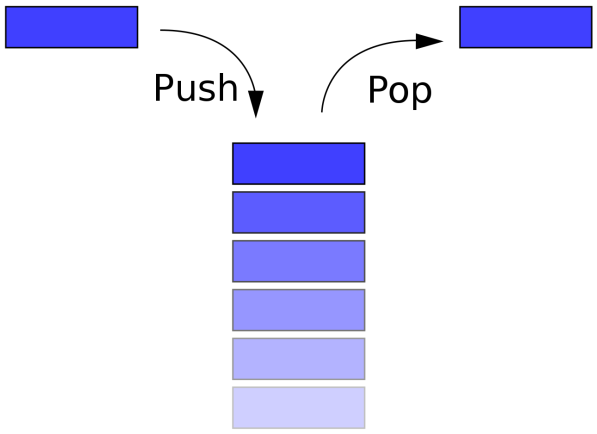
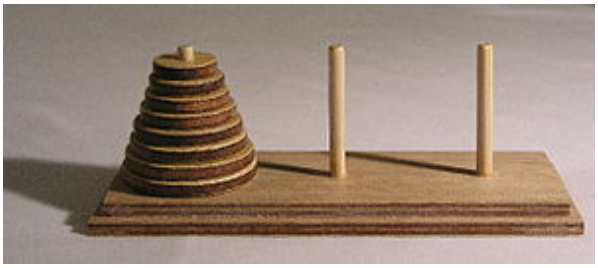
스터디 3주차 (엄태호)

스택 (Stack)



출처: https://ko.wikipedia.org/wiki/스택



출처:
https://en.wikipedia.org/wiki/Tower_of_Hanoi

- 스택은 LIFO(Last In First Out, 후입선출) 구조를 가진 자료구조로 위 그림처럼 한 방향에서만 데이터를 넣고 뺄 수 있다.
- 즉, 1개 이상의 데이터를 넣은 뒤 데이터를 뺄 때는 가장 최근에 삽입된 데이터부터 처음에 삽입된 데이터순으로 데이터가 삭제된다.
- 오른쪽 사진과 같은 하노이의 탑을 연상하면 이해가 쉽다.
  가장 최근에 쌓인 원판일수록 상단에 쌓이게 되며 원판을 제거할 때는 상단 → 하단으로만 제거가 가능하다.

스택의 구성요소

- `push(element)`
  스택에 요소를 추가할 때 사용되는 메소드로 가장 최근에 push를 통해 삽입된 요소가 가장 처음으로 삭제된다.
- `pop()`
  스택에서 요소를 제거할 때 사용되는 메소드로 가장 마지막에 삽입된 요소를 반환한다.

- `top`
  push메소드가 호출 될 경우 새로 삽입될 요소의 위치를 가르키며 요소가 비어있는 위치이다.
- `peek()`
  `top` -1 값과 같으며 가장 마지막에 삽입된 요소를 가르키고 있다.
  pop메소드와 반환 값은 같지만 요소를 삭제하지 않는다는 점이 다르다.

스택 구현

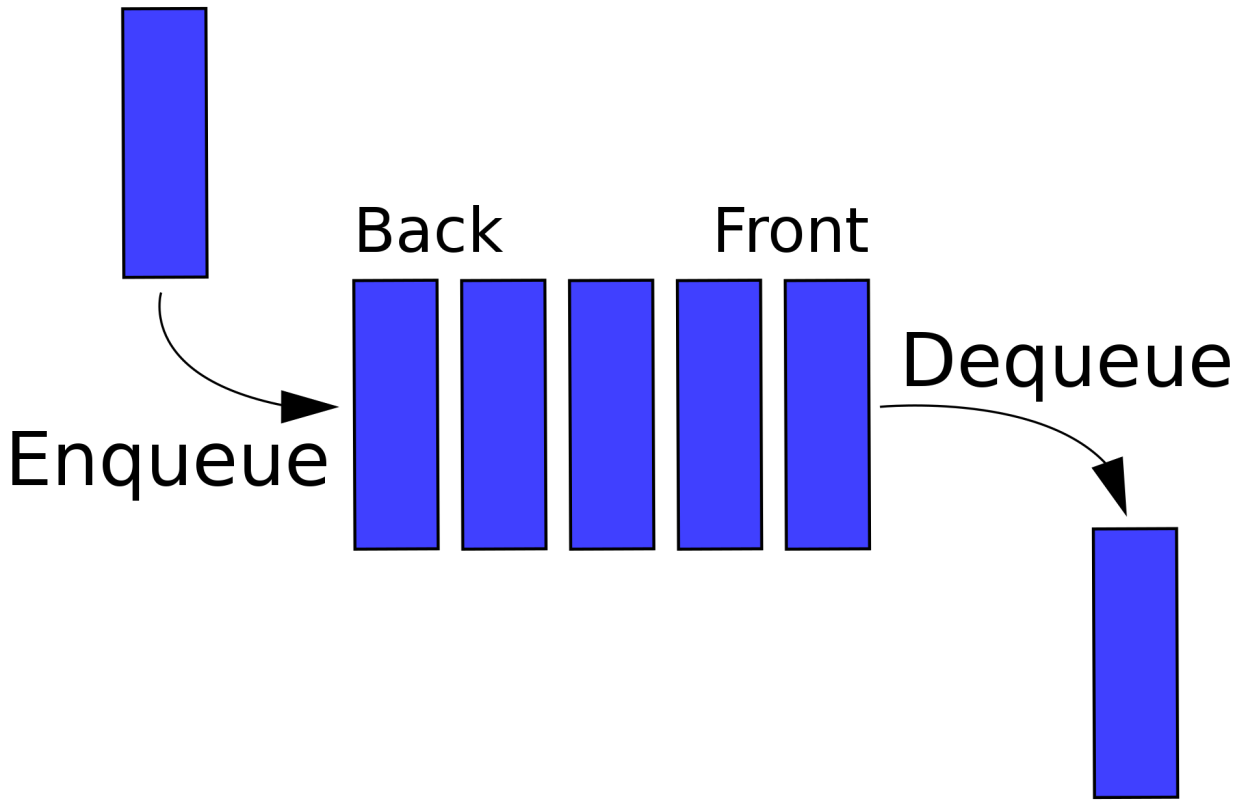
- 배열의 빌트인 메소드를 사용하지 않고 구현하였다.

```
class Stack {
    #top = 0;

    constructor() {
        this.data = [];
    }

    push(element) {
        this.data[this.#top++] = element;
    }

    pop() {
        if (this.#top === 0) {
            return;
        }

        return this.data[--this.#top];
    }

    peek() {
        if (this.#top === 0) {
            return;
        }

        return this.data[this.#top-1];
    }
}
```

스택 사용 예시

- 재귀 함수
- 웹페이지 방문기록
- 실행취소
- 진법변환
- 회문판별 및 역순문자열

큐 (Queue)



출처: https://ko.wikipedia.org/wiki/큐_(자료_구조)

- 큐는 FIFO (First In First Out, 선입선출)구조를 가진 자료구조로 스택과 다르게 한쪽 방향에서 삽입, 한쪽 방향에서 삭제가 이루어진다.
- 식당에서 웨이팅을 하거나 드라이브 스루등이 실생활에서의 대표적인 예이다.
- 다른 큐 종류와 구분하기 위해 선형 큐(Linear Queue)라고 부르기도 하며 일반적으로 큐를 말할 땐 이 선형 큐를 가르킨다.

큐의 구성요소

- `enqueue(element)`
  큐에 데이터를 삽입할 때 사용하는 메소드이다. 가장 최근에 enqueue에 의해 삽입된 데이터는 가장 마지막에 삭제된다.
- `dequeue()`
  큐에서 데이터를 삭제할 때 사용하며 큐 내부 요소 중 가장 첫 번째로 삽입된 요소가 반환된다.
- `peek()`
  큐에 저장된 요소 중 가장 먼저 저장된 요소를 반환하며 삭제하진 않는다.
- `clear()`
  큐에 저장된 요소들을 모두 삭제해 초기화시킨다.
- `length`
  큐에 저장된 요소들의 개수를 반환한다.

큐 구현해보기

- 배열의 빌트인 메소드를 사용하지 않고 구현해보았다.

```
class Queue {
    #rear = 0;
    #front = 0;

    constructor() {
        this.data = [];
    }

    enqueue(element) {
        this.data[this.#rear++] = element;
    }

    dequeue() {
        if (this.#front === undefined) {
            return;
        }

        const item = this.data[this.#front];
```

