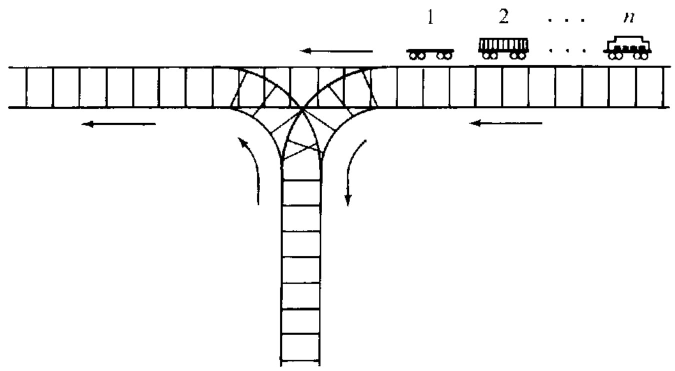
IV. Stacks

A. Introduction

1. Consider the 4 problems on pp. 170-1

(1) Model the discard pile in a card game

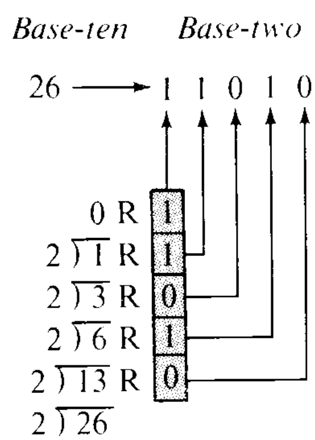
(2) Model a railroad switching yard



(3) Parentheses checker

(4) Calculate and display base-two representation

$26 = ????????_2$



Remainders are generated in right-to-left order. We need to "stack" them up, then print them out from top to bottom.

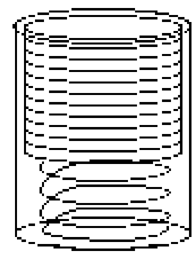
Need a "last-discarded-first-removed," "last-pushed-onto-first-removed," "last-stored-first-removed, " "last-generated-first-displayed" structured data type.

In summary ... a **LIFO (last-in-first-out)** structure.

2. Definition of a stack as an ADT (abstract data type):

A stack is **an ordered collection of data items in which access is possible only at one end, called the *top* of the stack. Its basic operations are:**

1. Construct a stack (usually empty)
2. Check if stack is empty
3. Push: Add an element at the top of the stack
4. Top: Retrieve the top element of the stack
5. Pop: Remove the top element of the stack



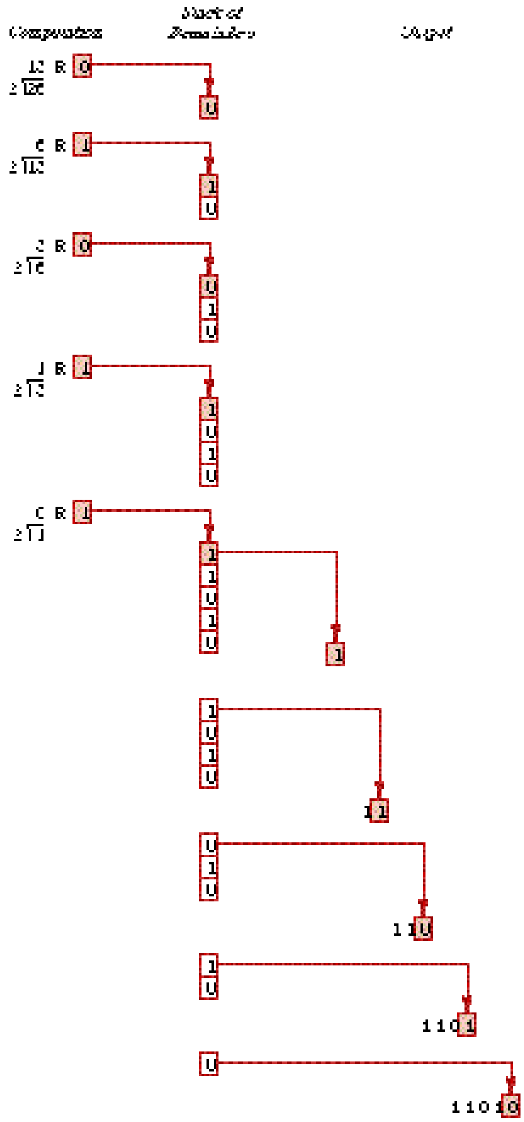
The terminology comes from a spring-loaded stack of plates in a cafeteria:
• Adding a plate pushed those below it are pushed down in the stack
• When a plate is removed from the stack, those below it pop up one position.

