수식의 표기법

- 프로그램을 작성할 때 수식에서 +, -, *, /와 같은 이항연산자는 2개의 피연산자들 사이에 위치
- 이러한 방식의 수식 표현이 중위표기법(Infix Notation)
- 컴파일러는 중위표기법 수식을 후위표기법(Postfix Notation)으로 바꾼다.
  - 그 이유는 후위표기법 수식은 괄호 없이 중위표기법 수식을 표현할 수 있기 때문
- 전위표기법(Prefix Notation): 연산자를 피연산자들 앞에 두는 표기법

수식의 표기법

- **중위표기법 수식과 대응되는 후위표기법, 전위표기법 수식**
  - 모든 경우에서 피연산자의 순서는 동일!!!

중위표기법	후위표기법	전위표기법
A + B	A B +	+ A B
A + B – C	A B + C –	+ A – B C
A + B * C – D	A B C * + D –	– + A * B C D
(A + B) / (C – D)	A B+ C D – /	/ + A B – C D
A / B – C + D * E – A * C	A B / C – D E * + A C * -	- + - / A B C * D E * A C

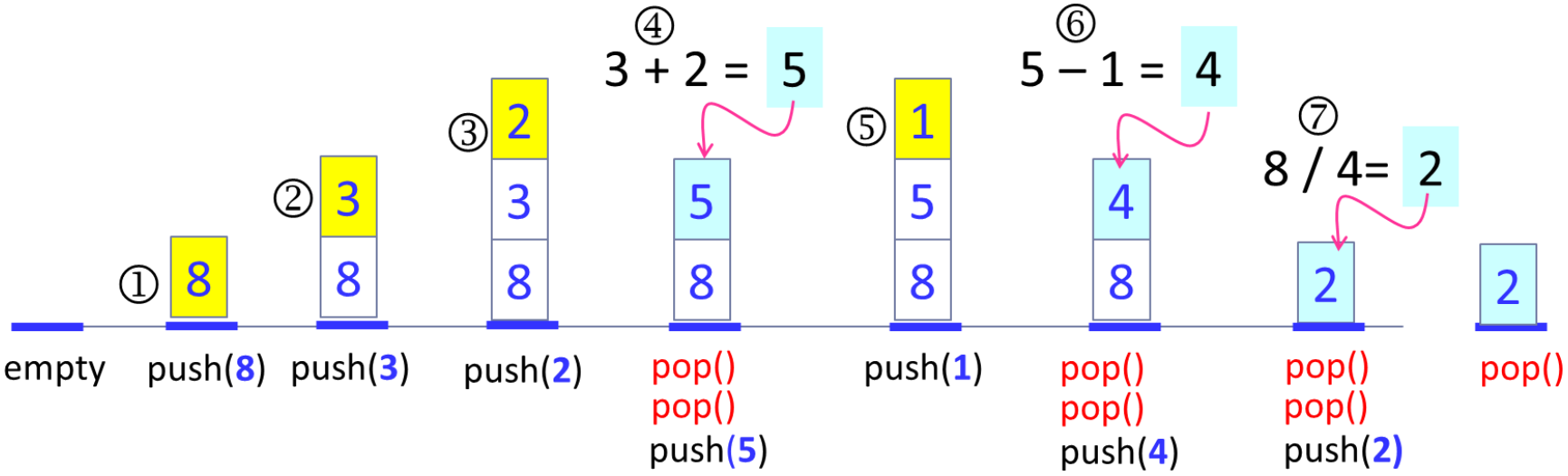
후위표기법 수식 계산

**[핵심 아이디어] 피연산자는 스택에 push하고,
연산자는 2회 pop하여 계산한 후 push**

- **후위표기법으로 표현된 수식 계산 알고리즘**
  - 입력을 좌에서 우로 문자를 한 개씩 읽는다. 읽은 문자를 C라고하면
  - [1] C가 피연산자이면 스택에 push
  - [2] C가 연산자(op)이면 pop을 2회 수행한다. 먼저 pop된 피연산자가 A이고, 나중에 pop된 피연산자가 B라면, (A op B)를 수행하여 그 결과 값을 push

[예 제]

