Stacks
(Section 2)

Linked list implementation of stack
Typical Application of stacks
Evaluation of expressions (infix, postfix, prefix)
References and further details

By:

Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Linked list implementation of stack

You have already seen array implementation of stack in which stack elements are stored in subsequent addresses in the main memory. To store/retrieve the elements, you just use the address of stack and then move up and down linearly. In array implementation, the memory addresses of the elements are continuous. One of the advantages of the continuous memory assignment is that, to search next element/space, you just either step down or step up. This gives easier access and fast execution (performance).

(fig - 1)

The major disadvantage of array implementation is that, the storage space is limited to programmer defined size only. Let's look at two cases for the worse scenario:

(i) If a programmer thinks that he needs lots of memory space for stack and makes an array of 65,536 bytes (64KB), and the program he made hardly have 2,000 elements of data type 'integer', then total space needed will be just 4,000 bytes. Rest of the memory space in the stack will just be of no use.
(ii) And if the programmer assigned 3,000 bytes in the stack but there are 2,000 elements (requires 4,000 bytes of space), then 500 of the elements will not get any room in the stack. This situation is called **stack overflow**. You might have seen this error in older MS-DOS programs.

So now, what if we could make some kind of mechanism so that we can assign any amount of memory space for stack elements whenever required??

This facility can be provided using linked list. Every element (known as a node) in a linked list have two parts:
(i) information part,
and (ii) address of next element.

To hold such two types of parts, one element must have two type of variables. These variables are kept encapsulated by using '**structure**'. Graphical representation of a typical linked list is given below:

(fig - 2)

Whenever a new element has to be added, a new room is created for the element and then linked. Here, as you can see, we can have dynamic allocation of the rooms required for the elements.

In short, a linked list is a list of elements scattered around the main memory using **dynamic memory allocation**.

We will discuss about linked list in detail in 'Lists' subsection.

Typical Application of stacks

The behavior of stack can be used for many useful tasks. Let's look at some of them.

Decimal to binary conversion

Decimal numbers are converted into binary by taking the mod with 2.

E.g. (fig - 3)

To form a binary number, we must arrange the remainders in opposite order. Therefore, there are two steps to convert from decimal to binary.

(i) calculate the remainders
(ii) arrange the remainders in suitable form

The burden of doing these two steps can be easily used by using stack.

General idea behind the implementing the decimal-binary conversion is to put the remainders into a stack as they appear and read back the stack and print the remainders.

(fig - 4)

Sample code: decimal to binary conversion (PROGRAM - 01)

```
void decimalToBinary(int n){
      int remainder;

      /* Let's find out the remainders and then put into the stack */
      while(n > 0){
            remainder = n % 2;
            push(remainder);
            n = n / 2;
      }

      /* we have finished putting the remainders in the stack, so now,
      just print the remainders */

      while(stack.tos >= 0){
            remainder = pop();
            printf("%d", remainder);
      }
}
```

Printing strings/characters in reverse order

As we saw for the decimal to binary number conversion, we just do the same method to print a given string in reverse order.

(fig - 5)

Sample code: printing strings in reverse order as user types the characters (PROGRAM - 02)

```
void main(){
```

```
        char c;

        do{
                c = getch();
                push(c);
        }while(c != '0');

        printf("\nThe reversed characters are : ");

        while(stack.tos > 0){
                c = pop();
                printf("%c", c);
        }
}
```

This method can be used to check whether a given string is **palindrome** or not? A palindrome string is that, it contains same characters either looking from left hand side or right hand side. Examples of palindrome are: NURSESRUN, WASITACATISAW, PALPA, MADAM etc. You can find lots of palindromes at http://www.cs.brown.edu/people/nfp/palindrome.html

Homework - 1: Write algorithm/pseudocode to check whether the given string is a palindrome string or not. (Using stack operations).

