Stacks and Queues

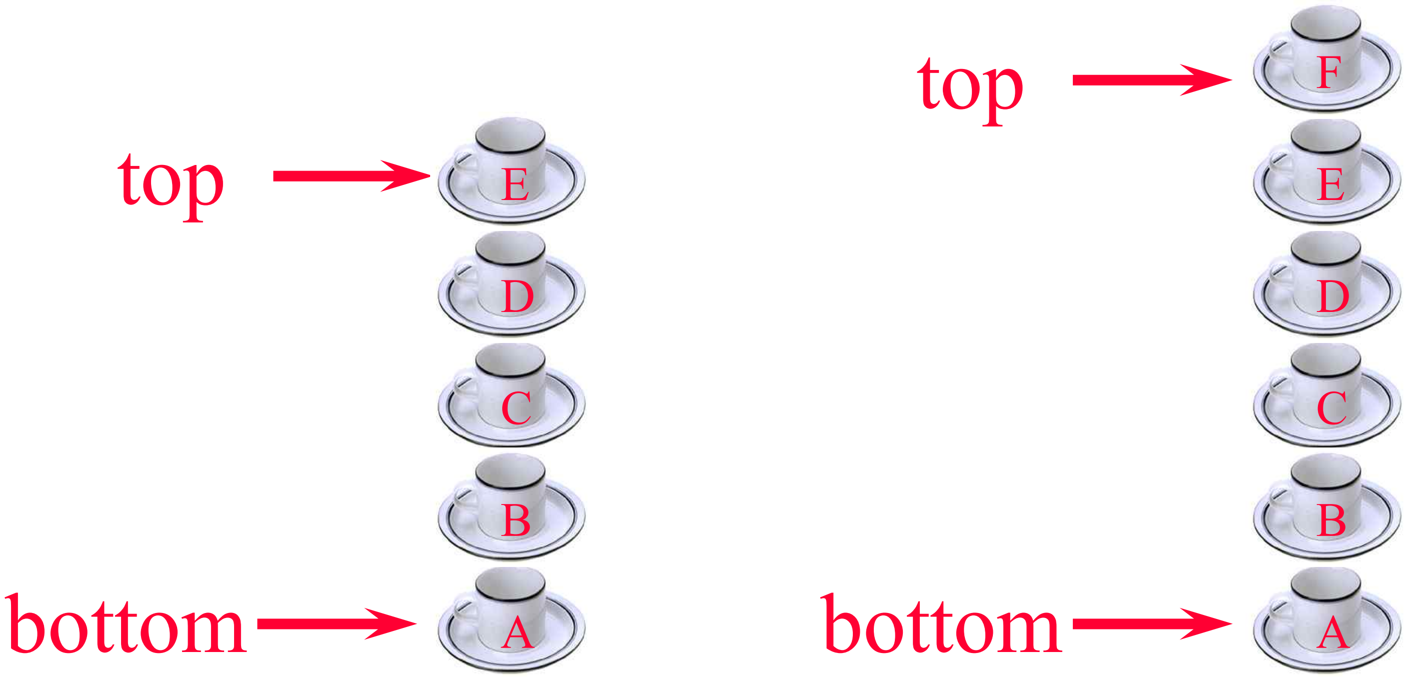
Chapter 3

Contents

Stacks

- Ordered linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.

Stack Of Cups



- Add a cup to the stack.
- Remove a cup from the stack.
- A stack is a LIFO list.

Stacks

- Standard operations:
  - IsEmpty … return true iff stack is empty
  - IsFull … return true iff stack has no remaining capacity
  - Top … return top element of stack
  - Push … add an element to the top of the stack
  - Pop … delete the top element of the stack

ADT of Stack

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*

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*) ::=

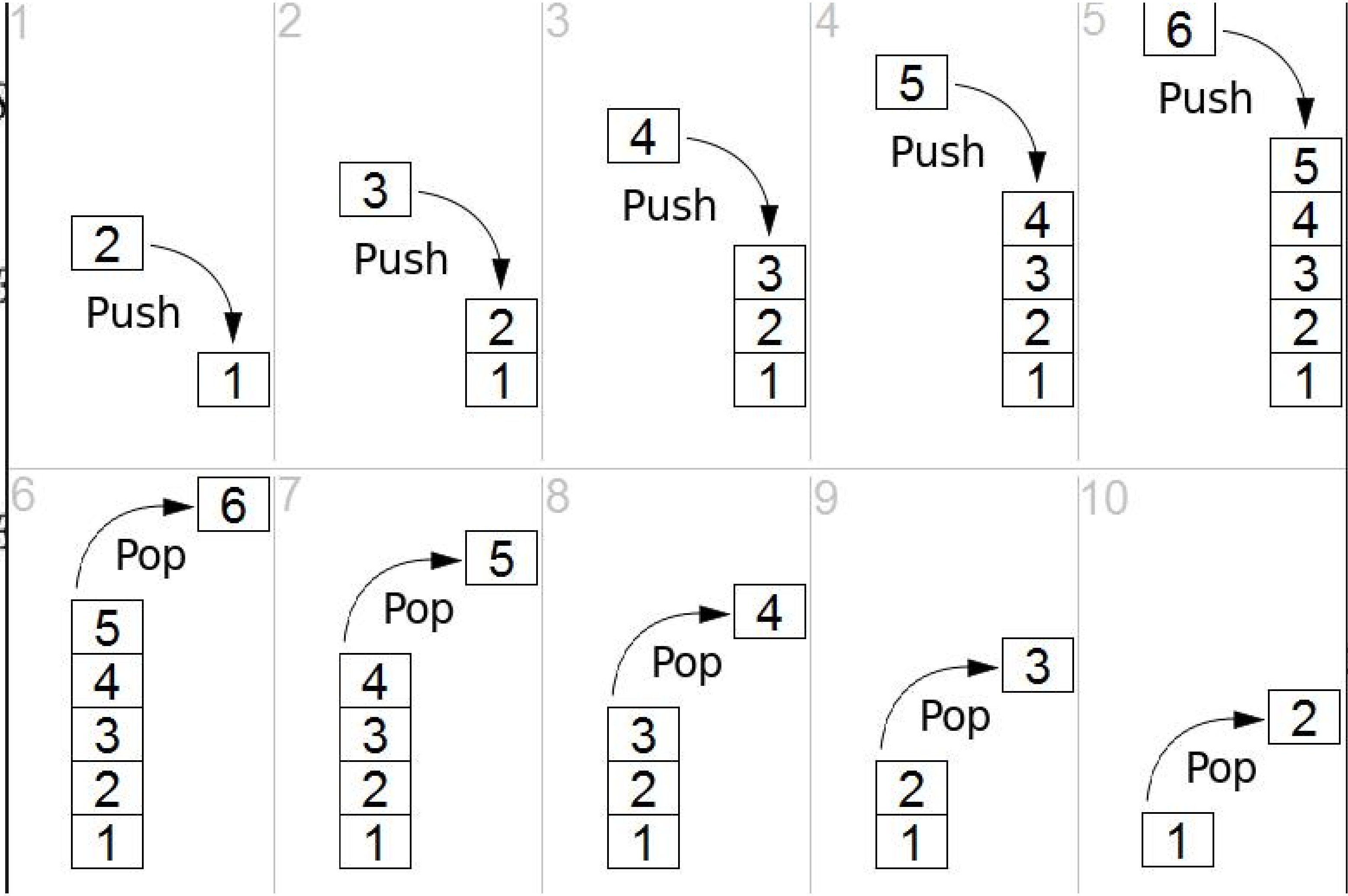
create an empty stack whose maximum size is *maxStackSize*

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

if (number of elements in *stack* == *maxStackSize*)

return *TRUE*

else return *FALSE*


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
Pop
6
5
4
3
2
1
7
Pop
5
4
3
2
1
8
Pop
4
3
2
1
9
Pop
3
2
1
10
Pop
2
1

he stack.

Stack creation in C

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

```
#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;
```

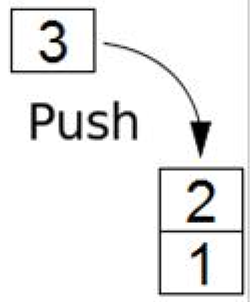
- Use a 1D array to represent a stack.
- Stack elements are stored in stack[0] through stack[top].

Implementation of Stack operations in C

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

stack, top 모두 전역

top➔

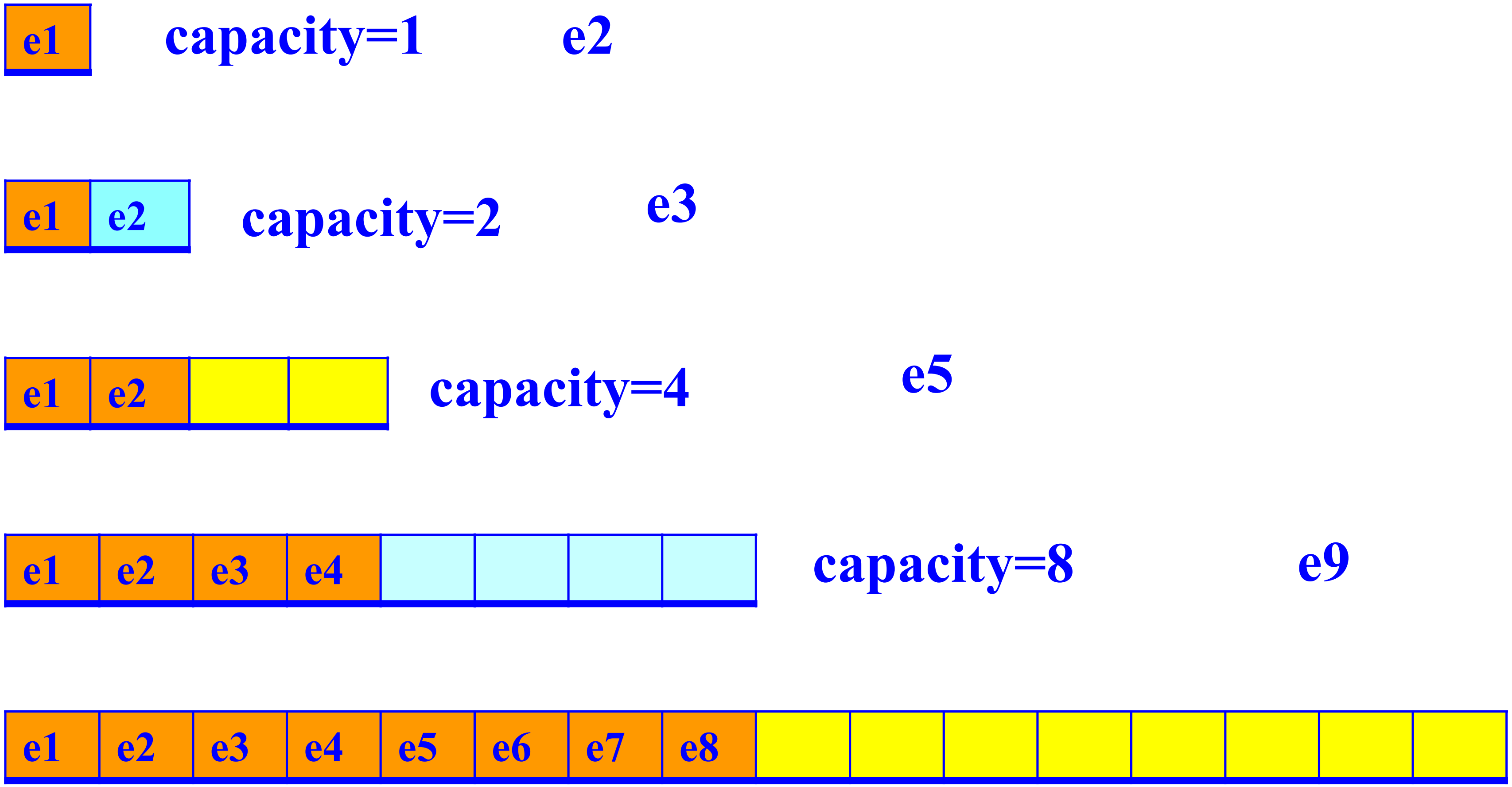



```
element pop()
{/* delete and return the top element from the stack */
   if (top == -1)
      return stackEmpty(); /* returns an error key */
   return stack[top--];
}
```

top➔


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

array doubling



Stacks using dynamic arrays

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

```
typedef struct {
    int key;
        /* other fields*/
  } element;
```

```
#define MALLOC(p,s) \
  if(!((p) = malloc(s))){ \
     fprintf(stderr, "insufficient memory"); \
     exit(EXIT_FAILURE); \
  }
```

element *stack;

MALLOC(stack, sizeof(*stack));

int capacity = 1; **stack=(element*)malloc(sizeof(element));**

int top = -1;

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

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

Stacks using dynamic arrays: array doubling (1)

- When stack is full, double the capacity using **REALLOC**.
  - Called ***array doubling***

```
stack=(element*)realloc(stack, 2*capacity*sizeof(element));
```

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

```
#define REALLOC(p,s) \
  if(!((p) = realloc(p,s))){ \
    fprintf(stderr, “insufficient memory”);  \
    exit(EXIT_FAILURE); \
  }
```

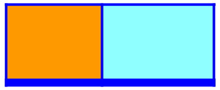
Stacks using dynamic arrays: array doubling (2)

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

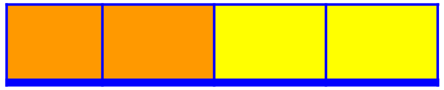
```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE-1)
      stackFull();
   stack[++top] = item;
}
```

