Chap 3. Stacks and Queues (2)

Contents

Contents

3.4 Circular Queues Using Dynamically Allocated Arrays

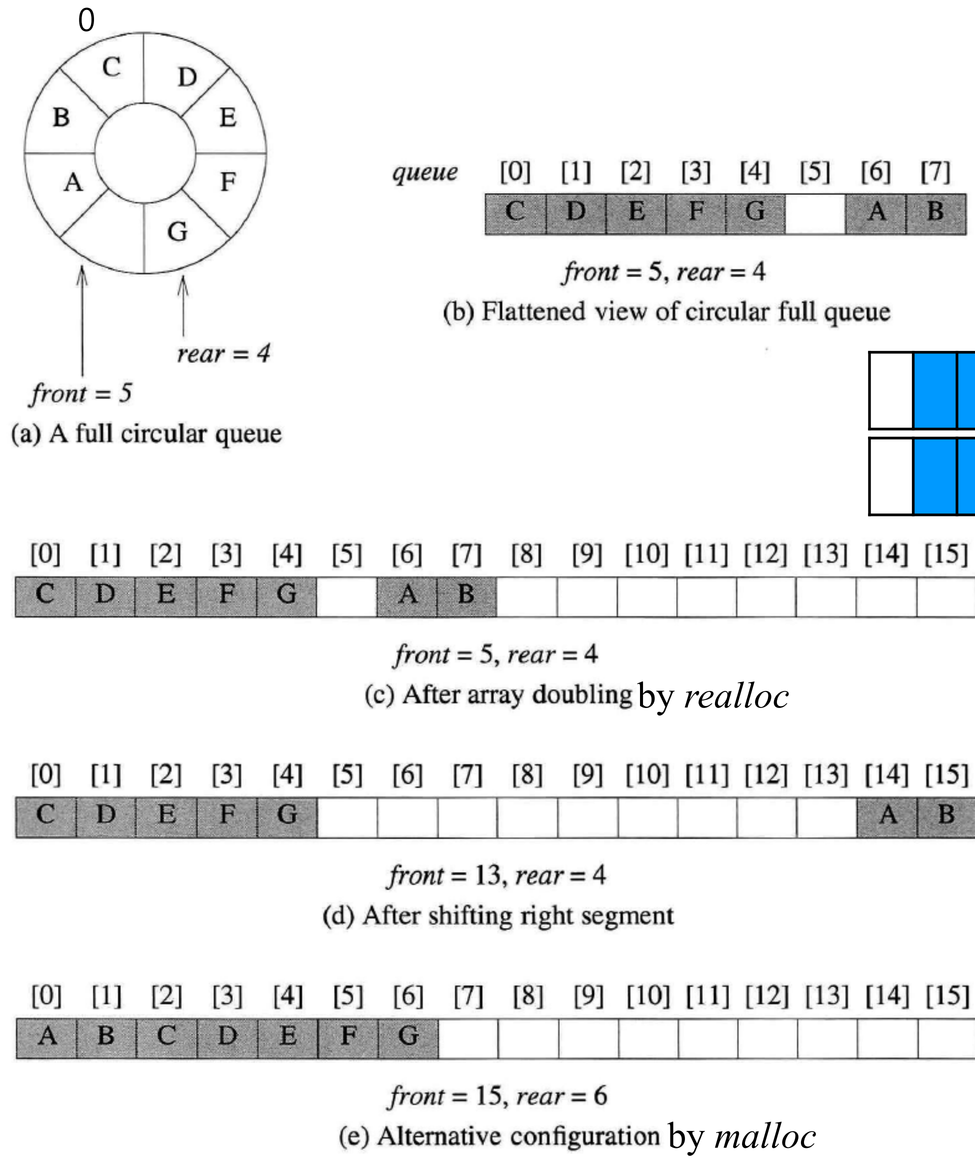
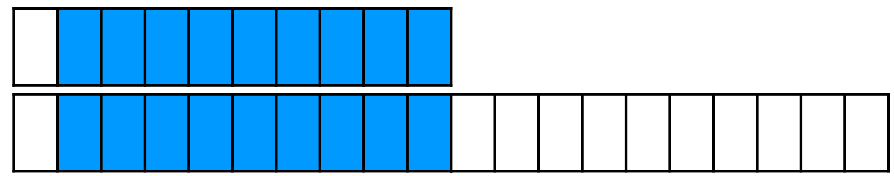


Figure 3.7: Doubling queue capacity



최악의 경우 이동

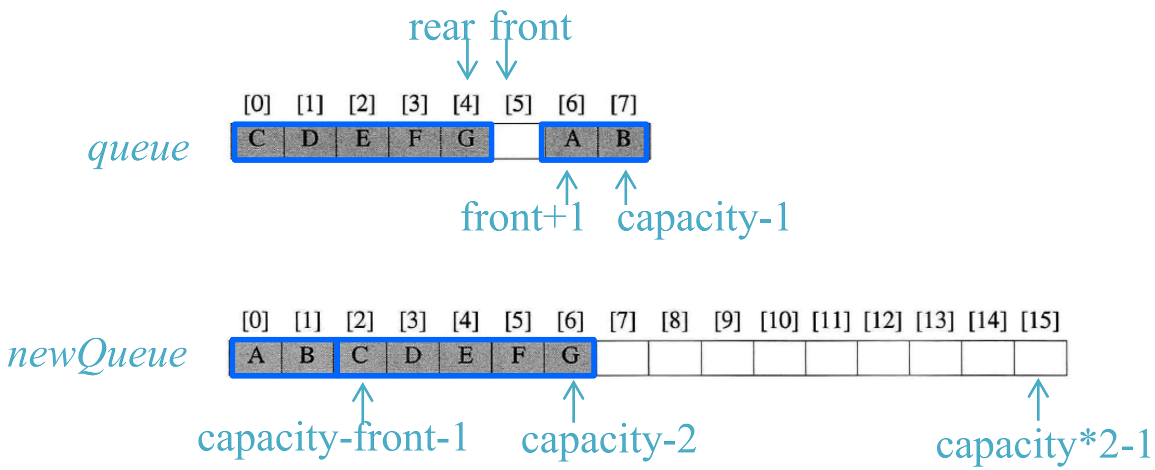
(case 1) (b)→(c)→(d)
(case 2) (b)→(e) , Program 3.10

- Figure 3.7 (b)→(e)

(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.

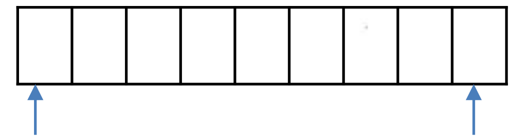
(3) Copy the first segment (i.e., the elements *queue* [0] through *queue* [*rear*]) to positions in *newQueue* beginning at *capacity* −*front*−1.

