Chap 3. Stacks and Queues

3.1 Stacks

- A ***stack*** is an ordered list in which insertions (also called ***push***es and adds) and deletions (also called ***pop***s and removes) are made at one end called the ***top***.

- Given a stack $S = (a_0, \cdot \cdot \cdot, a_{n-1})$, we say that a_0 is the ***bottom element***, a_{n-1} is the ***top element***, and a_i is on top of element a_{i-1}, $0 < i < n$.

- The restrictions on the stack imply that if we add the elements A, B, C, D, E to the stack, in that order, then E is the first element we delete from the stack.

- Since the last element inserted into a stack is the first element removed, a stack is also known as a ***Last-In-First-Out (LIFO)*** list.

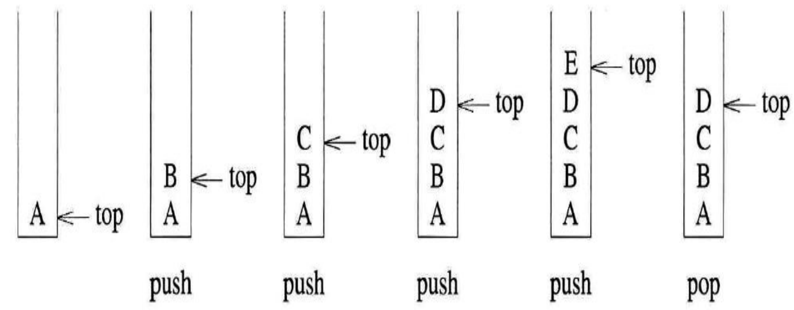


Figure 3.1: Inserting and deleting elements in a stack

ADT *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all $stack \in Stack$, $item \in element$, $maxStackSize \in$ 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*

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;
}
```

Program 3.1: Add an item to a stack

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

Program 3.2: Delete from a stack

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

Program 3.3: Stack full

Stack
element stack[MAX_STACK_SIZE];
top : top element의 index.
초기화: top = -1;
empty 조건: top == -1
Full 조건: top == MAX_STACK_SIZE-1

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

Example 3.1 [***System stack***]: The ***system stack*** is used by a program at run-time to process function calls. Whenever a function is invoked, the program creates a structure, referred to as an ***activation record*** or a ***stack frame***, and places it on top of the system stack.

Initially, the activation record for the invoked function contains only **a pointer to the previous stack frame** and ***a return address*** (and **the arguments (parameter values) passed to the routine (if any)**). The previous stack frame pointer points to the stack frame of the invoking function, while the return address contains the location of the statement to be executed after the function terminates. Since only one function executes at any given time, the function whose stack frame is on top of the system stack is chosen.

If this function invokes another function, **the local variables**, except those declared *static*, and **the parameters of the invoking function** are added to its stack frame. A new stack frame is then created for the invoked function and placed on top of the system stack. When this function terminates, its stack frame is removed and the processing of the invoking function, which is again on top of the stack, continues.

일반적으로 **System stack**이라 하지 않고 **Call stack** 이라 한다.

Assume that we have a main function that invokes function a_1. Figure 3.2(a) shows the system stack before a_1 is invoked; Figure 3.2(b) shows the system stack after a_1 has been invoked. **Frame pointer** *fp* is a pointer to the current stack frame. The system also maintains separately **a stack pointer**, *sp*, which we have not illustrated.

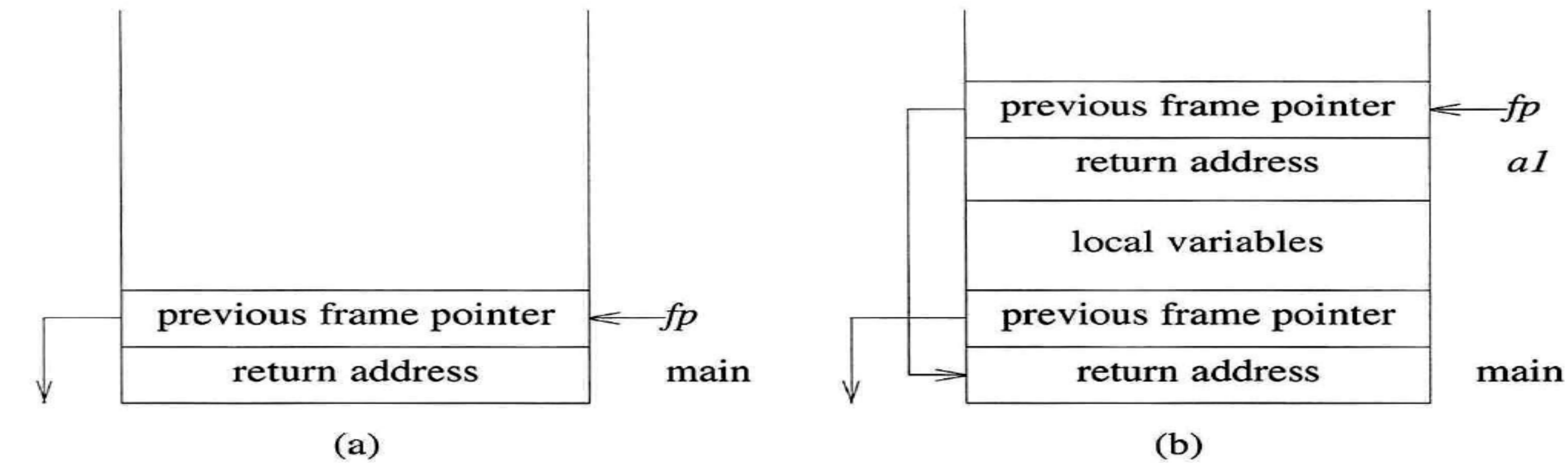


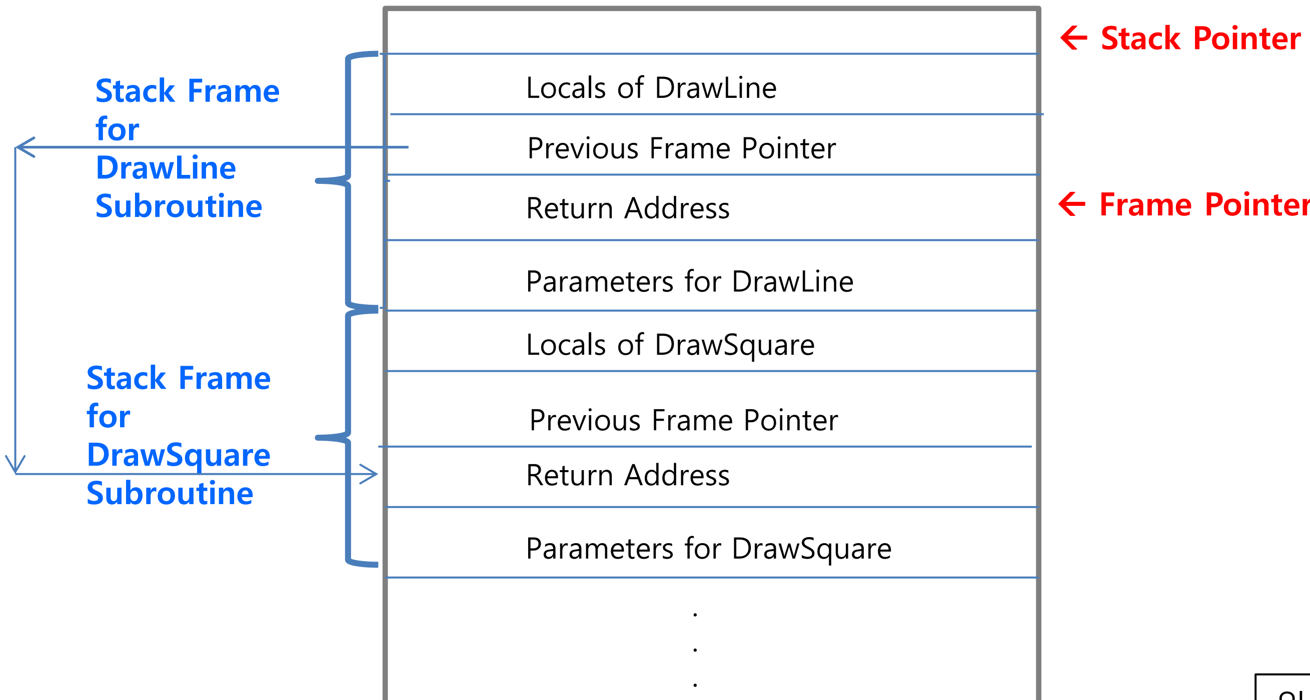
Figure 3.2: System stack after function call

Since all functions are stored similarly in the system stack, it makes no difference if the invoking function calls itself. That is, a recursive call requires no special strategy; the run-time program simply creates a new stack frame for each recursive call.

```
main()
{
 …
 DrawSquare(a, b, c);
 …
}
```

```
DrawSquare(int a, int b, int c)
{
   int s, t;
   …
   DrawLine(w, x, y, z);
   …
}
```

```
DrawLine(int w, int x, int y, int z)
{
   int m, n;
   …
}
```



call stack layout

일반적으로 **System stack**이라 하지 않고 **Call stack** 이라 한다.

Stack pointer : points to the top of the Call Stack.
Frame pointer == Stack Base Pointer : points to the return address.

3.2 Stacks Using Dynamic Arrays

The following implementation of CreateS, IsEmpty, and IsFull uses **a dynamically allocated array stack** whose **initial capacity** (i.e., maximum number of stack elements that may be stored in the array) is 1. Specific applications may dictate other choices for the initial capacity.

***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;

Array doubling: we double array capacity whenever it becomes necessary to increase the capacity of an array.

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

Program 3.4: Stack full with array doubling

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

- Complexity of Array Doubling

  Let final value of capacity be 2^k for some k, k>0.

  Number of pushes is at least $2^{k-1}+ 1$.

  The total time spent on array doubling is

  $O(\sum_{i=1}^{k} 2^i) = O(2^{k+1}) = O(2^k)$.

  So, although the time for an individual push is O(*capacity*), the time for all *n* pushes remains O(*n*).

3.3 Queues

- A ***queue*** is an ordered list in which insertions (also called ***addition***s, puts, and pushes) and ***deletion***s (also called removals and pops) take place at different ends.

- The end at which new elements are added is called the ***rear***, and that from which old elements are deleted is called the ***front***.

- The restrictions on a queue imply that if we insert A, B, C, D, and E in that order, then A is the first element deleted from the queue.

- Since the first element inserted into a queue is the first element removed, queues are also known as ***First-In-First-Out (FIFO)*** **lists**.

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*

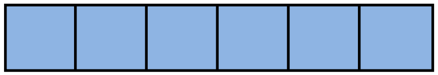
Element DeleteQ(*queue*) ::=
if (IsEmptyQ(*queue*)) **return**
else remove and return the *item* at front of queue.

ADT 3.2: Abstract data type *Queue*

Queue Representation

- Use a 1D array *queue*

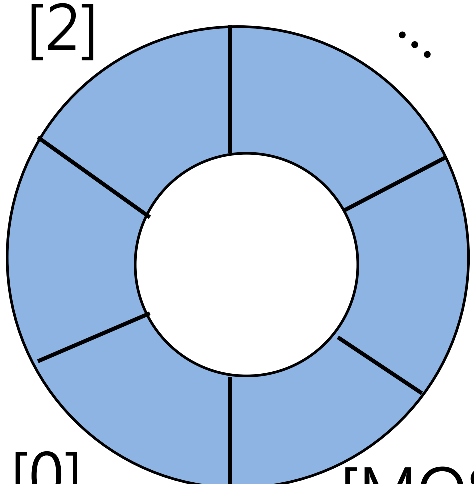
queue[]



Queue
front : 가장 앞 **element** 바로 앞을 가리키고,
rear : 가장 뒤 **element**를 가리킨다.
초기화: **front = rear = -1;**
empty 조건: **front == rear**
Full 조건: 삽입시 **rear == MAX_QUEUE_SIZE-1**

```
#define MAX_QUEUE_SIZE 100
typedef struct {
        int key;
        /* other fields */
} element;
element queue[MAX_QUEUE_SIZE];
```

- Circular Queue



Circular Queue
front : 가장 앞 **element** 바로 앞을 가리키고,
rear : 가장 뒤 **element**를 가리킨다.
초기화: **front = rear = 0**
empty 조건: **front == rear**
full 조건: 삽입시 **(rear+l) % MAX_QUEUE_SIZE == front;**

아래 방법은
front : 가장 앞 element를 가리키고,
rear : 가장 뒤 element를 가리킨다.
초기화: front = rear = -1;
empty 조건: rear < front
Full 조건: 삽입시 rear == MAX_QUEUE_SIZE-1

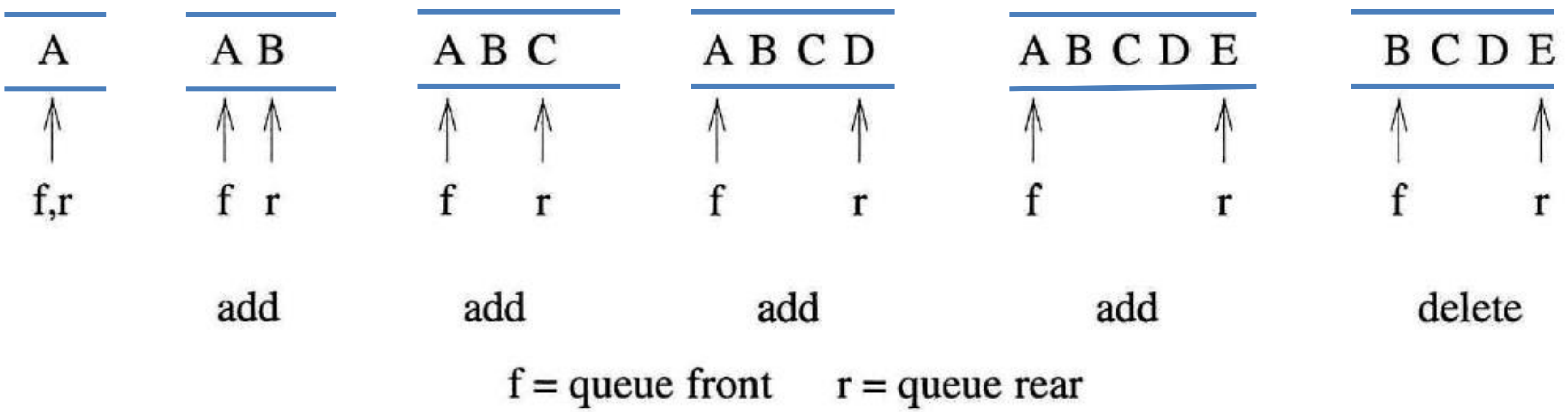


Figure 3.4: Inserting and deleting elements in a queue

위 방법은 참고로만 이해하고 사용하지 말 것.

Bus Stop Queue

Queue
front : 가장 앞 element 바로 앞을 가리키고,
rear : 가장 뒤 element를 가리킨다.
초기화: front = rear = -1;
empty 조건: front == rear
Full 조건: 삽입시 rear == MAX_QUEUE_SIZE-1

Sequential representation: using *1D array*

```
Queue CreateQ(maxQueueSize) ::=
#define MAX_QUEUE_SIZE 100
typedef struct {
        int key;
        /* other fields */
} element;
element queue[MAX_QUEUE_SIZE];
int front = -1;
int rear = -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;
}
```

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

This sequential representation of a queue has pitfalls that are best illustrated by an example.

Example 3.2 [*Job scheduling*]: Queues are frequently used in computer programming, and a typical example is the creation of **a job queue by an operating system**. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. Figure 3.5 illustrates how an operating system might process jobs if it used a sequential representation for its queue.

front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
−1	0	J1				Job 1 is added
−1	1	J1	J2			Job 2 is added
−1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure 3.5: Insertion and deletion from a sequential queue

It should be obvious that as jobs enter and leave the system, the queue gradually shifts to the right. This means that eventually the rear index equals MAX_QUEUE_SIZE - 1, suggesting that the queue is full. In this case, queueFull should move the entire queue to the left so that the first element is again at *queue*[0] and *front* is at - 1. It should also recalculate *rear* so that it is correctly positioned. Shifting an array is very time-consuming, particularly when there are many elements in it. In fact, queueFull has a worst case complexity of O(MAX_QUEUE_SIZE).

Circular Queue

Queue CreateQ(*maxQueueSize*) ::=

```
#define MAX_QUEUE_SIZE 100
typedef struct {
        int key;
        /* other fields */
} element;
```

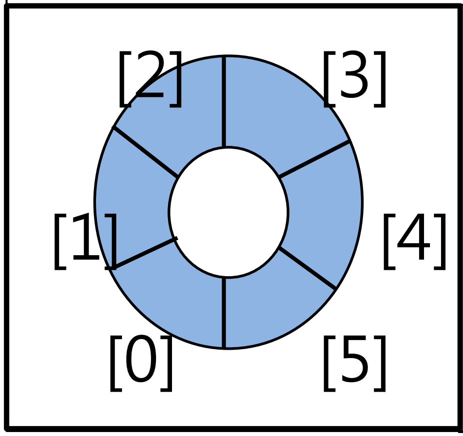
element queue[MAX_QUEUE_SIZE];

int front = 0;

int rear = 0;

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

Boolean IsFullQ(*queue*) ::= (rear+1) % MAX_QUEUE_SIZE == front



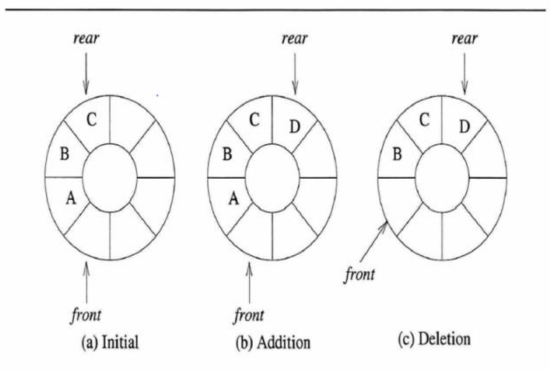


Figure 3.6: Circular queue

```
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;
}
```

Program 3.7: Add to a circular 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];
}
```

Program 3.8: Delete from a circular queue

3.4 Circular Queues Using Dynamically Allocated Arrays

Suppose that a dynamically allocated array is used to hold the queue elements. Let ***capacity*** be the number of positions in the array ***queue***. To add an element to a full queue, we must first increase the size of this array using a function such as ***realloc***. As with dynamically allocated stacks, we use ***array doubling***. However, it isn't sufficient to simply double array size using *realloc*.

