七大查找算法（Python）

程序員代碼實驗室 昨天

作者：ls秦
原文：https://blog.csdn.net/qq_38328378/article/details/80936783

查找算法-- 簡介

查找（Searching）就是根據給定的某個值，在查找表中確定一個其關鍵字等於給定值的數據元素。
查找表（Search Table）：由同一類型的數據元素構成的集合
關鍵字（Key）：數據元素中某個數據項的值，又稱為鍵值
主鍵（Primary Key）：可唯一的標識某個數據元素或記錄的關鍵字
查找表按照操作方式可分為：
1.靜態查找表（Static Search Table）：只做查找操作的查找表。它的主要操作是：
①查詢某個“特定的”數據元素是否在表中
②檢索某個“特定的”數據元素和各種屬性

2.動態查找表（Dynamic Search Table）：在查找中同時進行插入或刪除等操作：
①查找時插入數據
②查找時刪除數據

順序查找

算法簡介
順序查找又稱為線性查找，是一種最簡單的查找方法。適用於線性表的順序存儲結構和鍊式存儲結構。該算法的時間複雜度為O(n)。

基本思路
從第一個元素m開始逐個與需要查找的元素x進行比較，當比較到元素值相同(即m=x)時返回元素m的下標，如果比較到最後都沒有找到，則返回-1。
優缺點
缺點：是當n 很大時，平均查找長度較大，效率低；
優點：是對錶中數據元素的存儲沒有要求。另外，對於線性鍊錶，只能進行順序查找。
算法實現

```
# 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):
  length = len(lis)
  for i in range(length):
    if lis[i] == key:
      return i
    else:
      return False

if __name__ == '__main__':
  LIST = [1, 5, 8, 123, 22, 54, 7, 99, 300, 222]
  result = sequential_search(LIST, 123)
  print(result)
```

二分查找

算法簡介

二分查找（Binary Search），是一種在有序數組中查找某一特定元素的查找算法。查找過程從數組的中間元素開始，如果中間元素正好是要查找的元素，則查找過程結束；如果某一特定元素大於或者小於中間元素，則在數組大於或小於中間元素的那一半中查找，而且跟開始一樣從中間元素開始比較。如果在某一步驟數組為空，則代表找不到。

這種查找算法每一次比較都使查找範圍縮小一半。

算法描述

給予一個包含 個帶值元素的數組A

1、 令L為0 ， R為n-1

2、 如果L>R，則搜索以失敗告終

3、 令m (中間值元素)為 |(L+R)/2|

4、 如果AmT，令R為m - 1 並回到步驟二

複雜度分析

時間複雜度：折半搜索每次把搜索區域減少一半，時間複雜度為 O(logn)

空間複雜度：O(1)

算法實現

```
# 针对有序查找表的二分查找算法

def binary_search(lis, key):
  low = 0
  high = len(lis) - 1
  time = 0
  while low < high:
    time += 1
```

```
    mid = int((low + high) / 2)
    if key < lis[mid]:
      high = mid - 1
    elif key > lis[mid]:
      low = mid + 1
    else:
      # 打印折半的次数
      print("times: %s" % time)
      return mid
  print("times: %s" % time)
  return False

if __name__ == '__main__':
  LIST = [1, 5, 7, 8, 22, 54, 99, 123, 200, 222, 444]
  result = binary_search(LIST, 99)
  print(result)
```

插值查找

算法簡介

插值查找是根據要查找的關鍵字key與查找表中最大最小記錄的關鍵字比較後的 查找方法，其核心就在於插值的計算公式 (key-a[low])/(a[high]-a[low])*(high-low)。

時間複雜度o(logn)但對於表長較大而關鍵字分佈比較均勻的查找表來說，效率較高。

算法思想

基於二分查找算法，將查找點的選擇改進為自適應選擇，可以提高查找效率。當然，差值查找也屬於有序查找。

注：對於表長較大，而關鍵字分佈又比較均勻的查找表來說，插值查找算法的平均性能比折半查找要好的多。反之，數組中如果分佈非常不均勻，那麼插值查找未必是很合適的選擇。

複雜度分析

時間複雜性：如果元素均勻分佈，則O（log log n），在最壞的情況下可能需要O（n）。

空間複雜度：O（1）。

算法實現

```
# 插值查找算法

def binary_search(lis, key):
  low = 0
  high = len(lis) - 1
  time = 0
  while low < high:
    time += 1
    # 计算mid值是插值算法的核心代码
    mid = low + int((high - low) * (key - lis[low])/(lis[high] - lis[low]))
```

```
    print("mid=%s, low=%s, high=%s" % (mid, low, high))
    if key < lis[mid]:
      high = mid - 1
    elif key > lis[mid]:
      low = mid + 1
    else:
      # 打印查找的次数
      print("times: %s" % time)
      return mid
  print("times: %s" % time)
  return False

if __name__ == '__main__':
  LIST = [1, 5, 7, 8, 22, 54, 99, 123, 200, 222, 444]
  result = binary_search(LIST, 444)
  print(result)
```

斐波那契查找

算法簡介

斐波那契數列，又稱黃金分割數列，指的是這樣一個數列：1、1、2、3、5、8、13、21、....，在數學上，斐波那契被遞歸方法如下定義：F(1)=1，F(2)=1，F(n)=f(n-1)+F(n-2)（n>=2）。該數列越往後相鄰的兩個數的比值越趨向於黃金比例值（0.618）。

算法描述

斐波那契查找就是在二分查找的基礎上根據斐波那契數列進行分割的。在斐波那契數列找一個等於略大於查找表中元素個數的數F[n]，將原查找表擴展為長度為Fn，完成後進行斐波那契分割，即F[n]個元素分割為前半部分F[n-1]個元素，後半部分F[n-2]個元素，找出要查找的元素在那一部分並遞歸，直到找到。

