STACK

FROM chapter 3 of

Mark Allen Weiss, *Data structures and algorithm analysis*,

and

Adam Drozdek, *Data structures and algorithms in C++*

Stack

A *stack* is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.

Example: Consider a stack of trays in a cafeteria:

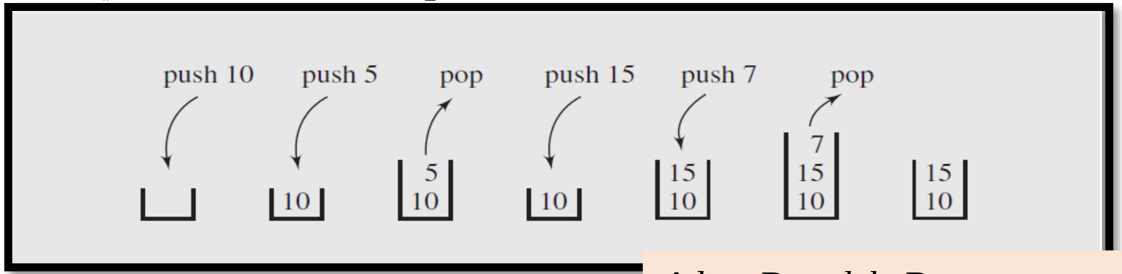
- new trays are put on the top of the stack and taken off the top.
- The last tray put on the stack is the first tray removed from the stack.

A stack is called ***LIFO structure: last in/first out.***

Unlike queue, in stack both ends are not used:

Stack Opeations

- STACK Operations
  - A tray can be taken only if there is at least one tray on the stack, and
  - a tray can be added to the stack only if there is enough room;
- A stack is defined in terms of operations that change its status. The operations are as follows:
    - ■ *clear()*—Clear the stack.
    - ■ *isEmpty()*—Check to see if the stack is empty.
    - ■ *push(x)*—Put the element *x* on the top of the stack.
    - ■ *pop()*—Take the topmost element from the stack.
    - ■ *topItem()*—Return the topmost element in the stack without removing it.

