

# 高频数据结构——哈希表与堆

主讲人 令狐冲  
课程版本 v7.0

# 数据结构类面试问题的三种考法

数据结构可以认为是一个数据存储集合以及定义在这个集合上的若干操作（功能）  
他有如下的三种考法：

考法1：问某种数据结构的基本原理，并要求实现

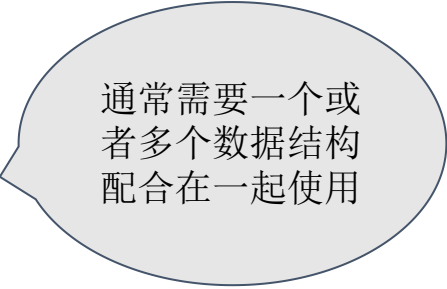
例题：说一下 Hash 的原理并实现一个 Hash 表

考法2：使用某种数据结构完成事情

例题：归并 K 个有序数组

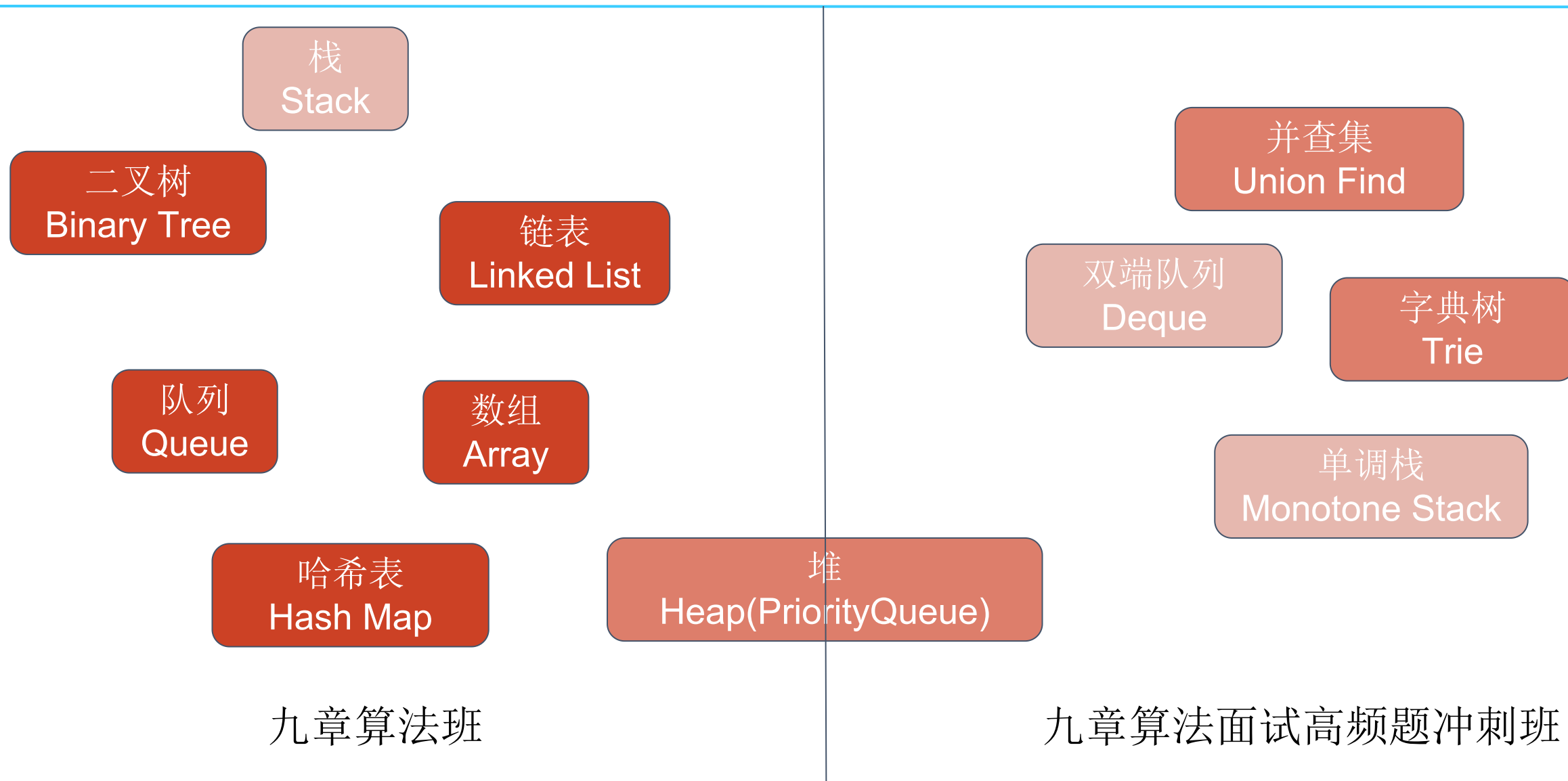
考法3：实现一种数据结构，提供一些特定的功能

例题：最高频 K 项问题



通常需要一个或者多个数据结构配合在一起使用

# 面试中可能会考察的数据结构



# 数据结构时间复杂度的衡量方法

数据结构通常会提供“**多个**”对外接口  
只用一个时间复杂度是很难对其进行正确评价的  
所以通常要对每个接口的时间复杂度进行描述

比如你需要设计一个 **Set** 的数据结构，提供 **lowerBound** 和 **add** 两个方法。**lowerBound** 的意思是，找到比某个数大的最小值

## 算法1: $O(n)$ lowerBound $O(1)$ add

使用数组存储，每次打擂台进行比较，插入就直接插入到数组最后面

## 算法2: $O(\log n)$ lowerBound $O(\log n)$ add

使用红黑树（Red-black Tree）存储，Java 里的 TreeSet，C++ 里的 map

上面两个算法谁好谁坏呢？

