Unit V
Stacks

- Basic concept, stack Abstract Data Type, Representation of Stacks Using Sequential Organization, stack operations, Multiple Stacks, Applications of Stack- Expression Evaluation and Conversion, Polish notation and expression conversion, Need for prefix and postfix expressions, Postfix expression evaluation, Linked Stack and Operations.
- Recursion- concept, variants of recursion- direct, indirect, tail and tree, backtracking algorithmic strategy, use of stack in backtracking.
- Case Studies : Android- multiple tasks/multiple activities and back-stack, Tower of Hanoi, 4 Queens problem.

Stacks

- Stack: what is it?
- ADT
- Applications
- Implementation(s)

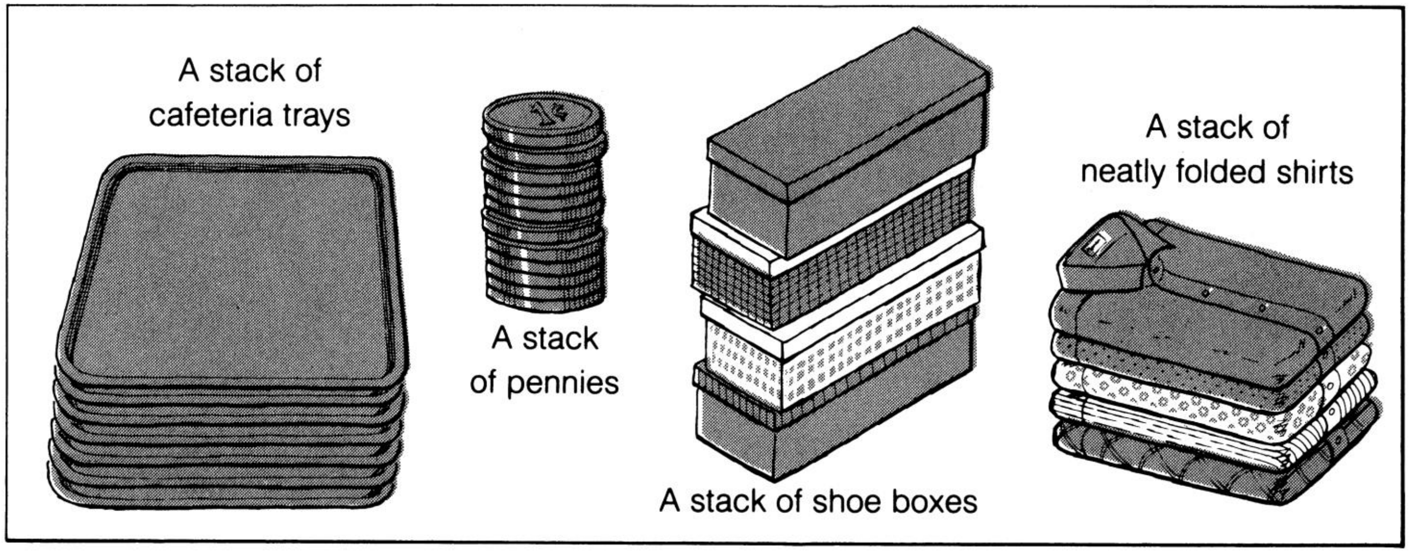
What is a stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed
- Example

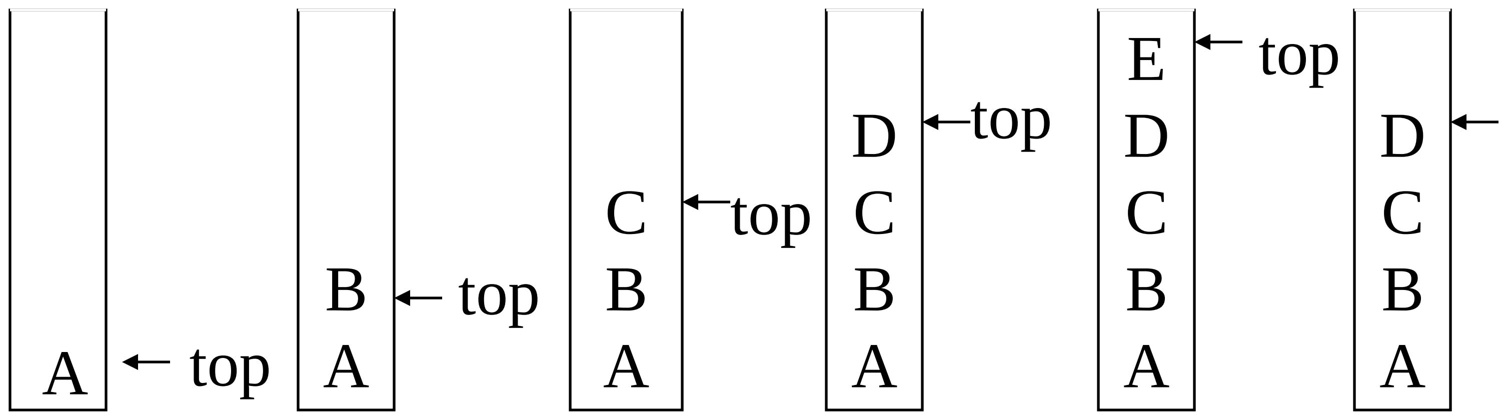
- Which is the first element to pick up?

Stack Definition

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Last In First Out



What is the difference?

- Stack & Queue vs. Array
  - Arrays are data storage structures while stacks and queues are specialized DS and used as programmer's tools.
- Stack – a container that allows push and pop
- Queue - a container that allows enqueue and dequeue
- No concern on implementation details.
- In an array any item can be accessed, while in these data structures access is restricted.
- They are more abstract than arrays.

Stack Applications

- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack (read more from your text)
  - Evaluating expressions

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

BASIC STACK OPERATIONS

- Initialize the Stack.
- Pop an item off the top of the stack (delete an item)
- Push an item onto the top of the stack (insert an item)
- Is the Stack empty?
- Is the Stack full?
- Clear the Stack
- Determine Stack Size

Stack ADT

objects: *a finite ordered list with zero or more elements.*
methods:

for all stack ∈ Stack, item ∈ element, max_stack_size ∈ *positive integer*

Stack *createS*(max_stack_size) *::=*

    create an empty stack whose maximum size is max_stack_size

Boolean *isFull*(stack, max_stack_size) *::=*

    if *(number of elements in* stack == max_stack_size*)*
    return *TRUE*
    else return *FALSE*

Stack *push*(stack, item) *::=*

    if *(IsFull(*stack*))* stack_full
    else *insert* item *into top of* stack *and* ***return***

Stack ADT (cont'd)

```
Boolean isEmpty(stack) ::=
                if(stack == CreateS(max_stack_size))
                return TRUE
                else return FALSE
Element pop(stack) ::=
                if(IsEmpty(stack)) return
                else remove and return the item on the top
                    of the stack.
```

Run-time Stack

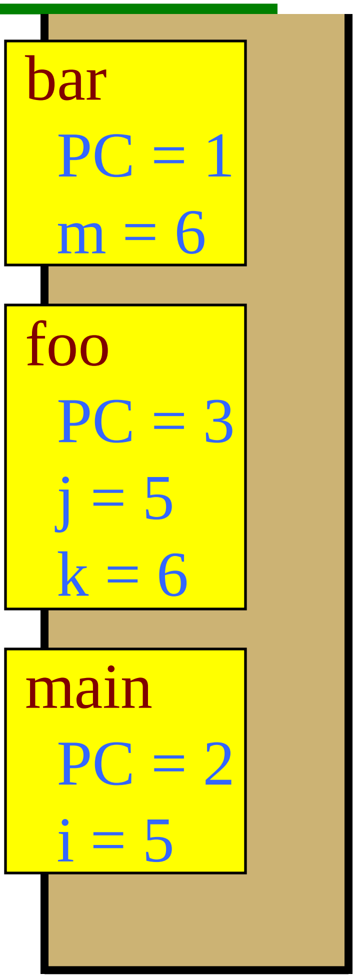
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
    int i;

   i = 5;
    foo(i);
    }

foo(int j) {
    int k;
    k = j+1;
    bar(k);
    }

bar(int m) {
    …
    }
```



Array Implementation of the Stacks

- The stacks can be implemented by the use of ***arrays*** and ***linked lists***.
  - One way to implement the stack is to have a data structure where a variable called top keeps the location of the elements in the stack (array)
  - An array is used to store the elements in the stack

Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- last index in the array is the *top*
- Increment *top* when one element is pushed, decrement after pop

Stack Implementation: CreateS, isEmpty, isFull

Stack *createS(max_stack_size)* *::=*

```
#define MAX_STACK_SIZE   100 /* maximum stack size */
  typedef struct {
              data_type  ST[MAX_STACK_SIZE];
               int top;
           } stack;

stack  s;
```

Stack intitialization

```
void  initialize_stack(stack *sp)
{
          sp->top = -1;
}
```

Stack empty

```
int   is_stack_empty(int top)
{
          if(top ==-1)
               return 1;
         else
               return 0;
}
```

Stack full

```
int   is_stack_full(int top)
{
          if(top ==MAX_STACK_SIZE-1)
               return 1;
          else
               return 0;
}
```

Push

```
void push(stack *sp, element item)
{
      if (stack is not full)
       {
            sp->top = sp->top + 1;
            sp->ST[sp->top] = item;
     }
     else
        display stack is full;
}
```

Pop

```
element pop(stack *sp)
{
    if (stack is not empty)
      {
         item = sp->ST[sp->top];
          sp->top = sp->top – 1;
           return item;
      }
       else
           return an error key
}
```

List Based Stack Implementation:

```
 typedef struct  STACK {
                int data ;
                 struct STACK *down;
             } stack;

stack  *top;

Initialize top = NULL;
```

List-based Stack Implementation: Push

```
void push(int X)
{
    stack *node;
    node  = (stack *) malloc (sizeof (stack));
  if (node == NULL)
   {
      printf(“ Stack memory is full\n”);
     exit(1);
    }
    else
   {
     node->data = X;
    node->down = top;
    top  = node;
   }
}
```

Pop