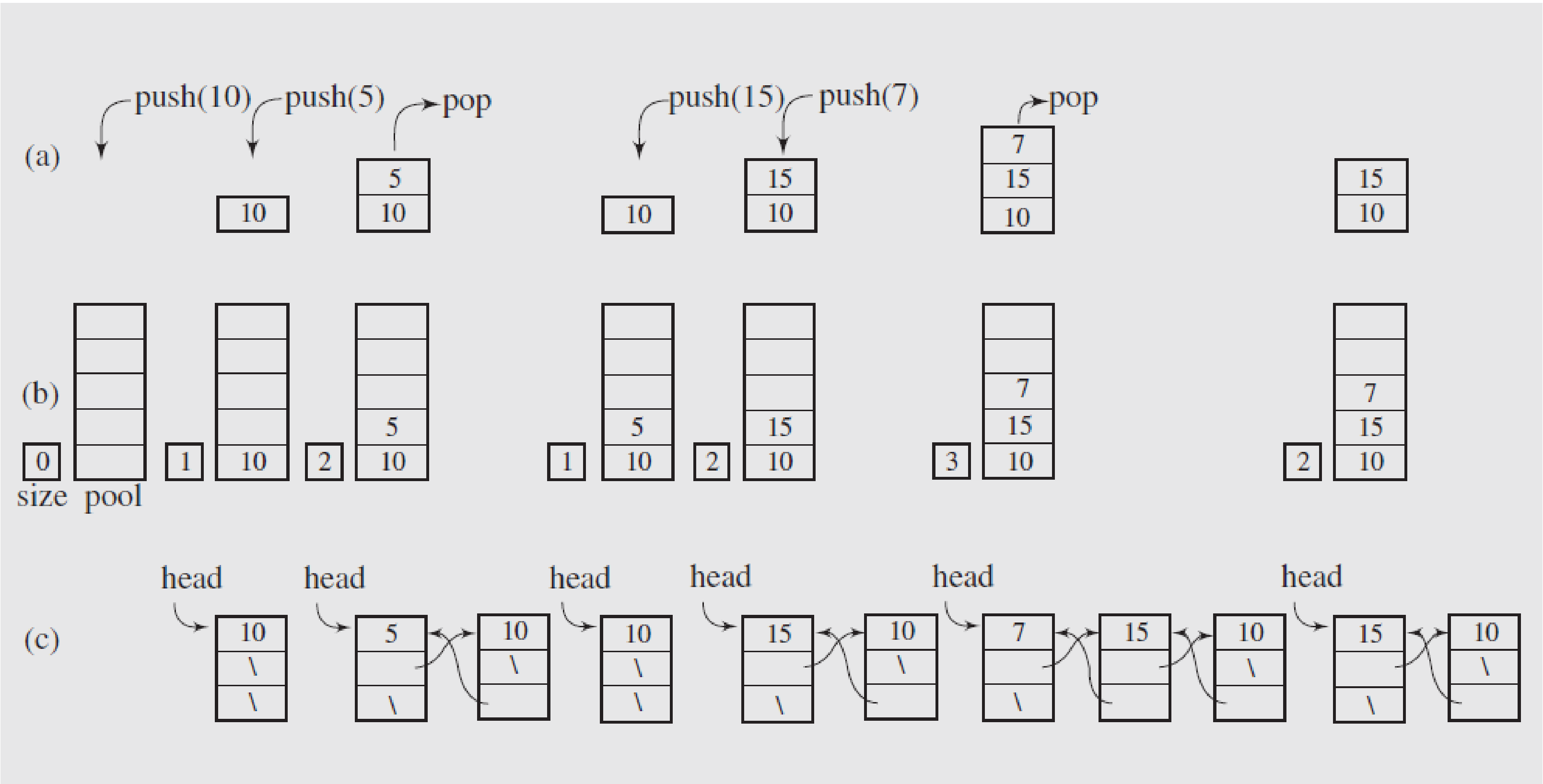


Adam Drozdek, Data structures and algorithms in C++

Implmentation of Stack

- Stack can be implemented using
  - Arrays
  - Linked List

FIGURE 4.6 A series of operations executed on (a) an abstract stack and the stack implemented (b) with a vector and (c) with a linked list.



STACK OPS

- It should come as no surprise that if we restrict the operations allowed on a list, those operations can be performed very quickly.

