1 基本数据结构

- 4种简单而强大的数据结构：**栈**、**队列**、**双端队列**和**列表**。它们都是有序的数据集合，其元素的顺序取决于添加顺序或移除顺序。
- **线性数据结构**：一旦某个元素被添加进来，它与前后元素的相对位置将保持不变。

1.1 栈

1.1.1 何谓栈

- **栈**是有序集合，添加操作和移除操作总发生在同一端，即“顶端”，另一端则被称为“底端”。
- 最新添加的元素将被最先移除，这种排序原则被称作**LIFO**（last-in first-out），即后进先出。

1.1.2 栈抽象数据类型

1. `Stack()`创建一个空栈。它不需要参数，且会返回一个空栈。
2. `push(item)`将一个元素添加到栈的顶端。它需要一个参数`item`，且无返回值。
3. `pop()`将栈顶端的元素移除。它不需要参数，但会返回顶端的元素，并且修改栈的内容。
4. `peek()`返回栈顶端的元素，但是并不移除该元素。它不需要参数，也不会修改栈的内容。
5. `isEmpty()`检查栈是否为空。它不需要参数，且会返回一个布尔值。
6. `size()`返回栈中元素的数目。它不需要参数，且会返回一个整数。

1.1.3 用Python实现栈

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        return self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

```
>>> s = Stack() # 栈内容：[]
>>> s.isEmpty() # 栈内容：[]
True
>>> s.push(4) # 栈内容：[4]
>>> s.push('dog') # 栈内容：[4, 'dog']
>>> s.peek() # 栈内容：[4, 'dog']
'dog'
>>> s.push(True) # 栈内容：[4, 'dog', True]
>>> s.size() # 栈内容：[4, 'dog', True]
3
>>> s.isEmpty() # 栈内容：[4, 'dog', True]
False
>>> s.push(8.4) # 栈内容：[4, 'dog', True, 8.4]
>>> s.pop() # 栈内容：[4, 'dog', True]
8.4
>>> s.pop() # 栈内容：[4, 'dog']
True
>>> s.size() # 栈内容：[4, 'dog']
2
```

1.1.4 匹配括号

- 由一个空栈开始，从左往右依次处理括号。
- 如果遇到左括号，便通过`push`操作将其加入栈中，以此表示稍后需要有一个与之匹配的右括号。
- 反之，如果遇到右括号，就调用`pop`操作。
- 只要栈中的所有左括号都能遇到与之匹配的右括号，那么整个括号串就是就是匹配的；如果栈中有任何一个左括号找不到与之匹配的右括号，则括号串就是不匹配的。
- 在处理完匹配的括号串之后，栈应该是空的。

```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == '(':
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index += 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
>>> parChecker('((()()))')
True
>>> parChecker('(((())))')
True
>>> parChecker('((()(()())))')
True
>>> parChecker('(((((((())')
False
>>> parChecker('())))')
False
>>> parChecker('((()()(()')
False
```

1.1.5 普通情况：匹配符号

- 要处理新类型的符号，可以轻松扩展括号匹配检测器。
- 每一个左符号都将被压入栈中，以待之后出现对应的右括号。
- 唯一的区别在于，当出现右符号时，必须检测其类型是否与栈顶的左符号类型相匹配。
- 如果两个符号不匹配，那么整个符号串也就不匹配。
- 同样，如果整个符号串处理完成并且栈是空的，那么就说明所有符号正确匹配。

```
def parChecker(symbolString):
    s = Stack()

    balanced = True
    index = 0

    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in '([{':
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False

        index += 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(left, right):
    lefts = '([{'
    rights = ')]}'

    return lefts.index(left) == rights.index(right)
```

```
>>> parChecker('{{([][])}()}')
True
>>> parChecker('[[{{(())}}]]')
True
>>> parChecker('[][][](){}')
True
>>> parChecker('([)]')
False
>>> parChecker('((()]))')
False
>>> parChecker('[{()]')
False
```

