Roll No: EE18B067 Name:Abhishek Sekar

Date: November 22, 2021

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

Solution:
Code:

```
#importing queue

from queue import Queue

class Stack: #parent class for the stack

    def __init__(self): #constructor

        #defining two queues
        self.q1 = Queue()
        self.q2 = Queue()

        #a counter which maintains current no of elements
        self.size = 0 #initialize size with 0

    def push(self,x):
        self.q1.put(x) #enqueue to queue 1
        self.size += 1 #increase the size by 1

    def pop(self):

        #if no elements in the q1
        if(self.q1.empty()):
            return

        #otherwise leave one element in q1 and push the others in q2
        while(self.q1.qsize() != 1):
            self.q2.put(self.q1.get())

        #pop the remaining element from q1
        popped_ele = self.q1.get()
        self.size -= 1
```

```
        #swap the two queues
        self.q1,self.q2 = self.q2,self.q1

    def length(self):
        return self.size

    def print_stack(self):
        for n in reversed(list(self.q1.queue)):
            print(n)
```

Sample Input and Output:

```
What operation do you want to perform? 1:push 2:pop  1
Please enter the number you want to push  1
Printing the contents of the stack below
1
Do you wanna continue performing operations on the stack? (y/n)  y
What operation do you want to perform? 1:push 2:pop  1
Please enter the number you want to push  2
Printing the contents of the stack below
2
1
Do you wanna continue performing operations on the stack? (y/n)  y
What operation do you want to perform? 1:push 2:pop  1
Please enter the number you want to push  3
Printing the contents of the stack below
3
2
1
Do you wanna continue performing operations on the stack? (y/n)  y
What operation do you want to perform? 1:push 2:pop  1
Please enter the number you want to push  4
Printing the contents of the stack below
4
3
2
1
Do you wanna continue performing operations on the stack? (y/n)  y
What operation do you want to perform? 1:push 2:pop  2
Printing the contents of the stack below
3
2
1
Do you wanna continue performing operations on the stack? (y/n)  y
What operation do you want to perform? 1:push 2:pop  2
Printing the contents of the stack below
2
1
Do you wanna continue performing operations on the stack? (y/n)  n
Final contents of the stack after all operations
```

```
2
1
```

Explanation:
Some Definitions:

- Push: Inserting element into stack. Element is inserted at the top of the stack.
- Pop: Deleting an element from the stack. Element is deleted from the top of the stack.
- Enqueue: Inserting an element into queue.Element is inserted at the top of the queue.
- Dequeue: Deleting an element from the queue. Element is deleted from the bottom of the queue.

Shown below is a figure which illustrates the algorithm implemented above for the given example.

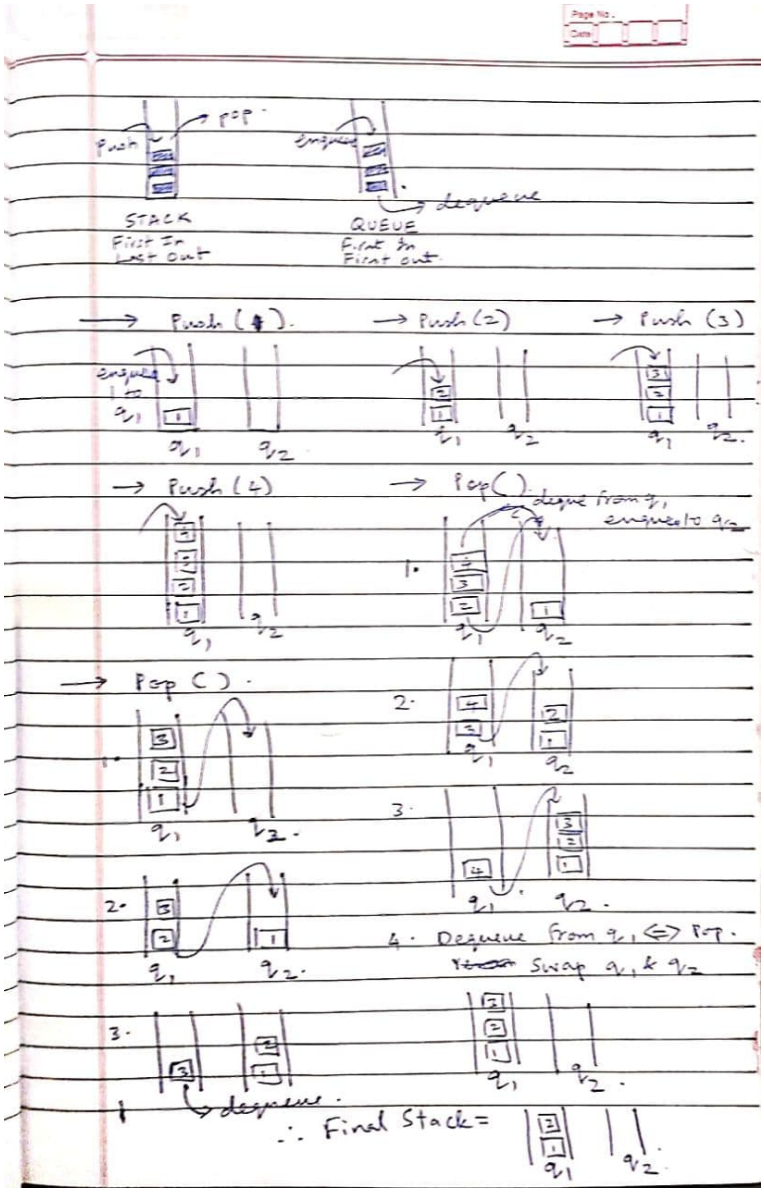

Figure 1: Example of Stack using two queues

We'll have to compromise either on the push() operation or the pop() operation while implementing the stack using two queues. Since it is more likely for us to insert elements into the stack, the push() operation has been optimized in this implementation.

Push() operation

For the push() operation we just enqueue the element to be pushed into the first queue as shown in the figure. This has a similar functionality as pushing an element into a stack.

Pop() operation

There are three steps involved for the pop operation.

- We first repeatedly dequeue elements from the first queue and these elements are enqueued into the second queue in an orderly manner as shown in the figure.
- When there is just one element left in the first queue, we dequeue that element, but don't enqueue it to the second queue like we did before. This is equivalent to the pop() operation.
- To be consistent, we swap the first and the second queues as the first queue is used for representing the stack.

Time Complexity:

For the push() operation, we just enqueue an element and hence the operation is of order $\mathcal{O}(1)$.

For the pop() operation, we need to repeatedly dequeue the elements from the first queue and enqueue all but the last element into the second queue. Additionally, we perform a swap operation. Therefore, the total number of operations are n dequeues + n-1 enqueues + 1 swap operation. Thus, the time complexity of this operation is $\mathcal{O}$(n) where n is the number of elements in the stack.

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.

Solution:

Output of hashing function for all the keys:

- Key 5: Output of hashing function = 5
- Key 28: Output of hashing function = 1
- Key 19: Output of hashing function = 1
- Key 15: Output of hashing function = 6
- Key 20: Output of hashing function = 2
- Key 33: Output of hashing function = 6
- Key 12: Output of hashing function = 3
- Key 17: Output of hashing function = 8
- Key 10: Output of hashing function = 1

Hash Table

Slots : Chaining

0 : [ ]
1 : [28] ↔ [19] ↔ [10]
2 : [20]
3 : [12]
4 : [ ]
5 : [5]
6 : [15] ↔ [33]
7 : [ ]
8 : [17]

Where ↔ represents the appropriate chaining.

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.

Solution:
Some definitions:

- Ancestor: The ancestor of node x is any node which comes above it.By convention, x is an ancestor of itself.
- Successor: The successor of node x is the smallest node that is greater than node x.

Proof:
Note: For a binary search tree, the left child is smaller than the parent and the right child is larger. Moreover, the parent is larger than it's left sub tree and smaller than the right sub tree.
We can try proving this using an inductive approach. As T contains y the successor to x, we know that some ancestor of x will be the left child of some parent node z. We'll try proving that z is the successor to x and therefore z = y.

- Base Case:

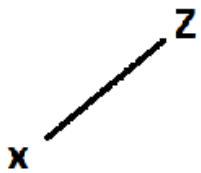

Figure 2: Case I

Supposing z is the parent node of x and x is it's left child. Then we have z > x. If z ≠ y, then we have some y, such that, $x < y < z$. Therefore, from this inequality, using the properties of a binary search tree, we have y to be in the left sub tree of z and the right sub tree of x which is a contradiction as x doesn't have a right sub tree. Therefore, z = y.

- Alternative Case: Supposing x is the right child of its parent node a_1. Say a_1 is the left child of z and z is such that $z \neq y$. Then we have, $z > x > a_1$.Now, we have $z > y > x > a_1$. Therefore, y has to come in the right sub tree of a_1 or x. This is a contradiction as x has no right sub tree and the right child of a_1 is x. Therefore, z = y.

- Let us extend the above cases. If a_1 has a parent a_2 such that a_1 is the right child of a_2. We have a_2 being the right child of its parent a_3 and so on and so forth until there is an ancestor a_n who is the left child of z. Then, the above derived relationship still holds, we have $z > x > a_1 > a_2 > a_3 \ldots > a_n$. Therefore, if $z \neq y$ it will yet again result in a contradiction as seen above as x will still be the maximum value of the right sub tree of any of the ancestors.

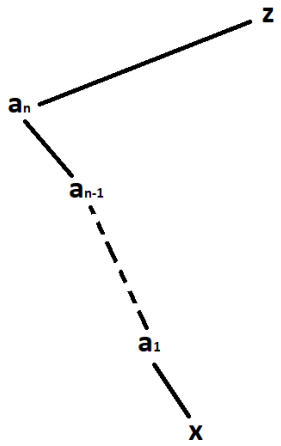

Figure 3: Case II

Therefore, from this, we see that y is the lowest ancestor of x whose left child is also an ancestor of x.

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

Solution:
Converting Left tree to Right

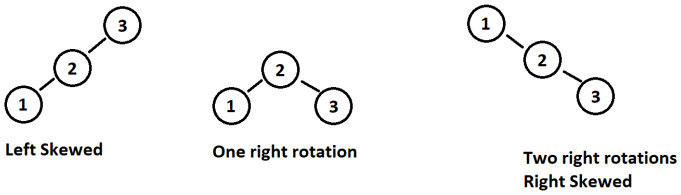

Figure 4: Converting a Left Skewed tree to a Right Skewed one using rotations

From the above example, we see that every right rotation we perform increases the right path length by 1.
Therefore, in the worst case, to obtain a right skewed tree from a left skewed one, we'll need n-1 right rotations if n is the number of nodes in the binary tree.
Similarly, we can likewise argue that any right skewed binary tree can be converted to a left skewed one using n-1 left rotations.
Therefore, any arbitrary binary tree can be converted to the left or the right skewed binary tree by using n-1 rotations.
Final Proof:
Let our initial configuration of a certain node be A and let it be B at the final configuration of the binary tree.
To get the right skewed binary tree from A, we need atmost n-1 right rotations as shown above.
Now, if B we're not to be the right skewed binary tree, we'll have to do atmost n-1 left rotations to arrive at B.
Therefore, our total number of rotations can not exceed 2(n-1) and is therefore $\mathcal{O}$(n).
Therefore, any n-node binary tree can be converted to any other n-node binary tree using $\mathcal{O}$(n) rotations.

5. Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

Solution:
Code:

```
class Node: #the node of a circular linked list, holds data and points to next node

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

class Circular_List:

    def __init__(self): #constructor
        self.head   = None #initialize the head node with NULL
        self.length = 0 #length of the circular linked list

    def print_list(self): #prints the circular list

        if(self.head == None): #if head is NULL list empty
            print('List empty')
            return

        cll = f"{self.head.data}" #start off with the head node
        temp = self.head.next
        while(temp != self.head): #advance till list back to beginning

            cll += f"-> {temp.data}"
            temp = temp.next

        print(cll)
        return

    def append(self, data): #inserting at the "end"

        self.insert(data, self.length)
        return

    def insert(self,data, index): #insert at a particular index from the head
```

```
        if((index > self.length)|(index < 0)): #faulty index

            print("Faulty Index input")
            return

        if(self.head == None): #list empty

            self.head    = Node(data) #insert at beginning
            self.length += 1
            return

        temp = self.head

        ind_trav = (self.length-1)*(index - 1 == -1) + (index-1)*(index -1 != -1) #traverse

        for i in range(ind_trav):
            temp = temp.next

        #insert data after the above node at the indexˆth position

        after_node = temp.next
        temp.next  = Node(data)
        temp.next.next = after_node

        if (index == 0): #make the newly inserted node the head node
            self.head = temp.next

        self.length += 1
        return

    def delete(self, index):

        if((index >= self.length)|(index < 0)): #faulty index

            print("Faulty Index input")
            return

        if(self.length == 1): #only head node in the array
            self.head    = None
            self.length -= 1
```

```
        temp = self.head

        ind_trav = (self.length-1)*(index - 1 == -1) + (index-1)*(index -1 != -1) #traverse

        for i in range(ind_trav):
            temp = temp.next

        after_node = temp.next.next
        temp.next  = after_node #remove the node at "index" from the chain

        if(index == 0):
            self.head = after_node

        self.length -= 1
        return

    def search(self,data):

        temp = self.head

        for i in range(self.length):
            if(temp.data == data):
                return i
            temp = temp.next

        return None

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

Sample Input and Output:

```
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  2
Please enter the number you want to append  1
Printing the contents of the circular list below
1
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  2
Please enter the number you want to append  2
Printing the contents of the circular list below
1-> 2
Do you wanna continue performing operations on the circular list? (y/n)  y
```

```
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  2
Please enter the number you want to append  3
Printing the contents of the circular list below
1-> 2-> 3
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  2
Please enter the number you want to append  4
Printing the contents of the circular list below
1-> 2-> 3-> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  4
Please enter the number you want to search for  2
Element 2 lies at index no: 1
Printing the contents of the circular list below
1-> 2-> 3-> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  3
Please enter the index of the element you want to delete  1
Printing the contents of the circular list below
1-> 3-> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:append 3:delete 4:search  1
Please enter the number you want to insert  5
Please enter the index you want to insert it in  1
Printing the contents of the circular list below
1-> 5-> 3-> 4
Do you wanna continue performing operations on the circular list? (y/n)  n
Final contents of the circular list after all operations
1-> 5-> 3-> 4
```

Explanation
In the algorithm implementation above, the code generalizes the insert and delete operation so as to allow the insertion/deletion anywhere in the circular list.
Moreover, the insertion and deletion have been implemented in an index based fashion, meaning, the insertion/deletions are made at the specified index.
Insertion and deletion can be made using a value based manner too by fusing a search option with them.
Firstly, the circular list is initialized with the head node being NULL.
Insert:

- Firstly, we check if the entered index is a valid one and if it isn't, we throw an error.
- We then check if the circular list is empty and in that case, we insert our data to the head node and the head node points to itself forming a one element circular list.
- For all other cases, we traverse along the queue from the head node till we arrive at the desired index. In the case of appending, we traverse till one element before the head node as the linked list is circular.
  As it is a singly linked list, each element points to the next element and that is the only way we can access any element.
  Thus, as we traverse through the circular list, we keep updating the temp node to the current node.
  As we reach the node before the index, we perform the insertion and make the inserted node point to the next node to continue the chain of links.
  Lastly, if we are to insert at index 0, we just re-index the circular list to make the inserted node the head node.

Delete:

- As in the previous case, we first check if the entered index is a valid one. However, here we'd want the index to be strictly lesser than the length as the indexing starts from 0.

- We then check the boundary case when the circular list comprises just of the head node. In that case, we just reinitialize the head node to Null and this is equivalent to deleting the head node.
- For all other cases, as in the previous case, we traverse the circular list from the head using a temp variable till the desired index is reached. Once the desired index is reached, we make it point to the node after the one it is currently pointing to. Thus, by removing the link to the node at the index, we delete the node. As in the previous case, we just re-index the circular list if we were to delete the node at index 0.

Search:

- We initialize the temp variable as the head node of the circular queue.
- We then traverse through the circular queue and as we traverse we keep checking if the value held by the node is the value we are searching for.
  If it is, we return the appropriate index. If it is not, we just update our temp variable to the next node and continue traversing the array.
  If the value we are searching for isn't present in circular queue None is returned.

Time Complexity:
As the given circular list is single linked, we'll have to traverse throughout the circular list as each node points to the next element.
Therefore, unless we are inserting or deleting at a special location such as the head node, the worst case time complexity for all three dictionary operations is $\mathcal{O}(n)$ where n is the number of elements in the circular list.
Additional Optimization
Code:

```
#dictionary operations with keys

class Node_2:

    def __init__(self,key,data): #constructor
        self.key    = key
        self.data   = data
        self.next   = self

class CLL_dictionary:

    def __init__(self):

        self.head   = None #initialize the head node with NULL
        self.length = 0 #length of the circular linked list

    def print_list(self): #prints the circular list

        if(self.head == None): #if head is NULL list empty
            print('List empty')
```

