Credit:
Shaikh Nayeem Uddin

- Data Structure
  - Stack
    - What is Stack?
      - A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This ordering principle is sometimes called LIFO, last-in first-out.
    - Six Abstract Data Type of Stack.
      - Stack()
      - push(item)
      - pop()
      - peek()
      - is_empty()
      - size()
    - Implement Stack

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

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

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

    - Convert decimal to any base using Stack

```
import Stack

def base_converter(decimal_number, base):
    digits = "123456789ABCDEF"
    remainder_stack = Stack()

    while decimal_number > 0:
        remainder = decimal_number % base
        remainder_stack.push(remainder)
        decimal_number //= base

    new_string = ""
    while not remainder_stack.is_empty():
        new_string = new_string +
digits[remainder_stack.pop()]

    return new_string
```

Credit:
Shaikh Nayeem Uddin

- Postfix Evaluation

```
import Stack

def postfix_eval(postfix_expr):
    operand_stack = Stack()
    token_list = postfix_expr.split()

    for token in token_list:
        if token in "0123456789":
            operand_stack.push(int(token))
        else:
            operand2 = operand_stack.pop()
            operand1 = operand_stack.pop()
            result = do_math(token, operand1, operand2)
            operand_stack.push(result)
    return operand_stack.pop()


def do_math(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
```

- Queue
  - What is Queue?
    - Queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle.
  - Seven abstract data type of Queue?
    - Queue()
    - enqueue(item)
    - dequeue()
    - is_empty()
    - len()
    - iter()
    - clear()
  - Implement Queue

```
class Queue:
    def __init__(self):
        self._items = []

    def is_empty(self):
        return self._items == []

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

    def __iter__(self):
```

Credit:
Shaikh Nayeem Uddin

```
        for item in self._items:
            yield item

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

    def dequeue(self):
        return self._items.pop(0)

    def clear(self):
        self._items = []
```

- Deques
  - What is Deque?
    - It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.
  - Implement Deque

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def add_front(self, item):
        self.items.append(item)

    def add_rear(self, item):
        self.items.insert(0, item)

    def remove_front(self):
        return self.items.pop()

    def remove_read(self):
        return self.items.pop(0)

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

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

  - What is Priority Queue?
    - Priority queue is the queue in which items are assigned a priority and the items with a higher priority are dequeued first. However, all items with the same priority still obey the FIFO principle.

- Tree

Credit:
Shaikh Nayeem Uddin

Credit:
Shaikh Nayeem Uddin

- Basic question.
  - What is root?
    - The topmost node of the tree is known as the root node.
  - What is leaf nodes?
    - Nodes that have no children are known as leaf nodes.
  - What is binary tree?
    - A binary tree is a tree in which each node can have at most two children.
  - What is level of root node?
    - 0
  - What is depth of a node?
    - The depth of a node is distance from the root.
  - What is height of a tree? How to calculate height?
    - Height is number of levels in a tree.
    - Top level + 1 == height.
  - What is full binary tree?
    - A full binary tree is a binary tree in which each interior node contains two children.
  - What is perfect binary tree?
    - A perfect binary tree is a full binary tree in which all leaf nodes are at the same level.
- Tree Traversal sequence
  - Preorder Traversal
    - root.left.right
  - Inorder Traversal
    - left.root.right
  - Postorder Traversal
    - left.right.root
- Tree Traversal code
  - Preorder Traversal

```
def preorder(root):
 if root is not None:
        print(root.data)
        preorder(root.left)
        preorder(root.right)
```

  - Inorder Traversal

```
def preorder(root):
    if root is not None:
        print(root.left)
        preorder(root.data)
        preorder(root.reght)
```

  - Postorder Traversal

```
def preorder(root):
    if root is not None:
        print(root.left)
        preorder(root.right)
        preorder(root.data)
```

Credit:
Shaikh Nayeem Uddin

Credit:
Shaikh Nayeem Uddin

- Graph
  - Difference between BFS-DFS:

BfS	DfS
Breadth first Search.	Depth first Search.
Uses Queue data structure.	Uses Stack data structure.
BfS is slower.	DfS is faster.
It require more memory.	It require less memory.
Backtracking is not allowed.	Backtracking is allowed.
BfS is optimal for finding the shortest path.	DfS is not optimal for this.

  - What is Graph?
    - a graph is a set V of vertices and a set E of edges, such that each edge in E connects two of the vertices in V. The term node is also used here as a synonym for vertex.
  - What is undirected graph?
    - An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.
  - What is directed graph?
    - A directed graph or digraph is a graph in which all the edges point in a single direction.
  - What is Adjacency Matrix?
    - An adjacency matrix maintains an n×n matrix, for a graph with n vertices. Each entry is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the entry will be 0.
  - What is Adjacency List?
    - The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array Ai is a list, which contains all the vertices that are adjacent to vertex i.
  - Code for DFS.

```
def DFS(self, s):
    visited = [False for i in range(self.V)]
    stack = []
    stack.append(s)
    while (len(stack)):
        s = stack[-1]
        stack.pop()
        if (not visited[s]):
            print(s, end=" ")
            visited[s] = True
        for node in self.adj[s]:
            if (not visited[node]):
                stack.append(node)
```

Credit:
Shaikh Nayeem Uddin

Credit:
Shaikh Nayeem Uddin

- Code for BFS.

```
def BFS(self, s):
    visited = [False] * (max(self.graph) + 1)
    queue = []
    queue.append(s)
    visited[s] = True

    while queue:
        s =queue.pop(0)
        print(s, end = "")

        for i in self.graph(s):
            if not visited[i]:
                queue.append(i)
                visited = True
```

- Mid-terms question
  - Question 1
    - a) What is Data Structure? What is ADT and API?
      - A data structure is a way to store a non-constant amount of data, supporting a set of operations to interact with that data. The set of operations supported by a data structure is called an interface (ADT/API).
    - b) What is the difference between ADT/API and Data strucTures? Expain giving example of some ADT/APIs and data structures.
      - • A data structure is a way to store data, with algorithms that support operations on the data
      - • Collection of supported operations is called an interface also known as API (Application Programming Interface) or ADT (Abstract Data Type)
      - • Interface is a specification: what operations are supported (the problem!)
      - • Data structure is a representation: how operations are supported (the solution!)
      - • Example of interfaces: Sequence and Set
      - • Example of data structure: Linked List, Stack, Queue.
  - Question 2
    - a) for i in range(0, n, 1):
      - for j in range(0, n, 1):
        - for k in range(0, n, -1):
          - print(k)

      What would be the asymptotic time complexity of the program above in terms of n?

Credit:
Shaikh Nayeem Uddin

- Here first and second line is depending on the value of n, so asymptotic time complexity of first line in O(n), and asymptotic time complexity of second line is O(n^2). here third line will not be executed, because if the value of n is negative fist two line will not run, and if the value is positive third loop will not run, so the asymptotic time complexity of the program is O(n^2).

- By comparing the worst case asymptotic running time of Arrays (static) and Linked Lists show where Arrays have advantage and where Linked Lists have advantage. How can the advantage of Linked Lists be increased through data structure augmentation?
  - If we compare worst case asymptotic running time of Arrays and Linked List, Array is good for printing at a given index, and Linked list is good for adding or removing in the head, if we save the address of the last node in a variable of Linked List, then then advantage of Linked Lists be increased.

- Question 3
  - Why Static Arrays are bad for dynamic operations?
    - Static Array is bad for dynamic operation, because if the array is full then we have to allocate memory manually to add more elements, and we have to copy all elements in the new memory.
  -
- Question 4
  - See question from the question paper, I can't type that much.

```
def insertNode(start, value, pos):
    head = start
             data = value
             position = pos
             new_node = LList(data)
    itr = head
    counter = 1
    temp = None
    while counter != position - 1:
        counter += 1
        itr = itr.y
    temp = itr.y
    itr.y = new_node
    new_node.y = temp
    return head
```