Consider the full queue of Figure 3.7(a). This figure shows a queue with seven elements in an array whose capacity is 8. To visualize array doubling when a circular queue is used, it is better to **flatten out the array** as in Figure 3.7(b). Figure 3.7(c) shows the array after array doubling by *realloc*.

To get a proper circular queue configuration, we must slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in Figure 3.7(d). The array doubling and the slide to the right together copy at most 2*capacity - 2 elements. The number of elements copied can be limited to capacity - 1 by customizing the array doubling code so as to obtain the configuration of Figure 3.7(e). This configuration may be obtained as follows:

(1) Create a new array *newQueue* of twice the capacity.

(2) Copy the second segment (i.e., the elements *queue* [*front* + 1] through *queue*[*capacity* - 1]) to positions in *newQueue* beginning at 0.

(3) Copy the first segment (i.e., the elements *queue* [0] through *queue*[*rear*]) to positions in newQueue beginning at *capacity* - *front* - 1.

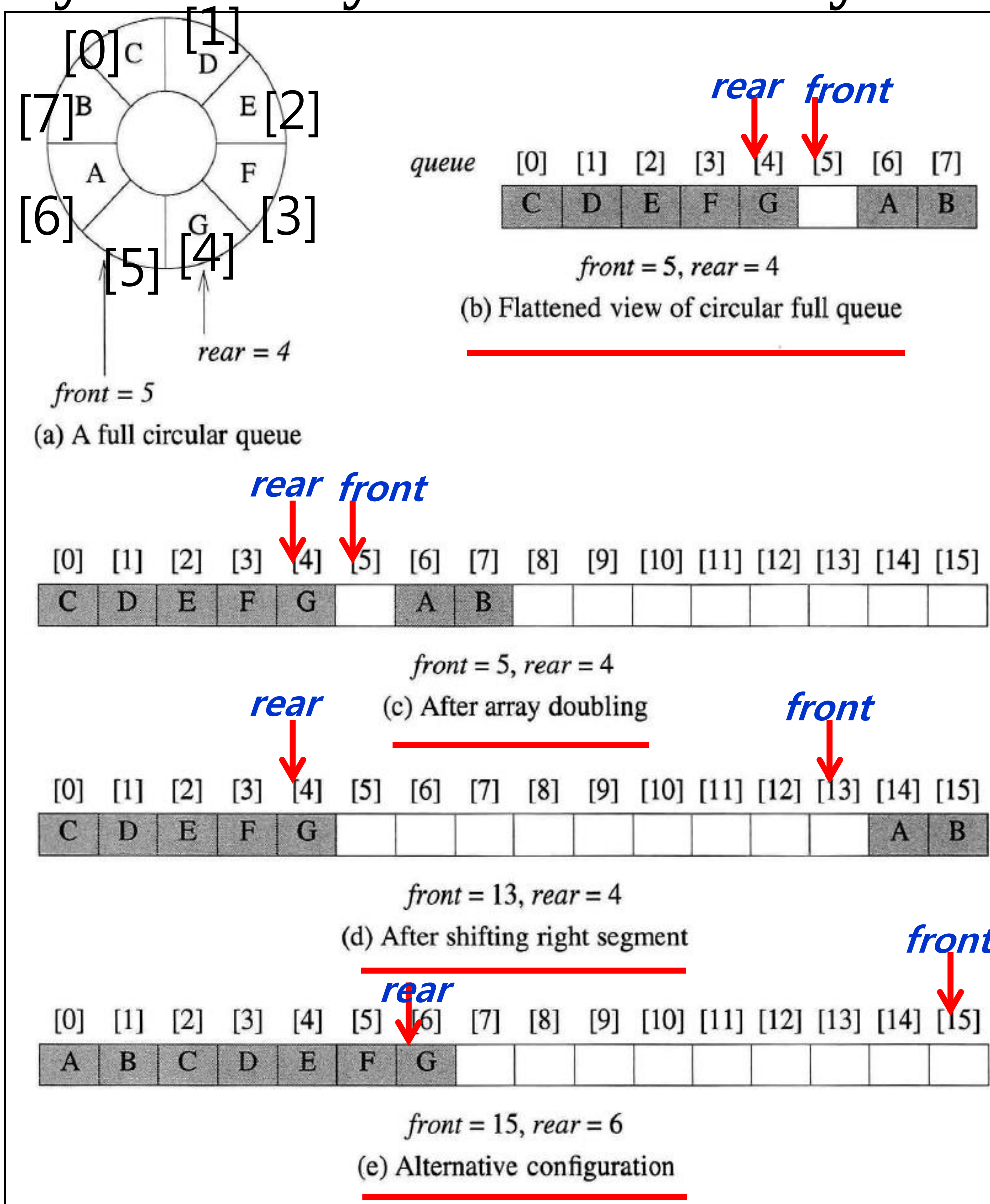


Figure 3.7: Doubling queue capacity

copy(a, b, c) copies elements from locations a through b-1 to locations beginning at c.

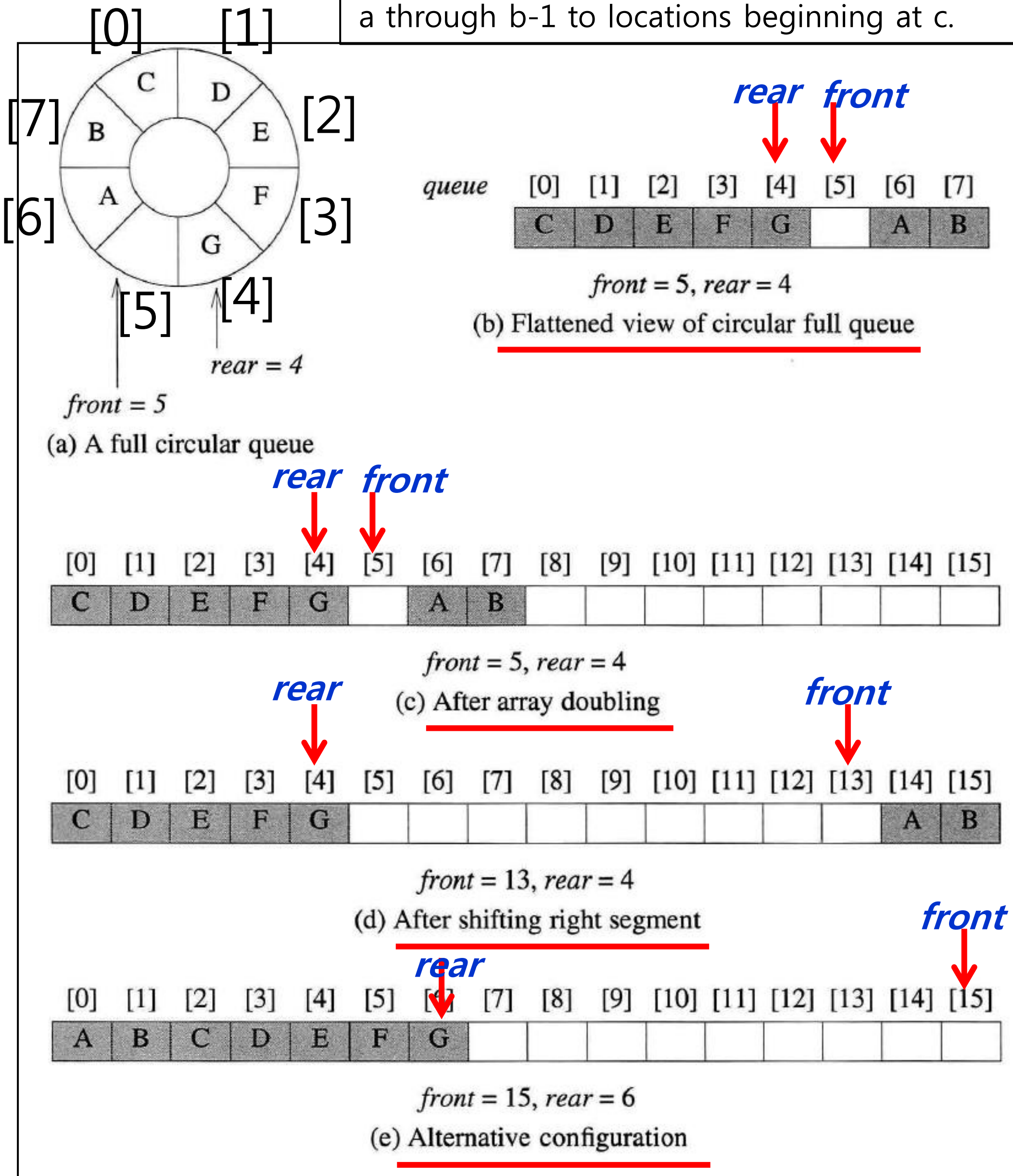


Figure 3.7: Doubling queue capacity

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

Program 3.9: Add to a circular queue

```
void queueFull()
{
   /* allocate an array with twice the capacity */
   element* newQueue;
   MALLOC(newQueue, 2 *capacity*sizeof(*queue));
   /* copy from queue to newQueue */
   int start = (front+1) % capacity;
   if (start <= 1) { /* 좌로밀착 */
      copy(queue+start, queue+start+capacity-1, newQueue);
   } else {/* queue wraps around */
      copy(queue+start, queue+capacity, newQueue);
      copy(queue, queue+rear+1, newQueue+capacity-start);
   }
   /* switch to newQueue */
   front = 2 * capacity - 1;
   rear = capacity - 2;
   capacity *= 2;
   free (queue);
   queue = newQueue;
}
```

Program 3.10: Doubling queue capacity

왼쪽 그림은 realloc()을 사용한 경우이고, 오른쪽 queueFull()은 malloc()을 사용한 경우이다.

Stack, Queue, Circular Queue 문제 [3 점]

우리는 “3.1 Stacks, 3.2 Stacks Using Dynamic Arrays, 3.3 Queues, 3.4 Circular Queues Using Dynamically Allocated Arrays”에서 Stack, Queue, Circular Queue를 공부하였다. 이들을 다음의 방법으로 실습해 보고자 한다.

우리 회사는 직원의 <id, 이름, 주소>를 보관한다. 우리 회사의 직원 수용 한도는 10 명이며 아래 사항을 최대 10 개의 엔트리를 가지는 배열로 (a) stack으로 구현하기, (b) Queue로 구현하기, 그리고 (c) Circular Queue로 구현하기를 해 보고자 한다.

(1) 다음 직원들이 차례로 입사하였다. 이들을 keyboard로 부터 받아 들이시오. 모두 보관한 후 입사 순으로 출력하시오.
<10, 일지매, 대구시 북구>, <20, 홍길동, 부산 영도구>, <90, 춘향, 전남 남원>, <40, 월매, 전남 남원>, <30, 이순신, 서울>, <60, 을지문득, 평양>, <99, 차두리, 서울 강남구>, <88, 박지성, 경기도 수원>.

(2) 3명이 그 자료구조의 성격대로 차례로 퇴사하였다. 즉, stack은 나중 입사자가 먼저 퇴사, queue와 circular queue는 먼저 입사자가 먼저 퇴사한다. 그들을 퇴사 순으로 출력하시오.

(3) 다음 직원들이 차례로 입사하였다. 이들을 keyboard로 부터 받아 들이시오. 모두 보관한 후 전체 직원들을 입사 순으로 출력하시오.
<33, 이승엽, 대구>, <44, 이세돌, 전남>, <66, 송중기, 서울>, <77, 송혜교, 경기도>.

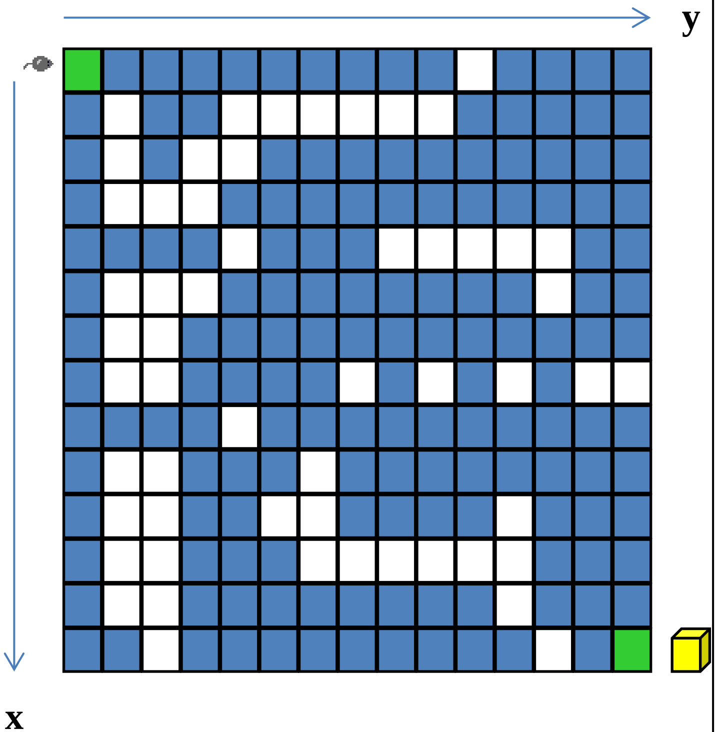
(4) 6명이 그 자료구조의 성격대로 차례로 퇴사하였다. 그들을 퇴사 순으로 출력하시오.

(5) 회사에 남아 있는 직원들을 입사 순으로 차례로 출력하시오.

3.5 A Mazing Problem

Mazes have been an intriguing subject for many years. Experimental psychologists train rats to search mazes for food, and many a mystery novelist has used an English country garden maze as the setting for a murder. We also are interested in mazes since they present **a nice application of stacks**. In this section, we develop a program that runs a maze. Although this program takes many ***false paths*** before it finds a correct one, once found it can correctly rerun the maze without taking any false paths.