3. If we had a stack class we could use it to easily develop a short program for the base-conversion problem.

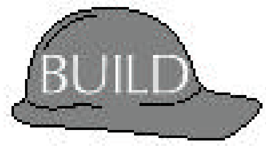
BASE-CONVERSION ALGORITHM (See p. 171-2)

/* This algorithm displays the base-2 representation of a base-10 number.
Receive: a positive integer *number*.
Output: the base-two representation of *number*.
--*/

1. Create an empty stack to hold the remainders.
2. While *number* ≠ 0:
   a. Calculate the *remainder* that results when *number* is divided by 2.
   b. Push *remainder* onto the stack of remainders.
   c. Replace *number* by the integer quotient of *number* divided by 2.
3. While the stack of remainders is not empty:
   a. Retrieve and remove the *remainder* from the top of the stack of remainders.
   b. Display *remainder*.


Computation
Stack of Remainders
1 1 0 1 0

```
/* Program that uses a stack to convert the base-ten
 * representation of a positive integer to base two.
 *
 * Input:  A positive integer
 * Output: Base-two representation of the number
 *******************************************************/

#include "Stack.h"           // our own -- <stack> for STL version
#include <iostream>
using namespace std;

int main()
{
  unsigned number,          // the number to be converted
           remainder;       // remainder when number is divided by 2
  Stack stackOfRemainders; // stack of remainders
  char response;            // user response

  do
  {
    cout << "Enter positive integer to convert: ";
    cin >> number;

    while (number != 0)
    {
      remainder = number % 2;
      stackOfRemainders.push(remainder);
      number /= 2;
    }

    cout << "Base-two representation: ";
    while (!stackOfRemainders.empty() )
    {
      remainder = stackOfRemainders.top();
      stackOfRemainders.pop();
      cout << remainder;
    }

    cout << endl;
    cout << "\nMore (Y or N)? ";
    cin >> response;
  }
  while (response == 'Y' || response == 'y');
}
```



B. Building a `Stack` Class

Two steps:

1. Design the class; and
2. Implement the class.

1. Designing a `Stack` Class

Designing a class consists of identifying those operations that are needed to manipulate the "real-world" object being modeled by the class. Time invested in this design phase will pay off, because it results in a well-planned class that is easy to use.

Note: The operations are described independently of how the class will be implemented. At this point, we have no idea what data members will be available, so the operations must be described in some manner that does not depend on any particular representation of the object.

The resulting specification then constitutes the "blueprint" for building the class.

From definition of stack as ADT, we must have (at least) the following operations:

- *Construction*: Initializes an empty stack.)
- *Empty* operation: Examines a stack and return false or true depending on whether the stack contains any values:
- *Push* operation: Modifies a stack by adding a value at the top of the stack:
- *Top* operation: Retrieves the value at the top of the stack:
- *Pop* operation: Modifies a stack by removing the value at the top of the stack:

To help with debugging, add early on:

- *Output*: Displays all the elements stored in the stack.

2. Implementing a `Stack` Class

Two steps:

1. Define data members.
2. Define the operations

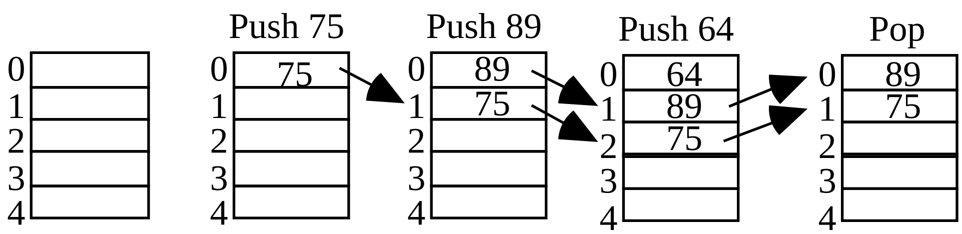
a. Selecting Data Members.

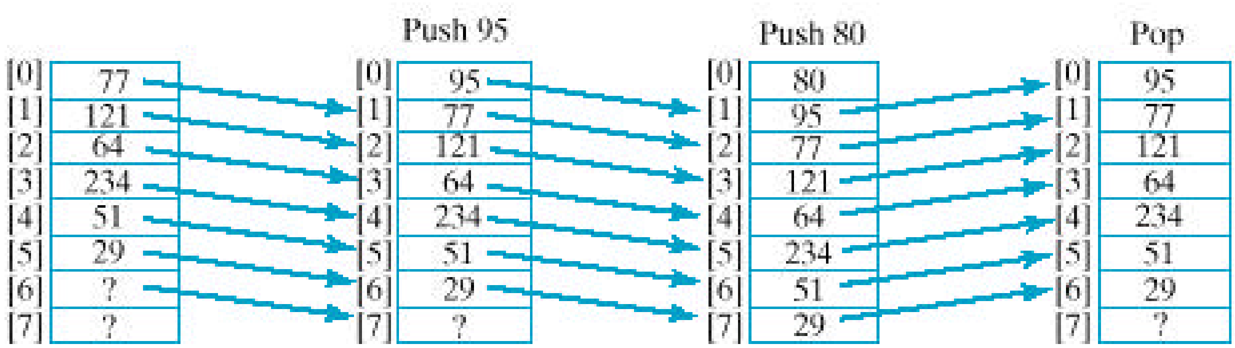
A stack must store a collection of values, so we begin by considering what kind of storage structure(s) to use.

