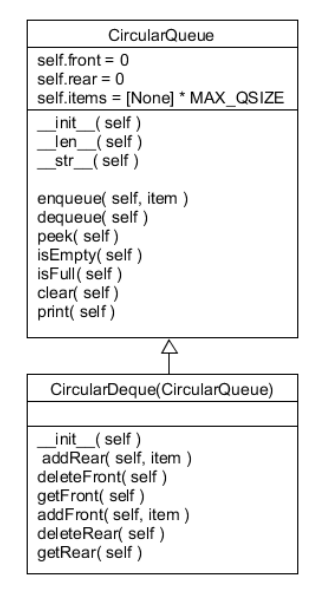
**Data Structures 2023-2**

**Lab 04: Queue Abstract Data Type**

**ID: 2021136150 Name: 양 동 재 (Yang Dong Jae)**

1. **Task-1: Implement Deque Data Structure**

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. Write code for Deque abstract data type and test it.

**Deque Operations:** The following basic operations are performed on the deque:

* addFront(): Adds an item at the front of Deque.
* addRear(): Adds an item at the rear of Deque.
* deleteFront(): Deletes an item from front of Deque.
* deleteRear(): Deletes an item from rear of Deque.
* getFront(): Gets the front item from queue.
* getRear(): Gets the last item from queue.
* isEmpty(): Checks whether Deque is empty or not.
* isFull(): Checks whether Deque is full or not.

|  |
| --- |
| **Code** |
| MAX\_QSIZE = 50  class CircularQueue:  def \_\_init\_\_(self):  self.front = 0  self.rear = 0  self.items = [None] \* MAX\_QSIZE  def \_\_len\_\_(self):  return (self.rear - self.front + MAX\_QSIZE) % MAX\_QSIZE  def \_\_str\_\_(self):  return str(self.items)  def enqueue(self, item):  if self.isFull():  raise Exception("Queue is full")  self.items[self.rear] = item  self.rear = (self.rear + 1) % MAX\_QSIZE  def dequeue(self):  if self.isEmpty():  raise Exception("Queue is empty")  item = self.items[self.front]  self.front = (self.front + 1) % MAX\_QSIZE  return item  def peek(self):  if self.isEmpty():  raise Exception("Queue is empty")  return self.items[self.front]  def isEmpty(self):  return self.front == self.rear  def isFull(self):  return (self.rear + 1) % MAX\_QSIZE == self.front  def clear(self):  self.front = self.rear = 0  self.items = [None] \* MAX\_QSIZE  class CircularDeque(CircularQueue):  def \_\_init\_\_(self):  super().\_\_init\_\_()  def addRear(self, item):  self.enqueue(item)  def deleteFront(self):  return self.dequeue()  def getFront(self):  return self.peek()  def addFront(self, item):  if self.isFull():  raise Exception("Deque is full")  self.front = (self.front - 1 + MAX\_QSIZE) % MAX\_QSIZE  self.items[self.front] = item  def deleteRear(self):  if self.isEmpty():  raise Exception("Deque is empty")  self.rear = (self.rear - 1 + MAX\_QSIZE) % MAX\_QSIZE  return self.items[self.rear]  def getRear(self):  if self.isEmpty():  raise Exception("Deque is empty")  return self.items[(self.rear - 1 + MAX\_QSIZE) % MAX\_QSIZE] |

**Results/Output**

|  |
| --- |
| **Test code** |
| dq = CircularDeque()  dq.addFront(1)  dq.addRear(2)  dq.addFront(3)  dq.addRear(4)  print("Deque:", dq)  print("Front:", dq.getFront())  print("Rear:", dq.getRear())  dq.deleteFront()  dq.deleteRear()  print("Deque after deletions:", dq) |
| **Result** |
| **Deque: [2, 4, None, None, None, None, None, None, 3, 1]**  **Front: 3**  **Rear: 4**  **Deque after deletions: [2, 4, None, None, None, None, None, None, None, 1]** |

1. **Task2:** Ticketing Counter system (Simulation)

A computer simulation can be developed to model this Ticketing Counter system using the Queue data structure.

****

An object-oriented solution with multiple classes.

* CircularQueue : Data structure to hold the passengers.
* Passenger : store info related to a passenger.
* TicketAgent : store info related to an agent.
* TicketCounterSimulation : manages the actual simulation.

**Code**

|  |
| --- |
| **Code** |
| class Passenger:  def \_\_init\_\_(self, id, name):  self.id = id  self.name = name  def \_\_str\_\_(self):  return f"Passenger ID: {self.id}, Name: {self.name}"  class TicketAgent:  def \_\_init\_\_(self, id, name):  self.id = id  self.name = name  self.serving\_passenger = None  def assign\_passenger(self, passenger):  self.serving\_passenger = passenger  def serve\_passenger(self):  if self.serving\_passenger:  print(f"{self.name} is serving {self.serving\_passenger}")  # time.sleep(random.uniform(1, 3))  print(f"{self.name} has served {self.serving\_passenger}")  self.serving\_passenger = None  class TicketCounterSimulation:  def \_\_init\_\_(self, num\_agents, num\_passengers):  self.agents = [TicketAgent(i, f"Agent {i}") for i in range(1, num\_agents + 1)]  self.passenger\_queue = CircularQueue()  self.passengers = [Passenger(i, f"Passenger {i}") for i in range(1, num\_passengers + 1)]  def simulate(self):  for passenger in self.passengers:  self.passenger\_queue.enqueue(passenger)  while not self.passenger\_queue.isEmpty():  for agent in self.agents:  if agent.serving\_passenger is None and not self.passenger\_queue.isEmpty():  passenger = self.passenger\_queue.dequeue()  agent.assign\_passenger(passenger)  agent.serve\_passenger() |

**Results/Output**

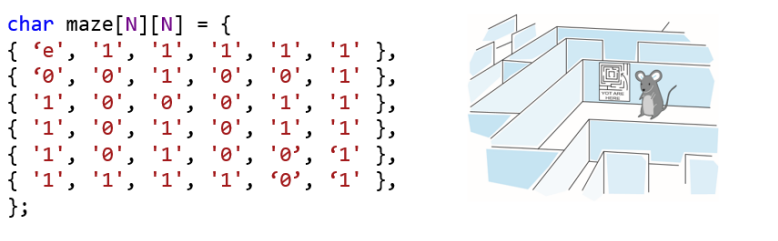
|  |
| --- |
| **Test code** |
| num\_agents = 3  num\_passengers = 15  simulation = TicketCounterSimulation(num\_agents, num\_passengers)  simulation.simulate() |
| **Result** |
| **Agent 1 is serving Passenger ID: 1, Name: Passenger 1**  **Agent 1 has served Passenger ID: 1, Name: Passenger 1**  **Agent 2 is serving Passenger ID: 2, Name: Passenger 2**  **Agent 2 has served Passenger ID: 2, Name: Passenger 2**  **Agent 3 is serving Passenger ID: 3, Name: Passenger 3**  **Agent 3 has served Passenger ID: 3, Name: Passenger 3**  **Agent 1 is serving Passenger ID: 4, Name: Passenger 4**  **Agent 1 has served Passenger ID: 4, Name: Passenger 4**  **Agent 2 is serving Passenger ID: 5, Name: Passenger 5**  **Agent 2 has served Passenger ID: 5, Name: Passenger 5**  **Agent 3 is serving Passenger ID: 6, Name: Passenger 6**  **Agent 3 has served Passenger ID: 6, Name: Passenger 6**  **Agent 1 is serving Passenger ID: 7, Name: Passenger 7**  **Agent 1 has served Passenger ID: 7, Name: Passenger 7**  **Agent 2 is serving Passenger ID: 8, Name: Passenger 8**  **Agent 2 has served Passenger ID: 8, Name: Passenger 8**  **Agent 3 is serving Passenger ID: 9, Name: Passenger 9**  **Agent 3 has served Passenger ID: 9, Name: Passenger 9**  **Agent 1 is serving Passenger ID: 10, Name: Passenger 10**  **Agent 1 has served Passenger ID: 10, Name: Passenger 10**  **Agent 2 is serving Passenger ID: 11, Name: Passenger 11**  **Agent 2 has served Passenger ID: 11, Name: Passenger 11**  **Agent 3 is serving Passenger ID: 12, Name: Passenger 12**  **Agent 3 has served Passenger ID: 12, Name: Passenger 12**  **Agent 1 is serving Passenger ID: 13, Name: Passenger 13**  **Agent 1 has served Passenger ID: 13, Name: Passenger 13**  **Agent 2 is serving Passenger ID: 14, Name: Passenger 14**  **Agent 2 has served Passenger ID: 14, Name: Passenger 14**  **Agent 3 is serving Passenger ID: 15, Name: Passenger 15**  **Agent 3 has served Passenger ID: 15, Name: Passenger 15** |

