

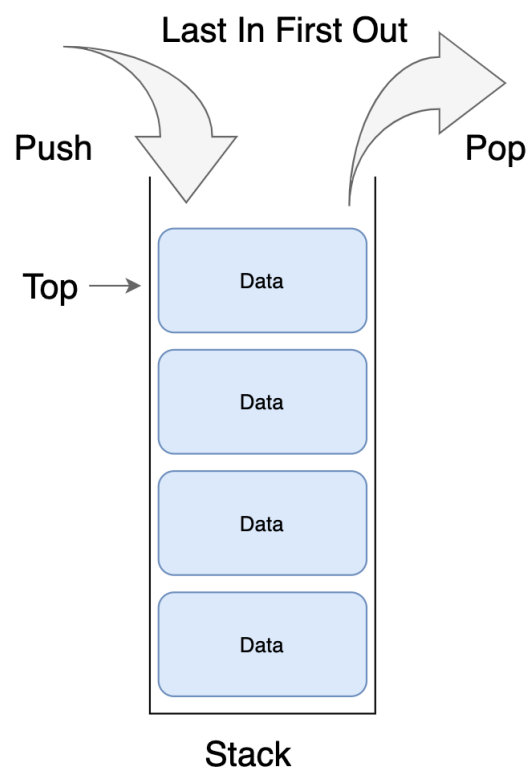
## 2

# 자료구조

## 스택(Stack)



**Stack**은 후입선출(**LIFO**, Last In First Out) 형식으로 데이터를 저장하는 자료구조이다.



- 후입선출은 시간 순서상 가장 최근에 추가한 데이터가 가장 먼저 나오는 방식이다.
- Queue와 달리 top에서 삽입(push)과 삭제(pop)이 모두 이루어진다.
- 삽입/삭제 연산은  $O(1)$ 의 시간복잡도를 갖는다.

스택은 단독으로도, 다른 자료구조를 구현하는 데에도 자주 사용된다.



### Stack의 다양한 활용

1. LIFO 특성을 활용한 문제
2. DFS(깊이 우선 탐색)에 사용
3. 재귀
4. 괄호 적합성 판단 등

## 스택의 사용법

- 자바에서는 **java.util.Stack** 이라는 클래스를 통해서 스택의 동작 기능을 제공한다.

```
import java.util.Stack;

public class StackUse {
    public static void main(String[] args) {

        Stack<String> stack = new Stack<>();

        stack.push("data1");
        stack.push("data2");
        stack.push("data3");
        System.out.println(stack);

        System.out.println(stack.pop());

        System.out.println(stack);

        System.out.println(stack.peek());

        System.out.println(stack.empty());
    }
}
```

### ▼ 결과

[data1, data2, data3]

data3

[data1, data2]

data2

false

### pop()

맨 마지막에 넣은 데이터를 가져오면서 삭제.  $O(1)$

### push()

새로운 데이터를 맨 마지막에 추가.  $O(1)$

### peek()

맨 마지막(top) 데이터를 반환.  $O(1)$

### empty() , isEmpty()

스택이 비어있는지 확인.  $O(1)$

### contains()

특정 데이터가 스택에 포함되어 있는지를 확인.  $O(n)$

### clear()

스택에 있는 모든 데이터를 null값으로 할당.

### size()

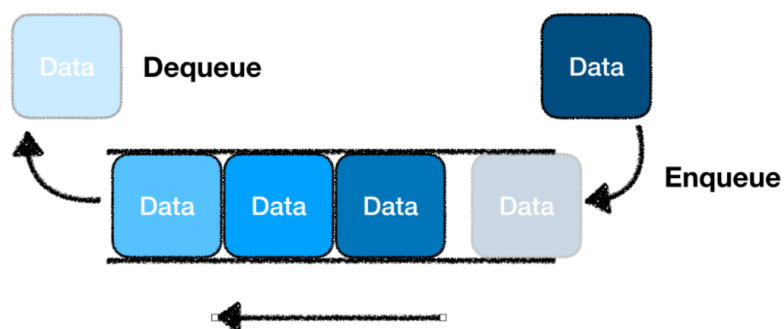
현재 스택에 들어있는 데이터의 개수를 반환.

## 큐(Queue)



### Queue

queue는 **FIFO**(First In First Out)형식으로 데이터를 **선입선출**하도록 저장하는 자료구조이다.



