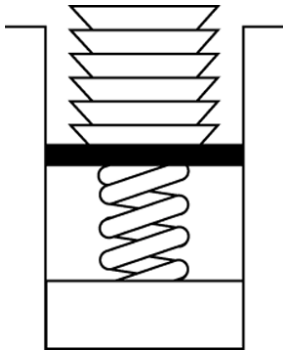
COMPSCI 105 S2 C: Principles of Computer Science

Chapter 7 – Stacks

Outline on Stacks

- What is a Stack?
- Stack ADT
- Applications
  - Line Editing
  - Bracket Matching
  - Postfix Calculation
- Implementation of Stack (Array-Based)
- Implementation of Stack (Reference-Based)

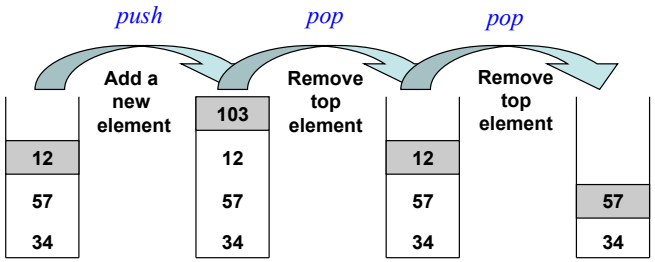
What is a Stack?



Can you think of other examples of stacks?

More formally

- Only add to the top of the Stack
- Only remove from the top of the Stack
- Last In – First Out (LIFO) behaviour



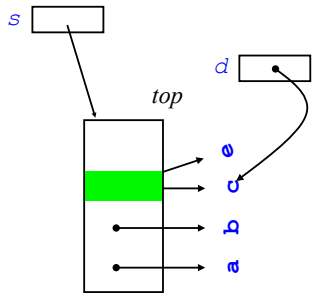
Stack ADT Interface

- We can use Java Interface to specify Stack ADT Interface

```
public interface StackInterface {

  public boolean isEmpty();
  // Determines whether a stack is empty.

  public void push(Object newItem) throws
  StackException;
  // Adds newItem to the top of the stack.

  public Object pop() throws StackException;
  // Removes the top of a stack.

  public void popAll();
  // Remove all items from the stack.

  public Object peek() throws StackException;
  // Retrieves the top of a stack.
}
```


Simple example

```
StackInterface s = new Stack();
s.push("a");
s.push("b");
s.push("c");
d = s.peek();
s.pop();
s.push("e");
s.pop();
```



Applications

- Many application areas use stacks:
  - Line editing
  - Bracket matching
  - Postfix expression calculation
  - Converting from infix to postfix
  - Search algorithms
  - Function and recursive calls

Line Editing

- A line editor would place the characters read into a buffer but may use a backspace symbol (denoted by '←') to do error correction
- Refined Task
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

e.g.,

Input : `abc_defgh←2klpq←←←wxyz`

Corrected Input : `abc_defg2klpwxyz`

Reversed Output : `zyxwplk2gfed_cba`

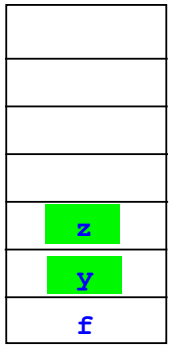
Informal Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : fgh←r←←yz

Corrected Input : fyz

Reversed Output : zyf



Stack

Pseudo Algorithm

```
StackInterface s = new Stack();
// read the lines & correcting mistakes
Read a new character ch;
while (ch is not the end-of-line) {
  if (ch != '←') {
    s.push (ch);
  }
  else if (!s.isEmpty()) {
    s.pop();
  }
  Read a new character ch;
}

// write the line in reverse order
while (!s.isEmpty()) {
  newChar = s.pop();
  Write newChar;
}
```

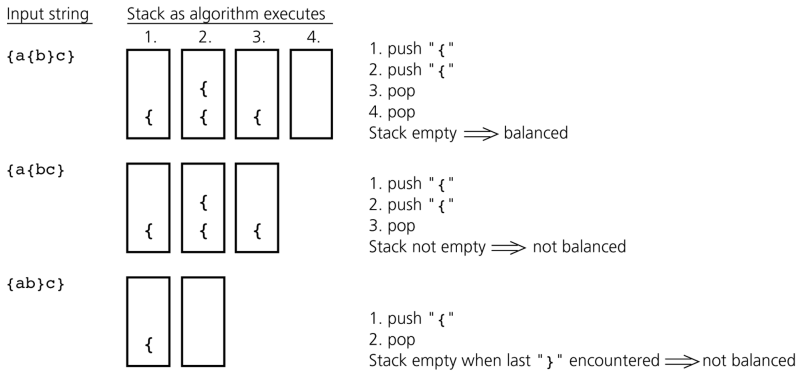

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - An Example: {a,(b+f[4])*3,d+f[5]}
  - Bad Examples:

```
(..)..)          // too many closing brackets
(..(..)          // too many open brackets
[..(..]..)       // mismatched brackets
```


Bracket Matching Problem

- Ensures that pairs of brackets are properly matched

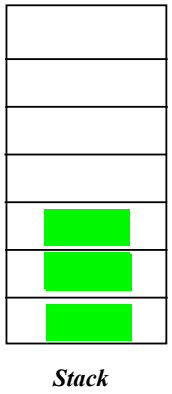


Informal Procedure

Initialise the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        pop from the stack
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

```
{a,(b+f[4])*3,d+f[5]}
```



Stack

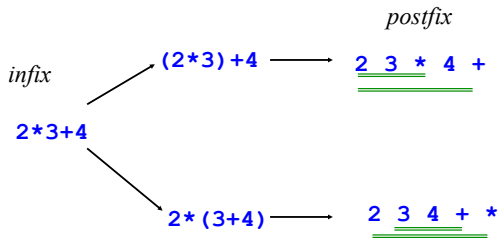
Pseudo Algorithm

```
public static boolean balanced() {
 StackInterface s = new Stack();
 boolean balancedSoFar = true;
 while (balancedSoFar && (not end of line)) {
   read a new char ch;
   if ((ch == '(') || (ch == '{') ||  (ch == '[')) {
     s.push(ch);
   }
   else if ((ch == ')') || (ch == '}') ||  (ch == ']')) {
     if (s.isEmpty()) {
       balancedSoFar = false;
       status = "No matching open brace";
     }
     else {
       d = s.pop ();
       if !(match (ch, d)) {
         balancedSoFar = false;
         status = "Wrong pair of matching brace";
       } else { /* do nothing */ }
     } } } // end while loop

 if (!balancedSoFar) { return false; }
 else if (!s.isempty()) {
        status = "Too many open parentheses"; return false; }
      else { return true; };
}
```


Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**



Evaluating Postfix Expression

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Key entered	Calculator action		Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Informal Procedure

Initialize stack
For each item read.
    If it is an operand,
        push on the stack
    If it is an operator,
        pop arguments from stack;
        perform operation;
        push result onto the stack

Expr

```
2       s.push (2)
3       s.push (3)
4       s.push (4)
+       arg2 = s.pop ()
        arg1 = s.pop ()
        s.push (arg1 + arg2)
*       arg2 = s.pop ()
        arg1 = s.pop ()
        s.push (arg1 * arg2)
```

Stack

Pseudo Algorithm

```
StackInterface s = new Stack();

while ( not end of line ) {
  read a new item ch
  if ( isOperand(ch) ) {
    s.push(valueof(ch));  // assume single digit
  }
  else
  {
    arg2 = (Integer) s.pop() ;
    arg1 = (Integer) s.pop() ;
    Integer res = compute(ch, arg1, arg2);
    s.push(res);
  } // end if
} // end while
```


Converting Infix Expressions to Equivalent Postfix Expressions

- operand – append it to the output string *postfixExp*
- "(" – push onto the stack
- operator
  - If the stack is empty, push the operator onto the stack
  - If the stack is not empty
    - pop the operators of greater or equal precedence from the stack and append them to *postfixExp*. Stop when encounter either a "(" or an operator of lower precedence or when the stack is empty.
    - push the new operator onto the stack.
- ")" – pop the operators off the stack and append them to the end of *postfixExp* until encounter the match "("
- end of the string – append the remaining contents of the stack to *postfixExp*

(Page 378 - Pseudo algorithm)

Converting Infix Expressions to Equivalent Postfix Expressions

a – (b + c * d) / e

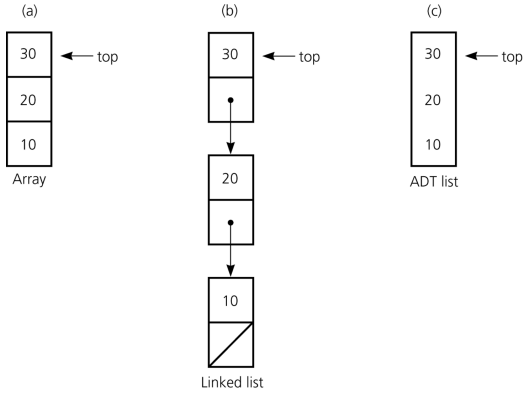
ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	`postfixExp` until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to `postfixExp`

Stack and recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an *activation record* that is pushed onto a stack, which stores the local environment of the current context before recursion
  - When a return is made from a recursive call, the stack is popped, which brings back the local environment of the previous context from the *activation record*
- Stacks can be used to implement a non recursive version of a recursive algorithm

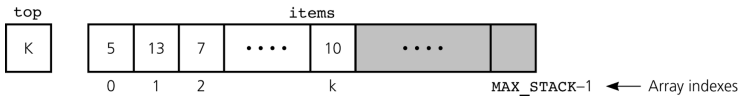
Implementations of the ADT Stack

- Array-Based implementation
- Reference-Based implementation
- An implementation that uses ADT List



Array-Based Implementation

- Can use Array with a top index pointer as an implementation of ADT Stack



Array-Based Implementation

```
public class StackArrayBased implements StackInterface {

  final int MAX_STACK = 50;  // maximum size of stack
  private Object items[];
  private int top;

  public StackArrayBased() {
    items = new Object[MAX_STACK];
    top = -1;
  }  // end default constructor

  public boolean isEmpty() {
    return top < 0;
  }  // end isEmpty

  public boolean isFull() {
    return top == MAX_STACK-1;
  }  // end isFull
```

Array-Based Implementation

```
public void push(Object newItem) throws StackException {
  if (!isFull()) {
    items[++top] = newItem;
  }
  else {
    throw new StackException("StackException on " +
                             "push: stack full");
  }  // end if
}  // end push

public void popAll() {
  items = new Object[MAX_STACK];
  top = -1;
}  // end popAll
```


Array-Based Implementation

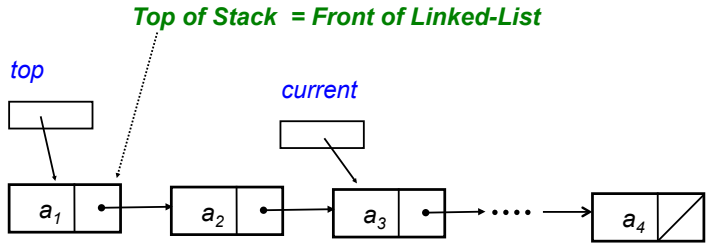
```
public Object pop() throws StackException {
  if (!isEmpty()) {
    return items[top--];
  }
  else {
    throw new StackException("StackException on " +
                             "pop: stack empty");
  }  // end if
}  // end pop

public Object peek() throws StackException {
  if (!isEmpty()) {
    return items[top];
  }
  else {
    throw new StackException("Stack exception on " +
                             "peek - stack empty");
  }  // end if
}  // end peek
}  // end StackArrayBased
```


Example

```
public class StackTest {
  public static final int MAX_ITEMS = 15;
  public static void main(String[] args) {
    StackInterface stack = new StackArrayBased();
    Integer items[] = new Integer[MAX_ITEMS];
    for (int i=0; i<MAX_ITEMS; i++) {
      items[i] = new Integer(i);
      if (!stack.isFull()) {
        stack.push(items[i]);
      }  // end if
    }  // end for
    while (!stack.isEmpty()) {
      // cast result of pop to Integer
      System.out.println((Integer)(stack.pop()));
    }  // end while
  }  // end main
}
```


Reference-Based Implementation

- Can use Linked List as implementation of the ADT Stack



Reference-Based Implementation

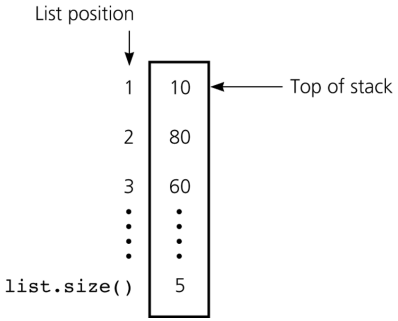
```
public class StackReferenceBased implements StackInterface
  {
  private Node top;

  public StackReferenceBased() {
    top = null;
  }  // end default constructor

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

  public void push(Object newItem) {
    top = new Node(newItem, top);
  }  // end push

  public void popAll() {
    top = null;
  }  // end popAll
```


Reference-Based Implementation

```
  public Object pop() throws StackException {
    if (!isEmpty()) {
      Node temp = top;
      top = top.getNext();
      return temp.getItem();
    }
    else {  throw new StackException("StackException on " +
                                     "pop: stack empty");
    }  // end if
  }  // end pop

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return top.getItem();
    }
    else {  throw new StackException("StackException on " +
                                     "peek: stack empty");
    }  // end if
  }  // end peek
}  // end StackReferenceBased
```


ADT List Implementation

- Can use ADT List as implementation of the ADT Stack



ADT List Implementation

```
public class StackListBased implements StackInterface {
  private ListInterface list;

  public StackListBased() {
    list = new ListReferenceBased();
  }  // end default constructor

  public boolean isEmpty() {
    return list.isEmpty();
  }  // end isEmpty

  public void push(Object newItem) {
    list.add(0, newItem);
  }

  public void popAll() {
    list.removeAll();
  }  // end popAll
```

ADT List Implementation

```
  public Object pop() throws StackException {
    if (!list.isEmpty()) {
      Object temp = list.get(0);
      list.remove(0);
      return temp;
    }
    else { throw new StackException("StackException on " +
                                    "pop: stack empty");
    }  // end if
  }  // end pop

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return list.get(0);
    }
    else { throw new StackException("StackException on " +
                                  "peek: stack empty");
    }  // end if
  }  // end peek
}  // end StackListBased
```


Comparing implementations

- Fixed size versus dynamic size
  - An array-based implementation
    - Uses fixed-sized arrays
      - Prevents the push operation from adding an item to the stack if the stack's size limit has been reached
  - A reference-based implementation
    - Does not put a limit on the size of the stack
- Linked List versus reference-based
  - Linked list approach
    - More efficient
  - ADT list approach
    - Reuses an already implemented structure
      - Much simpler to write
      - Saves time

The Java Collections Framework Class - Stack

- JCF contains an implementation of a stack class called Stack (generic)
- Derived from Vector
- Includes methods: peek, pop, push, and search
- search returns the 1-based position of an object on the stack

Summary

- The ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - search algorithms for backtracking
  - Strong relationship between recursion and stack
- Stack can be implemented by Array, Reference based, and ADT Linked Lists
- Pages 394 – self-test exercises 1, 2, 3, 4, 5, 6, 7, 8.