강의명: 자료구조

숙제 번호: 5

숙제 제목: Queues and Lists(큐 및 리스트)

학생 이름: 김성현

학번: 201910783

1. Queue(큐)

1.1

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

#include <stdio.h>

#include <stdlib.h>

#include "queue.h"

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

void init\_queue(QUEUE \* q)

{

q->front = 0;

q->rear = 0;

}

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

void print\_queue(QUEUE \* q)

{

int f, r, n = 0;

if (q->front < q->rear)

n = (q->rear - q->front);

else

n = (7 - q->front) + (q->rear + 1);

if (q->front == q->rear)

n = 0;

printf("(%d|%d|%d:", q->front, q->rear, n);

if (!empty\_queue(q)) {

int tmp = q->front;

do {

tmp = (tmp + 1) % (MAX\_QUEUE\_SIZE);

printf("%d", q->data[tmp]);

if(tmp == q->rear)

break;

printf(",");

} while (tmp != q->front);

}

printf(")");

printf("\n");

}

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

int empty\_queue(QUEUE \* q)

{

if (q->front == q->rear)

return 1;

else

return 0;

}

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

int full\_queue(QUEUE \* q)

{

if (q->front == (q->rear + 1) % MAX\_QUEUE\_SIZE)

return 1;

else

return 0;

}

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

int en\_queue(QUEUE \* q, int item)

{

if (full\_queue(q) == 1)

return ERROR;

else {

q->rear = (q->rear + 1) % MAX\_QUEUE\_SIZE;

q->data[q->rear] = item;

return q->rear;

}

}

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

int de\_queue(QUEUE \* q)

{

if (empty\_queue(q) == 1)

return ERROR;

else {

q->front = (q->front + 1) % MAX\_QUEUE\_SIZE;

return q->front;

}

}

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

\*이렇게 queue.c를 프로그램한 이유:

먼저 void init\_queue(QUEUE \* q)함수는 큐의 초기화 함수이고 또 structure QUEUE안에는 data와 함께 선입선출을 위한 front, rear가 같이 선언되어 있기 때문에 초기화 함수에서는 이 front와 rear 모두 동일하게 q->front = 0과 q->rear = 0으로 선언해주면서 완전히 큐가 비어있는 상태인 front = 0, rear = 0의 상태로 만들어주도록 하였습니다. 다만 여기서 front가 rear가 같은 경우, 즉 큐가 비어있는 경우는 if문으로 따로 주어 이경우 비어서 item이 아예 없기에 n값이 0이 되도록 하였습니다.

그 다음 void print\_queue(QUEUE \* q)함수는 큐에 저장된 data들을 출력하는 함수입니다. 다만 이 함수는 출력 형식이 (front|rear|item갯수|i1~in)의 형식이기 때문에, 일단 front와 rear는 이미 q안에 있기 때문에 그냥 그대로 사용하면 되고, n은 item의 개수이기 때문에 먼저 n을 구해주고 그 다음 형식에 맞는 출력을 하는 것으로 하였습니다. 이에 따라 n, 즉 item의 개수를 먼저 구하였는데, item의 개수 n은 front가 rear보다 작은 경우 / front가 rear보다 큰 경우로 나누어서 구할 수 있습니다. 먼저 front가 rear보다 작은 경우는 아직 array의 끝인 7까지 넘어가지 않은 경우 + front자리는 빈자리이기 때문에 rear – front를 하게 되면 정확히 item의 개수만큼 나오게 됩니다. 그리고 array의 끝자리인 7을 넘어서 다시 0으로 가서 또 채워지는 경우에는 front가 rear보다 큰 경우임으로 먼저 front부터 7까지의 개수인 7 – front를 구하고 남은 item들은 0부터 rear까지의 item임으로 rear + 1을 통해 구해 이 둘을 더해주게 되면 총 개수가 나오게 됩니다. 이렇게 총 item개수 n을 구하고 나면 먼저 출력의 앞부분 형식에 맞춰 출력하기 위해 “(front|rear|item갯수 n:”의 형태로 출력하였습니다. 그 다음 큐가 비어있을 경우 출력할 item이 없기 때문에 item을 출력하는 부분을 empty\_queue를 검사하는 if문 안에 넣어 놓았습니다. 이렇게 if문을 넘어 item을 출력하기 위해 do~while문을 이용하여 임시 변수 tmp에 front의 값을 넣어주고 이를 moduls연산자를 같이 사용하여 순환 형식으로 1씩 계속 update하면서 차례차례 큐의 element들을 방문하여 출력하도록 하였습니다. 또 하나의 item을 출력하고 ,를 출력하는데, 그 전에 중간체크로 tmp, 즉 읽던 부분이 rear인지 아닌지를 체크해 아니면 ,를 정상적으로 출력하고, 만약에 rear라면 읽을 수 있는 끝부분에 도달한 것이므로 break하여 반복문을 빠져 나오고 이 모든 것은 tmp가 front의 값과 다시 같아지지 않는다는 조건 하에 작동하도록 하여 혹여 있을 이상한 값 읽기나 반복 읽기를 방지하였습니다. 그 다음 모든 걸 빠져나오고 나서 닫는 괄호 )를 출력하여 형식을 완성시키고 다음 라인으로 넘겼습니다.

