**자료구조 실습 보고서**

**실습 5. 큐**

**2016년 4월 6일**

**학번: 201404051**

**이름: 정 용 석**

**1. 실습 문제 소개**

**5-1) 개선된 큐 구현**

강의 자료를 바탕으로 작성한 큐를 사용했을 시에 일어나는 포화 문제를 해결하는 문제이다. 큐 배열에 자료를 입력하고 삭제했을 때 첫 번째 자료를 가리키는 front가 증가함으로서 입력 삭제 이 후, 배열의 크기만큼 자료를 더 저장할 수 없는 문제를 해결하기 위해, 자료 삭제 시, front는 증가하지 않고, 배열의 남은 자료를 모두 앞으로 재배열하는 형식으로 해결해야한다. 그러므로 front는 -1의 값을 고정으로 가지게 된다.

**5-2)원형 큐 구현**

실습 5-1에서 작성한 큐의 문제점은 자료 삭제 시 매번 배열을 재배치 시켜주어야 하는 번거로움이 생기므로 큐의 이점을 살리지 못할뿐더러 자료의 처음을 가리키는 front의 의미가 없어진다. 이의 문제점을 개선하기 위해 배열을 재배치하는 것이 아닌 배열의 처음과 끝을 연결 시켜 원형의 배열 형식으로 바꾸어 주는 형식이다.

**5-3)원형 덱 구현**

실습 5-2의 원형 큐는 한쪽 방향으로 밖에 자료를 추가할 수 없다는 불편함이 있다. 이를 해결하기 위해 원형 큐를 원형 덱으로 구현하는 문제이다. 기본적인 원형 큐의 구조와 기능을 가지면서 자료를 front혹은 rear 방향으로 추가, 삭제할 수 있어야 한다.

**2. 소스코드**

**5-1) 개선된 큐 구현(\*수정 된 소스코드)**

char deQueue(Q\_TYPE \*Q){ //DEQUEUE 수정

int i; int deque\_Item;

if (isEmpty(Q)) {

printf("Error: Queue is Empty!\n");

exit(1);

}

else {

//삭제할 자료를 저장한 뒤, 배열을 한칸씩 앞으로 재배열하고 삭제한 자료를 출력

deque\_Item = Q->queue[(Q->front)+1];

for(i = 1; i < Q->rear+1; i++)

Q->queue[(Q->front)+i] = Q->queue[(Q->front)+1+i];

Q->rear--;

return deque\_Item;

}

}

**5-2)원형 큐 구현(\*수정된 소스코드)**

void printQ(Q\_TYPE \*Q) //수정

{

int i;

printf("Queue: ");

//배열을 콘솔에 출력 시, front+1부터 rear까지 자료 출력

for (i = Q->front+1; i != (Q->rear+1)%Q\_SIZE; i=(i+1)%Q\_SIZE)

printf("[%d] %c ",i, Q->queue[i]);

printf("\n");

}

Q\_TYPE \*createQueue( ) //수정

{

Q\_TYPE \*Q;

Q = (Q\_TYPE \*)malloc(sizeof(Q\_TYPE));

//수정함 front = rear = 0

Q->front = Q->rear = 0;

return Q;

}

int isFull(Q\_TYPE \*Q) //수정

{

//rear의 다음 배열이 Front을 가리키면 FULL

if ((Q->rear+1) % Q\_SIZE == Q->front)

return 1;

else

return 0;

}

void enQueue(Q\_TYPE \*Q, char item)

{

if (isFull(Q)) {

printf("Error: Queue is Full!\n");

exit(1);

}

else {

//rear을 통하여 자료 입력, mod를 통하여 한칸 씩 추가

Q->rear = (Q->rear+1) % Q\_SIZE;

Q->queue[Q->rear] = item;

}

}

char deQueue(Q\_TYPE \*Q) //수정

{

if (isEmpty(Q)) {

printf("Error: Queue is Empty!\n");

exit(1);

}

else {

//자료 삭제 시, front+1 이 가르키는 자료 출력

Q->front = (Q->front+1) % Q\_SIZE;

return Q->queue[Q->front];