不一定谁好谁坏！要看这两个方法被调用的频率如何。

如果 `lowerBound` 很少调用, `add` 非常频繁, 则算法1好。

如果 `lowerBound` 和 `add` 调用的频率都差不多, 或者 `lowerBound` 被调用得更多, 则算法2好

不过通常来说, 在面试中的问题, 我们会很容易找到一个像算法1这样的实现方法, 其中一个操作时间复杂度很大, 另外一个操作时间复杂度很低。


通常的解决办法都是让快的操作慢一点, 让慢的操作快一点, 这样总体得到一个更快的复杂度。

# 哈希表 Hash

支持操作:  $O(1)$  Insert /  $O(1)$  Find /  $O(1)$  Delete

问: 这些操作都是  $O(1)$  的前提条件是什么?

A: 没有哈希冲突

B: key是 $O(1)$ 大小 

# $O(\text{size of key})$

哈希表（HashMap / unordered\_map / dict）

任何操作的时间复杂度从严格意义上来说

都是  $O(\text{size of key})$  而不是  $O(1)$

你不可能在  $O(1)$  的时间内判断 2 个 1m 长的字符串是否相等



请在互动课中学习如下先修知识

---

Hash Table, Hash Map 和 Hash Set 的区别是啥

什么是 Hash Function (产生HashCode的函数), 作用以及实现原理

什么是 Open Hashing 什么是 Closed Hashing

什么是 Rehashing (重哈希)



