Queue

Thursday, 3 August 2023 6:32 AM

① Queue implementation using Circular list

→ Advantages → cache friendly for arrays
→ O(1) time for all methods operation

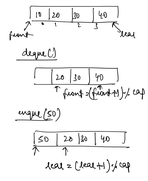
q = MyQueue(capacity) ; q.enque(n) ; q.deque() ; q.size() ; q.getFront() ; q.getRear() ; q.isEmpty()

Circular list implementation of Queue

```
class MyQueue :
    def __init__(self,c):
        self.l = [None]*c
        self.cap = c
        self.size = 0
        self.front = 0

    # class methods
```

rear = (front + size − 1) % cap




```
    def getFront(self):
        if self.size==0:
            return None
        else:
            return self.l[self.front]

    def getRear(self):
        if self.size==0:
            return None
        else:
            rear=(self.front+self.size-1)%self.cap
            return self.l[rear]

    def enque(self,x):
        if self.size == self.cap:
            return
        else:
            rear=(self.front+self.size-1)%self.cap
            rear=(rear+1)%self.cap
            self.l[rear]=x
            self.size = self.size+1

    def deque(self):
        if self.size==0:
            return None
        else:
            rem=self.l[self.front]
            self.front=(self.front+1)%self.cap
            self.size=self.size-1
            return rem

    def isEmpty(self):
        if self.size==0:
            return True
        else:
            return False
    def isFull(self):
        if self.size==self.cap:
            return True
        else:
            return False
    def size(self):
        return self.size
```

② Implement Stack using Queue

Idea : Two Queues : q1 → To keep the actual items
q2 → To be used as an auxiliary queue

push(10) : q1 = [10], q2 = []

push(20) :
(a) q1 = [], q2 = [10]
(b) q1 = [20], q2 = [10]
(c) q1 = [20,10], q2 = []

push(30)
(a) q1 = [], q2 = [20,10]
(b) q1 = [30], q2 = [20,10]
(c) q1 = [30,20,10], q2 = []

push(x) → (a) Move everything from q1 to q2
(b) Enque x to q1
(c) Move everything back from q2 to q1

pop() → Remove front of q1
top() → return front of q1
size() → return size of q1

Idea for optimized solution :

Two queues : q1 → To keep the actual items
q2 → to be used as an auxiliary queue

push(x): (a) Enqueue x to q2
(b) Move Everything from q1 to q2
(c) Swap the variables q1 and q2

pop() : remove front of q1
top() : return front of q1
size() : return size of q1

```
class stack:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()
    def push(self,x):
        self.q2.append(x)
        while self.q1:
            self.q2.append(self.q1.popleft())
        self.q1, self.q2 = self.q2, self.q1

    def pop(self):
        if self.q1:
            self.q1.popleft()

    def top(self):
        if self.q1:
            return self.q1[0]
        else:
            return None

    def size(self):
        return len(self.q1)
```

③ Reverse a Queue

I/p : q = [20, 10, 15, 30] (front … rear)
O/p : q = [30, 15, 10, 20] (front … rear)

Iterative Solution
① Create an empty stack
② Move all items of q to stack
③ Move all items back to q

```
def reverseQueue(q):
    st=[]
    while q:
        st.append(q.popleft())
    while st:
        q.append(st.pop())
```

q is an object of deque() from collections

Time : O(n)
Space : O(n)

Recursive Soln

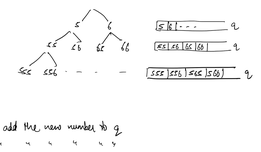
```
def revQ(q):
    if len(q)==0:
        return
    x = q.popleft()
    revQ(q)
    q.append(x)
```

Time : O(n)
Space : O(n)

④ Generate first n numbers with the given digits

Naive ⇒ Traverse through all natural numbers while we have not generated the n numbers
⇒ For every traversed number, check if it has digits as given only. If yes, print the number and increment the count.

