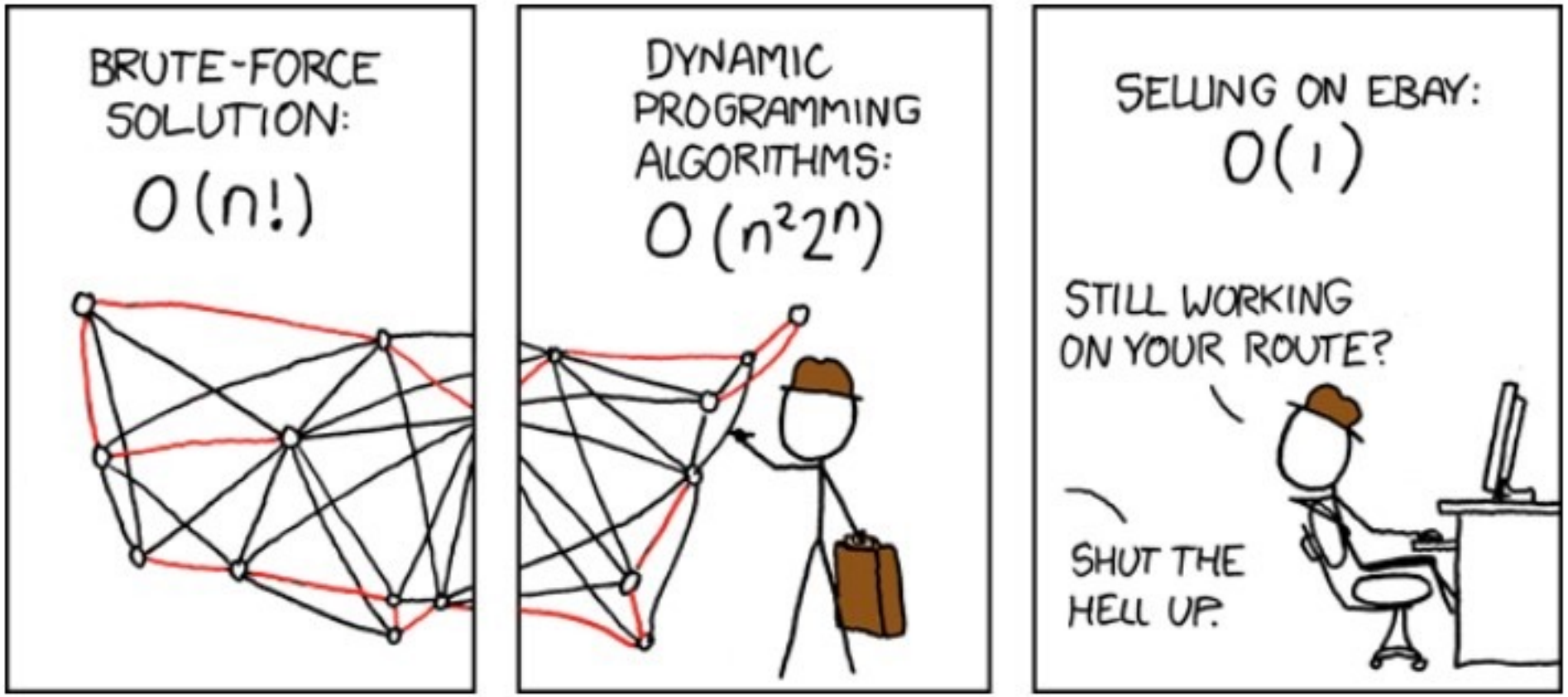
20차시

외판원 순회

외판원 순회 BOJ 11811

- 모든 도시를 순회하고 돌아오는데 최소 비용이 걸리도록 하는 문제
- a도시에서 b도시로 가는 비용과 b도시에서 a도시로 가는 비용은 동일하지 않을 수 있다
- Traveling Salesman Problem(TSP)라고 불리며 CS에서 많이 접하게 되는 문제이다

외판원 순회 BOJ 11811



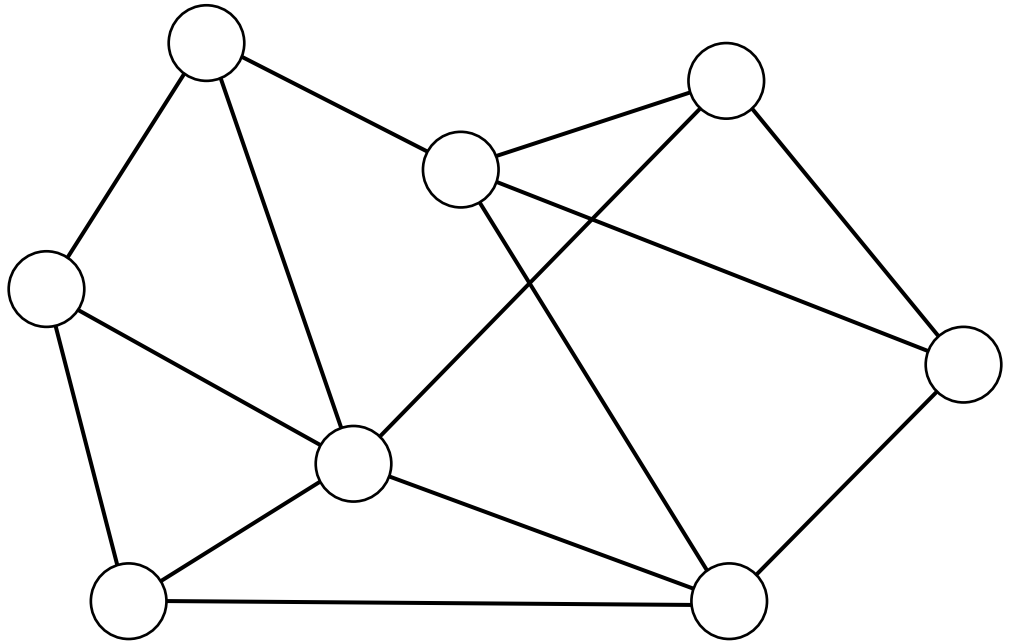
외판원 순회 BOJ 11811

- 가장 간단한 풀이
- 모든 경우의 수를 다 해본다
- N개의 도시를 방문할 경우를 세야하므로 N!의 경우의 수가 존재한다

- 더 줄일 수 있는 방법은 없을까?

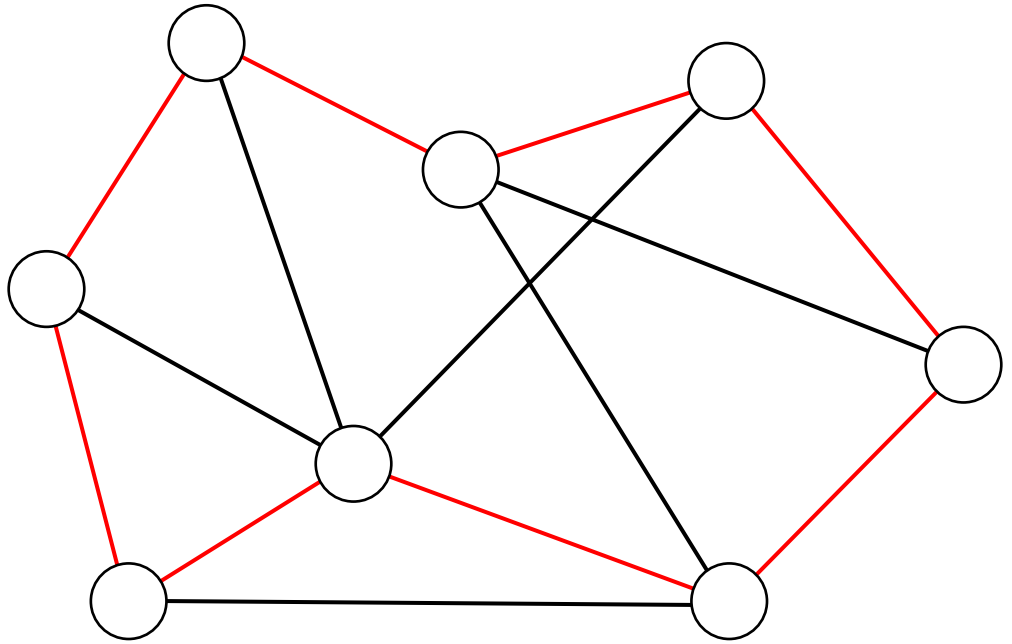
외판원 순회 BOJ 11811

- 우선 어느 도시에서 출발할 지 정해보자



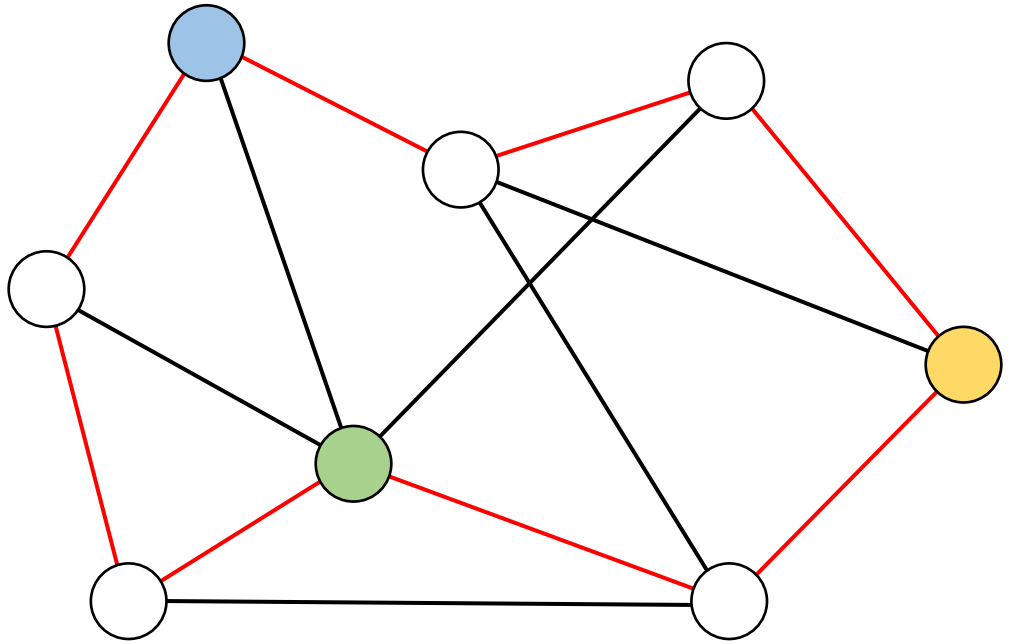
외판원 순회 BOJ 11811

- 빨간색 최소 비용이 드는 정답이라고 가정하자



외판원 순회 BOJ 11811

- 어떤 도시에서 시작하여도 정답은 동일하다, 즉 시작도시는 아무 곳이나 해도 된다



외판원 순회 BOJ 11811

- 도시가 5개가 있다고 생각해보자
- 이때 가능한 경우를 살펴보자

  1 → 2 → 3 → 4 → 5 → 1

  4 → 1 → 2 → 3 → 5 → 4

- 1 → 2 → 3으로 가는 경로는 겹치는 것을 알 수 있다

외판원 순회 BOJ 11811

- 모든 경우의 수를 다 살펴보아야 답을 구할 수 있다
- 1 → 2 → 3 → 4 → 5 → 1으로 갈 때 1 → 2 → 3 을 기억하고 있다면

  4 → 1 → 2 → 3 → 5 → 4을 살펴볼 때 4 → 1 + 기억한 값 +3 → 5 → 4로 찾는 다면 계산 횟수를 줄일 수 있다

- 우리는 방문했던 경로들을 **기억**하고 있어야한다

외판원 순회 BOJ 11811

- 기존에 구한 값을 기억하는 방법을 Memoization 기법이라고 부른다
- 주로 다이나믹 프로그래밍에서 사용된다
- 기억하는 방법은 주로 배열을 사용하며, 배열을 사용하기 애매한 경우 Map과 같은 Key, Value 로 저장한다
- Map: 이진 트리를 이용한 자료구조로 빠르게 값을 저장하고 찾는 데 사용된다
- ex) 배열을 이용했을 때 메모리 낭비가 심한 경우 Map을 사용

외판원 순회 BOJ 11811

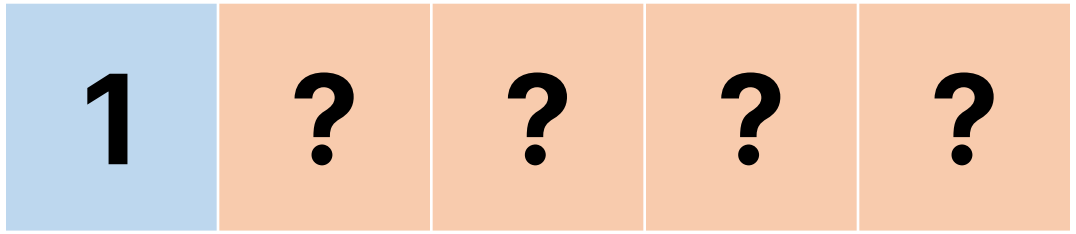
- 무슨 값들을 저장하고 있어야 할까?
- 어떤 도시들을 방문했는지 기억하고 있어야 한다
- 어떤 도시를 방문했는지 어떻게 표시하는가?

- 여러 개의 변수를 사용할 수 있지만 비트마스킹을 통해 하면 간편하다

외판원 순회 BOJ 11811

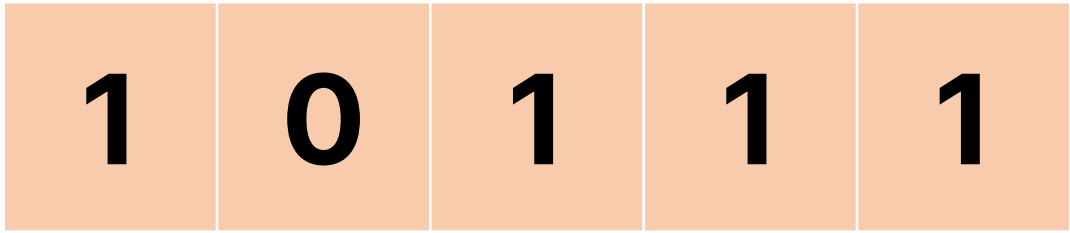
- 5개의 도시라면 5개의 비트가 필요하다
- 1번째 비트가 1이면 1번째 도시를 방문했다는 뜻이다

  1번째 비트가 0이면 1번째 도시를 방문하지 않았다는 뜻이다



외판원 순회 BOJ 11811

- 비트 10111은 다음과 같이 해석할 수 있다
- 첫번째 도시 방문
- 두번째 도시 미방문
- 세번째 도시 방문
- 네번째 도시 방문
- 다섯번째 도시 방문



외판원 순회 BOJ 11811

- 비트를 사용하면 좋은 점 배열로써 다룰 수 있다
- 해당 비트를 배열의 인덱스로써 사용하면 쉽게 값을 저장, 접근할 수 있다

1	0	1	1	1

외판원 순회 BOJ 11811

- 외판원 순회는 다이나믹 프로그래밍을 기반으로 사용한다
- 우리가 저장할 배열: dp[방문한 도시들(비트마스킹)][현재 있는 도시]
- 배열에 저장하는 값: 나머지 도시들을 방문하고 출발 도시로 갈 때의 최소 비용

- 해당 비용들을 저장해두고 이후에 재사용될 때 저장해둔 값을 이용해 연산을 줄인다
- 저장된 값이 없는 경우, 구해야한다

외판원 순회 BOJ 11811

```
int tsp(int visited, int cur) {
    if (__builtin_popcount(visited) == n) {
        if (!arr[cur][0])
            return INF;
        return arr[cur][0];
    }
    if (dp[visited][cur])
        return dp[visited][cur];
    dp[visited][cur] = INF;
    for (int i = 1; i < n; i++) {
        if (visited & (1 << i))
            continue;
        if (!arr[cur][i])
            continue;
        dp[visited][cur] = min(dp[visited][cur], tsp(visited | (1 << i), i) + arr[cur][i]);
    }
    return dp[visited][cur];
}
```

외판원 순회 BOJ 11811

- visited는 현재까지 방문한 도시의 종류(비트마스킹)
- cur은 현재 방문 중인 마지막 도시를 의미한다
- 출발 도시의 경우 어느 도시에서나 출발해도 괜찮으므로 0번 도시로 고정
- 함수가 리턴하는 값은 현재 이러한 도시들을 방문했으며, 어느 도시에 있을 때, 출발 도시까지 갈 때의 최소값

```
int tsp(int visited, int cur) {
}
```

외판원 순회 BOJ 11811

- visited의 비트의 개수가 n개인 경우 모든 도시를 방문한 것을 의미한다
- 모든 도시를 방문하고 출발 도시로 돌아와야 하므로 마지막 도시(cur)에서 출발 도시(0)으로 돌아올 수 있어야한다

```
if (__builtin_popcount(visited) == n) {
    if (!arr[cur][0])
        return INF;
    return arr[cur][0];
}
```

외판원 순회 BOJ 11811

- 만일 현재 방문한 도시와 마지막 도시를 기준으로 출발 도시로 돌아가는 최소값이 저장되어 있다면 그 값을 리턴함으로 연산을 줄임
- Memoization

```
if (dp[visited][cur])
    return dp[visited][cur];
```

외판원 순회 BOJ 11811

- dp배열은 0으로 초기화 되어있다
- 0은 방문한 적이 없다는 뜻을 의미한다
- 아직 최소값을 알 수 없으므로 출발 도시로 돌아갈 수 없다는 의미로 INF를 넣고, 이후 최신화 한다

```
dp[visited][cur] = INF;
```

외판원 순회 BOJ 11811

- 다음 도시의 후보지는 모든 도시이다(for문의 범위)
- 만일 내가 방문한 도시였다면 재방문할 수 없음(continue)
- 만일 내가 갈 수 없는 도시라면 방문할 수 없음(continue)

```
for (int i = 1; i < n; i++) {
        if (visited & (1 << i))
            continue;
        if (!arr[cur][i])
            continue;
        dp[visited][cur] = min(dp[visited][cur], tsp(visited | (1 << i), i) + arr[cur][i]);
    }
```

외판원 순회 BOJ 11811

- 방문하지 않은 도시이며, 방문할 수 있는 경우 현재 기억하고 있는 값을 최소값으로 최신화 한다

```
for (int i = 1; i < n; i++) {
        if (visited & (1 << i))
            continue;
        if (!arr[cur][i])
            continue;
        dp[visited][cur] = min(dp[visited][cur], tsp(visited | (1 << i), i) + arr[cur][i]);
    }
```

외판원 순회 BOJ 11811

```
int tsp(int visited, int cur) {
    if (__builtin_popcount(visited) == n) {
        if (!arr[cur][0])
            return INF;
        return arr[cur][0];
    }
    if (dp[visited][cur])
        return dp[visited][cur];
    dp[visited][cur] = INF;
    for (int i = 1; i < n; i++) {
        if (visited & (1 << i))
            continue;
        if (!arr[cur][i])
            continue;
        dp[visited][cur] = min(dp[visited][cur], tsp(visited | (1 << i), i) + arr[cur][i]);
    }
    return dp[visited][cur];
}
```

Pointer

메모리 할당

- 기존에 만들어진 자료형만큼만 사용하는 것이 아니라 내가 원하는 만큼 메모리를 할당할 수 있다

- malloc(size): size만큼 메모리를 할당한 후, 메모리 할당에 성공한 경우 그 주소를 돌려주며, 실패한 경우 nullptr을 돌려준다
- free(ptr): ptr주소의 할당된 메모리를 반환한다

메모리 할당

- ptr은 100바이트만큼 메모리를 할당 시도
- int_ptr은 int 5개의 공간만큼 메모리를 할당 시도
- malloc의 기본 돌려주는 형태는 void*이므로 형변환 필요

```
void *ptr = malloc(100);
int *int_ptr = (int *)malloc(sizeof(int) * 5);

free(ptr);
free(int_ptr);
```

void*

- void 포인터라고 부른다
- 포인터는 해당 주소를 해석할 자료형이 필요하다고 언급
- 실질적으로 포인터는 메모리 주소만을 나타낼 뿐 실질적으로 반드시 자료형이 필요하다고 할 수 없음(나는 해석하지 않겠다)
- 해석할 자료형을 정하지 않은 포인터이다

형 변환

- 기존에 사용하던 변수의 형태를 바꾸는 것
- int를 long long으로 변환하거나 char를 int로 바꾸는 등 자주 사용해 왔다

- 포인터 또한 가능하다

형 변환

- 포인터는 어떠한 형태로 해석할지 자료형을 갖고 있다
- 포인터 변수는 자료형에 무관하게 모두 메모리 주소를 갖고 있다

- 즉, 어떠한 메모리 주소를 어떠한 자료형으로 해석할 지는 선택이며, 변환에 제약이 없다

형 변환

- 하나의 포인터 주소를 가지고 여러 형태로 해석이 가능
- 결과가 어떻게 될까

```
int a = 65;
void *ptr = &a;

cout << *(int *)ptr << endl;
cout << *(long long *)ptr << endl;
cout << *(char *)ptr;
```

형 변환

- 65
- 쓰레기 값
- A

```
int a = 65;
void *ptr = &a;

cout << *(int *)ptr << endl;
cout << *(long long *)ptr << endl;
cout << *(char *)ptr;
```

형 변환

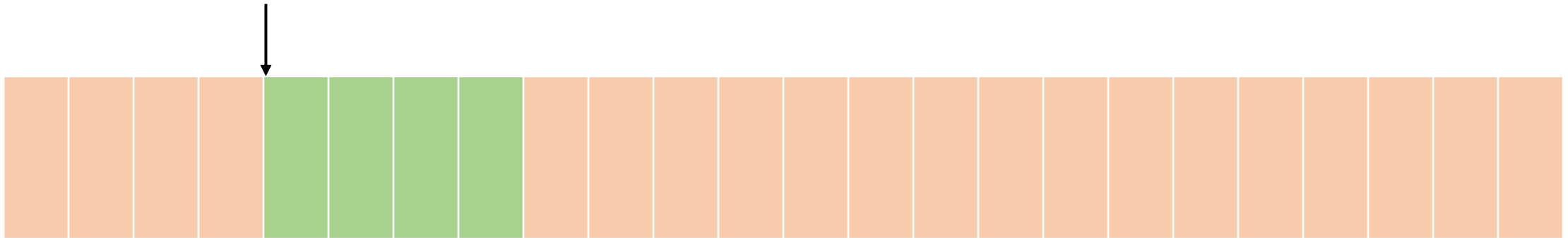
- 변수 a는 4바이트 만큼만 사용함
- long long은 8바이트를 가져와 사용
- 즉, 내가 값을 설정하지 않은 메모리 부분까지 값을 가져와 해석

```
int a = 65;
void *ptr = &a;

cout << *(int *)ptr << endl;
cout << *(long long *)ptr << endl;
cout << *(char *)ptr;
```

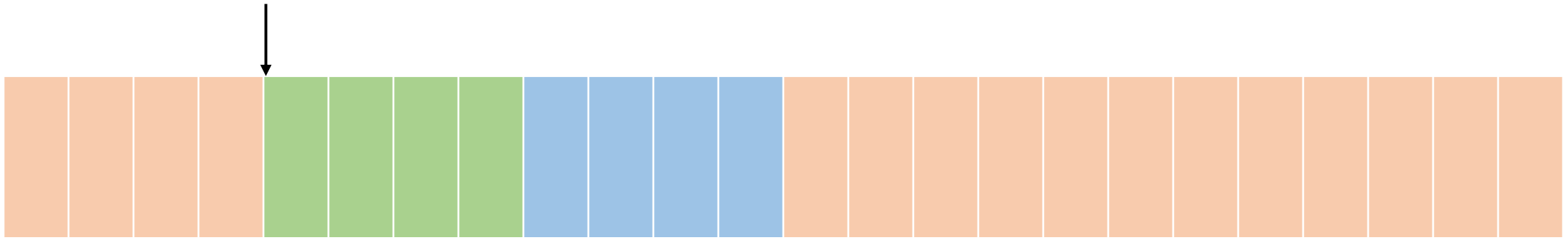
형 변환

- int a는 4바이트만을 사용
- 초록색 범위는 65라는 값으로 초기화 되어있다



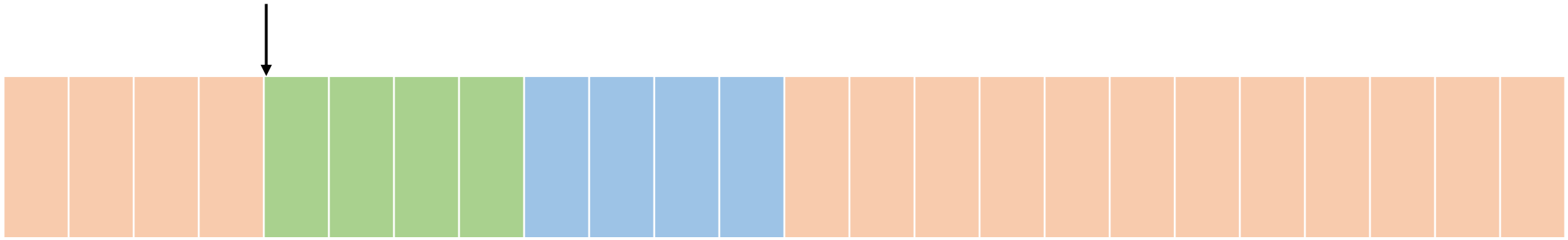
형 변환

- 만일 a의 주소를 long long으로 해석하려고 한다면 기존의 초록색을 넘어서 파란색까지 사용해 값을 해석
- 파란색에는 초기화 되어있지 않은 값을 담고 있음



형 변환

- 반대로 파란색에 특정한 값을 담을 수도 있음
- 만일 파란색이 어떤 특정한 값을 저장한 변수라면 내가 원하는 값으로 해킹할 수 있다
- 오버플로우를 이용한 해킹 방식



Queue

Queue

- First in First out(FIFO)
- 먼저 들어온 원소가 먼저 나가는 자료구조
- 간단하게 대기열을 생각하면 된다(게임 큐, 은행 대기열)

Queue

- 값이 들어오는 건 항상 오른쪽에서 들어오며, 왼쪽부터 값이 나가게 된다

Queue Push(Insert)

- 항상 값은 오른쪽에서 들어와서 왼쪽 끝부터 채워진다

Queue Push(Insert)

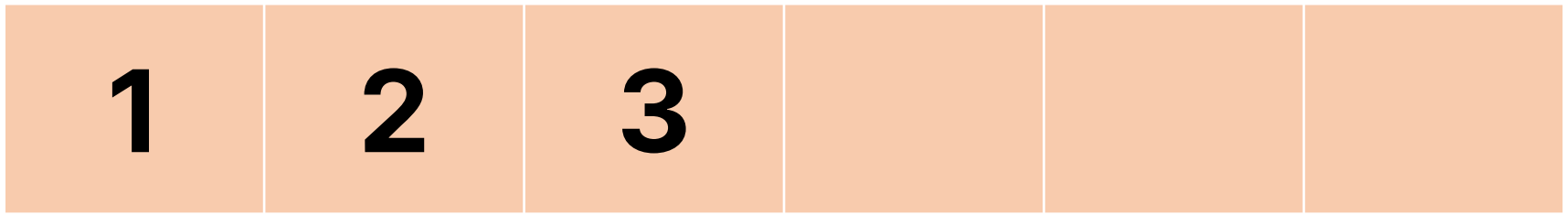
- 1 삽입

Queue Push(Insert)

- 2 삽입

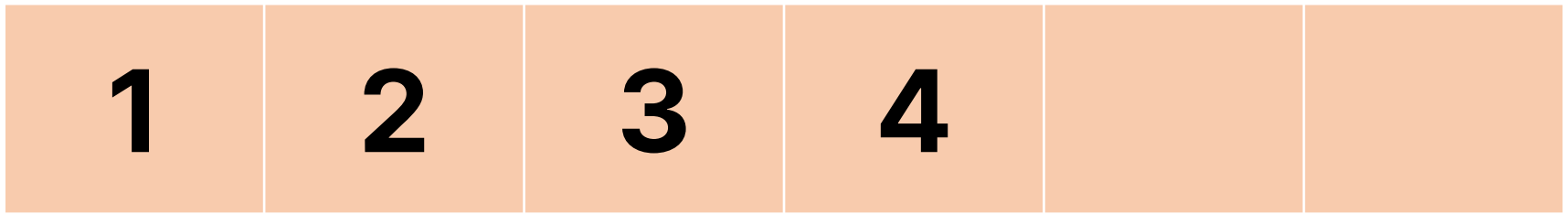
Queue Push(Insert)

- 3 삽입



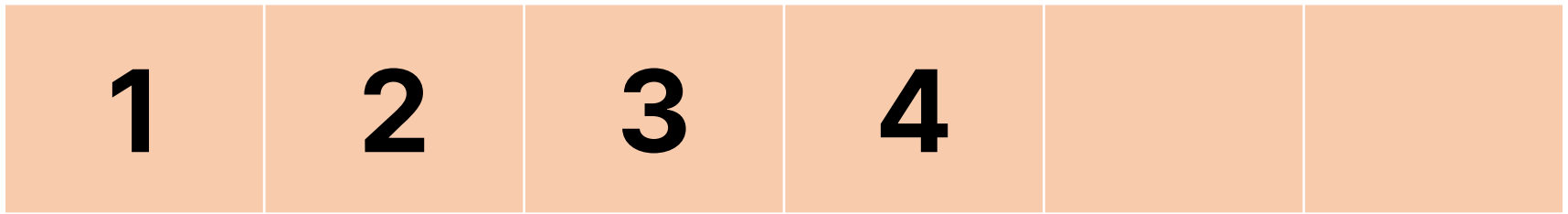
Queue Push(Insert)

- 4 삽입



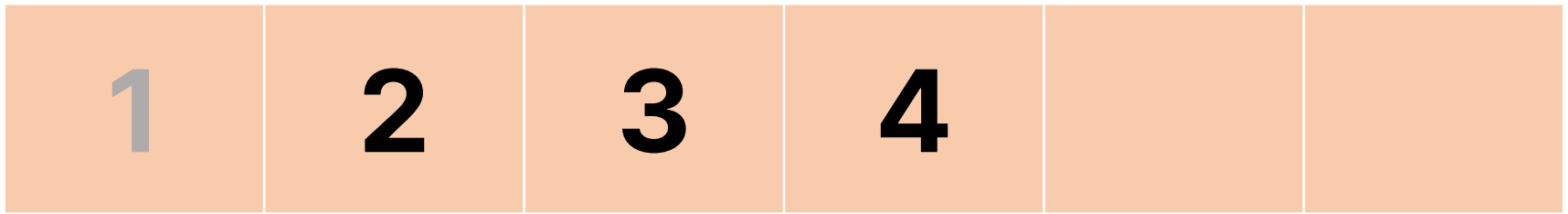
Queue Pop(Delete)

- 값을 뺄 때는 가장 처음에 넣은 값부터 뺀다



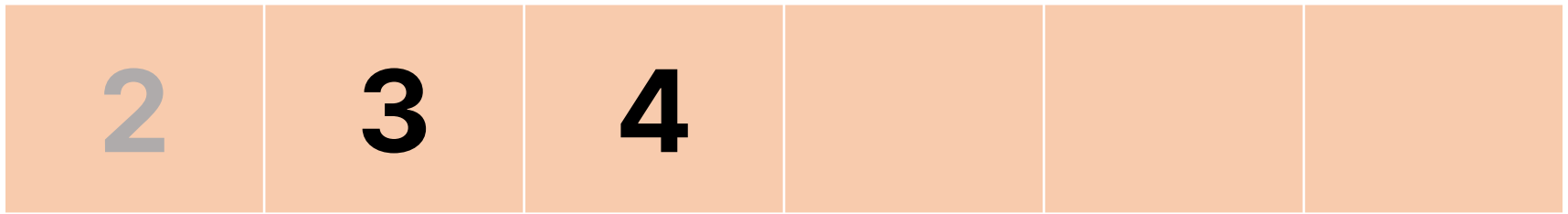
Queue Pop(Delete)

- Pop() -> 1제거



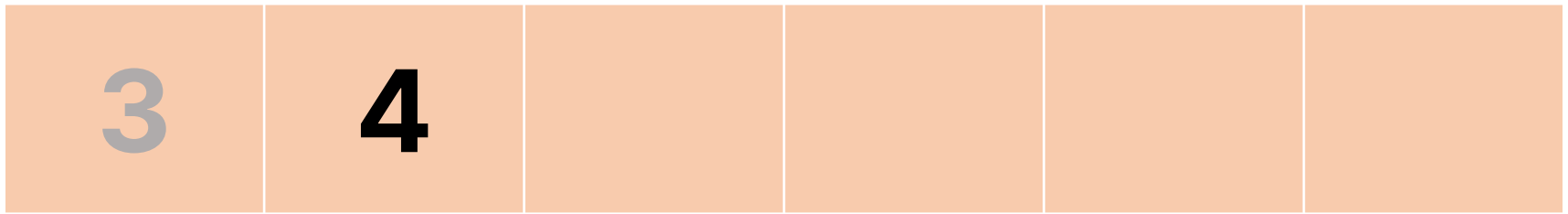
Queue Pop(Delete)

- Pop() -> 2제거



Queue Pop(Delete)

- Pop() -> 3제거



Queue Pop(Delete)

- Pop() -> 4제거

Queue

- Queue에서 데이터를 제거하는 과정을 보자
- 만일 남아있는 모든 데이터를 왼쪽으로 한 칸 옮긴다면 굉장히 많은 시간이 소요될 것이다
- 스택에서는 top이라는 변수가 존재해 top의 값만 이동해주면 굉장히 빠르게 값을 제거할 수 있었다

Queue

- 큐에도 이와 마찬가지인 변수가 존재한다
- 스택에서는 데이터가 들어가는 곳과 나가는 곳이 동일해 변수가 하나만 있으면 됐다
- 큐의 경우, 데이터가 나가는 위치를 알리는 변수와 데이터가 들어올 위치를 알리는 변수 2개가 존재한다
- 앞으로 데이터가 나가며 뒤로 데이터가 새로 들어온다
- front와 rear 변수를 이용한다

Queue

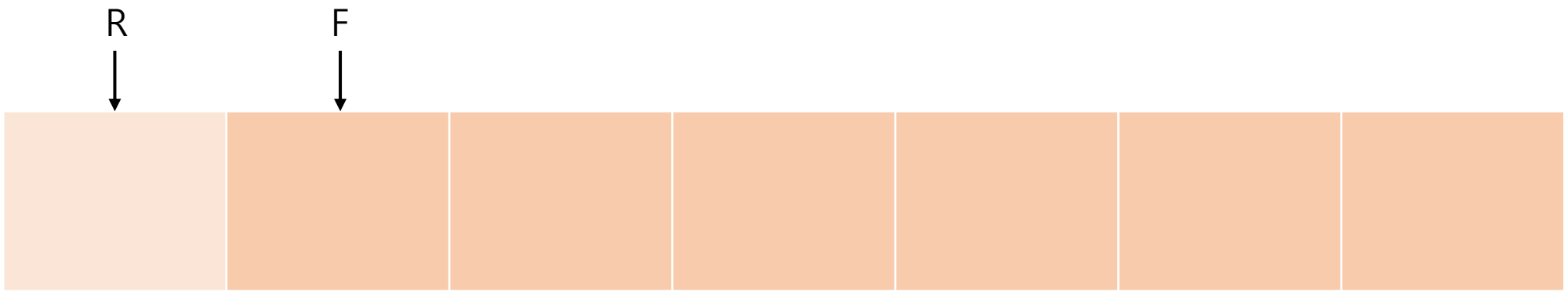
- front는 제일 앞 데이터가 있는 위치를 알려준다
- 데이터를 제거할 때는 front의 위치를 한 칸 뒤로 움직이면 된다
- rear는 제일 뒤 데이터가 있는 위치를 알려준다
- 데이터를 추가할 때는 rear를 한 칸 뒤로 움직이고 데이터를 rear에 추가하면 된다

Queue

- 따라서 데이터가 존재하는 경우 front는 rear와 같은 위치(데이터가 1개)이거나 front가 rear보다 앞 쪽에 존재한다
- rear가 front보다 앞에 존재하는 경우, 큐에 데이터가 존재하지 않는 것이다

Queue

- 제일 초기에는 rear가 임의의 -1을 가르킨다

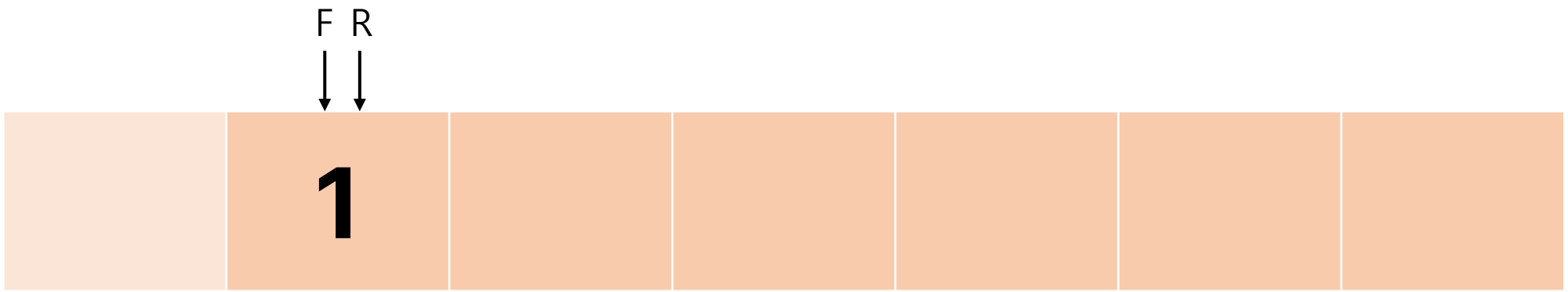


Queue

- 1 삽입

Queue

- 1 삽입

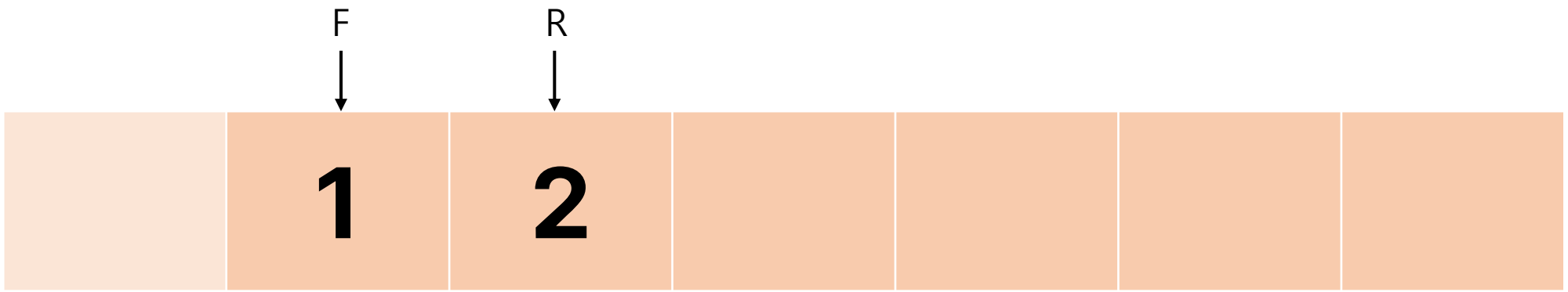


Queue

- 2 삽입

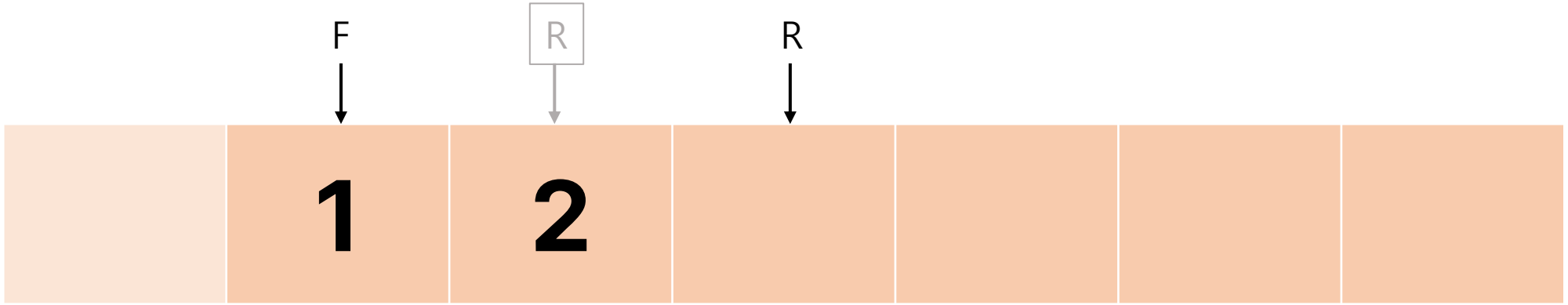
Queue

- 2 삽입



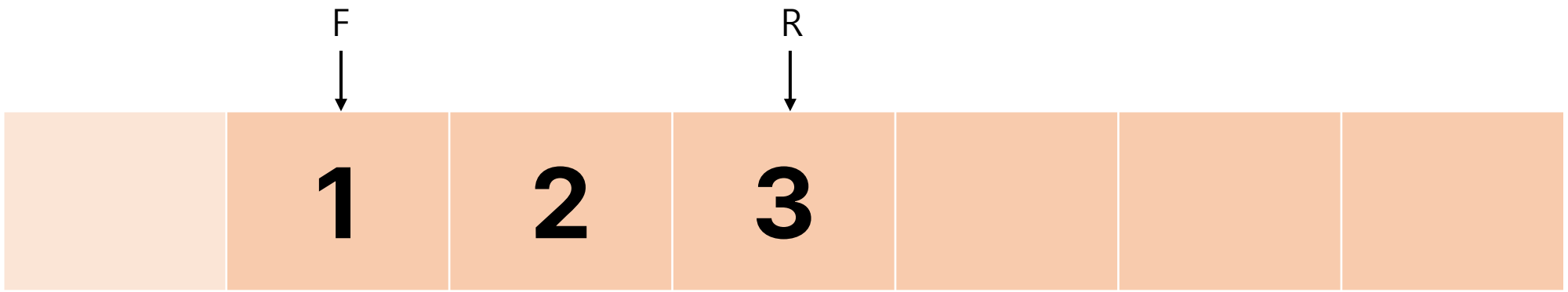
Queue

- 3 삽입



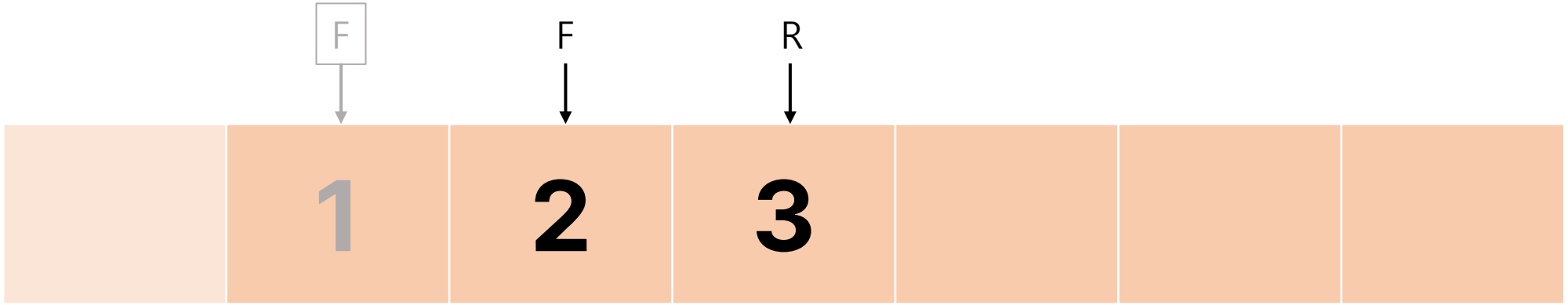
Queue

- 3 삽입



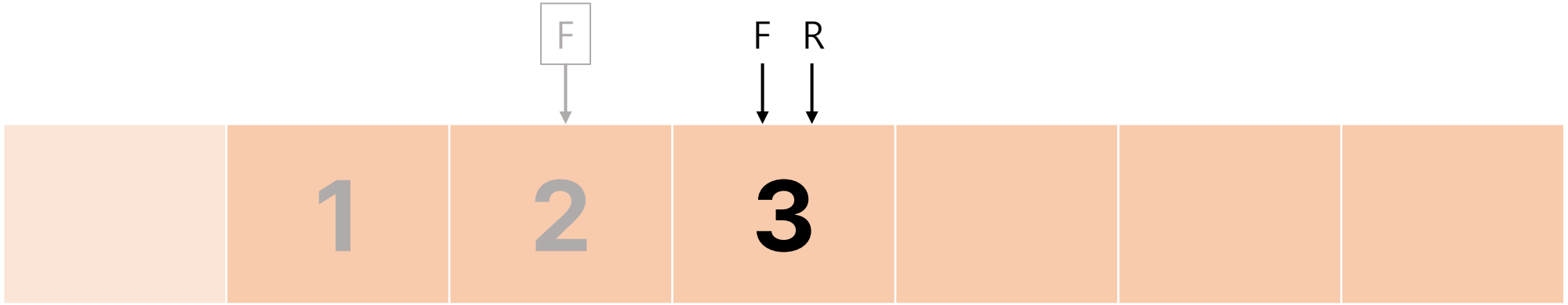
Queue

- Pop() -> 1 제거



Queue

- Pop() -> 2 제거



Queue

- 이런 식으로 데이터의 이동을 없애며, 큐에서 빠르게 데이터 삭제와 삽입을 할 수 있다
- 하지만 아직 단점이 남아있다

- 큐가 한번 사용한 자리는 다시 사용되지 않으며, 낭비된다는 것이다

  -> 다시 사용할 수 없을까?

Circular Queue

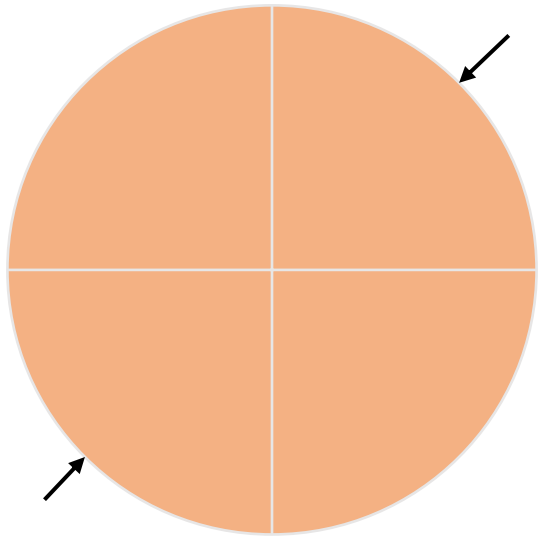
- 원형 큐는 앞선 선형 큐, 배열과 같은 일자로 만들어진 자료구조에서 생기는 문제점들을 보완하기 위하여 나왔다
- Front와 Rear가 배열의 끝에 도달하는 경우, 배열이 꽉 차서 더 이상 데이터를 추가하지 못하거나 연산이 불가능한 경우를 제외하면 인덱스를 처음으로 돌아간다

Circular Queue

- 선형 큐에 비하여 더 많은 양의 작업을 수행할 수 있다
- 선형 큐는 배열을 전부 사용한 경우, 더 이상 push 할 수 없어 활용도가 낮다
- 원형 큐의 경우, 앞에 공간이 남아있다면 해당 공간을 재사용 해, 꽉 찬 상태가 아니라면 계속 재사용이 가능하다

Circular Queue

- 배열에서는 끝까지 가면 Front와 Rear를 더 이상 움직이지 못했지만, 원형 큐에서는 계속해서 돌릴 수 있다



STL Queue

- #include <queue>
- queue<자료형> 변수명;

STL Queue

- void queue.push(x): 큐에 x를 삽입
- void queue.pop(): 큐에서 원소를 하나 제거, 큐가 비어있으면 런타임 에러
- 자료형 queue.front(): 큐의 제일 앞의 원소를 리턴, 큐가 비어있으면 런타임 에러
- bool queue.empty(): 큐가 비어있으면 true, 아니면 false
- size_type queue.size(): 큐의 크기를 리턴

Double-ended Queue

- Deque라는 자료구조
- 큐는 한쪽에서 데이터를 넣으면 반대 쪽에서 데이터를 빼야하며, 스택은 데이터의 삽입과 삭제가 같은 위치에서 이루어졌다
- 간단하게 이 둘을 조합한 형태이다
- 배열이 존재한다면 왼쪽과 오른쪽, 양쪽에서 삽입과 삭제가 가능한 형태의 자료구조이다

Deque

- 그림과 같이 양쪽에서 모두 데이터의 삽입과 삭제가 가능하다

Deque Push Back(Insert)

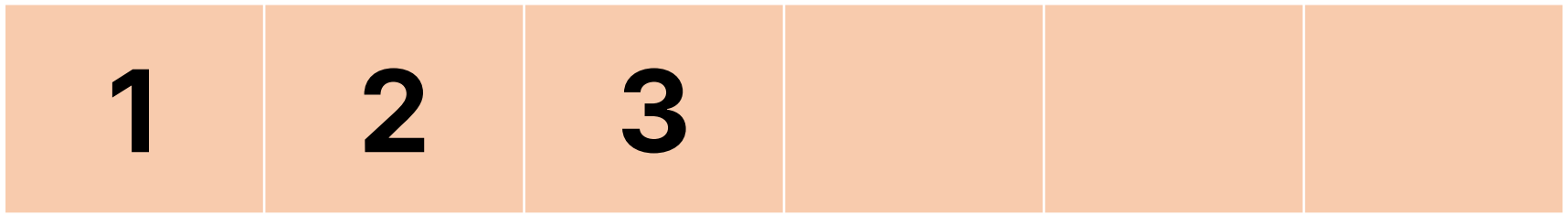
- 1 삽입

Deque Push Back(Insert)

- 2 삽입

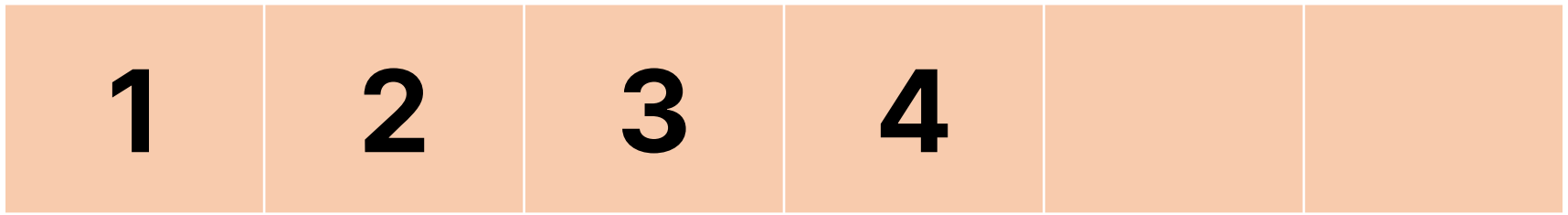
Deque Push Back(Insert)

- 3 삽입



Deque Push Back(Insert)

- 4 삽입



Deque Push Front(Insert)

- 1 삽입

Deque Push Front(Insert)

- 2 삽입

Deque Push Front(Insert)

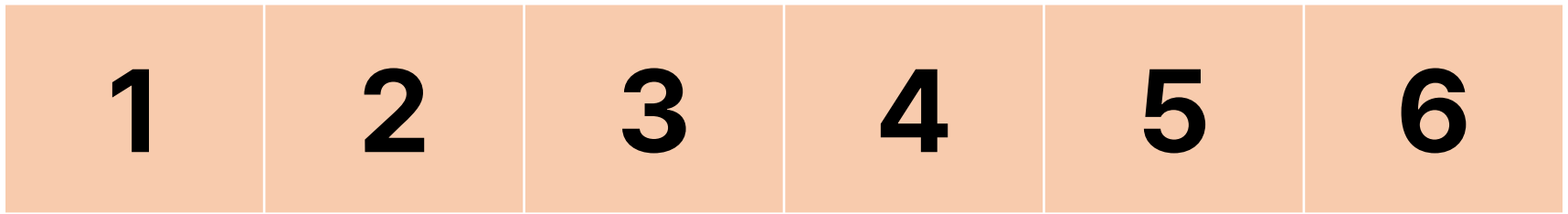
- 3 삽입

Deque Push Front(Insert)

- 4 삽입

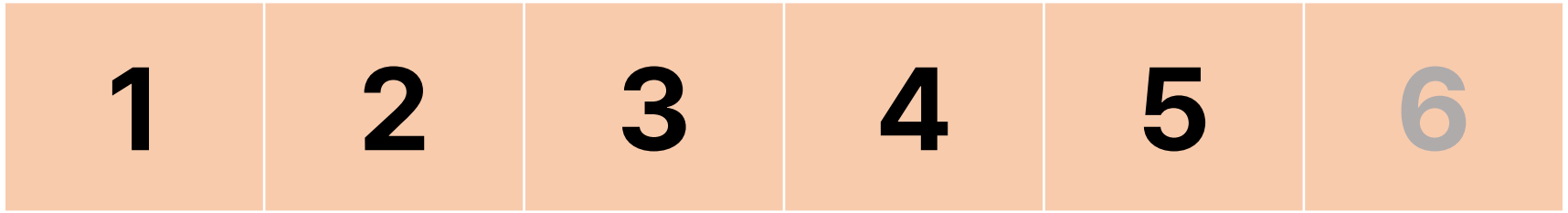
Deque Pop(Delete)

- 기본 덱이 다음과 같이 존재한다 해보자



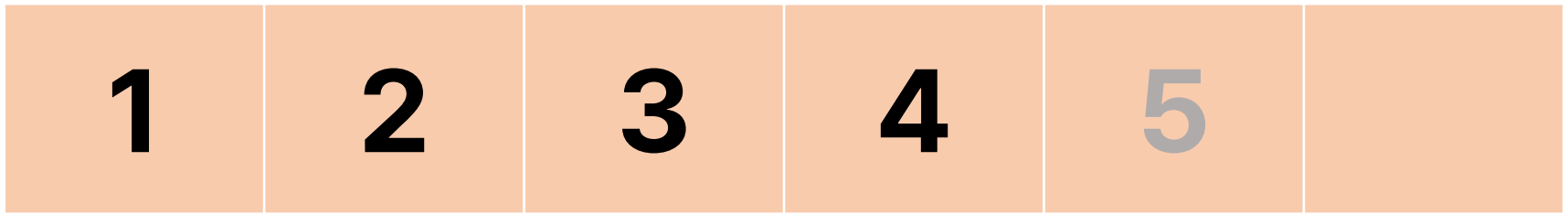
Deque Pop Back(Delete)

- Pop Back()->6제거



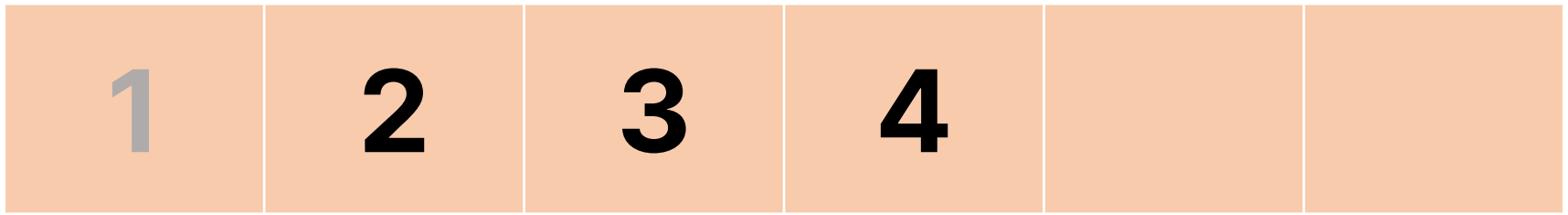
Deque Pop Back (Delete)

- Pop Back()->5제거



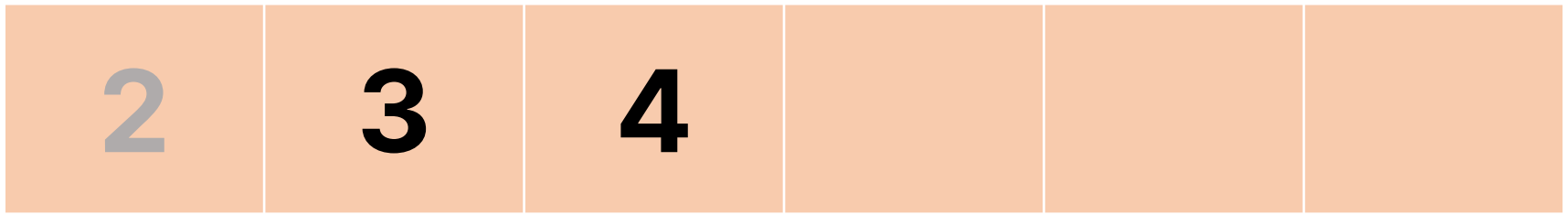
Deque Pop Back (Delete)

- Pop Front()->1제거



Deque Pop Back (Delete)

- Pop Front()->2제거



STL Queue

- #include <queue>
- #include <deque> <- 큐 사용 불가능
- deque<자료형> 변수명;

- 사용법은 벡터와 90% 유사하다

STL Deque

- void deque.push_front(x): 덱에 첫번째 원소로 x를 삽입
- void deque.pop_front(): 덱에서 첫번째 원소를 제거, 덱이 비어있으면 런타임 에러
- 자료형 deque.front(): 덱의 제일 앞의 원소를 리턴, 덱이 비어있으면 런타임 에러

STL Deque

- void deque.push_back(x): 덱에 마지막 원소로 x를 삽입
- void deque.pop_back(): 덱에서 마지막 원소를 제거, 덱이 비어있으면 런타임 에러
- 자료형 deque.back(): 덱의 제일 뒤의 원소를 리턴, 덱이 비어있으면 런타임 에러

STL Deque

- bool deque.empty(): 덱 비어있으면 true, 아니면 false
- size_type deque.size(): 덱의 크기를 리턴

문제

- 큐 BOJ 10845
- 덱 BOJ 10866
- 카드2 BOJ 2164
- 식당 메뉴 BOJ 26043
- 회전하는 큐 BOJ 1021

Coordinate Compression

제일 많은 숫자

- 간단한 예시를 생각해보자
- 1부터 100까지 범위의 숫자가 100만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

제일 많은 숫자

- 100까지 개수를 저장할 배열을 만들고 개수를 늘려주며 제일 많은 것을 세주면 된다

```
int num;
int cnt[101];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

제일 많은 숫자

- 과연 가능할까?

```
int num;
int cnt[2147483648];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

제일 많은 숫자

- 배열의 크기는 어떨까?
- int이므로 한 칸에 4바이트이다
- 즉, 2,147,483,648칸 * 4바이트 이므로 8,589,934,592바이트이다
- 약 8.5기가의 메모리를 사용해야한다

제일 많은 숫자

- 다른 문제로 살펴보자
- 1, 100, 10000 3가지 숫자 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

제일 많은 숫자

- 앞선 예시를 그대로 사용할 수 있다

```
int num;
int cnt[10001];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

제일 많은 숫자

- 하지만 값이 3개만 들어오는데 모든 변수를 사용하지 않는다

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

제일 많은 숫자

- 어떤 방식을 더 많이 사용하는가?

```
int num;
int cnt[10001];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

제일 많은 숫자

- 우리는 필요한 만큼의 변수 공간만 만들어 사용한다
- 또한, 비트마스킹처럼 각 배열에 의미를 부여할 수 있다

제일 많은 숫자

- cnt배열은 각 1, 100, 10000의 개수를 의미한다

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 문제를 다시 살펴보자
- 우리는 2,147,483,647개의 숫자를 위한 공간을 만들 필요가 없다

제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다

- 문제에서는 10만개의 숫자가 들어온다고 적혀 있다
- 몇 종류의 숫자가 들어올지는 모르지만 제일 많은 경우, 10만 종류의 숫자가 들어온다
- 따라서 우리는 10만개의 숫자를 위한 공간만 만들어 주면 된다

좌표 압축

- 입력으로 주어지는 서로 다른 두 수 사이에 존재하는 사용하지 않는 수들을 우리는 존재하지 않는다 가정하고 사용한다
- 1, 100, 10000이 들어올 때 이 3가지 숫자를 각각 0, 1, 2라고 생각하고 사용하지 않는 수들을 없다고 생각하는 것이다
- 사용하지 않는 수를 없애고 값들을 압축하는 방식, 좌표 압축이라고 부른다

좌표 압축

- 좌표 압축은 어디에 쓰일까?

- 모든 문제에 쓰일 수 있다
- 단순히 모든 문제의 입력을 귀찮게 만들 수 있다
- 문제의 알고리즘을 알아도, 입력 그대로 사용한다면 메모리 초과를 만들게 하는 문제

좌표 압축

- 좌표 압축은 어디에 쓰일까?

- 모든 문제에 쓰일 수 있다
- 단순히 모든 문제의 입력을 귀찮게 만들 수 있다
- 문제의 알고리즘을 알아도, 입력 그대로 사용한다면 메모리 초과를 만들게 하는 문제

좌표 압축

- 앞선 문제에서도 숫자 10만개를 주어질 때 제일 많은 수를 세는 프로그램은 크게 어렵지 않다
- 하지만 입력의 범위에 따라서 기존에 풀 수 있던 문제를 못 풀 수 있다

좌표 압축

- 그렇다면 어떻게 하면 될까?

- 우리는 들어오는 수들을 모두 기억하고 중복된 수가 존재하는지 확인해야한다
- 그 이후에 각각의 수들에게 연속되는(1, 2, 3, 4,...) 번호를 부여해야 한다

좌표 압축

- 45 1 60 20 78 30 1 20 20 30
- 10개의 수가 입력으로 들어온다고 가정하자
- 중복을 제거하는 경우 45 1 60 20 78 30 6개의 숫자가 남는다

- 우선 들어왔던 수들을 기억하고 있어야 한다
- 기억하고 있어야 중복된 수가 들어오는지를 알 수 있다

좌표 압축

- 값들이 들어올 때, 존재하는지 확인하려면 어떻게 해야 할까?

- 들어왔던 모든 수를 탐색하며 중복인지를 확인해야한다
- 들어왔는지 확인하는 배열을 사용하려면 마찬가지로 수의 범위만큼 배열의 칸이 필요하므로 불가능하다

좌표 압축

- 지금까지 들어온 수 들을 결국 모두 확인해야한다
- 모든 수 N개에 대하여 K번째 수는 K-1개의 숫자를 확인하므로 $O(N^2)$의 시간복잡도를 보인다(비효율적)

- 매번 확인을 하지 말고 마지막에 한번에 확인을 하자

좌표 압축

- 모든 수를 배열에 넣는다
- 모든 수를 정렬하면 같은 수들은 묶인다
- 정렬의 시간복잡도는 $O(\log N)$이므로 훨씬 빠르다

좌표 압축

- 또한 정렬하는 이유는 또 존재한다
- 좌표 압축을 하는 경우 무작위로 값을 부여하는 경우, 수의 대소관계가 직관적으로 보이지 않는다
- 입력으로 주어지는 수를 오름차순에 맞추어 좌표 압축 값을 부여하면 수의 **대소관계를 유지**하면서 좌표압축을 할 수 있다

좌표 압축

- 기존의 수를 좌표 압축된 수로 바꾸어 줘야 한다
- ex) 45는 3(0 base) 또는 4(1 base)로 바꾸어 줘야 한다

- 따라서 우리는 기존의 수가 몇 번째 수인지 찾아야한다

1	20	30	45	60	78

좌표 압축

- 두가지 방법이 존재한다. 주로 이분 탐색을 사용한다

- 브루트포스 $O(N)$
- 이분 탐색 $O(\log N)$

문제

- 좌표 압축 BOJ 18870