# LRU Cache

<http://www.lintcode.com/problem/lru-cache/>

<http://www.jiuzhang.com/solutions/lru-cache/>

Example: [2 1 3 2 5 3 6 7]

```
class ListNode {
    public int key, val;
    public ListNode next;

    public ListNode(int key, int val) {
        this.key = key;
        this.val = val;
        this.next = null;
    }
}

private int capacity, size;
private ListNode dummy, tail;
private Map<Integer, ListNode> keyToPrev;

public LRUCache(int capacity) {
    this.capacity = capacity;
    this.keyToPrev = new HashMap<Integer, ListNode>();
    this.dummy = new ListNode(0, 0);
    this.tail = this.dummy;
}

private void moveToTail(int key) {
    ListNode prev = keyToPrev.get(key);
    ListNode curt = prev.next;

    if (tail == curt) {
        return;
    }

    prev.next = prev.next.next;
    tail.next = curt;
    curt.next = null;

    if (prev.next != null) {
        keyToPrev.put(prev.next.key, prev);
    }
    keyToPrev.put(curt.key, tail);

    tail = curt;
}
```

```
public int get(int key) {
    if (!keyToPrev.containsKey(key)) {
        return -1;
    }

    moveToTail(key);

    // the key has been moved to the end
    return tail.val;
}

public void set(int key, int value) {
    // get method will move the key to the end of the linked list
    if (get(key) != -1) {
        ListNode prev = keyToPrev.get(key);
        prev.next.val = value;
        return;
    }

    if (size < capacity) {
        size++;
        ListNode curt = new ListNode(key, value);
        tail.next = curt;
        keyToPrev.put(key, curt);

        tail = curt;
        return;
    }

    // replace the first node with new key, value
    ListNode first = dummy.next;
    keyToPrev.remove(first.key);

    first.key = key;
    first.val = value;
    keyToPrev.put(key, first);

    moveToTail(key);
}
```

```
class ListNode:
    def __init__(self, key=None, val=None):
        self.key = key
        self.val = val
        self.next = None

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.key_to_prev = {}
        self.dummy = ListNode()
        self.tail = self.dummy

    def move_to_tail(self, key):
        prev = self.key_to_prev[key]
        curt = prev.next

        if self.tail == curt:
            return

        prev.next = prev.next.next
        self.tail.next = curt
        curt.next = None

        if prev.next != None:
            self.key_to_prev[prev.next.key] = prev
        self.key_to_prev[curt.key] = self.tail

        self.tail = curt

    def get(self, key):
        if key not in self.key_to_prev:
            return -1

        self.move_to_tail(key)

        # the key has been moved to the end
        return self.tail.val

    def set(self, key, value):
        # get method will move the key to the end of the linked list
        if self.get(key) != -1:
            prev = self.key_to_prev[key]
            prev.next.val = value
            return

        if len(self.key_to_prev) < self.capacity:
            curt = ListNode(key, value)
            self.tail.next = curt
            self.key_to_prev[key] = self.tail

            self.tail = curt
            return

        first = self.dummy.next
        del self.key_to_prev[first.key]

        # replace the first node with new key, value
        first.key = key
        first.val = value
        self.key_to_prev[key] = self.dummy
```

# LRU Cache

- Java 中有一个 LinkedHashMap, 本质上是 DoublyLinkedList + HashMap
  - `HashMap<key, DoublyListNode> DoublyListNode {`
  - `prev, next, key, value;`
  - `}`
  - Python 里是 OrderedDict
- 
- 新节点从尾部加入
  - 老节点从头部移走

问: Singly List 是否可行?

# Singly List 是否可行？

可以，在 Hash 中存储 Singly List 中的 prev node 即可  
如 linked list = dummy->1->2->3->null 时  
hash[1] = dummy, hash[2] = node1 ...

