Chapter 43 - Stacks

The inspiration for the data structure known as a stack was, believe it or not, a cafeteria. More specifically, it was the push down stack of trays in a cafeteria. In a push down stack of trays, clean trays are added to the top and customers retrieve trays from the top. In fact, the only access to the stack is from the top! Stacks are an example of a last in, first out (LIFO) data structure.

Stacks are usually implemented as a linked list that is accessed from a pointer referred to as top. The function push places a node on the stack. The function pop removes a node from the stack and usually makes the information in the node available. For convenience, a Boolean function isEmpty may be included, which returns true when the stack is empty and false when the stack is not empty.

Stacks are most often created using a dynamic list. To implement a stack as a specialized dynamic list, a struct is needed. This will consist of the data fields and a pointer to the next node in the stack. The same basic node that is used in most linked lists is applicable here.

```
struct nodeType {
    dataType dataField1, dataField2, ... dataFieldn;
    nodeType *next;
};
```

Creation of the list structure is very simple, as usual. The top pointer must be initialized to NULL to create the empty stack.

```
nodeType *top;
top = NULL;
```

The function to determine if the stack is empty is very simple. If the pointer top is NULL, the stack is empty. If the stack is empty, the function returns true, else it should return false.

```
bool isEmpty() {
    if (top == NULL) return true;
    return false;
}
```
To push is just to add a new node to the top of the stack. The pointer top must be set to point to the new node. The new node's pointer to the next node must always be set to point to whatever top pointed to previously.

```c
nodeType *temp;

temp = new nodeType();
temp->dataField1 = data;
temp->dataField2 = data;
...
temp->dataFieldn = data;

temp->next = top;
top = temp;
```

Pop requires that the data be preserved so that it can be used. This can be done using a variable of nodeType. This can be a new variable or the same dynamic node. If a new node variable is used, the original one must be deleted. In either event, top must be updated to point to whatever the next pointer of the popped node points to. Of course, an empty stack cannot be popped.

```c
nodeType temp;
If (!isEmpty()) {
    temp = *top;
    temp.next = NULL;
    nodeType tempPointer;
    tempPointer = top;
    top = top->next;
    delete tempPointer;
}
```

Disposing of the stack is the same as any linked list.

```c
nodeType *temp;
while (top != NULL) {
    temp = top;
    top = top->next;
    delete temp;
}
```
Programming Exercise 43.1

Create a stack class based on the following struct. Use this class in a program to reverse the order of the words of any sentence that the user types in.

```c
struct nodeType {
    char name[25];
    noteType *next;
};
```

Programming Exercise 43.2

Contrary to popular opinion, many products are often not sold FIFO (First-In-First-Out, like in a queue), but are sold LIFO (Last-In-First-Out, like in a stack). For example, it is a common (though not universal) practice to sell bread in a LIFO stack, with the freshest product always being place at the front or most accessible part of the display. This way the grocer can always truthfully tell his / her customers that they are receiving the freshest possible food.

Given the following struct and file organization, create a struct class and use it to write a program that computes the average age (in days) of a loaf of bread when it is purchased. Assume that all bread on the shelf Monday morning is removed and the stack starts empty. If the transaction is a delivery (D), the bread is added to the stack. If the transaction is a sale (S), bread is removed from the stack and the number of days old is recorded. Create your own file of data for 7 days to test your program.

**Struct**

```c
struct transaction {
    char transType, dayDelivered;
    transaction *next;
}
```

**File Organization**

```
TranstypeDayTranstypeDay...
```

An example of the data file organization is

```
D1D1D1D1D1D1D1S1S1S1S2S2D3D3 ...
```

Note that

- D = DELIVERY, S = SALE
- 1..7 ARE MON - SUN
Programming Exercise 43.3

Railroad switching yards are responsible for putting train cars together in a desired order. A simple railroad switching yard consists of an incoming track that connects directly to an outgoing track, plus a side track that connects to the incoming and outgoing tracks and can be entered from either direction.

To change the order of cars in a train, cars can be moved directly back and forth from the incoming and outgoing sides. Cars from either side can also be moved to the siding or brought back from the siding to either side.

Create a stack class and queue based on a mutual struct. (Hint: The queue class will have to be able to dequeue and enqueue from both ends as needed.) Use these in a program to allow the user to reorder the cars in a train. The user should be able to

1. specify the length of the train to be generated in the incoming queue
2. pass the front car in the incoming queue to the back outgoing queue or push it onto a stack
3. pass the back car in the outgoing queue onto the front of the incoming queue or push it onto a stack
4. pop a car off the stack and onto the front of the incoming queue or the back of the outgoing queue

The contents of both queues and the stack should be displayed after each action the user takes.
Programming Exercise 43.4

Normally, people write and use mathematical expressions that have operators between operands, called *infix notation* (the operators inside the operands). For example:

\[ 2 + 3 \times 6 \]

is written in infix notation. To evaluate this example, a set of rules called the *order of operations* comes into play. According to these rules, multiplication must be done before addition, so this expression is evaluated as:

\[ 3 \times 6 \text{ equals } 18, \ 18 + 2 \text{ is equal to } 20, \text{ so the result of the expression is } 20. \]

