Stacks

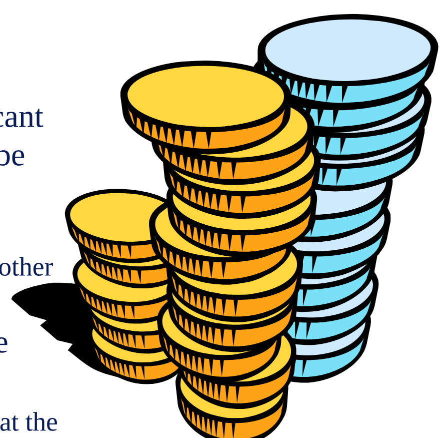
Overview

- A stack is a data structure for storing…
  - ...an ***ordered*** collection of items…
  - ...such that items can only be inserted and removed at ***one end*** (called the ***top***)
- A stack is a ***last-in-first-out*** (**LIFO**) data structure
  - Entries are taken out of the stack in the ***reverse*** order of their insertion
- Another name for a stack is a ***pushdown store***

Stacks

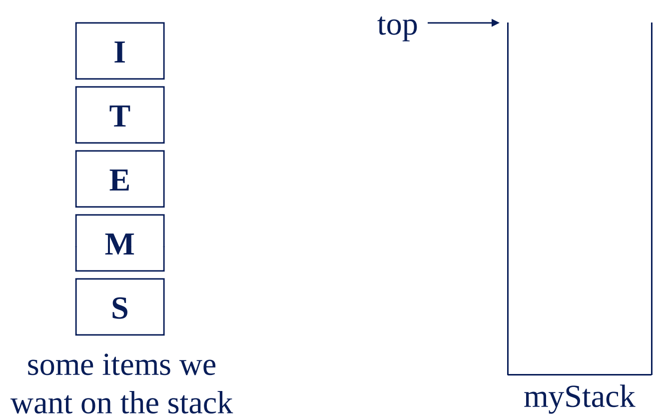
Overview

- When we say that a stack is "ordered"…
  - …we mean that there is ***some positional order*** that is significant with respect to how items can be ***accessed***
    - *I.e.*, there is an item that can be accessed first, another second, another third, and so forth
  - …we ***don't*** mean that items are sorted
    - In fact, it is not even necessary that the items are sortable (can be compared)



