作业 2

xwzeng

2023 年 4 月 13 日

1

双端队列的 ADT 为：

ADT Deque {

数据对象：D={D1,D2,…,Dn}

基本操作：

D.init(D0)：使用序列 D0 初始化队列 D

len(D)：返回双端队列 D 的长度

D.is_empty()：检查双端队列 D 是否为空，如为空则返回 True

D.add_first(e)：向双端队列 D 的队头添加一个元素

D.add_last(e)：向队列 D 的队尾添加一个元素

D.delete_first()：从双端队列 Q 中移除并返回第一个元素

D.delete_last()：从双端队列 Q 中移除并返回最后一个元素

D.first()：在不移除的前提下，返回双端队列的第一个元素

D.last()：在不移除的前提下，返回双端队列的最后一个元素

D.clear()：清空队列 D

} ADT Deque

定义 Empty 异常类。

[1]:
```
class Empty(Exception):
    """Error attempting to access an element from an empty container."""
    pass
```

定义基于数组的双端队列类。在这里我自定义了一个基于列表的动态数组类 `DynamicArray`（在 Source Code/Ch05/dynamic_array.py 的基础上新添加了 `is_empty`、`locate`、`delete` 方法），然后让双端队列 `ArrayDeque` 继承该类。

[2]:
```python
from my_array import DynamicArray # Import the DynamicArray class

class ArrayDeque(DynamicArray):
    def __init__(self, D0 = None):
        """Initialize the deque using D0."""
        # note: we do not consider other data types of D0 except Nonetype,
        # DynamicArray, ArrayDeque and list
        super().__init__() # use inherited construction method
        if type(D0) == DynamicArray or type(D0) == ArrayDeque:
            self._resize(D0._capacity)
            self._A = D0._A
            self._n = D0._n
        if type(D0) == list:
            self._resize(len(D0))
            for value in D0:
                self.append(value)
    def __len__(self):
        """Return the number of elements in the deque."""
        return self._n
    def is_empty(self):
        """Return True if the deque is empty."""
        return self._n == 0
    def add_first(self, e):
        """Add an element to the front of the deque."""
        self.insert(0, e)
    def add_last(self, e):
        """Add an element to the back of the deque."""
        self.append(e)
    def delete_first(self):
        """Remove and return the element from the front of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        value = self._A[0]
        self.delete(0)
        return value
    def delete_last(self):
```

```
        """Remove and return the element from the front of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        value = self._A[self._n - 1]
        self.delete(self._n - 1)
        return value
    def first(self):
        """Return (but do not remove) the element at the front of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        return self._A[0]
    def last(self):
        """Return (but do not remove) the element at the back of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        return self._A[self._n - 1]
    def clear(self):
        """Remove all elements from the deque."""
        if self.is_empty():
            raise Empty("Deque is empty.")
        while self._n != 0:
            self.delete_last()
```

[4]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all' # 显示全部输出结果
```

使用 4. 数组、栈与队列（2）.pdf 中第 34 页的例子进行验证，发现结果一致。

[5]:
```
D = ArrayDeque()
D.add_last(5)
D.add_first(3)
D.add_first(7)
D.first()
D.delete_last()
len(D)
D.delete_last()
D.delete_last()
```

```
D.add_first(6)
D.last()
D.add_first(8)
D.is_empty()
D.last()
```

```
[5]: 7
```

```
[5]: 5
```

```
[5]: 2
```

```
[5]: 3
```

```
[5]: 7
```

```
[5]: 6
```

```
[5]: False
```

```
[5]: 6
```

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

尝试课件例子中没有的操作：用 D0 对双端队列进行初始化、清空双端队列，结果均符合预期。

```
[6]: D0 = [1, 2, 3, 4, 5] # the type of D0 can also be DynamicArray or ArrayDeque
     D = ArrayDeque(D0)
     D.first()
```

```
D.delete_last()
len(D)
D.clear()
D.is_empty() # check whether all the elements in the deque are deleted
```

```
[6]: 1
```

```
[6]: 5
```

```
[6]: 4
```

```
[6]: True
```

2

快速高效逆置单向链表（没有哨兵节点）的递归算法思路为：

1. 输入的参数是一个单向链表的头节点。
2. 基本情况为链表为空或链表只有头节点，返回输入的头节点。
3. 令新的头节点为：以当前头节点指向的下一个节点为头节点的链表逆置后的头节点（递归到最后就是原链表的最后一个节点）。
4. 把当前遍历的头节点加到它后面节点逆序后链表的尾部。
5. 让当前遍历的头节点指向 None。

伪代码：

Algorithm reverse(head):
Input: The head node of a singly linked list L
Output: The new head node of the reversed linked list
if head is None or head.next is None **then**
    return head
else
    new_head = reverse(head.next)
    head.next.next = head
    head.next = None
    return new_head

代码实现：

[7]:
```
class Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = 'element', 'next'
    def __init__(self, element, next):
        self.element = element
        self.next = next
```

[8]:
```
def generate_linkedlist(x: list) -> Node:
    """Transform the list to a linked list and return the head node."""
    if len(x) == 0:
        return 'Please input a non-empty list.'
    head = Node(x[0], None)
    tmp = head
    for i in x[1:]:
        tmp.next = Node(i, None)
        tmp = tmp.next
    return head
```

[9]:
```
def display_linkedlist(head: Node):
    """Display all the elements in the linked list."""
    if head.next is None:
        return 'Please input a head node of a non-empty singly linked list.'
    x = []
    tmp = head
    while tmp is not None:
        x.append(str(tmp.element))
        tmp = tmp.next
    print('->'.join(x))
```

[10]:
```
def reverse_linkedlist_1(head: Node) -> Node:
    """Reverse a singly linked list using recursion."""
    if head is None or head.next is None:
        return head
    else:
        new_head = reverse_linkedlist_1(head.next)
        head.next.next = head
```

```
        head.next = None
        return new_head
```

[11]:
```
head = generate_linkedlist([1, 2, 3, 4, 5, 6])
display_linkedlist(head)
```

```
1->2->3->4->5->6
```

[12]:
```
new_head = reverse_linkedlist_1(head)
display_linkedlist(new_head)
```

```
6->5->4->3->2->1
```

3

仅使用固定额外空间的逆置单向链表（没有哨兵节点）算法思路为：

1. 输入的参数是一个单向链表的头节点。
2. 检查链表是否为空或链表只有头节点。
3. 用 `tmp_now` 表示当前节点，用 `tmp_prev`、`tmp_next` 分别表示原链表中当前节点的上一个、下一个节点。
4. 记录原链表的头节点和其后两个节点，让下一个节点指向头节点，再让头节点指向 None，转变为尾节点。
5. 将曾经的 `tmp_prev` 更新为 `tmp_now`，再将 `tmp_now` 更新为 `tmp_next`，然后记录原链表中当前节点的下一个节点 `tmp_next`，最后将 `tmp_now` 指向 `tmp_prev`。
6. 重复进行第 5 步，直到即将前往的节点为空时停止，返回当前节点 `tmp_now`。

伪代码：

Algorithm reverse(head):
Input: The head node of a singly linked list L
Output: The new head node of the reversed linked list
if head is None or head.next is None **then**
    return head
tmp_now = head.next
tmp_next = tmp_now.next

tmp_now.next = head
head.next = None
while tmp_next not None **do**
    tmp_prev = tmp_now
    tmp_now = tmp_next
    tmp_next = tmp_now.next
    tmp_now.next = tmp_prev
return tmp_now

代码实现：

[13]:
```
def reverse_linkedlist_2(head):
    """Reverse a singly linked list using only a constant amount of additional␣
 ↪space."""
    if head is None or head.next is None:
        return head
    tmp_now = head.next # the 2nd node
    tmp_next = tmp_now.next # record the 3st node
    tmp_now.next = head # make 2nd node point to head
    head.next = None # make head node point to None
    while tmp_next is not None:
        tmp_prev = tmp_now # previous node
        tmp_now = tmp_next # current node
        tmp_next = tmp_now.next # next node
        tmp_now.next = tmp_prev # make current node point to previous node
    return tmp_now
```

[14]:
```
head = generate_linkedlist([1, 2, 3, 4, 5, 6])
display_linkedlist(head)
```

```
1->2->3->4->5->6
```

[15]:
```
new_head = reverse_linkedlist_2(head)
display_linkedlist(new_head)
```

```
6->5->4->3->2->1
```

4

定义链表 LinkedList 类，该类含有头节点（哨兵节点），且只定义了以下在本题中需要用到的函数：

- `is_empty`: 检查链表是否为空。
- `first`: 返回头部元素的值。
- `remove_first`: 在头部删除元素。
- `add_last`: 在尾部添加元素。
- `display`: 可视化链表，便于查看结果。

[16]:
```
class LinkedList:
    #------------------------- nested _Node class -------------------------
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'
        def __init__(self, element, next):
            self._element = element
            self._next = next
    #------------------------- linked list methods -------------------------
    def __init__(self):
        """Create an empty linked list."""
        self._head = self._Node(None, None)
        self._tail = None
        self._size = 0
    def __len__(self):
        """Return the number of elements in the linked list."""
        return self._size
    def is_empty(self):
        """Return True if the linked list is empty."""
        return self._size == 0
    def first(self):
        """Return (but do not remove) the element at the head."""
        if self.is_empty():
            raise Empty('Linked list is empty.')
        return self._head._next._element
    def remove_first(self):
        """Remove an element at the head."""
        if self.is_empty():
```

```
            raise Empty('Linked list is empty.')
        self._head._next = self._head._next._next
        self._size -= 1
    def add_last(self, e):
        """Insert an element at the tail."""
        if self.is_empty():
            self._tail = self._Node(e, None)
            self._head._next = self._tail
        else:
            self._tail._next = self._Node(e, None)
            self._tail = self._tail._next
        self._size += 1
    def display(self):
        """Display all the elements in the linked list."""
        if self.is_empty():
            raise Empty('Linked list is empty.')
        else:
            x = []
            for i in range(self._size):
                x.append(str(self.first()))
                self.add_last(self.first())
                self.remove_first()
            print('->'.join(x))
```

定义输入有序列表输出有序单向链表的函数。为避免用户输入的列表是乱序的，在数据结构转换前先对输入列表进行排序，以保证生成的单向链表是有序的。

[17]:
```
def ordered_linkedlist(x: list) -> LinkedList:
    x = sorted(x)
    out = LinkedList()
    for i in x:
        out.add_last(i)
    return out
```

定义将两个有序的单向链表合并为一个有序链表，且合并后使原链表为空的函数。该算法思路为：

1. 创建新链表以存放结果。

2. 当单向链表 a, b 都非空时，比较 a, b 的头元素，将二者之间更小的元素加入新链表的尾部，然后在原链表中删除该头元素。

3. 当 a, b 至少有一个为空时：

   - 将非空的单向链表中的头元素添加到新链表的尾部，在原链表中删除该头元素。
   - 循环上一步直至该单向链表为空，返回合并后的新链表。

该算法避免了在尾部删除元素（时间复杂度高），算法效率很高。

[18]:
```
def merge_linkedlist(a: LinkedList, b: LinkedList) -> LinkedList:
    out = LinkedList()
    while len(a) > 0 and len(b) > 0:
        if a.first() <= b.first():
            out.add_last(a.first())
            a.remove_first()
        else:
            out.add_last(b.first())
            b.remove_first()
    while len(a) > 0:
        out.add_last(a.first())
        a.remove_first()
    while len(b) > 0:
        out.add_last(b.first())
        b.remove_first()
    return out
```

将列表转换为有序的单向链表 A, B。

[19]:
```
A = ordered_linkedlist([2, 6, 4, 7, 0])
B = ordered_linkedlist([5, 2, 3, 1, 9, 8, 6])
A.display()
B.display()
type(A), type(B)
```

```
0->2->4->6->7
1->2->3->5->6->8->9
```

[19]:
```
(__main__.LinkedList, __main__.LinkedList)
```

将有序的单向链表 A, B 合并为新的有序链表 AB_merged。

[20]:
```
AB_merged = merge_linkedlist(A, B)
AB_merged.display()
type(AB_merged)
```

```
0->1->2->2->3->4->5->6->6->7->8->9
```

[20]:
```
__main__.LinkedList
```

检查合并后原链表 A, B 是否都为空。

[21]:
```
A.is_empty(), B.is_empty()
```

[21]:
```
(True, True)
```

5

由树的性质，树的节点数等于所有节点度的总和加 1。

$$n = \sum_{i=1}^{n} d_i + 1$$

在本题中，所有内部节点的度都为 k，叶子节点的度为 0。令 n_I 表示内部节点的个数，有：

$$n = \sum_{i=1}^{n_I} k + 1 = k * n_I + 1$$

又由树的性质，树的节点总数 = 内部节点数 + 叶子节点数。令 n_E 表示叶子节点的个数，有：

$$n = n_I + n_E$$

联立两式，求解得到：

$$n_E = n - \frac{n-1}{k}$$

6

a)

$n = 2023$，由完全二叉树的定义，除了最底层，其他层都应被填满。根据二叉树的性质，有：

$$\begin{aligned} 2^{(h)} - 1 \leq 2023 \leq 2^{(h+1)} - 1 \\ \log_2(2024) - 1 \leq h \leq \log_2(2024) \\ 9.98 \leq h \leq 10.98 \\ \Rightarrow h = 10 \end{aligned}$$

该二叉树的高度 $h = 10$。

考虑到根节点所在的层，树的层数应为高度 +1，即这棵完全二叉树有 11 层。

b)

首先考虑高度为 10 的完美二叉树，共有节点 $n = 2^{11} - 1 = 2047$ 个。其最底层都应为叶子节点（度为 0），该二叉树叶子节点的个数应为 $n_0 = 2^{10} = 1024$ 个。

在本题中，$n = 2023$，相比于完美二叉树第 10 层少了 24 个节点，即第 9 层有 $24/2 = 12$ 个叶子节点（度为 0，没有孩子节点），因此该树的叶子节点数应为 $1024 - 24 + 12 = 1012$ 个。

c)

由 b) 中分析可知，第 10 层比完美二叉树少的节点数为 24，是偶数，说明在这棵树中不存在度为 1 的内部节点，即内部节点（含根节点）的度都为 2。问题转化为求该树的内部节点个数。由树的性质，树的节点总数 = 内部节点数 + 叶子节点数，故有：

$$n = n_2 + n_0$$
$$n_2 = 2023 - 1012 = 1011$$

因此该二叉树有 1011 个度为 2 的节点。

7

令哨兵节点为根节点的父节点，且根节点为哨兵节点的左孩子节点，加入哨兵节点可以简化链式二叉树的实现过程。

我写了一个新的 `MyLinkedBinaryTree` 类，它继承了 Source Code Ch08 中的 `LinkedBinaryTree` 类，重写了 5 个方法：`__init__`（取消 `_root`，增加 `_sentinel` 哨兵节点）、`root`、`_add_root`、`_delete` 和 `_attach`，详见”week5_7.py”。

由于篇幅原因，这里仅展示新的 `_delete` 方法，此时不再需要考虑删除根节点导致的指向性问题和条件判断，更加简洁。

[ ]:
```python
def _delete(self, p): # 重写删除位置 p 节点的方法
    node = self._validate(p) # 将位置拆包为节点
    if self.num_children(p) == 2: # 当该节点有两个孩子时报错
        raise ValueError('Position has two children')
    child = node._left if node._left else node._right # 有可能为空
    if child is not None:
        child._parent = node._parent  # 让孩子节点的父亲变为它原来父节点的父节点
```

```
    parent = node._parent
    if node is parent._left: # 让新的父节点指向孩子节点
        parent._left = child
    else:
        parent._right = child
    self._size -= 1
    node._parent = node # 将该节点的父亲指向自己，便于内存回收
    return node._element
```

简单测试一下新类。

[22]:
```
from week5_7 import MyLinkedBinaryTree

T = MyLinkedBinaryTree()
root = T._add_root(1)
root == T.root() # 检查根节点返回位置是否正确
n1 = T._add_left(root, 2)
len(T)
n2 = T._add_left(n1, 3)
n3 = T._add_right(n1, 4)
T.depth(n3)
T.height(n1)
T._delete(root) # 删除根节点
T.root() == n1 # 检查根节点是否变为原根节点的左孩子 n1
```

[22]: True

[22]: 2

[22]: 2

[22]: 1

[22]: 1

[22]: True

附加题

在这里我简单解释一下这题的思路，具体代码实现请见 `week5_ 附加题.py`。

我通过继承这次作业第 7 题写的 MyBinaryTree 类（包含哨兵节点），构造了一个二叉搜索树类 MyBinarySearchTree。在该类中，我主要实现了以下功能：

1. 重写 `__init__` 构造方法：输入正整数 n 与一个长度为 n 的整数列表，严格按照顺序和题目要求构造二叉树。为了确保结果的唯一性，我直接在初始化时就将度为 0 或 1 的节点的孩子节点用'#' 补全。

2. 注意，补全实际上没有增加这棵树的节点的数量、以及以某一节点作为根节点的子树的节点数量，因此我重写了 `_add_left`、`_add_right` 函数，添加了一个布尔型参数 `complete`，缺省值为 `False`。当 `complete` 为 `False` 时，节点数量正常增长；当 `complete` 为 `True` 时，这两个函数转换为补全模式，仅将'#' 添加至节点，而不增加整棵树节点和子树节点数量。另外，我在这里为了简化问题，没有对树的高度、深度、孩子节点数量等进行修正，因为实现二叉搜索树不需要使用这些功能，且可以通过对字符串的条件判断排除它们对二叉搜索树的功能的影响。

3. 新的节点类 `_Node`：添加了一个新的属性 `_subtreenodes`，表示以该节点为根节点的子树的节点数量。需要注意的是缺省值应为 1 而非 0，这是因为如果该节点的度为 0，以它为根节点的子树的节点不为 0，根节点也应被包含在内。

4. 继承父类中的 `Position` 类：添加了一个 `subtreenodes` 方法，返回以某个节点为根节点的子树的节点数量。

5. 重写了 `_add_root` 方法：为哨兵节点的 `_subtreenodes` 属性赋值为 2（子树节点包含哨兵节点和根节点）。其实不赋值也可以，因为哨兵节点没有意义。

6. 重写了 `_add_left`、`_add_right` 方法：在原先添加孩子节点的基础上增加了一个功能，即将这个新添加的节点的所有祖先节点的 `_subtreenodes` 属性都增加 1。

7. 重写了 `_delete` 方法：增加了一个功能，即将这个被删除节点的所有祖先节点的 `_subtreenodes` 属性都减少 1。

8. 新建 `_complete` 方法：为使输出结果的表示意义唯一，对于度为 0 或 1 的节点，用参数 `fill` 补全成度为 2 的节点，缺省值为'#'。

9. 新建 `display` 方法：通过参数 `method` 指定遍历方式，然后以空格隔开输出遍历的节点元素值。前序遍历、中序遍历、后序遍历都在 Ch08/Source Code 中 `binary_tree.py` 和 `tree.py` 被定义了，因此我这里就没有再重新写。

10. 新建 `_subtree_search` 方法：递归搜索以 p 为根节点的子树中是否存在元素 e，若不存在或该位置的元素值为字符串'#' 则返回 None。

11. 新建 `find_position` 方法：返回元素值为 e 的节点的位置。

下面对作业要求中的例子进行简单测试。

[23]:
```
from week5_ 附加题 import MyBinarySearchTree
```

[24]:
```
n = int(input('请输入一个正整数 n: \n'))
numbers = input('\n请输入 n 个正整数，以空格隔开，代表存储在节点的关键字: \n').
 ↪split()
numbers = [int(number) for number in numbers]
BST = MyBinarySearchTree(n, numbers)
print('\n前序遍历: ')
BST.display(method='preorder')
print('\n中序遍历: ')
BST.display(method='inorder')
print('\n后序遍历: ')
BST.display(method='postorder')
print('\n____ 进入查询状态 ____')
while True:
    value = input('\n请输入您想要查询其节点数目的子树的根节点的关键字: \n')
    if value.lower() == 'q':
        print('\n____ 结束查询 ____')
        break
    value = int(value)  # 转化为整数
    p = BST.find_position(value)
    if not p:
        print(0)
    else:
        print(p.subtreenodes())
```

```
请输入一个正整数 n:
13

请输入 n 个正整数，以空格隔开，代表存储在节点的关键字:
10 4 11 2 6 12 13 1 3 5 8 7 9
```

前序遍历:

```
10 4 2 1 # # 3 # # 6 5 # # 8 7 # # 9 # # 11 # 12 # 13 # #
```

中序遍历:

```
# 1 # 2 # 3 # 4 # 5 # 6 # 7 # 8 # 9 # 10 # 11 # 12 # 13 #
```

后序遍历:

```
# # 1 # # 3 2 # # 5 # # 7 # # 9 8 6 4 # # # # # 13 12 11 10
```

____ 进入查询状态 ____

请输入您想要查询其节点数目的子树的根节点的关键字:

```
6
5
```

请输入您想要查询其节点数目的子树的根节点的关键字:

```
2
3
```

请输入您想要查询其节点数目的子树的根节点的关键字:

```
10
13
```

请输入您想要查询其节点数目的子树的根节点的关键字:

```
15
0
```

请输入您想要查询其节点数目的子树的根节点的关键字:

```
Q
```

____ 结束查询 ____