Stacks and Queues

Stacks

A stack is a basic data structure, where insertion and deletion of items takes place at one end called top of the stack. This structure is used all throughout programming. The basic concept can be illustrated by thinking of your data as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

A stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data. There are basically three operations that can be performed on stacks. They are 1) inserting an item into a stack (push). 2) deleting an item from the stack (pop). 3) displaying the contents of the stack (pip).

Below are some of operations a stack data type normally supports:

```plaintext
Stack<item-type> Operations

push(new-item:item-type)  
    Adds an item onto the stack.

top():item-type  
    Returns the last item pushed onto the stack.

pop()  
    Removes the most-recently-pushed item from the stack.

is-empty():Boolean  
    True if no more items can be popped and there is no top item.

is-full():Boolean  
    True if no more items can be pushed.

get-size():Integer  
```

---

en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queue#Converting_a_decimal_number_into_a_binary_number
Linked List Implementation

The basic linked list implementation is one of the easiest linked list implementations you can do. Structurally it is a linked list.

```plaintext
type Stack<item_type>
  data list:Singly Linked List<item_type>

constructor()
  list := new Singly-Linked-List()
end constructor

Most operations are implemented by passing them through to the underlying linked list. When you want to **push** something onto the list, you simply add it to the front of the linked list. The previous top is then "next" from the item being added and the list's front pointer points to the new item.

```plaintext
method push(new_item:item_type)
  list.prepend(new_item)
end method
```

To look at the **top** item, you just examine the first item in the linked list.

```plaintext
method top():item_type
  return list.get-begin().get-value()
end method
```

When you want to **pop** something off the list, simply remove the first item from the linked list.

```plaintext
method pop()
  list.remove-first()
end method
```

A check for emptiness is easy. Just check if the list is empty.

```plaintext
method is-empty():Boolean
  return list.is-empty()
end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```plaintext
method is-full():Boolean
  return False
end method
```

A check for the size is again passed through to the list.

```plaintext
method get-size():Integer
  return list.get-size()
end method
```
A real Stack implementation in a published library would probably re-implement the linked list in order to squeeze the last bit of performance out of the implementation by leaving out unneeded functionality. The above implementation gives you the ideas involved, and any optimization you need can be accomplished by inlining the linked list code.

Performance Analysis

In a linked list, accessing the first element is an $O(1)$ operation because the list contains a pointer that checks for empty/fullness as done here are also $O(1)$, depending on what time/space tradeoff is made. Most of the time, users of a Stack do not use the getSize() operation, and so a bit of space can be saved by not optimizing it.

Since all operations are at the top of the stack, the array implementation is now much, much better.

```java
public class StackArray implements Stack {
    protected int top;
    protected Object[] data;
    ...

    Assert.pre(!isFull(),"Stack is not full.");
```

will fail, raising an exception. Thus it makes more sense to implement with Vector (see StackVector) to allow unbounded growth (at cost of occasional O(n) delays).

Complexity:

All operations are O(1) with exception of occasional push and clear, which should replace all entries by null in order to let them be garbage-collected. Array implementation does not replace null entries. The Vector implementation does.

Applications of Stacks

Using stacks, we can solve many applications, some of which are listed below.

Converting a decimal number into a binary number

The logic for transforming a decimal number into a binary number is as follows:

* Read a number
* Iteration (while number is greater than zero)
However, there is a problem with this logic. Suppose the number, whose binary form we want to find is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the \textit{LIFO} property of the stack. Initially we push the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we pop one digit at a time from the stack and print it. Therefore we get the decimal number is converted into its proper binary form.

**Algorithm:**

1. Create a stack
2. Enter a decimal number, which has to be converted into its equivalent binary form.
3. iteration1 (while number > 0)
   - 3.1 digit = number \% 2
   - 3.2 Push digit into the stack
   - 3.3 If the stack is full
     - 3.3.1 Print an error
     - 3.3.2 Stop the algorithm
   - 3.4 End the if condition
   - 3.5 Divide the number by 2
4. End iteration1
5. iteration2 (while stack is not empty)
   - 5.1 Pop digit from the stack
   - 5.2 Print the digit
6. End iteration2
7. STOP

**Towers of Hanoi**

One of the most interesting applications of stacks can be found in solving a puzzle called Tower of Hanoi. According to an old Brahmin story, the existence of the universe is calculated in terms of the time taken by a number of monks, who are working all the time, to move 64 disks from one pole to another. But there are some rules about how this should be done, which are:

1. You can move only one disk at a time.
2. For temporary storage, a third pole may be used.
3. You cannot place a disk of larger diameter on a disk of smaller diameter.\footnote{1}

Here we assume that A is first tower, B is second tower & C is third tower.
Towers of Hanoi step 1
Towers of Hanoi step 2
Towers of Hanoi step 3
Output: (when there are 3 disks)

Let 1 be the smallest disk, 2 be the disk of medium size and 3 be the largest disk.

<table>
<thead>
<tr>
<th>Move disk</th>
<th>From peg</th>
<th>To peg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Output: (when there are 4 disks)

<table>
<thead>
<tr>
<th>Move disk</th>
<th>From peg</th>
<th>To peg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

The C++ code for this solution can be implemented in two ways:

**First Implementation (Without using Stacks)**