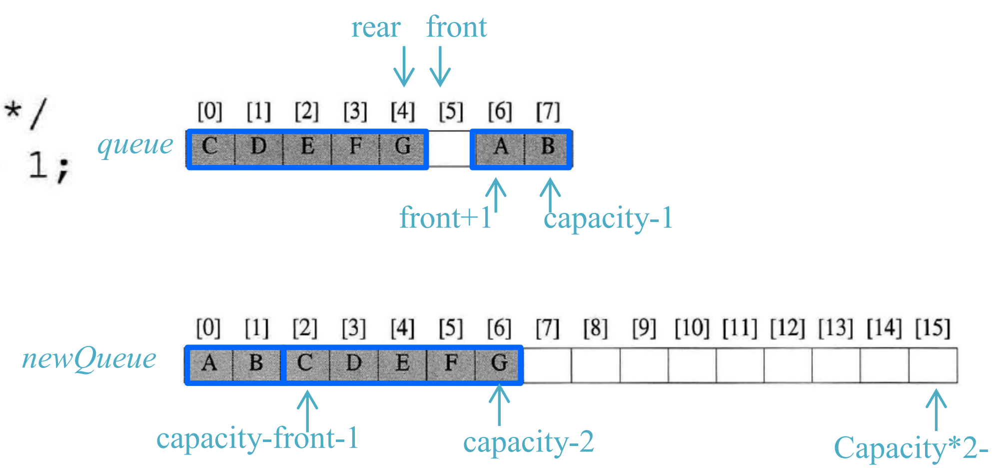


```
void queueFull()
{   int start;
    /* allocate an array with twice the capacity */
    element* newQueue;
    MALLOC(newQueue, 2 * capacity * sizeof(*queue));

    /* copy from queue to newQueue */
    start = (front+1)  % capacity;
①   if (start  <  2)
        /* no wrap around */
        copy(queue+start, queue+start+capacity-1, newQueue);
②   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;
}
```



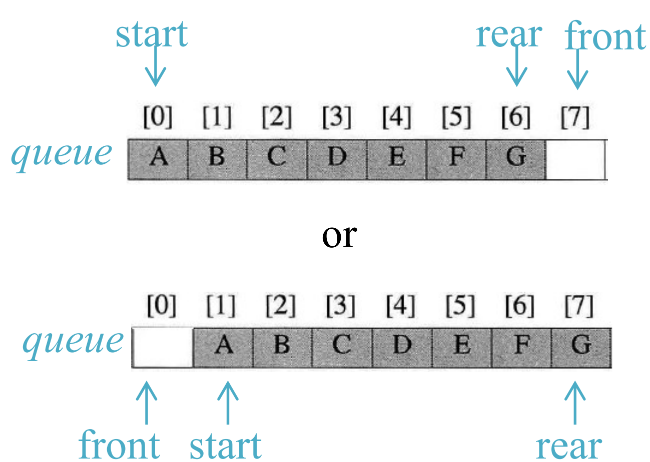


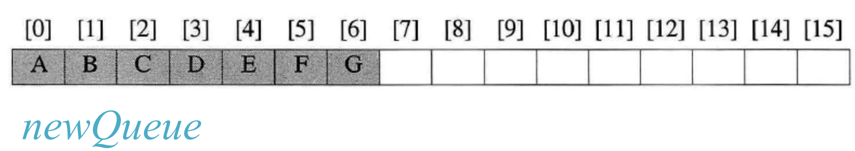
Program 3.10: Doubling queue capacity < Figure 3.7 (b)→(e) >

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

```
int start = (front+1)  % capacity;
if (start  <  2)
    /* no wrap around */
    copy(queue+start, queue+start+capacity-1, newQueue);
```

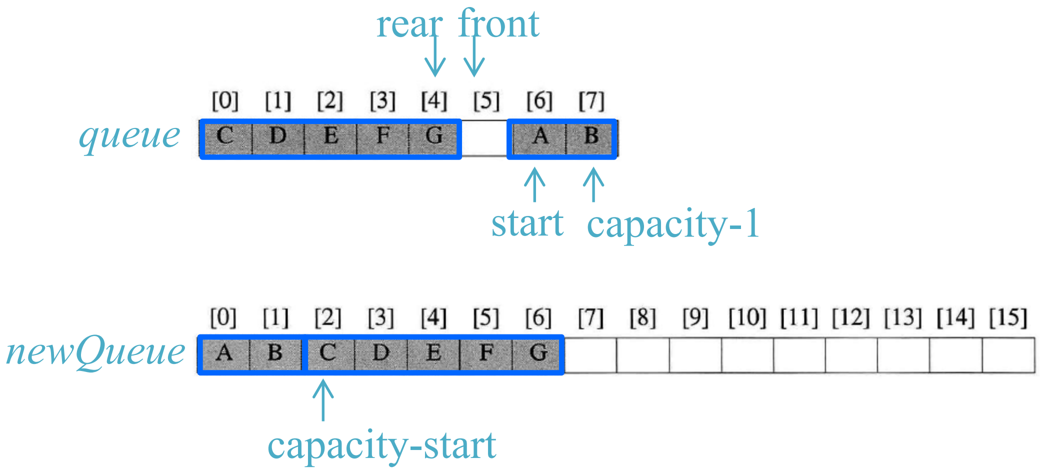
rear+1



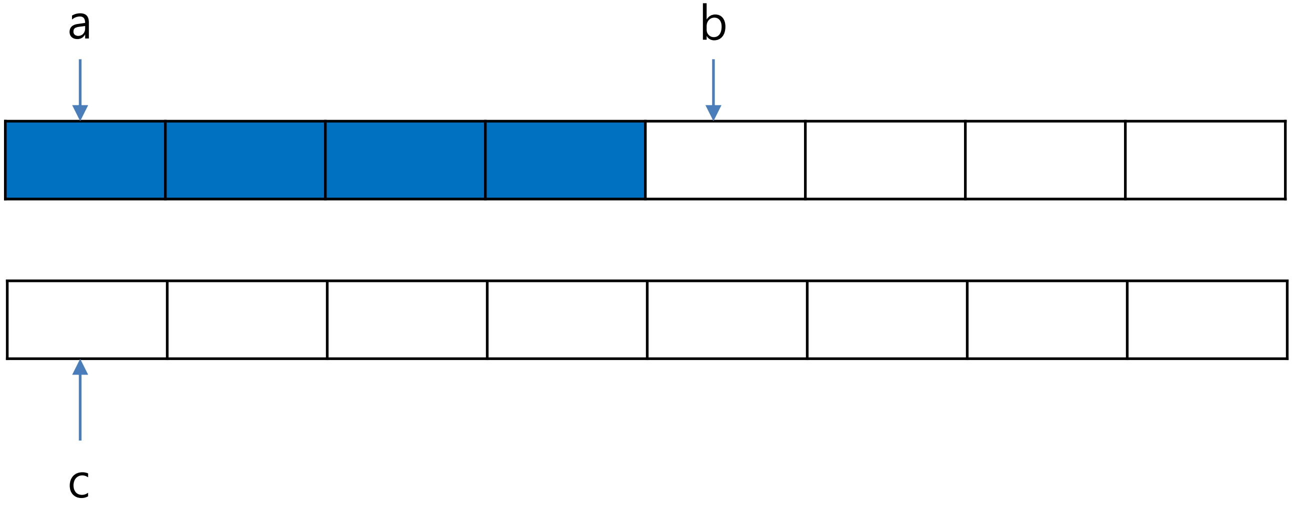


- ②

```
else
{/* queue wraps around */
    copy(queue+start, queue+capacity, newQueue);
    copy(queue, queue+rear+1, newQueue+capacity-start);
}
```



```
void copy( element *a, element *b, element *c)
{
while( a != b )
*c++ = *a++;
}
```



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

Add to a circular queue

3.5 A Mazing Problem

- Rat in a maze
  - Experimental psychologists train rats to search mazes for food

- For us, a nice application of *stacks*
  - Searching the maze for an entrance to exit path.

