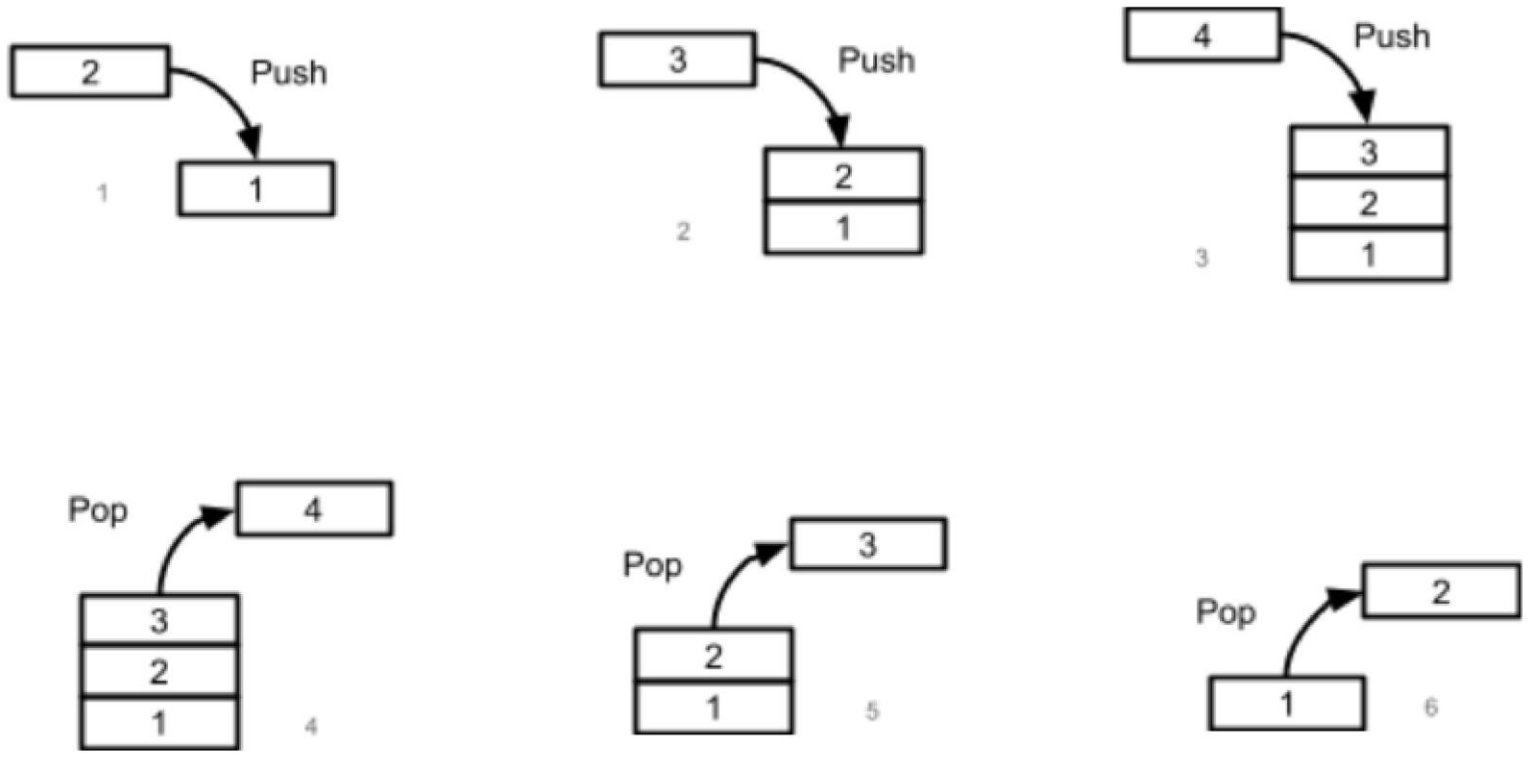
بسم الله الرحمن الرحیم

ساختمان‌های داده

جلسه ۱۳

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

Stack



Stack

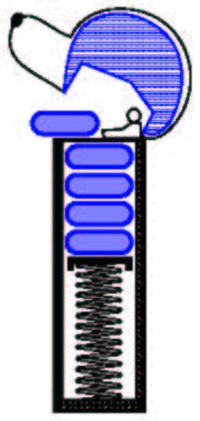
A ***stack*** is a container of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, “last”) object can be removed at any time.

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored



Stack Interface in C++

- C++ interface corresponding to our Stack ADT
- Uses an exception class StackEmpty
- Different from the built-in C++ STL class stack
- STL: Standard Template Library

```
template <typename E>
class Stack  {
public:
   int size() const;
   bool empty() const;
   const E& top() const
         throw(StackEmpty);
   void push(const E& e);
   void pop() throw(StackEmpty);
}
```

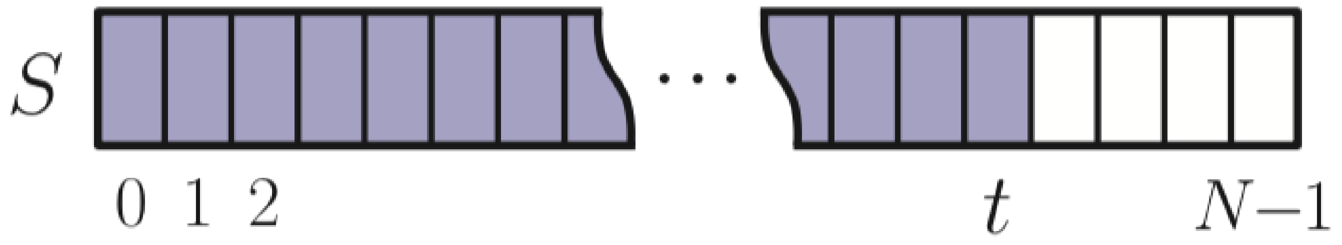
Example Implementation?

- اول آرایه را top بگیریم؟

Example Implementation: Array-based Stack

IUT-ECE

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



```
Algorithm size():
    return $t + 1$
Algorithm empty():
    return $(t < 0)$
Algorithm top():
    if empty() then
        throw StackEmpty exception
    return $S[t]$
Algorithm push($e$):
    if size() $= N$ then
        throw StackFull exception
    $t \leftarrow t + 1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() then
        throw StackEmpty exception
    $t \leftarrow t - 1$
```

Example Implementation: Array-based Stack

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };         // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);  // constructor from capacity
  int size() const;                    // number of items in the stack
  bool empty() const;                  // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);        // pop the stack
  // ...housekeeping functions omitted
private:                               // member data
  E* S;                                // array of stack elements
  int capacity;                        // stack capacity
  int t;                               // index of the top of the stack
};
```

Example Implementation: Array-based Stack

IUT-ECE

```
template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                              // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                              // is the stack empty?

template <typename E>                              // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                              // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                              // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --t;
}
```

Example Implementation: Array-based Stack

IUT-ECE

