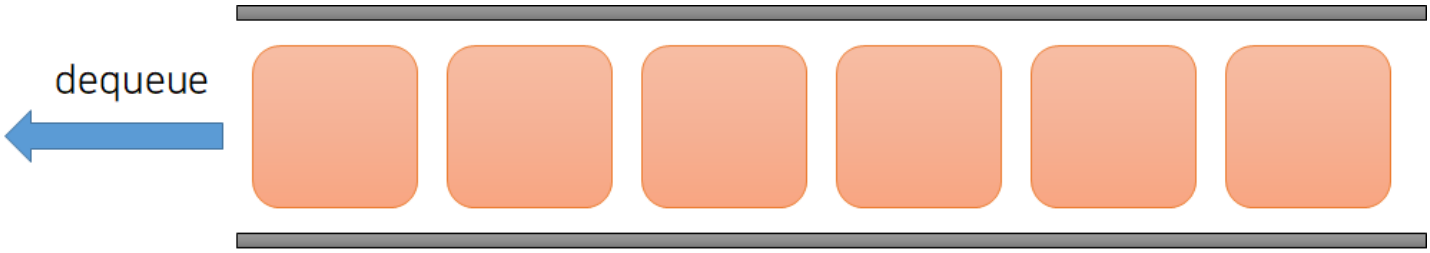
Lab04 - Queue Data Structure

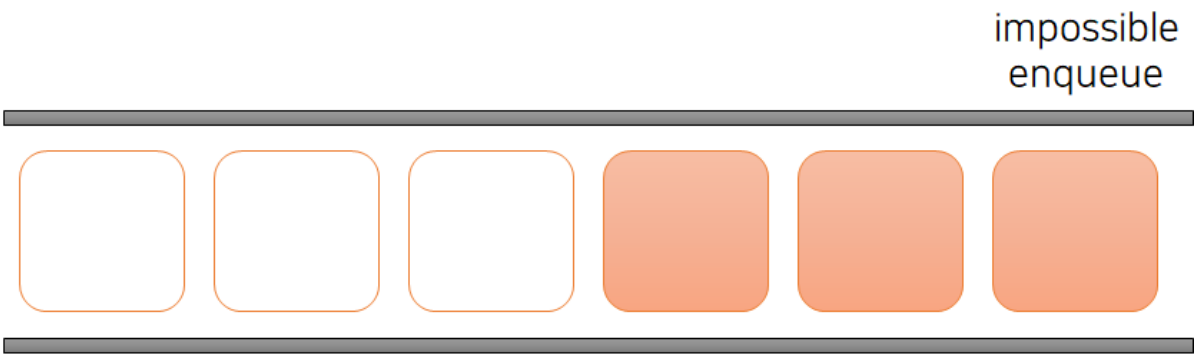
`2019136037 KimJuwon`

1. Implement Ticket Counter simulation

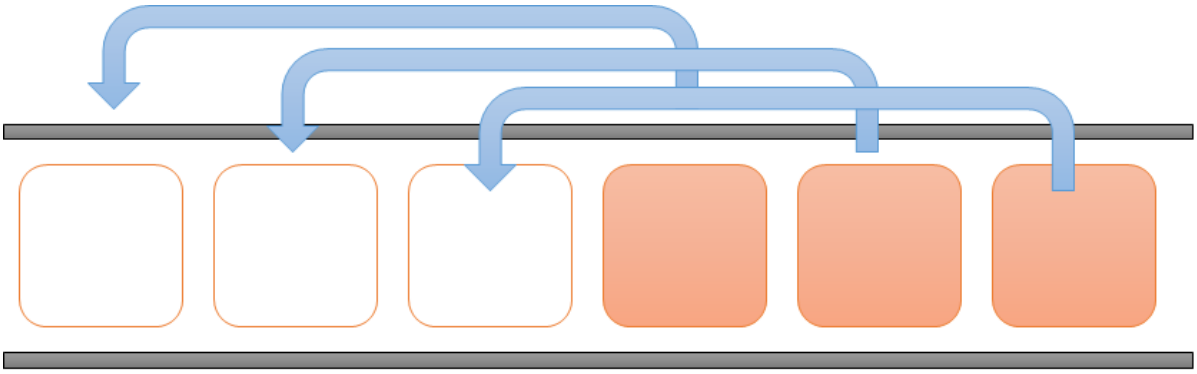
CircularQueue

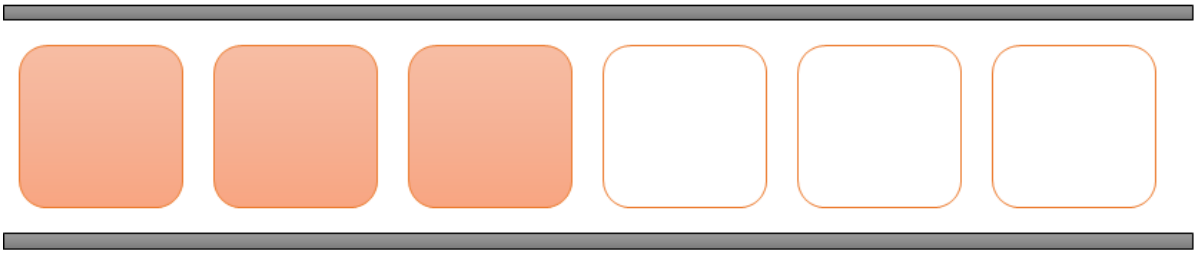
- Circular queue is data structure that complement queue.
- The disadvantage of queue is that it cannot be enqueue when it is full of capacity. In other words, overflow occurs.





- So we need to move the data one by one, but this increases the time complexity.





Average of time complexity : O(n)

- In order to solve these difficulties, we need to use **circular queue**.
- Circular queue is data structure that connect the front and back parts of the queue. if the queue is full, it is efficient because there is no need to perform a data move operation.
- In queue, the front and rear pointers increase **(pointer + 1) % queueCapacity. Because it is for moving from the last index to the first index.

init(self)

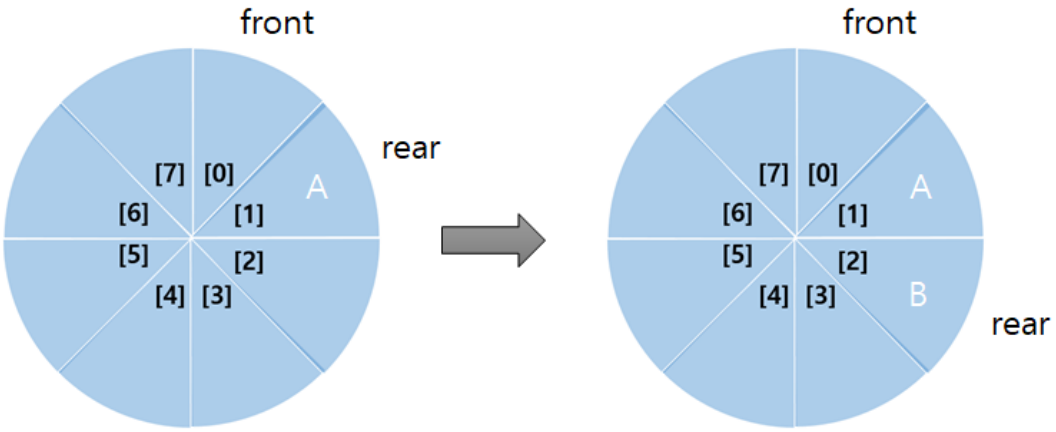
- Initialize the member variables and arrangement.

len(self)

- Return ((rear - front + queueCapacity) % queueCapacity).

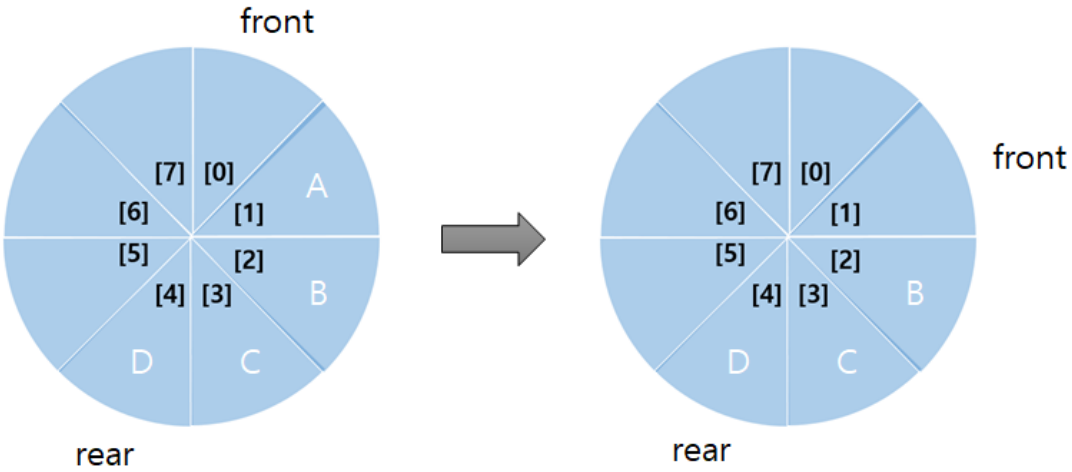
enqueue(self, item)

- If queue is not full, move rear to next space. And, insert item in the space indicated by rear.



dequeue(self)

- If queue is not empty, move front to next space. And, return item in the space indicated by front.

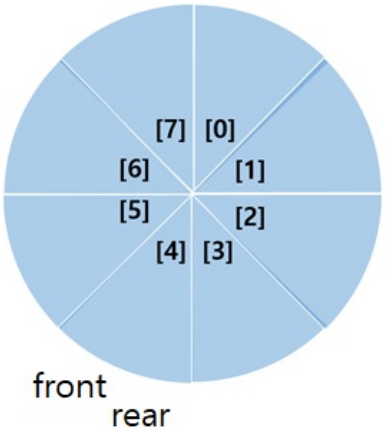


peek(self)

- If queue is not empty, return item in the space indicated by the next space of front.

isEmpty(self)

- If front is same for rear, the queue is empty. so, return (front == rear)



isFull(self)

- if next space of rear is same for front, the queue is full. so, return (front == (rear + 1) % queueCapacity).

clear(self)

- clear is that making empty in queue. So, make front and rear the same.

print(self)

- Outputs data (front + 1) or higher (rear + 1) or less.

```
class CircularDeque(CircularQueue):
    def __init__(self):
        super().__init__()

    def addFront(self, item):
        if not self.isFull():
            self.items[self.front] = item
            self.front = (self.front - 1 + MAX_QSIZE) % MAX_QSIZE

    def addRear(self, item):
        self.enqueue(item)

    def deleteFront(self):
        return self.dequeue()

    def deleteRear(self):
        if not self.isEmpty():
            item = self.items[self.rear]
            self.rear = (self.rear - 1 + MAX_QSIZE) % MAX_QSIZE
            return item

    def getFront(self, item):
        return self.peek()

    def getRear(self):
        return self.items[self.rear]
```

Passenger

init(self, pID, arrivalTime)

- Initialize the member variables.

getPID(self)

- Return pID (passenger ID).

timeArrived(self)

- Return arrivalTime.

```
class passenger:
    def __init__(self, pID, ArrivalTime):
        self._pID = pID
        self._arrivalTime = ArrivalTime

    def getPID(self):
        return self._pID

    def timeArrived(self):
        return self._arrivalTime
```

TicketAgent

init(self, aID)

- Initialize the member variables.

getAID(self)

- Return AID

isFree(self)

- Return the presence or absence of passenger.

isFinished(self, curTime)

- Returns (passenger is exist) && (equal curtime and stoptime)

startService(self, passenger, stopTime)

- Initialize passenger and stopTime.

stopService(self)

- Remove the passenger and return the original passenger.

```
class TicketAgent:
    def __init__(self, aID):
        self._aID = aID
        self._passenger = None
        self._stopTime = -1

    def getAID(self):
        return self._aID

    def isFree(self):
        return self._passenger is None

    def isFinished(self, CurTime):
        return self._passenger is not None and CurTime == self._stopTime

    def startService(self, passenger, stopTime):
        self._passenger = passenger
        self._stopTime = stopTime

    def stopService(self):
        thepassenger = self._passenger
        self._passenger = None
        return thepassenger
```

TicketCounterSimulation

init(self, numAgents, numMinutes, betweenTime, serviceTime)

- Initialize the member variables.

run(self)

- Perform the service from curTime to numMinutes.

```
class TicketCounterSimulation:
    def __init__(self, numAgents, numMinutes, betweenTime, serviceTime):
        self._arriveprob = 1.0 / betweenTime
        self._serviceTime = serviceTime
        self._numMinutes = numMinutes
        self.served = 0

        self._passengers = CircularQueue()
        self._Agents = [None] * numAgents
        for i in range(numAgents):
            self._Agents = TicketAgent(i + 1)

        self._totalWaitTime = 0
        self._numPassengers = 0

    def run(self):
        for curTime in range(self._numMinutes):
            self._handleArrival(curTime)
            self._handleBeginService(curTime)
            self._handleEndService(curTime)
        self.printResult()
```

```
def printResult(self):
    numServed = self._numPassengers - len(self._passengers)
    avgwait = float(self._totalWaitTime) / numServed
    print("")
    print("Number of passengers served()")
    print("Number of passengers remaining in line")
    print("The average wait time was")

# Handle Customer Arrival
def _handleArrival(self, curTime):
    prob = randint(0.0, 1.0)
    if 0.0 <= prob and prob <= self._arriveprob:
        person = Passenger(self._numPassengers + 1, curTime)
        self._passengers.enqueue(person)
        self._numPassengers += 1
        print("Time Passenger")

# Begin Customer Service
def _handleBeginService(self, curTime):
    i = 0
    while i < len(self._Agents):
        if self._Agents[i].isFree() and not self._passengers.isEmpty() and curTime != self._numMinutes:
            passenger = self._passengers.dequeue()
            self.served +=1
            stoptime = curTime + self._serviceTime
            self._Agents[i].starService(passenger, stoptime)
            self._totalWaitTime += (curTime - passenger.timeArrivaed())
            print("Time Agent started serving")

# End Customer Service
def _handleEndService(self, curTime):
    i = 0
    while i <len(self._Agents):
        if self._Agents[i].isFinished(curTime):
            passenger = self._Agents[i].stopService()
            print(":Time Agent stoped serving")

        i += 1
```

2. Implement Deque Data Structure

CircularDeque (extends CircularQueue)

init(self)

- Initialize the member variables by calling parent object(CircularQueue)'s constructor.

addFront(self, item)

- If queue is not full, insert item in the space indicated by front. And, move front to previous space((**front - 1 + queueCapacity) % queueCapacity**).

addRear(self, item)

- call parent object's enqueue() method.

deleteFront(self)

- call parent object's dequeue() method.

deleteRear(self)

- If queue is not empty, move rear to previous space and return data. To return data, original data of the front space is kept in temporary variables and returned.

getFront(self, item)

- call parent object's peek() method.

getRear(self, item)

- Return data in space indicated by rear.

```
class CircularDeque(CircularQueue):
    def __init__(self):
        super().__init__()

    def addFront(self, item):
        if not self.isFull():
            self.items[self.front] = item
            self.front = (self.front - 1 + MAX_QSIZE) % MAX_QSIZE

    def addRear(self, item):
        self.enqueue(self)

    def deleteFront(self):
        return self.dequeue()

    def deleteRear(self):
        if not self.isEmpty():
            item = self.items[self.rear]
            self.rear = (self.rear - 1 + MAX_QSIZE) % MAX_QSIZE
            return item

    def getFront(self, item):
        return self.peek()

    def getRear(self):
        return self.items[self.rear]
```

3. Solve the Maze problem through DFS and BFS using different data structures

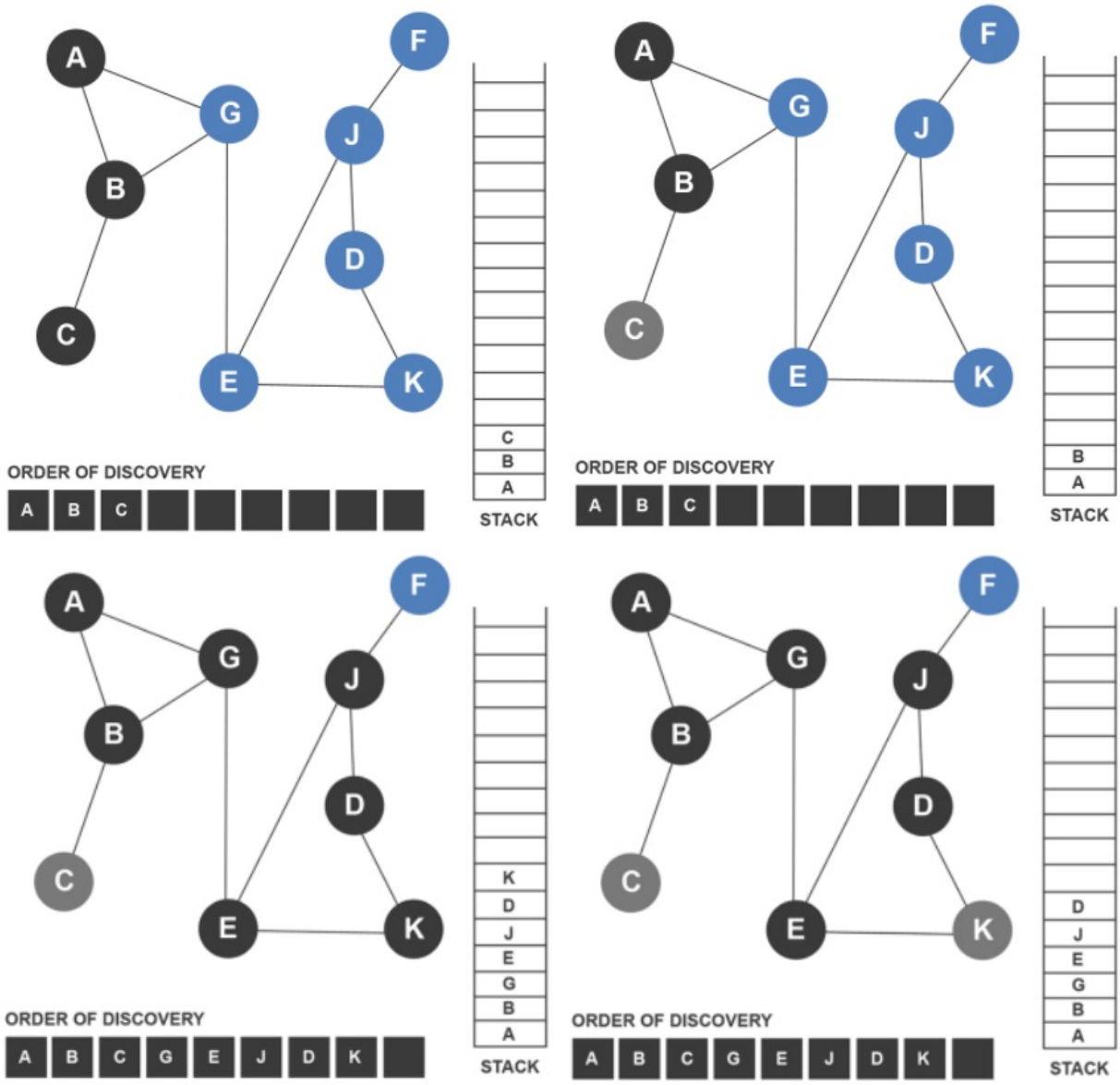
Maze

What is DFS(Depth First Search)?

In exploring, DFS is an algorithm that prioritizes deep data.

DFS uses a stack and performs the following algorithm.

1. Push the start data into the stack, and visit start data.
2. Check top of stack data.
3. If the top data has an adjacent data that has not been visited, the data is put into the stack and visited. If there is no adjacent data not visited, subtract the top data from the stack.
4. Repeat 2 and 3 until the stack is empty.

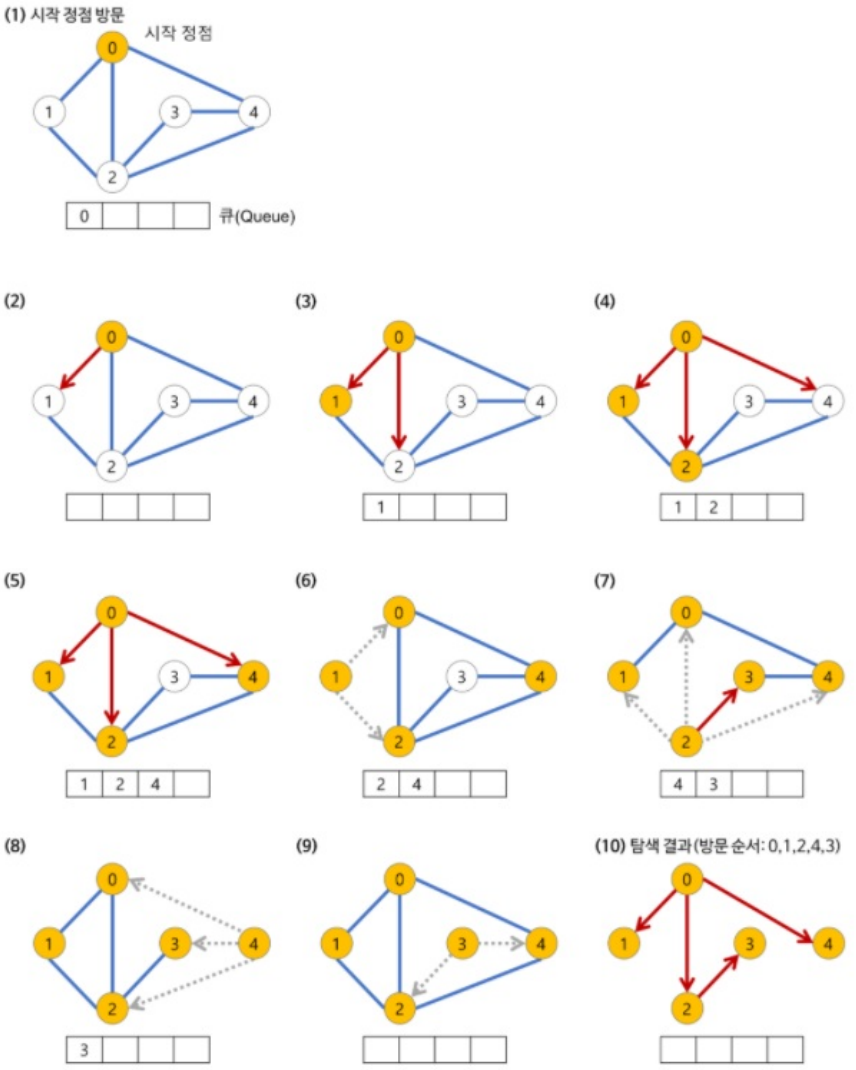


What is BFS(Breadth First Search)?

In exploring, DFS is an algorithm that prioritizes breadth data.

DFS uses a queue and performs the following algorithm.

1. Push the start data into the queue, and visit start data.
2. Dequeue one data from the queue.
3. Visits unvisited data among the data linked to that data, and enqueue them into the queue one by one.
4. Repeat 2 and 3 until the queue is empty.



getMap(self)

- Initialize two-dimensional map and return it.

isValidPos(self, x, y, map)

- Check if it is in a valid position. Both row and column should be 0 to (mazeSize - 1), and '0' or 'x'.

DFS1(self)

- Use circular queue like stacks.
    - Replace stack's push() with circular deque's addFront().
    - Replace stack's pop() with circular deque's deleteFront().

  The rest of the explanation is in the comments.

DFS2(self)

- Use Stack.

  The rest of the explanation is in the comments.

DFS3(self)

- Use circular queue like stacks.
    - Replace stack's push() with circular deque's addRear().
    - Replace stack's pop() with circular deque's deleteRear().

  The rest of the explanation is in the comments.

BFS1(self)

- Use circular deque.
    - addRear()
    - deleteFront()

  The rest of the explanation is in the comments.

BFS2(self)

- Use deque.
    - enqueue()
    - dequeue()

  The rest of the explanation is in the comments.

```
class Maze:
    MAZE_SIZE = 6

    def getMap(self):
        map = [['1', '1', '1', '1', '1', '1'],
               ['e', '0', '1', '0', '0', '1'],
               ['1', '0', '0', '0', '1', '1'],
               ['1', '0', '1', '0', '1', '1'],
               ['1', '0', '1', '0', '0', 'x'],
               ['1', '1', '1', '1', '1', '1']]
        return map

    def isValidPos(self, x, y, map):
        if (x < 0 or y < 0 or x >= self.MAZE_SIZE or y >= self.MAZE_SIZE):
            return False
        else:
            return map[x][y] == '0' or map[x][y] == 'x'
```

```
def DFS1(self):
    # Get map
    map = self.getMap()
    # Push(addFront) start data into circular deque
    deq = CircularDeque()
    entry = Cell(0, 1)
    deq.addFront(entry)
    print('\n DFS1: using Deque Data Structure: ')

    while not deq.isEmpty():
        here = deq.deleteFront()
        print(here, end = '->')
        x = here.row
        y = here.col
        if map[x][y] == 'x': # target point
            return True
        else:
            map[x][y] = '.' # Mark visited
            # Inspect to see if it can be moved to an adjacent location
            if self.isValidPos(x - 1, y, map):
                deq.addFront(Cell(x - 1, y))
            if self.isValidPos(x + 1, y, map):
                deq.addFront(Cell(x + 1, y))
            if self.isValidPos(x, y - 1, map):
                deq.addFront(Cell(x, y - 1))
            if self.isValidPos(x, y + 1, map):
                deq.addFront(Cell(x, y + 1))

    return False
```

```
def DFS2(self):
    # Get map
    map = self.getMap()
    # Push start data into stack
    stack = Stack()
    entry = Cell(0, 1)
    stack.push(entry)
    print('\n DFS2: using Stack Data Structure: ')

    while not stack.isEmpty():
        here = stack.pop()
        print(here, end = '->')
        x = here.row
        y = here.col
        if map[x][y] == 'x': # target point
            return True
        else:
            map[x][y] = '.' # Mark visited
            # Inspect to see if it can be moved to an adjacent location
            if self.isValidPos(x - 1, y, map):
                stack.push(Cell(x - 1, y))
            if self.isValidPos(x + 1, y, map):
                stack.push(Cell(x + 1, y))
            if self.isValidPos(x, y - 1, map):
                stack.push(Cell(x, y - 1))
            if self.isValidPos(x, y + 1, map):
                stack.push(Cell(x, y + 1))
```

```
def DFS3(self):
    # Get map
    map = self.getMap()
    # Push(addRear) start data into circular deque
    deq = CircularDeque()
    entry = Cell(0, 1)
    deq.addRear(entry)
    print('\n DFS3: using Deque Data Structure: ')

    while not deq.isEmpty():
        here = deq.deleteRear()
        print(here, end = '->')
        x = here.row
        y = here.col
        if map[x][y] == 'x': # target point
            return True
        else:
            map[x][y] = '.' # Mark visited
            # Inspect to see if it can be moved to an adjacent location
            if self.isValidPos(x - 1, y, map):
                deq.addRear(Cell(x - 1, y))
            if self.isValidPos(x + 1, y, map):
                deq.addRear(Cell(x + 1, y))
            if self.isValidPos(x, y - 1, map):
                deq.addRear(Cell(x, y - 1))
            if self.isValidPos(x, y + 1, map):
                deq.addRear(Cell(x, y + 1))

    return False
```

```
def BFS1(self):
    # Get map
    map = self.getMap()
    # Push(addRear) start data into circular deque
    deq = CircularDeque()
    entry = Cell(0, 1)
    deq.addRear(entry)
    print('\n BFS1: using Deque Data Structure: ')

    while not deq.isEmpty():
        here = deq.deleteFront()
        print(here, end = '->')
        x = here.row
        y = here.col
        if map[x][y] == 'x': # target point
            return True
        else:
            map[x][y] = '.' # Mark visited
            # Inspect to see if it can be moved to an adjacent location
            if self.isValidPos(x - 1, y, map):
                deq.addRear(Cell(x - 1, y))
            if self.isValidPos(x + 1, y, map):
                deq.addRear(Cell(x + 1, y))
            if self.isValidPos(x, y - 1, map):
                deq.addRear(Cell(x, y - 1))
            if self.isValidPos(x, y + 1, map):
                deq.addRear(Cell(x, y + 1))

    return False
```

```
def BFS2(self):
    # Get map
    map = self.getMap()
    # Push(enqueue) start data into queue.
    que = CircularQueue()
    entry = Cell(0, 1)
    que.enqueue(entry)
    print('\n BFS2: using Queue Data Structure: ')

    while not que.isEmpty():
        here = que.dequeue()
        print(here, end = '->')
        x = here.row
        y = here.col
        if map[x][y] == 'x': # target point
            return True
        else:
            map[x][y] = '.' # Mark visited
            # Inspect to see if it can be moved to an adjacent location
            if self.isValidPos(x - 1, y, map):
                que.enqueue(Cell(x - 1, y))
            if self.isValidPos(x + 1, y, map):
                que.enqueue(Cell(x + 1, y))
            if self.isValidPos(x, y - 1, map):
                que.enqueue(Cell(x, y - 1))
            if self.isValidPos(x, y + 1, map):
                que.enqueue(Cell(x, y + 1))

    return False
```