1.1.6 将十进制数转换成二进制数

1. $233_{10} = 2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$
2. $11101001_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

- “除以2”算法用一个简单的循环不停地将十进制数除以2，并且记录余数。
- 第一次除以2的结果能够用于区分奇数和偶数。
- 如果是偶数，则余数为0，因此个位上的数字为0；如果是奇数，则余数为1，因此个位上的数字为1。
- 可以将要构建的二进制数看成一系列数字；计算出的第一个余数是最后一位。

```
def divideBy2(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ''
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString
```

```
>>> divideBy2(233)
'11101001'
```

- 可以将`divideBy2`函数修改成接受一个十进制数以及希望转换的进制基数，“除以2”则变成“除以基数”。
- 当基数超过10时，不能再直接使用余数，这是因为余数本身就是两位的十进制数。
- 因此，需要创建一套数字来标示大于9的余数。

```
def baseConverter(decNumber, base):
    digits = '0123456789ABCDEF'

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ''
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString
```

```
>>> baseConverter(233, 8)
'351'
>>> baseConverter(233, 16)
'E9'
```

1.1.7 前序、中序和后序表达式

- 前序和后序表达式不需要括号，这两种表达式中的运算符所对应的操作数是明确的。
- 只有中序表达式需要额外的符号来消除歧义。
- 若要将任意复杂的中序表达式转换成前序表达式或后序表达式，可以先将其写作完全括号表达式，然后将括号内的运算符移到左括号处（前序表达式）或者右括号处（后序表达式）。

1.1.7.1 从中序到后序的通用转换法

当遇到左括号时，需要将其保存，以表示接下来会遇到高优先级的运算符；那个运算符需要等到对应的右括号出现才能确定其位置；当右括号出现时，便可以将运算符从栈中取出来。

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。

```
import string

```

```
def infixToPostfix(infixexpr):
    prec = {}
    prec['*'] = 3
    prec['/'] = 3
    prec['+'] = 2
    prec['-'] = 2
    prec['('] = 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)
```

```
>>> infixToPostfix('( A + B ) * ( C + D )')
'A B + C D + *'
>>> infixToPostfix('( A + B ) * C')
'A B + C *'
>>> infixToPostfix('A + B * C')
'A B C * +'
```

1.1.7.2 计算后序表达式

当遇到一个运算符时，需要用离它最近的两个操作数来计算。

1. 创建空栈`operandStack`。
2. 使用字符串方法`split`将输入的后序表达式转换成一个列表。
3. 从左往右扫描这个标记列表。

    - 如果标记是操作符，将其转换成整数并且压入`operandStack`栈中。
    - 如果标记是运算符，从`operandStack`栈中取出两个操作数。第一次取出右操作数，第二次取出左操作数。进行相应的算术运算，然后将运算结果压入`operandStack`栈中。
4. 当处理完输入表达式时，栈中的值就是结果。将其从栈中返回。

```
def postfixEval(postfixExpr):
    operandStack = Stack()

    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in '0123456789':
            operandStack.push(token)
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token, int(operand1), int(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
```

```
>>> postfixEval('4 5 6 * +')
34
>>> postfixEval('7 8 + 3 2 + /')
3
```

1.2 队列

1.2.1 何谓队列

- **队列**是有序集合，添加操作发生在“尾部”，移除操作则发生在“头部”。
- 最新添加的元素必须在队列的尾部等待，在队列中时间最长的元素则排在最前面，这种排序原则被称作**FIFO**（first-in first-out），即先进先出。

1.2.2 队列抽象数据类型

1. `Queue()`创建一个空队列。它不需要参数，且会返回一个空队列。
2. `enqueue(item)`在队列的尾部添加一个元素。它需要一个元素作为参数，不返回任何值。
3. `dequeue()`从队列的头部移除一个元素。它不需要参数，且会返回一个元素，并修改队列的内容。
4. `isEmpty()`检查队列是否为空。它不需要参数，且会返回一个布尔值。
5. `size()`返回队列中元素的数目。它不需要参数，且会返回一个整数。

1.2.3 用Python实现队列

```python
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        return self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

```
>>> q = Queue() # 队列内容：[]
>>> q.isEmpty() # 队列内容：[]
True
>>> q.enqueue(4) # 队列内容：[4]
>>> q.enqueue('dog') # 队列内容：['dog', 4]
>>> q.enqueue(True) # 队列内容：[True, 'dog', 4]
>>> q.size() # 队列内容：[True, 'dog', 4]
3
>>> q.isEmpty() # 队列内容：[True, 'dog', 4]
False
>>> q.enqueue(8.4) # 队列内容：[8.4, True, 'dog', 4]
>>> q.dequeue() # 队列内容：[8.4, True, 'dog']
4
>>> q.dequeue() # 队列内容：[8.4, True]
'dog'
>>> q.size() # 队列内容：[8.4, True]
2
```

1.2.4 模拟：传土豆

在这个游戏中，孩子们围成一圈，并依次尽可能快地传递一个土豆。在某个时刻，大家停止传递，此时手里有土豆的孩子就得退出游戏。重复上述过程，直到只剩下一个孩子。

- 该程序接受一个名字列表和一个用于计数的常量`num`，并且返回最后一人的名字。
- 使用队列来模拟一个环，假设握着土豆的孩子位于队列的头部。
- 在模拟传土豆的过程中，程序将这个孩子的名字移出对垒，然后立刻将其插入队列的尾部。
- 随后，这个孩子会一直等待，直到再次到达队列的头部。
- 在出列和入列`num`次之后，此时位于队列头部的孩子出局，新一轮游戏开始。
- 如此反复，直到队列中只剩下一个名字（队列的大小为1）。

```
def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()
```

```
>>> hotPotato(['Bill', 'David', 'Susan', 'Jane', 'Kent', 'Brad'], 7)
'Susan'
```

1.3 双端队列

1.3.1 何谓双端队列

- **双端队列**是与队列类似的有序集合。
- 与队列不同的是，双端队列对在哪一端添加和移除元素没有任何限制。
- 新元素既可以被添加到前端，也可以被添加到后端。
- 同理，已有的元素也能从任意一端移除。

1.3.2 双端队列抽象数据类型

1. `Deque()`创建一个空的双端队列。它不需要参数，且会返回一个空的双端队列。
2. `addFront(item)`将一个元素添加到双端队列的前端。它接受一个元素作为参数，没有返回值。
3. `addRear(item)`将一个元素添加到双端队列的后端。它接受一个元素作为参数，没有返回值。
4. `removeFront()`从双端队列的前端移除一个元素。它不需要参数，且会返回一个元素，并修改双端队列的内容。
5. `removeRear()`从双端队列的后端移除一个元素。它不需要参数，且会返回一个元素，并修改双端队列的内容。

6. `isEmpty()`检查双端队列是否为空。它不需要参数，且会返回一个布尔值。
7. `size()`返回双端队列中元素的数目。它不需要参数，且会返回一个整数。

1.3.3 用Python实现双端队列

```
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        return self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

```
>>> d = Deque() # 双端队列内容：[]
>>> d.isEmpty() # 双端队列内容：[]
True
>>> d.addRear(4) # 双端队列内容：[4]
>>> d.addRear('dog') # 双端队列内容：['dog', 4]
>>> d.addFront('cat') # 双端队列内容：['dog', 4, 'cat']
>>> d.addFront(True) # 双端队列内容：['dog', 4, 'cat', True]
>>> d.size() # 双端队列内容：['dog', 4, 'cat', True]
4
>>> d.isEmpty() # 双端队列内容：['dog', 4, 'cat', True]
False
>>> d.addRear(8.4) # 双端队列内容：[8.4, 'dog', 4, 'cat', True]
>>> d.removeRear() # 双端队列内容：['dog', 4, 'cat', True]
8.4
>>> d.removeFront() # 双端队列内容：['dog', 4, 'cat']
True
```