複雜度分析

最壞情況下，時間複雜度為O(log2n)，且其期望復雜度也為O(log2n)。

算法實現

```
# 斐波那契查找算法
# 时间复杂度O(log(n))

def fibonacci_search(lis, key):
  # 需要一个现成的斐波那契列表。其最大元素的值必须超过查找表中元素个数的数值。
  F = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
     233, 377, 610, 987, 1597, 2584, 4181, 6765,
     10946, 17711, 28657, 46368]
  low = 0
  high = len(lis) - 1

  # 为了使得查找表满足斐波那契特性，在表的最后添加几个同样的值
  # 这个值是原查找表的最后那个元素的值
```

```
  # 添加的个数由F[k]-1-high决定
  k = 0
  while high > F[k]-1:
    k += 1
  print(k)
  i = high
  while F[k]-1 > i:
    lis.append(lis[high])
    i += 1
  print(lis)

  # 算法主逻辑。time用于展示循环的次数。
  time = 0
  while low <= high:
    time += 1
    # 为了防止F列表下标溢出，设置if和else
    if k < 2:
      mid = low
    else:
      mid = low + F[k-1]-1

    print("low=%s, mid=%s, high=%s" % (low, mid, high))
    if key < lis[mid]:
      high = mid - 1
      k -= 1
    elif key > lis[mid]:
      low = mid + 1
      k -= 2
    else:
      if mid <= high:
        # 打印查找的次数
        print("times: %s" % time)
        return mid
      else:
        print("times: %s" % time)
        return high
  print("times: %s" % time)
  return False

if __name__ == '__main__':
  LIST = [1, 5, 7, 8, 22, 54, 99, 123, 200, 222, 444]
  result = fibonacci_search(LIST, 444)
  print(result)
```

樹表查找

1、二叉樹查找算法。

算法簡介

二叉查找樹是先對待查找的數據進行生成樹，確保樹的左分支的值小於右分支的值，然後在就

行和每個節點的父節點比較大小，查找最適合的範圍。這個算法的查找效率很高，但是如果使用這種查找方法要首先創建樹。

算法思想

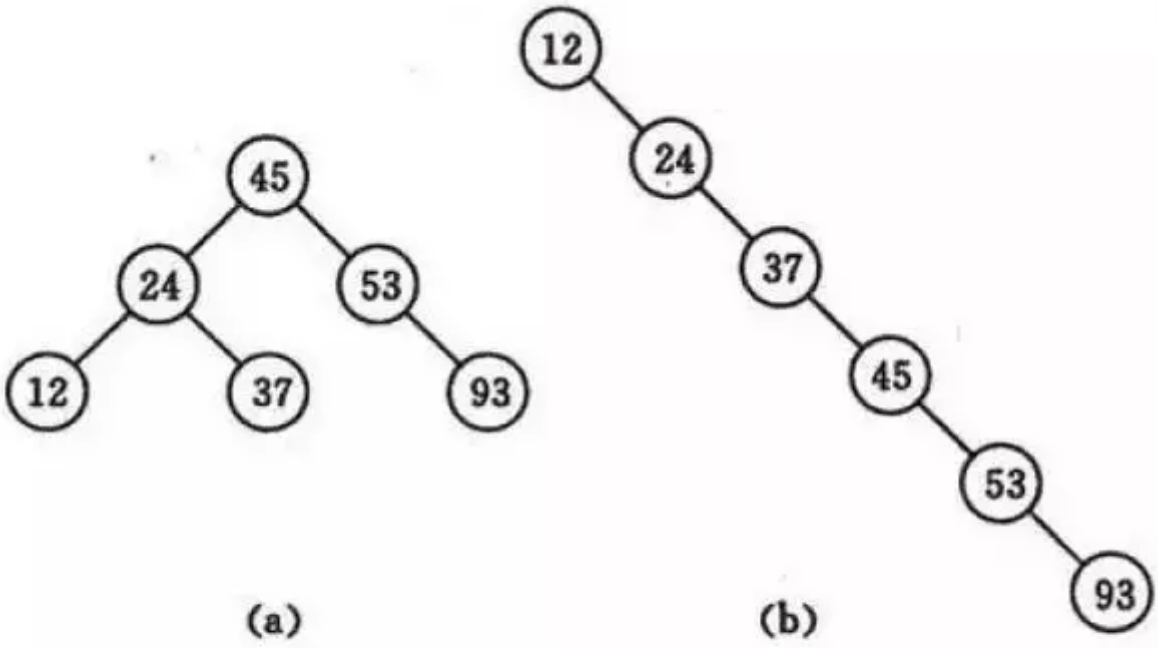
二叉查找樹（BinarySearch Tree）或者是一棵空樹，或者是具有下列性質的二叉樹：

