ChatGPT

백준 코딩테스트 Java 기본 문법 정리

1. 표준 입력 처리: BufferedReader와 StringTokenizer

백준과 같은 코딩 테스트에서는 `Scanner`보다 `BufferedReader`를 사용한 입력 처리가 더 효율적입니다. `BufferedReader`는 8KB의 버퍼를 사용하여 입력을 한 번에 읽어들이기 때문에, 1KB 버퍼의 `Scanner`에 비해 속도가 훨씬 빠릅니다 [1]. `BufferedReader`로 한 줄 전체를 입력받고, `StringTokenizer`나 `String.split()` 등을 활용해 필요한 단위로 분리합니다. `StringTokenizer`는 입력 문자열을 구분자(예: 공백, 콤마 등)를 기준으로 토큰(token)으로 분리해주며, 효율적인 반복 처리를 위해 많이 사용됩니다 [2].

아래 예시는 `BufferedReader`와 `StringTokenizer`를 사용하여 첫 줄에는 정수 `N`을 읽고, 둘째 줄에서 공백으로 구분된 N개의 숫자를 읽어 배열에 저장하는 코드입니다. 각 단계에 주석으로 설명을 달았습니다:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class ExampleInput {
  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    // 한 줄을 문자열로 읽고 정수로 변환
    int N = Integer.parseInt(br.readLine()); // 첫 번째 줄에서 N (정수) 입력
    int[] arr = new int[N];              // 크기가 N인 정수 배열 선언

    // 다음 줄에서 공백으로 구분된 N개의 숫자들을 읽음
    StringTokenizer st = new StringTokenizer(br.readLine(), " ");
    for (int i = 0; i < N; i++) {
      // StringTokenizer의 nextToken()으로 토큰 단위 문자열 얻고 int로 변환
      arr[i] = Integer.parseInt(st.nextToken());
    }

    br.close(); // 다 사용한 후 BufferedReader 닫기 (자원 해제)

    // (예시 출력) 배열 내용 출력
    for (int num : arr) {
      System.out.print(num + " ");
    }
    // 위 코드는 입력받은 숫자들을 그대로 출력합니다.
  }
}
```

위 코드에서 `br.readLine()`은 한 줄 전체를 문자열로 읽어오며, 필요에 따라 `Integer.parseInt()`로 숫자로 변환합니다. `StringTokenizer st = new StringTokenizer(line, " ");`는 문자열 `line`을 공

백 기준으로 분리하는 예시입니다. `st.hasMoreTokens()`는 남은 토큰이 있는지 확인하고, `st.nextToken()`은 다음 토큰 문자열을 가져옵니다 [3]. (`for`문에서 반복 횟수를 알고 있으므로 바로 `for`로 순회했지만, 알 수 없다면 `while(st.hasMoreTokens())` 형태로 토큰 처리를 할 수도 있습니다.)

주요 메서드 요약:

- `BufferedReader.readLine()` – 입력으로부터 한 줄을 읽어 `String`으로 반환합니다. (더 이상 읽을 줄이 없으면 `null` 반환)
- `Integer.parseInt(String)` – 문자열을 파싱하여 `int` 정수로 변환합니다.
- `StringTokenizer.hasMoreTokens()` – 남아있는 토큰이 있는지 여부를 반환 (`true/false`).
- `StringTokenizer.nextToken()` – 다음 토큰을 `String`으로 반환하고 토큰 목록에서 제거합니다.
- `StringTokenizer(String str, String delim)` – 문자열 `str`을 구분자 `delim` 기준으로 분리하는 `StringTokenizer` 객체 생성. (구분자를 지정하지 않으면 기본적으로 공백으로 분리)

2. 출력 처리 방식: BufferedWriter와 StringBuilder

표준 출력 시에는 `System.out.println`을 반복적으로 호출하면 시간이 오래 걸릴 수 있으므로, `BufferedWriter`나 `StringBuilder`를 이용해 출력 내용을 한꺼번에 모아 처리하는 방법이 자주 쓰입니다. `BufferedWriter`는 출력에 버퍼를 사용하여 효율적으로 처리할 수 있고, `StringBuilder`는 문자열을 미리 다 만들어 놓은 후 한 번에 출력할 때 유용합니다.

BufferedWriter 사용 예시

`BufferedWriter`는 `Writer` 클래스의 일종으로, 버퍼에 데이터를 모아두었다가 한 번에 출력할 수 있습니다. `write()` 메서드로 내용을 쓰고, `newLine()`으로 개행 문자를 삽입할 수 있습니다 [4]. 출력이 모두 끝나면 `flush()`를 호출하여 버퍼의 내용을 실제 출력으로 밀어내고, 다 사용한 후 `close()`로 스트림을 닫아줍니다. 예를 들어 여러 줄의 결과를 출력해야 할 때 다음과 같이 사용할 수 있습니다:

```
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
import java.io.IOException;

public class ExampleOutput {
  public static void main(String[] args) throws IOException {
    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

    bw.write("안녕하세요!");      // 문자열 출력
    bw.newLine();              // 개행(줄 바꿈)
    bw.write("BufferedWriter 예제입니다.");
    bw.newLine();

    bw.flush();  // 버퍼에 모인 출력 내용을 밀어내기
    bw.close();  // 스트림을 닫고 자원 해제
  }
}
```

위 코드에서는 `bw.write(...)`로 출력할 문자열을 버퍼에 쓰고, `bw.newLine()`으로 줄바꿈을 추가했습니다. `flush()`를 호출하여 버퍼에 쌓인 내용을 모두 출력했고, 마지막에 `close()`로 BufferedWriter를 닫았습니다.

(백준과 같은 온라인 채점 환경에서는 `close()` 없이 `flush()`만 호출해도 출력이 완료되지만, 명시적으로 닫아 주는 것이 자원 관리에 좋습니다.)

주요 메서드 (BufferedWriter):

- `write(String)`, `write(char[])` – 문자열이나 문자 배열 등의 내용을 출력 버퍼에 씁니다 [5].
- `newLine()` – 새 줄 문자를 추가하여 줄을 바꿉니다 [5].
- `flush()` – 버퍼에 모인 내용을 실제 출력 스트림으로 밀어냅니다 [5].
- `close()` – 스트림을 닫고 시스템 자원을 반환합니다 (flush를 내장하고 있음).

StringBuilder를 활용한 출력 예시

`StringBuilder`는 가변(mutable)한 문자열 객체로, 문자열을 효율적으로 연결하거나 수정할 때 사용됩니다 [6]. 출력이 많은 경우 `StringBuilder`로 모든 출력을 하나의 `String`으로 만든 뒤 한꺼번에 출력하면 효율적입니다. 예를 들어, 아래 코드는 `StringBuilder`를 사용하여 여러 값을 하나의 문자열로 연결하고 최종적으로 한 번만 출력합니다:

```
public class ExampleOutput2 {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();

    // 출력할 문자열들을 StringBuilder에 추가 (개행 포함)
    sb.append("안녕하세요!").append("\n");       // append로 문자열 추가
    sb.append("StringBuilder 출력 예제입니다.\n"); // \n으로 줄바꿈