- **enqueue:** 큐의 맨 뒤(rear)에 데이터를 추가
- **dequeue:** 큐의 맨 앞(front)에서 데이터를 삭제
- 스택과 달리 데이터 삽입은 rear에서 삭제는 front에서 이루어진다.



## Stack의 다양한 활용

### 1. BFS(넓이 우선 탐색)에 사용

## 큐의 사용법

- 자바에서 큐는 LinkedList로 생성되어 있어서 Queue와 LinkedList 모두 import 해주어야 한다.

```
import java.util.LinkedList;
import java.util.Queue;

public class StackUse {
    public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();

        queue.offer("data1");
        queue.offer("data1");
        queue.add("data1");
        queue.add("data1");

        System.out.println(queue);

        System.out.println(queue.remove());
        System.out.println(queue.poll());

        System.out.println(queue);
    }
}
```

### ▼ 결과

```
[data1, data2, data3, data4]
data1
data2
[data3, data4]
```

### **add()**

큐에 데이터 추가. 큐에 공간이 없다면 `IllegalStateException` 발생

### **offer()**

큐에 데이터 추가. 값 추가에 실패한다면 `false` 반환.

### **remove()**

큐의 맨 앞의 데이터를 반환하고 삭제. 값이 없다면 `NoSuchElementException` 발생

### **poll()**

큐의 맨 앞의 데이터를 반환하고 삭제. 값이 없다면 `null` 반환.

### **peek()**

큐의 맨 앞의 데이터를 반환. 값이 없다면 `null` 반환.

### **element()**

큐의 맨 앞의 데이터를 반환. 없다면 `NoSuchElementException` 발생

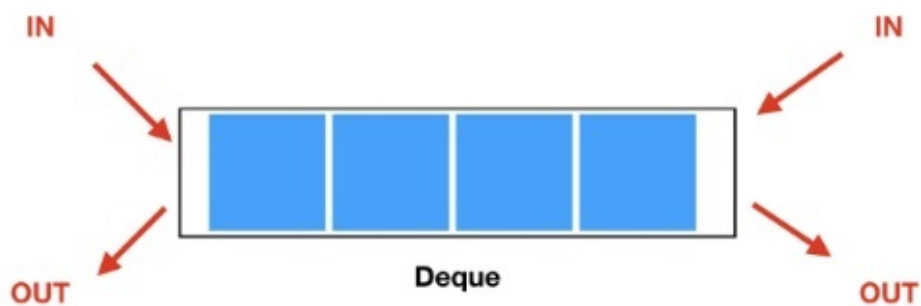
### **clear()**

큐 비우기

## 덱(Deque)



**Deque**은 Double-Ended Queue의 줄임말로 **큐의 양쪽으로 데이터 삽입과 삭제**를 수행할 수 있는 자료구조 이다.



- 덱은 입력하고 출력하는 방향에 따라서 스택으로 사용할 수도, 큐로 사용할 수도 있다.

- 한쪽으로만 입력 가능하도록 설정한 덱을 스크롤(scroll)
- 한쪽으로만 출력 가능하도록 설정한 덱을 셸프(shelf)라고 한다.

## 덱의 사용법

- 자바에서의 Deque은 인터페이스에 정의되어 있고, 이를 구현한 ArrayDeque, LinkedBlockingDeque, ConcurrentLinkedDeque, LinkedList 등의 클래스가 있다.

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.LinkedList;
import java.util.concurrent.ConcurrentLinkedDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class StackUse {
    public static void main(String[] args) {

        Deque<String> deque1 = new ArrayDeque<>();
        Deque<String> deque2 = new LinkedBlockingDeque<>();
        Deque<String> deque3 = new ConcurrentLinkedDeque<>();
        Deque<String> linkedList = new LinkedList<>();

    }
}
```

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeUse {
    public static void main(String[] args) {

        // 스택 구현
        Deque<String> stack= new ArrayDeque<>();

        stack.addFirst("Element1");
        stack.addFirst("Element2");
        stack.addFirst("Element3");
        System.out.println(stack.removeFirst());
        System.out.println(stack.removeFirst());
        System.out.println(stack.removeFirst());

        // 큐 구현
        Deque<String> queue = new ArrayDeque<>();

        queue.addFirst("Element1");
        queue.addFirst("Element2");
        queue.addFirst("Element3");
        System.out.println(queue.removeLast());
        System.out.println(queue.removeLast());
    }
}
```

```

        System.out.println(queue.removeLast());
    }
}

```

## Deque에 값 삽입

### **addFirst(), push()**

덱의 앞쪽에 데이터를 삽입. 덱의 용량을 초과한다면 Exception 발생.

