Stacks

The unorganized person’s data structure
Stack characteristics

- Entries are ordered in terms of access -- both insertion and removal take place at same spot (top of stack)
- Specialized type of container class; defining characteristic is insertion/removal order
- LIFO = last in, first out; entries are removed in reverse order of insertion
Stack operations

- Push -- insert item on stack
- Pop -- remove item from stack
- Peek -- examine, but don’t remove item
- isEmpty -- reports whether or not stack is empty
- size -- returns # of items in stack
- constructor
Stack operations

• Important to know if stack is empty -- attempt to remove an item from an empty stack is an *underflow* error

• Depending on implementation, may be necessary to check if stack is full -- attempt to add item to a full stack is an *overflow* error
Implementation of Stack ADT

• The Java API provides a generic stack implementation based on the Vector type
• We will use this implementation for programs that use stacks; but it is worth looking behind the scenes to see how a stack might be implemented if the API class didn’t exist
Implementation of Stack ADT

• Stacks can be array based or linked list based

• Invariant for array implementation:
  – The number of items stored in the stack is found in member variable manyItems
  – Items are stored in member variable data, an array with the stack bottom at data[0] and the stack top at data[manyItems - 1]
Stack class -- array version

public class ArrayStack<E> implements Cloneable {
    private Object[] data;
    private int manyItems;
    final int INITIAL_CAPACITY = 10;
    public ArrayStack() {
        manyItems = 0;
        data = new Object[INITIAL_CAPACITY];
    }
}
Stack ADT continued

```java
public ArrayStack(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("initialCapacity too small: "+ initialCapacity);

    manyItems = 0;
    data = new Object[initialCapacity];
}
```
Clone method

```java
public ArrayStack<E> clone() {
    ArrayStack<E> answer;
    try {
        answer = (ArrayStack<E>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException("This class does not implement Cloneable");
    }
    answer.data = data.clone();
    return answer;
}
```
Resize method (ensureCapacity)

public void ensureCapacity(int minimumCapacity) {
    Object biggerArray[];
    if (data.length < minimumCapacity)
    {
        biggerArray = new Object[minimumCapacity];
        System.arraycopy(data, 0, biggerArray, 0, manyItems);
        data = biggerArray;
    }
}
public void trimToSize() {
    Object trimmedArray[];
    if (data.length != manyItems) {
        trimmedArray = new Object[manyItems];
        System.arraycopy(data, 0, trimmedArray, 0, manyItems);
        data = trimmedArray;
    }
}
Accessor methods

```java
public int getCapacity() {
    return data.length;
}

public int size() {
    return manyItems;
}

public boolean isEmpty() {
    return (manyItems == 0);
}
```
Pop method

// Precondition: stack is not empty
// Postcondition: top item is removed

public Object pop( ) {
    if (manyItems == 0)
        throw new EmptyStackException( );
    return data[--manyItems];
}
// Precondition: stack is not empty
// Postcondition: top item is revealed
public Object peek() {
    if (manyItems == 0)
        throw new EmptyStackException();
    return data[manyItems-1];
}
Push function

// Precondition: stack is not full
// Postcondition: an item is inserted on stack
public void push(E item) {
    if (manyItems == data.length) {
        ensureCapacity(manyItems*2 + 1);
    }
    data[manyItems] = item;
    manyItems++;
}
Stack ADT as linked list

• Uses instance of Node class and methods from this class

• Stack can grow & shrink as needed to accommodate data -- no fixed size
Invariant for LinkedStack

- The items in the stack are stored in a linked list, with the top of the stack stored at the head node, down to the bottom of the stack at the final node.
- The instance variable top is the head reference of the linked list of items.
public class LinkedStack<E> implements Cloneable {

    private Node<E> top;

    public LinkedStack() {
        top = null;
    }
}
Clone method

```java
public LinkedStack<E> clone() {
    LinkedStack<E> answer;
    try {
        answer = (LinkedStack<E>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException("This class does not implement Cloneable");
    }
    answer.top = Node.listCopy(top);
    return answer;
}
```
Accessor methods

public boolean isEmpty() {
    return (top == null);
}

public int size() {
    // The generic listLength method gets the type of E from top.
    return Node.listLength(top);
}
Push method

public void push(E item)
{
    top = new Node<E>(item, top);
}

stacks 21
Pop method

```java
public E pop() {
    E answer;
    if (top == null)
        throw new EmptyStackException();
    answer = top.getData();
    top = top.getLink();
    return answer;
}
```
public E peek() {
    if (top == null)
        throw new EmptyStackException();
    return top.getData();
}
Stack application examples

- Compilers use stacks for a variety of purposes:
  - syntax analysis: matching brackets, parentheses, etc.
  - activation records: structures associated with functions, keeping track of local variables, return address, etc.
Example application: balanced parentheses and brackets

- Pseudocode algorithm:
  - scan string left to right
  - if ‘(’ (or ‘[’ or ‘{’) is encountered, push on stack
  - if ‘)’ is encountered, and stack is not empty, pop one ‘(‘ -- if stack is empty, expression is unbalanced
  - if stack is empty when entire string has been scanned and analyzed, expression is balanced
A method to test for balanced parentheses

```java
public static boolean isBalanced(String expression)
// Postcondition: A true return value indicates that the parentheses in the
given expression are balanced. Otherwise the return value is false.
// Note that characters other than ( ) { } and [ ] are ignored.
{
    // Meaningful names for characters
    final char LEFT_NORMAL  = '(';
    final char RIGHT_NORMAL = ')';
    final char LEFT_CURLY   = '{';
    final char RIGHT_CURLY  = '}';
    final char LEFT_SQUARE  = '[';
    final char RIGHT_SQUARE = ']';
```
Balanced paren method
continued

Stack<Character> store = new Stack<Character>(); // Stores parens
int i; // An index into the string
boolean failed = false; // Change to true for a mismatch

for (i = 0; !failed && (i < expression.length()); i++) {
    switch (expression.charAt(i))
    {
        case LEFT_NORMAL:
        case LEFT_CURLY:
        case LEFT_SQUARE:
            store.push(expression.charAt(i));
            break;
    }
}

stacks
Balanced parens continued

case RIGHT_NORMAL:
    if (store.isEmpty() || (store.pop() != LEFT_NORMAL))
        failed = true;
    break;

case RIGHT_CURLY:
    if (store.isEmpty() || (store.pop() != LEFT_CURLY))
        failed = true;
    break;

case RIGHT_SQUARE:
    if (store.isEmpty() || (store.pop() != LEFT_SQUARE))
        failed = true;
    break;

} // ends switch

} // ends loop

return (store.isEmpty() && !failed);

} // ends method
A stack-based calculator

• Input to program is a fully-parenthesized expression -- examples:
  
  \((5.3 \times 1.2) / 3.1\)
  
  \((4 - 3)\)

• Two stacks are used -- one for operators, one for operands

• Right parenthesis is signal to pop the stacks and evaluate the expression
Algorithm for expression evaluation

• Evaluate leftmost, innermost expression; continue evaluating, left to right
  – Read each part of expression
  – Push numbers on operand stack, operators on operator stack
  – When right parenthesis is encountered, pop the stacks, evaluate, and push result on operand stack
Regular expressions and methods
hasNext and findInLine

- Scanner class has methods that can be used to look ahead into an input stream
- The hasNext method can be used simply, to see if there is more data to be read in an input buffer; we used this technique in CS1 to read data to the end of the file
- The Stack calculator’s evaluate method uses the hasNext method, as well as another Scanner method, findInLine, to determine if the data to be read is of a particular kind
Regular expressions and methods

hasNext and findInLine

• These two methods can take as arguments a previously defined regular expression
• A regular expression indicates what a data item (e.g. a whole number, real number, or String) should consist of
• We can use an instance of the Pattern class to define a regular expression, as shown on the following slide
Regular expressions and methods hasNext and findInLine

```java
public static final Pattern CHARACTER = Pattern.compile("\S.*?");
public static final Pattern UNSIGNED_DOUBLE = Pattern.compile("((\d+\.?\d*)|(\.\d+))(Ee][-+]?\d+).*?");
```

These patterns are used in the evaluate method to distinguish operators from operands.
Stack calculator: evaluate method

- **Precondition:** The string is a fully parenthesized arithmetic expression formed from non-negative numbers, parentheses, and the four operations 
  `// +, -, *, and /.`
- **Postcondition:** The string has been evaluated and the value returned.
- **Exceptions:**
  - Can throw an `NumberFormatException` if the expression contains characters other than digits, operations, parentheses and whitespace.
  - Can throw `IllegalArgumentException` if the input line is an illegal expression, such as unbalanced parentheses or a division by zero.
public static double evaluate(String s) {
    Scanner input = new Scanner(s);
    Stack<Double> numbers = new Stack<Double>();
    Stack<Character> operations = new Stack<Character>();
    String next;
    char first;

while (input.hasNext( )) {
    if (input.hasNext(UNSIGNED_DOUBLE)) {
        next = input.findInLine(UNSIGNED_DOUBLE);
        numbers.push(new Double(next));
    }
    else {
        next = input.findInLine(CHARACTER);
        first = next.charAt(0);
    }
}
evaluate method continued

switch (first) {
    case '+': case '-': case '*': case '/':
        operations.push(first);
        break;
    case ')
        evaluateStackTops(numbers, operations);
        break;
    case '(':
        break;
}

stacks 37
default :
    throw new IllegalArgumentException("Illegal character");
} // end switch