1）若任意節點的左子樹不空，則左子樹上所有結點的值均小於它的根結點的值；

2）若任意節點的右子樹不空，則右子樹上所有結點的值均大於它的根結點的值；

3）任意節點的左、右子樹也分別為二叉查找樹。

二叉查找樹性質：對二叉查找樹進行中序遍歷，即可得到有序的數列。



不同形态的二叉查找树

(a) 关键字序列为(45,24,53,12,37,93)的二叉排序树；

(b) 关键字序列为(12,24,37,45,53,93)的单支树

复雜度分析

它和二分查找一樣，插入和查找的時間複雜度均為O(logn)，但是在最壞的情況下仍然會有O(n)的時間複雜度。原因在於插入和刪除元素的時候，樹沒有保持平衡。

算法實現

```
# 二叉树查找 Python实现
class BSTNode:
    """
    定义一个二叉树节点类。
    以讨论算法为主，忽略了一些诸如对数据类型进行判断的问题。
    """
    def __init__(self, data, left=None, right=None):
```

```
        """
        初始化
        :param data: 节点储存的数据
        :param left: 节点左子树
        :param right: 节点右子树
        """
        self.data = data
        self.left = left
        self.right = right


class BinarySortTree:
    """
    基于BSTNode类的二叉查找树。维护一个根节点的指针。
    """
    def __init__(self):
        self._root = None

    def is_empty(self):
        return self._root is None

    def search(self, key):
        """
        关键码检索
        :param key: 关键码
        :return: 查询节点或None
        """
        bt = self._root
        while bt:
            entry = bt.data
            if key < entry:
                bt = bt.left
            elif key > entry:
                bt = bt.right
            else:
                return entry
        return None

    def insert(self, key):
        """
        插入操作
        :param key:关键码
        :return: 布尔值
        """
        bt = self._root
        if not bt:
            self._root = BSTNode(key)
            return
        while True:
            entry = bt.data
            if key < entry:
                if bt.left is None:
                    bt.left = BSTNode(key)
                    return
```

```
                bt = bt.left
            elif key > entry:
                if bt.right is None:
                    bt.right = BSTNode(key)
                    return
                bt = bt.right
            else:
                bt.data = key
                return

    def delete(self, key):
        """
        二叉查找树最复杂的方法
        :param key: 关键码
        :return: 布尔值
        """
        p, q = None, self._root     # 维持p为q的父节点，用于后面的链接操作
        if not q:
            print("空树！")
            return
        while q and q.data != key:
            p = q
            if key < q.data:
                q = q.left
            else:
                q = q.right
            if not q:               # 当树中没有关键码key时，结束退出。
                return
        # 上面已将找到了要删除的节点，用q引用。而p则是q的父节点或者None（q为根节点时）。
        if not q.left:
            if p is None:
                self._root = q.right
            elif q is p.left:
                p.left = q.right
            else:
                p.right = q.right
            return
        # 查找节点q的左子树的最右节点，将q的右子树链接为该节点的右子树
        # 该方法可能会增大树的深度，效率并不算高。可以设计其它的方法。
        r = q.left
        while r.right:
            r = r.right
        r.right = q.right
        if p is None:
            self._root = q.left
        elif p.left is q:
            p.left = q.left
        else:
            p.right = q.left

    def __iter__(self):
        """
        实现二叉树的中序遍历算法，
```

```
        展示我们创建的二叉查找树.
        直接使用python内置的列表作为一个栈。
        :return: data
        """
        stack = []
        node = self._root
        while node or stack:
            while node:
                stack.append(node)
                node = node.left
            node = stack.pop()
            yield node.data
            node = node.right

if __name__ == '__main__':
    lis = [62, 58, 88, 48, 73, 99, 35, 51, 93, 29, 37, 49, 56, 36, 50]
    bs_tree = BinarySortTree()
    for i in range(len(lis)):
        bs_tree.insert(lis[i])
    # bs_tree.insert(100)
    bs_tree.delete(58)
    for i in bs_tree:
        print(i, end=" ")
    # print("\n", bs_tree.search(4))
```

2、平衡查找樹之2-3查找樹（2-3 Tree）

2-3查找樹定義

和二叉樹不一樣，2-3樹運行每個節點保存1個或者兩個的值。對於普通的2節點(2-node)，他保存1個key和左右兩個自己點。對應3節點(3-node)，保存兩個Key，2-3查找樹的定義如下：

1）要么為空，要么：

2）對於2節點，該節點保存一個key及對應value，以及兩個指向左右節點的節點，左節點也是一個2-3節點，所有的值都比key要小，右節點也是一個2-3節點，所有的值比key要大。

3）對於3節點，該節點保存兩個key及對應value，以及三個指向左中右的節點。左節點也是一個2-3節點，所有的值均比兩個key中的最小的key還要小；中間節點也是一個2-3節點，中間節點的key值在兩個跟節點key值之間；右節點也是一個2-3節點，節點的所有key值比兩個key中的最大的key還要大。

2-3查找樹的性質

1）如果中序遍歷2-3查找樹，就可以得到排好序的序列；

2）在一個完全平衡的2-3查找樹中，根節點到每一個為空節點的距離都相同。（這也是平衡樹中“平衡”一詞的概念，根節點到葉節點的最長距離對應於查找算法的最壞情況，而平

