HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCUTURES AND ALGORITHMS

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

Week 8: Stack and Queue

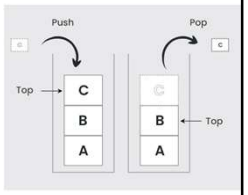
ONE LOVE. ONE FUTURE.

CONTENT

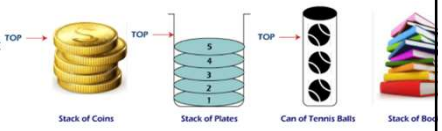
- **Stack**
- Exercise: Check the symmetry of the parentheses
- Queue
- Exercise: Find the fastest way out of the maze

STACK

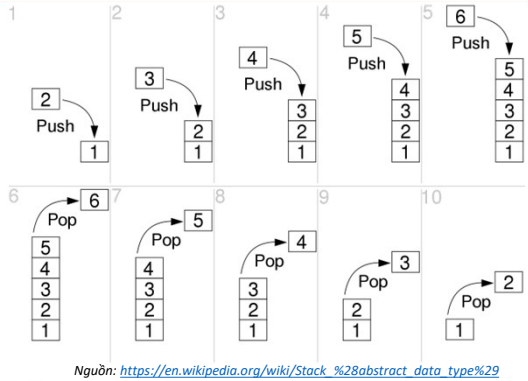
- A stack is a Linear Data Structure
- The operation of adding and removing elements on the list is performed at one end (top) of the list according to the Last In First Out principle (the last element added to the stack is the first element to be removed)).



- Basic operations with stack *S*:
  - *Push(x, S)*: insert element x into the stack
  - *Pop(S)*: remove the first element from the stack
  - *Top(S)*: Access the element at the top of the sta
  - *isEmpty(S)*: Returns true if the stack is empty



STACK



Nguồn: https://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29

IMPLEMENT STACKS WITH SINGLY LINKED LIST

- Data structures and Initialization

```
typedef struct Node{
    char value;
    struct Node* next;
}Node;

Node*makeNode(char v){
    Node* p = (Node*)malloc(sizeof(Node));
    p->value = v;
    p->next = NULL;
    return p;
}
```

Declare a struct for each element of a stack

Create a node of the stack

IMPLEMENT STACKS WITH SINGLY LINKED LIST

- Push

```
Node* push(Node * head, char v){
    Node * new_node = makeNode(v);
    if(head == NULL) return new_node;
    else{
        new_node->next = head;
        head = new_node;
        return head;
    }
}
```

Create a new node

If the stack is empty, return the created node

If the stack is not empty, the newly created node becomes the head of the list

IMPLEMENT STACKS WITH SINGLY LINKED LIST

- Pop

```
Node* pop(Node* head){
    if(head == NULL) return head;
    Node* p = head;
    head = head->next;
    free(p);
    return head;
}
```

If the stack is empty, return null

If the stack is not empty, delete the top element of the stack and return the top element of the stack

IMPLEMENT STACKS WITH SINGLY LINKED LIST

- Top and IsEmpty

```
char top(Node* head){
   return head->value;
}

bool isEmpty(Node* head){
    if(head == NULL) return true;

    return false;
}
```


SOME APPLICATIONS OF STACKS

- **Function call stack**: Stack is widely used to manage function calls and local variables in programming languages. When a function is called, its context (including arguments and local variables) is pushed onto the stack. When the function returns, its context is pushed off the stack, allowing correct nesting of the function.
- **Undo mechanism**: Stacks are used in applications that require the Undo feature, such as text editors, graphics software, or version management systems. Each user action can be pushed onto the stack, and undoing an action involves taking it off the stack to revert the change.

SOME APPLICATIONS OF STACKS

- **Backtracking Algorithms**: In algorithms like depth-first search (DFS) and backtracking algorithms (For example, solving puzzles like the N-Queens problem or Sudoku), a stack can be used to keeps track of which nodes or states have been visited. This allows for easy backtracking to explore alternatives when needed.
- **Expression Analysis**: Stack is used to analyze and understand expressions in compilers and interpreters. They help maintain operator precedence and evaluate expressions correctly.

SOME APPLICATIONS OF STACKS

- **Memory Management**: Stack plays an important role in memory management in computer systems. They are used to manage the function call stack, which stores information about function calls and local variables. The stack helps allocate memory for function calls and free it when functions return, preventing memory leaks.
- **Expression Analysis**: Stack is used to analyze and understand expressions in compilers and interpreters. They help maintain operator precedence and evaluate expressions correctly.

