CSC212

Data Structure

- Section FG

COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

Lecture 12
Stacks and Queues

Instructor: Feng HU

Department of Computer Science

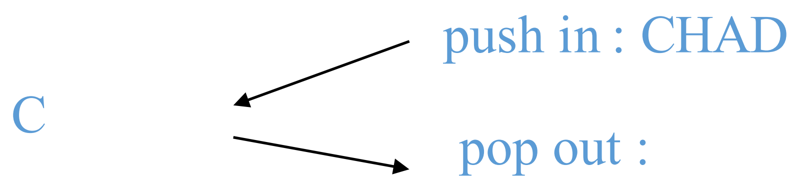
City College of New York

Topics

- Stacks (Chapter 7)
- Queues (Chapter 8, Section1 - 3)
- Priority Queues (Chapter 8, Section 4)
- References Return Values (Chapter 8, Section 5)

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion



Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd} & \text{push in : CHAD} \arrow[dl] \\ \text{CH} \arrow[dr] & \\ & \text{pop out :} \end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd}
 & \text{push in : CHAD} \arrow[dl] \\
\text{CHA} \arrow[dr] & \\
 & \text{pop out :}
\end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd}
 & \text{push in : CHAD} \arrow[dl] \\
\text{CHAD} \arrow[dr] & \\
 & \text{pop out :}
\end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd}
& \text{push in : CHAD} \arrow[dl] \\
\text{CHA} \arrow[dr] & \\
& \text{pop out :D}
\end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd}
 & \text{push in : CHAD} \arrow[dl] \\
\text{CH} \arrow[dr] & \\
 & \text{pop out :DA}
\end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

$$\begin{tikzcd}
 & \text{push in : CHAD} \arrow[dl] \\
C \arrow[dr] & \\
 & \text{pop out : DAH}
\end{tikzcd}$$

Stacks and the STL **stack**

- Definition
  - A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- LIFO
  - A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

push in : CHAD

pop out : DAHC

Stacks and the STL **stack**

- The STL stack class
  - a container class – holding many items
  - a template class – stack of integers, strings, ...
- How to use
  - #include <stack>
  - stack<int> s1;
- Implementation it ourselves! (stack code)
  - fixed-size or dynamic array, or linked list?
  - STL typically uses dynamic array
  - Functions: push, pop, empty, size , **top**

Queues and the STL queue

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ C $\longrightarrow$ take out :

Queues and the STL **queue**

- Definition
  - A **queue** is a data structure of *ordered* entries such that entries can only be inserted at one end (call the **rear**) and removed at the other end (call the **front**) – and the entry at the front of the queue is called the **first entry**
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ HC $\longrightarrow$ take out :

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ AHC $\longrightarrow$ take out :

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ DAHC $\longrightarrow$ take out :

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ DAH $\longrightarrow$ take out : C

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

$$\text{put in : CHAD} \longrightarrow \text{DA} \longrightarrow \text{take out : CH}$$

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ D $\longrightarrow$ take out : CHA

Queues and the STL **queue**

- Definition
  - A queue is a data structure of *ordered* entries such that entries can only be inserted at one end (call the rear) and removed at the other end (call the front) – and the entry at the front of the queue is called the first entry
- FIFO
  - A queue is a First-In/First-Out data structure. Entries are taken out of the queue in the same order that they were put into the queue

put in : CHAD $\longrightarrow$ $\longrightarrow$ take out : CHAD

Queues and the STL **queue**

- The STL queue class
  - a container class – holding many items
  - a template class – queue of integers, strings, ...
- How to use
  - #include <queue>
  - queue<char> q1;
- Implementation it ourselves! (queue code)
  - fixed-size or dynamic array, or linked list?
  - STL typically uses dynamic array
  - Functions: push, pop, empty, size, **front**

Priority Queues

- A priority queue is a container class that allows entries to be retrieved according to some specified priority levels.
  - The highest priority entry is removed first
  - Entries with equal priority can be removed according some criterion e.g. FIFO as an queue.
- STL priority_queue<Item> template class
  - #include <queue>
  - priority_queue<int> q2;
  - Functions push, pop, empty, size , **top** *(not **front**!)*
  - *Several ways to specify priority (p. 411)*

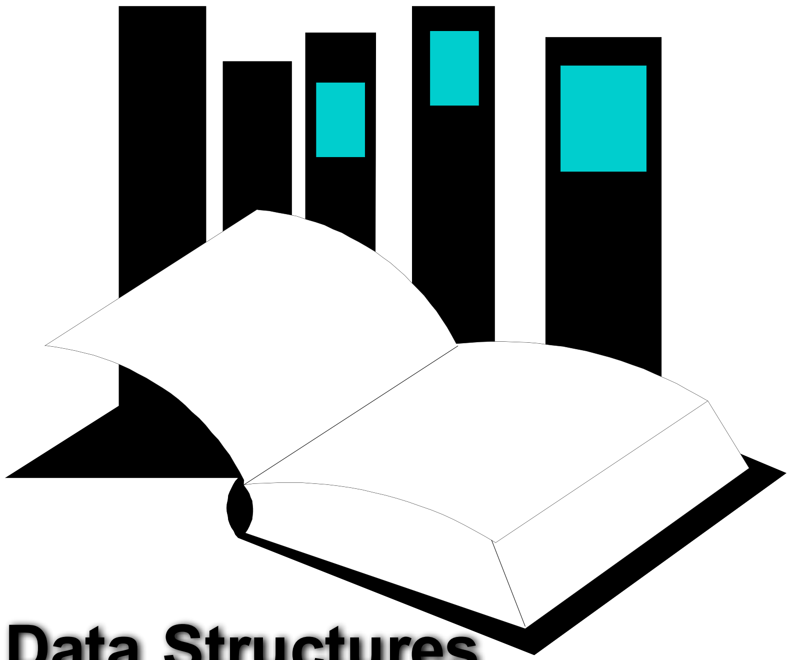
Reference Return Values for the stack, queue, and priority queue classes

- In STL, the top (for stack) and front (for queue) functions have reference return values, e.g. in stack class definition:
  - Item& top ();
  - const Item& top() const;
- Can be used to change the top item
  - If we declare
    - stack<int> b;
    - const stack<int> c;
  - Which ones are correct? =>

1. int i = b.top(); V
2. b.push(i); V
3. b.top() = 18; V
4. c.top() = 18; X
5. b.push(c.top()); V

Using a Stack

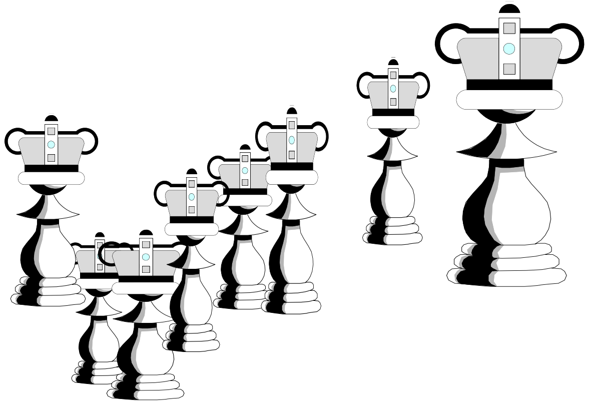
- Chapter 7 introduces the **stack** data type.
- Several example applications of stacks are given in that chapter.
- This presentation shows another use called **backtracking to solve the N-Queens problem**.

