강의명: 자료구조

숙제 번호: 4

숙제 제목: Stacks(스택)

학생 이름: 김성현

학번: 201910783

1. Stack(스택)

1.1

// ======================================================================

void init\_stack(STACK \* s)

{

s->top = -1;

}

// ======================================================================

void print\_stack(STACK \* s)

{

printf("(");

printf("%d : ", s->top+1);

for(int i = 0; i < MAX\_STACK\_SIZE; i++)

{

if(s->top == -1)

break;

printf("%d", s->data[i]);

if (i == s->top)

break;

printf(", ");

}

printf(")");

printf("\n");

}

// ======================================================================

int empty\_stack(STACK \* s)

{

if (s->top == -1) return 1;

else return 0;

}

// ======================================================================

int push\_stack(STACK \* s, int item)

{

if (s->top == (MAX\_STACK\_SIZE - 1))

return ERROR;

else

{

s->data[++(s->top)] = item;

return s->top;

}

}

// ======================================================================

int pop\_stack(STACK \* s)

{

int emy = 0;

emy = empty\_stack(s);

if (emy == 1)

return ERROR;

else if (emy == 0)

{

return s->data[(s->top)--];

}

}

\*<이렇게 프로그램한 이유>

: 먼저 스택 초기화 함수인 init\_stack(STACK \* s)함수는 스택이 배열형태로 되어 있고 안에 있는 element의 최상을 나타내는 top을 element가 아무것도 없으므로 배열의 시작(스택의 시작)인 0 바로 밑인 -1을 top에 주면서 초기화 시켰습니다.

그 다음 void print\_stack(STACK \* s)함수는 스택의 출력 함수이기에 형식에 맞춰 먼저 괄호를 열고 item 개수를 출력하기 위해 for문을 시작하기 전에 printf문 두 개로 출력하였습니다. 여기서 item 개수는 top index의+1이기 때문에 top+1을 넣었습니다.그 이후 스택 안에 있는 item을 전부 출력하기 위해 for문을 스택의 최대 크기 만큼 반복시켰고, 본격적으로 안에 있는 item들을 출력하기 전 스택이 비어있는 경우에는 if문에 top이 -1일 경우 바로 for문을 빠져나가도록 break하였습니다. 그 이후 별다른 사항이 없을 경우 bottom에서 top으로 출력하기 위해 i를 인덱스 삼아 printf로 item들을 출력시켰고 item의 끝, 즉 top에 도달했을 경우 item 출력을 종료하기 위해 if문에 조건으로 i가 top에 도달했을 경우 break로 빠져나갔습니다. 이렇게 빠져나가고 난 후 괄호를 닫아 출력을 종료하기 위해 printf로 닫는 괄호를 출력하고 \n으로 다음 라인으로 넘겼습니다.

그 다음 int empty\_stack(STACK \* s) 함수는 스택이 비었는지 확인하는 함수이기에 스택이 빈 경우인 top이 -1인 경우를 if문으로 주어 true일 경우 empty라고 판정하여 1을 return, empty가 아닌 경우는 0을 return하도록 하였습니다.

그 다음 int push\_stack(STACK \* s, int item)함수는 먼저 스택이 꽉 차 있을 경우에는 push를 하면 안되므로 top의 index가 배열의 index이니 MAX\_STACK\_SIZE – 1과 같은지를 if문 조건으로 주어 true인 경우 ERROR를 return하고, 그 외 정상적으로 push할 수 있을 때는 data배열에 먼저 top을 1 증가시켜 공간을 확보하고 그 다음 item을 넣는 식인 s->data[++(s->top)] = item 코드를 사용했습니다. 그 다음 문제 요구에 맞게 top을 return시켰습니다.

마지막으로 int pop\_stack(STACK \* s)함수는 먼저 empty\_stack함수로 스택이 비었는지 확인, 비었을 경우에는 pop을 하면 안되는 것을 구현하기 위해 if문에 조건으로 넣어 스택이 빈 경우 ERROR를 return하고, 그 외 정상적으로 스택이 있는 경우 즉 empty\_stack의 return값 emy가 0인 경우에는 먼저 item을 지우고 그 다음 스택 공간 top을 한 단계 없애는 식의 s->data[(s->top)--]코드를 사용하였습니다.