CONTENT

- Stack
- **Exercise: Check the symmetry of the parentheses**
- Queue
- Exercise: Find the fastest way out of the maze

EXERCISE: CHECK THE SYMMETRY OF THE PARENTHESES

- Given a sequence of brackets E where each element is a bracket of one of the following types: (,), [,], {, }. Write a program to check whether the parentheses are symmetric or not?
- Example:
  - ( ) [{ } ([ ])]: symmetric
  - ( ) [{ } ([ ] }]: non-symmetric

EXERCISE: CHECK THE SYMMETRY OF THE PARENTHESES

- Data:
  - A single line containing a string of characters representing a sequence of parentheses
- Result:
  - Write 1 if the bracket sequence is symmetrical and 0 if the bracket sequence is not symmetrical

stdin	stdout
( ) [{ } ([ ])]	1

stdin	stdout
( ) [{ } ([ ] }]	0

EXERCISE: CHECK THE SYMMETRY OF THE PARENTHESES

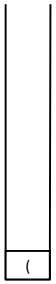
- **Algorithm**
  - Initialize an empty stack S
  - Browse the parentheses from left to right
    - If you encounter an opening parenthesis A, put that opening parenthesis in S
    - If you encounter a closing parenthesis B
      - If S is empty, the conclusion is that the parenthesis sequence E is not symmetrical
      - If S is not empty
        - Takes an opening parenthesis A from the stack S
        - If A and B are not symmetrical (opening and closing brackets are of different types), then the conclusion is that E is not symmetrical
  - At the end of the review, if S is not empty, the conclusion is that E is not symmetric, otherwise E is symmetric

EXAMPLE

- **EXAMPLE 1:** Sequence ( ) [({ })]
  - Initialize an empty stack, traverse the parentheses from left to right

EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack

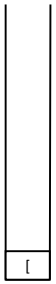


EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack
  - Step 2: Consider the next parenthesis as the closing parenthesis ")" -> remove the opening parenthesis from the stack and match "(" ")" -> the result is correct so we continue browsing

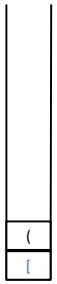
EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack
  - Step 2: Consider the next parenthesis as the closing parenthesis ")" -> remove the opening parenthesis from the stack and match "(" ")" -> the result is correct so we continue browsing
  - Step 3: Encounter opening parenthesis "[" -> put this opening parenthesis on the stack



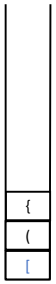
EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack
  - Step 2: Consider the next parenthesis as the closing parenthesis ")" -> remove the opening parenthesis from the stack and match "(" ")" -> the result is correct so we continue browsing
  - Step 3: Encounter opening parenthesis "[" -> put this opening parenthesis on the stack
  - Step 4: Encounter the opening parenthesis "(" □ put this opening parenthesis on the stack



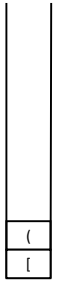
EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack
  - Step 2: Consider the next parenthesis as the closing parenthesis ")" -> remove the opening parenthesis from the stack and match "(" ")" -> the result is correct so we continue browsing
  - Step 3: Encounter opening parenthesis "[" -> put this opening parenthesis on the stack
  - Step 4: Encounter the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 5: Encounter opening parenthesis "{" -> put this opening parenthesis on the stack



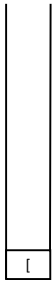
EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 1: Consider the next parenthesis as the opening parenthesis "(" -> put the opening parenthesis on the stack
  - Step 2: Consider the next parenthesis as the closing parenthesis ")" -> remove the opening parenthesis from the stack and match "(" ")" -> the result is correct so we continue browsing
  - Step 3: Encounter opening parenthesis "[" -> put this opening parenthesis on the stack
  - Step 4: Encounter the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 5: Encounter opening parenthesis "{" -> put this opening parenthesis on the stack
  - Step 6: Encounter a closing parenthesis "}" -> take an opening parenthesis "{" out of the stack
    - Matches "{" "}" -> the result is correct -> continue



EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 6: Encounter a closing parenthesis "}" -> take an opening parenthesis "{" out of the stack
    - Matches "{" "}" -> the result is correct -> continue
  - Step 7: Encounter a closing parenthesis ")" -> take an opening parenthesis "(" out of the stack
    - Matches "(" ")" -> the result is correct -> continue

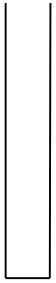


EXAMPLE

- **Example 1:** sequence ( ) [({ })]
  - Step 6: Encounter a closing parenthesis "}" -> take an opening parenthesis "{" out of the stack
    - Matches "{" "}" -> the result is correct -> continue
  - Step 7: Encounter a closing parenthesis ")" -> take an opening parenthesis "(" out of the stack
    - Matches "(" ")" -> the result is correct -> continue
  - Step 8: Encounter a closing parenthesis "]" -> take an opening parenthesis "[" out of the stack
    - Match "[" "]" -> now the parenthesis sequence has been reviewed and the stack is empty, so the result of this parenthesis sequence is symmetrical

EXAMPLE

- **Example 2:** Sequence ( ) [({ })]
  - Initialize an empty stack and traverse the parentheses from left to right

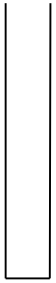


EXAMPLE

- **Example 2:** sequence ( ) [({ })]
  - Step 1: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack

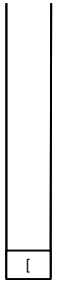
EXAMPLE

- **Example 2:** sequence ( ) [( )])
  - Step 1: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 2: Encounter the next closing parenthesis ")" -> take an opening parenthesis "(" out of the stack



