

Chap 3. Stacks and Queues (Part 1)

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

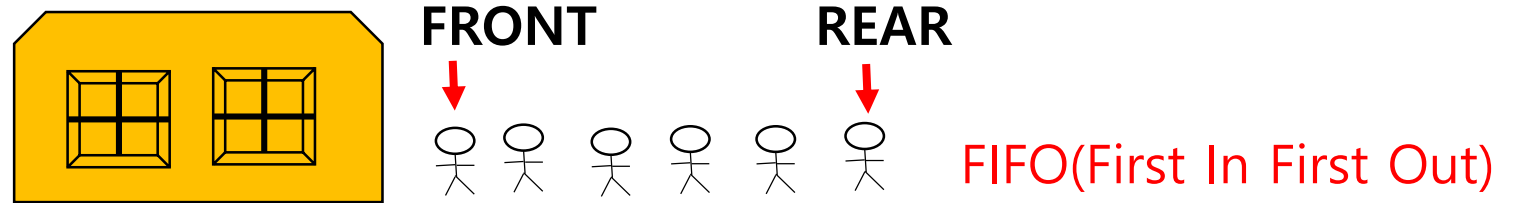
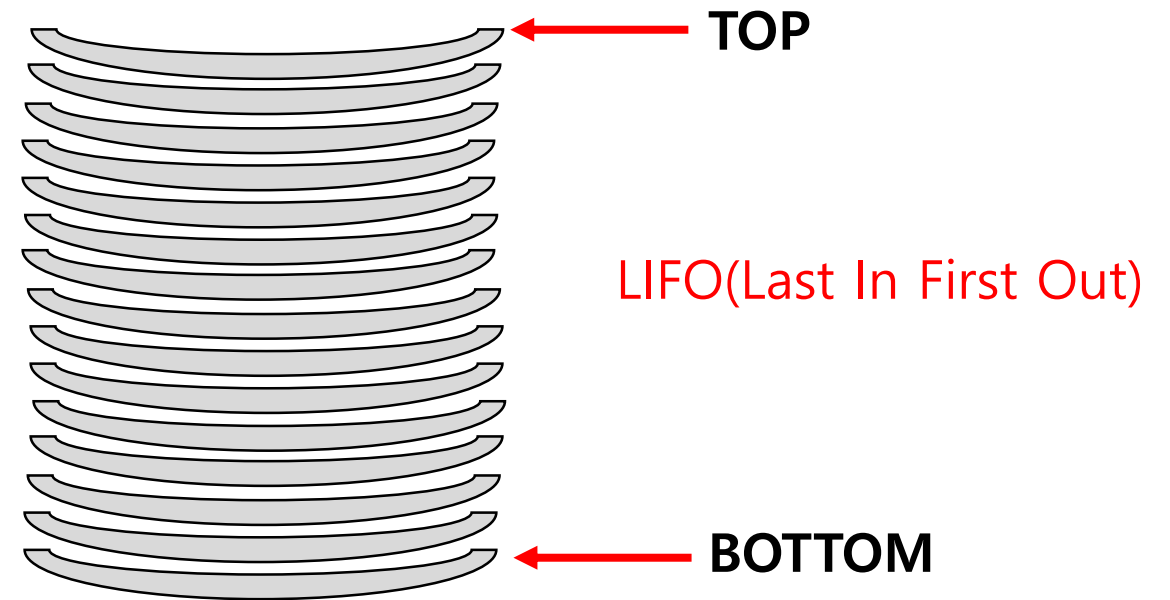
- Real life examples

- Stack

- Stack of dish plates
 - Stack of disks in Towers of Hanoi

- Queue

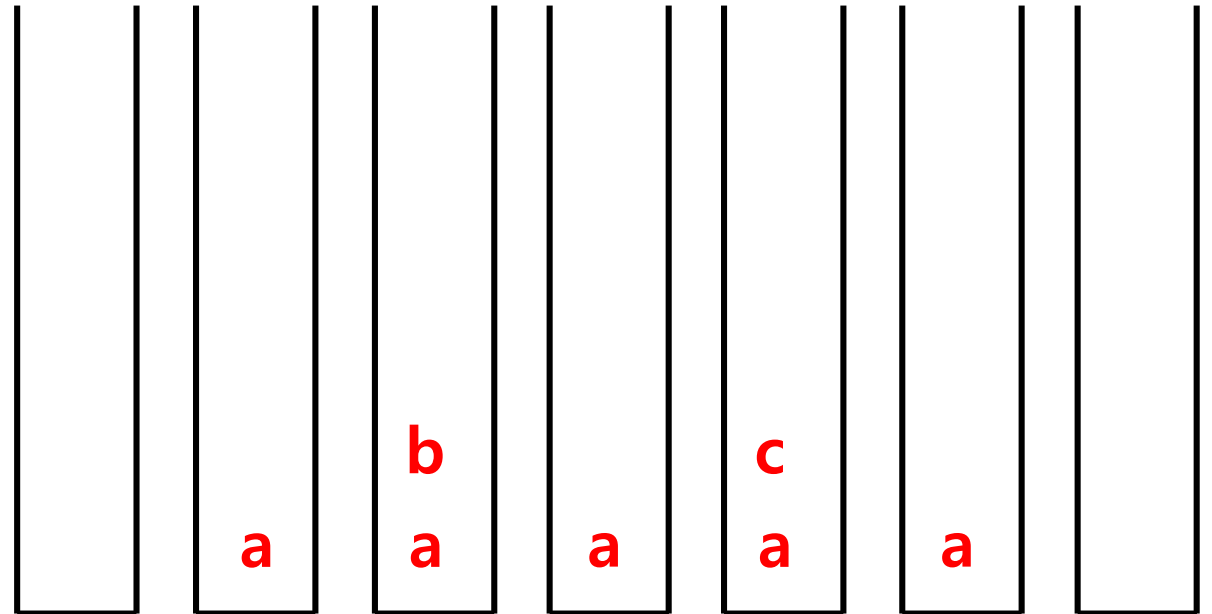
- Queue in a restaurant
 - Queue in a ticket office



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  
}
```

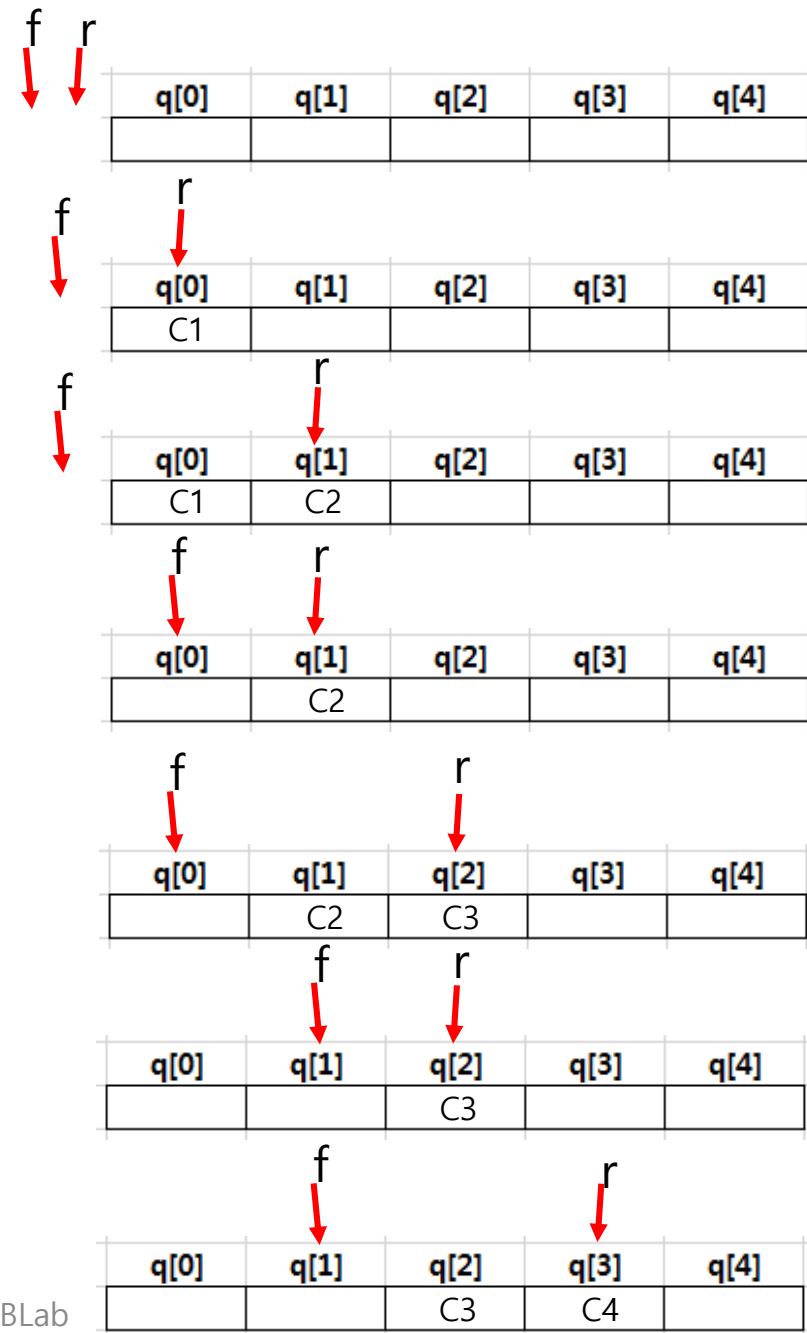
Array Representation of a Queue

- 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  $\leftarrow$  rear + 1  
    queue[rear]  $\leftarrow$  x  
}
```

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

```
queue_full() {  
    if(rear  $\geq$  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?