1. **Task-3:** Solve the Maze problem through DFS and BFS using different data structures

A Maze is given as N×N binary matrix of blocks where source block is maze[0][0] and destination

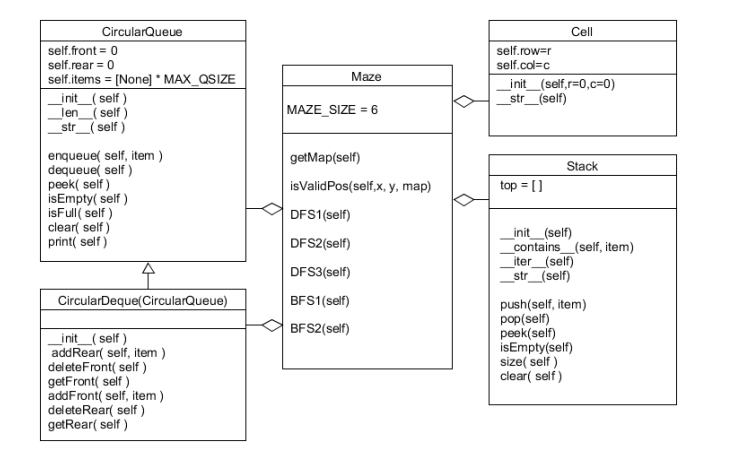
block is maze[N-1][N-1].

* A rat starts from the source and has to reach the destination. The rat can move only in four directions: upward, down, forward, and back word.
* In the maze matrix, 1 means the block is a dead end and 0 means the block can be used in the path from source to destination
* The task is to check if there exists any path so that the rat can reach at the destination or not.

****

**Write the following functions in Maze class**

* DFS1() : It searches maze using stack from CircularDeque (front end operations)
* DFS2() : It searches maze using Stack
* DFS1() : It searches maze using stack from CircularDeque (rear end operation)
* BFS1() : It searches maze using Queue from CircularDeque
* BFS2() : It searches maze using Queue from CircularQueue

****

**Code**

|  |
| --- |
| **Code** |
| class Maze:  Maze\_Size = 6  def \_\_init\_\_(self):  self.map = [['e', '1', '1', '1', '1', '1'],  ['0', '0', '1', '0', '0', '1'],  ['1', '0', '0', '0', '1', '1'],  ['1', '0', '1', '0', '1', '1'],  ['1', '0', '1', '0', '0', '1'],  ['1', '1', '1', '1', 'z', '1']]  def getMap(self):  return self.map  def printMaze(self):  for row in self.map:  print(" ".join(row))  print("\n")  def isValidPos(self, x, y, map):  return 0 <= x < Maze.Maze\_Size and 0 <= y < Maze.Maze\_Size and (map[x][y] != '1' and map[x][y] != 'X' or map[x][y] == 'e')  def DFS1(self):  stack = CircularDeque()  start\_x, start\_y = 0, 0 # Starting position  stack.addFront(Cell(start\_x, start\_y))  count = 0  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up  while not stack.isEmpty():  count += 1  # time.sleep(1)  # self.printMaze()  cell = stack.getFront()  x, y = cell.row, cell.col  found = False # Flag to track if a valid move is found  for dx, dy in directions:  new\_x, new\_y = x + dx, y + dy  if self.map[new\_x][new\_y] == 'z':  print("DFS1 based algorithm need to {}times to solve this problem".format(count))  return True  if self.isValidPos(new\_x, new\_y, self.map):  stack.addFront(Cell(new\_x, new\_y))  self.map[new\_x][new\_y] = 'X' # Mark as visited  found = True # Valid move found  if not found:  stack.deleteFront() # Backtrack if no valid moves are found  return False  def DFS2(self):  stack = Stack()  start\_x, start\_y = 0, 0 # Starting position  stack.push(Cell(start\_x, start\_y))  count = 0  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up  while not stack.isEmpty():  count += 1  cell = stack.pop()  x, y = cell.row, cell.col  # time.sleep(1) # Add a delay for visualization (adjust as needed)  # self.printMaze()  for dx, dy in directions:  new\_x, new\_y = x + dx, y + dy  if self.isValidPos(new\_x, new\_y, self.map):  stack.push(Cell(new\_x, new\_y))  if self.map[new\_x][new\_y] == 'z':  print("DFS2 based algorithm need to {}times to solve this problem".format(count))  return True  self.map[new\_x][new\_y] = 'X' # Mark as visited  return False    def DFS3(self):  stack = CircularDeque()  start\_x, start\_y = 0, 0 # Starting position  stack.addRear(Cell(start\_x, start\_y))  count = 0  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up  while not stack.isEmpty():  count += 1  # self.printMaze() # Print the maze at each step for visualization  # time.sleep(1) # Add a delay for visualization (adjust as needed)  cell = stack.deleteRear()  x, y = cell.row, cell.col  for dx, dy in directions:  new\_x, new\_y = x + dx, y + dy  if self.isValidPos(new\_x, new\_y, self.map):  if self.map[new\_x][new\_y] == 'z':  print("DFS3 based algorithm need to {}times to solve this problem".format(count))  return True  stack.addRear(Cell(new\_x, new\_y))  self.map[new\_x][new\_y] = 'X' # Mark as visited  return False  def BFS1(self):  queue = CircularDeque() # Using CircularDequeue as a queue  start\_x, start\_y = 0, 0 # Starting position  queue.enqueue(Cell(start\_x, start\_y))  count = 0  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up  while not queue.isEmpty():  count += 1  # time.sleep(1)  # self.printMaze()    cell = queue.dequeue()  x, y = cell.row, cell.col  for dx, dy in directions:  new\_x, new\_y = x + dx, y + dy  if self.map[new\_x][new\_y] == 'z':  print("BFS1 based algorithm need to {}times to solve this problem".format(count))  return True  if self.isValidPos(new\_x, new\_y, self.map):  queue.enqueue(Cell(new\_x, new\_y))  self.map[new\_x][new\_y] = 'X' # Mark as visited  return False  def BFS2(self):  queue = CircularQueue() # Using CircularQueue as a queue  start\_x, start\_y = 0, 0 # Starting position  queue.enqueue(Cell(start\_x, start\_y))  count = 0  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Right, Down, Left, Up  while not queue.isEmpty():  count += 1  # time.sleep(1)  # self.printMaze()  cell = queue.dequeue()  x, y = cell.row, cell.col  for dx, dy in directions:  new\_x, new\_y = x + dx, y + dy  if self.map[new\_x][new\_y] == 'z':  print("BFS2 based algorithm need to {}times to solve this problem".format(count))  return True  if self.isValidPos(new\_x, new\_y, self.map):  queue.enqueue(Cell(new\_x, new\_y))  self.map[new\_x][new\_y] = 'X' # Mark as visited  return False  class Cell:  def \_\_init\_\_(self, r=0, c=0):  self.row = r  self.col = c  def \_\_str\_\_(self):  return f"({self.row}, {self.col})"  class Stack:  def \_\_init\_\_(self):  self.top = []  def \_\_contains\_\_(self, item):  return item in self.top  def \_\_iter\_\_(self):  return iter(self.top)  def \_\_str\_\_(self):  return str(self.top)  def push(self, item):  self.top.append(item)  def pop(self):  if not self.isEmpty():  return self.top.pop()  def peek(self):  if not self.isEmpty():  return self.top[-1]  def isEmpty(self):  return len(self.top) == 0  def size(self):  return len(self.top)  def clear(self):  self.top = [] |

**Results/Output**

|  |
| --- |
| **Test code** |
| maze1 = Maze()  maze2 = Maze()  maze3 = Maze()  maze4 = Maze()  maze5 = Maze()  print("DFS1 Result:", maze1.DFS1())  print("DFS2 Result:", maze2.DFS2())  print("DFS3 Result:", maze3.DFS3())  print("BFS1 Result:", maze4.BFS1())  print("BFS2 Result:", maze5.BFS2()) |
| **Result** |
| **DFS1 based algorithm need to 16times to solve this problem**  **DFS1 Result: True**  **DFS2 based algorithm need to 14times to solve this problem**  **DFS2 Result: True**  **DFS3 based algorithm need to 14times to solve this problem**  **DFS3 Result: True**  **BFS1 based algorithm need to 14times to solve this problem**  **BFS1 Result: True**  **BFS2 based algorithm need to 14times to solve this problem**  **BFS2 Result: True** |

1. **Conclusion**

문제 1: Deque 자료 구조 구현

해결 방법:

Deque 자료 구조를 구현하기 위해 원형 배열을 사용하는 방식을 선택했습니다. 배열 크기는 최대 크기(`MAX\_QSIZE`)로 고정되어 있으며 원형 동작을 보장하기 위해 front와 rear 포인터를 사용했습니다. 다양한 연산을 수행하면서 원형 동작을 유지하도록 메서드를 구현했습니다.

느낀점:

Deque를 구현하는 과정에서 배열의 인덱스 관리와 연산들을 조절하는 부분이 까다로웠습니다. 원형 배열을 다루면서 인덱스 계산에 조심해야 했으며, Deque의 다양한 연산들을 효율적으로 처리하려는 노력이 필요했습니다.

문제 2: 티켓 카운터 시스템 시뮬레이션

해결 방법:

객체 지향적인 접근 방식을 사용하여 티켓 카운터 시스템 시뮬레이션을 개발했습니다. 여러 클래스를 사용하여 다음과 같이 구성했습니다.

- `CircularQueue`: 승객을 대기시키는 데이터 구조.

- `Passenger`: 승객 정보를 저장하는 클래스.

- `TicketAgent`: 카운터 직원 정보를 저장하는 클래스.

- `TicketCounterSimulation`: 실제 시뮬레이션을 관리하는 클래스.

시뮬레이션은 승객을 이용 가능한 직원에 할당하고, 서비스 과정을 시뮬레이트하며 대기열의 상태를 추적합니다.

느낀점:

문제 2번에서 티켓 카운터 시스템 시뮬레이션을 개발하면서 Queue 자료 구조의 중요성과 효율성을 직관적으로 이해할 수 있었습니다.

1.대기열 관리: CircularQueue를 사용하여 승객을 대기열에 추가하고 제거하는 데 큐가 어떻게 효과적인지 이해했습니다. 새로운 승객은 뒤에 추가되고, 서비스를 받은 승객은 앞에서 제거되므로, 먼저 온 승객이 먼저 서비스를 받을 수 있게 관리할 수 있었습니다.

2.서비스 할당: TicketCounterSimulation 클래스에서 사용 가능한 티켓 카운터 직원을 찾기 위해 큐를 사용했습니다. 이렇게 함으로써 승객과 직원 간의 효율적인 할당을 구현할 수 있었습니다. 먼저 대기열에 있는 승객에게 먼저 서비스를 할당하여 공정한 서비스를 제공할 수 있었습니다.

3. 동시성 관리: Queue를 사용하면 다수의 승객과 다수의 카운터 직원이 동시에 서비스를 받고 제공할 수 있는 시뮬레이션 환경을 구축할 수 있었습니다. 이것은 실제 세계의 상황과 유사한 복잡한 시스템을 모델링하는 데 중요합니다.

또한, Queue를 사용하여 대기열을 관리하면 알고리즘을 단순하게 유지할 수 있어서 코드의 가독성과 유지 보수성이 향상되었습니다. 이 경험을 통해 Queue 자료 구조의 활용성과 효율성을 직관적으로 이해할 수 있었으며, 실제 응용 프로그램에서 대기열을 관리하는 데 그 가치를 인식하게 되었습니다.

문제 3: DFS와 BFS를 사용하여 미로 문제 해결

해결 방법:

DFS와 BFS 알고리즘을 사용하여 미로 문제를 해결하기 위해 각각 다른 데이터 구조를 사용했습니다. 각 알고리즘에 대한 해결 방법은 다음과 같습니다:

* `DFS1()`: 스택(CircularDeque)과 front-end 연산을 사용하여 깊이 우선 탐색을 수행했습니다.
* `DFS2()`: 일반 스택을 사용하여 깊이 우선 탐색을 수행했습니다.
* `DFS3()`: 스택(CircularDeque)과 rear-end 연산을 사용하여 깊이 우선 탐색을 수행했습니다.
* `BFS1()`: 큐(CircularDeque)를 사용하여 너비 우선 탐색을 수행했습니다.
* `BFS2()`: 큐(CircularQueue)를 사용하여 너비 우선 탐색을 수행했습니다.

각 알고리즘은 미로를 지정된 데이터 구조를 사용하여 탐색하며 목적지까지의 경로를 추적합니다. 경로가 발견되면 True를 반환하고 그렇지 않으면 False를 반환합니다.

느낀점:

미로 문제를 해결하면서 다양한 데이터 구조를 사용하는 DFS와 BFS의 차이점과 장단점을 이해했습니다. 미로 문제를 해결하는 것은 재귀 및 스택과 큐를 사용한 경로 탐색에 대한 이해를 향상시키는 데 도움이 되었습니다. 또한, 각 알고리즘의 성능 및 효율성을 고려하는 것이 중요함을 깨달았습니다.

이번 과제를 통해 다양한 자료 구조와 알고리즘을 사용하여 다양한 문제를 해결하는 기회를 가졌습니다. 아래는 이 과제를 통해 얻은 최종 결론과 느낀점입니다.

최종 결론:

1. 자료 구조의 중요성: 이 과제를 통해 자료 구조의 선택이 문제 해결에 어떠한 영향을 미칠 수 있는지를 명확히 이해했습니다. 각 문제에 적합한 자료 구조를 선택하는 것은 알고리즘의 효율성과 문제 해결의 성공에 결정적인 역할을 합니다.

2. 알고리즘의 차이: 문제 3에서 DFS와 BFS 알고리즘을 사용하여 미로 문제를 해결하는 방법을 비교하면서, 각 알고리즘의 특징과 장단점을 이해했습니다. 각 알고리즘은 다른 상황에서 효과적이며, 문제에 따라 적절한 알고리즘을 선택해야 합니다.