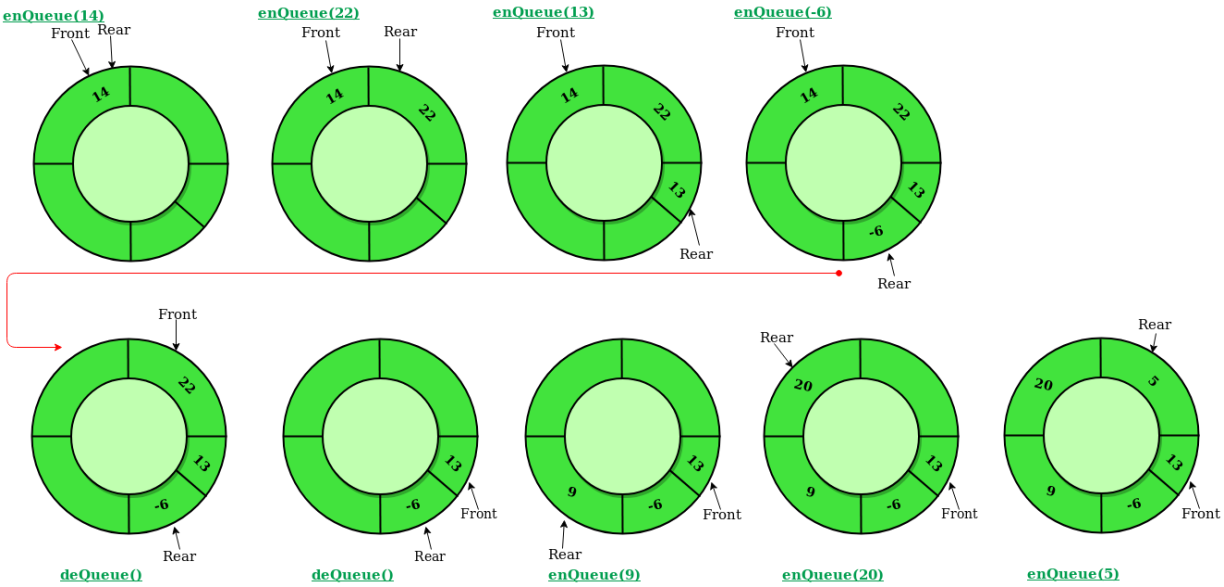
```
        delete this.data[this.#front--];

        return item;
    }

    peek() {
        if (this.#rear === 0) {
            return;
        }

        return this.data[this.#rear - 1];
    }

    clear() {
        this.data = [];
        this.#rear = 0;
        this.#front = 0;
    }

    get length() {
        return this.data.length;
    }
}
```

큐 사용 예시

- 프로세스 관리
- BFS 알고리즘 (그래프 탐색 알고리즘 중 하나)
- 프린터의 대기열

큐의 종류

- 원형 큐 (= 환형 큐, Circular Queue, Ring Buffer)



https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/

- 선형 큐의 문제점을 보완하기 위한 자료구조로 **선형 큐는 rear가 큐 사이즈의 최대치의 위치하게 되면 큐의 앞쪽이 비어있더라도 데이터를 삽입하지 못한다.**
  그래서, 앞쪽 인덱스가 비어있을 때 모든 요소를 순환하여 앞쪽으로 재배치 시켜주는 작업이 필요하다.
- 하지만, 원형 큐의 경우 front와 rear가 앞뒤로 순환하기 때문에 요소를 재배치하는데 들어가는 비용이 소모되지 않는다는 장점이 있다.
  front와 rear는 `front + 1 % size` 형태로 계속해서 순환한다.

- 우선순위 큐 (Priority Queue)

- 우선순위 큐는 큐의 모든 요소에 우선순위를 설정해 요소를 꺼낼 때 우선순위가 높은 요소가 우선순위가 낮은 요소보다 먼저 삭제되는 자료구조이다.
- 즉, 삽입순서를 따르는 일반 큐와는 달리 우선순위를 기준으로 데이터를 꺼낸다.
  다만, 우선순위가 같은 요소가 여러개인 경우에는 삽입순서를 따른다.
- 우선순위 큐는 배열, 링크드리스트, 힙, 이진탐색트리 등으로 구현할 수 있다.
  다만, 배열과 링크드리스트의 경우 데이터를 꺼낼 때 `O(n)`의 시간복잡도를 가지기 때문에 일반적으로 `O(log n)`의 시간복잡도를 가지는 힙 자료구조를 활용해 구현한다.
- **우선순위 큐의 구현 방식에 따른 시간 복잡도**

	삽입	삭제	우선순위가 높은 데이터 탐색

	삽입	삭제	우선순위가 높은 데이터 탐색
배열	O(1)	O(n)	O(n)
링크드 리스트	O(n)	O(1)	O(1)
바이너리 힙	O(log n)	O(log n)	O(1)
BST	O(log n)	O(log n)	O(1)

우선순위 큐 구현해보기

- 우선순위 큐를 배열로 구현한 예시로 가장 기본적인 속성들인 `enqueue`, `dequeue`, `peek` 3가지 메소드만 구현하였다.

```
class Item {
    constructor(value, priority) {
        this.value = value;
        this.priority = priority;
    }
}

class PriorityQueue {
    constructor() {
        this.queue = [];
    }

    enqueue(element) {
        if (this.queue.length === 0) {
            this.queue.push(element);
            return;
        }

        let added = false;

        for (let i = 0; i < this.queue.length; i++) {
            if (element.priority > this.queue[i].priority) {
                this.queue.splice(i, 0, element);
                added = true;
                break;
            }
        }

        // 추가가 되지 않았다면 우선순위가 가장 낮으므로 맨 뒤에 위치해야함.
        if (!added) {
            this.queue.push(element);
        }
    }
```

```
    dequeue() {
        if (this.queue.length === 0) {
            return;
        }

        return this.queue.shift().value;
    }

    peek() {
        if (this.queue.length === 0) {
            return;
        }

        return this.queue[0].value;
    }
}
```