Computer Science 331
Stacks

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #10

Outline

1. Definition
2. Applications
   - Parenthesis Matching
   - Evaluation of Recursive Programs
3. Implementation
   - Array-Based Implementation
   - Linked List-Based Implementation
4. Additional Information
   - Stacks in Java 1.5 and the Textbook

Definition of a Stack

A **stack** is a collection of objects that can be accessed in “last-in, first-out” order: The only visible element is the (remaining) one that was most recently added.

Attributes:

- *size*: The number of elements on the stack; $size \geq 0$ at all times
- *top*: The topmost element on the stack. This refers to `null`, a special value, if the stack is empty (that is, if $size = 0$)

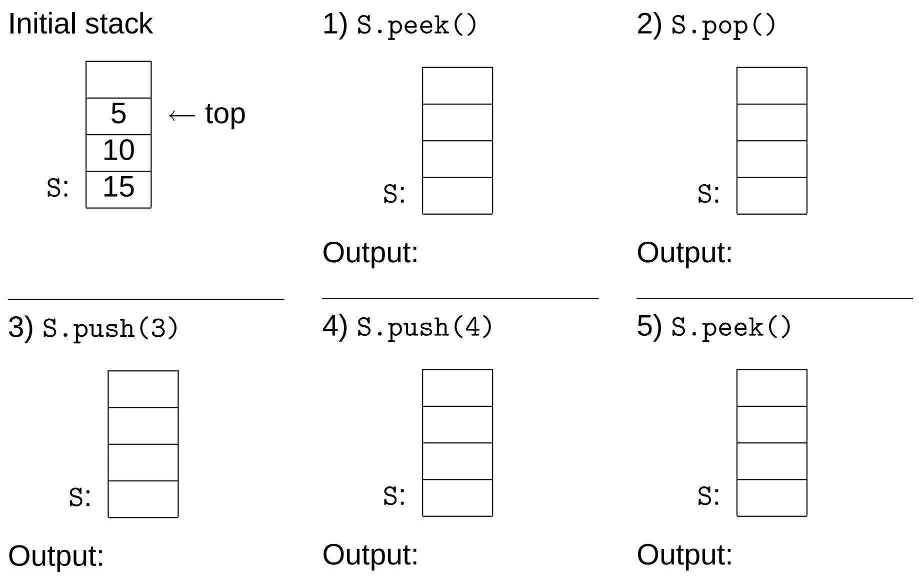
Definition of a Stack (cont.)

Operations:

- `Stack()`: Constructor; creates an empty stack
- `push( T element )`: Pushes `element` onto the top of the stack
- `pop()`: Removes the top element from the stack and returns the element it popped
- `peek()`:Returns the top element without removing it (so that the stack is unchanged)
- `size()`: Returns the number of elements on the stack
- `isEmpty()`: Reports whether the stack is empty

Operations `pop` and `peek` each have the **pre-condition** that the stack is nonempty and throw an *EmptyStackException* exception if this condition is not satisfied when they are called.

Example



Initial stack

1) `S.peek()`

Output:

2) `S.pop()`

Output:

3) `S.push(3)`

Output:

4) `S.push(4)`

Output:

5) `S.peek()`

Output:

Variation: Bounded Stacks

Size-Bounded Stacks — Similar to stacks (as defined above) with the following exception:

- Stacks are created to have a maximum *capacity* (possibly user-defined — so that two constructors are needed)
- If the capacity would be exceeded when a new element is added to the top of the stack then `push` throws a *StackOverflow* exception and leaves the stack unchanged

Most “hardware” and physical stacks are bounded stacks.

Further reading on stacks: Chapter 5

Problem: Parenthesis Matching

Consider an expression, given as a string of text, that might include various kinds of brackets.

How can we confirm that the brackets in the expression are properly matched? Eg. $[(3 \times 4) + (2 - (3 + 6))]$

Solution using a Stack:

- Begin with an empty bounded stack (whose capacity is greater than or equal to the length of the given expression)
-
-
-
-

Solution Using a Stack

Then parentheses are matched if and only if:

- Stack is never empty when we want to pop a left bracket off it, and
- Compared left and right brackets always *do* have the same type, and
- The stack is empty after the last symbol in the expression has been processed.

Provable by induction on the length of the expression.

Number of Stack Operations Required: *At most* two more than the length of the expression

Exercise: trace execution of this algorithm on the preceding example.

Problem: Evaluation of a Recursive Function

How is a recursive function (like this) evaluated on a computer?

```
public int **fib**(int $n$)
**if** $n == 0$ **then**
   **return** 0
**else if** $n == 1$ **then**
   **return** 1
**else**
   $x := \text{fib}(n-1)$
   $y := \text{fib}(n-2)$
   **return** $x + y$
**end if**
```

Solution Using a Stack

All information needed to support execution in a function is kept in an *activation record* (also called a *call frame*):

- space for parameters' values
- space for values of local variables
- space for location to which control should be returned

During program execution, one maintains a *process stack* of these activation records:

- When a function is called, create a new activation record to store information about it and push it onto the top of the stack; maintain information this call's progress on this
- When a function is finished, its activation record is popped off the stack and control is passed to the function whose activation record is currently on the top

Application To Example

Components of an Activation Record for This Function:

-
-
-
-

Exercise: Trace the behaviour of the process stack when `fib(4)` is computed.

Two possibilities

Array-based implementation:

- stack's contents stored in cells $0, \ldots, top - 1$; top element in $top - 1$
- use a dynamic array for a regular stack, static array for a bounded stack

Linked implementation:

- identify top of stack with the head of a singly-linked list
- works well because stack operations only require access to the top of the stack, and linked list operations with the head are especially efficient

Implementation Using an Array

Initial Stack

	5	← top
	10	
S:	15	

	0	1	2	3	4	5
S:	15	10	5	?	?	?

`top = 2`

Effect of `S.pop()`

S:	

	0	1	2	3	4	5
S:						

`top =`

Implementation Using an Array

Effect of `S.push(3)`

S:	

	0	1	2	3	4	5
S:						

`top =`

Effect of `S.push(4)`

S:	

	0	1	2	3	4	5
S:						

`top =`

Implementation of Stack Operations

```
public class ArrayStack<T> {
  private T[] stack;
  private int top;

  public ArrayStack()  {
  public boolean isEmpty()  {
  public int size()  {
  public void push(T x)  {
  public T peek() {
    if (isEmpty())  throw new EmptyStackException();

  }
  public T pop() {
    if (isEmpty())  throw new EmptyStackException();

  }
}
```

Cost of Operations

All operations cost $\Theta(1)$ (constant time, independent of stack size)

Problem: What should we do if the stack size exceeds the array size?

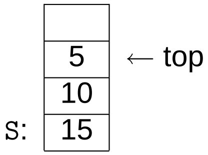
- modify push to reallocate a larger stack (or use a dynamic array)

```
public void push(T x) {
  ++top;
  if (top == stack.length) {
    T [] stackNew = (T[]) new Object[2*stack.length];
    System.arraycopy(stackNew,0,stack,0,stack.length);
    stack = stackNew;
  }
  stack[top] = x;
}
```

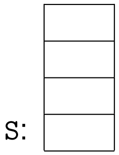
Revised cost (stack with *n* elements):

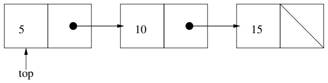
Implementation Using a Linked List

Initial Stack



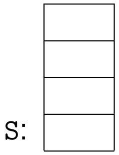
Effect of `S.pop()`



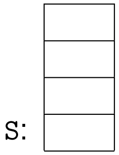


Implementation Using a Linked List

Effect of `S.push(3)`



Effect of `S.push(4)`



Implementation of Stack Operations

```
public class LinkedListStack<T> {
  private class StackNode<T> {
    private T value;
    private StackNode<T> next;

    private StackNode(T x, StackNode<T> n)
      { value = x; next = n; }
  }

  private StackNode<T> top;
  private int size;

  public LinkedListStack()
    {
  public boolean isEmpty()  {
  public int size()  { return size; }
```

Implementation of Stack Operations (cont.)

```
public void push(T x)  {

}

public T peek() {
  if (isEmpty())  throw new EmptyStackException();

}

public void pop() {
  if (isEmpty())  throw new EmptyStackException();

}
```

Cost of stack operations:

Stacks in Java and the Textbook

Implementation in Java 1.5:

- Java 1.5 includes a `Stack` class as an extension of the `Vector` class (a dynamic array).
  Unfortunately, this implementation is somewhat problematic — see page 271 of the textbook for details.

Implementation of Stacks in the Textbook (Section 5.3):

- Implementation using any class that implements the "List" interface
- Implementations "from Scratch" using ArrayList and Linked List

Programming Exercises:

1. Implement a `BoundedStack` class of your own using a static `Array`
2. Implement a `Stack` class of your own using `ArrayList`