FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 7

Semester 1, 2019

Objectives of this tutorial

- To understand Reverse Polish notation.
- To understand how stacks work and how can they be used in practical problems.
- To understand binary search

Exercise 1 *

- A mathematical expression is provided in a string, which may contain opening and closing parenthesis. Write a python function to determine if the parenthesis are balanced. **Hint:** This is easy if you use a Stack. The ADT of a Stack is:
  - `Stack(capacity)`: creates and returns a stack with given capacity.
  - `push(item)`: places an item at the top of the stack
  - `pop()`: removes and return the item at the top of the stack, if there is one.
  - `is_empty()`: returns true if and only if the stack is empty.
- Extend your function to include checks for balanced strings including also curly and square brackets.

Solution

```
from my_stack import Stack


def is_matched(expression):
        left_bracket = "({["
        right_bracket = ")}]"
        stack = Stack(len(expression))
        for character in expression:
                if character in left_bracket:
                        stack.push(character)
                elif character in right_bracket:
                        #  it wont be matched if the stack is empty
                        if stack.is_empty():
                                return False
                        # it wont be matched if the character
                        # I pop is not the equivalent on the left
                        if right_bracket.index(character) != \
                        left_bracket.index(stack.pop()):
                                return False
        return stack.is_empty()


def main():
        expression = input("Enter␣expression:␣")
        if is_matched(expression):
                print("Correct␣expression")
        else:
                print("Incorrect␣expression")


if __name__ == "__main__":
        main()
```

Exercise 2 *

- Consider the code below:

```
n = int(input("Enter a positive integer number: "))

while n > 1:
   n = n//2 # integer division
   print(n)
```

  What does it output for $n = 16$? What does it output for a $n = 2^k, k > 0$? For an arbitrary positive integer n, what is the $O()$ complexity of this code?

- Assume the class SortedList is an array implementation of the Sorted List ADT, as given in lectures. Write a method *index(self, item)* for SortedList which has a worst time complexity of *O(log(N))*, where N is the length of the list. The method *index* finds the first index of *item* in the list, and raises a *valueError* if the item is not in the list.

Solution

1. For $n = 16$, the code outputs 8, 4, 2, 1. For $n = 2^k$, it outputs $2^{k-1}, \ldots, 1$. That's $\log_2(n)$ terms. For each such printed term, a constant amount of operations is required, hence the complexity is $O(\log(n))$ for $n = 2^k$. For an arbitrary n, each printed term would be smaller than if we had inputted $n' = 2^k$ such that $n'/2 \leq n \leq n'$. Hence after at most $k = \log_2(n')$ iterations, we would reach 1, so the algorithm runs in $O(\log(n'))$. Since $\log_2(n') - 1 \leq \log_2(n) \leq \log_2(n')$, it is also correct to say that it runs in $O(\log(n))$.

2. We can use *binary search*:

```
def index(self, item):
        low = 0
        high = len(self)-1

        while low <= high:
                mid = (low + high)//2

                if item == self.the_array[mid]: # found item
                        if low == high: # found first item
                                return low
                        high = mid
                elif item < self.the_array[mid]:
                        high = mid - 1
                else
                        low = mid + 1

        raise ValueError(str(item) + " not in the list")
```

   The runtime analysis is similar to the one above: at every iteration, we divide the size of the list by two, and perform a constant amount of operations. Hence the runtime is $O(\log(n))$, where n is the size of the list.

Exercise 3 *

Consider a `Stack` ADT that implements a stack of strings using some data structure (you do not need to know which one) and defines the usual methods, where n is the size of the stack:

`Stack(n)`
`pop()`
`push(item)`
`size()`
`is_empty()`

Consider a Queue ADT that implements a queue of strings using some data structure (you do not need to know which one) and defines the usual methods, where n is the size of the queue:

```
Queue(n)
serve()
append(item)
size()
is_empty()
```

Use stack and queue operations to define the function

`reverse(my_queue)`

which takes a queue of strings called `my_queue`, returns a new one containing all non-empty strings from `my_queue` in reverse order, and does this by using a stack. Note that, at the end of the method, `my_queue` must contain the same elements as when it started, and in the same order (i.e., if you need to modify `my_queue`, make sure you leave it as it was).

For example, if `my_queue` has the following 5 elements :

```
"Hello", "Goodbye", "Not now", "", "Later"
```

where "Hello" is the item at the front, then the method will return the following queue, which has 4 elements with "Later" at the front:

```
"Later", "Not now", "Goodbye","Hello"
```

Solution

```
def reverse(my_queue):
        my_stack = Stack(my_queue.size())     # used to reverse
        result_q = Queue(my_queue.size())     # used for computing the result

        while not my_queue.is_empty():
                elem = my_queue.serve()
                my_stack.push(elem)
                result_q.append(elem)

        while not my_stack.is_empty():
                my_queue.append(result_q.serve())
                item = my_stack.pop()
                if item:                 # empty string is False in boolean context
                        result_q.append(item)

        return result_q
```

Exercise 4 *

Study the implementation below, which uses an array to implement a Queue. As opposed to the linear queue covered in the lectures, this implementation does not waste space.

```python
class CircularQueue:
        def __init__(self, size):
                assert size > 0, "Size must be positive"
                self.array = [None] * size
                self.reset()

        def reset(self):
                self.front = 0
                self.rear = 0
                self.count = 0

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

        def is_full(self):
                return self.count >= len(self.array)

        def serve(self):
                assert self.count > 0, "Empty queue"
                item = self.array[self.front]
                self.front = (self.front + 1) % len(self.array)
                self.count -= 1
                return item

        def append(self, item):
                assert not self.is_full(), "Full queue"
                self.array[self.rear] = item
                self.rear = (self.rear + 1) % len(self.array)
                self.count += 1
```

Write a Python method, *print_reverse_queue(self)*, for the class `CircularQueue`, which prints all the items in the queue from rear to front (without changing the queue).

Solution

```python
def print_reverse_queue(self):
        idx = self.rear
        for _ in range(self.count):
                print(self.the_array[idx])
                idx = (idx - 1) % len(self.the_array)
```