150. 逆波兰表达式求值

题目描述

给你一个字符串数组 $tokens$，表示一个根据**逆波兰表示法**表示的算术表达式。

请你计算该表达式，返回一个表示表达式值的整数。

注意

1. 有效的算符为 $+$、$-$、$*$ 和 $/$。
2. 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
3. 两个整数之间的除法总是**向零截断**。
4. 表达式中不含除零运算。
5. 输入是一个根据逆波兰表示法表示的算术表达式。
6. 答案及所有中间计算结果可以用 32 位整数表示。

示例

示例 1:

```
输入：tokens = ["2","1","+","3","*"]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9
```

示例 2:

```
输入：tokens = ["4","13","5","/","+"]
输出：6
解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6
```

示例 3:

```
输入：tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
输出：22
解释：
- 该算式转化为常见的中缀算术表达式为：
  $((10 * (6 / ((9 + 3) * -11))) + 17) + 5$
- 逐步计算：
  $((10 * (6 / (12 * -11))) + 17) + 5$
  $((10 * (6 / -132)) + 17) + 5$
  $((10 * 0) + 17) + 5$
  $(0 + 17) + 5$
  $17 + 5 = 22$

---

## 提示

- $1 \leq \text{tokens.length} \leq 10^4$
- $\text{tokens[i]}$ 是一个算符（$+$、$-$、$*$ 或 $/$），或是在范围 $[-200, 200]$ 内的一个整数。

---

## 什么是逆波兰表达式?

逆波兰表达式是一种**后缀表达式**，所谓后缀就是指算符写在操作数的后面。

**特点**:
1. 无需括号即可表示表达式优先级，例如：
   - 中缀表达式：$((1 + 2) * (3 + 4))$
   - 逆波兰表达式：$(1 \, 2 \, + \, 3 \, 4 \, + \, *)$
2. 适合用**栈操作**计算：
   - 遇到数字：入栈；
   - 遇到算符：从栈中弹出两个数字，计算后将结果压回栈中。
```

In [ ]:
```python
# 第一次尝试
from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            if type(token) is int:
                stack.append(token)
            elif token in "+-*/":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    result = stack.pop() - stack.pop()
```

```
                elif token == "*":
                    result = stack.pop() * stack.pop()
                elif token == "/":
                    result = stack.pop() / stack.pop()
                stack.append(result)
            else:
                raise TypeError

        return stack[-1]
```

你的代码有以下几个问题导致逻辑不正确：

1. **数字检测**：
   - 修正为使用 `token.lstrip('-').isdigit()` 检查是否为数字，因为输入的数字是字符串形式。
2. **除法逻辑**：
   - 使用 `int(a / b)` 或 `//` 进行向零截断。
3. **操作数顺序**：
   - 弹出两个操作数时，`b = stack.pop()` 是第二个操作数，`a = stack.pop()` 是第一个操作数。

返回栈顶元素：

使用 stack.pop() 而不是 stack[-1]，保证栈被正确清空。

除法运算：

改为 int(num_2 / num_1)，显式向零截断，避免符号不同导致的错误。

异常处理：

使用更具体的异常类型（ValueError），并提供详细的错误信息。

In [ ]:
```
# 第二次尝试，虽然有些结果对了，但是对于复杂的输入还是错误

from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            # 如果是数字，直接压入栈（判断是否为数字）
            if token.lstrip('-').isdigit():
                stack.append(int(token))
            elif token in "+-*/":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    # 注意这里先弹出的是被减数，在弹出减数
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 - num_1
                elif token == "*":
                    result = stack.pop() * stack.pop()
                elif token == "/":
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 // num_1
                stack.append(result)
            else:
                raise TypeError

        return stack[-1]
```

In [ ]:
```
# 第三次尝试，虽然有些结果对了，但是对于复杂的输入还是错误
# 这回对了

from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            # 如果是数字，直接压入栈（判断是否为数字）
            if token.lstrip('-').isdigit():
                stack.append(int(token))
            elif token in "+-*/":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    # 注意这里先弹出的是被减数，在弹出减数
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 - num_1
                elif token == "*":
                    result = stack.pop() * stack.pop()
                elif token == "/":
```

```
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = int(num_2 / num_1)
                stack.append(result)
            else:
                raise TypeError

        return stack.pop()
```

