Stacks

1 Introduction

Stacks are probably the single most important data structure of computer science. They are used across a broad range of applications and have been around for more than fifty years, having been invented by Friedrich Bauer in 1957.

A ***stack*** is a list in which insertions and deletions are allowed only at the front of the list. The front in this case is called the ***top***, insertions are called ***push*** operations, and deletions are called ***pop*** operations. Because the item at the top of the stack is always the one most recently inserted into the list, and is also the first one to be removed by a pop operation, the stack is called a ***last-in-first-out*** list, or a ***LIFO*** list for short, since the last item in is the first item out.

The restriction of insertions and deletions to the front of the list is not the only difference between stacks and general lists. Stacks do not support searching or access to anywhere else in the list. In other words, the only element that is accessible in a stack is the top element.

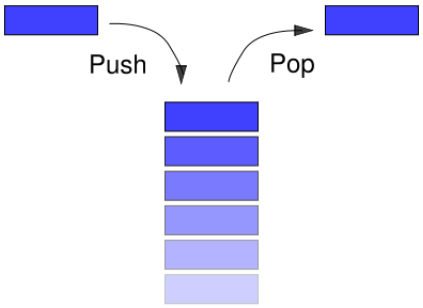

Figure 1: A stack

2 The Stack ADT

The fundamental methods that can be performed on a stack are

create() – Create an empty stack.

destroy() – Destroy a stack.

empty() – Return true if the stack is empty, otherwise return false.

push(new_item) - Add a new item to the top of the stack.

item pop() – Remove the top element from the stack and return it to the caller.

item top() – Return the top element of the stack without removing it.

The reason that the word "stack" is appropriate for this abstract data types is that it is exactly like a stack of trays in a cafeteria, the kind that are spring-loaded. New trays (presumably washed) are put on top of the stack. The next tray to be removed is the last put on. This is why the operations are called pop and push, because it conjures up the image of the spring-loaded stack of trays. We literally push against the spring, which wants to pop one off.

3 Refining the Stack ADT

The operations, written more formally using a slight modification of UML, are:

```
+empty():boolean  {query}
// returns true if the stack is empty, false if not
+push(in new_item:StackItemType): void  throw stack_exception
// pushes new_item on top of the stack; throws a stack_exception if
// it fails
+pop(): void  throw stack_exception
// removes the top element from the stack; if it fails
// it throws an exception (if the stack is empty for example)
+pop(out stack_top:StackItemType): void  throw stack_exception
// same as pop() above but stores the item removed into stack_top,
// throwing an exception if it fails
+top( out stack_top:StackItemType): void {query} throw stack_exception
// stores into stack_top the item on top of the stack,
// throwing an exception if it fails.
```

Notes.

1. Notice that there are two versions of `pop()`, one that has no parameters and no return value, and one that has an *out* parameter and no return value. In theory, when the stack is popped, we think of it as a deletion operation, discarding whatever was on the top of the stack. In practice, we may sometimes want to inspect the item that was deleted, and so the second version of `pop()` could be used to retrieve that top element and look at it after we have removed it from the stack.
2. The `top()` method has an out parameter. It could instead return the top element as its return value, but it is considered better coding style to pass it out through the parameter list, as otherwise it could be used in an expression with side effects.
3. The `pop()` and `top()` methods declare that they may throw an exception of type `stack_exception`, and that this is the only type of exception that they may throw. If a program tries to pop an empty stack, or to retrieve the top element of an empty stack, this is considered to be an error condition. Now that we have added C++ exception handling to our toolbox (in the preceding chapter), we can use it as a means of notifying the client software that an error has occurred.
4. The `push()`, also may throw an exception. No computer has infinite memory, and it is possible that when the stack tries to allocate more memory to store this new item, it fails. The exception in this case would indicate that the stack is "full."

4 Applications of Stacks

Stacks are useful for any application requiring LIFO storage. There are many, many of these.

- parsing context-free languages
- evaluating arithmetic expressions
- function call management
- traversing trees and graphs (such as depth first traversals)
- recursion removal

Example: A Language Consisting of Balanced Brackets

A stack can be used to recognize strings from the language of ***balanced brackets***. Brackets are either parentheses, curly braces, square brackets, angle brackets, or any other pair of characters designated to be left and right matches of each other, as well as characters that are not brackets of any kind. Brackets are balanced when they are used in matching pairs. To be precise, the following grammar defines a language of balanced brackets whose alphabet is the four different types of brackets and the letter `a`. The word "`NULLSTRING`" means the string with no characters in it.

```
word       = identifier
           | NULLSTRING
           | word word
           | (word) | <word> | [word] | {word}
identifier = a
```

The grammar generates the null string of course, as well as the word with the single letter 'a'. You can check that the following two-character words are also in the language that this grammar generates:

```
() {} [] <>
```

because there are derivations such as

```
word => (word) => (NULLSTRING) => ()
```

and similar ones for the other brackets. In general, this grammar generates words such as

```
aa(aa)[a(aa<aaa>a)aaaaaa]aa{aa{aaa}{aa{aaa}}aa}
```

Intuitively, these are words that never have overlapping brackets such as `[ ( ] )` or unmatched brackets such as `(aaa`.

The simplest bracket language has just a single type of bracket. Suppose we allow all lowercase letters into the alphabet but limit the brackets to curly braces. An example of a word with balanced braces is:

```
abc{defg{ijk}{l{mn}}op}qr
```

An example of a word with unbalanced braces is:

```
abc{def}}{ghij{kl}m
```

The following is an algorithm to recognize words in this language. **Recognizing** means identifying which words are in the language and which are not. A **recognition algorithm** returns either true or false, depending on whether the word is in the language.

```
initialize a new empty stack s whose entries hold a single character.
balanced = true;
while ( the end of the string has not been reached ) {
    copy the next character of the string into ch;
    if ( ch == "{" )
        s.push(ch);
    else if ( ch == "}" ) {
```

```
            if s.empty()
                return !balanced;
            else
                s.pop();
        }
    }
    if s.empty()
       return balanced;
    else
       return !balanced;
```

Of course this is a simple problem that can be solved with just a counter; increment the counter on reading a "{" and decrement on reading a "}". If the counter is ever negative, it is not balanced. If it is not zero when the end of the string is reached, it is not balanced. Otherwise it is balanced.

Exercise 1. Write and implement a recognition algorithm for the above language that just uses a counter rather than a stack.

The stack is needed when there are different types of matching parentheses. For example, suppose the string can have square brackets, curly braces, and regular parentheses, as in:

```
(abc)[ {d (ef) g{ijk}{l{mn}}op} ] qr
```

Then a counter solution will not work but a stack will. The idea is that when a right bracket is found, the stack is popped only if the bracket on top of the stack matches it. If it does not match, the string is not balanced. If the stack is empty, it is not balanced. When the entire string has been read, if the stack is not empty, it is not balanced. Assume that the function `matches(ch1, ch2)` returns true if `ch1` and `ch2` are matching brackets of any kind.

```
initialize a new empty stack s whose entries hold a single character.
balanced = true;
while ( the end of the string has not been reached ) {
    copy the next character of the string into ch;
    if ( ch == '[' || ch == '(' || ch == '{' )
        s.push(ch);
    else
        switch ( ch ) {
        case ']' :
        case ')' :
        case '}' :
           if s.empty()
               return !balanced;
           else {
               s.top( bracket );
               s.pop();
               if ( !matches(bracket, ch) )
                   return !balanced;
           }
           break;
        default: // a non-bracket -- ignore
    }
}
if s.empty()
   return balanced;
else
   return !balanced;
```

5 Stack Implementations

We did not need to know how a stack was implemented to use the stack to solve this problem. This is the power of data abstraction. But before we look at other applications, we will consider how one can implement a stack in C++. To do this, we need to make the interface precise enough so that we can implement the operations. In addition, because we will start to use exceptions to handle error conditions, we will create a class of stack exceptions that can be thrown as needed. If you are not familiar with exceptions, refer to the notes on exception handling.

The following exception class will be used by all stack implementations.

```
#include <stdexcept>
#include <string>
using namespace std;
class stack_exception: public logic_error
{
public:
    stack_exception(const string & message="")
         : logic_error(message.c_str())    {}
};
```

The `logic_error exception` is defined in `<stdexcept>`. Its constructor takes a `string`, which can be printed using the `what()` method of the class. Our `stack_exception` class is derived from this `logical_error` class and inherits the `what()` method. The constructor for our `stack_exception` can be given a string, which will be passed to the `logical_error` constructor as its initializing string.

The next logical step could be to derive separate exception types such as `stack_overflow`, or `stack_underflow`, as illustrated below.

```
class stack_overflow: public stack_exception
{
public:
    stack_overflow(const string & message=""): stack_exception(message) {}
};

class stack_underflow: public stack_exception
{
public:
    stack_underflow(const string & message=""): stack_exception(message) {}
};
```

but for simplicity we will not do this. The following is the specific interface we will implement:

```
typedef data_item StackItemType;
class Stack
{
public:
    // constructors and destructor:
    Stack();               // default constructor
    Stack(const Stack &);  // copy constructor
    ~Stack();              // destructor
```

```
    // stack operations:
    bool empty() const;
    // Determines whether a stack is empty.
    // Precondition: None.
    // Postcondition: Returns true if the stack is empty;
    // otherwise returns false.

    void push(const StackItemType& new_item) throw(stack_exception);
    // Adds an item to the top of a stack.
    // Precondition: new_item is the item to be added.
    // Postcondition: If the insertion is successful, new_item
    // is on the top of the stack.
    // Exception: Throws stack_exception if the item cannot
    // be placed on the stack.


    void pop() throw(stack_exception);
    // Removes the top of a stack.
    // Precondition: None.
    // Postcondition: If the stack is not empty, the item
    // that was added most recently is removed. However, if
    // the stack is empty, deletion is impossible.
    // Exception: Throws stack_exception if the stack is empty.


    void pop(StackItemType& top_item) throw(stack_exception);
    // Retrieves and removes the top of a stack.
    // Precondition: None.
    // Postcondition: If the stack is not empty, top_item
    // contains the item that was added most recently and the
    // item is removed. However, if the stack is empty,
    // deletion is impossible and top_item is unchanged.
    // Exception: Throws stack_exception if the stack is empty.


    void top(StackItemType& top_item) const
          throw(stack_exception);
    // Retrieves the top of a stack.
    // Precondition: None.
    // Postcondition: If the stack is not empty, top_item
    // contains the item that was added most recently.
    // However, if the stack is empty, the operation fails
    // and top_item is unchanged. The stack is unchanged.
    // Exception: Throws stack_exception if the stack is empty.
private:
   ...
};
```

There are several different ways to implement a stack, each having advantages and disadvantages.

Since a stack is just a special type of list, a stack can be implemented with a list. In this case, you can make a list a private member of the class and implement the stack operations by calling the list methods on the private list. In other words, we could do something like

```
class Stack
```

```
{
public:
    // public methods here
private:
    List items;
};
```

We could implement all of the member functions by using the `List` member functions of the hidden `List` named items.

It is more efficient to implement a stack directly, avoiding the unnecessary function calls that result from "wrapping" a stack class around a list class. Direct methods include using an array, or using a linked representation, such as a singly-linked list.

A stack implemented as an array is efficient because the insertions and deletions do not require shifting any data in the array. The only problem is that the array size might be exceeded. This can be handled by a resizing operation when it happens.

A stack implemented as a linked list overcomes the problem of using a fixed size data structure. It is also relatively fast, since the push and pop do not require many pointer manipulations. The storage is greater because the links take up extra bytes for every stack item.

5.1 Array Implementation

In the array implementation, two private data members are needed by the class: the array of items and an integer variable named `top`.

```
class Stack
{
public:
    // same interface as above
private:
    StackItemType  items[MAX_STACK];   // array of MAX_STACK many items
    int            top_index;          // indicates which index is the top
};
```

Remarks. The implementation is very straightforward:

- The variable `top_index` stores the array index of the current top of the stack. We cannot name it `top` because there is a method named `top`. Because 0 is the first array index, we cannot use 0 to mean that the stack is empty, because if `top_index == 0`, it means there is an item in `items[0]`. Therefore, `-1` is the value of `top_index` that denotes the empty stack. This implies that the constructor must initialize `top_index` to `-1`, and the `empty()` function must check if it is `-1` as well.
- The `push()` operation has to check if the maximum array size will be exceeded if it adds a new item. If so, it has to throw an exception to indicate this. If not, it increments `top` and copies the new item into `items[top_index]`.
- The `pop()` operation must ensure that the stack is not empty before it decrements `top_index`. There are two versions of `pop()`, the only difference being that one provides the top item before decrementing `top`.
- The `top()` operation returns the top item, throwing a stack exception in the event that the stack is empty.

```
Stack::Stack(): top_index(-1) {}


bool Stack::empty() const
{
    return top_index < 0;
}

void Stack::push(StackItemType new_item) throw(stack_exception)
{
    // if stack has no more room for another item
    if (top_index >= MAX_STACK-1)
        throw stack_overflow("stack_exception: stack full on push");
    else  {
        ++top_index;
        items[top_index] = new_item;
    }
}


void Stack::pop() throw(stack_exception)
{
    if ( empty()  )
        throw stack_exception("stack_exception: stack empty on pop");
    else
       --top_index;
}


void Stack::pop(StackItemType& top_item) throw(stack_exception)
{
    if ( empty()  )
        throw stack_exception("stack_exception: stack empty on pop");
    else {  // stack is not empty;
        top_item = items[top_index];
       --top_index;
    }
}


void Stack::top(StackItemType& top_item) const throw(stack_exception)
{
    if (empty())
        throw stack_exception("stack_exception: stack empty on top");
    else         // stack is not empty; retrieve top
        top_item = items[top_index];
}
```

5.2 Linked Implementation

A linked list implementation of a stack does not pre-allocate storage for the stack; it allocates nodes as needed. Each node will have a data item and a pointer to the next node. Because the linked list implementation hides the fact that it is using linked nodes, the node structure is declared within the private part of the class.

The only private data member of the `Stack` class is a pointer to the node at the top of the stack, which is the node at the front of the list. If the list is empty, the pointer is `NULL`. The private part of the `Stack` class would therefore be:

```
private:
   struct StackNode
   {
      StackItemType item;
      StackNode    *next;
   };
   StackNode *top_ptr;     // pointer to first node in the stack
};
```

Remarks.

- The constructor simply sets `top_ptr` to `NULL`, and the test for emptiness is whether or not `top_ptr` is `NULL`.
- The copy constructor has to traverse the linked list of the stack passed to it. It copies each node from the passed stack to the stack being constructed. It allocates a node, fills it, and attaches it to the preceding node
- The destructor repeatedly calls `pop()` to empty the list.
- The `push()` operation inserts a node at the front of the list.
- The two `pop()` operations remove the node at the front of the list, throwing an exception if the stack is empty. One provides this item in the parameter.
- Similarly, `top()` simply returns the item at the front of the list, throwing an exception if the stack is empty.
- In theory it is possible that the operating system will fail to provide additional memory to the object when it calls `new()` in both the copy constructor and the `push()` method, and this failure should be detected in a serious application.

```
// constructor
Stack::Stack() : top_ptr(NULL) { }

// copy constructor
Stack::Stack(const Stack& aStack) throw (stack_overflow)
{
    if (aStack.top_ptr == NULL)
        top_ptr = NULL;        // empty list
    else {
        // copy first node
        top_ptr        = new StackNode;
        if ( NULL == top_ptr )
            throw stack_overflow("copy constructor could not allocate node");
        top_ptr->item = aStack.top_ptr->item;

        // copy rest of list
        StackNode *newPtr = top_ptr ;   // new list pointer
        for (StackNode *origPtr = aStack.top_ptr->next; origPtr != NULL;
            origPtr = origPtr->next) {
            newPtr->next = new StackNode;
            if ( NULL == newPtr )
                throw stack_overflow("copy constructor could not allocate node");
            newPtr       = newPtr->next;
            newPtr->item = origPtr->item;
```

```
        }
        newPtr->next = NULL;
    }
}

// destructor, which empties the list by calling pop() repeatedly
Stack::~Stack()
{
    // pop until stack is empty
    while (!empty())
        pop();
}

bool Stack::empty() const
{
    return top_ptr == NULL;
}

void Stack::push(StackItemType newItem)  throw(stack_exception)
{
    // create a new node
    StackNode *newPtr = new StackNode;
    if ( NULL == newPtr )
        throw stack_overflow("push could not allocate node");
    // set data portion  of new node
    newPtr->item = newItem;

    // insert the new node
    newPtr->next = top_ptr;
    top_ptr = newPtr;
}

void Stack::pop() throw(stack_exception)
{
    if (empty())
        throw stack_exception("stack_exception: stack empty on pop");
    else { // stack is not empty;
        StackNode *temp = top_ptr;
        top_ptr = top_ptr->next;
        temp->next = NULL;
        delete temp;
    }
}

void Stack::pop(StackItemType& stackTop) throw(stack_exception)
{
     if (empty())
         throw stack_exception("stack_exception: stack empty on pop");
     else {  // stack is not empty; retrieve and delete top
         stackTop = top_ptr->item;
         StackNode *temp = top_ptr;
         top_ptr = top_ptr->next;

         // return deleted node to system
         temp->next = NULL;  // safeguard
         delete temp;
    }
}

void Stack::top(StackItemType& stackTop) const throw(stack_exception)
{
    if (empty())
        throw stack_exception("stack_exception: stack empty on top");
    else       // stack is not empty; retrieve top
        stackTop = top_ptr->item;
}
```

Performance Considerations A careful look at this implementation shows that the `push()`, `pop()`, `top()`, and `empty()` operations take a constant amount of time, independent of the size of the stack. In contrast, the destructor executes a number of instructions that is proportional to the number of items in the stack, and the copy constructor's running time is proportional to the size of the stack being copied.

5.3 The Standard Template Library Class stack[1]

5.3.1 About the Standard Template Library

The *Standard Template Library* (*STL*) contains, among other things, a collection of C++ class templates for the most commonly used abstract data types. The STL is part of all standard implementations of C++. A C++ class template is not a class, but a *template for a class*. A simple class template interface has the general form

```
template <typename T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

The T after the keyword `typename` in angle brackets is a placeholder. It means that all occurrences of the T in the interface will be replaced by the actual type that instantiates the template when it is used to define an object. In the code above, it is used in four different places.

The implementations of the class template member functions have the form

```
template <typename T>
void Container<T>::set ( T initial_data )
{
    mydata = new_data;
}

template <typename T>
T Container<T>::get() const
{
    return mydata);
}
```

5.3.2 The stack Template Class

To use the `stack` template class, you would write

```
#include <stack>
```

[1]The Standard Template Library was originally implemented as an independent library, but was eventually incorporated into the C++ standard.

at the top of the file using the `stack` template. The `stack` template class has the following member functions[2]:

```
bool empty() const;
size_type size() const;
T & top();
void pop();
void push( const T & x);
```

To declare an instance of a `stack` template in a program you would write `stack`<*actual type name*> *variable_name*, as in:

```
stack<int>    int_stack;    // stack contains ints
stack<string> string_stack; // stack contains strings
stack<char>   char_stack;   // stack contains chars.
```

Therefore, if you want to use the STL `stack` class instead of writing your own, you would declare a stack of the needed type, and make ordinary calls upon the member functions of the class, as in

```
stack<int> mystack;
int number;
while ( cin >> number ) {
    mystack.push(number);
}
```

The syntax for the function calls is the same whether it is a class template instantiation or not. Once it is instantiated, it looks like an ordinary class.

6 More Applications

6.1 Postfix Expressions

Remember that a postfix expression is either a simple operand, or two postfix expressions followed by a binary operator. An operand can be an object or a function call. The object can be constant or variable. Using letters of the alphabet as operands, an example of a postfix expression is

```
ab+cd-/e*f+
```

which is equivalent to the infix expression

```
((a+b)/(c-d))*e +f
```

6.1.1 Evaluating Postfix Expressions

Postfix is often used in command line calculators. Given a *well-formed*[3] postfix expression, how can we use a stack to evaluate it without recursion? To start, let us assume that the only operators are left-associative, binary operators such as `+`, `-`, `*`, and `/`. The idea is that the stack will store the operands as we read the string from left to right. Each time we find an operator, we will apply it to the two topmost operands, and push the result back onto the stack. The algorithm follows. The word "token" below means any symbol found in the string, excluding whitespace characters (if they are present.)

[2] In C++98 this is the complete set of member functions. Two others were added in 2011

[3] A well-formed string is one that can be generated by the grammar, i.e. is in the correct form.

```
create an empty stack that holds the type of values being computed
for each token tok in the string {
    if tok is an operand
        stack.push(tok);
    else if tok is an operator {
         let op denote the operation that tok provides
         stack.pop(right_operand);
         stack.pop(left_operand);
         result = left_operand op right_operand;
         stack.push(result);
    }
}
```

We will turn the stack on its side with the top facing *to the right* to demonstrate the algorithm. We will let the input string consists of single-digit operands and use blanks to separate the tokens for clarity. Suppose the following is the input:

```
8 6 + 9 2 - / 5 * 7 +
```

Each row of the following table shows the result of reading a single token from the input string and applying the body of the for-loop to it. Since the string has 11 tokens, it will take 11 iterations of the loop to compute the value of the expression.

Stack (right end is top)	**Input Remaining**	**Action to Take**
(*empty*)	`8 6 + 9 2 - / 5 * 7 +`	push 8 on stack
`8`	`6 + 9 2 - / 5 * 7 +`	push 6 on stack
`8 6`	`+ 9 2 - / 5 * 7 +`	8+6 = 14; push on stack
`14`	`9 2 - / 5 * 7 +`	push 9 on stack
`14 9`	`2 - / 5 * 7 +`	push 2 on stack
`14 9 2`	`- / 5 * 7 +`	9 - 2 = 7; push 7 on stack
`14 7`	`/ 5 * 7 +`	14 / 7 = 2; push 2 on stack
`2`	`5 * 7 +`	push 5 on stack
`2 5`	`* 7 +`	2*5 = 10; push 10 on stack
`10`	`7 +`	push 7 on stack
`10 7`	`+`	10+7=17; push 17 on stack
`17`		end-of-input; answer = 17

It is not hard to refine this algorithm and convert it to C++. This is left as an exercise.

6.1.2 Converting Infix to Postfix

We can also use a stack to convert infix expressions to postfix[4]. This time, though, the stack will store the *operators*, not the operands. Let us assume that the input string is well-formed and can contain any of the operators `+`, `-`, `*`, and `/` as well as parentheses, and that the operands are valid tokens, such as numeric literals and variables. We will use the term *simple operand* to refer to things like numeric literals and variables, and *compound operand* to refer to an operand made up of at least one operator or enclosed in parentheses, such as `(a)`.

[4]This is what most modern compilers do in order to create the machine instructions for these expressions

Some Observations

- The order of the operands is the same in postfix as it is in the infix expression. In other words, if `b` precedes `c` in the infix expression then b precedes `c` in the postfix as well. The operators may move relative to the operands, but not the operands with respect to each other. (This can be proved by induction on the size of the infix expression.)
- If, in an infix expression, the operand `b` precedes the operator `op`, then in the postfix expression, `b` will also precede `op`. Another way to say this is that the operators can only move to the right relative to the operands, never to the left.
- While the infix expression has parentheses, the postfix does not.
- For every '`(`' in the infix expression, there is a matching '`)`'. The sub-expression within these parentheses is a valid infix expression consisting of a left operand, an operator, and a right operand. These operands may be infix expressions that are not tokens, but regardless of whether they are simple or compound, the operands of all operators within the parentheses are contained within the parentheses also. This implies that the infix expression contained within a set of matching parentheses can be processed independently of what is outside those parentheses.

Since the order of the operands remains the same in the postfix as it was in the infix, the problem boils down to deciding where to put the operators in the postfix. These observations lead to a set of rules for making these decisions.

We initialize an empty output postfix expression before we begin. When we see a token that is a simple operand, we append a copy of it immediately to the right end of the output postfix expression. When we see an operator, we know that its right operand begins at the very next token, and so we scan across the infix string looking for the end of the right operand. When we find the end of the right operand, we append the operand to the output postfix and we put the operator after it.

The problem is complicated by the fact that the operators have varying precedence. Therefore, the end of the right operand will depend on the precedence of the operators that we find along the way. Thus, the expression

```
a+b*c
```

has a postfix of the form `abc*+` whereas the expression

```
a+b-c
```

has a postfix of the form `ab+c-`. In the first case, the right operand of the `+` is `b*c`, but in the second case, it is just `b`. When we read the `+`, we push it onto the stack, copy the `b` to the output postfix, and then we look at the next operator. If the precedence of the next operator is less than or equal to that of the `+`, we know that the `b` is the end of the right operand of `+` and we can pop the `+` and put it after the `b`. But if the operator has greater precedence, then the right operand is a compound operand whose end we have not yet found. For example, when reading the infix expression

```
a+b*c*d*e
```

after reading the `b`, we will find that every operator has greater precedence than `+`. The right operand of the `+` is the entire expression `b*c*d*e`, and the end of the right operand will be the `e`.

It will be easier to design the algorithm if we begin by assuming that the infix expressions do not have parentheses. In this case they are of the form

$$a_1 o_1 a_2 o_2 a_3 o_3 \cdots o_{n-1} a_n$$

where the o_j are operators and the a_j are simple operands. Assume that each time we read an operand, we append it to the right end of the output postfix string. Given a string like this, having just read an operator o_j from the string, how do we recognize the end of the right operand of o_j? We have found the end of its right operand if and only if

1. the token we have just read is an operator whose precedence is less than or equal to precedence(o_j), or
2. we reached the end of the input string.

In either case, the last operand that was appended to the right end of the output postfix string is the end of the right operand of o_j. We use the stack as a holding pen for the operators while we search for their right operands. To be precise, the invariant that will always be true is:

> If the stack is not empty, then the operator on top of the stack will have the property that its left operand is the most recently appended operand in the output postfix, but the end of the right operand has yet to be found.

Therefore, when we find the end of the right operand of the operator on the top of the stack, we pop that operator off of the stack and append it to the output postfix after its right operand. (We can argue by induction on the size of the stack that when we do this, the operator that is now at the top of the stack still satisfies the invariant.) This leads to the following algorithm:

```
While there are tokens to read from the infix string {
    Read a token from the infix string.
    If the token is an operand,
        append a copy to the postfix string.
    Otherwise, {
        if the token is an operator, then
            check if the stack is empty or not.
            If the stack is empty, {
                push the token onto the stack.
                // Notice that the invariant is now satisfied because this
                // is an operator whose right operand is not yet found
                // and its left operand was just appended to the postfix
            }
            Otherwise, {
                While the stack is not empty and
                    precedence(token) ≤ precedence(top)  {
                    pop the operator on top of the stack and
                    append it to the output postfix
                    // each operator that is popped has the property that we
                    // found the end of its right operand and its left operand
                    // is the most recently appended to the postfix
                }
                push the token onto the stack.
               (Notice that the invariant is now satisfied.)
            }
    }
}
While the stack is not empty {
```

```
        pop the operator on top of the stack and
        append it to the output postfix
    }
```

You can verify that the invariant will always be true at the end of each iteration of the outermost while-loop. When that loop ends, by Rule 2, the end of the right operand of every operator in the stack has been found, and these must be popped and appended to the postfix. It is because of the invariant that the topmost belongs to the left of the operators beneath it, because each time we pop an operator and append it, it forms a compound postfix expression that is the right operand of the operator now on top of the stack. (Think this through carefully.)

The following table demonstrates the above algorithm at work on the input string `a+b*c*d-e.`

Output Postfix	Stack	Input Infix Expression
		a+b*c*d-e
a		+b*c*d-e
a	+	b*c*d-e
ab	+	*c*d-e
ab	+*	c*d-e
abc	+*	*d-e
abc*	+	*d-e
abc*	+*	d-e
abc*d	+*	-e
abcd**	+	-e
abcd**+		-e
abcd**+	-	e
abcd**+e	-	
abcd**+e-		

This algorithm will work fine as long as there are no parentheses in the string. Adding parentheses is not much more work. Suppose that a pair of parentheses encloses a part of the infix expression, as in

```
a+b*(c+d-e*f)+g
```

Since the expression is assumed to be well-formed, the expression contained by the parentheses is also well-formed. Thus, we can treat this substring as a separate string that starts after the left parenthesis and that ends at the right parenthesis, and we can process it independently, treating the right parenthesis like the end-of-input character. Rules 1 and 2 apply to it as well, so we know that when we see the right parenthesis, we have found the end of the right operand of the operator on top of the stack.

We could, if we wanted, create a separate stack each time we found a left parenthesis, and process the infix contained within in that stack and then discard it. But this is inefficient. Instead we can use the single stack to process the parenthesized sub-expression, but putting a marker into the stack to denote the new "bottom". We can use the left parenthesis itself as a marker. Therefore, we put the left parenthesis into the stack when we find it. Our algorithm above will be modified so that the loop that compares the top of stack to the current token stops if, when it pops the stack, it sees the left parenthesis, since that implies we are within a parenthesized sub-expression. Additionally, when we see a right parenthesis, it will act like an end-of-string character for the current sub-expression, and so we will pop each operator off of the stack and append it to the output postfix until we see the left parenthesis, which will terminate that loop.

All of these considerations lead to the following algorithm:

```
create an empty stack to hold operands
for each token tok in the input infix string {
    switch (tok) {
        case operand:
            append tok to right end of postfix;
            break;
        case '(' :
            stack.push(tok);
            break;
        case ')':
            while ( stack.top() != '(' ) {
                append stack.top() to right end of postfix;
                stack.pop();
            }
            break;
        case operator:
            while ( !stack.empty() && stack.top() != '('
                    && precedence(tok) <= precedence(stack.top()) ) {
                append stack.top() to right end of postfix;
                stack.pop();
            }
            stack.push(operator);
            break;
    }
}
while ( !stack.empty() ) {
    append stack.top() to right end of postfix;
    stack.pop();
}
```

Example

The following table shows how the infix expression `a-(b+c*d)/e` would be processed by the above algorithm.

Output Postfix	Stack	Input Infix Expression
		a-(b+c*d)/e
a		-(b+c*d)/e
a	-	(b+c*d)/e
a	-(	b+c*d)/e
ab	-(	+c*d)/e
ab	-(+	c*d)/e
abc	-(+	*d)/e
abc	-(+*	d)/e
abcd	-(+*	)/e
abcd*	-(+	)/e
abcd*+	-(	)/e
abcd*+	-	/e
abcd*+	-/	e
abcd*+e	-/	
abcd*+e/	-	
abcd*+e/-		

Index