lyst)
19. insertionSort(lyst)
20. print(lyst)

## 更快的排序算法

### 快速排序O(nlogn)

#### 描述

从列表中间点位置选取一项，称之为基准点

将小于基准点的放置在列表左侧，大于基准点的放置在列表右侧

对于基准点左右的值列表，递归的重复这一个过程

每次遇到少于2各项的一个值列表，就结果这一个过程

#### 代码

1. def quickSort(tlist):
2. quickHelper(lyst,0,len(lyst)-1)
3. def partition(lyst,left,right):
4. middle = (left+right)//2
5. pivot = lyst[middle]
6. lyst[middle] = lyst[right]
7. lyst[right]=pivot
8. boundary=left
9. for index in range(left,right):
10. if lyst[index] <pivot:
11. lyst[index],lyst[boundary] = lyst[boundary],lyst[index]
12. boundary +=1
13. lyst[right],lyst[boundary] = lyst[boundary],lyst[right]
14. return boundary
15. def quickHelper(lyst,left,right):
16. if left<right:
17. pivotLocation = partition(lyst,left,right)
18. quickHelper(lyst,left,pivotLocation-1)
19. quickHelper(lyst,pivotLocation+1,right)
20. if \_\_name\_\_ == '\_\_main\_\_':
21. import random
22. lyst = [random.randint(1,100) for i in range(20)]
23. print(lyst)
24. quickSort(lyst)
25. print(lyst)

### 合并排序O(nlogn)

#### 描述

递归的找到列表的中间位置，直至列表大小均为一

递归的将两个排序好的列表组合为一个排序好的列表，直至结束

#### 代码

1. def merge(lyst, copyBuffer, low, middle, higt):
2. i1 = low
3. i2 = middle + 1
4. for i in range(low, higt + 1):
5. if i1 > middle:
6. copyBuffer[i] = lyst[i2]
7. i2 += 1
8. elif i2 > higt:
9. copyBuffer[i] = lyst[i1]
10. i1 += 1
11. elif lyst[i1] < lyst[i2]:
12. copyBuffer[i] = lyst[i1]
13. i1 += 1
14. else:
15. copyBuffer[i] = lyst[i2]
16. i2 += 1
17. for i in range(low, higt + 1):
18. lyst[i] = copyBuffer[i]
19. def mergeSort(lyst):
20. copyBuffer = [0 for i in range(len(lyst))]
21. mergeSortHelper(lyst, copyBuffer, 0, len(lyst) - 1)
22. def mergeSortHelper(lyst, copyBuffer, low, higt):
23. if low < higt:
24. middle = (low + higt) // 2
25. mergeSortHelper(lyst, copyBuffer, low, middle)
26. mergeSortHelper(lyst, copyBuffer, middle + 1, higt)
27. merge(lyst, copyBuffer, low, middle, higt)
28. if \_\_name\_\_ == '\_\_main\_\_':
29. import random
30. lyst = [random.randint(1, 100) for i in range(20)]
31. print(lyst)
32. mergeSort(lyst)
33. print(lyst)

## 指数算法，递归式的斐波那契带来的启示

### 问题

1. def fib(n):
2. if n<3:
3. return 1
4. else:
5. return fib(n-1) + fib(n-2)

在递归的计算斐波那契数列时，虽然程序很优雅，但是会产生对此递归调用，当问题规模变大时，会重复多次的调用子函数，情况会变得很糟糕，

### 改善

将递归调用，变更为循环相加，改方法可以将算法的性能提升到线性阶，由O（Kn） k≈1.63提升为O（n）

1. def fib2(n):
2. if n<3:
3. return 1
4. sum = 2
5. second = 1
6. count = 3
7. while count < n:
8. first = second
9. second = sum
10. sum = first+second
11. count +=1
12. return sum

## 算法探查器

探查是通过统计指令或者执行的时间来度量一个算法性能的过程

## 各种排序算法的比较

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **分类** | **算法** | **最好时间复杂度** | **最差时间复杂度** | **平均时间复杂度** | **空间复杂度** | **稳定性** | **关联性** |
| 插入排序 | 直接插入排序 | O(n)(优化后) | O(n2) | O(n2) | O(1) | 稳定 |  |
| 希尔排序 | O(n) | O(n2) | 不确定 | O(1) | 不稳定 | 基于直接插入排序 |
| 选择排序 | 直接选择排序 | O(n2) | O(n2) | O(n2) | O(1) | 不稳定 |  |
| 堆排序 | O(nlogn) | O(nlogn) | O(nlogn) | 就地排序-O(1) | 不稳定 | 应用了选择的理念 |
| 交换排序 | 冒泡排序 | O(n)(优化后) | O(n2) | O(n2) | O(1) | 稳定 |  |
| 快速排序 | O(nlogn) | O(n2) | O(nlogn) | 最好O(logn)，最差O(n) | 不稳定 | 基于冒泡排序 |
| 归并排序 | 归并排序 | O(nlogn) | O(nlogn) | O(nlogn) | O(n) | 稳定 |  |
| 基数排序 | 基数排序 | O(d\*(n+r))d是位数，r是基数，n是比较的数目 | O(d\*(n+r)) | O(d\*(n+r)) | O(n+r) | 稳定 |  |

## 小结

* 通常认为需要较少的运行时间及较少的运行内存的算法要优于那些需要较多资源的算法，但是通常需要在运行时间与所需内存之间进行权衡
* 可以时间用计算机时钟或者指定模块来估计算法的时间
* 计数指令提供了度量算法的工作量的另一种估计方法，指令基数可以显示出算法工作量递增和递减的速率，而且这个是独立于软硬件平台的
* 算法工作量增加的速率，可以用问题规模的函数来标识，复杂度分析要查看代码，仪表得出这些结论，这个表达式能使得人们能预计在任何计算机上执行一个算法会表现的多好或多差
* 大O表示法是算大运行时间性能的一种常用表示方法，这种表示法使用O(f(n))的形式，其中n是算法的问题大小，f(n)是要解决该问题所需工作量的一个表达式
* 运行时行为的常用表达式是 O（log2n）（对数阶）、O（n）（线性阶）、O（n2）（平方阶）、O（kn）（指数阶）
* 算法性能有不同的表现，最好情况、最坏情况、平均情况，例如，冒泡、插入培训最好情况是线性阶的，但是平均情况和最坏情况都是平方阶的
* 通常，减少算法复杂度的阶比调整代码以增强其性能更好
* 二叉搜索比一般搜索要快，但是二叉搜索的前提是空间内的数据是要先排序的
* n log n 排序算法是使用一种递归的、分治策略来突破呢n2的障碍
* 指数阶的算法在理论上对于处理较大规模的问题是不切实际的

