stacks, queues, lists

Most figures are taken from the book
Introduction to Algorithms, T.H. Cormen et al. (eds.), third edition, MIT Press, 2009

abstract data types (ADT)

- An ADT is a set of objects (of the same type, e.g., integers, real, boolean) with a set of operations
- Nowhere in an ADT definition is there any mention of how the set of operations are implemented
- The C++ class allows for ADT implementation with appropriate hiding of the implementation details; any part of the program that needs to perform an operation on the ADT can do so by calling the appropriate method.
- Conceptually, a data structure is an ADT

Lists

- List ADT
  - A sequence of objects A_0, A_2, …, A_{N-1}
  - Size of list = N; empty list is of size 0
  - Ai succeeds Ai-1, Ai-1 preceeds Ai
  - Common list operations: *printList*, *makeEmpty*, *find*, *insert*, *remove*, *findKth*, *next*, *findPrevious*, etc.
  - 34, 12, 52, 16, 12: find(52) => 2; insert(x,2) => 34, 12, x, 52, 16, 12; remove(52) => 34, 12, x, 16, 12

- Simple array implementation of lists
  - Vector class internally stores the array
  - Although fixed size initially, vector class allows the array to grow by doubling its size
  - Some operations can be expensive, e.g., *insert*/*remove*:

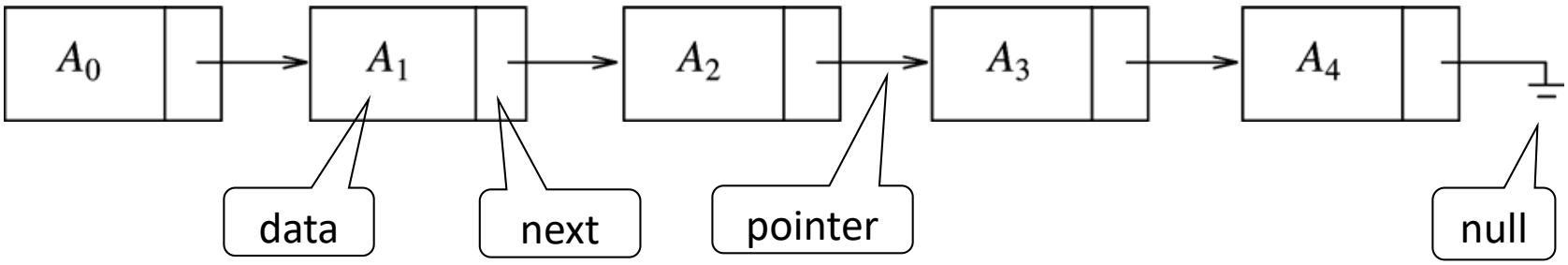
    insert in i=0 => shift all elements to the right

    remove from i=0=> shift all elements to the left

Linked lists

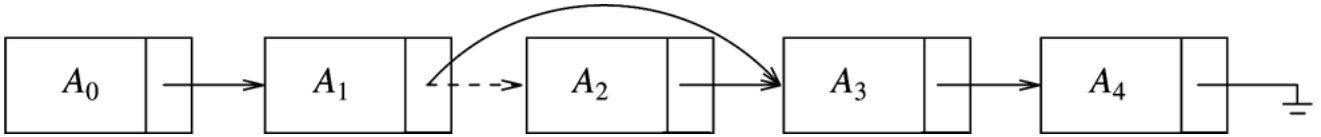
- A structure where objects are in linear order (as in arrays)
- To avoid the cost of *insert* and *remove*, create a set of nodes; each node contains the element and a link to a node containing its successor

  Unlike arrays, the order is specified by pointers
- Nodes are not necessarily contiguous memory cells (as in arrays)
- The link of each “internal” node is the *next* link, the link of the last node points to *nullptr*

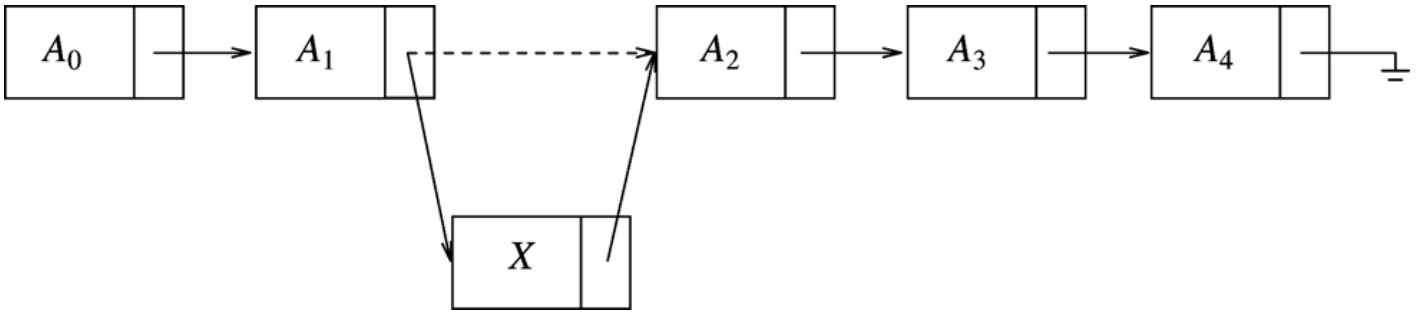


why linked lists?

- Operations such as find(x) are in linear time (as in arrays)
- findKth(i) is more expensive (not constant time O(1) as in arrays), due to traversing through the list
- However:
  - *remove* can be executed in O(1): one *next* pointer change



  - insert can be executed in a similar way: take a new node (*new*) and execute a two *next* pointer changes



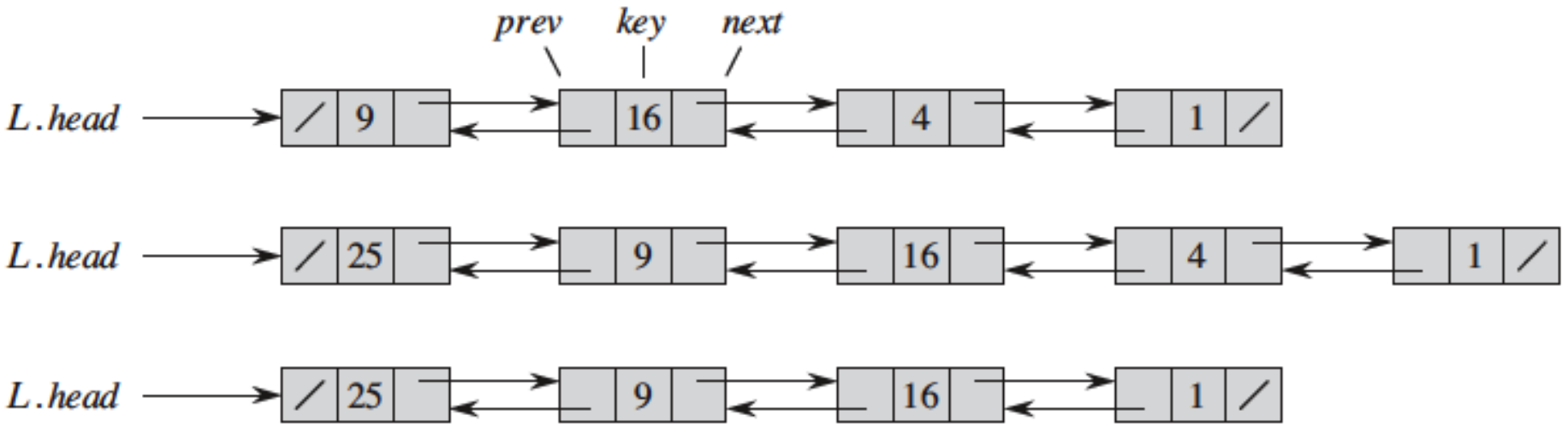
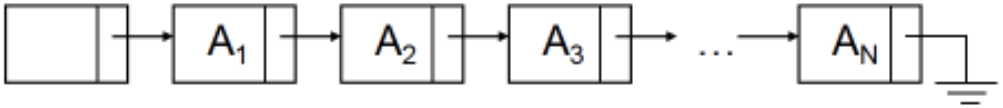


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

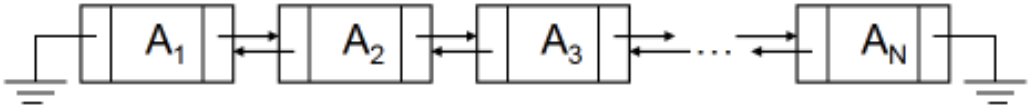
more details

- Header
  - A dummy node pointing to the first node of the list (existing in empty lists)

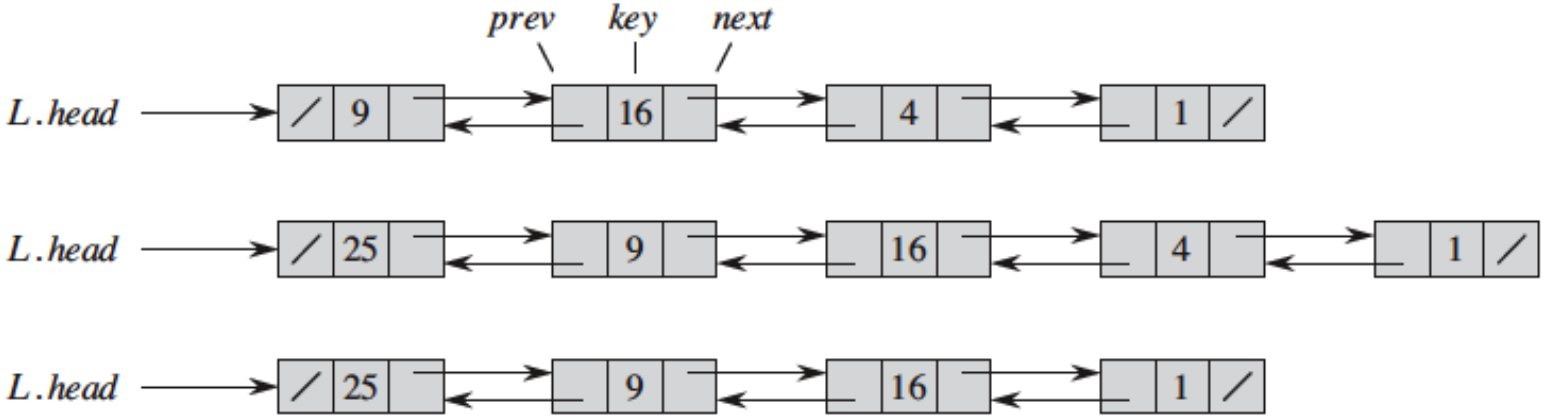


  - Remove from beginning: no re-linking, just change the pointer to the beginning of the list
  - Remove from the end: the next-to-last node becomes the new end; a bit tricky: once we find the last node we remove and must go to the new end (*think of it as a "path": which path takes you to the desired node?*)

- Doubly linked list
  - requires more memory
  - Simplifies deletion (find previous)



- Implementation details: Textbook, pp. 80-102.




```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

find the first element with key k in list L by linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

put x in the front (head)

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

Remove x from a linked list L. It must be given a pointer to x, and it then “splices” x out of the list by updating pointers.

sentinel

- A sentinel is a dummy object that allows us to simplify boundary conditions.
- Suppose that we provide list L with an object L.nil that represents NIL but has all the attributes of the other objects in the list.

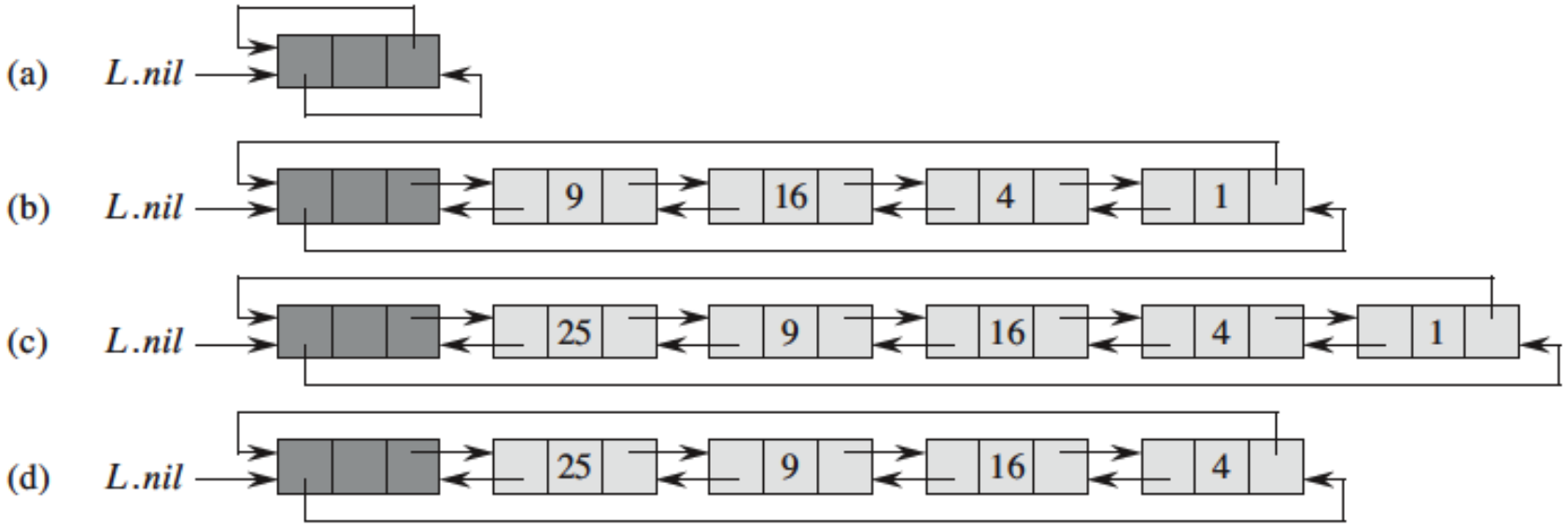
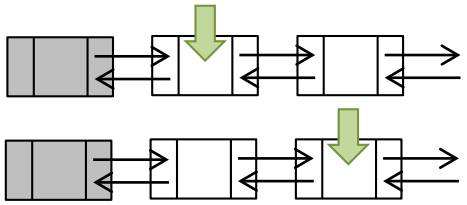


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

search, delete, insert

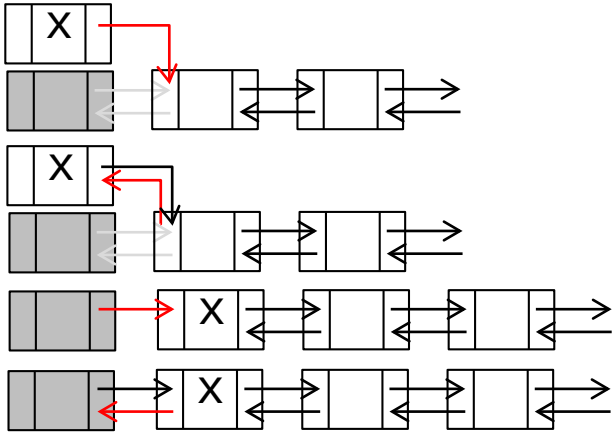
```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```



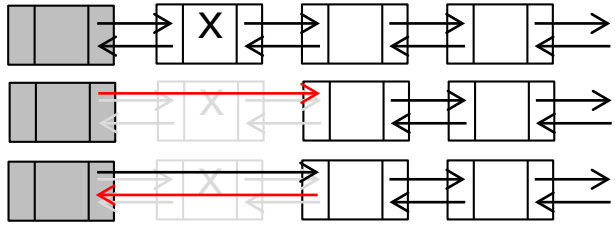
```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```



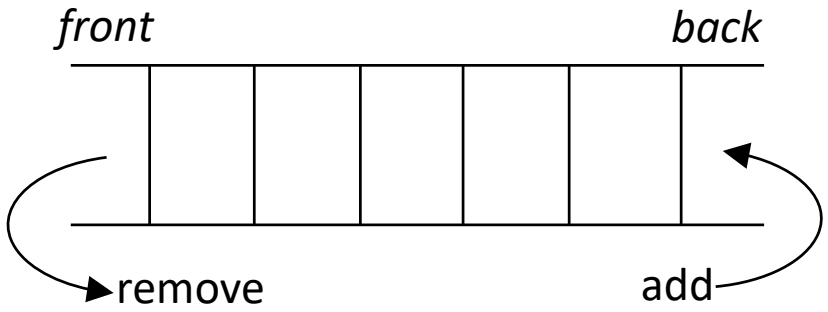

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```



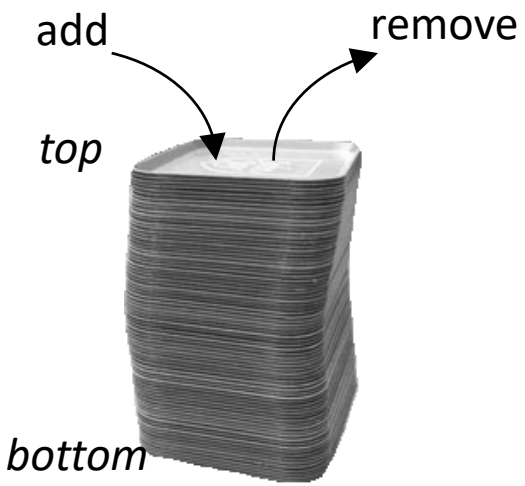
Queues and stacks

- Dynamic sets where elements are deleted in a predetermined order:



Queues: FIFO – delete the longest remaining (like a line of customers)

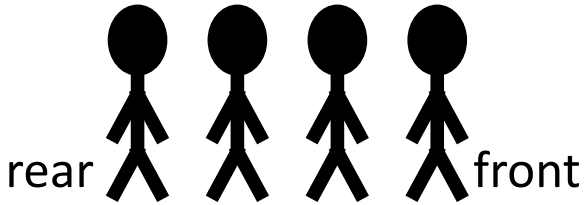
- Are appropriate for many real-world situations
  Example: A line to buy a movie ticket
- Have applications in computer science
  Example: A request to print a document
- A simulation
  A study to see how to reduce the wait involved in an application



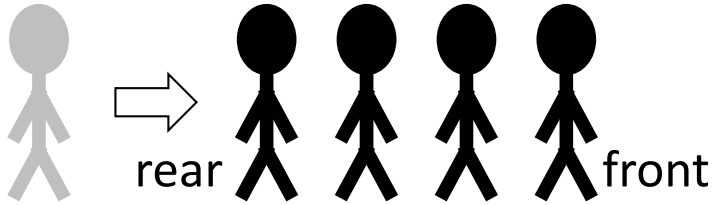
Stacks: a list with the restriction that insertion & deletion can be performed only at the end (or top) of the list; LIFO – delete the most recently inserted (like a physical stack)

queue class

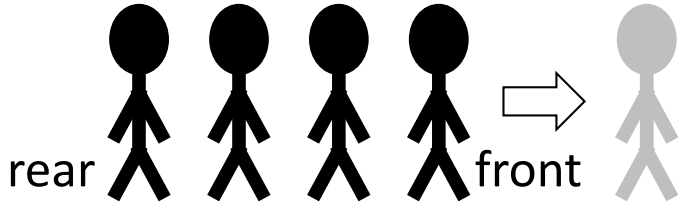
- A queue is like a line of people waiting



- New people must enter the queue at the rear. The C++ queue class calls this *push*, however the operation is called **enqueue**

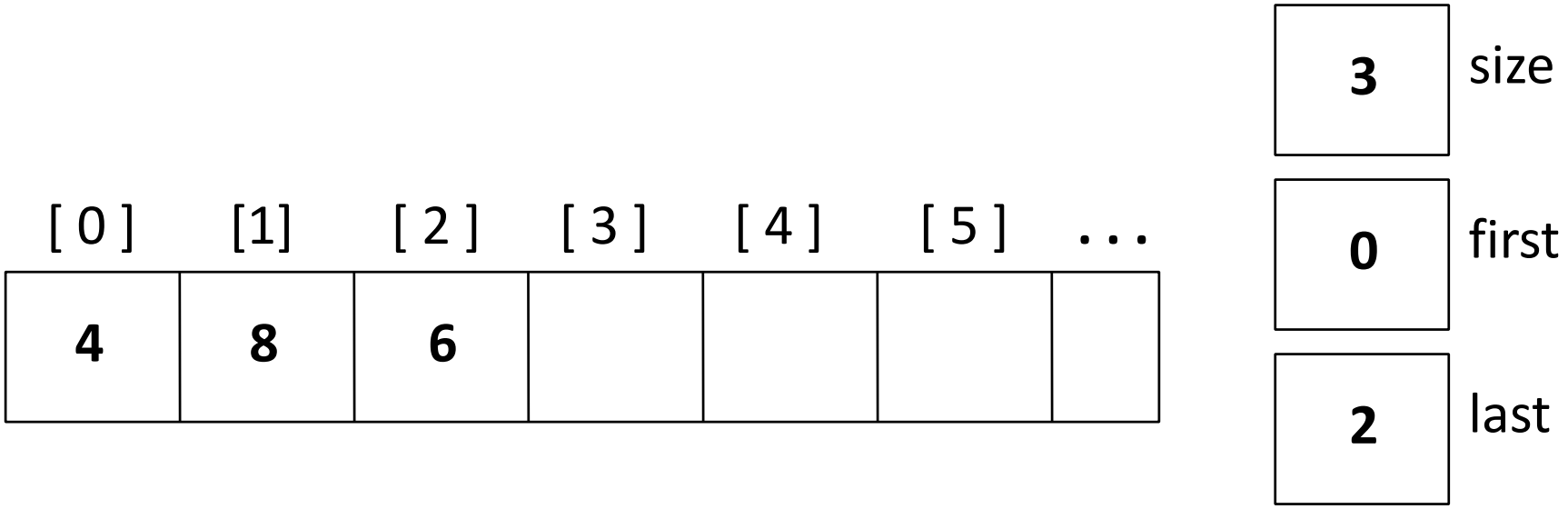


- When an item is taken from the queue, it always comes from the front. The C++ queue class call this *pop*, however the operation is called **dequeue**



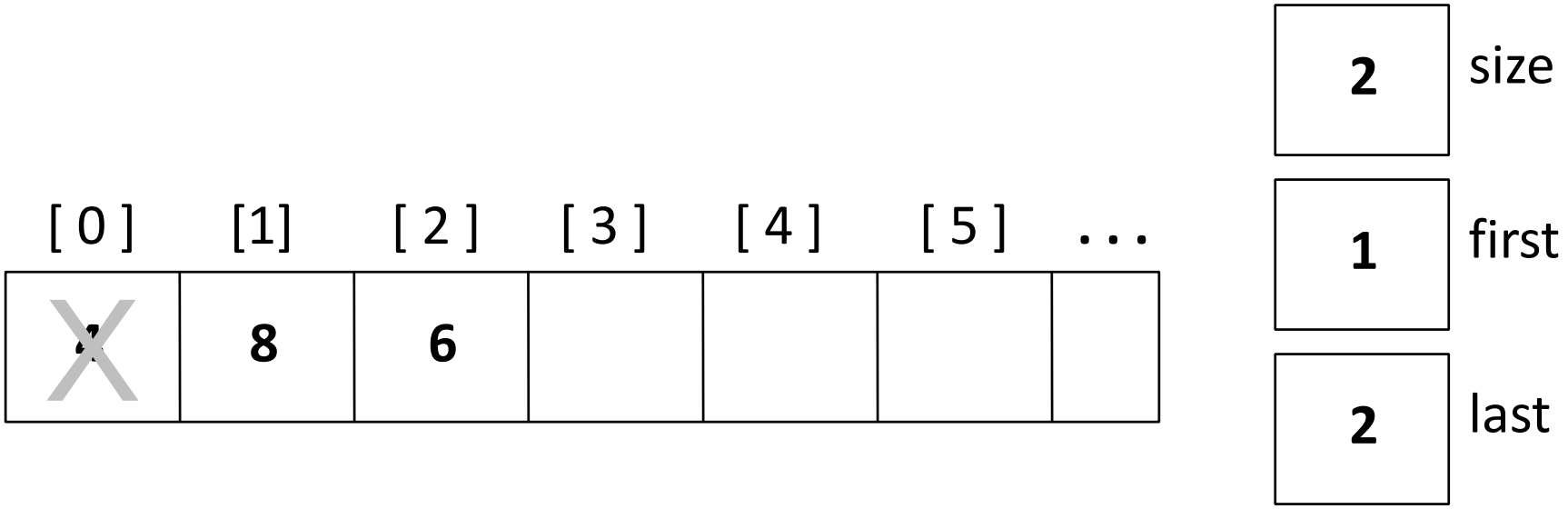
example

- Consider a queue containing the integers 4 (front), 8, and 6 (rear)
- Keep track of the number of items in the queue, index of the first element, and the index of the last element



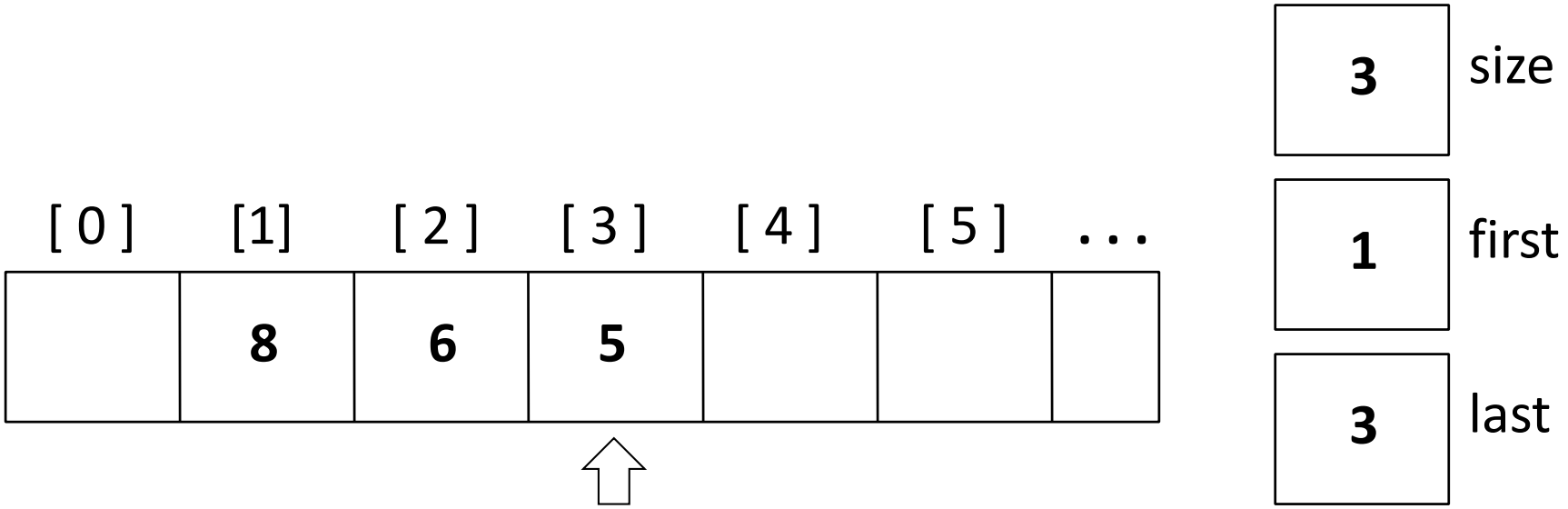
dequeue

- When an element leaves the queue, the size decreases and the first element changes



enqueue

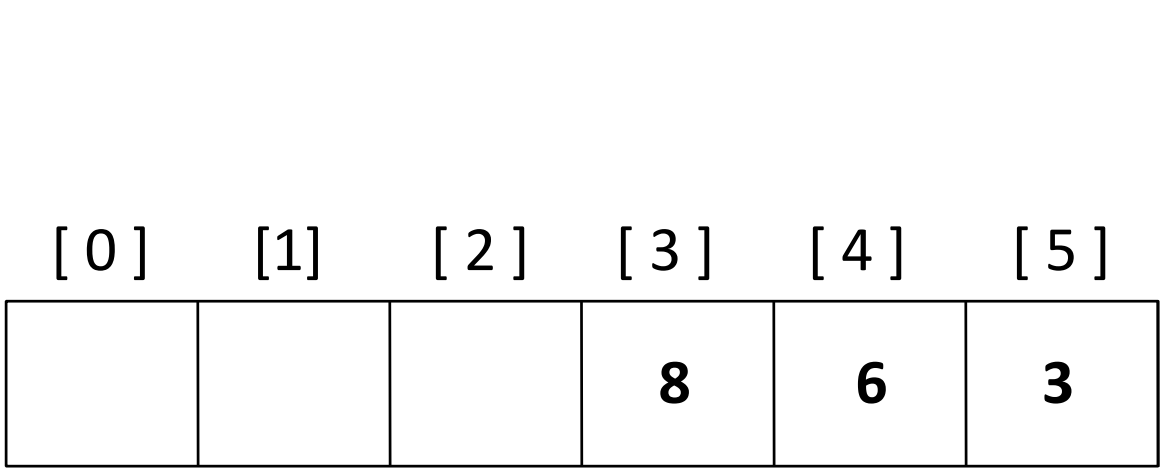
- When an element enters the queue, the size increases and last element changes

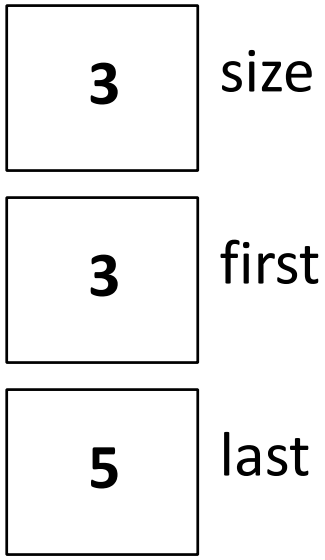


- For a fixed size array, a new item may enter if the current size of the queue is less than the size of the array
- For a dynamic array, we could increase the size of the array when the queue grows beyond the current array size

special case: the end

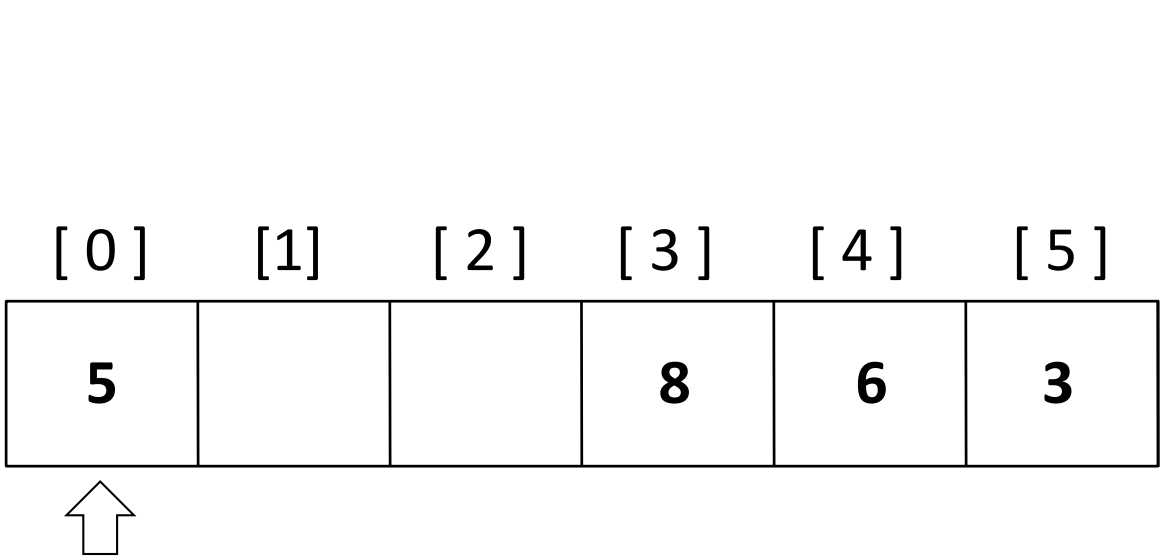
- Suppose we want to add an new element to a queue where the last index is 5

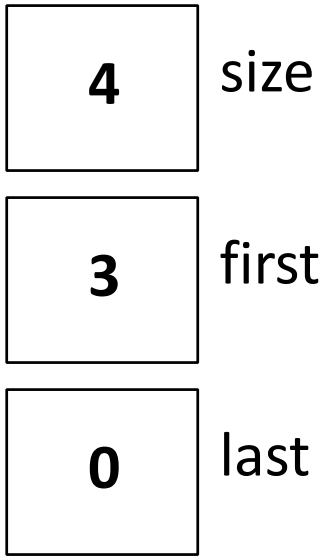




