CAPSTONE PROJECT

DEBUG_THUGS_01

GitHub Link:-

https://github.com/capstone-p3/Capstone

P 3 : Entry Queue Manager

- **Project :**You need to build an entry queue manager for a stadium. There are N entry gates. As people keep coming in, they can line up in any of the queues or switch queues anytime if they think that will get them a quicker entry. The entry queue manager helps them in making that decision by suggesting (i) the time for the last person in the queue to enter through any gate (assume that it takes p mins to enter any gate), (ii) the particular queue number(s) in the line who should be switching. The queue manager must be designed to minimize the time for M people to enter the stadium. Each gate has an initial random assignment of M/2 people (a gate may not have anyone assigned).

Group Members:

- Meet Barasara – 202301176
- Manav Kalavadiya – 202301054
- Tapas Santoki – 202301088
- Karan Makasana – 202301053

Output:-

```
Enter the number of entry gates: 3
Enter the total number of attendees: 15
Enter the time for a single attendee to enter any gate (in minutes): 2
Gate 1: 6 minutes
Gate 2: 6 minutes
Gate 3: 2 minutes
Attendee number 8 arriving.
Suggested queue for attendee 8: 3
Enter the chosen queue for attendee 8 (or same as suggested: 3): 2
 Attendee number 8 entering the stadium from queue 2
Attendee number 9 arriving.
Suggested queue for attendee 9: 3
Enter the chosen queue for attendee 9 (or same as suggested: 3): 3
 Attendee number 9 entering the stadium from queue 3
Attendee number 10 arriving.
Suggested queue for attendee 10: 3
Enter the chosen queue for attendee 10 (or same as suggested: 3): 1
 Attendee number 10 entering the stadium from queue 1
Attendee number 11 arriving.
Suggested queue for attendee 11: 3
Enter the chosen queue for attendee 11 (or same as suggested: 3): 3
 Attendee number 11 entering the stadium from queue 3
Attendee number 12 arriving.
Suggested queue for attendee 12: 3
Enter the chosen queue for attendee 12 (or same as suggested: 3): 3
 Attendee number 12 entering the stadium from queue 3
Attendee number 13 arriving.
Suggested queue for attendee 13: 1
Enter the chosen queue for attendee 13 (or same as suggested: 1): 1
 Attendee number 13 entering the stadium from queue 1
Attendee number 14 arriving.
Suggested queue for attendee 14: 2
Enter the chosen queue for attendee 14 (or same as suggested: 2): 3
 Attendee number 14 entering the stadium from queue 3
Attendee number 15 arriving.
Suggested queue for attendee 15: 2
Enter the chosen queue for attendee 15 (or same as suggested: 2): 2
 Attendee number 15 entering the stadium from queue 2
Gate 1: 10 minutes
Gate 2: 10 minutes
Gate 3: 10 minutes
```

First we have to enter the number of gates (N) , total number of attendees (M) and the time for a single attendee to enter any gate (p (in minutes)) .

In this input we have entered total number of entry gate (N) = 3 ,

Total number of attendees (M) = 15 and the time for a single attendee to enter any gate (p) = 2 min.

Now random function has arranged first (M/2 = 7) attendees to randomly to each gate. Now we can see that waiting time for gate 1 and 2 is 6 minutes as 3 attendees are in queue of these gates.

For gate 3, the waiting time is 2 minutes as 1 attendee is in the queue . Now attendee 8 arrives and we have suggested gate number 3 because in gate 3, waiting time for the last attendee is the least . We have also provided him choice to enter his preferred gate .

According to the gate chosen by attendee, we have updated the queue and the time for last person to enter the queue . We can see that attendee 8 has chosen gate 2 so we have updated accordingly .

If user gives an invalid input then suggested gate will be given to him . Time for last attendee to enter the gate will be updated according to attendee’s choice for attendee 9 ,10 and so on .

Time for last attendee to enter through every gate will be displayed.