# 数组和链表结构

## 数组数据结构

**数组简述**

数组表示的是可以在给定特定索引位置访问或替代的项的一个序列，例如python的list

**连续内存与随机访问**

计算机通过为数组项分配一段连续内存，在随机访问中，计算机通过执行一定数目的步骤，获取了第i项的位置，因此，不管数组多大，他访问的第一项和任意一项的时间都是相同的

**静态内存**

例如 pascal层级的语言，数组是静态结构，数组的长度和大小在编译的时候就已经确定

**动态内存**

例如java允许程序员创建动态数组来弥补这一个问题，动态数组占用了连续的内存并且支持随机访问，python是自动调整的

**物理大小**

组成数组单元的总数或者是创建数组的之后，用来指定其容量的数字

**逻辑大小**

数组当前可供应用程序使用的项（非空项）的数量

## 数组的操作

**增加数组的大小**

* 创建一个新的、更大的数组
* 将数据从旧的数组复制到新的数组
* 将旧的数组变量重新设置成为新的数组对象

**减小数组的大小**

* 创建一个新的、更小的数组
* 将数据从旧的数组复制到新的数组
* 将旧的数组变量重新设置成为新的数组对象

**在数组中插入一项**

* + 检查数组的可用空间
  + 从数组的末尾开始，将每一项向后移动一个位置，直到目标位置
  + 将新的项复制给目标位置
  + 将逻辑大小增加1

**从数组中删除一项**

* + 从目标位置+1的位置开始，将后面的项依次向前移动一个位置，直至最后一个项
  + 将逻辑大小减少1
  + 检查空间是否浪费

**装载因子**

数组中存储的项的数目除以数组的容量

## 链表结构

**节点**

链表的基本单位是节点。

单链表的节点包含一个数据项、到结构中下一个节点的链接。

双链表的节点包含到结构中上一个节点的链接、数据项、到结构中下一个节点的链接。

**单链表**

通过沿着一个外部的头链接来访问第一个项，然后通过第一个项产生的、串联起来的、单个的链条来访问其他的项，最后一个节点的next指向为None

**双链表**

双链表包含了两个方向的链接，因此用户很容易移动到一个项的后继项和前驱项，第二个外部链接叫做尾链接，因此双链表允许直接访问最后一个项，双链表第一个项没有指向前驱项的链接

**与数组的对比**

* 都表示了项的线性序列
* 使用链表无法使用位置索引直接访问链表中的吗某一项
* 链表一旦找到一个参入或者删除的点，就可以直接进行插入和删除，而不需要移动其他数据项
* 在每一次插入和删除的过程中，链表结构会调整大小，并且不需要额外的内存代价，也不需要复制数据项
* 数组需要连续的内存，链表不需要
* 链表不支持随机访问

## 单链表结构上的操作

**创建一个单列表**

1. class Node(object):
2. def \_\_init\_\_(self,data,next = None):
3. self.data = data
4. self.next = next
5. if \_\_name\_\_ == '\_\_main\_\_':
6. head = None
7. for count in range(1,6):
8. #<do something>
9. head = Node(count,head)

**遍历**

依次访问链表的每个节点，当遇到一个空连接的时候终止，None作为终止的哨兵

1. def traversal(self,head):
2. probe = head
3. while probe != None:
4. print(probe.data)
5. probe = probe.next

**搜索**

依次访问链表的每个节点，当遇到一个空连接后目标值的时候终止，None、目标值作为终止的哨兵

1. def search(head,target):
2. probe = head
3. while probe != None and target != probe.data:
4. probe = probe.next
5. if probe != None:
6. return 1  # target in linked structure
7. else:
8. return 0  # target not in linked structure

**替换**

依次访问链表的每个节点，当遇到一个空连接后目标值的时候替换成新的数值，None、目标值作为终止的哨兵

1.查找到替换

2.给定索引替换

1. def replace(head,newItem,target=None,index=None):
2. if target!=None:  #查找目标
3. probe = head
4. while probe != None and target != probe.next:
5. probe =probe.next
6. if probe != None:
7. probe.data = newItem
8. return True
9. else:
10. return False
11. if index != None:  #给定索引
12. # 0<index<len
13. probe = head
14. while index>0:
15. probe=probe.next
16. index -=1
17. probe.data=newItem

**在开始插入**

* 空列表：直接插入
* 非空列表：获取 head的next，将head的next指向新的节点，将新的节点的next指向原head的next

1. def insert(head,newItem):
2. head = Node(newItem, head)

**在结尾插入**

* 空列表：直接插入
* 非空列表：搜索到最后一个节点（next指向None），将改节点的next指向新节点

1. def insert(head,newItem):
2. newNode =Node(newItem)
3. if head is Node:
4. head =newNode
5. else:
6. probe = head
7. while probe.next!=None:
8. probe=probe.next
9. probe.next = newNode