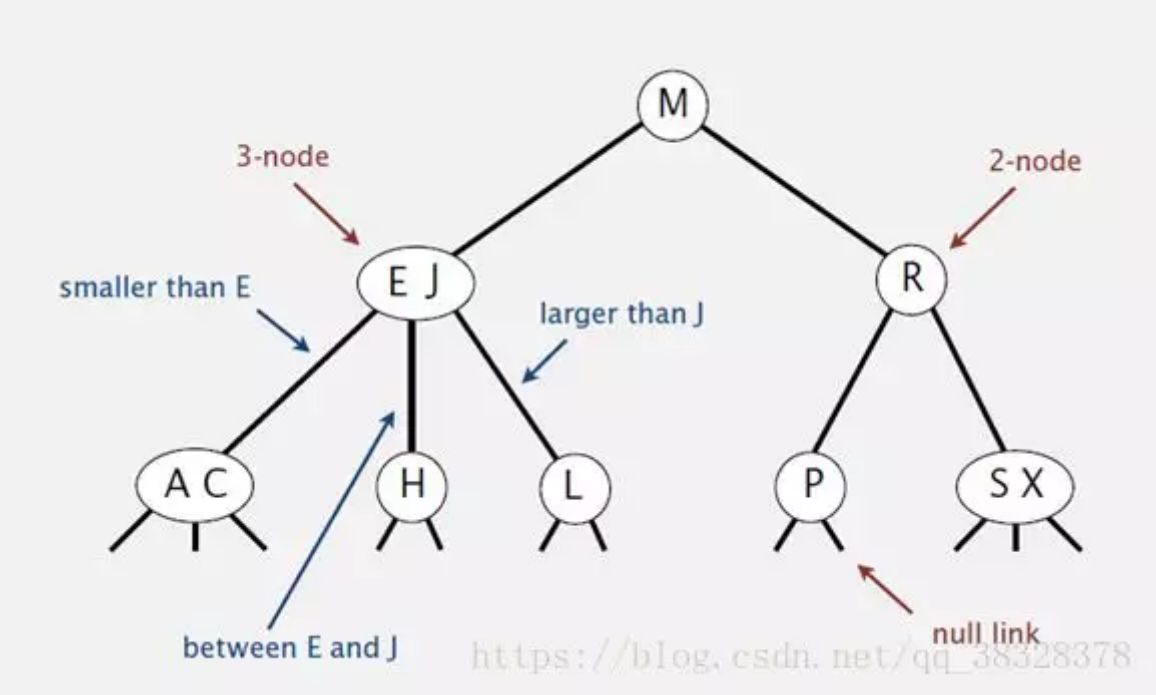
衡樹中根節點到葉節點的距離都一樣，最壞情況也具有對數複雜度。）

複雜度分析：

2-3樹的查找效率與樹的高度是息息相關的。

距離來說，對於1百萬個節點的2-3樹，樹的高度為12-20之間，對於10億個節點的2-3樹，樹的高度為18-30之間。

對於插入來說，只需要常數次操作即可完成，因為他只需要修改與該節點關聯的節點即可，不需要檢查其他節點，所以效率和查找類似。



算法實現

```
class Node(object):
    def __init__(self,key):
        self.key1=key
        self.key2=None
        self.left=None
        self.middle=None
        self.right=None
    def isLeaf(self):
        return self.left is None and self.middle is None and self.right is None
    def isFull(self):
        return self.key2 is not None
    def hasKey(self,key):
        if (self.key1==key) or (self.key2 is not None and self.key2==key):
            return True
        else:
            return False
    def getChild(self,key):
        if key<self.key1:
            return self.left
```

```
        elif self.key2 is None:
            return self.middle
        elif key<self.key2:
            return self.middle
        else:
            return self.right
class 2_3_Tree(object):
    def __init__(self):
        self.root=None
    def get(self,key):
        if self.root is None:
            return None
        else:
            return self._get(self.root,key)
    def _get(self,node,key):
        if node is None:
            return None
        elif node.hasKey(key):
            return node
        else:
            child=node.getChild(key)
            return self._get(child,key)
    def put(self,key):
        if self.root is None:
            self.root=Node(key)
        else:
            pKey,pRef=self._put(self.root,key)
            if pKey is not None:
                newnode=Node(pKey)
                newnode.left=self.root
                newnode.middle=pRef
                self.root=newnode
    def _put(self,node,key):
        if node.hasKey(key):
            return None,None
        elif node.isLeaf():
            return self._addtoNode(node,key,None)
        else:
            child=node.getChild(key)
            pKey,pRef=self._put(child,key)
            if pKey is None:
                return None,None
            else:
                return self._addtoNode(node,pKey,pRef)

    def _addtoNode(self,node,key,pRef):
        if node.isFull():
            return self._splitNode(node,key,pRef)
        else:
            if key<node.key1:
                node.key2=node.key1
                node.key1=key
```

```
                if pRef is not None:
                    node.right=node.middle
                    node.middle=pRef
            else:
                node.key2=key
                if pRef is not None:
                    node.right=Pref
            return None,None
    def _splitNode(self,node,key,pRef):
        newnode=Node(None)
        if key<node.key1:
            pKey=node.key1
            node.key1=key
            newnode.key1=node.key2
            if pRef is not None:
                newnode.left=node.middle
                newnode.middle=node.right
                node.middle=pRef
        elif key<node.key2:
            pKey=key
            newnode.key1=node.key2
            if pRef is not None:
                newnode.left=Pref
                newnode.middle=node.right
        else:
            pKey=node.key2
            newnode.key1=key
            if pRef is not None:
                newnode.left=node.right
                newnode.middle=pRef
        node.key2=None
        return pKey,newnode
```