그 다음 int empty\_queue(QUEUE \*q)함수는 큐의 empty여부를 체크하는 함수입니다. 큐가 비어있다는 것은 다시 말해 에러상황이 아니라는 상황 하에front와 rear가 같은 경우 임으로 이를 위해 if문의 조건에 front와 rear가 같은 경우를 주어 문제 조건에 맞추어 같을 경우 1을 return하고, 그 외의 경우, 즉 다를 경우는 0을 return하도록 하였습니다.

그 다음 int full\_queue(QUEUE \*q)함수는 위의 empty함수와 정반대로 full여부를 체크하는 함수입니다. empty함수와 정반대를 찾는 것이지만 그 형태는 비슷하기 때문에 기본적으로 if/else문을 써 full인 경우 1을 return하고, full이 아닌 그 외의 경우 0을 return하였습니다. 다만 조건에서 주의해야 할 점이 있는데, 큐는 항상 front와 rear사이에 최소한 1개의 빈공간이 있어야합니다. 그래서 조건에 front와 rear + 1이 같은 경우를 주는데, 여기서 순환, 즉 circular의 형태로 돌아갈 수 있게 modlus연산자를 사용하여 반환된 값을 비교하여 주었습니다.

그 다음 int en\_queue(QUEUE \*q, int item)함수는 큐의 맨 뒤 rear에 item을 삽입하는 함수입니다. 그러기 위해 우선 큐가 꽉찬 경우 삽입을 하면 안되기 때문에 if문의 조건에 full\_queue(q)를 주어 return이 1, 즉 꽉 차있을 경우 ERROR를 return하도록 하였습니다. 그외, 즉 빈공간이 있는 else의 경우 삽입을 해야하기에 우선 먼저 rear를 1증가시켜 공간을 확보시킵니다. 다만 여기서도 마찬가지로 moduls연산자를 사용하여 순환적으로, 즉 7에서 0으로 가는 것처럼 해주었습니다. 이렇게 증가된 공간인 rear index에 item을 넣고, 이 넣은 rear의 정보, 즉 값을 조건에 맞추어 return해주었습니다.

마지막으로 de\_queue(QUEUE \*q)는 함수의 맨 앞 front에서의 element를 삭제해주는 함수입니다. 그러기 위해 우선 en\_queue와 같은 맥락으로 큐가 빈 경우 삭제가 안되기 때문에 이를 체크하기 위해 if문으로 empty\_queue를 검사해주어 만약 비었을 경우 ERROR를 return해주도록 하였습니다. 그외, 정상적으로 뭔가 들어있는 경우의 else에서는 이제 삭제를 해주어야 하기 때문에 먼저 빈 공간을 지칭하고 있는 front을 마찬가지로 moduls연산자로 순환구조를 만들어 주면서 1 증가시켜 맨 앞의 삭제할 아이를 지칭하도록 합니다. 그 다음 이 아이를 지칭하는 front를 반환토록 하여 이 아이를 없는 취급할 수 있도록 하여 삭제된 것과 마찬가지의 의미로 만들어주었습니다.