Attempt #1:

Use an array with the top of the stack at position 0.

e.g., Push 75, Push 89, Push 64, Pop

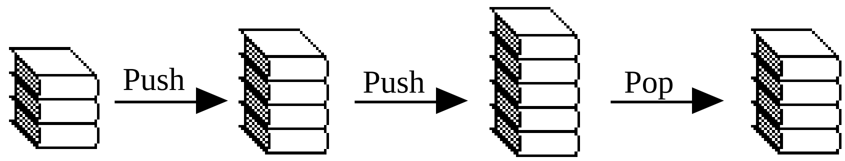




+ features: This models the operation of the stack of plates.
– features: Not efficient to shift the array elements up and down in the array.

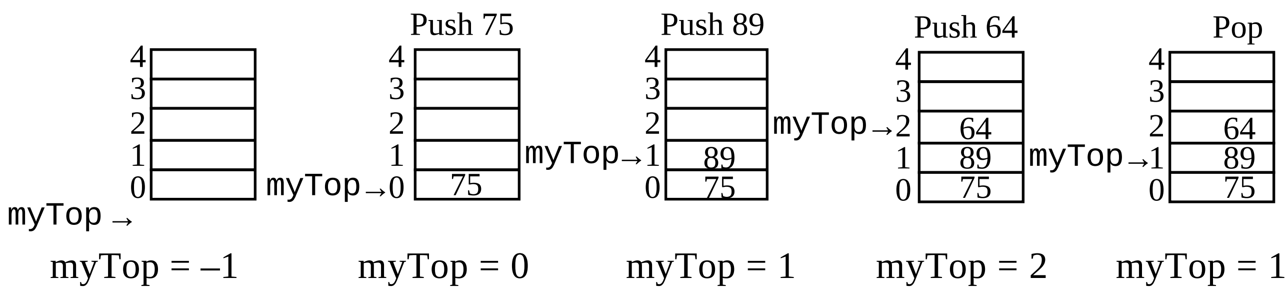
Attempt #2 — A Better Approach

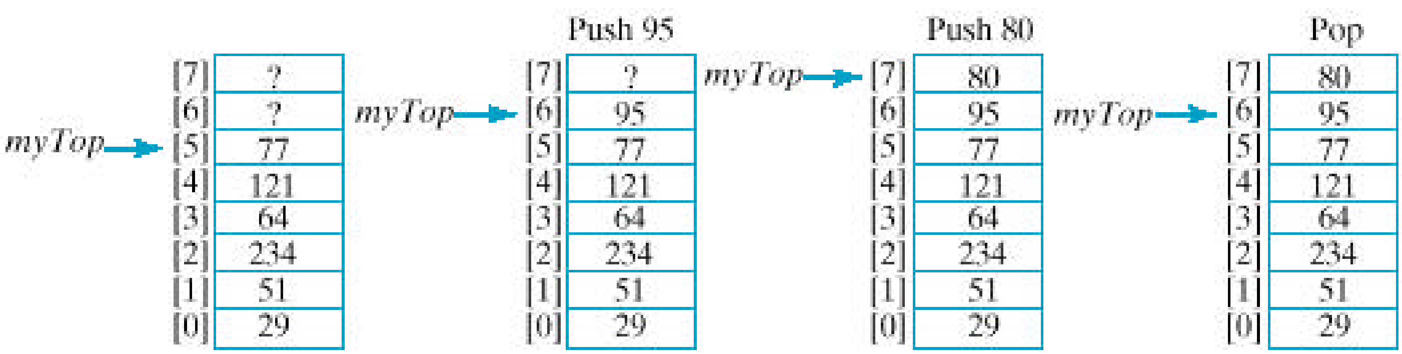
Instead of modeling the stack of plates, model a stack of books.



Keep the bottom of stack at position 0 and maintain a "pointer" `myTop` to the top of the stack.

e.g., Push 75, Push 89, Push 64, Pop





Note: No moving of array elements.

So, we can begin the declaration of our class by selecting data members:

- Provide an **array** data member to hold the stack elements.
- Provide a **constant** data member to refer to the **capacity of the array**.
- Provide an **integer** data member to indicate the ***top* of the stack**.

Problems: We need an array declaration of the form

```
ArrayElementType myArray[ARRAYCAPACITY];
```

— What type should be used?

