CS29003 ALGORITHMS LABORATORY
Assignment 2
Last Date of Submission: 26 – July – 2018

Suppose that you have an $n \times n$ maze of rooms. Each adjacent pair of rooms has a door that allows passage between the rooms. At some point of time some of the doors are locked, the rest are open. Your task is to determine whether there exists a route from a given room (s, t) to another room (u, v) through the open doors. The idea is to start a search at room no (s, t), then investigate rooms $(s_1, t_1), \ldots, (s_k, t_k)$ that can be reached from (s, t) and then those rooms that can be reached from each (s_i, t_i), and so on. There are many strategies that you can adopt for this search. We enumerate a few possibilities below. Note that there is no need to revisit a room during the search. You can maintain an $n \times n$ array of flags in order to keep track of the rooms that are already visited.

Strategy 1 (Use of Stacks): This strategy tries to explore the maze by going as far as possible through open doors. As soon as it encounters a blockage (no open doors), it backtracks and continues this process (of going as far as possible through open doors) again without revisiting any of the previous rooms. You can use a stack to implement this strategy. Initially push (s, t) to the empty stack. Subsequently, as long as the stack is not empty, consider the room (x, y) at the top of the stack. If (x, y) has a yet unvisited neighboring room, push that room at the top of the stack. If (x, y) does not have an unvisited neighboring room, pop (x, y) out of the stack. If during these operations, the desired room (u, v) ever appears at (the top of) the stack, then a route from (s, t) to (u, v) is detected. If the search completes (that is, the stack becomes empty) without ever having (u, v) in the stack, then there is no $(s, t) - (u, v)$ path.

Strategy 2 (Use of Queues): This strategy tries to explore the maze by first exploring all the immediate neighboring rooms available at any point, and then explores immediate neighbors of these rooms, and so on. You can use a queue to implement this strategy. Initially enqueue (s, t) to an empty queue. Subsequently, as long as the queue is not empty, look at the room (x, y) at the front of the queue. If $(x, y) = (u, v)$, then report success and return. Else dequeue (x, y) from the front and enqueue all unvisited neighboring rooms at the back of the queue. If the search stops (that is, the queue becomes empty) without ever having (u, v) at the front of the queue, report failure.

Hints for the Data Structure: First, you need to be able to keep track of the rooms and the door status (open/close). Note that if we visualize the maze in 2-D, there are two types of doors: horizontal and vertical. Create an $(n + 1) \times n$ array H for the horizontal doors and $n \times (n + 1)$ array V for the vertical doors. For each of the elements of the array, let an assignment of 0 indicate that the door is closed, and an assignment of 1 indicate that the door is open. In the Figure attached, for a 4×4 maze, sample H and V inputs are shown (**Figure: a,b,c**). Note that all the boundaries have been denoted with closed doors. You can assume the maximum value of n to be 20 and declare static arrays H and V.

Once the status of the doors has been initialized, during the execution of the algorithm, you will need an efficient way of reaching out to the open neighbors of any room. For each room, you can store the neighboring rooms with open doors as a linked list, which is populated from arrays H and V, and the head pointers to all the individual linked lists can be stored in a 2-D array of size $n \times n$. Note that for each room, there are atmost 4 neighboring rooms. **Figure: d,e** shows an example room with open and

closed doors as read from H and V, and the corresponding linked list storing the neighboring rooms with open doors.

Part 1. Create the data structure to store the maze

In this part, you will read n, H and V from the user. This information is sufficient to create the maze. You can then print it nicely as shown in the sample output. Define the following function that prints the maze.

```
void printmaze(int H[][20], int V[][21], int n);
```

You can represent rooms using the following data structure

```
typedef struct rm {
  int hInd;
  int vInd;
} room;
```

Now, use a 2D static array $maze$ such that (i, j)th element stores the head pointer of the linked list consisting of open neighbors for the room (i, j). Function $createmaze()$ below populates this.

```
struct node {
   room data;
   struct node *next;
};
typedef struct node node, *list;
list maze[20][20];
void createmaze(list maze[][20], int n, int H[][20], int V[][21]);
```

$createmaze()$ should also print the linked list representation as per the sample output.

Part 2. Implementation of stack

We require you to implement the stack using linked list. To implement the stack using linked list, note that the head of the linked list corresponds to the top of the stack, and the head pointer will suffice for all the operations. A *push* operation corresponds to an insertion at the head, while a *pop* corresponds to deletion at the head. Write the following functions.

```
typedef struct {
    struct node *head ;
} STACK ;
void init(STACK *s);//initializes the head pointer
int isempty(STACK s);//prints 1 if the stack is empty, 0 otherwise
void push(STACK *s, room data);
room pop(STACK *s);
```

Part 3. Implementation of the search strategy 1 using stack

The function takes the maze data structure, start and end rooms and uses strategy 1 to search if there is a path from the start room to the end room.

```
int strategy1(list maze[][20], int n, room start, room end);
```

Function returns 1 or 0 depending on whether or not there is a path.

Part 4. Implementation of queue

For queue, you need to store both the head and tail pointers to the linked list, which you will need for *dequeue* and *enqueue* operations, respectively. Write the following functions.

```
typedef struct {
    struct node *front, *rear ;
} QUEUE ;
void init(QUEUE *qP);//initializes the front and rear pointers
int isempty(QUEUE qP);//prints 1 if the queue is emptry, else 0
void enqueue(QUEUE *qP, room data);
room dequeue(QUEUE *qP);
```

Part 5. Implementation of the search strategy 2 using queue

The function takes the maze data structure, start and end rooms and uses strategy 2 to search if there is a path from the start toom to the end room.

```
int strategy2(list maze[][20], int n, room start, room end);
```

Function returns 1 or 0 depending on whether or not there is a path.

Main Function

Main() function does the following:

1. Reads n, H and V from the user in this order. It then reads the indices of the start and end rooms. It also prints the read values as per the sample output. In this assignment, you should read these values from an input text file (instead of typing these on terminal). Sample values are given in $input.txt$. To read values from $input.txt$, you can use the following:

   ```
   .\a.out < input.txt
   ```

   Note that this behaves in exactly the same way as if the values are input from the keyboard, and you do not need to use file handling.

2. Calls $printmaze()$ to print the maze
3. Calls $createmaze()$ to initialize and print the data structure for the maze
4. Calls $strategy1()$ and $strategy2()$ respectively to print if there is a path following these strategies.

Submission

Submit a single C/C++source file, preferably with a name $ass2.c$. Do not use global variables.

Sample Output

```
Enter the value of n
4
Enter the horizontal doors H
0 0 0 0
```

```
1 1 1 1
1 1 1 0
0 1 0 0
0 0 0 0
Enter the vertical doors V
0 0 0 0 0
0 1 0 1 0
0 0 1 1 0
0 1 1 1 0
Enter the indices of the start room s,t
3 0
Enter the indices of the end room u,v
0 2
The maze looks like:
    +---+---+---+---+
    |   |   |   |   |
    +   +   +   +   +
    |       |       |
    +   +   +   +---+
    |   |           |
    +---+   +---+---+
    |               |
    +---+---+---+---+
The linked list representation looks like:
(0,0)::-->(1,0)
(0,1)::-->(1,1)
(0,2)::-->(1,2)
(0,3)::-->(1,3)
(1,0)::-->(1,1)-->(0,0)-->(2,0)
(1,1)::-->(1,0)-->(0,1)-->(2,1)
(1,2)::-->(1,3)-->(0,2)-->(2,2)
(1,3)::-->(1,2)-->(0,3)
(2,0)::-->(1,0)
(2,1)::-->(2,2)-->(1,1)-->(3,1)
(2,2)::-->(2,1)-->(2,3)-->(1,2)
(2,3)::-->(2,2)
(3,0)::-->(3,1)
(3,1)::-->(3,0)-->(3,2)-->(2,1)
(3,2)::-->(3,1)-->(3,3)
(3,3)::-->(3,2)
Using Strategy 1 ...
A path is found using strategy 1 from (3,0) to (0,2)
Using Strategy 2 ...
A path is found using strategy 2 from (3,0) to (0,2)
```