# 01. 문자 A, B, C, D, E를 큐에 넣었다가 다시 꺼내어 출력하면 어떻게 되는가?

(1) A, B, C, D, E (2) E, D, C, B, A  
 (3) A, B, C, E, D (4) B, A, C, D, E

큐는 FIFO 형식으로 넣은 순서 그대로 출력된다.

# 02. 원형큐에서 front가 3이고 rear가 5라고 하면 현재 원형큐에 저장된 요소들의 개수는? (단, MAX\_QUEUE\_SIZE는 8이다.)

(1) 1 (2) 2  
 (3) 3 (4) 4

front는 첫번째 요소보다 한 칸 앞에 있고, rear는 마지막 요소에 있기에 4, 5에만 요소들이 있다.

# 03. 10, 20, 30, 40, 50을 큐에 넣었다고 가정하고 3개의 항목을 삭제하였다. 남아있는 항목은?

40, 50

큐는 FIFO 형식으로, 먼저 넣은 요소가 먼저 꺼내지기 때문에 2개 항목이 남아있다.

# 04. 다음 중 원형큐에서 공백상태에 해당하는 조건은?

(1) front==0 && rear==0  
 (2) front==(MAX\_QUEUE\_SIZE-1) && rear==(MAX\_QUEUE\_SIZE-1)  
 (3) front==rear  
 (4) front==(rear+1) % MAX\_QUEUE\_SIZE

원형큐에서는 front와 rear가 같으면 공백상태라고 한다.

# 05. 크기가 10이고 front가 3, rear가 5인 원형큐에서 새로운 항목이 삽입되었을 경우, front와 rear의 새로운 값은?

(1) front는 4, rear는 5 (2) front는 3, rear는 6  
 (3) front는 4, rear는 6 (4) 포화상태가 된다

새 항목이 삽입되었을 때 rear만 1칸 증가하게 되고 front는 변동이 없다.

# 06. 다음과 같은 원형큐에서 (a)에서 (c)까지의 연산을 차례로 수행한다고 하자. 수행이 완료된 후의 큐의 상태를 그려라. 현재 front는 0이고 rear는 2라고 하자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] |
|  | B | C |  |  |

(a) A 추가 (b) D 추가  
 (c) 삭제

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] |
|  |  | C | A | D |

A, D가 추가되었기에 rear는 4가 되고, 삭제를 한 번 하면서 1에 있던 B가 삭제되면서 front는 1이 된다.

# 07. 큐에 항목들을 삽입하고 삭제하는 연산은 시간 복잡도가 어떻게 되는가?

(1) O(1) (2) O(log₂n)  
 (3) O(n) (4) O(n²)

입력 크기에 관계없이 일정한 연산을 수행할 수 있기에 O(1)의 시간 복잡도를 가진다.

# 08. 원형큐에 큐에 존재하는 요소의 개수를 반환하는 연산 get\_count를 추가하여 보라. C언어를 이용하여 구현하여 보라.

int get\_count(QueueType \*q) {  
 if (q->rear >= q->front) {  
 return (q->rear – q->front);  
 } else {  
 return (MAX\_QUEUE\_SIZE – (q->front – q->rear) + 1);  
 }  
 return 0;  
}

# 09. 2개의 스택을 사용하여 큐를 구현할 수 있을까? 2개의 스택을 사용하여 큐를 구현해보자. 입력이 들어오면 스택 #1에 넣는다. 출력 요청이 들어보면 스택 #2에서 요소를 꺼낸다. 스택 #2가 비어있을 때는 스택 #1의 모든 요소를 꺼내서 스택 #2에 넣는다. 프로그램으로 작성해보자.

#include <stdio.h>  
 #include <stdlib.h>  
 #define MAX\_STACK\_SIZE 100  
  
 typedef int element;  
 typedef struct {  
 element data[MAX\_STACK\_SIZE];  
 int top;  
 } StackType;  
  
 // 스택 초기화 함수  
 void init\_stack(StackType \*s) {  
 s->top = -1;  
 }  
  
 // 공백 검사 함수  
 int is\_empty(StackType \*s) {  
 return (s->top == -1);  
 }  
 // 포화 검사 함수  
 int is\_full(StackType \*s) {  
 return (s->top == (MAX\_STACK\_SIZE – 1));  
 }  
 // 삽입함수  
 void push(StackType \*s, element item) {  
 if (is\_full(s)) {  
 fprintf(stderr, “스택 포화 에러\n”);  
 return;  
 } else s->data[++(s->top)] = item;  
 }  
 // 삭제함수  
 element pop(StackType \*s) {  
 if (is\_full(s)) {  
 fprintf(stderr, “스택 포화 에러\n”);  
 return;  
 } else s->data[(s->top)--];  
 }  
  
 int main() {  
 int len,   
 Stacktype s1, s2;  
 init\_stack(&s1);  
 init\_stack(&s2);  
  
 printf(“정수 배열의 크기: “);  
 scanf\_s(“%d”, &len);  
  
 // Enqueue  
 for (int i = 0; i < len; i++) {  
 scanf(“%d”, &num);  
 push(&s1, num);  
 }  
 // Dequeue  
 if (is\_empty(&s1) && is\_empty(&s2)) {  
 fprintf(stderr, “스택 공백 에러\n”);  
 exit(1);  
 }  
 if (is\_empty(&s2)) {  
 while(!(is\_empty(&s1)) {  
 push(&s2, pop(&s1));  
 }  
 int x = pop(&s2);  
}

# 10. 피보나치 수열을 효과적으로 계산하기 위하여 큐를 이용할 수 있다. 만일 피보나치 수열을 순환에 의하여 계산하게 되면 경우에 따라서는 많은 순환 함수의 호출에 의해 비효율적일 수 있다. 이를 개선하기 위하여 큐를 사용하는데 큐에는 처음에는 F(0)와 F(1)의 값이 들어가 있어 다음에 F(2)를 계산할 때 F(0)를 큐에서 제거한다. 그 다음에 계산된 F(b)를 다시 큐에 넣는다. 피보나치 수열은 다음과 같이 정의된다. 큐를 이용하여 피보나치 수열을 계산하는 프로그램을 작성하라. F(0) = 0, F(1) = 1 F(n) = F(n-1) + F(n-2) 『----------------------------------------- (원형큐 코드) -----------------------------------------』 int main(void) { int n; QueueType queue; init\_queue(&queue); enqueue(&queue, 0); enqueue(&queue, 1); printf(“계산할 피보나치 수열 횟수를 입력하시오: “); scanf\_s(“%d”, &n); if (n <= 1) { for (int i = 0; i <= n; i++) { printf(“%d “, q->data[i + 1]); } else { printf(“0 1 “); for (int i = 2; i <= n; i++) { int new = dequeue(&queue) + q->data[q->rear]; enqueue(&queue, new); printf(“%d “, q->data[q->rear]; } }

# 11. 회문(palindrome)이란 앞뒤 어느 쪽에서 읽어도 같은 말 · 구 · 문 등을 의미한다. 예를 들면 “eye”, “madam”, “radar” 등이다. 여기서 물론 구두점이나 스페이스, 대소문자 등은 무시하여야 한다. 덱을 이용하여 주어진 문자열이 회문인지 아닌지를 결정하는 프로그램을 작성하라. 다음 그림을 참조한다.

『---------------------------------------- (원형 덱 코드) ----------------------------------------』  
 int main(void) {  
 DequeType deque;  
 init\_deque(&deque);  
 bool flag = TRUE;  
  
 element string[100];  
 element edited[100];  
 int len, len\_edit;  
 printf(“문자열을 입력하시오: “);  
 scanf\_s(“%s”, &string);  
 len = strlen(string);  
  
 for (int i = 0; i < len; i++) {  
 if((isspace(string[i]) == FALSE) && (ispunct(string[i]) == FALSE)) {  
 edited[i] = tolower(string[i]);  
 }  
 len\_edit = strlen(edited);  
 for (int i = 0; i < len\_edit; i++) {  
 add\_rear(&deque, edited[i]);  
 }  
  
 if (len\_edit % 2 == 0) {  
 for (int i = 0; i < (len\_edit / 2); i++) {  
 if (delete\_front(&deque) != get\_front(&deque)) {  
 printf(“회문이 아닙니다”);  
 flag = FALSE;  
 break;  
 }  
 }  
 } else {  
 for (int i = 0; i < ((len\_edit / 2) – 1); i++) {  
 if (delete\_front(&deque) != get\_front(&deque)) {  
 printf(“회문이 아닙니다”);  
 flag = FALSE;  
 break;  
 }  
 }  
 }  
 if (flag) printf(“회문입니다”);  
 return 0;  
}

# 12. 태스크 스케줄링 알고리즘으로 A-Steal 알고리즘이 있다. A-Steal 알고리즘에서 각각의 CPU는 자신이 실행할 태스크가 저장된 덱을 가지고 있다. 하나의 CPU가 자신의 태스크를 종료했다면 다른 CPU가 실행할 태스크를 훔쳐서 실행할 수 있다. 이때 다른 CPU의 덱의 끝에 있는 요소를 가져온다. 간단하게 A-Steal 알고리즘을 구현해보자.

『---------------------------------------- (원형 덱 코드) ----------------------------------------』  
 int main(void) {  
 DequeType deque[3];  
 for (int i = 0; i < 3; i++) init\_deque(&deque[i]);  
  
 add\_rear(&deque[0], “Job1”);  
 add\_rear(&deque[0], “Job2”);  
 add\_rear(&deque[0], “Job3”);  
 add\_rear(&deque[1], “Job4”);  
 add\_rear(&deque[2], “Job5”);  
 add\_rear(&deque[2], “Job6”);  
  
 (UNKNOWN)  
}