    // 필요한 만큼 출력 내용을 누적한 후 한 번에 출력
    System.out.print(sb.toString()); // String으로 변환하여 출력
  }
}
```

위 코드에서 `sb.append()`를 사용하여 문자열을 차례로 이어붙였습니다. `append` 메서드는 연쇄적으로 호출 가능하여, 위처럼 `sb.append(...).append(...);` 형태로 연달아 사용할 수도 있습니다. 최종적으로 `sb.toString()`을 호출하여 완성된 문자열을 얻고 `System.out.print()`로 출력합니다. 이 방식은 출력량이 많을 때 `System.out.println`을 매번 호출하는 것보다 성능 면에서 유리합니다. (참고: `StringBuilder`는 쓰레드-세이프는 아니지만 단일 쓰레드 환경인 코딩 테스트에서는 문제가 되지 않습니다. 멀티쓰레드 환경이라면 `StringBuffer`를 사용할 수 있습니다.)

주요 메서드 (StringBuilder):

- `append(값)` – 문자열 끝에 값을 추가합니다. 다양한 오버로드가 있어 문자열, 문자, 숫자 등 모든 타입을 추가할 수 있습니다.
- `toString()` – 누적된 StringBuilder 내용을 `String`으로 변환합니다.
- `setLength(0)` – `StringBuilder` 내용을 초기화(길이를 0으로 재설정)하여 재사용할 때 활용합니다.
- (이 외에 `insert(int idx, 값)`, `delete(int start, int end)` 등으로 중간에 삽입/삭제, `reverse()`로 문자열 뒤집기 등의 메서드도 제공됩니다.)

3. 배열과 ArrayList 사용법

Java에서는 배열(`Array`)과 리스트(List)의 개념이 구분됩니다. **배열**은 한 번 크기를 정하면 변경할 수 없는 **고정 크기 자료구조**이고, 동일한 타입의 요소들만 저장할 수 있습니다 [7]. 반면 **ArrayList** 등 **리스트**는 크기가 동적으로 변하며, 요소 추가/삭제에 따라 자동으로 크기가 조절되는 **동적 배열** 형태입니다 [8]. 배열은 인덱스를 통해 직접 요소에 접근할 수 있고, 리스트는 컬렉션 프레임워크의 `List` 인터페이스 구현체로 다양한 편의 메서드를 제공합니다.

배열(Array) 사용 예시

배열 선언 시 타입과 크기를 지정해야 합니다. 선언 방법으로는 `타입[] 변수명 = new 타입[크기];` 또는 리터럴 값으로 바로 초기화하는 `타입[] 변수명 = {값1, 값2, ...};` 형태가 있습니다. 배열의 길이는 `배열변수.length` 속성으로 알 수 있습니다. 또한 `java.util.Arrays` 클래스의 정적 메서드들을 사용하여 배열을 쉽게 조작할 수 있습니다 (정렬, 검색 등).

```
import java.util.Arrays;
import java.util.Comparator;

public class ExampleArray {
  public static void main(String[] args) {
    // 배열 선언과 초기화
    int[] numbers = {5, 2, 8, 1, 3};     // 크기 5의 int 배열을 선언하고 리터럴로 초기화
    System.out.println("길이: " + numbers.length); // 배열 길이 출력 (5)
    System.out.println("첫 번째 원소: " + numbers[0]); // 인덱스로 접근 (첫 원소)

    // 배열 정렬 (오름차순)
    Arrays.sort(numbers);
    System.out.println(Arrays.toString(numbers));  // 정렬된 배열 출력 => [1, 2, 3, 5, 8]

    // 배열 정렬 (내림차순) - 방법 1: Comparator 사용
    Integer[] numbersObj = {5, 2, 8, 1, 3};
    Arrays.sort(numbersObj, Comparator.reverseOrder());
    System.out.println(Arrays.toString(numbersObj)); // 내림차순 정렬 결과 => [8, 5, 3, 2, 1]

    // 배열 정렬 (내림차순) - 방법 2: 오름차순 후 직접 뒤집기
    Arrays.sort(numbers);          // numbers 배열 오름차순 정렬 -> [1, 2, 3, 5, 8]
    // 배열 원소 순서 뒤집기 (swap 이용)
    for (int i = 0; i < numbers.length / 2; i++) {
      int temp = numbers[i];
      numbers[i] = numbers[numbers.length - 1 - i];
      numbers[numbers.length - 1 - i] = temp;
    }
    System.out.println(Arrays.toString(numbers));  // 내림차순 정렬 결과 => [8, 5, 3, 2, 1]

    // 배열 검색 (이진 탐색) - Arrays.binarySearch 사용 (배열은 정렬되어 있어야 함)
    Arrays.sort(numbers);                  // 이진 탐색 전 오름차순 정렬 [1, 2, 3, 5, 8]
    int idx = Arrays.binarySearch(numbers, 5);     // 값 5의 인덱스를 이진 탐색으로 찾음
    System.out.println("5의 인덱스: " + idx);       // 출력: 5의 인덱스: 3  (numbers[3] == 5)
  }
}
```

위 예시에서 `Arrays.sort(numbers)`는 **오름차순**으로 배열을 정렬합니다. 기본 타입 배열(`int[]`)은 오름차순 정렬만 직접 지원하므로, **내림차순**으로 정렬하려면 두 가지 방식을 사용할 수 있습니다: (1) 위 코드처럼 `Integer` 객체 배열로 변환하여 `Comparator.reverseOrder()`를 사용하는 방법, (2) 오름차순 정렬 후 직접 요소들을 반대 순서로 뒤집는 방법 등입니다. 또한 `Arrays.binarySearch(array, value)`는 **이진 탐색**으로 배열에서 특정 값의 인덱스를 찾아주는데, 이 메서드를 사용하려면 배열이 **정렬된 상태**여야 함을 주의해야 합니다.

주요 기능 및 속성 (배열):

- **배열 선언** – 예: `int[] arr = new int[10];` (길이가 10인 int형 배열 생성), `String[] names = {"A", "B"};` (리터럴로 초기화)
- `length` **속성** – 배열의 길이를 나타내는 int 값. (`arr.length`)
- **인덱스를 통한 접근** – `arr[0]`, `arr[1] = ...` 등을 통해 요소 접근/대입.
- **정렬** – `Arrays.sort(arr)` (오름차순 정렬). 객체 배열의 경우 `Arrays.sort(arr, Comparator)`로 정렬 기준 지정 가능.
- **검색** – `Arrays.binarySearch(arr, key)` (이진 탐색으로 인덱스 찾기; 없으면 음수 반환).
- **기타** – `Arrays.toString(arr)` (배열 내용을 문자열로 리턴), `Arrays.fill(arr, value)` (전체를 특정 값으로 채우기) 등.

ArrayList 사용 예시

`ArrayList`는 Java의 동적 배열 구현체로, `List<E>` 인터페이스를 구현합니다. 크기를 동적으로 늘리거나 줄일 수 있고, 유용한 메서드들을 제공합니다. 선언 방법은 `ArrayList<타입> 리스트명 = new ArrayList<>();` 형태이며, `<타입>` 부분에 원소의 타입을 제네릭으로 명시합니다. (기본형은 사용할 수 없고, `Integer`, `Double` 등의 래퍼(wrapper) 클래스나 다른 객체 타입을 사용해야 합니다.)

```
import java.util.ArrayList;
import java.util.Collections;

public class ExampleArrayList {
  public static void main(String[] args) {
    // ArrayList 선언
    ArrayList<String> fruits = new ArrayList<>(); // 문자열을 담는 ArrayList 생성

    // 요소 추가
    fruits.add("Apple");        // 인덱스 0에 "Apple"
    fruits.add("Banana");       // 인덱스 1에 "Banana"
    fruits.add("Cherry");       // 인덱스 2에 "Cherry"
    fruits.add(1, "Blueberry");   // 인덱스 1 위치에 "Blueberry" 삽입 (나머지 뒤로 이동)

    System.out.println(fruits);   // 출력: [Apple, Blueberry, Banana, Cherry]

    // 요소 접근 및 변경
    String second = fruits.get(1); // 인덱스 1의 요소 얻기 ("Blueberry")
    fruits.set(2, "Orange");      // 인덱스 2의 요소를 "Orange"로 변경 (Banana -> Orange)
    System.out.println(fruits);   // 출력: [Apple, Blueberry, Orange, Cherry]

    // 요소 제거
    fruits.remove(3);           // 인덱스 3의 요소 제거 ("Cherry" 제거)
    fruits.remove("Apple");       // 값으로 제거: "Apple" 항목 제거
```

