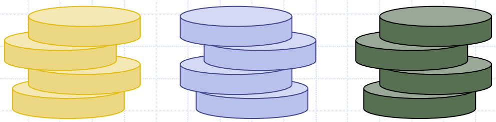
Stacks



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 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

```
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);
}
```

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting pop or top on an empty stack throws a StackEmpty exception

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

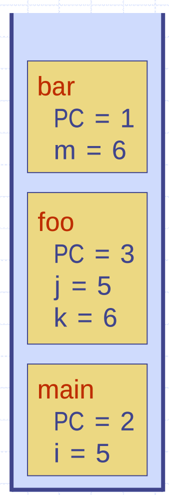
C++ Run-Time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```

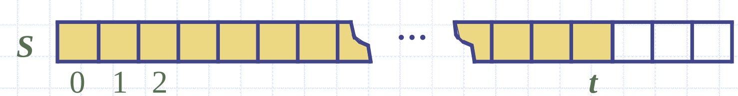


Array-based Stack

- 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

Algorithm *size*()
  return $t + 1$

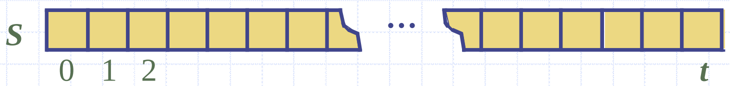
Algorithm *pop*()
  if *empty*() **then**
    throw *StackEmpty*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a StackFull exception
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(o)
  if $t = S.size() - 1$ **then**
    throw *StackFull*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



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 $O(1)$
- 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

Array-based Stack in C++

```
template <typename E>
class ArrayStack {
private:
  E* S; // array holding the stack
  int cap; // capacity
  int t; // index of top element
public:
  // constructor given capacity
  ArrayStack( int c) :
    S(new E[c]), cap(c), t(-1) { }
```

```
  void pop() {
    if (empty()) throw StackEmpty
          (“Pop from empty stack”);
    t--;
  }
  void push(const E& e) {
    if (size() == cap) throw
        StackFull(“Push to full stack”);
    S[++ t] = e;
}
… (other methods of Stack interface)
```


Example use in C++

* indicates top

```
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
```

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*.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*.empty() **then**
  return true {every symbol matched}
else return false {some symbols were never 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

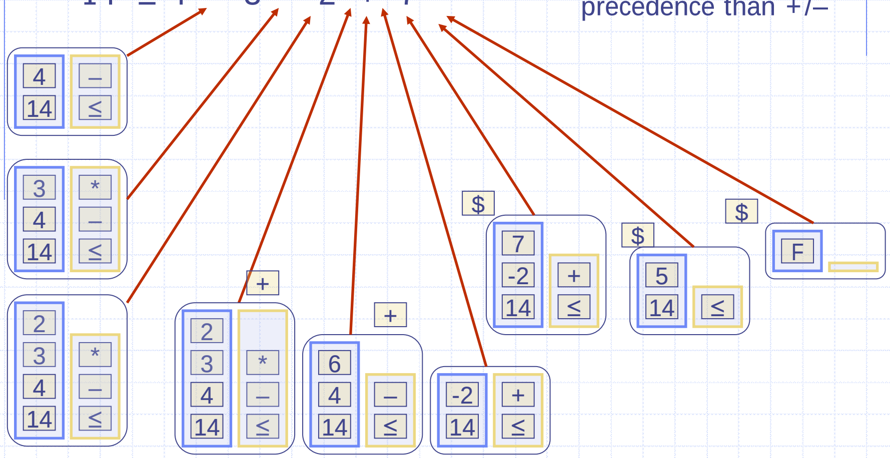
while there's another token z
  if isNumber(z) **then**
    valStk.push(z)
  else
    repeatOps(z);
    opStk.push(z)
repeatOps($);
return valStk.top()

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.

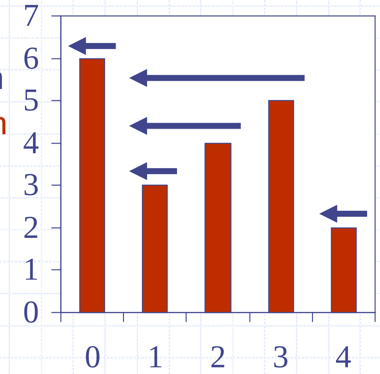
14 ≤ 4 – 3 * 2 + 7

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

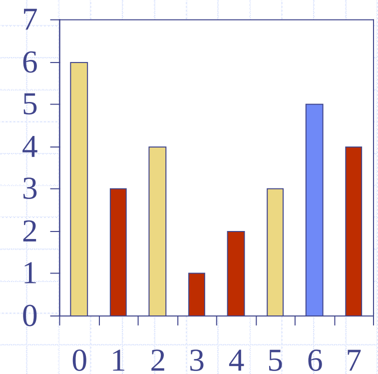
Quadratic Algorithm

Algorithm *spans1*(X, n)	
Input array X of n integers	
Output array S of spans of X	#
$S \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow 1$	n
while $s \le i \wedge X[i - s] \le X[i]$	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + 1$	$1 + 2 + \ldots + (n - 1)$
$S[i] \leftarrow s$	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)	#
$S \leftarrow$ new array of n integers	n
$A \leftarrow$ new empty stack	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
while ($\neg A.empty() \wedge$ $X[A.top()] \leq X[i]$) **do**	n
$A.pop()$	n
if $A.empty()$ **then**	n
$S[i] \leftarrow i + 1$	n
else	
$S[i] \leftarrow i - A.top()$	n
$A.push(i)$	n
return S	1