Parentheses checking in expressions

In computers, we used to provide syntax and expressions either in user shell or in other editors. Similarly, while developing or writing programs for user interfaces, we might encounter problems for expression **parsing**. Parsing is a method/process that is related to find out the actual meaning of the given string.

e.g.
in C, let's consider we want to write an statement, x = w / (y * z) + 2. If we write x = w / y * z + 2, then the value assigned to the variable 'x' will be different. And also if there we write x = w / (y * z + 2, then this statement does not hold any true value. This statement is said to be invalid as the parentheses are not matching. Validity of expressions for single type of parenthesis can be checked just by using a counter variable '**counter**'. Here, we scan each and every character in the string and increase the value of counter by 1 if open bracket is found and decrease the value of counter by 1 if the closing bracket is found. The expression will be valid if the value of counter is 0 at last and will be invalid if it is different from 0.

Example for parenthesis checking using counter method- blackboard exercise.

In this example, we tried for same type of parenthesis. What will be the case if there are 3 types of parentheses '(', '[', '{'. Then we might use 3 variables to check the validity. This method is not suitable as each and every time we have to determine which type of parenthesis we are currently working on. Further, to make worse case, if there is an expression as;

(x + [y *) z +]

Here, if we use the counter method, then we will get 0 count for each parentheses but, as you can see, the parentheses are not at proper place. In such case, stack can be used to make sure that there are correct number of parentheses in a given expression/statement at correct locations.

The general idea behind using stack for expression validity checking is that; every time an opening brace is found, we push the brace into stack. If we find a closing brace, then pop the element at the top of the stack, and match with the closing brace. The recently popped element is supposed to be open brace of same type. If it is of another type, then the expression is invalid. Let's look at an example;

(fig - 6)

Advantage

In counter method, the check the validity, we have to scan up to the end of the of the string. i.e. every characters need to be processed. This requires more processing time. But using stack, we can quit the checking job whenever we find a different brace.

Sample code: parenthesis checking in expressions (PROGRAM 03)

```
int isOpeningBrace(char c){
      if(c == '(' || c == '{' || c == '[')
            return 1;

      else
            return 0;
}

/* another version of isOpeningBrace */

int isOpeningBrace(char c){
      return (c == '(' || c == '{' || c== '[');
}

int samePair(char first, char second){

      if(first == '(' && second == ')')
            return 1;

      else if(first == '{' && second == '}')
            return 1;

      else if(first == '[' && second == '])
            return 1;

      else
            return 0;
}

/* Revised version of samePair */

int samePair(char first, char second){

      return ( (first == '(' && second == ')' ) || (first == '{' &&
            second == '}' ) || (first == '[' && second = ']' )  );

}

void main(){
      char *input;
      char element;
      int counter;

      printf("\nPlease provide an expression to check validity : ");
      gets(input);

      /* we are searching from the start of the string to end i.e.
```

```
      until the NULL character '\0' is not found */

      for(counter = 0; input[counter] != '\0'; counter++){

            if( isOpeningBrace(input[counter]))
                  push(input[counter]);

            else if(isClosingBrace(input[counter])){
                  element = pop();

                  if( ! samePair(element, input[counter])){
                        printf("\nThe expression doesn't contain right
                                    number of matching parentheses.");

                        return;
                  }

            }
      }

      if(stack.tos == 0)
            printf("\nThe expression contains right number of matching
            parentheses.);

      else
            printf("\nThe expression doesn't contain right number of
            matching parentheses.");

} // end of main program
```

Tracing of the program.

Evaluation of expressions (infix, postfix, prefix)

In expressions, we have **operands** and **operators**. Depending on the position of the operators, expressions are divided into three categories; infix, postfix, and prefix.

Infix

Generally, infix expressions are used in mathematics and other sciences. Here, operator lies between the operands.

e.g.
a + b
a * b - 6

Syntax: `<operand> <operator> <operand>`

The operands in right and left side of the operator must be a valid infix expression.

Postfix

In postfix expressions, the operator is kept after (to the right side of) the operands.

e.g.
a b +
a b + c - This expression is equivalent to a + b - c. and the same expression can be written as;
a b c - +

Syntax: `<operand> <operand> <operator>`

The operands should also be a valid postfix expression.