3、平衡查找樹之紅黑樹（Red-Black Tree）

紅黑樹的定義

紅黑樹是一種具有紅色和黑色鏈接的平衡查找樹，同時滿足：

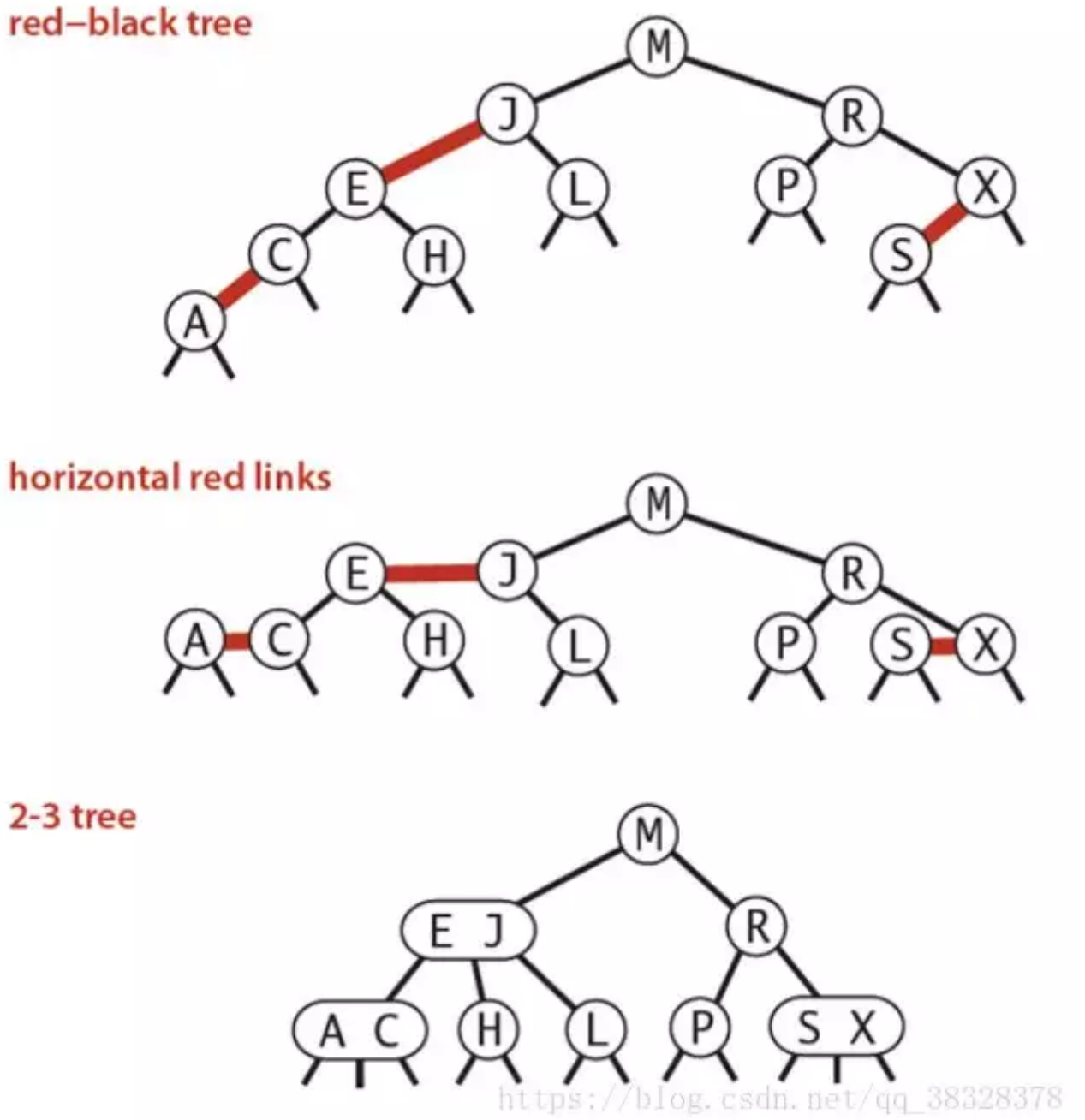
① 紅色節點向左傾斜；

②一個節點不可能有兩個紅色鏈接；

③整個樹完全黑色平衡，即從根節點到所以葉子結點的路徑上，黑色鏈接的個數都相同。

紅黑樹的性質

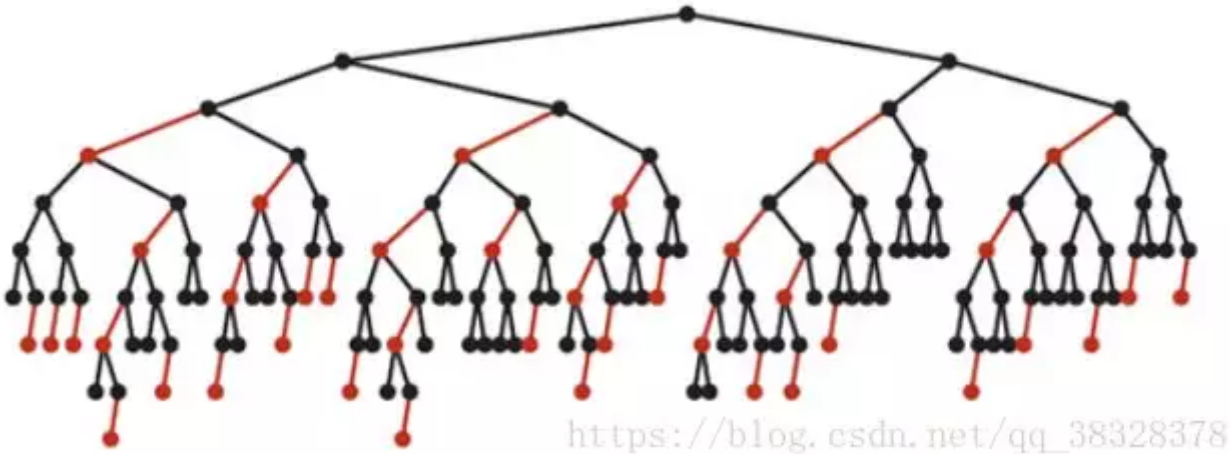
整個樹完全黑色平衡，即從根節點到所以葉子結點的路徑上，黑色鏈接的個數都相同（2-3樹的第2）性質，從根節點到葉子節點的距離都相等）。



