# 제 3 장 스택과 큐

# 3.1 스택 추상 데이터 타입

□ 스택과 큐

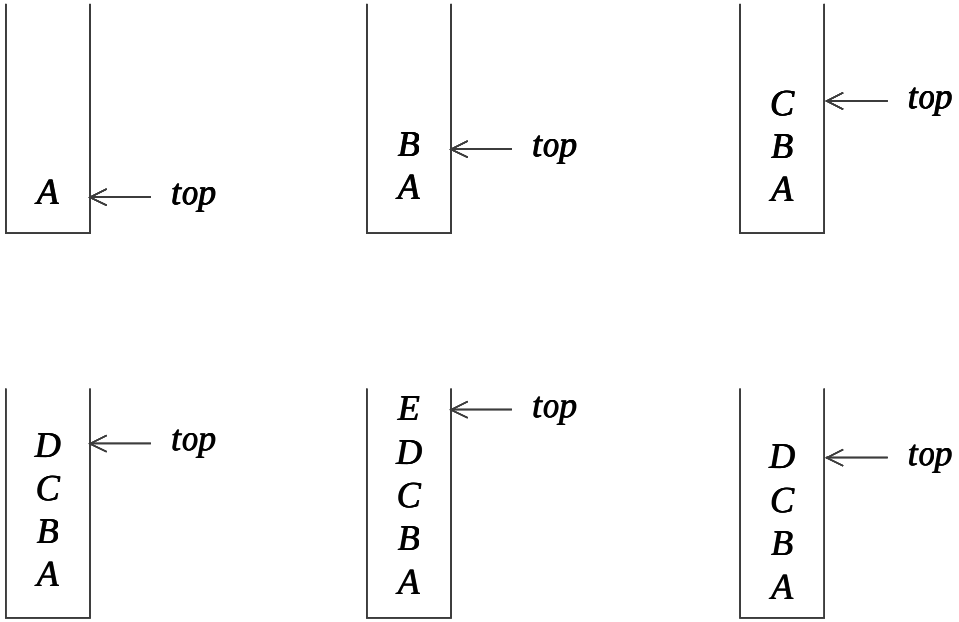
- 순서 리스트의 특별한 경우

□ 스택 (stack)

- 톱(top)이라는 한쪽 끝에서 모든 삽입과 삭제가 일어나는 순서 리스트
- 제일 나중에 들어온 원소가 제일 먼저 삭제된다

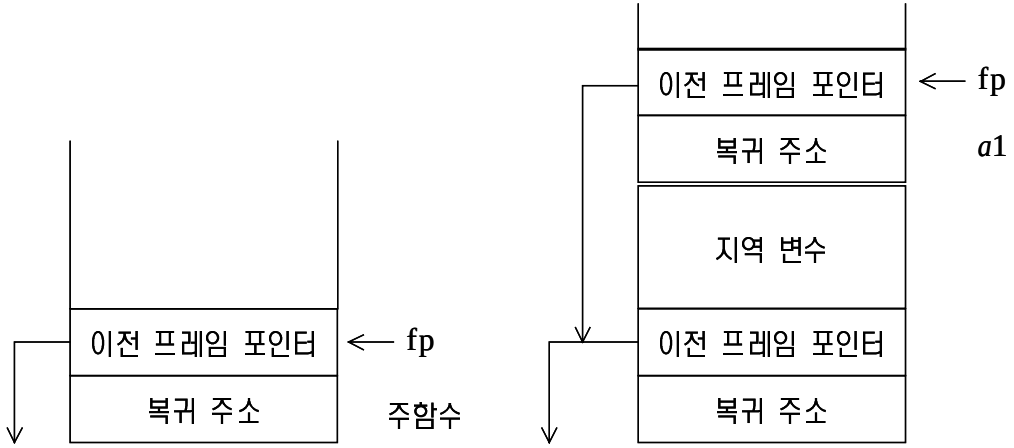
→ 후입 선출(Last-In-First-Out, LIFO) 리스트

▶ **페이지 103, 그림 3.1 : 스택에서의 삽입과 삭제**



✓ 시스템 스택 (system stack)

- 함수가 호출될 때마다 프로그램은 활성 레코드 또는 스택 프레임이라는 구조를 생성하고 이것을 시스템 스택의 top에 둔다
  - 함수의 호출에 대한 이해 요구 !!
- 페이지 104, 그림 3.2 : 함수 호출 뒤의 시스템 스택 참고



## ▶ 페이지 105, 구조 3.1 : 스택 추상 데이터 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_size*인 공백 스택을 생성
    *Boolean* IsFull(*stack, max_stack_size*)
      ::= if (*stack*의 원소수 == *max_stack_size*) return *TRUE*
        else return *FALSE*
    *Boolean* IsEmpty(*stack*)
      ::= if (*stack* == CreateS(*max_stack_size*)) return *TRUE*
        else return *FALSE*
    *Stack* Add(*stack, item*)
      ::= if (IsFull(*stack*)) *stack_full*
        else *stack*의 톱에 *item*을 삽입하고 return
    *Element* Delete(*stack*)
      ::= if (IsEmpty(*stack*)) *stack_empty*
        else 스택 톱의 *item*을 제거해서 반환

## □ 스택의 구현

- 일차원 배열 *stack*[MAX_STACK_SIZE]의 사용
  - 첫번째(최하위) 원소를 *stack*[0]에 저장
    두번째 원소를 *stack*[1]에 저장
    …
    *i*번째 원소를 *stack*[*i* – 1]에 저장
  - *top*은 스택의 최상위 원소를 가리킴
- 초기 : 공백 스택
  - *top* = -1

▶ **페이지 105 ~ 106, 스택 연산의 구현**

**페이지 106, 프로그램 3.1 : 스택에로의 삽입**

**& 페이지 106 ~ 107, 프로그램 3.2 : 스택으로부터 삭제**

```
Stack  CreateS(max_stack_size) ::=
        #define MAX_STACK_SIZE 100     /* 최대 스택 크기 */
        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;
```

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

IsFull(stack)

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

IsEmpty(stack)

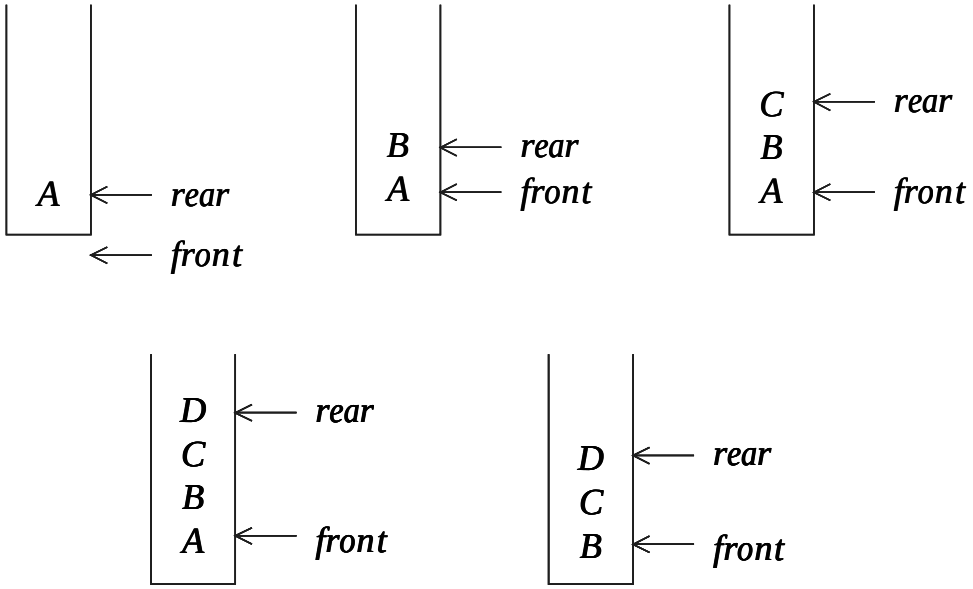
# 3.2 큐 추상 데이터 타입

□ 큐 (queue)

- 한쪽 끝(rear)에서 삽입이 일어나고 그 반대쪽 끝(front)에서 삭제가 일어나는 순서 리스트
- 제일 먼저 삽입된 원소가 제일 먼저 삭제된다

→ 선입선출(First-In-First-Out, FIFO) 리스트

✧ **페이지 108, 그림 3.4 : 큐에서의 삽입과 삭제 참고**



✓ 작업 스케줄링 (job scheduling)

- 운영체제에 의한 작업 큐(job queue)의 생성
- 시스템에 들어간 순서대로 처리

✧ **페이지 110, 그림 3.5 : 순차 큐에서의 삽입과 삭제**

| *front* | *rear* | *Q[0]* | *Q[1]* | *Q[2]* | *Q[3]* | 설명 |
|---|---|---|---|---|---|---|
| *-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* 1의 삭제 |

### ▶ 페이지 108 ~ 109, 구조 3.2 : 큐 추상 데이터 타입

structure *Queue*
  objects : 0개 이상의 원소를 가진 유한 순서 리스트
  functions : 모든 *queue* ∈ *Queue*, *item* ∈ *element*,
    *max_queue_size* ∈ *positive integer*
    *Stack* CreateQ(*max_queue_size*)
      ::= 최대 크기가 *max_queue_size*인 공백 큐를 생성
    *Boolean* IsFull(*queue*, *max_queue_size*)
      ::= if (*queue*의 원소수 == *max_queue_size*) return *TRUE*
        else return *FALSE*
    *Boolean* IsEmpty(*queue*)
      ::= if (*queue* == CreateQ(*max_queue_size*)) return *TRUE*
        else return *FALSE*
    *Stack* AddQ(*queue*, *item*)
      ::= if (IsFull(*queue*)) *queue_full*
        else *queue*의 톱에 *item*을 삽입하고 이 *queue*를 반환
    *Element* DeleteQ(*queue*)
      ::= if (IsEmpty(*queue*)) return
        else *queue*의 앞에 있는 *item*을 제거해서 반환

□ 큐의 구현

- 일차원 배열 *queue*[MAX_QUEUE_SIZE]로 표현
- 두 변수 *front*, *rear* 필요

**▶ 페이지 109, 큐 연산의 구현 & 페이지 110, 프로그램 3.3 : 큐에 삽입**
**& 페이지 110, 프로그램 3.4 : 큐에서의 삭제**

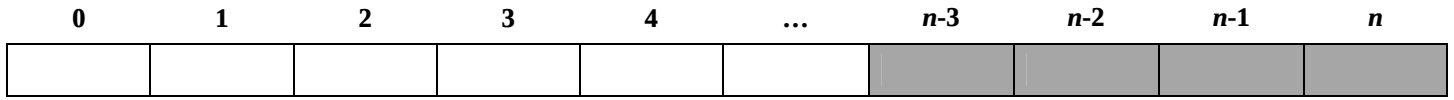
```
Queue  CreateQ(max_queue_size) ::=
        #define MAX_QUEUE_SIZE 100     /* 큐의 최대 크기 */
        typedef struct {
            int key;
            /* 다른 필드 */
        } element;
        element queue[MAX_QUEUE_SIZE];
        int rear = -1;
        int front = -1;
Boolean  IsEmpty(queue) ::= front == rear;
Boolean  IsFull(queue) ::= rear == MAX_STACK_SIZE – 1;
```

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

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

## ▶ 첫번째 표현

- 큐에 작업이 들어가고 나옴에 따라 큐가 점차 오른쪽으로 이동
- queue_full



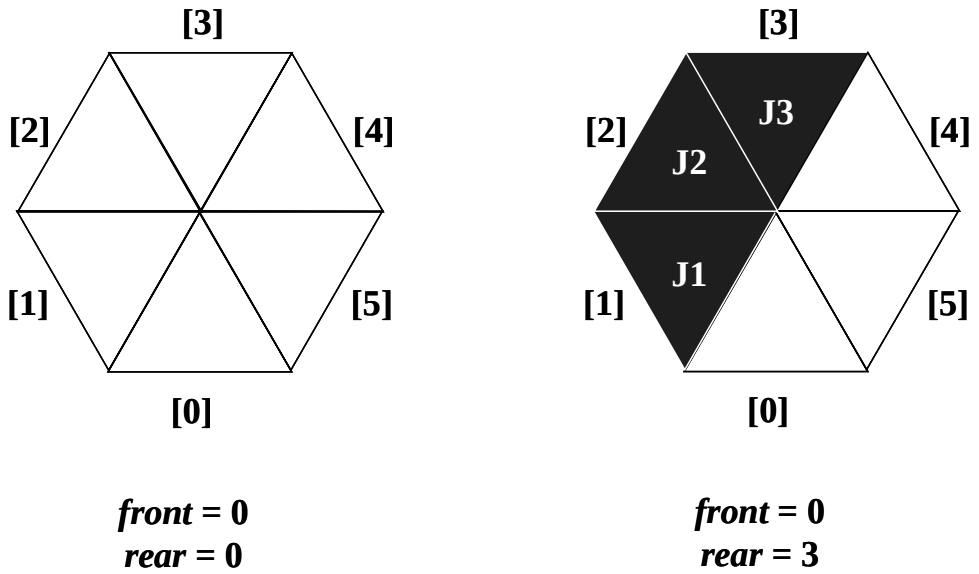

  - 반드시 큐에 *n*개의 원소가 있다는 것을 나타내는 것은 아니다
  - 첫번째 원소가 Q[0]에 오도록 조정할 필요가 있다
    : 전체 큐를 왼쪽으로 이동시켜야 한다
  - 배열 이동은 시간이 많이 든다
    : 효율적인 표현이 필요

✓ 작업 큐 동작 참조

⇒ 원형 큐 (circular queue)

- more efficient queue representation
- 배열 *queue*[MAX_QUEUE_SIZE]를 원형으로 가정
- *front*는 항상 큐의 첫번째 원소로부터 반시계 방향으로 하나 앞 위치를 가리키게 한다.
  - 큐가 비어 있을 때에만 *front* = *rear*
  - 초기 상태는 *front* = *rear* = 0

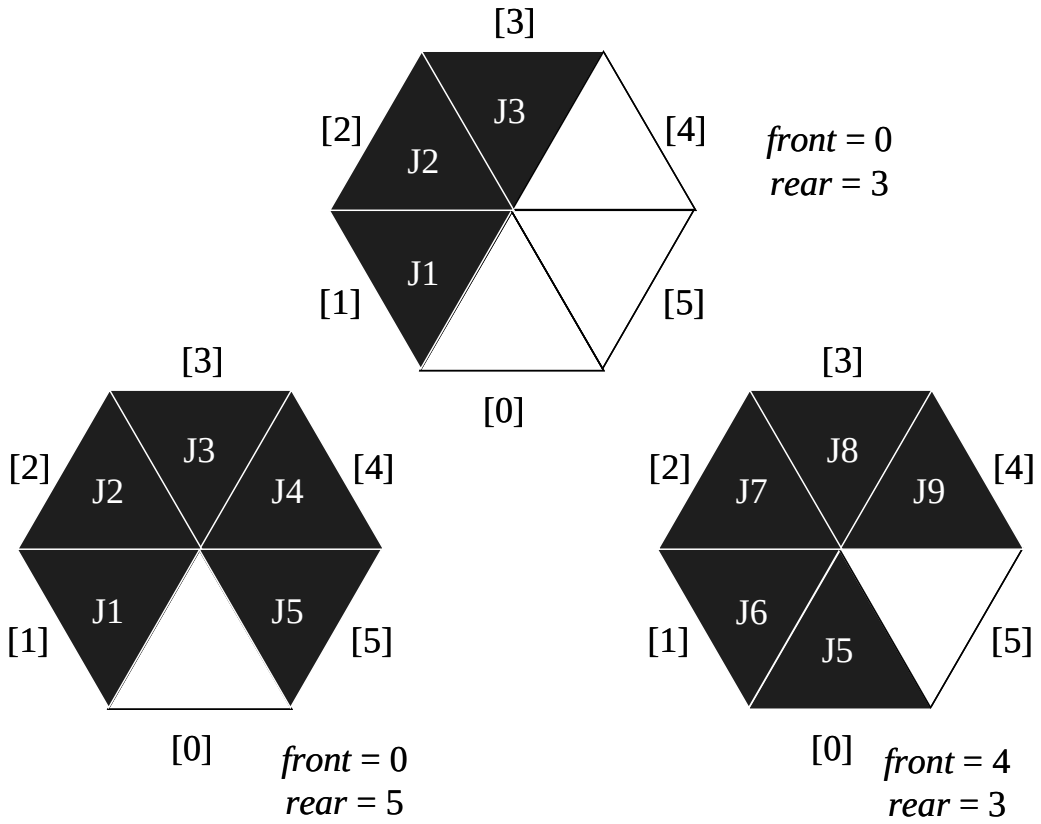
✧ **페이지 111, 그림 3.6 : 공백 원형 큐와 공백이 아닌 원형 큐**




***front* = 0**
***rear* = 0**

***front* = 0**
***rear* = 3**

✧ **페이지 111, 그림 3.7 : 포화 원형 큐**




- 실제로는 한 원소 분의 여유 공간이 있음
- 포화큐와 공백큐를 구별할 수 없으므로

✧ **페이지 112, 프로그램 3.5 : 원형 큐에서의 삽입**

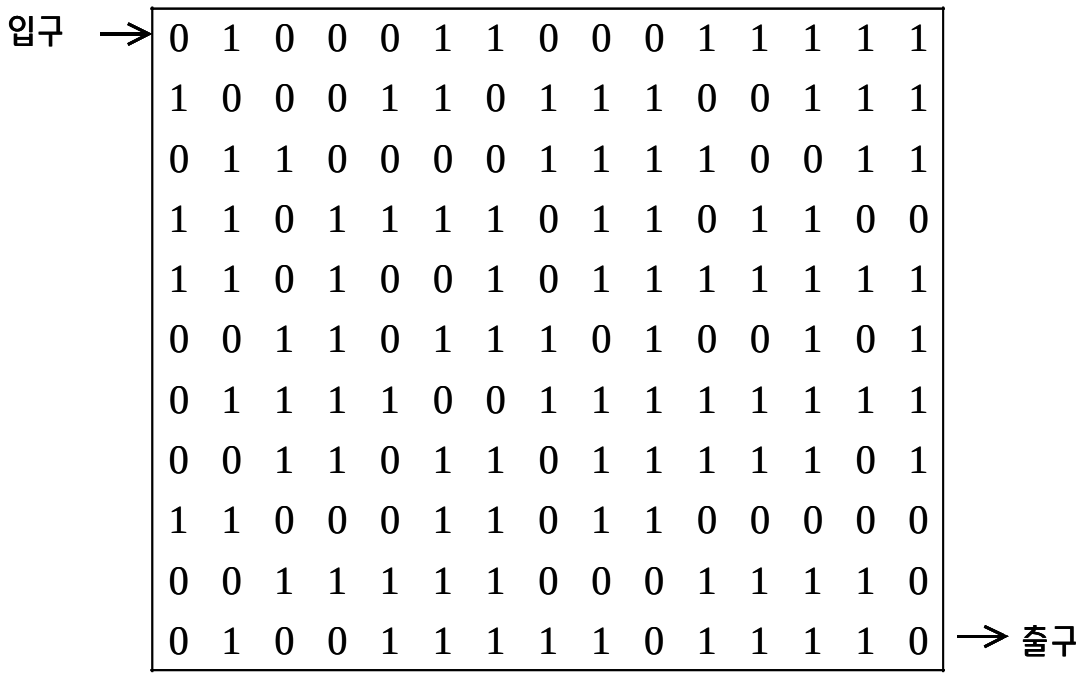
**& 페이지 112 ~ 113, 프로그램 3.6 : 원형 큐에서의 삭제**

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

# 3.3 미로 문제

▶ 페이지 114, 그림 3.8 : 예제 미로



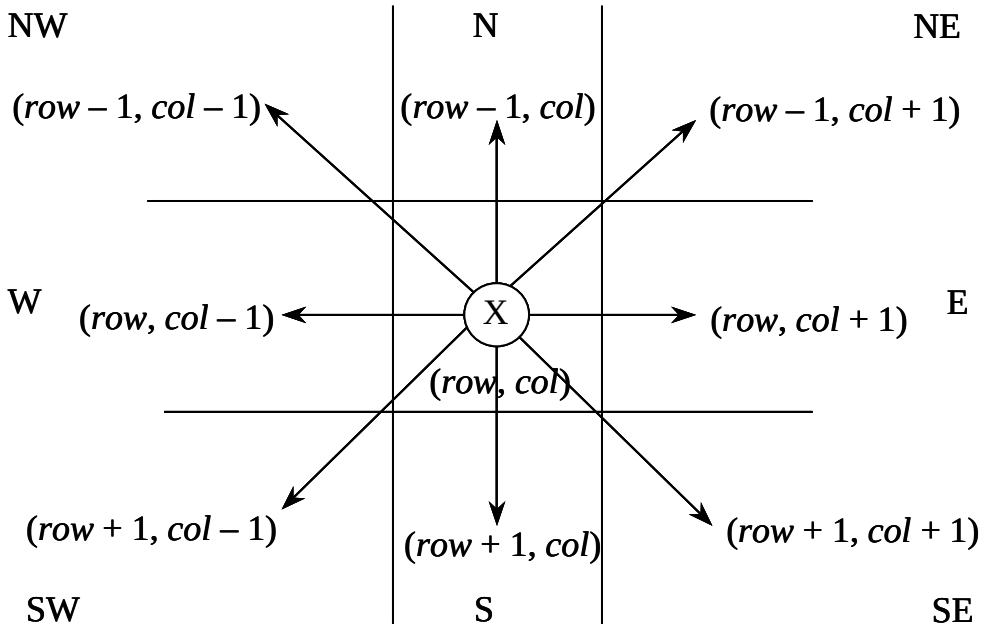
## □ 미로의 표현

- 이차원 배열, *maze*[*i*][*j*], $1 \le i \le m$, $1 \le j \le p$
  - 1 : 통로가 막힘
  - 0 : 통과할 수 있음
  - 입구 - *maze*[1][1]
  - 출구 - *maze*[*m*][*p*]
- 주의 !
  - 모든 위치가 여덟 개의 방향을 가지고 있는 것은 아니다
  - 경계 조건을 매번 검사하는 것을 피하기 위해서 미로의 주위를 1로 둘러싼다

□ 배열 *move*의 사용

- 이동할 수 있는 방향을 미리 배열 *move*에 표현

✧ **페이지 115, 그림 3.9 : 가능한 이동**




$(i, j)$에서 가능한 이동

✧ **페이지 116, 그림 3.10 : 이동 테이블 & 페이지 115, 이동 테이블의 선언**

```
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 이동할 위치, *maze*[*next_row*][*next_col*]

: *next_row* = *row* + *move*[*dir*].*vert*;

: *next_col* = *col* + *move*[*dir*].*horiz*;

## □ 미로 알고리즘의 설계

- 미로를 통하여 이동해 갈 때 여러 방향으로 이동할 수 있을 때 우선 하나의 길을 선택하고 현재 위치와 마지막 이동방향을 리스트에 기억해 둔다
- 각각의 새로운 방향에 대해, 북에서 시작하여 시계방향으로 가능성을 조사
- 같은 경로를 두번 가지 않기 위해 *mark*[*m* + 2][*n* + 2]을 0으로 초기화시켜 사용
  - *maze*[*m*][*n*] = 0

### ▶ 페이지 116 ~ 117, 프로그램 3.7 : 초기 미로 알고리즘

```
initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty) {
    <row, col, dir> = delete from top of stack; /* 스택의 톱에 있는 위치로 이동 */
    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) {
            /* 가능한 이동으로 아직 시도해보지 않은 위치 */
            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;
        }
    }
}
```

## □ 경로의 표현

- 삼원소 < *row*, *col*, *direction* >들의 집합
- 스택을 사용하는 것이 바람직하다
  - 나중에 들어간 삼원소들을 제일 먼저 제거하므로
- 스택 크기의 한계
  - 스택 크기 = $m \times p$(경로 길이)
  - 통과 경로의 길이 < $\lceil m / 2 \rceil (n + 1)$

✓ 페이지 117, 그림 3.11 : 긴 경로를 가진 미로

## ▶ 페이지 118 ~ 119, 프로그램 3.8 : 미로 탐색 알고리즘

```
void path(void)
{
    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) { /* dir 방향으로 이동 */
            next_row = row + move[dir].vert; 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 :\n"); printf("row col\n");
        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\n");
}
```

✍ 분석 : O(*mp*)

# 3.4 수식의 계산

## 3.4.1 서 론

□ 수식

- 피연산자(operand), 연산자(operator), 분리자(delimeter)로 구성

✓ $x = a / b - c + d * e - a * c$

□ 우선순위

- 각 연산자에 우선순위를 부여 (페이지 121, 그림 3.13 : C 언어의 우선순위 계층)

✓ $x = (((a / b) - c) + (d * e)) - (a * c)$

□ 표기법

- 후위 표기법 (postfix notation)
  - 연산자가 피연산자들 뒤에 온다
- 중위 표기법 (infix notation)
  - 피연산자 사이에 이항 연산자가 위치

✓ infix : $a / b - c + d * e - a * c$
postfix : $a\ b / c - d\ e * + a\ c * -$

✧ **페이지 122, 그림 3.13 : 중위 표기와 후위 표기**

| 중위 표기 | 후위 표기 |
|---|---|
| *2 + 3 * 4* | *2 3 4 * +* |
| *a * b + 5* | *a b * 5 +* |
| *(1 + 2) * 7* | *1 2 + 7 *** |
| *a * b / c* | *a b * c /* |
| *((a / (b – c + d) * (e – a) * c* | *a b c – d + / e a – * c **|
| *a / b – c + d * e – a * c* | *a b / c – d e * a c * –* |

## 3.4.2 후위 표기식의 연산

□ 수식을 표기하는 표준 방식은 중위 표기법

- 거의 모든 수식에 이 표기법을 사용
- 컴파일러(compiler)에서는 후위 표기법을 사용 !!

⇒ 후위 표기법의 장점

- 수식의 계산이 간단하다
- 괄호(parentheses)가 필요 없다
- 연산자의 우선순위가 의미 없다

## ▶ 페이지 122, 그림 3.14 : 후위 표기식의 연산

✓ 6 2 / 3 - 4 2 * +

| 토큰 | 스택 [0] | [1] | [2] | Top |
|---|---|---|---|---|
| 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 |

- 스택을 사용
  - 연산자를 만날 때까지 피연산자를 스택에 저장
  - 연산자를 만나면 필요한 만큼의 피연산자를 스택에서 가져와 연산을 실행하고 연산을 결과를 스택에 저장

▶ **페이지 123 ~ 124, 프로그램 3.9 : 후위 표기식을 처리하는 함수**

**페이지 124 ~ 125, 프로그램 3.10 : 입력 문자열로부터 토큰을 생성하는 함수**

```
void eval(void)
{   /* 전역 변수로 되어 있는 후위 표기식 expr을 연산한다 */
    precedence token; char symbol; int op1, op2;
    int n = 0;   /* 수식 문자열을 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 수행한 후, 그 결과를 스택에 삽입함 */
            op2 = delete(&top); op1 = delete(&top); /* 스택 삭제 */
            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); break;
            }
            token = get_token(&symbol, &n);
        }
        return delete(&top);
}
```

```
precedence get_token (char *symbol, int * n)
{
/* 다음 토큰을 취한다. symbol은 문자 표현이며, token은 그것의 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 ' '       : return eos;
          default : return operand;
   }
}
```

## 3.4.3 중위 표기에서 후위 표기로의 변환

□ 간단한 알고리즘

- 식을 전부 괄호로 묶는다
- 이항 연산자들을 모두 자기 오른쪽 괄호로 이동
- 괄호를 전부 삭제

✓ 예

- $a / b - c + d * e - a * c$

→ $((((a / b) - c) + (d * e)) - (a * c))$

→ $a\,b / c - d\,e * + a\,c * -$

□ 효과적인 알고리즘

✧ **페이지 126, 그림 3.15 : 중위 표기식의 후위 표기식으로의 변환**

– $a + b * c$

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

◆ 스택 사용

– 피연산자들은 즉시 출력

– 스택의 톱에 있는 연산자의 우선 순위가 스택에 들어올 연산자보다 작을 때는 연산자를 삽입

: 높은 우선 순위의 연산자는 낮은 우선 순위의 연산자보다 먼저 출력되어야 하므로

– 수식의 끝에 도달하면 스택의 모든 연산자를 출력

✧ **페이지 127, isp(in-stack precedence)와 icp(incoming precedence)**

