# Stack ADT(Draft)

Abdul Kadir Gorur

April 22, 2008

## 1 a stack ADT

A stack is a linear collection whose elements are added and removed from the same end. We say that a stack is processed in a last in, first out (LIFO) manner. That is, the last element to be put on a stack will be the first one that gets removed. The processing of a stack is shown in Figures 1 and 2. Usually a stack is depicted vertically, and we refer to the top of the stack as the end to which elements are added and from which they are removed.

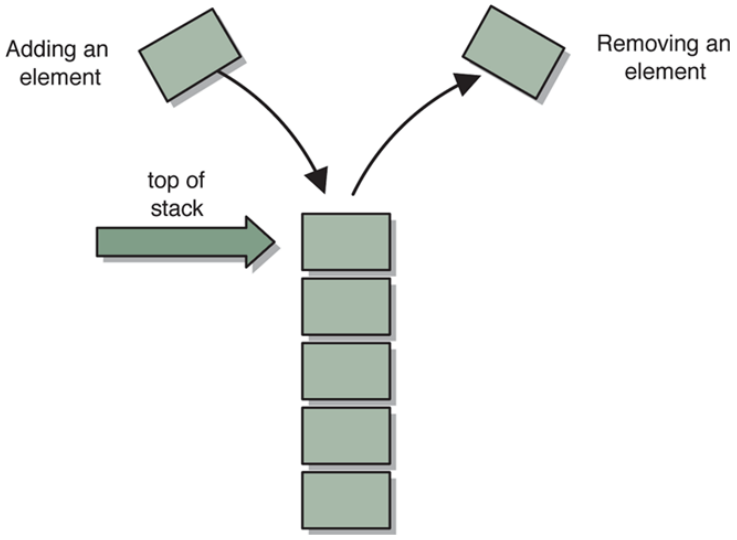


Figure 1: Stack

Recall from previous lectures that we define an abstract, data type (ADT) using a specific set of operations that establish the valid ways in which we can manage the elements stored in the data structure. We always want to use this concept to formally define the operations for a collection and work

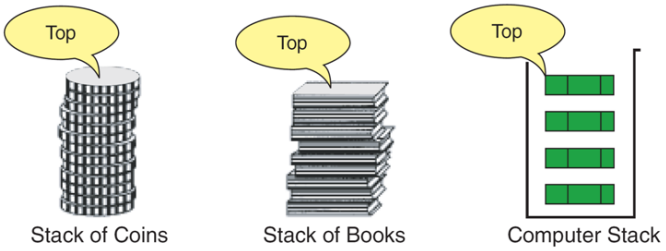


Figure 2: Stack

within the functionality it provides.

The operations for a stack ADT are listed in Figure 3. We say we push an element onto a stack and that we pop it off. We can also look at the top element of a stack, examining it or using it as needed, without actually removing it from the collection. There are also the general operations that allow us to determine if the stack is empty or, more specifically, how many elements it contains.

| Operation | Description |
|---|---|
| push | Adds an element to the top of the stack. |
| pop | Removes an element from the top of the stack. |
| itemAtTop | Examines the element at the top of the stack. |
| isStackEmpty | Determines if the stack is empty. |
| isStackFull | Determines if the stack is full. If array based implementation |
| size | Determines the number of elements on the stack. (Optional) |

Figure 3: The operations on a stack

Note that none of the stack operations in Figure 3 allow us to reach down into the stack to remove or reorganize the elements in the stack. That is the nature of a stack. All activity occurs at one end. If we discover that, in order to solve a particular problem, we need to access the elements in the middle or bottom of the collection, then a stack may not be the appropriate data structure to use.

*Although nothing prevents us from designing a data structure that allows us to perform other operations, such as moving the item at the top of the stack to bottom, the result would not be a stack.*

Stacks are used quite frequently in the computing world. For example,

the undo operation in a word processor is usually implemented using a stack. As we make changes to a document (add data, delete data, make format changes, etc.), the word processor keeps track of each operation by pushing some representation of it onto a stack. If we choose to undo an operation, the word processing software pops the most recently performed operation off the stack and reverses it. If we choose to undo again (undoing the second-to-last operation we performed), another element is popped from the stack. In most word processors, many operations can be reversed in this manner.

# 2 Basic Stack Operations

There are three basic operations on stack ADT. Push, pop and itemAt top. Push is used to insert data into the stack. Pop removes data from a stack and returns the data to the calling module. itemAtTop returns the data at the top of the stack without deleting the data from the stack.

## 2.1 Push

Push adds an item at the top of the stack. After the push, the new item becomes the top. The only potential problem with this simple operation is that we must ensure that there is room for the new item. If there is not enough room, the stack is in an **overflow** state and the item can not be added. Figure 4 shows the stack push operation.

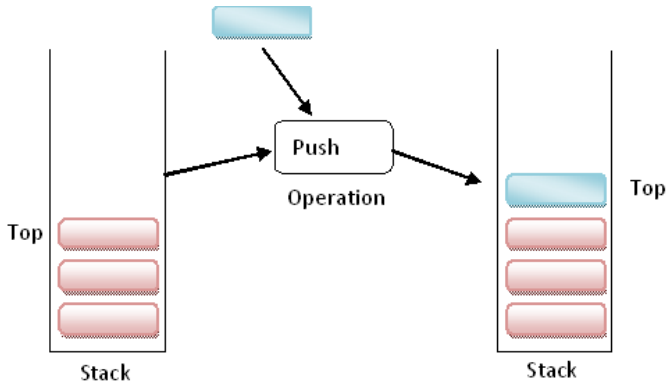


Figure 4: Push Stack Operation

## 2.2 Pop

When we pop a stack, we remove the item at the top of the stack and return it to the user. Because we have removed the top item, the next item, the next older item in the stack becomes the top. When the last item in the stack is deleted, the stack must be set to its empty state. If pop is called when the stack is empty, it is in an **underflow** state. The pop stack operation is shown in figure 5

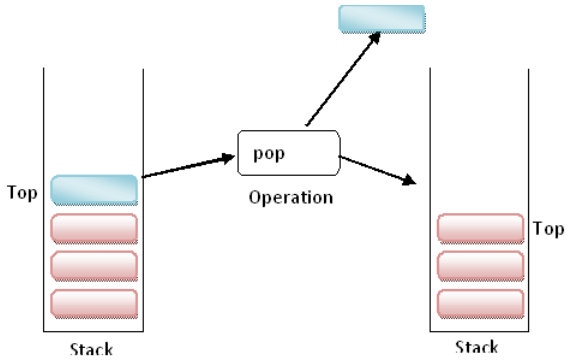


Figure 5: Pop Stack Operation

## 2.3 Stack Top(itemAtTop)

The third stack operation is stack top(or itemAtTop). This operation copies the item at the top of the stack:that is, it returns the data in the top element to the user but does not delete it. Stack top can also result in underflow if the stack is empty. The stack top operation is shown in figure 6

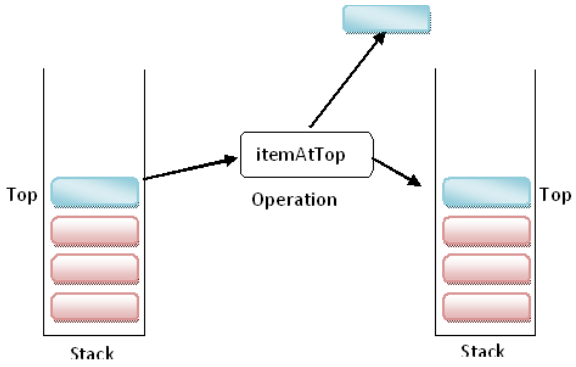


Figure 6: Stack Top Operation(itemAtTop)

# 3 Using Stacks

Consider the arithmetic statement in the assignment statement:

```
x = a * b + c
```

*Note: this is "infix" notation The operators are between the operands*
Compiler must generate machine instructions

```
LOAD  a
MULT  b
ADD  c
STORE  x
```