Copy 횟수

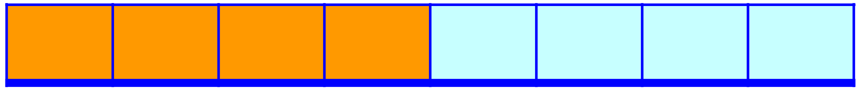
- 1번째 doubling: $\mathbf{2^{0}}$+1 push



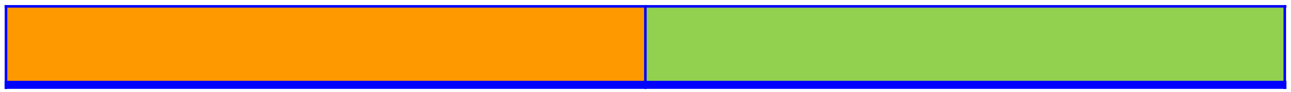
- 2번째 doubling: $\mathbf{2^{1}}$+1 push



- 3번째 doubling: $\mathbf{2^{2}}$+1 push



- 4번째 doubling: $\mathbf{2^{3}}$+1 push
- …



- k번째 doubling: $\mathbf{2^{k-1}}$+1 push

K번 doubling -> copy는 2^k-1 번 발생

n번 push 발생 -> doubling은 log2(n)=k 번 발생

- 만일 push가 32번 발생하였다➔
  - doubling횟수는?
  - copy횟수는?
    - 1번 doubling: 1
    - 2번 doubling: 2
    - 3번 doubling: 2^2
    - 4번 doubling: 2^3
    - 5번 doubling: 2^4
  - total copy횟수는?
    - $1+2+...+2^4 = 2^5-1$

Stacks using dynamic arrays: complexity of array doubling

- k번 doubling ➔ 최소 $2^{k-1}+1$번 push
- 매번 doubling시마다 2^{k-1}번 copy 가 발생
- 따라서 $\Sigma_{1<=i<=k} 2^{i-1}=2^k$
- ➔ $O(2^k)$ k는 doubling수
- push가 n번 발생 ➔ $\lceil \log_2 n \rceil$ 번 doubling
  - (2번째 push: 1번/ 3번째 push: 2번/5번째 3번...)
- 따라서 $O(2^{\log_2 n}) = O(n)$

Stacks using dynamic arrays: complexity of array doubling

- Let final value of capacity be 2^k
- Number of pushes is at least $2^{k-1} + 1$
- Total time spent on array doubling is $\Sigma_{1<=i<=k} 2^{i-1}$
- This is $O(2^k)$ k번 doubling발생할 경우
- So, although the time for an individual push is O(capacity), the time for all n pushes remains O(n)!

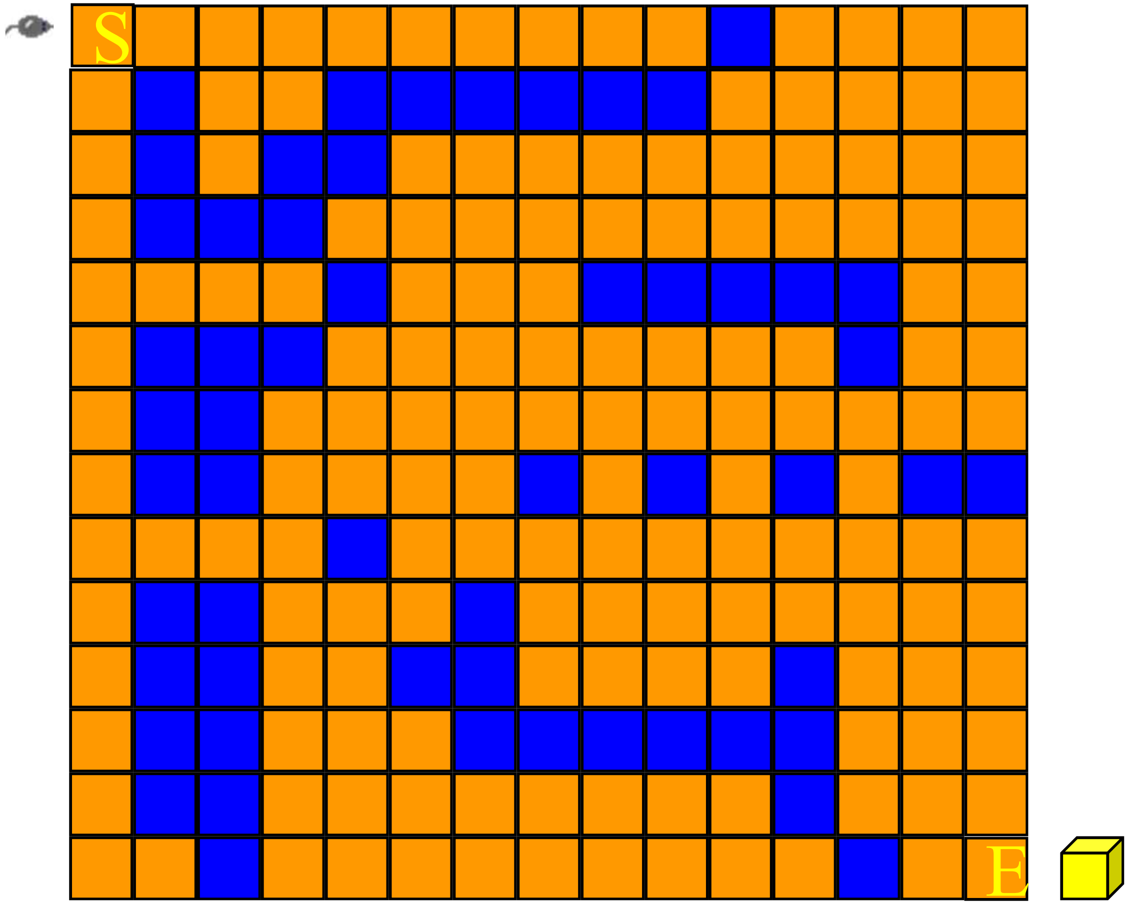
n번 push의 경우(n>2): array doubling회수는 <u>$\log_2 n$ +1</u>

$\log_2 n$번 doubling➔ 전체 time complexity $O(2^{\log_2 n})$ ➔ O(n)

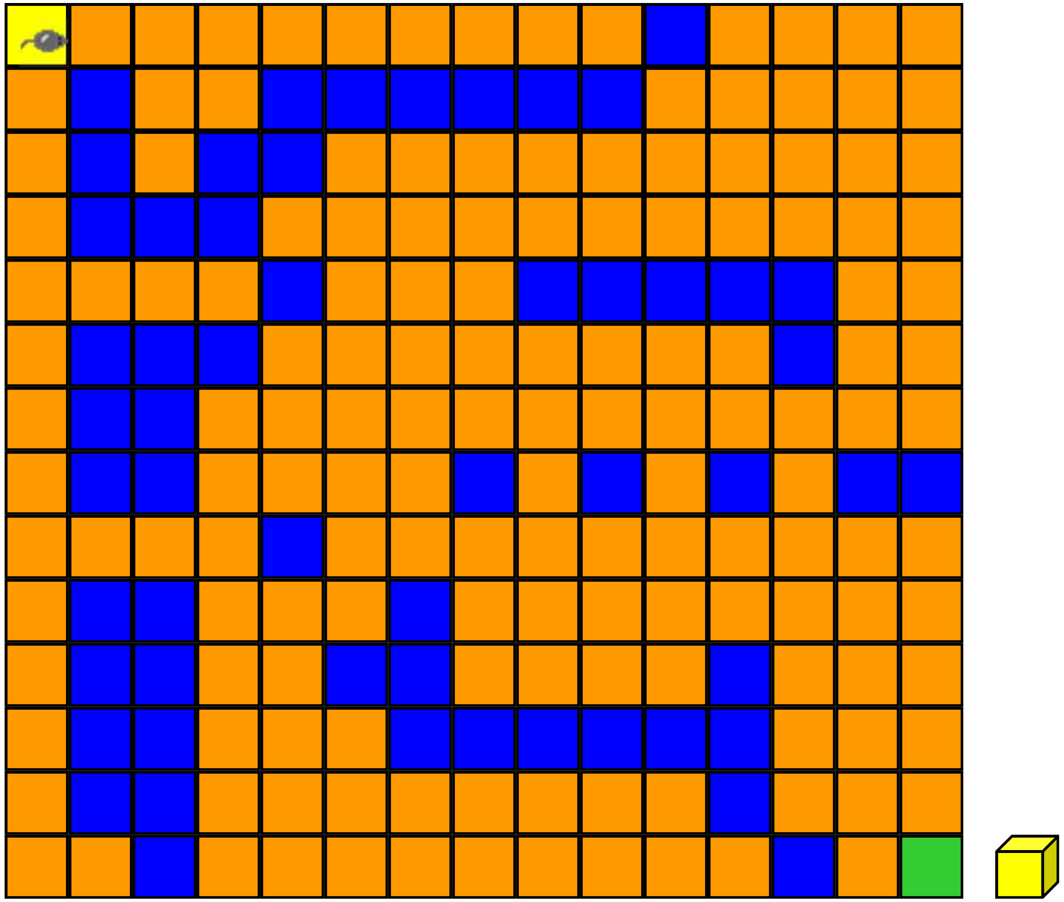
A Mazing Problem

- A maze is represented as a two-dimensional array
- The location of the rat in the maze can at any time be described by the row and column position
- We use compass points to specify the eight directions of movement
  - N, NE, E, SE, S, SW, W, NW

Rat In A Maze

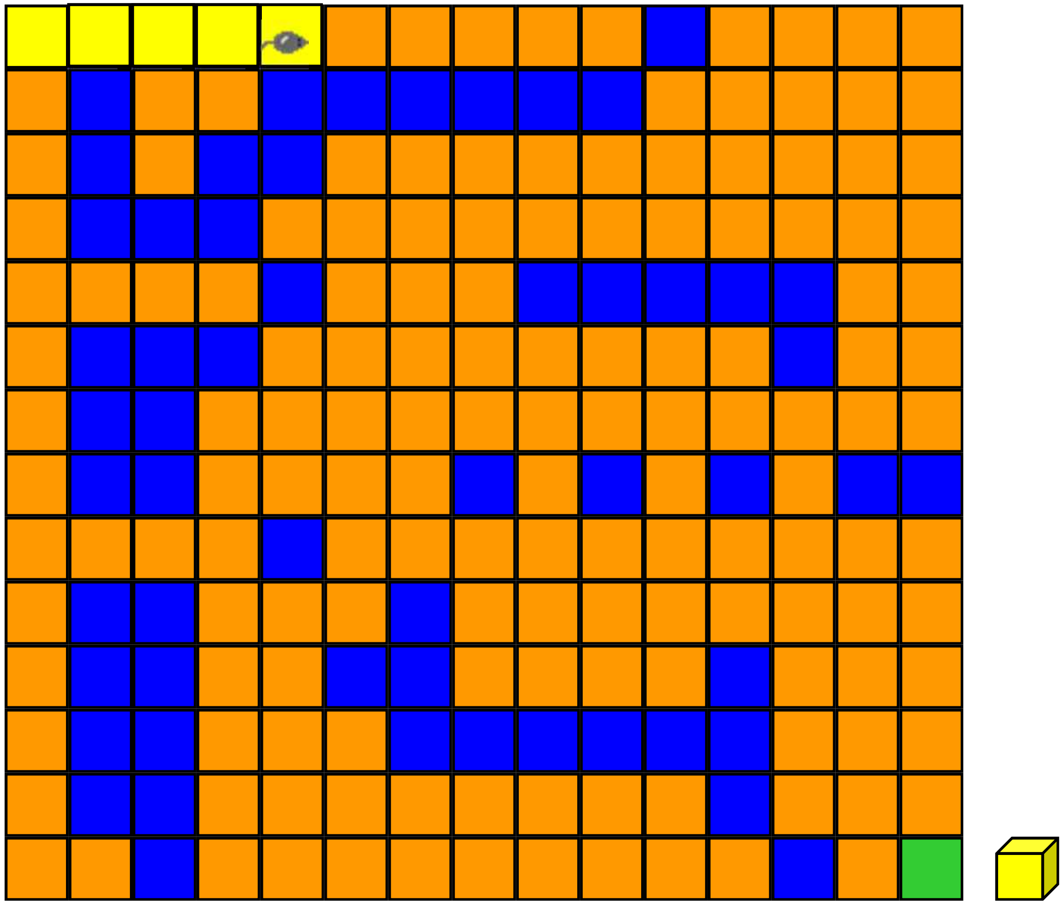


Rat In A Maze



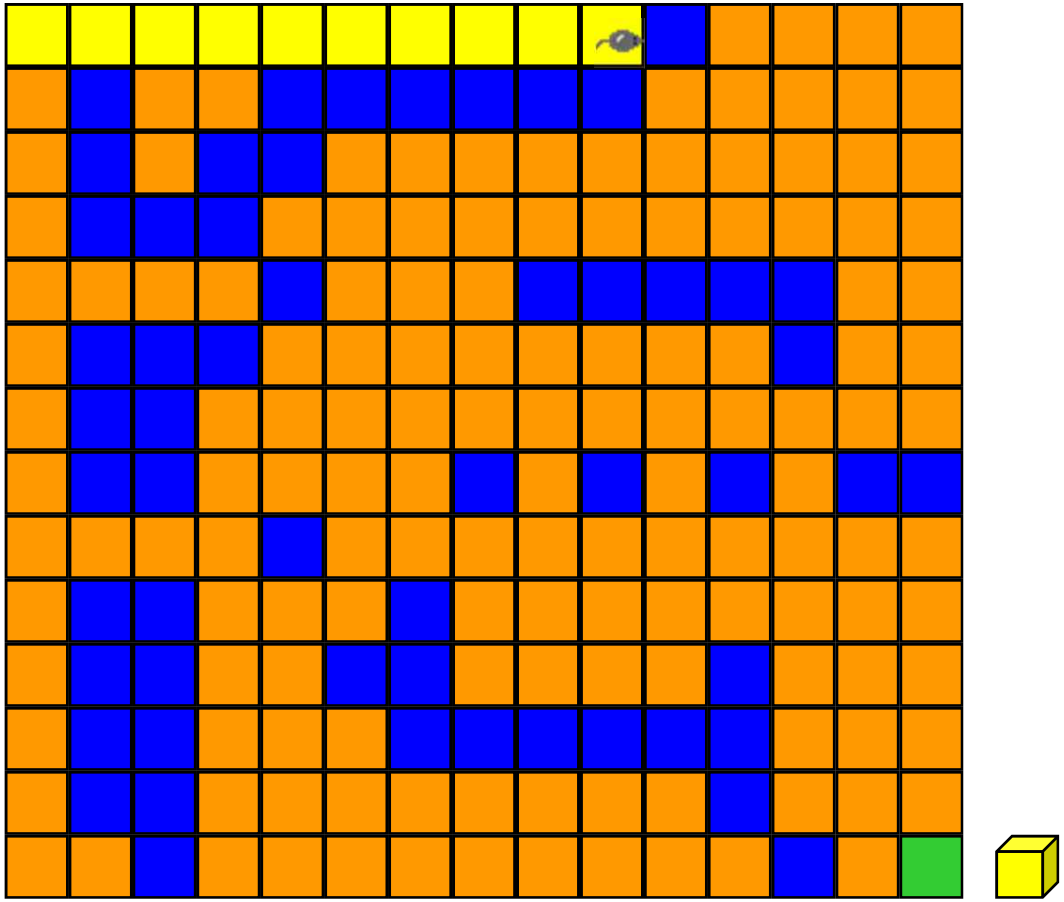
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



