STACKS

STACK: Definition

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.
- A stack can be represented in memory using a linear array or a linked list.

Abstract Data Type

- An ***abstract data type (ADT)***, is a logical description of how we view the data and the allowed operations without regard to how they'll be implemented. This means that we're only concerned with what the data represents and not with how it'll be constructed. This level of abstraction ***encapsulates*** the data and hides implementation details from the user's view, a technique called ***information hiding***.
- A ***data structure*** is an implementation of an abstract data type and requires a physical view of the data using some collection of primitive data types and other programming constructs.
- The separation of these two perspectives allows us to define complex data models for our problems without giving any details about how the model will actually be built. This provides an ***implementation-independent*** view of the data. Since there are usually many ways to implement an abstract data type, this independence allows the programmer to change implementation details without changing how the user interacts with it. Instead, the user can remain focused on the process of problem-solving.

What is a Stack?

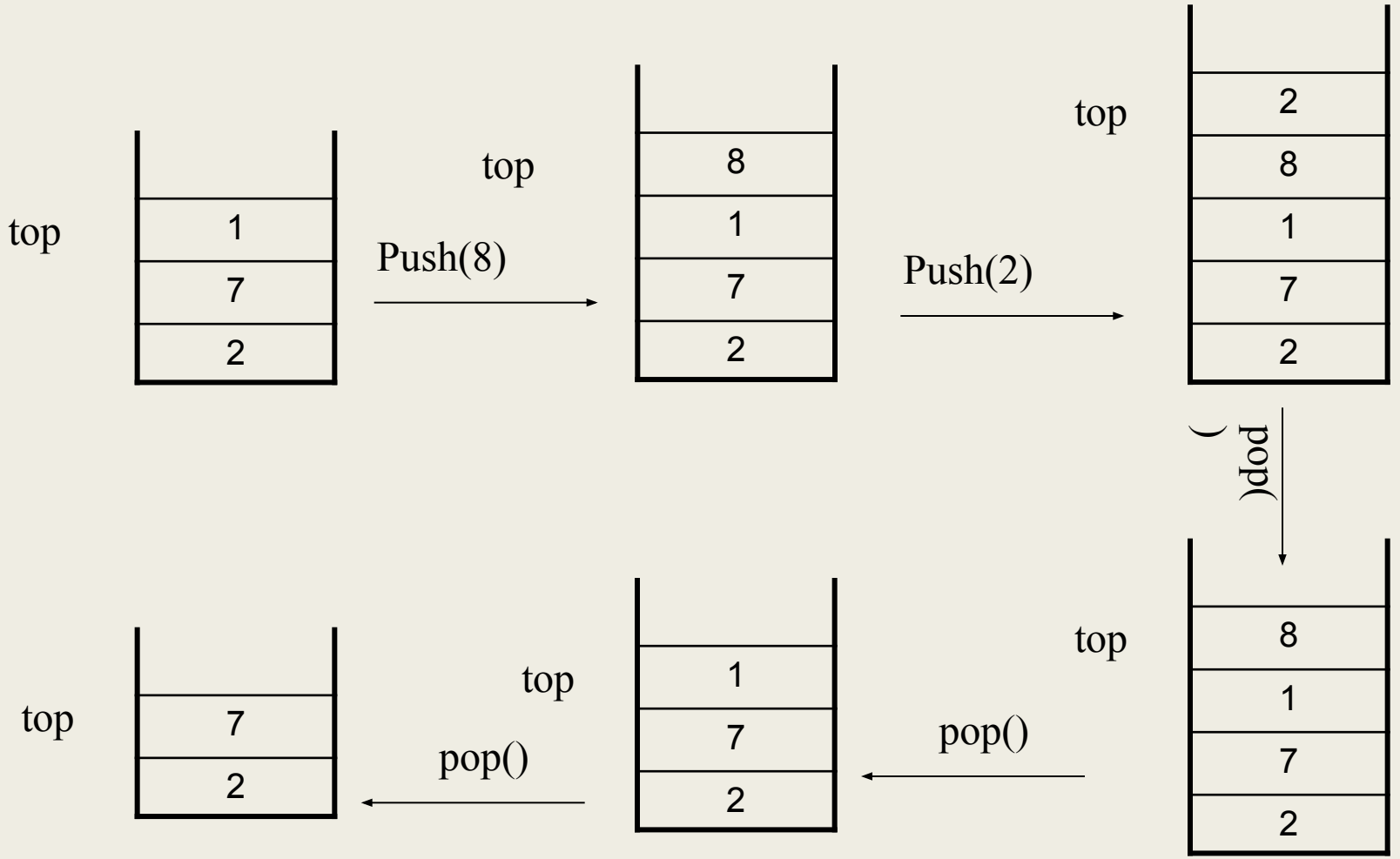
- Stack is a data structure in which data is added and removed at only one end called the top.
- To add (push) an item to the stack, it must be placed on the top of the stack.
- To remove (pop) an item from the stack, it must be removed from the top of the stack too.
- Thus, the last element that is pushed into the stack, is the first element to be popped out of the stack. i.e., **Last In First Out (LIFO)**

Stack

- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

- A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure.

An Example of Stack



Features of stacks

- Dynamic data structures
- Do not have a fixed size
- Do not consume a fixed amount of memory
- Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.

Operations on Stacks

- The following operations are performed on stacks:
- **CreateStack(S)** – to create **S** as an empty stack
- **Push(S, item)** – to push element ***item*** onto stack **S**
- **Pop(S, item) –** to remove the top element of the stack **S.**
- **Peep(S, item) –** to access the top element of the stack **S** without removing it from the stack **S.**
- **IsFull(S, status) –** to check whether the stack **S** is full
- **IsEmpty(S, status) –** to check whether the stack **S** is empty

Points to remember

- If the stack S is empty, then it is not possible to pop the stack S (Underflow). Similarly, Peep() operation is also not valid. Therefore, we must ensure that S is not empty before attempting to perform these operations.
- Likewise, if the stack S is full then it is not possible to push a new element on the stack S (Overflow). Therefore, we must ensure that S is not full before attempting to perform push operation.

Representation of Stack in Memory

- A stack can be represented in memory using a linear array or a linked list.

Representation using Array

- Unless stated, stack will be represented in memory using an array named ***stack***, an integer variable ***top*** that holds the index of the top element of the stack and another variable ***size*** that holds the value signifying the maximum number of elements that can be held by the stack.

0	1	2	3	4	5	6	7	8	9
8	10	12	7	11	9	55			

6

top

Creating an Empty Stack

- Before we can use a stack, it is to be initialized. As the index of array elements can take value in the range 0 to (size -1), the purpose of initializing the stack is served by assigning value -1 to the ***top*** variable.
- **Algorithm for creating stack**

CreateStack(S)

This sub algorithm assigns value -1 to variable top to put the stack in the initial state, i.e., empty.

1. Set top = -1
2. Return

Testing Stack for Underflow

- Before we remove an item from a stack, it is necessary to test whether stack still have some elements, i.e., to test that whether the stack is empty or not.
- If it is not empty then the pop operation can be performed to remove the top element.
- This test is performed by comparing the value of top with sentinel value -1

Algorithm for testing Underflow

IsEmpty (S, status)

This sub algorithm compares the value of ***top*** with the sentinel value (-1). If the value of the ***top*** is -1 it sets the variable ***status*** to true(underflow) else value false.

1. If (top = -1) then

   Set status = true

   Else

   Set status = false

   EndIf
2. Return

Testing Stack for Overflow

- Before we insert an item into a stack, it is necessary to test whether stack still have some space to accommodate the new elements, i.e., to test that whether the stack is full or not.
- If it is not full then the push operation can be performed to insert the element at the top of the stack.
- This test is performed by comparing the value of ***top*** with (size -1), the largest index value that the ***top*** can take.

Algorithm to check Overflow

IsFull (S, status)

This sub algorithm compares the value of ***top*** with (size-1). If the value of the ***top*** is (size-1) it sets the variable ***status*** to true(overflow) else value false.

1. If (top = (size-1)) then

   Set status = true

   Else

   Set status = false

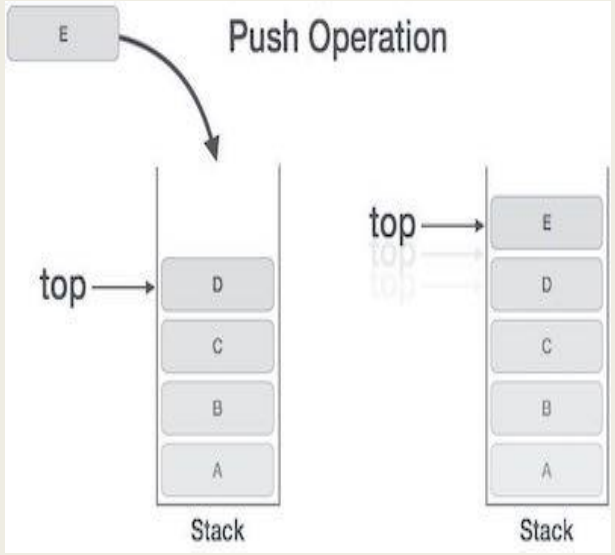
   EndIf
2. Return

Push Operation

- Before the push operation, stack is empty. Thus, the value of ***top*** will be -1. And if stack is not empty, then the value of the ***top*** will be the index of the element currently on the top.
- Therefore, before we insert the element onto the stack, the value of ***top*** is incremented so that it points to the new top of the stack.

Push Operation

- The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −
- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Push Operation: Algorithm

Push (S, item)

This sub-algorithm calls sub algorithm IsFull to see whether the stack is full. If the answer is true, it flags “overflow” condition and returns. However, if the stack is not full, it pushes (insert) new element item onto the top of the stack.

1. Call IsFull (S, status)
2. If (status = true) then

   Write: “Overflow”

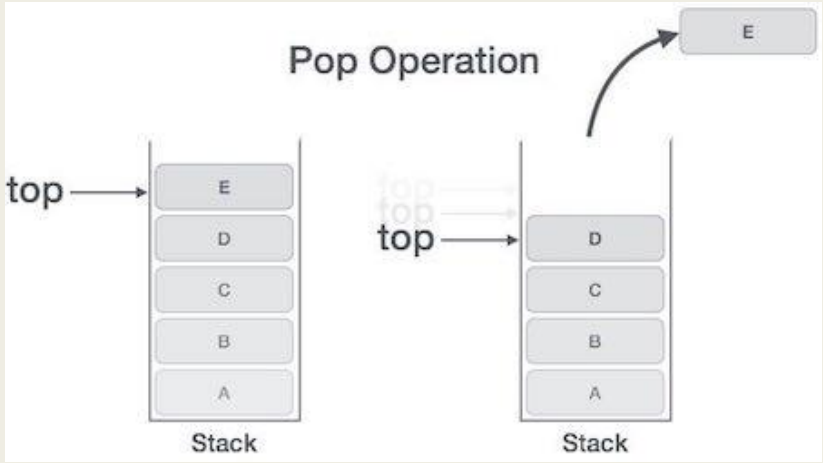
   Return

   EndIf
3. Set top = top + 1
4. Set S[top] = item
5. Return

Pop Operation

- The element at the top of the stack is assigned to a local variable and value of variable top is decremented so that it points to new top.
- A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Pop Operation: Algorithm

Pop (S, item)

This sub algorithm calls sub algorithm ***IsEmpty*** to check whether the stack is empty. If the answer is true, it flags "***underflow***" and exits. Otherwise, it pops (remove) the element from the top of the stack and assigns it to variable ***item***.

1. Call IsEmpty (S, status)
2. If (status = true) then

   Write: "Underflow"

   Return

   EndIf
3. Set item = S[top]
4. Set top = top-1
5. Return

Peep Operation

- There may be instances where we want to access the top element of the stack without removing it from the stack, i.e., without performing Pop operation.

Peep(S, item)

This algorithm calls sub-algorithm ***IsEmpty*** to check whether the stack is empty. If answer is true it flags "Underflow". Otherwise, it peeps (access top element without removing it) element from the top of the stack and assigns it to the variable ***item***.

1. Call IsEmpty (S, status)
2. If (status = true)

   Write: "Underflow"

   Return

   EndIf
3. Set item = S[top]
4. Return

Performance Analysis

- All of discussed operations runs in O(1) time, i.e., time taken by each operation is same irrespective of stack size.

Applications of Stack

- Representing Arithmetic Expressions
- Quicksort

Arithmetic Expressions

- Notations used for writing Arithmetic Expressions
- Converting expressions from one notation to another
- Evaluating Arithmetic notations

Arithmetic Notations

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

List of Operators

Symbol Used	Operation Performed	Precedence
* (asterisk)	Multiplication	Highest
/ (slash)	Division	Highest
% (percentage)	Modulus (Remainder)	Highest
+ (plus)	Addition	Lowest
- (hyphen)	Subtraction	Lowest

Arithmetic Notations

■ Infix Notation

Operators are written in-between their operands. This is the way we are used to writing expressions.

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and parentheses to allow users to override these rules. Infix notation is difficult to parse by computers in comparison to prefix or postfix notations.

Example: 3 + 4

■ Postfix Notation (Reverse Polish Notation)

Here, **Operators are written after their operands.**

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. This notation is faster because reverse Polish calculators do not need expressions to be parenthesized, fewer operations need to be entered to perform typical calculations.

Example: 3 4 +

■ Prefix Notation (Polish Notation)

Here, **Operators are written before their operands.**

Operators are evaluated left-to-right and brackets are unnecessary. The operators act on the two nearest values on the right.

Example: + 3 4

The following table briefly tries to show the difference in all three notations −

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Parsing Expressions

- Stack organized computers are better suited for post-fix notation then the traditional infix notation. Thus the infix notation must be converted to the post-fix notation.
- The conversion from infix notation to post-fix notation must take into consideration the operational hierarchy.
- To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

- When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

- As multiplication operation has precedence over addition, b * c will be evaluated first.

Associativity

- Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.
- Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

- The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −
- In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Infix to postfix conversion using stack

Steps to convert infix expression to postfix

1. Scan the given infix expression from left to right.
2. If the scanned character is an operand, then output it.
3. Else,
   - If the precedence of the scanned operator is greater than the precedence of the operator in the top of the stack(or the stack is empty or if the stack contains a ‘(‘), push it.
   - Else, Pop all operators from the stack whose precedence is greater than or equal to that of the scanned operator. Then, push the scanned operator to the top of the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is ‘(‘, push it to the stack.
5. If the scanned character is ‘)’, pop the stack and output characters until ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until infix expression is scanned completely.
7. Print the output.
8. Pop and output from the stack.

Consider the following Infix expression
(7-5)*(9/2)

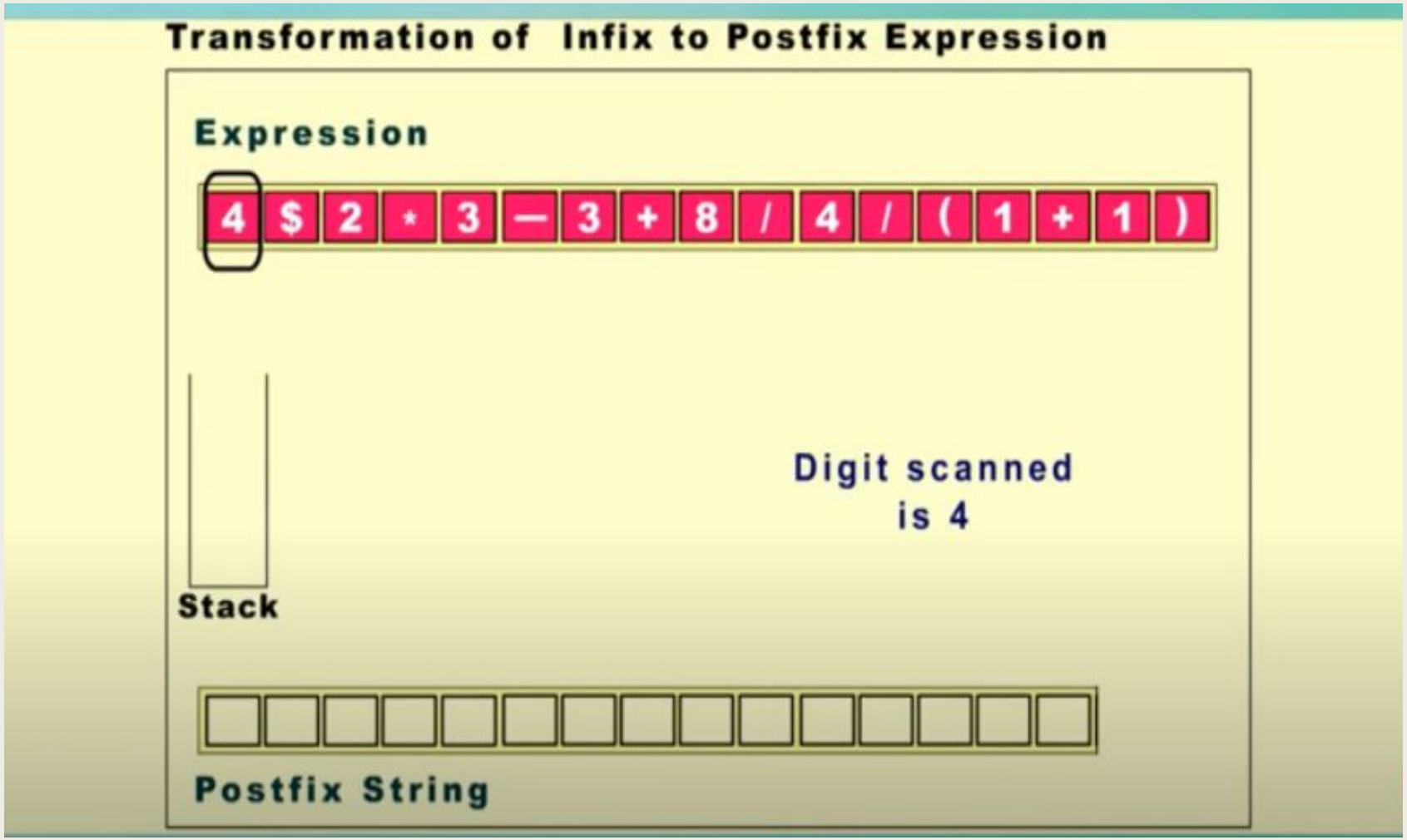
Conversion from infix to postfix

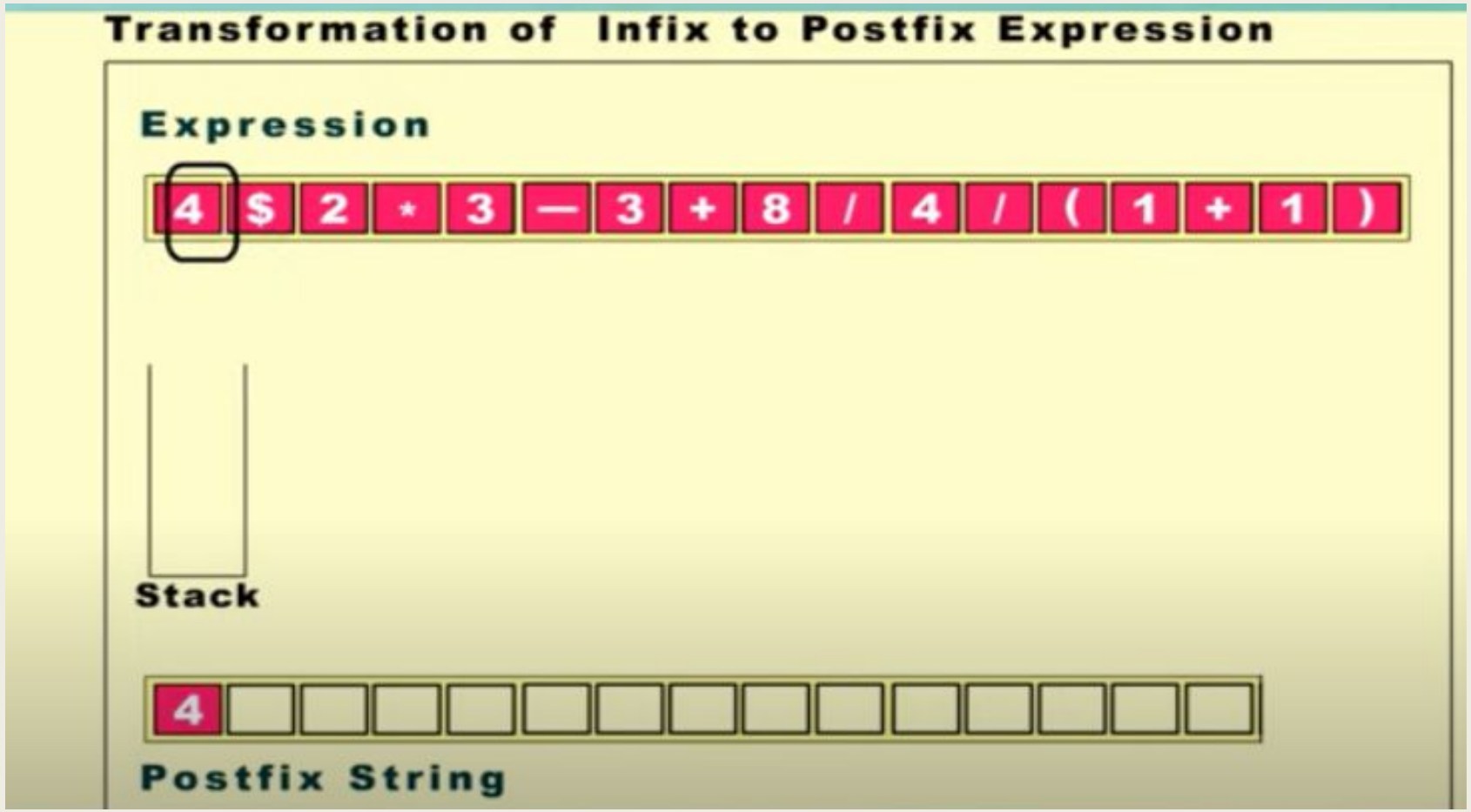
Character Scanned	Stack	Expression p
(	(	
7	(	7
-	(, -	7
5	(, -	7, 5
)	Empty	7, 5, -
*	*	7, 5, -
(	*, (	7, 5, -
9	*, (	7, 5, -, 9
/	*, (, /	7, 5, -, 9
2	*, (, /	7, 5, -, 9, 2
)	*	7, 5, -, 9, 2, /
End of expression	Empty	7, 5, -, 9, 2, /, *

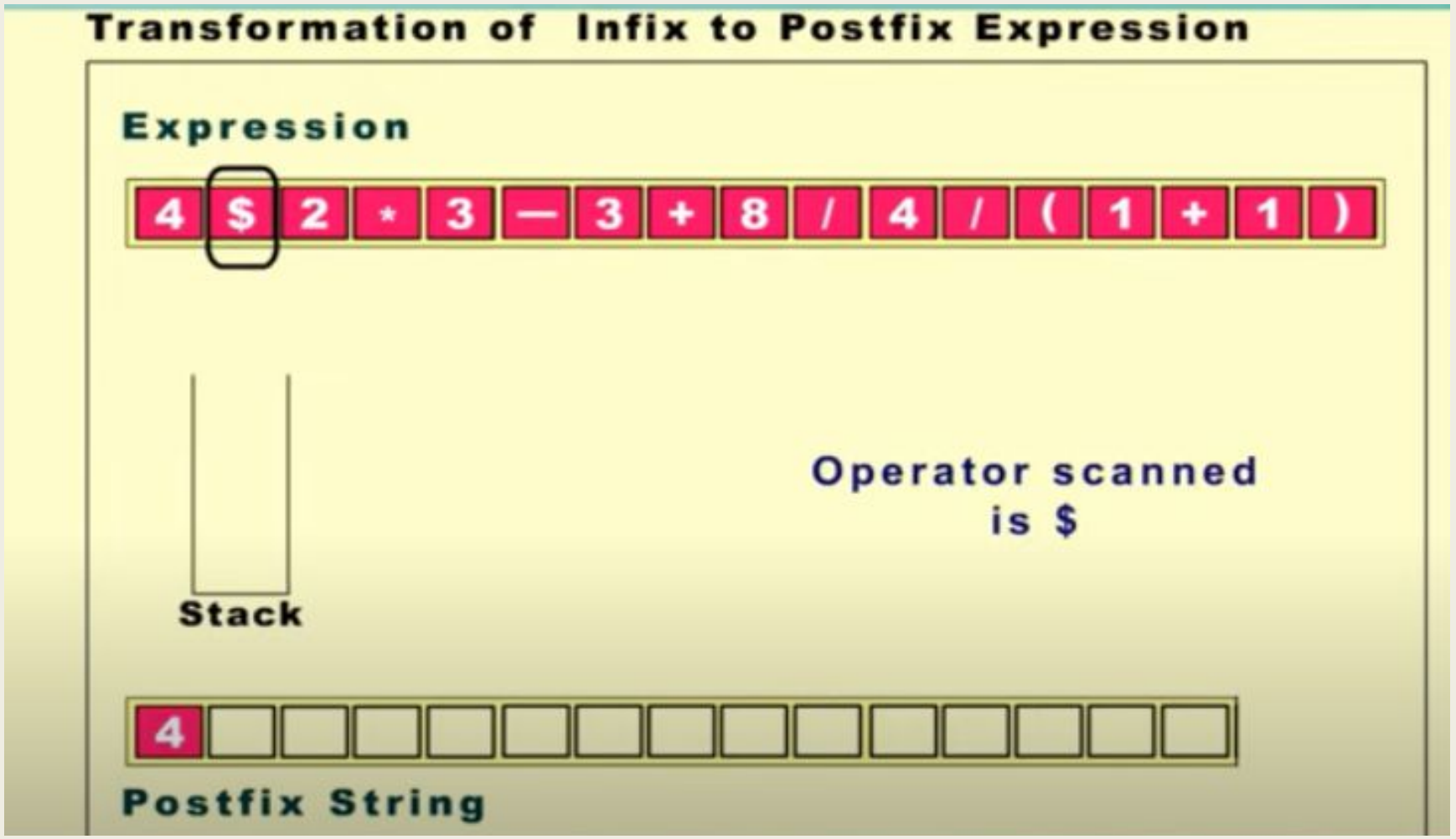
Evaluating Expression in Postfix Notation

