EE4371 - Assignment 4

EE19B094

Manvar Nisharg

This PDF contains the solutions to questions from assignment 4 and explanations to the source code submitted along.

Question 1

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

Answer 1

There are two ways to implement a stack using two queues.

Both methods have a cheap operation and a costly operation.

Method 1 : Costly $\text{pop}()$

In [ ]:
```
#Importing queue header files
from queue import Queue

#Class to implement stack using queue
class stack_using_queue:

        #Initilastion function for the class
        def __init__(self):
                self.Q1 = Queue()       #Queue number 1 (active queue)
                self.Q2 = Queue()       #Queue number 2

        #Function to push into the stack
        def push(self,value):
                self.Q1.put(value)      #Push number into the active queue

        #Function to pop from the stack
        def pop(self):

                #Return if stack is empty
                if self.Q1.empty():
                        return

                #Transfer all but last element from active queue (Q1) to the other queue (Q2)
                while self.Q1.qsize() != 1:
                        self.Q2.put(self.Q1.get())

                #Dequeue the remaining last element effectively poping the number from the stack
                dequeued = self.Q1.get()
                print("Popped : %d"%dequeued)

                #Swaping the active queue
                self.Q_temp = self.Q1
                self.Q1 = self.Q2
                self.Q2 = self.Q_temp

        #Function to get the top element of the stack
        def get_top(self):

                #Return if stack is empty
                if self.Q1.empty():
                        return

                #Transfer all but one elment from Q1 to Q2
                while self.Q1.qsize() != 1:
                        self.Q2.put(self.Q1.get())

                #Store the last element in a variable and pop it
                top = self.Q1.get()

                #Push it back into Q2
                self.Q2.put(top)

                #Swap Q1 and Q2
                self.Q_temp = self.Q1
                self.Q1 = self.Q2
                self.Q2 = self.Q_temp
                return top


#Testing
test = stack_using_queue()
test.push(1)
test.push(3)
test.push(2)
print(test.get_top())
test.pop()
print(test.get_top())
```

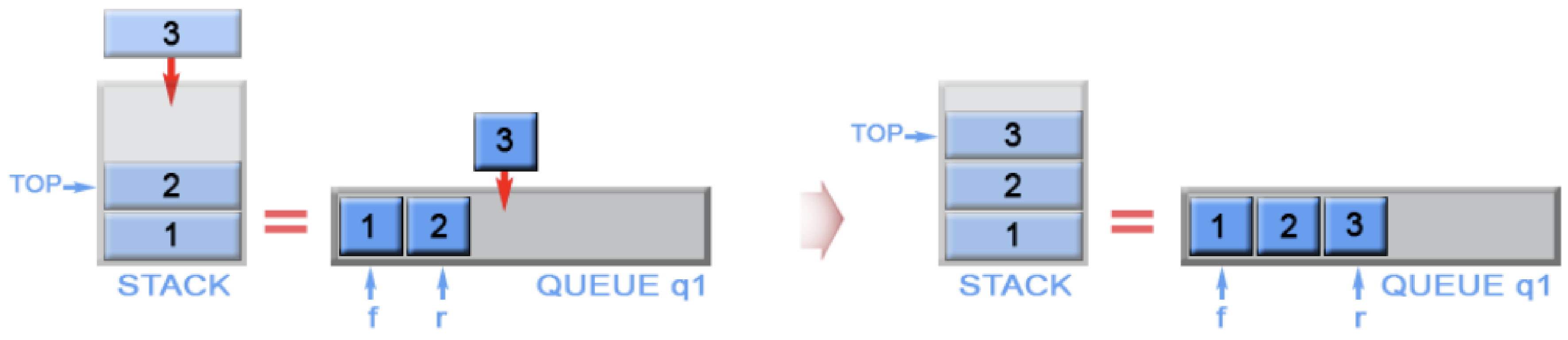
Output for the given code

-

```
3
Popped : 3
2
```

