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
  - `pop()`: removes and returns the last inserted element
- Auxiliary stack operations:
  - `top()`: returns the last inserted element without removing it
  - `size()`: returns the number of elements stored
  - `isEmpty()`: indicates whether no elements are stored

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from `top()` and `pop()` when stack is empty
- Different from the built-in Java class `java.util.Stack`

```java
public interface Stack<E> {
    int size();
    boolean isEmpty();
    E top();
    void push(E element);
    E pop();
}
```
Example

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Stack Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>push(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>pop()</td>
<td>3</td>
<td>(5)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>(5)</td>
</tr>
<tr>
<td>pop()</td>
<td>5</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>pop()</td>
<td>null</td>
<td>()</td>
</tr>
<tr>
<td>push(7)</td>
<td>–</td>
<td>(7)</td>
</tr>
<tr>
<td>push(9)</td>
<td>–</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>top()</td>
<td>9</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(4)</td>
<td>–</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>size()</td>
<td>3</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>pop()</td>
<td>4</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(6)</td>
<td>–</td>
<td>(7, 9, 6)</td>
</tr>
<tr>
<td>push(8)</td>
<td>–</td>
<td>(7, 9, 6, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>8</td>
<td>(7, 9, 6)</td>
</tr>
</tbody>
</table>

Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition.
- Java supports a general abstraction for errors, called exception.
- An exception is said to be “thrown” by an operation that cannot be properly executed.

- In our Stack ADT, we do not use exceptions.
- Instead, we allow operations pop and top to be performed even if the stack is empty.
- For an empty stack, pop and top simply return null.
Applications of Stacks

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

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

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method 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 \textit{size()}

\begin{verbatim}
return \( t + 1 \)
\end{verbatim}

Algorithm \textit{pop()}

\begin{verbatim}
if \textit{isEmpty()} then
  return null
else
  \( t \leftarrow t - 1 \)
  return \( S[t+1] \)
\end{verbatim}

Array-based Stack (cont.)

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

Algorithm \textit{push(o)}

\begin{verbatim}
if \( t = S.length - 1 \) then
  throw \textit{IllegalStateException}
else
  \( t \leftarrow t + 1 \)
  \( S[t] \leftarrow o \)
\end{verbatim}
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 Java

```java
public class ArrayStack<E> implements Stack<E> {
    // holds the stack elements
    private E[] S;
    // index to top element
    private int top = -1;
    // constructor
    public ArrayStack(int capacity) {
        S = (E[]) new Object[capacity]);
    }
}
```

```java
public E pop() {
    if isEmpty()
        return null;
    E temp = S[top];
    // facilitate garbage collection:
    S[top] = null;
    top = top - 1;
    return temp;
}
```

... (other methods of Stack interface)
Example Use in Java

```java
public class Tester {
    // … other methods
    public intReverse(Integer a[]) {
        Stack<Integer> s;
        s = new ArrayStack<Integer>();
        ArrayStack<Integer>();
        … (code to reverse array a) …
    }

    public floatReverse(Float f[]) {
        Stack<Float> s;
        s = new ArrayStack<Float>();
        … (code to reverse array f) …
    }
}
```

 Parentheses Matching

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

```java
public static boolean isMatched(String expression) {
    final String opening = "({[""; // opening delimiters
    final String closing = "}])";; // respective closing delimiters
    Stack<Character> buffer = new LinkedStack<>();
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1) // this is a left delimiter
            buffer.push(c);
        else if (closing.indexOf(c) != -1) { // this is a right delimiter
            if (buffer.isEmpty()) // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
                return false; // mismatched delimiter
        }
    }
    return buffer.isEmpty(); // were all opening delimiters matched?
}
```

© 2014 Goodrich, Tamassia, Goldwasser

---

HTML Tag Matching

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

```html
<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?
**HTML Tag Matching (Java)**

```java
public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new LinkedStack<>();
    int j = html.indexOf('<'); // find first '<' character (if any)
    while (j != -1) {
        int k = html.indexOf('>', j+1); // find next '>' character
        if (k == -1) // invalid tag
            return false;
        String tag = html.substring(j+1, k); // strip away < >
        if (!tag.startsWith('/')) // this is an opening tag
            buffer.push(tag);
        else { // this is a closing tag
            if (buffer.isEmpty()) // no tag to match
                return false; // mismatched tag
            if (!tag.substring(1).equals(buffer.pop()))
                return false; // mismatched tag
        }
        j = html.indexOf('<', k+1); // find next '<' character (if any)
    }
    return buffer.isEmpty(); // were all opening tags matched?
}
```

© 2014 Goodrich, Tamassia, Goldwasser

---

**Evaluating Arithmetic Expressions**

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.

© 2014 Goodrich, Tamassia, Goldwasser
Algorithm for Evaluating Expressions

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

Operator ≤ has lower precedence than +/-
Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm.
- Given 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.

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

for $i$ ← 0 to $n - 1$ do

  $s$ ← 1

  while $s \leq i \land X[i - s] \leq X[i]$ do

    $s$ ← $s + 1$

  $S[i]$ ← $s$

return $S$

**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 $i$ onto the stack

Linear Time Algorithm

- Each index of the array
  - Is pushed into the stack exactly one time
  - 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
    A \leftarrow {} new empty stack
    for i \leftarrow 0 to n - 1 do
        while (¬A.isEmpty()) ∧ X[A.top()] ≤ X[i]) do
            A.pop()
        if A.isEmpty() then
            S[i] \leftarrow i + 1
        else
            S[i] \leftarrow i - A.top()
            A.push(i)
    return S
```

© 2014 Goodrich, Tamassia, Goldwasser