}

}

**5-3)원형 덱 구현**

typedef struct {

char deque[Q\_SIZE];

int front, rear;

} Q\_TYPE;

Q\_TYPE \*initDeque() //구조체를 할당 초기화하고, front와 rear모두 -1

{

Q\_TYPE \*Q;

Q = (Q\_TYPE \*)malloc(sizeof(Q\_TYPE));

Q->front = Q->rear = -1;

return Q;

}

int isEmpty(Q\_TYPE \*Q) //front와 rear가 -1일 때 EMPTY

{

if (Q->front && Q->rear == -1)

return 1;

else

return 0;

}

int isFull(Q\_TYPE \*Q) //rear의 다음 배열이 front일 때 FULL

{

//수정함

if ((Q->rear + 1) % Q\_SIZE == Q->front)

return 1;

else

return 0;

}

void insertFront(Q\_TYPE \*Q, char item) //배열의 front방향으로 item 추가

{

if (isFull(Q)) {

printf("Deque is Full\n");

exit(1);

}

else if (isEmpty(Q)) //배열이 EMPTY일 때, 배열[0]에 item 저장 후, front와 rear모두 0을 가리킨다.

{

Q->front = Q->rear = 0;

Q->deque[Q->front] = item;

}

else

{

Q->front = ((Q->front) + Q\_SIZE - 1) % Q\_SIZE;

Q->deque[Q->front] = item;

}

}

void insertRear(Q\_TYPE \*Q, char item) //배열의 rear방향으로 item 추가

{

if (isFull(Q)) {

printf("Deque is Full\n");

exit(1);

}

else if (isEmpty(Q)) //배열이 EMPTY일 때, 배열[0]에 item 저장 후, front와 rear모두 0을 가리킨다.

{

Q->front = Q->rear = 0;

Q->deque[Q->rear] = item;

}

else

{

Q->rear = (Q->rear + 1) % Q\_SIZE;

Q->deque[Q->rear] = item;

}

}

char deleteFront(Q\_TYPE \*Q) //front의 자료 반환 후 삭제

{

char temp;

if (isEmpty(Q)) {

printf("Error: Queue is Empty!\n");

exit(1);

}

else if (Q->front == Q->rear) //배열에 저장된 자료가 1개일 때 자료 반환 후 EMPTY state

{

temp = Q->front;

Q->front = Q->rear = -1;

return temp;

}

else

{

temp = Q->front;

Q->front = (Q->front + 1) % Q\_SIZE;

return temp;

}

}

char deleteRear(Q\_TYPE \*Q) //rear의 자료 반환 후 삭제

{

char temp;

if (isEmpty(Q))

{

printf("Error: Queue is Empty!\n");

exit(1);

}

else if (Q->rear == Q->front) //배열에 저장된 자료가 1개일 때 자료 반환 후 EMPTY state

{

temp = Q->rear;

Q->rear = Q->front = -1;

return temp;

}

else

{

temp = Q->rear;

Q->rear = (Q->rear - 1) % Q\_SIZE;

return temp;

}

}

void printQ(Q\_TYPE \*Q)

{

int i, temp;

printf("Deque: ");

if (isEmpty(Q))

printf("Deque Empty");

else if (isFull(Q)) //배열이 FULL일 시, 모든 자료 출력

{

temp = 0;

for (i = Q->front; temp != Q\_SIZE; temp++){

printf("[%d] %c ", (i) % Q\_SIZE, Q->deque[(i) % Q\_SIZE]);

i++;

}

}

else //배열이 FULL이 아닐 시, 저장된 자료만 출력

{

for (i = Q->front; i != (Q->rear + 1) % Q\_SIZE; i = (i + 1) % Q\_SIZE)

printf("[%d] %c ", i, Q->deque[i]);

}

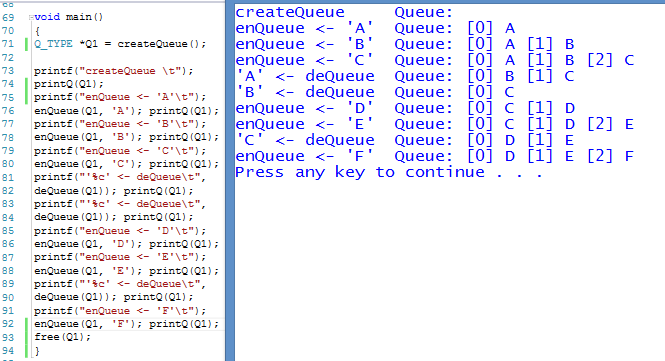
printf("\n");

}

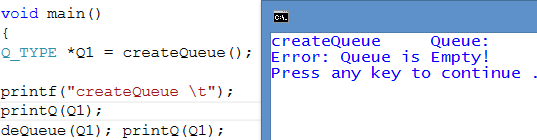
**3. 테스트 결과**

**5-1) 개선된 큐 구현**

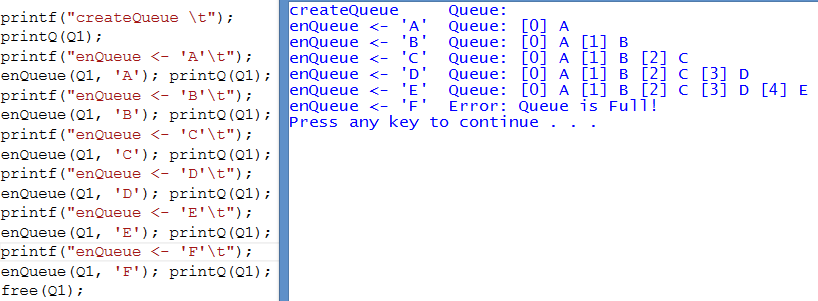
**i. 강의 자료 테스트 시나리오**



**ii. Empty Error**

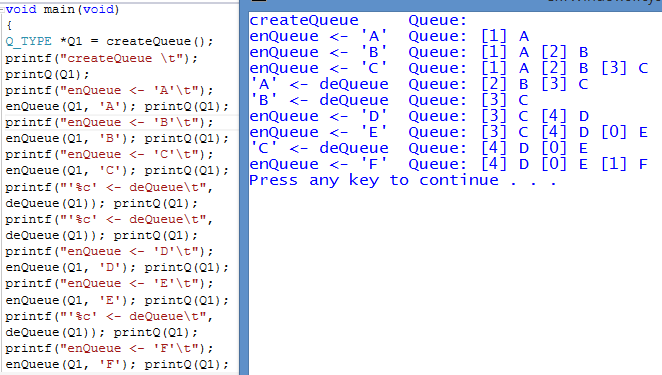


**iii. Full Error**

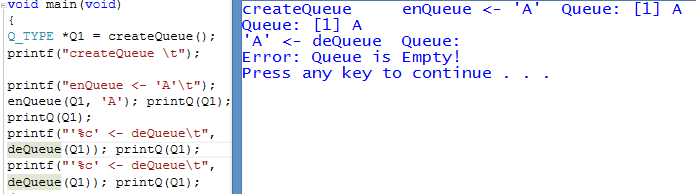


**5-2)원형 큐 구현**

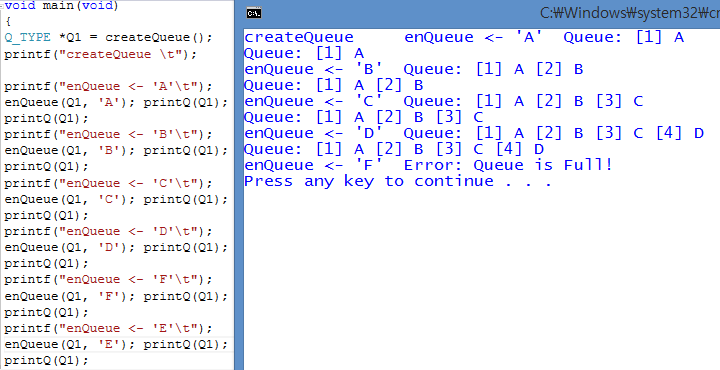