```
        System.out.println(fruits);    // 출력: [Blueberry, Orange]

        // 검색 기능
        boolean hasOrange = fruits.contains("Orange"); // "Orange" 포함 여부 확인 (true)
        int indexOfOrange = fruits.indexOf("Orange");   // "Orange"의 인덱스 (1)
        System.out.println("Orange 있음?: " + hasOrange + ", 위치: " + indexOfOrange);

        // 정렬
        fruits.add("Grape");
        fruits.add("Apple");
        Collections.sort(fruits);                // 오름차순 정렬 (알파벳 순)
        System.out.println("정렬 후: " + fruits);       // 출력: [Apple, Blueberry, Grape, Orange]
        Collections.sort(fruits, Collections.reverseOrder()); // 내림차순 정렬
        System.out.println("내림차순 정렬 후: " + fruits);     // 출력: [Orange, Grape, Blueberry, Apple]

        // 리스트 -> 배열 변환
        String[] arr = fruits.toArray(new String[0]);  // 리스트를 배열로 변환
        System.out.println("배열로 변환: " + java.util.Arrays.toString(arr));
    }
}
```

위 예시에서 `add` 메서드로 요소를 추가하고, `get(index)`로 값에 접근하며, `set(index, value)`로 값을 수정했습니다. `remove`는 인덱스로 제거하거나 객체 값으로 제거할 수 있습니다. `contains(Object)`는 리스트에 해당 값이 포함되어 있는지 여부를 반환하고, `indexOf(Object)`는 값이 처음으로 등장하는 인덱스를 돌려줍니다 (없으면 -1). 또한 `Collections.sort(list)`를 이용해 **오름차순 정렬**, `Collections.sort(list, Collections.reverseOrder())`로 **내림차순 정렬**을 수행했습니다. 마지막으로 `toArray(T[] a)` 메서드를 통해 리스트를 배열로 변환할 수도 있습니다.

주요 메서드 (ArrayList 및 List 인터페이스):

- `add(E e)` – 요소 `e`를 리스트 끝에 추가. (`add(int index, E e)`: 지정한 인덱스에 삽입도 가능)
- `get(int index)` – 해당 인덱스의 요소를 반환.
- `set(int index, E element)` – 해당 인덱스의 요소를 주어진 값으로 수정.
- `remove(int index)`, `remove(Object o)` – 인덱스의 요소 또는 주어진 값을 리스트에서 제거.
- `contains(Object o)` – 리스트에 해당 값이 포함되어 있으면 `true` [9].
- `indexOf(Object o)` – 해당 값이 저장된 **첫 인덱스**를 반환 (없으면 -1).
- `size()` – 리스트에 포함된 요소의 개수(길이)를 반환.
- `isEmpty()` – 리스트가 비어있는지 여부를 반환.
- `clear()` – 리스트의 모든 요소를 제거 (빈 리스트로 만듦).
- **정렬** – `Collections.sort(List)` (오름차순), `Collections.sort(List, Comparator)` (지정한 정렬기준으로 정렬).
  (참고: Java 8부터는 `list.sort(Comparator)` 메서드도 직접 제공합니다.)

4. Stack, Queue, Deque의 선언과 주요 메서드

스택, 큐, 덱은 **자료구조**에서 자주 등장하는 개념으로, Java에서는 각각 `java.util` 패키지의 클래스로 제공되거나 인터페이스로 지원됩니다. 코딩 테스트에서는 **DFS/BFS와 같은 탐색 알고리즘**이나 **자료구조 활용 문제**에서 이들을 빈번하게 사용합니다. 아래에서 각 구조의 특성과 사용법을 알아보고, 자주 쓰이는 메서드를 정리합니다.

스택(Stack)

스택은 **후입선출(LIFO, Last-In First-Out)** 구조의 자료구조입니다 [10]. 즉, 마지막에 추가된 요소가 가장 먼저 제거됩니다. Java에서는 `java.util.Stack` 클래스를 통해 스택을 사용할 수 있습니다. (스택은 내부적으로 `Vector`를 상속하여 구현되었는데, 스레드 동기화 등의 이유로 성능이 다소 떨어질 수 있으므로, 단순한 후입선출 용도로는 `ArrayDeque`를 스택처럼 사용하는 것도 고려됩니다. 하지만 여기서는 사용이 간편한 `Stack` 클래스를 설명합니다.)

스택은 **DFS(깊이 우선 탐색)** 구현이나, **괄호 문자열 체크** (예를 들어 백준의 괄호 문제), **뒤집기 기능** 등에 자주 활용됩니다. 주요 메서드로 `push`(요소 추가), `pop`(요소 제거 및 반환), `peek`(제거 없이 맨 위 요소 조회) 등이 있습니다.

```
import java.util.Stack;

public class ExampleStack {
  public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();  // 정수형 스택 생성

    // 데이터 넣기 (push)
    stack.push(10);
    stack.push(20);
    stack.push(30);
    System.out.println(stack);          // 출력: [10, 20, 30]  (스택의 toString, 최상단 30)

    // 데이터 꺼내기 (pop)
    int top = stack.pop();
    System.out.println("pop 결과: " + top); // 출력: pop 결과: 30  (마지막에 넣은 30을 꺼냄)
    System.out.println(stack);          // 출력: [10, 20]  (30이 제거된 상태)

    // 최상단 요소 조회 (peek)
    System.out.println("현재 top: " + stack.peek()); // 출력: 현재 top: 20  (맨 위 요소 확인, 제거되지 않음)

    // 기타 메서드들
    System.out.println("size: " + stack.size());    // 스택 크기 (요소 개수) -> 2
    System.out.println("isEmpty?: " + stack.isEmpty()); // 스택이 비었는지 -> false

    stack.push(40);
    System.out.println(stack.search(40));  // 값 40의 위치 검색 (1부터 시작, top 기준) -> 1
    System.out.println(stack.search(10));  // 값 10의 위치 검색 -> 3 (현재 스택 [10,20,40], 10은 바닥에서 3번째)
  }
}
```

위 코드에서 `stack.push(값)`으로 값을 넣고, `stack.pop()`으로 마지막에 넣은 값을 꺼냈습니다. `Stack`의 `toString()`은 스택 내용을 `[..., ...]` 형태로 보여주는데, **맨 끝의 값이 스택의 top**임을 알 수 있습니다. `pop()`은 제거된 값을 반환하므로, 필요한 경우 변수에 받아서 사용할 수 있습니다. `peek()`은 제거 없이 top에 무엇이 있는지만 확인합니다. `isEmpty()`는 스택이 비어있는지 확인하며, `size()`는 현재 들어있는 요소 개수를 반환합니다. `search(Object)` 메서드는 스택 내에서 해당 객체의 1-based 위치를 반환하며, top의 위치가 1입니다 (찾지 못하면 -1 반환).

주요 메서드 (Stack):

- `push(E item)` – 스택의 top에 요소를 추가 [11].
- `pop()` – 스택의 top 요소를 제거하면서 그 값을 반환 [12].
- `peek()` – 스택의 top 요소를 제거하지 않고 반환 [13].
- `isEmpty()` – 스택이 비어있으면 `true` 반환 [14].
- `size()` – 스택에 있는 요소의 개수를 반환.
- `search(Object o)` – 해당 요소가 스택에서 **위로부터 몇 번째**에 있는지 반환 (1부터 시작, 없으면 -1).

큐(Queue)

큐는 **선입선출(FIFO, First-In First-Out)** 구조를 가지며, 먼저 들어온 요소가 먼저 나가는 형태의 자료구조입니다 [15]. Java에서 큐를 구현하는 일반적인 방법은 `java.util.Queue` 인터페이스를 사용하고, 이를 `LinkedList`나 `ArrayDeque` 등으로 구현하는 것입니다. (`Queue`는 인터페이스라 직접 인스턴스화할 수 없고, `new LinkedList<>()` 또는 `new ArrayDeque<>()` 등으로 생성합니다.) 기본 동작은 `offer` / `add`로 요소를 넣고, `poll` / `remove`로 요소를 꺼내는 것입니다.

큐는 **BFS(너비 우선 탐색)** 구현에서 필수적으로 사용되고, 그 외에 **데이터를 순서대로 처리하는 문제**에서 활용됩니다.

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

public class  ExampleQueue {
  public static void main(String[] args) {
    Queue<String> queue = new LinkedList<>();   // 문자열을 담는 큐 생성 (LinkedList 구현 사용)

    // 데이터 넣기 (offer/add)
    queue.offer("A");
    queue.offer("B");
    queue.add("C");  // add도 offer와 동일하게 동작 (삽입 실패 시 예외발생 차이만 있음)
    System.out.println(queue);     // 출력: [A, B, C] (맨 앞이 A)

    // 데이터 꺼내기 (poll)
    String first = queue.poll();
    System.out.println("poll 결과: " + first);  // 출력: poll 결과: A  (처음 들어간 A 제거 및 반환)
    System.out.println(queue);              // 출력: [B, C]  (A가 제거된 상태)

    // 앞에 있는 요소 조회 (peek)
    System.out.println("현재 front: " + queue.peek()); // 출력: 현재 front: B  (맨 앞 요소 확인, 제거 안 함)

    // 기타 메서드들
    System.out.println("size: " + queue.size());    // 큐 길이 -> 2
    System.out.println("isEmpty?: " + queue.isEmpty()); // 큐가 비었는지 -> false
    System.out.println("contains(\"C\")?: " + queue.contains("C")); // 특정 값 포함 여부 -> true

    queue.clear();  // 모든 요소 제거 (초기화)
    System.out.println("clear 후 isEmpty?: " + queue.isEmpty());  // 출력: true
  }
}
```

