|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | Cairo University, Faculty of Engineering  Computer Engineering Department  Data Structures and Algorithms  CMPN102 | Spring 2020 | | |
| **Data Structures and Algorithms**  **Final Assessment Report** | |
| **Team Name: Team 7 - (3707 / 8-11)** | **Number of members:4** |
| **Email: muhab.hossam13@gmail.com** |  |
| **Muhab Hossam Eldin, ID:1180074**  **SaveRestaurant**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::RunSimulation( )   **Calls**:   * Order::SetFinishTime(int) * Order::GetID() * Order::GetArrTime() * Order::GetServTime() * Queue::dequeue() * GUI::PrintMessage()   **Function** **Logic** **description**:  Prompts the user to enter a string and creates a .txt file in the Output\_Files directory with the name of the given string, appends the restaurant’s info inside the file then closes the file.  **UpdateWaitingTime**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * Queue::getCounter() * PriorityQueue::getCount() * Queue::dequeue() * PriorityQueue::dequeue() * Queue::enqueue() * PriorityQueue::enqueue() * Restaurant::CalcPriority()   **Function** **Logic** **description**:  Increments the waiting time for each order by 1 every passing timestep where the order isn’t inservice, does that by dequeuing orders, incrementing the wait time then enqueuing them again and a special case is handled for the vip queue as it uses priority to enqueue so to prevent it from dequeuing the same order we enqueue into a temporary queue of orders, then dequeue into our primary queue. | |
| **Mazen Amr Ahmed, ID:1180005**  **UpdateUrgentOrders**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * No functions are called   **Function** **Logic** **description**:  This function basically just creates a new vip orders queue, and empties the old vip orders queue one by one checking for each vip order if it has spent enough time waiting to be turned into an urgent order. If so, it’s added to the urgent orders queue, and if not it is added to the new vip orders queue. After the old vip orders queue is emptied, it deletes it and sets the vip order queue pointer data member to point to the new one.  **AssignToCook**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * Restaurant::AssignOrder()   **Function** **Logic** **description**:  This function just acts as a manipulator of the AssignOrder functions as it calls it for each order with the different lists of cooks that can cook this order until all orders are either assigned or no cook is available to prepare them.  **LoadRestaurant**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  Bool: true if there was an error in loading the file and false otherwise.  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * Different load helper functions to load and initialize the different input types.   **Function** **Logic** **description**:  LoadRestaurant() asks the user for the name of the input file and then calls LoadRestaurant(ifstream&) which in turn calls different helper functions like LoadCooks and LoadEvents. LoadEvents calls different functions for different event types. All the load functions use the LoadValues template function which takes a list of pointers and initializes them with data from the load file. The load functions specific for the different event types as well as the LoadValues are all inline functions as they’re only separated into different functions for easier readability and management. All the load functions including the helper function return true if any error happens while loading. This goes up the stack similar to a catch/throw terminating the other load functions.  **AutoPromote**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No Return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * No functions are called   **Function** **Logic** **description**:  This function just goes through the orders at the front of the normal orders queue one by one checking if they have spent enough time waiting to be auto promoted. When the order is to be auto promoted its type is changed and it’s enqueued in the vip orders queue. Once the loop reaches an order that hasn’t spent enough time waiting it breaks since the queue is ordered by the time of arrival.  **Adham Ahmed Hassan, ID:1180068**  **Injury**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * std::rand() * Cook::SetIsInjured() * Cook::GetIsInjured()   **Function** **Logic** **description**:  Function begins by generating a random number then comparing this number to another one from loaded file if the random number generated is less than or equal the value from loaded file then this is considered the key for function operation, next we check the assigned cooks queue whether it is empty or not so as to dequeue from it in the next step if the queue isn’t empty and the cook isn’t already injured, we move to the next step where we check whether he is cooking or not. Then, we dequeue this cook from the queue, we apply the injury on him (we set the injury setter to true) we keep track of the count of injured cooks so we increment the counter used for this purpose. The injured cook’s speed would of course be affected so as the current order being served. So we get the order currently in preparation, reset its priority to a negative integer value where we double the finish time (as cook’s speed when injured decreases to half its value) and subtract the current time step to get the duration, this duration serves as the new priority then we make three moves:   1. We dequeue the order ahead of the queue so as not to be repeated twice. 2. We set the new finish time. 3. We enqueue the cook with a priority similar to this of the order.   **PromoteOrder**  **Member** **of**: Class Restaurant  **Inputs**:   1. Integer representing ID of the order to be promoted 2. Double representing money paid for promoting said order   **Returns**:  No return (void)  **Called** **By**:   * PromotionEvent::Execute()   **Calls**:   * Queue::toArray() * Order::SetType() * Order::SetWaitTime() * Order::SetMoney() * Order::GetMoney() * PriorityQueue::enqueue() * Restaurant::CalcPriority()   **Function** **Logic** **description**:  Function begins by creating an array normal which includes the normal orders (same contents of normal order queue). First we loop on the created array to find the order of passed ID required to be promoted then we keep a flag of its index in the array. We then keep dequeuing from the normal order queue till its empty (as if destroying this queue). So, right now we have an empty queue of normal orders and an array which acts as a copy of the queue of normal orders before dequeueing it, so, we loop on this array and keep enqueuing its elements in the normal order queue under a condition that it’s not the order to be promoted in other words its index isn’t equal to the flag we used to keep track of order to be promoted. Once this order is found we make a couple of modifications related to it:   1. We set its type to VIP (part of the promotion action). 2. We set its wait time to zero (as if we start a new time counter so that it doesn’t count the previous wait time of the order and turn it to an urgent order although it didn’t wait the whole required period). 3. We set its money (cost) as in this case to the order initial cost plus the extra money paid for promotion. 4. We use priority calculator to calculate its priority 5. Then we enqueue it in the vip queue with the priority calculated in the above step.   **Moaz Tarek Ali, ID:1180013**  **AssignOrder**  **Member** **of**: Class Restaurant  **Inputs**:   1. Order pointer 2. Queue pointer of cook pointers   **Returns**:  Bool: returns false if the cook list is empty  **Called** **By**:   * Restaurant::AssignToCook()   **Calls**:   * Queue::dequeue() * Queue::enqueue() * Cook::SetOrder() * Cook::SetIsCooking() * PriorityQueue::dequeue() * PriorityQueue::enqueue() * Order::SetServTime() * Order::SetFinishTime() * Order::SetStatus()   **Function** **Logic** **description**:  the function takes an order and a cook list first it checks if the cook list is empty, if it is it returns false which indicates that the order wasn’t assigned to a cook else it assigns the order to the first cook available in the given list and changes the cook and order status and dequeues the cook from its list and enqueues it in the assigned list and also dequeues the order from its waiting list and enqueue it in the InServiceQueue and sets it’s priority by negative the time at which the order is finished  **CheckFinishedOrders()**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * PriorityQueue::peekFront() * PriorityQueue::dequeue() * Queue::enqueue() * Cook::GetOrder() * Cook::GetIsInjured() * Cook::SetIsInjured() * Cook::NeedBreak() * Cook::SetBreakTimeend() * Cook::GetType() * Cook::SetIsResting() * Order::SetStatus() * Order::GetType()   **Function** **Logic** **description**:  The assigned cooks queue is a priority queue sorted by the finish time of the assigned order so I check the first order in the queue if it’s finish time equal to the current time step then the order is finished then dequeue the order from the inServiceQueue and enqueue it in the finishedQueue and then change it’s status to DONE then increment the number of orders then dequeue the cook from assignedCooks and check if the cook is injured if he was injured then enqueue the cook in the restingCooks with priority equal to rest injury steps else if the cook need a break then enqueue the cook in the restingCooks with priority equal to the break time of the cook else enqueue the cook in its Queue then take the following order and check it  **CheckCookBreaks()**  **Member** **of**: Class Restaurant  **Inputs**:  No inputs  **Returns**:  No return (void)  **Called** **By**:   * Restaurant::InteractiveMode() * Restaurant::StepByStepMode() * Restaurant::SilentMode()   **Calls**:   * PriorityQueue::peekFront() * PriorityQueue::dequeue() * Queue::enqueue() * Cook::GetBreakTimeEnd() * Cook::GetType()   **Function** **Logic** **description**:  First check if the resting cook is empty then enters the while loop and check the first cook in the resting cooks as they are arranged by their break time end then if it is equal to the current time step then dequeue it from the resting cooks and then enqueue the cook in a cooks queue according to its type and the take the following cook | |