Introduction to Stacks

In a stack data structure, all insertions and deletions of entries are made at one end, called the top of the stack. Stack is useful in applications involving reversing. When we add an item to a stack, we say that we push it onto the stack, and when we remove an item, we say that we pop it out the stack. Note that the last item pushed onto the stack is always the first that will be popped out the stack. This property is called last in, first out, or LIFO for short.

The two problems which arise when we add to or remove from a stack are stack overflow and stack underflow situations. We might attempt to pop an entry out an empty stack or to push an entry onto a full stack. These conditions must be recognized and reported with the return of an error code. If the stack is not full, item is added to the top of the stack. If the stack is full, an error code of overflow is returned and the stack is left unchanged. Similarly, if the stack is not empty, the top of the stack is returned in item. If the stack is empty an error code of underflow is returned.

Here is an example illustrating the stack operation.

```cpp
#include <iostream>
#include <conio>

const int MAXARRAY=10;  //stack size
int stack[MAXARRAY];  //an array to be used as stack
int top = 0;    //stack pointer
int item;    //an item to be pushed onto the stack
int flag;

void Push();
void Pop();
bool Empty();
bool Full();

int main ()
{
    int tag;
    flag = 1;
    while(flag != 0)
    {
        cout << "Enter 1 to push onto the stack\n";
        cout << "or enter 0 to pop out the stack: ";
        cin >> tag;
        cout << endl;
        switch(tag){
        case 1:
            cout << "Enter item: ";
            cin >> item;
            cout << endl;
            Push();
            break;
        case 0:
            Pop();
```
Pop();
break;
default:
   cout<<"Incorrect value entered."<<" Enter 0 or 1.\n\n";
}
for (int i=0; i<=top-1; i++)
   cout << stack[i] << " ";
cout << "\n\n";
cout << "Enter 0 if you want to exit,\n";
cout << "or enter any integer to continue: ";
cin >> flag;
cout << endl;
}
getch();
return 0;
}

//*****Push an item onto the stack***********************
void Push()
{
   if (Full() == false)
   {
      stack[top]=item;
top++;
   }
else
   cout << "Stack overflow!\n\n";
}

//*****Pop an item out the stack**************************
void Pop()
{
   if (Empty() == true)
      cout << "Stack underflow!\n\n";
   else
      --top;
}

//*****Check whether the stack is empty*****************
bool Empty()
{
   if (top<=0)
      return true;
   else
      return false;
}

//*****Check whether the stack is full*******************
bool Full()
{
   if (top>=MAXARRAY)
      return true;
   else
      return false;
}
Application of stacks: Polish notation

The method of writing all operators either before their operands, or after them, is called Polish notation, in honour of its discoverer, the Polish mathematician Jan Lukasiewicz. When the operators are written before their operands, it is called the prefix form. When the operators come after their operands, it is called the postfix form, or, sometimes, reverse Polish form or suffix form. Finally, in this context, it is customary to use the coined phrase infix form to denote the usual custom of writing binary operators between their operands.

The expression $a \times b$ becomes $\times ab$ in prefix form and $ab \times$ in postfix form. In the expression, $a + b \times c$, the multiplication is done first, so we convert it first, obtaining first $a + (bc \times)$ and then $abc \times +$ in postfix form.

The prefix form of this expression is $+(a \times bc)$. Note that prefix and postfix forms are not related by taking mirror images or other such simple transformations. Note also that parentheses have been omitted in the Polish forms.

The method of evaluating postfix expressions can be accomplished by the following four rules, which are repeatedly applied until all operators have been processed:

1. Find the leftmost operator in the expression.
2. Select the operand(s) immediately to the left of the operator found.
3. Perform the indicated operator.
4. Replace the operator and the operand(s) by the result.

The rules for evaluation of prefix expressions can be established in similar way.

Conversion of infix expressions to Polish notation

When we convert an expression in infix form into postfix form or prefix form we take into account the order in which the operators are carried out. Usually, we distinguish between unary and binary operations. The first requires a single operand for execution, while the latter takes two operands. We carry out the operators in order of the their priorities. The priorities of the operator are listed in the following table:

<table>
<thead>
<tr>
<th>Operators</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>^, all unary operators</td>
<td>6</td>
</tr>
<tr>
<td>\times / %</td>
<td>5</td>
</tr>
<tr>
<td>Symbol</td>
<td>Contents of the stack (rightmost symbol is on top)</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>+ - (binary)</td>
<td>4</td>
</tr>
<tr>
<td>== != &lt; &lt;= &gt; &gt;= &amp; &amp;</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>variable</td>
</tr>
</tbody>
</table>

In this table `^` and `||` denote exponentiation and OR operators, respectively.

Let us consider the problem of converting an infix expression containing parenthesized expression. The algorithm works in the following way:

1. Initialize stack contents to the left parentheses whose precedence number is greater than that of any other operator.
2. Scan the leftmost symbol in the given infix expression and denote it as the current input symbol.
3. Repeat through step 6 while the current symbol is not the last right parentheses, a counterpart of the left parentheses pushed onto the stack first.
4. Remove and output all stack symbols whose priority values are greater than or equal to the precedence of the current input symbol.
5. Push the current input symbol onto the stack.
6. Scan the leftmost symbol in the infix expression and let it be the current input symbol.