Prefix

In prefix expressions, the operator is kept before (to the left side of) the operands.

e.g.
+ a b
- + a b c, or + a - b c

Syntax: `<operator> <operand> <operand>`

The operands should also be valid prefix expression.

Class work:

Find out prefix and postfix of given expressions;

(a + b) / (c - d)
(a + (b / c - d) * e) + f
A $ B * C - D + E / F / (G + H)

Find infix expression of given expressions;

A B + C F $ / G E D + / -
A D E + B $ * -

You might be wondering that, people are using infix expressions in core mathematics, in C-code or any other programming languages, what is the use of prefix and postfix?

Precedence Order

Operators in infix expressions have precedence order. The evaluation of infix expressions depend on the precedence order. For example, precedence of '/' operator is higher than '+' operator. We are already used to the evaluation of infix expressions.

Order	Operator
1	uniary minus (negation), !
2	$ (power)
3	*, /, %
4	+, -
5	<, <=, >=, >
6	==, !=
7	&&
8	||

In prefix and postfix expressions, the precedence order doesn't hold.

Conversion

When higher level programming languages came into existence one of the major hurdles faced by the computer scientists was to generate machine language instructions which would properly evaluate any arithmetic expression. To convert a complex assignment statement such as:

X = A/B + C * D - F * G/Q

into a correct instruction sequence was a formidable task. That it is no longer considered so formidable is a tribute to the elegant and simple solutions that the computer scientists came out with. As of today this conversion is considered to be one of the minor aspects of compiler writing. To fix the order of evaluation of an expression each language assigns to each operator a priority. Even after assigning priorities how can a compiler accept an expression and produce correct code? The answer is given by reworking the expression into a form we call *postfix* notation. If *e* is an expression with operators and operands, the conventional way of writing *e* is called infix, because the operators come in between the operands (Unary operators precede their operand). The postfix form of an expression calls for each operator to appear after its operands. For example:

infix; A*B/C has a postfix form: AB*C/

If we study the postfix form we see that the multiplication comes immediately after its two operands A and B. Now imagine that A * B is computed and stored in T. Then we have the division operator '/' coming immediately after its two operands T and C.

Notice three features of the postfix expression:

- The operands maintain the same order as in the equivalent infix expression.
- Parentheses are not needed to designate the expression unambiguously.
- While evaluating the postfix expression the priority of the operators is no longer relevant.

Algorithm (PROGRAM 04)

Considerations:

- The string is saved in 'infixString' as; char infixString[20];
- There is an empty stack 'operatorStack' created to hold the operators and an empty string 'postFixString'.

1. Create empty operatorStack and empty postFixString.

2. Start from the left end of the infixString i.e. infixString[0].

3. Repeat steps 4 - 9 until we reach to the end of the string i.e. '\0' or NULL.

4. symbol = next character in the infixString. Use a variable to hold the status of the elements e.g. counter.

5. If symbol is a white-space, then discard it.

6. If symbol is an opening bracket, then push the symbol into operatorStack.

7. If the symbol is a closing bracket,

   Pop the values in the operatorStack and append to the postFixString until we do not find an opening bracket.

   After finding (popping) the opening bracket, discard both opening and closing brackets.

8. If symbol is an operator

   i. if operatorStack is empty, push the operator into the operatorStack.
   ii. else

   if(precedence of the operator at the top of the stack >= precedence of the current symbol), then

   pop and append the elements in the stack to the postFixString, until the precedence of operator in top of stack is greater or equal to the symbol.

   else i.e. the precedence is low

   push the operator into the operatorStack.

9. Else, (if the symbol is an operand)

   append the symbol to the postFixString.

10. If there are some elements remaining in the stack, then just pop and append them to the postFixString.

Tracing of the algorithm.