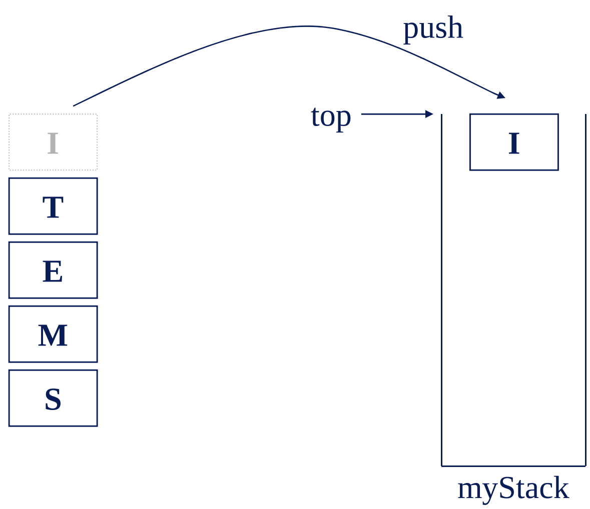
Stacks

Basic Concepts



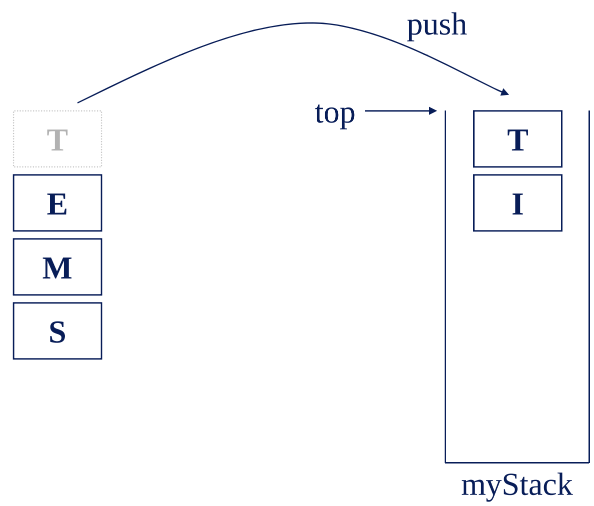
Stacks

Basic Concepts
Pushing Items onto a Stack



Stacks

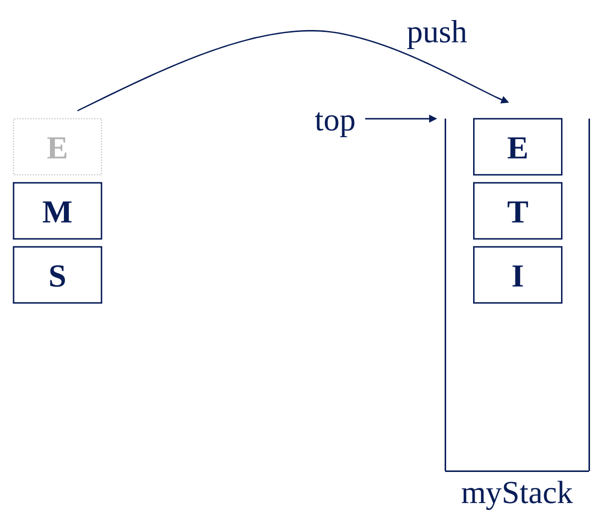
Basic Concepts
Pushing Items onto a Stack



5

Stacks

Basic Concepts
Pushing Items onto a Stack

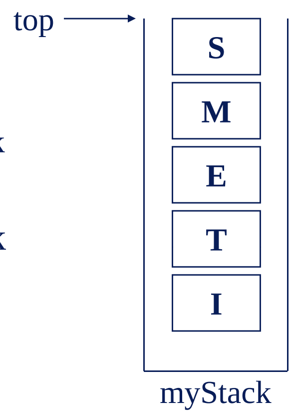


6

Stacks

Basic Concepts
Pushing Items onto a Stack

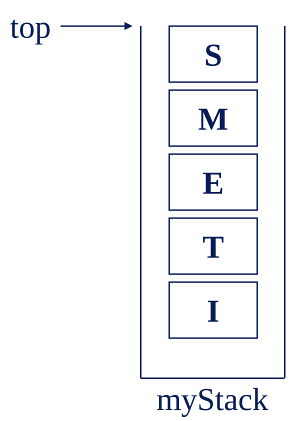
- After all the items are ***push***ed onto the stack...
  - ...they are in ***reverse*** order
  - ...the ***first*** item that was pushed onto the stack is at the ***bottom*** of the stack
- The ***top*** always has the item that is the ***last*** to be pushed onto the stack
  - ...relative to the other items that are still part of the stack



Stacks

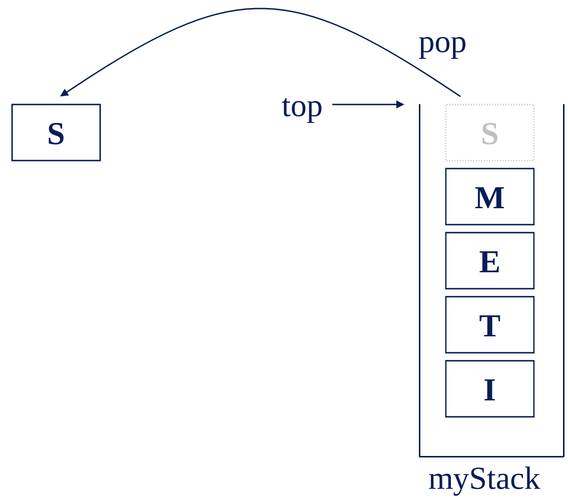
Basic Concepts
Popping Items off a Stack

- Items are removed from a stack <u>ONLY</u> at the ***top***...
  - ...the ***first item popped*** off the stack is thus the ***last item pushed*** onto the stack
  - ...in our illustration, **S** is the item that will be popped off the top