Pushing an element

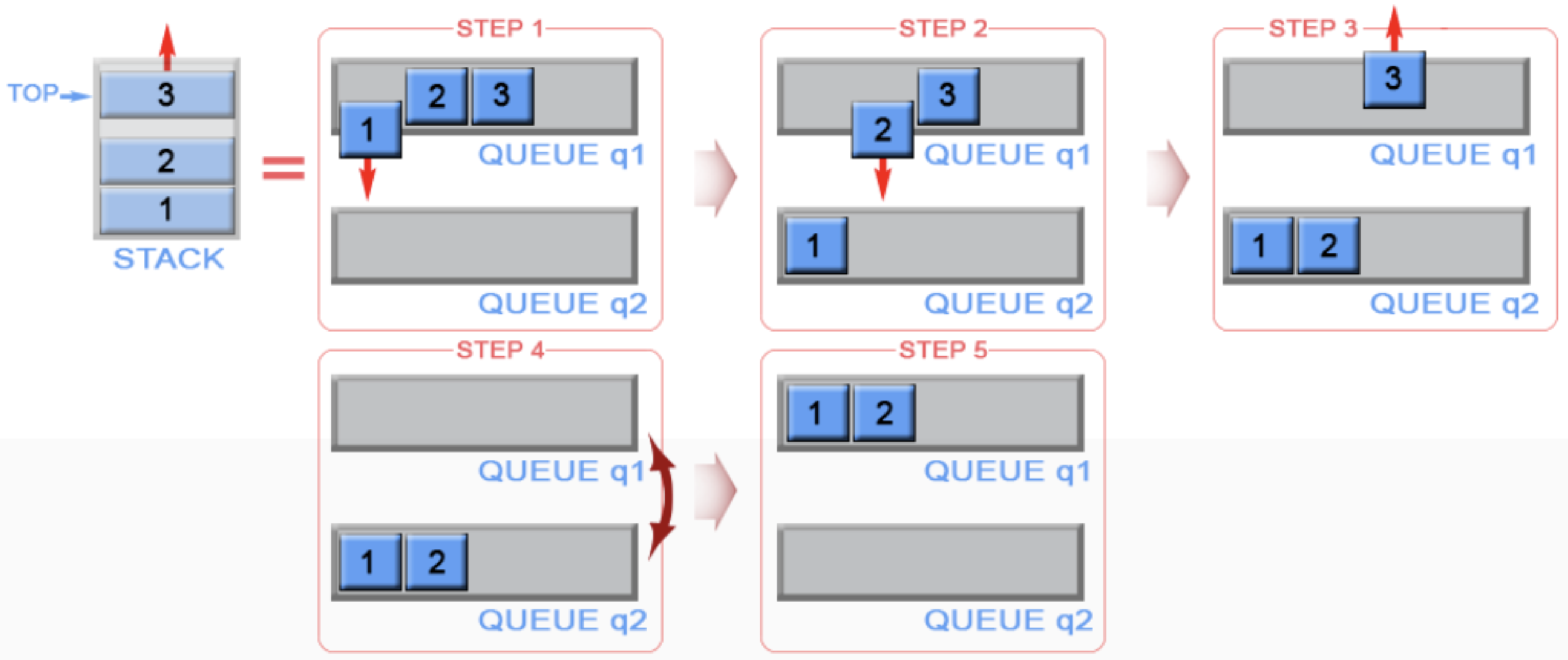
We simply push the element into Q1 (active queue)
Thus, this will take only $O(1)$ time.



Pushing element “3” to the stack

Popping an element

We first transfer all but the last element from Q1 to Q2, then dequeue the last element and then swap the Queues
Thus, clearly this will take $O(n)$ time.



Popping element “3” from the stack

Method 2 : Costly $\text{push}()$

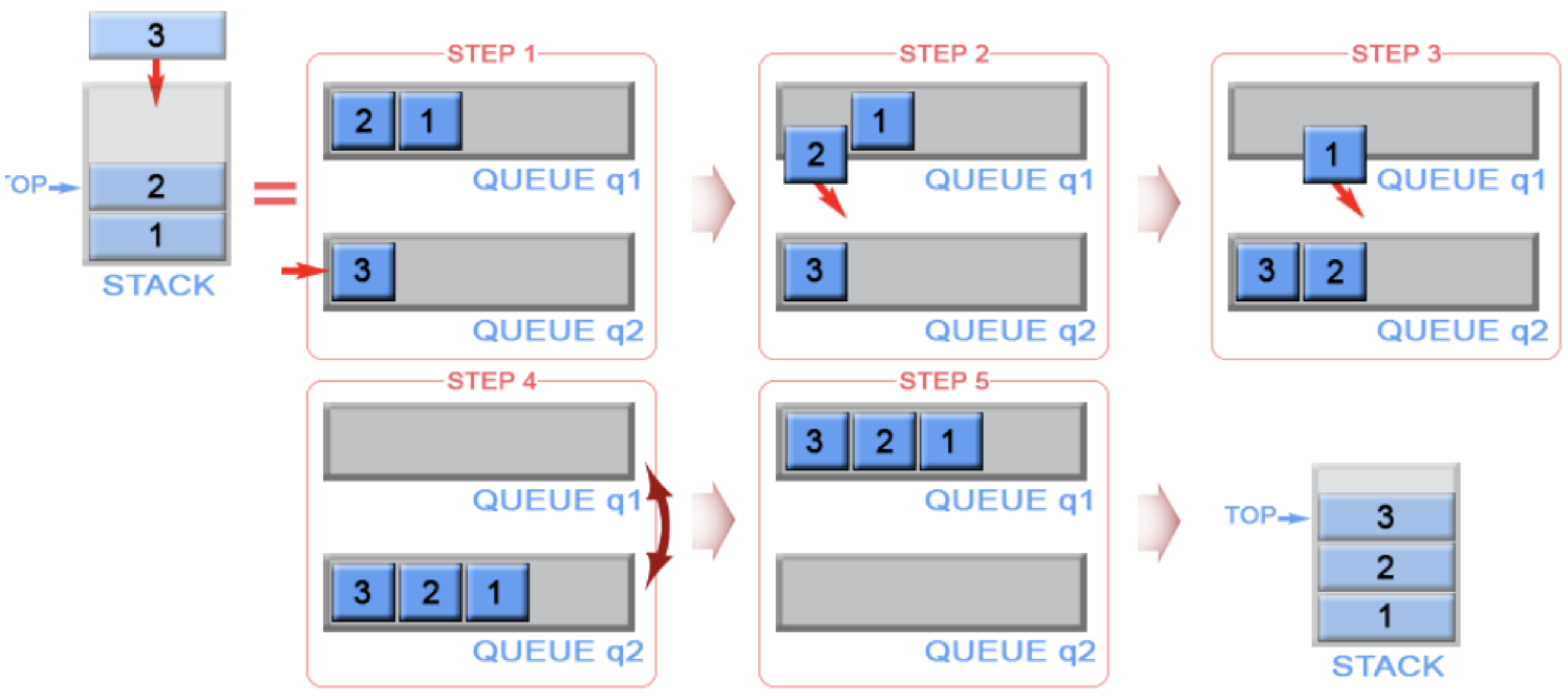
In [ ]:
```python
#Importing queue header files
from queue import Queue

#Class to implement stack using two queues
class stack_using_queue:

    #Initilastion function for the class
    def __init__(self):

        self.Q1 = Queue()   #Queue number 1 (active queue)
        self.Q2 = Queue()   #Queue number 2

    #Function to push into the stack
    def push(self,value):

        self.Q2.put(value)  # Push value first in empty Q2

        #Transfer all elements from active queue (Q1) to the other queue (Q2)
        while self.Q1.qsize():
            self.Q2.put(self.Q1.get())

        #Swap the active queue
        self.Q_temp = self.Q1
        self.Q1 = self.Q2
        self.Q2 = self.Q_temp

    def pop(self):

        #Return if stack is empty
        if (self.Q1.empty()):
            return

        #Dequeue from active queue (Q1)
        dequeued = self.Q1.get()
        print("Popped : %d"%dequeued)

    def get_top(self):

        #Return if stack is empty
        if (self.Q1.empty()):
            return -1

        #Else return the top element
        return self.Q1.queue[0]


#Testing
test = stack_using_queue()
test.push(1)
test.push(3)
test.push(2)
print(test.get_top())
test.pop()
print(test.get_top())
```

Output for the given code

-
```
3
Popped : 3
2
```

Pushing an element

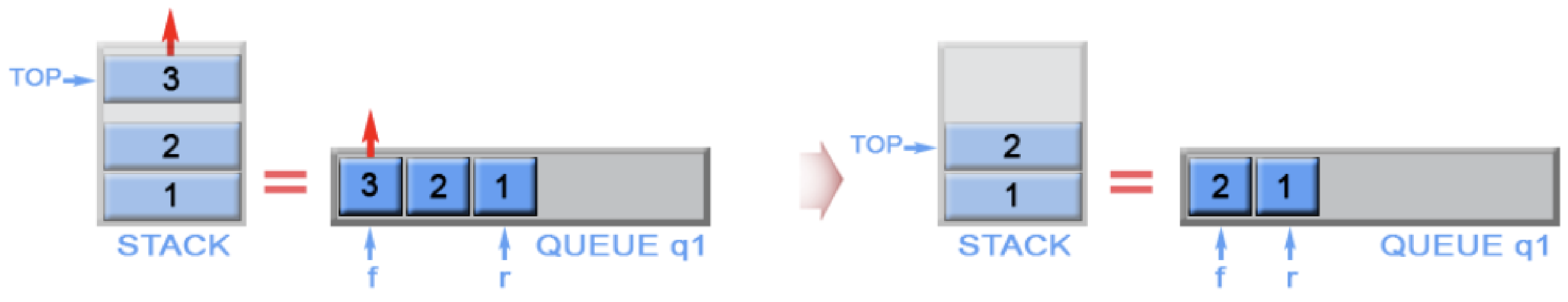
We first push the element into Q2, transfer the elements into Q1 to Q2, and lastly swap the queues. Thus, this will take $O(n)$ time.



Pushing element "3" to the stack

Popping an element

We simply dequeue from the active queue (Q1)
Thus, clearly this will take only $O(1)$ time.



Popping element "3" from the stack

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 $h(k) = k \, mod \, 9$.

Answer 2

Firstly, let us see what collisions and chaining are.

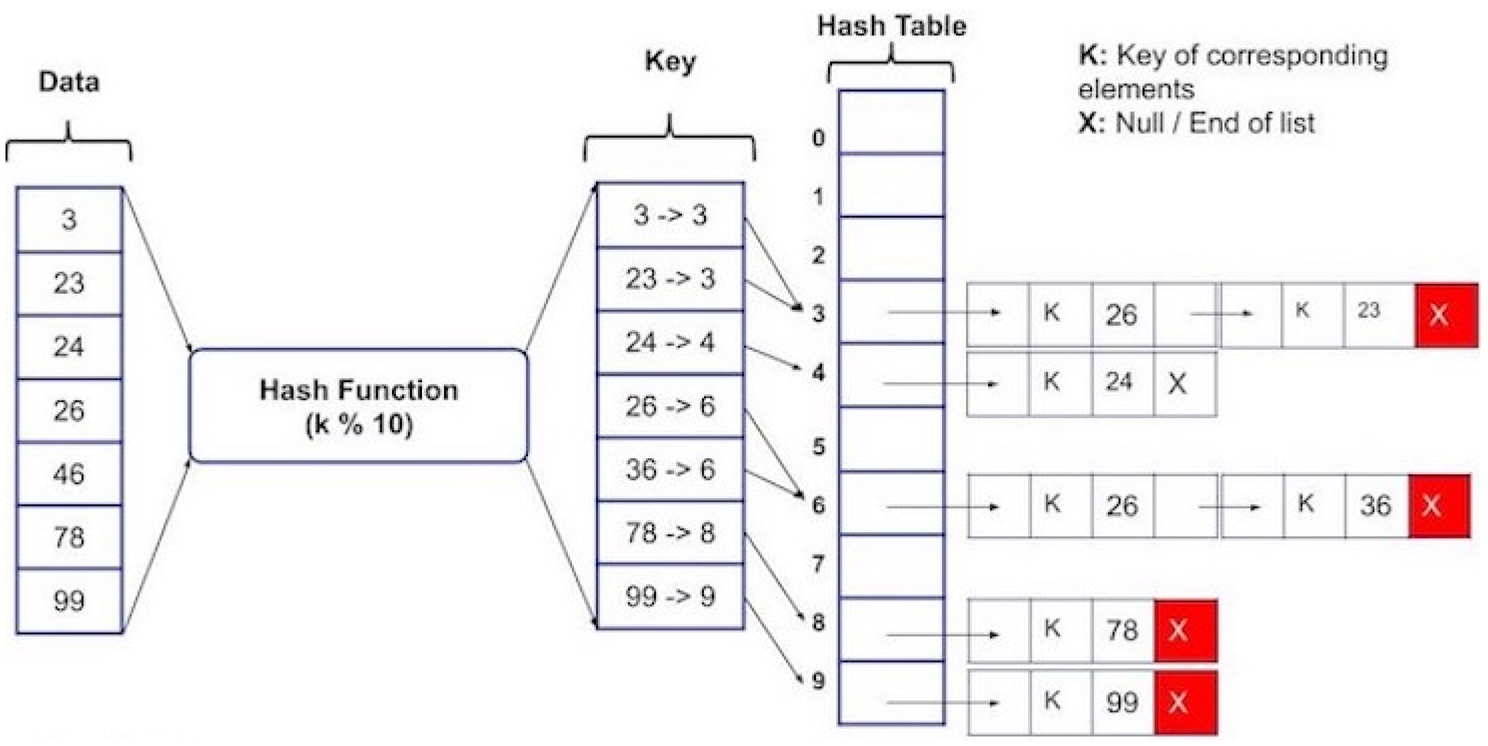
Collisions in hash tables

- We know that hashing is a ADT (Abstract Data Type) that maps its key to values using some relation known as hash function.
- It uses an array of buckets or slots to store values. During lookup/storing, the key is hashed and the resulting hash indicates where the corresponding value is/to-be stored.
- However, it is possible that multiple keys get the same hash value. This is called collision. An ideal hash function minimises collisions.
- Usually some method is used to handle collisions, one of which is chaining.

Chaining in hash tables

- In this method, values having same hash values are stored in the form of lists.
- Thus, each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash. New key-value pairs are added to the end of the list.
- Lookup algorithm searches through the list to find matching key.
- Initially table slots contain nulls. List is being created, when value with the certain hash is added for the first time.

Consider the following example.

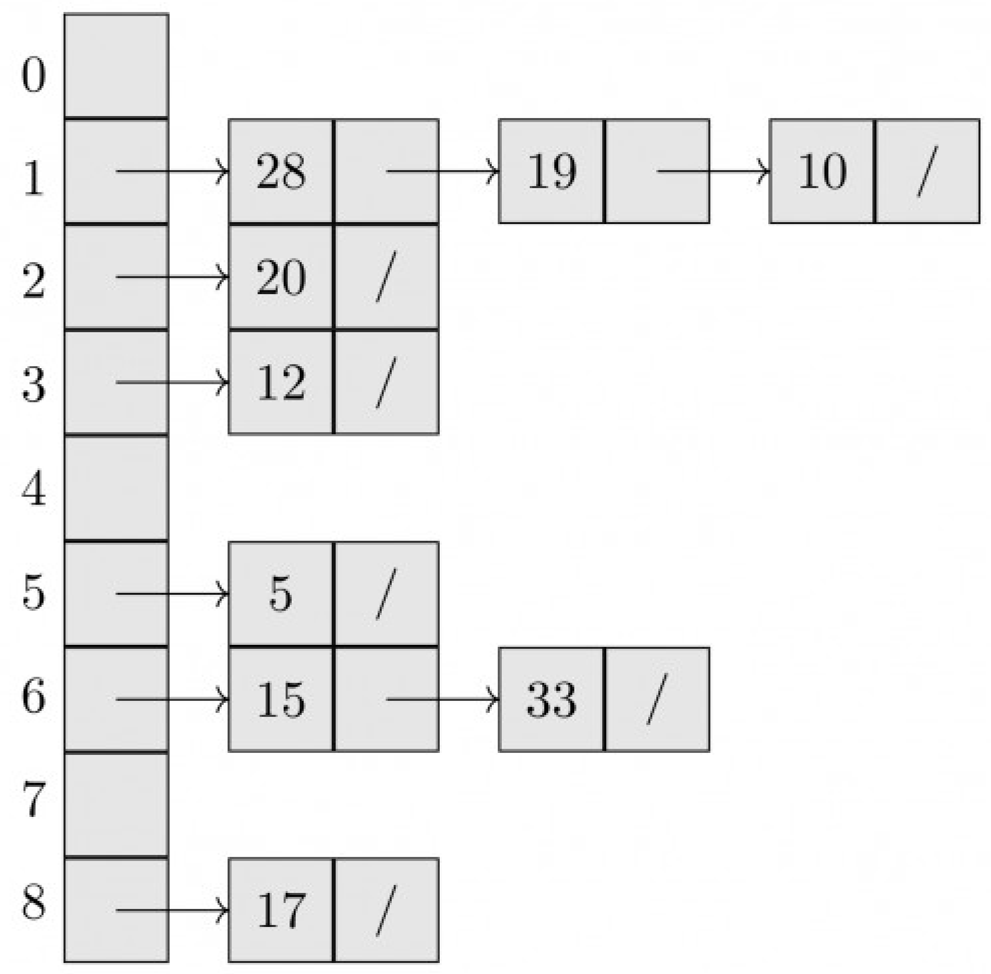


Now, in our question the keys to be added are [5, 28, 19, 15, 20, 33, 12, 17, 10] in a hash table with hash function $h(k) = k \; mod \; 9$ and which uses chaining to resolve collisions.

Calculating, the hash value for each key for the given hash functions gives us the following.

Value of key	Value of hash function h(k)
5	5
28	1
19	1
15	6
20	2
33	6
12	3
17	8
10	1

Clearly, there are collisions for some keys. Thus, now inserting the values into a hash table with nine buckets with collisions handled with chaining.



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.

Answer 3

Firstly, let us see the definitions of successor and ancestor.

Successor

Successor of a node (say A) is the node that follows A in the inorder traversal of the concerned BST. It can also be defined as the the node with smallest value which is greater than the value of A. Clearly, there exists no successor for the node with the largest value i.e. the last node in the inorder traversal.

Ancestor

Consider a BST rooted at R. Now, consider a node (say A) in the BST. Ancestors of A are defined as the nodes we need to traverse to reach A from R (including R and A).

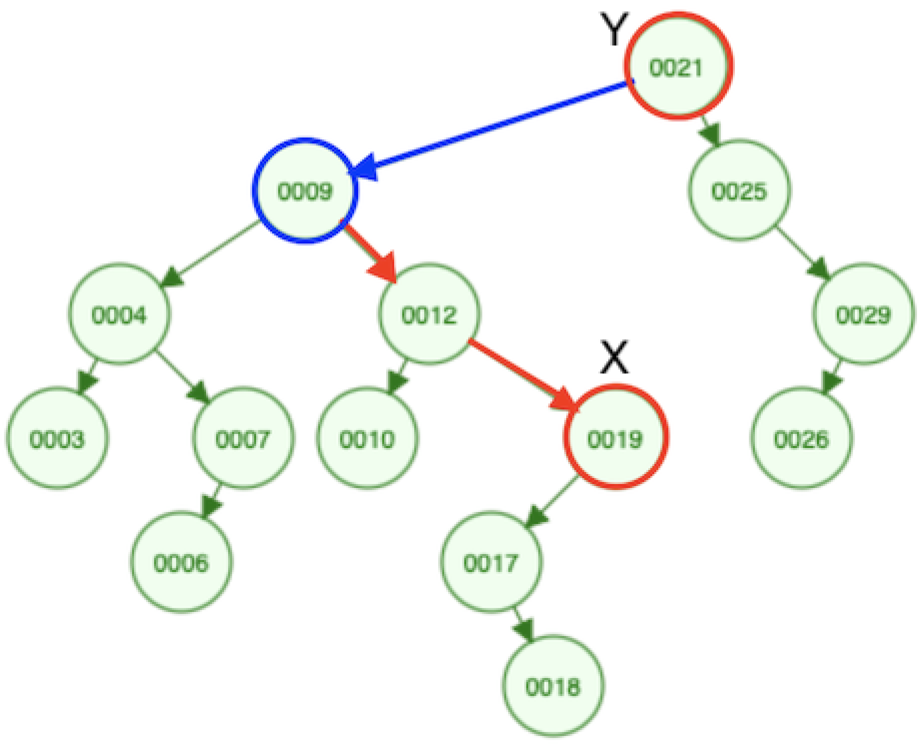
Now, in our question we need to prove : Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.
Consider, the following proof.