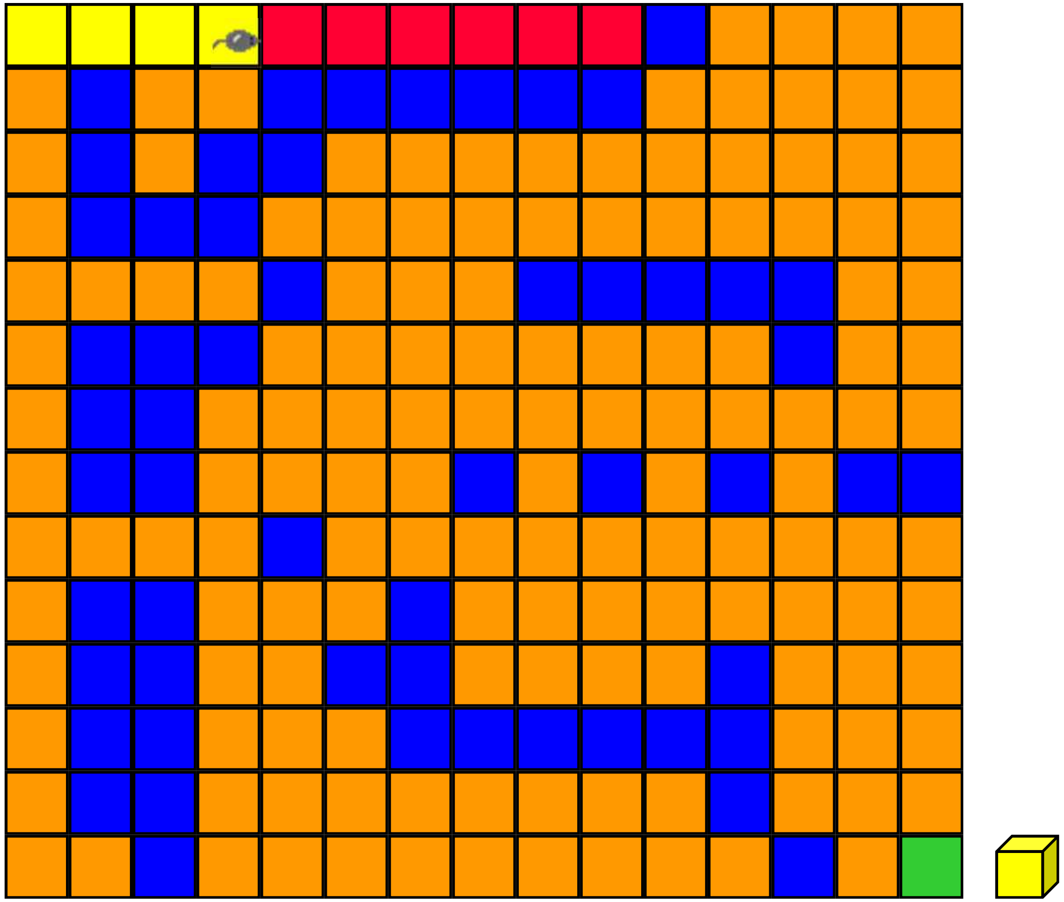
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



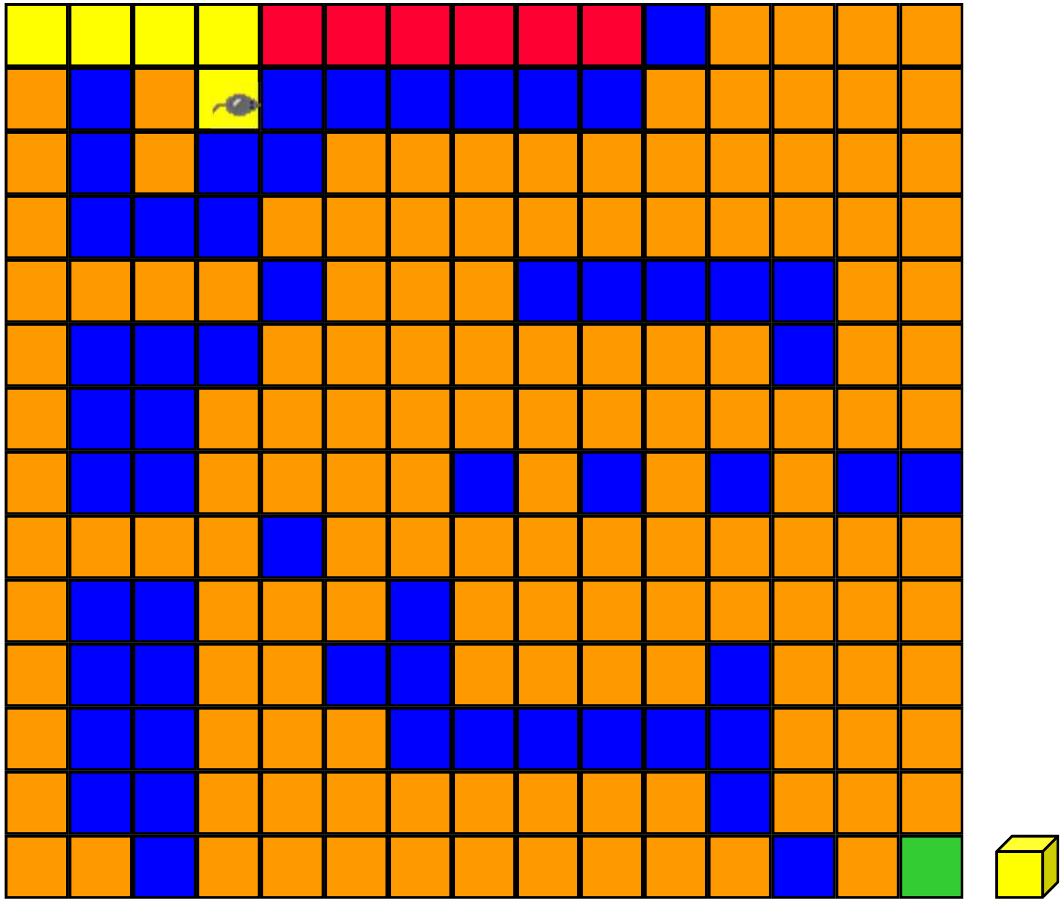
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



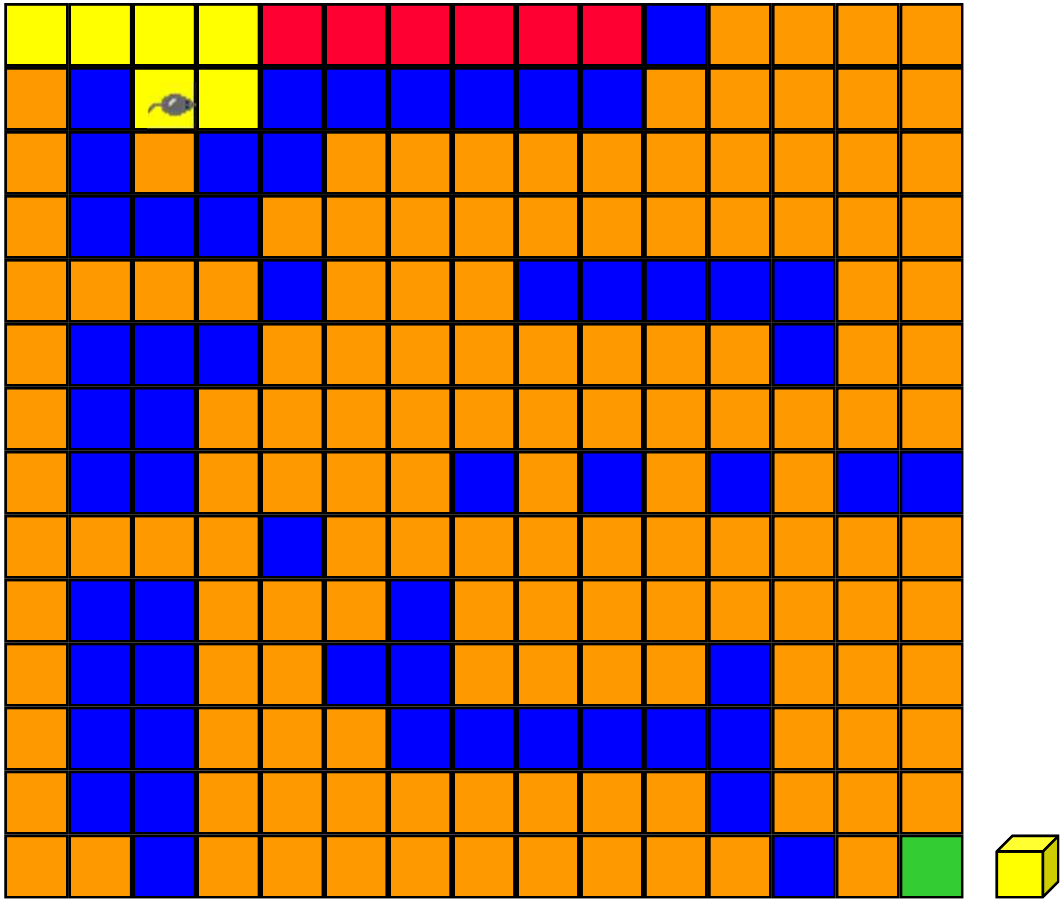
- Move down.

Rat In A Maze



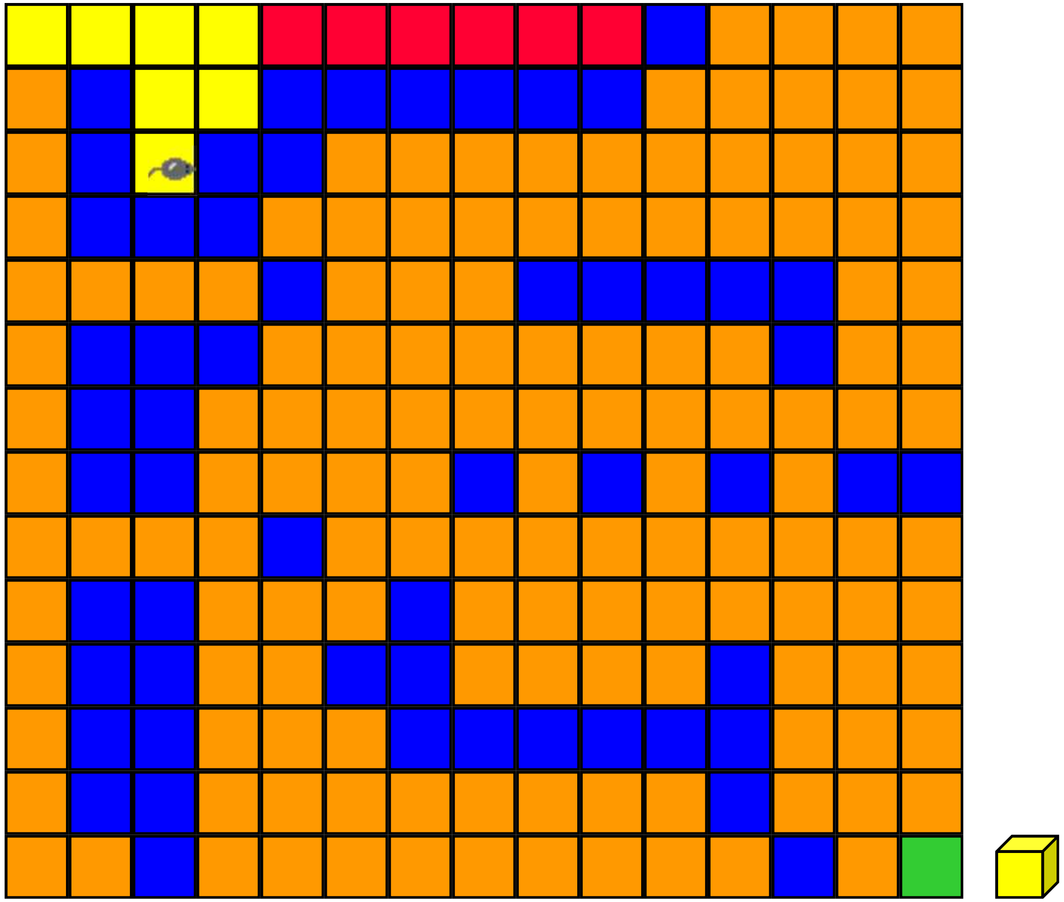
- Move left.

Rat In A Maze



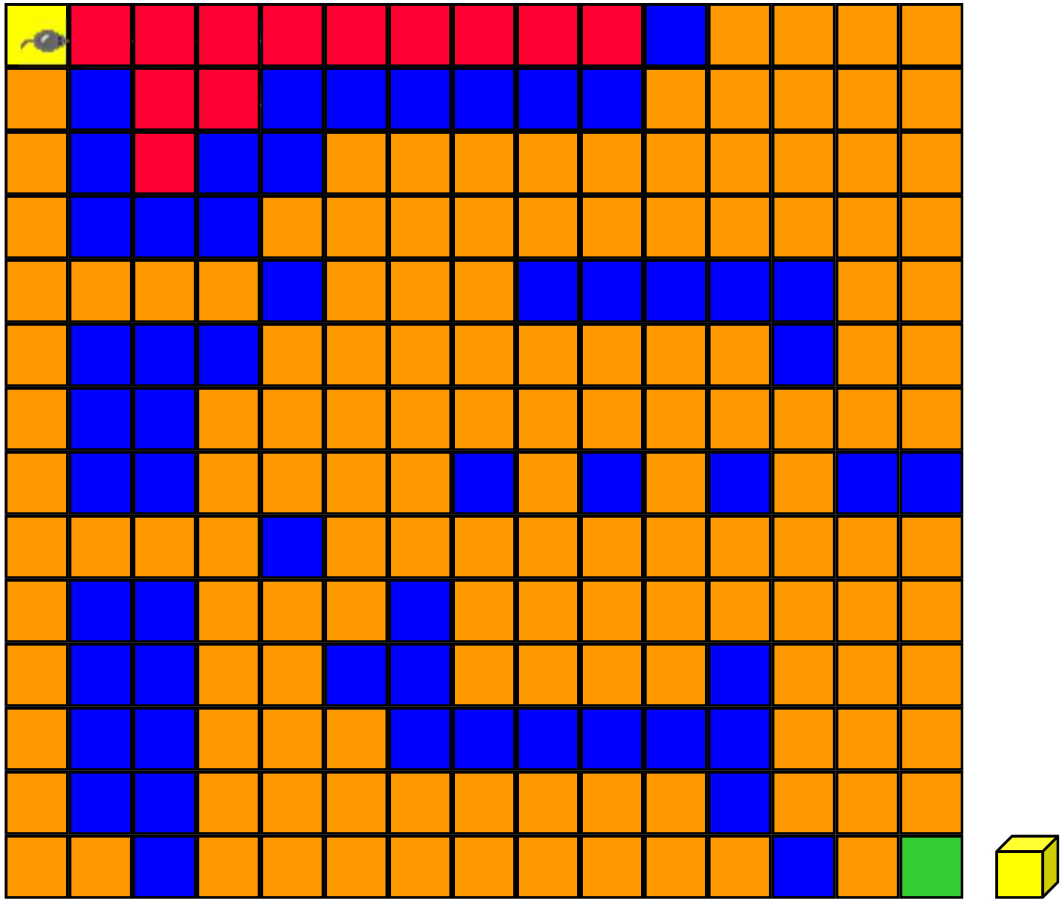
- Move down.

Rat In A Maze



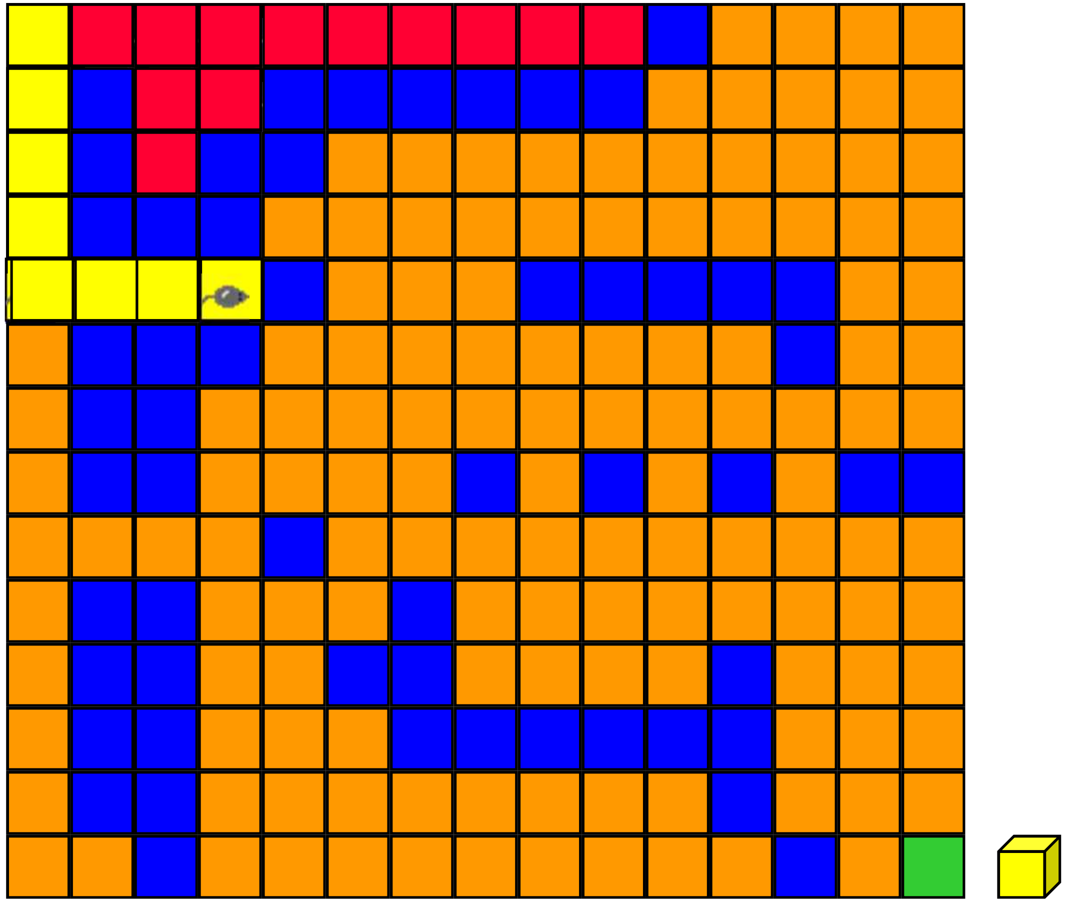
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



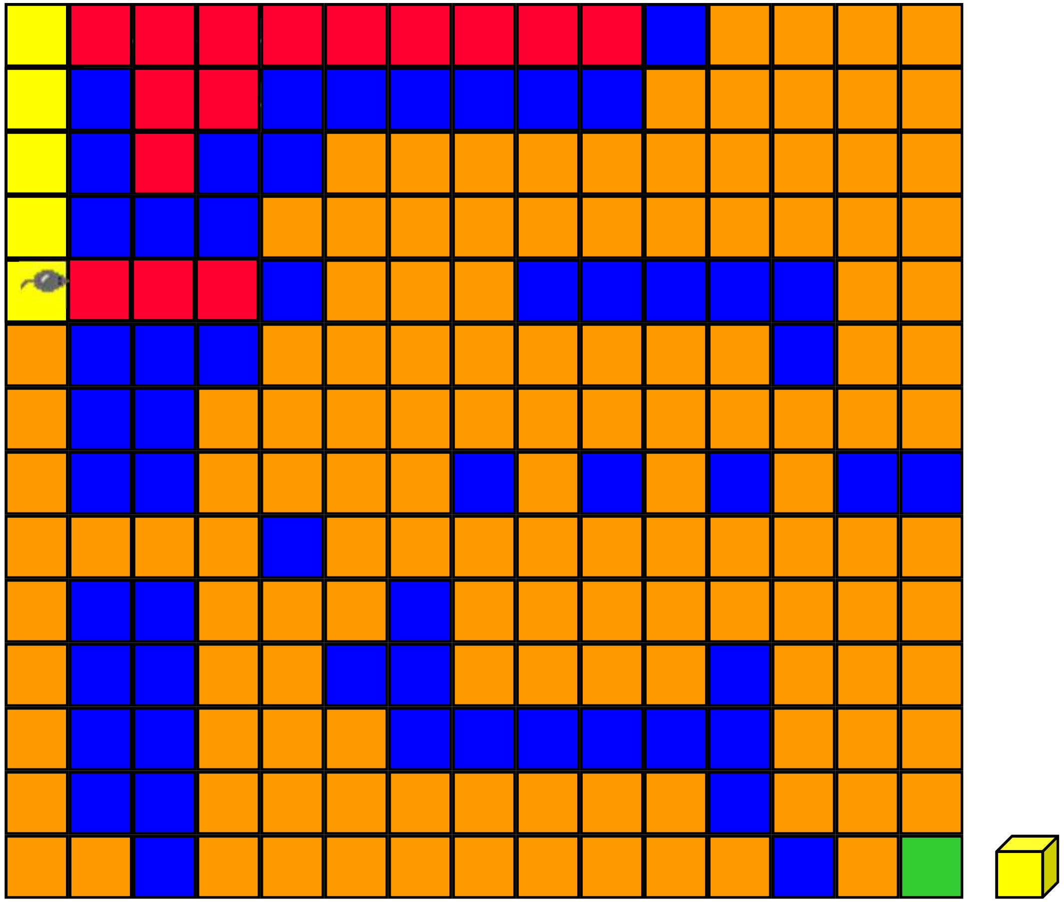
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



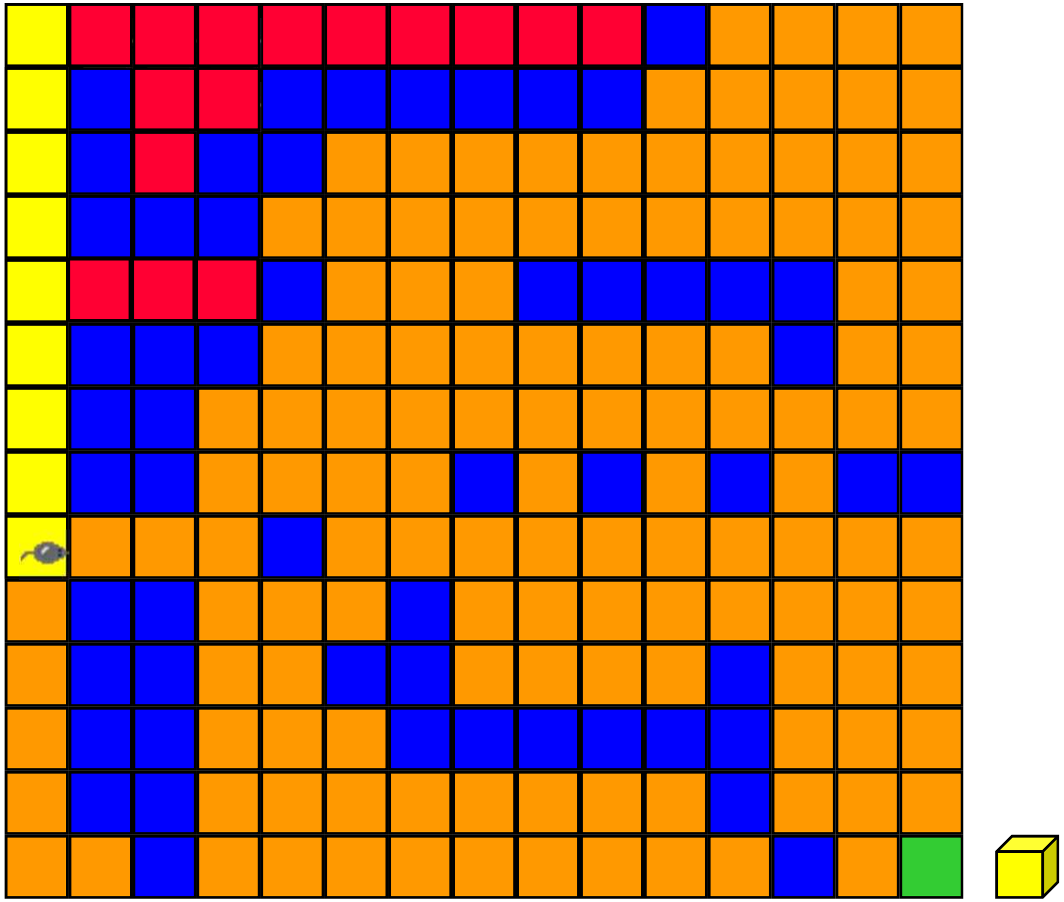
- Move right.
- Backtrack.

Rat In A Maze



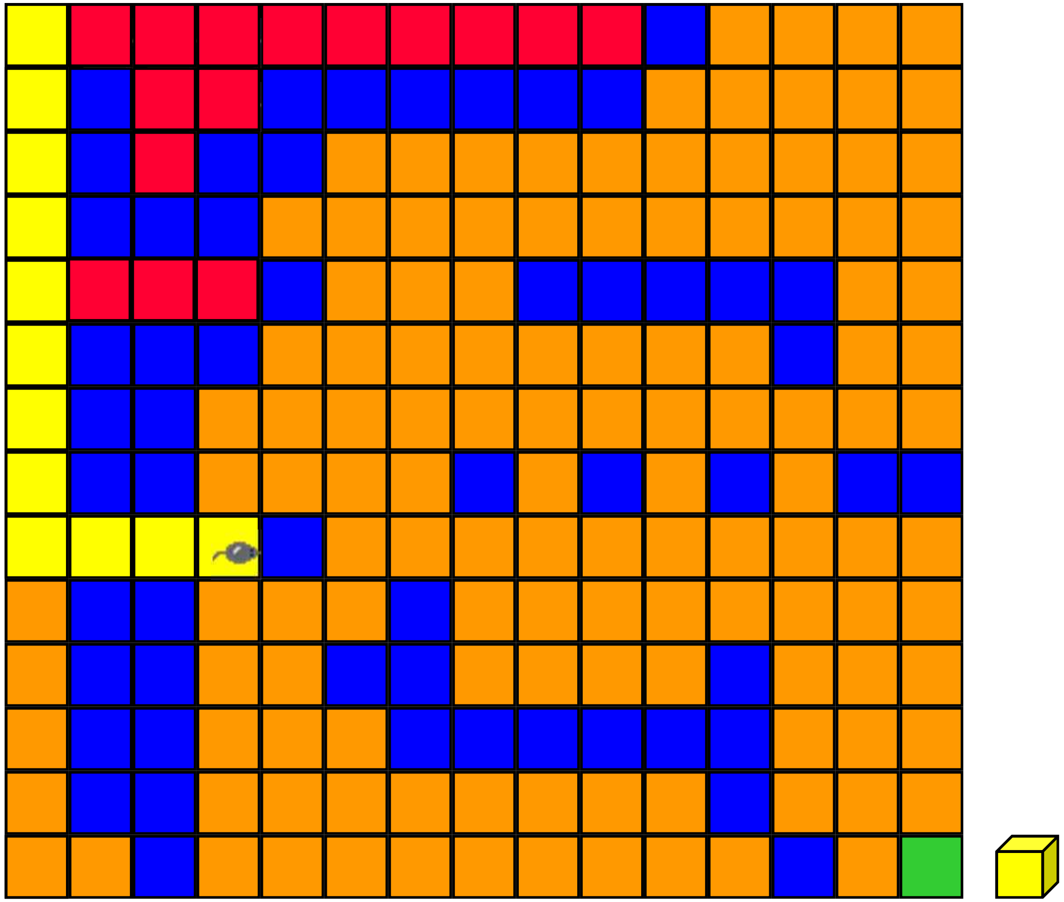
- Move downward.

Rat In A Maze



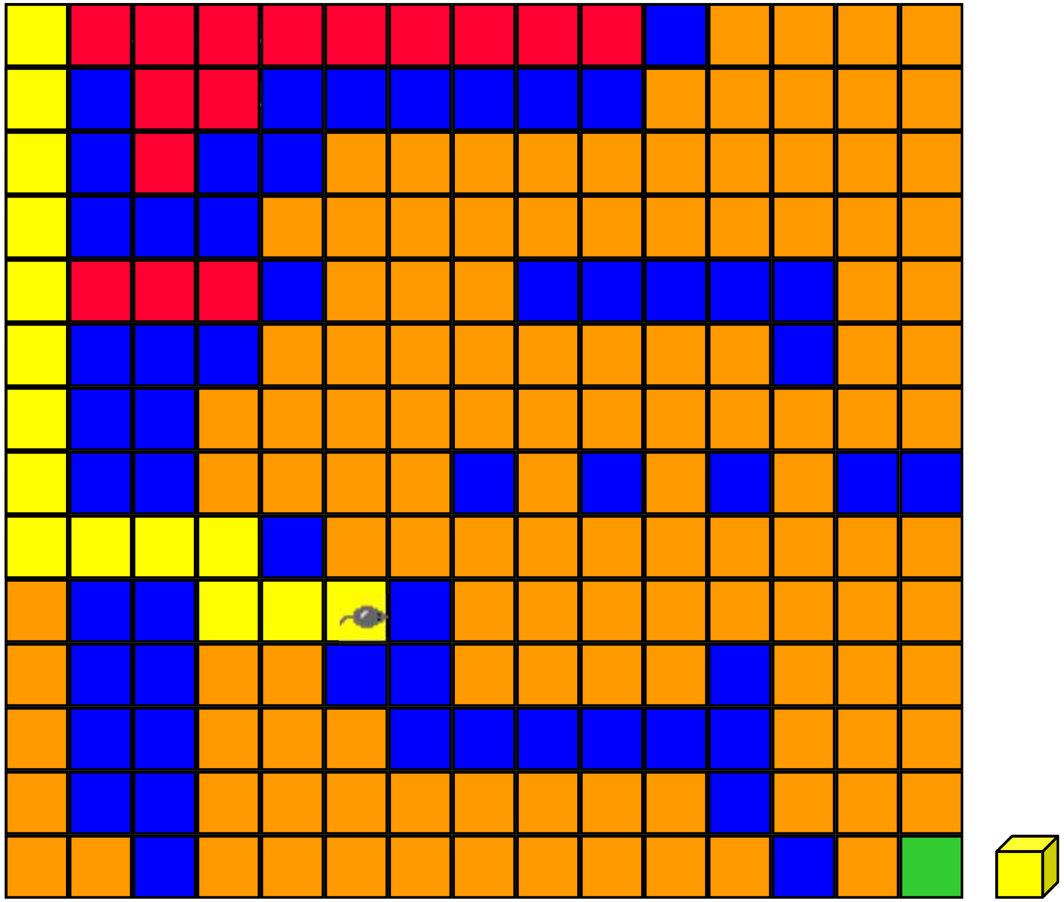
- Move right.

Rat In A Maze



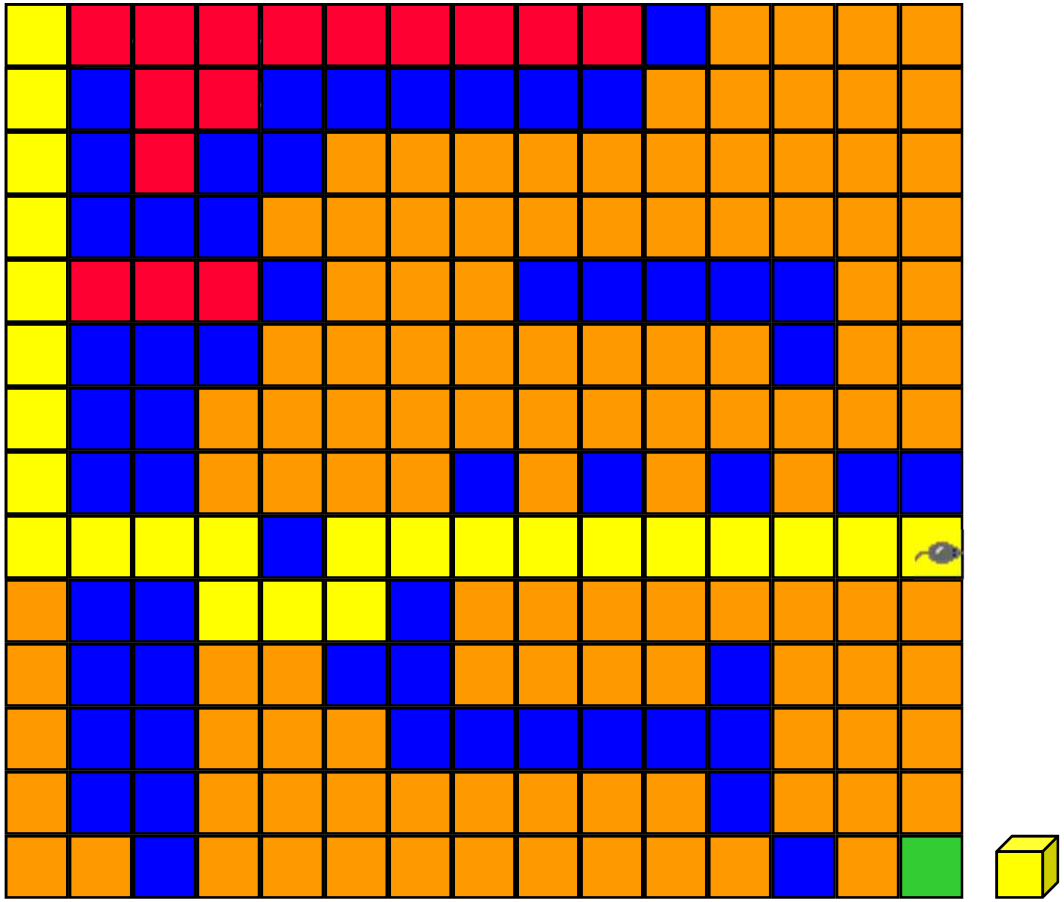
- Move one down and then right.

Rat In A Maze



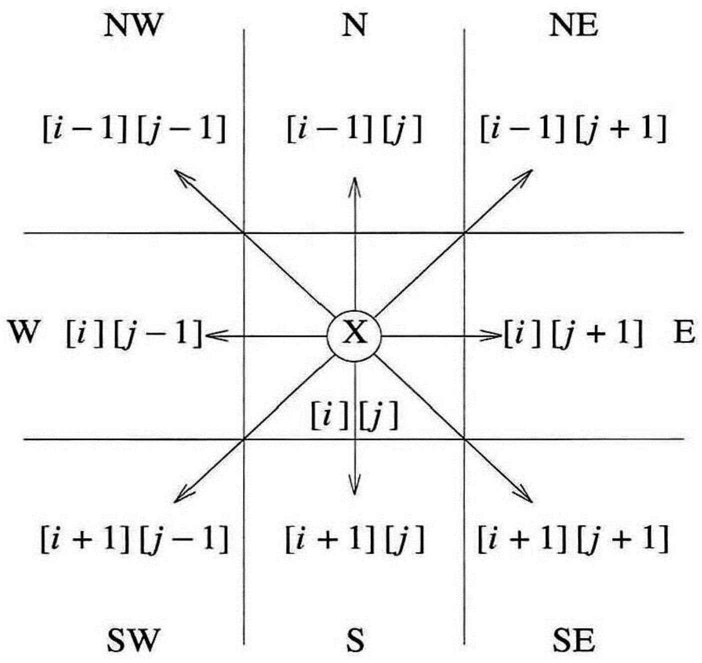
- Move one up and then right.

Rat In A Maze



- Move down to exit and eat cheese.
- Path from maze entry to current position operates as a stack.

Maze problem: allowable 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

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

- Finding the position of the next move, *maze[nextRow][nextCol]*
  - *next_row = row + move[dir].vert;*
  - *next_col = col + move[dir].horiz;*

[5][4]에서 SE쪽

➔ [5+1][4+1]

Maze problem: maze algorithm using stack

```
#define MAX_STACK_SIZE 100
typedef struct {
        short int row;
        short int col;
        short int dir;
      } element;

element stack[MAX_STACK_SIZE];
```

Maze problem의 자료구조

- (1) stack[MAX_SIZE]: row, col, direction
  – 방문한 지점을 저장하되, direction은 다음 ~~방문 위치~~를 나타냄. 찾을 방향
- (2) maze[ROW][COL]:
  – 미로 구성(1, 0): 0 통과 가능, 1은 blocked.
- (3) mark[ROW][COL]:
  – 해당 지점 과거 방문여부 표시

- 변수 row, col: 현재 좌표, dir: 현재 진행방향
- 변수 next_row, next_col: 현재 지점 대비 다음 진행할 좌표

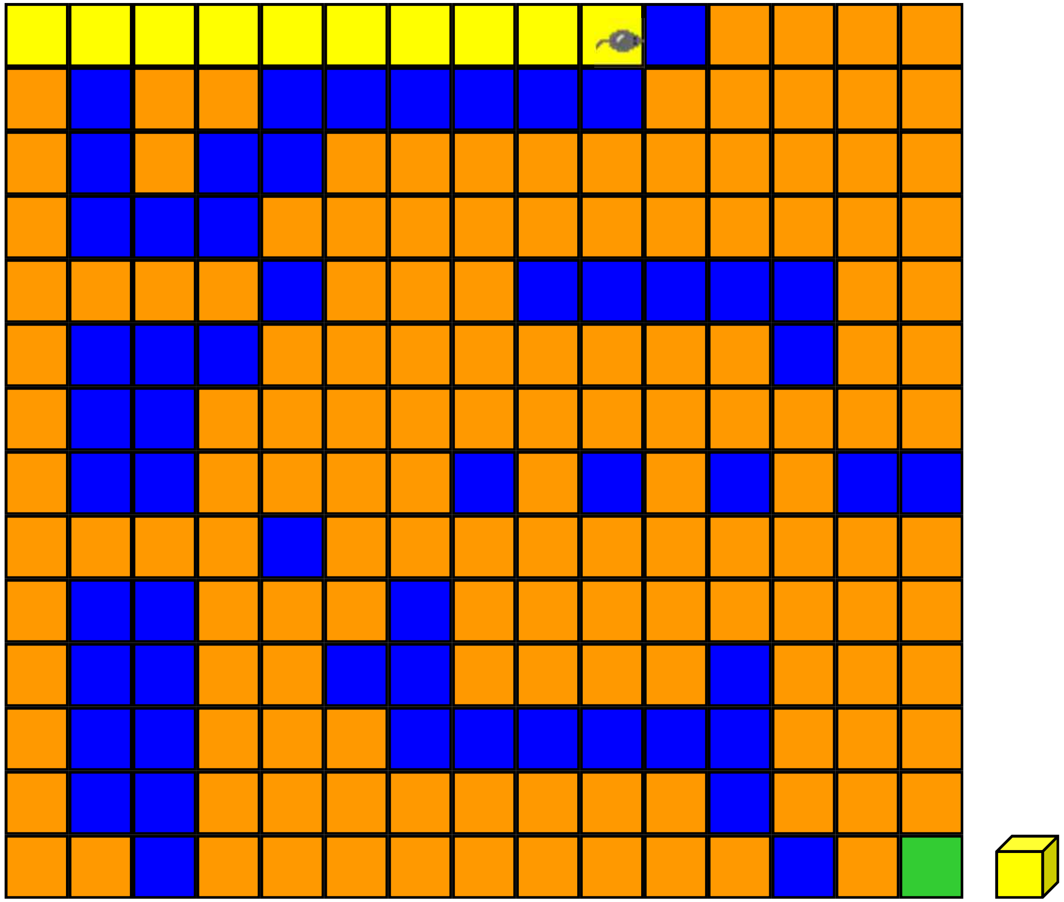
Maze problem: maze algorithm using stack

```
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");
```

Program 3.11: Initial maze algorithm

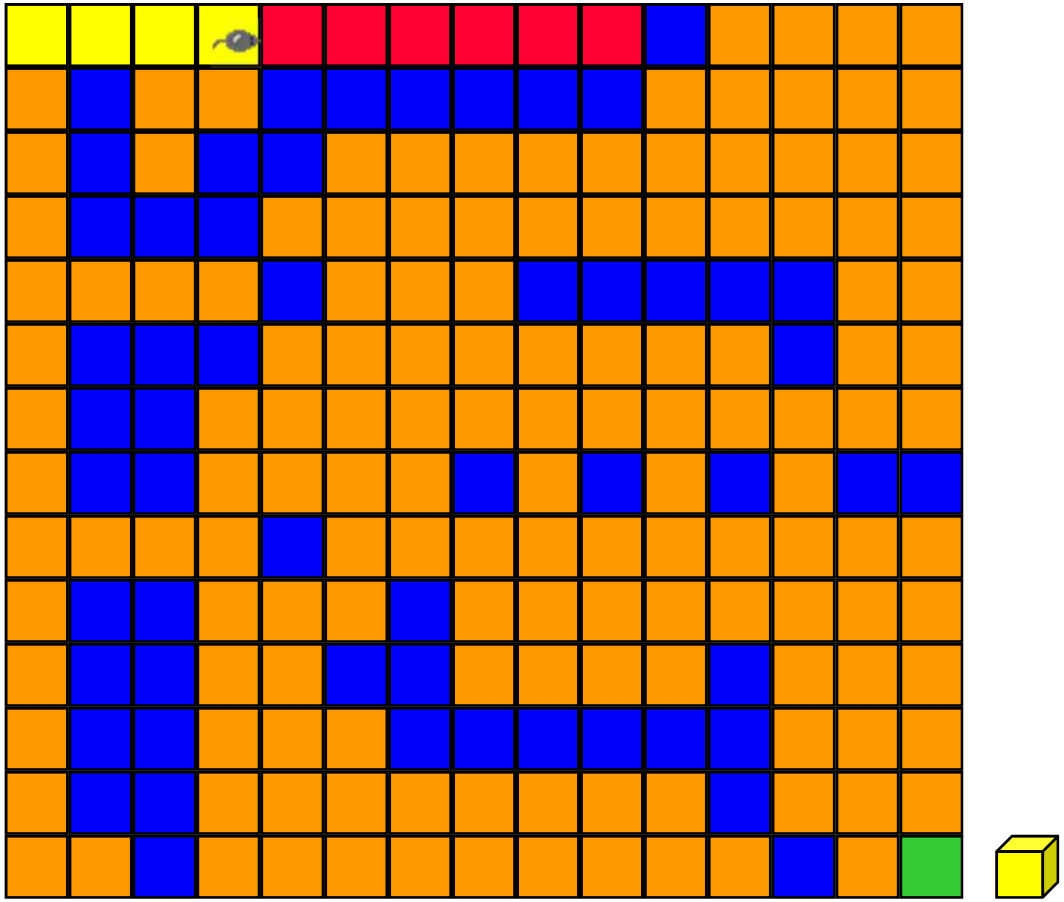
```
while (stack is not empty) {
  /* move to position at top of stack */
  <row,col,dir> = delete from top of stack;
  while (there are more moves from current position) {
     <nextRow, nextCol> = coordinates of next move;
     dir = direction of move;
     if ((nextRow == EXIT_ROW) && (nextCol == 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;
    }
```

Rat In A Maze



- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



- Move down.

Maze problem: maze algorithm using stack

```
void path(void)
{
  /* output a path through the maze if such a path
   exists */
   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) {
           /* move in direction dir*/
           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;   push(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");
   }
```

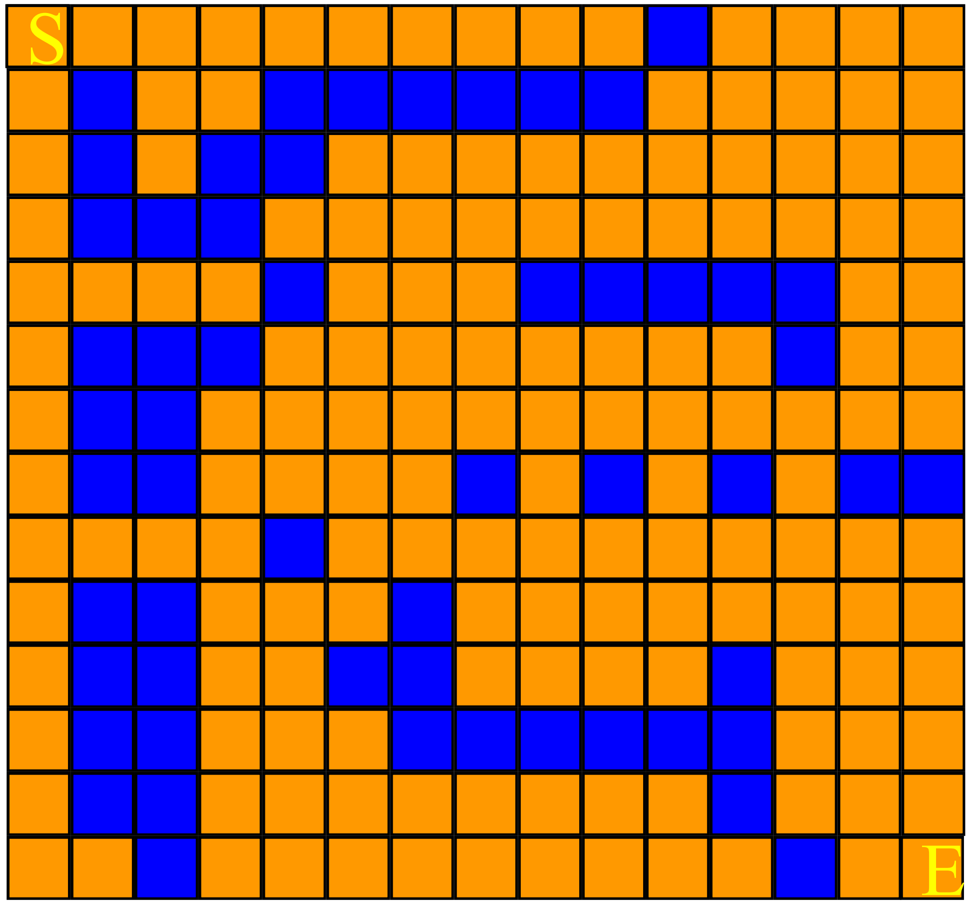
- **analysis of path:**

  each position within the maze is visited no more than once

  worst case complexity : **O(*mp*)**

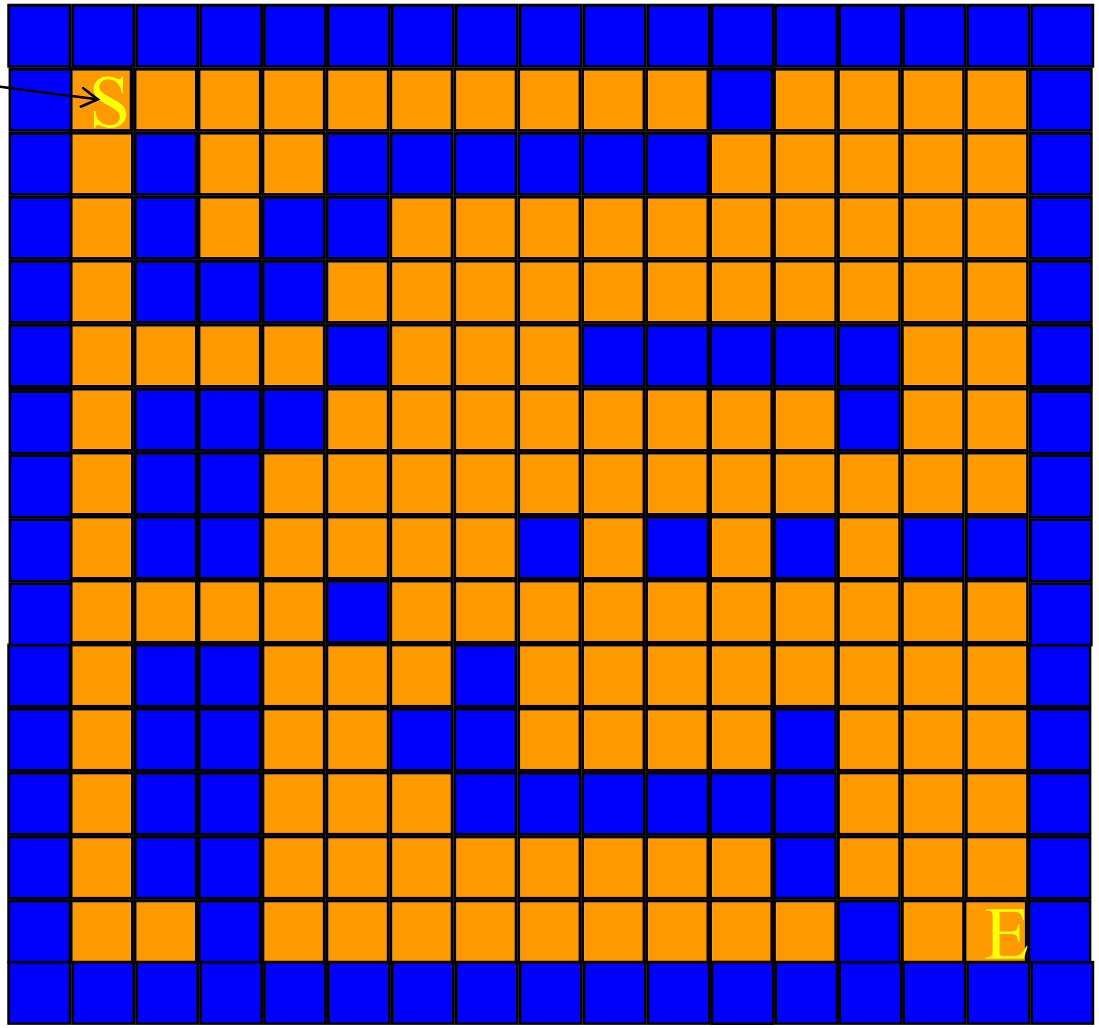
  (m: rowsize, p: colsize)

Rat In A Maze Again



Rat In A Maze Again

[1][1]



not every position has eight neighbors.

```
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;
```

스택에서 하나 pop하여, row/col/dir를 읽어들임.

```
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;
}
```

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

```
}
```

해당 position에서 8방향을 다 찾아봤고, Finish에 도달못했으면 빠져나옴 ➔ (스택에서 다시 position하나를 pop해서 try)

```
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;
}
```

현 position에서 갈 수 있는 다음 위치 (nextrow, nextcol)

다음위치가 진행가능하면➔ 현 위치를 stack에 push하되, 다음번 진행방향을 하나 증가시켜서 저장한다.

다음 위치가 종점이거나, 진행가능하지 않으면, 현재 위치의 방향을 하나 증가시켜서 다시 탐색한다.

```c
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");
```

next_row, next_col이 출구이면, 해당 현재 좌표는
스택에 push되지 않으므로,
현재 좌표(row, col)및 출구(EXIT_ROW, EXIT_COL)는
따로 출력함.

Evaluation of Expressions

- Expressions
  - ((*rear*+1==*front*)||((*rear*==*MAX_QUEUE_SIZE*-1)&& !*front*))
  - operators, operands, parentheses
- Assignment statements
  - x = a/b-c+d*e-a*c
  - a = 4, b=c=2, d=e=3
    - ((4/2)-2)+(3*3)-(4*2) = 1
    - (4/(2-2+3))*(3-4)*2 = -2.66666…
  - x = ((a/b)-c)+(d*e)-(a*c)
  - x = (a/(b-c+d))*(e-a)*c

Precedence rule (1)

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

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Precedence rule (2)

<< >>	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

Evaluating Postfix Expressions

- **Infix notation** 2+3*4
  - binary operator is in-between its two operands
- **Prefix notation** ++top
  - operator appears before its operands
- **Postfix notation** top++
  - Each operator appears after its operands
  - Used by compiler
  - No parantheses
  - To evaluate expression, we make **a single left-to-right scan** of it(no precedence hierarchy)
  - Use stack

Evaluating Postfix Expressions

- Fig 3.13 Infix and postfix notation

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

- Evaluation of postfix expression 6 2/3-4 2*+

Token	Stack [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

Evaluating Postfix Expressions

- Fig 3.13 Infix and postfix notation

피연산자(operand)순서는 안바뀜.