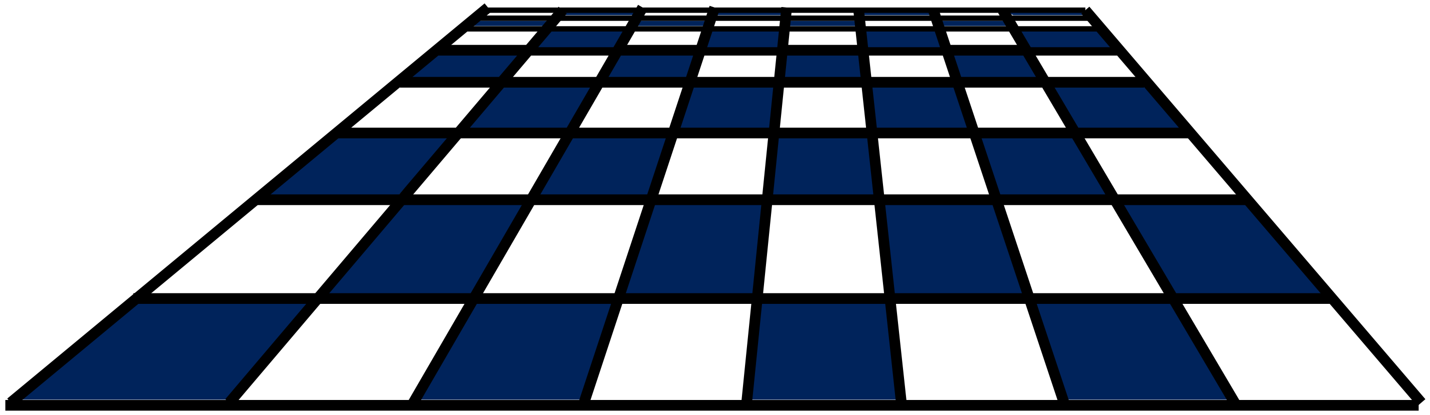


Data Structures and Other Objects Using C++

The N-Queens Problem

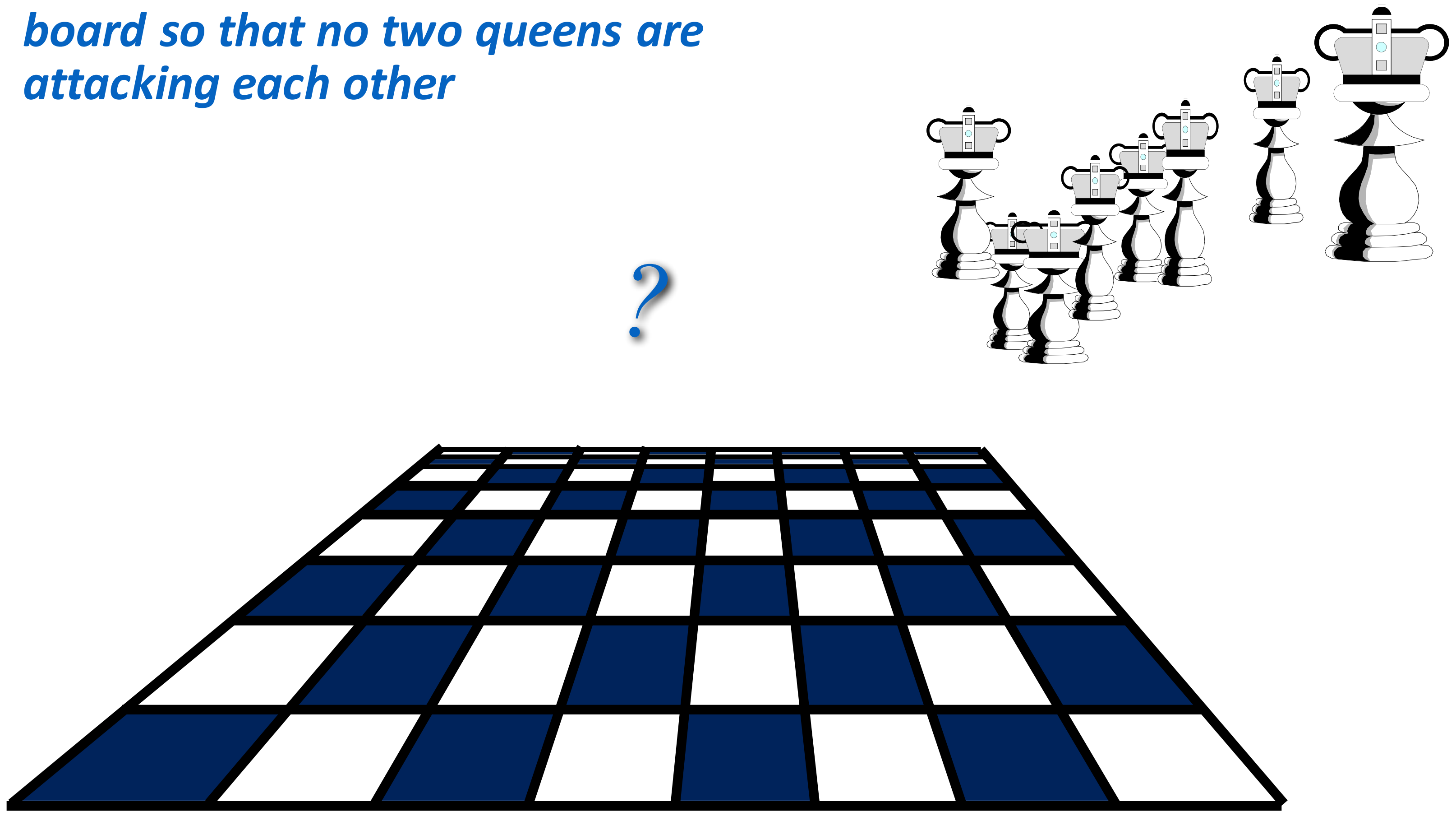
- Suppose you have 8 chess queens...
- ...and a chess board





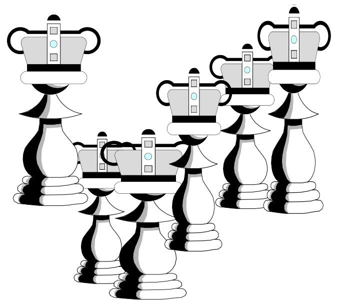
The N-Queens Problem

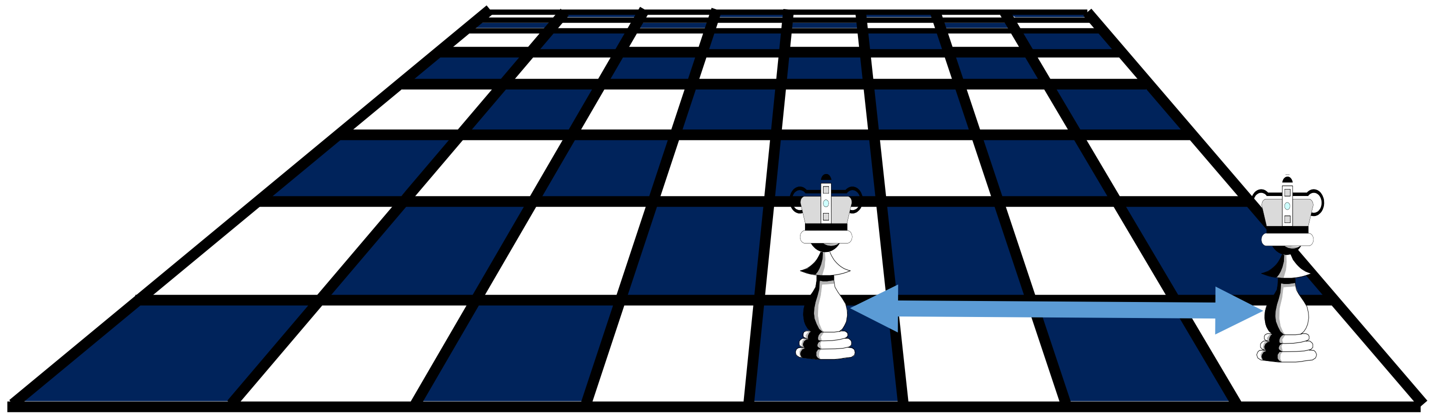
Can the queens be placed on the board so that no two queens are attacking each other



The N-Queens Problem

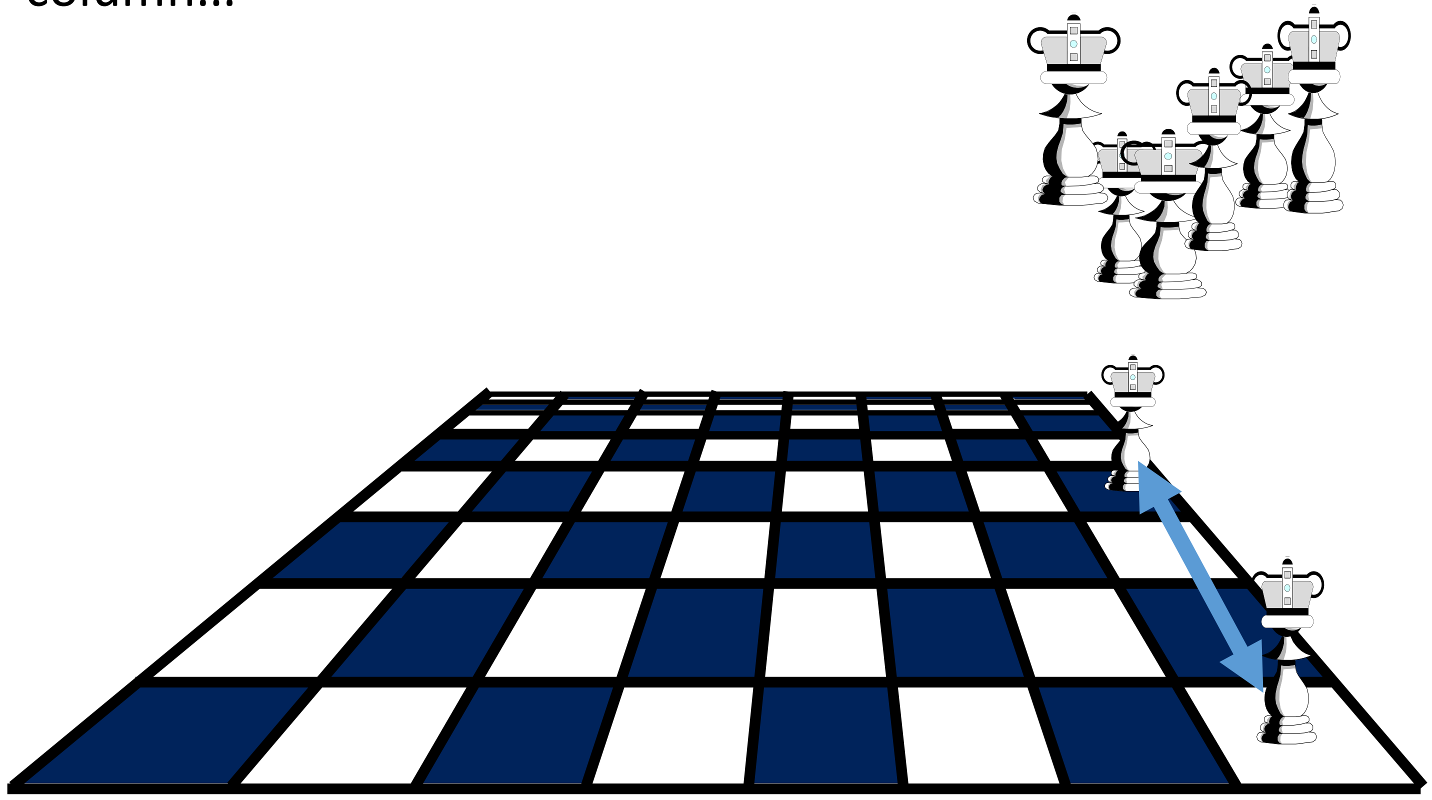
Two queens are not allowed in the same row...





