Assignment 4 Solutions:

This document contains a complete solution for some questions and an added report for the questions which have been solved as codes in JuPyter Notebooks. It also has snapshots of the JuPyter notebooks, along with this document, the submission also contains attached JuPyter Notebooks and its .html files for easier viewing or checking codes as required.

Complete solutions for questions 2,3,5 are in this document since those don't need any code and are mathematical or logical proofs. Supplementary reports for questions 1,4 is also included in this document since those are code-based questions.

Question 1:

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

A little bit about the 2 data structures.

Queue : One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows FIFO (First-In-First-Out) methodology, i.e., the data item stored first will be accessed first.
Basic operations of queue : Basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

To enqueue a new element, the rear pointer is used and to dequeue the front pointer is used.

Stacks : It is a LIFO (Last-in First-out) data structure. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.
Basic Operations : A stack is used for the following two primary operations −

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- peek() − get the top data element of the stack, without removing it.
- isFull() − check if stack is full.
- isEmpty() − check if stack is empty.

So as it is clearly visible, the functionality of a stack and queue is not much different other than the fact that a queue is FIFO and a stack is LIFO.
To implement a stack using 2 queues, it can be done using 2 methods.
Method 1: Push implementation is done in O(n) and pop implementation is done in O(1)
Method 2: Push implementation is done in O(1) and pop implementation is done in O(n)

Our general idea is that while implementing any of the functions, we will use the other queue as a buffer memory to store data temporarily. At the start and end of the functions being called one queue will always be empty.

Method 1:
First dequeue original queue into second queue, add the new element to the original queue, it will automatically become the first element and then enqueue the original queue from the second queue.

By doing this, our newly PUSHed element is now always at the front of the queue (which happens for a stack) and while POPing elements, always dequeue from the original queue since it always has the latest added element at the front of the queue which ensures that our so-formed data structure is LIFO like a stack.

Method 2:
In this, while PUSHing an element, just enqueue it to the original queue. To POP an element, dequeue all but the last element from the original queue and enqueue it into the second queue. Then again dequeue from second queue and enqueue into the original queue thus making the latest added element as the front of the queue and when POP is called it returns this which makes this implementation a LIFO type.

Code for method 1:

In [1]:
```
from queue import Queue

class Stack:

    def __init__(self):

        # Two inbuilt queues
        self.q1 = Queue()
        self.q2 = Queue()

        self.current_size = 0    # To maintain current number of elements


    def push(self, x):
        self.current_size += 1   #Since an element is added, increase number of elements by 1

        # Push x first in empty q2
        self.q2.put(x)

        # Push all the remaining elements in q1 to q2.
        while (not self.q1.empty()):
            self.q2.put(self.q1.queue[0])
            self.q1.get()

        # swap the names of two queues since q1 is our original queue
        # and q2 is the one we are keeping as an empty buffer
        self.q = self.q1
        self.q1 = self.q2
        self.q2 = self.q

    def pop(self):

        # if no elements are there in q1
        if (self.q1.empty()):
            return "Cannot pop since the queue is empty"
        self.q1.get()
        self.current_size -= 1   #Since we are removing an element, decrease number of elements by 1

    def peek(self):     #Returns the value of element at the top
        if (self.q1.empty()):
            return "The queue is empty"
        return self.q1.queue[0]

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

In [2]:
```
s = Stack()
list1 = [1,4,2,8,5,7]

for x in list1:
    s.push(x)        #To make a stack of the elements in the order as above
                     #Current size should 6 and peek should return the last element (i.e 7) since it is LIFO

print("current size: ", s.size())
print("Element at top is:",s.peek())

s.pop()
s.pop()
                     #After popping twice, top element should be 8 and size should be 4
print("current size: ", s.size())
print("Element at top is:",s.peek())

s.push(9)
                     #After element 9 is pushed into stack, element at top should be 9 and size should be 5
print("current size: ", s.size())
print("Element at top is:",s.peek())
```

```
current size:  6
Element at top is: 7
current size:  4
Element at top is: 8
current size:  5
Element at top is: 9
```

Code for method 2:

In [1]:
```
from queue import Queue

class Stack:

    def __init__(self):