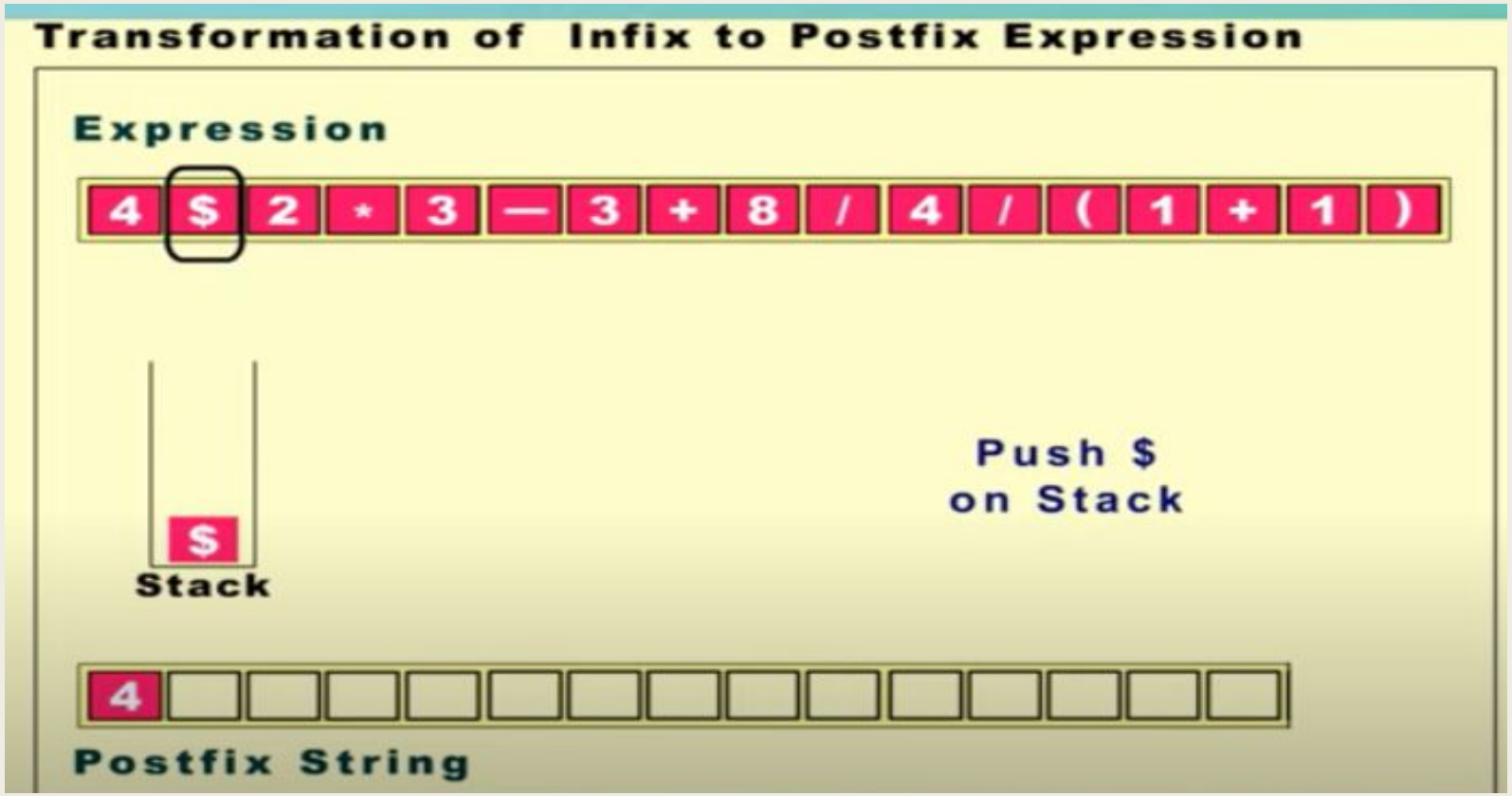
- Expression: 7 5 – 9 2 / *

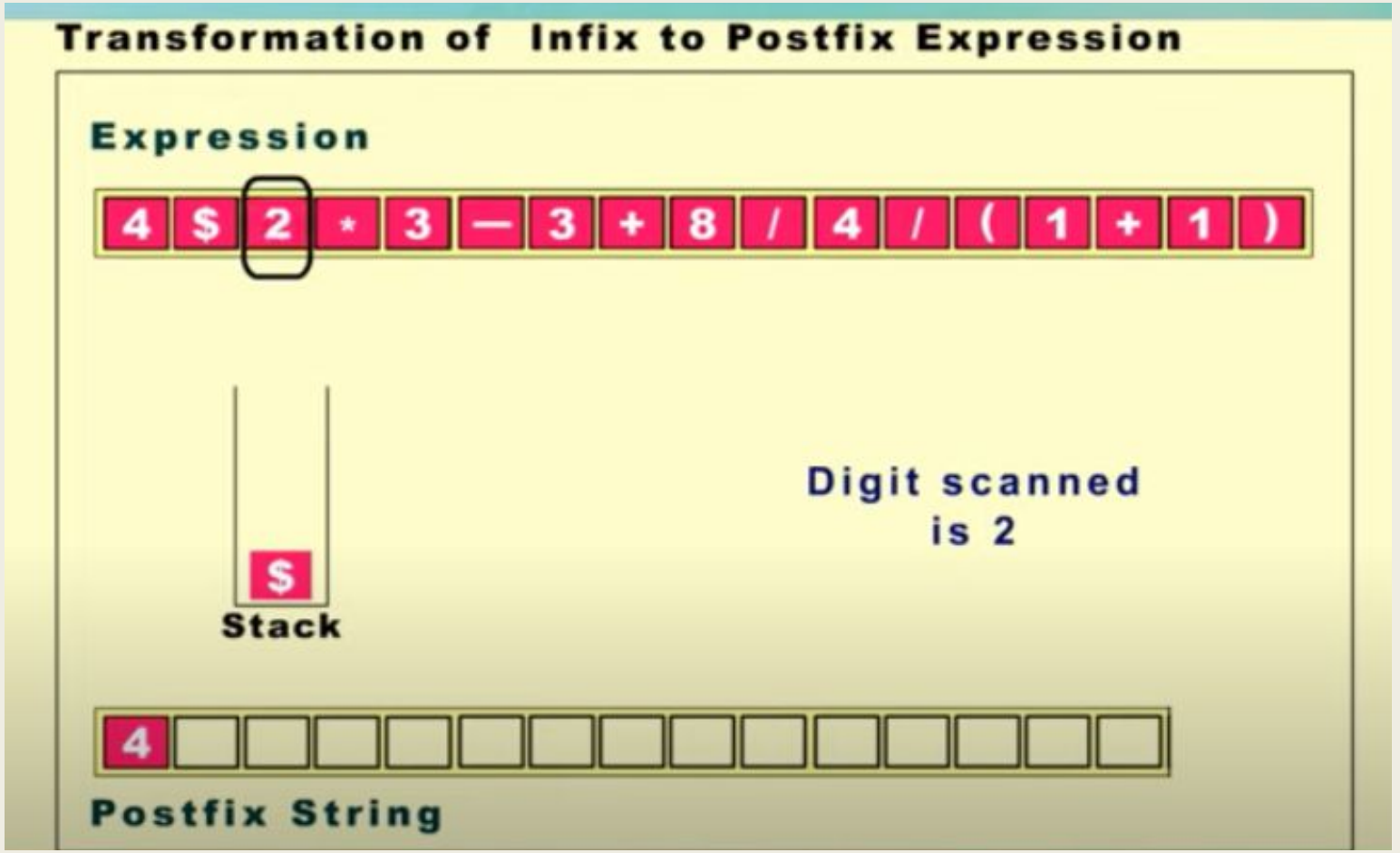
Character Scanned	Stack
7	7
5	7, 5
-	2
9	2, 9
2	2, 9, 2
/	2, 4.5
*	9
End of Expression	9

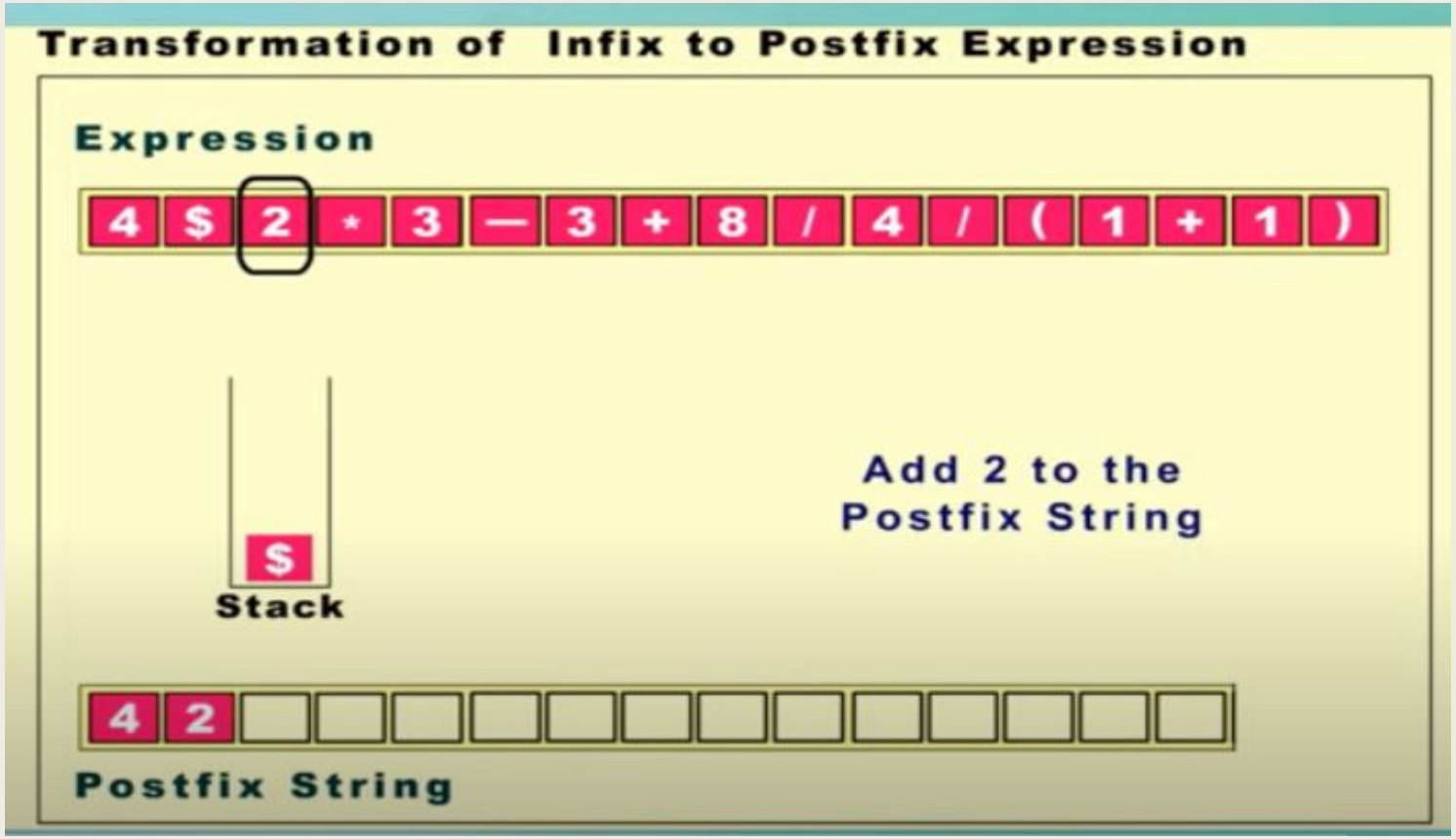

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 – 3 + 8 / 4 / (1 + 1)
Digit scanned
is 4
Stack
Postfix String

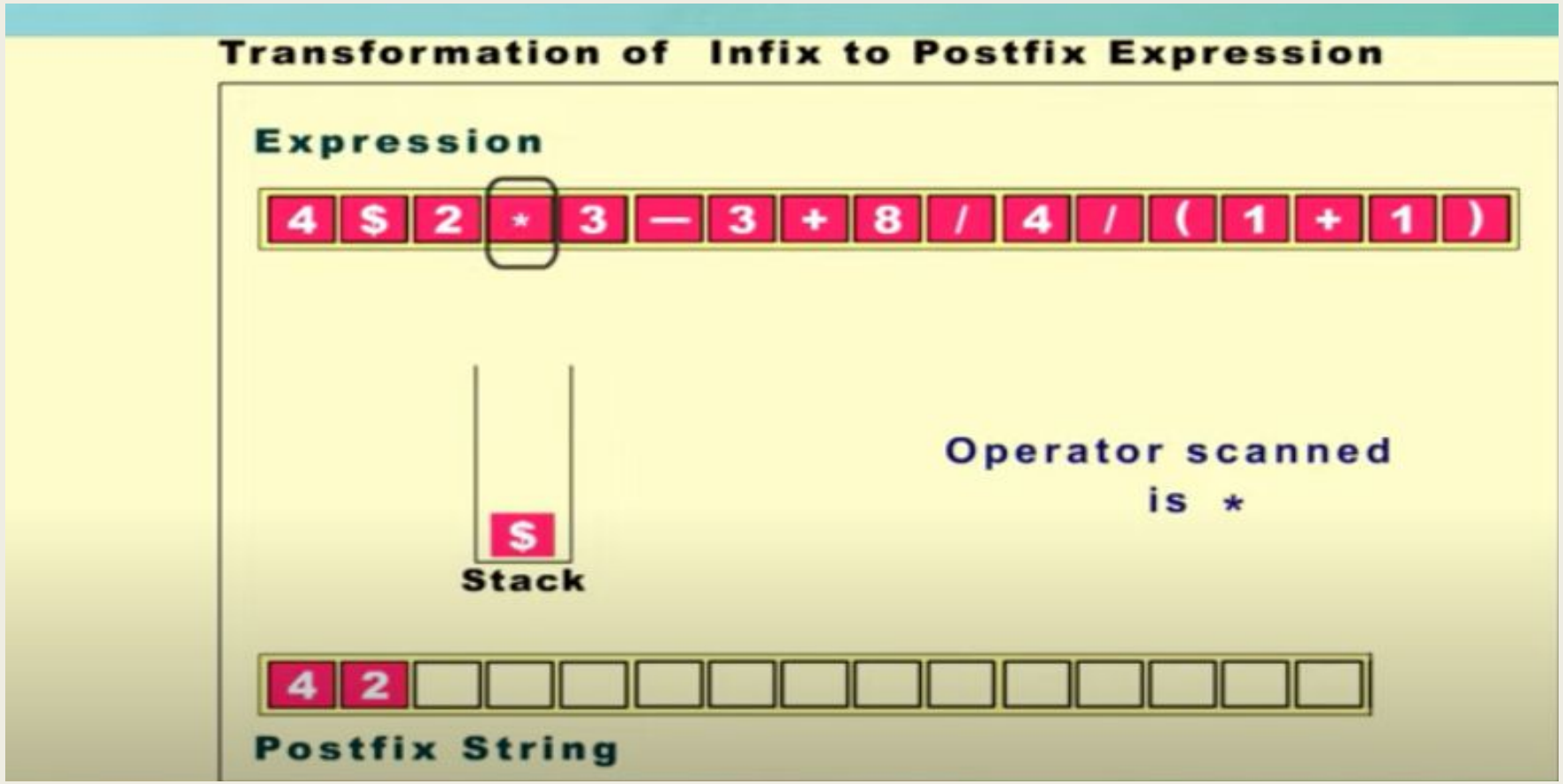

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Stack
4
Postfix String

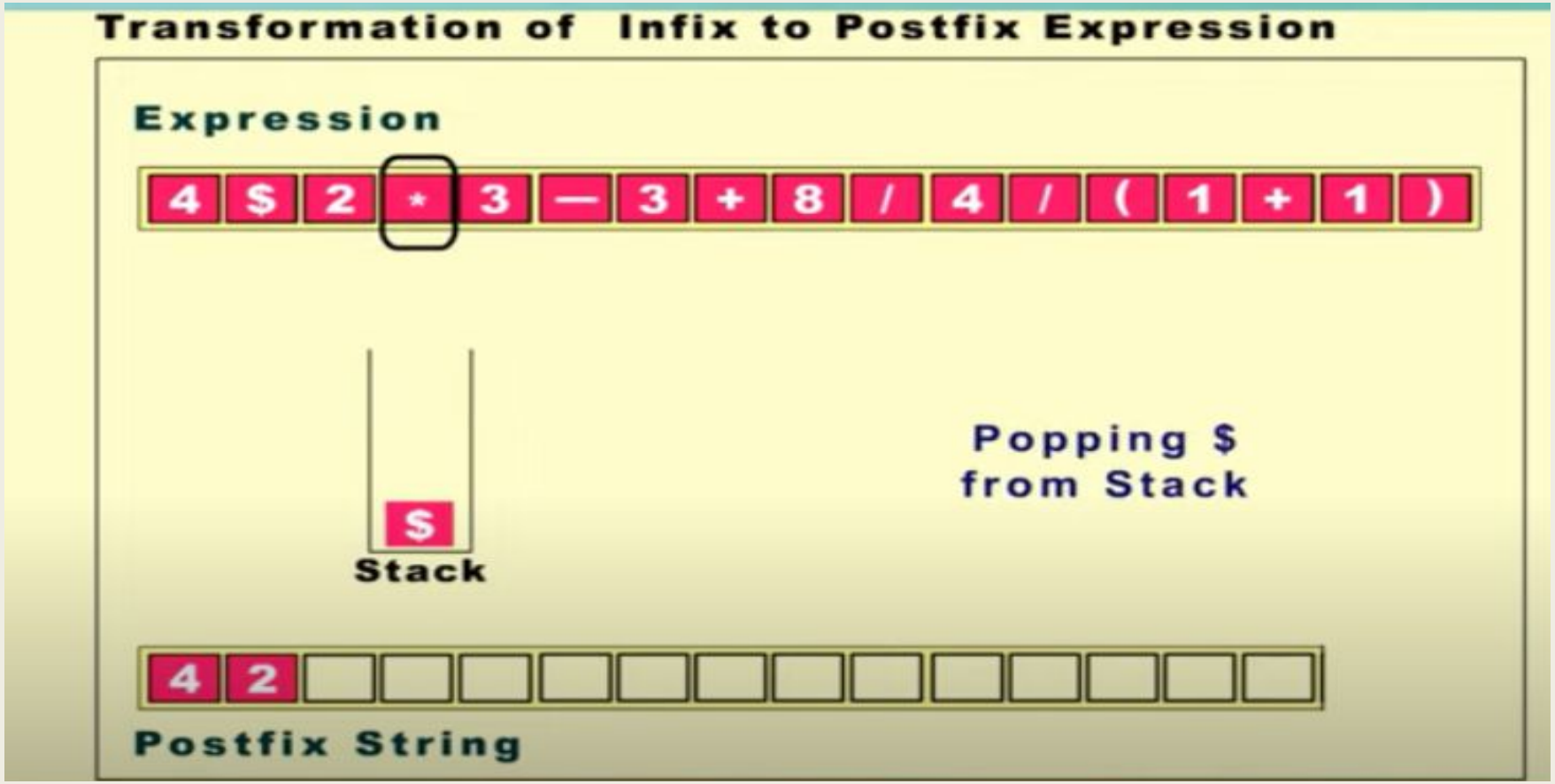

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Stack
Operator scanned is $
4
Postfix String

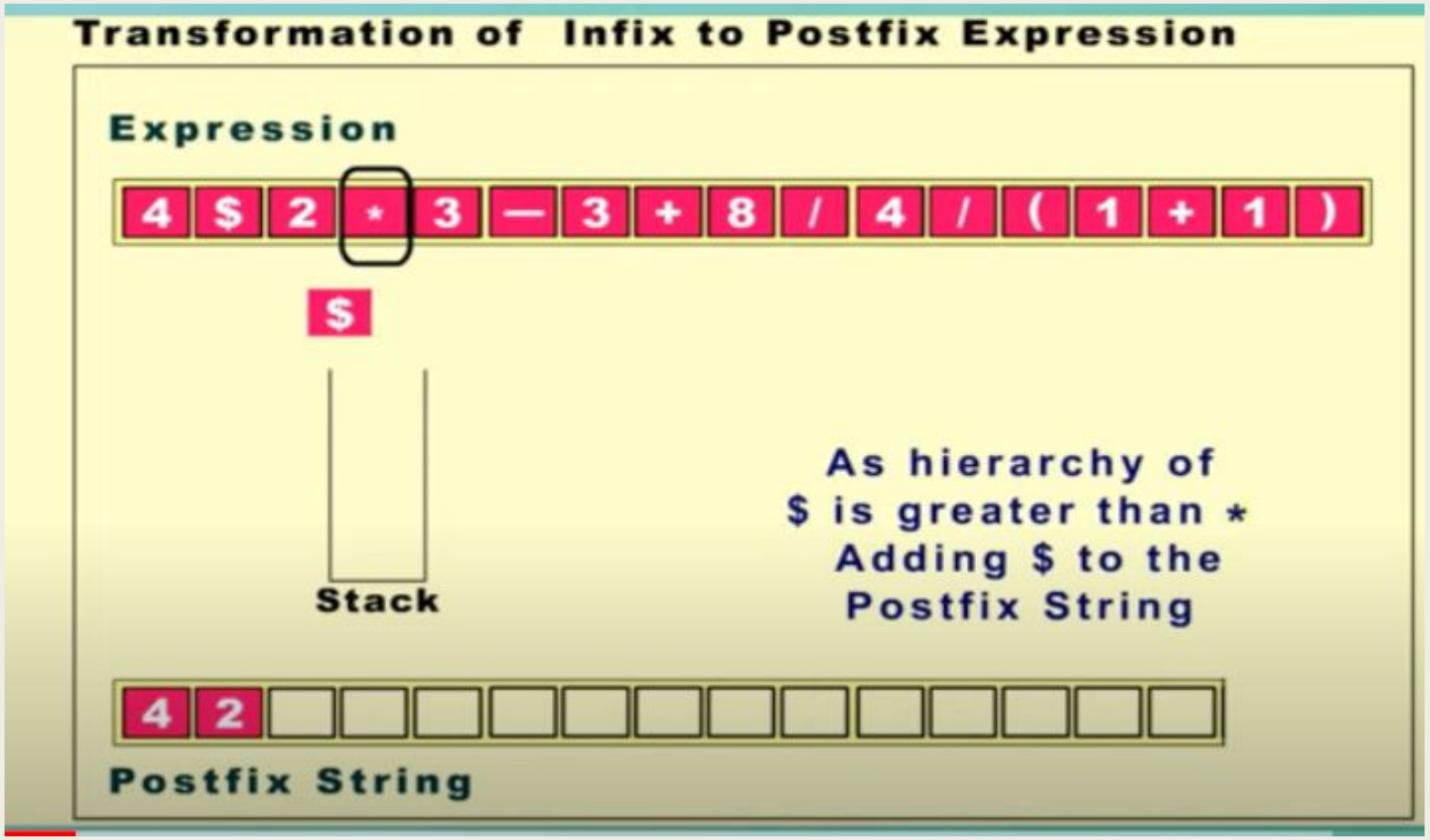

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
$
Stack
Push $
on Stack
4
Postfix String

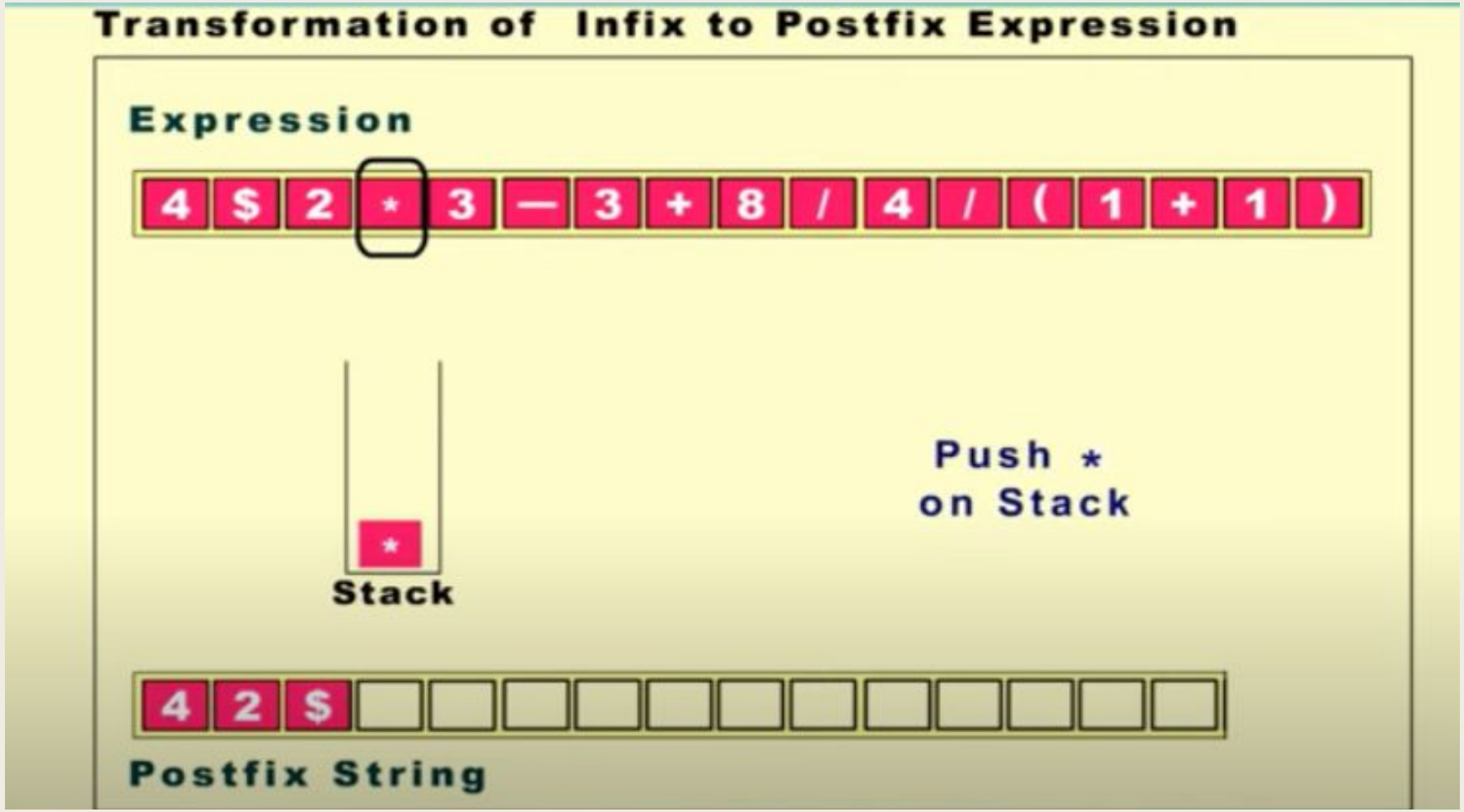

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 – 3 + 8 / 4 / (1 + 1)
$
Stack
Digit scanned
is 2
4
Postfix String