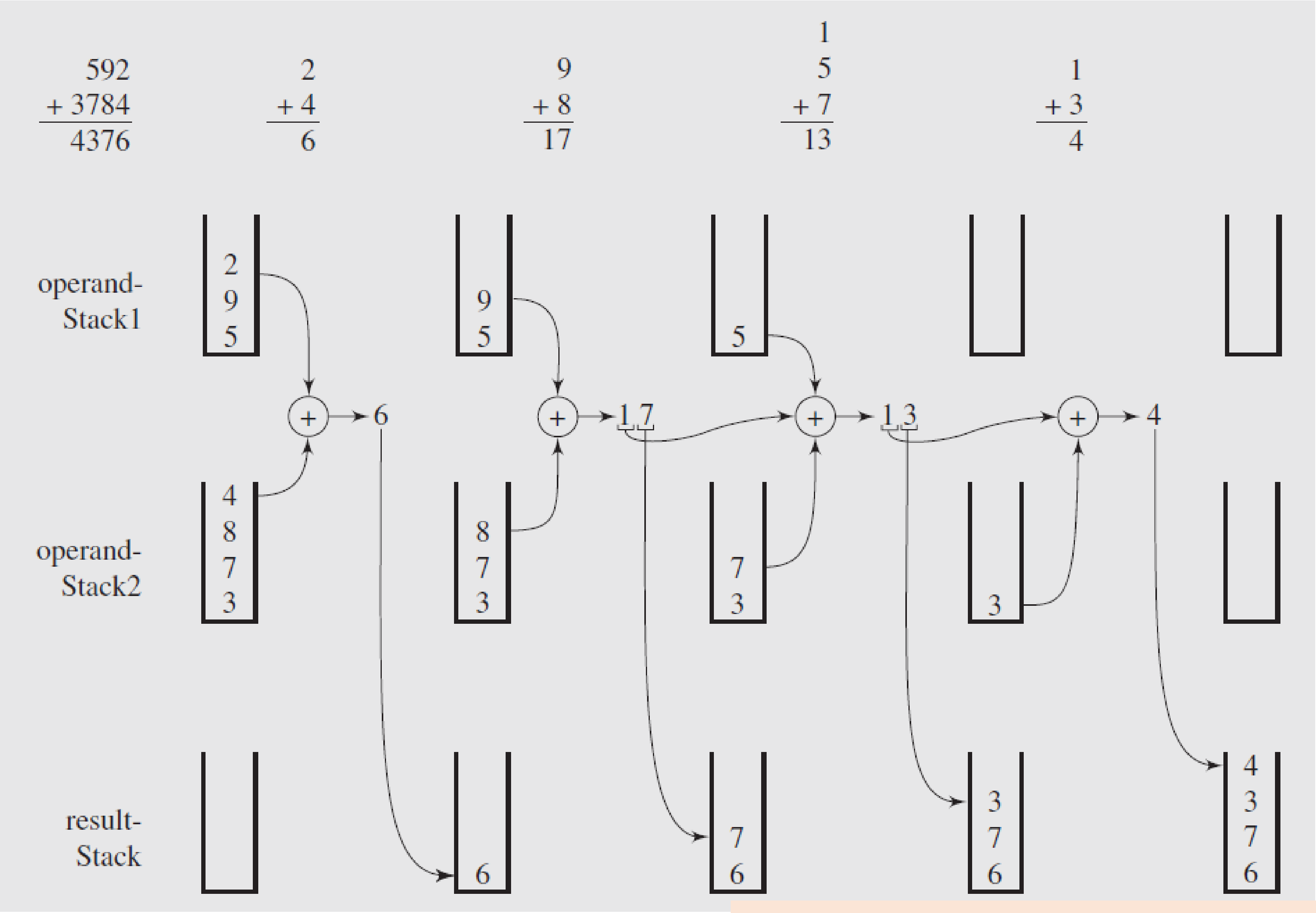
- The big surprise, however, is that the small number of operations left are so powerful and important.

Applications of Stack

The stack is very useful in situations when data have to be stored and then retrieved in reverse order.

- Undo-Redo in a Text Editor
- Adding Large Numbers
- Matching delimiters in a program.
- Evaluation of Fully Parenthesized Expression
- Converting Infix notation to PostFix
- System Stack
- Go Back and Forward in a Browser

Application 1 – Adding Large Numbers



Adam Drozdek, Data structures and algorithms in C++

Application 1 – Adding Large Numbers

```
addingLargeNumbers()
    read the numerals of the first number and store the numbers corresponding to
    them on one stack;
    read the numerals of the second number and store the numbers corresponding
    to them on another stack;
    carry = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to carry;
        push the unit part on the result stack;
        store carry in carry;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Application2 - Matching delimiters

- Matching delimiters in a program.
  - Delimiter matching is part of compiler: No program is considered correct if the delimiters are mismatched.
  - In C++ programs, we have the following delimiters: parentheses “(” and “)”, square brackets “[” and “]”, curly brackets “{” and “}”, and comment delimiters “/*” and “ */”.

```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
```

These examples are statements in which mismatching occurs:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

Matching delimiters

```
delimiterMatching(file)
  read character ch from file;
   while not end of file
      if ch is '(', '[', or '{'
            push(ch);
      else if ch is ')', ']', or '}'
            if ch and popped off delimiter do not match
               failure;
      else if ch is '/'
            read the next character;
            if this character is '*'
                 skip all characters until "*/" is found and report an error
                 if the end of file is reached before "*/" is encountered;
            else ch = the character read in;
                 continue; // go to the beginning of the loop;
   // else ignore other characters;
      read next character ch from file;
   if stack is empty
       success;
   else failure;
```

Processing string with Delimiter Matching Algorithm using Stack

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
((	(	w + y));
((	w	+y));
((	+	y));
((	y	));
(	)	);
empty	)	;
empty	;	

INFIX POSTFIX

- What should be the answer of
  - $4 + 5 + 6 * 2 =$
    - using simple calculator = 30
    - using C++ precedence rules or scientific calculator = 21

- A scientific calculator generally comes with parentheses, so we can always get the right answer by parenthesizing, but with a simple calculator we need to remember intermediate results.
- postfix
  - 4 5 6 2 * + +

INFIX POSTFIX

- Consider another expression
  - $4 * 2 + 5 + 6 * 3 =$
  - A typical evaluation sequence for this example
    - multiply 4 and 2 , saving this answer as A_1.
    - We then add 5 and A_1, saving the result in A_1.
    - We multiply 6 and 3, saving the answer in A_2, and
    - finish by adding A_1 and A_2, leaving the final answer in A_1.

- We can write this sequence of operations as follows:
    - $4\ 2 * 5 + 6\ 3 * +$
  - This notation is known as **postfix**, or **reverse Polish notation**, and is evaluated exactly as we have described above.

Application3- INFIX and POSTFIX

Infix	Postfix
a+b*c	abc*+
a*b+c*d	ab*cd*+
(a+b)*(c+d)/e-f	ab+cd+*e/f-
a/b-c+d*e-a*c	ab/c-de*+ac*-
a+b/c*(e+g)+h-f*i	abc/eg+*+h+fi*-

INFIX POSTFIX

- The easiest way to evaluate PostFix is to use a stack.
  - When a number is seen, it is pushed onto the stack;
  - when an operator is seen, the operator is applied to the two numbers that are popped from the stack, and
  - the result is pushed onto the stack

Algorithm to Evaluate Expressions in RPN

- while (not end of expression) {
  - Get next input symbol
  - if input symbol is an operand then
    - push it into the stack
  - else if it is an operator then
    - pop the operands from the stack
    - apply operator on operands
    - push the result back onto the stack
- }
- The top of stack is answer.

Algorithm to Evaluate Expressions in PostFix

6 5 2 3 + 8 * +3 + *

The first four symbols are placed on the stack. The resulting stack is

topOfStack →	3 2 5 6

Next, a '+' is read, so 3 and 2 are popped from the stack, and their sum, 5,

topOfStack →	5 5 6

Next, 8 is pushed.

topOfStack →	8 5 5 6

Now a '$*$' is seen, so 8 and 5 are popped, and $5 * 8 = 40$ is pushed.

topOfStack →	40 5 6

Next, a '+' is seen, so 40 and 5 are popped, and $5 + 40 = 45$ is pushed.

topOfStack →	45 6

Now, 3 is pushed.

topOfStack →	3 45 6

Mark Allen Weiss, Data structures and algorithm analysis

Algorithm to Evaluate Expressions in PostFix

6 5 2 3 + 8 ∗ +3 + ∗

Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$.

topOfStack → 48
6

Finally, a '∗' is seen and 48 and 6 are popped; the result, $6 * 48 = 288$, is pushed.

topOfStack → 288

6 * (5+((2+3)*8)+3))

Mark Allen Weiss, Data structures and algorithm analysis

Algorithm to Evaluate Expressions in RPN

(a+b)*(c+d) → ab+cd+*
Assuming a=2, b=6, c=3, d=-1

Input Symbol	Stack	Remarks
a	a	Push
b	a b	Push
+	8	Pop a and b from the stack, add, and push the result back
c	8 c	Push
d	8 c d	Push
+	8 2	Pop c and d from the stack, add, and push the result back
*	16	Pop 8 and 2 from the stack, multiply, and push the result back. Since this is end of the expression, hence it is the final result.

Algorithm for Infix to RPN Conversion

- Not only can a stack be used to evaluate a postfix expression but we can also use a stack to convert an expression in **infix** to postfix

- We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules.

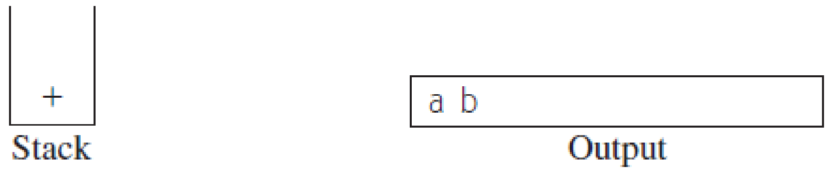
- We will further assume that the expression is legal.

Algorithm for Infix to PostFix Conversion

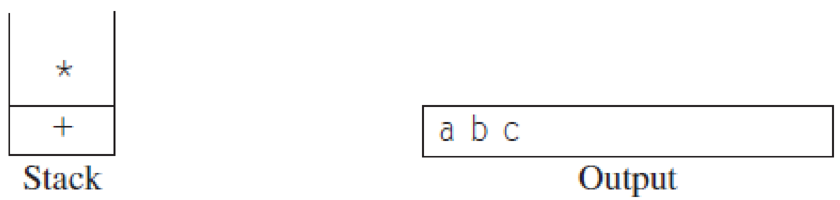
1. Initialize a stack of operators
2. While (! End of Input) {
   a. Get the next input symbol
   b. If input symbol is
      i. "(" — push
      ii. ")" — pop and display stack element until a left parenthesis is encountered, but do not display it.
      iii. An operator: — if (stack is empty or token has higher precedence than the element at Top Of Stack)
         push
         else
         pop and display the Top Stack element <u>and repeat step # iii.</u>
         Note: "(" is only pop when ")"encountered
      iii. An Operand: — Display
3. }
4. Until the stack is empty, pop and display

In FIX a + b * c + (d * e + f) * g

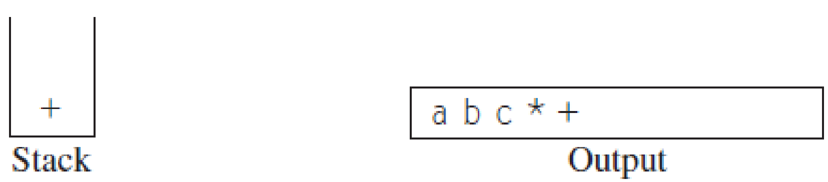
Then + is read and pushed onto the stack. Next b is read and passed through to the output. The state of affairs at this juncture is as follows:



Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have



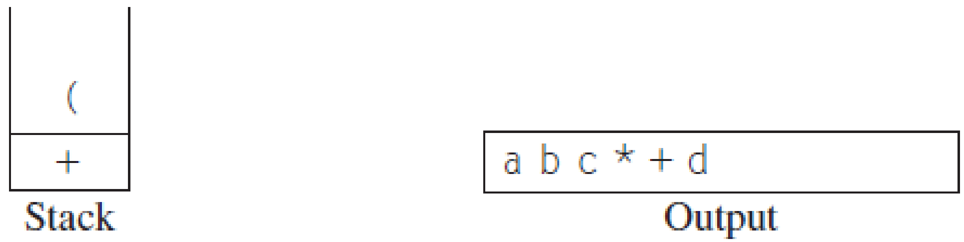
The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of *lower* but equal priority, on the stack; and then push the +.



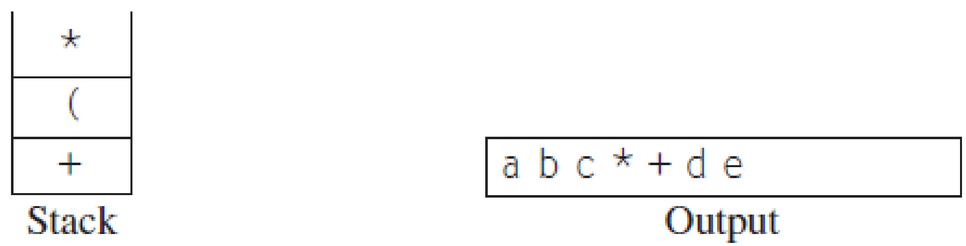
Mark Allen Weiss, Data structures and algorithm analysis

In FIX a + b * c + (d * e + f) * g

The next symbol read is a `(`. Being of highest precedence, this is placed on the stack. Then `d` is read and output.



We continue by reading a `*`. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, `e` is read and output.



Mark Allen Weiss, Data structures and algorithm analysis

In FIX a + b * c + (d * e + f) * g

The next symbol read is a +. We pop and output * and then push +. Then we read and output f.

Stack
+
(
+

Output: a b c * + d e * f

Now we read a), so the stack is emptied back to the (. We output a +.

Stack
+

Output: a b c * + d e * f +

We read a * next; it is pushed onto the stack. Then g is read and output.

Stack
*
+

Output: a b c * + d e * f + g

The input is now empty, so we pop and output symbols from the stack until it is empty.

Stack

Output: a b c * + d e * f + g * +

this conversion requires *O*(*N*) time and works in one pass through the input.

We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority

Mark Allen Weiss, Data structures and algorithm analysis

a+b*c/(d+e) → a b c * d e + / +

Input Symbol	Stack	Remarks
a		Operand – display – RPN → a
+	+	Push as stack is empty
b	+	Operand – display – RPN → a b
*	+ *	Push as * has higher precedence than +
c	+ *	Operand – display – RPN → a b c
/	+ /	Pop * and push / as * and / have the same precedence but / has higher precedence than + – RPN → a b c *
(	+ / (	Push
d	+ / (	Operand – display – RPN → a b c * d
+	+ / (+	Push as + has higher precedence than (
e	+ / (+	Operand – display – RPN → a b c * d e
)	+ /	Pop till "(" is found – RPN → a b c * d e +
End of input		Pop remaining symbols from the stack and display RPN → a b c * d e + / +

Application 4 - System Stack

Function Calls

- The problem here is that when a call is made to a new function
  - all the variables local to the calling routine need to be saved by the system,
    - since otherwise the new function will overwrite the memory used by the calling routine's variables.
  - Address of the next instruction in the calling program must be saved in order to resume the execution from the point of subprogram call.
- Clearly, all of this work can be done using a stack
  - That is exactly what happens in virtually every programming language that implements recursion.

Activation Records

Activation record is a data structure which keeps important information about a sub program.

The information stored in an activation record includes

- the address of the next instruction to be executed, and
- current value of all the local variables and parameters. i.e. the context of a subprogram is stored in the activation record.

When a subprogram is called, its activation record is created and pushed into the System stack.

When the subprogram ends

- its activation record is popped from the stack and destroyed-
- The control returns back to the calling function restoring its context

Activation records

```
int main()
{
    int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;

}
void A(){
    C();
    statement 5;
}
```

C()

Parameters & local variables:

Return Address: statement 5

A()

Parameters & local variables:

Return Address: statement 2

Main()

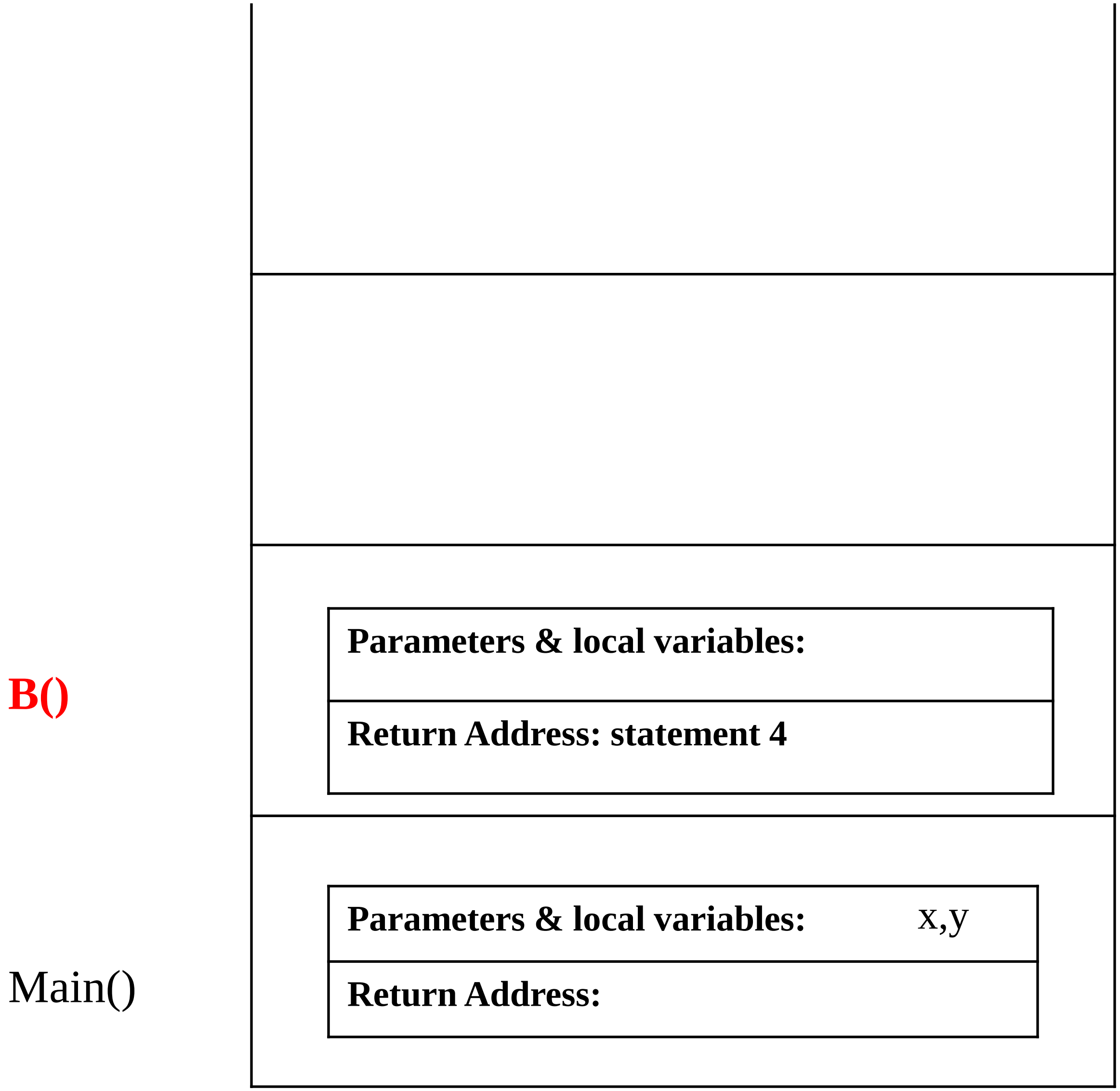
Parameters & local variables: x,y

Return Address:

Activation records