```cpp
void TowersofHanoi(int n, int a, int b, int c)
{
    //Move top n disks from tower a to tower b, use tower c for intermediate storage.
    if(n > 0)
    {
        TowersofHanoi(n-1, a, c, b);  //recursion
        cout << " Move top disk from tower " << a << " to tower " << b << endl ;
    }
}
```

en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues#Converting_a_decimal_number_into_a_binary_number
Second Implementation (Using Stacks)

```c
// Global variable , tower [1:3] are three towers
arrayStack<int> tower[4];
void TowerOfHanoi(int n)
{
    // Preprocessor for moveAndShow.
    for (int d = n; d > 0; d--)
        //initialize
tower[1].push(d);
    moveAndShow(n, 1, 2, 3);  /*move n disks from tower 1 to tower 3 using
tower 2 as intermediate tower*/
}

void moveAndShow(int n, int a, int b, int c)
{
    // Move the top n disks from tower a to tower b showing states.
    // Use tower c for intermediate storage.
    if(n > 0)
    {
        moveAndShow(n-1, a, c, b);  //recursion
        int d = tower[x].top();
        tower[x].pop();  //tower y
tower[y].push(d);
        showState();  //show state of 3 towers
        moveAndShow(n-1, c, b, a);  //recursion
    }
}
```

However complexity for above written implementations is O(2^n). So it's obvious that problem can only be solved for small values of n (generally n <= 30). In case of the monks, the number of turns taken to transfer 64 disks, by following the above rules, will be 18,446,744,073,709,551,615; which will surely take a lot of time!!

[1]

[2]

**Expression evaluation and syntax parsing**

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines.

**Evaluation of an Infix Expression that is Fully Parenthesized**

**Input:** \(((2 \ast 5) - (1 \ast 2)) / (11 - 9))

**Output:** 4

**Analysis:** Five types of input characters

* Opening bracket
* Numbers
* Operators
* Closing bracket
* New line character
Data structure requirement: A character stack

Algorithm

1. Read one input character
2. Actions at end of each input
   - Opening brackets
     - Push into stack and then Go to step (1)
   - Number
     - Push into stack and then Go to step (1)
   - Operator
     - Push into stack and then Go to step (1)
   - Closing brackets
     - Pop it from character stack
       - (2.4.1) if it is opening bracket, then discard it, Go to step (1)
       - (2.4.2) Pop is used three times
         - The first popped element is assigned to op2
         - The second popped element is assigned to op
         - The third popped element is assigned to op1
         - Evaluate op1 op op2
         - Convert the result into character and push into the stack
         - Go to step (2.4)
   - New line character
     - Pop from stack and print the answer
       - STOP

Result: The evaluation of the fully parenthesized infix expression is printed on the monitor as follows:

**Input String:** \(((2 \times 5) - (1 \times 2)) / (11 - 9))\)

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Stack (from bottom to top)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(((2</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(((2 *</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(((2 * 5</td>
<td>2 \times 5 = 10 and push</td>
</tr>
<tr>
<td>)</td>
<td>((10</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>((10 -</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>((10 - (</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>((10 - (1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>((10 - (1 *</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>((10 - (1 * 2</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>((10 - 2</td>
<td>1 \times 2 = 2 &amp; Push</td>
</tr>
<tr>
<td>)</td>
<td>(8</td>
<td>10 - 2 = 8 &amp; Push</td>
</tr>
<tr>
<td>/</td>
<td>(8 /</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(8 / (</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(8 / (11</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>(8 / (11 -</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(8 / (11 - 9</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>(8 / 2</td>
<td>11 - 9 = 2 &amp; Push</td>
</tr>
<tr>
<td>)</td>
<td>4</td>
<td>8 / 2 = 4 &amp; Push</td>
</tr>
<tr>
<td>New line</td>
<td>Empty</td>
<td>Pop &amp; Print</td>
</tr>
</tbody>
</table>

C Program
int main (int argc, char *argv[]) {
    struct ch *charactop;
    struct integer *integertop;
    char rd, op;
    int i = 0, op1, op2;
    charactop = cclearstack();
    integertop = iclearstack();
    while(1) {