        # Two inbuilt queues
        self.q1 = Queue()
        self.q2 = Queue()

        self.current_size = 0    # To maintain current number of elements


    def push(self, x):
        self.current_size += 1   #Since an element is added, increase number of elements by 1

        # Simply add element to original queue
        self.q1.put(x)


    def pop(self):

        # if no elements are there in q1
        if (self.q1.empty()):
            return "Cannot pop since the queue is empty"

        # Push all the remaining elements in q1 to q2.
        while (self.q1.qsize() != 1):
            self.q2.put(self.q1.queue[0])
            self.q1.get()

        # swap the names of two queues since q1 is our original queue
        # and q2 is the one we are keeping as an empty buffer
        self.q = self.q1
        self.q1 = self.q2
        self.q2 = self.q


        self.current_size -= 1   #Since we are removing an element, decrease number of elements by 1

    def peek(self):     #Returns the value of element at the top
        if (self.q1.empty()):
            return "The queue is empty"
        return self.q1.queue[-1]

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

In [2]:
```
s = Stack()
list1 = [1,4,2,8,5,7]

for x in list1:
    s.push(x)        #To make a stack of the elements in the order as above
                     #Current size should 6 and peek should return the last element (i.e 7) since it is LIFO

print("current size: ", s.size())
print("Element at top is:",s.peek())

s.pop()
s.pop()
                     #After popping twice, top element should be 8 and size should be 4
print("current size: ", s.size())
print("Element at top is:",s.peek())

s.push(9)
                     #After element 9 is pushed into stack, element at top should be 9 and size should be 5
print("current size: ", s.size())
print("Element at top is:",s.peek())
```

```
current size:  6
Element at top is: 7
current size:  4
Element at top is: 8
current size:  5
Element at top is: 9
```

The 2 methods are very similar, the only difference is that in the first method we dequeue and enqueue to put the newly added element at the top during the PUSH function call and in the second method we dequeue and enqueue to put the newly added element at the top during the POP function call.

Question 2:

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h (k) = k mod 9.

Hashing : Hashing is a data structure that is used to store a large amount of data, which can be accessed in O(1) time by operations such as search, insert and delete. Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends on the efficiency of the hash function used.
In this case : hash function h(k) = k mod 9
Hashing should always generate values between 0 to m-1 where m is the size of the hash table.

While hashing, the hashing function may lead to a collision, that is two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. This approach does not reserve any space that will not be taken up, but has the disadvantage that in order to find a particular item, lists will have to be traversed. However, adding the hashing step still speeds up retrieval considerably.

In our case :

The keys are given as 5, 28, 19, 15, 20, 33, 12, 17, 10
So, the hash table becomes as follows : where hash function is given by key mod 9

Key(k)	Hash function(h(k))
5	5
28	1
19	1
15	6
20	2
33	6
12	3
17	8
10	1

As we can see very clearly, there are a few collisions :
$h(15) = h(33) = 6$,
$h(28) = h(19) = h(10) = 1$
These collisions are resolved by chaining (linked list approach) as I described above. This is how the final hash table looks like:

Index (h(k))	Key
0	
1	10 -> 19 -> 28
2	20
3	12
4	
5	5
6	15 -> 33
7	
8	17

With this chaining technique, as shown in the table above, elements can be accessed even after collision. To access a key : Access the value of A[h(k)]. If there is NULL or single value stored in it, then the case is trivial, simply the element can be accessed. If there's a linked list, we traverse the linked list to find the desired value.

Question 3:

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

We will use these 2 properties of a BST for proving these claims.

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.

CLAIM (A) : y must be an ancestor of x

We will use the method of contradiction for this purpose, let us assume that y is not an ancestor of x.

If so, let w denote the first common ancestor of x and y. By the property of BST stated above, $x < w < y$ so y cannot be the successor of x. So, y is an ancestor of x. This is a contradiction to our initial assumption; hence, the initial assumption is incorrect. Therefore, y must be an ancestor of x.

