자료구조 (Data Structure)
3주차: 스택

스택

- Last In First Out (LIFO) 자료구조
- 가장 나중에 들어온 데이터가 가장 먼저 나간다
- 데이터 추가/삭제는 한쪽 끝에서만 일어난다

오늘 목표

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

단방향 링크드 리스트로 구현한 스택

- 처음 노드가 마지막 노드보다 추가/삭제가 쉽다
- top: 리스트의 처음 노드
- PUSH: 단방향 링크드 리스트의 INSERT_FIRST
- POP: 단방향 링크드 리스트의 DELETE_FIRST,
  이전 top의 데이터 반환

필수 헤더

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

단방향 노드 개요

- 데이터와 다음 데이터 정보의 묶음
- next: 다음 노드 정보
- value: 저장된 데이터

단방향 노드 C 코드

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

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;
}
```

링크드리스트-스택 구조체 개요

- top: 현재 스택에서 가장 최근에 추가된 노드
- size: 노드 개수(항상 $0 \le size$)
- 유효 구간: top부터 next를 따라 size개 노드

링크드리스트-스택 구조체 C 코드

```
struct lstack {
        struct node *top;
        int size;
};
```

LSTACK_INIT(stack) 개요

- stack을 빈 스택으로 초기화한다

- 동작: stack의 top을 NULL로 저장한다

  stack의 size를 0으로 저장한다

LSTACK_INIT(stack) C 코드

```
void lstack_init(struct lstack *stack)
{
    stack->top = NULL;
    stack->size = 0;
}
```

LSTACK_PUSH(stack, value) 개요

- 데이터가 value인 새 노드를 만들고 top 앞에 연결한다

- 동작: stack의 top에 node_create(value, stack의 top)
  을 저장한다
  stack의 size를 1 증가시킨다

LSTACK_PUSH(stack, value) C 코드

```
void lstack_push(struct lstack *stack, int value)
{
    stack->top = node_create(value, stack->top);
    stack->size++;
}
```

LSTACK_POP(stack) 개요

- stack의 top을 삭제하고, 데이터를 반환한다
- 조건: stack이 비어있지 않음
- 동작: old_top에 stack의 top을 저장한다

  v에 old_top의 value를 저장한다

  stack의 size를 1 감소시킨다

  stack의 top에 old_top의 next를 저장한다

  old_top을 삭제한다
- 반환: v

LSTACK_POP(stack) C 코드

```
int lstack_pop(struct lstack *stack)

{

        if (stack->size == 0) exit(1);

        struct node *old_top = stack->top;

        int v = old_top->value;
```

LSTACK_POP(stack) C 코드

```
    stack->size--;

    stack->top = old_top->next;

    free(old_top);

    return v;

}
```

LSTACK_CLEAR(stack) 개요

- stack에 남아 있는 모든 노드를 free하고 비운다
- 동작: 현재 top에서부터 next를 따라가며 노드 free
- 결과: top=NULL, size=0

LSTACK_CLEAR(stack) C 코드

```
void lstack_clear(struct lstack *stack)
{
    struct node *cur = stack->top;
    while (cur) {
        struct node *next = cur->next;
        free(cur);
```

LSTACK_CLEAR(stack) C 코드

```
        cur = next;
    }
    stack->top = NULL;
    stack->size = 0;
}
```

LSTACK_TEST() 개요

- LSTACK_PUSH, LSTACK_POP 기능을 확인해본다.
- 먼저 1, 2, 3을 차례로 PUSH한다
- POP한 결과를 출력하는 것을 3회 반복한다
- 기대 출력: 3 2 1

LSTACK_TEST() C 코드

```
void lstack_test()
{
    struct lstack s;
    lstack_init(&s);
    for (int i = 1; i <= 3; i++)
        lstack_push(&s, i); // 1, 2, 3
```

LSTACK_TEST() C 코드

```
    printf("%d ", lstack_pop(&s)); // 3

    printf("%d ", lstack_pop(&s)); // 2

    printf("%d\n", lstack_pop(&s)); // 1

    lstack_clear(&s);

}
```

퀴즈 — 링크드리스트 스택 PUSH

- 초기 상태: top->value == 2이고, top->size == 1.

  lstack_push(&s, 5) 호출 뒤 올바른 상태는?

  A) top->value = 5, size = 2

  B) top->value = 2, size = 1

  C) top->value = 2, size = 2

  D) top->value = 5, size = 1

배열-스택 구조체 개요

- 데이터를 배열의 오른쪽부터 순서대로 저장한다
- data: 배열
- capacity: 최대 저장량
- top: 마지막에 추가된 데이터의 인덱스

  (스택이 비어있으면, top == capacity + 1)

배열-스택 구조체 C 코드

```
struct astack {
        int *data;
        int capacity;
        int top;
};
```

ASTACK_INIT(stack, cap) 개요

- cap개 데이터를 저장가능하게 stack을 초기화한다
- 조건: cap > 0
- 동작: stack의 data에 새로운 배열을 저장한다

  (배열 크기: cap + 1)

  stack의 capacity에 cap을 저장한다

  stack의 top에 cap + 1을 저장한다

ASTACK_INIT(stack, cap) C 코드

```
void astack_init(struct astack *stack, int cap)
{
    if (cap <= 0) exit(1);
    stack->data = malloc(sizeof(int) * (cap + 1));
    if (!stack->data) exit(1);
    stack->capacity = cap;
    stack->top = cap + 1;
}
```

ASTACK_PUSH(stack, val) 개요

- stack에 val을 추가한다
- 조건: stack의 top > 1
- 동작: stack의 top을 1 감소시킨다

  stack의 data[stack의 top]에 val을 저장한다

ASTACK_PUSH(stack, val) C 코드

```
void astack_push(struct astack *stack, int val)
{
    if (stack->top == 1) exit(1);
    stack->data[--stack->top] = val;
}
```

ASTACK_POP(stack) 개요

- stack에서 top을 삭제하고 데이터를 반환한다
- 조건: stack이 비어있지 않음 (top ≤ capacity)
- 동작: v에 stack의 data[stack의 top]을 저장한다
  stack의 top을 1 증가시킨다
- 반환: v

ASTACK_POP(stack) C 코드

```
int astack_pop(struct astack *stack)
{
    if (stack->top == stack->capacity + 1) exit(1);
    return stack->data[stack->top++];
}
```

ASTACK_DATA_FREE(stack) 개요

- astack가 보유한 data 배열을 free하고 비운다
- 결과: data=NULL, capacity=0, top=1

ASTACK_DATA_FREE(stack) C 코드

```
void astack_data_free(struct astack *stack)
{
	int *to_free = stack->data;
	stack->capacity = 0;
	stack->data = NULL;
	stack->top = 1;
	free(to_free);
}
```

ASTACK_TEST() 개요

- ASTACK_PUSH, ASTACK_POP 기능을 확인해본다.
- 먼저 크기 3으로 INITIALIZE한다
- 먼저 10, 20, 30을 차례로 PUSH한다
- POP한 결과를 출력하는 것을 3회 반복한다
- 기대 출력: 30 20 10

ASTACK_TEST() C 코드

```
void astack_test()
{
    struct astack s;
    astack_init(&s, 3);
    for (int i = 10; i <= 30; i += 10)
        astack_push(&s, i); // 10, 20, 30
```

ASTACK_TEST() C 코드

```
    printf("%d ", astack_pop(&s)); // 30

    printf("%d ", astack_pop(&s)); // 20

    printf("%d\n", astack_pop(&s)); // 10

    astack_data_free(&s);

}
```

퀴즈 — 배열 스택의 초기 상태

- cap=2으로 astack_init(&s, 2) 직후 초기 스택일 때, s.top의 값은?

  A) 0

  B) 1

  C) 2

  D) 3

미로 정보: 2차원 배열 (배열의 배열)

- 미로[i][j]: (행 i, 열 j) 위치의 상태
- 0 : 가도 됨
- 1 : 갈 수 없거나 이미 갔음
- 처음 주어진 미로 정보를 수정해도 된다고 가정

CREATE_SAMPLE_MAZE() 개요

- 미로 샘플 2차원 배열을 만들어서 반환한다

- 동작: maze에 행 개수만큼 메모리 할당

  maze의 각 행에 열 개수만큼 메모리 할당

  maze의 테두리에 1 저장

  sample에 미로 샘플 저장 (2차원 배열 초기화)

  maze 안쪽에 sample 데이터 복사

  maze 반환

CREATE_SAMPLE_MAZE() C 코드

```
int **create_sample_maze()

{

    int **maze = malloc(sizeof(int *) * 7);

    if (!maze) return NULL;

    for (int i = 0; i < 7; i++) {

        maze[i] = malloc(sizeof(int) * 7);
```

CREATE_SAMPLE_MAZE() C 코드

```
        if (!maze[i]) {
            for (int j = 0; j < i; j++)
                free(maze[j]);
            free(maze);
            return NULL;
        }
}
```

CREATE_SAMPLE_MAZE() C 코드

```
for (int i = 0; i < 7; i++) {
        maze[0][i] = 1;
        maze[6][i] = 1;
        maze[i][0] = 1;
        maze[i][6] = 1;
}
```

CREATE_SAMPLE_MAZE() C 코드

```
int sample[5][5] = {
        {0, 1, 0, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 0, 0},
};
```

CREATE_SAMPLE_MAZE() C 코드

```
for (int i = 1; i <= 5; i++) {
        int i_s = i - 1;
        for (int j = 1; j <= 5; j++) {
                int j_s = j - 1;
                maze[i][j] = sample[i_s][j_s];
        }
}
```

CREATE_SAMPLE_MAZE() C 코드

```
    return maze;
}
```

MAZE_FREE(maze, nrows) 개요

- 2차원 배열 maze의 각 행을 free하고, 마지막에 행 포인터 배열을 free한다
- 인자: maze(행 포인터 배열), nrows(행 개수)
- 예) create_sample_maze()에서 7×7 이차원 배열을 만들었다면 nrows=7

MAZE_FREE(maze, nrows) C 코드

```
void maze_free(int **maze, int nrows)
{
    if (!maze) return;
    for (int i = 0; i < nrows; i++)
        free(maze[i]);
    free(maze);
}
```

좌표 구조체

- row: 행 좌표
- col: 열 좌표
- prev_size: 현재 점을 탐색 경로에 추가하기 직전에 경로의 길이 (시작점은 0)

좌표 구조체

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

MAZE_STATUS(maze, point) 개요

- 반환: maze[point의 row][point의 col]
- 0 : 아직 처리하지 않은 통로
- 1 : 벽/이미 방문/방문 예정 (모두 방문 계획 금지)

MAZE_STATUS(maze, point) C 코드

```
int maze_status(int **maze, struct point point)
{
    return maze[point.row][point.col];
    // 0 : 방문 계획 가능
    // 1 : 방문 계획 금지
}
```

MAZE_MARK(maze, point) 개요

- maze[point의 row][point의 col]에 1 저장
- 방문했거나 방문 예정 표시

MAZE_MARK(maze, point) C 코드

```
void maze_mark(int **maze, struct point point)
{
    maze[point.row][point.col] = 1;
    // 추가 방문 계획 금지 표시
}
```

좌표 리스트/스택 구조체 개요

- points: 좌표 구조체의 배열
- size: 저장된 좌표 개수
- capacity: 최대 저장량

좌표 리스트/스택 구조체 C 코드

```
struct alstack {
        struct point *points;
        int size;
        int capacity;
};
```

ALSTACK_CREATE(cap) 개요

- list에 새 좌표 리스트/스택 저장
- list의 points에 새 배열 저장 (크기: cap + 1)
- list의 capacity에 cap 저장
- list의 size에 0 저장
- 반환: list

ALSTACK_CREATE(cap) C 코드

```
struct alstack *alstack_create(int cap)
{
    if (cap <= 0) return NULL;
    struct alstack *list = malloc(sizeof(struct alstack));
    if (!list) return NULL;
    list->points
        = malloc(sizeof(struct point) * (cap + 1));
```

ALSTACK_CREATE(cap) C 코드

```
if (!list->points) {
        free(list);
        return NULL;
}
list->capacity = cap;
list->size = 0;
```

ALSTACK_CREATE(cap) C 코드

```
    return list;
}
```

ALSTACK_PUSH(list, point) 개요

- 조건: list의 size < list의 capacity
- size 1 증가
- list의 points[size]에 point 저장

ALSTACK_PUSH(list, point) C 코드

```
void alstack_push(struct alstack *list,
                  struct point point)
{
    if (list->size == list->capacity) exit(1);
    list->points[++list->size] = point;
}
```

ALSTACK_POP(list) 개요

- 조건: list의 size > 0
- tmp에 list의 points[size] 저장
- list의 size 1 감소
- 반환: tmp

ALSTACK_POP(list) C 코드

```
struct point alstack_pop(struct alstack *list)

{

        if (list->size == 0) exit(1);

        return list->points[list->size--];

}
```

ALSTACK_CUT(list, new_size) 개요

- 조건: 0 ≤ new_size ≤ list의 size
- list의 size를 new_size로 변경

ALSTACK_CUT(list, new_size) C 코드

```
void alstack_cut(struct alstack *list, int new_size)
{
    if (new_size < 0 || new_size > list->size) exit(1);
    list->size = new_size;
}
```

ALSTACK_FREE(list) 개요

- alstack가 보유한 points 배열과 자신을 free한다

- 주의: alstack은 alstack_create()로 동적 할당됨,
  따라서, free(list) 필요

ALSTACK_FREE(list) C 코드

```
void alstack_free(struct alstack *list)
{
    if (!list) return;

    free(list->points);

    list->points = NULL;

    free(list);

}
```

PUSH_NBRS(maze, to_visit, point) 개요

- n_added에 0 저장
- 반복: point의 상하좌우 좌표 next에 대하여
  만약: STATUS(maze, next) == 0 이면
    // next의 prev_size는 point까지 경로 길이
    next의 prev_size에 point의 prev_size + 1 저장
    maze_mark(maze, next)
    alstack_push(to_visit, next)
    n_added 1 증가
- 반환: n_added // 추가된 좌표 개수

NEXT_POINT(point, direction) 개요

- point의 좌표를 direction 방향으로 이동한 좌표 반환
- direction: 1(상), 2(하), 3(좌), 4(우)
- prev_size는 변경하지 않음

NEXT_POINT(point, direction) C 코드

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

    if (direction == 1) {

        next.row--;

    } else if (direction == 2) {

        next.row++;
```

NEXT_POINT(point, direction) C 코드

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

PUSH_NBRS(maze, to_visit, point) C 코드

```
int push_nbrs(int **maze, struct alstack *to_visit,
                        struct point point)
{
        int n_added = 0;

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

                struct point next = next_point(point, direction);

                if (maze_status(maze, next) == 0) {

                        maze_mark(maze, next);
```

PUSH_NBRS(maze, to_visit, point) C 코드

```
                next.prev_size = point.prev_size + 1;

                alstack_push(to_visit, next);

                n_added++;

            }

        }

        return n_added;

}
```

FIND(maze, start, end, len) 개요 1/4

- to_visit에 alstack_create(len * len) 저장 // 갈 곳들
- path에 alstack_create(len * len) 저장 // 경로 기록
- going_back에 0 저장 // 막다른 곳 체크
- start의 prev_size에 0 저장
- maze_mark(maze, start)
- alstack_push(to_visit, start)