Most compilers convert an expression in infix notation to postfix(the operators are written after the operands). So a␣*␣b␣+␣c becomes a␣b␣*␣c␣+
**Advantage:** expressions can be written without parentheses

| Infix | Postfix | Prefix |
|---|---|---|
| A + B | A B + | + A B |
| A * B + C | A B * C + | + * A B C |
| A * (B + C) | A B C + * | * A + B C |
| A - (B - (C - D)) | A B C D— | -A-B-C D |
| A - B - C - D | A B-C-D- | —A B C D |

Table 1: Postfix and Prefix Examples

## 3.1 Evaluating Postfix Expressions

Traditionally, arithmetic expressions are written in an infix notation, meaning that the operator is placed between its operands in the form:

```
<operand> <operator> <operand>
```

such as in the expression
4 + 5
When evaluating an infix expression, we rely on **precedence** rules to determine the order of operator evaluation. For example, the expression
4 + 5*2

evaluates to 14 rather than 18 because of the precedence rule that says in the absence of parentheses, multiplication evaluates before addition.

In a postfix expression, the operator comes after its two operands. Therefore a postfix expression takes the form

```
<operand> <operand> <operator>
```

For example, the postfix expression

```
6␣9␣-
```

is equivalent to the infix expression

```
6␣-␣9
```

A postfix expression is generally easier to evaluate than an infix expression because precedence rules and parentheses do not have to be taken into account. The order of the values and operators in the expression are sufficient to determine the result. Programming language compilers and runtime environments often use postfix expressions in their internal calculations for this reason. The process of evaluating a postfix expression can be stated in one simple rule: Scanning from left to right, apply each operation to the two operands immediately preceding it. At the end we are left with the final value of the expression. Consider the infix expression we looked at earlier:
4 + 5 * 2
In postfix notation, this expression would be written
4 5 2 * +

### 3.1.1 Evaluating Postfix Expressions by Hand:underlining method

1. Scan the expression from left to right to find an operator.
2. Locate (”‘underline”’) the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:
2 3 4 + 5 6 − − ∗
2 $\underline{3\ 4\ +}$ 5 6 − − ∗

$2\ \mathbf{7}\ 5\ 6\ -\ -\ *$
$2\ 7\ \underline{5\ 6\ -}\ -\ *$
$2\ 7\ \mathbf{-1}\ -\ *$
$2\ \underline{7\ -1\ -}*$
$2\ \mathbf{8}\ *$
$\underline{2\ 8\ *}$
**16**

### 3.1.2 Evaluating Postfix Expressions by Stack

```
Initialize an empty stack
Repeat the following until the end of the expression is encountered
   Get the next token (variable(or value, operator) in the expression
   if Operand
       push onto stack
   else if Operator
       Pop 2 values from stack
       Apply operator to the two values
       Push resulting value back onto stack
   end if
end repeat
value of expression is the (only) number left in stack
```

