Data Sturctures and Algorithms

Assignment 2

Introduction:

In this notebook I will implement a singly linked list and stack ADT in Python. I will also provide a brief description of the Stack ADT and Queue ADT, list their key operations and support operations, and provide two real-world examples of each. The purpose of this assignement is to give us an understanding of fundamental data structures and their implementations.

Part 1:

1. Linked List:

A **linked list** is a sequence of data structures, which are connected via links. Linked lists are a very commonly used data structure.

Singly linked lists contain nodes which have a *data* field and a *next* field. The *next* field points to the next node in the list of nodes.

Time Complexity:

Search: O(n)
Insert: O(1)
Delete: O(1)

In [63]:

```
class Node:                         # the node class
    def __init__(self, data):
        self.data = data            # holds the node's data
        self.next = None            # points to the nest node in the list
```

In [64]:

```
class myLinkedList:                 # the singly linked list class
    def __init__(self):
        self.head = None            # the head/front item in the list

    def list_traversal(self):
        if self.head is None:
            print("List has no elements.")
            return
        else:
            n = self.head           # assigns the head of the list to the variable n
            while n is not None:
                print(n.data, " ") # print the value of n.data
                n = n.next          # assign the next item in the list to n

    def add_first(self, data):
        new_node = Node(data)       # create a node with the new date and assigns the no
de to the new_node variable
        new_node.next = self.head  # assigns the current head of the list to new_node.n
ext
        self.head = new_node        # make new_node the new head of the list

    def add_last(self, data):
        new_node = Node(data)
        if self.head is None:       # check if list is empty
            self.head = new_node    # if so, make new_node the head of the list
            return
        n = self.head               # assign head of list to variable n
        while n.next is not None:  # while there is still another item in the list
            n = n.next              # assign the next item in the list to n
        n.next = new_node           # assign new_node to the next of the last item in th
e list

    def remove_first(self):
        if self.head is None:       # check if list is empty
            print("The list has no elements.") # inform user of empty list
        self.head = self.head.next # make the second item in the list the new head of t
he list
```

In [65]:

```
newLinkedList = myLinkedList()      # Initialise singly linked list newLinkedList
```

In [66]:

```
newLinkedList.add_last(5)           # add items 5, 10, 15 to list
newLinkedList.add_last(10)
newLinkedList.add_last(15)
```

In [67]:

```
newLinkedList.list_traversal()      # traverse the list and print values
```

```
5
10
15
```

In [68]:

```
newLinkedList.add_first(20)        # add 20 at start of list
```

In [69]:

```
newLinkedList.list_traversal()     # traverse the list and print values
```

```
20
5
10
15
```

In [70]:

```
newLinkedList.remove_first()       # remove the first item (20) from the list
```

In [71]:

```
newLinkedList.list_traversal()     # traverse the list and print values
```

```
5
10
15
```

2.

Stack:

A stack is a commonly used Abstract Data Type in programming languages. It gets its name from its bahaviour. It behaves like a real world stack, for example, a stack of cards. Like a stack of cards, the stack ADT can only be accessed at one end. It is only possible to access the element at the top of the stack. The stack allows us to remove or add elements at the top position only. It acts as a LIFO (Last-In-First-Out) structure. The element that is inserted last is accessed or removed first. Inserting an item into the stack is called "pushing" and removing the last item added to the stack is called "popping".

Key Operations:

push(x) - Inserts object x onto the top of the stack.
Input: Object
Output: None

pop() - Removes the top object and returns it. If the stack is empty an error occurs.
Input: None
Output: Object

Support Operations:

size() - Returns the size of the stack, i.e. the number of objects in the stack.
Input: None
Output: Integer

is_empty() - Returns a boolean which indicates whether the stack is empty or not.
Input: None
Output: Boolean

top() or **peek()** - Returns the top object of the stack. It does not remove the object from the stack. An error occurs if the stack is empty.
Input: None
Output: Object

Real World Data Examples:

1.

An undo button. Take for example the undo function on Microsoft Word. This acts like a stack ADT. Each letter, indentation, deletion, edit that the user makes is added to the top of the stack. When the user clicks the undo button the most recent object is removed from the top of the stack. If the user clicks it again, then the next most recent object will be removed from the top of the stack.

2.

Processing of nested structures. A nested structure is one that can contain instances of itself embedded within itself. For example, algebraic expressions can be nested because a subexpression of an algebraic expression can be another algebraic expression. Stacks can be used to process nested structures such as checking for balanced parentheses and evaluation of postfix expressions.

