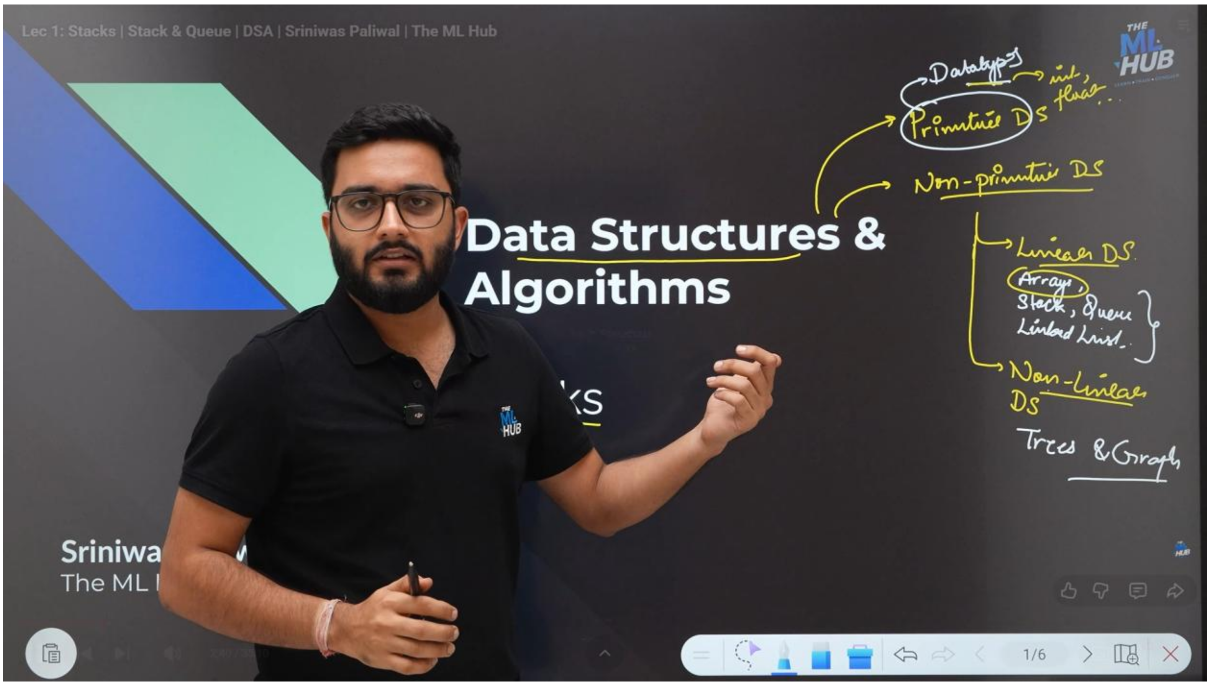
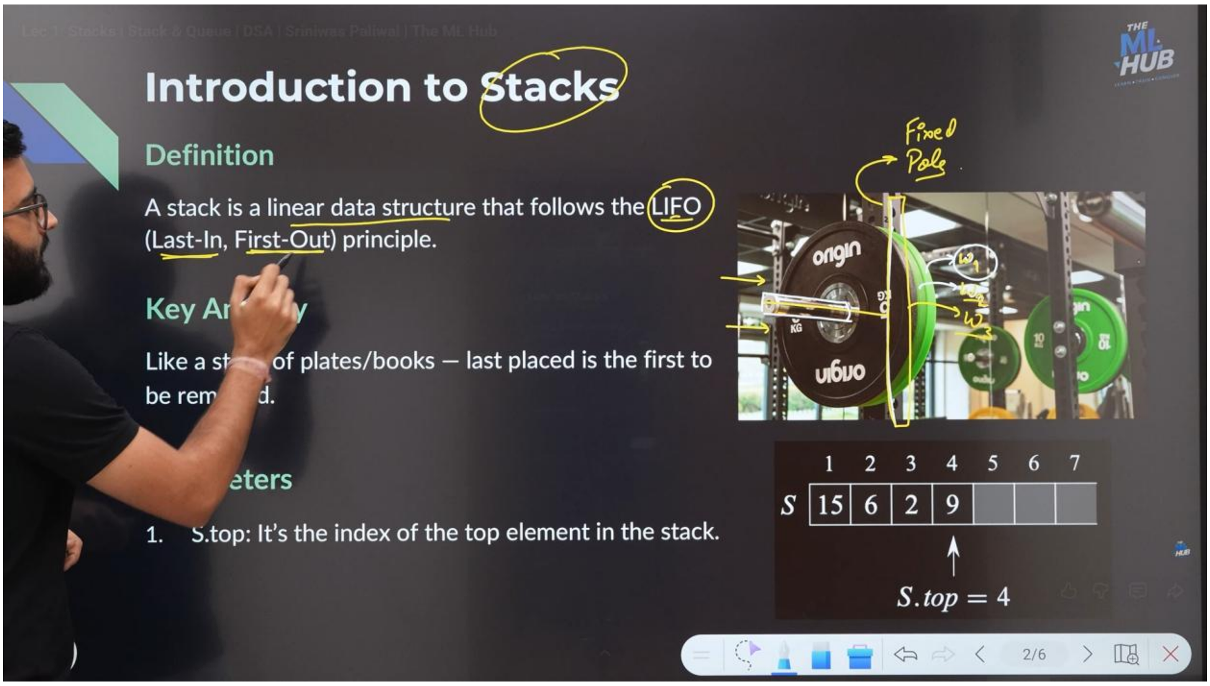
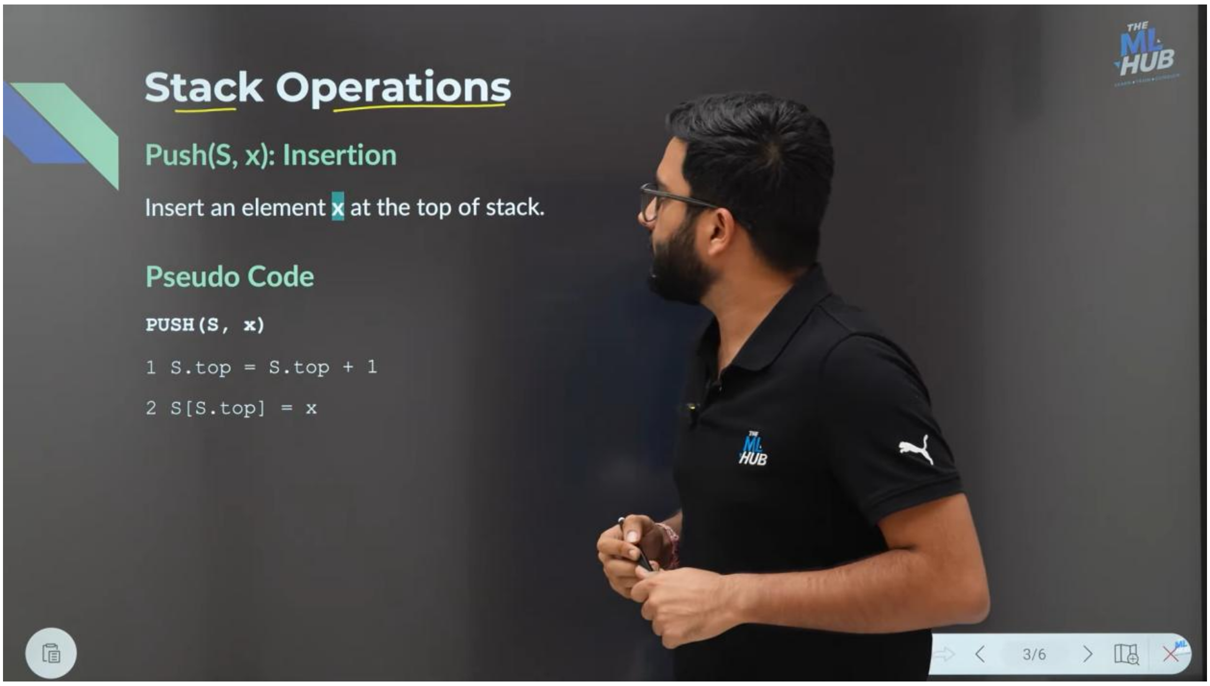
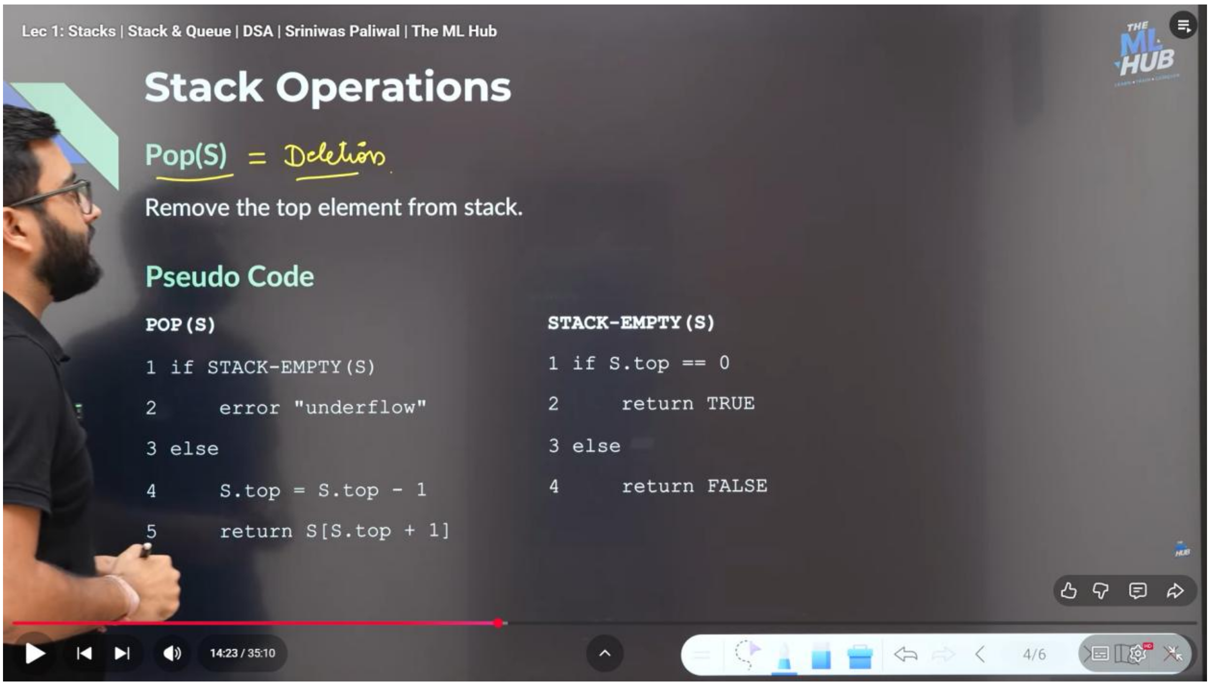

Lec 1: Stacks | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
THE ML HUB
Data Structures & Algorithms
Sriniwa
The ML
Datatypes
Primitive DS
int, float...
Non-primitive DS
Linear DS
Arrays
Stack, Queue
Linked List
Non-Linear DS
Trees & Graph
1/6

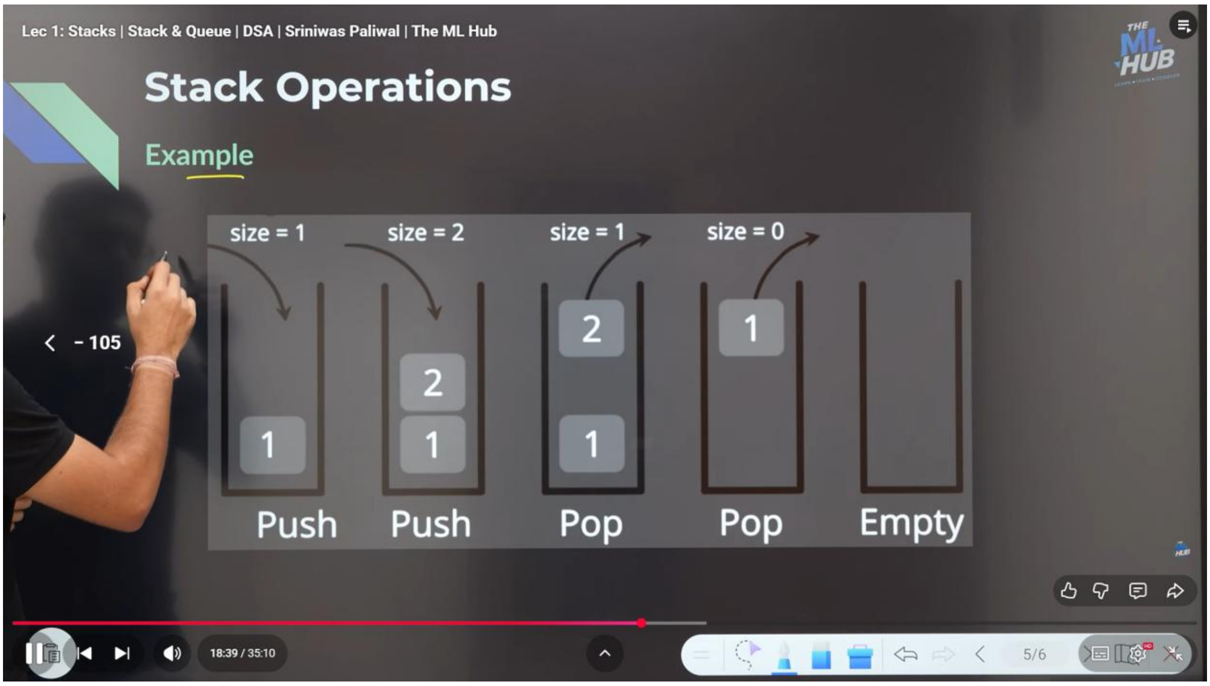

Lec 1: Stacks | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
THE ML HUB
Introduction to Stacks
Definition
A stack is a linear data structure that follows the LIFO (Last-In, First-Out) principle.
Key Ar
Like a st of plates/books — last placed is the first to be rem d.
eters
1. S.top: It's the index of the top element in the stack.
Fixed Pole
origin
1 2 3 4 5 6 7
S 15 6 2 9
S.top = 4
2/6

