개발 역량강화를 위한 알고리즘 정복 CAMP

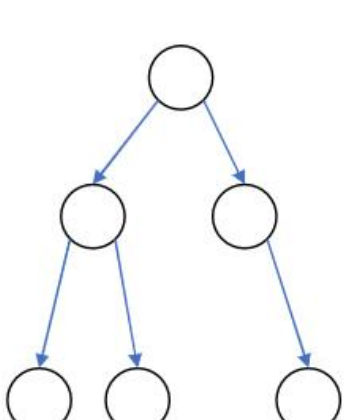
CHAPTER06 - Basic Data Structure

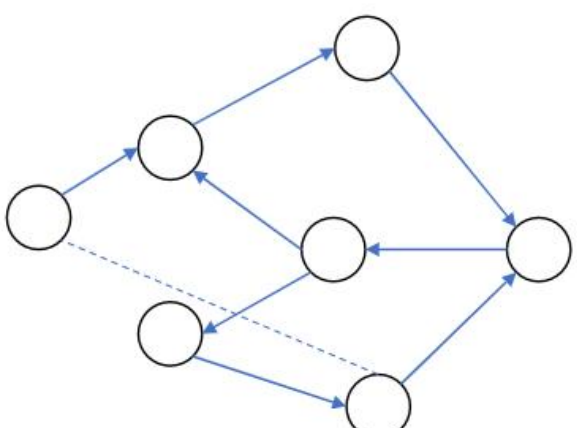
Dongyi Kim
dykim@codingmonster.net

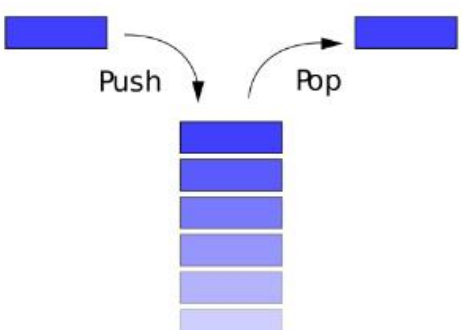
자료구조 (Review)

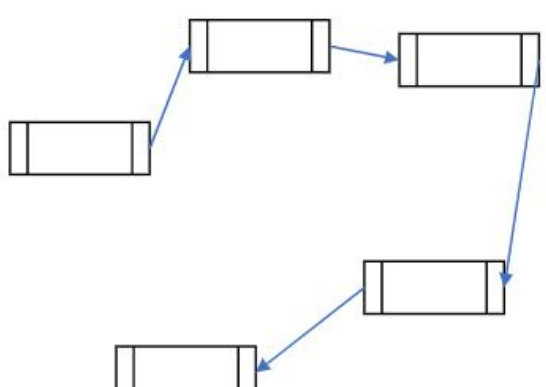
Definition>

- 다수의 데이터들을 원하는 목적에 맞게 효율적으로 저장/관리하는 방법
- 자주 사용하는 데이터 혹은 연산에 유리한 구조로 데이터를 변형하여 관리하기 위함
- 1차원적인 메모리 구조를 논리적으로 다차원 구조처럼 활용하려는 시도.
  - 실제로는 다른 데이터들처럼 1차원 메모리에 저장되겠지만,
    데이터 간의 추상적인 관계를 어떤 값으로 정의하고, 정의에 모순되지 않도록 사용하면 된다.









자료구조를 사용하는 이유 (Review)

각 데이터마다 가지는 특징이 다르며,
용도마다 데이터를 활용해 수행하고자 하는 처리가 다르다.

데이터의 특징과 사용 용도에 맞게 알맞은 자료구조를 선택해야 한다.

살면서 유명한 알고리즘을 직접적으로 개발에 사용하는 일은 드물 수 있지만,
유명한 자료구조는 개발하면서 항상 사용할 수 밖에 없다.

- 그리고 언어별로 바로 사용할 수 있는 형태로 제공하고 있다.
- 배열, 리스트, Map(Dictionary), Set, Heap(Priority Queue), Stack, Queue, …

그렇기에 각 자료구조의 주요 용도와 장/단점은 무엇인지 고려하며 사용해야 한다.

배열의 특성 (Review)

배열은 가장 기본적인 선형 자료구조다. 메모리 상에서 연속된 공간을 차지하며, 각 칸은 인덱스로 구분된다.

배열의 이름과 인덱스를 통해 특정 위치(순서)에 해당하는 원소에 빠르게 접근할 수 있다.

번호/순서를 기준으로 관리되는 데이터들을 저장하기에 가장 간단하고 편하다.

```
int[] arr = new int[n];
```

배열의 특성 – Pros and Cons (Review)

Pros>

- Index를 통한 각 원소의 접근이 O(1)만에 이루어진다.
- 상대적인 순서와 위치를 기준으로 한 관리와 탐색이 용이하다.
- N개의 데이터를 저장하기 위해 O(N)만큼의 공간만을 사용할 수 있다.
- 정렬을 통해 배열에서의 순서와 원소의 값 사이의 대소 관계가 일치하게 할 수 있다.

Cons>

- 데이터의 '값'을 기준으로 한 탐색이 비효율적이다.
  - 정렬을 통해 부분적으로 극복할 수 있다.
- 공간의 크기 변경이 자유롭지 못하다.
  - 새 공간을 만들고 기존의 내용들을 모두 Copy & Paste해야 한다.
- 데이터의 불규칙적인 추가/삭제 시 사용하지 않는 공간에 대한 관리가 번거롭다.
- 여러 데이터의 순서 변경이 비효율적이다.
  - 배열의 Shift연산, 순서 뒤집기 등
- 사용 가능한 인덱스가 0부터 연속적인 정수로 제한되어 있다.
  - 큰 인덱스 번호를 사용하기 힘들다.

이번 챕터에서는?

가장 자주 사용되는 대표적인 자료구조의 정의와 장/단점에 대해 알아본다.

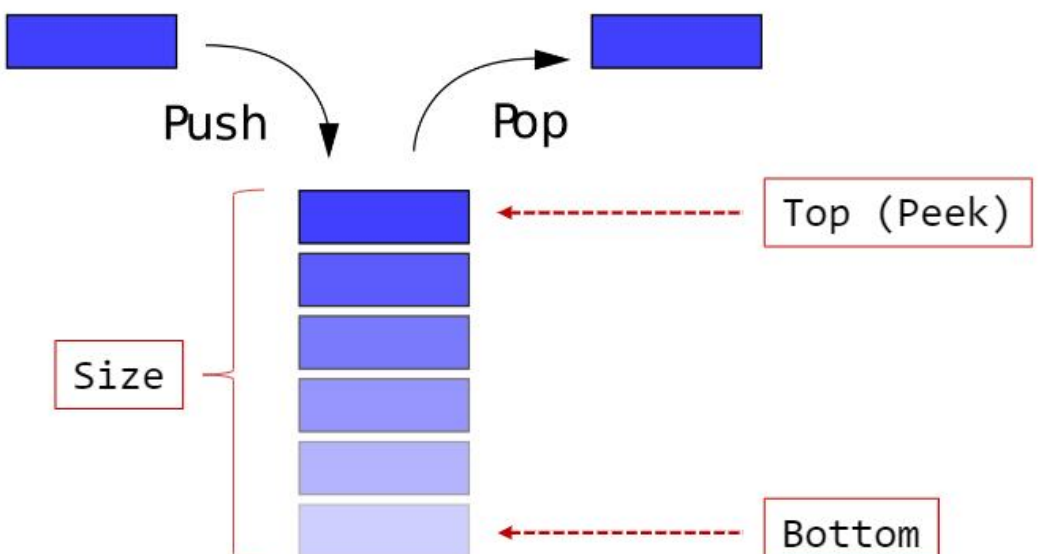
- Stack
- Queue
- Priority Queue
- Set
- Map

각 자료구조와 배열이 가지는 장/단점을 비교해보고, 문제별로 적합한 자료구조를 선택해서 적용해본다.

Stack

순차적으로 데이터를 저장하며, 가장 마지막 원소의 접근이 가능한 자료구조.

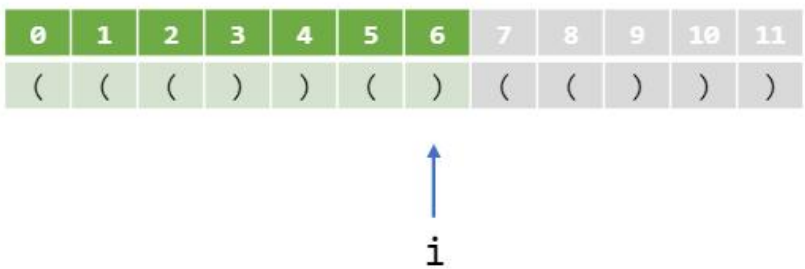
- 가장 뒤(Top)의 원소 한정이지만, 빠르게 추가/삭제할 수 있는 가변적인 구조
- 추가/삭제가 일어날 때 마다 Top에 대한 갱신을 쉽게 할 수 있다.
- 내부 구현은 보통 Linked List 혹은 배열로 되어 있다.



* 운영체제는 프로그램의 지역변수와 함수 호출도 스택구조로 관리한다.

Problem 06A. 괄호 문자열

Stack은 데이터를 순차적으로 조회하다가,
불필요한 데이터를 제거하며 전체적인 순서를 유지해야할 때 사용하면 용이하다.



현재 i번째 문자까지 검사를 완료했다고 가정하자.

s[1] ~ s[6]까지의 문자열은 이후에 유효성을 고려하는 데에 필요할까?

Problem 06A. 괄호 문자열

특정 괄호 문자에 대해 짝을 결정하는 것은 쉽지 않다.

- 하지만 바로 좌/우로 인접한 괄호쌍은 자명하게 서로 짝이다.

