Data Structure and Algorithm I

Homework #2

Due: 5pm, Thursday, October 21, 2010

Submit the answers for problem 2-3 through the CEIBA system (electronic copy) or to the TA in R432 (hard copy). You also need to submit the answers of problem 1 through the CEIBA system.

Problem 1. (40%) In section 3.6.2 of the textbook, we have discussed how to evaluate postfix expressions and how to convert an infix expression to a postfix expression. In this problem, you are required to write a program that utilizes a **stack** or a **queue**

1. to convert an infix expression to a prefix expression and
2. to evaluate a prefix expression.

Prefix expressions are another type of expression representations without parentheses. Some examples are given in Table 1.

Infix	**Prefix**
$2 + 3 * 4$	$+2 * 3\ 4$
$(1 + 2) * 7$	$* + 1\ 2\ 7$
$2 + 3 * (4 - 6) + 7$	$+ + 2 * 3 - 4\ 6\ 7$

Table 1: Some examples of the conversion from an infix expression to a prefix expression

Precedence	**Operators**
1	$+-$
2	$*/\%$

Table 2: The precedence of operators used in this problem

Your program should take the input from the standard input device (stdin), and we have the same assumptions used in the textbook - the expression contains only the binary operators $+$, $-$, $*$, $/$,and $\%$ (all integer operations, including divisions) and the operands

in the expression are single digit integers. See Table 2 for the precedence of all operators used in this problem. In addition, you will need to consider the case that an expression contain parentheses. Note that you have to directly evaluate the expression from the prefix expression (we will look at your source code). The maximum length of the input string (the infix expression) is 1024 characters and the result of expression evaluation will not be larger or smaller than what can be stored in a signed 32-bit integer.

Hint: this problem can be solved by using a stack.

Please use the following input/output format:

Input format: One single line which contains an infix expression.

Example:

```
2+3*(4-6)+7
```

Output format: Two lines. The first line shows the prefix expression converted from the infix expression and the second line shows the result of the expression evaluation.

Example:

```
++2*3-467
3
```

You must upload your homework in the format of a compressed zip file to the CEIBA, and the zip file should include the following three files:

1. The source code (.c file),
2. A shell script to compile the source (.sh), and
3. A document in PDF format to describe how your program/algorithm works.

Your score of 40% is divided into two parts: correctness (with 4 test cases)(32%) and explanations in the document(8%).

Problem 2. (30%) Explain how to use two stacks, A and B, to implement the functionalities of a queue. Assume that you can operate on the stacks with the following functions:

- *empty(Stack s)*; Check if stack s is empty. Return 1 if it is empty and 0 otherwise.
- *full(Stack s)*; Check if stack s is full. Return 1 if it is full and 0 otherwise.
- *push(Stack s, Element e)*; Push element e to the top of stack s.
- *pop(Stack s)*; Pop an element from the top of stack s and return the element. If stack s is empty, return error.

1. Please use the above functions to implement the following functions of the queue:

   - *empty(Queue q)*; Check if queue q is empty. Return 1 if it is empty and 0 otherwise. (4%)
   - *full(Queue q)*; Check if queue q is full. Return 1 if it is full and 0 otherwise. (4%)
   - *add(Queue q, Element e)*; Add element e to the rear end of queue q. If queue q is full, return error. (4%)
   - *delete(Queue q)*; Remove an element from the front end of queue s and return the element. If queue q is empty, return error. (8%)

   Describe the algorithm for each function with any language (English, Chinese, C, pseudo code, etc.).

2. (10%) What is the time complexity of your *delete* function? Please use the big-oh notation, $O(g(m))$, where m is the number of elements currently in the queue.

Problem 3. (30%) Suppose we use the following declarations for a program using singly linked lists.

```
typedef struct {
int number;
listNode *next;
} listNode;
```

```
listNode *lead=NULL;
listNode *list=NULL, *listB=NULL;
```

The variable *lead* points to the head of the singly linked list. Assume that the linked list *lead* is already sorted based on *number* in each *listNode* (from the largest to the smallest).

1. Please implement the function *insertSorted(listNode *new)* using the C language. The function inserts the new node (pointed by the parameter *new*) to the existing list *lead* in a way that the whole linked list is still sorted. Note that you also have to consider the cases that the linked list is empty (*lead*==*NULL*). (10%)

2. Please implement the function *divideList(int searchNum)* using the C language. The function divides the original linked list to three parts. (20%)

   - List pointed by the parameter *listA*. This list contains all the list nodes with *number*==*searchNum*.
   - List pointed by the parameter *listB*. This list contains all the list nodes with *number*<*searchNum*. However the list needs to be reversed (in other words, the list is sorted from the smallest to the largest).
   - List pointed by the parameter *lead*. This list contains all the list nodes from the original list but not the ones in the above two lists. The list still maintains its original order (from the largest to the smallest). See Figure 1 for an example. Note that all 3 lists could be empty in some cases and your function needs to handle them.

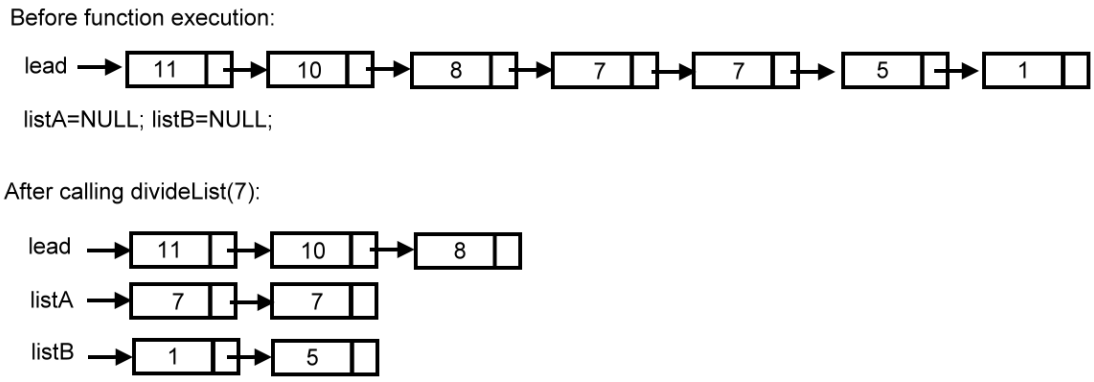

Figure 1: An example of the use of *divideList()*.