Implementation in C

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

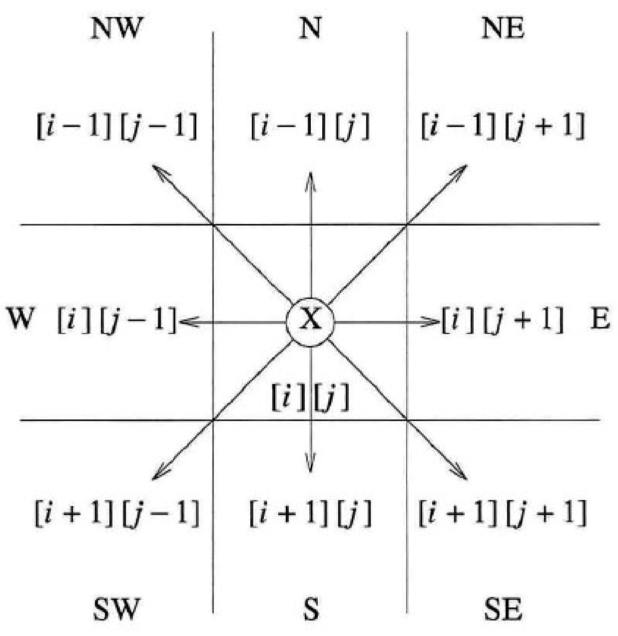
- Assumptions
  - Rat starts at the top left,
    exits at the bottom right

maze[12,15]

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

- The current location of the rat in the maze
  - *maze[row][col]*

- The possible *8 moves* from the current position



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

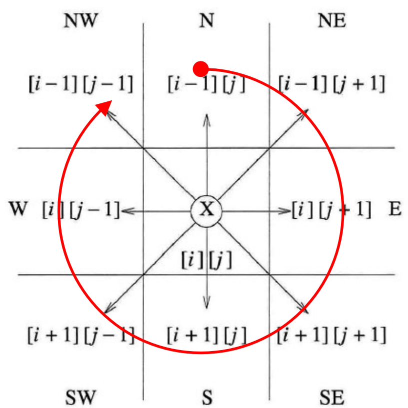
- To avoid checking for boarder conditions
  - We can surround the maze by a boarder of ones.