```
ArrayStack<int> A;                         // A = [], size = 0
A.push(7);                                 // A = [7*], size = 1
A.push(13);                                // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();          // A = [7*], outputs: 13
A.push(9);                                 // A = [7, 9*], size = 2
cout << A.top() << endl;                   // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();          // A = [7*], outputs: 9
ArrayStack<string> B(10);                  // B = [], size = 0
B.push("Bob");                             // B = [Bob*], size = 1
B.push("Alice");                           // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();          // B = [Bob*], outputs: Alice
B.push("Eve");                             // B = [Bob, Eve*], size = 2
```

Code Fragment 5.6: An example of the use of the ArrayStack class. The contents of the stack are shown in the comment following the operation. The top of the stack is indicated by an asterisk ("$*$").

Example Implementation: Array-based Stack

- Performance
  - Let $\boldsymbol{n}$ be the number of elements in the stack
  - The space used is $\boldsymbol{O}(\boldsymbol{n})$
  - Each operation runs in time $\boldsymbol{O}(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception
- Linked-list based Stack in the text (Goodrich…,Chapter 5.1.5)

Example: Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Stack in C++ STL

IUT-ECE

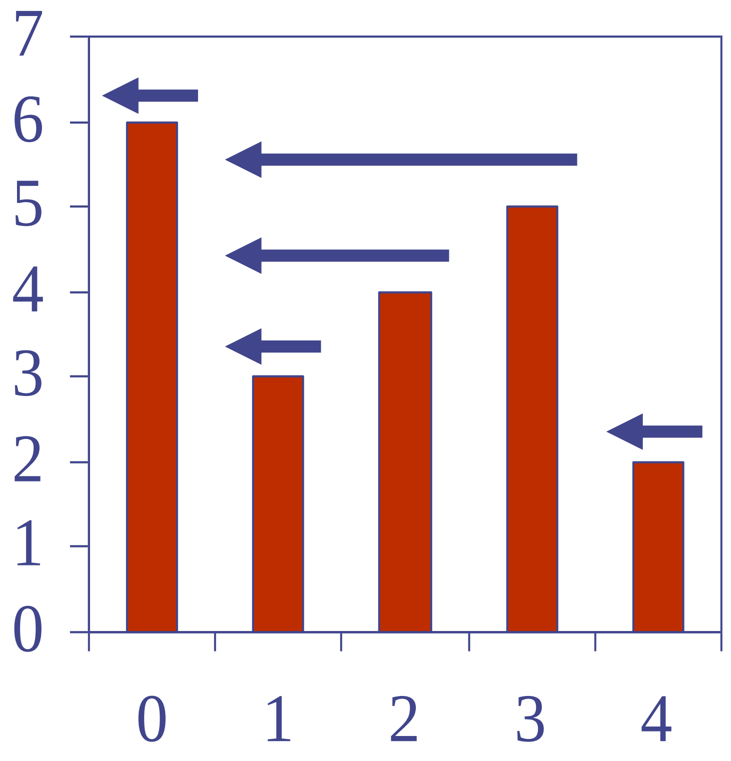
```
#include <stack>
using std::stack;                  // make stack accessible
stack<int> myStack;                // a stack of integers
```

size():	Return the number of elements in the stack.
empty():	Return true if the stack is empty and false otherwise.
push(e):	Push e onto the top of the stack.
pop():	Pop the element at the top of the stack.
top():	Return a reference to the element at the top of the stack.

Example: Computing Spans

- Given an 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 $X[j] \leq X[i]$

- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

Example: Computing Spans

Algorithm: span1

			i		
X	6	3	4	5	2
S	1	1	2	3	1

- Loop over *i* = 0, 1, 2, 3, 4
- For each *i*, compute S[i]. How?
  - From X[i] downward on, compute the number of elements which are consecutively smaller than X[i]

Example: Computing Spans

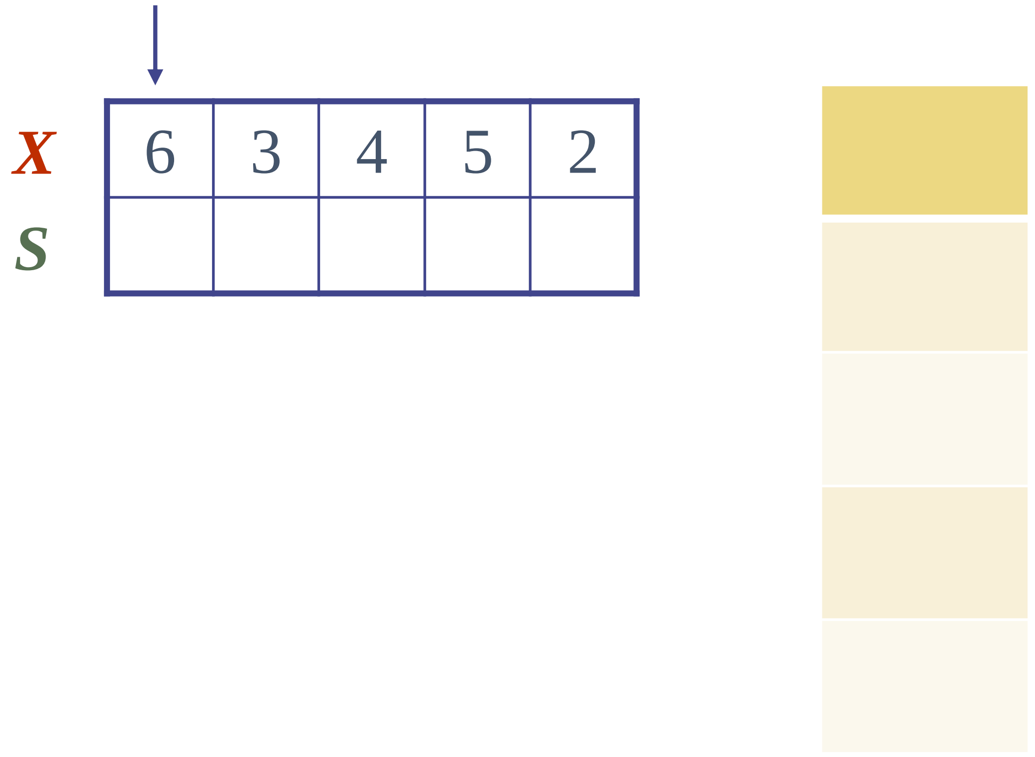
```
Algorithm spans1(X, n)
    Input array X of n integers
    Output array S of spans of X                        #
    S ← new array of n integers                         n
    for i ← 0 to n − 1 do                               n
        s ← 1                                           n
        while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

- Algorithm ***spans1*** runs in $O(n^2)$ time

Example: Computing Spans

IUT-ECE

Algorithm: span2



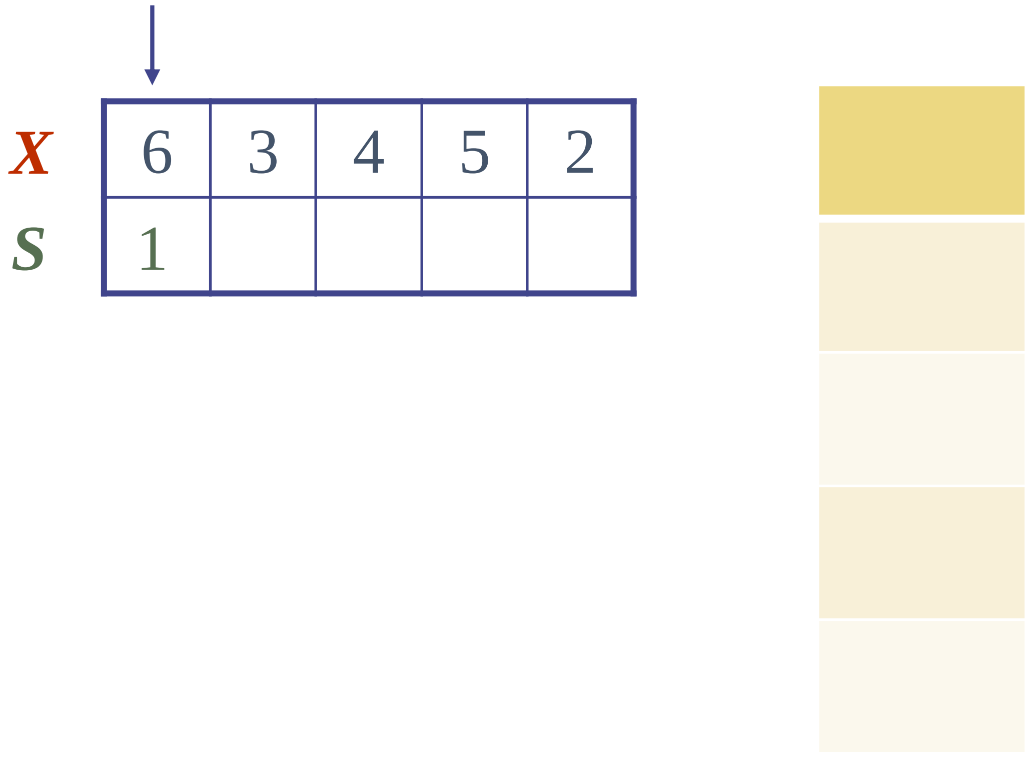
Stack for “index”

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
         while (¬A.empty() ∧
                 X[A.top()] ≤ X[i] ) do
             A.pop()
         if A.empty() then
             S[i] ← i + 1
         else
             S[i] ← i − A.top()
         A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



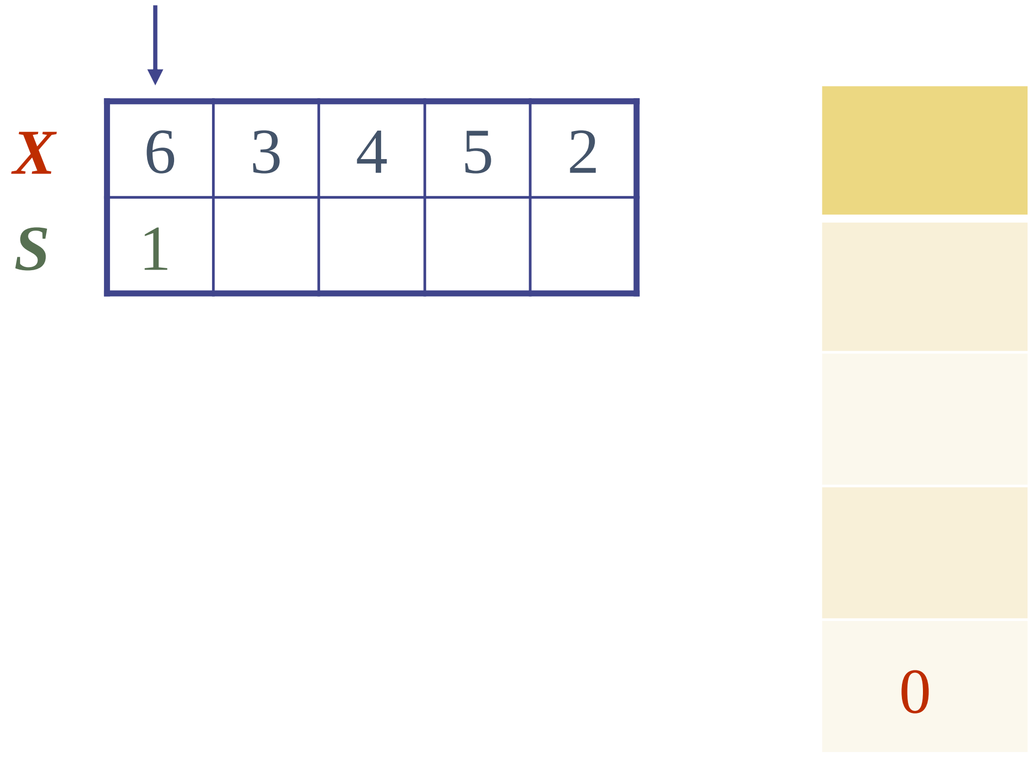
Stack for “index”