Solution (for now): Use the **`typedef`** mechanism:

```
typedef int StackElement;
// put this before the class declaration
```

— What about the capacity?

```
const int STACK_CAPACITY = 128;
// put this before the class declaration
```

— Then declare the array as a data member in the private section:

```
StackElement myArray[STACK_CAPACITY];
```

Notes:

1. The `typedef` makes `StackElement` a **synonym** for `int`. Putting it outside the class makes it accessible throughout the class and in any file that `#include`s `Stack.h`. If in the future we want a stack of reals, or characters, or . . ., we need only change the `typedef`:

   ```
   typedef double StackElementType;
   ```
   or
   ```
   typedef char StackElementType;
   ```
   or . . .

   When the class library is recompiled, the type of the array's elements will be `double` or `char` or . . .

2. A more modern alternative that doesn't require using (and changing a `typedef` is to use the **template** mechanism to build a `Stack` class whose element type is left unspecified. The element type is then passed as a special kind of parameter at compile time. We'll describe this soon. This is the approach used in the **Standard Template Library (STL).**

3. Putting the `typedef` and declaration of `STACK_CAPACITY` ahead of the class declaration makes these declarations easy to find when they need changing.

4. If the type `StackElement` or the constant `STACK_CAPACITY` were defined as public members inside the class declaration, they could be accessed outside the class but would require qualification:

```
Stack::STACK_CAPACITY
Stack::StackElement
```

5. If we were to make the constant `STACK_CAPACITY` a class member we would probably make it a **static** data member:

```
static const int STACK_CAPACITY = 128;
```

   This makes it a property of the class usable by all class objects, but they do not have their own copies of `STACK_CAPACITY`.

So, we can begin writing `Stack.h`:

`Stack.h`

```
/* Stack.h provides a Stack class.
 *
 * Basic operations:
 *   Constructor:  Constructs an empty stack
 *   empty:   Checks if a stack is empty
 *   push:    Modifies a stack by adding a value at the top
 *   top:     Accesses the top stack value; leaves stack unchanged
 *   pop:     Modifies a stack by removing the value at the top
 *   display: Displays all the stack elements
 * Class Invariant:
 *   1. The stack elements (if any) are stored in positions
 *      0, 1, . . ., myTop of myArray.
 *   2. -1 <= myTop < STACK_CAPACITY
 -------------------------------------------------------------------*/

#ifndef STACK
#define STACK

const int STACK_CAPACITY = 128;
typedef int StackElement;

class Stack
{

/***** Function Members *****/
public:
   . . .

/***** Data Members *****/
private:
  StackElement myArray[STACK_CAPACITY];
  int myTop;

}; // end of class declaration
. . .

#endif
```

b. Function Members

- *Constructor* :

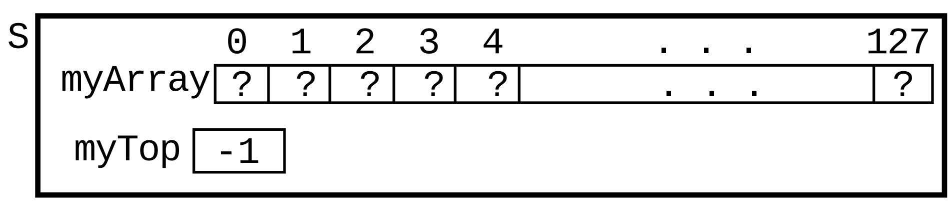
Simple enough to inline? **Yes**

```
class Stack
{
public:
/* --- Constructor ---

   Precondition:  A stack has been declared.
   Postcondition: The stack has been constructed as an
                    empty stack.
-------------------------------------------------------------*/
Stack();
   . . .
};// end of class declaration

inline Stack::Stack() {myTop = -1;}
```

A declaration

```
Stack S;
```

will construct S as follows:



- *empty:*

Receives: Stack containing it as a function member (implicitly)
Returns: True if stack is empty, false otherwise.
Member function? Yes
`const` function? (Shouldn't alter data members?) Yes
Simple enough to inline? Yes

```
class Stack
{
public:
   . . .
/* --- Is the Stack empty? ---
 * Receive: stack containing this function (implicitly)
 * Returns: true if the Stack containing this function is empty
 *          and false otherwise

**************************************************************/

bool empty() const;
   . . .
};// end of class declaration

inline bool Stack::empty() const
{ return (myTop == -1); }
```

Test driver:

```
#include <iostream>
using namespace std;
#include "Stack.h"
int main()
{
  Stack s;
  cout << boolalpha << "s empty? " << s.empty() << endl;
}
```