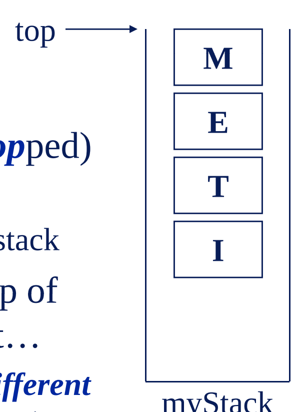
Stacks

Basic Concepts
Popping Items off a Stack



Stacks

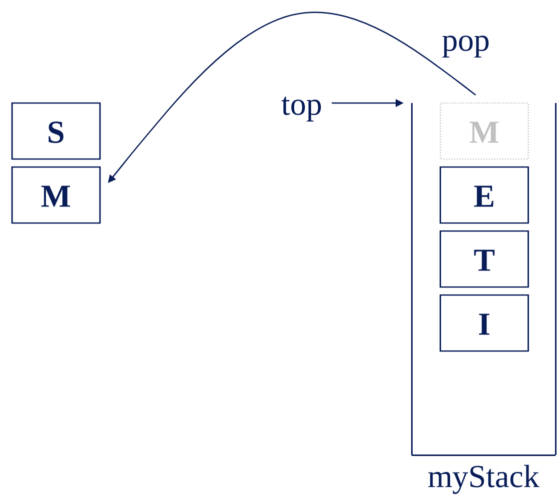
Basic Concepts
Popping Items off a Stack



- When an item is removed (***pop***ped) from a stack…
  - ...it is ***no longer a part*** of the stack
- To examine the item at the top of the stack ***without*** removing it…
  - ...would require the use of a ***different*** operation (that may be called ***top***)

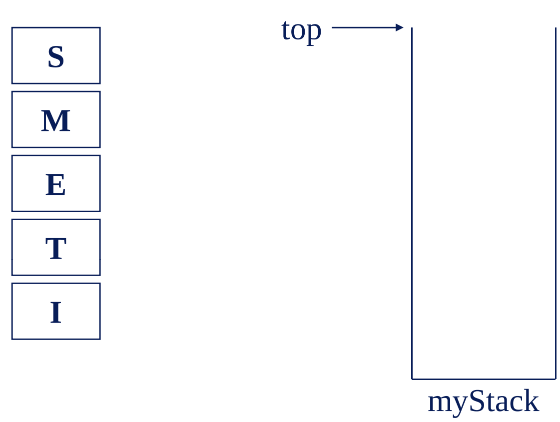
Stacks

Basic Concepts
Popping Items off a Stack



Stacks

Basic Concepts
Popping Items off a Stack



Stacks

Basic Concepts
Accessing the Stack

- At any one time ...
  - …we can only access the ***top item*** of the stack
- If you need to access the third item (say) ...
  - …we must first pop the two that are "above" it
- Textbook authors use as an analogy...
  - …a PEZ dispenser which holds little rectangular hard candies
  - …to get the third candy you have to remove the first two from the dispenser

Stacks

Stack Errors

- There are ***two common errors*** that can occur when accessing a stack:
  - ***Stack Underflow***
    - The condition resulting from trying to ***pop*** (remove) an item from an ***empty*** stack
  - ***Stack Overflow***
    - The condition resulting from trying to ***push*** an item (add an item to) a ***full*** stack

Stacks — Stack Errors

- In order to avoid a stack underflow…
  - ...a stack ADT should provide a function that tests whether a stack is empty
    - Appropriately, this would be a public member function in a C++ class
- In order to avoid a stack overflow…
  - ...we should check that the stack is not full before pushing a new item
    - This implies a *fixed capacity* stack
    - We could use a function that tests whether a stack is full
      - Implies that such a function is available
    - We could also compare the ***stack capacity*** with its ***current size***
      - Implies that we have the means to obtain the two values

Stacks — Stack Template Class

- Since we are likely to need stacks of different kinds of things…
  - ...it makes sense to implement the stack with a template class
- Two implementations of a stack template class…
  - ...are shown in Section 7.3 of the textbook
  - ...one using a static *array*, the other using a *linked list*
- The stack template class implements some general operations required of a stack