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



Stack for “index”

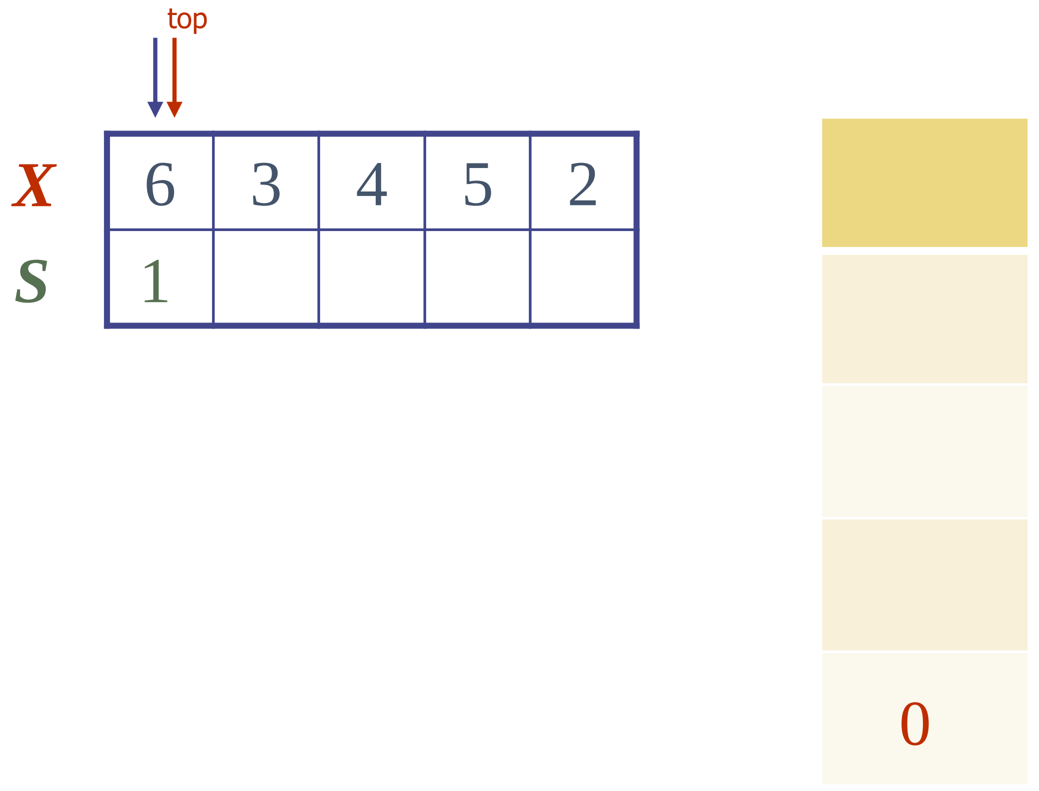
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n – 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i – A.top()
        A.push(i)
    return S
```

Mojtaba Khalili

Example: Computing Spans

IUT-ECE

Algorithm: span2



Stack for "index"

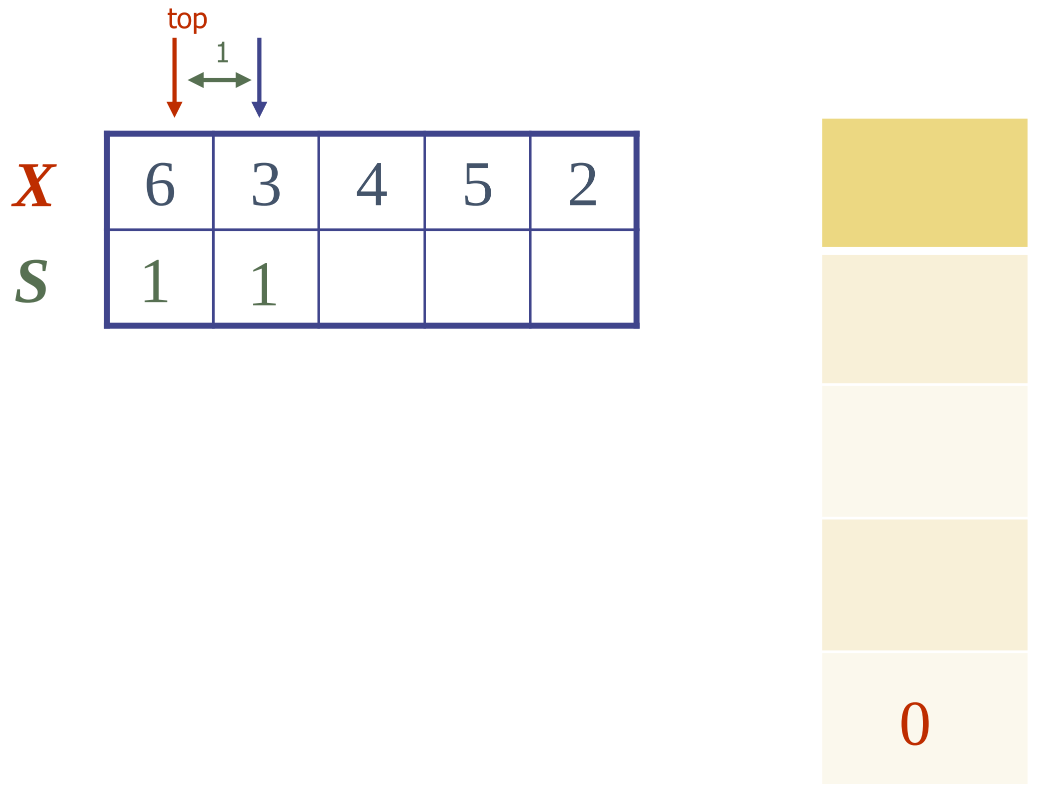
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Mojtaba Khalili

Example: Computing Spans

IUT-ECE

Algorithm: span2



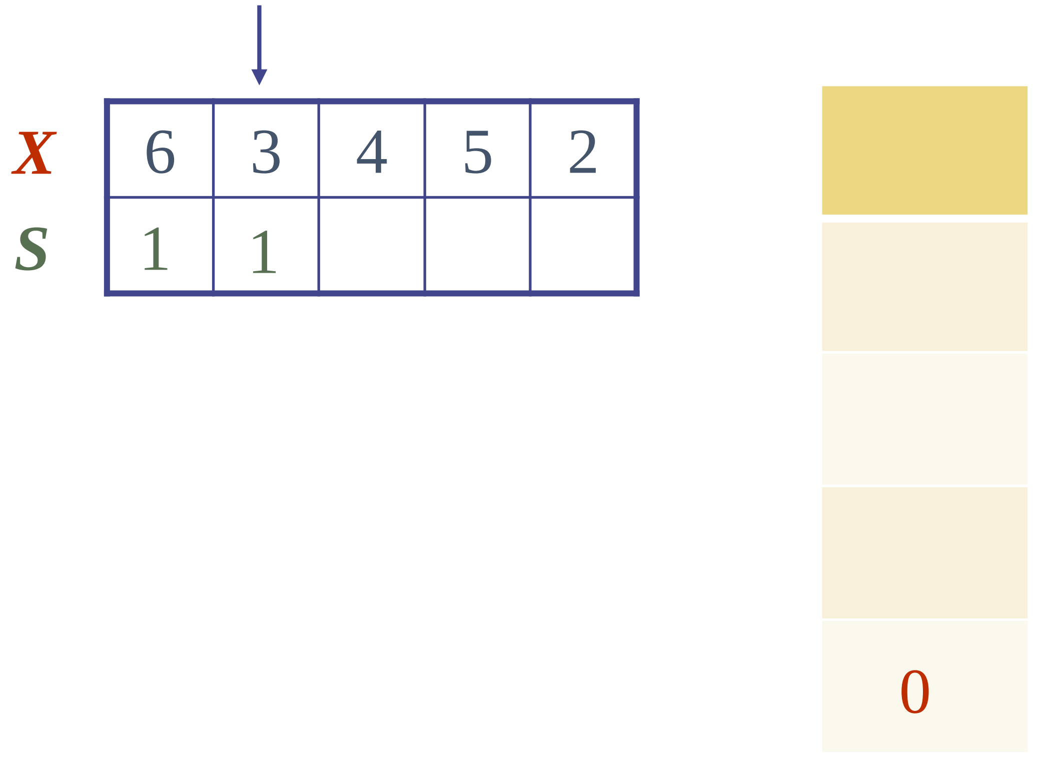
Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n – 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i – A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



Stack for "index"

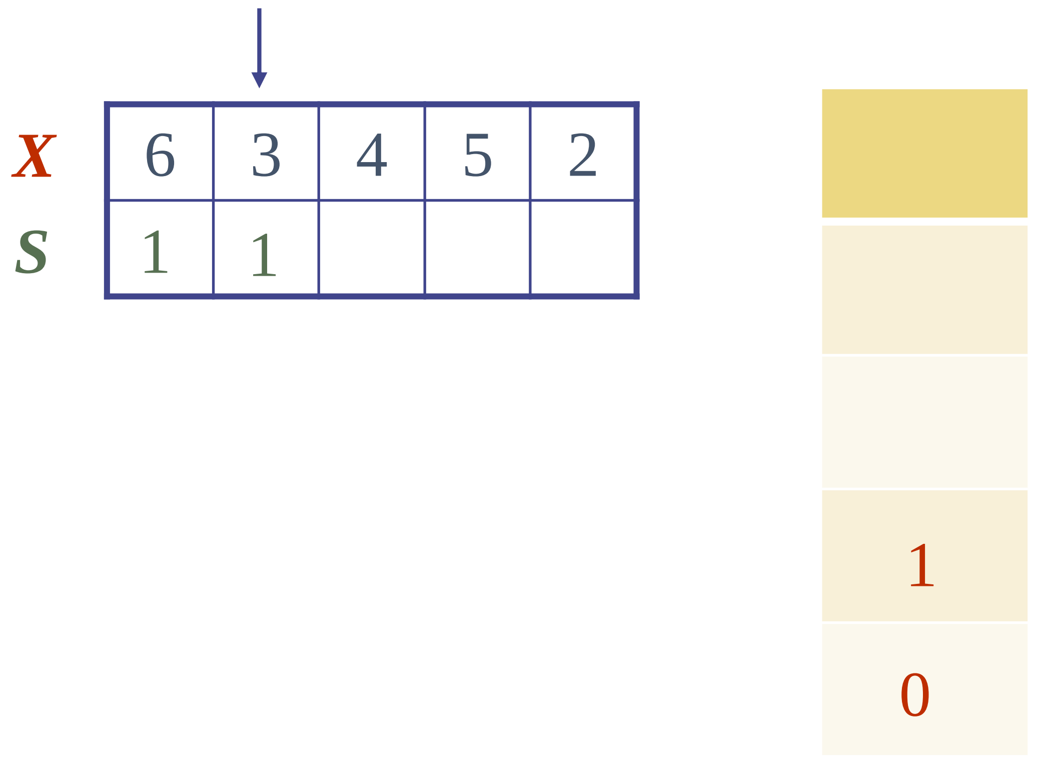
