제 3 장 스택과 큐

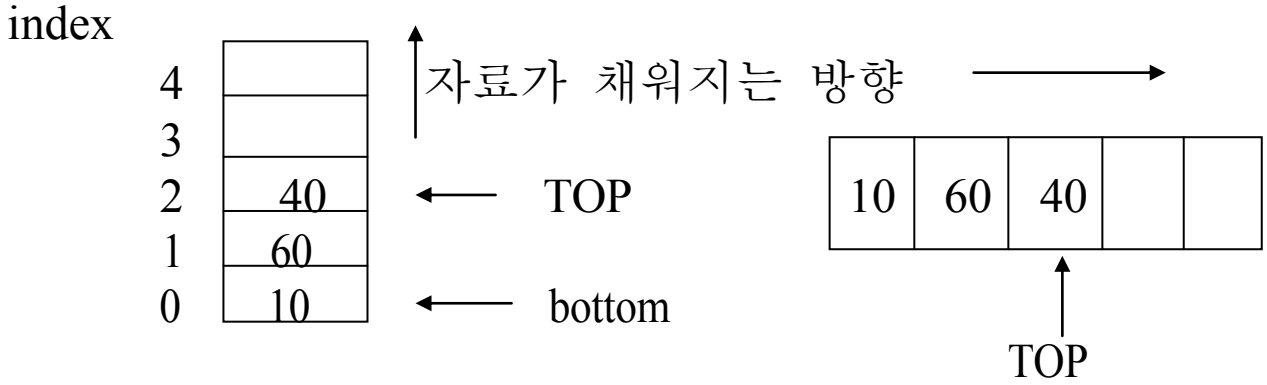
1. STACK 의 정의

- 모든 데이터의 삽입과 삭제가 한쪽 끝에서만 일어나는 리스트 따라서 stack 내의 임의의 곳에 자료를 삽입 또는 삭제 불가
- stack 에 저장되는 자료형은 배열과 같이 동일 해야 함
- LIFO(Last in First Out) 리스트라고 한다.

  (예: 음식점의 접시 쌓기: 새로닦은 접시는 맨위로 정렬, 사용은 맨위의 접시부터 사용)
- stack 은 배열(array)과 링크드 리스트(linked-list)로 구현된다.
- Stack 을 배열로 구현 하면 배열의 크기가 한정되어, 정적 스택 (static stack) 이라고 하고, 링크드 리스트로 구현하면 스택의 크기가 가변적이어서 동적 스택 (dynamic stack) 이라 한다.

1.1. 기본 개념

그림은 10, 60, 40 세개 정수가 저장된 stack 의 구조이다. stack 의 맨 위를 top 이라고 하고 맨 아래를 bottom 이라고 하며, stack 의 크기는 5 이다.



10, 60, 40 의 3 개의 데이터가 저장된 스택이 모습, 스택의 크기는 5

1.2. 알고리즘 [스택 ADT]

---

structure *Stack*

objects: 0개 이상의 원소를 가진 유한 순서 리스트

functions:

***Stack* CreateS(max_stack_size) ::=**

최대 크기가 max_stack_size인 공백 스택을 생성

***Boolean* IsFull(stack, max_stack_size) ::=**

boolean 쓰는 이유는 내가 넣고 싶은 자리에 자리가 있는지 확인하려고

if (stack의 원소수 == max_stack_size) return TRUE

else return FALSE

***Stack* Add(stack, item) ::=**

if (IsFull(stack)) stack_full

else stack의 톱에 item을 삽입하고 return

***Boolean* IsEmpty(stack) ::=**

if (stack == CreateS(max_stack_size)) return TRUE

else return FALSE

***Element* Delete(stack) ::=**

if (IsEmpty(stack)) return

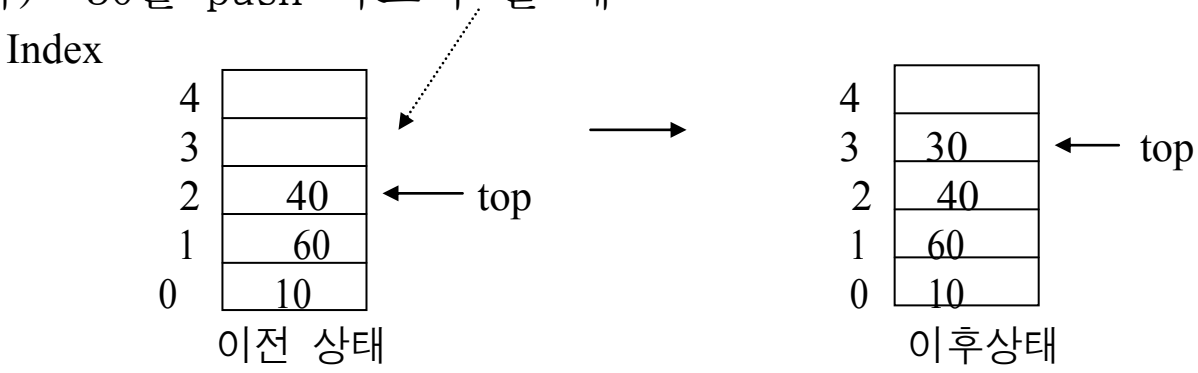
else 스택 top의 item을 제거해서 반환

```
Stack CreateS(max_stack_size) ::=
        #define MAX_STACK_SIZE 100 /*최대스택크기*/
        typedef struct {
                        int key;
                       } element;

        element stack[MAX_STACK_SIZE];
        int top = -1;
  Boolean IsEmpty(Stack) ::= top < 0;
  Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE-1;
```

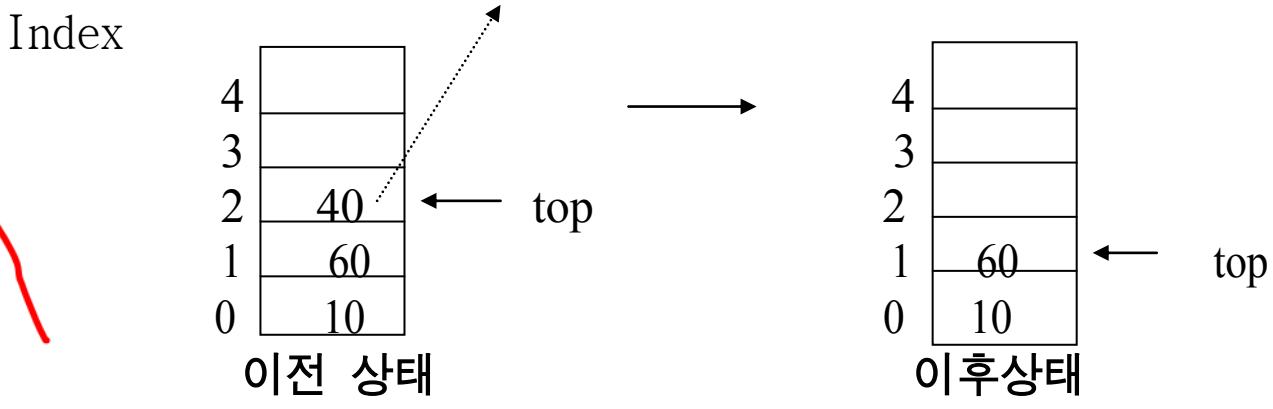
```
void add(int *top, element item)                    (PUSH)
{   /* 전역 stack에 item을 삽입 */
    if (*top >= MAX_STACK_SIZE-1) {
        stack_full();     return;
    }
    stack[++*top] = item;   }
```

예) 30을 push 하고자 할 때



```
element delete(int *top)                            (POP)
{   /* stack의 최상위 원소를 반환 */
    if (*top == -1)
        return stack_empty();        /* 오류 key를 반환 */
    return stack[(*top)--];   }
```

예) 40을 pop 하고자 할 때



스택 프로그램 (Procedural, not Class)

```
/*****************************************************/
// File Name:        ARRAYSTK1.CPP                    //
// Description :   <Array Implementation of a Stack>  //
//                  Class 사용하지 않은 code          //
/*****************************************************/
.............................
const int stackSize = 3;
int stack[stackSize];   int top;

void main()
{  void create_stack(), push(int), traverse_stack();
   int pop(); int isFull(), isEmpty();
   int num; char input[10];

   create_stack();

   while (1) {
      cout << "Enter the command(push, pop, traverse, exit):";
      cin >> input;

      if (strcmp(input, "push") == 0) {
          if (!isFull()) {
             cout << "Enter an integer to push => ";
             cin >> num;
             push(num);    }
          else    cout << "Stack is full!₩n";
      }
      else if (strcmp(input, "pop") == 0) {
           if (!isEmpty()) {
              num = pop();
              cout << num << " is popped.₩n";
           }
           else    cout << "Stack is empty!₩n";
```

이 코드 자세히 보고 코드 참조해서 연구할것

포인터 초기화 데이터 추가 데이터 출력

```
        }
         else if (strcmp(input, "traverse") == 0) displayStack();
         else if (strcmp(input, "exit") == 0) exit(0);
         else cout << "Bad Command!₩n";
    }
}

void create_stack() {  top = -1; }

int isFull() {
      if (top == stackSize - 1)   return 1;
      else    return 0;  }

int isEmpty() {
      if (top == -1)   return 1;
      else    return 0; }

void push(int num) {
      ++top;  stack[top] = num;
}

int pop() {  return (stack[top--]); }


void displayStack()
{   int sp;
    if (isEmpty())  cout << "Stack is empty!" << endl;
    else {
         sp = top;
         while (sp != -1) {
               cout <<   stack[sp];   --sp;
         }
        cout << endl;
    }
}
```

```
// classarraystack1.cpp   (Class)
// Stack implementation with arrays

const int Stack_Size = 4;

class Stack {
  private:
      int stack[Stack_Size];   int top;
  public:
      Stack() {top = -1;}
      void push(int val) {stack[++top] = val;}
      int pop()   {return stack[top--];}
      int isEmpty() {return top == -1;}
      int isFull()  {return top == Stack_Size - 1;}
      void displayStack();
  };

void Stack::displayStack()
{
      int sp;    sp = top;
      while (sp != -1) { cout <<   stack[sp--]; }
      cout << endl; };

void main()
{     Stack s1;
      s1.push(10); s1.push(20); s1.push(30); s1.push(40);
      s1.displayStack();

      if (s1.isFull())   cout << "Stack is full₩n";

      cout << "Pop: " << s1.pop() << endl;
      cout << "Pop: " << s1.pop() << endl;
      cout << "Pop: " << s1.pop() << endl;
      cout << "Pop: " << s1.pop() << endl;
      if (s1.isEmpty())
          cout << "Stack is  empty₩n"; }
```

2. 수식의 계산 (Evaluation of Expression)

수식의 표현

- 중위 표기(*infix notation*) : a * b / c
- 후위 표기(*postfix notation*) : a b * c /
- 전위 표기(*prefix notation*) : / * a b c

1) Postfix Evaluation (후위표기)

. Infix 표기는 가장 보편적인 수식의 표기법,
. 대부분의 compiler 는 후위 표기법을 사용한다.
- 괄호(parenthesis) 사용 안함
- 연산자의 우선순위 없음 (L -> R 순서의 계산)
- 계산이 간편

● Algorithm:
1. Initialize stack
2. Repeat until end-of-expression
   . Get next token
   . If **"token = operand"** then PUSH onto Stack
   else "**token=operator**"
   . POP two operands from stack
   . Apply the operator to these
   . Push the results onto stack

1+2 에서
1, 2 는 operand
+는 operator

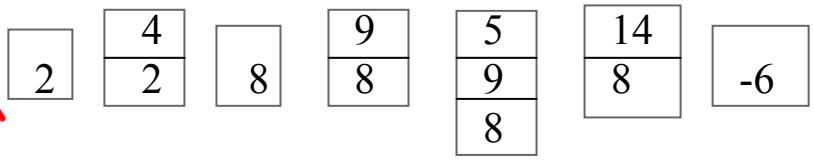
3. When end-of-expression, its value(result) is on top of Stack

ex) 24*95+- (7character string)