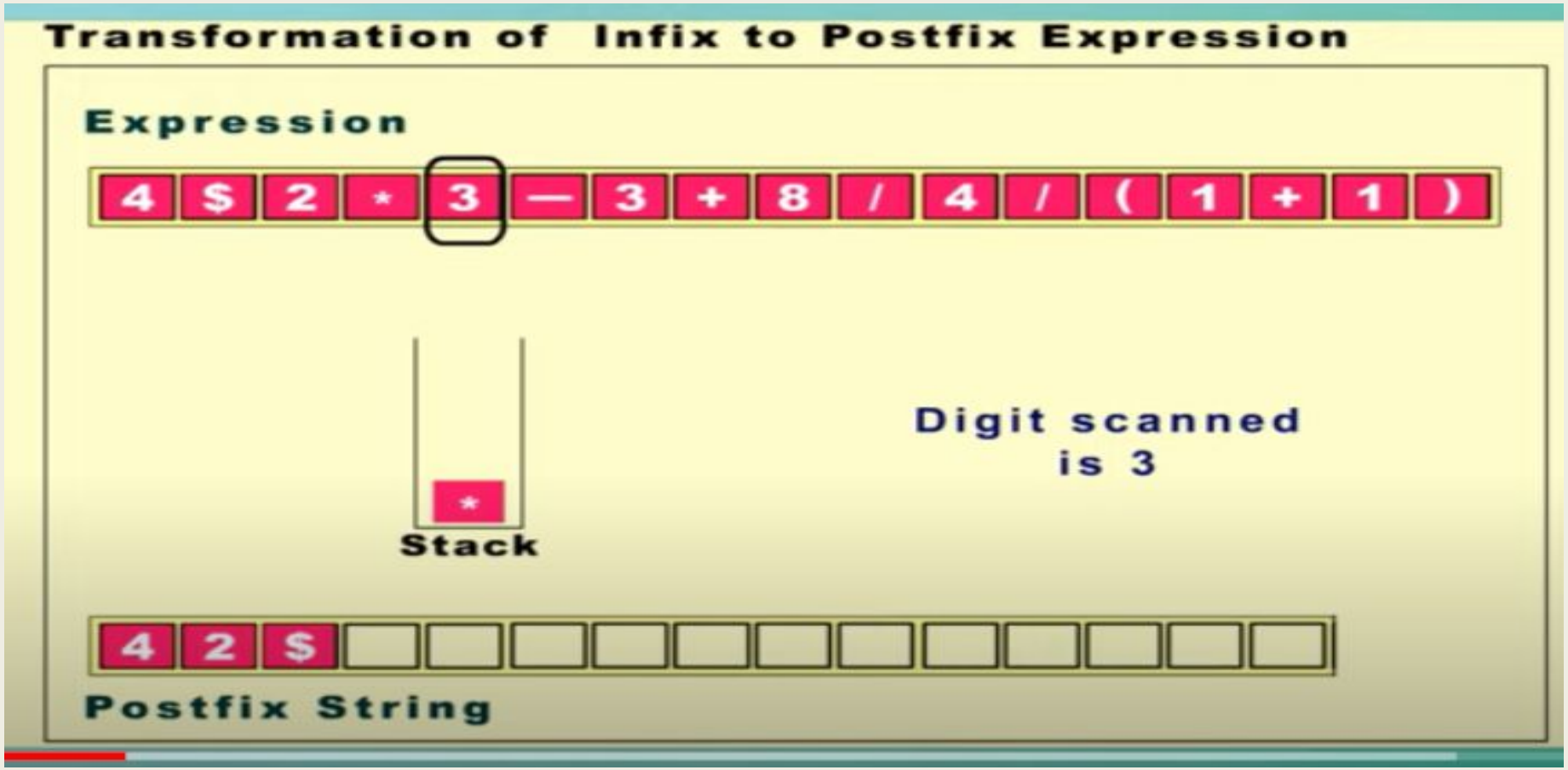

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
$
Stack
Add 2 to the
Postfix String
4 2
Postfix String

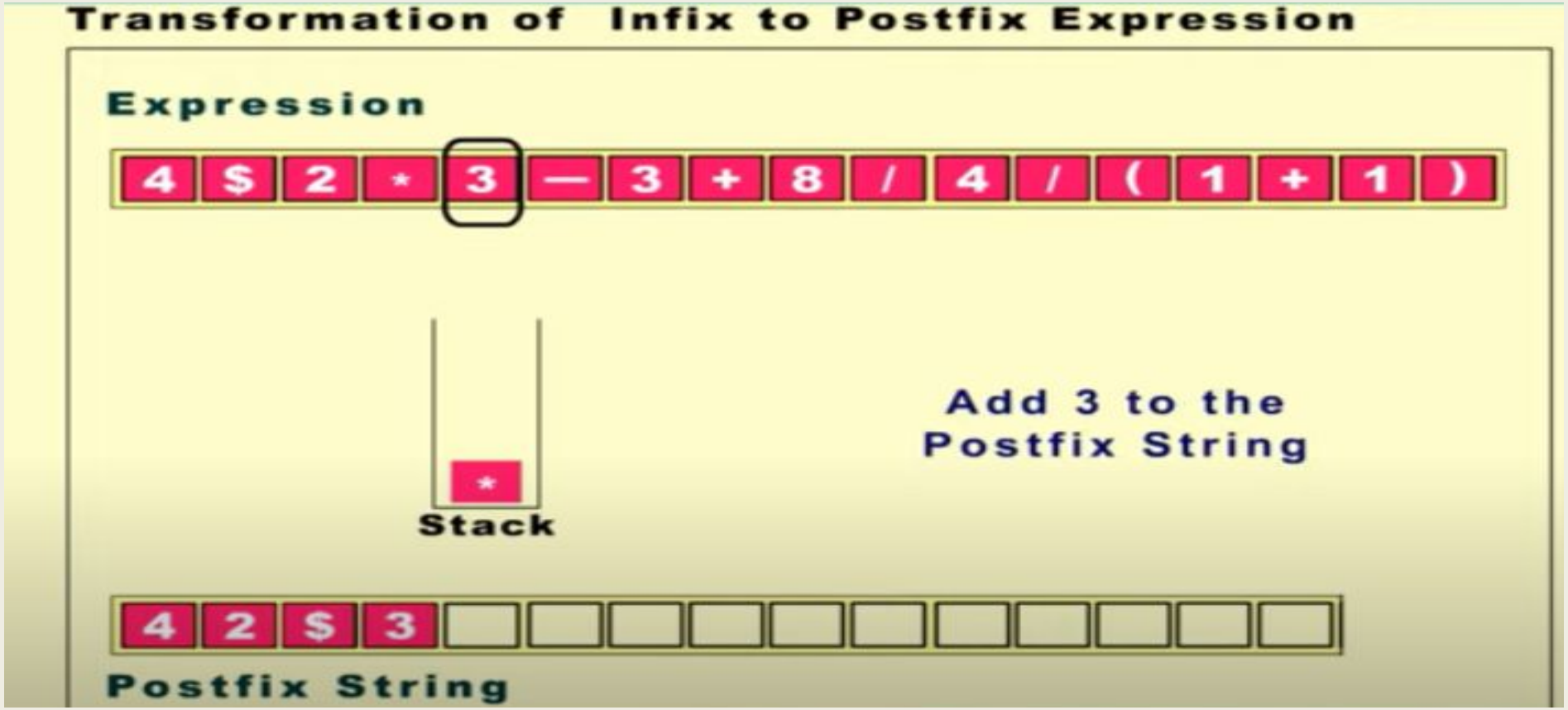

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
$
Stack
Operator scanned is *
4 2
Postfix String

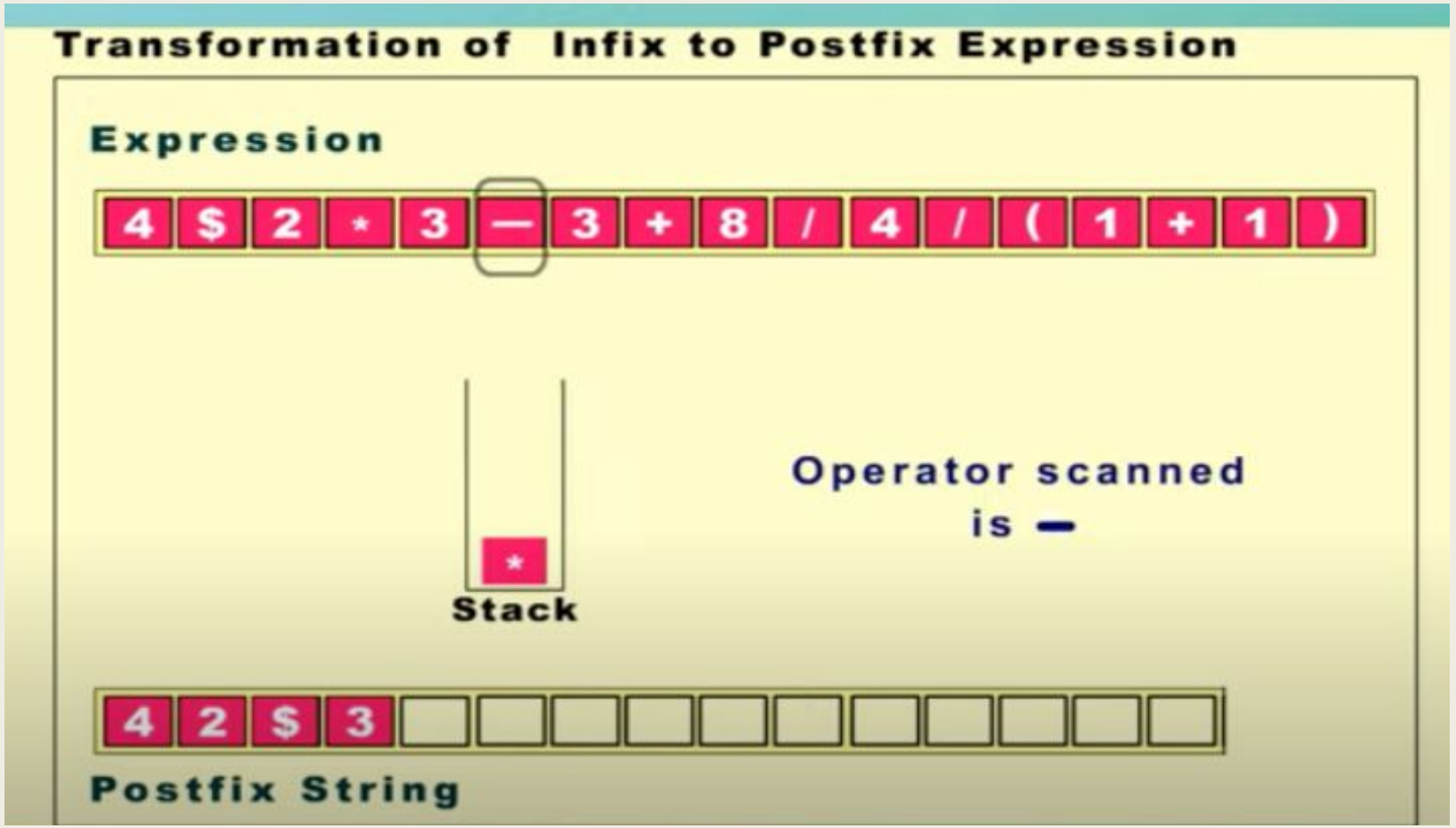

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
$
Stack
Popping $
from Stack
4 2
Postfix String

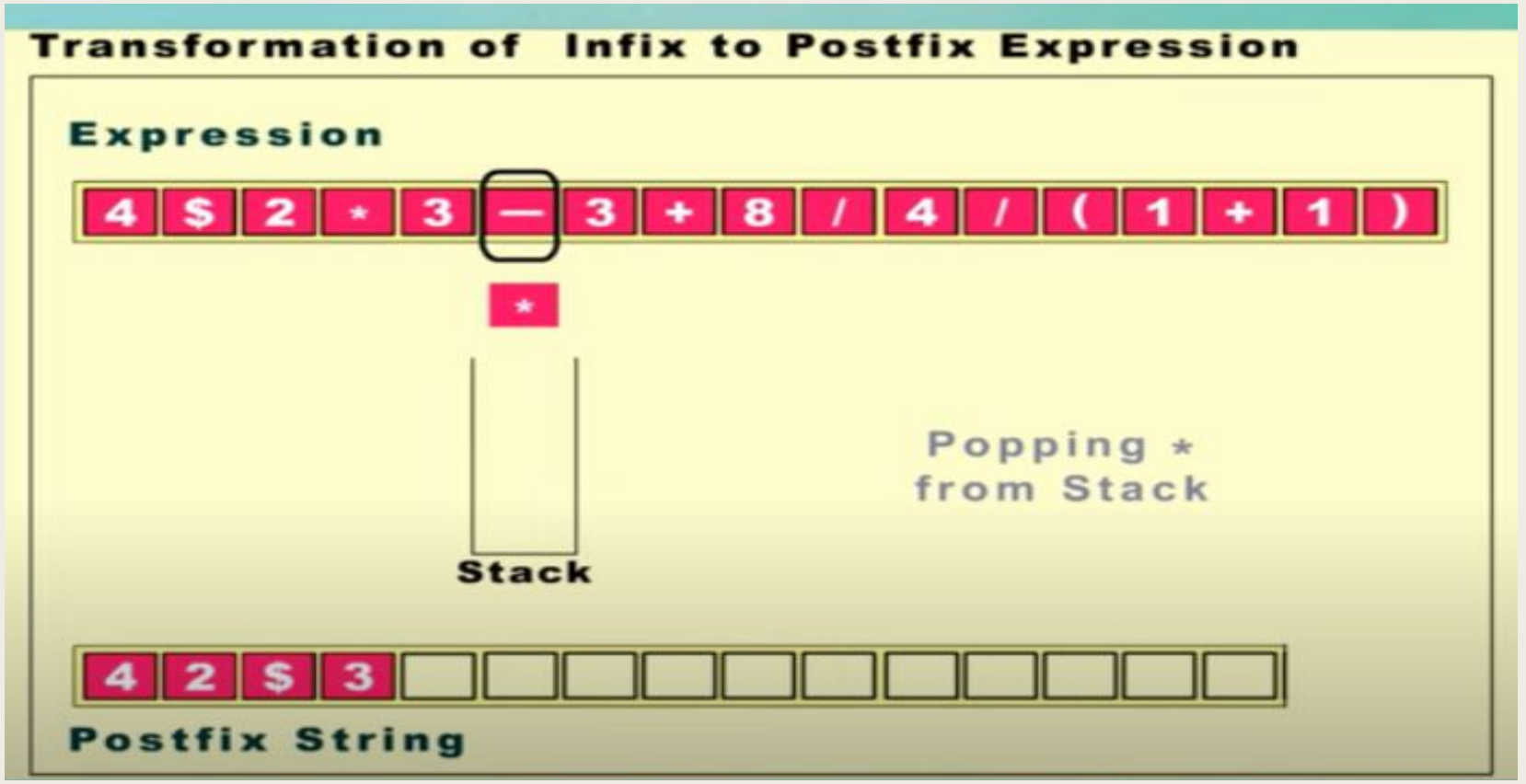

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
$
Stack
As hierarchy of
$ is greater than *
Adding $ to the
Postfix String
4 2
Postfix String

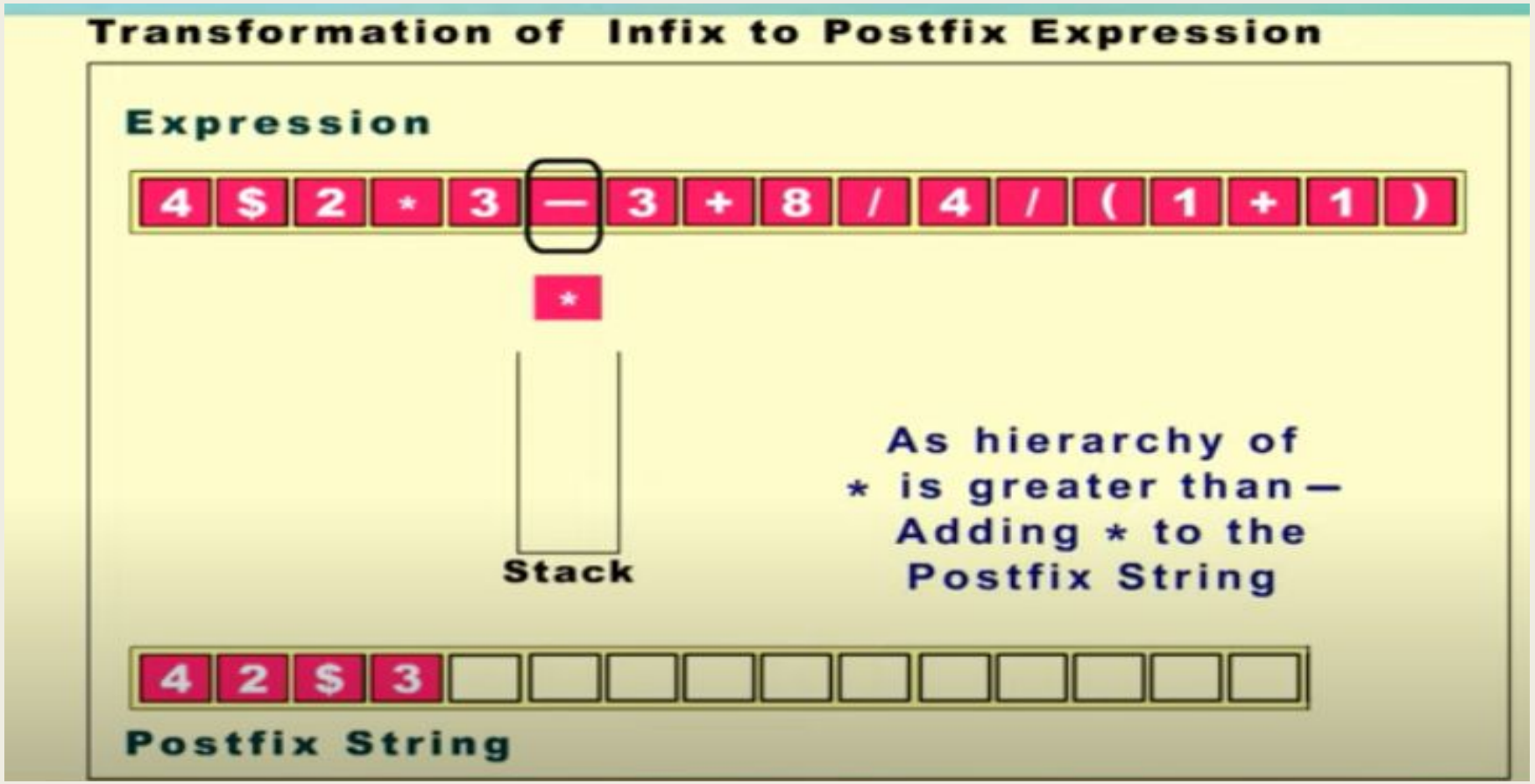

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Push * on Stack
*
Stack
4 2 $
Postfix String

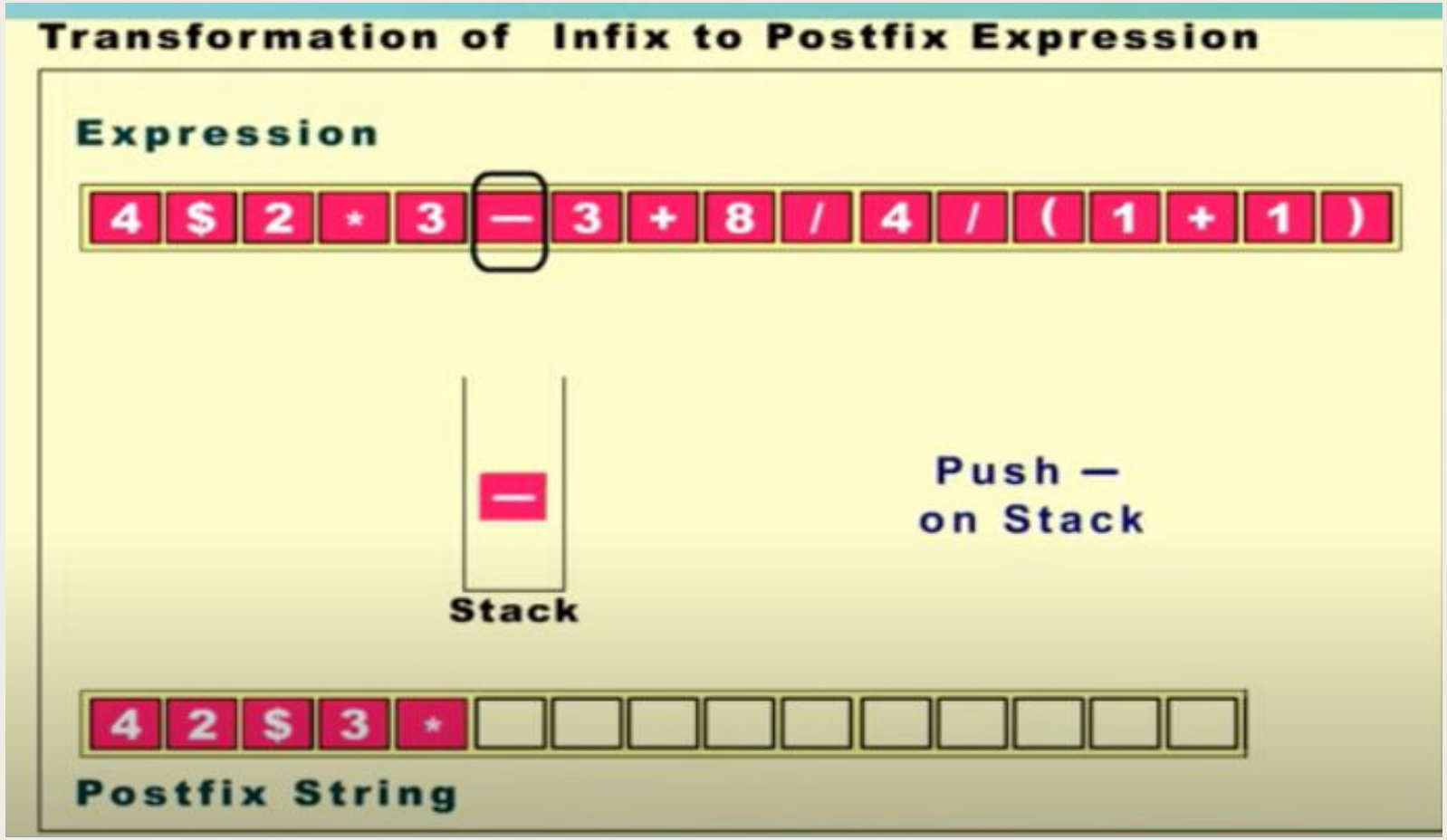

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
*
Stack
Digit scanned
is 3
4 2 $
Postfix String

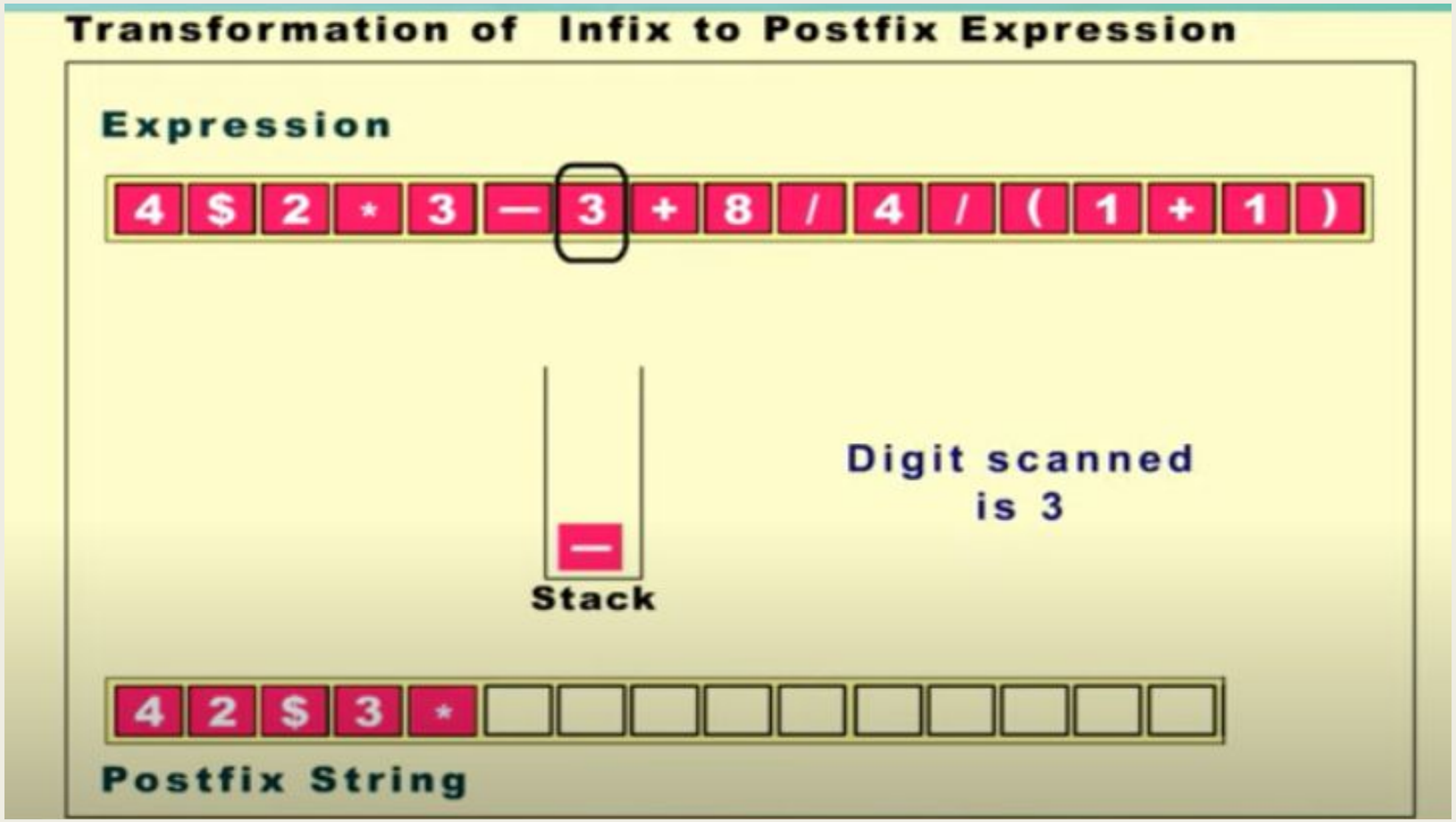

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add 3 to the
Postfix String
*
Stack
4 2 $ 3
Postfix String

