Algorithms and Data Structures 2

11 – Abstract data types

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Abstract data types**
  - Definition
  - Operations
  - Implementations
- **Stack**
  - Array implementation
  - Resizable array implementation
  - Linked list implementation

Abstract Data Types (ADTs)

- **Used to abstract the structure of data from the data itself**

- **An ADT specifies**
  - A user-defined data type
  - Operations on that data type

- **Examples**
  - Set, Multiset (bag)
  - List
  - Stack
  - Queue, Priority queue, Double ended queue

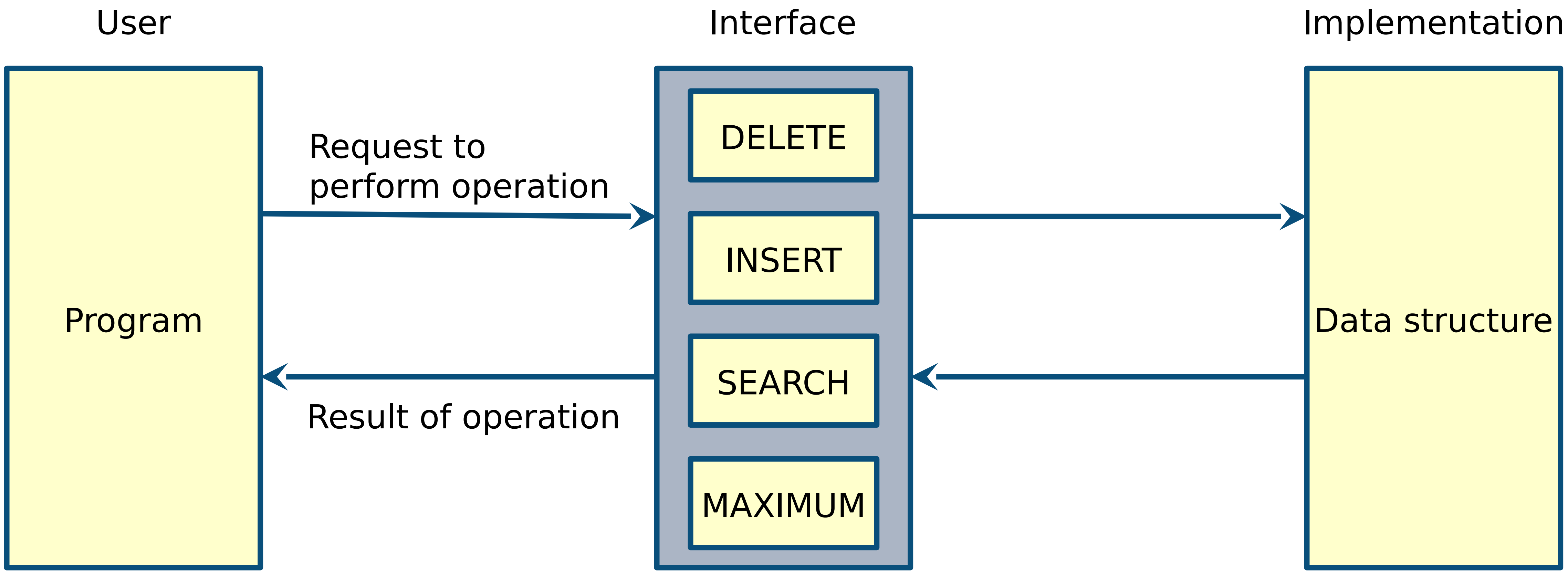
ADTs vs data structures

- **An ADT is a class of objects whose logical behaviour is defined by a set of values and a set of operation**
  - User's point of view

- **Data structures are concrete representations of data and implementations of the procedures for its manipulation**
  - Implementer's point of view

- **Data structures serve as the basis for ADTs**
  - The ADT defines the logical form of the data type
  - The data structure implements the physical form of the data type

ADTs vs data structures



Wall of ADT operations

Stack

- **The Stack ADT stores arbitrary elements**
- **Insertions and deletions follow the LIFO (last-in-first-out) policy**

- **Main stack operations**
  - PUSH(S,x): insert element x in stack S
  - POP(S): remove and return the most recently inserted element from stack S

- **Auxiliary stack operations**
  - PEEK(S): return the most recently inserted element from stack S (sometimes called TOP(S))
  - STACK-SIZE(S): return the number of elements stored in stack S
  - STACK-EMPTY(S): test whether no elements are stored in stack S