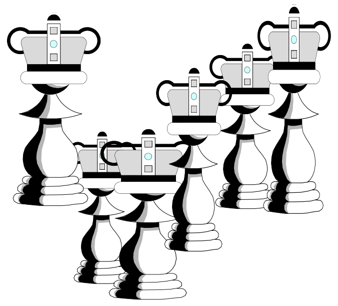
The N-Queens Problem

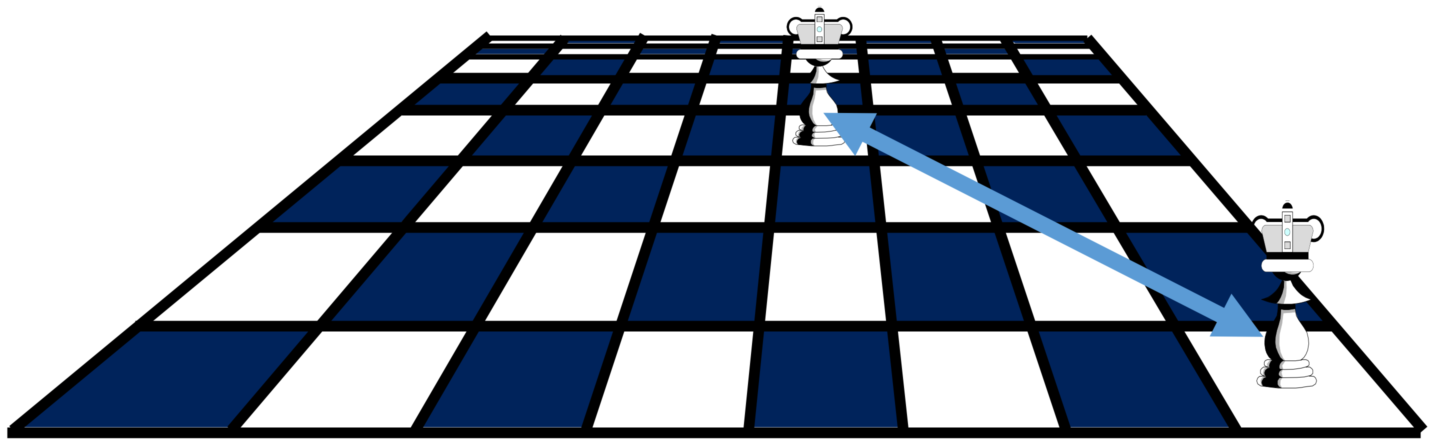
Two queens are not allowed in the same row, or in the same column...



The N-Queens Problem

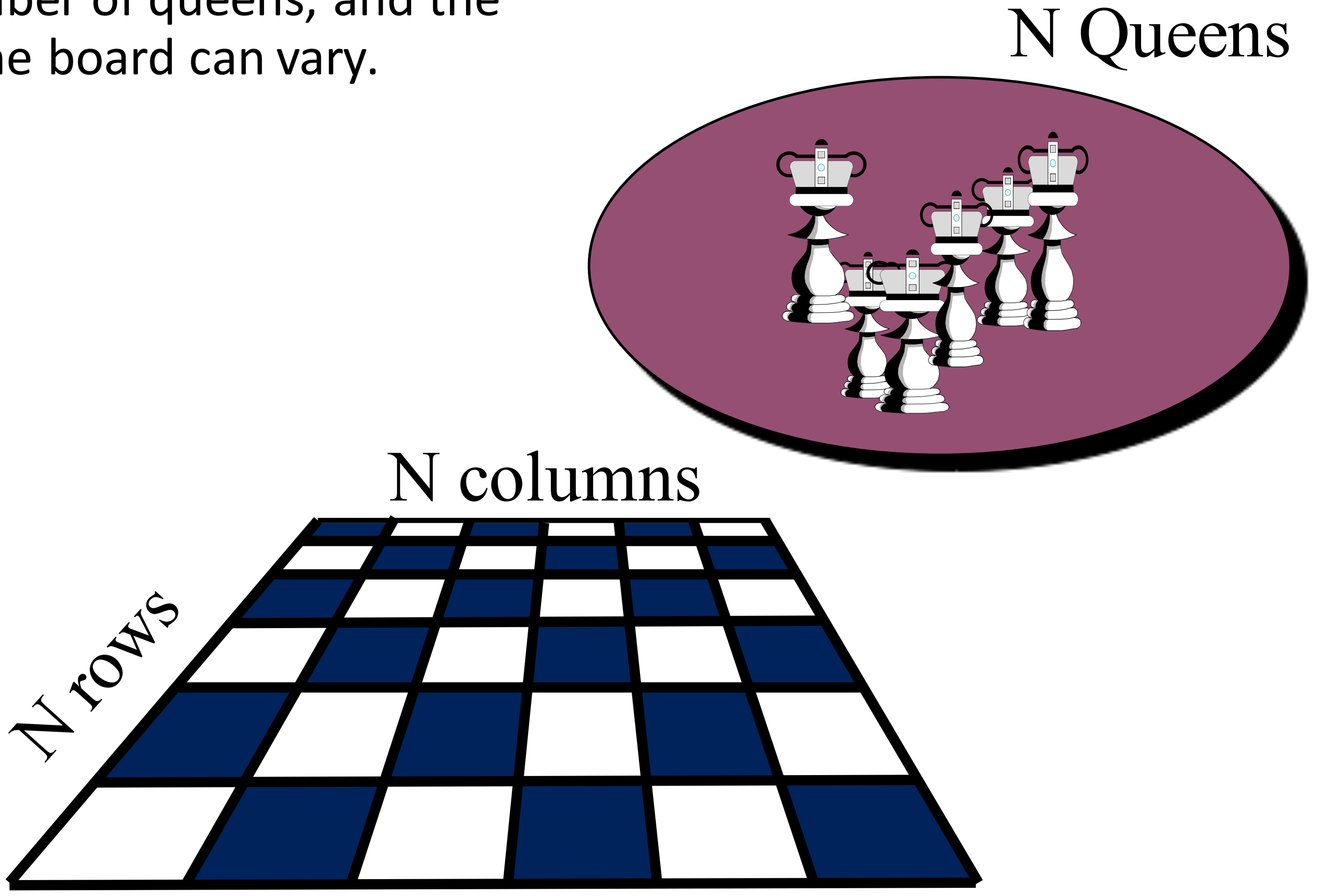
Two queens are not allowed in the same row, or in the same column, or along the same diagonal.





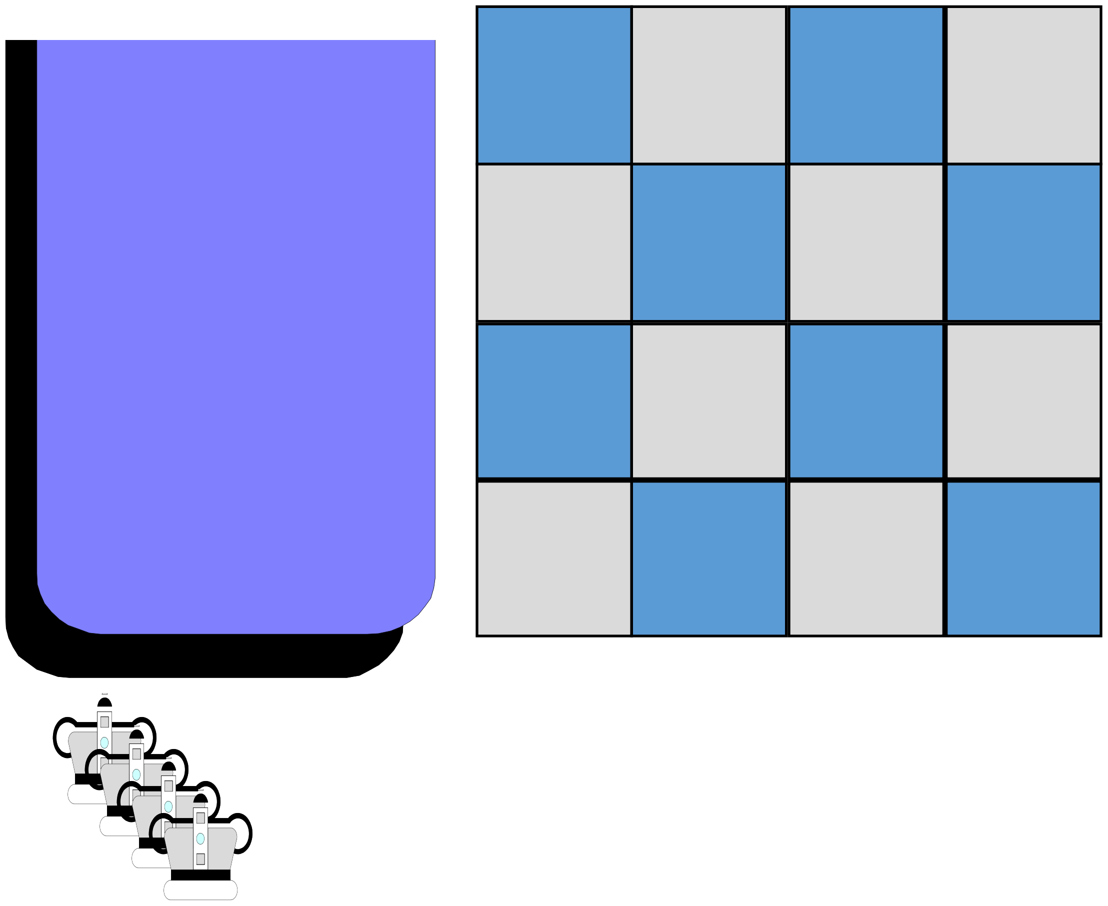
The N-Queens Problem

The number of queens, and the size of the board can vary.