1.3.4 回文检测器

- **回文**是指从前往后读和从后往前读都一样的字符串，例如radar、toot和madam。
- 按从左往右的顺序将字符串中的字符添加到双端队列的后端。
- 利用双端队列的双重性，其前端是字符串的第一个字符，后端是字符串的最后一个字符。
- 由可以从前后两端移除元素，因此我们能够比较两个元素，并且只有在二者相等时才继续。
- 如果一直匹配第一个和最后一个元素，最终会处理完所有的字符（如果字符数是偶数），或者剩下只有一个元素的双端队列（如果字符数是奇数）。
- 任何一种结果都表明输入字符串是回文。

```
def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual
```

```
>>> palchecker('lsdkjfskf')
False
>>> palchecker('toot')
True
```

1.4 列表

- **列表**是元素的集合，其中每一个元素都有一个相对于其他元素的位置。
- 为简单起见，我们假设列表中没有重复元素。

1.4.1 无序列表抽象数据类型

1. `List()`创建一个空列表。它不需要参数，且会返回一个空列表。
2. `add(item)`假设元素`item`之前不在列表中，并向其中添加`item`。它接受一个元素作为参数，无返回值。
3. `remove(item)`假设元素`item`已经在列表中，并从其中移除`item`。它接受一个元素作为参数，并且修改列表。
4. `search(item)`在列表中搜索元素`item`。它接受一个元素作为参数，并且返回布尔值。
5. `isEmpty()`检查列表是否为空。它不需要参数，并且返回布尔值。
6. `length()`返回列表中元素的个数。它不需要参数，并且返回一个整数。

7. `append(item)`假设元素`item`之前不在列表中，并在列表的最后位置添加`item`。它接受一个元素作为参数，无返回值。
8. `index(item)`假设元素`item`已经在列表中，并返回该元素在列表中的位置。它接受一个元素作为参数，并且返回该元素的下标。
9. `insert(pos, item)`假设元素`item`之前不在列表中，同时假设`pos`是合理的值，并在位置`pos`处添加元素`item`。它接受两个参数，无返回值。
10. `pop()`假设列表不为空，并移除列表中的最后一个元素。它不需要参数，且会返回一个元素。
11. `pop(pos)`假设在指定位置`pos`存在元素，并移除该位置上的元素。它接受位置参数，且会返回一个元素。

1.4.2 实现无序列表：链表

- 无序列表需要维持元素之间的相对位置，但是并不需要再连续的内存空间中维护这些位置信息。
- 必须指明列表中第一个元素的位置。
- 一旦知道第一个元素的位置，就能根据其中的链接信息访问第二个元素，接着访问第三个元素，依次类推。
- 指向链表第一个元素的引用被称作**头**。
- 最后一个元素需要知道自己没有下一个元素。

1. `Node`类
    - **节点（node）**是构建链表的基本数据结构。
    - 每一个节点对象都必须持有至少两份信息：首先，节点必须包含列表元素，我们称之为节点的**数据变量**；其次，节点必须保存指向下一个节点的引用。

```
class Node:
    def __init__(self, initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, newdata):
        self.data = newdata

    def setNext(self, newnext):
        self.next = newnext
```

```
>>> temp = Node(93)
>>> temp.getData()
93
```

2. `UnorderedList`类

- 每一个列表对象都保存了指向列表头部的引用。

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

```
>>> mylist = UnorderedList()
```

- `isEmpty`方法检查列表的头部是否为指向`None`的引用。

```
def isEmpty(self):
    return self.head == None
```

```
>>> mylist.isEmpty()
True
```

- 为了将元素添加到类表中，需要实现`add`方法。
- 由于本例中的列表是无序的，因此新元素相对于已有元素的位置并不重要。
- 添加新节点最简便的位置就是头部，或者说链表的起点。
- 我们把新元素作为列表的第一个元素，并且把已有的元素链接到该元素的后面。

```
def add(self, item):
    temp = Node(item)
    temp.setNext(self.head) # 将新节点的next引用指向当前列表中的第一个节点
    self.head = temp # 修改列表的头节点，使其指向新创建的节点
```