        rd = argv[1][i++];
        switch(rd) {
        case '(': charactop = cpush(charactop, rd);
        case ')': integertop = ipop (integertop, &op1);
                  charactop = cpop (charactop, &op);
                  while(op != '(')
                    { integertop = ipush (integertop, eval(op, op1, op2));
                      charactop = cpop (charactop, &op);
                      if (op != '(')
                        { integertop = ipop(integertop, &op2);
                          integertop = ipop(integertop, &op1);
                        }
                    }
        break;
        case '\0': while (!= cemptystack(charactop))
                    { charactop = cpop(charactop, &op);
                      integertop = ipop(integertop, &op2);
                      integertop = ipop(integertop, &op1);
                      integertop = ipush(integertop, eval(op, op1, op2));
                    }
        integertop = ipop(integertop, &op1);
        printf("\n The final solution is: %d", op1);
        return 0;
        default: integertop = ipush(integertop, rd - '0');
        }
    }
}

int eval(char op, int op1, int op2) {
    switch (op) {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '/': return op1 / op2;
    case '*': return op1 * op2;
    }
}

Output of the program:

Input entered at the command line: (((2 * 5) - (1 * 2) / (11 - 9)) [3]

Evaluation of Infix Expression which is not fully parenthesized

Input: (2 * 5 - 1 * 2) / (11 - 9)

Output: 4

Analysis: There are five types of input characters which are:
We do not know what to do if an operator is read as an input character. By implementing the priority rule for operators, we have a solution to this problem.

The *Priority rule* we should perform comparative priority check if an operator is read, and then push it. If the stack *top* contains an operator of priority higher than or equal to the priority of the input operator, then we *pop* it and print it. We keep on performing the priority check until the *top* of stack either contains an operator of lower priority or if it does not contain an operator.

**Data Structure Requirement for this problem:** A character stack and an integer stack

**Algorithm:**

1. Read an input character
2. Actions that will be performed at the end of each input
   - Opening brackets
     - (2.1) *Push* it into stack and then Go to step (1)
   - Digit
     - (2.2) *Push* into stack, Go to step (1)
   - Operator
     - (2.3) Do the comparative priority check
       - (2.3.1) if the character stack's *top* contains an operator with equal or higher priority, then *pop* it into op
         - Pop a number from integer stack into op2
         - Pop another number from integer stack into op1
         - Calculate op1 op op2 and *push* the result into the integer stack
     - Closing brackets
       - (2.4) *Pop* from the character stack
       - (2.4.1) if it is an opening bracket, then discard it and Go to step (1)
       - (2.4.2) To op, assign the popped element
         - Pop a number from integer stack and assign it op2
         - Pop another number from integer stack and assign it to op1
         - Calculate op1 op op2 and push the result into the integer stack
         - Convert into character and *push* into stack
         - Go to the step (2.4)
   - New line character
     - (2.5) *Print* the result after popping from the stack

**Result:** The evaluation of an infix expression that is not fully parenthesized is printed as follows:

**Input String:** \((2 \times 5 - 1 \times 2) / (11 - 9)\)
<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Character Stack (from bottom to top)</th>
<th>Integer Stack (from bottom to top)</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>5</td>
<td>* has higher priority</td>
</tr>
<tr>
<td>-</td>
<td>*</td>
<td>2 5</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>*</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>10 1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>- *</td>
<td>10 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(- *)</td>
<td>10 1 2</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>(</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>/(</td>
<td>8 11</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>/(-)</td>
<td>8 11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>/(-)</td>
<td>8 11 9</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>/</td>
<td>8 2</td>
<td></td>
</tr>
<tr>
<td>New line</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C Program