```
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 };
```

✧ **페이지 127 ~ 128, 프로그램 3.11 : 중위 표기를 후위 표기로 변환하는 함수**

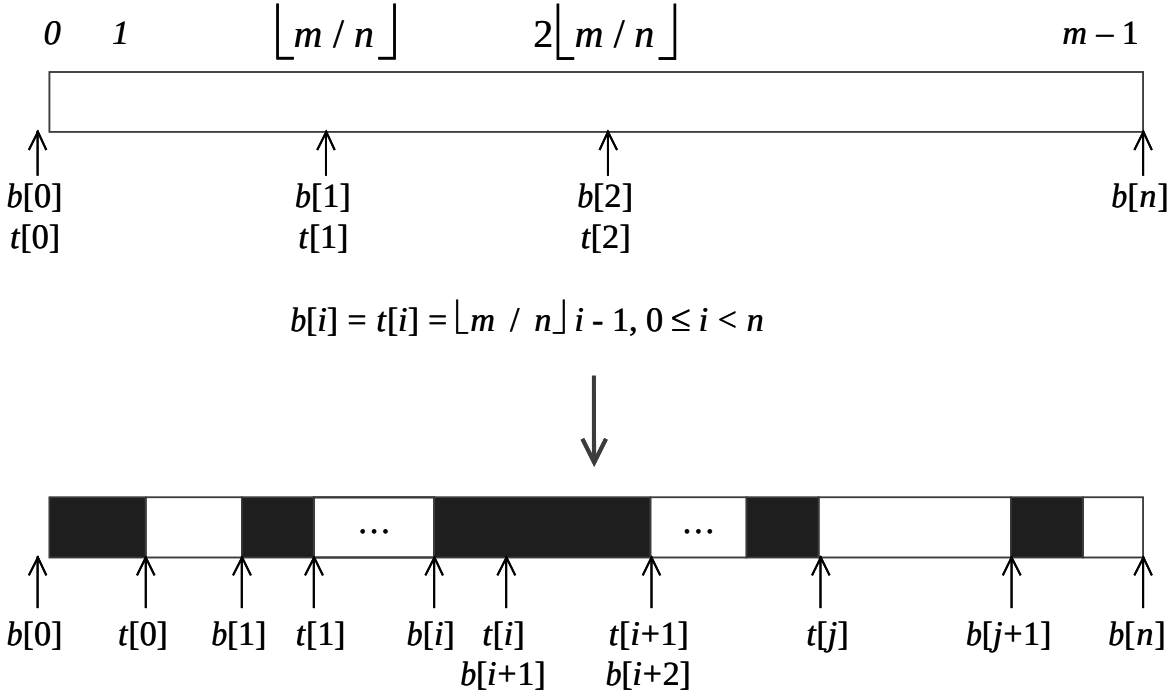
```
void postfix(void)
{
    char symbol; precedence token; int n = 0;
    int top = 0; stack[0] = eos; /* eos를 스택에 놓는다 */
    for (token = get_token(&symbol, &n); token != eos;
            token = get_token(&symbol, &n)) {
        if (token == operand) printf("%c", symbol);
        else if (token == rparen) { /* 왼쪽 괄호가 나올 때까지 토큰들을 제거해서 출력 */
            while (stack[top] != lparen) print_token(delete(&top));
            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);
        }
    }
    while ((token = delete(&top)) != eos) print_token(token); printf("\n");
}
```

## 3.5 다중 스택과 큐

□ 이용 가능한 기억장소 $V(1:m)$을 $n$조각으로 나누어 $n$개의 스택에 각각 한 조각씩 할당

- 각 스택 $i$에 대하여 $B(i)$는 그 스택의 최하단 원소보다 하나 작은 위치를 나타내고, $T(i)$는 톱원소를 나타낸다
- $i$번째 스택이 비어있으면 $B(i) = T(i)$가 된다
- 초기조건
  - $B(i) = T(i) = \lfloor m/n \rfloor (i-1), 1 \le i \le n$
- 스택 $i$는 $B(i) + 1$로부터 $B(i+1)$의 위치까지 커질 수 있다
- $B(n+1) = m$으로 한다

▶ **페이지 131, 그림 3.18 : *m*개의 스택에 대한 초기 구성**



▶ **페이지 131 ~ 132, 프로그램 3.12 : 스택 *stack_no*에 *item* 삽입**

**& 페이지 132, 프로그램 3.13 : 스택 *stack_no*에서 *item* 삭제**

```
void add(int i, element item)
{
    // item을 i번째 스택에 삽입
    if (top[i] == boundary[i + 1])
        stack_full(i);
    memory[++top[i]] = item;
}

void delete(int i)
{
    // i번째 스택에서 top 원소를 제거
    if (top[i] == boundary[i])
        return stack_empty(i);
    return memory[top[i]--];
}
```

□ *stack_full*(*i*)의 처리

- *v*에 빈공간이 있는지를 확인해서 있다면 *i*번째 스택에 빈 공간을 제공해 줄 수 있도록 스택을 이동시켜야
- 스택 *j*와 *j* + 1사이에 빈 공간을 가지는 즉 $t(j) < b(j+1)$되는 최소의 $j(i < j \le n)$를 찾는다
  - 그러한 *j*가 있다면 스택 $i+1, i+2, \ldots, j$를 오른쪽으로 한 자리씩 옮겨서 스택 *i*, *i* + 1 사이에 공간을 만든다 ($v(1)$은 최좌단, $v(m)$은 최우단이다)
- 위와 같은 *j*가 없으면 스택 *i*의 왼쪽을 조사하여 *j*가 $1 \le j < i$ 인 *j*에 대해 스택 *j*와 스택 *j* + 1사이에 빈 공간이 있는, 즉 $t(j) < b(j+1)$ 되는 최대 *j*를 찾는다
  - 그런 *j*가 있으면 스택 $j+1, j+2, \ldots, i$를 왼쪽으로 옮겨서 스택 *i*와 *i* + 1 사이에 빈공간을 만든다
- 위의 조건을 만족하는 *j*가 없으면 *v*의 *m*개 공간은 모두 사용되고 있으며, 빈 공간은 없다