①	②	③	④	⑤	⑥	⑦
8	3	2	+	1	−	/



중위표기법 수식을 후위표기법으로 변환

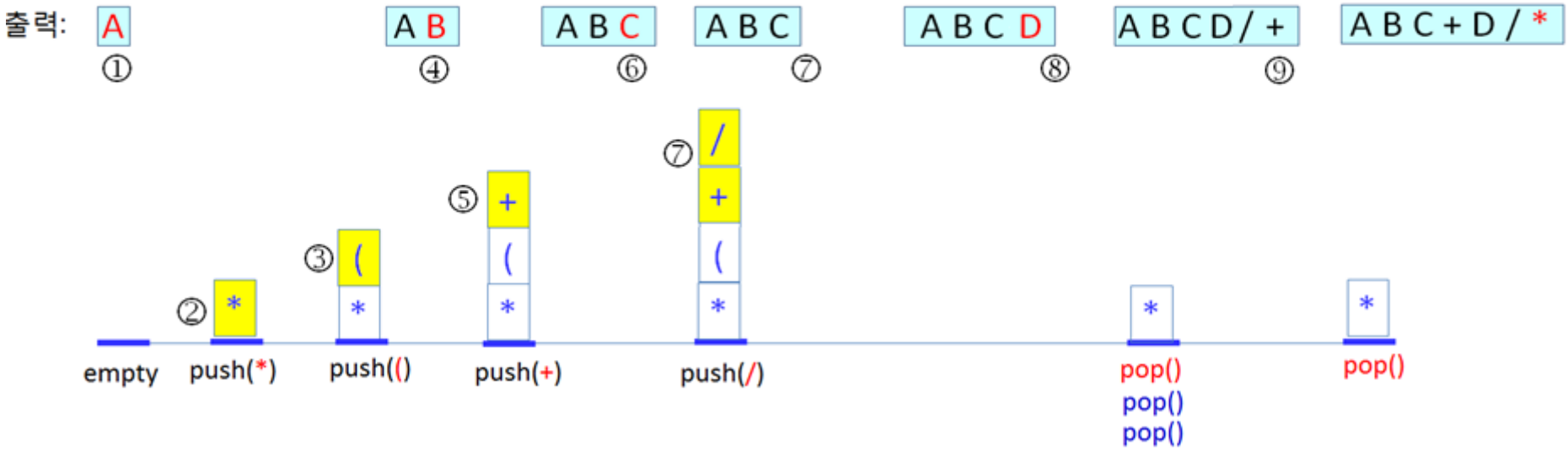
[핵심 아이디어] 왼쪽 괄호나 연산자는 스택에 push하고, 피연산자는 출력

- **입력을 좌에서 우로 문자를 1개씩 읽는다. 읽은 문자가**
  - 피연산자이면, 읽은 문자를 출력
  - 왼쪽 괄호이면, push
  - 오른쪽 괄호이면, 왼쪽 괄호가 나올 때까지 pop하여 출력. 단, 오른쪽이나 왼쪽 괄호는 출력하지 않음
  - 연산자이면, 자신의 우선순위보다 낮은 연산자가 스택 top에 올 때까지 pop하여 출력하고 읽은 연산자를 push
- **입력을 모두 읽었으면 스택이 empty될 때까지 pop하여 출력**

[예 제]

①	②	③	④	⑤	⑥	⑦	⑧	⑨
A	*	(	B	+	C	/	D	)

출력:



중위표기법 수식을 후위표기법으로 변환

- 예) A+B*C로부터 ABC*+를 생성

다음 토큰	스택	출력
없음	공백	없음
A	공백	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
완료	공백	ABC*+

- 예) A*(B+C)*D로부터 ABC+*D*를 생성

다음 토큰	스택	출력
없음	공백	없음
A	공백	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
완료	공백	ABC+*D*

닫는 괄호를 만나면
여는 괄호까지를 스택에서 pop하여 출력

동일한 우선순위인 *을 만났는데
*는 왼쪽 결합을 하므로,
스택에 있는 *을 빼서 출력하고, 입력받은 *을 스택에 저장

중위표기법 수식을 후위표기법으로 변환

- **우선순위를 기반으로 스택킹(stacking)과 언스택킹(unstacking)을 수행**
  - '(' 괄호 때문에 복잡해짐
    - 스택 속에 있을 때는 낮은 우선순위의 연산자처럼 동작
    - 그 외의 경우 높은 우선순위의 연산자처럼 동작
  - isp(in-stack precedence)와 icp(incoming precedence)으로 해결
    - 연산자는 자신의 isp가 새로운 연산자의 icp보다 산술적으로 작거나 같을 때 스택 밖으로 나온다.

연산자	ISP(In Stack Priority)	ICP(In Coming Priority)
(	8	0
단항-, !	1	1
*, /, %	2	2
+. –	3	3
<, ≤, ≥, >	4	4
==, !=	5	5
&&	6	6
||	7	7
#(eos)		8

