作业 7 问题回顾

二分查找、哈希表、快速排序

李睢

sui@pku.edu.cn

May 18, 2021

目录

编程习惯

- 每一个变量都意味着后续所有编码需要随时维护的“不变量”，设计时要谨慎，避免作用重复或不易使用的变量
- 所有常数需要定义成变量，方便后续维护修改（例如扩容阈值）
- 重复的代码要定义成函数，避免复制粘贴，减少错误、方便修改（例如哈希函数）
- 访问变量前考虑检查 None、数组越界、容器为空等
- 高复杂度的操作尽量不做，做一次够用就决不做第二次

```
def get(self, key: int) -> int:
hashvalue = key % self.table_size
if key in self.slots[hashvalue]: # O(n)
    return self.data[hashvalue][self.slots[hashval」
    ↪  ue].index(key)] # 再次O(n)
return None
```

语法问题

- 确认一个变量不是 None
  - 正确做法 if a is not None
  - 错误做法 if a == None
  - 错误做法 if a
- 分清对象的变量（对象独享）与类的变量（所有对象共享）

```
class Set:
    nums = 0
    size = 11
    # 写在这里的变量被所有 Set 对象共享，某一个对象的
    ↪  变量变了其他的对象都会变
```

语法问题

- 避免重复的条件判断

```
if self.slots[hashvalue] is None:
    return False
elif self.slots[hashvalue] is not None: # 万一写错了
↪   怎么办
    ...
```

- 分类讨论的情况，尽量用 else 或 elif，不要并列 if

```
hashvalue = self.rehash(hashvalue)
if self.slots[hashvalue] is None:
    return False # 如果这里其他操作可能改变 if 的条件
if self.slots[hashvalue] == key:
    return True
```

语法问题

- 避免多余的条件判断

```
def __contains__(self, key: int) -> bool:
    ret = self.get(key)
    if ret == None:
        return False
    else:
        return True
    # return self.get(key) is not None
```

- 避免多余的相等判断 if x == True

```
if self.slots[hashvalue].init(key) == True
```

语法问题

- 避免多余的条件判断

```
def add(self, key: int) -> None:
hashvalue = self.hashfunction(key)
if self.slots[hashvalue] is None:
    self.slots[hashvalue] = key
    self.full = self.full + 1
elif self.slots[hashvalue] is not None:
    # 以上几行 if-elif 都是多余的
    while self.slots[hashvalue] is not None and
    ↪  self.slots[hashvalue] != key:
        hashvalue = self.rehash(hashvalue)
        if self.slots[hashvalue] is None:
            self.slots[hashvalue] = key
            self.full = self.full + 1
```

语法问题

- return 语句不一定只出现在函数的最后
- 下例 found、stop 变量应换成在合适的时候 break 或 return

```
def __contains__(self, key: int) -> bool:
hashvalue = self.hashfunction(key)
found = False
stop = False
position = hashvalue
while self.slots[position] is not None and not found and
↪   not stop:
    if self.slots[position] == key:
        found = True              # return True
    else:
        position = (position + 1) % self.table_size
        if position == hashvalue:
            stop = True           # return False
return found                      # return False
```

代码风格

- 二元运算符（+-*/%= 等等）左右两边添加空格
- 逗号后加空格
- 函数之间要有空行分隔
- python 的变量命名习俗：
  - 变量名、包名、函数名：**蛇形命名法（snake_case）**
  - 类名、异常名：大驼峰命名法（UpperCamelCase）
  - 全局常量：尖叫的蛇形命名法（SCREAMING_SNAKE_CASE）

```
module_name, package_name, ClassName, method_name,
↪ ExceptionName, function_name, GLOBAL_CONSTANT_NAME,
↪ global_var_name, instance_var_name,
↪ function_parameter_name, local_var_name
```

