顺序查找

- 算法简介
  - 顺序查找又称为线性查找，是一种最简单的查找方法。适用于线性表的顺序存储结构和链式存储结构。该算法的时间复杂度为O(n)。
- 基本思路
  - 从第一个元素m开始逐个与需要查找的元素x进行比较，当比较到元素值相同(即m=x)时返回元素m的下标，如果比较到最后都没有找到，则返回-1。
- 优缺点
  - 缺点：是当n 很大时，平均查找长度较大，效率低；
  - 优点：是对表中数据元素的存储没有要求。另外，对于线性链表，只能进行顺序查找。
- 算法实现

```
# 最基础的遍历无序列表的查找算法
# 时间复杂度O(n)
def sequential_search(lis, key):
    """顺序查找"""
    length = len(lis)
    for i in range(length):
        if lis[i] == key:
            return i
        else:
            continue
    return False


def linear_search(sequence, target):
    """线性查找"""
    for index, item in enumerate(sequence):
        if item == target:
            return index
    return False
```

二分查找

- 算法简介
  - 二分查找（Binary Search），是一种在有序数组中查找某一特定元素的查找算法。查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则查找过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。这种查找算法每一次比较都使查找范围缩小一半。
- 算法描述
  给予一个包含 n个带值元素的数组A
  1、 令 L为0 ， R为 n-1 ；
  2、 如果L>R，则搜索以失败告终 ；
  3、 令 m（中间值元素)为 ⌊(L+R)/2⌋；
  4、 如果 AmT，令 R为 m - 1 并回到步骤二；
- 复杂度分析
  - 时间复杂度：折半搜索每次把搜索区域减少一半，时间复杂度为 O(logn)
  - 空间复杂度：O(1)
- 算法实现

```
def binary_search(lis, key):
    """非递归形式二分查找"""
    low = 0
    high = len(lis) - 1
    while low < high:
        # 防止溢出
        mid = (high - low)//2 + low
        if key < lis[mid]:
            high = mid - 1
        elif key > lis[mid]:
            low = mid + 1
        else:
            return mid
    return False


def binary_search(nums, key, start=0, end=None):
    """递归实现二分查找"""
    # 初始化时end的值
    end = len(nums) - 1 if end is None else end
    # 递归出口
    if end < start:
        return False
    # 防止溢出
    mid = (end - start) // 2 + start
    if key > nums[mid]:
        return binary_search(nums, key, start=mid + 1, end=end)
    elif key < nums[mid]:
        return binary_search(nums, key, start=start, end=mid - 1)
    elif key == nums[mid]:
        return mid
```

哈希查找

- 算法简介
  - 哈希表就是一种以键-值(key-indexed) 存储数据的结构，只要输入待查找的值即key，即可查找到其对应的值。
- 算法思想
  - 哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。
- 算法流程
  1. 用给定的哈希函数构造哈希表；
  2. 根据选择的冲突处理方法解决地址冲突,常见的解决冲突的方法：拉链法和线性探测法。
  3. 在哈希表的基础上执行哈希查找。
- 复杂度分析
  单纯论查找复杂度：对于无冲突的Hash表而言，查找复杂度为O(1)（注意，在查找之前我们需要构建相应的Hash表）。
- 算法实现

```
class HashTable:
    def __init__(self, size):
        assert size > 0, 'array size must be > 0'
        self.elem = [None for _ in range(size)]  # 使用list数据结构作为哈希表元素保存方法
        self.count = size  # 最大表长
        self.time = 0  # 记录成功插入数据的个数

    def hash(self, key):
        return key % self.count  # 散列函数采用除留余数法

    def insert_hash(self, key):
        """插入关键字到哈希表内"""
        if self.time == self.count:  # 当空间已满，无法继续插入时报错
            raise ValueError('no space')
        address = self.hash(key)  # 求散列地址
        while self.elem[address]:  # 当前位置已经有数据了，发生冲突。
            address = (address + 1) % self.count  # 线性探测下一地址是否可用
        self.elem[address] = key  # 没有冲突则直接保存。
        self.time += 1

    def search_hash(self, key):
        """查找关键字，返回布尔值"""
        star = address = self.hash(key)
        while self.elem[address] != key:
            address = (address + 1) % self.count
            if not self.elem[address] or address == star:  # 说明没找到或者循环到了开始的位置
                return False
        return True


if __name__ == '__main__':
    list_a = [12, 67, 56, 16, 25, -37, 22, 29, -15, -47, 48, 34]
    hash_table = HashTable(12)
    # hash_table = HashTable(10)
    # hash_table = HashTable(-4)
    for i in list_a:
        hash_table.insert_hash(i)

    # print hash_table.elem
    # print len(hash_table.elem)
    # for i in hash_table.elem:
    #     if i:
    #         print((i, hash_table.elem.index(i)))
    # print("\n")

    print(hash_table.search_hash(-15))
    print(hash_table.search_hash(37))
```