複雜度分析

最壞的情況就是，紅黑樹中除了最左側路徑全部是由3-node節點組成，即紅黑相間的路徑長度是全黑路徑長度的2倍。

下圖是一個典型的紅黑樹，從中可以看到最長的路徑(紅黑相間的路徑)是最短路徑的2倍：



算法實現

```
#红黑树
from random import randint

RED = 'red'
BLACK = 'black'

class RBT:
    def __init__(self):
       # self.items = []
        self.root = None
        self.zlist = []

    def LEFT_ROTATE(self, x):
        # x是一个RBTnode
        y = x.right
        if y is None:
            # 右节点为空，不旋转
            return
        else:
            beta = y.left
            x.right = beta
            if beta is not None:
                beta.parent = x

            p = x.parent
            y.parent = p
            if p is None:
                # x原来是root
                self.root = y
            elif x == p.left:
                p.left = y
            else:
                p.right = y
            y.left = x
            x.parent = y

    def RIGHT_ROTATE(self, y):
```

```
        # y是一个节点
        x = y.left
        if x is None:
            # 右节点为空，不旋转
            return
        else:
            beta = x.right
            y.left = beta
            if beta is not None:
                beta.parent = y

            p = y.parent
            x.parent = p
            if p is None:
                # y原来是root
                self.root = x
            elif y == p.left:
                p.left = x
            else:
                p.right = x
            x.right = y
            y.parent = x

    def INSERT(self, val):

        z = RBTnode(val)
        y = None
        x = self.root
        while x is not None:
            y = x
            if z.val < x.val:
                x = x.left
            else:
                x = x.right

        z.PAINT(RED)
        z.parent = y

        if y is None:
            # 插入z之前为空的RBT
            self.root = z
            self.INSERT_FIXUP(z)
            return

        if z.val < y.val:
            y.left = z
        else:
            y.right = z

        if y.color == RED:
            # z的父节点y为红色，需要fixup。
            # 如果z的父节点y为黑色，则不用调整
            self.INSERT_FIXUP(z)

        else:
```

```
            return

    def INSERT_FIXUP(self, z):
        # case 1:z为root节点
        if z.parent is None:
            z.PAINT(BLACK)
            self.root = z
            return

        # case 2:z的父节点为黑色
        if z.parent.color == BLACK:
            # 包括了z处于第二层的情况
            # 这里感觉不必要啊。。似乎z.parent为黑色则不会进入fixup阶段
            return

        # 下面的几种情况，都是z.parent.color == RED:
        # 节点y为z的uncle
        p = z.parent
        g = p.parent  # g为x的grandpa
        if g is None:
            return
            #   return 这里不能return的。。。
        if g.right == p:
            y = g.left
        else:
            y = g.right

        # case 3-0:z没有叔叔。即：y为NIL节点
        # 注意，此时z的父节点一定是RED
        if y == None:
            if z == p.right and p == p.parent.left:
                # 3-0-0:z为右儿子,且p为左儿子，则把p左旋
                # 转化为3-0-1或3-0-2的情况
                self.LEFT_ROTATE(p)
                p, z = z, p
                g = p.parent
            elif z == p.left and p == p.parent.right:
                self.RIGHT_ROTATE(p)
                p, z = z, p

            g.PAINT(RED)
            p.PAINT(BLACK)
            if p == g.left:
                # 3-0-1:p为g的左儿子
                self.RIGHT_ROTATE(g)
            else:
                # 3-0-2:p为g的右儿子
                self.LEFT_ROTATE(g)

            return

        # case 3-1:z有黑叔
        elif y.color == BLACK:
            if p.right == z and p.parent.left == p:
                # 3-1-0:z为右儿子,且p为左儿子,则左旋p
```

```
                # 转化为3-1-1或3-1-2
                self.LEFT_ROTATE(p)
                p, z = z, p
            elif p.left == z and p.parent.right == p:
                self.RIGHT_ROTATE(p)
                p, z = z, p

            p = z.parent
            g = p.parent

            p.PAINT(BLACK)
            g.PAINT(RED)
            if p == g.left:
                # 3-1-1:p为g的左儿子，则右旋g
                self.RIGHT_ROTATE(g)
            else:
                # 3-1-2:p为g的右儿子，则左旋g
                self.LEFT_ROTATE(g)

            return


        # case 3-2:z有红叔
        # 则涂黑父和叔，涂红爷，g作为新的z，递归调用
        else:
            y.PAINT(BLACK)
            p.PAINT(BLACK)
            g.PAINT(RED)
            new_z = g
            self.INSERT_FIXUP(new_z)

    def DELETE(self, val):
        curNode = self.root
        while curNode is not None:
            if val < curNode.val:
                curNode = curNode.left
            elif val > curNode.val:
                curNode = curNode.right
            else:
                # 找到了值为val的元素,正式开始删除

                if curNode.left is None and curNode.right is None:
                    # case1:curNode为叶子节点：直接删除即可
                    if curNode == self.root:
                        self.root = None
                    else:
                        p = curNode.parent
                        if curNode == p.left:
                            p.left = None
                        else:
                            p.right = None

                elif curNode.left is not None and curNode.right is not None:
                    sucNode = self.SUCCESOR(curNode)
                    curNode.val, sucNode.val  = sucNode.val, curNode.val
```

```
                    self.DELETE(sucNode.val)

                else:
                    p = curNode.parent
                    if curNode.left is None:
                        x = curNode.right
                    else:
                        x = curNode.left
                    if curNode == p.left:
                        p.left = x
                    else:
                        p.right = x
                    x.parent = p
                    if curNode.color == BLACK:
                        self.DELETE_FIXUP(x)

                curNode = None
        return False

    def DELETE_FIXUP(self, x):
        p = x.parent
        # w:x的兄弟结点
        if x == p.left:
            w = x.right
        else:
            w = x.left

        # case1:x的兄弟w是红色的
        if w.color == RED:
            p.PAINT(RED)
            w.PAINT(BLACK)
            if w == p.right:
                self.LEFT_ROTATE(p)
            else:
                self.RIGHT_ROTATE(p)

        if w.color == BLACK:
            # case2:x的兄弟w是黑色的，而且w的两个孩子都是黑色的
            if w.left.color == BLACK and w.right.color == BLACK:
                w.PAINT(RED)
                if p.color == BLACK:
                    return
                else:
                    p.color = BLACK
                    self.DELETE_FIXUP(p)

            # case3:x的兄弟w是黑色的，而且w的左儿子是红色的，右儿子是黑色的
            if w.left.color == RED and w.color == BLACK:
                w.left.PAINT(BLACK)
                w.PAINT(RED)
                self.RIGHT_ROTATE(w)

            # case4:x的兄弟w是黑色的，而且w的右儿子是红
            if w.right.color == RED:
                p.PAINT(BLACK)
```

```
                w.PAINT(RED)
                if w == p.right:
                    self.LEFT_ROTATE(p)
                else:
                    self.RIGHT_ROTATE(p)

    def SHOW(self):
        self.DISPLAY1(self.root)
        return self.zlist

    def DISPLAY1(self, node):
        if node is None:
            return
        self.DISPLAY1(node.left)
        self.zlist.append(node.val)
        self.DISPLAY1(node.right)

    def DISPLAY2(self, node):
        if node is None:
            return
        self.DISPLAY2(node.left)
        print(node.val)
        self.DISPLAY2(node.right)

    def DISPLAY3(self, node):
        if node is None:
            return
        self.DISPLAY3(node.left)
        self.DISPLAY3(node.right)
        print(node.val)

class RBTnode:
    '''红黑树的节点类型'''
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None

    def PAINT(self, color):
        self.color = color

def zuoxuan(b, c):
    a = b.parent
    a.left = c
    c.parent = a
    b.parent = c
    c.left = b

if __name__ == '__main__':
    rbt=RBT()
    b = []

    for i in range(100):
        m = randint(0, 500)
```

```
        rbt.INSERT(m)
        b.append(m)

    a = rbt.SHOW()
    b.sort()
    equal = True
    for i in range(100):
        if a[i] != b[i]:
            equal = False
            break

    if not equal:
        print('wrong')
    else:
        print('OK!')
```

