**Report**

**For Data Structures # 5**

**학과 :**

**학번 :**

**이름 :**

컴퓨터공학과

20184071

김도현

**문제 1**

* 문제
  + 2개의 스택을 사용하여 큐를 구현하라.
  + 큐를 이용하여 피보나치 수열을 계산하는 프로그램을 2개의 스택을 사용하여 작성하고, 테스트하라.

1. 문제의 요구사항

- 두개의 Stack 을 통해 Queue 를 구현한다

- 그 구현된 Queue 를 통해 피보나치 수열을 구현한다

- 사용자에게 n 번째 항을 입력 받고, 0번째부터 n 번째 항까지의 피보나치 수열을 구한 후 출력을 한다.

2. 문제 분석

|  |  |
| --- | --- |
| Input | N 번째 항을 구해야 하므로 양의 정수 n |
| Output | 양의 정수 n 번째 항까지의 피보나치 수열 |
| Process | 2가지 스택을 이용하여 큐를 구현해야 한다. 그렇기 때문에, 스택에 대한 연산을 먼저 구현해야 한다. 스택의 연산을 구현했으면, 큐를 구현해야 한다.  2가지 스택을 이용하여 큐를 구현하는 방법은, 스택의 성질을 이용해야 한다. 스택은LIFO의 성질인, 후입선출 방식을 갖는다. 그러나 큐는 FIFO 성질인, 선입선출 방식을 갖는다. 즉 스택은 1 2 3 이 차례로 들어왔으면, 출력이 3 2 1 순으로 (역순으로) 진행되고, 큐는 1 2 3 이 차례로 들어왔으면, 출력이 1 2 3 순으로 진행된다.  이 성질을 가지고 스택 2개로 큐가 구현이 가능하다. 구현 방법은 다음과 같다.  예를 들어 1 2 3 이 들어왔으면 1 2 3 순으로 출력해야 하므로, 처음 스택에 1 2 3을 저장한다. 이 처음 스택은 3 2 1 순으로 출력되게 되는데, 이 처음 스택을 출력하면서 두번째 스택에 넣으면 된다. 즉, 1 2 3 이 있던 것을 두번째 스택에 넣으면 3 2 1 이 될 건데, 이를 2번째 스택에서 pop을 수행하면, 1 2 3 순으로 출력된다. 즉, 스택을 가지고 입력된 데이터를 2번 뒤집는 행위를 수행하는 것이다.  그러나 아무 때나 처음 스택의 값을 두번째로 옮기면 안된다.  예를 들어 1 2 pop 3 4 가 들어왔는데, 처음 스택에 1 2 를 넣고, pop을 수행하면 1 이 반환되야 하므로 처음 스택은 빈 값이 되고 두번째 스택은 2 1 순이 되게 될 것이다. 그후 pop을 하면 1 이 삭제되어 두번째 스택은 2가 될 것이다. 근데 3 4 를 넣고, pop을 하면 2 가 나와야 하므로, 아무 때나 처음 스택의 값을 두번째 스택에 옮기면 안된다.  **그렇기 때문에 두번째 스택이 비었을 때만 처음 스택의 값을 두번째 스택에 옮겨야 한다.**  그러므로 스택 두개로 큐를 구현하는 알고리즘은 다음과 같다.  첫번째로 enQueue 함수다. enQueue 함수는 다음과같이 구현할 수 있다.  1. 처음 스택이 가득 찼나 확인한다. 가득 찼으면 enQueue 를 수행할 수 없다.  2. 처음 스택이 가득 차지 않았으면, 처음 스택에 입력 받은 값을 넣는다.  두번째로 deQueue 함수다. deQueue 함수는 다음과 같이 구현할 수 있다.  1. 두번째 스택이 비었나 확인한다. 비지 않았으면 두번째 스택의 pop 을 수행하고 반환한다.  2. 스택이 비었으면, 처음 스택이 비었나 확인한다. 비었으면, 큐에 값이 없으므로, 프로그램을 종료한다  3. 그 외의 경우, 즉, 두번째 스택엔 값이 없고 처음 스택엔 값이 있는 경우다. 이 경우 while 문을 통하여 처음 스택의 값을 모두 두번째 스택에 옮긴다. 모두 옮겼으면, 두번째 스택에서 pop 을 수행하고 반환한다.  여기 까지가 스택 두개로 큐를 구현하는 방법이다. 피보나치 수열을 구하는 방법은 다음과 같다.  1. 입력받은 n 이 0이거나 1인지 확인한다. 0이거나 1이면 그냥 n 을 반환한다.  2. 입력받은 n 이 0이거나 1이 아닌 경우, 0부터 n 까지 반복을 한다. 반복을 할 때, 값이 두가지가 기본적으로 있어야 하므로 0이거나 1일때는 enQueue 를 수행하여 큐에 값을 넣는다.  3. 0이거나 1이 아닌 경우엔, 두번의 deQueue 연산을 통해 전전항과 전항을 가져온다. 지금 항은 전전항과 전항을 더한 값이므로, 계산을 진행한다. 그 후, 전전항은 다음 피보나치 수열의 항을 구하는데 필요 없으므로, 전항과 지금항을 enQueue 연산을 통해 큐를 넣는다.  4. 3 번의 작업을 n 까지 진행했으면 큐엔, n -1 번째 항, n 번째 항의 값이 저장되어 있다. 그렇기 때문에 dequeue 를 한번 수행하여 큐에 n 번째 항만 저장되게 한다. 그 후, deQueue 를 한 값을 반환하면 n 번째 항이 반환된다.  피보나치 수열까지 구현이 완료되었다. 그러나 피보나치 수열은 n 번째 항은 항상 0<=n 이므로 do while 문을 사용하여 0보다 작은 수가 들어올 수 없게 한다. |
| Example | 1. n 이 음수인 경우  2. n 이 0인 경우  3. n 이 1인 경우  4. n 이 2이상인 경우  해당 결과는 5번째 실행결과에서 확인 할 수 있다. |

