Data Structures

Dr.T.VEERAMANI

CSE/SIST

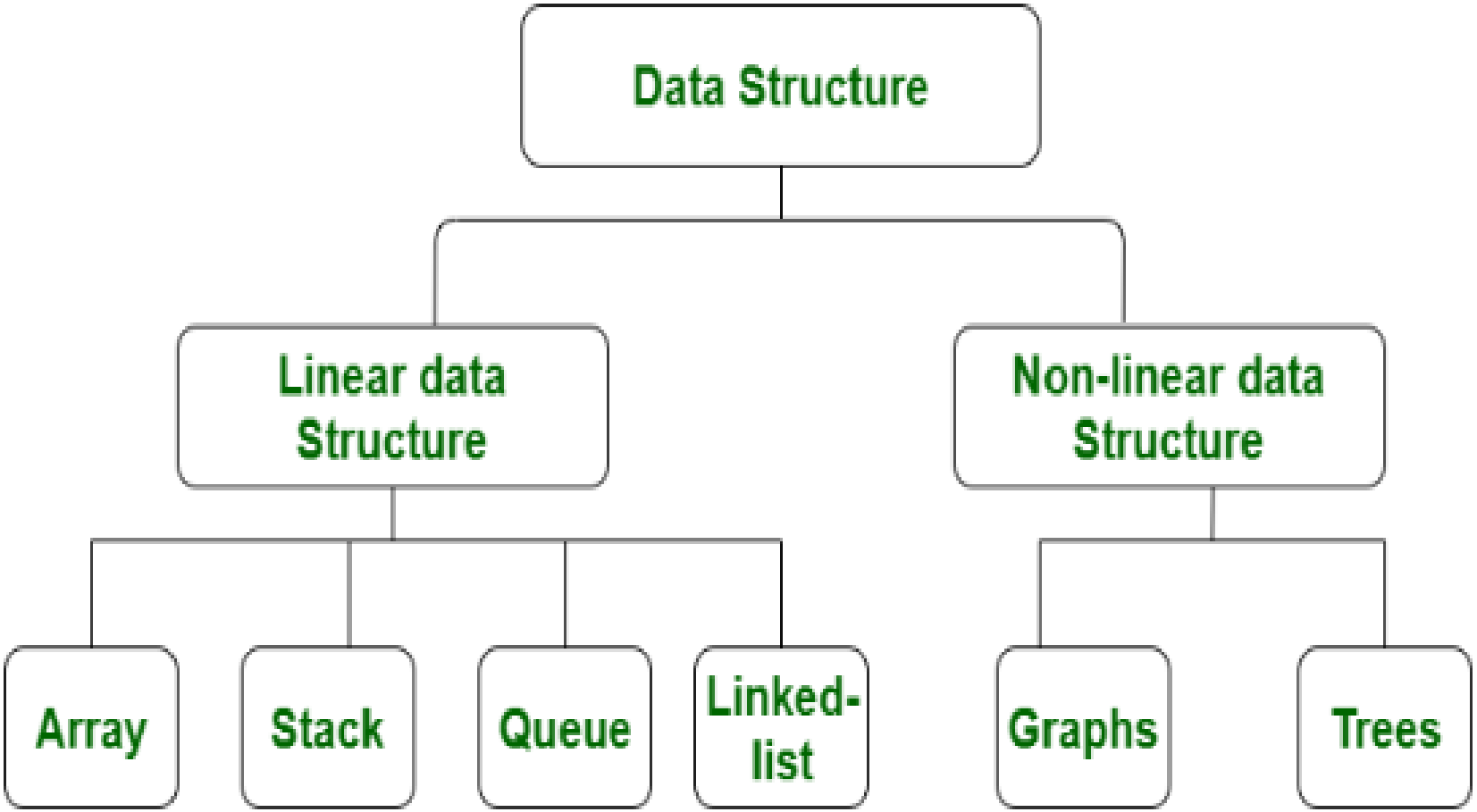
UNIT 3 STACKS

Topics:

Basic Stack Operations –

- Representation of a Stack using Arrays - Algorithm for Stack Operations –
- Stack Applications: Reversing list –
- Factorial Calculation –
- Infix to postfix Transformation –
- Evaluating Arithmetic Expressions

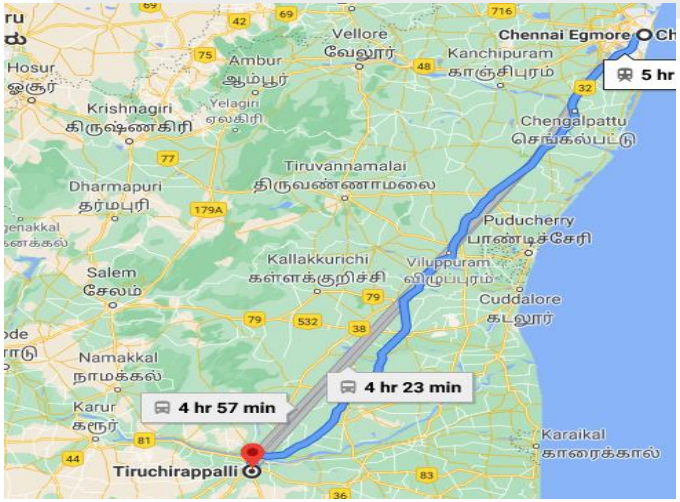
Data Structures

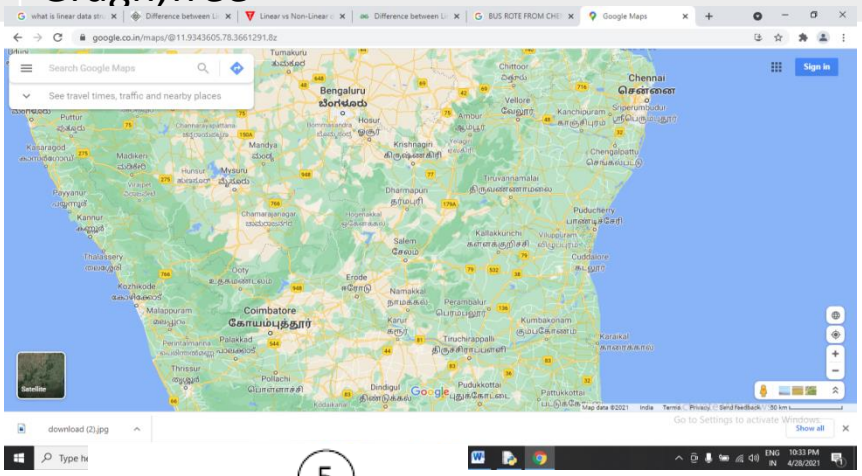


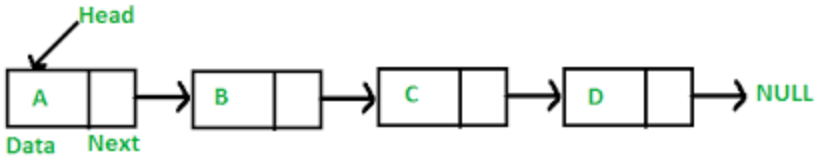
Define Data structure?

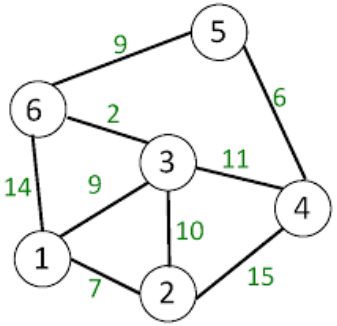
- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently

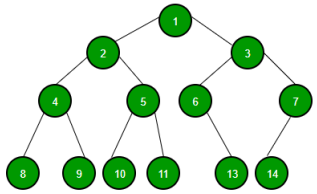
LINEAR-DS	NON LINEAR-DS
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner
In linear data structure, single level is involved.(SINGLE RUN ONLY) Stack,queue	Whereas in non-linear data structure, multiple levels are involved.(NOT TRAVERSAL DATA IN SINGLE RUN) Gragh,Tree












MAERSK


LUGGAGE

STACK

- A stack is a data structure that stores items in an Last-In/First-Out manner. This is frequently referred to as LIFO. (or)
- A list with the restriction that insertion and deletion can be performed only from one end ,called the top




Stacks

- Stack is a collection of objects
- Last-In First-Out
- Fundamental operations are Pushing & Popping
  - Push(R)

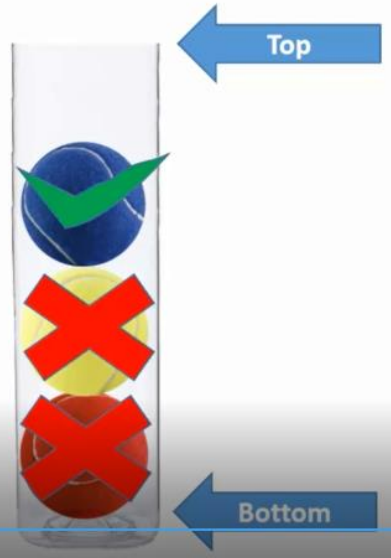
Stacks

- Stack is a collection of objects
- Last-In First-Out
- Fundamental operations are Pushing & Popping
  - Push(R)

Stacks

- Stack is a collection of objects
- Last-In First-Out
- Fundamental operations are Pushing & Popping
  - Push(R)
  - Push(Y)
  - Push(B)

Stacks



- Stack is a collection of objects
- Last-In First-Out
- Fundamental operations are Pushing & Popping
  - Push(R)
  - Push(Y)
  - Push(B)

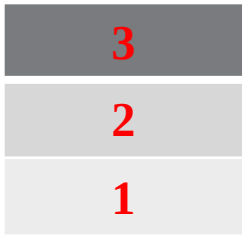
whether a stack is an abstract data type or a data structure?

ADT	DATA STRUCTURE
In an **Abstract Data Type (or ADT)**, there is a set of rules or description of the operations that are allowed on data. It is based on a user point of view i.e., how a user is interacting with the data. However, we can choose to implement those set of rules differently.	However, when we choose to implement a stack in a particular way, it organizes our data for efficient management and retrieval. So, it can be seen as a data structure also.

- It is an abstract data type (interface)
- Basic operations: pop(), push() and peek()
- **LIFO** structure: last in first out
- In most high level languages, a stack can be easily implemented either with arrays or linked lists
- A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack

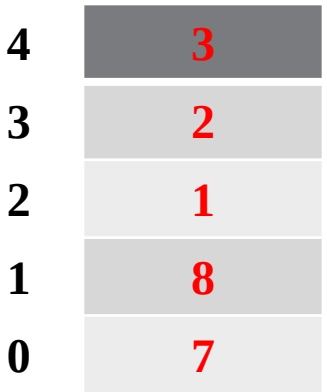
Stack

- **Linear Data Structures**
  - Collections of Similar **Data Type**
  - Limited Access is Possible
    - Set of **Rules and Principles** followed
      - **Insertion and Deletion** Possible from only one end
  - **Last In First Out (LIFO)**
  - **First In Last Out (FILO)**
    - **Insertion – Push()**
    - **Deletion – Pop()**
    - **End – Top() | peek()**
  - isEmpty()
  - isFull()
  - Search(),*** Time complexity O(1)
  - Traverse
  - Minimum Element
  - Maximum Element ,



Logical representation of Stack with Basic Operations

- For Memory allocation
  - Static Memory Allocation (Array)
  - Dynamic Memory Allocation (Linked List)



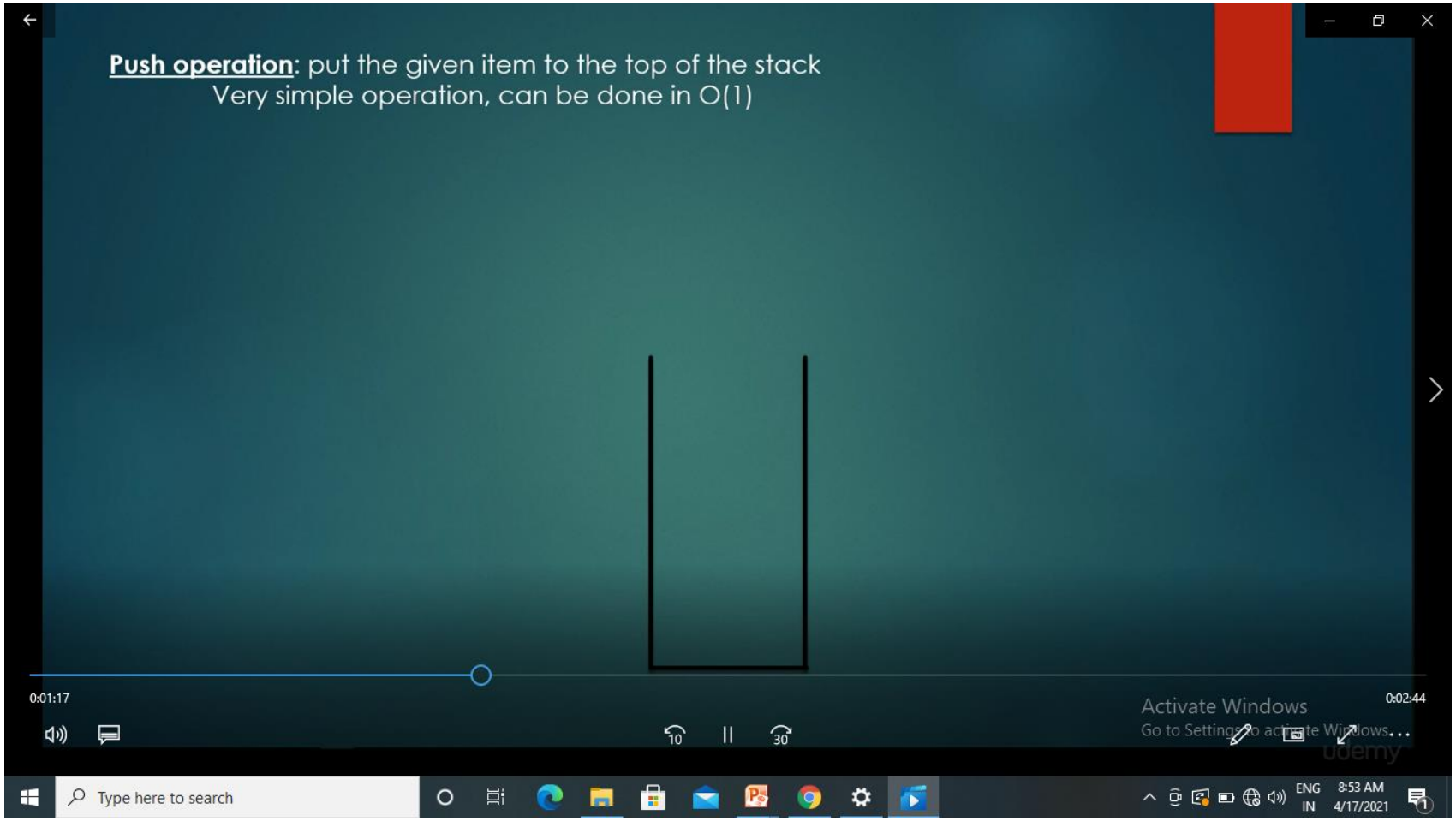
Size=5, top=-1

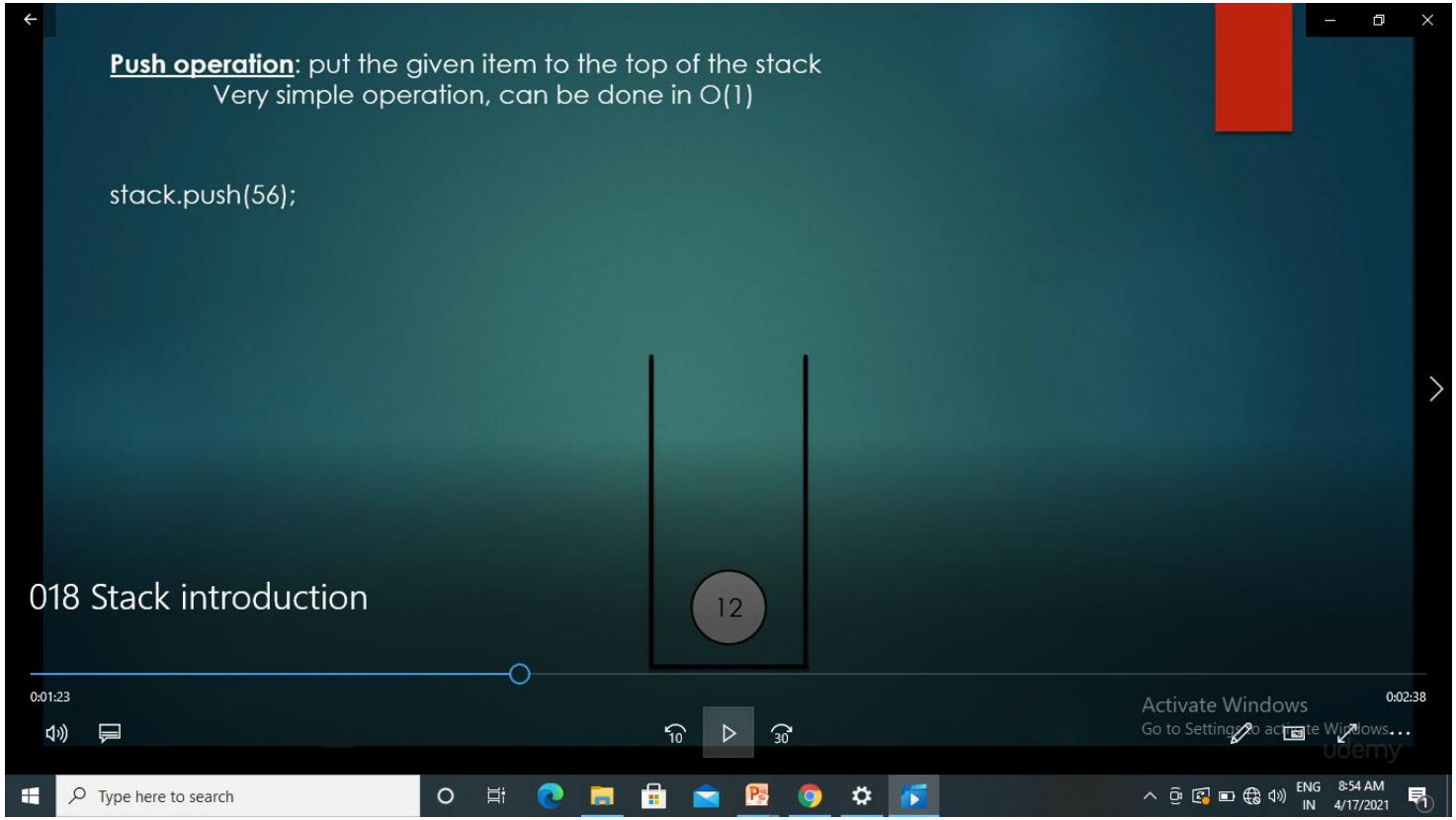
Pop() → Underflow Condition
Push(7)
Top++ →0
Push(8)
Top++→1
Pop()
Top--→0

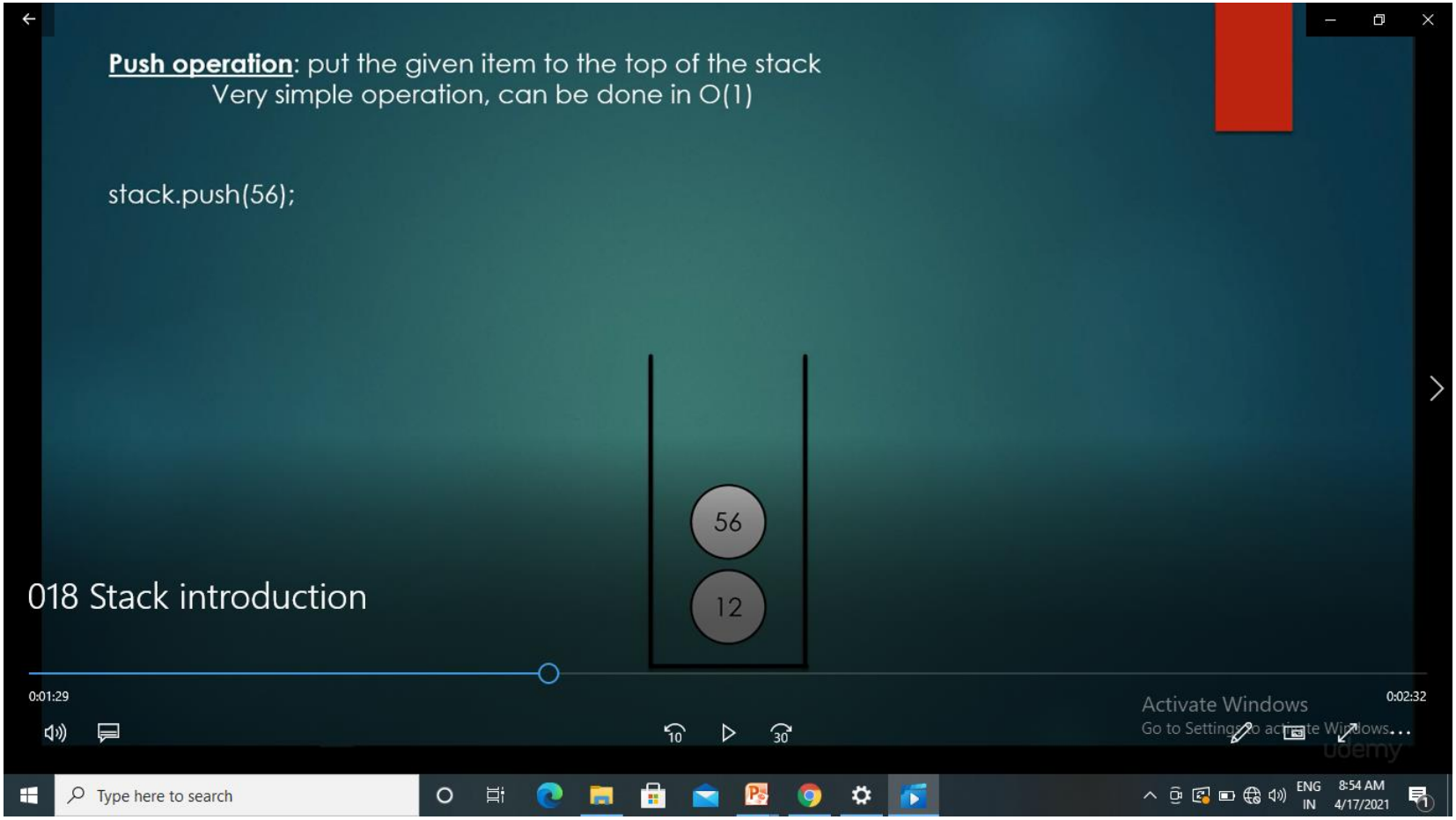
Top=size-1 → isFull()
Top=-1 →isEmpty()

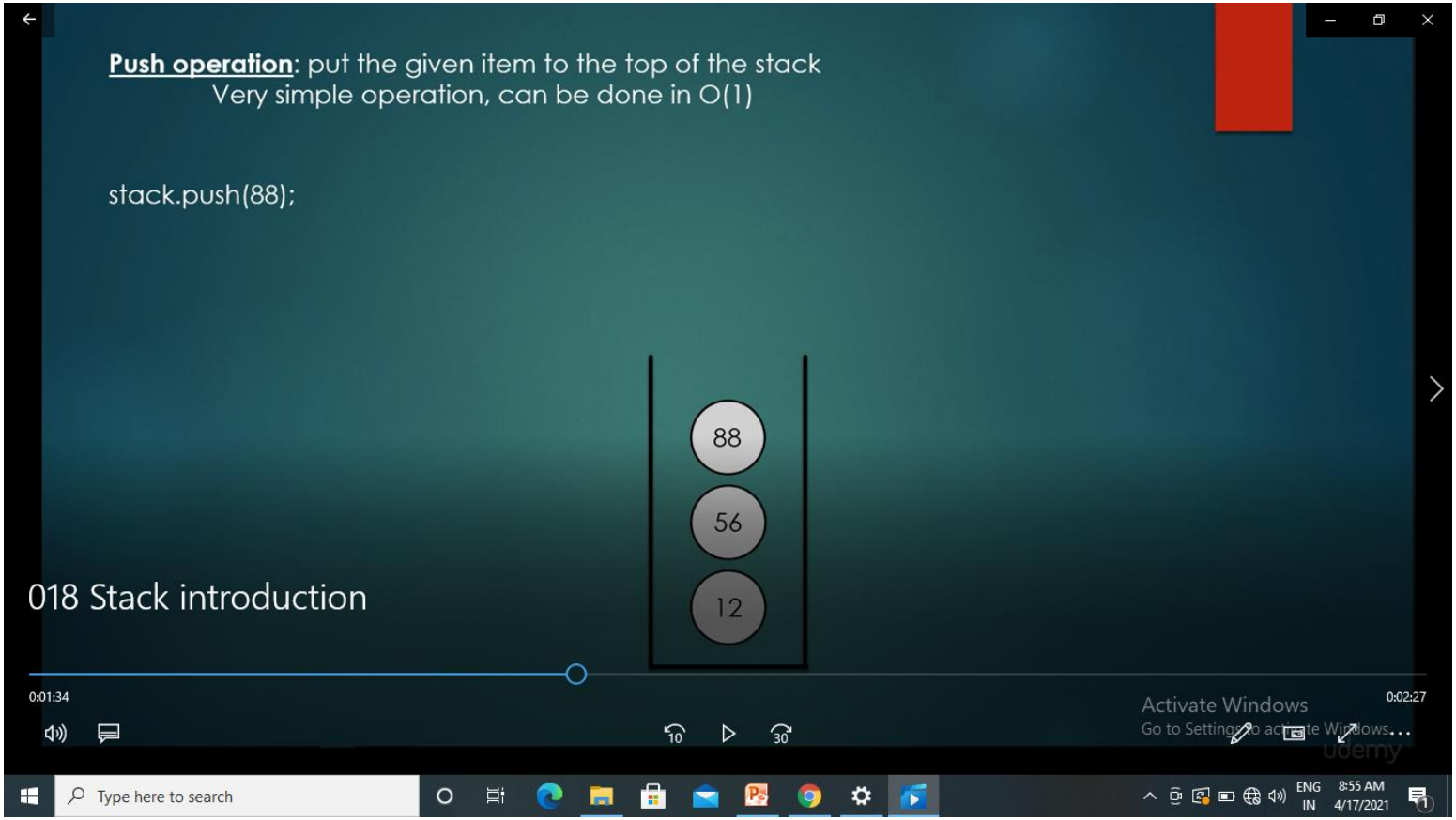
Functions

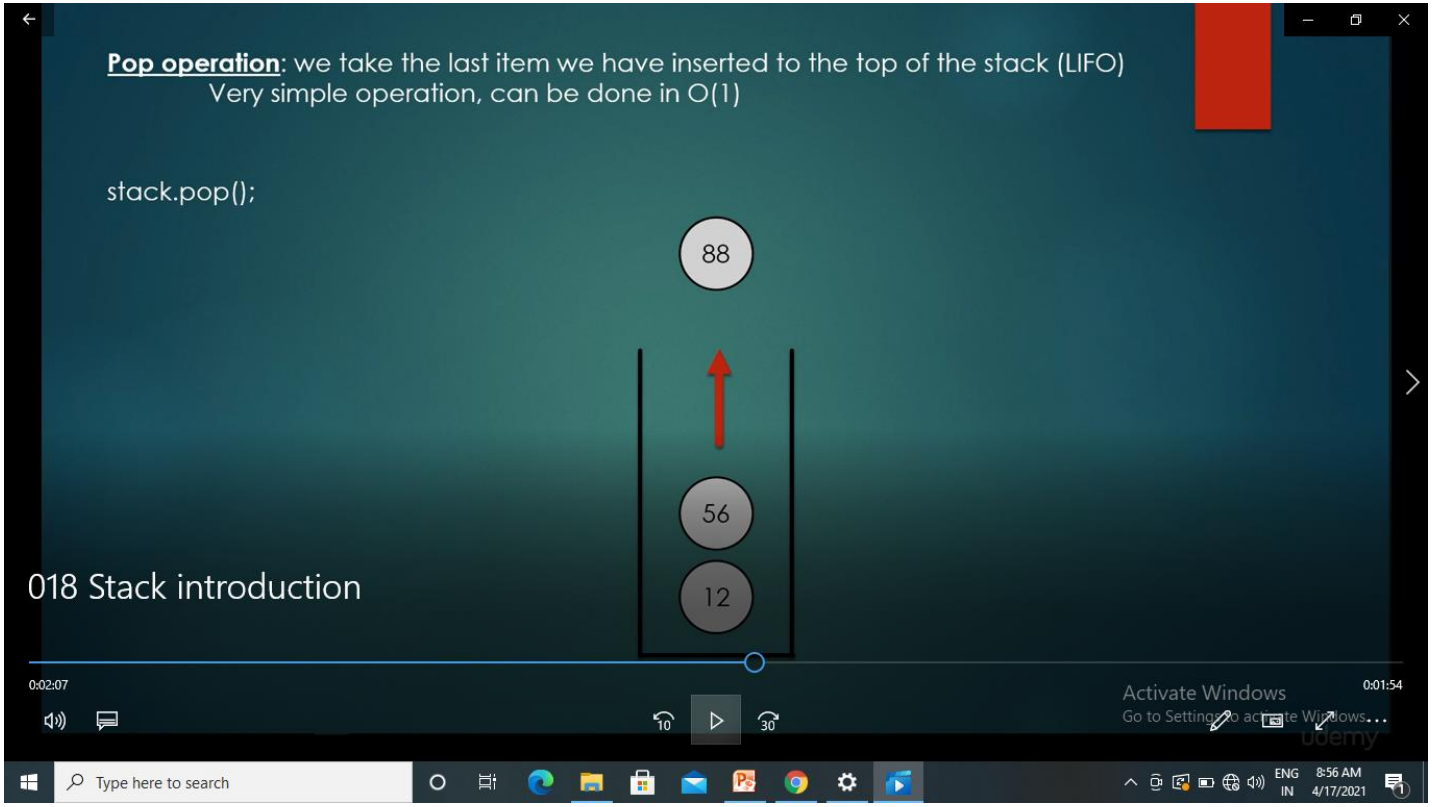
- empty() – Returns whether or not the stack is empty
- size() – Returns the size of the stack
- top() – Returns a reference to the top most element of the stack
- push(g) – Adds the element ‘g’ at the top of the stack
- pop() – Deletes the top most element of the stack

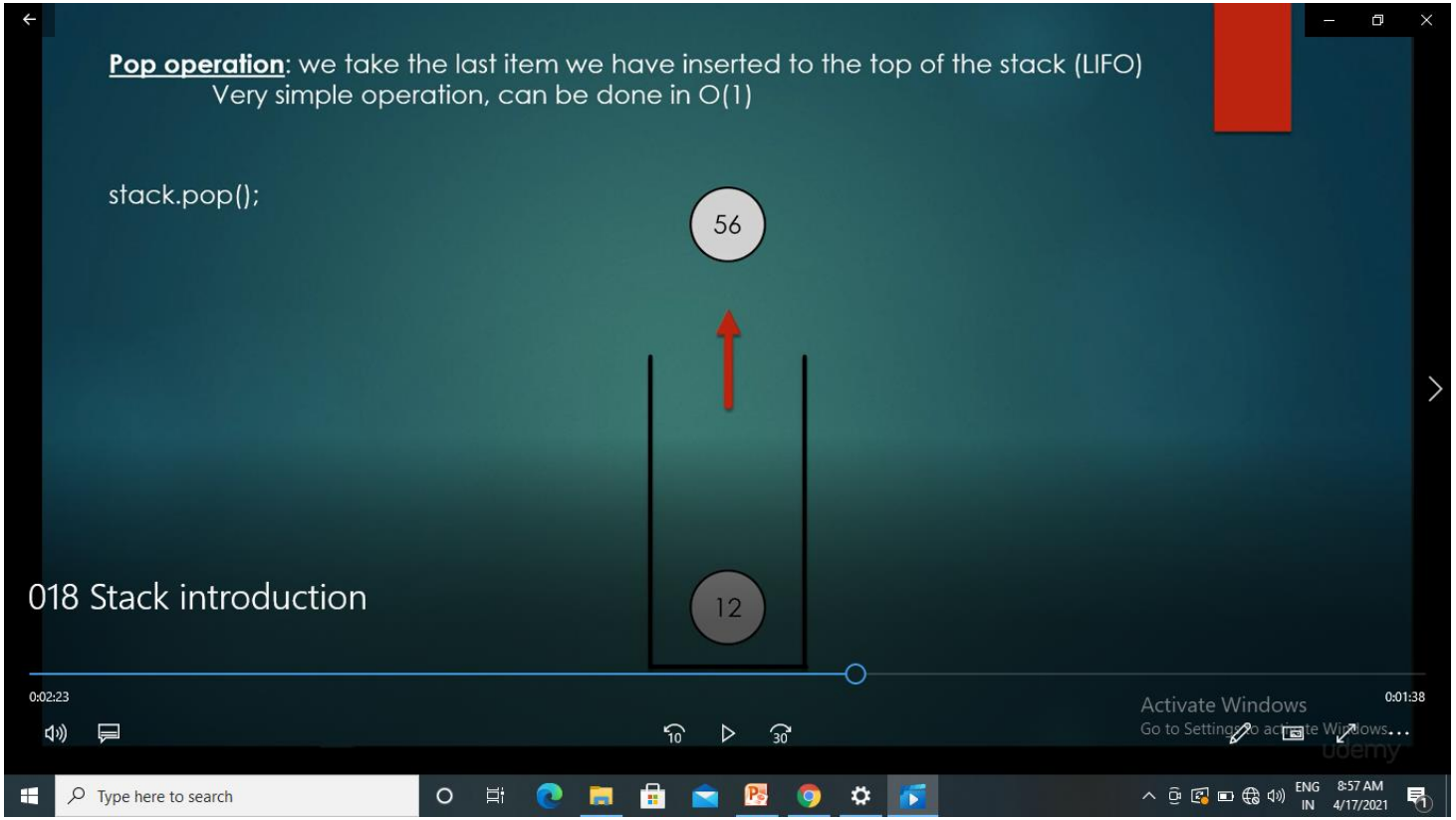

Push operation: put the given item to the top of the stack
Very simple operation, can be done in O(1)
0:01:17
0:02:44
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
8:53 AM
4/17/2021