**在任何位置插入**

* 该节点的next指向None：将其next指向新的节点，新节点的next仍为None
* 该节点的next指向非None：获取next并将其next指向新的节点，新节点的next从原获取到的next

1. def insert(head,newItem):
2. if head is None or index <=0:
3. head = Node(newItem,head)
4. else:
5. probe = head
6. while probe.next != None and index >1:
7. probe = probe.next
8. index  -=1
9. probe.next = Node(newItem,probe.next)

**在开始处删除**

获取第一个节点的nexe，将head的next指向为原第一个节点的next

1. def delete():
2. removedItem = head.data
3. head = head.next
4. return removedItem

**在结尾处删除**

* 只有一个节点，head的next指向为None
* 代码搜索到倒数第二个节点并将其next指向为None

1. def delete():
2. removedItem = head.data
3. if head.next is None:
4. head=None
5. else:
6. probe = head
7. while probe.next.next != None:
8. probe=probe.next
9. removedItem =probe.next.data
10. probe.next=None
11. return removedItem

**从任意位置删除**

i≤0：删除第一项代码

0<i<len：搜索到i-1的节点，删除其后面的一个节点

i≥n：删除最后一个节点

1. def delete(head ,index):
2. if index <0 or head.next is None:
3. removedItem = head.data
4. head=head.next
5. return removedItem
6. else:
7. probe = head
8. while index>1 and probe.next.next != Node:
9. probe=probe.next
10. index -=1
11. removedItem =probe.next.data
12. probe.next=probe.next.next
13. return removedItem

**链表的优点**

链表的优点不是时间性能，而是内存性能，当调整数组的大小的时候，其时间和内存都是线性的，当调整链表结构的大小时，其时间和内存都是常数的，此外。链表结构中没有浪费内存的问题，链表的物理大小不会超过其逻辑大小

**链表的缺点**

链表有额外的内存代价，因为链表的结构要为指针使用n个内存单元格，这个代价在双链表中更变本加厉，因为双链表有两个链接

## 链表的变体

### 带有一个哑头节点的循环链表结构

**格式**

手机屏幕截图

描述已自动生成

**优点**

插入和删除操作只需要考虑一种情况，无需考虑在第一位重置head的特殊情况

### 双链表结构

**优点**

从给定的点向前移动

直接移动到最后一个点

**缺点**

双链表结构中需要额外的指针

需要额外的线性的内存使用量

## 小结

* 数据结构是表示一个集合中包含的数据的一个对象
* 数组是一个数据集合，它支持按照位置对某一项随机访问，且这种访问时间是常数
* 在创建数组的时候，给定了用于存储数据的位置的一个数目，并且数组的长度保持固定。插入和删除需要移动的元素，创建一个新的，较大的或较小的数组，可能也需要移动数据元素
* 二维数组通过矩形网格中的行和列来定位每一个数据值
* 链表结构一个数据结构，它包含了0个或多个节点，一个节点包含了一个数据项，以及到其他节点的一个或多个链接
* 单链表结构的节点包含了一个数据项和到下一个节点的一个链接，双链表结构中的节点还包含了一个到前一个节点的链接
* 在链表结构中插入和删除操作，不需要移动数据元素，最多只需要创建一个节点。在线性的链表结构中插入、删除和访问等操作，所需时间都是线性的
* 在链表结构中，使用头节点可以简化一些操作，例如，添加或删除第一个项的操作

# 接口、实现和多态

## 接口简述

**接口功能**

* 降低了用户的学习难度
* 允许用户以即插即用的方式，快速的将资源整合起来
* 让用户有机会在相同资源的不同实现中做出选择
* 允许具体实现者对资源的实现做出修改，而不影响其用户的代码