二分查找 – 题目描述

- 实现二分查找算法，从整数的数组 nums 中，寻找目标 target。如果 target 在数组中，返回 target 所在的下标；否则返回 $-1 - idx$，其中 idx 是 target 应被插入的位置的下标。数组 nums 中不包含重复元素。
- 代码模板：bsearch.py，请把代码填写到对应的位置，并在注释中对自己的实现做出简单的解释。请在提交前做充分的测试以确认程序的正确性。
- 样例输入：nums = [-1,0,3,5,9,12], target = 9
- 样例输出：4
- 样例输入：nums = [1,3,5,6], target = 2
- 样例输出：-2 （target 应被插入在 3 的位置，下标为 1，-1-1=-2）
- 可参考 Leetcode 相关题目来测试程序的正确性：
  - 35. 搜索插入位置
  - 704. 二分查找

二分查找

- 实现细节
  - 时刻想清楚：“不变量”是什么？开还是闭区间？
  - 确保循环终止：保证每次问题规模至少减小 1
  - 重点分析算法即将终止的、特殊的情形
- 常见问题
  - 没有考虑清楚边界情形，导致无法处理某些数据
  - 做了过多多余的边界处理，或多余的额外的比对
  - 数组切片操作是 O(n) 的，不可以在 O(log(n)) 的算法中使用
  - 结束状态的前一步不一定是 left==right，也可能是 left+1=right。例如 nums=[1,2], target=0

二分查找 – 测试样例

1. [], 0
2. [0], 0
3. [0, 1, 2, 3, 4, 5, 6, 7], 6
4. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 10
5. [0, 2, 4, 6, 8, 10, 12, 14, 16, 18], -1
6. [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24], 9
7. [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28], 17
8. 10000 内随机 1000 个数，target 不在数组中
9. 100000 内随机 10000 个数，target 不在数组中
10. 10000000 内随机 3000000 个数，target 不在数组中

二分查找 – 参考代码 1

```
def bsearch(nums, target):
    left = 0
    right = len(nums) - 1
    while left <= right:
        mid = (right + left) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1 - left
```

- “不变量”
  - 若 target 在 nums 中，则其下标在 [left, right] 中
  - 大于 target 的最小元素下标在 [left, right + 1] 中
  - 小于 target 的最大元素下标在 [left - 1, right] 中
- 终止情形
  - left = right + 1
  - 若 target 不在 nums 中
    - left 为大于 target 最小的元素下标
    - right 为小于 target 最大的元素下标

right和left指针可以交叉

二分查找 – 参考代码 2

```python
def bsearch(nums, target):
    left = 0
    right = len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return -1 - left # or -1 - right
```

二分查找 – 参考代码 3

```
def bsearch(nums, target):
    left = -1
    right = len(nums)
    while left + 1 < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid
        else:
            right = mid
    return -1 - right
```

二分查找 – 应用举例

- nums 是单调递增的数组，找到其中第一个大于等于 target 的元素
- nums 是单调递增的数组，找到与 target 最接近的 k 个元素

找到一个位置然后对k再进行扫描，至少有一边有k//2长度

- nums 是单调递增的数组，其中数字可重复，找到 target 出现第一次、最后一次的位置
- nums 是一个先递增后递减的数组，找到其中的最大值
- nums 是一个由单调递增的数组“旋转”（把最后一个元素挪到了头部）几次得到的数组，找到其中最小的元素

Map – 题目描述

- 采用数据链（chaining）的冲突解决技术来实现 ADT Map（功能类似 python 内置的字典 dict），其中 key 和 value 均为整数，需要实现的方法包括：
  - `m=Map()`：创建对象，具体实现为构造函数 `__init__(self)`
  - `m.put(key, value)`：将 key-value 关联加入映射
  - `m.get(key)`：查找关联的数据值，若不存在返回 None
  - `m.remove(key)`：删除 key 对应的 value 关联，不存在则直接返回
  - `m[key]=value`：调用 put 方法。具体实现在 `__setitem__(self, key, value)`
  - `m[key]`：调用 get 方法。具体实现在 `__getitem__(self, key)`
  - `del m[key]`：调用 remove 方法。具体实现在 `__delitem__(self, key)`
  - `len(m)`：返回 m 中 key-val 对的个数，具体实现在 `__len__(self)`
  - `key in m`：返回布尔值表示 key 是否存在于 m 的 key 中。具体实现在 `__contains__(self, key)`
- 代码模板：map.py，请把代码填写到对应的位置，并在注释中对自己的实现做出简单的解释。请在提交前做充分的测试以确认程序的正确性。
- 可参考 Leetcode 相关题目来测试程序的正确性：
  - 706. 设计哈希映射