CLAIM 1 : Node **y** must be an ancestor of **x** We prove this by using proof by contradiction. We assume that y is not an ancestor of x. If so, let a denote the first common ancestor of x and y. By the property of BST stated above,either x < a < y or x > a > y but in our case,as x < y, we have x < a < 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 2 : The left child of **y** must be an ancestor of **x** Again, we prove this using proof by contradiction. Assume that y.left (the left child of **y**) is not an ancestor of **x**. If so, then as y is an ancestor of x (since we have proved **Claim 1**) therefore atleast one of it's child must be an ancestor of x, therefore the right child of y would be an ancestor of x, implying x > y which is false. Therefore, by contradiction the left child of y must be an ancestor of x.
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.

Thus, using proof of contradiction we have proved both of the above statements.

Consider, the following example



In the following BST, let **x** be the node with the value 19. As mentioned in the question, the right subtree of x is empty. Now, we can see the node with the smallest value, greater than value of x (19) is the node with value 21 (root), therefore that node would be **y**.
Now, the ancestors of x include the nodes [21, 9, 12, 19]. On inspection, we see that the lowest node from the set whose left child is also an ancestor of x is 21 (y). Thus, this example follows the statement given in the question

Question 4

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

Answer 4

Here we need to implement an originally recursive algorithm as an iterative algorithm using an 'explicit stack'.

Firstly we will look at the recursive algorithm.

In [ ]:
```
#Function to implement the recursion algorithm
def permutation_recursion(initial_list,numbers,n_bit):

        #Value to check if we encountered no zeroes
        no_zero = 1

        #Loop through the n_bit[ ] list
        for i in range(len(n_bit)):

                #If we encounter a 0, change it to 1, push corresponding number into numbers[ ]
                if(n_bit[i]==0):
                        no_zero=0
                        n_bit[i]=1
                        numbers.append(initial_list[i])

                        #Make recursion call
                        permutation_recursion(initial_list,numbers,n_bit)

                        #Revert the changes for backtracking
                        n_bit[i]=0
                        numbers.pop()

        #If we encounter no 0s, print the numbers[ ] list and return
        if no_zero:
                print(numbers)


n = int(input("Enter n : "))
inital_list = []
for i in range(n):
        initial_list.append(i+1)
numbers = []
n_bit = [0]*n

permutation_recursion(initial_list,numbers,n_bit)
```

Output for some input

-

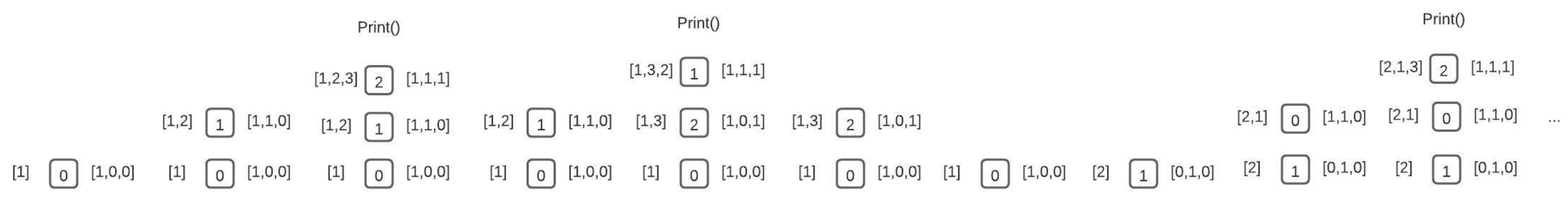
```
Enter n : 3
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

In the above code there are three arrays.
1) **initial_list[ ]** : This is the array storing the numbers from 1 to n.
2) **numbers[ ]** : This list stores the numbers which are currently active.
3) **n_bit[ ]** : This list conatains n bits. If the bit at index 'i' is 1 means that the number in index 'i' in initial_list[ ] is currently active or in numbers[ ] list.

In each call of the recursive function, we start traversing the n_bit[ ] array and for every 0 we encounter, we first make that element 1, then push the corresponding element of initial_list[ ] into the numbers[ ], and then we make a recursion call. After the recursion call is returned we revert back the changes i.e. make the 1 as 0 and pop the number from the numbers[ ].
And if we see that we encounter no 0s while traversing, we simply print the initial_list[ ] and return the function as the base case.

Now, in the implementation of recursive algorithms, all the local variables will be pushed onto a stack before pushing the new function call onto the call-stack. In our case, the index at which the recursive call is made is a local variable and thus it will be pushed onto the stack.

Let us try and visualise some of the output values



Here, we see the implementation of the recursive algorithm. The centre value is the stack and the local index values pushed onto the stack. To the left of the stack contains the current numbers[ ] list and to the right is the current n_bit[ ] list.

Now, let us look at the iterative version of the same algorithm using explicit stack

In [ ]:

```
#Function to implement above recursion algorithm iteratively using explicit stack.
def permutation_stack(initial_list):

        #Initialse lists
        numbers = []
        n_bit = [0]*len(initial_list)

        #Initalise explicit stack and address value
        stack = []
        address = 10

        #Loop to implement the algorithm
        while 1:


                #The case when in the previous interation of the while loop a recursive call was made
                if address == 10:
                        #Value to check if we encountered no zeroes
                        no_zeros = 1

                        #Loop through the n_bit[ ]
                        for i in range(len(initial_list)):

                                #If we encounter a 0, make it 1, push the corresponding number into numbers[ ]
                                #push the corresponding index onto the stack and break the loop
                                #Pushing the value is equivalent to making a recursive call
                                if n_bit[i] == 0:
                                        no_zeros = 0
                                        n_bit[i] = 1
                                        numbers.append(initial_list[i])
                                        stack.append(i)
                                        break

                        #If we encounter no zeros, we print the numbers[ ] list
                        if no_zeros:
                                print(numbers)

                                #Now we need to backtrack and start poping the stack as we return the function here
                                #so we change the address to 20
                                address = 20



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

                        #Index where we left off for that level
                        index = stack.pop()

                        #Make the n_bit value at index 0, and pop the corresponding number from the numbers[ ]
                        numbers.pop()
                        n_bit[index] = 0

                        #Iterate the n_bit[ ] from index
                        for i in range(index+1,len(initial_list)):

                                #If we encounter a 0, push the corresponding index onto the stack and break the loop
                                if n_bit[i] == 0:
                                        n_bit[i] = 1
                                        numbers.append(initial_list[i])
                                        stack.append(i)
                                        #Again, as a recursive call is made, we change the address to 10
                                        address = 10
                                        break

                                #If we find no zeros, means we returened the function, so keep the address as 20 for ne



                #When the stack size is 0, break the while loop and return the function
                if len(stack) == 0:
                        break

n = int(input("Enter n : "))
initial_list = []
```

```
for i in range(n):
    initial_list.append(i+1)

permutation_stack(initial_list)
```

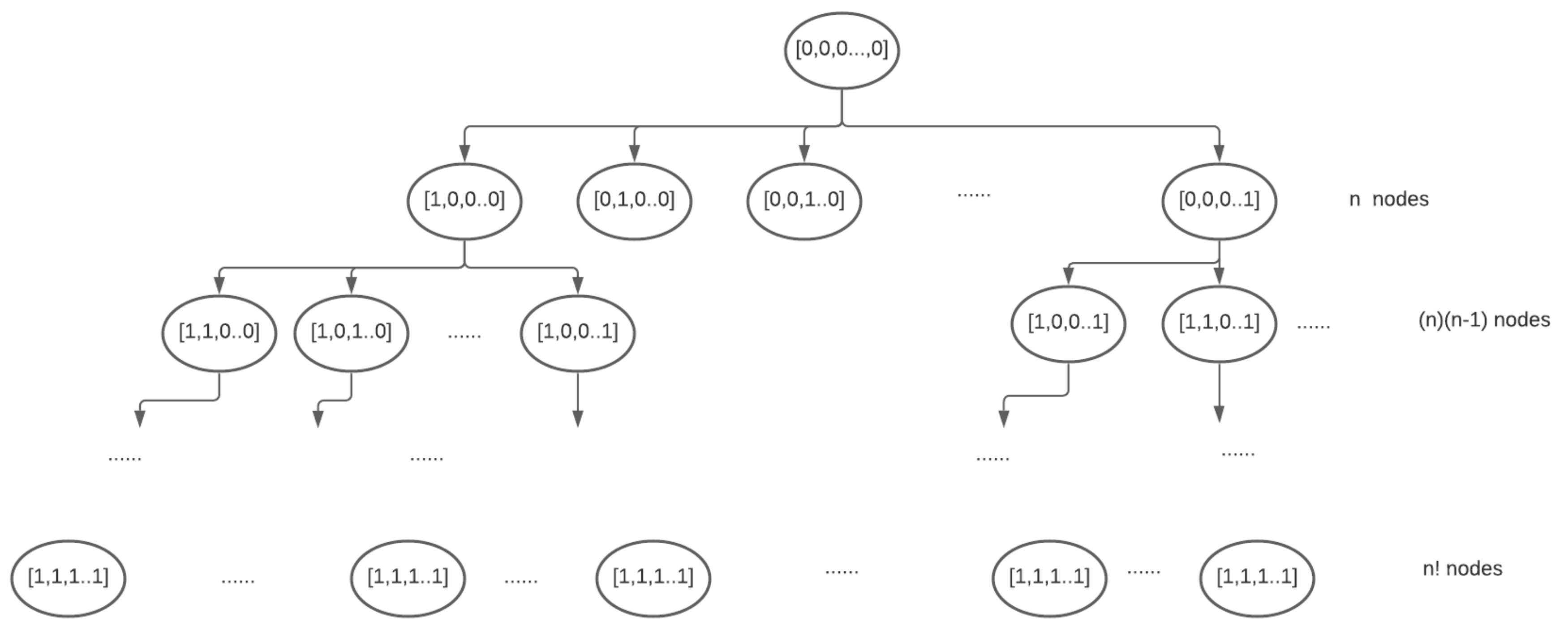
Output for some input