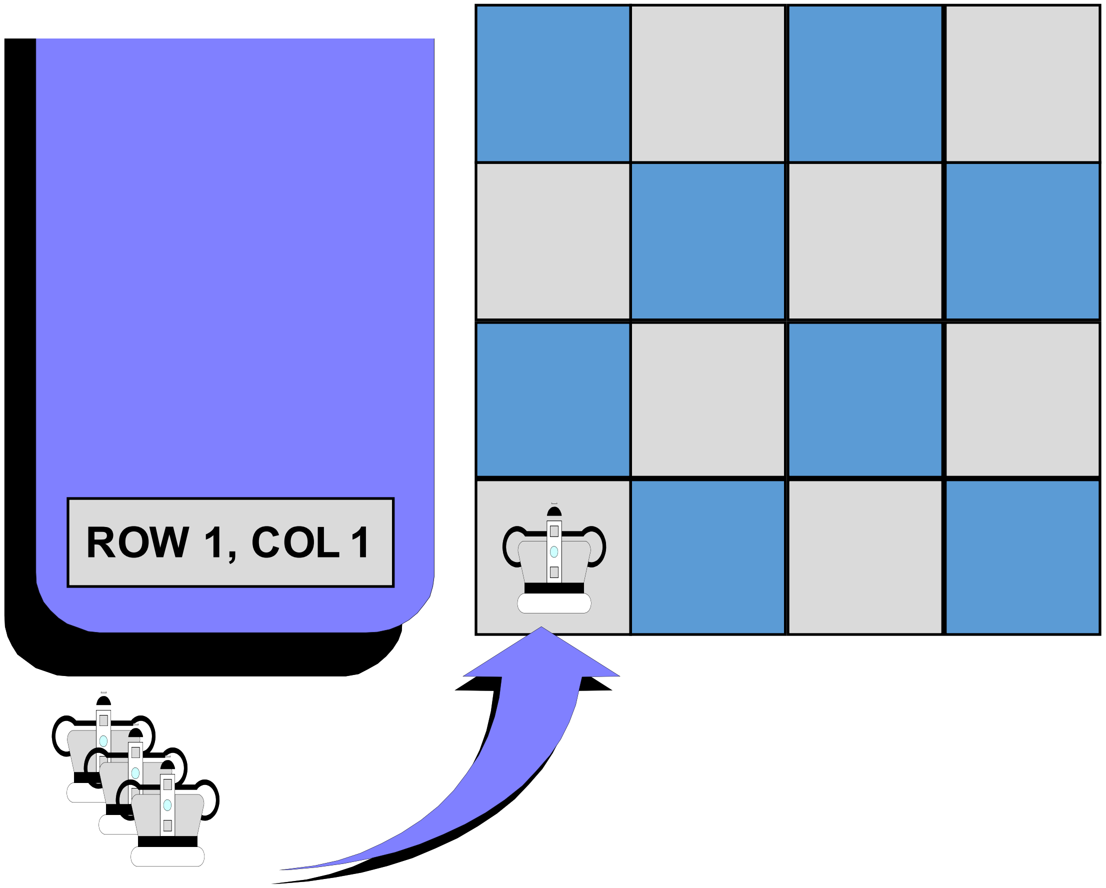
How the program works

The program uses a stack to keep track of where each queen is placed.



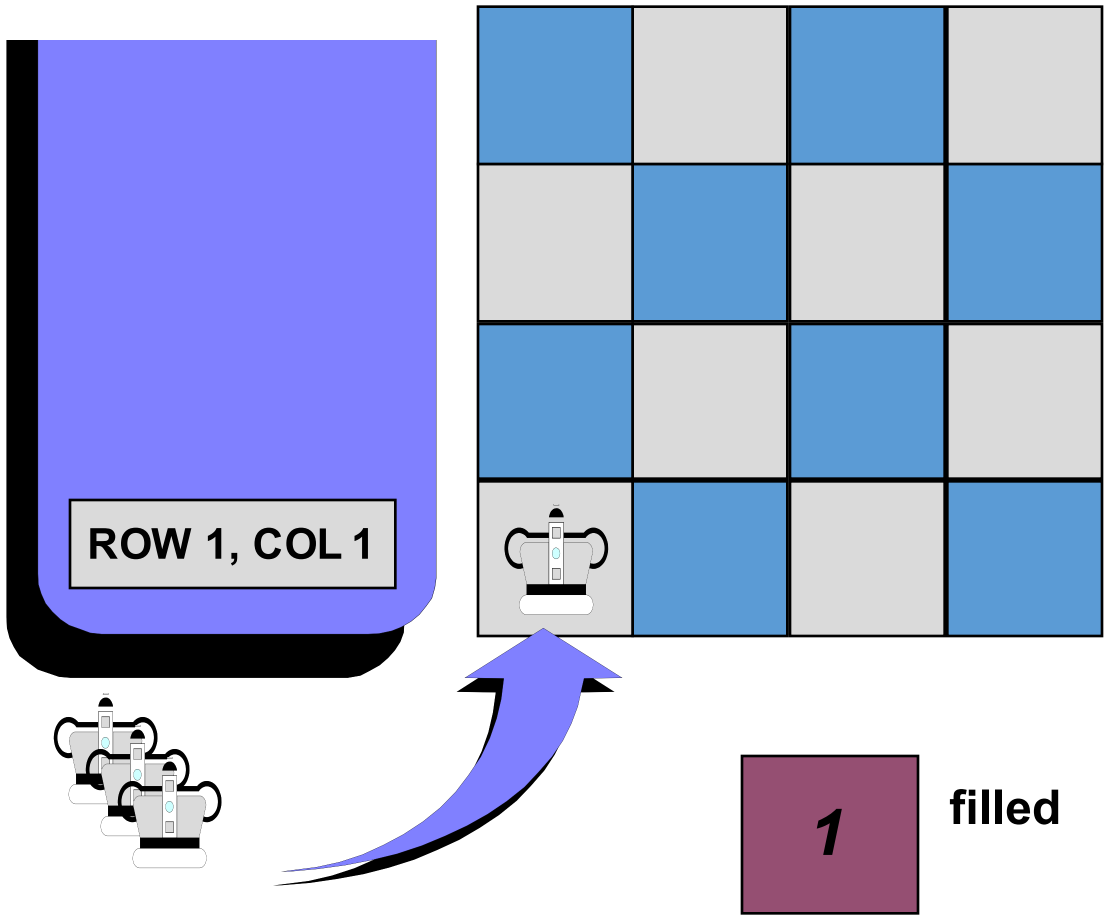
How the program works

Each time the program decides to place a queen on the board, the position of the new queen is stored in a record which is placed in the stack.



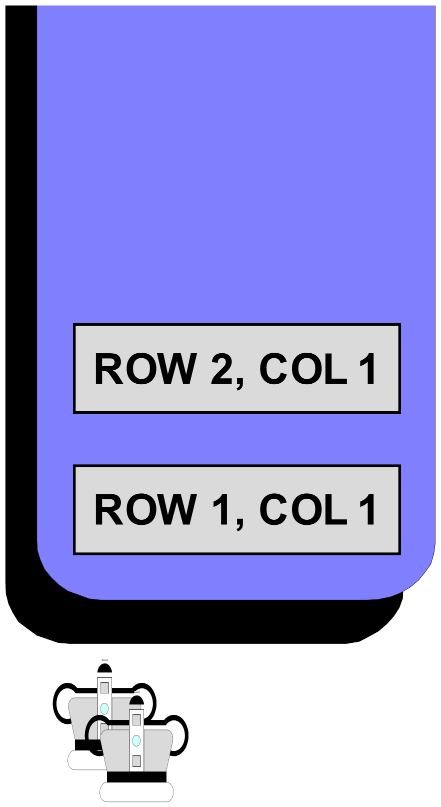
How the program works

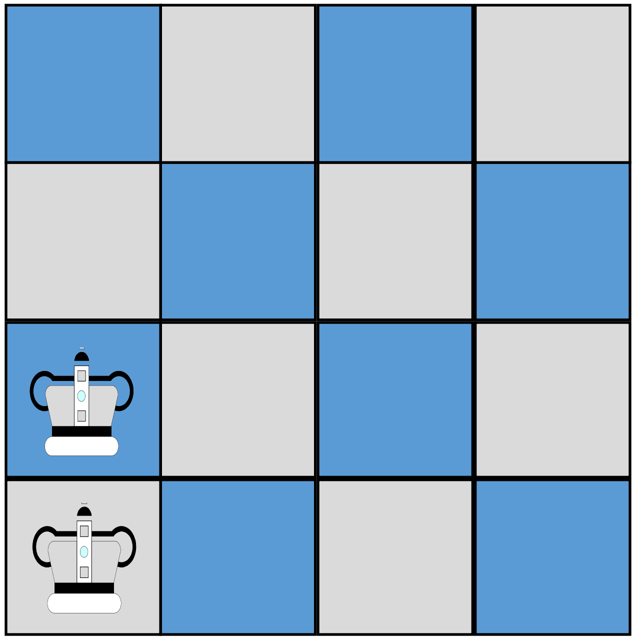
We also have an integer variable to keep track of how many rows have been filled so far.



How the program works

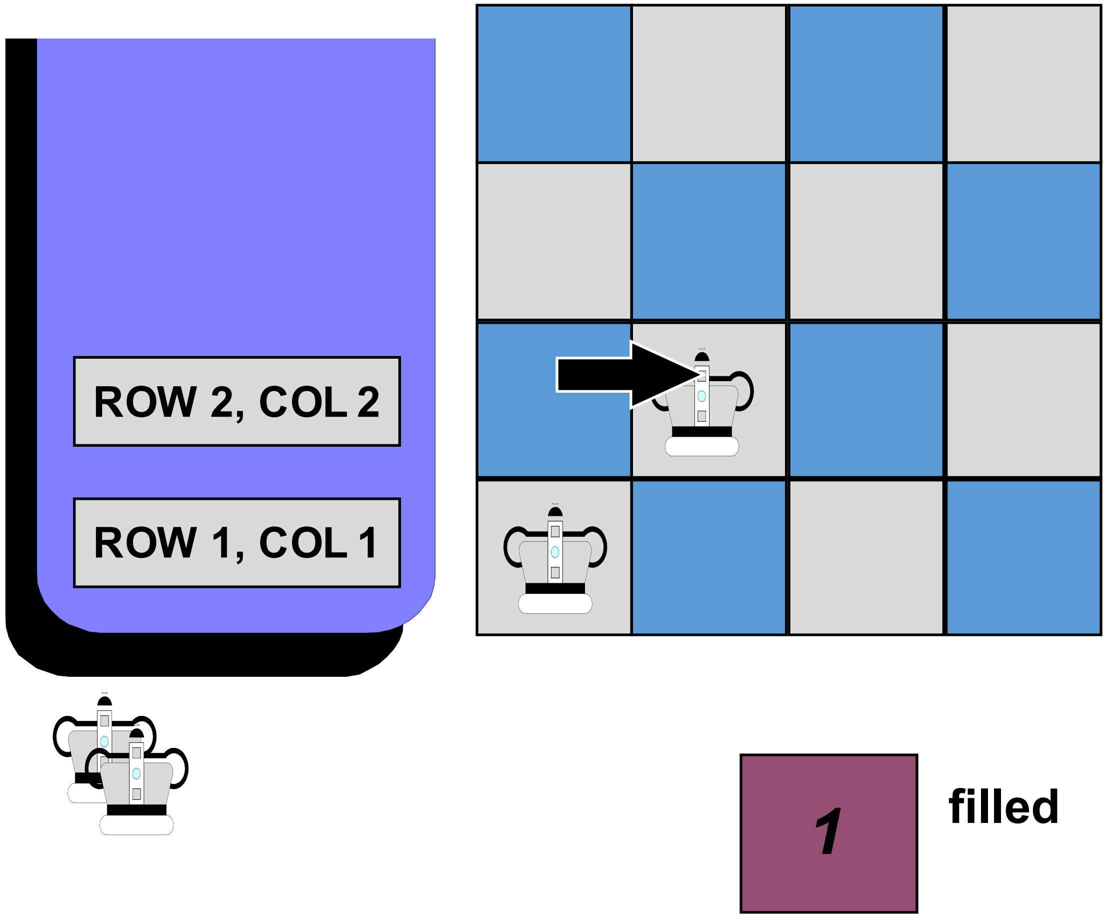
Each time we try to place a new queen in the next row, we start by placing the queen in the first column...