Map 实现细节

- 用什么数据结构实现基于数据链的冲突处理?
  - 数组：查找 O(n), 插入 O(1)
  - 有序数组：查找 O(log(n)), 插入 O(n)
  - 链表：查找 O(n), 插入 O(1)
  - 有序链表：查找 O(n), 插入 O(1)
  - 二叉搜索树：查找 O(log(n)), 插入 O(log(n))
  - 哈希表：查找 O(1), 插入 O(1)，但空间开销大，后续仍需冲突处理
- O(1) 时间的 len 方法：设立一个变量作为计数器，每次添加元素时大小加 1，删除时大小减 1
  - 注意什么时候应该增加，什么时候不应该增加
  - 扩容时注意该变量的维护

Map – 常见问题

- ___len___(self) 不是 O(1) 的
- contains(self, key) 直接扫描，没有用哈希
- put() 在 key 已经存在的时候重复插入了
- 没有用题目要求的数据链，而用了下一题的开放定址冲突解决技术
- 复制粘贴自己以前作业的链表实现，结果里面有 bug
- 二维数组的初始化

```
slots=[[]] * self.table_size                    # wrong
slots=[[] for _ in len(self.table_size)]        # good
slots=[[None] for _ in len(self.table_size)]    # ?
```

Map – 常见问题

- 变量设计过于复杂，key 和 value 分开在 slots 和 data 两个链表中存储，导致代码重复过多而出错

```
if previous_slots == None:
    self.slots[hashvalue] = current_slots.get_next()
    self.data[hashvalue] = current_slots.get_next()
else:
    previous_slots.set_next(current_slots.get_next())
    previous_data.set_next(current_slots.get_next())
```

  解决方法

  - key 和 value 存到同一个 Node 中，避免两条链表
  - 链表常驻伪头节点（dummy head），避免当前节点是否位于链表头的额外条件判定

Map – 测试样例

1. put 1 10 put 2 20 get 1 contains 100005 len
2. put -1 -10 len get -1 remove -1 len get -1
3. contains 1 get 1 remove 1 put 1 10 len get 1
4. put 1 20 put 1 30 len get 1 remove 1 len get 1
5. put 1 10 put 100004 20 put 200007 30 len get 200007 len get 100004
6. put 1 10 put 100004 20 put 200007 30 len remove 1 len get 1 get 200007
7. put 1 10 put 100004 20 put 200007 30 len remove 100004 len get 1 get 200007
8. 随机 1000 次操作，key 为 10 以内的数及额外 2 个与其冲突的数
9. 随机 10000 次操作，key 为 100 以内的数及额外 4 个与其冲突的数
10. 随机 100000 次操作，key 为 10000 以内的数及额外 10 个与其冲突的数

Map 参考代码 1 – 链表

```
class Map:
    class Node:
        def __init__(self, key=None, value=None):
            self.key = key
            self.value = value
            self.next = None

    def __init__(self):
        self.table_size = 100003
        self.slots = [None] * self.table_size
        self.size = 0

    def hash(self, key: int) -> int:
        return key % self.table_size

    def put(self, key: int, value: int) -> None:
        idx = self.hash(key)
```

Map 参考代码 1 – 链表

```
# 槽为空，建立链表头结点
if self.slots[idx] is None:
    self.slots[idx] = self.Node(key, value)
    self.size += 1
else:
    # 寻找当前 key 直到链表末尾
    cur = self.slots[idx]
    while cur.key != key and cur.next is not None:
        cur = cur.next
    if cur.key != key:
        # 没找到，在末尾添加
        cur.next = self.Node(key, value)
        self.size += 1
    else:
        # 已存在，直接赋值
        cur.value = value
```