EXAMPLE

- **Example 2:** sequence ( ) [( )])
  - Step 1: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 2: Encounter the next closing parenthesis ")" -> take an opening parenthesis "(" out of the stack
  - Step 3: Encounter the next parenthesis, the opening parenthesis "[" -> put this opening parenthesis on the stack

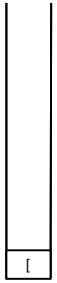


EXAMPLE

- **Example 2:** sequence ( ) [( )])
  - Step 1: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 2: Encounter the next closing parenthesis ")" -> take an opening parenthesis "(" out of the stack
  - Step 3: Encounter the next parenthesis, the opening parenthesis "[" -> put this opening parenthesis on the stack
  - Step 4: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack

EXAMPLE

- **Example 2:** sequence ( ) [( )])
  - Step 1: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 2: Encounter the next closing parenthesis ")" -> take an opening parenthesis "(" out of the stack
  - Step 3: Encounter the next parenthesis, the opening parenthesis "[" -> put this opening parenthesis on the stack
  - Step 4: The next parenthesis is the opening parenthesis "(" -> put this opening parenthesis on the stack
  - Step 5: Encounter the next parenthesis, the closing parenthesis "}" -> take out an opening parenthesis, the parenthesis "(":
    - Match "(" "}" -> the matching result is wrong, so we conclude that the given parentheses are not symmetric.



IMPLEMENTATION

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct Node{
        char c;
        struct Node* next;
}Node;
Node* top;
char s[1000001];
Node* makeNode(char c){
        Node* p=(Node*)malloc(sizeof(Node));
        p->c = c; p->next = NULL; return p;
}
```


IMPLEMENTATION

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct Node{
        char c;
        struct Node* next;
}Node;
Node* top;
char s[1000001];
Node* makeNode(char c){
        Node* p=(Node*)malloc(sizeof(Node));
        p->c = c; p->next = NULL; return p;
}
```

```
void push(char c){
        Node* p = makeNode(c);
        p->next = top; top = p;
}
char pop(){
        if(top == NULL) return ' ';
        Node* tmp = top; top = top->next;
        char res = tmp->c;
        free(tmp);
        return res;
}
```


IMPLEMENTATION

```
int match(char a, int b){
  if(a == '(' && b == ')') return 1;
  if(a == '{' && b == '}') return 1;
  if(a == '[' && b == ']') return 1;
  return 0;
}
```


IMPLEMENTATION

```
int match(char a, int b){
  if(a == '(' && b == ')') return 1;
  if(a == '{' && b == '}') return 1;
  if(a == '[' && b == ']') return 1;
  return 0;
}
```

```
int check(char* s){
  for(int i = 0; i < strlen(s); i++){
    if(s[i] == '(' || s[i] == '{' || s[i] == '['
      push(s[i]);
    else{
      if(top==NULL) return 0;
      char o = pop();
      if(!match(o,s[i])) return 0;
    }
  }
  return top == NULL;
}
```

IMPLEMENTATION

```
int match(char a, int b){
  if(a == '(' && b == ')') return 1;
  if(a == '{' && b == '}') return 1;
  if(a == '[' && b == ']') return 1;
  return 0;
}
```

```
int main(){
  scanf("%s",s);
  int res = check(s);
  printf("%d",res);
  return 0;
}
```

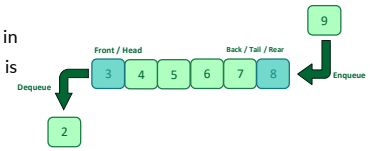
```
int check(char* s){
  for(int i = 0; i < strlen(s); i++){
    if(s[i] == '(' || s[i] == '{' || s[i] == '['
      push(s[i]);
    else{
      if(top==NULL) return 0;
      char o = pop();
      if(!match(o,s[i])) return 0;
    }
  }
  return top == NULL;
}
```


CONTENT

- Stack
- Exercise: Check the symmetry of the parentheses
- **Queue**
- Exercise: Find the fastest way out of the maze

QUEUE

- A queue is a linear data structure with two ends, head and tail
- The operation of adding new elements is performed in the tail. The operation of removing elements is performed in the head
- Operating principle: FIFO – First In First Out
- Basic operations on Q queue:
  - enqueue(x, Q): insert element x into the queue
  - dequeue(Q): remove an element from the queue
  - isEmpty(Q): returns true if the queue is empty



Idea of a queue

Source: https://www.geeksforgeeks.org/queue-data-structure/

ILLUSTRATION THE OPERATIONS OF A QUEUE

Operation	Queue state	Returned element
Create an empty queue	Empty	-
Enqueue 1	1	-
Enqueue 2	1, 2	-
Enqueue 3	1, 2, 3	-
Dequeue	2, 3	1
Enqueue 4	2, 3, 4	-
Enqueue 5	2, 3, 4, 5	-
Dequeue	3, 4, 5	2
Dequeue	4, 5	3
Check if empty	4, 5	0