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Mojtaba Khalili

Example: Computing Spans

IUT-ECE

Algorithm: span2



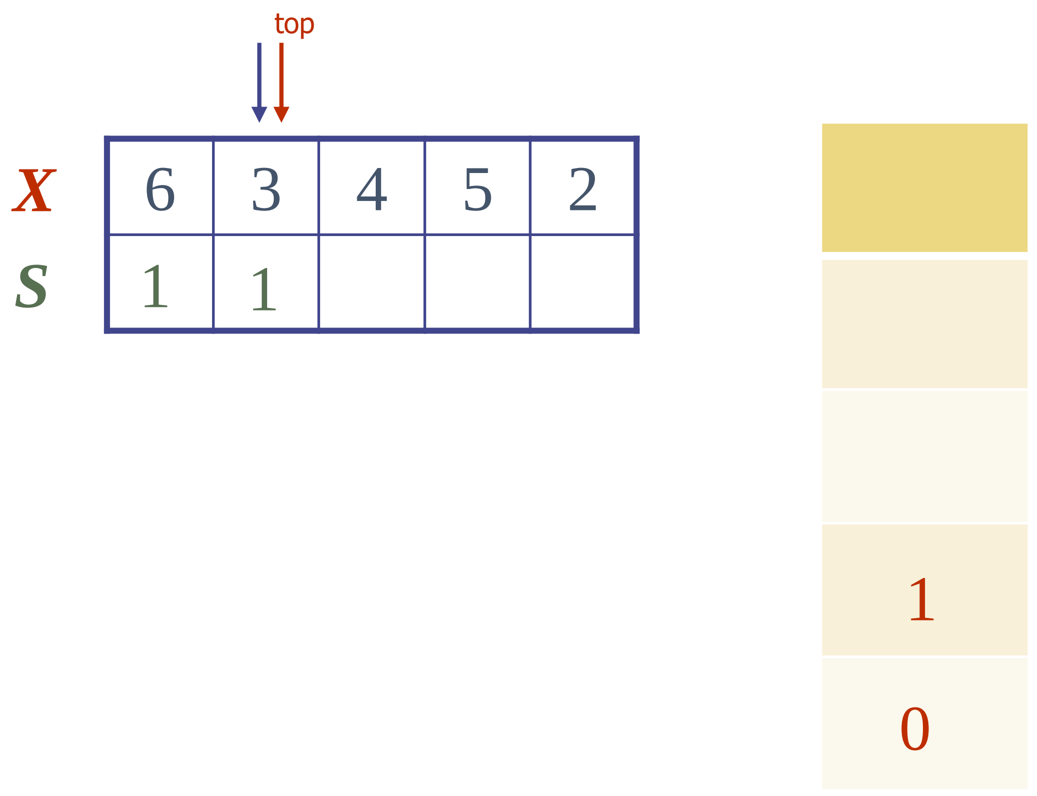
Stack for “index”

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

IUT-ECE

Example: Computing Spans

Algorithm: span2



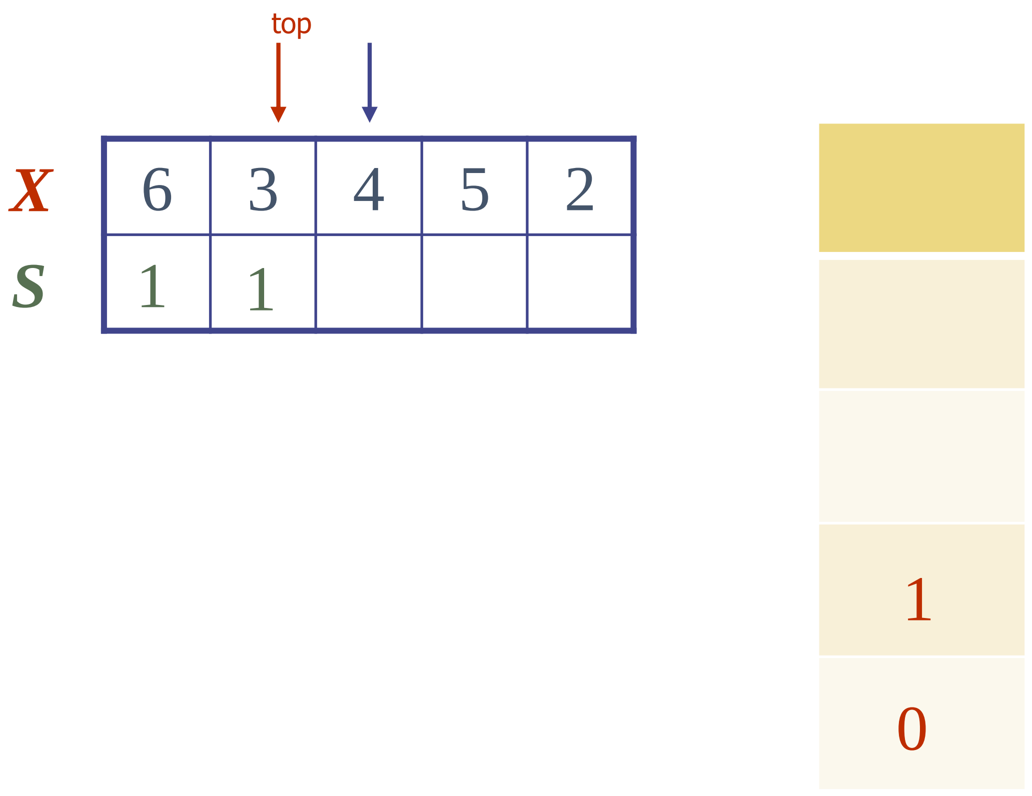
Stack for “index”

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



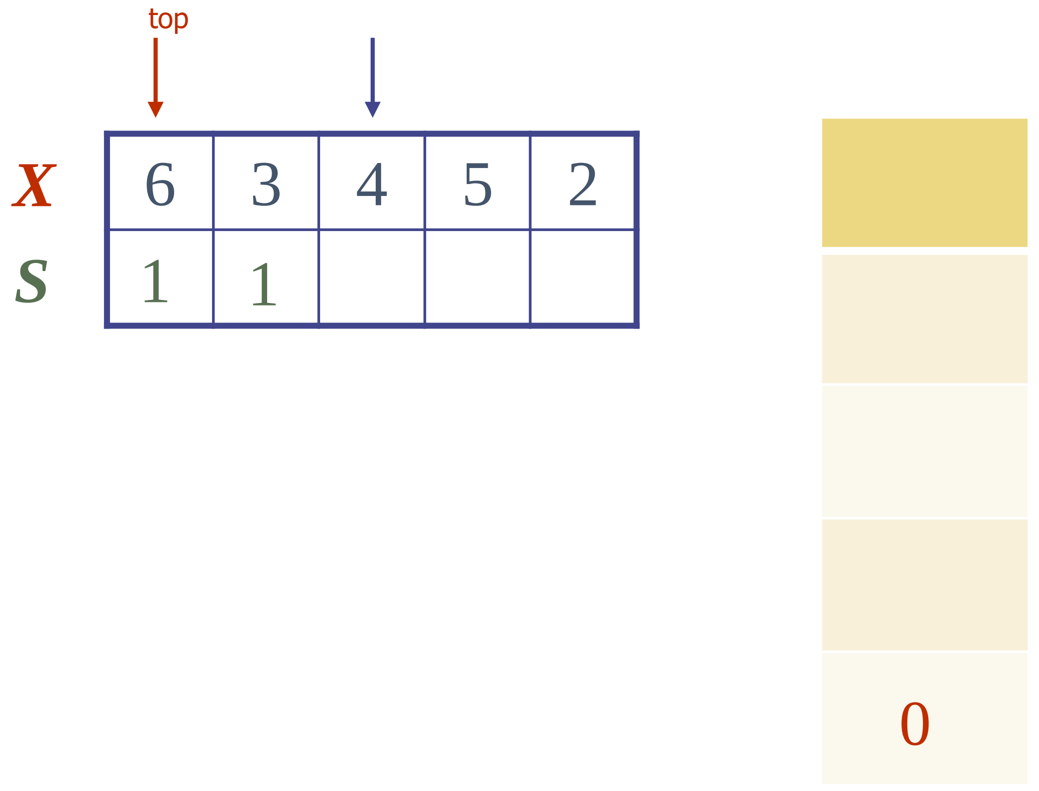
Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



Stack for "index"

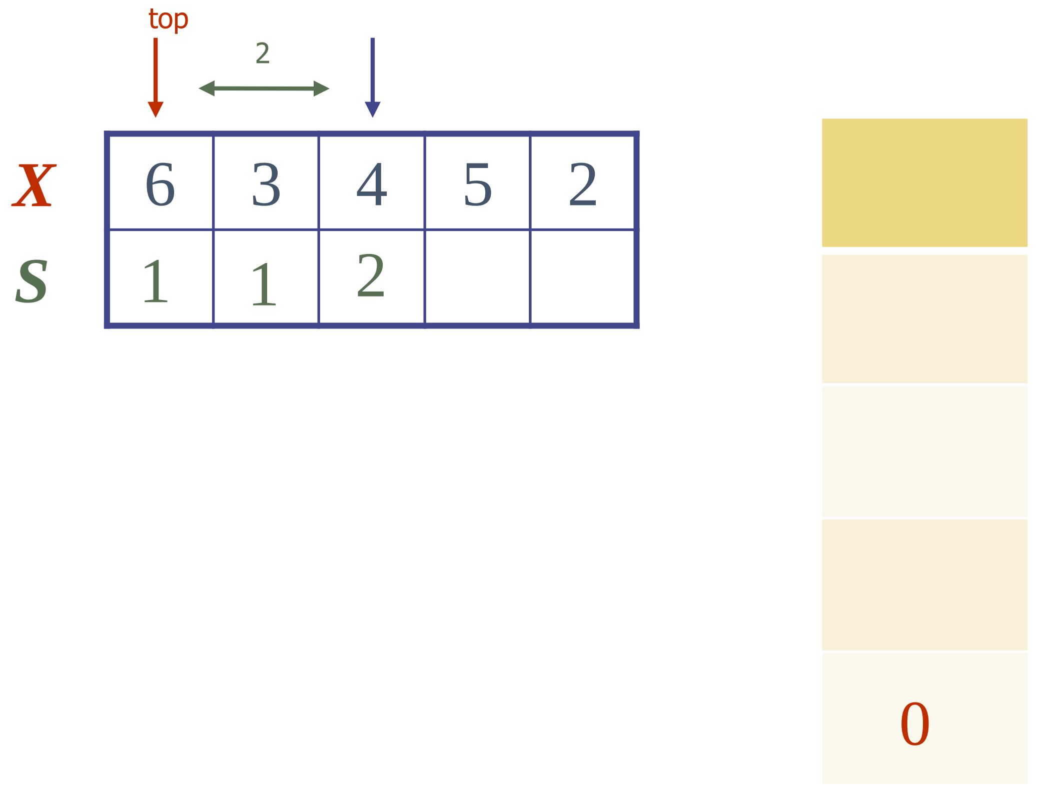
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
           A.pop()
        if A.empty() then
           S[i] ← i + 1
        else
           S[i] ← i − A.top()
        A.push(i)
    return S
```