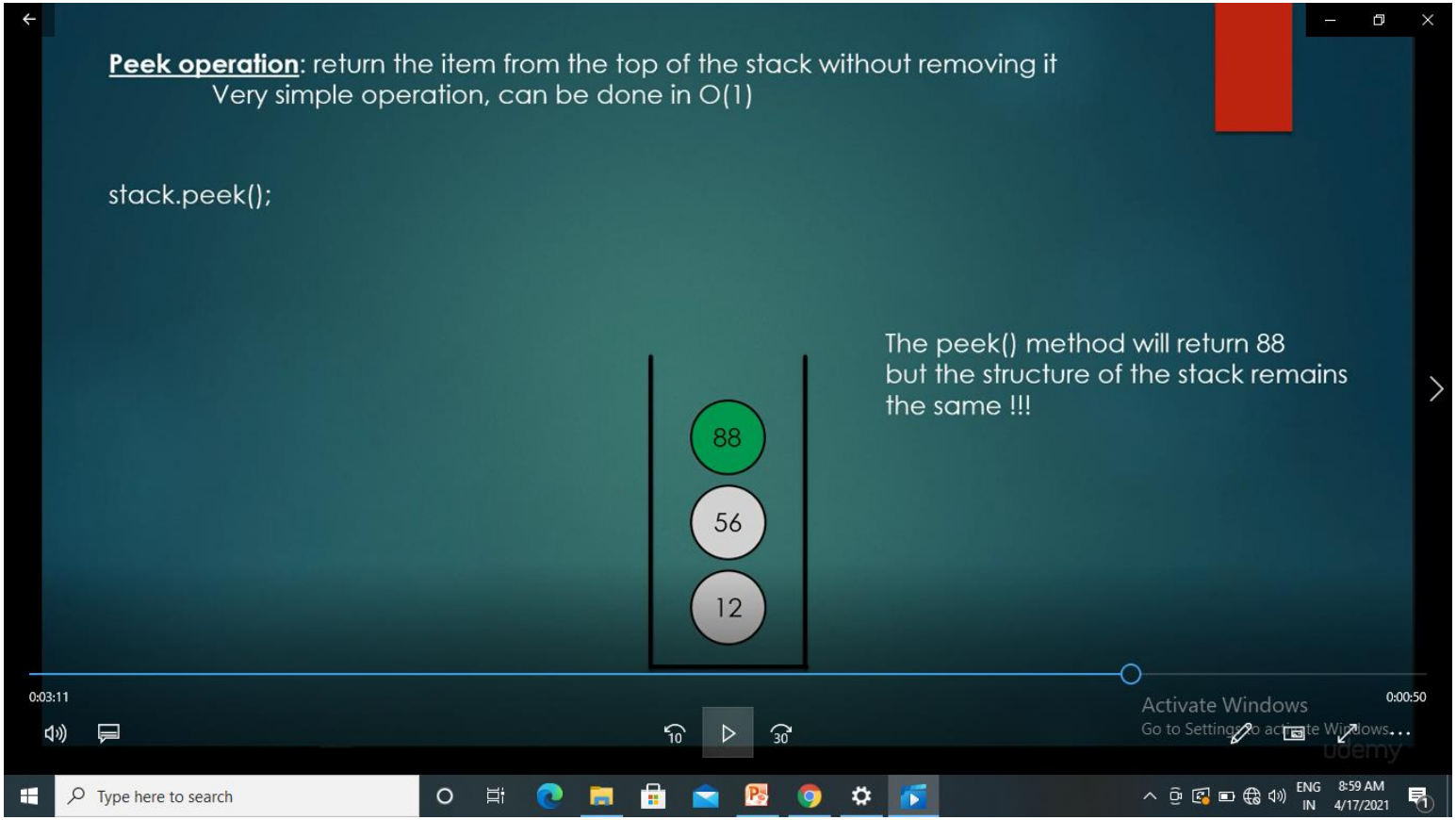

Push operation: put the given item to the top of the stack
Very simple operation, can be done in O(1)
stack.push(56);
12
018 Stack introduction
0:01:23
0:02:38
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
8:54 AM
4/17/2021


Push operation: put the given item to the top of the stack
Very simple operation, can be done in O(1)
stack.push(56);
56
12
018 Stack introduction
0:01:29
0:02:32
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
8:54 AM
4/17/2021


Push operation: put the given item to the top of the stack
Very simple operation, can be done in O(1)
stack.push(88);
88
56
12
018 Stack introduction
0:01:34
0:02:27
Activate Windows
Go to Settings to activate Windows.
udemy
Type here to search
ENG
IN
8:55 AM
4/17/2021


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in O(1)
stack.pop();
88
56
12
018 Stack introduction
0:02:07
0:01:54
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
8:56 AM
4/17/2021


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in O(1)
stack.pop();
56
12
018 Stack introduction
0:02:23
0:01:38
Activate Windows
Type here to search
ENG
IN
8:57 AM
4/17/2021


Peek operation: return the item from the top of the stack without removing it
Very simple operation, can be done in O(1)
stack.peek();
The peek() method will return 88
but the structure of the stack remains
the same !!!
88
56
12
0:03:11
0:00:50
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
8:59 AM
4/17/2021

Applications

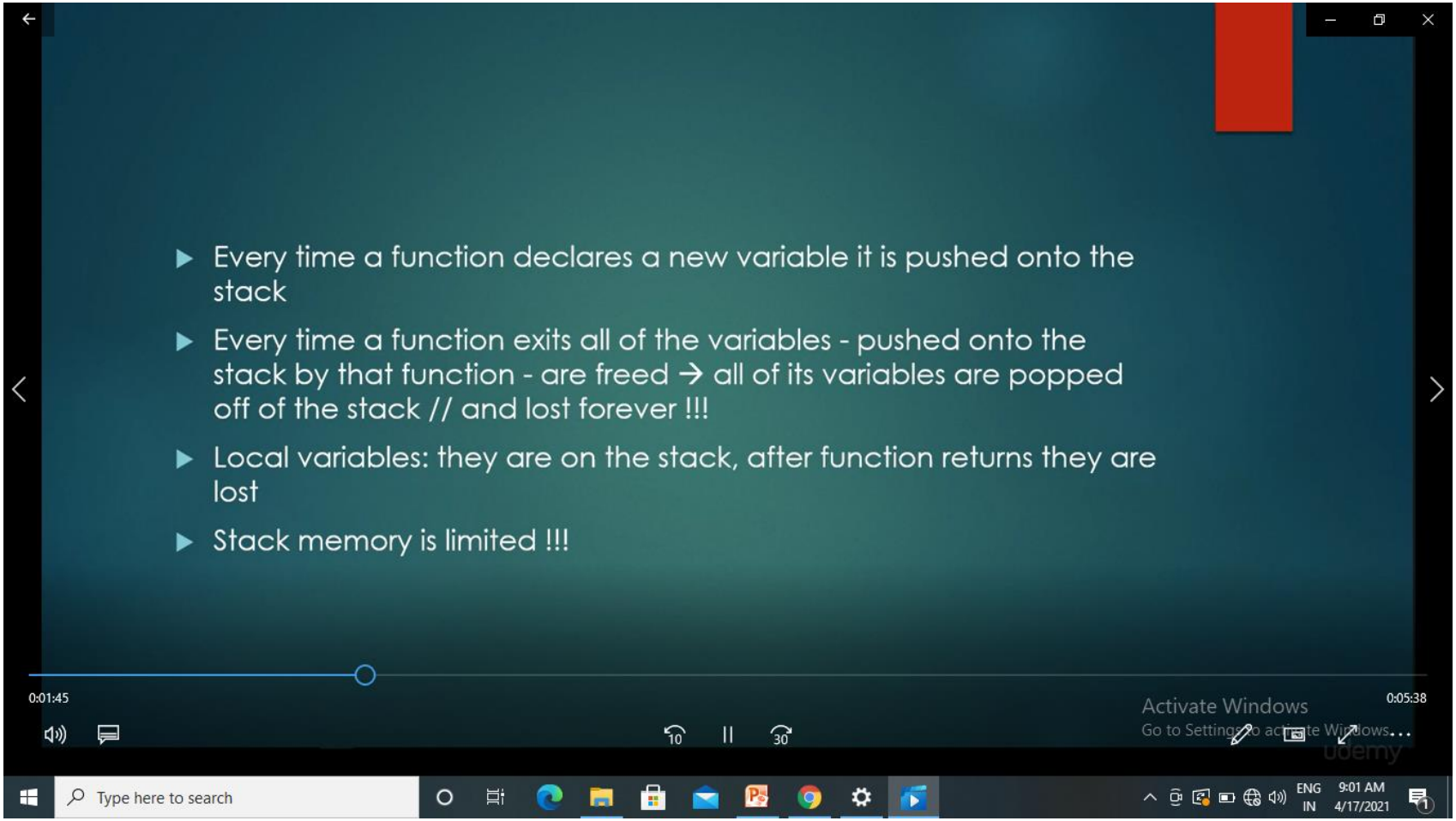
- Reverse a String → abcd=dcba
- Undo Operations in Text Editor
- Recursion / Function Call
- Balance of Parenthesis
- Infix to Prefix or Postfix Conversion
- Topological Sorting
- DFS
- Tree Traversal
- Evaluating Postfix Expression

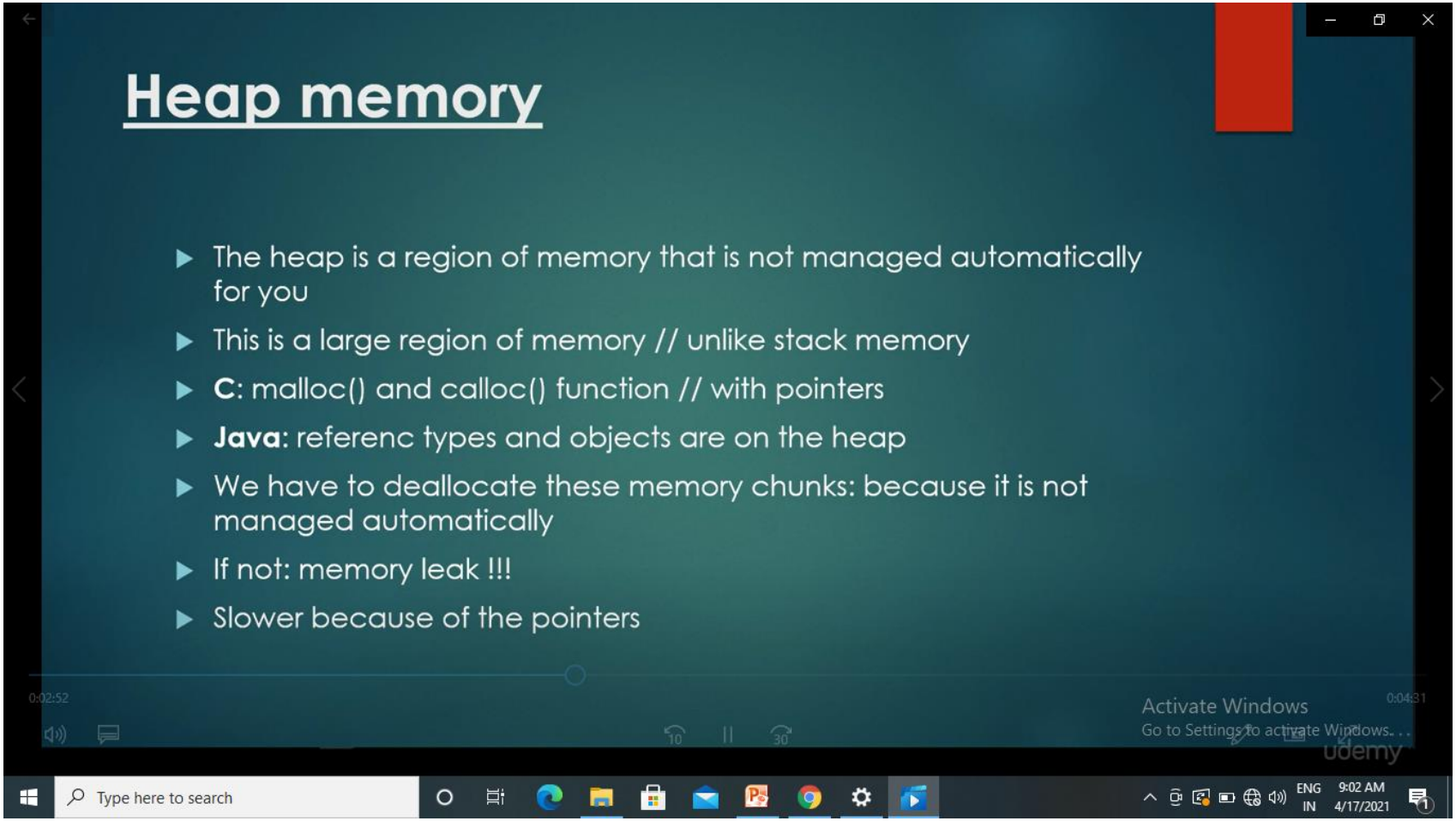
Applications

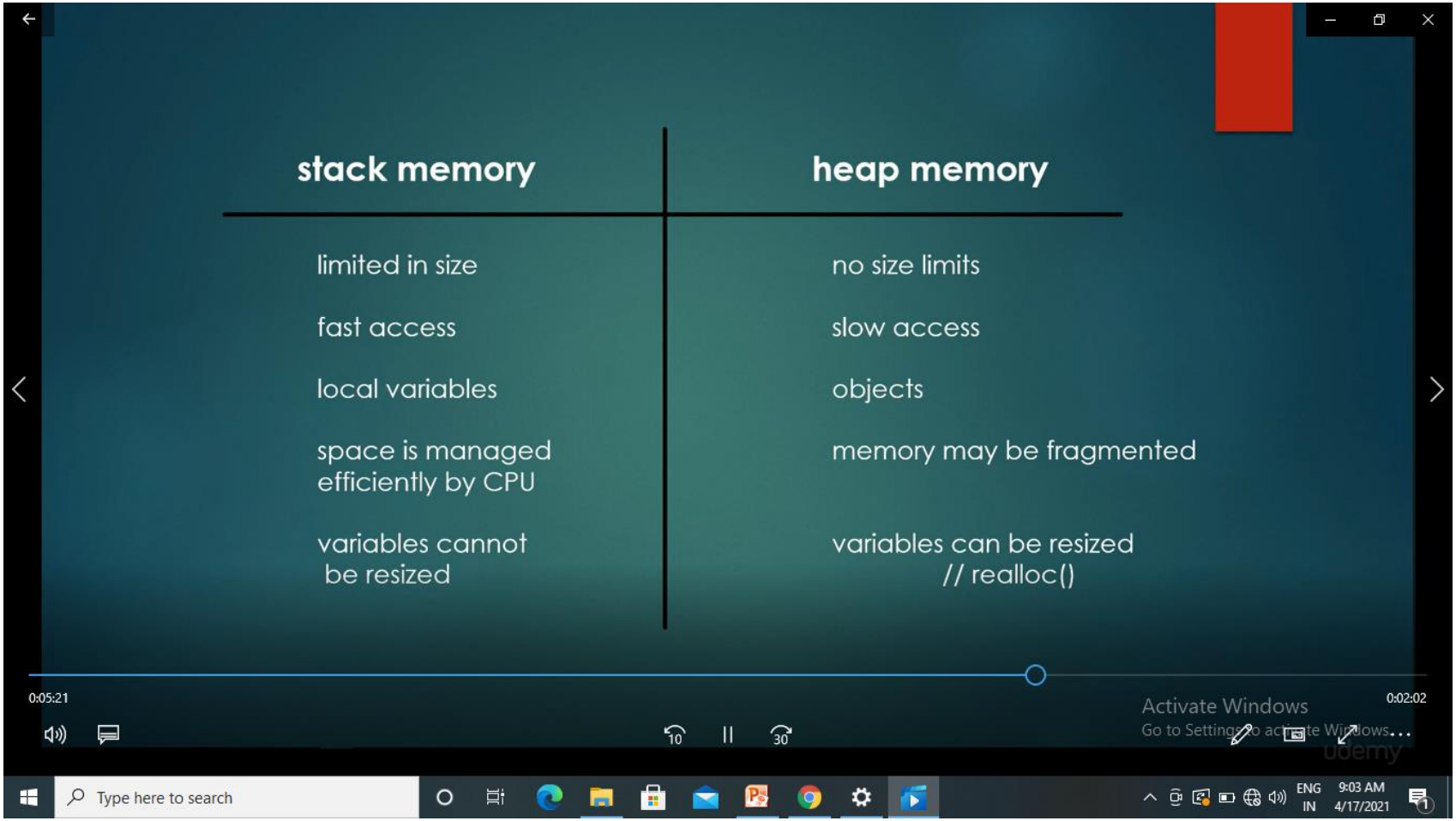
Stacks are used in backtracking algorithms
Stacks are also used in syntax parsing for many compilers
It is used for forward and backward features in web browsers
It is used in Algorithms like Tower of Hanoi, tree traversals, histogram problem and also in graph algorithms like Topological Sorting.
.

Stack „call stack"

- Most important application of stacks: stack memory
- It is a special region of the memory (in the RAM)
- A call stack is an abstract data type that stores information about the active subroutines / methods / functions of a computer program
- The details are normally hidden and automatic in high-level programming languages
- Why is it good?
- It keeps track of the point to which each active subroutine should return control when it finishes executing
- Stores temporary variables created by each function


Every time a function declares a new variable it is pushed onto the stack
Every time a function exits all of the variables - pushed onto the stack by that function - are freed → all of its variables are popped off of the stack // and lost forever !!!
Local variables: they are on the stack, after function returns they are lost
Stack memory is limited !!!
0:01:45
0:05:38
Activate Windows
Go to Settings to activate Windows...
Type here to search
ENG
IN
9:01 AM
4/17/2021


Heap memory
The heap is a region of memory that is not managed automatically for you
This is a large region of memory // unlike stack memory
C: malloc() and calloc() function // with pointers
Java: referenc types and objects are on the heap
We have to deallocate these memory chunks: because it is not managed automatically
If not: memory leak !!!
Slower because of the pointers
0:02:52
0:04:31
Activate Windows
Go to Settings to activate Windows. . .
udemy
Type here to search
ENG
IN
9:02 AM
4/17/2021


stack memory
heap memory
limited in size
fast access
local variables
space is managed efficiently by CPU
variables cannot be resized
no size limits
slow access
objects
memory may be fragmented
variables can be resized
// realloc()
0:05:21
0:02:02
Activate Windows
Go to Settings to activate Windows...
udemy
Type here to search
ENG
IN
9:03 AM
4/17/2021

Steps-push -ArraySatck

- **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.

Algorithm for push operation

A simple algorithm for Push operation can be derived as

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Example

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Steps-pop

- 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.

Algorithm-pop

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Peek() algorithm

- begin procedure peek
- return stack[top]
- end procedure

Algorithm full()

Algorithm of isfull() function –

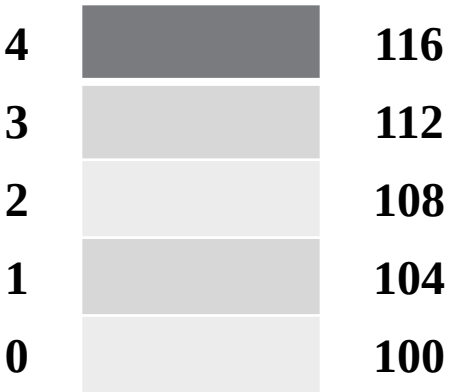
```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Algorithm of isempty() function –

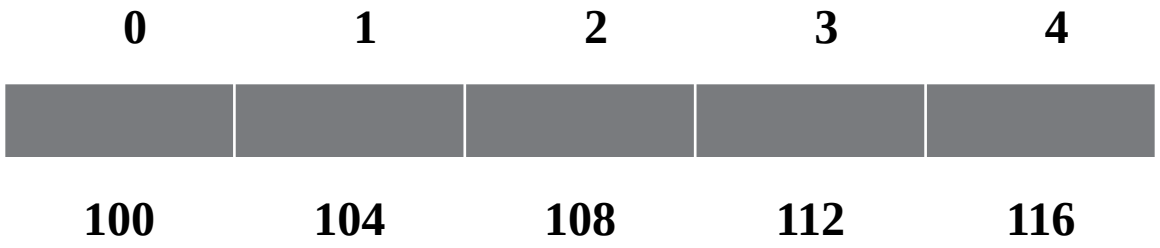
```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Code for Stack- Static Memory Allocation

- Array → int a[5];
- Stack → int stack[5];



5*4=20 Bytes




```
void main ()
{
push();          // Insertion
pop();           // Deletion
peek();          // Top
display();       // Traverse
}
```

Code for Stack- Push()

```
# define N 5;
int stack[N];
int top=-1;

void push()
{
int x ;
     printf("Enter the Element:");
     scanf("%d", &x);
if(top==N-1)
{
     print("Overflow");
}
else
{
     top++;
     stack[top]=x;
}
```

Code for Stack- Pop()

```
void pop()
{
int item;
if (top==-1)
{
    printf("Underflow");
}
else
{
    item=stack[top];
    top--;
    printf("%d", item);
}
```

Code for Stack- peek()

```
void peek()
{
if(top==-1)
{
    printf(“Stack is Empty”);
}
else
{
    printf(“%d”, stack[top]);
}
```

Code for Stack- display()

```
void display()
{
int i;
    for(i=top;i>=0;i--)
          {
                printf(“%d”,stack[i]);
          }
}
```

Alternative Way

```
# define N 5;
int stack[N];
int top=-1;
void main()
{
int ch;
clrscr();
do
{
      printf("Enter Choice: 1. Push 2. Pop 3.Peek 4. Display");
      scanf("%d",&ch);
            switch(ch)
               {
                     case 1: push();
                           break;
                     case 2: pop();
                           break;
                     case 3: peek();
                           break;
                     case 4: display();
                           break;
                     default : printf("Invalid Choice");
               }
}
while(ch!==0);
getch();  }
```

Implementing a Python =stack

- list
- collections.deque
- queue.LifoQueue
- LINKED LIST

LIST(ARRAY IMPLEMNTATION IN PYTHON)

- PUSH----------→ append()
- POP----------→pop()
- >>> stack = []
- >>> stack.append(12)
- >>> stack.append(23)
- >>> stack.append(12)
- >>> stack
- [12, 23, 12]

```
>>> stack.pop()
12
>>> stack
[12, 23]
>>> stack.pop()
23
>>> stack.pop()
12
>>> stack.pop()
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    stack.pop()
IndexError: pop from empty list
```

Array_stack –python

```
class ArrayStack:

    items = [];

    def __init__(self):
        self.items = []

    def  push(self, element):
        self.items.append(element)
```

```
def  pop(self):
    if len(self.items) == 0:
        print('Can not pop form an empty list')
    self.items.pop();
    print('Last item has been poped from the list')

def  printStack(self):
    for i in self.items:
        print(i)
```

```
def isEmpty(self):
        print(len(self.items) == 0)

    def peek(self):
        if len(self.items) == 0:
            print('Stack is Empty')
        print(self.items[len(self.items) - 1]) ;
s = ArrayStack()
s.push(8)
s.printStack()
s.push(23)
```

- s.push(45)
  s.printStack()
  s.isEmpty()
  s.pop()
  s.printStack()
  s.pop()
  s.peek()
  s.printStack()

module name is collection
collection module has deque class
dequestands for double ended queue

- collection module
- >>> import collections
- >>> stack = collections.deque
- >>> stack
- <class 'collections.deque'>
- >>> stack = collections.deque()
- >>> stack
- deque([])
- >>> stack.append(90)
- >>> stack.append(89)
- >>> stack
- deque([90, 89])

```
>>> stack.pop()
89
>>> not stack
False
>>> stack.pop()
90
>>> not stack()
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    not stack()
TypeError: 'collections.deque' object is not callable
>>> stack[-1]
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    stack[-1]
IndexError: deque index out of range
```

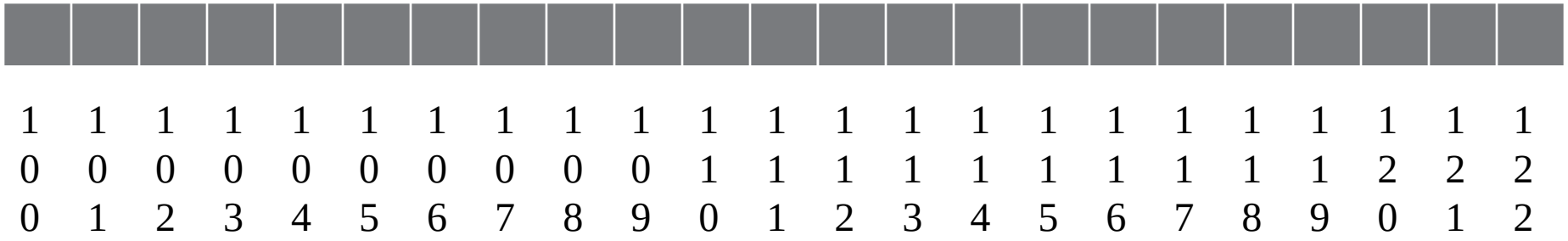
Queue module lifoqueue Put(), get() method

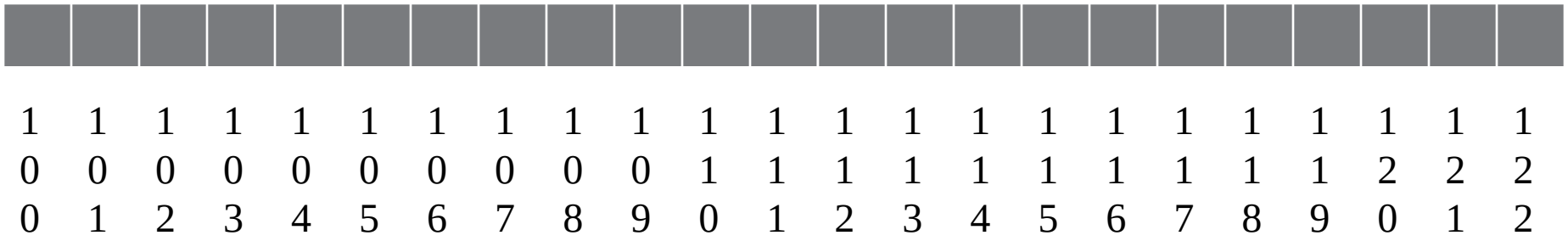
- >>> import collections
- >>> import queue
- >>> stack = queue.LifoQueue()
- >>> stack.put(10)
- >>> stack.put(20)
- >>> stack.get()
- 20
- >>> import queue
- >>> stack = queue.LifoQueue(3)

```
>>> stack.put(10)
>>> stack.put(20)
>>> stack.put(23)
>>> stack.put(23)
>>> import queue
>>> stack = queue.LifoQueue(2)
>>> stack.put(10)
>>> stack.put(20)
>>> stack.put(23,timeout=1)
```

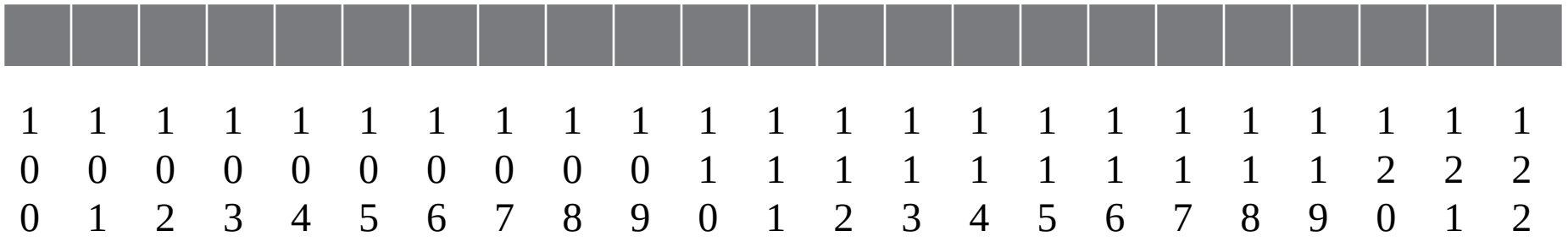
- >>> stack.get()
- 20
- >>> stack.get()
- 10
- >>> stack.get(timeout=1)
- Traceback (most recent call last):
- File "<pyshell#8>", line 1, in <module>
- stack.get(timeout=1)
- _queue.Empty

Linked List


1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2


1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2

Linked List



50*4=200 bytes

10*4=40 bytes

160 bytes waste

Linked List also a collection

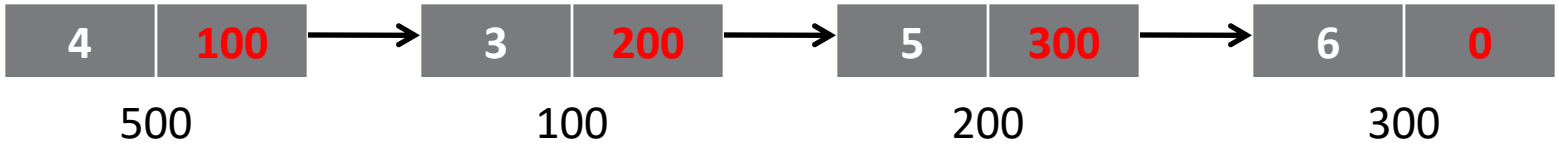
Two Problems :

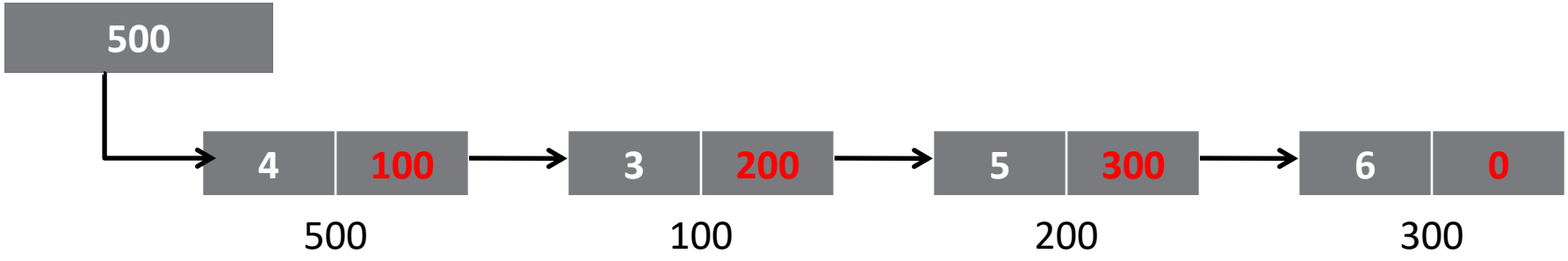
→Unused Memory

→Rearrangement of fresh block of memory unit

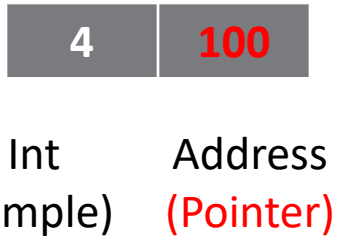
Linked List

- Single node consist of 8 bytes
  - 4 byte for int data
  - 4 byte for memory address (32 bit machine)
  - Random Access not possible
  - Only Sequential Access Possible





How to declare node in Linked List?