In [ ]:
```
# 答案，  用了更高级的写法

from operator import add, sub, mul

def div(x, y):
    # 使用整数除法的向零取整方式
    return int(x / y) if x * y > 0 else -(abs(x) // abs(y))

class Solution(object):
    op_map = {'+': add, '-': sub, '*': mul, '/': div}

    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token not in {'+', '-', '*', '/'}:
                stack.append(int(token))
            else:
                op2 = stack.pop()
                op1 = stack.pop()
                stack.append(self.op_map[token](op1, op2))  # 第一个出来的在运算符后面
        return stack.pop()
```

In [ ]:
```
# 实际是三个双指针

class Solution:
    def trim_spaces(self , s):
        # 双指针
        left, right = 0, len(s) - 1
        while left<=right and s[left] == ' ':
            left += 1
        while left<=right and s[right] ==' ':
            right = right-1 # 用改变指针的方法去掉空格?

        tmp = []
        # 处理单词中间的空格
        while left<=right:
            if s[left] != " ":
                tmp.append(s[left])
            elif not tmp or tmp[-1] != ' ': # 这里加入了一个判断，如果 tmp 是空列表，tmp[-1] 会抛出 IndexError，  所以确保不为空
                tmp.append(s[left])
            left += 1
        return tmp

    def reverse_string(self, list, left, right):
        while left<right:
            # 从list的头和尾反转
            list[left], list[right] = list[right], list[left]
            left += 1
            right -= 1
        return None

    def reverse_each_word(self, word):
        left, right = 0, 0
        word_len = len(word)
        while left < word_len:
            while right < word_len and word[right] != ' ':
                right += 1
            self.reverse_string(word, left, right - 1)
            left = right + 1
            right = left  # 重置 `right` 与 `left` 同步


    def reverseWords(self, s: str) -> str:
        letter_list = self.trim_spaces(s)
        self.reverse_string(letter_list, 0, len(letter_list) -1)
        self.reverse_each_word(letter_list)
        return ''.join(letter_list)
```

239. 滑动窗口最大值

题目描述

给定一个整数数组 $nums$ 和一个大小为 k 的滑动窗口，从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例

示例 1：

输入：
$nums = [1,3,-1,$
$-3,5,3,6,7]$, $k = 3$

输出： $[3,3,5,5,6,7]$

解释：

滑动窗口的位置	最大值
$[1\ 3\ -1]\ -3$ $5\ 3\ 6\ 7$	3
$1\ [3\ -1\ -3]$ $5\ 3\ 6\ 7$	3
$1\ 3\ [-1\ -3$ $5]\ 3\ 6\ 7$	5
$1\ 3\ -1\ [-3\ 5$ $3]\ 6\ 7$	5
$1\ 3\ -1\ -3$ $[5\ 3\ 6]\ 7$	6
$1\ 3\ -1\ -3\ 5$ $[3\ 6\ 7]$	7

示例 2：

输入： $nums = [1]$, $k = 1$

输出： $[1]$

In [1]:
```python
# 第一次尝试，问题如下
from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        deq = deque()
        max_list = []
        # pre compute initial deq
        for i in range(len(nums)-k):
            if i < k:
                deq.append(nums[i])
            else:
                max_list.append(max(deq))
                deq.append(nums[i])
                deq.popleft()

        return max_list
```

```
  Cell In[1], line 8
    for i in range(len(nums)):
                              ^
SyntaxError: incomplete input
```

你的代码有以下几个问题：

1. 滑动窗口未完全覆盖整个数组

- **问题**：

  ```
  for i in range(len(nums) - k):
  ```
  - 这个循环的范围是 `len(nums) - k`，导致滑动窗口未完全覆盖数组的末尾部分。
  - 例如，当 `nums = [1,3,-1,-3,5,3,6,7]` 和 `k = 3` 时，实际应该有 6 个滑动窗口，但你的代码只计算了 5 个。
- **解决方法**：改为遍历整个数组：

  ```
  for i in range(len(nums)):
  ```