IMPLEMENT A QUEUE WITH A SINGLY LINKED LIST

- Data structure and Initialization

```
typedef struct Node{
    char value;
    struct Node* next;
}Node;

Node*makeNode(char v){
    Node* p = (Node*)malloc(sizeof(Node));
    p->value = v;
    p->next = NULL;
    return p;
}
```

Declare a struct for each element of a queue

Create a node (element) for the queue

IMPLEMENT A QUEUE WITH A SINGLY LINKED LIST

- Enqueue

```
Node* enqueue(Node * head, char v){
    Node * new_node = makeNode(v);
    if(head == NULL) return new_node;
    else{
        new_node->next = head;
        head = new_node;
        return head;
    }
}
```

Create a new node

If the queue is empty, return the newly created element

If the queue is not empty, make the newly created node the head of the list

IMPLEMENT A QUEUE WITH A SINGLY LINKED LIST

- Dequeue

```
char dequeue(Node** head){
    if(*head == NULL) return NULL;
    Node* p, *q;
    p = *head;
    char ans;

    if(p->next == NULL){
        ans = p->value;
        free(p);
        *head = NULL;
        return ans;
    }

    while(p->next->next != NULL){
        p = p->next;
    }
    q = p->next;
    p->next = NULL;
    ans = q->value;
    free(q);

    return ans;
}
```

If the queue is empty, return NULL

If the queue has only one element

If the queue has more than one element, get the last element of the list

SOME APPLICATIONS OF QUEUES

- **Breadth First Search (BFS):** BFS is an algorithm for traversing or searching in tree and graph data structures. It uses a queue to discover nodes level by level, making it an important tool for solving graph-related problems.
- **Task scheduling**: Queues are used in operating systems to schedule tasks or processes for execution. Tasks are placed into a queue and the operating system executes them in the order they are added, ensuring fairness in resource distribution.

SOME APPLICATIONS OF QUEUES

- **Web server request handling**: Web servers use queues to manage incoming HTTP requests. Each incoming request is placed into a queue and processed by threads or workflows, allowing the server to process multiple requests at once.
- **Buffering in I/O operations**: Queues are used to buffer data during I/O operations. For example, when data is read from a file or network drive, it is often placed in a queue before processing to smooth out variations in the rate at which the data is received.

SOME APPLICATIONS OF QUEUES

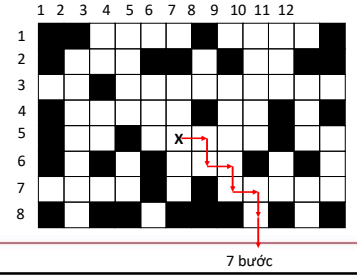
- **Task management in multithreading**: In multithreaded applications, queues can be used to manage tasks that need to be executed concurrently. Execution threads remove tasks from the queue and execute the corresponding work.
- **Order fulfillment in warehouses**: In logistics and storage management, queues can be used to manage the order fulfillment process. Orders are placed into a queue for selection, packing and delivery to ensure efficient processing.

CONTENT

- Stack
- Exercise: Check the symmetry of the parentheses
- Queue
- **Exercise: Find the fastest way out of the maze**

MAZE

- **Problem:** A rectangular maze is represented by a 0-1 NxM matrix in which A[i,j] = 1 represents cell (i,j) as a brick wall and A[i,j] = 0 represents cell (i,j) are empty cells and can be moved into. From an empty cell, we can move to 1 of 4 neighboring cells (up, down, left, right) if that cell is empty.
  - Starting from an empty call in the maze, find the shortest way out of the maze



MAZE

- Data:
  - Line 1: write 4 positive integers n, m, r, c in which n and m are respectively the number of rows and columns of matrix A (1 <= n,m <= 999) and r, c are the indexes respectively. Row and column numbers of the starting cell.
  - Line i+1 (i=1,...,n): the ith line of matrix A
- Result:
  - Write the shortest number of steps needed to exit the maze, or write the value -1 if no path can be found to exit the maze..

MAZE

- Data:
  - Line 1: write 4 positive integers n, m, r, c in which n and m are respectively the number of rows and columns of matrix A (1 <= n,m <= 999) and r, c are the indexes respectively. Row and column numbers of the starting cell.
  - Line i+1 (i=1,...,n): the ith line of matrix A
- Result:
  - Write the shortest number of steps needed to exit the maze, or write the value -1 if no path can be found to exit the maze..