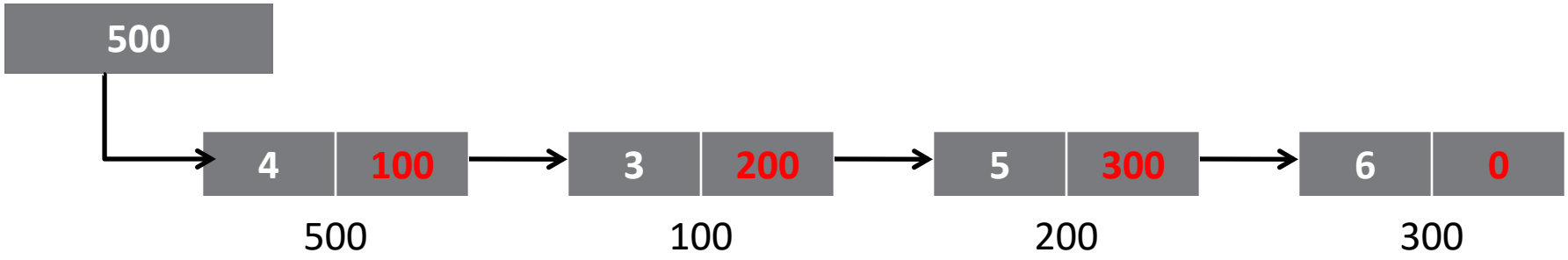


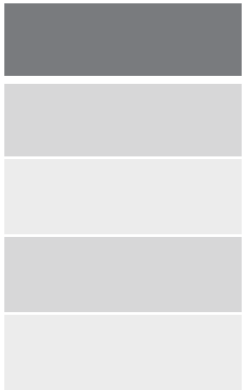

```
struct node
{
int data;
struct node * next;
};

int *a;
float *p;
```

Briefly describe about LL

- Linear Data Structure in Non Consecutive location
- Extra Space Required
- Insert/Delete Easy
- Search O(n)
- Binary Search is not Possible in linked list
- Dynamic size
- Memory Allocation at run time

Stack Implementation using Linked List





- Store the data in the stack as node
- Push() and Pop() →O(1)
- Where there is a top in LL?
  - Front? Tail?
- Tail → Insert and Delete →O(n)
- Front →Insert and Delete →O(1)

Steps push an element into linked list

- Step-1 :Assign Top=0
- Step=2:create a new node
- Step =3 :new node push into the stack
- Step-4:(Assume)-new node Address is 1000 i.e memory location
- Step 5: insert element(example-4) into the newnode data part
- Step 5:Assign the Top pointer value to the newnode link part(now first node address is 0)

- Step 6: Top pointer hold the Address of first new node memory address i.e1000
- Step 7:(second) new node is created
- Step- 8 push data into 2nd new node data part now 2nd new node Address 2000
- Step -9 top pointer Address is assign to the 2nd new node link part
- Step 10- top is assigned the Address of 2 nd new node address

Stack Operations using Linked List(To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

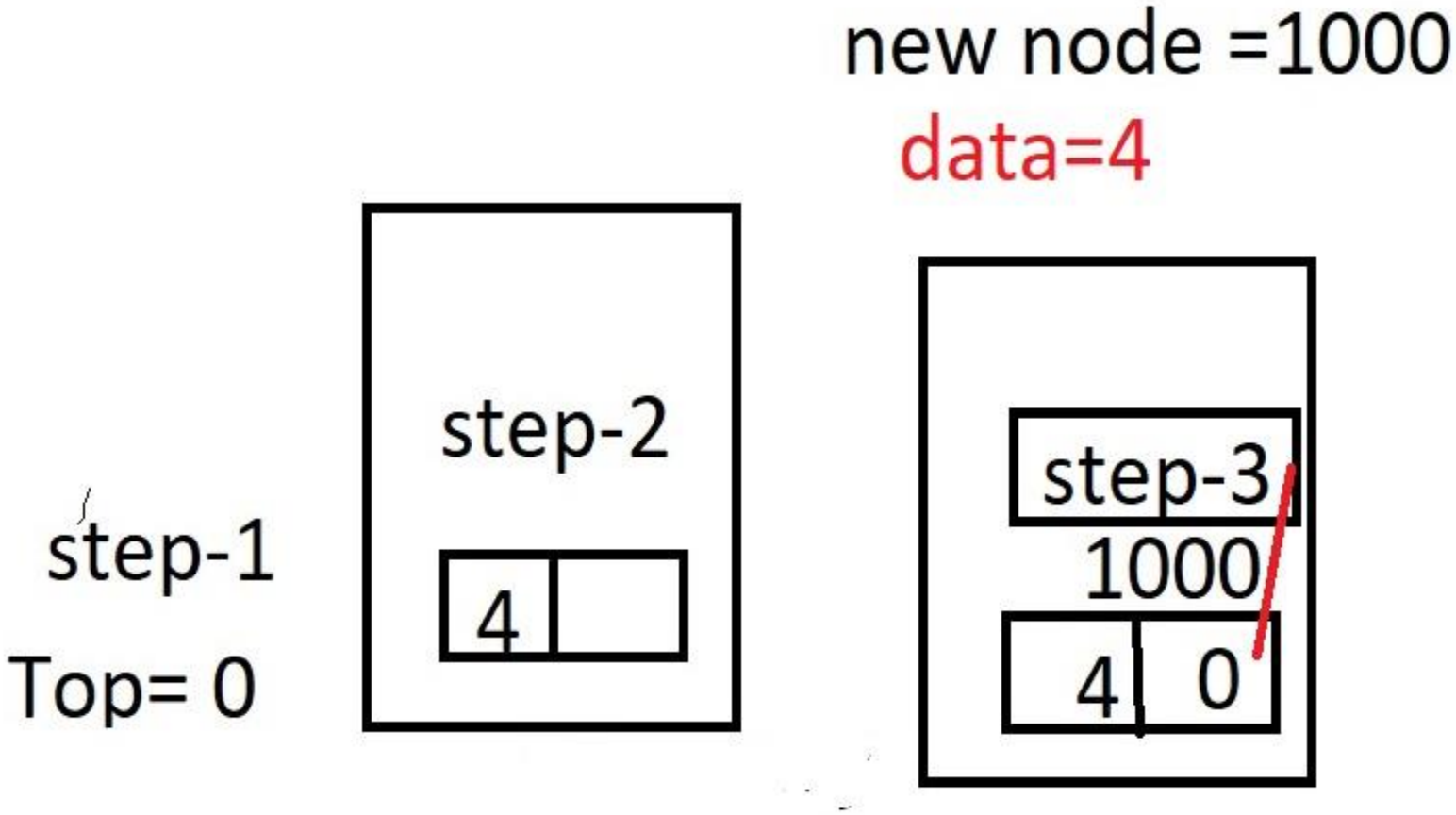
pop() - Deleting an Element from a Stack

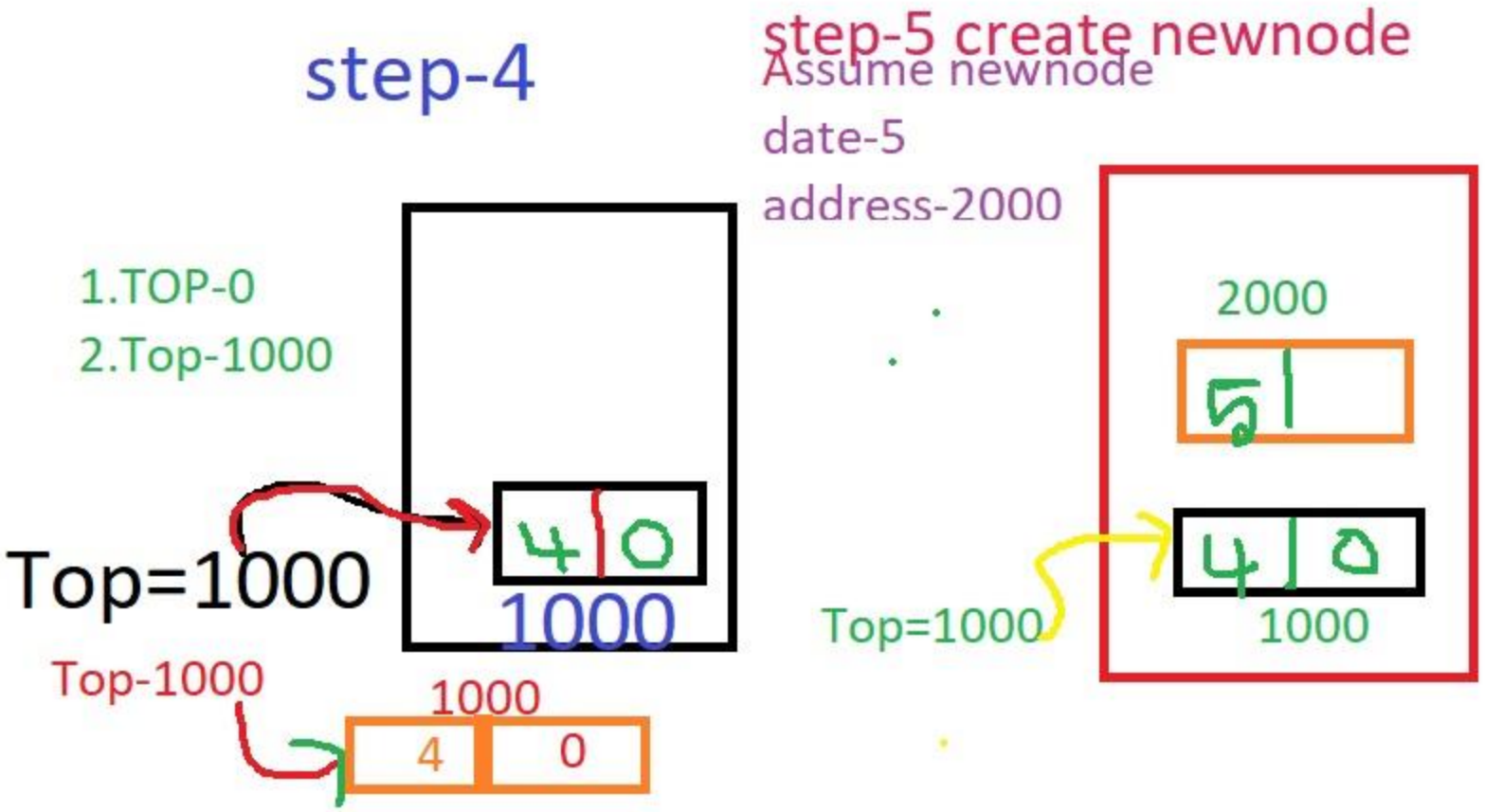
We can use the following steps to delete a node from the stack...

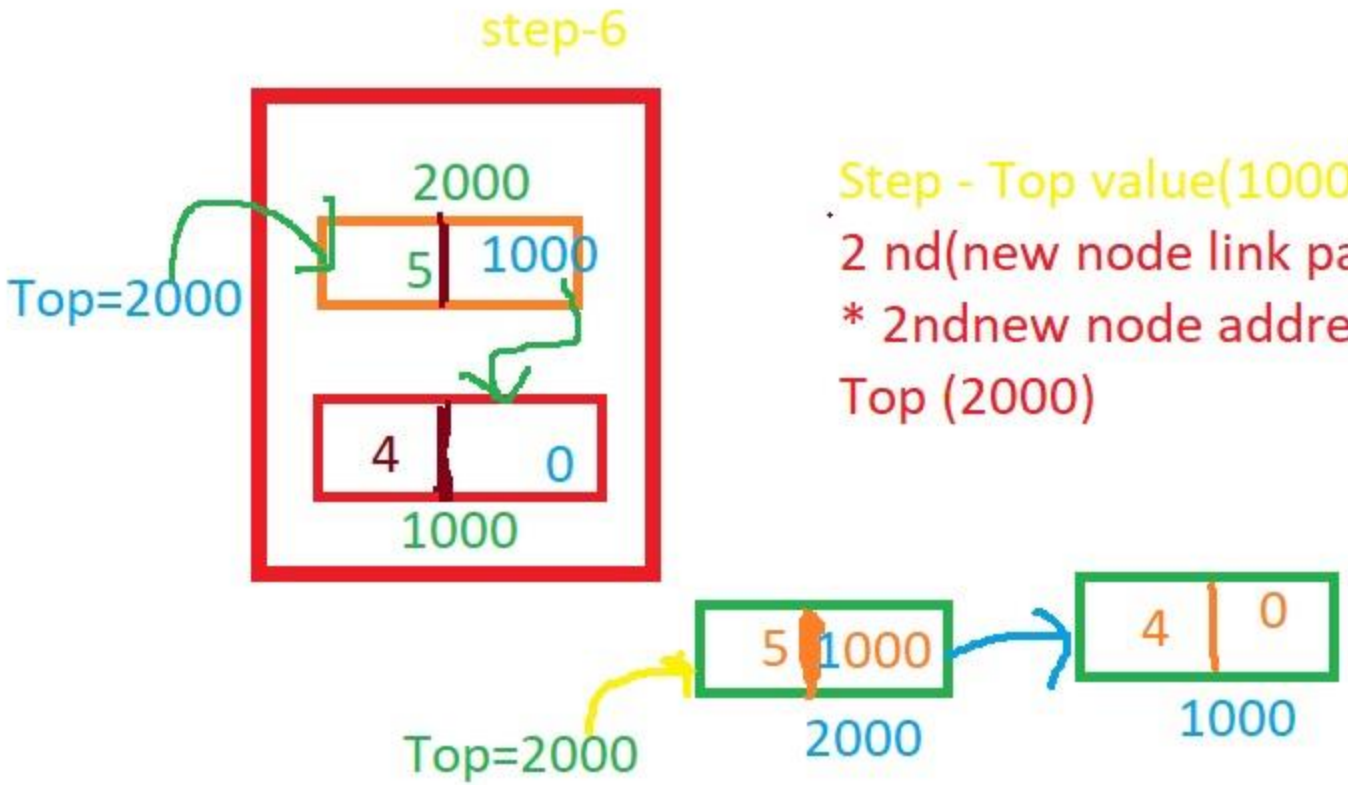
- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top** → **next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements(We can use the following steps to display the elements (nodes) of a stack...)

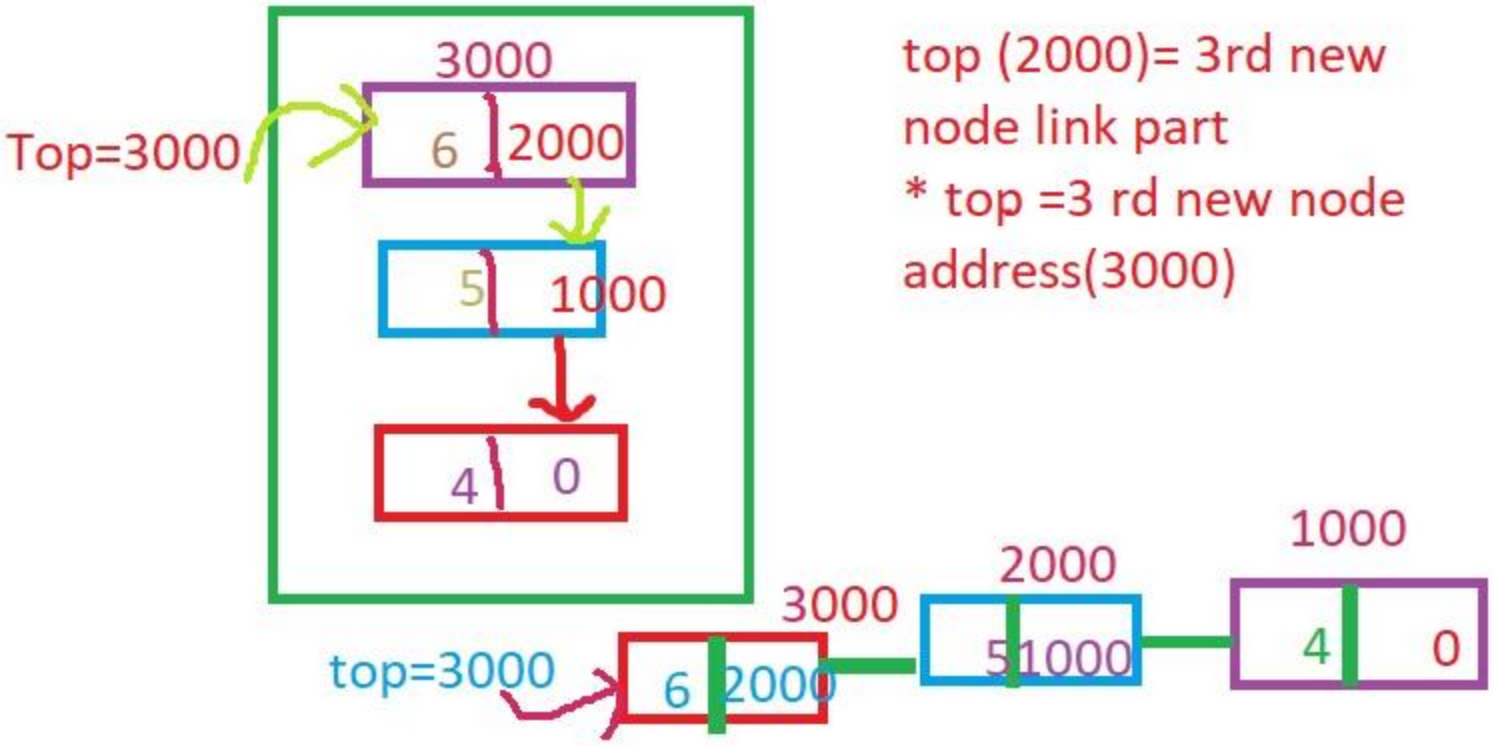
- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.


new node =1000
data=4
step-1
Top= 0
step-2
4
step-3
1000
4
0


step-4
1.TOP-0
2.Top-1000
Top=1000
4
0
1000
Top-1000
1000
4
0
step-5 create newnode
Assume newnode
date-5
address-2000
2000
5
1
4
0
Top=1000
1000

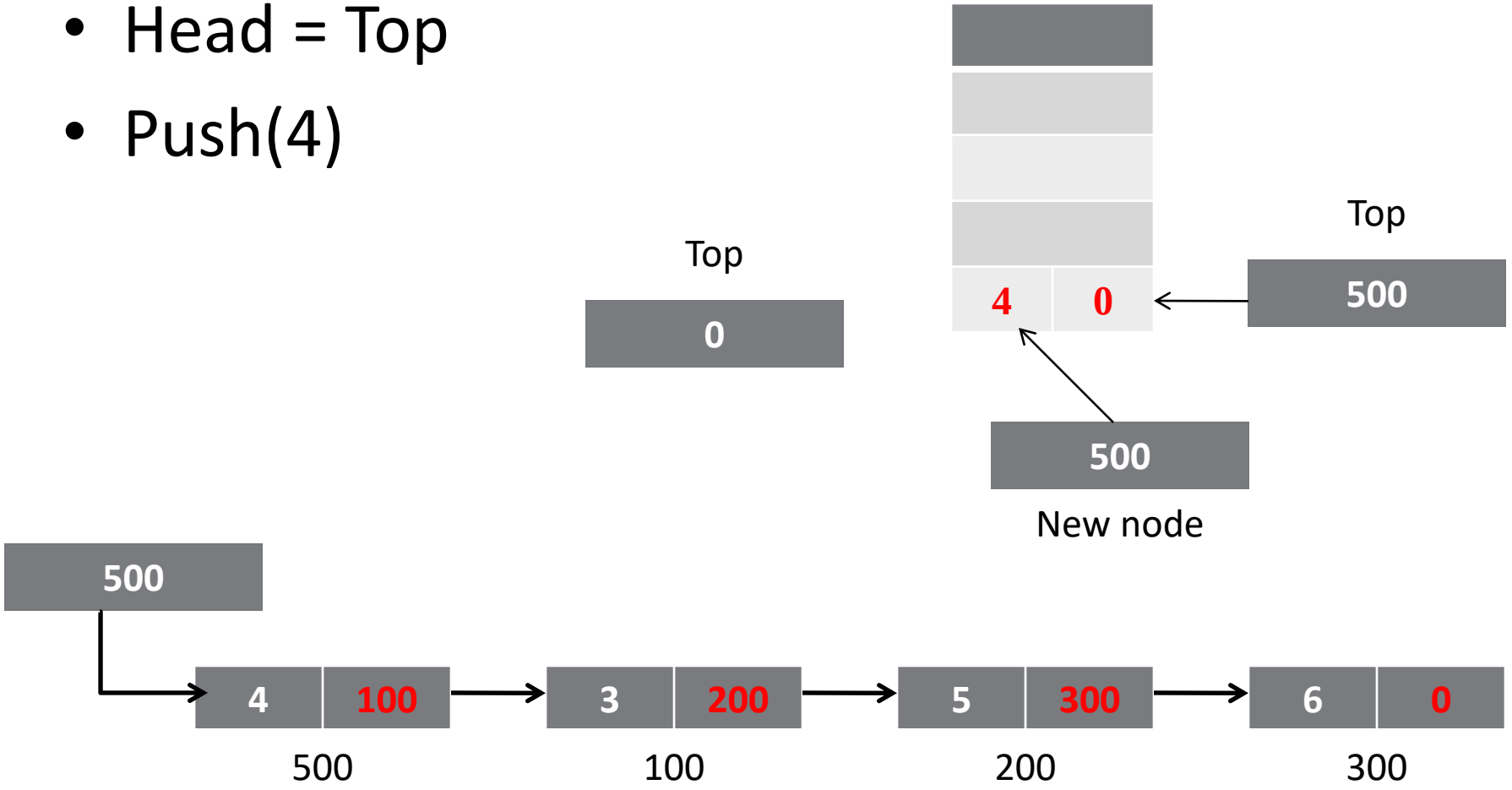

step-6
2000
5
1000
Top=2000
4
0
1000
Step - Top value(1000) =
2 nd(new node link part)
* 2ndnew node address=
Top (2000)
5
1000
2000
4
0
1000
Top=2000

step= 7 (Assume)create new node data 6 address 3000



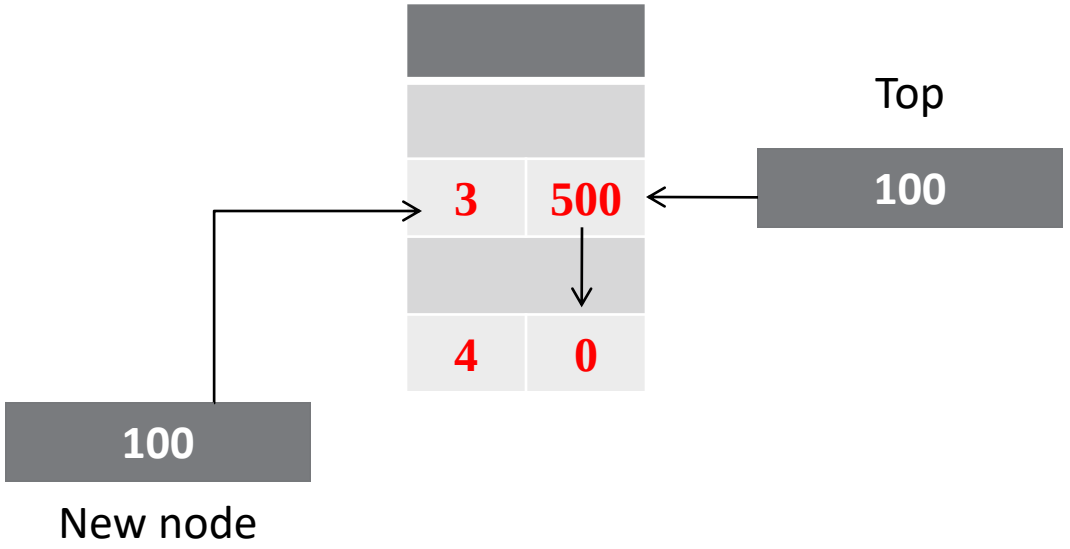
Memory Representation

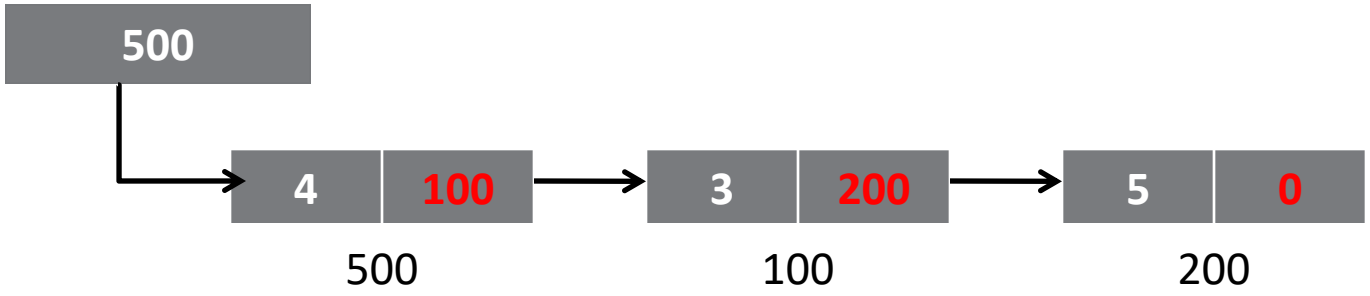
- Head = Top
- Push(4)



Memory Representation

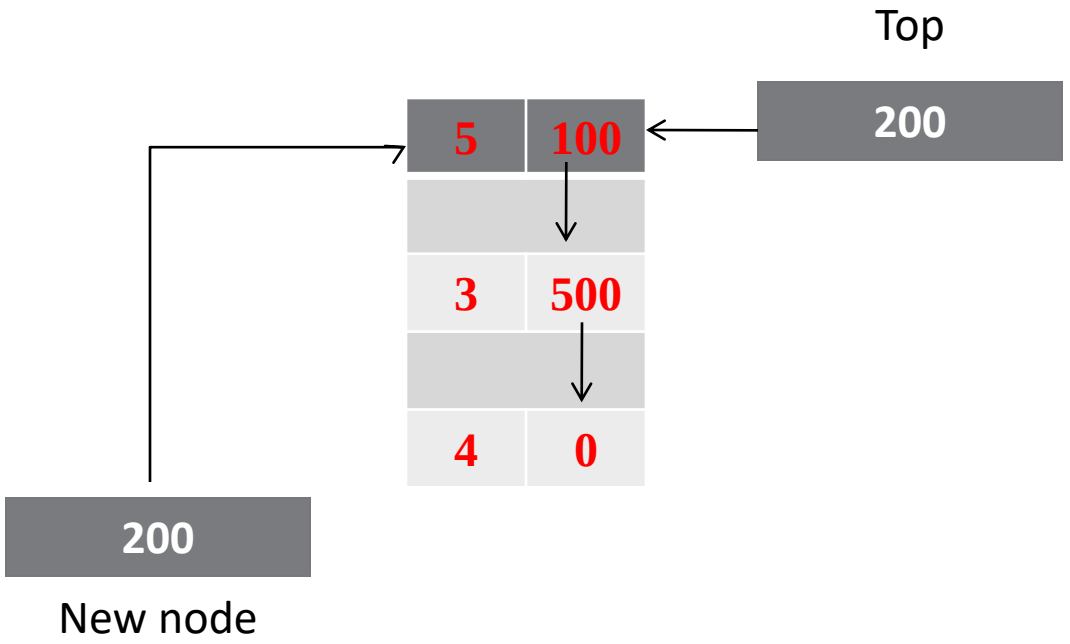
- Head = Top
- Push(3)

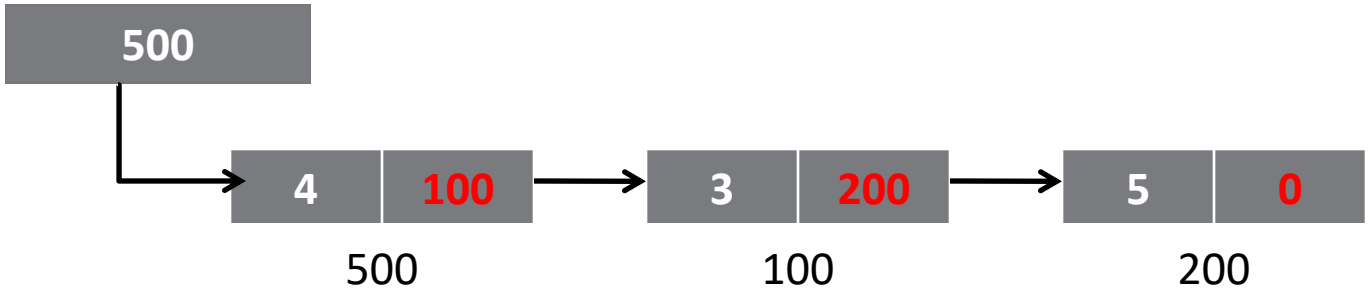




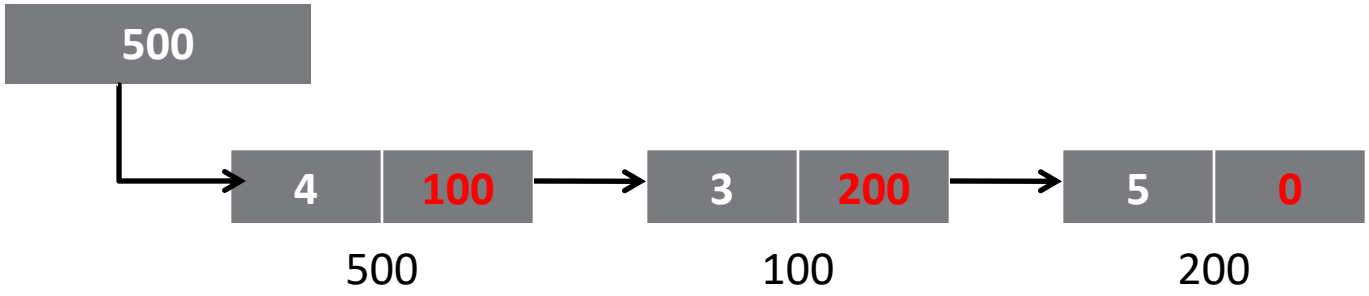
Memory Representation

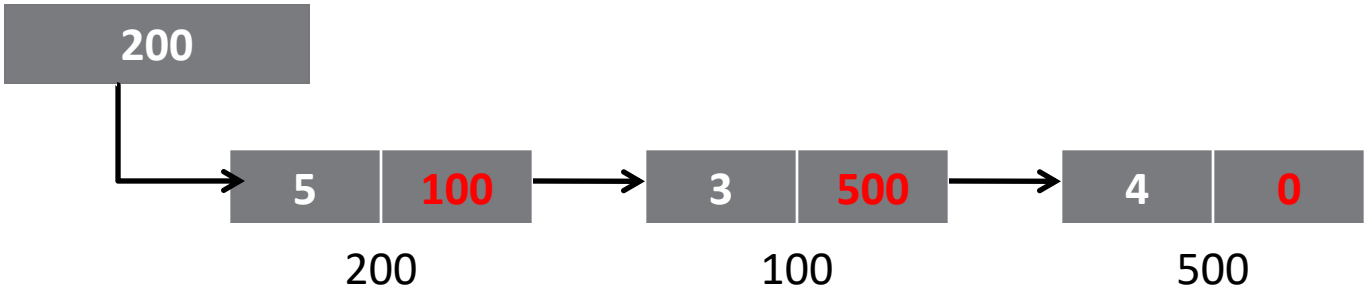
- Head = Top
- Push(5)

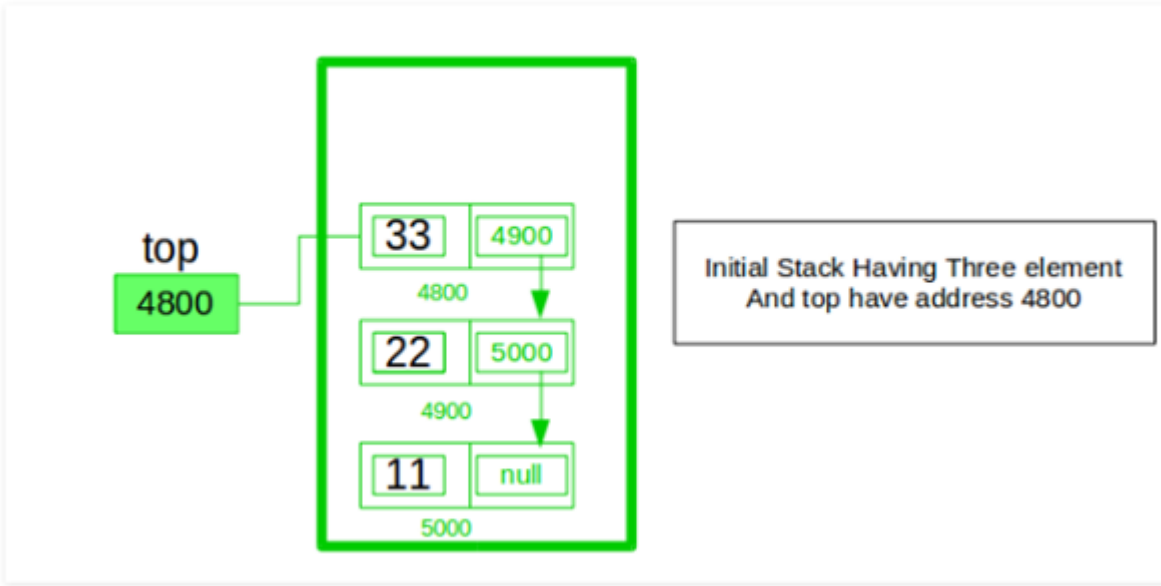


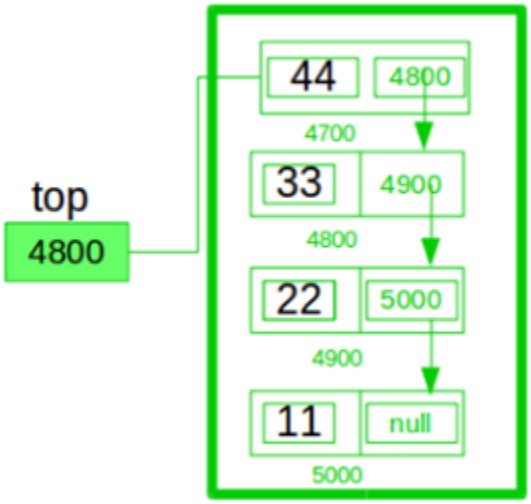


LL vs Stack in LL

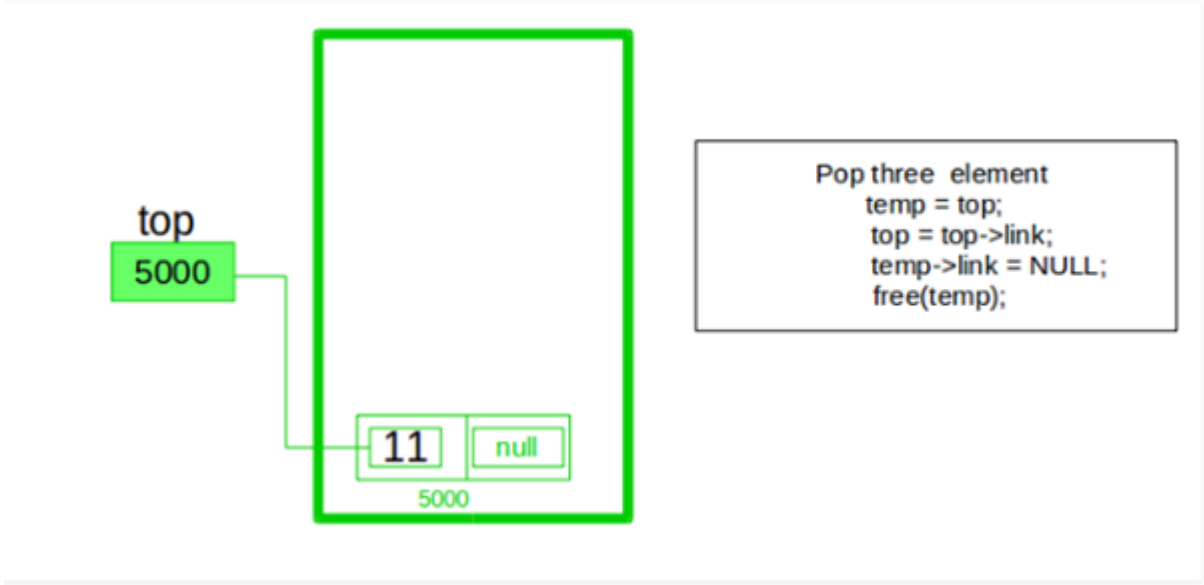





top
4800
33
4900
4800
22
5000
4900
11
null
5000
Initial Stack Having Three element
And top have address 4800



First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top

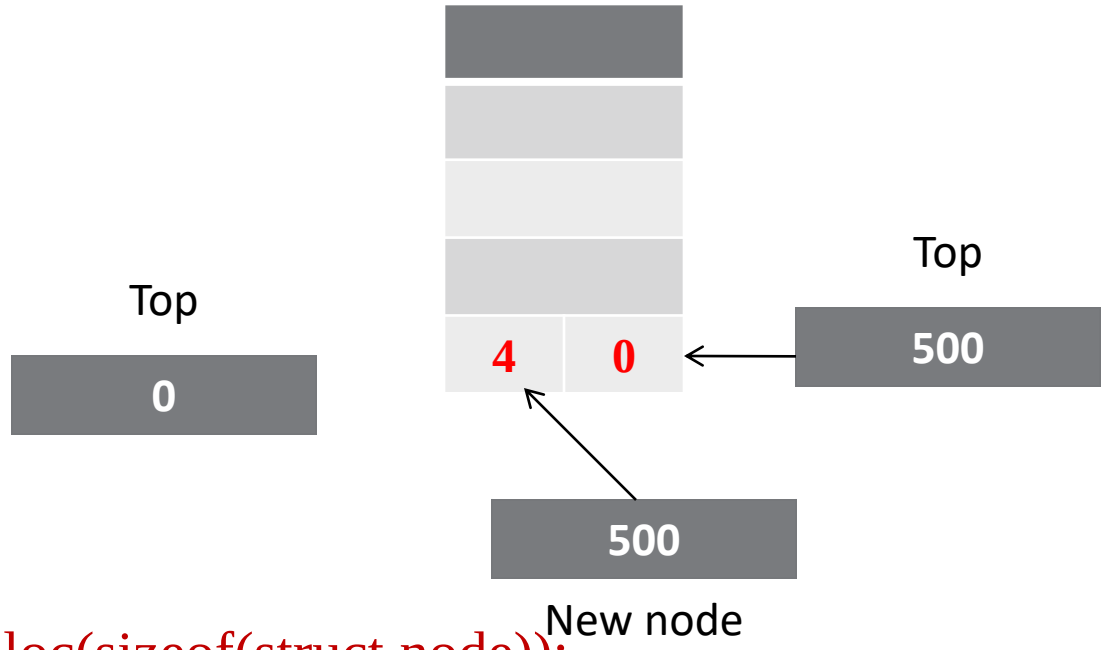

top
5000
11
null
5000
Pop three element
temp = top;
top = top->link;
temp->link = NULL;
free(temp);

Code for Stack- Dynamic Memory Allocation c-code

```
void main()
{
    push(4);
    push(3);
    push(5);
    display();
    peek();
    pop();
    peek();
    display();
}
```

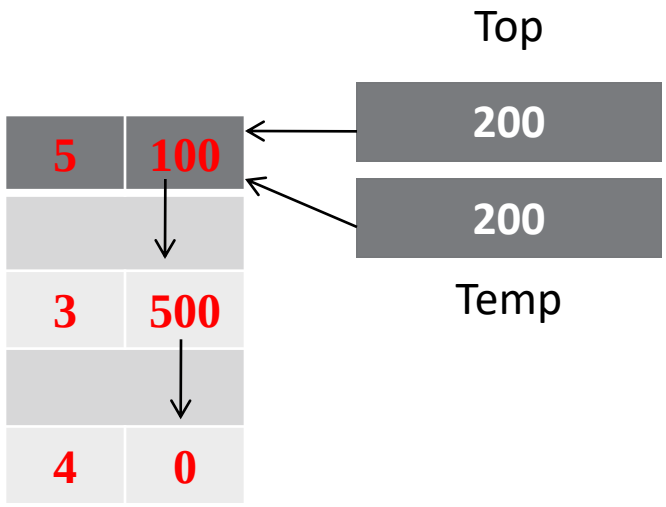
Code for Stack- Push()

```
struct node
{
int data;
struct node *link;
}
struct node *top=0;
void push(int x)
{
struct node *newnode;
newnode=(struct node*)malloc(sizeof(struct node));
newnode→data=x;
newnode→next=top;
top=newnode;
}
// No overflow condition checking needed
```



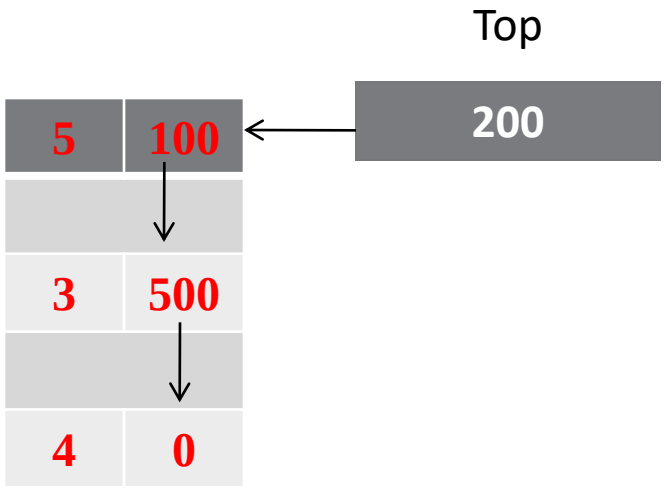
Code for Stack- display()

```
void display()
{
    struct node * temp;
    temp=top;
    if (top==0)
    {
        printf("stack is empty");
    }
    else
    {
        while(temp!=0)
            {
                printf("%d",temp→data);
                temp=temp→link;
            }
    }
}
```



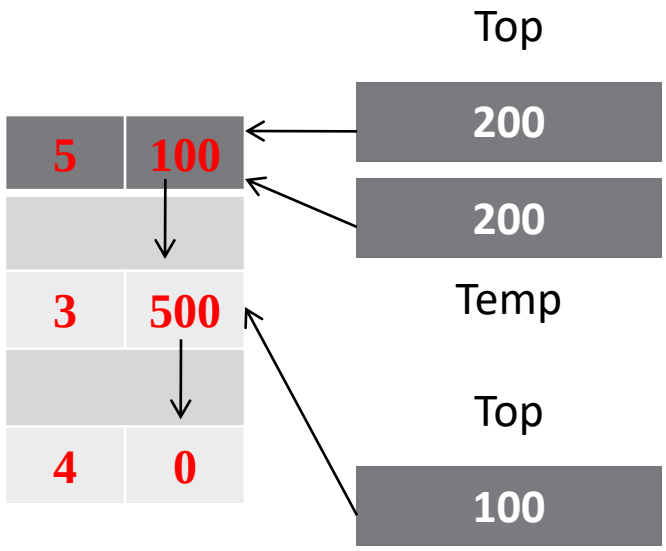
Code for Stack- peek()

```
void peek()
{
    if(top==0)
    {
        printf("stack is empty");
    }
    else
    {
        printf("top element is %d", top→data);
    }
}
```



Code for Stack- pop()

```
void pop()
{
    struct node * temp;
    temp=top;
    if (top==0)
        {
            printf("underflow");
        }
    else
        {
            printf("pop element is %d", top→data);
            top=top→link;
            free(temp);
        }
}
```



implement-a-stack-using-singly-linked-list-python code

```
class Node:
        # Class to create nodes of linked list
    # constructor initializes node automatically
    def __init__(self,data):
        self.data = data
        self.next = None
    class Stack:
        # head is default NULL
    def __init__(self):
        self.head = None
```

Checks if stack is empty

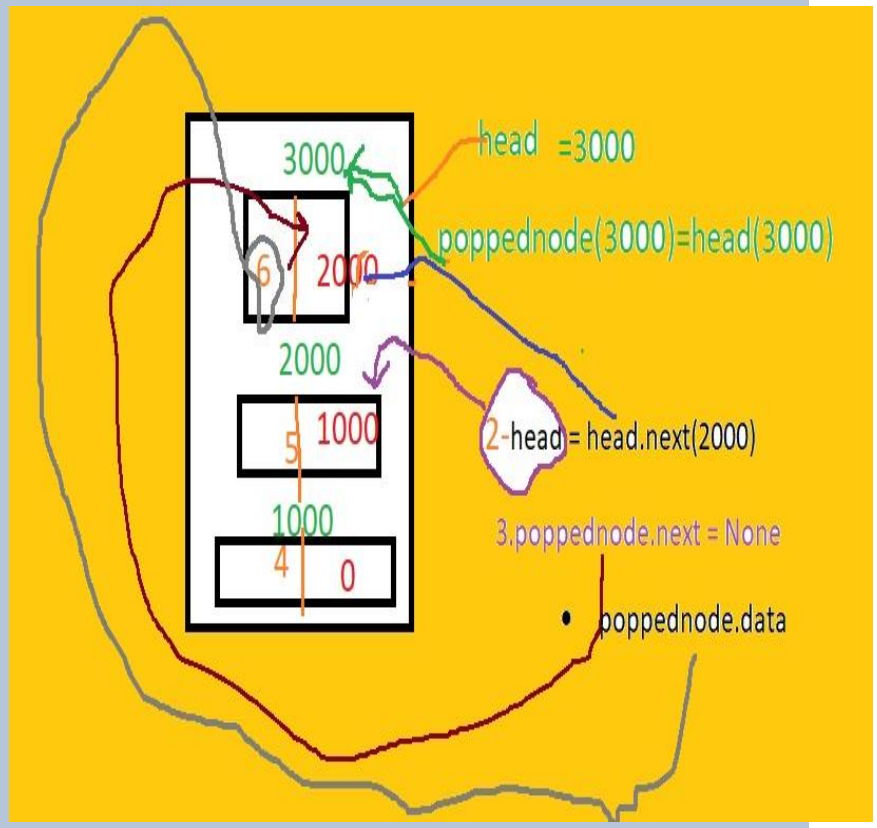
```
def isempty(self):
        if self.head == None:
            return True
        else:
            return False
```

Method to add data to the stack
adds to the start of the stack

```
def push(self,data):

        if self.head == None:
            self.head=Node(data)

        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode
```

Remove element that is the current head (start of the stack

```
def pop(self):

    if self.isempty():
        return None

    else:
    # Removes the head node and makes
    #the preceeding one the new head
        poppednode = self.head
        self.head = self.head.next
        poppednode.next = None
        return poppednode.data
```



Returns the head node data

```
def peek(self):

        if self.isempty():
            return None

        else:
            return self.head.data

```

```
# Prints out the stack
    def display(self):

        iternode = self.head
        if self.isempty():
            print("Stack Underflow")

        else:

            while(iternode != None):

                print(iternode.data,"->",end = " ")
                iternode = iternode.next
            return
```

```
# Driver code
MyStack = Stack()

MyStack.push(11)
MyStack.push(22)
MyStack.push(33)
MyStack.push(44)

# Display stack elements
MyStack.display()
 # Print top element of stack
print("\nTop element is ",MyStack.peek())
```

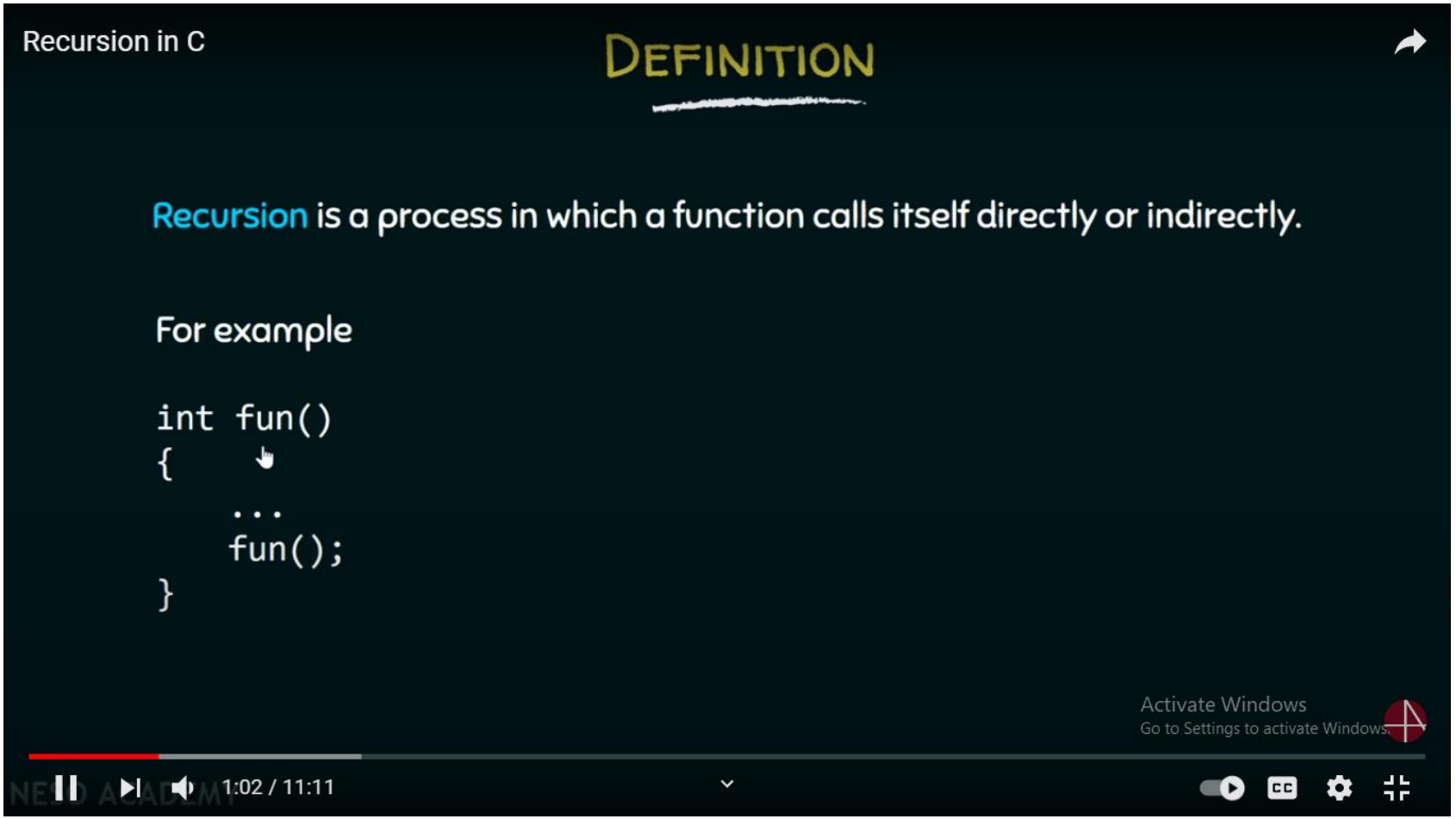
Delete top elements of stack

- MyStack.pop()
- MyStack.pop()
-
- # Display stack elements
- MyStack.display()
-
- # Print top element of stack
- print("\nTop element is ", MyStack.peek())

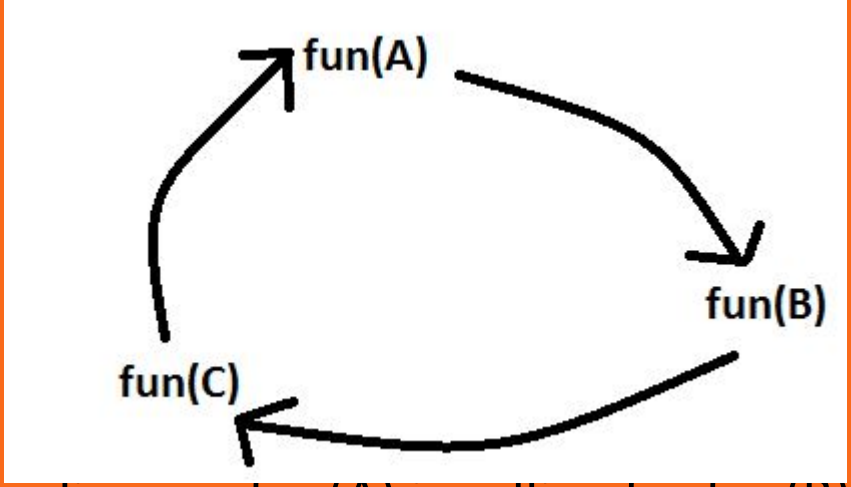
- 44->33->22->11->
- Top element is 44
- 22->11->
- Top element is 22

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
- Note: All the recursive algorithms can be transformed into asimple meyhod with stacks.
- If we use recursion ,the OS will use stack anyway!!
- Programmers define exit condition from the function otherwise it willgo into an infinite loop.


Recursion in C
DEFINITION
Recursion is a process in which a function calls itself directly or indirectly.
For example
int fun()
{
...
fun();
}
Activate Windows
Go to Settings to activate Windows.
1:02 / 11:11

- **Indirect Recursion**: In this recursion, there may be more than one functions and they are calling one another in a circular manner.



- From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.

Indirect Recursion

A function is said to be indirect recursive if it calls another function and this new function calls the first calling function again.

```
int func1(int n)
{
if (n<=1)
return 1;
else
 return func2(n);
}
int func2(int n)
 {
return func1(n);
 }
```

Program

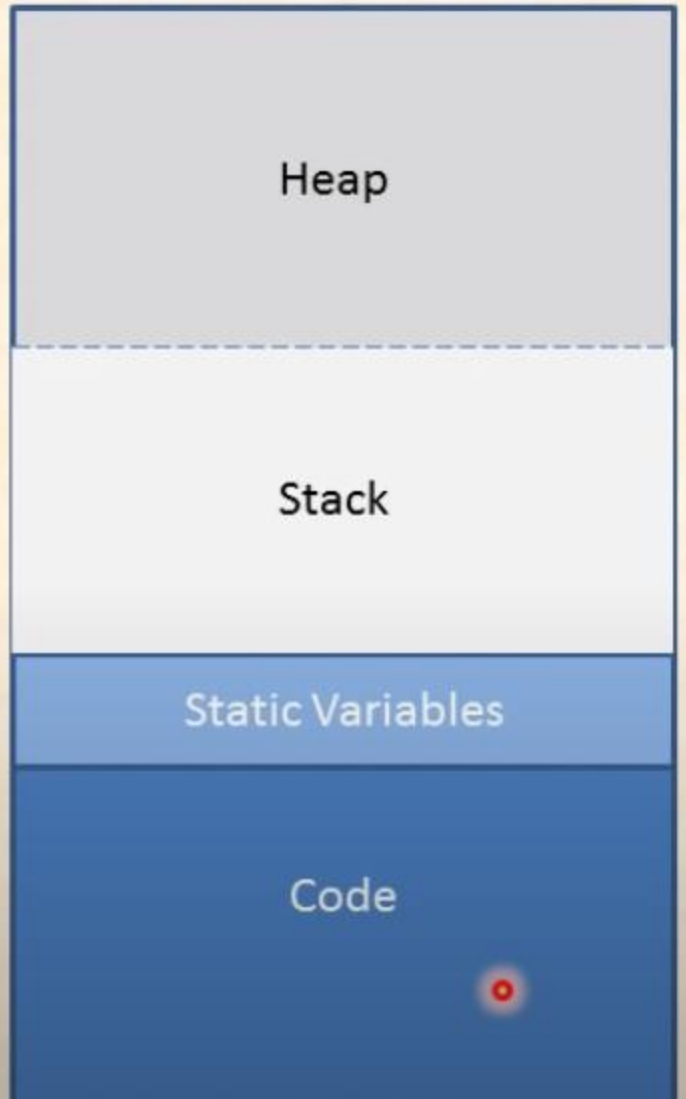
```
void main()
{
    int a,b,c;
    float x;
    char c;

    srintf("Enter 3 numbers");
    scanf("%d%d%d"@);
    ----------------
    ----------------
    }
```

Data

Instruction

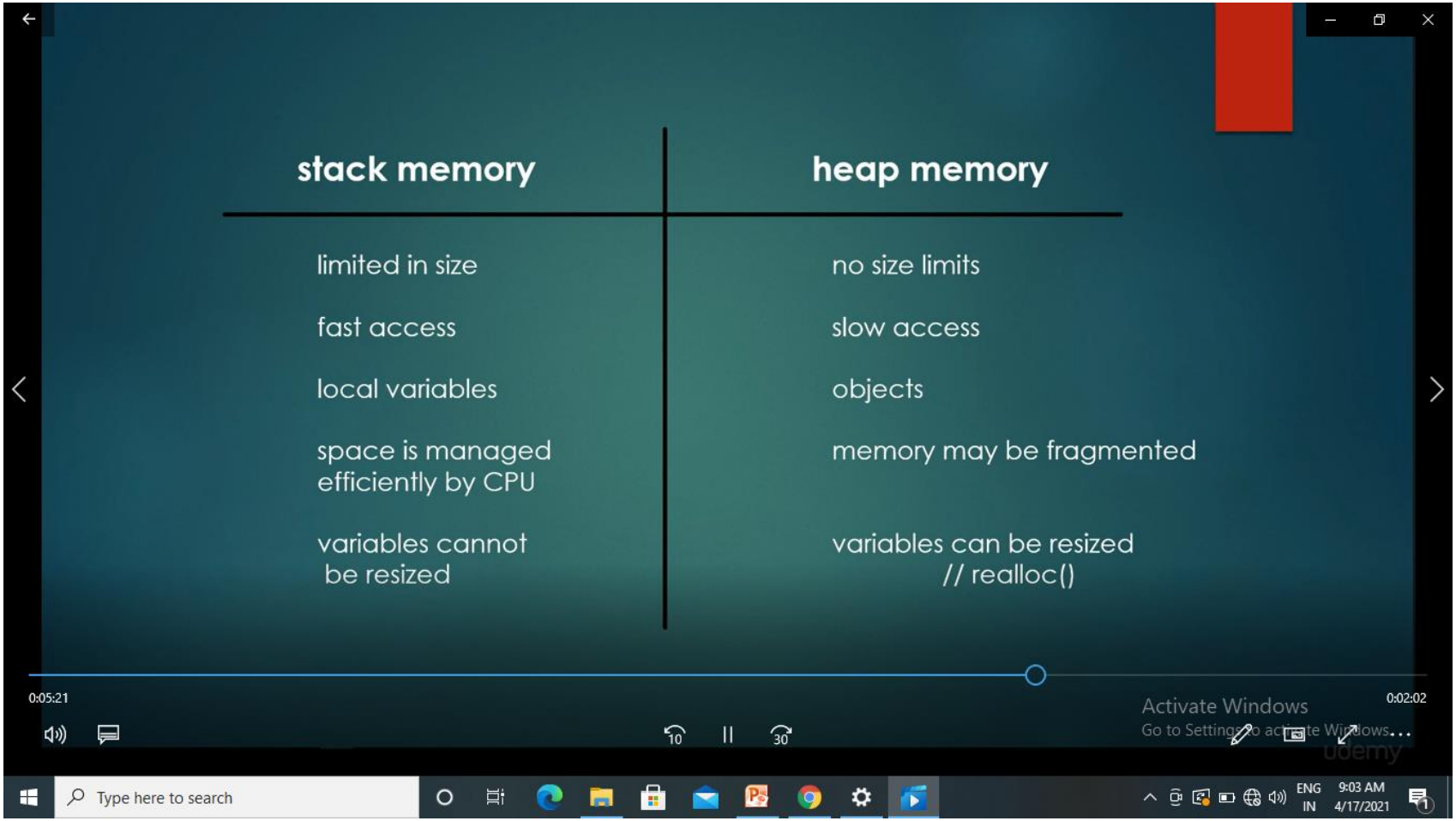
Recursion and Stack - English

Memory



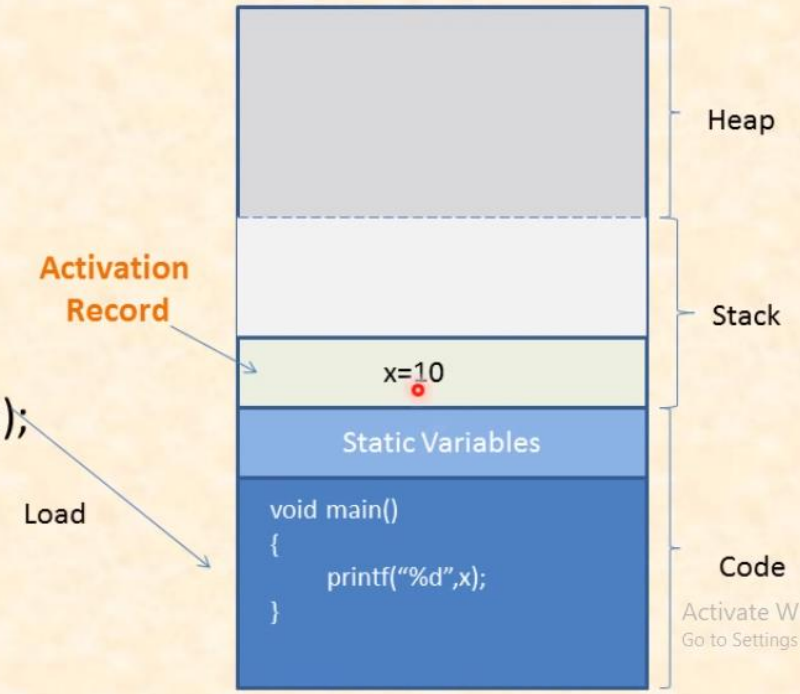
Mute (m)

- One portion of main memory(RAM) ouccupy by code section
- Remaining memory is divided into three section
- These are use for running our program
- Code secyion instruction, statements are kept in this section
- Satck varible or data of the function store here
- Heap is used for dynamic allocation of memory
- Stativ portions is belongs to code section when declarestatic global variable that timeuse forthat


stack memory
heap memory
limited in size
fast access
local variables
space is managed efficiently by CPU
variables cannot be resized
no size limits
slow access
objects
memory may be fragmented
variables can be resized
// realloc()
0:05:21
0:02:02
Activate Windows
Go to Settings to activate Windows...
udemy
Type here to search
ENG
IN
9:03 AM
4/17/2021

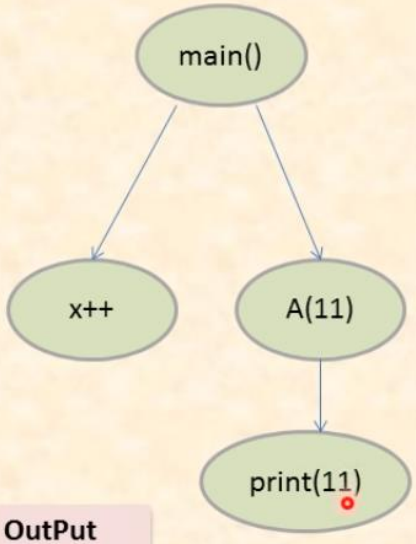
Loading of Program

```
void main()
{
        int x=10;
        printf("%d",x);
}
```



Function Call

```
void A(int a)
{
        printf("%d",a);
}

void main()
{
        int x=10;
        x++;
        A(x);
}
```

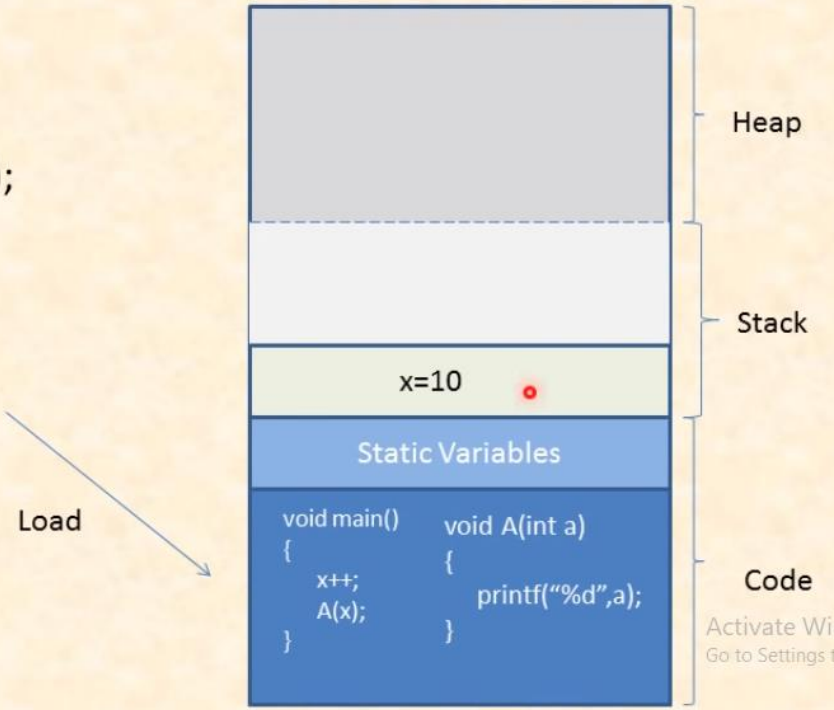


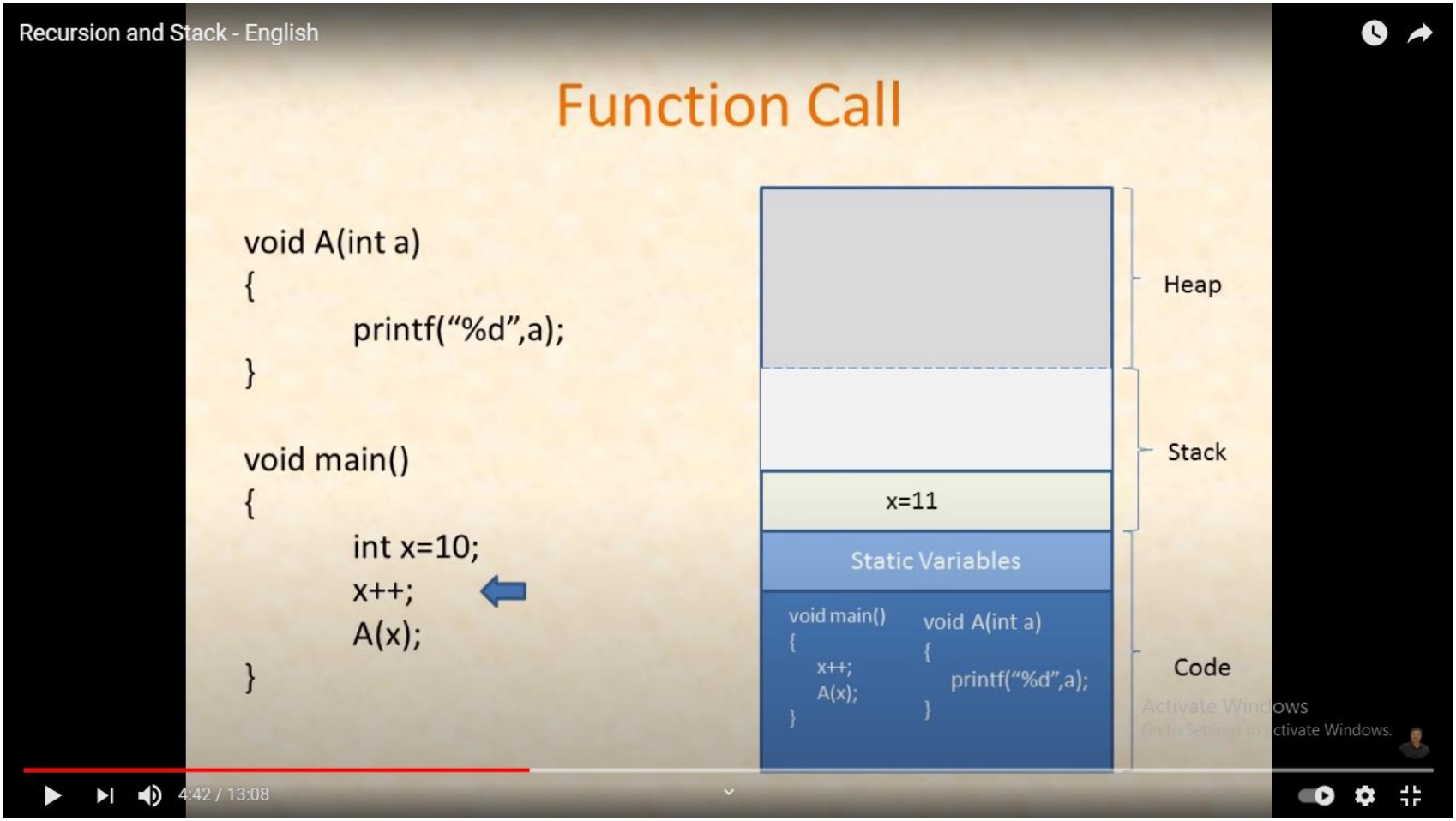
OutPut

11

Function Call

```
void A(int a)
{
        printf("%d",a);
}

void main()
{
        int x=10;
        x++;
        A(x);
}
```

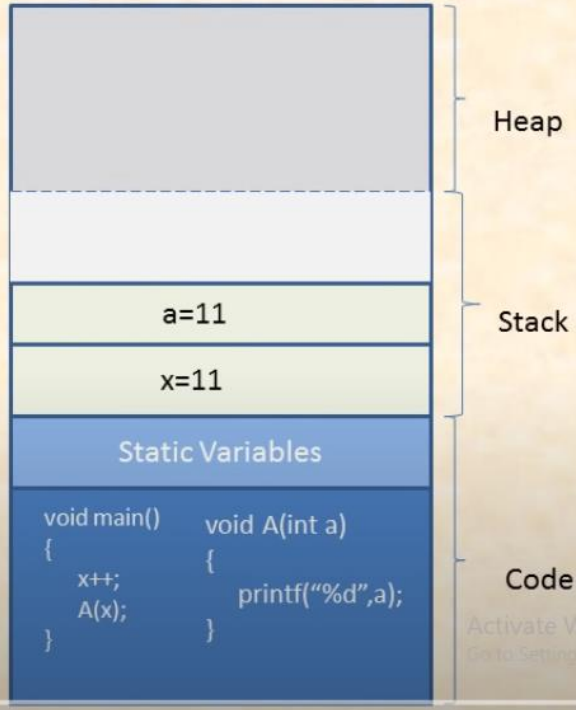



Recursion and Stack - English
Function Call
void A(int a)
{
printf("%d",a);
}
void main()
{
int x=10;
x++;
A(x);
}
Heap
Stack
x=11
Static Variables
void main()
{
x++;
A(x);
}
void A(int a)
{
printf("%d",a);
}
Code
Activate Windows
Go to Settings to activate Windows.
4:42 / 13:08