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

- Predefining the 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;
        short int horiz;
        } offsets;
offsets move[8]; /* array of moves for each direction */
```

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

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

- Since we do not know which choice is best,
  - we save our current position and
  - arbitrarily pick a possible move.

- By saving our current position,
  - we can *return to it* and *try another path* if we take a hopeless path.

- We examine the possible moves
  - starting from the north and moving clockwise.

- Maintaining a second 2D array, ***mark***
  - to record the maze positions already checked

  - initialize the *mark*'s entries to *zero*
  - When we visit a position *maze[row][col]*, we change *mark[row][col]* to *one*

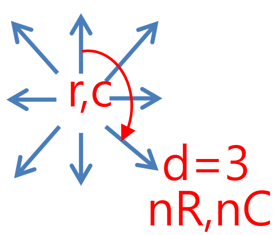
1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	1
1	0	0	1	0	1	1	1	1
1	1	1	1	1	0	0	1	1
1	0	1	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1
1	0	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1

maze

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

mark

현 위치 (r, c)에서 **탐색방향<8** 이고 **경로가 발견되지 않은** 한 다음을 반복
 현 위치 **(r, c)** 에서 계산한 다음 위치 **(nR, nC)**에 대해
 ① if **출구인 경우**
 경로발견!
 ② else if **이동가능하고 이전에 방문하지 않은 경우**
 push(백트래킹 후 탐색할 위치와 방향) // push(r, c, d++)
 다음위치 방문했음을 표시
 다음위치로 이동
 ③ else
 탐색방향증가 // d++
 현 위치에서 **탐색방향==8 이면 스택에서 돌아갈 위치를 가져와서**
 위의 과정을 반복, 스택이 empty이면 경로 발견 실패



Q1. 언제 push를 수행하는가?

Q2. 언제 pop을 수행하는가?

Maze Search : Example

maze

entrance [1,1]

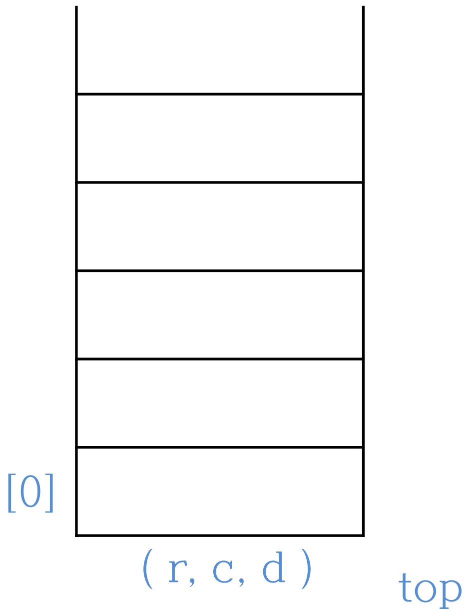