Stack

- **Direct applications**
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Syntax parsing

- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures

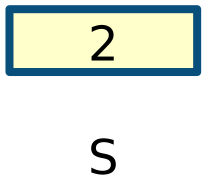
Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

S

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

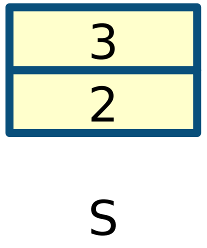


2

S

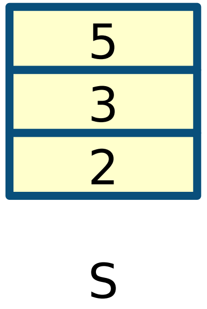
Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



5

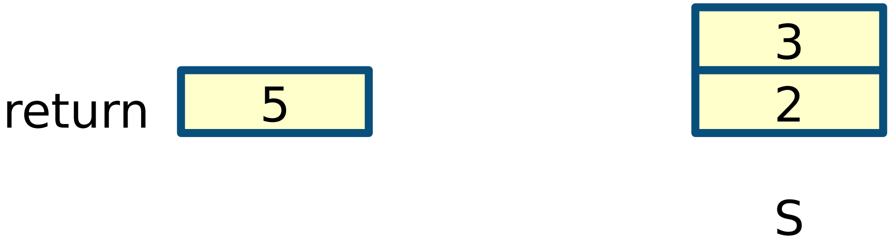
3

2

S

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



return 5

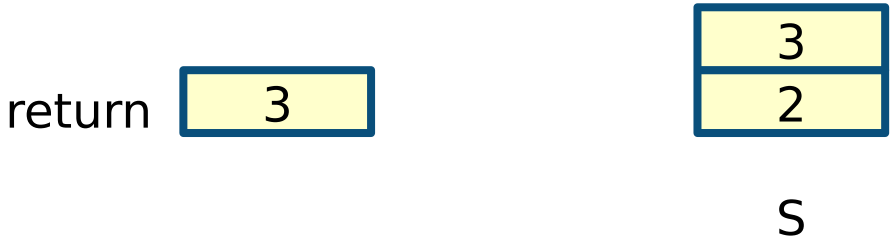
3

2

S

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



return 3

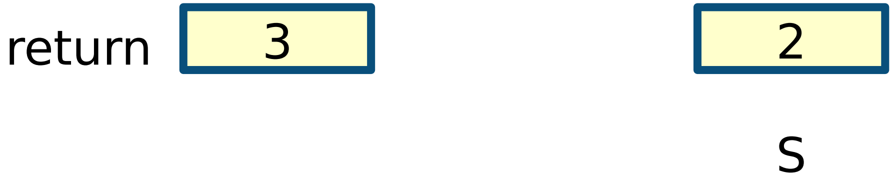
3

2

S

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



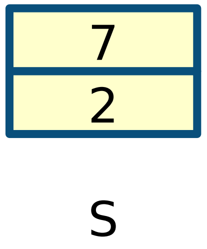
return 3

2

S

Example

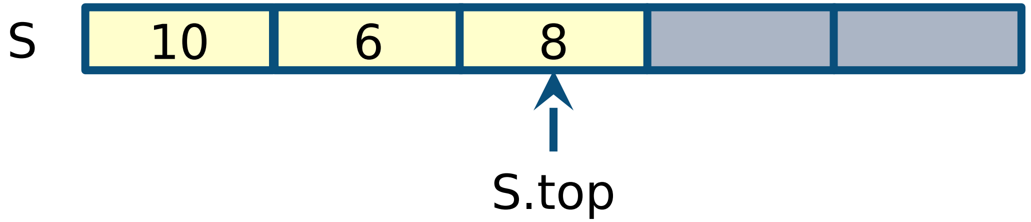
- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)



Array implementation

- **A simple way of implementing a bounded stack is to use an array**
  - Add elements from left to right
  - An attribute S.top keeps track of the index of the top element

- **Array S[0..n-1] implements a stack of at most n elements**
- **The stack consists of subarray S[0..S.top] where S.top < n**
  - S[0] is the element at the bottom of the stack
  - S[S.top] is the element at the top



Operations

- **Operations on the stack add/remove elements from the right end of the array and update S.top**
  - When S.top = -1 the stack is empty