If the writer of the expression had intended for 2 to be added to 3 first, the result multiplied times six, the order of operations for the infix notation expressions dictates that \(2+3\) must be enclosed in parenthesis. This gives

\[ (2 + 3) \times 6 \]

which is evaluated as

\[ 2 + 3 \text{ equals } 5, \ 5 \times 6 \text{ equals } 30, \text{ so the result of the expression is } 30. \]

The order of operations for infix notation consists of many other rules beyond what is used above. For example, how to handle unary operators (positive and negative signs). Students spend many years in math classes learning to master the order of operations for infix notation.

Since people almost exclusively use infix notation to write mathematical expressions, computer languages almost exclusively allow programmers to use infix notation. However, if a compiler allowed infix expressions into the binary code used in the compiled version of a program, the resulting code would be larger than needed and very inefficient. Because of this, compilers convert infix expressions into *postfix notation* expressions, which have a much simpler set of rules for expression evaluation.

Postfix notation gets its name from the fact that operators in a postfix expression follow the operands that they specify an operation on. Here are some examples of equivalent infix and postfix expressions:

<table>
<thead>
<tr>
<th>Infix Notation</th>
<th>Postfix Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 + 3 )</td>
<td>( 2 \ 3 + )</td>
</tr>
<tr>
<td>( 2 + 3 \times 6 )</td>
<td>( 3 \ 6 \times 2 + )</td>
</tr>
<tr>
<td>( (2 + 3) \times 6 )</td>
<td>( 2 \ 3 + 6 \times )</td>
</tr>
<tr>
<td>( A / (B \times C) + D \times E - A - C )</td>
<td>( A \ B \ C / \ D \ E \times + A \ C \times - )</td>
</tr>
</tbody>
</table>
Where as infix notation expressions need a long list or rules for evaluation, postfix expressions need very few. To evaluate a postfix expression requires use of a stack. Here's how to evaluate a postfix expression:

Moving from left to right,

1. examine the next token (an operator or operand)
2. if the token is an operand, push it on to the stack
3. if the token is an operator, pop the first two operands from the stack and perform the operation indicated by the operator, then push the results back onto the stack
4. repeat from 1 until all tokens are examined

When there are no tokens left in the expression, the answer is the sole remaining value in the stack. If there is more than one value left on the stack or if a pop occurs on an empty stack at any time, the original postfix expression was not valid.

Create a stack class that handles integer tokens. Use this class to create a program to evaluate postfix expressions. In this program, the user should be able to enter a postfix expression that can consist of integers and the operators +, -, *, / and ^ for addition, subtraction, multiplication, division and raise to a power.

**Programming Exercise 43.5**

Alter the program of 43.4 so that the tokens of the expression entered by the user are stored in a queue prior to its evaluation.

**Programming Exercise 43.6**

It is easy to see why a compiled program would use postfix expressions over infix. However, to make this possible, compilers must convert the usual human written infix expressions into postfix expressions. The idea behind this conversion is that if the infix expression was to be fully parenthesized, that is all operations and their operands were to be fully enclosed in parenthesis without changing the order of evaluation of the expression in any way, the equivalent postfix expression could be formed by moving all operators so that each takes the place of its corresponding right (closing) parenthesis and eliminating all left parenthesis. For example:

\[
A \times B + C / D \quad \text{becomes} \quad ( (A \times B) + (C / D) )
\]

which can be transformed

\[
( (A \times B) + (C / D) )
\]

which results in

\[
A B \times C D / +
\]
To save time, the step of fully parenthesizing an infix expression before moving the operators, then moving the operators, can be combined into one step by just moving the operators wherever a close parenthesis would be placed and not inserting the open parenthesis. To do this, priorities must be assigned to operators that are in the stack and in the infix queue. These will be:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority when in the Stack</th>
<th>Priority when in the Infix Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>)</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>^</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>*, /</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>+, - (add, subtract)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

The process of converting an infix expression to a postfix expression can be described as follows:

1. receive the infix expression in a queue of tokens
2. get the first token from the infix expression queue
3. if there is a token from the infix expression
   4. if the token is an operand, place it on the back of the queue that will contain the postfix expression
   5. else if the token is a close parenthesis, pop an operator token from the stack and write it to the to the queue that will contain the postfix expression until the operator found is an open parenthesis (do not write either parenthesis to the postfix queue!)
   6. else
      7. while the priority of the token on the stack top is greater than the priority of the incoming token from infix queue, pop the operator off of the stack and write it to the postfix queue
      8. push the token from the expression onto the stack top
      9. get another token from the infix expression queue and repeat from step 3
   10. pop all remaining operators from the stack and place them on the queue containing the postfix expression

Add the ability to receive an infix expression from the user to 43.5. Change the program so that the user only has to enter an infix expression, which the program then converts to postfix and evaluates it.