enqueue

- The new element goes at the front of the array if the location is not used



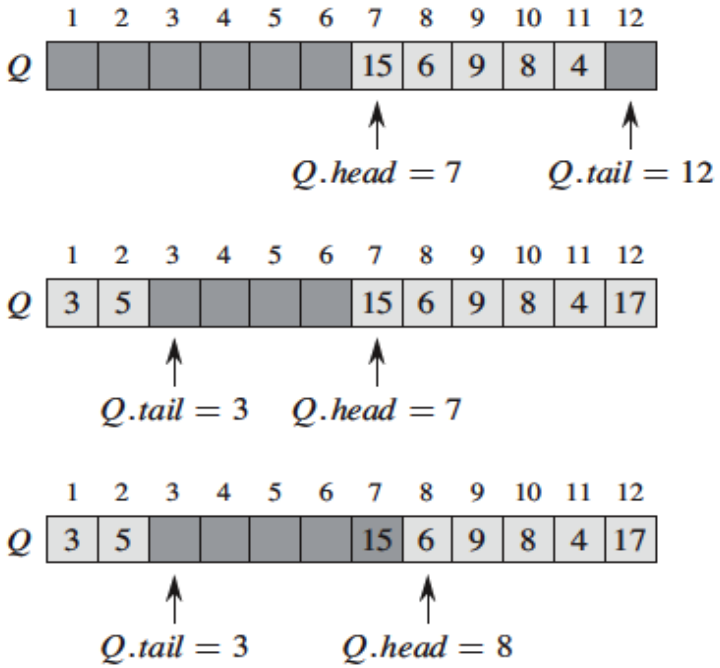


array implementation

- ENQUEUE=insert, DEQUEUE=delete
- Q[1…n]; Q.head = head of queue; Q.tail = tail of queue
- Q.head = Q.tail => queue empty
- dequeue(empty_queue) => queue underflow (error)
- Queue.head = Q.tail + 1 (circular fashion) => queue full

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```



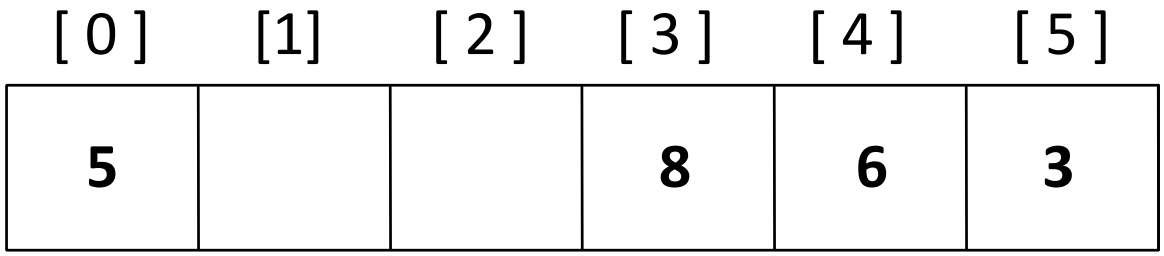
A queue implemented using an array Q[1…12]. Queue elements appear only in the lightly shaded positions.

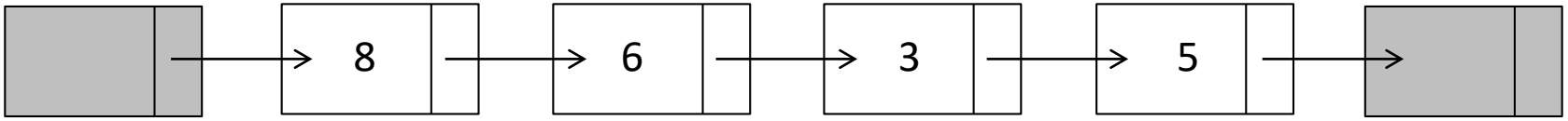
Top: the queue has 5 elements, in locations Q[7… 11]

Middle: the configuration of the queue after the calls ENQUEUE(Q,17), ENQUEUE(Q,3), and ENQUEUE(Q,5).

Bottom: The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

linked list implementation



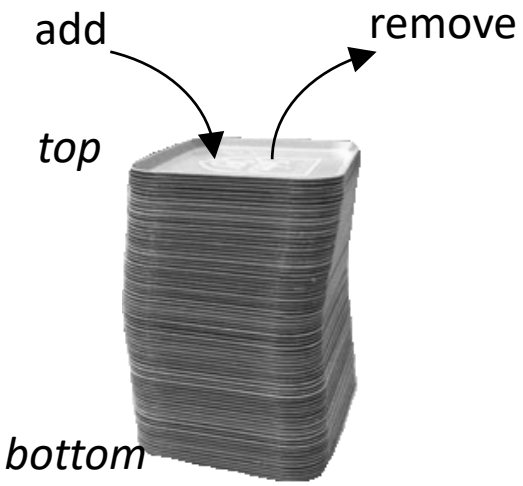


Does it matter which end of a singly-linked list we use for the front of the queue?

We could put the front of the queue at the end of the linked list, but it would be hard to dequeue. Why? (think *path*)

stacks

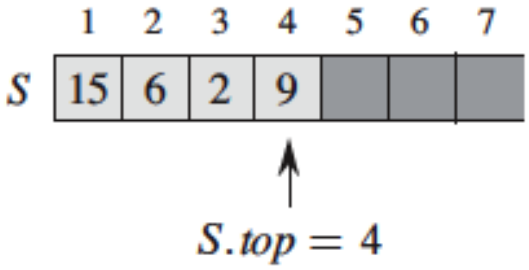
- a list with the restriction that insertion & deletion can be performed only at the end (or top) of the list; LIFO – delete the most recently inserted (like a physical stack)

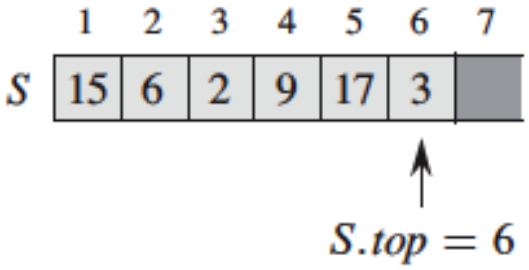


stacks

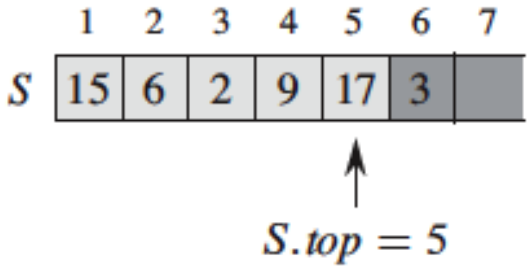
- PUSH=insert, POP=delete
- S[1…S.top], where S[1] is at the bottom
- S.top=0 => stack is empty
- pop(empty_stack) => stack underflow (error)
- S.top > n (max size) => stack overflow (error)

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```





Stack S after the calls PUSH(S,17) and PUSH(S,3)



Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed.

example

http://cpp.datastructures.net

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

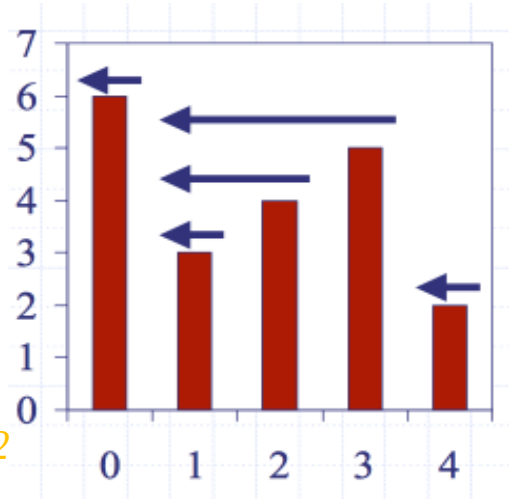
The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock is less than or equal to the price on the given day.

<u>More general problem statement:</u>

Given an array X, the span S[i] of X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] and such that X[j] ≤ X[i]

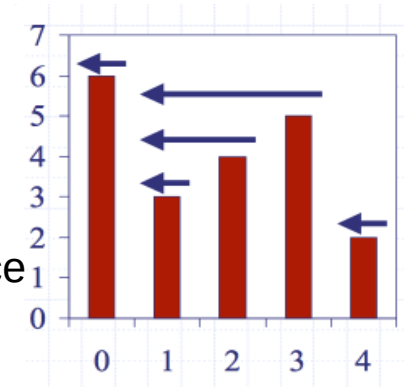
X = [6 3 4 5 2] ⟹ S = [1 1 2 3 1]

```
Algorithm spans1(X, n)
    Input array X of n integers
    Output array S of spans of X                    #
    S ← new array of n integers                     n
    for i ← 0 to n − 1 do                           n
        s ← 1                                       n
        while s ≤ i ∧ X[i − s] ≤ X[i]               1 + 2 + …+ (n − 1)   =n(n-1)/2
            s ← s + 1                               1 + 2 + …+ (n − 1)
        S[i] ← s                                    n
    return S                                        1
```



$O(n^2)$

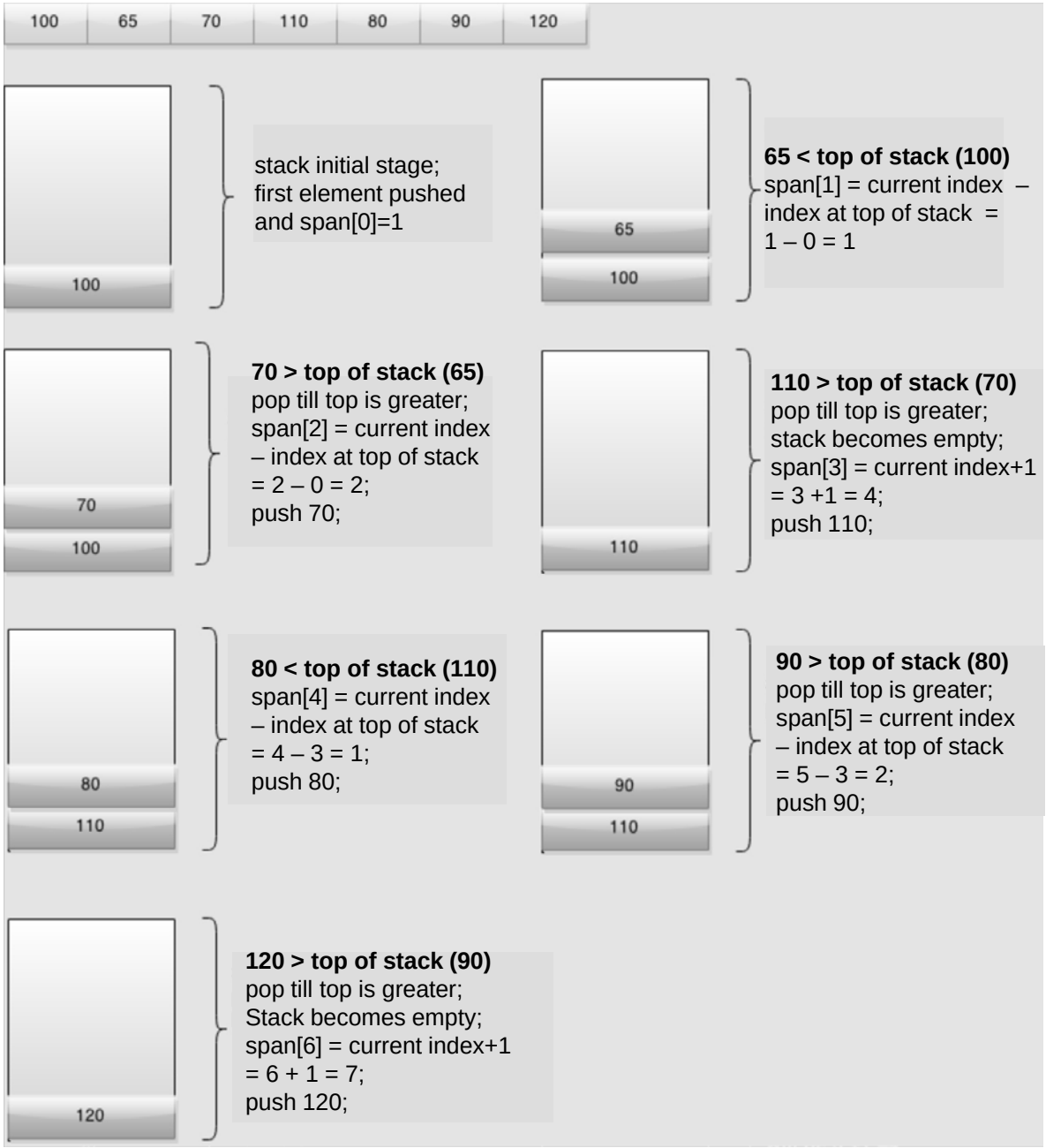
example (cnt'd)



- We see that S[i] on "day i" (array i-th element) can be easily computed if we know the closest day preceding i, such that the price on that day is greater than the price on day i. If such a day exists, let's call it h(i), otherwise, we define h(i) = -1.
- The span is now computed as S[i] = i – h(i).

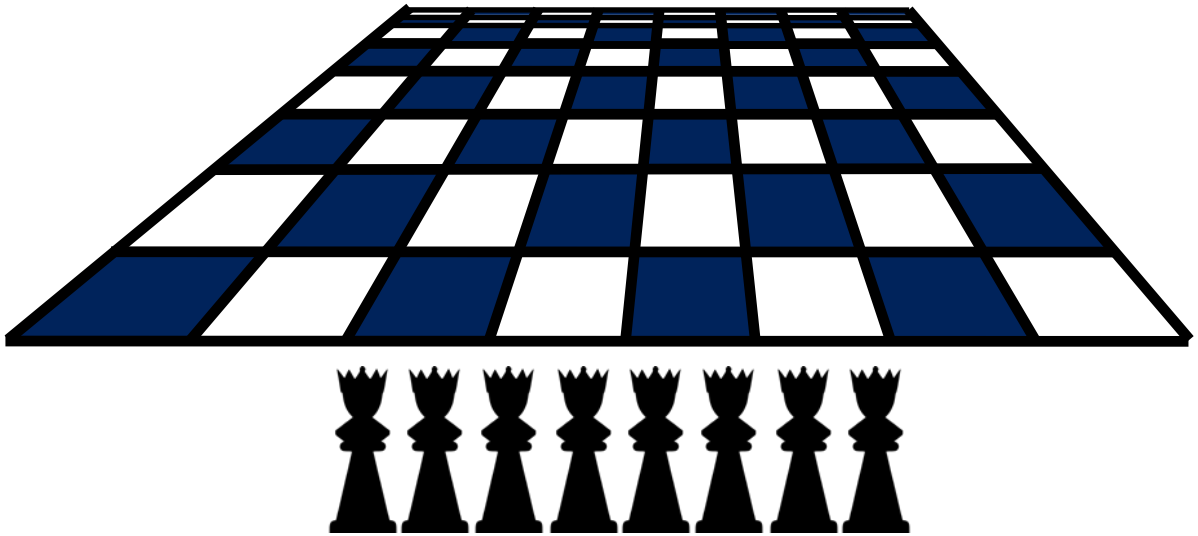
- In other words: when was the last day I saw a price greater than today's price?

- We keep in a stack the indices of the elements/days "visible when looking back"
- We scan the array from left to right
  - Let i be the current index
  - We *pop* (delete) indices from the stack until we find index j such that X[i] < X[j]
  - We set S[i]←i−j
  - We *push* (insert) x onto the stack (the value of day i)


100 65 70 110 80 90 120
