Data Structures: Stacks and Queues

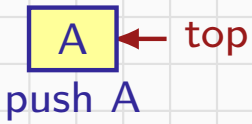
Wei-Mei Chen

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

Stacks

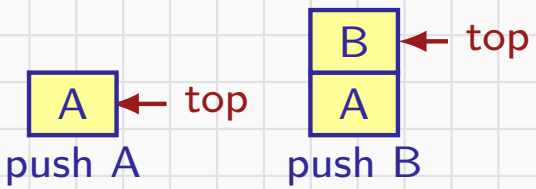
Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



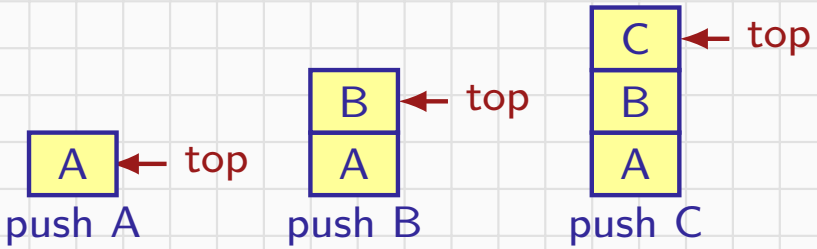
Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



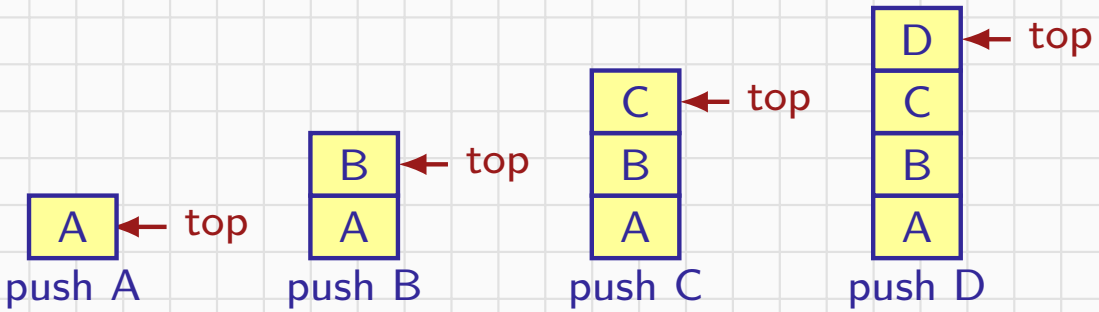
Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



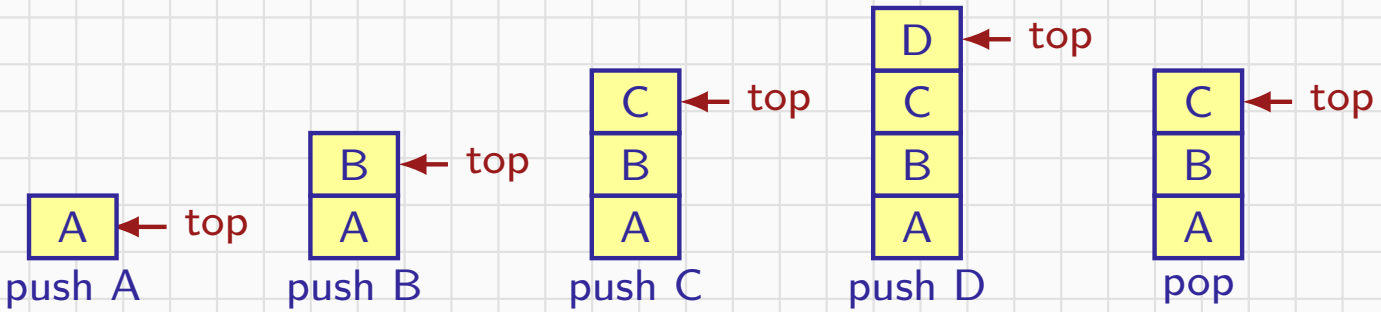
Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



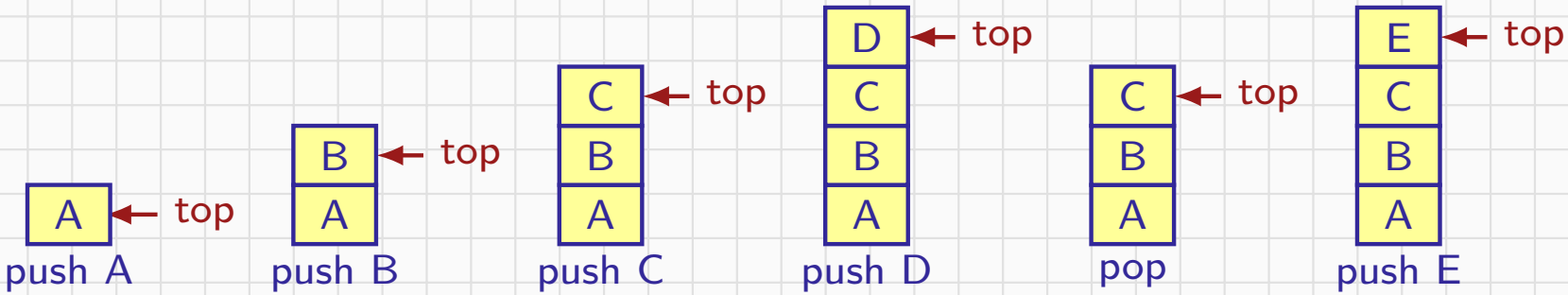
Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



Stacks

1. A stack is an ordered list in which insertions and deletions are made at one end called the top.
2. A stack is also known as a Last-In-First-Out (LIFO) list.



System Stack

- Stack frame of function call

  Whenever a function is invoked, the program creates a structure, referred to as an activation record or a stack frame, and places it on top of the system stack.
- Initially, activation record for the invoked function contains
  - a pointer to the previous stack frame

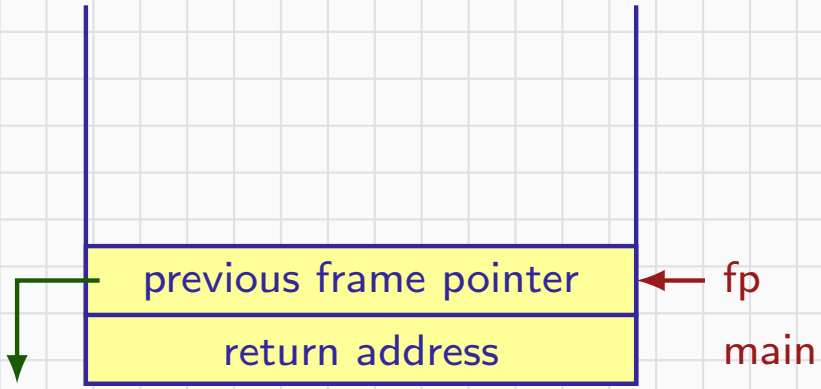
    points to the stack frame of the invoking frame
  - a return address

    contains the location of the statement to be executed after the function terminates
- If this function invokes another function, the local variables and the parameters of the invoking function are added to its stack frame.

  ☞ A new stack frame is created for the invoked function (i. e. top).

System Stack after a Function Call

- System stack after function call **a1**
  fp: a pointer to current stack frame

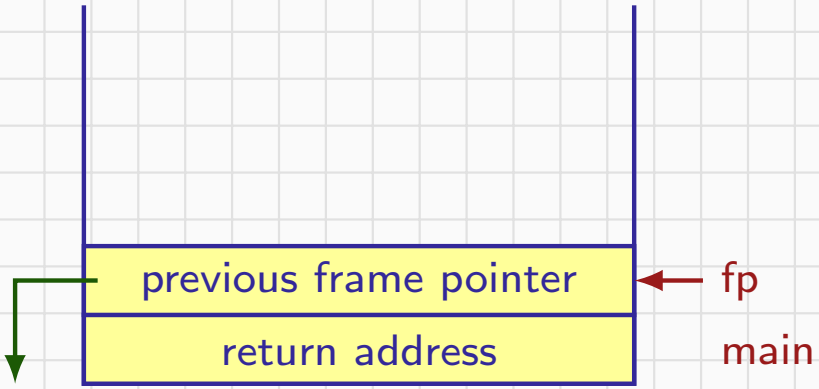


system stack before **a1** is invoked

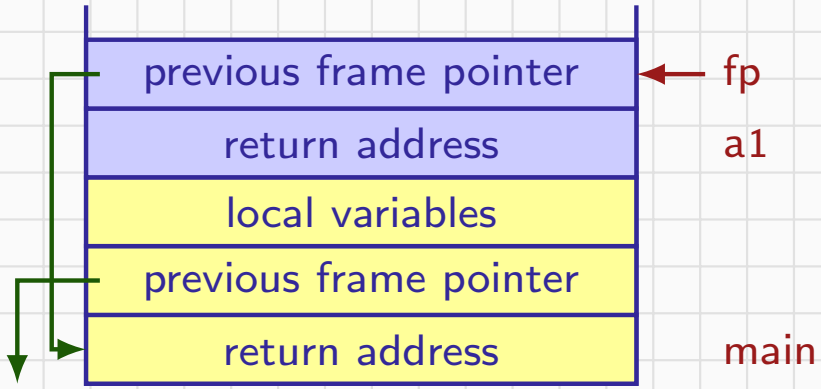
- When this function terminates, its stack frame is removed and processing of the invoking function continues.

System Stack after a Function Call

- System stack after function call **a1**
  fp: a pointer to current stack frame



system stack before **a1** is invoked



system stack after **a1** is invoked

- When this function terminates, its stack frame is removed and processing of the invoking function continues.

ADT Stack

---

structure *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer
    Stack CreateS(*max_stack_size*) ::=
                create an empty stack whose maximum size is *max_stack_size*
    Boolean IsFull(*stack*, *max_stack_size*) ::=
                if (number of elements in *stack* == *max_stack_size*)
                return *TRUE*
                else return *FALSE*
    Stack Add(*stack*, *item*) ::=
                if (IsFull(*stack*)) *stack_full*
                else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
                if (*stack* == CreateS(*max_stack_size*))
                return *TRUE*
                else return *FALSE*
    Element Delete(*stack*) ::=
                if (IsEmpty(*stack*)) **return**
                else remove and return the *item* on the top of the stack.

Structure 3.1: Abstract data type *Stack*

An Array Implementation of a Stack

The easiest way to implement

```
Stack CreateS(maxStackSize) ::=
    #define MAX_STACK_SIZE 100 /* maximum stack size */
        typedef struct element {
            int key;
             /* other fields */
        } element;
        element stack[MAX_STACK_SIZE];
        int top = -1;
    Boolean IsEmpty(Stack) ::= top < 0;
    Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE - 1;
```

☞ If **top** is set to -1 ➠ empty stack

```
void push(element item)
{  if (top >= MAX_STACK_SIZE-1)
       stackFull();
   stack[++top] = item;
}
```

```
element pop()
{  if (top == -1)
       return stackEmpty();
   return stack[top--];
}
```

Stack Using Dynamic Arrays

No MAX_STACK_SIZE limitations ➠ array doubling

```
Stack CreateS(maxStackSize) ::=
        typedef struct {
            int key;
             /* other fields */
        } element;
        element *stack;
        MALLOC(stack,sizeof(*stack));
        capacity = 1;
        int top = -1;
    Boolean IsEmpty(Stack) ::= top < 0;
    Boolean IsFull(Stack) ::= top >= capacity - 1;
```

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

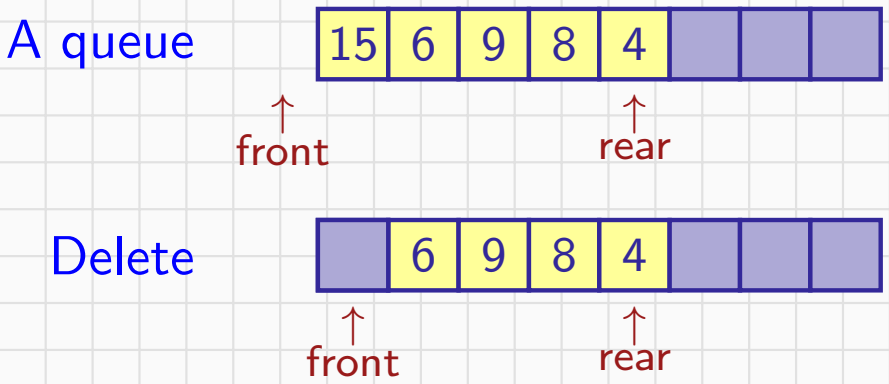
Queues

Queues

- A queue is an ordered list in which all insertion take place one end, called the rear and all deletions take place at the opposite end, called the front.
- It is also known as First-In-First-Out (FIFO) lists.

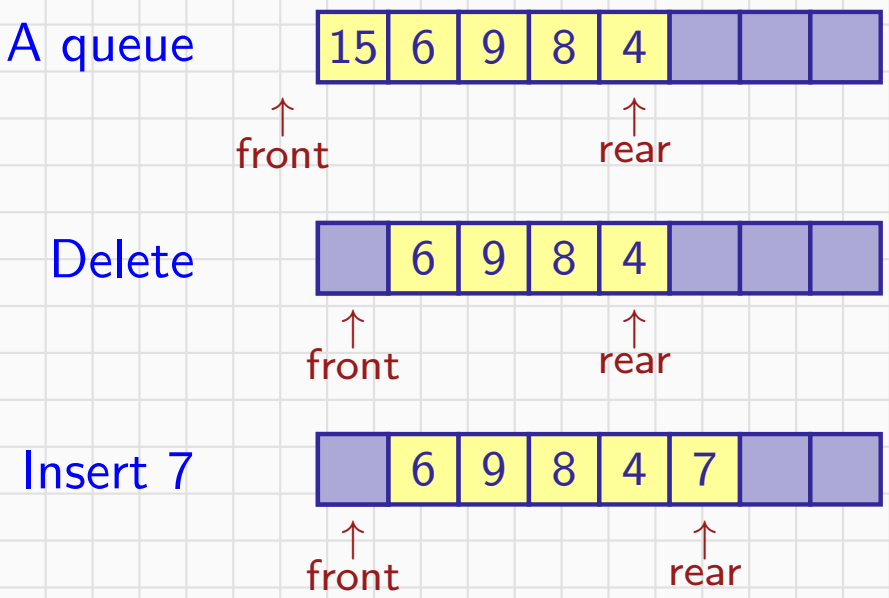
Queues

- A queue is an ordered list in which all insertion take place one end, called the rear and all deletions take place at the opposite end, called the front.
- It is also known as First-In-First-Out (FIFO) lists.



Queues

- A queue is an ordered list in which all insertion take place one end, called the rear and all deletions take place at the opposite end, called the front.
- It is also known as First-In-First-Out (FIFO) lists.



ADT Queue

structure *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *queue* ∈ *Queue*, *item* ∈ *element*, *max_queue_size* ∈ positive integer
    Queue CreateQ(*max_queue_size*) ::=
                create an empty queue whose maximum size is *max_queue_size*
    Boolean IsFullQ(*queue*, *max_queue_size*) ::=
                if (number of elements in *queue* == *max_queue_size*)
                return *TRUE*
                else return *FALSE*
    Queue AddQ(*queue*, *item*) ::=
                if (IsFullQ(*queue*)) *queue_full*
                else insert *item* at rear of *queue* and return *queue*
    Boolean IsEmptyQ(*queue*) ::=
                if (*queue* == CreateQ(*max_queue_size*))
                return *TRUE*
                else return *FALSE*
    Element DeleteQ(*queue*) ::=
                if (IsEmptyQ(*queue*)) **return**
                else remove and return the *item* at front of queue.

Structure 3.2: Abstract data type *Queue*

An Array Implementation of a Queue

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

An Array Implementation of a Queue

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

```
void addq(element item)
{
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull();
    queue[++rear] = item;
}
```

```
element deleteq()
{
    if (front == rear)
        return queueEmpty();
    return queue[++front];
}
```

Job Scheduling

- OS: A sequential representation of queue.