// ======================================================================

1.2

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "stack.h"

// ======================================================================

int main(void)

{

int i, r;

STACK s1;

printf("===== test-stack =====\n");

init\_stack(&s1);

print\_stack(&s1);

for (i = 0; i < MAX\_STACK\_SIZE + 100; i++) {

if (push\_stack(&s1, i) == ERROR)

break;

print\_stack(&s1);

}

for (i = 0; i < MAX\_STACK\_SIZE + 100; i++) {

if (pop\_stack(&s1) == ERROR)

break;

print\_stack(&s1);

}

r = empty\_stack(&s1);

if (r != 1)

printf("ERROR: stack\_empty\n");

else

printf("OK: stack\_empty\n");

return EXIT\_SUCCESS;

}

\*<분석 후 설명>

: test-stack.c를 분석해보면 먼저 for문으로 i를 0부터 MAX\_STACK\_SIZE인 12 + 100 = 112까지 스택에 push하고 push할 때마다 출력하는데, 중간에 push함수가 ERROR를 return하는 경우, 즉 MAX\_STAK\_SIZE인 12개가 꽉 찬 경우에는 if문으로 break하여 for문을 종료시킵니다.그 다음 비슷한 for문으로 pop하여 스택이 완전히 빌 때까지 pop할 때마다 그 결과를 출력합니다. 스택이 비는 순간은 if문-break로 잡습니다. 그 다음 최종 체크로 스택이 완전히 비었는지를 empty\_stack함수로 체크하여 return값이 1이 아니면, 즉 안 비었으면 ERROR를 출력, return값이 1이면 정상적으로 됬다고 출력합니다.

// ======================================================================

1.3

S1910783@oak:hw04$ gcc stack.c test-stack.c -o test-stack

s1910783@oak:hw04$ ./test-stack

===== test-stack =====

(0 : )

(1 : 0)

(2 : 0, 1)

(3 : 0, 1, 2)

(4 : 0, 1, 2, 3)

(5 : 0, 1, 2, 3, 4)

(6 : 0, 1, 2, 3, 4, 5)

(7 : 0, 1, 2, 3, 4, 5, 6)

(8 : 0, 1, 2, 3, 4, 5, 6, 7)

(9 : 0, 1, 2, 3, 4, 5, 6, 7, 8)

(10 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

(11 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

(12 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

(11 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

(10 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

(9 : 0, 1, 2, 3, 4, 5, 6, 7, 8)

(8 : 0, 1, 2, 3, 4, 5, 6, 7)

(7 : 0, 1, 2, 3, 4, 5, 6)

(6 : 0, 1, 2, 3, 4, 5)

(5 : 0, 1, 2, 3, 4)

(4 : 0, 1, 2, 3)

(3 : 0, 1, 2)

(2 : 0, 1)

(1 : 0)

(0 : )

OK: stack\_empty

2. Matching parenthesis(괄호 검사)

2.1

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "match.h"

// ======================================================================

void match(char \*expr)

{

STACK s;

char ch, open\_ch;

int i, n = strlen(expr);

init\_stack(&s);

int r = 0;

int last = 0;

for(i = 0; i < n; i++) {

ch = expr[i];

switch (ch) {

case '(': case '[': case'{':

push\_stack(&s, ch);

break;

case ')': case ']': case'}':

if (empty\_stack(&s) == 1)

{

r = i;

break;

}

else

{

open\_ch = pop\_stack(&s);

if ((open\_ch == '(' && ch != ')') || (open\_ch == '[' && ch != ']') || (open\_ch == '{' && ch != '}'))

{

r = i;

}

break;

}

last = i;

}

}

if (empty\_stack(&s) != 1)

printf("FALF at %d \n", n);

else

{

if (r == 0)

printf("MATCH \n");

else

printf("FALF at %d \n", r);

}

}

// ======================================================================

이렇게 프로그램한 이유

: void match(char \*expr)함수는 argument로 들어온 문자열의 괄호 검사함수입니다. 먼저 사전준비로 스택 전용 s와 문자열 각각 비교를 위한 ch, pop한 글자 확인을 위한 open\_ch, 문자열 길이 저장을 위한 변수 n, 에러사항일 경우 그 사항의 스택index를 저장하기 위한 r, last를 선언하였습니다. 여기서 argument로 들어온 문자열 길이 측정을 위해 보다 간편하고 실용적인 문자열 길이 계산 함수인 string.h 라이브러리의 strlen()함수를 사용하였습니다.

그 다음 괄호 검사 알고리즘에 맞춰 for반복문을 일일히 반복하여 비교하기 위해 사용하였고, 각 순환마다 ch에 expr의 다음 문자를 넣었습니다. 그 다음 여는 괄호, 닫는 괄호 구분을 위해 ch를 swtich에 넣어 case를 여는 괄호 3개와 닫는 괄호 3개로 나누었습니다. 여는 괄호 3개의 case에는, 즉 여는 괄호가 나오면 그 문자를 스택에 push하여 나중에 꺼내어 비교할 수 있게 하였습니다. 그 다음 닫는 괄호 case, 즉 닫는 괄호가 나오면 일단 스택이 비어있으면 빼박 에러이기 때문에 if문 조건에 empty\_stack함수를 넣어 1이 return된다면 해당 문자열 index인 i를 임시 저장소인 r에 저장시켰습니다. 그 외 스택이 비어있지 않고 비교를 해야하는 경우에는 일반 스택에 있는 거를 차례차례 꺼내어 비교하기 위해 open\_ch에 저장하고 if문 조건에 open\_ch에 각 여는 괄호들을 각각 경우를 부여하고 ch와 비교하기 위해 이를 &&연잔자를 사용해 각 짝이 맞는 닫는 괄호와 비교하는 총 3개의 경우를 or(||)연산자로 하여 만약 맞지 않으면 이 문자열의 index에 해당하는 i를 임시 저장소 r에 저장하였습니다. 그 이후 최종 체크로 스택이 비었는지를 체크, 스택이 비어있지 않으면 닫는 괄호가 끝까지 안나온 경우이니 문자열의 끝부분에 문제가 있는 것이기에 문자열의 끝부분에 해당하는 n에서 실패했다고 출력했고, 그 외 스택이 빈 경우에는 r이 index에 해당하는 i가 아닌 초기값인 0 그대로인 경우에는 정상적인 것이기 때문에 MATCH를 출력, 만약 이럴 경우가 아닌 경우, 즉 r에 index i가 들어가 있는 경우에는 i번째에서 실패했기 때문에 이 FALF in %d🡺i를 출력하였습니다.

2.2

s1910783@oak:hw04$ gcc match.c stack.c test-match.c -o test-match

s1910783@oak:hw04$ ./test-match

expr1=[{()}]

MATCH

expr2=[{(]}]

FALF at 3

expr3=[({}()[])]{{([]{()}())}}

MATCH

expr4=[({}()[]}]

FALF at 8

expr4=[[({}()[])]

FALF at 11

3. Maze problem(미로 문제)

3.1

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "maze.h"

// ======================================================================

void print\_maze(MAZE \* maze)

{

for (int r = 0;r < MAX\_MAZE\_SIZE; r++) {

for (int c = 0; c < MAX\_MAZE\_SIZE; c++) {

if (maze->map[maze->here\_r][maze->here\_c] == 'x') {

maze->map[maze->here\_r][maze->here\_c] = '\*';

printf("%c", maze->map[r][c]);

maze->map[maze->here\_r][maze->here\_c] = 'x';

continue;

}

maze->map[maze->here\_r][maze->here\_c] = '\*';

printf("%c", maze->map[r][c]);

}

printf("\n");

}

}

// ======================================================================

void push\_loc(MAZE \* maze, int r, int c)

{

if (r < 0 || c < 0) return;

if (maze->map[r][c] != '1' && maze->map[r][c] != '.') {

push\_stack(maze->stack, r);

push\_stack(maze->stack, c);

print\_stack(maze->stack);

}

}

// ======================================================================

int run\_maze(MAZE \* maze)

{

int r, c;

while (maze->map[maze->here\_r][maze->here\_c] != 'x') {

r = maze->here\_r;

c = maze->here\_c;

maze->map[r][c] = '.';

push\_loc(maze, r, c + 1);

push\_loc(maze, r + 1, c);

push\_loc(maze, r, c - 1);

push\_loc(maze, r - 1, c);

if (empty\_stack(maze->stack) == 1) {

return FAILED;

}

else {

maze->here\_c = pop\_stack(maze->stack);

maze->here\_r = pop\_stack(maze->stack);

return FINDING;

}

}

return FOUND;

}

// ======================================================================

이렇게 프로그램한 이유

: 먼저 void print\_maze(MAZE \* maze)함수는 maze에 포함된 map을 출력하는 함수이기에 map은 2차원 배열로 구성되어 있는 것을 출력하려면 for문 중첩을 해야 하기 때문에 MAX\_MAZE\_SIZE에 맞춰서 for문 중첩을 사용하였습니다. 그리고 만약 maze의 현 위치가 출구인 x인 경우 출구 부분을 현재위치표시는 \*로 바꿔주고 출력하고 다시 x로 바꿔주는 형식을 반복하여 출구에 도달했다라는 것을 구현했습니다. 그외 정상적인 진행상황에서는 현재 위치를 \*로 바꾼 다음 행인 r과 열인 c를 c먼저 쫙 출력하고 나서 다음 r행으로 넘어가서 다시 c를 쫙 출력하고 하는 식으로 maze->map[r][c]를 출력하였습니다. 그러고 나서 \n으로 다음 줄로 넘어갔습니다.

그 다음 void push\_loc(MAZE \* maze, int r, int c)함수는 들어오는 r,c의 위치 정보를 받아서 스택에 넣을 수 있는지를 체크해서 넣을 수 없으면 그냥 넘어가고 넣을 수 있으면 스택에 push해버리는 함수입니다. 먼저 r,c가 음수인 경우는 map에 아예 없는 못가는 곳이므로 바로 return하여 끝내버립니다. 이 경우가 아닌 경우에는 2차로 r,c위치정보가 갈 수 없는 벽이거나 현재 위치인 .인 경우가 아닌 경우에는 push를 하기 위해 조건문에 !=연산자와 두 경우를 모두 만족해야하니 &&연산자를 사용하여 이 조건을 충족하는 경우 r,c 순서로 차례대로 push\_stack함수를 이용하여 maze의 스택에 집어 넣습니다. 그런 다음 문제 조건에 맞게 print\_stack을 부르기 위해 if문 밖에서 불렀습니다.

그 다음 마지막이자 가장 중요한 int run\_maze(MAZE \* maze)함수는 maze안에 있는 정보로 maze를 한 단계 진행시키는 함수입니다. 이를 위해 while반복문을 통해 maze의 현재위치정보 here\_r, here\_c가 출구인 ‘x’를 만나지 않을 경우에만 while을 돌고 만나면 반복을 끝낼 수 있게 조건을 주었습니다. 그 이후 map의 r,c위치에 현재 위치를 표시할 수 있게 maze->map의 r,c 위치에 .을 넣게 하였습니다. 그렇게 반복을 시작하면 동, 서, 남, 북 모든 방향을 다 탐색해서 갈 수 있는 곳을 스택에 push해야하기 때문에 r,c에 각각 maze에 들어있는 here\_r, here\_c정보를 넣어주었습니다. 그 다음 이 r,c를 탐색 해야하는 순서인 북, 서, 남, 동의 반대로 스택에 넣어야 했으므로 이 순서의 역순서인 동(r, c+1), 남(r+1, c), 서(r, c-1), 북(r-1, c)순서로 push\_loc를 하여 갈 수 있는 곳들을 선별하여 스택에 넣었습니다. 그 다음 이제 찾은 중인지, 아니면 아예 불가능한지를 판별하기 위해 먼저 if문으로 empty\_stack을 사용, 스택이 비어있을 경우에는 길이 아예 존재하지 않고 벽으로 막혀있는 경우이기에 FAIFED를 return하였고, 이렇게 막혀있지 않은 경우, 즉 길이 있는 경우에는 스택에서 r,c의 위치정보이니 반대 순서인 c, r의 순서로 pop하여 끄집어내어 현재 위치인 maze의 here\_r, here\_c에 넣어주었습니다. 이러한 것들을 반복하다보면 출구인 x에 도달할 것이고, x에 도달한다면 while문을 빠져 나올 것이기에 이럴 경우는 성공한 것임을 알려주기 위해 while 밖에서 FOUND를 return하게 하였습니다.

3.2

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "maze.h"

// ======================================================================

char map1[MAX\_MAZE\_SIZE][MAX\_MAZE\_SIZE] = {

{'1', '1', '1', '1', '1', '1'},

{'e', '0', '1', '0', '0', '1'},

{'1', '0', '0', '0', '1', '1'},

{'1', '0', '1', '0', '1', '1'},

{'1', '0', '1', '0', '0', 'x'},

{'1', '1', '1', '1', '1', '1'},

};

// ======================================================================

int main(void)

{

int r, step = 1;

MAZE maze1;

maze1.map = map1;

maze1.stack = malloc(sizeof(STACK));

init\_stack(maze1.stack);

maze1.here\_r = 1;

maze1.here\_c = 0;

for (;;) {

printf("== step %d ==\n", step++);

print\_maze(&maze1);

r = run\_maze(&maze1);

if (r == FAILED) {

printf("FAILED!\n");

break;

}

if (r == FOUND) {

printf("FOUND!\n");

break;

}

}

return EXIT\_SUCCESS;

}

// ======================================================================

분석 후 설명

: test-maze.c 파일 안에는 먼저 우리가 탐색해야할 테스트 미로인 map1이 MAX\_MAZE\_SIZE에 맞게 전역변수로 2차원 배열로 선언되어 있습니다. 그 다음에는 본격적으로 main함수가 시작되는데, 우선 테스트해야할 maze1, run\_maze의 성공 실패 정보를 저장하기 위한 변수r, 미로찾기 단계 표시를 위한 step변수(초기화1)이 선언되어 있습니다. 그리고 이제 테스트해야할 maze1의 map에 map1을 넣었고, maze1의 스택은 동적할당을 통해 STACK type의 크기만큼 공간을 확보하고, init\_stack으로 초기화시킨 것을 볼 수 있습니다. 그리고 maze1의 현재위치정보인 here\_r과 here\_c에 처음 시작 위치인 1,0을 넣었습니다. 그리고 나서 for문 무한 반복을 통해 step, 즉 각 단계제목 출력과 함께 r에 maze1의 run\_maze결과를 저장시켜 if문을 사용, 실패인 FAILED나 성공인 FOUND면 어쨌든 미로찾기가 끝난 상태이니 break로 나가주고, 만약 아직 찾는 중인 FOUNDING을 return되었으면 아무소리없이 다시 반복해서 새로 또 미로의 진행상황이 출력됩니다. 이렇게 모든 미로찾기 절차가 끝나면 EXIT\_SUCCESS를 return해서 함수를 끝냅니다.

3.3

s1910783@oak:hw04$ gcc maze.c stack.c test-maze.c -o test-maze

s1910783@oak:hw04$ ./test-maze

== step 1 ==

111111

\*01001

100011

101011

10100x

111111

(2 : 1, 1)

== step 2 ==

111111

.\*1001

100011

101011

10100x

111111

(2 : 2, 1)

== step 3 ==

111111

..1001

1\*0011

101011

10100x

111111

(2 : 2, 2)

(4 : 2, 2, 3, 1)

== step 4 ==

111111

..1001

1.0011

1\*1011

10100x

111111

(4 : 2, 2, 4, 1)

== step 5 ==

111111

..1001

1.0011

1.1011

1\*100x

111111

== step 6 ==

111111

..1001

1.\*011

1.1011

1.100x

111111

(2 : 2, 3)

== step 7 ==

111111

..1001

1..\*11

1.1011

1.100x

111111

(2 : 3, 3)

(4 : 3, 3, 1, 3)

== step 8 ==

111111

..1\*01

1...11

1.1011

1.100x

111111

(4 : 3, 3, 1, 4)

== step 9 ==

111111

..1.\*1

1...11

1.1011

1.100x

111111

== step 10 ==

111111

..1..1

1...11

1.1\*11

1.100x

111111

(2 : 4, 3)

== step 11 ==

111111

..1..1

1...11

1.1.11

1.1\*0x

111111

(2 : 4, 4)

== step 12 ==

111111

..1..1

1...11

1.1.11

1.1.\*x

111111

(2 : 4, 5)

== step 13 ==

111111

..1..1

1...11

1.1.11

1.1..\*

111111

FOUND!

3.4

<step 1>에서 print\_maze함수를 통해 현재 위치가 \*로 표기되면서 maze가 출력되고, 그 다음 북, 서 ,남, 동의 순서대로 갈 수 있는 위치를 파악해서 갈 수 있는 곳을 역순으로push\_loc함수를 통해 스택에 넣은 결과 (2:1, 1)로 나왔다는 것을 알 수 있습니다. 그 다음 run\_maze함수의 for문 반복으로 그 다음 미로찾기 sequence로 넘어가고 그전에 스택에 있던 위치정보 하나가 pop되어 다음 현재위치(1, 1)로 되었기에 run\_maze함수에서 짯던 대로 지나갔던 위치가 .으로 바뀌었다는 것을 알 수 있습니다.

그렇게 넘어간 <step2>에서 다음 위치가 또 다시 print\_maze함수로 현재위치가 \*로 바꿔며 maze가 출력되고, 그리고 또 run\_maze함수를 통해 현재 위치가 .으로 되고, 역순으로 스택 검사를 맡으며 psuh한 결과 스택에는 (2:2, 1)로 들어가 있다는 것을 알 수 있다. 또 다시 같은 방식으로 다음 sequence로 넘어가 스택에 있던 위치정보 하나가 pop되어 다음 현재위치(2, 1)이 되고 지나간 위치는 .으로 되었습니다.

그렇게 넘어간 <step3>에서 같은 방식으로 현재위치\*로 해서 maze가 출력되고 또 지나간 곳은 .으로 되고, 갈 수 있는 곳을 pusg\_loc함수를 통해 스택에 넣은 결과 나오는 순서가 북, 서, 남, 동으로 됨으로 들어가는 순서는 동, 남, 서, 북에 맞추어 들어갈 수 있는 위치가 나올 때마다 또 출력하였으므로 저렇게 스택이 동 먼저 남이 들어간 형태로 2번 출력되었습니다.

끝.