# Insert Delete GetRandom $O(1)$

<http://www.lintcode.com/problem/insert-delete-getrandom-o1/>

<http://www.jiuzhang.com/solutions/insert-delete-getrandom-o1/>

类似的题: <http://www.lintcode.com/problem/load-balancer/>

```
ArrayList<Integer> nums;
HashMap<Integer, Integer> num2index;
Random rand;

public RandomizedSet() {
    // do initialize if necessary
    nums = new ArrayList<Integer>();
    num2index = new HashMap<Integer, Integer>();
    rand = new Random();
}

// Inserts a value to the set
// Returns true if the set did not already contain the specified element or false
public boolean insert(int val) {
    if (num2index.containsKey(val)) {
        return false;
    }

    num2index.put(val, nums.size());
    nums.add(val);
    return true;
}
```

```
// Removes a value from the set
// Return true if the set contained the specified element or false
public boolean remove(int val) {
    if (!num2index.containsKey(val)) {
        return false;
    }

    int index = num2index.get(val);
    // not the last one then swap the last one with this val
    if (index < nums.size() - 1) {
        int last = nums.get(nums.size() - 1);
        nums.set(index, last);
        num2index.put(last, index);
    }
    num2index.remove(val);
    nums.remove(nums.size() - 1);
    return true;
}

// Get a random element from the set
public int getRandom() {
    return nums.get(rand.nextInt(nums.size()));
}
```



```
import random

class RandomizedSet(object):

    def __init__(self):
        # do initialize if necessary
        self.nums, self.val2index = [], {}

    # @param {int} val Inserts a value to the set
    # Returns {bool} true if the set did not already contain the specified element or false
    def insert(self, val):
        if val in self.val2index:
            return False

        self.nums.append(val)
        self.val2index[val] = len(self.nums) - 1
        return True

    # @param {int} val Removes a value from the set
    # Return {bool} true if the set contained the specified element or false
    def remove(self, val):
        # Write your code here
        if val not in self.val2index:
            return False

        index = self.val2index[val]
        last = self.nums[-1]

        # move the last element to index
        self.nums[index] = last
        self.val2index[last] = index

        # remove last element
        self.nums.pop()
        del self.val2index[val]
        return True

    # return {int} a random number from the set
    def getRandom(self):
        return self.nums[random.randint(0, len(self.nums) - 1)]
```

## \* Follow up: 允许重复的数

<http://www.lintcode.com/problem/insert-delete-getrandom-o1-duplicates-allowed/>

<http://www.jiuzhang.com/solutions/insert-delete-getrandom-o1-duplicates-allowed/>

实现较为困难，看懂参考代码即可，不用太纠结

99%的人面试的时候都做不出来，面试时通常给个思路就可以了

## Insert Delete GetRandom O(1) 面试评分标准

---

### **Strong Hire:**

Bug Free 的实现无重复版本的代码，并给出有重复版的基本思路即可

### **Hire / Weak Hire:**

实现无重复版本的代码，无需太多提示或者较少提示，代码 Bug 不多

### **No Hire / Strong No:**

无法无重复版本的给出正确算法，或者无法正确实现，代码 Bug 过多

# 休息 5 分钟

总结一道题的经验，胜过刷十道题  
把你的代码和总结发到九章面试题交流社区  
[www.jiuzhang.com/solutions](http://www.jiuzhang.com/solutions)

# First Unique Number in Data Stream

<http://www.lintcode.com/problem/first-unique-number-in-data-stream/>

<http://www.jiuzhang.com/solutions/first-unique-number-in-data-stream/>

给一个连续的数据流,写一个函数返回终止数字到达时的第一个唯一数字  
(包括终止数字),如果找不到这个终止数字, 返回 -1.

```
public int firstUniqueNumber(int[] nums, int number) {
    Map<Integer, Integer> counter = new HashMap<>();

    boolean is_break = false;
    for (int i = 0; i < nums.length; i++) {
        int num = nums[i];
        if (!counter.containsKey(num)) {
            counter.put(num, 1);
        } else {
            counter.put(num, counter.get(num) + 1);
        }
        if (num == number) {
            is_break = true;
            break;
        }
    }
    if (!is_break) {
        return -1;
    }
}
```

```
for (int i = 0; i < nums.length; i++) {
    int num = nums[i];
    if (counter.get(num) == 1) {
        return num;
    }
    if (num == number) {
        break;
    }
}

return -1;
```

```
def firstUniqueNumber(self, nums, number):  
    counter = {}  
    for num in nums:  
        counter[num] = counter.get(num, 0) + 1  
        if num == number:  
            break  
    else:  
        return -1  
  
    for num in nums:  
        if counter[num] == 1:  
            return num  
        if num == number:  
            break  
  
    return -1
```

