Datastructures

자료구조

- 컴퓨터에서 데이터를 효율적으로 탐색/조회할 수 있도록 저장하는 것을 다루는 분야
- 상황에 따라 적절한 자료구조를 채택해야 하는 필요성

- 선형(Linear) 자료구조: 스택, 큐, 덱, ...
- 비선형(Non-linear) 자료구조: 힙, 그래프, ...

자료구조

- 실제 자료구조를 구현하는 것은 2학년 자료구조 수업에서 진행
- 대부분의 자료구조는 이미 라이브러리로 구현돼 있음
- 어떻게 사용하는지, 문제에 어떻게 적용하는지를 위주로 설명

Abstract Data Type

- 실제 자료구조의 구현을 신경쓰지 않고, 자료구조의 기능만을 간략하게 설명
- 자료구조에 필요한 기능들을 써 두고, 이를 빠르게 수행할 수 있도록 하는 구현을 생각
- Stack, Queue, Dequeue ADT에 존재하는 하나하나의 계산은 $O(1)$에 작동

Stack ADT

- 한 쪽으로만 데이터를 삽입/삭제할 수 있는 자료구조
- First In, Last Out (FILO)
- `push(x)` : 스택의 가장 위에 `x`를 삽입
- `top()` : 스택의 가장 위 원소를 확인
- `pop()` : 스택의 가장 위에 존재하는 원소를 제거
- `isEmpty()` : 스택이 비어있다면 `true`, 그렇지 않다면 `false`

Queue ADT

- 한 쪽으로만 데이터를 삽입하고, 반대쪽으로만 삭제할 수 있는 자료구조
- First In, First Out (FIFO)
- `enqueue(x)` : 큐에 `x`를 삽입 (rear / back 쪽)
- `dequeue()` : 큐의 front쪽에서 원소를 제거
- `front()` : 큐의 front에 위치하는 원소를 확인
- `isEmpty()` : 큐가 비어있다면 `true`, 그렇지 않다면 `false`

Dequeue ADT

- Double-ended queue, 양 쪽으로 모두 삽입/삭제할 수 있는 자료구조
- `push_front(x) / push_back(x)` : 앞/뒤쪽에 x를 삽입
- `pop_front() / pop_back()` : 덱의 앞/뒤쪽에서 원소 제거
- `front() / back()` : 덱의 앞/뒤쪽에 있는 원소 확인
- `isEmpty()` : 덱이 비어있다면 `true`, 그렇지 않다면 `false`

Set ADT

- 순서와 관계없이 중복을 허용하지 않는 집합 자료구조
- `add(x)` : 집합에 x를 삽입
- `remove(x)` : 집합에서 x를 제거
- `union(s)` : 다른 집합 s와 합집합 연산
- `intersection(s)` : 다른 집합 s와 교집합 연산
- `find(x)` : 집합에 x가 존재하는지 확인
- `isEmpty()` : 집합이 비어있다면 `true`, 그렇지 않다면 `false`

Map ADT

- Key-Value 형식으로 매핑하는 자료구조, Key는 유일
- `put(key, value)` : key에 value를 매핑
- `remove(key)` : key에 해당하는 매핑을 제거
- `find(key)` : key가 존재하는지 확인
- `isEmpty()` : 매핑 테이블이 비어있다면 `true`, 그렇지 않다면 `false`

std::stack

```
stack<int> stk;
for(int i = 1; i <= 5; i++) stk.push(i);
cout << stk.size() << '\n';
while(!stk.empty()){
    int top = stk.top();
    cout << top << '\n';
    stk.pop();
}
```

괄호 BOJ 9012

- 주어진 괄호 문자열이 올바른 괄호 문자열인지 확인하는 알고리즘
- 스택을 사용하는 것으로 많이 알려져 있음

- 머릿속에 스택을 먼저 넣어두지 말고, 이 문제를 어떻게 해결할지부터 알아보자

괄호 BOJ 9012

- 주어진 괄호 문자열이)))인 경우에는 왜 올바른 괄호 문자열이 아닐까?
-)(인 경우에는?

괄호 BOJ 9012

- 주어진 괄호 문자열이)))인 경우에는 왜 올바른 괄호 문자열이 아닐까?
-)(인 경우에는?
-)와 대응하는 (가 없기 때문, 항상 여는 괄호와 닫는 괄호는 일대일 대응
- 일대일 대응이면서도 순서가 중요, 반드시 여는 괄호가 앞에 등장해야 함
- 괄호 쌍이 만들어졌다면, 해당 괄호 쌍에 사용된 괄호는 **다른 쌍에 영향을 주지 않음**
- **더 이상 필요없는 괄호 쌍은 없어져도 무방함**

괄호 BOJ 9012

- 순서대로 문자 하나씩 받으면서, 자료구조에 하나씩 저장
- 만약 자료구조에 아무 문자도 없는데)가 들어온 경우는 올바른 괄호 문자열이 아님
- 자료구조에)가 들어와야 하는데, (가 **가장 끝에 있는 경우**에는 (를 뽑아내면 됨
-)가 들어오는데)가 끝에 있는 경우, 올바른 괄호 문자열이 아님
- 머릿속의 자료구조를 생각해 보니, 한 쪽으로만 삽입과 삭제를 반복한다
- **스택을 사용해서 구현하자!**

괄호 BOJ 9012

```
string s; cin >> s;
stack<char> stk;
bool flag = true;
for(auto c : s){
    if (c == '(') stk.push(c);
    else {
        if (stk.empty() || stk.top() == ')') {
            flag = false;
            break;
        }
        else stk.pop();
    }
}
if (!stk.empty()) flag = false;
if (flag) cout << "YES" << '\n';
else cout << "NO" << '\n';
```

괄호 BOJ 9012

- 사실 스택이라는 자료구조를 활용하지 않고 더 쉽게 문제를 해결할 수 있다
- 괄호 쌍의 특성 상, 여는 괄호와 닫는 괄호의 개수가 같음
- (를 +1,)를 -1과 매핑해서 계산
- 모든 문자열을 처리한 뒤 계산값이 음수가 나오는 경우는 어떤 경우일까?

std::queue

```
queue<int> q;
for(int i = 1; i <= 5; i++) q.push(i);
cout << q.size() << '\n';
while(!q.empty()){
    int front = q.front();
    cout << front << '\n’;
    q.pop();
}
```

카드2 BOJ 2164

- 카드 덱을 활용하는 문제
- 가장 위에 있는 카드를 버린다
- 가장 위에 있는 카드를 가장 아래로 옮긴다
- 마지막에 남는 카드를 구하기
- $N \leq 500\,000$

카드2 BOJ 2164

- $O(N)$의 시간복잡도를 가지는 풀이를 찾아내야 한다
- 카드를 버리는 것, 옮기는 것 모두 $O(1)$에 해내면 시뮬레이션을 통해 풀 수 있다
- 위에서 카드를 버리고, 아래에서 카드를 새로 추가한다
- 자료구조 **큐의 형태**와 같다

카드2 BOJ 2164

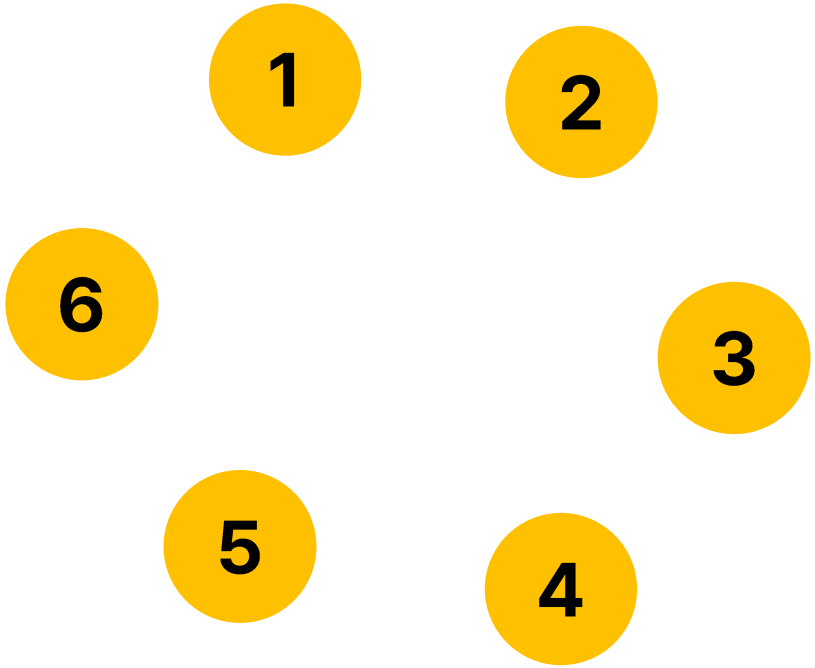
```
queue<int> q;
for(int i = 1; i <= N; i++) q.push(i);
while(q.size() != 1){
    q.pop();
    q.push(q.front());
    q.pop();
}
cout << q.front() << '\n';
```

요세푸스 문제 0 BOJ 11866

- 원형으로 앉아있는 N명의 사람들, K번째 사람이 차례로 제거된다
- 제거된 사람은 세지 않는다
- 제거되는 순서를 구해 보자

요세푸스 문제 0 BOJ 11866

- $N = 6, K = 4$
- **4 2 1 3 6 5**



요세푸스 문제 0 BOJ 11866

- 큐에 형태에 사람들을 저장하고, 앞에서 뽑고 뒤에 추가하는 과정은 무엇을 의미할까?
- $K-1$번 했다면, 큐의 가장 앞에 있는 사람은 K번째 사람
- 제거하고 돌리는 것을 반복하자

std::deque

```
deque<int> dq;
for(int i = 1; i <= 10; i++) dq.push_back(i);
cout << dq.size() << '\n';
for(int i = 0; i < 5; i++){
    int front = dq.front();
    cout << front << '\n';
    dq.pop_front();
}
for(int i = 0; i < 5; i++){
    int back = dq.back();
    cout << back << '\n';
    dq.pop_back();
}
```

AC BOJ 5430

- 정수 배열에 두 가지 연산을 할 수 있다
- 배열 전체를 뒤집기, 배열의 첫 번째 수를 버리기
- 수행한 뒤의 배열의 상태 출력하기

AC BOJ 5430

- 배열의 첫 번째 수는 뒤집으면 마지막 수가 되고, 그 반대의 경우도 마찬가지
- 매번 배열을 뒤집는 과정을 시뮬레이션한다면 시간복잡도가 어떻게 될까?

Priority Queue

- 큐와 비슷하게 동작하지만, 원소의 우선순위에 따라 front가 달라지는 자료구조
- C++에서는 기본적으로 최대 힙, Python의 경우 최소 힙
- 삽입, 삭제 $O(logN)$

```
priority_queue<int> pq;
for(int i = 1; i <= 5; i++) pq.push(i);
while (!pq.empty()) {
    cout << pq.top() << " " ;
    pq.pop();
}
// 5 4 3 2 1
```

최소 힙 BOJ 1927

- 배열에 자연수를 넣거나, 가장 작은 값을 제거하는 문제
- Naïve: $O(N^2)$
- $O(NlogN)$에 문제를 해결해야 함

최소 힙 BOJ 1927

- 배열에 자연수를 넣거나, 가장 작은 값을 제거하는 문제
- 우선순위 큐를 선언하고, 삽입/삭제를 진행하자
- C++에서는 최대 힙으로 구현돼있기 때문에, 비교 연산자를 설정해야 한다
- 또는 부호를 바꿔 우선순위 큐에 삽입할 수 있다

```
priority_queue<int, vector<int>, greater<>> pq;
for(int i = 1; i <= 5; i++) pq.push(i);
while (!pq.empty()) {
    cout << pq.top() << " " ;
    pq.pop();
}
// 1 2 3 4 5
```

std::map, set

- Key의 대소관계에 따라 Red-black Tree로 구현
- 삽입, 삭제, 검색에 $O(logN)$의 시간복잡도를 가진다

```
map<int, string> mp;
mp[1] = "apple";
mp[2] = "banana";
mp[3] = "candy";
cout << mp[1] << '\n';
if (mp.find(4) == mp.end())
    cout << "key 4 Not found" << '\n';
```

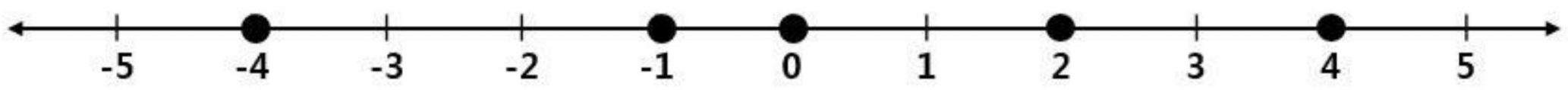
```
set<int> s;
for(int i = 1; i <= 10; i++) s.insert(i);
s.erase(10);
if (s.find(10) == s.end())
    cout << "10 not exists in set" << '\n';
```

std::unordered_map, unordered_set

- 해시 매핑을 통한 삽입, 삭제, 조회가 평균 $O(1)$
- 해시 충돌 발생 시 $O(N)$

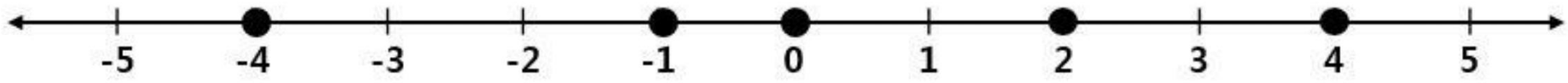
Three Dots BOJ 13423

- 수직선상의 많은 점들 중, 세 점을 골라 연속한 두 점의 간격이 같은 세 점의 개수
- $N \leq 1\,000$
- $|x_i| \leq 10^9$



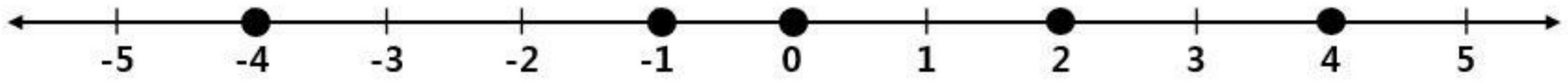
Three Dots BOJ 13423

- 수직선상의 많은 점들 중, 세 점을 골라 연속한 두 점의 간격이 같은 세 점의 개수
- Naïve: 세 점에 대해 브루트포스, $O(N^3)$
- $N \leq 1\,000$이기 때문에 시간초과



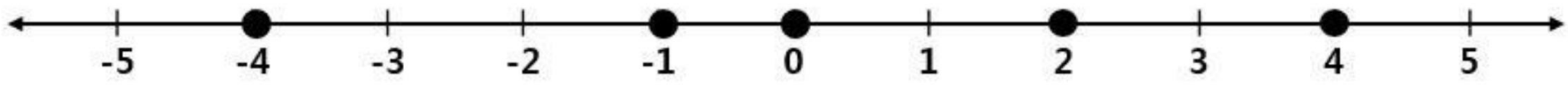
Three Dots BOJ 13423

- 수직선상의 많은 점들 중, 세 점을 골라 연속한 두 점의 간격이 같은 세 점의 개수
- 시간복잡도 상, $O(N^2)$에 문제를 해결해야 함
- 점 두개를 잡고, 나머지 한 개의 점이 존재하는지 여부를 빠르게 찾으면 된다



Three Dots BOJ 13423

- 수직선상의 많은 점들 중, 세 점을 골라 연속한 두 점의 간격이 같은 세 점의 개수
- 모든 점을 set에 넣어두고, 두 점을 고른다
- 세 번째 점이 존재해야 하는 좌표가 set에 존재한다면 정답을 증가시킨다
- hash-set의 경우 $O(N^2)$, tree-set의 경우 $O(N^2 logN)$



Three Dots BOJ 13423

- 수직선상의 많은 점들 중, 세 점을 골라 연속한 두 점의 간격이 같은 세 점의 개수

```
for(int i = 0; i < N; i++){
    cin >> arr[i];
    s.insert(arr[i]);
}
for(int i = 0; i < N; i++){
    for(int j = i+1; j < N; j++){
        ll diff = abs(arr[i] - arr[j]);
        if (s.find(max(arr[i], arr[j]) + diff) != s.end()) ans++;
    }
}
```

자료구조를 활용하는 문제들

- 문제를 보고 바로 스택! 큐!를 떠올리는 것에는 많은 훈련이 필요하다
- 문제를 해결해나가면서 어떤 특징을 가지는지 파악
- 문제를 해결하기 위한 자료구조가 무엇인지는 모르지만, 어떤 연산이 필요한지 확인
- 해당 연산을 지원하는 자료구조를 적절히 활용

오큰수 BOJ 11866

- 수열에서, 각 원소마다 오큰수를 계산하자
- 오큰수란, 원소 A_i보다 오른쪽에 있으면서 A_i보다 큰 수 중에서 가장 왼쪽에 있는 수
- $[9, 1, 5, 3, 6, 2]$의 각 원소의 오큰수는 $[-1, 5, 6, 6, -1, -1]$

오큰수 BOJ 11866

- 오큰수란, 원소 A_i보다 오른쪽에 있으면서 A_i보다 큰 수 중에서 가장 왼쪽에 있는 수
- 간단한 계산 방법은, 각 원소마다 오른쪽으로 향하면서 큰 수가 나오면 그만두는 방법
- 해당 방법의 시간복잡도는 얼마나 될까?
- 최악의 경우는 배열이 어떻게 주어졌을 때일까?

오큰수 BOJ 11866

- 오큰수란, 원소 A_i보다 오른쪽에 있으면서 A_i보다 큰 수 중에서 가장 왼쪽에 있는 수
- 배열의 성질을 조금 더 파악해 보자
- 문제를 풀 때에는 어떤 자료구조를 사용해야 하는지 알 수 없음
- 머릿속의 자료구조가 어떤 특징을 가지는지 생각하자

오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$
- 자신보다 큰 수를 만나는 순간 오큰수가 **확정**된다
- 더 큰 수가 나중에 오더라도 신경쓸 필요 없음
- 자신보다 작은 수가 온다면, 오큰수가 확정되지 않았으니 아직 지우면 안 됨

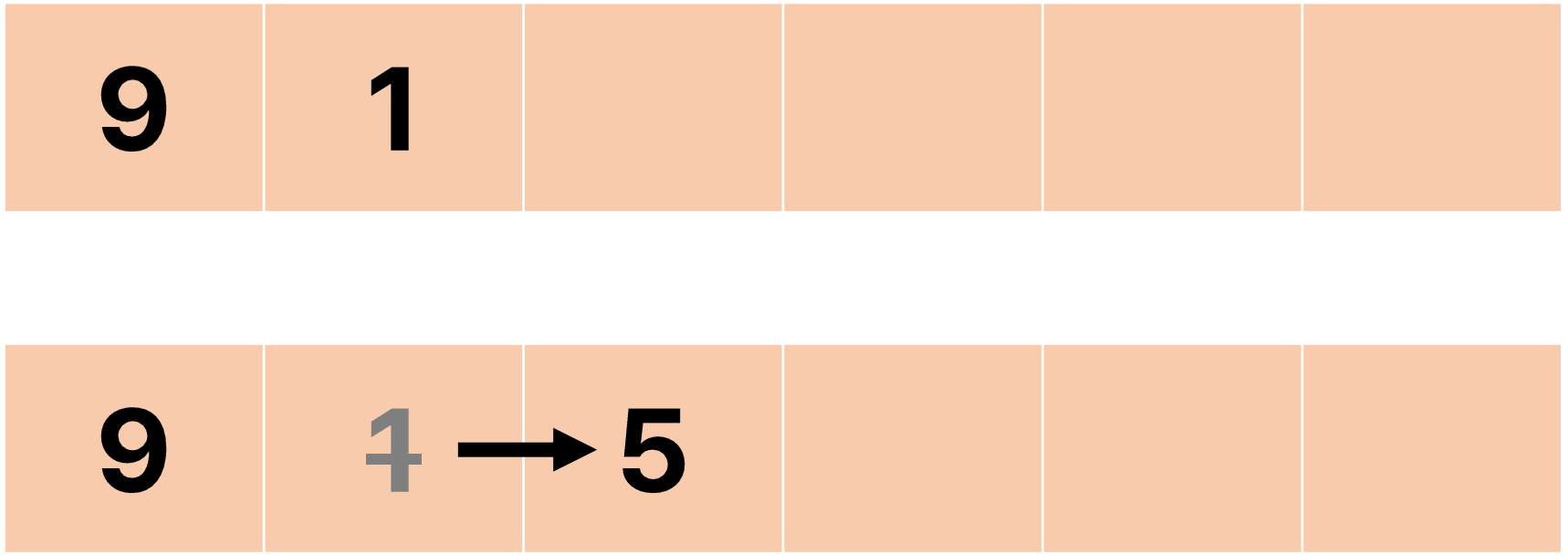
오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$
- 자신보다 큰 수를 만나는 순간 오큰수가 **확정**된다
- 더 큰 수가 나중에 오더라도 신경쓸 필요 없음
- 자신보다 작은 수가 온다면, 오큰수가 확정되지 않았으니 아직 지우면 안 됨

- 확정된 수는 자료구조에서 삭제해도 된다

오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$, 오큰수 배열 $[-1, 5, -1, -1, -1, -1]$



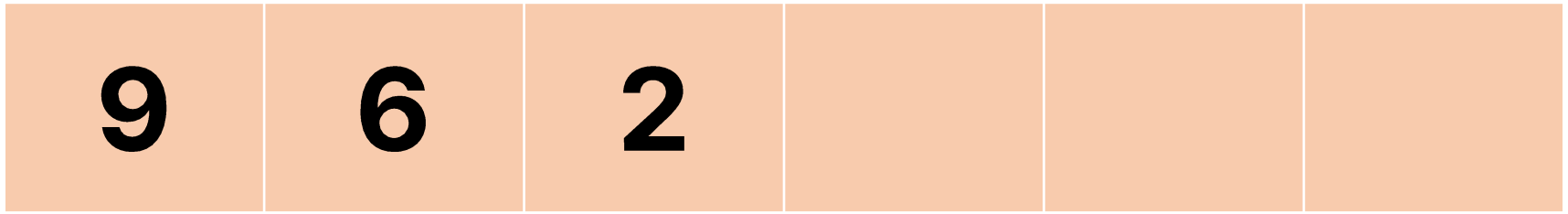
오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$, 오큰수 배열 $[-1, 5, 6, 6, -1, -1]$

9	5	3			
9	5	~~3~~ → 6			
9	~~5~~ → 6				

오큰수 BOJ 11866

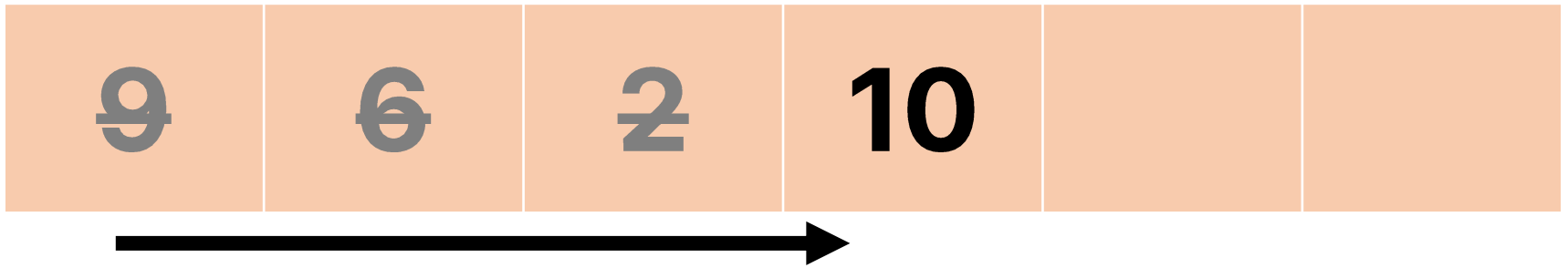
- $[9, 1, 5, 3, 6, 2]$, 오큰수 배열 $[-1, 5, 6, 6, -1, -1]$



- 만약 이 상태에서 10이 들어온다면?

오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$, 오큰수 배열 $[-1, 5, 6, 6, -1, -1]$



- 만약 이 상태에서 10이 들어온다면?
- 모든 앞의 수보다 크므로 정해지지 않았던 수들의 오큰수가 한 번에 정해짐

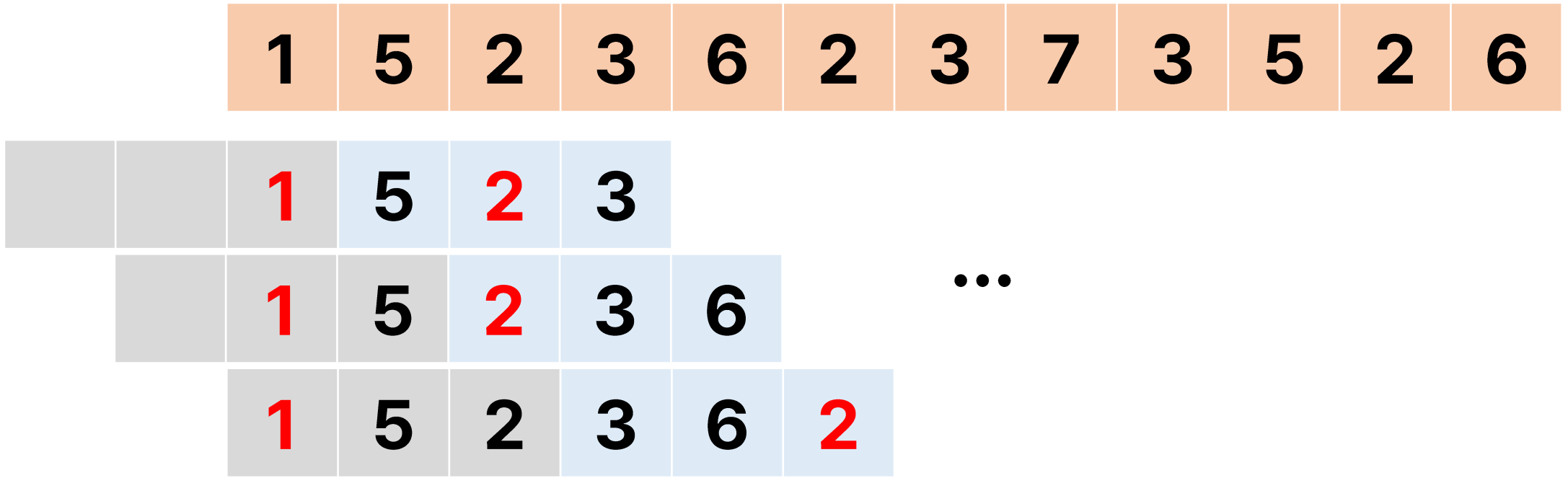
오큰수 BOJ 11866

- $[9, 1, 5, 3, 6, 2]$, 오큰수 배열 $[-1, 5, 6, 6, -1, -1]$
- 스택을 명시적으로 사용하지는 않았지만, 한 쪽으로만 삽입/삭제가 이루어짐

- 스택을 활용하면 빠르게 풀 수 있음, 한 쪽 삽입/삭제가 $O(1)$이므로
- 총 시간복잡도는 $O(N)$

최솟값 찾기 BOJ 11003

- N개의 수가 들어있는 배열에, 길이 L의 부분 수열에 대해 최솟값들을 구하는 문제



최솟값 찾기 BOJ 11003

- N개의 수가 들어있는 배열에, 길이 L의 부분 수열에 대해 최솟값들을 구하는 문제

1	5	2	3	6	2	3	7	3	5	2	6

- Naïve: $O(NL)$
- $L \leq N \leq 5\ 000\ 000$

최솟값 찾기 BOJ 11003

- 앞에서 뒤로 훑어나가면서 몇가지 성질을 파악해 보자

1	5	2	3	6	2	3	7	3	5	2	6

- 새로 들어오는 수가 더 크거나 같다면, 이후에 사용될 여지가 있다
- 길이가 지난 수들은 기억할 필요가 없다
- 새로 들어오는 수가 더 작다면, 지금 기억하고 있는 수들은 필요가 없어진다

최솟값 찾기 BOJ 11003

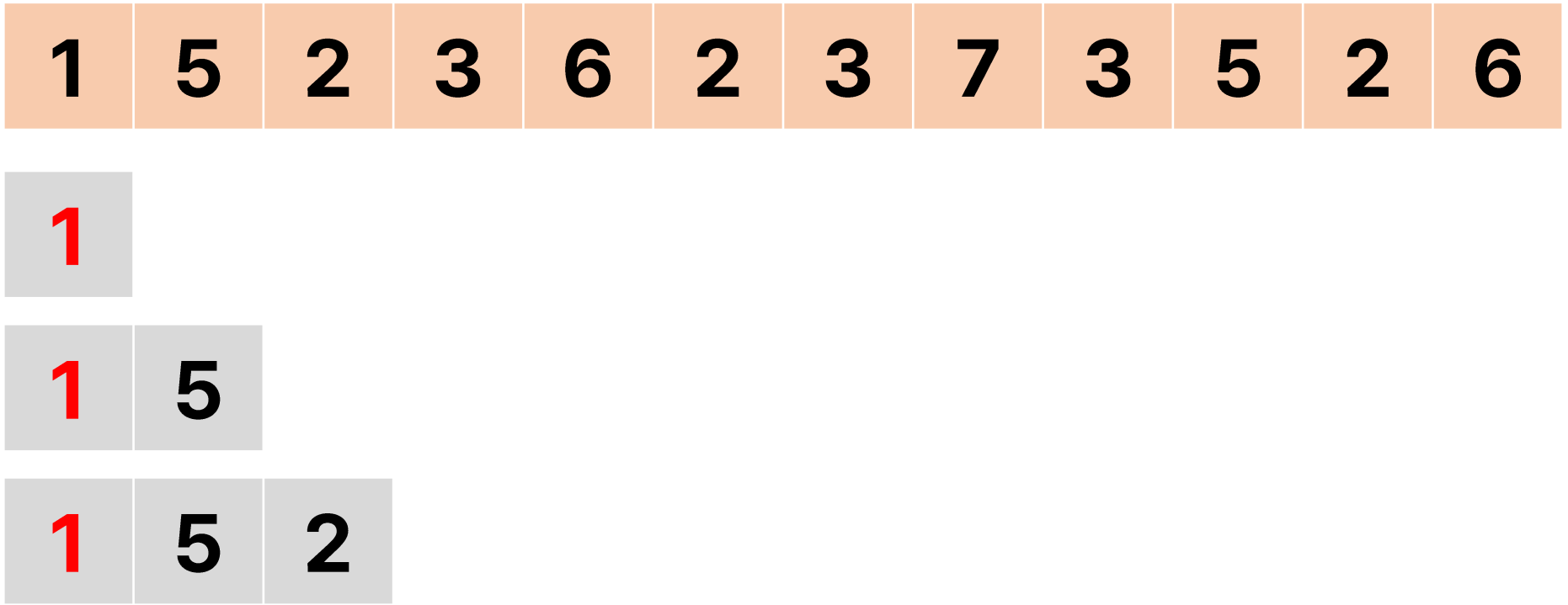
- 앞에서 뒤로 훑어나가면서 몇가지 성질을 파악해 보자

1	5	2	3	6	2	3	7	3	5	2	6

- 최솟값만 찾으면 되므로, 순서대로 자료구조에 수를 넣는다
- 만약 수가 현재 인덱스와 L보다 많이 차이나는 경우, 필요없는 수이다
- 해당 **자료구조에서** 필요한 수 중에서 **최솟값**을 꺼내면 된다

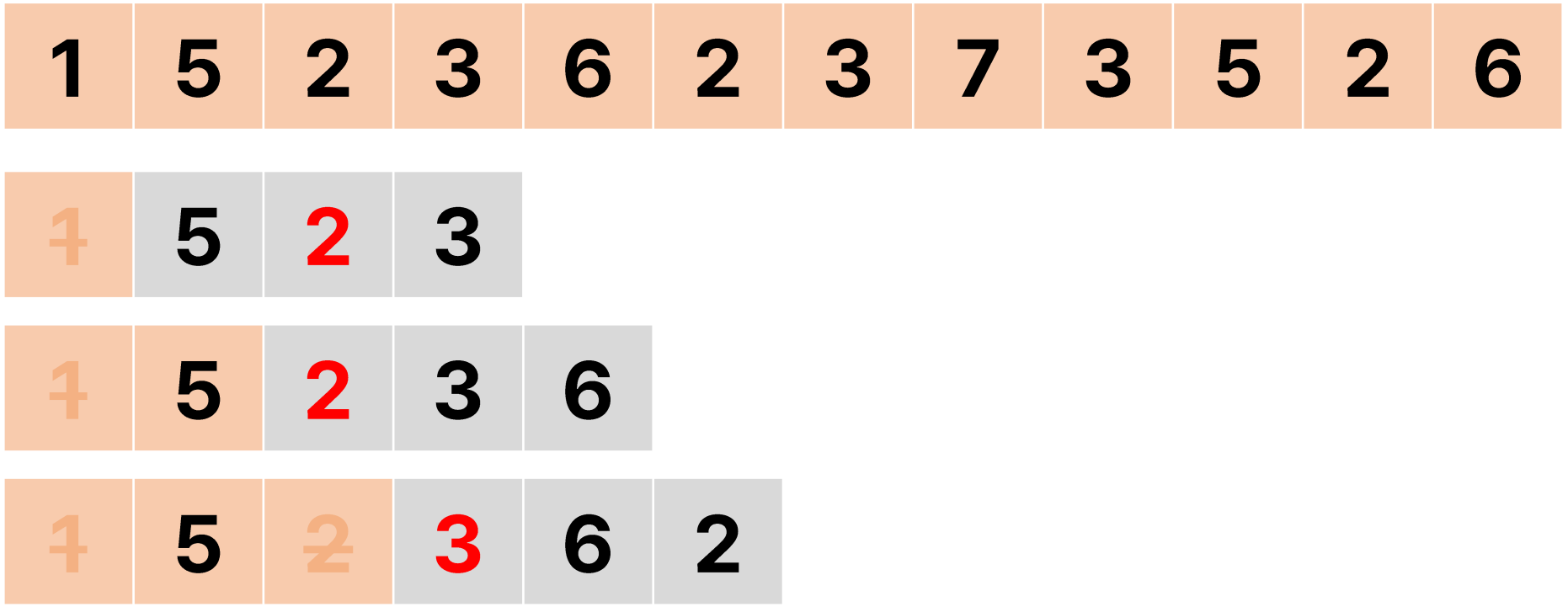
최솟값 찾기 BOJ 11003

- Priority Queue를 사용한 풀이



최솟값 찾기 BOJ 11003

- Priority Queue를 사용한 풀이



최솟값 찾기 BOJ 11003

- 모든 수가 우선순위 큐에 들어가므로 $O(NlogN)$

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
for(int i = 0; i < N; i++){
    cin >> t;
    while (!pq.empty() && pq.top().second + L <= i) pq.pop();
    pq.emplace(t, i);
    …
}
```

최솟값 찾기 BOJ 11003

- 앞에서 뒤로 훑어나가면서 몇가지 성질을 파악해 보자

1	5	2	3	6	2	3	7	3	5	2	6

- 새로 들어오는 수가 더 크거나 같다면, 이후에 사용될 여지가 있다
- 길이가 지난 수들은 기억할 필요가 없다
- **새로 들어오는 수가 더 작다면, 해당 수보다 큰 기억하고 있는 수들은 필요가 없어진다**

최솟값 찾기 BOJ 11003

1	5	2	3	6	2	3	7	3	5	2	6

- 1, 5가 삽입될 때에는 순서대로 삽입
- 2가 들어오는 순간, **5는 불필요한 정보**
- 더 나중에 들어오는 것을 우선시해야 하는데, 나중에 들어오는 수가 더 작으므로

최솟값 찾기 BOJ 11003

1	5	2	3	6	2	3	7	3	5	2	6

- L 구간을 넘어가는 경우에도 왼쪽에서 제거해줘야 함
- 오른쪽으로 삽입, 양쪽으로 제거
- Double-ended queue를 사용하자

최솟값 찾기 BOJ 11003

1	5	2	3	6	2	3	7	3	5	2	6

- 구간을 벗어나는 원소에 대해서 왼쪽에서 뽑기
- 지금 들어가는 원소보다 큰 수에 대해서 오른쪽에서 뽑기
- 모든 원소가 덱에서 삽입/삭제되므로 $O(N)$

- 위와 같은 방식으로 자료구조를 관리한다면 어떤 방식으로 저장될까?

Monotone Queue, Stack

- 원소들이 오름차순 / 내림차순으로 관리되는 경우 Monotone하다고 한다
- Monotone하게 자료구조를 관리할 경우 가질 수 있는 이점이 많음