# 제 3 장. 스택과 큐

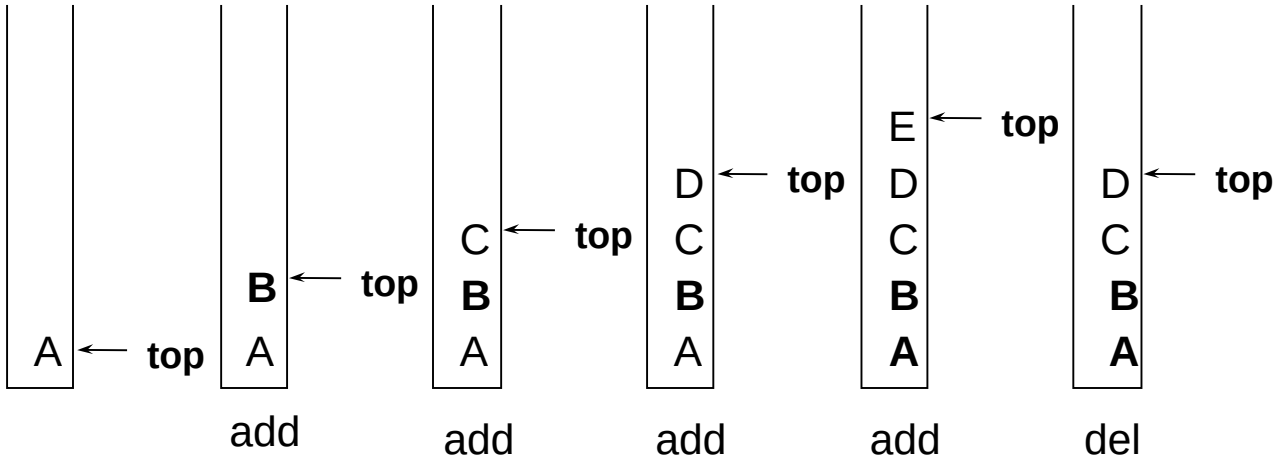
- 순서 리스트의 특별한 경우

  순서 리스트: $A=a_0, a_1, \cdots, a_{n-1}, n \geq 0$

- 스택(stack)
  - 톱(top)이라고 하는 한쪽 끝에서 모든 삽입(push)과 삭제(pop)가 일어나는 순서 리스트
    - 스택 $S=(a_0,\ldots..,a_{n-1})$:
    - $a_0$는 bottom, $a_{n-1}$은 top의 원소
    - $a_i$는 원소 $a_{i-1}(0<i<n)$의 위에 있음
  - 후입선출(LIFO, Last-In-First-Out) 리스트

# 스택(Stack)

- 원소 A,B,C,D,E를 삽입하고 한 원소를 삭제하는 예



[ 스택 추상 데이타 타입 ]

structure *Stack*

objects: 0개 이상의 원소를 가진 유한 순서 리스트

functions:

모든 *stack*∈ *Stack, item*∈*element, max_stack_size*∈*positive integer*

*Stack* CreateS(max_stack_size) ::=
최대 크기가 max_stack_size인 공백 스택을 생성

*Boolean* IsFull(stack, max_stack_size) ::=
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)) stack_empty
else 스택 톱의 item을 제거해서 반환

- 스택 추상 데이터 타입의 구현
  - 일차원 배열 stack[MAX_STACK_SIZE] 사용
  - i번째 원소는 stack[i-1]에 저장
  - top은 스택의 최상위 원소를 가리킴 (초기: top = -1)

*Stack* CreateS(max_stack_size) ::=

```
#define MAX_STACK_SIZE 100 /*최대스택크기*/
typedef struct {
            int key;
            /* 다른 필드 */
            } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

*Boolean* IsEmpty(Stack) ::= top < 0;

*Boolean* IsFull(Stack) ::= top >= MAX_STACK_SIZE-1;

- 스택에 대한 접근은 최상위 원소에 대한 포인터를 통해

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

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

# 큐 (queue)

- 한쪽 끝(rear)에서 삽입이 일어나고 그 반대쪽 끝(front)에서 삭제가 일어나는 순서 리스트
  - 큐 $Q=(a_0,a_1,\ldots,a_{n-1})$:
  - $a_0$는 앞(front) 원소
  - $a_{n-1}$은 뒤(rear) 원소
  - $a_i$는 $a_{i-1}$($1 \le i < n$)의 뒤에 있다고 함
- 선입선출(FIFO, First-In-First-Out) 리스트
- 그림 3.4 큐에서의 삽입과 삭제

[ 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

# 큐의 구현: 1차원 배열 이용

- 변수 front는 큐에서 첫 원소의 위치보다 하나 작은 위치
- 변수 rear는 큐에서 마지막 원소의 위치