4、B樹和B+樹（B Tree/B+ Tree）

B樹簡介

B 樹可以看作是對2-3查找樹的一種擴展，即他允許每個節點有M-1個子節點。

①根節點至少有兩個子節點；

②每個節點有M-1個key，並且以升序排列；

③位於M-1和M key的子節點的值位於M-1 和M key對應的Value之間；

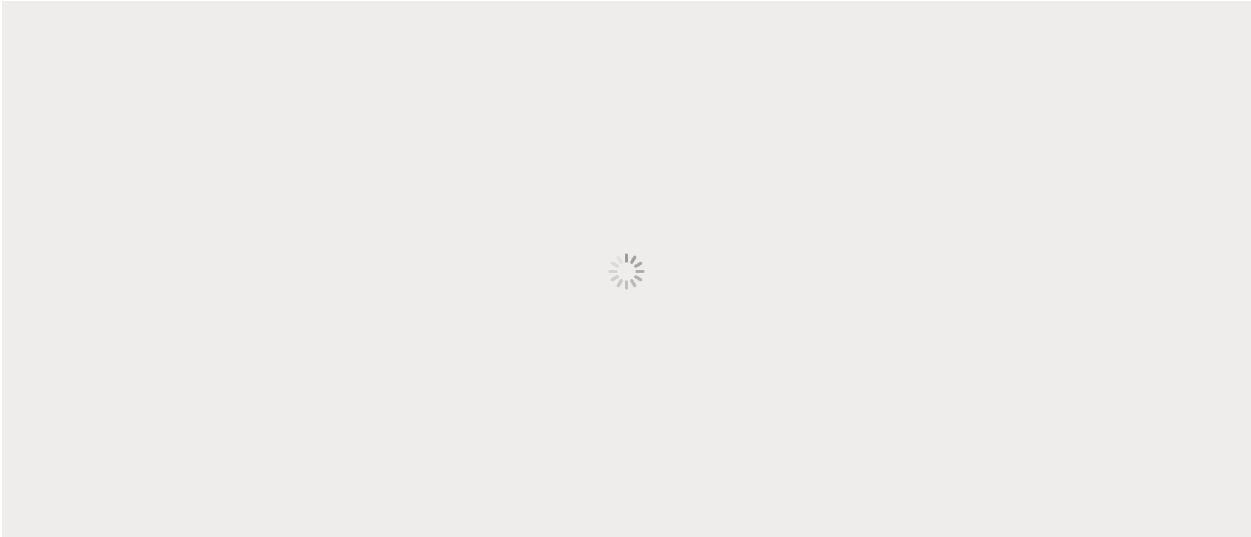
④非葉子結點的關鍵字個數=指向兒子的指針個數-1；

⑤非葉子結點的關鍵字：K[1], K[2], …, K[M-1]；且K[i] ；

⑥其它節點至少有M/2個子節點；

⑦所有葉子結點位於同一層；

如：（M=3）



B樹算法思想

B-樹的搜索，從根結點開始，對結點內的關鍵字（有序）序列進行二分查找，如果命中則結束，否則進入查詢關鍵字所屬範圍的兒子結點；重複，直到所對應的兒子指針為空，或已

經是葉子結點；

B樹的特性

1.關鍵字集合分佈在整顆樹中；

2.任何一個關鍵字出現且只出現在一個結點中；

3.搜索有可能在非葉子結點結束；

4.其搜索性能等價於在關鍵字全集內做一次二分查找；

5.自動層次控制；

由於限制了除根結點以外的非葉子結點，至少含有M/2個兒子，確保了結點的至少利用率，其最底搜索性能為O(LogN)

B+ 樹簡介

B+樹是B-樹的變體，也是一種多路搜索樹：

1.其定義基本與B-樹同，除了：

2.非葉子結點的子樹指針與關鍵字個數相同；

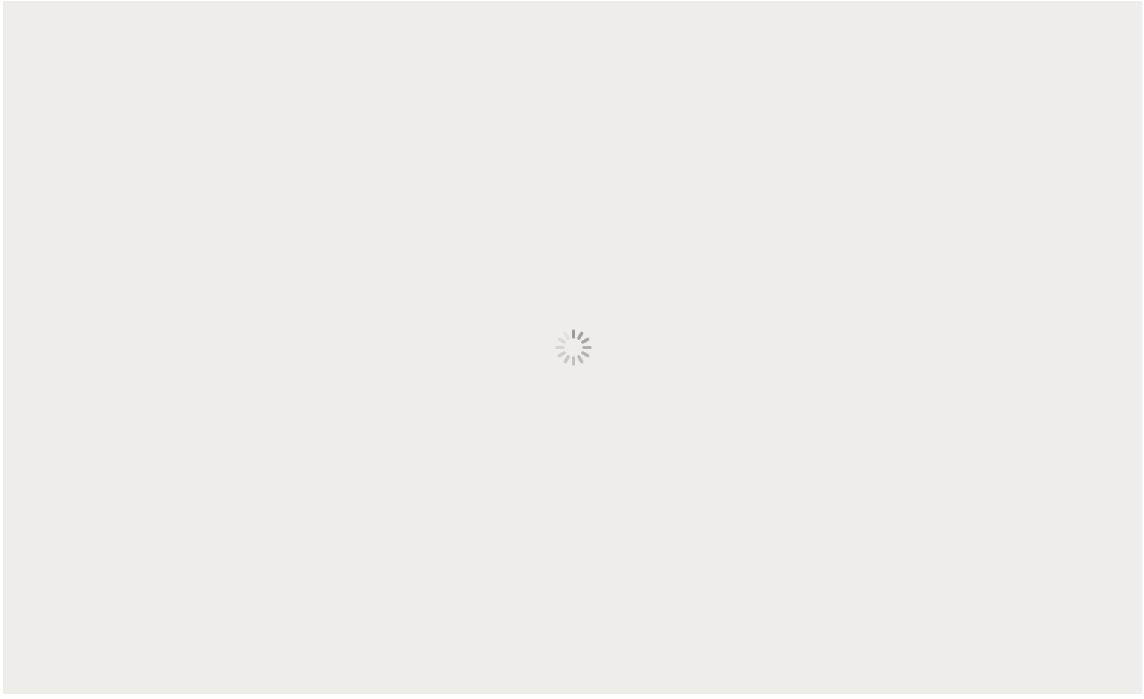
3.非葉子結點的子樹指針P[i]，指向關鍵字值屬於[K[i], K[i+1])的子樹

