STACK ADT
Stack

• The stack is a LIFO (Last-in First-out) linear data structure.

• The only data element that can be removed is the most recently added element.
Stack ADT Specification

- **Elements:** Stack elements can be of any type, but we will assume StackElement.

- **Structure:** Any mechanism for determining the elements order of arrival into the stack.
Stack ADT Continued

- **Domain**: The number of stack elements is bounded. A stack is considered full if the upper-bound is reached. A stack with no elements is considered empty.

- **Operations**: There are seven operations as follows:
Stack ADT Continued

function create (s: Stack, isCreated: boolean)
results: if s cannot be created, isCreated is false; otherwise, isCreated is true, the stack is created and is empty

function terminate (s: Stack)
results: stack s no longer exists
Stack ADT Continued

function isFull (s: Stack)
    results: returns true if the stack is full; otherwise false is returned

function isEmpty (s: Stack)
    results: returns true if the stack is empty; otherwise, false is returned

function push (s: Stack, e: StackElement)
    requires: isFull (s) is false
    results: element e is added to the stack as the most recent element
Stack ADT Continued

function pop (s: Stack, e: StackElement)
    requires: isEmpty(s) is false
    results: The most recently added element is removed and assigned to e

function peek (s: Stack, e: StackElement)
    requires: isEmpty(s) is false
    results: The most recently added element is assigned to e but not removed
Testing your Data Structure

• Your customer will abuse your data structure

• Your data structure should never crash the customer's code
  • code defensively

• Test each each function
  • test each function’s requires statement
  • test boundary conditions (full/empty)
  • test bad input
  • test functions called in the wrong order
What are Stacks Useful for?

• Web browser history.

• “undo” in applications.

• Memory stack.
Ex. 1: Converting Decimal to Binary

Here is an algorithm for converting a decimal number to its binary equivalent:

- Read a number
- While number is greater than 0
  - Find the remainder after dividing the number by 2
  - Print the remainder
  - Divide the number by 2
- End the iteration

What is the problem with this algorithm?

How can a stack be used to fix the problem?
Ex. 2: Balancing Parentheses

- Parentheses in algebraic expressions need to be balanced in order for the expression to be correct.

- Which of the following are valid expressions?
  - \{a^2 - [ (c - d)^2 + (e - f)^2 ] \}
  - \{a - [ (b + c) ) ) - (d + e) ] \}
  - \{a - [ [ (b + c) - (d + e) ] \}
  - \{a - [ (b + c) - (d + e) ] \}

- How can a stack be used to test if an expression’s parentheses are balanced?
Stack Representation

• In stk.h

#define MAX_STACK 1024
#define TRUE 1
#define FALSE 0

typedef short int BOOLEAN;
typedef char DATATYPE;

typedef struct Stack
{
   int top;
   DATATYPE data[MAX_STACK];
} Stack;
Stack Functions

BOOLEAN stkCreate (Stack *);
BOOLEAN stkTerminate (Stack *);
BOOLEAN stkIsFull (Stack *);
BOOLEAN stkIsEmpty (Stack *);
BOOLEAN stkPush (Stack *, DATATYPE);
BOOLEAN stkPop (Stack *, DATATYPE *);
BOOLEAN stkPeek (Stack *, DATATYPE *);
Balancing Parentheses

• Assume that all of the functions have been implemented, how are you going to use a stack to test if parentheses are balanced?