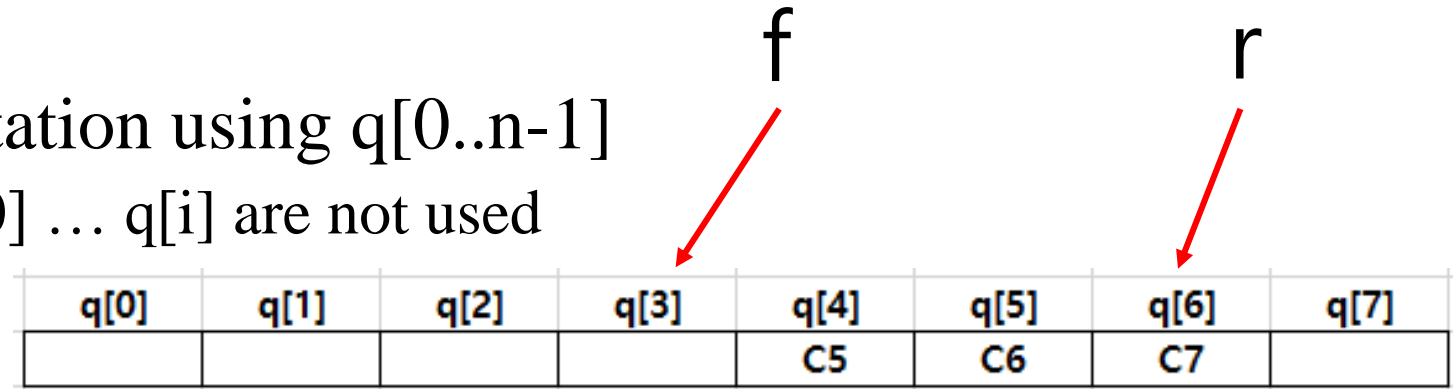
Circular Queue

- Problem in queue representation using $q[0..n-1]$

- When $\text{front} = i$ ($i \geq 0$) : $q[0] \dots q[i]$ are not used

- Eventually,

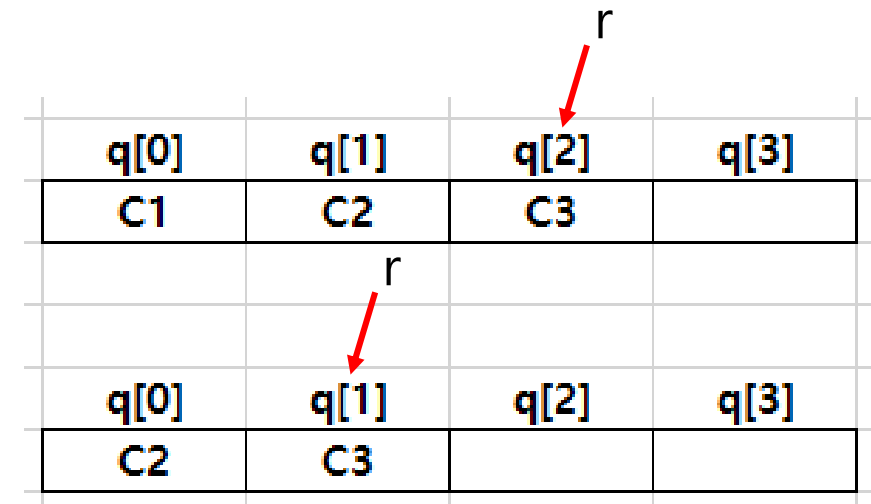
- $\text{front} = \text{rear} = n-1$
- empty queue ($\text{front} = \text{rear}$) as well as
- full queue ($\text{rear} = n-1$) while the entire array is available
- No further $\text{insert}()$ is allowed



- Possible but inefficient solution

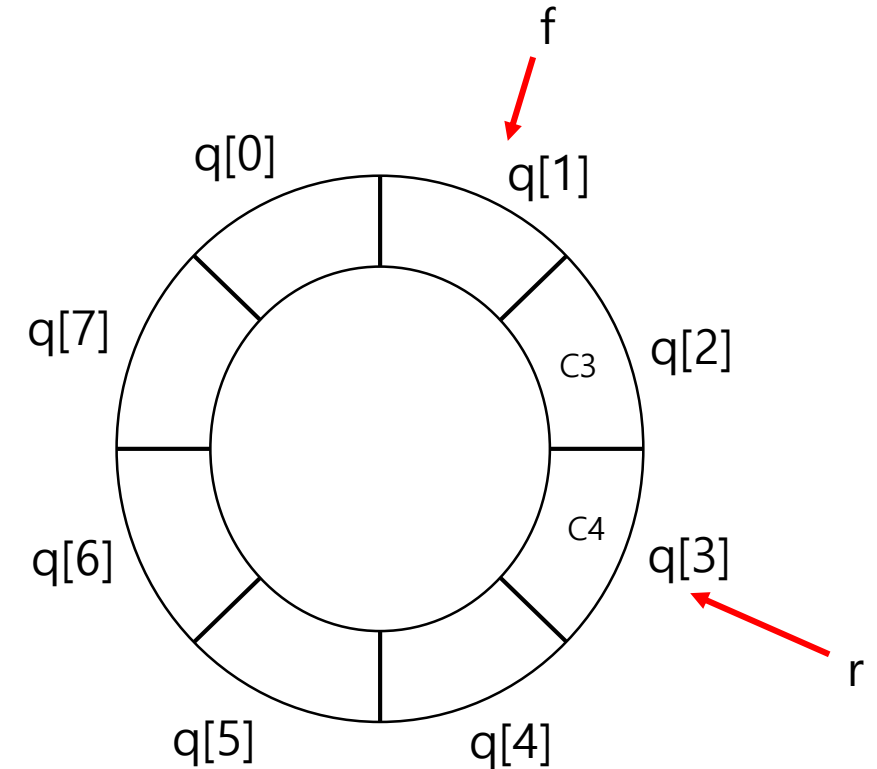
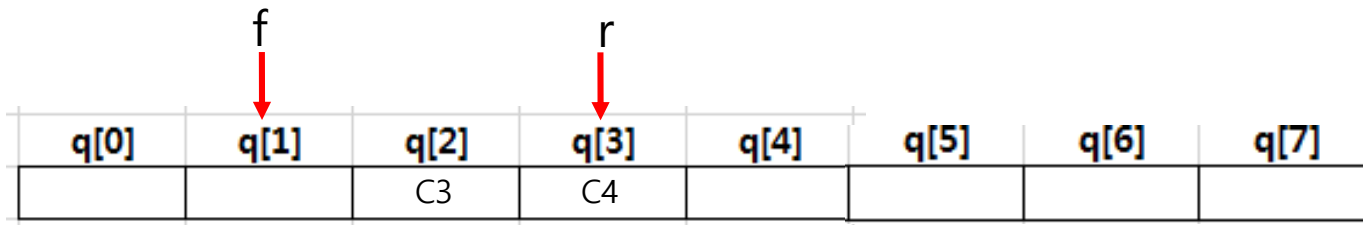
- Shift left every element after $\text{deleteQ}()$
- $\text{front} = -1$ always: no need of maintaining front
- Time complexity of $\text{deleteQ}()$: not $O(1)$ any more

- Circular queue is the solution



Circular Queue

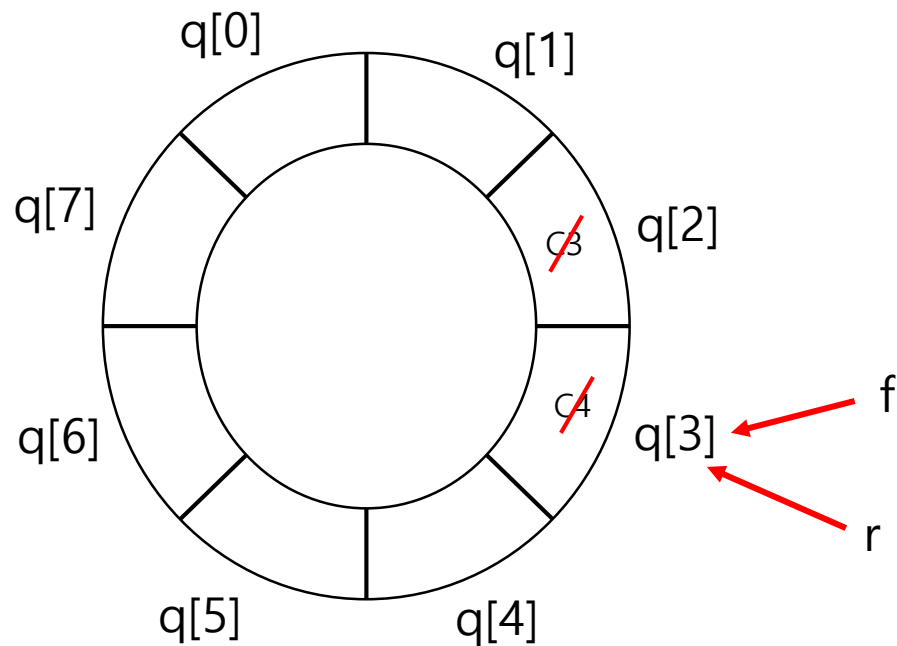