4.B-樹是開區間；

5.為所有葉子結點增加一個鏈指針；

6.所有關鍵字都在葉子結點出現；

如：（M=3）



B+樹算法思想

B+的搜索與B-樹也基本相同，區別是B+樹只有達到葉子結點才命中（B-樹可以在非葉子結點命中），其性能也等價於在關鍵字全集做一次二分查找；

B+樹的特性

1.所有關鍵字都出現在葉子結點的鏈錶中（稠密索引），且鏈錶中的關鍵字恰好是有序的；

2.不可能在非葉子結點命中；

3.非葉子結點相當於是葉子結點的索引（稀疏索引），葉子結點相當於是存儲（關鍵字）數據的數據層；

4.更適合文件索引系統；

算法實現

```
# -*- coding: UTF-8 -*-
# B树查找

class BTree:  #B树
    def __init__(self,value):
        self.left=None
        self.data=value
        self.right=None

    def insertLeft(self,value):
        self.left=BTree(value)
        return self.left

    def insertRight(self,value):
        self.right=BTree(value)
        return self.right

    def show(self):
        print(self.data)

def inorder(node):  #中序遍历：先左子树，再根节点，再右子树
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)

def rinorder(node):  #倒中序遍历
    if node.data:
        if node.right:
            rinorder(node.right)
        node.show()
        if node.left:
            rinorder(node.left)

def insert(node,value):
```

```
    if value > node.data:
        if node.right:
            insert(node.right,value)
        else:
            node.insertRight(value)
    else:
        if node.left:
            insert(node.left,value)
        else:
            node.insertLeft(value)

if __name__ == "__main__":
    l=[88,11,2,33,22,4,55,33,221,34]
    Root=BTree(l[0])
    node=Root
    for i in range(1,len(l)):
        insert(Root,l[i])
    print("中序遍历（从小到大排序 ）")
    inorder(Root)
    print("倒中序遍历（从大到小排序）")
    rinorder(Root)
```

5、樹表查找總結

二叉查找樹平均查找性能不錯，為O(logn)，但是最壞情況會退化為O(n)。在二叉查找樹的基礎上進行優化，我們可以使用平衡查找樹。平衡查找樹中的2-3查找樹，這種數據結構在插入之後能夠進行自平衡操作，從而保證了樹的高度在一定的範圍內進而能夠保證最壞情況下的時間複雜度。但是2-3查找樹實現起來比較困難，紅黑樹是2-3樹的一種簡單高效的實現，他巧妙地使用顏色標記來替代2-3樹中比較難處理的3-node節點問題。紅黑樹是一種比較高效的平衡查找樹，應用非常廣泛，很多編程語言的內部實現都或多或少的採用了紅黑樹。

除此之外，2-3查找樹的另一個擴展——B/B+平衡樹，在文件系統和數據庫系統中有著廣泛的應用。

分塊查找

算法簡介

要求是順序表，分塊查找又稱索引順序查找，它是順序查找的一種改進方法。

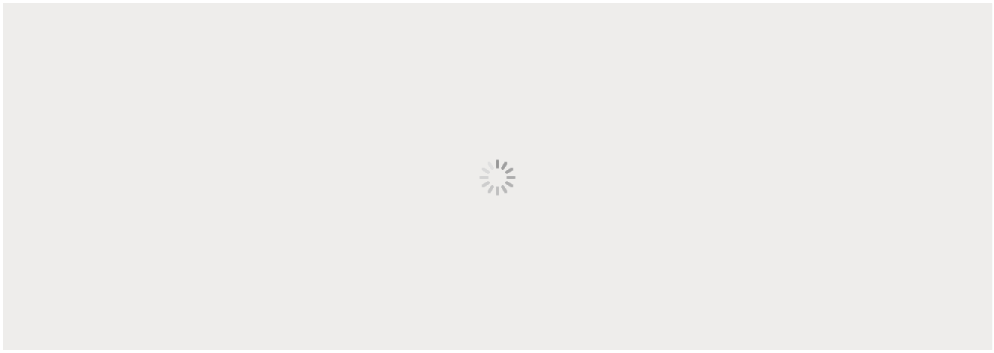
算法思想

將n個數據元素"按塊有序"劃分為m塊（m ≤ n）。

每一塊中的結點不必有序，但塊與塊之間必須"按塊有序"；

即第1塊中任一元素的關鍵字都必須小於第2塊中任一元素的關鍵字；

而第2塊中任一元素又都必須小於第3塊中的任一元素，……



算法流程

1、先選取各塊中的最大關鍵字構成一個索引表；

2、查找分兩個部分：先對索引表進行二分查找或順序查找，以確定待查記錄在哪一塊中；

3、在已確定的塊中用順序法進行查找。

複雜度分析

時間複雜度：O(log(m)+N/m)

哈希查找

算法簡介

哈希表就是一種以鍵-值(key-indexed) 存儲數據的結構，只要輸入待查找的值即key，即可查找到其對應的值。

算法思想

哈希的思路很簡單，如果所有的鍵都是整數，那麼就可以使用一個簡單的無序數組來實現：將鍵作為索引，值即為其對應的值，這樣就可以快速訪問任意鍵的值。這是對於簡單的鍵的情況，我們將其擴展到可以處理更加複雜的類型的鍵。

算法流程

1）用給定的哈希函數構造哈希表；

2）根據選擇的衝突處理方法解決地址衝突；

常見的解決衝突的方法：拉鍊法和線性探測法。

3）在哈希表的基礎上執行哈希查找。

複雜度分析

單純論查找複雜度：對於無衝突的Hash表而言，查找複雜度為O(1)（注意，在查找之前我們需要構建相應的Hash表）。

算法實現

```
# 忽略了对数据类型，元素溢出等问题的判断。

class HashTable:
  def __init__(self, size):
    self.elem = [None for i 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

  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"""
    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。

  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)
  for i in list_a:
    hash_table.insert_hash(i)

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

  print(hash_table.search_hash(15))
  print(hash_table.search_hash(33))
```

閱讀原文

喜歡此內容的人還喜歡

AVL樹/紅黑樹/B樹/B+樹概念掃盲大法-數據結構與算法小結3（升級版）

笨鳥慢飛

滴滴~每日算法速遞【第五期】

Python新視野