- **The array storing the stack elements may become full/empty**
  - If we push into a full stack, the stack overflows
  - If we try to pop an empty stack, the stack underflows

- **Overflows are limitation of the array-based implementation not intrinsic to the Stack ADT**
  - In our pseudocode we will ignore stack overflows

Operations

```
STACK-EMPTY(S)
  return S.top = -1
```

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

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else S.top := S.top - 1
    return S[S.top + 1]
```

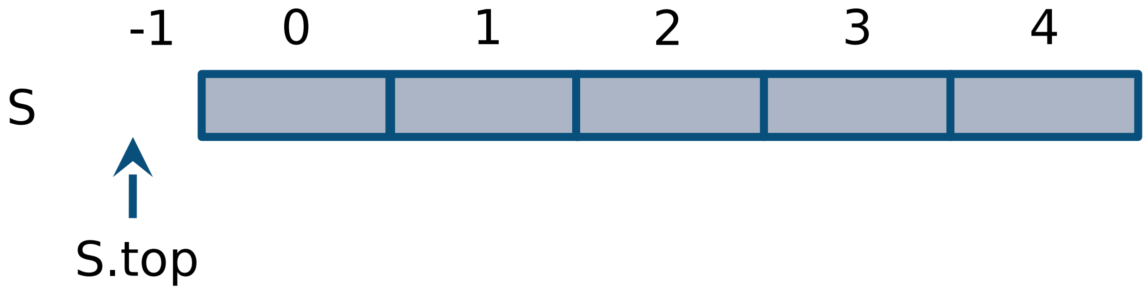
Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

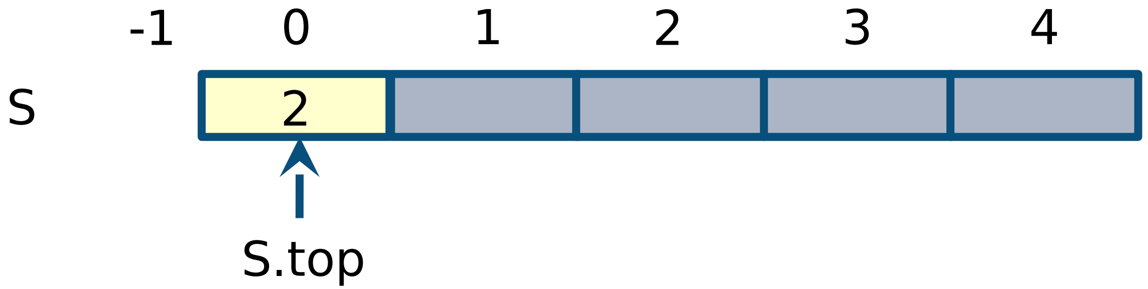
Initialise S
S can contain at most 5 elements



Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

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



	-1	0	1	2	3	4
S		2				

S.top

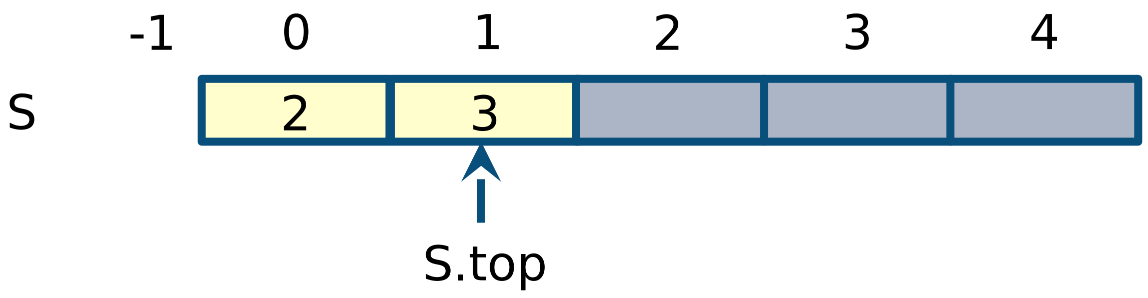
S.top is incremented
Element 2 is stored in the array

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

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

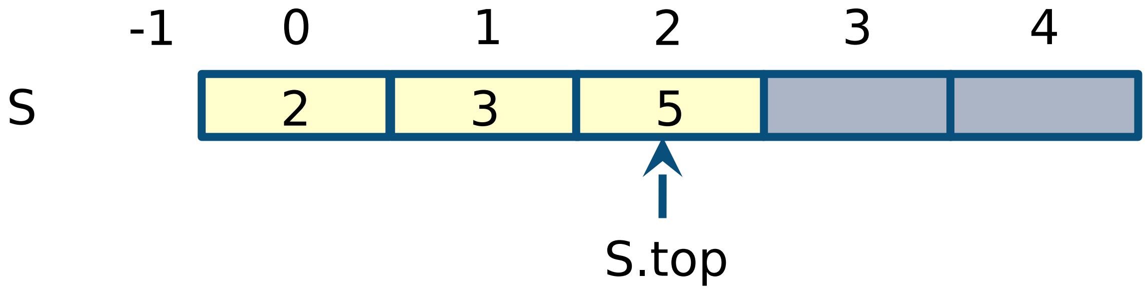
S.top is incremented
Element 3 is stored in the array



Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

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



	-1	0	1	2	3	4
S		2	3	5		

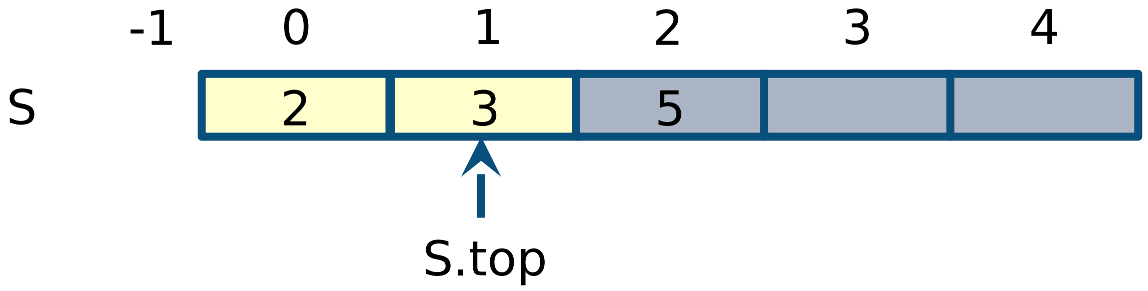
S.top is incremented
Element 5 is stored in the array

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

S.top is decremented
Element 5 is returned

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else S.top := S.top - 1
    return S[S.top + 1]
```