```c
int main (int argc, char *argv[]) {  
    struct ch *charactop;  
    struct integer *integertop;  
    char rd, op;  
    int i = 0, op1, op2;  
    charactop = cclearstack();  
    integertop = iclearstack();  
    while(1) {  
        rd = argv[1][i++];  
        switch(rd) {  
        case '+':  
        case '-':  
        case '/':  
        case '*':  
            while ((charactop->data) != '(' && !emptystack(charactop))  
                {  
                    if(priority(rd) > {priority(charactop->data)  
                        break;  
                    else  
                        {  
                            charactop = cpop(charactop, &op);  
                            integertop = ipop(integertop, &op2);  
                            integertop = ipop(integertop, &op1);  
                            integertop = ipush(integertop, eval(op, op1, op2));  
                        }  
                    }  
                    charactop = cpush(charactop, rd);  
                    break;  
                case '(' : charactop = cpush(charactop, rd);  
                break;  
                case ')': integertop = ipop (integertop, &op2);  
                    integertop = ipop (integertop, &op1);  
                    charactop = cpop (charactop, &op);  
```
while (op != '(') {
    integerop = ipush (integerop, eval(op, op1, op2));
    charactop = cpop (charactop, &op);
    if (op != ')') {
        integerop = ipop (integerop, &op2);
        integerop = ipop (integerop, &op1);
    }
    break;
} 

\nWhile (\n != cemstack (charactop)) {
    charactop = cpop (charactop, &op);
    integerop = ipop (integerop, &op2);
    integerop = ipop (integerop, &op1);
    integerop = ipush (integerop, eval(op, op1, op2));
    integerop = ipop (integerop, &op1);
    printf (\n The final solution is: %d", op1);
    default: integerop = ipush (integerop, rd - '0');
}

int eval (char op, int op1, int op2) {
    switch (op) {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '/': return op1 / op2;
    case '*': return op1 * op2;
    }
}

int priority (char op) {
    switch (op) {
    case '^':
    case '$': return 3;
    case '*':
    case '/': return 2;
    case '+':
    case '-': return 1;
    }
}

Output of the program:
\nInput entered at the command line: (2 * 5 - 1 * 2) / (11 - 9)
\nOutput: 4 [3]

Evaluation of Prefix Expression
\nInput: x + 6 * ( y + z ) ^ 3
\nOutput: 4

Analysis: There are three types of input characters

* Numbers
* Operators
* New line character (\n)

Data structure requirement: A character stack and an integer stack

Algorithm:
1. Read one character input at a time and keep pushing it into the character stack until the new line character is reached.

2. Perform pop from the character stack. If the stack is empty, go to step (3)
   - Number
     2.1) Push in to the integer stack and then go to step (1)
     2.2) Assign the operator to op
       - Pop a number from integer stack and assign it to op1
       - Pop another number from integer stack
       - and assign it to op2
       - Calculate op1 op op2 and push the output into the integer stack. Go to step (2)

3. Pop the result from the integer stack and display the result

Result: The evaluation of prefix expression is printed as follows:

Input String: / - * 2 5 * 1 2 - 11 9

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Character Stack (from bottom to top)</th>
<th>Integer Stack (from bottom to top)</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>/ - * 2</td>
<td>/ - * 2 5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/ - * 2</td>
<td>/ - * 2 5 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>/ - * 2 5</td>
<td>/ - * 2 5 1 2</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>/ - * 2 5 * 1</td>
<td>/ - * 2 5 * 1 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>/ - * 2 5 * 1 2 -</td>
<td>/ - * 2 5 * 1 2 - 11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/ - * 2 5 * 1 2 - 11</td>
<td>/ - * 2 5 * 1 2 - 11 9</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>/ - * 2 5 * 1 2 - 11 9</td>
<td>/ 11</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/ - * 2 5 * 1 2 - 11 9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11 - 9 = 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2 2 1</td>
<td>2 2 5</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2 2 5</td>
<td>1 2 5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 * 2 = 2</td>
<td>2 5</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>2 5 2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2 2 10</td>
<td>5 * 2 = 10</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>2 8</td>
<td>10 - 2 = 8</td>
<td></td>
</tr>
<tr>
<td>Stack is empty</td>
<td>8 / 2 = 4</td>
<td>Print 4</td>
<td></td>
</tr>
</tbody>
</table>

C Program

```c
int main (int argc, char *argv[])
{
    struct ch *charactop = NULL;
    struct integer *integertop = NULL;
    char rd, op;
    int i = 0, op1, op2;
    charactop = cclearstack();
    integertop = iclearstack();
    ```
rd = argv[1][i];
while(rd != '0')
{
    charactop = cpush(charactop, rd);
    rd = argv[1][i+1];
}
while(!emptystack(charactop))
{
    charactop = cpop(charactop, rd);
    switch(rd)
    {
    case '+':
    case '-':
    case '/':
    case '*':
        op = rd;
        integertop = ipop(integertop, &op2);
        integertop = ipop(integertop, &op1);
        integertop = ipush(integertop, eval(op, op1, op2));
        break;
    default: integertop = ipush(integertop, rd - '0');
    }
}

int eval(char op, int op1, int op2)
{
    switch(op)
    {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '/': return op1 / op2;
    case '*': return op1 * op2;
    }
}

int priority (char op)
{
    switch(op)
    {
    case '^':
    case '$': return 3;
    case '*':
    case '/': return 2;
    case '+':
    case '-': return 1;
    }
}