3. 의사 코드를 사용하여 알고리즘 작성

큐의 연산 구현

Queue {

Stack InputStack

Stack OutputStack

}

Procedure initQue(Q)

Q.InputStack.top <- -1

Q.OutputStack.top <- -1

End initQue

Procedure enQueue(Q, int n)

Push(Q.inputStack, n)

End enQueue

Procedure deQueue(Q)

If !is\_empty(Q.OutputStack) then return pop(Q.outputStack)

Else if (is\_empty(Q.inputStack) then Exit()

Else

While (!is\_empty(Q.inputStack))

Push(Q.OutputStack, pop(Q.inputStack))

Repeat

Return pop(Q.OutputStack)

endIf

end deQueue

큐의 연산 구현 끝

피보나치 수열 구현

Procedure fibo(int n)

If n = 0 or n = 1 then return 1

endif

Q = Queue()

initQueue(Q)

for I <-0 by n i++

if I = 0 or I = 1 then enQueue(Q,i)

else

int first <- deQueue(Q)

int second <- deQueue(Q)

int temp <- first + second

enQueue(Q,second)

enQueue(Q,temp)

endIF

repeat

deQueue(Q)

return deQueue(Q)

end fibo

4. C 언어로 작성한 프로그램

#include <stdio.h>

#define MAX 100

typedef char elemant;

typedef struct { //스택 구조체를 정의하는 부분

elemant data[MAX];

int top;

}stack;

typedef stack\* stack\_ptr; //포인터를 좀더 쓰기 쉽게 stack\_ptr 정의

typedef struct { //스택을 2개를 사용하는 큐 정의

stack InputStack; //큐에는 스택이 2개가 있다.

stack OutputStack;

} Queue;