```
int  pop()
 {
     int X;
    stack *temp;
    if (top == NULL)
    {
      printf(“The stack is empty\n”);
      exit(1);
    }
    else
   {
      X = top->data;
      temp = top;
     top = top->down;
    free(temp);
    return X;
}
```

Algorithm Analysis

push	O(?)
pop	O(?)
isEmpty	O(?)
isFull	O(?)

- What if *top* is stored at the beginning of the array?

Stack Summary

- Stack Operation Complexity for Different Implementations

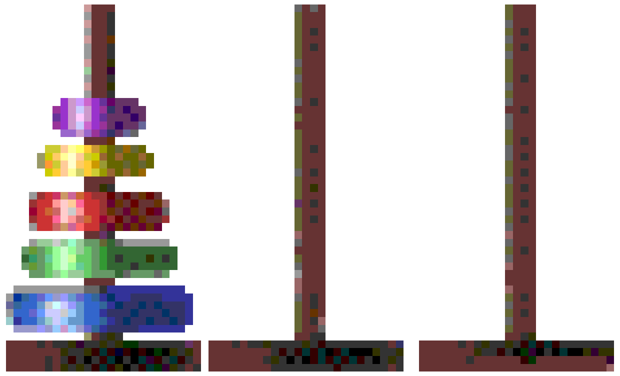
	Array Fixed-Size	List Singly-Linked
Pop()	O(1)	O(1)
Push(o)	O(1)	O(1)
Top()	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)

A Legend
The Towers of Hanoi

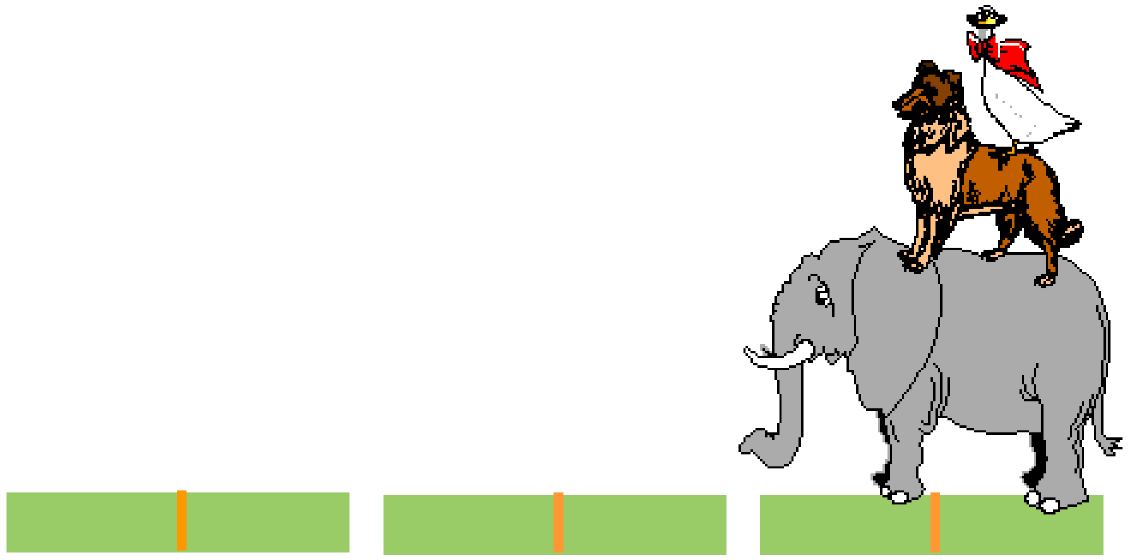
- *In the great temple of Brahma in Benares, on a brass plate under the dome that marks the center of the world, there are 64 disks of pure gold that the priests carry one at a time between these diamond needles according to Brahma's immutable law: No disk may be placed on a smaller disk. In the begging of the world all 64 disks formed the Tower of Brahma on one needle. Now, however, the process of transfer of the tower from one needle to another is in mid course. When the last disk is finally in place, once again forming the Tower of Brahma but on a different needle, then will come the end of the world and all will turn to dust.*

The Towers of Hanoi
A Stack-based Application

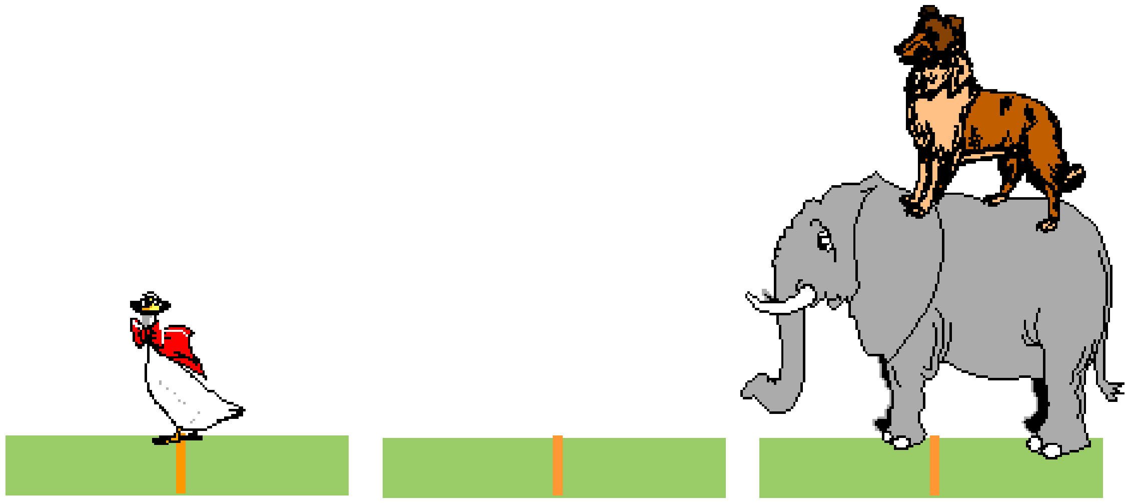
- GIVEN: three poles
- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.



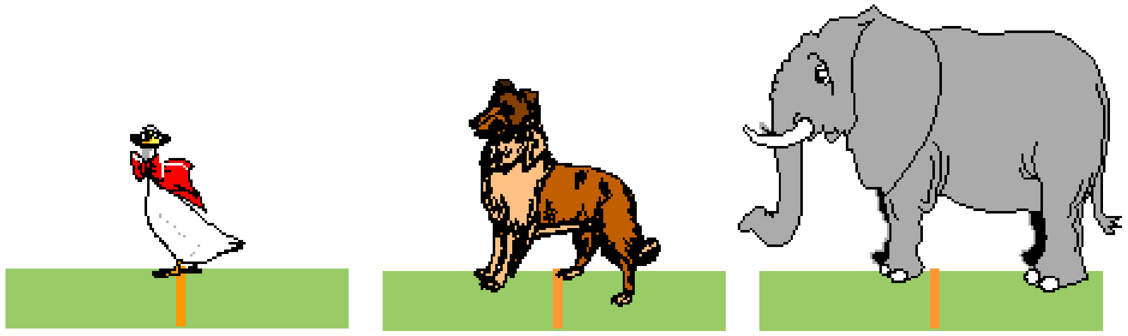
Towers of Hanoi



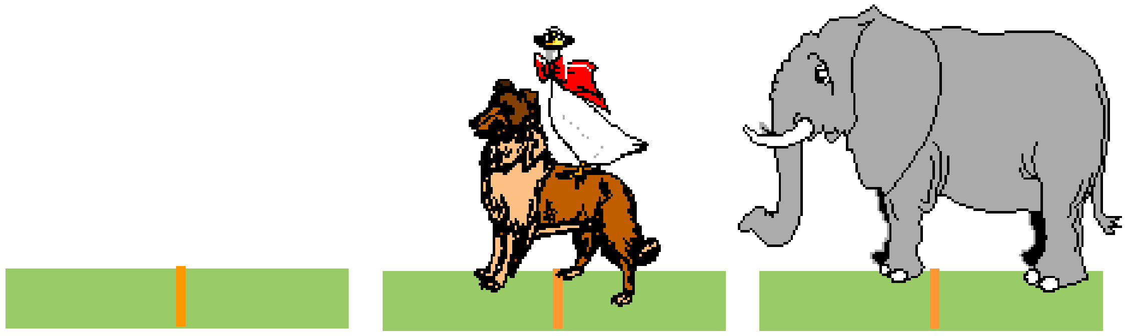
Towers of Hanoi



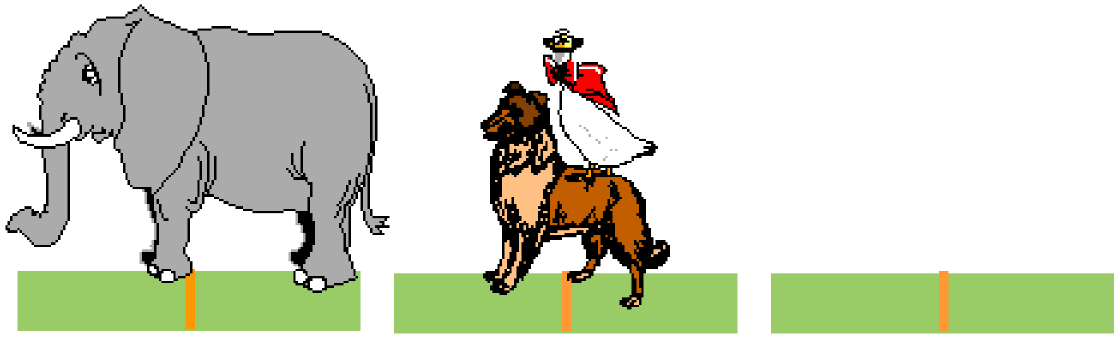
Towers of Hanoi



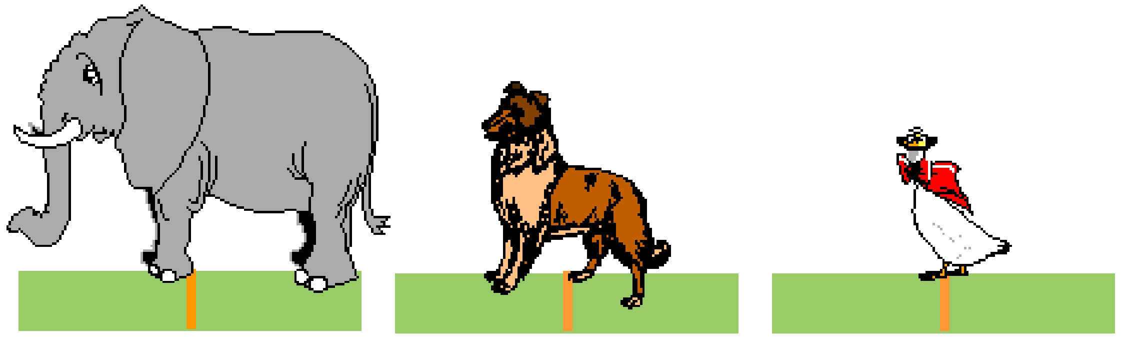
Towers of Hanoi



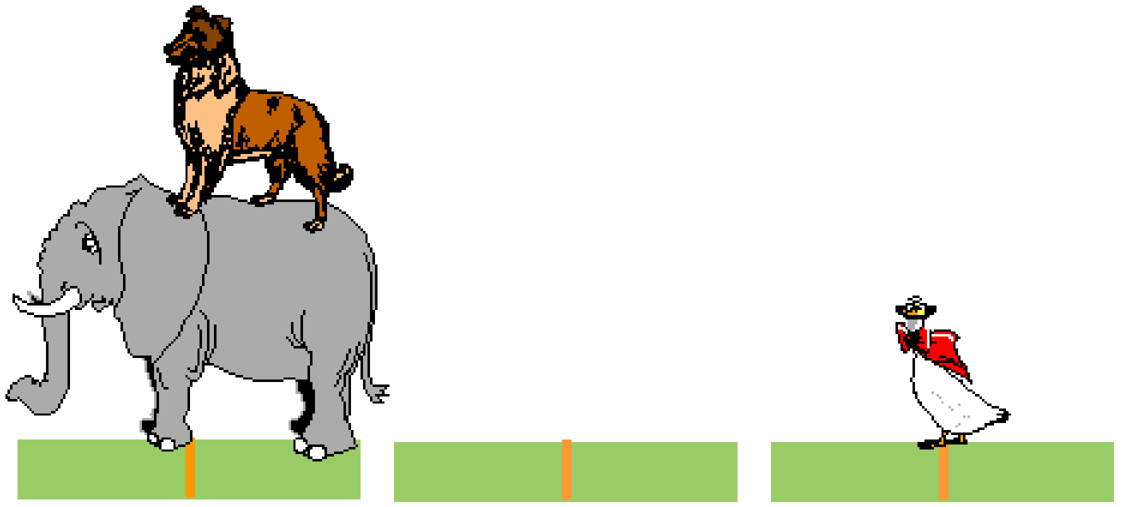
Towers of Hanoi



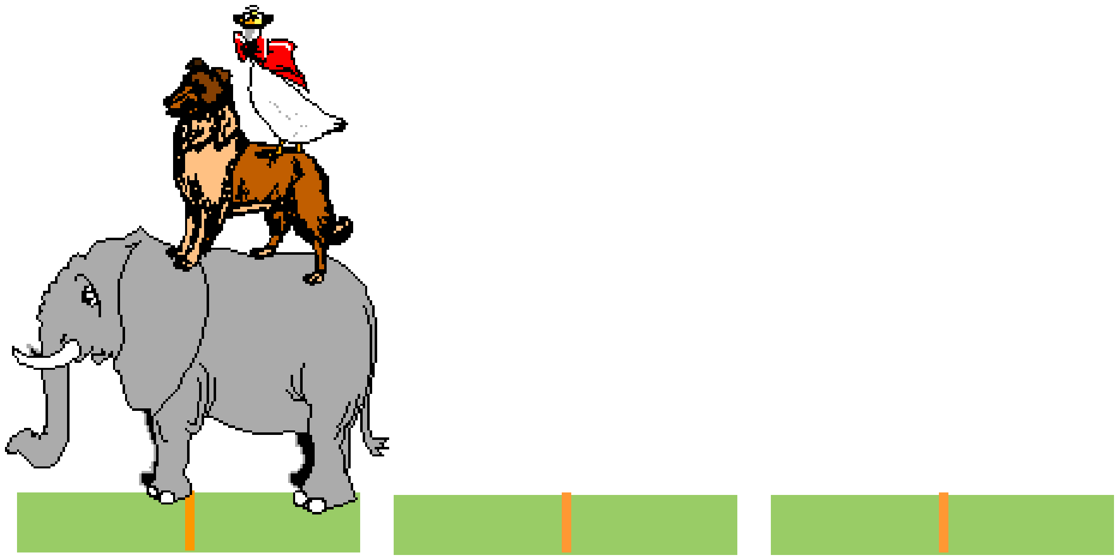
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



Towers of Hanoi – Recursive Solution