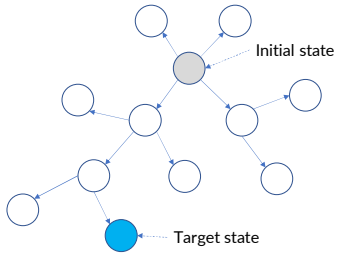
MAZE

Example:

stdin	stdout
8 12 5 6 1 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 1 0 1 1 1 0 1 0 1	7

ALGORITHM

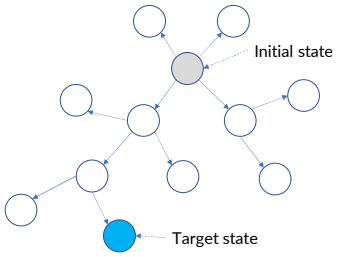
- The breadth-first search algorithm finds the shortest path on the state transition model:
  - Initial state
  - Target state
  - Each state s will have a set of N(s) of neighboring states



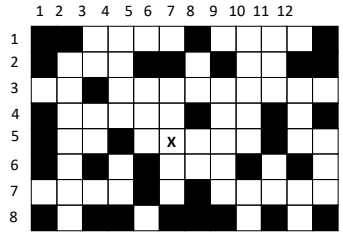
ALGORITHM

- The breadth-first search algorithm finds the shortest path on the state transition model:

```
findPath(s0, N){
  Init Queue
  Queue.PUSH(s0);
  visited[s0] = true;
  while Queue not empty do{
    s = Queue.POP();
    for x ∈ N(s) do{
       if not visited[x] and check(x) then{
            if target(x) then
               return x;
            else{
               Queue.PUSH(x);
               visited[x] = true;
            }
       }
    }
  }
```



ILLUSTRATION

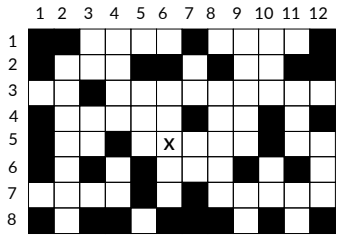


Initialize an empty Q queue

Insert state (5,6) into Q

(5,6)

ILLUSTRATION

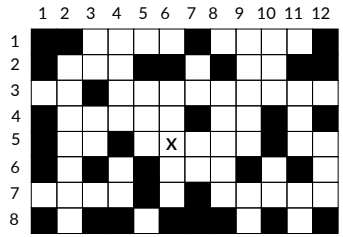


Take (5,6) out of Q

Put the state (5,7), (5, 5), (4, 6), (6,6) into Q

(5,6)	(5,7)	(5,5)	(4,6)	(6,6)

ILLUSTRATION

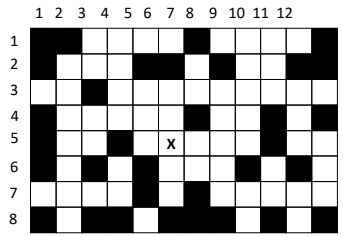


Take (5,7) out of Q

Put states (6,7), (5, 8) into Q

~~(5,7)~~	(5,5)	(4,6)	(6,6)	(6,7)	(5,8)

ILLUSTRATION

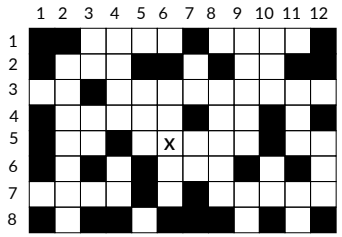


Take (5,5) out of Q

Insert state (4,5) into Q

	(5,5)	(4,6)	(6,6)	(6,7)	(5,8)	(4,5)	

ILLUSTRATION

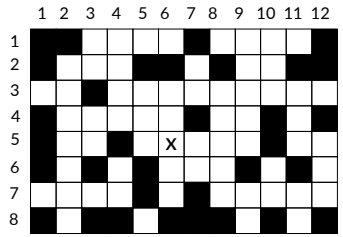


Take (4,6) out of Q

Insert state (3,6) into Q

	(4,6)	(6,6)	(6,7)	(5,8)	(4,5)	(3,6)	

ILLUSTRATION

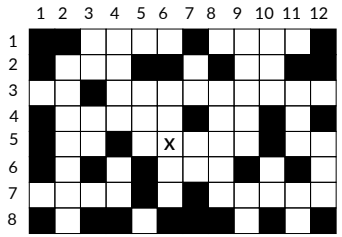


Take (6,6) out of Q

Insert state (7,6) into Q

	(6,6)	(6,7)	(5,8)	(4,5)	(3,6)	(7,6)	

ILLUSTRATION

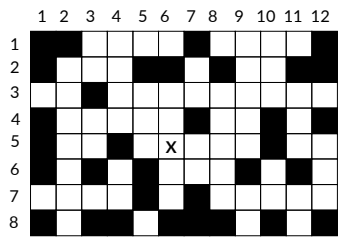


Take (6,7) out of Q

Insert state (6,8) into Q

	(6,7)	(5,8)	(4,5)	(3,6)	(7,6)	(6,8)	

ILLUSTRATION

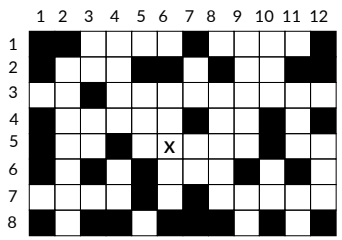


Take (5,8) out of Q

Insert states (5,9) and (4,8) into Q

~~(5,8)~~	(4,5)	(3,6)	(7,6)	(6,8)	(5,9)	(4,8)	

