1. 课程： 链表

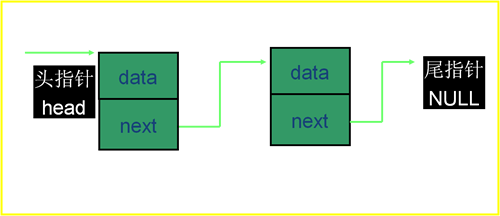
* 1. 什么是单向链表;（了解）
* 2. 为什么需要单向链表;（了解）
* 3. 单链表的实现;（掌握）
* 4. 单循环链表的含义及操作;(了解)
* 5. 双向链表的含义及操作.(了解)

1. 单向链表

2.1 什么是单向链表？

* + - 1. 单向链表又称**单链表**,是一种线性表.
      2. 每个存储单元分为**元素域**和**链接域**。链接域里存放**下一个**结点的位置信息.
      3. 有一个变量p保存链表的第一个结点的地址,这个变量称为**头指针**. 如图,头指针(head)指向链表的第一个结点(**首结点**,或**头结点)**，从头指针出发能到表中的任意结点.
      4. 最后一个结点**(尾结点)**的链接域指向NULL.

分析：由于存储单元中有链接域,存储空间不必连续.



* 1. 结点的实现

我们可以用python程序来**实现**单向链表这一数据结构。我们将数据结构定义成类。我们定义数据结构时，要把数据本身和可对它实施的操作放到一起，作为一个类。

1. **class** LNode(object):
2. # 初始化 方法 给对象的两个域赋值
3. **def** \_\_init\_\_(self, item):
4. # item 存放数据元素
5. self.item = item
6. #\_next 是下一结点的标识
7. self.\_next = None
   1. 单向链表的操作

1. 判断链表是否为空: is\_empty()

2. 求链表长度: length()

3. 遍历: travel()

4. 在头(尾)部添加结点: add(item) (append(item))

5. 在指定位置添加结点: insert(pos,item)

6. 删除结点 : remove(item)

7. 查找结点 : search(item)

* 1. 单向链表的实现

Python中，为一个变量赋值,比如

1. a = 100

变量a的本质是什么？

a = 100的含义：**让a这个变量指向100，即a保存对象100的地址**.

**Python中，任何变量保存的都是对象的地址**. 这样有什么好处？

* + - 1. Python变量不需要提前声明(Python中没有类型声明);
      2. 换句话说，Python是动态类型的.

1. a = 100
2. **print** type(a)
3. a = "hello"
4. **print** type(a)

单链表的实现

结点1:

1. data = 100    # data 变量指向对象100，即保存了100的地址
2. next = Node2 　# next变量保存了Node2的地址

结点2:

1. data = 200    # data 变量指向对象200，即保存了200的地址
2. next = Node3 　# next变量保存了Node3的地址

……

结点N:

1. data = 400    # data 变量指向对象400，即保存了400的地址
2. next = NULL 　# next变量不指向任何对象

效果：　用"="来实现存储单元的串联

用Python代码实现单链表:

定义结点类

1. **class** Node(object):
2. """结点"""
3. **def** \_\_init\_\_(self,item):
4. self.elem = item # elem 是 self的属性
5. self.next = None # 初始时,还不知链接域指向哪里

定义单链表及初始化

1. **class** SingleLinkList(object):
2. """定义单链表"""
3. **def** \_\_init\_\_(self,node=None): # 默认参数为None
4. self.\_head = node # 私有属性

判断链表是否为空: is\_empty()

1. **def** is\_empty(self):
2. """链表是否为空"""
3. **return** self.\_head == None

获得链表长度: length()

1. **def** length(self):
2. """返回链表长度"""
3. cur = self.\_head
4. #count 记录数量
5. count = 0  #可处理空表
6. **while** cur != None:
7. count += 1
8. cur = cur.next
9. **return** count

遍历链表: travel()

1. **def** travel(self):
2. """遍历链表"""
3. cur = self.\_head
4. **while** cur != None:
5. **print** cur.elem
6. cur = cur.next

链表头部增加元素: add(item)

1. **def** add(self,item):
2. """链表头部增加元素"""
3. node = Node(item)
4. #注意以下操作的先后顺序
5. node.next = self.\_head  #将 self.\_\_head 指向的对象的地址保存到node.next
6. self.\_head = node #  使self.\_\_head 指向新的结点

链表尾部增加元素: append(item)

1. **def** append(self,item):
2. """链表尾部增加元素"""
3. node = Node(item)
4. **if** self.is\_empty():
5. self.\_head = node
6. **else**:
7. cur = self.\_head
8. **while** cur.next != None:
9. cur = cur.next
10. cur.next = node

在指定位置添加元素: insert(pos,item)

1. **def** insert(self,pos, item):
2. """
3. 在指定位置添加元素
4. :param pos     从0开始
5. """
6. **if** pos <= 0:
7. self.add(item)
8. **elif** pos > self.length()-1:
9. self.append(item)
10. **else**:
11. pre = self.\_head
12. count = 0      #为了找到pos而设置
13. **while** count < pos-1:
14. pre = pre.next
15. count += 1
16. node = Node(item)  # 构建结点
17. node.next = pre.next
18. pre.next = node

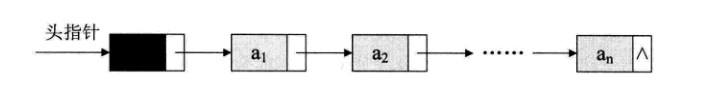
删除结点: remove(item)

1. **def** remove(self,item):
2. """删除结点"""
3. cur = self.\_head
4. pre = None
5. **while** cur != None:
6. **if** cur.elem == item:
7. #判断是否为头结点
8. **if** cur == self.\_head:
9. self.\_head = cur.next
10. **else**:
11. pre.next = cur.next # 删除结点
12. **break**
13. **else**:
14. pre = cur
15. cur = cur.next

查找结点是否存在: search(item)

1. **def** search(self,item):
2. """查找结点是否存在"""
3. cur = self.\_head
4. **while** cur != None:
5. **if** cur.elem == item:
6. **return** True
7. **else**:
8. cur = cur.next
9. **return** False
   1. 带头结点的线性链表

在插入/删除时可能修改链表的首指针,带来了不必要的麻烦. 见前面的再头部添加结点那个例子.故,给单链表增加一个**头结点**.如图:



头结点的特征:

1.头结点的数据域无意义

2.头结点的指针域指向线性表的第一个结点

3.有**头指针**指向头结点

4.头结点指针域为NULL时,线性表为空表

* 1. 链表与顺序表的区别

1. 链表失去了随机读取的优点

2. 链表增加了结点的指针域，空间开销比较大

3. 各种操作的时间复杂度的对比(如表１所示)

|  |  |  |
| --- | --- | --- |
| 操作 | 链表 | 顺序表 |
| 访问元素 | O(n) | O(1) |
| 在头部插入 | O(1) | O(n) |
| 在尾部插入 | O(1) | O(1) |
| 在中间插入 | O(1) | O(n) |

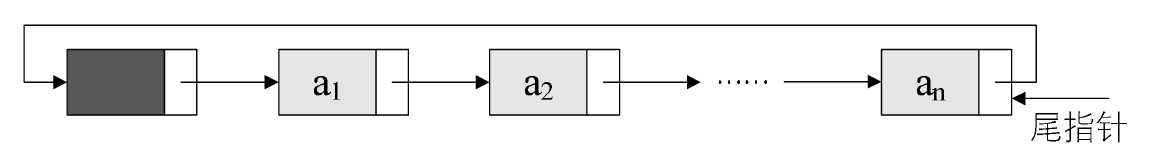
表１．链表和顺序表的基本操作的时间复杂度对比

1. 单向循环链表
   1. 什么是循环链表？

1. 是另一种形式的链式存储结构.

2. 使线性表最后一个结点的指针(尾指针)指向链表的首结点，这种形式的线性表就叫**循环链表**.

3. 循环链表呈环状.

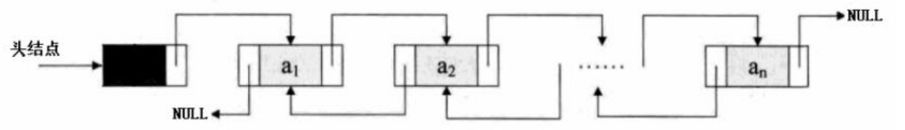
循环链表的特点:

* + - 1. 从表中任意一个结点出发,均可访问到表中的其他结点.
      2. 为避免死循环,在循环链表的第一个结点前,附加一个特殊的结点来作标识,此结点称为循环链表的**头结点.**
      3. 从一个结点出发,依次对每个结点执行某种操作,一旦回到该结点,则表示该操作已经完成.
  1. 循环链表的操作和实现

单向链表的操作和单链表**基本一致**,惟一的**差别**是:循环结束条件是cur是否为None,而是它是否等于**首指针**head.

1. 双向链表
   1. 为什么需要双向链表?

单向链表中,找某一结点的后继很方便(时间复杂度=O(1)),但找其前驱则需要遍历整个链表(时间复杂度=O(n)). 为了克服这个缺点,引入**双向链表**. 例如:



* 1. 什么是双向链表?