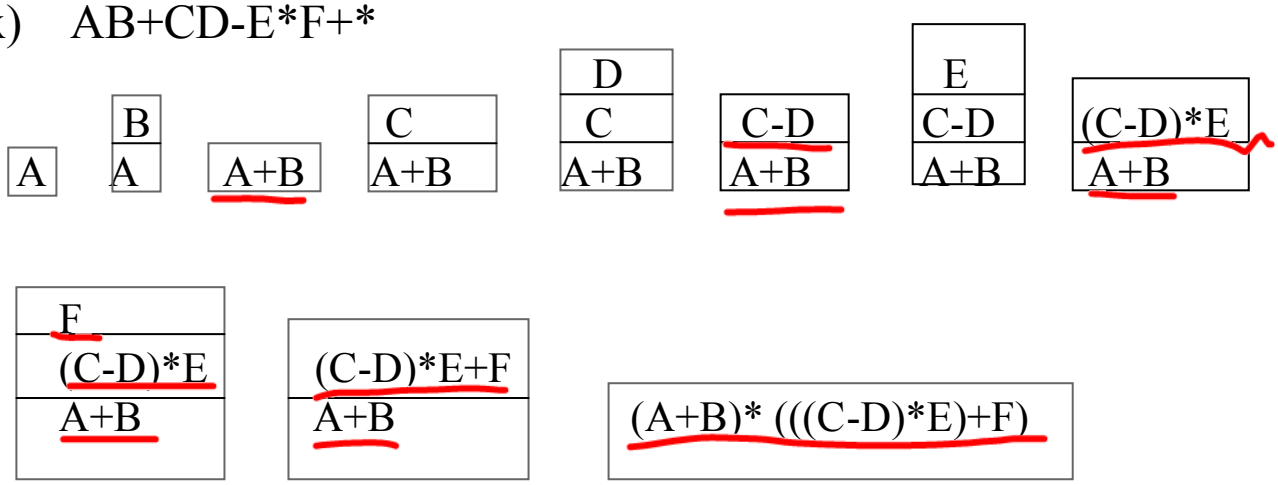
=(2×4)-(9+5)

Y=stack
X= stack

Stack=X op Y



ex) AB+CD-E*F+*



* Postfix Evaluation

```
typedef enum { lparen, rparen, plus, minus, times, divide,mod, eos, operand }
            precedence;
int stack[MAX_STACK_SIZE];   /* 전역 배열 */
char expr[MAX_EXPR_SIZE];    /* 입력 스트링 */
```

```
int eval(void)
{
    precedence token;      char symbol;      int op1,op2;
    int n = 0; /* 수식 스트링을 위한 카운터 */      int top = -1;

    token = get_token (&symbol,  &n);

    while (token != eos) {
        if "token == operand"
            add(&top, symbol-'0'); / *스택 삽입 (PUSH) */
        else {
```

0을 빼주는 이유:ASCII코드값에 맞춰주려고

```
    /* 두 피연산자를 삭제하여 연산을 수행후, 결과를 스택에 삽입 */
            op2 = delete(&top);     /* 스택 삭제 (POP)*/
            op1 = delete(&top);
```

```
            switch(token) {
                case plus:      add(&top, op1+op2);     break;
                case minus:     add(&top, op1-op2);     break;
                case times:     add(&top, op1*op2);     break;
                case divide:    add(&top, op1/op2);     break;
                case mod:       add(&top, op1%op2);
            }
        }
        token = get_token(&symbol, &n);
    }
    return delete(&top); /* 결과를 반환 */
}
```

```
precedence get_token (char *symbol, int * n)
{
/* 다음 토큰을 취한다. symbol은 문자 표현이며, token은 그것의 열거
된 값으로 표현되고, 명칭으로 반환된다. */

    *symbol = expr[(*n)++];

    switch (*symbol) {
        case '(' :   return lparen;
        case ')' :   return rparen;
        case '+' :   return plus;
        case '-' :   return minus;
        case '/' :   return divide;
        case '*' :   return times;
        case '%' :   return mod;
        case ' ' :   return eos;
        default :   return operand;
    }
}
```

2) Infix to Postfix conversion

1. Initialize stack
2. While NOT end-of-expression
   . Get next token

   . If token is
   "(" : then PUSH

   ")" : then POP and display elements in stack until left parenthesis is encountered

   Operator: if "**token (higher priority)** >> **top element**" then **PUSH token** onto stack
   else **POP and Display** top element
   PUSH token onto Stack

   Operand: Display
3. End-of-expression, then POP and Display until stack is empty

ex) 7*8-(2+3)$

Input	stack	output
7		7
*	*	7
8	*	7 8
-	-	7 8 *
(	(-	7 8 *
2	(-	7 8 * 2
+	+ (-	7 8 * 2
3	same	7 8 * 2 3
)	-	7 8 * 2 3 +
$		7 8 * 2 3 + -

&만나면 while문을 돌려 stact안에 있는거 다 출력(꺼냄)`

수식 받아서 이거하는거 실습

Priority: 우선순위

)	3	+, -	1
*, /	2	(	0

ex) A*(B+C)*D$ => ABC+*D* 연습할것

---

isp(in-stack precedence)와 icp(incoming precedence)

precedence stack[MAX_STACK_SIZE];

/* isp 와 icp 배열 -- 인덱스는 연산자의 우선순위 값 */

static int **isp[ ]** = {.....};

static int **icp[ ]** = { .... };

---

void **postfix** (void) 이거 집에서 꼭 해볼것. 3/25 LAB에 나옴.

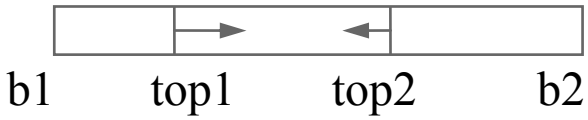
```
{
    char symbol;        precedence token;     int n = 0;
    int top = 0; /* eos 를 스택에 놓는다. */     stack[0] = eos;

   for (token= get_token(&symbol,&n);   token!=eos;
                token = get_token(&symbol,&n))
    {
        if (token = operand)       printf("%c", symbol);
        else if (token = rparen) {
              while (stack[top] != lparen)   /*왼쪽 괄호가 나올 때까지*/
                 print_token(delete(&top));   /*토큰들을 제거해서 출력시킴*/
                           delete(&top); /* 좌괄호를 버린다 */
            }
       else { /* symbol 의 isp token 의 icp symbol 을 제거하고 출력 */
            while (isp[stack[top]] >= icp[token])     print_token(delete(&top));
            add(&top, token);   }
    }
    while ((token=delete(&top)) != eos)     print_token(token);
    printf("\n");   }
```

---

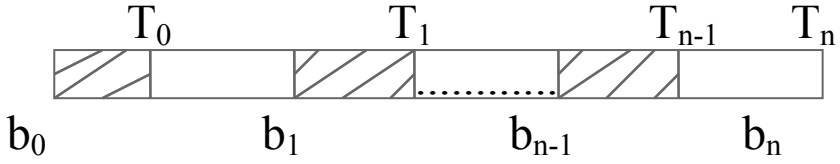
3. 다중 스택과 큐

ex) 하나의 array 에 두개의 stack 사용할 경우



. PUSH 때 top 증가
. Stackfull check 는 top1=top2 를 check 하면 된다

ex) n 개의 stack



. $Stack_k$ is empty?: top[k] = bottom[k]
. $Stack_k$ is Full?: top[k] = bottom[k+1]

- Add

```
. if (top[k] = bottom[k+1]) then
                stack-full(k);
  else    stack[top[k]] = item;
```

- Delete

```
. if top[k] = bottom[k]
          return stack-empty(k);
  return    stack[top[k]-1];
```

4. QUEUE

4.1 QUEUE 의 정의

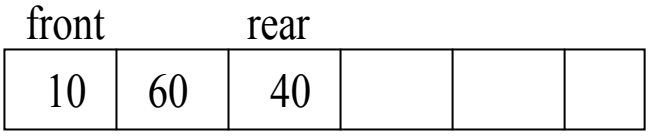
1) 데이터의 삽입과 삭제는 한쪽 끝(rear)과 다른 한쪽 끝(front)에서 발생한다. (임의의 곳에 자료를 삽입/삭제 불가능)

- Stack: 1 pointer, Queue: 2 pointer

$Q = (a_1, a_2, \ldots\ldots a_n)$

Front rear

(먼저 add 된 노드) (나중 add된 노드)



2) 자료형은 배열처럼 동일 해야 한다.