ILLUSTRATION

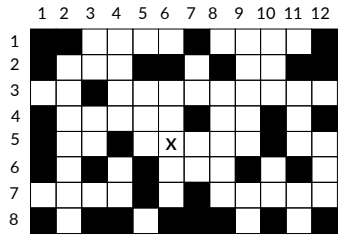


Take (4,5) out of Q

Insert states (4,4) and (3,5) into Q

~~(4,5)~~	(3,6)	(7,6)	(6,8)	(5,9)	(4,8)	(4,4)	(3,5)	

ILLUSTRATION

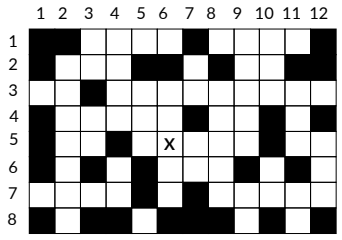


Take (3,6) out of Q

Insert state (3,7) into Q

~~(3,6)~~	(7,6)	(6,8)	(5,9)	(4,8)	(4,4)	(3,5)	(3,7)	

ILLUSTRATION

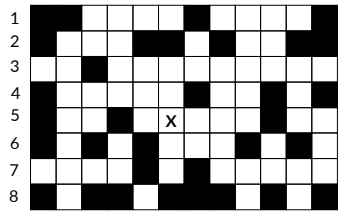


Take (7,6) out of Q

No new states can be added to Q

~~(7,6)~~	(6,8)	(5,9)	(4,8)	(4,4)	(3,5)	(3,7)	

ILLUSTRATION

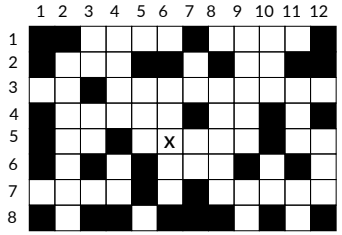


Take (6,8) out of Q

Bring the state (7,8) into Q

	(6,8)	(5,9)	(4,8)	(4,4)	(3,5)	(3,7)	(7,8)	

ILLUSTRATION

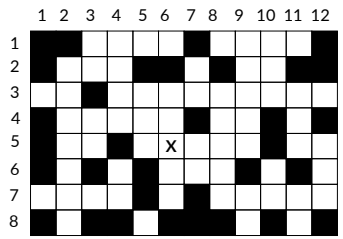


Take (5,9) out of Q

Bring the state (4,9) into Q

	(5,9)	(4,8)	(4,4)	(3,5)	(3,7)	(7,8)	(4,9)	

ILLUSTRATION

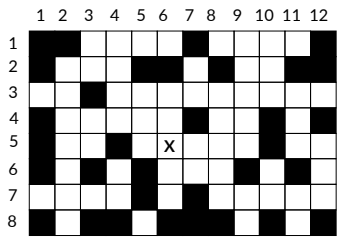


Take (4,8) out of Q

Bring the state (3,8) into Q

	(4,8)	(4,4)	(3,5)	(3,7)	(7,8)	(4,9)	(3,8)	

ILLUSTRATION

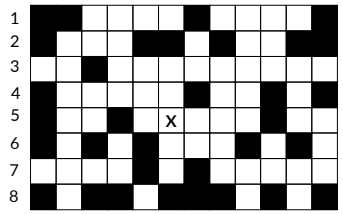


Take (4,4) out of Q

Put the state (4,3), (3,4) into Q

	(4,4)	(3,5)	(3,7)	(7,8)	(4,9)	(3,8)	(4,3)	(3,4)	

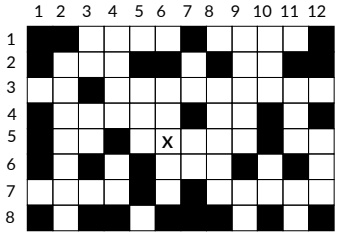
ILLUSTRATION



Take (3,5) out of Q
No new states can be added to Q

	(3,5)	(3,7)	(7,8)	(4,9)	(3,8)	(4,3)	(3,4)	

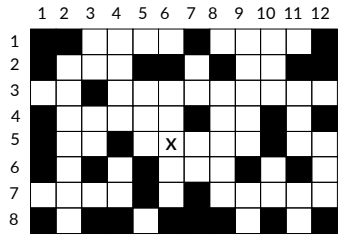
ILLUSTRATION



Take (3,7) out of Q
Bring the new state (2,7) into Q

	(3,7)	(7,8)	(4,9)	(3,8)	(4,3)	(3,4)	(2,7)	

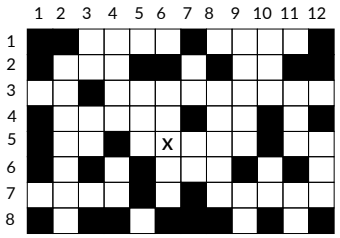
ILLUSTRATION



Take (7,8) out of Q
Bring new status (7,9) into Q

	(7,8)	(4,9)	(3,8)	(4,3)	(3,4)	(2,7)	(7,9)	