Pseudocode

```
##  Include necessary libraries.


## STRUCTURE ENTRYGATE QUEUE
// Members
  attendees: list of integers (represents attendee IDs)
  estimatedWaitTime: integer (represents estimated wait time in minutes)


## FUNCTION ESTIMATEWAITTIME(queueLength, p)
  // Simple wait time estimation (queue length * processing time per attendee)
  return queueLength * p
```

```
## FUNCTION SUGGESTSHOETESTQUEUE(queues)
  shortestQueue = 0
  shortestWaitTime = INFINITY
  for i = 0 to queues.length - 1
    if queues[i].estimatedWaitTime < shortestWaitTime
      shortestQueue = i
      shortestWaitTime = queues[i].estimatedWaitTime
    endif
    go for next iteration
  return shortestQueue

## FUNCTION RANDOM_GATE(N)
  // Generate a random integer between 0 and N-1
```

```
  return random(0, N-1)

## MAIN FUNCTION ()

// Initialize variables
N = number of entry gates
M = total number of attendees
p = time for a single attendee to enter any gate ( in minuutes )

gates = vector of size N (EntryGateQueue)

// Initialize entry gate queues with estimated wait time set to 0 initially
for i = 0 to N-1
  gates[i].estimatedWaitTime = 0
```

```
// Randomly assign half attendees to each gate
for i = 0 to M/2-1
  randomQueue = random_gate(N)
  gates[randomQueue].attendees.push_back(i + 1)

// display time for last attendee in queue for each gate after randomly assigning half attendees

for i = 0 to N - 1
print " Gate " , i+1 ,  estimateWaitTime(queues[i].attendees.size(), p) , " minutes "

// Update initial wait times for all queues
for i = 0 to N-1
  gates[i].estimatedWaitTime = estimateWaitTime(gates[i].attendees.size(), p)
```

```
// Simulate attendee arrival loop
for attendee = M/2 to M-1
  // Suggest shortest queue
  shortestQueue = suggestShortestQueue(gates)

// Display suggested queue for attendees
  print "Attendee number", attendee + 1, "arriving."
  print "Suggested queue:", shortestQueue + 1

 // Allow the attendee to choose a queue

input chosenQueue
```

```
if chosenQueue < 1 or chosenQueue > N
    print "Invalid queue chosen. Defaulting to suggested queue", chosenQueue + 1
// default chosen queue is suggested queue
 chosenQueue = shortestQueue


   // Update queue information and estimate time
  gates[chosenQueue - 1].attendees.push_back(attendee + 1)
  gates[chosenQueue - 1].estimatedWaitTime = estimateWaitTime(gates[chosenQueue - 1].attendees.size(),
p)


  print "Attendee number", attendee + 1, "entering the stadium from queue", chosenQueue


// Display final wait times
for i = 0 to N-1
```

```
  print "Gate", i + 1, ":", estimateWaitTime(gates[i].attendees.size(), p), "minutes"

 end main ()
```

TIME COMPLEXITIES

–TIME COMPLEXITIES–

1. estimateWaitTime(int queueLength, int p):

This function simply performs a multiplication (queueLength * p). Multiplication operations have constant time complexity O(1).

2. suggestShortestQueue(const vector<EntryGateQueue>& queues):

In the worst case scenario, this function might need to iterate through all queues in the queues vector to find the one with the shortest estimated wait time. The number of iterations is directly proportional to the size of the queues vector (number of queues, N).

Iterating through a vector has linear time complexity O(N) (in the worst case).

3. The time complexity of the function random_gate(int N) is constant time O(1).

4. Main Function Portions:

The main function performs various operations, so we will analyze the time complexity of specific code sections:

** Initializing queues:

Iterates through N queues to set their estimated wait time. This has a linear time complexity of O(N).

** Randomly assigning attendees (M/2 times):

This loop iterates M/2 times, but within the loop, constant time operations like random number generation and adding to a list occur.

The overall complexity is O (M/2) = O(M).

** Updating estimated wait time a fter initial assignment:

Similar to initializing queues, iterating through N queues leads to a linear time complexity of O(N).

** Attendee arrival loop (M/2-M iterations):

This loop iterates through the remaining attendees (M/2-M). Within the loop, various operations happen
Suggesting shortest queue: As discussed earlier, this has a worst-case complexity of O(N).

User input and updating queue: These are considered constant time operations nm

Printing messages: Constant time.

queues[chosenQueue - 1].attendees.push_back(attendee + 1) : This statement adds an element (attendee ID) to the end of a linked list (attendee).

The time complexity of adding to the end of a linked list is generally considered O(1) (constant time) on average.

queues[chosenQueue - 1].estimatedWaitTime = estimateWaitTime(queues[chosenQueue - 1].attendees.size(), p):

This calls the estimateWaitTime function, which has a constant time complexity of O(1) as discussed earlier (simple multiplication).

Since the dominant operation inside the loop can be suggestShortestQueue (O(N)), the overall loop complexity becomes O(M * N) in the worst case.

** Final wait times: iterating through N queues leads to linear time complexity O(N).

********** OVERALL TIME COMPLEXITY **********

When considering a typical scenario with a larger number of attendees (M) compared to queues (N), the overall complexity can be approximated as:

O(M).

However, if the number of queues (N) becomes very large compared to attendees (M), the final wait time calculation might become more significant ,

leading to an overall complexity closer to O(N).

Why Linked List instead of Vector??

Choosing between linked list and vectors:-

Memory Overhead:

Vectors store elements contiguously in memory, which can be slightly more memory-intensive compared to linked lists. This is because vectors might allocate extra unused space to accommodate future growth.

Linked lists only allocate memory for the nodes themselves, and the pointers within each node contribute minimally to the overall memory usage. If memory efficiency is a major concern, especially for large queues, a linked list might be preferable.

This reallocation process can be expensive and takes more time then a single append operation. If you need random access to elements by index or frequent insertions/deletions in the middle, a vector would be preferable due to its contiguous memory layout but in our code we don't need any middle element.

If memory usage is a critical concern and you only need insertion at the end with occasional removals from the front in other side . If contiguous memory is not present then you don't need to move all elements to insert any other attendee . Shifting all existing elements in the vector one position forward to make space for the new element at the front.

Summary :-

For a queue implementation, especially when using push_front (or similar enqueue at the front operations), a linked list offers a clear advantage due to its efficient constant time insertions at the beginning. This aligns well with the FIFO nature of queues. While vectors might be suitable for some queue scenarios, their inefficiency with push_front and if memory is not contiguous and we want to add more attendee then we need to move all element it makes them a less desirable choice in this specific case .