- Array $q[0..n-1]$ is conceived as a circle
- Initialization
 - $\text{front} = \text{rear} = 0$
- Both front & rear moves clockwise
 - $\text{front}(\text{or rear}) \leftarrow \text{front}(\text{or rear}) + 1$
 - $\text{front}(\text{or rear}) \leftarrow (\text{front}(\text{or rear}) + 1) \bmod n$



Circular Queue: one problem

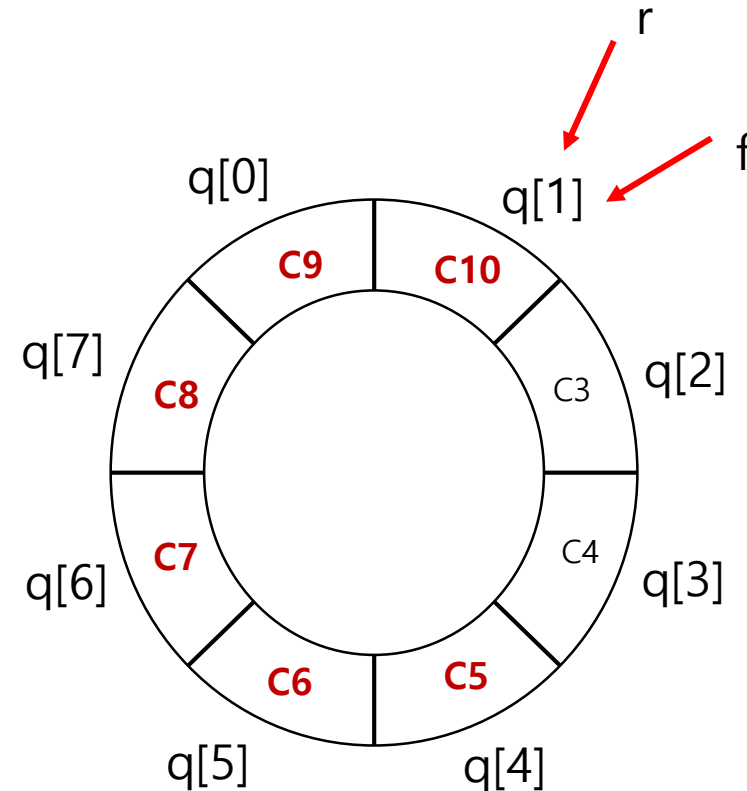
- Queue empty

- After C3 & C4 are deleted
- $\text{front} = \text{rear}$



- Queue full

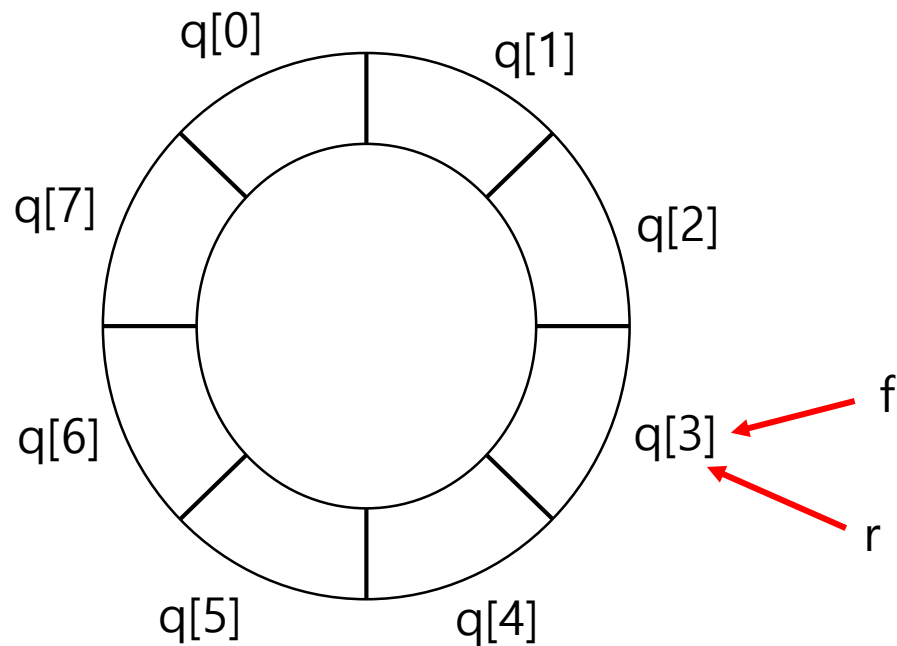
- After C5 to C10 are inserted
- $\text{front} = \text{rear}$



Circular Queue: Empty & Full

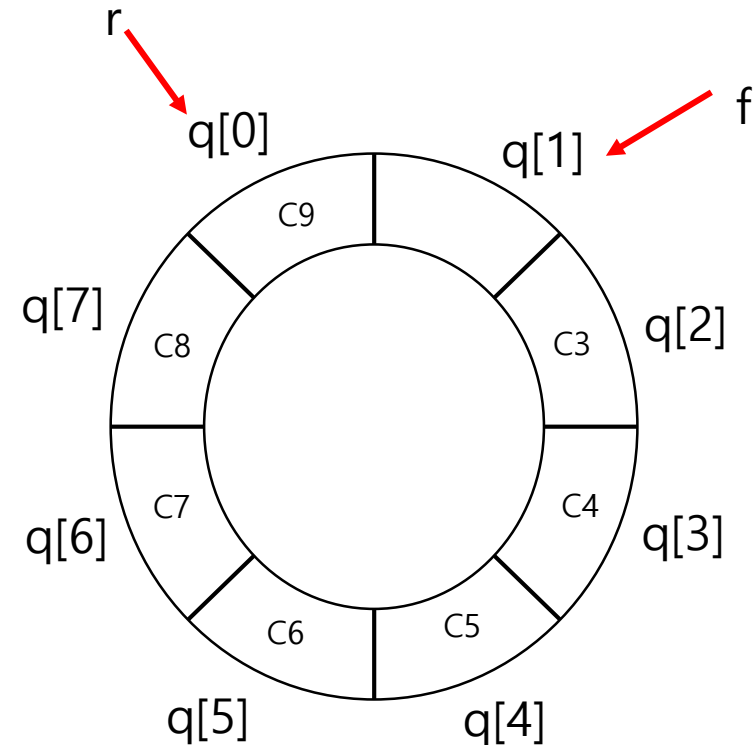
- Queue empty

- $\text{front} = \text{rear}$



- Queue full

- one element of array is *not* used
 - $\text{front} = (\text{rear} + 1) \bmod n$



Circular Queue operations

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

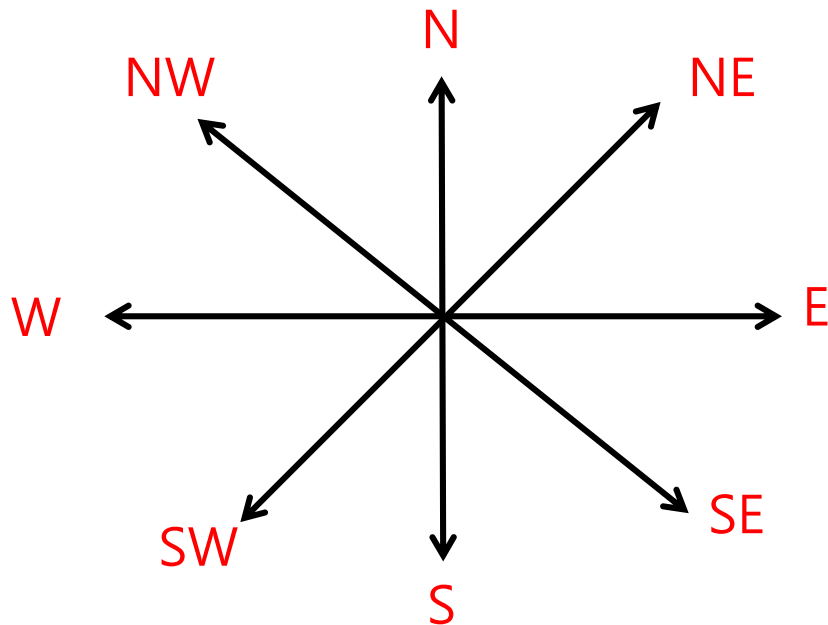
```
deleteQ() {  
    if(queue_empty()) handle_error  
    front  $\leftarrow$  (front + 1) mod n  
    x  $\leftarrow$  queue[front]  
    return(x)  
}
```

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

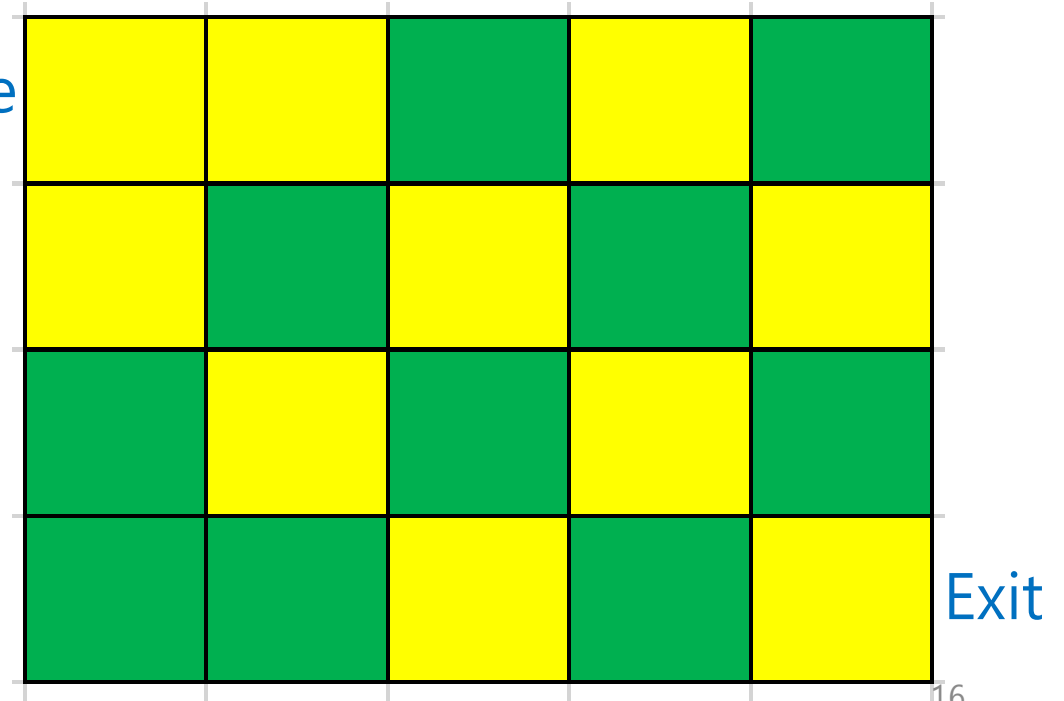
```
queue_empty() {  
    if(front = rear) return TRUE  
    else return FALSE  
}
```

Maze problem

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

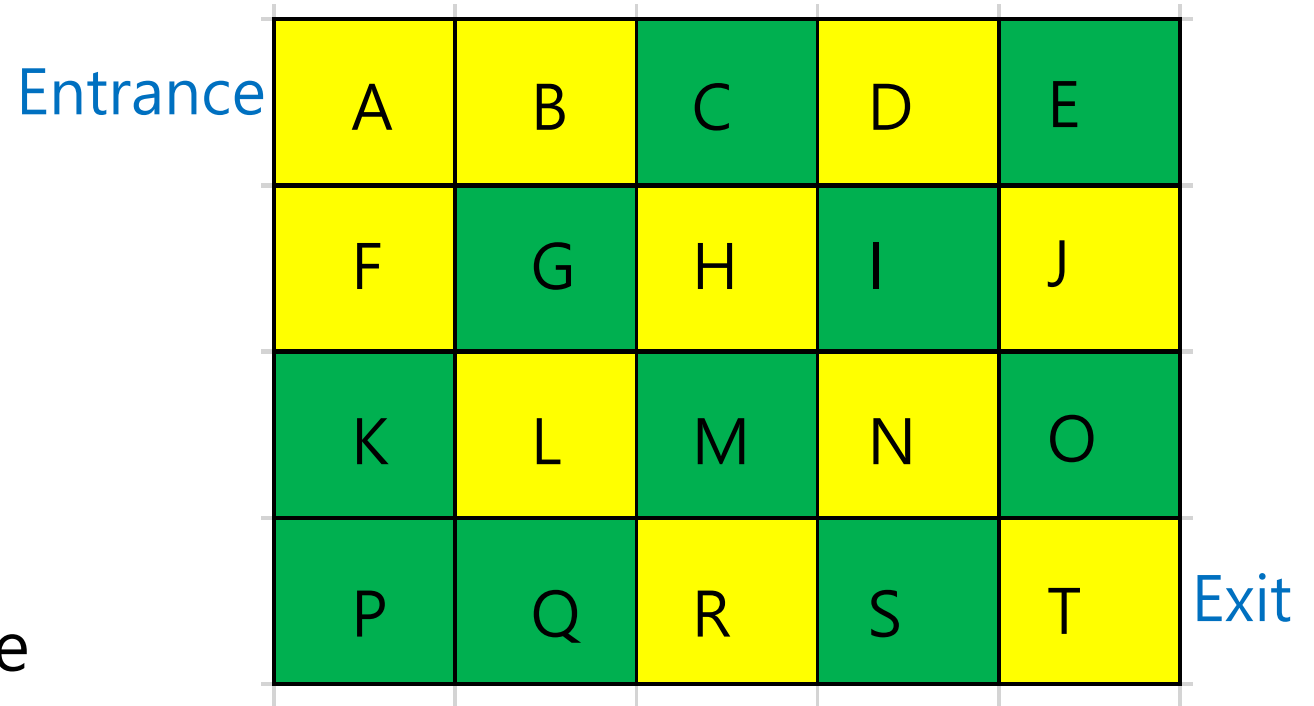


Entrance



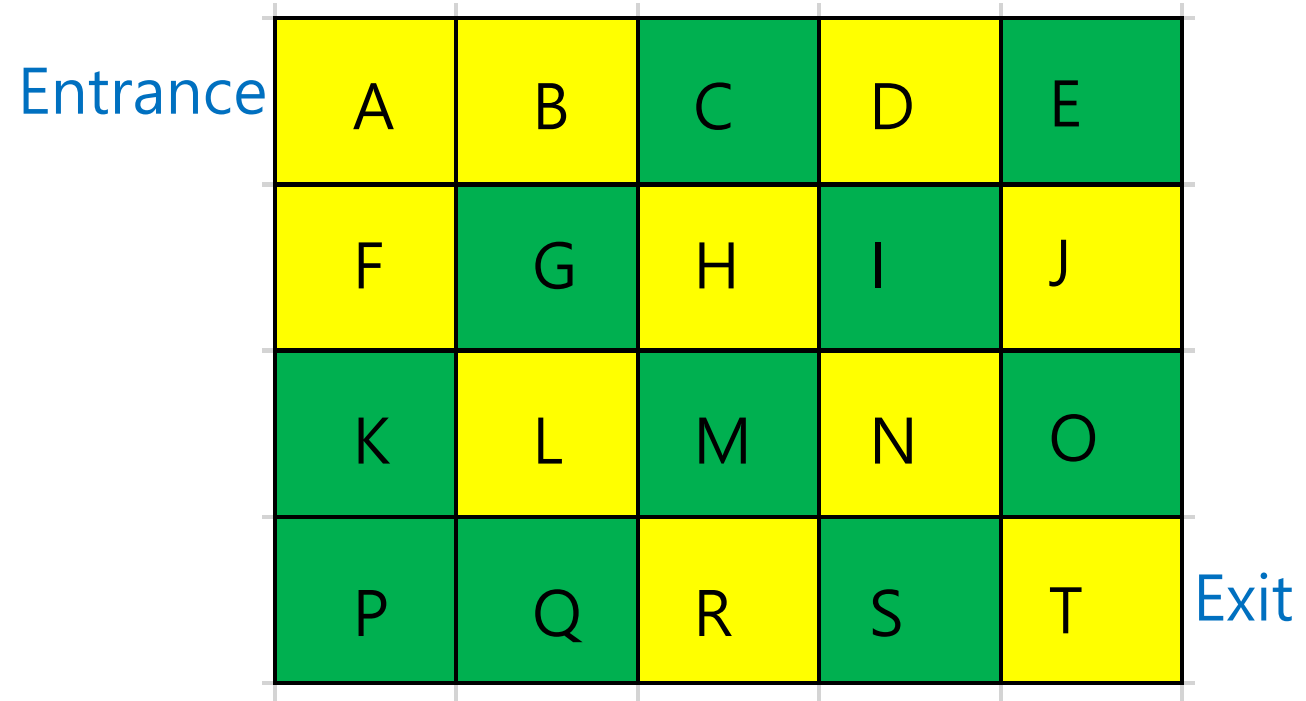
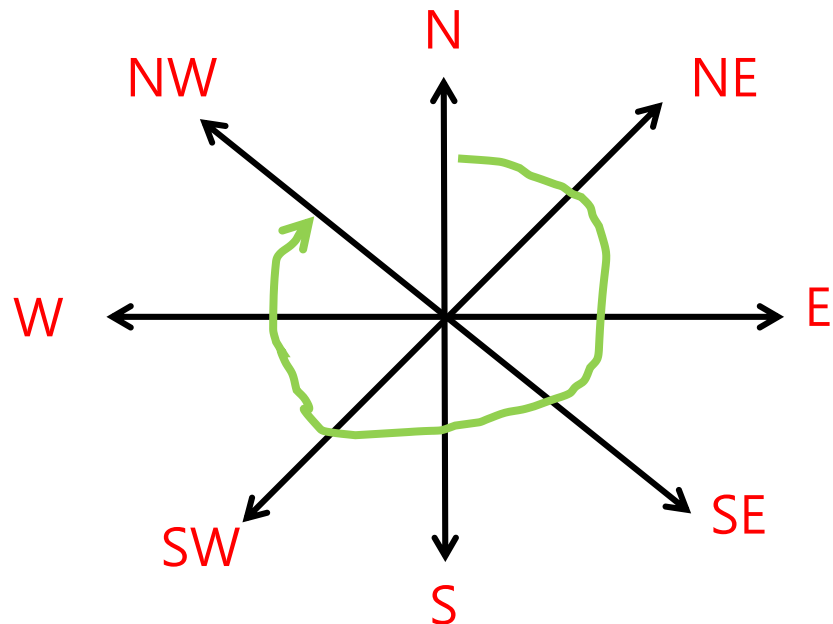
Legal paths

- ABHNT
- ABHLRNT
- ABHDJNT
- AFLRNT
- AFLHNT
- AFLHDJNT
- *Not* interested in paths like
 - AB~~H~~LRN~~H~~DJNT
 - AB~~H~~NJD~~H~~LRNT
- Which one of legal paths would your algorithm find?



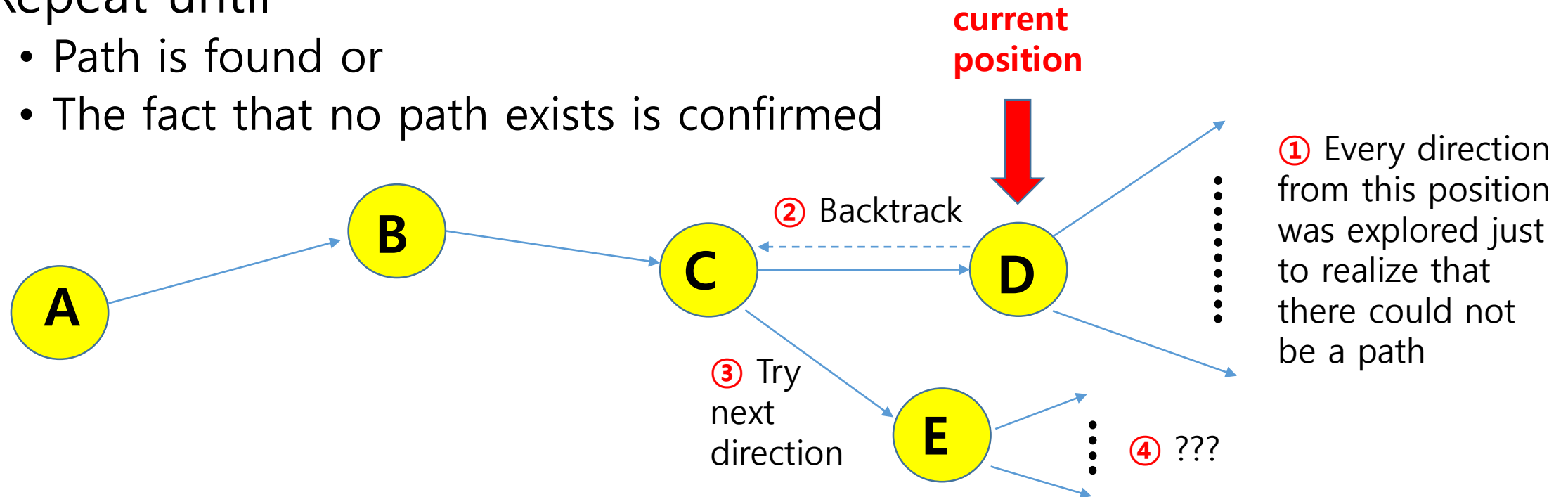
Order of directions to explore

- 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

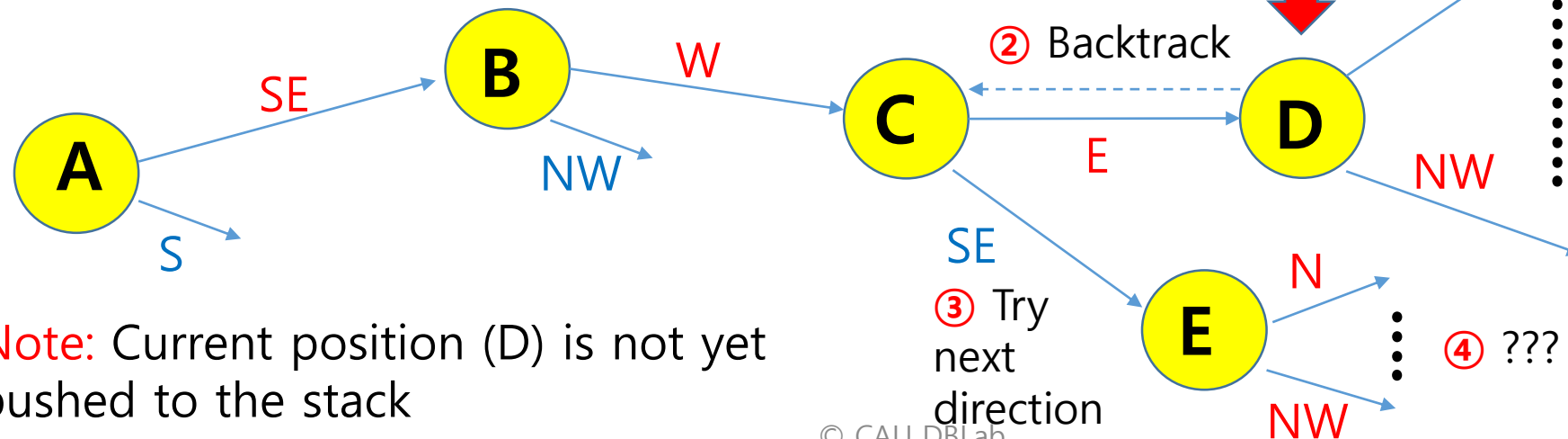
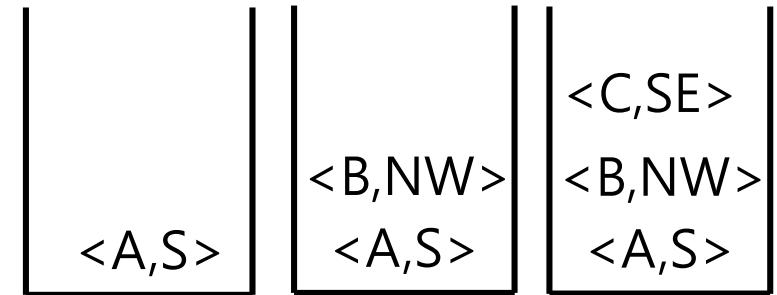


Q&A

- 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., $N \rightarrow NE \rightarrow \dots \rightarrow NW$)
 - Otherwise, backtrack
 - **Q:** backtrack은 어느 위치로?
 - **A:** stack의 top에 저장되어 있는 위치로
 - **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
 - **Q:** stack의 용도를 요약하면?
 - **A:** 현재 탐색 중인 path 및 이동 방향에 대한 정보를 기억

Stack

- Saves
 - Current path
 - i.e., sequence of $\langle \text{position}, \text{direction} \rangle$'s
- Top element of stack $\langle X, d \rangle$
 - backtrack: move back to position X
 - try from direction d

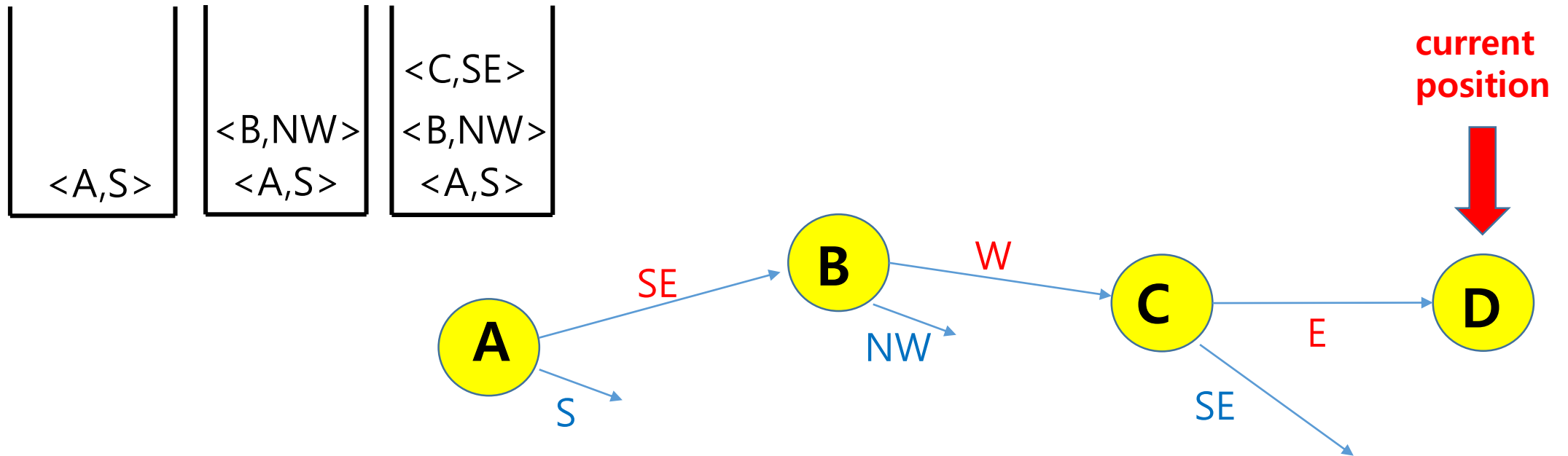


① Every direction from this position was explored just to realize that there could not be a path

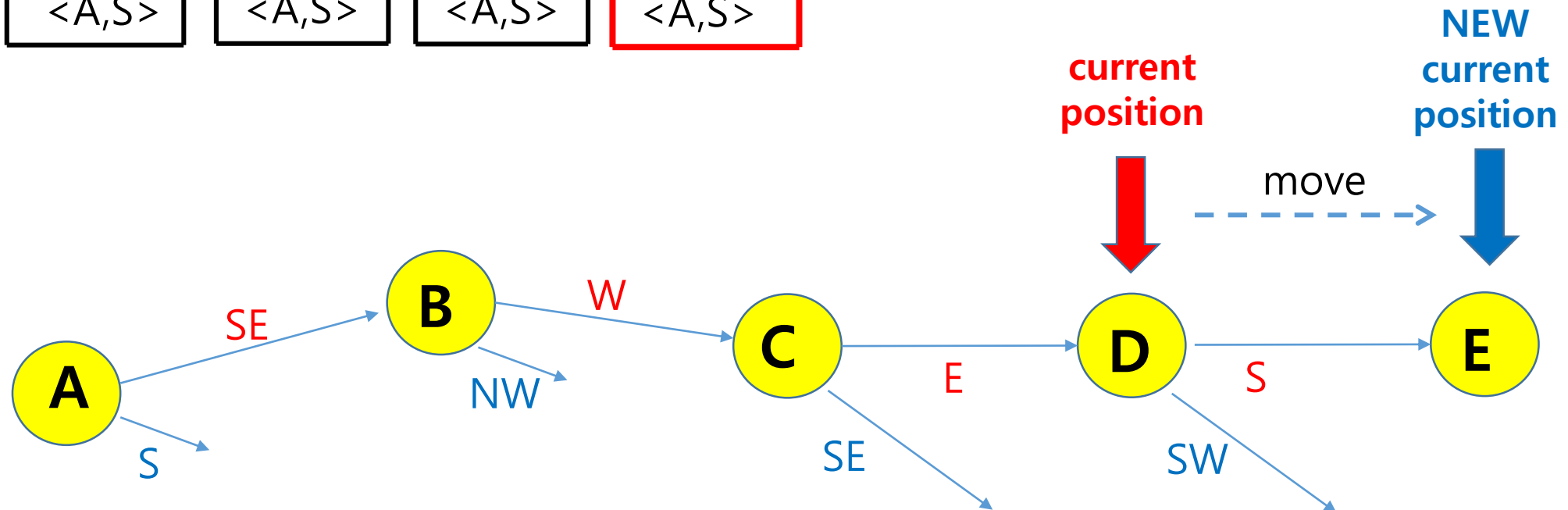
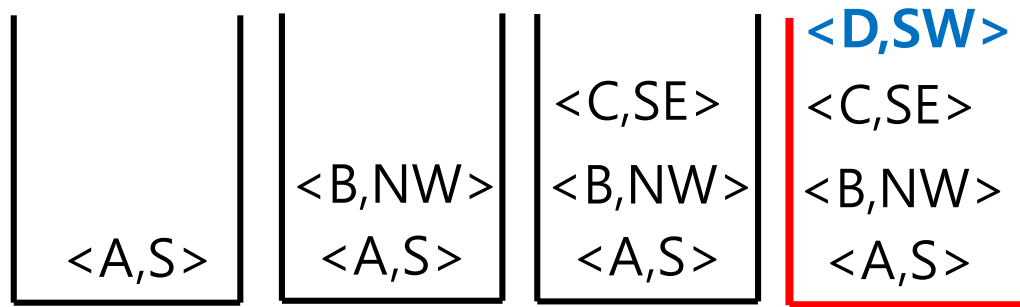
- **Note:** Current position (D) is not yet pushed to the stack

Stack updates

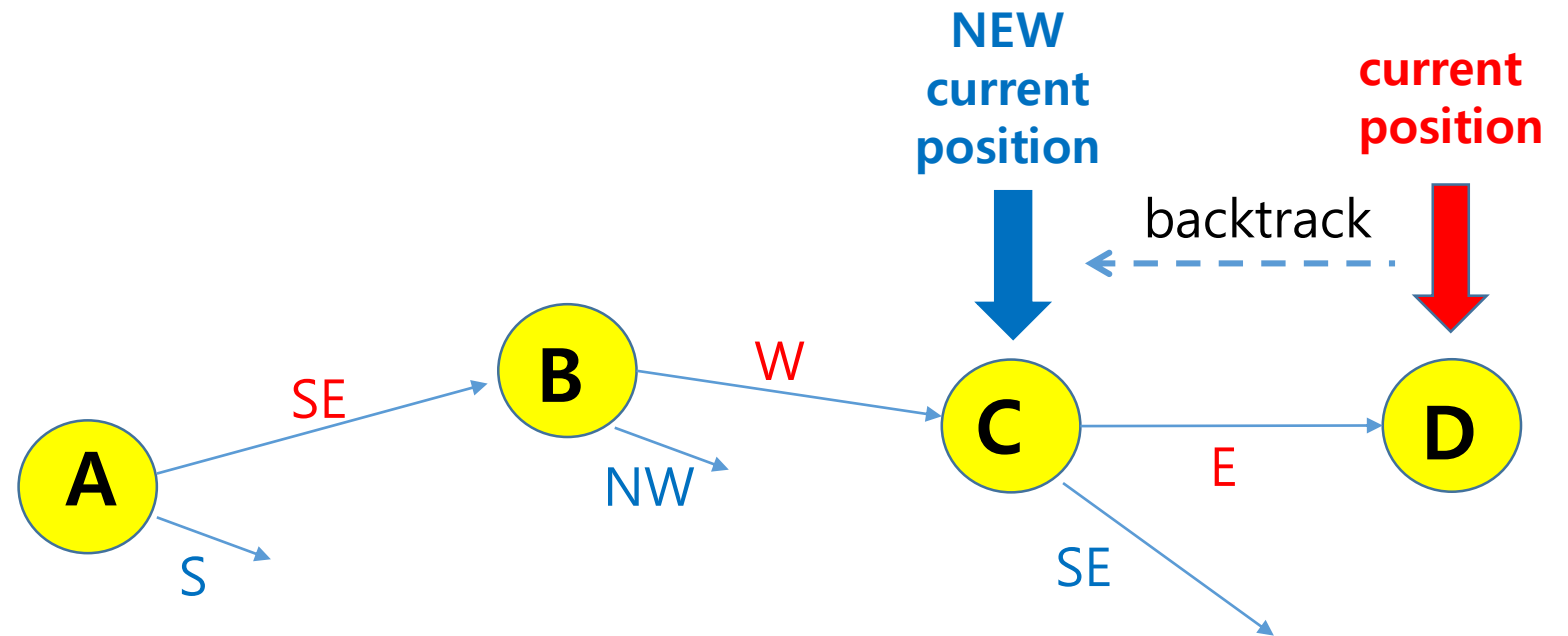
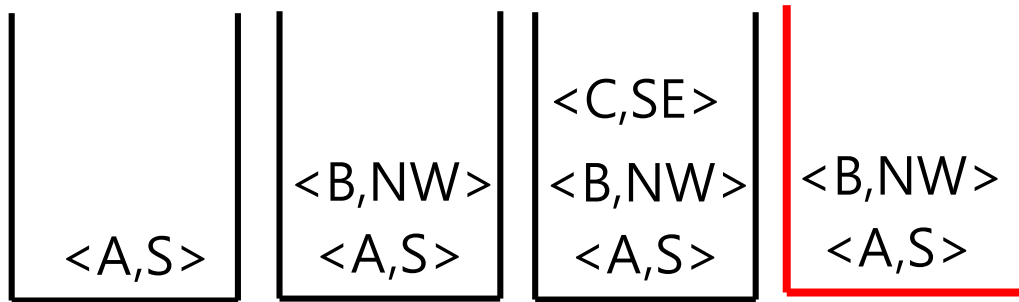
- Done when current position is changed
 - Move: push
 - Backtrack: pop

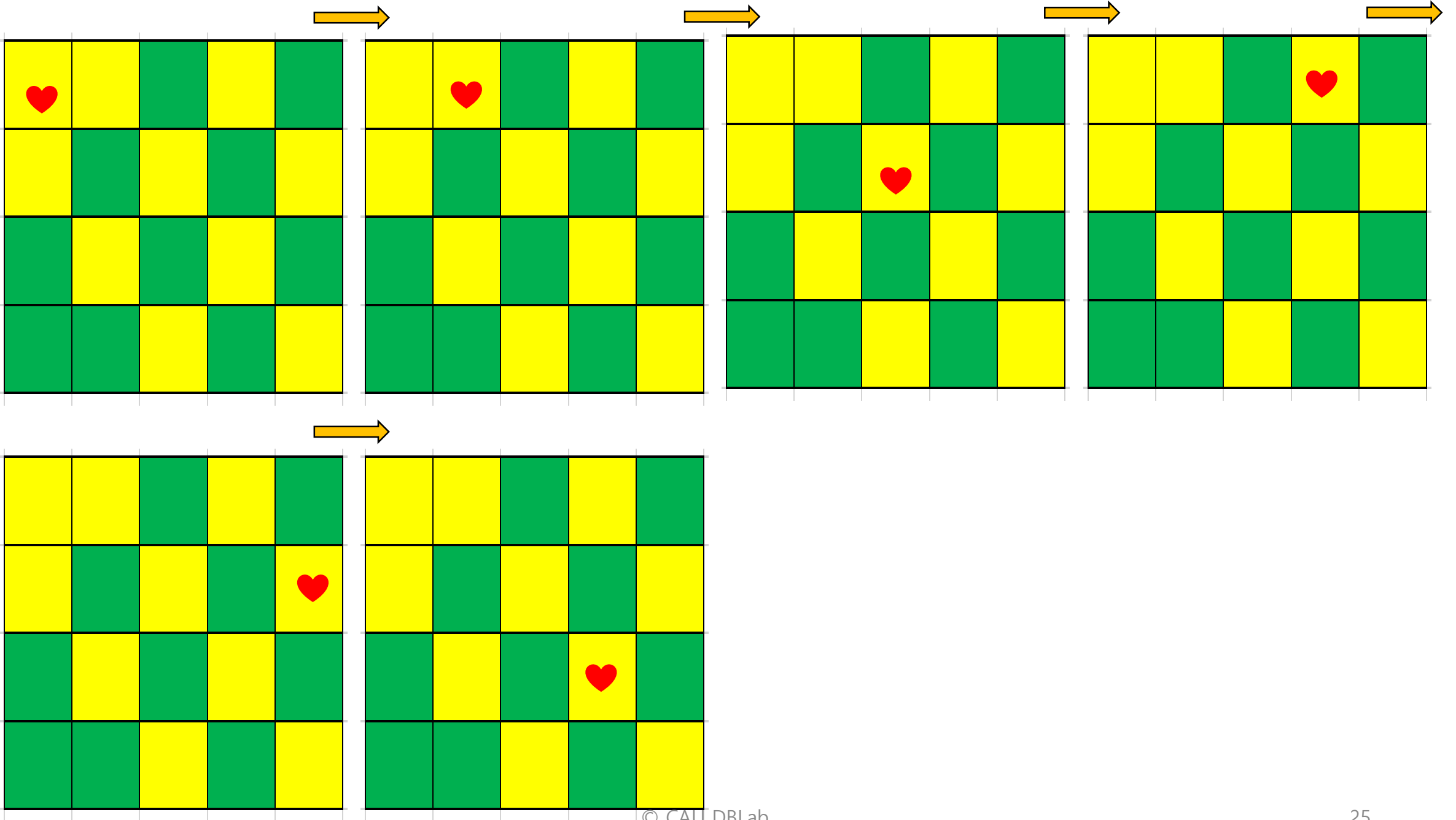


Stack updates: Push



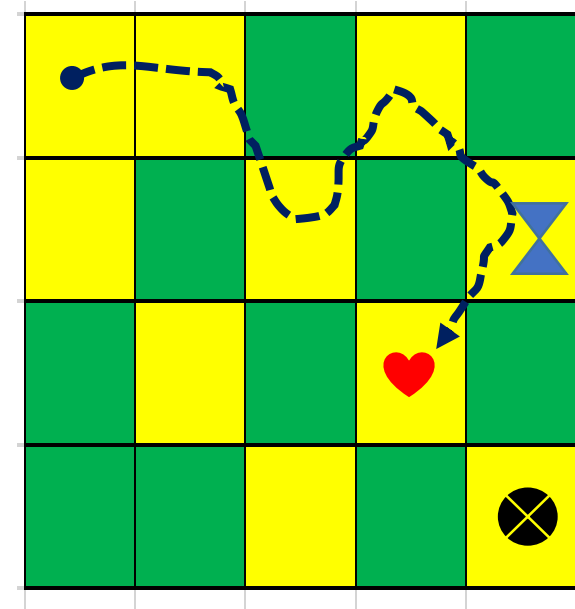
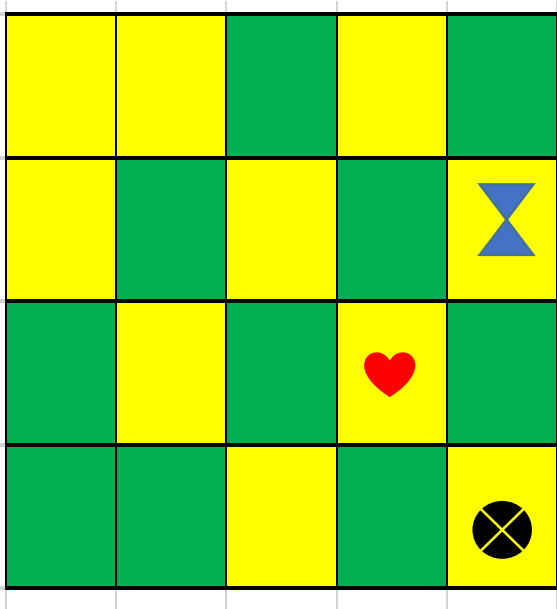
Stack updates: Pop



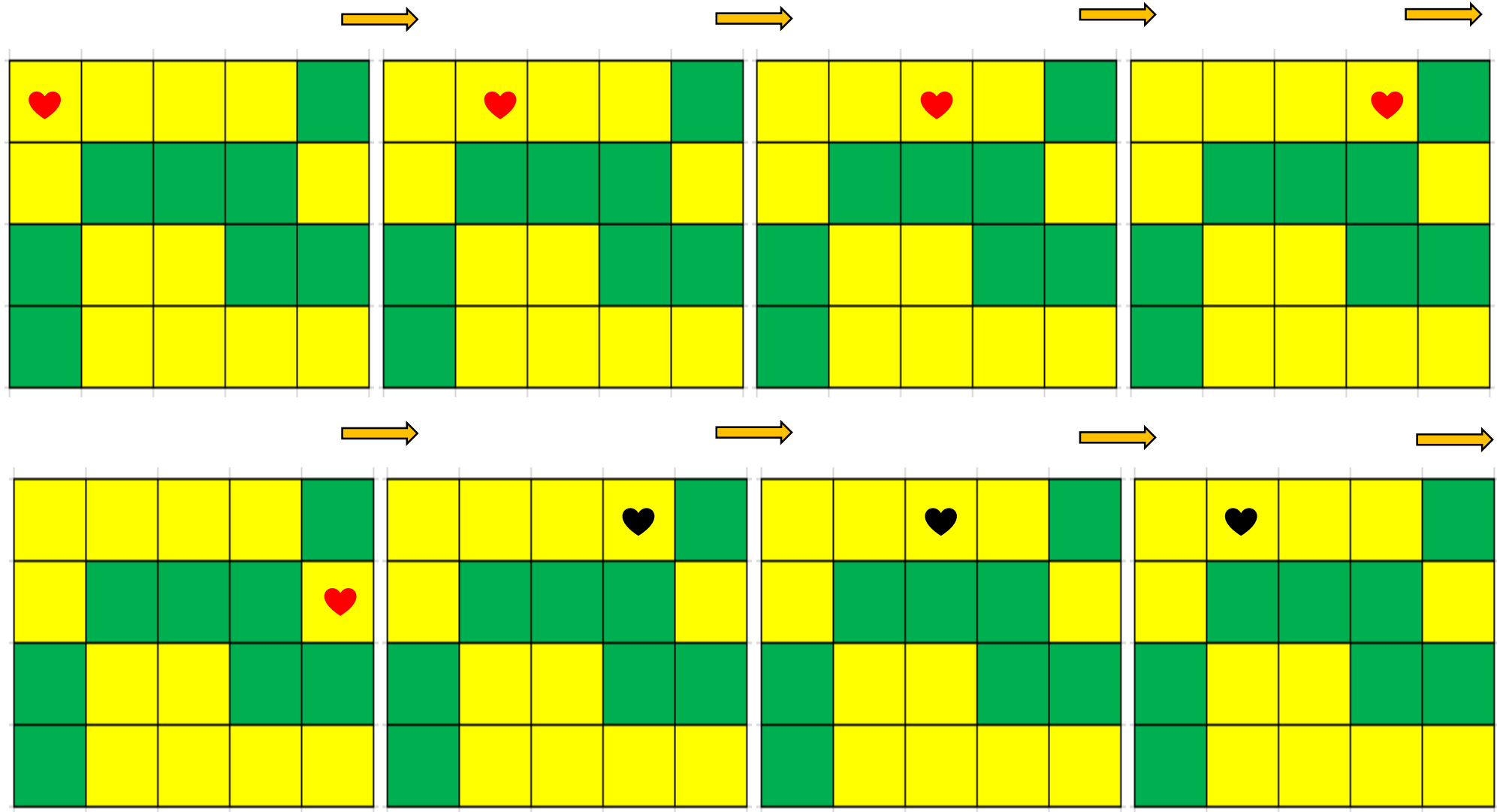


Termination of algorithm

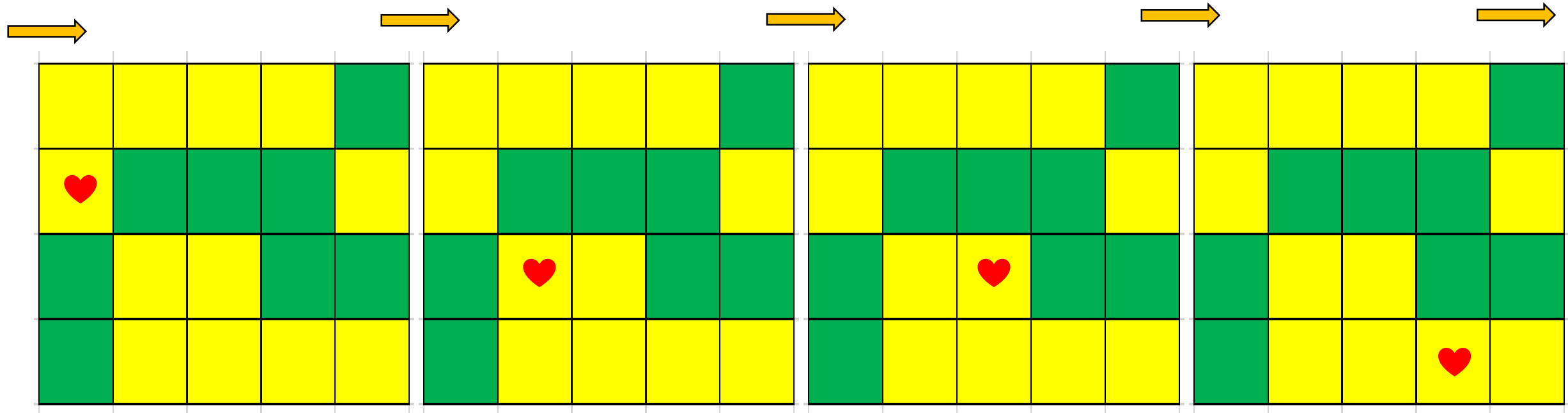
- Current position (♥) is *not* the exit (⊗)
- Top of the Stack
 - *Not* saving the current position (♥)
 - But saves the position of ⋈



Example with backtracking



Example with backtracking (cont'd)

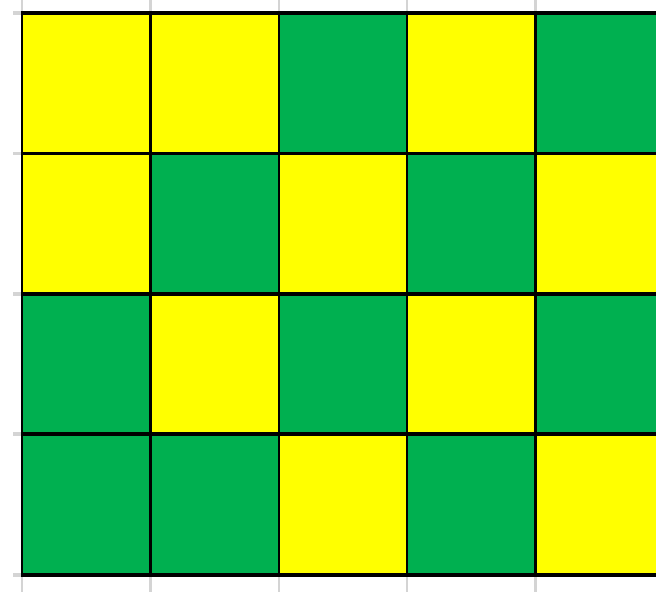


- ♥ Move to a new position
- ♠ backtrack

Data structures

- Representation of a maze
 - 2-dim array `maze[][]`
 - $\text{maze}[i][j] = \begin{cases} 0 & \text{if open} \\ 1 & \text{if blocked} \end{cases}$
 - `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>`

Entrance

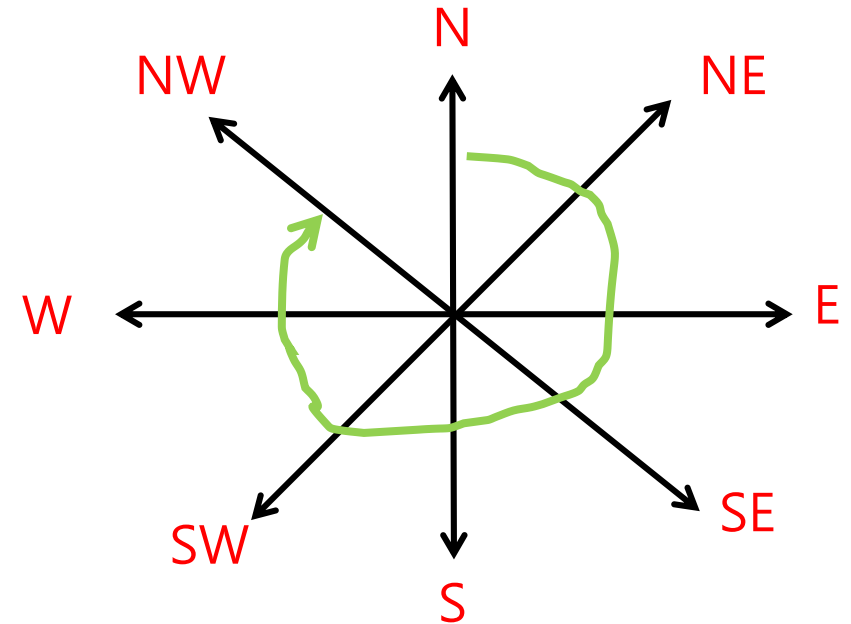


Exit

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 $\langle i, j \rangle$: `mark[i][j] ← 1`
 - Condition that move to position $\langle i, j \rangle$ is legal
 - `maze[i][j]=0 AND mark[i][j]=0`
- Order of moving directions
 - Assign integers in increasing order
 - $\{N, NE, \dots, NW\} = \{0, 1, \dots, 7\}$
 - Next direction \leftarrow current direction + 1
- Type of stack elements
 - $\langle \text{position}, \text{direction} \rangle$
 - $\langle \text{row}, \text{col}, \text{direction} \rangle$
- 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

current
position



P

d

EXIT



Q

① Q is the exit
A path is found!

current
position



P

legal
move

d

new
current
position



Q

North

d'

Push(<P, d'>)

② Q is *not* the exit &
move from P to Q is legal

current
position



P

illegal
move

d

Q

d'

③ Q is *not* the exit &&
move from P to Q is *illegal*

Pseudo code of maze algorithm

initialize stack & enter the maze, marking the entrance position $\langle 1,1 \rangle$ as visited

while (stack is not empty && you have not yet found the path) {

pop() to get the position P to move back to & direction d to move from P
backtrack to P

while(d is legitimate direction && you have not yet found the path) {

figure out the new position Q using the current position P and d

① **if** (Q is the exit) //now you have found the path

else if (move from P to Q is legal) {

push() to save P and d' (next direction after d) //for later backtrack

move to Q //meaning that Q becomes the new current position P (i.e., $P \leftarrow Q$)

mark Q as visited

d \leftarrow North //first direction to try from the new current position P

}

③ **else** d \leftarrow d' (next direction after d)

}

}

$\langle 1,1 \rangle$, N

$\langle 1,1 \rangle$, NE

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: $+ab$

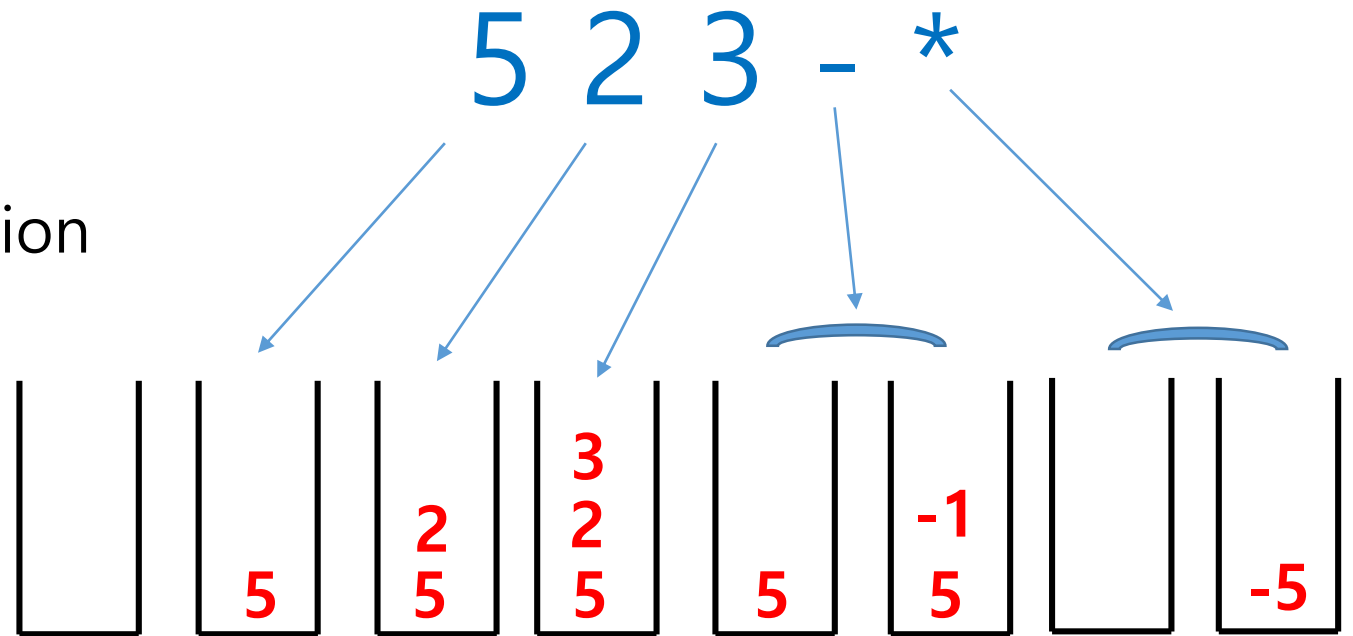
Postfix expression

- 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

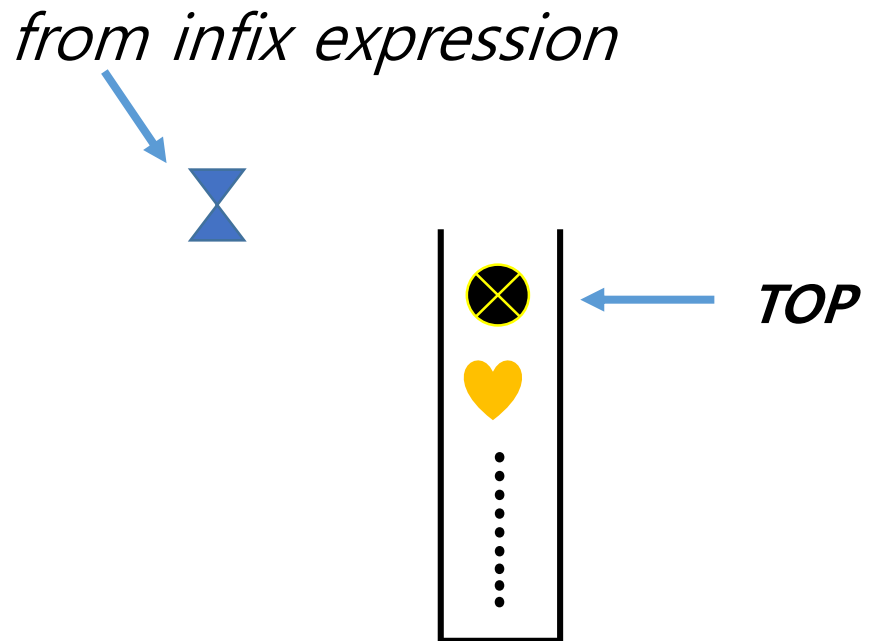
- 1-pass scan of postfix expression from left to right
- Cases:
 - Operand X: push(X)
 - Binary operator \odot :
 1. $Z \leftarrow \text{pop()} // \text{Operand2}$
 2. $Y \leftarrow \text{pop()} // \text{Operand1}$
 3. $W \leftarrow Y \odot Z$
 4. Push(W)
- Termination: end of expression
- Final result 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 \odot or '('
 - **in-stack precedence** of stack[top] \geq **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., $\text{icp}(+) < \text{isp}(*)$
- Example

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

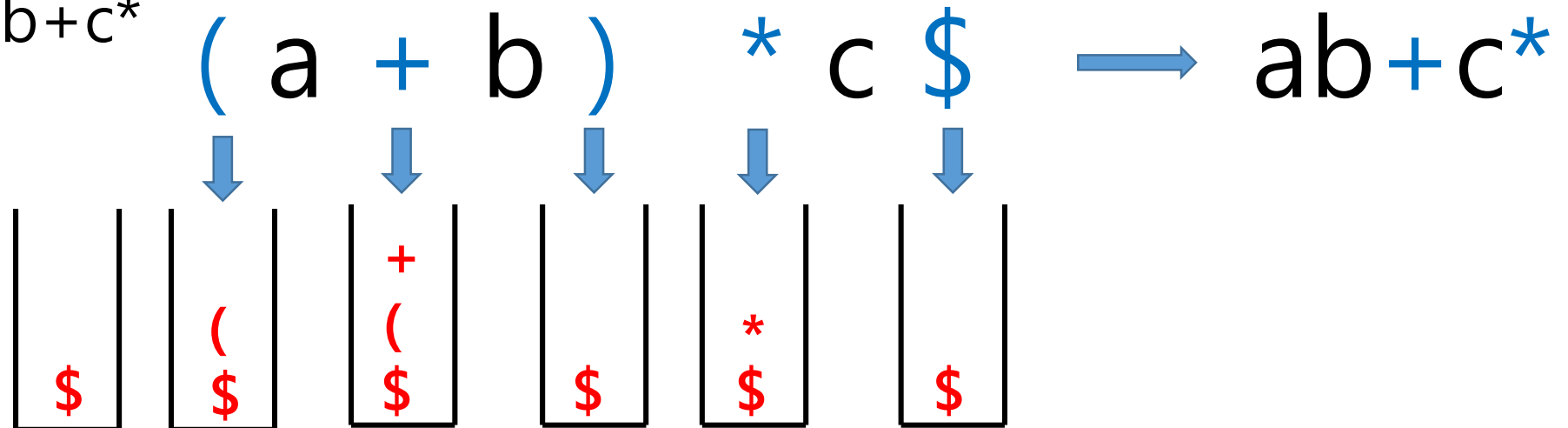
- Initialization of stack: push(end of string)
- End of string
 - $\text{isp}=\text{lowest}$: enforce unconditional push of any incoming token
 - $\text{icp}=\text{lowest}$: after scanning of infix string is over, enforce unconditional pop of any token in the stack
- (
 - $\text{icp}=\text{highest}$: enforce unconditional push of '('
 - $\text{isp}=\text{lowest}$: 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]
- n circular queues
- 양방향 스택 (n=2)
 - Two stacks: one grows to the other, sharing free space inbetween

