|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | Cairo University, Faculty of Engineering  Computer Engineering Department  Data Structures and Algorithms  CMP102/CMPN102 | Spring 2020 | | |
| **Data Structures and Algorithms**  **Final Assessment Report** | |
| **Team Name: 1-4\_3708\_T1** | **Number of members: 4** |
| **Email: alisaidelnaggar@gmail.com** |  |
| **Section1: Ali said El-Naggar, 1180036**  **Function1\_PrintInfoCurrentTime**  **Input:**  Int currentTimeStep: integer number of the current time steps  **Return:**  Void  **Call to:**  Restaurant::MainSimulator()  **Calls from:**  Restaurant::CalculatingNumberOfOrdersDone(int\*Arrayofnumber)  BaseList<T>::GetCount()  PriorityQueueMax<T>::GetCount()  PriorityQueueMin<T>::GetCount()  Queue<T>::GetCount()  Order::GetID()  Cook::GetID() const  Cook::GetType() const  Cook::getMakingOrder() const  Order::GetType() const  GUI::PrintMessageML(string\*msg,int lines) const  Queue<T>::Dequeue(T & frntEntry)  Std::to\_string(int\_val)  **Function Logic description:**  This function prints the information that happened in the input Time Step, which are the input Time Step, The Number of waiting orders of each type, the Number of available cooks of each type, also cooks that were assigned to an order, where when a cook is assigned to an order, this cook will be enqueued in the Assigned Queue, such that we will dequeue all these cooks and print the cook ID and type, also the ID and type of the assigned order, and it also prints the Number of Orders Served so far of each type.  **Function2\_CalculatingNumberofOrderDone**  **Input:**  Int\*Arrayofnumber: array of number pointer  **Return:**  Void  **Call to:**  Restaurant::PrintInfoCurrentTime(int CurrentTimeStep)  **Calls from:**  Order::GetType() const  Queue<T>::Queue()  Queue<T>::Dequeue(T & frntEntry)  Queue<T>::Enqueue(T & newEntry)  **Function Logic description:**  This function calculates the Number of each type of the Done orders by looping over the OrdersAllDone Queue by dequeuing it in a temp queue, then add one to its type count, and then after dequeuing OrdersAllDone to temp, we enqueue them again to OrdersAllDone with the Same Order.  **Function3\_MainSimulator**  **Input:**  PROG\_MODE Mode : the Mode of the program (silent, interactive or step by step)  **Return:**  Void  **Call to:**  Restaurant::MainSimulator()  **Calls from:**  Restaurant::AssigningOrders(int CurrentTimeStep)  Restaurant::CheckAutoPro(int CurrentTimeStep)  Restaurant::CheckInjuredCooks(int CurrentTimeStep)  Restaurant::CheckUrgency(int CurrentTimeStep)  Restaurant::ExecuteEvents(int CurrentTimeStep)  Restaurant::FillDrawingList()  Restaurant::PrintInfoCurrentTime(int CurrentTimeStep)  Restaurant::SaveFile()  Restaurant::ReadFromFile()  Restaurant::UpdateCooksandOrdersstatus(int CurrentTimeStep)  GUI::UpdateInterface()  GUI::ResetDrawingList()  GUI::waitForClick() const  BaseList<T>::isEmpty() const  PriorityQueueMin<T>:: isEmpty() const  PriorityQueueMax<T>:: isEmpty() const  Queue:: isEmpty() const  Std::to\_string  **Function logic description:**  this function is the Main function of this project, first it read an input file from the User which contain information about the restaurant, then it execute the event which corresponds to the current time step, then it calls AssigningOrders that assign Orders to the available cooks, after that this function generate a random number, where if this number is less than the Injure probability, it will call CheckInjuredCooks, which injure the first Busy cook if there is any, after that it check if there is an urgent order by calling check Urgency, then it does the same to normal orders by checking the auto promotion by calling CheckAutoProm, then it Updates the Cooks and Order Status and handle the finishing and breaking effect by calling UpdateCooksandOrdersstatus, after that it checks the mode of the program, where if it is not in silent mode, it will call printInfoCurrentTime and update the interface to update the GUI, if it is silent it won’t print anything, and if it is in interactive it will wait for the mouse click, and if it is in step by step it will call sleep with freezes the program for a minute, then it increase the timestep by 1 and repeat all the previous, until there is no waiting or serving orders and exit the while loop, after that it will save the output file by calling SaveFile.  **Function4\_UpdateCookandOrdersstatus**  **(update Busy, Break and rest cooks)**  **Input:**  Int CurrentTimeStep: integer current time step  **Return:**  Void  **Call to:**  Restaurant:: MainSimulator()  **Calls from:**  Cook::GetBreakEndTime()  Order::GetFinishTime()  Restaurant::ReturnCookToRightList(Cook \*pCook)  Restaurant::UpdateCook(Cook\* pCook,int CurrentTimeStep)  Queue<T>::dequeue(const T & newEntry)  Cook::getMakingORder() Const  PriorityQueueMin<T>::peekFront(T & frntEntry) const  **Function logic description**  If a working cook finished his order, this function will dequeue it form the list, as the first cook, will have the min finishTime, and call updateCook on it, and If a cook in the rest or Break finished his rest or Break, it will be dequeued and call ReturnCookToRightList which will return Cook to the right list.  **Function5\_ReturnCookToRightList**  **Input:**  Cook\* pCook: a pointer to a Cook  **Return:**  Void  **Call to:**  Restaurant:: UpdateCook(Cook \*pCook, int CurrentTimeStep)  Restaurant:: UpdateCooksandOrdersstatus(int CurrentTimeStep)  **Calls from:**  Cook:: GetType() const  Cook:: SetStausOfCook(Cook\_Status)  Queue<T> :: enqueue (const T & newEntry)  **Function logic description:**  It will Call SetStausOfCook and make this cook available, and check the type of this cook and add it to its right list.  **Section2: Ahmed Osama, 1180075**  **Function1\_CheckedInjuredCooks**  **Member of:** Class Cook  **Inputs:**  Int CurrentTimeStep: integer number of the current steps  Float R: A float random number generated  **Returns:**  Void  **Call To:**  Restaurant::MainSimulator()  **Calls from “injured cook”**  From order:  getArrivalTime()  getFinishTime()  getnumberofdishes()  getServingTime()  getSpeed()  getWaitTime()  setFinishTime (int x)  setServingTime (int x)  Dequeue & enqueue from priority queue min<T>  From cook:  getMakingOrder()  **Function Logic description:**  **Logic:**  It checks if the random number bigger than the injury probability to perform the function by making busy cooking finish the order after finishing it the cook will be in injury cook list  **Method:**  The function will be performed by dequeue the busy cook from working cook list then the busy cook will be assigned to make an order. To calculate the new serving time after injury first calculate time step of serving add arrival time of the order to the order waiting time, second to calculate how many time steps the cook worked subtract the current time step from the serving time +1 that we calculated before third we calculate the number of done & remaining dishes, done dishes by multiply time steps that the cook worked by the speed of the busy cook then the remaining dishes by subtracting the number of dishes of the order from number of done dishes, Now we will set the speed of the cook to half after being injured then the new serving time will be number of worked time steps plus the remaining dishes divided the new speed of injury cook.  Final step is to set the serving & finishing time, serving time will be the last serving time we calculated and finish time will be arrival time plus serving time plus waiting time of the order all in getters then enqueue to the injury cook list.  **Function2\_updateCooksAndOrdersStatus**  **(adding the injury Cooks Status only)**  **Input:**  Int currentTimeStep: integer number of the current time steps  **Return:**  Void  **Call to:**  MainSimulator() from restaurant class  **Calls from:**  **Cook:**  getMakingOrder()  getInjuryRest()  setStatusOfCook(Cook\_status)  **Order:**  getFinishTime()  dequeue & enqueue from priorityQueueMin<T>  **Function Logic description:**  Changing the cook status from injury to break | |
