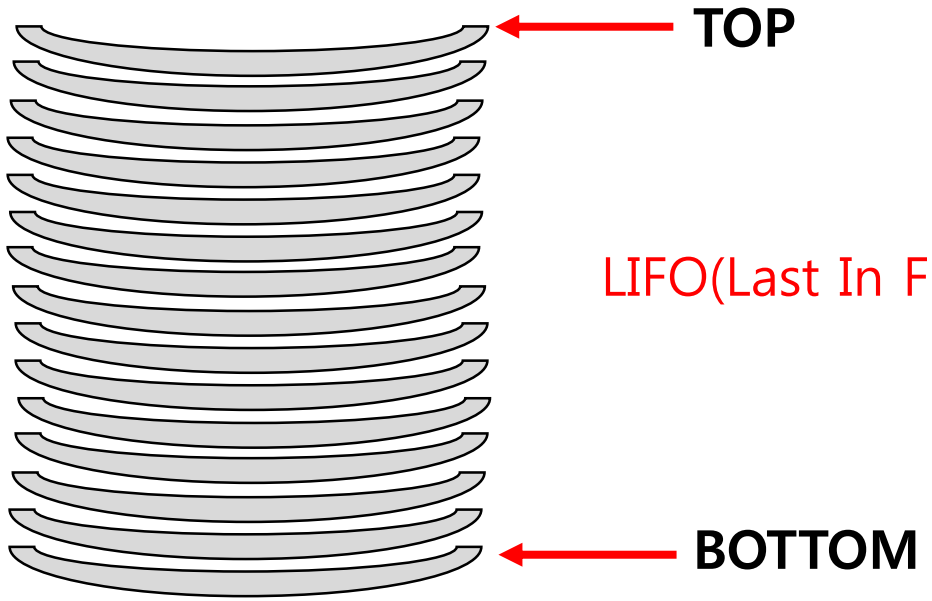
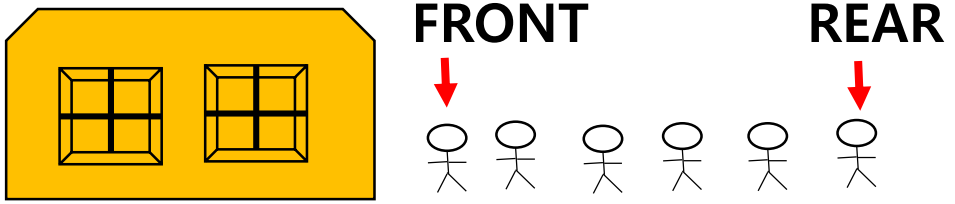
Chap 3. Stacks and Queues

Stack and Queue

- Special cases of ordered list
  - Insert and delete of an element are made at certain positions only
  - Stack
    - Insert (PUSH): at TOP
    - Delete(POP): at TOP
  - Queue
    - Insert (enqueue, add): at REAR
    - Delete(dequeue): at FRONT
- Real life examples
  - Stack
    - Stack of dish plates
    - Stack of disks in Towers of Hanoi
  - Queue
    - Queue in a restaurant
    - Queue in a ticket office



LIFO(Last In First Out)

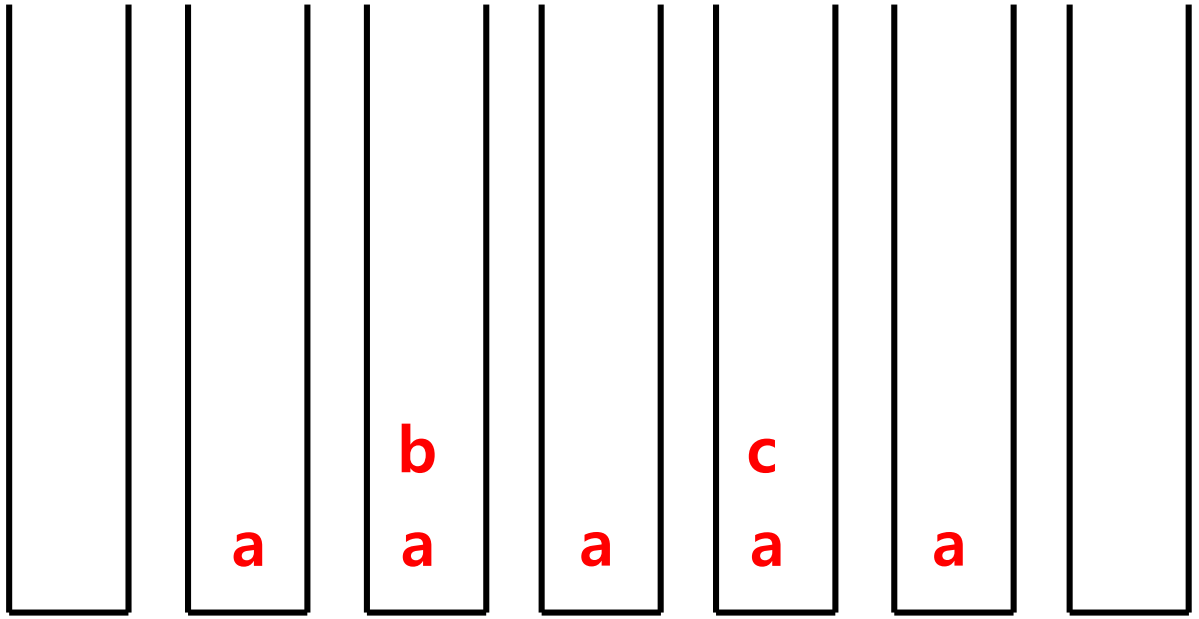


FIFO(First In First Out)

Examples in Computer Science

- Stack
  - Function call and return address
  - Stack of return addresses
- Queue
  - Job scheduling in operating system
  - Queue of jobs for CPU time

```
…                    f1(….) {
f1(…)                  …..
a: …                   f2(…)
                       b: …
                       f3(…)
                       c: ….
                     }
```



Representation of Stacks and Queues

- Array
  - Static memory allocation
    - n: max. number of elements in the list
    - Array[0..n-1]
  - Space utilization might be low
  - Overflow
  - Stack: top (stack pointer)
  - Queue: front, rear
- Linked list (Chapter 4)
  - dynamic memory allocation
  - malloc() & free() functions in C

Array Representation of a Stack

- Determine max. stack size: MAX_STACK_SIZE
- Data type of stack elements: T
- Declaration of array stack[]: T stack[MAX_STACK_SIZE];
- Initialization of top: int top = -1; //stack empty
- Example:

```
#define MAX_STACK_SIZE 20
typedef struct {
    int ID;
    char color;
} dish_plate;
dish_plate stack[MAX_STACK_SIZE];
int top = -1;
```

Stack operations

```
push(x) {
    if(stack_full()) handle_error
    top ← top + 1
    stack[top] ← x
}

pop() {
    if(stack_empty()) handle_error
    x ← stack[top]
    top ← top – 1
    return(x)
}
```

```
stack_full() {
    if(top>=MAX_STACK_SIZE-1)
        // if(top=MAX_STACK_SIZE-1)
    return TRUE
    else return FALSE
}

stack_empty() {
    if(top<0) return TRUE
        //if(top=-1) return TRUE
    else return FALSE
}
```

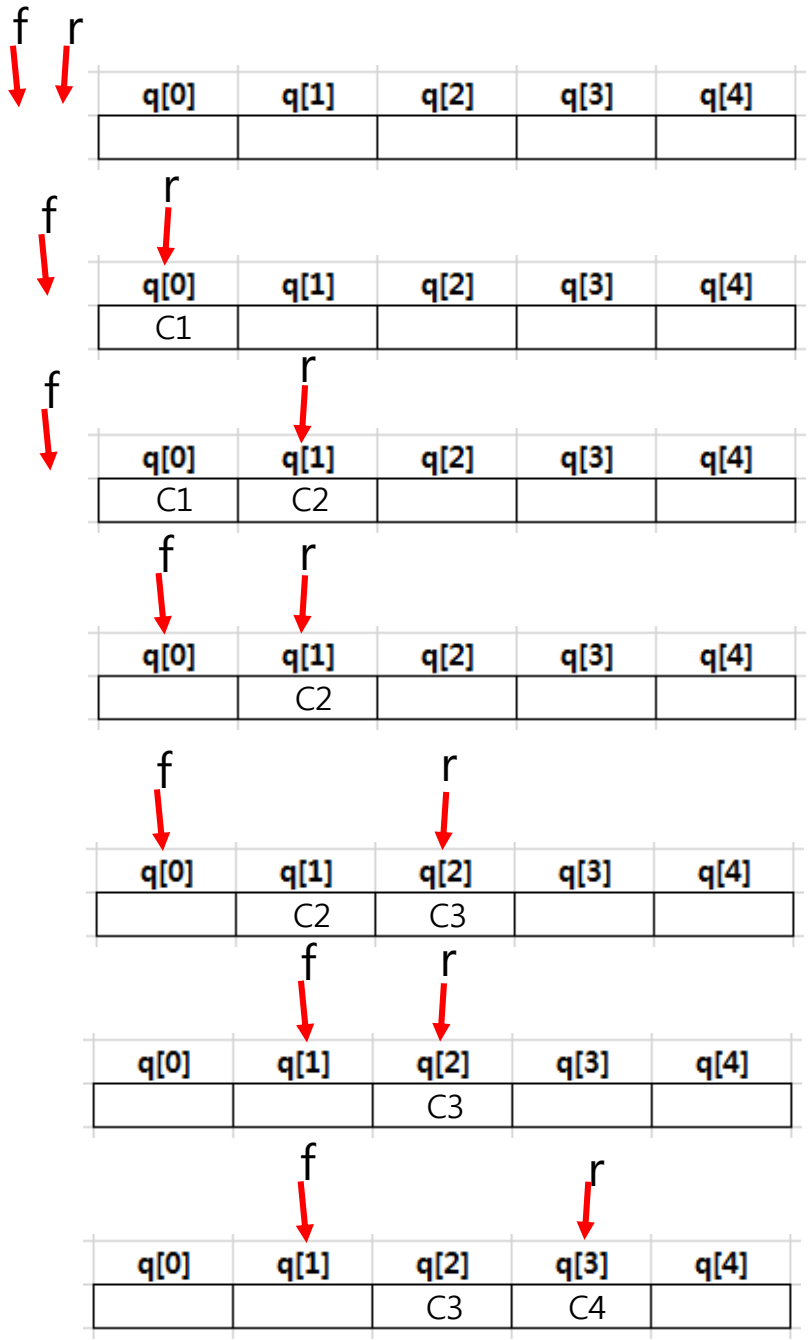
Array Representation of a Queue

- Determine max. stack size: MAX_QUEUE_SIZE
- Data type of stack elements: T
- Declaration of array stack[]: T queue[MAX_QUEUE_SIZE];
- Initialization of front & rear: int front = rear = -1; //queue empty
- Example:

```
#define MAX_QUEUE_SIZE 20
typedef struct {
    int waitingNum;
    char name[20];
} Client;
Client queue[MAX_QUEUE_SIZE];
int front = -1;
int rear = -1;
```

Example

- Initialization
  - empty queue
  - f=r=-1
- client C1 arrives & waits
- client C2 arrives & waits
- client C1 is served
- client C3 arrives & waits
- client C2 is served
- client C4 arrives & waits
- Observations
  - front points to *one element before* the front element of the queue
  - rear points to the rear element of the queue