//함수 원형을 정의하는 부분

typedef Queue\* Queue\_ptr;

void initStack(stack\_ptr inputStack);

void initQueue(Queue\_ptr q);

void enQueue(Queue\_ptr q, elemant data);

elemant deQueue(Queue\_ptr q);

int is\_empty(stack inputStack);

int is\_full(stack inputStack);

elemant peek(stack inputStack);

void push(stack\_ptr inputStack, elemant data);

elemant pop(stack\_ptr inputStack);

//함수 원형을 정의하는 부분 끝

int main() {

int times;

do {

printf("구할 피보나치 수열의 항을 구하세요 (양수) : ");

scanf("%d", &times);

} while (times < 0); //do while 문을 사용하여 음수가 들어오는 경우를 막는다.

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

printf("%d ", fibo(i)); //피보나치 수열의 times 번째 항까지 구한후 출력

}

return 0;

}

//-------------스택 관련 함수 정의하는 부분 시작----------------

//스택 초기화 함수, 구조체의 top 을 -1 로 설정

void initStack(stack\_ptr inputStack) {

inputStack->top = -1;

}

//스택이 비었나 검사하는 함수

int is\_empty(stack inputStack) {

return (inputStack.top == -1); //top 이 -1 이면 1 아니면 0 반환

}

//스택이 가득찼나 검사하는 함수

int is\_full(stack inputStack) {

return (inputStack.top == MAX - 1); //top 이 배열의 최대치면 1 아니면 0 반환

}

//스택의 최상단을 확인하는 함수 , 해당 프로그램에선 사용하지 않는 함수

elemant peek(stack inputStack) {

if (is\_empty(inputStack)) { //스택이 비었으면 return 할수 없으므로 프로그램 종료

printf("peek 을 수행할수 없습니다.");

exit();

}

return inputStack.data[inputStack.top]; //스택이 안비었으면 상단의 값 반환

}

//스택에 값을 넣는 함수

void push(stack\_ptr inputStack, elemant data) {

if (is\_full(\*inputStack)) { //스택이 가득찼으면 push 를 할수 없다.

printf("push 를 수행 할 수 없습니다.");

exit();

}

inputStack->data[++(inputStack->top)] = data; //스택이 가득차지 않았으면 top 을 1 증가후 값 삽입

}

//스택에 값을 제거하는 함수

elemant pop(stack\_ptr inputStack) { //스택이 비었으면 pop 을 할수 없다

if (is\_empty(\*inputStack)) {

printf("더이상 pop 을 수행 할 수 없습니다");

exit();

}

return inputStack->data[(inputStack->top)--]; //스택이 안비었으면 값을 반환후 top 을 1 감소시킨다.

}

//-------------스택 관련 함수 정의하는 부분 끝----------------

//------------큐 관련 함수 정의하는 부분 시작-------------------

//큐를 초기화 하는 함수

void initQueue(Queue\_ptr q) {

//Queue 구조체에 존재하는 input outputStack 을 초기화 해준다.

initStack(&q->InputStack);

initStack(&q->OutputStack);

}

//큐에 데이터를 넣는 함수

void enQueue(Queue\_ptr q, elemant data) {

//큐에 데이터를 넣는 함수인데, 데이터는 inputStack 에 먼저 넣어야한다.

// 1 2 3 을 넣었으면 1 2 3 이 출력되야 하므로

//inputStack 엔 1 2 3 순으로 저장되어야 한다.

push(&q->InputStack, data); //inputStack 에 값을 넣는다.

//inputStack 이 full 인 경우의 동작은 push 에서 정해져있다.

}

//큐에서 데이터를 가져오는 함수

elemant deQueue(Queue\_ptr q) {

//OutputStack 이 비지 않았으면

if (!is\_empty(q->OutputStack)) {

return pop(&q->OutputStack); //outPut 스택의 마지막으로 들어간 값을 반환한다.

}

else if (is\_empty(q->InputStack)) { //만약 OutputStack 과 inputStack 둘다 비었으면

printf("deQueue 를 수행할수 없습니다.");// deQueue 를 수행할수 없다

exit();

}

else {//inputStack 이 안비었을때 경우

while (!is\_empty(q->InputStack)) { //inputStack 이 빌때까지 수행

//inputStack 에 있는 값을 outputStack 에 옮긴다.

//즉 inputStack 이 1 2 3 으로 저장되어있고 OutputStack 이 값이 없으면

// inputStack 은 값이 없어지고 OutputStack 은 3 2 1 이 된다

// 즉 먼저 들어왔던 1 과 2 3 순으로 반환된다 >> 큐의 구조를 볼수 있다.

push(&q->OutputStack, pop(&q->InputStack));

}

//옮긴게 끝났으면 OutputStack 의 최상단을 반환한다

return pop(&q->OutputStack);

}

}

//------------큐 관련 함수 정의하는 부분 끝-------------------

int fibo(int n) { //n 번째 항까지 구하는 피보나치 수열

if (n == 0 || n == 1) { //피보나치 수열이 1이거나 0번쨰 항을 구하는 경우

return n; //그냥 n을 반환한다

}

//아래는 n 이 0 또는 0이 아닌 경우다

Queue a; //큐를 생성해주고

Queue\_ptr q; //큐를 조금더 쉽게 ( & 를 쓰지 않기 위해) Queue\_ptr 을 생성

q = &a; //q 는 a 의 포인터

initQueue(q); //q 를 초기화

for (int i = 0; i <= n; i++) { //n 번쨰 항까지 계산해야 하므로

if (i == 0 || i == 1) {

enQueue(q, i); //i가 0과 1인 경우엔 큐에 0 또는 1을 넣는다.

}

else { //i 가 0 1 이 아닌 경우

int first = deQueue(q); //전 전 항 > 1 2 이면 1

int second = deQueue(q);// 전항 > 1 2 이면 2

int temp = first + second; //피보나치는 전항과 2번쨰 전항을 더하는 것이다.

enQueue(q, second); //second 가 이제 맨 앞의 수가 되고

enQueue(q, temp); //temp 가 그 다음수가 된다

}

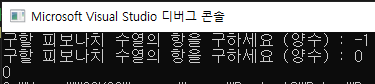
}

deQueue(q); //for 문이 끝나면 큐엔 전항, 지금항 이 저장되어있으므로 전항을 지운다

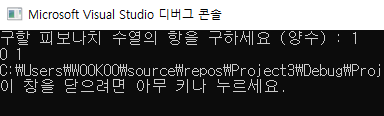
return deQueue(q); //지금항만 저장되어있으므로 그 값을 반환한다.

}

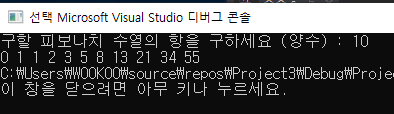
5. 실행 결과



1. 번인 n 이 음수인 경우와 2번인 0인 경우. 음수인 경우 다시 입력하고, 0인 경우 0이 반환된다



3. 피보나치 수열이 1인 경우 1번쨰 항까지 인 0 번째 항 1번째 항인 0 1 이 출력된다.



4. 피보나치 수열이 10 인 경우. 0번째부터 10번째인 0 1 1 2 3 5 8 13 21 34 55 가 출력된다.

6. 의견

큐를 가지고 구현하는 게 더 쉽고 간편한 것 같다. 스택 2개를 사용하여, 구현하기 위해서 조금 더 생각 해봐야하는 문제는 있었다. 큐의 성질과, 스택의 성질을 정확하게 이해하고 있으면, 어렵지 않게 큐를 구현할 수 있었다. 또한, 2주전 실습에서 큐를 이용한 피보나치 수열을 그대로 가져와서 작성할 수 있었으므로, 구현 시간은 더더욱 짧았다.

해당 소스는

<https://github.com/wookoo/Data-Structure/blob/master/5.%ED%81%90/5.%20%EC%8A%A4%ED%83%9D%20%EB%91%90%EA%B0%9C%EB%A1%9C%20%EA%B5%AC%ED%98%84%ED%95%9C%20%ED%81%90%EB%A1%9C%20%ED%94%BC%EB%B3%B4%EB%82%98%EC%B9%98%20%EC%88%98%EC%97%B4.c> 에서 확인 가능합니다.

QR 코드를 스캔하시면 해당 소스로 이동 가능합니다.



해당 소스로 이동하는 QR 코드

**문제 2**

* 문제
* 덱을 이용하여 주어진 문자열이 회문인지를 판단하는 프로그램을 작성하고, 테스트하라.

1. 문제의 요구사항

- Stack 의 LIFO 와 Queue 의 FIFO 성질을 동시에 가진 Deque 을 구현한다.

- 구현된 Deque 로 문자열이 회문인지 판별한다

- 단, Front 단에서 가져오는 deleteFront 와 Rear 단에서 가져오는 delete\_rear 함수를 가지고 구현해야 한다. 그렇지 않으면 과제의 취지인 Deque을 사용하여 구현해야 되는 의미가 없어진다.

2. 문제 분석

|  |  |
| --- | --- |
| Input | 회문을 판별할 문자열 변수 String |
| Output | String 의 회문의 유무 |
| Process | 1. 덱을 구현한다  2. get\_symbol 혹은 String 헤더의 strlen 함수를 통하여 for 문 또는 while 을 가지고 입력된 String 을 모두 덱에 삽입한다  3. 만약 데이터가 abba 라고 가정하면 덱엔 다음과 같은 꼴로 저장되어 있을 것이다.   |  |  | | --- | --- | | A | a | | b | b |   그러면 전단과 후단에서 값을 가져오는 delete\_front 와 delete\_rear 함수를 호출해서 각각 비교한다.  4. 두 연산을 수행하면 덱은 다음과 같은 꼴로 저장되어 있을 것이다.   |  |  | | --- | --- | |  |  | | b | B |   전단과 후단에서 가져온 변수 두가지가 일치하나 확인한다. 일치하면 또 delete\_front 와 delete\_rear 를 사용하여 전단과 후단의 값을 또 비교한다. 이 작업을 덱이 빌 때 까지 반복하는데, 덱이 비어버리면 입력받은 String 은 회문이다.  5. 회문이 아닌 경우는 다음과 같다. 덱이 비어있지 않은 상황에서 전단과 후단의 데이터가 다를 경우엔, 회문이 아님을 출력해주면 된다.  6. 문자열이 홀수일때는, delete\_rear 가 정상적으로 수행되었는데, delete\_front 가 덱에값이 없어서 정상적으로 수행되지 않을 경우다. 이 경우도 회문이다.  덱을 구현할 때 주의해야 할 점은 다음과 같다.  Delete\_rear 함수는 맨 뒤의 데이터를 지우는 것이 때문에 데이터를 삭제 후, 큐를 시계 반대 방향으로 회전시켜야 한다.  Add\_front 함수는 맨 앞에 데이터를 추가하는 것이기 때문에 데이터를 추가 후, 큐를 시계 반대방향으로 회전(front 추가) 해야 한다. |
| Example | 1. 회문인 String 이 abccba (짝수) 인 경우  2. 회문인 String 이 abcdcba (홀수) 인 경우  3. 회문이 아닌 String 이 abcd 인 경우 |

3. 의사 코드를 사용하여 알고리즘 작성

덱의 연산 구현

Procedure init(DQ)

DQ.front <- 0

DQ.rear <- 0

End init

Procedure is\_empty(DQ)

Return DQ.front = DQ.rear

End Is\_empty

Procedure is\_full(DQ)

Return (DQ.rear + 1 ) % 큐의 최대 크기 = DQ.front

End is\_full

Procedure delete\_front(DQ)