Output of the program:

Input entered at the command line: / - * 2 5 * 1 2 - 11 9

Output: 4 [3]

Conversion of an Infix expression that is fully parenthesized into a Postfix expression

Input: (((8 + 1) - (7 - 4)) / (11 - 9))

Output: 8 1 + 7 4 - - 11 9 - /

Analysis: There are five types of input characters which are:

* Opening brackets
* Numbers
* Operators
* Closing brackets
* New line character (\n)

Requirement: A character stack

Algorithm:
1. Read an character input
2. Actions to be performed at end of each input
   - Opening brackets: Push into stack and then Go to step (1)
   - Number: Print and then Go to step (1)
   - Operator: Push into stack and then Go to step (1)
   - Closing brackets:
     - (2.4.1) If it is an operator, print it, Go to step (1)
     - (2.4.2) If the popped element is an opening bracket, discard it and go to step (1)
   - New line character: STOP

Therefore, the final output after conversion of an infix expression to a postfix expression is as follows:
<table>
<thead>
<tr>
<th>Input</th>
<th>Operation</th>
<th>Stack (after op)</th>
<th>Output on monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>( )</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>( )</td>
<td>)</td>
</tr>
<tr>
<td>8</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>(2.3) Push operator into stack</td>
<td>( )</td>
<td>( +</td>
</tr>
<tr>
<td>1</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '+' print it</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push operator into stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>7</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push the operator in the stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>4</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '-' print it</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>/</td>
<td>(2.3) Push the operand into the stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push into the stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>11</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push the operand into the stack</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>9</td>
<td>(2.2) Print it</td>
<td>( )</td>
<td>8</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '-' print it</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '/' print it</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '/' print it</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>New line character</td>
<td>(2.5) STOP</td>
<td></td>
<td>Stack is empty</td>
</tr>
</tbody>
</table>

**Rearranging railroad cars**

**Problem Description**

It's a very nice application of stacks. Consider that a freight train has \( n \) railroad cars. Each to be left at different station. They're numbered 1 through \( n \) & freight train visits these stations in order \( n \) through 1. Obviously, the railroad cars are labeled by their destination. To facilitate removal of the cars from the train, we must rearrange them in ascending order of their number(i.e. 1 through \( n \)). When cars are in this order, they can be detached at each station. We rearrange cars at a shunting yard that has **input track**, **output track** & \( k \) holding tracks between input & output tracks(i.e. **holding track**).
Solution Strategy

To rearrange cars, we examine the cars on the the input from front to back. If the car being examined is next one in the output arrangement , we move it directly to output track. If not , we move it to the holding track & leave it there until it's time to place it to the output track. The holding tracks operate in a LIPO manner as the cars enter & leave these tracks from top. When rearranging cars only following moves are permitted:

- A car may be moved from front (i.e. right end) of the input track to the top of one of the holding tracks or to the left end of the output track.
- A car may be moved from the top of holding track to left end of the output track.

The figure shows a shunting yard with $k = 3$, holding tracks $H1$, $H2$ & $H3$, also $n = 9$. The $n$ cars of freight train begin in the input track & are to end up in the output track in order 1 through $n$ from right to left. The cars initially are in the order 5,8,1,7,4,2,9,6,3 from back to front. Later cars are rearranged in desired order.

A Three Track Example

- Consider the input arrangement from figure , here we note that the car 3 is at the front, so it can't be output yet, as it to be preceded by cars 1 & 2. So car 3 is detached & moved to holding track $H1$.
- The next car 6 can't be output & it is moved to holding track $H2$. Because we have to output car 3 before car 6 & this will not possible if we move car 6 to holding track $H1$.
- Now it's obvious that we move car 9 to $H3$.

The requirement of rearrangement of cars on any holding track is that the cars should be preferred to arrange in ascending order from top to bottom.

- So car 2 is now moved to holding track $H1$ so that it satisfies the previous statement. If we move car 2 to $H2$ or $H3$, then we've no place to move cars 4,5,7,8. The least restrictions on future car placement arise when the new car $\lambda$ is moved to the holding track that has a car at its top with smallest label $\Psi$ such that $\lambda < \Psi$. We may call it an assignment rule to decide whether a particular car belongs to a specific holding track.
- When car 4 is considered, there are three places to move the car $H1,H2,H3$. The top of these tracks are 2,6,9. So using above mentioned Assignment rule, we move car 4 to $H2$.
- The car 7 is moved to $H3$.
- The next car 1 has the least label, so it's moved to output track.
- Now it's time for car 2 & 3 to output which are from $H1$(in short all the cars from $H1$ are appended to car 1 on output track).

The car 4 is moved to output track. No other cars can be moved to output track at this time.

- The next car 8 is moved to holding track $H1$.
- Car 5 is output from input track. Car 6 is moved to output track from $H2$, so is the 7 from $H3$,8 from $H1$ & 9 from $H3$.

Quicksort

Sorting means arranging a group of elements in a particular order. Be it ascending or descending, by cardinality or alphabetical order or variations thereof. The resulting ordering possibilities will only be limited by the type of the source elements.
Quicksort is an algorithm of the *divide and conquer* type. In this method, to sort a set of numbers, we reduce it to two smaller sets, and then sort these smaller sets.

This can be explained with the help of the following example:

Suppose A is a list of the following numbers:

```
48  36  12  60  84  98  44  65  108  24  96  72
```

In the reduction step, we find the final position of one of the numbers. In this case, let us assume that we have to find the final position of 48, which is the first number in the list.

To accomplish this, we adopt the following method. Begin with the last number, and move from right to left. Compare each number with 48. If the number is smaller than 48, we stop at that number and swap it with 48.

In our case, the number is 24. Hence, we swap 24 and 48.

```
24  36  12  60  84  98  44  65  108  48  96  72
```

The numbers 96 and 72 to the right of 48, are greater than 48. Now beginning with 24, scan the numbers in the opposite direction, that is from left to right. Compare every number with 48 until you find a number that is greater than 48.

In this case, it is 60. Therefore we swap 48 and 60.

```
24  36  12  48  84  98  44  65  108  60  96  72
```

Note that the numbers 12, 24 and 36 to the left of 48 are all smaller than 48. Now, start scanning numbers from 60, in the right to left direction. As soon as you find lesser number, swap it with 48.

In this case, it is 44. Swap it with 48. The final result is:

```
24  36  12  44  84  98  43  65  108  60  96  72
```

Now, beginning with 44, scan the list from left to right, until you find a number greater than 48.
Such a number is 84. Swap it with 48. The final result is:

```
24 36 12 44 48 98 84 65 108 60 96 72
```

Now, beginning with 84, traverse the list from right to left, until you reach a number lesser than 48. We do not find such a number before reaching 48. This means that all the numbers in the list have been scanned and compared with 48. Also, we notice that all numbers less than 48 are to the left of it, and all numbers greater than 48, are to it's right.

The final partitions look as follows:

```
24 36 12 44 48 98 77 65 108 60 96 72
```

Therefore, 48 has been placed in it's proper position and now our task is reduced to sorting the two partitions. This above step of creating partitions can be repeated with every partition containing 2 or more elements. As we can process only a single partition at a time, we should be able to keep track of the other partitions, for future processing.

This is done by using two stacks called LOWERBOUND and UPPERBOUND, to temporarily store these partitions. The addresses of the first and last elements of the partitions are pushed into the LOWERBOUND and UPPERBOUND stacks respectively. Now, the above reduction step is applied to the partitions only after it's boundary values are popped from the stack.

We can understand this from the following example:

Take the above list A with 12 elements. The algorithm starts by pushing the boundary values of A, that is 1 and 12 into the LOWERBOUND and UPPERBOUND stacks respectively. Therefore the stacks look as follows:

```
LOWERBOUND: 1
UPPERBOUND: 12
```

To perform the reduction step, the values of the stack top are popped from the stack. Therefore, both the stacks become empty.

```
LOWERBOUND: {empty}
UPPERBOUND: {empty}
```

Now, the reduction step causes 48 to be fixed to the 5th position and creates two partitions, one from position 1 to 4 and the other from position 6 to 12. Hence, the values 1 and 6 are pushed into the LOWERBOUND stack and 4 and 12 are pushed into the UPPERBOUND stack.

```
LOWERBOUND: 1, 6
UPPERBOUND: 4, 12
```

For applying the reduction step again, the values at the stack top are popped. Therefore, the values 6 and 12 are popped. Therefore the stacks look like:

```
LOWERBOUND: 1
UPPERBOUND: 4
```

en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queue#Converting_a_decimal_number_into_a_binary_number
The reduction step is now applied to the second partition, that is from the 6th to 12th element.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>84</td>
<td>65</td>
<td>108</td>
<td>60</td>
<td>96</td>
<td>72</td>
</tr>
<tr>
<td>72</td>
<td>84</td>
<td>65</td>
<td>108</td>
<td>60</td>
<td>96</td>
<td>98</td>
</tr>
<tr>
<td>72</td>
<td>84</td>
<td>65</td>
<td>98</td>
<td>60</td>
<td>96</td>
<td>108</td>
</tr>
<tr>
<td>72</td>
<td>84</td>
<td>65</td>
<td>96</td>
<td>60</td>
<td>98</td>
<td>108</td>
</tr>
</tbody>
</table>