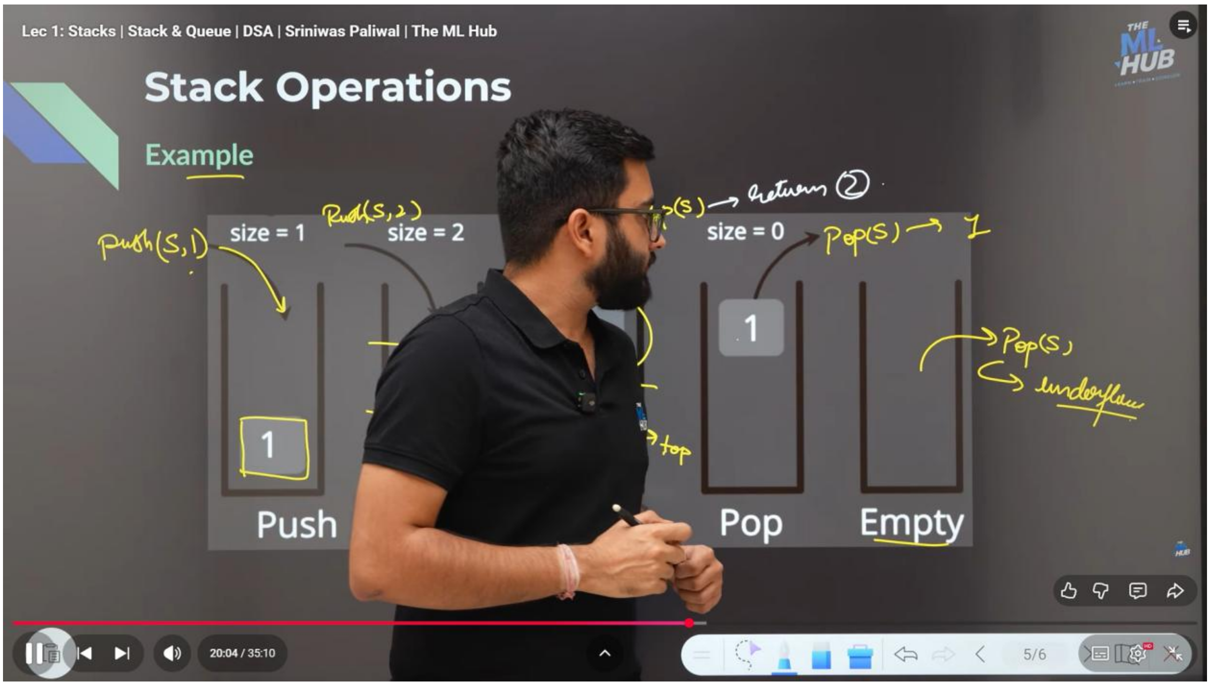


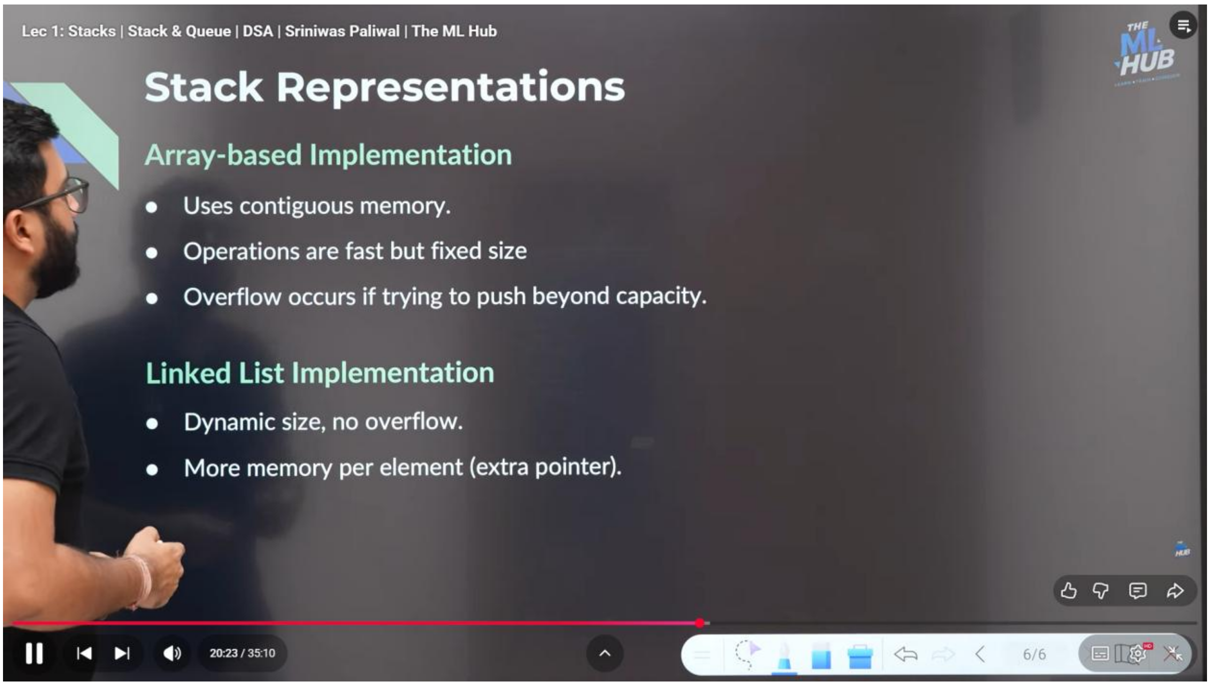
0(1)

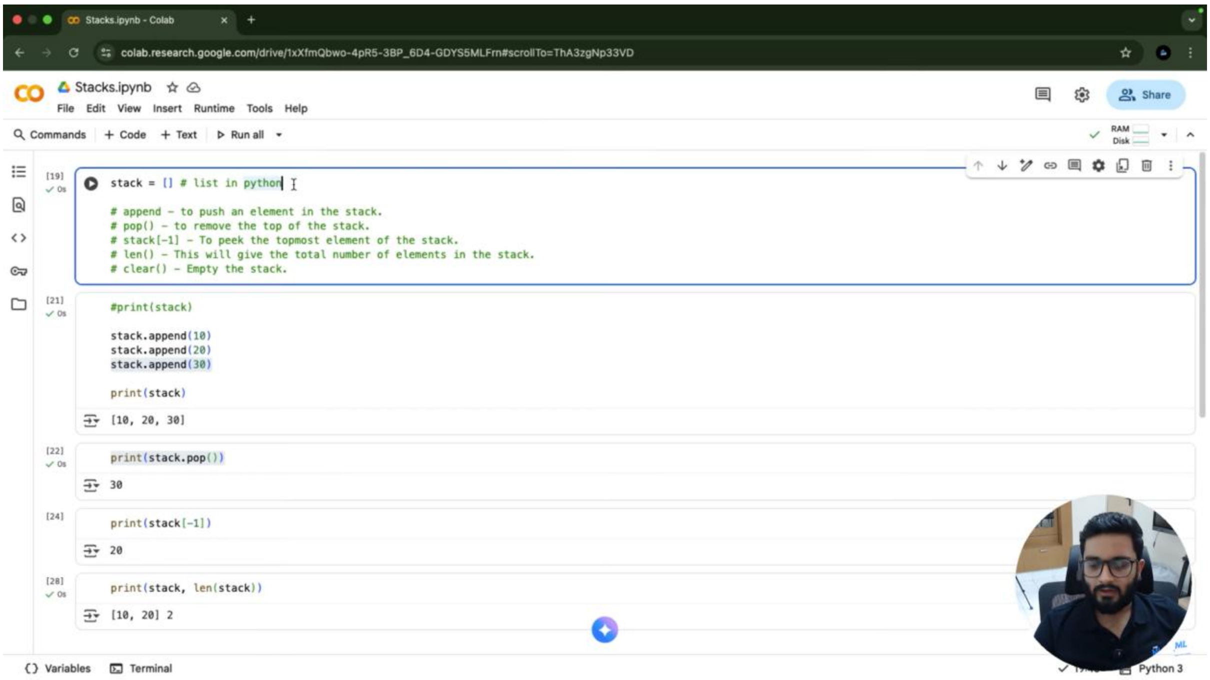


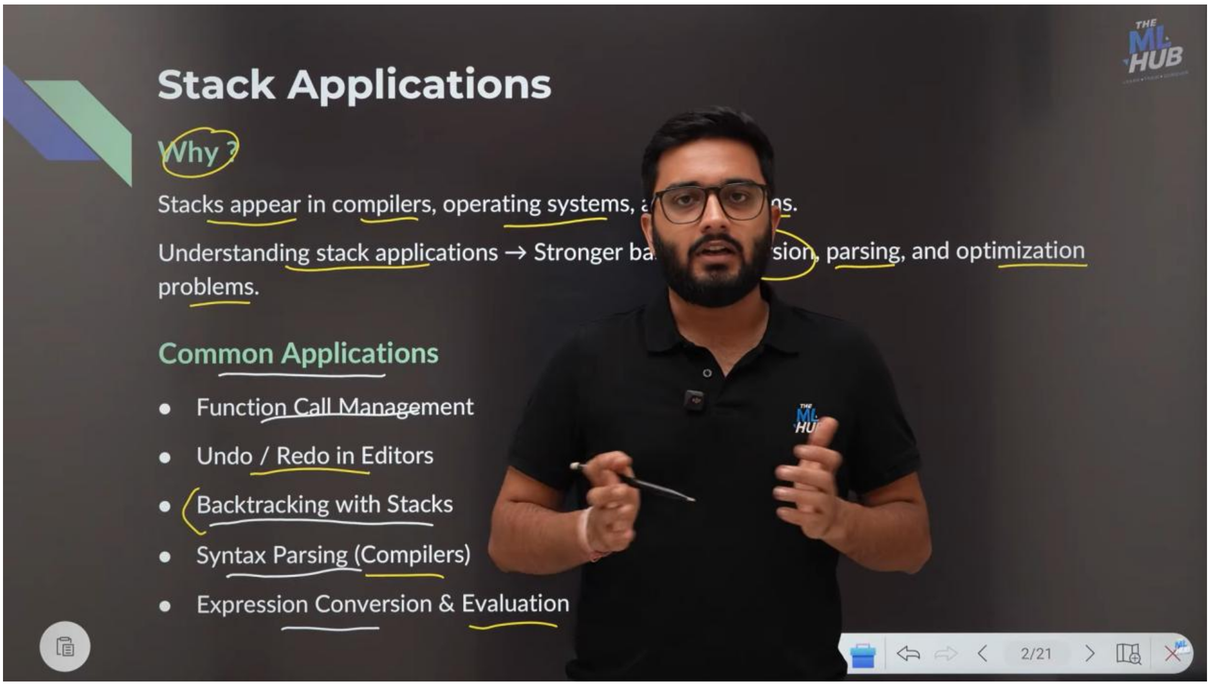
0(1)-> time complexity


Lec 1: Stacks | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Stack Operations
Example
size = 1
size = 2
size = 1
size = 0
1
2
1
2
1
1
Push
Push
Pop
Pop
Empty
- 105
18:39 / 35:10
5/6

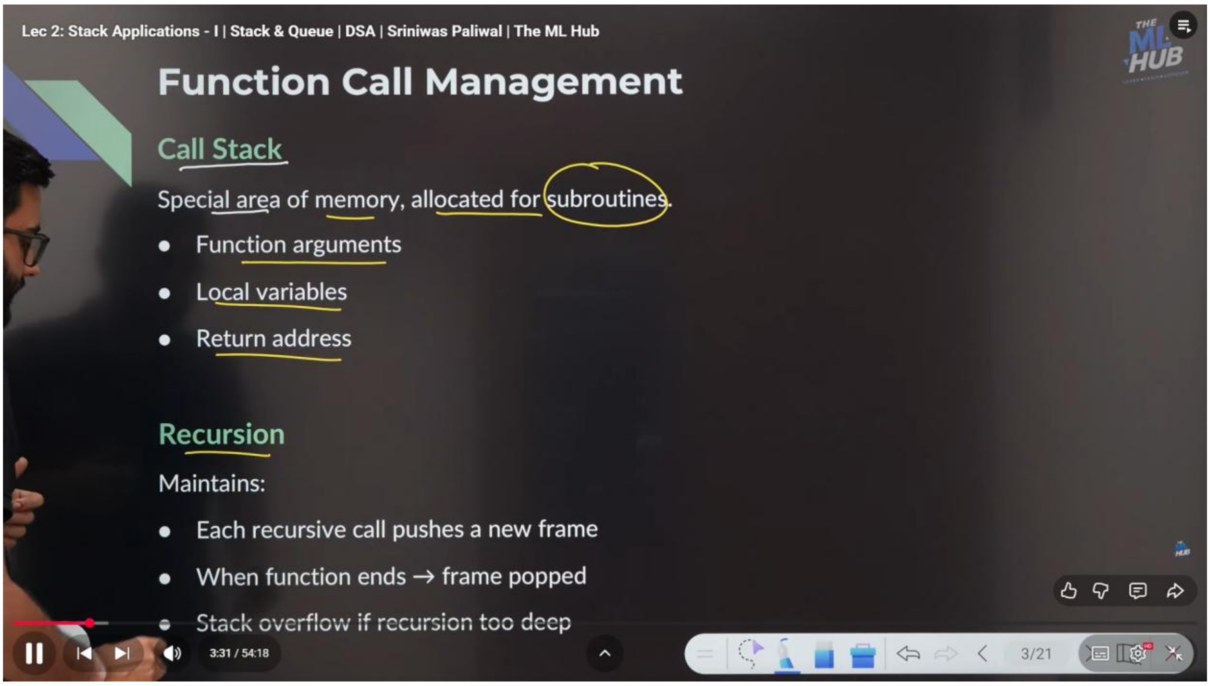

Lec 1: Stacks | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Stack Operations
Example
Push(S,1)
Push(S,2)
size = 1
size = 2
size = 0
return 2
Pop(S) → 1
1
1
top
Pop(S)
underflow
Push
Pop
Empty
20:04 / 35:10
5/6


Lec 1: Stacks | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Stack Representations
Array-based Implementation
Uses contiguous memory.
Operations are fast but fixed size
Overflow occurs if trying to push beyond capacity.
Linked List Implementation
Dynamic size, no overflow.
More memory per element (extra pointer).
20:23 / 35:10
6/6

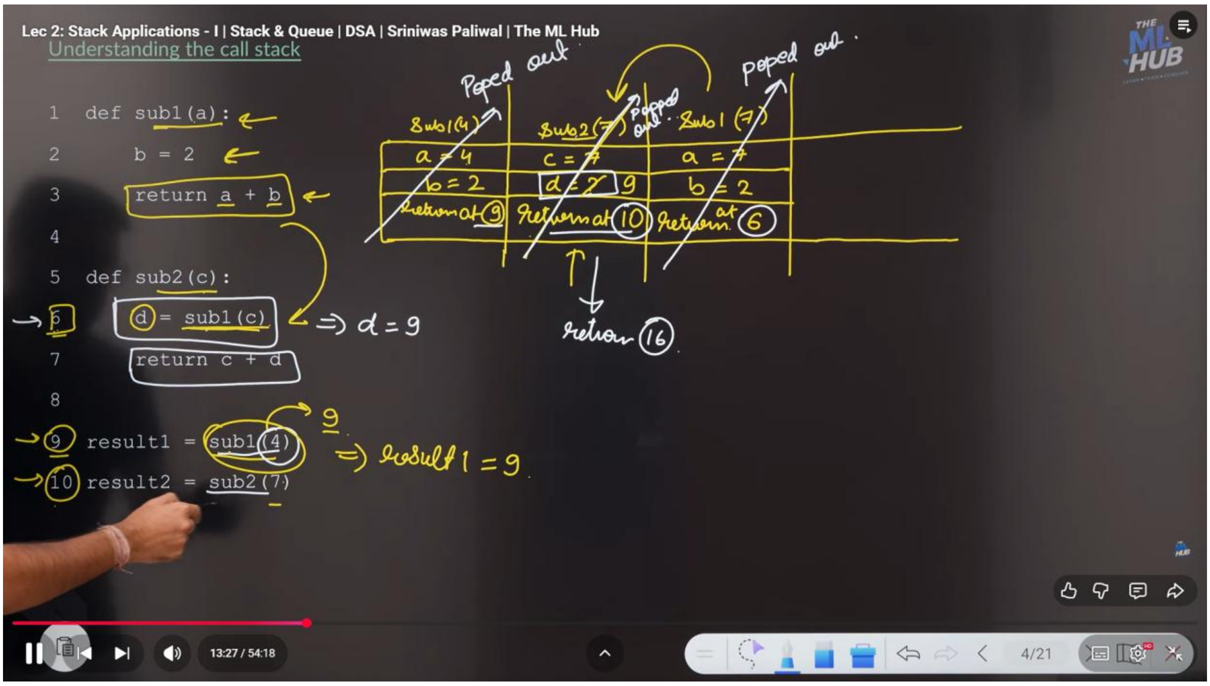

Stacks.ipynb - Colab
colab.research.google.com/drive/1xXfmQbwo-4pR5-3BP_6D4-GDYS5MLFrn#scrollTo=ThA3zgNp33VD
Stacks.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
stack = [] # list in python
# append - to push an element in the stack.
# pop() - to remove the top of the stack.
# stack[-1] - To peek the topmost element of the stack.
# len() - This will give the total number of elements in the stack.
# clear() - Empty the stack.
#print(stack)
stack.append(10)
stack.append(20)
stack.append(30)
print(stack)
[10, 20, 30]
print(stack.pop())
30
print(stack[-1])
20
print(stack, len(stack))
[10, 20] 2
Variables Terminal
Python 3

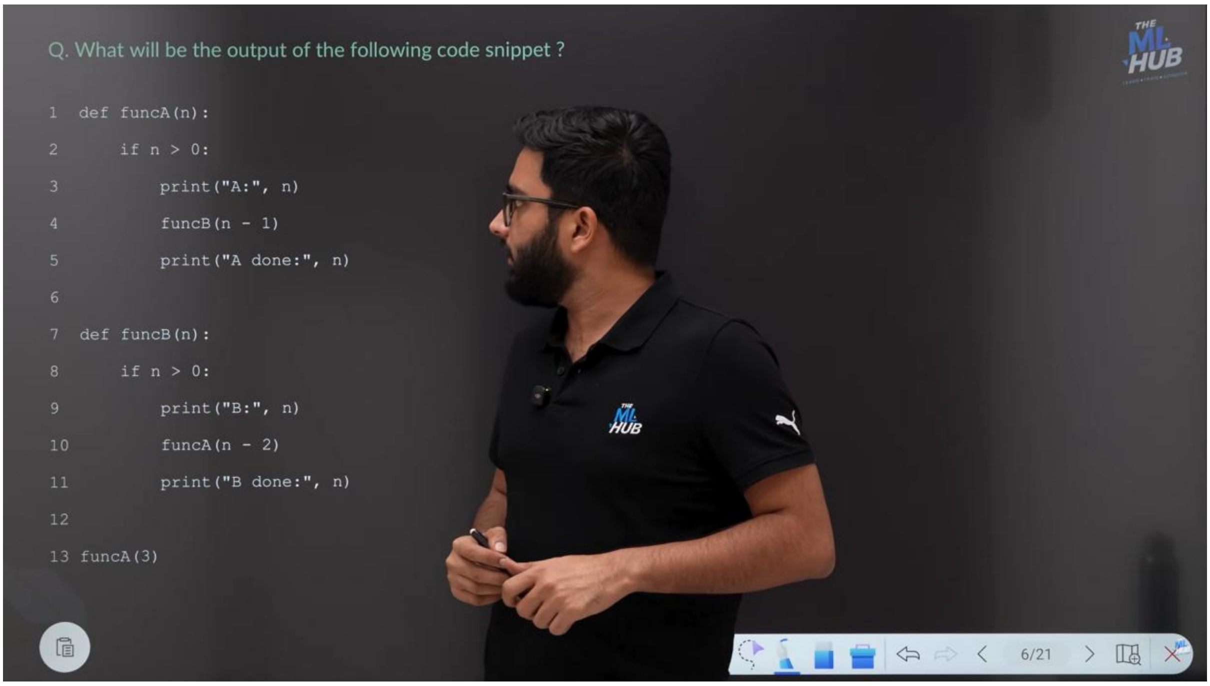


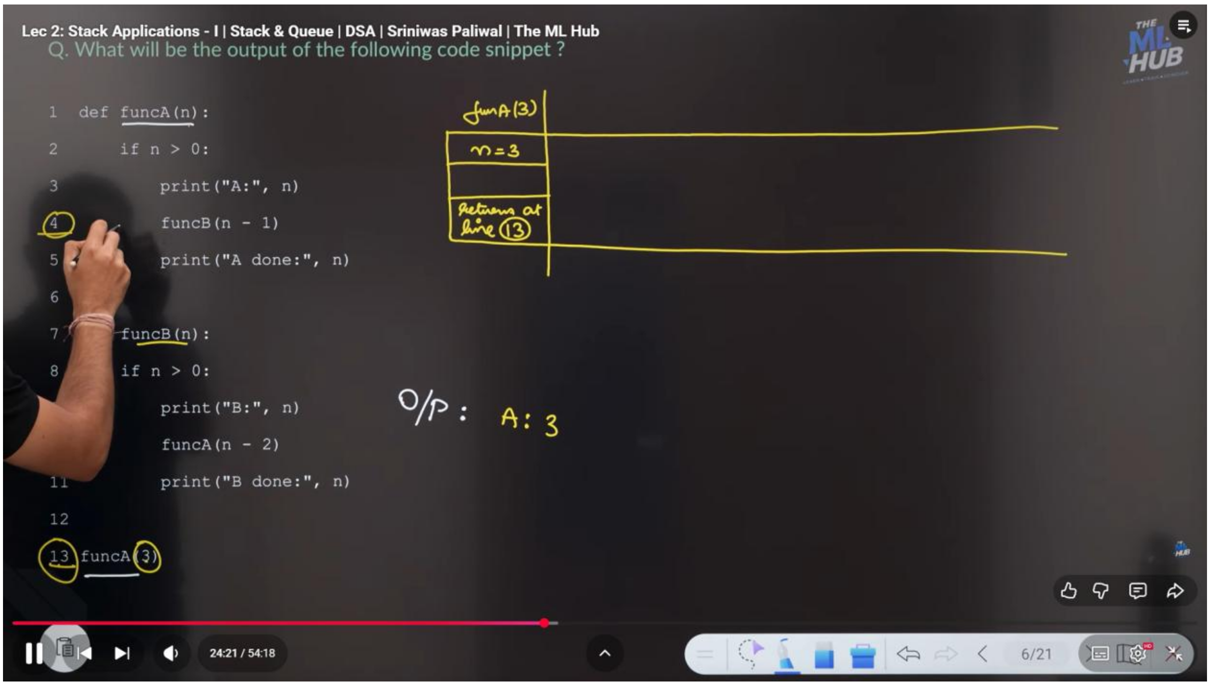
THE ML HUB
Stack Applications
Why ?
Stacks appear in compilers, operating systems,
ns.
Understanding stack applications → Stronger ba
sion, parsing, and optimization
problems.
Common Applications
Function Call Management
Undo / Redo in Editors
Backtracking with Stacks
Syntax Parsing (Compilers)
Expression Conversion & Evaluation
2/21