### **offerFirst()**

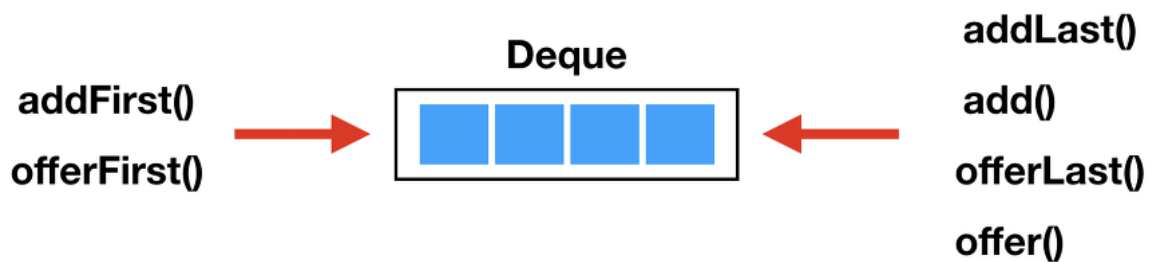
덱의 앞쪽에 데이터를 삽입. 덱의 용량을 초과한다면 false 반환.

### **addLast(), add()**

덱의 마지막에 데이터를 삽입. 덱의 용량을 초과한다면 Exception 발생

### **offerLast(), offer()**

덱의 마지막에 데이터를 삽입. 덱의 용량을 초과한다면 false 반환.



## Deque에 값 삭제

### **removeFirst(), remove(), pop()**

덱의 앞쪽에서 데이터를 뽑아 제거하고 값을 반환. 덱이 비어있다면 Exception 발생.

### **pollFirst(), poll()**

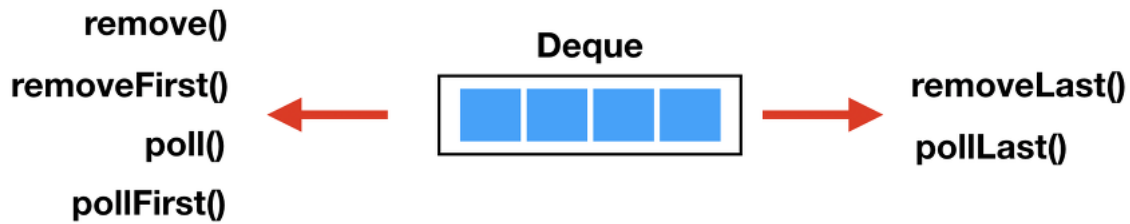
덱의 앞쪽에서 데이터를 뽑아 제거하고 값을 반환. 덱이 비어있다면 null 반환.

### **removeLast()**

덱의 마지막에서 데이터를 뽑아 제거하고 값을 반환. 덱이 비어있다면 Exception 발생.

### **pollLast()**

덱의 마지막에서 데이터를 뽑아 제거하고 값을 반환. 덱이 비어있다면 null 반환.



## Deque 원소 확인

### getFirst()

덱의 앞쪽 데이터 값을 반환. 덱이 비어있으면 Exception 발생.

### peekFirst(), peek()

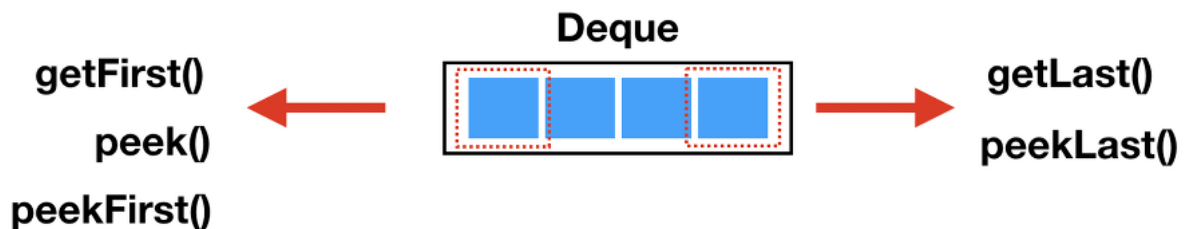
덱의 앞쪽 데이터 값을 반환. 덱이 비어있으면 null이 반환.

### getLast()

덱의 마지막 데이터 값을 반환. 덱이 비어있으면 Exception 발생.

### peekLast()

덱의 마지막 데이터 값을 반환. 덱이 비어있으면 null이 반환.