- Rat In A Maze

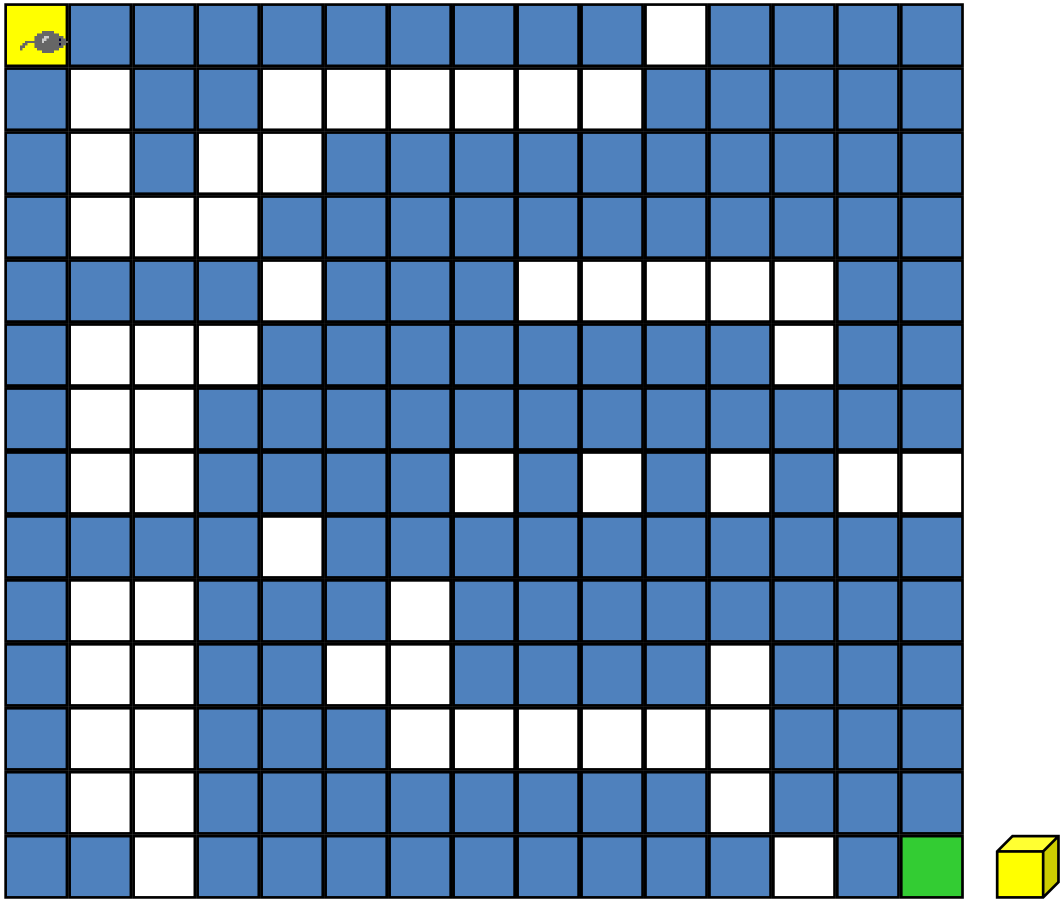


미로의 표현: maze[row][col]
열린 길: maze[x][y] = 0; // blue
막힌 길: maze[x][y] = 1; // white
시작 위치: <0, 0>
출구 위치: <xf, yf>
미로에서 방문한 위치 표현: mark[row][col]
초기화: mark[x][y] = 0;
방문하면: mark[x][y] = 1;
이동 가능 검사 순서: <우측(0), 아래(1), 좌측(2), 위(3)>
현재위치로 돌아왔을 때를 대비한
<현재 위치 정보, 다음 이동 가능 검사 순서 정보>의 보관:

```
#define MAX_STACK_SIZE 100
typedef struct {
    short  row;     //  x
    short  col;     //  y
    short  dir;     //  <x,y>로 돌아 왔을 때 이동 방향
} element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

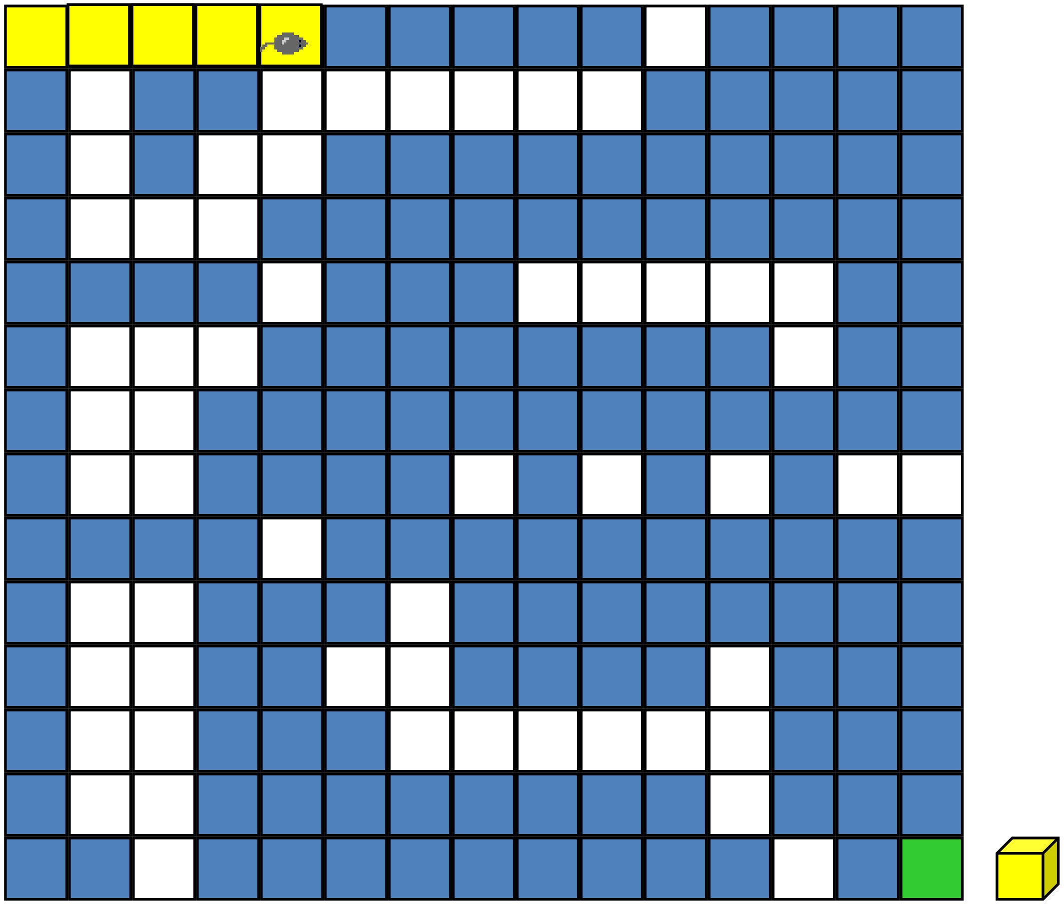
Forward Movement: <x1, y1>에서 <x2, y2>로 이동 시 수행할 작업
← (maze[x2][y2] = 0 and mark[x2][y2] == 0) 이면 이동가능
mark[x2][y2] = 1;
stack[++top] = <x1, y1, ++dir>;
x = x2; y = y2, dir = 0;
Backward Movement: <x1, y1>에서 4 방향 모두 검사완료 시
if (top == -1) {돌아갈 곳 없음; exit(1) }
<x, y, dir> = stack[top--];
경로 발견시
(1) stack[0] 부터 순서대로 각 엔트리의 <위치 정보> 출력,
(2) 현재 위치 출력, (3) 출구 위치 출력.

Rat In A Maze



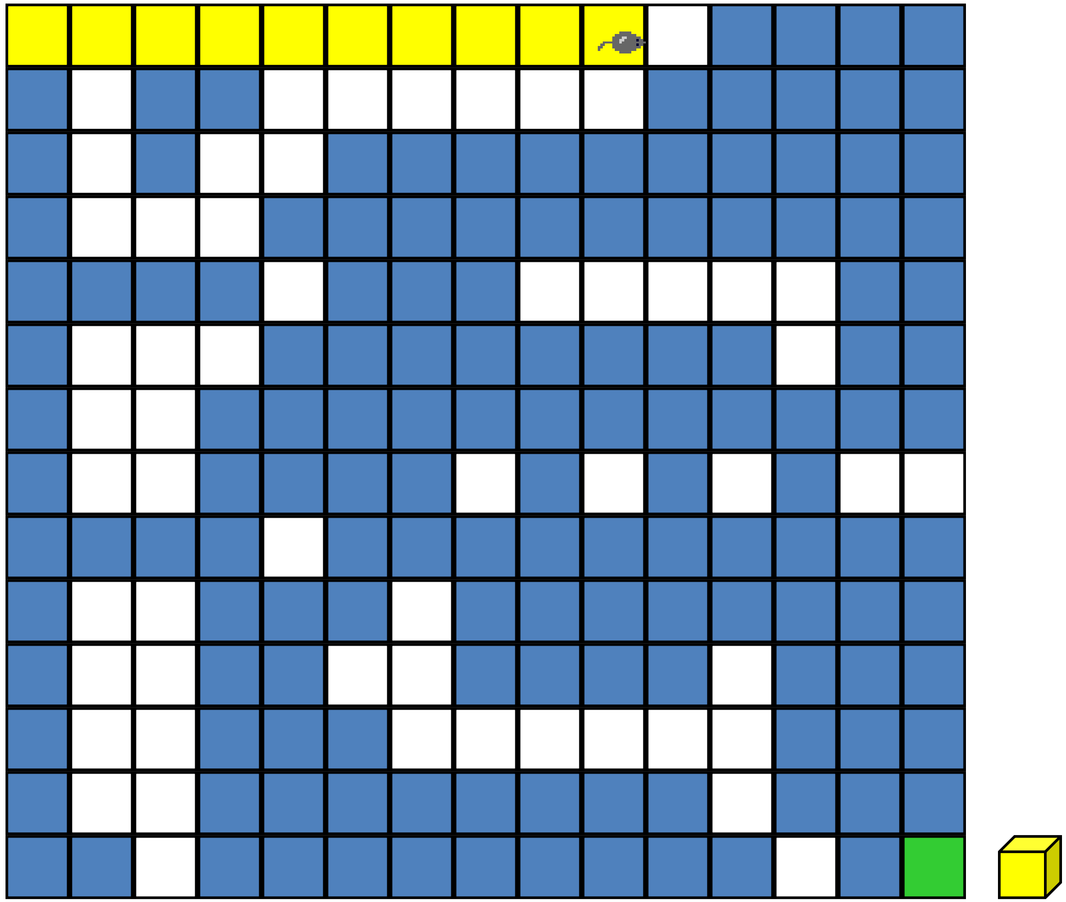
- Move order is: right(0), down(1), left(2), up(3).
- Block positions to avoid revisit.

Rat In A Maze



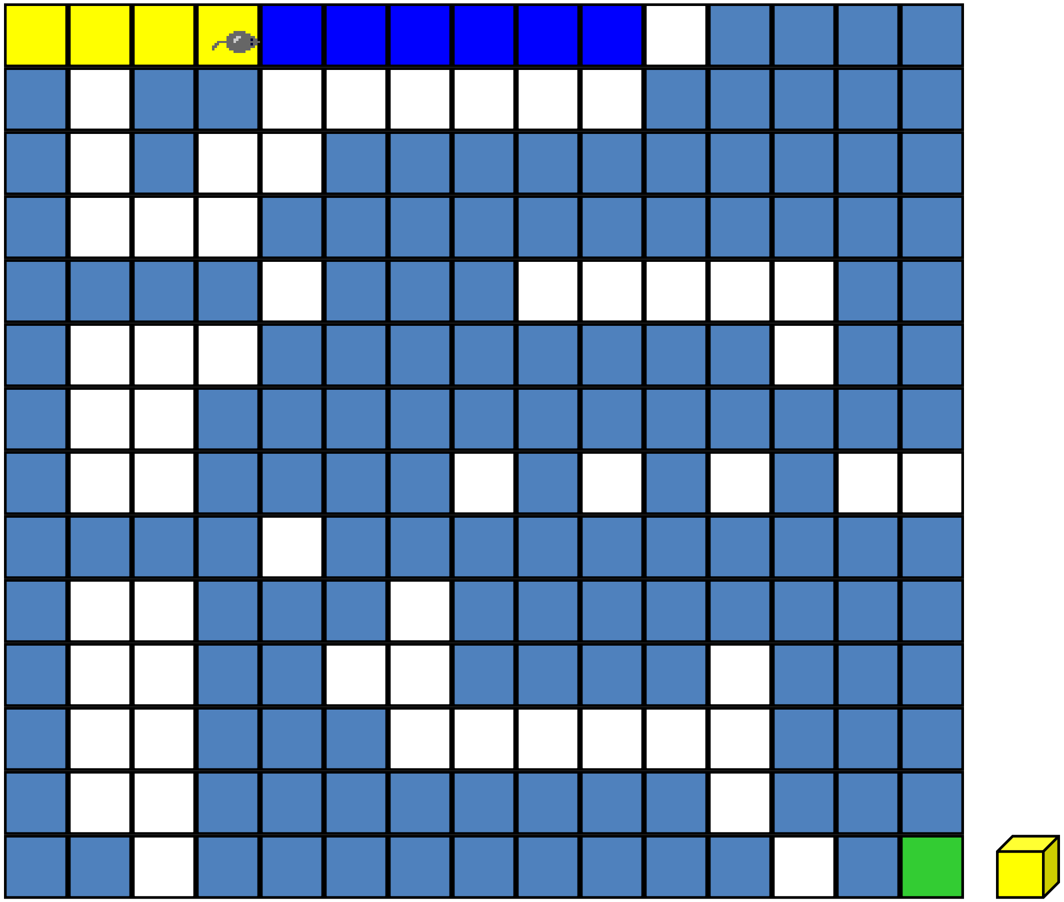
- Move order is: right, down, left, up.
- Block positions to avoid revisit.

Rat In A Maze



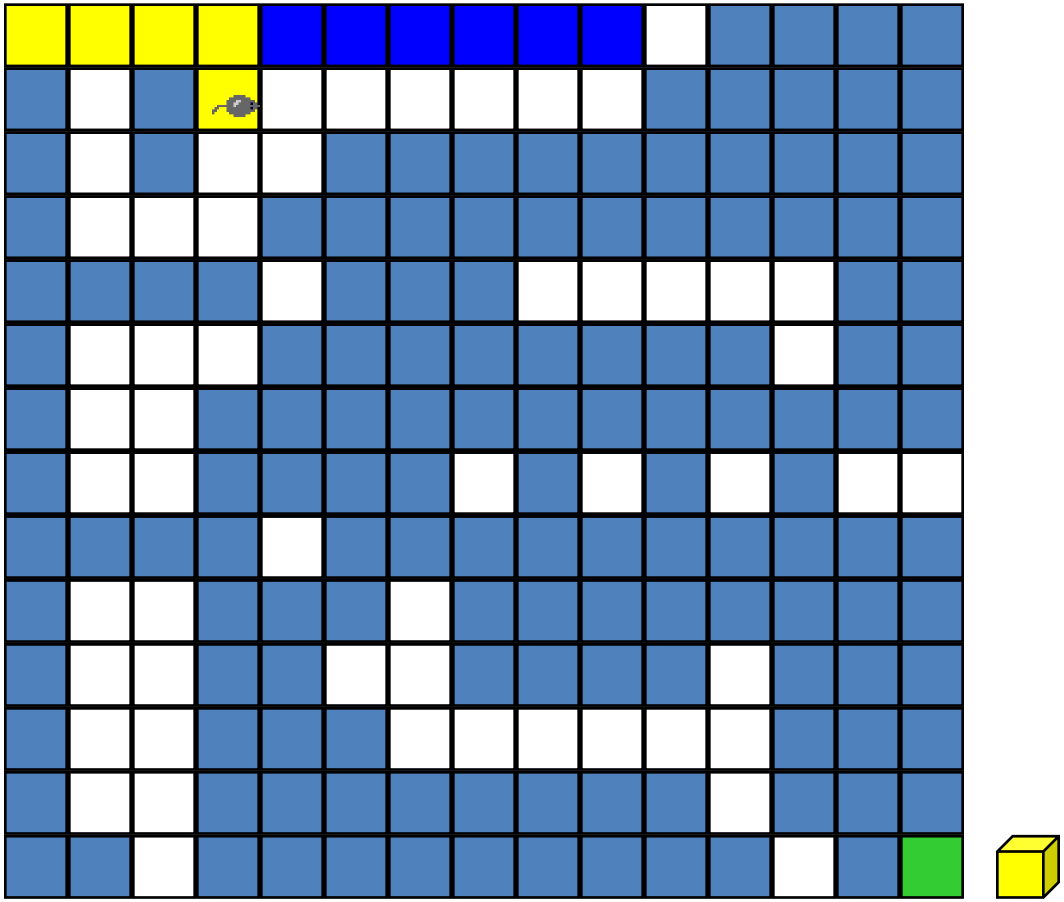
- Move backward until we reach a square from which **a forward move** is possible.

Rat In A Maze



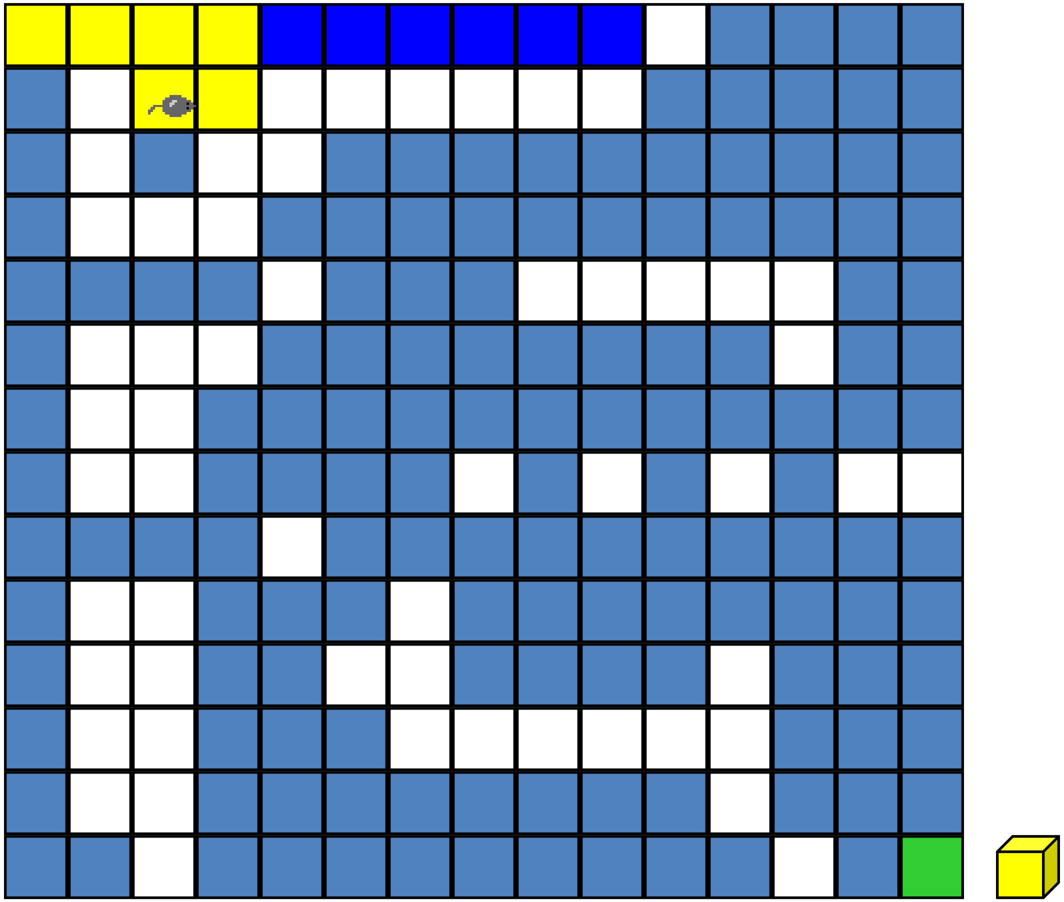
- Move down.

Rat In A Maze



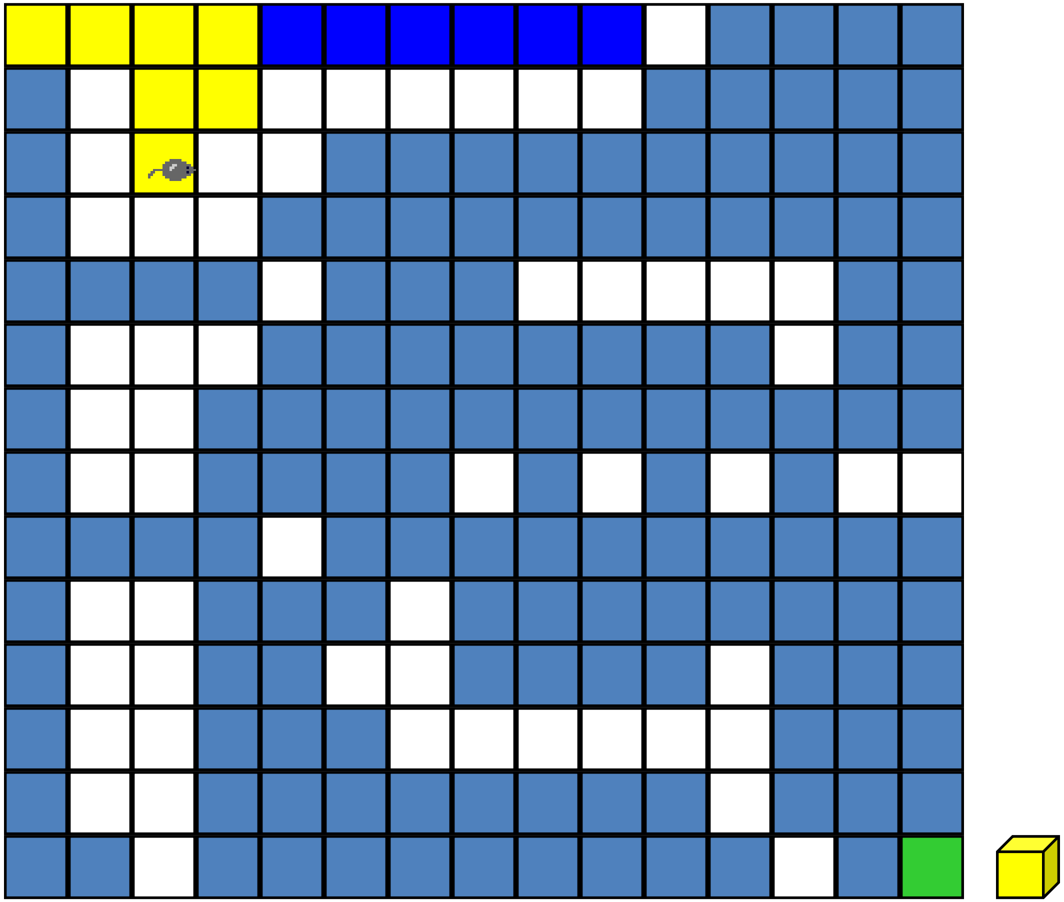
- Move left.

Rat In A Maze



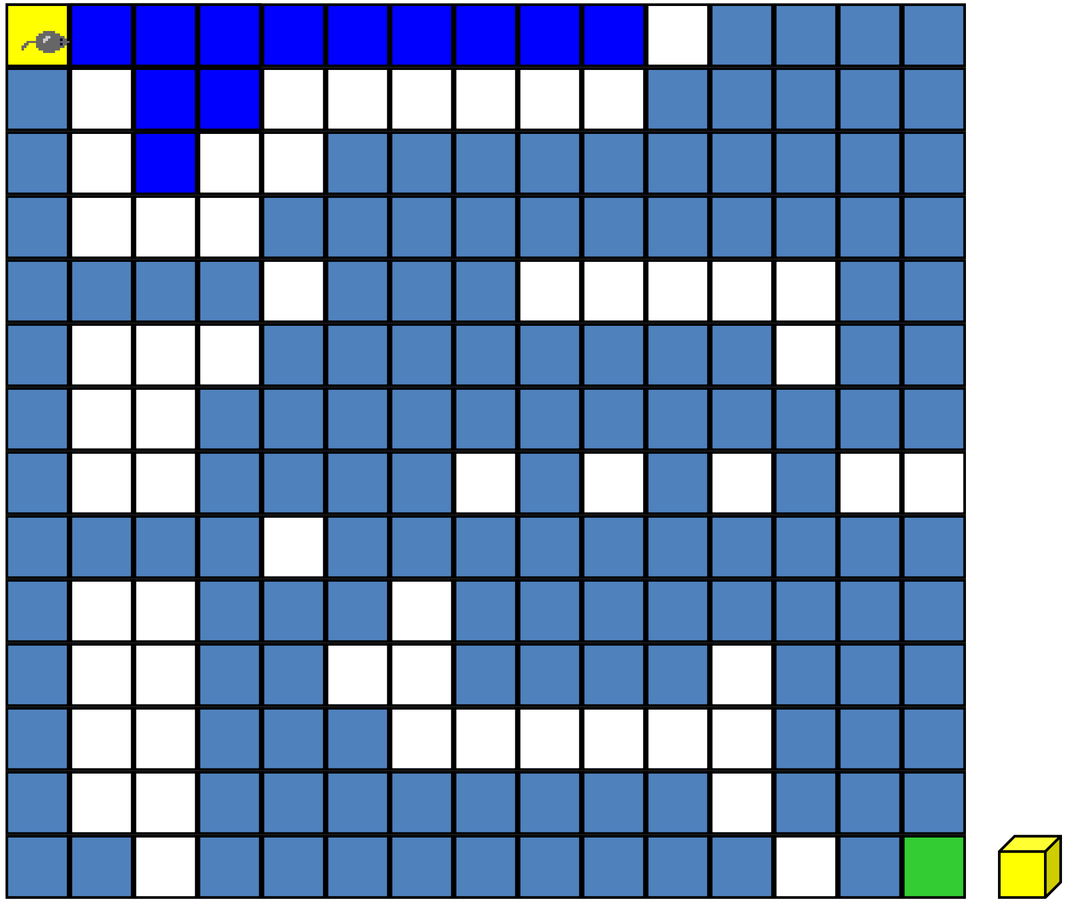
- Move down.

Rat In A Maze



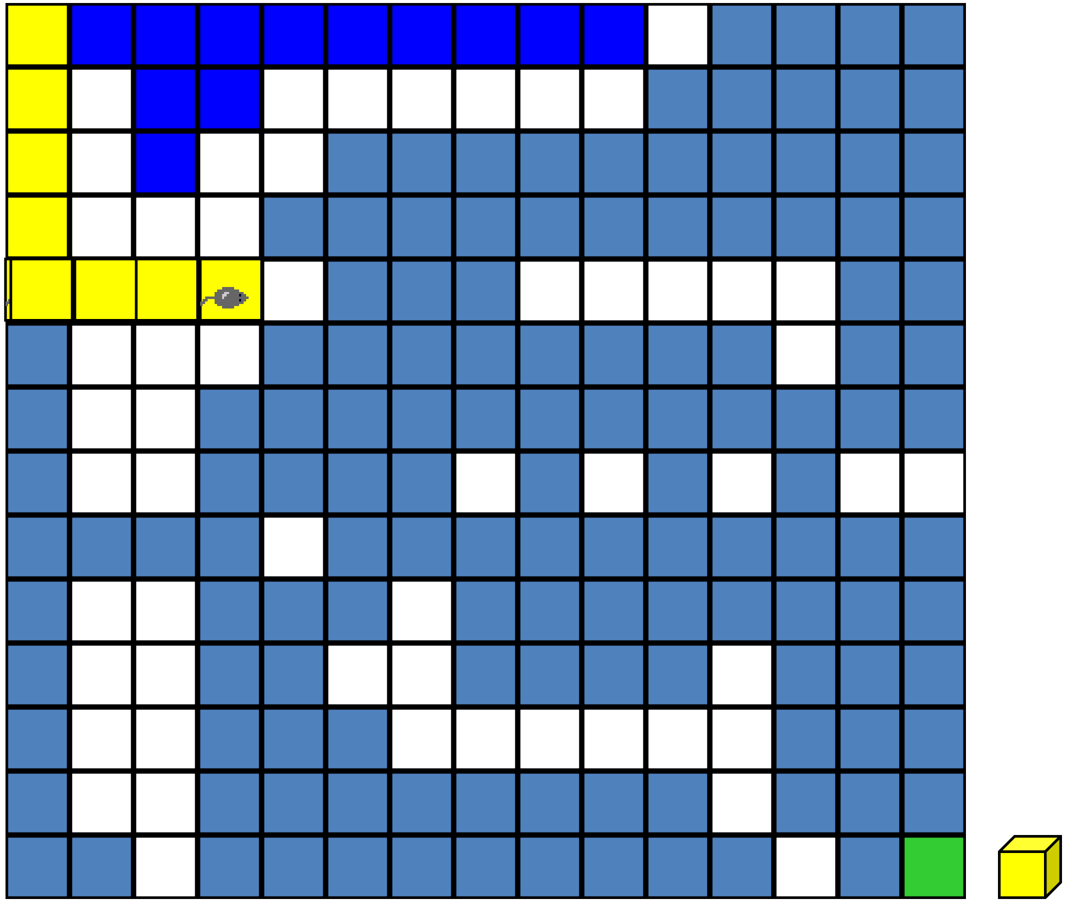
- Move backward until we reach a square from which **a forward move** is possible.

Rat In A Maze



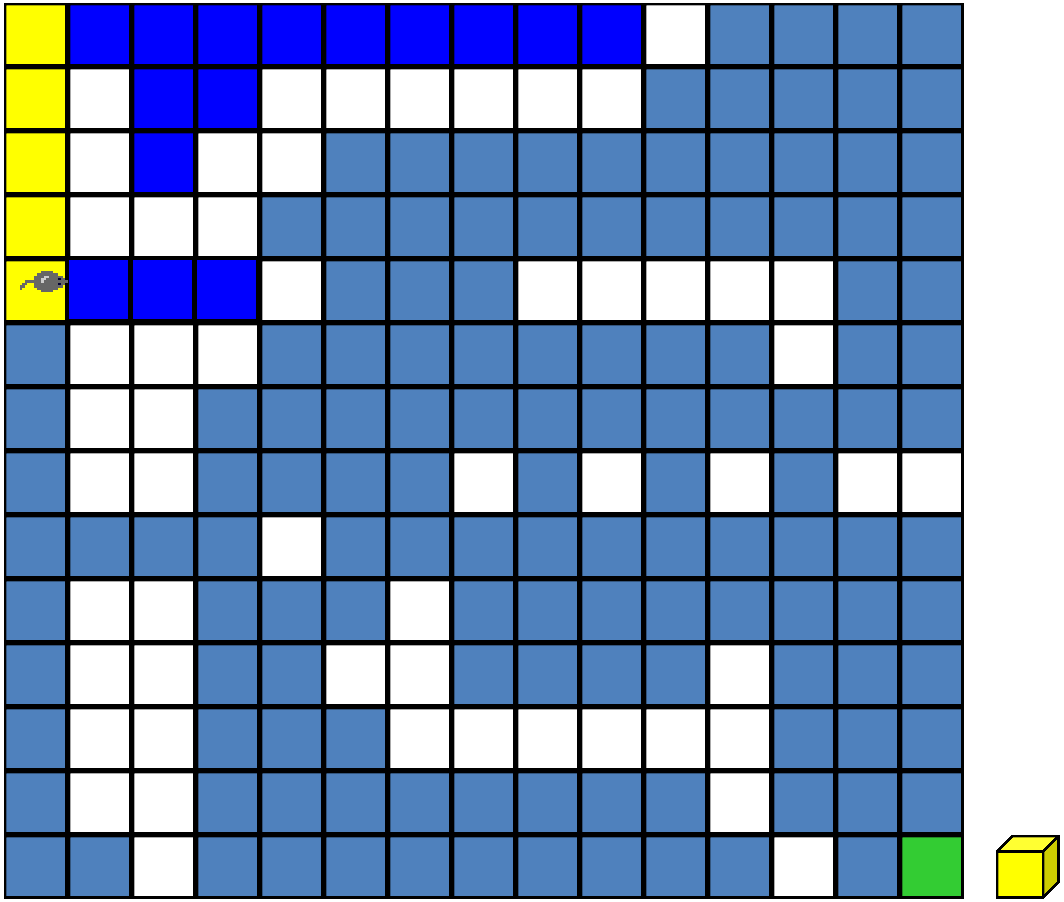
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



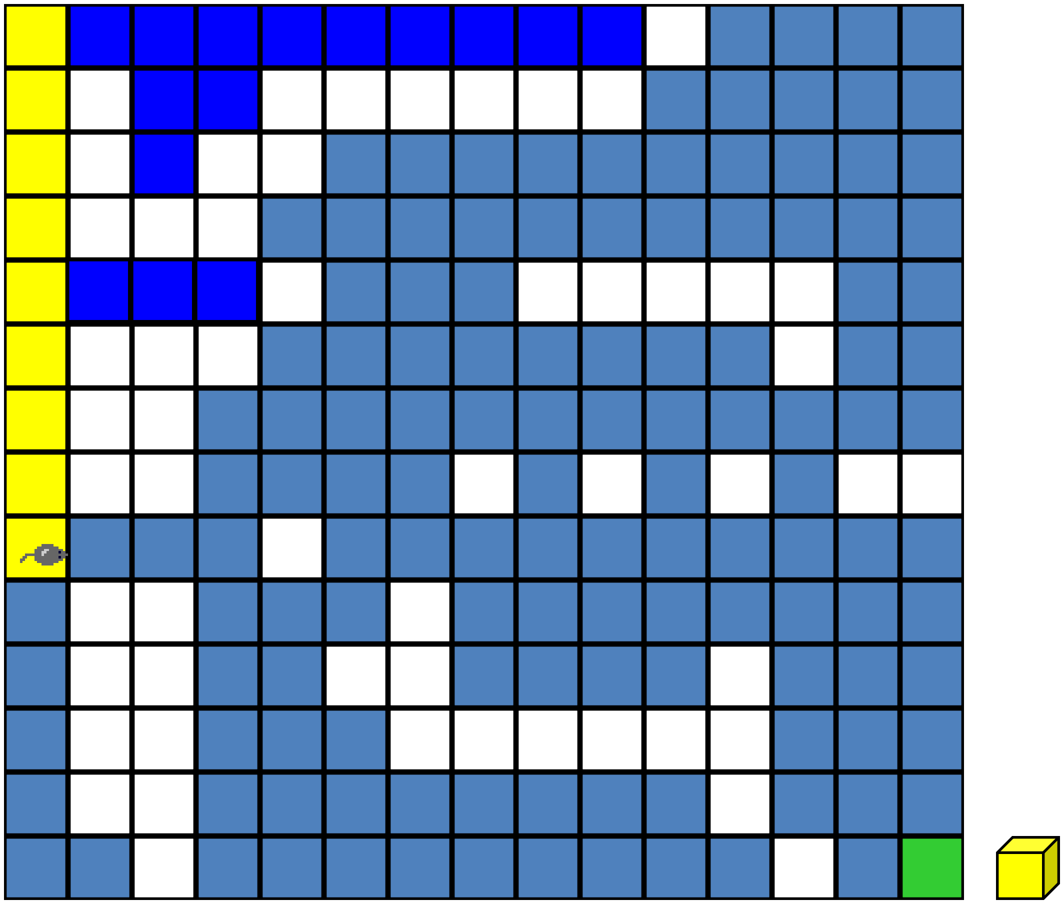
- Move right.
- **Backtrack**.

Rat In A Maze



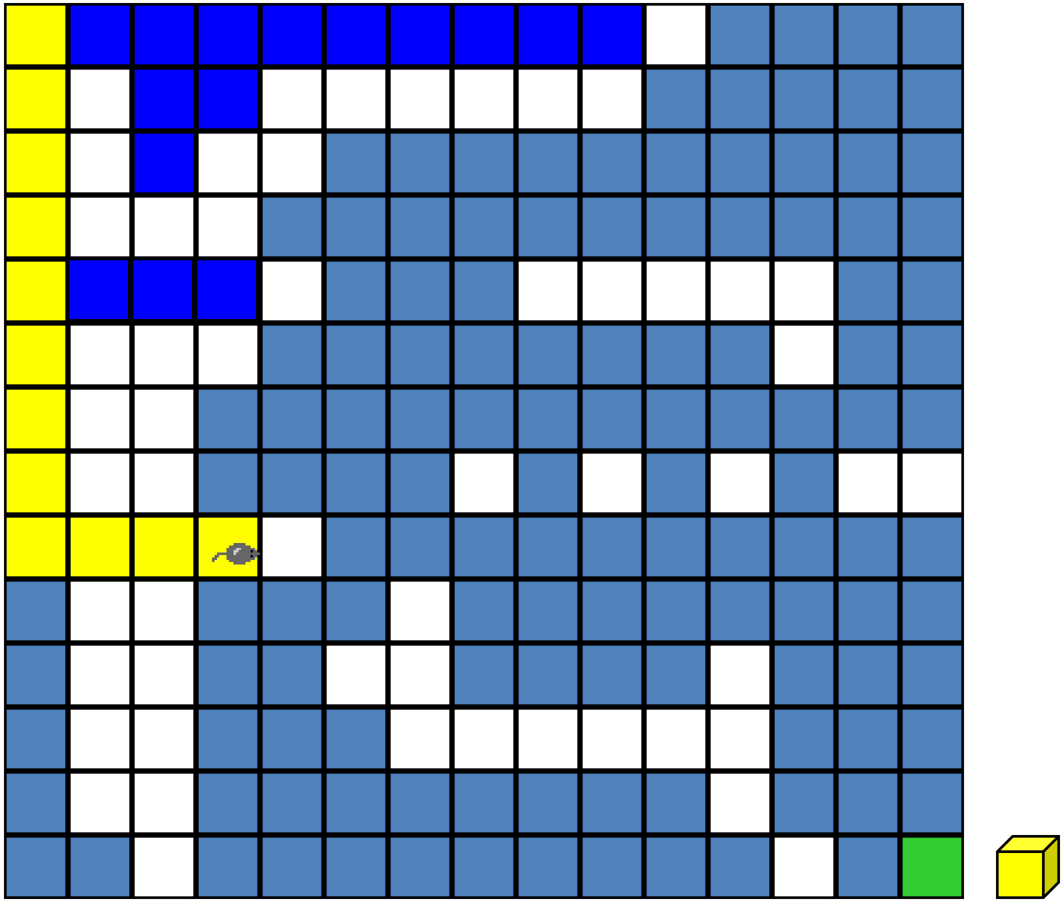
- Move downward.

Rat In A Maze



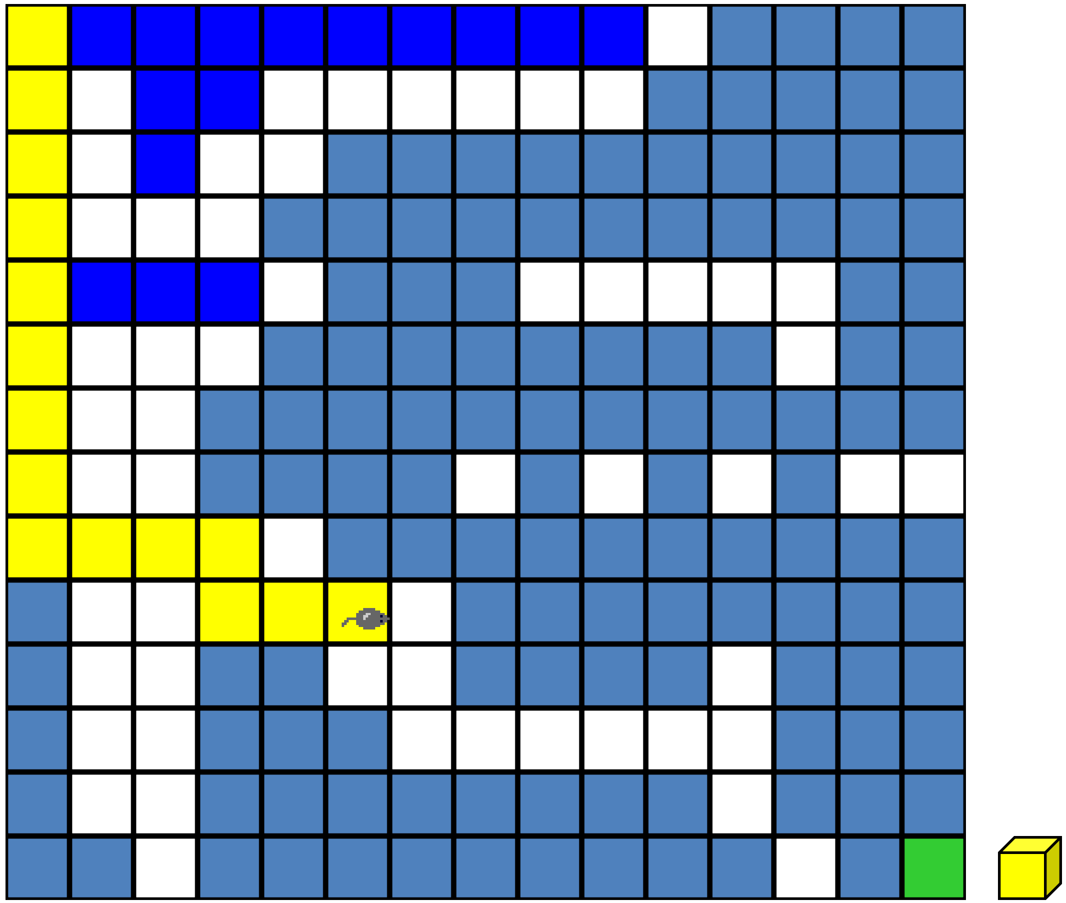
- Move right.

Rat In A Maze



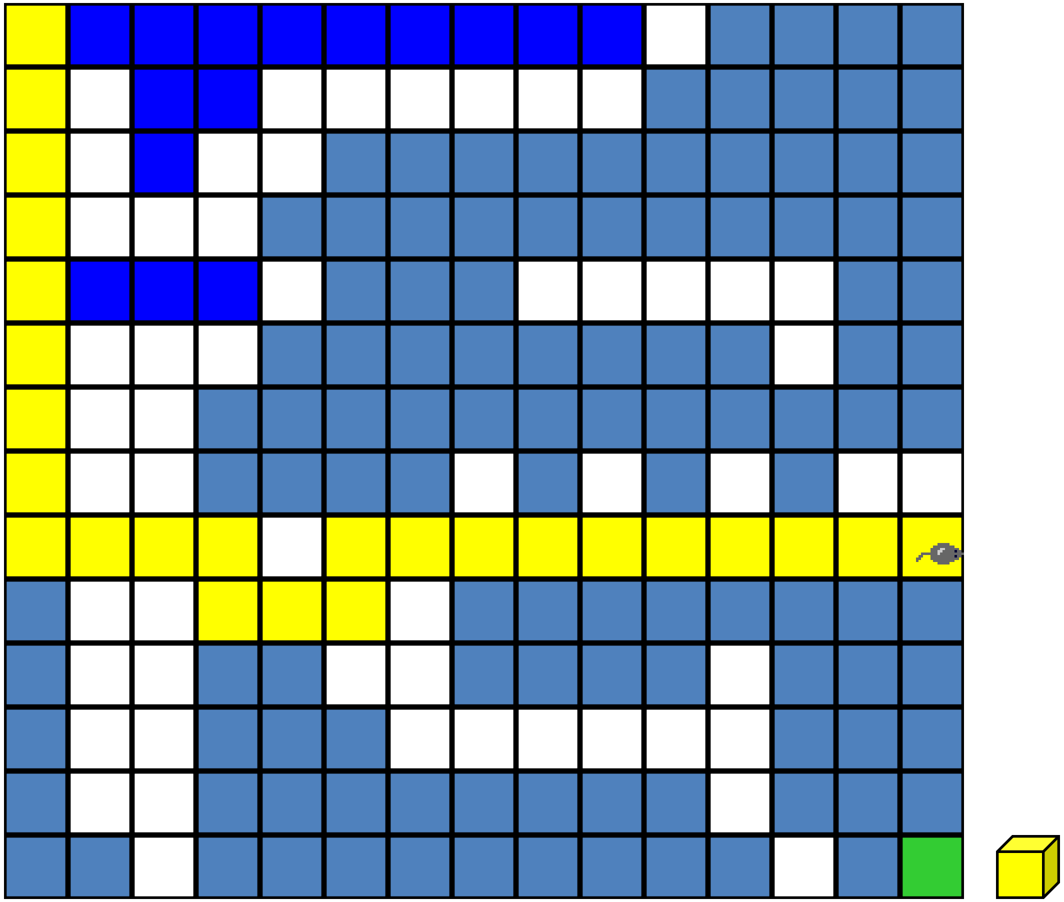
- Move one down and then right.

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***.