After the reduction step, 98 is fixed in the 11th position. So, the second partition has only one element. Therefore, we push the upper and lower boundary values of the first partition onto the stack. So, the stacks are as follows:

**LOWERBOUND:** 1, 6  
**UPPERBOUND:** 4, 10

The processing proceeds in the following way and ends when the stacks do not contain any upper and lower bounds of the partition to be processed, and the list gets sorted.

**The Stock Span Problem**

In the stock span problem, we will solve a financial problem with the help of stacks.

Suppose, for a stock, we have a series of $n$ daily price quotes, the *span* of the stock's price on a particular day is defined as the maximum number of consecutive days for which the price of the stock on the current day is less than or equal to its price on that day.

**An algorithm which has Quadratic Time Complexity**

**Input:** An array $P$ with $n$ elements

**Output:** An array $S$ of $n$ elements such that $P[i]$ is the largest integer $k$ such that $k \leq i + 1$ and $P[y] \leq P[i]$ for $j = i - k + 1, \ldots, i$

**Algorithm:**

1. Initialize an array $P$ which contains the daily prices of the stocks
2. Initialize an array $S$ which will store the span of the stock
3. for $i = 0$ to $i = n - 1$
   3.1 Initialize $k$ to zero
   3.2 Done with a false condition
   3.3 repeat
       3.3.1 if ( $P[i - k] \leq P[i]$ ) then
           Increment $k$ by 1
       3.3.2 else
           Done with true condition
   3.4 Till ($k > i$) or done with processing
     Assign value of $k$ to $S[i]$ to get the span of the stock
4. Return array $S$
Now, analyzing this algorithm for running time, we observe:

- We have initialized the array S at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time.

- The repeat loop is nested within the for loop. The for loop, whose counter is $i$ is executed $n$ times. The statements which are not in the repeat loop, but in the for loop are executed $n$ times. Therefore these statements and the incrementing and condition testing of $i$ take $O(n)$ time.

- In repetition of $i$ for the outer for loop, the body of the inner repeat loop is executed maximum $i + 1$ times. In the worst case, element $S[i]$ is greater than all the previous elements. So, testing for the if condition, the statement after that, as well as testing the until condition, will be performed $i + 1$ times during iteration $i$ for the outer for loop. Hence, the total time taken by the inner loop is $O(n(n + 1)/2)$, which is $O(n^2)$.

The running time of all these steps is calculated by adding the time taken by all these three steps. The first two terms are $O(n)$ while the last term is $O(n^2)$. Therefore the total running time of the algorithm is $O(n^2)$.

**An algorithm that has Linear Time Complexity**

In order to calculate the span more efficiently, we see that the span on a particular day can be easily calculated if we know the closest day before $i$, such that the price of the stocks on that day was higher than the price of the stocks on the present day. If there exists such a day, we can represent it by $h(i)$ and initialize $h(i)$ to be -1.

Therefore the span of a particular day is given by the formula, $s = i - h(i)$.

To implement this logic, we use a stack as an abstract data type to store the days $i$, $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to $i$, we pop the days when the price of the stock was less than or equal to $p(i)$ and then push the value of day $i$ back into the stack.

Here, we assume that the stack is implemented by operations that take $O(1)$ that is constant time. The algorithm is as follows:

**Input:** An array $P$ with $n$ elements and an empty stack $N$

**Output:** An array $S$ of $n$ elements such that $P[i]$ is the largest integer $k$ such that $k <= i + 1$ and $P[y] <= P[i]$ for $j = i - k + 1, \ldots, i$

**Algorithm:**

1. Initialize an array $P$ which contains the daily prices of the stocks
2. Initialize an array $S$ which will store the span of the stock
3. for $i = 0$ to $i = n - 1$
   3.1 Initialize $k$ to zero
   3.2 Done with a false condition
   3.3 while not (Stack $N$ is empty or done with processing)
      3.3.1 if ($P[i] >= P[N.top()]$) then
      Pop a value from stack $N$
      3.3.2 else
      Done with true condition
3.4 if Stack $N$ is empty
   3.4.1 Initialize $h$ to -1
3.5 else
   3.5.1 Initialize stack top to $h$
   3.5.2 Put the value of $h - i$ in $S[i]$
   3.5.3 Push the value of $i$ in $N$

4. Return array $S$

Now, analyzing this algorithm for running time, we observe:

- We have initialized the array $S$ at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time.
The while loop is nested within the for loop. The for loop, whose counter is \( i \) is executed \( n \) times. The statements which are not in the repeat loop, but in the for loop are executed \( n \) times. Therefore these statements and the incrementing and condition testing of \( i \) take \( O(n) \) time.