Map 参考代码 1 – 链表

```
def get(self, key: int) -> int:
    idx = self.hash(key)
    if self.slots[idx] is None:
        return None
    cur = self.slots[idx]
    while cur.key != key and cur.next is not None:
        cur = cur.next
    if cur.key != key:
        return None
    else:
        return cur.value

def remove(self, key: int) -> None:
    idx = self.hash(key)
    if self.slots[idx] is None:
        return
    prev = None
```

Map 参考代码 1 – 链表

```
        cur = self.slots[idx]
        while cur.key != key and cur.next is not None:
            prev = cur
            cur = cur.next
        if cur.key != key:
            return
        else:
            if prev is None:
                self.slots[idx] = cur.next
            else:
                prev.next = cur.next
            self.size -= 1

    def __setitem__(self, key: int, value: int) -> None:
        self.put(key, value)

    def __getitem__(self, key: int) -> int:
```

Map 参考代码 1 – 链表

```
        return self.get(key)

    def __delitem__(self, key: int) -> None:
        self.remove(key)

    def __len__(self) -> int:
        return self.size

    def __contains__(self, key: int) -> bool:
        idx = self.hash(key)
        if self.slots[idx] is None:
            return False
        cur = self.slots[idx]
        while cur.key != key and cur.next is not None:
            cur = cur.next
        return cur.key == key
```

Map 参考代码 2 – list

数院 20 林逸云

```
class Map:
def __init__(self):
    self.table_size = 100003
    self.slots = [[] for _ in range(self.table_size)]
    self.data = [[] for _ in range(self.table_size)]
    self.size=0

def put(self, key: int, value: int) -> None:
    hashvalue=key%self.table_size
    for i in range(len(self.slots[hashvalue])):
        if key==self.slots[hashvalue][i]:
            self.data[hashvalue][i]=value
            return
    self.slots[hashvalue].append(key)
    self.data[hashvalue].append(value)
    self.size+=1
```

Map 参考代码 2 – list

数院 20 林逸云

```
def get(self, key: int) -> int:
    hashvalue=key%self.table_size
    for i in range(len(self.slots[hashvalue])):
        if key==self.slots[hashvalue][i]:
            return self.data[hashvalue][i]
    return None

def remove(self, key: int) -> None:
    hashvalue=key%self.table_size
    for i in range(len(self.slots[hashvalue])):
        if key==self.slots[hashvalue][i]:
            del self.slots[hashvalue][i]
            del self.data[hashvalue][i]
            self.size-=1
            return
```

Map 参考代码 2 – list

数院 20 林逸云

```
        return

    def __setitem__(self, key: int, value: int) -> None:
        self.put(key,value)

    def __getitem__(self, key: int) -> int:
        return self.get(key)

    def __delitem__(self, key: int) -> None:
        self.remove(key)

    def __len__(self) -> int:
        return self.size

    def __contains__(self, key: int) -> bool:
        hashvalue=key%self.table_size
```

Map 参考代码 2 – list

数院 20 林逸云

```
for i in range(len(self.slots[hashvalue])):
    if key==self.slots[hashvalue][i]:
        return True
return False
```

Set – 题目描述

- 采用开放定址冲突解决技术的散列表，课件中是固定大小的。请用散列表实现一个 ADT Set（功能类似 python 内置的集合 set），使得在负载因子达到某个阈值之后，散列表的大小能自动增长。散列表中的元素均为整数，需要实现的方法包括：
  - `s=Set()`：创建对象，具体实现为构造函数 `__init__(self)`
  - `s.add(key)`：将 key 加入集合
  - `key in s`：返回布尔值表示 key 是否存在于集合中。具体实现在 `__contains__(self, key)`
  - `len(s)`：返回 s 中不同 key 的个数，具体实现在 `__len__(self)`
  - 不要求：`s.remove(key)`：从集合中删除 key，不存在则直接返回。remove 方法在这种实现中比较复杂，不做强制要求
- 代码模板：set.py，请把代码填写到对应的位置，并在注释中对自己的实现做出简单的解释。请在提交前做充分的测试以确认程序的正确性。
- 可参考 Leetcode 相关题目来测试程序的正确性：
  - 705. 设计哈希集合

