EE441- Programming Assignment 2

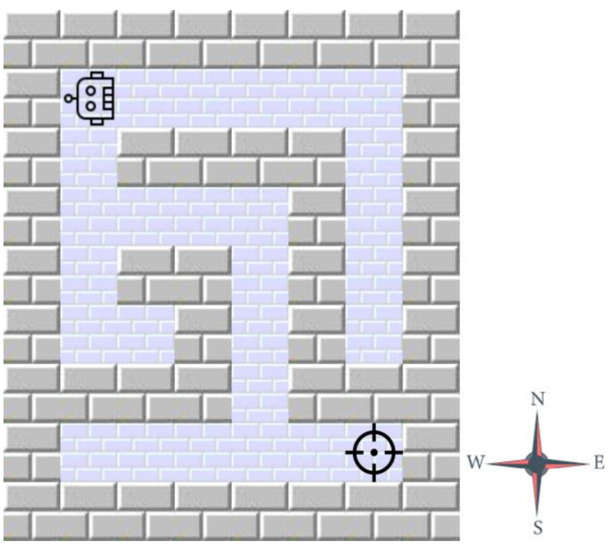
Due Date: 07.12.2022, 23:55

For your questions: Ferhat Gölbol – ferhatg@metu.edu.tr

This assignment consists of two parts. You are going to create a separate Code::Blocks project for both parts. Don't forget to write comments to your code as they are also graded.

Part 1 – Maze Solver [60 points]

A maze is a path or collection of paths, typically from a start point to a goal. In this part, you are going to implement a text-based maze class and a solver for it.




```
########
#E.....#
#.####.#
#....#.#
#.##.#.#
#..#.#.#
####.###
#.....T#
########
```

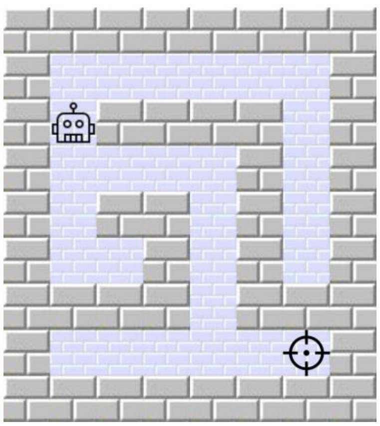
Figure 1: A maze and its text-based representation.The agent is facing East.

In this implementation, the terms "left", "right", "forward" and "back" are used for the directions with respect to the agent's current orientation; while "west", "north", "east" and "south" are used for absolute directions. In the text-based representation,

- walls are denoted with '#',
- empty cells are denoted with '.',
- target is denoted with 'T',
- current location is denoted with 'W', 'N', 'E' or 'S', depending upon the orientation.

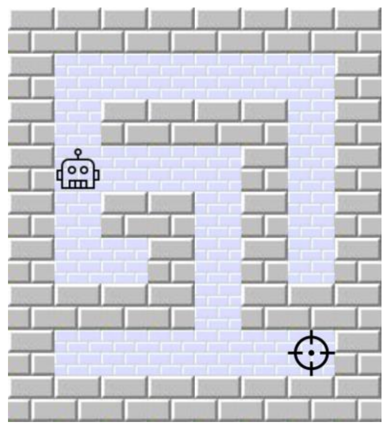
1) Implement a Maze class. Current state of the maze needs to be stored in a 2D char array. Maze dimensions can be up to 20x20 blocks. Your class needs to have the following members at minimum:

- A constructor that reads the initial state from a file (see the example in the appendix) and a copy constructor. First line of the file contains number of rows and columns, and following lines contain the text-based representation of maze state.
- `can_move_left`, `can_move_forward`, `can_move_right` and `can_move_back` functions that return true if the agent can move left, forward, right or back respectively, otherwise return false. These functions should not change the maze state.
- `move_left`, `move_forward`, `move_right` and `move_back` functions that turn and move the agent left, forward, right or back respectively, if possible. These functions should also confirm validity of the move. For example, if `move_left` is called while the agent cannot move left, maze state should not be changed.
- `print_state` function that prints the text-based representation of the current maze state.
- `is_solved` function that returns true if the agent is at the target, false otherwise.



```
########
#......#
#S####.#
#....#.#
#.##.#.#
#..#.#.#
####.###
#.....T#
########
```

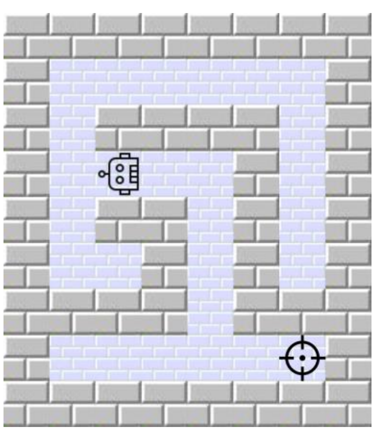
(a)



```
########
#......#
#.####.#
#S...#.#
#.##.#.#
#..#.#.#
####.###
#.....T#
########
```

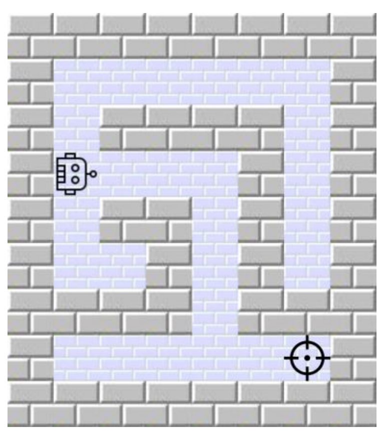
(b)

Figure 2: (a) Maze state after move_right() command at initial state in Fig. 1. At this state, can_move_forward() returns true, while can_move_left() and can_move_right() return false. (b) Maze state after move_forward() command at state (a). At this state, can_move_forward() and can_move_left() return true, while can_move_right() returns false.



```
########
#......#
#.####.#
#.E..#.#
#.##.#.#
#..#.#.#
####.###
#.....T#
########
```

(c)



```
########
#......#
#.####.#
#W...#.#
#.##.#.#
#..#.#.#
####.###
#.....T#
########
```

(d)

Figure 3: (c) Maze state after move_left() command at state (b). (d) Maze state after move_back() command at state (c).

2) Implement a mixed StackQueue<T> template class. Your class needs to have

- push_front and push_rear methods that store a new element at the front/rear of the storage;
- pop_front and pop_rear methods that remove and return the element at the front/rear of the storage,
- peek_front method that returns the element at front without modifying the storage
- print_elements method that prints all the stored elements from rear to front. For this method, you can assume that class T has a proper implementation of &operator<<, that is, "T x; cout << x;" prints the element x.

Your implementation should be able to store at least 1024 elements.

3) Implement the 2-pass maze solver algorithm, "The Left Hand Rule", explained below.

- Start with a maze to be solved and an empty StackQueue<char>. The characters 'L', 'F', 'R' and 'B' will be used for left turn, forward motion, right turn and backwards motion, respectively.
- In the first pass, use the StackQueue as a stack. Repeat the following steps until the maze is solved:
    - If the agent can move left, move left. If the last element in the stack is not 'B', push an 'L' to the stack, denoting left turn. If the last element is 'B', see (*) below.
    - If the agent cannot move left but can move forward, move forward. If the last element in the stack is not 'B', push an 'F'. If the last element is 'B', see (*) below.
    - If the agent cannot move left or forward but can move right, move right. If the last element in the stack is not 'B', push an 'R'. If the last element is 'B', see (*) below.
    - If the agent cannot move left, forward, or right; move back. Push a 'B'.

    * Observe the following: if the agent moves left, then back, then left again, the resulting motion is equivalent to moving forward. For the optimal solution, instead of storing 'L', 'B' and 'L', you should store an 'F' in the stack. Thus, if the last element is a 'B' and the element before that is 'L' and current motion is a left turn, your function should remove the 'B' and 'L' from the stack and push an 'F'. A list of possible equivalences are:

    - LBL -> F
    - LBF -> R
    - LBR -> B
    - FBL -> R
    - FBF -> B
    - RBL -> B

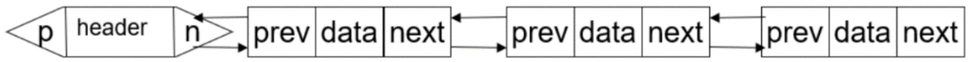
- In the second pass, use the StackQueue in the first pass as a queue. Reinitialize the maze and for each element in the queue, if the element is 'L' move left, if it is 'F' move forward and if it is 'R' move right. The agent should reach target location through the shortest path.

At each step, print StackQueue contents and maze state, see expected_output.txt

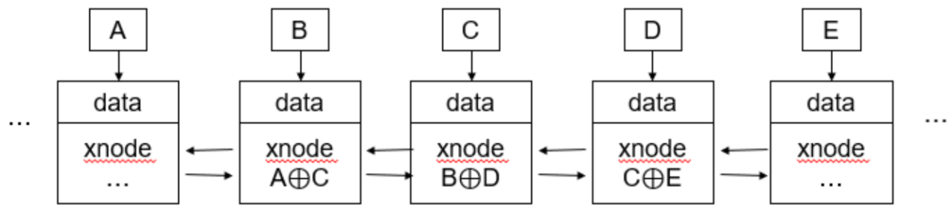
Note: Your implementation will be tested against different mazes. In all test cases, you can assume that there exists a solution, the maze does not have loops and a properly implemented Left Hand Rule algorithm can find the solution with the 1024-element StackQueue.

Part 2 – Linked List Implementation [40 points]

Consider a doubly linked list depicted below (note that it is linear, not circular):



As can be seen, each node contains one data field and two address fields. In this question, we will investigate the memory-efficient version of Doubly Linked Lists, which will be created using only one address field for each node. This is called XOR Linked List, as the list uses a bitwise XOR operation to save space for one address. A sample representation is provided below:



Node class declaration is provided below:

```
template <class T>
class Node
{
    public :
        T data;
        // Xor of next node and previous node
        Node<T>* xnode;
};
```

The method to XOR the addresses is given as follows:

```
template <class T>
Node<T>* Xor(Node<T>* x, Node<T>* y)
{
    return reinterpret_cast<Node<T>*>(
        reinterpret_cast<uintptr_t>(x)
        ^ reinterpret_cast<uintptr_t>(y));
}
```

A method for inserting a node at the beginning of the XORed LinkedList and marking the newly inserted node as the head is provided below:

```
template <class T>
void insert(Node<T>* &head_ref, T data)
{
    // Allocate memory for the new node
    Node<T>* new_node = new Node<T>();
    new_node -> data = data;

    // Since the new node is inserted at the
    // start, xnode of new node will always be
    // Xor of current head and NULL
    new_node -> xnode = head_ref;

    // If the linkedlist is not empty, then xnode of
    // present head node will be XOR of the new node
    // and node next to current head */
    if (head_ref != NULL)
    {
        // *(head_ref)->xnode is Xor of (NULL and next).
        // If we XOR Null with next, we get next
        head_ref->xnode = Xor(new_node, head_ref->xnode);
    }

    // Change head
    head_ref = new_node;
}
```

The following method prints the contents of XORed LinkedList from beginning to end:

```
template <class T>
void printList(Node<T>* head){
    Node<T>* currPtr = head;
    Node<T>* prevPtr = NULL;
    Node<T>* nextPtr;

    cout << "The nodes of Linked List are: \n";

    // Till condition holds true
    while (currPtr != NULL) {
        // print current node
        cout << currPtr -> data;

        // get the address of next node: currPtr->xnode is
        // {nextPtr (XOR) prevPtr}, so {currPtr->xnode (XOR) prevPtr} will
        // be {nextPtr (XOR) prevPtr (XOR) prevPtr}, which is nextPtr
        nextPtr = Xor(prevPtr, currPtr -> xnode);

        // update prevPtr and currPtr for next iteration
        prevPtr = currPtr;
        currPtr = nextPtr;
    }
    cout<<endl;
}
```

1) Implement the Doubly Linked List explained above. Note that you need to implement extra methods which are not given above, for example to delete a node. Your implementation needs to have

- `push_front` and `push_rear` methods that insert a new element at the beginning/end of the doubly linked list;
- `pop_front` and `pop_rear` methods that remove and return the element at the beginning/end of the doubly linked list,
- `peek_front` method that returns the element at beginning without modifying the linked list
- `print_elements` method that prints all the stored elements from end to beginning.

2) Replace the StackQueue of the solver function you wrote in Part 1.3 with this doubly linked list and show that it can still solve the maze.

Hint: Unlike the functions of StackQueue class, functions given in Part 2 take head pointer as the first argument. If you write a wrapper class for this implementation which has the same functions with the same arguments as StackQueue, you should be able to use the solver function you wrote in Part 1.3 without modification.

```
template <class T>
class LL_wrapper{
    private :
        Node<T>* head_node;
        …
    public :
        void push_front(T data){
            insert(head_node, data);
        }
        …
};
```

Appendix: You can use the following code to read the input maze from file.

```
#include <fstream>

/* reading the maze from file */
ifstream input_file; /* input file stream */
input_file.open ("input_maze.txt");

int nrow, ncol; /* number of rows and columns */
input_file >> nrow >> ncol;  /* read the size from file */

char state[9][8];   /* CAUTION! actual maze size might be different */

for(int i=0; i<9; ++i){
    for(int j=0; j<8; ++j){
        input_file >> state[i][j];
    }
}
input_file.close();
```