100
stack initial stage;
first element pushed
and span[0]=1
65
100
65 < top of stack (100)
span[1] = current index –
index at top of stack =
1 – 0 = 1
70
100
70 > top of stack (65)
pop till top is greater;
span[2] = current index
– index at top of stack
= 2 – 0 = 2;
push 70;
110
110 > top of stack (70)
pop till top is greater;
stack becomes empty;
span[3] = current index+1
= 3 +1 = 4;
push 110;
80
110
80 < top of stack (110)
span[4] = current index
– index at top of stack
= 4 – 3 = 1;
push 80;
90
110
90 > top of stack (80)
pop till top is greater;
span[5] = current index
– index at top of stack
= 5 – 3 = 2;
push 90;
120
120 > top of stack (90)
pop till top is greater;
Stack becomes empty;
span[6] = current index+1
= 6 + 1 = 7;
push 120;

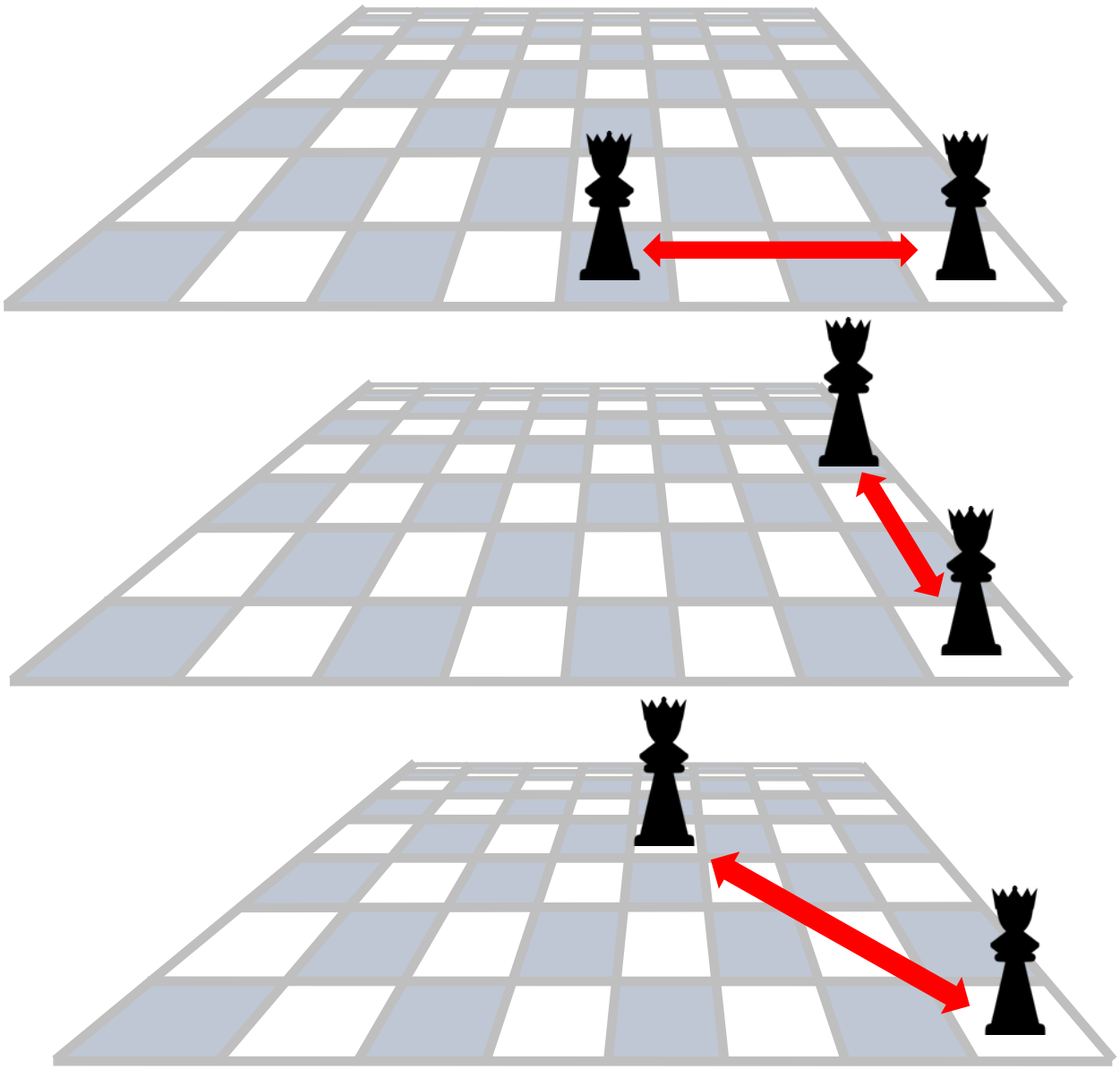
Example: N-Queens problem

Suppose you have a chess board (8x8) and 8 queens



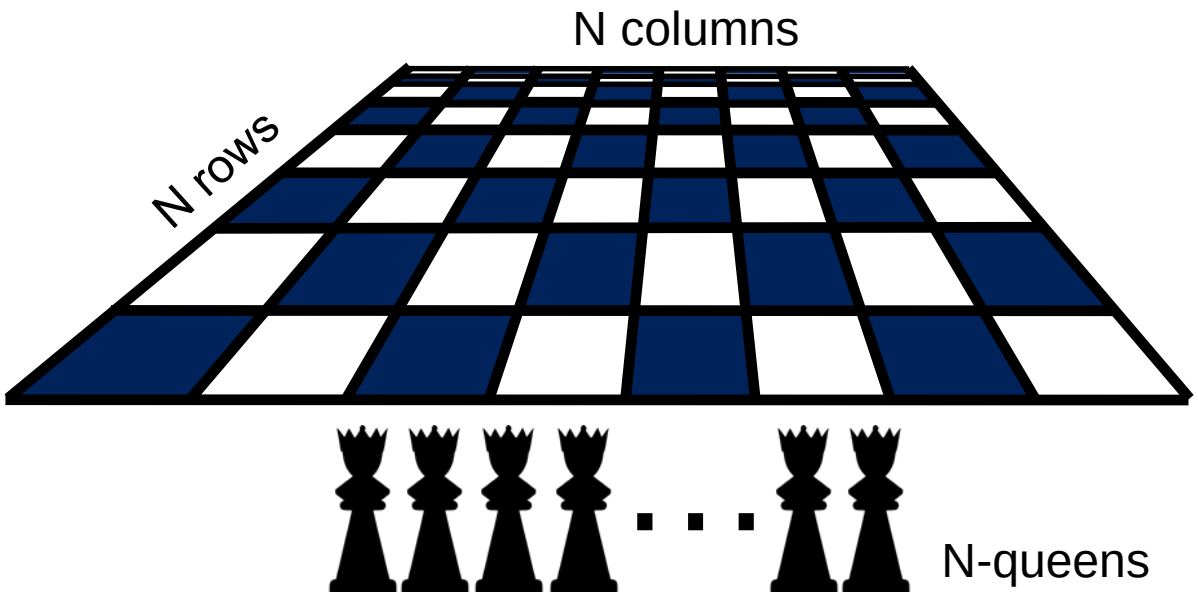
Can the queens be placed on the board so that no two queens can attack each other?

Two queens cannot be in the same row, in the same column, or along the same diagonal



….and more general

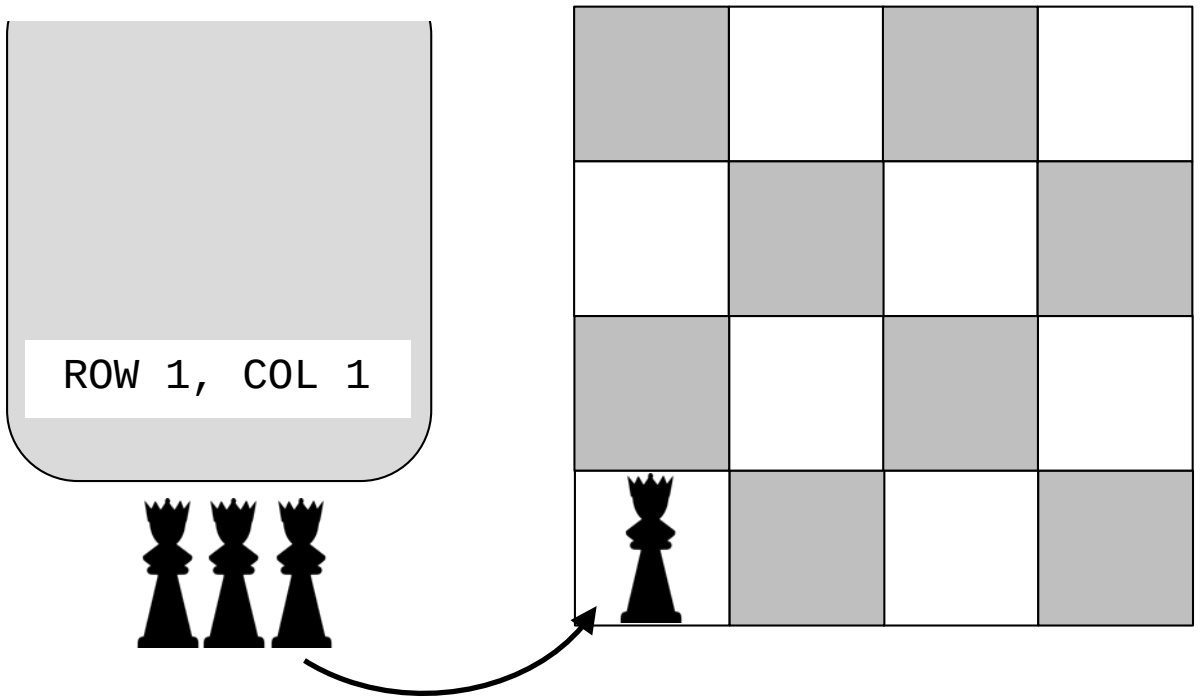
- Now assume the number of queens may vary and the chess board can have varying size



algorithm

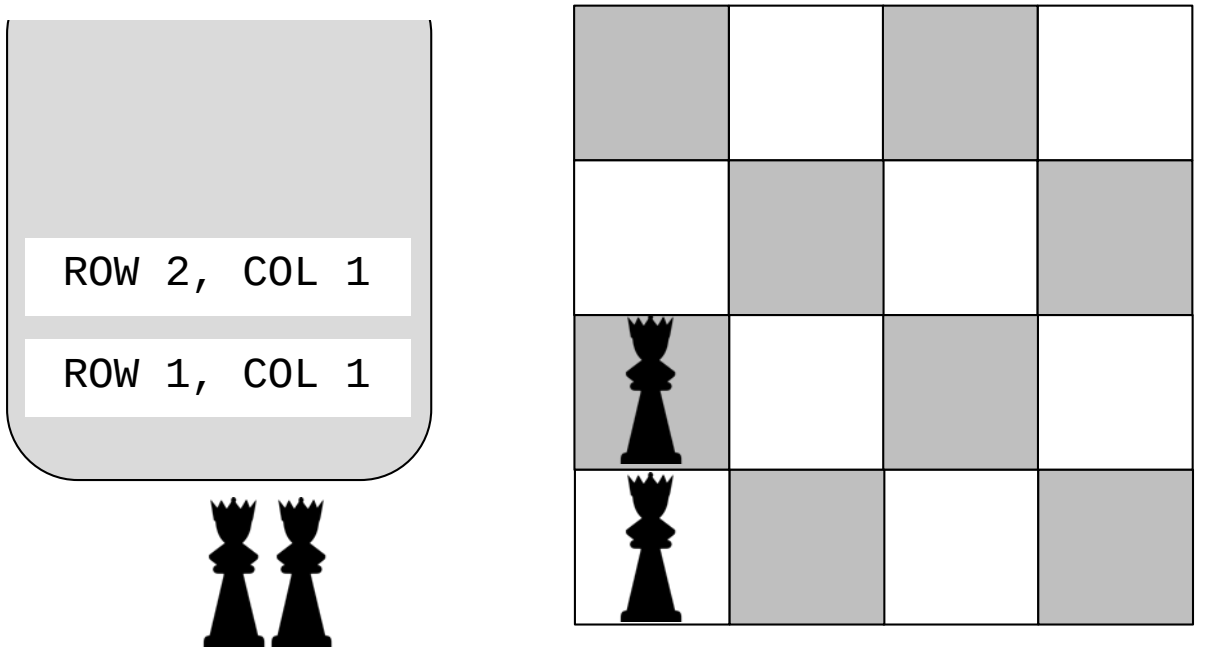
- Use a stack to keep track of where each queen is placed
- We use a counter (`filled`) to keep track how many rows have been filled