Set 实现细节 – 扩容

- 负载达到一定比例（例如 70%）就要扩容，否则会导致哈希表的查找效率严重下降
- 每次扩容增加的大小要与当前大小成比例（例如翻倍），这样才可以保证每次 add 操作的均摊时间复杂度为 O(1)
- 扩容是影响比较大的操作，注意“不变量”的维护
  - 每次扩容要重新插入所有元素，保证哈希性质正确（否则需要修改查找的办法，并导致查找复杂度剧增）
  - 注意维护计数器变量，避免在重插入时导致其额外增加

Set 实现细节 – remove

- 核心问题是删掉当前元素后，需要保证能够找到之前因为冲突而被放到当前位置之后的元素
- 三个思路
  1. 从删掉之后留下的空位开始继续向后扫描有内容的位置，将后面的元素向前挪动，补上空缺的位置
     - 但是，一些元素如果被挪到比自身哈希值小的位置会导致永远找不到，所以只有哈希值比空位小的元素才可以挪到空位处。
     - 由于哈希表下标位置是循环的，这个判断比较难以想清楚
  2. 将删掉的元素用占位符替代，之后的 add、contains、remove 寻找元素时均需要把占位符当作已占用的格子继续扫描
     - 但是，占位符占用大量空间而不计数，必须有个办法重复利用空白的位置
     - 如果每次插入方法直接插入在第一次找到的占位符处，可能导致后面还有重复元素的问题，必须向后寻找可能存在的相同元素并删除
     - 即使这样，大量占位符的存在仍然大幅降低了程序的效率，可能需要定期清理（在保证均摊 O(1) 前提下）

Set 实现细节 – remove

3. 额外用一个数组存储每个位置因添加时因冲突而向后的数量，以决定只有的 add、contains、remove 方法是否向后寻找
   - 但是，删掉的元素占用空间而不计数，拖慢查找效率，需要类似上一种方法，考虑各种额外的处理和优化

Set – 测试样例

1. add 1 add 2 contains 1 contains 12
2. add -1 len contains -1 add 10 len contains -1 contains 10
3. contains 1 add 1 contains 1 add 1 len contains 1
4. add 1 add 12 add 23 contains 12 len contains 2
5. add 1 add 3 add 0 add 11 add 23 contains 23 contains 3 len
6. add 1 add 3 add 5 add 12 contains 12 add 23 contains 23 add 34 contains 34 add 45 add 56 add 67 add 78 add 89 contains 89 contains 1
7. add 0 add 1 add 2 add 3 add 4 add 5 add 6 add 7 add 8 add 9 add 10 add 11 add 12 add 13 add 14 add 15 add 16 add 17 add 18 add 19 add 100 add 101 add 102 add 103 add 104 len contains -1 contains 20 contains 3 contains 7 contains 102
8. 随机 1000 次操作，key 为 110 以内的数
9. 随机 10000 次操作，key 为 1100 以内的数

Set – 测试样例

10. 随机 300000 次操作，key 为 11000 以内的数
11. 20% 概率 remove，随机 30000 次操作，key 为 110 以内的数
12. （额外试验，无人通过）50% 概率 remove，随机 500000 次操作，key 为 110000 以内的数

Set 参考代码

```python
class Set:
    # 全局常量
    TABLE_LOAD_THRESHOLD = 0.5
    TABLE_GROWTH_RATE = 2

    def __init__(self):
        self.table_size = 11
        self.slots = [None] * self.table_size
        self.size = 0

    def hash(self, key: int) -> int:
        return key % self.table_size

    def rehash(self, old_hash: int) -> int:
        return (old_hash + 1) % self.table_size
```