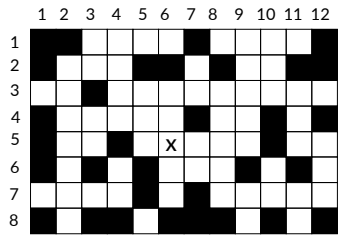
ILLUSTRATION



Take (4,9) out of Q
Bring the new state (3,9) into Q

	(4,9)	(3,8)	(4,3)	(3,4)	(2,7)	(7,9)	(3,9)	

ILLUSTRATION

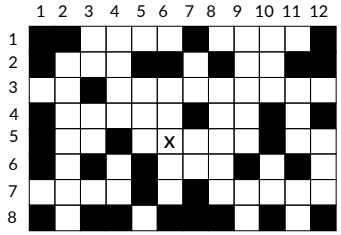


Take (3,8) out of Q

No new states can be added to Q

~~(3,8)~~	(4,3)	(3,4)	(2,7)	(7,9)	(3,9)	

ILLUSTRATION

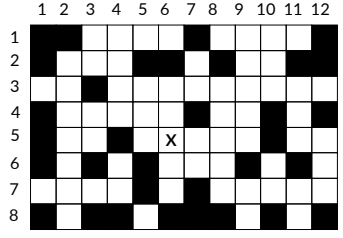


Take (4,3) out of Q

Bringing new states (4,2), (5,3) into Q

~~(4,3)~~	(3,4)	(2,7)	(7,9)	(3,9)	(4,2)	(5,3)	

ILLUSTRATION

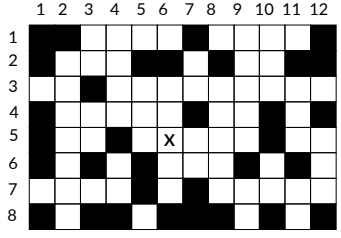


Take (3,4) out of Q

Bring the new state (2,4) into Q

~~(3,4)~~	(2,7)	(7,9)	(3,9)	(4,2)	(5,3)	(2,4)	

ILLUSTRATION

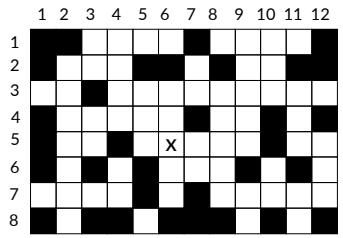


Take (2,7) out of Q

No new states can be added to Q

~~(2,7)~~	(7,9)	(3,9)	(4,2)	(5,3)	(2,4)	

ILLUSTRATION

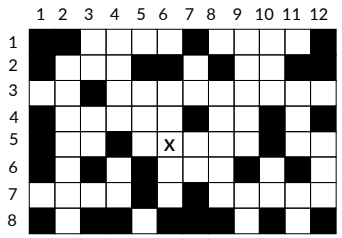


Take (7,9) out of Q

Bring new states (8,9), (7,10) into Q

	~~(7,9)~~	(3,9)	(4,2)	(5,3)	(2,4)	(8,9)	(7,10)	

ILLUSTRATION

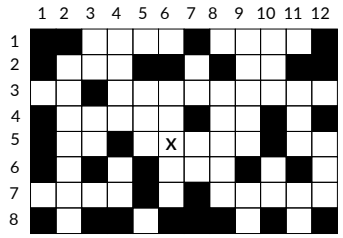


Take (3,9) out of Q

Bring new states (2,9), (3,10) into Q

	~~(3,9)~~	(4,2)	(5,3)	(2,4)	(8,9)	(7,10)	(2,9)	(3,10)	

ILLUSTRATION

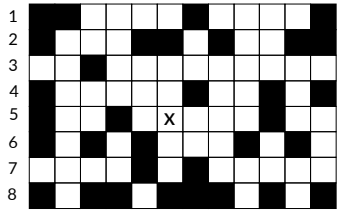


Take (4,2) out of Q

Bring new states (3,2), (5,2) into Q

	~~(4,2)~~	(5,3)	(2,4)	(8,9)	(7,10)	(2,9)	(3,10)	(3,2)	(5,2)	

ILLUSTRATION

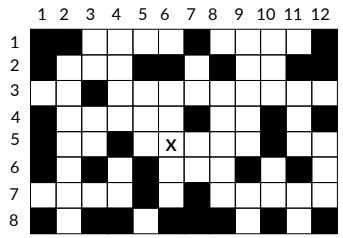


Take (5,3) out of Q

No new states can be added to Q

	~~(5,3)~~	(2,4)	(8,9)	(7,10)	(2,9)	(3,10)	(3,2)	(5,2)	

ILLUSTRATION

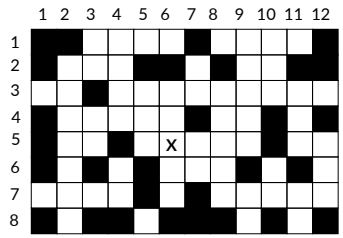


Take (2,4) out of Q

Bring the new state (1,4) into Q

	~~(2,4)~~	(8,9)	(7,10)	(2,9)	(3,10)	(3,2)	(5,2)	(1,4)	