Let's trace the algorithm for ((A + B) / D) $ (E – F) * G

Step	Symbol	Postfix String	Operator Stack
1	(		(
2	(		(, (
3	A	A	(, (
4	+	A	(, (, +
5	B	AB	(, (, +
6	)	AB+	(
7	/	AB+	(, /
8	D	AB+D	(, /
9	)	AB+D/	
10	$	AB+D/	$
11	(	AB+D/	$, (
12	E	AB+D/E	$, (
13	-	AB+D/E	$, (, -
14	F	AB+D/EF	$, (, -
15	)	AB+D/EF-	$
16	*	AB+D/EF-$	*
17	G	AB+D/EF-$G	*

Now, we are at the end of the postfix string but there is still one operator left in the stack. So, pop the operator and append to the postfix string. Therefore, the postfix of ((A + B) / D) $ (E – F) * G is: **AB+D/EF-$G***

Evaluation

Infix expressions are evaluated by assigning the precedence values to the operators.

The implementation of infix evaluation in computers requires lots of jobs to do;

- first, the total number of operators and parentheses are counted and checked so that the expression is valid.
- then, operands associated with each and every operators and their precedence order is calculated
- now, every operators are applied to their respective operands.
- and finally, the total value comes.

High level programming languages like C, Java etc. support infix expressions. Infix expressions are more understandable than prefix and postfix expressions. But in digital computers, implementation of infix expression evaluation is a difficult task. The evaluation process is quite vague in the sense that if the length of expression is high, then lots of processing time is required for the evaluation of the expression.

New efficient algorithms/methods has been identified to simplify the evaluation of expressions.

Postfix expression evaluation

Most of the time, infix and prefix expressions are converted into postfix expressions and then postfix evaluation algorithms are applied. During the evaluation of expression, stack is used.

Algorithm (PROGRAM 05)

Considerations:

- The string is saved in 'postString' as; char postString[20];
- There is an empty stack created to hold the operands.

```
1.    Start from the left end of the postString i.e. postString[0].

2.    While (we are not at the end of the string), repeat steps 3, 4,
      and 5.

3.    Symbol = next character in the string.

4.    If symbol is an operand

            Push the operand into the stack.

5.    Else i.e. if the symbol is an operator

            operand2 = pop operand from stack.
            operand1 = pop next operand from stack

            Calculate the resulting value by using the formula;

            value = <operand1> <operator> <operand2>

            and put the value into the operand stack.

6.    Final value = pop from stack and print.
```

Tracing of the algorithm

AB + C $ DE – F *+ G –
Where, A = 3, B = 1, C = 2, D = 7, E = 4, F = 2, G =5
So now, the expression becomes: **3 1 + 2 $ 7 4 - 2 * + 5 -**

Step	Symbol	Operator1	Operator2	Result	Stack
1	3 (A)				3
2	1 (B)				3, 1
3	+	3	1	4	4
4	2 (C)				4, 2
5	$	4	2	16	16
6	7 (D)				16, 7
7	4 (E)				16, 7, 4
8	-	7	4	3	16, 3
9	2 (F)				16, 3, 2
10	*	3	2	6	16, 6
11	+	16	6	22	22
12	5				22, 5
13	-	22	5	17	17

Pop the value from stack and hence, the result is: 17.

References and further details

Look at references book to get more ideas on stack and its implementation.

Programming Issues

1. While writing programs, it is desirable to make a good user interface. One of the most popular problem between students while writing C program code is that, how to ask integer from a user. Most of the people use `scanf`, `atoi` functions to scan numbers directly or scan the string and then evaluate it to the required integer. But both of these two tricks fail for certain cases. Try to write you own function that reads only integers. If the user provides incorrect data, the program should not get down rather, it must display a polite error message like; 'Invalid Input, Please provide only integer data :" or similar.
2. As you already saw, array implementation of stack is not desirable. If you are planning to develop programs that use stack heavily, then think of implementing linked list.
3. Before converting a given infix expression, it is better to check the expression validity using the metrics similar to 'parentheses checking.'
4. If you are using algorithms that use stack during its operation, then make sure that the contents of stack is as it was before the algorithm. In postfix expression evaluation, after popping the final value off the stack, there should not be any element left in the stack. This feature can be used to make a program that checks validity of postfix expression. Similarly, if the stack goes underflow, then also the expression is not a valid operation.