Set 参考代码

```
def add(self, key: int) -> None:
    # 找到可能的插入位置
    i = self.hash(key)
    while self.slots[i] is not None and self.slots[i] != key:
        i = self.rehash(i)
    # key 已存在，返回
    if self.slots[i] == key:
        return
    # 插入
    self.slots[i] = key
    self.size += 1
    # 扩容，注意成员变量的维护
    if self.size / self.table_size >= TABLE_LOAD_THRESHOLD:
        old_slots = self.slots
        self.table_size *= TABLE_GROWTH_RATE
        self.slots = [None] * self.table_size
        self.size = 0
```

Set 参考代码

```
            for slot in old_slots:
                if slot is not None:
                    self.add(slot)


def __contains__(self, key: int) -> bool:
    i = self.hash(key)
    while self.slots[i] is not None and self.slots[i] != key:
        i = self.rehash(i)
    return self.slots[i] == key

def __len__(self) -> int:
    return self.size
```

Set 参考代码

```
def remove(self, key: int) -> None:
    # 找到 key 可能在的位置
    i = self.hash(key)
    while self.slots[i] is not None and self.slots[i] != key:
        i = self.rehash(i)

    # 未找到，返回
    if self.slots[i] != key:
        return

    # 删除，得到一个空位
    self.slots[i] = None
    self.size -= 1


    # 反复寻找哈希值不大于空位的元素，填补空位
```

Set 参考代码

```
while True:
    j = self.rehash(i)
    while self.slots[j] is not None:
        hash_sj = self.hash(self.slots[j])
        if i < j and hash_sj <= i: 证明j之前都是连续的
            break
        if i > j and hash_sj <= i and hash_sj > j:
            break                    找到边界又返回，但要保证j仍然是
        j = self.rehash(j)               在其哈希值之后才可以挪
    if self.slots[j] is None:
        return
    self.slots[i] = self.slots[j]
    self.slots[j] = None
    i = j
```

Set 不同的 remove 实现 1

物院 18 沈定宇

```
def remove(self, key: int) -> None:
    # 本程序还实现了 remove 方法
    # 减少元素时判断是否需要减容
    if self.length < self.min_load_factor * self.table_size
    ↪  and self.length > self.default_size:
        self.antiresize()
    hashvalue = self.hashfunction(key)
    while self.slots[hashvalue] != None and
    ↪  self.slots[hashvalue] != key:
        hashvalue = self.rehash(hashvalue)
    if self.slots[hashvalue] == key:
        self.slots[hashvalue] = None
        self.length -= 1
        # 需要用一个缓存，把其后所有可能由于它而产生冲突的 key
        ↪  全部存起来重新在散列表中分配位置
        keycache = []
```

Set 不同的 remove 实现 1

物院 18 沈定宇

```
        hashvalue = self.rehash(hashvalue)
        nextkey = self.slots[hashvalue]
        self.slots[hashvalue] = None
        while nextkey != None:
            keycache.append(nextkey)
            self.length -= 1
            hashvalue = self.rehash(hashvalue)
            nextkey = self.slots[hashvalue]
            self.slots[hashvalue] = None
        for key in keycache:
            self.add(key)
```

局部有大量冲突时复杂度可能变成平方量级

Set 不同的 remove 实现 2

经院 17 刘安澜

```
class Set:
    def __init__(self):
        self.table_size = 11
        self.slots = [None] * self.table_size
        self.length=0

    def add(self, key: int) -> None:
        hashvalue=key%self.table_size
        nextslot=hashvalue
        written=False
        while self.slots[nextslot]!=None:# 如果冲突就 +1 线性寻找，
        ↪ 由于之前可能存在冲突过的元素被删除，所以结束条件是找到
        ↪ None 才能保证没有重复元素
            if self.slots[nextslot]=='' and not written:# 之前删
            ↪ 除过的空槽插入
                self.slots[nextslot]=key
```