Queue operations

```
insertQ(x) {
    if(queue_full()) handle_error
    rear ← rear + 1
    queue[rear] ← x
}
```

```
deleteQ() {
    if(queue_empty()) handle_error
    front ← front + 1
    x ← queue[front]
    return(x)
}
```

```
queue_full() {
    if(rear >= MAX_QUEUE_SIZE-1)
        // if(rear=MAX_QUEUE_SIZE-1)
    return TRUE
    else return FALSE
}
```

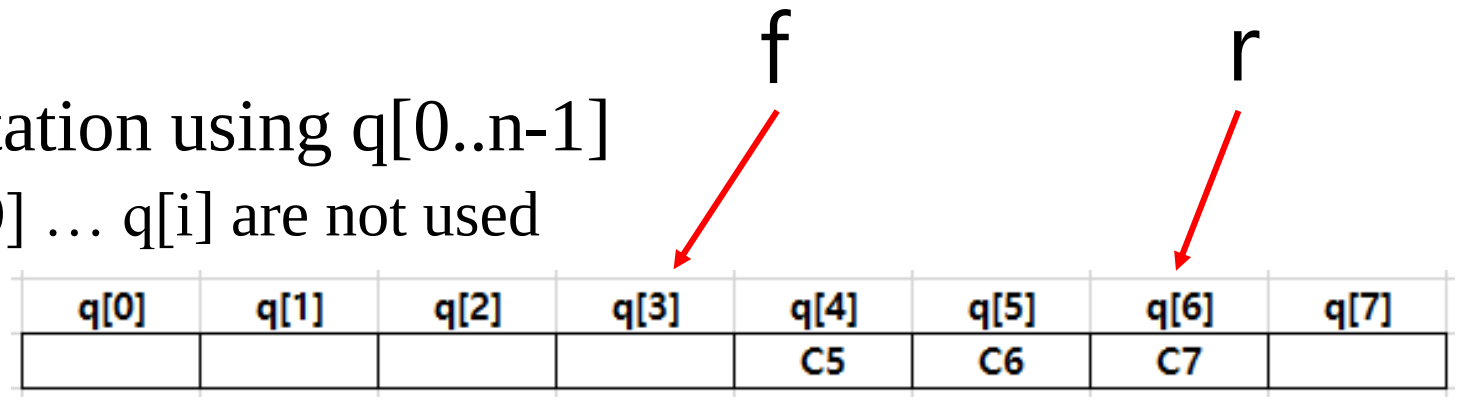
```
queue_empty() {
    if(front = rear) return TRUE
    else return FALSE
}
```

Time complexity of stack & queue operations

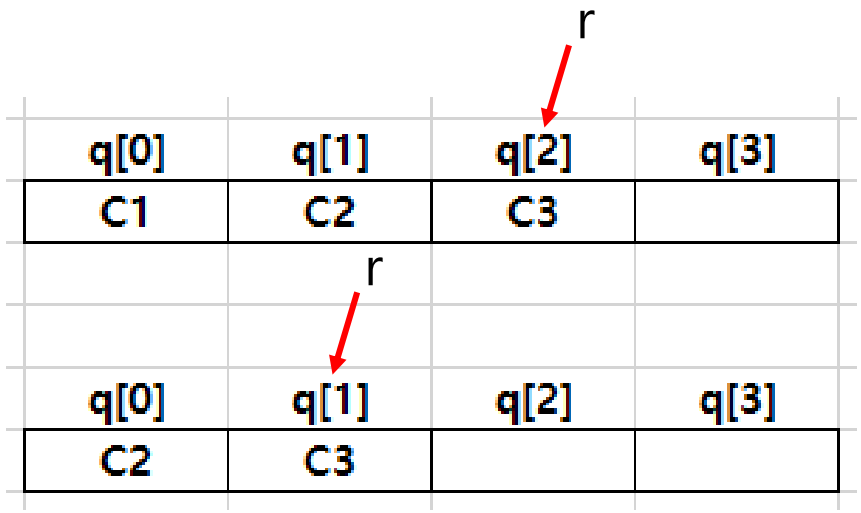
- Stack of size n
  - Push: O(1)
  - Pop: O(1)
- Queue of size n
  - Insert: O(1)
  - Delete: O(1)
- Why?

Circular Queue

- Problem in queue representation using q[0..n-1]
  - When front = i (i >= 0) : q[0] … q[i] are not used
  - Eventually,
    - front = rear = n-1
    - empty queue (front = rear) as well as
    - full queue (rear = n-1) while the entire array is available
    - No further insert() is allowed

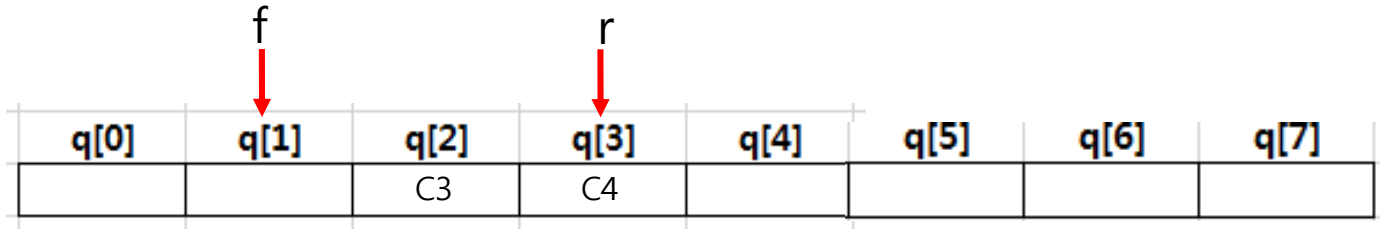


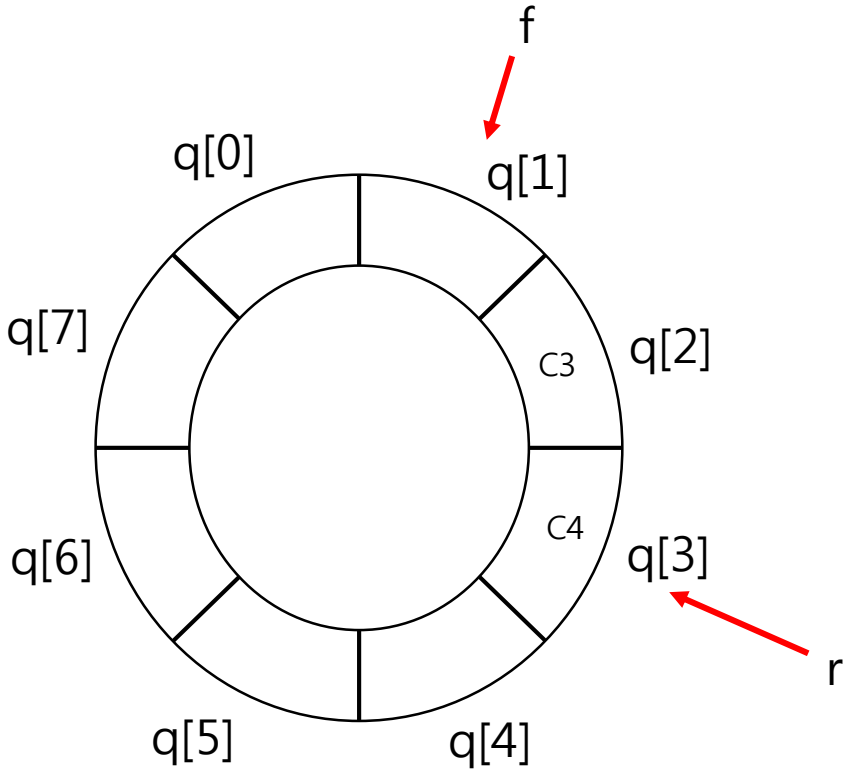
- Possible but inefficient Solution
  - Shift left every element after deleteQ()
  - front = -1 always: no need of maintaining front
  - Time complexity of deleteQ(): not O(1) any more
- Circular queue is the solution



Circular Queue

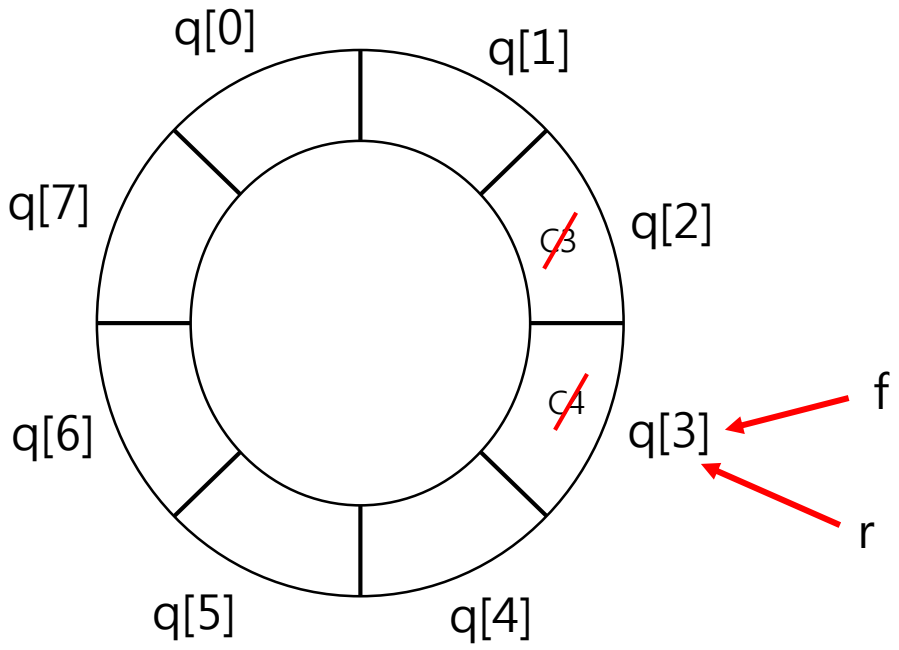
- Array q[0..n-1] is conceived as a circle
- Initialization
  - front = rear = 0
- Both front & rear moves clockwise
  - front(or rear) ← front(or rear) + 1
  - front(or rear) ← (front(or rear) + 1) mod n