# Follow up: 只遍历一次

<http://www.lintcode.com/problem/first-unique-number-in-data-stream-ii/>

<http://www.jiuzhang.com/solutions/first-unique-number-in-data-stream-ii/>

实现一个具有加一个新的数和返回第一个独特的数两个方法的数据结构



```
private ListNode head, tail;
private Map<Integer, ListNode> numToPrev;
private Set<Integer> duplicates;

public DataStream() {
    // dummy
    head = new ListNode(0);
    tail = head;

    numToPrev = new HashMap<>();
    duplicates = new HashSet<>();
}

private void remove(int number) {
    ListNode prev = numToPrev.get(number);
    prev.next = prev.next.next;
    numToPrev.remove(number);

    // change tail and prev of next
    if (prev.next != null) {
        numToPrev.put(prev.next.val, prev);
    } else {
        tail = prev;
    }
}
```

```
public void add(int number) {
    if (duplicates.contains(number)) {
        return;
    }

    if (numToPrev.containsKey(number)) {
        remove(number);
        duplicates.add(number);
    } else {
        ListNode node = new ListNode(number);
        numToPrev.put(number, tail);
        tail.next = node;
        tail = node;
    }
}

public int firstUnique() {
    if (head.next != null) {
        return head.next.val;
    }
    return -1;
}
```

```
def __init__(self):
    self.dummy = ListNode(0)
    self.tail = self.dummy
    self.num_to_prev = {}
    self.duplicates = set()

def add(self, num):
    if num in self.duplicates:
        return

    if num not in self.num_to_prev:
        self.push_back(num)
        return

    # find duplicate, remove it from hash & linked list
    self.duplicates.add(num)
    self.remove(num)
```

```
def remove(self, num):
    prev = self.num_to_prev.get(num)
    del self.num_to_prev[num]
    prev.next = prev.next.next
    if prev.next:
        self.num_to_prev[prev.next.val] = prev
    else:
        # if we removed the tail node, prev will be the new tail
        self.tail = prev

def push_back(self, num):
    # new num add to the tail
    self.tail.next = ListNode(num)
    self.num_to_prev[num] = self.tail
    self.tail = self.tail.next

def firstUnique(self):
    if not self.dummy.next:
        return None
    return self.dummy.next.val
```

# Data Stream 相关问题

<https://www.lintcode.com/problem/?tag=data-stream>

数据流问题 = 数据只能遍历一次

Data Stream 大都和 **Sliding Window** 有关

这类问题我们将在《九章算法面试高频题冲刺班》中深入讲解

# 什么是 Data Stream 类问题？

---

只允许遍历一次！

<http://www.lintcode.com/problem/first-unique-number-in-a-stream-ii/>

<http://www.jiuzhang.com/solutions/first-unique-number-in-a-stream-ii/>

其他 Data Stream 的相关问题：

<http://www.lintcode.com/tag/data-stream/>

## 哈希表的其他练习题

---

- <http://www.lintcode.com/problem/subarray-sum/>
- <http://www.lintcode.com/problem/copy-list-with-random-pointer/>
- <http://www.lintcode.com/problem/anagrams/>
- <http://www.lintcode.com/problem/longest-consecutive-sequence/>

# Heap

支持操作:  $O(\log N)$  Add /  $O(\log N)$  Remove /  $O(\log N)$  Pop /  $O(1)$   
Min or Max

Heap 的基本原理详见互动课

Java: PriorityQueue

C++: priority\_queue

Python: heapq

# heapq / PQ vs Heap

主要区别是什么？

# heapq / PQ vs Heap

主要区别是什么？

heapq / PQ 是 Heap 的实现，Heap 是数据结构，PQ是具体实现  
heapq / PQ 的 remove 操作是  $O(n)$  的





# 构建一个 heap 的时间复杂度？

是  $O(n)$  还是  $O(n\log n)$  ？

# 构建一个 heap 的时间复杂度？

是  $O(n)$  还是  $O(n \log n)$  ?