| **Function3\_SaveFile()**  **Input:**  No inputs  **Return:**  Void  **Call to:**  Restaurant::MainSimulator()  **Calls from:**  Order::GetArrivalTime()  Order::GetFinishTime()  Order::GetID()  Order::GetServingTime()  Order::GetType() const  Order::GetWaitTime()  Order::getAutoPromoted()  GUI::GetString() const  GUI::PrintMessage(string msg) const  GUI::UpdateInterface()  Queue<T>::toArray(int & count)  Std::basic\_ofstream::Is\_open() const  Std::basic\_ofstream::close() const  **Function Logic description:**  Initialize a counter equal zero then make array of pointers called done order by calling to array function to make an array of id , ft , at , wt and st we will fill the array by making a for loop with I less than the size of array (counter) in the loop will make a value for id , ft , at , wt and st by a pointer from DoneOrder then another for loop for making counts for the cooks by making pointer from DoneOrder GetType() for each cook type then print the tables of output liked the given format.  **Section3: Mohamed Amr Afifi, 1180062**  **Function1\_Assigingorders**  **Member of:** Class Restaurant  **Inputs:**  Int CurrentTimeStep: integer indicating the current time step  **Returns:**  Void  **Calls from “Assigning Order”**  Restaurant::Assignorder(Cook \* pCook, Order \* pOrder, int CurrentTimeStep, Cook\_Status CookStat)  Baselist<T>::RemoveFirst(T &peekfrnt)  Baselist<T>::ReturnFirst( T &peekfrnt)  PriorityQueuemax<T>::dequeue( T &peekfrnt)  Queue<T>::dequeue(T & frntEntry)  Queue<T> :: isEmpty() const  PriorityQueueMax<T>::peekFront(T & frntEntry) const  Queue<T>::peekFront(T & frntEntry) const  **Calls to “Assigning Order”**  Restaurant::MainSimulator()  **Function Logic description:**  **Logic:**  The function is responsible for assigning order to the cooks in the order of the VIP ( first priority ),then Vegan  ( second priority ) ,then Normal ( third priority )  **Method:**  At first of the function it checks if the VIP order list has order or not and (&) it also checks the availability of the VIP cook list or (||) the availability of Normal Cook list or (||) the availability of the Vegan cook list if any of these cooks is available the cook will no longer be available in its corresponding cook list then it will call the function of assignorder() that makes the actions of assigning the order itself . the next priority is for the Vegan order So it checks if the vegan cook list has order or not and (&) it checks the availability of the vegan cook list only because vegan orders could be done only by vegan cooks so if a vegan cook is found available then it removes this cook from the available cook lists and assignorder() to complete the other actions of assigning the order itself . The last priority comes to Normal Order it checks if there’s waiting normal orders or not and (&) it checks the availability of the Normal cook list or (||) the availability of the Vip Cook if any was found available then it will be assigned to the corresponding cook and assign order will be called to complete the other actions of assigning an order such as setting the serving time , arrival time, computing finishtime  **Function2\_Assignorder**  **Member of:** Class Restaurant  **Inputs:**  **int current time step :** integer indicating the current time step .  **Cook\***: The cook that the order will be assigned to.  **Order\***: the order required to be assigned .  **Cook\_status**: the Cook status (to check urgency) default value is “Busy”  **Returns:**  Void  **Calls from “Assign Order”**  Order::GetArrivalTime()  Order::GetFinishTime()  Order::GetNumberOfDishes()  Order::GetServingTime()  Cook::GetSpeed() const  Order ::GetWaitTime()  Order::SetFinishTime(int x)  Order::SetServingTime(int x)  Cook::SetStausOfCook(Cook\_Status Cookstat)  Order::SetWaitTime(int x)  Queue<T>::enqueue(const T & newEntry)  PriorityQueueMin<T>::enqueue(const T & newEntry, int p)  Cook::setMakingOrder(Order \*)  Order::setStatus(ORD\_STATUS s)  **Calls from “Assigning Order”**  Restaurant::AssigningOrders(int CurrentTimeStep)  Restaurant ::CheckUrgency(int CurrentTimestep)  **Function Logic description:**  **Logic:’**  The function is responsible for assigning order actions such as setting the arrival time and setting the number of dishes and adding the number of dishes set waiting time  **Method:**  First of all the function is responsible to connecting the order to the cook and changing the status of the cook ,then it set the waiting time of the order according to the time of assignment,.then the function checks the status of the cook if urgent it will set the speed for the of serving the cook to the speed of the cook divided by 2 otherwise it sets the normal speed ,then it sets the finish time of the order and also setting the status of the order to serving . Lastly the function enqueues the cook that the order is assigned and enqueue the order of the to the serving list and to assign list that is needed for drawing  **Function3\_“UpdateCook”**    **Member of:** Class Restaurant  **Inputs:**  **Int CurrentTimeStep**: integer indicating the current time step  **Cook\* cook** : the cook required to be updated  **Returns:**  Void  **Calls from “UpdateCook ”**  Cook..GetBreakEndTime()  Cook::GetBreakTime() const  Cook::GetCookStatus() const  Cook ::GetFinishedOrders()  Cook::SetBreakEndTime(int x)  Cook ::SetFinishedOrders(int x)  Cook ::SetStausOfCook(Cook\_Status)  Restaurant::ReturnCookToRightList(Cook \* pCook)  PriorityQueueMin<T>::dequeue(T & frntEntry)  Queue<T>::enqueue(const T & newEntry)  PriorityQueueMin<T>::enqueue(const T & newEntry, int p)  Cook::getBreakAfterN()  Cook::getInjuryRest()  Cook ::getMakingOrder() const  Cook::setMakingOrder(Order \*)  Order::setStatus(ORD\_STATUS s)  **Calls to “UpdateCook ”**  Restaurant ::UpdateCooksandOrdersstatus(int CurrentTimeStep)  **Function Logic description:**  **Logic:**  The function is responsible for updating the cook finished order and updating the orders done and removing the order from serving list  **Method:**  The function first the function gets the order of the cook passed to the function and changes the order status of the finished order to done then it enqueues the finished order to the done orders and dequeuing the finished order from the serving orders and then he updates the number of finished orders by adding this finished order to the number of total finished orders and then it checks the order status if he is injured in order to be sent to the rest period then the functions checks if this cook is needed to be sent to break or not after finishing this order and if yes it sets his status to break after making all this processes the cook is sent to the (returnCooktorightlist) to see if this cook needs to go back to his corresponding list  **Section 4: Abdelrhman Mahmoud Hosny, 1180057**  **Function 1 “ Set Cook Speed”**  **Member of :** Class Cook  **Inputs:**   * SpdMax , SpdMin   **Returns :**   * None   **Called To:**   * Cook(int ID, int minSpd, int maxSpd, int minBrk, int maxBrk) (Cook)   **Called From:**   * rand(void)   **Function Logic description:**   * The function makes use of the mod properties and randomization to make it so that the cooks have a speed that’s random and between the two numbers SpdMax and SpeedMin * Formula is Speed = SpdMin + randomNumber mod (SpdMax-SpdMin + 1)   **N.B.** Set Break speed works the same way and has same call hierarchy | |
| **Function 2 “Check Auto Promotion”**  **Member of:** Class Restaurant  **Inputs:**   * CurrentTimeStep   **Returns:**   * None   **Called To:**   * MainSimulator(PROG\_MODE Mode) (Restaurant)   **Called From:**   * GetArrivalTime() (Order) * PriorityEquation(Order \* Ord) (Restaurant) * RemoveFirst(T & frntEntry) (BaseList<T>) * ReturnFirst(T & frntEntry) const (BaseList<T>) * enqueue(const T & newEntry, int p) (PriorityQueueMax<T>) * setAutoPromoted(bool K) (Order) * setType(ORD\_TYPE s) (Order)   **Function Logic description:**   * Gets repeated every time step and compares the first order in Normal Order Queue (FIFO), if the first order waits more than the promotion variable It gets moved to VIP Priority Queue   **Function 3 “CheckUrgency”**  **Member of:** Class Restaurant  **Inputs:**   * Current Time step   **Returns:**   * None   **Called To:**   * MainSimulator(PROG\_MODE Mode) (Restaurant)   **Called From:**   * AssignOrder(Cook \* pCook, Order \* pOrder, int CurrentTimeStep, Cook\_Status CookStat) (Restaurant) * GetArrivalTime() (Order) * GetCount() (BaseList<T>) & ( PriorityQueueMax<T>) * InsertLast(const T&NewEntry)(BaseList<T>) * PriorityEquation(Order \* Ord) (Restaurant) * RemoveFirst(T & frntEntry) (BaseList<T>) * dequeue(T & frntEntry) (PriorityQueueMax<T>) * dequeue(T & frntEntry) (PriorityQueueMin<T>) * enqueue(const T & newEntry, int p) (PriorityQueueMax<T>) * isEmpty() const (PriorityQueueMin<T>)   **Function Logic description:**   * Gets repeated every timestep and compares the wait time of each order if it is higher than UrgentVariable , It forces one of the cooks from break to take this order and if no cooks are on break ,It checks the Resting Cooks List and if not empty forces one to go and make the order but at half speed | |