Recursion and Stack - English

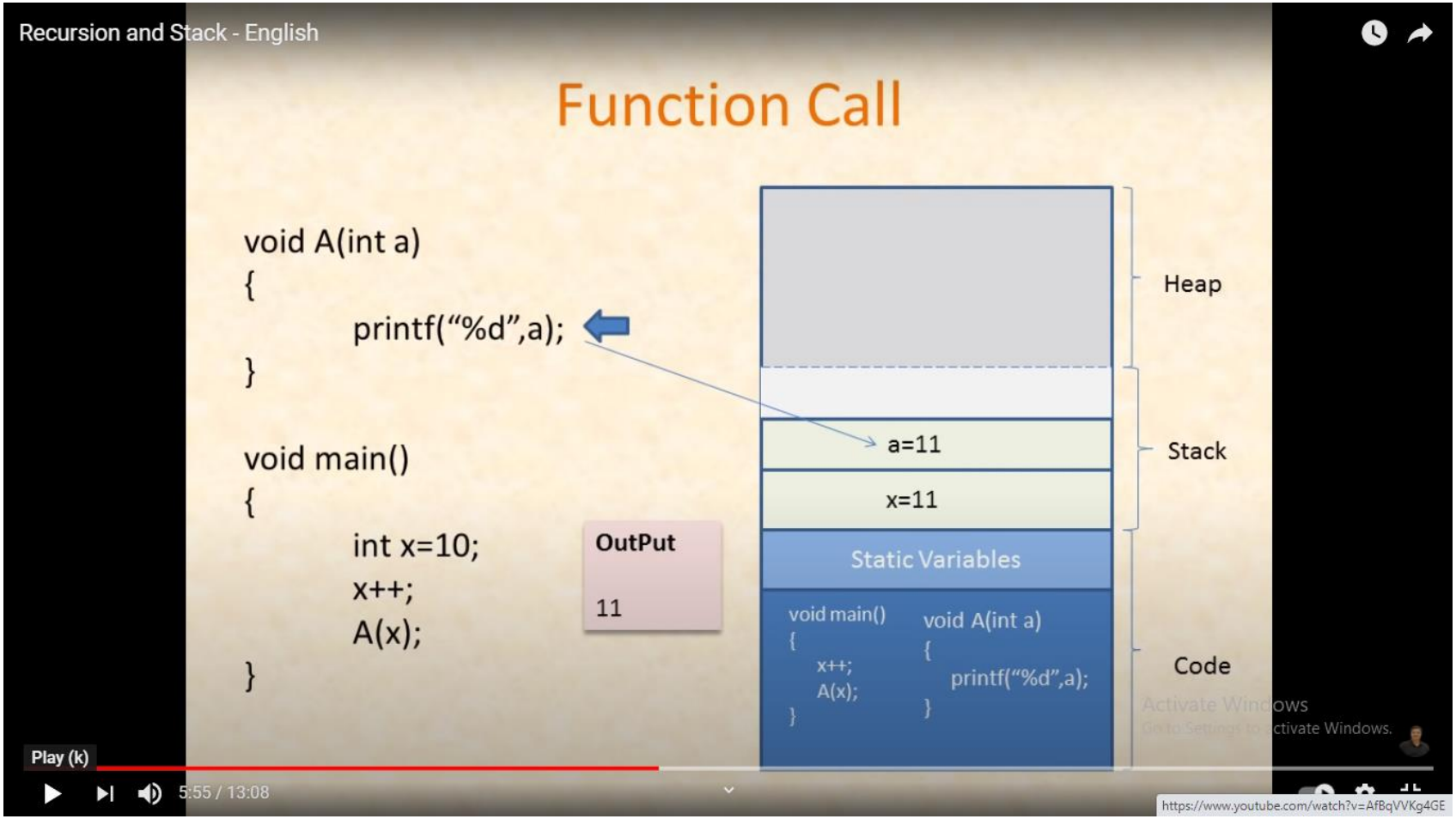
Function Call

```
void A(int a)
{
        printf("%d",a);
}

void main()
{
        int x=10;
        x++;
        A(x);
}
```



Play (k)

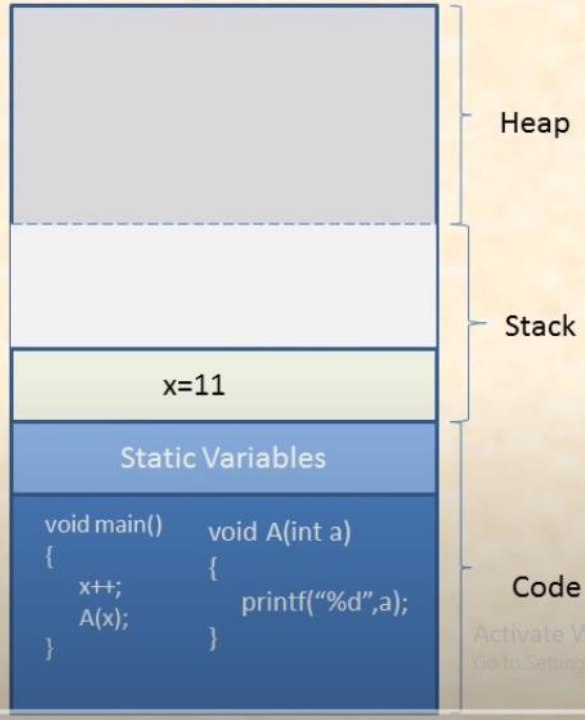
4:59 / 13:08


Recursion and Stack - English
Function Call
void A(int a)
{
printf("%d",a);
}
void main()
{
int x=10;
x++;
A(x);
}
OutPut
11
Heap
Stack
Code
a=11
x=11
Static Variables
void main()
{
x++;
A(x);
}
void A(int a)
{
printf("%d",a);
}
Play (k)
5:55 / 13:08
https://www.youtube.com/watch?v=AfBqVVKg4GE

Recursion and Stack - English

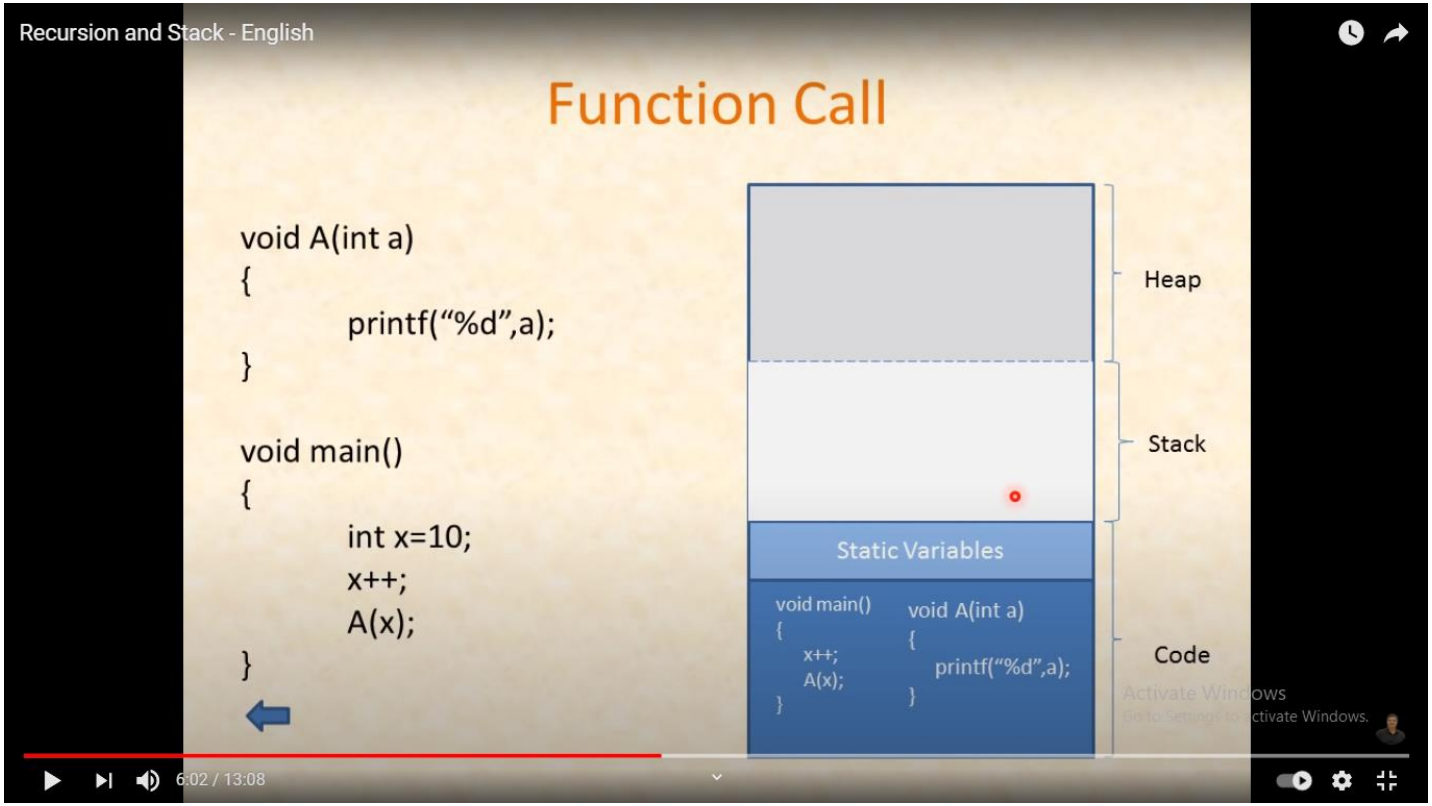
Function Call

```
void A(int a)
{
        printf("%d",a);
}

void main()
{
        int x=10;
        x++;
        A(x);
}
```



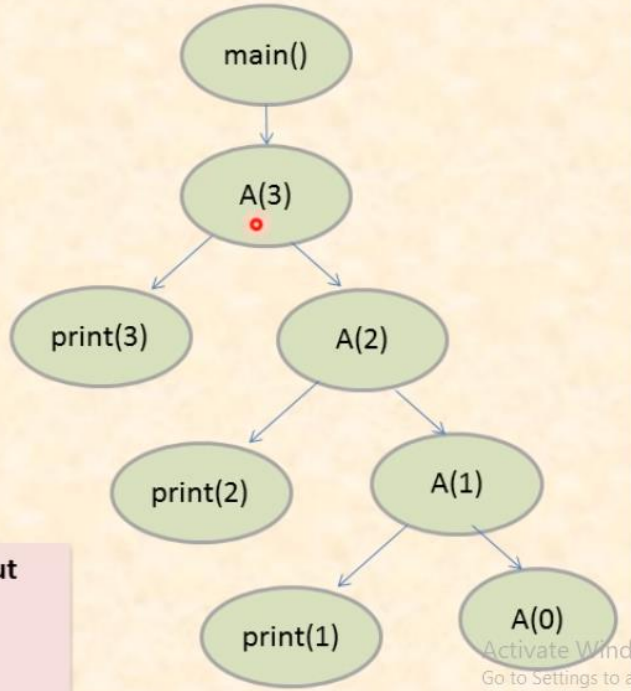
Play (k)

5:58 / 13:08


Recursion and Stack - English
Function Call
void A(int a)
{
printf("%d",a);
}
void main()
{
int x=10;
x++;
A(x);
}
Heap
Stack
Static Variables
void main()
{
x++;
A(x);
}
void A(int a)
{
printf("%d",a);
}
Code
Activate Windows
Go to Settings to activate Windows.
6:02 / 13:08

Recursion (Tail Recursion)

```
void A(int n)
{
    if(n>0)
    {
        printf("%d",n);
        A(n-1);
    }
}

void main()
{
        int x=3;

        A(x);
}
```



OutPut

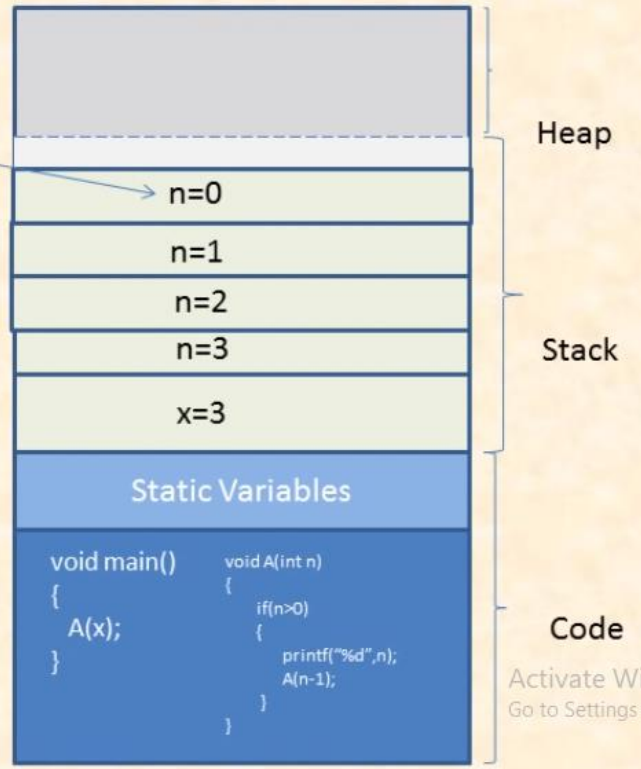
3
2
1

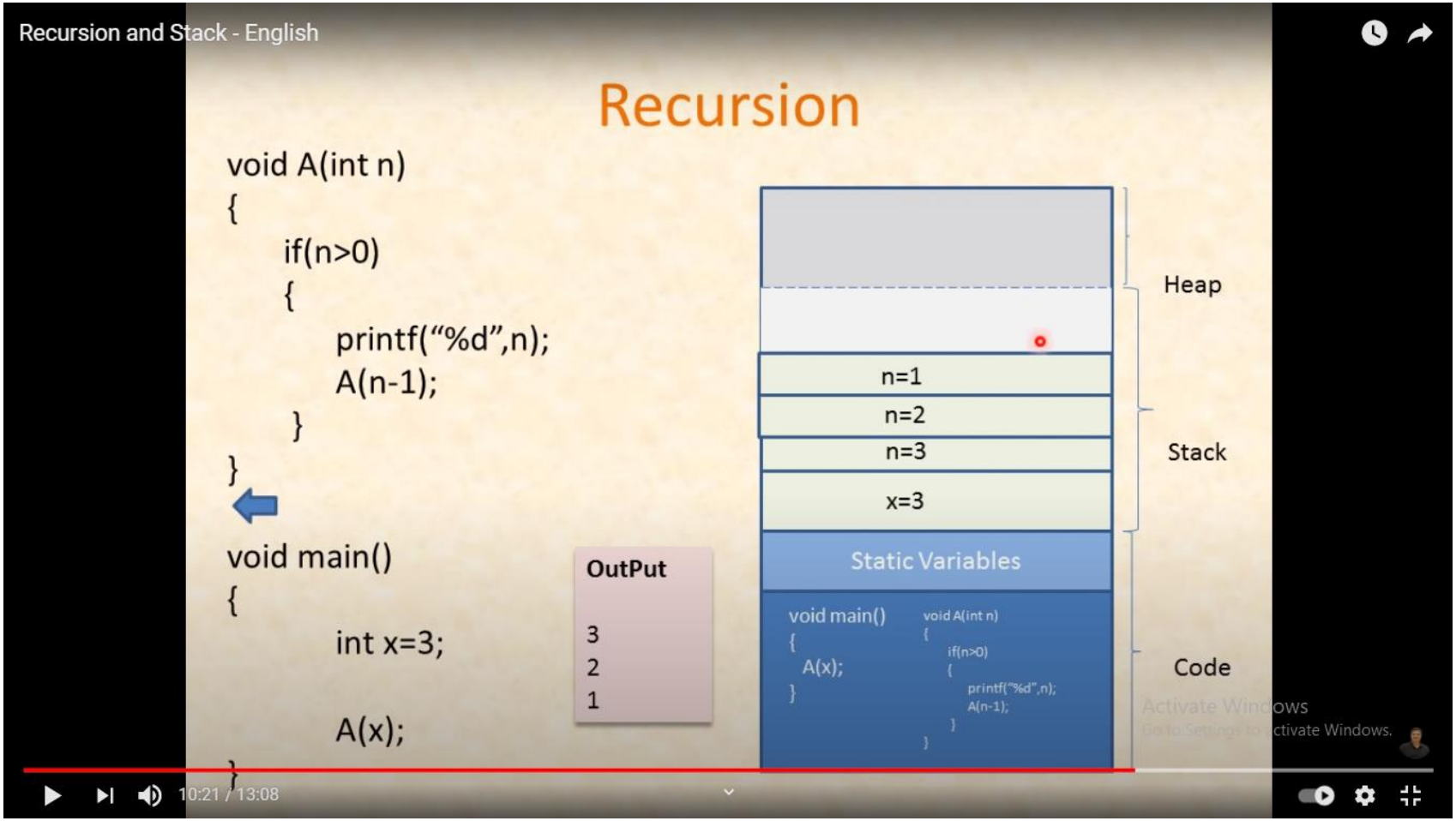
Recursion

```
void A(int n)
{
    if(n>0)
    {
        printf("%d",n);
        A(n-1);
    }
}

void main()
{
        int x=3;

        A(x);
}
```

OutPut

3
2
1



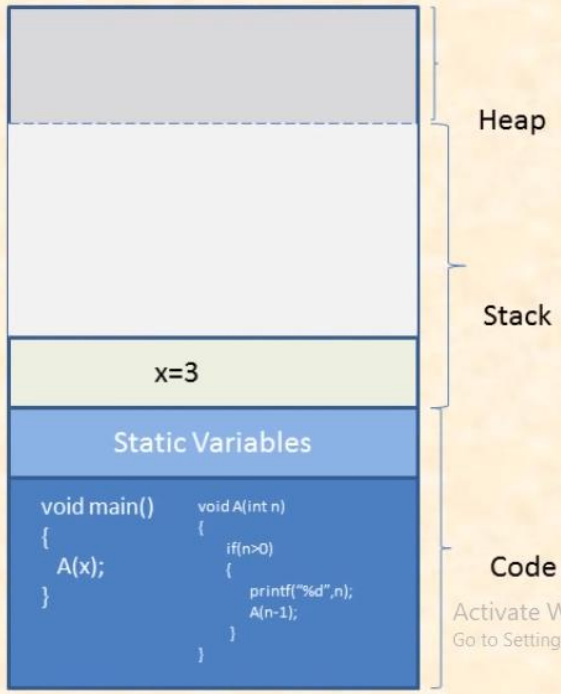

Recursion and Stack - English
Recursion
void A(int n)
{
if(n>0)
{
printf("%d",n);
A(n-1);
}
}
void main()
{
int x=3;
A(x);
}
OutPut
3
2
1
Heap
n=1
n=2
n=3
x=3
Stack
Static Variables
void main()
{
A(x);
}
void A(int n)
{
if(n>0)
{
printf("%d",n);
A(n-1);
}
}
Code
10:21 / 13:08

Recursion

```
void A(int n)
{
    if(n>0)
    {
        printf("%d",n);
        A(n-1);
    }
}

void main()
{
        int x=3;

        A(x);
}
```

OutPut

3
2
1

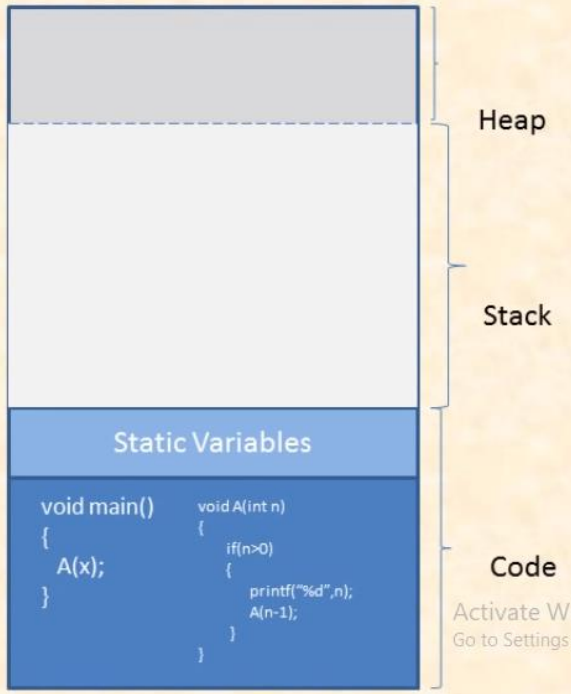


Recursion

```
void A(int n)
{
    if(n>0)
    {
        printf("%d",n);
        A(n-1);
    }
}

void main()
{
        int x=3;

        A(x);
}
```

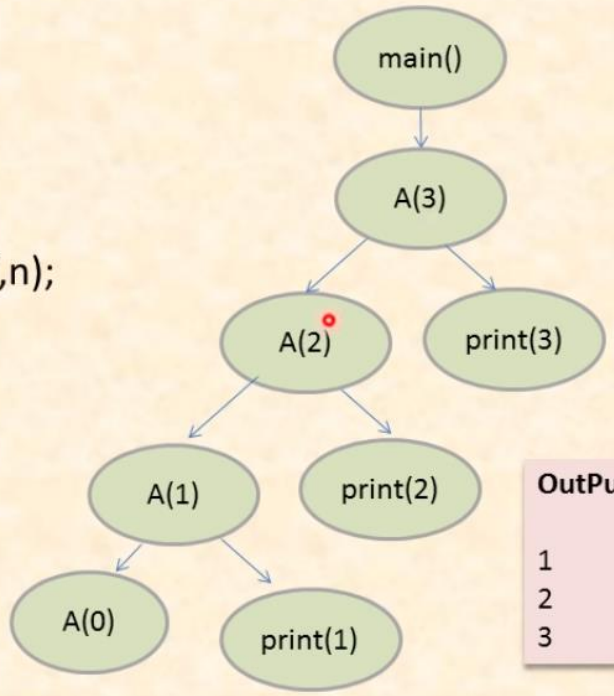
OutPut

3
2
1



Recursion Example 2 (Head Recursion)

```
void A(int n)
{
    if(n>0)
    {
        A(n-1);
        printf("%d",n);
    }
}

void main()
{
        int x=3;

        A(x);
}
```

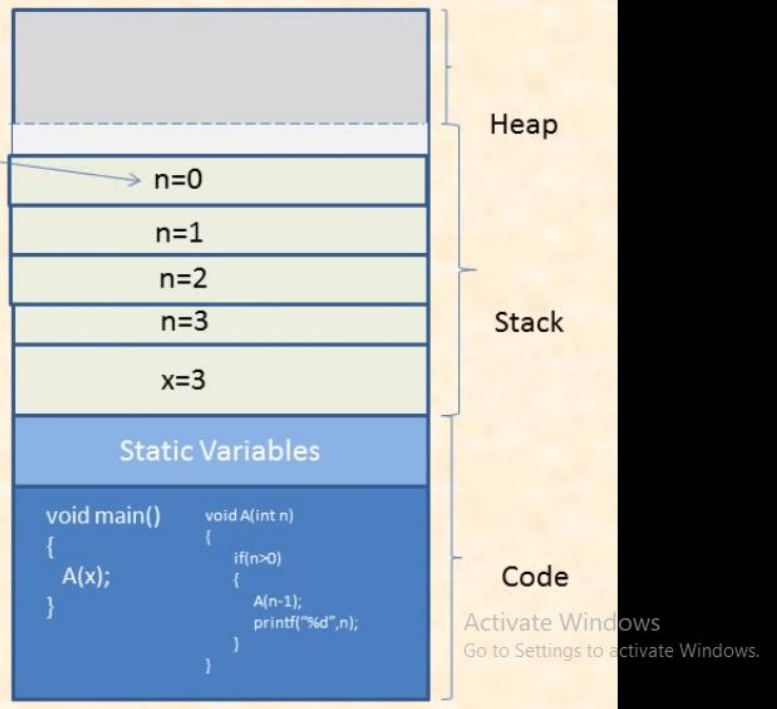


OutPut

1
2
3

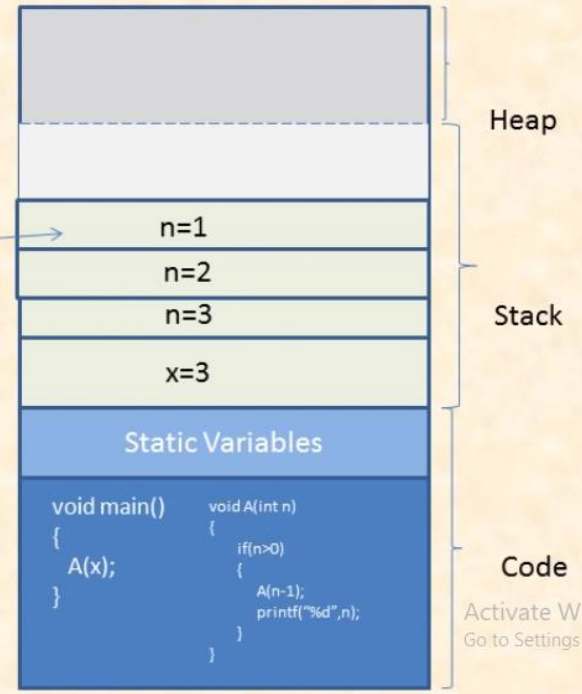
Recursion Example 2

```
void A(int n)
{
    if(n>0)
    {
        A(n-1);
        printf("%d",n);
    }
}

void main()
{
        int x=3;

        A(x);
}
```



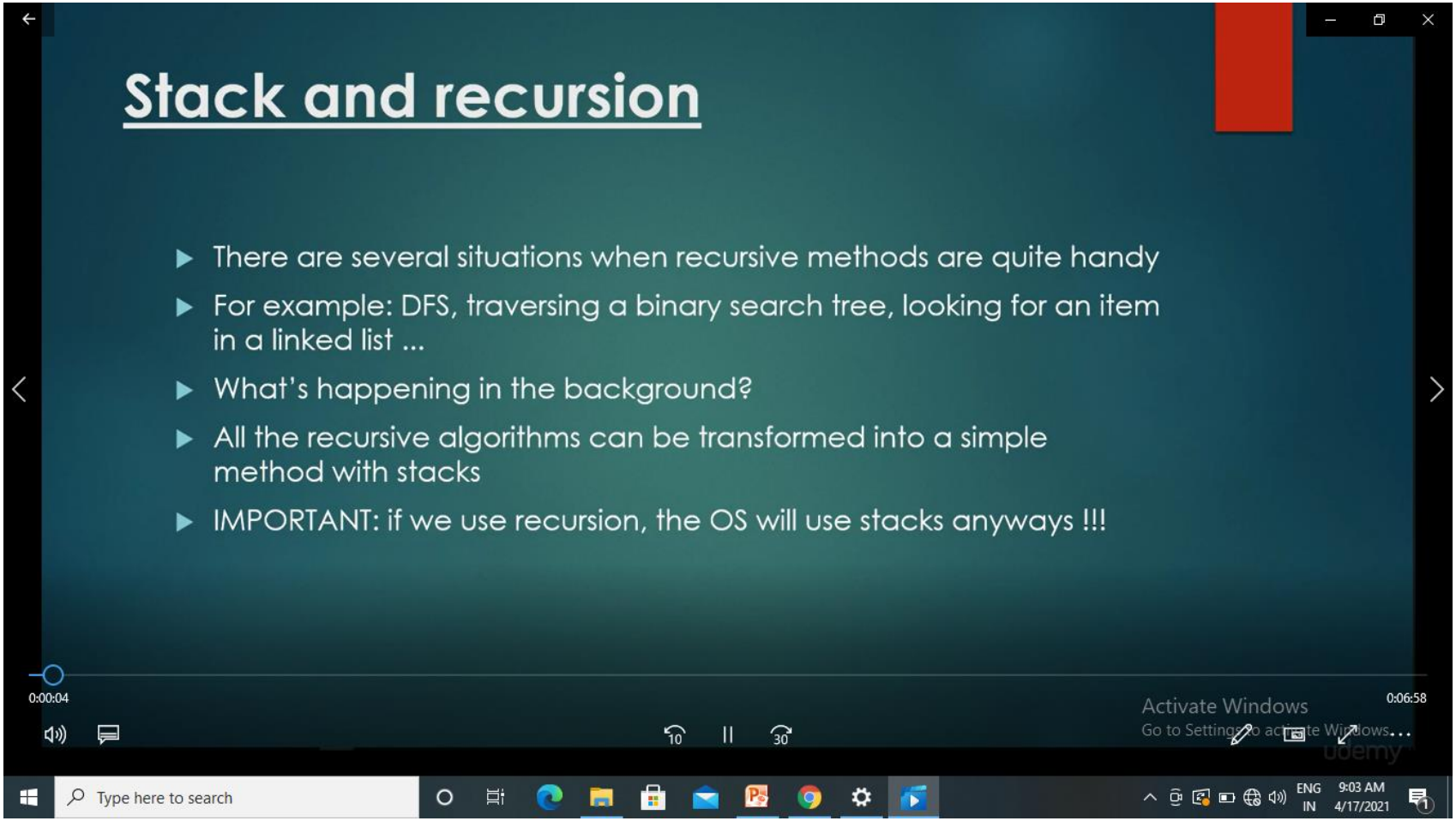
Recursion Example 2

```
void A(int n)
{
    if(n>0)
    {
        A(n-1);
        printf("%d",n);
    }
}

void main()
{
        int x=3;

        A(x);
}
```



OutPut

1


Stack and recursion
There are several situations when recursive methods are quite handy
For example: DFS, traversing a binary search tree, looking for an item in a linked list ...
What's happening in the background?
All the recursive algorithms can be transformed into a simple method with stacks
IMPORTANT: if we use recursion, the OS will use stacks anyways !!!
0:00:04
0:06:58
Activate Windows
Type here to search
ENG
IN
9:03 AM
4/17/2021

Differences Between Recursion and Iteration

Recursion

① Recursion can be defined as :
"A function called by itself repeatedly"

② Recursion is a process, always applied to a function.

③ Recursion reduces the size of the Code.

④ Recursion is Slow in Execution

⑤ A Conditional Statement decides the termination of recursion.

Iteration

① Iteration can be defined as
"When a loop repeatedly executes a set of instructions until the condition becomes false".

② Iteration is applied to the set of instructions which we want to get repeatedly executed and loops.

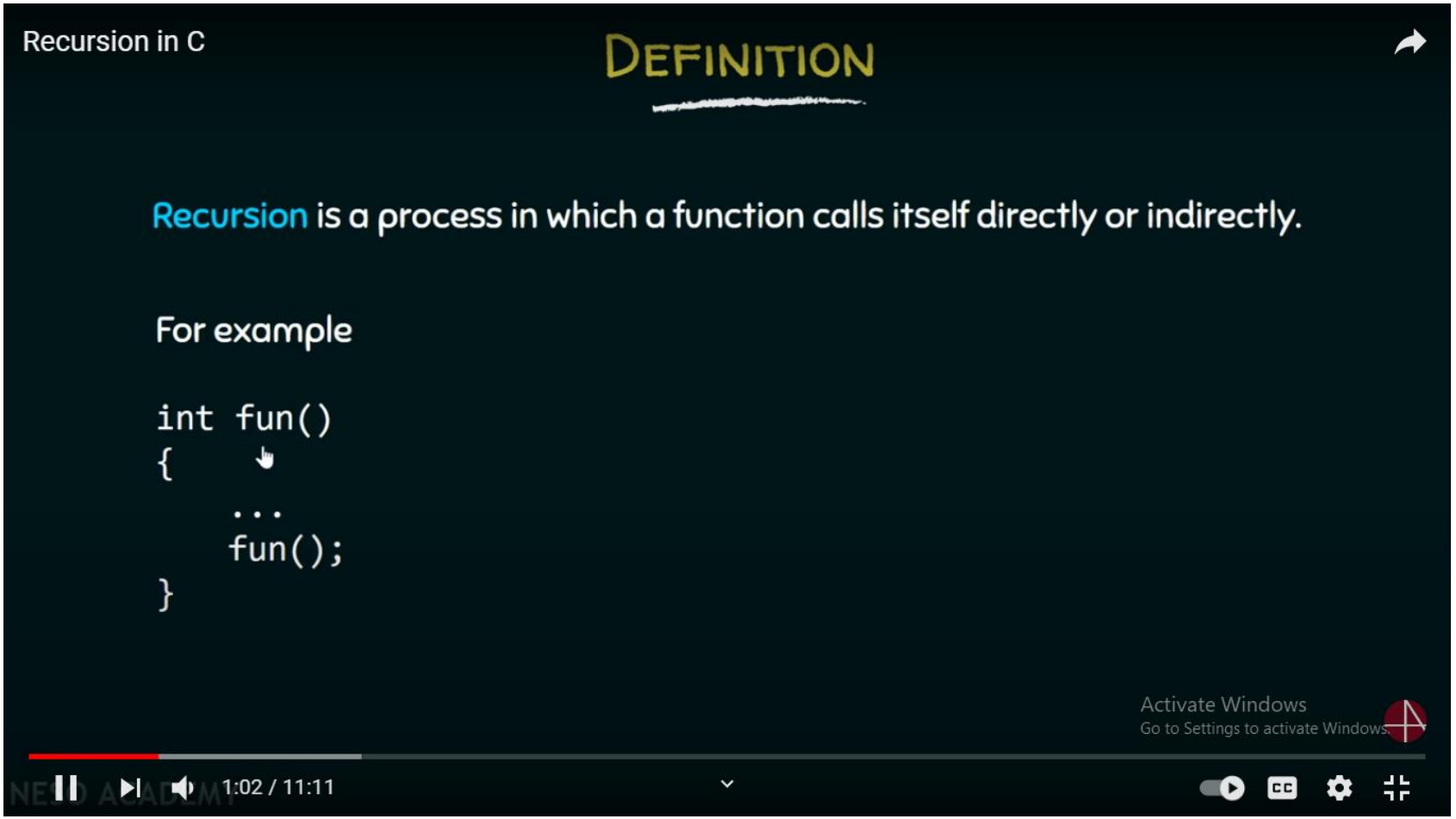
③ Iteration makes the Code longer.

④ Iteration is fast in Execution.

⑤ Control variable value decides the termination of Iteration Statement.