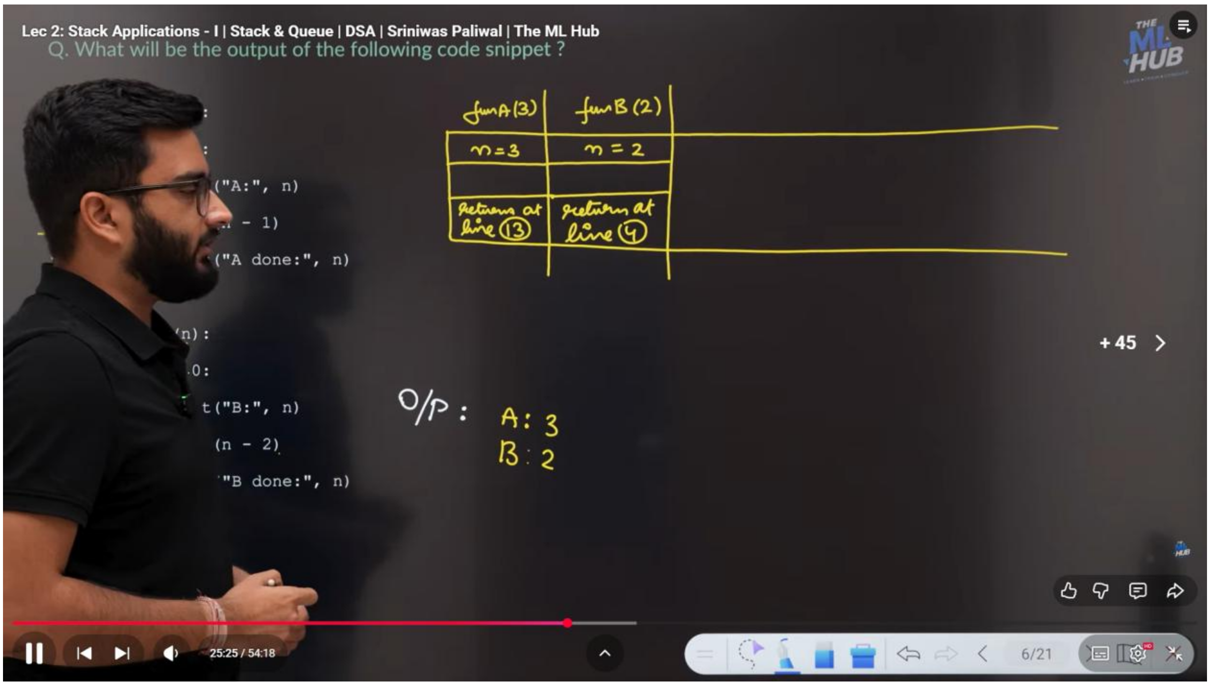


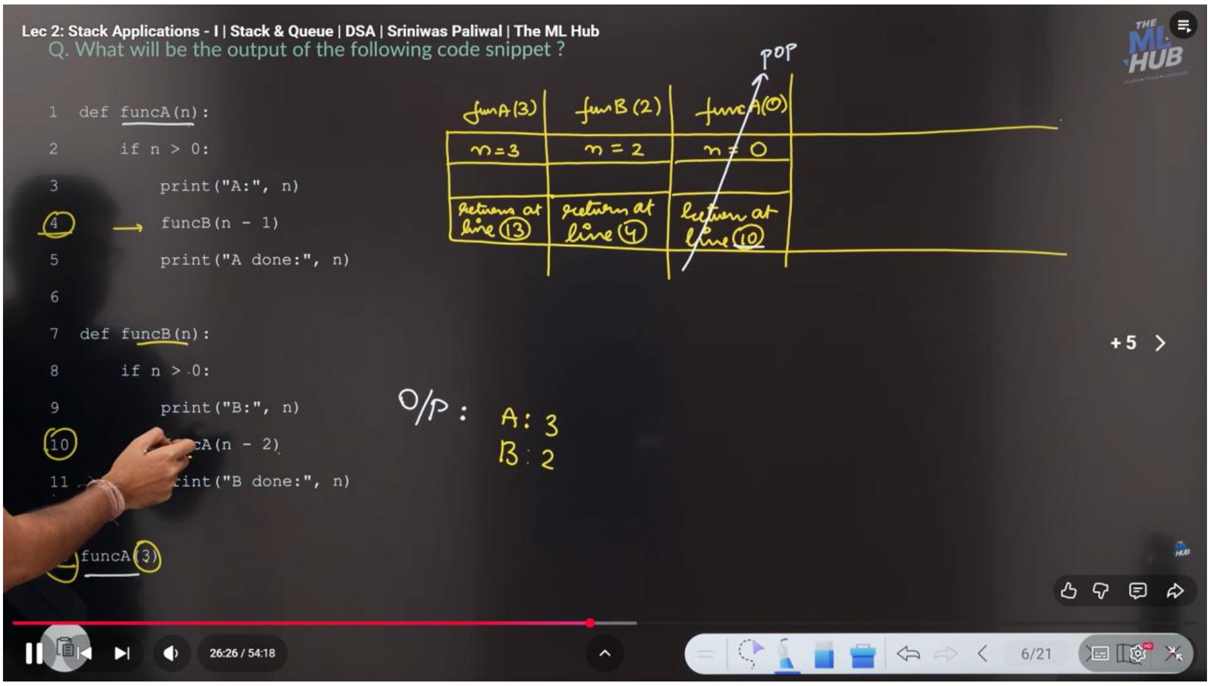
Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
THE ML HUB
Function Call Management
Call Stack
Special area of memory, allocated for subroutines.
Function arguments
Local variables
Return address
Recursion
Maintains:
Each recursive call pushes a new frame
When function ends → frame popped
Stack overflow if recursion too deep
3:31 / 54:18
3/21

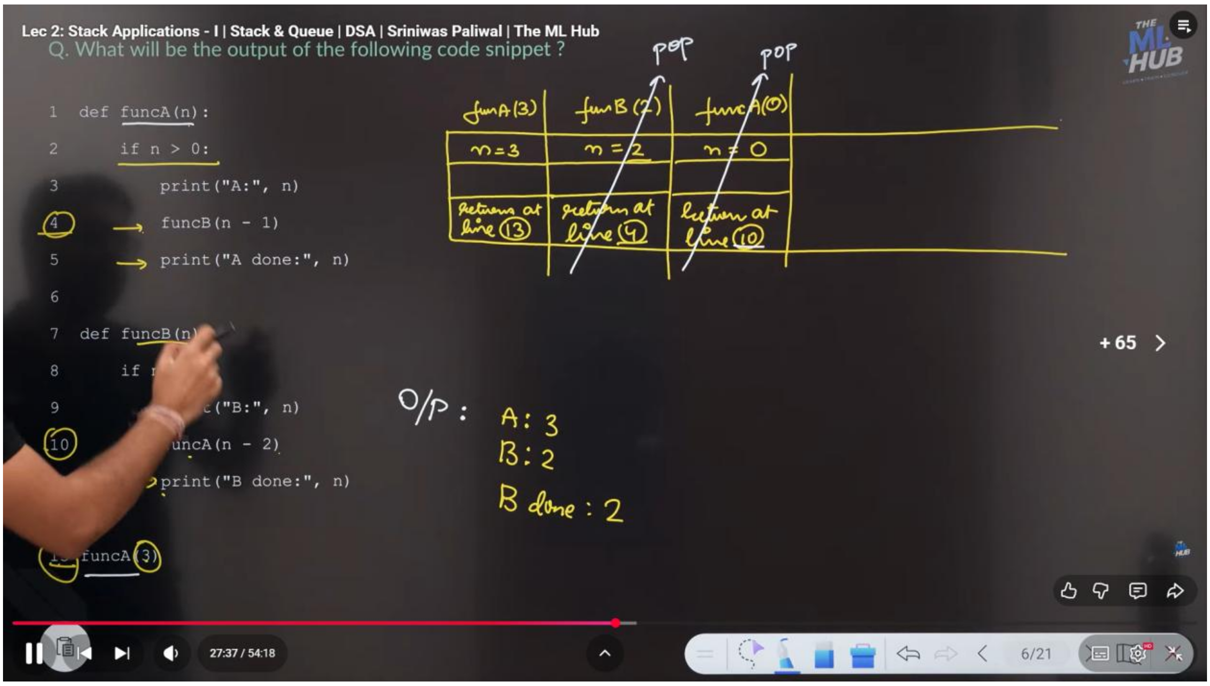

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Understanding the call stack
def sub1(a):
b = 2
return a + b
def sub2(c):
d = sub1(c)
return c + d
result1 = sub1(4)
result2 = sub2(7)
⇒ d = 9
9
⇒ result1 = 9
Poped out
Poped out
Popped out
sub1(4)
a = 4
b = 2
return at 9
sub2(7)
c = 7
d = 9
return at 10
sub1(7)
a = 7
b = 2
return at 6
return 16
13:27 / 54:18
4/21

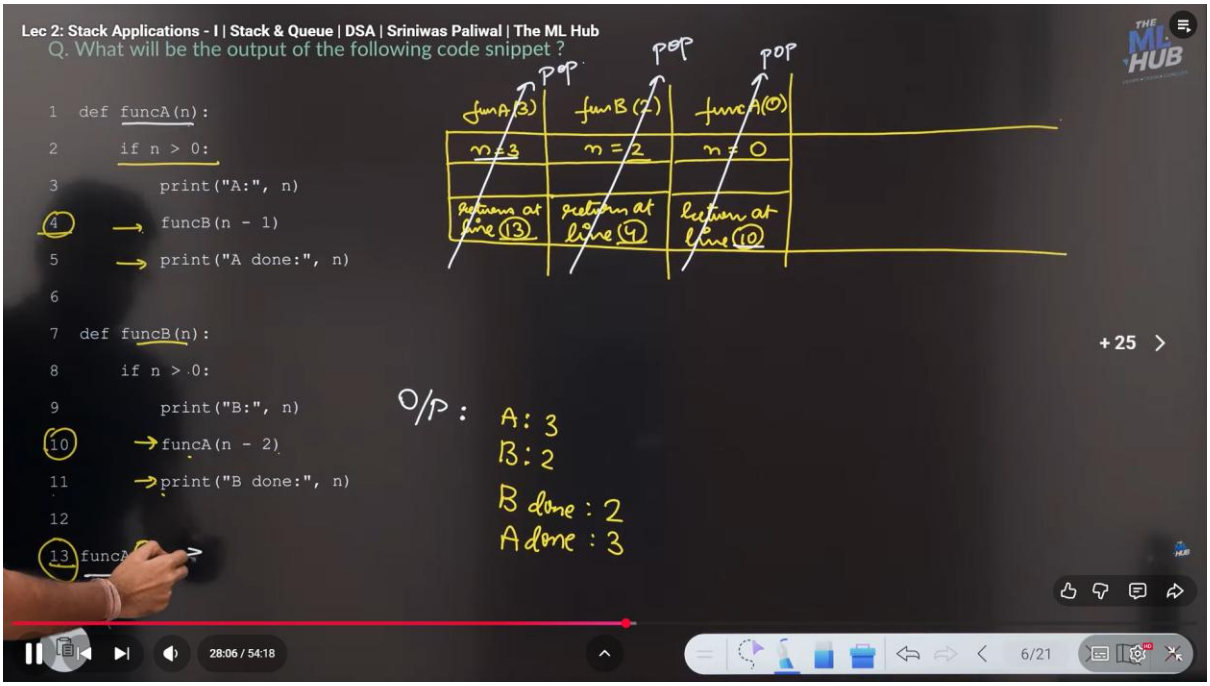

Q. What will be the output of the following code snippet ?
def funcA(n):
if n > 0:
print("A:", n)
funcB(n - 1)
print("A done:", n)
def funcB(n):
if n > 0:
print("B:", n)
funcA(n - 2)
print("B done:", n)
funcA(3)
6/21

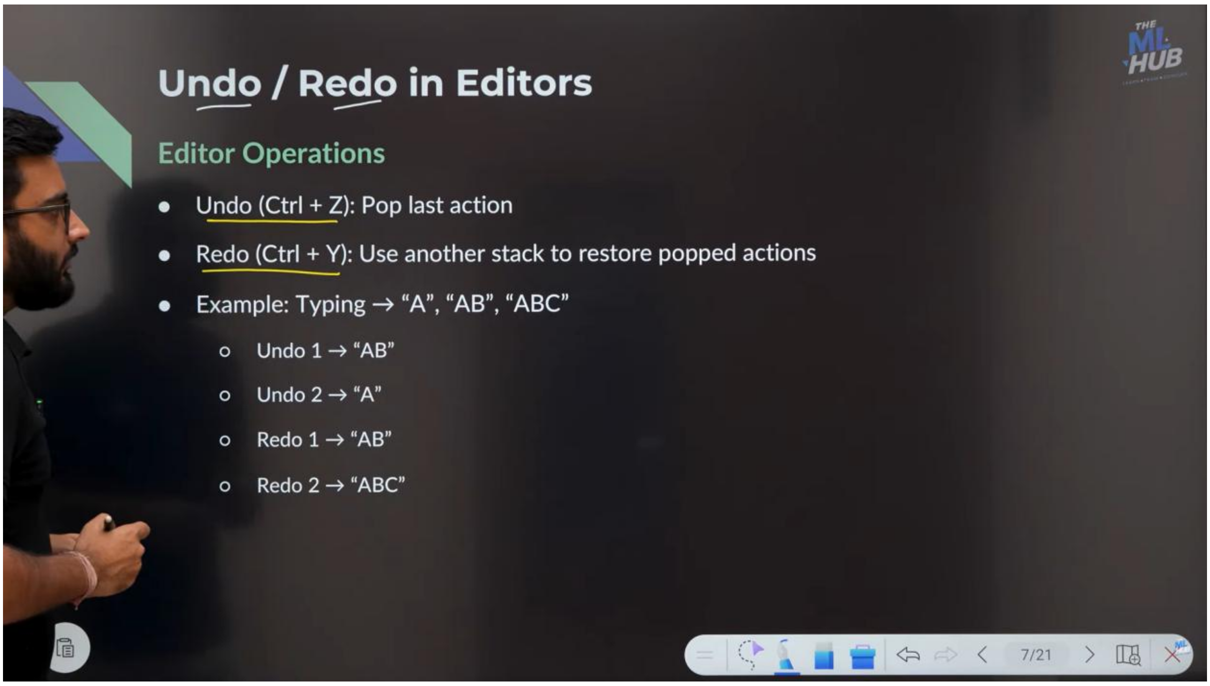

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Q. What will be the output of the following code snippet ?
1 def funcA(n):
2 if n > 0:
3 print("A:", n)
4 funcB(n - 1)
5 print("A done:", n)
6
7 funcB(n):
8 if n > 0:
print("B:", n)
funcA(n - 2)
11 print("B done:", n)
12
13 funcA(3)
funA(3)
n=3
Return at line 13
O/P : A: 3
24:21 / 54:18
6/21