`filled = 1`



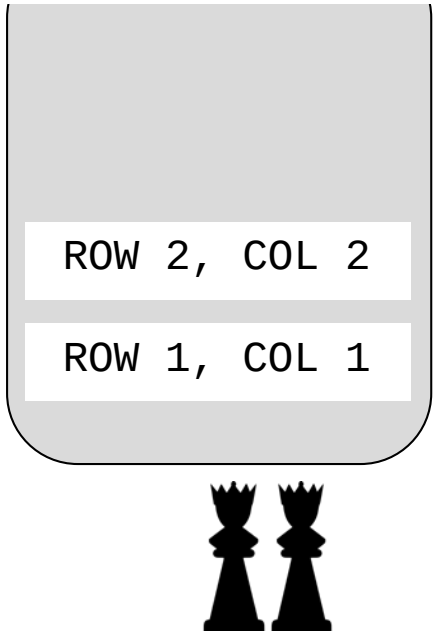
Each time we try to place a new queen in the next row, we start by placing the queen in the first column

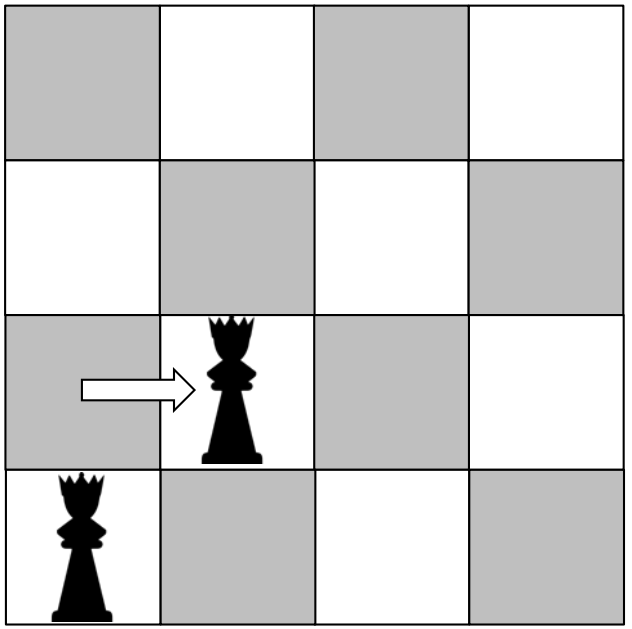
`filled = 1`



If there is a conflict with another queen, then we shift the queen to the next column, …and we keep shifting until no conflict

`filled = 1`

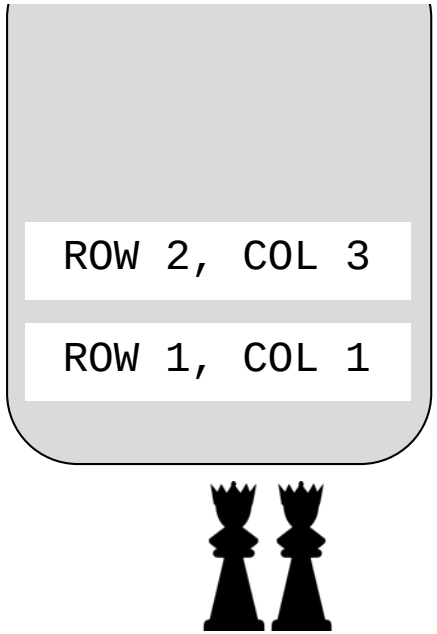


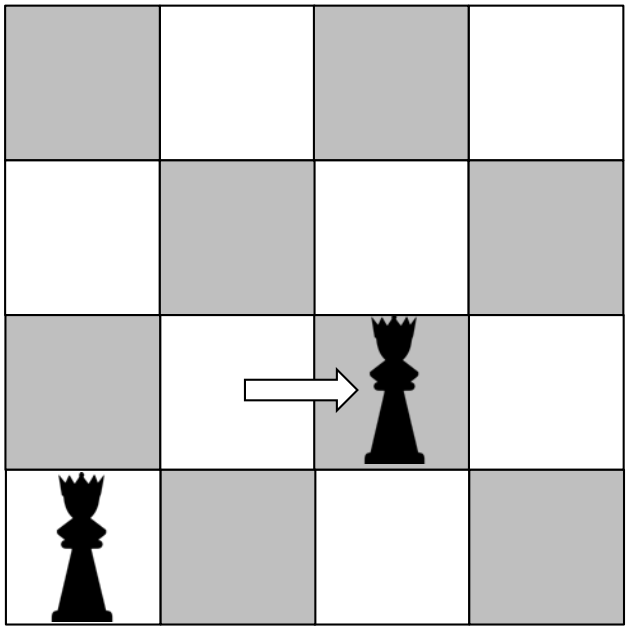


…and we keep shifting until no conflict

When no conflict, stop and increase the counter

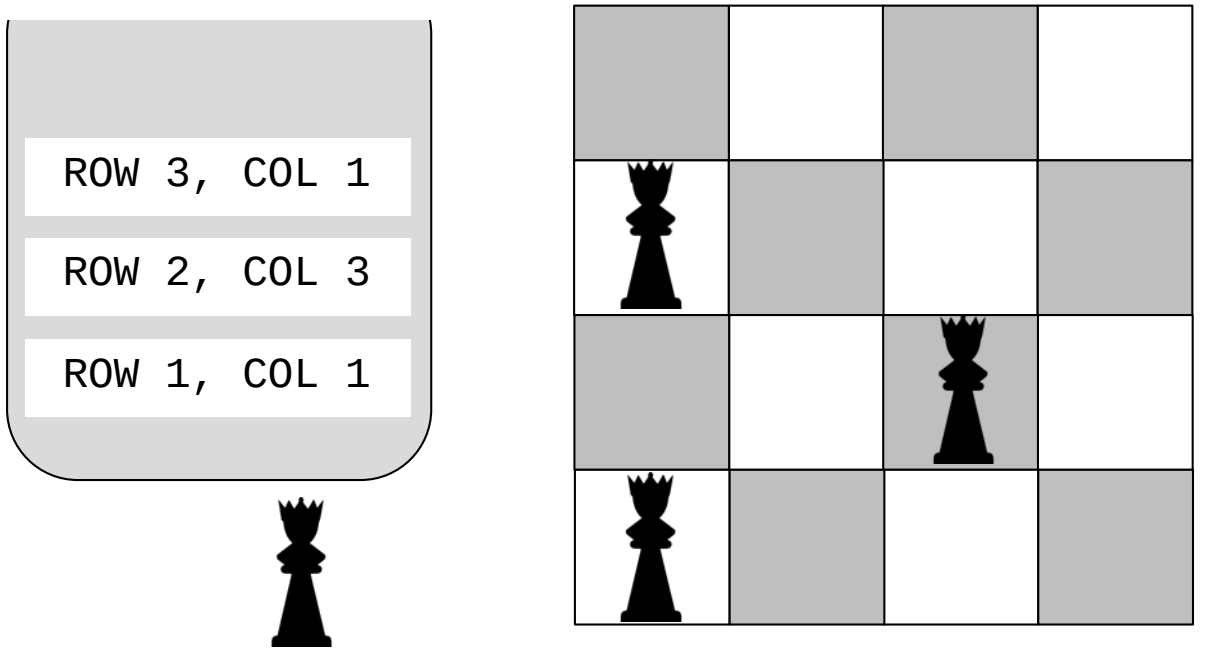
```
filled = 1
filled = 2
```



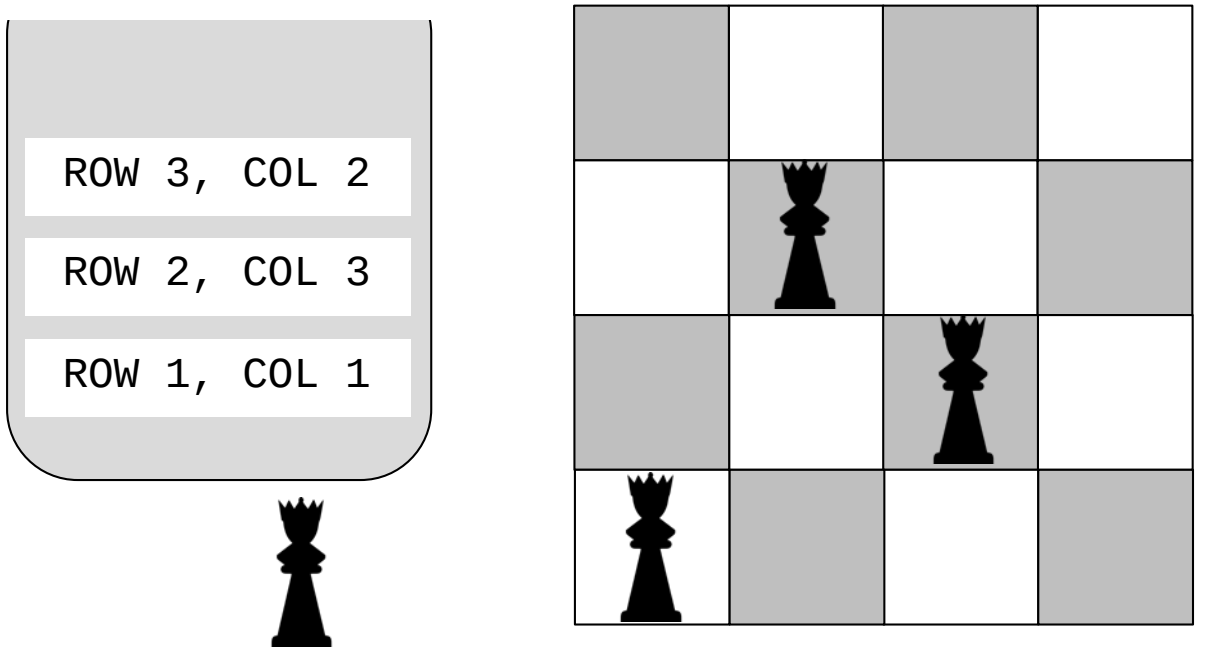


…and repeat

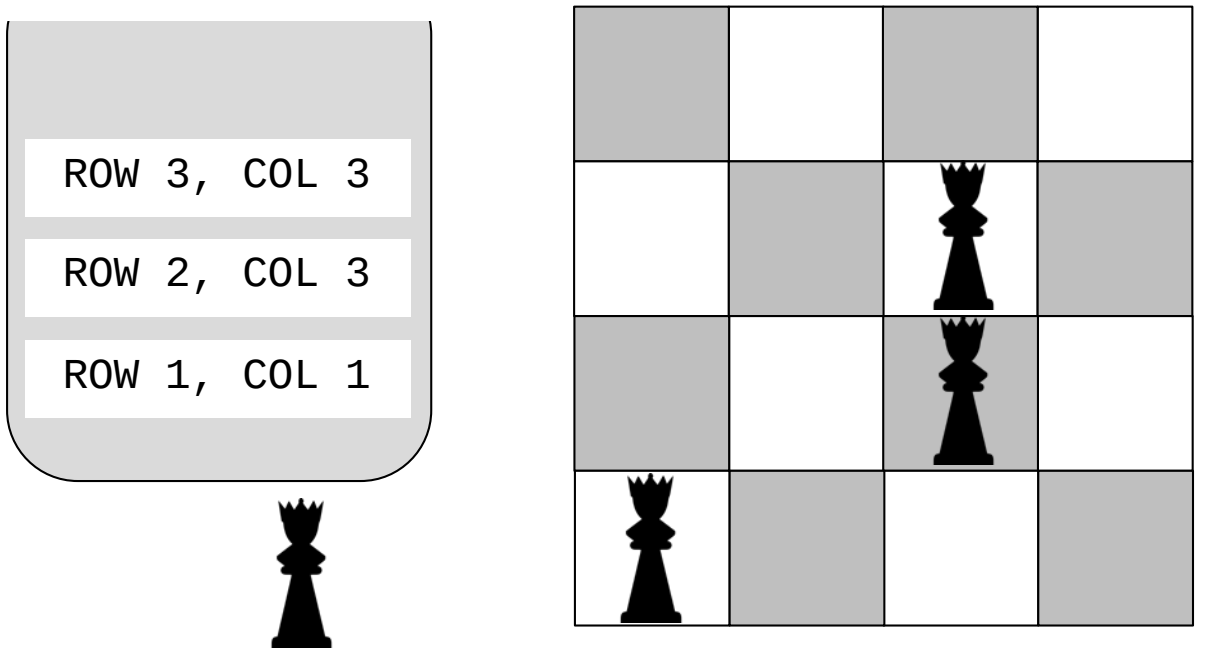
`filled = 2`



filled = 2

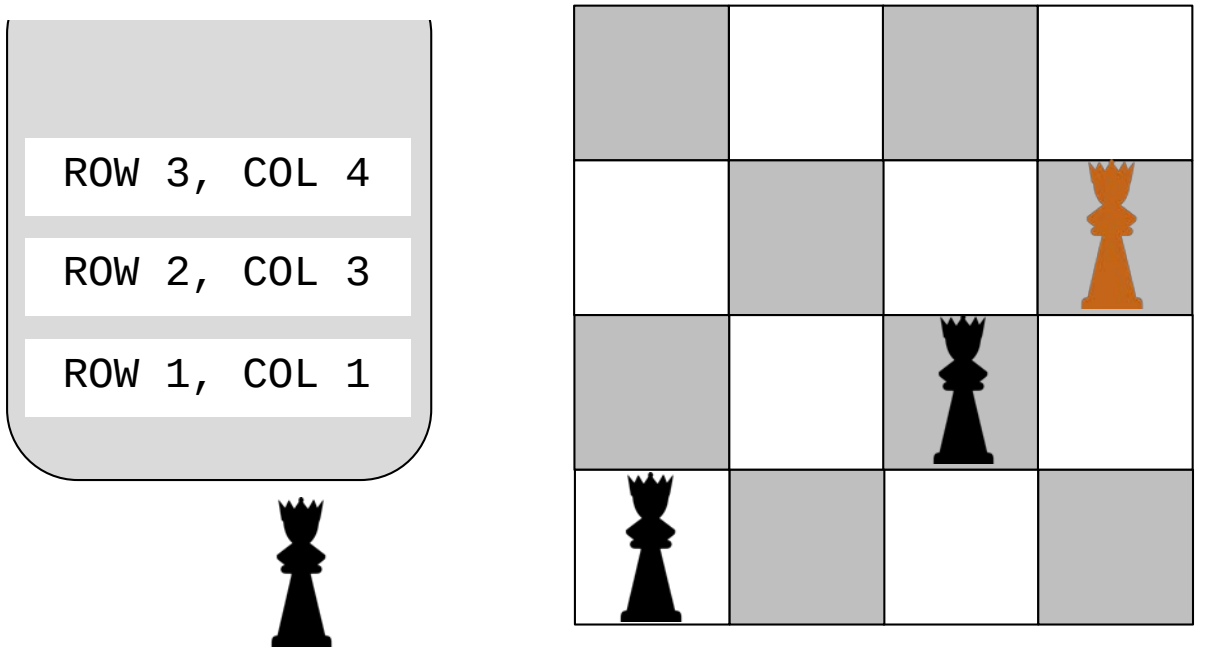


filled = 2

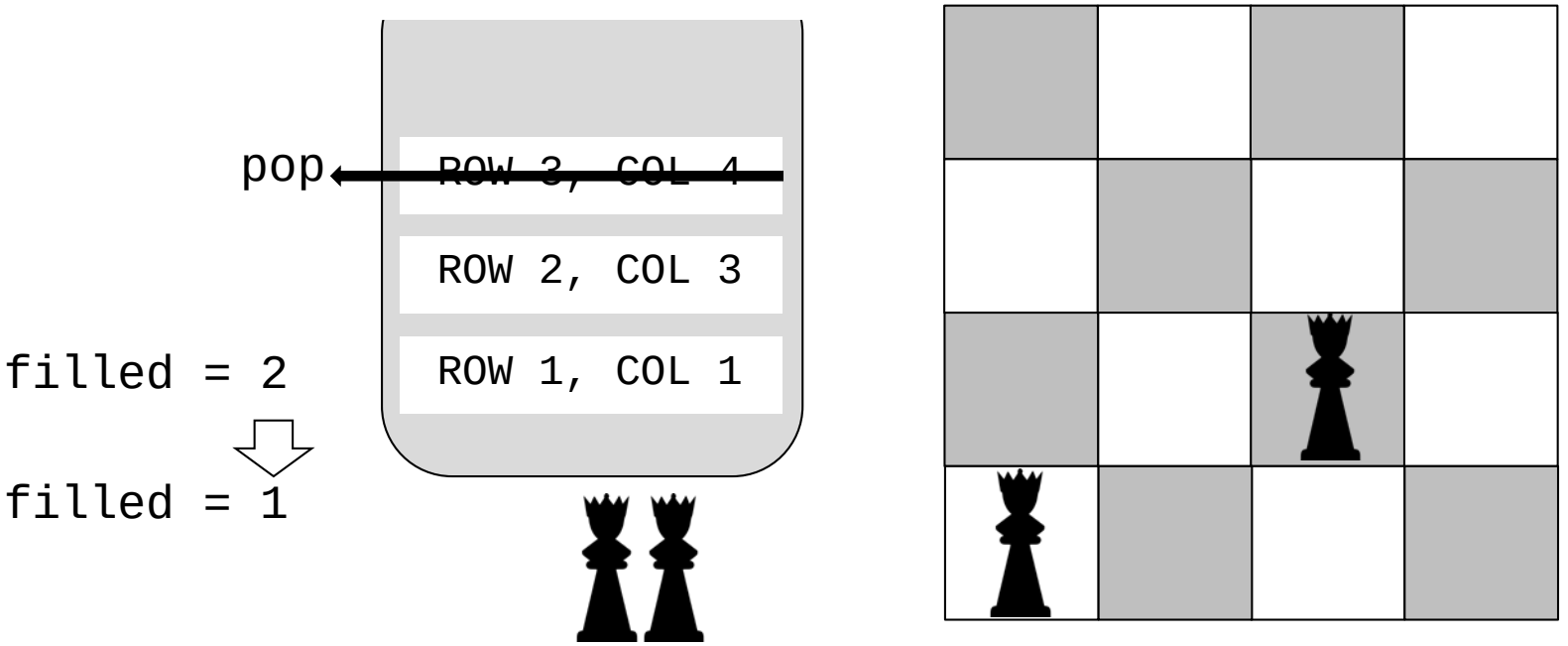


Still a conflict without any more shifts available.

`filled = 2`

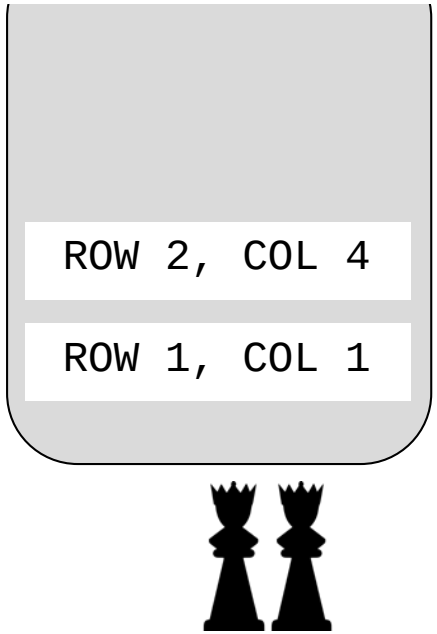


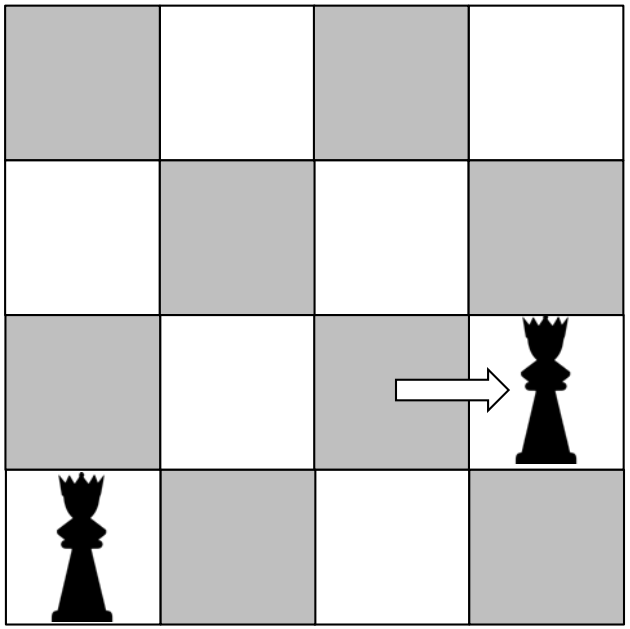
- pop the stack,
- reduce `filled` by 1
- continue working on the previous row



filled = 1

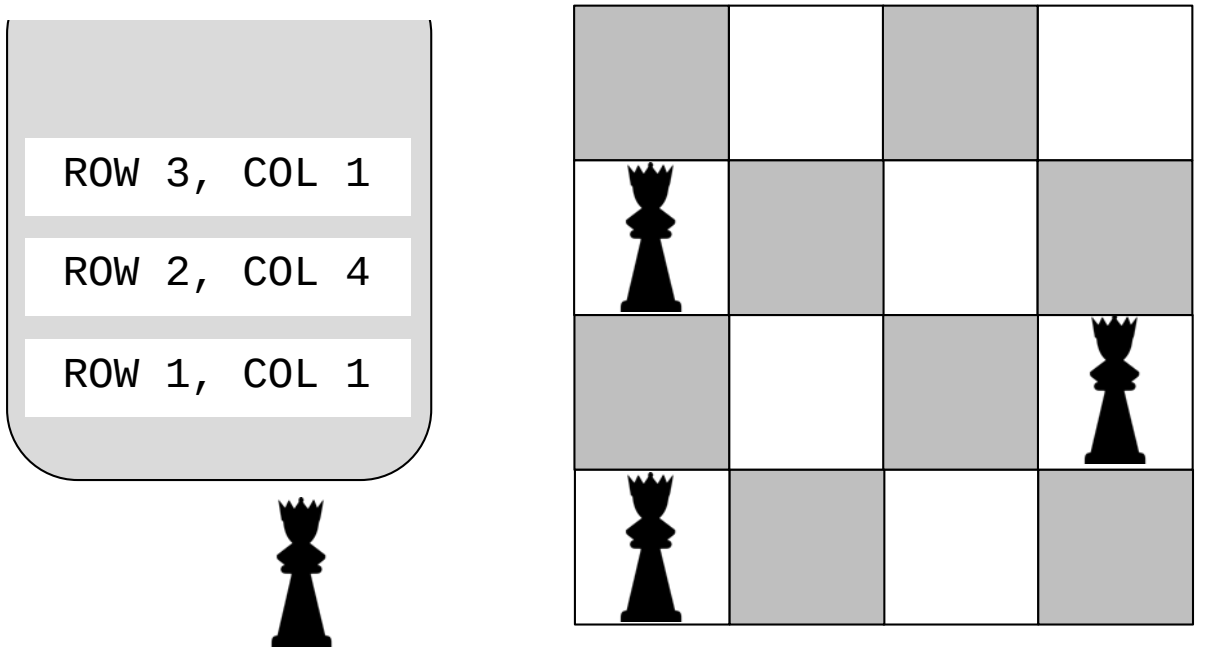
filled = 2





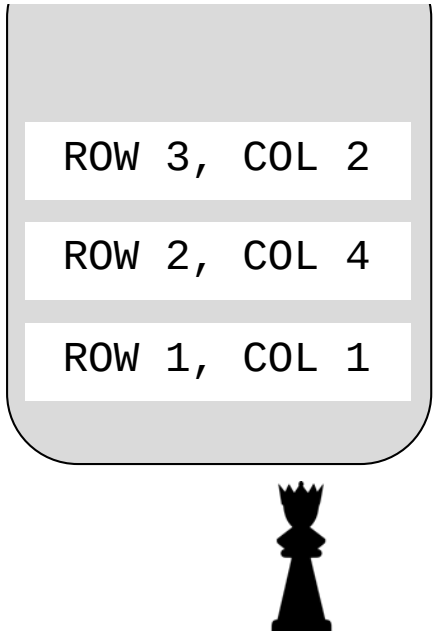
…and start again with next row

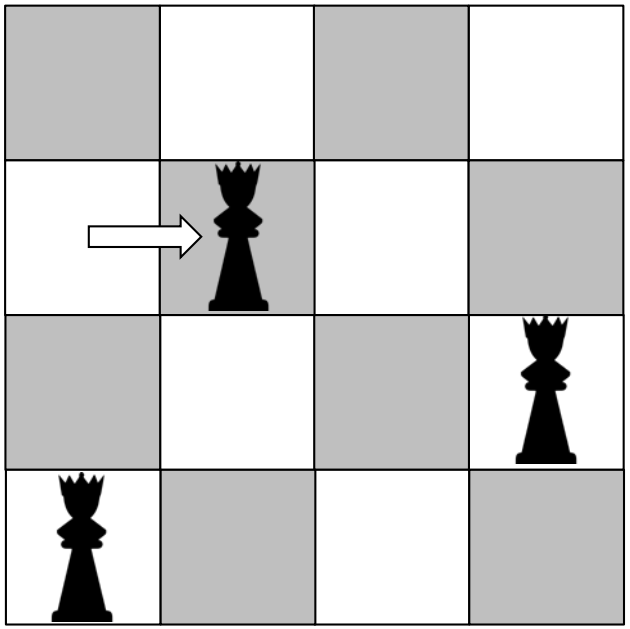
`filled = 2`



filled = 2

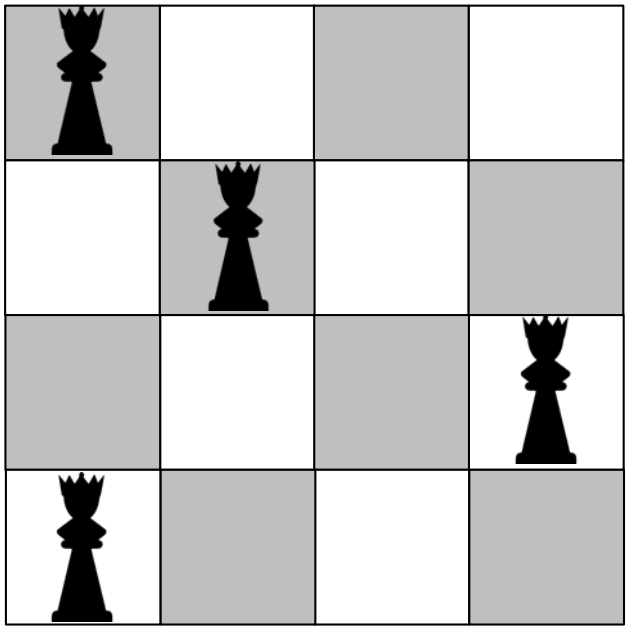
filled = 3





filled = 3

```
ROW 4, COL 1
ROW 3, COL 2
ROW 2, COL 4
ROW 1, COL 1
```



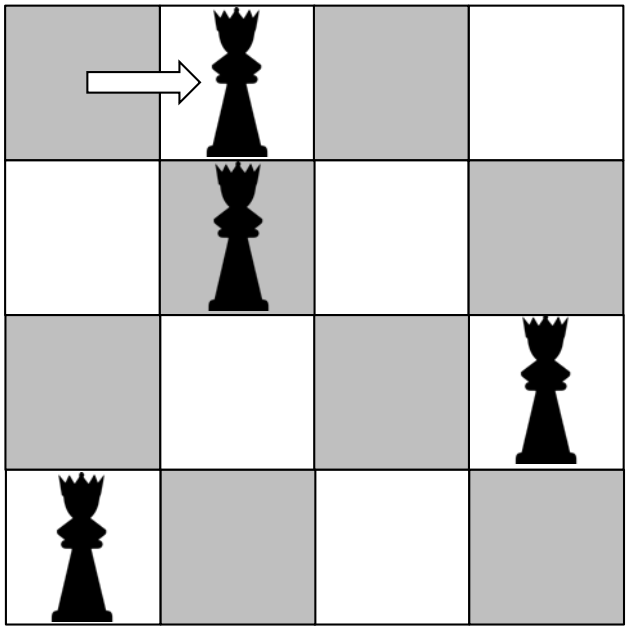
filled = 3

ROW 4, COL 2

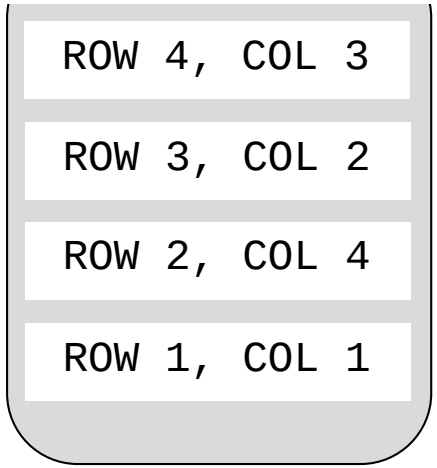
ROW 3, COL 2

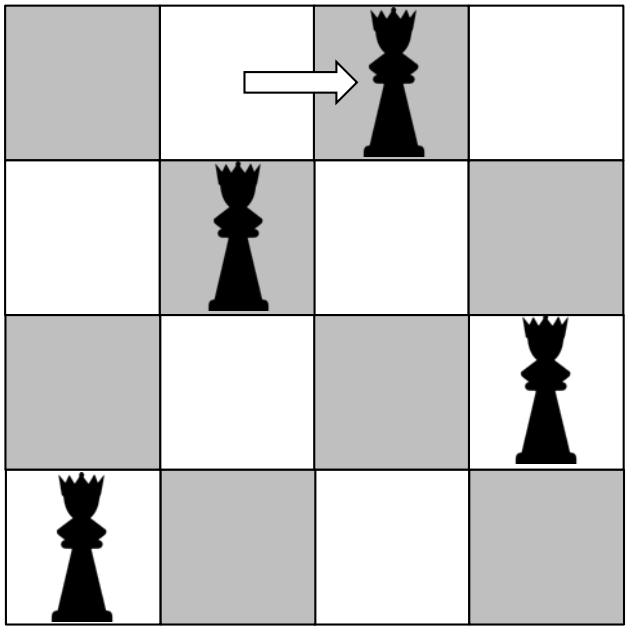
ROW 2, COL 4

ROW 1, COL 1



filled = 3

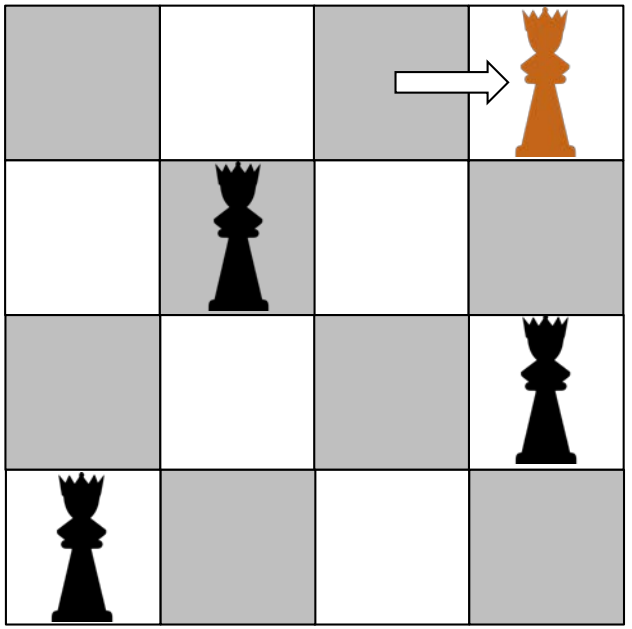




Still a conflict without any more shifts available.

`filled = 3`

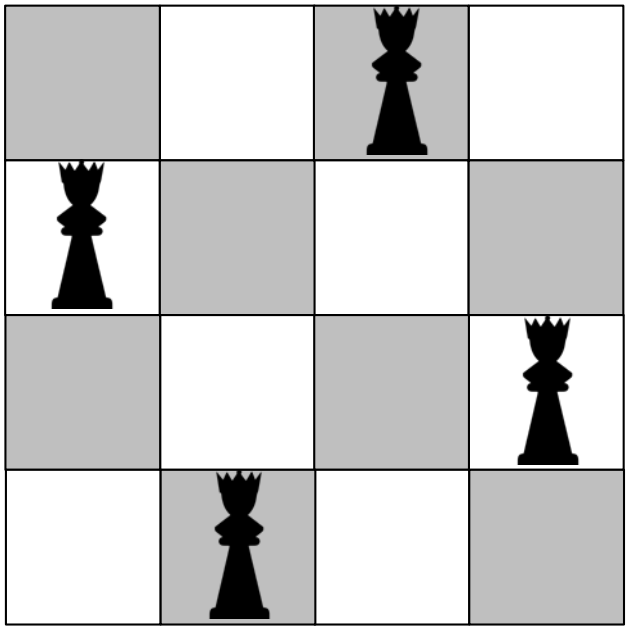
```
ROW 4, COL 4
ROW 3, COL 2
ROW 2, COL 4
ROW 1, COL 1
```



Pop row 4 => shift row 3 => conflict => pop row 3 => pop row 2 => shift row 1 => repeat

`filled = 4`

```
ROW 4, COL 3
ROW 3, COL 1
ROW 2, COL 4
ROW 1, COL 2
```



pseudocode

```
Initialize stack
push first queen position onto the stack, and filled=0
Repeat:
    if no conflicts with queens {
        filled = filled+1;
        if filled = N {
            done
        }
        else {
        move to next row and place a queen in first column
        }
    }
    else if a conflict and there is room to shift {
        move current queen to the right, adjusting the stack record (new
    position)
        }
    else if there is a conflict and there is no room to shift {
    Backtrack: keep popping the stack and filled=filled-1 until you reach a
    row where the queen can be shifted; shift that queen
    }
```

conclusions

- We saw array / linked list / backtracking implementations
- Arrays are a waste of space when fewer elements are used
- Arrays cannot handle dynamic data applications (new data to be added, permanently deleted data = “stretching”/“shrinking”)
- We need dynamic allocation of memory for new or fewer elements
- This is why we use what we call “linked lists”
- In specific applications, to implement LIFO and FIFO, we use stacks and queues respectively.