```
int main()
{
      int x,y;
      statement1;
      A();
      statement2;
      statement3;
      B();
      statement4;

}
void A(){
      C();
      statement 5;
}
```



Application 5 - Recursion

Practice Questions

- Convert PostFix Expression to Infix
- Evaluate Parenthesized InFix Expression using Stack
  - Example (a+(b/c))

- Convert Infix to PostFix such that it deals with + - * and division operators (assign +- equal precedence and * / equal precedence)

  Following the above idea
  - A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -.
  - Our algorithm does the right thing, because these operators associate from left to right.
  - This is not necessarily the case in general, since exponentiation associates right to left: 2^{2^3} = 28 = 256, not 43 = 64.
  - Add exponentiation to the repertoire of operators

Questions (Adams book)

- Reverse the element on Stack S
  - a. use additional stacks
  - b. one additional queue
  - c. using one additional stack and some additional nonarray variables

- Put the elements on the stack S in ascending order using one additional stack and some additional nonarray variables.

- Transfer elements from stack S1 to stack S2 so that the elements from S2 are in the same order as on S1
  - a. using one additional stack
  - b. using no additional stack but only some additional nonarray variables

Questions (Adams book)

- **Suggest an implementation of a stack to hold elements of two different types, such as structures and float numbers.**

- Using additional nonarray variables, order all elements on a queue using also
  - a. two additional queues
  - b. one additional queue

- Find if the elements in the stack form a palindrome or not. You can use one additional stack.