CLAIM (B) : The left child of y must be an ancestor of x
Again, we will use the method of contradiction, let us assume that the left child of y is not an ancestor of x. If so, then (since we have proved **Claim (A)**), the right child of y would be an ancestor of x, implying $x > y$.
Now, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let w denote this lowest ancestor. Then w must be in the left subtree of y, which implies $w < y$, contradicting the fact that y is the successor of x.

Hence, we have proven that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Question 4:

Describe a non-recursive algorithm for enumerating all permutations of the numbers {1, 2, ...,n} using an explicit stack.

Essentially what this means is we as a user should give all the push and pop commands to the stack explicitly, in general while implementing a code, call function and return act as push as pop respectively for stack implicitly but since here it is mentioned we should implement using explicit stack, we will convert a recursive function call to an iterative type using the explicit stack commands append for pushing and pop for popping memory from stack.

Code for the same is as follows:

In [1]:
```
def permutation_stack(n):     #Function to implement printing of permutation of numbers from 1 to n
                              #using iterative algorithm using explicit stack

    current_numbers = []      #Initialising list to be printed
    kth_bit = n*[0]           #Initialising kth_bit array, a 0 appearing in kth position corresponds to k not
                              #being in current_numbers array and 1 appearing corresponds to it being there.

    numbers_list = [x for x in range(1,n+1)]    #Making a list of numbers from 1 upto n

    count = 0                 #setting count to 0, will increase it by 1 everytime a permutation is printed
    stack = []                #Initialising explicit stack
    case = 0                  #Initialising case, since there are 2 cases in iteration

    while 1:                  #While loop to implement iteration

        if case == 0:         #Case when a recursive call was made in the previous iteration of while loop

            no_zeros = 1      #Value to check if we encountered no zeroes in kth_bit array

            for i in range(len(numbers_list)):     #Loop through kth_bit array

                if kth_bit[i] == 0:    #If we encounter a zero, push its index to stack and break the loop
                    no_zeros = 0
                    kth_bit[i] = 1
                    current_numbers.append(numbers_list[i])
                    stack.append(i)
                    break

            if no_zeros:              #If no zeros occurs ( i.e all entries have occured ) print the permutation
                print(current_numbers)
                count += 1
                case = 1              #Since we have returned value here, we need to start popping entries from
                                      #stack so set case to 1 in which we pop entries and do recursive type call
                                      #using iterative form of while loop


        else:                 #Case when a function was returned in the previous iteration of while loop

            index = stack.pop()     #The last index from where recursion call was made on that level in the
                                    #previous iteration

            current_numbers.pop()   #Pop the corresponding number from output array and set the kth_bit of that
                                    #index to 0
            kth_bit[index] = 0
```

```
            for i in range(index+1,len(numbers_list)):    #Iterate through kth_bit array looking for zeroes

                if kth_bit[i] == 0:        #If we encounter a zero, push its index to stack and break the loop
                    kth_bit[i] = 1
                    current_numbers.append(numbers_list[i])
                    stack.append(i)
                    case = 0               #Again as a recursive call is made here, we need to set case to 0
                    break

            if len(stack) == 0:
                break
    return count                           #We are already printing all permutations, return count as a double
                                           #check that all cases have been iterated through, value of this is n!
```