Output

```
s empty? true
```

- *push:*

Receives: Stack containing it as a function member (implicitly)
Value to be added to stack
Return: Modified Stack (implicitly)
Member function? Yes
`const` function? No
Simple enough to inline? Probably not

```
class Stack
{
public:
   . . .
/* --- Add a value to the stack ---
 *
 * Receive:   The Stack containing this function (implicitly)
 *            A value to be added to a Stack
 * Pass back: The Stack (implicitly), with value added at its
 *            top, provided there's space
 * Output:    "Stack full" message if no space for value
 ************************************************************/

void push(const StackElement & value);
   . . .

}; // end of class declaration
```

Definition In `Stack.cpp`

```
void Stack::push(const StackElement & value)
{
  if (myTop < STACK_CAPACITY - 1)  // Preserve stack invariant
  {
    ++myTop;
    myArray[myTop] = value;
  }                     // or simply, myArray[++myTop] = value;
  else
    cerr << "*** Stack is full -- can't add new value ***\n"
         << "Must increase value of STACK_CAPACITY in Stack.h\n";
}
```

Add at bottom of driver:

```
for (int i = 1; i <= 128; i++) s.push(i);
cout << "Stack should now be full\n";
s.push(129);
```

Output

```
s empty? true
Stack should now be full
*** Stack is full -- can't add new value ***
```

- *Output:*

  So we can test our operations.
  Receives: Stack containing it as a function member (implicitly)
  Output: Contents of Stack, from the top down.
  Member function? Yes
  `const` function? (Shouldn't alter data members?) Yes
  Simple enough to inline? No

  Prototype:

```
/* --- Display values stored in the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 *          The ostream out
 * Output:  The Stack's contents, from top down, to out
 **************************************************************/

void display(ostream & out) const;
```

  Definition in Stack.cpp:

```
void Stack::display(ostream & out) const
{
  for (int i = myTop; i >= 0; i--)
    out << myArray[i] << endl;
}
```

  Modify driver:

```
/*
for (int i = 1; i <= 128; i++) s.push(i);
  cout << "Stack should now be full\n";
s.push(129);
*/
for (int i = 1; i <= 4; i++) s.push(2*i);
  cout << "Stack contents:\n";
s.display(cout);
cout << "s empty? " << s.empty() << endl;
```

  Output

```
s empty? true
Stack contents:
8
6
4
2
s empty? false
```

- *top:*

  Member function? <u> Yes</u>
  `const` function? <u>Yes</u>
  Simple enough to inline? <u>Probably not</u>

  <u>Prototype</u>:

```
/* --- Return value at top of the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 * Return:  The value at the top of the Stack, if nonempty
 * Output:  "Stack empty" message if stack is empty
 ************************************************************/

StackElement top() const;
```

  <u>Definition (in `Stack.cpp`):</u>

```
StackElement Stack::top() const
{
  if (myTop >= 0)
    return (myArray[myTop]);
  cerr << "*** Stack is empty ***\n";
}
```

  <u>Add to driver at bottom</u>:

```
cout << "Top value: " << s.top() << endl;
```

  <u>Output</u>

```
Stack contents:
8
6
4
2
s empty? false
Top value: 8
```

- *pop:*

  Member function? Yes
  `const` function? No
  Simple enough to inline? Probably not

  Prototype:

```
/* --- Remove value at top of the stack ---
 *
 * Receive:   The Stack containing this function (implicitly)
 * Pass back: The Stack containing this function (implicitly)
 *             with its top value (if any) removed
 * Output:    "Stack-empty" message if stack is empty.
 **************************************************************/

void pop();
```

  Definition (in `Stack.cpp`):

```
void Stack::pop()
{
  if (myTop >= 0)    // Preserve stack invariant
    myTop--;
  else
    cerr << "*** Stack is empty -- can't remove a value ***\n";
}
```

  Add to driver at bottom:

```
while (!s.empty())
{
  cout << "Popping " << s.top() << endl;
  s.pop();
}
cout << "s empty? " << s.empty() << endl;
```

  Output

```
Stack contents:
8
6
4
2
s empty? false
Top value: 8
Popping 8
Popping 6
Popping 4
Popping 2
s empty? true
```

C. Two Applications of Stacks

Use of Stacks in Function Calls

Whenever a function begins execution (i.e., is activated), an *activation record* (or *stack frame*) is created to store the *current environment* for that function. Its contents include:

parameters
caller's state information (saved) (e.g., contents of registers, return address)
local variables
temporary storage

What kind of data structure should be used to store these when a function calls other functions and interrupts its own execution so that they can be recovered and the system reset when the function resumes execution?

Clearly need LIFO behavior. (Obviously necessary for recursive functions.)
So use a stack. Since it is manipulated at run-time, it is called the **run-time stack**.

What happens when a function is called:

(1) Push a copy of its activation record onto the run-time stack
(2) Copy its arguments into the parameter spaces
(3) Transfer control to the address of the function's body

So the top activation record in the run-time stack is always that of the function currently executing.

What happens when a function terminates?

(1) Pop activation record of terminated function from the run-time stack
(2) Use new top activiation record to restore the environment of the interrupted function and resume execution of it.

Examples:

```
. . .
int main()
{ . . .
   f2(...);
   f3(...);
}
void f1(...) {. . .}
void f2(...) {... f1(...); ...}
void f3(...) {... f2(...); ...}
```

```
int factorial(int n)
{ if (n < 2)
      return 1;
   else
      return n * factorial(n-1);
}
```

What happens to the run-time stack when the following statement executes?

```
int answer = factorial(4);
```

This pushing and popping of the run-time stack is the real overhead associated with function calls that inline functions avoid by replacing the function call with the body of the function.

Application to Reverse Polish Notation

1. What is RPN?

   A notation for arithmetic expressions in which operators are written after the operands. Expressions can be written without using parentheses.

   Developed by Polish logician, Jan Lukasiewics, in 1950's

Infix notation: operators written **between** the operands
Postfix " (**RPN**): operators written **after** the operands
Prefix " : operators written **before** the operands

Examples:

INFIX	RPN (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

2. Evaluating RPN Expressions

a. "By hand": Underlining technique:

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

```
Example:      2 3 4 + 5 6 - - *
           →  2 3 4 + 5 6 - - *  →  2 7 5 6 - - *
           →  2 7 5 6 - - *      →  2 7 -1 - *
           →  2 7 -1 - *  →  2 8 *  →  2 8 *  →  16
```

b. Algorithm — using a stack of operands

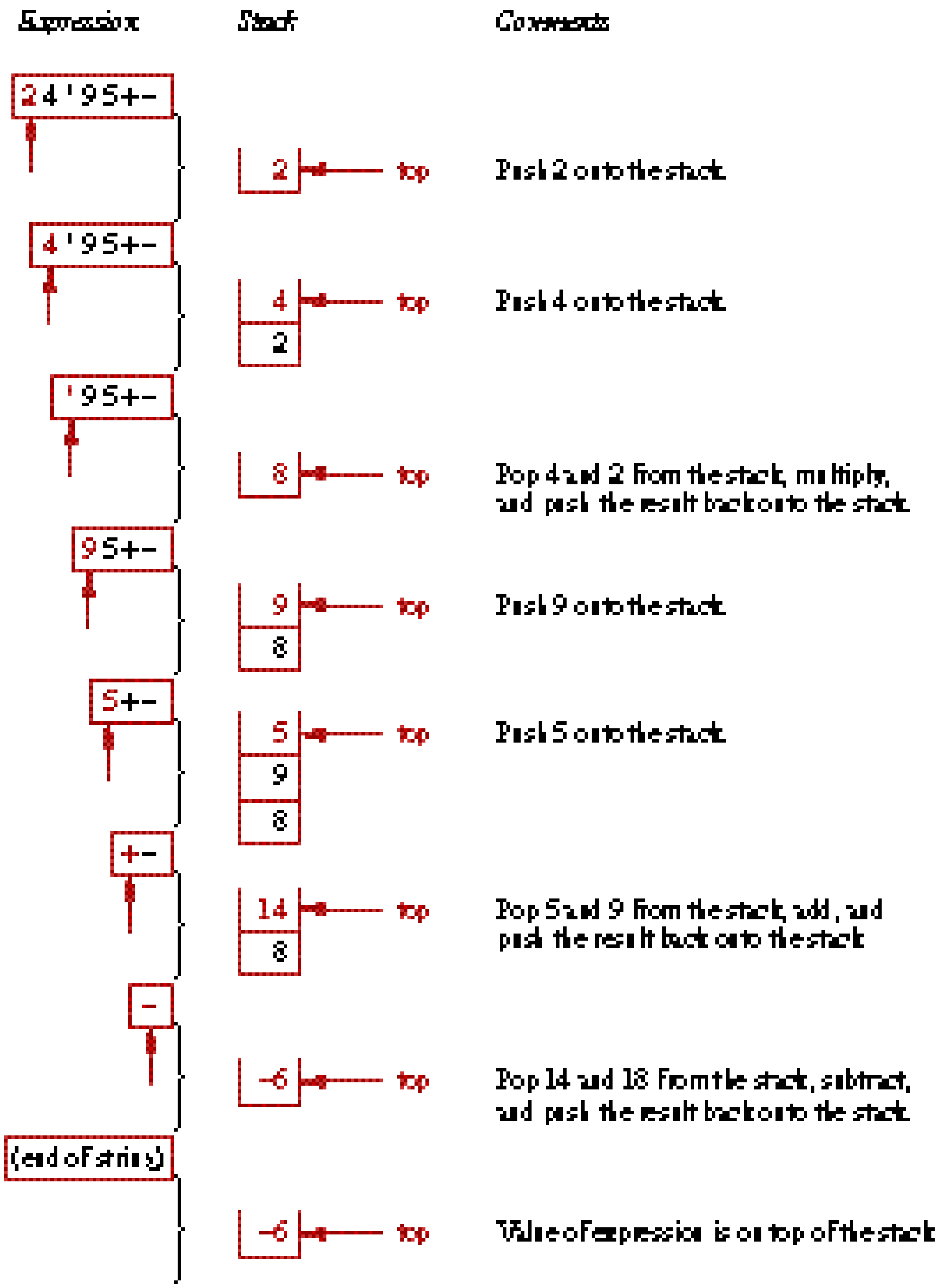
ALGORITHM TO EVALUATE RPN EXPRESSIONS

Receive: An RPN expression.
Return: A stack whose top element is the value of the RPN expression (unless an error occurred).
Note: Uses a stack to store operands.

---

1. Initialize an empty stack.
2. Repeat the following until the end of the expression is encountered:
   a. Get the next token (constant, variable, arithmetic operator) in the RPN expression.
   b. If the token is an operand, push it onto the stack. If it is an operator, then do the following:
      (i) Pop the top two values from the stack. (If the stack does not contain two items, an error due to a malformed RPN expression has occurred, and evaluation is terminated.)
      (ii) Apply the operator to these two values.
      (iii) Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack (and, in fact, must be the only value in the stack).

Example: See p. 172.



<u>To generate code, change (ii) and (iii) to:</u>

(ii') Generate code: LOAD $operand_1$
op $operand_2$
STORE TEMP#

(iii') Push TEMP# onto stack.

Example: Generate code for A B + C D + *

c. Unary minus causes problems — use different symbol

Example: `5 3 - -` → `5 3 - -` → `5 -3 -` → **8**

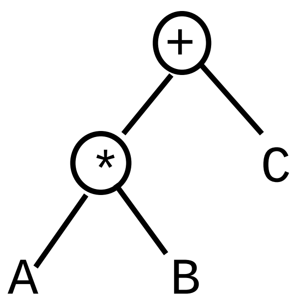
`5 3 - -` → `5 3 - -` → `2 -` → **-2**

We'll use a different symbol: `5 3 ~ -`   `5 3 - ~`

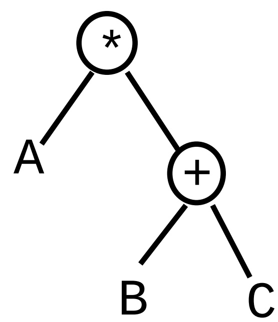
3. Converting from Infix to RPN

a. "By hand": Represent infix expression as an *expression tree***:**

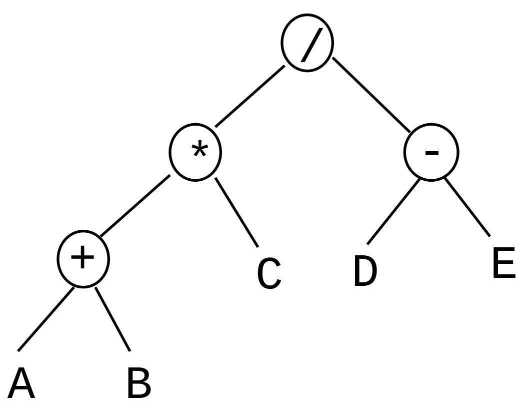
A * B + C



A * (B + C)



((A + B) * C) / (D - E)



Traverse the tree in *Left-Right-Parent* order to get **RPN**
Traverse tree in *Parent-Left-Right* order to get **Prefix**
Traverse tree in *Left-Parent-Right* order to get **Infix** [must insert ()'s]

b. 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.

Examples:

```
A * B + C  →  ((A * B) + C)

           →  ((A B * C +  →  A B * C +


A * (B + C) →  (A * (B + C) )

            →  (A ( B C + * →  A B C + *

((A + B) * C) / (D - E) → (((A + B) * C) / (D - E) )
                        → (((A B + C * (D E - /
                        →  A B + C * D E - /
```

c. Algorithm — using a stack of operators

ALGORITHM TO CONVERT AN INFIX EXPRESSION TO RPN

Accepts: An infix expression.
Output: The RPN expression.
Note: Uses a stack to store operators.

1. Initialize an empty stack of operators.

2. While no error has occurred and the end of the infix expression has not been reached:

   a. Get the next input *Token* (constant, variable, arithmetic operator, left parenthesis, right parenthesis) in the infix expression.

   b. If *Token* is

      (i) a left parenthesis: Push it onto the stack.

      (ii) a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display it. (It is an error if the stack becomes empty with no left parenthesis found.)