---

2. 滑动窗口的初始化逻辑有误

- **问题**：

  ```
  if i < k:
      deq.append(nums[i])
  ```
  - 这里初始化了前 `k` 个元素的 `deq`，但并没有在初始化阶段计算第一个窗口的最大值。
- **解决方法**：在遍历的过程中，确保每次滑动窗口向右移动时计算当前窗口的最大值。

---

3. 未正确维护滑动窗口最大值

- **问题**：
  - 每次直接调用 `max(deq)` 来获取当前窗口的最大值，这样的计算复杂度是 $O(k)$，导致整体复杂度为 $O(n \cdot k)$，在 n 和 k 都很大时性能较差。
- **解决方法**： 使用一个**双端队列**来存储当前窗口中元素的索引，并保持队列中索引对应的元素按值从大到小排列：
  - 保证队列头部总是当前窗口的最大值。

In [ ]:
```python
# 尝试2， 对了，分三类情况，但效率不高，超时了
# 这个复杂度是 n*k

from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        deq = deque()
        max_list = []
        # pre compute initial deq
        for i in range(len(nums)):
            if i < k - 1:
                deq.append(nums[i])
            elif i == k-1:
                deq.append(nums[i])
                max_list.append(max(deq))
            else:
                deq.append(nums[i])
                deq.popleft()
                max_list.append(max(deq))


        return max_list
```

单调队列

核心思想是 每次加入值的后，把最大值前面的所有值都弹出

这样popleft时弹出的就一定是最大值

如果加入的新值就是最大值，那么说明左边的元素（当前最大值）需要弹出了

In [ ]:
```python
from collections import deque
from typing import List

class  MyQueue:
    def __init__(self):
        self.queue = deque()

    #每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
    #同时pop之前判断队列当前是否为空。
    def pop(self, value):
        if self.queue and value == self.queue[0]:
            self.queue.popleft()

    #如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。
    #这样就保持了队列里的数值是单调从大到小的了。
    def push(self, value):
        while self.queue and value > self.queue[-1]:
            self.queue.pop()
        self.queue.append(value)

    def front(self):
        return self.queue[0]

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        que = MyQueue()
        max_list = []
        # pre compute initial deq
        for i in range(len(nums)):
            if i < k - 1:
                que.push(nums[i])
            elif i == k-1:
                que.push(nums[i])
                max_list.append(que.front())
            else:
                que.pop(nums[i - k]) #滑动窗口移除最前面元素
                que.push(nums[i]) #滑动窗口前加入最后面的元素
                max_list.append(que.front())

        return max_list
```

347. 前 k 个高频元素

题目描述

给你一个整数数组 $nums$ 和一个整数 k，请你返回其中出现频率前 k 高的元素。你可以按**任意顺序**返回答案。

示例

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

提示

- $1 \leq \text{nums.length} \leq 10^5$
- k 的取值范围是 [1, 数组中不相同的元素的个数]
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。

In [7]:
```python
# 第一次尝试，用 deque 模拟 min heap

from collections import deque
from typing import List
from collections import Counter


# 先统计一遍 元素：count，用counter
# 然后用 min heap 维护 k 个元素， 每次加进一个元素，pop出最小的（堆顶）

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        min_heap = MinHeap(k)

        for key, value in freq.items():
            min_heap.push(value)

        return min_heap.que


class MinHeap:
    def __init__(self, k):
        self.que = deque()
        self.k = k
    def push(self, num):
        if len(self.que) == 0:
            self.que.append(num)
        elif len(self.que) < self.k:
            if num >= self.que[0]:
                self.que.appendleft(num)
            else:
                self.que.append(num)
        # else, evaluate the input num and only keep the highest k values
        else:
            if num >= self.que[0]:
                self.que.appendleft(num)
                self.que.pop()
            elif num > self.que[-1]:
                self.que.pop()
                self.que.append(num)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[7], line 74
     71     print("所有测试用例通过！")
     73 # 执行测试
---> 74 test_solution()

Cell In[7], line 69, in test_solution()
     67     result = solution.topKFrequent(nums, k)
     68     # 验证结果是否符合预期
---> 69     assert sorted(result) == sorted(expected), f"Test case {idx + 1} failed: {result} != {expected}"
     71 print("所有测试用例通过！")

AssertionError: Test case 1 failed: deque([3, 2]) != [1, 2]
```