Stacks

Stack Template Class

- The stack template class has the following member functions:
  - constructor → creates an empty stack
  - **`void push(const Item& entry)`**
  - **`void pop()`**
  - **`Item top() const`**
  - **`size_type size() const`**
  - **`bool empty() const`**

Stacks

Usefulness of Stacks

- Although a stack is a very simple concept, it has several very useful applications → examples:
  - ***Run-time stack***
    - When one function calls another, the state (activation record) of the calling function gets pushed onto the run-time stack
    - When a function ends, control returns to the function that called it by popping the calling function's state off the stack
  - Stacks are also used by compilers to check the syntax of source code
  - ...

Stacks

- One simple application of a stack is…
  - ...to reverse the order of the characters in a word
    - Example: if the user typed in ESIOTROT, the program would output TORTOISE
- While this is a trivial example…
  - ...it demonstrates the common mechanism for putting a stack to use

Stacks

- A pseudocode for reversing a word:

```
While (there are more characters of the word to read)
    Read a character and push it onto the stack
While (the stack is not empty)
    Output the top character and pop it off the stack
```

- This is exactly the process…
  - …portrayed in the illustration shown earlier
  - ...where the word involved is **ITEMS**

- An application that is closely related to evaluating arithmetic (and Boolean) expressions is…
  - ...checking for ***balanced parentheses***
  - ...which means that for every `'('` in an expression there is a matching `')'`
- One way of doing it is…
  - ...to use stacks
  - ...but there are alternatives

- Consider the string `(X+Y*(Z+7))*(A+B))`
  - Each left parenthesis has a corresponding right parenthesis
- The `is_balanced` function would take a string like this as an argument and...
  - ...return ***true*** if the parentheses balance
  - ...return ***false*** if they don't

Stacks

Putting Stacks to Use
Checking for Balanced Parentheses

- The algorithm is simple...

```
Scan the characters in the string from left to right
    if (character read is a left parenthesis)
      Push it onto the stack
    else if  (character read is a right parenthesis)
      Pop a left parenthesis off the stack
    else
      Do nothing (i.e., simply discard the character read)
```

- *(continued)*

Stacks

Putting Stacks to Use
Checking for Balanced Parentheses

- The parentheses are ***balanced***...
  - …if the stack is ***empty*** when the **`is_balanced`** function has read the entire string
- The parentheses are ***not balanced***…
  - …if the **`is_balanced`** function produces a stack ***underflow*** error during processing
    - I.e., algorithm dictates a pop but stack is found to be empty
  - …if there are any ***left parentheses left in the stack*** (stack not empty) when the **`is_balanced`** function has read and processed the entire string
    - I.e., stack is not empty when the algorithm ends

Stacks

Putting Stacks to Use
Checking for Balanced Parentheses

Input: **(( ) ( ))**

Initial empty stack	Read and push first (	Read and push second (	Read first) and pop matching (	Read and push third (	Read second) and pop matching (	Read third) and pop the last (
	(	(	(	(	(	
		(		(		

Stacks

Putting Stacks to Use
Evaluating Arithmetic Expressions

- Suppose we want a program that would evaluate an arithmetic expression such as...
  `((((12 + 9) / 3) + 7.2) * (( 6 - 4) / 8))`
- The expression will consist of...
  - ...non-negative integer or floating-point numbers
  - ...along with the basic arithmetic operators (+, -, *, /)
  - ...as well as the parentheses themselves

Additional Assumptions

- The expression is ***formed correctly***
  - (expression operator expression)
    - An expression can be an integer or floating-point number or another expression
    - An operator can be +, -, *, /
- Each expression is ***fully parenthesized***
  - Each operation has a pair of matching parentheses surrounding its arguments

Stacks
Putting Stacks to Use
Evaluating Arithmetic Expressions

Consider How We'd Do It Manually

- (((6 + 9) / 3) * (6 - 4))
  - Evaluate the ***innermost*** expressions
    - (6 + 9) and (6 -4)
- ((15 / 3) * 2)
  - Evaluate the ***innermost*** expression
    - (15 / 3)
- (5 * 2)
  - Evaluate the ***innermost*** expression → 10