위 코드에서 `offer`와 `add`는 큐의 끝에 요소를 추가하는 기능을 합니다. 두 메서드는 기능적으로 유사하지만, **큐가 가득 찬 상황에서** `offer`는 `false`를 반환하고 `add`는 예외를 발생시킨다는 차이가 있습니다. 일반적으로 코딩 테스트 환경에서는 용량 제한이 없는 `LinkedList`나 `ArrayDeque`를 사용하므로 둘의 차이는 크지 않습니다. `poll()`은 큐의 **맨 앞(front)** 요소를 제거하면서 반환하고, 큐가 비어있을 때는 `null`을 반환합니다 (`remove()`는 비어있을 때 예외 발생). `peek()`은 제거 없이 맨 앞 요소를 조회하며, 비어있으면 `null`을 반환합니다 (`element()`는 비어있을 때 예외).

주요 메서드 (Queue 인터페이스):

- `offer(E e)` – 큐의 맨 뒤에 요소를 추가 (용량 초과 등으로 삽입 실패하면 `false` 반환) [16].
- `add(E e)` – 큐에 요소 추가 (용량 초과 등 실패 시 예외 발생).
- `poll()` – 큐의 맨 앞 요소를 제거하며 반환; 큐가 비어있으면 `null` 반환 [17].
- `remove()` – 맨 앞 요소 제거 및 반환; 비어있으면 예외 발생.
- `peek()` – 맨 앞 요소를 제거하지 않고 반환; 큐가 비어있으면 `null` 반환 [18].
- `element()` – 맨 앞 요소 조회; 비어있으면 예외 발생.
- `isEmpty()`, `size()`, `clear()`, `contains(Object)` – 리스트와 유사하게, 큐가 비었는지 여부, 요소 개수, 전체 제거, 특정 원소 포함 여부 등을 제공.

덱(Deque, Double-Ended Queue)

덱(Deque)은 **양쪽 끝에서 삽입과 삭제가 모두 가능한** 자료구조로, 스택과 큐의 기능을 모두 가지고 있습니다 [19]. `Deque`는 인터페이스이며, 구현체로는 주로 `LinkedList`나 `ArrayDeque`을 사용합니다. 덱을 이용하면 **앞쪽**으로 넣고 빼거나 **뒤쪽**으로 넣고 빼는 연산을 모두 효율적으로 할 수 있습니다. 문제 유형에 따라 덱은 **양쪽 끝에서 요소를 뽑아 처리해야 하는 시뮬레이션**이나 **슬라이딩 윈도우**, **회전하는 큐 문제** 등에 자주 등장합니다 (예: 백준의 회전하는 큐 문제에서 덱 활용).

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

public class ExampleDeque {
  public static void main(String[] args) {
    Deque<Integer> deque = new ArrayDeque<>();  // 정수를 담는 덱 생성 (ArrayDeque 구현 사용)

    // 앞쪽으로 넣기 (addFirst) / 뒤쪽으로 넣기 (addLast)
    deque.addFirst(1);  // 덱 앞에 1 추가
    deque.addFirst(2);  // 덱 앞에 2 추가 -> 현재 덱: [2, 1]
    deque.addLast(3);   // 덱 뒤에 3 추가 -> [2, 1, 3]
    deque.offerLast(4); // 덱 뒤에 4 추가 (offerLast는 addLast와 동일) -> [2, 1, 3, 4]
    System.out.println(deque);  // 출력: [2, 1, 3, 4]

    // 앞쪽에서 꺼내기 (pollFirst) / 뒤쪽에서 꺼내기 (pollLast)
    int front = deque.pollFirst();  // 앞에서 하나 꺼내 제거
    int back  = deque.pollLast();   // 뒤에서 하나 꺼내 제거
    System.out.println("front = " + front + ", back = " + back); // 출력: front = 2, back = 4
    System.out.println(deque);      // 출력: [1, 3]  (앞에서 2, 뒤에서 4가 제거됨)

    // 앞쪽/뒤쪽 요소 조회 (peekFirst/peekLast)
    System.out.println("맨 앞: " + deque.peekFirst()); // 출력: 맨 앞: 1
    System.out.println("맨 뒤: " + deque.peekLast());  // 출력: 맨 뒤: 3
```