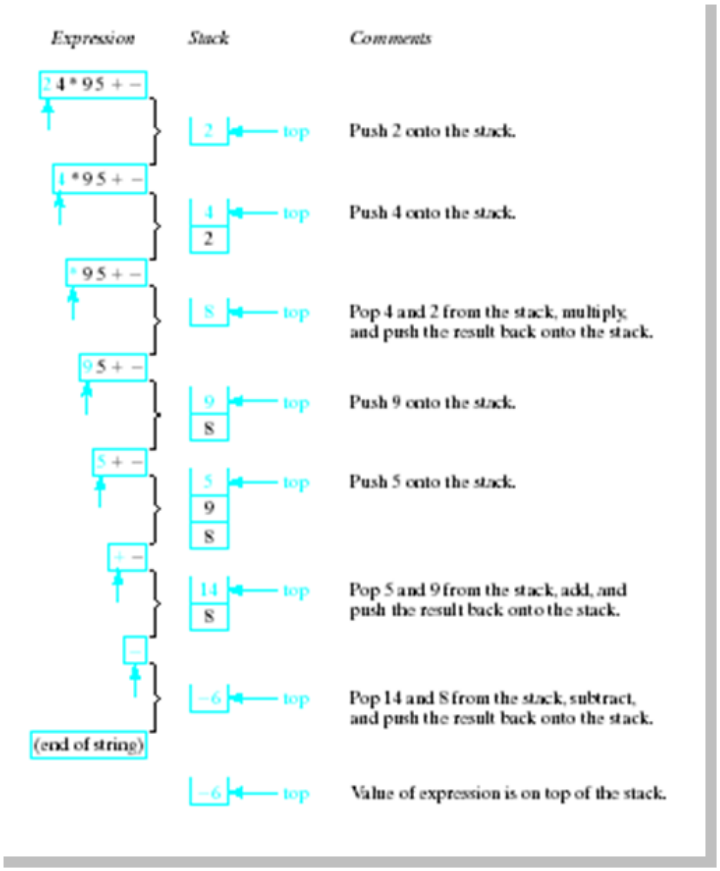


Figure 7: Evaluation of Postfix

## 3.2 Converting Infix to Postfix

By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.
2. Replace each right parenthesis by the corresponding operator.
3. Erase all left parentheses.

**Example1:**

Given infix expression: $A * B + C$ write the corresponding postfix expression.

Step1: fully parenthesize the expression:
$((A * B) + C)$

Step2:Replace each right parenthesis by the corresponding operator.
$((AB * C+$

Step3:Erase all left parentheses
$AB * C+$

**Example2:**

Given infix expression:$A*(B+C)$ write the corresponding postfix expression.
Step1: fully parenthesize the expression:
$(A * (B + C))$

Step2:Replace each right parenthesis by the corresponding operator.
$(A(BC + *$

Step3:Erase all left parentheses
$ABC + *$

### 3.2.1 Stack Algorithm

```
Initialize an empty stack of operators
While no error &&   !end of expression
   Get next input "token" from infix expression
```

```
   If token is "(" :
       push onto stack
    else if  ")" :
       pop and display stack elements until "(" occurs, do not display it
    else if operator
            if operator has higher priority than top of stack
                 push token onto stack
            else
                 pop and display top of stack
                 repeat comparison of token with top of stack
    else operand
         display it
end while
When end of infix reached, pop and display stack items until empty
```

### 3.2.2 Converting Infix to Postfix

The expression syntax that we started with (the usual one) is called an infix system, because the binary operator comes between its operands ( a + b ). The other system is called postfix because the operator comes after its operands ( ab+ ). To every infix expression there corresponds a postfix expression that has the same effect. The converse is not true because, as we have seen, there is an ambiguity. The infix expression a + b + c can be represented as either abc++ or ab+c+ in infix. The reason for the ambiguity is the lack of brackets in the infix expression. If the infix expression were '**fully bracketed**' there would be no ambiguity. Thus (a + b) + c pairs with ab+c+ and a + (b + c) pairs with abc++.

I now lets look at the problem of converting infix expressions to postfix expressions. First simplify the problem by only considering '**fully bracketed**' infix expressions. We will then look at the more difficult problem of handling general infix expressions.

The 'fully bracketed' version of an expression like a + b*c/d is (a + ((b*c)/d )). Note at once that there is a very serious problem here. The above expression should not be bracketed as ((a + b)*(c/d )) – that's something entirely different. This is where we meet the concept of operator precedence. You do multiplications before additions, and so on. The main reason for restricting to fully bracketed expressions is to avoid this problem. Now lets see the syntax for Fully Bracketed Infix Expressions (fbie's).

#### FBIE Grammar

The Grammar for Fully Bracketed Infix Expressions is as follows:

The alphabet is A = V B P where
V = a, b, c,..., z, B =+,-,*,/ and P =(,)
The abstract alphabet is = expr , var , op
The initial symbol is expr .
The productions are
expr ( expr op expr )
expr var
var a— b— c—...— z; and
op + — - —*—/

The following are examples of fbie. You should check that they are derivable from the above grammar.