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

- 接下来要实现的方法（`length`、`search`和`remove`）都基于**链表遍历**技术。
- 遍历是指系统地访问每一个节点，具体做法是用一个外部引用从列表的头节点开始访问。
- 随着访问每一个节点，我们将这个外部引用通过“遍历”下一个引用来指向下一个节点。

```
def length(self):
    current = self.head
    count = 0
    while current != None:
        count += 1
        current = current.getNext()

    return count
```

```
>>> mylist.isEmpty()
False
>>> mylist.length()
6
```

- 在无序列表中搜索一个值同样也会用到遍历技术。
- 事实上，如果遍历到列表的末尾，就意味着要找的元素不在列表中。
- 如果在遍历过程中找到所需的元素，就没有必要继续遍历了。

```
def search(self, item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()

    return found
```

```
>>> mylist.search(17)
True
>>> mylist.search(66)
False
```

- 为了将包含元素的节点移除，需要将其前面的节点中的`next`引用指向`current`之后的节点。
- 然而，并没有反向遍历链表的方法。
- 这一困境的解决方法就是在遍历链表时使用两个外部引用。
- `current`与之前一样，标记在链表中的当前位置；新的引用`previous`总是指向`current`上一次访问的节点。

```
def remove(self, item):
    current = self.head
    previous = None
```

```
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```

```
>>> mylist.remove(26)
>>> mylist.length()
5
```

1.4.3 有序列表抽象数据类型

1. `OrderedList()`创建一个空有序列表。它不需要参数，且会返回一个空列表。
2. `add(item)`假设`item`之前不在列表中，并向其中添加`item`，同时保持整个列表的顺序。它接受一个元素作为参数，无返回值。
3. `remove(item)`假设`item`已经在列表中，并从其中移除`item`。它接受一个元素作为参数，并且修改列表。
4. `search(item)`在列表中搜索`item`。它接受一个元素作为参数，并且返回布尔值。
5. `isEmpty()`检查列表是否为空。它不需要参数，并且返回布尔值。
6. `length()`返回列表中元素的个数。它不需要参数，并且返回一个整数。
7. `index(item)`假设`item`已经在列表中，并返回该元素在列表中的位置。它接受一个元素作为参数，并且返回该元素的下标。
8. `pop()`假设列表不为空，并移除列表中的最后一个元素。它不需要参数，且会返回一个元素。
9. `pop(pos)`假设在指定位置`pos`存在元素，并移除该位置上的元素。它接受位置参数，且会返回一个元素。

1.4.4 实现有序列表

- 在实现有序列表时必须记住，元素的相对位置取决于它们的基本特征。
- `OrderedList`类的构造方法与`UnorderedList`类的相同。

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

  - 因为`isEmpty`和`length`仅与列表中的节点数目有关，而与实际的元素值无关，所以这两个方法在有序列表的实现与在无序列表中一样。

- 同理，由于仍然需要找到目标元素并且通过更改链接来移除节点，因此`remove`方法的实现也一样。
- 剩下的两个方法，`search`和`add`需要做一些修改。
- 如果目标元素不在列表中，可以利用元素有序排列这一特性尽早终止搜索。

```
def search(self, item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()

    return found
```

- 需要修改最多的是`add`方法，我们需要再已有链表中为新节点找到正确的插入位置。
- 当访问完所有节点（`current`是`None`）或者当前值大于要添加的元素时，就找到了插入位置。

```
def add(self, item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext = temp
```

1.4.5 链表分析

- 在分析链表操作的时间复杂度时，考虑其是否需要遍历列表。
- `isEmpty`方法的时间复杂度是$O(1)$。
- `length`方法的时间复杂度是$O(n)$。
- 无序列表的`add`方法的时间复杂度是$O(1)$。
- 有序列表的`search`、`remove`和`add`方法的时间复杂度是$O(n)$。