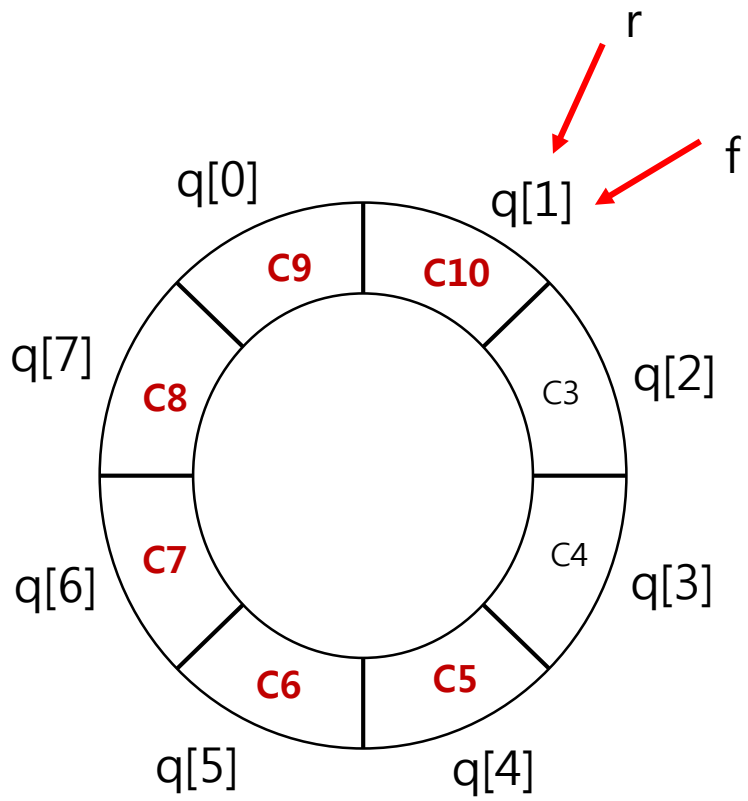


Circular Queue: one problem

- Queue empty
  - After C3 & C4 are deleted
  - front = rear

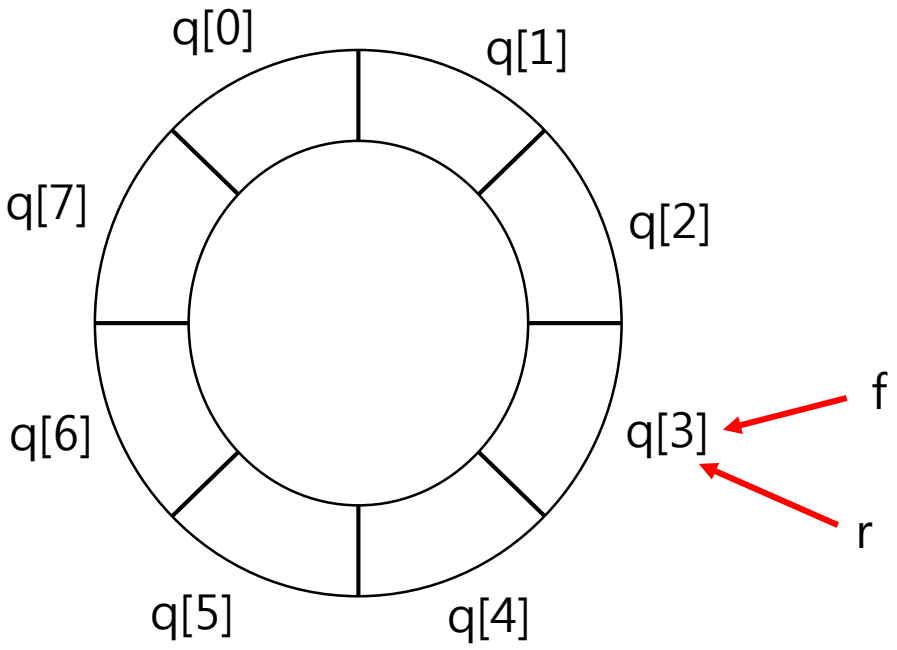


- Queue full
  - After C5 to C10 are inserted
  - front = rear

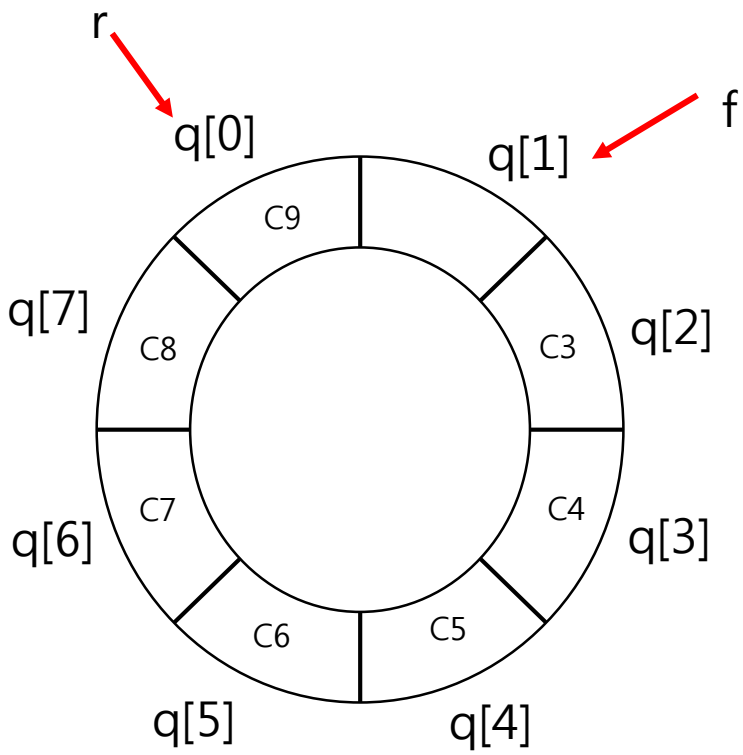


Circular Queue: Empty & Full

- Queue empty
  - front = rear



- Queue full
  - one element of array is *not* used
  - front = (rear + 1) mod n



Circular Queue operations

```
insertQ(x) {
    if(queue_full()) handle_error
    rear ← (rear + 1) mod n
    queue[rear] ← x
}
```

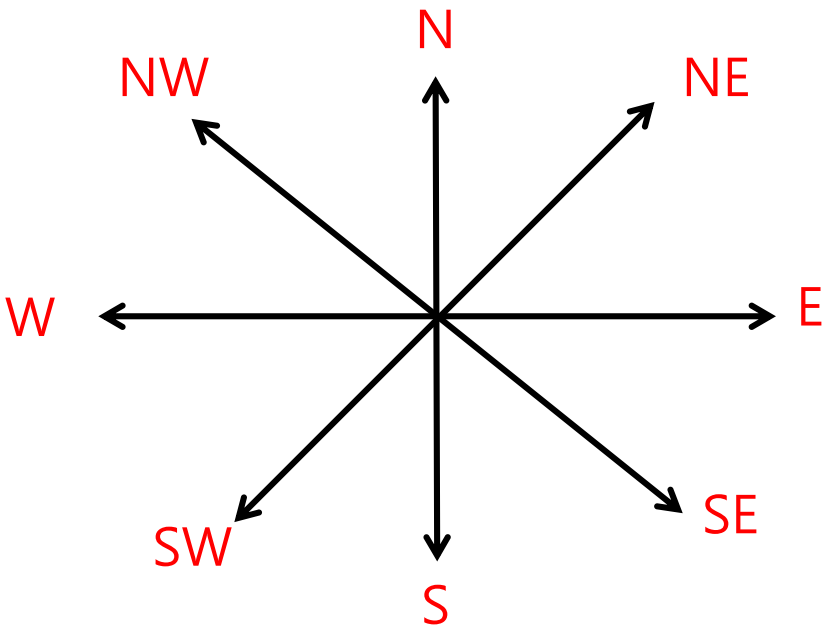
```
deleteQ() {
    if(queue_empty()) handle_error
    front ← (front + 1) mod n
    x ← queue[front]
    return(x)
}
```

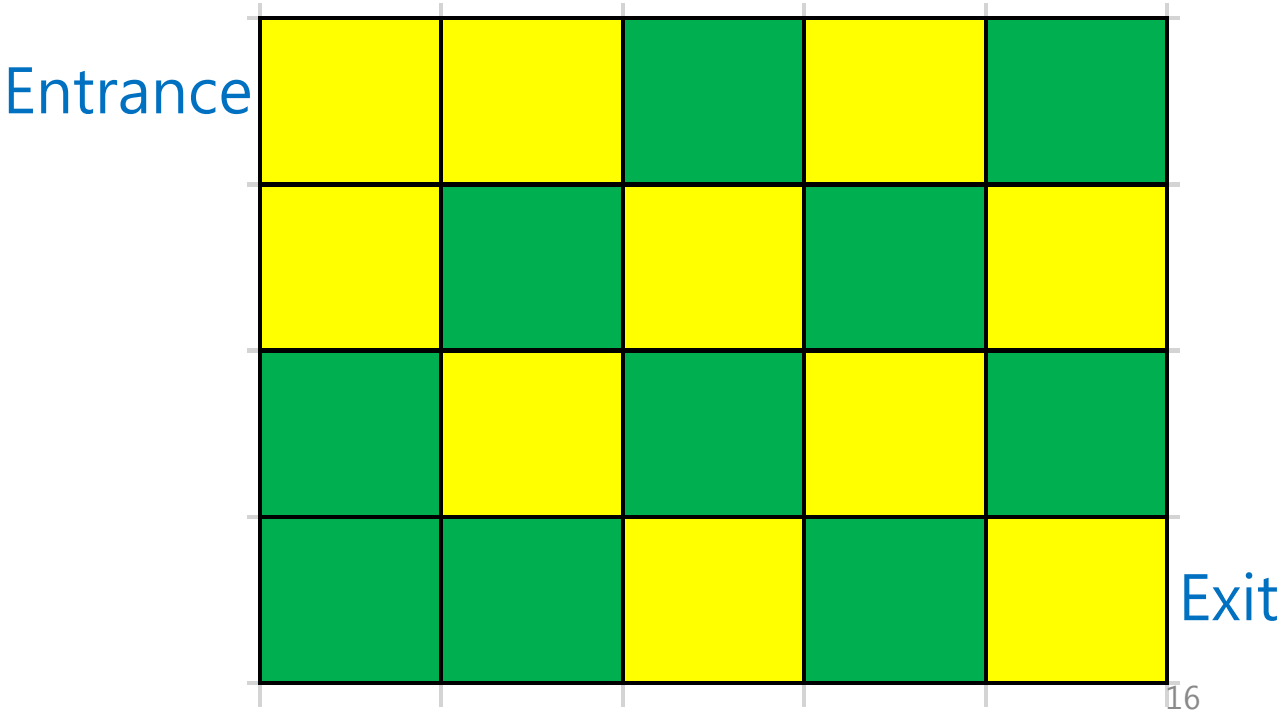
```
queue_full() {
    if((rear + 1) mod n = front) return TRUE
    else return FALSE
}
```

```
queue_empty() {
    if(front = rear) return TRUE
    else return FALSE
}
```

Maze problem

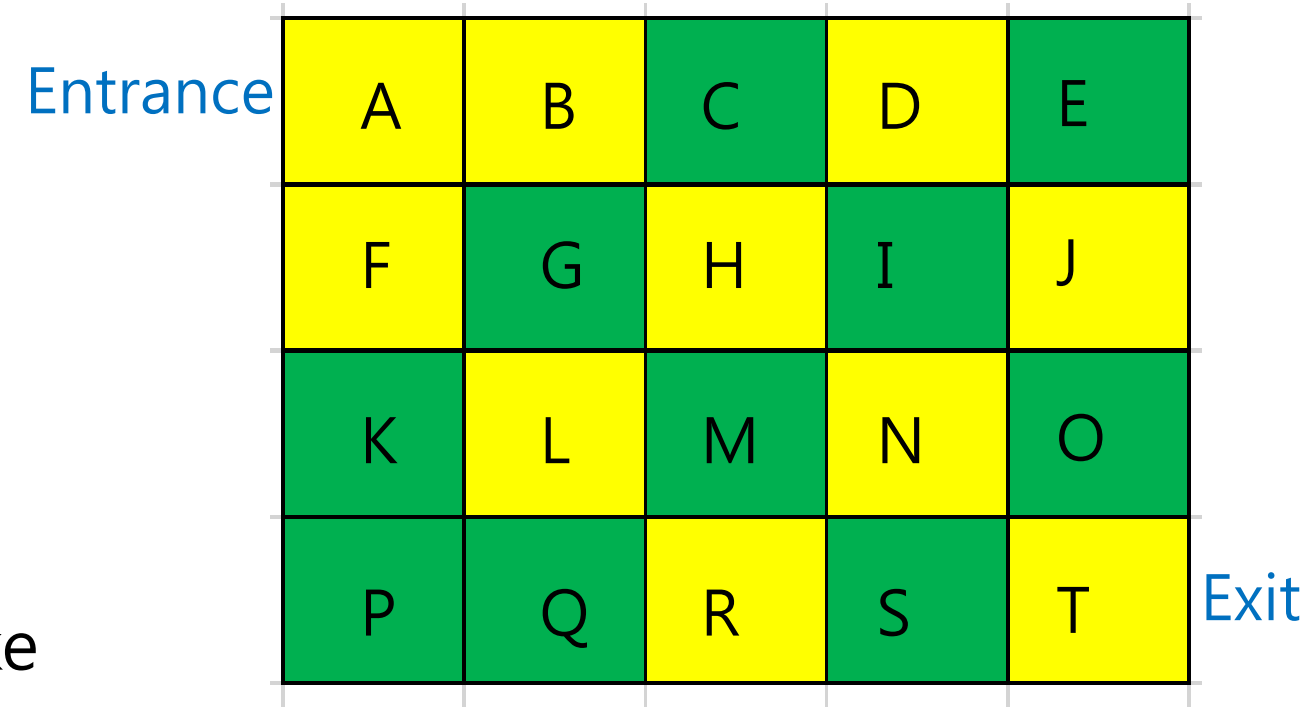
- Stack application
- Find a path from Entrance to Exit
  - Yellow: open, Green: blocked
  - Possible movements: 8 directions





