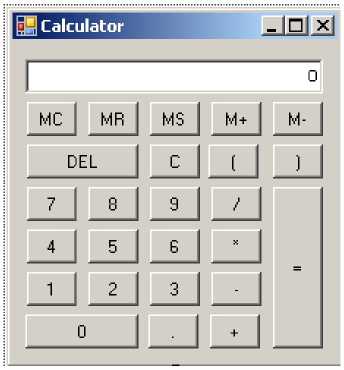
Faculty of Arts & Sciences
Department of Computer Science
CMPS 212—Intermediate Programming with Data Structures
Spring 2012-13
Mahmoud Bdeir

Lab 10
Due: 8:00 am, April 29, 2013

Problem 1: Write a simple calculator program with a GUI that performs the following arithmetic operations: Addition, Subtraction, Multiplication, and Division.

Infix to Postfix Conversion



Converting an infix expression to a postfix expression uses a technique called *operator precedence parsing,* which is commonly used in the syntax parsing phase of a compiler. The technique is quite simple and consists of scanning an expression left to right, and using an operator stack.

Assume that the only binary operators we are interested in are: + - * /
Both + and - have the same precedence and * and / have the same precedence, which is greater than that for + and -.

We also have to handle the fact that parenthesis override the regular precedence rules. We treat a left parenthesis as a higher precedence operator in the input, but a low precedence operator when on the operator stack.

Infix to Postfix Algorithm

1. Start scanning the infix expression left to right, ignoring whitespace.
2. For each operand encountered, just output the value.
3. If an operator is encountered on the input, we use an operator stack in the following manner:
   3.1. Initially the operator stack has a special end-of-input marker on the top of the stack. If the operator on the top of the stack has less precedence than the current operator, we just push the current operator onto the stack.
   3.2. If the operator on top has a higher or equal precedence, we pop it from the stack, send it to the output, and push the current operator onto the stack.
4. If we encounter a left parenthesis, we push it on the stack.
5. If we encounter a right parenthesis, we pop the stack and output all operators until we see a left parenthesis. The left and right parenthesis symbols are discarded.
6. The end-of-line marker in the input should be encountered only when the end-of-input marker is on the operator stack. This signifies the end of parsing. If not, the expression is not a proper expression.

Postfix Expression Evaluation Algorithm

1. When you see an operand, push it onto the stack.
2. When you see an operator:
   a. Pop the last two operands off of the stack.
   b. Apply the operator to them.
   c. Push the result onto the stack.
3. When you're done, the one remaining stack element is the result.

Instructions

1- Must write a class that will convert an expression from infix to postfix notation

2- Must write a class that will take a postfix expression and compute it
3- All features must be implemented including parenthesis and memory buttons
4- Calculation must be performed correctly
5- Use the Stack data structure provided by the Java Collections Framework
6- GUI must resemble the prototype shown above

Buttons:
MC: Memory Clear
MR: Memory Recall, outputs what is currently in memory
MS: Memory Store, store the current value in memory
M+: Add the current value to memory
M-: Subtract the current value from memory
C: Clear the current value

Problem 2:

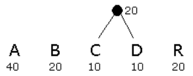
Phase I: Construct the binary tree to represent the Huffman Encoding of the letters

1. Read text file
2. Count the frequency of each letter occurring in the text, for example: A occurs 4,526 times. B occurs 8,136 time, etc.
3. Construct a binary tree to store all the letter frequencies following this algorithm:
    a. For each letter associate a frequency (number of times it occurs)
    b. Create a binary tree whose children are the encoding units with the smallest frequencies
    c. The frequency of the root is the sum of the frequencies of the leaves
    d. Repeat this procedure until all the encoding units are in the binary tree

For example, say we have the following frequency table:

C	D	B	R	A
10	10	20	20	40

1. Smallest number are 10 and 10 (C and D), so build the binary tree like this:

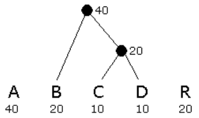


Step 1

2. C and D have already been used, and the new node above them (call it C+D) has value 20

C+D	B	R	A
20	20	20	40

3. The smallest values are B, C+D, and R, all of which have value 20
4. Connect any two of these, the binary tree now looks like this:

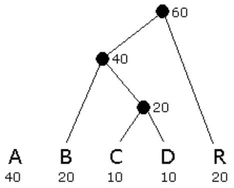


Step 4

5. The smallest values is R, while A and B+C+D all have value 40

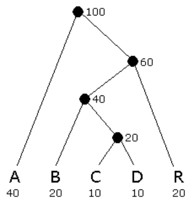
R	B+C+D	A
20	40	40

6. Connect R to either of the others:



Step 6

7. Connect the final two nodes

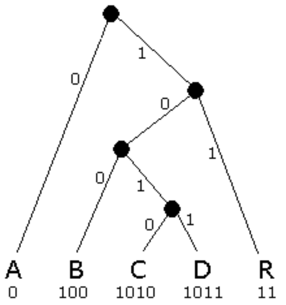


Step 7

Now you have a binary tree whose leaves are the letters used in the text

Phase II: Construct the dictionary that maps each letter used in the text to a sequence of ones and zeroes

1. Traverse the binary tree you built in Phase I using any traversal algorithm (LNR, NLR, LRN) building a string of ones and zeroes along the way:
    a. When you visit a nodes' left child you save a zero, when you visit a node's right child you save a one
    b. When you reach a leaf (a letter), you store the letter as a key in a map, then the string you have been building as the value:



A	0
B	100
C	1010
D	1011
R	11

Dictionary (Java Map)

Phase III: Create the compressed file

1. Use a `StringBuilder` class
2. For each letter in the original text add a string to the `StringBuilder` with the value obtained from the map, for example: if the first letter is A, you look that up in the dictionary, find 0 as value, and store the string 0 in the `StringBuilder`
3. Take eight characters (ones and zeroes) at a time for the string (use the `subSequence` method), convert that string into a byte and add that byte to a byte array:
    a. For example if the first subsequence is '01001101' then convert that into the integer 77, then cast that as a byte and store it into the array
4. The last subsequence may not be eight characters long, so pad it left with zeroes before converting it into a byte
5. Write the byte array to a new file on disk.
6. Compare the size of the new compressed file with that of the original file.