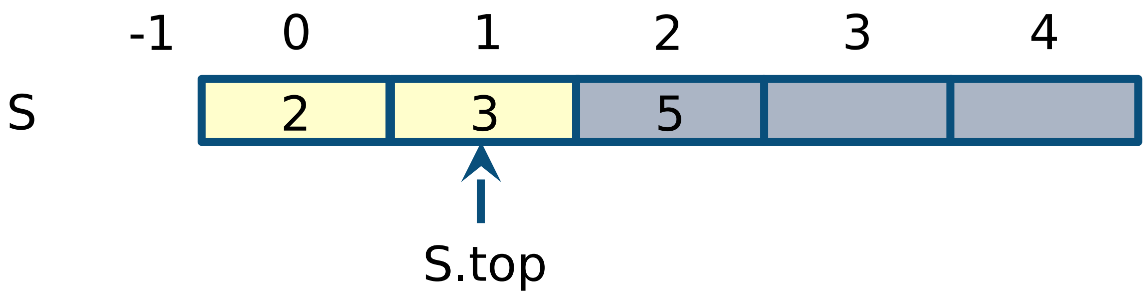
	-1	0	1	2	3	4
S		2	3	5		

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
PEEK(S)
  if STACK-EMPTY(S)
    error “underflow”
  else
    return S[S.top]
```

Element 3 is returned

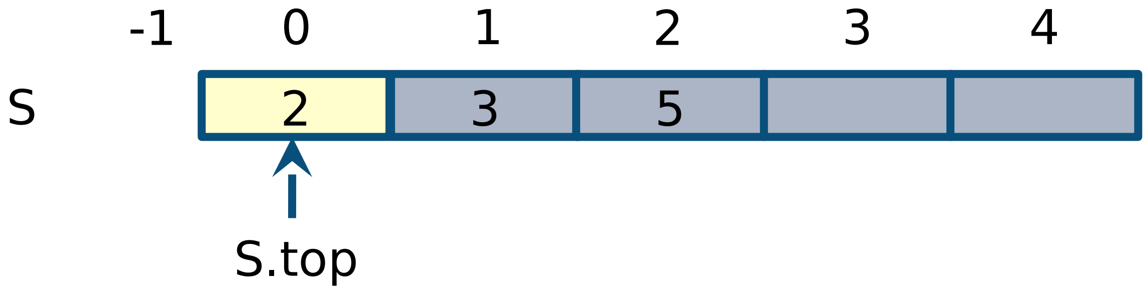


Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else S.top := S.top - 1
    return S[S.top + 1]
```