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

### 3.2.3 a conversion algorithm of a fbie to postfix

A conversion algorithm which will take a fbie and convert it into the corresponding postfix expression.

```
algorithm fbie-to-postfix(s,n)
// convert the fbie s of length n into postfix.
begin
  let ans be empty string
  initialise the stack
  for i = 1 to n begin
    if isvar(si) then append si to ans
    if isop(si) then push(si)
    if si = ( then ignore
    if si =) then pop stack and append result to ans
  end
  return ans as result
end.
```

Let's work through an example. Suppose we want to translate

```
s = (((a + b)*(e - f )) + g)
```

The algorithm gives the trace shown in figure 8 The result is ab+ef-*g+. Note that the stack is empty at the end. Logically, it must be. If it is not

| item | action | ans | stack |
|---|---|---|---|
| 1 ( | ignore | empty | empty |
| 2 ( | ignore | empty | empty |
| 3 ( | ignore | empty | empty |
| 4 a | append | a | empty |
| 5 + | push | a | + |
| 6 b | append | ab | + |
| 7 ) | pop & append | ab+ | empty |
| 8 * | push | ab+ | * |
| 9 ( | ignore | ab+ | * |
| 10 e | append | ab+e | * |
| 11 - | push | ab+e | *- |
| 12 f | append | ab+ef | *- |
| 13 ) | pop & append | ab+ef- | * |
| 14 ) | pop & append | ab+ef-* | empty |
| 15 + | push | ab+ef-* | + |
| 16 g | append | ab+ef-*g | + |
| 17 ) | pop & append | ab+ef-*g+ | empty |

Figure 8: Translating the fbie s = (((a + b)*(e - f )) + g)

then s was not a fbie. Again note that the action of the routine is to put var symbols into the answer as they come, but to put op symbols onto the 'pending' stack until we know what they are to be applied to. The occurence of a right bracket tells us that the most recent pending operation is to be activated, so we take it off the stack and put it into the answer.

## 3.3 Symbol Balancing

A useful tool for checking your code is to see if the (), , and [] symbols balance properly.
For example the sequence $\cdots[\cdots(\cdots)\cdots]\cdots$ is legal but the sequence $\cdots[\cdots(\cdots]\cdots)\cdots$ is not. The presence of one misplaced symbol can result in hundreds of worthless compiler diagnostic errors!

### 3.3.1 Symbol Balancing Algorithm

```
Make an empty stack.
Repeat Read characters until end of file (EOF).
  If the character is an opening symbol ([{,
            push it onto the stack.
  If it is a closing symbol )]}, then
      if the stack is empty report an error.
  else
      pop the stack.
  If the symbol popped is not the corresponding opening symbol,
      report an error.
End Reat
At EOF, if the stack is not empty report an error
```

```
#include<stack.h> //character stack
#define isOpen(a)(a=='(' || a=='[' || a=='{')
#define isClosed(a)(a==')' || a==']' || a=='}')
int isBalanced(char *str);
int main()
{
   char *str="[{})";
   int stat=isBalanced(str);
   if(stat==1)
      printf("\nmatching!\n");
   else
      printf("\nnot matching!\n");
```

```
   return 0;
}

int isBalanced(char *str)
{
   int i;
   STACK s;
   initStack(&s);
   i=0;
   while(str[i]!='\0'){
    if( isOpen(str[i]) )
       push(&s,str[i]);
    else if( isClosed( str[i] ) ){
      if(isEmpty(&s))
          return 0;
      if ( str[i]=='}' && stackTop(&s)=='{' )
          pop(&s);
      if ( str[i]==']' && stackTop(&s)=='[' )
          pop(&s);
      if ( str[i]==')' && stackTop(&s)=='(' )
          pop(&s);
   }
   i++;
  }
  if(!isEmpty(&s))
      return 0;
  return 1;
}
```

# 4 Examples

Write a program to convert from decimal number into binary number using stack.

```
#include"stack.h"
int main()
{
   Stack s;
   int num;
   int digit;
   initStack(&s);
```

```
   printf("\nEner an integer:");
   scanf("%d",&num);
   while(num>0){
       digit=num%2;
       push(&s,digit);
       num=num/2;
   }//end while
   printf("\nBinary Number is\n");
   while(!stackEmpty(&s) ){
      printf("%d",pop(&s) );
   }
   printf("\n");
   return 0;
}
```

# 5 Excercise

1. Implement Stack ADT using List ADT described in previous chapter.