-
  ```
  Enter n : 3
  [1, 2, 3]
  [1, 3, 2]
  [2, 1, 3]
  [2, 3, 1]
  [3, 1, 2]
  [3, 2, 1]
  ```

Here all the three lists i.e. numbers[ ], n_bit[ ],and initial_list[ ] have the same meaning as before. The stack[ ] list is the explicit stack or the equivalent call-stack. The while loop is divided into two parts. The case where address is 10 is executed when in the previous iteration, a recursion call was made. And the case where address is 20 is executed when in the previous iteration, a function was returned.
Just like the recursive algorithm, 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 n_bit[ ] list from the next index. The visualisation of this algorithm is exactly the same as the one of the recursive algorithm.

Time and space complexity

To clearly, understand the time complexity, lets make a recursion tree of n_bit[ ] for the algorithm.



Now, we know that for each node we traverse the n_bit[ ] list once. Therefore the time for one node is O(n). Therefore for total time,

$$\begin{aligned}
total\ time &= no.\ of\ nodes \cdot O(n) \\
&= (\frac{n!}{1!} + \frac{n!}{2!} + \frac{n!}{3!} + \dots \frac{n!}{n!}) \cdot O(n). \\
&= n!(\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots \frac{1}{n!}) \cdot O(n). \\
Taking\ the\ upper\ bound\ case\ where\ n \to \infty \\
&= n!(\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots\ upto\ infinity\ ) \cdot O(n). \\
&= n!(e^1) \cdot O(n). \\
&= O(n! \cdot n)
\end{aligned}$$

$$\therefore \underline{Time\ complexity\ =\ O(n! \cdot n)}$$

For, space complexity we see that the size of the stack would be highest when we are on the lowest level of the recursion tree. So, its maximum size would be equal to the (number of nodes between root and the lowest level)*O(1) as we store constant numbers in each node i.e. index values.Therefore it would be equal to O(n). Also, n_bit[ ] and numbers[ ] are of size O(n).

$$\underline{Space\ complexity\ =\ O(n)}$$

Question 5

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

Answer 5

Consider, the following proof

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 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.

The best case is when the given BST is already a right-going chain, we are finished. Otherwise, there is at least one left-going edge in the tree. On this edge we can perform right rotation. After the rotation, the child node moves to the right of its parent node, which means that the number of nodes in the right subtree is larger by 2. Thus the number of nodes in the right subtree grows every time we do the right rotation. Consequently, after at most n —1 right rotations, the whole tree gets transformed into the right-going chain.

Now our aim is to prove the original claim using this fact. Assume we want to transform binary-search tree T1 with n nodes to T2. First we transform T1 to a right-going chain, using the above procedure, which takes at most n-1 rotations. Now we reason as follows: we can transform T2 to the same right-going tree using at most n-1 rotations. Since left rotations are inverses of right transformations, we use the left-rotations on right-going tree to obtain tree T2.

Thus, to summarize:

a) Converting T1 into a right-going chain takes r_1 rotations, then r < n
b) Perform inverse rotations of those rotations which converts the T2 into a right-going chain takes r_2 rotations then r_2 < n

Total Rotations needed $R_{total} \ = \ r_1 \ + r_2 \ < 2 \cdot n$

Hence we can conclude that $R_{total} \ = \ O(n)$