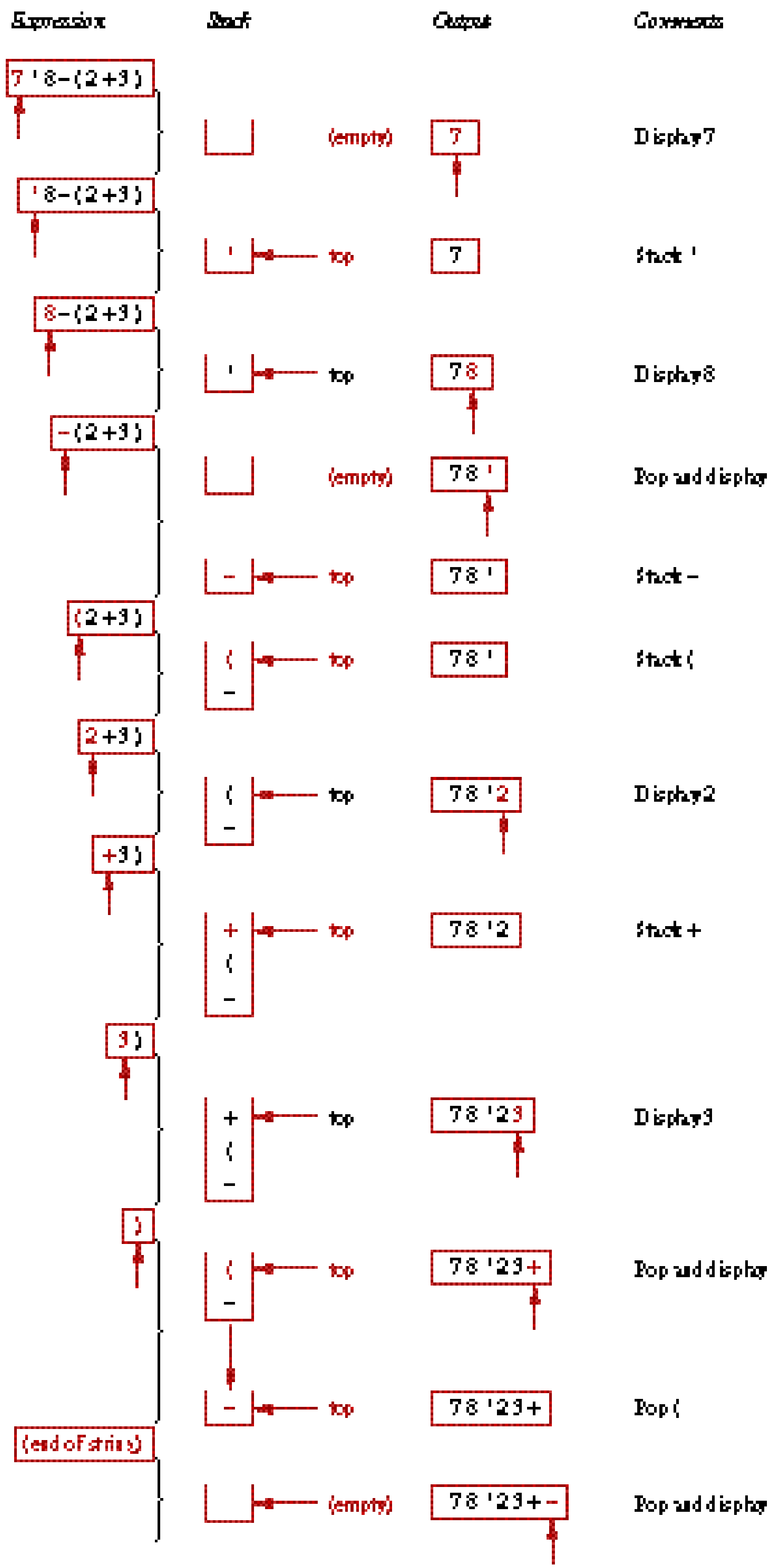
      (iii) an operator: If the stack is empty or *Token* has a higher priority than the top stack element, push *Token* onto the stack.

      Otherwise, pop and display the top stack element; then repeat the comparison of *Token* with the new top stack item.

      Note: A left parenthesis in the stack is assumed to have a lower priority than that of operators.

      (iv) an operand: Display it.

3. When the end of the infix expression is reached, pop and display stack items until the stack is empty.


Expression
Stack
Output
Comments
7 ' 8 − (2 + 3)
(empty)
7
Display 7
' 8 − (2 + 3)
top
Stack '
8 − (2 + 3)
7 8
Display 8
− (2 + 3)
7 8 '
Pop and display
Stack −
(2 + 3)
Stack (
2 + 3)
7 8 ' 2
Display 2
+ 3)
Stack +
3)
7 8 ' 2 3
Display 3
)
7 8 ' 2 3 +
Pop and display
Pop (
(end of string)
7 8 ' 2 3 + −
Pop and display