
Data Structure & Algorithm

목 차

1. 자료구조와 알고리즘 개요

1.1 자료구조 개요

✓ 자료구조(Data Structure)는 데이터를 메모리나 디스크에 어떠한 형태로 저장할지에 대한 방법입니다.
✓ 아래와 같이 같은 내용의 데이터일지라도 다양한 형태(순서)로 저장할 수 있습니다.
✓ 이 데이터를 이용해서 수행하는 연산과, 이후에 추가하는 데이터 형태를 고려합니다.

1.2 알고리즘 개요

✓ 알고리즘(Algorithm)은 저장된 데이터를 조작하고 처리하는 절차입니다.
✓ 저장된 데이터를 정렬하는 정렬 알고리즘, 원하는 데이터를 검색하는 검색 알고리즘이 있습니다.
✓ 같은 알고리즘을 이용해서 검색을 하더라도, 현재 데이터가 저장되어있는 형태(자료구조)에 따라 수행시간이 달라집니다.

'세종'을 찾기 위한 더 효율적인 방법은 없을까?

조선 왕 목록 중 '세종' 찾기

세종

모든 왕을 하나씩 꺼내서 이름이 '세종' 인지 확인한다.

1.3 학습필요성

✓ 어플리케이션에서 데이터를 저장하는 형태에 따라 연산의 횟수, 소요시간에서 많은 차이가 발생한다.
✓ 저장한 데이터를 이용해서 어떠한 연산을 하는가에 따라 적용해야 하는 자료구조, 알고리즘이 다릅니다.
✓ 데이터를 삽입하는 시점 또는 검색하는 시점에 복잡한 연산을 합니다. (trade-off 고려)

사전에서 내가 원하는 단어를 어떻게 찾을까?

학사관리 시스템에서 학과별, 학번별 학생명단을 빠르게 조회 할 수 있을까?

입사지원서를 어떻게 분류하고, 정렬할 것인가?

2. 자료구조 기초

2.1 Array (1/4)

✓ 같은 종류의 데이터를 순차적으로 저장합니다.
✓ 배열은 인덱스로 접근이 가능합니다. 인덱스를 이용해서 해당 위치에 데이터를 저장하거나, 조회합니다.
✓ 배열은 처음 저장공간을 생성하는 시점에 크기를 지정하고, 이 크기는 변경되지 않습니다.

2.1 Array (2/4)

- ✓ 배열은 인덱스를 이용해서 값을 한 번에 접근하기 때문에 빠른 연산이 가능합니다.(순차접근 아님) → 장점
- ✓ 하지만 인덱스를 직접 입력하다 보니 예약된 범위를 쉽게 벗어나는 경우가 많습니다. → 단점
- ✓ 배열은 최초에 생성한 범위 내에서만 인덱스로 접근합니다.
- ✓ 인덱스 값이 범위를 벗어나면 Array Index Out Of Bound Exception이 발생합니다.

score[2]
배열의 2번째 원소를 조회한다.

score[5]
할당한 크기를 벗어난 위치(5번째)를 접근하면 오류발생

score[-1] = 70
할당한 크기를 벗어난 위치(-1번째)를 접근하면 오류발생

2.1 Array (3/4) – 순차접근

✓ 배열에 저장한 원소에 하나씩 순차접근을 하기 위해서는 반복문을 사용합니다.
✓ 배열은 크기가 정해져 있기 때문에 for문 (또는 for~each)을 이용해서 접근할 수 있습니다.
✓ 배열명.length 속성을 이용해서 배열의 크기를 알 수 있습니다.

```
public void traverse() {
  //
  int[] scoreArray = new int[5];
  scoreArray[0] = 85;
  scoreArray[1] = 90;
  scoreArray[2] = 95;
  scoreArray[3] = 70;
  scoreArray[4] = 100;

  for(int i=0; i<scoreArray.length; i++) {
    int value = scoreArray[i];
    System.out.println(i + " : " + value);
  }

  for(int score : scoreArray) {
    System.out.println(score);
  }
}
```

실행결과

```
0 : 85
1 : 90
2 : 95
3 : 70
4 : 100
85
90
95
70
100
```

2.1 Array (4/4) – 실습

- ✓ 아래와 같이 원소가 5개인 정수형 배열(int [])을 선언하고 값을 저장합니다.
- ✓ price 배열을 순회, 역 순회 하면서 원소를 출력합니다.
- ✓ 모든 원소의 합계, 평균을 출력하고, 최대값과 최소값을 출력합니다.
- ✓ 중간에 원소를 끼워 넣기 할 경우에는 어떻게 해야 할지 토의해봅니다.

