BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Work Integrated Learning Programmes Division
M.Tech (Data Science and Engineering)

Design Document

12/27/2020

Data Structures and Algorithms Design

Assignment 1 – PS19 - [Box Office]

1. Problem Statement

A movie theatre manager needs your help to implement a ticketing system for the box office. He has a list of specific asks in how the ticketing system should work. Below are the asks:

- There are only 'w' number of box office windows in the theatre. Each window can have at max 'n' number of people waiting in line.

- To start with, only one window is opened. If the number of people waiting in line in that window exceeds n, then the next window is opened and people can join the line in that window. Likewise, if both the first and second windows have n number of people waiting in each queue, then a third window is opened. This can go on until the maximum number of windows w is reached. Let us assume that once a window is opened it never closes. A new window is only opened if all open windows are full.

- Each person can buy only one ticket. So, the system should not allot more than one ticket per person. Let us assume that the system issues one ticket each across all open windows. When a ticket is issued, the count of the number of people in each open queue is reduced by 1.

- When a new person has to join the queue, the system has to prompt him to join a queue such that they are issued a ticket as fast as possible. The system prompts the person based on these factors:
    - First it looks for an open window with the least number of people and prompts that window number. If more than one window has the least number of people, then the system can prompt the person to join the first window (smaller window Id) it encounters with the least number of people.
    - If the queues of all open windows are full and a new window can be opened, then the new person is prompted to join the new queue for the new box office window.
    - If all queues for all windows are full, a corresponding message is displayed. That person need not be considered in the next iteration.

- After a queue is prompted to a person, the person or system cannot change the queue.

2. Detailed Design

2.1 Data Structure used to implement the algorithm

The data structure used to implement the algorithm is a **Queue**.

Queue is an abstract data type or a linear data structure, in which the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head).

This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

For the Box Office problem statement we use **Lists** in python for Implementation of Queue.

The basic structure of the movie box office will be:

```
def __init__ (self, w, n):

    self.n = n
    self.w = w
    self.queues = [[None for i in range(n)] for j in range(w)]
    self.starts = [0 for i in range(w)]
    self.ends = [0 for i in range(w)]
    self.sizes = [0 for i in range(w)]
    self.open = [False for i in range(w)]
    self.open[0] = True
```

The number of box office windows ‘w’ and max length of each queue ‘n’ should be read from the inputPS19.txt file. The input is mention in the following format.

ticketSystem:3:5

where 3 is the number of windows ‘w’ and 5 is the queue size ‘n’

2.2 Operations to be performed

1. **def isOpen (self, windowId):** This function returns True if the box office window is open and False if it is yet to be opened (closed). This function is called when the following tag "isOpen" is encountered in the inputPS19.txt file.

   isOpen:1

   If the box office window is open it enters the following string into the outputPS19.txt file

   isOpen:1 >> True

   else it enters

   isOpen:1 >> False

2. def getWindow (self, windowId): This function returns the queue (number of people waiting) in front of the window. (Will return empty queue if window is closed). This function is called when the following tag "getWindow" is encountered in the inputPS19.txt file.

getWindow:1

If the box office window is open and there are people in the queue it outputs the queue into the outputPS19.txt file

getWindow:1 >> [1, 2, 3, 4, 5]

else it enters

getWindow:1 >> []

3. def addPerson (self, personId): This function is called to add a new person to one of the open window queues. It returns windowId of the window where the person should go to and 1 if all the queues are full. This function is called when the following tag "addPerson" is encountered in the inputPS19.txt file.

addPerson:1

addPerson:2

If the person is added to a queue the below string is entered into the outputPS19.txt file

addPerson:1 >> w1

addPerson:2 >> w1

If all queues are full, the following string is entered into the outputPS19.txt file

addPerson:1 >> all queues are full

4. def giveTicket (self): This function is called to issue a ticket at every open box office window with a queue of at least one person. The function is called when the following tag "giveTicket" is encountered in the inputPS19.txt file.

giveTicket:

giveTicket:

The giveTicket function outputs the total number of tickets issued in that instance of execution.

The number of tickets issued is entered into the outputPS19.txt file.

giveTicket: >> 2

giveTicket: >> 1

2.3 Pseudo-code for each of the operations

Algorithm `isOpen (windowId)`

Input Integer `windowId`, **List** `Open` of max `w` integers
Output True/False

Check if given `windowId` is valid
Return value from the List **`Open[windowId-1]`**

Algorithm `getWindow(windowid)`

Input Integer `windowId`, **Nested list** `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers
Output List `Queue` containing max of `n` integers

Check if given `windowId` is valid
return sub-queue **`Queue[…..]`** from nested-queue **`Queues[windowId-1]`**

Algorithm `addPerson (personid)`

Input List `Open` of `w` integers, **Nested list** `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers
Output `Integer` `windowId/-1`

```
Set current min_window_size as n
for each element in List (Open[….])
        find length of element  from Nested List  Queues[element]
        if  length < min_window_size
            Set new  min_window_size  as  length
            Set current  windowId as  index of element

if min_window_size = n
        for each sub-list in nested List (Queues[….])
            if length(sublist) < n
               append personid at rear  of that sublist
               open a new window
               return windowId

else
        append personid at rear of current windowId
         open a new window
          return windowId
```

Algorithm `giveTicket()`

Input Nested list `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers

Output Integer `ticket_dispatched`

for each `sub-list` in nested List (`Queues[….]`)
    if `length(sublist)` > 0
        `pop` element from front
        increment `ticket_dispatched`
return `ticket_dispatched`

3. Time complexity analysis

Method	Time Complexity
Algorithm `isOpen (windowId)` **Input Integer** `windowId`, **List** `Open` of max `w` integers **Output** True/False Check if given `windowId` is valid Return value from the List **`Open[windowId-1]`**	**O(1)** **Will take a constant time for each step**
Algorithm `getWindow(windowid)` **Input Integer** `windowId`, **Nested list** `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers **Output List** `Queue` containing max of `n` integers Check if given `windowId` is valid return sub-queue **`Queue[…..]`** from nested-queue **`Queues[windowId-1]`**	**O(1)** **Will take a constant time for each step**
Algorithm `addPerson (personid)` **Input List** `Open` of `w` integers, **Nested list** `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers **Output `Integer` `windowId/-1` Set current `min_window_size` as `n` for each `element` in List (`Open[….]`) find `length` of `element` from Nested List `Queues[element]` if `length < min_window_size` Set new `min_window_size` as `length` Set current `windowId as` index of `element` if `min_window_size = n` for each sub-list in nested List (Queues[….]) if length(sublist) < n `append personid` at `rear` of that sublist open a new window return `windowId` else append `personid` at `rear` of current `windowId` open a new window return `windowId`	**0(2*w)** **Depends on twice the number of windows as 2 for loops are run for "w" times**
Algorithm `giveTicket()` **Input Nested list** `Queues` consisting of max `w` sub-list and each sub-list consisting max of `n` integers **Output Integer** `ticket_dispatched` for each `sub-list` in nested List (`Queues[….]`) if `length(sublist)` > 0 `pop` element from front increment `ticket_dispatched` return `ticket_dispatched`	**0(w)** **Depends on the number of windows as length of each window is checked and elements are popped accordingly**

The time complexity analysis for given problem statement is O(2*w), where “w” is the number of windows

O(2 * number of windows)

**Time for reading the input file and writing to output file are ignored. Also, the total time of operation will also depend on the number of input commands specified.