} // end else

} // end while

if (numbers.size() != 1)
    throw new IllegalArgumentException
        ("Illegal input expression");

    return numbers.pop( );
} // end method
evaluateStackTops method

- **Precondition:** The top of the operations stack contains +, -, *, or /, and the numbers stack contains at least two numbers.
- **Postcondition:** The top two numbers have been popped from the numbers stack, and the top operation has been popped from the operations stack. The two numbers have been combined using the operation (with the second number popped as the left operand). The result of the operation has then been pushed back onto the numbers stack.
- **Exceptions:** Throws an IllegalArgumentException if the stacks are illegal or if the operation results in a division by zero.
public static void evaluateStackTops(Stack<Double> nums,
                                    Stack<Character> ops) {
    double operand1, operand2;
    if ((nums.size() < 2) || (ops.isEmpty()))
        throw new IllegalArgumentException("Illegal expression");
    operand2 = nums.pop();
    operand1 = nums.pop();

    // continued on next slide
evaluateStackTops method

switch (ops.pop( ) ){
    case '+':
        nums.push(operand1 + operand2);
        break;
    case '-':
        nums.push(operand1 - operand2);
        break;
}
evaluateStackTops method

case '*':
    numbers.push(operand1 * operand2);
    break;

case '/':
    numbers.push(operand1 / operand2);
    break;

default: throw new IllegalArgumentException
        ("Illegal operation");

} // end switch

} // end method
The trouble with infix ...

• Rules for expression evaluation seem simple -- evaluate expression left to right, results of each sub-expression becoming operands to larger expression -- but …

• All operators are not created equal -- multiplication & division take precedence over addition & subtraction ...
The trouble with infix ...

• So it isn’t really all that simple -- we must first scan for higher-precedence operations, and evaluate these first -- and that’s not so easy to program -- so …

• In the calculator program, we rely on parentheses to tell us which operations to perform when -- hence the need for fully-parenthesized expression
Alternatives to infix -- prefix

• Prefix notation, a.k.a. Polish notation

• Operator precedes operands
  – infix expression:  A + B
  – prefix expression:  +AB

• Parentheses become unnecessary
  – infix expression:  (A + B) * C
  – prefix expression:  * + A B C
Converting from infix to prefix

• Write out operands in original order
• Place operators in front of their operands
• If there’s a compound expression, the prefix expression may have two or more operators in a row
• If parentheses are not present, pay attention to precedence
Conversion examples

A + B + C  >>>>>>>>  + + A B C
A - B + C  >>>>>>>>  + - A B C
A + (B - C) >>>>>>>>  + A - B C
A * ((B + C) - D) / E >>>>> / * A - + B C D E
A + B * C / D >>>>> + A / * B C D
A * B + C - D / E >>>> - + * A B C / D E
Prefix evaluation

- scan left to right until we find the first operator immediately followed by pair of operands
- evaluate expression, and replace the “used” operator & operands with the result
- continue until a single value remains
Prefix Example

+ * / 4 2 3 9  // original expression
+ * 2 3 9    // 4/2 evaluated
+ 6 9        // 2*3 evaluated
15           // 6+9 evaluated
Another example

* - + 4 3 5 / + 2 4 3  // original expression
* - 7 5 / + 2 4 3    // 4+3 evaluated
* 2 / + 2 4 3        // 7-5 evaluated
* 2 / 6 3            // 2+4 evaluated
* 2 2                // 6/3 evaluated
4                    // 2*2 evaluated
Prefix summary

- Operands (but often not operators) same order as infix
- Expression designated unambiguously without parentheses
- Improvement on infix, but still not quite as simple to evaluate as one might wish -- have to deal with exceptions
Alternative II: Postfix

• Postfix is also known as reverse Polish notation -- widely used in HP calculators
• In postfix notation, operators appear after the operands they operate on
• As with prefix, operand order is maintained, and parentheses are not needed
• Postfix expression is not merely a reverse of the equivalent prefix expression
Postfix expression examples

• Simple expression:
  – Original Expression: $A + B$
  – Postfix Equivalent: $A B +$

• Compound expression with parentheses:
  – original: $(A + B) \times (C - D)$
  – postfix: $A B + C D - \ast$

• Compound expression without parentheses:
  – original: $A + B \times C - D$
  – postfix: $A B C \ast + D -$
Postfix expression evaluation

