Data Structures and Lab

Queue Data Structure

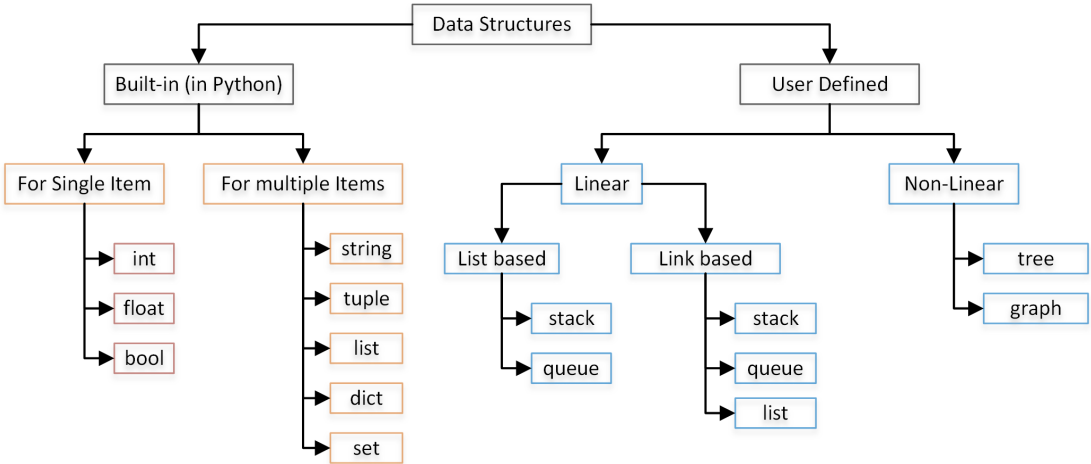
Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- Data Structures and Algorithms Using Python by Rance D. Necaise
- Data Structures Using Python by Y.K. Choi and I.G. Chan (Korean)
- https://www.geeksforgeeks.org/data-structures/
- https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/Python1a_OOP.html

Data Structures

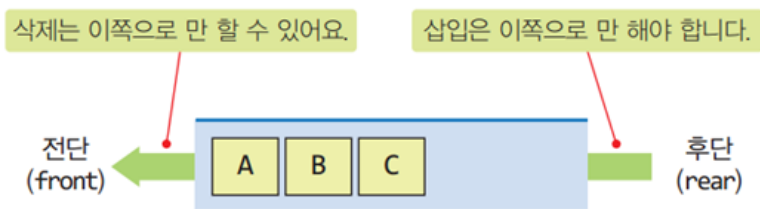


Contents

Queue Data Structure

Queue Data Structure

- Queue is also a special case of list data structure
- In contrast to stack, a queue is a FIFO (First-In First-Out) structure.
- A queue is a linear structure for which items can be only inserted at one end and can be removed at the other end.
- One end, is called the front (or head) of the queue, and other end is called the back (or rear or tail).



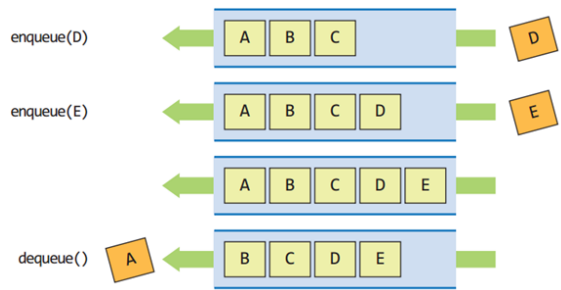
Queue Data Structure (cont...)

- Examples from daily life
  - A line of people waiting for a bank teller
  - A line of persons waiting to check out at a supermarket
  - A line of persons waiting to purchase a ticket for a film
  - A line of planes waiting to take off at an airport
  - A line of cars at a toll both

Queue Data Structure (cont...)