是  $O(n)$ ，用 Heapify

Python: `heapq.heapify(...)`

# 遍历一个 heap 的时间复杂度？

将 heap 中的元素按大小顺序拿出来  
比如 Java 中可以用 Iterator 来遍历

# 遍历一个 heap 的时间复杂度？

将 heap 中的元素按大小顺序拿出来

比如 Java 中可以用 Iterator 来遍历

$O(n \log n)$

# Ugly Number II

<http://www.lintcode.com/problem/ugly-number-ii/>

<http://www.jiuzhang.com/solutions/ugly-number-ii/>

设计一个算法，找出只含素因子2，3，5 的第 n 小的数。

# 方法1: 三个指针法

用三个指针分别指向没有乘过2,没有乘过3,没有乘过5的第一个数

优点: 时间复杂度低  $O(n)$

缺点: 思维复杂度比较高

```
public int nthUglyNumber(int n) {
    List<Integer> ugllys = new ArrayList<Integer>();
    ugllys.add(1);

    int p2 = 0, p3 = 0, p5 = 0;
    // p2, p3 & p5 share the same queue: ugllys

    for (int i = 1; i < n; i++) {
        int lastNumber = ugllys.get(i - 1);
        while (ugllys.get(p2) * 2 <= lastNumber) {
            p2++;
        }
        while (ugllys.get(p3) * 3 <= lastNumber) {
            p3++;
        }
        while (ugllys.get(p5) * 5 <= lastNumber) {
            p5++;
        }

        ugllys.add(Math.min(
            Math.min(ugllys.get(p2) * 2, ugllys.get(p3) * 3),
            ugllys.get(p5) * 5
        ));
    }

    return ugllys.get(n - 1);
}
```

```
def nthUglyNumber(self, n):
    ugllys = []
    ugllys.append(1)

    p2, p3, p5 = 0, 0, 0
    # p2, p3 & p5 share the same queue: ugllys

    for i in range(1, n):
        last_number = ugllys[i - 1]
        while ugllys[p2] * 2 <= last_number:
            p2 += 1
        while ugllys[p3] * 3 <= last_number:
            p3 += 1
        while ugllys[p5] * 5 <= last_number:
            p5 += 1

        ugllys.append(min(
            ugllys[p2] * 2,
            ugllys[p3] * 3,
            ugllys[p5] * 5
        ))

    return ugllys[n - 1]
```

## 方法2: 堆大法

每次都去找堆里最小的数

优点：思维复杂度低

缺点：时间复杂度略低，但是可以接受  $O(n\log n)$



```
public int nthUglyNumber(int n) {
    Queue<Long> heap = new PriorityQueue<Long>();
    HashSet<Long> visited = new HashSet<Long>();
    heap.add(Long.valueOf(1));
    visited.add(Long.valueOf(1));

    Long[] primes = new Long[3];
    primes[0] = Long.valueOf(2);
    primes[1] = Long.valueOf(3);
    primes[2] = Long.valueOf(5);

    Long val = null;
    for (int i = 0; i < n; i++) {
        val = heap.poll();
        for (int j = 0; j < 3; j++) {
            if (!visited.contains(val * primes[j])) {
                heap.add(val * primes[j]);
                visited.add(val * primes[j]);
            }
        }
    }
    return val.intValue();
}
```

```
import heapq

class Solution:
    def nthUglyNumber(self, n):
        heap = [1]
        visited = set([1])

        val = None
        for i in range(n):
            val = heapq.heappop(heap)
            for factor in [2, 3, 5]:
                if val * factor not in visited:
                    visited.add(val * factor)
                    heapq.heappush(heap, val * factor)

        return val
```

# 在线算法 vs 离线算法

在线算法 = 数据结构设计类问题 = 数据流问题 = 数据不可二次访问 = 多次输入和输出

离线算法 = 一次输入输出 = 数据是一开始给定的 = 数据可以多次访问

# Top K 问题离线算法

<http://www.lintcode.com/problem/k-closest-points/>

<http://www.jiuzhang.com/solutions/k-closest-points/>

Microsoft / Apple / Facebook

# 方法1: Quick Select

- 0. 计算所有的点到原点的 distance --  $O(n)$
  - 1. Quick Select 去找到 kth smallest distance --  $O(n)$
  - 3. 遍历所有 distance 找到 top k smallest distance --  $O(n)$
  - 4. 找到的 top k smallest points 按 distance 排序并返回 --  $O(k \log k)$
- 总体时间复杂度 --  $O(n + k \log k)$