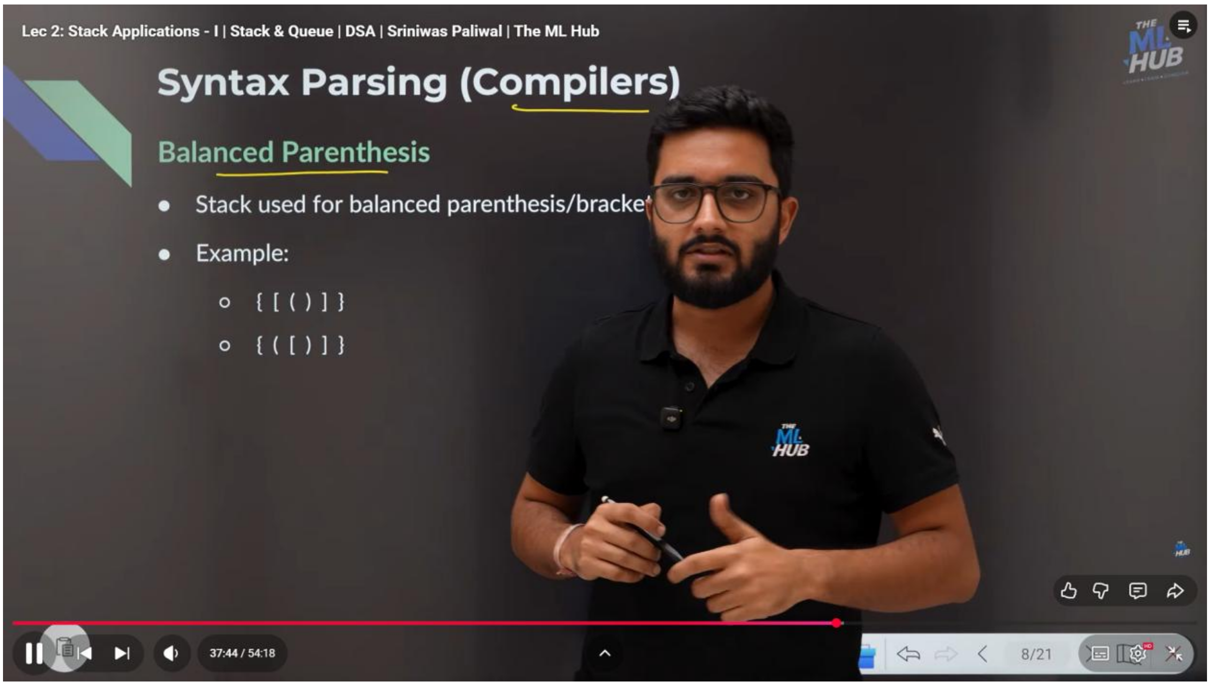

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Q. What will be the output of the following code snippet ?
("A:", n)
- 1)
("A done:", n)
n):
0:
t("B:", n)
(n - 2)
"B done:", n)
funA(3)
funB(2)
n=3
n = 2
Return at line 13
return at line 4
O/P : A: 3
B: 2
+ 45
25:25 / 54:18
6/21

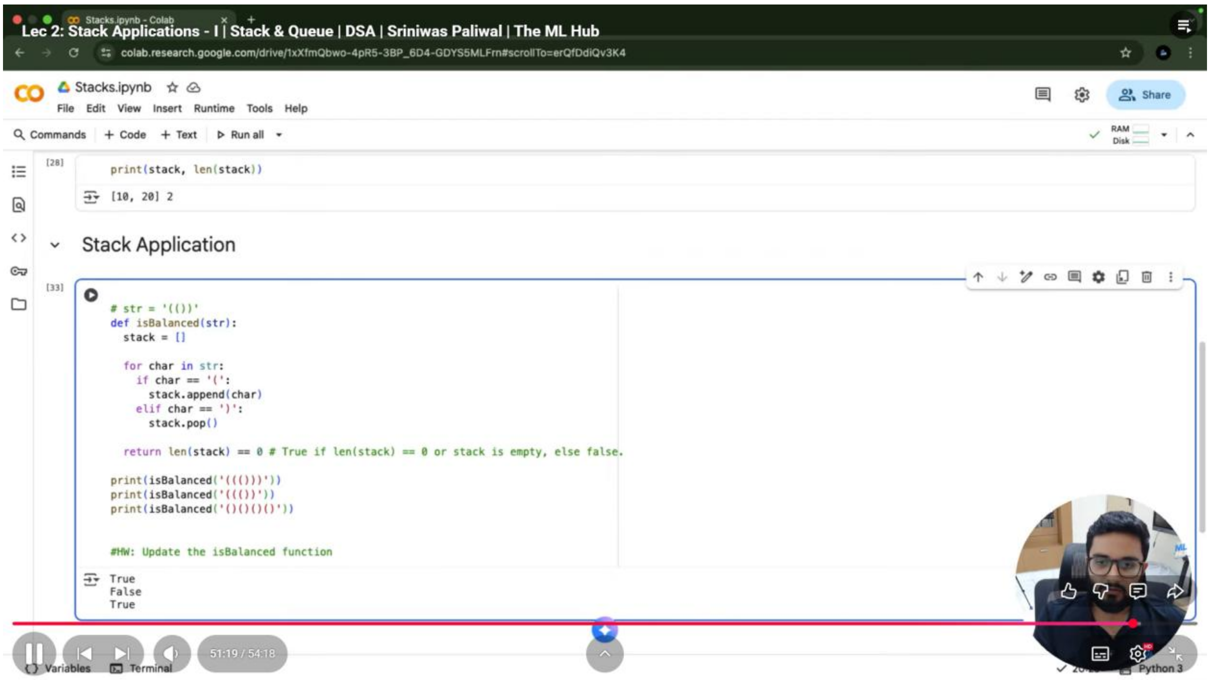

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Q. What will be the output of the following code snippet ?
def funcA(n):
if n > 0:
print("A:", n)
funcB(n - 1)
print("A done:", n)
def funcB(n):
if n > 0:
print("B:", n)
print("B done:", n)
funcA(3)
funA(3)
funB(2)
funcA(0)
n=3
n = 2
n = 0
return at line 13
return at line 4
return at line 10
pop
O/P :
A: 3
B: 2
26:26 / 54:18
6/21

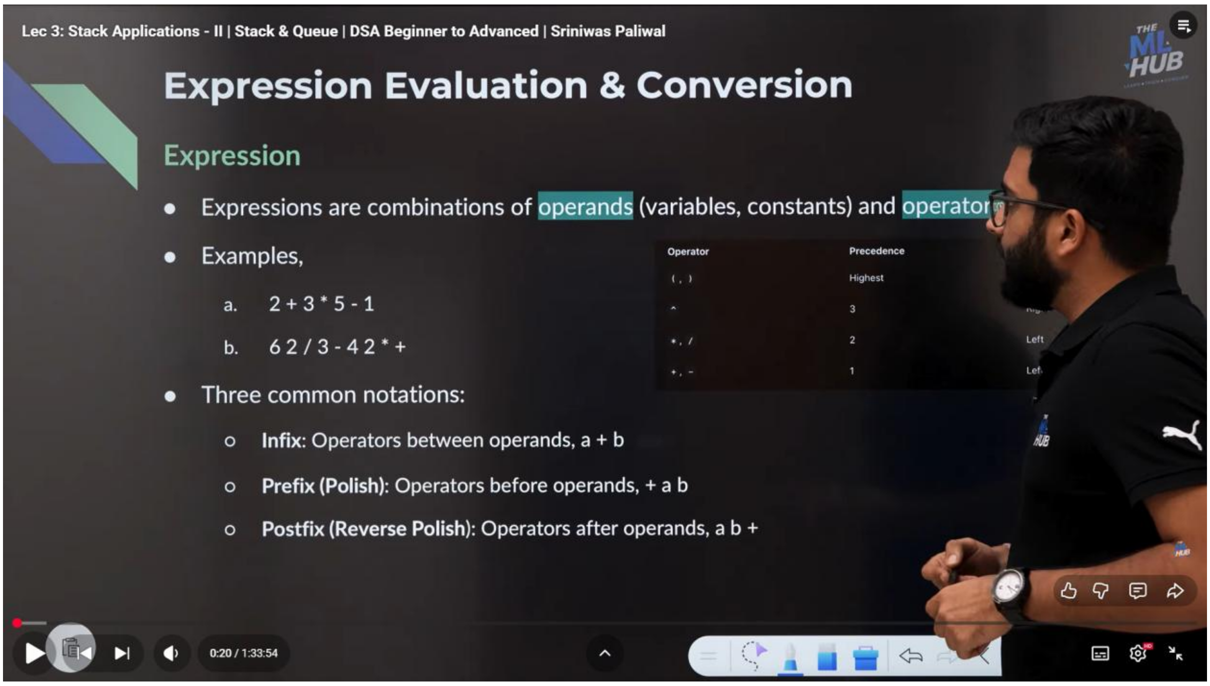

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Q. What will be the output of the following code snippet ?
def funcA(n):
if n > 0:
print("A:", n)
funcB(n - 1)
print("A done:", n)
print("B done:", n)
funcA(3)
funA(3)
funB(2)
funcA(0)
n=3
n = 2
n = 0
return at line 13
return at line 4
return at line 10
pop
pop
O/P :
A: 3
B: 2
B done : 2
27:37 / 54:18
6/21

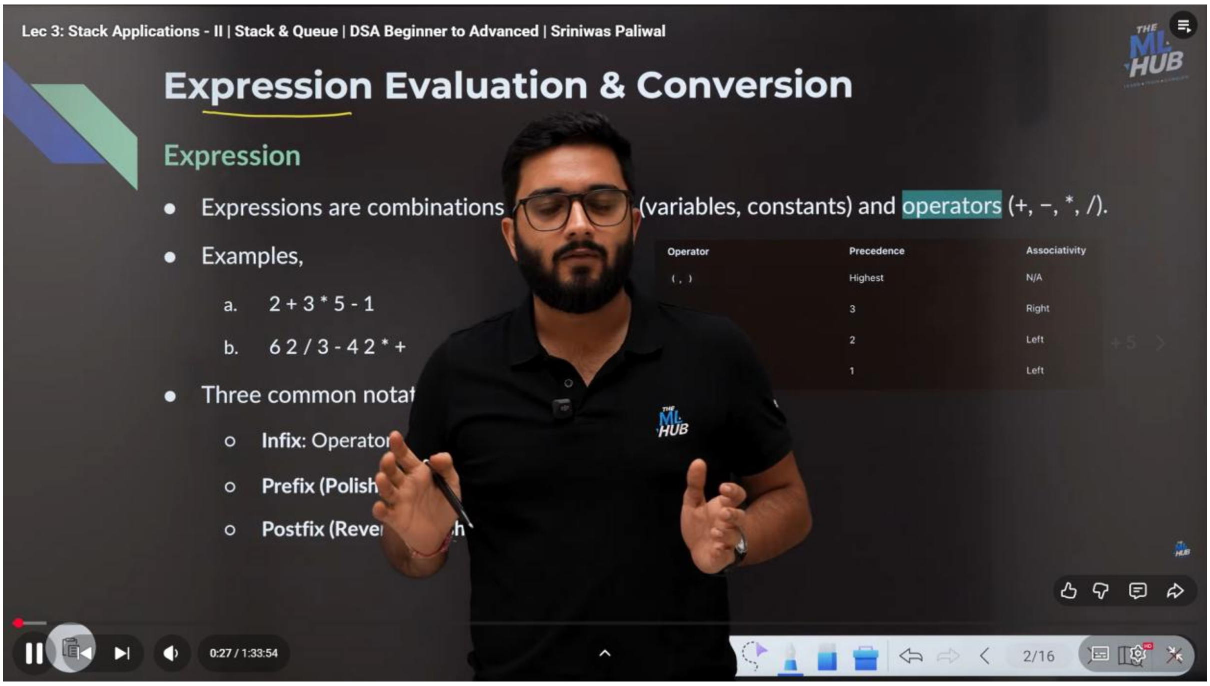

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Q. What will be the output of the following code snippet ?
def funcA(n):
if n > 0:
print("A:", n)
funcB(n - 1)
print("A done:", n)
def funcB(n):
if n > 0:
print("B:", n)
funcA(n - 2)
print("B done:", n)
pop
pop
pop
funA(3)
funB(2)
funcA(0)
n=3
n=2
n=0
returns at line 13
return at line 4
return at line 10
O/P :
A: 3
B: 2
B done : 2
A done : 3
+25
28:06 / 54:18
6/21