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

- As jobs enter and leave the system, the queue gradually shift to right. In this case, queueFull should move the entire queue to the left so that the first element is again at queue[0], front is at -1, and rear is correctly positioned.
- Shifting an array is very time-consuming, queueFull has a worst case complexity of $O(MAX\_QUEUE\_SIZE)$.

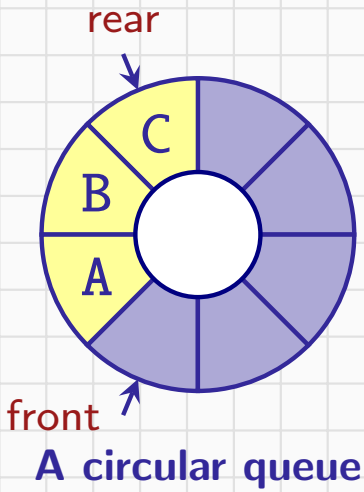
A Circular Queue

- We can obtain a more efficient representation if we regard the array **queue[MAX_QUEUE_SIZE]** as circular.
- an array ➠ a circular queue

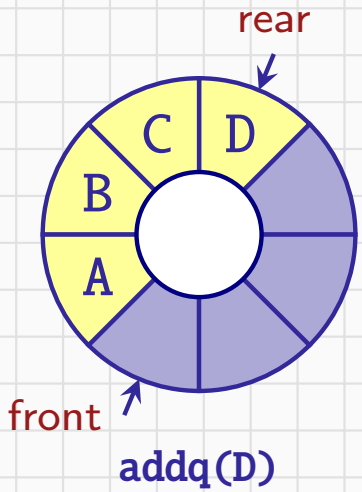
  front: one position counterclockwise from the first element

  rear: current end
- The queue is empty iff front == rear

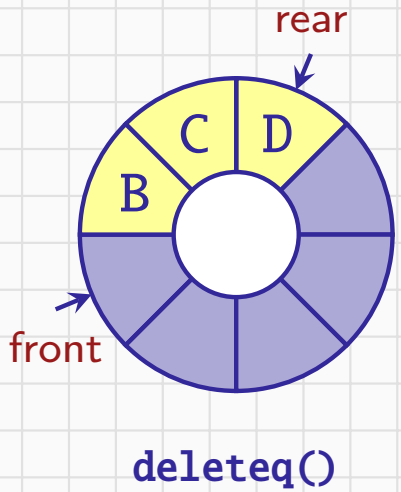
  Problem: one space is left when queue is full.



A circular queue



addq(D)



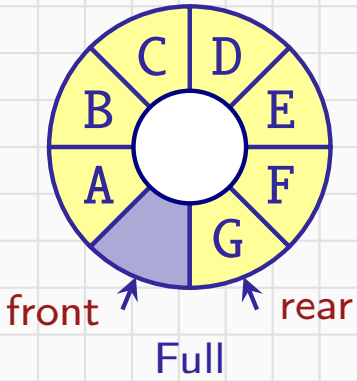
deleteq()

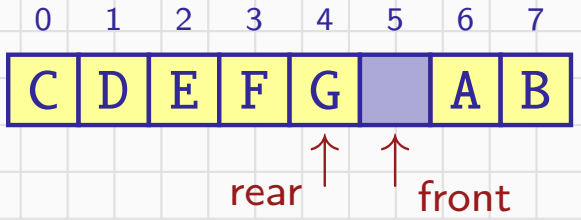
An Array Implementation of a Circular Queue

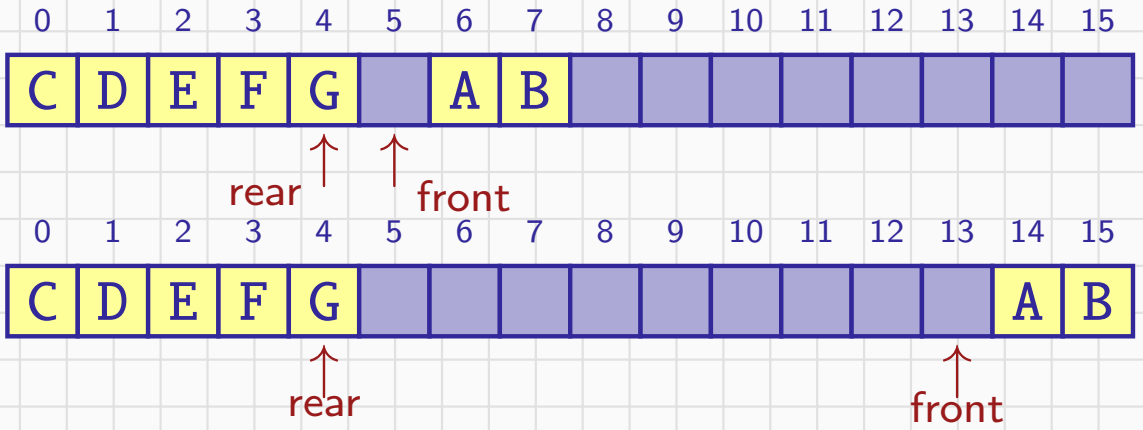
```
void addq(element item)
{
    rear = (rear +1) % MAX_QUEUE_SIZE;
    if (front == rear)
        queueFull(); /* Extend the size:  an exercise */
    queue[rear] = item;
}
```

```
element deleteq()
{
    element item;
    if (front == rear)
        return queueEmpty();
    front = (front+1) % MAX_QUEUE_SIZE
    return queue[front];
}
```

Circular Queues Using Dynamic Arrays

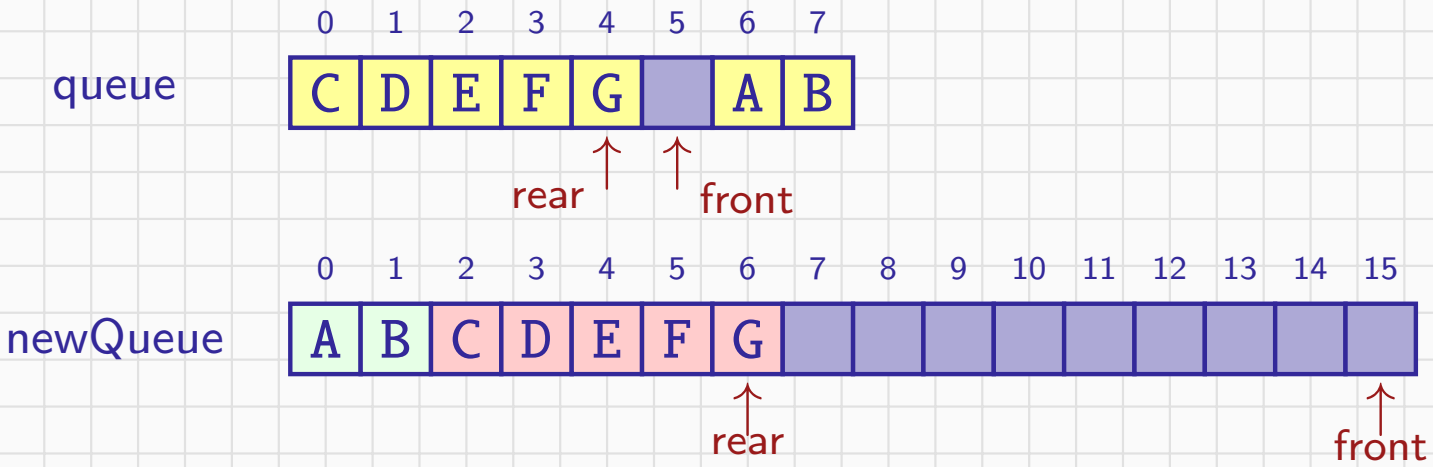






Doubling Queue Capacity

- Create a new array newQueue of twice the capacity
- Copy the second segment (from queue[front+1] to queue[capacity-1]) to positions in newQueue beginning at 0
- Copy the first segment (from queue[0] to queue[rear]) to positions in newQueue beginning at capacity-front-1
- capacity = 8



```
void addq(element item)
{   rear = (rear + 1) % capacity
    if (front == rear )
        queueFull();
    queue[rear] = item;
}
```

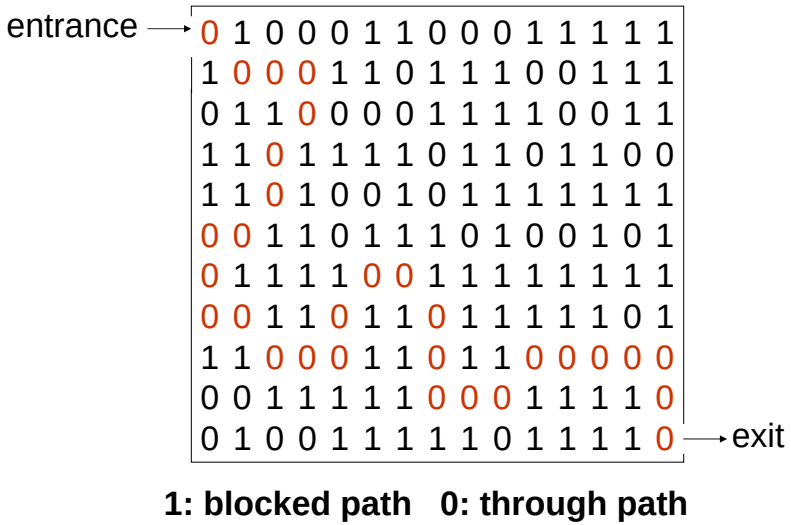
The function copy(a,b,c) copies elements from location a through b-1 to location beginning at c.

```
element queueFull()
{  element* newQueue;
    MALLOC(newQueue, 2 * capacity * sizeof(queue));
    int start = (front + 1) % capacity;
    if (start < 2)
       copy(queue + start, queue + start + capacity - 1, newQueue);
    else
    {   copy(queue + start, queue + capacity, newQueue);
        copy(queue, queue + rear + 1, newQueue + capcity - start);
    }
    front = 2 * capacity - 1;
    rear = capacity - 2;
    capacity *= 2;
    free(queue);
    queue = newQueue;
}
```

Maze

The Maze Problem

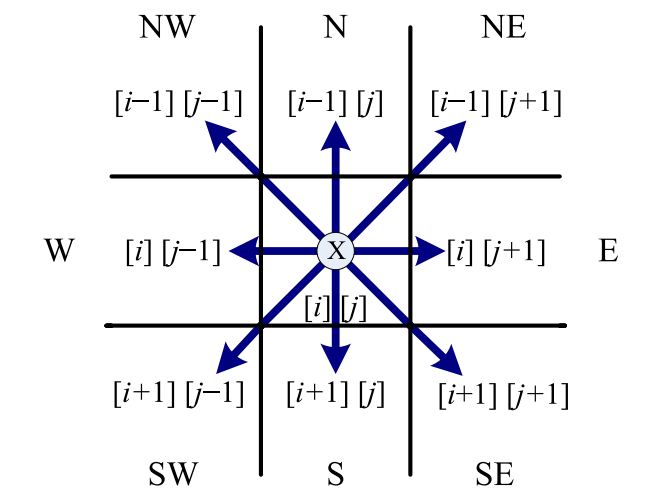
- Representation of the maze
  - The most obvious choice is a two dimensional array
  - 0s: the open paths 1s: the barriers
- Notice that not every position has eight neighbors.
- To avoid checking for these border conditions we can surround the maze by a border of ones. Thus an $m \times p$ maze will require an $(m+2) \times (p+2)$ array
- The entrance: $maze[1][1]$
  The exit: $maze[m][p]$



A Possible Implementation

☞ Directions

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

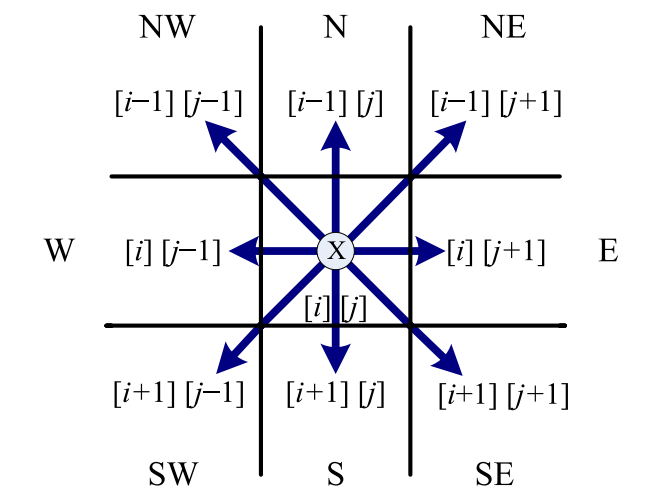



```
typedef struct offsets {
    short int vert;
    short int horiz;
} offsets;
offsets move[8];
```

A Possible Implementation

☞ Directions

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 offsets {
    short int vert;
    short int horiz;
} offsets;
offsets move[8];
```

Current position: *maze*[*row*][*col*]

Next position :

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

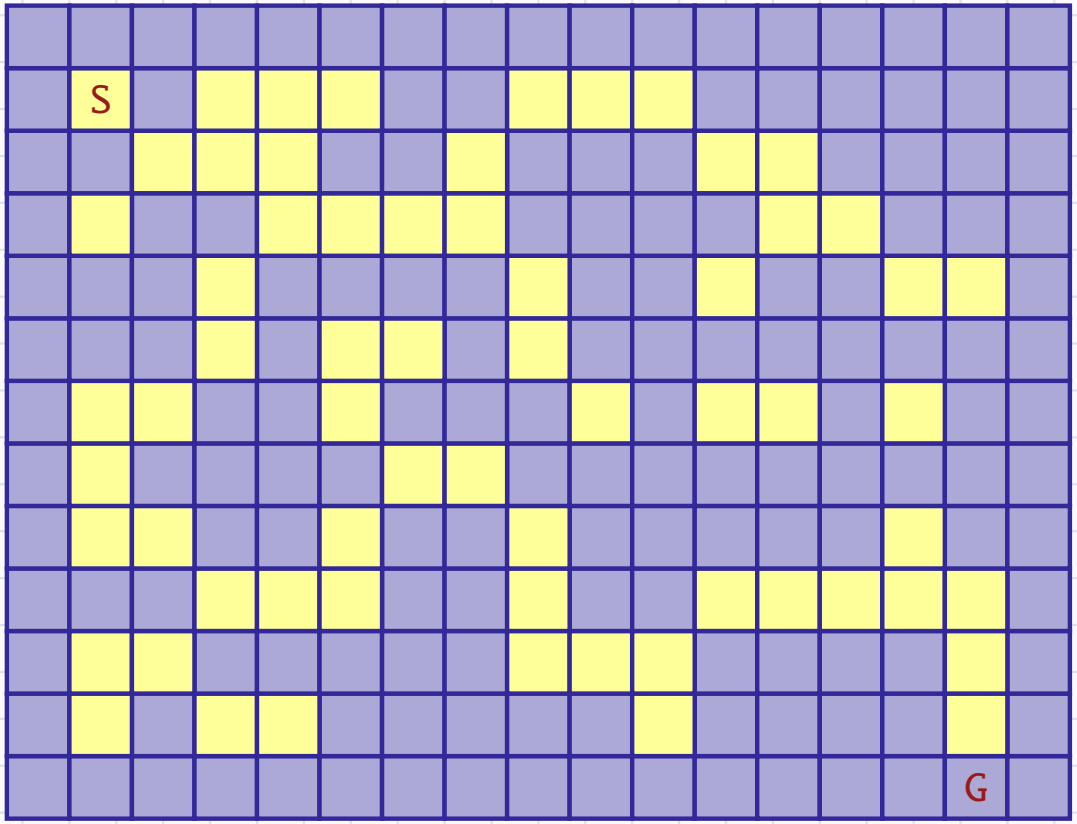
A Maze Traversal Algorithm

☞ A two-dimensional array mark:
to record the maze positions checked