```
            return

        cll_data = f"{self.head.data}" #start off with the head node
        cll_keys = f"{self.head.key}"
        temp = self.head.next
        while(temp != self.head): #advance till list back to beginning

            cll_data += f" -> {temp.data}"
            cll_keys += f" -> {temp.key}"

            temp = temp.next

        print("Keys:",cll_keys)
        print("Data:",cll_data)
        return

    def search(self,key):

        temp = self.head

        for i in range(self.length):
            if(temp.key == key):
                return i
            temp = temp.next

        return None

    def insert(self,key,data):

        if(self.head == None): #list empty

            self.head    = Node_2(key,data) #insert at beginning
            self.length += 1
            return

        #search to see if the key is already present in the dictionary

        index = self.search(key)

        if(index == None): #if key is not present, insert at the end
```

```
            temp       = self.head
            for i in range(self.length - 1):
                temp = temp.next

            after_node = temp.next
            temp.next  = Node_2(key,data)
            temp.next.next = after_node #recreate the chain

            self.head  = temp.next.next #as we want the newly inserted node to be the last
            self.length += 1

            return

        else:

            temp = self.head

            ind_trav = (self.length-1)*(index - 1 == -1) + (index-1)*(index -1 != -1) #trave

            for i in range(ind_trav):
                temp = temp.next

            #insert data after the above node at the index^th position

            #after_node = temp.next
            after_node = temp.next.next
            temp.next  = Node_2(key,data)
            temp.next.next = after_node

            if (index == 0): #make the newly inserted node the head node
                self.head = temp.next

            self.length += 1
            return


    def delete(self,key):

        index = self.search(key)
```

```
        if(index == None):
            print('The given key doesnt exist')
            return

        else:

            if(self.length == 1): #only head node in the array
                self.head    = None
                self.length -= 1

            temp = self.head

            ind_trav = (self.length-1)*(index - 1 == -1) + (index-1)*(index -1 != -1) #trave

            for i in range(ind_trav):
                temp = temp.next

            after_node = temp.next.next
            temp.next  = after_node #remove the node at "index" from the chain

            if(index == 0):
                self.head = after_node

            self.length -= 1
            return


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

Sample Input and Output:

```
What operation do you want to perform? 1:insert 2:delete 3:search  1
Please enter the number you want to insert  1
Please enter the key you want to insert it in  a
Printing the contents of the circular list below
Keys: a
Data: 1
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  1
Please enter the number you want to insert  2
Please enter the key you want to insert it in  b
```

```
Printing the contents of the circular list below
Keys: a -> b
Data: 1 -> 2
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  1
Please enter the number you want to insert  3
Please enter the key you want to insert it in  c
Printing the contents of the circular list below
Keys: a -> b -> c
Data: 1 -> 2 -> 3
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  1
Please enter the number you want to insert  4
Please enter the key you want to insert it in  d
Printing the contents of the circular list below
Keys: a -> b -> c -> d
Data: 1 -> 2 -> 3 -> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  3
Please enter the key you want to search for  d
Key d lies at index no: 3
Printing the contents of the circular list below
Keys: a -> b -> c -> d
Data: 1 -> 2 -> 3 -> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  2
Please enter the key of the element you want to delete  c
Printing the contents of the circular list below
Keys: a -> b -> d
Data: 1 -> 2 -> 4
Do you wanna continue performing operations on the circular list? (y/n)  y
What operation do you want to perform? 1:insert 2:delete 3:search  1
Please enter the number you want to insert  3
Please enter the key you want to insert it in  d
Printing the contents of the circular list below
Keys: a -> b -> d
Data: 1 -> 2 -> 3
Do you wanna continue performing operations on the circular list? (y/n)  n
Final contents of the circular list after all operations
Keys: a -> b -> d
Data: 1 -> 2 -> 3
```

Optimizations:
The above code is a slightly modified version of the code we discussed earlier. Here, we just perform all the operations in a key based manner since ideally in a dictionary we would not like to tamper with the data.
The approach remains the same as before with one slight difference.
We first search for the key and obtain the index where it is stored at and perform the insert or delete operation at that index. If the key isn't present, we get an error message in the case of delete operation and for insert we just append the new key at the end.
The time complexity of these operations still remain $\mathcal{O}$(n) as we haven't changed our approach.