1.2

s1910783@oak:hw05$ gcc queue.c test-queue.c -o test-queue

s1910783@oak:hw05$ ./test-queue

===== test-queue =====

(0|0|0:)

(0|1|1:10)

(0|2|2:10,11)

(1|2|1:11)

(1|3|2:11,12)

(1|4|3:11,12,13)

(2|4|2:12,13)

(2|5|3:12,13,14)

(2|6|4:12,13,14,15)

(3|6|3:13,14,15)

(3|7|4:13,14,15,16)

(3|0|5:13,14,15,16,17)

(4|0|4:14,15,16,17)

(4|1|5:14,15,16,17,18)

(4|2|6:14,15,16,17,18,19)

(5|2|5:15,16,17,18,19)

(5|3|6:15,16,17,18,19,20)

(5|4|7:15,16,17,18,19,20,21)

(6|4|6:16,17,18,19,20,21)

(6|5|7:16,17,18,19,20,21,22)

OK: full\_queue

(7|5|6:17,18,19,20,21,22)

(0|5|5:18,19,20,21,22)

(0|6|6:18,19,20,21,22,24)

(1|6|5:19,20,21,22,24)

(2|6|4:20,21,22,24)

(2|7|5:20,21,22,24,25)

(3|7|4:21,22,24,25)

(4|7|3:22,24,25)

(4|0|4:22,24,25,26)

(5|0|3:24,25,26)

(6|0|2:25,26)

(6|1|3:25,26,27)

(7|1|2:26,27)

(0|1|1:27)

(0|2|2:27,28)

(1|2|1:28)

(2|2|0:)

(2|3|1:29)

(3|3|0:)

OK: empty\_queue

1.3

int main(void)

{

int item = 10, r;

QUEUE q1;

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

init\_queue(&q1);

print\_queue(&q1);

for (;;) {

if (en\_queue(&q1, item++) == ERROR)

break;

print\_queue(&q1);

if (en\_queue(&q1, item++) == ERROR)

break;

print\_queue(&q1);

if (de\_queue(&q1) == ERROR)

break;

print\_queue(&q1);

}

r = full\_queue(&q1);

if (r != 1)

printf("ERROR: full\_queue\n");

else

printf("OK: full\_queue\n");

for (;;) {

if (de\_queue(&q1) == ERROR)

break;

print\_queue(&q1);

if (de\_queue(&q1) == ERROR)

break;

print\_queue(&q1);

if (en\_queue(&q1, item++) == ERROR)

break;

print\_queue(&q1);

}

r = empty\_queue(&q1);

if (r != 1)

printf("ERROR: empty\_queue\n");

else

printf("OK: empty\_queue\n");

return EXIT\_SUCCESS;

}

\*이 함수가 하는 일:

Test-queue.c의 main함수가 하는 일은 말그대로 우리가 만든 queue.c 안에 있는 큐를 위한 기능들이 제대로 되었는지 검사하도록 초기화, 출력을 하고 삽입, 삭제를 반복하면서 제대로 되었는지 체크하는 함수라고 할 수 있다. ====test-queue===같은 구분을 위해 써놓은 부분은 제외하고 주요한 부분만 설명하자면 먼저 QUEUE구조체의 q1을 선언하여 일단 우리가 초기화를 제대로 짜놨는지 체크를 위해 이 q1을 초기화 함수 init\_queue함수와 print\_queue함수에 q1을 넣어주고 출력하여 체크합니다. 그 다음 for무한반복을 통해 en\_queue함수를 이용, q1에 item들을 1증가시키고 삽입, 또 1증가시키고 삽입하는 등 2번의 삽입과 1번 삭제하는 구조의 if문을 주어 총 3번의 과정에서 어느 하나라도 ERROR, 즉 꽉 찼음이 나오면 바로 끝내도록 하고, 아니라면 계속 2번 삽입, 1번 삭제의 형식을 반복하도록 합니다. 이렇게 for문 무한반복을 빠져나오면 변수 r에 full\_queue의 반환값을 넣어 full체크함수의 정상작동도 체크하면서 q1의 full여부도 체크하여 꽉 찼을 경우 OK, 그러지 않을 경우는 삽입 삭제 반복 중 무언가 이상이 있었던 것이기에 ERROR를 출력하도록 하는 역할도 하도록 하게 된 것을 볼 수 있습니다. 그리고 이렇게 q1이 꽉 찼다면 거기서 부터는 이제 반대로 또 for문을 통해 2번 삭제하고, item을 1씩 증가시키면서 1번 삽입, 즉 2번 삭제 1번 삽입의 3번 작동을 반복하면서 다시 정상적으로 비워지는 과정을 체크합니다. 이것도 끝났다면 이제 마무리로 r에 empty\_queue함수의 반환값을 주어 empty함수의 정상작동도 체크하고, q1도 정상적으로 비워졌으면 OK를 출력하고 그러지 않으면 ERROR를 출력하여 정삭적으로 싹 비워졌는지도 체크합니다. 그러고 마무리로 main을 싹 빠져나갑니다.