## 기타 메서드

### removeFirstOccurrence(Object o)

덱의 앞쪽부터 탐색하여 입력한 Object o와 동일한 첫번째 요소를 제거.

### removeLastOccurrence(Object o)

덱의 앞쪽부터 탐색하여 입력한 Object o와 동일한 첫번째 요소를 제거,

### contain(Object o)

덱에 입력한 Object o와 동일한 요소가 포함되어 있는지 확인.

### size()



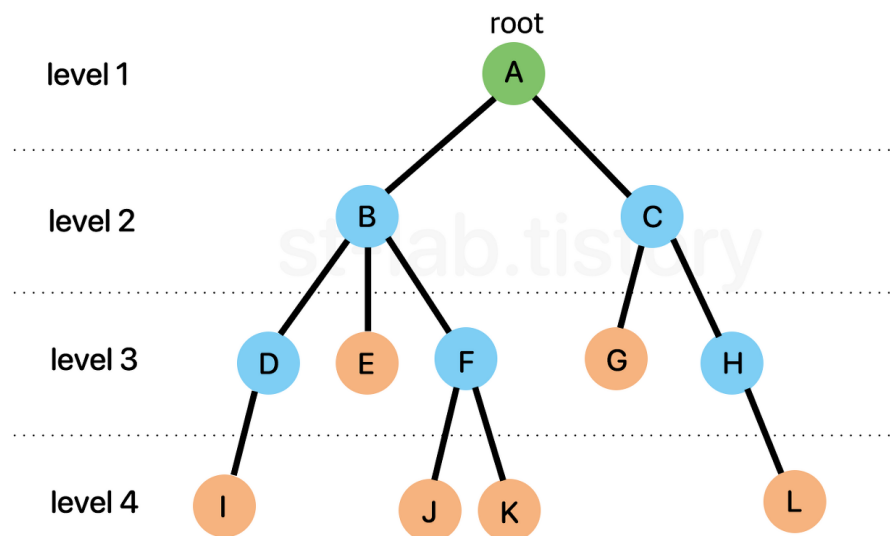
덱의 크기

## 힙(Heap)



**Heap**은 완전 이진 트리의 한 종류로 **우선순위 큐**를 위해 만들어진 자료구조이다.  
여러 개의 값들 중에서 최댓값이나 최솟값을 빠르게 찾아낸다.

### ▼ Tree



- **루트노드(root node):** 가장 처음이 되는 노드. 트리에선 오직 하나 (녹색노드)
- **부모노드(parent node):** 자기 자신과 연결된 노드 중 자신보다 높은 노드  
(ex. D의 부모노드 B)
- **자식노드(child node):** 자기 자신과 연결된 노드 중 자신보다 낮은 모든 노드  
(ex. C의 자식노드 G, H)
- **단말노드(leaf node):** 자식 노드가 없는 노드  
(주황색 노드)
- **내부노드(internal node):** 단말노드가 아닌 노드
- **형제 노드(sibling node):** 부모가 같은 노드 (ex. 형제노드 D, E, F의 부모노드는 B)

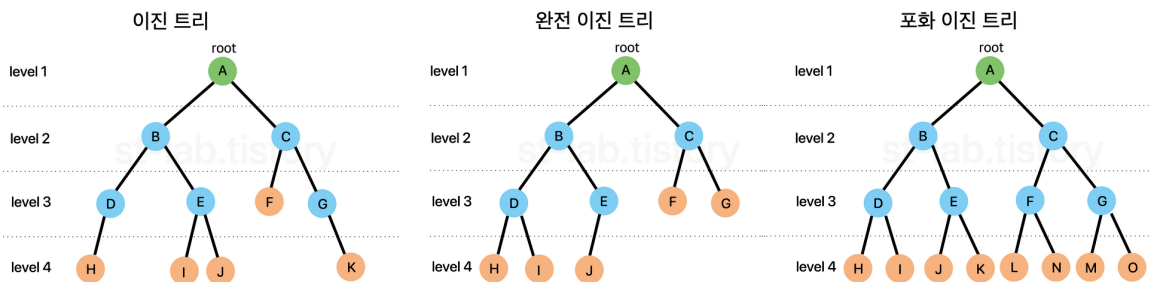
- **깊이(depth):** 특정 노드에 도달하기 위해 거쳐야하는 '간선의 개수'  
(ex. E의 깊이:  $A \rightarrow B \rightarrow E$  이므로 2)
- **레벨(level):** 특정 깊이에 있는 노드들의 집합 (ex. D, E, F, G, H)
- **차수(degree):** 특정 노드가 자식노드와 연결 된 개수 (ex. F의 차수: 2{J, K})

## 이진 트리(Binary Tree)

모든 노드의 최대 차수를 2로 제한한 것. 즉, 자식 노드를 최대 2개까지만 가질 수 있다.