```
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;
       int front = -1;
  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];
}
```

# 큐의 순차적 표현의 문제점 예

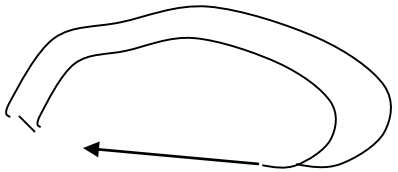
- 예제 3.2 [작업 스케쥴링]:
  - 운영체제의 작업 큐(job queue)
  - 작업들이 큐에 들어가고 나옴에 따라 큐는 점차 오른쪽으로 이동

→결국 rear 값이 MaxSize-1과 같아져서 큐가 포화상태가 됨

| front | rear | Q(1) [1] [2] [3] [4] [5] [6] ... | comments |
|---|---|---|---|
| -1 | -1 | queue empty | initial |
| -1 | 0 | J1 | job 1 joins Q |
| -1 | 1 | J1 J2 | job 2 joins Q |
| -1 | 2 | J1 J2 J3 | job 3 joins Q |
| 0 | 2 | J2 J3 | job 1 leaves Q |
| 0 | 3 | J2 J3 J4 | job 4 joins Q |
| 1 | 3 | J3 J4 | job 2 joins Q |

# 순차적 큐의 문제 해결: 원형 큐

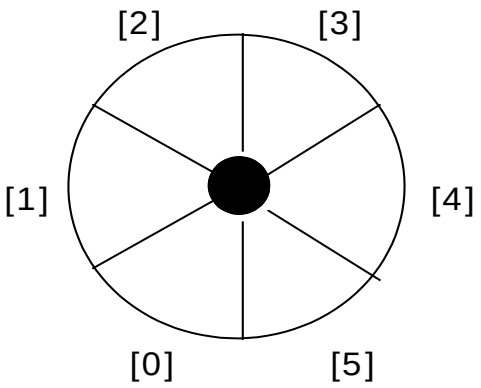
- 배열 queue[MaxSize]를 원형으로 취급
  →rear == MaxSize-1이면 q[0]가 빈 경우 q[0]에 삽입



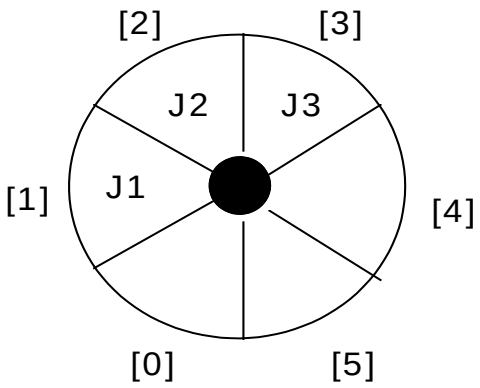
- MaxSize를 최대원소로 사용하면 원형큐는 포화와 공백 모두 front == rear가 됨
  - 이를 구분하기 위해 포화상태는 원소수가 MaxSize-1일 경우로 함
  - front == rear 는 공백상태로만 사용

- 초기: front == rear == 0

공백 큐

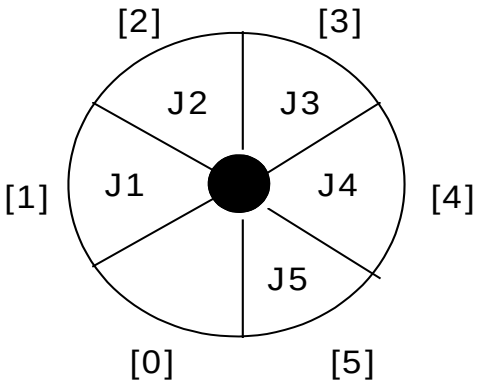



front = 0
rear = 0



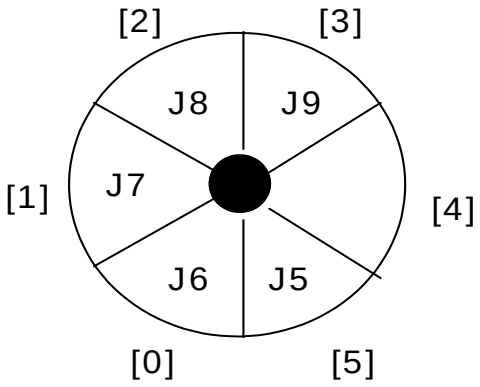

front = 0
rear = 3

포화 큐




front = 0
rear = 5

포화 큐




front = 4
rear = 3

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

---