**i. 강의 자료 테스트 시나리오**



**ii. Empty Error**

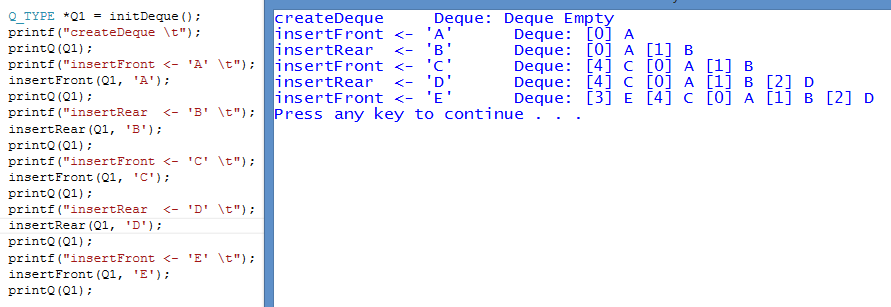


**iii. Full Error**

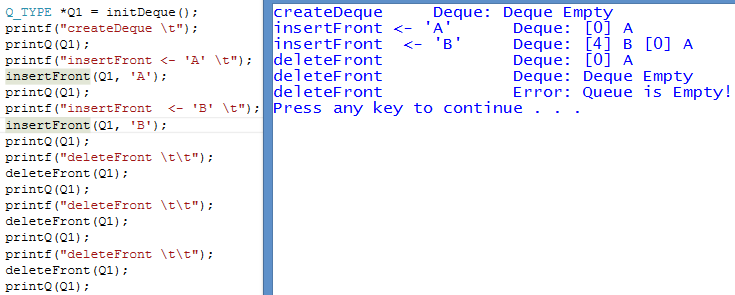


**5-3)원형 덱 구현**

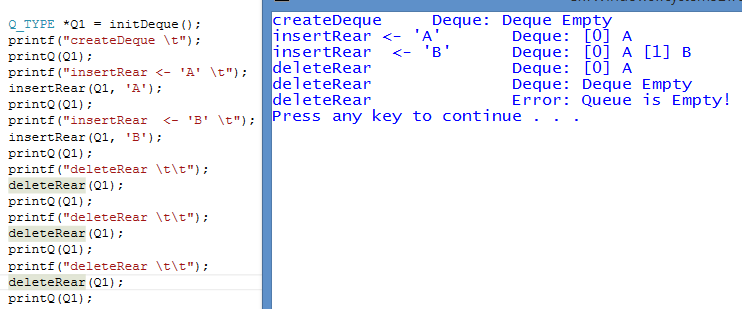
**i. 일반적인 경우**



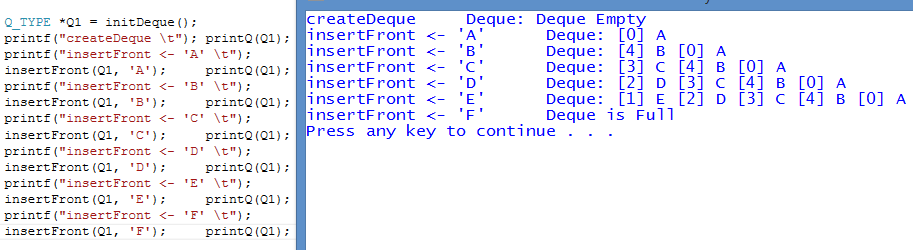
**ii. EMPTY ERROR 1(deleteFront로 test)**



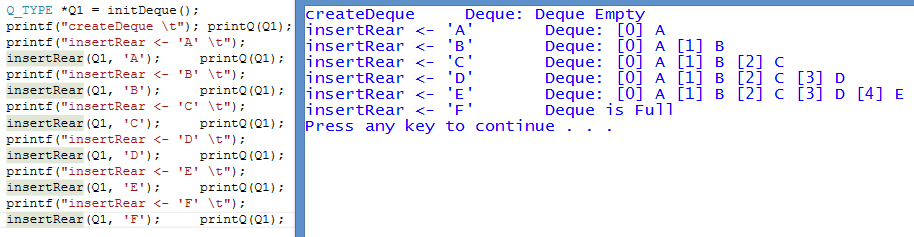
**iii. EMPTY ERROR 2(deleteRear로 test)**