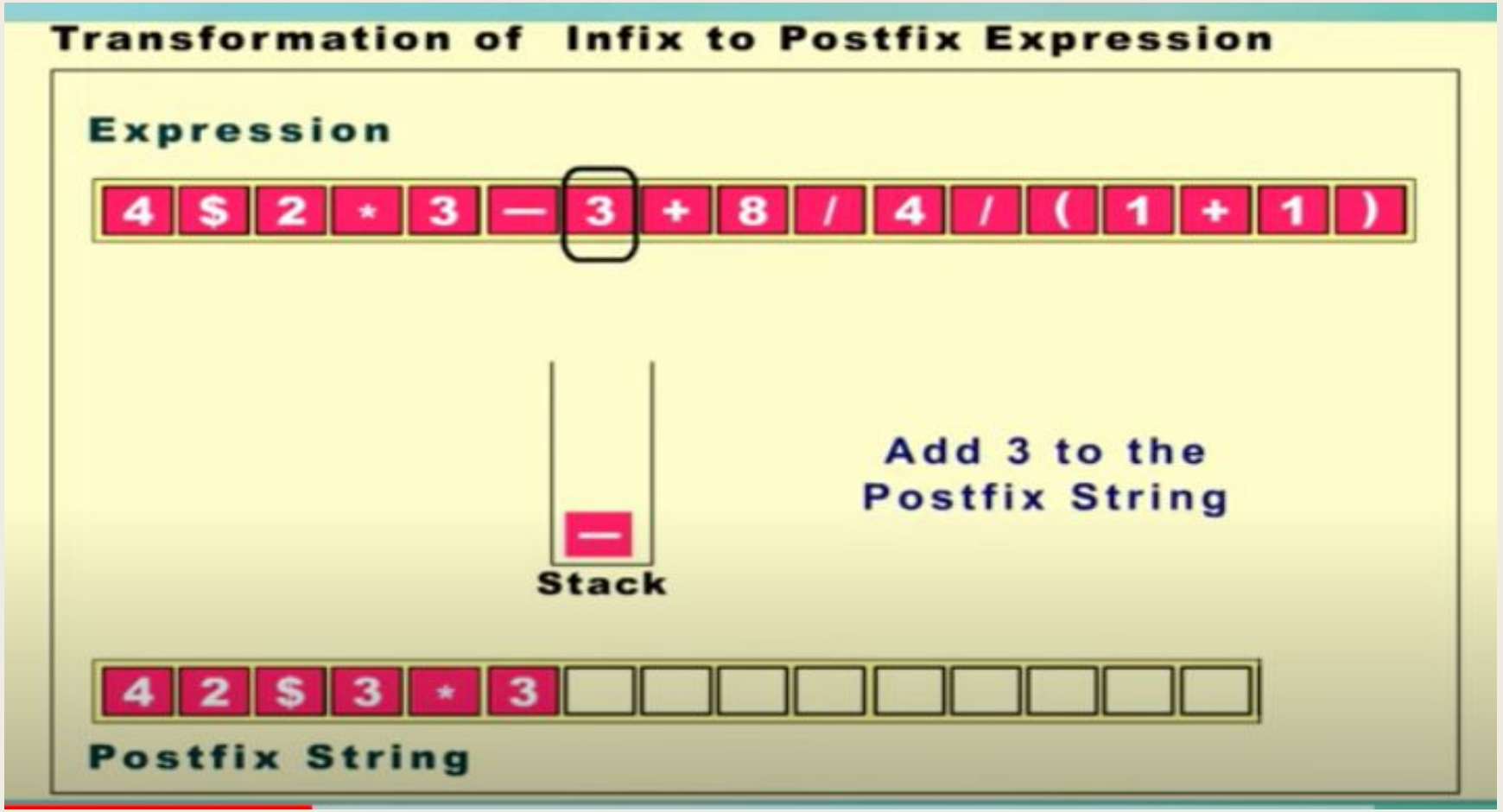

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
*
Stack
Operator scanned is —
4 2 $ 3
Postfix String

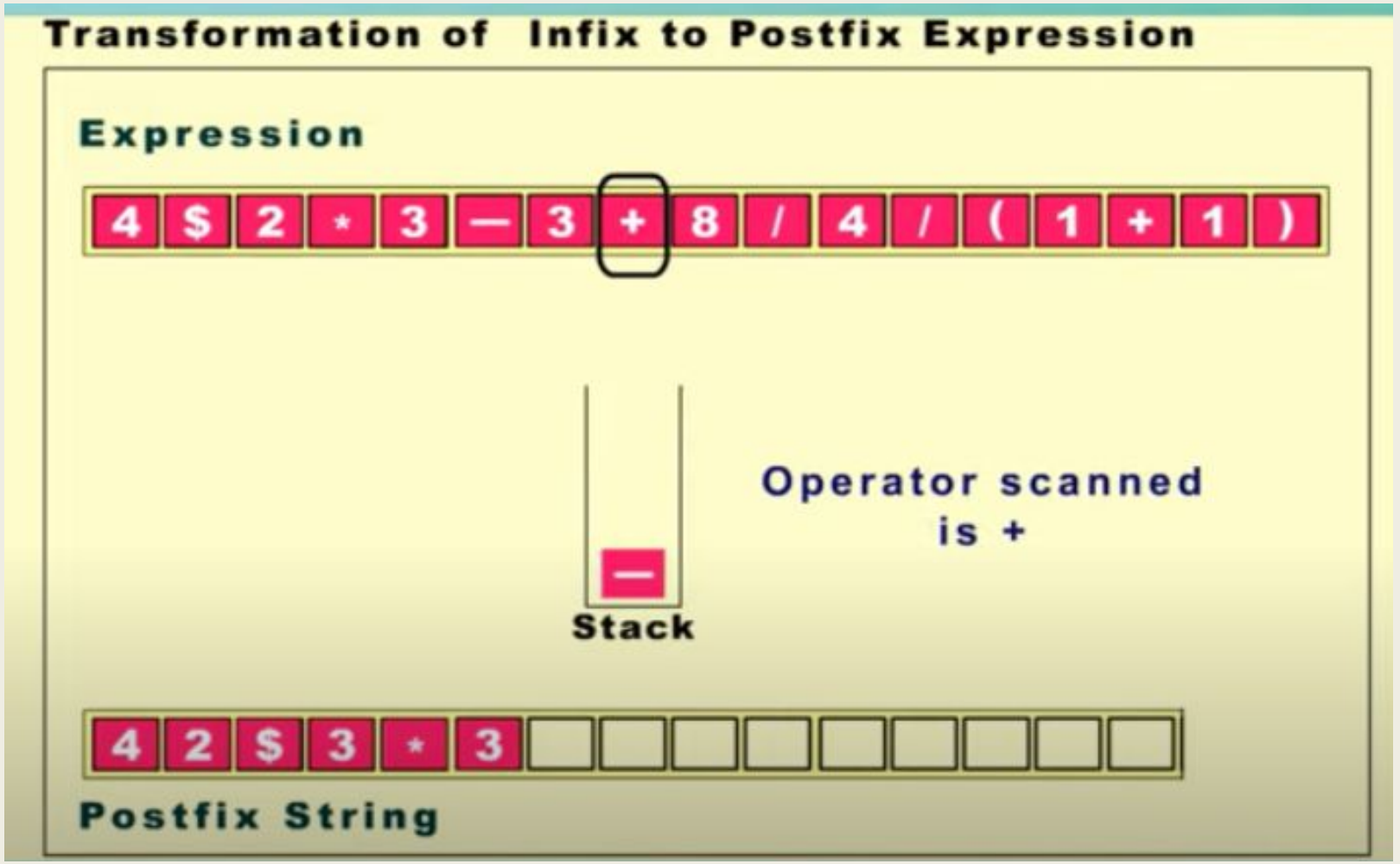

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
*
Stack
Popping *
from Stack
4 2 $ 3
Postfix String

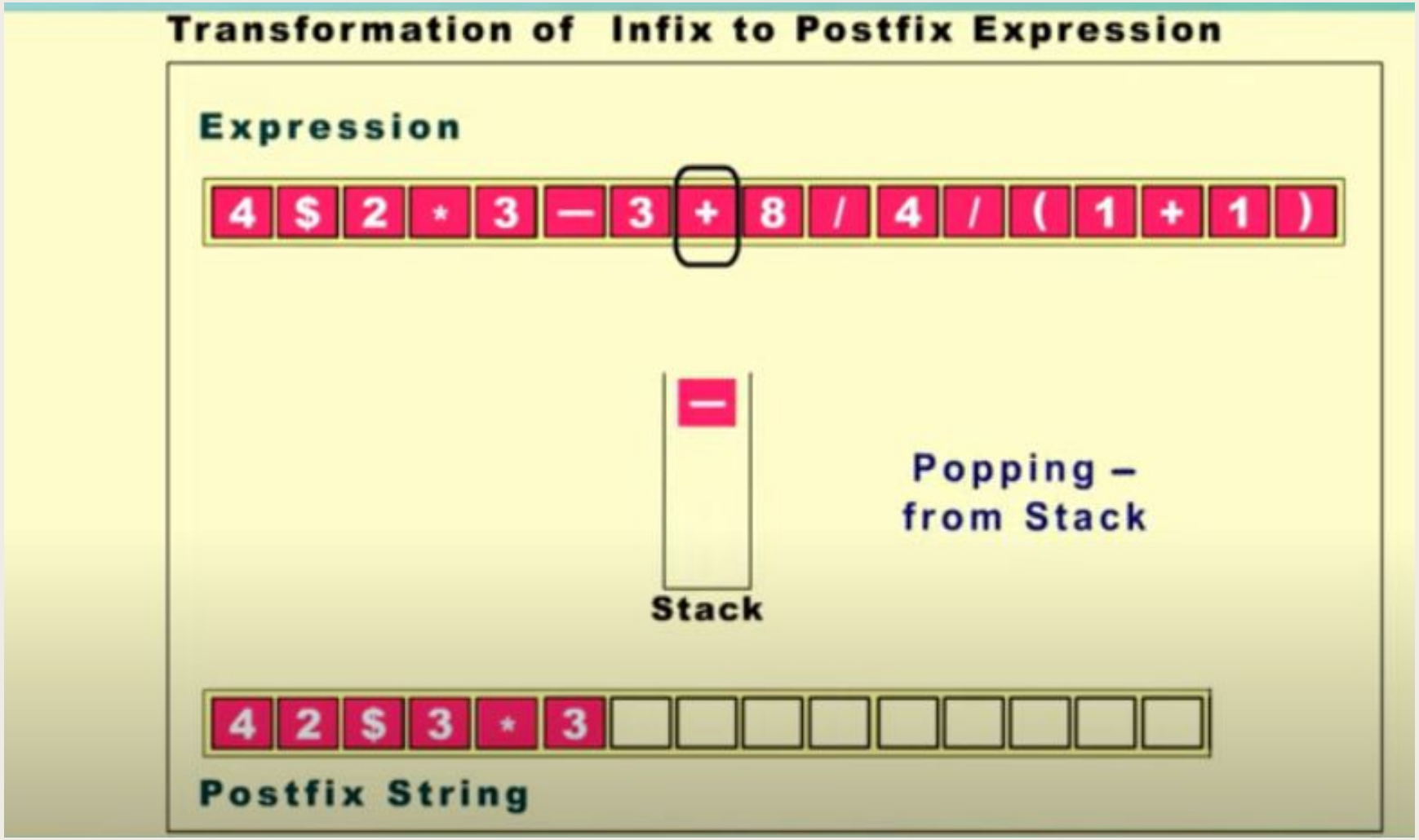

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
*
Stack
As hierarchy of * is greater than — Adding * to the Postfix String
4 2 $ 3
Postfix String

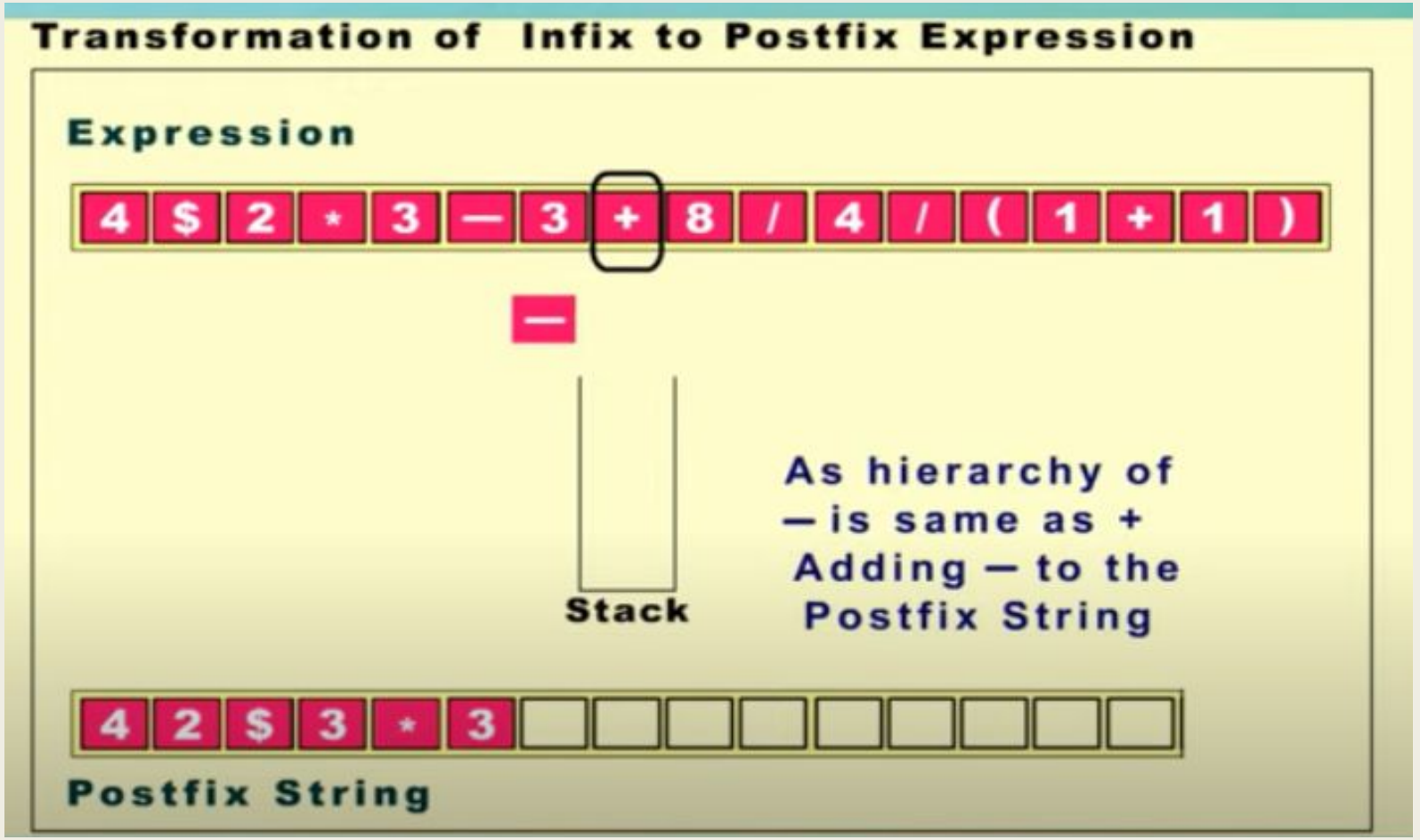

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
—
Stack
Push —
on Stack
4 2 $ 3 *
Postfix String

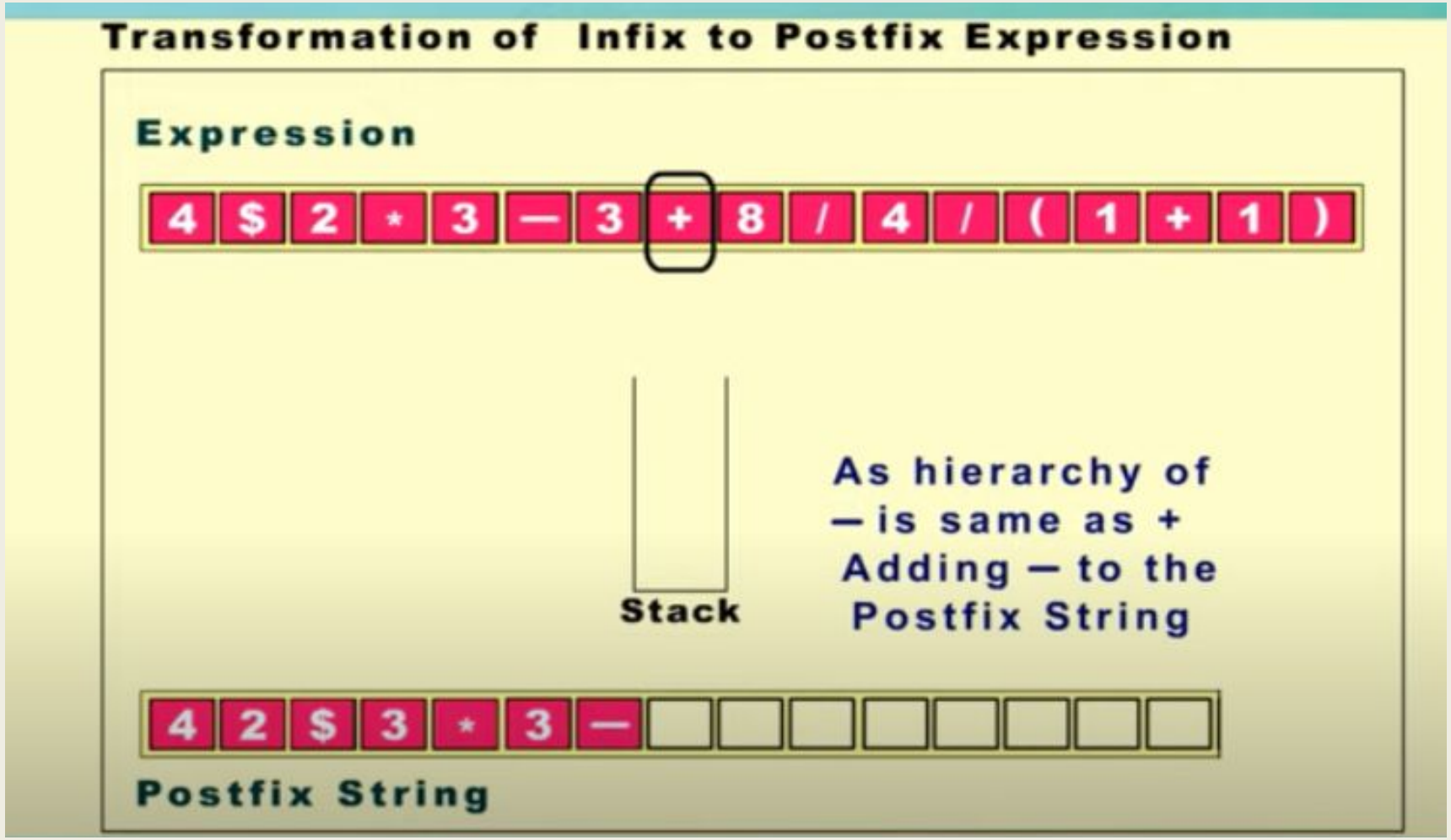

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
—
Stack
Digit scanned
is 3
4 2 $ 3 *
Postfix String

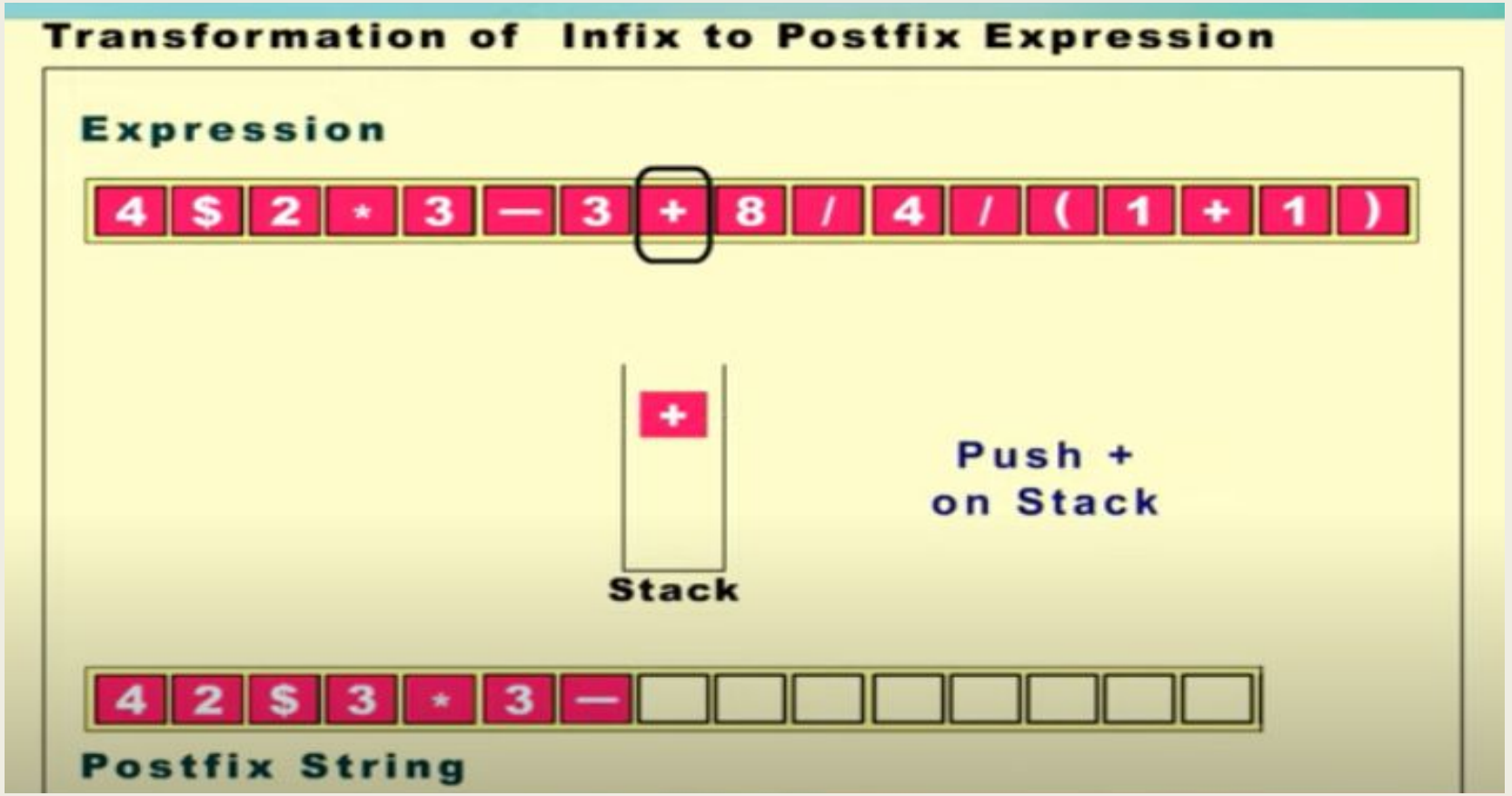

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
—
Stack
Add 3 to the
Postfix String
4 2 $ 3 * 3
Postfix String

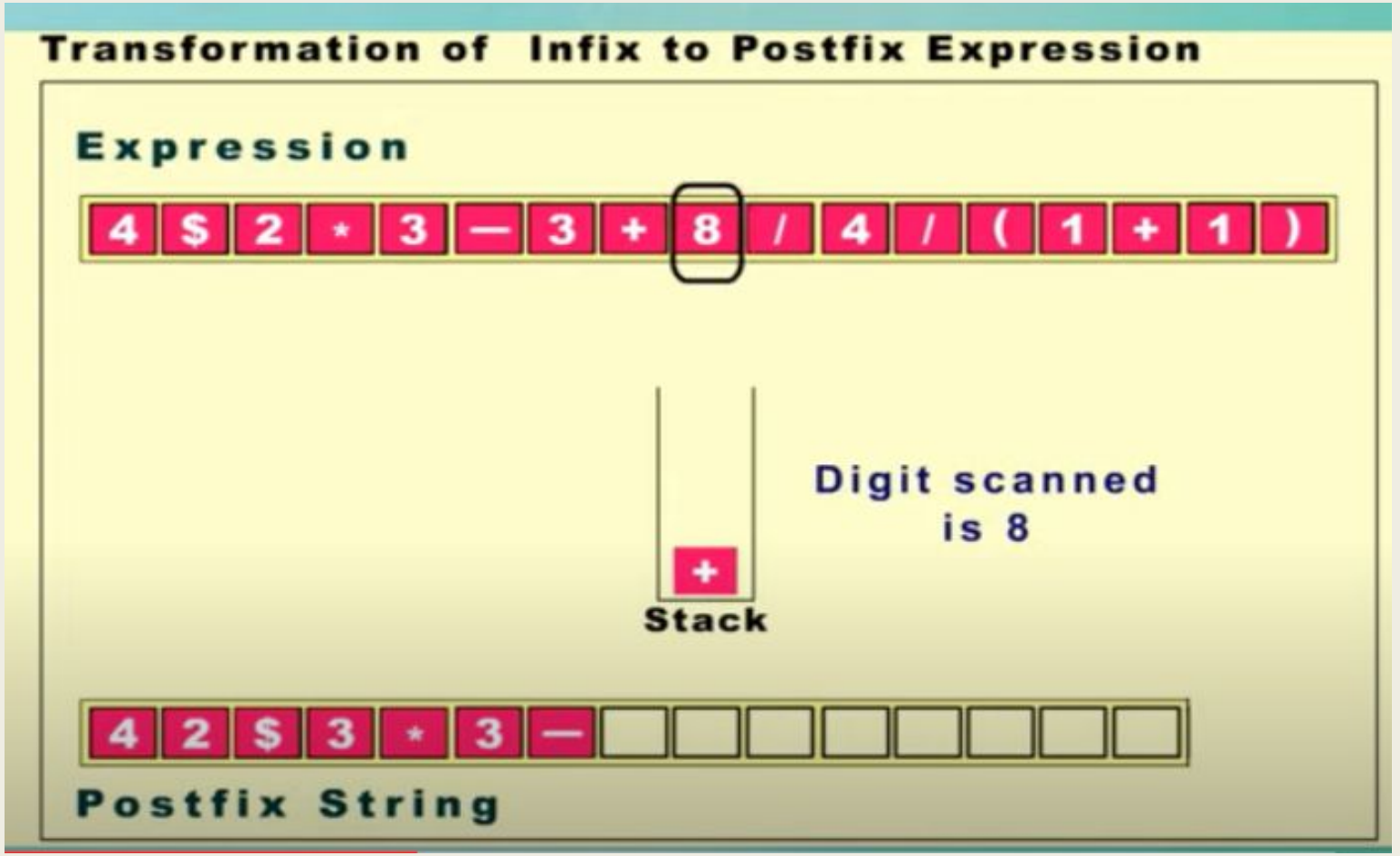

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 – 3 + 8 / 4 / (1 + 1)
Operator scanned is +
–
Stack
4 2 $ 3 * 3
Postfix String

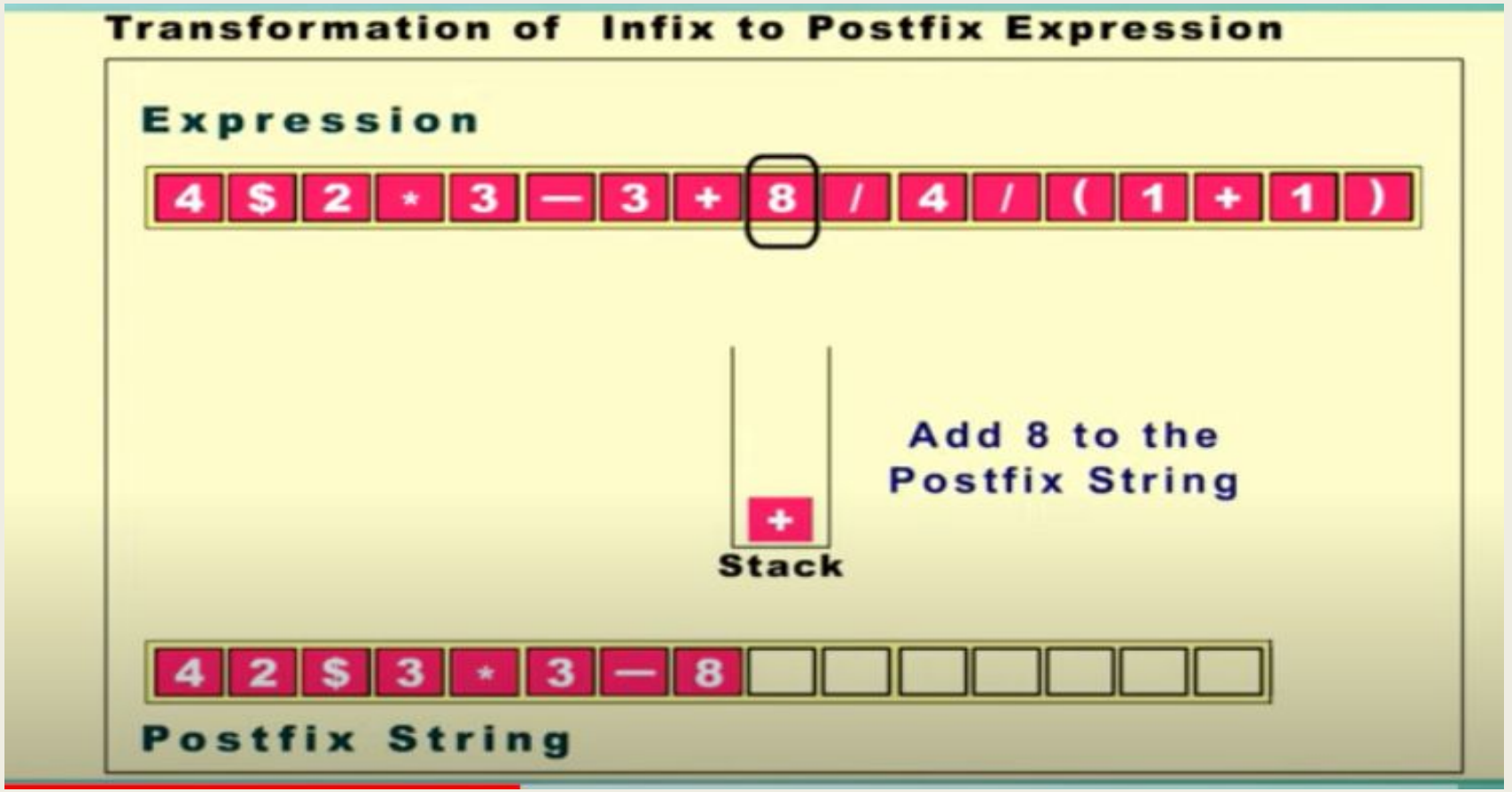

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
—
Stack
Popping –
from Stack
4 2 $ 3 * 3
Postfix String

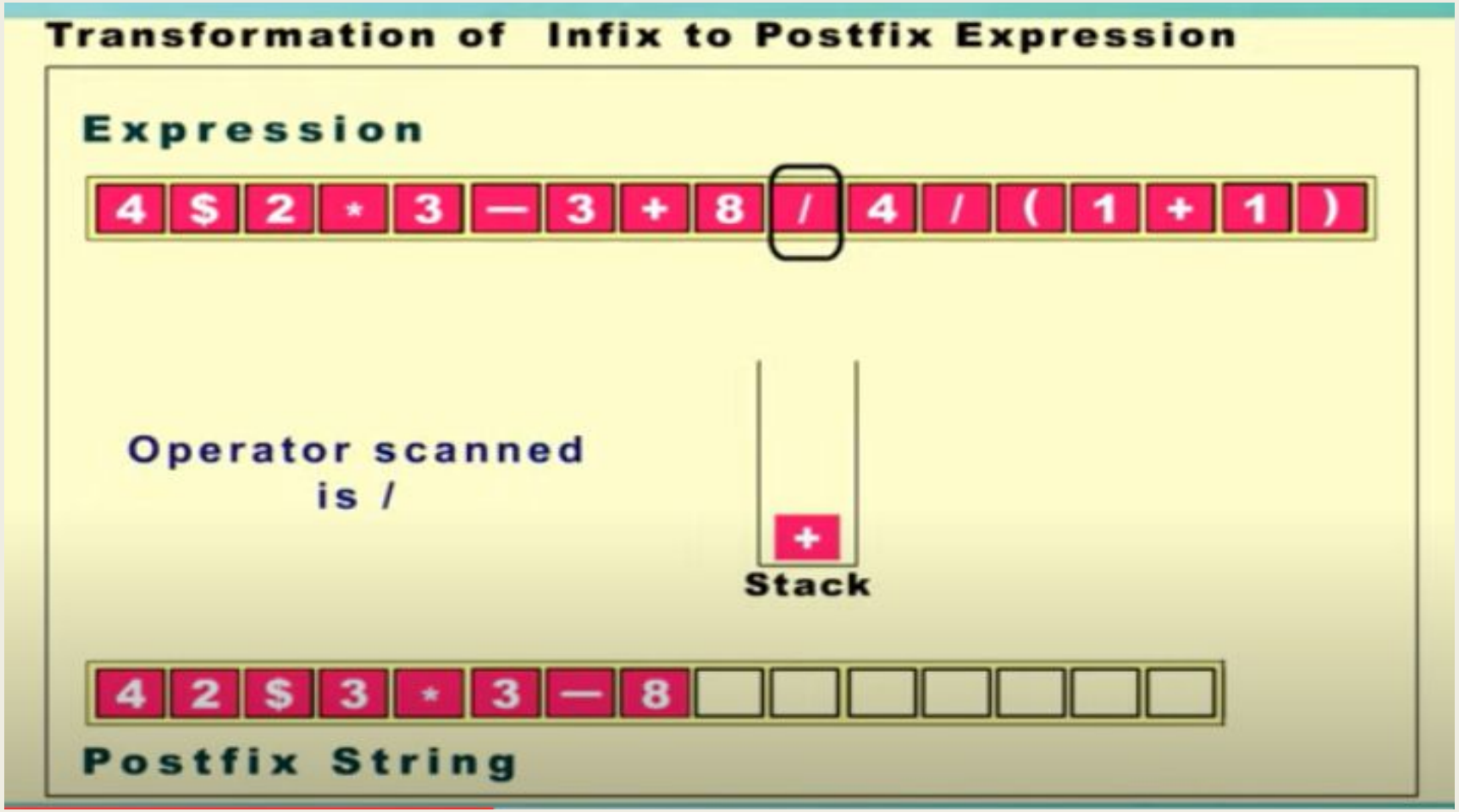

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
—
Stack
As hierarchy of
— is same as +
Adding — to the
Postfix String
4 2 $ 3 * 3
Postfix String

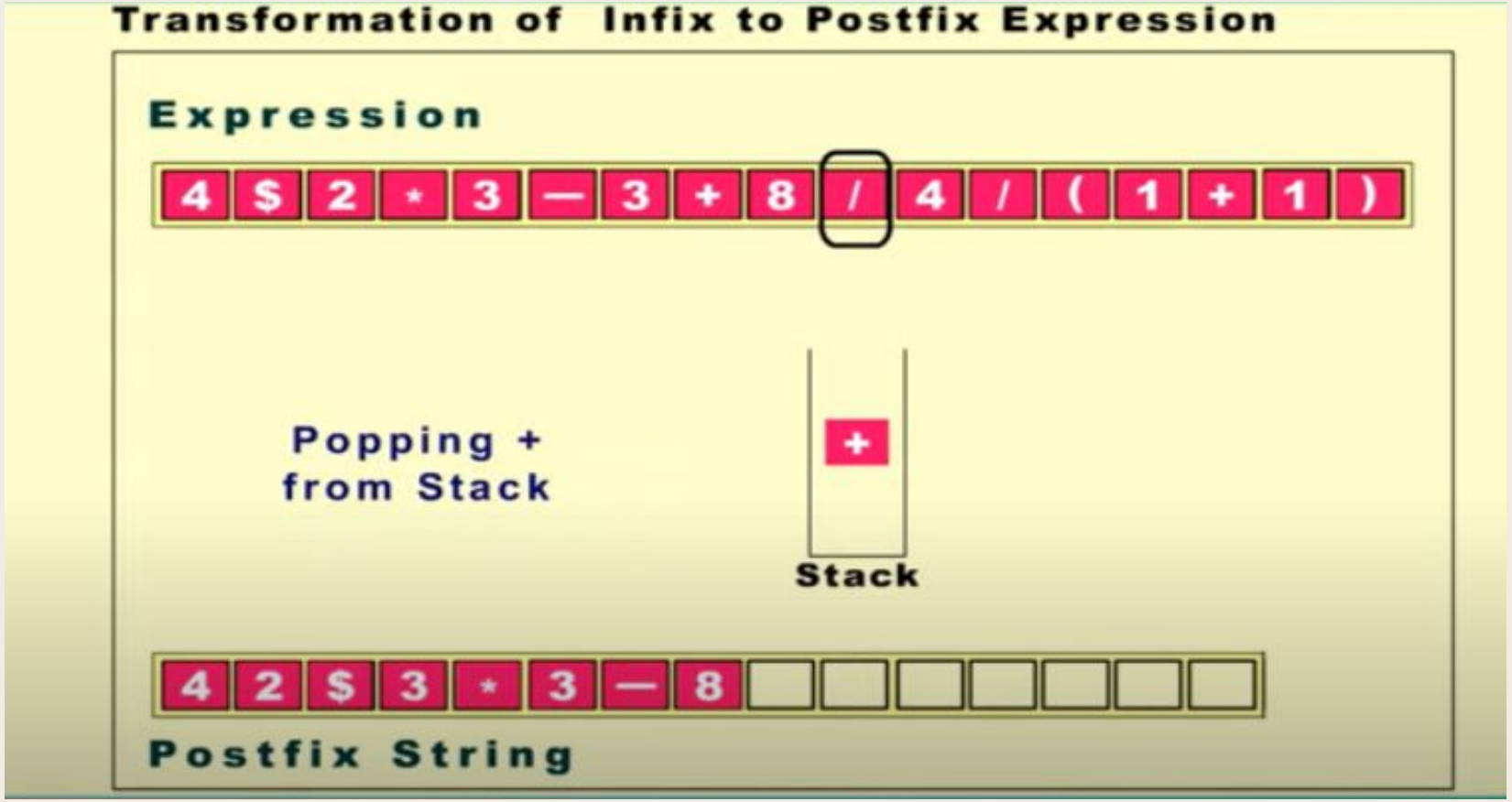

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Stack
As hierarchy of
— is same as +
Adding — to the
Postfix String
4 2 $ 3 * 3 —
Postfix String

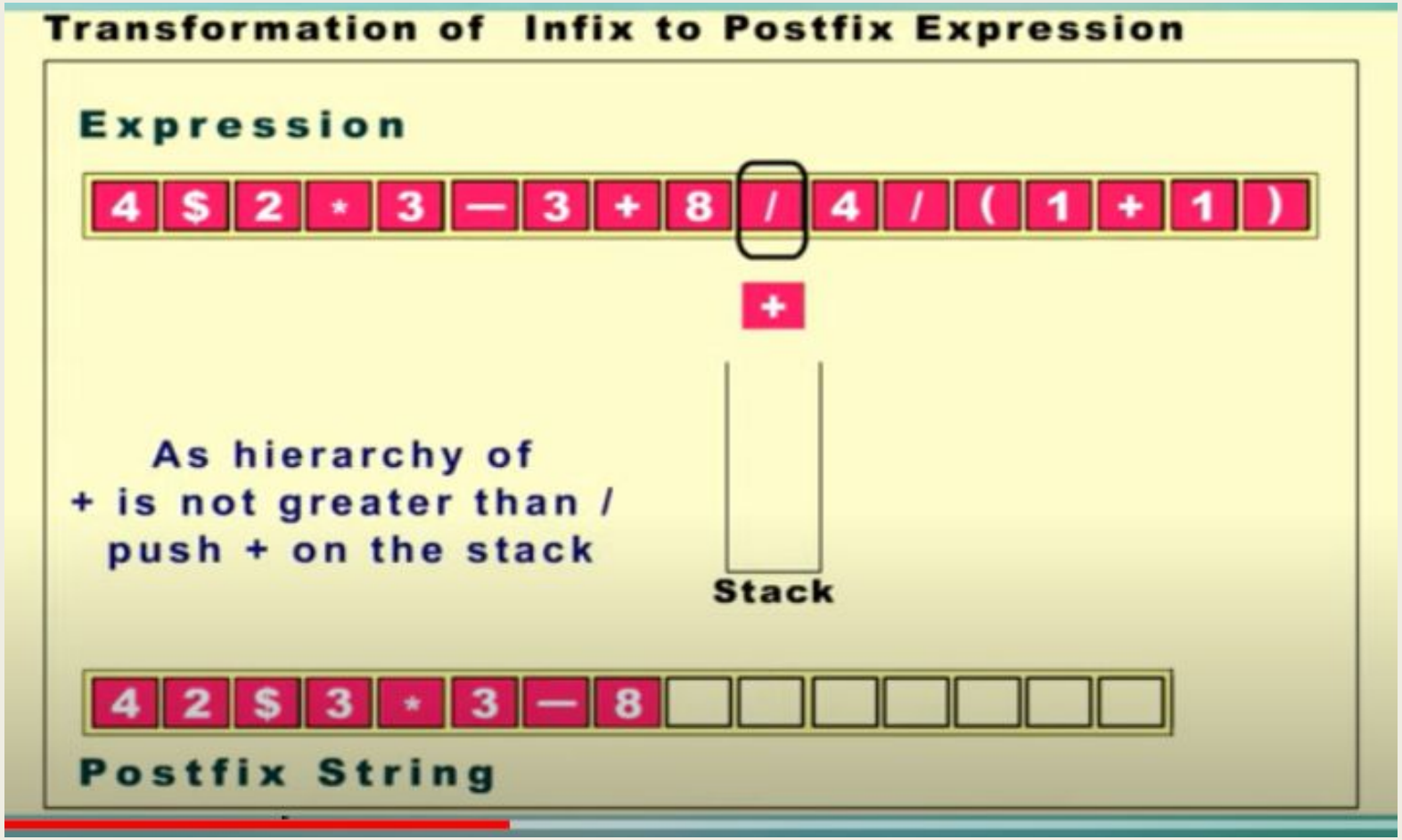

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
+
Stack
Push +
on Stack
4 2 $ 3 * 3 —
Postfix String

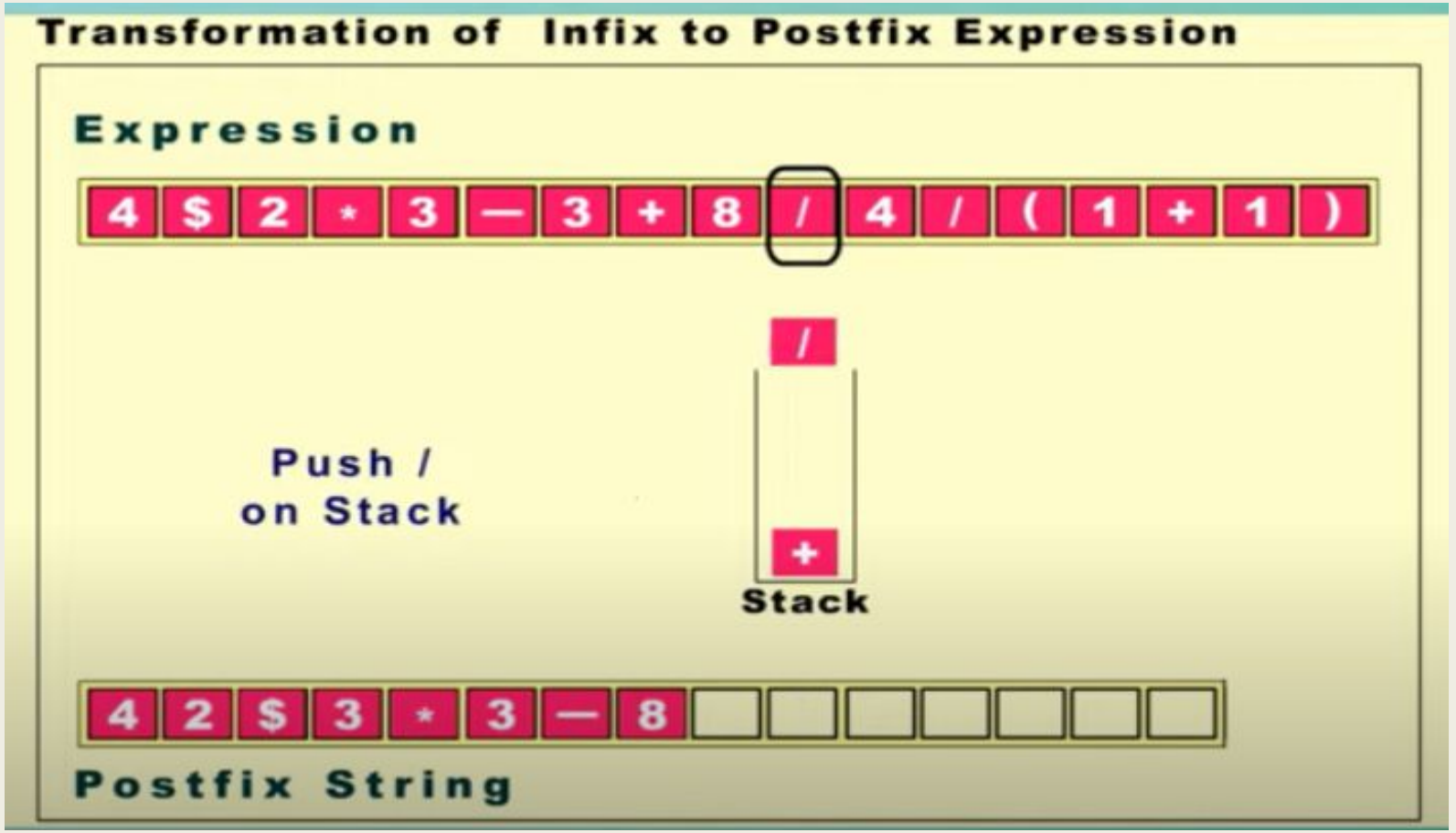

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
+
Stack
Digit scanned
is 8
4 2 $ 3 * 3 —
Postfix String

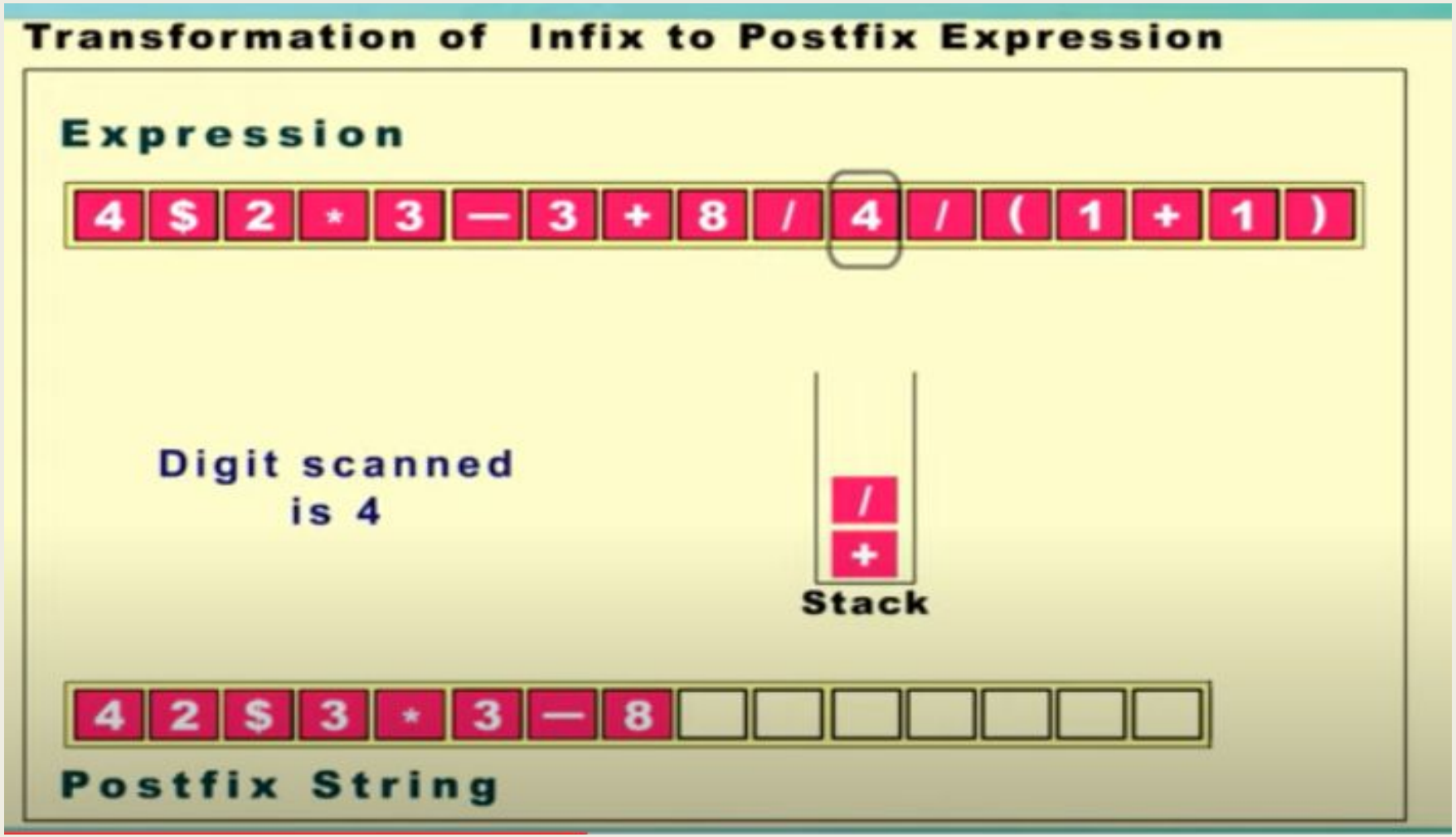

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
+
Stack
Add 8 to the
Postfix String
4 2 $ 3 * 3 — 8
Postfix String

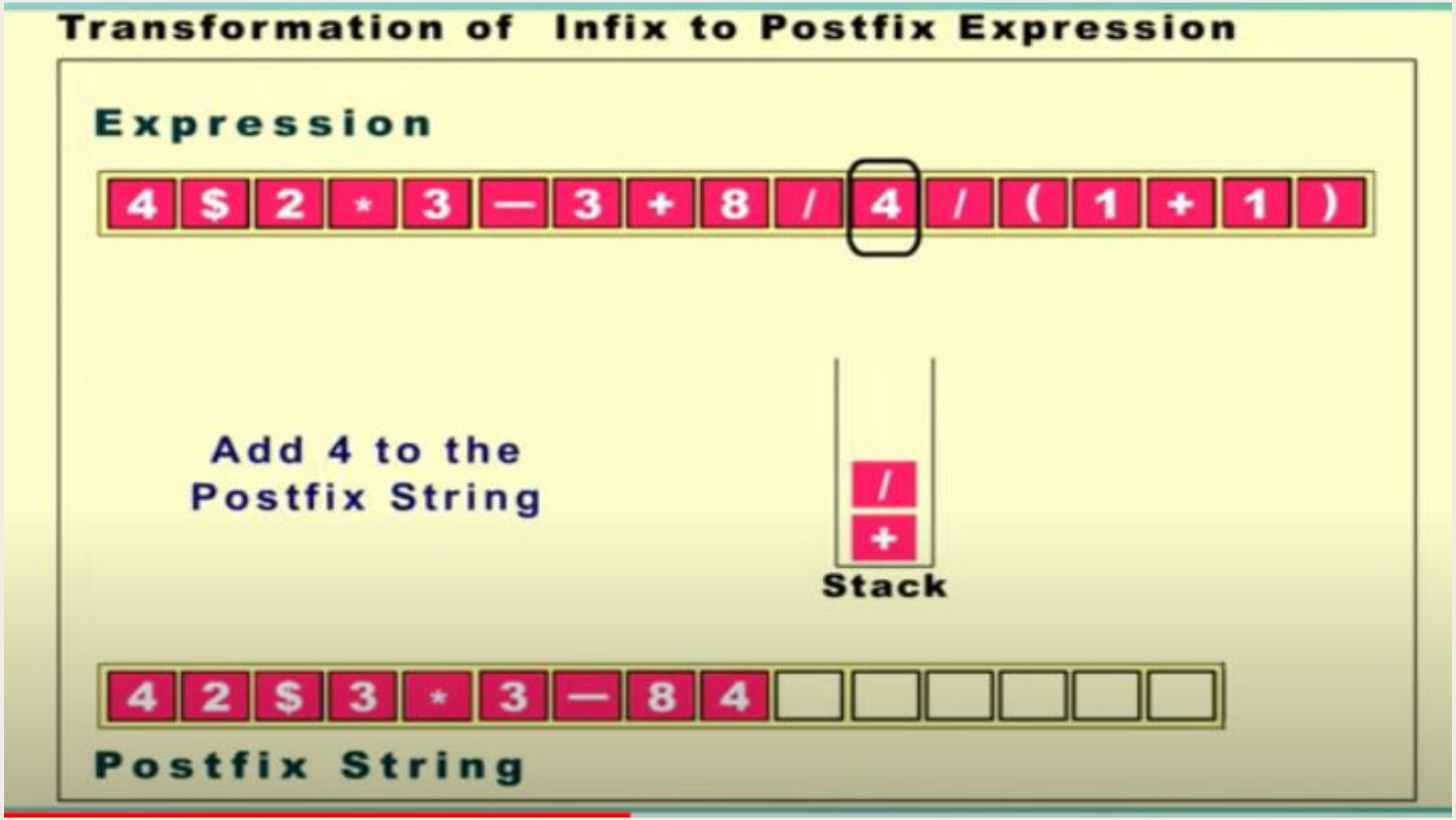

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Operator scanned is /
+
Stack
4 2 $ 3 * 3 — 8
Postfix String

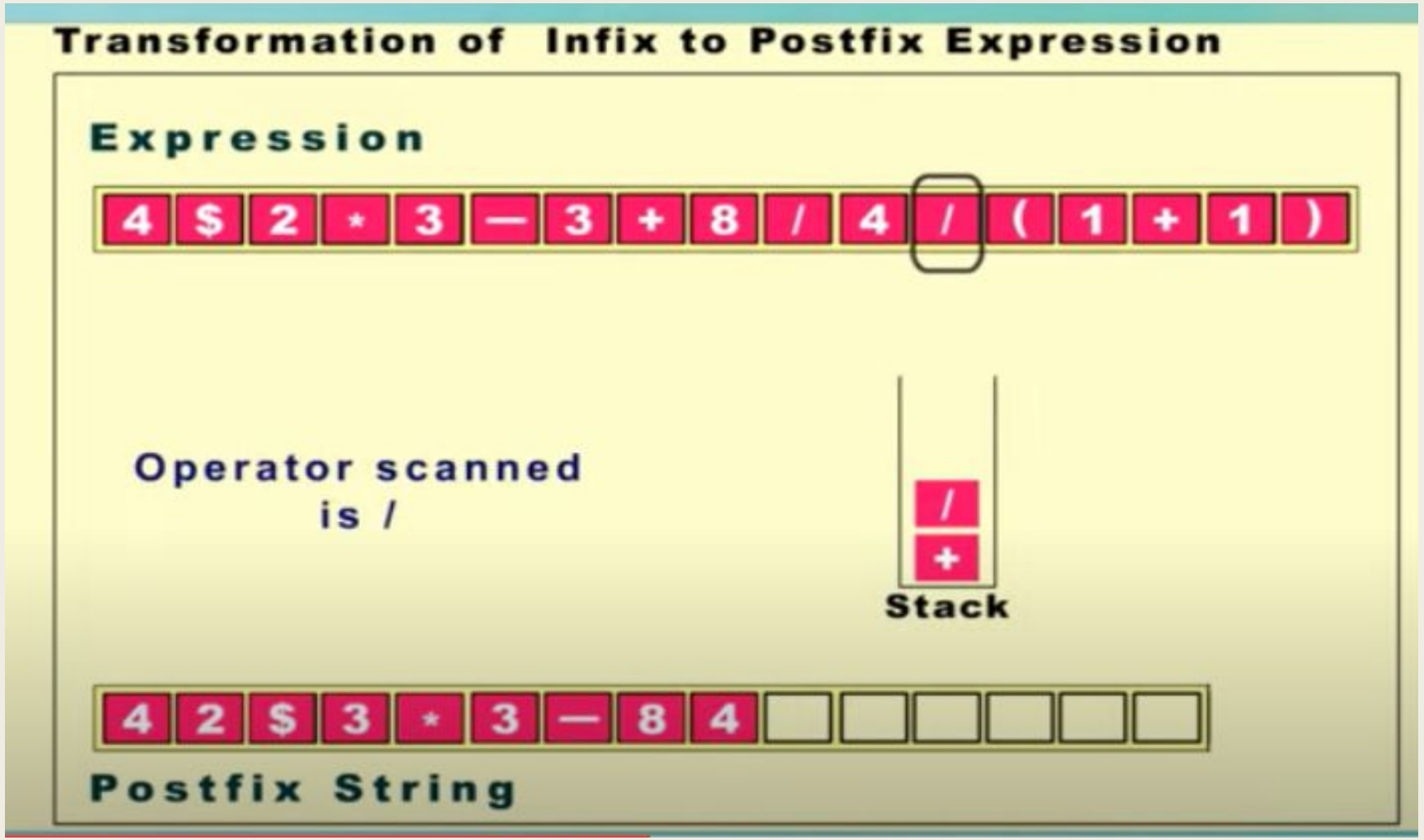

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Popping +
from Stack
+
Stack
4 2 $ 3 * 3 — 8
Postfix String

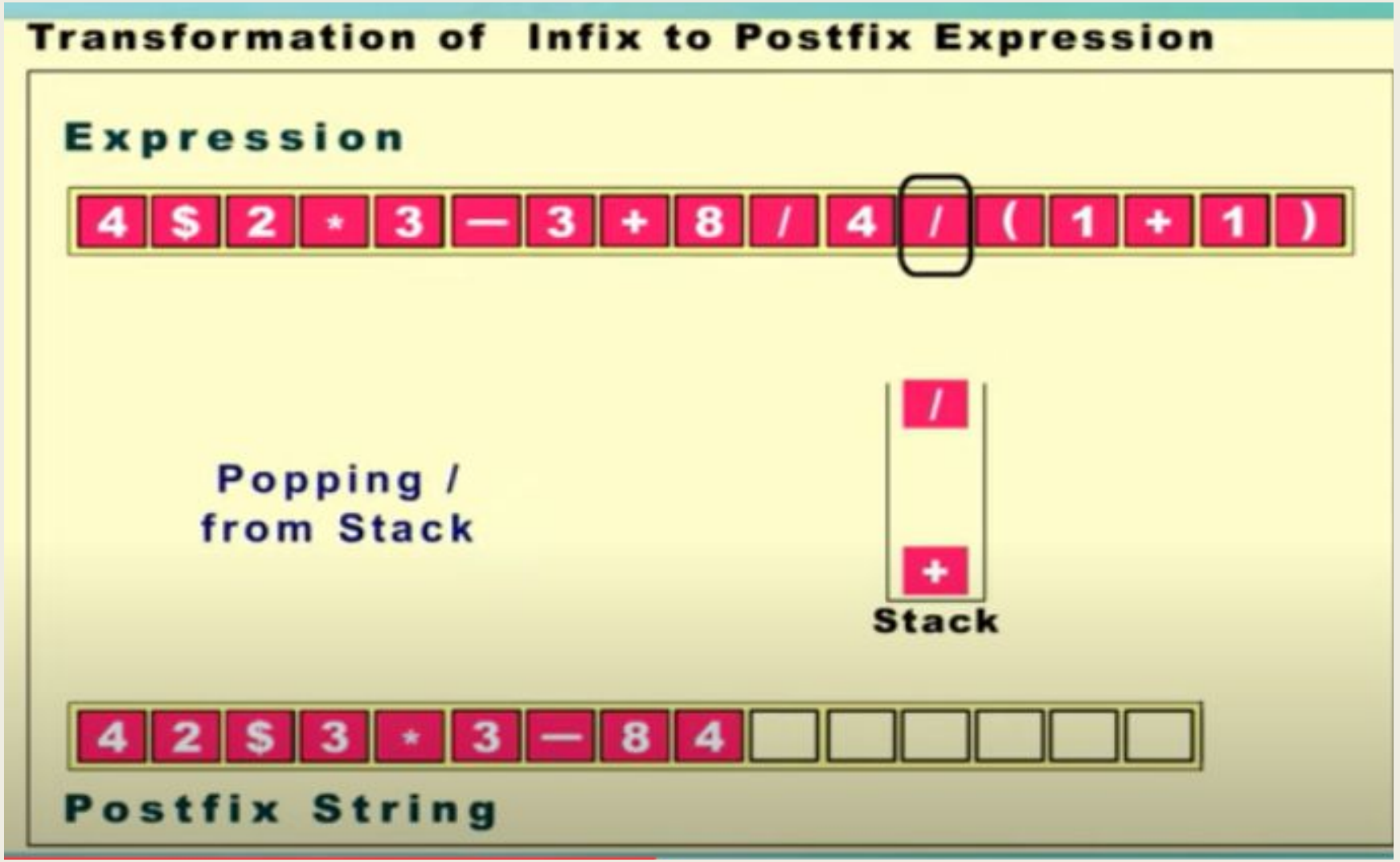

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
+
As hierarchy of
+ is not greater than /
push + on the stack
Stack
4 2 $ 3 * 3 — 8
Postfix String

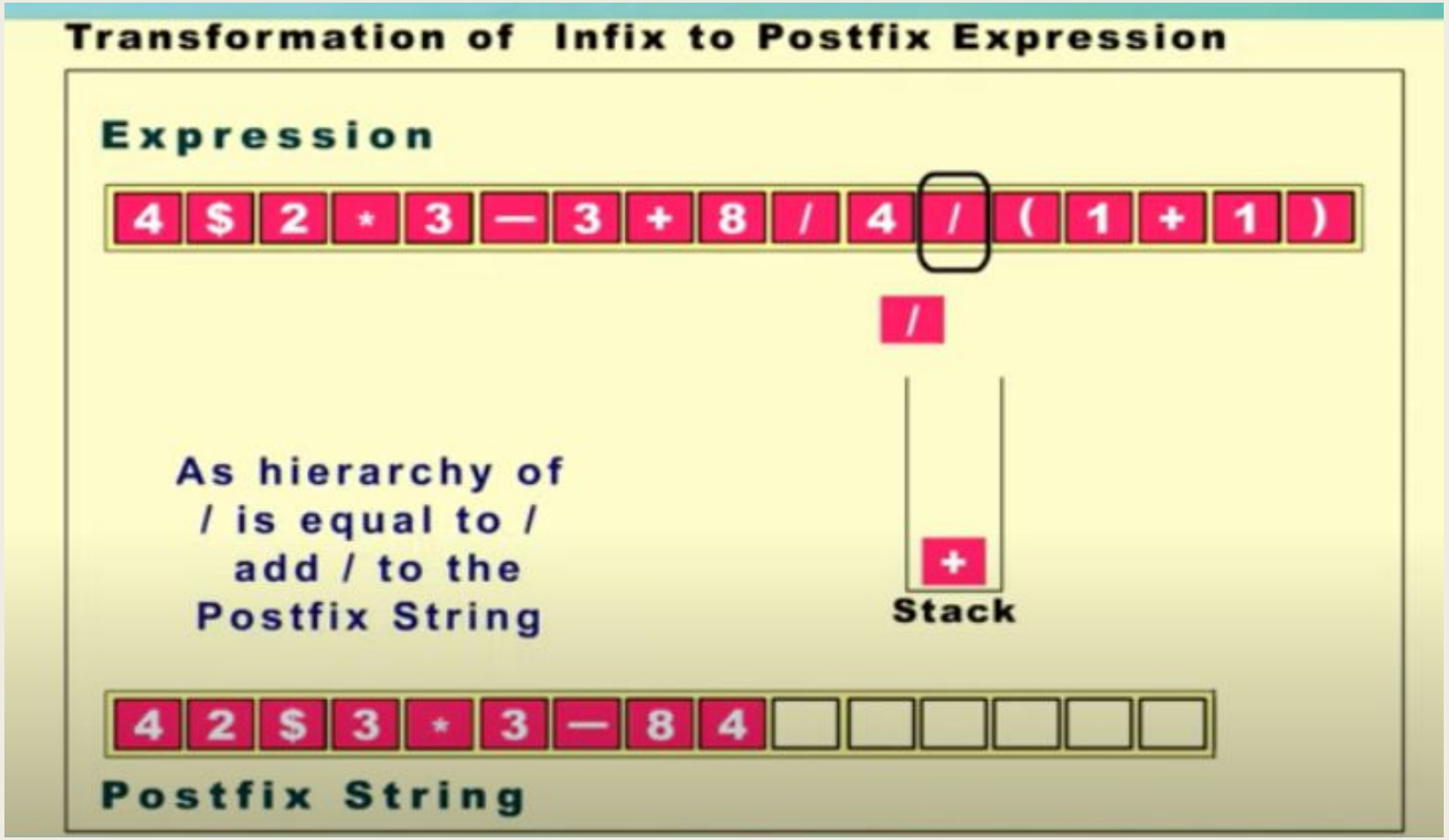

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
/
+
Stack
Push /
on Stack
4 2 $ 3 * 3 — 8
Postfix String

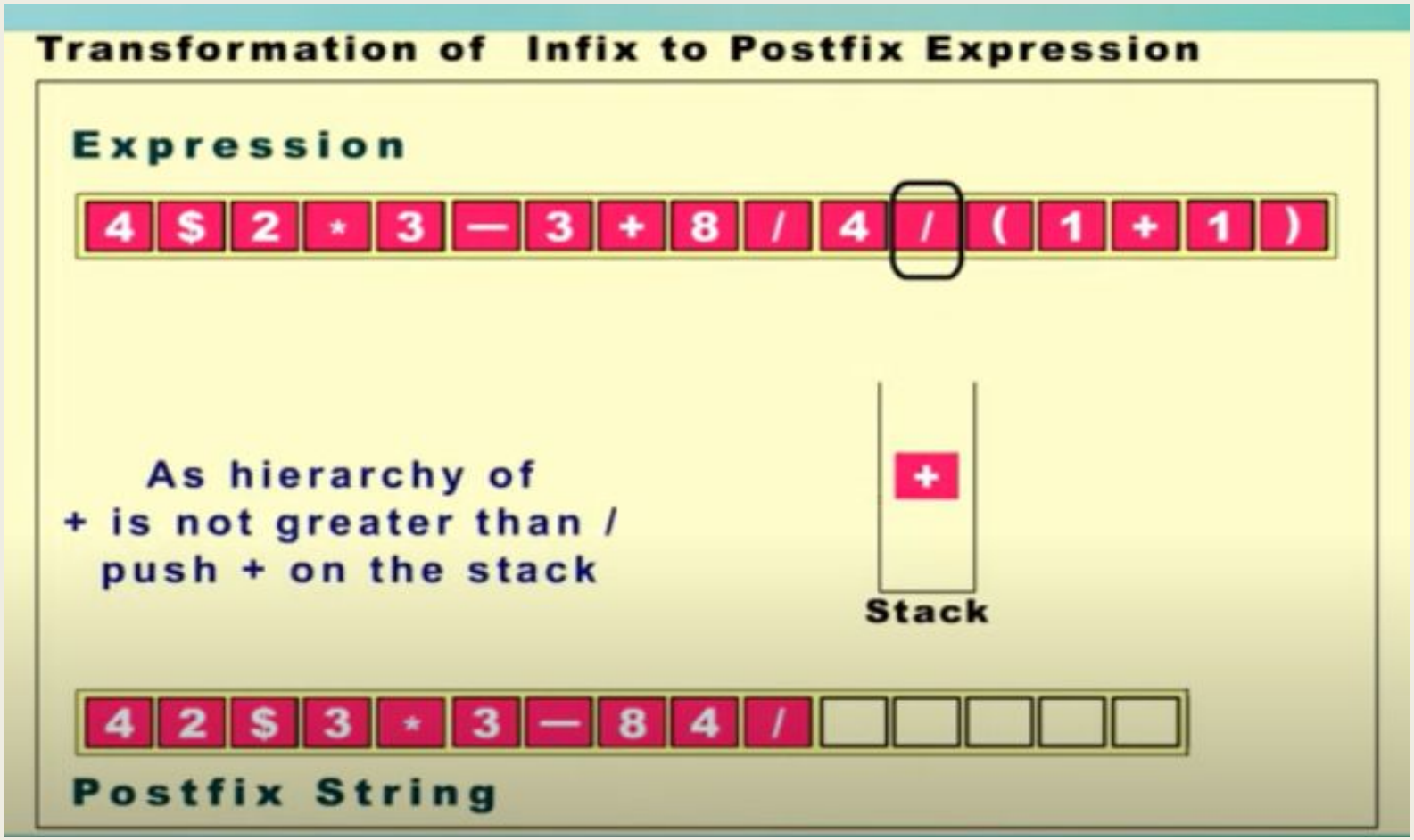

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Digit scanned
is 4
/
+
Stack
4 2 $ 3 * 3 — 8
Postfix String

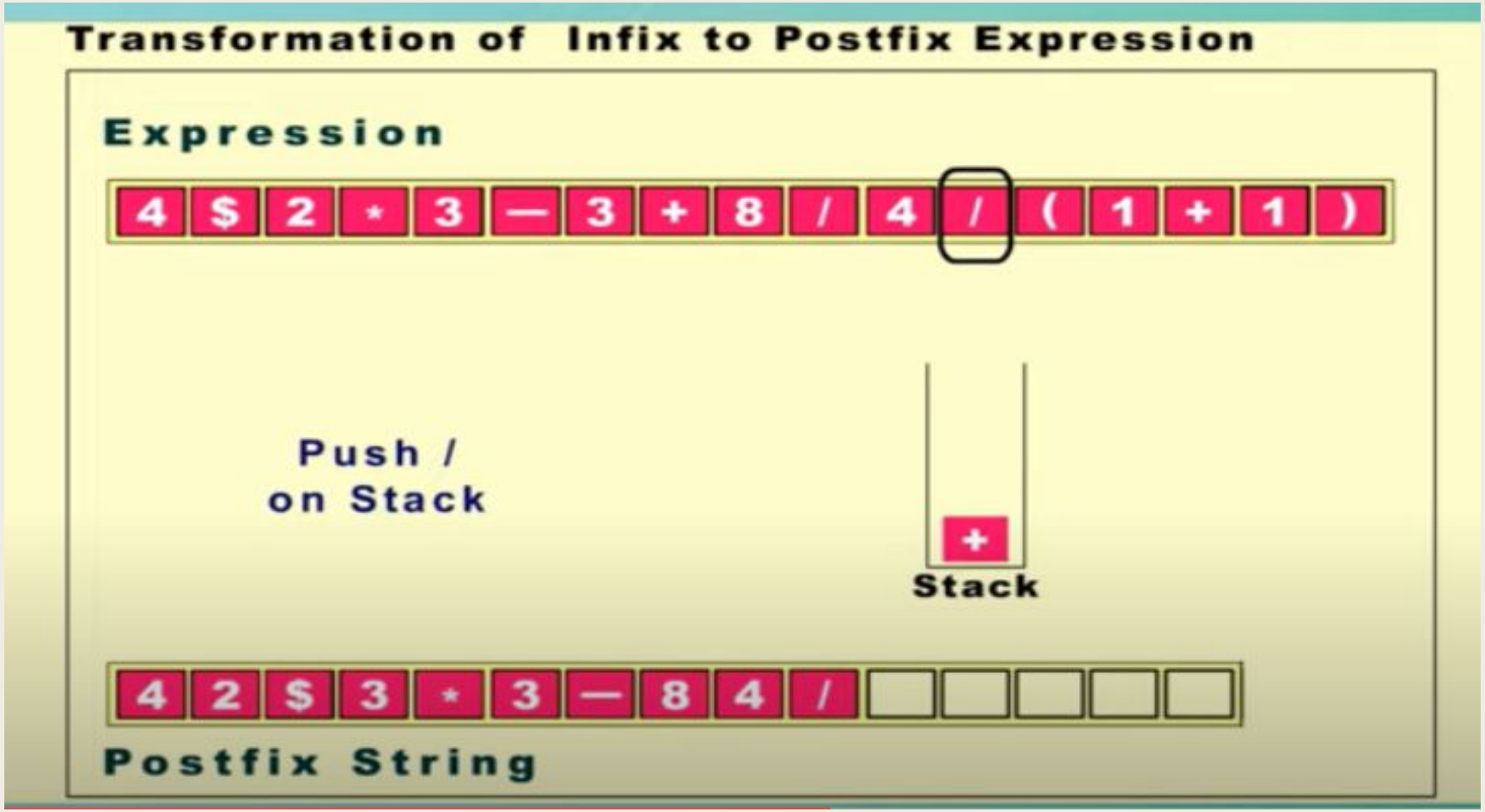

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add 4 to the
Postfix String
/
+
Stack
4 2 $ 3 * 3 — 8 4
Postfix String

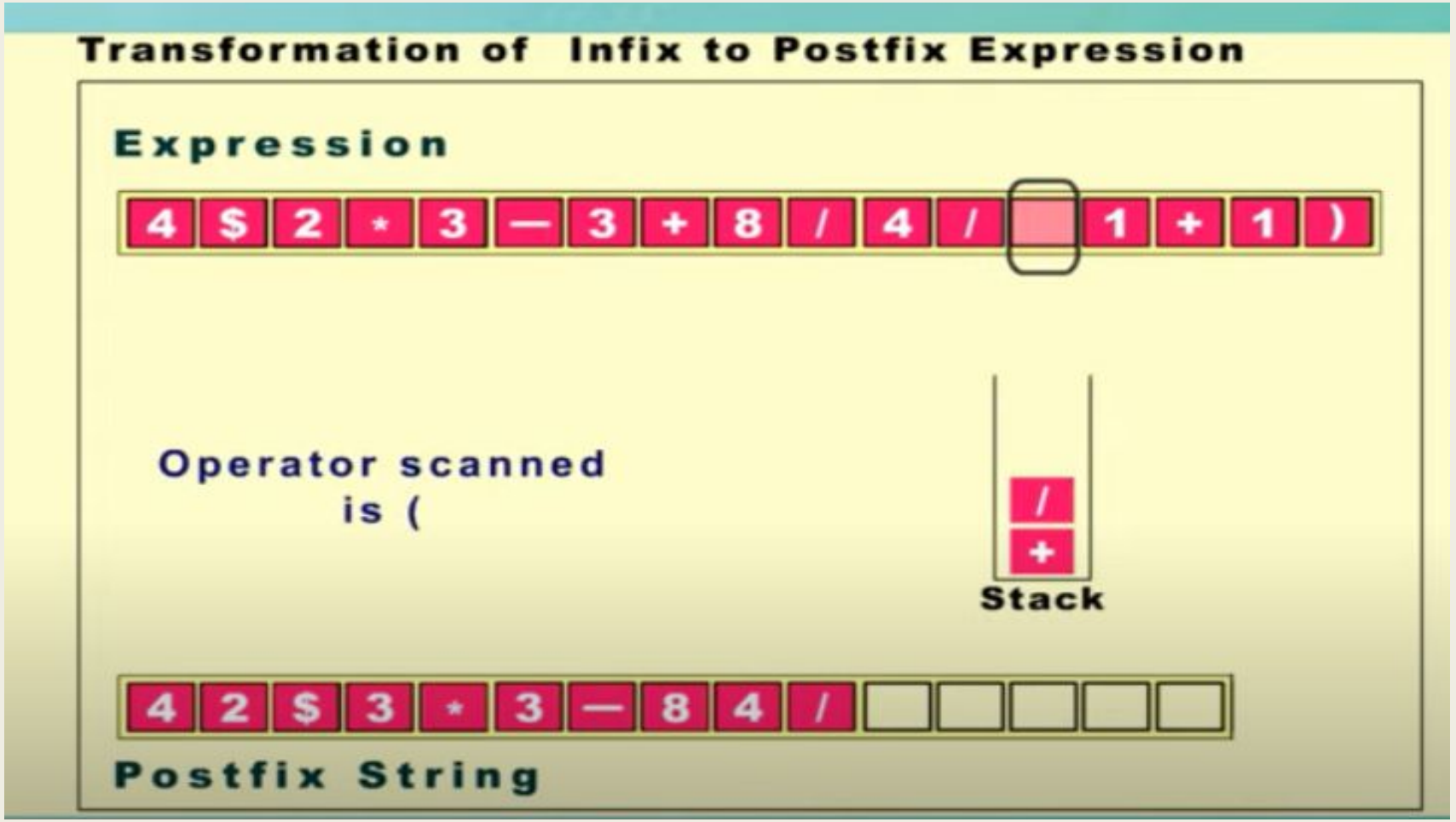

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Operator scanned is /
/
+
Stack
4 2 $ 3 * 3 — 8 4
Postfix String

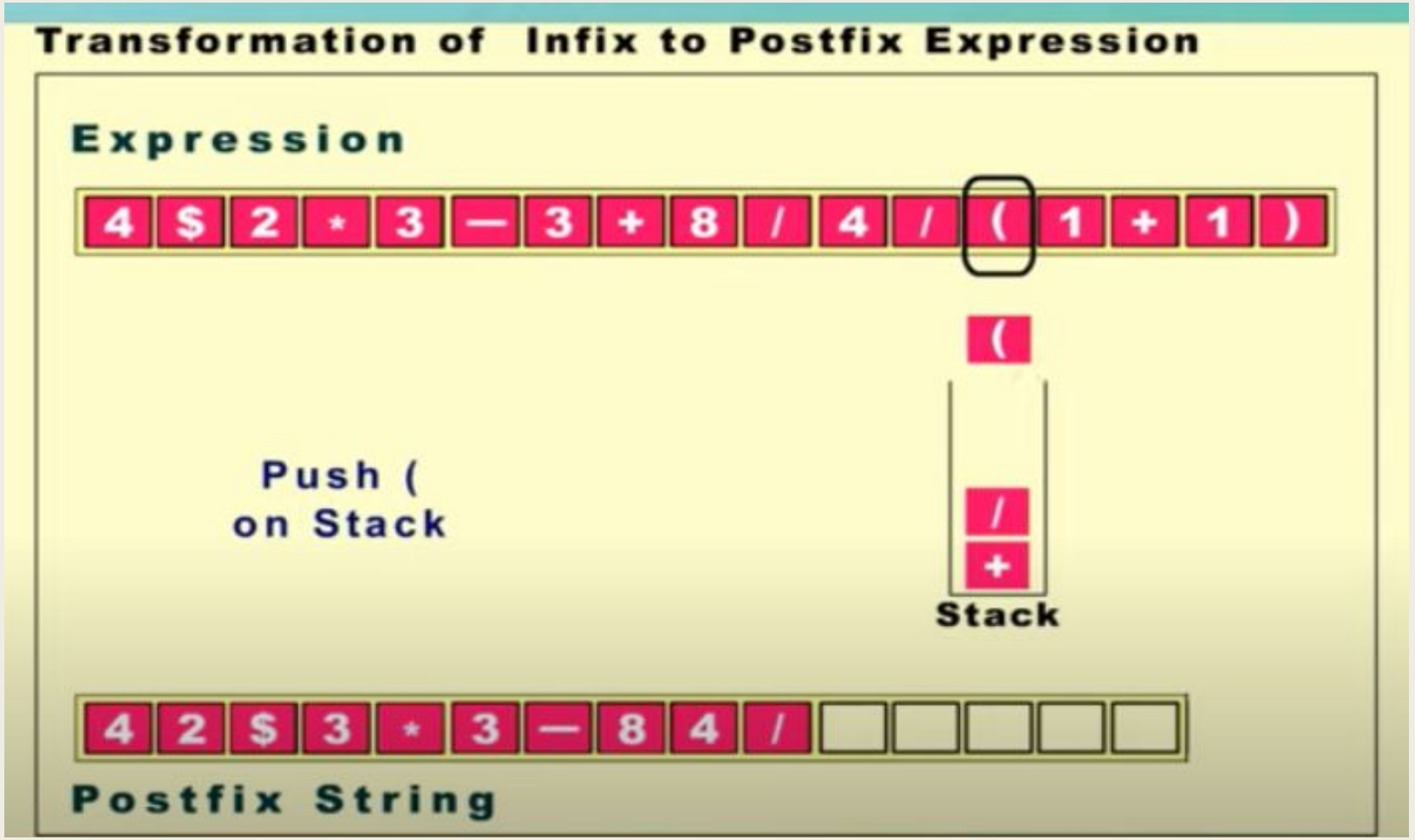

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Popping /
from Stack
/
+
Stack
4 2 $ 3 * 3 — 8 4
Postfix String

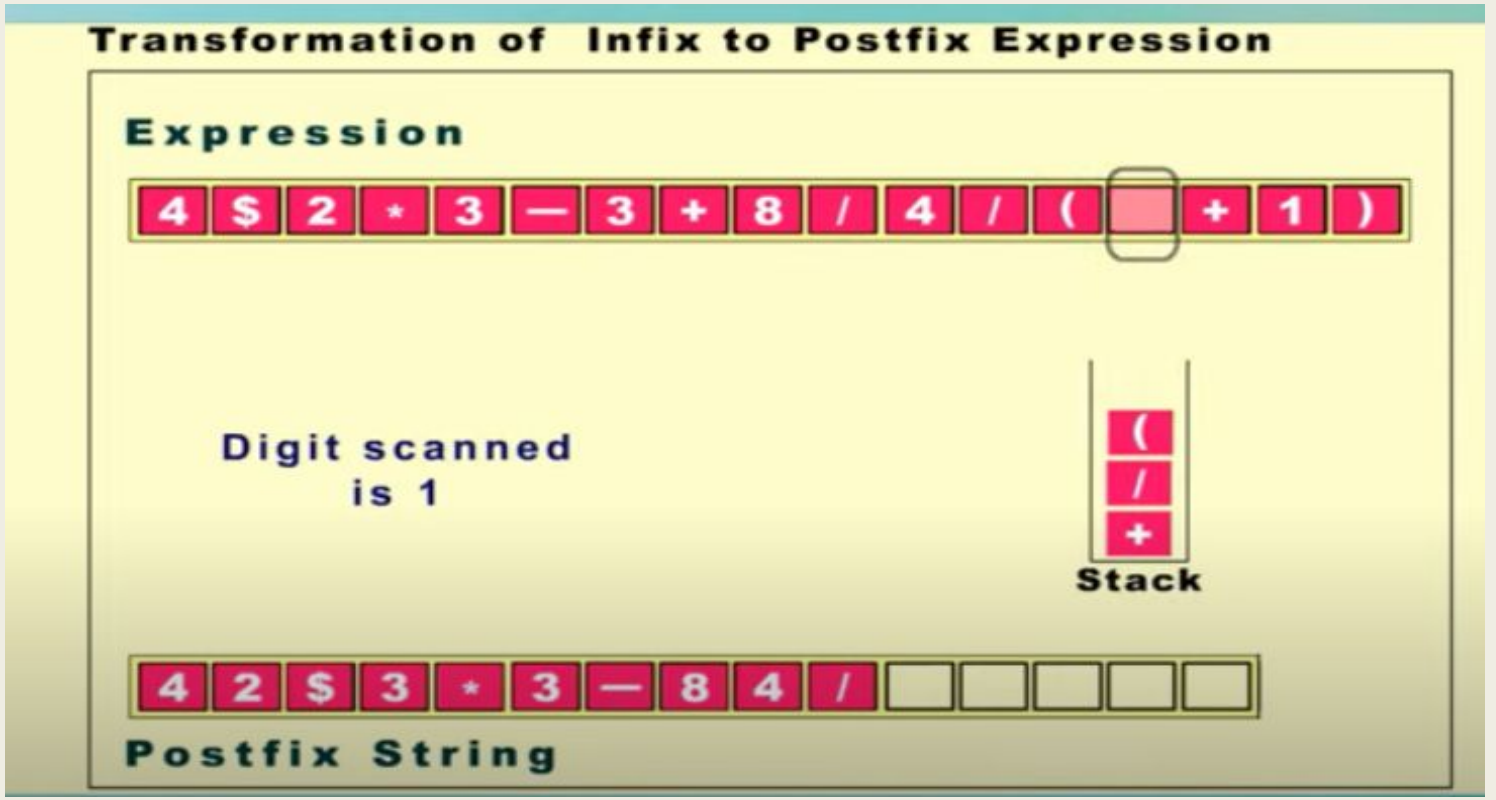

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
/
As hierarchy of / is equal to / add / to the Postfix String
+
Stack
4 2 $ 3 * 3 — 8 4
Postfix String

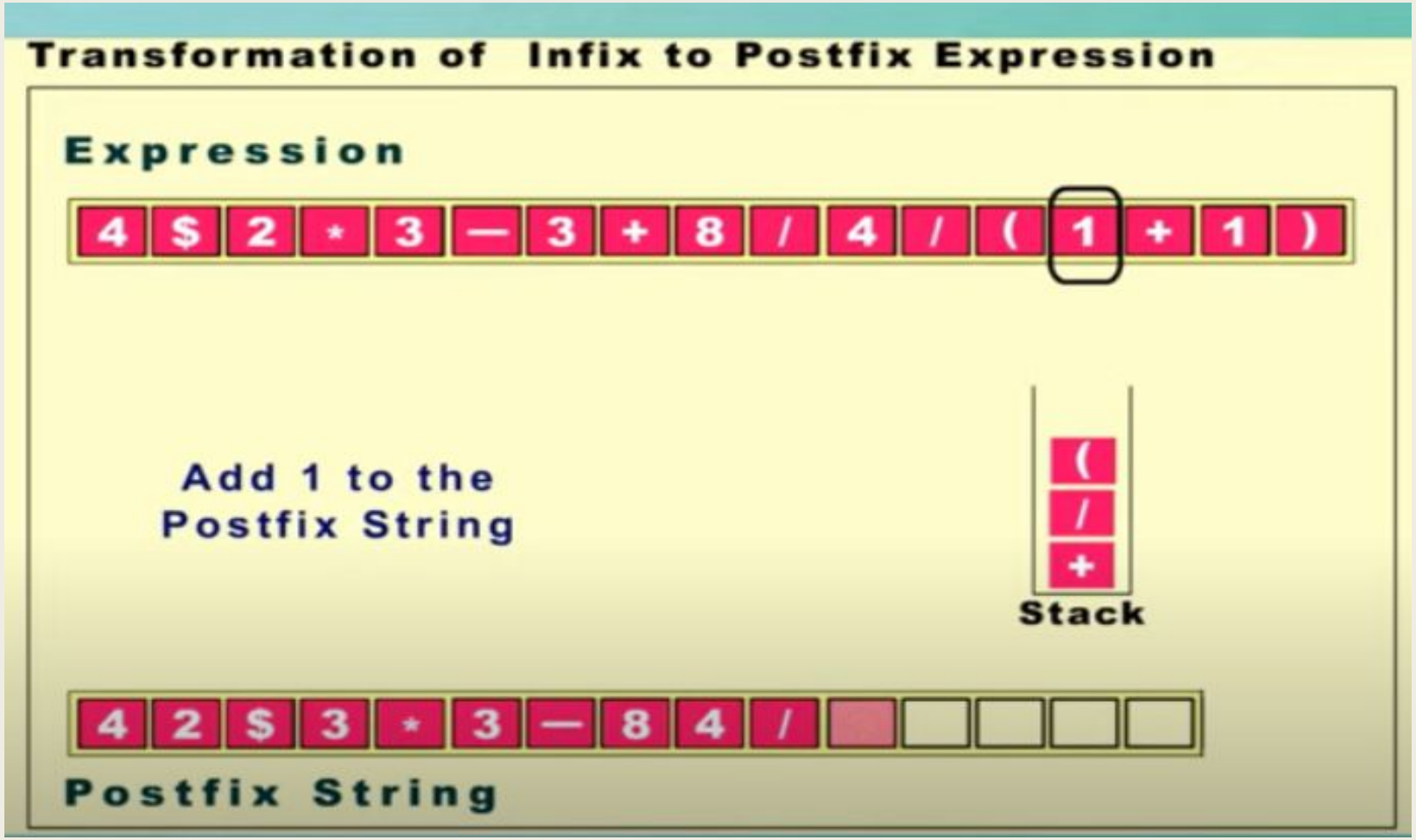

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
As hierarchy of
+ is not greater than /
push + on the stack
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

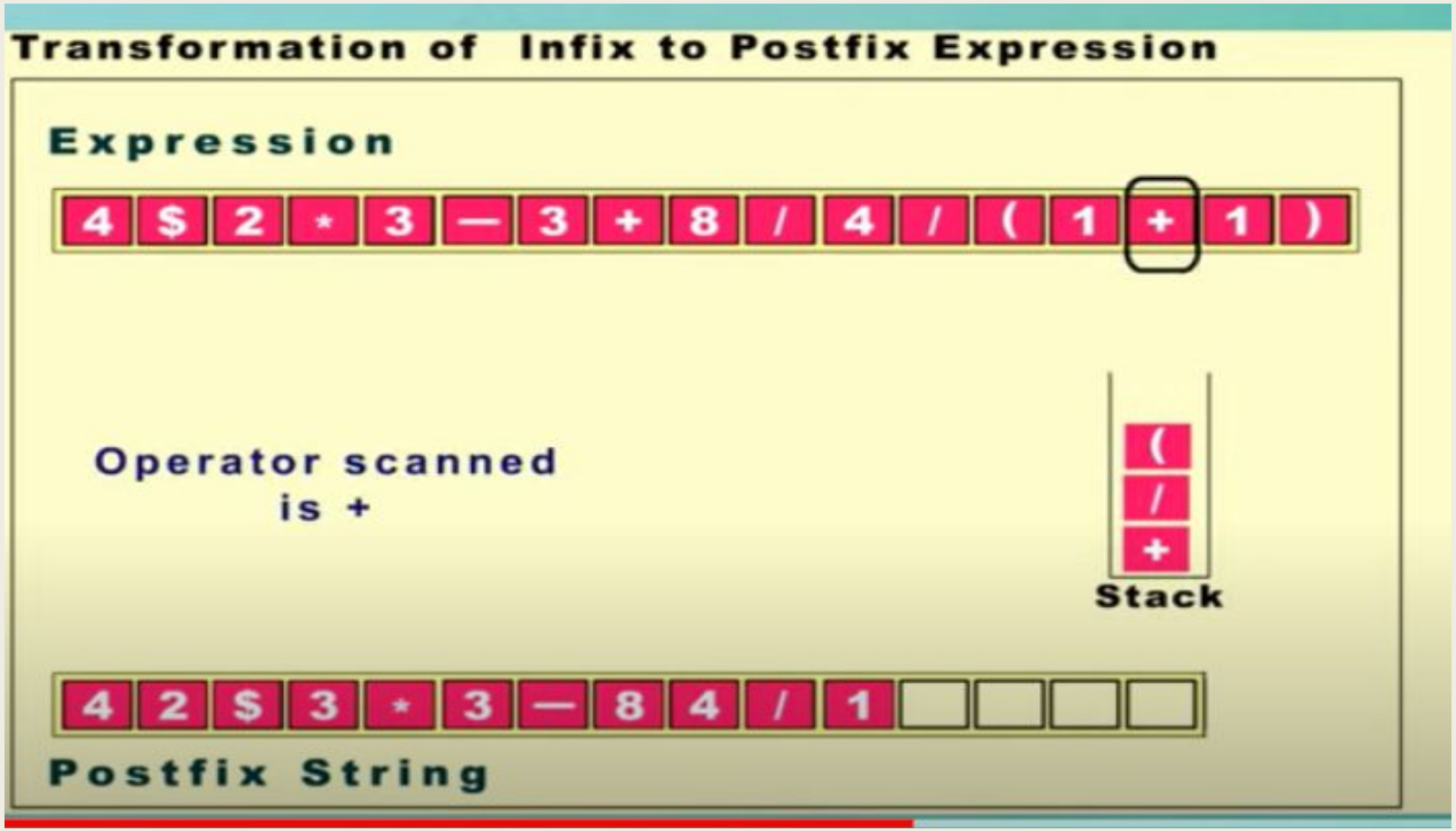

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Push /
on Stack
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

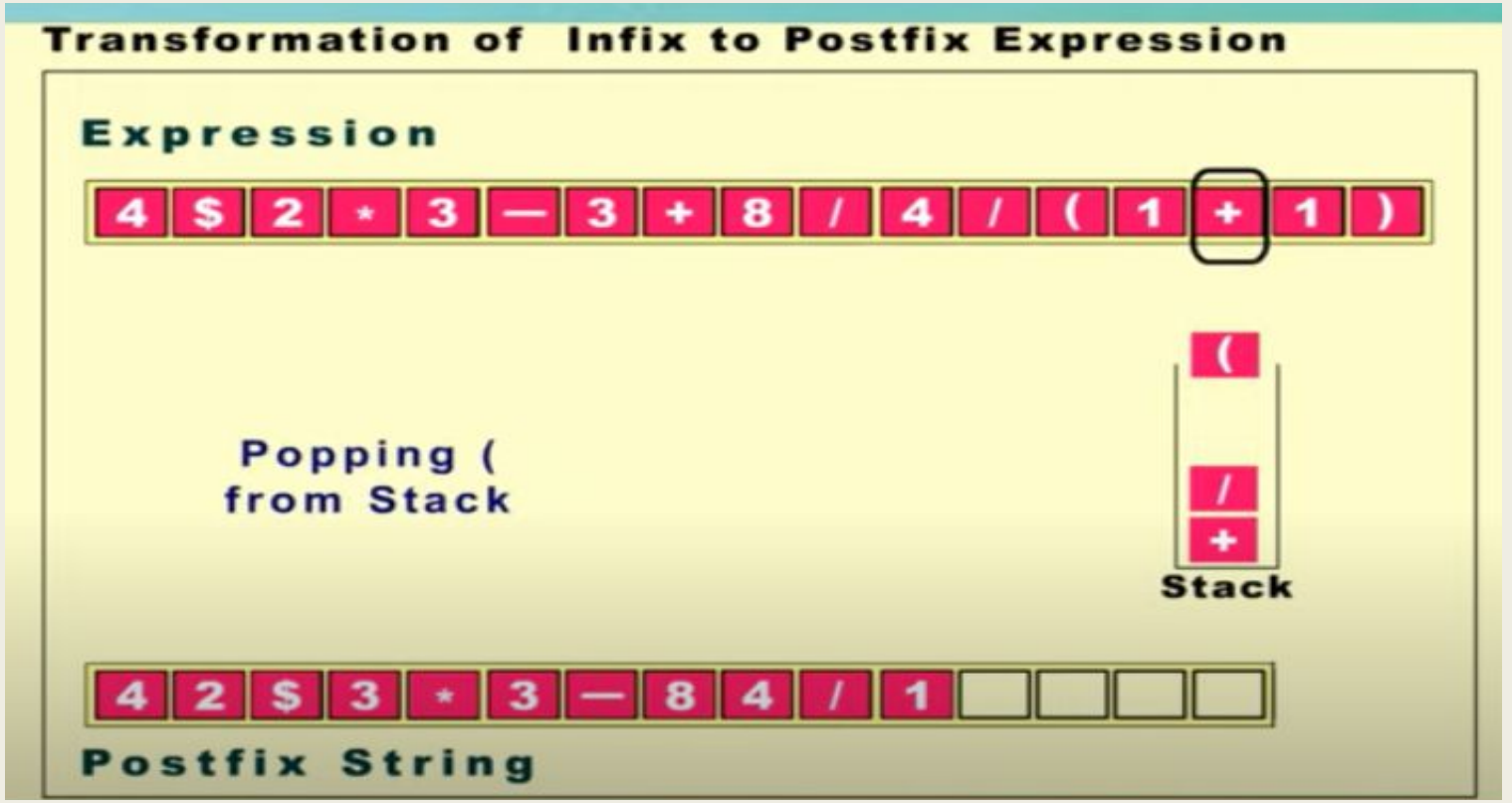

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / 1 + 1)
Operator scanned
is (
/
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

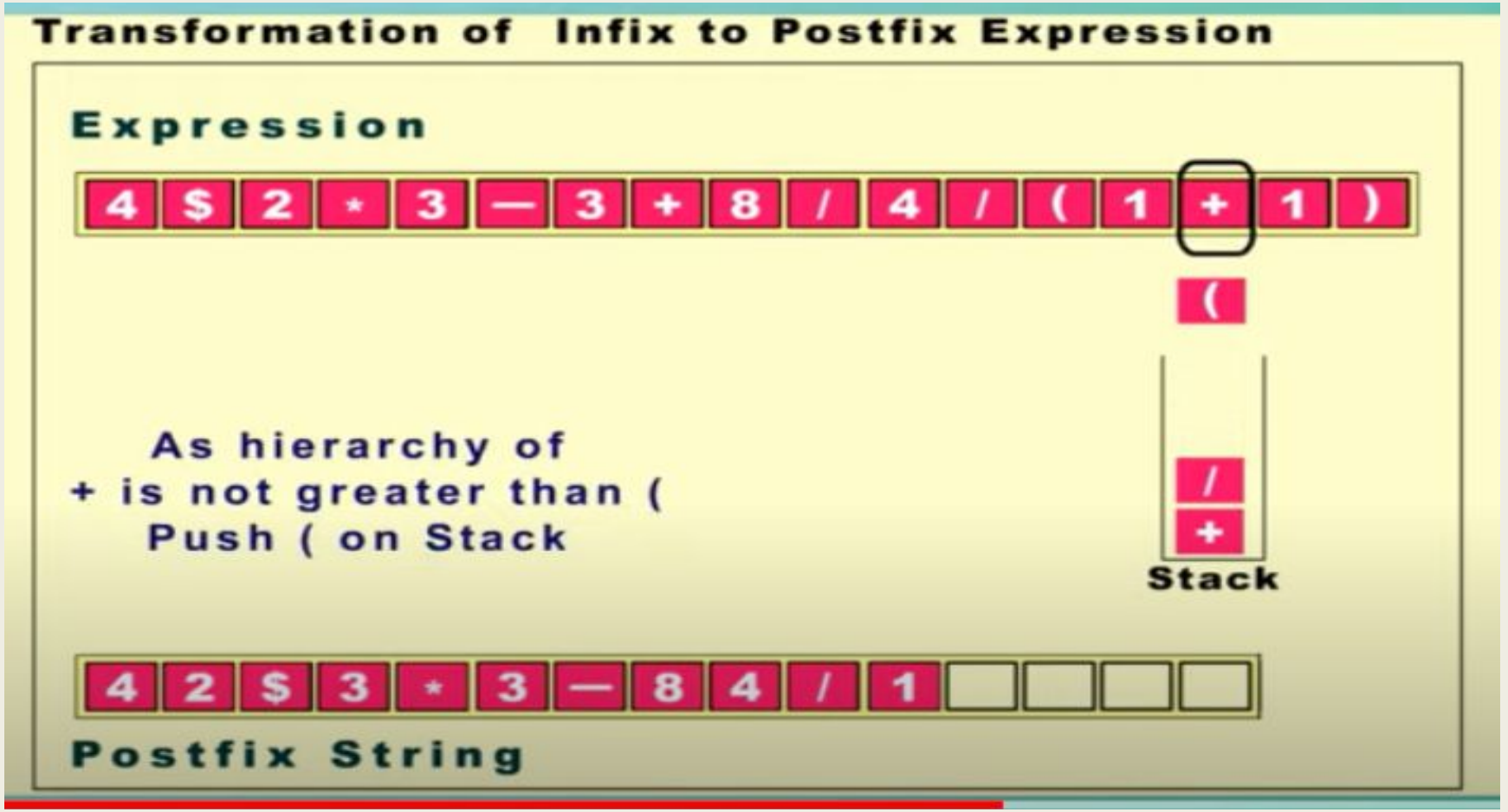

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
(
Push (
on Stack
/
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

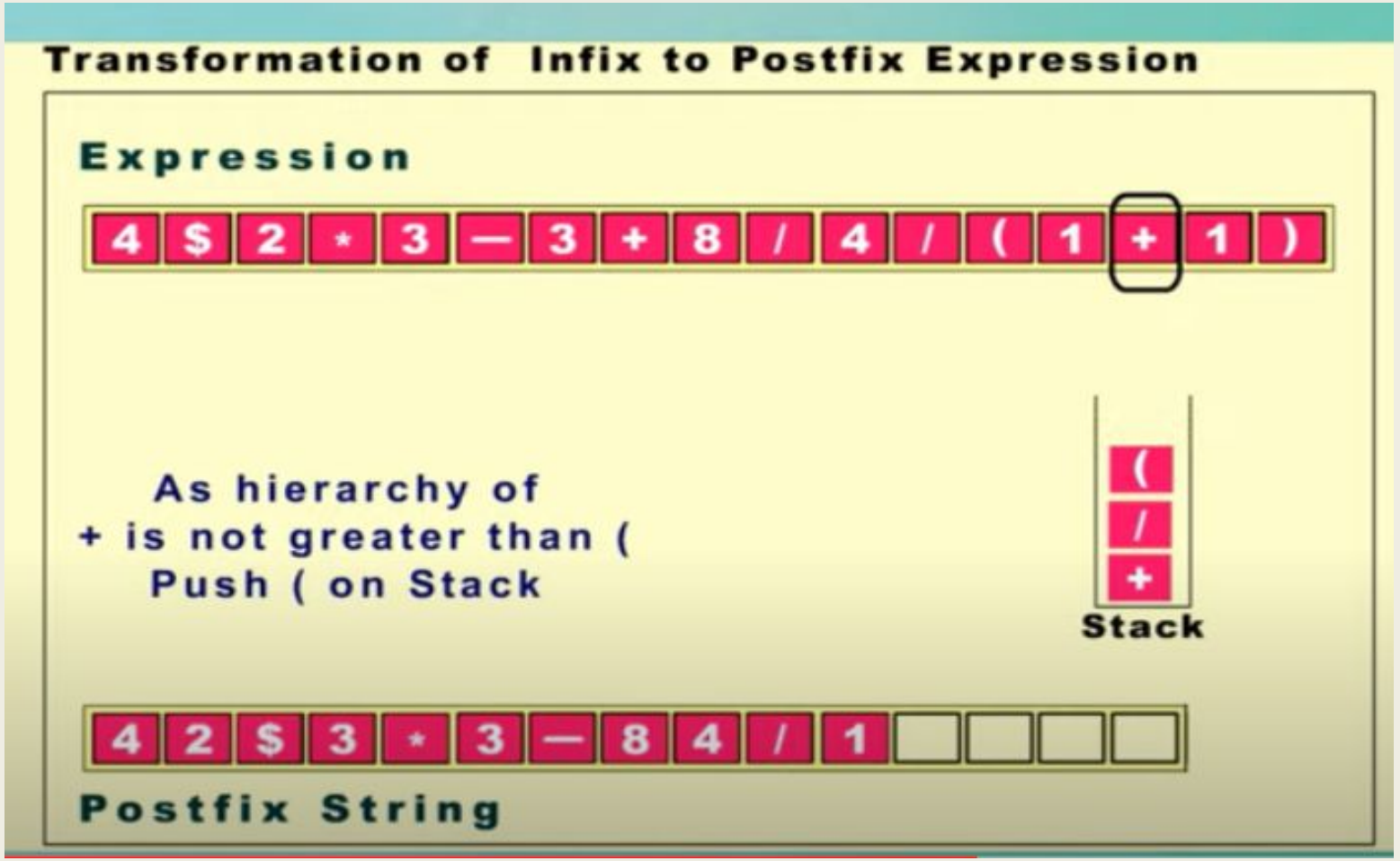

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (+ 1)
Digit scanned
is 1
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

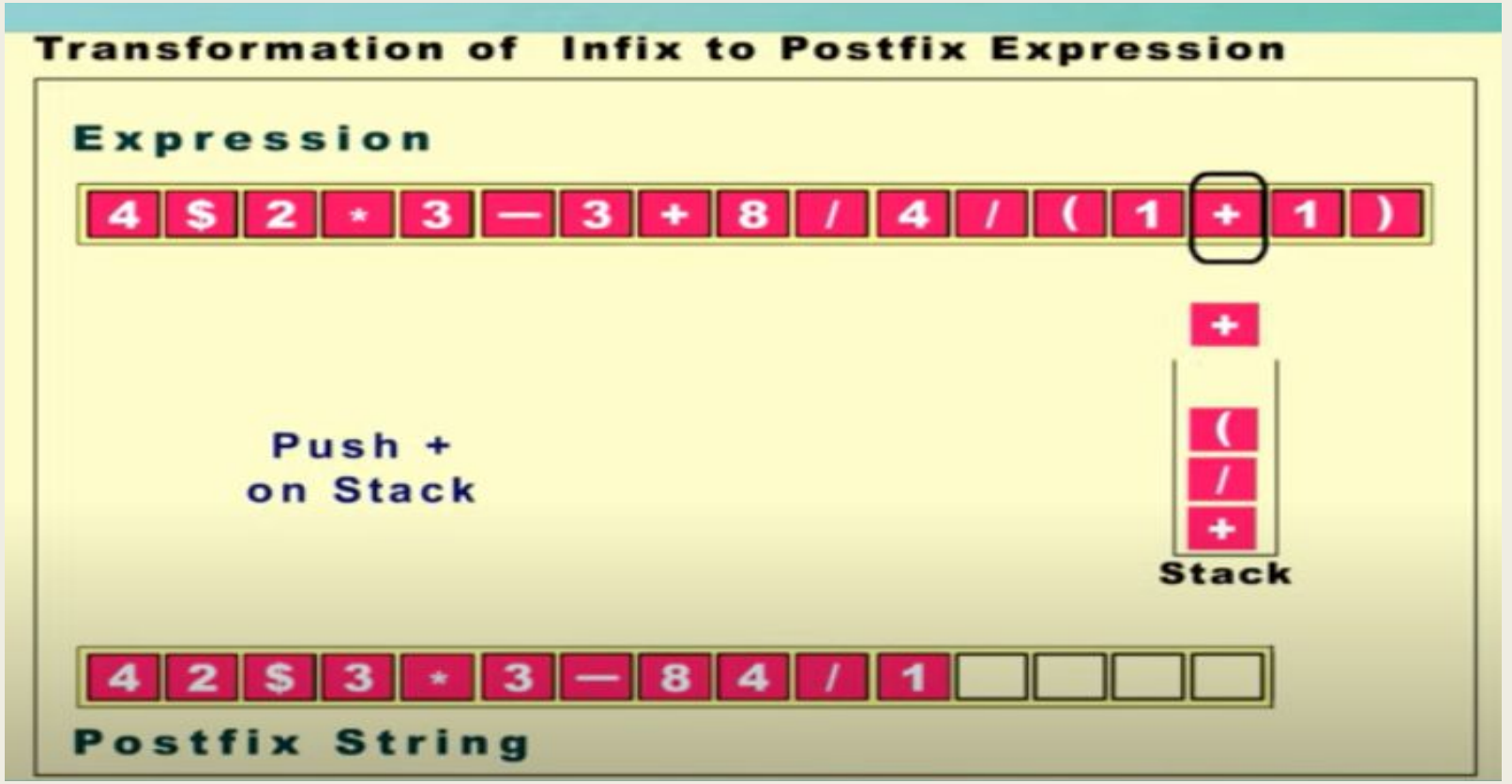

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add 1 to the
Postfix String
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 /
Postfix String

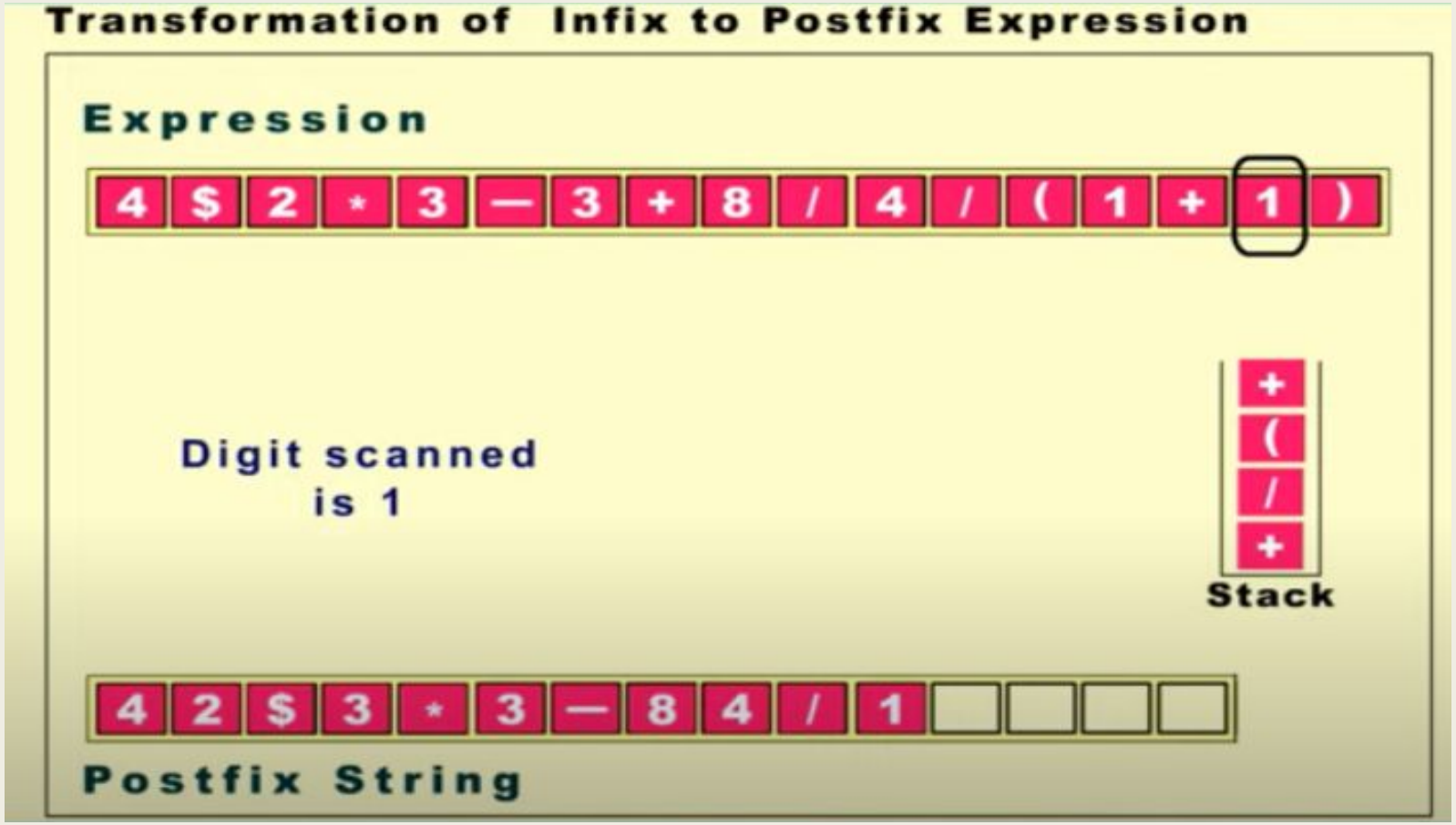

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Operator scanned
is +
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String

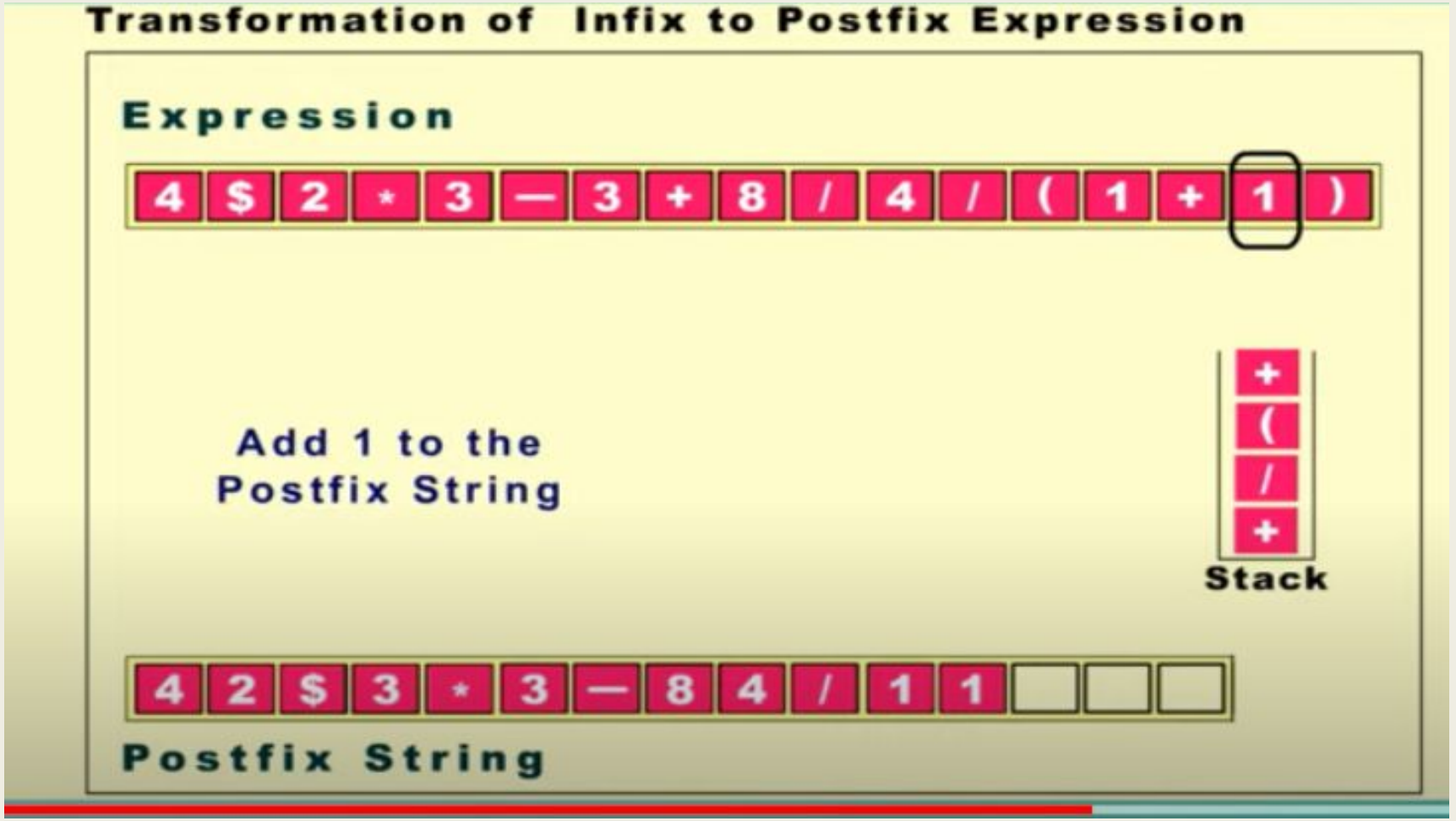

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Popping (
from Stack
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String

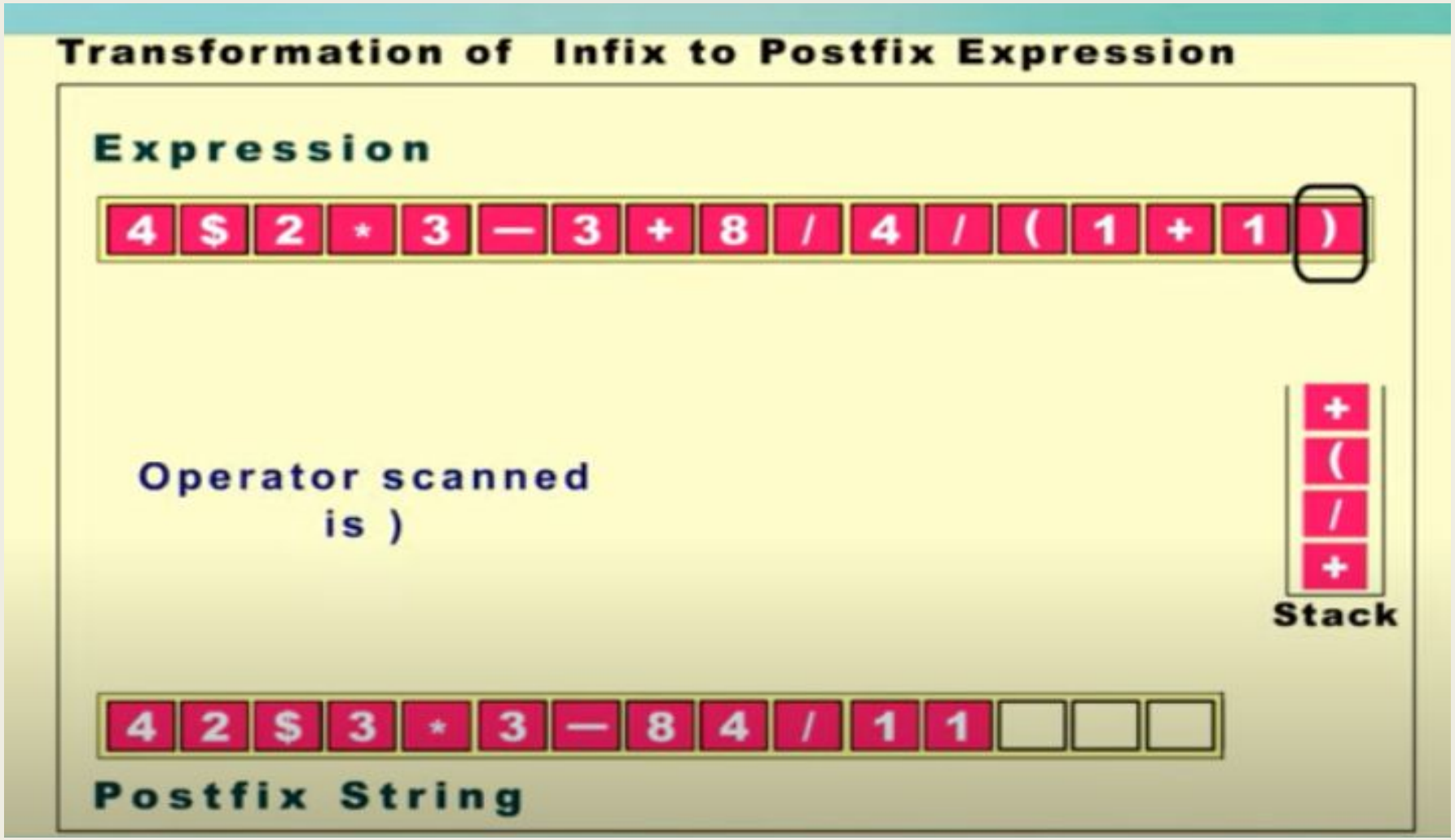

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
(
As hierarchy of
+ is not greater than (
Push (on Stack
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String

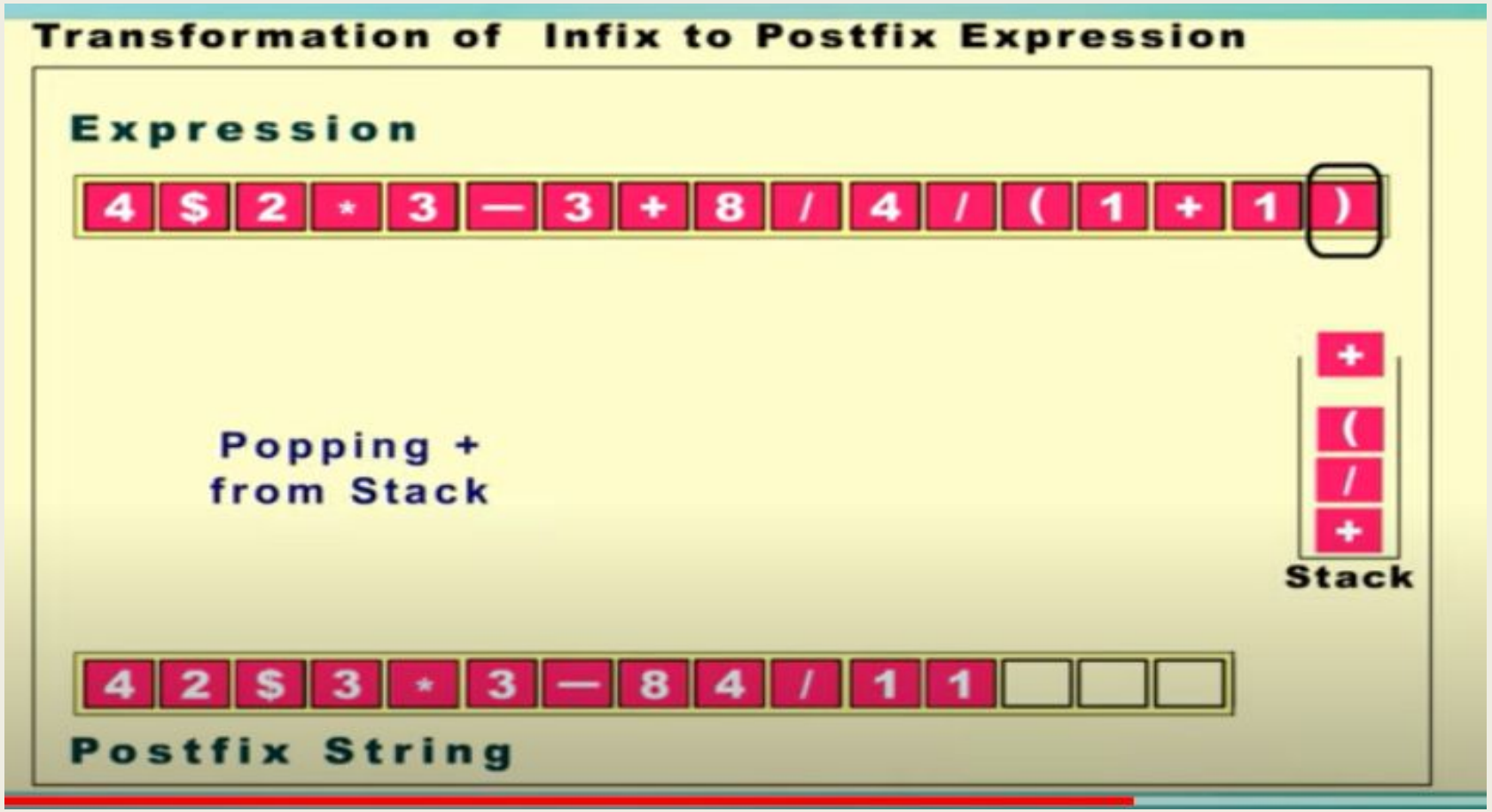

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
As hierarchy of
+ is not greater than (
Push (on Stack
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String

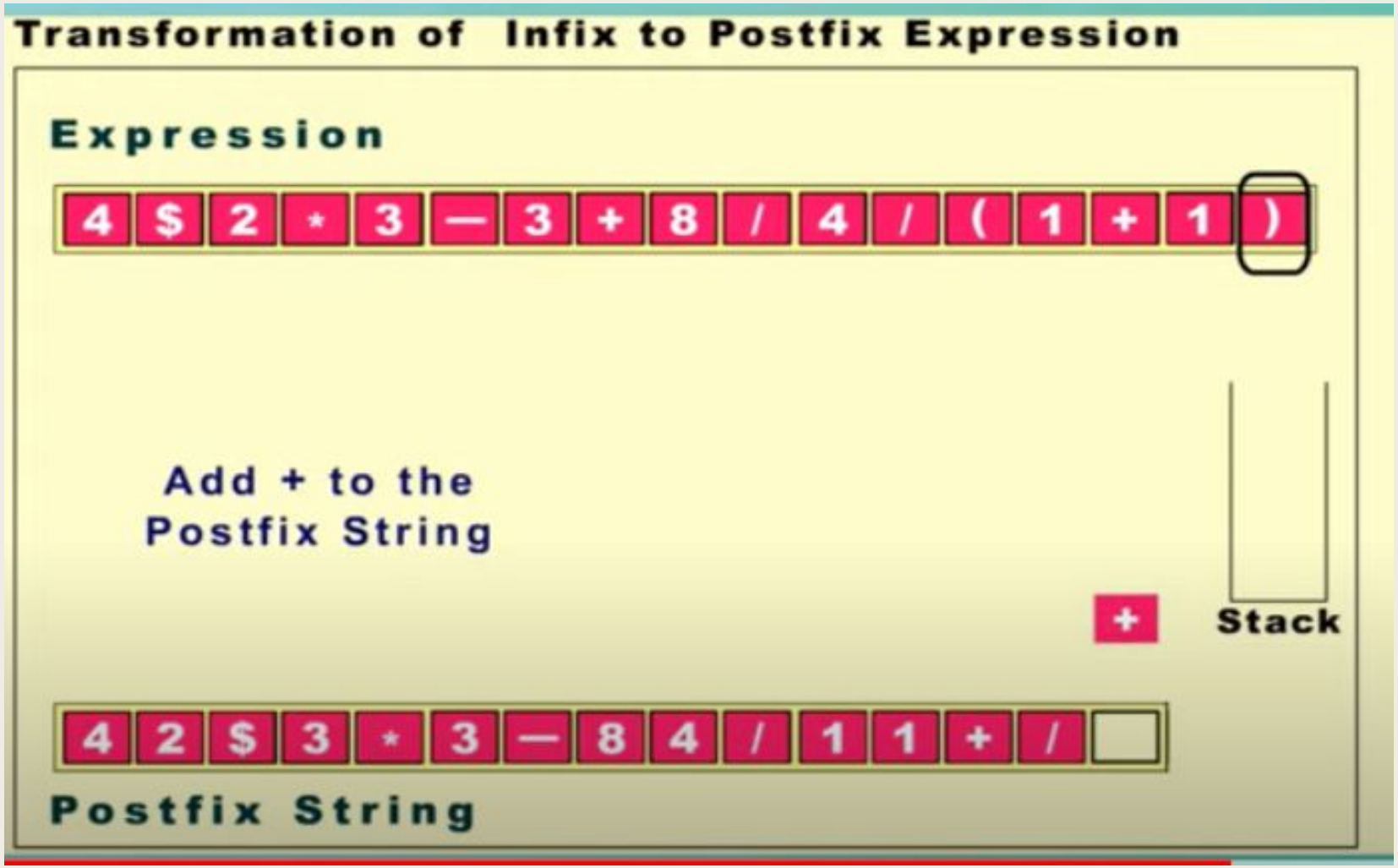

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
+
(
/
+
Stack
Push +
on Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String

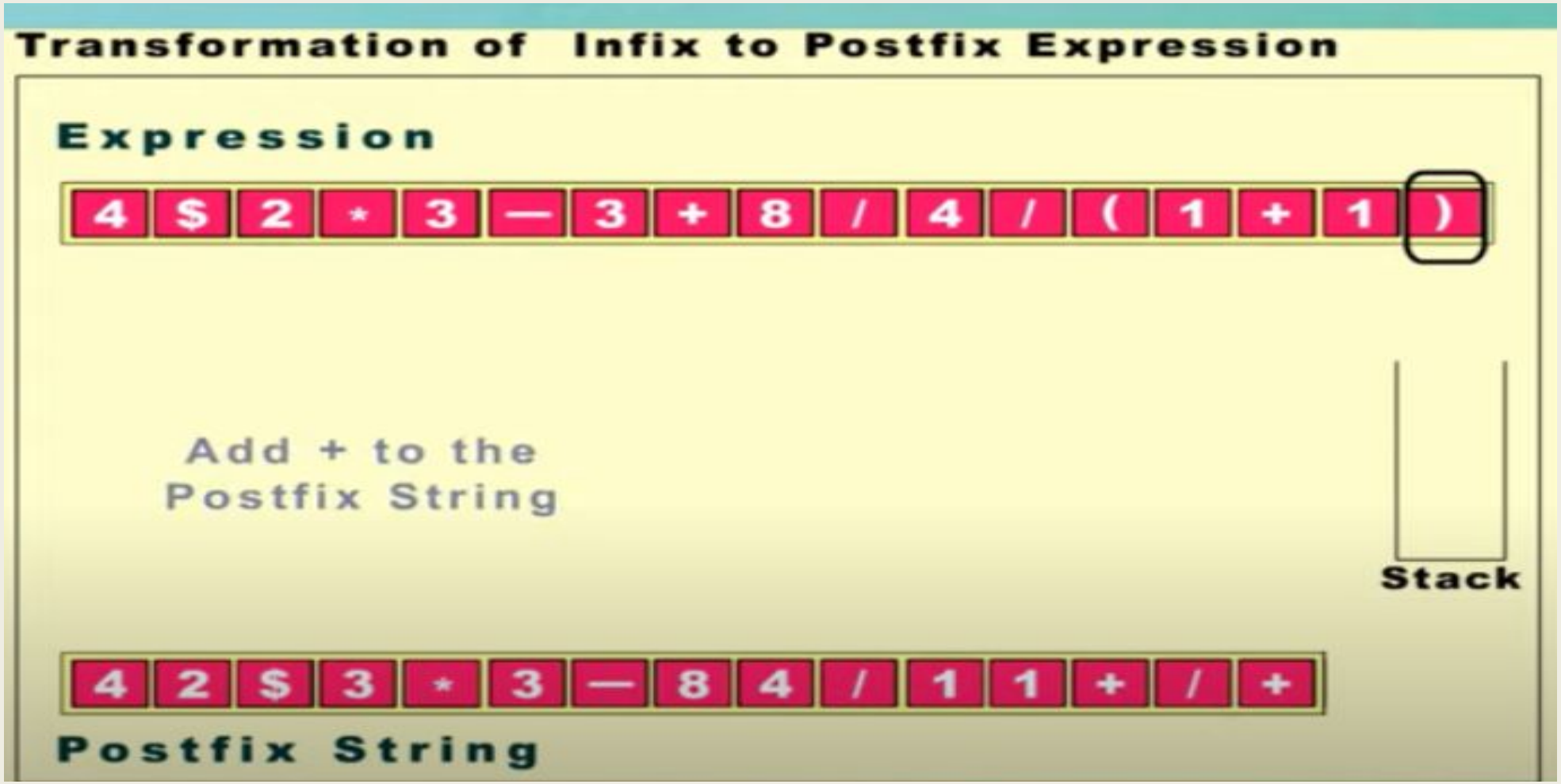

Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Digit scanned
is 1
+
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1
Postfix String


Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add 1 to the
Postfix String
+
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1 1
Postfix String


Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Operator scanned is)
+
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1 1
Postfix String


Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Popping +
from Stack
+
(
/
+
Stack
4 2 $ 3 * 3 — 8 4 / 1 1
Postfix String


Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add + to the
Postfix String
+
Stack
4 2 $ 3 * 3 — 8 4 / 1 1 + /
Postfix String


Transformation of Infix to Postfix Expression
Expression
4 $ 2 * 3 — 3 + 8 / 4 / (1 + 1)
Add + to the
Postfix String
Stack
4 2 $ 3 * 3 — 8 4 / 1 1 + / +
Postfix String

Infix to Prefix Conversion

- Steps to convert infix expression to prefix

1. First, reverse the given infix expression.
2. Scan the characters one by one.
3. If the character is an operand, copy it to the prefix notation output.
4. If the character is a closing parenthesis, then push it to the stack.
5. If the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis.
6. If the character scanned is an operator
   - If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack.
   - If the operator has precedence lesser than the top of the stack, pop the operator and output it to the prefix notation output and then check the above condition again with the new top of the stack.
7. After all the characters are scanned, reverse the prefix notation output.

Quick Sort-

- Quick Sort is a famous sorting algorithm.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm.

Quick Sort Algorithm-

- Consider-
- a = Linear Array in memory
- beg = Lower bound of the sub array in question
- end = Upper bound of the sub array in question

How Does Quick Sort Works?

- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

- The division performed is such that-
- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

- After dividing the array into two sections, the pivot is set at its correct position.
- Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

Quicksort Algorithm

PartitionArray(*a, beg, end, loc*)

Here *a* is a linear array in memory, and *beg* and *end* represents the lower bound and upper bound of the sub array in question. It partition the array into two halves by placing the partitioning element at index *loc*, which is its final position.

```
1.  Set left = beg, right = end, loc = beg
2.  Set done = false
3.  Repeat steps 4 to 10 While ( not done )
4.      Repeat step 5 While (a[loc]≤a[right]) and (loc≠right))
5.          Set right = right - 1
        Endwhile
6.      If ( loc = right ) then
            Set done = true
        Else if ( a[loc] > a[right] ) then
            Interchange a[loc] and a[right]
            Set loc = right
        Endif
7.      If ( not done ) then do step 8
8.          Repeat step 9 While (a[loc]≥a[left]) and (loc≠left))
9.              Set left = left + 1
            Endwhile
10.         If ( loc = left ) then
                Set done = true
            Else if ( a[loc] < a[left] ) then
                Interchange a[loc] and a[left]
                Set loc = left
            Endif
        Endif
    Endwhile
11. Return
```

QuickSortIterative(*a*, *n*)

Here *a* is a linear array of size *n* (>1) in memory. This algorithm sorts this array in ascending order using quick sort method, handling the left sub array first and then the right sub array. The variable *loc* is used to hold the index of splitting element. To keep track of the sub arrays it uses stack, of integers, explicitly.

```
1.  CreateStack(STK)
2.  Call Push(STK, 0)                  // push lower index of starting array
3.  Call Push(STK, n-1)                // push upper index of starting array
4.  Repeat steps 5 to 11 While ( STK is not empty )
5.     Call POP(STK, ub)               // pop upper index into up
6.     Call POP(STK, lb)               // pop upper index into lb
7.     Call PartitionArray( a, lb, ub, loc )
8.     Call Push(STK, loc+1)           // push lower index of right subarray
9.     Call Push(STK, ub)              // push upper index of right subarray
10.    Call Push(STK, lb)              // push lower index of left subarray
11.    Call Push(STK, loc-1)           // push upper index of left subarray
    Endwhile
12. Exit
```