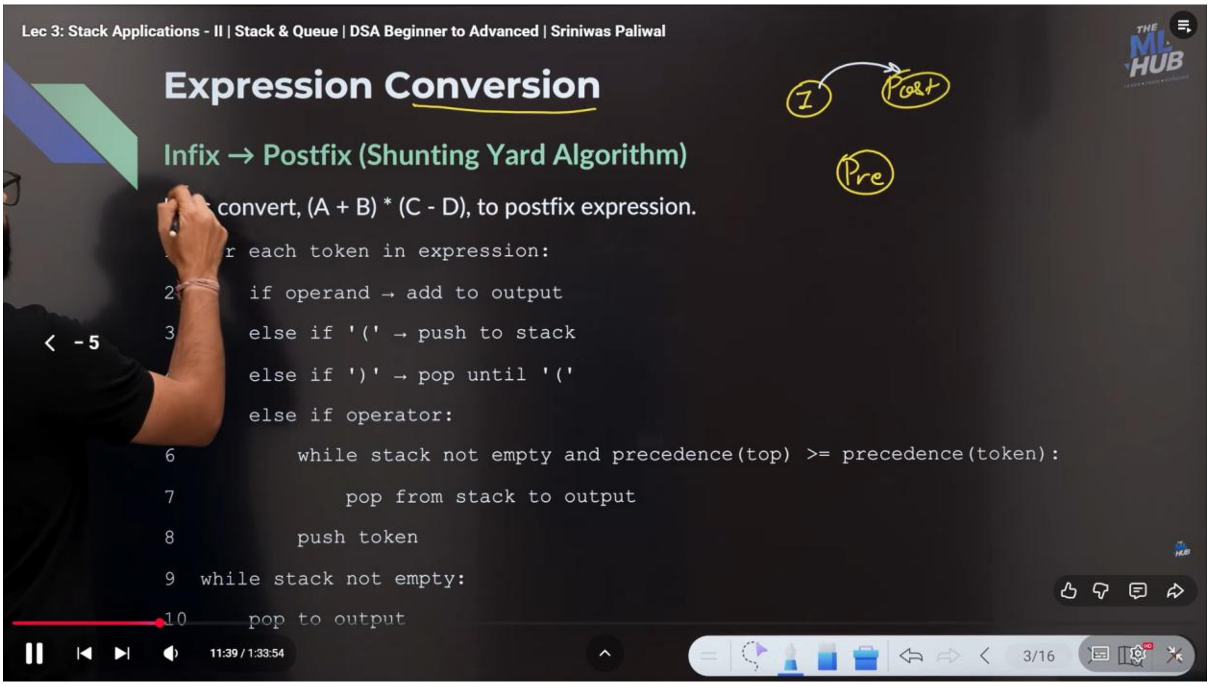
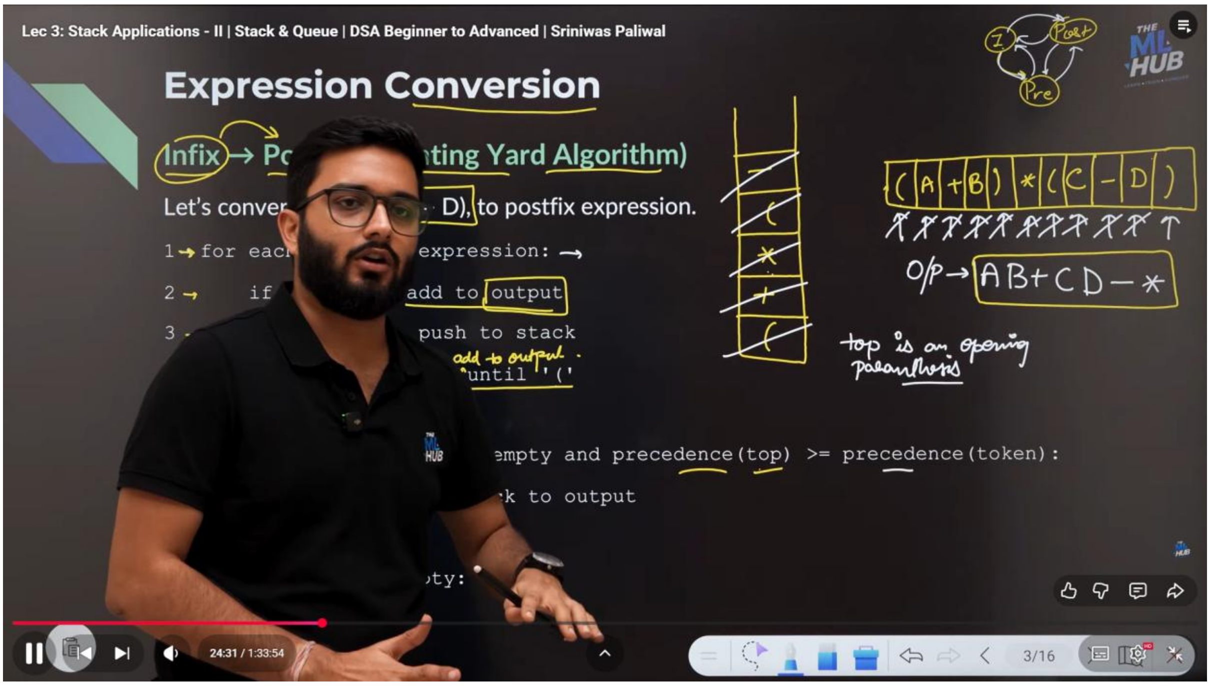

Undo / Redo in Editors
Editor Operations
Undo (Ctrl + Z): Pop last action
Redo (Ctrl + Y): Use another stack to restore popped actions
Example: Typing → “A”, “AB”, “ABC”
Undo 1 → “AB”
Undo 2 → “A”
Redo 1 → “AB”
Redo 2 → “ABC”
7/21

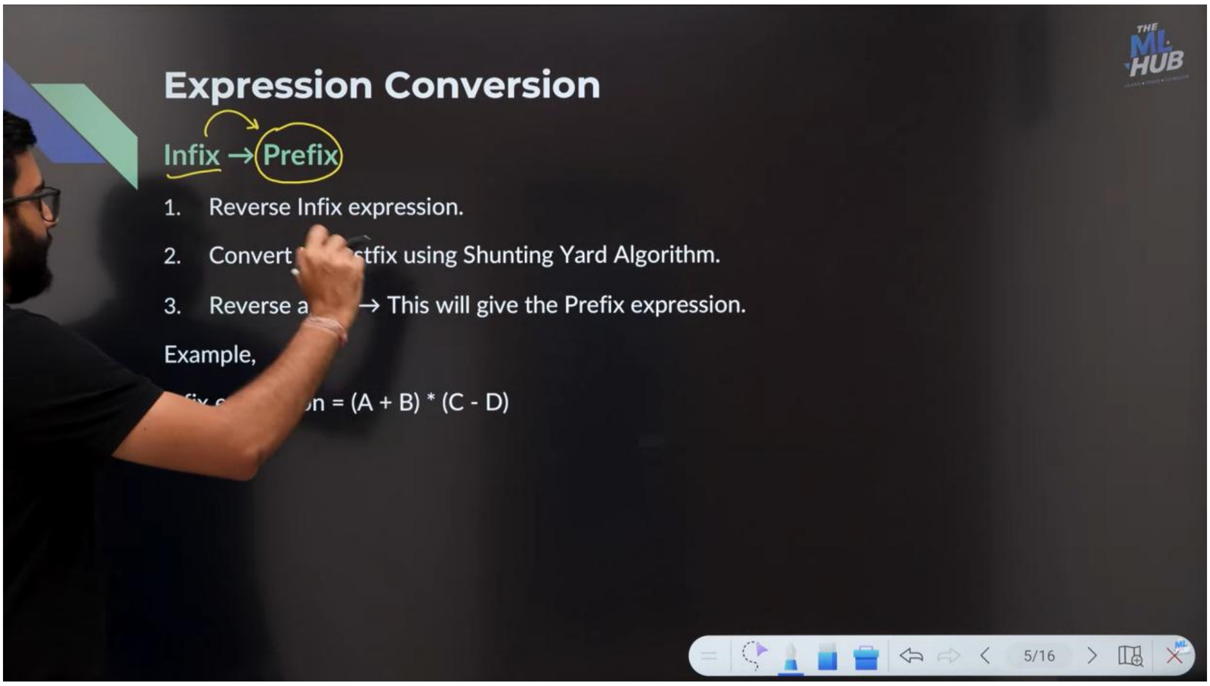

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
Syntax Parsing (Compilers)
Balanced Parenthesis
Stack used for balanced parenthesis/bracke
Example:
{ [( )] }
{ ([ )] }
37:44 / 54:18
8/21

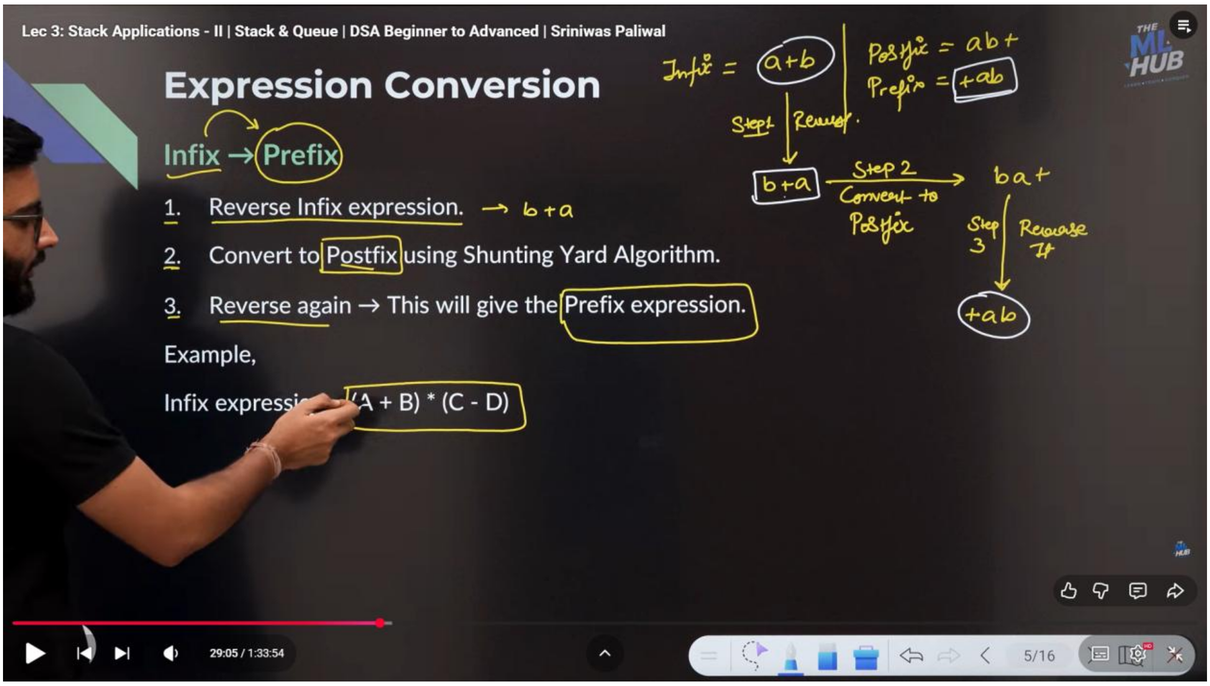

Lec 2: Stack Applications - I | Stack & Queue | DSA | Sriniwas Paliwal | The ML Hub
colab.research.google.com/drive/1xXfmQbwo-4pR5-3BP_6D4-GDYS5MLFrn#scrollTo=erQfDdiQv3K4
Stacks.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
print(stack, len(stack))
[10, 20] 2
Stack Application
# str = '(())'
def isBalanced(str):
  stack = []
  for char in str:
    if char == '(':
      stack.append(char)
    elif char == ')':
      stack.pop()
  return len(stack) == 0 # True if len(stack) == 0 or stack is empty, else false.
print(isBalanced('((()))'))
print(isBalanced('((())'))
print(isBalanced('()()()()'))
#HW: Update the isBalanced function
True
False
True
51:19 / 54:18
Variables
Terminal

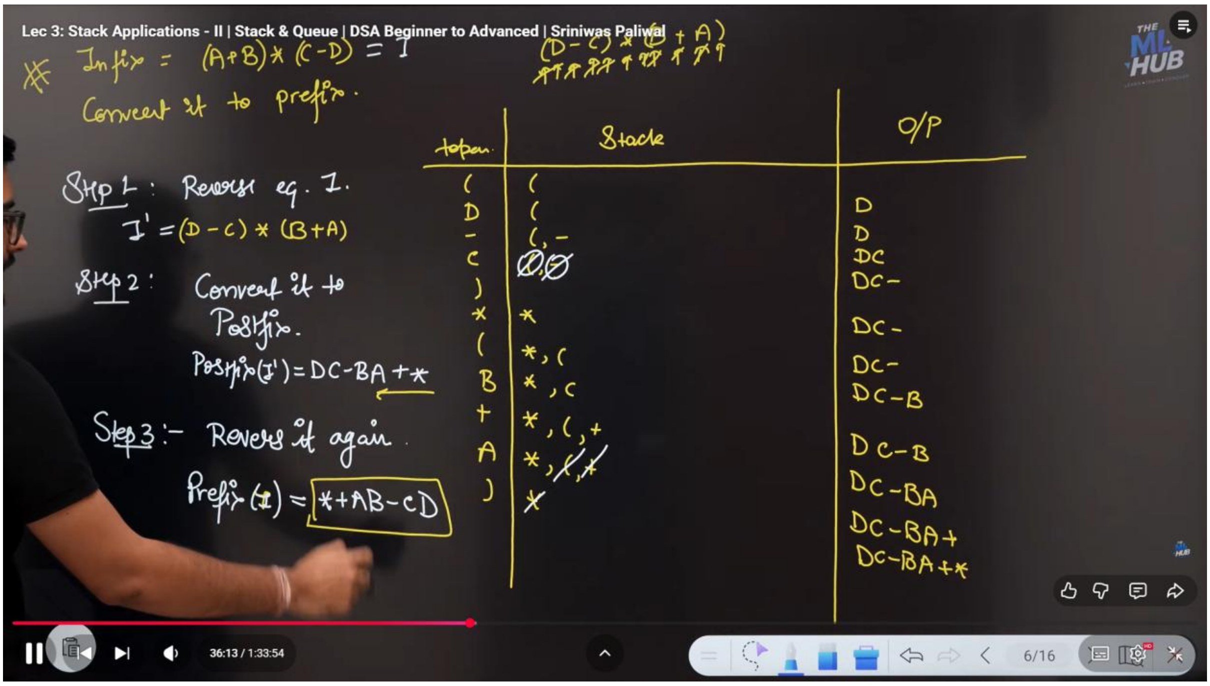

Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Evaluation & Conversion
Expression
Expressions are combinations of operands (variables, constants) and operator
Examples,
a. 2 + 3 * 5 - 1
b. 6 2 / 3 - 4 2 * +
Operator
Precedence
(,)
Highest
^
3
*, /
2
Left
+, -
1
Three common notations:
Infix: Operators between operands, a + b
Prefix (Polish): Operators before operands, + a b
Postfix (Reverse Polish): Operators after operands, a b +
0:20 / 1:33:54


Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Evaluation & Conversion
Expression
Expressions are combinations
(variables, constants) and operators (+, −, *, /).
Examples,
a. 2 + 3 * 5 - 1
b. 6 2 / 3 - 4 2 * +
Operator
Precedence
Associativity
(,)
Highest
N/A
3
Right
2
Left
1
Left
Three common notat
Infix: Operato
Prefix (Polish
Postfix (Reve
0:27 / 1:33:54
2/16

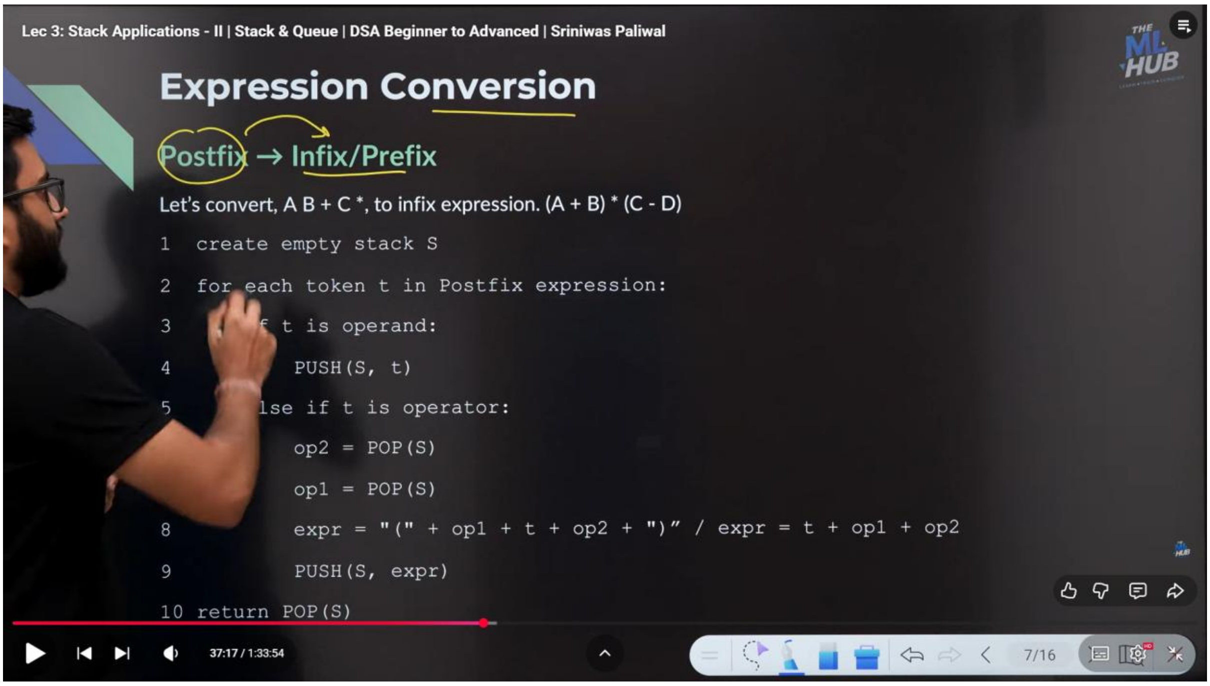

Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Conversion
I → Post
Pre
Infix → Postfix (Shunting Yard Algorithm)
convert, (A + B) * (C - D), to postfix expression.
r each token in expression:
if operand → add to output
else if '(' → push to stack
else if ')' → pop until '('
else if operator:
while stack not empty and precedence(top) >= precedence(token):
pop from stack to output
push token
while stack not empty:
pop to output
11:39 / 1:33:54
3/16

Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Conversion
Infix → Po ting Yard Algorithm)
Let's conver D), to postfix expression.
1 → for eac expression:
2 → if add to output
3 push to stack
add to output
until '('
empty and precedence(top) >= precedence(token):
k to output
ty:
((A + B) * (C - D))
O/P → AB+CD-*
top is an opening paranthesis
24:31 / 1:33:54
3/16


Expression Conversion
Infix → Prefix
1. Reverse Infix expression.
2. Convert
stfix using Shunting Yard Algorithm.
3. Reverse a
→ This will give the Prefix expression.
Example,
n = (A + B) * (C - D)
5/16

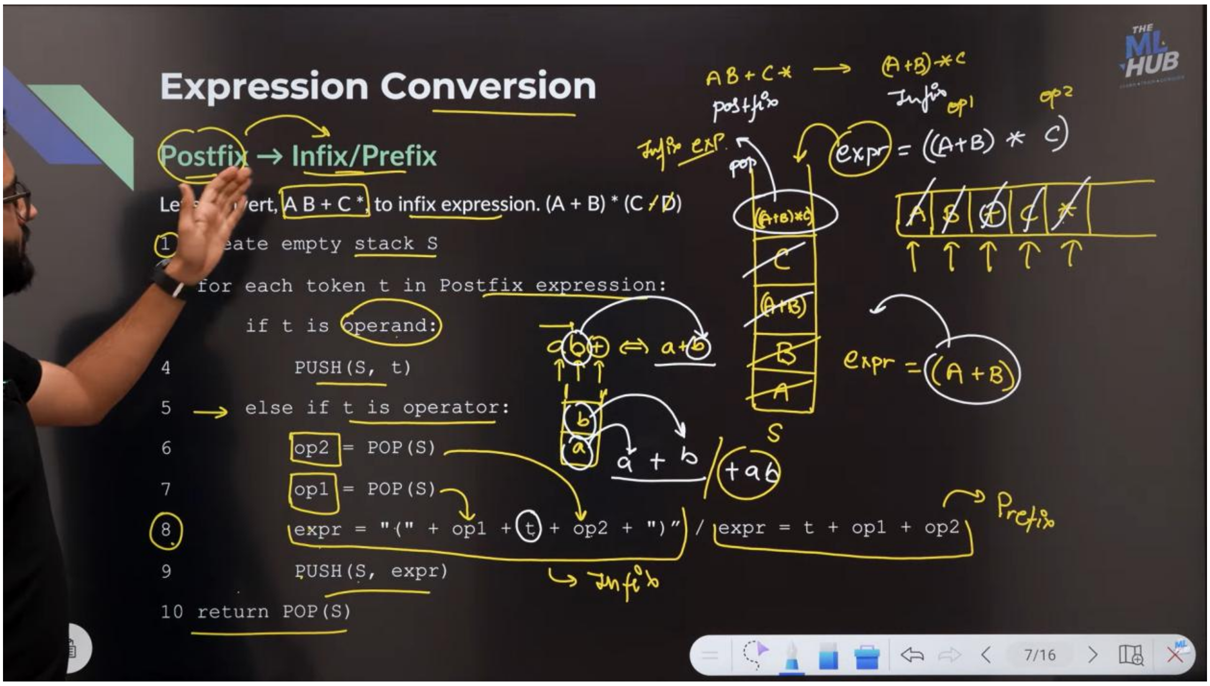

Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Conversion
Infix → Prefix
1. Reverse Infix expression. → b+a
2. Convert to Postfix using Shunting Yard Algorithm.
3. Reverse again → This will give the Prefix expression.
Example,
Infix expressi
A + B) * (C - D)
Infix = a+b
Postfix = ab+
Prefix = +ab
Step1 Reverse.
b+a
Step 2 Convert to Postfix
ba+
Step 3 Reverse It
+ab
29:05 / 1:33:54
5/16

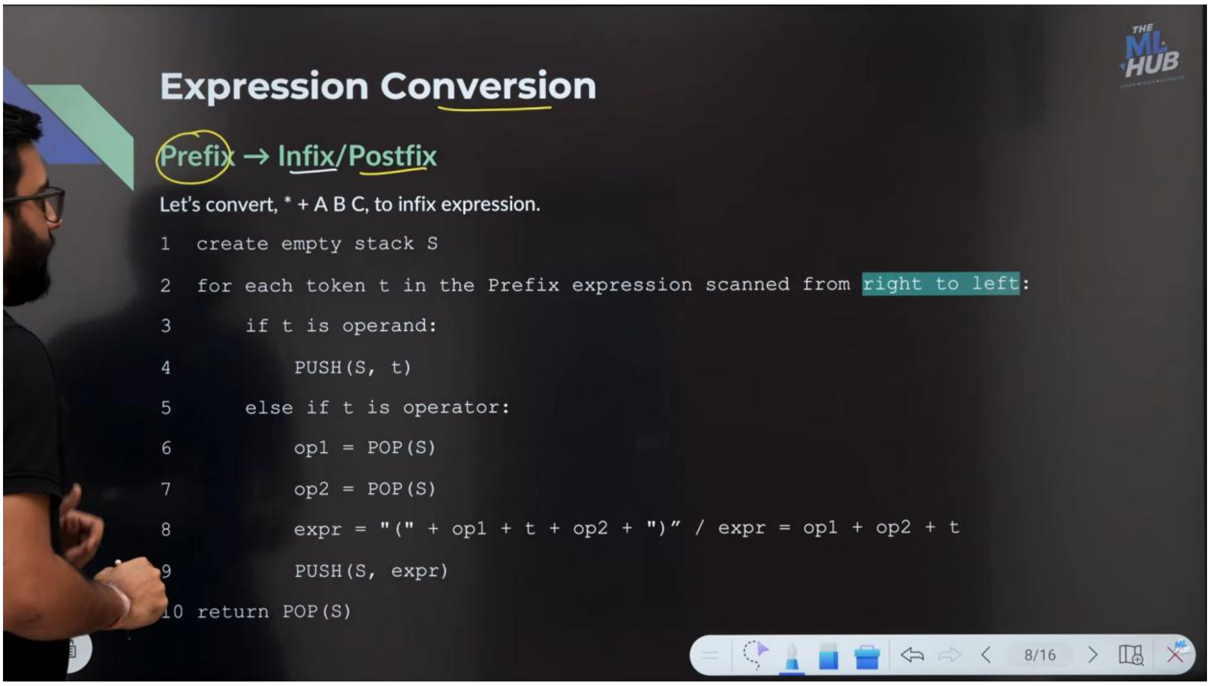


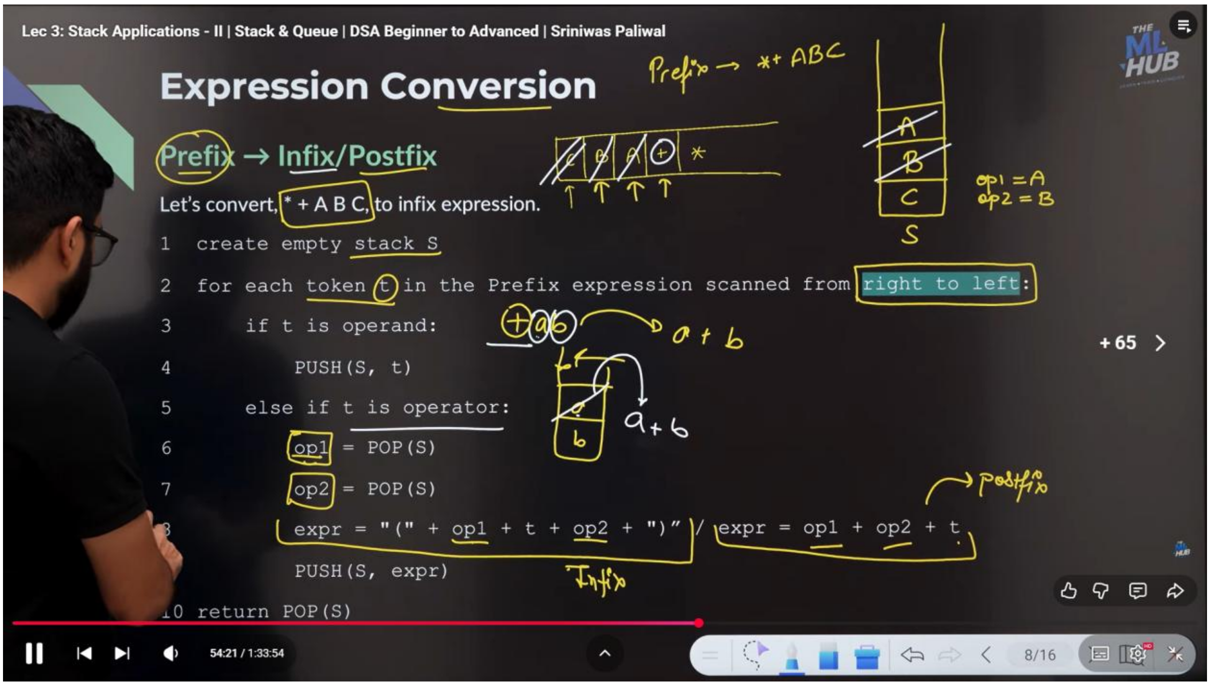
Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Infix = (A+B) * (C-D) = I
Convert it to prefix.
Step 1: Reverse eq. I.
I' = (D-C) * (B+A)
Step 2: Convert it to Postfix.
Postfix(I') = DC-BA+*
Step 3:- Reverse it again.
Prefix(I) = *+AB-CD
token
Stack
O/P
(D - C) * (B + A)
D
D
DC
DC-
DC-
DC-
DC-B
DC-B
DC-BA
DC-BA+
DC-BA+*
36:13 / 1:33:54
6/16