Infix	Postfix
2+3*4	
a*b+5	
(1+2)*7	
a*b/c	
((a/(b-c+d))*(e-a)*c	
a/b-c+d*e-a*c	

Evaluating Postfix Expressions

- Evaluation of postfix expression 6 2/3-4 2*+

Token	Stack			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

- Evaluation of postfix expression

  (operand는 0~9 라고 가정함)

  3 4 2 * + 5 – 3 / 2 1 * + 5 7 % -

Evaluating Postfix Expressions

- Representation of stack and expression

```
#define MAX_STACK_SIZE 100  /* maximum stack size */
#define MAX_EXPR_SIZE 100   /* max size of expression */
typedef enum { lparen, rparen, plus, minus, times, divide,
               mod, eos, operand } precedence;
int stack[MAX_STACK_SIZE];  /* global stack */
char expr[MAX_EXPR_SIZE];   /* input string */
```

- Function to evaluate a postfix expression

```
int eval(void)
{
  precedence token;
  char symbol;
  int op1,op2;
  int n = 0;
     /* counter for the expression string */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos) {
    if (token == operand)
      push(symbol-'0'); / * stack insert */
    else {
  /* remove two operands, perform operation,
     and return 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 = get_token(&symbol, &n);
  }
  return pop(); /* return result */
```

eos: end of string

```
#define MAX_STACK_SIZE 100  /* maximum stack size */
#define MAX_EXPR_SIZE 100   /* max size of expression */
typedef enum { lparen, rparen, plus, minus, times, divide,
               mod, eos, operand } precedence;
int stack[MAX_STACK_SIZE];  /* global stack */
char expr[MAX_EXPR_SIZE];   /* input string */
```

```c
int eval(void)
{/* evaluate a postfix expression, expr, maintained as a
    global variable. '\0' is the the end of the expression.
    The stack and top of the stack are global variables.
    getToken is used to return the token type and
    the character symbol. Operands are assumed to be single
    character digits */
   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 */
}
```

n: 입력 string의 인덱스

char expr[ ]

0	1	2	3	4	5	6	7	8	9
‘6’	‘2’	‘/’	‘3’	‘-’	‘4’	‘2’	‘*’	‘+’	‘\0’

'3' - '0' =?

```
char symbol;
int op1, op2;
int n = 0; /* counter for the
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);
}
```

Evaluating Postfix Expressions

- Function to get a token from the input string

char expr[ ]

0	1	2	3	4	5	6	7	8	9
'6'	'2'	'/'	'3'	'-'	'4'	'2'	'*'	'+'	'\0'

```
precedence get_token (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 '\0' : return eos;
           default  : return operand; /* no error checking,default is operand */
  }
}
```

int eval()

```
char symbol;
int op1, op2;
int n = 0; /* counter for the
int top = -1;
token = getToken(&symbol, &n);
```

Evaluating Postfix Expressions: examples

1. Simple expression

Input : a+b*c

Output : abc*+

Translation of a+b*c to postfix

2. Parenthesized expression

Input: a*(b+c)*d

Output: abc+*d*

Translation of a*(b+c)*d to postfix

(비교) a*b+c*d

Evaluating Postfix Expressions: postfix conversion

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

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

```
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=get_token(&symbol,&n); token!=eos;
                 token=get_token(&symbol,&n))
   {
         if (token == operand)
            printf("%c", symbol);
         else if (token == rparen)
         {
  /* unstuck tokens until left parenthesis */
     while (stack[top] != lparen)
        print_token(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])
        print_token(pop());
     push(token);
   }
  }
  while ((token=pop()) != eos)
        print_token(token);
  printf("\n");
}
```

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

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays – index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */

static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

operand는 출력하고,
operator는 우선순위를
고려하여 출력하거나
스택에 push함

```
precedence stack[MAX_STACK_SIZE];
                  /* (   )   +   -   *   /  %  eos */
static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

- a+b*c

- a*b+c

stack의 top에 있는
operator보다 우선순위가
높은 operator가 오면 push,
그렇지 않으면 top을
pop해서 출력.

```
precedence stack[MAX_STACK_SIZE];
                /* (   )   +   -   *   /   %  eos */
static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

- a+b*c

- (a+b)*c

왼쪽괄호는 무조건 push해야함
➔ icp의 왼쪽괄호 우선순위가 가장 높음.

왼쪽괄호는 일단 stack에 push된 이후에는 오른쪽괄호가 올 때 까지 하는 일이 없음➔ isp 우선순위가 0임.

```
precedence stack[MAX_STACK_SIZE];
                    /* (  )  +  -  *  /  %  eos */
static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

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

- ((a+b)%(c*d))/(e+f/g)

```
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");
 }
```

Input: a*(b+c)*d
Output: abc+*d*

```
char symbol;
precedence token;
int n = 0;
int top = 0;    /*
stack[0] = eos;
```

```
precedence stack[MAX_STACK_SIZE];
                /* (  )  +  -  *  /  %  eos */
static int isp[] = { 0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

0	1	2	3	4	5	6	7	8	9
a	*	(	b	+	c	)	*	d	'\0'

```
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);
```

stack에 남은 operator출력

```
char symbol;
precedence token;
int n = 0;
int top = 0;    /*
stack[0] = eos;
```

precedence stack[MAX_STACK_SIZE];

```
                  /* (  )  +  -  *  /  %  eos */
static int isp[] = { 0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

Input: ((((1/ 2) - 3) + (4 * 5)) - 6 * 7)
Output:

```
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);
```

```c
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=get_token(&symbol,&n); token!=eos;
            token=get_token(&symbol,&n))
    {
        if (token == operand)
            printf("%c", symbol);
        else if (token == rparen)
        {
        /* unstuck tokens until left parenthesis */
            while (stack[top] != lparen)
                print_token(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])
                print_token(pop());
            push(token);
        }
    }
    while ((token=pop()) != eos)
        print_token(token);
    printf("\n");
}
```

Input: a*(b+c)*d
Output: abc+*d*

```c
precedence stack[MAX_STACK_SIZE];
                   /* (  )  +   -   *   /  %  eos */
static int isp[] = {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

top=0

Evaluating Postfix Expressions: analysis

- n : number of tokens in the expression
- extracting tokens and outputting them : $\theta(n)$
- in two while loop, the number of tokens that get stacked and unstacked is linear in n : $\theta(n)$

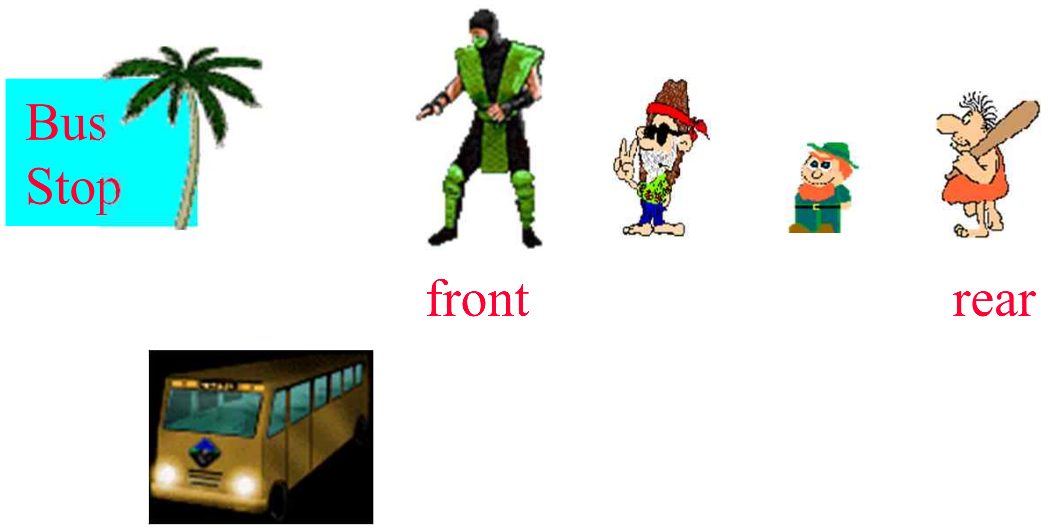
total time complexity : $\theta(n)$

Queues

- Ordered linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

Bus Stop Queue

Example: Bus Stop Queue



Bus Stop Queue

front rear

Bus Stop Queue

Queue Operations

- IsFullQ … return true iff queue is full
- IsEmptyQ … return true iff queue is empty
- AddQ … add an element at the **rear** of the queue
- DeleteQ … delete and return the **front** element of the queue

ADT of Queues

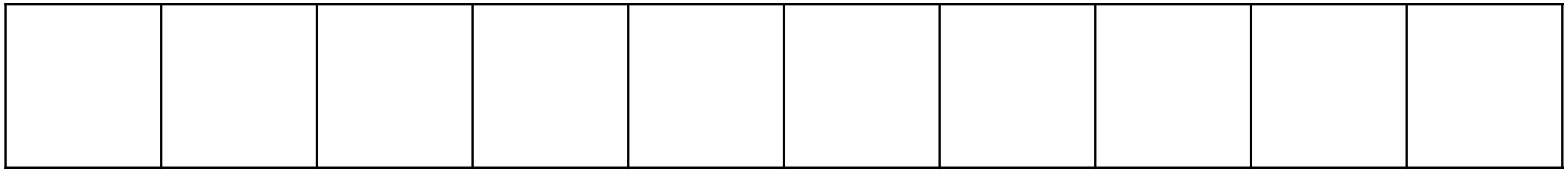
ADT *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer
    Queue CreateQ(*maxQueueSize*) ::=
        create an empty queue whose maximum size is *maxQueueSize*
    Boolean IsFullQ(*queue*, *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*

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.

array로 queue 구현



(a 1) (a 2) (a 3) d d (a 4) (a 5) d d d d

- front: -1
- rear: -1
- addq ➔ `queue[++rear] = item;`
- deleteq ➔ `return queue[++front];`

Creation of Queue in C

```
Queue CreateQ(max_queue_size) ::=
    #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
```

- Use a 1D array to represent a queue.
- Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

Queue in an Array

a	b	c	d	e										

0 1 2 3 4 5 6

- DeleteQ() => delete queue[0]
  - O(1) time
- AddQ(x) => if there is capacity, add at right end
  - O(1) time

Implementation of Queue operations in C

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

Program 3.5: Add to a queue

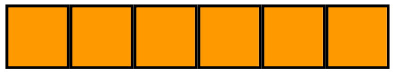
```
element deleteq()
{/* remove element at the front of the queue */
   if (front == rear)
      return queueEmpty(); /* return an error key */
   return queue[++front];
}
```

Program 3.6: Delete from a queue

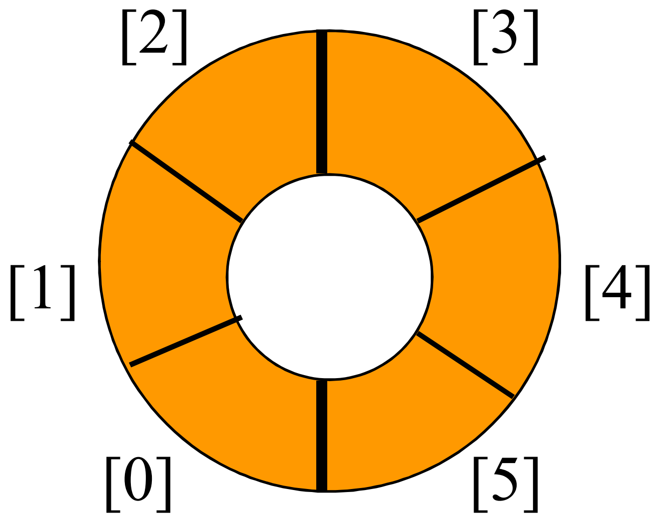
More efficient queue representation: Circular Queue

- Use a 1D array queue.

queue[]

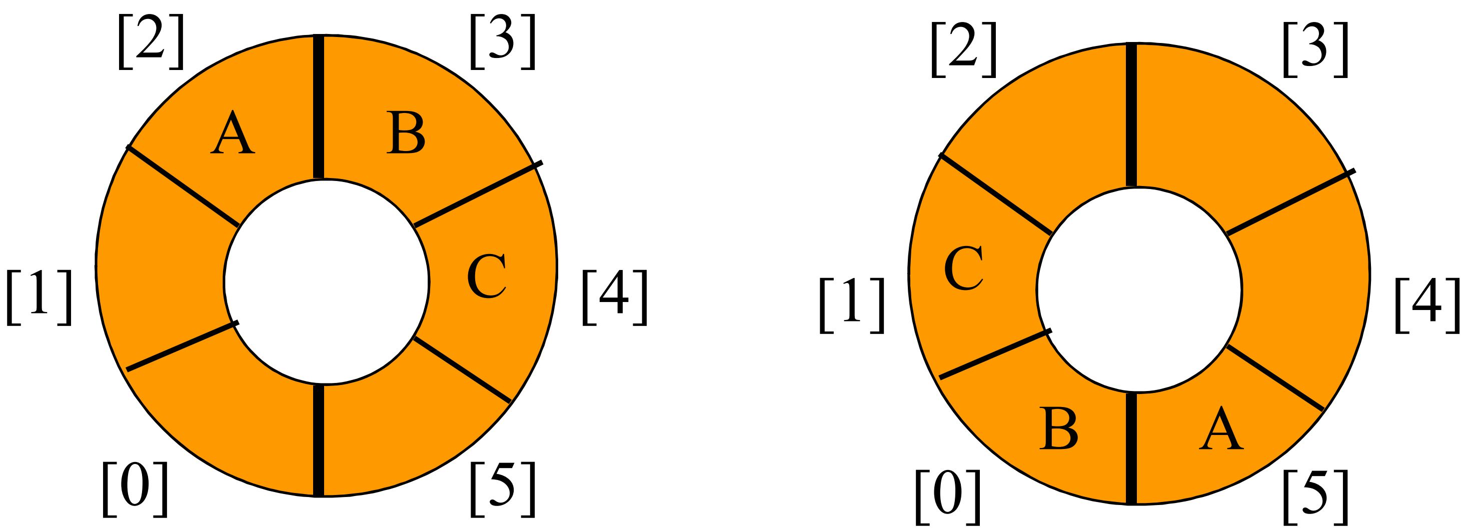


- Circular view of array.