```
        // 스택처럼 사용하기 (push, pop)
        deque.push(10);   // = addFirst(10)
        System.out.println(deque);     // 출력: [10, 1, 3]
        int top = deque.pop();         // = removeFirst()
        System.out.println("pop 결과: " + top);  // 출력: pop 결과: 10
    }
}
```

위 코드에서는 `ArrayDeque`를 사용하여 덱을 생성했습니다. `deque.addFirst(x)`는 덱의 앞쪽에 값을 넣고, `deque.addLast(x)` (또는 `offerLast`)는 덱의 뒤쪽에 값을 넣습니다. `pollFirst()`는 앞쪽에서 값을 꺼내 제거하고, `pollLast()`는 뒤쪽에서 값을 꺼내 제거합니다. `peekFirst()`, `peekLast()`는 양 끝의 값을 제거 없이 확인만 합니다. 또한 `Deque` 인터페이스는 스택의 메서드인 `push`와 `pop`도 제공하는데, 내부적으로 `push(e)`는 `addFirst(e)`와 동일하고, `pop()`은 `removeFirst()`와 동일하게 동작하여 덱의 앞쪽을 스택 top처럼 사용하게 합니다. 위 예시에서 `deque.push(10)`으로 10을 맨 앞에 넣고, `deque.pop()`으로 맨 앞(최근에 넣은 10)을 뽑아냈습니다.

주요 메서드 (Deque 인터페이스):

- **앞쪽 삽입**: `addFirst(E e)`, `offerFirst(E e)` – 덱의 앞쪽에 요소 추가.
- **뒤쪽 삽입**: `addLast(E e)`, `offerLast(E e)` – 덱의 뒤쪽에 요소 추가.
- **앞쪽 제거**: `pollFirst()`, `removeFirst()` – 덱 앞쪽 요소 제거 및 반환 (`pollFirst`는 비어있으면 `null`, `removeFirst`는 예외).
- **뒤쪽 제거**: `pollLast()`, `removeLast()` – 덱 뒤쪽 요소 제거 및 반환.
- **앞쪽 조회**: `peekFirst()`, `getFirst()` – 덱 앞쪽 요소 반환 (제거 안 함; `getFirst`는 비어있으면 예외).
- **뒤쪽 조회**: `peekLast()`, `getLast()` – 덱 뒤쪽 요소 반환.
- **스택 호환 메서드**: `push(E e)` (앞쪽에 삽입) [20], `pop()` (앞쪽 요소 제거 및 반환) 등.
- **기타**: `isEmpty()`, `size()`, `contains(Object)`, `clear()` 등 컬렉션 공통 메서드 사용 가능.

5. HashMap, TreeMap, HashSet, TreeSet의 사용법과 주요 메서드

Java의 컬렉션 프레임워크에는 **Map**과 **Set** 인터페이스를 구현한 다양한 클래스들이 있습니다. 여기서는 코딩 테스트에서 많이 활용되는 `HashMap`, `TreeMap` (Map 계열)과 `HashSet`, `TreeSet` (Set 계열)의 기본 사용법과 주요 메서드를 살펴보겠습니다.

Map 인터페이스 – HashMap과 TreeMap

Map은 키(key)-값(value) 쌍으로 데이터를 저장하는 구조입니다. 한 키에는 한 값만 대응되며, 키는 중복을 허용하지 않습니다. Java에서 대표적인 구현체로 **HashMap**과 **TreeMap**이 있습니다.

- **HashMap**은 내부적으로 **해시테이블**을 사용하여 데이터를 저장하므로, 일반적으로 대부분의 연산(삽입, 검색, 삭제)이 평균적으로 **O(1)**의 시간복잡도를 가집니다 [21]. 키 객체의 `hashCode()`와 `equals()`가 중요하며, 순서를 유지하지 않습니다 (저장한 순서와 출력 순서가 같지 않을 수 있음). 키로 사용할 객체는 `hashCode`/`equals`가 잘 정의되어야 하며, `String`이나 `Integer`처럼 기본적으로 잘 정의된 것들을 많이 사용합니다.

- **TreeMap**은 **이진 검색 트리**(레드-블랙 트리) 기반으로 구현된 맵으로, 키를 자동으로 **정렬된 순서**(기본적으로 오름차순)로 유지합니다 [22] [23]. 모든 삽입, 검색, 삭제 연산이 평균 **O(log N)**의 시간복잡도를 가지며, 정렬된

키 순서가 필요할 때 사용합니다. 예를 들어 키의 최소값, 최대값, 범위 검색 등이 필요한 경우 유용하며, `firstKey()`, `lastKey()` 등의 메서드를 제공합니다.

두 Map 모두 `Map<K,V>` 인터페이스를 구현하므로 기본적인 사용법은 유사합니다. 아래는 `HashMap`과 `TreeMap`의 간단한 사용 예시입니다:

```
import java.util.HashMap;
import java.util.TreeMap;
import java.util.Map;

public class  ExampleMap {
  public static void main(String[] args) {
    // HashMap 예시
    Map<String, Integer> scores = new HashMap<>();  // 학생 이름 -> 점수 매핑
    scores.put("Alice", 90);
    scores.put("Bob", 85);
    scores.put("Charlie", 95);
    scores.put("Bob", 88);  // 키 중복시 마지막 값으로 덮어씀 (Bob의 점수를 88로 수정)

    System.out.println("Alice 점수: " + scores.get("Alice"));   // 출력: Alice 점수: 90
    System.out.println("Bob 점수: " + scores.get("Bob"));      // 출력: Bob 점수: 88
    System.out.println("Dave 점수: " + scores.get("Dave"));    // 없는 키 -> null 출력
    System.out.println("Dave 점수 (기본값): " + scores.getOrDefault("Dave",
0)); // 키 없으면 기본값 0 -> 출력: 0

    scores.remove("Alice");                  // Alice 항목 제거
    System.out.println("Alice 포함?: " + scores.containsKey("Alice")); // 출력: Alice 포함?: false

    System.out.println("키 집합: " + scores.keySet());   // 모든 키 출력 (순서는 일정하지 않음)
    System.out.println("값 집합: " + scores.values());   // 모든 값 출력

    System.out.println("----------");

    // TreeMap 예시
    TreeMap<Integer, String> rankMap = new TreeMap<>();  // 등수 -> 이름 매핑 (키로 정렬)
    rankMap.put(3, "Gold");
    rankMap.put(1, "Bronze");
    rankMap.put(2, "Silver");

    System.out.println("등수 맵: " + rankMap);       // 출력: {1=Bronze, 2=Silver, 3=Gold} (키 순으로 정렬
됨)
    System.out.println("1등: " + rankMap.get(1));      // 출력: 1등: Bronze
    System.out.println("최소 키: " + rankMap.firstKey()); // 출력: 최소 키: 1
    System.out.println("최대 키: " + rankMap.lastKey());  // 출력: 최대 키: 3
    System.out.println("최소 항목: " + rankMap.firstEntry()); // 출력: 최소 항목: 1=Bronze
    System.out.println("2보다 큰 첫 키: " + rankMap.higherKey(2)); // 출력: 2보다 큰 첫 키: 3
    System.out.println("2보다 작은 첫 키: " + rankMap.lowerKey(2)); // 출력: 2보다 작은 첫 키: 1
  }
}
```

먼저 `HashMap` 예시에서, `put(key, value)`로 키-값을 저장하고, `get(key)`으로 값을 가져왔습니다. 없는 키에 대해 `get`을 호출하면 `null`을 반환하며, 이때 `getOrDefault(key, 기본값)`을 쓰면 키가 없을 경우 지정한 기본값을 반환해 편리합니다. `remove(key)`는 해당 키의 엔트리를 제거합니다. `containsKey(key)`로 특정 키 존재 여부를, `containsValue(value)`로 특정 값의 존재 여부를 확인할 수 있습니다. `keySet()`은 모든 키를 `Set`으로 반환하고, `values()`는 모든 값을 컬렉션으로 반환합니다. HashMap은 순서를 보장하지 않으므로 출력 결과의 순서는 예측과 다를 수 있습니다.

`TreeMap` 예시에서는 키로 정수 등수를 사용해 값을 넣었습니다. 출력에서 보듯이 `rankMap`은 키를 자동으로 정렬하여 `{1=..., 2=..., 3=...}` 형태로 출력됩니다. `firstKey()` / `lastKey()`는 가장 작은 키와 가장 큰 키를 반환하고, `firstEntry()` / `lastEntry()`는 해당 키-값 쌍(`Map.Entry`)을 반환합니다. `higherKey(x)`는 주어진 값 `x`보다 큰 **최소 키**를, `lowerKey(x)`는 `x`보다 작은 **최대 키**를 찾아줍니다. (유사하게 `ceilingKey(x)`는 `x` 이상 가장 작은 키, `floorKey(x)`는 `x` 이하 가장 큰 키를 반환합니다.) 이러한 정렬 기반 메서드들은 TreeMap에서만 제공되며, 정렬된 자료가 필요 없으면 일반적으로 HashMap을 사용하는 편이 성능이 더 좋습니다 [22].

주요 메서드 (Map 공통, HashMap/TreeMap):

- `put(K key, V value)` – 키-값 쌍 저장. 이미 존재하는 키이면 값을 덮어씀 [24].
- `get(Object key)` – 주어진 키에 대응되는 값 반환 (없으면 `null`).
- `getOrDefault(Object key, V defaultValue)` – 키에 대한 값이 없으면 기본값 반환 [25].
- `remove(Object key)` – 해당 키의 엔트리를 제거 [26].
- `containsKey(Object key)`, `containsValue(Object value)` – 특정 키 또는 값의 존재 여부 확인.
- `keySet()`, `values()`, `entrySet()` – 키, 값, 또는 (키,값) 엔트리들의 뷰(View) 컬렉션 반환 (반복 또는 출력용).
- **(TreeMap 전용)** `firstKey()`, `lastKey()` – 최소/최대 키 반환 [27].
- **(TreeMap 전용)** `firstEntry()`, `lastEntry()` – 최소/최대 키에 대한 키-값 엔트리(`Map.Entry`) 반환 [28].
- **(TreeMap 전용)** `higherKey(K key)`, `lowerKey(K key)` – 주어진 키보다 큰 **최소 키** / 작은 **최대 키** 반환 [29].
- **(TreeMap 전용)** `ceilingKey(K key)`, `floorKey(K key)` – 주어진 키 이상/이하의 첫 키 반환.

(TreeMap은 이외에도 NavigableMap 인터페이스로서 subMap 등 범위 뷰를 제공하지만, 코딩 테스트 기본 문법 범위에서는 위 메서드들만 알아도 충분합니다.)

Set 인터페이스 – HashSet과 TreeSet

Set은 **중복을 허용하지 않는** 컬렉션으로, 순서가 보장되지 않는 집합 구조입니다. 대표 구현체로 **HashSet**과 **TreeSet**이 있습니다. 사용법은 `Set<E> set = new HashSet<>();` 혹은 `new TreeSet<>();` 형태로 선언합니다.

- **HashSet**은 내부적으로 HashMap과 유사하게 **해싱**을 통해 구현되어 있습니다. 따라서 **평균 O(1)** 시간에 요소를 추가, 제거, 탐색할 수 있고 [21], 데이터의 **중복을 자동으로 걸러주기 때문에** 중복 제거 용도로 자주 쓰입니다 [30]. 순서를 유지하지 않으며, 넣는 순서와 상관없이 저장됩니다.

- **TreeSet**은 TreeMap과 마찬가지로 **이진 검색 트리(레드-블랙 트리)** 기반으로 구현되어, 요소를 **정렬된 순서**로 유지합니다 [31]. 따라서 요소 추가/삭제/탐색에 **O(log N)**의 시간이 걸리지만, 자동 정렬 및 **범위 검색** 등의 기능 (`subSet`, `first()`, `last()`, `higher()`, `lower()` 등)을 제공합니다. 기본 정렬 기준은 자연 순서(예: 숫자는 오름차순, 문자열은 사전순)이며, 필요에 따라 생성자에 `Comparator`를 전달해 커스텀 정렬을 할 수도 있습니다.

아래는 `HashSet`과 `TreeSet`의 사용 예시입니다:

```
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Set;
import java.util.Arrays;

public class  ExampleSet {
  public static void main(String[] args) {
    // HashSet 예시
    Set<String> cities = new HashSet<>();
    cities.add("Seoul");
    cities.add("New York");
    cities.add("Seoul");  // 중복된 값 추가 시도
    cities.add("London");
    System.out.println("도시 집합: " + cities);     // 출력 예: [London, Seoul, New York] (순서 관계 없음)
    System.out.println("Size: " + cities.size());   // 출력: Size: 3  ("Seoul"은 한 번만 저장됨)
    System.out.println("Seoul 포함?: " + cities.contains("Seoul")); // 출력: Seoul 포함?: true
    cities.remove("New York");
    System.out.println("뉴욕 제거 후: " + cities);   // 출력: 뉴욕 제거 후: [London, Seoul]
    cities.clear();
    System.out.println("비웠나요?: " + cities.isEmpty()); // 출력: 비웠나요?: true

    System.out.println("-----------");

    // TreeSet 예시
    TreeSet<Integer> numbers = new TreeSet<>();
    numbers.addAll(Arrays.asList(5, 1, 3, 2, 4));   // 여러 값 추가 (자동 정렬됨)
    System.out.println("TreeSet: " + numbers);      // 출력: TreeSet: [1, 2, 3, 4, 5]
    numbers.add(3); // 중복값 3 추가 시도 (무시됨, 예외 없음)
    System.out.println("3 추가 후: " + numbers);    // 출력: 3 추가 후: [1, 2, 3, 4, 5] (변화 없음)

    System.out.println("최소값: " + numbers.first()); // 출력: 최소값: 1
    System.out.println("최대값: " + numbers.last());  // 출력: 최대값: 5
    System.out.println("3 초과 첫 값: " + numbers.higher(3)); // 출력: 3 초과 첫 값: 4
    System.out.println("3 미만 첫 값: " + numbers.lower(3));  // 출력: 3 미만 첫 값: 2
    System.out.println("5 이하 첫 값: " + numbers.floor(5));  // 출력: 5 이하 첫 값: 5
    System.out.println("5 이상의 첫 값: " + numbers.ceiling(5)); // 출력: 5 이상의 첫 값: 5
  }
}
```

`HashSet` 예시에서는 `"Seoul"`을 두 번 `add` 했지만 한 번만 저장되어 있는 것을 확인できます. `HashSet`의 `add`는 반환값이 `boolean`으로, 추가에 성공하면 `true`, 이미 존재해서 추가가 안 되면 `false`를 반환합니다. `contains(Object)`로 값 존재를 빠르게 확인할 수 있고, `remove(Object)`로 값을 제거할 수 있습니다. HashSet은 요소간 순서가 없으므로 출력 결과의 순서는 일정하지 않습니다 30 . 모든 요소를 제거할 때는 `clear()`를 사용합니다.

`TreeSet` 예시에서는 정수들을 추가한 뒤 출력하면 자동으로 **정렬된 오름차순**으로 저장된 것을 볼 수 있습니다. 중복 추가를 시도해도 아무 변화가 없으며 예외도 발생하지 않습니다. `first()`와 `last()` 메서드로 최소값과 최대값

을 쉽게 얻을 수 있고, `higher(E e)`는 주어진 값보다 큰 **다음 값**을, `lower(E e)`는 주어진 값보다 작은 **이전 값**을 반환합니다. `ceiling(E e)`는 **주어진 값 이상**에서 가장 작은 값을, `floor(E e)`는 **주어진 값 이하**에서 가장 큰 값을 반환하므로, 해당 값이 집합에 있으면 본인을, 없으면 바로 다음/이전 값을 주는 메서드입니다. 이러한 메서드들은 이진 탐색 트리 구조인 TreeSet의 장점을 보여줍니다.

주요 메서드 (Set 공통, HashSet/TreeSet):

- `add(E e)` – 집합에 요소 추가. 추가 성공 시 `true`, 이미 존재하면 `false` 반환 [32].
- `addAll(Collection)` – 다른 컬렉션의 모든 요소를 추가 (합집합 연산처럼 동작).
- `contains(Object o)` – 해당 요소가 집합에 존재하면 `true` [33].
- `remove(Object o)` – 해당 요소를 제거. 제거 성공 시 `true` (없으면 `false`).
- `size()` – 집합에 담긴 요소 수.
- `isEmpty()` – 집합이 비어있는지 여부.
- `clear()` – 모든 요소 제거 (집합 비우기).
- **(TreeSet 전용)** `first()`, `last()` – 정렬 순서상의 최소/최대 요소 반환 [34].
- **(TreeSet 전용)** `higher(E e)`, `lower(E e)` – 주어진 값보다 큰 **다음 요소** / 작은 **이전 요소** 반환 [35].
- **(TreeSet 전용)** `ceiling(E e)`, `floor(E e)` – 주어진 값 이상/이하인 첫 요소 반환.

(참고: HashSet과 TreeSet 모두 `Iterator`나 향상된 for문(`for-each`)을 통해 요소를 순회(iterate)할 수 있습니다. HashSet은 순서가 없음을 유의하고, TreeSet은 정렬된 순서로 순회됩니다. 또한, LinkedHashSet이라는 구현체는 HashSet처럼 해시 기반이지만 입력된 순서를 유지하므로, 필요에 따라 사용할 수 있습니다.)

6. PriorityQueue 사용법 (기본 오름차순 및 사용자 정의 정렬)

PriorityQueue는 **우선순위 큐** 구조를 제공하는 클래스입니다. 우선순위 큐는 들어간 순서와 상관없이 **우선순위가 높은 요소**를 먼저 나오는 큐입니다. Java의 `PriorityQueue`는 기본적으로 **작은 값이 높은 우선순위**(낮은 숫자가 먼저 나옴)인 **Min-Heap(최소 힙)**으로 동작합니다 [36]. 예를 들어 정수를 넣으면 작은 정수부터 `poll()`에 의해 나오게 됩니다.

PriorityQueue는 내부적으로 힙(완전이진트리 형태)을 사용하며, 요소 추가/제거에 O(log N)의 시간이 걸립니다. 기본 순서를 바꾸고 싶을 때 (예: 큰 값부터 나오게 하거나, 객체의 특정 비교 기준에 따라 우선순위 정의) **Comparator를 제공**하여 우선순위를 직접 정의할 수 있습니다. 또는 저장되는 객체 클래스가 `Comparable`을 구현하여 자연순서를 갖도록 할 수도 있습니다.

아래는 `PriorityQueue`의 기본 사용과 사용자 정의 Comparator를 통한 역순 정렬(최대 힙) 사용 예시입니다:

```
import java.util.PriorityQueue;
import java.util.Comparator;

public class ExamplePQ {
  public static void main(String[] args) {
    // 기본 PriorityQueue (오름차순: 작은 값이 우선)
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    minHeap.offer(5);
    minHeap.offer(1);
    minHeap.offer(9);
    minHeap.offer(3);
    System.out.print("오름차순 Poll 순서: ");
    while (!minHeap.isEmpty()) {
```

```
            System.out.print(minHeap.poll() + " ");
        }
        // 출력: 오름차순 Poll 순서: 1 3 5 9  (작은 숫자부터 poll)

        // 사용자 정의 PriorityQueue (내림차순: 큰 값이 우선)
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
        maxHeap.add(5);
        maxHeap.add(1);
        maxHeap.add(9);
        maxHeap.add(3);
        System.out.print("\n내림차순 Poll 순서: ");
        while (!maxHeap.isEmpty()) {
            System.out.print(maxHeap.poll() + " ");
        }
        // 출력: 내림차순 Poll 순서: 9 5 3 1  (큰 숫자부터 poll)
    }
}
```

첫 번째 `minHeap`은 기본 설정으로 **오름차순 우선순위 큐**입니다. `offer`(또는 `add`)로 값을 추가하고, `poll`로 값을 꺼낼 때 넣은 순서와 관계없이 항상 **가장 작은 값**부터 나오게 됩니다. 예시의 출력 결과에서 `1, 3, 5, 9` 순으로 나오는 것을 확인할 수 있습니다.

두 번째 `maxHeap`은 `PriorityQueue`를 생성할 때 `Comparator.reverseOrder()`를 인자로 넘겨주어 **내림차순 우선순위 큐**로 만든 것입니다. 이렇게 하면 큰 값이 먼저 나오므로, `poll` 시 `9, 5, 3, 1` 순으로 출력됩니다. (Comparator를 직접 람다나 익명 클래스 형태로 제공하여 더 복잡한 기준의 우선순위를 정의할 수도 있습니다. 예를 들어 객체의 특정 필드값을 기준으로 정렬하는 등.)

주요 메서드 (PriorityQueue):

- `offer(E e)` – 큐에 요소 추가 (`add`와 동일한 기능). 자동으로 힙 특성에 따라 자리가 정해집니다 [37].
- `poll()` – 가장 우선순위 높은 (작은 값 또는 정의된 기준상의 최대/최소) 요소를 제거하며 반환. 비어있으면 `null`.
- `peek()` – 우선순위 최고인 요소를 제거하지 않고 반환. 비어있으면 `null`.
- `isEmpty()`, `size()` – 큐가 비었는지 여부, 요소 개수.
- **기타**: `Comparator<? super E> comparator()` – 사용중인 비교자를 반환 (없으면 자연 순서). (PriorityQueue는 순회 시 정렬된 순서를 보장하지 않습니다. 필요한 경우 `poll()`를 사용해 하나씩 꺼내거나, `toArray() / Arrays.sort()` 등을 이용해야 합니다.)

7. String과 StringBuilder의 사용법과 차이점

문자열을 다루는 것은 코딩 테스트에서 아주 빈번합니다. Java에서는 `String` 클래스가 불변(immutable) 객체로 제공되고, 문자열 리터럴을 편하게 사용할 수 있습니다. 반면 `StringBuilder`는 가변(mutable) 객체로 문자열을 효율적으로 조작할 때 유용합니다 [6]. 두 클래스의 주요 차이점과 사용법을 정리하면 다음과 같습니다:

- **String (불변 객체)**: 한 번 생성되면 그 문자열 값이 변하지 않습니다. 예를 들어 문자열 연결 연산(`+`)을 하면 기존 `String`을 수정하는 것이 아니라 새로운 `String` 객체를 생성합니다. 이러한 특성 때문에 작은 문자열 조작에는 문제가 없지만, 반복문에서 문자열을 누적 연결하면 성능상 비효율적입니다. 대신 `String`은 **문자열 리터럴**을 직접 사용할 수 있고, equals 비교나 해시키(key)로 활용될 때 안정적인 동작을 합니다. 자주 쓰

이는 메서드로는 `length()`, `charAt(index)`, `substring(begin, end)`, `indexOf(str)` 등이 있고, `split(regex)`을 통해 구분자로 문자열을 나누는 기능도 있습니다.

- **StringBuilder (가변 객체)**: 문자열을 추가(`append`), 수정, 삭제할 수 있는 가변 클래스입니다. 내부 버퍼를 가지고 있어 크기를 동적으로 변경하며, 문자열 연결 시 새로운 객체를 만들지 않고 내부에서 내용을 변경하므로 메모리와 속도 면에서 유리합니다 [38]. 특히 **반복문으로 문자열을 누적할 때** `StringBuilder`를 사용하면 훨씬 빠르게 처리할 수 있습니다 (예: 대량의 출력을 한 번에 모아서 처리하거나, 문자열을 점진적으로 만들어가는 알고리즘). 사용 후 최종 결과가 필요하면 `toString()`으로 `String` 객체를 얻습니다. 주요 메서드로 `append(...)`, `insert(offset, ...)`, `delete(start, end)`, `replace(start, end, str)`, `reverse()` 등이 있으며, `length()`와 `charAt()`도 제공하여 문자열처럼 다룰 수 있습니다.

다음 예시는 `String`과 `StringBuilder`의 사용 차이를 보여줍니다:

```
public class ExampleString {
  public static void main(String[] args) {
    // String (불변) 사용 예시
    String s = "Hello";
    s += " World"; // 문자열 연결 -> 새로운 String 생성
    System.out.println(s);            // 출력: Hello World
    System.out.println("길이: " + s.length());    // 출력: 길이: 11
    System.out.println("5번째 문자: " + s.charAt(4)); // 출력: 5번째 문자: o  (0부터 시작하여 4번째 인덱스 문자)
    System.out.println("World 포함?: " + s.contains("World")); // 출력: World 포함?: true
    System.out.println("llo로 끝나나?: " + s.endsWith("llo"));  // 출력: llo로 끝나나?: false

    // StringBuilder (가변) 사용 예시
    StringBuilder sb = new StringBuilder();
    sb.append("Hello");
    sb.append(" World");     // 동일한 문자열 연결이지만 새로운 객체 생성 안 함
    sb.insert(5, ",");      // 인덱스 5 위치에 콤마 추가 -> "Hello, World"
    System.out.println(sb.toString());        // 출력: Hello, World
    sb.replace(0, 5, "Hi");   // 인덱스 0~4 문자열을 "Hi"로 교체 -> "Hi, World"
    System.out.println(sb.toString());        // 출력: Hi, World
    sb.delete(3, sb.length()); // 인덱스 3부터 끝까지 삭제 -> "Hi,"  (문자열 일부 제거)
    System.out.println(sb.toString());        // 출력: Hi,
    sb.setLength(0);       // StringBuilder 초기화 (길이를 0으로 설정)
    sb.append("ABC").append("DEF");
    sb.reverse();          // 문자열 뒤집기
    System.out.println(sb.toString());        // 출력: FEDCBA
  }
}
```

위 코드에서 `String s`에 `"Hello"`를 대입한 후 `s += " World"`를 하자, 내용은 `"Hello World"`로 보이지만 내부적으로는 새로운 `String` 객체가 만들어져 `s`가 가리키도록 합니다. 이어서 몇 가지 `String` 메서드를 사용했는데, `length()`로 문자열 길이를 확인하고, `charAt(4)`로 0부터 세어 4번째 인덱스(5번째 문자)를 가져왔습니다. `contains(str)`은 부분 문자열 포함 여부, `endsWith(str)`는 해당 문자열로 끝나는지 여부를 boolean으로 알려줍니다. (또한 `startsWith(str)`, `substring(begin, end)`, `toUpperCase()`, `toLowerCase()` 등 다양한 메서드가 `String`에 있습니다.)

`StringBuilder sb` 부분에서는 동일한 문자열 연결 작업을 `append`로 수행했습니다. `append`를 두 번 호출해도 하나의 `StringBuilder` 객체 내에서 버퍼가 변경될 뿐이고, 최종 `toString()` 결과는 `"Hello World"`가 됩니다. 이어서 `insert(5, ",")`를 호출해 인덱스 5 위치 (문자 ' ' 공백 자리)에 쉼표를 삽입하여 `"Hello, World"`로 만들었습니다. `replace(0, 5, "Hi")`는 인덱스 0부터 4까지의 문자열("Hello")을 "Hi"로 바꾸어 `"Hi, World"`가 되었고, `delete(3, sb.length())`는 인덱스 3부터 끝까지 (", World" 부분) 삭제하여 `"Hi,"`만 남겼습니다. `setLength(0)`은 StringBuilder를 빈 상태로 초기화하는 방법이며, 그 후 `"ABC"`와 `"DEF"`를 append하여 `"ABCDEF"`를 만들고, `reverse()`로 뒤집어서 `"FEDCBA"`가 출력되었습니다. 이처럼 `StringBuilder`는 String과 달리 내용 변경이 자유롭고 다양한 조작 메서드를 제공합니다.

요약 – 언제 어떤 것을 사용해야 할까?:
- **문자열 상수나 짧은 문자열 조합** 정도는 그냥 `String`을 사용해도 무방합니다. `String`은 편리한 리터럴 표기와 풍부한 메서드를 갖추고 있으며, 불변 객체라 다룰 때 실수로 내용이 변조되지 않는 장점이 있습니다. 또한 HashMap의 키 등으로 활용시 안정적입니다. - **문자열을 반복문 등에서 누적하거나 큰 문자열을 만들어야 하는 경우** `StringBuilder` (또는 멀티스레드 환경에서는 `StringBuffer`)를 사용하는 것이 좋습니다 [6]. 예를 들어 백준에서 많은 출력을 처리할 때, 하나의 StringBuilder에 모든 결과를 `append`한 후 한 번에 출력하면 시간 효율이 크게 향상됩니다. - **문자열 처리 알고리즘**에서 문자열을 거꾸로 뒤집거나 특정 위치에 문자를 삽입/삭제하는 등의 작업이 필요하면 `StringBuilder`/`StringBuffer`를 활용하는 것이 편리합니다 (`reverse()`, `insert()`, `delete()` 메서드 등).

자주 사용하는 메서드 (String & StringBuilder):

- **String 클래스**:
- `length()` – 문자열 길이 반환.
- `charAt(index)` – 해당 인덱스의 문자 반환.
- `substring(begin, end)` – 부분 문자열 추출.
- `equals(Object)` – 문자열 비교 (내용이 같은지 여부, `==`은 객체 동일성 비교이므로 문자열 비교시에는 `equals` 사용!).
- `indexOf(String)` – 특정 문자열이 처음 등장하는 위치 인덱스 (없으면 -1).
- `contains(CharSequence)` – 문자열에 부분 문자열이 포함되어 있는지 여부.
- `startsWith(String)` / `endsWith(String)` – 접두사/접미사 여부 확인.
- `split(String regex)` – 정규식 또는 구분자로 문자열을 분리하여 문자열 배열 반환.
- `trim()` – 문자열 앞뒤의 공백 제거 (중간 공백은 제거하지 않음).

- **StringBuilder 클래스**:

- `append(Object)` – 끝에 다양한 타입의 데이터를 추가 (연결). [39]
- `insert(int offset, String)` – 지정한 위치에 문자열/문자 삽입. [40]
- `delete(int start, int end)` – 지정 범위의 문자를 삭제. [41]
- `replace(int start, int end, String)` – 지정 범위의 문자열을 새로운 문자열로 대체.
- `reverse()` – 문자열 내용 뒤집기. [42]
- `toString()` – 현재 버퍼의 문자열을 `String` 형태로 반환.
- `length()` – 현재 문자열 길이 (capacity와 다를 수 있음).
- `setLength(int newLength)` – 문자열 길이를 강제로 변경 (줄이는 경우 뒷부분 삭제, 늘리는 경우 널 문자 `\0`로 채움).

以上와 같이 Java에서 **입출력 효율을 높이는 방법**부터 **기본 자료구조 활용법**, **정렬**, **우선순위 큐**, **문자열 처리**까지 살펴보았습니다. 이러한 내용은 Java 8 기준으로 작성되었으며, 백준 코딩테스트의 **문자열 처리**, **자료구조 활용**, **정렬 문제**, **BFS/DFS 탐색** 등에 유용하게 활용할 수 있습니다. 각 주제별 예시 코드를 천천히 분석해보고, 필요할 때 응용하면 많은

코딩 테스트 문제를 효과적으로 풀 수 있을 것입니다. 祝 코딩 테스트 준비에 도움이 되길 바랍니다! Good luck! [1] [2]

---

[1] [2] [3] [Java] BufferedReader, StringTokenizer
https://velog.io/@postrel63/%EB%B0%B1%EC%A4%80-BufferedReader-StringTokenizer

[4] [5] [6] [38] [39] [40] [41] [42] JAVA - BufferedReader, BufferedWriter, ,StringBuilder StringTokenizer 사용법 — 설명을 위한 기록
https://aibigdata-study.tistory.com/entry/JAVA-BufferedReader-BufferedWriter-StringBuilder-StringTokenizer-%EC%82%AC%EC%9A%A9%EB%B2%95

[7] [8] [JAVA] Array, List, ArrayList 개념 정리
https://zks145.tistory.com/62?category=1100496

[9] [21] 자바 Set, HashSet, TreeSet, HashMap 정리
https://n1tjrgns.tistory.com/102

[10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [36] [37] LinkedList, Stack, Queue, Deque, PriorityQueue 사용법 정리
https://zks145.tistory.com/124

[22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [JAVA] HashMap, HashSet, TreeMap, TreeSet 사용 방법
https://zks145.tistory.com/73