Differences Between Recursion and Iteration

Recursion	Iteration
⑥ The Stack is used to implement recursion. It stores the set of new local variables and parameters each time the function is called.	⑥ Iteration does not uses the stack
⑦ Infinite recursion can lead to a system crash.	⑦ Infinite loops uses CPU cycles repeatedly.
⑧ Recursive functions are easy to write, but they do not perform well as compared to Iteration.	⑧ Iteration is hard to write but their performance is good as compared to recursion.


Recursion in C
DEFINITION
Recursion is a process in which a function calls itself directly or indirectly.
For example
int fun()
{
...
fun();
}
Activate Windows
Go to Settings to activate Windows.
1:02 / 11:11

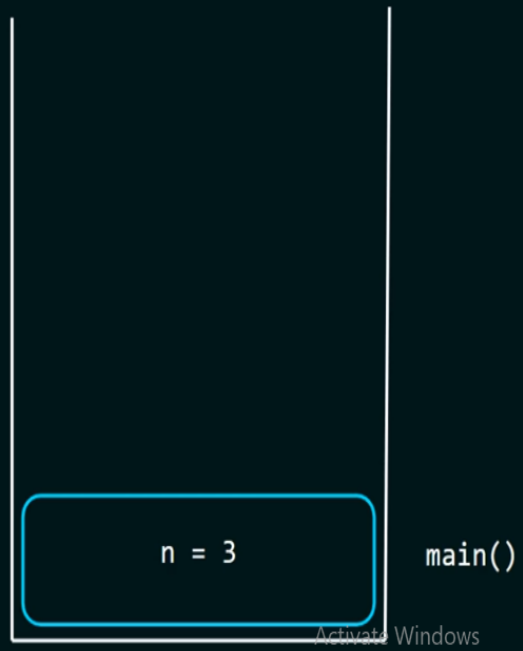
Indirect Recursion

A function is said to be indirect recursive if it calls another function and this new function calls the first calling function again.

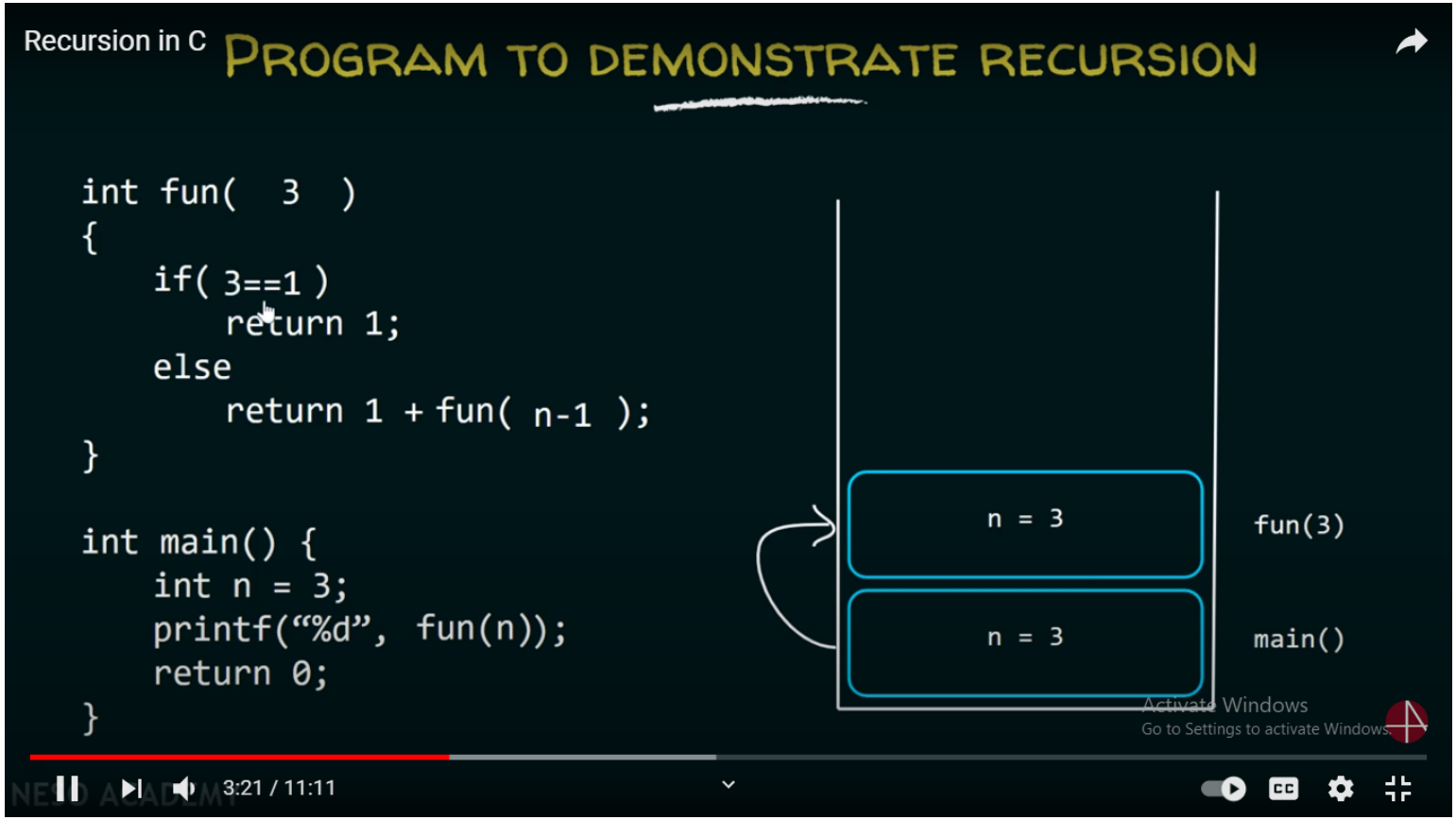
```
int func1(int n)
{
if (n<=1)
return 1;
Else
 return func2(n);
} int func2(int n)
 {
return func1(n);
 }
```

PROGRAM TO DEMONSTRATE RECURSION

```
int fun(int n)
{
    if( n==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```



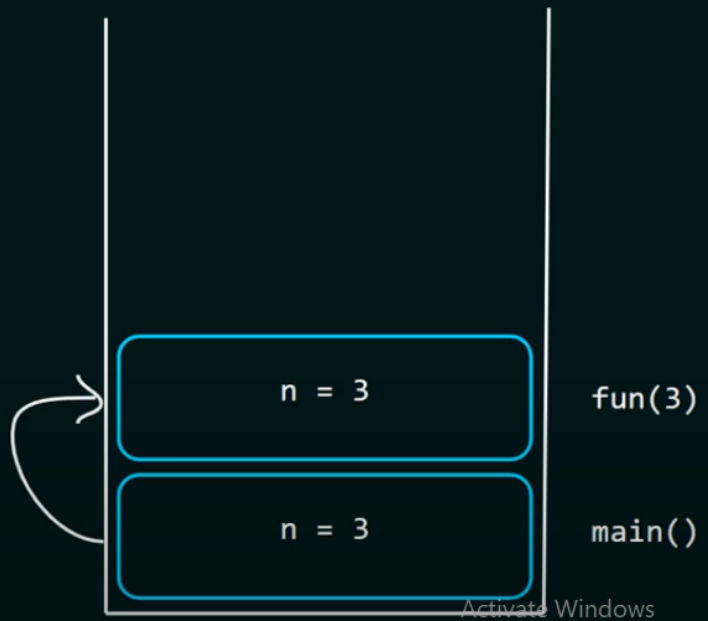
NESO ACADEMY


Recursion in C
PROGRAM TO DEMONSTRATE RECURSION
int fun(3)
{
if(3==1)
return 1;
else
return 1 + fun(n-1);
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
n = 3
fun(3)
n = 3
main()
Activate Windows
Go to Settings to activate Windows.
3:21 / 11:11

Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  3  )
{
    if(False)
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

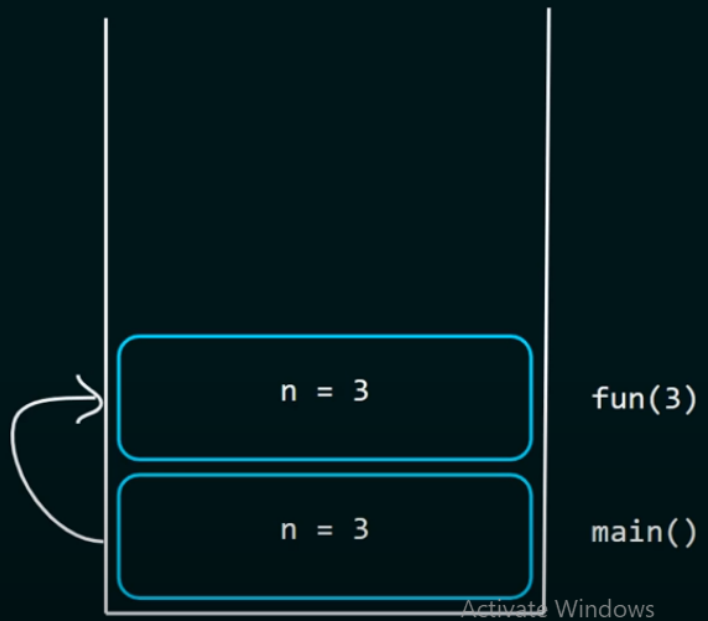


Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

Press Esc to exit full screen

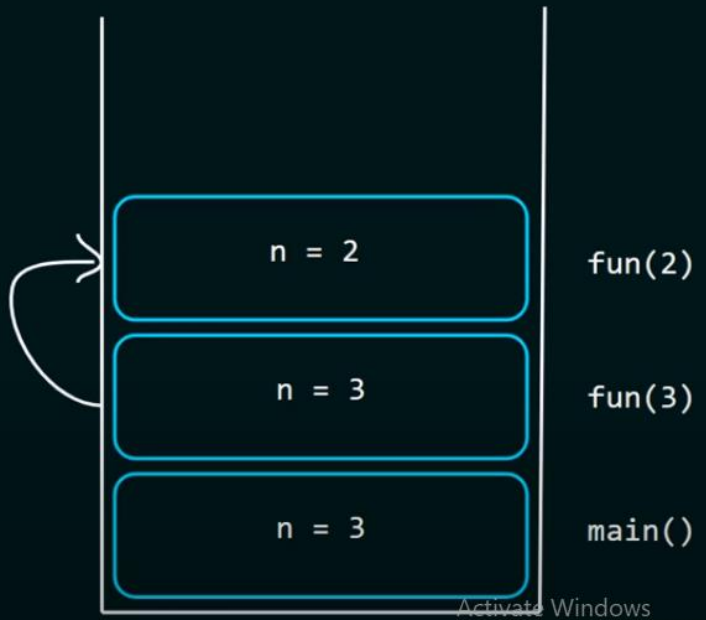
```
int fun(  3  )
{
    if(False)
        return 1;
    else
        return 1 + fun(  2  );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```



Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  2  )
{
    if( n==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

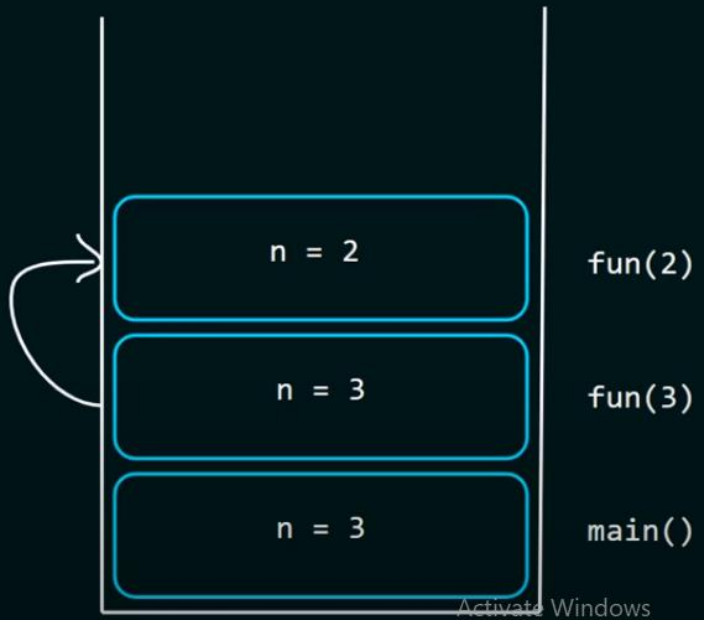


Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

Press Esc to exit full screen

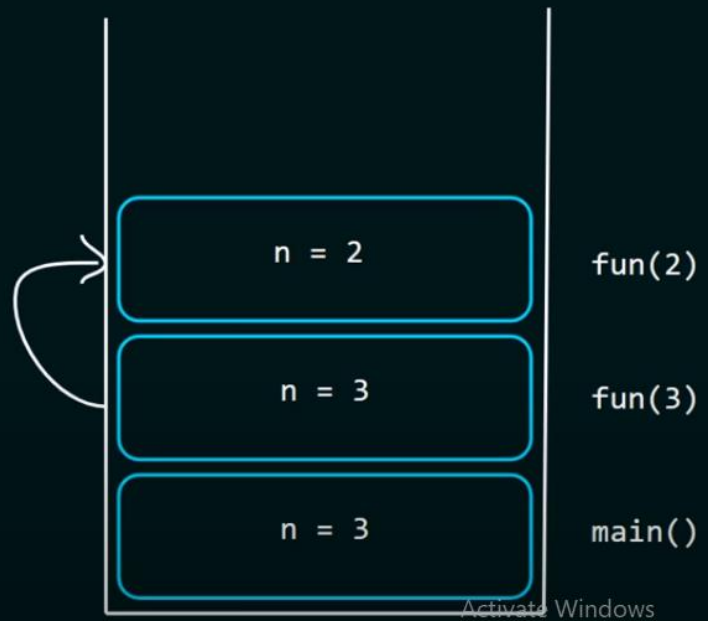
```
int fun(  2  )
{
    if( 2==1 )
        return 1;
    else
        return 1 + fun( n-1 );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

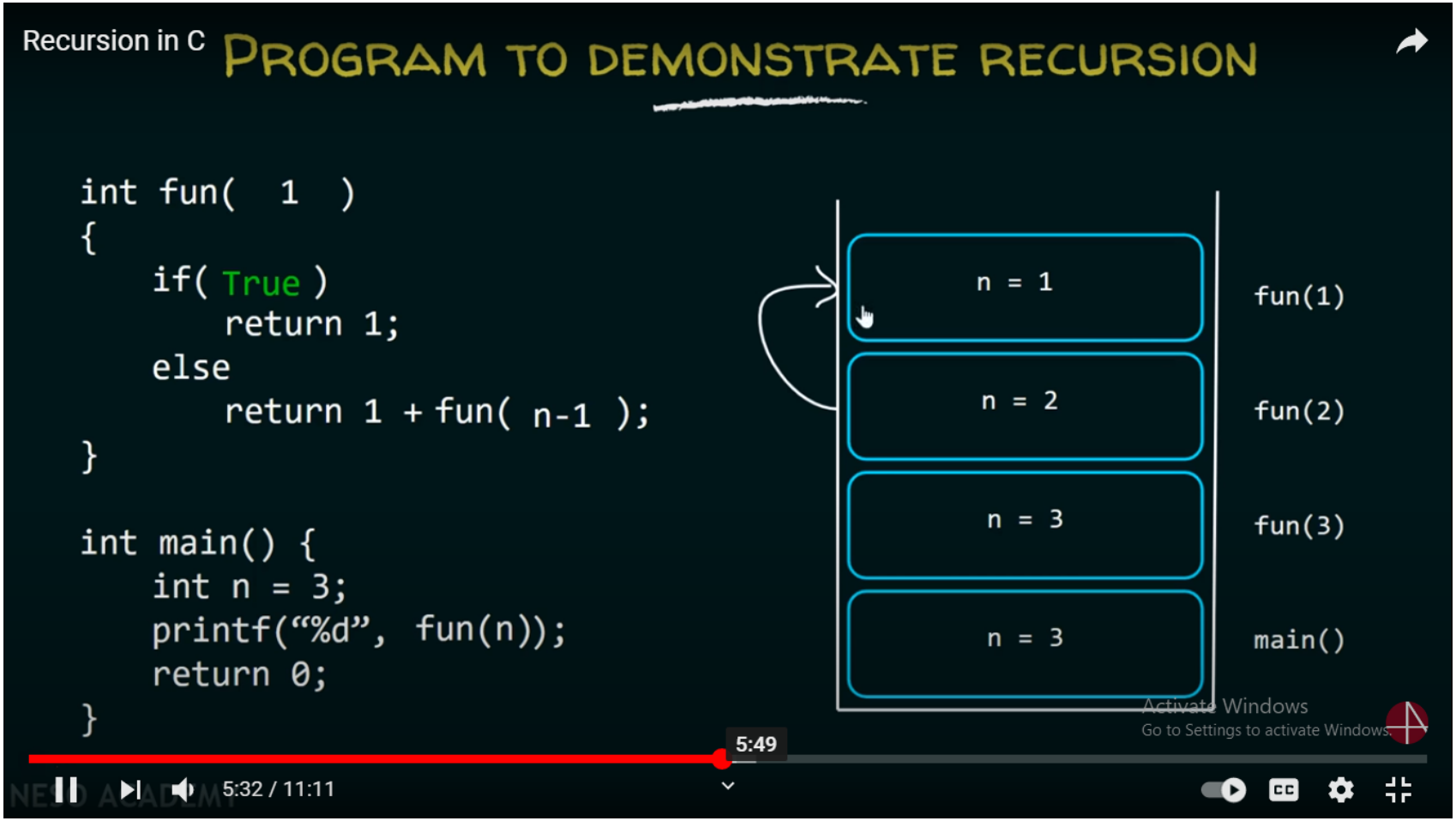


Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  2  )
{
    if(False)
        return 1;
    else
        return 1 + fun(  1  );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

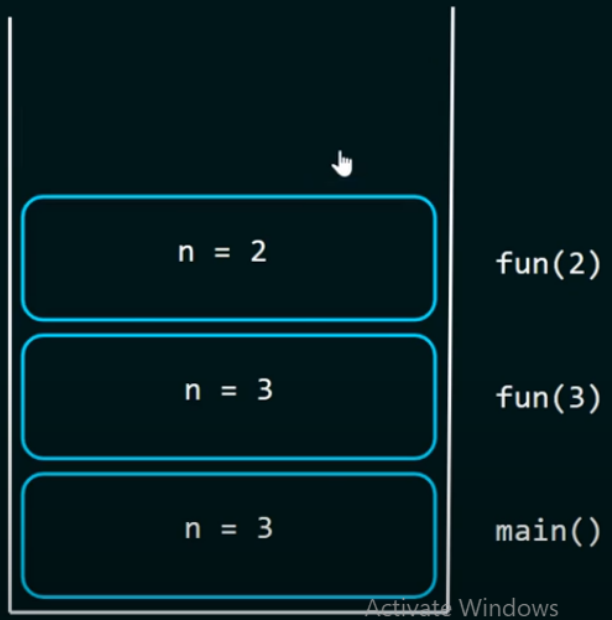



Recursion in C
PROGRAM TO DEMONSTRATE RECURSION
int fun(1)
{
if(True)
return 1;
else
return 1 + fun(n-1);
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
n = 1
n = 2
n = 3
n = 3
fun(1)
fun(2)
fun(3)
main()
Activate Windows
Go to Settings to activate Windows.
5:49
5:32 / 11:11

Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

```
int fun(  2  )
{
    if( n==1 )
        return 1;
    else
        return 1 + fun(  1  );
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

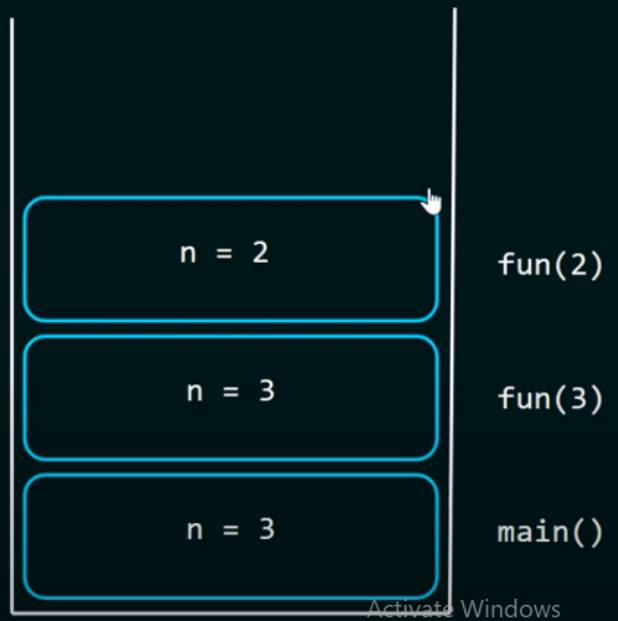


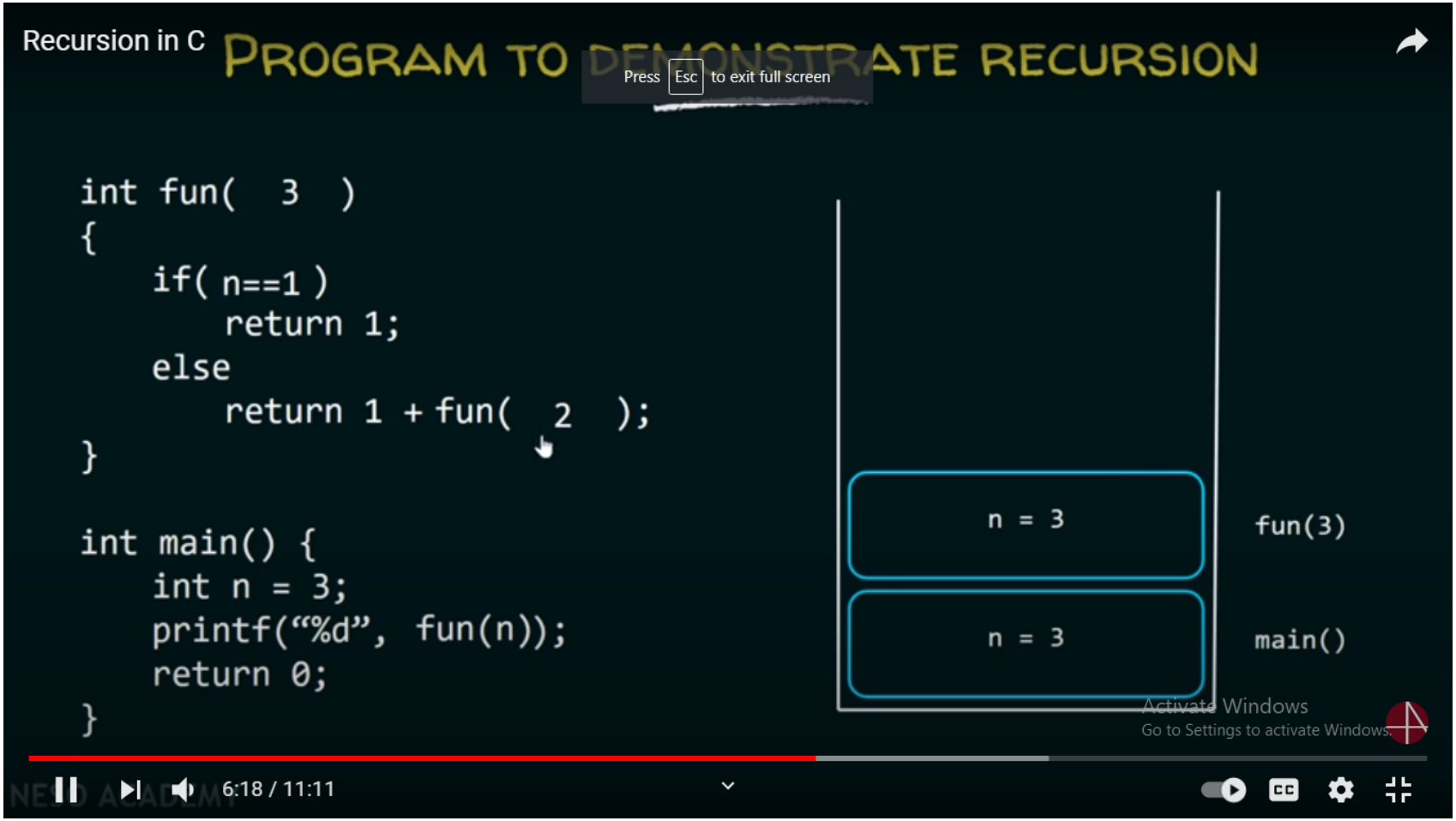
Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

Press Esc to exit full screen

```
int fun(  2  )
{
    if( n==1 )
        return 1;
    else
        return 1 + 1
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```



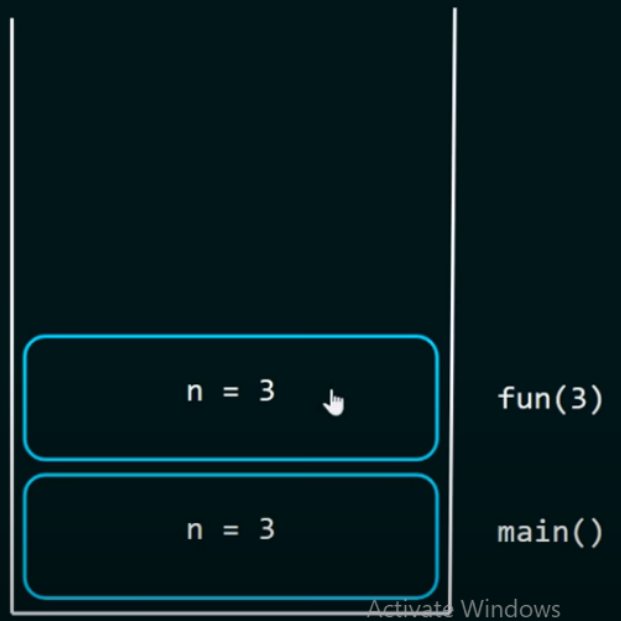

Recursion in C
PROGRAM TO DEMONSTRATE RECURSION
Press Esc to exit full screen
int fun(3)
{
if(n==1)
return 1;
else
return 1 + fun(2);
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
n = 3
fun(3)
n = 3
main()
Activate Windows
Go to Settings to activate Windows.
6:18 / 11:11

Recursion in C

PROGRAM TO DEMONSTRATE RECURSION

Press Esc to exit full screen

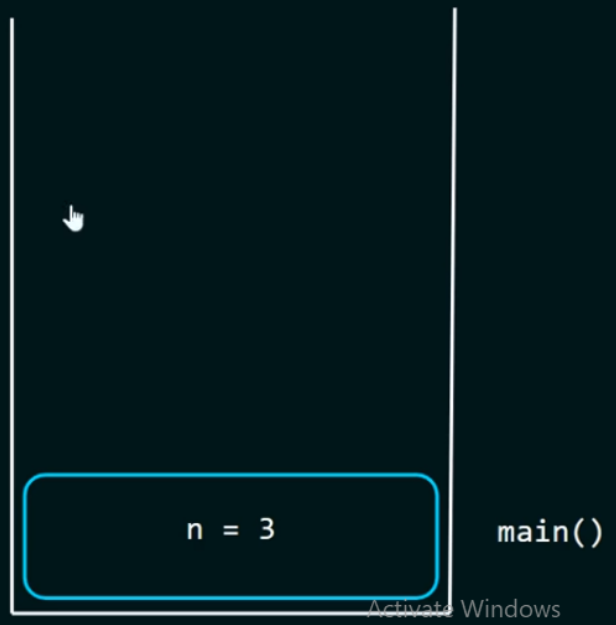
```
int fun(  3  )
{
    if( n==1 )
        return 1;
    else
        return 1 + 2
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

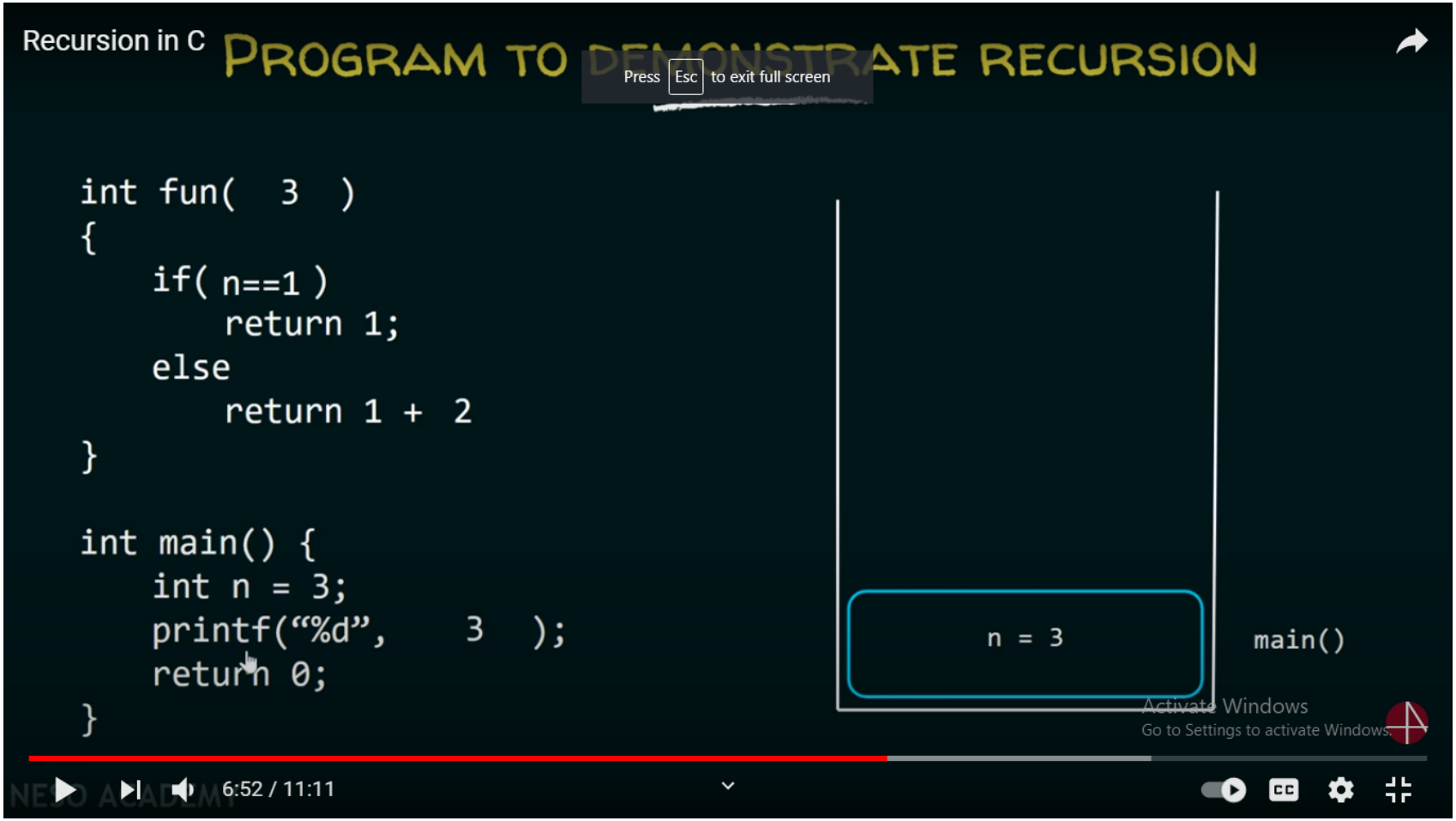


6:38 / 11:11

Program to demonstrate recursion

```
int fun(  3  )
{
    if( n==1 )
        return 1;
    else
        return 1 +  2
}

int main() {
    int n = 3;
    printf("%d", fun(n));
    return 0;
}
```

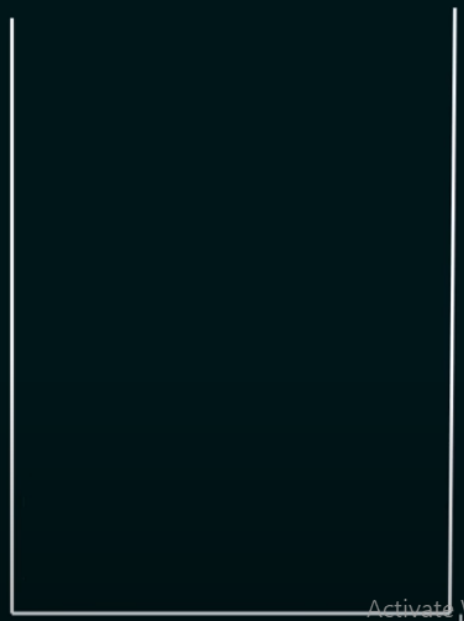



Recursion in C
PROGRAM TO DEMONSTRATE RECURSION
Press Esc to exit full screen
int fun(3)
{
if(n==1)
return 1;
else
return 1 + 2
}
int main() {
int n = 3;
printf("%d", 3);
return 0;
}
n = 3
main()
Activate Windows
Go to Settings to activate Windows.
6:52 / 11:11