\*main과 학생의 화면 출력과의 일치 여부 비교 설명:

위에 있는 출력과 한 번 비교해보자. 젤 첫 줄에 (0|0|0:)을 통해 초기화가 잘 되었고 이에 대한 출력도 정상적으로 main에 있는 대로 되었음을 알 수 있다. 그 다음 두 번째 줄을 보게 되면 item이 1번 삽입되어 10이 들어가있고, rear와 n이 1증가하여 1이 되어 (0|1|1:10)으로 출력되었음을 알 수 있다. 그 다음 세번째 줄을 보면 11이 삽입되어 또 한번의 삽입이 일어났고 rear와 n이 또 1증가하여 2가 되어 (0|2|2:10,11)로 출력되었음을 알 수 있다. 또 그 다음 줄인 네번째 줄을 보면은 앞부분, front부분인 10이 삭제되고 front는 삭제의 흔적으로 1이 되었고, n도 1줄어들어 (1|2|1:11)이 되어있음을 알 수 있다. 이렇게 다음 줄, 또 다음 줄을 보면 item의 값이 1씩 증가하면서 2번 삽입 되었다가 또 앞부분이 1번 삭제되어 줄어들었음을 알 수 있다. 이렇게 계속 item 증가-2번삽입 1번 삭제를 circular큐에서 rear가 7에서 또 0으로 순환되는 식을 보여주면서 계속 가다가 n=7이 되는, 정상적인 circular큐의 구조가 되어 main의 fill\_empty검사 부분을 거쳐 OK: full\_queue가 출력되어 있음을 우리는 볼 수 있다.

다시 그 다음부분은 즉 2-1부분 시작부터 앞에서 1번, 그 다음 줄에서도 1번 삭제되고 그 다음 줄에서는 item이 계속해서 1증가되서 삽입되고 또 그 다음 줄부터 2번 삭제 1번 삽입을 반복하여 circular 큐가 완전히 빌때까지 2번 삭제 1번 삽입을 반복하는 형식으로 main과 동일하게 이어지고 있음을 우리는 출력을 통해 알 수 있다. 그렇게 가다가 보면 마지막에 큐가 완전히 비었음과 함께 main에서 본대로 empty검사를 하여 OK: empty\_queue가 출력되었음을 우리는 볼 수 있다.

2. List(리스트)

2.1

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

#include <stdio.h>

#include <stdlib.h>

#include "list.h"

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

void print\_list(NODE \* list)

{

int n = length\_list(list);

int l = 0;

printf("(%d:", n);

for (NODE \*p = list;p != NULL; p = p->link) {

printf("%d", p->data);

l++;

if (l == n)

break;

printf(",");

}

printf(")");

printf("\n");

}

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

int length\_list(NODE \* list)

{

int l = 0;

NODE \*p;

for (p = list; p != NULL; p = p->link)

l++;

return l;

}

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

NODE \*nth\_list(NODE \* list, int n)

{

int tmp = 0;