### 완전 이진 트리(Complete binary tree)

- 마지막 레벨을 제외한 모든 노드가 채워져있다.
- 모든 노드들은 왼쪽부터 채워져 있다.



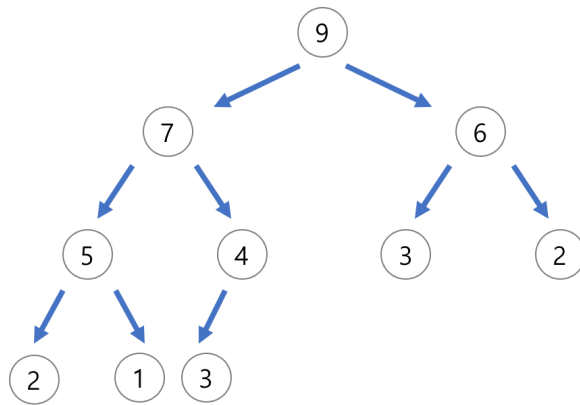
### 힅에서 최댓값과 최솟값을 빠르게 찾아내는 방법.

‘부모 노드는 항상 자식 노드보다 우선순위가 높다.’ 라는 조건을 만족시키면서 완전이진트리 형태로 채운다.

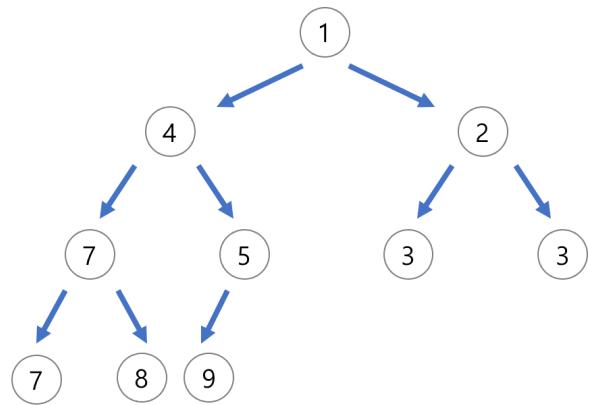
부모 노드와 자식 노드간의 관계만 신경쓰면 되기 때문에 형제 간 우선순위는 고려되지 않는다.

- 힅의 삽입/삭제는 트리 구조이기 때문에  $O(\log n)$
- 이진 탐색 트리에서는 중복된 값을 허용하지 않지만 힅 트리에서는 허용한다.
- 힅은 반정렬 상태(느슨한 정렬 상태)를 유지한다.
  - 큰 값이 상위 레벨에 있고 작은 값이 하위 레벨에 있음.
  - 형제 간의 우선순위 고려되지 않음.