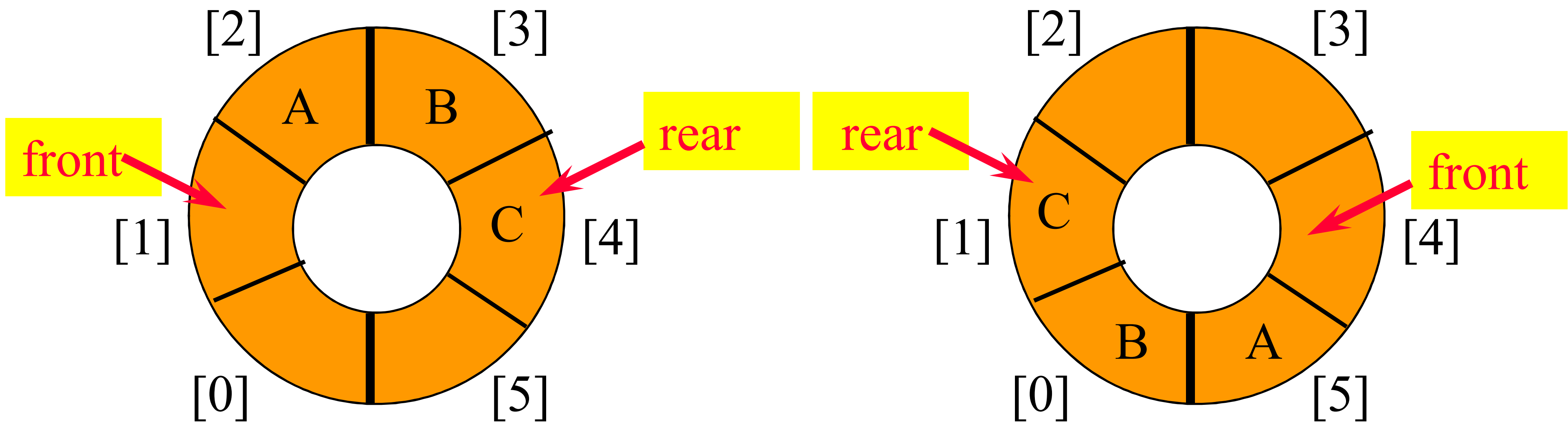
More efficient queue representation: Circular Queue

- Possible configurations with 3 elements.

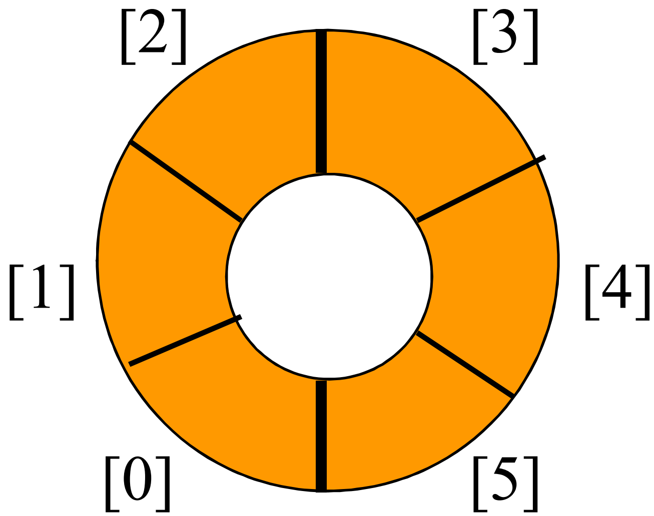


More efficient queue representation: Circular Queue

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



Circular Queue를 Array로 구현

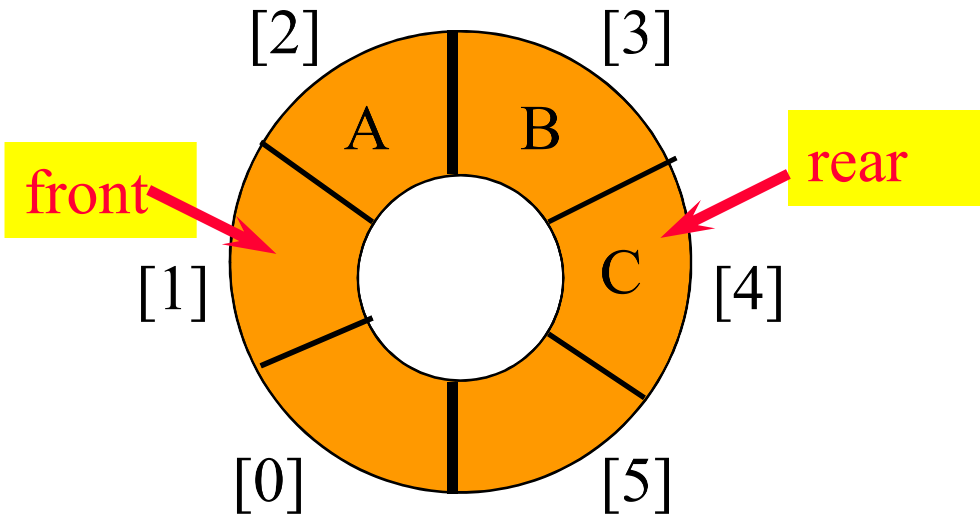


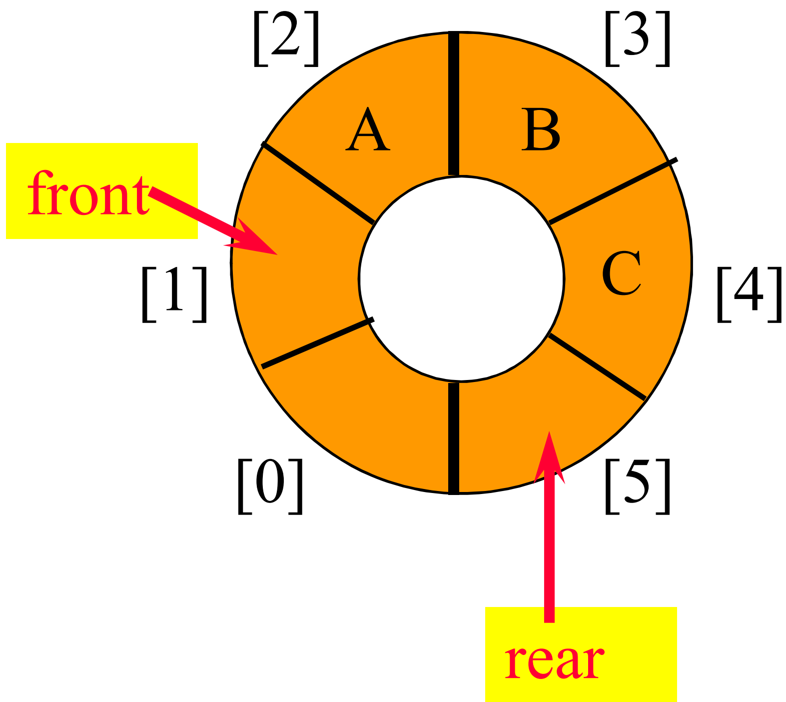
(a 1) (a 2) (a 3) (a 4) (a 5) d d (a 6) (a 7) d d d d d

- front: 0
- rear: 0

More efficient queue representation: Circular Queue

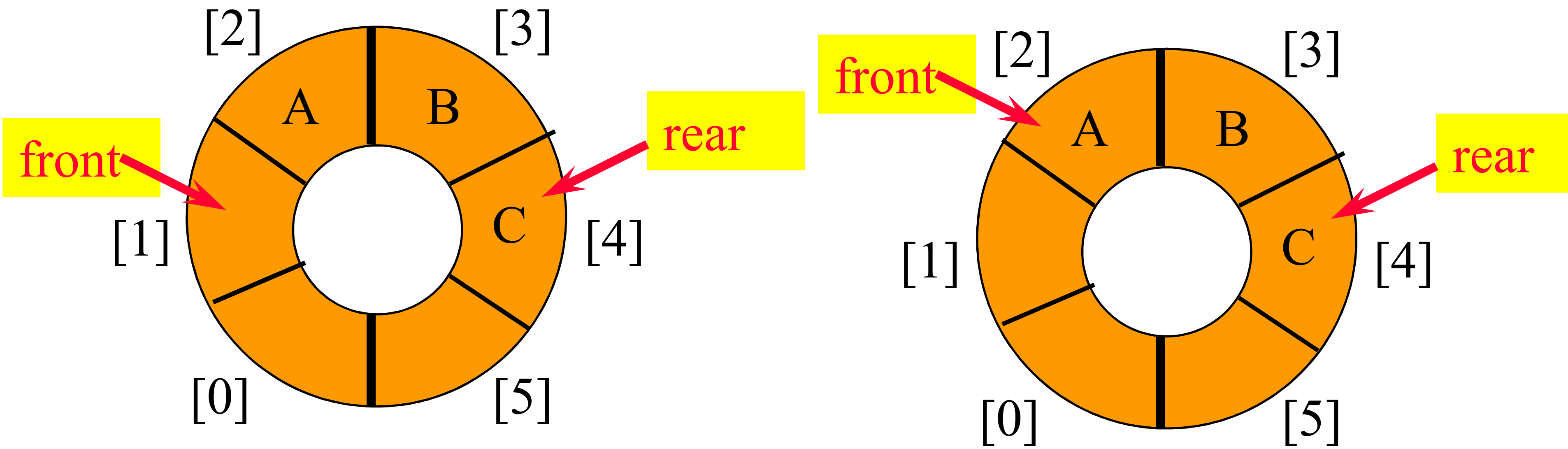
- Adding an element in the circular queue.
  - Move rear one clockwise.
  - Then put into queue[rear].





More efficient queue representation: Circular Queue

- Delete an element from the circular queue.
  - Move front one clockwise.
  - Then extract from queue[front].



Operations for circular queue

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

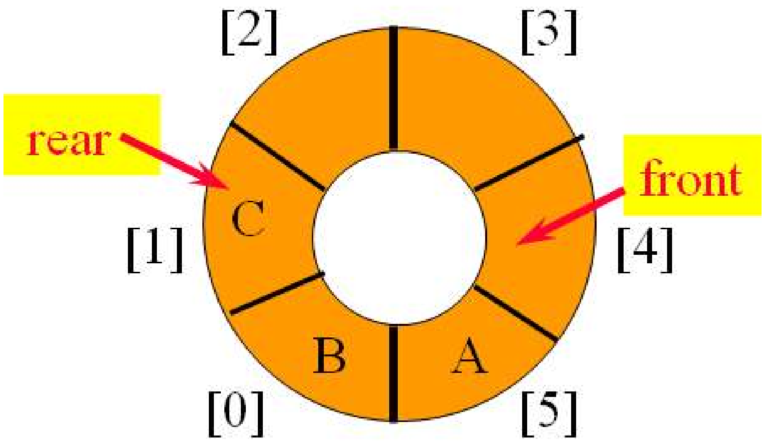
```
if(rear==MAX_QUEUE_SIZE)
    rear=0;
else
    rear++;
```

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

※가장 앞 원소는 front보다 하나 앞에, 가장 뒤 원소는 rear 위치에 있음.

→ queue가 비어있는 상태를 제외하고는 front==rear인 경우가 없음.

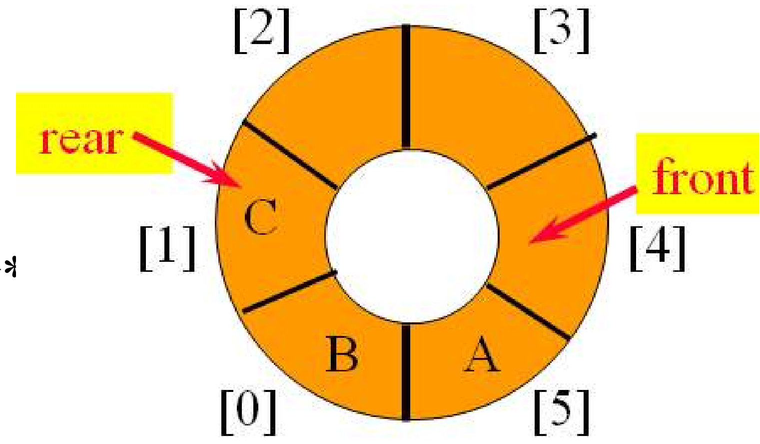
```
void addq(element item)
{ /* add an item to the queue */
    rear = (rear+1) % MAX-QUEUE-SIZE;
    if (front == rear)
        queueFull(); /*print error and exit*/
    queue[rear] =item;
}
```




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

Program 3.5: Add to a queue

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




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

Program 3.6: Delete from a queue