NODE \*p;

for (p = list; p != NULL; p = p->link) {

tmp++;

if (tmp == n)

return p;

}

return NULL;

}

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

NODE \*add\_list(NODE \* list, int item)

{

NODE \*node;

node = (NODE \*)malloc(sizeof(NODE));

node->data = item;

node->link = list;

return node;

}

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

NODE \*delete\_list(NODE \* list)

{

NODE \*node;

if (list == NULL) return NULL;

node = list;

list = node->link;

free(node);

return list;

}

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

\*이렇게 프로그램한 이유:

먼저 void print\_list(NODE \* list)함수는 연결 리스트를 출력하는 함수입니다. 연결 리스트 출력을 위해 저는 새로운 NODE pointer 타입 p를 선언해 여기에 list를 넣어주고 for문을 돌려 다음 리스트가 null이 되기 전까지 쭉 출력해주다가 리스트의 길이만큼 진행하였다면, 즉 리스트의 끝부분에 도달하였다면 for문을 빠져나와 괄호를 닫아 마무리하는 것으로 전체적인 구성을 만들었습니다. 세부적으로 설명하자면 먼저 리스트의 길이를 구하기 위해 length\_list(list)함수를 사용하여 길이를 변수 n에 저장시켰습니다. 그 다음 진행정도를 저장하기 위한 변수 l을 선언하고 출력의 첫 시작으로 여는괄호와 배열의 길이 n, 즉 item의 개수를 출력시켰습니다. 그 다음 각 item들의 출력을 위해서 for문을 시작하여 NODE pointer타입 p를 선언해 여기에 list를 저장시키고, 이 p가 NULL이 아닐 동안만 반복을 하게 하여 정확하게 값이 있는 부분만 돌게 하고, p를 다음 주소가 저장되어 있는 link값으로 바꾸어 주어 다음 리스트로 넘어가도록 하여 모든 list들을 방문하도록 하였습니다. 이런 반복 속에서 data에 있는 item을 하나하나 차례대로 출력하고, 하나를 출력할 때마다 진행상태 표시를 위해 l을 계속 1씩 증가시켰습니다. 그리고 형식에 맞추기 위해 l, 즉 진행 상태가 리스트의 길이와 동일해졌는 경우, 즉 진행 상황이 끝에 다다랐을 경우 ,출력없이 바로 for문을 빠져나가도록 if문 break를 사용하였습니다. 이럴 경우가 아닌 정상적으로 다음 리스트로 넘어가야할 경우 if문 밖에 ,를 출력하도록 하였습니다. 이렇게 for문을 빠져나오게 되면 출력 형식의 마무리를 위해 닫는 괄호를 출력하고 다음 라인으로 넘어가도록 하였습니다.

그 다음 int length\_list(NODE \* list)함수는 리스트의 길이를 반환하는 함수입니다. 길이 반환을 위해 먼저 진행 상태를 저장할 수 있는 intiger형 변수 l과 리스트 방문을 위해 NODE pointer 타입 변수 p를 선언합니다. 그 다음 for문을 시작하여 p에 list를 넣어 p가 list가 되게 하고, p가 null인 끝부분까지 다다르도록 하는데 p는 다음 주소인 link로 가도록 for문을 만들었습니다. 이렇게 하면 한번 반복 = 리스트 한 개 방문이 되기 때문에 한번 반복할 때마다 l++을 하여 1개씩 현재 상태를 증가시켜 마지막에는 총 방문 횟수, 즉 총 리스트 개수를 나타낼 수 있도록 하였습니다. 그리고 이런 for문이 종료되면 총 방문 횟수이자 리스트 개수인 l을 return하여 조건에 맞추어 주었습니다.