## 힙의 종류



-최대 힙(max heap)-

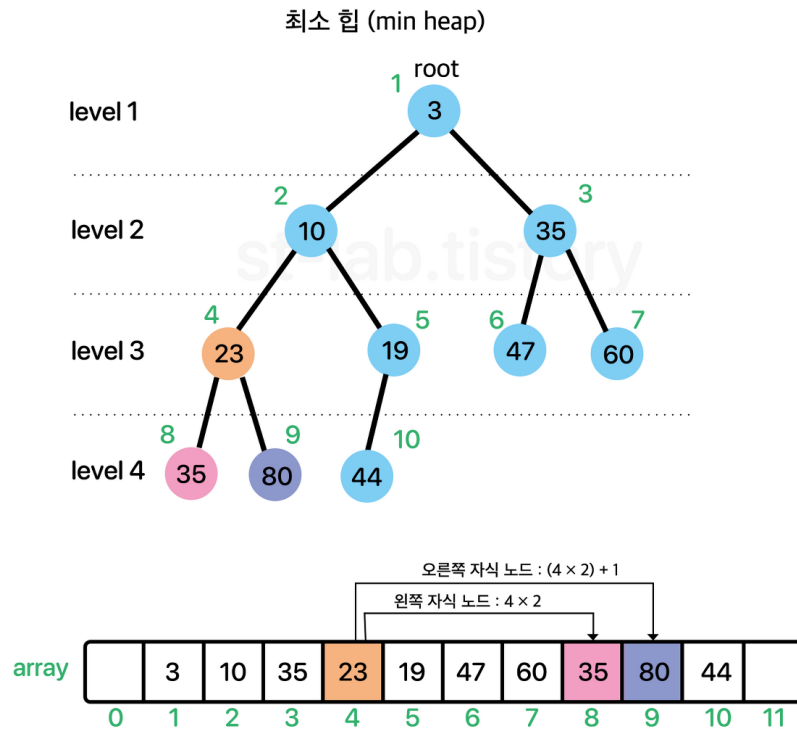


-최소 힙(min heap)-

- 최대 힙(max heap)
  - 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전 이진 트리
  - $\text{key}(\text{부모 노드}) \geq \text{key}(\text{자식 노드})$
- 최소 힙(min heap)
  - 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전 이진 트리
  - $\text{key}(\text{부모 노드}) \leq \text{key}(\text{자식 노드})$

## 힙의 구현

- 힙은 보통 배열을 이용하여 구현한다.
- 인덱스는 1번째부터 시작한다.
  - 인덱스  $i$ 의 부모노드:  $i / 2$
  - 인덱스  $i$ 의 왼쪽 자식노드:  $i * 2$ ,  $(i * 2) + 1$
  - 인덱스  $i$ 의 오른쪽 자식노드:  $(i * 2) + 1$



### 힙의 삽입

- 마지막 위치에 노드를 만들어 삽입한다.
- 부모 노드와 비교해 최대/최소 힙을 만족하도록 자리를 바꾼다.

### 힙의 삭제

- 루트 노드를 삭제한다.
- 마지막 노드를 루트에 옮기고, 마지막 노드를 삭제한다.
- 자식 노드와 비교해 최대/최소 힙을 만족하도록 자리를 바꾼다.

### 자바 배열을 이용한 Heap 구현하기

## 우선순위 큐 사용하기

- 자바에서 우선순위 큐는 Min heap으로 동작하고, Max heap은 Collections가 필요하다.

```
import java.util.Collections;
import java.util.PriorityQueue;

public class PriorityQueueUse {
    public static void main(String[] args) {

        //오름차순 Min heap
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        //내림차순 Max heap
        PriorityQueue<Integer> pqHightest = new PriorityQueue<>(Collections.reverseOrder());

        pq.add(3);
        pq.add(2);
        pq.offer(1);

        System.out.println(pq);

        pqHightest.add(1);
        pqHightest.add(2);
        pqHightest.offer(3);

        System.out.println(pqHightest);

    }
}
```

#### ▼ 결과

[1, 3, 2]

[3, 1, 2]

## 삽입 연산

### add()

큐에 공간이 없어서 삽입 실패 시 exception 발생.

### offer()

큐에 공간이 없어서 삽입 실패 시 null 반환.

## 삭제 연산

### poll()

맨 앞에 위치한 원소 제거 후 반환. 큐가 비어있다면 null 반환.

### remove()

맨 앞에 위치한 원소 제거 후 반환. 큐가 비어있다면 exception 발생.

## removeIf()

- 람다 표현식으로 파라미터 전달 가능
- 필터링 조건

```
//pq: [1, 2, 3, 4, 5]
pq.removeIf(n -> (n % 2 == 0)); //pq: [1, 3, 5]
```

## removeAll()

- 파라미터로 넣은 컬렉션의 겹치는 원소를 제거

```
//pq1: [1, 2, 3, 4, 5]
//pq2: [1, 3, 5]
pq1.removeAll(pq2); //pq1: [2, 4]
pq1.remove(9); //false
```

## clear()

큐의 모든 데이터를 비운다.

## 접근 연산

### peek()

맨 앞에 위치한 원소 반환. 큐가 비어있다면 null 반환.

### iterator()

```
//pq: [1, 3, 2]
Iterator<Integer> iter = pq.iterator();
while(iter.hasNext()) {
    Integer i = iter.next();
    System.out.print(i + " ");
}
```

## 기타

### size()

큐의 사이즈

### toArray()

큐에 저장된 데이터를 배열로 가져온다.

```
for(Object i : pq.toArray()) {  
    System.out.println(i);  
}  
// 1 3 2
```

참고하면 좋을 자료

[JAVA Collection 시간복잡도/특징 정리](#)

[덱 순회방법](#)

[JAVA로 Heap 직접 구현하기](#)

[우선순위 큐 사용하기](#)