3) FIFO(First-In-First-Out) 리스트라고 한다.(제일 먼저 입력된 것이 제일먼저 제거됨)

4) 큐는 배열과 링크드 리스트로 구현된다

5) 배열로 구현시는 정적큐(static queue) 라고 하고, 링크드 리스트로 구현시에는 동적 큐(dynamic queue) 라고 한다.

[큐 추상 데이타 타입]

---

structure *Queue*

objects: 0개 이상의 원소를 가진 유한 순서 리스트

functions:

모든 queue∈Queue, item∈element,
max_queue_size∈positive integer

Queue CreateQ(max_queue_size) ::=
최대 크기가 max_queue_size인 공백 큐를 생성

Boolean IsFullQ(queue, max_queue_size) ::=
if (queue의 원소수 == max_queue_size) return TRUE
else return FALSE

Queue AddQ(queue, item) ::=
if (IsFull(queue)) queue_full
else queue의 뒤에 item을 삽입하고 이 queue를 반환

Boolean IsEmptyQ(queue) ::=
if (queue == CreateQ(max_queue_size)) return TRUE
else return FALSE

Element DeleteQ(queue) ::=
if (IsEmpty(queue)) return
else queue의 앞에 있는 item을 제거해서 반환

수학 5개

Queue CreateQ(max_queue_size) ::=

```
#define MAX_QUEUE_SIZE 100 /* 큐의 최대크기 */
typedef struct {
                    int key;
                    /* 다른 필드 */
                    } element;
 element queue[MAX_QUEUE_SIZE];
 int rear = -1;      /* front 와 rear를 –1로 지정함으로서, 큐를
 int front = -1;     /*  초기화 한다.
```

Boolean IsEmptyQ(queue) ::= front == rear

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

```
void addq(int *rear, element item)
{
    /* queue에 item을 삽입 */
    if (*rear == MAX_QUEUE_SIZE-1)
        queue_full();
        return;
    }
    queue[++*rear] = item;
}
```

```
element deleteq(int *front, int rear)
{
/* queue의 앞에서 원소를 삭제 */
    if (*front == rear)
        return queue_empty(); /* 에러 key를 반환 */
    return queue[++*front];
}
```

```
void create_queue()
 {   front = -1;
      rear = -1;
}

Int queue_full()
 {    if (top==queue_size –1)   return 1;
      Else   return 0;
 }

int queue_empty()
{ if (front ==rear) return 1;
    else   return 0;
}

void print_queue()
{
        int qp;

        if (queue_empty())
              cout << "Queue is Empty!\n";
        else {
           i = front + 1;
           cout << "-- Print Queue --\n";
           while (i <= rear) {
                    cout <<   queue[i];
                    i = i + 1;
           }
        }
}
```

예제 [작업 스케쥴링] : 운영체제에 의한 작업 큐(job queue)의 생성

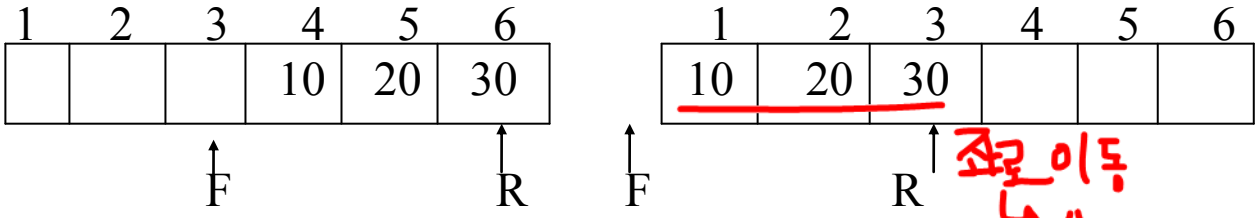
front	rear	Q[0] Q[1] Q[2] Q[3]	설 명
-1	-1		공백큐
-1	0	J1	Job 1의 삽입
-1	1	J1 J2	Job 2의 삽입
-1	2	J1 J2 J3	Job 3의 삽입
0	2	J2 J3	Job 1의 삭제
1	2	J3	Job 2의 삭제

* Problems with Queue

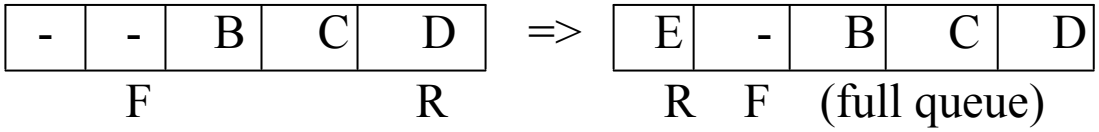
위의 예에서 보듯이 작업이 큐에 들어오고 나감에 따라 큐는 전체적으로 오른쪽으로 **shift** 된다. 즉, **rear index** 가 큐의 **maxsize** 와 동일하게 되어 큐는 **full** 이 된다.

■ 해결책:

1) **Front = 0** 이 되도록, 전체 **Q** 를 왼쪽으로 이동
(**Q** 에 많은 원소 있을 때는 상당한 처리시간 필요)



2) 환상 Q (Circular Queue) 이용



5. Circular Queue

. 배열로 선언: Q[0. . n-1]

. If R=n-1 이면, 다음은 Q[0]에 삽입

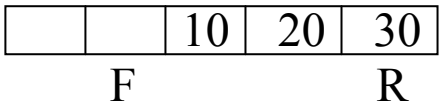
=> solution 1

```
void enqueue (int item)
{
  rear = (rear+1) % QUEUESIZE;
  if (front == rear)
        cout << "Queue is full";
   else
        queue[rear] = item;
}
```

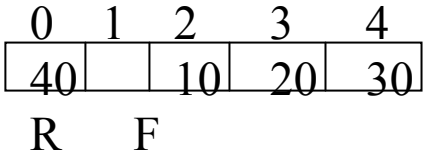
```
void dequeue ()
{
   int item;
  if (front==rear) {
        cout<< “Queue is empty”;
        exit(-1);
   }
  else {
       front=(front+1) %queuesize;
       return queue[front];
  }
```

* 문제점: 시작할때 rear 은 1 부터 시작하기 때문에 Queue[0]는 비어있고, Queue[4] 이후,.. rear=front 되면 Queuefull 이 되기 때문에 Memory 의 one space 가 항상 비어있게 된다.

ex) 0 1 2 3 4 F=1, R = 4, n = 5

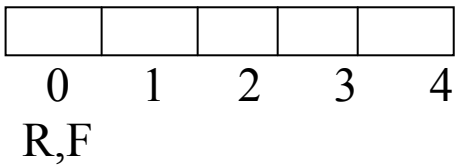


⇨ 40 insert: R=(r+1) mod n = 5 mod 5 = 0



⇨ 50 insert:
R=(0+1) mod n = 1 => F=R, Queuefull 발생, Enqueue 불가능

⇨ Delete



Therefore, F=R, Q empty

■ Alternative Method#1 ➔

```
void enqueue(int item)
   {
      if (front == rear)
         cout << "Queue is full";
      else {
         Queue[rear] = item;
         rear=(rear+1) % QueueSize;
       }
}
```

```
int dequeue()
{
  int item;
  if (front == rear) {
      cout << "Queue is empty";
        exit(-1);
       }
  else {
     item = Queue[front];
     front= (front + 1) % QueueSize;
     return item; }      }
```

q: 위 코드의 문제점?

=> queue 가 full 인지 empty 인지 test 하는 조건이 같다.
즉, full 인 상태에서 dequeue 하면 Queue is empty message 발생되고 empty 상태에서 enqueue 하면 Queue is full message 발생

- Alternative method #2 (flag 사용)

* Enqueue 할때마다 flag =1, dequeue 할때마다 flag = 0 으로 setting

```
void enqueue(int item)
{
   if ((front == rear) && (flag == 1))
      cout << "Queue is full";
   else {
      Queue[rear] = item;
      rear = (rear + 1) % QueueSize;
      flag = 1
   }
}
```

```
int dequeue()
{
 int item;
if ((front == rear) && (flag == 0)){
    cout << "Queue is empty";
     exit(1);
 }
  else {
     item = Queue[front];
     front = (front + 1) QueueSize;
       flag = 0;
    return item; }
 }
```

- alternative method #3 (count 사용)
  - 즉, enqueue 때마다 +count, Dequeue 때마다 –count
    If count ==0 then empty, if count==Queuesize then Full.