If (is\_empty(DQ) then 프로그램 종료

Else

DQ.front <- (DQ.front +1) % 큐의 최대 크기

Return DQ.data[DQ.front]

endIF

end delete\_front

Procedure add\_rear(DQ, N)

If (is\_full(DQ)) then 프로그램 종료

Else

DQ.rear <- (DQ.rear +1) % 큐의 최대 크기

DQ.data[DQ.rear] <- N

End if

End add\_rear

Procedure add\_front(DQ, N)

If (is\_full(DQ)) then 프로그램 종료

Else

DQ.data[DQ.front] <- N

DQ.front <- (DQ.front -1 + 큐의 최대크기 ) % 큐의 최대 크기

endIF

end add\_front

Procedure delete\_rear(DQ)

If (is\_empty(DQ)) then 프로그램 종료

Else

Item <- DQ.data[DQ.rear]

DQ.rear <- (DQ.rear -1 + 큐의 최대 크기) % 큐의 최대 크기

endIF

end delete\_rear

회문 판별 구현

IsTrue <- True

While ((sym <- get\_symbol(String)) != NULL) do //덱에 모든 정보를 넣는 작업

Add\_rear(DQ,sym)

Repeat

While(!is\_empty(DQ)) do

Char front <- delete\_front(DQ)

If (!is\_empty(DQ)) then

Char rear <- delete\_rear(DQ)

If (rear != front) then

IsTrue <- False

Break;

endIF

endIF

repeat

if(IsTrue) then print(“회문”)

else then print(“회문 아님”)

EndIf

4. C언어를 이용한 프로그램 구현

#include <stdio.h>

#include <stdlib.h>

#define MAX\_QUE\_SIZE 10

typedef int element;

typedef struct { //원형 큐가 가지는 front rear 필드

int front;

int rear;

element data[MAX\_QUE\_SIZE]; //데이터를 저장하는 배열

} Deque;

typedef Deque\* Deque\_ptr;

//함수 원형 정의 부 시작

void error(char \*String);

void init(Deque\_ptr dq);

int is\_empty(Deque\_ptr dq);

int is\_full(Deque\_ptr dq);

element delete\_front(Deque\_ptr dq);

void add\_rear(Deque\_ptr dq, element data);

void add\_front(Deque\_ptr dq, element data);

element delete\_rear(Deque\_ptr dq);

char get\_symbol(char \*String);

//함수 원형 정의부 끝

int main() {

Deque a;

Deque\_ptr dq = &a;

init(dq);

char String[] = "abcba";

char sym;

int IsTrue = 1; //String 이 참인지 거짓인지 확인하기 위한 변수

while ((sym = get\_symbol(String)) != NULL) {

add\_rear(dq, sym); //String 의 모든 요소를 덱에 삽입

}

while (!is\_empty(dq)) { //덱이 비지 않을때 까지 반복

char front;

front = delete\_front(dq); //일단 값이 무조건 존재 하므로 전면에서 값을 가져온다

if (!is\_empty(dq)) { //문자열이 짝수인 경우

char rear;

rear = delete\_rear(dq); //후면에서 값을 가져온다

if (rear != front) { //전면부와 후면부 즉, 처음값과 마지막 값이 같지 않으면

IsTrue = 0; //그것은 회문이 아니기에 0으로 만들고

break; //while 문종료

}

}

}

if (IsTrue) { //회문인 경우와 회문이 아닌 경우를 출력

printf("회문");

}

else {

printf("회문 아님");

}

return 0; //메인함수 종료

}

void error(char \*String) { //오류 출력 함수

printf("%s", String); //입력받은 오류를 출력하고 프로그램 종료

exit(1);

}

void init(Deque\_ptr dq) { //덱을 초기화, front 와 rear 을 0으로 바꿈

dq->front = 0;

dq->rear = 0;