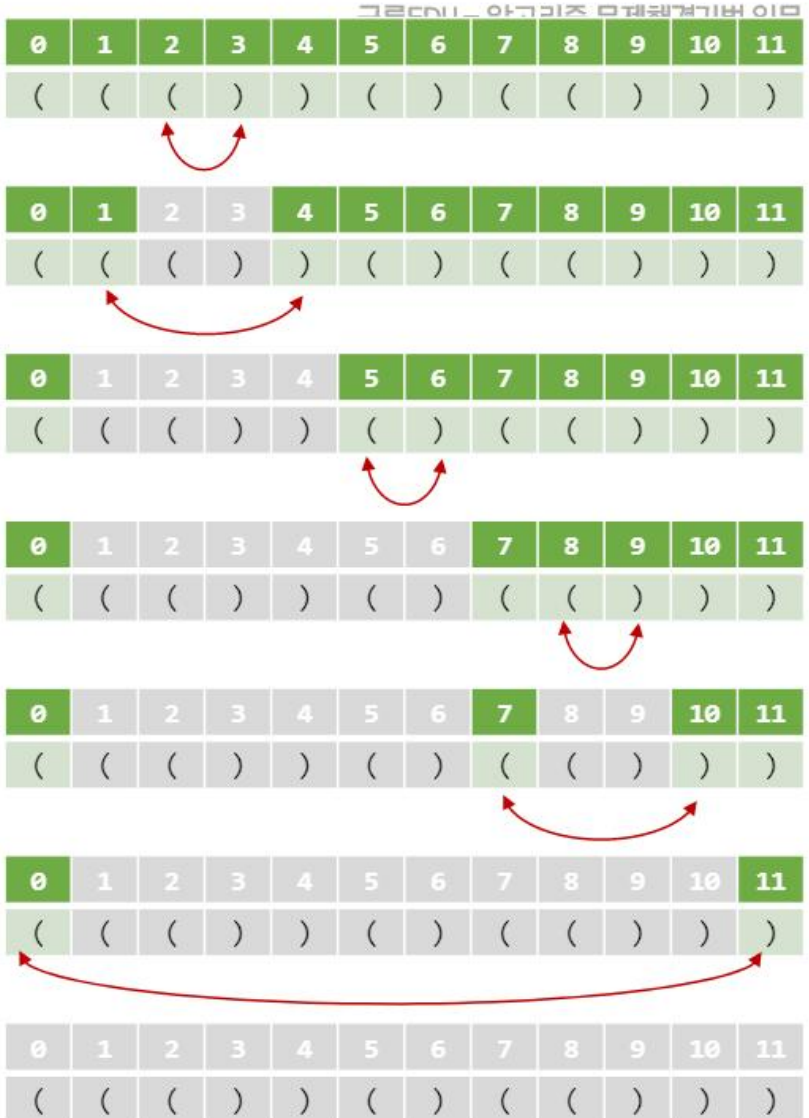
이미 짝이 자명해진 괄호들은 없다고 가정해보자.

- 짝이 자명한 괄호들을 제거했을 때 인접한 괄호쌍들도 짝이다.

즉, 짝이 자명한 괄호들은 제거하는 것이 편리하다.

- 하지만 배열은 중간 원소 삭제가 비효율적이다.
- 인접한 괄호들 끼리만 짝이 되므로 순차적으로 검사할 수 있다.

그러므로, 스텍을 사용하자!



Problem 06A. 괄호 문자열

```
// 현재 짝을 찾지 못한 괄호들을 저장할 스텍
Stack<Parenthesis> stack = new Stack<>();
```

왼쪽에서부터 괄호 문자를 하나씩 고려 대상에 추가한다.

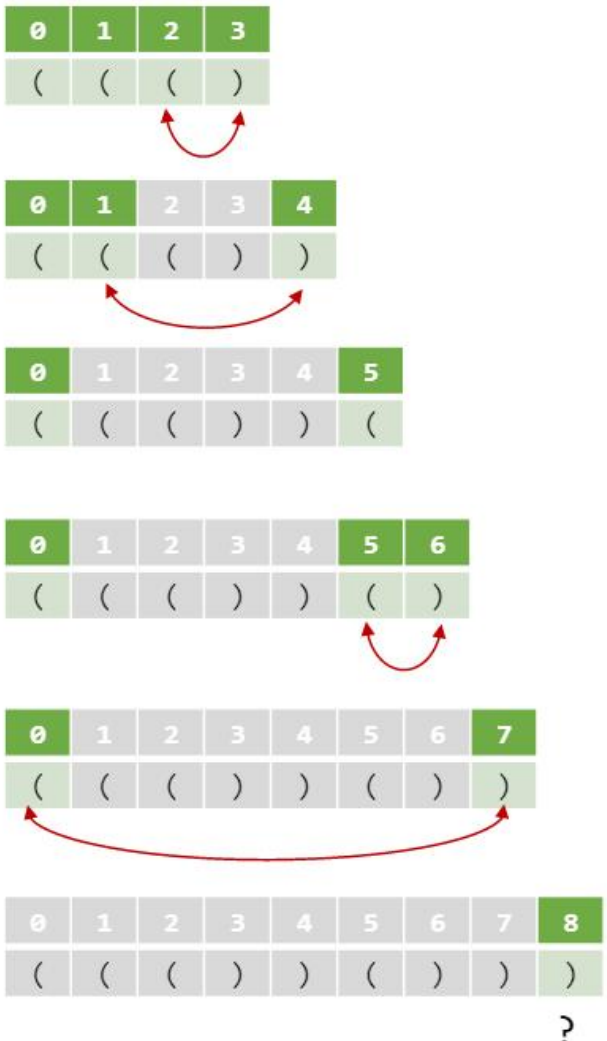
- 회색 영역은 '제거 완료 된 영역'이라고 고려하자

가장 마지막에 등장한 두 문자가 서로 짝이 맞다면?

- 둘은 서로 자명하게 짝지어지므로 고려 안해도 된다.
- 고려하지 않기 위해 Stack에서 Pop하여 제거한다.

가장 마지막에 등장한 닫힌 괄호에 대해서

- 대응되는 열린 괄호가 존재하지 않는다면?
- 짝이 맞지 않는 문자열임이 자명하다.



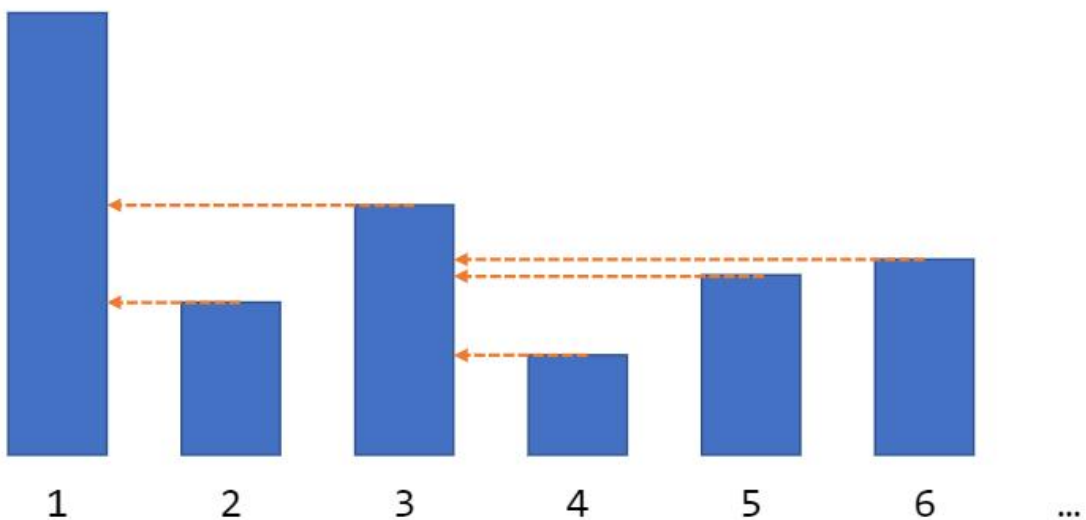
* 어차피 닫힌 괄호는 열린 괄호에 대한 검증을 위해서만 사용한다. 즉, 스택에는 열린 괄호만 저장해도 충분하다

Problem 06A. 괄호 문자열

```
for (int i = 0; i < n; i+= 1) {
   // 왼쪽부터 오른쪽의 괄호를 차례로 조회한다.
   Parenthesis p = parentheses[i];

   if (p.type == Parenthesis.OPEN) {
      // 열린 괄호라면 짝을 찾을 때 까지 스택에 보관한다
      stack.push(p);
   } else if (p.type == Parenthesis.CLOSE) {
      // 닫힌 괄호 p에 대하여

      if (stack.size() > 0 && stack.peek().type == Parenthesis.OPEN) {
         // 가장 마지막에 추가된 열린 괄호와 짝을 맞출 수 있으므로 제거한다
         stack.pop();
      } else {
         // 짝을 맞출 수 있는 열린 괄호가 없다면 올바르지 않은 괄호 문자열이다.
         return false;
      }
   }
}

if (stack.size() > 0) {
   // 아직 스택에 짝을 찾지 못한 열린 괄호가 남아있다.
   return false;
}

return true;
```

Problem 06B. 탑

각 탑은 왼쪽으로 레이저를 송신하며, 송신탑 높이 이상의 탑이 레이저를 수신한다.
왼쪽에 있는 탑부터 차례로 송신 대상이 되는 탑을 계산해 나간다고 생각해보자
대상 타워를 찾기 위해 이전의 타워들을 계속 조회해야 한다.

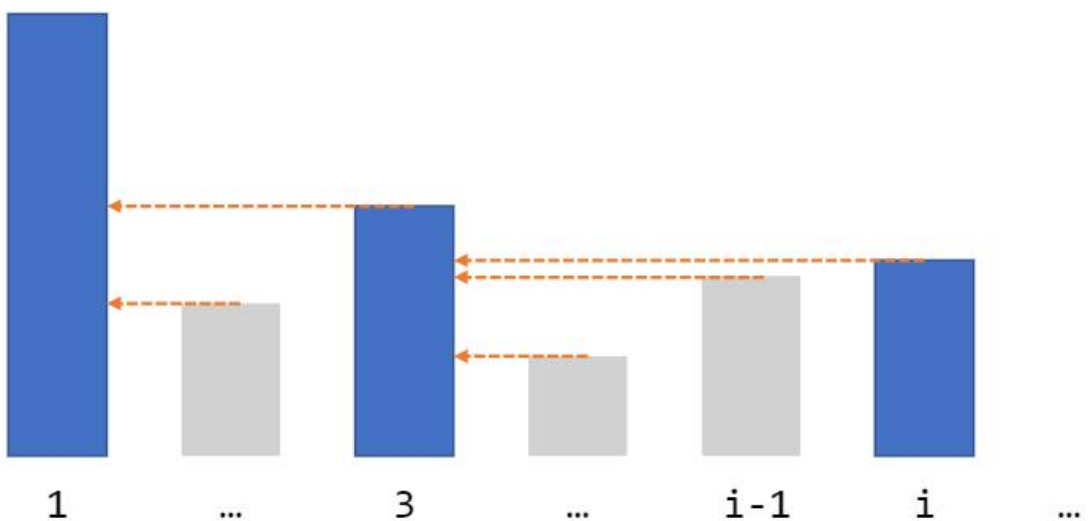


Problem 06B. 탑

i번 타워의 높이를 알고 있다.

(i+1)~N번 타워가 송신하게 될 레이저를 수신할 수 있는 후보 탑은 (1~i)번 탑 전부인가?

- 1~(i-1)번 타워들 중 i번 타워보다 높이가 낮은 타워는 앞으로 레이저를 수신할 수 있을까?



Problem 06B. 탑

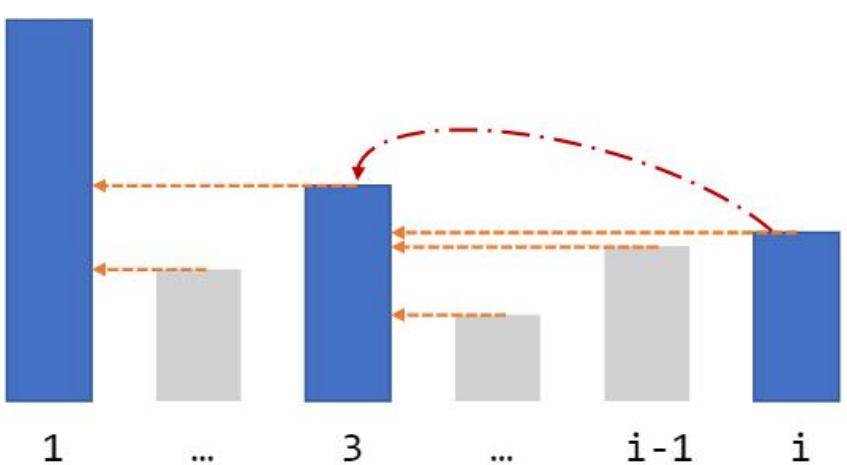
높이가 h인 타워가 i번째에 등장했다고 가정하자,

그렇다면 높이가 [1, h-1]인 모든 이전의 타워는 다른 레이저를 수신할 수 없다.

- 그러므로 고려 대상에서 제외하자.

```
// 현재 다른 타워의 신호를 수신할 가능성이 있는 타워들
Stack<Tower> touchableTowers = new Stack<>();
```

회색 타워가 저장되어 있지 않다고 가정해보자, i번째 타워의 레이저는 항상 바로 왼쪽의 타워가 수신한다.



Problem 06B. 탑

```
for(int i = 0 ; i < n ; i ++) {
   Tower t = towers[i];     // 각 타워 t에 대해 차례로 고려
   Tower target = null;     // 타워 t의 신호를 수신할 후보 ( 초기값 null )

   while(touchableTowers.isEmpty() == false
         && touchableTowers.peek().height < t.height){
      // t보다 높이가 낮은 타워들은 이후에도 수신 가능성이 없으므로 제거
      touchableTowers.pop();
   }

   // t 이상의 높이를 가진 타워가 남아있다면?
   if(touchableTowers.size() > 0){
      // t는 해당 스텍의 마지막 타워를 타겟으로 정하게 된다.
      target = touchableTowers.peek();
   }

   // 계산한 타겟 정보를 저장한다.
   t.setTargetTower(target);

   // t는 마지막에 등장했으므로 다른 타워의 신호를 수신할 수 있다. 등록.
   touchableTowers.push(t);
}
```

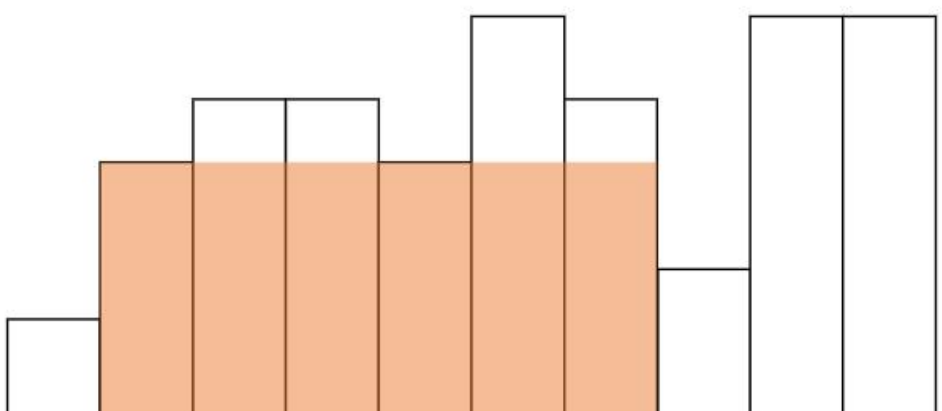
Problem 06C. 히스토그램

막대 히스토그램 내부에 존재할 수 있는 직사각형의 최대 넓이를 계산해야 한다.

- 직사각형은 히스토그램과 수직/수평 해야 한다.

앞의 문제처럼 단순히 높이를 가지고 쓸모 있는지/없는지를 판단하기는 힘들어 보인다.

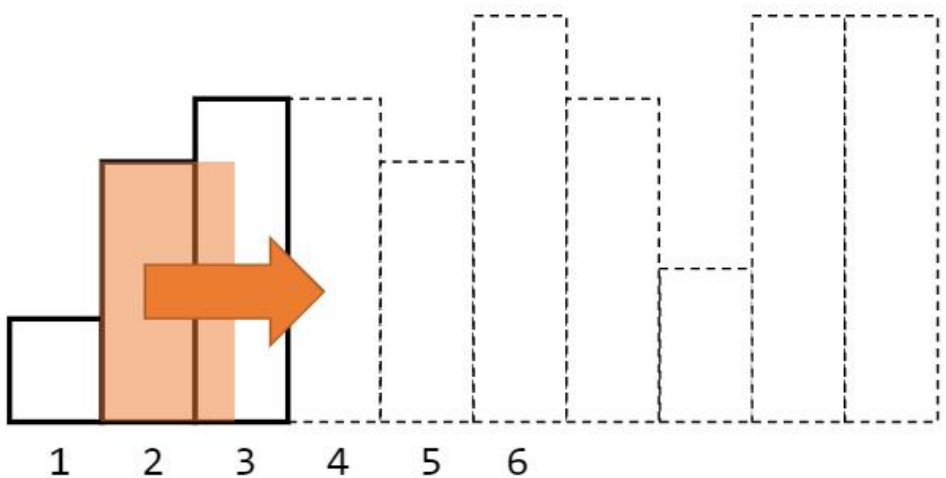
- 모든 범위를 조회하면 $O(N^2)$의 시간이 소요된다.



* 이 문제는 풀이가 정말 다양한 재미있는 문제다. 기억해두자.

Problem 06C. 히스토그램

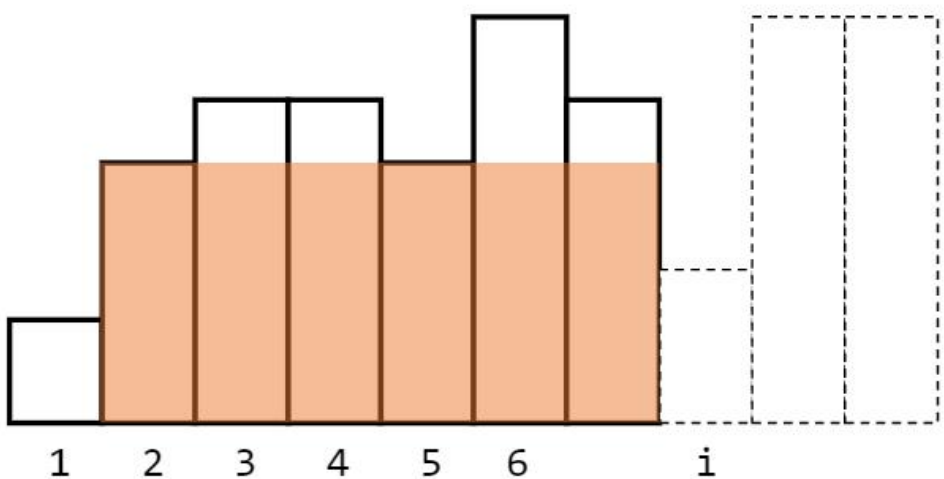
각 막대는 자기 자신의 높이를 유지하며 오른쪽으로 확장해 나간다고 생각해보자.
이 막대의 확장이 멈추는 시점은 언제인가?



Problem 06C. 히스토그램

각 막대는 최초로 자기 자신보다 높이가 낮은 막대가 등장할 때 확장을 멈춘다.

⇔ i번째 막대가 등장했을 때, 그 이전에 자기 자신보다 높이가 높던 막대는 확장을 멈춘다.

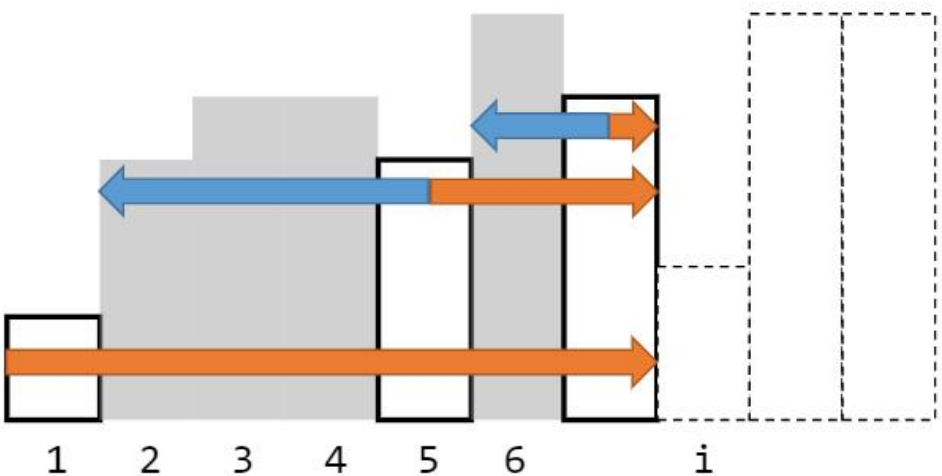


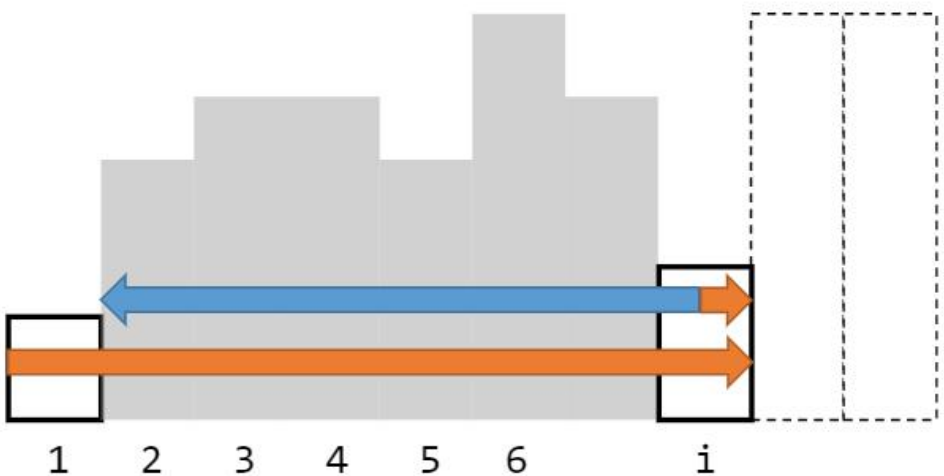
Problem 06C. 히스토그램

```
// 현재 우측으로 확장 가능성이 있는 히스토그램들
Stack<Histogram> continuedHistograms = new Stack<>();
```

계속 오른쪽으로 확장할 수 있는 막대들의 정보만을 저장했다고 가정하자.

- 이미 제거된 막대들(회색)은 최대로 확장한 넓이가 이미 결정되어 처리된 것이다.
- i번을 삽입하기 전, 스택에서 i번 막대보다 높이가 높은 막대들에 대한 최종 넓이 결정을 수행해야 한다.





```
// 구현상 편의를 위해 가장 왼쪽에 높이 0까지 가상의 히스토그램 추가
continuedHistograms.push( new Histogram(-1, 0) );
for(int i = 0 ; i < n + 1 ; i++) {
   // 왼쪽부터 오른쪽 히스토그램까지 차례로
   Histogram h;
   if( i < n ){
      h = histograms[i];
   }else{ // if ( i == n )
      // 구현상 편의를 위해 가장 오른쪽에 높이 0까지 가상의 히스토그램 추가
      h = new Histogram(n,0);
   }
   // 이전에 확장중이던 히스토그램들 중, h보다 높이가 높은 히스토그램들은
   // 더 이상 확장될 수 없다 => 최대 넓이가 결정된다.
   while ( continuedHistograms.size() > 1
         && continuedHistograms.peek().height >= h.height ){
      // 확장중이던 히스토그램
      Histogram lh = continuedHistograms.pop();

      // 그 이전의 히스토그램 (왼쪽 확장의 끝)
      Histogram bh = continuedHistograms.peek();

      // 현재 추가된 h의 바로 왼쪽까지 확장중이었다.
      long width = Math.abs(h.leftX - bh.rightX);
      long height = lh.height;
      long area =  width * height;

      // 최대 값 갱신
      answer = Math.max(answer, area);
   }
   // h를 새로이 추가한다
   continuedHistograms.push(h);
}
```

Queue

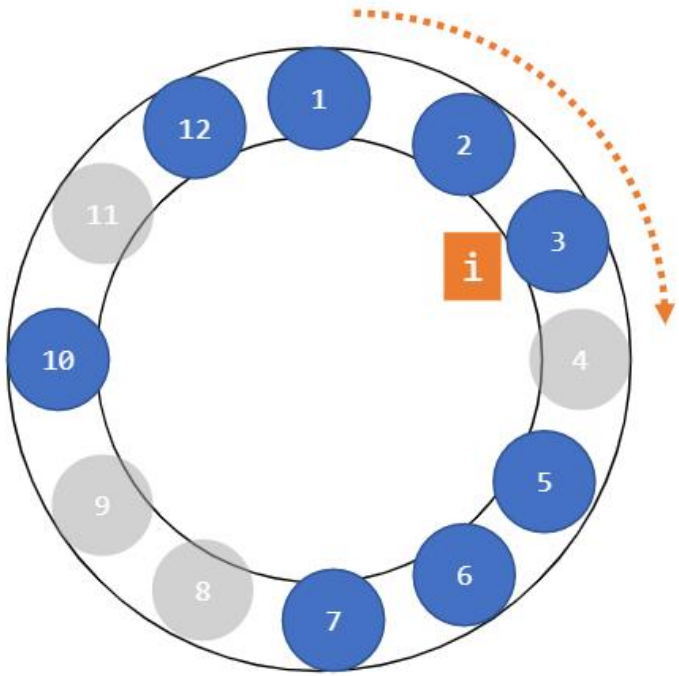
순차적으로 데이터를 저장하며, 가장 앞/뒤 원소의 접근이 가능한 자료구조.

- 가장 앞의 원소를 삭제할 수 있다. (dequeue, poll, pop)
- 가장 뒤에 원소를 추가할 수 있다. (enqueue, add, push)
- 각 추가/삭제에 소요되는 시간이 적고, 가변적으로 관리하기 효율적이다.
- 환형 배열을 사용한 방식과 Linked List를 사용한 방식의 구현이 있다.

Problem 06D. 조세퍼스 문제

‘게임에서 제외되지 않은 사람’만을 고려하며 순환하는 규칙을 가진다.

즉, 배열에서 한칸씩 순회하는 방식은 게임에서 이미 제외된 플레이어에 대한 고려를 하면 비효율적이다.



이미 게임에서 제외된 플레이어가 남아있지 않다면?

- 또한 현재 k명이 남아있다면, k%M번만 점프해도 된다.
  - M번마다 순환 할 테니까

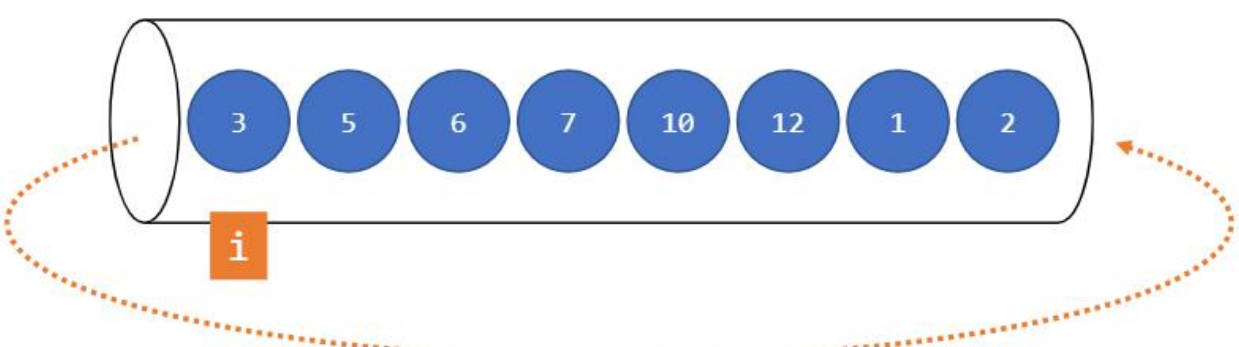
- 이 문제는 수학적으로 빠르게 푸는 방법도 존재한다. (어렵다)
- 단, 마지막에 제거되는 사람만을 계산할 수 있다.

Problem 06D. 조세퍼스 문제

각 플레이어들을 Queue에 차례로 저장해 두었다고 생각해보자.

큐의 Front에 해당하는 원소가 '현재 조회중인 원소'라고 하자.

```
// 아직 게임에서 제외되지 않는 플레이어들의 리스트
Queue<Player> playerQueue = new LinkedList<>();
for(int i = 0 ; i < n; i+= 1){
   playerQueue.add(players[i]);
}
```



Problem 06D. 조세퍼스 문제

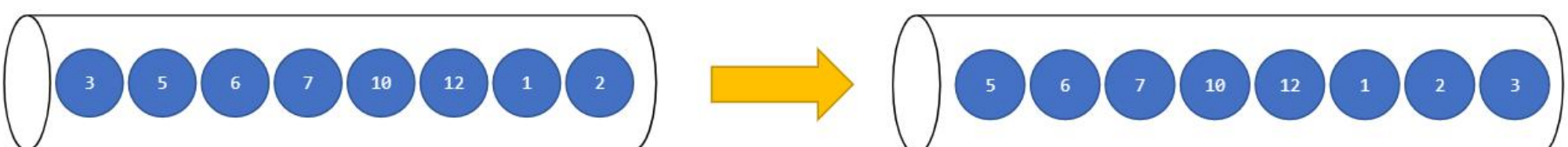
각 플레이어들을 Queue에 차례로 저장해 두었다고 생각해보자.

큐의 Front에 해당하는 원소가 '현재 조회중인 원소'라고 하자.

게임에서 제외해야 할 플레이어라면 Pop하면 된다.

그렇지 않다면, Pop한 후 다시 가장 뒤에 Push해주면 된다.

- '현재 살아남은 유저'에 대한 정보만을 큐에 저장하며 관리할 수 있다.



Problem 06D. 조세퍼스 문제

이처럼, '고려 해야 할 대상만을 순서대로 누적, 사용'하는 방식에 효율적이다.

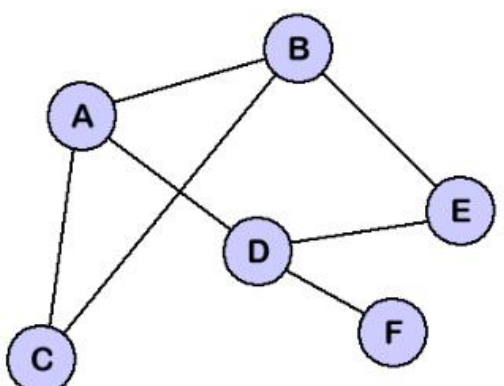
```
for(int i = 0 ; i < n ; i++){
   // (m-1)명의 사람을 건너뛴다.
   int jump = m % playerQueue.size();
   for(int j = 0; j<jump - 1; j+=1){
      Player p = playerQueue.poll();
      playerQueue.add(p);
   }

   // m번째 사람은 게임에서 제외한다.
   Player dead = playerQueue.poll();

   // 제외 리스트에 추가한다.
   deadPlayers.add(dead);
}
```

인접 리스트(Adjacency List)

그래프는 여러 정점 간의 1:1 관계를 표현한 모형이다.

- 대표적인 비선형 자료구조
- 하나 이상의 정점(Node/Vertex)을 가진다.
- 두 정점 사이를 잇는 간선(Edge)가 존재할 수 있다.
  - 간선은 단방향/양방향 모두 가능하다.
  - 단순한 관계 뿐만 아니라 가중치 등 부가 정보를 포함할 수 도 있다.

이런 데이터는 어떻게 저장해야 하나?

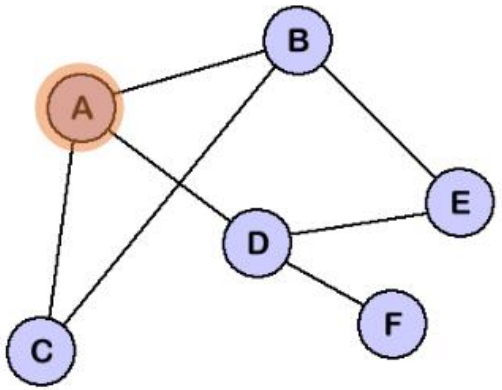


* 단순 2차원 배열으로 저장하는 Adjacency Matrix 기법도 있다.

인접 리스트(Adjacency List)

각 정점별로 나누어서 생각해보자.

- 한 정점에 대해 연결 된 모든 노드들을 나열해보자.



  - $Adj(A) = \{ B, C, D \}$
  - $Adj(B) = \{ A, C, E \}$
  - …

즉, 각 노드에 대해서 나누어 생각하면 연결 된 노드와 간선을 1차원적으로 나열할 수 있다.

각 간선의 연결 정보를 각 노드에 저장하고, 모든 노드의 정보를 하나의 리스트로 표현하면 결과적으로 전체 그래프를 리스트로 표현할 수 있다.

$$V = \{A, B, C, D, E, F\}$$
$$E = \{ Adj(A), Adj(B), \dots, Adj(F)\}$$
$$G(V, E) = Some\ Graph$$

Problem 06E. 폭탄 제거

두 개체 간의 1:1 관계는 그래프로서 표현하는게 편리하다.

이 문제의 그래프는 아래와 같은 특징이 있다.

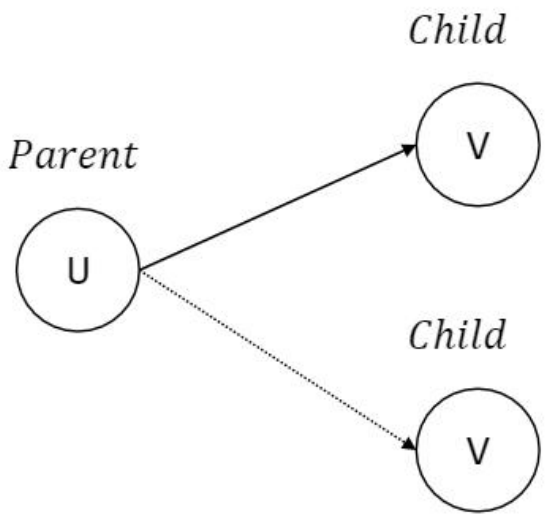
- 두 폭탄의 관계는 일방적이다 ⇔ 단방향 간선으로 표현된다.

U가 해체되면 V가 함께 폭발한다.
⇔ U, V를 모두 해체하기 위해서는 V를 먼저 해체해야 한다.

Problem 06E. 폭탄 제거

각 노드의 정보를 저장하기위한 클래스를 설계하자

```
class Bomb {
   public final int index;
   private int childCount; // 이 폭탄이 터졌을 때 함께 폭발하는 연쇄 폭탄의 수
   private ArrayList<Bomb> parentBombs; // 이 폭탄의 폭발을 유발할 수 있는 폭탄 리스트

   public void addParentBombs(Bomb parentBomb) {
      this.parentBombs.add(parentBomb);
      parentBomb.childCount += 1;
   }
   public ArrayList<Bomb> getParentBombs() {
      return this.parentBombs;
   }
   // 현재 이 폭탄이 제거되었을 때 폭발할 수 있는 폭탄의 수
   public int getChildCount() {
      return childCount;
   }
   public void remove() {
      // 이 폭탄을 제거한다.
      // 그러므로 부모 폭탄들은 연쇄 폭탄 하나가 사라진 꼴이 된다.
      for (int i = 0; i < parentBombs.size(); i += 1) {
         Bomb p = parentBombs.get(i);
         p.childCount -= 1;
      }
   }
}
```



각 노드(폭탄)은 제거될 때 부모의 childCount를 낮춘다.

childCount가 0인 폭탄은 제거해도 된다.

Problem 06E. 폭탄 제거

제거 가능한 노드를 Queue에 추가하고, 반대로 큐에 있는 폭탄들을 차례로 제거해 나간다.

```
// 현재 제거 완료된 폭탄들의 목록
ArrayList<Bomb> removedList = new ArrayList<>();

// 제거해도 전혀 이상이 없는 폭탄들의 목록
Queue<Bomb> removableBombs = new LinkedList<>();

// 현재 연쇄 폭탄이 없는 폭탄들은 제거 가능한 목록에 추가한다.
for(int i = 0 ; i < n; i+=1 ){
   if(bombs[i].getChildCount() == 0){
      removableBombs.add(bombs[i]);
   }
}
```

Problem 06E. 폭탄 제거

```
// 제거 가능한 폭탄이 남아있는 동안 반복한다.
while(removableBombs.isEmpty() == false){
   // 현재 연쇄폭탄이 없는 폭탄 b를 제거하고
   Bomb b = removableBombs.poll();

   // 부모 폭탄들에 대해 연쇄폭탄 정보를 갱신한다
   b.remove();

   // 그리고 제거 완료 목록에 추가한다.
   removedList.add(b);

   // 이 폭탄의 부모 폭탄들 중, 연쇄 폭탄이 모두 제거된 폭탄들은
   // 제거 가능하다. 그러므로 큐에 추가한다.
   ArrayList<Bomb> parents = b.getParentBombs();
   for(int i = 0 ; i < parents.size();  i+= 1){
      Bomb p = parents.get(i);
      if(p.getChildCount() == 0){
         removableBombs.add(p);
      }
   }
}

// 모든 n개의 폭탄이 제거 완료되었다면 true, else false
if(removedList.size() == n){
   return true;
}else{
   return false;
}
```

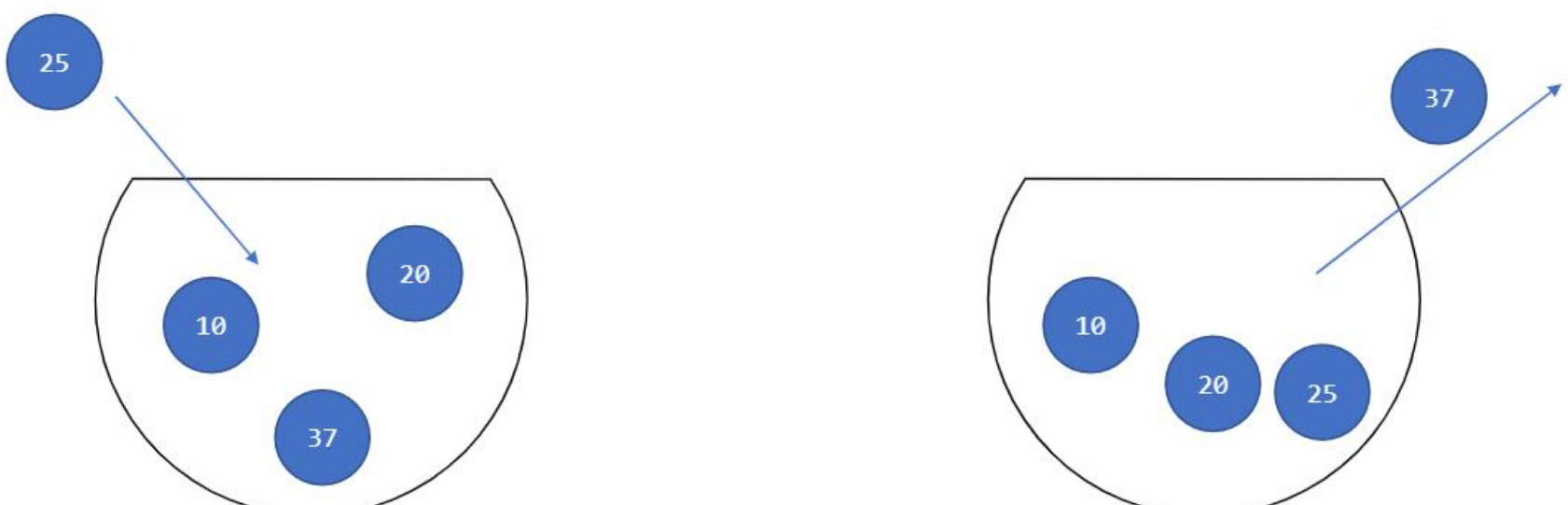
제거 가능한 폭탄은 차례로 제거 해 나간다.

그리고 해당 폭탄이 제거 됨으로써 영향을 받는 노드들에 대한 갱신 작업을 처리해준다.

Priority Queue (Heap)

우선순위 큐(Priority Queue)는 여러 데이터들 중 가장 우선순위가 높은 데이터에 대한 빠른 갱신과 접근이 필요할 때 사용한다.

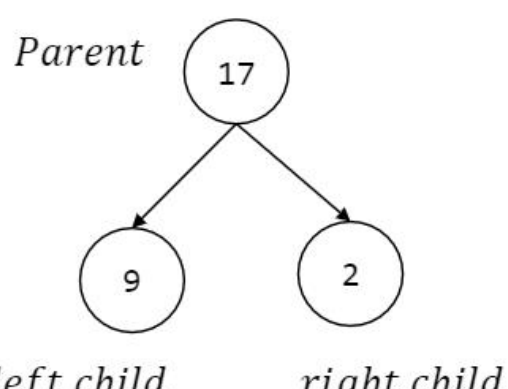
- 비교 가능한(Comparable) 데이터는 자유롭게 추가할 수 있다.
- 접근/삭제가 가능한 원소는 '저장된 전체 데이터들 중'에 가장 우선순위가 높은 데이터다.
- 일반적으로 Heap구조로 구현하며, 배열과 Linked List로도 Heap을 구현 할 수 있다.
- N개의 데이터를 차례로 삽입한 후, 차례로 모두 Pop하면 정렬된 상태로 꺼내 올 수 있다.

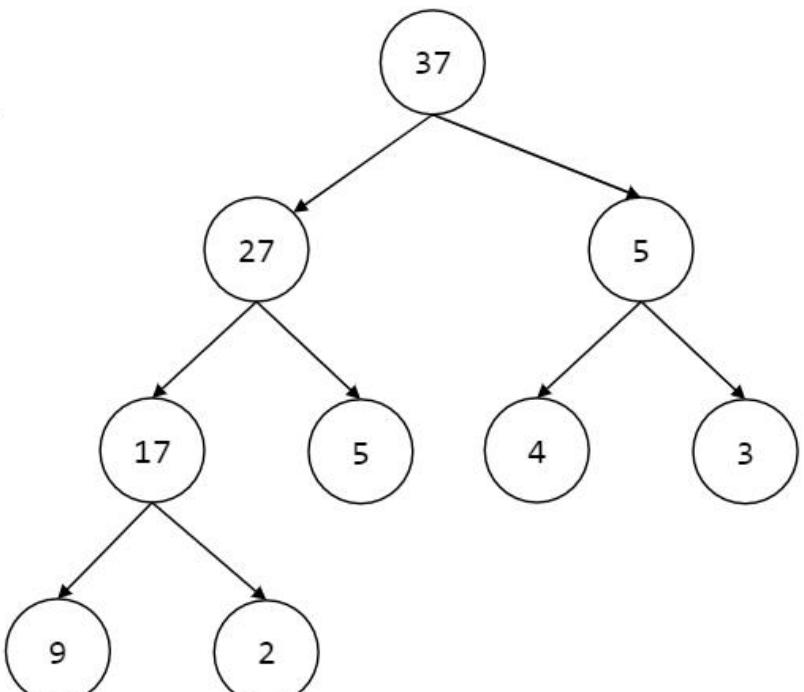


Priority Queue (Heap)

Heap은 완전 이진 트리(Complete Binary Tree)구조로 데이터를 관리하며
각 노드가 아래의 조건을 만족하면 된다.

- 부모 노드는 양 자식 노드보다 높은 우선순위를 가진다.





Problem 06F. 폭탄 제거 순서 정하기

앞 문제와 같이 해결할 수 있다. 단순히 큐를 우선순위 큐로 변경하면 된다.

```
// poll시에 제거 가능한 폭탄들 중 가장 인덱스가 작은 번호가 반환될 수 있도록
// PriorityQueue를 사용한다. 폭탄들의 우선순의는 compareTo 메소드로 정의했다.
PriorityQueue<Bomb> removableBombs = new PriorityQueue<>();
```

우선순위 큐는 항상 pop시에 우선순위가 높은 원소를 peek()으로 돌려준다.

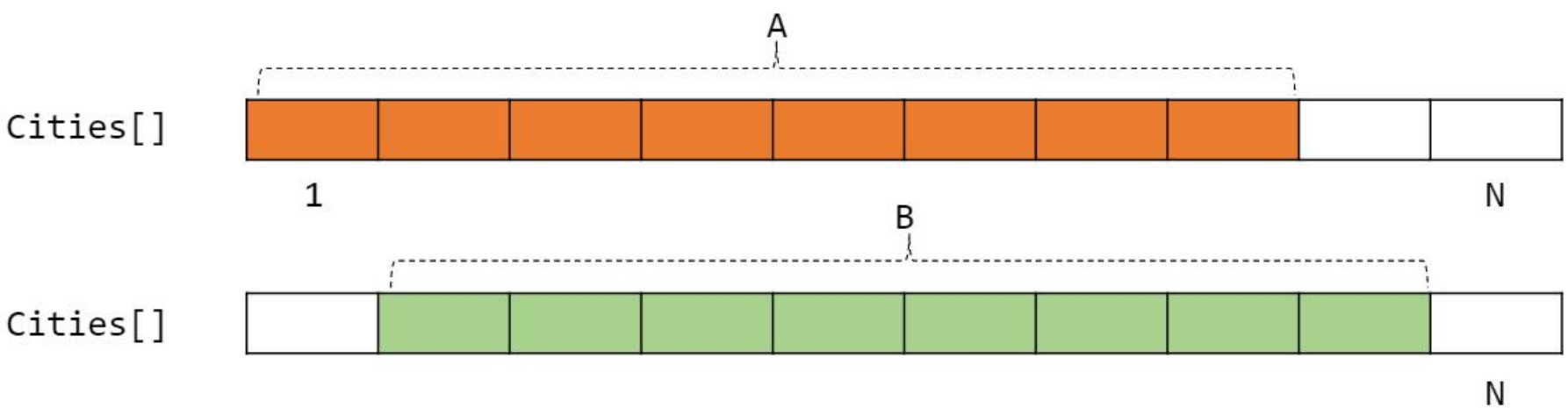
- 저장된 원소들 중 가장 우선순위가 높은 원소가 먼저 나오게 된다.
- ⇔ 제거 가능한 폭탄들 중 인덱스가 빠른 원소가 먼저 나오게 된다.

```
@Override
public int compareTo(Bomb o) {
    // 인덱스가 작을수록 priorityQueue에서 먼저 poll 되도록
    // 우선순위를 정의한다.
    return this.index - o.index;
}
```

Problem 06G. 불안정 지역

'연속한 K개의 …' => 슬라이딩 윈도우?

But, 원소가 하나씩 추가/제거됨에 따라서 이에 대한 갱신이 쉽지 않다.
단순한 연산으로 해당 원소들의 변화에 대한 계산이 쉽지 않기 때문.

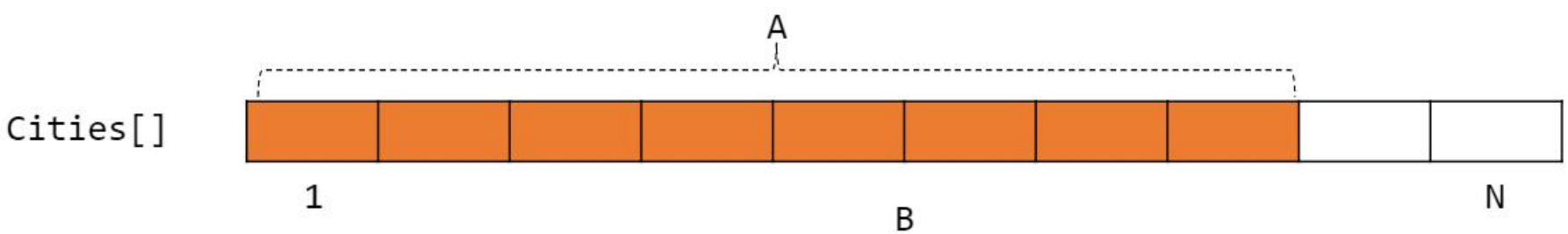


Problem 06G. 불안정 지역

필요한 정보 : 해당 영역에 속하는 원소들 중 최대/최소 값

해당 영역에 속하지 않는 숫자의 정보는 필요하지 않다.

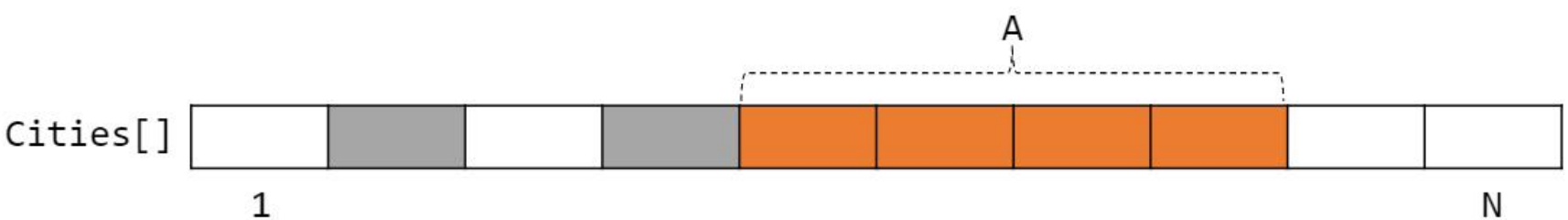
But, 그렇다면 꼭 항상 삭제해야 할 필요는 있을까?



Problem 06G. 불안정 지역

우리가 정답을 구해야 할 관심 영역은 A라고 가정하자.

불필요한 범위의 두 원소가 해당 범위의 최대/최소값이 아니라면,
해당 원소들의 존재 여부가 최대/최소값 계산 결과에 영향을 미치는가?



Problem 06G. 불안정 지역

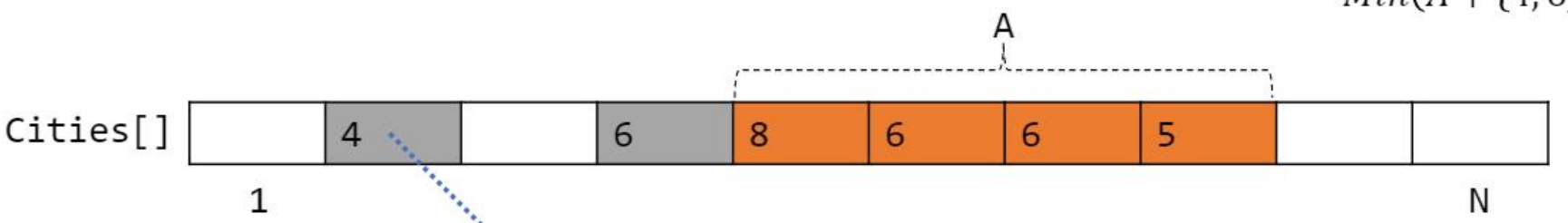
Lazy Update를 하자!

- 지금 당장에 꼭 필요한 업데이트가 아니라면 미루어 두자
- 결과 계산에 꼭 필요한 업데이트만 지금 하자

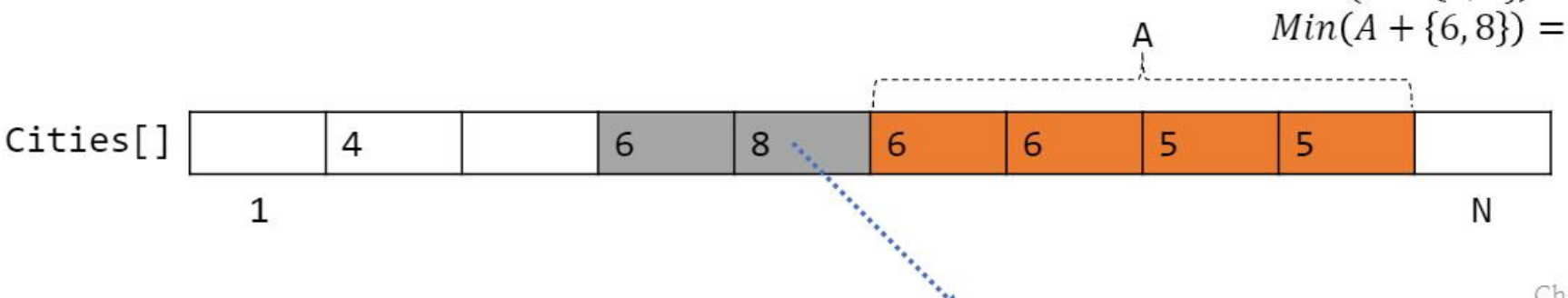
불필요한 숫자가 최대/최소가 된다면 그 때 제거하자.

- 최대/최소가 된다면 우선순위 큐로 쉽게 알 수 있을 테니까

$Max(A + \{4,6\}) = Max(A)$
$Min(A + \{4,6\}) \neq Min(A)$



$Max(A + \{6,8\}) \neq Max(A)$
$Min(A + \{6,8\}) = Min(A)$



Problem 06G. 불안정 지역

우선순위 큐에 '현재 고려해야 할 범위의 모든 원소'는 포함시키되,
'범위에 속하지는 않지만 결과에 영향을 주지 않는 원소'는 일단 제거하지 말자.

이후 각 우선순위 큐의 Top(Peek)이 범위 밖의 원소라면 그 때 제거해주면 된다.

```
// 소득이 가장 작은 도시부터 pop되는 우선순위 큐
PriorityQueue<City> rangeMinimum = new PriorityQueue<>();

// 소득이 가장 높은 도시부터 pop되는 우선순위 큐
PriorityQueue<City> rangeMaximum = new PriorityQueue<>(Collections.reverseOrder());
```

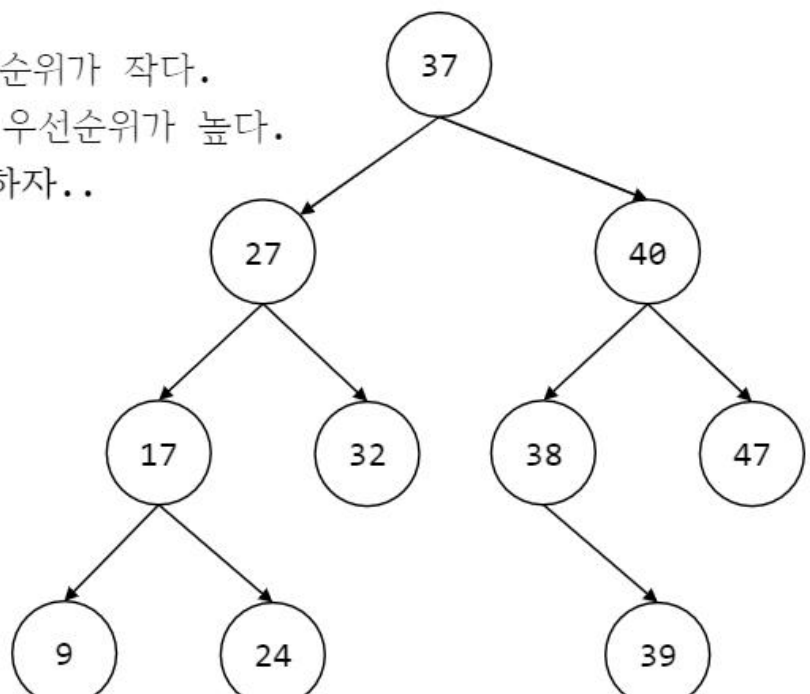
Problem 06G. 불안정 지역

```
for(int i = 0 ; i < n ; i+=1){
   City c = cities[i]; // 각 도시 c에 대하여 차례로

   // c를 오른쪽 끝으로 하는 k개의 연속된 도시에 대해
   int rightEnd = i;
   int leftEnd = rightEnd - k + 1;

   // c를 각 우선순위 큐에 추가한다
   rangeMaximum.add(c);
   rangeMinimum.add(c);
   // 최대 혹은 최소 소득을 가진 도시가 검사 범위 밖에 있다면
   // 불필요한 정보이므로 모두 pop하여 제거한다.
   while(rangeMaximum.isEmpty() == false  && rangeMaximum.peek().index < leftEnd){
      rangeMaximum.poll();
   }
   while(rangeMinimum.isEmpty() == false  &&  rangeMinimum.peek().income < leftEnd){
      rangeMinimum.poll();
   }
   // 검사 범위 내에 존재하는 최대/최소 소득간의 차이를 계산한다
   int minIncome = rangeMinimum.peek().income;
   int maxIncome = rangeMaximum.peek().income;
   int diff = maxIncome - minIncome;
   // 최대의 차이를 갱신한다
   answer = Math.max(answer, diff);
}
```

Map & Set (Binary Search Tree)

Map과 Set구조는 Comparable한 데이터를 빠르게 추가/삭제/접근할 수 있는 자료구조다.

- Binary Search Tree구조를 사용하여 구현된다.
- 원소의 추가/삭제/접근에 최대 $O(N)$의 시간이 소요된다.
  - Balancing 처리를 통해 $O(\log_2 N)$로 줄일 수 있다.
- 부모 노드를 기준으로 아래의 두 조건을 만족한다.
  - 왼쪽 자식 노드와 왼쪽 모든 자손 노드는 부모 노드보다 우선순위가 작다.
  - 오른쪽 자식 노드와 오른쪽 모든 자손 노드는 부모 노드보다 우선순위가 높다.
- 구현이 복잡하고, 구현에 따른 성능 차이가 크므로 구현체를 사용하자..



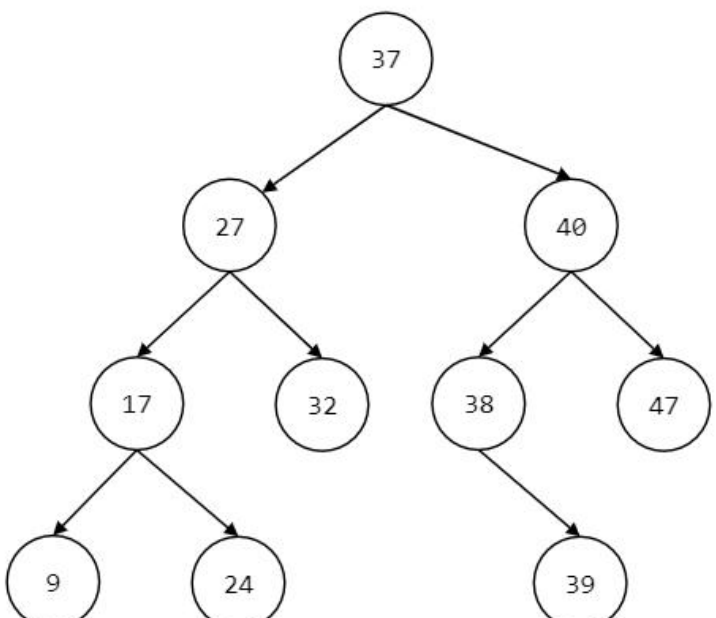
일반적인 구현체에서 중복처리는 별도로 해주지 않는다!

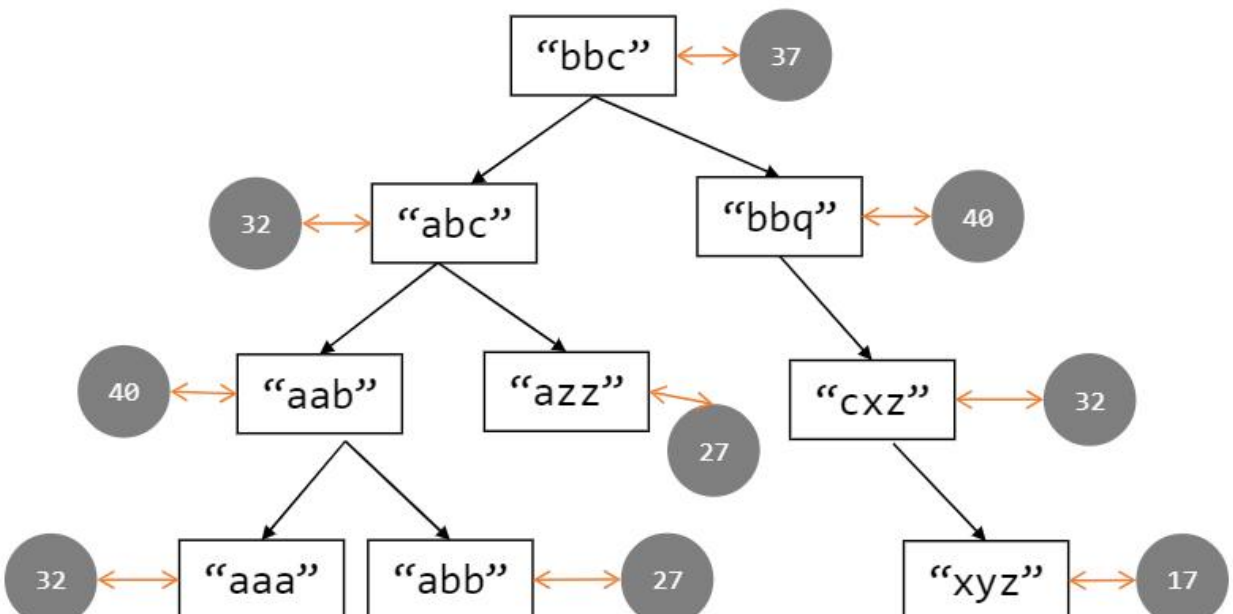
- 중복 Key에 대한 관리가 필요하다면 Chaining을 하자

Set vs Map

결과적으로 많이 다르지 않다.

- Set은 데이터의 우선순위를 기준으로 관리되는 일반적인 BST다.
- Map은 데이터를 관리하되 우선순위를 별도의 Key데이터로 관리한다.
  - 탐색만을 위한 별도의 key라는 데이터가 추가되었다고 생각하자.





Set – Basic Methods (Java/C++)

같은 구현체이지만 언어에 따라서 용법과 명칭이 다르다.

```java
public static void test(){
    TreeSet<Integer> mySet = new TreeSet<>();

    mySet.add(5);
    mySet.add(6);
    mySet.remove(6);

    if(mySet.contains(5)){
        //exist
    }else{
        //no such element
    }

}
```

```cpp
int main(){
    set<int> mySet;

    mySet.insert( 5 );
    mySet.insert( 6 );
    mySet.erase( 6 );

    if(mySet.count(5) > 0){
        //exist
    }else{
        //no such element
    }

    return 0;
}
```

Map – Basic Methods (Java/C++)

같은 구현체이지만 언어에 따라서 용법과 명칭이 다르다.

```java
public static void test(){
   TreeMap<String, Integer> myMap = new TreeMap<>();

   myMap.put("HI", 1);
   myMap.put("BYE", 2);
   myMap.remove("BYE");

   if(myMap.containsKey("HI")){
      //exist
   }else{
      //no such element
   }

}
```

```cpp
int main(){
    map<string, int> mySet;

    mySet.insert( make_pair("HI", 1) );
    mySet.insert( make_pair("BYE", 2) );
    mySet.erase( "BYE" );

    if(mySet.count("HI") > 0){
        //exist
    }else{
        //no such element
    }

    return 0;
}
```

Problem 06H. 중복 제거하기

Set을 한 번 사용해보는 목적의 튜토리얼 문제.

- 갱신, 검사가 잦을 때에는 Set, Map을 사용하자.

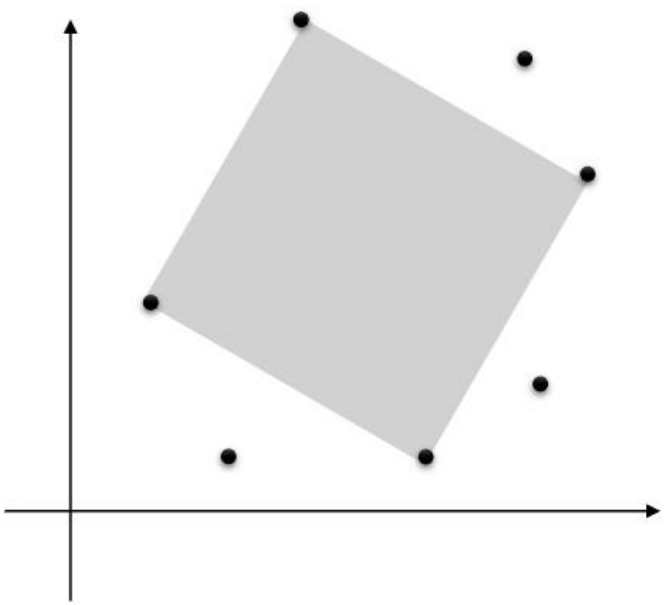
```
public static void main(String[] args) throws Exception {
   int N = scanner.nextInt();

   TreeSet<Integer> integers = new TreeSet<>();

   for(int i = 0 ; i < N ; i++){
      // integers := 이전까지 등장한 모든 정수를 저장한 집합
      int X = scanner.nextInt();

      if(integers.contains(X)){
         // X와 같은 정수가 입력된 적 있다면
         writer.write("DUPLICATED\n");
      }else{
         // X와 같은 정수가 입력된 적 없다면 추가한다.
         integers.add(X);
         writer.write("OK\n");
      }
   }

   writer.flush();
   writer.close();
}
```

Problem 06I. 정사각형

주어진 N개의 점 중 네 개를 골라서 '정사각형'을 만들어야 한다.

모든 점을 다 반복문으로 골라 볼 필요가 있을까?

- 두/세/네 카드 문제를 떠올려보자
- 점의 좌표를 알면 존재 여부는 Set으로 빠르게 검사할 수 있다.



```java
// 모든 점을 Set에 저장한다
TreeSet<Point2D> pSet = new TreeSet<>();
for (int i = 0; i < n; i += 1) {
   pSet.add(points[i]);
}
```

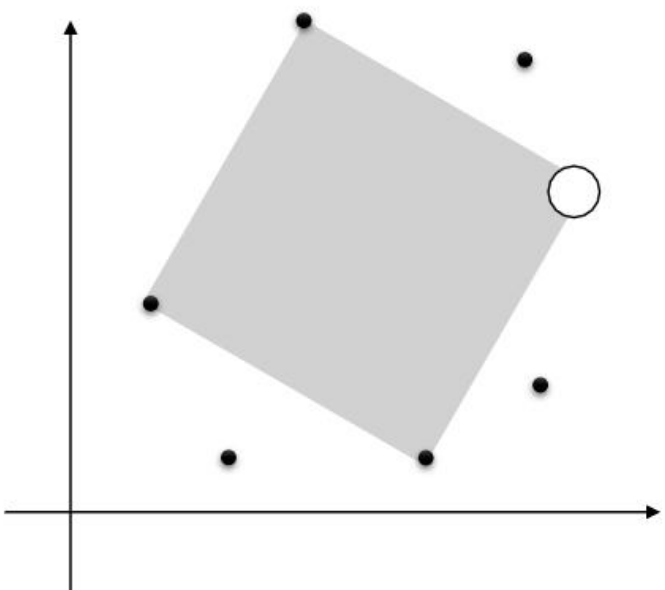
```java
@Override
public int compareTo(Point2D other) {
   // 각 좌표의 우선순위를 비교하기 위한 비교 연산자

   // x좌표가 다르다면 x좌표를 기준으로 비교한다.
   if (this.x != other.x) {
      return this.x - other.x;
   }

   // x좌표가 같다면 y좌표를 기준으로 비교한다.
   return this.y - other.y;
}
```

Problem 06I. 정사각형

점 세 개를 반복문으로 결정하면 점 하나를 결정할 수 는 있다!

but, 시간도 느리고 계산 과정이 생각보다 번거롭다

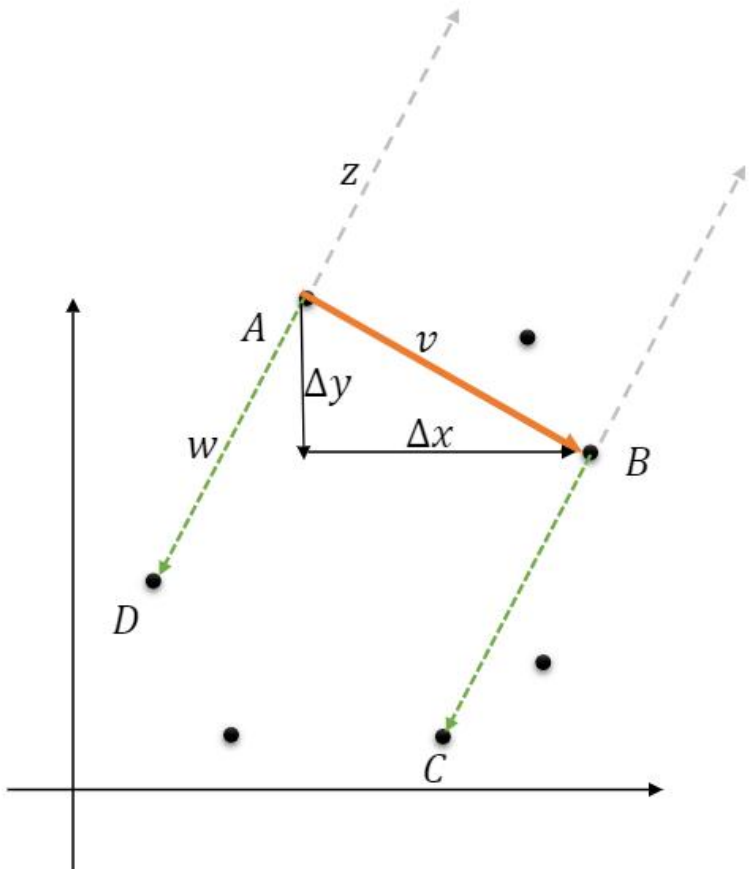
- $O(N^3)$



Problem 06I. 정사각형

'직사각형'이라면 문제가 어려워지지만, '정사각형'이기에 간단(?) 해진다.

인접한 두 개의 점 A, B를 결정하면 선분 $\overline{AB}$가 포함되는 정사각형은 두 개 뿐이다.



$$v = B - A =< \Delta x, \Delta y >$$

$$w =< -\Delta y, \Delta x >$$

$$z =< \Delta y, -\Delta y >$$

벡터의 성질에 의해 $v \perp w$ 이고 $v \perp z$이다.

또한 $\|v\| = \|w\| = \|z\|$ 이 성립한다.

∴ 두 점 A, B와 벡터 v로 P, Q를 결정할 수 있으며, 사각형 ABCD는 정사각형이다.

```
for (int i = 0; i < n; i += 1) {
   Point2D pa = points[i];
   for (int j = 0; j < n; j += 1) {
      Point2D pb = points[j];
      // 두 기준점 pa와 pb를 지정한다.
      // 선분 pa-pb가 정사각형의 한 변이라고 하자.

      // 두 점의 거리(선분의 길이)의 제곱은 정사각형의 넓이가 된다.
      long area = pa.getSquaredDistanceTo(pb);

      // 이미 구한 사각형보다 넓이가 작을 수 밖에 없다면 건너뛴다.
      if (area < answer) {
         continue;
      }

      // pa->pb방향의 x, y 좌표에 대한 거리를 구한다
      int dx = pb.x - pa.x;
      int dy = pb.y - pa.y;

      // 벡터 <dx, dy>를 90도로 회전시키면 <-dy, dx>가 된다.
      // pa와 pb에 벡터 <-dy, dx>를 각각 더해 정사각형을 구성하는 두 점
      // pd, pc를 계산할 수 있다.
      Point2D pd = new Point2D(pa.x - dy, pa.y + dx);
      Point2D pc = new Point2D(pb.x - dy, pb.y + dx);

      // pd, pc와 결정적이므로 이 점이 pSet에 존재하는 점인지 검사하면 된다.
      if(pSet.contains(pc) && pSet.contains(pd)){
         answer = Math.max(answer, area);
      }
   }
}
```

Problem 06J. 빈도수 세기

간단히 Map을 사용해보는 튜토리얼 문제

```
for(int i = 0 ; i < N; i+= 1){
   // frequencyMap := 이전에 입력된 정수들의 빈도수를 저장하고 있다.
   int X = scanner.nextInt();

   // 한 번도 등장한 적 없는 숫자라면
   if(frequencyMap.containsKey(X) == false){
      // 빈도수를 0으로 초기화한다.
      frequencyMap.put(X, 0);
   }

   // 현재 frequencyMap에 저장된 적 있는 숫자의 갯수를 저장한다.
   int c = frequencyMap.size();

   // X가 등장한 빈도수를 갱신하여 저장한다.
   int f = frequencyMap.get(X) + 1;
   frequencyMap.put(X, f);

   // 정보를 출력한다.
   writer.write(String.format("%d %d\n", c, f));
}
```

Problem 06K. 시장 추천하기

한 번 이상 등장한 후보의 이름과, 최대 득표수가 필요하다!

- 물론, for-each 반복문이나 map.keySets()같은 기능을 사용해도 되지만 연습 겸 직접 구현해보자

Problem 06K. 시장 추천하기

한 번 이상 등장한 후보의 이름과, 최대 득표수가 필요하다!

```
// 각 후보에 대해 득표수를 가산해나간다.
for (int i = 0; i < N; i += 1) {
   String name = scanner.next();

   if (frequencyMap.containsKey(name) == false) {
      // 1. 등장한적 있는 모든 이름은 이 코드를 지나간다.
      // 2. 모든 이름은 각각 한 번씩만 이 코드를 수행한다.
      existingNames.add(name);
      frequencyMap.put(name, 0);
   }

   int f = frequencyMap.get(name) + 1;
   frequencyMap.put(name, f);

   // 득표수를 갱신하며 최대 득표수도 갱신한다.
   maxFrequency = Math.max(maxFrequency, f);
}
```

Problem 06K. 시장 추천하기

모든 후보의 이름을 알고 찾고자 하는 후보들의 득표수도 알고 있다.

- Map에서 이름을 통해 후보의 득표수를 알아낼 수 있다.
- 득표수를 비교하여 일치하면 이름을 저장하자

```
// 가장 많은 득표수를 알고 있으므로,
// 모든 후보에 대해 이 득표수를 획득한 후보들을 winnerList에 저장한다.
ArrayList<String> winnersList = new ArrayList<>();
for (int i = 0; i < existingNames.size(); i += 1) {
   String name = existingNames.get(i);
   if (frequencyMap.get(name) == maxFrequency) {
      winnersList.add(name);
   }
}
```

Fin.

감사합니다.

* 이 PPT는 네이버에서 제공한 나눔글꼴과 아모레퍼시픽의 아리따부리 폰트를 사용하고 있습니다.