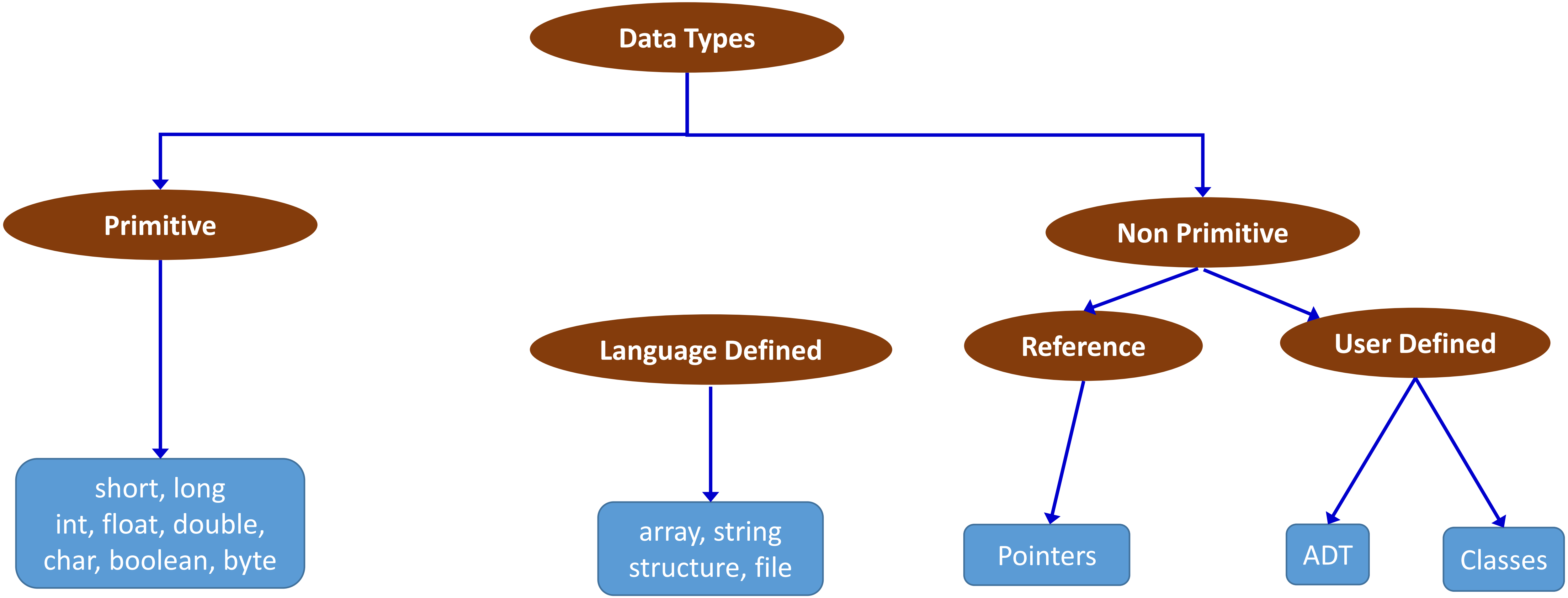
Abstract Data Types STACK

PROF. NAVRATI SAXENA

What are Data Types?

- Data Types
  - A set of values
  - Operations that can be performed on those values.
  - Example: data type integer can take values -32768 ~ 32767 with common Arithmetic operations
  - Some popular data types
    - int (represents integer)
    - float
    - double
    - char
    - boolean
    - byte

Hierarchy of Data Types



Abstract Data Types (ADT)

- A special kind of datatype, whose behavior is defined by a set of values and set of operations

- Interface in ADT defines:
  - The type of data that can be stored
  - The actual value of the data and
  - The operations that can be performed on the data

- Implementation in ADT defines:
  - Data organization
  - Algorithms for these operations

ADT Benefits

- Code is easier to understand
- Efficient code can be programmed by changing the ADT without changing the whole program
- ADTs can be reused
- They facilitate unit testing