FIND(maze, start, end, len) C 코드 1/4

```
struct alstack *find(int **maze, struct point start,
                                    struct point end, int len)
{
        struct alstack *to_visit = alstack_create(len * len);

        if (!to_visit) return NULL;

        struct alstack *path = alstack_create(len * len);

        if (!path) {

                alstack_free(to_visit);
```

FIND(maze, start, end, len) 1/4

```
        return NULL;
}
int going_back = 0;
start.prev_size = 0;
maze_mark(maze, start);
alstack_push(to_visit, start);
```

FIND(maze, start, end, len) 2/4

- 반복: to_visit이 비어있지 않음

  point에 alstack_pop(to_visit) 저장

  만약: going_back == 1

    alstack_cut(path, point의 prev_size)

  alstack_push(path, point)

FIND(maze, start, end, len) 2/4

```
while (to_visit->size > 0) {
    struct point point = alstack_pop(to_visit);
    if (going_back == 1) {
        alstack_cut(path, point.prev_size);
    }
    alstack_push(path, point);
```

FIND(maze, start, end, len) 3/4

만약: point와 end의 좌표가 동일

    alstack_free(to_visit)

    반환: path // 도착점 포함 경로

n에 push_nbrs(maze, to_visit, point) 저장

만약: n == 0

    going_back에 1 저장 // 막다른 곳 처리 예정

FIND(maze, start, end, len) 3/4

```
if (point.row == end.row
    && point.col == end.col) {
      alstack_free(to_visit);
      return path; // 도착점 포함 경로
}
```

FIND(maze, start, end, len) 3/4

```
int n = push_nbrs(maze, to_visit, point);

if (n == 0) {

        going_back = 1; // 막다른 곳 처리

}
```

FIND(maze, start, end, len) 4/4

그외:

going_back에 0 저장

다음 반복으로

- alstack_free(to_visit) // path는 호출 함수에서 free
- alstack_cut(path, 0) // path에 못찾음 표시
- 반환: path

FIND(maze, start, end, len) 4/4

```
        else {
            going_back = 0;
        }
    }

    alstack_free(to_visit);

    alstack_cut(path, 0); // 못찾음

    return path;
}
```

MAZE_TEST() 개요

- 시작점 (1,1) → 도착점 (5,5) 경로를 찾는다.

MAZE_TEST() C 코드

```
void maze_test()
{
    int **maze = create_sample_maze();
    if (!maze) {
        printf("maze create failed\n");
        return;
    }
```

MAZE_TEST() C 코드

```
struct point start = {1, 1, 0};
struct point end   = {5, 5, 0};
struct alstack *path = find(maze, start, end, 5);
if (!path) {
    printf("path alloc failed\n");
    maze_free(maze, 7);
    return;
}
```

MAZE_TEST() C 코드

```
if (path->size == 0) {
    printf("no path\n");
} else {
    for (int i = 1; i <= path->size; i++)
        printf("(%d,%d), ", path->points[i].row,
                path->points[i].col);
    printf("\n");
}
```

MAZE_TEST() C 코드

```
    //리소스 정리
    maze_free(maze, 7);
    alstack_free(path);
}
```

퀴즈 — 이웃 처리 순서(push_nbrs)

- push_nbrs에서 이웃에 대한 올바른 처리 순서는?

  A) PUSH, MARK, prev_size 설정

  B) STATUS 검사, PUSH, prev_size 설정, MARK

  C) STATUS 검사, MARK, prev_size 설정, PUSH

  D) STATUS 검사, MARK, PUSH

퀴즈 — 자원 정리

- maze_test() 종료 시 올바른 정리 순서로 옳은 것은?

  A) free(maze)만 호출한다

  B) alstack_free(path) 후 maze_free(maze, 7)

  C) maze_free(maze, 7) 후
  free(path->points)만 호출

  D) 아무 것도 free하지 않는다

요약

- 단방향 링크드 리스트로 구현한 스택에서 데이터 추가/삭제
- 배열로 구현한 스택에서 데이터 추가/삭제
- 미로 찾기 예제