CS10003:
Programming & Data Structures

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

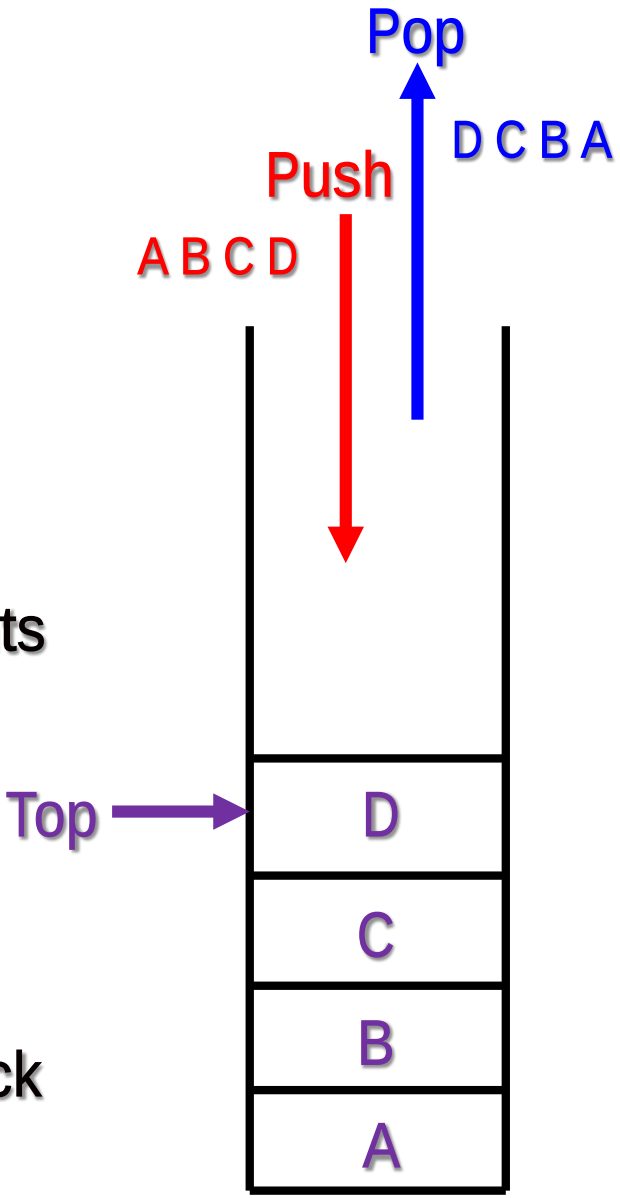
Autumn 2020

Stacks

Stacks and Basic Operations

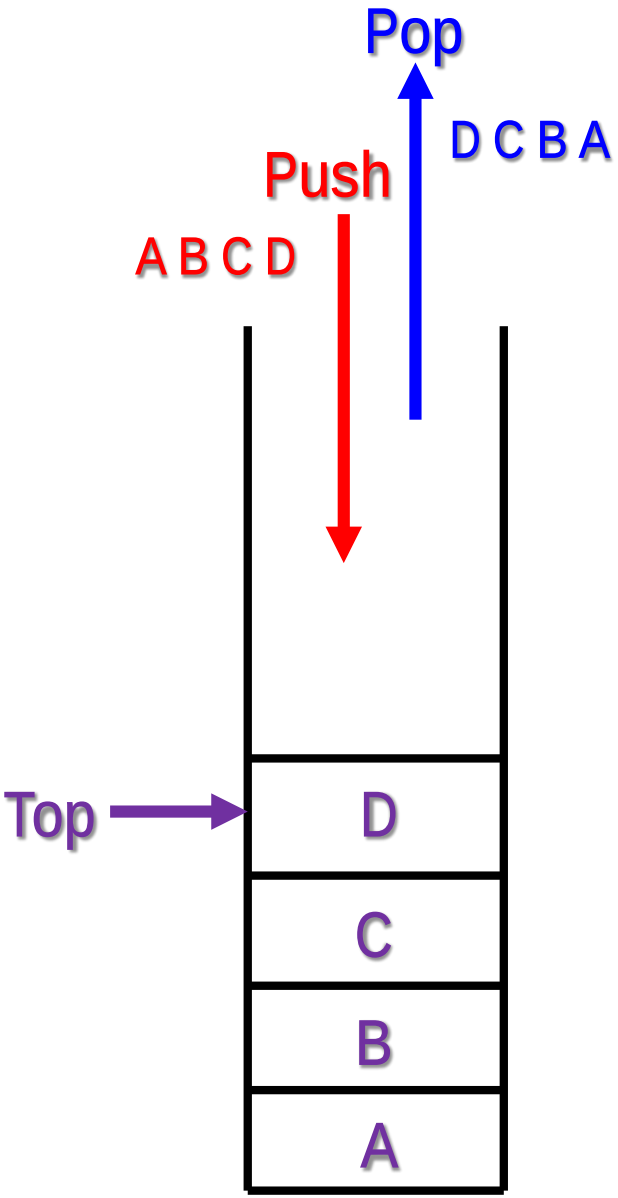
- Property:
  - Last-In First-Out (LIFO) Data Structure