0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

mark

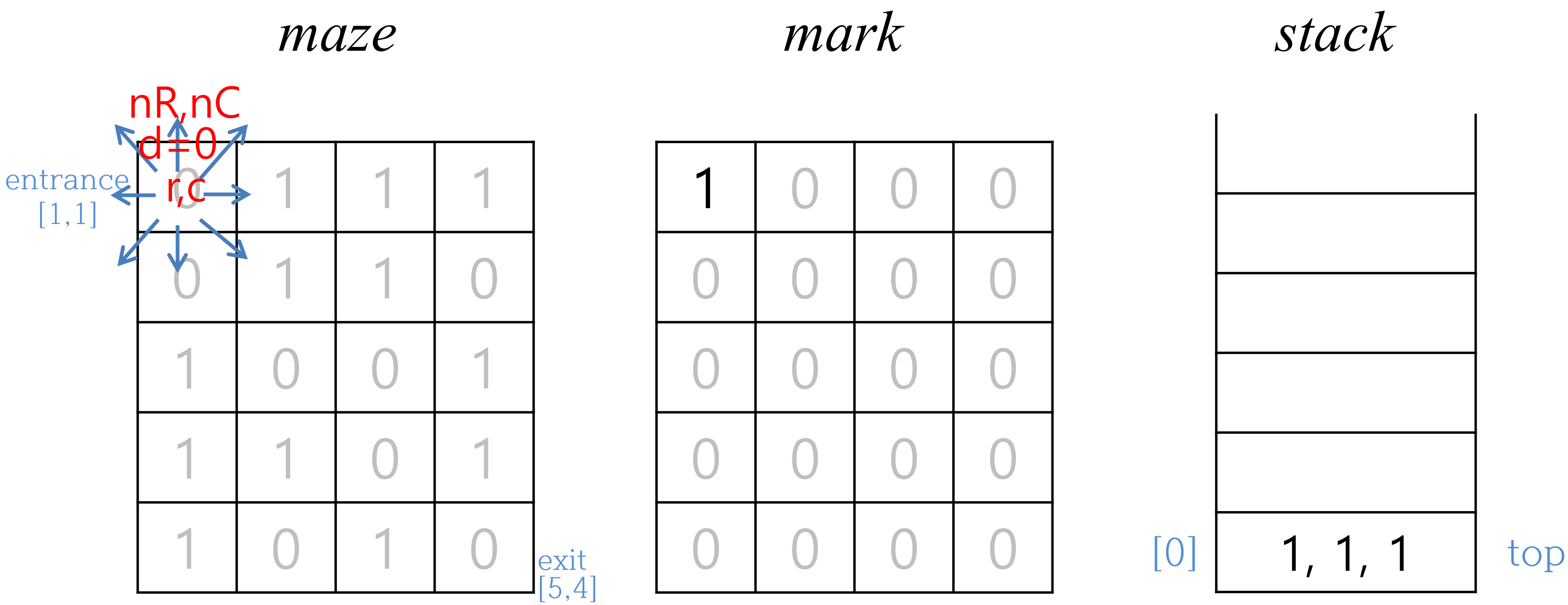
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Program initialization

Maze Search : Example



Initialization of function path()

(1, 1) 방문
push(1, 1, 1)

Maze Search : Example

maze

nR,nC

d=1

entrance [1,1]

r,c

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

exit [5,4]

mark

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

stack

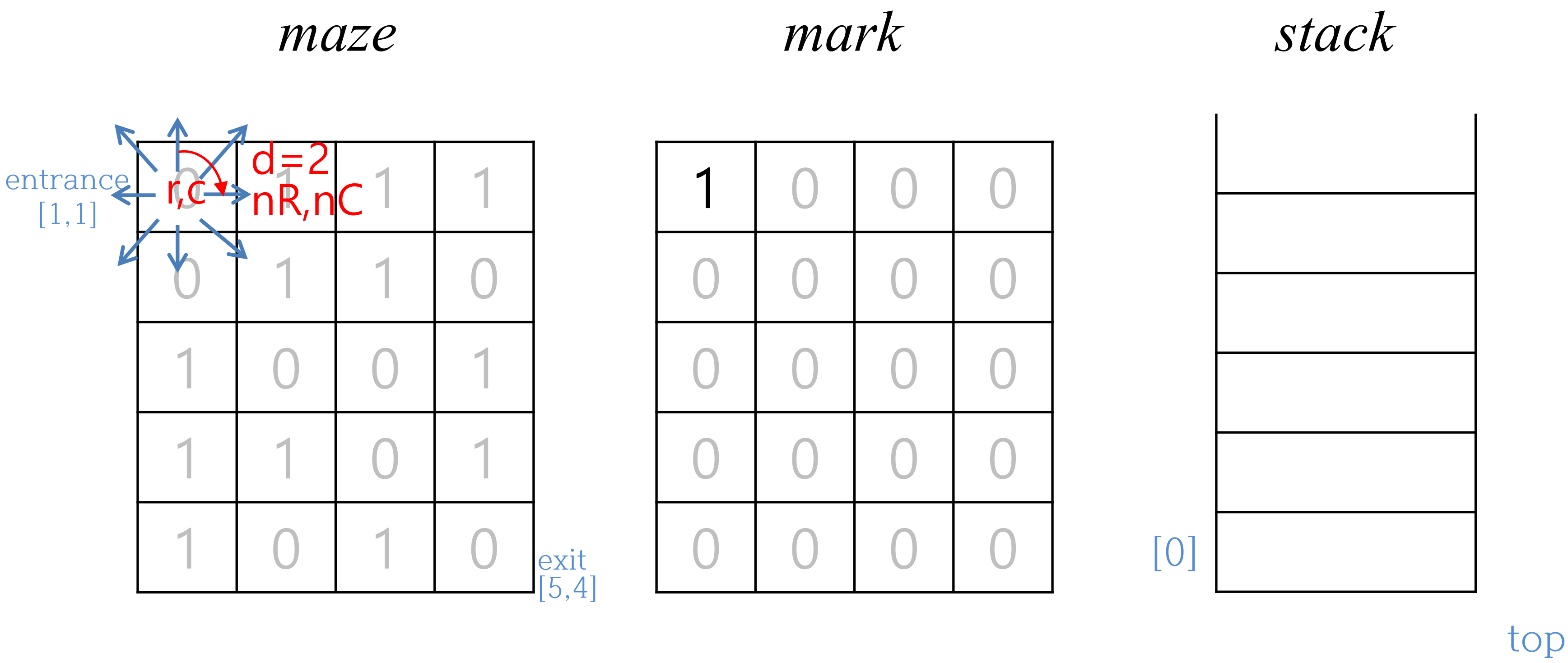
[0]

top

pop()

(r, c, d) = (1, 1, 1)

Maze Search : Example



Maze Search : Example

maze

entrance [1,1]

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

r,c

d=3

nR,nC

exit [5,4]

mark

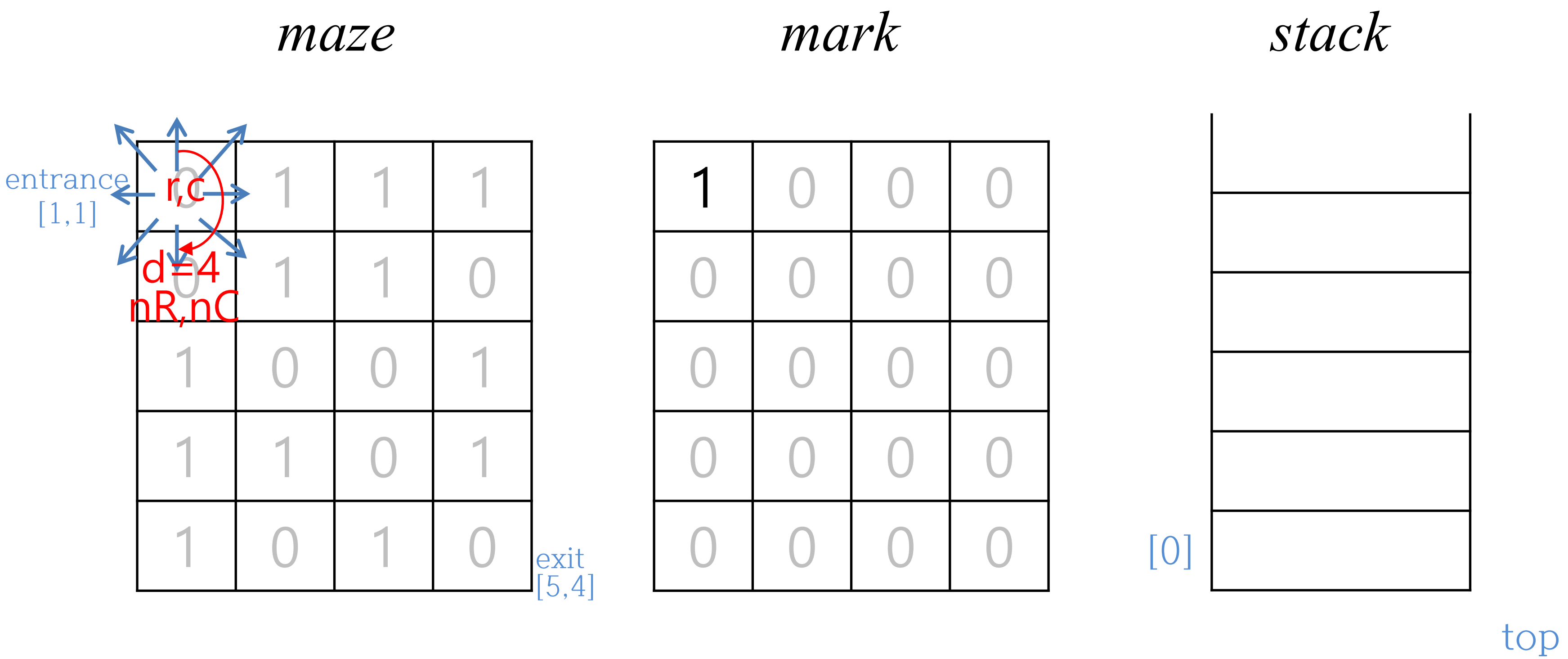
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack

[0]

top

Maze Search : Example



Maze Search : Example

maze

entrance [1,1]

d=0

nR,nC

r,c

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

exit [5,4]

mark

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

stack

[0]	1, 1, 5	top

push(1, 1, 5)
(1, 2) 방문, (r, c, d) = (1, 2, 0)

Maze Search : Example

maze

entrance [1,1]

nR,nC

d=1

r,c

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

exit [5,4]

mark

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

stack

[0]	1, 1, 5	top

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

r,c

d=2

nR,nC

mark

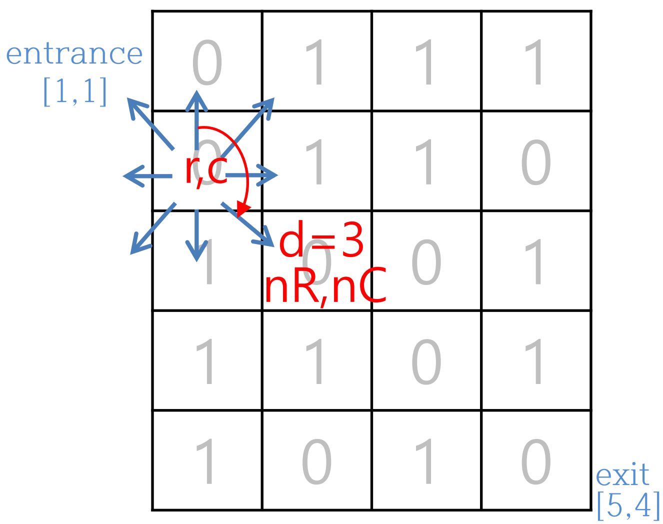
1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack

[0]	1, 1, 5	top

Maze Search : Example

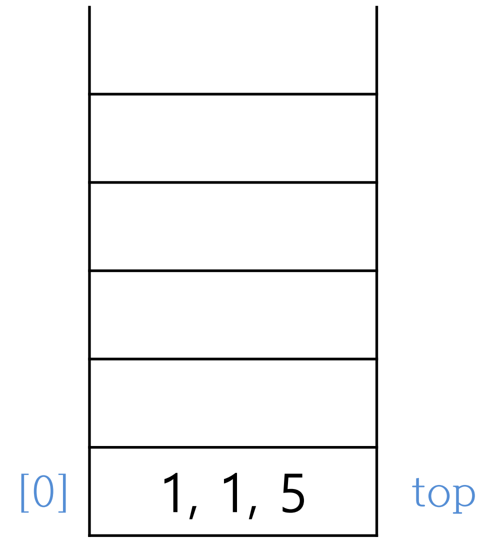
maze



mark

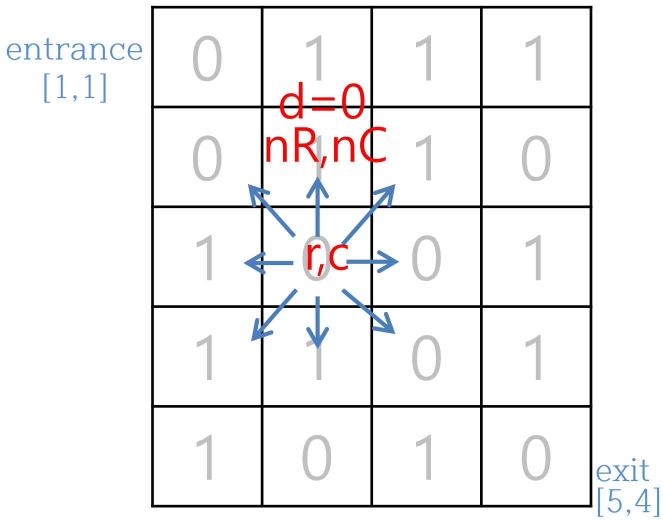