A Mazing Problem : Implementation in C

y

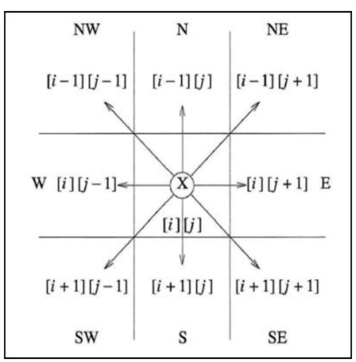
entrance

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

x

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



미로의 표현: maze[row+2][col+2]
　　열린 길: maze[x][y] = 0;
　　막힌 길: maze[x][y] = 1;
시작 위치: <1, 1>
출구 위치: <xf, yf> 🡸 <12, 15>

미로에서 방문한 위치 표현: mark[row+2][col+2]
　　초기화: mark[x][y] = 0;
　　방문하면: mark[x][y] = 1;

이동 가능 검사 순서의 표현: move[8];
　　<N(0), NE(1), E(2), SE(3), S(4), SW(5), W(6), NW(7)>의 순서

현재위치로 돌아왔을 때를 대비한 <현재 위치 정보, 다음 이동 가능 검사 순서 정보>의 보관: stack[MAX_STACK_SIZE], top = -1;

Forward Movement: <x1, y1>에서 <x2, y2>로 이동 시 수행할 작업
🡸 (maze[x2][y2] = 0 and mark[x2][y2] == 0) 이면 이동가능
　　mark[x2][y2] = 1;
　　stack[++top] = <x1, y1, ++dir>;
　　x = x2; y = y2, dir = 0;
Backward Movement: <x1, y1>에서 8 방향 모두 검사완료 시
　　if (top == -1) {돌아갈 곳 없음; exit(1) }
　　<x, y, dir> = stack[top--];
경로 발견시
(1) stack[0] 부터 순서대로 각 엔트리의 <위치 정보> 출력,
(2) 현재 위치 출력, (3) 출구 위치 출력.

A Mazing Problem : Implementation in C

- Representation of a maze
  - *A two-dimensional array*
  - 0 : the open paths, 1 : the barriers

y

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

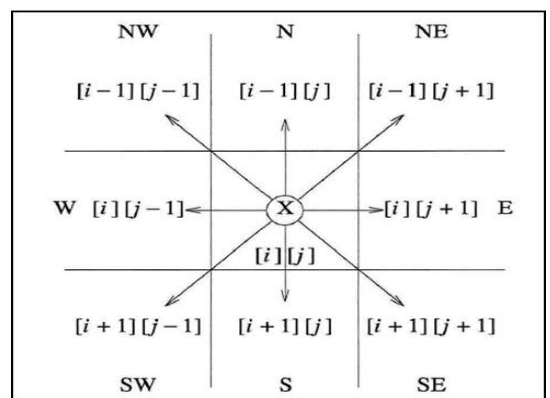
exit

x

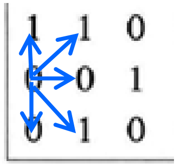
- Assumptions
  - Rat starts at **the top left**.
  - Exits at **the bottom right**.

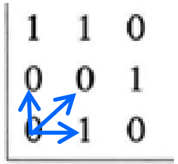
- The location of the rat in the maze
  - can be described by the *row* and *column* position
  - *maze*[*row*][*col*]

- The possible *8 moves* from this position



- Not every position has eight neighbors.
  - If [*row*, *col*] is on a border, then less than eight.



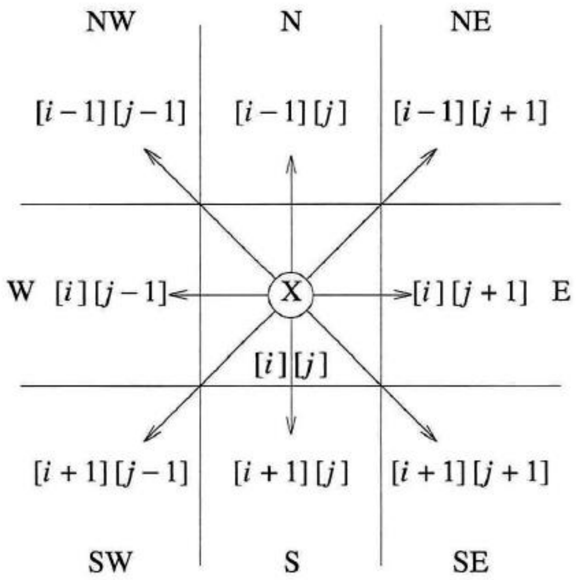


- To avoid checking for boarder conditions
  - We can surround the maze by a boarder of ones.

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

< $m \times p$ maze >
$(m+2) \times (p+2)$ array, ***maze***
entrance : *maze*[1][1]
exit : *maze*[*m*][*p*]

- Predefine possible directions to move, in an array ***move***



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;      // x
                short int horiz;     // y
}  offsets;
offsets move[8];  /* array of moves for
                      each direction */
```