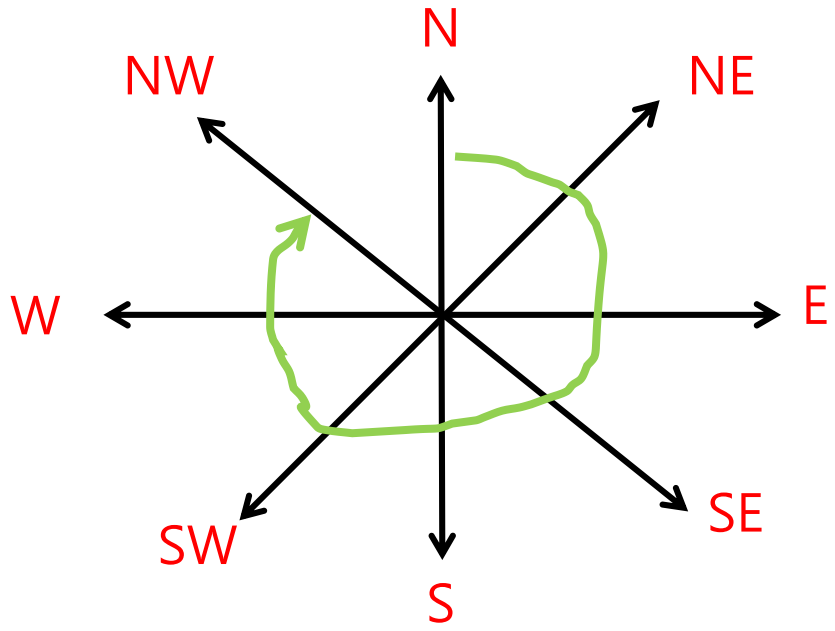
Legal paths

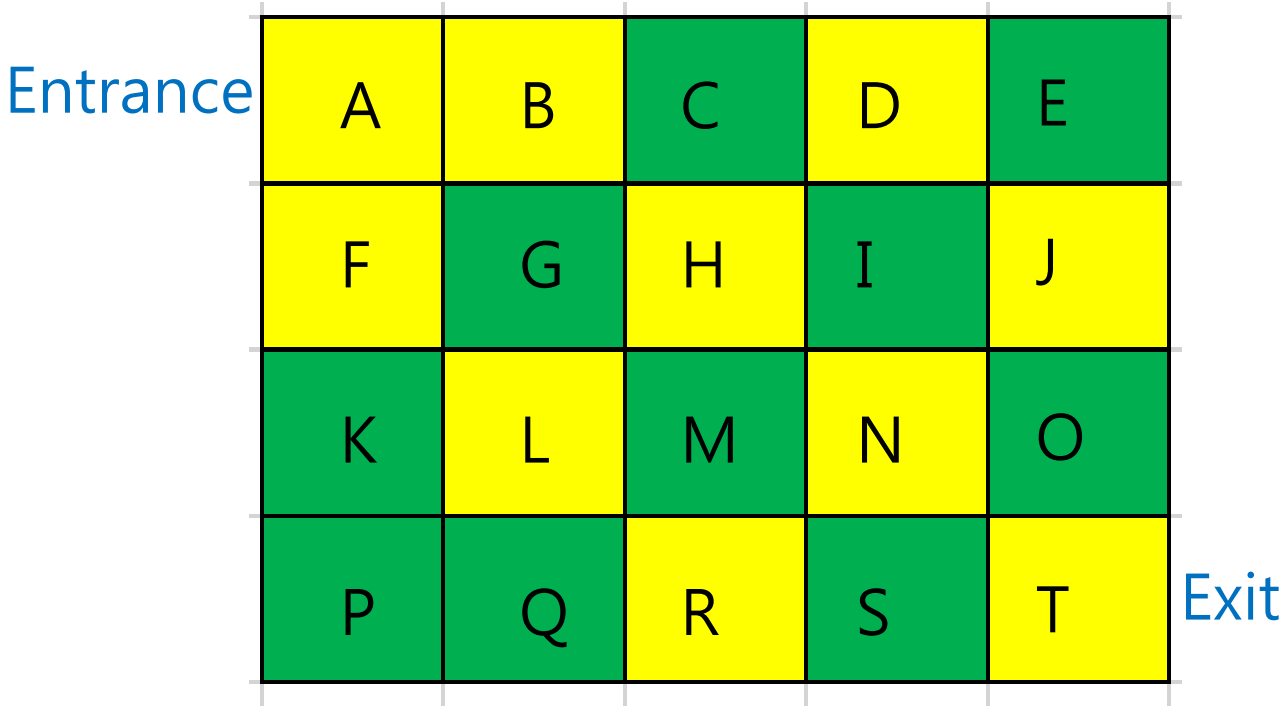
- ABHNT
- ABHLRNT
- ABHDJNT
- AFLRNT
- AFLHNT
- AFLHDJNT
- *Not* interested in paths like
  - ABHLRNHDJNT
  - ABHNJDHLRNT
- Which one of legal paths would your algorithm find?



Order of directions to explore

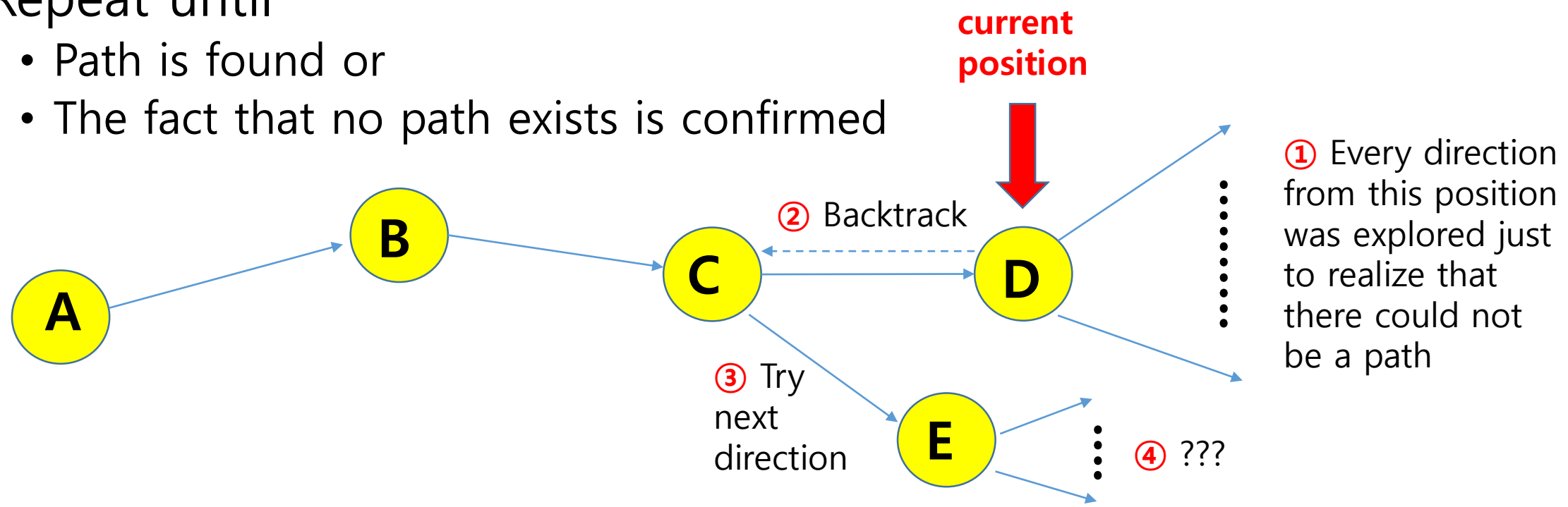
- Example
  - Starting from North
  - Next direction to be explored is determined clockwise
- Path found: ABHDJNT





Overview of algorithm

- In the current position,
  - If possible, move to a neighboring position
  - Otherwise, backtrack
- Repeat until
  - Path is found or
  - The fact that no path exists is confirmed