S.top is decremented
Element 3 is returned



	-1	0	1	2	3	4
S		2	3	5		

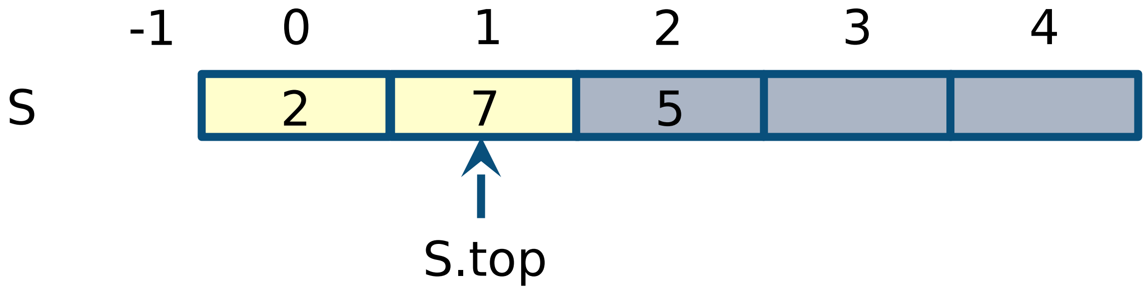
S.top

Example

- **What is the stack formed by carrying out the following sequence of instructions?**
  - PUSH(S,2)
  - PUSH(S,3)
  - PUSH(S,5)
  - POP(S)
  - PEEK(S)
  - POP(S)
  - PUSH(S,7)

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

S.top is incremented
Element 7 is stored in the array



	-1	0	1	2	3	4
S		2	7	5		

S.top

Performance and limitations

- **Let n be the size of the array**
  - The space used is O(n) (independent of number of elements in the stack)
  - Each operation runs in time O(1)

- **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

Resizable array implementation

- **Same as the implementation with normal arrays but the size of the underlying array can grow or shrink**
  - No overflows
  - Memory requirement is O(cs) where c is a constant and s is the number of elements in the stack

- **Simple implementation**
  - Double the underlying array when it is full
  - Half the underlying array when it is one-quarter full (c=4)

- **Expanding the array by a constant proportion ensures that inserting n elements takes O(n) time overall**
  - Each insertion takes amortised constant time

Operations

```
RESIZE(S,n’) // n’ is the new capacity
  new S’[0..n’-1]
  for i = 0 to S.top
    S’[i] := S[i]
  S := S’
```

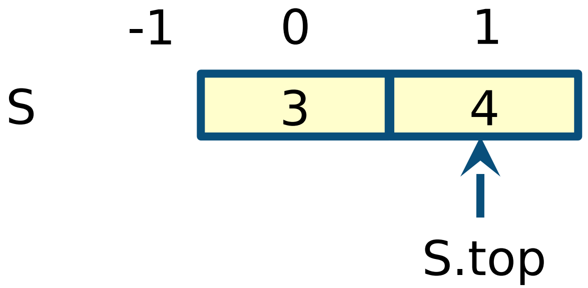
```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else
    x := S[S.top]
    S.top := S.top – 1
    if S.top > 0 and S.top = n/4
      RESIZE(S,n/2)
    return x
```

```
PUSH(S,x)
  if S.top = n - 1
    RESIZE(S,2*n)
  S.top := S.top + 1
  S[S.top] := x   // no overflow
```

Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)
  - n = 2

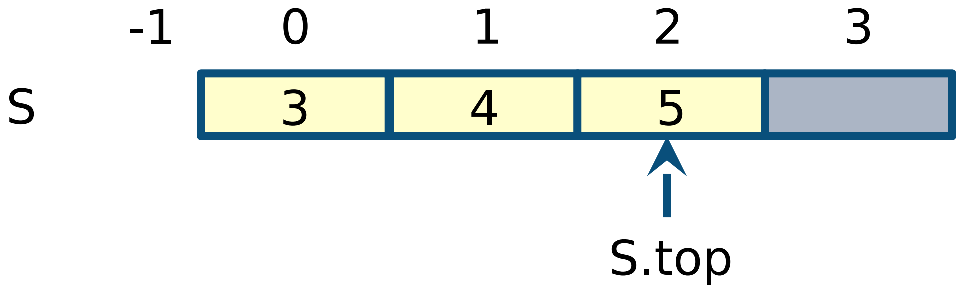
```
PUSH(S,x)
  if S.top = n - 1
    RESIZE(S,2*n)
  S.top := S.top + 1
  S[S.top] := x   // no overflow
```



Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)
  - After resizing n = 4

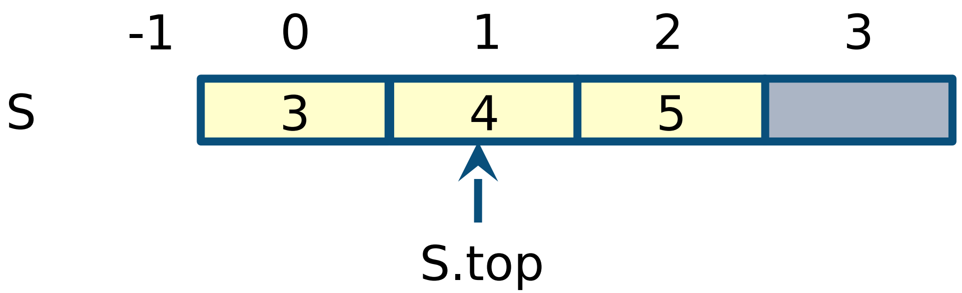
```
PUSH(S,x)
  if S.top = n - 1
    RESIZE(S,2*n)
  S.top := S.top + 1
  S[S.top] := x   // no overflow
```



Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)
  - n = 4 and x = 5
  - After decrementing S.top we resize

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else
    x := S[S.top]
    S.top := S.top – 1
    if S.top > 0 and S.top = n/4
      RESIZE(S,n/2)
    return x
```



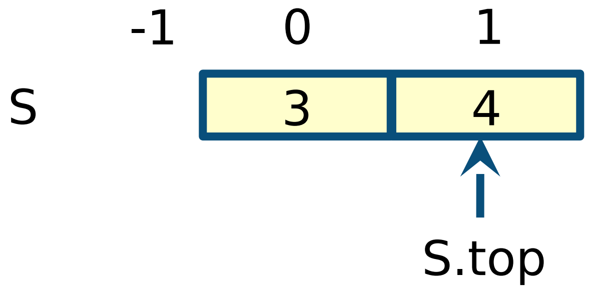
	-1	0	1	2	3
S		3	4	5	

S.top

Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)
  - After resizing n = 2
  - Return x = 5

```
POP(S)
  if STACK-EMPTY(S)
    error “underflow”
  else
    x := S[S.top]
    S.top := S.top – 1
    if S.top > 0 and S.top = n/4
      RESIZE(S,n/2)
    return x
```



In practice, we shrink only up to a given threshold to avoid repeated resizing occurring when the array is too small

Amortised analysis

- **Analysis technique in which the average of running times is considered**

- **Example: consider n+1 push operations on a stack of size n**
  - The first n operations are O(1)
  - Operation n+1 is O(n) because it requires to resize the array (allocate a new array and copy over n values)

- **The amortised running time of the push operation is obtained by taking the average of n+1 push operations**
  - Sum of the running times of each operation divided by total number of operations
  - $\underbrace{(O(1) + O(1) + \ldots + O(1)}_{\text{n times}} + O(n))/(n+1) = (nO(1) + O(n))/(n+1) = O(n)/O(n) = O(1)$

Linked list implementation

- **The stack ADT can be easily implemented with linked lists**
  - L.head implements S.top
  - PUSH is implemented by INSERT at the head
  - POP is implemented by DELETE-HEAD
- **Both operations can be performed in constant time (see Lecture 10)**
  - To perform operation STACK-SIZE in constant time, keep track of size with attribute S.size
- **No overflows as new elements are dynamically allocated**

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

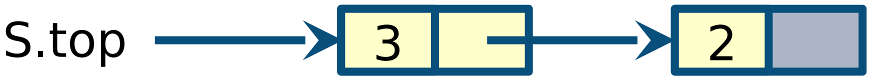
```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```

Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```

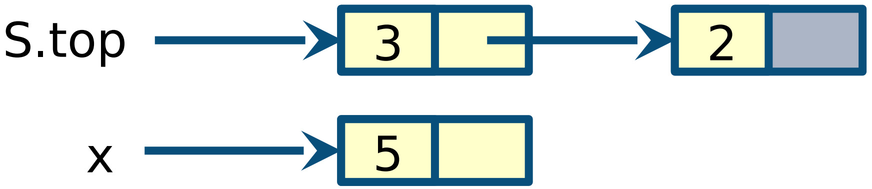


Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```

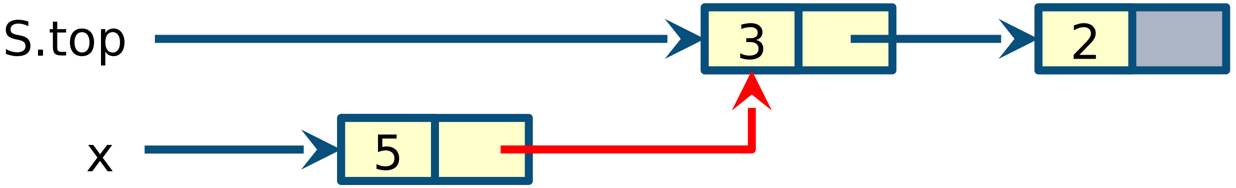


Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```

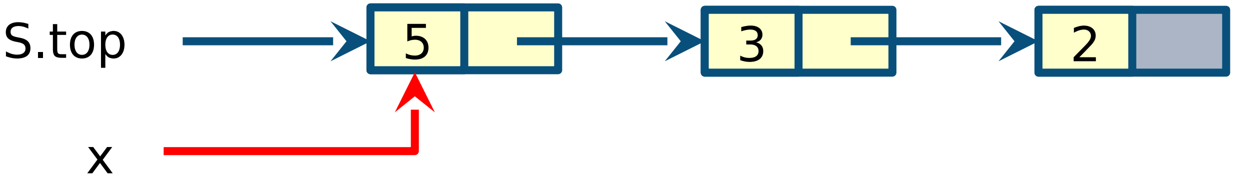


Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

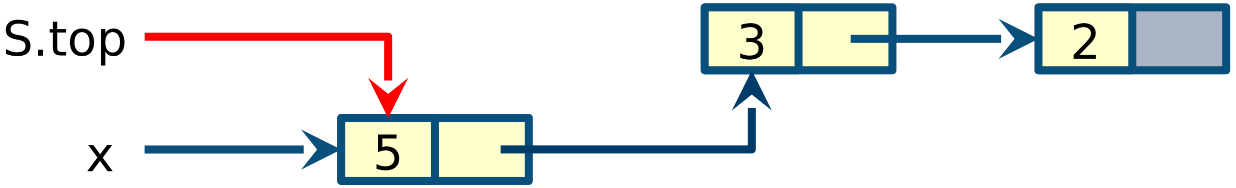
```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```



Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```



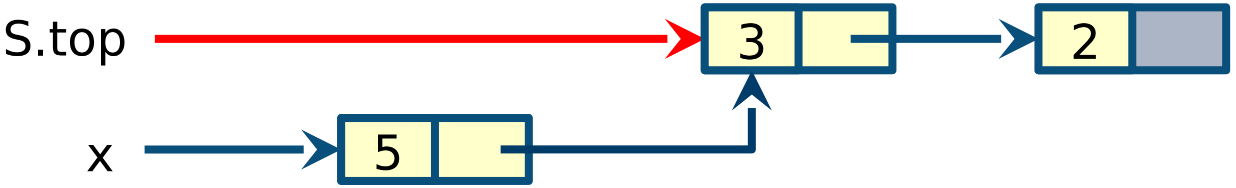
```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error “underflow”
```

Example

- **We perform the following operations on the stack below**
  - PUSH(S,5)
  - POP(S)

```
PUSH(S,x)
  x.next := S.top
  S.top := x
```

```
POP(S)
  if S.top != NIL
    x := S.top
    S.top := S.top.next
    return x
  else
    error "underflow"
```



Summary

- **Abstract data types**
  - Definition
  - Operations
  - Implementations
- **Stack (LIFO)**
  - Array implementation
  - Resizable array implementation
  - Linked list implementation