**iv. FULL ERROR 1(insertFront로 test)**



**v. FULL ERROR 2(insertRear로 test)**



**4. 작성자 코멘트**

**5-1) 개선된 큐 구현**

이 실습 같은 경우는 굉장히 간단하다. 이미 구현된 코드는 그대로 유지하면서 배열에 있는 자료를 삭제할 때 사용하는 함수만 수정하면 되었다. 원본의 코드는 삭제와 동시에 front가 가리키는 배열 번호가 1씩 올라간다. 하지만 그렇게 하지 않고, front는 -1로 유지하면서 front+1이 가리키는 배열을 반환하고 그 뒤의 나머지 자료를 한 칸씩 앞으로 이동시키는 코드만 추가해주었다. 물론 rear은 삭제될 시, 1씩 줄어든다.

**5-2)원형 큐 구현**

5-1에서 구현한 큐 같은 경우는 사실 상 front를 굳이 만들 이유가 없다. Front는 -1로 고정이기 때문에 front가 가진 기능을 극대화 시킬 수 없는 단점이 있었다. 이를 극복하기 위해 배열의 처음과 끝을 연결시킨 원형의 형태인 원형 큐를 구현했다. 원형 큐 같은 경우는 자료 삽입 시 rear의 방향으로 추가 하는 방법은 동일하다. 하지만 자료 삭제 시 원본 큐와 동일하게 front의 값이 1씩 올라가게 된다. 하지만 다른 점은 rear가 배열의 끝을 가리키는 상태에서 자료를 추가했을 시에 다시 처음으로 돌아와서 rear가 0을 가리키게 된다는 점이다. 이를 구현하기 위해서 사용한 연산자가 바로 %, mod연산자 이다. 이를 사용함으로서 rear가 배열의 크기를 넘어갔을 때도 다시 처음으로 돌아올 수 있게 해준다. 출력 시에도 mod 연산자를 통해서 front+1부터 rear까지의 자료를 문제없이 출력할 수 있게 하였다. 여기서 중요한 점은 front는 빈 공간으로 설정함으로서 배열의 EMPTY나 FULL의 상태를 확인할 때 용이하게 만들어 준다.

**5-3)원형 덱 구현**

원형 덱 같은 경우는 원형 큐와는 다르게 자료의 입출력 방향이 자유롭다는 점에 있다. 또한, front가 가리키는 배열을 비워 두지 않게 하여 메모리의 빈 공간을 없애준다. 수정해야 할 사항이 굉장히 많다고 판단했기에, 코딩 전 어떤 식으로 구현을 할지에 대해 생각해야 했다. 일단 배열이 EMPTY일 경우에는 front와 rear가 모두 -1이게 하고, FULL일 경우에는 (rear+1%Q\_SIZE), 즉 rear의 다음 배열이 front일 시에 배열이 모두 FULL이라고 설정을 하였다.

이렇게 설정함으로서 추가적으로 생각해야 하는 것이 첫번째 자료가 입력되었을 때이다. 배열에 자료가 1개만 있을 시, front와 rear가, 즉 시작점이0이 된다. 즉, 사용자가 첫 데이터를 insertFront할 지, insertRear 할 지 모르기 때문에 이 두 함수에 추가하는 데이터가 첫 자료일 경우 예외 처리를 추가 해주었다.

사실 상 가장 큰 문제는 printQ에 있었다. 기존의 printQ를 사용했을 때의 문제점은 바로 배열이 꽉 차 있을 때, for문의 (i != (Q->rear + 1) % Q\_SIZE)의 한계로 마지막 자료는 출력이 되지 않는다. 어떻게 생각하면 큰 문제는 아니지만 간결함을 위해 1개의 for문을 통해 구현하고자 했지만, 원형 배열이라는 구조의 한계로 도무지 간단한 구현 방법이 생각나지 않았다. 따라서 else if를 통하여 배열이 FULL일 시 출력방법을 새로 생성하였다.