In [16]:
```python
# 第二次尝试，用 deque 模拟 min heap，  修正最开始应该是保留最小值

# 这个方法在插入时需要遍历一遍queue，效率比较低
```

```python
from collections import deque
from typing import List
from collections import Counter


# 先统计一遍 元素：count，用counter
# 然后用 min heap 维护 k 个元素， 每次加进一个元素，pop出最小的（堆顶）

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        min_heap = MinHeap(k)

        for key, value in freq.items():
            min_heap.push((value,key))

        return [num for _, num in min_heap.que]



class MinHeap:
    def __init__(self, k):
        self.que = deque()  # 使用 deque 存储堆
        self.k = k

    def push(self, freq_elem):
        """freq_elem 是一个元组 (频率, 元素)"""
        freq, num = freq_elem

        # 如果堆未满，直接插入到合适的位置
        if len(self.que) < self.k:
            self._insert(freq_elem)
        # 如果堆已满，只在当前频率大于最小值时插入
        elif freq > self.que[0][0]:
            self.que.popleft()  # 移除堆顶最小值
            self._insert(freq_elem)

    def _insert(self, freq_elem):
        """按升序插入 freq_elem (频率, 元素)"""
        freq, _ = freq_elem

        # 遍历查找插入位置
        for i in range(len(self.que)):
            if freq < self.que[i][0]:  # 比较频率，找到比当前值大的位置插入
                self.que.insert(i, freq_elem)
                return
        # 如果未找到插入位置，直接添加到队尾
        self.que.append(freq_elem)
```

复杂度分析

- **时间复杂度**：
  - 统计频率：$O(n)$，其中 n 是数组长度。
  - 堆操作：每个元素插入堆的时间复杂度为 $O(k)$，最多插入 m 个元素（m 是数组中不同的元素数）。
  - 总复杂度：$O(n + m \cdot k)$。
- **空间复杂度**：
  - 使用 `deque` 存储堆：$O(k)$。
  - 使用 `Counter` 存储频率：$O(m)$。
  - 总空间复杂度：$O(m + k)$。

In [ ]:
```python
# 学习答案，用heapq

#时间复杂度：O(nlogk)
#空间复杂度：O(n)
import heapq
class Solution:

    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        #要统计元素出现频率
        map_ = {} #nums[i]:对应出现的次数
        for i in range(len(nums)):
            map_[nums[i]] = map_.get(nums[i], 0) + 1

        #对频率排序
        #定义一个小顶堆，大小为k
        pri_que = [] #小顶堆

        #用固定大小为k的小顶堆，扫描所有频率的数值
        for key, freq in map_.items():
            heapq.heappush(pri_que, (freq, key))
            if len(pri_que) > k: #如果堆的大小大于了K，则队列弹出，保证堆的大小一直为k
                heapq.heappop(pri_que)

        #找出前K个高频元素，因为小顶堆先弹出的是最小的，所以倒序来输出到数组
        result = [0] * k
        for i in range(k-1, -1, -1):
            result[i] = heapq.heappop(pri_que)[1]
        return result
```

In [22]:
```python
# 第三次尝试，用heapq直接实现最小堆，直接存储 (value,key) pair，并按照 (value,key) 中value排序
# 23ms -> 7ms，效率提升很多

import heapq
from collections import Counter
from typing import List
# heap也可以兼容tuple，但是默认按照 tuple 的 index 0 排序， heapq没有直接支持对元组第二个元素排序的选项
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:

        freq = Counter(nums)
        min_heap = []

        for key, value in freq.items():
            heapq.heappush(min_heap, (value,key)) # 注意这里频率在前
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        return [key for _, key in min_heap]
```

