




Stack/Queue

 fastcampus(자료구조 이론)

 programmers

1 스택(Stack)

사용 예

1. 스택 구조
2. 알아둘 용어
2. 스택 구조와 프로세스 스택
3. 자료 구조 스택의 장단점
4. 파이썬 리스트 기능에서 제공하는 메서드로 스택 사용
5. 구현
6. 시간 복잡도 Big O

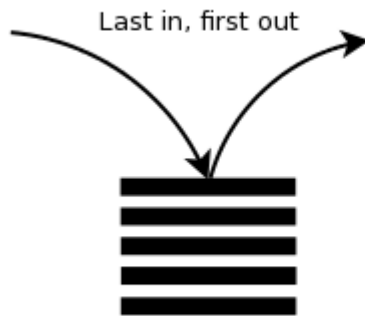
2 대표적인 데이터 구조4: 큐(Queue)

어디에 큐가 많이 쓰일까?

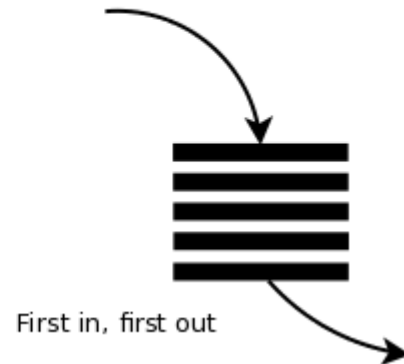
사용 예(시간 순서대로 처리)

1. 큐 구조
 2. 알아둘 용어
 3. 파이썬 queue 라이브러리 활용해서 큐 자료 구조 사용하기
 4. 구현
 5. 시간복잡도 Big O
- + Circular Queue
- + Deque

Stack:



Queue:



🤔 programmers

- L2. 기능개발
- 문제

1 스택(Stack)

- 데이터를 제한적으로 접근할 수 있는 구조
 - 한쪽 끝에서만 자료를 넣거나 뺄 수 있는 구조
- 가장 나중에 쌓은 데이터를 가장 먼저 빼낼 수 있는 데이터 구조
 - 큐: FIFO 정책
 - 스택: LIFO 정책

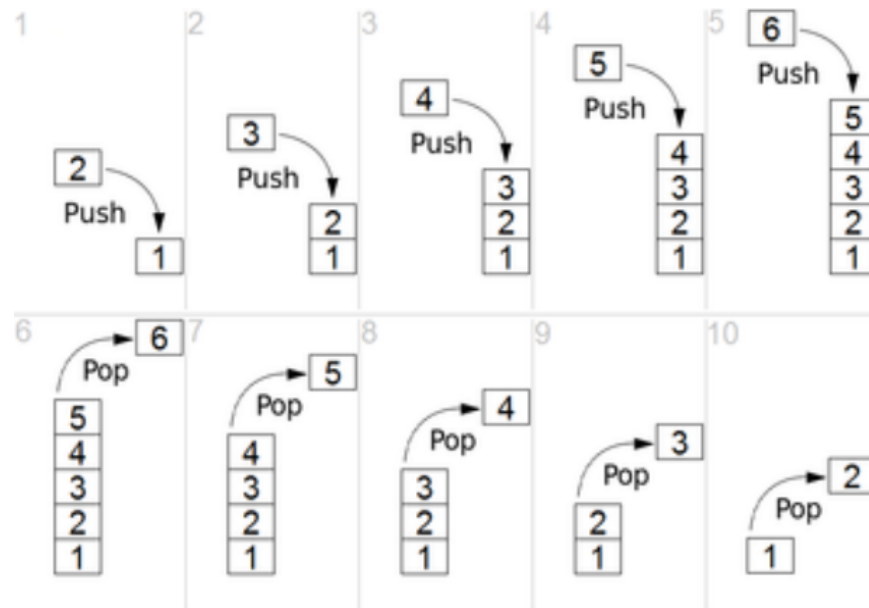
사용 예

- 웹 브라우저 방문기록 (뒤로 가기) : 가장 나중에 열린 페이지부터 다시 보여준다.
- 역순 문자열 만들기 : 가장 나중에 입력된 문자부터 출력한다.
- 실행 취소 (undo) : 가장 나중에 실행된 것부터 실행을 취소한다.
- 수식의 괄호 검사 (연산자 우선순위 표현을 위한 괄호 검사)

1. 스택 구조

- 스택은 LIFO(Last In, First Out) 또는 FILO(First In, Last Out) 데이터 관리 방식을 따름

- **LIFO**: 마지막에 넣은 데이터를 가장 먼저 추출하는 데이터 관리 정책
- **FILO**: 처음에 넣은 데이터를 가장 마지막에 추출하는 데이터 관리 정책
- 대표적인 스택의 활용
 - 컴퓨터 내부의 프로세스 구조의 함수 동작 방식
- Visualgo 사이트에서 시연



2. 알아둘 용어

- **push()**: 데이터를 스택에 넣기
- **pop()**: 데이터를 스택에서 꺼내기
- **peek()**: 마지막 위치(top)에 해당하는 데이터 읽음
이 때, top의 변화는 없음

2. 스택 구조와 프로세스 스택

```
# 재귀 함수
def recursive(data):
    if data < 0:
        print('ended')
    else:
        print(data)
        recursive(data-1)
        print('returned', data)

recursive(4)
```

```
> result
4
3
2
1
0
ended
returned 0
returned 1
returned 2
returned 3
returned 4
```

3. 자료 구조 스택의 장단점

- 장점
 - 구조가 단순해서, 구현이 쉽다.
 - 데이터 저장/읽기 속도가 빠르다.
- 단점 (일반적인 스택 구현시)
 - 데이터 최대 갯수를 미리 정해야 한다.
 - 파이썬의 경우 재귀 함수는 1000번까지만 호출이 가능함
 - 저장 공간의 낭비가 발생할 수 있음
 - 미리 최대 갯수만큼 저장 공간을 확보해야 함

스택은 단순하고 빠른 성능을 위해 사용되므로, 보통 배열 구조를 활용해서 구현하는 것이 일반적임.

이 경우, 위에서 열거한 단점이 있을 수 있음

4. 파이썬 리스트 기능에서 제공하는 메서드로 스택 사용

```
data_stack = list()
data_stack.append(1)
data_stack.append(2)

data_stack      # [1,2]
```

```
data_stack.pop # 2
```

5. 구현

```
stack_list = list()

def push(data):
    stack_list.append(data)

def pop():
    data = stack_list[-1]
    del stack_list[-1]
    return data
def peek(data):
    if data:
        return data[-1]

for index in range(10):
    push(index)

pop() # 9
```

6. 시간 복잡도 Big O

- Insertion $O(1)$
- Deletion $O(1)$
- Search $O(n)$

2 대표적인 데이터 구조4: 큐(Queue)

어디에 큐가 많이 쓰일까?

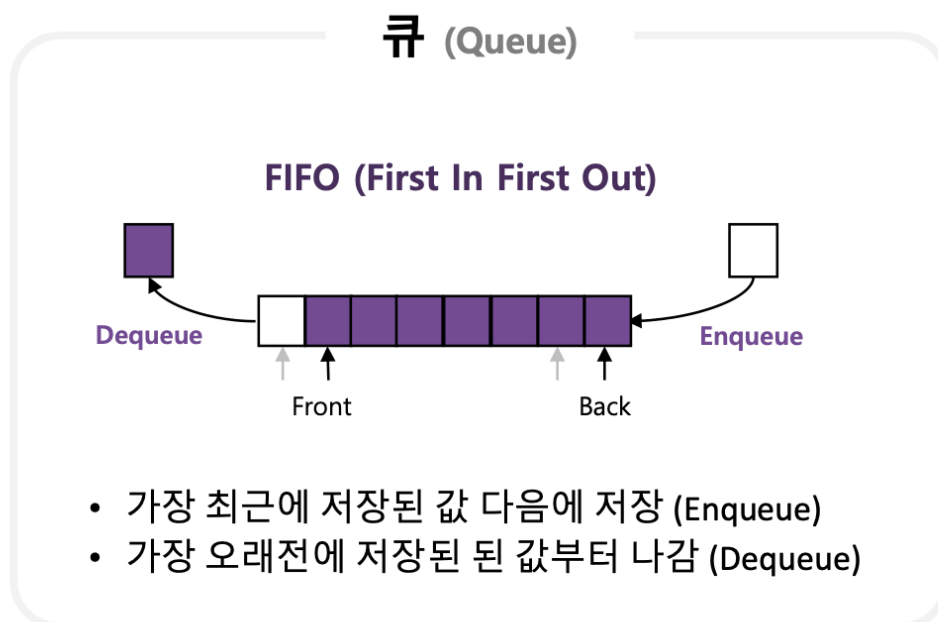
- 멀티 태스킹을 위한 프로세스 스케줄링 방식을 구현하기 위해 많이 사용됨
 - 큐의 경우에는 장단점 보다는(특별히 언급되는 장단점이 없음), 큐의 활용 프로세스 스케줄링 방식을 함께 이해해두는 것이 좋음

사용 예(시간 순서대로 처리)

- 데이터를 입력된 순서대로 처리할 때 (콜센터 대기 순서)
- 프로세스 관리
- 은행 업무
- 우선순위가 같은 작업 예약
- BFS(너비 우선 탐색) 알고리즘
- 캐시(Cache) 구현

1. 큐 구조

- 줄을 서는 행위와 유사
- 가장 먼저 넣은 데이터를 가장 먼저 꺼낼 수 있는 구조
 - 음식점에서 가장 먼저 줄을 선 사람이 제일 먼저 음식점에 입장하는 것과 동일
 - **FIFO** (First-In, First-Out) 또는 **LILO**(Last-In, Last-Out) 방식으로 **스택과 꺼내는 순서가 반대**



2. 알아둘 용어

- **Enqueue:** 큐에 데이터를 넣는 기능
- **Dequeue:** 큐에서 데이터를 꺼내는 기능
- Visualgo 사이트에서 시연

3. 파이썬 queue 라이브러리 활용해서 큐 자료 구조 사용하기

- queue 라이브러리에는 다양한 큐 구조로 Queue(), LifoQueue(), PriorityQueue() 제공
- 프로그램을 작성할 때 프로그램에 따라 적합한 자료 구조를 사용
 - Queue(): 가장 일반적인 큐 자료 구조(FIFO)

```
import queue
data_queue = queue.Queue()

data_queue.put('funcoding')
data_queue.put(2)
data_queue.qsize() # 2, 데이터 크기
data_queue.get()   # 'funcoding' : 먼저 들어간 값 나옴
data_queue.qsize() # 1
data_queue.get()   # 2, 나머지 값 나옴
data_queue.qsize() # 0
```

- LifoQueue(): 나중에 입력된 데이터가 먼저 출력되는 구조 (스택 구조라고 보면 됨) (LIFO)

```
import queue
data_queue = queue.LifoQueue()

data_queue.put('funcoding')
data_queue.put(3)
data_queue.qsize() # 2
data_queue.get()   # 3: 나중에 들어간 값 나옴
data_queue.qsize() # 1
data_queue.get()   # 'funcoding' 나머지 값 나옴
data_queue.qsize() # 0
```

- PriorityQueue(): 데이터마다 우선순위를 넣어서, 우선순위가 높은 순으로 데이터 출력

```

import queue
data_queue = queue.PriorityQueue()

data_queue.put((2, 'yeardream'))
data_queue.put((5, 1))
data_queue.put((10, 'ssafy'))
data_queue.qsize() # 3
data_queue.get()   # (2, 'yeardream'), 2등 이니까 먼저 출력
data_queue.qsize() # 2
data_queue.get()   # (5, 1)

```

4. 구현

```

queue_list = list()

def enqueue(data):
    queue_list.append(data)

def dequeue():
    data = queue_list[0]
    del queue_list[0]
    return data

for index in range(10):
    enqueue(index)
len(queue_list) # 10
dequeue()       # 0, 1, 2, 3, ..., 9

```

5. 시간복잡도 Big O

- Insertion $O(1)$
- Deletion $O(1)$
- Search $O(n)$

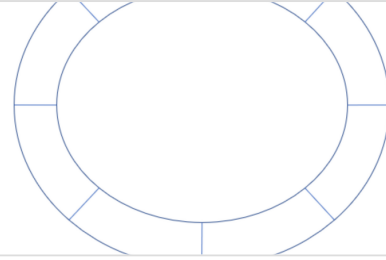
+ Circular Queue

배열로 구현 된 선형 큐의 경우 데이터의 삽입/삭제 시 데이터들을 앞으로/뒤로 당겨주는 과정이 필요해 최악의 경우 $O(n)$ 의 시간복잡도를 가지게 됨. 이러한 선형 큐의 단점을 극복한 구조가 원형 큐임

파이썬으로 구현하는 원형 큐(Circular Queue)

배열로 구현된 선형 큐(Linear Queue)의 경우 데이터의 삽입/삭제 시 데이터들을 앞으로/뒤로 당겨주는 과정이 필요해 최악의 경우 $O(n)$ 의 시간복잡도를 가지게 된다.