```
void deleteq(int *front, int rear)
{
   element item;
   /* queue의 front 원소를 삭제하여 그것을 item에 놓음 */
   if (*front == rear)
      return queue_empty(); /* queue_empty는 에러 키를 반환 */
   *front = (*front+1) % MAX_QUEUE_SIZE;
   return queue[*front];
}
```

# 미로 문제

- 스택을 응용할 수 있는 예
- 미로는 1≤i≤m, 1≤j≤p인 이차원 배열 maze[m][p]로 표현
  - 1 : 막힌 통로, 0 : 열린 통로
  - maze[1][1]에서 출발
  - maze[m][p]에 출구

- 이동은 0이 있는 사각형만
  - 다음장 그림
  - 경계조건을 매번 검사하지 않고 미로주위를 1로 만듦, maze[m+2][p+2]를 사용, 첨자 i=0, m+1, j=0, p+1일 때는 모두 1

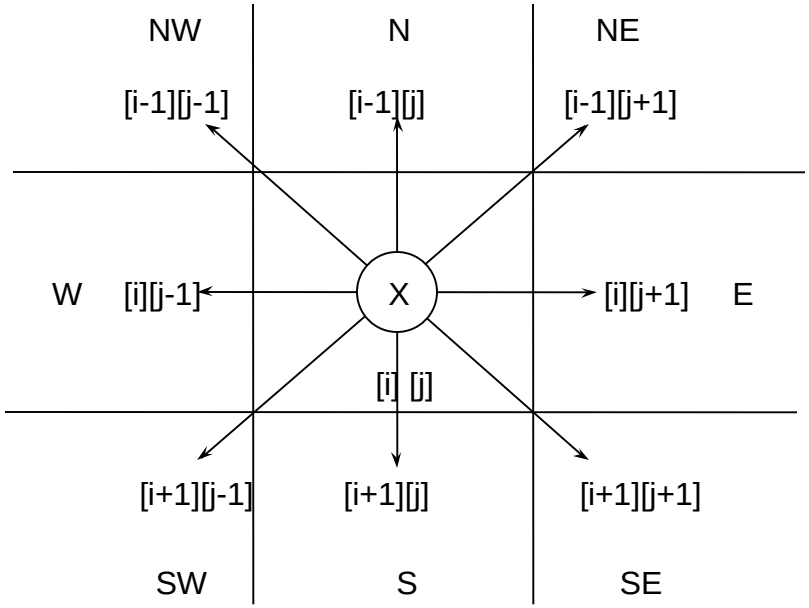
입구

0 1 0 0 0 1 1 0 0 0 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 0 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0

출구

# 미로에서 가능한 이동

- 현재의 위치 x : maze[i][j] 에서 가능한 이동




- 경계위치에서는 8방향이 아님
  - I = 1 or m, j=1 or m
  - 3방향, 5방향만 가능한 경우

- 이동할 수 있는 방향들을 정의

```
typedef struct  {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8];  /* 여덟 방향 이동에 대한 배열 */
```

| Name | Dir | move[dir].vert | move[dir].horiz |
|---|---|---|---|
| N | 0 | -1 | 0 |
| NE | 1 | -1 | 1 |
| E | 2 | 0 | 1 |
| SE | 3 | 1 | 1 |
| S | 4 | 1 | 0 |
| SW | 5 | 1 | -1 |
| W | 6 | 0 | -1 |
| NW | 7 | -1 | -1 |

- 다음 이동 위치 : *maze[next_row][next_col]*
  - *next_row = row + move[dir].vert;*
  - *next_col  = col + move[dir].horiz;*

- 경로의 탐색
  - 현위치 저장 후 방향 선택 : N에서 시계방향
  - 잘못된 경로의 선택 시는 backtracking후 다른 방향 시도
  - 시도한 경로의 재시도 방지

    mark[m+2][n+2] : 초기치 0

    mark[row][col] = 1 /* 한번 방문한 경우 */

  - 경로 유지위한 스택 사용