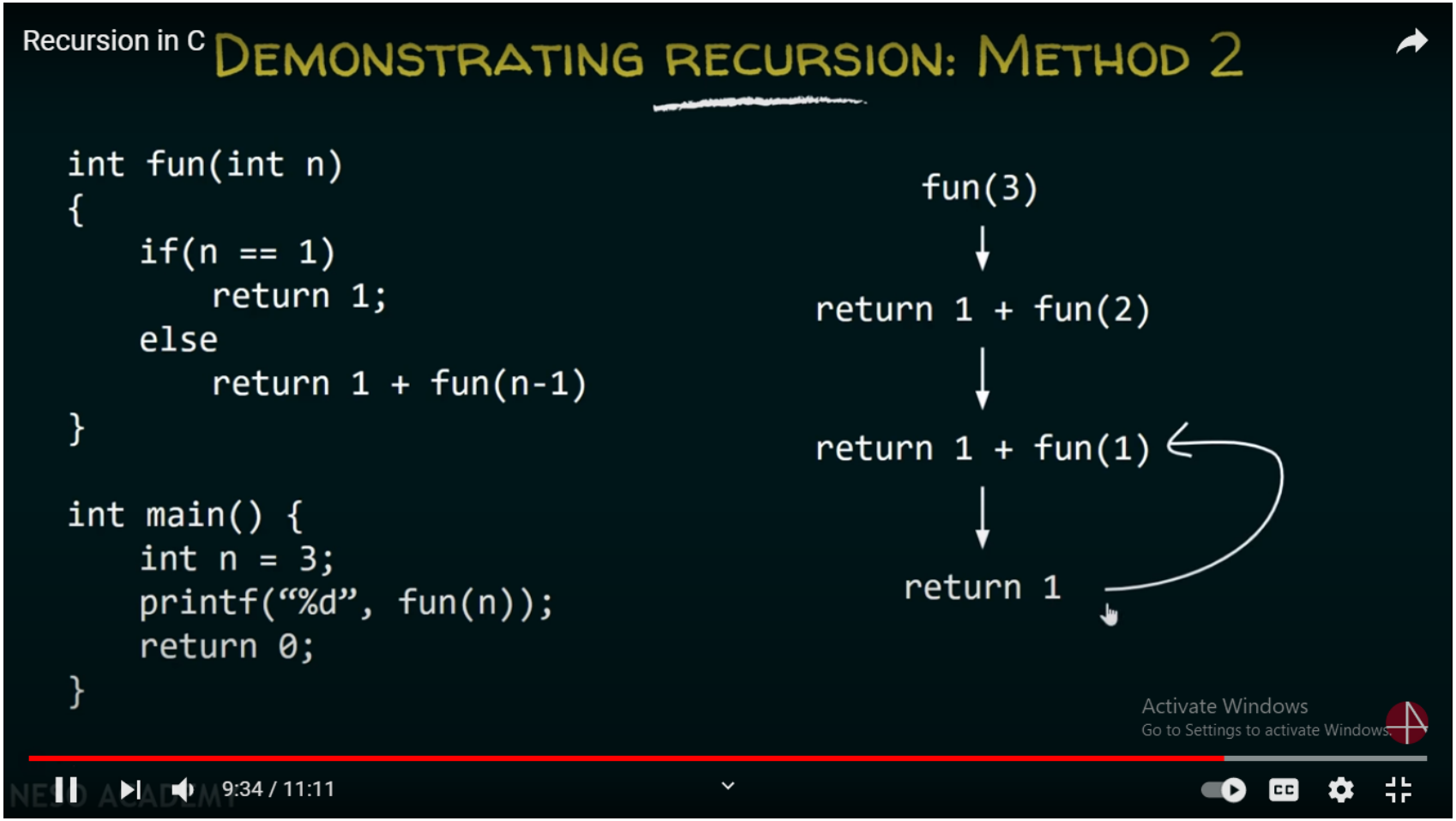
Recursion in C

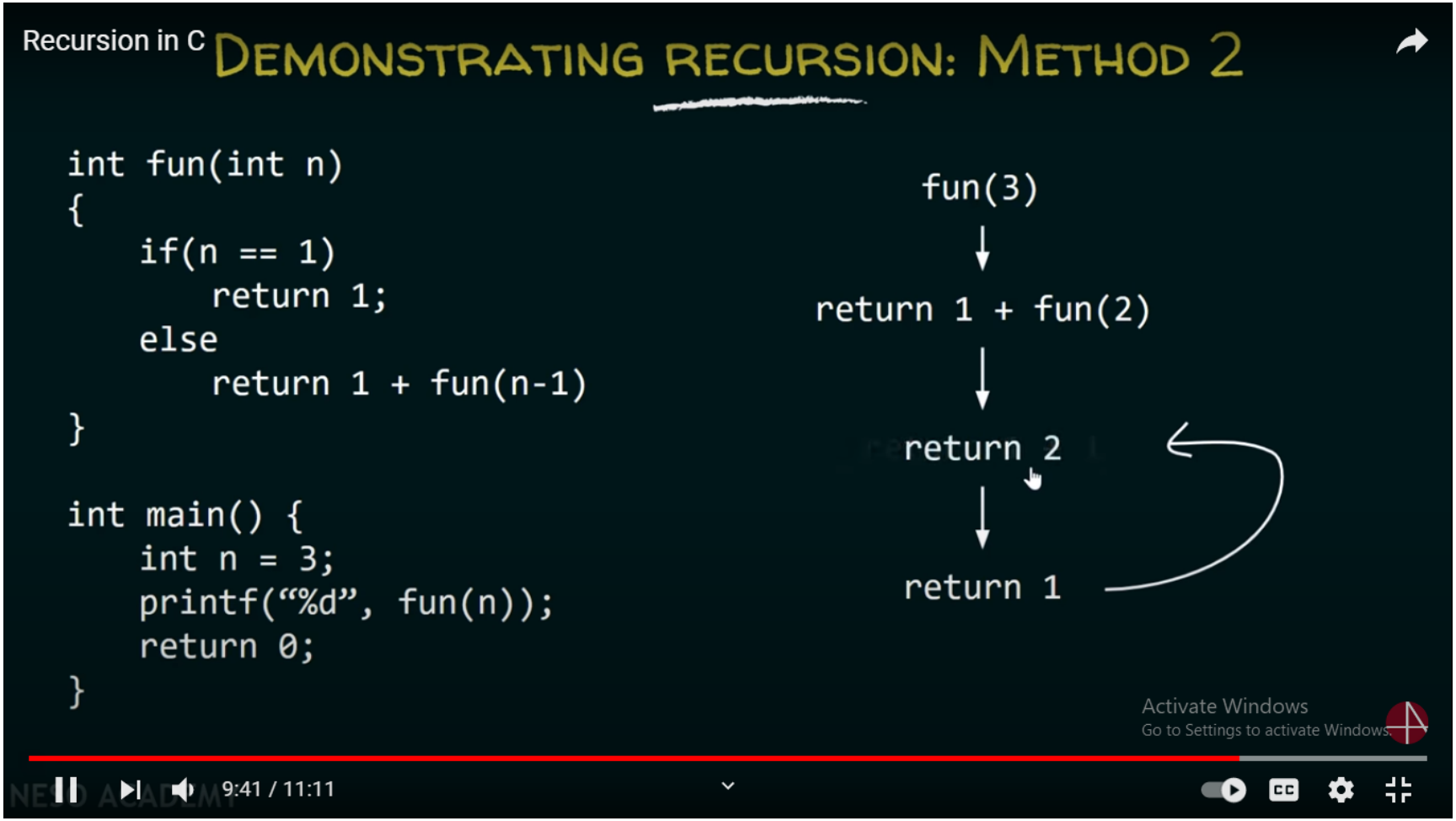
PROGRAM TO DEMONSTRATE RECURSION

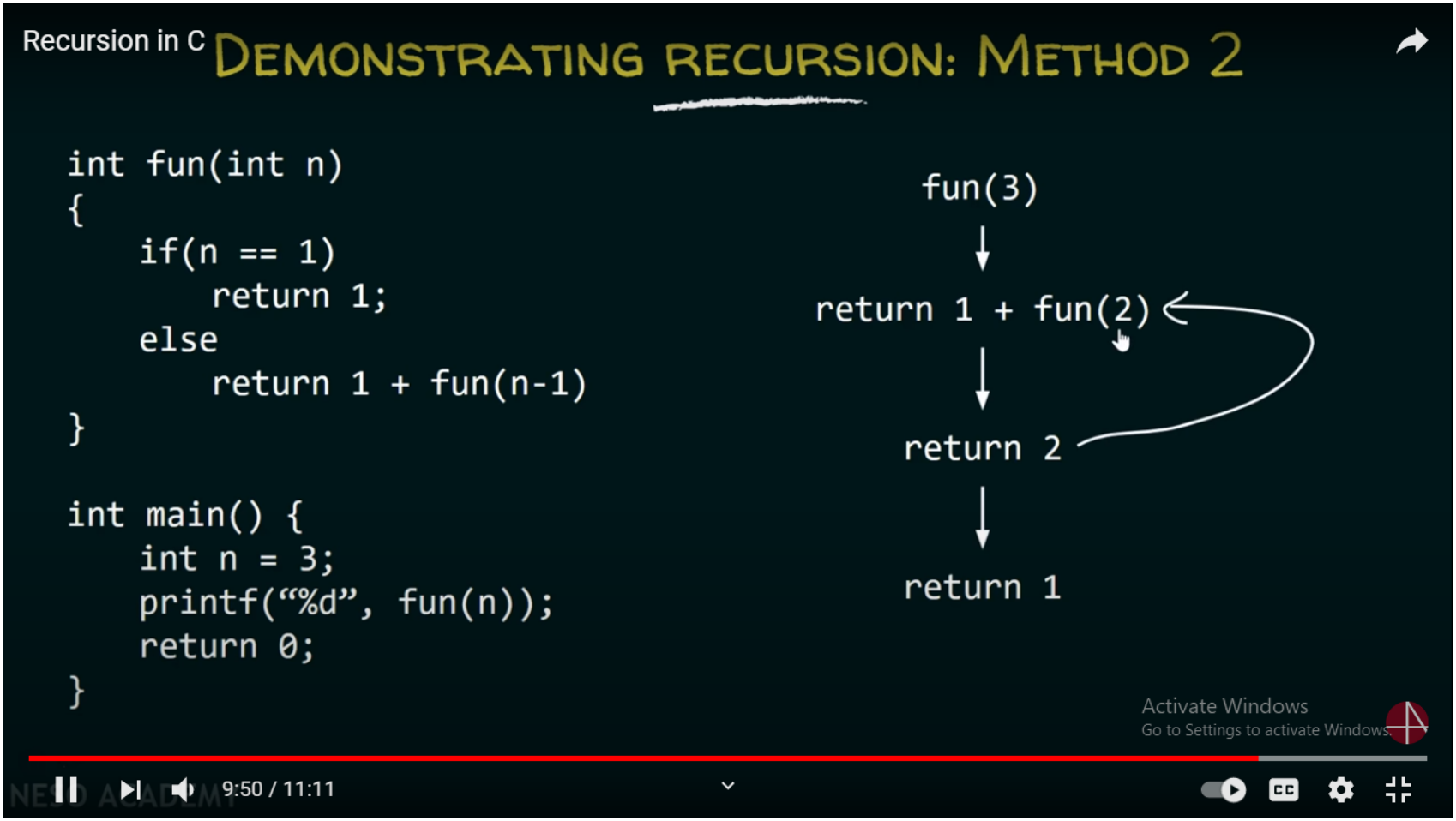
```
int fun(  3  )
{
    if( n==1 )
        return 1;
    else
        return 1 + 2
}

int main() {
    int n = 3;
    printf("%d",    3   );
    return 0;
}
```

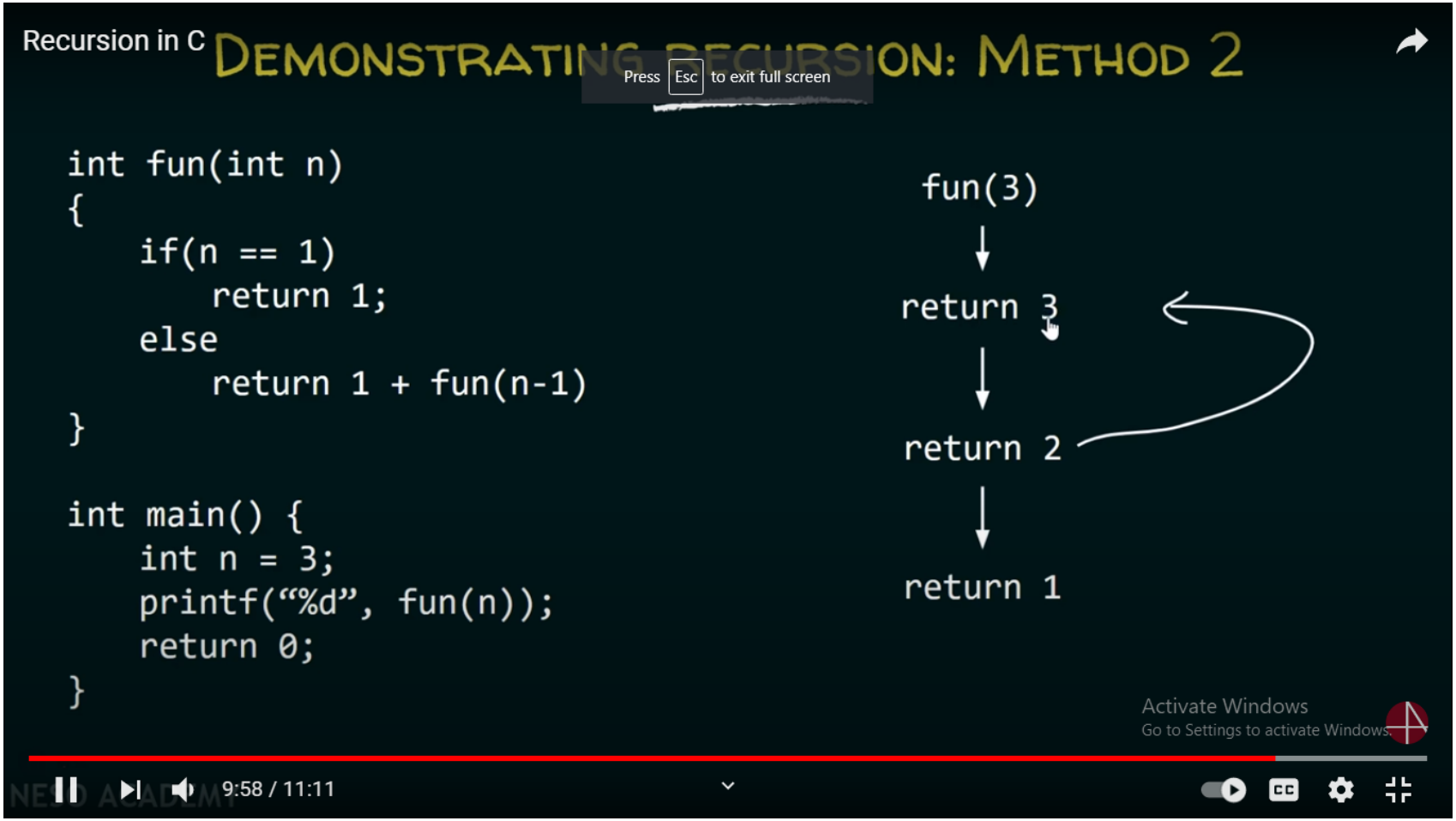


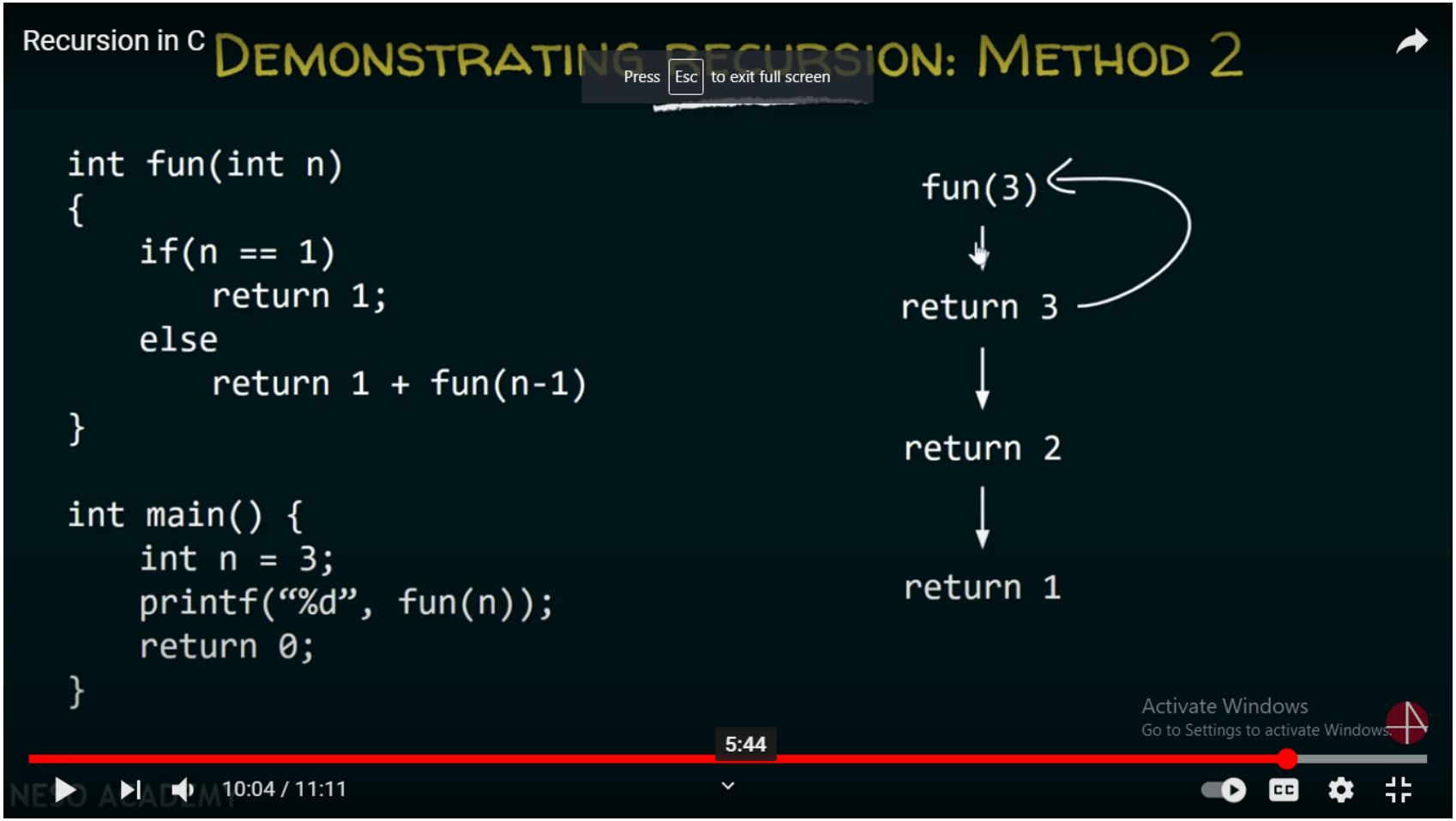
Output: 3

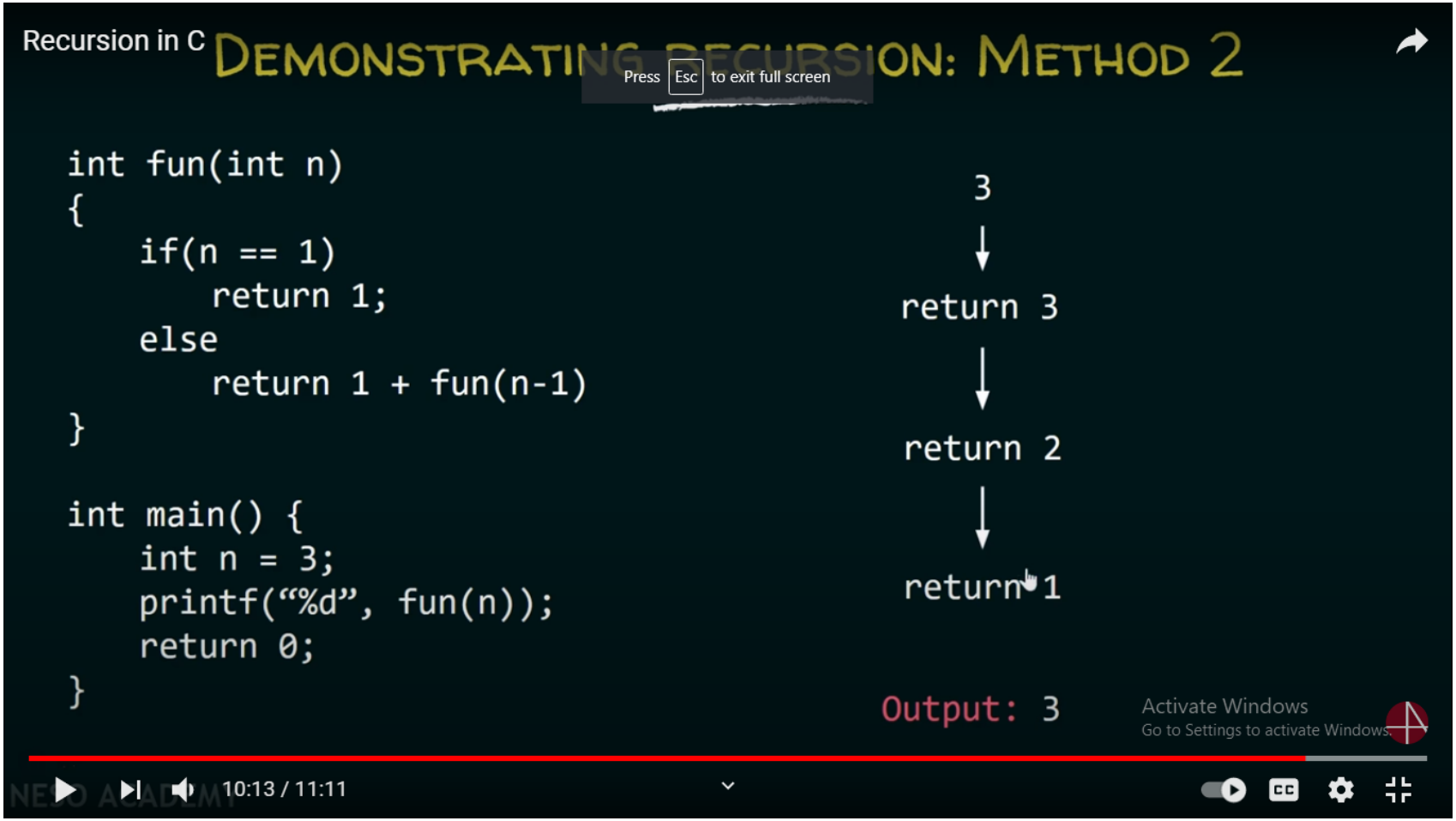

Recursion in C
DEMONSTRATING RECURSION: METHOD 2
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
fun(3)
return 1 + fun(2)
return 1 + fun(1)
return 1
Activate Windows
Go to Settings to activate Windows.
9:34 / 11:11


Recursion in C
DEMONSTRATING RECURSION: METHOD 2
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
fun(3)
return 1 + fun(2)
return 2
return 1
Activate Windows
Go to Settings to activate Windows.
9:41 / 11:11

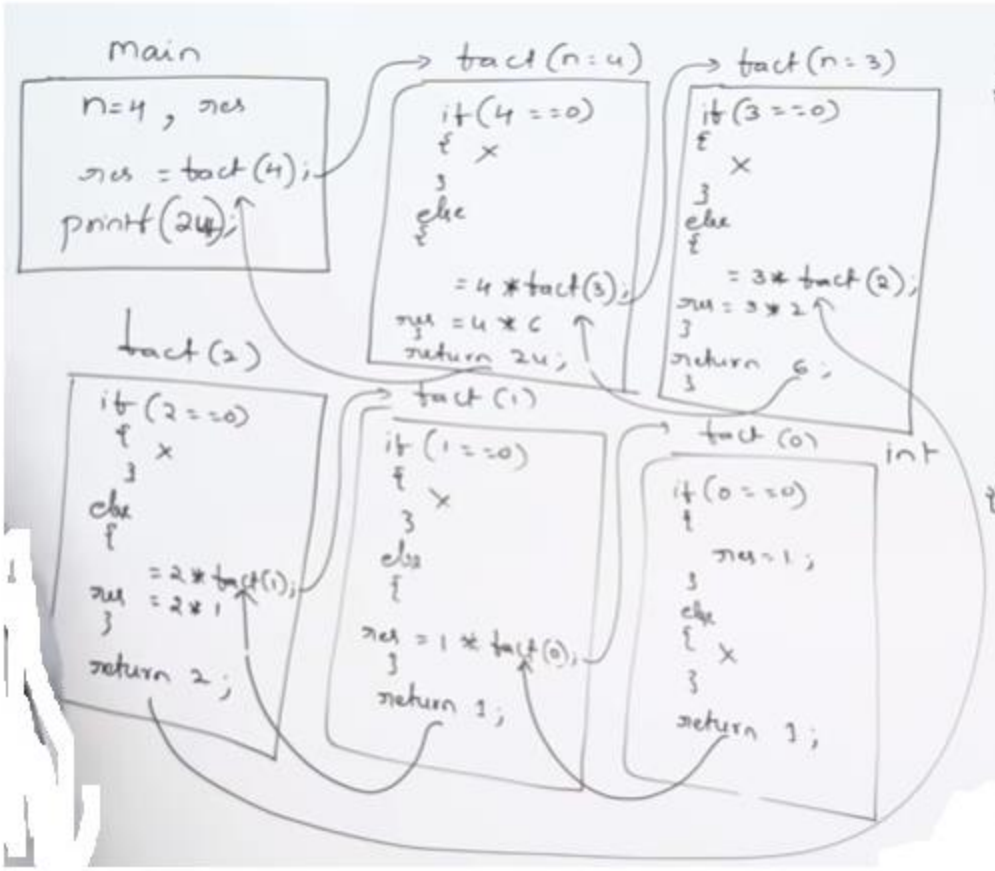

Recursion in C
DEMONSTRATING RECURSION: METHOD 2
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
fun(3)
return 1 + fun(2)
return 2
return 1
Activate Windows
Go to Settings to activate Windows.
9:50 / 11:11


Recursion in C
DEMONSTRATING RECURSION: METHOD 2
Press Esc to exit full screen
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
fun(3)
return 3
return 2
return 1
Activate Windows
Go to Settings to activate Windows.
9:58 / 11:11


Recursion in C
DEMONSTRATING RECURSION: METHOD 2
Press Esc to exit full screen
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
fun(3)
return 3
return 2
return 1
Activate Windows
Go to Settings to activate Windows.
5:44
10:04 / 11:11


Recursion in C
DEMONSTRATING RECURSION: METHOD 2
Press Esc to exit full screen
int fun(int n)
{
if(n == 1)
return 1;
else
return 1 + fun(n-1)
}
int main() {
int n = 3;
printf("%d", fun(n));
return 0;
}
3
return 3
return 2
return 1
Output: 3
Activate Windows
Go to Settings to activate Windows.
10:13 / 11:11

RECURSION



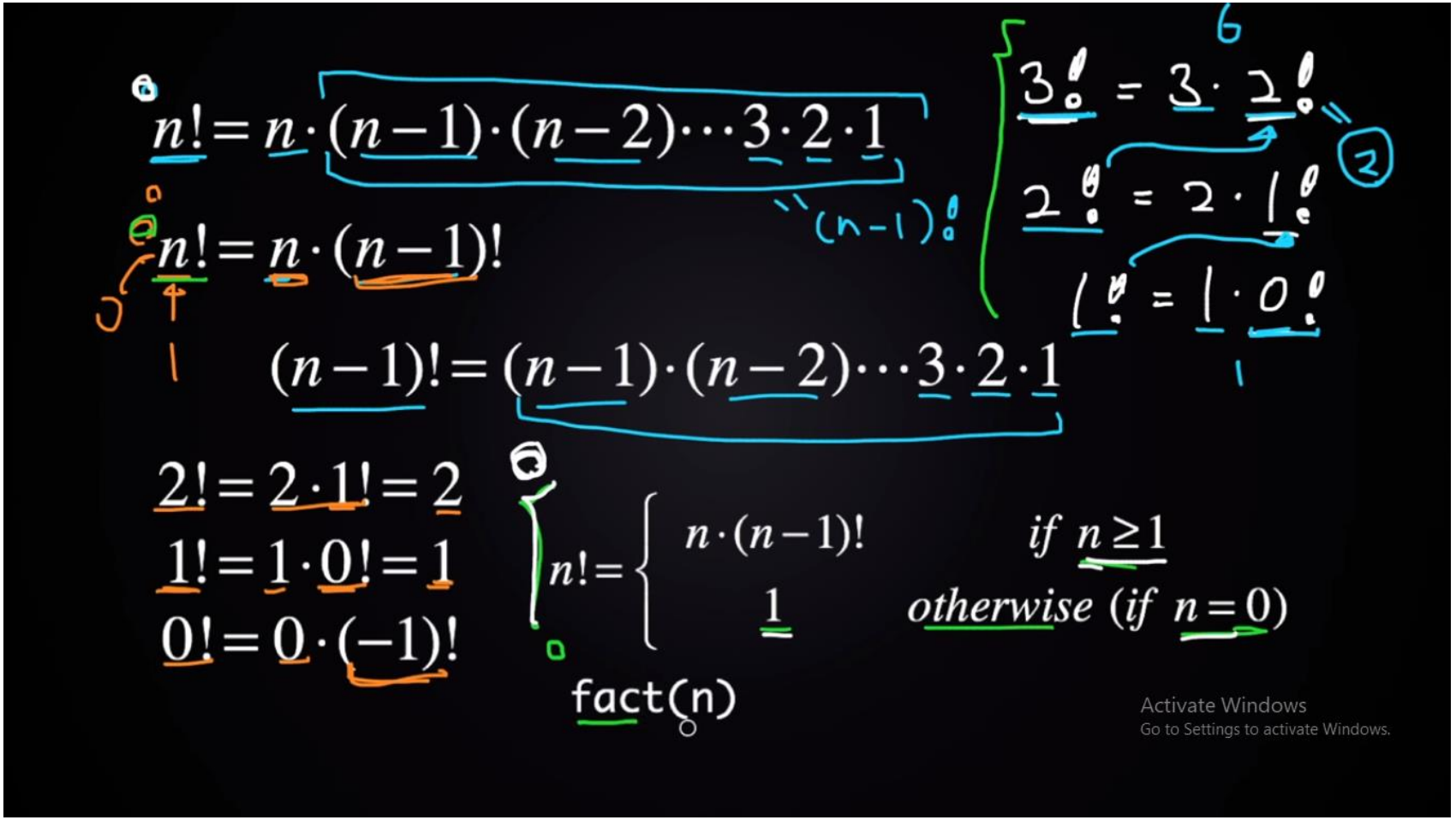

```
main( )
{
    int n, res;
    pf("Enter n valu
    sf("%d", &n);
    res = fact(n);
    printf(" result : %
}
fact(int n)
{
    int res;
    if(n==0)
        res = 1;
    else
    {
        res = n * fact(n-
    }
    return res
```

Factorial application of stack using recursion

- The factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 6 is 1*2*3*4*5*6 = 720.
- Factorial is not defined for negative numbers
- factorial of zero is one, 0! = 1.

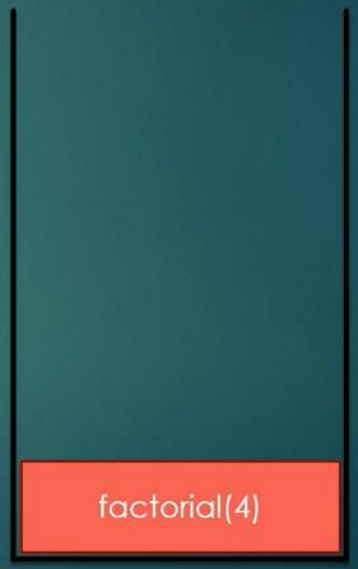
Algorithm

```
function result = factorial ( N )
  {
    if ( N == 0 )
      {
        result = 1;
      }
    else
      {
        result = N * factorial(N-1);
      }
  }
```


$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
$(n-1)!$
$n! = n \cdot (n-1)!$
$3! = 3 \cdot 2!$
$2! = 2 \cdot 1!$
$1! = 1 \cdot 0!$
$(n-1)! = (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
$2! = 2 \cdot 1! = 2$
$1! = 1 \cdot 0! = 1$
$0! = 0 \cdot (-1)!$
$n! = \begin{cases} n \cdot (n-1)! & \text{if } n \geq 1 \\ 1 & \text{otherwise (if } n = 0) \end{cases}$
fact(n)
Activate Windows
Go to Settings to activate Windows.