price

0	1	2	3	4
1,000	800	750	1,300	9,000

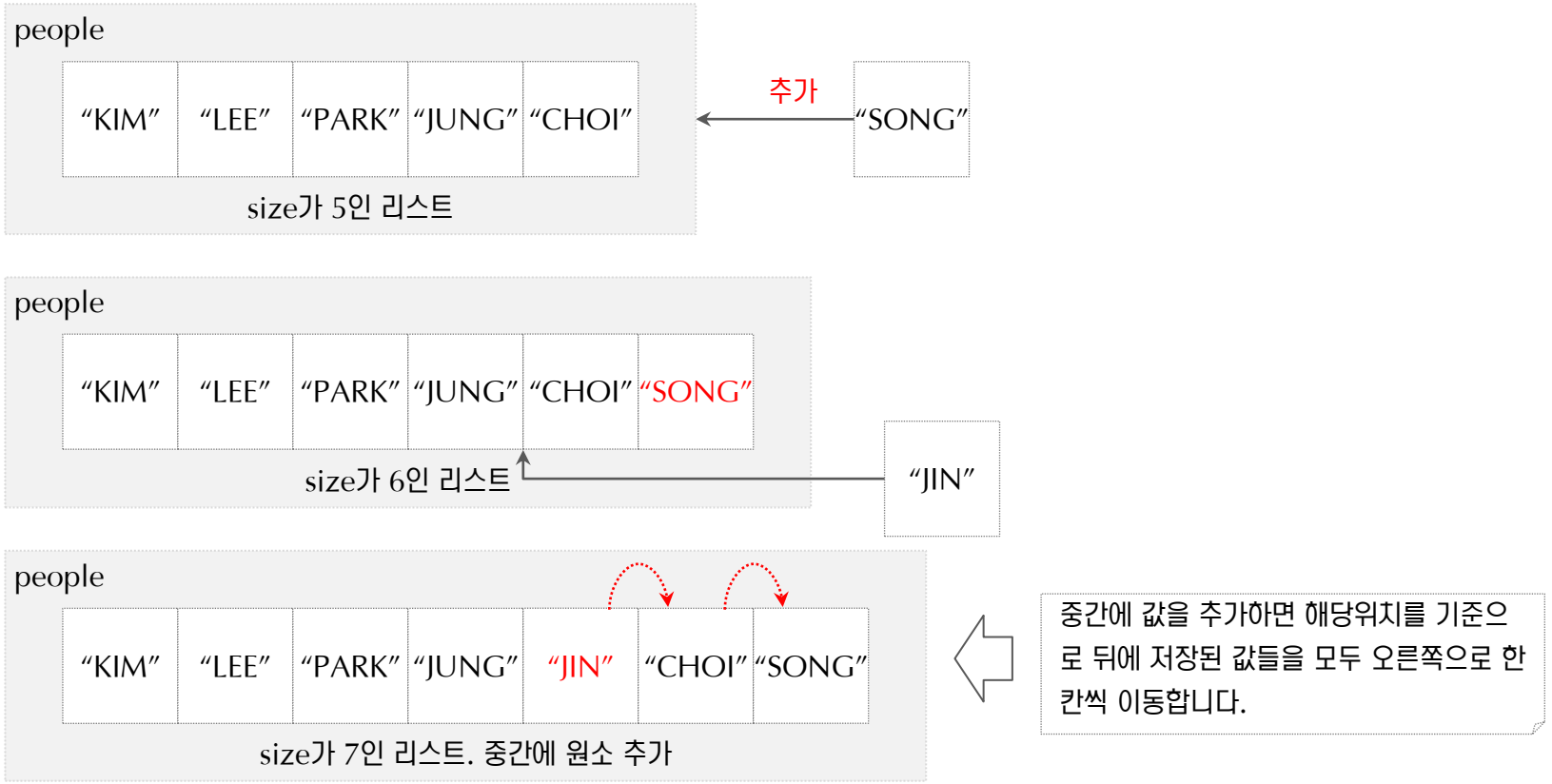
원소가 5개인 price 배열

실행결과

```
=== 순회하면서 출력 ===
0 : 1000
1 : 800
2 : 750
3 : 1300
4 : 9000
=== 역순으로 순회하면서 출력 ===
4 : 9000
3 : 1300
2 : 750
1 : 800
0 : 1000
=== 합계 ===
12850
=== 평균 ===
2570
=== 최대값 ===
9000
=== 최소값 ===
750
```