그 다음 NODE \*nth\_list(NODE \* list, int n)함수는 n번째(n>=1)요소의 주소를 반환하는 함수이다. 이를 위해 전체적인 구조는 length\_list함수처럼 for문에 NODE pointer 타입 p를 null이 될 때까지 link를 타고 넘어가면서 돌리는 것인데, 여기서 차이점은 바로 tmp이다. 얼핏보면 tmp도 l처럼 현태 상태 체크를 위해 계속 tmp++을 해주지만, 바로 그 순간, if문을 걸어 체크포인트를 만들어준다. if문을 걸어 현재 진행 상태 tmp가 우리가 알고싶은 n에 도달 했을 시, 그 시점에서 우리의 p는 n번째 요소가 되어있기 때문에 p를 return해주는 것이다. 이런 체크포인트에 걸리지 않았을 시 그냥 for문을 아무런 제지없이 빠져나오기 때문에 이는 오류에 해당, 이 경우 NULL을 return해준다.

그 다음 NODE \*add\_list(NODE \* list, int item)함수는 list의 맨 앞에 item이 저장된 새 node를 생성하여 추가한 후 수정된 list의 주소를 반환하는 함수이다. 먼저 새로 집어넣을 node를 생성하기 위해 빈 node인 NODE pointer 타입의 node를 선언하여 이 node의 크기를 실제 리스트의 node크기에 맞추기 위해 동적할당함수 malloc을 활용, NODE의 크기만큼 만들어 NODE pointer 타입으로 형변환시켜 기본 틀을 만들어 주었습니다. 그 다음 node의 data에 item을 넣어 집어넣을 준비를 하시고, node의 link의 다음 주소로써 리스트의 시작인 list를 넣어줌으로써 새로 만든 노드의 link를 기존 list 앞에 연결시켜 간단하게 완성 시키고 수정된 list의 첫 주소인 node를 반환해주었습니다. 다만 list는 full이란 것이 존재하지 않는 자료구조이기에 이렇게 삽입할 때는 아무런 조건이 없습니다.

마지막으로 NODE \*delete\_list(NODE \* list)함수는 list의 맨 앞 node를 제거한 후 수정된 list의 주소를 반환하는 함수입니다. delete함수는 add와 다르게 리스트가 비어있을 경우 삭제가 불가함으로 if문으로 조건을 주어 이럴 경우 문제 조건에 맞게 NULL을 return해주었습니다. 그 다음 정상적인 경우는 새로 선언한 NODE pointer 타입 node에 list를 넣어주어 빼줄 준비를 합니다. 그 다음 원래 헤드 포인터 값인 list에 node->link, 즉 두번째node를 가리키는 주소를 넣어주어 node라는 것을 어디에도 연결되지 않은 독립적으로 만들어 줍니다. 이 node가 바로 첫번째 node이니 이걸 free함수를 통해 메모리를 싹 없애줌으로써 삭제하고, 두번째 node, 즉 새로운 리스트의 첫번째 node를 가리키는 새 리스트 list를 return해주어 마무리하였습니다.

2.2

<test-list0>

s1910783@oak:hw05$ gcc list.c test-list0.c -o test-list0

s1910783@oak:hw05$ ./test-list0

===== test-list0: print\_list =====

(0:)

(1:10)

(2:10,20)

(3:10,20,30)

(4:10,20,30,40)

<test-list1>

s1910783@oak:hw05$ gcc list.c test-list1.c -o test-list1

s1910783@oak:hw05$ ./test-list1

===== test-list1: length\_list =====

(0:)

(1:10)

(2:10,20)

(3:10,20,30)

(4:10,20,30,40)

<test-list2>

s1910783@oak:hw05$ gcc list.c test-list2.c -o test-list2

s1910783@oak:hw05$ ./test-list2

===== test-list2: nth\_list =====

OK: nth\_list

OK: nth\_list

OK: nth\_list

OK: nth\_list

<test-list3>

s1910783@oak:hw05$ gcc list.c test-list3.c -o test-list3

s1910783@oak:hw05$ ./test-list3

===== test-list3: add\_list =====

(0:)

(1:40)

(2:30,40)

(3:20,30,40)

(4:10,20,30,40)

<test-list4>

s1910783@oak:hw05$ gcc list.c test-list4.c -o test-list4

s1910783@oak:hw05$ ./test-list4

===== test-list4: delete\_list =====

(4:20,30,40,50)

(3:30,40,50)

(2:40,50)

(1:50)

(0:)

끝.