Set 不同的 remove 实现 2

经院 17 刘安澜

```
                written=True# 标记已经填入过元素
            elif self.slots[nextslot]==key:# 如果找到 key 则将其
            ↪  删除，保持 set 的元素唯一性
                if written:
                    self.slots[nextslot]=''
                else:
                    written = True
                self.length-=1
                break
            nextslot=(nextslot+1)%self.table_size
        if not written:
            self.slots[nextslot]=key
        self.length+=1
        if len(self)>=0.5*self.table_size:# 负载因子大于 0.5 时扩
        ↪  列，否则速度会很慢
            self.resize()
```

Set 不同的 remove 实现 2

经院 17 刘安澜

```
        pass

    def resize(self):
        oldtable=self.slots
        self.table_size=self.table_size*5# 将列表扩充成为原来的五
        ↪  倍
        self.slots=[None] * self.table_size
        self.length=0
        for slot in oldtable:# 对于旧列表中的元素一一进行添加
            if slot!=None and slot!='':
                self.add(slot)

    def __contains__(self, key: int) -> bool:
        found=False
        hashvalue=key%self.table_size
        while self.slots[hashvalue]!=None and not found:
```

Set 不同的 remove 实现 2

经院 17 刘安澜

```
            if self.slots[hashvalue]==key:
                found=True
            else:
                hashvalue=(hashvalue+1)%self.table_size# 继续向下
                ↪  线性寻找，直到搜寻到空元素
        return found

    def __len__(self) -> int:
        return self.length

    def remove(self, key: int) -> None:
        hashvalue=key%self.table_size
        while self.slots[hashvalue]!=None:
            if self.slots[hashvalue]==key:
                self.slots[hashvalue]=''
                self.length-=1
```

Set 不同的 remove 实现 2

经院 17 刘安澜

```
                break
            else:
                hashvalue=(hashvalue+1)%self.table_size# 继续向下
                ↪  线性寻找，直到搜寻到空元素
        pass
```

可能需要定期清理多余的占位符，以保证效率不下降

Set 有序版本的实现

化院 18 刘玉昕

```
class Set:
    def __init__(self,table_size =11):
        self.table_size=table_size
        self.size=0
        self.slots=[None] * self.table_size# 槽
        self.items=[]# 下标为 key 放入顺序，值为 key
        self.index=[]# 下标为 key 放入顺序，值为在 slots 中的下标
        self.order=[None]*self.table_size# 下标为 key 在 slot 中
        ↪  的下标，值为 key 放入顺序

    def add(self, key: int) -> None:
        if not self.__contains__(key):
            length=self.table_size
            if self.size/length>0.5:
                length=length*2
                self.resize(length)
```

Set 有序版本的实现

化院 18 刘玉昕

```
self.items.append(key)
hashvalue=self.hashfunction(key,length)
if self.slots[hashvalue]==None:
    self.slots[hashvalue]=key
    self.index.append(hashvalue)
    self.order[hashvalue]=self.size
    self.size+=1
elif self.slots[hashvalue]!=None and
↪   self.slots[hashvalue]!=key:
    nextslot=self.rehash(hashvalue,length)
    while self.slots[nextslot]!=None and
    ↪   self.slots[nextslot]!=key:
        nextslot=self.rehash(nextslot,length)
    if self.slots[nextslot]==None:
        self.slots[nextslot]=key
        self.index.append(nextslot)
```

```
                self.order[nextslot]=self.size
                self.size+=1

    def __contains__(self, key: int) -> bool:
        startslot=self.hashfunction(key,self.table_size)
        found=False
        stop=False
        pos=startslot
        while self.slots[pos]!=None and not found and not stop:
            if self.slots[pos]==key:
                found=True
            else:
                pos=self.rehash(pos,self.table_size)
                if pos==startslot:
                    stop=True
        return found
```