Algorithm begins with pushing left parentheses onto the stack. Then it scans the leftmost symbol and pushes it onto the stack. If the next symbol is the one whose priority is lower than the priority of the symbols in the stack, then we pop those symbols off the stack and add to the right of the postfix expression in the same order. Then push a new symbol onto the stack. This procedure continues until all the symbols of infix expression have been scanned, pushed onto the stack and then added to the postfix expression. If the right parentheses is encountered we pop all the symbol out the stack until the rightmost left parentheses and add them to the right of the expression.

Here is example illustrating the above algorithm.
Consider another example of conversion of infix form into postfix form. Let \( x = (-13 + (4^2 - 3 \times 2 + 6)^{1/2}) / (2 \times 5) \) be the given expression.

<table>
<thead>
<tr>
<th>No</th>
<th>Symbol</th>
<th>Stack</th>
<th>Postfix expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>=</td>
<td>(X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>~</td>
<td>(=~</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>(=−13</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>(=+)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(</td>
<td>(=+)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>(=+4)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>(=+)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>(=+2)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>−</td>
<td>(=−)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>(=−3)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>*</td>
<td>(=−*)</td>
<td>X13−4,2^3</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>(=−*2)</td>
<td>X13−4,2^3</td>
</tr>
<tr>
<td>15</td>
<td>+</td>
<td>(=+)</td>
<td>X13−4,2^3,2^*−6</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>(=+6)</td>
<td>X13−4,2^3,2^*−6</td>
</tr>
<tr>
<td>17</td>
<td>)</td>
<td>(=)</td>
<td>X13−4,2^3,2^*−6+</td>
</tr>
<tr>
<td>18</td>
<td>^</td>
<td>(=^2)</td>
<td>X13−4,2^3,2^*−6+</td>
</tr>
<tr>
<td>19</td>
<td>½</td>
<td>(=^½)</td>
<td>X13−4,2^3,2^*−6+</td>
</tr>
<tr>
<td>20</td>
<td>)</td>
<td>(=)</td>
<td>X13−4,2^3,2^*−6+^</td>
</tr>
<tr>
<td>21</td>
<td>/</td>
<td>(=/)</td>
<td>X13−4,2^3,2^*−6+^</td>
</tr>
<tr>
<td>22</td>
<td>(</td>
<td>(=/)</td>
<td>X13−4,2^3,2^*−6+^</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>(=/2)</td>
<td>X13−4,2^3,2^*−6+^</td>
</tr>
<tr>
<td>24</td>
<td>*</td>
<td>(=/*)</td>
<td>X13−4,2^3,2^*−6+^2</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>(=/*5)</td>
<td>X13−4,2^3,2^*−6+^2</td>
</tr>
<tr>
<td>26</td>
<td>)</td>
<td>(=)</td>
<td>X13−4,2^3,2^<em>−6+^2,5</em></td>
</tr>
<tr>
<td>27</td>
<td>)</td>
<td>(=)</td>
<td>X13−4,2^3,2^<em>−6+^2,5</em>/=</td>
</tr>
</tbody>
</table>
Expression trees

An expression tree is built up from the simple operands and operators of an (arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree, and the operators as the interior nodes. For each binary operator, the left tree contains left and right subtrees. For a unary operator, one subtree will be empty.

We traditionally write some unary operators to the left of their operands, such as “-” (unary negation) or the standard functions like \( \log() \), and \( \cos() \). Others are written on the right, such as the factorial function \( ! \), or the function that takes the square of a number, \( ()^2 \). Sometimes, either side is permissible, such as the derivative operator, which can be written as \( \frac{d}{dx} \) on the left, or the incrementing operator ++ in the C.

The expression tree of a simple expression is shown below. In this figure the operands and operators of the expression \( (a < b) \| (c < d) \) are assigned to the nodes of the binary tree, where || stands for logical OR operator.

Traversal of binary tree

In many applications it is necessary, not only to find a node within a binary tree, but to be able to move through all the nodes of the binary tree, visiting each one in turn. If there are \( n \) nodes in the binary tree, then there are \( n! \) different orders in which they could be visited, but most of these have little regularity or pattern. When we write an algorithm to traverse a binary tree we shall almost always wish to proceed so that the same rules are applied at each node. At a given node, then, there are three tasks we shall wish to do in some order: We shall visit the node itself; we shall traverse its left subtree; we shall traverse its right subtree. The tree distinguished traversals are: preorder (itself, left, right), inorder (left, itself, right), and postorder (left, right, itself).

As an example we can write down the prefix and postfix forms of an arithmetic expression, starting from its expression tree. Consider the figure given above. The
results of preorder, inorder and postorder traversals are as follows:

- Preorder: $\|< ab < cd$
- Inorder: $a < b || < cd$
- Postorder: $ab < cd <||$

Consider an example of $x = (-13 + (4^2 - 3 \times 2 + 1)^{1/2}) / (2 \times 5)$, that we analized in the previous section. Corresponding expression tree is demonstrated below.

Let us traverse the tree in postorder. The resulting expression is $X13 \sim 4, 2^4, 3, 2 \times -6 + 1/2 \times + 2, 5 \times /=$, which matches with the one we obtained in the previous section.