```
Initialize a stack to the maze's entrance coordinates and direction to NE;
while (stack is not empty){
   <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 ) {
         mark[nextRow][nextCol]=1;
         add <row, col, dir> to the top of the stack;
         row = nextRow;
         col = nextCol;
         dir = north;
      }
   }
}
printf("No path found\n");
```

```
/*stack:  to keep history*/
#define MAX_STACK_SIZE 100
typedef struct {
     short int row;
     short int col;
     short int dir;
} element;
element stack[MAX_STACK_SIZE];
```

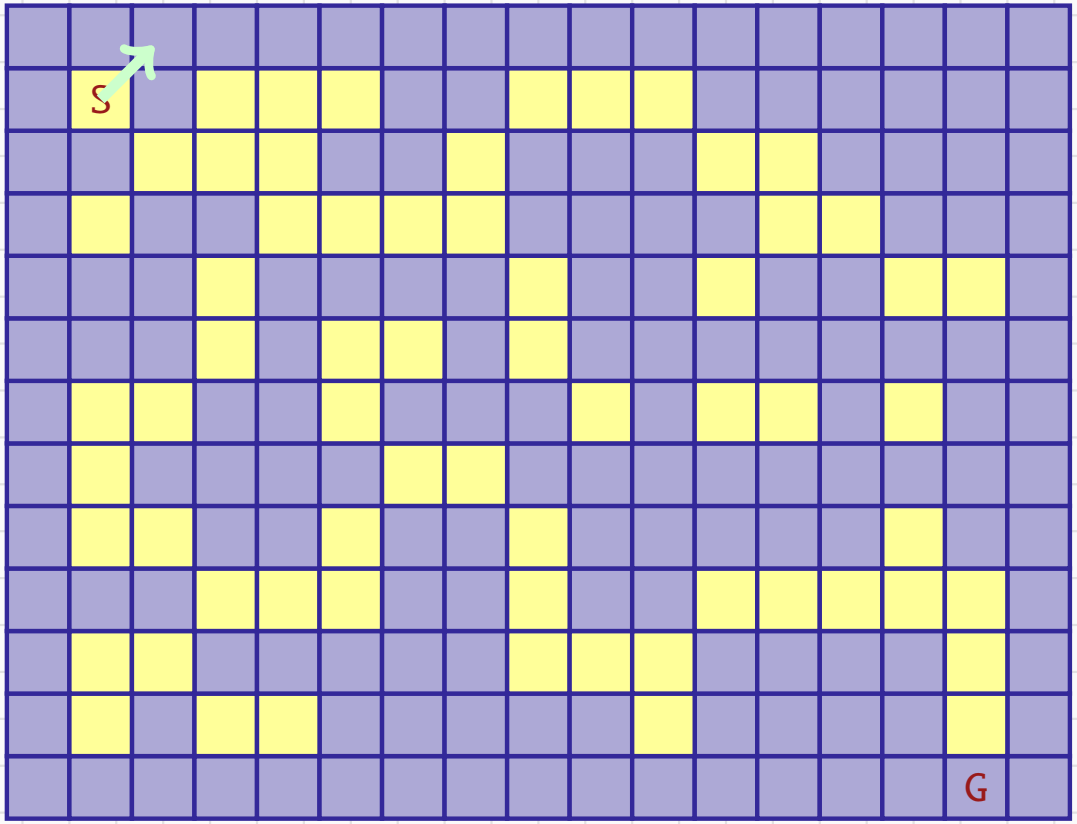
Finding a Path



1	1	1
r	c	d

stack

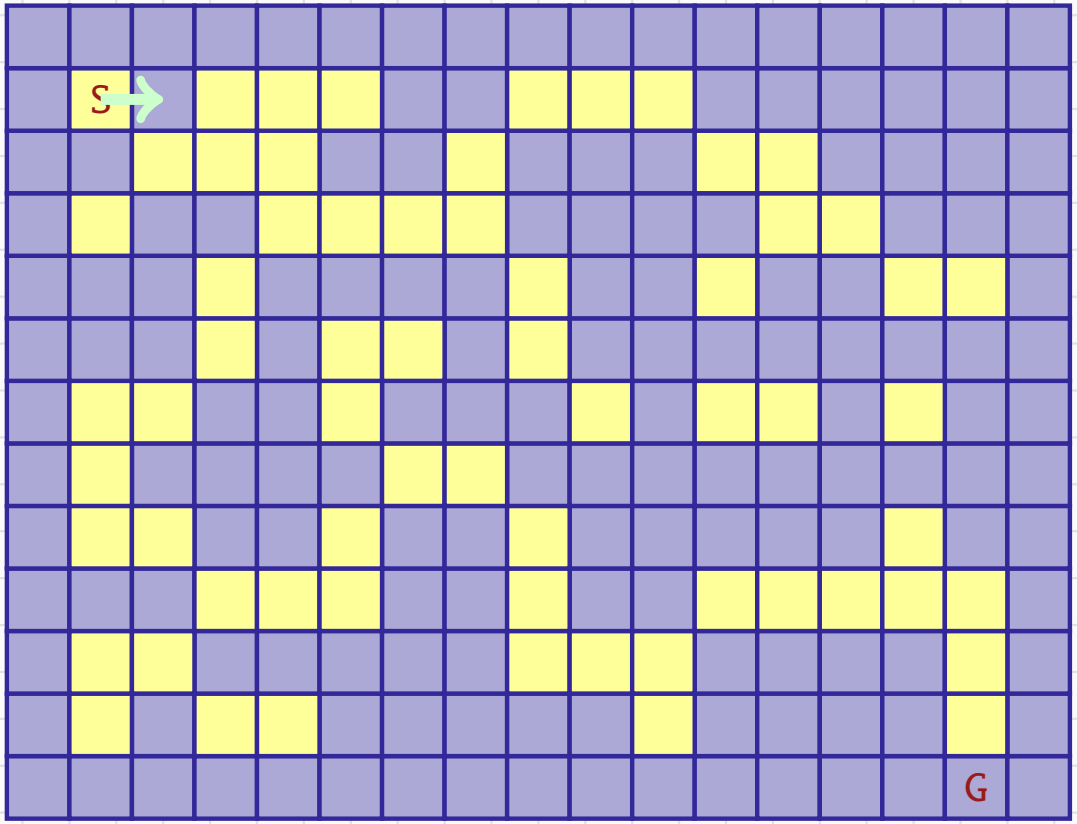
Finding a Path



r c d

stack

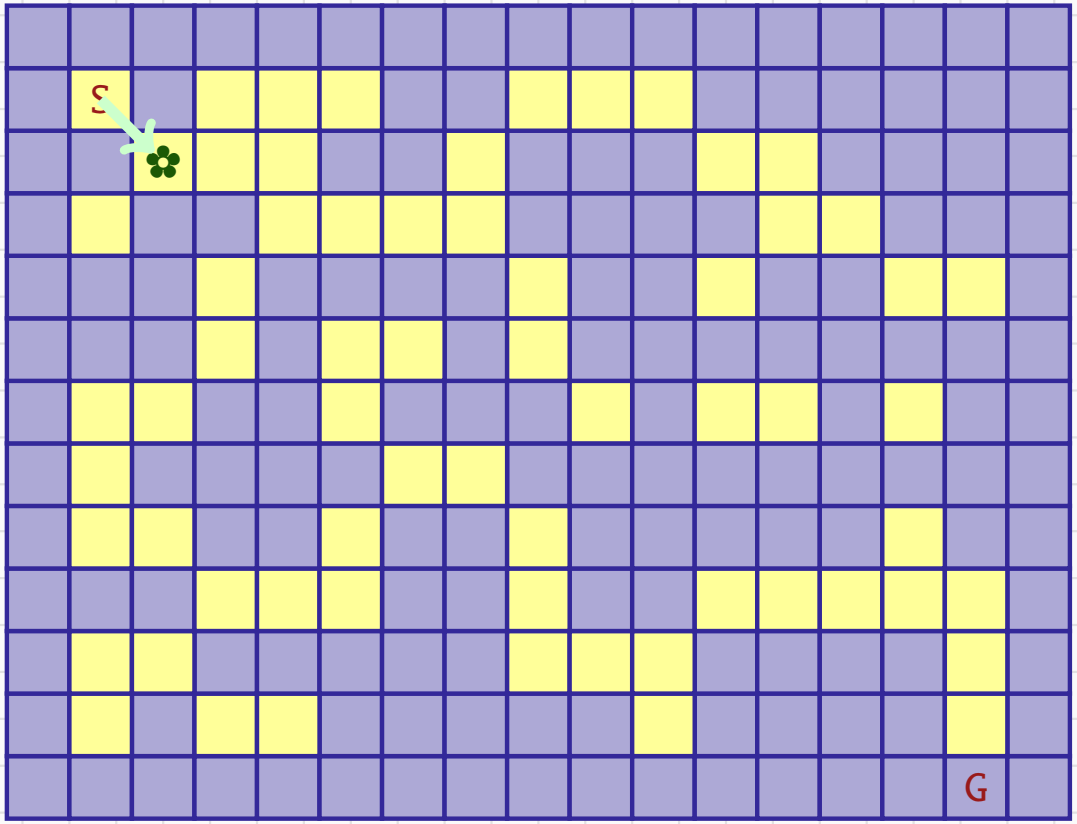
Finding a Path



r c d

stack

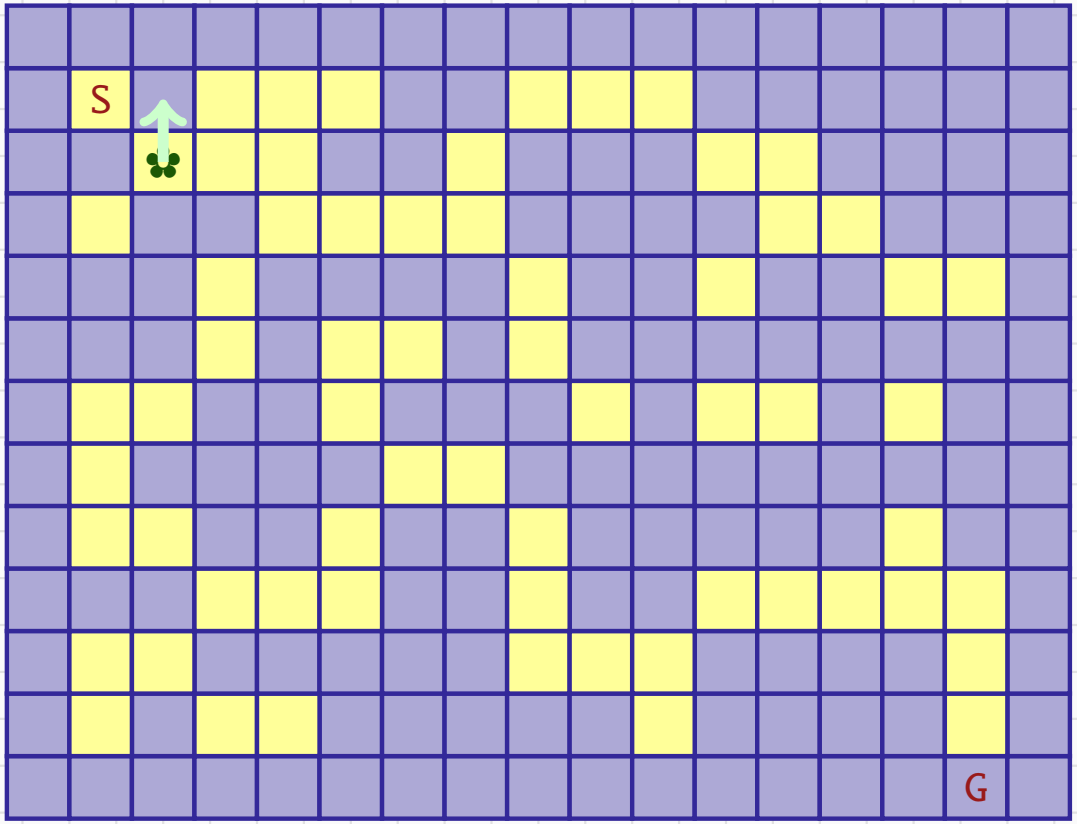
Finding a Path



1	1	4
r	c	d

stack

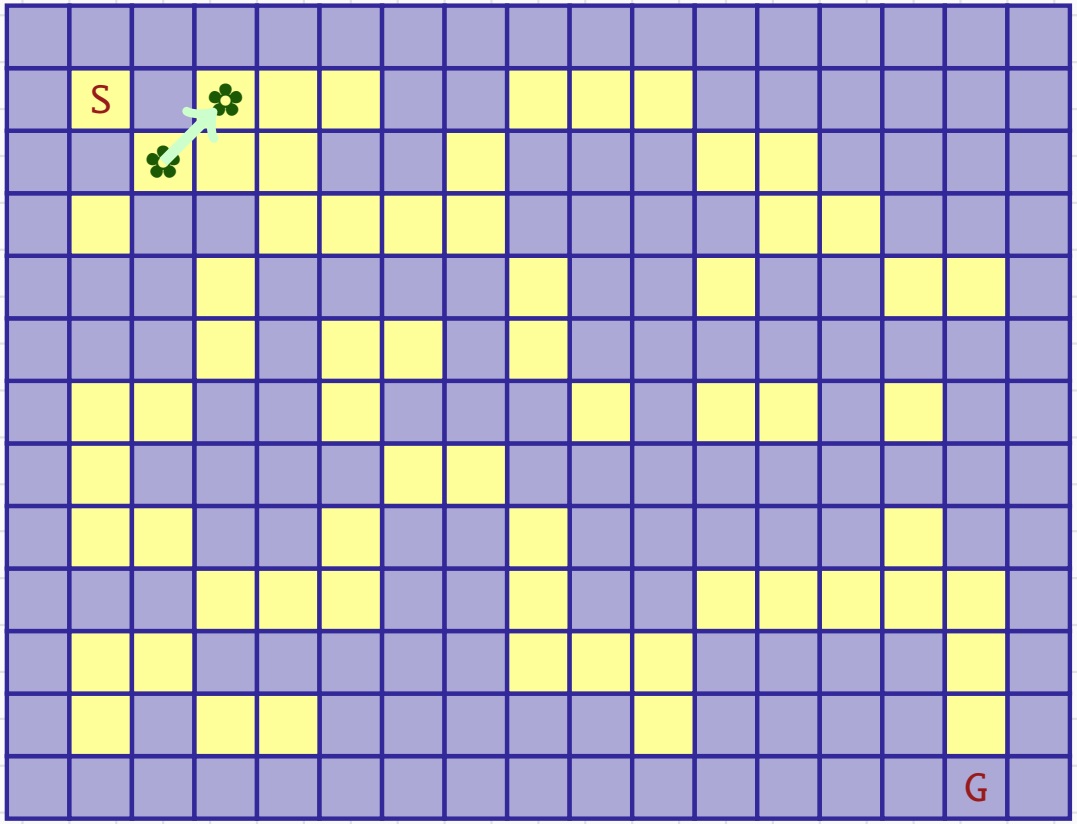
Finding a Path



1	1	4
r	c	d

stack

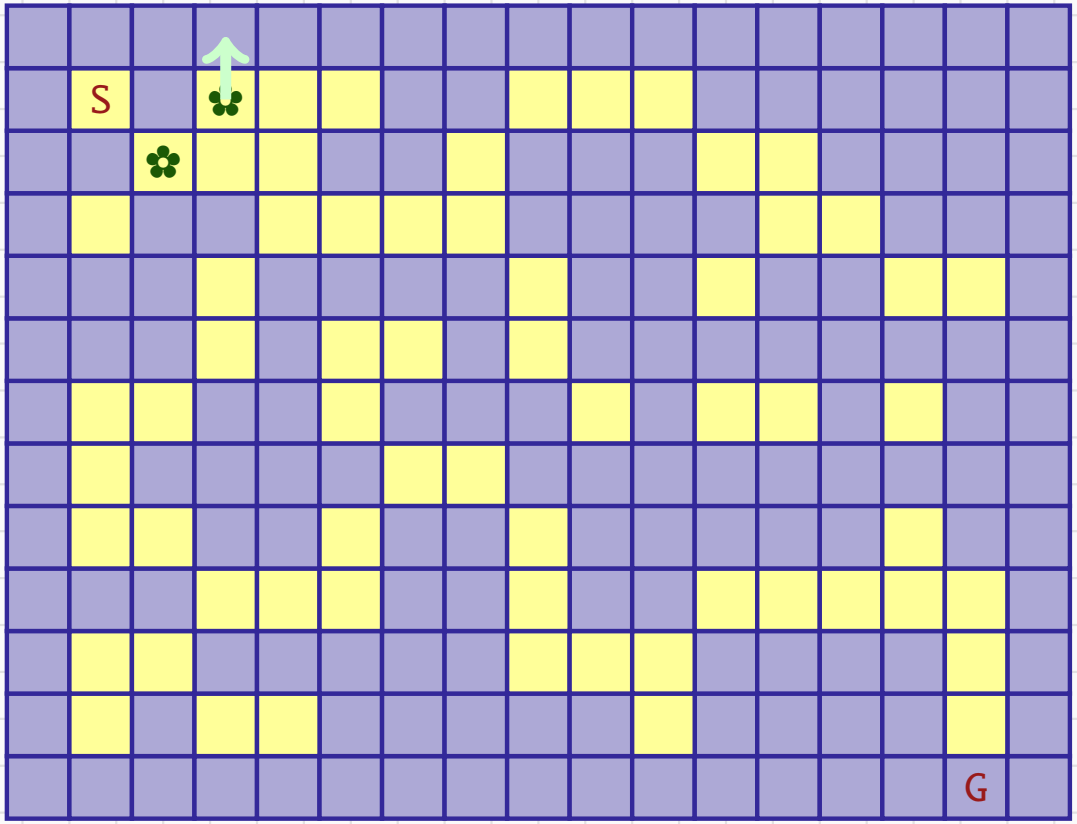
Finding a Path



2	2	2
1	1	4
r	c	d

stack

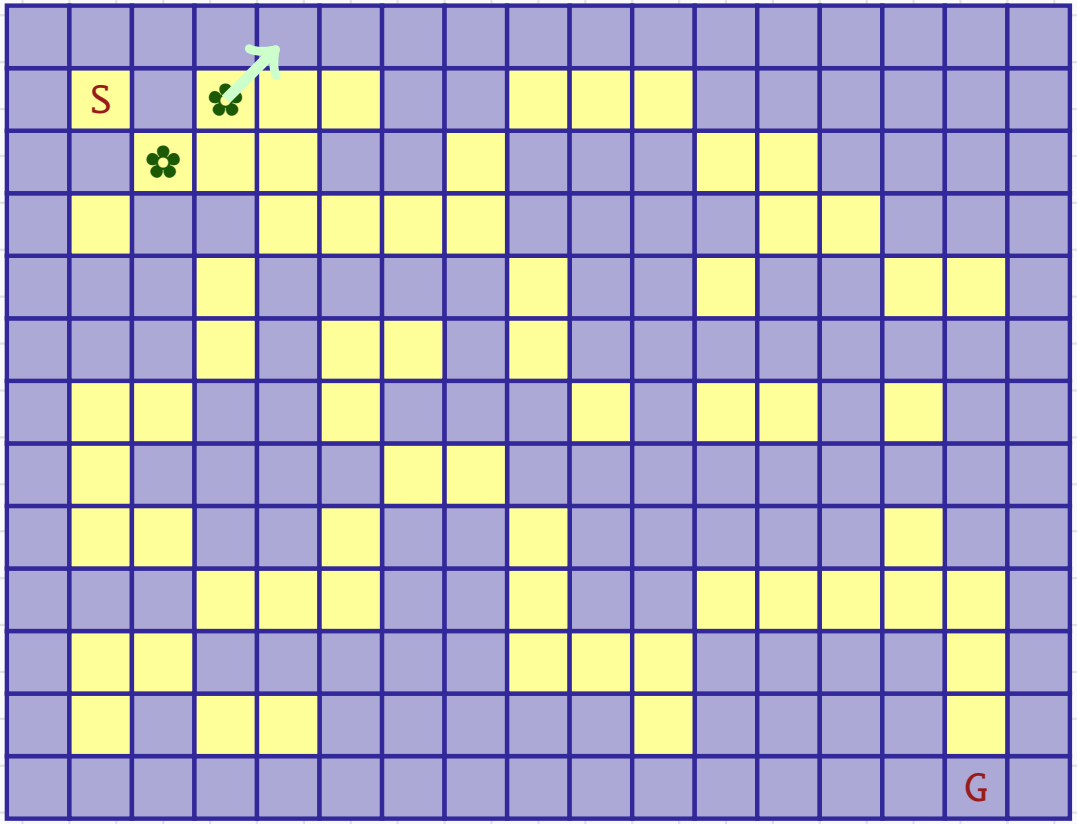
Finding a Path



2	2	2
1	1	4
r	c	d

stack

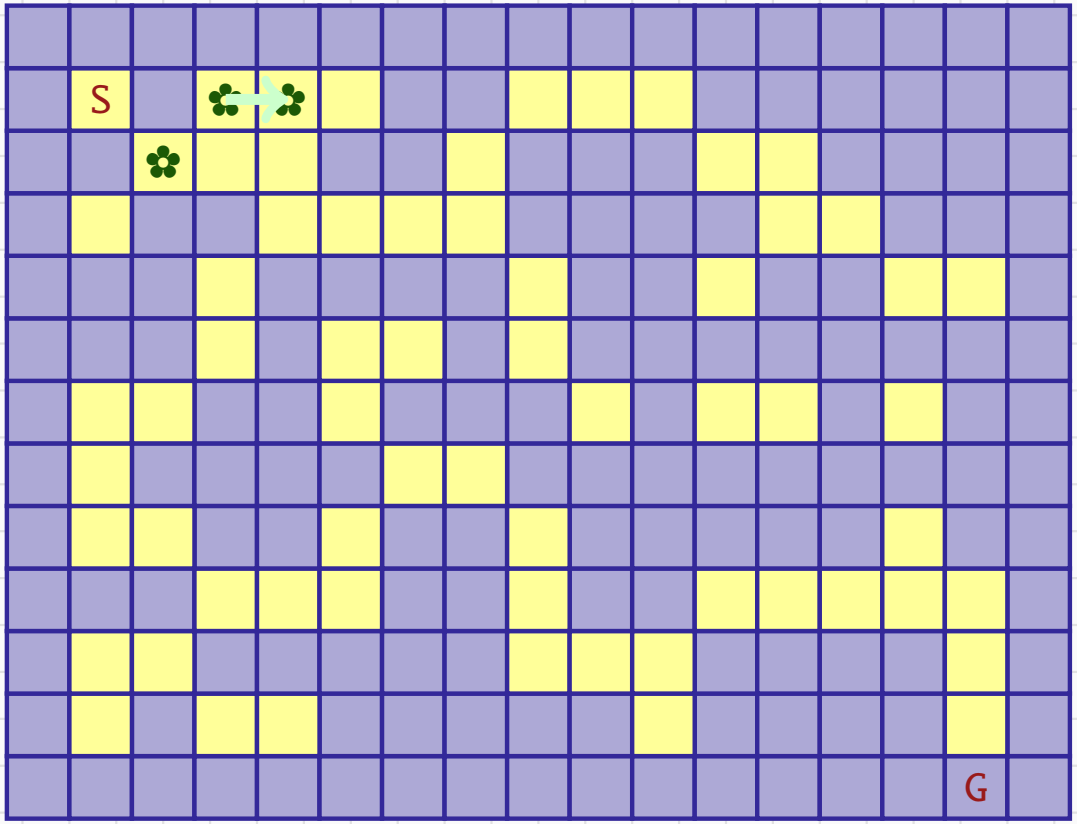
Finding a Path



2	2	2
1	1	4
r	c	d

stack

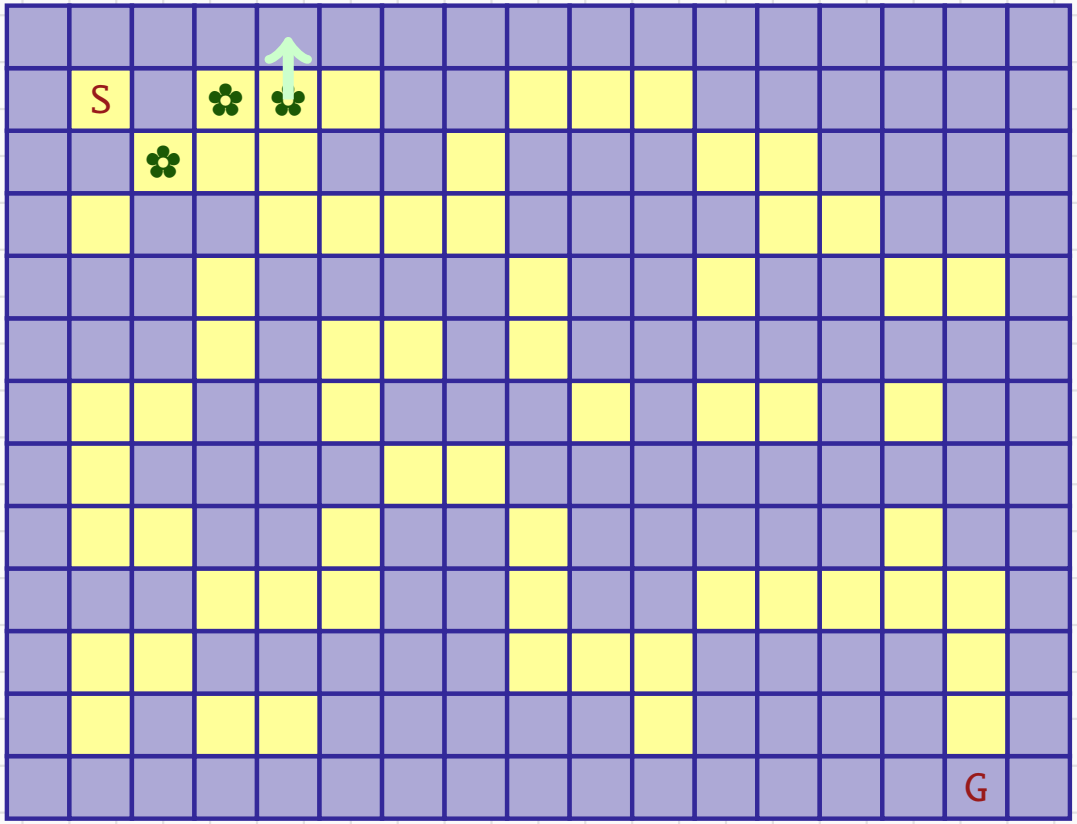
Finding a Path



1	3	3
2	2	2
1	1	4
r	c	d

stack

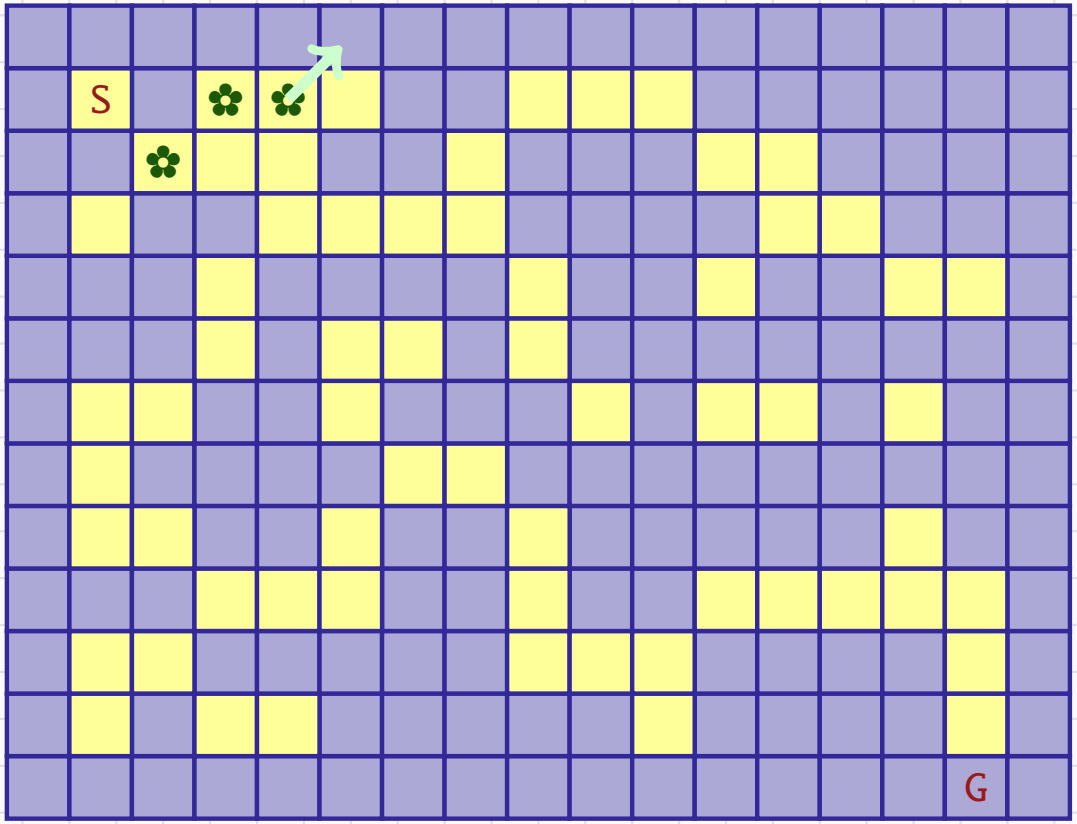
Finding a Path



r	c	d
1	3	3
2	2	2
1	1	4

stack

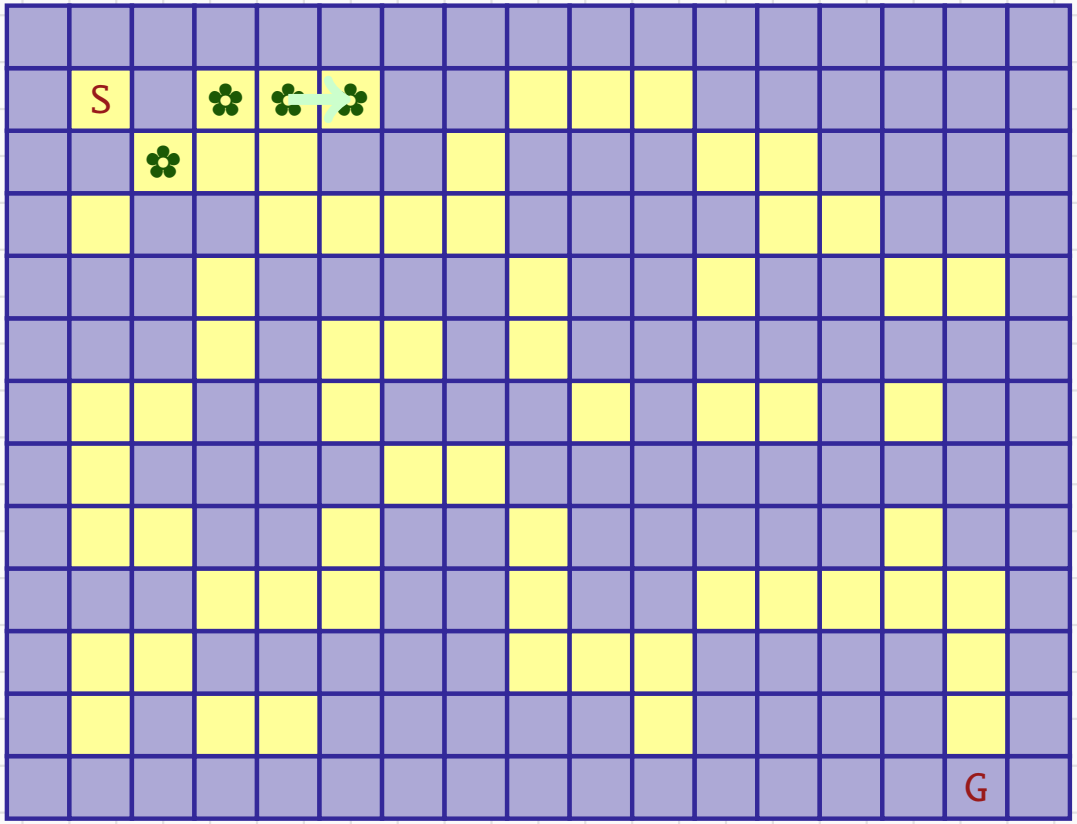
Finding a Path



r	c	d
1	3	3
2	2	2
1	1	4

stack

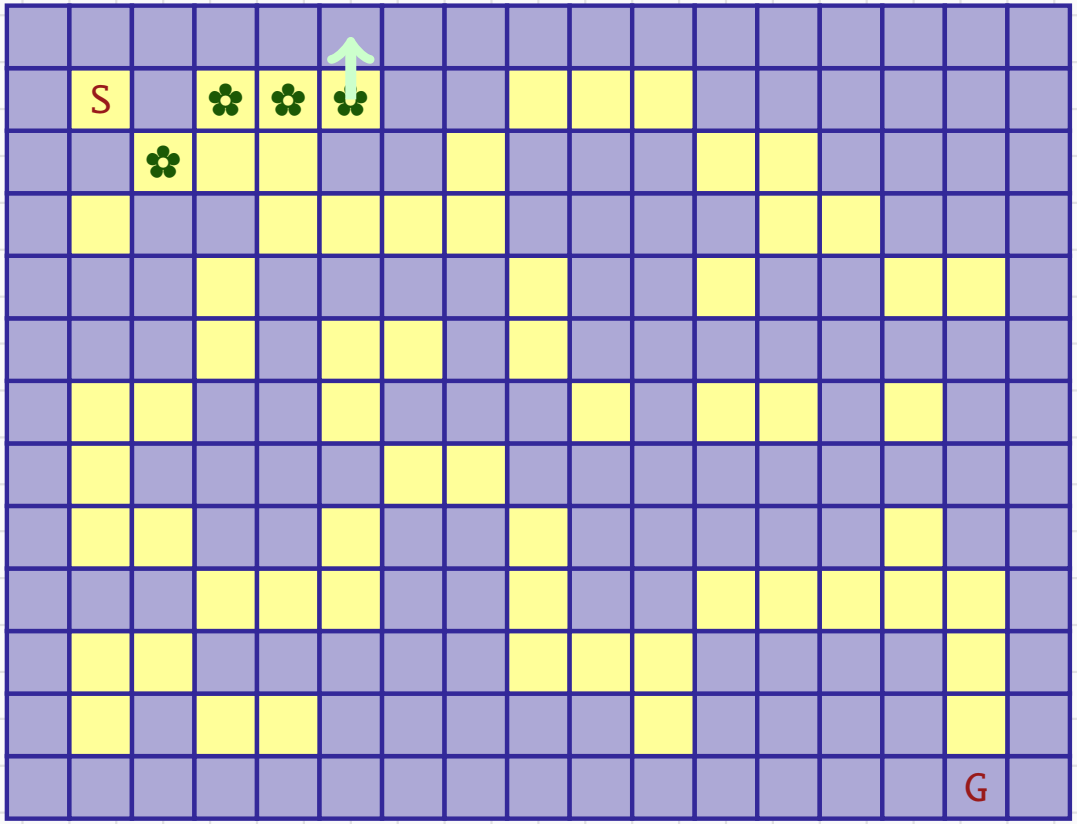
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

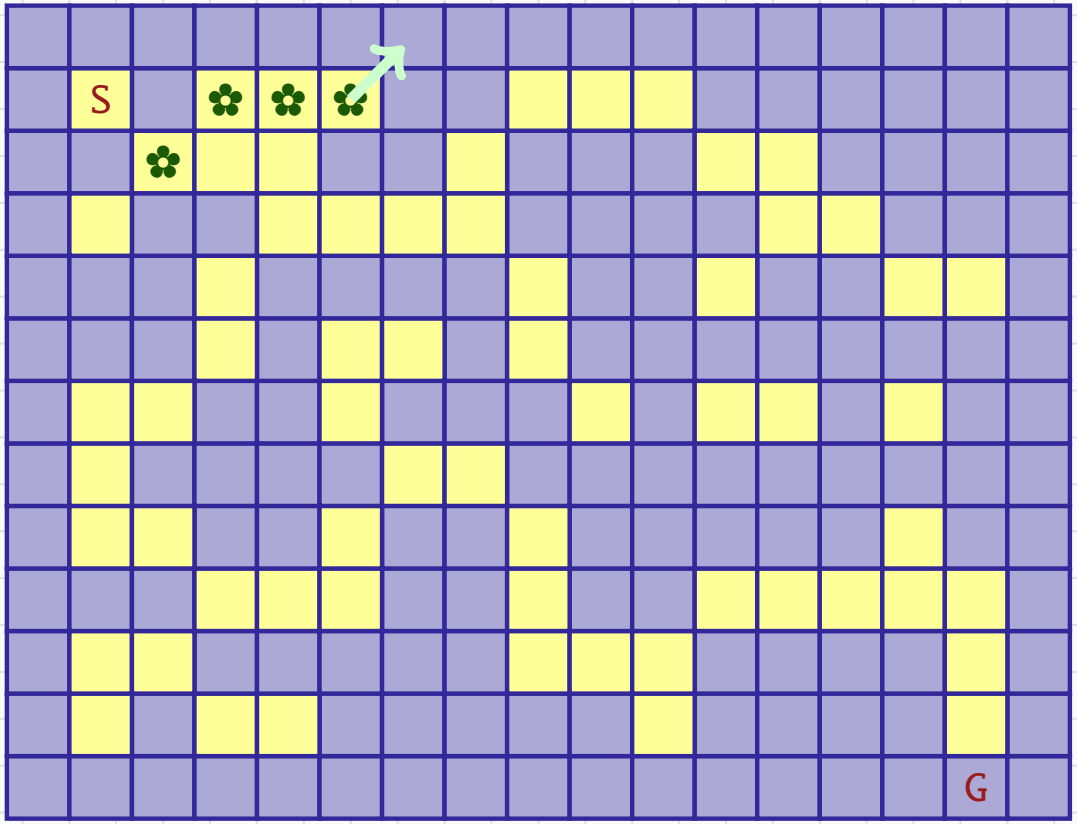
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

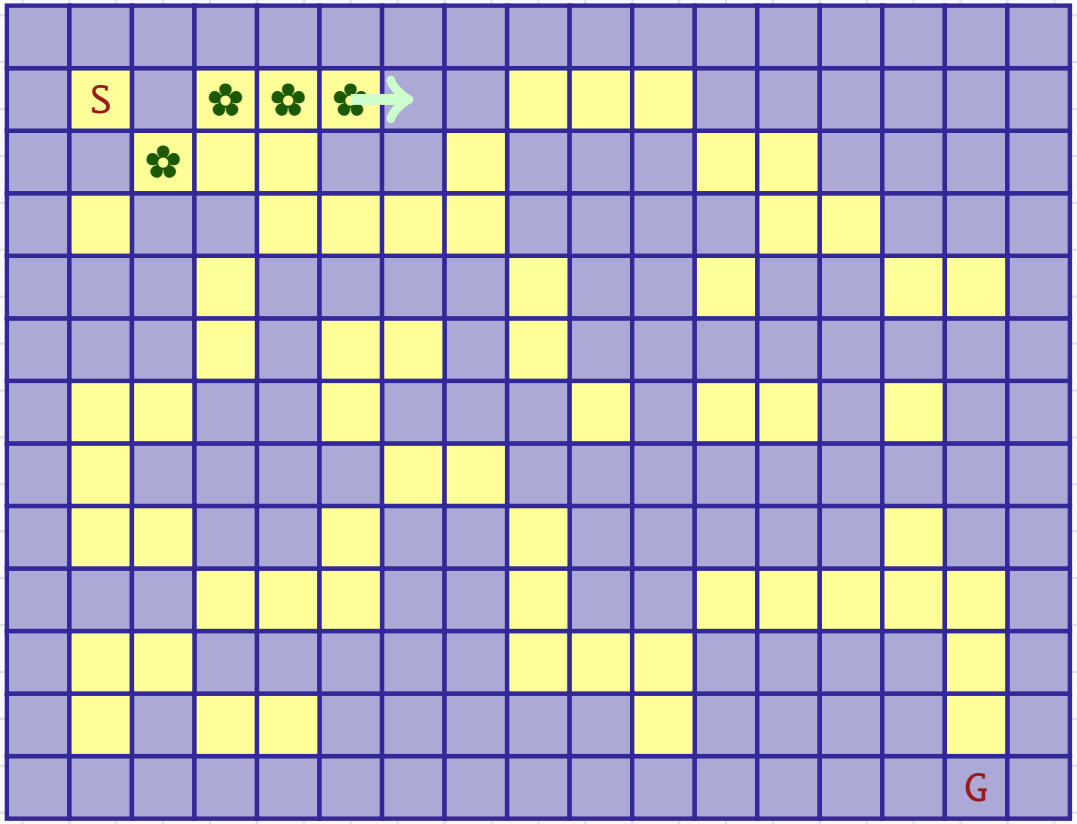
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

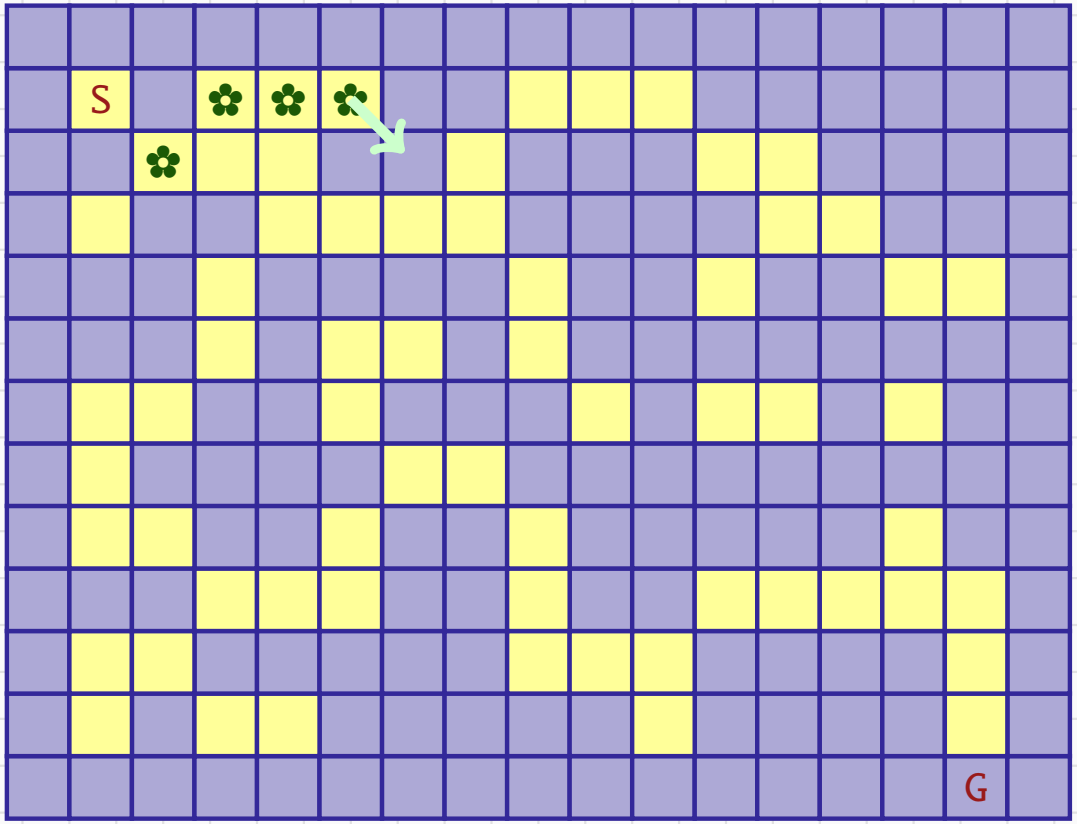
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