In [2]:
```
n = 4
count = permutation_stack(n)
print(count)
```

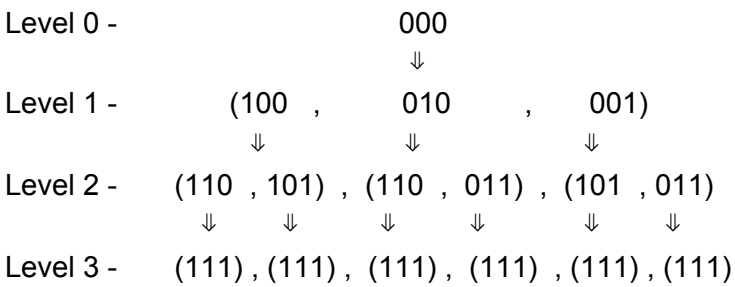
```
[1, 2, 3, 4]
[1, 2, 4, 3]
[1, 3, 2, 4]
[1, 3, 4, 2]
[1, 4, 2, 3]
[1, 4, 3, 2]
[2, 1, 3, 4]
[2, 1, 4, 3]
[2, 3, 1, 4]
[2, 3, 4, 1]
[2, 4, 1, 3]
[2, 4, 3, 1]
[3, 1, 2, 4]
[3, 1, 4, 2]
[3, 2, 1, 4]
[3, 2, 4, 1]
[3, 4, 1, 2]
[3, 4, 2, 1]
[4, 1, 2, 3]
[4, 1, 3, 2]
[4, 2, 1, 3]
[4, 2, 3, 1]
[4, 3, 1, 2]
[4, 3, 2, 1]
24
```

I have used 4 lists :

1. numbers_list[ ] - I have generated this list using n, it contains values from 1 to n
2. kth_bit[ ] - List which stores index of the numbers present in current_numbers[ ]
3. current_numbers[ ] - List of numbers which we are currently active, will print this once it is full with all n entries in any order (will essentially print it n! times each time with 1 permutation)
4. stack[ ] - The explicit stack or the equivalent call-stack.

The while loop is divided into two parts. The case where case is 0 is executed when in the previous iteration, a recursion call was made. And the case where case is 1 is executed when in the previous iteration, a function was returned.
We push the index whenever we make a recursion call onto the stack. While backtracking we pop the stack to find the index where we made a recursion call and start iterating the kth_bit[ ] list from the next index.

Since this iterative code using explicit stack is a representation of a recursive function, it will also have levels of iteration (similar to levels of recursion)
A visualization of the kth_bit for n = 3 is :



In output, the number corresponding to which index became 1 first will appear first, i.e outputs will be of the form - 123 , 132 , 213 , 231 , 312 , 321

Time complexity :
As illustrated above, for any n, nodes at kth level will be $\frac{n!}{(n-k)!}$ and for each node we are traversing through kth_bit once hence time complexity is O(n) for 1 node.
There are total of $\sum_{k=0}^{n} \frac{n!}{(n-k)!}$ for k going from 0 to n nodes.

$\Rightarrow Time\ complexity = O(n) \times \sum_{k=0}^{n} \frac{n!}{(n-k)!} \leq O(n) \times n! \times (\sum_{0}^{\infty} \frac{1}{k!})$

$\Rightarrow Time\ complexity = O(n) \times n! \times e^1$ ….using Taylor series of e^x at x = 1.

$\Rightarrow Time\ complexity = O(n \times n!)$

Space complexity:

Our code has n levels, and the maximum size of stack happens when we are the lowest level in the tree so its maximum size is the number of nodes between root and the lowest level i.e n.
Hence space complexity is of O(n).

Question 5:

Show that any n-node binary tree can be converted to any other n-node binary tree using O (n) rotations.

To transform an arbitrary n-node binary tree to any other n-node binary tree we can transform the n-node binary tree into a right-going chain and then transform the chain to any other n-node binary tree back.
Now, if we perform a single right rotation on the parent of any node, then that node would be added to the right-going chain as desired. No elements would be removed. We know that when we perform a right rotation, the length of the rightmost path increases by at least 1. So, if we start with the rightmost path of length 1, which is the worst case, we need to perform at most (n − 1) rotations to make it into a right-going chain.

Therefore, if we have a binary tree with n-nodes X given and wish to transform it into another n-node binary tree Y, it can be done in 2 steps as follows:

1. Convert X into a right-going chain takes r1 rotations, then r1 < n since there are a total of n nodes so number of rotations required < number of total nodes.
2. Perform inverse rotations of those rotations which convert Y into a right-going chain takes r2 rotations then r2 < n, again since there are a total of n nodes so number of rotations required < number of total nodes.

Hence, the number of rotations is r1 + r2 for converting a n-node Binary Tree to another n-node Binary Tree and r1 + r2 < 2n, which implies it can be done in O(n) rotations.