双向链表的特征:

* + - 1. 每个结点包括**数据域**,**左方向指针域**(llink)和**右方向指针域**(rlink);
      2. llink指向结点的直接前驱,首结点的llink指向尾结点;
      3. rlink 指向结点的直接后续,尾结点的rlink指向首结点.
  1. 双向链表的操作和实现
     + 1. 访问结点的前驱和后继,只需一次搜索;
       2. 插入和删除时要修改两个方向的指针,每个方向的操作的实现过程与单链表相同.

1. 课程总结
   1. 重点

1. 掌握”Python是动态类型的”这一观念.

2. 掌握单链表操作和实现.

* 1. 难点

1. 理解链表的操作.

2. Python是动态类型的,但它也是强类型语言.你只能对一个对象进行适合该类型的有效的操作.

* 1. 实现单链表的Python代码:

1. #!/usr/bin/python
2. # -\*- coding: utf-8 -\*-
4. **class** Node(object):
5. """结点"""
6. **def** \_\_init\_\_(self,item):
7. self.elem = item # elem 是 self的属性
8. self.next = None # 初始时,不知链接域指向哪里
10. **class** SingleLinkList(object):
11. """定义单链表"""
12. **def** \_\_init\_\_(self,node=None): # 默认参数为None
13. self.\_head = node # 私有属性
15. **def** is\_empty(self):
16. """链表是否为空"""
17. **return** self.\_head == None
19. **def** length(self):
20. """返回链表长度"""
21. cur = self.\_head
22. #count 记录数量
23. count = 0  #也可处理空表
24. **while** cur != None:
25. count += 1
26. cur = cur.next
27. **return** count
29. **def** travel(self):
30. """遍历链表"""
31. cur = self.\_head
32. **while** cur != None:
33. **print** cur.elem
34. cur = cur.next
36. **def** add(self,item):
37. """链表头部增加元素"""
38. node = Node(item)
39. #注意以下操作的先后顺序
40. node.next = self.\_head  # self.\_\_head 指向的对象的地址保存到node.next
41. self.\_head = node #  使self.\_\_head 指向新的结点
43. **def** append(self,item):
44. """链表尾部增加元素"""
45. node = Node(item)
46. **if** self.is\_empty():
47. self.\_head = node
48. **else**:
49. cur = self.\_head
50. **while** cur.next != None:
51. cur = cur.next
52. cur.next = node
54. **def** insert(self,pos, item):
55. """
56. 在指定位置添加元素
57. :param pos     从0开始
58. """
59. **if** pos <= 0:
60. self.add(item)
61. **elif** pos > self.length()-1:
62. self.append(item)
63. **else**:
64. pre = self.\_head
65. count = 0      #为了找到pos而设置
66. **while** count < pos-1:
67. pre = pre.next
68. count += 1
69. node = Node(item)  # 构建结点
70. node.next = pre.next
71. pre.next = node
73. **def** remove(self,item):
74. """删除结点"""
75. cur = self.\_head
76. pre = None
77. **while** cur != None:
78. **if** cur.elem == item:
79. #判断是否为头结点
80. **if** cur == self.\_head:
81. self.\_head = cur.next
82. **else**:
83. pre.next = cur.next # 删除结点
84. **break**
85. **else**:
86. pre = cur
87. cur = cur.next
89. **def** search(self,item):
90. """查找结点是否存在"""
91. cur = self.\_head
92. **while** cur != None:
93. **if** cur.elem == item:
94. **return** True
95. **else**:
96. cur = cur.next
97. **return** False
99. **if** \_\_name\_\_ == "\_\_main\_\_":
100. ll =SingleLinkList()
101. **print** "is empty? ",ll.is\_empty()
102. **print** "length= ",ll.length()
104. ll.append(200)
105. **print** "length= ", ll.length()
106. ll.add(8)
107. ll.append(3)
108. ll.append(4)
109. **print** "length= ", ll.length()
111. ll.insert(-1,10)
112. ll.travel()
113. **print** "length= ", ll.length()
114. ll.insert(3,100)
115. ll.travel()
116. **print** "length= ", ll.length()
117. ll.remove(4)
118. ll.travel()
119. **print** "length= ", ll.length()
120. ll.search(100)
121. 课后练习
     * + 1. 带头结点的单链表head为空的判定条件是\_\_\_\_\_.
122. head = None
123. head.next = None
124. head.next = head
125. head != None
     * + 1. 假设我们需要频繁地在线性表的一端插入和删除元素.如果用顺序表实现,应该以那一端作为操作端? 为什么? 如果用链表实现呢?
         2. 测试如下Python代码,看看它实现了什么功能? 解释为什么可以这样写.
126. a = 100
127. b = 200
128. a, b = b, a
129. 扩展知识或课外阅读推荐
     1. 课外阅读

刘清,张小勇,王琼 <<数据结构>>(第三版),电子工业出版社 (2010)