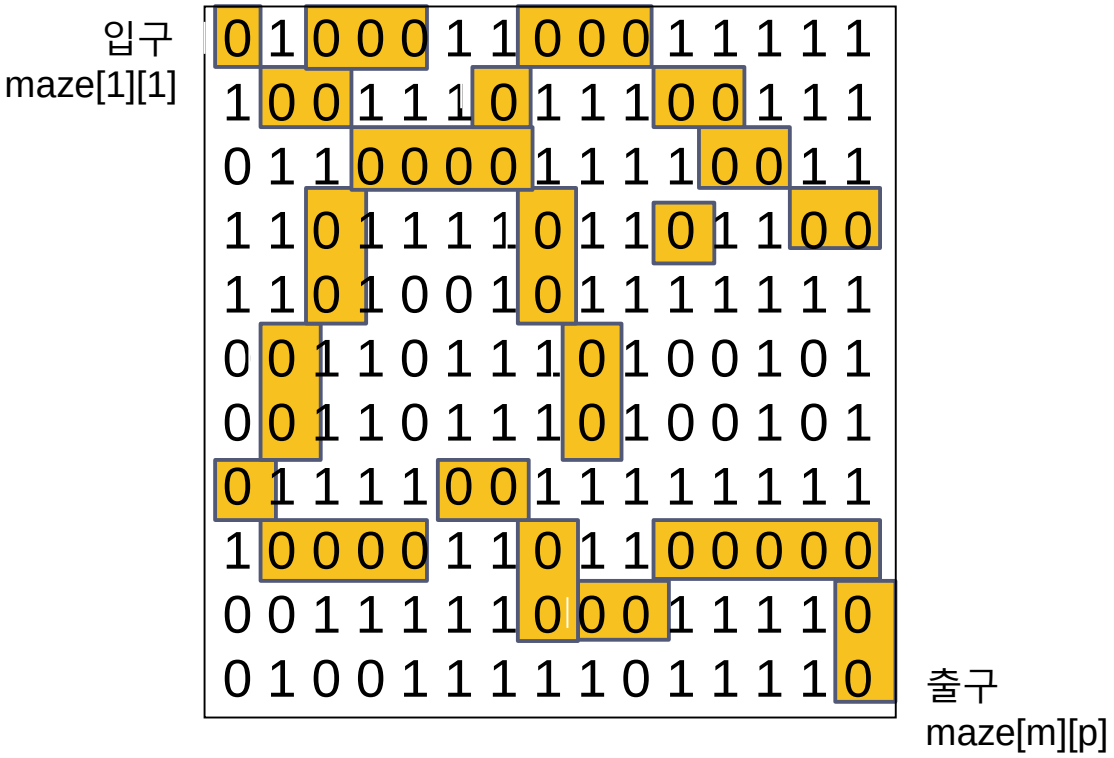
```
void getPostfix (Expression e)
{ // e로 전달되는 infix 표현에서 postfix 표현을 얻어내는 함수.
  // NextToken은 다음 token 읽어들이는 함수. 마지막 token은 “#”으로 설정
  Stack<Token> stack; // stack 초기화
  stack.Push('#');          // “#”은 stack bottom 으로도 사용
  for (Token x = NextToken(e); x != '#'; x = NextToken(e))  // ‘#’이 들어올 때까지 처리
     if (x is an operand)   // 피연산자(값)는 그대로 출력
          cout << x;
     else if (x == ')')     // 닫는 괄호를 만나면, 여는 괄호를 얻을 때까지 pop
     {
          for (; stack.Top() != '('; stack.Pop())
                cout << stack.Top();
          stack.Pop(); // unstack '('
     }
     else {                    // 연산자를 만나면
          for (; isp(stack.Top()) <= icp(x); stack.Pop())  // 스택에 들어있는 우선순위가
                cout << stack.Top();                       // 높은 연산자를 꺼내서 출력
          stack.Push(x);
    }

  // 입력을 모두 읽어들였으면, 스택에 남아있는 연산자를 모두 출력
  for (; !stack.IsEmpty(); cout << stack.Top(), stack.Pop());
  cout << endl;
}
```

스택 자료구조의 응용

- 미로 찾기
- 트리의 방문(4장)
- 그래프의 깊이우선탐색(9장)
- 프로그래밍에서 매우 중요한 함수(메소드) 호출 및 재귀호출도 스택 자료구조를 바탕으로 구현

미로 문제(1)

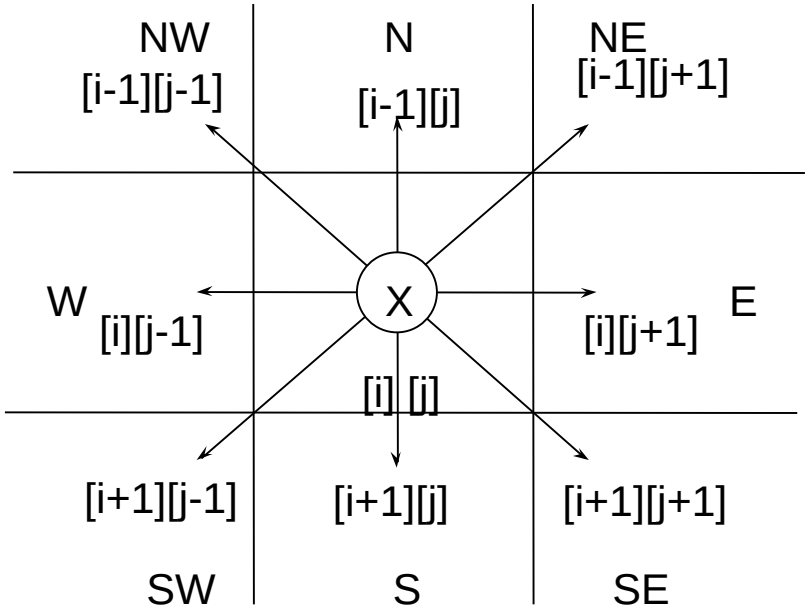
- **미로(maze)는 1≤i≤m이고 1≤j≤p인 이차원 배열 maze[m][p]로 표현**
  - 1 : 통로가 막혀 있음 / 0 : 통과할 수 있음



예제 미로(경로를 찾을 수 있는가?)

미로 문제(2)

- **현재의 위치 x : maze[i][j]**



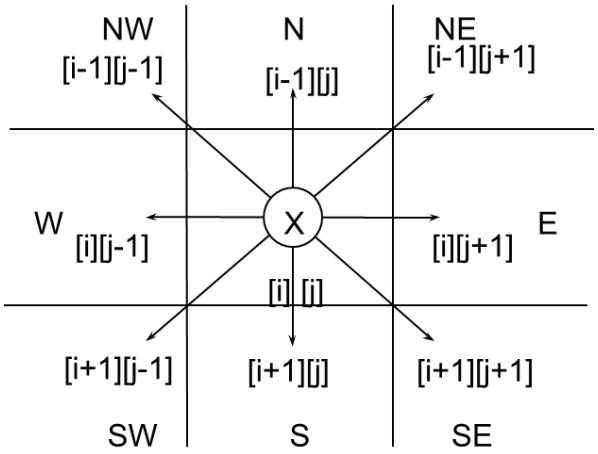
가능한 이동

- 가능한 8개의 이동방향을 탐색하여 이동할 수 있는 방향으로 시도

- i=1이거나 m 또는 j=1이거나 p인 경계선에 있는 경우
  가능한 방향은 8방향이 아니라 3방향만 존재
  - 경계 조건을 매번 검사하는 것을 피하기 위해 미로의 주위를 1로 둘러쌈
  - 배열은 maza[m+2][p+2]로 선언되어야 함

미로 문제(3)

- **이동할 수 있는 방향들을 배열 move에 미리 정의하는 방법**



q	move[q].x	move[q].y
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

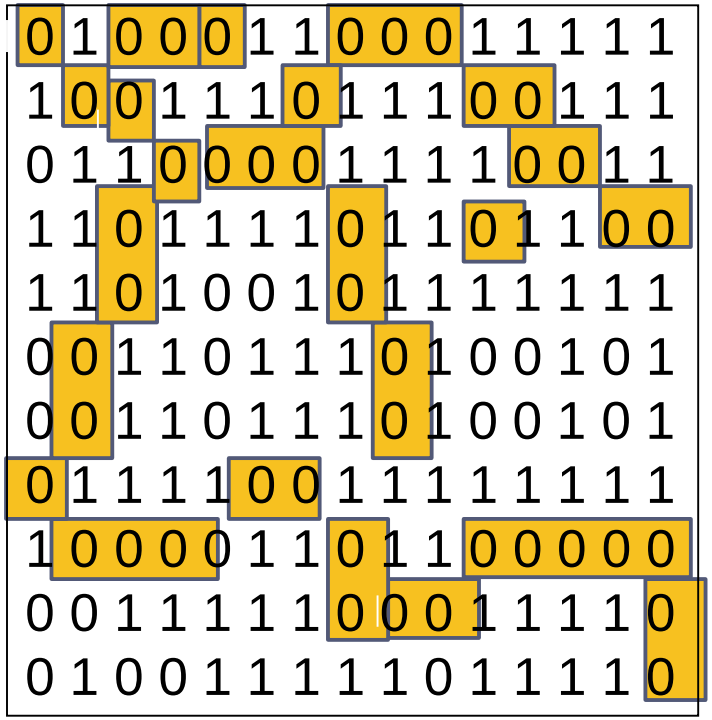
이동 테이블

[I][j]에서 SW, 즉 [g][h]로 이동
→g = i + move[SW].x;
h = j + move[SW].y;

- 미로 이동 시, 현재의 위치와 직전 이동 방향을 저장한 후 한 방향을 선택한다.

미로 문제(1)

입구
maze[1][1]



출구
maze[m][p]

예제 미로(경로를 찾을 수 있는가?)

	2	2	N → NE → E
step1	1	1	N → NE → E → SE → S

	2	3	N → NE → E → SE
	1	4	W
	1	3	SE
	2	2	E
step5	1	1	S

	1	4	SE → S → SW → W
	1	3	SE
	2	2	E
step4	1	1	S

	1	5	N → ..→W → NW
	1	4	N → NE → E → SE
	1	3	SE
	2	2	E
step3	1	1	S

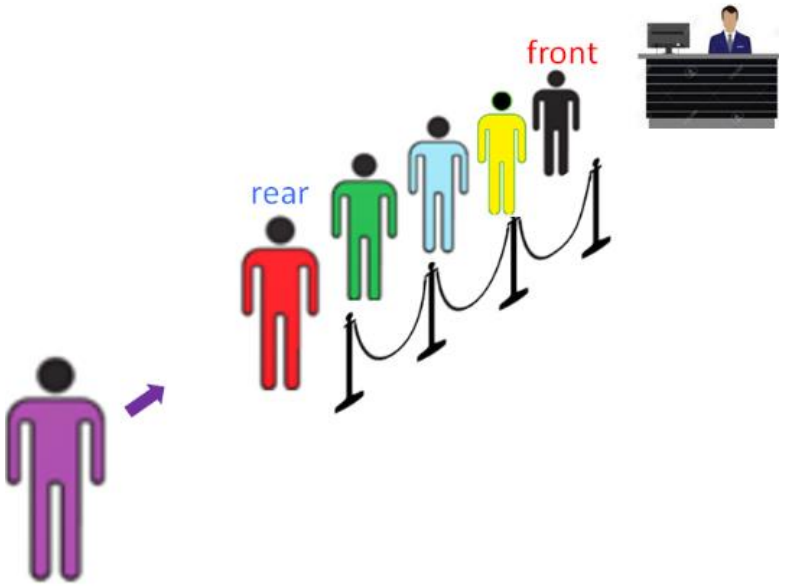
	1	3	N → NE → E → SE
	2	2	E
step2	1	1	S

수행시간

- 배열로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 소요
- 배열 크기를 확대 또는 축소시키는 경우에 스택의 모든 item들을 새 배열로 복사해야 하므로 O(N) 시간이 소요
  - 상각분석: 각 연산의 평균 수행시간은 O(1) 시간
- 단순연결리스트로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 걸리는데, 연결리스트의 앞 부분에서 노드를 삽입하거나 삭제하기 때문
- 배열과 단순연결리스트로 구현된 스택의 장단점은 2장의 리스트를 배열과 단순연결리스트로 구현하였을 때의 장단점과 동일

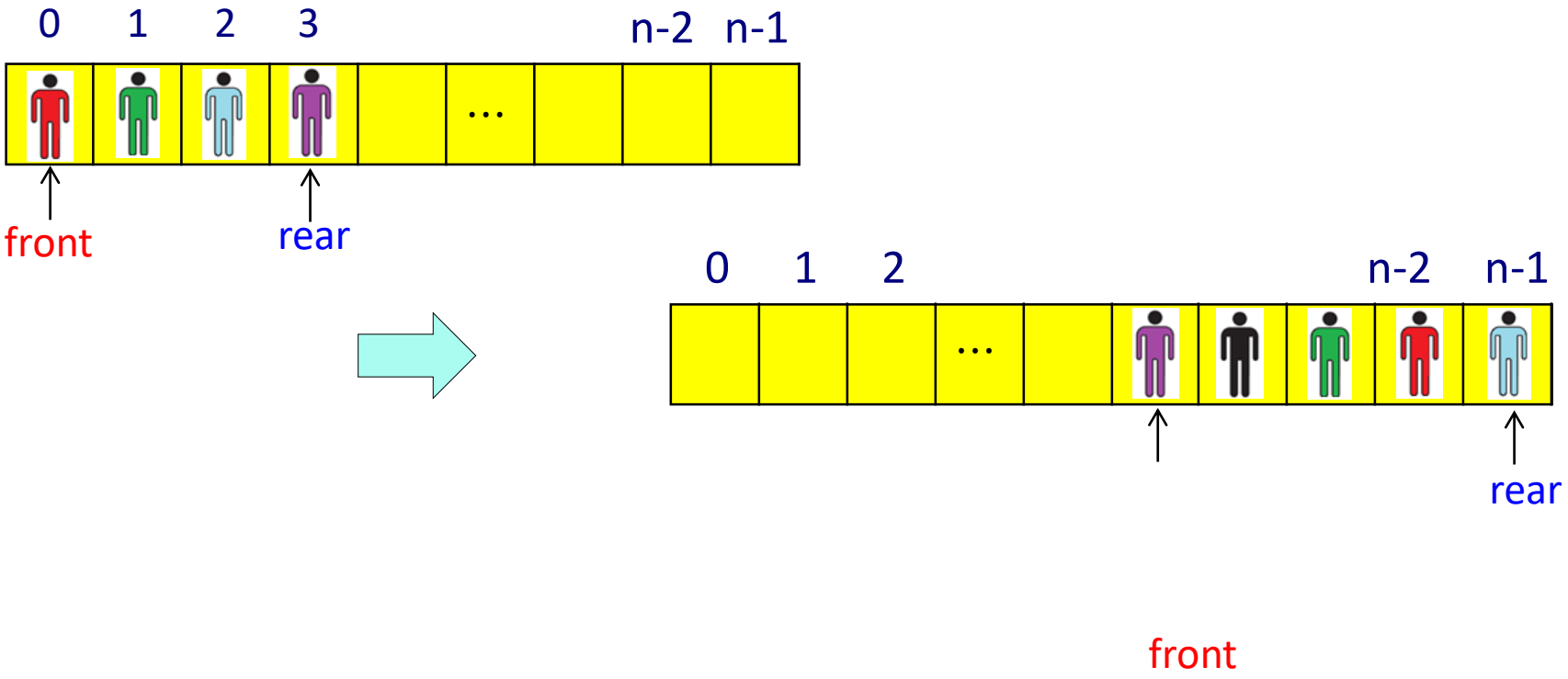
3.2 큐

- 큐(Queue): 삽입과 삭제가 양 끝에서 각각 수행되는 자료구조
- 일상생활의 관공서, 은행, 우체국, 병원 등에서 번호표를 이용한 줄서기가 대표적인 큐
- 선입 선출(First-In First-Out, FIFO) 원칙하에 item의 삽입과 삭제 수행



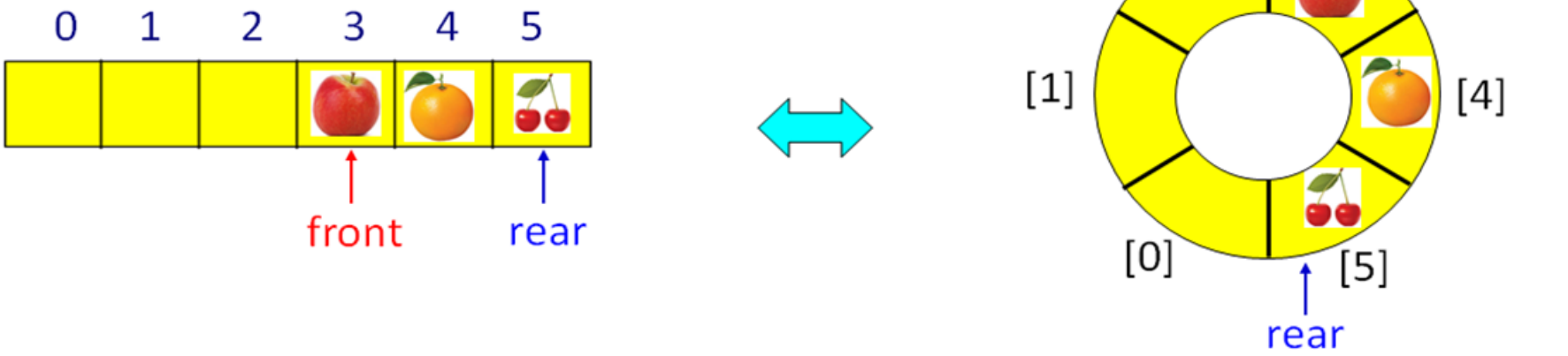
배열로 큐를 구현하는 경우의 문제점

- **큐에서 삽입과 삭제를 거듭하게 되면 그림과 같이 큐의 item들이 배열의 오른쪽 부분으로 편중되는 문제가 발생**
  - 왜냐하면 새 item들은 뒤에 삽입되고 삭제는 앞에서 일어나기 때문



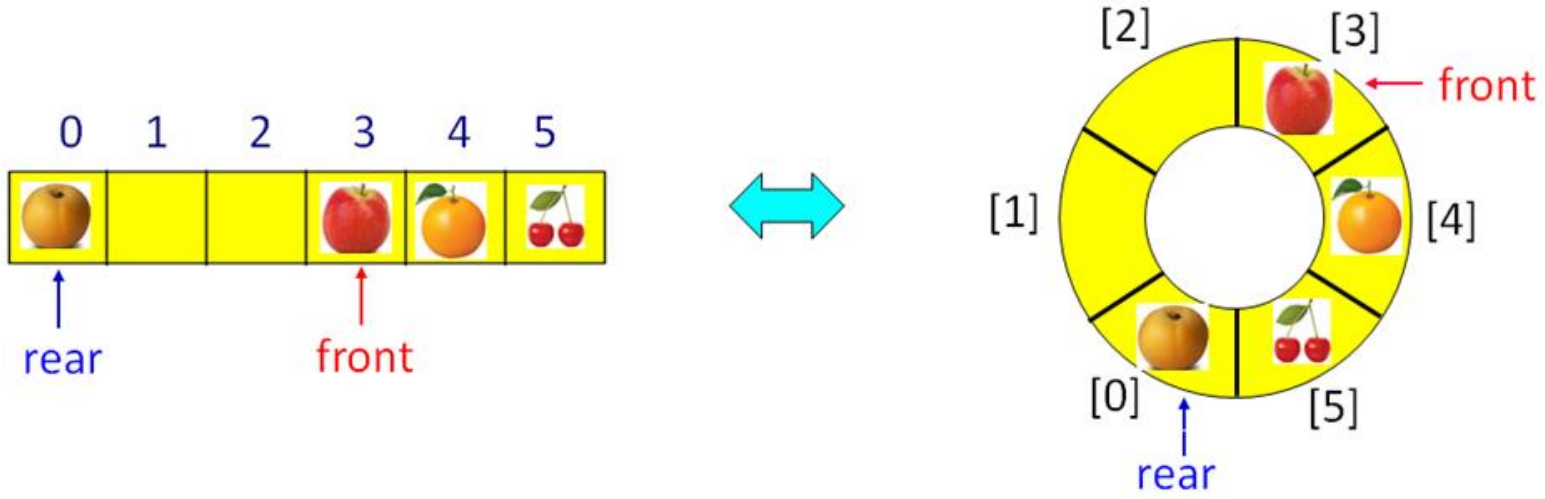
항목 이동 해결 방안

- **[방법 1] 큐의 item들을 배열의 앞부분으로 이동.**
  - 수행시간이 큐에 들어있는 item 의 수에 비례하는 단점
- **[방법 2] 배열을 원형으로, 즉, 배열의 마지막 원소가 첫 원소와 맞닿아 있다고 여김**



원형 큐의 구현

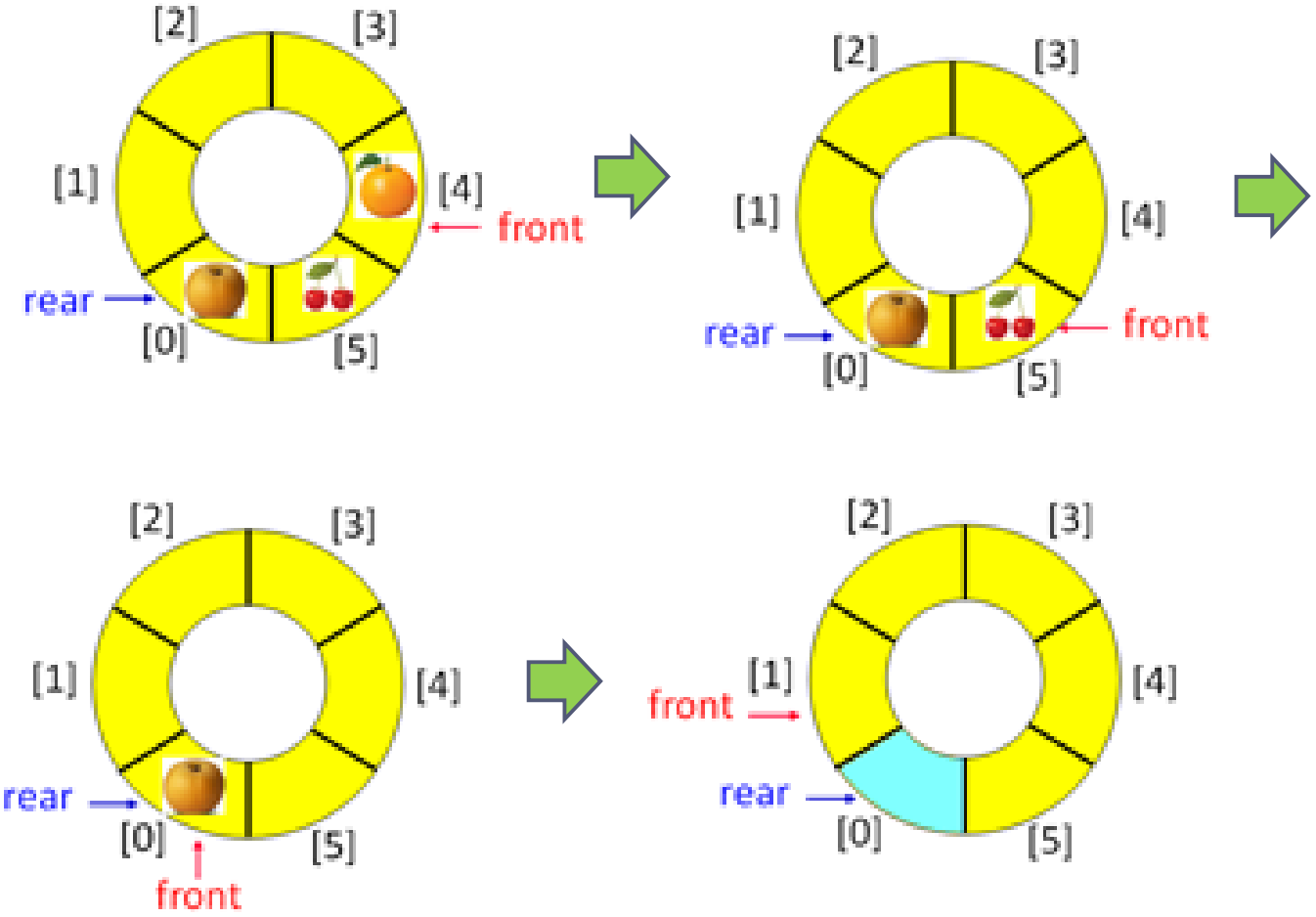
- 새 item 삽입 후



- 배열의 앞뒤가 맞닿아 있다고 생각하기 위해 배열의 rear 다음의 비어있는 원소의 인덱스
- rear = (rear + 1) % N
- 여기서 N은 배열의 크기이다.