```
  void hanoi (int discs,
                   Stack fromPole,
                   Stack toPole,
                   Stack aux) {
Disc d;
if( discs >= 1) {
        hanoi(discs-1, fromPole, aux, toPole);
        d = fromPole.pop();
        toPole.push(d);
        hanoi(discs-1,aux, toPole, fromPole);
}
```

Is the End of the World Approaching?

- Problem complexity 2^n
- 64 gold discs
- Given 1 move a second

→ 600,000,000,000 years until the end of the world ☺

Stack Application : Expression Conversion

-

  INFIX, **PREFIX**, & **POSTFIX** EXPRESSIONS

Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands. Example of familiar operators include +,-,*, /, ^
- Considering these definitions of operands and operators now we can write an example of expression as x+y*z.

Infix Notation

- **INFIX:** From our schools times we have been familiar with the expressions in which operands surround the operator, e.g. x+y, 6*3 etc this way of writing the Expressions is called infix notation.
- We usually write algebraic expressions like this: a + b
- This is called **infix notation**, because the operator ("+") is inside the expression
- A problem is that we need parentheses or precedence rules to handle more complicated expressions:

For Example : a + b * c = (a + b) * c ?

= a + (b * c) ?

Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d
- This is done by assigning operator priorities.
  - priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
  - a + b - c
  - a * b / c / d

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
  - (a + b) * (c – d) / (e – f)

Infix, Postfix, & Prefix notation

- There is no reason we can't place the operator somewhere else.
- How ?
- **Infix** notation : a + b
- **Prefix** notation : + a b
- **Postfix** notation: a b +

Other Names

- **Prefix** notation was introduced by the Polish logician **Lukasiewicz**, and is sometimes called “Polish notation”. In the prefix notation, operator comes before the operands, e.g. +xy, *+xyz etc.
- **Postfix** notation is sometimes called “reverse Polish notation” or **RPN.** They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc.

Why ?

- **Question:** Why would anyone ever want to use anything so “unnatural,” when infix seems to work just fine?

- **Answer:** With postfix and prefix notations, parentheses are no longer needed!

Example

infix	***postfix***	***prefix***
(a + b) * c	a b + c *	* + a b c
a + (b * c)	a b c * +	+ a * b c

Infix form : *<identifier> <operator> <identifier>*

Postfix form : *<identifier> <identifier> <operator>*

Prefix form : *<operator> <identifier> <identifier>*

Convert from Infix to Prefix and Postfix (Practice)

- x
- $x + y$
- $(x + y) - z$
- $w * ((x + y) - z)$
- $(2 * a) / ((a + b) * (a - c))$

Convert from Postfix to Infix (Practice)

- $3\ r\ -$
- $1\ 3\ r\ -\ +$
- $s\ t\ *\ 1\ 3\ r\ -\ +\ +$
- $v\ w\ x\ y\ z\ *\ -\ +\ *$

WHY

- Why to use these weird looking PREFIX and POSTFIX notations when we have simple INFIX notation?

- To our surprise INFIX notations are not as simple as they seem specially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property
  - For example expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).

- To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Infix Expression Is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier to evaluate expressions that are in these forms.

- Due to above mentioned problem of considering operators' Priority and Associative property while evaluating an expression using infix notation, we use prefix and postfix notations.

- Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Priority and Associative property (order of brackets).
  - E.g. x/y*z becomes */xyz in prefix and xy/z* in postfix. Both prefix and postfix notations make **Expression Evaluation** a lot easier.

Examples of infix to prefix and post fix

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

  infix: 2+5 *postfix*: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

*Suppose that we would like to rewrite A+B*C in postfix*

- Applying the rules of precedence,we obtained

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication,Let D=BC*

A+D Convert the addition

A(D)+

ABC*+ Postfix Form

Postfix Examples

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

Why ? No brackets necessary !

When do we need to use them… ☺

- So, what is actually done is expression is scanned from user in infix form; it is converted into prefix or postfix form and then evaluated without considering the parenthesis and priority of the operators.

Algorithm for Infix to Postfix

1) Examine the next element in the input.
2) If it is operand, output it.
3) If it is opening parenthesis, push it on stack.
4) If it is an operator, then
   - i) If stack is empty, push operator on stack.
   - ii) If the top of stack is opening parenthesis, push operator on stack
   - iii) If it has higher priority than the top of stack, push operator on stack.
   - iv) Else pop the operator from the stack and output it, repeat step 4
5) If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6) If there is more input go to step 1
7) If there is no more input, pop the remaining operators to output.

Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

	(	^	*	/	+	-
ICP	5	4	2	2	1	1
ISP	0	3	2	2	1	1

```
precedence stack[MAX_STACK_SIZE];
/* ISP and ICP arrays -- index is value of precedence
opening bracket,power,multiply/divide, plus/minus */
static int ISP [ ] = {0, 3, 2, 1};
static int ICP [ ] = {5, 4, 2,1 };
```

ISP: in-stack precedence
ICP: incoming precedence

Infix to Postfix

```
for(Each  input character in the input expression)
 {
         if(input character is operand)  // checking for operand
             Ouput it to postfix expressionpostfix[k] = infix[i];
         else
         {
           if(input character is closing bracket)
          {
             do
              {    pop a character  from  the stack and ouput it to postfix   expression.
              }while( stack of top is not equal to opening bracket);
              Pop the opening bracket
          }
          else  // it is one of the operator
          {
              while(ICP[input operator] <= ISP[ stack[top]] && stack is not empty)
                    pop a character  from  the stack and output it to postfix  expression.
              push the input operator on the stack
          }
         }
 }
 while(stack is not empty)
    Pop a character  from  the stack and output it to postfix  expression.
```

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Infix to Postfix Conversion example

(((A + B) * (C - E)) / (F + G))

- stack: <empty>
- output: []

Infix to Postfix

((A + B) * (C - E)) / (F + G))

- stack: (
- output: []

Infix to Postfix

(A + B) * (C - E)) / (F + G))

- stack: ((
- output: []

Infix to Postfix

A + B) * (C - E)) / (F + G))

- stack: (((
- output: []

Infix to Postfix

+ B) * (C - E)) / (F + G))

- stack: (((
- output: [A]

Infix to Postfix

B) * (C - E)) / (F + G))

- stack: (((+
- output: [A]

Infix to Postfix

) * (C - E)) / (F + G))

- stack: (((+
- output: [A B]

Infix to Postfix

* (C - E)) / (F + G))

- stack: ((
- output: [A B +]

Infix to Postfix

(C - E)) / (F + G))

- stack: ((*
- output: [A B +]

Infix to Postfix

C - E)) / (F + G))

- stack: ((* (
- output: [A B +]

Infix to Postfix

- E)) / (F + G))

- stack: ((* (
- output: [A B + C]

Infix to Postfix

E)) / (F + G))

- stack: ((* (-
- output: [A B + C]

Infix to Postfix

)) / (F + G))

- stack: ((* (-
- output: [A B + C E]

Infix to Postfix

) / (F + G))

- stack: ((*
- output: [A B + C E -]

Infix to Postfix

/ (F + G))

- stack: (
- output: [A B + C E - *]

Infix to Postfix

(F + G))

- stack: (/
- output: [A B + C E - *]

Infix to Postfix

F + G))

- stack: (/ (
- output: [A B + C E - *]

Infix to Postfix

+ G))

- stack: (/ (
- output: [A B + C E - * F]

Infix to Postfix

G))

- stack: (/ (+
- output: [A B + C E - * F]

Infix to Postfix

))

- stack: (/ (+
- output: [A B + C E - * F G]

Infix to Postfix

)

- stack: (/
- output: [A B + C E - * F G +]

Infix to Postfix

- stack: <empty>
- output: [A B + C E - * F G + /]

Example

- (5 + 6) * 9 +10

will be

- 5 6 + 9 * 10 +

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:
((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:
(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
ξξ	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= ⊠	assignment	2	right-to-left
,	comma	1	left-to-right

Evaluation of a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we **push** it into a stack. When we reach an operator, its operands will then be top two elements on the stack
- We can then **pop** these two elements, perform the indicated operation on them, and **push** the result on the stack.
- So that it will be available for use as an operand of the next operator.

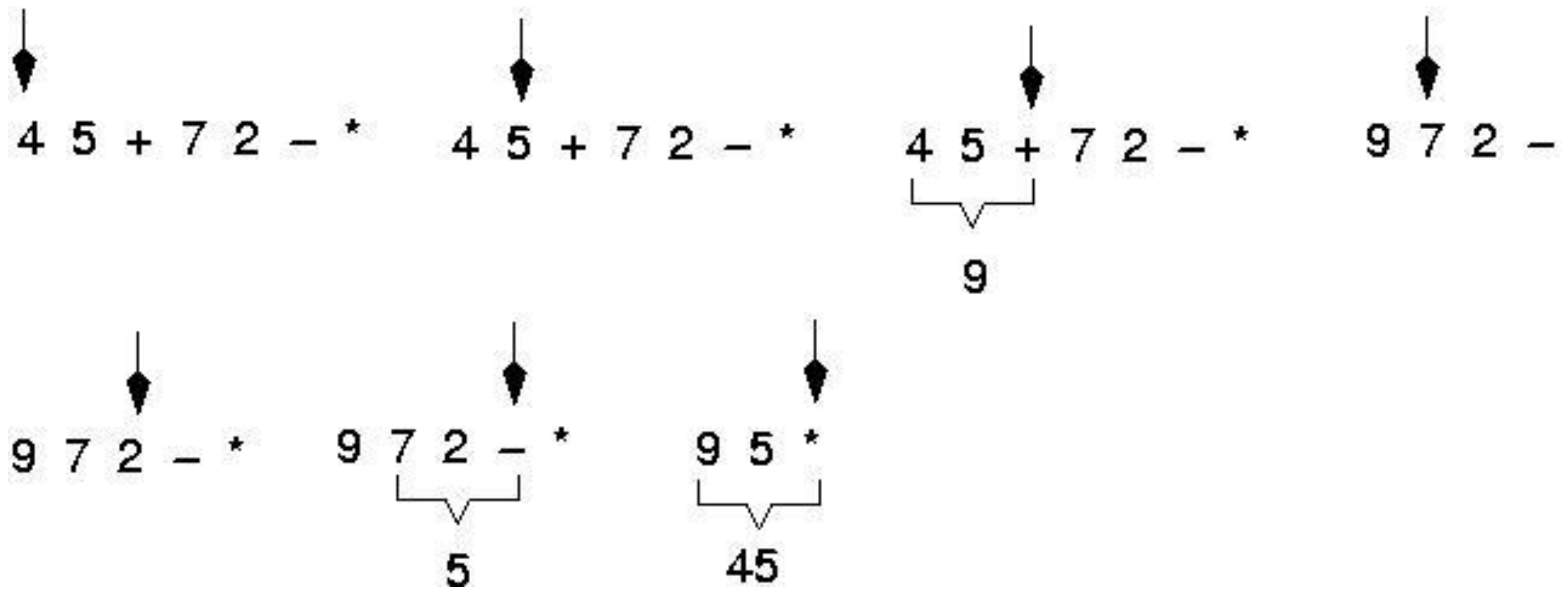
Evaluating Postfix Notation

- Use a stack to evaluate an expression in postfix notation.
- The postfix expression to be evaluated is scanned from left to right.
- Variables or constants are pushed onto the stack.
- When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

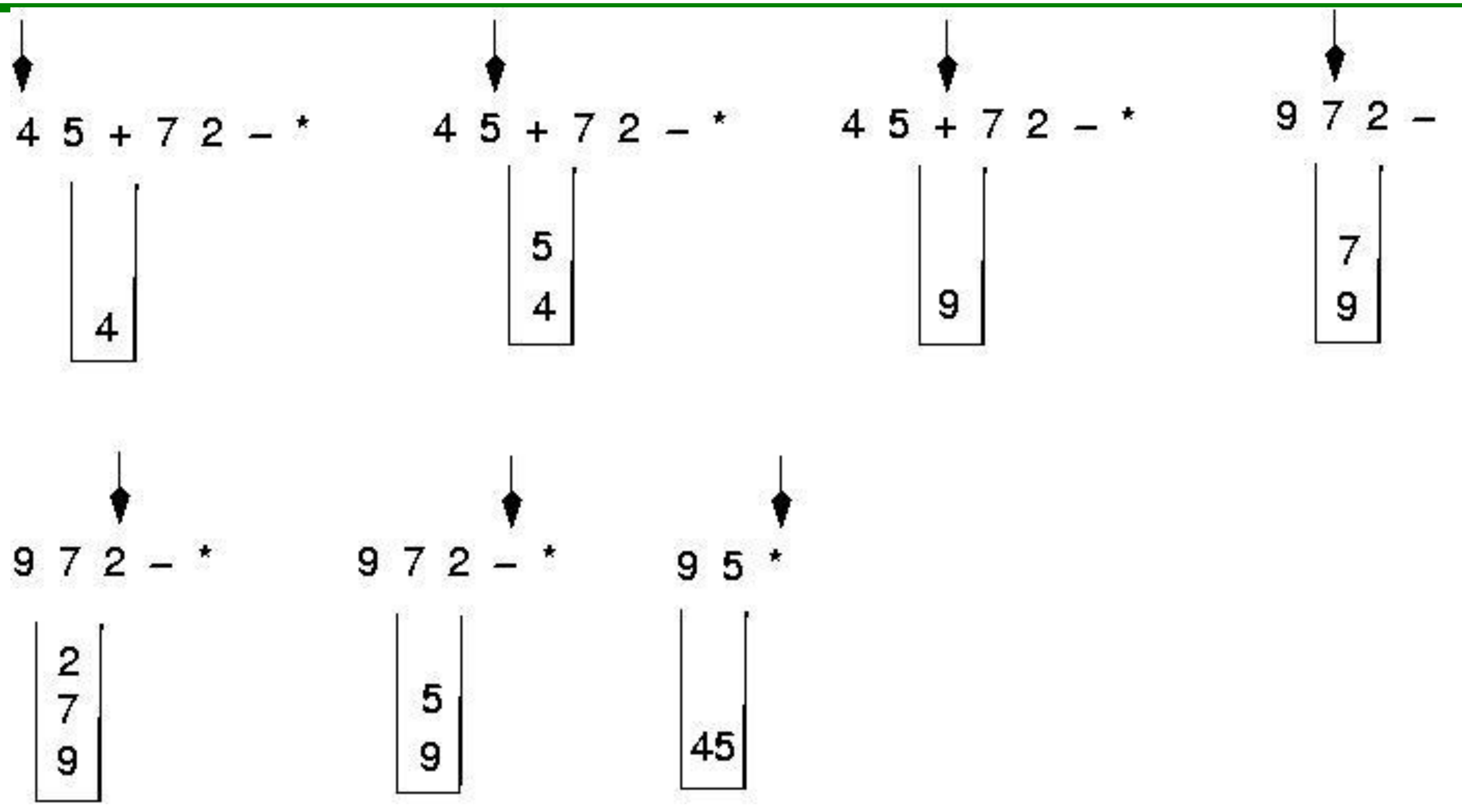
Evaluating a postfix expression

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack,apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Example: postfix expressions (cont.)



Postfix expressions: Algorithm using stacks (cont.)



Algorithm for evaluating a postfix expression (Cond.)