Algorithm Design Considerations

- Trying to keep track of ***innermost*** expressions is *complicated* and *unnecessary*
- We can simply evaluate the ***leftmost of the innermost*** expressions each time through
  - ...thus eliminating the need to keep track
- We keep evaluating the ***leftmost innermost*** expression until we are finished

Previous Example Revisited

- ((**(6 + 9)** / 3) * (6 - 4))
- (**(15 / 3)** * (6 - 4))
- (5 * **(6 - 4)**)
- **(5 * 2)**
- 10

Algorithm Design Considerations

- The ***end*** of the ***leftmost of the innermost expressions*** is always indicated by a ***right parenthesis***
  - Useful for us to find the expression to be evaluated next
- We will use two stacks:
  - One stack contains ***numbers*** and ***intermediate values***
    - We will refer to it as the *numbers stack*
  - The other stack contains ***operators*** from the input
    - We will refer to it as the *operators stack*

Algorithm Design Considerations

- Since a stack is a LIFO structure...
  - ...it will turn out that the ***correct two numbers*** are on ***top of the numbers stack***...
  - ...at the same time that the ***appropriate operation*** is at the ***top of the operators stack***

Algorithm: Description/Example

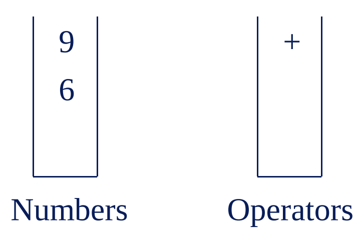
- Start by reading the input ***up until the first right parenthesis***
  - Each ***number*** along the way is ***pushed onto the numbers stack***
  - Each ***operator*** along the way is ***pushed onto the operators stack***
  - (left parentheses are simply discarded)
  - (each right parenthesis serves as a flag for some "computational" actions to be taken – to be described)

Algorithm: Description/Example

(((6 + 9) / 3) * (6 - 4))

↑

first right parenthesis



Algorithm: Description/Example

- Each time we see a ***right parenthesis***, we combine the top 2 numbers (from the numbers stack) with the top operator (from the operators stack)
  - **Continuing with our example: 6 + 9**
    - Note that the ***left*** argument <u>should be</u> the ***second*** number removed from the (numbers) stack → makes a big difference with some operators (*e.g.*, "6 - 9" vs "9 - 6" or "6/9" vs "9/6")
- Perform the operation to get the ***intermediate result*** (6+9 = 15), and ***push it back onto the numbers stack***

Algorithm: Description/Example

(((6 + 9) / 3) * (6 - 4))

next right parenthesis

Numbers	Operators
3	/
15	

Algorithm: Description/Example

(((6 + 9) / 3) * (6 - 4))

↑

next right parenthesis

Numbers	Operators
4	-
6	*
5	

Algorithm: Description/Example

(((6 + 9) / 3) * (6 - 4))

↑

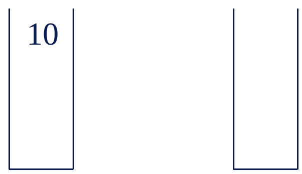
next right parenthesis

Numbers	Operators
2	*
5	

Algorithm: Description/Example

(((6 + 9) / 3) * (6 - 4))

END of input has been reached
Numbers stack contains the desired result as the only item



Numbers Operators

Algorithm: Summary

- When a number is encountered in the input...
  - ...push it onto the numbers stack
- When an operator is encountered in the input...
  - ...push it onto the operators stack
- When a right parenthesis is encountered in the input...
  - ...pop the top two numbers and combine them with the top operation
  - …evaluate the intermediate result and push it onto the numbers stack

Algorithm: Summary

- When a left parenthesis or blank is encountered in the input...
  - ...simply ignore (discard) it
  - (NOTE: A more complete algorithm would need to process the left parentheses in some way to ensure that each left parenthesis is balanced by a right parenthesis → our assumption is that the input is ***correctly formed***)
- When the END of input is encountered...
  - ...pop the numbers stack to obtain the desired result

Textbook Readings

- Chapter 7
  - Section 7.1
  - Section 7.2
  - Section 7.3