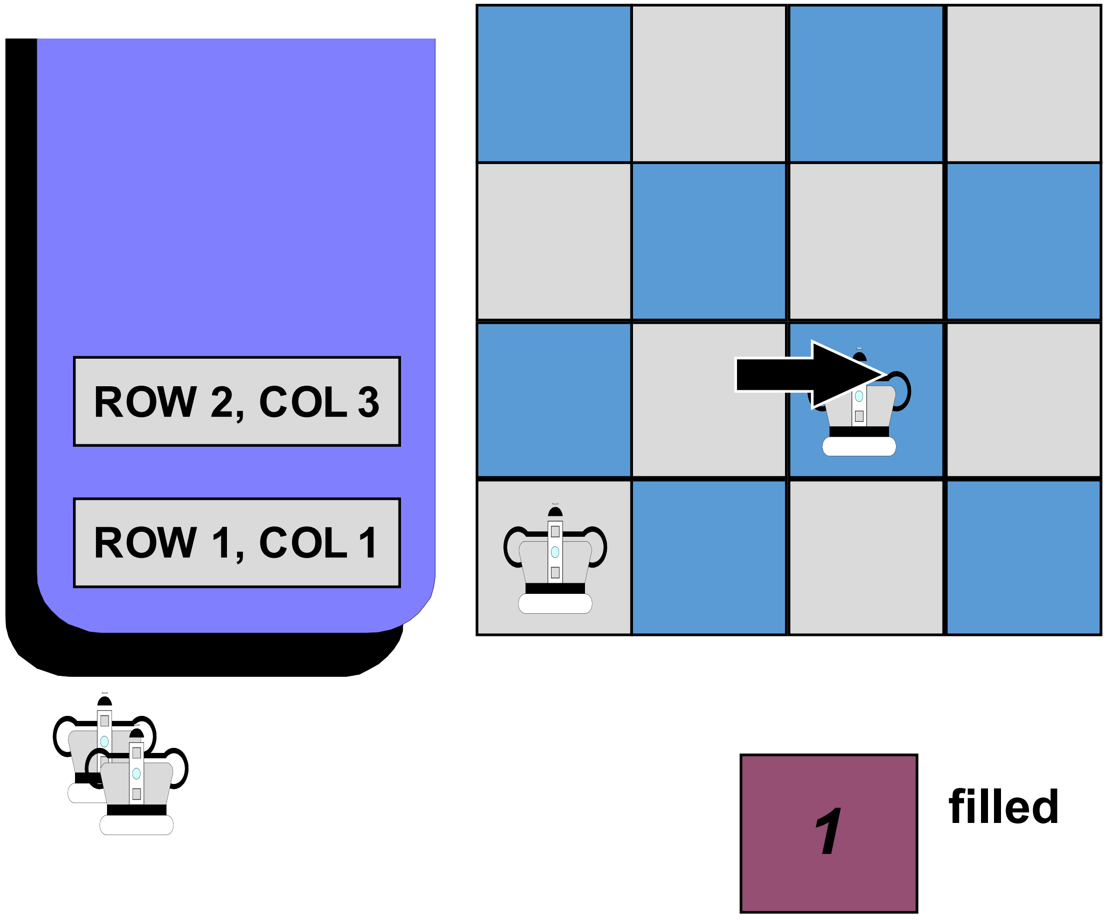
How the program works

...if there is a conflict with another queen, then we shift the new queen to the next column.



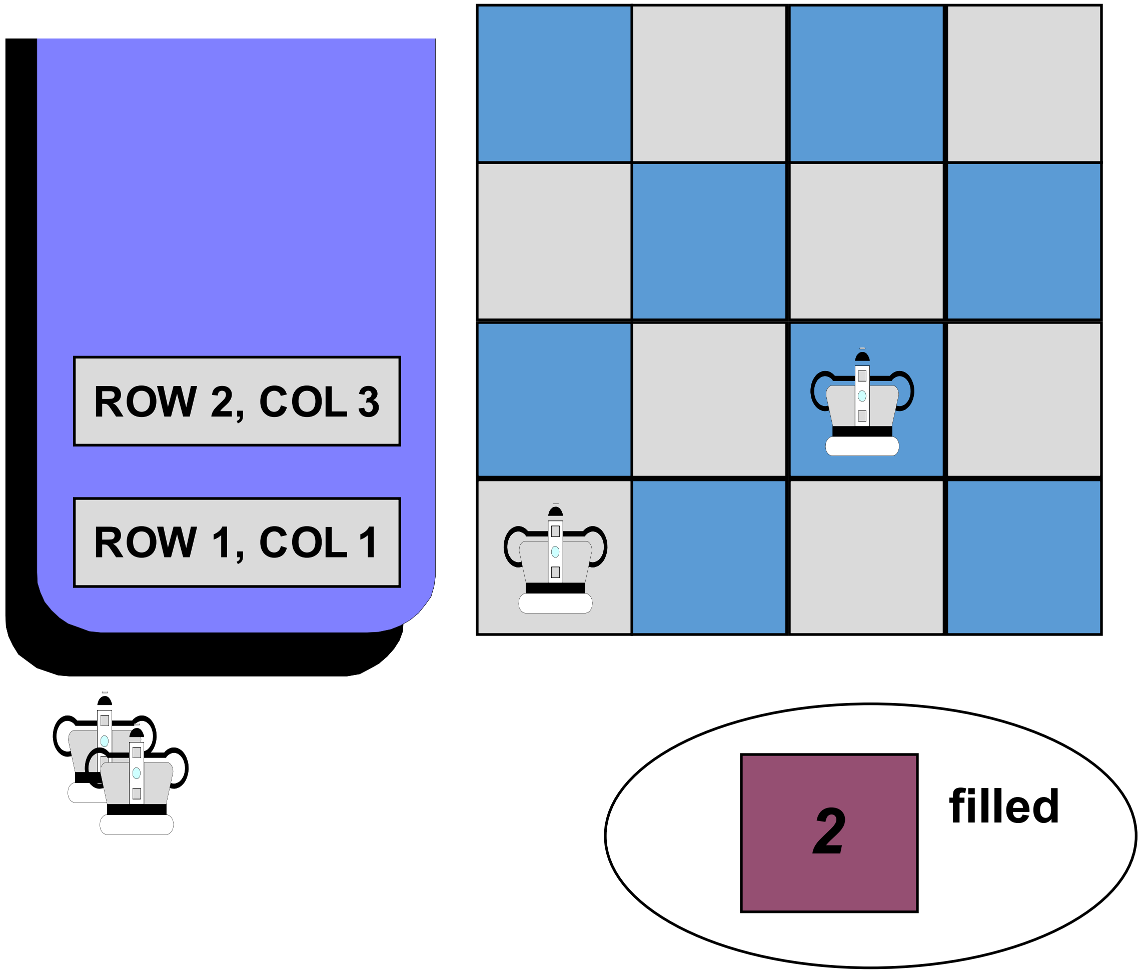
How the program works

If another conflict occurs, the queen is shifted rightward again.



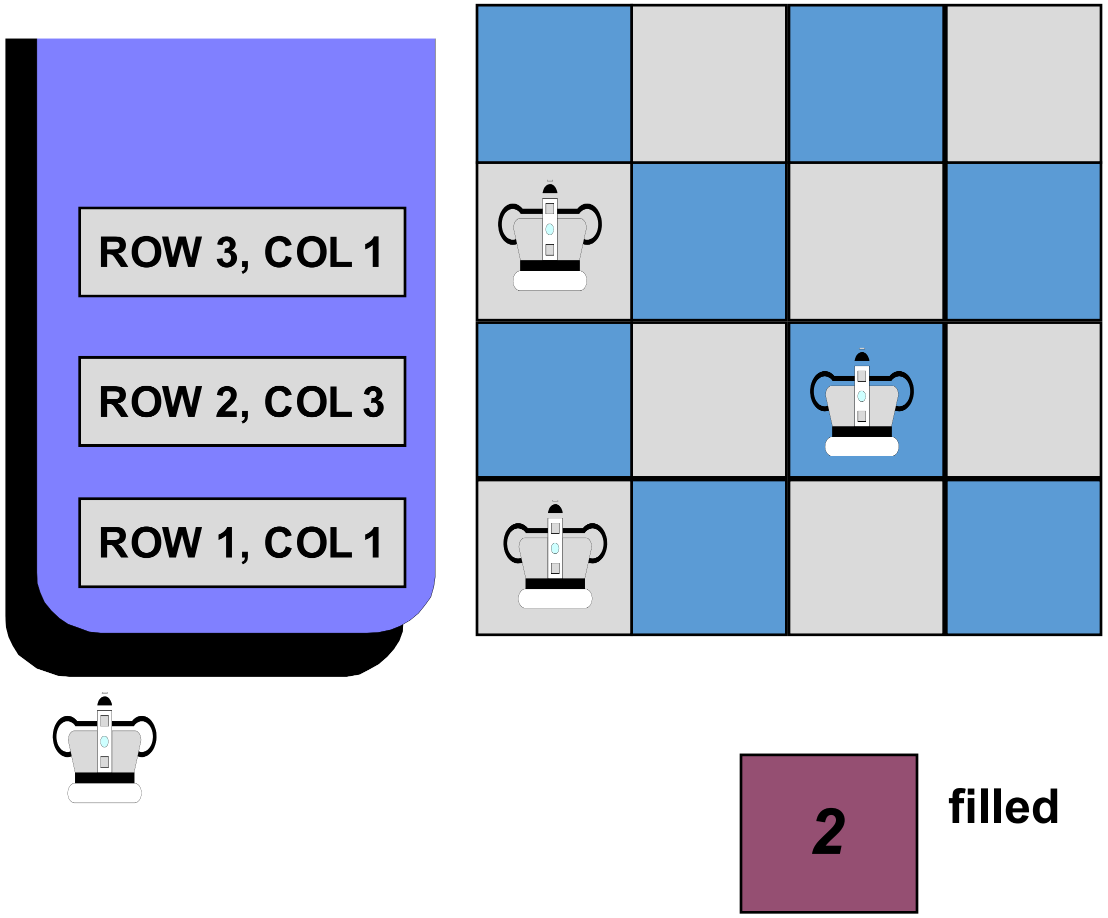
How the program works

When there are no conflicts, we stop and add one to the value of filled.