```
WHILE more input items exist
{
    If symb is an operand
        then push (opndstk,symb)
    else  //symbol is an operator
    {
        Opnd2=pop(opndstk);
        Opnd1=pop(opndnstk);
        Value = result of applying symb to opnd1 & opnd2
        Push(opndstk,value);
    }               //End of else
} // end while
Result = pop (opndstk);
```

*Question : Evaluate the following expression in postfix : 623+-382/+*2^3+*

Final answer is

- 49
- 51
- 52
- 7
- None of these

*Evaluate- 623+-382/+*2^3+*

Symbol	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

*Evaluate- 623+-382/+*2^3+*

Symbol	opnd1	opnd2	value	opndstk
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Infix to Prefix

- Will require two stacks :
  - Containing operators for assigning priorities
  - Containing operands or operand expressions.

- Everytime we get the operand, push it into operand stack.
- Operator will be pushed in operatorstack as per rules of previous method.
- When an operator is poped from operator stack, the corresponding two operands are poped from operandstack, say O1 and O2 respectively, We form the prefix expression as operator, O1 , O2 and this prefix expression will be treated as single entity and pushed on operandstack.
- Whenever opening bracket is received, push it onto operator stack.
- Whenever closing bracket is received, we pop the operators from operator stack, till opening bracket is recieived.
- At the end operator stack will be empty and operand stack will contain single operand in the form of prefix expression.

Prefix to infix or Postfix

- Common stack for operators and operands.
- The first conversion will take place only when we get successive operands.
- For conversion, we will first push all the operators, till we get an operand. Push that operand also in the stack.
- Now whenever we get the operand, we should check the stacktop. If stacktop contains operator, push the operand. When stacktop is operand, then the current operand will be opnd2. Pop the operand, it will be opnd1 and pop the operator. Form the required expression and again push in the stack.
- Remember in this case, stack will never contain two successive operands. Even after forming a new expression, which hence forth behaves as an operand, before pushing it into the stack, check whether the top is operator, otherwise repeat the procedure. The operators will be pushed unconditionally in the stack.
- When we are actually building the infix expressions, we will always be inserting the brackets. Hence at the end of the infix expression, we get a fully parenthesized form.

Application: Parenthesis Matching

- Problem: match the left and right parentheses in a character string
- (a*(b+c)+d)
  - Left parentheses: position 0 and 3
  - Right parentheses: position 7 and 10
  - Left at position 0 matches with right at position 10
- (a+b))*((c+d)
  - (0,4)
  - Right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - Left parenthesis at 7 has no matching right parenthesis

Parenthesis Matching

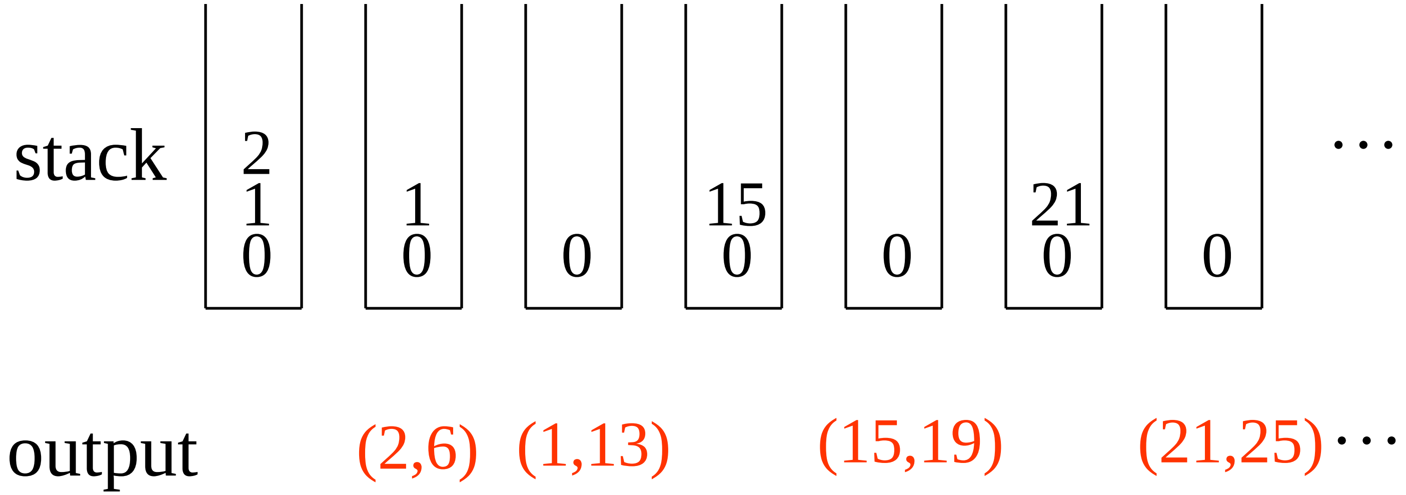
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.

    (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- **How do we implement this using a stack?**

1. Scan expression from left to right
2. When a left parenthesis is encountered, add its position to the stack
3. When a right parenthesis is encountered, remove matching position from the stack

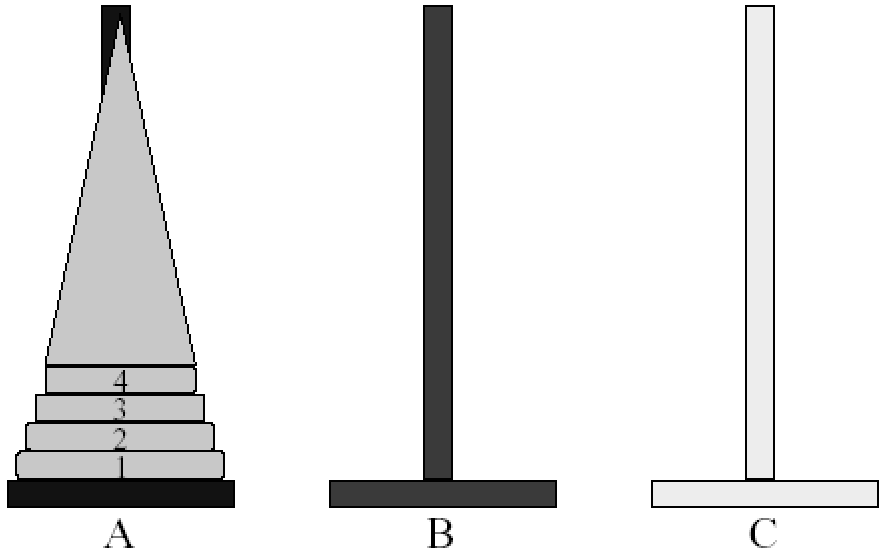
Example of Parenthesis Matching

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)



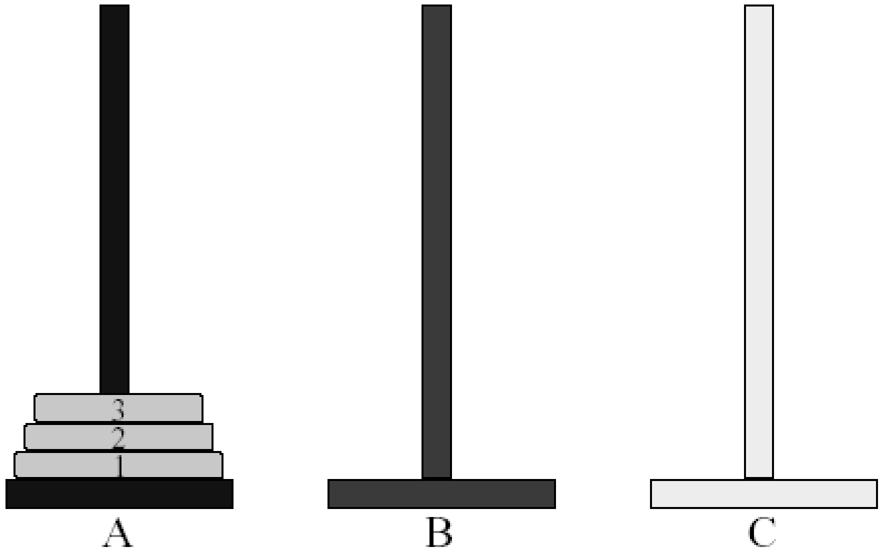
– Do the same for (a-b)*(c+d/(e-f))/(g+h)

Application: *Towers of Hanoi*

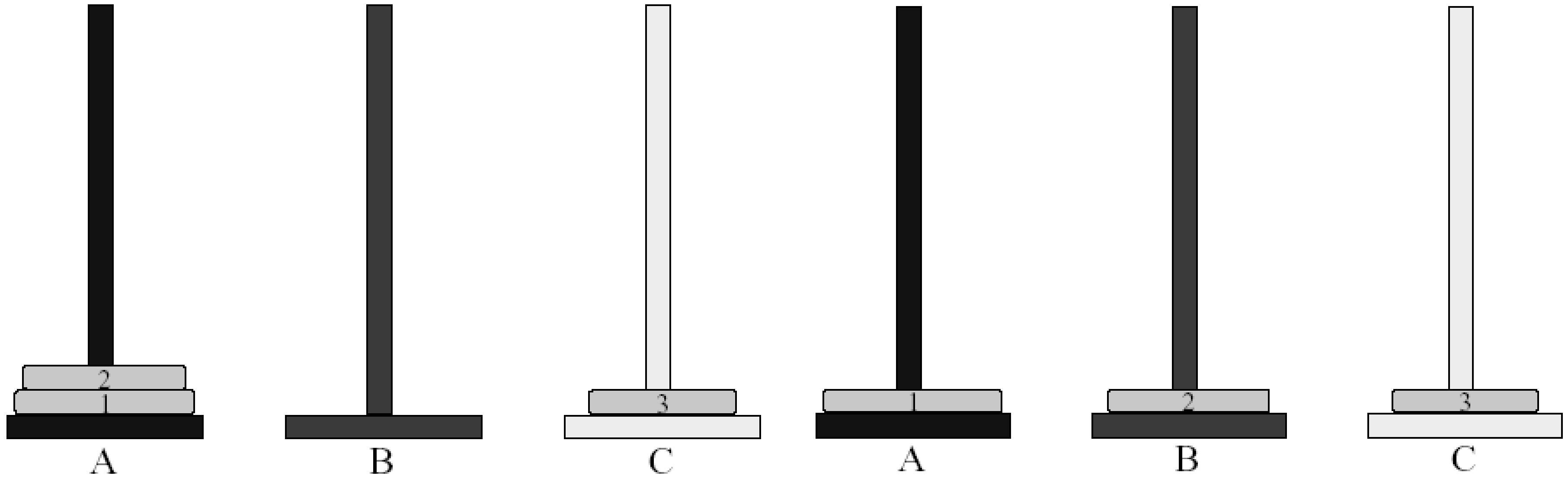


- The ancient Tower of Brahma ritual
- ***n*** disks to be moved from tower **A** to tower **C** with the following restrictions:
  - Move 1 disk at a time
  - Cannot place larger disk on top of a smaller one

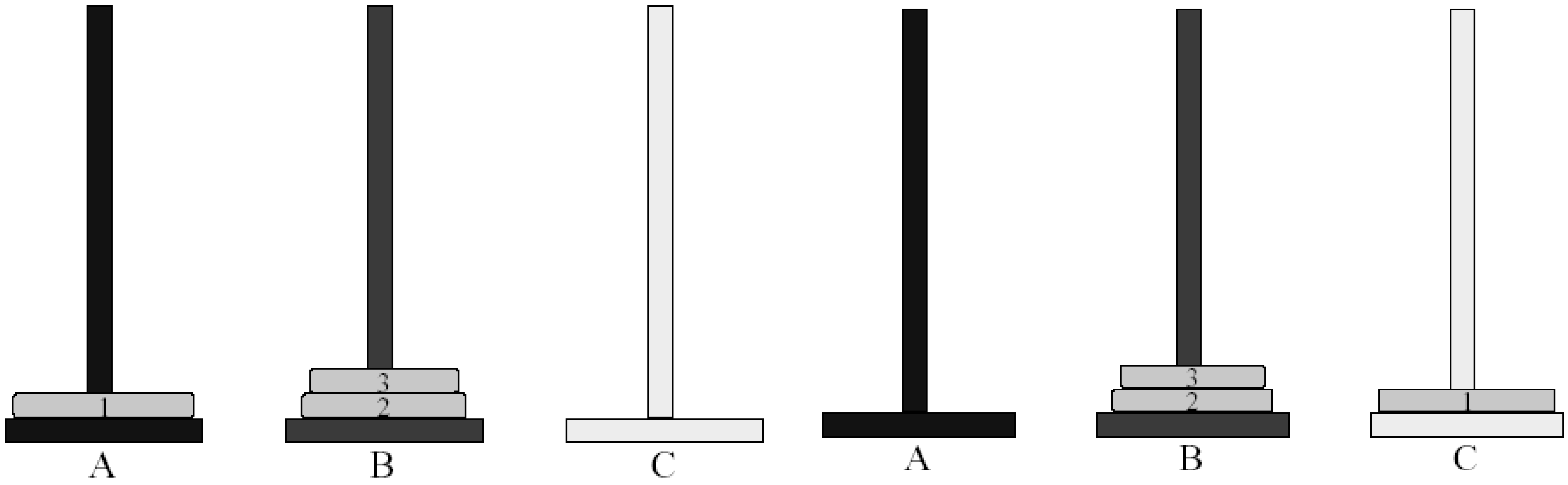
Let's solve the problem for 3 disks



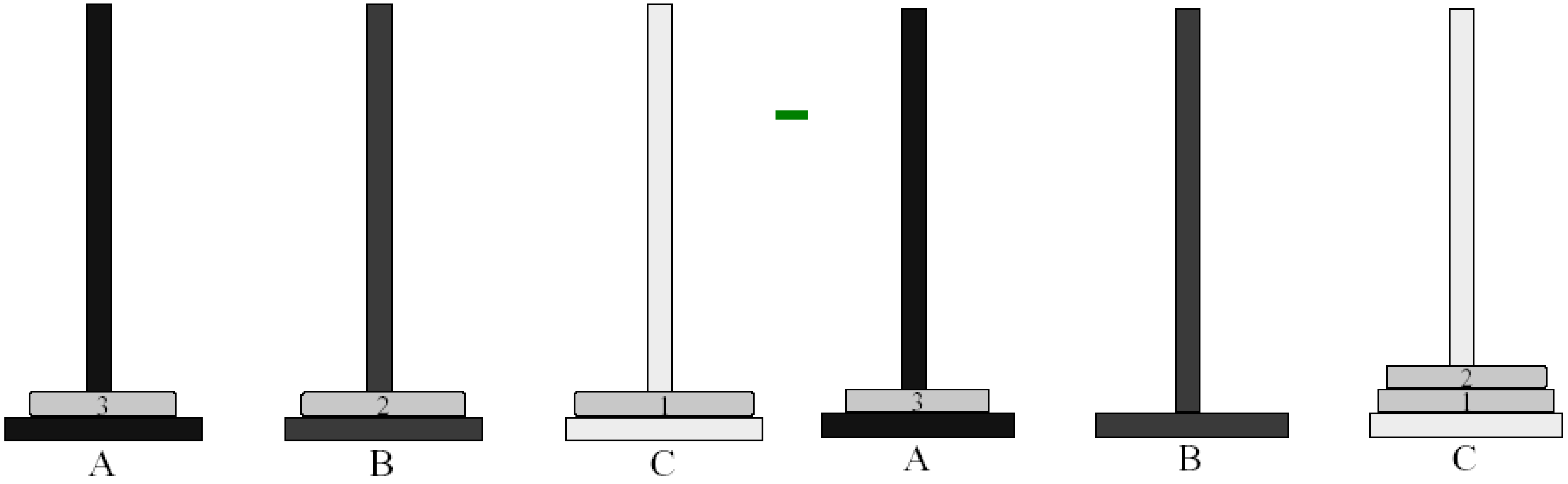
Towers of Hanoi (1, 2)



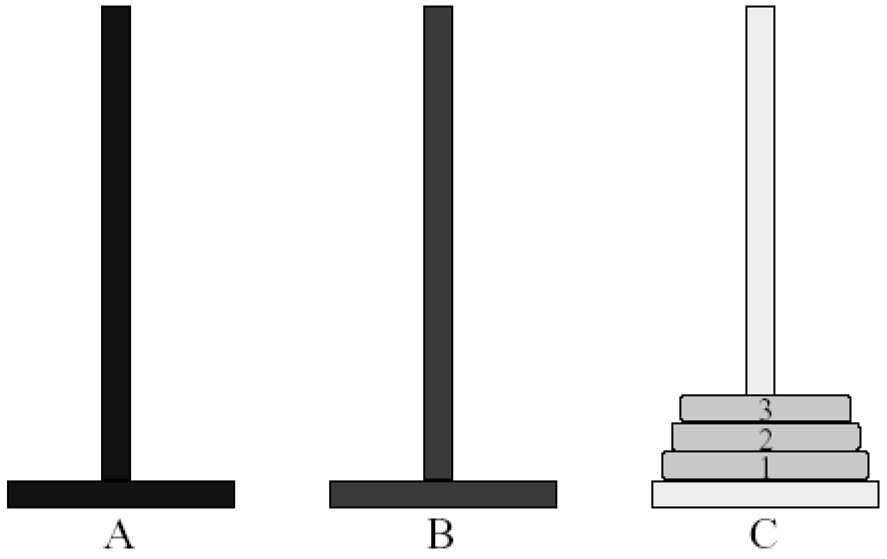
Towers of Hanoi (3, 4)



Towers of Hanoi (5, 6)



Towers of Hanoi (7)



- So, how many moves are needed for solving 3-disk Towers of Hanoi problem?

➔ 7

Time complexity for Towers of Hanoi

- A very elegant solution is to use **recursion.**
- The minimum number of moves required is $\boldsymbol{2^n\text{-}1}$
- Time complexity for Towers of Hanoi is $\Theta(2^n)$, which is exponential!
- Since disks are removed from each tower in a LIFO manner, each tower can be represented as a stack

Stack Applications

- Reversing Data: We can use stacks to reverse data. (example: files, strings)
  Very useful for finding palindromes.

Consider the following pseudocode:

```
1)  read (data)
2)  loop (data not EOF and stack not full)
        1) push (data)
        2) read (data)
3)  Loop (while stack notEmpty)
        1) pop (data)
        2) print (data)
```

Stack Applications

- Converting Decimal to Binary: Consider the following pseudocode

```
1)  Read (number)
2)  Loop (number > 0)
        1) digit = number modulo 2
        2) print (digit)
        3) number = number / 2
// from Data Structures by Gilbert and Forouzan
```

The problem with this code is that it will print the binary
number backwards. (ex: 19 becomes 11001000 instead of 00010011.)
To remedy this problem, instead of printing the digit right away, we can
push it onto the stack. Then after the number is done being converted, we
pop the digit out of the stack and print it.

Stack Applications

- Postponement: Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a + b (what we use in grammar school)
- Postfix: a b +

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Infix to Postfix Conversion

- Rules:
  - Operands immediately go directly to output
  - Operators are pushed into the stack (including parenthesis)
    - Check to see if stack top operator is less than current operator
    - If the top operator is less than, push the current operator onto stack
    - If the top operator is greater than the current, pop top operator and push onto stack, push current operator onto stack
    - Priority 2: * /
    - Priority 1: + -
    - Priority 0: (

If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

Infix to Postfix Example

A + B * C - D / E

Infix	**Stack(bot->top)**	**Postfix**
a) A + B * C - D / E		
b) + B * C - D / E		A
c) B * C - D / E	+	A
d) * C - D / E	+	A B
e) C - D / E	+ *	A B
f) - D / E	+ *	A B C
g) D / E	+ -	A B C *
h) / E	+ -	A B C * D
i) E	+ - /	A B C * D
j)	+ - /	A B C * D E
k)		A B C * D E / - +

Infix to Postfix Example #2

A * B - (C + D) + E

	Infix	**Stack(bot->top)**	**Postfix**
a)	A * B - (C - D) + E	empty	empty
b)	* B - (C + D) + E	empty	A
c)	B - (C + D) + E	*	A
d)	- (C + D) + E	*	A B
e)	- (C + D) + E	empty	A B *
f)	(C + D) + E	-	A B *
g)	C + D) + E	- (	A B *
h)	+ D) + E	- (	A B * C
i)	D) + E	- (+	A B * C
j)	) + E	- (+	A B * C D
k)	+ E	-	A B * C D +
l)	+ E	empty	A B * C D + -
m)	E	+	A B * C D + -
n)		+	A B * C D + - E
o)		empty	A B * C D + - E +

Postfix Evaulation

Operand: push

Operator: pop 2 operands, do the math, pop result
back onto stack

1 2 3 + *

	Postfix	Stack(bot -> top)
a)	1 2 3 + *	
b)	2 3 + *	1
c)	3 + *	1 2
d)	+ *	1 2 3
e)	*	1 5 // 5 from 2 + 3
f)		5 // 5 from 1 * 5

Backtracking

- Stacks can be used to backtrack to achieve certain goals.
- Usually, we set up backtrack tokens to indicate a backtrack opportunity.

Multistacks

- Here only one single one-dimensional array (Q) is used to store multiple stacks.
- B (i) denotes one position less than the position in Q for bottommost element of stacks i.
- T (i) denotes the top most element of stack i.
- m denotes the maximum size of the array being used.
- n denotes the number of stacks.
- We also assume that equal segments of array will be used for each stack.
- Initially let B (i) = T (i) = [m/n]*(i-1) where 1<= i <= n.
- Again we can have ***push*** or ***pop*** operations, which can be performed on each of these stacks .

Algorithms

Push (i, x)

```
{
    If (((i<n) && (T(i)==B(i+1)) || ((i>=n) && (T(i) ==m)));
        Then call STACK_FULL ;
    Else
    {
        T(i) = T(i) + 1;
        Q[T(i)] = x ;
    }
}
```

Algorithms

Pop (i, x)

```
{
    if ( T(i) = = B(i) )
        Then print that the stack is empty.
    Else
    {
        x = Q[T(i)];
        T[i] = T[i] – 1;
    }
}
```

ALGORITHM TO BE APPLIED WHEN T [i] = = B [i] CONDITION IS ENCOUNTERED WHILE DOING PUSH OPERATION.

STACK_FULL

{

1. Find j such that i < j <= n and there is a free space between stack j and stack (j +1).
   if such a j exist , then move stack i+1 , i+2 ,.........till j one position to the right and hence create space for element between stack i and i +1.
2. Else Find j such that 1 <= j < i and there is a free space between stack j and stacK(j +1). if such a j exist , then move stack j+1 , j+2 ,.......till i one position to the left and hence create space for element between stack i and i +1 .
3. Else If none of above is possible then there is no space left out in the one-dimensional array used hence print no space for push operation.

}

Multistacks

- Here only one single one-dimensional array (ST) is used to store multiple stacks.
- MAX(i) denotes the max position stack i can hold.
- Top(i) denotes the top most element of stack i.
- m denotes the maximum size of the array being used.
- n denotes the number of stacks.
- We also assume that equal segments of array will be used for each stack.
- Initially let MAX(i) = [m/n]*i where 1<= i <= n.
- Top(i) = (m/n * (i-1)) -1 ;
- Again we can have ***push*** or ***pop*** operations, which can be performed on each of these stacks .

Algorithms

Push (i, x)

```
{
    If (Top(i) == MAX(i) - 1);
        Then call STACK_FULL ;
    Else
    {
        Top(i) = Top(i) + 1;
        ST[Top(i)] = x ;
    }
}
```

Algorithms

Pop (i, x)

```
{
        if ( Top(i) = = (m/n*(i-1)) - 1)
                Then print that the stack is empty.
        Else
        {
                x = ST[Top(i)];
                Top[i] = Top[i] – 1;
        }
}
```

Recursion : Importance of stack in recursion

- Recursion is more than just a programming technique. It has two other uses in computer science and software engineering, namely:
- as a way of describing, defining, or specifying things.
- as a way of designing solutions to problems (divide and conquer).

Recursive definitions

- A recursive definition is one in which something is defined in terms of itself
- Almost every algorithm that requires looping can be defined iteratively or recursively
- All recursive definitions require two parts:
  - *Base case*
  - *Recursive step*
- The recursive step is the one that is defined in terms of itself
- The recursive step must always move closer to the base case

Recursion

- Recursion can be seen as building objects from objects that have set definitions. Recursion can also be seen in the opposite direction as objects that are defined from smaller and smaller parts. "Recursion is a different concept of circularity."(Dr. Britt, Computing Concepts Magazine, March 97, pg.78)

Iterative Definition

- In general, we can define the factorial function in the following way:

$$\text{Factorial}(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{array}\right]$$

$$\begin{aligned} 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120 \end{aligned}$$

Iterative Definition

- This is an ***iterative*** definition of the factorial function.
- It is iterative because the definition only contains the algorithm parameters and not the algorithm itself.
- This will be easier to see after defining the recursive implementation.

Recursive Definition

- We can also define the factorial function in the following way:

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{bmatrix}$$

Iterative vs. Recursive

- **Iterative**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Function does NOT calls itself

- **Recursive**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times \text{factorial}(n-1) & \text{if } n>0 \end{cases}$$

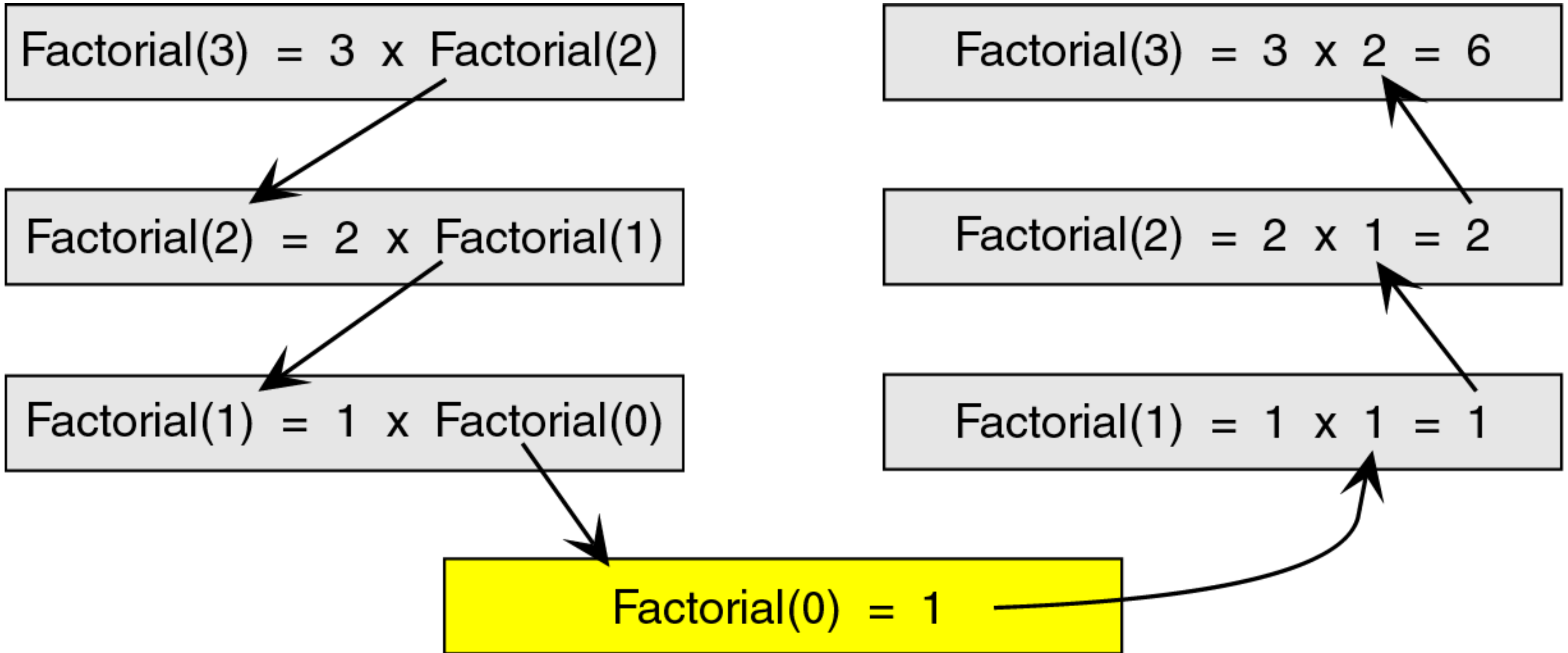
Function calls itself

Iteration vs. recursion

- Some things (e.g. reading from a file) are easier to implement iteratively
- Other things (e.g. mergesort) are easier to implement recursively
- Others are just as easy both ways
- When there is no real benefit to the programmer to choose recursion, iteration is the more efficient choice
- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version, are equivalent

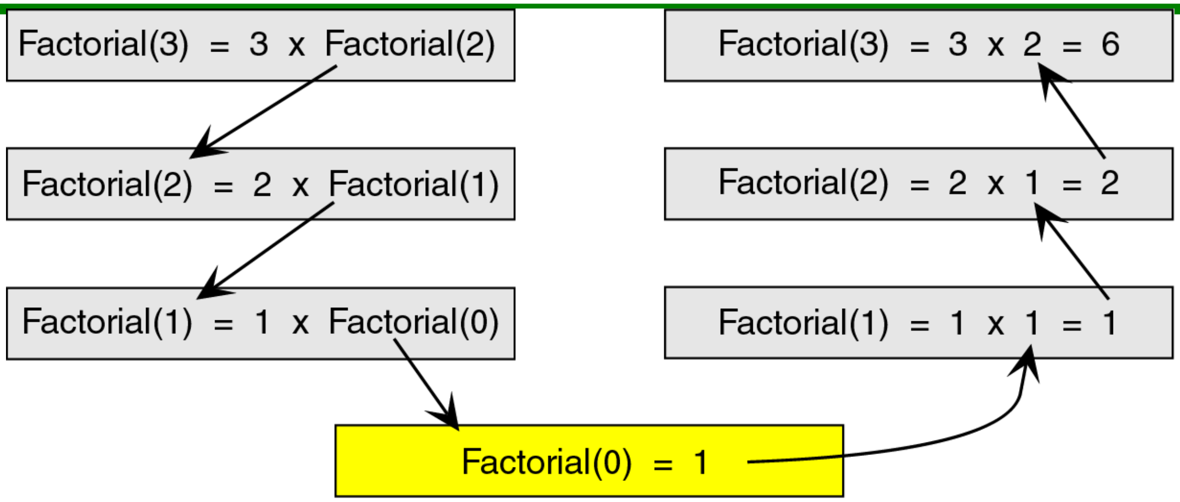
Recursion

- To see how the recursion works, let's break down the factorial function to solve factorial(3)



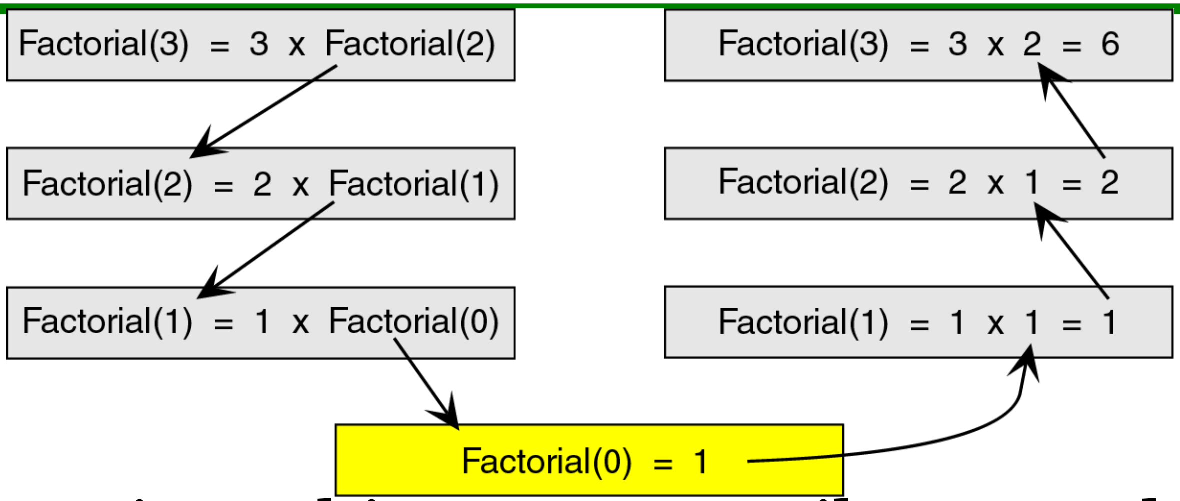
Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

Breakdown


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

- Here, we see that we start at the top level, factorial(3), and simplify the problem into 3 x factorial(2).
- Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(3) = 3 x 2 = 6
Factorial(2) = 2 x 1 = 2
Factorial(1) = 1 x 1 = 1

- We continue this process until we are able to reach a problem that has a known solution.
- In this case, that known solution is factorial(0) = 1.
- The functions then return in reverse order to complete the solution.

Breakdown

- This known solution is called the ***base case***.
- Every recursive algorithm must have a base case to simplify to.
- Otherwise, the algorithm would run forever (or until the computer ran out of memory).

Breakdown

- The other parts of the algorithm, excluding the base case, are known as the general case.
- For example:

  3 x factorial(2) ➔ general case
  2 x factorial(1) ➔ general case
  etc ...

Breakdown

- After looking at both iterative and recursive methods, it appears that the recursive method is much longer and more difficult.
- If that's the case, then why would we ever use recursion?
- It turns out that recursive techniques, although more complicated to solve by hand, are very simple and elegant to implement in a computer.

Iteration vs. Recursion

- Now that we know the difference between an iterative algorithm and a recursive algorithm, we will develop both an iterative and a recursive algorithm to calculate the factorial of a number.
- We will then compare the 2 algorithms.

Iterative Algorithm

```
factorial(n) {
    i = 1
    factN = 1
    loop (i <= n)
        factN = factN * i
        i = i + 1
    end loop
    return factN
}
```

The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.

Recursive Algorithm

```
factorial(n) {
   if (n = 0)
        return 1
   else
        return n*factorial(n-1)
   end if
}
```

Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version ➔

we have eliminated the loop and implemented the algorithm with 1 'if' statement.

How Recursion Works

- To truly understand how recursion works we need to first explore how any function call works.
- When a program calls a subroutine (function) the current function must suspend its processing.
- The called function then takes over control of the program.

How Recursion Works

- When the function is finished, it needs to return to the function that called it.
- The calling function then 'wakes up' and continues processing.
- One important point in this interaction is that, unless changed through call-by- reference, all local data in the calling module must remain unchanged.

How Recursion Works

- Therefore, when a function is called, some information needs to be saved in order to return the calling module back to its original state (i.e., the state it was in before the call).
- We need to save information such as the local variables and the spot in the code to return to after the called function is finished.

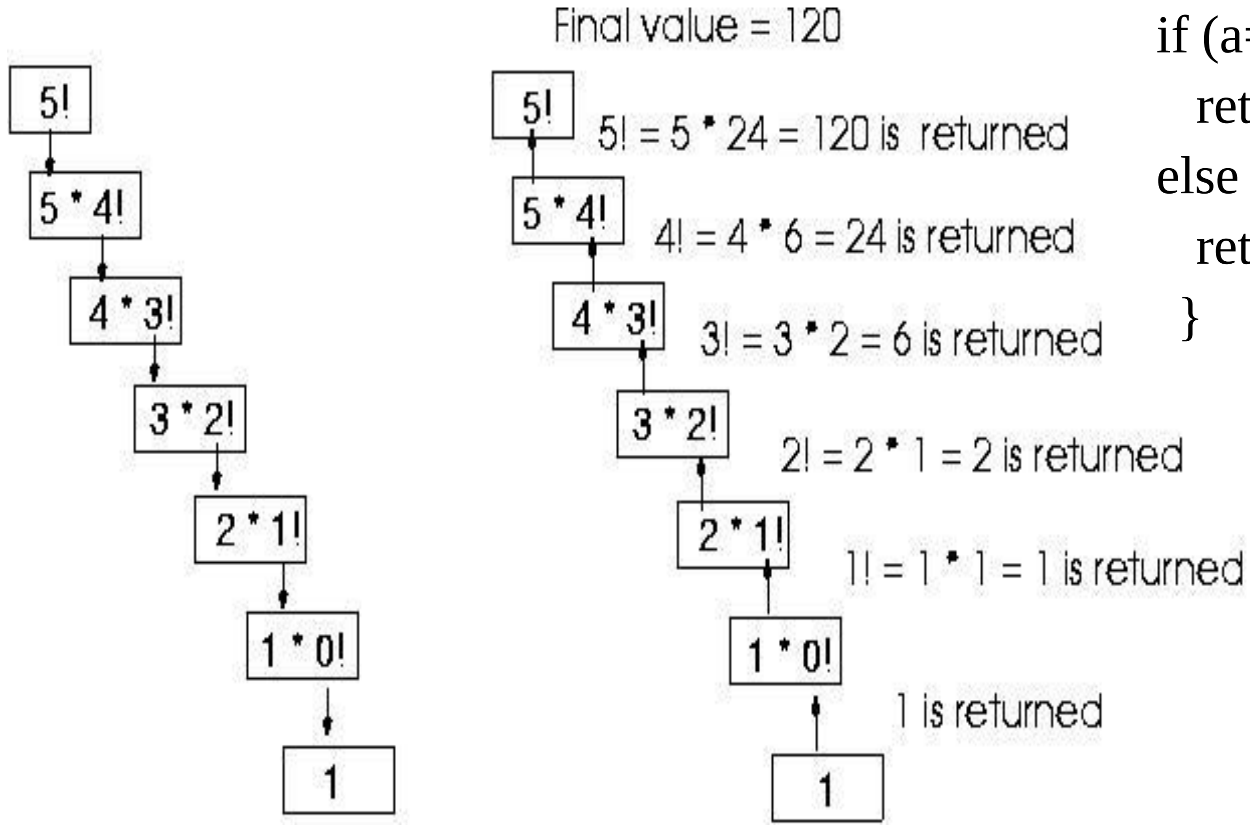
How Recursion Works

- To do this we use a stack.
- Before a function is called, all relevant data is stored in a ***stackframe***.
- This stackframe is then pushed onto the system stack.
- After the called function is finished, it simply pops the system stack to return to the original state.

How Recursion Works

- By using a stack, we can have functions call other functions which can call other functions, etc.
- Because the stack is a first-in, last-out data structure, as the stackframes are popped, the data comes out in the correct order.

Tracing the example



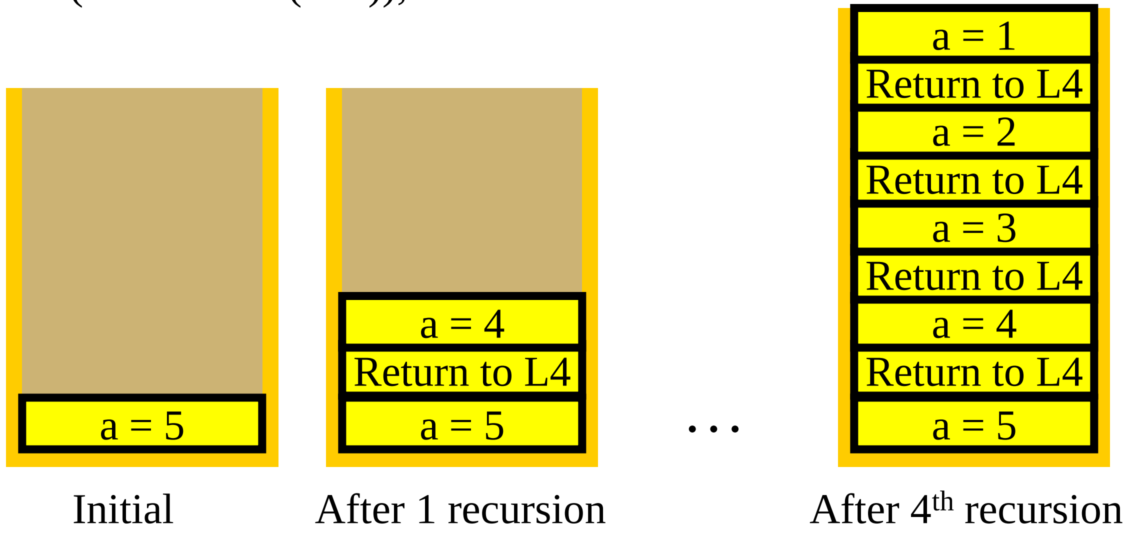
Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1! = 1 * 1 = 1 is returned
1 is returned

```
int factorial(int a)
{
  if (a==0)
     return(1);
  else
     return(a * factorial( a-1));
   }
```

RECURSION !

Watching the Stack

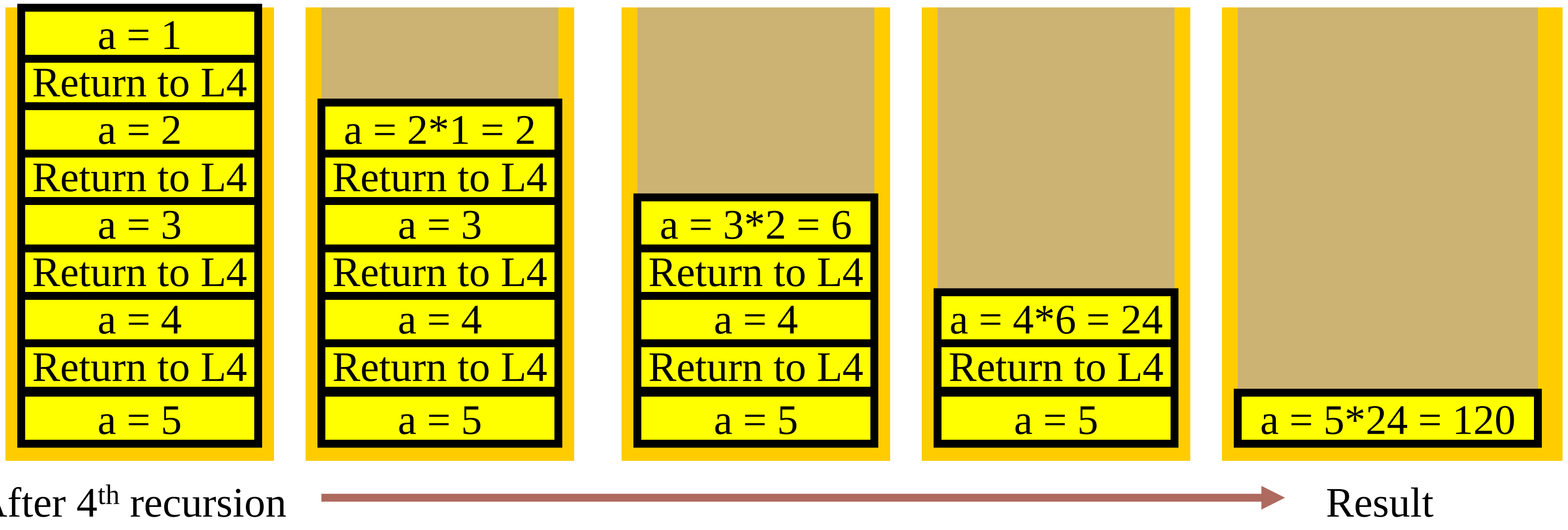
```
public int factorial(int a){
        if (a==1)
          return(1);
        else
          return(a * factorial( a-1));
}
```



Every call to the method creates a new set of local variables !

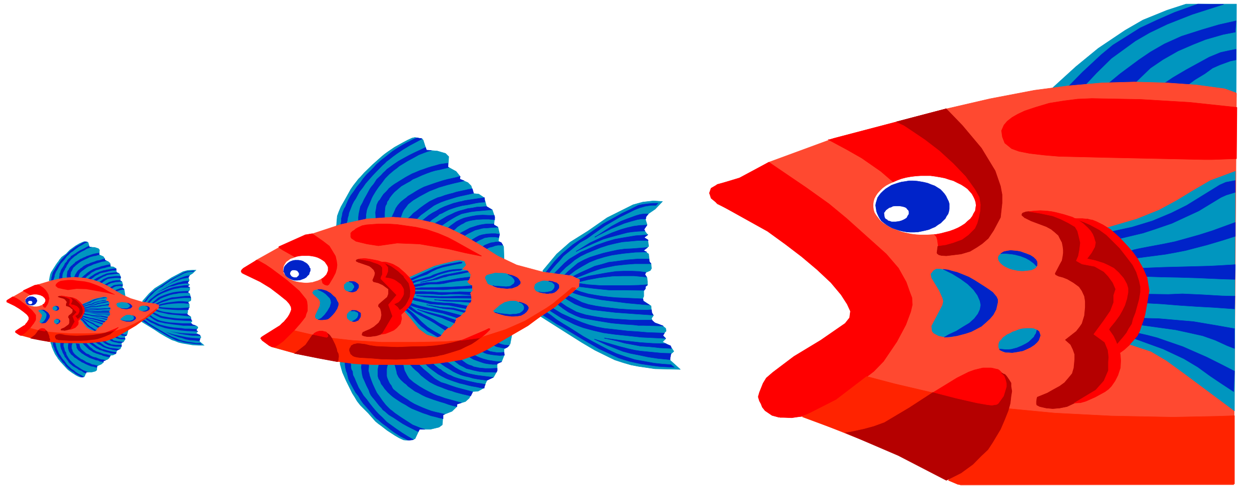
Watching the Stack

```
public int factorial(int a){
            if (a==1)
              return(1);
            else
              return(a * factorial( a-1));
}
```



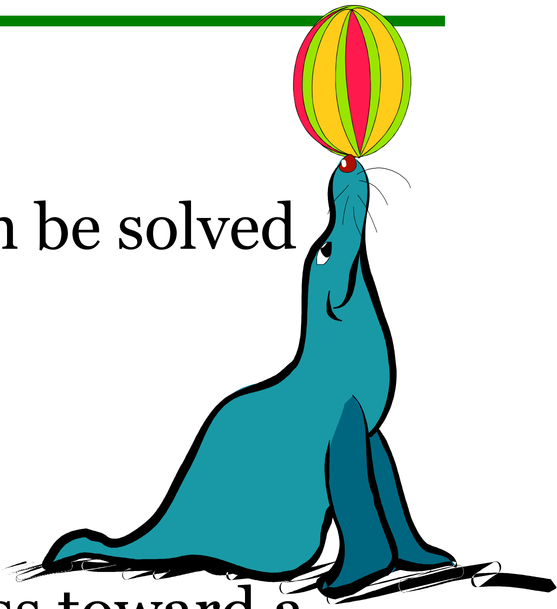
Basic Recursion

- What we see is that if we have a base case, and if our recursive calls make progress toward reaching the base case, then eventually we terminate. We thus have our first two fundamental rules of recursion:



Basic Recursion

- 1. Base cases:
  - Always have at least one case that can be solved without using recursion.
- 2. Make progress:
  - Any recursive call must make progress toward a base case.



Recursive methods

- Recursive methods either use *direct* or *indirect* recursion
- Direct recursion is when a method has a call to itself inside the body of the method
- Indirect recursion is when a method m_1 calls another method m_2 which calls other methods up to m_n, which calls m_1

Limitations of Recursion

- Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems.
- Recursive solutions can be easier to understand and to describe than iterative solutions.

Main disadvantage of programming recursively

- The main disadvantage of programming recursively is that, while it makes it easier to write simple and elegant programs, it also makes it easier to write inefficient ones.
- when we use recursion to solve problems we are interested exclusively with correctness, and not at all with efficiency. Consequently, our simple, elegant recursive algorithms may be inherently inefficient.

Limitations of Recursion

- By using recursion, you can often write simple, short implementations of your solution.
- However, just because an algorithm *can* be implemented in a recursive manner doesn't mean that it *should* be implemented in a recursive manner.

Limitations of Recursion

- Recursion works the best when the algorithm and/or data structure that is used naturally supports recursion.
- One such data structure is the tree (more to come).
- One such algorithm is the binary search algorithm that we discussed earlier in the course.

Limitations of Recursion

- Recursive solutions may involve extensive overhead because they use calls.
- When a call is made, it takes time to build a stackframe and push it onto the system stack.
- Conversely, when a return is executed, the stackframe must be popped from the stack and the local variables reset to their previous values – this also takes time.

Limitations of Recursion

- In general, recursive algorithms run slower than their iterative counterparts.
- Also, every time we make a call, we must use some of the memory resources to make room for the stackframe.

Limitations of Recursion

- Therefore, if the recursion is deep, say, factorial(1000), we may run out of memory.
- Because of this, it is usually best to develop iterative algorithms when we are working with large numbers.

Application

- One application of recursion is reversing a list.
- Before we implemented this function using a stack.
- Now, we will implement the same function using recursive techniques.