자료구조 (Data Structure)
4주차: 큐

큐

- First In First Out (FIFO) 자료구조
- 가장 먼저 들어온 데이터가 가장 먼저 나간다
- 데이터 추가/삭제는 반대쪽 끝에서 일어난다

오늘 목표

- 단방향 링크드 리스트로 구현한 큐
- 배열로 구현한 큐
- 미로 찾기

필수 헤더

- #include <stdio.h>
- #include <stdlib.h>

링크드 리스트-큐 구조체

- 더미 노드를 쓰면 추가/삭제가 쉬워지므로 제일 앞에 더미 노드를 항상 두는 방식으로 변경

- front: 최근에 dequeue된 노드 (초기값 더미 노드)
- rear: 최근에 추가된 노드 (초기값 더미 노드)
- size: 데이터 개수 (초기값 0)

- LQUEUE_ENQUEUE: 새 노드를 rear 뒤에 연결
- LQUEUE_DEQUEUE: front 삭제 / 데이터 반환

링크드 리스트-큐 구조체 C 코드

```
struct lqueue {
        struct node *front;
        struct node *rear;
        int size;
};
```

단방향 노드 개요

- 데이터와 다음 데이터 정보의 묶음

- next: 다음 노드 정보
- value: 저장된 데이터

- NODE_CREATE: 노드 생성하는 함수

단방향 노드 C 코드

```
struct node {
        struct node *next;
        int value;
};
```

NODE_CREATE(value, next) C 코드

```
struct node *node_create(int value, struct node *next)
{
    struct node *node = malloc(sizeof(struct node));
    if (!node) exit(1);
    node->value = value;
    node->next = next;
    return node;
}
```

LQUEUE_CREATE() 개요

- 비어 있는 큐를 만들어서 반환한다

- 동작: q에 새로운 LQUEUE를 저장한다

  dummy에 NODE_CREATE(0, NULL)을 저장한다

  q의 front에 dummy를 저장한다

  q의 rear에 dummy를 저장한다

  q의 size에 0을 저장한다

- 반환: q

LQUEUE_CREATE() C 코드

```
struct lqueue *lqueue_create()
{
    struct lqueue *q = malloc(sizeof(struct lqueue));
    if (!q) exit(1);
    q->front = q->rear = node_create(0, NULL); // 더미
    q->size = 0;
    return q;
}
```

LQUEUE_ENQUEUE(q, value)

- 데이터가 value인 노드를 만들어 rear 뒤에 추가한다
- 조건: q가 실제 큐를 가리킨다
- 동작: node에 NODE_CREATE(value, NULL)를 저장

  q의 rear의 next에 node를 저장한다

  q의 rear에 node를 저장한다

  q의 size를 1 증가시킨다

LQUEUE_ENQUEUE(q, value) C 코드

```
void lqueue_enqueue(struct lqueue *q, int value)
{
    if (!q) exit(1);
    q->rear->next = node_create(value, NULL);
    q->rear = q->rear->next;
    ++q->size;
}
```

LQUEUE_DEQUEUE(q)

- 첫 데이터 노드를 더미로 돌리고 데이터를 반환한다.

- 조건: q의 front의 next 존재
- 동작: old_head에 q의 front를 저장한다

  q의 front에 old_head의 next를 저장한다

  old_head를 삭제한다

  q의 size를 1 감소시킨다
- 반환: q의 front의 value

LQUEUE_DEQUEUE(q) C 코드

```
int lqueue_dequeue(struct lqueue *q)
{
        if (!q) exit(1);
        struct node *old_head = q->front;
        if (!old_head || !old_head->next) exit(1);
        q->front = old_head->next;
        q->size--;
        free(old_head);
        return q->front->value;
}
```

배열로 구현한 큐 구조체

- 배열 인덱스의 범위는 1..capacity

- data: 데이터를 저장할 배열 (크기 capacity + 1)
- capacity: 배열의 최대 저장량
- front: 가장 오래된 데이터의 인덱스 (초기값 1)
- rear: 가장 최신 데이터의 인덱스 (초기값 0)
- size: 저장된 데이터 개수 (초기값 0)

배열-큐 구조체 C 코드

```
struct aqueue {
        int *data;
        int capacity;
        int front;
        int rear;
        int size;
};
```

SIMPLE_AQUEUE_ENQUEUE(q, value)

- q의 끝에 데이터 value를 추가한다
- 조건: q의 rear < q의 capacity
- 동작: q의 rear를 1 증가시킨다

  q의 data[q의 rear]에 value를 저장한다

  q의 size를 1 증가시킨다

SIMPLE_AQUEUE_ENQUEUE(q, value)

```
void simple_aqueue_enqueue(struct aqueue *q, int v)
{
    if (!q) exit(1);

    if (q->rear >= q->capacity) exit(1);

    q->data[++q->rear] = v;

    ++q->size;
}
```

SIMPLE_AQUEUE_DEQUEUE(q)

- q의 첫번째 데이터를 삭제하고 데이터를 반환한다

- 조건: q의 front ≤ q의 rear
- 동작: tmp에 q의 data[q의 front]를 저장한다

  q의 front를 1 증가시킨다

  q의 size를 1 감소시킨다
- 반환: tmp

SIMPLE_AQUEUE_DEQUEUE(q)

```
int simple_aqueue_dequeue(struct aqueue *q)
{
    if (!q || q->front > q->rear) exit(1);

    int tmp = q->data[q->front++];

    q->size--;

    return tmp;
}
```

단순하게 배열로 구현한 큐의 특징

- 초기 상태: front = 1, rear = 0, size = 0
- 비어있는 상태: 항상 size == 0, front > rear
- 크기: size == rear - front + 1
- 추가 불가 상태: rear == capacity

원형 배열로 구현한 큐

- 원형 배열: 인덱스가 capacity를 넘어가면
  다시 1이 된다

- size로 비어있음과 꽉차있음을 체크한다

CIRCULAR_AQUEUE_ENQUEUE(q, value)

- q의 끝에 데이터 value를 추가한다
- 조건: q의 size < q의 capacity
- 동작: q의 rear를 1 증가시킨다
  만약: q의 rear가 q의 capacity를 넘어갔으면
  q의 rear에 1을 저장한다
  q의 data[q의 rear]에 value를 저장한다
  q의 size를 1 증가시킨다

CIRCULAR_AQUEUE_ENQUEUE(q, value)

```
void circular_aqueue_enqueue(struct aqueue *q, int v)
{
    if (!q || q->size >= q->capacity) exit(1);

    ++q->rear;
    if (q->rear > q->capacity)
        q->rear = 1;
    q->data[q->rear] = ;
    ++q->size;
}
```

CIRCULAR_AQUEUE_DEQUEUE(q)

- q의 front 데이터를 삭제하고 반환한다

- 조건: q의 size > 0
- 동작: tmp에 q의 data[q의 front]를 저장한다
  q의 front를 1 증가시킨다
  만약: q의 front가 q의 capacity를 넘어갔으면
  q의 front에 1을 저장한다
  q의 size를 1 감소시킨다
- 반환: tmp

CIRCULAR_AQUEUE_DEQUEUE(q)

```
int circular_aqueue_dequeue(struct aqueue *q)
{
	if (!q || q->size == 0) exit(1);

	int tmp = q->data[q->front++];
	if (q->front > q->capacity)
		q->front = 1;
	q->size--;
	return tmp;
}
```

미로 정보: 2차원 배열 (배열의 배열)

- 미로[i][j]: (행 i, 열 j) 위치의 상태
- 0 : 통로(아직 처리하지 않음)
- 1 : 벽
- row*len + col : 최단경로에서 이전 좌표의 선형 index
- 0행과 0열은 모두 벽(1)로 두어, 내부 좌표에서 0/1과의 값 충돌을 방지함

좌표 구조체

- row: 행 좌표
- col: 열 좌표
- prev_size: 출발점부터의 거리 (출발점은 0)

좌표 구조체 C 코드

```
struct point {
        int row;
        int col;
        int prev_size;
};
```

MAZE_STATUS(maze, point)

- 반환: maze[point의 row][point의 col]

- 0 : 미방문 통로
- 1 : 벽
- ≥ len+1 : 최단거리 계산됨 - 이전 좌표의 선형 index
- -1 : 출발점 표시

MAZE_STATUS(maze, point) C 코드

```
int maze_status(int **maze, struct point point)
{
    return maze[point.row][point.col];
}
```

MAZE_MARK(maze, point, from)

- maze[point의 row][point의 col]에 from 저장

- 최단경로에서 이전 좌표가 (i, j)일 때 from == i * len + j
  // 일차적으로 처리가 됐다는 정보가 저장됨

MAZE_MARK(maze, point, from) C 코드

```
void maze_mark(int **maze, struct point point, int val)
{
    maze[point.row][point.col] = val;
}
```

좌표 선형 큐

- points: 좌표 구조체의 배열
- front: 꺼낼 좌표의 인덱스
- rear: 마지막으로 저장된 좌표의 인덱스

좌표 선형 큐 C 코드

```
struct point_queue {
        struct point *points;
        int front;
        int rear;
};
```

POINT_QUEUE_CREATE(capacity) C 코드

```
struct point_queue *point_queue_create(int cap)
{
    struct point_queue *q
        = malloc(sizeof(struct point_queue));
    if (!q) return NULL;

    q->points = malloc(sizeof(struct point) * (cap + 1));
```

POINT_QUEUE_CREATE(capacity) C 코드

```
    if (!q->points) {
        free(q);
        return NULL;
    }

    q->front = 1;
    q->rear = 0;
    return q;
}
```

POINT_QUEUE_FREE(q) C 코드

```
void point_queue_free(struct point_queue *q)
{
    if (!q) return NULL;
    free(q->points);
    free(q);
}
```

POINT_QUEUE_ENQUEUE(q, point) C 코드

```
void point_queue_enqueue(struct point_queue *q,
                                        struct point point)
{
    q->points[++q->rear] = point;
}
```

POINT_QUEUE_DEQUEUE(q) C 코드

```
struct point
point_queue_dequeue(struct point_queue *q)
{
	return q->points[q->front++];
}
```

NEXT_POINT(point, direction) 개요

- point의 좌표를 direction 방향으로 이동한 좌표 반환
- distance는 point의 distance에서 1 증가
- direction: 1(상), 2(하), 3(좌), 4(우)

NEXT_POINT(point, direction) C 코드

```
struct point next_point(struct point point, int direction)
{
        struct point next = point;
        next.distance++;
        if (direction == 1) {                    // 상
                next.row--;
        } else if (direction == 2) {             // 하
                next.row++;
```

NEXT_POINT(point, direction) C 코드

```
    } else if (direction == 3) {            // 좌
        next.col--;
    } else if (direction == 4) {            // 우
        next.col++;
    }
    return next;
}
```

PUSH_NBRS(maze, len, to_visit, point)

- point_index에 point의 row * len + point의 col 저장
- 반복: point의 상하좌우 좌표 next에 대하여

  만약: STATUS(maze, next) == 0 이면

  MARK(maze, next, point_index)

  ENQUEUE(to_visit, next)

PUSH_NBRS(maze, len, to_visit, point)

```
void push_nbrs (
        int **maze,
        int len,
        struct point_queue *to_visit,
        struct point point
) {
    int point_index = point.row * len + point.col;

    for (int direction = 1; direction <= 4; direction++) {
```

PUSH_NBRS(maze, len, to_visit, point)

```
            struct point next = next_point(point, direction);

            if (maze_status(maze, next) == 0) {
                maze_mark(maze, next, point_index);
                point_queue_enqueue(to_visit, next);
            }
        }
    }
```

FIND(maze, len, start, end) 개요 1/2

- to_visit에 CREATE(len * len) 저장 // 갈 곳들
- start의 distance에 0 저장
- MARK(maze, start, -1)
- ENQUEUE(to_visit, start)
- 반복: to_visit이 비어있지 않음

FIND(maze, start, end, len) C 코드 1/2

```
int find (
        int **maze,
        int len,
        struct point start,
        struct point end
) {
    struct point_queue *to_visit
        = point_queue_create(len * len);
    if (!to_visit) return -2;                    // 에러 코드
```

FIND(maze, start, end, len) C 코드 1/2

```
start.distance = 0;
maze_mark(maze, start, -1);
point_queue_enqueue(to_visit, start);

while (to_visit->front <= to_visit->rear) {
```

FIND(maze, len, start, end) 개요 2/2

point에 DEQUEUE(to_visit) 저장
만약: point와 end의 좌표가 동일
반환: point.distance
PUSH_NBRS(maze, len, to_visit, point)

- 반환: -1 (못찾음)

FIND(maze, start, end, len) C 코드 2/2

```
        struct point p = point_queue_dequeue(to_visit);
        if (p.row == end.row && p.col == end.col) {
            point_queue_free(to_visit);
            return point.distance;   // 경로 거리
        }
        push_nbrs(maze, len, to_visit, p);
    }
    point_queue_free(to_visit);
    return -1;
}
```

DECODE_PATH(maze, len, end, path_len)

path에 새 좌표 배열 저장 (크기: path_len + 1)
point에 end 저장
반복: i를 path_len - 1부터 0까지
    path[i]에 point를 저장한다
    prev_data에 STATUS(maze, point)를 저장한다
    point의 row에 prev_data / len를 저장한다
    point의 col에 prev_data % len를 저장한다
반환: path

DECODE_PATH(maze, len, end, path_len)

```
struct point *decode_path (
        int **maze,
        int len,
        struct point end,
        int path_len
) {
    struct point *path
        = malloc(sizeof(struct point) * (path_len + 1));
    if (!path) return NULL;
```

DECODE_PATH(maze, len, end, path_len)

```
    struct point point = end;
    for (int i = path_len; i >= 0; i--) {
        path[i] = point;
        int prev_data = maze_status(maze, point);
        if (prev_data < 0) break;
        point.row = prev_data / len;
        point.col = prev_data % len;
    }
    return path;
}
```

MAZE_TEST() 개요

- maze 이차원 배열에 미로 정보를 저장한다,
- m_row_ptrs에 maze의 행 배열들의 배열을 저장한다,
- 출발점, 도착점을 찾고, find()를 실행한다,
- find() 실행 결과에 따라 에러 처리를 한다,
- decode_path()를 실행한다,
- 결과를 화면에 출력한다

MAZE_TEST() C 코드

```
void maze_test() {
    int maze[7][7] = {
        {1, 1, 1, 1, 1, 1, 1},
        {1, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 1, 0, 1, 1},
        {1, 0, 1, 0, 0, 0, 1},
        {1, 0, 1, 0, 1, 0, 1},
        {1, 0, 0, 0, 1, 0, 1},
        {1, 1, 1, 1, 1, 1, 1},
    };
```

MAZE_TEST() C 코드

```
int *m_row_ptrs[7];
for(int i = 0; i < 7; i++)
        m_row_ptrs[i] = maze[i];

struct point start = {1, 1, 0};
struct point end   = {5, 5, 0};

int path_len = find(m_row_ptrs, 7, start, end);
```

MAZE_TEST() C 코드

```
if (path_len < 0) {
    if (path_len == -1) printf(no path found\\n);
    if (path_len == -2) printf(malloc failed\\n);
    return;
}

struct point *path
     = decode_path(m_row_ptrs, 7, end, path_len);
```

MAZE_TEST() C 코드

```
    for (int i = 1; i <= path_len; i++)
        printf((%d,%d), , path[i].row, path[i].col);
    printf(\n);

    //리소스 정리
    free(path);
}
```

요약

- 단방향 링크드 리스트로 구현한 스택에서 데이터 추가/삭제
- 배열로 구현한 스택에서 데이터 추가/삭제
- 미로 찾기 예제
- 다음주: 삭제할 때 추가된 순서대로인 큐