- Finding the position of the next move, *maze*[*nextRow*][*nextCol*]

nextRow = row + move[dir].vert;

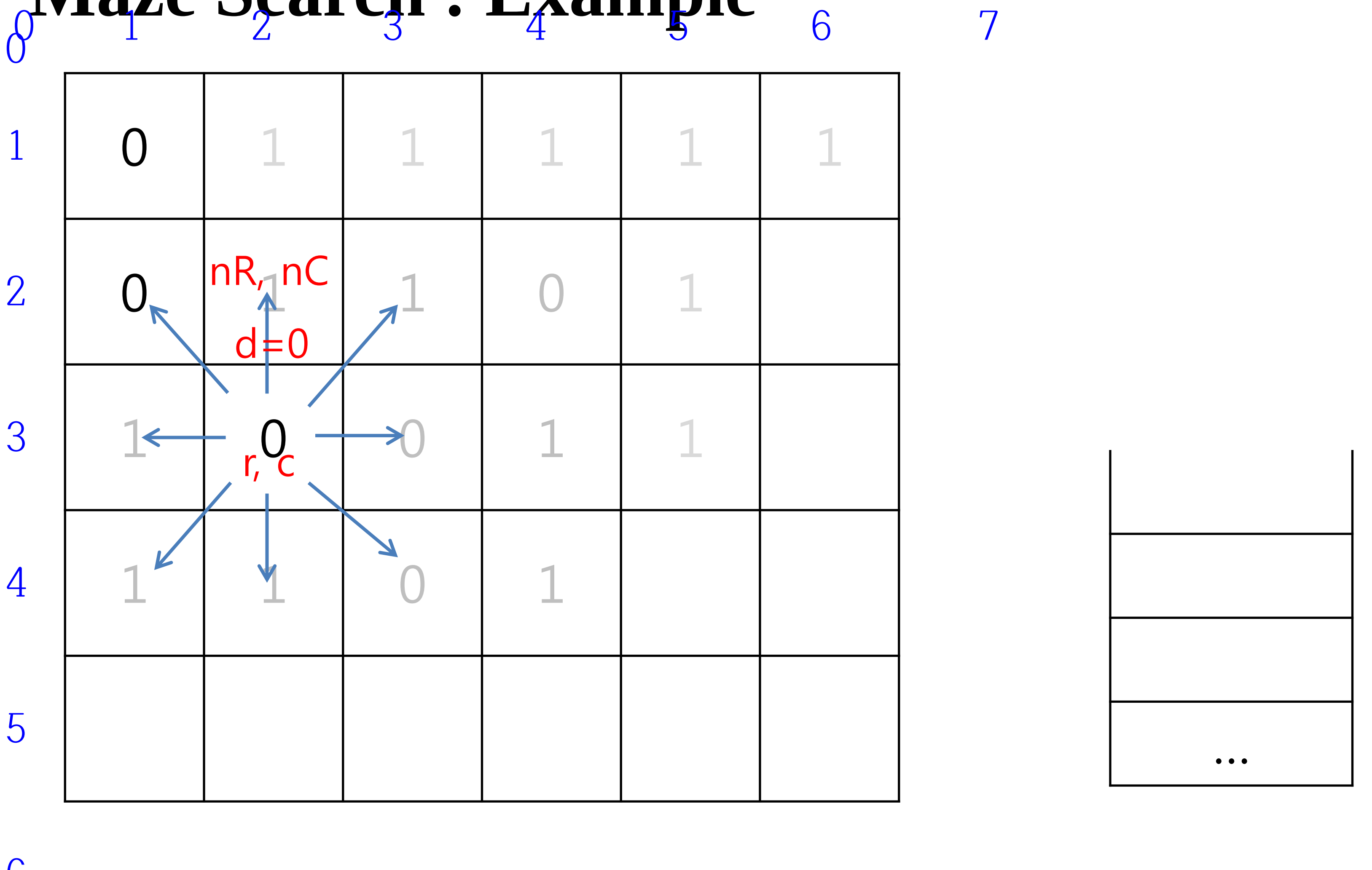
nextCol = col + move[dir].horiz;

- We maintain a 2D array, ***mark***, to record the maze positions already checked.
  - We initialize the *mark*'s entries to zero
  - When we visit a *maze*[*row*][*col*], we change *mark*[*row*][*col*] to one

- Stack

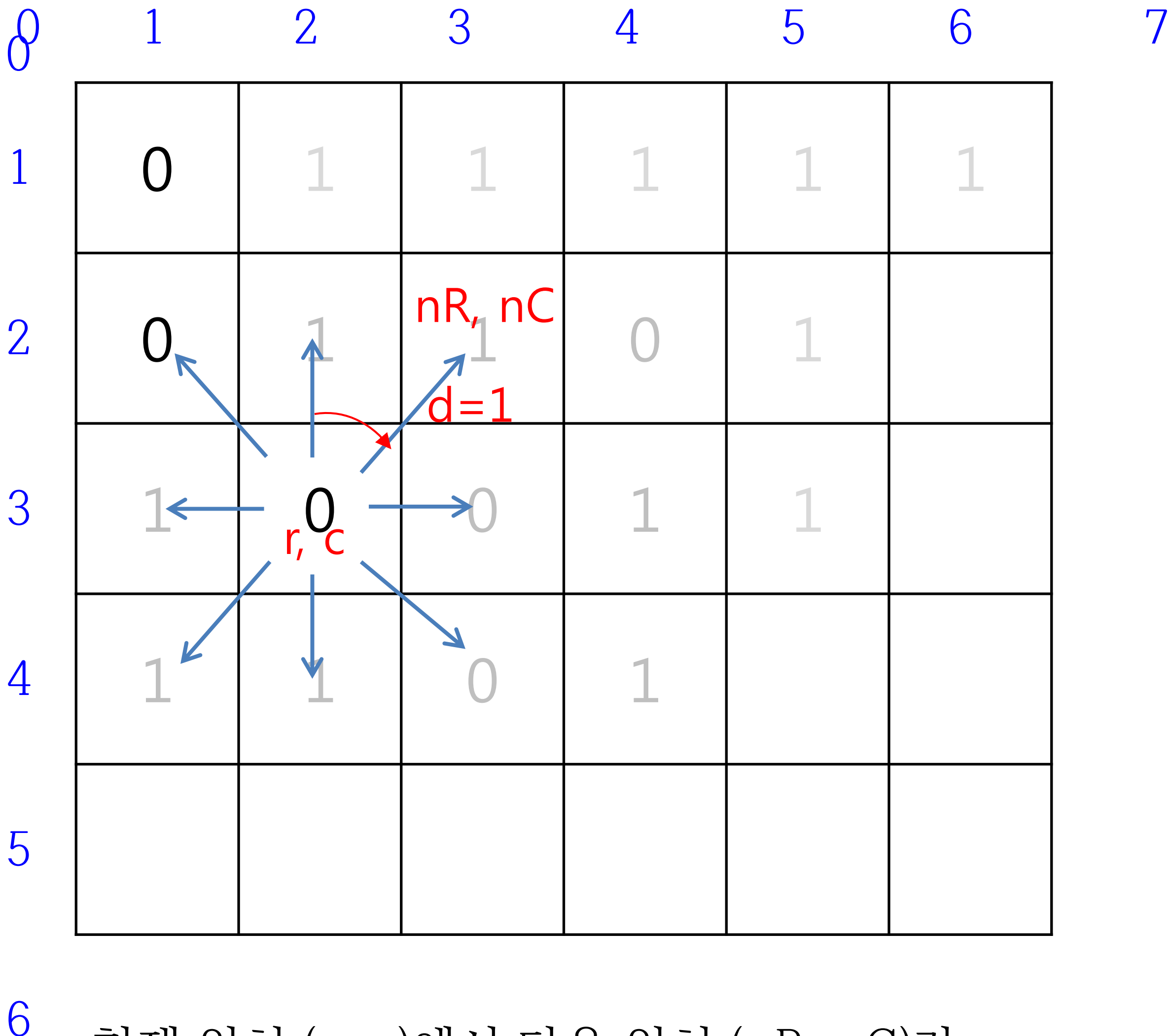
```
#define MAX_STACK_SIZE 100
typedef struct {
        short int row;    // x
        short int col;    // y
        short int dir;
} element;

element stack[MAX_STACK_SIZE];
```

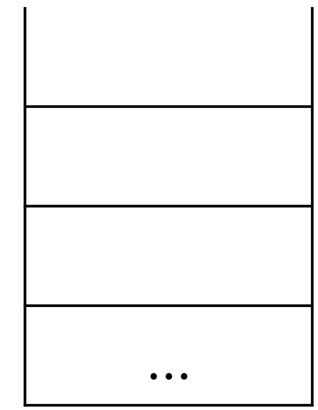
Maze Search : Example



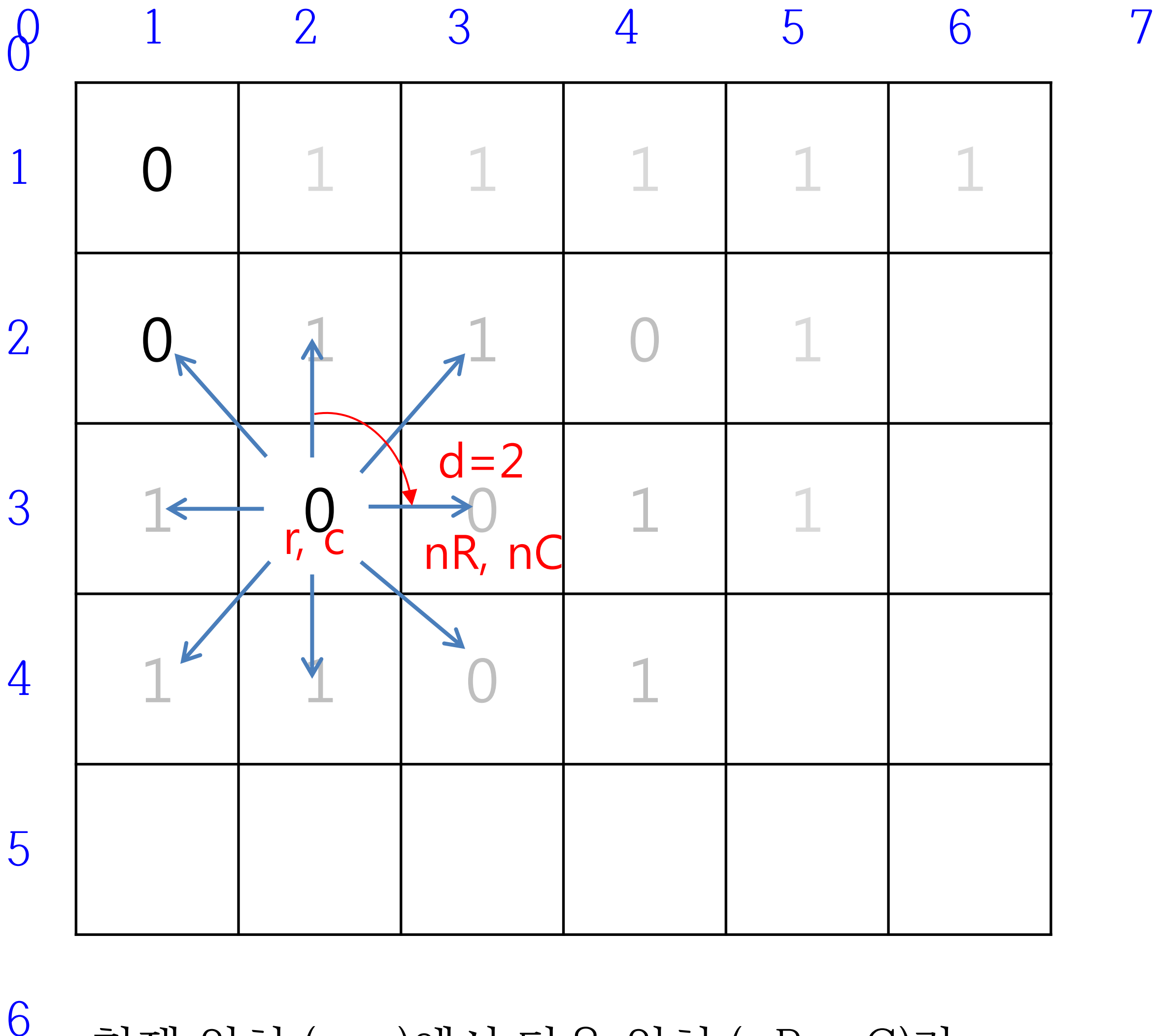
현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
No!