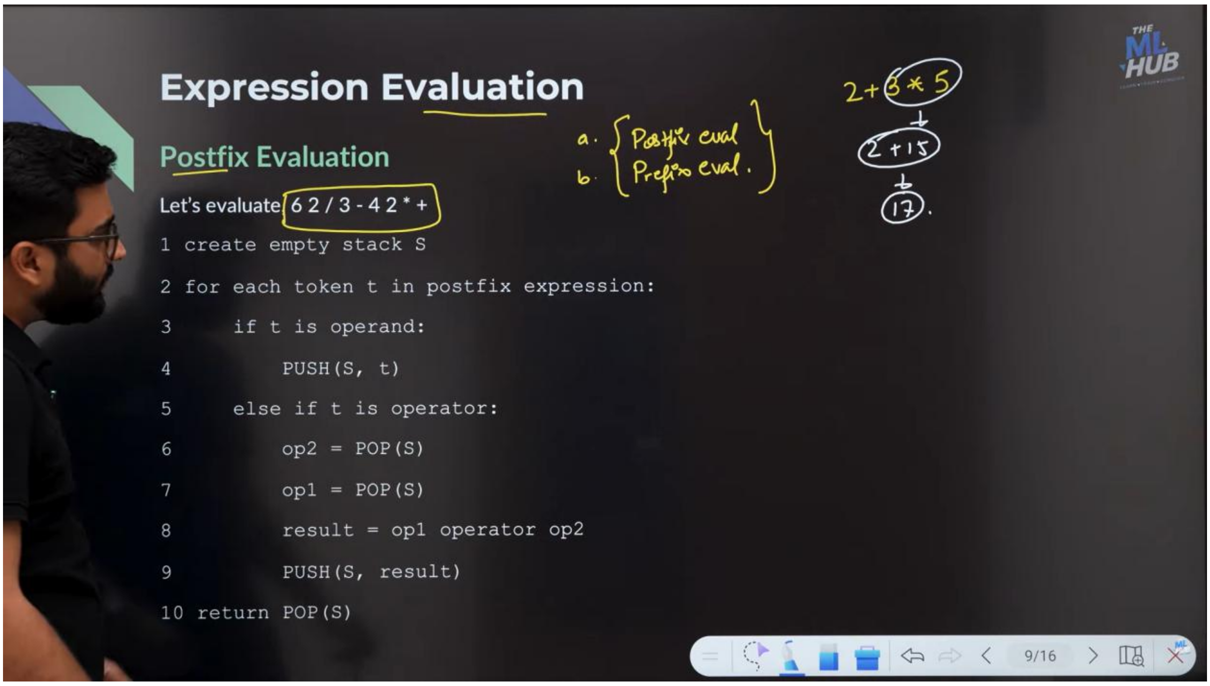


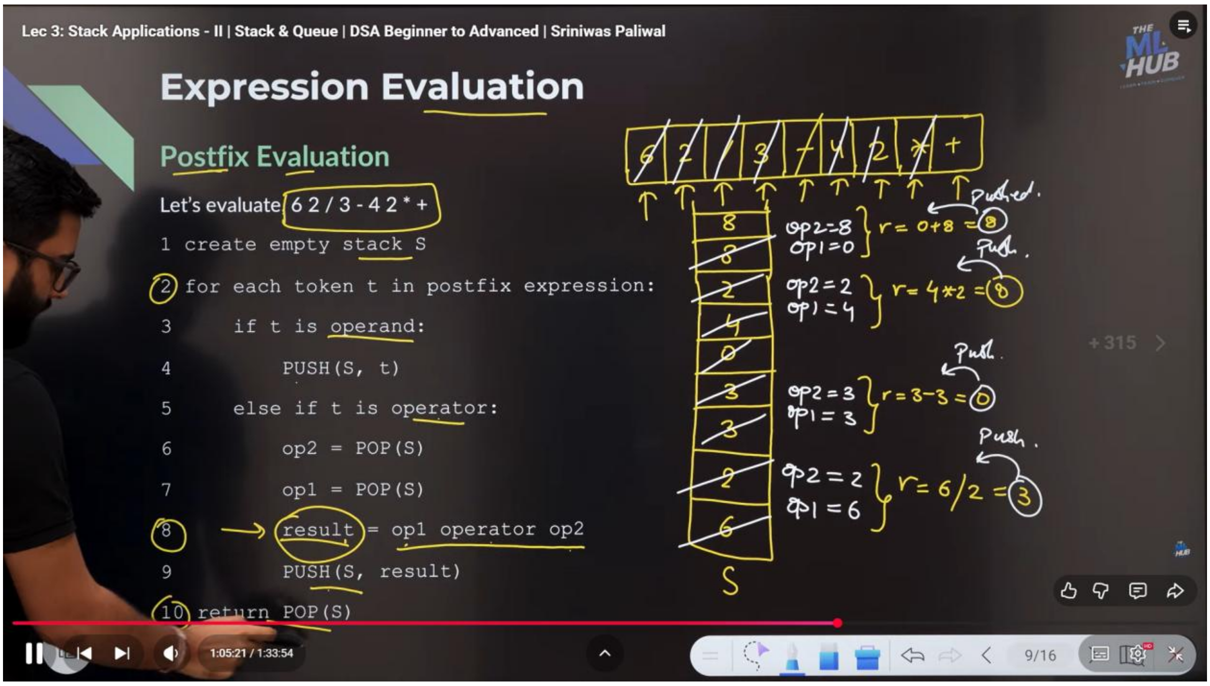
Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Conversion
Postfix → Infix/Prefix
Let's convert, A B + C *, to infix expression. (A + B) * (C - D)
1 create empty stack S
2 for each token t in Postfix expression:
3 if t is operand:
4 PUSH(S, t)
5 else if t is operator:
op2 = POP(S)
op1 = POP(S)
8 expr = "(" + op1 + t + op2 + ")" / expr = t + op1 + op2
9 PUSH(S, expr)
10 return POP(S)
37:17 / 1:33:54
7/16

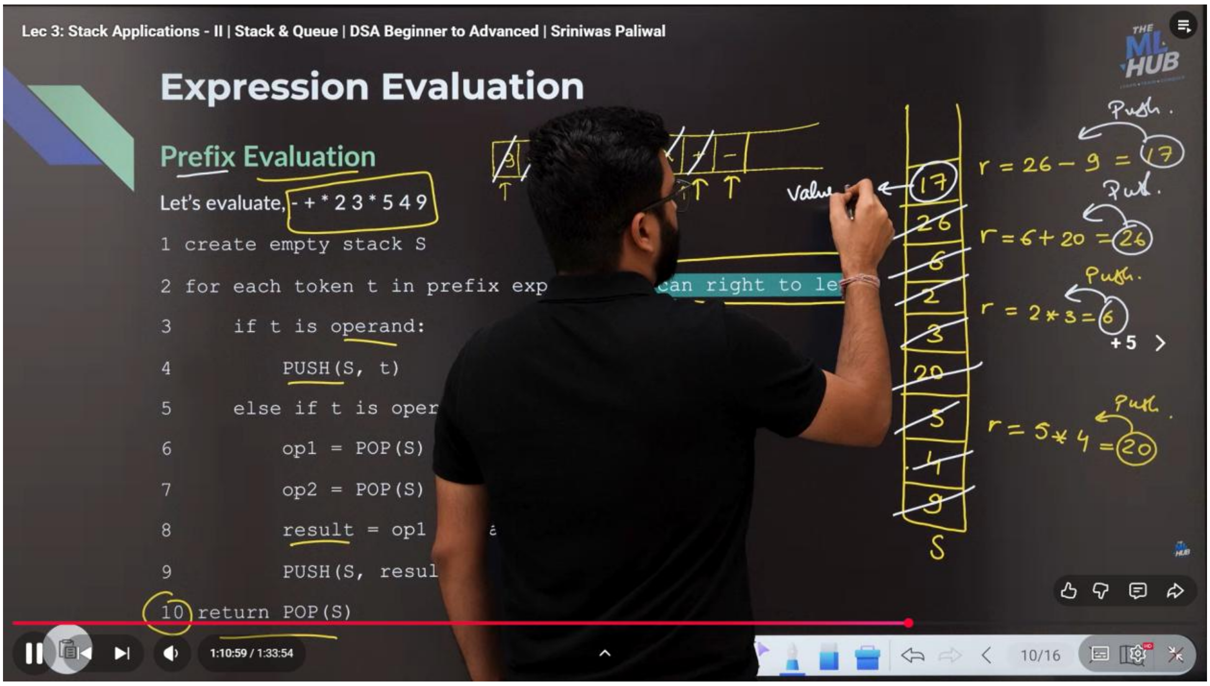

Expression Conversion
Postfix → Infix/Prefix
Le ert, A B + C *, to infix expression. (A + B) * (C / D)
eate empty stack S
for each token t in Postfix expression:
if t is operand:
PUSH(S, t)
else if t is operator:
op2 = POP(S)
op1 = POP(S)
expr = "(" + op1 + t + op2 + ")" / expr = t + op1 + op2
PUSH(S, expr)
return POP(S)
AB+C* → (A+B)*C
postfix
Infix
op1
op2
expr = ((A+B) * C)
pop
S
expr = (A+B)
a + b
+ab
Prefix
Infix
7/16


Expression Conversion
Prefix → Infix/Postfix
Let's convert, * + A B C, to infix expression.
create empty stack S
for each token t in the Prefix expression scanned from right to left:
if t is operand:
PUSH(S, t)
else if t is operator:
op1 = POP(S)
op2 = POP(S)
expr = "(" + op1 + t + op2 + ")" / expr = op1 + op2 + t
PUSH(S, expr)
return POP(S)
8/16


Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Conversion
Prefix → Infix/Postfix
Let's convert, * + A B C, to infix expression.
1 create empty stack S
2 for each token t in the Prefix expression scanned from right to left:
3 if t is operand:
4 PUSH(S, t)
5 else if t is operator:
6 op1 = POP(S)
7 op2 = POP(S)
8 expr = "(" + op1 + t + op2 + ")" / expr = op1 + op2 + t
9 PUSH(S, expr)
10 return POP(S)
Prefix → *+ ABC
op1 = A
op2 = B
S
a + b
Infix
postfix
54:21 / 1:33:54
8/16
+65


Expression Evaluation
Postfix Evaluation
Let's evaluate 6 2 / 3 - 4 2 * +
1 create empty stack S
2 for each token t in postfix expression:
3 if t is operand:
4 PUSH(S, t)
5 else if t is operator:
6 op2 = POP(S)
7 op1 = POP(S)
8 result = op1 operator op2
9 PUSH(S, result)
10 return POP(S)
a. Postfix eval
b. Prefix eval.
2+3*5
2+15
17.
9/16


Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Evaluation
Postfix Evaluation
Let's evaluate 6 2 / 3 - 4 2 * +
1 create empty stack S
2 for each token t in postfix expression:
3 if t is operand:
4 PUSH(S, t)
5 else if t is operator:
6 op2 = POP(S)
7 op1 = POP(S)
8 result = op1 operator op2
9 PUSH(S, result)
10 return POP(S)
op2=8 op1=0 r = 0+8 = 8
Pushed.
op2=2 op1=4 r = 4*2 = 8
Push.
op2=3 op1=3 r = 3-3 = 0
Push.
op2=2 op1=6 r = 6/2 = 3
Push.
S
1:05:21 / 1:33:54
9/16


Lec 3: Stack Applications - II | Stack & Queue | DSA Beginner to Advanced | Sriniwas Paliwal
Expression Evaluation
Prefix Evaluation
Let's evaluate - + * 2 3 * 5 4 9
1 create empty stack S
2 for each token t in prefix exp
scan right to le
3 if t is operand:
4 PUSH(S, t)
5 else if t is oper
6 op1 = POP(S)
7 op2 = POP(S)
8 result = op1
9 PUSH(S, resul
10 return POP(S)
r = 26 - 9 = 17
Push.
r = 6 + 20 = 26
Push.
r = 2 * 3 = 6
Push.
r = 5 * 4 = 20
S
1:10:59 / 1:33:54
10/16