Mojtaba Khalili

Example: Computing Spans

IUT-ECE

Algorithm: span2

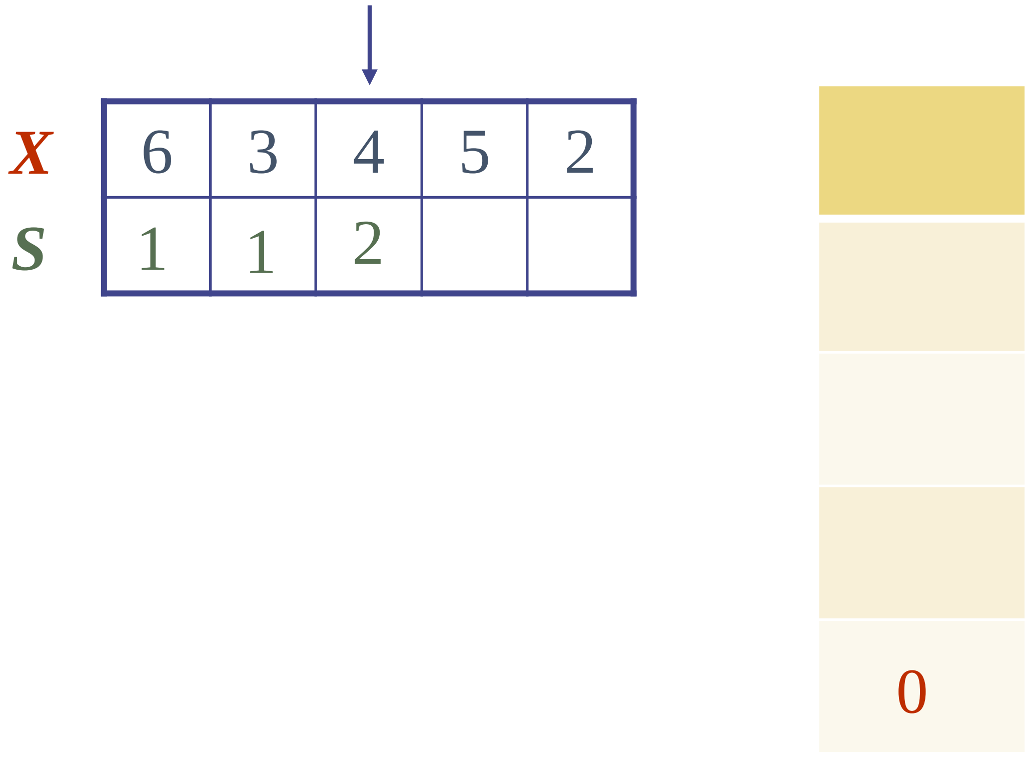


Stack for “index”

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

Algorithm: span2



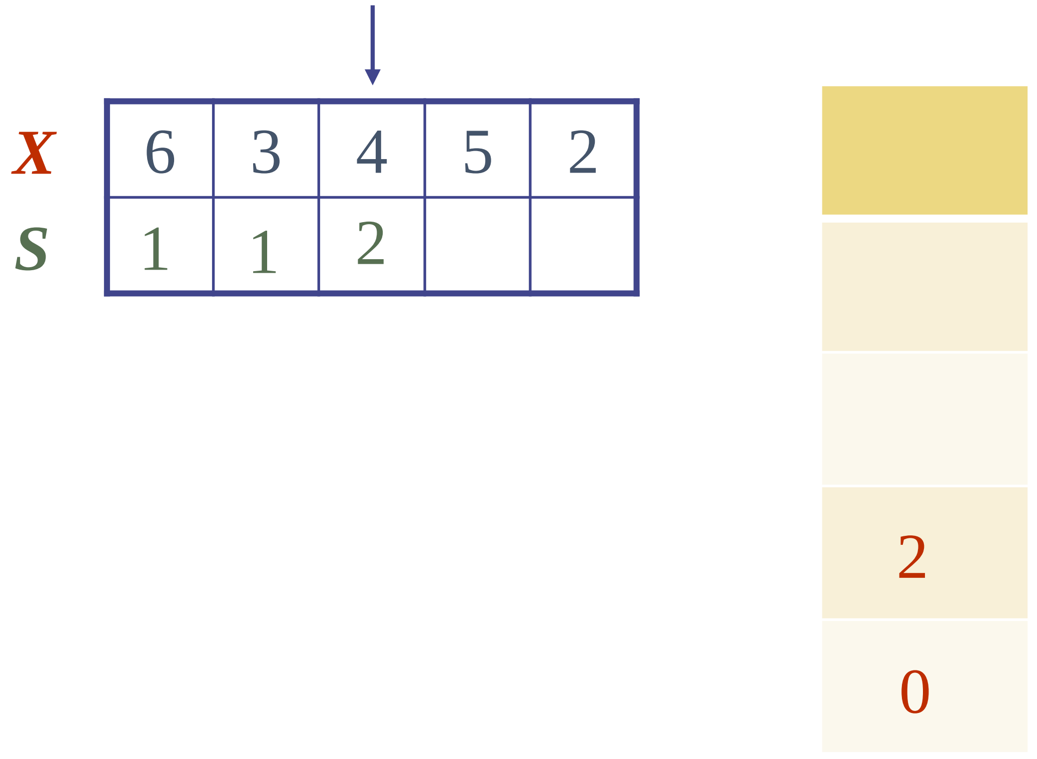
Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n – 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i – A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