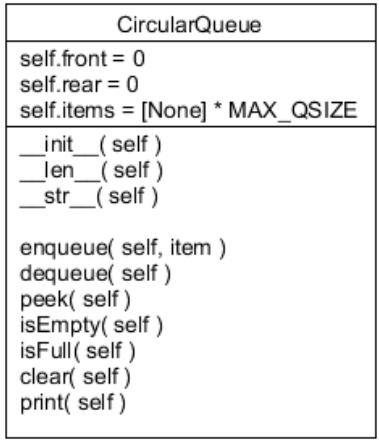
FIFO principle

- Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
- Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
- Insert operation is called enqueue operation.
- Delete operation is called dequeue operation.



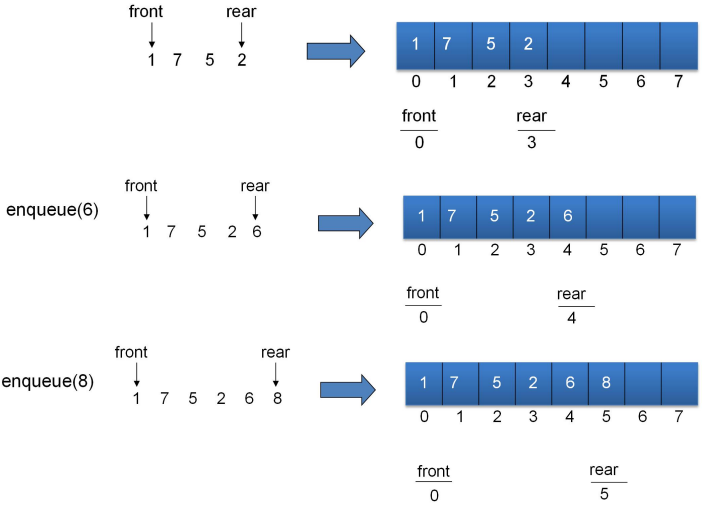
Queue Data Structure (cont...)

- Queue abstract data type ADT
  - Object: Collection of data items such that First-In, First-Out (FIFO) mechanism is maintained
  - Operations
    - enqueue(e) : It adds an element e at the rear end of the queue
    - dequeue() : It removes the element from the front end of the queue.
    - isEmpty(): It returns true if the queue is empty, otherwise false
    - peek(): It returns the element at the front end without removing it from the queue
    - isFull(): It returns true if the queue is full, false otherwise
    - size(): It returns the number of items in the queue
    - display(): It displays all elements stored in the queue



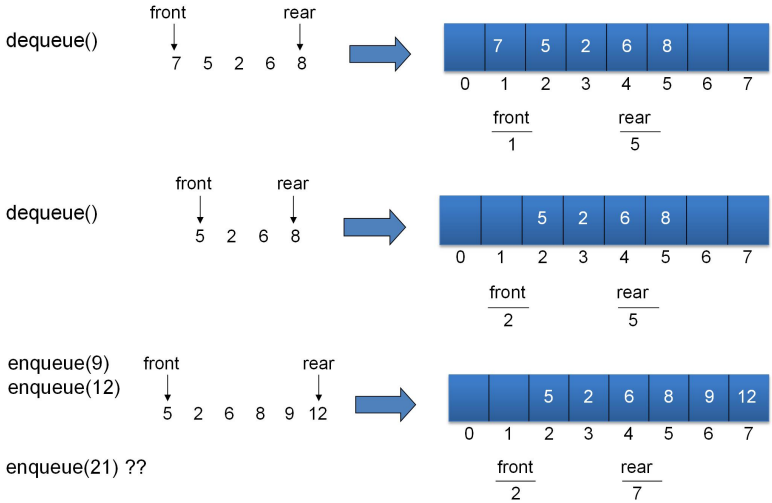
Implementation of Queue Data Structure

- Consider the queue below and think about few queue operations



Implementation of Queue Data Structure (cont...)