Types of ADTs

Linear ADTs

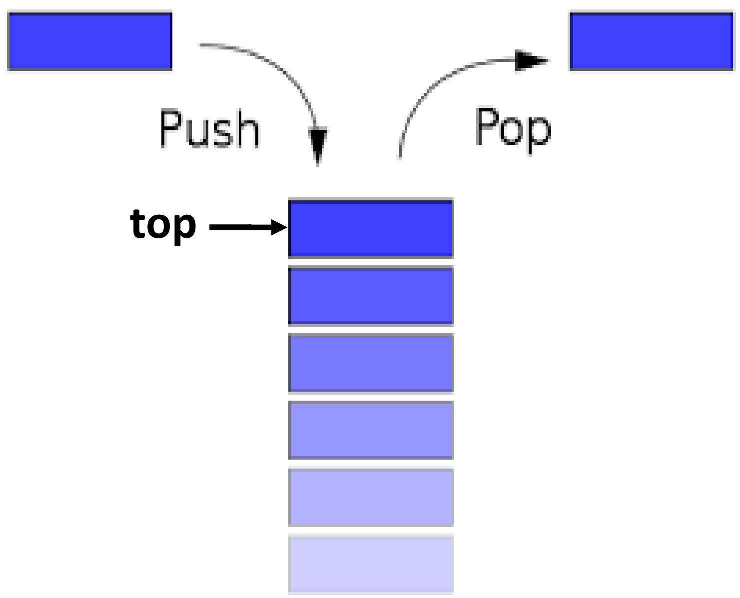
- Restricted Lists
  - Stack
  - Queue
- General Lists
  - Arrays
  - Linked List
  - Doubly Linked List
  - Circular Linked List

Non-Linear ADTs

- Trees
- Graphs
- Hash Tables

STACK

- Last-In-First-Out (LIFO) or FILO (First In Last Out).
  - Data entered last will be the first to get accessed
- Addition/removal of items always takes place at same end (Top)
- Base represents bottom and contains item has been in stack the longest
- Most recently added to be removed first – LIFO / FILO
- Top: newer items

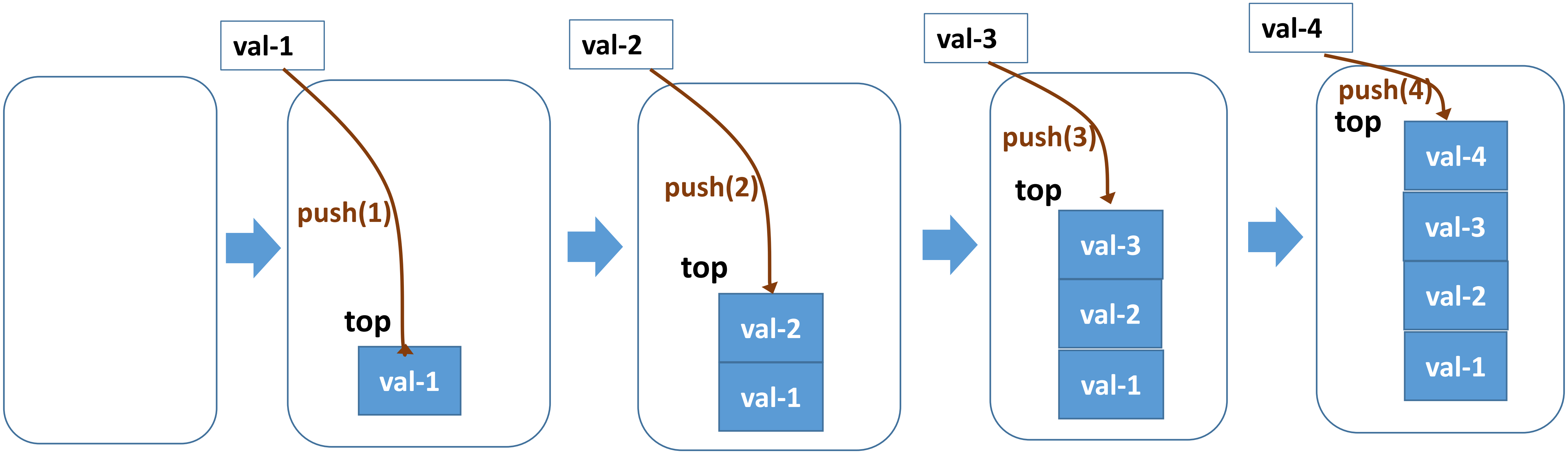


Operations of Stack (1/3)

- **PUSH**: Adding elements to the Top of Stack
- **POP**: Deleting elements and accessing elements from the Top of Stack
- **TOP**: Points to to the current newest element of the stack

Operations of Stack (2/3)

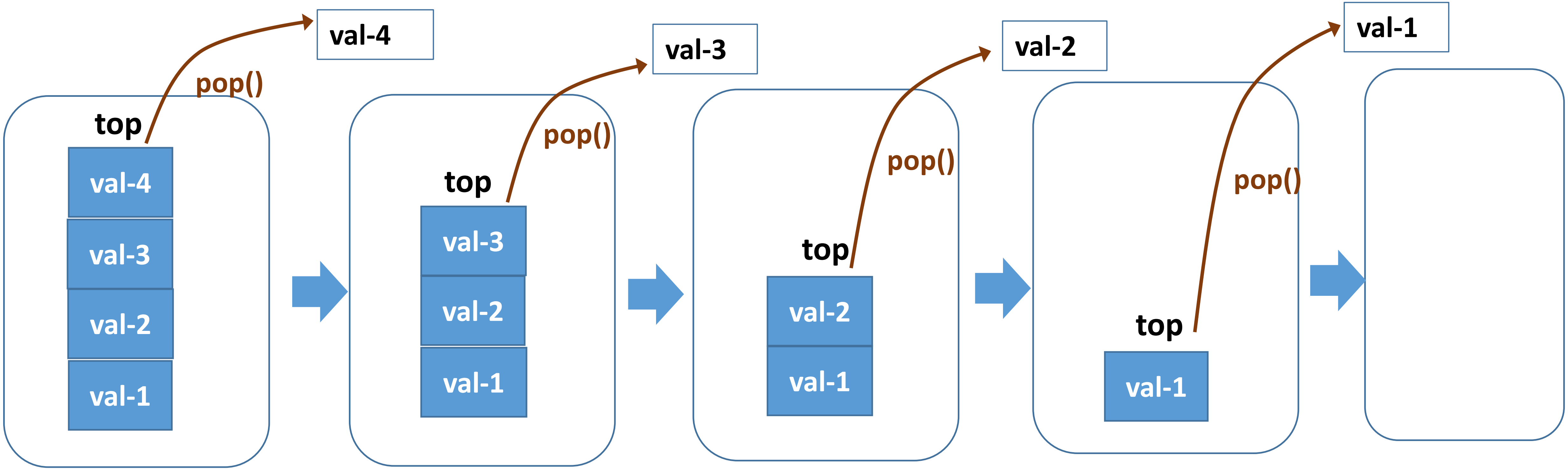
PUSH OPERATION IN A STACK



void push(x,S): Insert element x into the stack S

Operations of Stack (3/3)

POP OPERATION IN A STACK

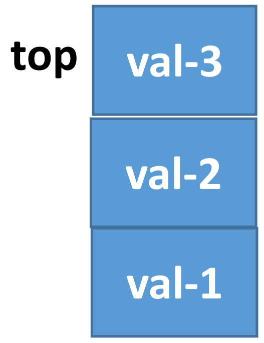


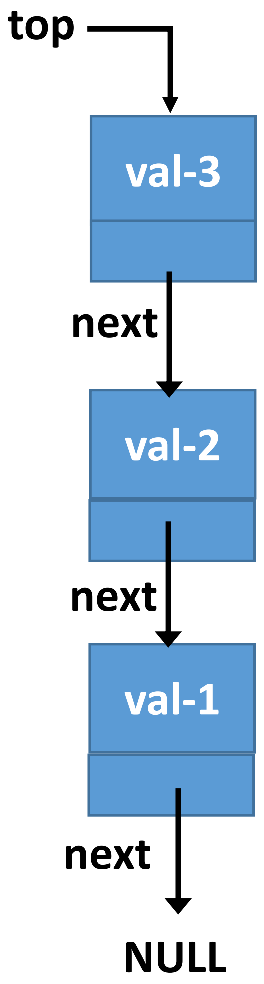
datatype pop(S): Return the last element inserted into the stack S

boolean IsStackEmpty(S): Return True if the stack S is empty

Representation of Stack in Memory

- Two ways of stack implementation:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)





Stack Conditions

STACK OVERFLOW

Necessary to check if the stack is full

Attempt to add an item to a full stack

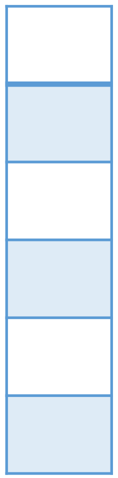
-> Stack Overflow Error

STACK UNDERFLOW

Necessary to check if the stack is empty

Attempt to remove an item from an empty stack

-> Stack Underflow Error



Top = NULL

PUSH OPERATION

The new element is added at the topmost position of the stack.

PUSH (STACK, TOP, ITEM)

STACK is the array with N elements; TOP is the pointer to the top of the element of the array; ITEM to be inserted.

Step 1: if TOP = N then [Check Overflow]

PRINT " STACK is Full or Overflow Error"

Exit

[End if]

Step 2: TOP = TOP + 1 *[Increment the TOP]*

Step 3: STACK[TOP] = ITEM *[Insert the ITEM]*

Step 4: Return

POP OPERATION

POP (STACK, TOP, ITEM)

STACK is the array with N elements; TOP is the pointer to the top of the element of the array

Step 1: if TOP = NULL then [Check Underflow]

PRINT “ STACK is Empty or Underflow”

Exit [End if]

Step 2: ITEM = STACK[TOP] *[copy the TOP Element]*

Step 3: TOP = TOP - 1 *[Decrement the TOP]*

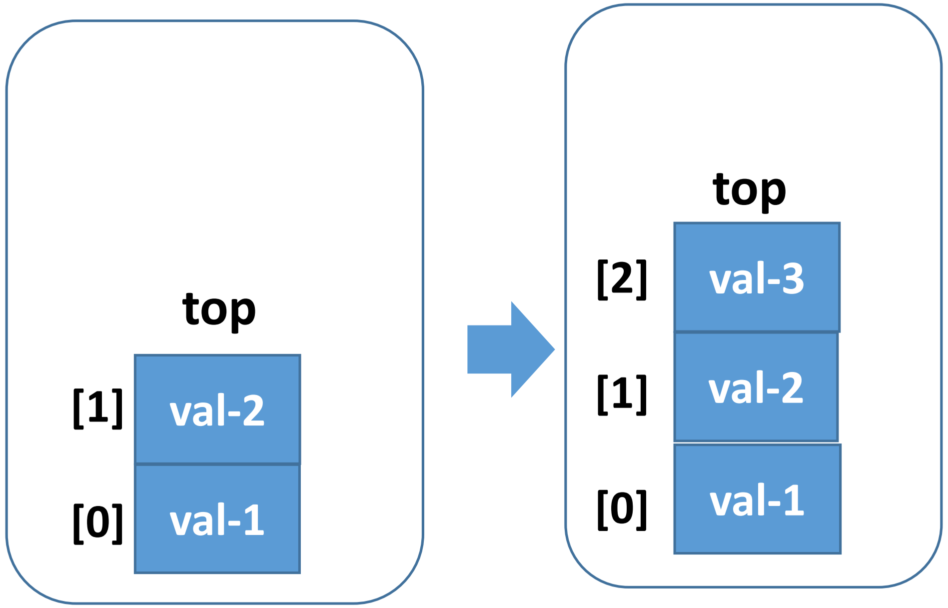
Step 4: Return

STACK: ARRAY IMPLEMENTATION (1/2)

EACH OPERATION TAKES O(1) TIME

createStack(S):
Define an array S for some fixed size N
top ← -1

push(x, S):
if top = (N-1)
        ERROR
else
        top ← top+1
        S[top] ← x

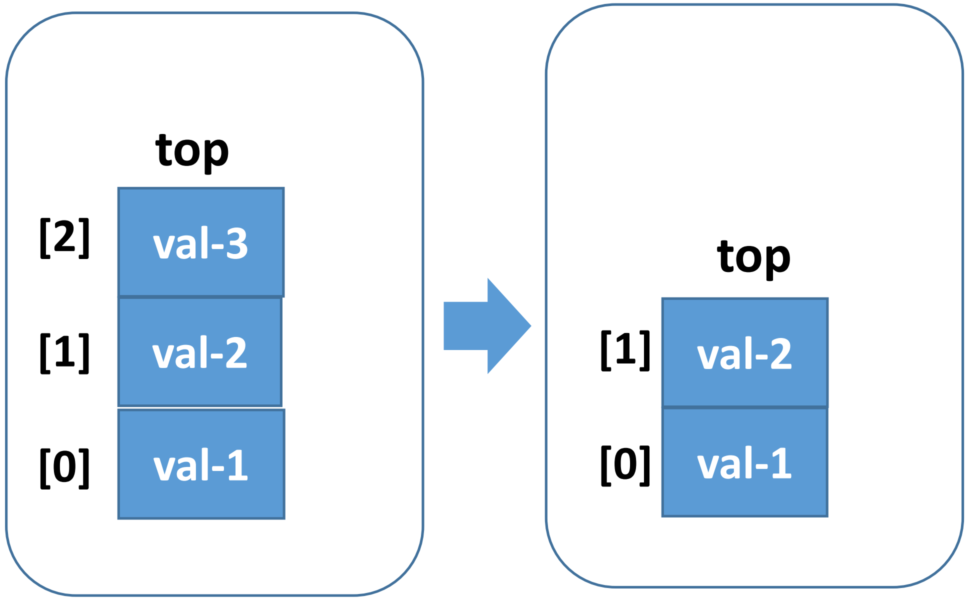


STACK: ARRAY IMPLEMENTATION (2/2)

EACH OPERATION TAKES O(1) TIME

isStackEmpty(S):
if (top < 0)
    return TRUE
else
    return FALSE

pop(S):
if (isStackEmpty())
    ERROR
else
    item ← S[top]
    top ← top -1
    return (top)



EXAMPLE OPERATIONS IN STACK

createStck()

push(A, S)

push(B, S)

push(C, S)

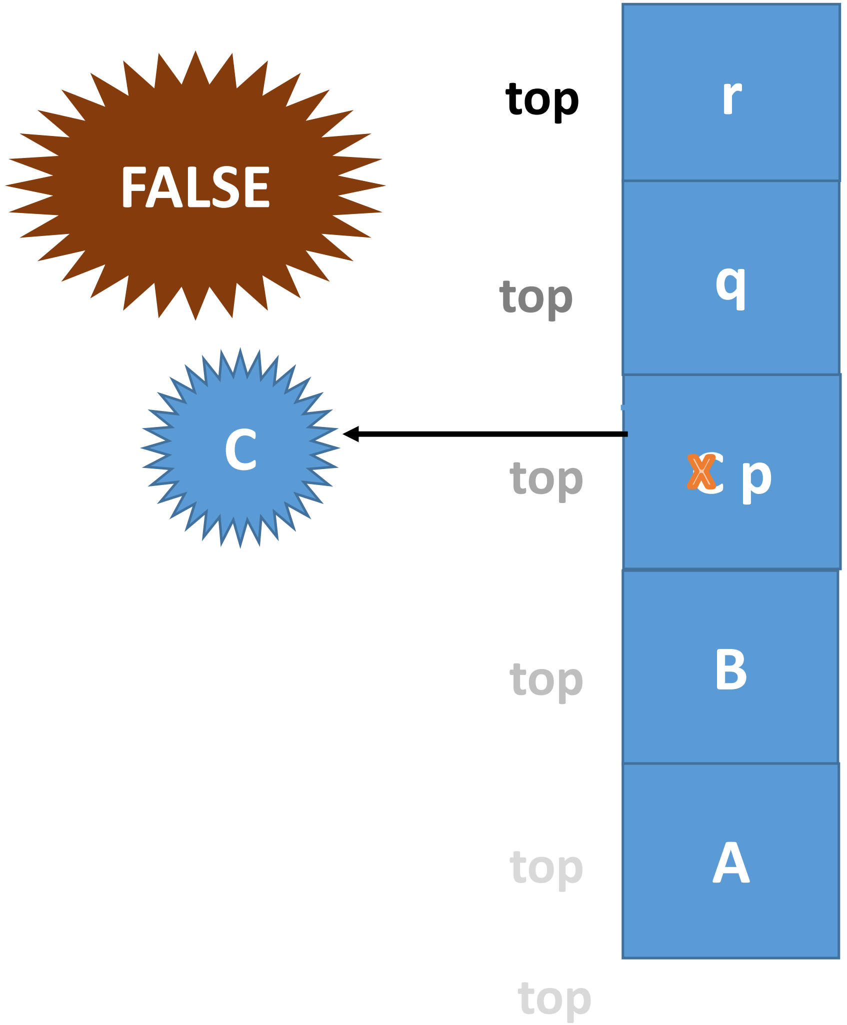
pop (S)

isStackEmpty(S)

push(p, S)

push(q, S)

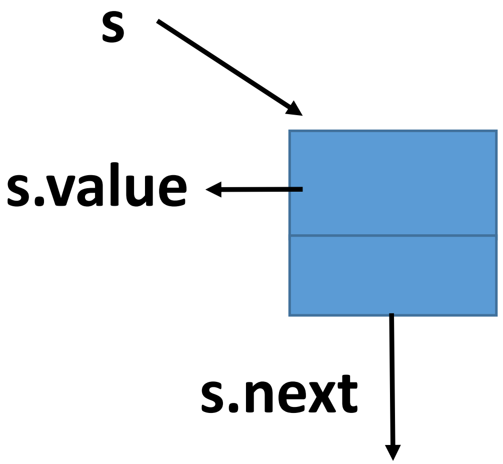
push (r, S)



STACK: POINTER IMPLEMENTATION (1/3)

- Pointers facilitate dynamic implementation

- We would need:
  - A pointer which points to the first element of the stack
  - Each element of the stack, called as Node
  - Each Node has:
    - The data/element (s.element)
    - Pointer to the next node (s.next)

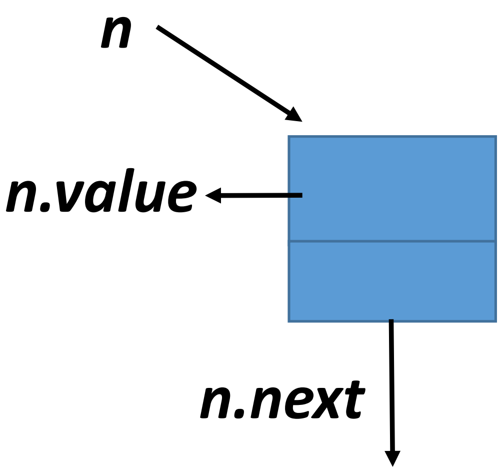


STACK: POINTER IMPLEMENTATION (2/3)

EACH OPERATION TAKES O(1) TIME

createStack(S):
Define a pointer top
top ← NULL

push(x, S):
create a new node ***n***
n.element ← x
n.next ← top
top ← n



STACK: POINTER IMPLEMENTATION (3/3)

EACH OPERATION TAKES O(1) TIME

isStackEmpty(S):

if top = NULL

    return TRUE

else

    return FALSE

pop(S):

If isStackEmpty(S)

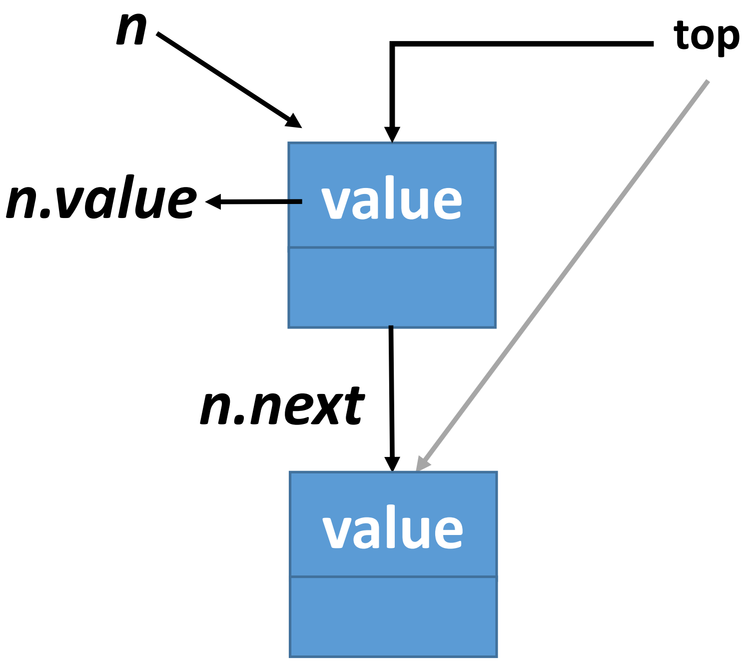
    ERROR

else

    element ← top.element

    top ← top.next

    return (element)



APPLICATIONS OF STACK

Applications using Stack (1/2)

- Function Call Mechanisms and Recursive Programming

- Reversing words: push the word to stack - letter by letter; pop letters from the stack

- Expression Conversion
  - In-fix to Post-fix
  - In-fix to Pre-fix

- Backtracking
  - Undo mechanisms in word editors

Applications using Stack (2/2)

- Language Processing: Compiler' syntax check for matching braces
- Conversion of decimal number to binary
- To solve tower of Hanoi
- Check if delimiters are matched
  - Matching of opening and closing symbols: {,},[,],(,)
  - Check: {{a}[b]{[{c}](d(e)f)}((g))} and ({[a}b(c)])
- Wearing/Removing Bangles

Arithmetic Expression (1/2)

- An expression is a combination of operands and operators that after evaluation results in a single value.

- Operand consists of constants and variables.

- Operators consists of {, +, -, *, /,),] etc.

- Expression can be
  - **Infix Expression**
  - **Postfix**
  - **Prefix Expression**

Arithmetic Expression (2/2)

Infix Expression: If an operator is in between two operands, it is called infix expression.

- Example: a + b, where a and b are operands and + is an operator.

Postfix Expression: If an operator follows the two operands, it is called postfix expression.

- Example: ab +

Prefix Expression: an operator precedes the two operands, it is called prefix expression.

- Example: +ab

Evaluating Expressions

Infix	Prefix	Postfix
$x + y$	$+xy$	$xy +$
$x + y * z$	$+x * yz$	$xyz * +$
$(x + y) * (z - p)$	$* +xy - zp$	$xy + zp -*$
$y * y - 4 * x * z$		$yy * 4xz ** -$
$40 - 3 * 5 + 1$		

Infix Expressions Evaluation

$$\boldsymbol{Exp}: (((\mathbf{4} * \mathbf{5}) - (\mathbf{1} * \mathbf{6}))/(\mathbf{10} - \mathbf{3}))$$

Input symbol	Stack (top to bottom)	Operation
(		
(		
(		
4	4	
*	4*	
5	4*5	
)	20	
-	20-	
(	20-	4*5=20 and push
1	20-1	
*	20-1*	
6	20-1*6	
)	20-6	1*6=6 and push
)	14	20-2=18 and push
/	14/	
(	14/	
10	14/10	
-	14/10-	
3	14/10-3	
)	14/7	10-3=7 and push
)	2	14/7=2 and push
New line	Empty	Pop and print

Infix Expressions Evaluation

- Five types of input characters
  - Opening brackets
  - Numbers
  - Operators
  - Closing brackets
  - New line characters
- Data structure requirements: A character stack

Algorithm: Infix Expressions Evaluation

1. Read one input character at a time

2. switch (input)

    case (Opening bracket) : Go to step (1)
    case (Number) : Push into stack and then Go to step (1)
    case (Operator) : Push into stack and then Go to step (1)
    case (Closing bracket) : op2 = pop (s)
        op = pop (s)
        op1 = pop (s)
        result = computer (op1 op op2)
        Push result into stack and Go to step (1)
    case (new line character) : Pop from stack and print the answer

3. STOP

Converting Infix to Postfix Expressions (1/2)

1. Create an empty stack and an empty postfix output string

2. Scan the infix input string left to right

3. If the current input token is an operand
   - Append it to the output string

4. If the current input token is an operator: Pop off all operators that have equal or higher precedence; Append them to the output string; Push the operator onto the stack.

Converting Infix to Postfix Expressions (2/2)

5. If the current input token is opening parenthesis '(', push it onto the stack

6. If the current input token is closing parenthesis ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('

7. If the end of the input string is found, pop all operators and append them to the output string.

Postfix Expression Evaluation

1. Scan the expression left to right
2. push values (constant) or variables (operands)
3. When an operator is found, compute the result by applying the operation to the preceding two operands (by popping two operands from the stack)
4. Push the result back to stack
5. When finished, print the stack

Postfix Expression Evaluation

1. Read one input character at a time

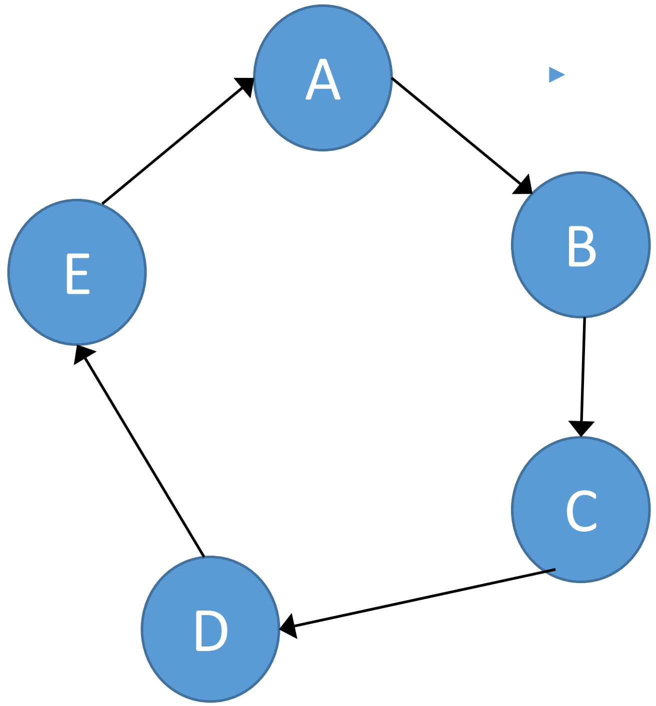
2. switch (input)
   case (Operand)
   case (Number) : Push input into stack and then Go to step (1)
   case (Operator) : op2 = pop (s)
   op1 = pop (s)
   result = computer (op1 Operator op2)
   Push result back into stack and Go to step (1)
   case (new line character) : Pop from stack and print the answer

3. STOP

Recursion – Basics

- What is Recursion?
  - A procedure calls **itself**
  - Procedure A calls procedure B, which, in turn, calls procedure A again.

- A simple picture
  - Each node represents a procedure
  - Each edge (connection) represents a procedure call
  - Recursion => Forms a **CIRCLE**
  - Example: A => B => C => D => E => A



Recursion: One Example

$$fact\ (n) = \begin{cases} 1, if\ n = 0 \\ n\ \times fact(n-1), if\ n > 0 \end{cases}$$

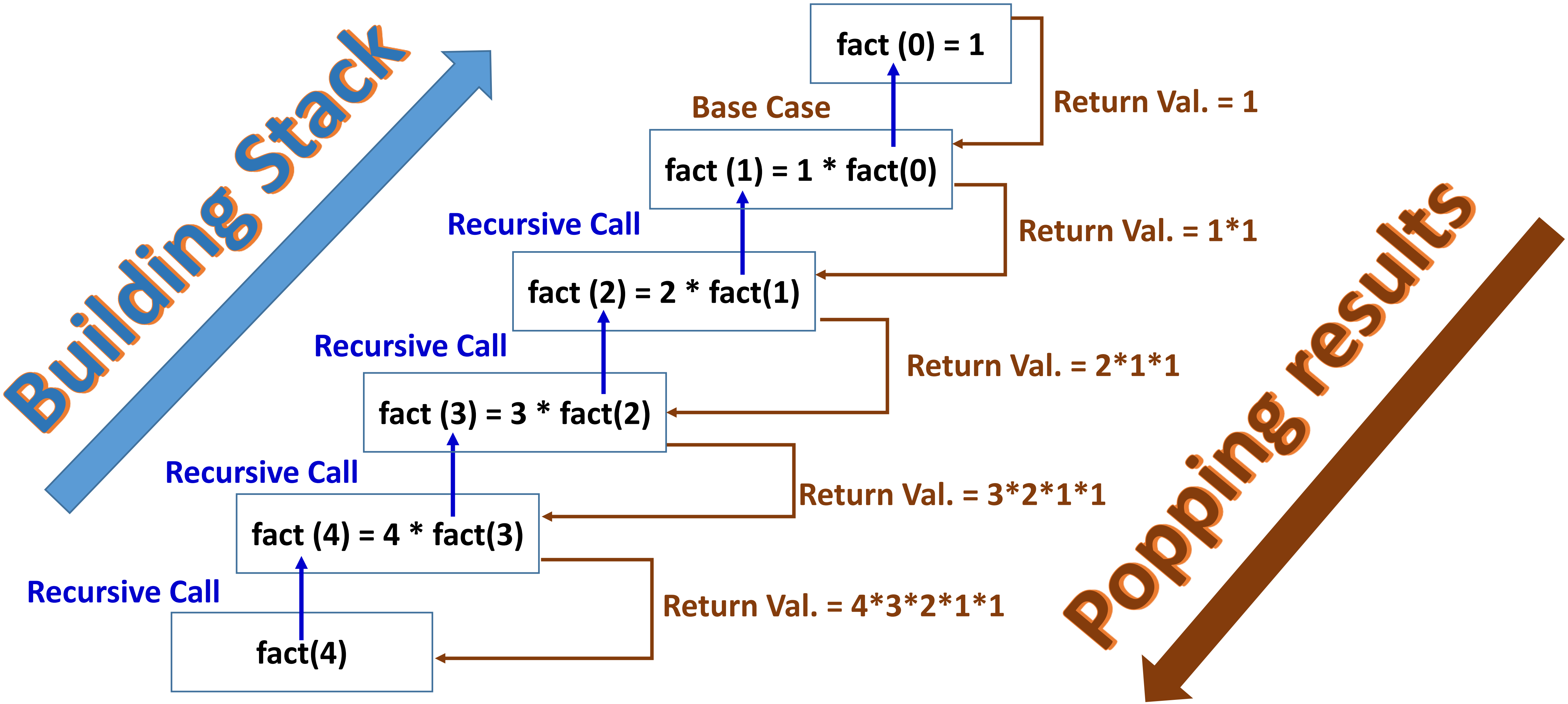
- Input: integer n such that n >= 0

- Output: [n × (n-1) × (n-2) × … × 1]

1. if n is 0, return 1
2. otherwise, return [n × factorial(n-1)]

- end factorial

How Recursion Works?



STACK: Static Implementation using C

```
#define MAX 10
typedef struct stack {

        int arr[MAX];
        int top;
} STACK;
void createStack(STACK *);
int stackEmpty( STACK *);
int stackFull ( STACK *);
void push (STACK *, int item);
int pop (STACK *);
```

```
void createStack(STACK*s){
        s->top=-1;
}
```

```
void push ( STACK *s, int item ) {
   if ( stackFull(s)) {
          printf ("\nStack is full");
          return;
}
    s->top++ ;
    s-> arr[s ->top] = item;
}
```

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

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

```
int pop( struct stack *s ) {
   int item;
   if ( stackEmpty(s)) {
        printf ("\nStack is empty");
       return(-1);
 }
    item = s ->arr[s -> top];
    s -> top--;
    return item;
}
```

STACK: Dynamic Implementation using C

```c
struct node {
      int data ;
      struct node *link ;
} ;
void createStack(struct node ** )
void push ( struct node **, int );
int pop ( struct node ** );
void delStack ( struct node ** );
Int stackEmpty( struct node **);
```

```c
void push ( struct node **top, int item ) {
      struct node *temp;
      temp = (struct node*) malloc(sizeof(struct node ));
      if ( temp == NULL )
            printf ("\nStack is full");
      temp -> data = item;
      temp -> link = *top;
      *top=temp;
}
```

```c
int stackEmpty( struct node **tos) {
      return ( *tos == NULL );
}
```

```c
void createStack(struct node ** top )
{
  *top = NULL;
}
```

```c
int pop ( struct node **top ) {
      struct node *temp;
      int item;
      if (stackEmpty(top)) {
         printf ("\nStack is empty");
         return 0 ;
      }
      temp = *top;
      item = temp -> data;
      *top = ( *top ) -> link;
      free ( temp );
      return item;
}
```

Thank You!