}

int is\_empty(Deque\_ptr dq) { //덱이 빈 경우는 front 와 rear 가 같은 경우다

return dq->front == dq->rear;

}

int is\_full(Deque\_ptr dq) { //덱이 가득 찬 경우는 rear 에서 1을 더한 값이 front 와 같은경우

return (dq->rear + 1) % MAX\_QUE\_SIZE == dq->front;

}

element delete\_front(Deque\_ptr dq) { //덱의 앞에서 값을 지우는 경우

if (is\_empty(dq)) {

error("덱이 비었습니다!");

}

//덱이 가득 비지 않았으면

dq->front = (dq->front + 1) % MAX\_QUE\_SIZE; //시계방향으로 덱을 돌리고

return dq->data[dq->front]; //값을 반환

}

void add\_rear(Deque\_ptr dq, element data) {

if (is\_full(dq)) {

error("덱이 가득 찼습니다.");

}

//덱이 가득차지 않았으면

dq->rear = (dq->rear + 1) % MAX\_QUE\_SIZE; //시계방향으로 덱을 돌리고

dq->data[dq->rear] = data; //값을 삽입

}

void add\_front(Deque\_ptr dq, element data) {

if (is\_full(dq)) {

error("덱이 가득 찼습니다.");

}

//덱이 가득 차지 않았으면

dq->data[dq->front] = data; //값을 삽입하고

dq->front = (dq->front - 1 + MAX\_QUE\_SIZE) % MAX\_QUE\_SIZE; //시계 반대방향으로 덱을 돌림

}

element delete\_rear(Deque\_ptr dq) {

if (is\_empty(dq)) {

error("덱이 비었습니다!");

}

//덱이 비지 않았으면

element item = dq->data[dq->rear]; //값을 가져오고

dq->rear = (dq->rear - 1 + MAX\_QUE\_SIZE) % MAX\_QUE\_SIZE; //시계 반대방향으로 덱을 돌리고

return item; //값을 반환

}

char get\_symbol(char \*String) { //for strlen(String) 과 비슷함

static int index = 0; //static 변수 선언, 함수 call 마다 1 씩 increase

if (String[index] != NULL) { //NULL 값이 아니면

return String[index++]; //반환후 index 1 증가

}

else

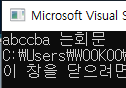
{

return NULL;

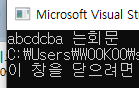
}

}

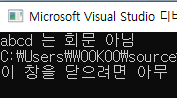
5. 실행결과



1 .String 이 짝수고, 회문인 경우



2. String 이 홀수고, 회문인 경우



3. String 이 회문이 아닌 경우

6. 의견

스택으로 회문인지 판별하는 것은 이미 중간고사 시험 문제에 나온 터라 어렵지 않게 구현할 수 있었다.

원래는 중간고사때 스택을 이용하여 구현 한 것 처럼, delete\_front 함수를 안 쓰고 덱을 스택처럼 사용하여 구현을 해볼까 생각해보다, 이는 문제의 취지인 덱을 사용하기 와 맞지 않아서 다른 알고리즘인 delete\_front , delete\_rear 를 통해 구현을 하였다

또한, 스택은 값을 다 넣고 또 index 하나 하나 스택의 값과 String의 값을 비교해야 되지만, 덱 같은 경우 rear 값과 front 값을 가져와서 확인하면 비교하면 되기 때문에 조금 더 융통성 있고, 직관적이게 알고리즘을 생각 할 수 있었다.

해당 소스는

<https://github.com/wookoo/Data-Structure/blob/master/5.%ED%81%90/6.%20%EB%8D%B1%EC%9D%84%20%EC%9D%B4%EC%9A%A9%ED%95%9C%20%ED%9A%8C%EB%AC%B8%20%ED%8C%90%EB%B3%84.c> 에서 확인 가능합니다.

또한 아래의 QR 코드를 통해서도 확인이 가능합니다.



소스코드로 이동하는 QR 코드