Lab 3: Stacks

Your Exercise: Implement a stack using linked lists.

Our in-class implementation of the `Stack` class uses the Python `list` data type as the underlying data structure. Instead, replace the use of the Python `list` data type with write your own `Node` class, which we used to build a singly linked-list.

Your Solution

Use a variety of code, Markdown (text) cells below to create your solution. Nice outputs would be timing results, and even plots. You will be graded not only on correctness, but the clarity of your code, descriptive text and other output. Keep it succinct.

GitHub: abhijit-baruah

In [1]:

```
# start by creating the Node class that sets data in a node and creates an empty next node
class Node:

    def __init__(self,data):
        self.data = data
        self.next = None
```

In [2]:

```
class Stack():

    # we initialize a default NULL head
    def __init__(self):
        self.head = None

    # check whether stack is empty
    def is_empty(self):
        if self.head == None:
            return True
        else:
            return False

    # peeking the stack
    def peek(self):
        if self.head == None:
            return None
        else:
            return self.head.data

    # method to add data to the stack
    def push(self, item):
        # if this is the first item in the stack
        if self.head == None:
            self.head = Node(item)
        else:
            # create a new_node and set the item into its data field
            new_node = Node(item)
            # shift head node to this new node
            new_node.next = self.head
            self.head = new_node

    # method to pop data from stack
    def pop(self):
        # if stack is non-empty
        if self.head:
            # pick the to-be-popped node and call it pop_node
            pop_node = self.head
            # shift the head node before popping to the preceeding head node
            self.head = self.head.next
            # reset the pop_node for subsequent popping
            pop_node.next = None
            return pop_node.data

        # if stack is empty
        else:
            return None

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count += 1
            current = current.next
        return count

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

```
    def __bool__(self):
        return self.size() != 0

    def __repr__(self):
        return "Stack()"

    def __str__(self):
        if self.head != None:
            current = self.head
            v = "[ Stack: " + str(current.data)
            current = current.next
            while current != None:
                v += ' --> ' + str(current.data)
                current = current.next
            v += " ]"
            return v
        else:
            return "[Stack is empty]"

    def __contains__(self, item):
        current = self.head
        found = False
        while current != None and not found:
            if current.data == item:
                found = True
            else:
                current = current.next
        return found
```

Testing

Balanced Symbol Checker

When finished, test your updated `Stack` class by running the **balanced symbol checker** from the textbook.

First we check the basic functionality of this new `Stack` class

In [3]:

```
s = Stack()
s_empty = Stack()
```

In [4]:

```
s # checks __repr__
```

Out[4]:

```
Stack()
```

In [5]:

```
s.push(1)
s.push(2)
s.push(3)
s.push(4)
s.push(5)
```

In [6]:

```
print(s) # checks __str__
```

```
[ Stack: 5 --> 4 --> 3 --> 2 --> 1 ]
```

In [7]:

```
s_empty
```

Out[7]:

```
Stack()
```

In [8]:

```
print(s_empty)
```

```
[Stack is empty]
```

In [9]:

```
# check is_empty
print(f'Is stack:s empty? {s.is_empty()}')
print(f'Is stack:s_empty empty? {s_empty.is_empty()}')
```

```
Is stack:s empty? False
Is stack:s_empty empty? True
```

In [10]:

```
# check peek
print(f'The top element of the stack is: {s.peek()}')
```

```
The top element of the stack is: 5
```

In [11]:

```
# pop/delete top element of stack
s.pop()
print(f'Stack after deleting one top item is s: {s}')
```

```
Stack after deleting one top item is s: [ Stack: 4 --> 3 --> 2 --> 1 ]
```

In [12]:

```
# pop next top item
s.pop()
```

Out[12]:

```
4
```

In [13]:

```
# check size after popping next item
print(f'Size of the stack is {s.size()}')
```

```
Size of the stack is 3
```

In [14]:

```
len(s) # check __len__
```

Out[14]:

```
3
```

In [15]:

```
# check __contains__
print(f'Is 2 in stack:s ? {2 in s}')
print(f'Is 8 in stack:s ? {8 in s}')
```

```
Is 2 in stack:s ? True
Is 8 in stack:s ? False
```

In [16]:

```
bool(s)
```

Out[16]:

```
True
```

In [17]:

```
bool(s_empty)
```

Out[17]:

```
False
```

Checking with Balanced Symbol Checker

In [18]:

```
def matches(_open, close):
    openers = "([{<"
    closers = ")]}>"
    return openers.index(_open) == closers.index(close)
```

In [33]:

```
def par_checker(symbol_string):
    s = Stack()
    balanced = True
    index = 0

    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]

        if symbol in "([{<":
            s.push(symbol)
        else:  # we have a ")"
            if s.is_empty():
                # There is no left parens in the stack to match this right one
                balanced = False
            else:
                # Pop out a right parens to reduce stack size
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False

        index += 1

    return balanced and s.is_empty()
```

In [34]:

```
# check using a balanced stack of symbols
s1 = '{{([][])}()}'
print(f'Is {s1} balanced ? {par_checker(s1)}')
```

```
Is {{([][])}()} balanced ? True
```

In [35]:

```
# check using an unbalanced stack of symbols
s2 = '[{()]'
print(f'Is {s2} balanced ? {par_checker(s2)}')
```

```
Is [{()] balanced ? False
```

So, the two examples mentioned above show the importance of stacks and also that the new `Stack()` class that we created using `Node` works correctly.

In [ ]:

```

```