Algorithms and Data Structure 7

Yiping Deng

April 9, 2018

Problem 1

a)

The following is the concrete implementation in Python. Python is by-default generic, since it is not statically typed language. Also, note that in the class, there is no recursive call. Every operation is $\Theta(1)$.
Note: The python code includes a small unit test. You can simply run

```
$ python -m doctest -v stack.py
```

```python
class StackNode:
    """
    A basic linked list implementation
    """
    def __init__(self, data = None, nxt = None):
        self.data = data
        self.next = nxt

class Stack:
    """
    Implementation of Stack According to the code in the assignment
    >>> s = Stack(2)
    >>> s.isEmpty()
    True
    >>> s.push(5)
    >>> s.isEmpty()
    False
    >>> s.current_size
    1
    >>> s.push(4)
    >>> s.current_size
    2
    >>> s.pop()
    4
    >>> s.pop()
    5
    >>> s.isEmpty()
    True
    """
    def __init__(self, size = None):
        # there is no recursive call so constant time
        if size is None:
            self.size = -1
        elif size <= 0:
            raise Exception('Invalid size')
        else:
```

```
            self.size = size

        self.current_size = -1
        self.top = None

    def push(self, data):
        # constant time`
        # check the size
        if self.size != -1 and self.current_size == self.size:
            raise Exception('Stack Overflow Exception')

        if self.top is None:
            self.top = StackNode(data) #create a node
            self.current_size = 1 #set the proper size
        else:
            self.top = StackNode(data, self.top) #simply append the linked list
            self.current_size = self.current_size + 1 # increment

    def pop(self):
        # constant time
        if self.top is None:
            raise Exception('Stack Underflow Exception')

        val = self.top.data
        self.top = self.top.next # move to next one
        self.current_size = self.current_size - 1
        return val

    def isEmpty(self):
        # constant time
        return self.top is None
```

b)

We can implement a queue via two stacks. The first stack store the input, or freshly enqueued elements. The second stack store the output, or the elements that is ready is dequeue. When we try to dequeue, and if the second stack is empty, dump the first stack into the second one(poping from stack 1 and push into stack 2). The implementation in Python is also included here.
Note: The python code includes a small unit test. You can simply run

```
$ python -m doctest -v queue.py
```

```
from stack import Stack

class Queue:
    """
    A simple Queue using two stack
    >>> q = Queue()
    >>> q.enqueue(2)
    >>> q.enqueue(3)
    >>> q.enqueue(4)
    >>> q.dequeue()
    2
    >>> q.enqueue(5)
```

```
    >>> q.dequeue()
    3
    >>> q.dequeue()
    4
    >>> q.dequeue()
    5
    """
    def __init__(self):
        # initalize two Stack
        self.s1 = Stack()
        self.s2 = Stack()

    def enqueue(self, data):
        # just put it into stack 2
        self.s1.push(data)

    def prepare_dequeue(self):
        # dump stack 1 into stack 2
        if self.s2.isEmpty():
            while not self.s1.isEmpty():
                self.s2.push(self.s1.pop())

    def dequeue(self):
        self.prepare_dequeue() #prepare
        if self.s2.isEmpty():
            raise Exception('Queue Underflow Exception')
        return self.s2.pop()
```

Problem 2

a)

To reverse a linked list in-place(or in-situ), we have to abandon our regular functional recursive approach. Instead, we just have to reverse the direction of the pointer. The following Python code is a example of how to reverse a linked list in place. Since we only uses 3 reference, "next", "prev", and "current". The space complexity is obviously constant. Also, since there is only one "while" loop iterate through the linked list, and every operation within the "while" takes constant time, so the time complexity is linear(proportional to the length of the linked list). Thus, the time complexity is $O(n)$.
Note: The python code includes a small unit test. You can simply run

```
$ python -m doctest -v linkedlist.py
```

```
class Node:
    # a basic implementation of linked list node in python
    def __init__(self, data = None, nxt = None):
        self.data = data
        self.next = nxt

def reverse(head):
    """
    test the reverse function
    >>> head = Node(1, Node(2, Node(3)))
    >>> rv = reverse(head)
    >>> rv.data
```

```
    3
    >>> rv = rv.next
    >>> rv.data
    2
    >>> rv = rv.next
    >>> rv.data
    1
    >>> rv.next is None
    True
    """
    # just call the helper
    prev = None
    current = head
    nxt = None
    while current is not None:
        nxt = current.next
        current.next = prev
        prev = current
        current = nxt
    return prev
```

b)

Such question is just in-order tranversal of BST. We can write it recursively. Since it will tranverse every element in the tree only once, the total running time is $\Theta(n)$
Note: The python code includes a small unit test. You can simply run

```
$ python -m doctest -v binarytree_to_list.py
```

```
class TreeNode:
    def __init__(self, data, left = None, right = None):
        self.left = left
        self.right = right
        self.data = data
```

```
from linkedlist import Node
from binarytree import TreeNode

def treeToList(root):
    """
    This function will convert a tree into linkedlist
    >>> tree = TreeNode(5, TreeNode(3, TreeNode(2), TreeNode(4)), TreeNode(9))
    >>> lst = treeToList(tree)
    >>> lst.data
    2
    >>> lst = lst.next
    >>> lst.data
    3
    >>> lst = lst.next
    >>> lst.data
    4
```

```
    >>> lst = lst.next
    >>> lst.data
    5
    >>> lst = lst.next
    >>> lst.data
    9
    """
    # basic reversed in-order traversal of BST
    # first right node, then current, then left
    rst = None # the result

    def tranverse(node):
        nonlocal rst #capture the variable rst
        if node is None:
            return
        tranverse(node.right)
        rst = Node(node.data, rst) #append the result
        tranverse(node.left)

    tranverse(root)
    return rst
```

c)

We can just build a binary search tree that has no right node. Simply append smaller elements in order on the left, and we will have a binary search tree.
The running time of such operation is $\Theta(n)$ since we will iterate through every element only once.
Note: The python code includes a small unit test. You can simply run

```
$ python -m doctest -v list_to_binarytree.py
```

```
from binarytree import TreeNode
from linkedlist import Node

def listToTree(head):
    # We just simply append everything to the left node
    """
    Convert A sorted linkedlist into binary tree
    >>> lst = Node(1, Node(2, Node(3, Node(4))))
    >>> tree = listToTree(lst)
    >>> tmp = []
    >>> while tree is not None:
    ...     tmp.append(tree.data)
    ...     tree = tree.left
    >>> tmp
    [4, 3, 2, 1]
    """
    rst = None
    while head is not None:
        rst = TreeNode(head.data, rst, None) # append to the left
        head = head.next
    return rst
```