```
#define MAX_STACK_SIZE 100 /* 스택의 최대 크기 */
typedef struct {
        short int row;
        short int col;
        short int dir;
        } element;
element stack[MAX_STACK_SIZE];
```

  - MAX_STACK_SIZE 는 스택에 저장되는 최대 원소 수. 미로의 각 위치는 기껏해야 한번만 방문 되므로 mp

```
initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty) {
   /* 스택의 톱에 있는 위치로 이동*/
   <rol, col, dir> = delete from top of stack;
   while (there are more moves from current position)
      <next_row, next_col> = coordinate of next move;
      dir = direction of move;
      if ((next_row == EXIT_ROW) && (next_col == EXIT_COL)) success ;
      if(maze[next_row][next_col]==0)&&mark[next_row][next_col]==0) {
      /* 가능하지만 아직 이동해보지 않은 이동 방향 */
         mark[next_row][next_col] = 1;
         /* 현재의 위치와 방향을 저장 */
         add <row, col, dir> to the top of the stack;
         row = next_row;
         col = next_col;
         dir = north;
      }
   }
}
printf("No path found");
```

```
void path(void) //프로그램 3.8 미로 탐색 함수
{/* 미로를 통과하는 경로가 있으면 그 경로를 출력한다. */
   int i, row, col, next_row, next_col, dir, found=FALSE;
   element position;
   mark[1][1]=1; top=0; stack[0].row=1; stack[0].col=1; stack[0].dir=1;
   while (top>-1 && !found) {
      position = delete(&top);
      row = position.row;       col = position.col;
      dir = position.dir;
      while (dir<8 && !found) {
         next_row = row + move[dir].vert;  /* dir 방향으로 이동 */
         next_col = col + move[dir].horiz;
         if (next_row==EXIT_ROW && next_col==EXIT_COL)          found = TRUE;
         else if (!maze[next_row][next_col] && !mark[next_row][next_col]) {
            mark[next_row][next_col]) = 1;
            position.row = row; position.col = col; position.dir = ++dir;
            add(&top, position);
            row = next.row; col = next.col; dir = 0;
         }
         else ++dir;
      }
 }
```

```
if (found) {
      printf("The path is:");
      printf("row  col");
      for (i=0; i<=top; i++)
         printf("%2d%5d", stack[i].row, stack[i].col");
      printf("%2d%5d", row, col);
      printf("%2d%5d", EXIT_ROW, EXIT_COL);
   }
   else printf("The maze does not have a path");
}
```

- path 알고리즘의 분석
  - while문은 스택의 크기에 종속적이고 최대 mp
  - while 안의 while은 최대 8번 즉 O(1)
  - 따라서 O(mp)

# 수식의 계산

| 토큰 | 우선순위 | 결합성 |
|---|---|---|
| () [] -> | 17 | LR |
| -- ++ | 16 | L R |
| & * unary - | 15 | R L |
| * / % | 13 | L R |
| + - | 12 | L R |
| > >= < <= | 10 | L R |
| == != | 9 | L R |
| && | 5 | L R |
| \|\| | 4 | L R |

- 수식의 표현
  - 중위 표기(*infix notation*) : a * b / c
  - 후위 표기(*postfix notation*) : a b * c /

  예) x = a / b − c + d * e − a * c
  x = ((((a / b) − c) + (d * e)) − (a * c))

- 후위 표기 : x = a b / c − d e * a c * − +
  - 괄호 사용 않함
  - 왼쪽에서 오른쪽으로 계산
  - 연산자의 우선순위 없음

- 후위 표기식의 연산 : 6 2 / 3 - 4 2 * +

| 토큰 | 스택 | | | Top |
|---|---|---|---|---|
| | [0] | [1] | [2] | |
| 6 | 6 | | | 0 |
| 2 | 6 | 2 | | 1 |
| / | 6/2 | | | 0 |
| 3 | 6/2 | 3 | | 1 |
| - | 6/2-3 | | | 0 |
| 4 | 6/2-3 | 4 | | 1 |
| 2 | 6/2-3 | 4 | 2 | 2 |
| * | 6/2-3 | 4*2 | | 1 |
| + | 6/2-3+4*2 | | | 0 |

- 스택과 수식의 표현 방법