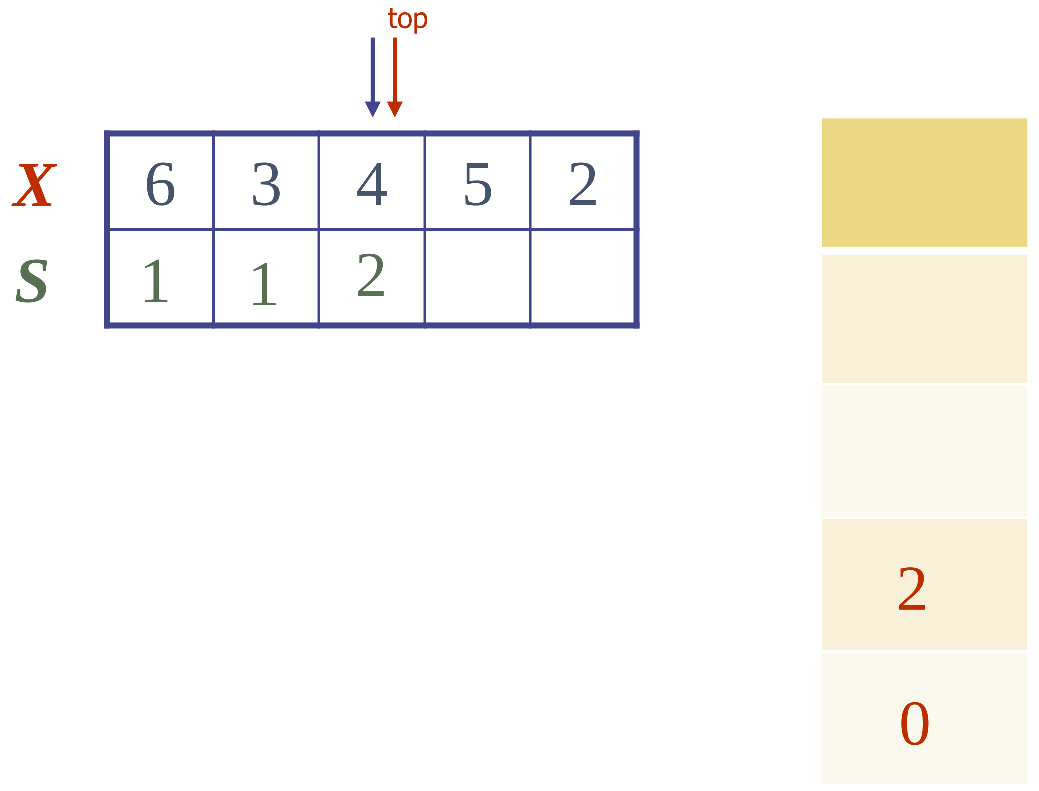
Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2

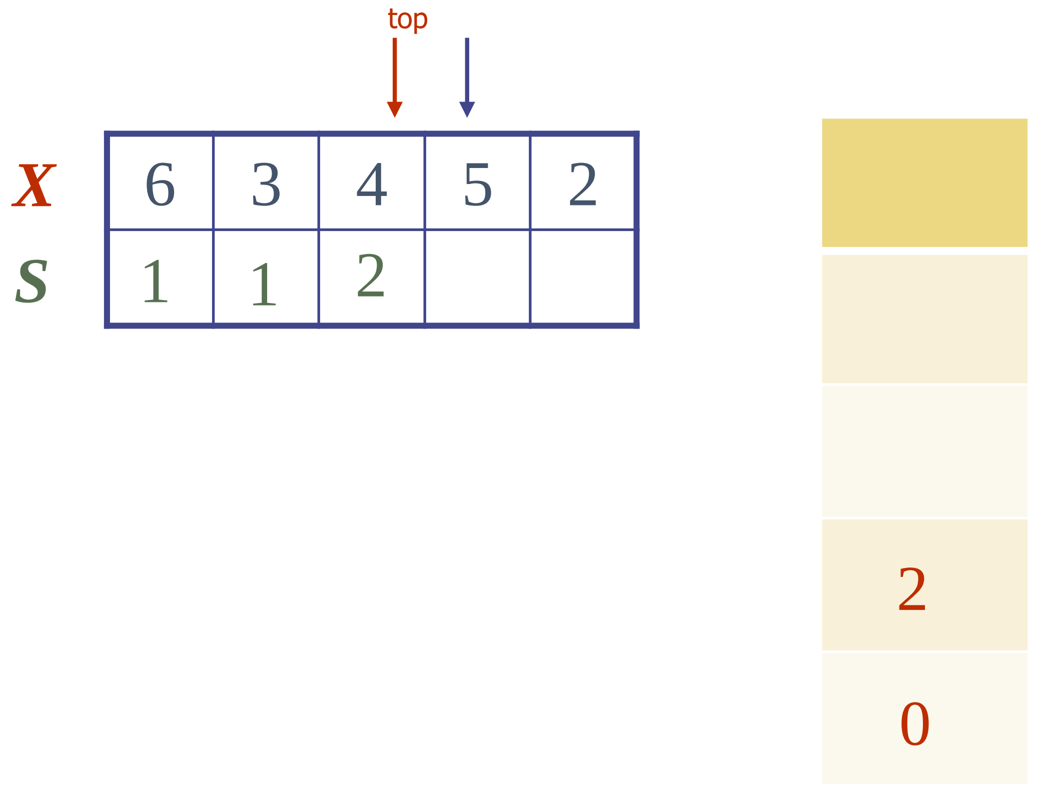


Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

Algorithm: span2



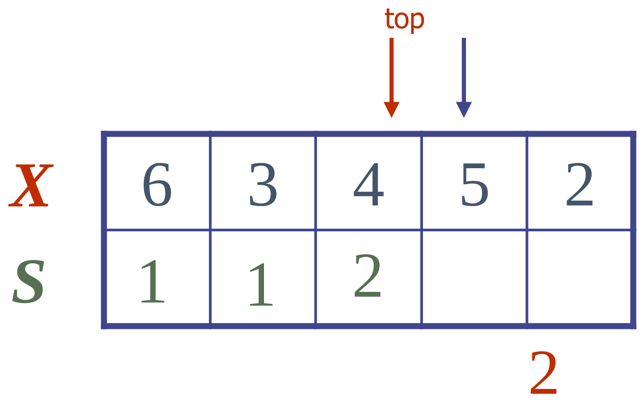
Stack for "index"

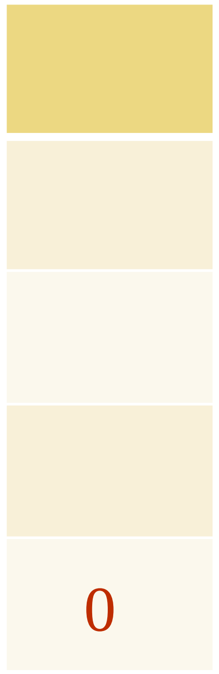
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2