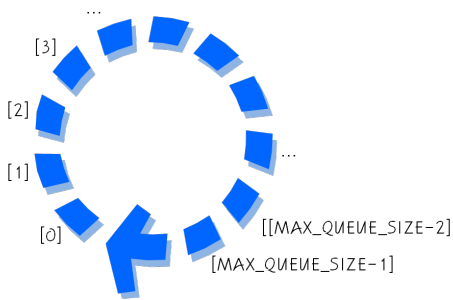
- Few more operations



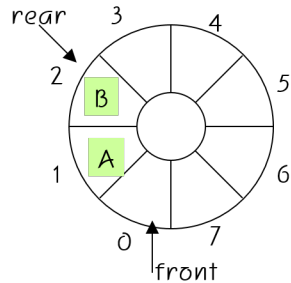
Implementation of Queue Data Structure (cont...)

Circular Queue

- To overcome the shifting elements problem, basic idea is to picture the list/array as a circular list.



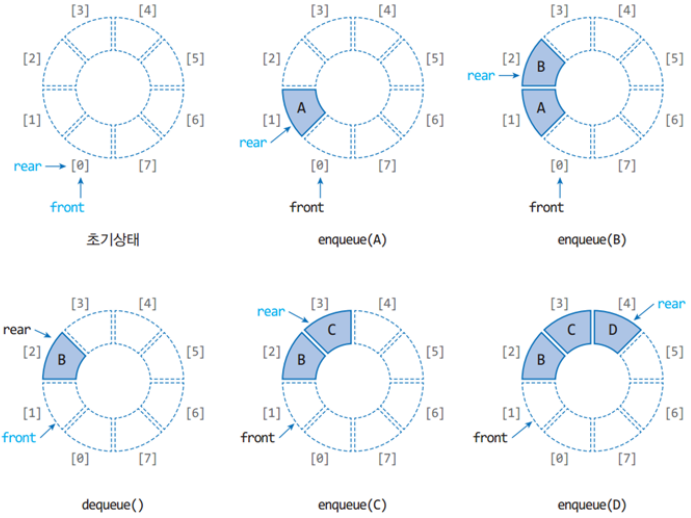
- It requires two variables to manage front and rear ends
  - front : index before the first element
  - rear : index for the last element



- Items can be added/removed without having to shift the remaining items in the process.
- Introduces the concept of a maximum capacity queue that can become full.

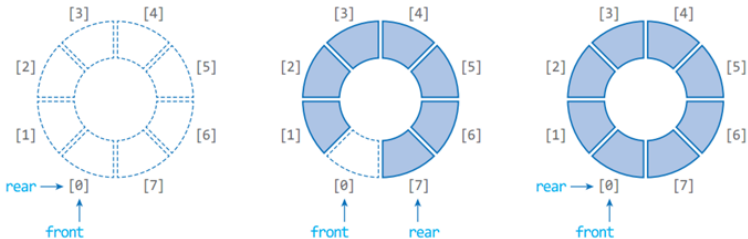
Implementation of Queue Data Structure (cont...)

- Behavior of rear and front variables in circular queue



Implementation of Queue Data Structure (cont...)

- Full and empty queue: Use the remainders (modulo operations) to rotate the index round.



- Wrong indexes : front % MAX_QUEUE_SIZE==(rear+1) % MAX_QUEUE_SIZE
- Full queue : front ==(rear+1) % MAX_QUEUE_SIZE
- Empty queue : front == rear
- rear = (rear+1) % MAX_QUEUE_SIZE
- front = (front+1)% MAX_QUEUE_SIZE

Implementation of Queue Data Structure (cont...)

- The Circular Queue is implemented using Python List data structure
- The size of the list must be predetermined
- It is important to determine empty and full queues

```
MAX_QSIZE = 10                          # 원형 큐의 크기
class CircularQueue :
    def __init__( self ) :              # CircularQueue 생성자
        self.front = 0                  # 큐의 전단 위치
        self.rear = 0                   # 큐의 후단 위치
        self.items = [None] * MAX_QSIZE # 항목 저장용 리스트 [None,None,...]
    def isEmpty( self ) : return self.front == self.rear
    def isFull( self ) : return self.front == (self.rear+1)%MAX_QSIZE
    def clear( self ) : self.front = self.rear
```

Implementation of Queue Data Structure (cont...)

- enqueue() and dequeue() functions

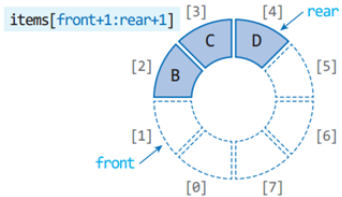
```
def enqueue( self, item ):
    if not self.isFull():                    # 포화상태가 아니면
        self.rear = (self.rear+1)% MAX_QSIZE  # rear 회전
        self.items[self.rear] = item          # rear 위치에 삽입

def dequeue( self ):
    if not self.isEmpty():                   # 공백상태가 아니면
        self.front = (self.front+1)% MAX_QSIZE # front 회전
        return self.items[self.front]         # front위치의 항목 반환

def peek( self ):
    if not self.isEmpty():
        return self.items[(self.front + 1) % MAX_QSIZE]
```

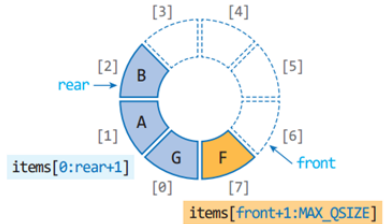
```
def size( self ) :
    return (self.rear - self.front + MAX_QSIZE) % MAX_QSIZE
```

Implementation of Queue Data Structure (cont...)

- Display the items in the queue (__str__(self) or display(self))

```
def display( self ):
    out = []
    if self.front < self.rear :
        out = self.items[self.front+1:self.rear+1]          # 슬라이싱
    else:
        out = self.items[self.front+1:MAX_QSIZE] ₩          # 다음 줄에 계속...
            + self.items[0:self.rear+1]                     # 슬라이싱
    print("[f=%s,r=%d] ==> "%(self.front, self.rear), out)
```





Applications of Queue Data Structure

- Queues have a wide range of applications
  - Queue can be used to solve scheduling and parallel programming problems.
  - Queue can be used in breadth-first search (BFS) on a tree or graph data structure.
  - Queue can be used in scheduling print jobs between printer and computer (buffering)
  - Queue can be used in buffering in real-time video streaming
  - Queue can be used simulations (airplanes at airports, queues at banks)
  - Queue can be used to model data packets in communications

- Computers can be used to model and simulate real-world systems and phenomena using Queue data structure.

Ticket Counter simulation

Ticket Counter simulation

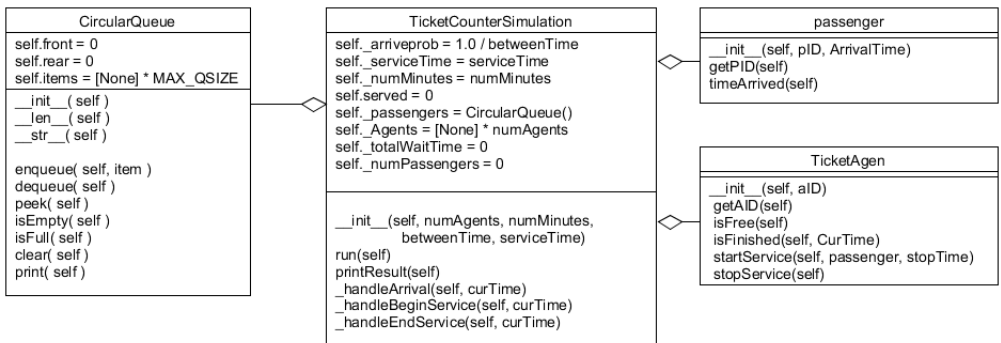
- A computer simulation can be developed to model this Ticketing Counter system.
- How many ticket agents are needed at certain times of the day in order to provide timely services to passengers?
- Too many agents will cost the airline money.
- Too few will result in angry passengers (i.e. passengers have to wait for long times).

Ticket Counter simulation (cont...)

- Queuing System : A system where customers must stand in line awaiting service.
- A queue structure is used to model the system.
- Simple systems only require a single queue.
- The goal is to study certain behaviors or outcomes.
  - average wait time
  - average queue length
  - average service time
- Consists of a sequence of significant events that cause a change in the system. Some sample events include:
  - Passenger arrival
  - Start or end of service
  - Customer departure
- To correctly model a queuing system, some events must occur at random. (i.e. passenger arrival)

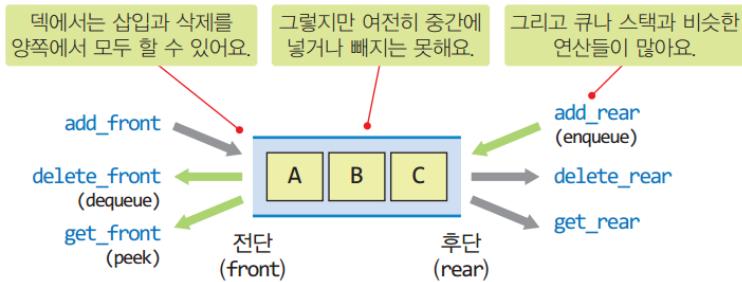
Ticket Counter simulation (cont...)

- An object-oriented solution with multiple classes.
  1. Passenger : store info related to a passenger.
  2. TicketAgent : store info related to an agent.
  3. TicketCounterSimulation : manages the actual simulation.



Deque Data Structure

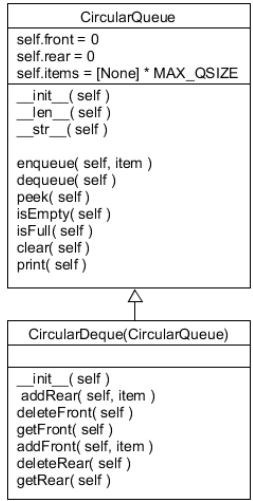
- Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.



- Since Deque supports both stack and queue operations, it can be used as both.
- The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

Deque Data Structure (cont...)

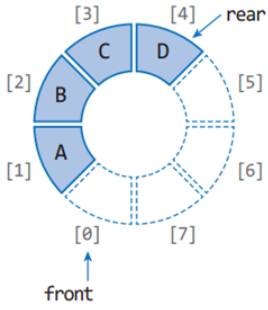
- Operations on Deque: Mainly the following four basic operations are performed on deque:
  - addFront(): Adds an item at the front of Deque.
  - addRear(): Adds an item at the rear of Deque.
  - deleteFront(): Deletes an item from front of Deque.
  - deleteRear(): Deletes an item from rear of Deque.
  - 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.

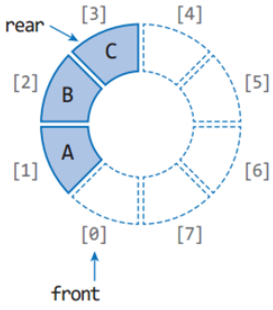


Deque Data Structure (cont...)

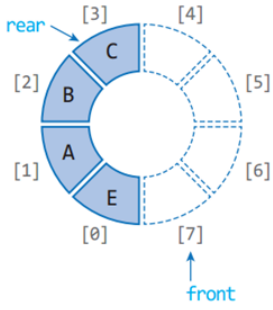
- queue operations - (addRear(),enqueue()), (deleteFront()), dequeue() (getFront(), peek())
- deleteRear(), addFront(), getRear()

```
front ← (front-1 + MAX_QSIZE) % MAX_QSIZE
rear ← (rear -1 + MAX_QSIZE) % MAX_QSIZE
```





deleteRear()



addFront(E)

Deque Data Structure (cont...)

- Code for : deleteRear(), addFront(), getRear()

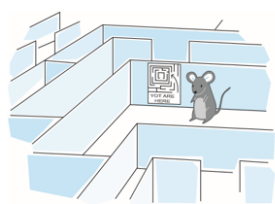
```
def addFront( self, item ):                    # 새로운 기능: 전단 삽입
    if not self.isFull():
        self.items[self.front] = item          # 항목 저장
        self.front = self.front - 1            # 반시계 방향으로 회전
        if self.front < 0 : self.front = MAX_QSIZE - 1

def deleteRear( self ):                        # 새로운 기능: 후단 삭제
    if not self.isEmpty():
        item = self.items[self.rear];          # 항목 복사
        self.rear = self.rear - 1              # 반시계 방향으로 회전
        if self.rear < 0 : self.rear = MAX_QSIZE - 1
        return item                            # 항목 반환

def getRear( self ):                           # 새로운 기능: 후단 peek
    return self.items[self.rear]
```

Maze Problem

Maze Problem

- A Maze is given as $N \times N$ binary matrix of blocks where source block is *maze*[0][0] and destination block is maze[N-1][N-1].
- A rat starts from source and has to reach the destination. The rat can move only in four directions: upward, down, forward, backword .
- 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.

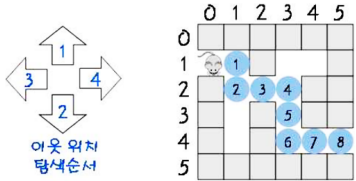
```
char maze[N][N] = {
{ 'e', '1', '1', '1', '1', '1' },
{ '0', '0', '1', '0', '0', '1' },
{ '1', '0', '0', '0', '1', '1' },
{ '1', '0', '1', '0', '1', '1' },
{ '1', '0', '1', '0', '0', '1' },
{ '1', '1', '1', '1', '0', '1' },
};
```

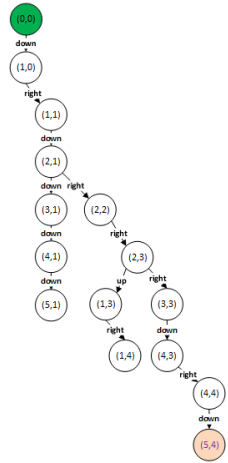


- The task is to check if there exists any path so that the rat can reach at the destination or not.

Maze Problem (cont...)

State Space Tree for Maze Problem

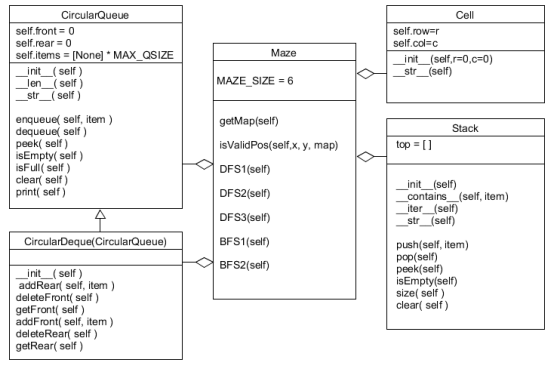




1. **States:** location of the rate in the maze at (row, col)
2. **Initial state:** The first cell maze[0][0]
3. **Actions:** up, down, left, right
4. **Transition model:** upon an action new location in maze[row][col]
5. **Goal test:** reaching at exit in maze[5][4]
6. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

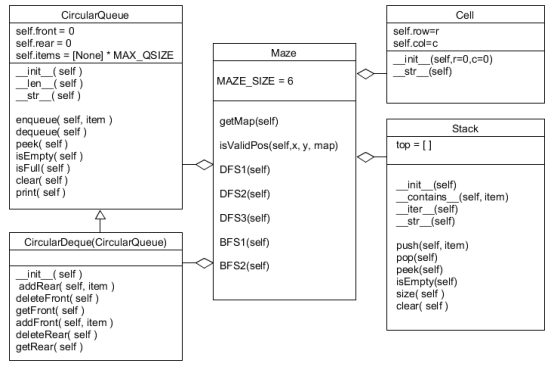
Maze Problem (cont...)

- Depth First Search (DFS) with an explicit Stack
  1. create a stack object
  2. Choose the initial cell and push it onto the stack
  3. While the stack is not empty
     - 3.1 Pop a cell from the stack and make it a current cell
     - 3.2 if the current cell is "exit", return True (problem solved)
     - 3.3 If the current cell has any neighbours/children. make cells and push them onto the stack
  4. return False, (Could not find exit)



Maze Problem (cont...)

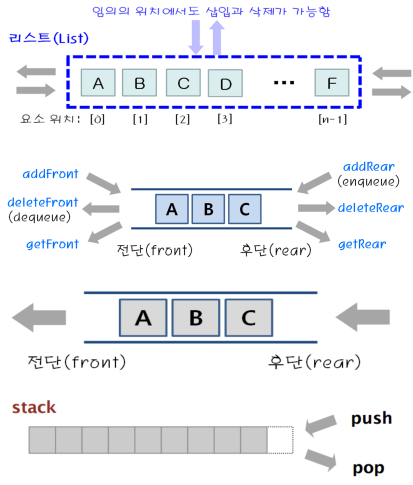
- Breadth First Search (BFS) with an explicit Queue
  1. create a queue object
  2. Choose the initial cell and enqueue it into the queue
  3. While the queue is not empty
     - 3.1 dequeue a cell from the queue and make it a current cell
     - 3.2 if the current cell is "exit", return True (problem solved)
     - 3.3 If the current cell has any neighbours/children. make cells and enqueue them into the queue
  4. return False (Could not find exit)



Linear Data Structures (Summary)

Linear Data Structures (Summary)

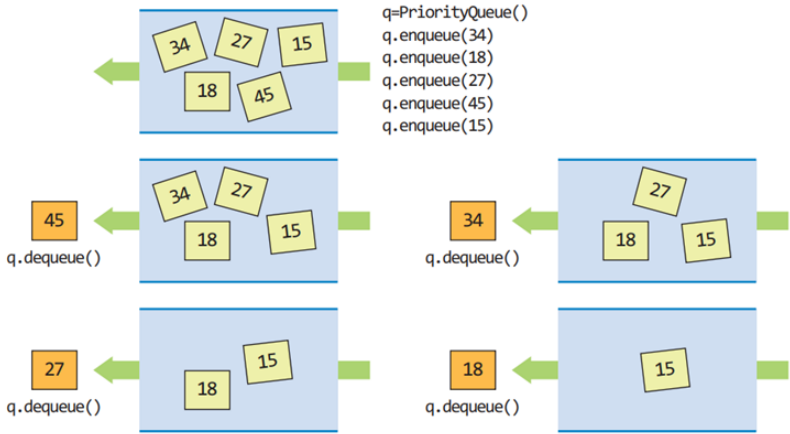
- List is a more general data structure. Items can be inserted and deleted at any position
- Deque, Queue and Stack are special cases of List data structure
- Deque is a specific data structure. Items can only be inserted and deleted at rear and front ends
- Queue is a more specific data structure. Items can only be inserted at rear end and can be deleted at front end
- Stack is also a more specific data structure. Items can be inserted and deleted one end



Priority Queue Data Structure

Priority Queue Data Structure

- In Priority Queue, every item has a priority associated with it. An element with high priority is dequeued before an element with low priority. (it is a non-linear data structure)



Priority Queue Data Structure (cont...)

- A priority queue is implemented using Heap.