Now, observe the inner while loop during \( i \) repetitions of the for loop. The statement done with a true condition is done at most once, since it causes an exit from the loop. Let us say that \( t(i) \) is the number of times statement Pop a value from stack \( N \) is executed. So it becomes clear that while not (Stack \( N \) is empty or done with processing) is tested maximum \( t(i) + 1 \) times.

Adding the running time of all the operations in the while loop, we get:

\[
\sum_{i=0}^{n-1} t(i) + 1
\]

An element once popped from the stack \( N \) is never pushed back into it. Therefore,

\[
\sum_{i=1}^{n-1} t(i)
\]

So, the running time of all the statements in the while loop is \( O(n) \)

The running time of all the steps in the algorithm is calculated by adding the time taken by all these steps. The run time of each step is \( O(n) \). Hence the running time complexity of this algorithm is \( O(n) \).

**Related Links**

- Stack (Wikipedia)

**Queues**

A queue is a basic data structure that is used throughout programming. You can think of it as a line in a grocery store. The first one in the line is the first one to be served. Just like a queue.

A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data.

```
Queue<item-type> Operations

enqueue(new-item:item-type)
    Adds an item onto the end of the queue.
front():item-type
    Returns the item at the front of the queue.
dequeue()
    Removes the item from the front of the queue.
is-empty():Boolean
    True if no more items can be dequeued and there is no front item.
is-full():Boolean
    True if no more items can be enqueued.
get-size():Integer
    Returns the number of elements in the queue.
```

All operations except $\text{get-size()}$ can be performed in $O(1)$ time. $\text{get-size()}$ runs in at worst $O(N)$.

Linked List Implementation

The basic linked list implementation uses a singly-linked list with a tail pointer to keep track of the back of the queue.

```plaintext

type Queue<item_type>  
data list:Singly Linked List<item_type>  
data tail:List Iterator<item_type>  

constructor()  
list := new Singly-Linked-List()  
tail := list.get-begin() # null  
end constructor
```

When you want to $\text{enqueue}$ something, you simply add it to the back of the item pointed to by the tail pointer. So the previous tail is considered next compared to the item being added and the tail pointer points to the new item. If the list was empty, this doesn't work, since the tail iterator doesn't refer to anything.

```plaintext

method enqueue(new_item:item_type)  
if is-empty()  
list.prepend(new_item)  
tail := list.get-begin()  
else  
list.insert_after(new_item, tail)  
tail.move-next()  
end if  
end method
```

The $\text{front}$ item on the queue is just the one referred to by the linked list's head pointer.

```plaintext

method front():item_type  
return list.get-begin().get-value()  
end method
```

When you want to $\text{dequeue}$ something off the list, simply point the head pointer to the previous from head item. The old head item is the one you removed of the list. If the list is now empty, we have to fix the tail iterator.

```plaintext

method dequeue()  
list.remove-first()  
if is-empty()  
tail := list.get-begin()  
end if  
end method
```

A check for emptiness is easy. Just check if the list is empty.

```plaintext

method is-empty():Boolean  
return list.is-empty()  
end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```plaintext

method is-full():Boolean
```

en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queue#Converting_a_decimal_number_into_a_binary_number
return False
end method

A check for the size is again passed through to the list.

```java
method get-size():Integer
    return list.get-size()
end method
end type
```

Performance Analysis

In a linked list, accessing the first element is an $O(1)$ operation because the list contains a pointer directly to it. Therefore, enqueue, front, and dequeue are a quick $O(1)$ operations.

The checks for empty/fullness as done here are also $O(1)$.

The performance of `get-size()` depends on the performance of the corresponding operation in the linked list implementation. It could be either $O(n)$, or $O(1)$, depending on what time/space tradeoff is made. Most of the time, users of a Queue do not use the `get-size()` operation, and so a bit of space can be saved by not optimizing it.

Circular Array Implementation

Performance Analysis

Priority Queue Implementation

Related Links

- Queue (Wikipedia)

Deques

As a start

- Deque (Double-Ended QUEue)

Data Structures

Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

References

1. ↑ Dromey, R.G. *How to Solve it by Computer*. Prentice Hall of India.
2. ↑ Data structures, Algorithms and Applications in C++ by Sartaj Sahni