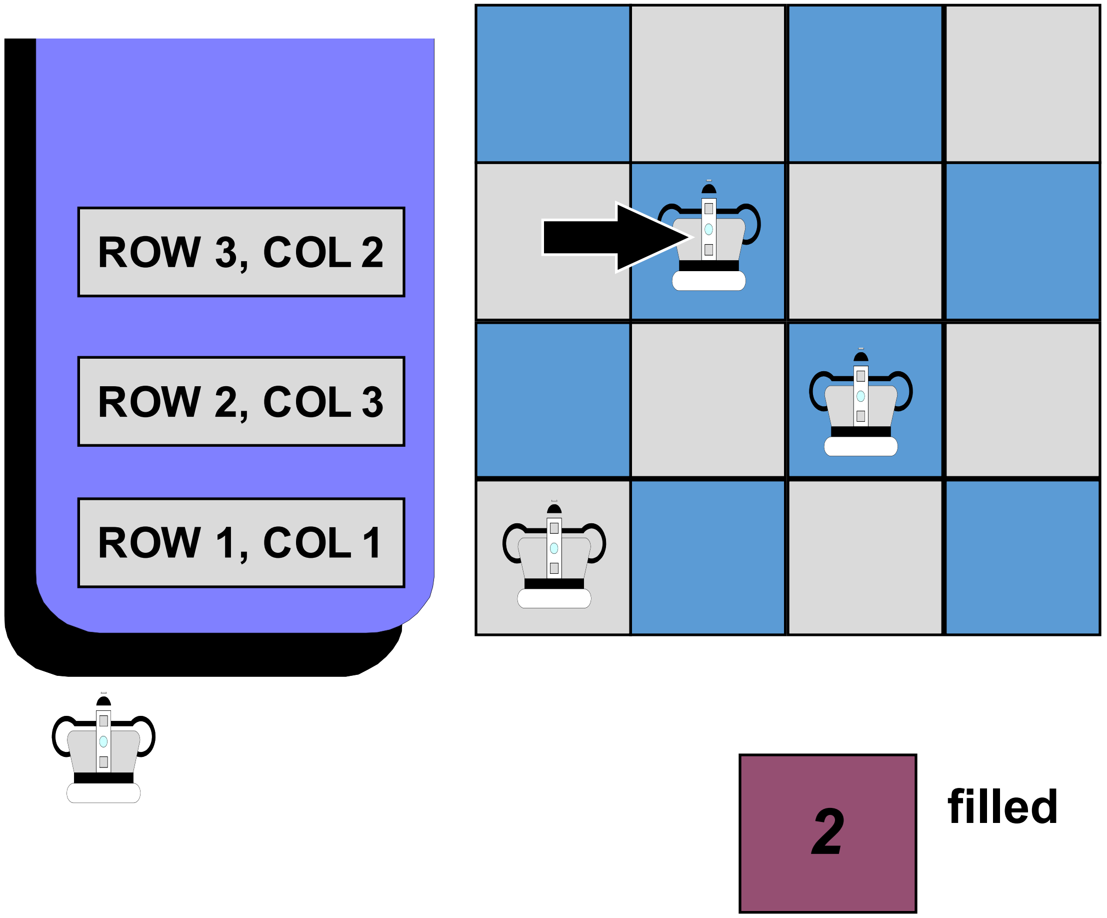
How the program works

Let's look at the third row. The first position we try has a conflict...



How the program works

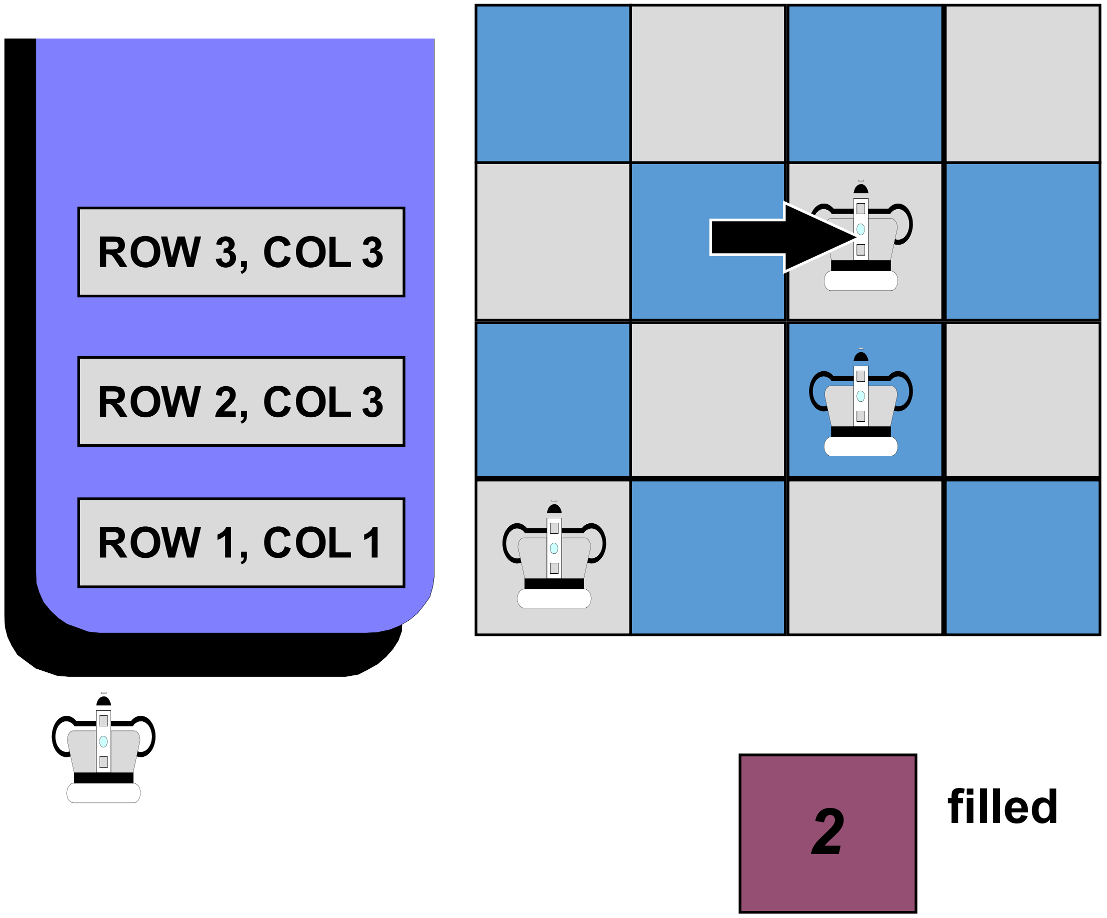
…so we shift to column 2. But another conflict arises…



How the program works

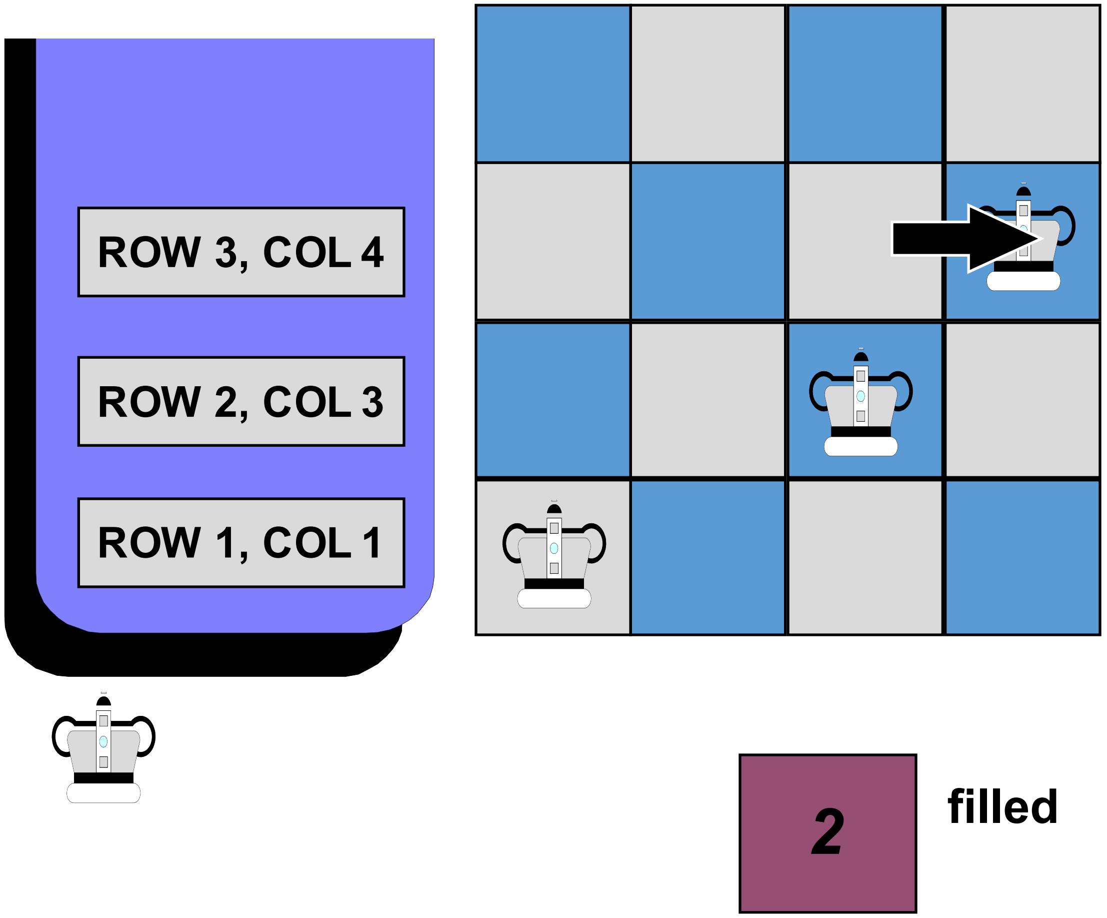
...and we shift to the third column.