## 方法2: 还是堆大法

基于堆的方法，堆里从远到近排序。

当堆里超过  $k$  个元素的时候，就 pop 掉一个。

总体时间复杂度 --  $O(n\log k)$

# Top K 问题在线算法

<http://www.lintcode.com/problem/top-k-largest-numbers-ii/>

<http://www.jiuzhang.com/solutions/top-k-largest-number-ii/>

Follow up: Top K Frequent Elements

```
Queue<Integer> minheap;
int k;

public Solution(int k) {
    this.k = k;
    minheap = new PriorityQueue<Integer>();
}

public void add(int num) {
    minheap.add(num);
    if (minheap.size() > k) {
        minheap.poll();
    }
}

public List<Integer> topk() {
    Iterator it = minheap.iterator();
    List<Integer> result = new ArrayList<Integer>();
    while (it.hasNext()) {
        result.add((Integer) it.next());
    }
    Collections.sort(result, Collections.reverseOrder());
    return result;
}
```

```
import heapq
class Solution:
    def __init__(self, k):
        self.k = k
        self.heap = []

    def add(self, num):
        heapq.heappush(self.heap, num)
        if len(self.heap) > self.k:
            heapq.heappop(self.heap)

    def topk(self):
        return sorted(self.heap, reverse=True)
```

## Top K Largest Number II 面试评分标准

### **Strong Hire:**

能完整实现代码，时间复杂度最优，代码没有大 Bug，无需提示

能够对 Follow up 问题提出解决方案（不一定要实现）

### **Hire / Weak Hire:**

能够完整实现代码，代码 Bug 不多

无需提示或者很少需要提示做出最优复杂度的版本

### **No Hire / Strong No:**

无法用最优的算法实现出来



- <http://www.lintcode.com/en/problem/high-five/>
- <http://www.lintcode.com/problem/merge-k-sorted-arrays/>
- <http://www.lintcode.com/problem/data-stream-median/>
- <http://www.lintcode.com/problem/top-k-largest-numbers/>
- <http://www.lintcode.com/problem/kth-smallest-number-in-sorted-matrix/>

# 独孤九剑 —— 破掌式

高级数据结构 Cheat Sheet

高级数据结构	各类操作时间复杂度	能够解决哪些问题	考察频率	学习难度	哪里可以学到
Heap 堆	$O(\log n)$ push, pop $O(1)$ top	全局动态找最大找最小	高	低（掌握应用）	九章算法班，强化班
Hash Map 哈希表	$O(1)$ insert, find, delete	查询元素是否存在, <b>key-value</b> 查询问题	高	低（掌握应用）	九章算法班
Trie 前缀树	$O(L)$ insert, find, delete	和哈希表解决问题类似，查询元素是否存在, <b>key-value</b> 查询问题	中	低	九章算法面试高频题冲刺班
UnionFind 并查集	$O(1)$ union, find	动态合并集合并判断两个元素是否在同一个集合, <b>MST</b>	中	低	九章算法面试高频题冲刺班
Balanced BST 平衡排序二叉树（如红黑树 Red-black Tree）	$O(\log n)$ insert, find, delete, max, min, lower, upper	动态增删查改并支持同时找全局最大最小值 找比某个数大的最小值和比某个数小的最大值时可以用（尽可能接近）	低	高	Google
Skip List 跳跃表	$O(\log n)$ insert, find, delete, max, min, lower, upper	和 <b>Balanced BST</b> 解决的问题一样，并能一直维持一个有序链表	低	高	系统设计班
Binary Indexed Tree 树状数组	$O(\log n)$ insert, delete, range sum	增删改的同时，解决区间求和问题	低	中	树状数组与线段树
Segment Tree 线段树	$O(\log n)$ insert, delete, find, range max, range min, range sum, lower, upper $O(1)$ global max, global min	增删改的同时，解决区间求值问题, <b>max/min/sum</b> 等等 可以完全替代	低	中	树状数组与线段树

- 在互动课中继续学习栈和队列的相关面试题
  - 最大栈 / 最小栈
  - 两个队列实现栈 / 两个栈实现队列
- 在互动课中继续学习外排序算法与数组合并类问题
  - 外排序算法与 K 路归并算法
  - 数组合并的相关问题