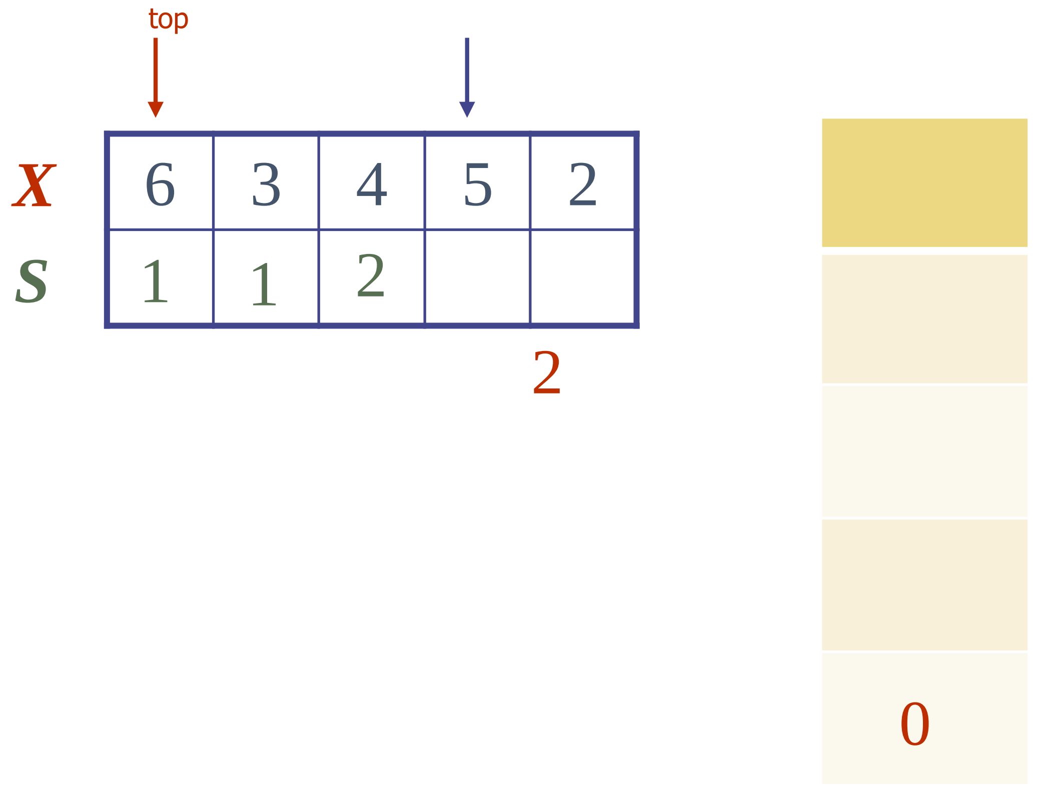
Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n – 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i – A.top()
        A.push(i)
    return S
```

Example: Computing Spans

IUT-ECE

Algorithm: span2



Stack for "index"

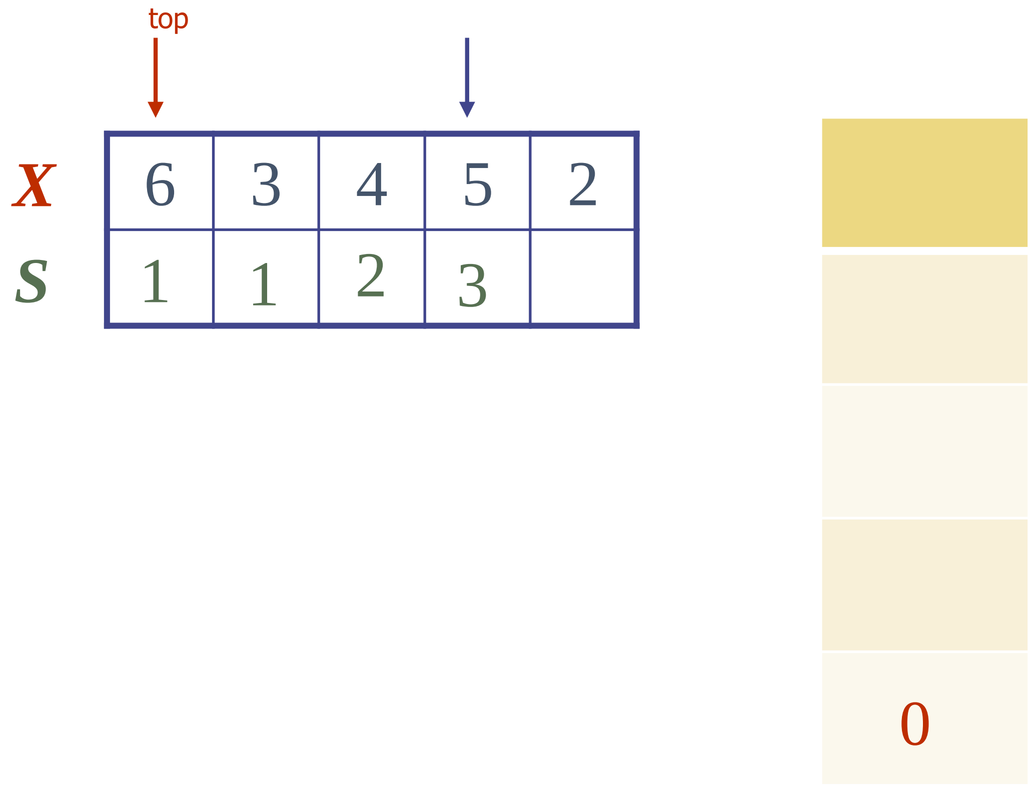
```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
            A.pop()
        if A.empty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Mojtaba Khalili

Example: Computing Spans

IUT-ECE

Algorithm: span2

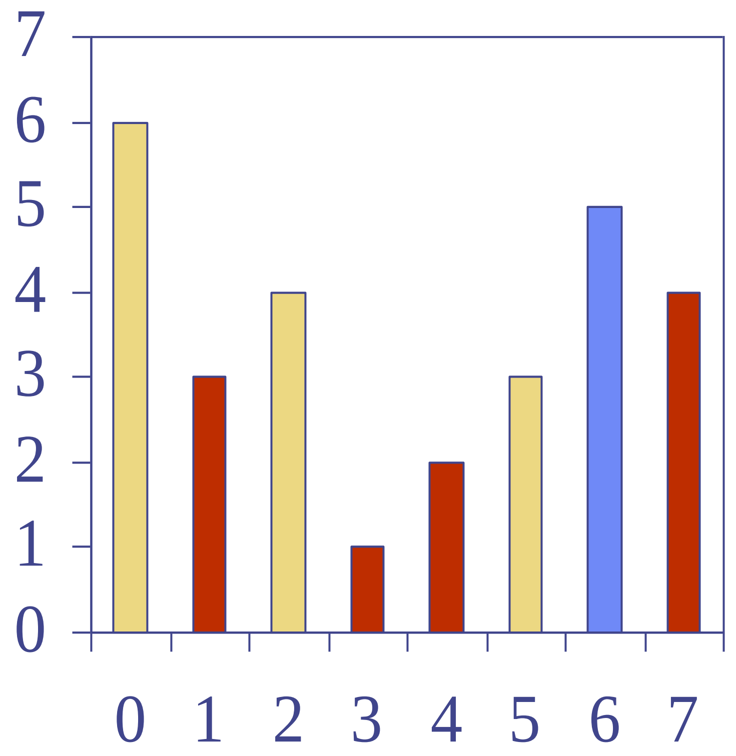


Stack for "index"

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
     for i ← 0 to n − 1 do
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do
           A.pop()
        if A.empty() then
           S[i] ← i + 1
        else
           S[i] ← i − A.top()
        A.push(i)
    return S
```

Example: Computing Spans

- We keep in a stack the indices of the elements visible when “looking back”
- We scan the array from left to right
  - Let $\boldsymbol{i}$ be the current index
  - We pop indices from the stack until we find index $\boldsymbol{j}$ such that $\boldsymbol{X}[\boldsymbol{i}] < \boldsymbol{X}[\boldsymbol{j}]$
  - We set $\boldsymbol{S}[\boldsymbol{i}] \leftarrow \boldsymbol{i} - \boldsymbol{j}$
  - We push $\boldsymbol{x}$ onto the stack



Example: Computing Spans

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most ***n*** times
- Algorithm ***spans2*** runs in ***O***(***n***) time

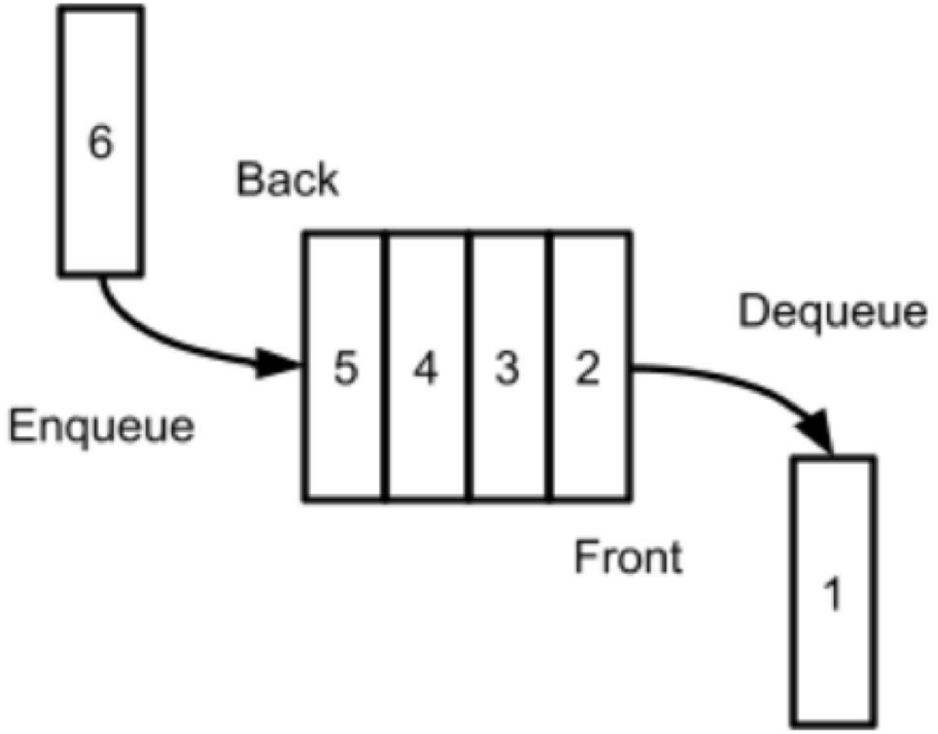
```
Algorithm spans2(X, n)                                  #
    S ← new array of n integers                         n
    A ← new empty stack                                 1
    for i ← 0 to n − 1 do                               n
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do                  n
            A.pop()                                     n
        if A.empty() then                               n
            S[i] ← i + 1                                n
        else
            S[i] ← i − A.top()                          n
        A.push(i)                                       n
    return S                                            1
```

Queue

Another fundamental data structure is the ***queue***, which is a close relative of the stack. A queue is a container of elements that are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the ***rear*** and are removed from the ***front***. The metaphor for this terminology is a line of people waiting to get on an amusement park ride. People enter at the rear of the line and get on the ride from the front of the line.

Queue

IUT-ECE



The Queue ADT (§5.2)

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - dequeue(): removes the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

مثال

Operation	***Output***	*front* ← *Q* ← *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*“error”*	()
empty()	*true*	()

Queue Interface in C++

- C++ interface corresponding to our Queue ADT
- Requires the definition of exception QueueEmpty
- Often dequeue returns an object

```
template <typename E>
class Queue {
public:
   int size() const;
   bool empty() const;
   const E& front() const
      throw(QueueEmpty);
   void enqueue (const E& e);
   void dequeue()
      throw(QueueEmpty);
};
```

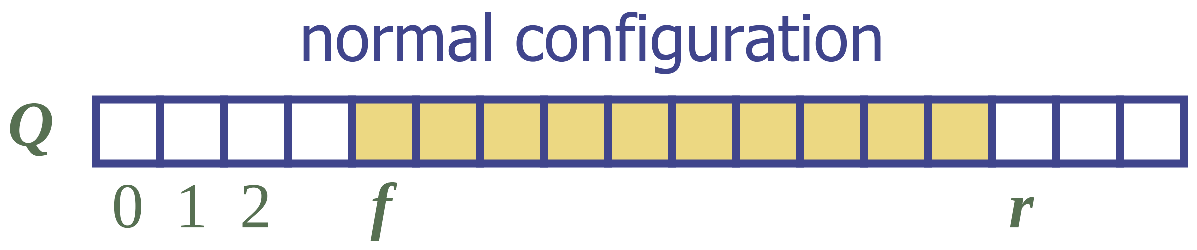
Array-based Queue?

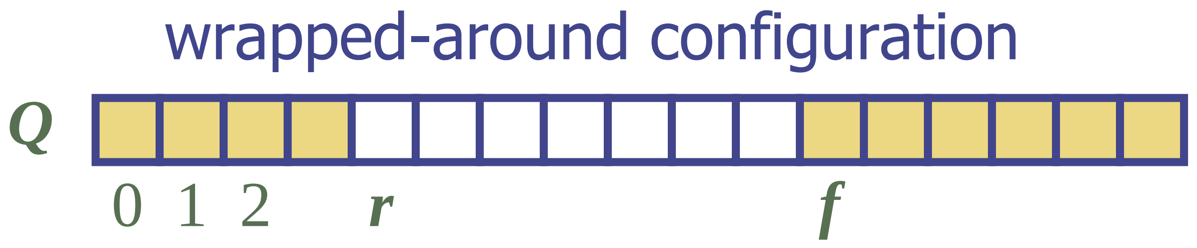
IUT-ECE

- اگر ابتدای آرایه را first صف در نظر بگیریم؟
- هزینه dequeue؟ O(n)
- برعکس چه؟

Array-based Queue

- Use an array of size N in a circular fashion
- Three variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
  - n number of items in the queue





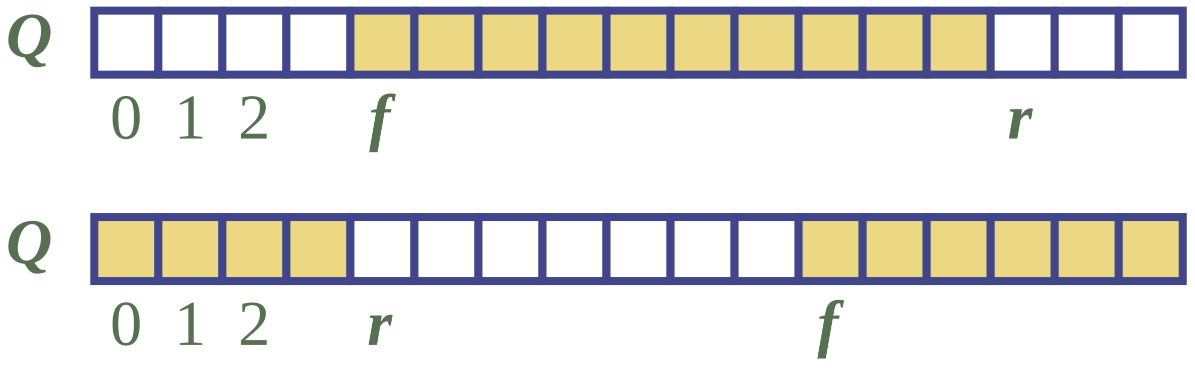
Queue Operations

IUT-ECE

- Use *n* to determine size and emptiness

```
Algorithm size()
    return n

Algorithm empty()
    return (n = 0)
```

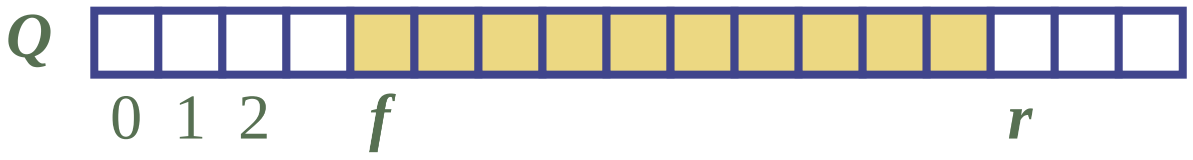


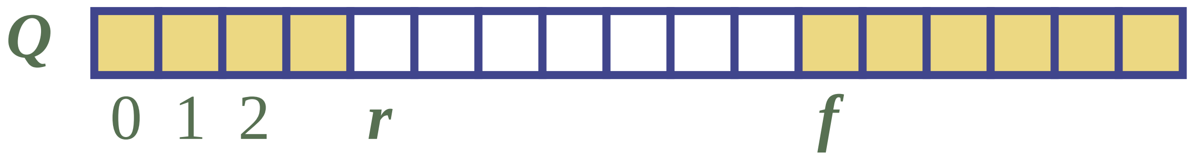
Queue Operations

IUT-ECE

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

```
Algorithm enqueue(o)
  if size() = N – 1 then
    throw QueueFull
  else
    Q[r] ← o
    r ← (r + 1) mod N
    n ← n + 1
```

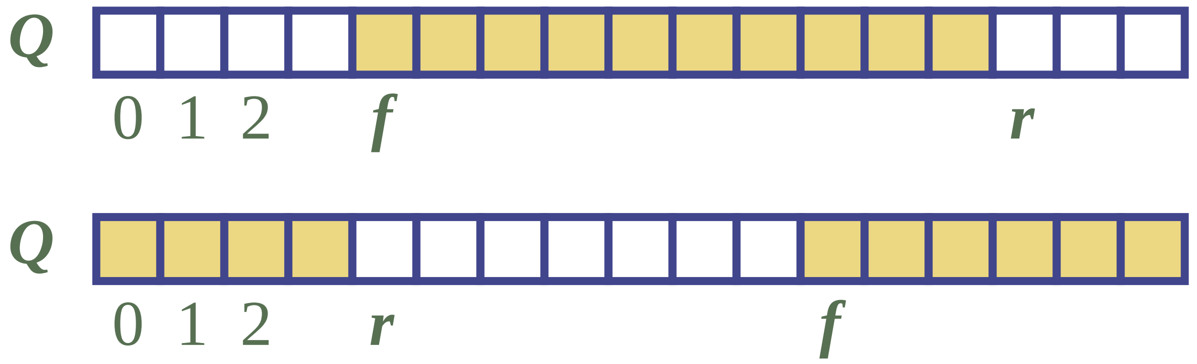




Queue Operations

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if empty() then
    throw QueueEmpty
  else
    $f \leftarrow (f + 1) \bmod N$
    $n \leftarrow n - 1$
```



Queue in C++ STL

```
#include <queue>
using std::queue;                        // make queue accessible
queue<float> myQueue;                    // a queue of floats
```

size():	Return the number of elements in the queue.
empty():	Return true if the queue is empty and false otherwise.
push(e):	Enqueue e at the rear of the queue.
pop():	Dequeue the element at the front of the queue.
front():	Return a reference to the element at the queue's front.
back():	Return a reference to the element at the queue's rear.

محدودیت‌های Queue

- اگر عنصری را از اول صف خارج کردیم اما نیاز شد دوباره به اول برش گردانیم؟
- اگر عنصری وارد ته صف کردیم اما خواستیم حذفش کنیم؟

Double-Ended Queues

- A queue-like data structure that supports insertion and deletion at both the front and the rear of the queue.

IUT-ECE

DEQUE in C++ STL

```
#include <deque>
using std::deque;                       // make deque accessible
deque<string> myDeque;                  // a deque of strings
```

size():	Return the number of elements in the deque.
empty():	Return true if the deque is empty and false otherwise.
push_front(e):	Insert e at the beginning the deque.
push_back(e):	Insert e at the end of the deque.
pop_front():	Remove the first element of the deque.
pop_back():	Remove the last element of the deque.
front():	Return a reference to the deque's first element.
back():	Return a reference to the deque's last element.

How to implement DEQUE?

- Question
  - Which (elementary) data structure are you going to use to implement DEQUE?
    - Array, singly linked list, doubly linked list, circular linked list
  - What happens if you use others?
- Deque by a doubly linked list

Operation	***Time***
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$