- Typical Operations:
  - isEmpty: determines if the stack has no elements
  - isFull: determines if the stack is full (in case of a bounded sized stack)
  - top: returns the top element in the stack
  - push: inserts an element into the stack
  - pop: removes the top element from the stack



Stacks and Basic Operations

- Implementation Aspects:
  - Using Array
    - Pre-declared size of elements
  - Using Linked List
    - top element is the head of the list
    - push is like inserting at the front of the list
    - pop is like deleting from the front of the list



Basic Operations over Stacks

Initialization:

```
typedef struct stkArr{
   int data[MAX];
   int top;
} stack;

stack *s;
s->top = -1;
```

Emptiness Check:

```
int isEmpty (stack *s){
  if(s->top == -1)
    return 1;
  else return 0;
}
```

Overflow Check:

```
int isFull (stack *s){
  if (s->top >= MAX-1)
    return 1;
  else return 0;
}
```

Seek Top Element:

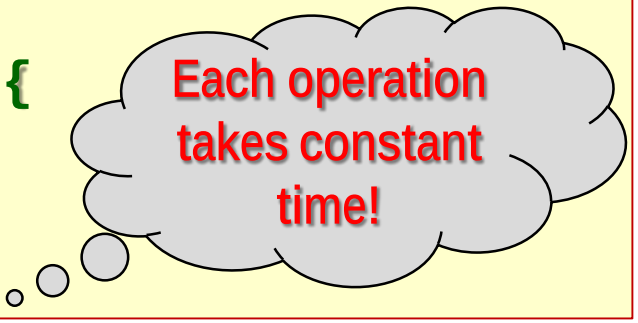
```
int top(stack *s){
  if(!isEmpty(s))
    return s->data[s->top];
}
```

Push Element:

```
void push(stack *s, int elm){
  if(!isFull(s))
    s->data[++(s->top)] = elm;
}
```

Pop Element:

```
void pop(stack *s){
  if(!isEmpty(s))
    --(s->top);
}
```



- Array based Implementation uses pre-defined fixed sized (MAX) stack, *whereas*
- Linked-List requires little higher space (to keep extra pointer to next node) for storing each element

Applications: Parenthesis Matching Problem

- If Only '(' and ')' are allowed
  - Can be found without using Stacks
    - Keep a variable count and Increment or Decrement count when '(' or ')' is encountered (ignore all other characters)
    - Parenthesis Unbalanced if –
      - count becomes less than zero at any intermediate point
      - count is non-zero at end (Balanced only when count=0 at end)
- Example:

```
( ( ( ) ( ) ) ( ( ( ) ) ( ) ) )
1 2 3 2 3 2 1 2 3 4 3 2 3 2 1 0

( ( ) ( )       ( ( ( ) ) ) )   ( ( )
1 2 1 2 1       1 2 3 2 1 0 -1  0 1 0
```

Parenthesis Matching Problem: *Revisited*

- If '( )', '{ }' and '[ ]' all are allowed
  - Three separate count variables (for each type of parenthesis) will NOT do
    - Check this Example: **{2*(3+5}-[8-2)/3]**
    - Wrongly report Balanced if the above procedure is followed
  - Solution: *(use Stack)*
    - Push every opening parenthesis '(', '{' or '[' into a stack
    - For every closing parenthesis ')', '}' or ']', pop the top element of the stack and match for '(', '{' or '[', respectively
    - If any mismatch, flag Unbalanced
    - Otherwise report Balanced at end