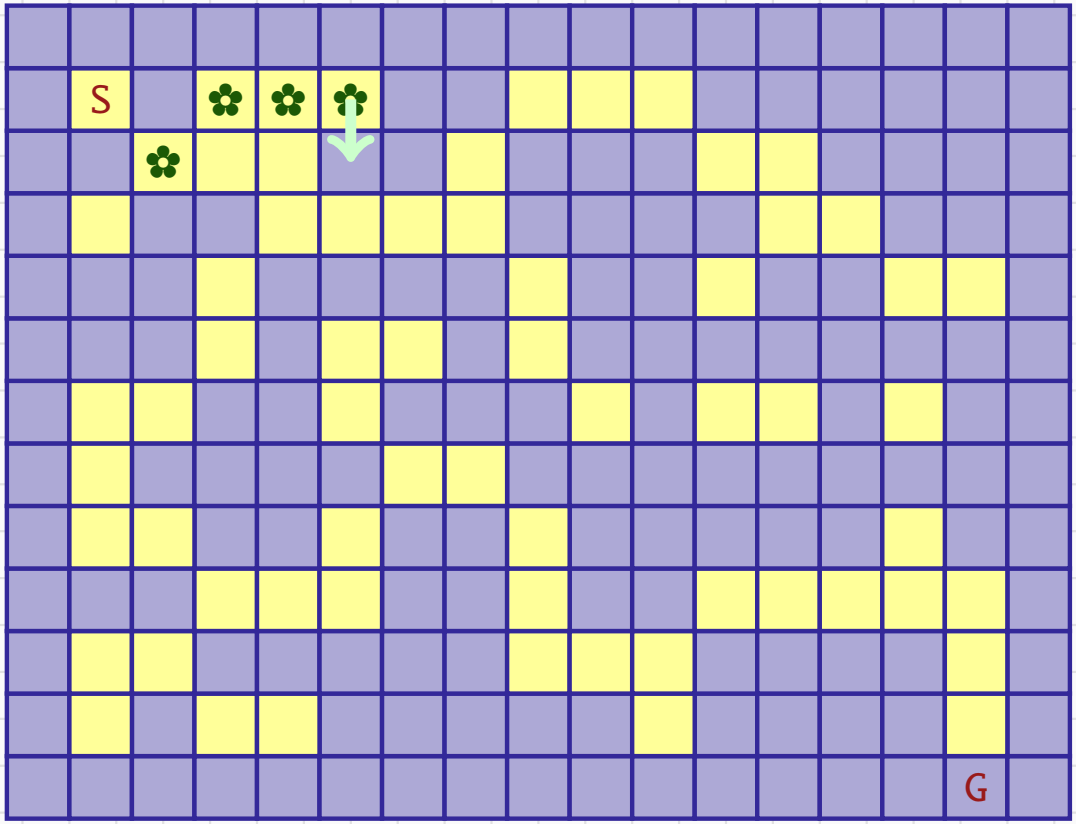
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

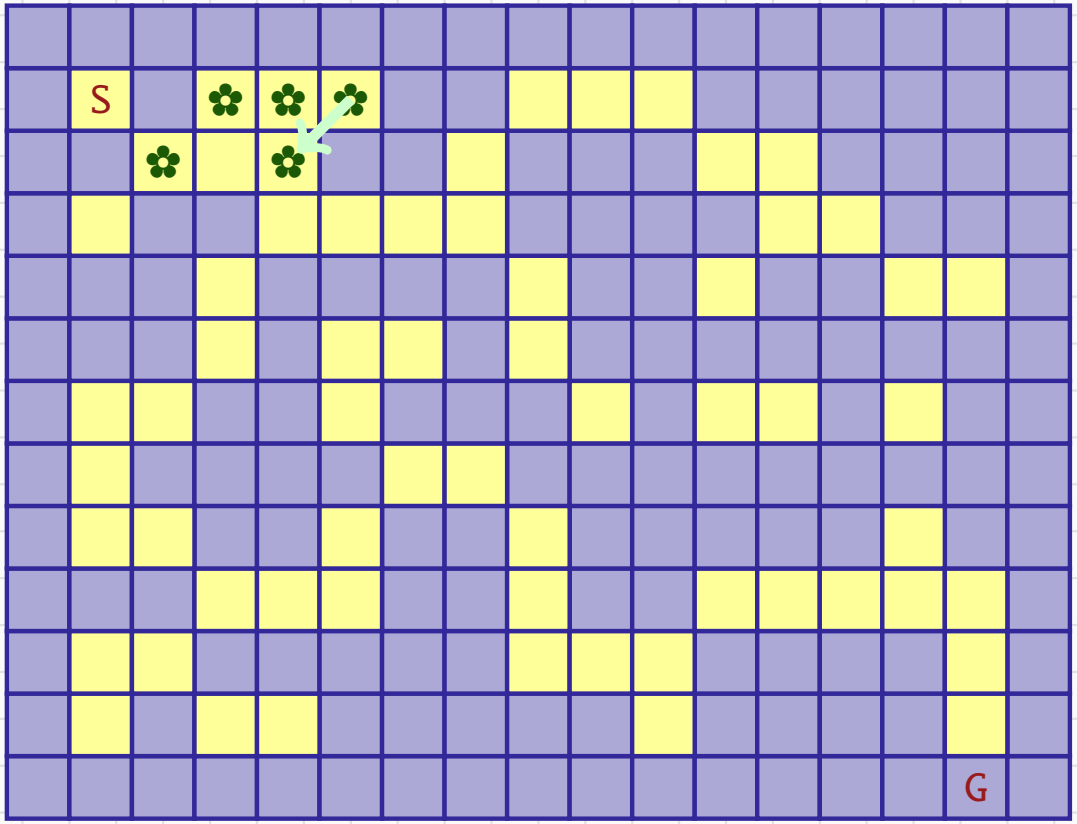
Finding a Path



r	c	d
1	4	3
1	3	3
2	2	2
1	1	4

stack

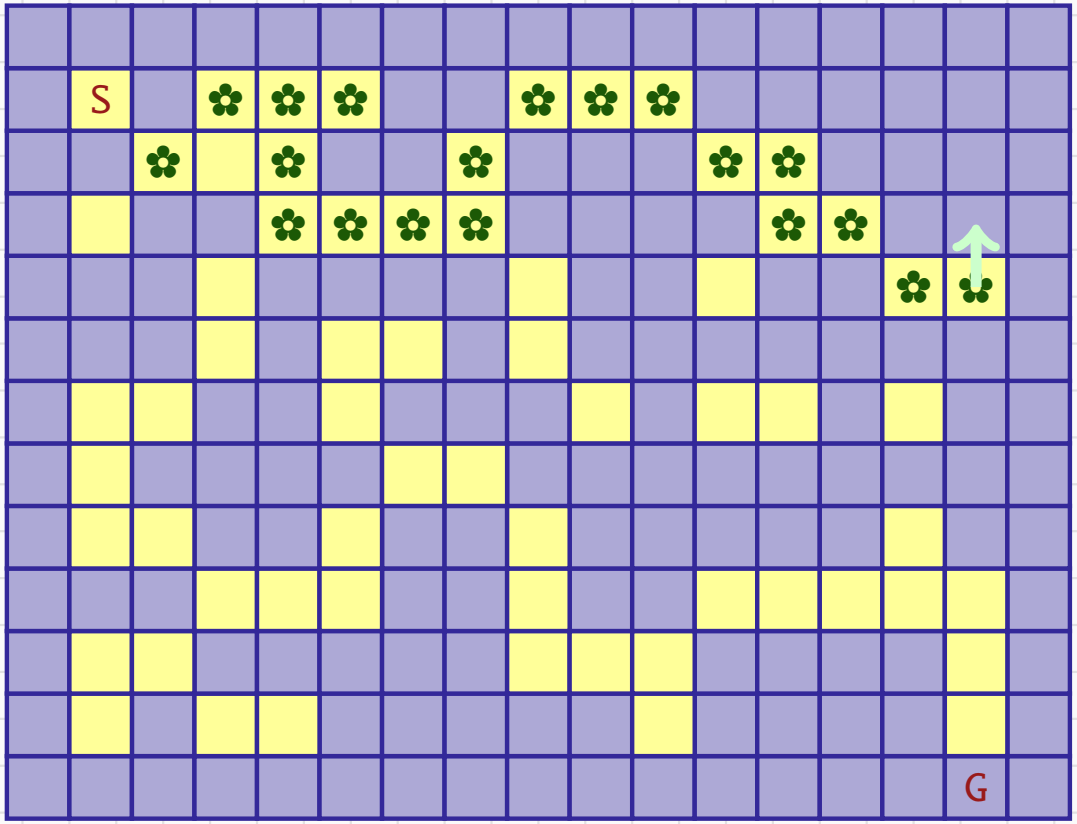
Finding a Path



r	c	d
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

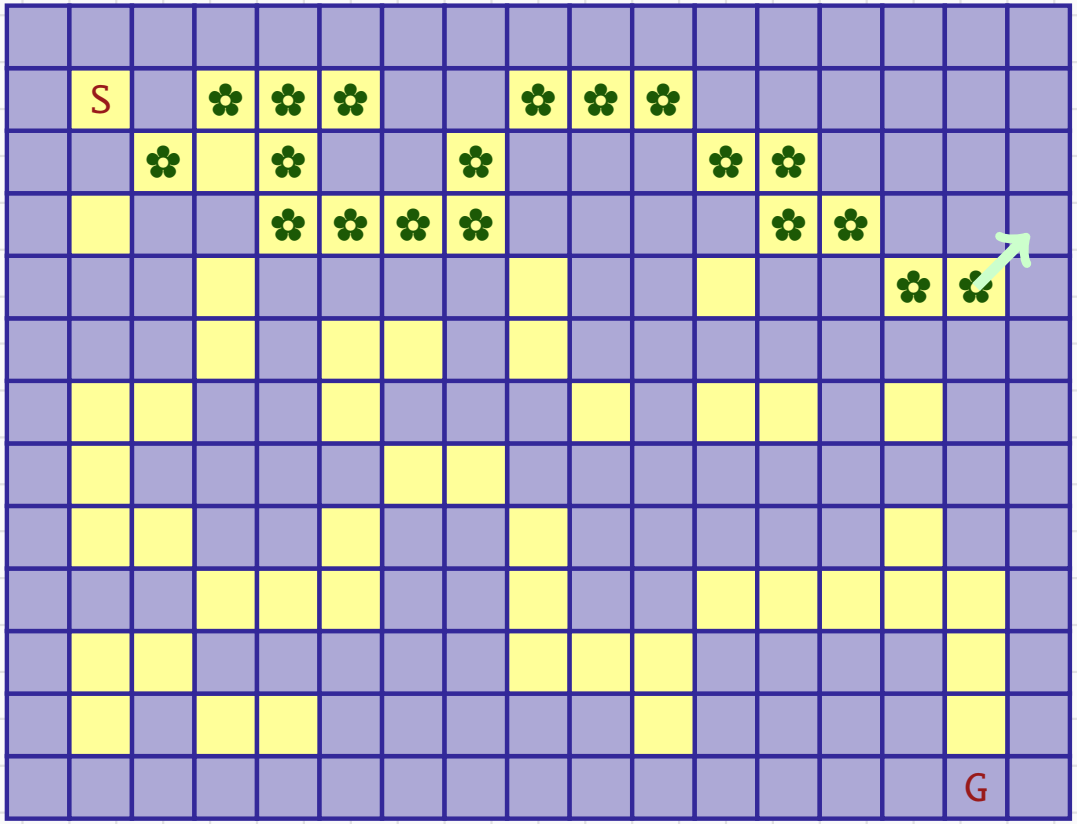
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

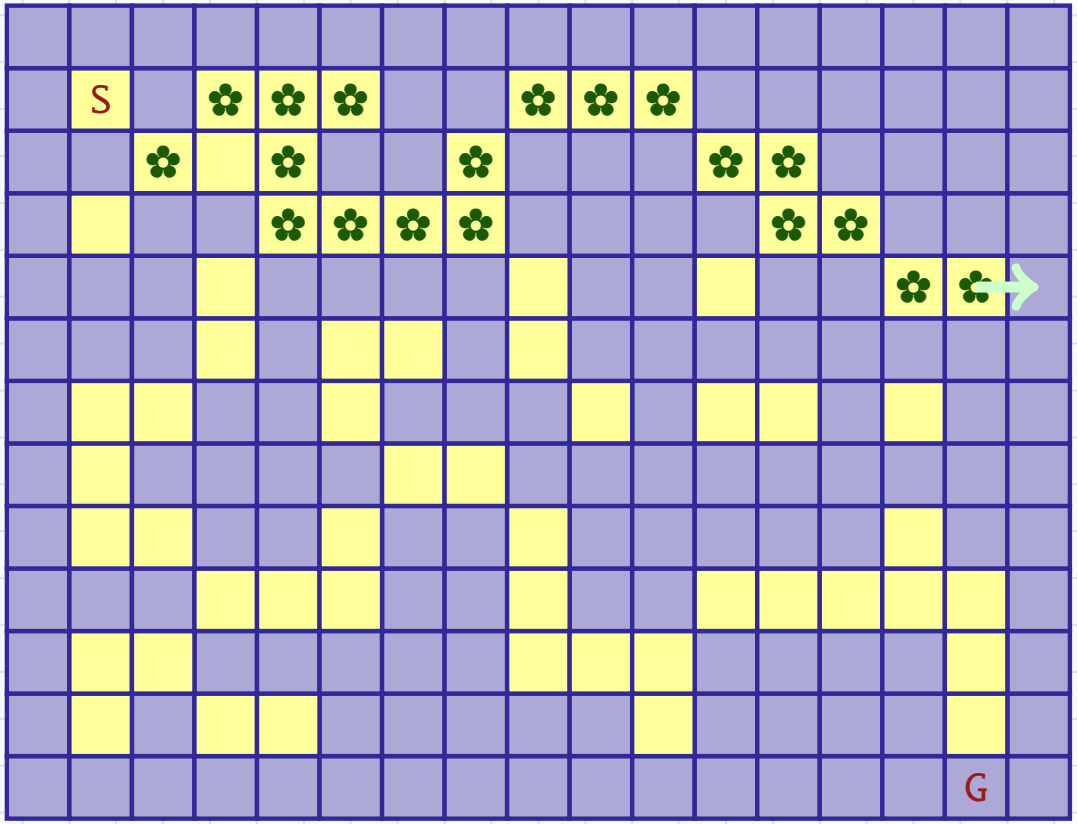
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

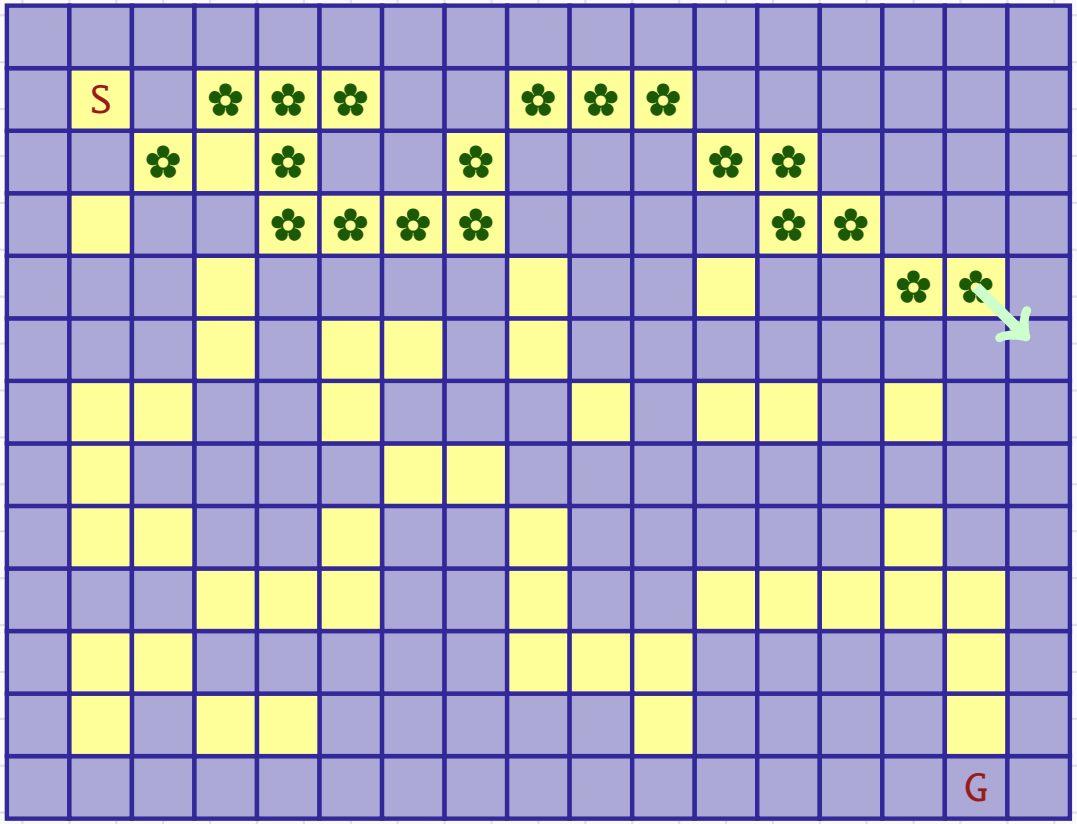
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

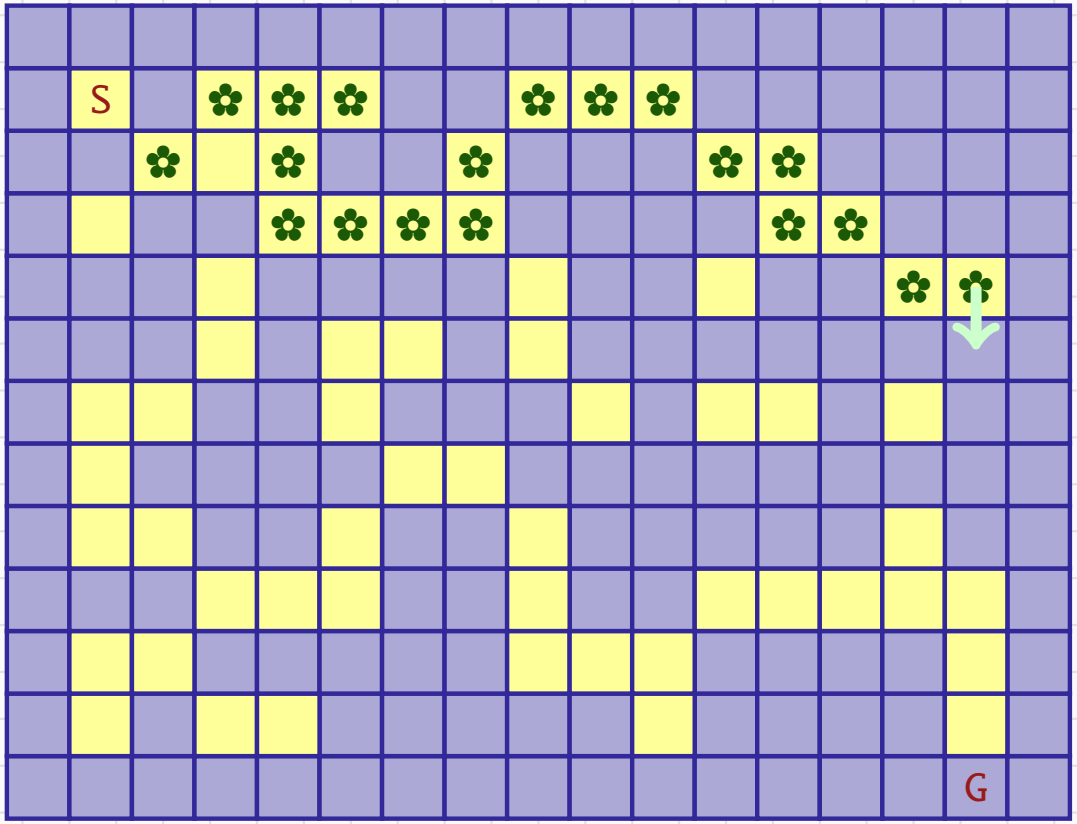
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

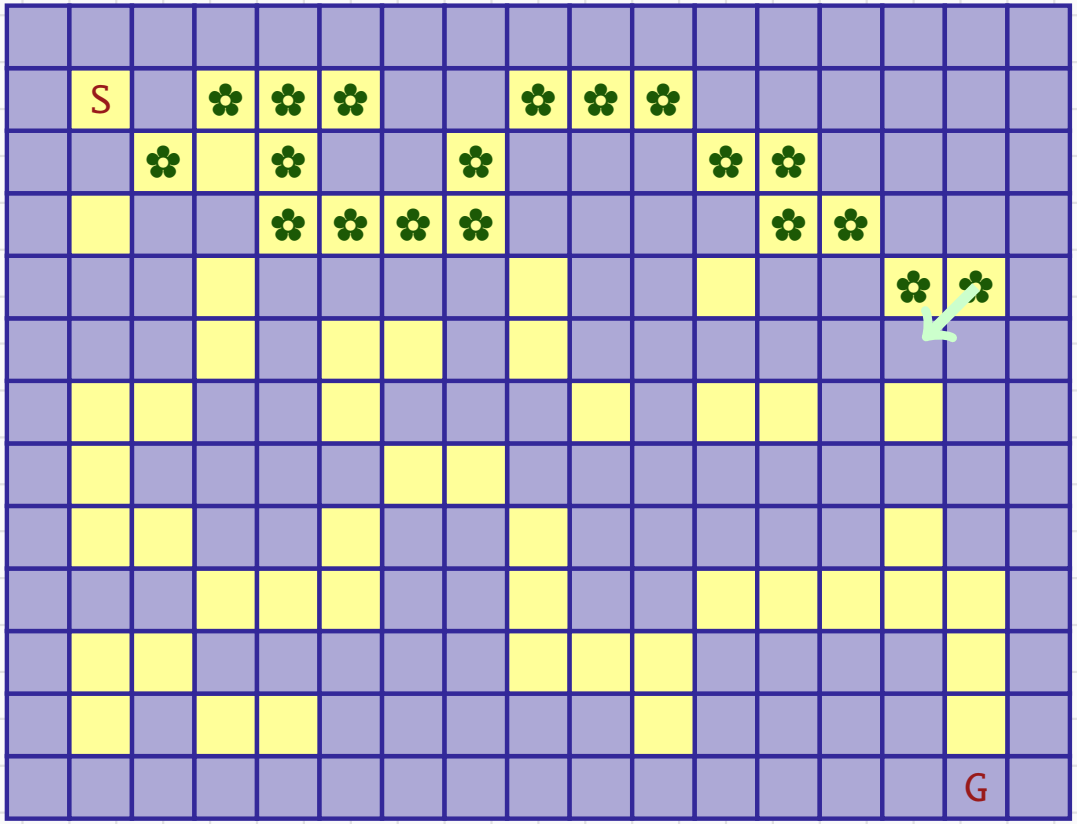
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

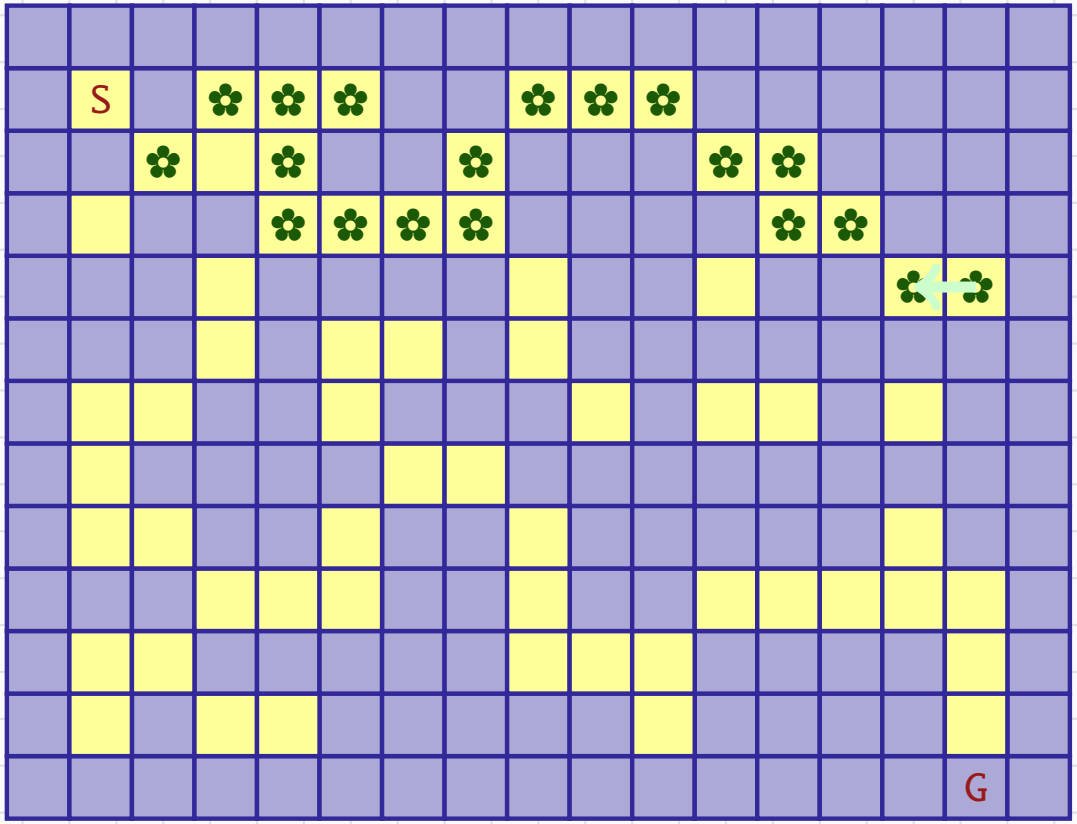
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

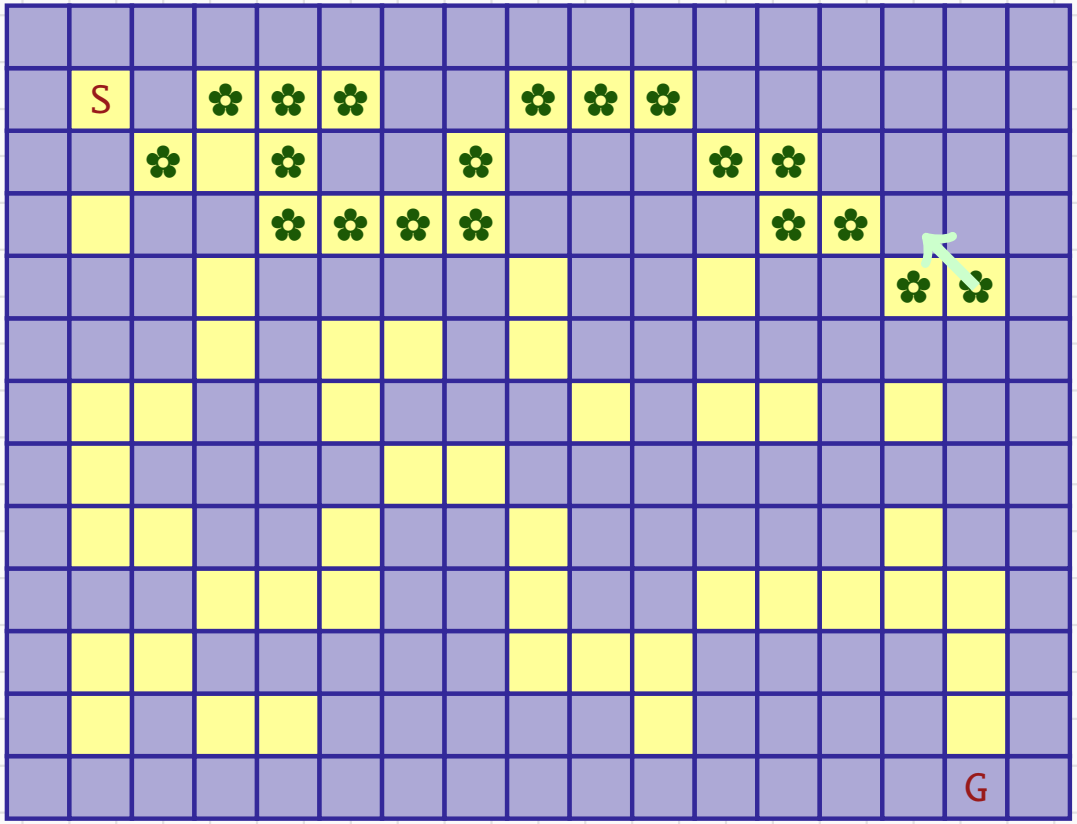
