69. x 的平方根

计算并返回x的平方根，其中x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:
输入: 4
输出: 2

示例 2:
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,由于返回类型是整数，小数部分将被舍去。

Python3，二分查找法，时间复杂度o(logn)。查找区间左闭右开，即[m, n)，所以二分查找之前，右端点的值 + 1，同时，循环的条件使用严格的小于，而不是≤。

使用左闭右开带来的好处是，当查找区间为∅，跳出while循环时，左右端点相同，所以不需要思考使用左端点还是右端点。

如果while里的if判断使用了 > 号，则找到了（上界+1）的数值；如果if判断使用了≥号，则找到了下界的数值。

In [1]:
```
class Solution:
    def mySqrt(self, x: int) -> int:
        l=0
        r=x+1
        while l<r:
            mid=l+(r-l)//2
            if mid*mid>x:
                r=mid
            else:
                l=mid+1
        return l-1
s = Solution()
result1=s.mySqrt(4)
print(result1)
result2 = s.mySqrt(8)
print(result2)
```

```
2
2
```

241. 为运算表达式设计优先级

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 * 。

示例 1:
输入: "2-1-1"
输出: [0, 2]
解释:
((2-1)-1) = 0
(2-(1-1)) = 2
示例 2:
输入: "23-45"
输出: [-34, -14, -10, -10, 10]
解释:
(2(3-(45))) = -34
((23)-(45)) = -14
((2(3-4))5) = -10
(2((3-4)5)) = -10
(((23)-4)5) = 10

相当于是根据所给式子构造运算二叉树的问题，
所以每次遍历其中的每个运算符作为根结点，
该运算符前后的内容分别递归得到左右子树可能的构造，每种左右子树组合得到一种构造结果

In [2]:
```
class Solution:
    def diffWaysToCompute(self, input):
        res = []
        ops = {'+': lambda x, y: x+y, '-': lambda x,
               y: x-y, '*': lambda x, y: x*y}
        for indx in range(1, len(input)-1):
            if input[indx] in ops.keys():

                for left in self.diffWaysToCompute(input[:indx]):
                    for right in self.diffWaysToCompute(input[indx+1:]):
                        res.append(ops[input[indx]](left, right))
        if not res:
            res.append(int(input))

        return res


s = Solution()
result1 = s.diffWaysToCompute('2-1-1')
print(result1)
result2 = s.diffWaysToCompute("2*3-4*5")
print(result2)
```

```
[2, 0]
[-34, -10, -14, -10, 10]
```

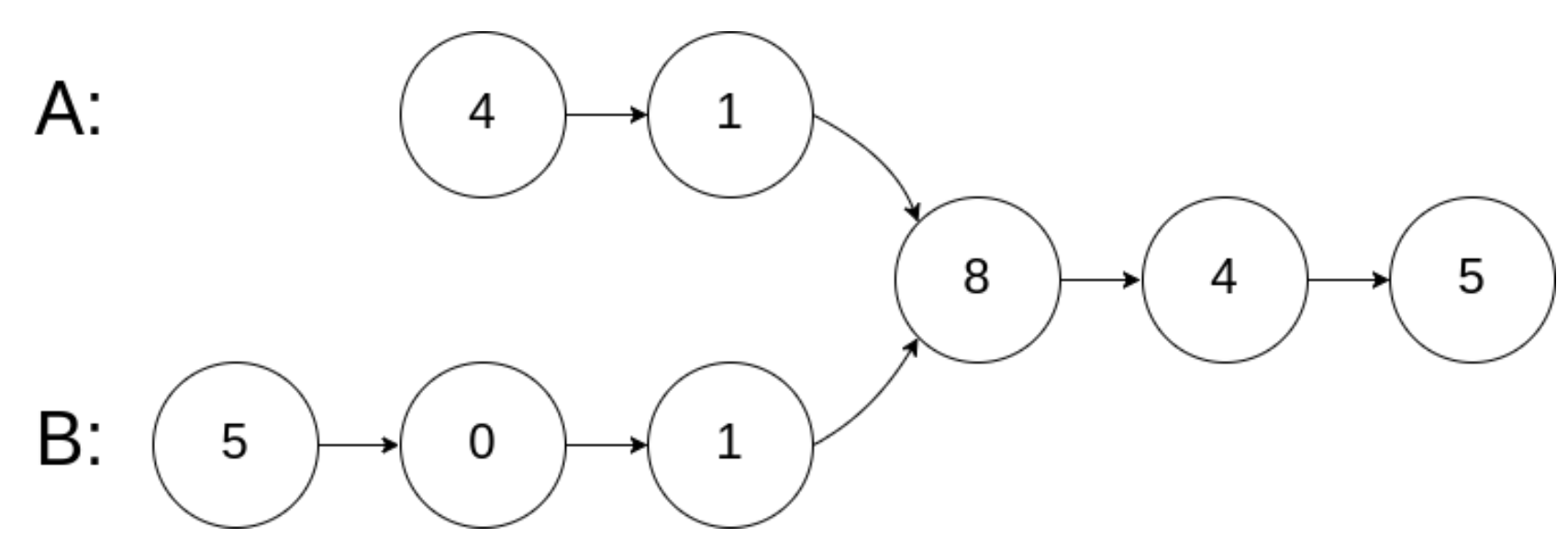
160. 相交链表

编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：

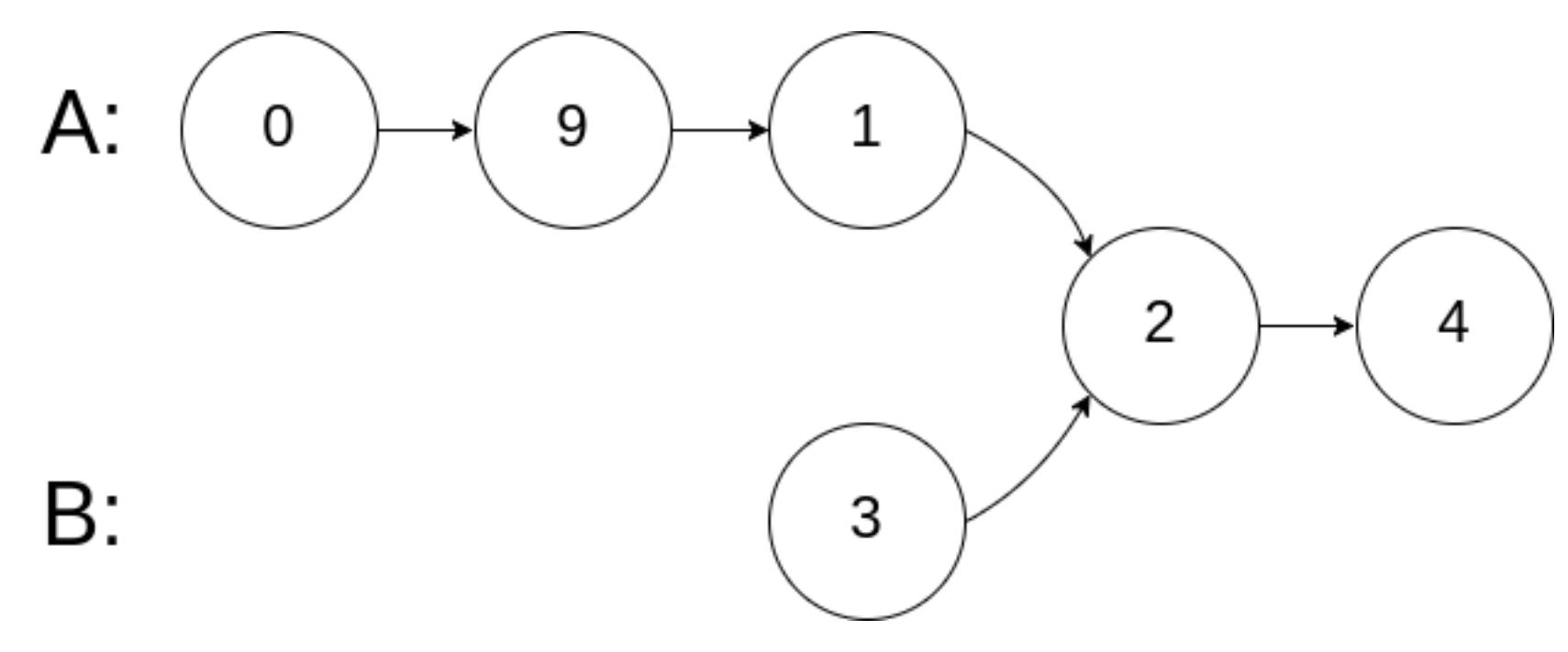
在节点 c1 开始相交。

示例 1：



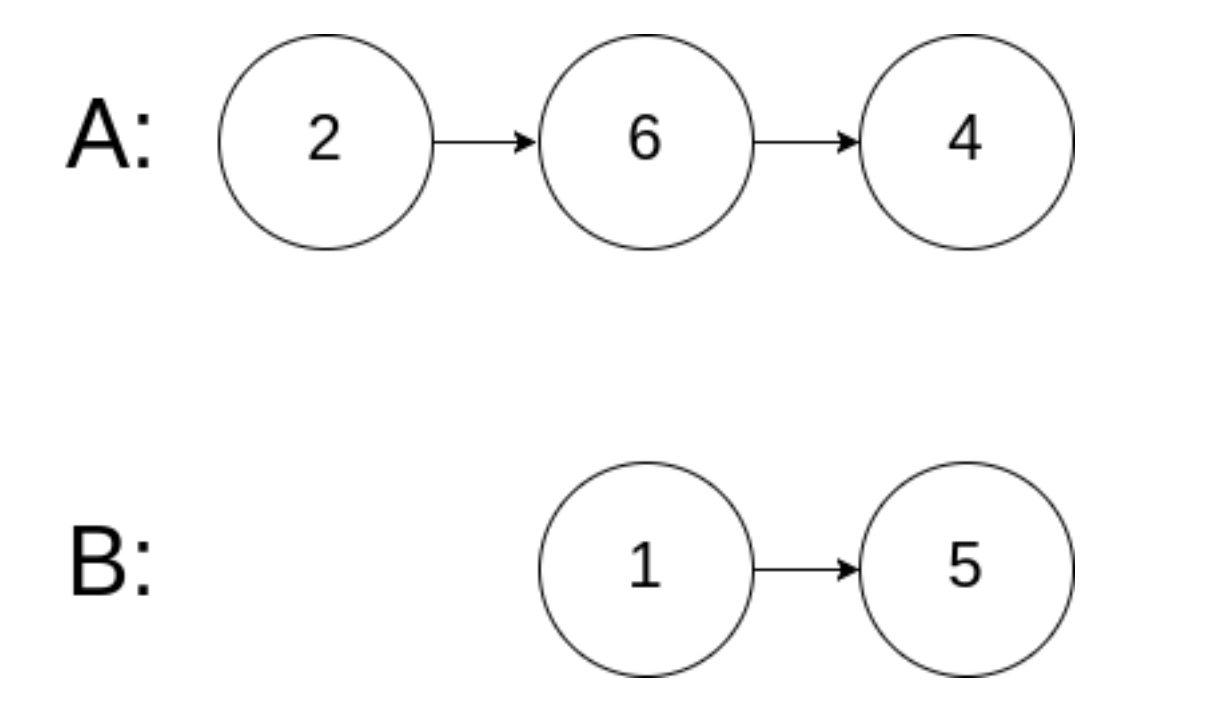
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。 解释：这两个链表不相交，因此返回 null。

注意：

如果两个链表没有交点，返回 null.
在返回结果后，两个链表仍须保持原有的结构。
可假定整个链表结构中没有循环。
程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

思想：

解法一: 第一遍循环，找出两个链表的长度差N
第二遍循环，长链表先走N步，然后同时移动，判断是否有相同节点

解法二:

链表到尾部后,跳到另一个链表的头部, 相遇点即为intersection points.

1. 两数之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例: 给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]

In [3]:
```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        hashmap = {}
        for index, num in enumerate(nums):
            another_num = target - num
            if another_num in hashmap:
                return [hashmap[another_num], index]
            hashmap[num] = index
        return None


s = Solution()
nums = [2, 7, 11, 15]
target = 9
result = s.twoSum(nums, target)
print(target)
```

```
9
```

242. 有效的字母异位词

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1:
输入: s = "anagram", t = "nagaram"
输出: true

示例 2:
输入: s = "rat", t = "car"
输出: false
说明:
你可以假设字符串只包含小写字母。

In [4]:
```
class Solution(object):
    def isAnagram(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        return sorted(s)==sorted(t)

sol = Solution()
s = "anagram"
t = "nagaram"
result1 = sol.isAnagram(s,t)
print(result1)
s = "rat"
t = "car"
result2 = sol.isAnagram(s,t)
print(result2)
```

```
True
False
```

232. 用栈实现队列

使用栈实现队列的下列操作：

push(x) -- 将一个元素放入队列的尾部。
pop() -- 从队列首部移除元素。
peek() -- 返回队列首部的元素。
empty() -- 返回队列是否为空。
示例:

MyQueue queue = new MyQueue();

queue.push(1);
queue.push(2);
queue.peek(); // 返回 1
queue.pop(); // 返回 1
queue.empty(); // 返回 false
说明:

你只能使用标准的栈操作 -- 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）。

In [5]:
```
class MyQueue(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.instack = []
        self.outstack = []

    def push(self, x):
        """
        Push element x to the back of queue.
        :type x: int
        :rtype: None
        """
        self.instack.append(x)

    def pop(self):
        """
        Removes the element from in front of queue and returns that element.
        :rtype: int
        """
        if len(self.outstack) == 0:
            while self.instack:
                self.outstack.append(self.instack.pop())
        return self.outstack.pop()

    def peek(self):
        """
        Get the front element.
        :rtype: int
        """
        if len(self.outstack) == 0:
            while self.instack:
                self.outstack.append(self.instack.pop())
        return self.outstack[-1]
    def empty(self):
        """
        Returns whether the queue is empty.
        :rtype: bool
        """
        return len(self.instack) == 0 and len(self.outstack) == 0


# Your MyQueue object will be instantiated and called as such:
obj = MyQueue()
obj.push(1)
obj.push(2)
param_2 = obj.pop()
param_3 = obj.peek()
param_4 = obj.empty()
print(param_2)
print(param_3)
print(param_4)
```

```
1
2
False
```

In [ ]: