Seth Cram

CS121

Make sure to use standard namespace on every file. In the listQ.h interface program, create a LinkedList class. As global variables above the class, fashion two integers to store the current x and y coordinates of each node that is dequeued from the list before it’s deleted. Within that class create a private node struct that contains a node pointer, variable to store each maze character, an X-coordinate variable, and a Y-coordinate variable. These coordinate variables will be used later on to keep track of the coordinates of each node enqueued to the list. Next, make private node pointers linked to the start and end of the list, so the list can function as a queue. Then, create a public linked list constructor function that sets start to NULL and end to start. Finally, create the following functions also in the public portion of the list class, void AddNodeToEnd( char, int, int ), void DeleteNode(), void PrintNodes(), and int Size(). AddNodeToEnd takes the above argument to pass in the maze character, x-coordinate, and y-coordinate the maze character was found at. The Size function needs the integer return type in order to output the size of the list. Then fabricate the listQ.cpp implementation file to define the function prototypes in the public section of the list class. Make sure to include the previous listQ.h file in order to gain access to the list class. Also specify all the previous functions as members of the LinkedList class. Make sure to deal with the list in a dynamic fashion so use (->) instead of (.) here.

Now, onto the queue. Create another interface file called queueL.h that’s in the same folder as the other two, and include listQ.cpp to get access to the list class and its defined functions. Then, code a private LinkedList object, a pointer to a pointer of a variable, width and height integers, startX and startY integers, endX and endY integers, and a character to represent every character used in the maze and an empty character. Next, in the public section write void EnQueue( char, int, int ), void DeQueue(), bool IsEmpty(), void PrintN(), void ShowMaze(), void StartOfNewMaze(), bool WalkMaze( bool ), and a fully defined void Fill() function. A fully defined fill function is required to give the rest of the functions in the queue class access to those values stored in the array. Enqueue needs to take the above argument in order to pass the maze character and its position onto the AddNodeToEnd function. IsEmpty() requires a boolean return type in order to pass on whether the linked list is indeed empty or not. The WalkMaze function needs to return a boolean in order to specify whether the maze is actually solvable or not. That same function takes a boolean as an argument to help in the termination of the recursive search for the goal this function searches for in the maze. The fill function needs to read in all the maze values from the chosen file, using user input and if statements to choose the file, and a dynamically allocated array utilizing the character pointer to a pointer specified in the private part of the queue class. In order to read in values from a file the fstream library needs to be included at the top of the file. Reading and storing the width and height values before reading in the maze characters is also necessary. When allocating data for the array, we need to set a surrounding border of empty values in order to compare characters when enqueueing nodes later on.

Onto the queue implementation file, queueL.cpp. Make sure this includes queueL.h to give us access to everything we’ve been working on up until now. When defining the EnQueue function, pass the given values into the AddNodeToEnd function. When dequeueing make use of the DeleteNode function in the list class. IsEmpty should utilize the Size function, and compare that to zero. PrintN() includes the PrintNodes function. The ShowMaze function just includes a for and nested for loop that output the current values in the maze array.

WalkMaze has to DeQueue a node from the list, set new integer coordinates to the ones of the dequeued node, and then mark the current maze array spot as visited. This function then goes on to check if the current position is next to the goal position, and if not true goes on to enqueue only the characters below, above, to the right, and left of the current position that are moveable tiles. Next, check using the IsEmpty function to make sure the maze is still solvable, if it isn’t then set the value passed to the function we are in, WalkMaze, to true. The coming if statement should check if the value passed to this function is true, and if so, returns false, so the maze is unsolvable. Lastly, check to see if WalkMaze is true, and if so return true, effectively beginning the recursion it’ll go through when solving the maze.

In StartOfNewMaze() as a default, set the value passed to WalkMaze as false. Go on to search through the already established maze array and set the start and end point. Use a for and nested for loop to search for these points. Then, enqueue the position of the starting point., and check if the WalkMaze function is solvable or not, effectively calling it to try and solve the maze.

For our final file, include queueL.cpp to get access to all our functions and classes thus far. Manufacture an int main that creates a choice variable and allows the user to pick how many mazes they want to solve. In a for loop allocate memory for a queue class object, and use that object to call the fill, ShowMaze, StartOfNewMaze, and ShowMaze function. This should display the maze right after it’s read into the array, say if it’s solvable or not, and then display the maze again this time showing the characters that the queue visited in its path to the goal.