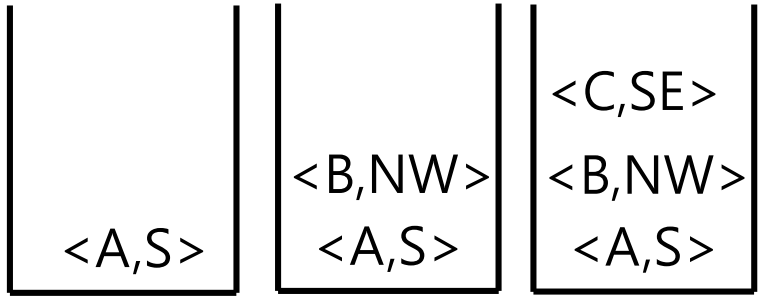


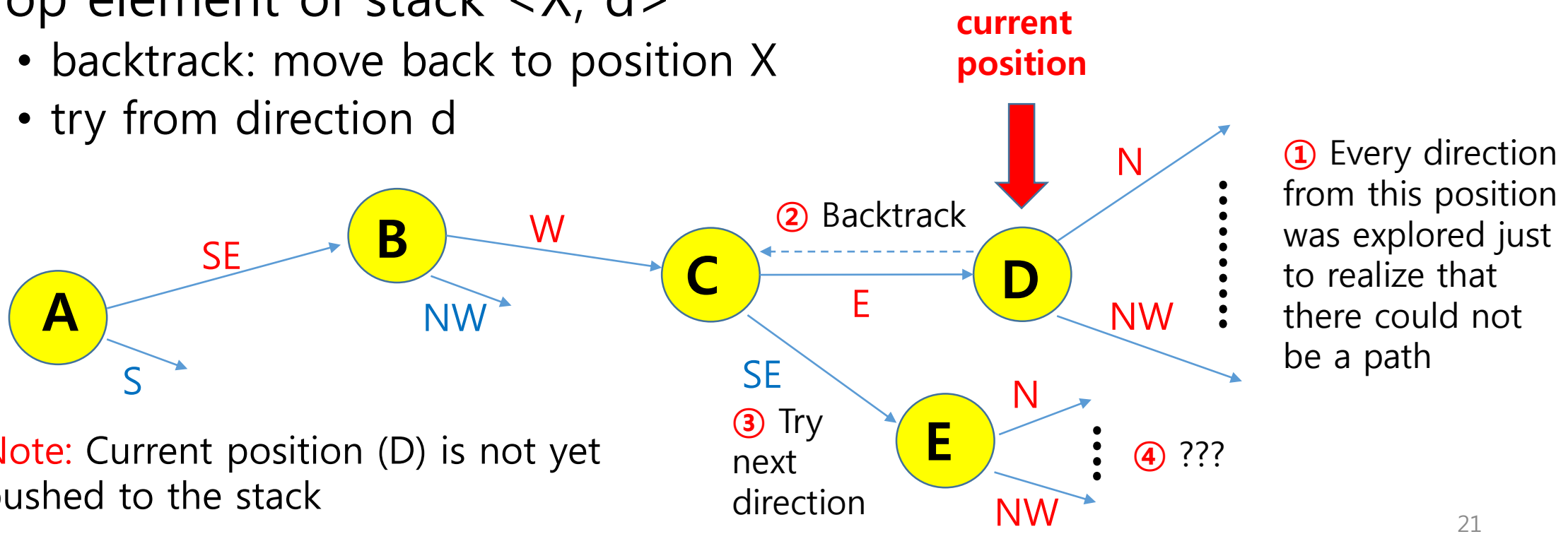
Q&A

- In the current position,
  - If possible, move to a neighboring position
    - **Q:** 만약에 이동 가능한 방향이 여럿이면?
    - **A:** 아직 시도하지 않은 방향 중 미리 정해진 순서에 따라 다음 순번 방향을 선택 (e.g., N→NE→… →NW)
  - Otherwise, backtrack
    - **Q:** backtrack은 어느 위치로?
    - **A:** stack의 top에 저장되어 있는 위치로
    - **Q:** stack의 용도는?
    - **A:** 현재 탐색 중인 path 및 이동 방향에 대한 정보를 기록
    - **Q:** backtrack 후에는 어느 방향으로 이동을 시도?
    - **A:** 그 정보도 stack에 저장되어 있음. Backtrack 시 stack의 top을 pop하여 그 내용을 보면 backtrack해서 갈 위치와 거기서 이동을 시도할 방향이 기록되어 있음

Stack

- Saves
  - Current path
  - i.e., sequence of <position, direction>'s
- Top element of stack <X, d>
  - backtrack: move back to position X
  - try from direction d

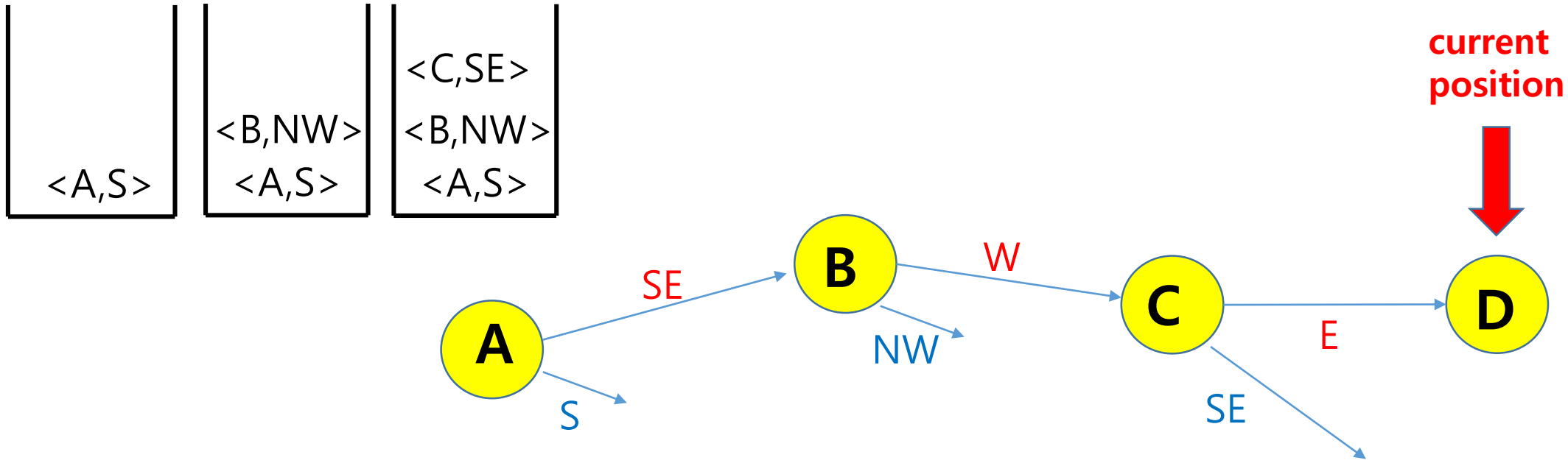




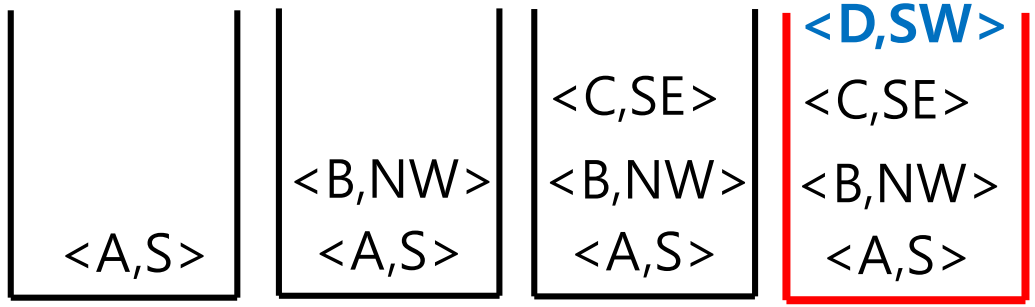
- Note: Current position (D) is not yet pushed to the stack