원형 큐의 구현

- 연속된 삭제 연산 수행

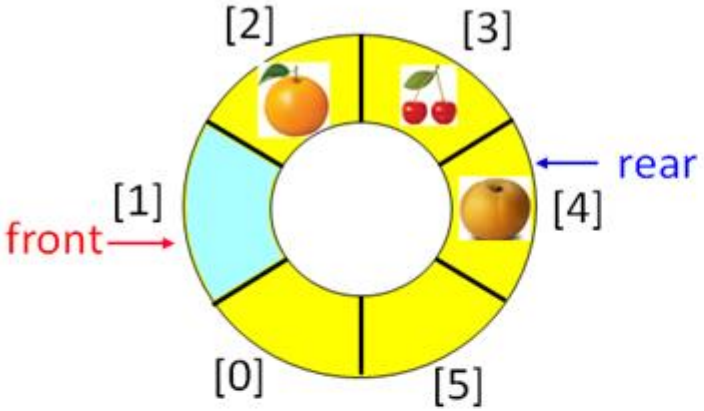


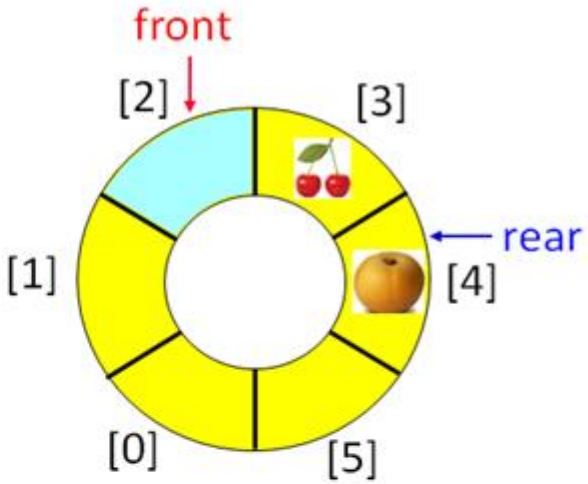
큐의 마지막 item을 삭제한 후에 큐가empty 임에도 불구하고 rear는 삭제된 item을 아직도 가리키고 있다.

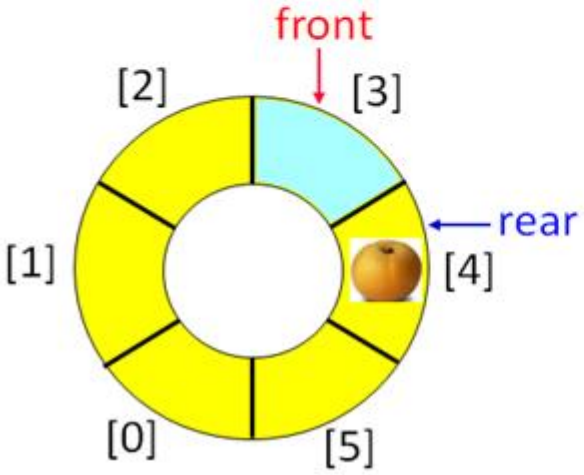
큐가 empty일 때 문제 해결 방안

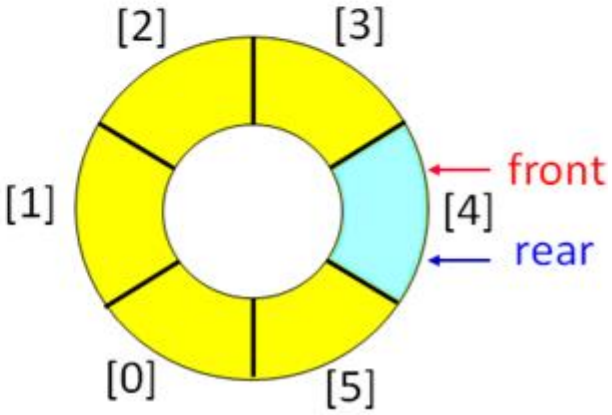
- **[방법 1] item을 삭제할 때마다 큐가 empty가 되는지 검사하고, 만일 empty가 되면, front = rear = 0을 만든다.**
  - 삭제할 때마다 empty 조건을 검사하는 것은 프로그램 수행의 효율성이 저하
- **[방법 2] front를 실제의 가장 앞에 있는 item의 바로 앞의 비어있는 원소를 가리키게 한다.**
  - 배열의 크기가 N이라면 실제로 N-1개의 공간만 item들을 저장하는데 사용
  - [방법 1]에서 item을 삭제할 때마다 조건을 한번 더 검사하는 것은 '이론적인' 수행시간을 증가시키지 않으나 일반적으로 프로그램이 수행될 때 조건을 검사하는 프로그램의 실제 실행시간은 검사하지 않는 프로그램보다 더 오래 소요됨

연속된 삭제 연산 수행









Empty가 되면 front == rear가 된다.