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

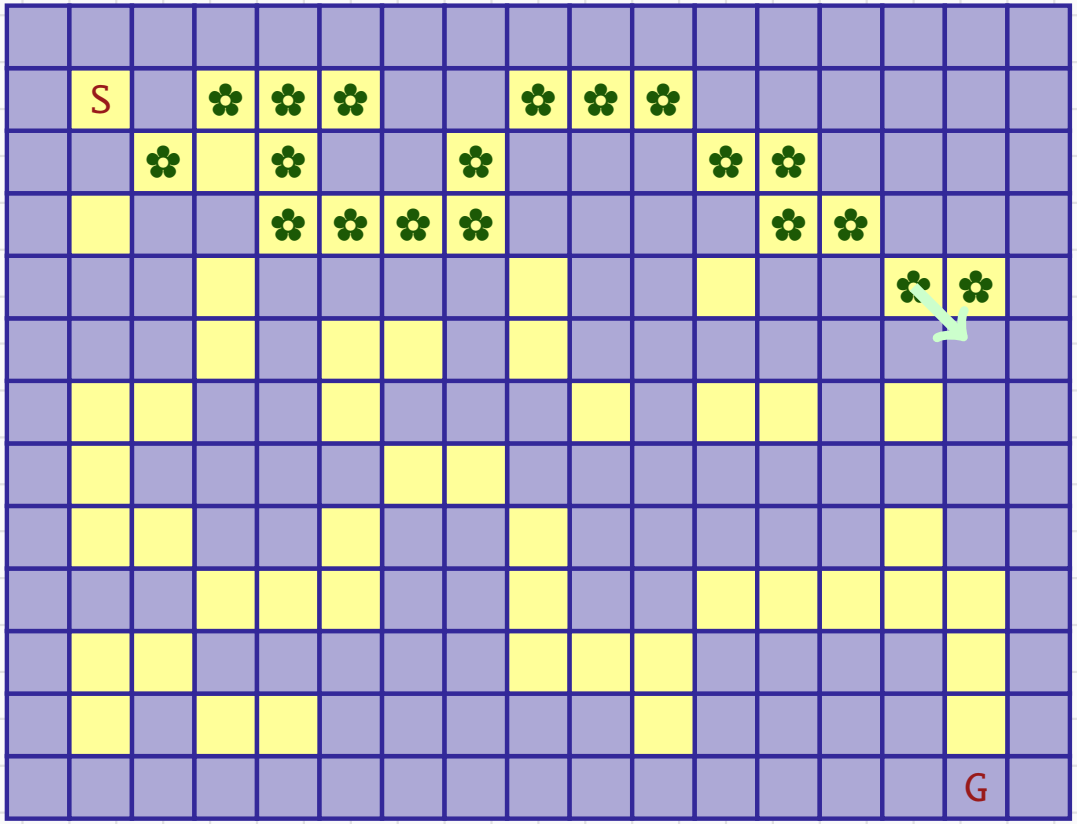
Finding a Path



r	c	d
4	14	3
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

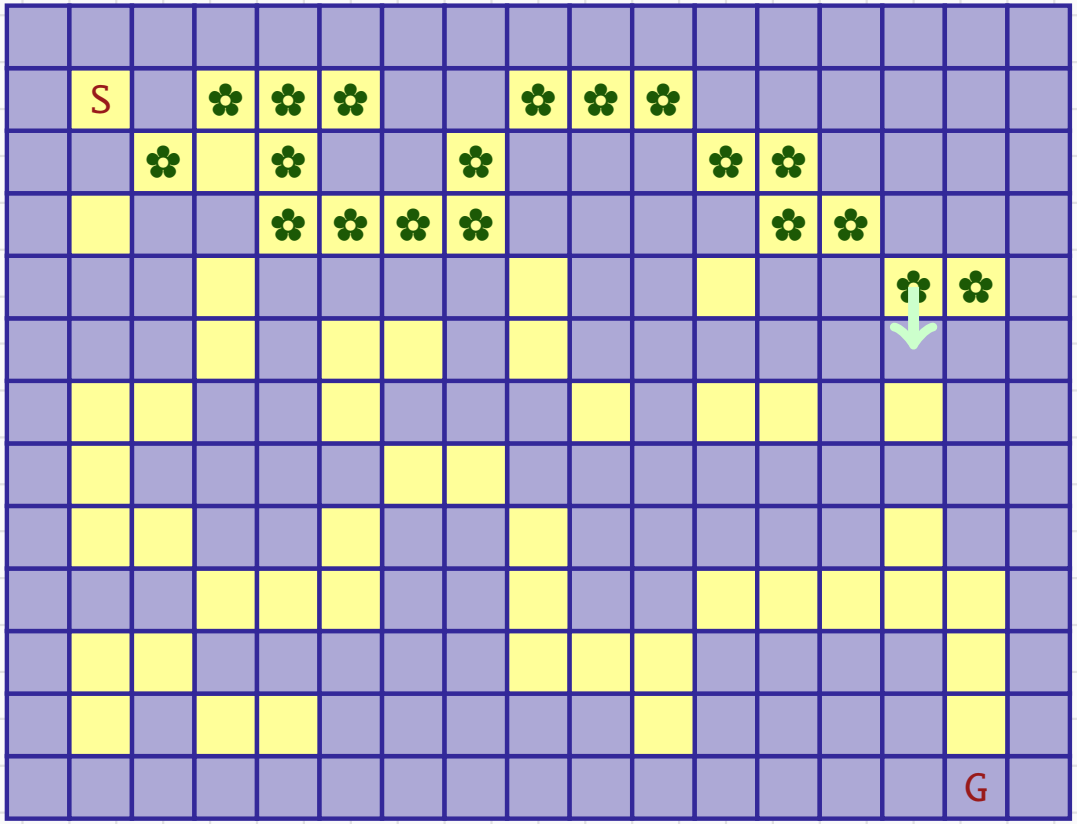
Finding a Path



	pop	
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4
r	c	d

stack

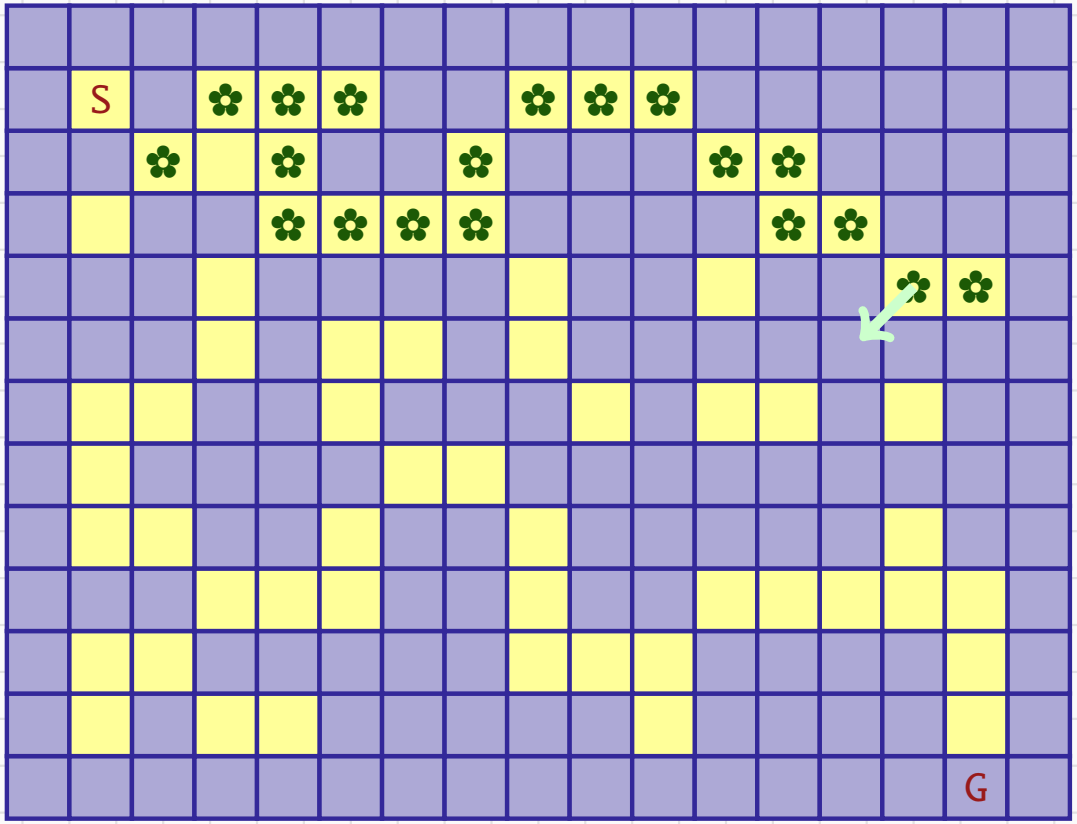
Finding a Path



r	c	d
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

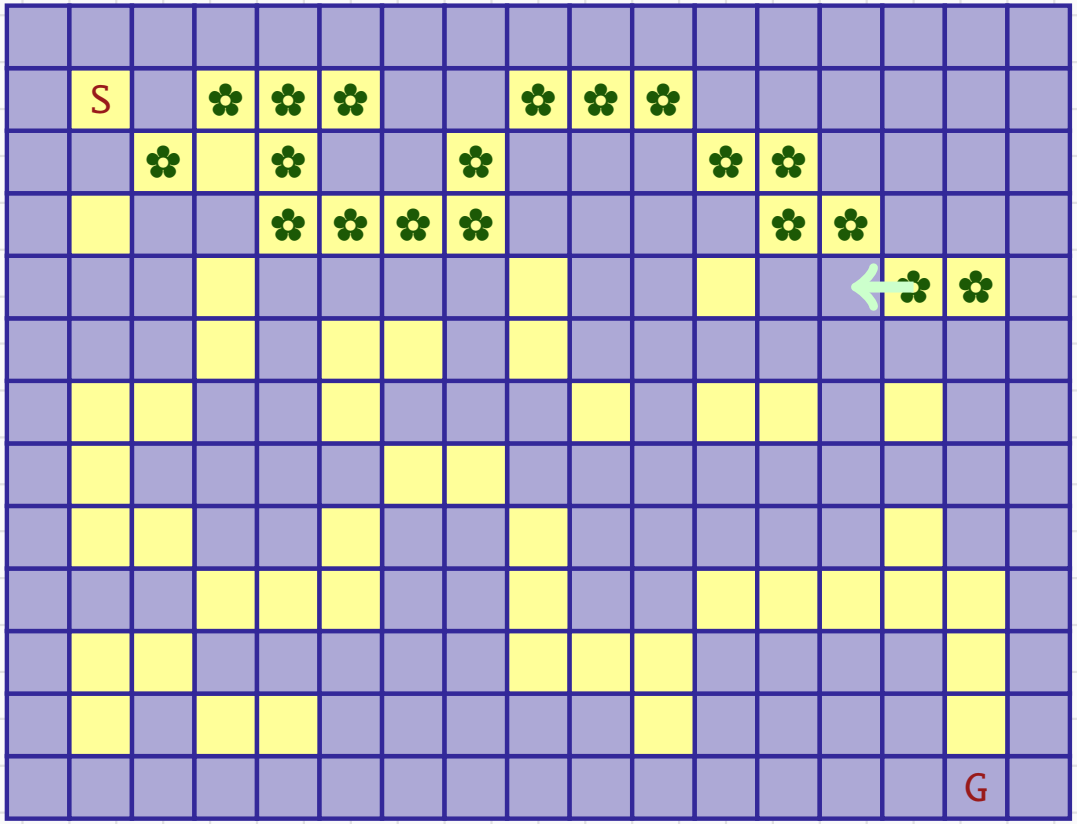
Finding a Path



r	c	d
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

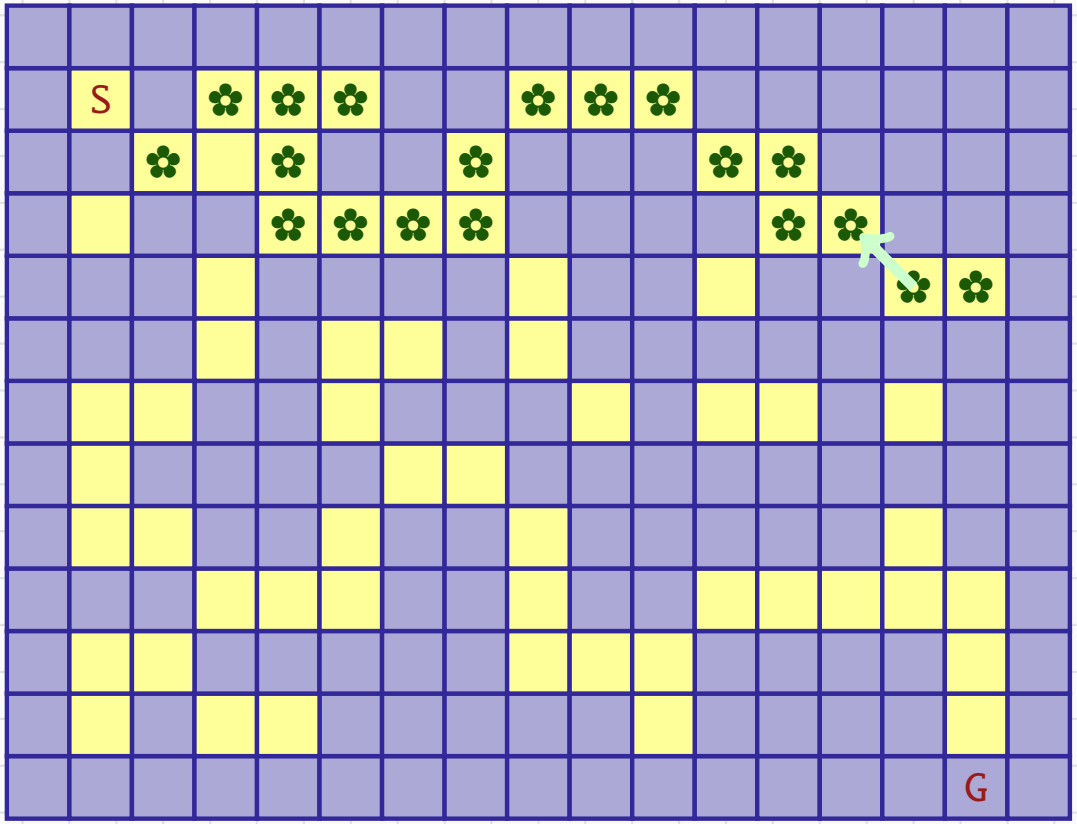
Finding a Path



r	c	d
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

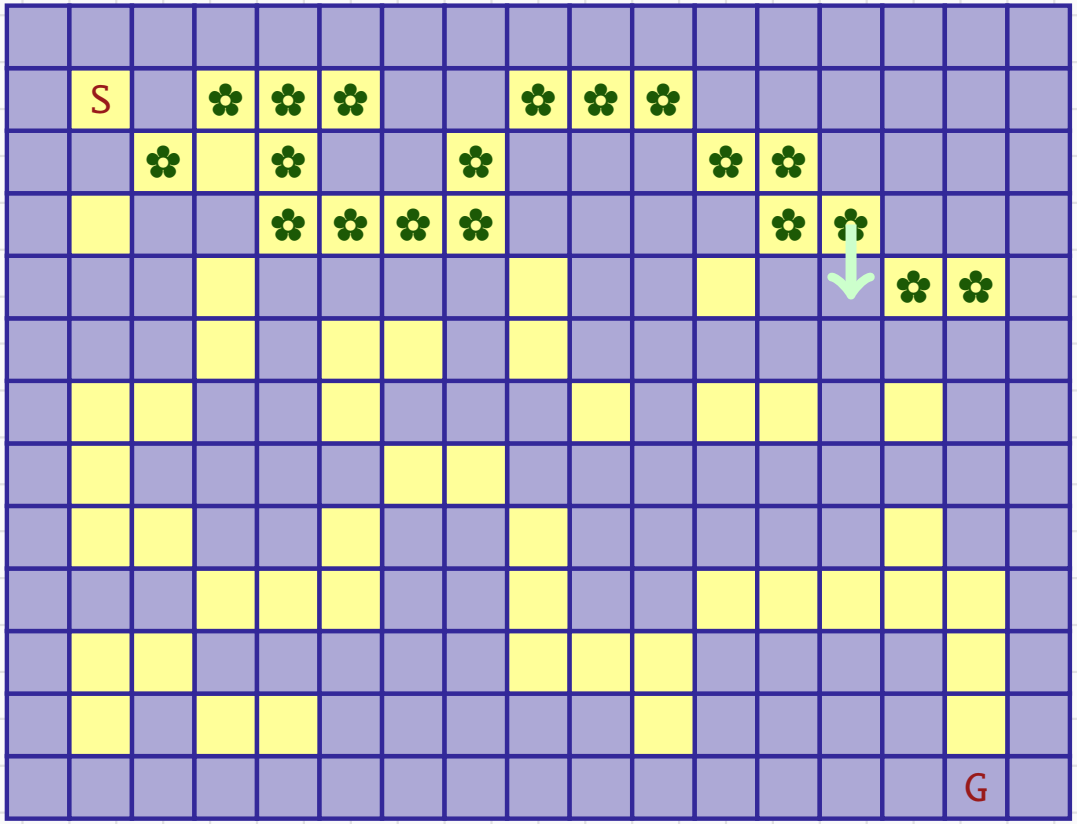
Finding a Path



r	c	d
3	13	4
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

Finding a Path



pop

r	c	d
2	12	4
2	11	3
1	10	4
1	9	3
1	8	3
2	7	2
3	6	2
3	5	3
2	4	4
1	5	6
1	4	3
1	3	3
2	2	2
1	1	4

stack