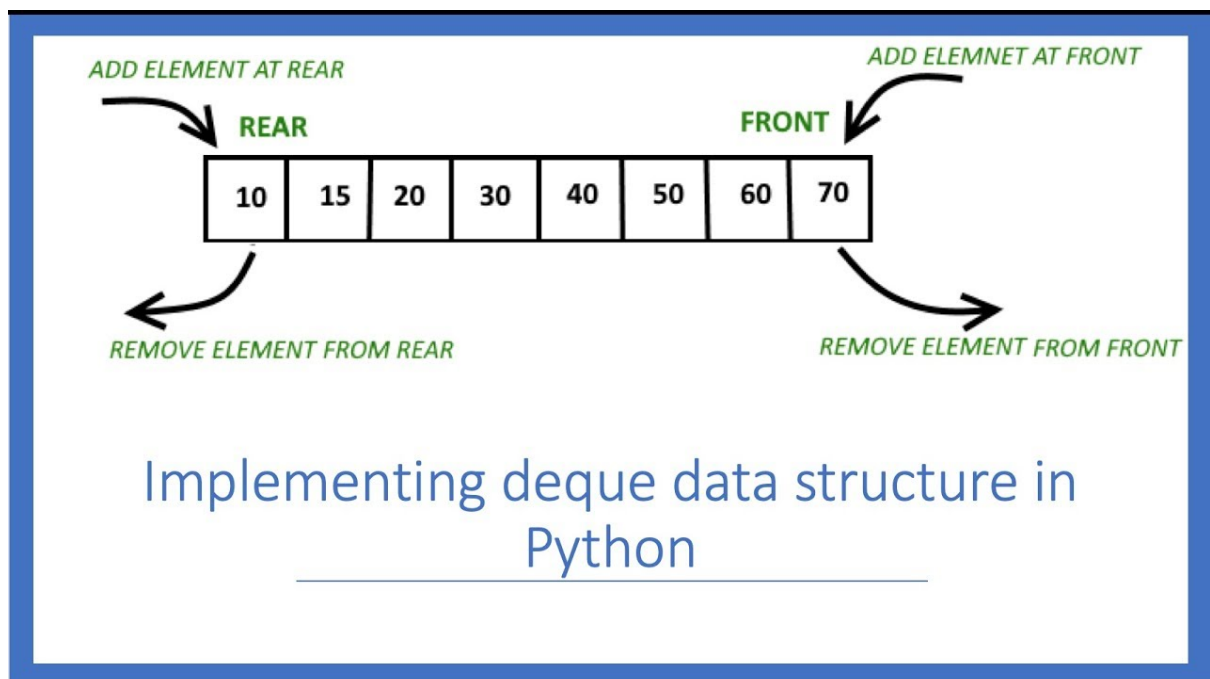
🔗 <https://codingsmu.tistory.com/123>



+ Deque

double-ended queue로, 큐의 앞과 뒤 모두에서 삽입 및 삭제가 가능한 큐임

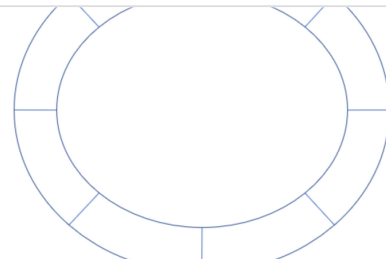
덱은 원형 큐를 확장하면 손쉽게 구현 가능



파이썬으로 구현하는 덱(Deque)

double-ended queue로, 큐의 앞과 뒤 모두에서 삽입 및 삭제가 가능한 큐를 의미한다. 덱은 원형 큐(Circular Queue)를 확장하면 손쉽게 구현할 수 있다.

🔗 <https://codingsmu.tistory.com/124?category=871719>



- 스택/큐 요약

자료구조	동작	코드	설명
큐	초기화	<code>queue = []</code>	빈 리스트를 만듦
	자료 추가(enqueue)	<code>queue.append(data)</code>	리스트의 맨 뒤에 자료를 추가
	자료 삭제(dequeue)	<code>data = queue.pop(0)</code>	리스트의 맨 앞(0번 위치)에서 자료를 꺼냄
스택	초기화	<code>stack = []</code>	빈 리스트를 만듦
	자료 추가(push)	<code>stack.append(data)</code>	리스트의 맨 뒤에 자료를 추가
	자료 삭제(pop)	<code>data = stack.pop()</code>	리스트의 맨 뒤에서 자료를 꺼냄