3주차: 스택과 큐

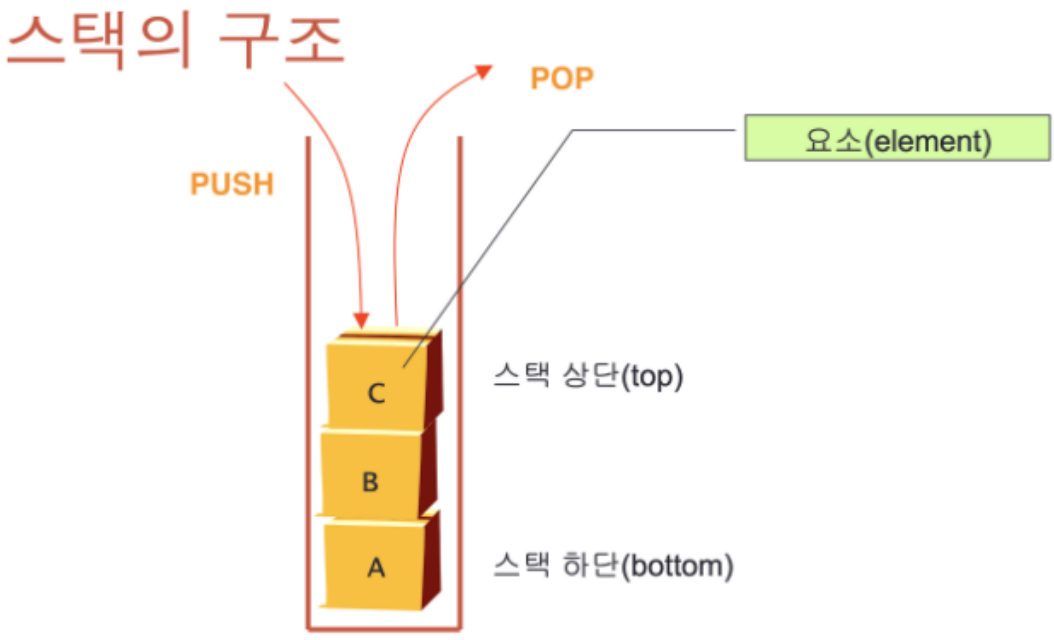
스택(stack)

- 스택은 연결리스트인데 뒤로 넣고 뒤로만 뺄 수 있습니다. push와 pop만 할 수 있으며,

  스택은 실행이 되는 특정한 순서를 따르는 **선형적** 데이터 구조입니다.

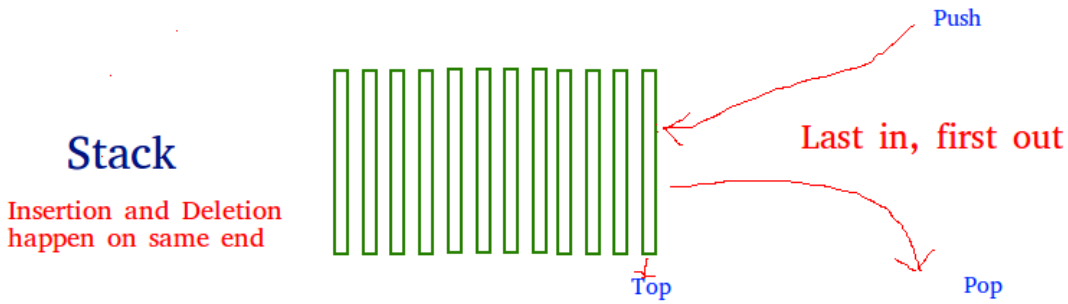
  위에서 스택은 연결리스트로 뒤로 넣고 뒤로만 뺄 수 있다고 했는데..

  스택의 구조를 보면 위에서 넣고 위에서 빼고 있습니다.



순서는 두 가지가 존재합니다.

- 첫째, LIFO(Last In First Out) : 마지막에 들어온 데이터가 먼저가 먼저 나가는 데이터 구조로, 흔히 **후입선출** 구조라고도 부릅니다.
- 둘째, FILO(First In Last Out) : 처음 들어온 데이터가 마지막에 나가는 데이터 구조로, 흔히 **선입후출**라고도 이야기합니다.



3가지 기본 작업이 스택에서 수행됩니다.

- Push : 스택 안으로 데이터를 넣습니다. 만약 스택이 꽉 찬 상태에서 데이터를 더 넣을 경우, Overflow condition(오버플로우 상태)라고 이야기합니다.
- Pop : 스택 안에 있는 데이터를 제거합니다. 아이템은 넣어진 역순으로 빠져나옵니다. 만약 스택이 비어있는 상태에서 데이터를 꺼내려는 경우, Underflow condition(언더플로우 컨디션)라고 이야기합니다.
- Peek or Top : 스택의 꼭대기 요소를 반환합니다.
- isEmpty : 만약 스택이 비어있다면, true를 반환하고, 비어있지 않다면 false를 반환합니다.

#구현

스택을 구현하는 두 가지 방법이 존재하는데,

배열(array)을 사용하거나 **링크드 리스트(linked list)**를 사용합니다.

1.스택을 배열을 사용할 경우

```
var Stack = (function() {
  function Stack() {
    this.top = null;
    this.count = 0;
  }
  function Node(data) {
    this.data = data;
```

```
    this.next = null;
  }
  Stack.prototype.push = function(data) {
    var node = new Node(data);
    node.next = this.top;
    this.top = node;
    return ++this.count;
  };
  Stack.prototype.pop = function() {
    if (!this.top) { // stack underflow 방지
      return false;
    }
    var data = this.top.data;
    this.top = this.top.next;
    // 예전 this.top의 메모리 정리
    this.count--;
    return data;
  };
  Stack.prototype.stackTop = function() {
    return this.top.data;
  };
  return Stack;
})();
```

호출 결과

```
var stack = new Stack();
stack.push(1); // 1
stack.push(3); // 2
stack.push(5); // 3
stack.pop(); // 5
stack.stackTop(); // 3
```

배열을 이용했을 경우의 장점과 단점(pros and cons)

장점: 구현하기 쉽다. 포함되지 않은 포인터로서 메모리는 저장된다.

단점: 다이내믹하지 않다. 런타임에 의해서 배열이 증가하거나 작아지지 않는다.

2.링크드리스트를 사용할 경우

```
var LinkedList = (function() {
  function LinkedList() {
    this.length = 0;
    this.head = null;
  }
  function Node(data) {
```

```
    this.data = data;
    this.next = null;
  }
  LinkedList.prototype.add = function(value) {
    var node = new Node(value);
    var current = this.head;
    if (!current) { // 현재 아무 노드도 없으면
      this.head = node; // head에 새 노드를 추가합니다.
      this.length++;
      return node;
    } else { // 이미 노드가 있으면
      while(current.next) { // 마지막 노드를 찾고.
        current = current.next;
      }
      current.next = node; // 마지막 위치에 노드를 추가합니다.
      this.length++;
      return node;
    }
  };
  LinkedList.prototype.search = function(position) {
    var current = this.head;
    var count = 0;
    while (count < position) { // position 위치만큼 이동합니다.
      current = current.next;
      count++;
    }
    return current.data;
  };
  LinkedList.prototype.remove = function(position) {
    var current = this.head;
    var before;
    var remove;
    var count = 0;
    if (position == 0) { // 맨 처음 노드를 삭제하면
      remove= this.head;
      this.head = this.head.next; // head를 두 번째 노드로 교체
      this.length--;
      return remove;
    } else { // 그 외의 다른 노드를 삭제하면
      while (count < position) {
        before = current;
        count++;
        current = current.next;
      }
      remove = current;
      before.next = remove.next;
      // remove 메모리 정리
      this.length--;
      return remove;
    }
  };
  return LinkedList;
})();
```

호출 결과

```
var list = new LinkedList();
list.add(1);
list.add(2);
list.add(3);
list.length; // 3
list.search(0); // 1
list.search(2); // 3
list.remove(1);
list.length; // 2
```

링크드리스트의 장점과 단점(pros and cons)

장점: 런타임에 의해 스택의 실행이 증가하거나 줄어듭니다.

단점: 포인터를 포함해야 하기 때문에 추가적인 메모리가 필요로 합니다.

