基本数据结构

1 栈

- 后进先出 (LIFO, last-in first-out)

1.1 栈操作与实现

- 栈操作
  - `Stack()` 创建一个空栈。
  - `push(item)` 将 `item` 添加到栈的顶端。
  - `pop()` 将栈顶端的元素移除，并返回该元素。
  - `peek()` 返回栈顶端的元素，但是并不移除该元素。
  - `isEmpty()` 检查栈是否为空。
  - `size()` 返回栈中元素的数目。

- 实现

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

1.2 栈的应用

1.2.1 括号匹配，进制转换

1.2.2 前序、中序和后序表达式

- 前序表达式和后序表达式都不需要括号，中序表达式是最不理想的算式表达式!

- 中序表达式转换为后序表达式
  - 算法：

    1. 创建用于保存运算符的空栈 `opstack`，以及一个用于保存结果的空列表。

2. 使用字符串方法`split`将输入的中序表达式转换成一个列表。

3. 从左往右扫描这个标记列表，

    - 如果标记是*操作数*，将其添加到结果列表的末尾；
    - 如果标记是*左括号*，将其压入`opstack`栈中；
    - 如果标记是*右括号*，反复从`opstack`栈中移除元素，直到移除对应的左括号。将从栈中取出的每一个运算符都添加到结果列表的末尾；
    - 如果标记是*运算符*，将其压入`opstack`栈中。**但是，在这之前，需要先从栈中取出优先级更高或相同的运算符，并将它们添加到结果列表的末尾。**

4. 当处理完输入表达式以后，检查`opstack`。将其中所有残留的运算符全部添加到结果列表的末尾。

idea: 括号内可看作独立部分（通过 3-2 & 3-3 独立解决），于是只需考虑最基本的两种情况：

- `A + B * C` → `A B C * +`
- `A * B + C` → `A B * C +`

可以发现，对于第二种情况，若未考虑 3-4 **加粗部分**，由于栈的 LIFO 性质，`+` 将在 `*` 之前，违背了优先顺序。事实上，处理某个运算符时，可将栈内的优先级更高的运算符视为在括号内，于是同 3-3 操作，将其取出并添加到结果列表末尾。

- 实现

```
import Stack, string
def infixToPostfix(infixexpr):
    prec = {"*": 3, "/": 3, "+": 2, "-": 2, "(": 1}
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in string.ascii_uppercase:
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and \
            (prec[opStack.peek()] >= prec[token]):
```

```
                postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())

    return " ".join(postfixList)
```

- 计算后序表达式
  - 非常 trivial ，两个两个取操作数，注意从栈中取出后要颠倒顺序。
  - 实现

```
import Stack
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop() # 注意顺序
            result = doMath(token, operand1, operand2)
            operandStack.push(result)

    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
```

2 队列、双端队列

2.1 队列应用

2.1.1 基数排序 (radix sort)

- 属于分配式排序 (distribution sort)，或称桶排序 (bucket sort)。
- 例如，对一列十进制正整数从小到大排序：
  - 算法：k 依次从 1 到最长数据位数，
    1. 创建 10 个空的 `Queue` 实例 (编号 i 为 0~9)；
    2. 按 `lst` 内数字顺序，将第 k 位数字为 i 的数入队至第 i 个队列中；
    3. 清空 `lst` ；
    4. 按队列编号顺序，以 FIFO 的顺序依次将元素出队并添加到 `lst` 末尾。
  - 实现：

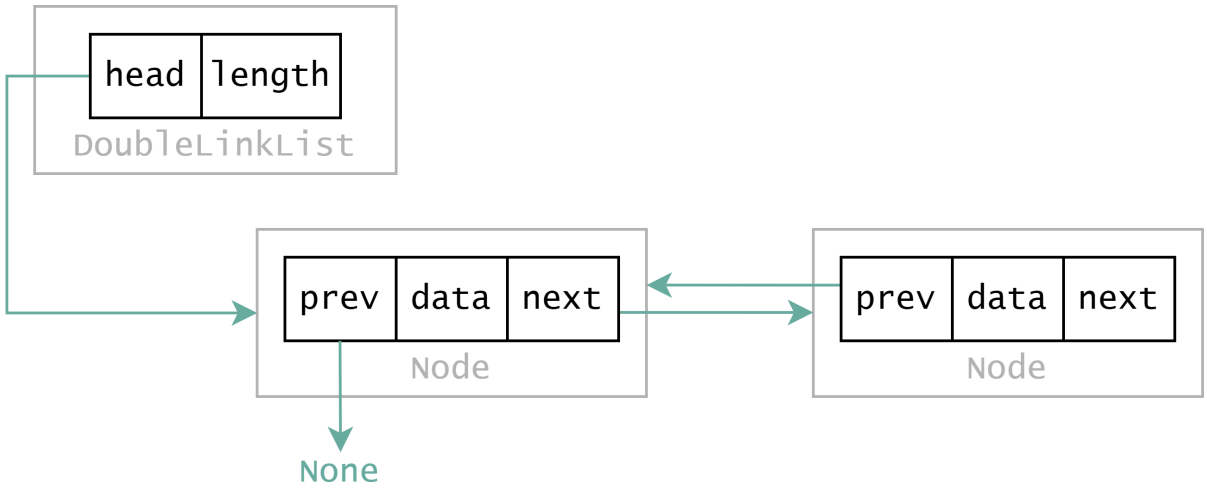
```
def RadixSort(lst):
    max_len = len(str(max(lst)))
    for i in range(max_len):
        queues = [Queue() for _ in range(10)]
        # 注意：queues = [Queue()] * 10 生成的是一个含有10个指向同一个 Queue 实例的列表！
        for item in lst:
            digit = int(item/(10**i) % 10)
            queues[digit].enqueue(item)
        lst = []
        for q in queues:
            while not q.isEmpty():
                lst.append(q.dequeue())
    return lst
```