Applications: Arithmetic Expression Evaluation

- Arithmetic Expressions

Standard Format

  - Infix (operator within operands):

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

  - Prefix (operator before operands):

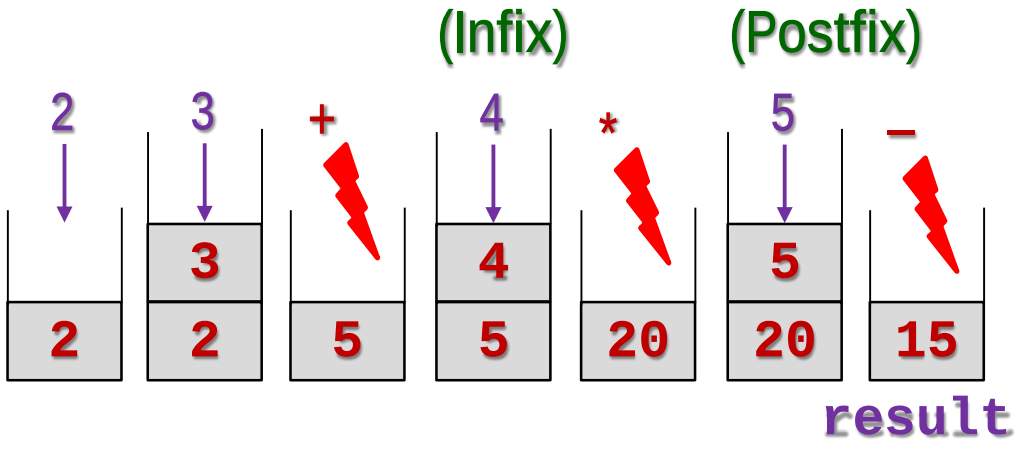
    `- * c + a b / + d e f`

  - Postfix (operator after operands):

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

- Example:

  `expr: (2+3)*4-5` → `23+4*5-`



Evaluate Postfix Expression:

```
void postfixEval(Expression expr){
  initialize stack S;
  x ← get next token from expr;
  while(x is NOT end of expr){
    if(x is operand) push(S, x);
    else{ // x is operator
      if(!empty(S)){
        x1 ← top(S); pop(S);
      }
      else exit with error;
      if(!empty(S)){
        x2 ← top(S); pop(S);
      }
      else exit with error;
      result ← x1 <x> x2;
      push(S, result);
    }
    x ← get next token from expr;
  }
  output(top(S)); pop(S);
  if(!empty(S)) exit with error;
}
```

Applications: Expression Conversion

- Infix to Postfix Conversion
  - Operands in same order
  - Operators are rearranged
  - Operators after operands
  - Brackets are deleted

Operators are popped out from stack whenever ISP ≥ ICP condition holds!

Symbol	In-Stack Priority (ISP)	In Coming Priority (ICP)
)		
^	3	4
*/	2	2
+ –	1	1
(	0	4

Evaluate Postfix Expression:

```
void infix_postfix(Expression expr){
  initialize stack S; push(S,'#');
  x ← get next token from expr;
  while(x is NOT end of expr){
    if(x is operand) output(x);
    else if(x is ')'){
      while((y=pop(S)) != '('){
        output(y);
    else{
      while(ISP(y=pop(S)) >= ICP(x))
        output(y);
    }
    x ← get next token from expr;
  }
  while((y=pop(S)) != '#')
    output(y);
}
```

Example Expressions

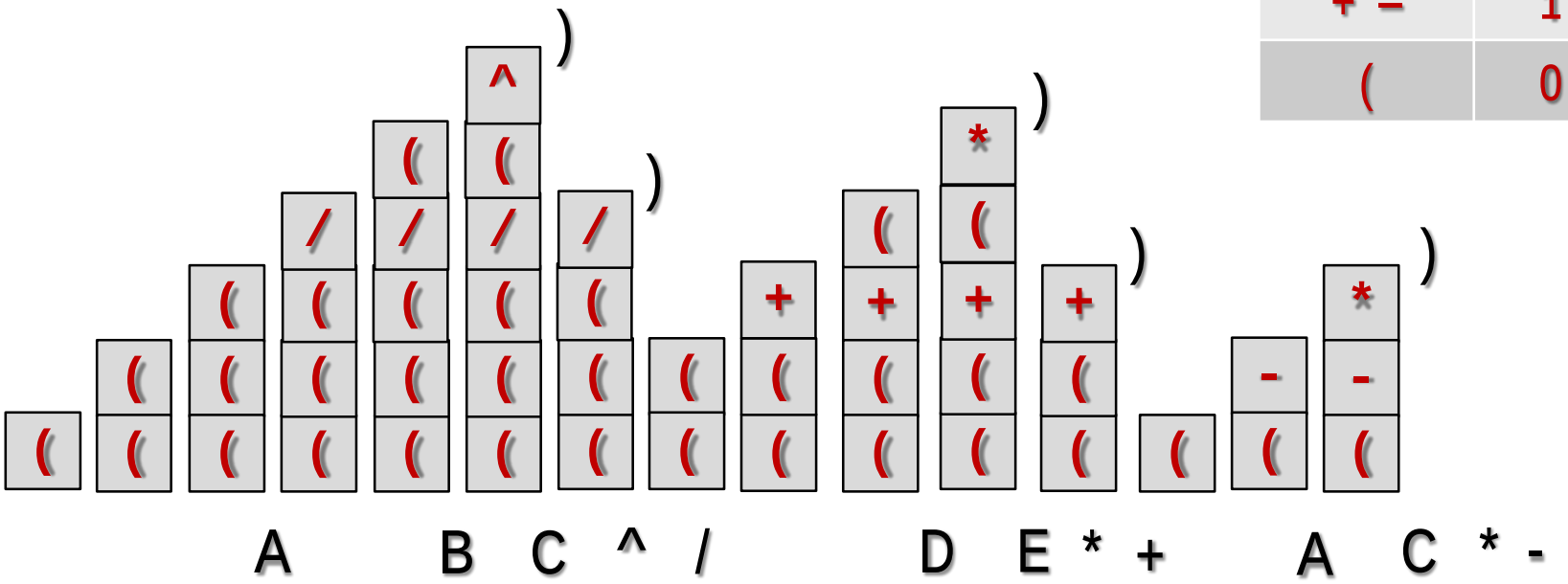
Infix: **(((A/(B^C))+(D*E))–A*C)**

Postfix: **A B C ^ / D E * + A C * -**

Example: *Revisited*

Symbol	ISP	ICP
)		
^	3	4
* /	2	2
+ –	1	1
(	0	4

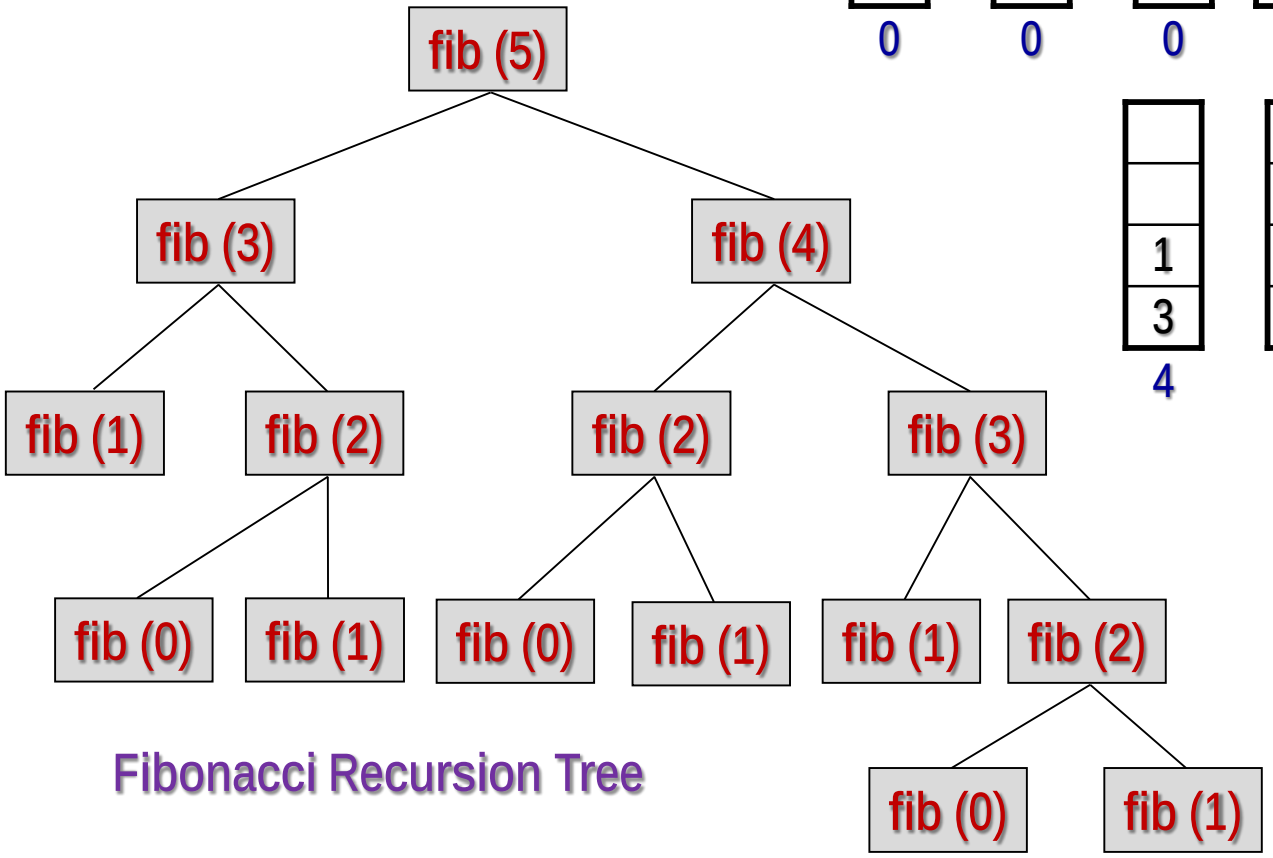
- Infix: **(((A/(B^C))+(D*E))–A*C)**



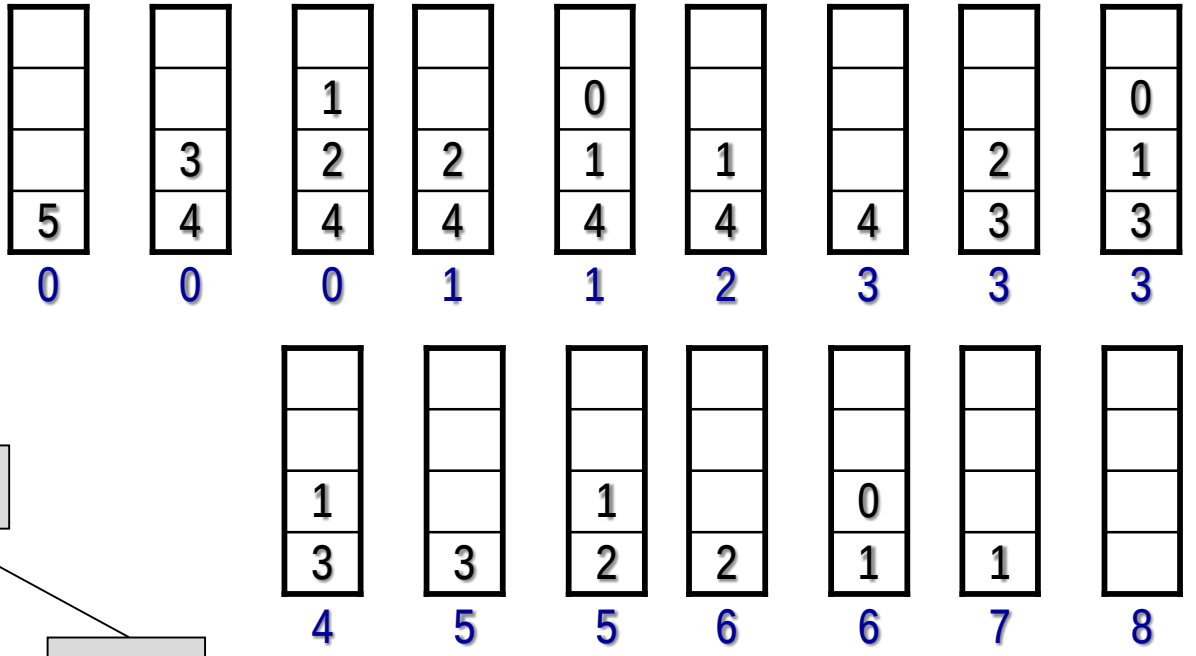
- Postfix: **A B C ^ / D E * + A C * –**

Recursions use Stacks Implicitly!

- Fibonacci Number Computation (using Recursion)
  - Recurrence: Fib(n) = Fib(n-1) + Fib(n-2), if n > 1 and Fib(0) = Fib(1) = 1



Fibonacci Recursion Tree



Fibonacci Recursion Stack

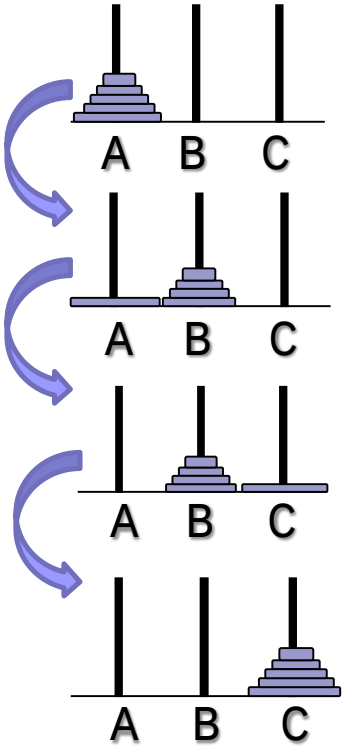
Nested Recursive Function Calls use Stacks to Store Return Addresses

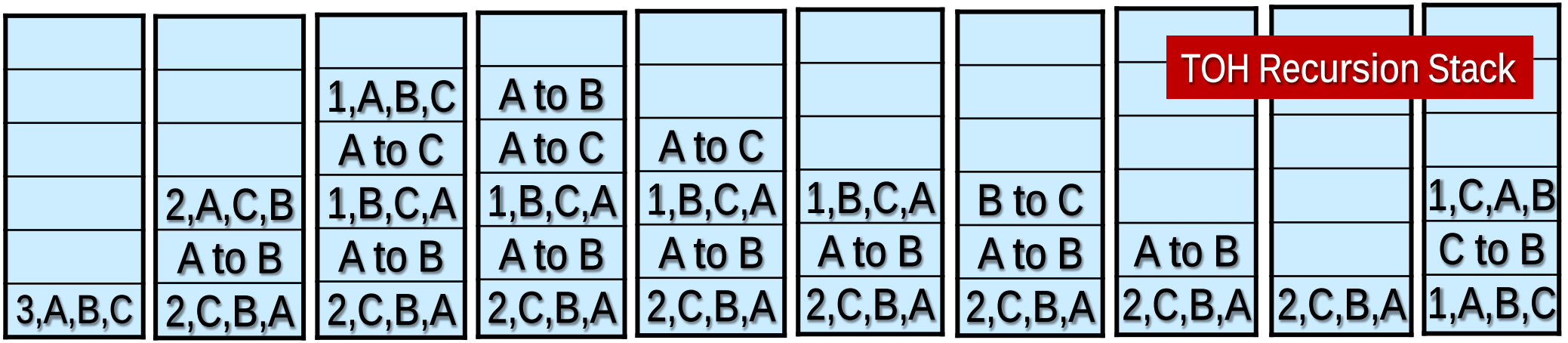
Recursions use Stacks Implicitly!

- Tower of Hanoi (TOH)

```
void towers (int n, char from, char to, char aux)
{
      /* Base Condition */
       if (n==1) {

         printf ("Disk 1 : %c -> %c \n", from, to);
         return ;
       }
      /* Recursive Condition */
       towers (n-1, from, aux, to);
       printf ("Disk %d : %c -> %c\n", n, from, to);
       towers (n-1, aux, to, from);
}
```





Thank You!