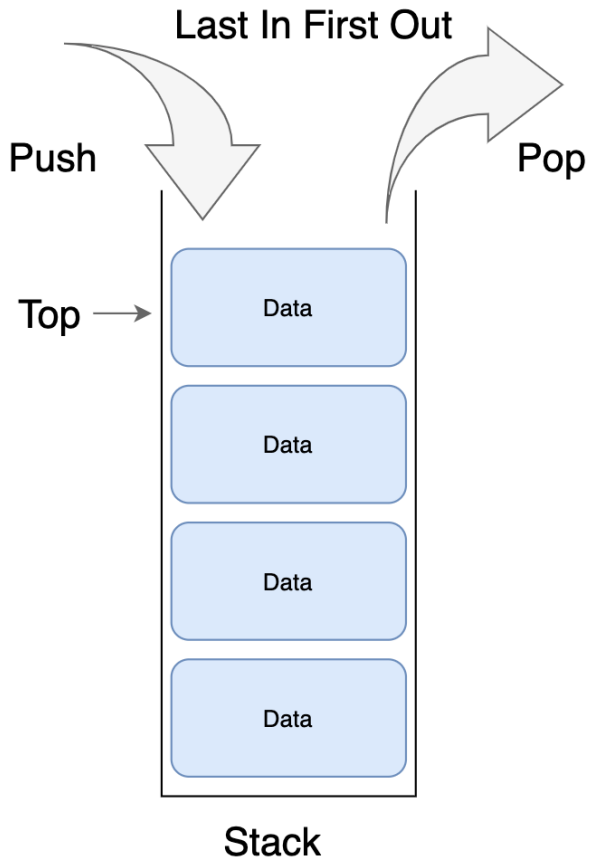
스택



●스택이란?

-주어진 일련의 데이터들을 층층이 쌓아 올리는 형태의 자료구조

-스택의 최상단에 위치하는 top에서 원소의 삽입 및 삭제가 이루어짐

-Last In First Out – 마지막으로 들어온 데이터가 먼저 나감

Ex)쌓여있는 책 더미

●스택의 operations

-push(): 스택의 top에 데이터를 새롭게 추가함

-pop(): 스택의 top에 위치하는 데이터를 제거한 후 값을 반환함

-peek(): 스택 top에 존재하는 데이터를 제거하지 않고 값을 반환함

-isFull(): 스택이 다 찼는지 확인

-isEmpty(): 스택이 비어 있는지 확인

●스택과 재귀함수

-재귀(recursion)함수: 함수의 정의부에서 자기 자신을 호출하는 함수

-프로그램 내에서 함수 혹은 메서드가 호출되는 방식은 스택 자료구조 차원에서 접근할 수 있음.

함수가 최초로 호출되었을 때 그 함수는 스택에 push되는 걸로 간주함. 함수의 실행이 종결되었을 때 그 함수는 스택에서 pop되는 걸로 간주함. 현재 돌아가고 있는 함수가 스택의 최상단에 위치함.

●링크드 리스트를 이용한 스택 구현(파이썬)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        if self.top is None:
            self.top = Node(data)
        else:
            node = Node(data)
            node.next = self.top
            self.top = node

    def pop(self):
        if self.top is None:
            return None
        node = self.top
        self.top = self.top.next
        return node.data

    def peek(self):
        if self.top is None:
            return None
        return self.top.data

    def is_empty(self):
        return self.top is None
```

●배열을 이용한 스택 구현(파이썬)

-스택 생성

stack = list() #비어있는 리스트를 생성함으로써 빈 스택에서 출발

-push()

stack.append(3) #append를 이용하여 리스트의 맨 끝에 원소를 삽입

-pop()

stack.pop #pop을 이용하여 가장 마지막에 삽입된 원소를 제거, 삭제된 원소 반환

-peek()

stack[-1]

●스택의 시간 복잡도

-push(), pop(), peek()의 시간 복잡도는 모두 O(1)

●스택의 활용

-괄호 판별

-중위 표기식 -> 후위 표기식

중위 표기식: A*B+C / 후위 표기식: AB*C+

1. 피연산자(숫자)의 경우 그대로 출력한다
2. 연산자의 경우, stack의 top의 우선순위 >= 연산자의 우선순위일 경우, 자신보다 우선순위가 낮은 연산자가 top에 올 때까지 pop하고 자신을 push한다. Stack top의 우선순위 < 연산자의 우선순위일 경우, 연산자를 스택에 push한다.
3. 여는 괄호를 만날 경우, 무조건 스택에 push한다. 닫는 괄호를 만날 경우, 여는 괄호가 나올 때까지 연산자를 스택에서 pop한다. 괄호는 후위 표기식에서 출력하지 않는다.
4. 중위 표기식을 모두 읽은 후 스택에 남아있는 연산자들을 출력한다.

Ex) A+B/C*D*(E+F) -> ABC/D*EF+*+

큐

-First In First Out – 먼저 들어온 데이터가 먼저 나감

-queue는 대기열이라는 뜻으로, 큐의 작동 방식은 대기열과 유사함

●큐의 operations

-enqueue(): 큐의 rear에 새로운 데이터를 삽입

-dequeue(): 큐의 front에 존재하는 데이터를 제거한 후 값을 반환함

-peek(): 큐의 front에 있는 데이터를 제거하지 않고 값을 반환함

-isFull(): 큐가 다 찼는지 확인

-isEmpty(): 큐가 비어 있는지 확인

●링크드 리스트를 이용한 큐 구현(파이썬)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def enqueue(self, data):
        if self.front is None:
            self.front = self.rear = Node(data)
        else:
            node = Node(data)
            self.rear.next = node
            self.rear = node
```

```
    def dequeue(self):
        if self.front is None:
            return None
        node = self.front
        if self.front == self.rear:
            self.front = self.rear = None
        else:
            self.front = self.front.next
        return node.data

    def is_empty(self):
        return self.front is None
```

●배열을 이용한 큐 구현(파이썬)

-큐 생성

queue = list() #비어있는 리스트를 생성함으로써 빈 큐에서 출발

-enqueue()

queue.append(7) #append를 이용하여 리스트의 맨 끝에 원소를 삽입

-dequeue()

Pop(0) #최전방 원소는 최좌측에 있으므로 가장 첫 원소를 제거

-peek()

queue[0] -> 순서상 맨 끝 원소가 아닌, 가장 첫 원소임에 주목

●큐의 시간 복잡도(배열로 구현할 경우)

-enqueue()의 시간 복잡도는 O(1)

-데이터를 삭제할 경우, 뒤쪽 데이터들을 모두 한 칸씩 밀어야만 원하는 삭제가 가능하므로, dequeue()의 시간 복잡도는 O(n)

-peek()의 시간 복잡도는 O(1)

●collections 모듈을 이용한 큐 구현(파이썬)

-from collections import deque

-주요 함수: append(), popleft()

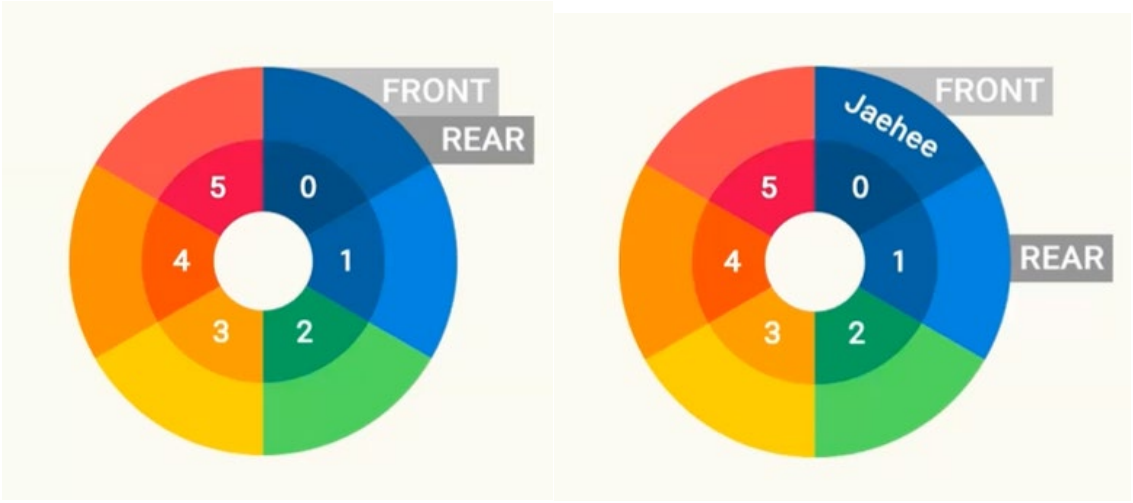
-내부적으로 링크드 리스트를 통해 구현되며, enqueue(), dequeue(), peek()의 시간 복잡도가 모두 O(1)

●환형 큐(Circular Queue)

-선형이 아닌 환형 대기열의 형태로 데이터를 제어하는 자료구조

-삭제에 비교적 높은 시간 복잡도를 요하는 일반 큐 대비 우월

-일반적으로 크기를 고정해둔 상태에서 환형 큐를 이용



-front: 주어진 환형 큐의 최전방 원소와 함께 움직임(dequeue할 때 움직임)

-rear: 주어진 환형 큐의 최후방 원소와 함께 움직임(enqueue할 때 움직임)

-환형 큐에서 enqueue를 시도할 경우 rear가 가리키고 있었던 위치에 데이터가 삽입되며 새 데이터 삽입으로 인해 rear은 한 칸 밀려나게 됨 -> O(1)

-환형 큐에서 deque를 시도할 경우 front가 가리키고 있던 위치의 데이터가 삭제되며(None) 데이터 삭제로 인해 front는 한 칸 밀려나게 됨 -> O(1)

●환형 큐 구현(파이썬)

```
class cQueue :

  def __init__(self, size) :
    self.size = size
    self.front = 0
    self.rear = 0
    self.data_list = list()
    for i in range(0, size) :
      self.data_list.append(None)     ◀ index로 동적 배열을 자유로이 제어하기 위함
```

```
(… 클래스 정의부에서 이어짐 …)

  def enqueue(self, data) :
    if (self.rear + 1 - self.front) % self.size == 0 :
      print('QUEUE FULL')                ◀ 꽉 찬 Queue에는 데이터를 삽입할 수 없음
    else :
      self.data_list[self.rear] = data
      self.rear = (self.rear + 1) % self.size
```

```
(… 클래스 정의부에서 이어짐 …)

  def dequeue(self) :
    if (self.front == self.rear) :
      print('QUEUE EMPTY')               ◀ 텅 빈 Queue에서 데이터를 제거할 수는 없음
    else :
      self.data_list[self.front] = None
      self.front = (self.front + 1) % self.size
```