Yet another conflict arises...



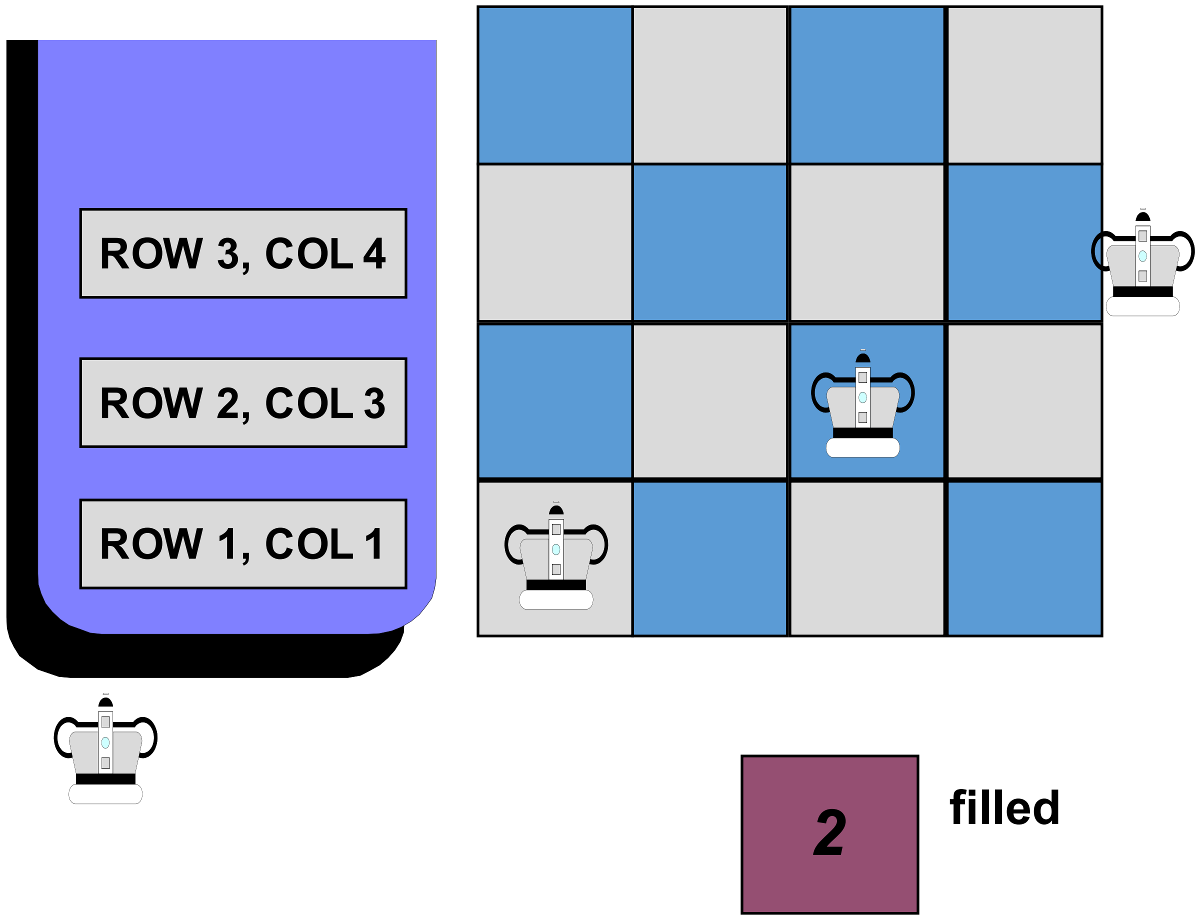
How the program works

...and we shift to column 4. There's still a conflict in column 4, so we try to shift rightward again...



How the program works

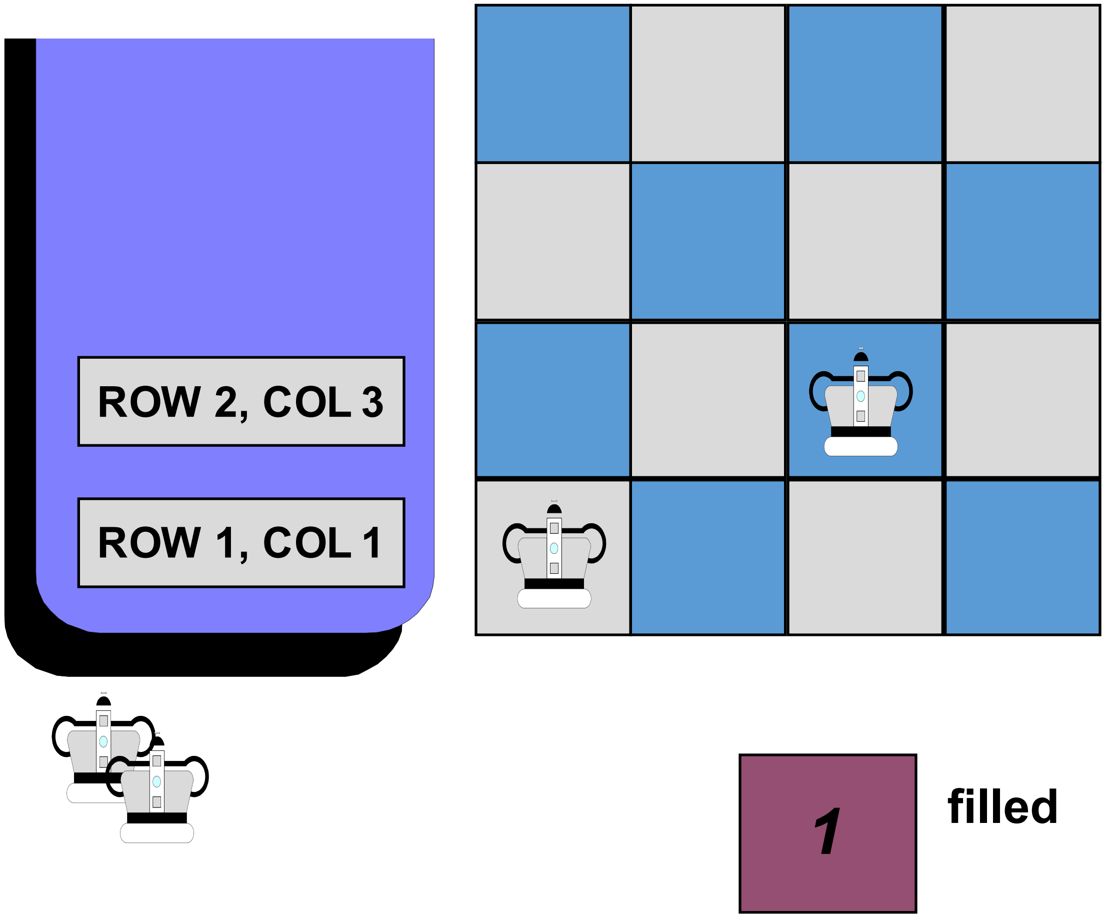
...but there's nowhere else to go.



How the program works

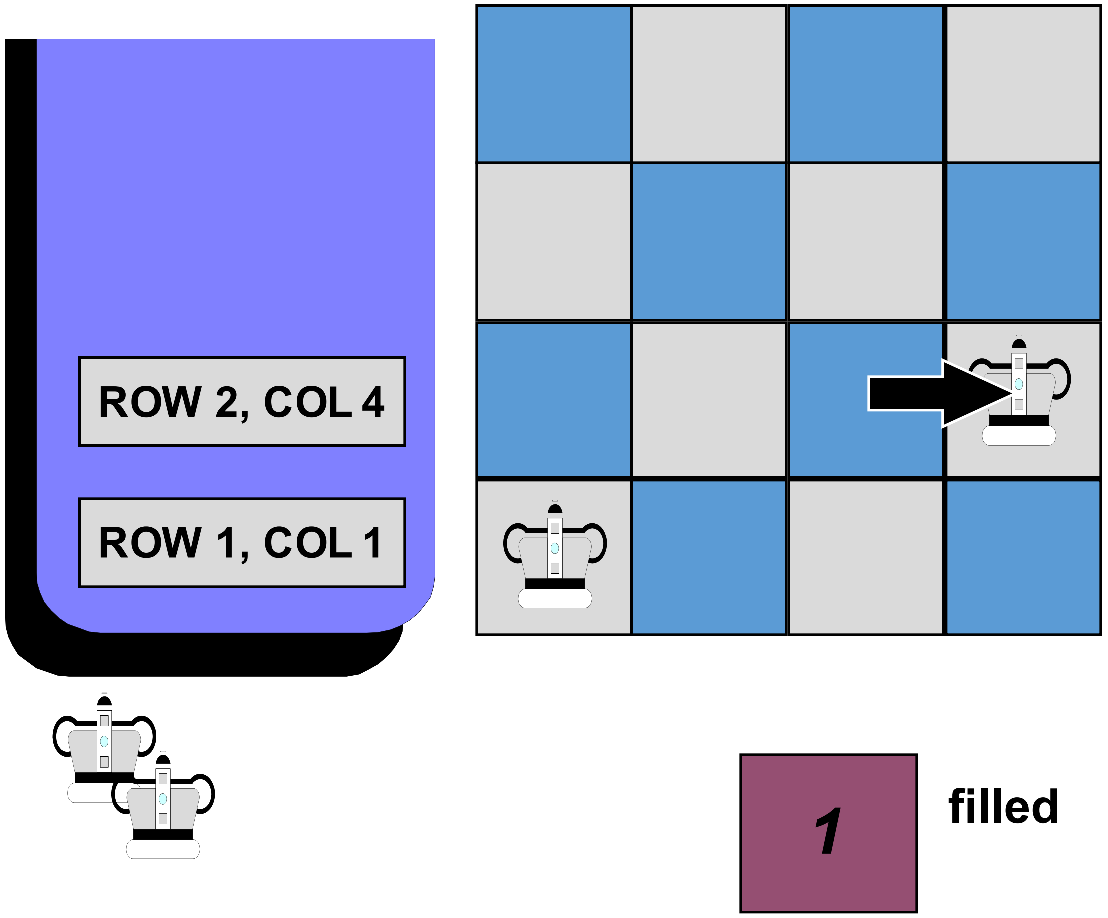
When we run out of room in a row:

- pop the stack,
- reduce filled by 1
- and continue working on the previous row.



