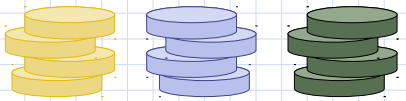
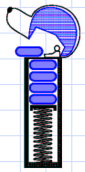
Stacks



Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

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 and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

Example

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]
S.is_empty()	True	[ ]
S.pop()	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language that supports recursion
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- 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

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$ (amortized in the case of a push)

Array-based Stack in Python

```
class ArrayStack:
  """LIFO Stack implementation using a Python list as underlying storage."""

  def __init__(self):
    """Create an empty stack."""
    self._data = [ ]                          # nonpublic list instance

  def __len__(self):
    """Return the number of elements in the stack."""
    return len(self._data)

  def is_empty(self):
    """Return True if the stack is empty."""
    return len(self._data) == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._data.append(e)                      # new item stored at end of list

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._data[-1]                     # the last item in the list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._data.pop( )                  # remove last item from list
```


Parentheses Matching

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

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
  if *X*[*i*] is an opening grouping symbol **then**
  S.push(*X*[*i*])
  else if *X*[*i*] is a closing grouping symbol **then**
  if *S*.is_empty() **then**
  return false {nothing to match with}
  if *S*.pop() does not match the type of *X*[*i*] **then**
  return false {wrong type}
if *S*.isEmpty() **then**
  return true {every symbol matched}
else return false {some symbols were never matched}

Parentheses Matching in Python

```
def is_matched(expr):
  """Return True if all delimiters are properly match; False otherwise."""
  lefty = '({['                               # opening delimiters
  righty = ')}]'                              # respective closing delims
  S = ArrayStack( )
  for c in expr:
    if c in lefty:
      S.push(c)                               # push left delimiter on stack
    elif c in righty:
      if S.is_empty( ):
        return False                          # nothing to match with
      if righty.index(c) != lefty.index(S.pop( )):
        return False                          # mismatched
  return S.is_empty( )                        # were all symbols matched?
```

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm in Python

```
1  def is_matched_html(raw):
2    """Return True if all HTML tags are properly match; False otherwise."""
3    S = ArrayStack( )
4    j = raw.find('<')                  # find first '<' character (if any)
5    while j != −1:
6      k = raw.find('>', j+1)           # find next '>' character
7      if k == −1:
8        return False                   # invalid tag
9      tag = raw[j+1:k]                 # strip away < >
10     if not tag.startswith('/'):      # this is opening tag
11       S.push(tag)
12     else:                            # this is closing tag
13       if S.is_empty( ):
14         return False                 # nothing to match with
15       if tag[1:] != S.pop( ):
16         return False                 # mismatched delimiter
17     j = raw.find('<', k+1)           # find next '<' character (if any)
18   return S.is_empty( )               # were all opening tags matched?
```


Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
  doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

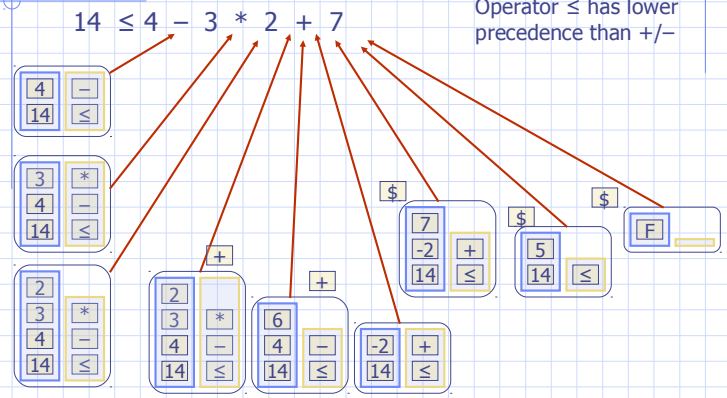
Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

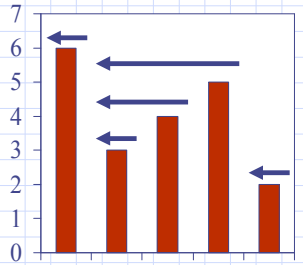
Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/−



Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- 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 $X[j] \le X[i]$
- 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

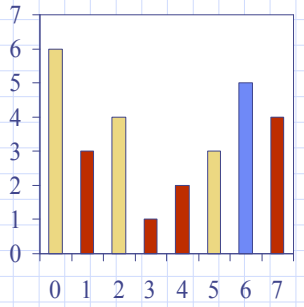
Quadratic Algorithm

```
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

Computing Spans with a Stack

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



Linear Algorithm

- 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.is_empty() ^
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.is_empty() then                    n
            S[i] ← i + 1                        n
        else
            S[i] ← i - A.top()                  n
        A.push(i)                               n
    return S                                    1
```