```
#define MAX_STACK_SIZE 100   /* 스택의 최대 크기 */
#define MAX_EXPR_SIZE 100    /* 수식의 최대 크기 */
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)
{ /* 전역변수로 되어 있는 후위 표기식 expr을 연산한다. '₩0'은 수식의 끝을
     나타낸다. stack과 top은 전역 변수이다. 함수 get_token은 토큰의 타입과
     문자 심벌을 반환한다. 피연산자는 한 문자로 된 숫자임을 가정 */
   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 */
      else {
         /* 두 피연산자를 삭제하여 연산을 수행후, 결과를 스택에 삽입 */
         op2 = delete(&top); /* 스택 삭제 */
         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 그
   것의 열거된 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 NULL : return eos;
      default : return operand;
   }
}
```

- 중위 표기에서 후위 표기로의 변환
  - 왼쪽에서 오른쪽으로 조사해 가면서 후위식을 생성
  - 피연산자는 바로 출력 수식에 전달
  - 연산자의 출력 순서는 우선순위에 의해 좌우, 정확한 위치를 알 때까지 연산자를 스택에 저장

- 예제 3.3 중위 표기식 a+b*c를 후위 표기식으로 변환

| 토큰 | 스택 [0] | [1] | [2] | Top | Output |
|---|---|---|---|---|---|
| a | | | | -1 | a |
| + | + | | | 0 | a |
| b | + | | | 0 | ab |
| * | + | * | | 1 | ab |
| c | + | * | | 1 | abc |
| eos | | | | -1 | abc*+ |

- 예제 3.4 중위 표기식 a*(b+c)*d를 후위 표기식으로 변환

| 토큰 | 스택 [0] | [1] | [2] | Top | Output |
|---|---|---|---|---|---|
| a | | | | -1 | a |
| * | * | | | 0 | a |
| ( | * | ( | | 1 | a |
| b | * | ( | | 1 | ab |
| + | * | ( | + | 2 | ab |
| c | * | ( | + | 2 | abc |
| ) | * | | | 0 | abc+ |
| * | * | | | 0 | abc+* |
| d | * | | | 0 | abc+*d |
| eos | | | | -1 | abc+*d* |

- 연산자의 스택킹과 언스택킹
  - 왼쪽 괄호는 스택 속에 있을 때는 낮은 우선 순위, 그 외의 경우에는 높은 우선 순위
  - 왼쪽 괄호는 스택에 쌓이고, 대응하는 오른쪽 괄호가 나올 때에만 스택에서 삭제
  - isp(in-stack precedence)와 icp(incoming precedence) 정의하여 스택속의 연산자 isp가 새로운 연산자의 icp보다 크거나 같을 때만 스택에서 삭제

```
precedence stack[MAX_STACK_SIZE];
  /* isp와 icp 배열 -- 인덱스는 연산자 lparen, rparen,
     plus, minus, times, divide, mod, eos의 우선순위 값 */
static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0} ;
static int icp[] = {20, 19, 12, 12, 13, 13, 13, 0} ;
```

예) isp[plus] = isp[2] = 12, 오른쪽 괄호 19, 왼쪽 괄호 0, 20, eos는 0

n : 수식의 토큰 수일 때, $\Theta(n)$

```
void postfix(void)
{ /* 수식을 후위 표기식으로 출력한다. 수식 문자열, 스택, top은 전역적이다.
     */
    char symbol;
    precedence token;
    int n = 0;
    int top = 0; /* eos를 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 {
//symbol의 isp가 token의 icp보다 크거나 같으면 symbol을 제거하고 출력
      while (isp[stack[top]] >= icp[token])
        print_token(delete(&top));
      add(&top, token);
    }
  } // end of for-loop
  while ((token=delete(&top)) != eos)
    print_token(token);
  printf("₩n");
}
```