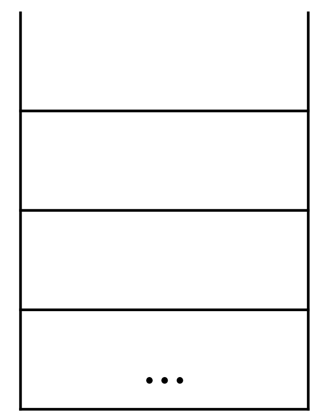
※배열의 경계부분 생략,
방문한 위치는 0을 진하게 표시



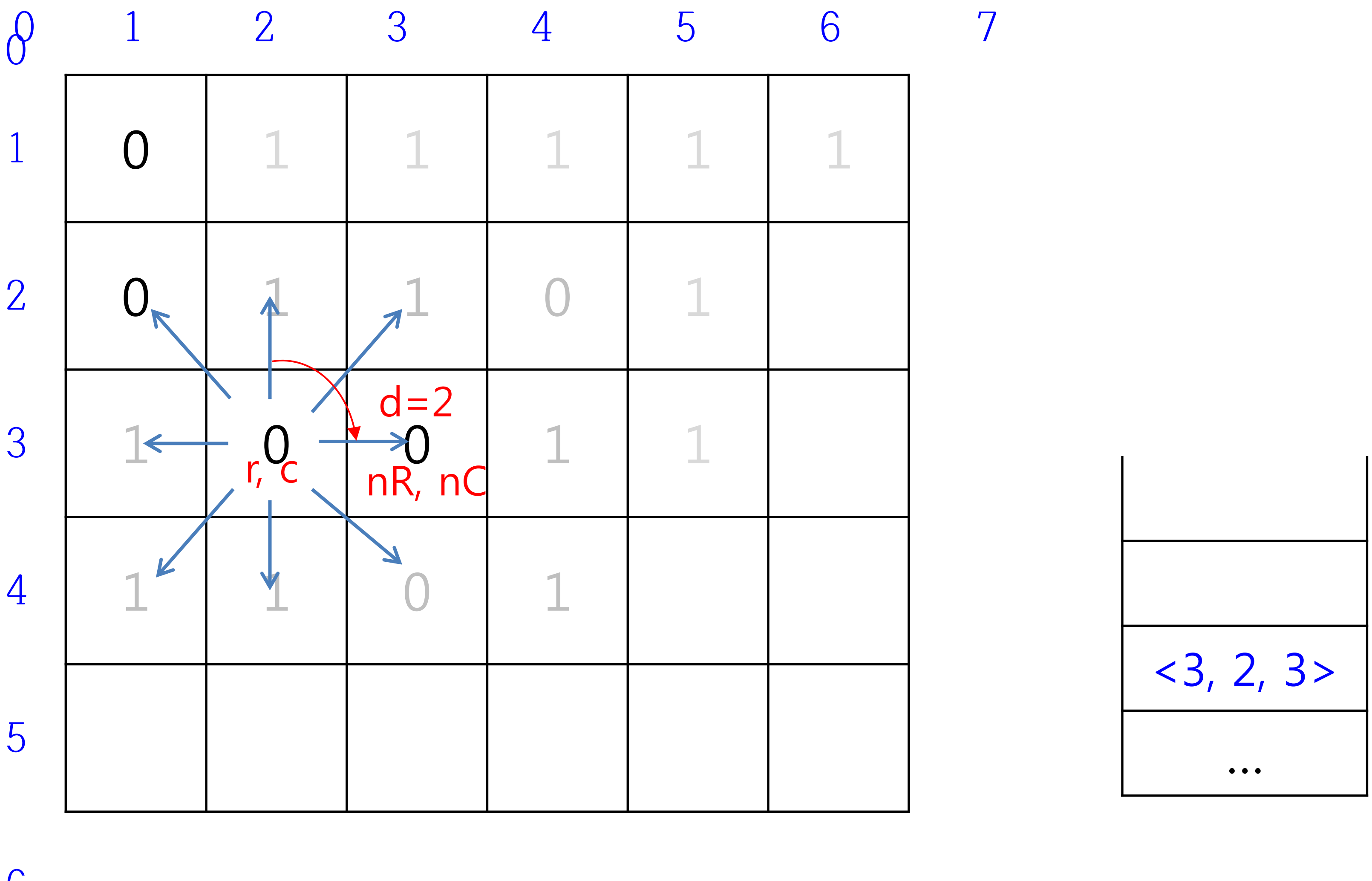


현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
No!

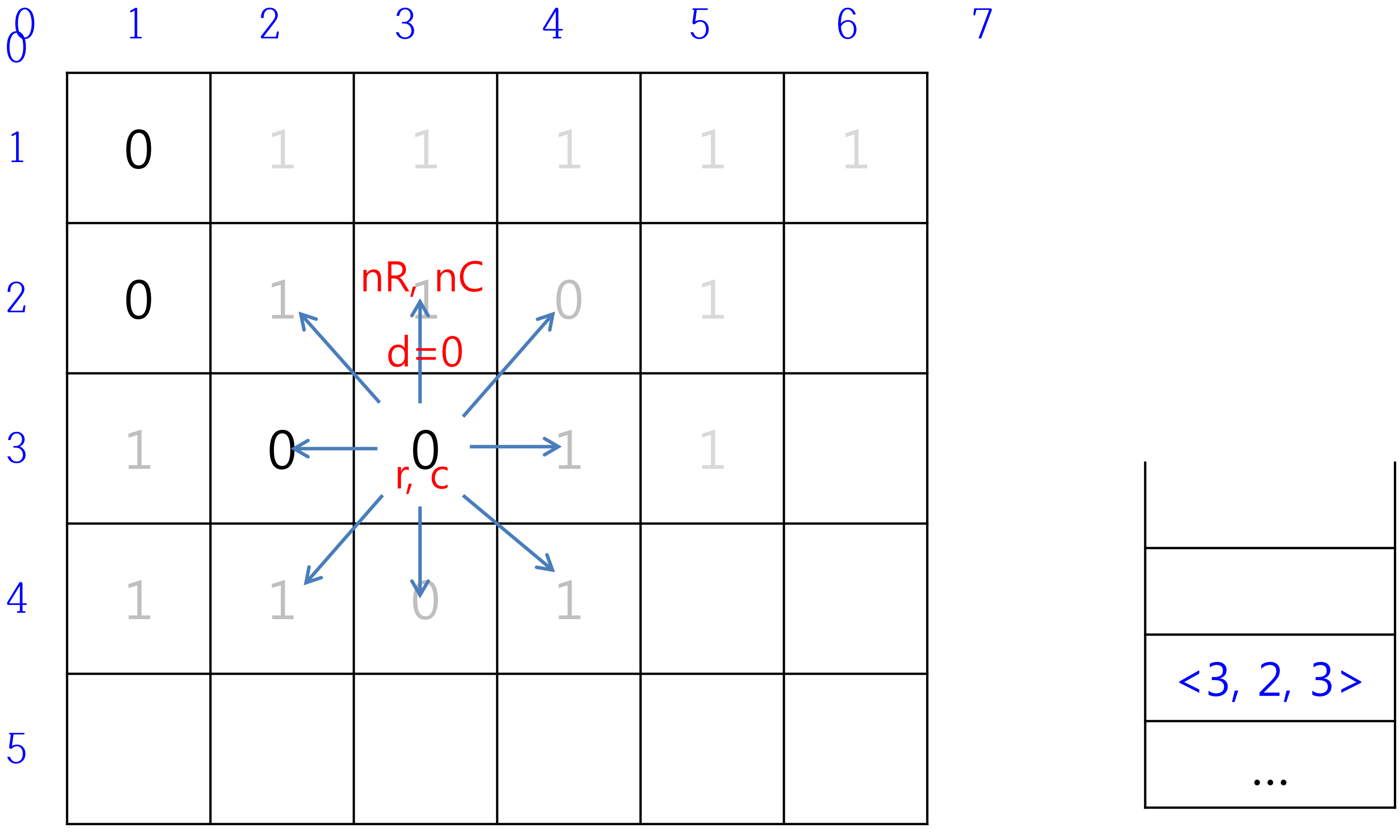




현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
Yes!

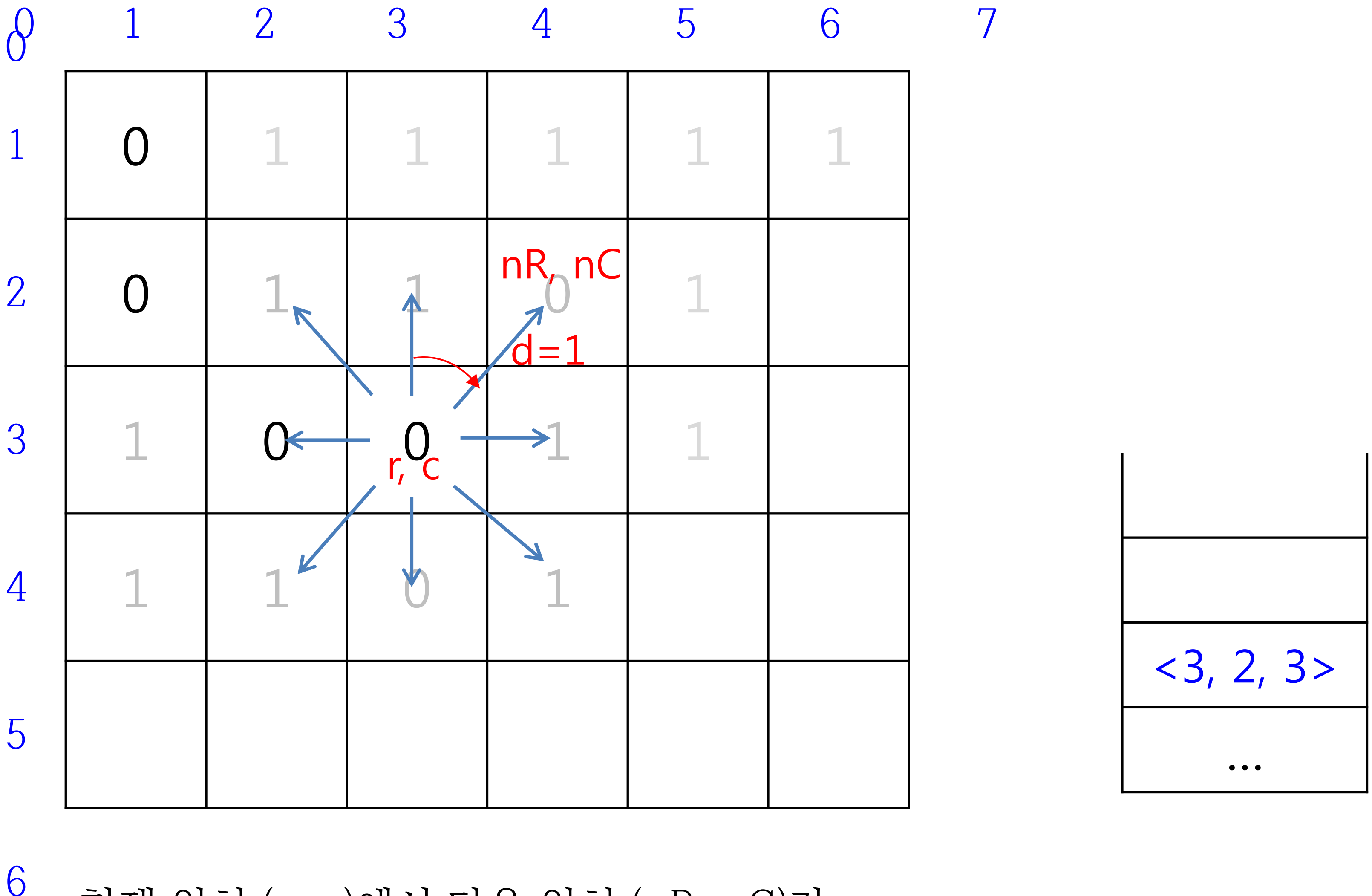


① (nR, nC) 위치를 방문했음을 표시
② (r, c) 에서 다음 번에 검사할 방향 (r, c, ++d) = (3, 2, 3)을 스택에 Push
③ r, c, d를 nR, nC, 0으로 업데이트함 (이동)

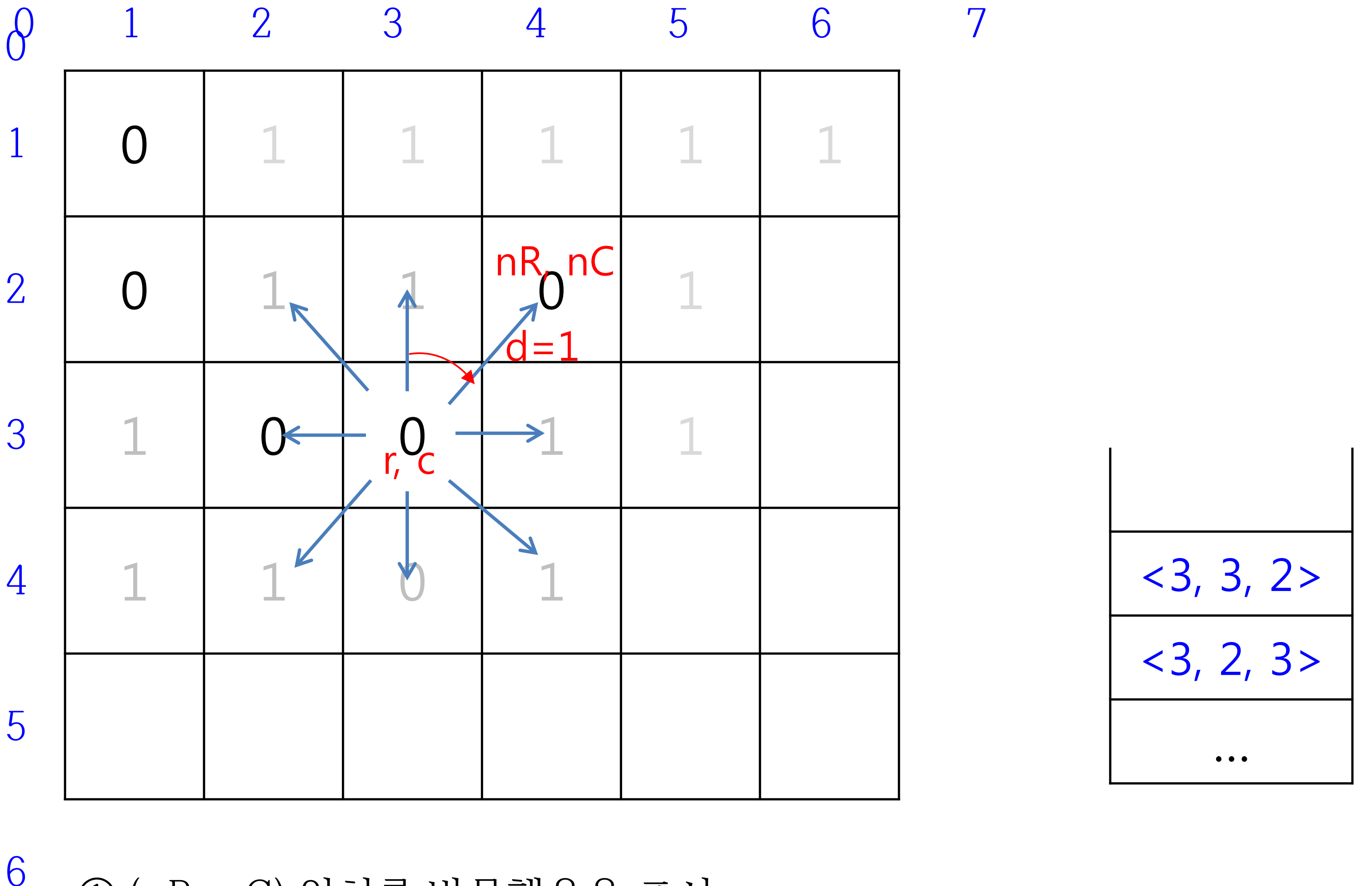


6

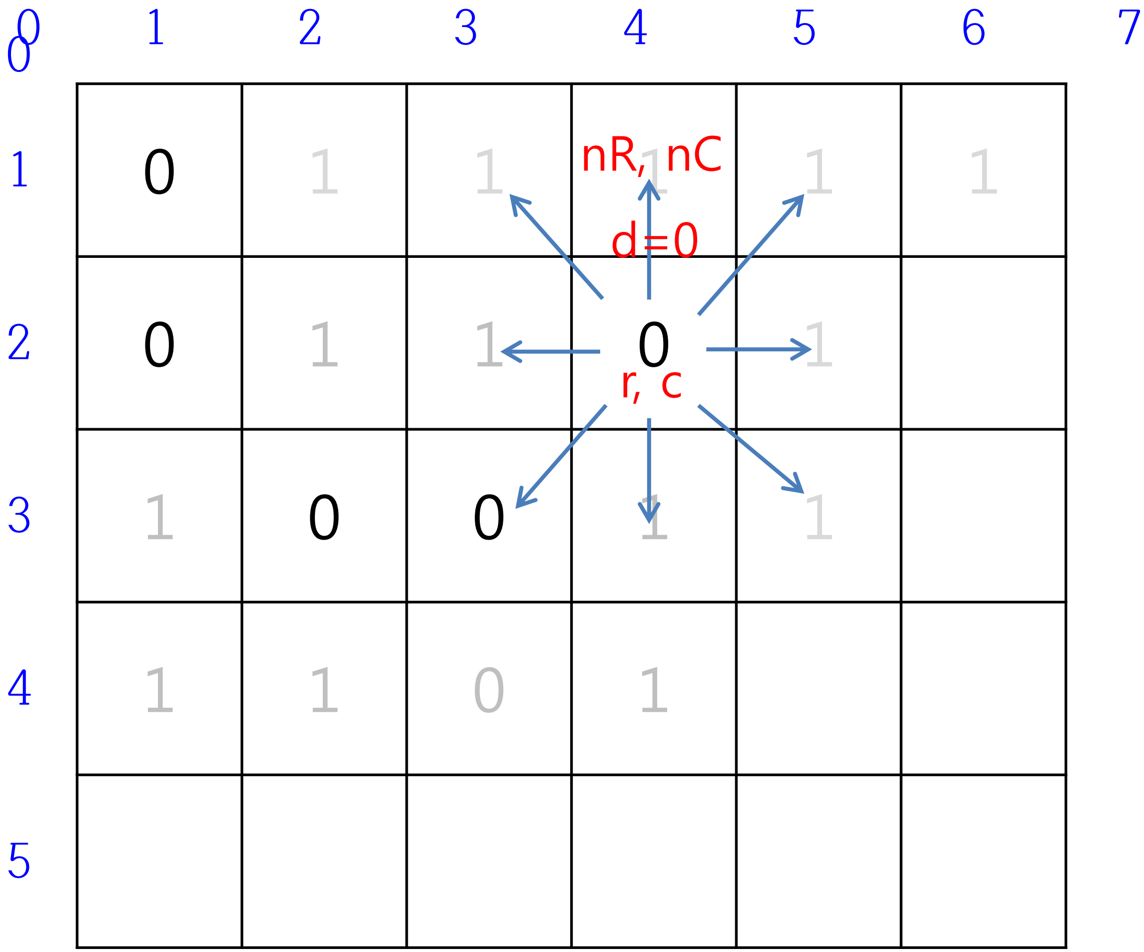
현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
No!

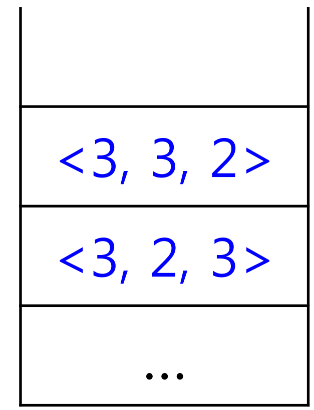


현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
Yes!



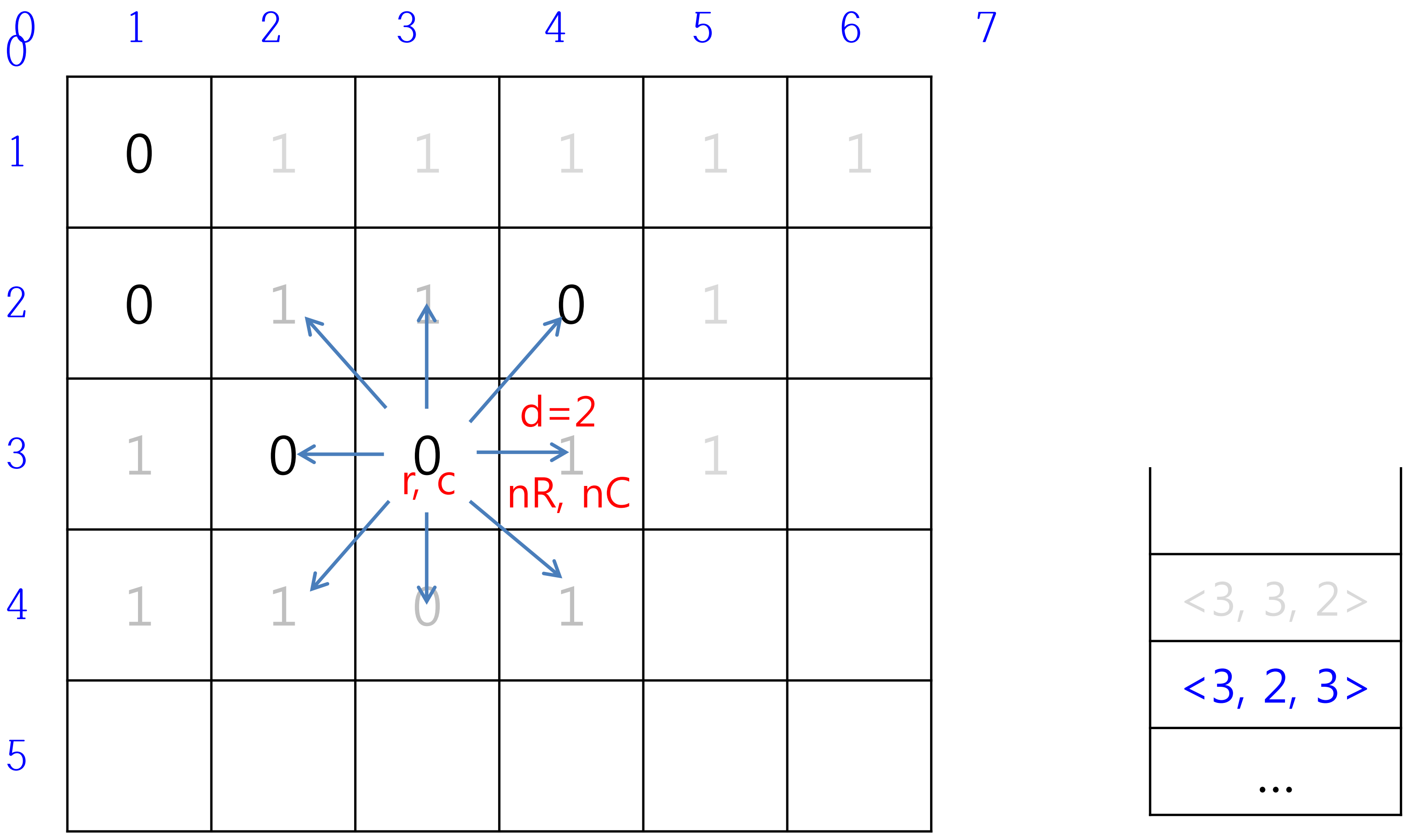
① (nR, nC) 위치를 방문했음을 표시
② (r, c) 에서 다음 번에 검사할 방향 (r, c, ++d) = (3, 3, 2)를 스택에 Push
③ r, c, d를 nR, nC, 0으로 업데이트함 (이동)



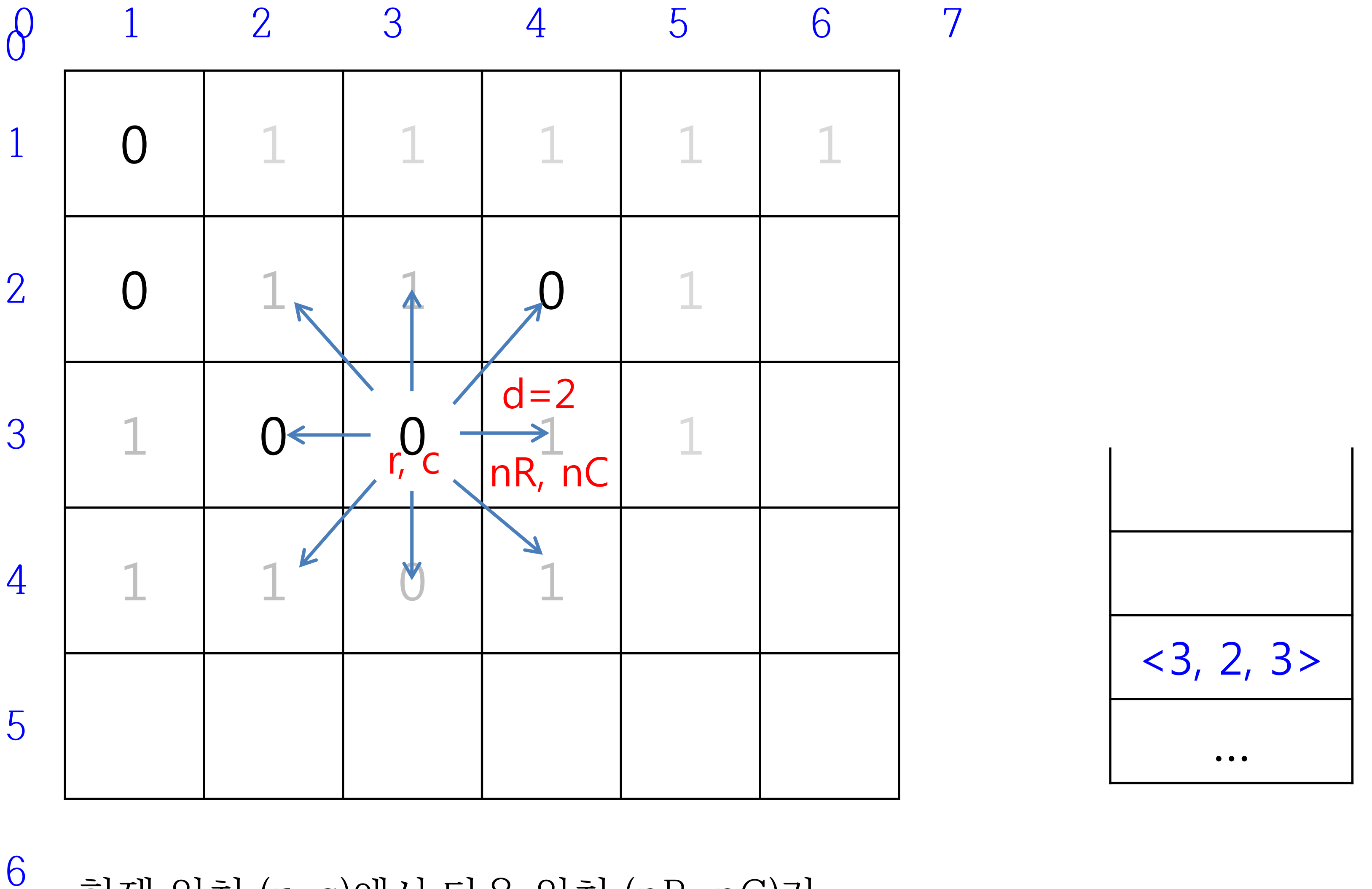


현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
8방향 (d : 0 ~ 7)에 대해 모두 No!

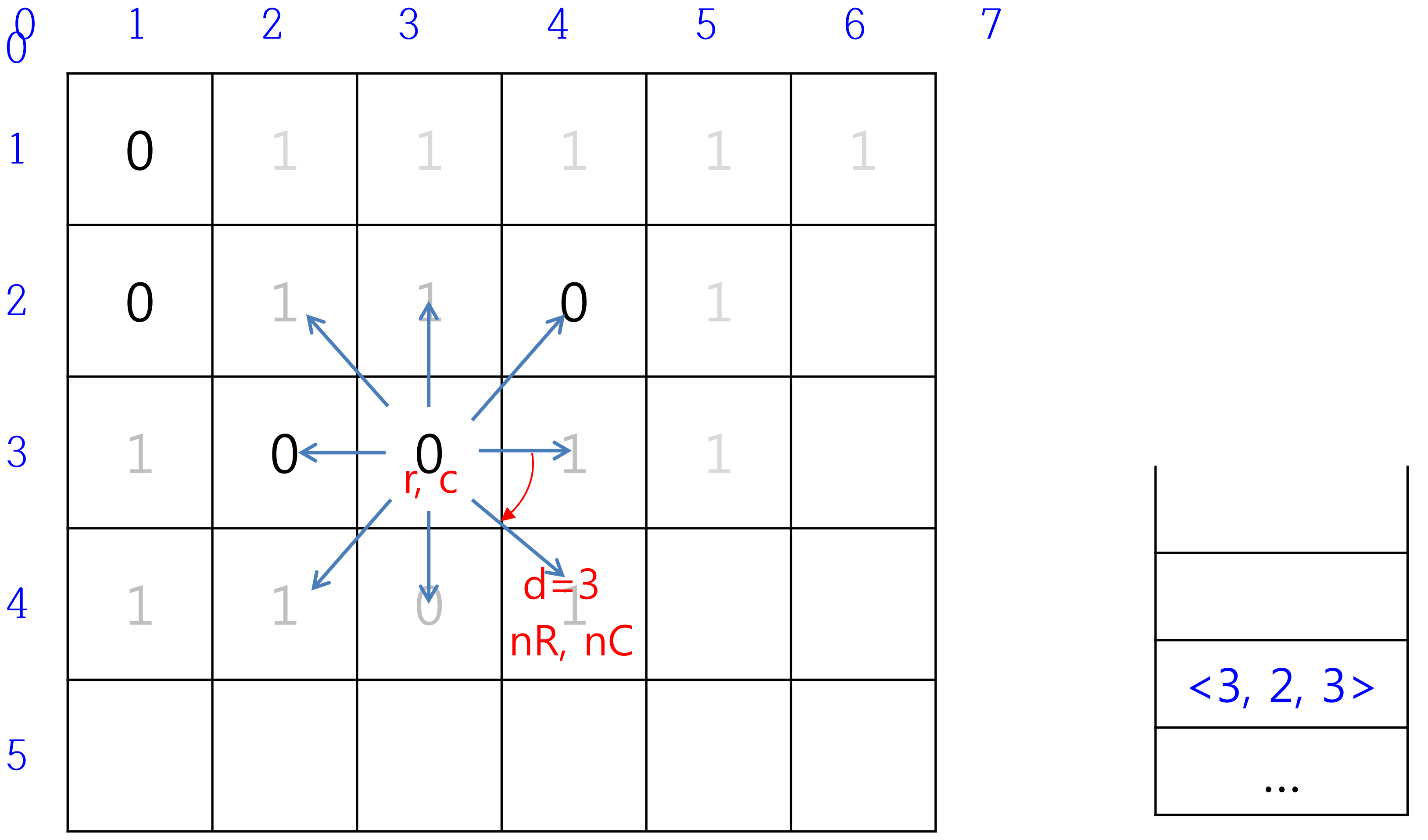
스택이 비어 있지 않고 경로가 발견되지 않았는가? Yes



6 스택에서 pop 하여
현재 위치 r, c, d 를 지정하라.

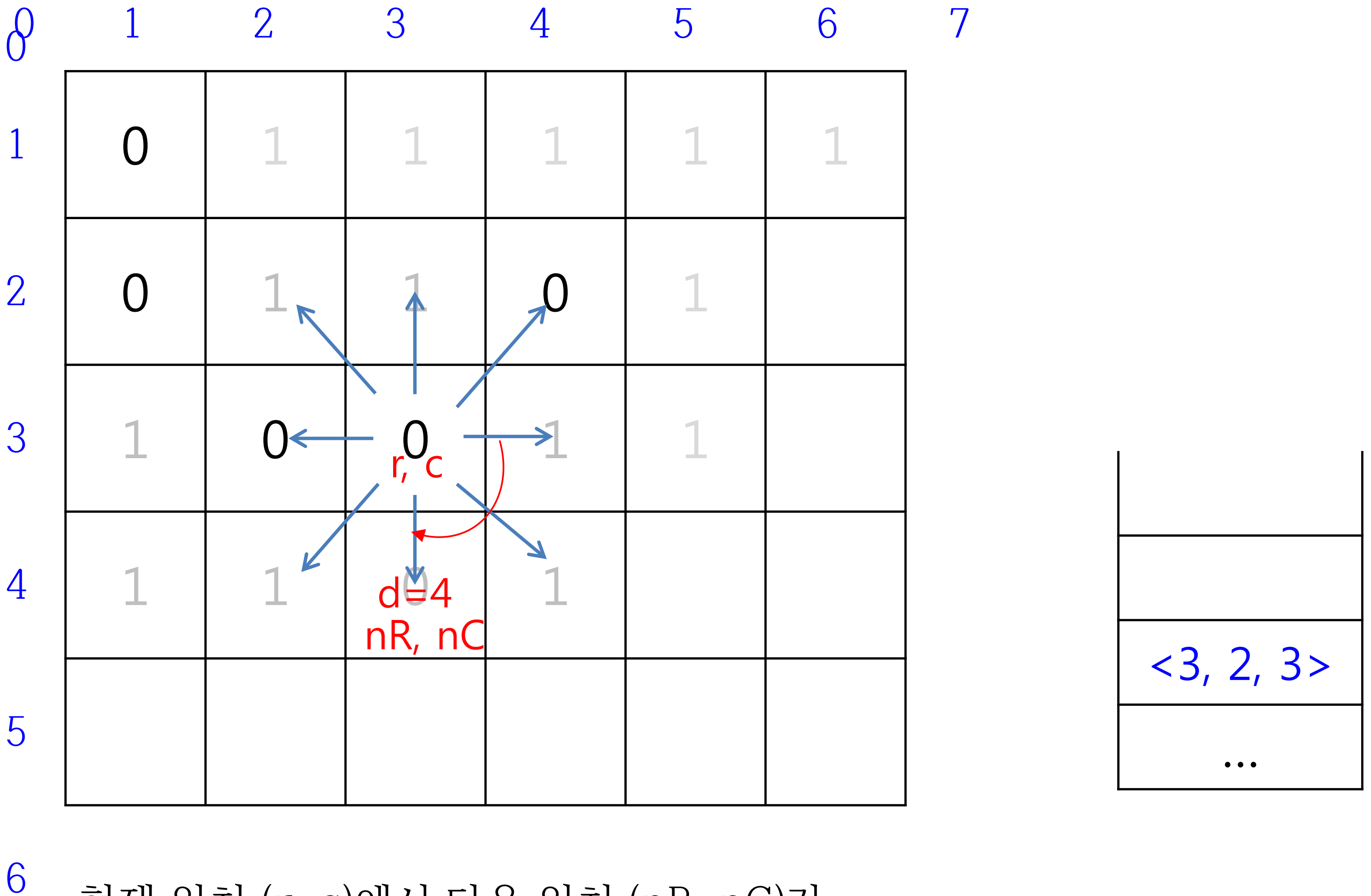


현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
No!

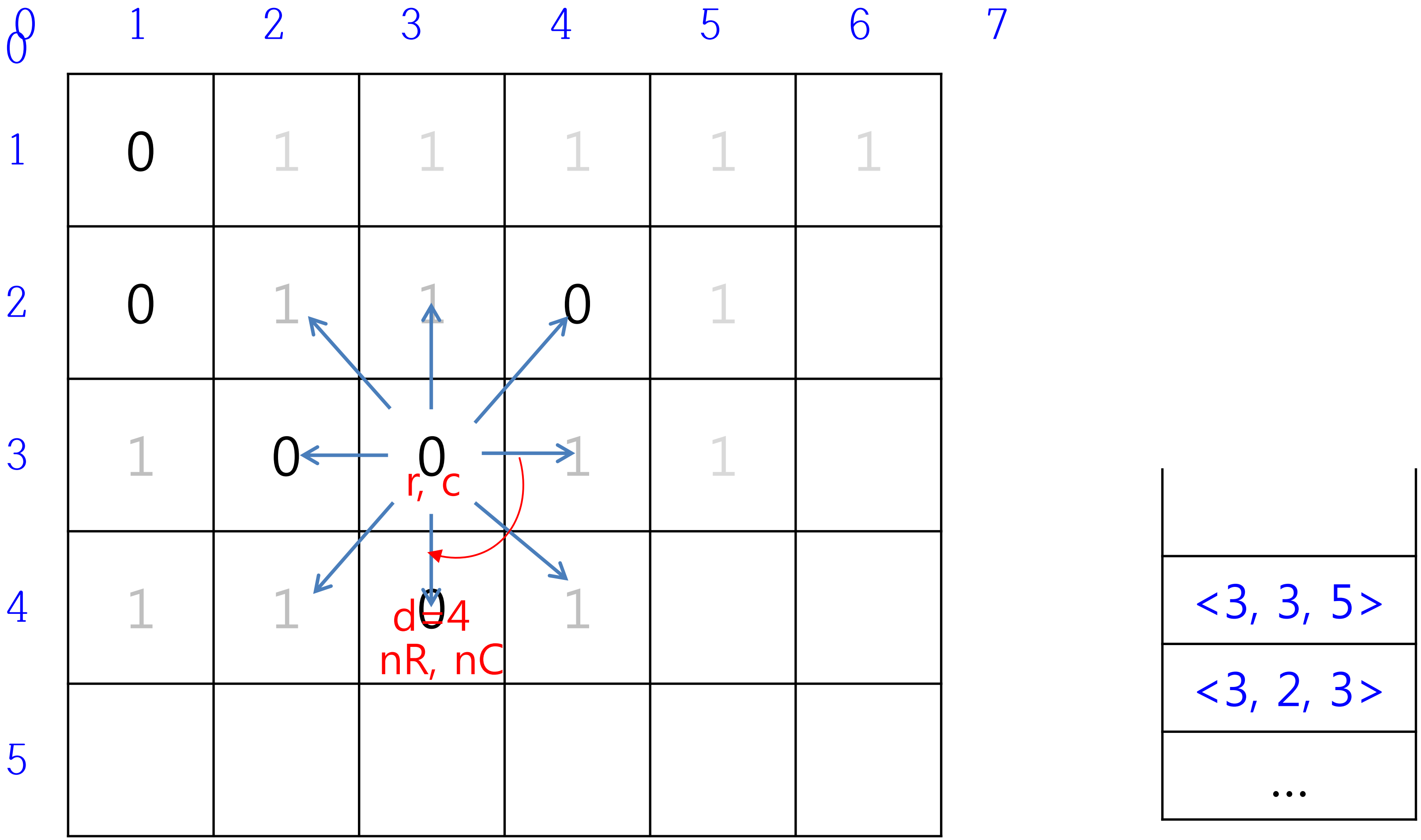


6

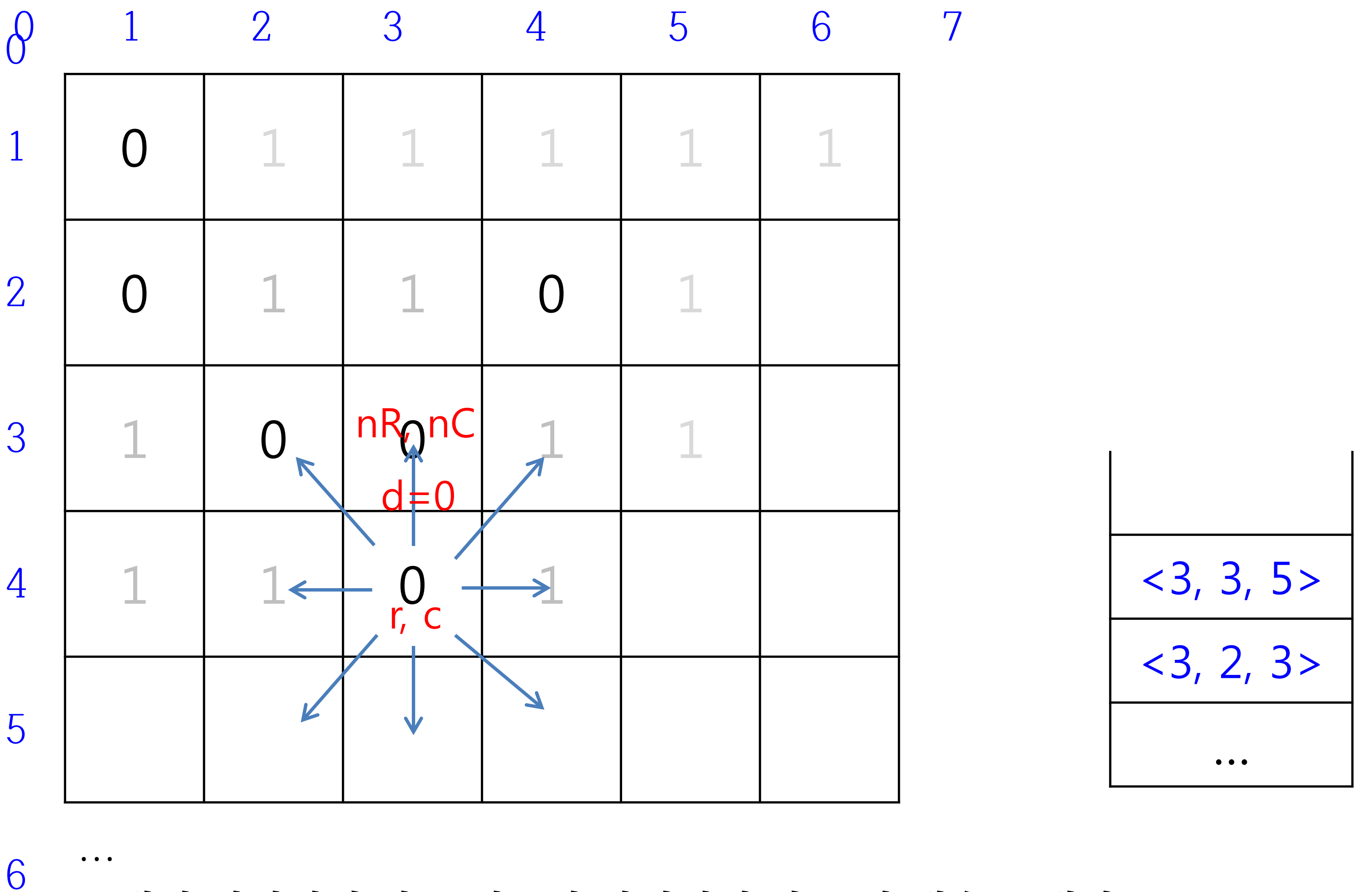
현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
No!



현재 위치 (r, c)에서 다음 위치 (nR, nC)가
이전에 방문하지 않았고 이동 가능한가?
Yes!



① (nR, nC) 위치를 방문했음을 표시
② (r, c) 에서 다음 번에 검사할 방향 (r, c, ++d) = (3, 3, 5)를 스택에 Push
③ r, c, d를 nR, nC, 0으로 업데이트함 (이동)



…

스택이 비어있지 않고 경로가 발견되지 않은 한 계속 수행함

경로가 발견되었다면
① stack[0] 부터 순서대로 출력 ② 현재 위치 출력 ③ 출구 위치 출력

```
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

```c
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 = -1; position.row=1; position.col=1;  position.dir=1; push(position);
    top= 0; stack[0].row= 1; stack[0].col =1; stack[0].dir=1;
    while (top >= 0 && !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;
        }
    }
    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");
}
```

Program 3.12: Maze search function

analysis of path:

- each position within the maze is visited no more than once
- worst case complexity : **O(*mp*)**, where m and p are, respectively, the number of rows and columns of the maze.

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

```c
#define MAX_STACK_SIZE 100
typedef struct {
        short int row;    // x
        short int col;    // y
        short int dir;
} element;
element stack[MAX_STACK_SIZE];
```

Maze (미로) 문제 [4 점]

우리는 “3.5 A Mazing Problem”에서 주어진 maze(미로)에 대한 출구를 구하는 방법을 공부하였다. 이제 우리는 아래 두 maze를 각각 maze1.txt와 maze2.txt에 저장한 후, 각각을 대상으로 입구에서부터 출구에 이르는 경로를 구하고자 한다.

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

maze1.txt

entrance	0	1	1	1	1	1	1	
	0	1	0	0	1	1	1	
	0	0	1	0	1	1	1	
	1	1	1	1	0	0	1	
	0	1	1	0	1	0	1	
	1	0	0	0	1	0	1	
	0	1	1	1	1	1	0	exit

maze2.txt

각 maze에 대하여 다음을 수행하시오.

1. 그 maze를 읽어 들인다.
2. 그 maze에 대한 입구에서부터 출구에 이르는 경로를 출력한다.

3.6 Evaluation of Expressions

3.6.1 Expressions

- Complex expressions
  - $((rear+1==front)||((rear==MAX\_QUEUE\_SIZE-1)\&\& !front))$
  - operators, operands, parentheses

- Complex assignment statements
  - x = a / b – c + d * e – a * c

- The order in which the operations are performed?
  - If 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

Token	Operator	Precedence[1]	Associativity
() [] → .	function call array element struct or union member	17	left-to-right
-- ++	decrement, increment[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

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

Figure 3.12: Precedence hierarchy for C

※ *Parentheses are used to override precedence,* and expressions are always evaluated from the innermost parenthesized expression first.

Within any programming language, there is **a precedence hierarchy** that determines the order in which we evaluate operators. Figure 3.12 contains the precedence hierarchy for C. We have arranged the operators from highest precedence to lowest. Operators with the same precedence appear in the same box.

3.6.2 Evaluating Postfix Expressions

- **Infix notation**
  - binary operator is in-between its two operands

- **Prefix notation**
  - operator appears before its operands

- **Postfix notation**
  - Each operator appears after its operands
  - Used by compiler
  - *Parentheses-free notation*
  - To evaluate expression, we make a single left-to-right scan of it (no precedence hierarchy)
  - Use ***stack***

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

Figure 3.13: Infix and postfix notation

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

Figure 3.14: Postfix evaluation

아래 **Postfix expression** 을 수행하시오
6 2/3-4 2*+

- 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 } token_type;

int stack[MAX_STACK_SIZE];  /* global stack */
char expr[MAX_EXPR_SIZE];   /* global input string (a postfix expression)*/
```

Besides the usual operators, the enumerated type also includes an end-of-string (***eos***) operator.

```
int eval(void)
{
    /* evaluate a postfix expression, expr, maintained as a global variable.
     '\0' is 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
     */
    token_type tokenType;
    char tokenChar;
    int opdl, opd2;
    int tokenIndex = 0; /* counter for the expression string */
    int top = -1;
    while (1) {
        tokenType= getToken(&tokenChar, &tokenIndex);
        if (tokenType == eos) break;
        if (tokenType == operand) push(tokenChar - '0');
        else {
                /* pop two operands, perform operation, and
                   push result to the stack */
                opd2 = pop();
                opdl = pop();
                switch(token_type) {
                        case plus: push(opdl+opd2);   break;
                        case minus: push(opdl-opd2);break;
                        case times: push(opdl*opd2); break;
                        case divide: push(opdl/opd2);break;
                        case mod: push(opdl%opd2);
                }
        }
    }
    return pop();
}
```

Program 3.13: Function to evaluate a postfix expression

```
precedence getToken(char *tokenChar_ptr, int *tokenIndex_ptr)
{
    /* 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 * /
    *tokenChar_ptr= expr[(*tokenIndex_ptr)++];
    switch (*tokenChar_ptr) {
                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 * /
    }
}
```

Program 3.14: Function to get a token from the input string

token_char-'0' makes a single digit integer.
'0' : ASCII value of 48

3.6.3 Infix to Postfix

Produce a postfix expression from an infix one.

Input : a/b -c +d*e -a *c

Output: ab/c-de*+ac*-

Idea:

Scan the infix expression left-to-right.

During this scan,

- operands are passed to the output expression as they are encountered.
- the order in which the operators are output depends on their precedence.
  - ◆ Since we must output the higher precedence operators first, we save operators until we know their correct placement.
  - ◆ A stack is one way of doing this, but removing operators correctly is problematic.

Example 3.3 [*Simple expression*]:

- Input : a+b*c
- Output : abc*+

Infix-to-Postfix Transformation Rule:

Token generation by ***scanning*** left to right.

- ***Operands*** are passed to the output expression.
- ***Operators*** are stacked and unstacked *by their precedence*.

if (isp < icp) then stack the incoming operator; // 예: + < *; * < (,
else {
while (isp >= icp) { pop the operator; } // 예: + >), *>+, * = *
if the operator is either ')' or end-of-string(eos) then {};
else stack the operator;
}

- the end-of-string(eos)
- isp(In-stack precedence) and icp(Incoming precedence)

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	*		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc*+*

Incoming operator

In-stack operator at top

push

Figure 3.15: Translation of $a+b*c$ to postfix

Example 3.4 [Parenthesized expression]:

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

Infix-to-Postfix Transformation Rule:

Token generation by ***scanning*** left to right.

- ***Operands*** are passed to the output expression.
- ***Operators*** are stacked and unstacked *by their precedence*.

  if (isp < icp) then stack the incoming operator; // 예: + < *; * < (,
  else {
      while (isp >= icp) { pop the operator; } // 예: + >), *>+, * = *
      if the operator is either ')' or end-of-string(eos) then {};
      else stack the operator;
  }

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
*	*			0	*a*
(>	*	(push		1	*a*
b	*	(		1	*ab*
+ >	*	(	+ push	2	*ab*
c	*	(	+	2	*abc*
)	* pop→print			0	*abc* +
* =	* push			0	*abc* +*
d	*			0	*abc* +**d*
eos				0	*abc* +**d**

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

Right parenthesis is never put on the stack.

- isp(In-stack precedence) and icp(Incoming precedence)

```
precedence stack[MAX_STACK_SIZE];

/* isp and icp arrays – index is value of token_type
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 };
```

Q. *isp* [*plus*] ?

Because we want unstacking to occur when we reach the end of the string, we give the *eos* token a low precedence (0). These precedences suggest that we remove an operator from the stack only if its instack precedence is greater than or equal to the incoming precedence of the new operator.

Infix-to-Postfix Transformation Rule:

Token generation by ***scanning*** left to right.

- ***Operands*** are passed to the output expression.
- ***Operators*** are stacked and unstacked *by their precedence*.

```
if (isp < icp) then stack the incoming operator; // 예: + < *;  * < (,
else {
    while (isp >= icp) { pop the operator; } // 예: + > ), *>+, * = *
    if the operator is either ')' or end-of-string(eos) then {};
    else stack the operator;
}
```

```
int stack_top = 0;
void postfix(void)
{/* output the postfix of the expression.
  The expression string, the stack, and top are global */
    char tokenChar;
    token_type tokenType;
    int tokenIndex = 0;
    stack_top = 0;
    stack[0] = eos;
    while(1) {
        tokenType= getToken(&tokenChar, &tokenIndex);
        if (tokenType == eos) break;
        if (tokenType == operand) printf("%c", tokenChar);
        else if (tokenType == rparen) {
                /* unstack tokens until left parenthesis */
                while (stack[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[stack_top]] >= icp[tokenType]) printToken(pop());
                push(tokenType);
        }
    }
    while ((tokenType= pop()) != eos) printToken(tokenType);
    printf("\n");
}
```

Program 3.15: Function to convert from infix to postfix

- Analysis of postfix
  - 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 : $\theta(n)$
  - So, the complexity of function *postfix* is $\theta(n)$.

Infix-to-Postfix 변환과 Postfix Expression의 연산 문제 [4 점]

우리는 "3.6 Evaluation of Expressions"에서 주어진 infix expression을 Postfix expression으로 변환하는 방법과 Postfix Expression을 연산하는 방법을 공부하였다. 우리는 모든 피연산자(operand)는 단숫자(single digit. 예를 들어, 3 5 2 등)이고 연산자(operator)는 binary operator인 +, -, *, /, %의 다섯 가지로 제한하며 소괄호('('와 ')')로 묶는 것이 가능하고 모든 연산들은 정수 연산들이라 가정한다.

다음을 무한 반복 수행하시오.

1. 하나의 Infix-Expression을 키보드로 부터 읽어 들인다. 단, 그 infix-expression에 오류가 없어야 한다.
2. 그 Infix-Expression에 대한 Postfix-Expression을 구하여 출력한다.
3. 그 Postfix-Expression을 수행한 결과를 출력한다.

예

Infix notation	Postfix notation	result value
5 * 6 % 7	→ 5 6 * 7 %	→ 2
(5+6)*8+9/(3+5*8)+7	→ 5 6 + 8 * 9 3 5 8 * + / + 7 +	→ 95
3 * (4 + 5) / 6	→ 3 4 5 + * 6 /	→ 4
1 - (5 - 2) * 4 / 2	→ 1 5 2 - - 4 * 2 /	→ -5