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}


int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}


int result = factorial(4)
```

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}


int result = factorial(4)
```

0:04:51

0:02:11

ENG IN 9:06 AM 4/17/2021

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

0:05:00

0:02:02

Type here to search

ENG IN 9:06 AM 4/17/2021

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

0:05:08

0:01:54

Type here to search

ENG IN 9:07 AM 4/17/2021

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

0:05:36

0:01:26

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

Factorial: factorial(4)

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

Factorial: factorial(4)

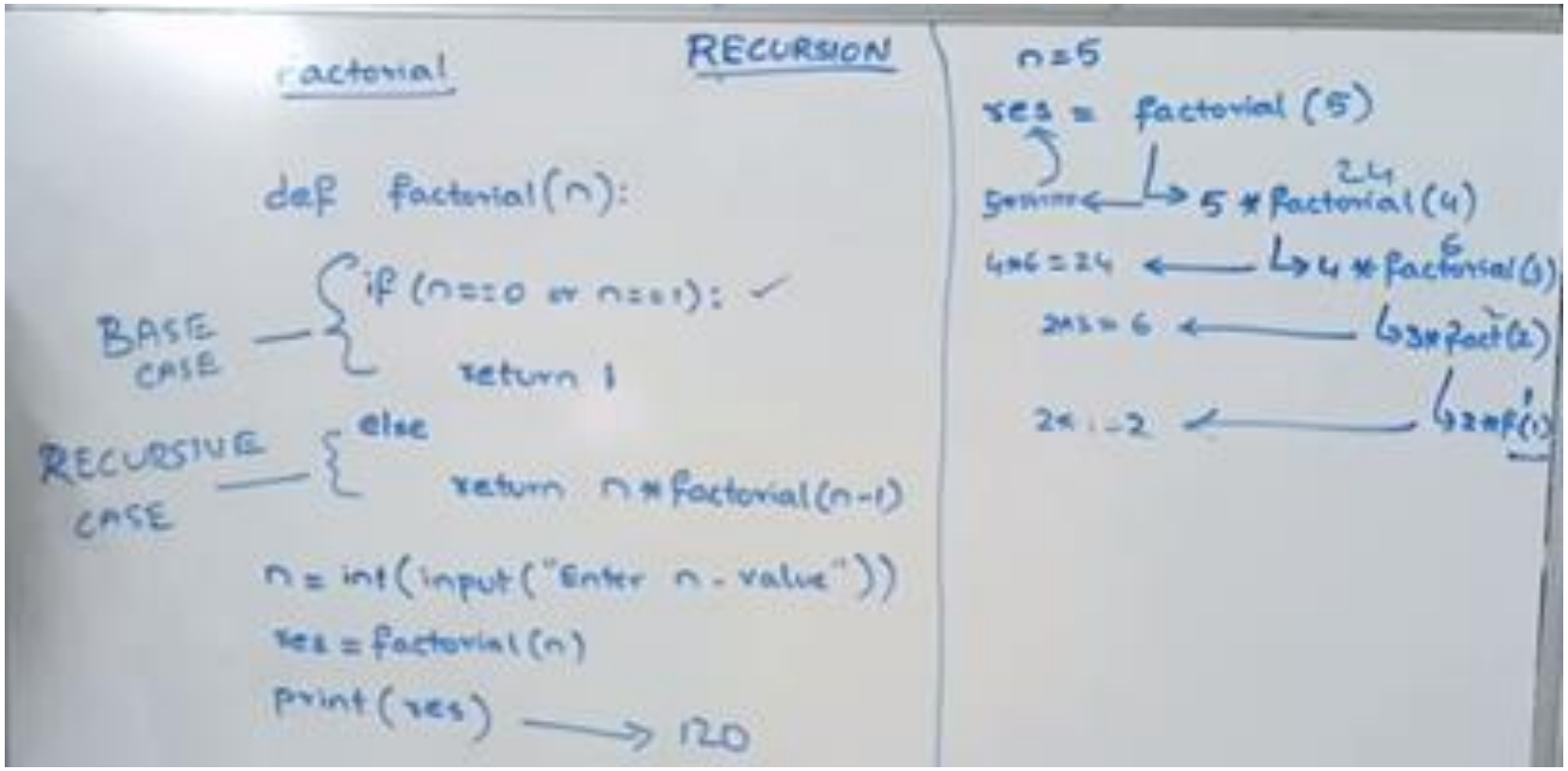
Conclusion: recursive method calls are going to be piled up in the stack

```
public void factorial(int n) {

        if( n == 0 )
                return 1;

        return n * factorial(n-1);
}
```

020 Stacks and recursive method calls

int result = factorial(4)

Result will be 4*3*2*1 = 24 !!!


factorial
RECURSION
def factorial(n):
if (n==0 or n==1):
BASE CASE
return 1
RECURSIVE CASE
else
return n*factorial(n-1)
n = int(input("Enter n-value"))
res = factorial(n)
print(res)
120
n=5
res = factorial(5)
5 * factorial(4)
24
4*6=24
4 * factorial(3)
6
3*2=6
3*fact(2)
2*1=2
2*f(1)

```
from stack import stack as Stack
 def stk_factorial(n):
 stk = Stack(n-1)
 # create a stack for n-1 elements
     while n > 1:
 stk.push(n)
n -= 1 # decrement n by 1
 result = 1
    while not stk.is_empty():
n = stk.pop()
 result *= n
# multiply result by n
   return result
```

Factorial program

```
#include<stdio.h>
int find_factorial(int);
 int main()
{
int num, fact;
//Ask user for the input and store it in num
printf("\nEnter any integer number:");
 scanf("%d",&num);
//Calling our user defined function
fact =find_factorial(num);
//Displaying factorial of input number
printf("\nfactorial of %d is: %d",num, fact);
 return 0;
 }
```

```
int find_factorial(int n)
 {
//Factorial of 0 is 1
if(n==0)
 return(1);
 //Function calling itself: recursion
return(n*find_factorial(n-1)); }
```

- **Output:**
- Enter any integer number: 4 factorial of 4 is: 24

```
# Python 3 program to find
# factorial of given number
def factorial(n):

    # Checking the number
    # is 1 or 0 then
    # return 1
    # other wise return
    # factorial
    if (n==1 or n==0):

        return 1

    else:

        return (n * factorial(n - 1))

# Driver Code
num = 5;
print("number : ",num)
print("Factorial : ",factorial(num))
```

Output:

Number : 5

Factorial : 120

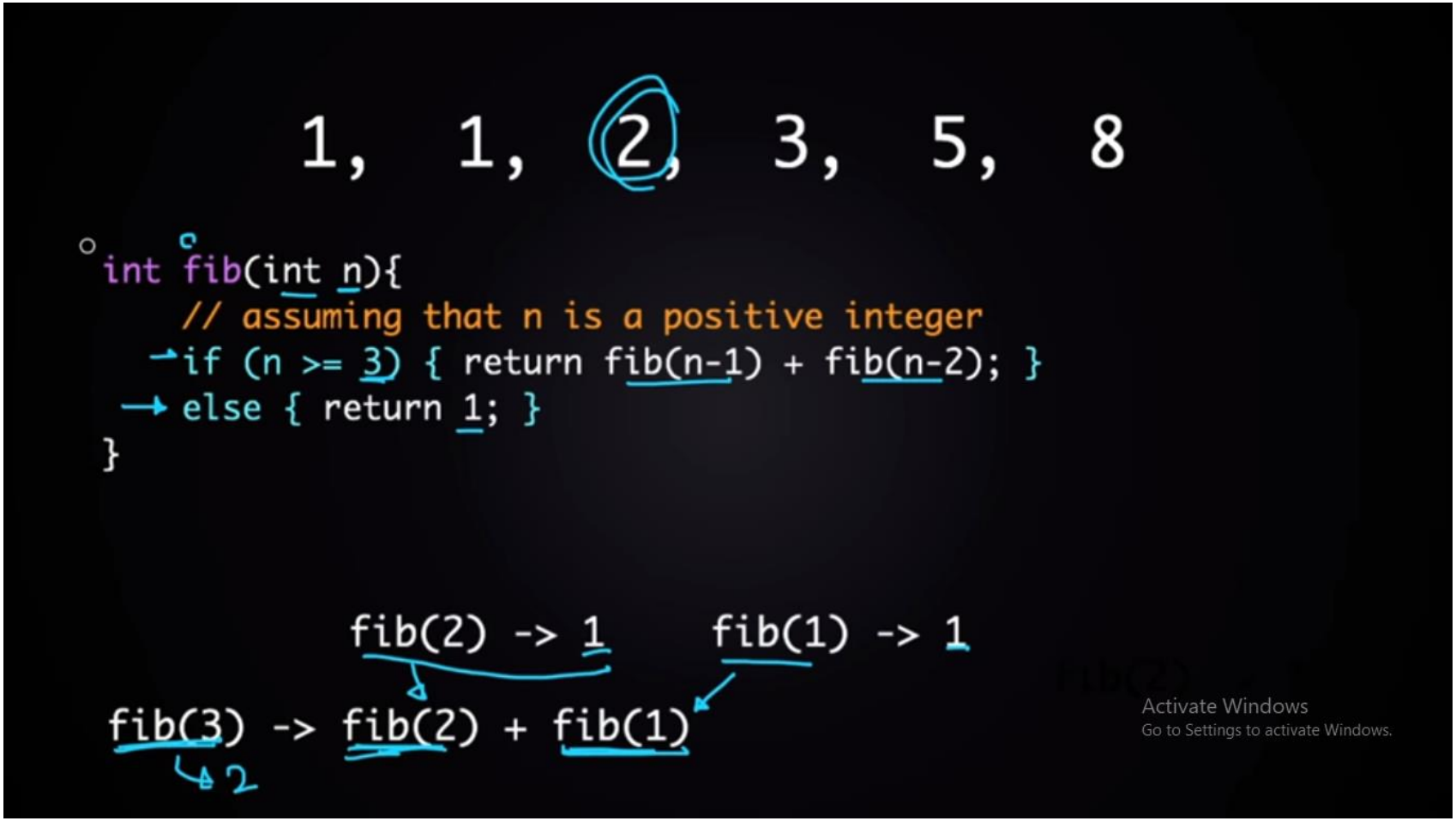
What Is The Fibonacci Sequence?

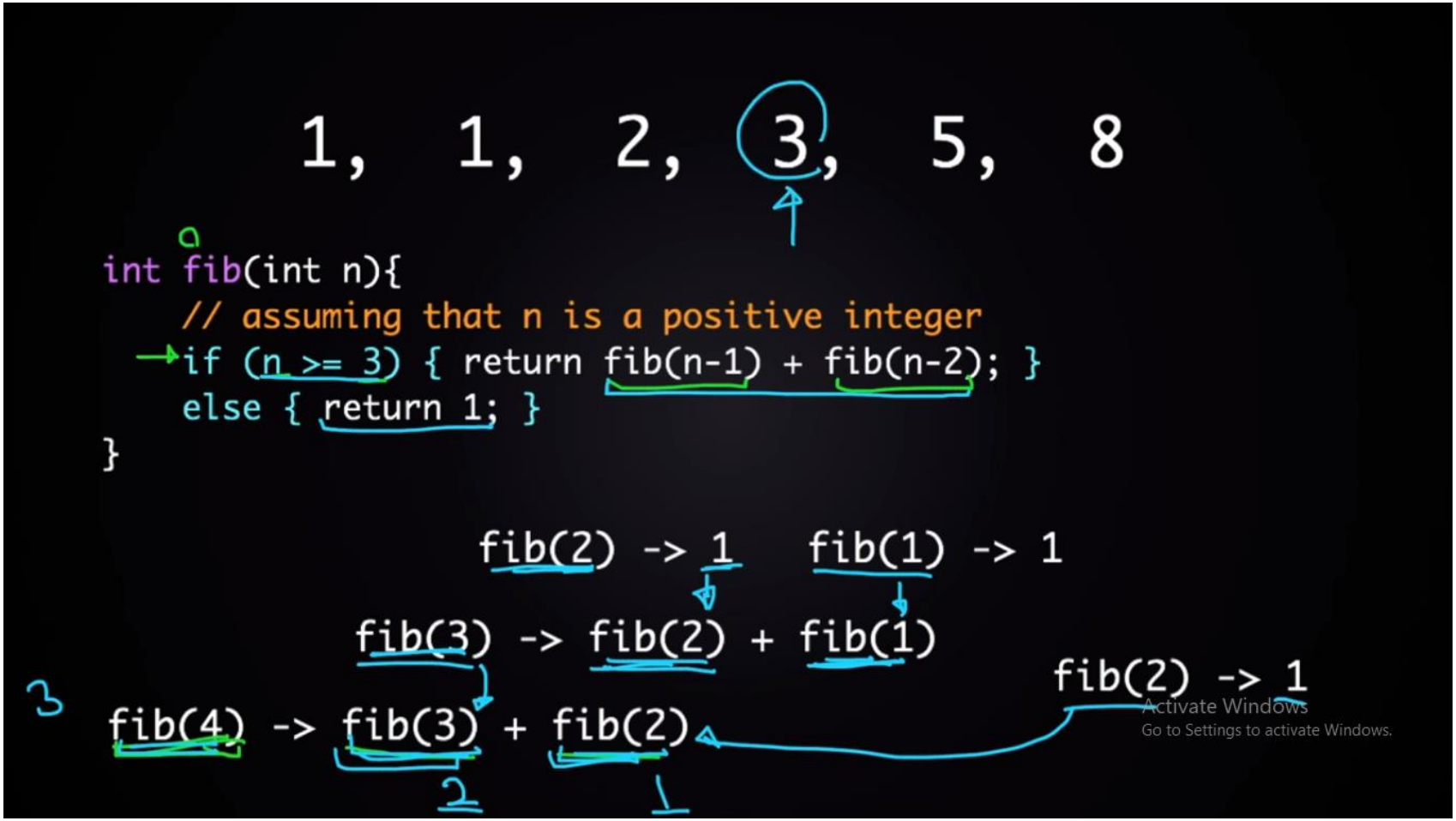
The Fibonacci sequence is a series of numbers where a number is the addition of the last two numbers, starting with 0, and 1.

The Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Written as a rule, the expression is:

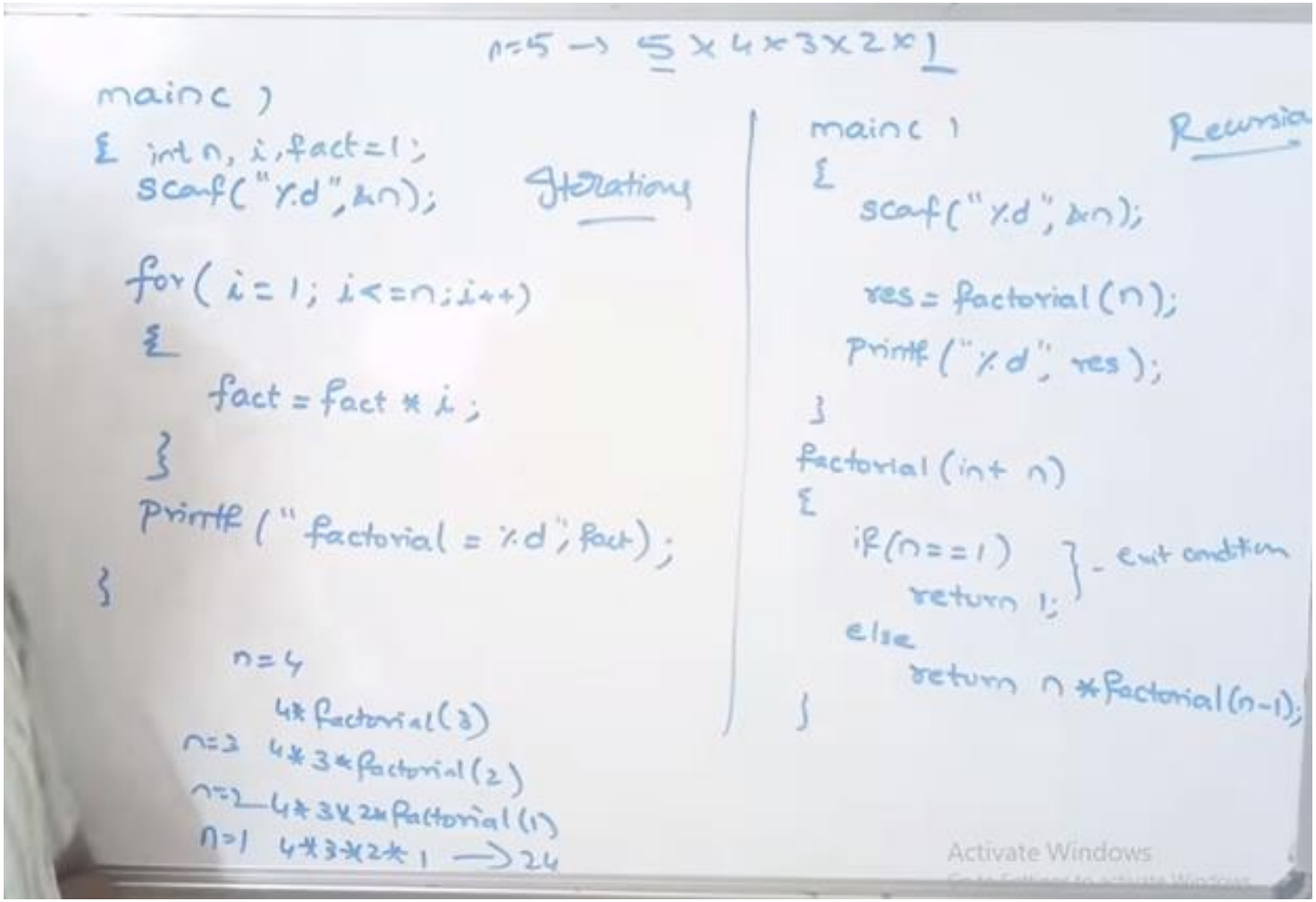
$$X_n = X_{n-1} + X_{n-2}$$




1, 1, 2, 3, 5, 8
int fib(int n){
// assuming that n is a positive integer
if (n >= 3) { return fib(n-1) + fib(n-2); }
else { return 1; }
}
fib(2) -> 1
fib(1) -> 1
fib(3) -> fib(2) + fib(1)
fib(2) -> 1
fib(4) -> fib(3) + fib(2)
Activate Windows
Go to Settings to activate Windows.

```
# Function for nth Fibonacci number
def Fibonacci(n):

    # Check if input is 0 then it will
    # print incorrect input
    if n < 0:
        print("Incorrect input")

    # Check if n is 0
    # then it will return 0
    elif n == 0:
        return 0

    # Check if n is 1,2
    # it will return 1
    elif n == 1 or n == 2:
        return 1

    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

# Driver Program
print(Fibonacci(9))                    ////output -34(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

Factorial program using recursion and iteration



Stack Application: Expression conversion ,Evaluation using stack

Evaluation of arithmetic expressions by compilers

infix to postfix conversion

infix to prefix conversion

Postfix to prefix conversion

Prefix to postfix conversion

evaluation of postfix ,prefix expressions

prerequisite

- Expression
- Operand
- Operator
- Delimiter
- Precedence
- Associativity
- Representation of Infix,postfix,prefix
- Conversion rules,evaluation rules

Expression

- An algebraic expression is a legal combination of operands and the operators. a / b - c + d * e - a * c

Operand

- Operand is the quantity (unit of data) on which a mathematical operation is performed.
  - Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.

Operator

- Operator is a symbol which signifies a mathematical or logical operation between the operands.
- Example of familiar operators include

+,-,*, /, ^

- Considering these definitions of operands and operators now we can write an example of expression as x+y*z.

Precedence

- When there are more than one operator in the expression, then the order of their evaluation is decided by the operator precedence over others. For eg, consider the expression a + b * c, here b*c should be evaluated first as the precedence of multiplication is higher than addition.

- 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.
- a + b * c --> a + (b * c)
- Here, multiplication operation has precedence over addition, b * c will be evaluated first.

Associativity

- The rule for operators having the same precedence order is described by their Associativity.
- For example, consider the expression a + b - c. Here, the precedence order of “+” and “-” is the same and both are left associative, therefore, (a+b) is evaluated first.
- Note: The above behavior can be altered using parentheses in the expression. For example, in the expression (a+b)*c, the a+b part of the expression will be evaluated first in spite of multiplication having higher precedence than addition.

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.

(a + b) * (c – d) / (e – f)

Precedence and Associativity

- Precedence and associativity determines the order of evaluation of an expression.
- 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.
- Use parenthesis for a + b to be evaluated first, like (a + b)*c

Precedence and Associativity

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

**Detailed table for precedence and associativity could be consulted from online resources.

Infix to Postfix

Infix: A + B * C

Postfix:

Order of operation

1) Parentheses ( ) { } [ ]
2) Exponents (right to left)
3) Multiplication and division (left to right)
4) Addition and Subtraction (left to right)

Example → Infix , Prefix and Postfix

- Expression → Infix

  <Operand> <Operator> <Operand>

- Expression → Prefix

  <Operator> <Operand> <Operand>

- Expression → Postfix

  <Operand> <Operand> <Operator>

- Infix to postfix expression operand and operator may change but order in which operand left to right will not change , the order of operator may change,this is an important observation in both infix and postfix forms

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + (B * C)	+ A * B C	A B C * +
(A + B) * C	* + ABC	A B + C *

Infix, Postfix and Prefix Expressions

- INFIX: From our schools times we have been familiar with the expressions in which operands surround the operator, e.g. x+y, 6*3 etc this way of writing the Expressions is called infix notation.
- POSTFIX: Postfix notation are also Known as Reverse Polish Notation (RPN). They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc.
- PREFIX: Prefix notation also Known as Polish notation.In the prefix notation, as the name only suggests, operator comes before the operands, e.g. +xy, *+xyz etc.

Infix Notations(Demerits)

- Easy for Human
- Difficult for computer devices
- Costly in terms of time and space consumption
- While we use infix expressions in our day to day lives. Computers have trouble understanding this format because they need to keep in mind rules of operator precedence and also brackets.
- Prefix and Postfix expressions are easier for a computer to understand and evaluate.

Infix, Postfix, Prefix

Postfix

(Reverse polish notation)

<operand><operand><operator>

Infix	Prefix	Postfix
2+3	+2 3	2 3+
p-q	-pq	pq-
a+b*c	+a*bc	abc*+

Infix (a+b*c) ⇑ human-readable

Prefix and Postfix ⇑ Good for machines

a+(b*c)

⇓

a+(bc*)

⇓

a(bc*)+

⇓

abc*+

postfix

Postfix syntax was proposed in 1950 by some computer scientists.in postfix notation operator is placed after operands

- Programmatically, postfix expression is easiest to parse and least costly in terms of time and memory to evaluate and that's why this was actually invented
- Prefix expression can also be evaluated in similar time and memory

Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators,but compiler doesn't have integrated order of operators.
- Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very in-efficient.
- To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.

Evaluate expression without using stack- compiler scan the input multiple times

- The compiler scans the given expression either from left to right or from right to left.
  - **a + b / c - d**
- At first, compiler scans the expression to evaluate the expression **b / c**
- then scans the expression again to add **a** to it.
- The result is then subtracted with **d** after another scan.
- This repeated scanning makes the process very inefficient and time consuming. It will be much easier if the expression is converted to prefix (or postfix) before evaluation.
- The corresponding expression in prefix form is: **-+a/bcd**. The prefix expressions can be easily evaluated using a stack.

Algorithm

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association.

If the association is left to right, pop and print the top of the stack and then push the incoming operator.

If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

• 8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Pseuodocode(infix to postfix)

```
Infix ToPostfix(exp)
{
Create a stack  S
res ←------ empty string
for i  ←---- 0 to length(exp) -1
{
if exp[i] is operand
        res ←--- res + exp[i]
```

```
else if exp[i] is operator
{
While ( ! S.empty() && HasHigherPrec(s.top(),exp[i])
{
res ←--- res+ S.top()
S.pop()
}
s.push(exp[i])
}}
While(!S.empty())
{
res←----res + s.top()
S.pop()
}
return res
}
```

Examples

K+L-M*N+(O^P)*W/U/V*T+Q

Input Expression	Stack							Postfix Expression
K								K
+	+							K
L	+							KL
-	-							KL+
M	-							KL+M
*	-	*						KL+M
N	-	*						KL+MN
+	+							KL+MN*-
(	+	(						KL+MN*-
O	+	(						KL+MN*-O
^	+	(	^					KL+MN*-O
P	+	(	^					KL+MN*-OP
)	+							KL+MN*-OP^

K+L-M*N=(O^P)*W/U/V*T+Q

INPUT EXPRESSIO	STACK	POSTFIX EXPRESSION
*	+*	KL+M*-0P^
W	+*	KL+M*-0P^W
/	+/	KL+M*-0P^W*
U	+/	KL+M*-0P^W*U
/	+/	KL+M*-0P^W*U/
V	+/	KL+M*-0P^W*U/V
*	+*	KL+M*-0P^W*U/V/
T	+*	KL+M*-0P^W*U/V/T
+	+	KL+M*-0P^W*U/V/T*+
Q		KL+M*-0P^W*U/V/T*+Q
		KL+M*-0P^W*U/V/T*+Q+

EvaluatePostfix

Important: infix to postfix conversion operator to push onto the stack whereas postfix evaluation operand push onto the stack

```
EvaluatePostfix(exp)
{
Create a stack S
For i←--- 0 to length(exp)-1
{
If(exp[i] is operand)
        push(exp[i])
```

- else if (exp[i] is operator)
- {
- Op2←----pop()
- Op1←---pop()
- res←--- perform(exp[i],op1,op2)
- push(res)
- }
- }return top of stack
- }

2 3*5 4 * +9-

Input expression	stack	Res(postfix evaluaton expression)
2	2	
3	2 3	
*	6	Op1=2,op2=3 (op1 *OP2)=6
5	6 5	
4	6 5 4	
*	6 20	Op1=5,op2=4 (op1 *OP2)=20
+	26	Op1=6,op2=20 (op1 *OP2)=26
9	26 9	
-		Op1=26,op2=9 (op1 *OP2)=17

Postfix Expression Evaluation

Infix →a+b*c-d/e^f

Postfix →abc*+def^/-

234*+16 2 3^/-

<Operand> <Operand> <Operator>

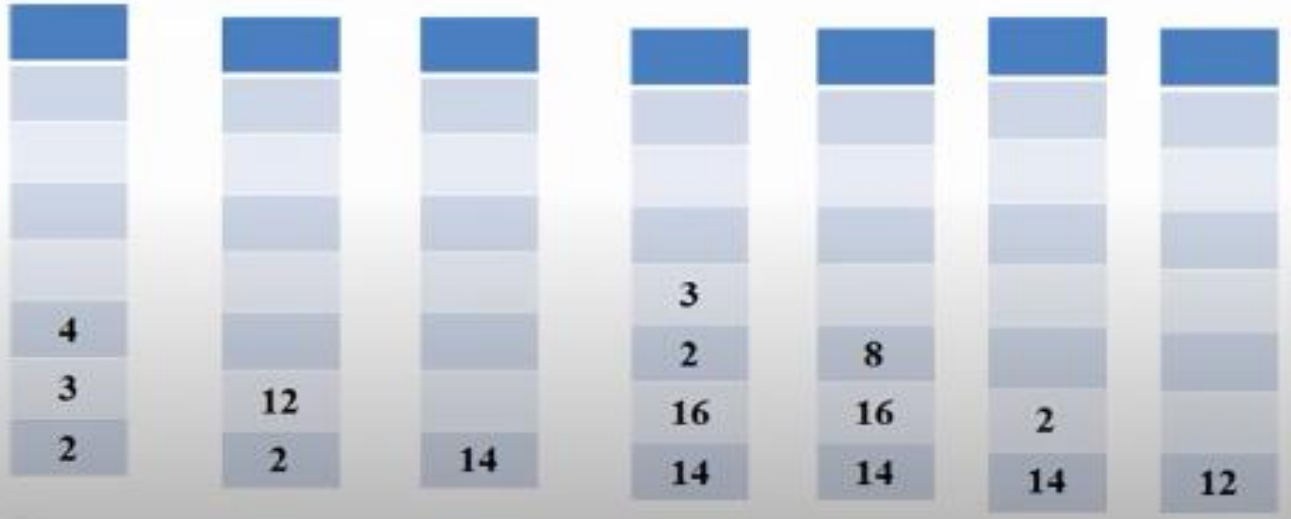
234*+16 2 3^/-
2 12 + 16 2 3 ^ /-
14 16 2 3 ^ / -
14 16 8 / -
14 2 -
12

Postfix Expression Evaluation using Stack

Infix →a+b*c-d/e^f

Postfix →abc*+def^/-

234*+16 2 3^/-



- infix expression **((A * (B + D)/E) – (F * (G + H / K)))** to convert into its equivalent postfix expression

Label No.	Symbol Scanned	Stack	Expression
1	(	(	
2	(	((	
3	A	((	A
4	*	((*	A
5	(	((*(	A
6	B	((*(	AB
7	+	((*(+	AB

((A * (B + D)/E) –(F * (G + H / K)))			
Label No.	Symbol Scanned	Stack	Expression
8	D	((*(+	ABD
9	)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12	)	(	ABD+E/*
13	-	(-	ABD+E/*
14	(	(-(	ABD+E/*
15	F	(-(	ABD+E/*F

((A * (B + D)/E) –(F * (G + H / K)))			
Label No.	Symbol Scanned	Stack	Expression
16	*	(-(*	ABD+E/*F
17	(	(-(*(	ABD+E/*F
18	G	(-(*(	ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH
21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23	)	(-(*	ABD+E/*FGHK/+
24	)	(-	ABD+E/*FGHK/+*
25	)		ABD+E/*FGHK/+*-

Python program to convert infix expression to postfix

```
# Class to convert the expression
class Conversion:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

```

```
    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"
```

```
    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

    # A utility function to check is the given character
    # is operand
    def isOperand(self, ch):
        return ch.isalpha()

    # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False
```

```
def infixToPostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack
            elif i  == '(':
                self.push(i)
```

```
# If the scanned character is an ')', pop and
        # output from the stack until and '(' is found
        elif i == ')':

            while( (not self.isEmpty()) and
                        self.peek() != '('):
                a = self.pop()
                self.output.append(a)
            if (not self.isEmpty() and self.peek() != '('):
                return -1
            else:
                self.pop()
```

```
    # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

        print "".join(self.output)

# Driver program to test above function
exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

Output:

abcd^e-fgh*+^*+i-

Algorithm to Convert Infix To Prefix

- Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent prefix expression Y.

1. Reverse the infix expression.
2. Make Every " (" as ") " and every ") " as " ("
3. Push " (" onto Stack, and add ") " to the end of X.
4. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
5. If an operand is encountered, add it to Y.
6. If a left parenthesis is encountered, push it onto Stack.

7.If an operator is encountered ,then :

- Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
- Add operator to Stack.
  [End of If]

8.If a right parenthesis is encountered ,then :

- Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
- Remove the left Parenthesis.
  [End of If]
  [End of If]

9.Reverse the prefix expression.

10END.

- **Example**
- Infix Expression : A + (B * C - (D / E ^ F) * G) * H
- Reverse the infix expression :
  H *) G *) F ^ E / D (- C * B (+ A
- Make Every “ (” as “) ” and every “) ” as “ (”
  H * (G * (F ^ E / D) - C * B) + A
- Output:Reverse the postfix expression :
  +A*-*BC*/D^EFGH

Convert expression to prefix form

(H * (G * (F ^ E / D) - C * B) + A)

:

Scanned	Stack	Prefix Expression	Description
	(		Start
H	(	H	
*	(*	H	
(	(*(	H	
G	(*(	HG	
*	(*(*	HG	
(	(*(*(	HG	
F	(*(*(	HGF	

(H * (G * (F ^ E / D) - C * B) + A)

^	(*(*(^	HGF	
E	(*(*(^	HGFE	
/	(*(*(/	HGFE^	' ^ ' is at highest precedence then ' / '
D	(*(*(/	HGFE^D	
)	(*(*	HGFE^D/	
-	(*(-	HGFE^D/*	
C	(*(-	HGFE^D/*C	
*	(*(-*	HGFE^D/*C	
B	(*(-*	HGFE^D/*CB	
)	(*	HGFE^D/*CB*-	POP from top on Stack, that's why ' * ' come first
+	(+	HGFE^D/*CB*-*	' * ' is at highest precedence then ' + '
A	(+	HGFE^D/*CB*-*A	
)	Empty	HGFE^D/*CB*-*A+	END

Evaluation of Prefix expression

- Input: * 3 / + 6 4 5
  Output: 6

- Algorithm to Evaluate Prefix Expression

1.Steps to evaluate prefix expression is same as evaluation of postfix expression with one additional step i.e. step 1.

2.Reverse the prefix expression.

3.Scan the reversed prefix expression from left to right.

4.If the scanned character is an operand, then push it to the stack.

5.Else if the scanned character is an operator, pop two operands from the top of the stack and evaluate them with the scanned operator and push the result into the stack. (Note: here evaluation should be **first_pop_operand operator second_pop_operand**.)

6.Repeat steps 2 and 3 until all the characters in reversed prefix expression are scanned.

7.Print the final result from the top of the stack.

Step by step evaluation of prefix expression “* 3 / + 6 4 5” is shown below in the table. In step 1, the prefix expression is reversed and hence the new expression becomes “5 4 6 + / 3 *

Steps	Scanned Character	Stack	Action
Initially		[Empty Stack]	
1	5	5	Push 5
2	4	5,4	Push 4
3	6	5,4,6	Push 6
4	+	5,10	Pop 6, Pop 4 and Push 6+4 = 10
5	/	2	Pop 10, Pop 5 and Push 10/5 = 30
6	3	2,3	Push 3
7	*	6	Pop 3, Pop 2 and Push 3*2 = 6
Final Result:-		6	

Postfix :second_pop_operand operator first_pop_operand.)”-”

	5-4=1
4	first_pop_operand
5	second_pop_operand

- Prefix: **first_pop_operand operator second_pop_operand.)** “-” 4-5=-1

4	first_pop_operand
5	second_pop_operand

convert Postfix to Prefix

- The following converter converts an postfix expression to a prefix expression.
- **Change the expression and converter will convert postfix to prefix step by step.**

1.Scan the given postfix expression from **left to right** character by character.

2.If the character is an operand, push it into the stack.

3.But if the character is an operator, pop the top two values from stack.

4.Concatenate this operator with these two values (**operator+2nd top value+1st top value**) to get a new string.

4.Now push this resulting string back into the stack.

5.Repeat this process untill the end of postfix expression. Now the value in the stack is the desired prefix expression.

Postfix: ABC/-AK/L-*
Prefix: *-A/BC-/AKL

Input String	Postfix Expression	Stack (Prefix)
ABC/-AK/L-*	BC/-AK/L-*	A
ABC/-AK/L-*	C/-AK/L-*	AB
ABC/-AK/L-*	/-AK/L-*	ABC
ABC/-AK/L-*	-AK/L-*	A/BC
ABC/-AK/L-*	AK/L-*	-A/BC

ABC/-AK/L-*	K/L-*	-A/BCA
ABC/-AK/L-*	/L-*	-A/BCAK
ABC/-AK/L-*	L-*	-A/BC/AK
ABC/-AK/L-*	-*	-A/BC/AKL
ABC/-AK/L-*	*	-A/BC-/AKL
ABC/-AK/L-*		*-A/BC-/AKL

Prefix to Postfix-steps

- **Change the expression and converter will convert prefix to postfix step by step.**
- 1.Scan the given prefix expression from right to left character by character.
- 2.If the character is an operand, push it into the stack

Prefix: *-A/BC-/AKL
Postfix: ABC/-AK/L-*

3.But if the character is an operator, pop the top two values from stack.

4.Concatenate this operator with these two values (**operator+1st top value+2nd top value**) to get a new string.

5.Now push this resulting string back into the stack.

6.Repeat this process untill the end of prefix expression. Now the value in the stack is the desired postfix expression.

Input String	Prefix Expression	Stack (Postfix)
*-A/BC-/AKL	*-A/BC-/AKL	
*-A/BC-/AKL	*-A/BC-/AK	L
*-A/BC-/AKL	*-A/BC-/A	LK
*-A/BC-/AKL	*-A/BC-/	LKA
*-A/BC-/AKL	*-A/BC-	LAK/
*-A/BC-/AKL	*-A/BC	AK/L-
*-A/BC-/AKL	*-A/B	AK/L-C

Input String	Prefix Expression	Stack (Postfix)
A/BC-/AKL	*-A/	AK/L-CB
*-A/BC-/AKL	*-A	AK/L-BC/
*-A/BC-/AKL	*-	AK/L-BC/A
*-A/BC-/AKL	*	AK/L-ABC/-
*-A/BC-/AKL		ABC/-AK/L-*