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 NO_ERROR 0
#define ERROR_STACK_EMPTY 1
#define ERROR_STACK_FULL 2
#define ERROR_NO_STACK_CREATE 3
#define ERROR_NO_STACK_TERMINATE 4
#define ERROR_NO_STACK_MEMORY 5

#define MAX_STACK_ELEMENTS 100
```

```
typedef short int ERRORCODE;
typedef char DATATYPE;

typedef struct Stack *StackPtr;
typedef struct Stack
{
  int size;
  DATATYPE data[MAX_STACK_ELEMENTS];
  int top;
} Stack;
```

Stack Functions

```
extern ERRORCODE stkCreate (StackPtr psStack);
extern ERRORCODE stkTerminate (StackPtr psStack);
extern ERRORCODE stkIsFull (const StackPtr psStack,
                            bool *pbIsFull);
extern ERRORCODE stkIsEmpty (const StackPtr psStack,
                             bool *pbIsEmpty);
extern ERRORCODE stkPush (StackPtr psStack, DATATYPE value);
extern ERRORCODE stkPop (StackPtr psStack, DATATYPE *pValue);
extern ERRORCODE stkPeek (const StackPtr psStack,
                          DATATYPE *pValue);
extern ERRORCODE stkSize (const StackPtr psStack, int *pSize);
```

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?