Time Complexity:

Push operation: O(1)
Pop operation: O(1)
Top operation: O(1)
Search operation: O(n)

Queue:

A queue is an abstract data type that is somewhat similar to a stack. A fundamental difference is that a queue can be accessed at both ends. It acts like a real life queue, for example, a queue for a shop checkout. One end of the queue is used for the insertion of data (enqueue) and the other end is used for the removal of data (dequeue). The queue acts as a FIFO (First-In-First-Out) structure. The element that is inserted into the queue first will also be accessed first. Elements can be inserted into the queue at any time, but only the element which has been in the queue the longest may be removed.

Key Operations:

enqueue(x) - Inserts object x at the rear of the queue.
Input: Object
Output: None

dequeue() - Removes the object that is at the front of the queue and returns it. If the queue is empty an error occurs.
Input: None
Output: Object

Support Operations:

size() - Returns the size of the queue, i.e. the number of objects in the queue.
Input: None
Output: Integer

is_empty() - Returns a boolean which indicates whether the stack is empty or not.
Input: None
Output: Boolean

front() - Returns the front object in the queue. It does not remove the object from the queue. An error occurs if the queue is empty.
Input: None
Output: Object

Real World Data Examples:

1.

Access to a shared resource. A printer is an example of a shared resource. Jobs are scheduled in the order that they arrive to the printer. A new job is added to the end of the queue. When the printer is ready to print the next job it takes this job from the front of the queue, i.e. the job that has been in the queue for the longest period of time.

2.

Online ticket sales. Online ticket sellers can make use of the queue ADT. This is to ensure that those who enter the queue first have access first to the tickets. The user is placed at the end of the queue and must wait until he/she reaches the front of the queue.

Time Complexity:

Enqueue operation: O(1)
Dequeue operation: O(1)

Part 2:

1. Stack ADT

In [72]:

```
class Node:                             # the node class
    def __init__(self, data):
        self.data = data                # holds the node's data
        self.next = None                # points to the next node in the stack
```

In [73]:

```
class Stack:                            # the Stack class
    def __init__ (self):
        self.head = None                # the head/front object in the stack

    def is_empty(self):
        if self.head == None:           # check if there is no head in the stack, i.e. an
empty stack
            return True
        else:
            return False                # return false if stack is not empty

    def push(self, data):
        if self.head == None:           # if stack is empty:
            self.head = Node(data)      # Create a node with data value and assign it to
 head of stack
        else:
            newNode = Node(data)        # else: create a node with data value and assign
 to newNode
            newNode.next = self.head    # assign the current head of the stack to the new
Node.next
            self.head = newNode         # make newNode the new head of the stack

    def pop(self):
        if self.is_empty():             # check if stack is empty
            return None                 # if so, return none
        else:
            poppedNode = self.head      # else: assign head of stack to variable poppedNo
de
            self.head = self.head.next # assign the next object in the stack to be the
 head of the stack
            poppedNode.next = None      # set poppedNode's next to None
            return poppedNode.data      # return the data held in the popped node

    def top(self):
        if self.is_empty():             # check if stack is empty
            return None                 # if so, return None
        else:
            return self.head.data       # else return the data at the head of the stack

    def size(self):
        temp = self.head                # assign the head of the stack to the temp variab
le
        count = 0                       # initiate a counter

        while (temp):                   # while there is a temp variable
            count += 1                  # increment count by 1
            temp = temp.next            # assign the next object in the stack to temp var
iable
        return count                    # return count, the number of objects in the stac
k
```

2.

In [74]:

```
S = Stack()           # Initialise the Stack - S
```

In [75]:

```
S.push(5)              # Adds 5 to top of the stack. Returns None.
```

In [76]:

```
S.push(3)              # Adds 3 to top of the stack. Returns None.
```

In [77]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (3).
```

Out[77]:

3

In [78]:

```
S.push(2)              # Adds 2 to top of the stack. Returns None.
```

In [79]:

```
S.push(8)              # Adds 8 to top of the stack. Returns None.
```

In [80]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (8).
```

Out[80]:

8

In [81]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (2).
```

Out[81]:

2

In [82]:

```
S.push(9)              # Adds 9 to top of the stack. Returns None.
```

In [83]:

```
S.push(1)              # Adds 1 to top of the stack. Returns None.
```

In [84]:

```
S.pop()                # Removes the top element from the stack and returns its value
 (1).
```

Out[84]:

1

In [85]:

```
S.push(7)              # Adds 7 to top of the stack. Returns None.
```

In [86]:

```
S.push(6)              # Adds 6 to top of the stack. Returns None.
```

In [87]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (6).
```

Out[87]:

6

In [88]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (7).
```

Out[88]:

7

In [89]:

```
S.push(4)              # Adds 4 to top of the stack. Returns None.
```

In [90]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (4).
```

Out[90]:

4

In [91]:

```
S.pop()                # Removes the top object from the stack and returns its value
 (9).
```

Out[91]:

9

In [92]:

```
S.top()                # Returns the top object in the stack (5), i.e. the most recently
added.
                       # It does not remove the object from the stack.
```

Out[92]:

5

In [93]:

```
S.is_empty()          # Returns a boolean which indicates whether the stack is empty or
not.
                      # In this case the stack is not empty so it returns False.
```

Out[93]:

```
False
```

In [94]:

```
S.size()              # Returns the size of the stack, i.e. the number of objects in th
e stack which is 1 in this case.
```

Out[94]:

```
1
```

Another example of the Stack class in operation:

In [95]:

```
NewStack = Stack()    # Initialise the Stack - NewStack
```

In [96]:

```
NewStack.is_empty()   # Check if stack is empty. Returns boolean True.
```

Out[96]:

```
True
```

In [97]:

```
NewStack.size()       # Returns the size of the stack (0).
```

Out[97]:

```
0
```

In [98]:

```
NewStack.push("Bottom object in stack.")  # Adds string to top of the stack. Returns No
ne.
```

In [99]:

```
NewStack.is_empty()   # Check if stack is empty. Returns boolean False.
```

Out[99]:

```
False
```

In [100]:

```
NewStack.size()       # Returns the size of the stack (1).
```

Out[100]:

```
1
```

In [101]:

```
NewStack.push("Middle object in stack.")  # Adds string to top of the stack. Returns No
ne.
```

In [102]:

```
NewStack.size()       # Returns the size of the stack (2).
```

Out[102]:

```
2
```

In [103]:

```
NewStack.push("Top object in stack.")     # Adds string to top of the stack. Returns No
ne.
```

In [104]:

```
NewStack.size()       # Returns the size of the stack (3).
```

Out[104]:

```
3
```

In [105]:

```
NewStack.top()        # Returns the top object in the stack ("Top object in stack").
                      # It does not remove the object from the stack.
```

Out[105]:

```
'Top object in stack.'
```

In [106]:

```
NewStack.pop()        # Removes the top object from the stack and returns its value ("T
op object in stack")
```

Out[106]:

```
'Top object in stack.'
```

In [107]:

```
NewStack.size()       # Returns the size of the stack (2).
```

Out[107]:

```
2
```

In [108]:

```
NewStack.top()        # Returns the top object in the stack ("Middle object in stack")
                      # It does not remove the object from the stack.
```

Out[108]:

```
'Middle object in stack.'
```

In [109]:

```
NewStack.pop()        # Removes the top object from the stack and returns its value ("M
iddle object in stack")
```

Out[109]:

```
'Middle object in stack.'
```

In [110]:

```
NewStack.size()        # Returns the size of the stack (1).
```

Out[110]:

```
1
```

In [111]:

```
NewStack.top()         # Returns the top object in the stack ("Bottom object in stac
k").
                       # It does not remove the object from the stack.
```

Out[111]:

```
'Bottom object in stack.'
```

In [112]:

```
NewStack.pop()         # Removes the top object from the stack and returns its value
 ("Bottom object in stack")
```

Out[112]:

```
'Bottom object in stack.'
```

In [113]:

```
NewStack.size()         # Returns the size of the stack (0).
```

Out[113]:

```
0
```

3.

Suppose an initially empty stack S has executed a total of 35 push operations, 15 top operations, and 10 pop operations, 3 of which raised Empty errors that were caught and ignored. What is the current size of S?

In the above scenario, the current size of S will be **28**.
In order for 3 of the pop operations to raise errors the stack S must have been empty 3 out of the 10 times that pop was attempted.
The 15 top operations have no effect on the contents of S.
The 35 push operations added 35 items to the stack and the 7 successful pop operations removed 7 items.
Therefore, 35 - 7 = **28**.

In [ ]: