제5장

큐 (queue)

큐

개념과 응용

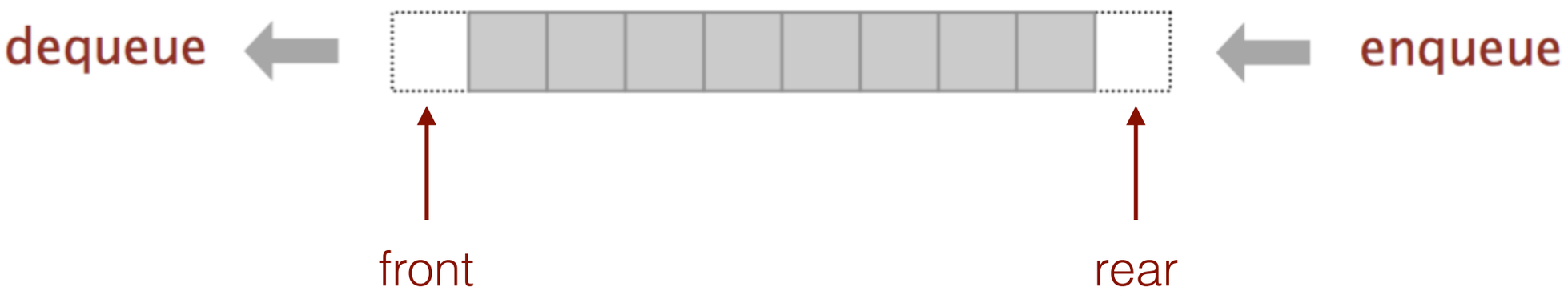
Queue

- 큐(queue) 역시 스택과 마찬가지로 일종의 리스트
- 단 데이터의 삽입은 한쪽 끝에서, 삭제는 반대쪽 끝에서만 일어남
- 삽입이 일어나는 쪽을 **rear**, 삭제가 일어나는 쪽을 **front**라고 부름
- **FIFO (First-In, First-Out)**라고 불림
- 예: 프린터 큐, 등

큐가 지원하는 연산

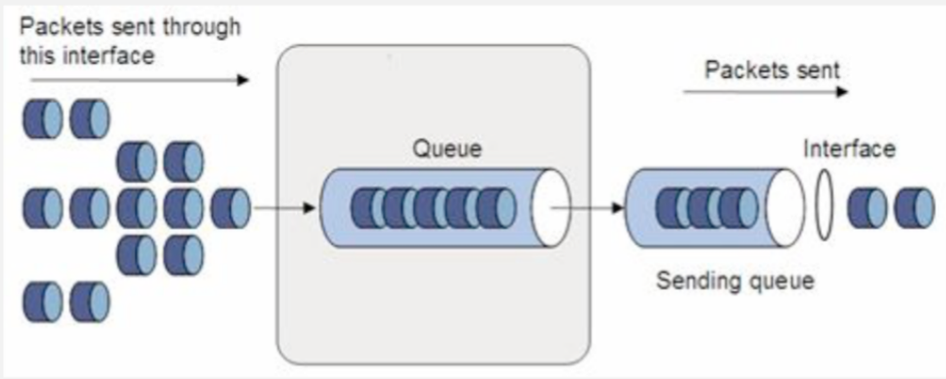
- **insert, enqueue, offer, push: 큐의 rear에 새로운 원소를 삽입하는 연산**
- **remove, dequeue, poll, pop: 큐의 front에 있는 원소를 큐로부터 삭제하고 반환하는 연산**
- **peek, element, front: 큐의 front에 있는 원소를 제거하지 않고 반환하는 연산**
- **is_empty: 큐가 비었는지 검사**


top
push
pop


dequeue
enqueue
front
rear

큐의 응용

- **CPU 스케쥴링**
  - multitasking 환경에서 프로세스들은 큐에서 CPU가 할당되기를 기다린다.
- **데이터 버퍼**
  - 네크워크를 통해 전송되는 패킷(packet)들은 도착한 순서대로 버퍼에 저장되어 처리되기를 기다린다.
- **그 외에도 자원을 공유하는 대부분의 경우에 큐가 사용됨**

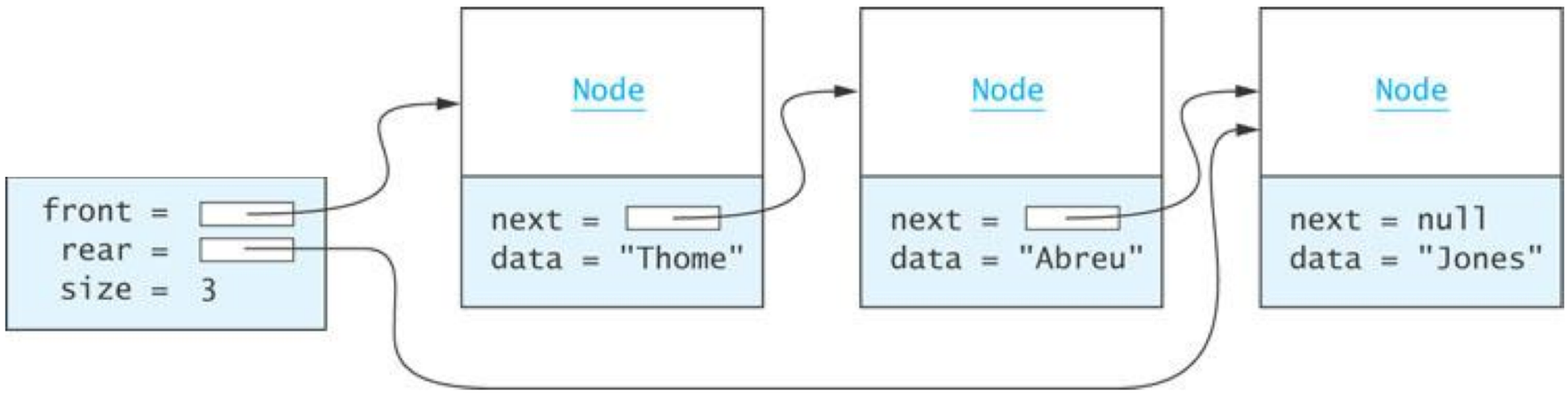


큐의 구현

배열 혹은 연결리스트를 이용한

Linked List로 구현

- 큐의 **rear**에서는 삽입, **front**에서는 삭제가 일어남. 따라서 연결리스트의 앞쪽을 **front**, 뒤쪽을 **rear**로 하는 것이 유리함
- 삽입을 하기 위해서는 마지막 노드의 주소를 항상 기억해야 함



front **rear**

queueADT.h

```
#ifndef QUEUEADT_H
#define QUEUEADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct queue_type *Queue;

Queue create();
void destroy(Queue q);
void make_empty(Queue q);
bool is_empty(Queue q);
void enqueue(Queue q, Item i);
Item dequeue(Queue q);
Item peek(Queue q);
int get_size(Queue q);

#endif
```

연결리스트로 구현: queueADT.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "queueADT.h"

struct node {
    Item data;
    struct node *next;
};

struct queue_type {
    struct node *front;
    struct node *rear;
    int size;
};

void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

int get_size(Queue q)
{
    return q->size;
}
```

연결리스트로 구현: queueADT.c

```
Queue create()
{
    Queue q = malloc(sizeof(struct queue_type));
    if (q == NULL)
        terminate("Error in create: queue could not be created.");
    q->front = NULL;
    q->rear = NULL;
    q->size = 0;
    return q;
}
```

```
void destroy(Queue q)
{
    make_empty(q);
    free(q);
}

void make_empty(Queue q)
{
    while (!is_empty(q))
        dequeue(q);
    q->size = 0;
}
```

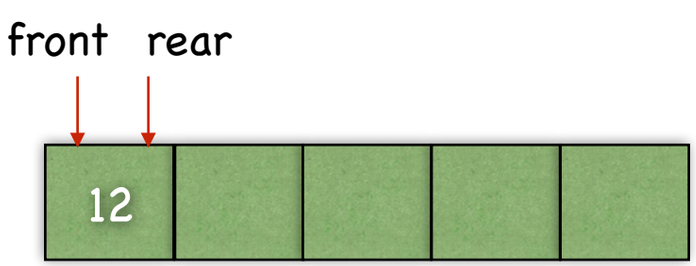
연결리스트로 구현: queueADT.c

```
bool is_empty(Queue q)
{
    return q->front == NULL;    /* or return q->size == 0 */
}

void enqueue(Queue q, Item i)
{
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL)
        terminate("Error in push: queue is full.");
    new_node->data = i;
    new_node->next = NULL;
    if (q->front == NULL) {
        q->front = new_node;
        q->rear = new_node;
    }
    else  {
        q->rear->next = new_node;
        q->rear = new_node;
    }
    q->size++;
}
```

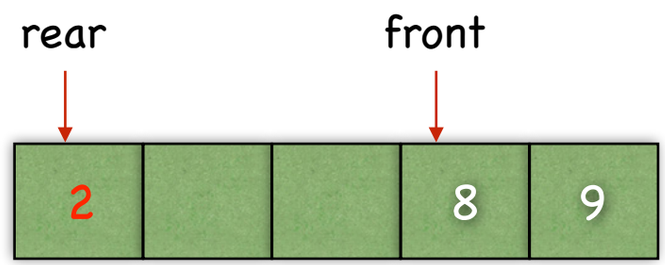
연결리스트로 구현: queueADT.c

```
Item dequeue(Queue q)
{
    struct node *old_front;
    Item i;

    if (is_empty(q))
        terminate("Error in dequeue: queue is empty.");

    old_front = q->front;
    i = old_front->data;
    q->front = old_front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(old_front);
    q->size--;
    return i;
}

Item peek(Queue q)
{
    if (is_empty(q))
        terminate("Error in peek: queue is empty.");
    return q->front->data;
}
```

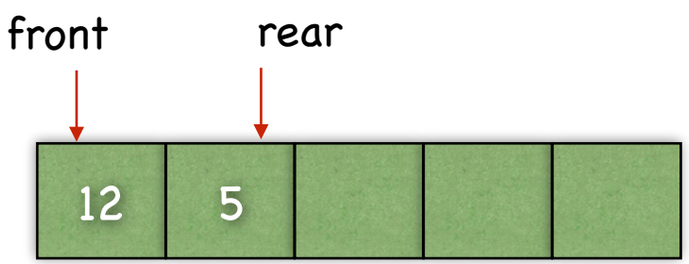
환형 배열을 이용한 구현



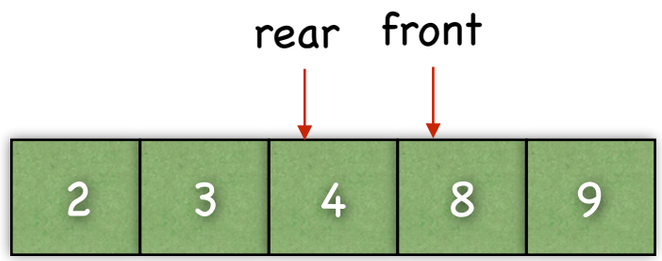
12 삽입



2 삽입

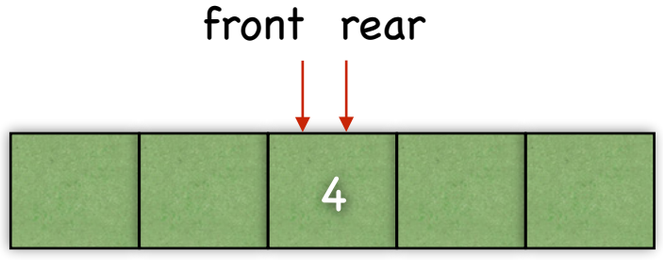


5 삽입

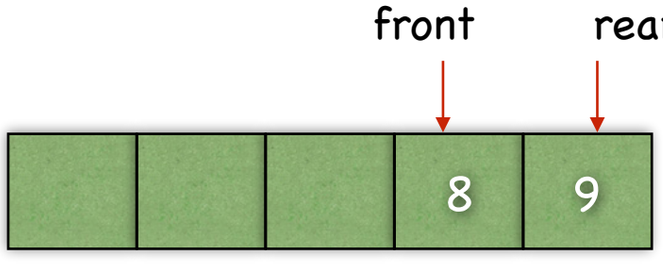


3, 4 삽입

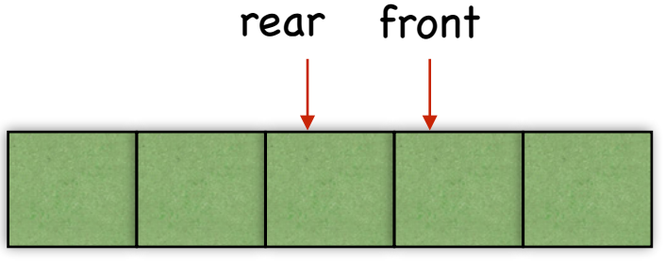
7, 8, 9 삽입



4번 연속 삭제



3번 연속 삭제



4의 삭제

배열로 구현: queueADT.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "queueADT.h"

#define INIT_CAPACITY 100

struct queue_type {
    Item *contents;      /* 배열 */
    int front;
    int rear;
    int size;            /* 저장된 데이터의 개수 */
    int capacity;        /* 배열 contents의 크기 */
};

void terminate(const char *message)
{
    printf("%s\n", message);
    exit(1);
}

int get_size(Queue q)
{
    return q->size;
}
```

배열로 구현: queueADT.c

```
Queue create()
{
    Queue q = (Queue)malloc(sizeof(struct queue_type));
    if (q == NULL)
        terminate("Error in create: queue could not be created.");
    q->contents = (Item *)malloc(INIT_CAPACITY * sizeof(Item));
    if (q->contents == NULL) {
        free(q);
        terminate("Error in create: queue could not be created.");
    }
    q->front = 0;
    q->rear = -1;
    q->size = 0;
    q->capacity = INIT_CAPACITY;
    return q;
}

void destroy(Queue q)
{
    free(q->contents);
    free(q);
}
```

배열로 구현: queueADT.c

```
void make_empty(Queue q)
{
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}
```

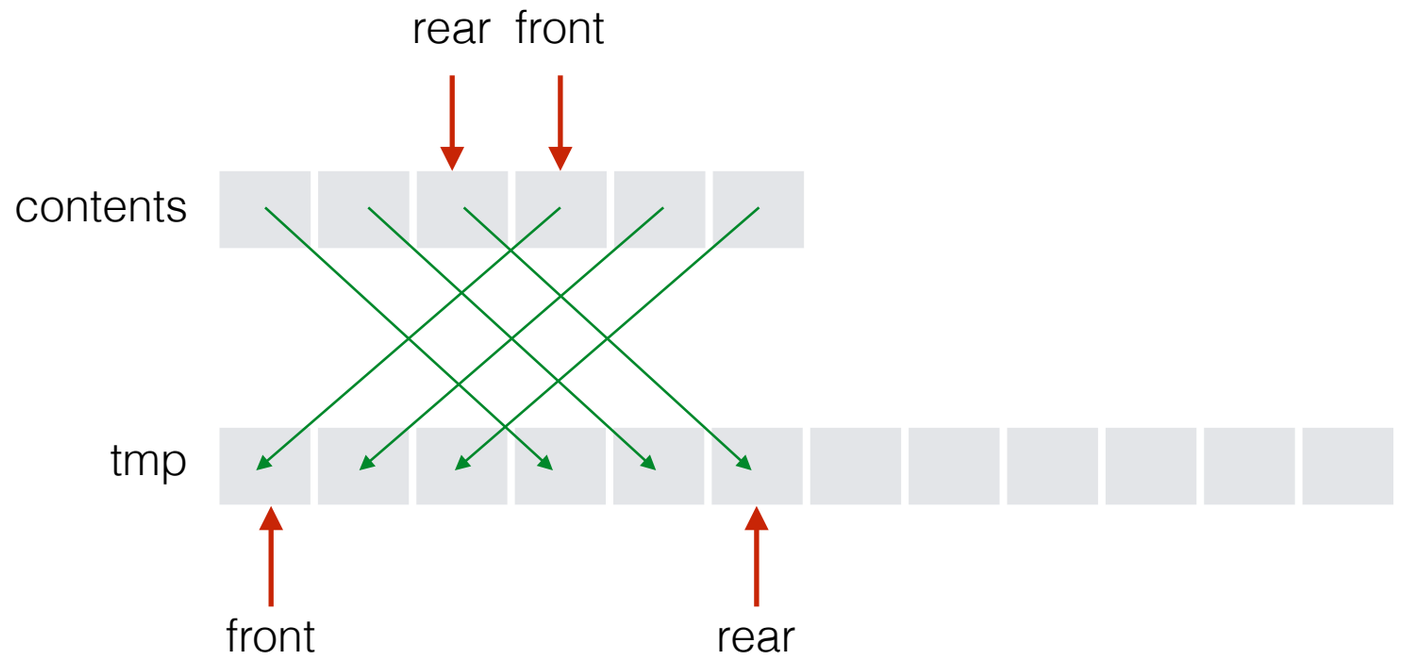
```
bool is_empty(Queue q)
{
    return q->size == 0;
}

bool is_full(Queue q)
{
    return q->size == q->capacity;
}
```

배열로 구현: queueADT.c

```
void enqueue(Queue q, Item i)
{
    if (is_full(q))
        reallocate(q);
    q->rear = (q->rear + 1)%q->capacity;
    q->contents[q->rear] = i;
    q->size++;
}

Item dequeue(Queue q)
{
    if (is_empty(q))
        terminate("Error in dequeue: queue is empty.");
    Item result = q->contents[q->front];
    q->front = (q->front + 1)%q->capacity;
    q->size--;
    return result;
}

Item peek(Queue q)
{
    if (is_empty(q))
        terminate("Error in peek: queue is empty.");
    return q->contents[q->front];
}
```

배열로 구현: queueADT.c

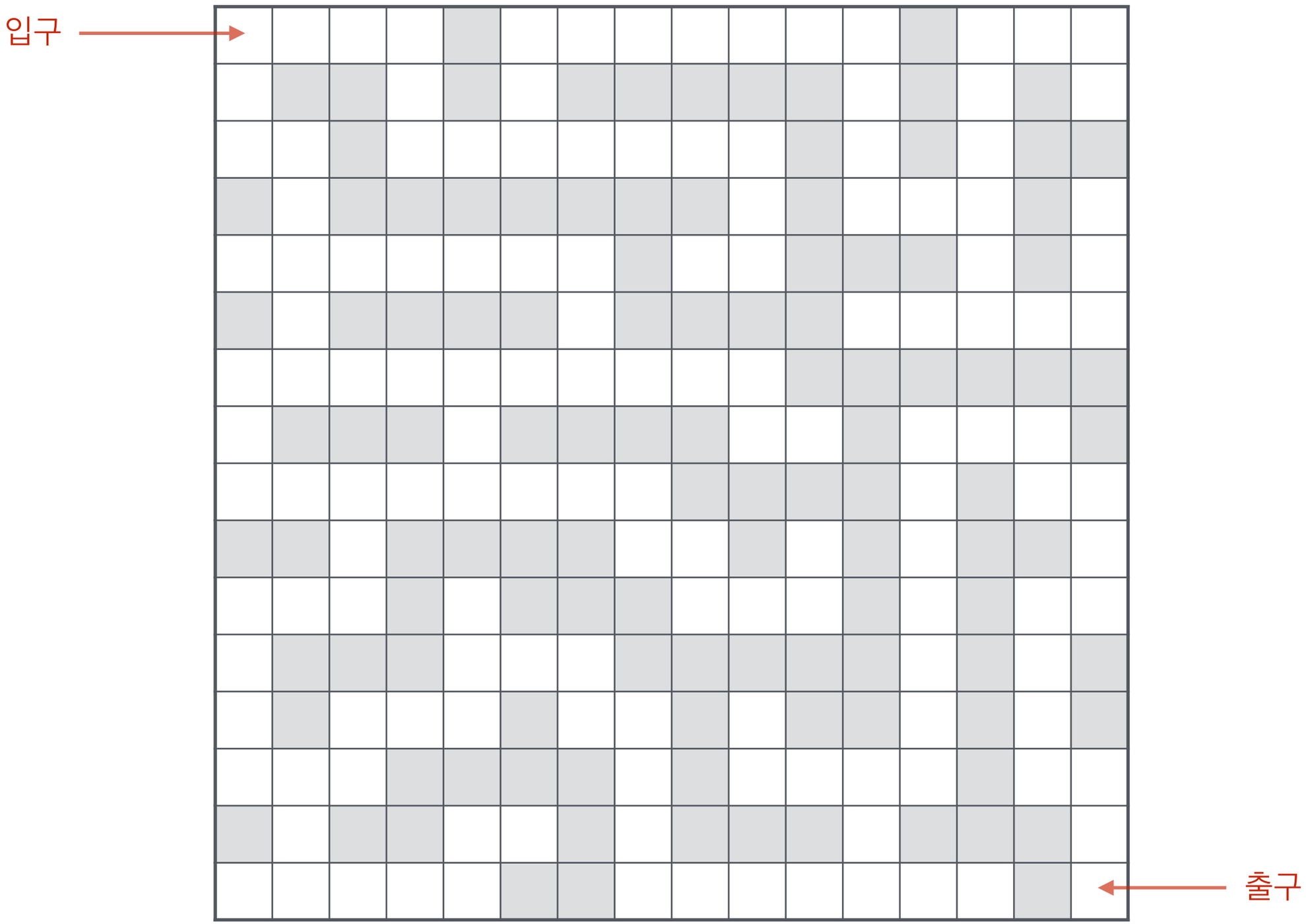
```
void reallocate(Queue q)
{
    Item *tmp = (Item *)malloc(2 * q->capacity * sizeof(Item));
    if (tmp == NULL) {
        terminate("Error in create: queue could not be expanded.");
    }

    int j = q->front;
    for (int i=0; i<q->size; i++)  {
        tmp[i] = q->contents[j];
        j = (j + 1)%q->capacity;
    }
    free(q->contents);

    q->front = 0;
    q->rear = q->size - 1;
    q->contents = tmp;
    q->capacity *= 2;
}
```



미로찾기

Maze Revisited

미로찾기

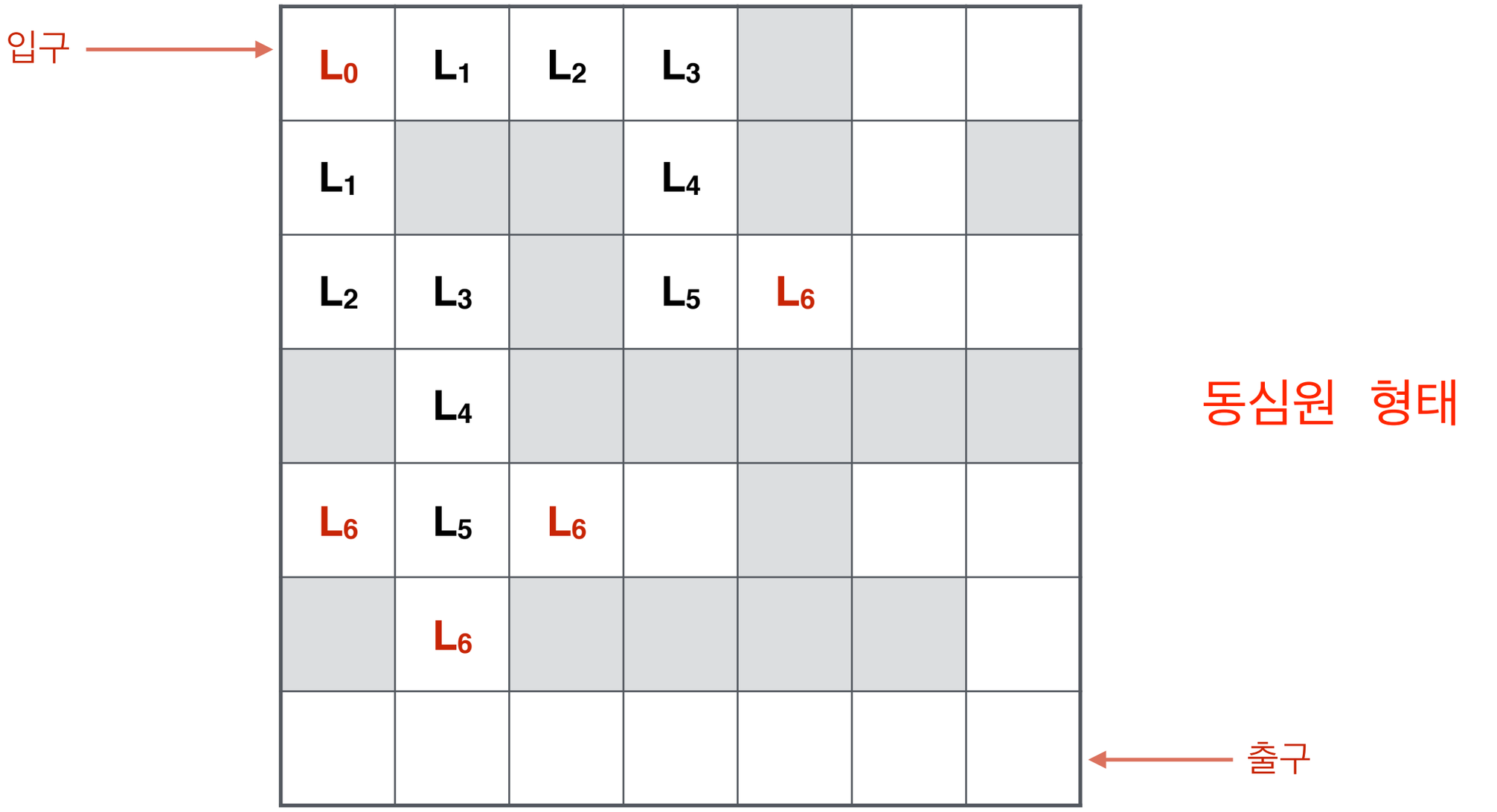


너비 우선 탐색으로 미로찾기

- **다음과 같은 순서로 셀(cell)들을 방문**
  - L_0 = {s}, 여기서 s는 출발 지점
  - L_1 = L_0에서 1번에 갈 수 있는 모든 셀들
  - L_2 = L_1에서 1번에 갈 수 있는 셀들 중에서 L_0에 속하지 않는 셀들

  ...
  - L_i = L_{i-1}에서 1번에 갈 수 있는 셀들 중에서 L_{i-2}에 속하지 않는 셀들

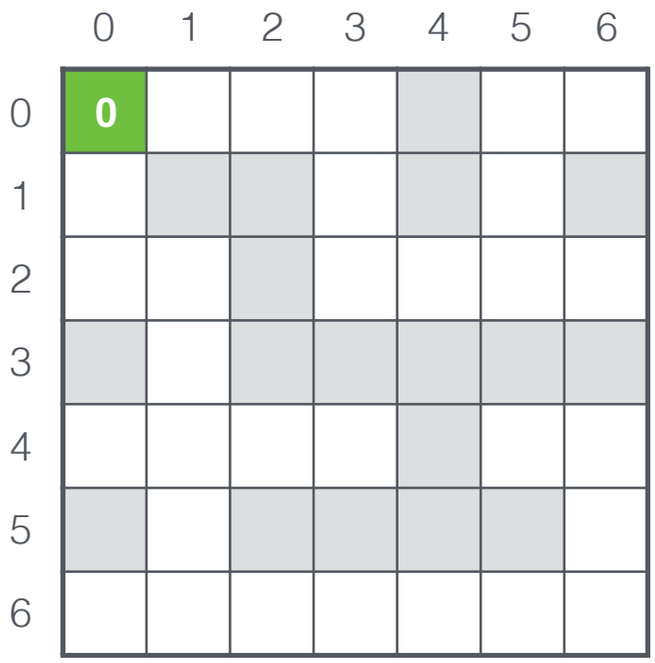
너비 우선 탐색으로 미로찾기



너비 우선 탐색으로 미로찾기

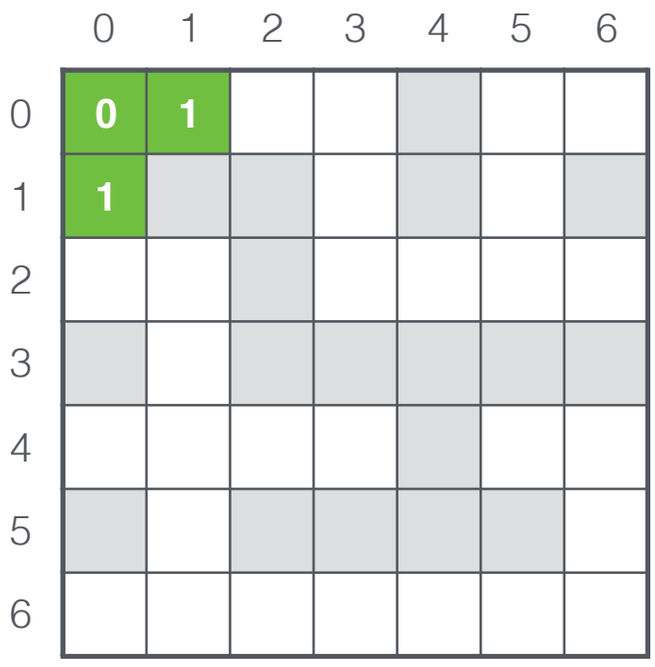
1. 하나의 큐를 만든다.
2. 위치 (0,0)는 이미 방문한 위치임을 표시하고, 큐에 위치 (0,0)을 넣는다.
3. 큐가 빌 때 까지 다음을 반복한다.
    1. 큐에서 하나의 위치 p를 꺼낸다.
    2. p에서 한 칸 떨어진 위치들 중에서 이동 가능하면서 아직 방문하지 않은 모든 위치들을 방문된 위치임을 표시하고 큐에 넣는다.
    3. 만약 그 위치가 출구라면 종료한다.

너비 우선 탐색으로 미로찾기



(0,0)

너비 우선 탐색으로 미로찾기



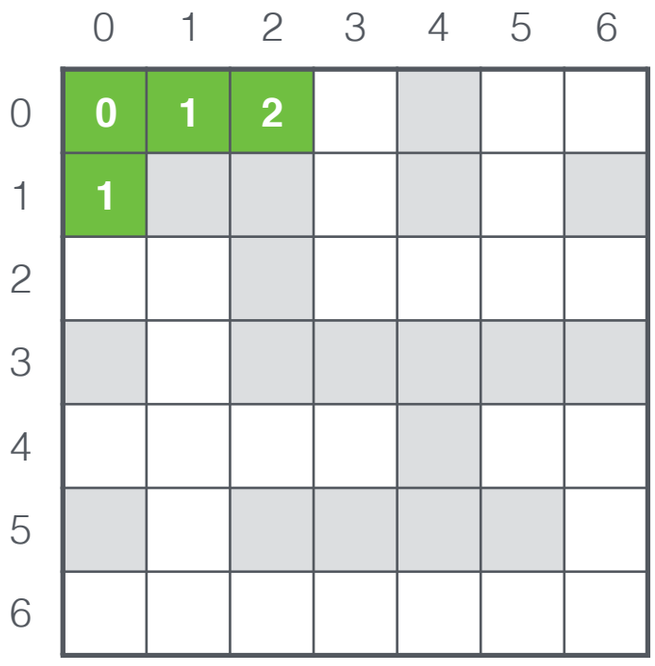
(0,0)

(0,0)

큐에서 (0,0)을 꺼낸 후 인접한 셀 중에서
아직 방문하지 않은 셀들인 (0,1)과 (1,0)을 방문하고
큐에 넣는다.

(0,1) **(1,0)**

너비 우선 탐색으로 미로찾기



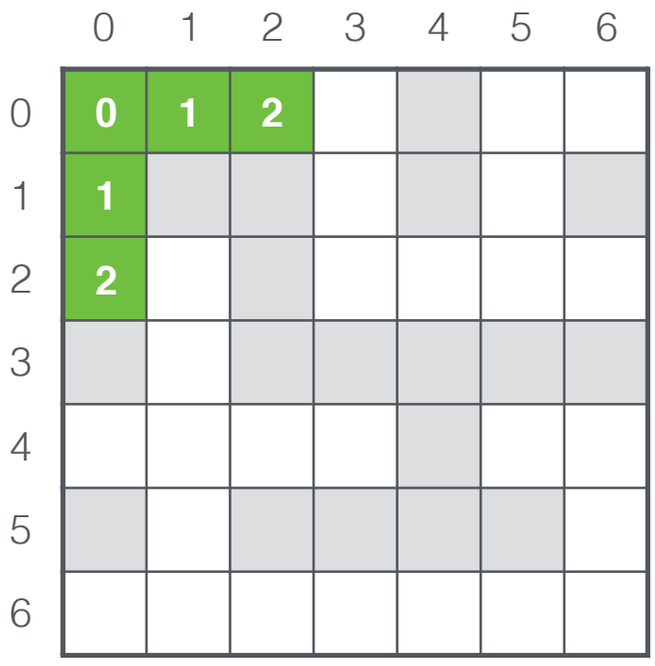
(0,1) (1,0)

(0,1)

큐에서 (0,1)을 꺼낸 후 인접한 셀 중에서 아직 방문하지 않은 셀인 (0,2)를 방문하고 큐에 넣는다.

(1,0) (0,2)

너비 우선 탐색으로 미로찾기



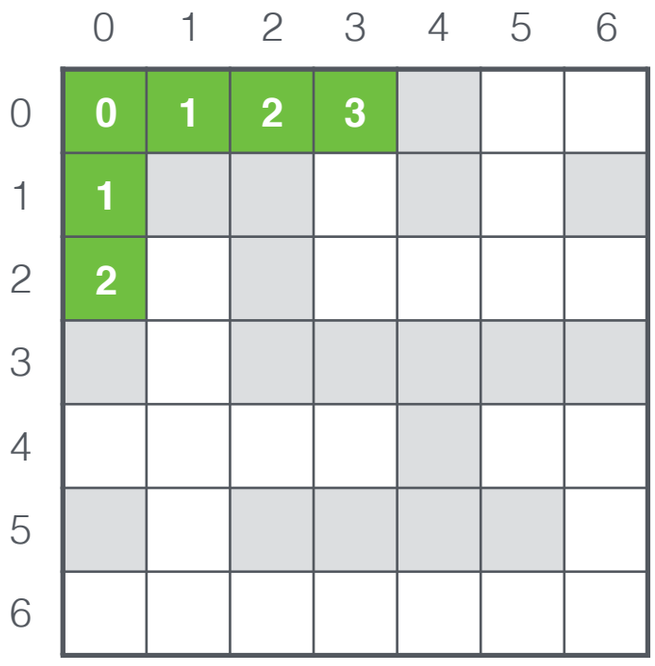
(1,0) **(0,2)**

(1,0)

큐에서 (1,0)을 꺼낸 후 인접한 셀 중에서 아직 방문하지 않은 셀인 (2,0)를 방문하고 큐에 넣는다.

(0,2) **(2,0)**

너비 우선 탐색으로 미로찾기



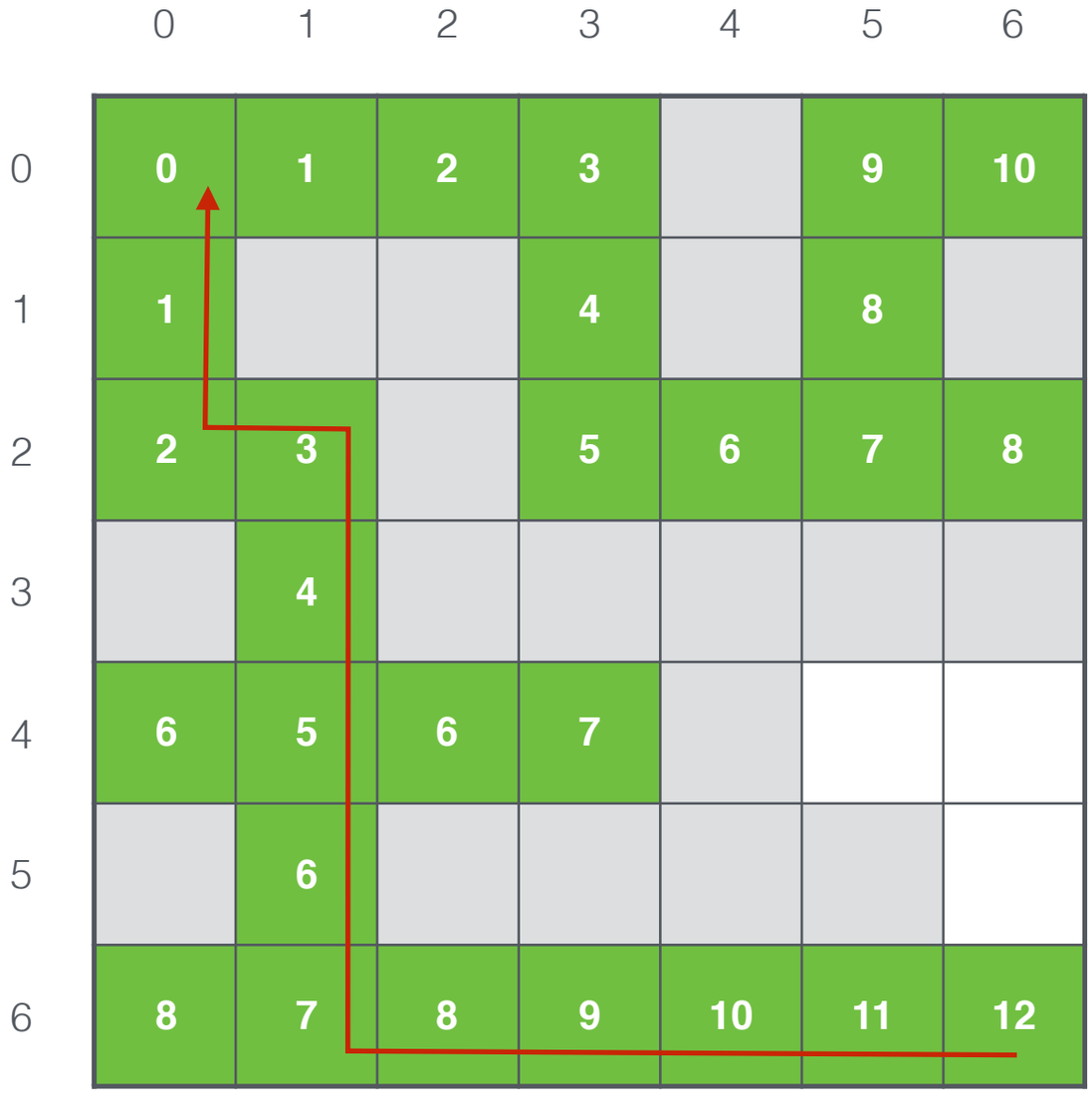
(0,2) **(2,0)**

(0,2)

큐에서 (0,2)를 꺼낸 후 인접한 셀 중에서 아직 방문하지 않은 셀인 (0,3)을 방문하고 큐에 넣는다.

(2,0) **(0,3)**

최종결과



출구에서 숫자가 감소하는 방향으로 따라가면
입구에 도달한다.

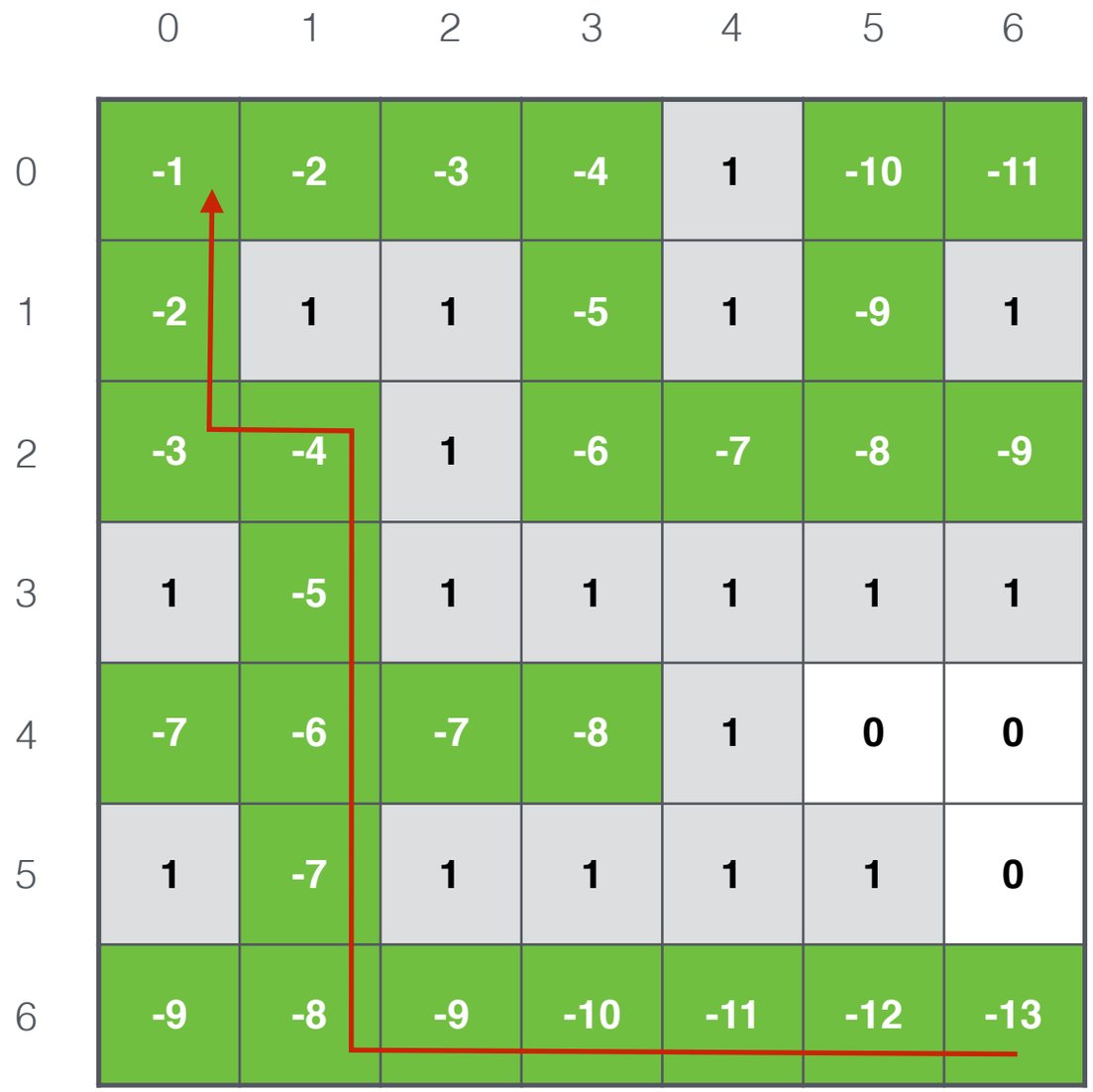
배열에 저장된 값은
출발점에서 그 지점까지의
최단경로의 길이이다.
왜?

maze.c

```
Queue queue = create_queue();
Position cur;
cur.x = 0;
cur.y = 0;

enqueue(queue, cur);

maze[0][0] = -1;
bool found = false;

while(!is_empty(queue)) {
    Position cur = dequeue(queue);
    for (int dir=0; dir<4; dir++) {
        if (movable(cur, dir)) {
            Position p = move_to(cur, dir);

            maze[p.x][p.y] = maze[cur.x][cur.y] - 1;

            if (p.x == n-1 && p.y == n-1) {
                printf("Found the path.\n");
                found = true;
                break;
            }
            enqueue(queue, p);
        }
    }
}
```

추가 배열을 쓰지 않기 위해서 방문 표시를 음수로 저장한다.

프로그램의 실제 최종결과



출구에서 숫자가 1 증가하는 방향으로 따라가면
입구에 도달한다.

큐의 변형

- **Deque (Double Ended Queue)**
  - 양 쪽 끝에서 삽입과 삭제가 허용되는 큐
  - "덱" 혹은 "디큐"라고 읽음
- **우선순위 큐 (priority queue)**
  - 큐에 들어온 순서와 무관하게 큐에 저장된 값들 중에서 가장 큰 값이 (혹은 가장 작은 값이) 가장 먼저 꺼내지는 큐
  - 대표적인 구현 방법으로는 이진 힙(binary heap)이 있음