In [24]:
```python
def test_solution():
    solution = Solution()

    # 测试用例列表
    test_cases = [
        # 示例测试用例
        {"nums": [1, 1, 1, 2, 2, 3], "k": 2, "expected": [1, 2]},
        {"nums": [1], "k": 1, "expected": [1]},

        # 边界测试用例
        {"nums": [4, 4, 4, 4], "k": 1, "expected": [4]},
        {"nums": [1, 2, 3, 4, 5], "k": 3, "expected": [3, 4, 5]},  # 修正后

        # 一般测试用例
        {"nums": [1, 2, 2, 3, 3, 3, 4, 4, 4, 4], "k": 2, "expected": [4, 3]},
        {"nums": [5, 5, 5, 1, 1, 2, 2, 3], "k": 3, "expected": [5, 1, 2]},
        {"nums": [8, 8, 8, 8, 7, 7, 7, 6], "k": 2, "expected": [8, 7]},
    ]

    # 遍历测试用例
    for idx, case in enumerate(test_cases):
        nums, k, expected = case["nums"], case["k"], case["expected"]
        # 获取结果
        result = solution.topKFrequent(nums, k)
        # 验证结果是否符合预期
        assert sorted(result) == sorted(expected), f"Test case {idx + 1} failed: {result} != {expected}"

    print("所有测试用例通过！")

# 执行测试
test_solution()
```

所有测试用例通过！

几个错误：

如果尽量保留你原来的代码逻辑，用 `deque` 实现，你需要修复以下问题：

问题 1：`MinHeap` 没有维护正确的堆结构

当前的 `MinHeap` 类只是简单操作 `deque` 的两端，没有维护堆的性质。`deque` 并不是堆，无法直接通过添加和删除来保持堆的最小堆性质。

修复方法：

- 按照最小堆的定义，每次插入一个新元素时，确保 `deque` 中的元素保持升序（最小值在队首）。
- 替换元素时，始终移除最小值（`deque` 的左端）。

问题 2：`MinHeap` 只存储频率，没有关联元素

在 `MinHeap.push()` 方法中，你只存储了频率 `value`，没有记录频率对应的元素 `key`，导致结果返回的是频率，而不是实际的数字。

修复方法：

- 修改 `MinHeap`，在 `deque` 中存储 `(频率, 元素)` 的元组。

In [ ]:
```python

```

Python Count 用法

在 Python 中，`count` 方法用于统计字符串、列表或其他可迭代对象中某个元素出现的次数。其使用场景主要有两种：**字符串**和**列表**。

---

1. `str.count`

用于统计一个子字符串在字符串中出现的次数。

语法

```
str.count(sub[, start[, end]])
```

- `sub`：要统计的子字符串。
- `start`（可选）：指定开始统计的位置（包含）。
- `end`（可选）：指定结束统计的位置（不包含）。

示例

```
# 示例 1：基本用法
s = "hello world"
print(s.count("o"))  # 输出: 2

# 示例 2：指定范围
print(s.count("l", 3, 8))  # 输出: 2 (统计索引 3 到 7 的部分)

# 示例 3：子字符串不存在
print(s.count("x"))  # 输出: 0
```

---

2. `list.count`

用于统计列表中某个元素出现的次数。

语法

```
list.count(element)
```

- `element`：要统计的目标元素。

示例

```
# 示例 1：基本用法
nums = [1, 2, 2, 3, 4, 2]
print(nums.count(2))  # 输出: 3

# 示例 2：元素不存在
print(nums.count(5))  # 输出: 0

# 示例 3：统计字符串元素
words = ["apple", "banana", "apple", "cherry"]
print(words.count("apple"))  # 输出: 2
```

---

3. 其他常见用法场景

(1) 用于统计字符或单词出现的次数

```
text = "Python is great and Python is fun"
print(text.count("Python"))  # 输出: 2
```

(2) 判断元素是否存在

虽然 `count` 可以统计元素是否出现，但 `in` 操作通常更高效。

```
nums = [1, 2, 3, 4]
# 判断是否存在某元素
print(nums.count(5) > 0)  # 输出: False
```

(3) 结合其他方法统计复杂条件

通过 `list.count` 或 `str.count` 与 `filter` 或列表解析结合使用：

```
nums = [1, 2, 2, 3, 4, 2]
# 统计列表中大于 2 的元素数量
greater_than_two_count = sum(1 for x in nums if x > 2)
print(greater_than_two_count)  # 输出: 2
```

注意事项

1. **`count` 的性能**：
   - 对于较大的字符串或列表，`count` 方法的性能较好，但如果需要更复杂的统计条件，可以考虑其他方法。
2. **区分大小写**：
   - 对于字符串，`count` 是区分大小写的：
     ```
     s = "Python python PYTHON"
     print(s.count("Python"))  # 输出: 1
     ```
3. **统计子字符串时的重叠问题**：
   - `str.count` 不会统计重叠的子字符串：

```
s = "aaaa"
print(s.count("aa"))  # 输出: 2
```

In [ ]:

Python 中 counter的用法

用最高效的方式输出一个统计数列中数字出现次数的字典

在 Python 中，统计数列中数字出现次数的最高效方式是使用 **`collections.Counter`**，因为它在底层实现中对性能进行了优化。

推荐方法：使用 `collections.Counter`

代码实现

```
from collections import Counter

# 示例数据
nums = [1, 2, 2, 3, 4, 2, 3, 1, 4, 4]

# 使用 Counter
frequency = Counter(nums)

print(frequency)  # 输出: Counter({2: 3, 4: 3, 1: 2, 3: 2})
```

输出结果

`Counter` 是一个字典的子类，其键是列表中的元素，值是元素出现的次数：

```
Counter({2: 3, 4: 3, 1: 2, 3: 2})
```

你可以像操作普通字典一样访问数据：

```
print(frequency[2])  # 输出: 3
print(dict(frequency))  # 转换为普通字典: {1: 2, 2: 3, 3: 2, 4: 3}
```

heapq 最小堆， 最大堆

`heapq` 是 Python 标准库中的一个模块，用于操作堆数据结构（heap）。堆是一种特殊的完全二叉树，最常见的是**最小堆**，其中每个父节点都小于或等于其子节点。

Python 的 `heapq` 默认实现最小堆，可以在 $O(\log n)$ 时间复杂度内完成插入和删除操作。

基本用法

1. 导入模块

```
import heapq
```

2. 最小堆基本操作

操作	方法	描述
插入元素	`heapq.heappush(heap, x)`	将元素 `x` 插入到堆中。
弹出最小值	`heapq.heappop(heap)`	移除并返回堆中的最小值。
查看最小值但不移除	`heap[0]`	直接访问堆顶元素（最小值）。
弹出并插入新元素（替换）	`heapq.heapreplace(heap, x)`	移除堆中最小值，并插入新元素 `x`。
创建堆	`heapq.heapify(iterable)`	将列表转换为堆，原地操作，时间复杂度 $O(n)$。
获取前 k 个最小值	`heapq.nsmallest(k, heap)`	返回堆中前 k 个最小值。
获取前 k	`heapq.nlargest(k, heap)`	返回堆中前 k 个最大值。

操作	方法	描述
个最 大值		

示例代码

最小堆示例

```python
import heapq

# 创建一个空堆
heap = []

# 插入元素
heapq.heappush(heap, 10)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)

print(heap)  # 输出: [1, 10, 5] （最小值在堆顶，顺序可能未排序）

# 弹出最小值
min_value = heapq.heappop(heap)
print(min_value)  # 输出: 1

# 查看堆顶最小值（不移除）
print(heap[0])  # 输出: 5
```

使用 `heapify` 创建堆

```python
import heapq

# 从列表创建堆
nums = [4, 10, 3, 5, 1]
heapq.heapify(nums)

print(nums)  # 输出: [1, 4, 3, 5, 10] （最小值在堆顶）
```

替换堆顶元素

```python
import heapq

# 创建一个堆
nums = [1, 3, 5, 7, 9]
heapq.heapify(nums)

# 替换堆顶元素
heapq.heapreplace(nums, 2)

print(nums)  # 输出: [2, 3, 5, 7, 9]
```