  - 复杂度：$O(n*k)$，其中 k 为最大数据位数

3　链表、双向链表

3.1　双向链表实现

- 图示



- 实现（只以 `remove` 方法为例）

```
class Node:
    def __init__(self, data, _prev=None, _next=None):
        self.prev = _prev
        self.data = data
        self.next = _next

class DoubleLinkList:
    def __init__(self):
        self.head = None
        self._length = 0
    def is_empty(self): return self._length == 0
    def length(self): return self._length
    def nodes_list(self): pass
    def add(self, data): pass
    def append(self, data): pass
    def insert(self, pos, data): pass
    def remove(self, data):
        cur = self.head
        found = False
        while not found:
            if cur.data == data:
                found = True
            else:
                cur = cur.next
        if cur == self.head:
```

```
            self.head = cur.next
            self.head.prev = None
        else:
            cur.prev.next = cur.next
            if cur.next != None:
                cur.next.prev = cur.prev
        self._length -= 1
    def modify(self, pos, data): pass
    def search(self, data): pass
```

- 注：实现单向链表的 `remove` 方法时，由于并没有反向遍历链表的方法，解决方法是在遍历时使用两个外部引用 `current` 与 `previous` ：

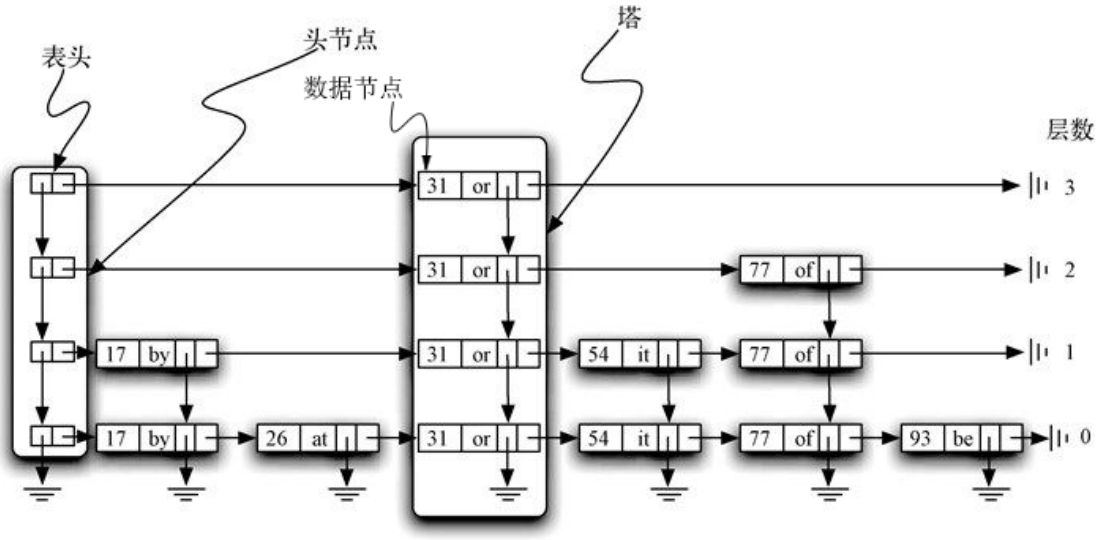
```
def remove(self, data):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.data == data:
            found = True
        else:
            previous = current
            current = current.next
    if previous == None:
        self.head = current.next
    else:
        previous.next = current.next
    self._length -= 1
```

4 跳表

- 映射 (map) 的又一实现方式

4.1 介绍

4.1.1 结构




```
class SkipList:
    def __init__(self):
        self.head = None
```

- 塔
  - 头节点

    ```
    class HeaderNode:
        def __init__(self):
            self.next = None
            self.down = None
    ```