1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example

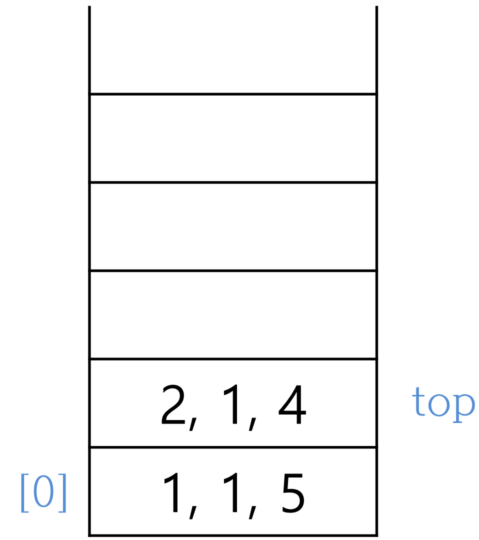
maze



mark

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

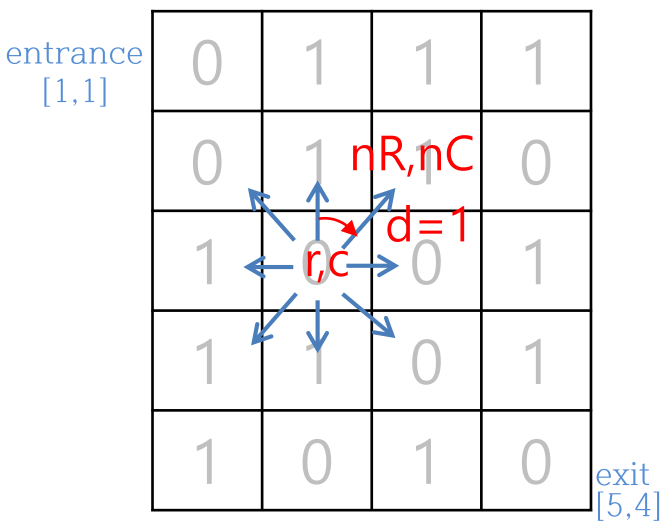
stack



push(2, 1, 4)
(3, 2) 방문, (r, c, d) = (3, 2, 0)

Maze Search : Example

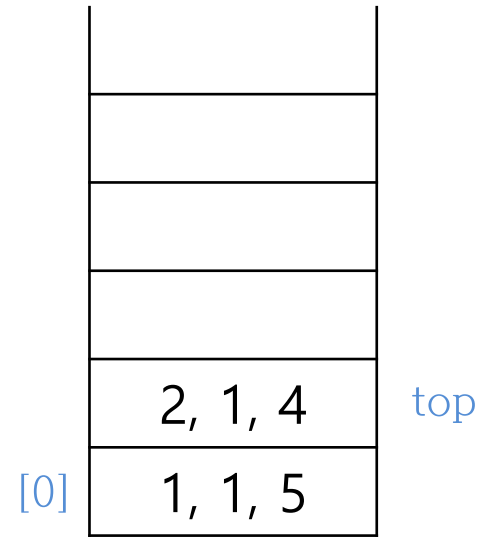
maze



mark

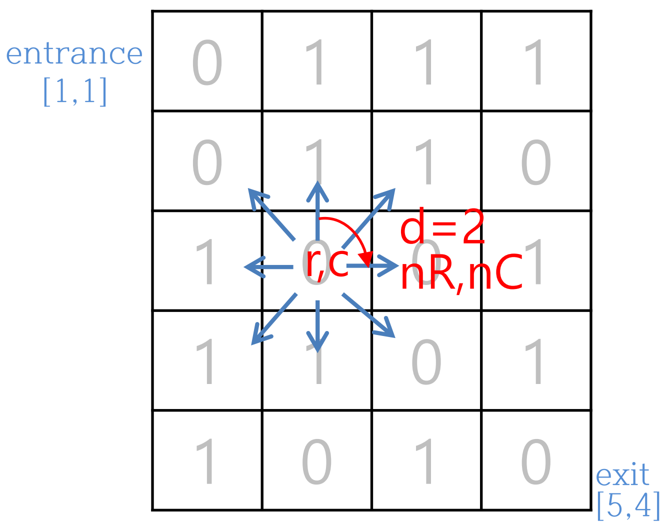
1	0	0	0
1	0	0	0
0	1	0	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example

maze



mark

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

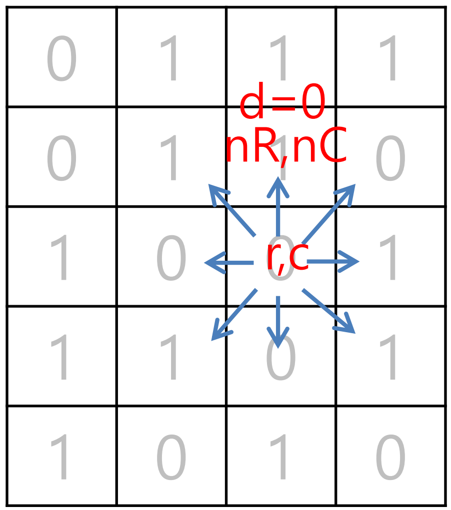
stack

	2, 1, 4	top
[0]	1, 1, 5	

Maze Search : Example

maze

entrance
[1,1]



exit
[5,4]

mark

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

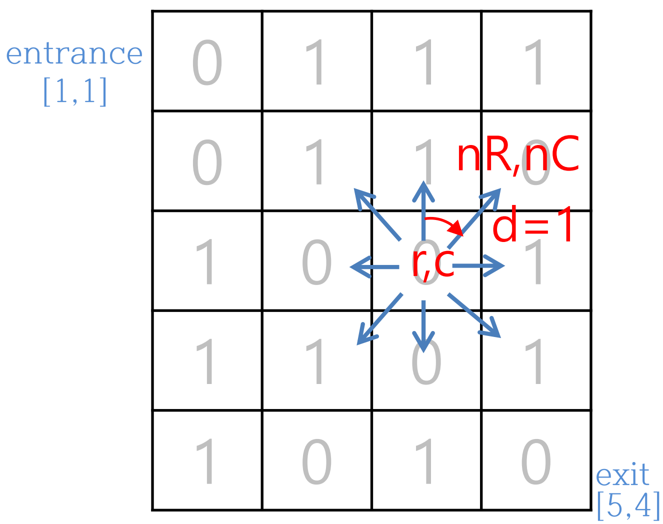
stack

top	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

push(3, 2, 3)
(3, 3) 방문, (r, c, d) = (3, 3, 0)

Maze Search : Example

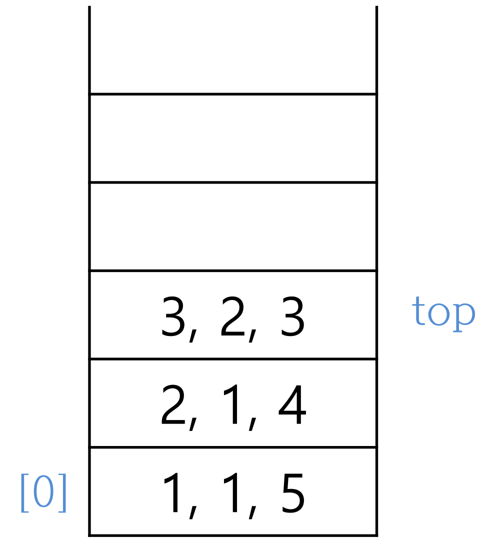
maze



mark

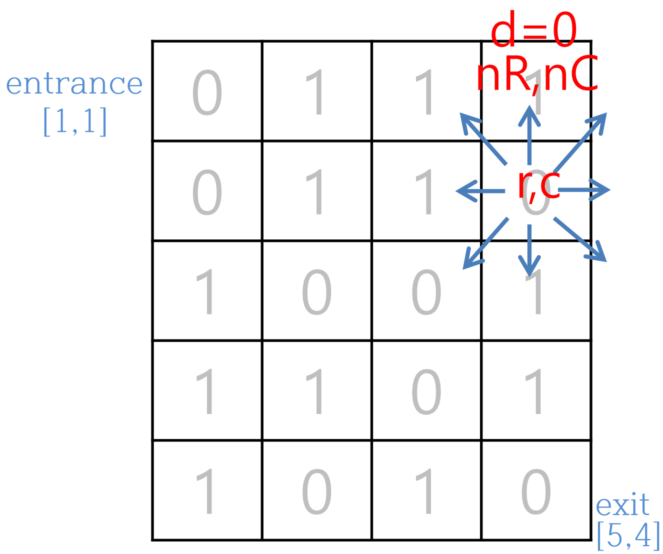
1	0	0	0
1	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example

maze



mark

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

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

push(3, 3, 2)
(2, 4) 방문, (r, c, d) = (2, 4, 0)

Maze Search : Example

maze

entrance [1,1]

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

nR,nC

d=1

r,c

exit [5,4]

mark

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

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
[0]	1, 1, 5

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

r,c

d=2

nR,nC

mark

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

stack

top	3, 3, 2
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze

entrance [1,1]

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

r,c

d=3

nR,nC

exit [5,4]

mark

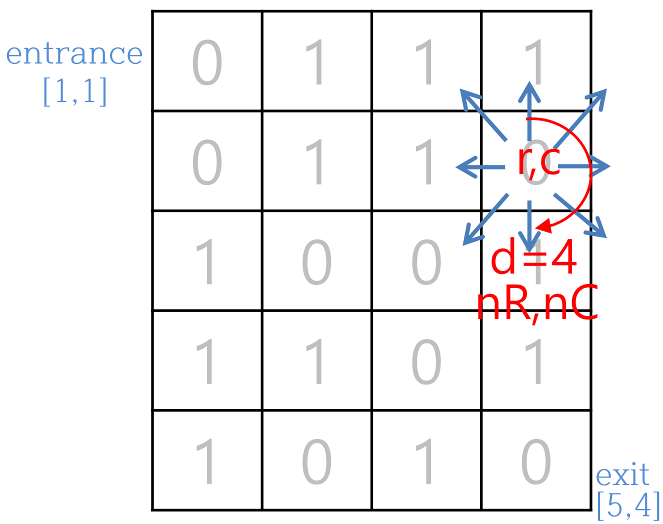
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

Maze Search : Example

maze



mark

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

stack

	3, 3, 2	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

r,c

d=5

nR,nC

mark

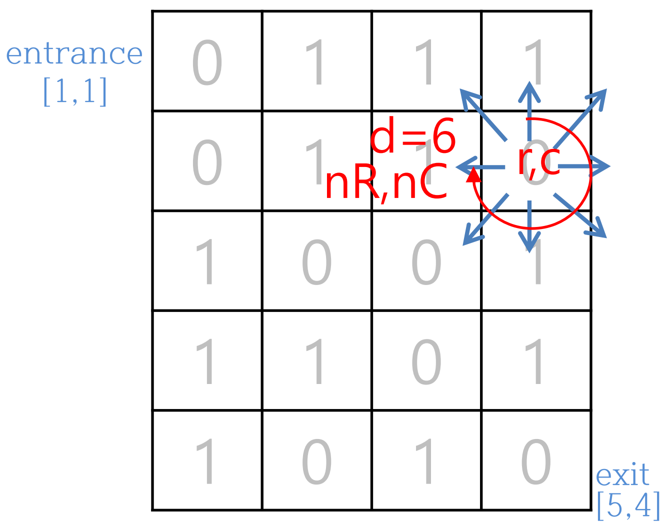
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

Maze Search : Example

maze



mark

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

stack

	3, 3, 2	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze

entrance [1,1]

d=7
nR,nC
r,c

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

exit [5,4]

mark

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

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

d=8

r,c

mark

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

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

(d < 8 && !found) ? No!

(top > -1 && !found) ? Yes!

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

r,c

d=2

nR,nC

mark

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

stack

top	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

pop()
(r, c, d) = (3, 3, 2)

Maze Search : Example

maze

entrance [1,1]

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

exit [5,4]

r,c

d=3

nR,nC

mark

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

stack

	3, 2, 3	top
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze

entrance [1,1]

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

r,c