Set 有序版本的实现

化院 18 刘玉昕

```
def __len__(self) -> int:
    return self.size

def remove(self, key: int) -> None:
    startslot=self.hashfunction(key,self.table_size)
    found=False
    stop=False
    pos=startslot
    while self.slots[pos]!=None and not found and not stop:
        if self.slots[pos]==key:# 找到 key
            found=True
        else:
            pos=self.rehash(pos,self.table_size)
            if pos==startslot:
                stop=True
```

Set 有序版本的实现

化院 18 刘玉昕

```
        if self.slots[pos]==key:
            self.slots[pos]=None# 删除
            order=self.order[pos]
            self.size-=1
            #items 与 index 中 order 后面的元素全部向前移
            for i in range(order,self.size):
                self.items[i]=self.items[i+1]
                self.index[i]=self.index[i+1]
            self.items.pop()
            self.index.pop()
            # 重置 slots 中元素的位置
            self.resize(self.table_size)

    def hashfunction(self,key,size):
        return key%size
```

Set 有序版本的实现

化院 18 刘玉昕

```
def rehash(self,oldhash,size):
    return (oldhash+3)%size

def resize(self,length):
    new_slots=Set(length)
    for i in range(self.size):
        new_slots.add(self.items[i])
    self.table_size=new_slots.table_size
    self.slots=new_slots.slots
    self.items=new_slots.items
    self.index=new_slots.index
    self.order=new_slots.order
```

- 建议优化变量结构组织，维护尽量少的变量
- while 循环的写法可以精简
- remove 当前实现效率不佳，可以用占位符 + 定期清理的方式

快速排序 – 题目描述

- 自行设计一种取“中值”的方法实现快速排序。请在注释中写一段说明，比较你的实现与课件中的快速排序算法的性能。输入为整数组成的数组，输出结果为升序排列。
- 代码模板：qsort.py，请把代码填写到对应的位置，并在注释中对自己的实现做出简单的解释，并分析对比原始的快速排序算法效率的变化。请在提交前做充分的测试以确认程序的正确性。
- 样例输入：5 1 1 2 0 0
- 样例输出：0 0 1 1 2 5
- 可参考 Leetcode 相关题目来测试程序的正确性：
  - 912. 排序数组

快速排序 – 常见问题

- 注意划分后保证 pivot 在中间的位置（左-pivot-右），或保证左右均非空，才能确保问题规模下降
- a, b, c = a', b', c' 这种赋值语句有风险（有两数相同时可能变成 a, a, b = b, a, a，等价于 a=b），慎用
- 固定取第一个和最后一个数做中值，会导致退化为 $O(n^2)$ 的算法，栈的深度为 $O(n)$，导致函数调用栈溢出。（python 的栈默认最多 1000 层）
- “在数组首尾和中间三点取中值作为 pivot”的方法，在进行划分之前必须对这三点先做排序，才能规避某些特定的坏情况
  - 该情形永远 O(n)：100, 1, 2, ..., 99
  - 该情形会导致上一种情况：100, 99, ..., 1

快速排序 – 测试样例

1. (empty list)
2. 0
3. 1 0
4. -1 -2 -3
5. 2 2 1 1 0 0
6. 3 1 4 1 5 9 2 6 5 3 5
7. 2 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5
8. 10000 内随机 1000 个数
9. 100000 内随机 30000 个数，从大到小排列
10. 1000000 内随机 100000 个数

快速排序 – 参考代码

```
def qsort(nums: [int]) -> [int]:
    def recursion(begin, end):
        if end - begin <= 1:
            return
        mid = (begin + end) // 2
        pivot = nums[mid]
        nums[begin], nums[mid] = nums[mid], nums[begin]
        i = begin + 1
        j = end - 1
        while i <= j:
            while i < end and nums[i] < pivot:
                i += 1
            while j >= begin + 1 and nums[j] >= pivot:
                j -= 1
            if i < j:
                nums[i], nums[j] = nums[j], nums[i]
        nums[j], nums[begin] = nums[begin], nums[j]
```

快速排序 – 参考代码

```
        recursion(begin, j)
        recursion(j + 1, end)
    recursion(0, len(nums))
    return nums
```