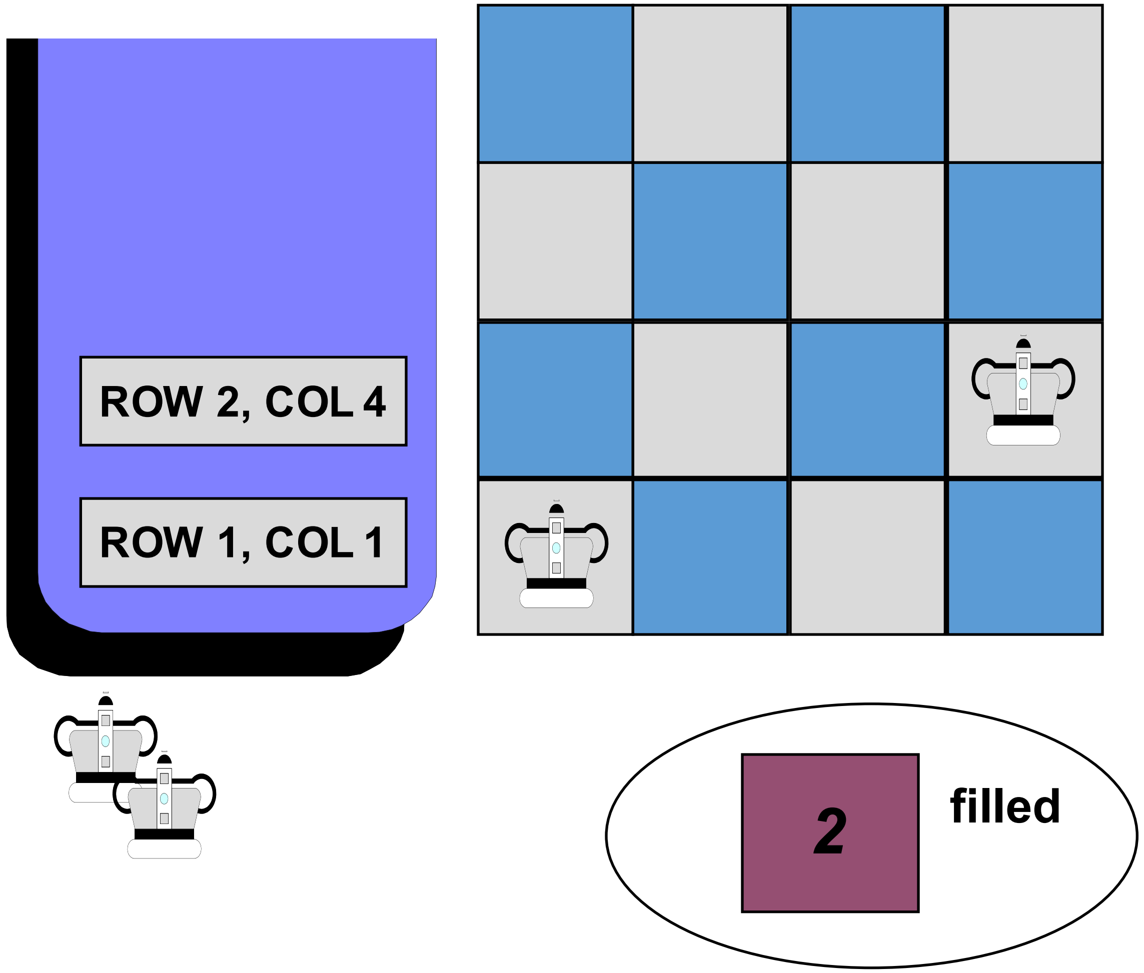
How the program works

Now we continue working on row 2, shifting the queen to the right.



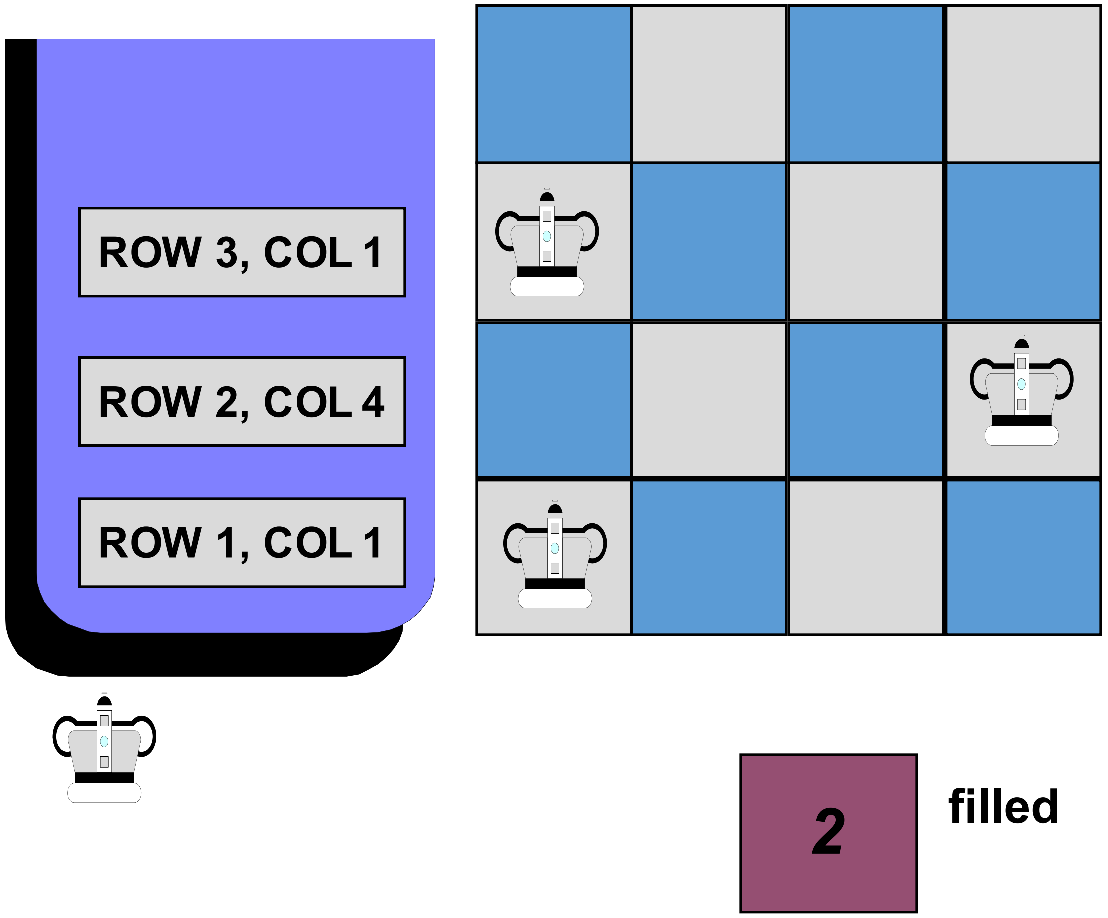
How the program works

This position has no conflicts, so we can increase filled by 1, and move to row 3.



How the program works

In row 3, we start again at the first column.



Pseudocode for N-Queens

❶ Initialize a stack where we can keep track of our decisions.

❷ Place the first queen, pushing its position onto the stack and setting filled to 0.

❸ repeat these steps

- if there are no conflicts with the queens...
- else if there is a conflict and there is room to shift the current queen rightward...
- else if there is a conflict and there is no room to shift the current queen rightward...

Pseudocode for N-Queens

❸ repeat these steps

- if there are no conflicts with the queens...

Increase **filled** by 1. If **filled** is now **N**, then the algorithm is done. Otherwise, move to the next row and place a queen in the first column.

Pseudocode for N-Queens

❸ repeat these steps

- if there are no conflicts with the queens...
- else if there is a conflict and there is room to shift the current queen rightward...

Move the current queen rightward, adjusting the record on top of the stack to indicate the new position.

Pseudocode for N-Queens

❸ repeat these steps

- if there are no conflicts with the queens...
- else if there is a conflict and there is room to shift the current queen rightward...
- else if there is a conflict and there is no room to shift the current queen rightward...

Backtrack!
Keep popping the stack, and reducing filled by 1, until you reach a row where the queen can be shifted rightward. Shift this queen right.

Pseudocode for N-Queens

❸ repeat these steps

- if there are no conflicts with the queens...
- else if there is a conflict and there is room to shift the current queen rightward...
- else if there is a conflict and there is no room to shift the current queen rightward...

Backtrack!
Keep popping the stack, and reducing filled by 1, until you reach a row where the queen can be shifted rightward. Shift this queen right.

Summary of stack for backtracking

- Stacks have many applications.
- The application which we have shown is called **backtracking**.
- The key to backtracking: Each choice is recorded in a stack.
- When you run out of choices for the current decision, you pop the stack, and continue trying different choices for the previous decision.

Summary and Homework

- Stacks (Read Chapter 7)
  - Self-Test: 1-5, 13-18
- Queues (Read Sections 8.1 – 8.3)
  - Self-Test: 1-5, 10,18-21
- Priority Queues (Read Section 8.4)
  - Self-Test: 25-27
- References Return Values (Read Section 8.5 and p. 302 in Chapter 6)
  - Self-Test: class note