BITS Pilani
K K Birla Goa Campus

innovate achieve lead

5. Stacks

Dr. Swati Agarwal

Agenda

Stacks

- A linear data structure used for storing data (similar to Linked List).
- Items are inserted and deleted at one end according to LIFO (last-in-first-out) principle.
- The last element inserted is the first one to be deleted.
- **Example:** web browser activities, pile of plates.

Operations on Stack

- *push (x):* insert an element x at the top of the stack.
- *pop():* remove from the stack and return the top object on the stack[1].

Additional Operations:

- *size():* returns the number of elements in Stack.
- *isEmpty():* returns a boolean indicating if the Stack is empty
- *top():* returns the top element on the stack (without removing it)[1].

[1] returns error if the Stack is empty.

Overflow, Underflow and Stack Operations

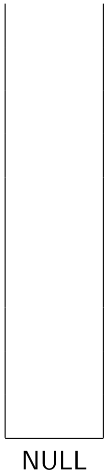
-
  ```
  function Push($S$, $x$)
      if $size() = N$ then
          $OVERFLOW$
      else
          $top \leftarrow top + 1$
          $S[top] \leftarrow x$
  ```

-
  ```
  function Pop($S$)
      if $isEmpty()$ then
          $UNDERFLOW$
      else
          $x \leftarrow S[top]$
          $S[top] \leftarrow Null$
          $top \leftarrow top - 1$
  ```

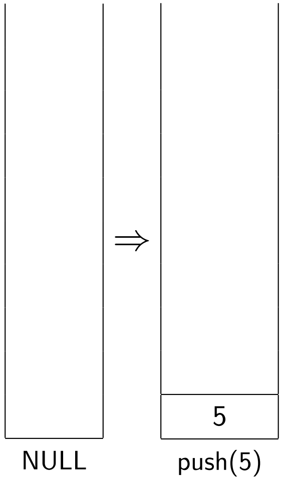
Recap

- Stack is a **Last-In-First-Out** (LIFO) data structure.
- Stack is a linear list with the restriction that insertion and deletion operations can be performed only from one end i.e. **top**.
- **ADT Operations:**
  1. push(x)
  2. pop()
- **Auxiliary Operations:**
  1. isEmpty()
  2. top()
  3. size()

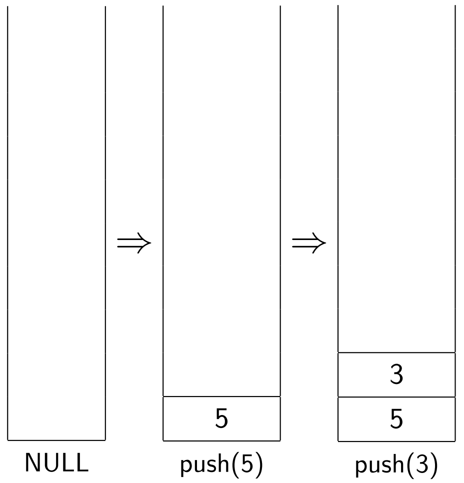
Stack Working Example: push(x)



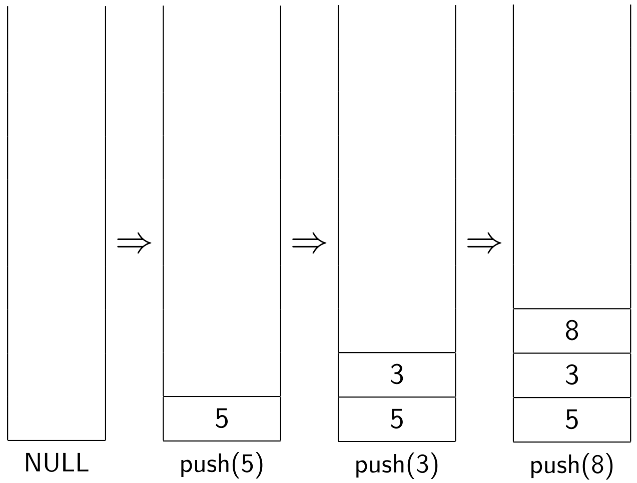
Stack Working Example: push(x)



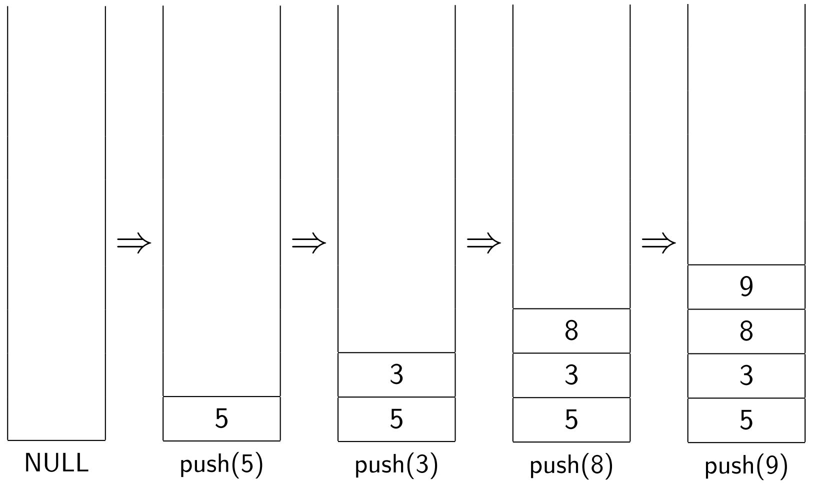
Stack Working Example: push(x)



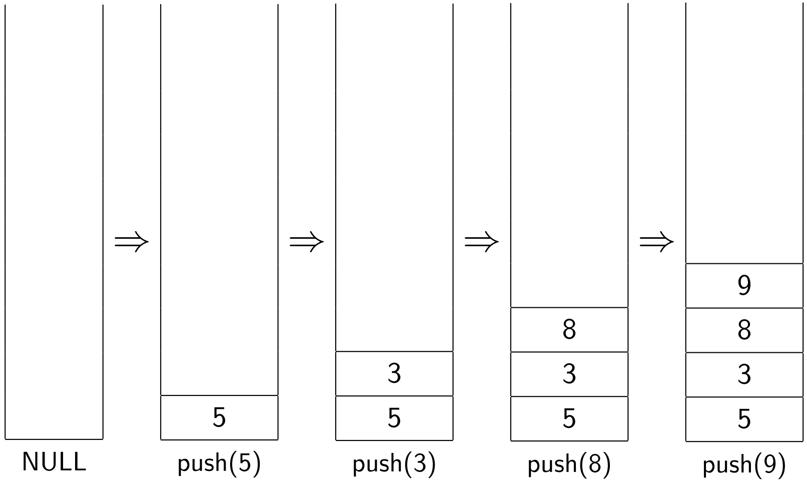
Stack Working Example: push(x)



Stack Working Example: push(x)

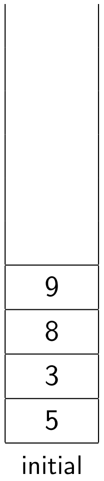


Stack Working Example: push(x)

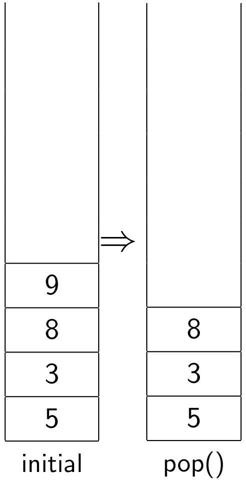


once the stack is full, the push(x) operation will throw OVERFLOW error.

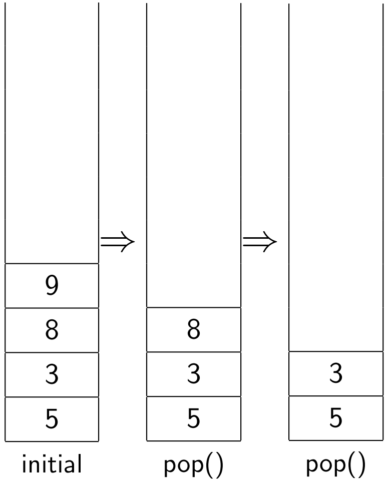
Stack Working Example: pop()



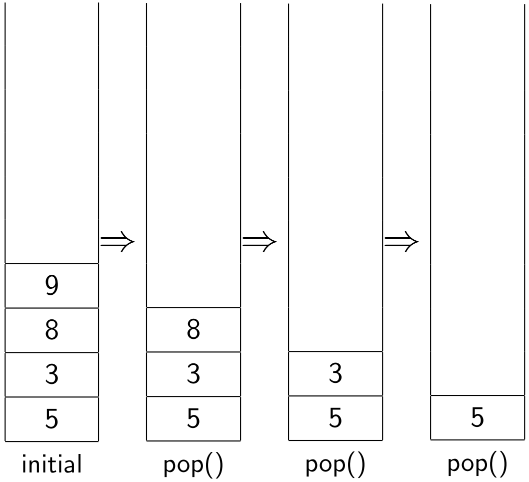
Stack Working Example: pop()



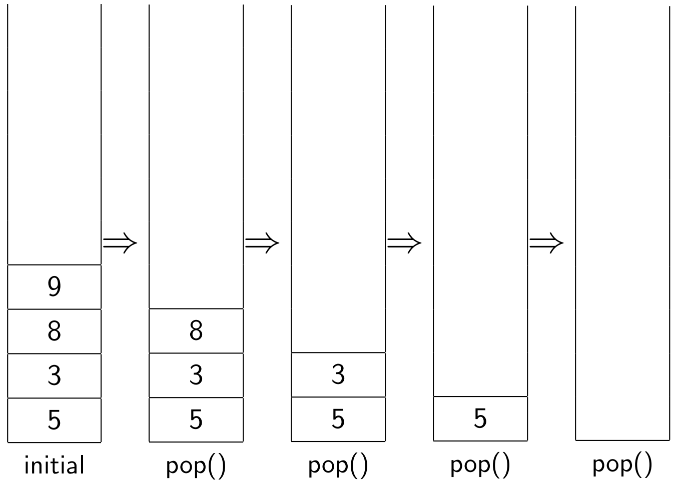
Stack Working Example: pop()



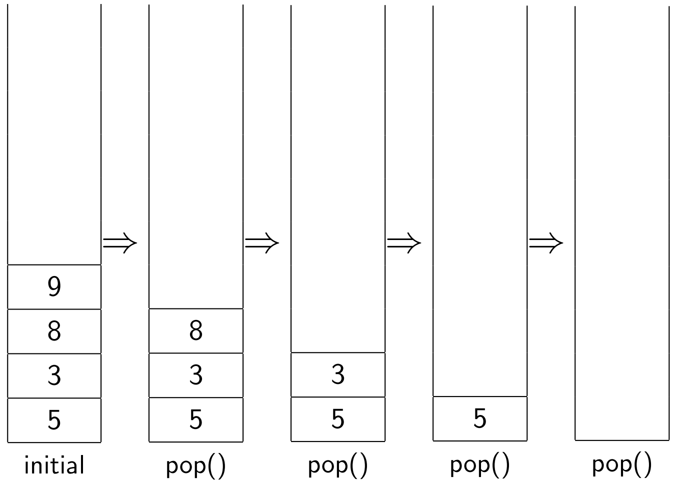
Stack Working Example: pop()



Stack Working Example: pop()



Stack Working Example: pop()



once the Stack is empty, the pop() operation will throw UNDERFLOW error.

Applications of Stack

- Function calls in a program (or recursion)
- to implement undo/redo operations
- Balanced parentheses in source code
- String reversal

Implementation of Stack

- using Arrays
- using Linked Lists

constraints to keep in mind

- insertion and deletion operations to be performed from one end.
- complexity of above operations is $\mathcal{O}(1)$

Stack implementation using Arrays

- Use array to store stack
- index is marked as top (a variable)
- is the stack is empty, $top = -1$
- push(x) only till array is not exhausted
- pop only when $top \geq 0$

```
function Push($S$, $x$)
    if $size() = N$ then
        $OVERFLOW$
    else
        $top \leftarrow top + 1$
        $S[top] \leftarrow x$
```

```
function Pop($S$)
    if $isEmpty()$ then
        $UNDERFLOW$
    else
        $x \leftarrow S[top]$
        $S[top] \leftarrow Null$
        $top \leftarrow top - 1$
```

Stack implementation using Arrays

Created an integer array A[10]=

Empty stack? top = -1

Stack implementation using Arrays

Created an integer array A[10]=

Empty stack? top = -1

- push(2); top = 0; A[ ]=

2									

Stack implementation using Arrays

Created an integer array A[10]=

Empty stack? top = -1

- push(2); top = 0; A[ ]=

2									

- push(10); top = 1; A[ ]=

2	10								

Stack implementation using Arrays

Created an integer array A[10]=

Empty stack? top = -1

- push(2); top = 0; A[ ]=

2									

- push(10); top = 1; A[ ]=

2	10								

- push(5); top = 2; A[ ]=

2	10	5							

Stack implementation using Arrays

Created an integer array A[10]=

Empty stack? top = -1

- push(2); top = 0; A[ ]=

2									

- push(10); top = 1; A[ ]=

2	10								

- push(5); top = 2; A[ ]=

2	10	5							

- after multiple push(x) operations: A[ ]=

2	10	5	15	30	7	9	14	16	35

top = 9

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 8$ A[ ]=

2	10	5	15	30	7	9	14	16	35

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 8$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 7$ A[ ]=

2	10	5	15	30	7	9	14	16	35

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 8$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 7$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 6$ A[ ]=

2	10	5	15	30	7	9	14	16	35

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 8$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 7$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 6$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- after multiple pop() operations: $top = -1$

Stack implementation using Arrays

current status of array ($top = 9$): A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 8$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 7$ A[ ]=

2	10	5	15	30	7	9	14	16	35

- pop(); $top = top - 1$; $top = 6$ A[ ]=

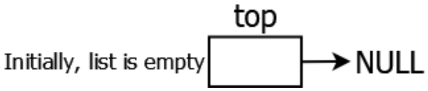
2	10	5	15	30	7	9	14	16	35

- after multiple pop() operations: $top = -1$
- no need to remove the values from array as they will be overwritten automatically with next push(x) operation.

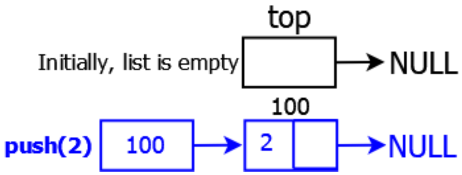
Stack implementation using Linked Lists

- Not one contiguous block of memory is allocated
- Insertion and deletion operations at the end of the list costs $\mathcal{O}(n)$ time.
- It is recommended to perform both $push(x)$ and $pop()$ operations at the beginning of the list $\mathcal{O}(1)$
- *head* of the list is marked as *top*

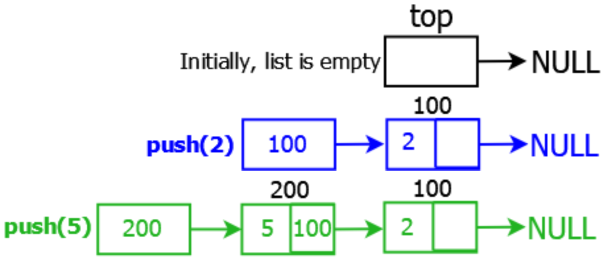
Stack implementation using Linked Lists



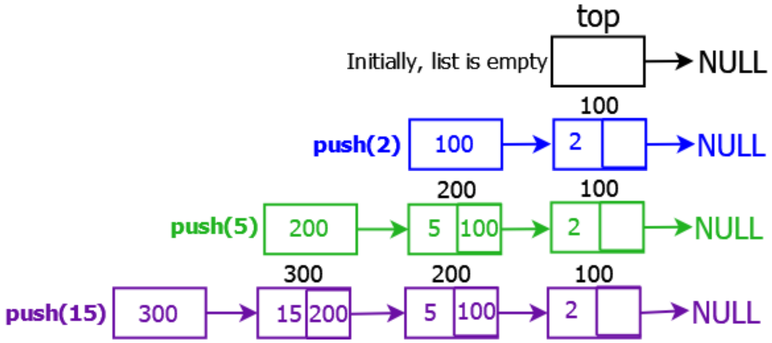
Stack implementation using Linked Lists



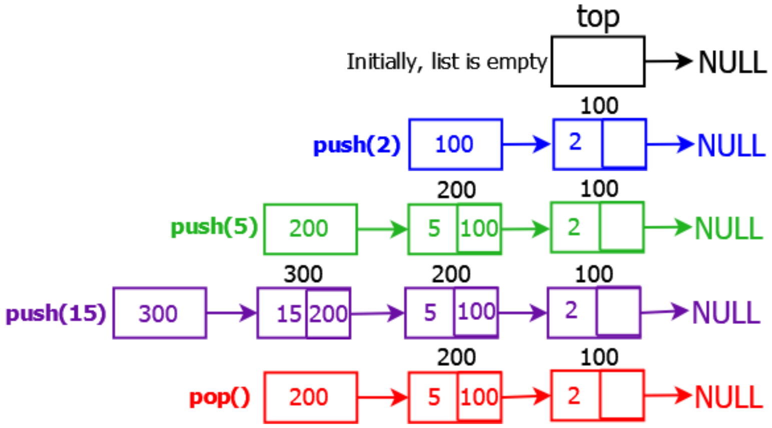
Stack implementation using Linked Lists



Stack implementation using Linked Lists



Stack implementation using Linked Lists



Polish Notations

- Arithmetic Expressions: $A + B$, $A * B$, A/B, $A + B * C - D * E$
- **Infix Notations:** <Operand> <Operator> <Operand>
- **Prefix Notations:** <Operator> <Operand> <Operand>
- **Postfix Notations**[2]**:** <Operand> <Operand> <Operator>

Order of Operations:

- Parentheses: (), {}, []
- Exponents: right to left
- Multiplication and Division: left to right
- Addition and Subtraction: left to right

[2]aka reverse polish notation

Polish Notations

Infix	Prefix	Postfix
$A + B$	$+AB$	$AB+$
$A * B$	$*AB$	$AB*$
$A + (B * C)$	$+A * BC$	$ABC * +$
$(A + B) * (C - D)$	$* + AB - CD$	$AB + CD - *$
$A + (B * C) \uparrow D$	$+A \uparrow *BCD$	$ABC * D \uparrow +$

Converting Notations (without Parentheses)

```
function INFIX_TO_POSTFIX(AExp)
    for {i=0; i < len(AExp); i++}
        if $type(AExp[i]) = operand$ then
            $string \leftarrow string + AExp[i]$
        else if $type(AExp[i]) = operator$ then
            while $!(S.isEmpty())\&\&HPrec(S.top(), AExp[i])$
                $string \leftarrow string + S.top()$
                $S.pop()$
            $S.push(AExp[i])$
    while $!(S.isEmpty())$
        $string \leftarrow string + S.top()$
        $S.pop()$
    return
```

Example- Infix to Postfix without ()

$a + b * c - d / e * h$

Input	Stack	Postfix Expression
	$\#$	
a	$\#$	a
$+$	$\#+$	a
b	$\#+$	ab
$*$	$\# + *$	ab
c	$\# + *$	abc
$-$	$\#-$	$abc * +$
d	$\#-$	$abc * +d$
$/$	$\# - /$	$abc * +d$
e	$\# - /$	$abc * +de$
$*$	$\# - *$	$abc * +de/$
h	$\# - *$	$abc * +de/h$
$\#$	$\#$	$abc * +de/h * -$

Converting Notations (with Parentheses)

Additional Rules

- Push an opening parenthesis to the Stack
- Pop all elements including opening parenthesis as soon as you get a closing parenthesis in the expression.
- Append the operators in the postfix expression string (excluding the parenthesis)
- Algorithm *Infix_to_Postfix*() remains same for rest of expression.

Example- Infix to Postfix with ()

$(a + b \uparrow c \uparrow d) * (e + (f/g))$

Input	Stack	Postfix
$($	$\#($	
a	$\#($	a
$+$	$\#(+$	a
b	$\#(+$	ab
$\uparrow$	$\#(+\uparrow$	ab
c	$\#(+\uparrow$	abc
$\uparrow$	$\#(+\uparrow\uparrow$	abc
d	$\#(+\uparrow\uparrow$	$abcd$
$)$	$\#$	$abcd\uparrow\uparrow+$
$*$	$\#*$	$abcd\uparrow\uparrow+$
$($	$\#*($	$abcd\uparrow\uparrow+$
e	$\#*($	$abcd\uparrow\uparrow+e$
$+$	$\#*(+$	$abcd\uparrow\uparrow+e$
$($	$\#*(+($	$abcd\uparrow\uparrow+e$
f	$\#*(+($	$abcd\uparrow\uparrow+ef$
$/$	$\#*(+(/$	$abcd\uparrow\uparrow+ef$
g	$\#*(+(/$	$abcd\uparrow\uparrow+efg$
$)$	$\#*(+$	$abcd\uparrow\uparrow+efg/$
$)$	$\#*$	$abcd\uparrow\uparrow+efg/+$
	$\#$	$abcd\uparrow\uparrow+efg/+*$

Infix to Prefix Conversion

- Reverse the expression
- Use postfix conversion algorithm
- reverse the output
  - push all characters to stack one by one.
  - pop all characters back once the expression is empty.

Example- Infix to Prefix

$(a + b \uparrow c) * d + e \uparrow f$

Input	Stack	Prefix Expression
f	$\#$	f
$\uparrow$	$\# \uparrow$	f
e	$\# \uparrow$	fe
$+$	$\#+$	$fe \uparrow$
d	$\#+$	$fe \uparrow d$
$*$	$\# + *$	$fe \uparrow d$
$)$	$\# + *)$	$fe \uparrow d$
c	$\# + *)$	$fe \uparrow dc$
$\uparrow$	$\# + *) \uparrow$	$fe \uparrow dc$
b	$\# + *) \uparrow$	$fe \uparrow dcb$
$+$	$\# + *)+$	$fe \uparrow dcb \uparrow$
a	$\# + *)+$	$fe \uparrow dcb \uparrow a$
$($	$\# + *$	$fe \uparrow dcb \uparrow a+$
	$\#$	$fe \uparrow dcb \uparrow a + *+$

reverse the expression: $+ * +a \uparrow bcd \uparrow ef$

Postfix to Infix Conversion

- Push the operand to the stack
- if the next element is an operator then pop two operands from the stack and place the operator between them
- push the resultant string back to the stack
- Repeat.

Example 1: postfix to infix conversion

$abcd \uparrow\uparrow + efg / + *$

Input	Stack
a	a
b	a, b
c	a, b, c
d	a, b, c, d
$\uparrow$	$a, b, (c \uparrow d)$
$\uparrow$	$a, (b \uparrow (c \uparrow d))$
$+$	$(a + (b \uparrow (c \uparrow d)))$
e	$(a + (b \uparrow (c \uparrow d))),\ e$
f	$(a + (b \uparrow (c \uparrow d))),\ e, f$
g	$(a + (b \uparrow (c \uparrow d))),\ e, f, g$
$/$	$(a + (b \uparrow (c \uparrow d))),\ e, (f/g)$
$+$	$(a + (b \uparrow (c \uparrow d))),\ (e + (f/g))$
$*$	$((a + (b \uparrow (c \uparrow d)))\ *\ (e + (f/g)))$

Infix expression: $((a + (b \uparrow (c \uparrow d)))\ *\ (e + (f/g)))$

Example 2: postfix to infix conversion

1 2 3 2↑↑ + 5 15 3/+*

Input	Stack
1	1
2	1, 2
3	1, 2, 3
2	1, 2, 3, 2
↑	1, 2, 9
↑	1, 512
+	513
5	513, 5
15	513, 5, 15
3	513, 5, 15, 3
/	513, 5, 5
+	513, 10
*	5130

Prefix to Infix Conversion

- Reverse the input (prefix) expression.
- Push the operand to the stack
- If the next element is an operator then pop two operands from the stack and place the operator between them
- Push the resultant string back to the stack
- Repeat.

Example: prefix to infix conversion

$+ * + a \uparrow bcd \uparrow ef$

reverse the expression: $fe \uparrow dcb \uparrow a + * +$

Input	Stack
f	f
e	f, e
$\uparrow$	$(f \uparrow e)$
d	$(f \uparrow e), d$
c	$(f \uparrow e), d, c$
b	$(f \uparrow e), d, c, b$
$\uparrow$	$(f \uparrow e), d, (c \uparrow b)$
a	$(f \uparrow e), d, (c \uparrow b), a$
$+$	$(f \uparrow e), d, ((c \uparrow b) + a)$
$*$	$(f \uparrow e), (d * ((c \uparrow b) + a))$
$+$	$((f \uparrow e) + (d * ((c \uparrow b) + a)))$

Reverse the resultant expression: $(((a + (b \uparrow c)) * d) + (e \uparrow f))$

Tower of Hanoi

- The Tower of Hanoi is a mathematical game or puzzle.
- It consists of three rods, and a number of disks of different sizes which can slide onto any rod.
- The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape.
- The objective of the puzzle is to move the entire stack to another rod.

Rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
3. No disk may be placed on top of a smaller disk.

Tower of Hanoi

The following is a procedure for moving a tower of N disks from a rod *start st* onto a *destination dest*, with *aux* being the remaining third or auxiliary rod:

Step 1 If $N>1$ then first use this procedure to move the $N-1$ smaller disks from rod *st* to rod *aux*.

Step 2 Now the largest disk, i.e. disk $N-1$ can be moved from rod *st* to rod *dest*.

Step 3 If $N>1$ then again use this procedure to move the $N-1$ smaller disks from rod *aux* to rod *dest*.

Recurrence Equation: $T(n) = 2T(n-1) + 1$