Stack and Its Implementation

Tessema M. Mengistu
Department of Computer Science
Southern Illinois University Carbondale
tessema.mengistu@siu.edu
Room - 3131
Outline

• Definition of Stack

• Usage of Stack
  – Checking for Balanced Algebraic Expression
  – Infix to Postfix Conversion
  – The Program Stack

• Array Based Implementation

• Linked List Based Implementation

• Vector Based Implementation
Stack

• Stack is an ADT that follow **last-in, first-out (LIFO)** behavior

• All additions added to one end of stack
  – Added to “top”
  – Called a “push” operation

• All removes to stack restricted to one end of stack
  – Remove only top entry
  – Called a “pop” operation
Stack
Stack ADT

• Data
  – A collection of objects having the same data type

• Operations
  – +push(newEntry: T): void
  – +pop(): T
  – +peek(): T
  – +isEmpty(): boolean
  – +clear(): void
public interface StackInterface<T>
{
    public void push(T newEntry);
    public T pop();
    public T peek();
    public boolean isEmpty();
    public void clear();
}
// end
Stackinterface ...

• Example:

```java
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.push("Joe");

String top = stringStack.peek(); // returns "Joe"
System.out.println(top + " is at the top of the stack.");

top = stringStack.pop(); // removes and returns "Joe"
System.out.println(top + " is removed from the stack.");

top = stringStack.peek(); // returns "Jane"
System.out.println(top + " is at the top of the stack.");
top = stringStack.pop(); // removes and returns "Jane"
System.out.println(top + " is removed from the stack.");
```
StackInterface ...
• Exercise

```java
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.pop();
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.pop();
```
Using a Stack to Process Algebraic Expressions

• Algebraic expressions composed of
  – Operands (variables, constants)
  – Operators (+, -, /, *, ^)

• Operators can be unary or binary

• Different precedence notations
  – Infix  a + b
  – Prefix  + a b
  – Postfix  a b +
Using a Stack to Process Algebraic Expressions

• Precedence must be maintained
  – Order of operators
  – Use of parentheses (must be balanced)

• Use stacks to evaluate parentheses usage
  – Scan expression
  – Push symbols
  – Pop symbols
Using a Stack to ...

- A **balanced expression** contains delimiters that are paired correctly
- Example
  
  \[
  a \{ b \left[ c (d + e)/2 - f \right] + 1 \} \quad - \text{balanced}
  \]
  
  \[
  a \{ b \left[ c (d + e)/2 - f \right] + 1 \} \} \quad - \text{not balanced}
  \]

**Algorithm**

1. Read the expression from left to right, ignoring other characters
2. When an opening delimiter is found store it
3. When a closing delimiter found, read the last saved opening delimiter
4. If they are the same goto 1
Using a Stack to ...

- Example

\[ a \{b \left[ c \left( d + e \right)/2 - f \right] + 1\} \]

- Let's use stack to store the delimiters
Using a Stack to ...

- Example

\{ a \ast \left[ b + c \ast \left( c/d \right) \right] - 6 \} + e \}

After push('{')
After push('[')
After push('(')
After pop()
Using a Stack to ... 

- Example

\[ [ ( ) ] ] \]
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
    nextCharacter = next character in expression
    switch (nextCharacter)
    {
        case '(': case '[': case '{':
            Push nextCharacter onto stack
            break
        
        case ')': case ']': case '}':
            if (stack is empty)
                isBalanced = false
            else
            {
                openDelimiter = top entry of stack
                Pop stack
                isBalanced = true or false according to whether openDelimiter and
                nextCharacter are a pair of delimiters
            }
            break
    }
}

if (stack is not empty)
    isBalanced = false
return isBalanced
Infix to Postfix Conversion

• Manual algorithm for converting infix to postfix

(a + b) * c

– Write with parentheses to force correct operator precedence

((a + b) * c)

– Move operator to right inside parentheses

((a b + ) c * )

– Remove parentheses

a b + c *
Infix to Postfix ...

• Algorithm basics
  – Scan expression left to right
  – When operand found, place at end of new expression
  – When operator found, save to determine new position
Infix to Postfix ...

- Example

\[ a + b \times c \]

<table>
<thead>
<tr>
<th>Next Character in Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(a)</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>(a)</td>
<td>+</td>
</tr>
<tr>
<td>(b)</td>
<td>(ab)</td>
<td>+</td>
</tr>
<tr>
<td>(*)</td>
<td>(abc)</td>
<td>+*</td>
</tr>
<tr>
<td>(c)</td>
<td>(abc)</td>
<td>+*</td>
</tr>
<tr>
<td></td>
<td>(abc)</td>
<td>+</td>
</tr>
</tbody>
</table>

Infix to Postfix ...

• Example

\[ a - b + c \]

<table>
<thead>
<tr>
<th>Next Character in Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( a )</td>
<td>( - )</td>
</tr>
<tr>
<td>( - )</td>
<td>( ab )</td>
<td>( - )</td>
</tr>
<tr>
<td>( b )</td>
<td>( ab - )</td>
<td></td>
</tr>
<tr>
<td>( + )</td>
<td>( ab - c )</td>
<td></td>
</tr>
<tr>
<td>( c )</td>
<td>( ab - c + )</td>
<td></td>
</tr>
</tbody>
</table>
Infix to Postfix Conversion

1. Operand
   – Append to end of output expression

2. Operator ^
   – Push ^ onto stack

3. Operators +, -, *, /
   – Pop from stack, append to output expression
   – Until stack empty or top operator has lower precedence than new operator
   – Then push new operator onto stack
Infix to Postfix Conversion

4. Open parenthesis
   – Push ( onto stack

5. Close parenthesis
   – Pop operators from stack and append to output
   – Until open parenthesis is popped.
   – Discard both parentheses
Example

\[ \frac{a}{b} \times (c + (d - e)) \]

<table>
<thead>
<tr>
<th>Next Character from Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>/</td>
</tr>
<tr>
<td>/</td>
<td>a</td>
<td>/</td>
</tr>
<tr>
<td>b</td>
<td>ab</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>ab/</td>
<td>*</td>
</tr>
<tr>
<td>(</td>
<td>ab/</td>
<td>*(</td>
</tr>
<tr>
<td>c</td>
<td>ab/c</td>
<td>*(</td>
</tr>
<tr>
<td>+</td>
<td>ab/c</td>
<td>*(</td>
</tr>
<tr>
<td>(</td>
<td>ab/c</td>
<td>*(</td>
</tr>
<tr>
<td>d</td>
<td>ab/cd</td>
<td>*(</td>
</tr>
<tr>
<td>-</td>
<td>ab/cd</td>
<td>*(</td>
</tr>
<tr>
<td>e</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd-e</td>
<td>*(</td>
</tr>
</tbody>
</table>
# Infix to Postfix

- **General rule:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>Append each operand to the end of the output expression.</td>
</tr>
<tr>
<td>Operator ^</td>
<td>Push ^ onto the stack.</td>
</tr>
<tr>
<td>Operator +, -, *, or /</td>
<td>Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.</td>
</tr>
<tr>
<td>Open parenthesis</td>
<td>Push ( onto the stack.</td>
</tr>
<tr>
<td>Close parenthesis</td>
<td>Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.</td>
</tr>
</tbody>
</table>
Infix to Postfix

• Exercise

\[(a - b * c) / (d * e ^ f * g + h)\]
Evaluating Postfix Expression

• Requires no rules of precedence
  – The order of its operators and operands dictates the order of operations

• Contains no parenthesis

• Algorithm
  – Scan the postfix expression
  – Save operands till operator is found
  – Apply the operator on the operands
    • The last saved operand is the second operand
  – Save the result
  – Continue till the expression is empty
• Example

\( \frac{a}{b} \) where \( a = 2 \), \( b = 4 \)
Evaluating Postfix Expressions

*Algorithm* evaluatePostfix(postfix)
  // Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
    nextCharacter = next nonblank character of postfix
    switch (nextCharacter)
    {
        case variable:
            valueStack.push(value of the variable nextCharacter)
            break
        case '+': case '-': case '*': case '/': case '^':
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in nextCharacter and its operands
            operandOne and operandTwo
            valueStack.push(result)
            break
        default: break
    }
}

return valueStack.peek()
• Example

\[ a \cdot b + c / \] where \( a = 2 \), \( b = 4 \) and \( c = 3 \)
• Exercise

\[ e \cdot b \cdot c \cdot a^* + d - \]

where \( a = 2, \ b = 3, \ c = 4, \ d = 5, \ e = 6 \)
The Program Stack

- When a program executes, a special location called the **program counter** (PC) references the current instruction.
- When a method is called, the program’s run-time environment creates an object called an **activation record**, or **frame**, for the method
  - Contains method’s arguments, local variables, and a reference to the current instruction (PC)
- At the time the method is called, the activation record is pushed onto a stack called the **program stack**
- The program stack often contains more than one activation record.
  - The record at the top of the stack belongs to the method that is currently executing
FIGURE 5-13 The program stack at three points in time:
(a) when main begins execution; (PC is the program counter)
The Program Stack

FIGURE 5-13 The program stack at three points in time: (b) when \texttt{methodA} begins execution; (PC is the program counter)
The Program Stack

FIGURE 5-13 The program stack at three points in time: (c) when methodB begins execution; (PC is the program counter)
Java Class Library : Stack

• Under java.util package

• Methods
  – public T push(T item);
  – public T pop();
  – public T peek();
  – public boolean empty();
Linked Implementation

• Consider push, pop, peek
  – Each involves top of stack
• Best to put top of stack at head node
  – Fastest, easiest to access
public class LinkedListStack<T> implements StackInterface<T>
{
    private Node topNode; // references the first node in the chain

    public LinkedListStack()
    {
        topNode = null;
    } // end default constructor

    ...

    private class Node
    {
        private T data; // entry in stack
        private Node next; // link to next node

        < Constructors and the methods getData, setData, getNextNode, and setNextNode are here. >
    } // end Node

} // end LinkedListStack
push Method

topNode

newNode

topNode

topNode
push Method

```java
public void push(T newEntry)
{
    Node newNode = new Node(newEntry, topNode);
    topNode = newNode;
}
// end push
```
peek Method

```java
def public T peek() {
    T top = null;
    if (topNode != null)
        top = topNode.getData();

    return top;
}  // end peek
```
pop Method

(a) Chain

Top entry of stack

Stack

topNode

(b) Returned to client

top

Stack
public T pop()
{
    T top = peek();
    if (topNode != null)
        topNode = topNode.getNextNode();
    return top;
} // end pop
Other Methods

```java
public boolean isEmpty()
{
    return topNode == null;
} // end isEmpty

public void clear()
{
    topNode = null;
} // end clear
```
Array Based Implementation

• Again the question:
  – Where to place the top entry?
Array Based Implementation
Array Based Implementation

• More efficient operations with bottom of stack at beginning of array
  – Top of stack at last occupied entry
• Must consider memory wastage of unused array elements
public class ArrayStack<T> implements StackInterface<T> {
    private T[] stack;       // array of stack entries
    private int topIndex;    // index of top entry
    private static final int DEFAULT_INITIAL_CAPACITY = 50;

    public ArrayStack()
    {
        this(DEFAULT_INITIAL_CAPACITY);
    } // end default constructor

    public ArrayStack(int initialCapacity)
    {
        // the cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] tempStack = (T[]) new Object[initialCapacity];
        stack = tempStack;
        topIndex = -1;
    } // end constructor

    < Implementations of the stack operations go here. >

    ...
push Method

```java
public void push(T newEntry)
{
    ensureCapacity();
    topIndex++;
    stack[topIndex] = newEntry;
} // end push

private void ensureCapacity()
{
    if (topIndex == stack.length - 1) // if array is full,
    // double size of array
        stack = Arrays.copyOf(stack, 2 * stack.length);
} // end ensureCapacity
```
peek Method

```java
public T peek()
{
    T top = null;
    if (!isEmpty())
        top = stack[topIndex];
    return top;
} // end peek
```
pop Method
pop Method

(b)

Array

0 1 2 3

null

Stack

Returned to client

Top entry of stack

topIndex
public T pop()
{
    T top = null;
    if (!isEmpty())
    {
        top = stack[topIndex];
        stack[topIndex] = null;
        topIndex--;
    } // end if

    return top;
} // end pop
Other Methods

```java
public clear()
{
    for (int i=0;i<topIndex;i++)
        stack[i]=null;
    topIndex=-1;
}
```
Vector-Based Implementation

• Vector is a structure which behaves like a high level array
  – Has methods to access entries
  – Grows in size as needed (hidden from client)
Constructors and Methods of Vector

• public Vector()
• public Vector(int initialCapacity)
• public boolean add(T newEntry)
• public T remove(int index)
• public void clear()
• public T lastElement()
• public boolean isEmpty()
• public int size()
import java.util.Vector;

/**
   A class of stacks whose entries are stored in a vector.
   @author Frank M. Carrano
*/

public class VectorStack<T> implements StackInterface<T>
{
    private Vector<T> stack; // last element is the top entry in stack
    private static final int DEFAULT_INITIAL_CAPACITY = 50;

    public VectorStack()
    {
        this(DEFAULT_INITIAL_CAPACITY);
    } // end default constructor

    public VectorStack(int initialCapacity)
    {
        stack = new Vector<T>(initialCapacity); // size doubles as needed
    } // end constructor

    < Implementations of the stack operations go here. >

} // end VectorStack
Stack Methods for Vector Implementation

```java
public void push(T newEntry) {
    stack.add(newEntry);
} // end push
```
pop Method

```java
public T pop()
{
    T top = null;
    if (!isEmpty())
    {
        top = stack.remove(stack.size() - 1);
    }
    return top;
} // end pop
```
peek Method

```java
public T peek()
{
    T top = null;
    if (!isEmpty())
        top = stack.lastElement();
    return top;
} // end peek
```
Other Methods

```java
public boolean isEmpty()
{
    return stack.isEmpty();
} // end isEmpty

public void clear()
{
    stack.clear();
} // end clear
```
Exercise

- Write a program that accepts a string from the user and displays the reverse of the string. Use Stack class form Java’s collection framework.