Stack, Queue and Linked List

1. Abstract Data Type and Data Structure

Abstract Data Type (ADT) is a specification of what operations it can support.

- It specifies interactions/operations.
- No code. It is not the actual implementation.
- There are often more than one way to implement an ADT.

Data Structure (DS) is the actual representation of the data and the algorithms to manipulate the data elements.

- Concrete implementation fo a ADT.
- One allowable opeation in ADT = one function implemented in DS

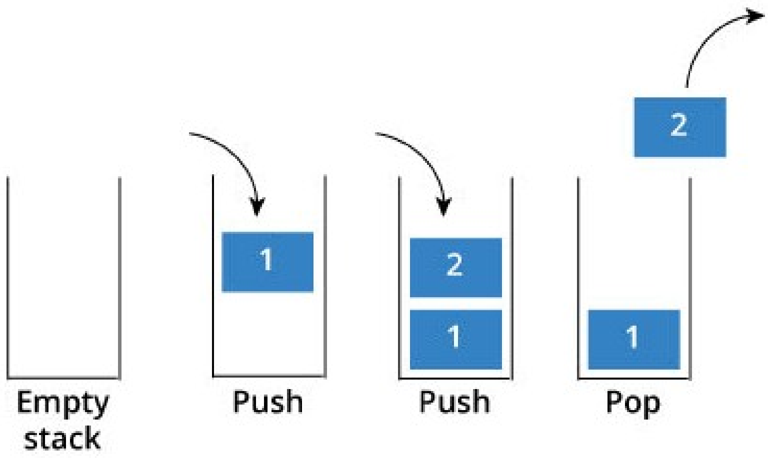
Advantages: Users only need to understand the allowable operations of an ADT before using it. No knowledge of actual implmentation is required.

The common ADTs are **Stack**, **Queue**, **Linked List** and **Binary Tree**.

2. Stack

Stack is an ADT which stores items in order in which they are added.

- Items can only be added to and removed from the **top of the stack**.
- The order is also called **Last-In-First-Out (LIFO)**.



https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd

Example Applications

- Detect missing symbols, e.g. missing opening or closing bracket.
- Reverse a sequence.

Operations

The basic operations of a stack is to add and remove item from its top.

- **push()**: Add item to the stack
- **pop()**: Remove item from the stack

Other supporting functions to be added are:

- **is_empty()**: Is stack empty?
- **size()**: How many items are in the stack?
- **peek()**: What is the next item to be removed?

Exercise 1

Define a `Stack` class which implements the operations of a Stack:

- Initialize an empty list `_items` in its initializer method.
- Implement `push()` and `pop()` functions with basic operations of a stack.

In [1]:

```
class Stack:

    def __init__(self):
        self._items = []

    def push(self, item):
        self._items.append(item)

    def pop(self):
        if self._items:
            return self._items.pop()
        else:
            return None
```

Test:

In [2]:

```
stack = Stack()
stack.push('apple')
stack.push('banana')
print(stack._items)
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

```
['apple', 'banana']
banana
apple
None
```

Exercise 2

Define a `Stack2` class which inherits from `Stack` class.

- Code the supplementry functions `size()` , `is_empty()` , `peek()`

In [3]:

```python
class Stack2(Stack):

    def size(self):
        return len(self._items)

    def is_empty(self):
        return len(self._items) == 0

    def peek(self):
        return self._items[-1] if self._items else None
```

Test:

In [4]:

```python
stack = Stack2()
stack.push('apple')
stack.push('banana')

print(stack.size())
stack.pop()
print(stack.peek())
stack.pop()

print(stack.is_empty())
print(stack.peek())
```

```
2
apple
True
None
```

Exercise 3

Implement a function `check_matching_symbols()` , which checks whether a string contains matching openning and closing symbols for `([{` and `}])` .

- It return `True` if all symbols in string are balanced, else it returns `False` .
- Use `Stack2` class for the implementation.

In [5]:

```
def check_matching_symbols(st):
    symbol_pairs = {'(':')', '[':']', '{':'}'}
    stack = Stack2()

    for ch in st:
        if ch in symbol_pairs.keys():
            stack.push(ch)
        elif ch in symbol_pairs.values():
            if stack.is_empty():
                return False
            item = stack.pop()
            if(ch != symbol_pairs[item]):
                return False

    return stack.is_empty()
```

Test:

In [6]:

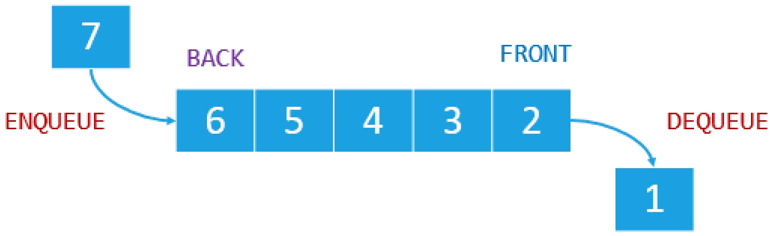
```
print(check_matching_symbols(')]}'))
print(check_matching_symbols('([{'))
print(check_matching_symbols('(ab[c]d{e(f)})'))
print(check_matching_symbols('(ab[c]d{ef)})'))
```

```
False
False
True
False
```

3. Queue

Queue holds an item in order which they are added.

- Items are added to the end and removed from the front.
- This order is also called First-In-First-Out (FIFO).



https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd

Example Applications

- Customer service queue
- Printing jobs at a printer

Operations

The basic operations of a queue is to add and remove item from queue.

- **enqueue()**: Add item to the queue
- **dequeue()**: Remove item from the queue

Other supporting functions to be added are:

- **is_empty()**: Is stack empty?
- **size()**: How many items are in the queue?
- **peek()**: What is the next item to be removed?

Exercise 1

Define a class `Queue` which implements basic operations of a queue.

- Initialize an empty list in its initializer.
- Code the `enqueue()` and `dequeue()` functions.

In [7]:

```
class Queue:

    def __init__(self):
        self._items = []

    def enqueue(self, item):
        self._items.insert(0, item)

    def dequeue(self):
        return self._items.pop() if self._items else None
```

Test:

In [8]:

```
q = Queue()
q.enqueue('apple')
q.enqueue('banana')
print(q._items)
print(q.dequeue())
print(q.dequeue())
print(q.dequeue())
```

```
['banana', 'apple']
apple
banana
None
```

Exercise 2

Define a class `Queue2` which inherits from class `Queue` .

- Code the supplementary functions of a queue, i.e. `size()` , `is_empty()` , `peek()` .

In [9]:

```
class Queue2(Queue):

    def size(self):
        return len(self._items)

    def is_empty(self):
        return len(self._items) == 0

    def peek(self):
        return self._items[-1] if self._items else None

```

In [10]:

```
q = Queue2()
q.enqueue('apple')
q.enqueue('banana')
print(q.size())
q.dequeue()
print(q.peek())
print(q.dequeue())
print(q.is_empty())
```

```
2
banana
banana
True
```

Exercise 3

Implement a **priority queue** where job with higher weight will be processed first. We need to code two classes `Job` and `PriorityQueue` .

The `Job` class has only one instance attribute, `weight` .

- Implement its `__str__()` method which returns its weight in string format.

The `PriorityQueue` class inherits from `Queue2` class by overriding its `enqueue()` method.

- The new `enqueue()` method inserts item at appropriate position so that items are maintained in ascending order by weight.

In [11]:

```
class Job:
    def __init__(self, weight):
        self.weight = weight

    def __str__(self):
        return str(self.weight)
```

In [12]:

```
class PriorityQueue(Queue2):

    def enqueue(self, item):
        for i, x in enumerate(self._items):
            if x.weight >= item.weight:
                self._items.insert(i, item)
                return

        # if queue is empty or only 1 item
        self._items.append(item)
```

In [13]:

```
import random

q = PriorityQueue()
nums = list(range(10))
random.shuffle(nums)

for i in nums:
    print(i, [str(i) for i in q._items])
    j = Job(i)
    q.enqueue(j)

while not q.is_empty():
    q.dequeue()
```

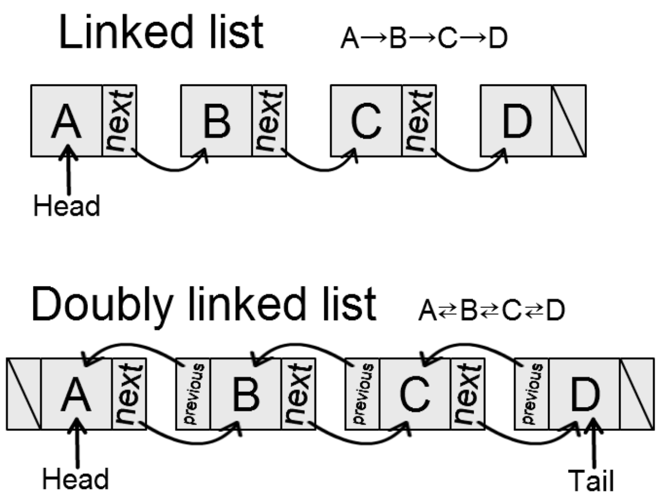
```
6 []
5 ['6']
4 ['5', '6']
0 ['4', '5', '6']
2 ['0', '4', '5', '6']
3 ['0', '2', '4', '5', '6']
1 ['0', '2', '3', '4', '5', '6']
9 ['0', '1', '2', '3', '4', '5', '6']
7 ['0', '1', '2', '3', '4', '5', '6', '9']
8 ['0', '1', '2', '3', '4', '5', '6', '7', '9']
```

4. Linked List

A **linked list** is a linear data structure which holds a collection of elements, called **Node**. These nodes may not be <u>not stored at contiguous memory locations</u>.

- Nodes can be accessed in a sequential way.
- Linked list doesnot provide random access to a node.
- Usually each node is **linked** to next node and/or previous node by storing their memory locations.

When the Nodes are connected with only the `next` pointer the list is called **Singly Linked List** and when it's connected by the `next` and `previous` pointers, the list is called **Doubly Linked List**.



https://medium.com/@lucasmagnum/sidenotes-linked-list-abstract-data-type-and-data-structure-fd2f8276ab53

Common Operations

Here are some of the operations

- **prepend()**: Add a node in the beginning
- **pop_first()**: Remove a node from the beginning
- **append()**: Add a node in the end
- **pop()**: Remove a node from the end
- **remove()**: Remove Node, which matches a value, from the list

Node

Linked list stores data in a collection of nodes. Each node contains a **data** and **pointer(s)** pointing to other node(s).

For **Singly Linked List**, node contains one pointer `next` pointing to next node.

- With only `next` pointer, it can only traverse forward along the link.

For **Doubly Linked List**, node contains two pointers `next` and `previous` pointing to next and previous node respectively.

- With both `next` and `previous` pointers, you can traverse forward and backward along the link.

Exercise 1: Node

Implement a class `Node` for Singly Linked List.

- It has an instance attribute `data` which holds data of the node, and another instance attribute `next` pointing to next node.
- Both instance attributes are initialized by input parameters in initializer method.
- It implements `__repr__()` method which returns string `Node(data->next.data)` , e.g. `Node(A->B)` if the value for current and next nodes are `A` and `B` respectively.

In [14]:

```
class Node:

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

    def __repr__(self):
        return "{}({}->{})".format(
            self.__class__.__name__,
            self.data,
            self.next.data if self.next else None)
```

In [15]:

```
node2 = Node('bcd')
node1 = Node('abc', node2)
print(node1)
```

```
Node(abc->bcd)
```

Excercise 2: Singly Linked List

A Singly Linked List contains an attribute `head` which points first node of the linked list.

Implement a Singly Linked List with following methods:

- Initializer method which initializes `head` to `None` since the initial linked list is empty.
- `is_empty()` method which returns True if linked list is empty
- `size()` method returns number of nodes in the list
- `contains()` method which return True if an item is found in the linked list

In [16]:

```
class SinglyLinkedList:

    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None

    def size(self):
        count = 0
        current = self.head
        while current:
            count = count + 1
            current = current.next
        return count

    def contains(self, data):
        current = self.head
        while current:
            if current.data == data:
                return True
            current = current.next

        return False
```

Test:

In [17]:

```
sll = SinglyLinkedList()
sll.head = Node('A')
sll.head.next = Node('B')
print(sll.is_empty())
print(sll.size())
print(sll.contains('B'))
print(sll.contains('C'))
```

```
False
2
True
False
```

Excercise 3: Singly Linked List

A Singly Linked List typically contains following methods.

- **prepend()**: Add a node in the beginning
- **pop_front()**: Remove a node from the beginning
- **remove()**: Remove Node, which matches a value, from the list

The `remove()` method will return `True` if a matching value is found in the linked list, else it will `return` False. The implementation needs to take care 4 scenarios:

- When the linked list is empty, i.e `head` is pointing to None
- When the item to be removed is the head node

- When the item to be removed is in any other node
- When the item to be removed is not found

Lets implement above methods in class `SinglyLinkedList2` which inherites from `SinglyLinkedList` class.

In [18]:

```
class SinglyLinkedList2(SinglyLinkedList):

    def prepend(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    def pop_front(self):
        node = self.head
        self.head = node.next if node else None
        return node

    def remove(self, data):
        if self.head is None:
            return False

        if self.head.data == data:
            self.head = self.head.next
            return True

        previous = self.head
        current = previous.next

        while current:
            if current.data == data:
                previous.next = current.next
                return True
            else:
                previous = current
                current = previous.next

        return False
```

Test:

In [19]:

```
sll = SinglyLinkedList2()
sll.prepend('D')
sll.prepend('C')
sll.prepend('B')
sll.prepend('A')
print(sll.pop_front())   # Remove first node 'A'
print(sll.remove('A'))   # Remove non-exists
print(sll.remove('B'))   # Remove head node
print(sll.remove('D'))   # Remove end node
print(sll.remove('C'))   # Remove last element
print(sll.pop_front())   # anymore node?
```

```
Node(A->B)
False
True
True
True
None
```

Excercise 4

A **Node** in a Doubly Linked List contains both `next` and `previous` attributes.

Lets implement a `NodeD` class which inherites from `Node` class.

- It overrides initilizer method add a `previous` attribute

In [20]:

```
class NodeD():
    def __init__(self, data=None, next_node=None, prev_node=None):
        self.data = data
        self.next = next_node
        self.previous = prev_node
```

In [21]:

```
class NodeD(Node):
    def __init__(self, data=None, next_node=None, prev_node=None):
        super().__init__(data, next_node)
        self.previous = prev_node
```

Test:

In [22]:

```
b = NodeD('B', NodeD('C'), NodeD('A'))
b.previous.next = b
b.next.previous = b
print(b.previous, b, b.next)
```

```
NodeD(A->B) NodeD(B->C) NodeD(C->None)
```

Exercise 5: Doubly Linked List

A Doubly Linked List contains an attribute `head` which points first node of the linked list, and another attribute `tail` which points to the last node.

Following methods are identical in Singly Linked List and Doubly Linked List. (You can copy the code from `SinglyLinkedList` class.)

- `is_empty()` method which returns True if linked list is empty
- `size()` method returns number of nodes in the list
- `contains()` method which return True if an item is found in the linked list

Implement a `DoublyLinkedList` class with following methods. Remember to use `Node2` class instead of `Node` class.

- Initializer method which initializes both `head` and `tail` to `None` since the initial linked list is empty.
- prepend(): Add a node in the beginning
- pop_first(): Remove a node from the beginning

In [23]:

```
class DoublyLinkedList:

    def __init__(self):
        self.head = None
        self.tail = None

    def prepend(self, data):
        node = NodeD(data)
        node.next = self.head
        if self.head:
            self.head.previous = node

        self.head = node

        # if this is the 1st node in list, head and tail pointing to same node
        if self.tail is None:
            self.tail = node

    def pop_first(self):
        if not self.head:
            return False
        node = self.head
        if node.next:
            node.next.previous = None
        self.head = node.next

        # if this is the last node to be removed
        if self.tail is node:
            self.tail = None
```

Test:

In [24]:

```
dll = DoublyLinkedList()
dll.prepend('A')
print(dll.head, dll.tail)
dll.prepend('B')
print(dll.head, dll.tail)
dll.pop_first()
print(dll.head, dll.tail)
dll.pop_first()
print(dll.head, dll.tail)
```

```
NodeD(A->None) NodeD(A->None)
NodeD(B->A) NodeD(A->None)
NodeD(A->None) NodeD(A->None)
None None
```

Reference

- https://www.geeksforgeeks.org/data-structures/linked-list/ (https://www.geeksforgeeks.org/data-structures/linked-list/)
- https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd (https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd)
- https://codeforwin.org/2015/09/singly-linked-list-data-structure-in-c.html (https://codeforwin.org/2015/09/singly-linked-list-data-structure-in-c.html)
- https://www.lynda.com/Python-tutorials/Python-Data-Structures-Stacks-Queues-Deques/779747-2.html (https://www.lynda.com/Python-tutorials/Python-Data-Structures-Stacks-Queues-Deques/779747-2.html)
- https://www.lynda.com/Python-tutorials/Python-Data-Structures-Linked-Lists/5007865-2.html (https://www.lynda.com/Python-tutorials/Python-Data-Structures-Linked-Lists/5007865-2.html)
- https://medium.com/@lucasmagnum/sidenotes-linked-list-abstract-data-type-and-data-structure-fd2f8276ab53 (https://medium.com/@lucasmagnum/sidenotes-linked-list-abstract-data-type-and-data-structure-fd2f8276ab53)