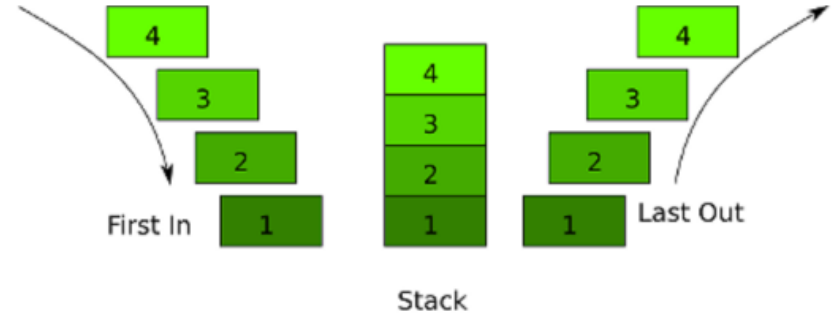
#스택에 관련한 시간 복잡도

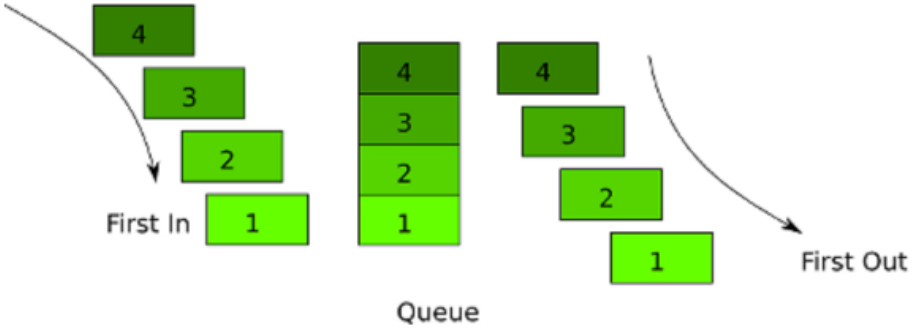
스택에서 사용되는 기본 작업들인 push, pop, isEmpty와 peek 메서드는 모두 O(1)의 시간 복잡도를 가집니다.

#큐

스택과 같이 큐는 같은 선형적 구조는 특정한 순서를 따릅니다.

순서는 FIFO(First In First Out) **선입선출**입니다.





스택과 큐의 차이점은 **제거**에 있습니다.

스택은 가장 최근에 추가된 데이터를 제거하지만, 큐는 가장 처음에 들어왔던 데이터를 삭제한다.

다시 이야기하면, **스택**은 가장 마지막에 쌓인 데이터를 제거하고 **큐**는 가장 앞에 추가된 데이터를 삭제한다.

#큐의 동작

주로 아래에 나오는 4가지 기본 동작에 의해 동작한다.

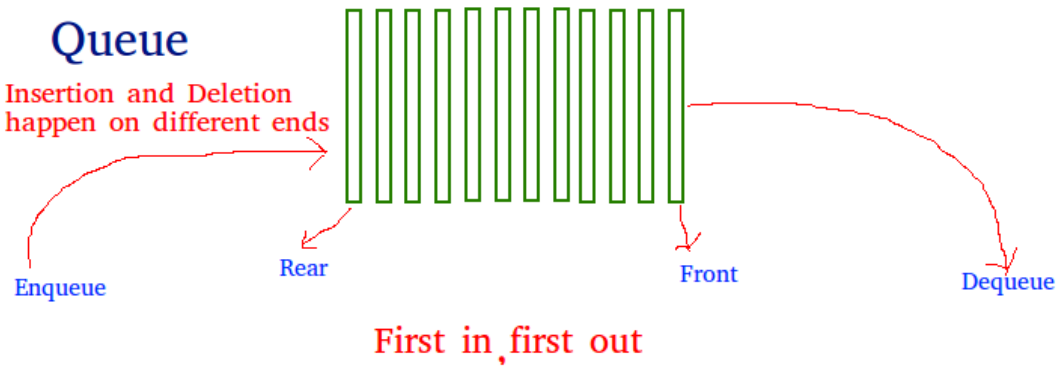
- Enqueue: 큐에 데이터를 추가한다. 만약 큐가 꽉 찼을 때, Overflow condition이 된다.
- Dequeue: 큐에서 데이터를 제거한다. 데이터는 들어온 순서대로 빠져나온다. 만약 큐가 비어져 있을 경우, Underflow condition이 된다.
- Front: 큐의 앞부분의 데이터를 가진다.
- Rear: 큐의 뒷부분의 데이터를 가진다.

쉽게 생각해서,

BTS 팬 사인회가 있어서, 많은 사람들이 줄을 서있습니다.

사람들은 줄을 서고 줄에서 나갑니다.

여기서 큐는 줄이라고 생각할 수 있고, 사람이 줄을 서는 것(Enqueue)과 줄을 빠지는 것(Dequeue)라고 볼 수 있습니다.



이미지를 함께 보시면, 위에서 이야기한 큐의 4가지 기본 작업에 대해 표현하고 있습니다.

스택에서는 제일 위를 가리키는 top이 있다면, 큐에는 맨 처음을 가리키는 head와 맨 끝을 가리키는 rear, 두 개가 있습니다.

#큐의 종류

큐를 구현하기 위해서, 우리는 두 가지 지표를 추적해야 합니다. - front(머리)와 rear(꼬리)

우리는

rear에 데이터를 추가하고 (enqueue),

front에 데이터를 빼냅니다 (dequeue).

만약, 우리가 간단히 front와 rear의 지표를 증가시킨다면, 문제가 될 수 있습니다.

> front가 rear에 (배열의 끝)에 도달할 수 있기 때문입니다.

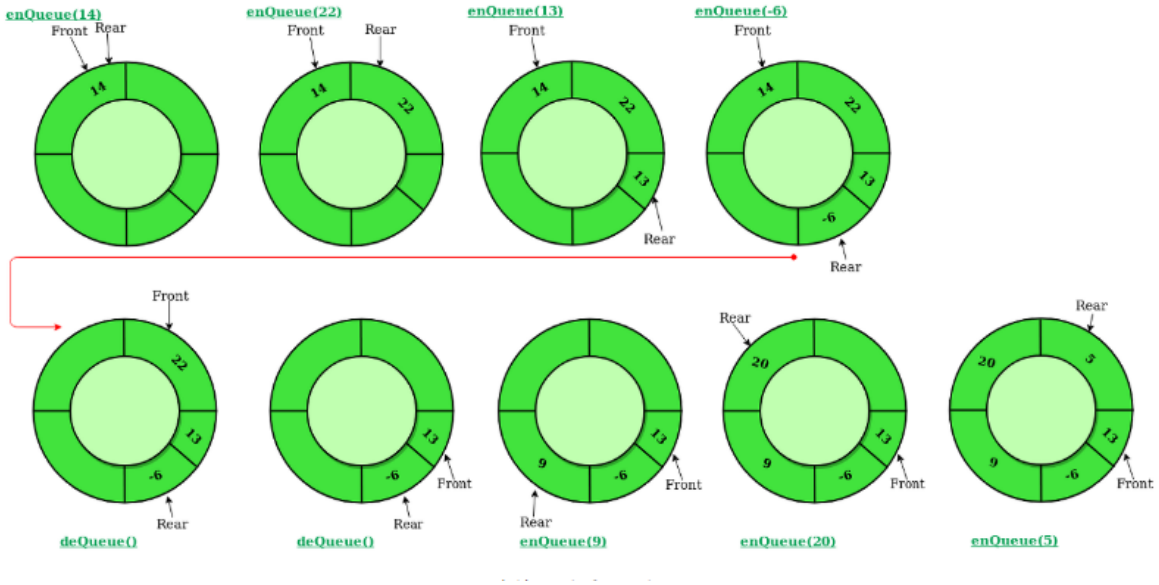
1. 순환 큐

여기에 대한 방법으로는 front와 rear를 **circular Queue(순환 큐)**(으)로 증가시켜주는 방법이 있습니다.