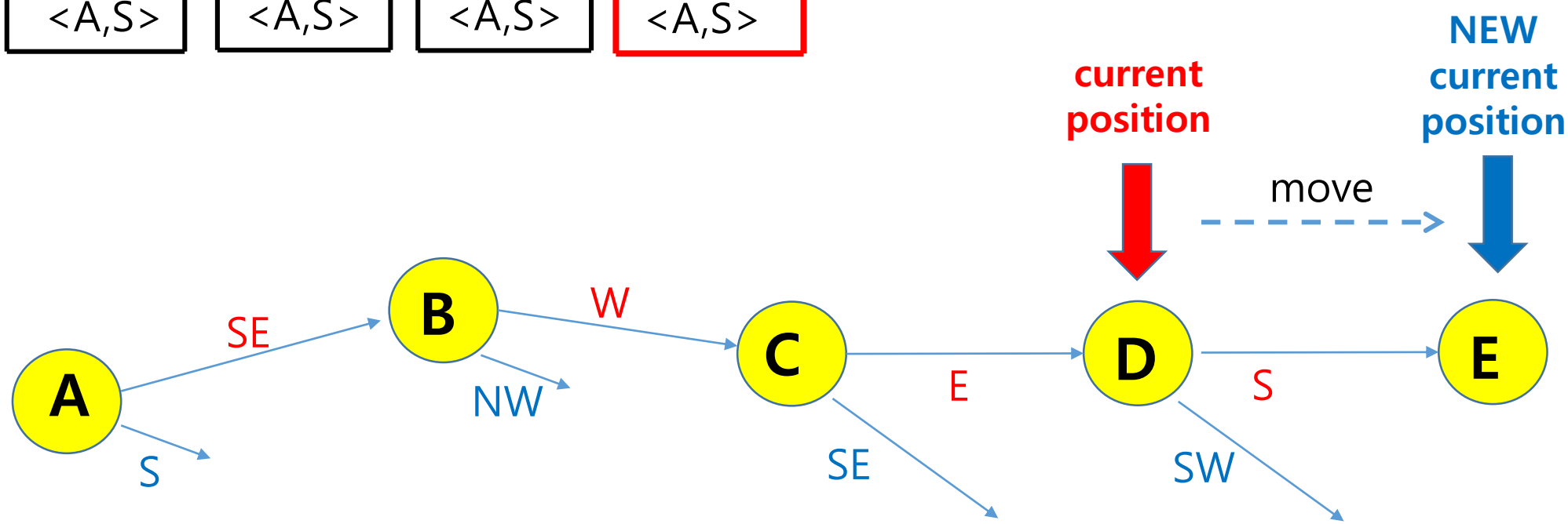
Stack updates

- Done when current position is changed
  - Move: push
  - Backtrack: pop

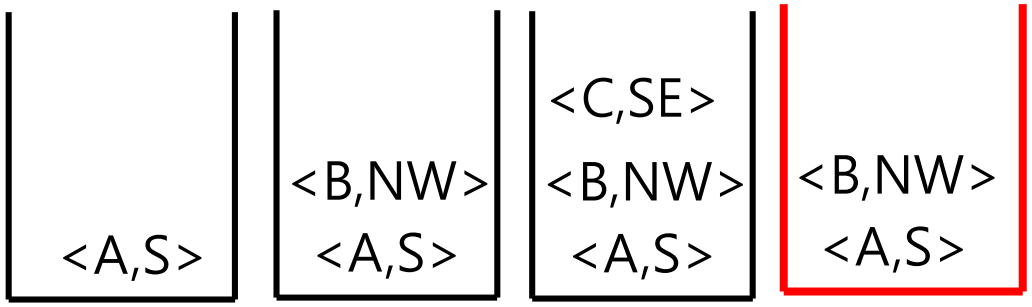


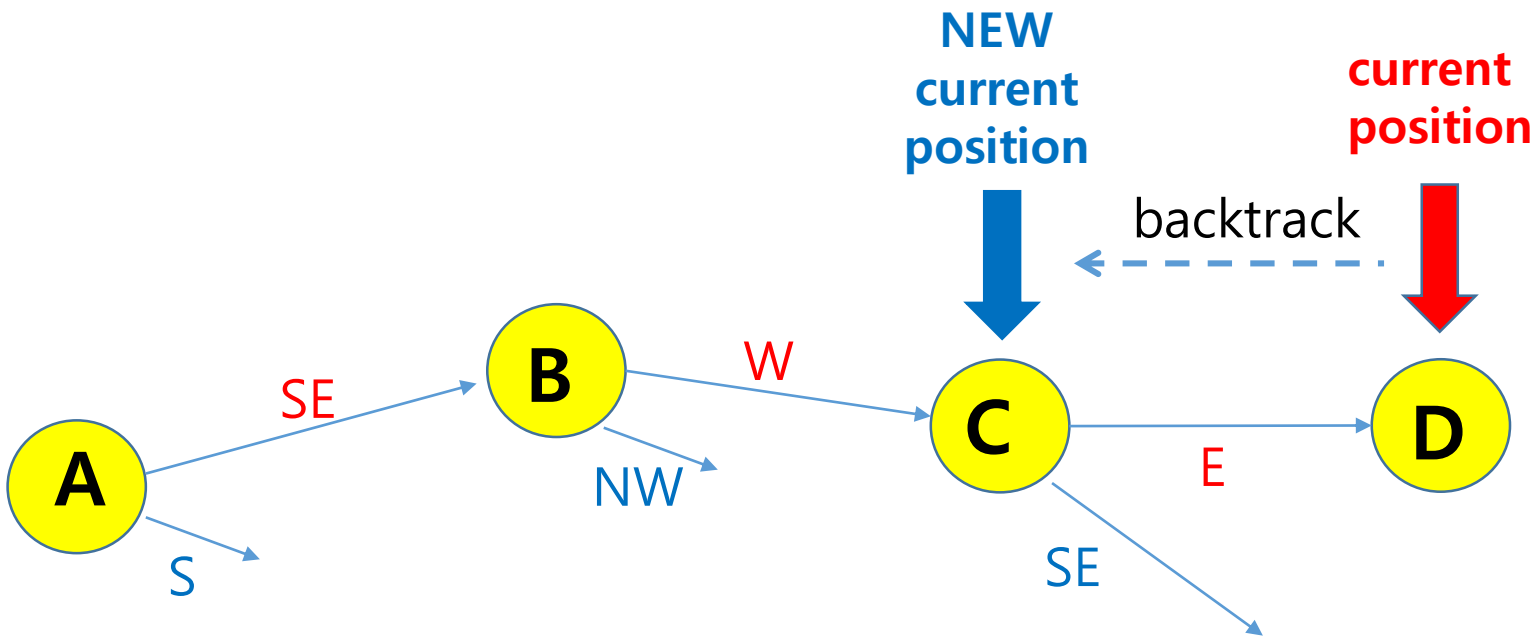
Stack updates: Push

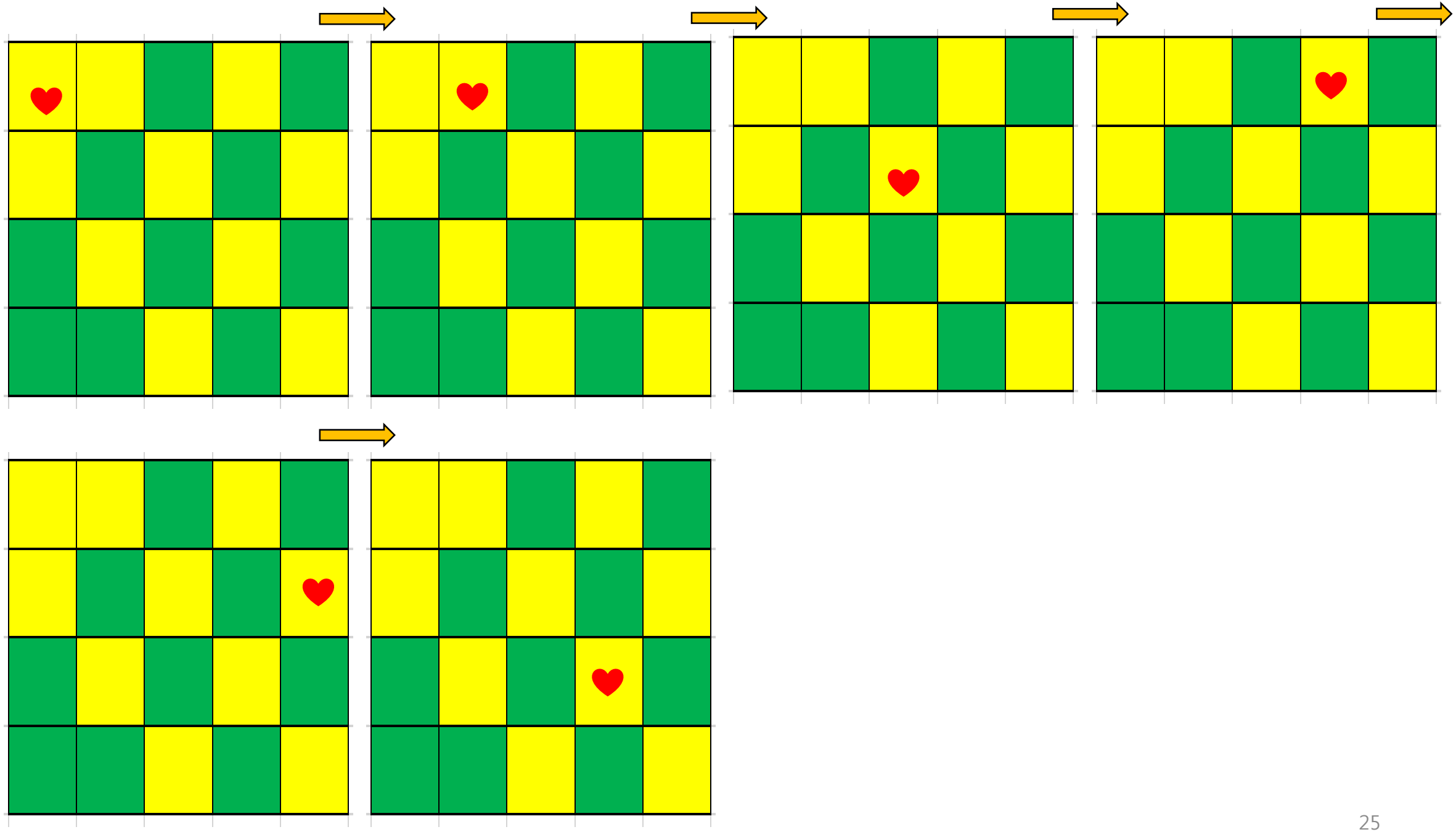




Stack updates: Pop

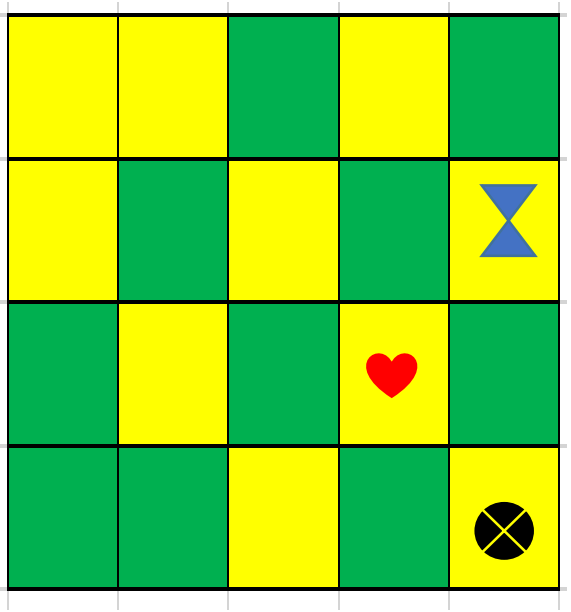


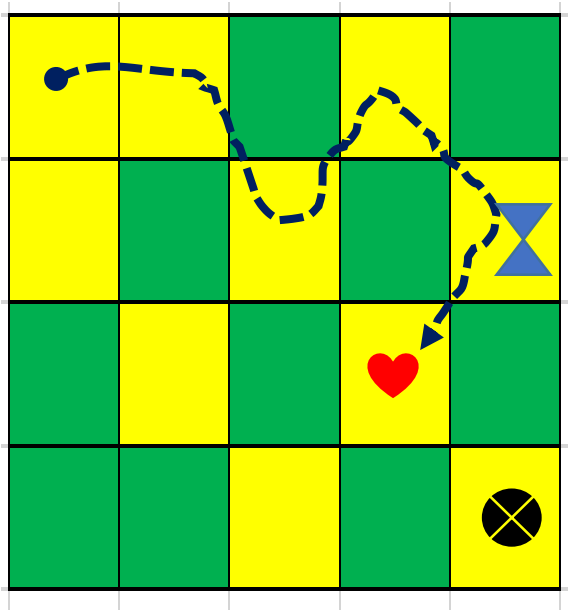




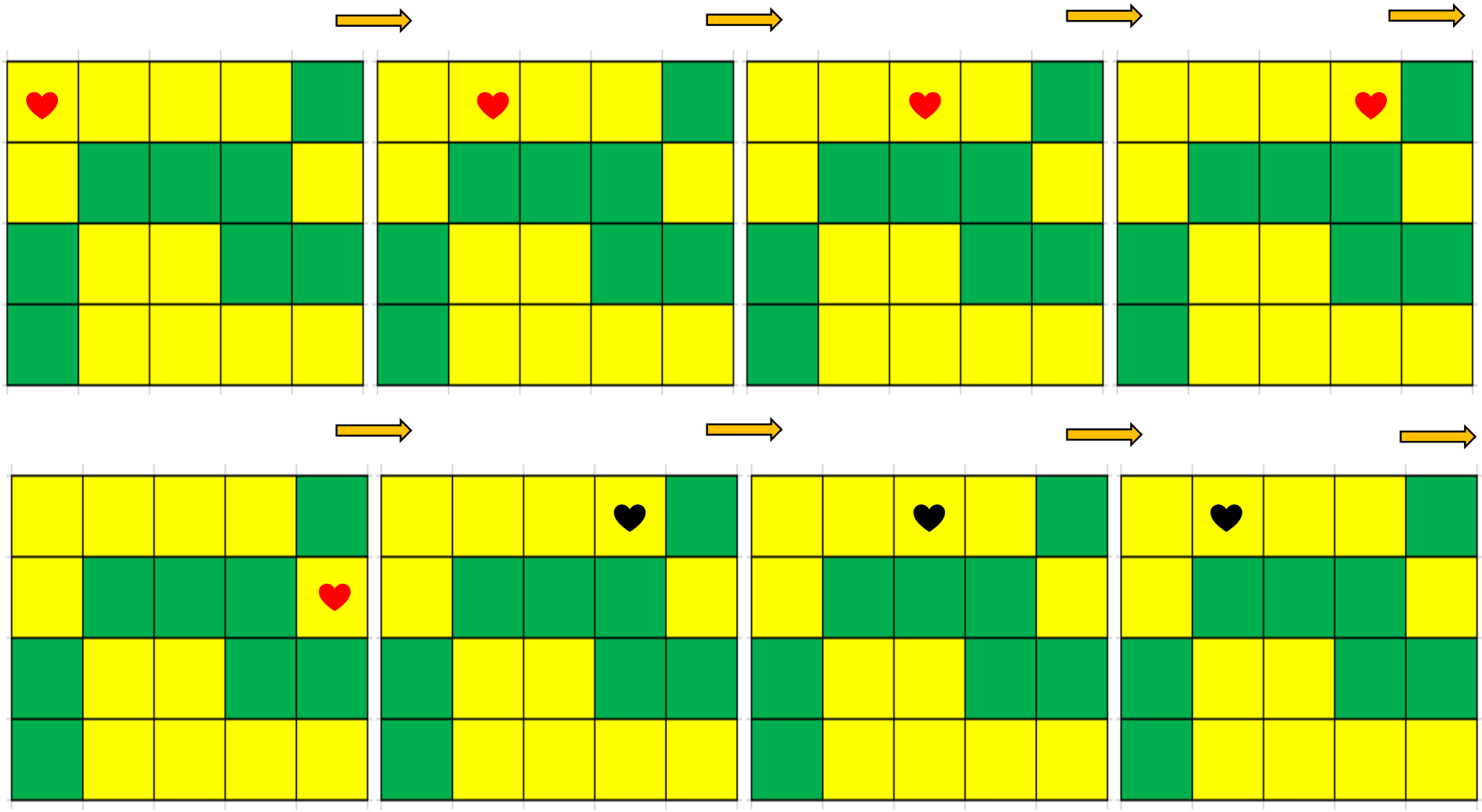
Termination of algorithm

- Current position (♥) is *not* the exit (⊗)
- Top of the Stack
  - *Not* saving the current position (♥)
  - But saves the position of ⧗