• Read expression left to right
• When an operand is encountered, save it & move on
• When an operator is encountered, evaluate expression, using operator & last 2 operands saved, saving the result
• When entire expression has been read, there should be one value left -- final result
Postfix evaluation using stack

• Postfix evaluation can easily be accomplished using a stack to keep track of operands
• As operands are encountered or created (through evaluation) they are pushed on stack
• When operator is encountered, pop two operands, evaluate, push result
Postfix calculator code

```java
import java.util.*;
import java.util.regex.*;

public class PostfixCalculator {
    private String expression;
    private Stack<Double> nums;

    public static final Pattern CHARACTER = Pattern.compile("\S.*?");
    public static final Pattern UNSIGNED_DOUBLE = Pattern.compile("((\d+\.?\d*)|(\.\d+))(Ee)[-+]?\d+)?.*?");
```
public PostfixCalculator () {
    nums = new Stack<Double>();
    expression = "";
}

Postfix calculator code
public void evalPostfix () {
    Scanner expression = new Scanner(this.expression);
    String next;
    do {
        if (expression.hasNext(UNSIGNED_DOUBLE)) {
            next = expression.findInLine(UNSIGNED_DOUBLE);
            nums.push(new Double(next));
        }
    }
}
Postfix calculator code

else {
    next = expression.findInLine(CHARACTER);
calculate(next);
}
} while (expression.hasNext());
Postfix calculator code

```java
public void calculate (String n) {
    if (nums.size() < 2)
        throw new IllegalArgumentException("Input expression: " +
                                            expression + " invalid");
    double op2 = nums.pop();
    double op1 = nums.pop();
    char op = n.charAt(0);
}```
Postfix calculator code

```cpp
switch (op) {
    case '+':  nums.push(op1 + op2);
              break;
    case '-':  nums.push(op1 - op2);
              break;
    case '*':  nums.push(op1 * op2);
              break;
    case '/':  nums.push(op1 / op2);
              break;
}
```
public double getResult() {
    if (nums.size() > 1 || nums.isEmpty())
        throw new IllegalArgumentException("Input expression: "
            + expression + " invalid");
    return (double)nums.pop();
}

public void setExpression (String e) {
    expression = e;
}
Postfix calculator code

public static void main (String [] args) {
  PostfixCalculator pc = new PostfixCalculator();
  Scanner kb = new Scanner (System.in);
  String input;
  do {
    System.out.print ("Enter a postfix expression (or Q to quit: ");
    input = kb.nextLine();
    if (input.equalsIgnoreCase("q"))
      System.out.println ("So long, and thanks for all the fish!");
  }
}
else {
    pc.setExpression(input);
    pc.evalPostfix();
    System.out.println("Your expression evaluates to: " +
                      pc.getResult());
}
} while (!input.equalsIgnoreCase("q"));
Translating infix to postfix

- Postfix expression evaluation is easiest type to program
- Next task is to take an infix expression and translate it into postfix for evaluation
- Some basic assumptions:
  - all operations are binary (no unary negative)
  - expressions are fully parenthesized
Translating infix to postfix

• General method:
  – move each operator to the right of its corresponding right parenthesis
  – eliminate parentheses

• Example:
  (((A + B) * C) - (E * (F + G))))
  (((A B) + C) * (E (F G) +) * ) -
  A B+ C * E F G + * -
Pseudocode for translation program

Do {
    if (left parenthesis) read & push
    else if (operand) read & write to output string
    else if (operator) read & push
    else if (right parenthesis)
        read & discard
        pop operator & write to output string
        pop & discard left parenthesis
} while (expression to read)
OK -- but what if expression *isn’t* fully parenthesized?

- We have to fall back on the rules for expression evaluation we know & love
  - do expression in parentheses first
  - do other operations in order of precedence -- in case of tie, leftmost sub-expression wins

- Example: \( A - ( B + C) \times D - E \)
- order: \( 3 \quad 1 \quad 2 \quad 4 \)
- Postfix: \( A \quad B \quad C \quad + \quad D \quad * \quad - \quad E \quad - \)
Algorithm for expression conversion

Do

  if (left parenthesis) read & push
else if (operand) read & write to file
else if (arithmetic operator)
    // continued on next slide

...
Conversion algorithm continued

while (stack not empty && stack.peek() != ‘(‘ &&
  op precedence lower than or same as
  stack.peek( ))
  pop stack & write to file
  read op
  push op
else // character should be ‘)’
  // next slide ...
Conversion algorithm continued

read & discard right paren

do

    pop stack & write to file

while (stack.peek( ) != ‘(‘)

pop & discard left paren

while (expression to read) // ends outer loop

while (stack not empty)

    pop & write to file