front와 rear가 연결되어 있는데요, 원래 1, 2, 3의 큐에선 3에서 끝나지만,

순환 큐는 rear인 3에서 다시 front인 1로 넘어갑니다.

```
this.rear.next = this.front
```



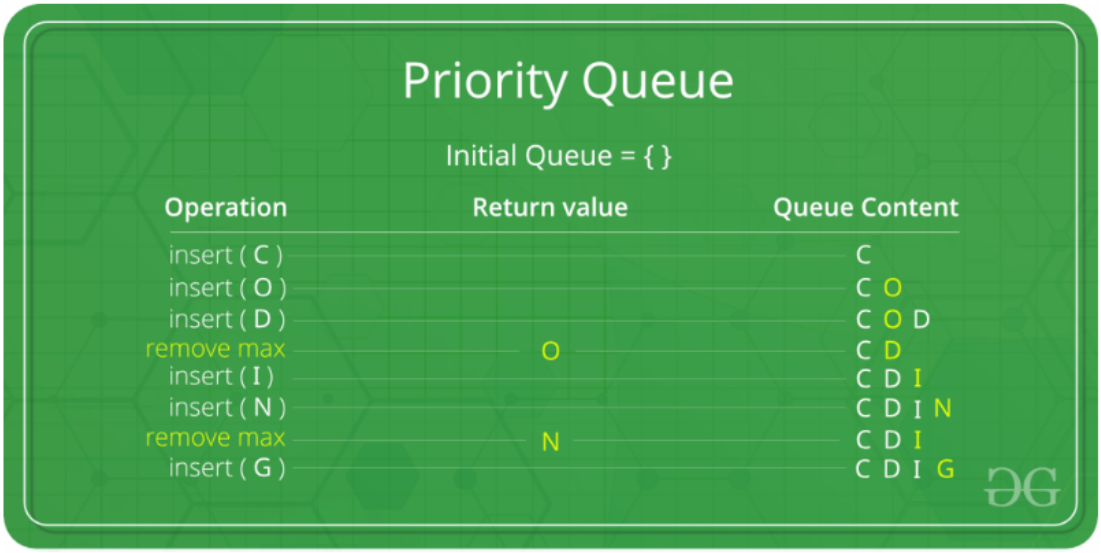
순환 큐가 사용되는 이유는 메모리 관리 측면인데,

자바스크립트에서는 메모리를 알아서 정리해주기 때문에, 효용성이 조금 떨어진다고 합니다!

2. 우선순위 큐

우선순위 큐는 enqueue와 dequeue는 같지만, enqueue 할 때, 제일 뒤에 넣는 것이 아닌, 우선순위를 따져 데이터를 넣습니다.

우선순위는 프로그래머가 직접 정해주면 됩니다. 다만, 우선순위 큐의 문제로, 보통 큐와 같이 구현하면 데이터를 삽입하기 힘들다는 단점이 있어 주로 힙 이라는 자료구조를 사용해서 구한다고 합니다.



#큐 구현하기

```
var Queue = (function() {
  function Queue() {
    this.count = 0;
    this.head = null;
    this.rear = null;
  }
  function Node(data) {
    this.data = data;
    this.next = null;
  }
  Queue.prototype.enqueue = function(data) {
    var node = new Node(data);
    if (!this.head) {
      this.head = node;
    } else {
      this.rear.next = node;
    }
    this.rear = node;
    return ++this.count;
  };
  Queue.prototype.dequeue = function() {
    if (!this.head) { // stack underflow 방지
      return false;
    }
    var data = this.head.data;
    this.head = this.head.next;
    // this.head 메모리 클린
    --this.count;
    return data;
```

```
  };
  Queue.prototype.front = function() {
    return this.head && this.head.data;
  };
  return Queue;
})();
```

큐 호출하기

```
var queue = new Queue();
queue.enqueue(1); // 1
queue.enqueue(3); // 2
queue.enqueue(5); // 3
queue.dequeue(); // 1
queue.front(); // 3
```

#큐의 시간 복잡도

Enqueue (insertion) : O(1)

Dequeue (deletion) : O(1)

Front (Get front) : O(1)

Rear (Get Rear) : O(1)

보조 공간: input size를 고려하여 알고리즘이 문제를 해결하기 위해 임시로 사용하는 공간

: O(N)

배열로 큐를 구현할 때의 장단점

장점: 구현하기 쉽다.

단점: 사이즈가 고정되어 있다 (정적인 데이터 구조)

만약에 enqueue와 dequeue로 큐가 큰 숫자를 가졌다면, 큐가 비어있더라도 큐에 요소를 삽입하지 못할 수 있다. (순환 큐로 대처할 수 있다.) 링크드리스트로 큐를 구현하는 게 더 쉽다.