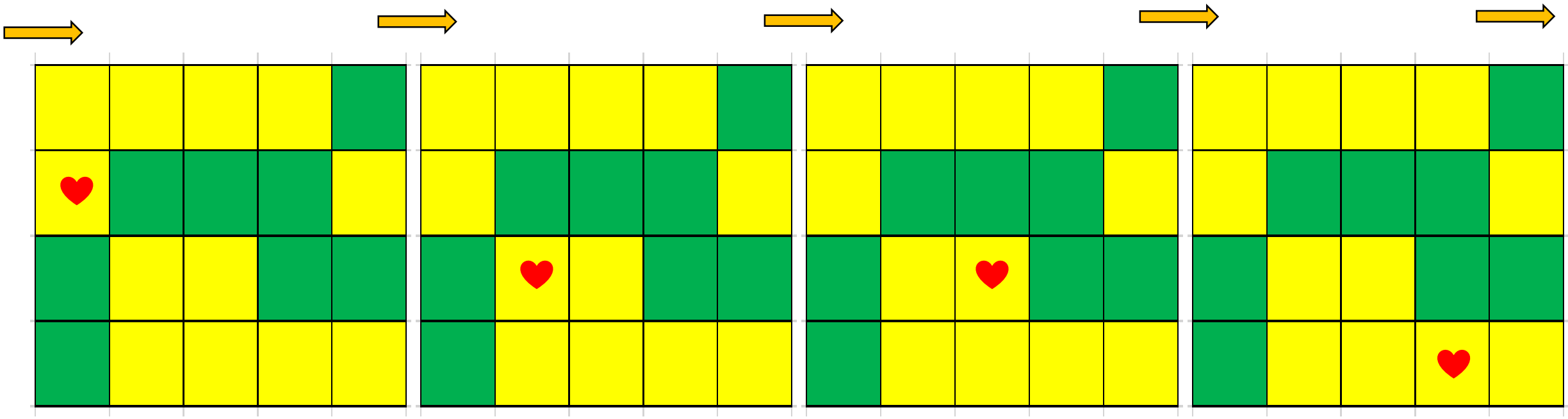




Example with backtracking



Example with backtracking (cont'd)

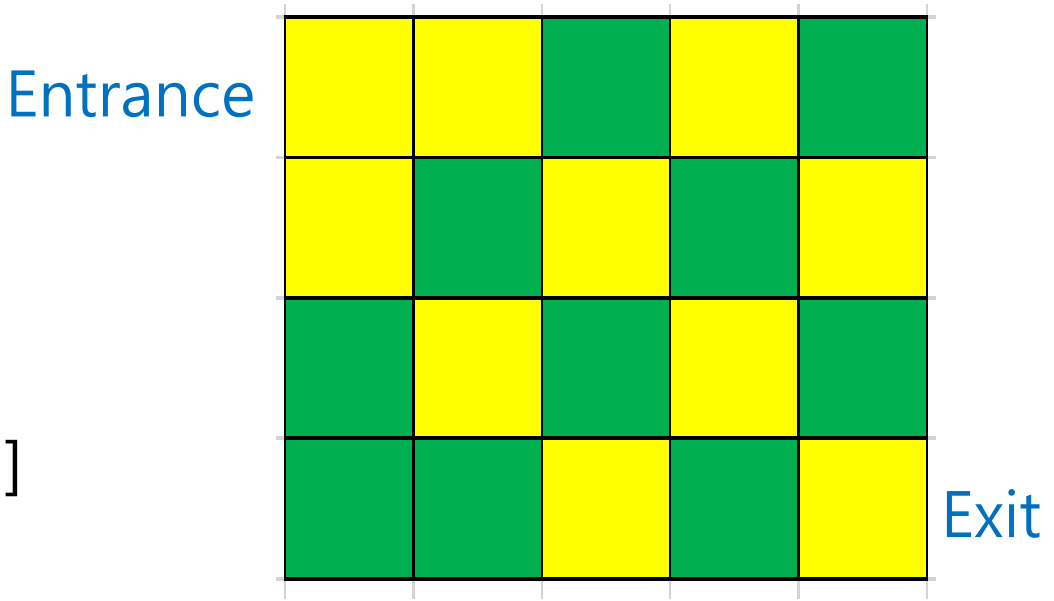


♥ Move to a new position

♥ backtrack

Data structures

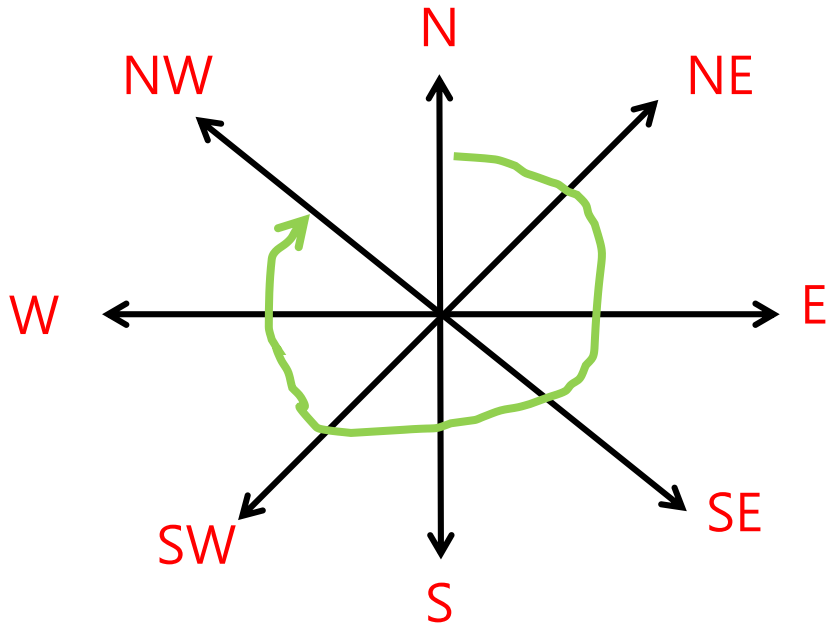
- Representation of a maze
  - 2-dim array maze[][]
  - maze[i][j] = $\begin{cases} 0 & if\ open \\ 1 & if\ blocked \end{cases}$
  - n x m maze: int maze[n+2][m+2]
  - Example
    - 4 x 5 maze: int maze[6][7]
  - why +2 ?
    - maze[0][*]=1
    - maze[n+1][*]=1
    - maze[*][0]=1
    - maze[*][m+1]=1
  - Entrance: maze[1][1]
  - Exit: maze[n][m]
- Position: <row, col>

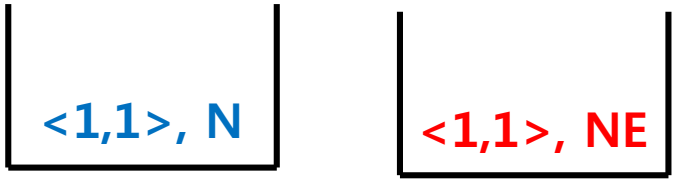


1	1	1	1	1	1	1
1	**0**	0	1	0	1	1
1	0	1	0	1	0	1
1	1	0	1	0	1	1
1	1	1	0	1	**0**	1
1	1	1	1	1	1	1

Data structures

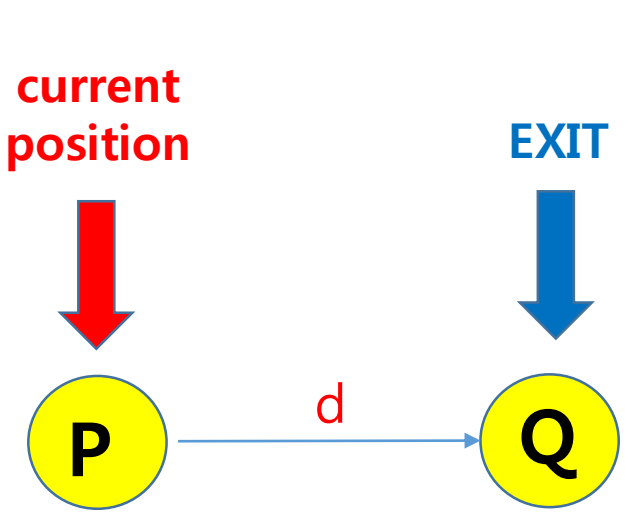
- Legal move
  - avoid to move to the position already tried
  - 2-dim array mark[][]
    - Initialization: mark[][] ← 0
    - When moving to Position <i, j>: mark[i][j] ← 1
  - Condition that move to position <i, j> is legal
    - maze[i][j]=0 AND mark[i][j]=0
- Order of moving directions
  - Assign integers in increasing order
  - {N, NE,…,NW} = {0,1, …, 7}
  - Next direction ← current direction + 1
- Type of stack elements
  - <position, direction>
  - <row, col, direction>
- Initialization of stack





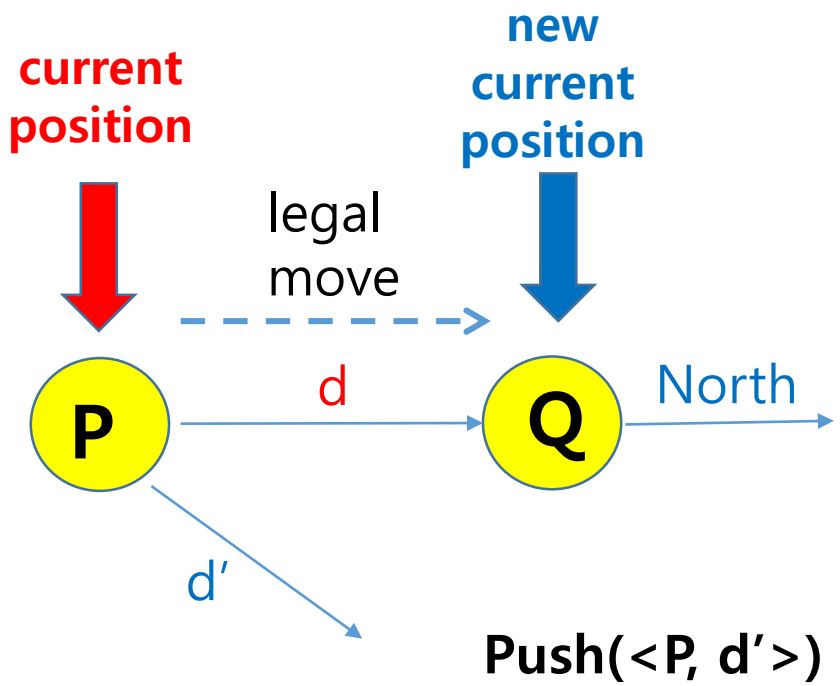
3 cases out of current position & moving direction

- P: current position
- d: moving direction
- Q: the position reached from P in direction d



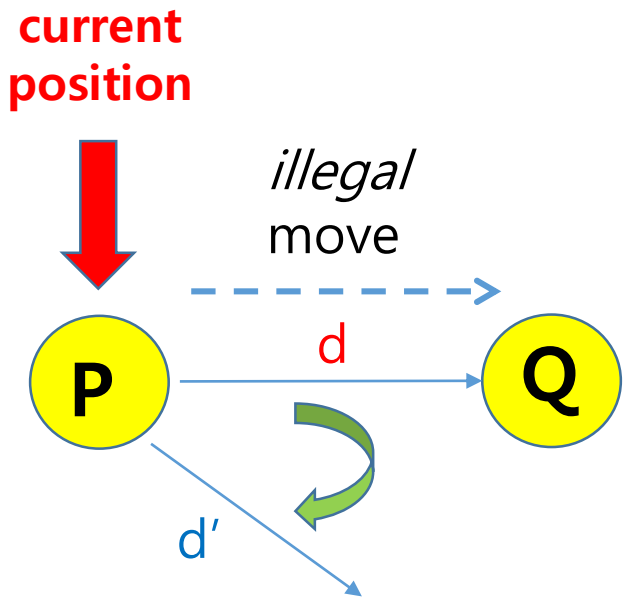
Q is the exit

A path is found!