获取前 k 个最小值

```python
import heapq

nums = [1, 8, 3, 9, 4, 2, 7]
k = 3

# 前 3 个最小值
print(heapq.nsmallest(k, nums))  # 输出: [1, 2, 3]

# 前 3 个最大值
print(heapq.nlargest(k, nums))  # 输出: [9, 8, 7]
```

优点和复杂度

1. **时间复杂度**：
   - 插入：$O(\log n)$
   - 弹出最小值：$O(\log n)$
   - 创建堆：$O(n)$
2. **空间复杂度**：
   - 堆占用的空间为 $O(n)$。

应用场景

1. **实时获取最小值或最大值**：
   - 动态维护数据流中的最小值/最大值。
2. **解决 Top-K 问题**：

- 找到数组中前 k 个最小值/最大值。

3. **优先队列**：
   - 模拟任务调度等需要动态优先级的数据结构。

如果有更多问题，可以进一步讨论！ 😊

In [ ]:

在 Python 中，`heapq` 默认是最小堆。如果需要**最大堆**，可以通过以下两种方式实现：

---

1. 使用取反数实现最大堆

这是 `heapq` 模拟最大堆的常见方法。通过存储元素的相反数，保持最小堆的行为，但逻辑上实现了最大堆。

示例代码
```python
import heapq

nums = [1, 8, 3, 9, 4, 2, 7]

# 创建最大堆（取反数）
max_heap = [-num for num in nums]
heapq.heapify(max_heap)

# 弹出最大值
max_value = -heapq.heappop(max_heap)
print(max_value)  # 输出: 9

# 插入新值
heapq.heappush(max_heap, -10)
print(-max_heap[0])  # 输出: 10 （堆顶最大值）
```

优点
- 简单高效。
- `heapq` 本身无需修改。

---

2. 自定义比较函数（推荐使用 `PriorityQueue`）

如果你希望避免取反数，可以使用 `queue.PriorityQueue` 或 `functools` 提供的 `key` 参数来控制排序行为。`PriorityQueue` 内部也是基于 `heapq` 实现的。

示例代码
```python
import heapq

class MaxHeap:
    def __init__(self):
        self.heap = []

    def push(self, val):
        # 插入元素时取反，保持最大堆性质
        heapq.heappush(self.heap, -val)

    def pop(self):
        # 弹出时还原原始值
        return -heapq.heappop(self.heap)

    def peek(self):
        # 查看堆顶值
        return -self.heap[0]

    def size(self):
        return len(self.heap)
```

使用示例
```python
max_heap = MaxHeap()

# 插入元素
max_heap.push(1)
max_heap.push(8)
max_heap.push(3)

print(max_heap.peek())  # 输出: 8 （最大值）

# 弹出最大值
print(max_heap.pop())  # 输出: 8
print(max_heap.pop())  # 输出: 3
print(max_heap.pop())  # 输出: 1
```

---

3. 使用 `PriorityQueue`

`queue.PriorityQueue` 是线程安全的堆实现，但其性能可能比直接使用 `heapq` 略低。

示例代码
```python
```

```python
from queue import PriorityQueue

pq = PriorityQueue()

# 插入元素（取反数以实现最大堆）
pq.put((-1, "Task 1"))
pq.put((-8, "Task 8"))
pq.put((-3, "Task 3"))

# 弹出最大值（还原值）
while not pq.empty():
    priority, task = pq.get()
    print(-priority, task)
```

输出
```text
8 Task 8
3 Task 3
1 Task 1
```

---

4. 使用 `functools` 自定义排序

使用 `heapq` 操作带权值的元组 (`(优先级, 值)`) 来实现最大堆。

示例代码
```python
import heapq
from functools import cmp_to_key

nums = [1, 8, 3, 9, 4, 2, 7]

# 自定义最大堆
max_heap = [(num, num) for num in nums]  # 元组存储值
heapq.heapify(max_heap)

# 弹出最大值
print(max_heap[0])
```

总结

选择:
- 小规模使用: **推荐第一种**, 通过取反数简单模拟最大堆。
- 线程安全且有复杂优先级队列: **推荐第三种**