Idea for Efficient Soln
① Create an empty queue q
② Add 5 and 6 to q
③ Run a loop n times :
⇒ take out an item from q
⇒ print the item
⇒ append 5 to the item and add the new number to q
⇒ " 6 " " " " " " " " " "




```
def printFirstN(n):
    q=deque()
    q.append('5')
    q.append('6')
    for i in range(n):
        cur = q.popleft()
        print(cur,end=" ")
        q.append(cur+'5')
        q.append(cur+'6')
```

Time O(n)
Space O(n)

Better Implementation

```
def printFirstN(n):
    q=deque()
    q.append(" ")
    q.append("6")
    i=0
    while (i+len(q))<n:
        cur = q.popleft()
        print(cur,end=" ")
        q.append(cur+'5')
        q.append(cur+"6")
        i+=2
    while i<n:
        print(q.popleft(),end=" ")
        i+=1
```

⑤ Design a data structure with min/max operations

⇒ Design a data structure that supports following operations in O(1) time:
① insertMin(x)
② insertMax(x)
③ getMin()
④ getMax()
⑤ extractMin()
⑥ extractMax()


```
from collections import deque
class MyDS :
    def __init__(self):
        self.dq=deque()
    def insertMin(self,x):
        self.dq.appendleft(x)
    def insertMax(self,x):
        self.dq.append(x)
    def extractMin(self):
        return self.dq.popleft()
    def extractMax(self):
        return self.dq.pop()
    def getMin(self):
        return self.dq[0]
    def getMax(self):
        return self.dq[-1]
```

⑥ Maximum of all subarrays of size k

Input element ⇒ n
subarray size ⇒ k
output size ⇒ (n−k+1)

Naive Time : O(n*k) Space : O(1)

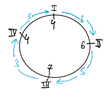
```
def printKMax(arr,k):
    for i in range(n-k+1):
        res = arr[i]
        for j in range(i+1,i+k):
            res = max(res, arr[j])
        print(res,end=" ")
```

Efficient Use doubly ended queue Time : O(n) Space : O(n)

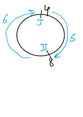
```
def printKMax(arr, k):
    dq = deque()
    for i in range(k):
        while dq and arr[i] >= arr[dq[-1]]:
            dq.pop()
        dq.append(i)
    print(arr[dq[0]],end=" ")
    for i in range(k,n):
        while dq and dq[0] <= i-k:
            dq.popleft()
        while dq and arr[i]>= arr[dq[-1]]:
            dq.pop()
        dq.append(i)
        print(arr[dq[0]],end=" ")
```

⑦ Find Circular Tour

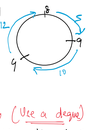
I/p : petrol[] = {4,8,7,4}
distance[] = {6,5,3,5}
O/p : 2



I/p : petrol[] = {6,5}
dist[] = {5,6}
O/p : 2



I/p : petrol[] = {8,9,4}
dist[] = {5,10,12}
O/p : -1



Naive Time : O(n²)

```
def printTour(pet,dist):
    n=len(pet)
    for start in range(n):
        curr_pet = 0
        end = start
        while True:
            curr_pet += (pet[end]-dist[end])
            if curr_pet<0:
                break
            end = (end+1)%n
            if end == start:
                return start+1
    return -1
```

Better solution (Use a deque) O(n)
① Keep adding items to the end of deque while curr_petrol is greater than equal to 0.
② Keep removing items from the front of deque while curr_petrol is negative.
→ We can avoid deque, extend array as deque.
→ Maintain start and end variables.
→ Requires 2n operations

Efficient Soln
If current petrol becomes negative at p_i, then none of petrol pumps from p_0 to p_i can be valid solution.
$p_0, p_1, \cdots, p_{i-1}, p_i, p_{i+1}, \cdots, p_{n-1}$
Let p_s be first petrol pump where current petrol becomes negative.
then $curr\_petrol = \sum_{i=0}^{s} (petrol[i] - dist[i])$

Algorithm

```
def printTour(pet,dist):
    start=0
    curr_pet=0
    prev_pet=0
    for i in range(n):
        curr_pet += (pet[i]-dist[i])
        if curr_pet<0:
            start=i+1
            prev_pet+=curr_pet=0
            curr_pet=0
    return (start+1) if ((curr_pet+prev_pet)>=0) else -1
```