Q is *not* the exit &

move from P to Q is legal



Q is *not* the exit &&

move from P to Q is *illegal*

Evaluation of postfix expressions

- Stack application
- Expression
  - Examples
    - a + b * c
    - (a + b) * c
  - Sequence of operands, operators, and parentheses
    - Parentheses: to override operator precedence
  - Notations for expression a+b
    - Infix: a+b
    - Postfix: ab+
    - Prefix: +ab

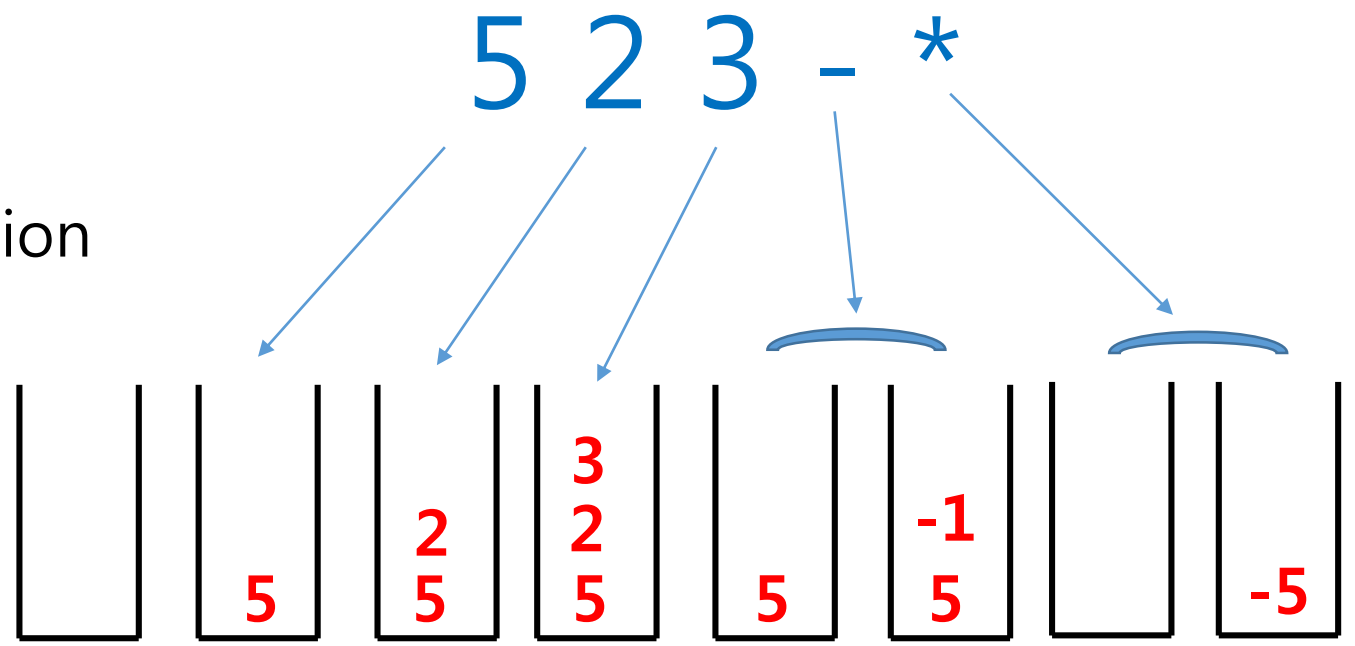
Postfix expression

- Examples
- Parenthesis-free
  - Makes evaluation simple
- Note
  - Sequence of operands only
    - a+b*c (infix): abc
    - abc*+ (postfix): abc
  - No difference between infix and postfix
- Evaluation
  - 1-pass scan from left to right
  - stack
- Tasks
  - Evaluation of postfix expression
  - Infix to postfix conversion

infix	postfix
a+b*c	abc*+
(a+b)*c	ab+c*

Evaluation of postfix expressions

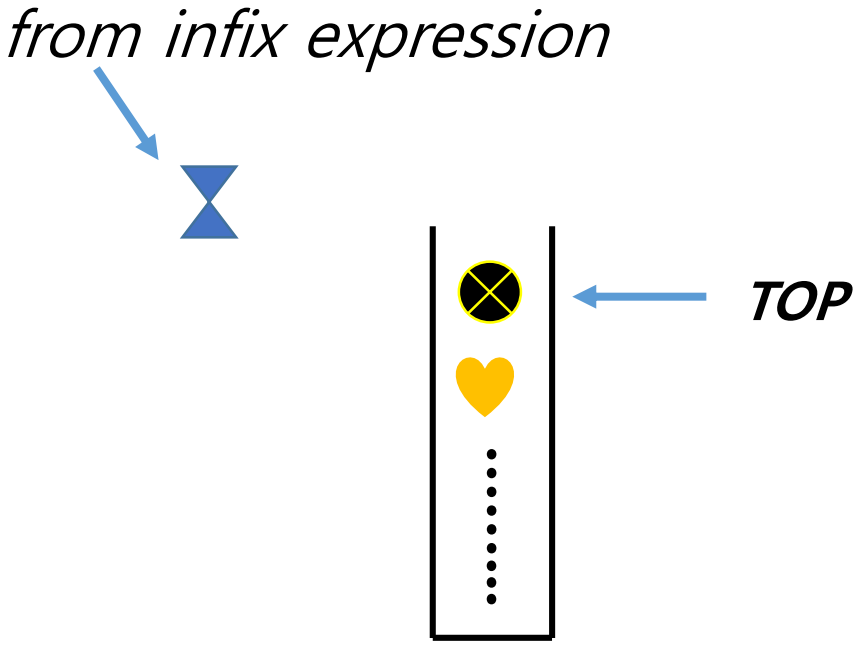
- 1-pass scan of postfix expression from left to right
- Cases:
  - Operand X: push(X)
  - Binary operator ⊙:
    1. Z ← pop() //Operand2
    2. Y ← pop() //Operand1
    3. W ← Y ⊙ Z
    4. Push(W)
- Termination: end of expression
- Final result of expression
  - at the top of the stack
- Example
  - Infix: 5*(2-3)
  - Postfix: 523-*



Infix to postfix conversion

- 1-pass scan of *infix* expression
- stack
- Cases:
  - Operand X: output(X) //operands will never be pushed to the stack
  - ')':
    - Repeat output(pop()) until finding its matching '(' at stack top
    - '(' at stack top: popped but not output
  - Others Y: Y is either operator ⊙ or '('
    - **in-stack precedence** of stack[top] >= **incoming precedence** of Y: output(pop())
    - Otherwise: push(Y)
  - End of infix expression
    - Complete conversion by (possibly repetitive) output(pop())'s

Infix to postfix conversion



switch(compare(
in-stack precedence of
and
incoming precedence of
)
) {
case >= : pop
output
case < : push
}

In-stack & in-coming precedence

- String of infix expression
  - Sequence of tokens
  - Example: (a+b)*c
    - String: (a+b)*c$ //$: end of string
    - Tokens: (, a, +, b,), *, c, $
- In-stack precedence
  - Token at stack top
- incoming precedence
  - Token from the string of infix expression

In-stack & incoming precedence

- isp & icp for short
- Not absolute values but relative values: e.g., icp(+) < isp(*)
- Example

	(	)	+	-	*	/	$(end of string)
in-stack	0	3	1	1	2	2	0
incoming	4	3	1	1	2	2	0

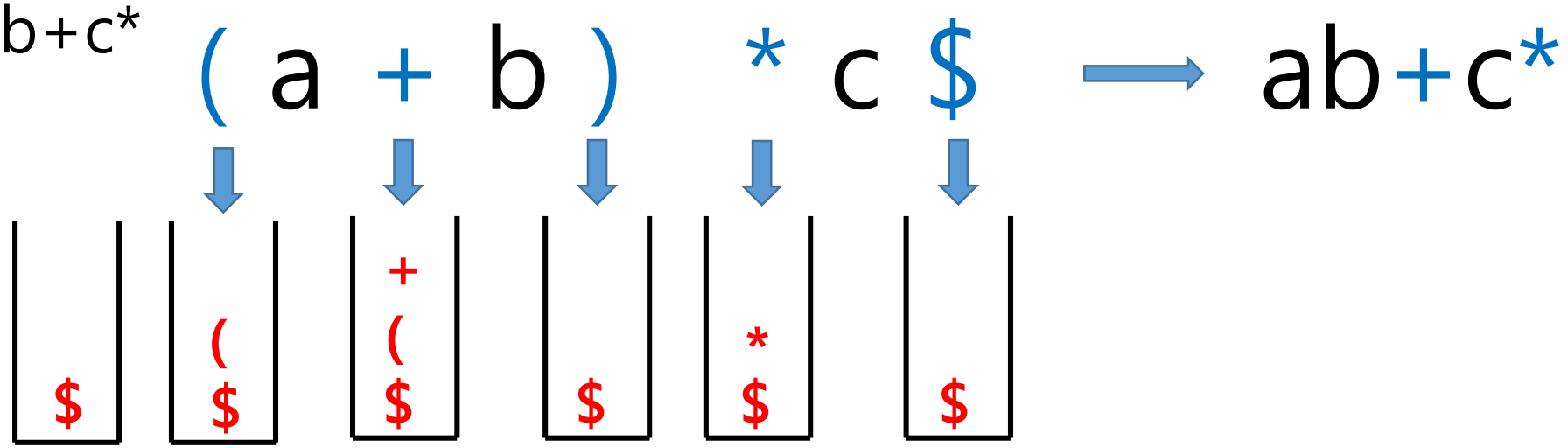
- Initialization of stack: push(end of string)
- End of string
  - isp=lowest: enforce unconditional push of any incoming token
  - icp=lowest: after scanning of infix string is over, enforce unconditional pop of any token in the stack
- (
  - icp=highest: enforce unconditional push of '('
  - isp=lowest: enforce unconditional push of any token after except for ')'

Example

- Precedence

	(	)	+	-	*	/	$(end of string)
in-stack	0	3	1	1	2	2	0
incoming	4	3	1	1	2	2	0

-): special treatment
- Infix: (a+b)*c$
- Postfix: ab+c*



Better presentation

- Stack 변화 및 postfix식 출력 과정
  - Token 별 순차적 표시
- Stack
  - 왼쪽에서 오른쪽으로 성장(grow)
  - 왼쪽 끝: bottom
  - 오른쪽 끝: top

token(infix)	satck	postfix
	$	
(	$(	
a	$(	a
+	$(+	a
b	$(+	ab
)	$	ab+
*	$*	ab+
c	$*	ab+c
$	$	ab+c*

Multiple stacks and queues

- n stacks (or queues) in a single array[0..m-1]
- n stacks
  - Stack 0 to stack n-1
  - Array of top[0..n-1]
  - Array of bottom[0..n-1]
- n circular queues
- 양방향 스택 (n=2)
  - Two stacks: one grows to the other, sharing free space inbetween