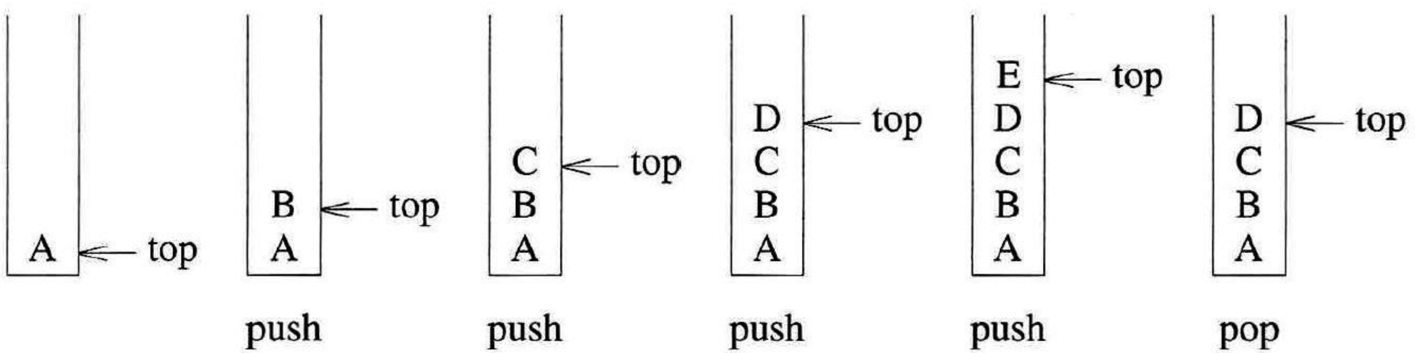
제3장 Stacks and queues

경북대학교 임경식 교수

3.1 Stacks

A *stack* is an ordered list in which insertions (also called pushes and adds) and deletions (also called pops and removes) are made at one end called the *top*. *(Last-In-First-Out, LIFO)*



3.1 Stacks

ADT *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer
    Stack CreateS(*maxStackSize*) ::=
        create an empty stack whose maximum size is *maxStackSize*
    Boolean IsFull(*stack*, *maxStackSize*) ::=
        if (number of elements in *stack* == *maxStackSize*)
        return *TRUE*
        else return *FALSE*
    Stack Push(*stack*, *item*) ::=
        if (IsFull(*stack*)) *stackFull*
        else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
        if (*stack* == CreateS(*maxStackSize*))
        return *TRUE*
        else return *FALSE*
    Element Pop(*stack*) ::=
        if (IsEmpty(*stack*)) **return**
        else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

3.1 Stacks

❖ Implementation of stack ADT using one-dimensional array

Stack CreateS(*maxStackSize*) ::=
```
      #define MAX_STACK_SIZE 100 /* maximum stack size */
      typedef struct {
              int key;
              /* other fields */
              } element;
      element stack[MAX_STACK_SIZE];
      int top = -1;
```
Boolean IsEmpty(Stack) ::= `top < 0;`

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

```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE-1)
      stackFull();
   stack[++top] = item;
}
```

```
void stackFull()
{
   fprintf(stderr, "Stack is full, cannot add element");
   exit(EXIT_FAILURE);
}
```

3.1 Stacks

```
element pop()
{/* delete and return the top element from the stack */
   if (top == -1)
      return stackEmpty(); /* returns an error key */
   return stack[top--];
}
```

3.2 Stacks with dynamic arrays

❖ Implementation of stack ADT using dynamic arrays

Stack CreateS() ::=
```
typedef struct {
        int key;
        /* other fields */
        } element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity = 1;
int top = -1;
```

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

Boolean IsFull(Stack) ::= `top >= capacity-1;`

```
void push(element item)
{/* add an item to the global stack */
  if (top >= MAX_STACK_SIZE-1)
    stackFull();
  stack[++top] = item;
}
```

// MAX_STACK_SIZE를 capacity로 대체

```
void stackFull()
{
   REALLOC(stack, 2 * capacity * sizeof(*stack))
   capacity *= 2;
}
```

3.3 Queues

A *queue* is an ordered list in which insertions and deletions take place at different ends, *rear* and *front*, *respectively. (First-In-First-Out, FIFO)*

ADT *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer
    Queue CreateQ(*maxQueueSize*) ::=
        create an empty queue whose maximum size is *maxQueueSize*
    Boolean IsFullQ(*queue*, *maxQueueSize*) ::=
        if (number of elements in *queue* == *maxQueueSize*)
        return *TRUE*
        else return *FALSE*
    Queue AddQ(*queue*, *item*) ::=
        if (IsFullQ(*queue*)) *queueFull*
        else insert *item* at rear of *queue* and return *queue*
    Boolean IsEmptyQ(*queue*) ::=
        if (*queue* == CreateQ(*maxQueueSize*))
        return *TRUE*
        else return *FALSE*
    Element DeleteQ(*queue*) ::=
        if (IsEmptyQ(*queue*)) **return**
        else remove and return the *item* at front of queue.

ADT 3.2: Abstract data type *Queue*

3.3 Queues

❖ Implementation of queue ADT using one-dimensional array

Queue CreateQ(*maxQueueSize*) ::=

```
      #define MAX_QUEUE_SIZE 100 /* maximum queue size */
      typedef struct {
              int key;
              /* other fields */
              } element;
      element queue[MAX_QUEUE_SIZE];
      int rear = -1;
      int front = -1;
```

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

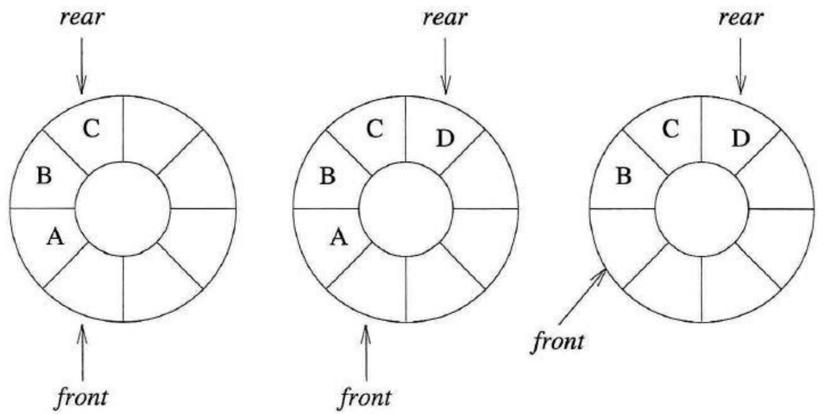
Boolean IsFullQ(*queue*) ::= `rear == MAX_QUEUE_SIZE-1`

```
void addq(element item)
{/* add an item to the queue */
  if (rear == MAX_QUEUE_SIZE-1)
    queueFull();
  queue[++rear] = item;
}

element deleteq()
{/* remove element at the front of the queue */
  if (front == rear)
    return queueEmpty(); /* return an error key */
  return queue[++front];
}
```

3.3 Queues

❖ Circular queue Implementation of queue ADT using one-dimensional array



(a) Initial (b) Addition (c) Deletion

```
if (rear == MAX_QUEUE_SIZE - 1) rear = 0;
else rear++;
```

or

(rear+ 1) % *MAX-QUEUE-SIZE.*

if front == rear, empty or full

```
void addq(element item)
{/* add an item to the queue */
   rear = (rear+1) % MAX_QUEUE_SIZE;
   if (front == rear)
      queueFull(); /* print error and exit */
   queue[rear] = item;
}
```

```
element deleteq()
{/* remove front element from the queue */
   element item;
      if (front == rear)
         return queueEmpty(); /* return an error key */
      front = (front+1) % MAX_QUEUE_SIZE;
      return queue[front];
}
```

3.5 A mazing problem

- An *m * p* maze will require an (m +2) x (*p* +2) array. The entrance is at position [1][1] and the exit at *[m][p].*

maze[(m+2)*(p+2)]

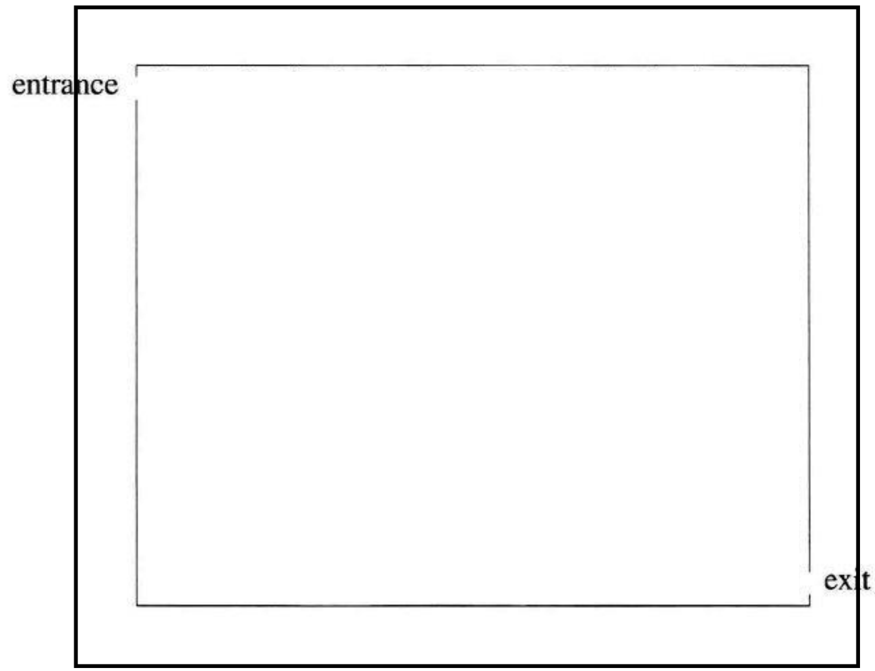
entrance

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

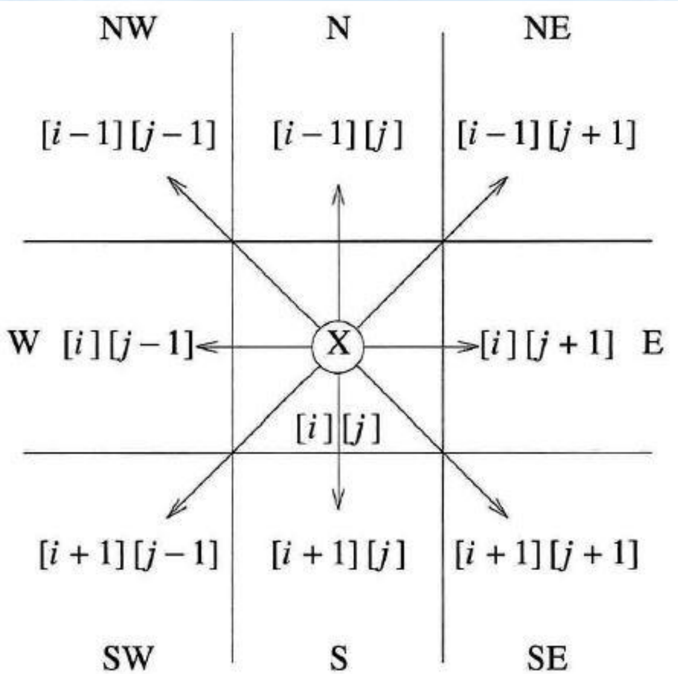
exit

maze

mark[(m+2)*(p+2)]



3.5 A mazing problem




```
typedef struct {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8]; /* array of moves
```

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

```
nextRow = row + move[dir].vert;
nextCol = col + move[dir].horiz;
```

3.5 A mazing problem

```
typedef struct {
        short int row;
        short int col;
        short int dir;
        } element;
```

```
initialize a stack to the maze's entrance coordinates and
direction to north;
while (stack is not empty) {
   /* move to position at top of stack */
   <row,col,dir> = delete from top of stack;
   while (there are more moves from current position) {
      <nextRow, nextCol> = coordinates of next move;
      dir = direction of move;
      if ((nextRow == EXIT_ROW) && (nextCol == EXIT_COL))
        success;
      if (maze[nextRow][nextCol] == 0 &&
                    mark[nextRow][nextCol] == 0) {
      /* legal move and haven't been there */
        mark[nextRow][nextCol] = 1;
        /* save current position and direction */
        add <row,col,dir> to the top of the stack;
        row = nextRow;
        col = nextCol;
        dir = north;
      }
   }
}
printf("No path found\n");
```

3.5 A mazing problem

```
void path(void)
{/* output a path through the maze if such a path exists */
  int i, row, col, nextRow, nextCol, 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 = pop();
    row = position.row;  col = position.col;
    dir = position.dir;
    while (dir <  8 && !found) {
      /* move in direction dir */
      nextRow = row + move[dir].vert;
      nextCol = col + move[dir].horiz;
      if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
        found = TRUE;
      else if ( !maze[nextRow][nextCol] &&
      ! mark[nextRow][nextCol]) {
        mark[nextRow][nextCol] = 1;
        position.row = row; position.col = col;
        position.dir = ++dir;
        push(position);
        row = nextRow; col = nextCol; dir = 0;
      }
      else ++dir;
    }
  }
```

3.5 A mazing problem

```
    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\n",row,col);
      printf("%2d%5d\n",EXIT_ROW,EXIT_COL);
    }
    else printf("The maze does not have a path\n");
}
```

Time complexity O(mp)
Space complexity O(mp)

3.6 Evaluation of expressions

- To evaluate a expression, convert from infix to postfix & evaluate postfix.

Token	Stack [0]	Stack [1]	Stack [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	*			0	*abc* +**d**

Figure 3.16: Translation of $a*(b+c)*d$ to postfix

Token	Stack [0]	Stack [1]	Stack [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

Figure 3.14: Postfix evaluation

3.6 Evaluation of expressions

Token	Operator	Precedence[1]	Associativity
() [] → .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

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

(=> isp == 0
icp == 20

) => isp == 19
icp == 19

Convert from infix to postfix

```
void postfix(void)
{/* output the postfix of the expression. The expression
    string, the stack, and top are global */
  char symbol;
  precedence token;
  int n = 0;
  int top = 0;    /* place eos on stack */
  stack[0] = eos;
  for (token = getToken(&symbol, &n); token != eos;
                              token = getToken(&symbol,&n)) {
    if (token == operand)
      printf("%c", symbol);
    else if (token == rparen) {
      /* unstack tokens until left parenthesis */
      while (stack[top] != lparen)
        printToken(pop());
      pop();  /* discard the left parenthesis */
    }
    else {
      /* remove and print symbols whose isp is greater
         than or equal to the current token's icp */
      while(isp[stack[top]] >= icp[token])
        printToken(pop());
      push(token);
    }
  }
  while ( (token = pop()) != eos)
    printToken(token);
  printf("\n");
}
```

Convert from infix to postfix

```
/* isp and icp arrays -- index is value of precedence
   lparen, rparen, plus, minus, times, divide, mod, eos */
int isp[] = {0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

```
typedef enum {lparen, rparen, plus, minus, times, divide,
                          mod, eos, operand} precedence;
```

```
precedence getToken(char *symbol, int *n)
{/* get the next token, symbol is the character
    representation, which is returned, the token is
    represented by its enumerated value, which
    is returned in the function name */
   *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; /* no error checking,
                                   default is operand */
   }
}
```

Evaluate postfix

```
precedence token;
char symbol;
int op1, op2;
int n = 0; /* counter for the expression string */
int top = -1;
token = getToken(&symbol, &n);
while (token != eos) {
  if (token == operand)
    push(symbol-'0'); /* stack insert */
  else {
    /* pop two operands, perform operation, and
       push result to the stack */
    op2 = pop(); /* stack delete */
    op1 = pop();
    switch(token) {
      case plus: push(op1+op2);
                 break;
      case minus: push(op1-op2);
                  break;
      case times: push(op1*op2);
                  break;
      case divide: push(op1/op2);
                   break;
      case mod: push(op1%op2);
    }
  }
  token = getToken(&symbol, &n);
}
return pop(); /* return result */
```

Time complexity : O(n) => theta(n)
Space complexity : O(n) => theta(n)
where n is the # of tokens

Thank You !

▌ 노력 없이 이룰 수 있는 것 아무것도 없다.