**先验条件**

先验条件是这样一条语句，只有该语句为真的时候，方法才能正确的执行操作

例如：删除一个元素的先验条件是这个元素存在于集合中

**后验条件**

后验条件说明了，当方法执行完毕后，什么条件会为真

例如：清除一个集合的后验条件是集合为空

**java 、python接口模板**

java提供了编写接口的语法，java接口本身不执行操作，但是它提供了实现类方法所必须遵守的一个模板，python没有这样的功能

**接口自身实现和设计的步骤**

1. 选择一个合适的数据结构来包含集合的项，并且确定可能需要表示集合状态的任何其他数据，并将这些数据赋值给\_\_init\_\_方法中的实力变量
2. 完成接口中指定方法的代码

## 开发一个基于数组的接口实现

尽量调用其他方法而将变量引用保持最小化，以便复用时修改最小

### 选择并初始化数据

1. class Array(object):
2. def \_\_init\_\_(self,capacity,fillValue=None):
3. self.\_items=list()
4. for count in range(capacity):
5. self.\_items.append(fillValue)
7. class ArrayBag(object):
9. DEFAULT\_CAPACITY =10
10. def \_\_init\_\_(self,sourceCollection =None):
11. self.\_item = Array(ArrayBag.DEFAULT\_CAPACITY)
12. self.size = 0
13. if sourceCollection:
14. for item in sourceCollection:
15. self.add(item)

### 完成容易的方法

1. def isEmpty(self):
2. return len(self) == 0
4. def \_\_len\_\_(self):
5. return self.size
7. def clear(self):
8. self.size = 0
9. self.\_item = Array(ArrayBag.DEFAULT\_CAPACITY)
11. def add(self, item):
12. self.\_items[len(self)] = item
13. self.size += 1

### 完成迭代器

1. def \_\_iter\_\_(self):
2. cursor = 0
3. while cursor < len(self):
4. yield self.\_items[cursor]
5. cursor += 1

### 完成使用迭代器的方法

1. def \_\_str\_\_(self):
2. return "{" + ", ".join(map(str, self)) + "}"
4. def \_\_add\_\_(self, other):
5. result = ArrayBag(self)
6. for item in other:
7. result.add(item)
8. return result
10. def \_\_eq\_\_(self, other):
11. if self is other:
12. return True
13. if type(self) != type(other) or len(self) != len(other):
14. return False
15. for item in self:
16. if not item in other:
17. return False
18. return True

### TO DO:in运算符和\_\_contains\_\_方法

### 完成remove方法

1. def remove(self, item):
2. if item not in self:
3. raise KeyError(str(item) + "not in bag")
4. targetIndex = 0
5. for targetItem in self:
6. if targetItem == item:
7. break
8. targetIndex += 1
9. for i in range(targetIndex, len(self) - 1):
10. self.\_items[i] = self.\_items[i + 1]
11. self.size -= 1

## ⏸TO:DO 开发一个基于链表的接口实现

## 小结

* 接口是软件资源用户可用的一组操作
* 接口中的内容是函数头和方法头，以及他们的文档
* 先验条件表明了在函数和方法能够正确完成其任务之前所必须为真的条件
* 后延条件表明了在函数和方法正确完成其任务之后所必须的条件为真
* 设计良好的软件系统会将接口与其实现分隔开来
* 实现是必须符合接口的函数、方法或类
* 可以根据接口来指定集合类型
* 一个集合类型拥有几个实现类
* 多态是在两个或多个实现中使用相同的运算符号、函数名或方法。
* 多态的函数示例是str和len。
* 多态运算符的函数示例是+和==。
* 多态的方法示例是add和isEmpty。
* 包集合类型是无序的，并且支持添加项、删除项和访问项的等操作
* 类图是描述类之间的关系的一种可视化表示方法
* 组合表示两个类之间的整体——部分关系
* 聚合表示两个类之间的一对多关系
* UML是描述软件资源之间关系的一种图形化表示方法

# TODO

### yield的使用

### 再看一遍 完成迭代器

### 第三章的练习题目

### 第四章的练习题目

## 算法探查器- Python性能分析工具Profile

## Timeit模块