```
void path(void)
{
     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) {
               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;
          }
     }
...
```

Analysis of `path`

- The size of stack

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- When searching the maze for the entrance to exit path, all positions (except the exit) with value zero will be on the stack when the exit is reached.
- The worst case of computing time of path is $O(mp)$, where m and p are the number of rows and columns of the maze respectively.

Expressions

Precedence Hierarchy

$$x = a/b - c + d * e - a * c$$

An expression contains operators, operands, and parentheses

precedence rule

+

associative rule

Token	Operator	Precedence[1]	Associativity
() [] –> . struct or union member	function call array element	17	left-to-right
–– ++	increment, decrement[2]	16	left-to-right
–– ++ ! ~ – + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ –	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += –= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

Postfix Expressions

- Expression forms
  - Prefix form: operator before its operands
  - Infix form: operator in the middle of its operands
  - Postfix form: operator after its operands
- Example:
  - Infix form: $a * b + c + d * e$
  - Postfix form: $ab * c + de * +$
- **Remark:**
  - Infix form:
    Familiar to human ➠ but not easy to calculate by computers
  - Postfix form:
    No need to know any precedence rule ➠ easy for computers
    Compilers use a parenthesis-free notation referred to as postfix

Postfix Evaluation

To evaluate an expression we make a single left-to-right scan of it.

- When encounter an **operand**, push it into the stack
- When encounter an **operator**, then pop out the last two operands and perform the operation on the two operands. Push the result back to the stack.

$(6/2-3)+4*2$

$=$ 6 2 / 3 - 4 2 * +

using a stack

token	stack [0]	stack [1]	stack [2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2 − 3			0
4	6/2 − 3	4		1
2	6/2 − 3	4	2	2
*	6/2 − 3	4 * 2		1
+	6/2 − 3 + 4 * 2			0

```
int eval(void){
    precedence token;
    char symbol;
    int op1, op2;
    int n=0, top = -1;
    token = getToken(&symbol, &n);
    while(token != eos) {
        if (token == operand)
            push( symbol-'0' );
        else {
            op2 = pop();
            op1 = pop();
            switch(token) {
                case plus:  push(op1+op2); break;
                case minus:  push(op1-op2); break;
                case times:  push(op1*op2); break;
                case divide:  push(op1/op2); break;
                case mod:  push(op1%op2);
            }
        }
        token = getToken(&symbol, &n);
    }
    return pop();
}
```

```
typedef enum {lparen, rparen, plus, minus,
    times, divide, mod, eos, operand
} precedence;
```

Assumptions:

- operators: $(,), +, -, *, /, \%$
- operands: single digit integer or variable of one character

```
precedence getToken(char *symbol, int *n)
{ /* get the next token, symbol is the character representation
    *symbol =expr[(*n)++];
    switch (*symbol) {
        case '(' :  return lparen;
        case ')' :  return rparen;
        case '+' :  return plus;
        case '-' :  return minus;
        case '/' :  return divide;
        case '*' :  return times;
        case '%' :  return mod;
        case '\0' :  return eos;
        default :  return operand;
    }
}
```

Infix to Postfix Transformation

We can produce a postfix expression from an infix one as follows:

- Fully parenthesize expression
  $a/b - c + d * e - a * c$
  $((((a/b) - c) + (d * e)) - (a * c))$
- All operators replace their corresponding right parentheses
  $((\underbrace{((a/b) - c)} + (d * e)) - (a * c))$
- Delete all parentheses
  $ab/c - de * + ac * -$

  The order of operands is the same in infix and postfix

Translation of $a + b * c$ to Postfix

- Two operators need to be reversed.
- Operators with high precedence must be output before those with lower precedence.
- In general operators with higher precedence must be output before those with lower precedence.

token	stack [0]	[1]	[2]	top	output
a				-1	a
$+$	$+$			0	a
b	$+$			0	ab
$*$	$+$	$*$		1	ab
c	$+$	$*$		1	abc
eos				-1	$abc*+$

Translation of $a * (b + c) * d$ to Postfix

- We stack operators until we reach the right parenthesis.
- We unstack until we reach the corresponding left parenthesis. Then delete left parenthesis.

token	stack [0]	stack [1]	stack [2]	top	output
a				-1	a
$*$	$*$			0	a
$($	$*$	$($		1	a
b	$*$	$($		1	ab
$+$	$*$	$($	$+$	2	ab
c	$*$	$($	$+$	2	abc
$)$	$*$			0	$abc+$
$*_2$	$*_2$			0	$abc + *$
d	$*_2$			0	$abc + *d$
eos				-1	$abc + *d*_2$

Stack or Unstack

Rule

- Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
- '(' has low in-stack precedence, and high incoming precedence.

op	(	)	+	−	*	/	%	*eos*
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
void postfix(void){
    char *symbol;
    precedence token;
    int n = 0, top = 0;
    stack[0] = eos;
    for (token = getToken(&symbol, &n); token != eos;
         token = getToken(&symbol, &n)){
        if (token == operand)
            printf("%c", symbol);
        else if (token == rparen){
            while (stack[top] != lparen)
                printToken (pop());
            pop();
        }
        else {
            while (isp[stack[top]] >= icp[token])
                printToken(pop());
            push(token);
        }
    }
    while ( (token = pop()) != eos)
        printToken(token);
    printf("\n");
}
```

Let n be the number of tokens.
The total time spent is $\Theta(n)$.