ILLUSTRATION



Take (8,9) out of Q

The target state (9,9) is generated corresponding to the position outside the maze

	~~(8,9)~~	(7,10)	(2,9)	(3,10)	(3,2)	(5,2)	(1,4)	

IMPLEMENTATION

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Node{
	int row;
	int col;
	int step;
	struct Node* next;
}Node;
```


IMPLEMENTATION

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Node{
	int row;
	int col;
	int step;
	struct Node* next;
}Node;
```

```
int n,m;
int A[1000][1000];
int startRow, startCol;
int visited[1000][1000];
Node* first;
Node* last;
int dr[4] = {0,0,1,-1};
int dc[4] = {1,-1,0,0};
```

IMPLEMENTATION

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Node{
        int row;
        int col;
        int step;
        struct Node* next;
}Node;
```

```
int n,m;
int A[1000][1000];
int startRow, startCol;
int visited[1000][1000];
Node* first;
Node* last;
int dr[4] = {0,0,1,-1};
int dc[4] = {1,-1,0,0};
```

```
Node* makeNode(int r,int c, int step){
  Node* p = (Node*)malloc(sizeof(Node));
  p->row = r; p->col = c;
  p->step = step; p->next = NULL;
  return p;
}
```


IMPLEMENTATION

```
int isEmpty(){
  return first == NULL && last == NULL;
}
void push(Node* p){
  if(isEmpty(){
    first = p; last = p;  return;}
  last->next = p; last = p;
}
Node* pop(){
  if(isEmpty()) return NULL;
  Node* tmp = first; first = first->next;
  if(first == NULL) last = NULL;
  return tmp;
}
```


IMPLEMENTATION

```
int isEmpty(){
  return first == NULL && last == NULL;
}
void push(Node* p){
  if(isEmpty(){
    first = p; last = p;  return;}
  last->next = p; last = p;
}
Node* pop(){
  if(isEmpty()) return NULL;
  Node* tmp = first; first = first->next;
  if(first == NULL) last = NULL;
  return tmp;
}
```

```
void input(){
  scanf("%d %d %d %d",&n,&m,&sr,&sc);
  for(int i = 1; i <= n; i++)
   for(int j = 1; j <= m; j++)
     scanf("%d",&A[i][j]);
}
```


IMPLEMENTATION

```
int isEmpty(){
  return first == NULL && last == NULL;
}
void push(Node* p){
  if(isEmpty(){
    first = p; last = p;  return;}
  last->next = p; last = p;
}
Node* pop(){
  if(isEmpty()) return NULL;
  Node* tmp = first; first = first->next;
  if(first == NULL) last = NULL;
  return tmp;
}
```

```
void input(){
  scanf("%d %d %d %d",&n,&m,&sr,&sc);
  for(int i = 1; i <= n; i++)
   for(int j = 1; j <= m; j++)
     scanf("%d",&A[i][j]);
}
```

```
int targetState(Node* s){
  return (s->row < 1 ||
    s->row > n || s->col < 1
      || s->col > m);
}
```

IMPLEMENTATION

```
void init(){
  first = NULL; last = NULL;
  for(int i = 0; i <= 1000; i++)
   for(int j = 0; j <= 1000; j++)
    visited[i][j] = 0;
  Node* startState = makeNode(sr,sc,0);
  push(startState); visited[sr][sc] = 1;
}
```


IMPLEMENTATION

```
void init(){
  first = NULL; last = NULL;
  for(int i = 0; i <= 1000; i++)
   for(int j = 0; j <= 1000; j++)
    visited[i][j] = 0;
  Node* startState = makeNode(sr,sc,0);
  push(startState); visited[sr][sc] = 1;
}
```

```
int solve(){
 init();
 while(!isEmpty()){
  Node* s = pop();
  for(int k = 0; k < 4; k++){
  int nr = s->row + dr[k];int nc= s->col + dc[k];
    if(visited[nr][nc]==0&& A[nr][nc]==0){
      Node* ns = makeNode(nr, nc, s->step + 1);
      push(ns);visited[nr][nc] = 1;
      if(targetState(ns)) return ns->step;
    }
   }
  }
  return -1;
}
```


IMPLEMENTATION

```
void init(){
  first = NULL; last = NULL;
  for(int i = 0; i <= 1000; i++)
   for(int j = 0; j <= 1000; j++)
    visited[i][j] = 0;
  Node* startState = makeNode(sr,sc,0);
  push(startState); visited[sr][sc] = 1;
}
```

```
int main(){
  input();
  int res = solve();
  printf("%d",res);
  return 0;
}
```

```
int solve(){
 init();
 while(!isEmpty()){
  Node* s = pop();
  for(int k = 0; k < 4; k++){
  int nr = s->row + dr[k];int nc= s->col + dc[k];
    if(visited[nr][nc]==0&& A[nr][nc]==0){
      Node* ns = makeNode(nr, nc, s->step + 1);
      push(ns);visited[nr][nc] = 1;
      if(targetState(ns)) return ns->step;
    }
   }
  }
  return -1;
}
```


HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !