Computer Science 331
Stacks

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #11

Outline

Definition of a Stack ADT

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.

It is easy to implement such a simple data structure extremely efficiently — and it can be used to several several interesting problems.

Indeed, a *stack* is used to execute recursive programs — making this one of the more widely used data structures (even though you generally don't notice it!)

A Stack Interface: Invariant

A stack interface `StackInt<E>` is defined on page 205 of the textbook. This is the basis for the following (which adds an appropriate interface invariant as well as preconditions and posconditions for method).

Interface Invariant:

- Used to provide access to a stack of objects of type E: The object that is visible at the top of the stack is the object that has most recently been pushed onto it (and not yet removed)

A Stack Interface: Methods

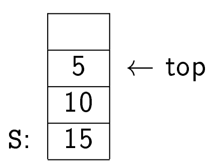
1. `void push(E obj)`:
   - *Precondition:* Interface invariant
   - *Postcondition:*
     a) The input object has been pushed onto the stack (which is otherwise unchanged)
2. `E peek()` (called `top` in the textbook):
   - *Precondition:*
     a) Interface Invariant
     b) The stack is not empty
   - *Postcondition:*
     a) Value returned is the object on the top of the stack
     b) The stack has not been changed
   - *Exception:* An `EmptyStackException` is thrown if the stack is empty when this method is called

A Stack Interface: Methods

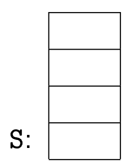
3. `E pop()`:
   - *Precondition:* Same as for `peek`
   - *Postcondition:*
     a) Value returned is the object on the top of the stack
     b) This top element has been removed from the stack
   - *Exception:* An `EmptyStackException` is thrown if the stack is empty when this method is called
4. `boolean isEmpty()`:
   - *Precondition:* Interface Invariant
   - *Postcondition:*
     a) The stack has not been changed.
     b) Value returned is `true` if the stack is empty and `false` otherwise

Example

Initial stack

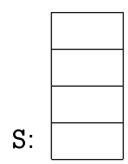


1) `S.peek()`



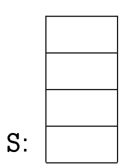
Output:

2) `S.pop()`



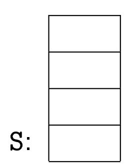
Output:

3) `S.push(3)`



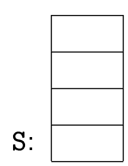
Output:

4) `S.push(4)`



Output:

5) `S.peek()`



Output:

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 (provable by induction on the length of the expression):

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

Solution Using a Stack (continued)

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.

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:

- space for parameter n
- space for local variable x
- space for local variable y
- space for return location

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

Two possibilities

Dynamic array implementation:

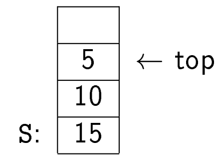
- stack's contents stored in cells $0, \ldots, top-1$; top element in $top-1$
- can use a static array if size of stack is bounded

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

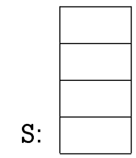
Initial Stack



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

`top = 2`

Effect of `S.pop()`

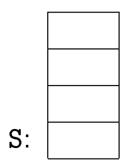


	0	1	2	3	4	5
S:						

`top =`

Implementation Using an Array

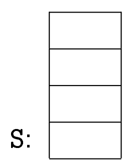
Effect of `S.push(3)`



	0	1	2	3	4	5
S:						

`top =`

Effect of `S.push(4)`



	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?

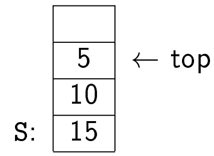
- modify push to reallocate a larger stack (or use a dynamic array)

```
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;
}
```

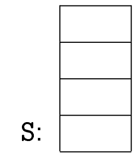
Revised cost:

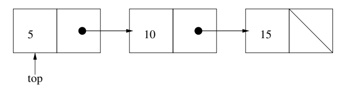
Implementation Using a Linked List

Initial Stack



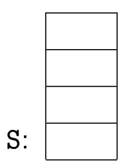
Effect of `S.pop()`



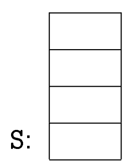


Implementation Using a Linked List

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:

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 `StackOverflowException` and leaves the stack unchanged
- A *static array* whose `length` is the stack's `capacity` can be used to implement a size-bounded stack, extremely simply and efficiently

Most "hardware" and physical stacks are bounded stacks.

Stacks in Java and the Textbook

Implementation in Java 1.6:

- Java 1.6 includes a `Stack` class as an extension of the `Vector` class (a dynamic array).
  Unfortunately, this implementation is somewhat problematic (Stack inheirit's Vector's methods, too!)

Implementation of Stacks in the Textbook (Section 5.1):

- Implementations "from Scratch" using arrays (for a bounded stack with fixed capacity) and a linked list