Assinment

March 15, 2023

#Part 1: DSA Problem 1: Stack Implement a stack data structure in Python. The stack should support the following operations:

push(item) - Add an item to the top of the stack.

pop() - Remove and return the item on the top of the stack.

peek() - Return the item on the top of the stack without removing it.

is_empty() - Return True if the stack is empty, else False.

[1]:
```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self):
        self.head = None
        self.size = 0

    def push(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head = new_node
        self.size += 1

    def pop(self):
        if self.head is None:
            raise Exception("Stack is empty")
        else:
            popped_node = self.head
            self.head = popped_node.next
            self.size -= 1
            return popped_node.value

    def peek(self):
        if self.head is None:
            raise Exception("Stack is empty")
        else:
            return self.head.value

    def is_empty(self):
        return self.size == 0

    def get_size(self):
        return self.size
```

the Node class represents each item in the queue, and the Queue class manages the queue by maintaining references to the head and tail nodes of the linked list and the size of the queue. The enqueue method adds a new node to the back of the queue, the dequeue method removes and returns the front node from the queue, the peek method returns the value of the front node without removing it, the is_empty method checks if the queue is empty, and the get_size method returns the size of the queue.

[2]:
```
my_stack = Stack()
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
```

[3]:
```
print(my_stack.pop())
print(my_stack.peek())
print(my_stack.get_size())
```

```
30
20
2
```

#Problem 2: Queue Implement a queue data structure in Python. The queue should support the following operations: enqueue(item) - Add an item to the back of the queue. dequeue() - Remove and return the item at the front of the queue. peek() - Return the item at the front of the queue without removing it. is_empty() - Return True if the queue is empty, else False.

[4]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def enqueue(self, value):
        new_node = Node(value)
        if self.tail is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    def dequeue(self):
        if self.head is None:
            raise Exception("Queue is empty")
        else:
            dequeued_node = self.head
            self.head = dequeued_node.next
            if self.head is None:
                self.tail = None
            self.size -= 1
            return dequeued_node.value

    def peek(self):
        if self.head is None:
            raise Exception("Queue is empty")
        else:
            return self.head.value

    def is_empty(self):
        return self.size == 0

    def get_size(self):
        return self.size
```

[5]:
```
my_queue = Queue()
my_queue.enqueue(100)
my_queue.enqueue(200)
my_queue.enqueue(300)
print(my_queue.dequeue())
print(my_queue.peek())
print(my_queue.get_size())
```

```
100
200
2
```

[ ]:
```

```

Problem 3: Binary Search Tree Implement a binary search tree (BST) data structure in Python.

The BST should support the following operations: insert(item) - Insert an item into the tree. delete(item) - Remove an item from the tree. search(item) - Return True if the item is in the tree, else False. size() - Return the number of nodes in the tree.

[6]:
```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None
        self.size = 0

    def insert(self, value):
        new_node = Node(value)
        if self.root is None:
            self.root = new_node
        else:
            current_node = self.root
            while True:
                if value < current_node.value:
                    if current_node.left is None:
                        current_node.left = new_node
                        break
                    else:
                        current_node = current_node.left
                else:
                    if current_node.right is None:
                        current_node.right = new_node
                        break
                    else:
                        current_node = current_node.right
        self.size += 1

    def search(self, value):
        current_node = self.root
        while current_node is not None:
            if value == current_node.value:
                return True
            elif value < current_node.value:
                current_node = current_node.left
            else:
                current_node = current_node.right
        return False
    def delete(self, value):
        def find_min_node(node):
            current_node = node
            while current_node.left is not None:
                current_node = current_node.left
            return current_node

        def delete_node(node, value):
            if node is None:
                return node
            if value < node.value:
                node.left = delete_node(node.left, value)
            elif value > node.value:
                node.right = delete_node(node.right, value)
            else:
                if node.left is None:
                    temp_node = node.right
                    node = None
                    return temp_node
                elif node.right is None:
                    temp_node = node.left
                    node = None
                    return temp_node
                temp_node = find_min_node(node.right)
                node.value = temp_node.value
                node.right = delete_node(node.right, temp_node.value)
            return node

        self.root = delete_node(self.root, value)
        if self.root is not None:
            self.size -= 1

    def get_size(self):
        return self.size
```

the Node class represents each node in the binary search tree, and the BST class manages the tree by maintaining a reference to the root node and the size of the tree. The insert method adds a new node to the tree based on the BST property, the search method searches for a node with the given value in the tree, the delete method removes a node with the given value from the tree while maintaining the BST property, and the get_size method returns the size of the tree.

[7]:
```
my_bst = BST()
my_bst.insert(4)
my_bst.insert(2)
my_bst.insert(1)
my_bst.insert(3)
my_bst.insert(6)
my_bst.insert(5)
my_bst.insert(7)
print(my_bst.search(5))
print(my_bst.get_size())
my_bst.delete(6)
print(my_bst.get_size())
```

```
True
7
6
```

[ ]:

Part 2: Python Problem 1: Anagram Checker Write a Python function that takes in two strings and returns True if they are anagrams of each other, else False. An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

[8]:
```
def is_anagram(str1, str2):
    # Convert the strings to lowercase and remove whitespace
    str1 = str1.lower().replace(" ", "")
    str2 = str2.lower().replace(" ", "")

    # Compare the sorted versions of the strings
    return sorted(str1) == sorted(str2)
```

the is_anagram function takes two strings as input, converts them to lowercase and removes whitespace, and then compares their sorted versions using the sorted function. If the sorted versions are the same, then the function returns True, indicating that the two strings are anagrams of each other. Otherwise, the function returns False.

[9]:
```
print(is_anagram("ear", "are"))
print(is_anagram("Ram", "arm"))
print(is_anagram("space", "paces"))
print(is_anagram("list", "silent"))
```

```
True
True
True
False
```

[ ]:

Problem 2: FizzBuzz Write a Python function that takes in an integer n and prints the numbers from 1 to n. For multiples of 3, print "Fizz" instead of the number. For multiples of 5, print "Buzz" instead of the number. For multiples of both 3 and 5, print "FizzBuzz" instead of the number.

Solution :- for this problem we can use a loop to iterate over the numbers from 1 to n, and use if-else statements to determine whether to print the number, "Fizz", "Buzz", or "FizzBuzz". Here is an example implementation:

```
[10]: def fizzbuzz(n):
          for i in range(1, n+1):
              if i % 3 == 0 and i % 5 == 0:
                  print("FizzBuzz")
              elif i % 3 == 0:
                  print("Fizz")
              elif i % 5 == 0:
                  print("Buzz")
              else:
                  print(i)
```

```
[11]: fizzbuzz(12)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
```

```
[12]: fizzbuzz(25)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
```

```
19
Buzz
Fizz
22
23
Fizz
Buzz
```

[ ]:

```

```

Problem 3: Fibonacci Sequence Write a Python function that takes in an integer n and returns the nth number in the Fibonacci sequence. The Fibonacci sequence is a series of numbers in which each number after the first two is the sum of the two preceding ones.

[13]:

```
#we can use a recursive function to calculate the nth number in the sequence.␣
 ↪Here is an example implementation:

def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

the fibonacci function takes an integer n as input and uses a recursive function to calculate the nth number in the Fibonacci sequence. If n is less than or equal to 1, then the function returns n. Otherwise, the function recursively calculates the nth number by adding together the (n-1)th and (n-2)th numbers in the sequence.

[14]:

```
print(fibonacci(0))
print(fibonacci(6))
print(fibonacci(7))
print(fibonacci(16))
```

```
0
8
13
987
```

[ ]:

```

```