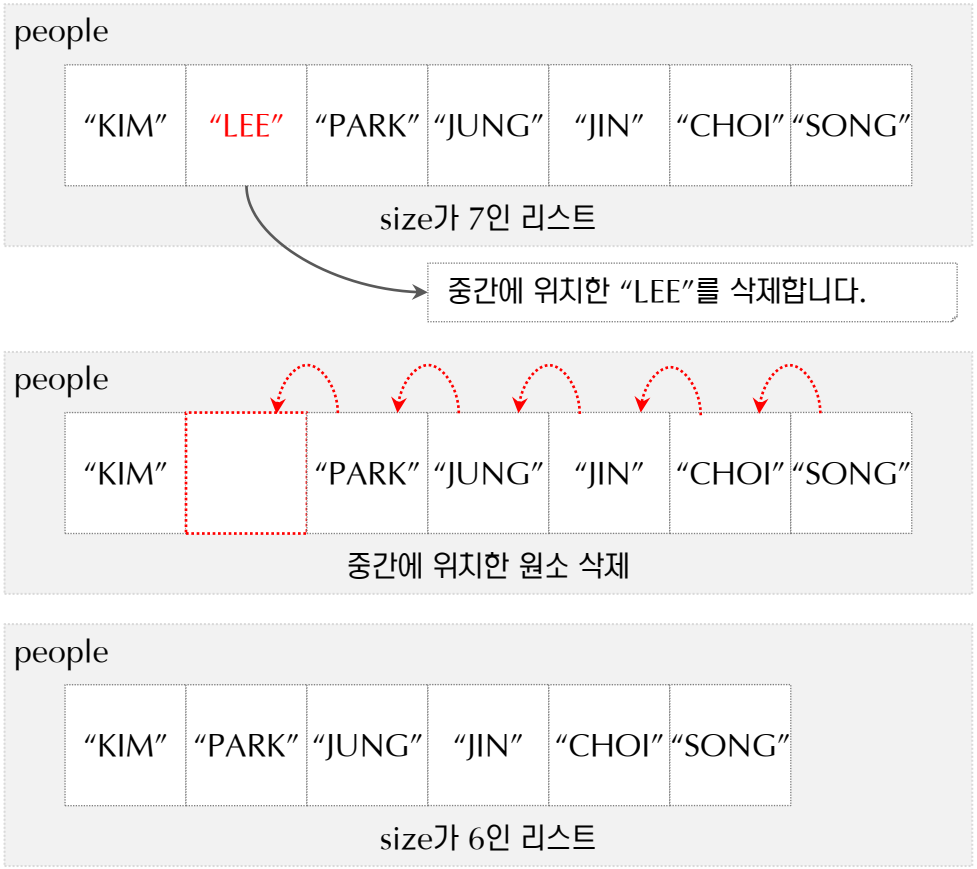
2.2 List (1/3)

- ✓ 리스트는 배열과 비슷하게 동작하고, 데이터를 목록형태(0부터 순차적으로)로 저장하고, 조회할 때 사용합니다.
- ✓ 배열과는 달리 가변 크기를 지원합니다. 데이터를 추가 할 때 마다 계속 증가하고, 중간에 삭제하면 크기도 같이 줄어듭니다.
- ✓ 리스트는 가변 크기를 지원하기 때문에 저장할 데이터 수가 고정적이지 않거나, 몇 건이 발생할지 알 수 없을 때 사용합니다.
- ✓ 인덱스를 이용해서 원하는 데이터에 한 번에 접근할 수 도 있습니다.



2.2 List (2/3)

✓ 리스트는 배열과 달리 중간에 원소를 추가하거나, 삭제할 수 있습니다.
✓ 리스트 마지막 위치에 원소를 추가하면 리스트 크기 증가 + 원소 추가 동작을 수행합니다.
✓ 중간에 원소를 추가하거나 삭제하면 해당하는 위치 기준으로 기존에 존재하는 원소를 앞, 뒤로 이동하는 연산을 수행합니다.
✓ 데이터를 중간위치에 자주 추가, 삭제하면 기존 데이터들을 이동하는 연산이 지속적으로 발생하기 때문에 연산이 느립니다.



2.2 List (3/3) – 실습

✓ 아래와 같은 기능을 수행하는 LIST를 구현합니다. LIST 내부에서는 배열을 이용해서 처리합니다.

✓ add()와 remove() 연산을 수행한 후 size()를 수행하면 결과값이 늘어나거나 줄어들어야 합니다.

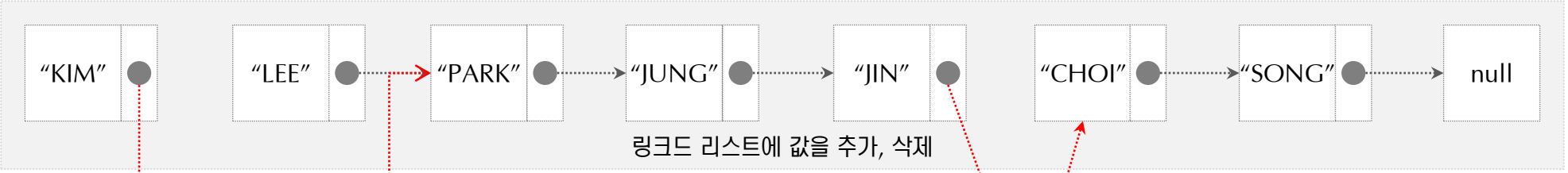
```
/**
 * 리스트에 항목을 추가한다.
 */
public void add(String name);

/**
 * <PRE>
 * 리스트에서 name에 해당하는 항목을 삭제한다.
 * 존재하지 않은 경우 아무 변화도 일어나지 않는다.
 * 동일한 내용의 항목이 2개 이상일 경우 앞에 저장된 항목 1개만 삭제한다.
 * </PRE>
 */
public void remove(String name);

/**
 * 리스트 크기를 반환한다.
 */
public int size();
```

```
/**
 * index 위치에 저장된 항목을 반환한다.
 */
public String get(int index);

/**
 * <PRE>
 * index 위치에 항목을 저장한다.
 * 기존에 존재하는 값은 덮어쓴다.
 * </PRE>
 */
public void set(int index, String name);
/**
 * name에 해당하는 항목이 존재하는지 여부를 반환한다.
 */
public boolean contains(String name);

/**
 * 리스트가 비었는지(empty) 여부를 반환한다.
 */
public boolean isEmpty();
```

2.3 Linked List

✓ 링크드 리스트는 각 원소가 다음 원소를 가리키는 링크를 포함합니다. 이 원소를 노드(NODE)라고 합니다.
✓ 단순 리스트와는 달리 링크드 리스트는 처음 노드 부터 시작해서 NEXT, NEXT… 링크를 참조하면서 원소를 순회합니다.
✓ 중간에 데이터를 추가하거나 삭제할 경우 다른 노드에는 영향이 없으므로 연산이 간단합니다. → 장점
✓ 모든 노드의 참조링크를 관리 해야 하는 어려움이 있습니다. 구현 복잡도가 증가합니다. → 단점

링크드 리스트에 저장된 노드



링크드 리스트에 값을 추가, 삭제

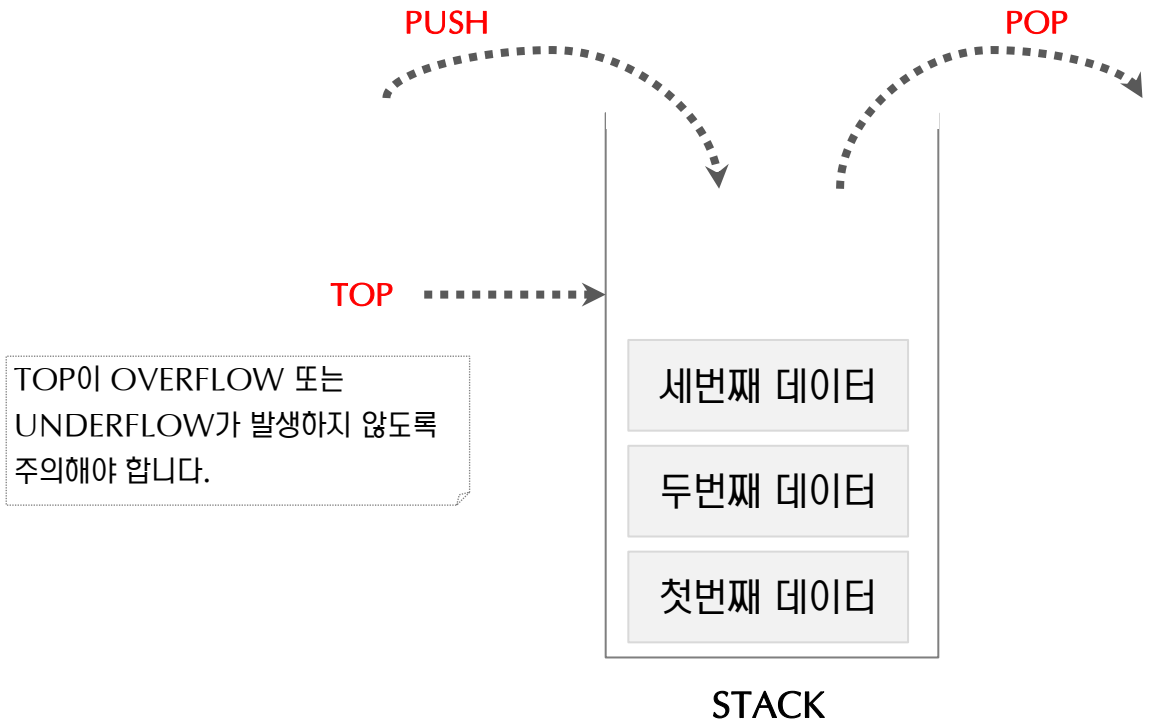
"LEE" 노드를 삭제합니다. "LEE"를 가리키던 "KIM" 노드의 참조링크를 "LEE".NEXT를 가리키도록 변경합니다.

"HA"

중간에 "HA"노드를 추가합니다. "HA".NEXT는 "JIN".NEXT를 저장하고, "JIN".NEXT는 "HA"를 가리키도록 변경합니다.

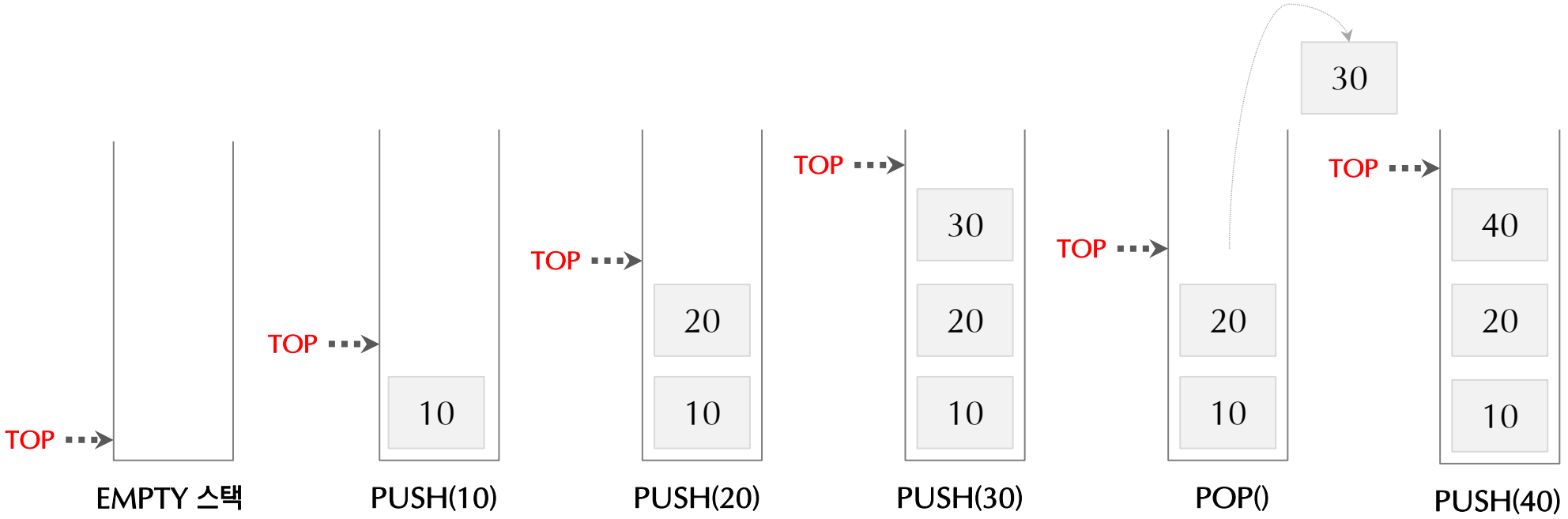
2.4 Stack (1/3)

✓ 데이터를 추가한 역순으로 조회하는 형태의 자료구조 입니다. LIFO(List In First Out)구조라고 합니다.
✓ 데이터를 차곡차곡 쌓아 올리는 형태입니다. 처음에 추가한 메시지는 가장 아래에, 마지막 데이터는 최상위에 존재합니다.
✓ 데이터 저장소의 한 쪽 끝은 막혀있고, 다른 한 쪽으로 입력/출력이 모두 발생합니다.
✓ Stack에 데이터를 저장하는 행위를 PUSH, 꺼내는 행위를 POP, 가장 최상위 데이터 위치를 TOP이라고 합니다.



2.4 Stack (2/3)

✓ 데이터를 추가한 역순으로 조회하는 형태의 자료구조 입니다. LIFO(Last In First Out)구조라고 합니다.
✓ 데이터를 차곡차곡 쌓아 올리는 형태입니다. 처음에 추가한 메시지는 가장 아래에, 마지막 데이터는 최상위에 존재합니다.
✓ 데이터 저장소의 한 쪽 끝은 막혀있고, 다른 한 쪽으로 입력/출력이 모두 발생합니다.
✓ Stack에 데이터를 저장하는 행위를 PUSH, 꺼내는 행위를 POP, 가장 최상위 데이터 위치를 TOP이라고 합니다.



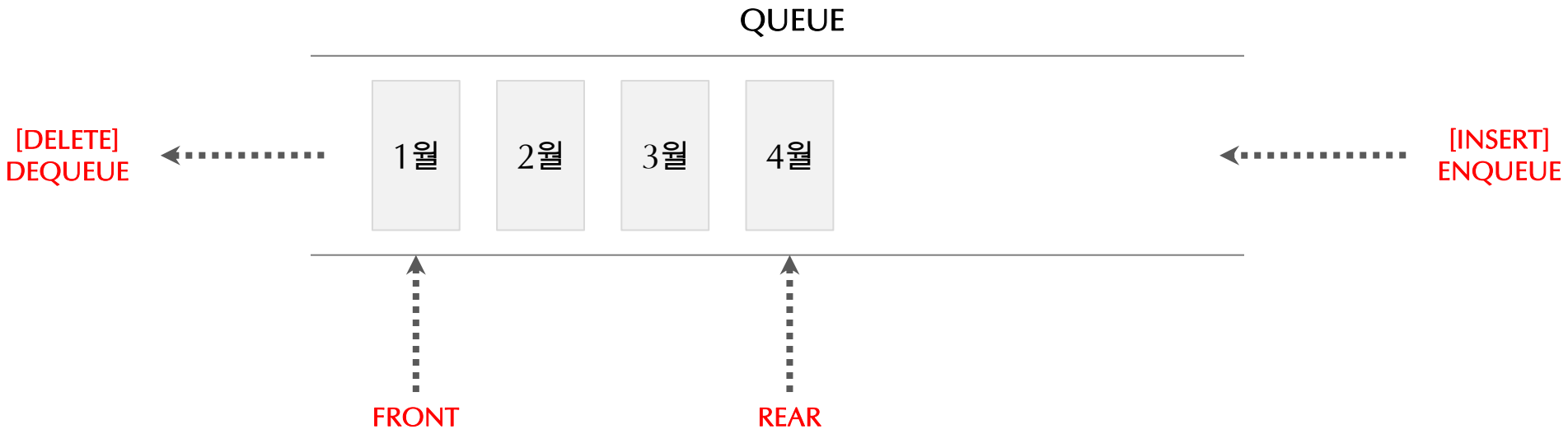
2.4 Stack (3/3) – 실습

✓ 아래와 같은 기능을 수행하는 Stack을 구현합니다. Stack 내부에서는 배열을 이용해서 처리합니다.
✓ push()는 Stack이 가득 차있으면 데이터 삽입이 안됩니다. 가득 차있을 경우 메시지를 출력합니다.
✓ pop()은 Stack이 비어있으면 데이터 반환이 안됩니다. 비어있을 경우 메시지를 출력합니다.
✓ 생성자를 이용해서 Stack 크기를 지정합니다. 예 : new Stack(10) → 크기가 10인 Stack 생성

```java
/**
 * 파라미터로 전달된 데이터를 스택에 삽입한다.
 * @param value
 */
public void push(int value);
/**
 * 스택에 저장된 데이터 중 가장 마지막에 삽입된 데이터를 반환한다.
 * @return
 */
public int pop();
/**
 * 스택이 가득찼는지 여부를 반환한다.
 * @return
 */
public boolean isFull();
/**
 * 스택이 비었는지 여부를 반환한다.
 * @return
 */
public boolean isEmpty();
```

2.5 Queue

- ✓ 데이터를 먼저 추가한 순으로 조회하는 형태의 자료구조 입니다. FIFO(First In First Out)구조라고 합니다.
- ✓ 데이터 저장소의 양쪽이 모두 열려있고, 한 쪽으로는 삽입, 다른 한 쪽으로는 조회 및 삭제를 수행합니다.
- ✓ QUEUE에는 내부적으로 두 개의 포인터가 존재하는데, FRONT는 조회 또는 삭제할 데이터를 가리키고, REAR는 마지막에 추가한 데이터를 가리킵니다.



은행에서 선착순으로 발급하는 번호표

3. 알고리즘 - 정렬

3.1 선택정렬(Selection Sort)

✓ 제자리정렬 알고리즘에 속합니다.(정렬할 데이터 수에 비하면 무시해도 좋을 만큼의 작은 저장공간을 사용하는 알고리즘)
✓ 가장 앞에 저장된 값을 기준으로 다른 값들과 비교해가면서 최소값을 찾아서 기준 값과 최소 값의 위치를 변경합니다.
✓ 기준 값 위치를 하나씩 다음 원소로 이동시키면서 최소값과 위치변경을 계속 수행합니다.
✓ 기준 값 위치가 마지막까지 이동하면 목록은 오름차순으로 정렬된 형태를 띄게 됩니다.

수행횟수	데이터						설명
1	90	60	0	50	40	30	첫 번째 원소 '90'과 최소값 '0' 자리 바꿈
2	0	60	90	50	40	30	두 번째 원소 '60'과 최소값 '30' 자리 바꿈
3	0	30	90	50	40	60	세 번째 원소 '90'과 최소값 '40' 자리 바꿈
4	0	30	40	50	90	60	네 번째 원소 '50'은 자리 유지
5	0	30	40	50	90	60	다섯 번째 원소 '90'과 최소값 '60' 자리 바꿈
-	0	30	40	50	60	90	오름차순으로 정렬된 모습

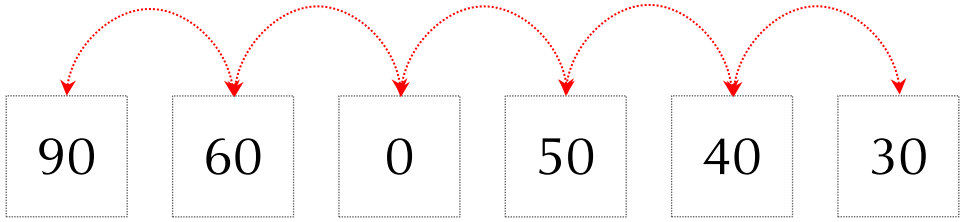
3.2 삽입정렬 (Insertion Sort)

✓ 데이터의 모든 요소를 이미 정렬된 부분과 비교해서 자신의 위치를 찾아 삽입하는 알고리즘입니다.
✓ 첫 번째 요소는 비교할 대상이 없기 때문에 적절한 위치에 정렬되어 있다고 가정하고 두 번째 요소부터 비교를 시작합니다.
✓ 비교할 기준 값 위치를 기준으로 왼쪽에 존재하는 데이터들은 이미 정렬된 데이터 입니다.

수행횟수	데이터						설명
1	90	60	0	50	40	30	두 번째 원소 '60'을 '90' 자리에 삽입
2	60	90	0	50	40	30	세 번째 원소 '0'을 '60'자리에 삽입
3	0	60	90	50	40	30	네 번째 원소 '50'을 '60' 자리에 삽입
4	0	50	60	90	40	30	다섯 번째 원소 '40'을 '50'자리에 삽입
5	0	40	50	60	90	30	여섯 번째 원소 '30'을 '40'자리에 삽입
-	0	30	40	50	60	90	오름차순으로 정렬된 모습

3.3 거품정렬 (Bubble Sort)

✓ 인접한 두 원소를 비교해서 정렬하는 방법입니다. 수행횟수는 많지만 코드가 단순합니다.
✓ 인접한 원소가 서로 자리 이동을 하는 모습이 거품(Bubble)같다고 해서 붙여진 이름입니다.
✓ 비교 횟수가 증가할 수록 가장 끝(오른쪽)에는 큰 값이 위치하게 됩니다.



#1

90	60	0	50	40	30
60	90	0	50	40	30
60	0	90	50	40	30
60	0	50	90	40	30
60	0	50	40	90	30
60	0	50	40	30	90

#2

60	0	50	40	30	90
0	60	50	40	30	90
0	50	60	40	30	90
0	50	40	60	30	90
0	50	40	30	60	90

#3

0	50	40	30	60	90
0	50	40	30	60	90
0	40	50	30	60	90
0	40	30	50	60	90

#4

0	40	30	50	60	90
0	40	30	50	60	90
0	30	40	50	60	90

#5

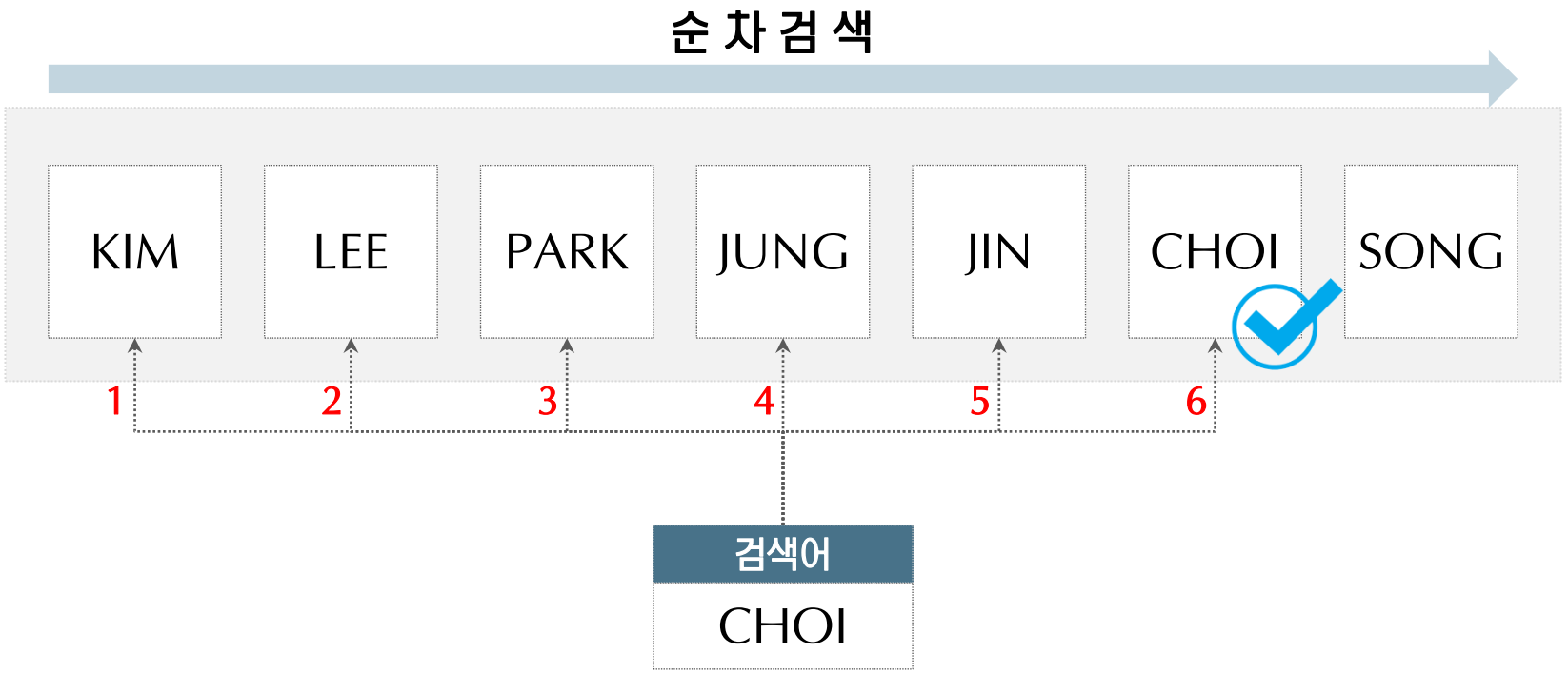
0	30	40	50	60	90

4. 알고리즘 - 검색

4.1 순차검색 (Sequential Search)

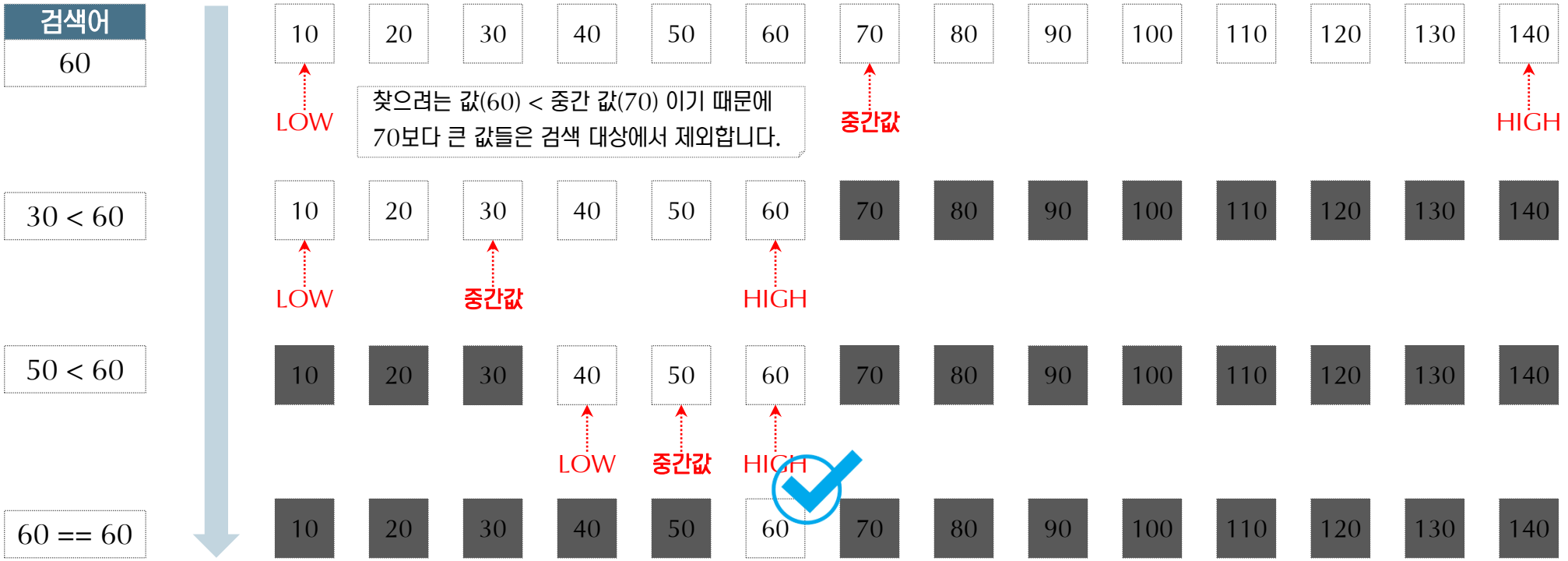
✓ 검색하려는 데이터를 목록에서 처음부터 끝까지 차례로 하나씩 비교하는 알고리즘입니다.

✓ 데이터가 많으면 비교횟수가 증가하는 단점이 있지만, 데이터가 정렬되어있지 않더라도 사용할 수 있다는 장점이 있습니다.



4.2 이진검색 (Binary Search)

✓ 검색하려는 값을 데이터 목록 중 중간에 위치한 값과 비교하면서 검색하는 방법입니다.
✓ 왼쪽 데이터 <= 기준 값 (목록 중간에 위치한 값) <= 오른쪽 데이터
✓ 데이터 목록은 정렬된 상태이어야만 이진 검색을 사용할 수 있습니다.



토론

✓ 질문과 대답

✓ 토론