d=4

nR,nC

exit [5,4]

mark

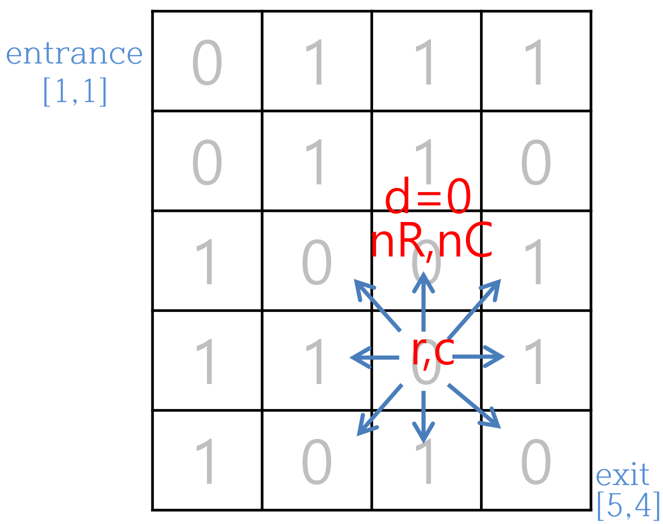
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

top	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze



mark

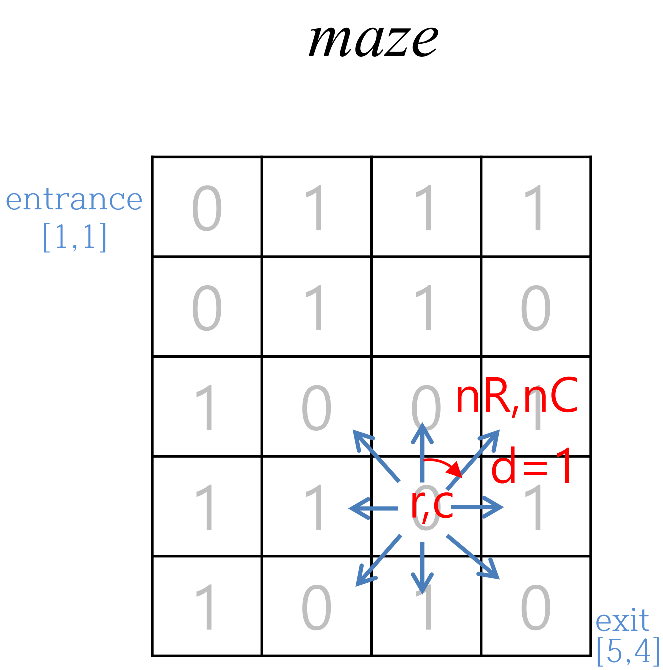
1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

stack

	3, 3, 5	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

push(3, 3, 5)
(4, 3) 방문, (r, c, d) = (4, 3, 0)

Maze Search : Example



mark

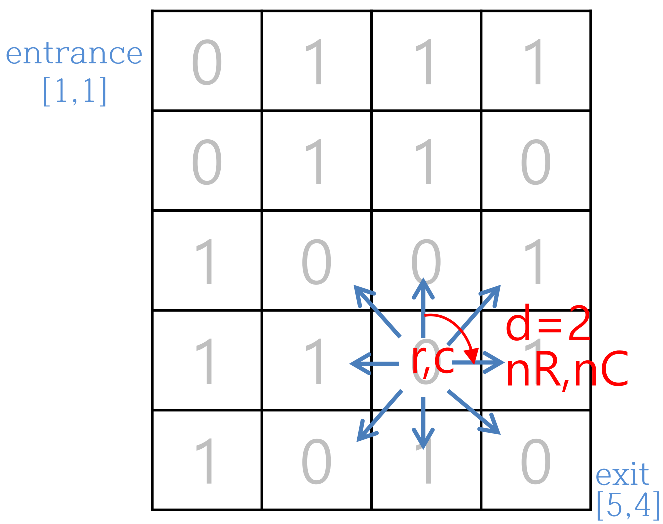
1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

stack

top	3, 3, 5
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze



mark

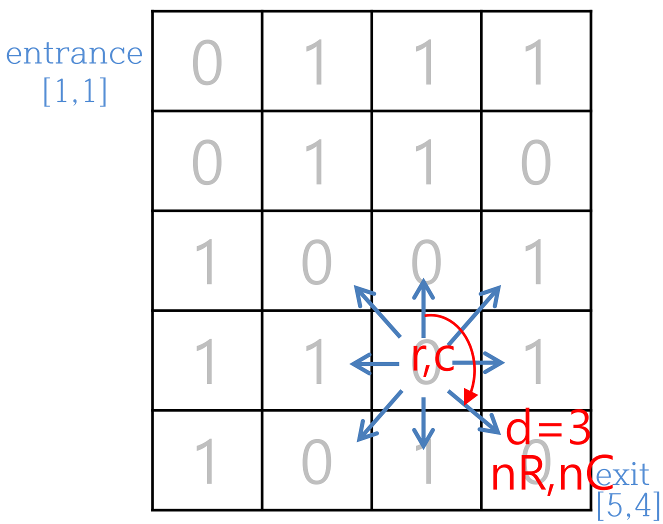
1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

stack

	3, 3, 5	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze



mark

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

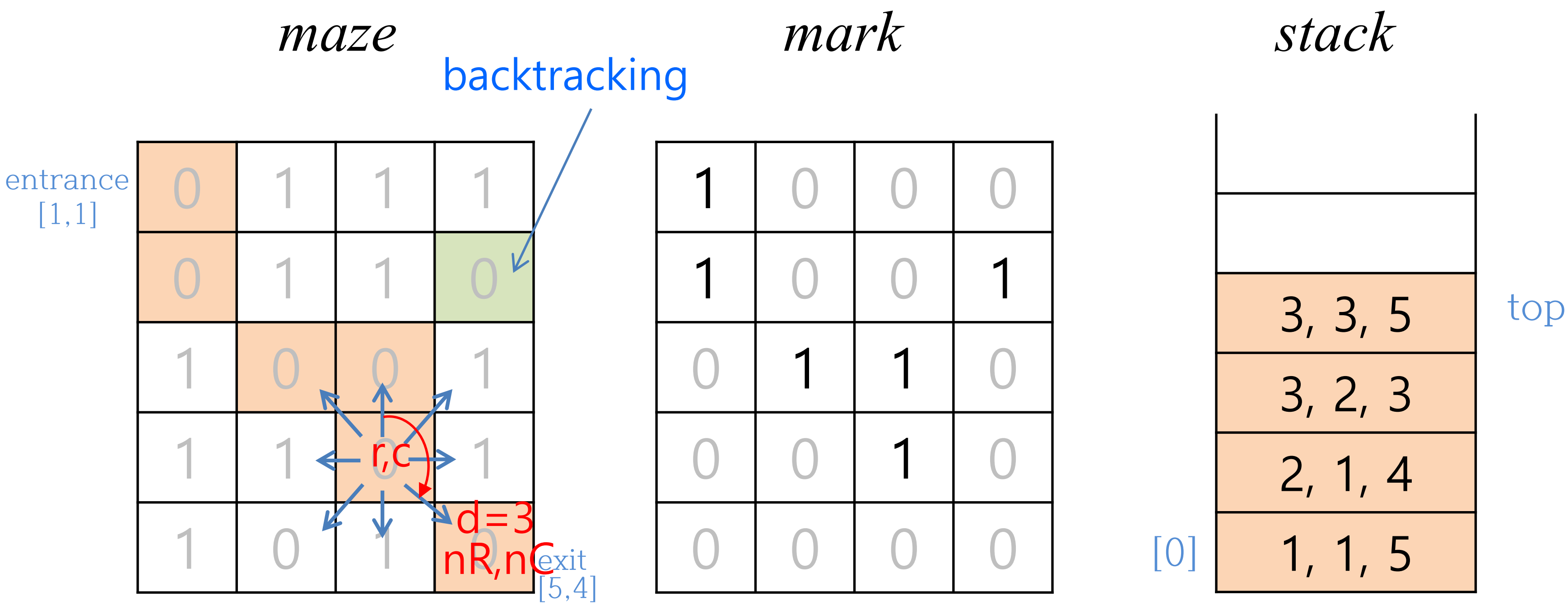
stack

3, 3, 5	top
3, 2, 3	
2, 1, 4	
[0] 1, 1, 5	

다음 위치 (nR, nC)가 출구(EXIT_ROW, EXIT_COL)임

경로발견!

Maze Search : Example



< 경로출력순서>

① stack[0] → stack[top]
② 현재 위치 (r, c)
③ 출구 위치 (EXIT_ROW, EXIT_COL)

path: (1, 1), (2, 1), (3, 2), (3, 3), (4, 3), (5, 4)

- Stack
  - Use the implementation of section 3.1 or 3.2

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

  - Capacity
    - Each position in the maze is visited no more than once.
    - An $m \times p$ maze has at most mp zeroes.
    - ***mp*** is sufficient for the stack capacity.

```
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 = 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;
      }
   }
   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*)**, for $m \times p$ maze