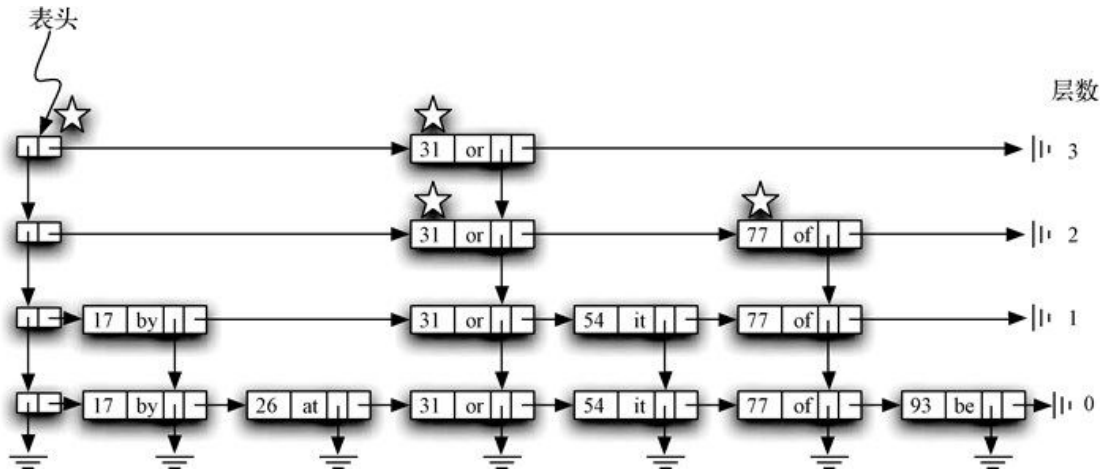
  - 数据节点

    ```
    class DataNode:
        def __init__(self, key, value):
            self.key = key
            self.val = value
            self.next = None
            self.down = None
    ```

- 层数
  - 每一层都是由数据节点构成的有序链表
  - 第 0 层是完整的有序链表

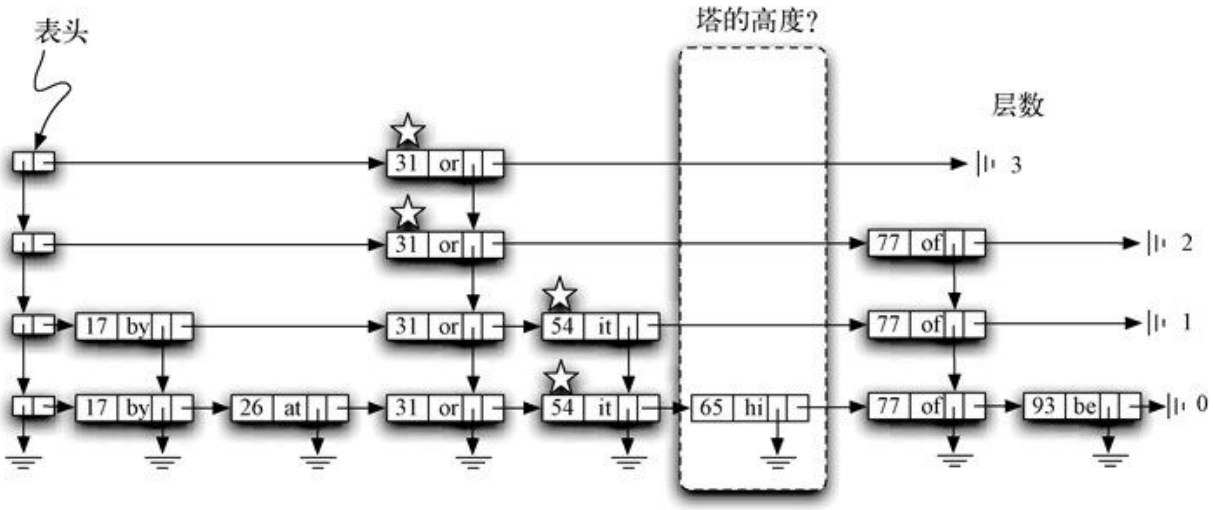
4.1.2 方法

- `search`



  - 从表头开始搜索，用 found, stop 两个布尔类型的变量控制查找
  - 往右查找
    - 若没有节点，则下移一层
    - 若有数据节点，比较键的大小
      - 目标键值小于右边键值：found 置为 True
      - 目标键值小于右边键值：下移一层。若已降至底层，将 stop 置为 True
      - 目标键值大于右边键值：右移一层
  - 进入下一层后，重复上述过程

- `insert`



  - 使用同样搜索策略，找到第 0 层的插入位置
  - 新建一个数据节点，并插入第 0 层的列表中。塔高：通过 “flip-the-coin” **随机确定**
  - 用栈保存每层的插入点

4.2 实现

- 详见 这里

4.3 分析

- `search` 和 `insert` 的时间复杂度均为 $O(\log n)$