Data Structure

Array

(Ordered, Fast Access)

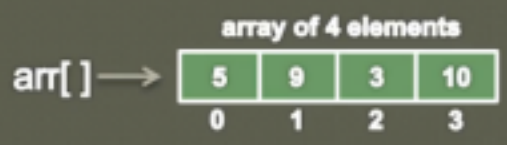
- An array is a collection of similar type of elements which has sorted contiguous memory location
- We can store only a fixed set of elements in a Java array
- Access a particular element = Array_Name[index]
- Random access takes constant time
- Types of array: 1-D, 2-D and multi dimensional array

What is an Array?

Chocolate box

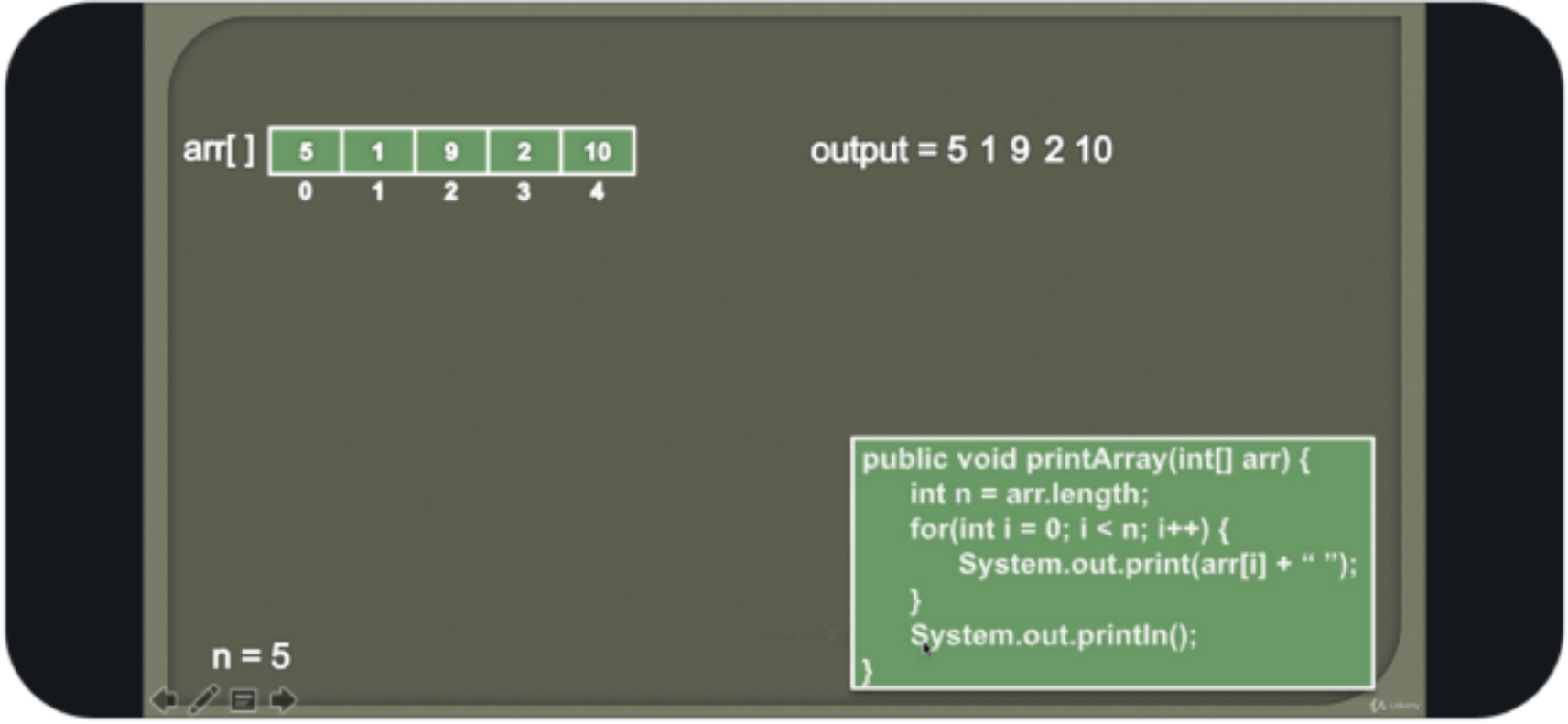
1. It's a collection (box) of chocolates.
2. All chocolate holding partitions are adjacent or contiguous.
3. Each partition has two neighbors except first and last one.
4. Size of box is fixed and cannot be modified.
5. Being adjacent each partition is indexed and can be determined by its position.

What is an Array?

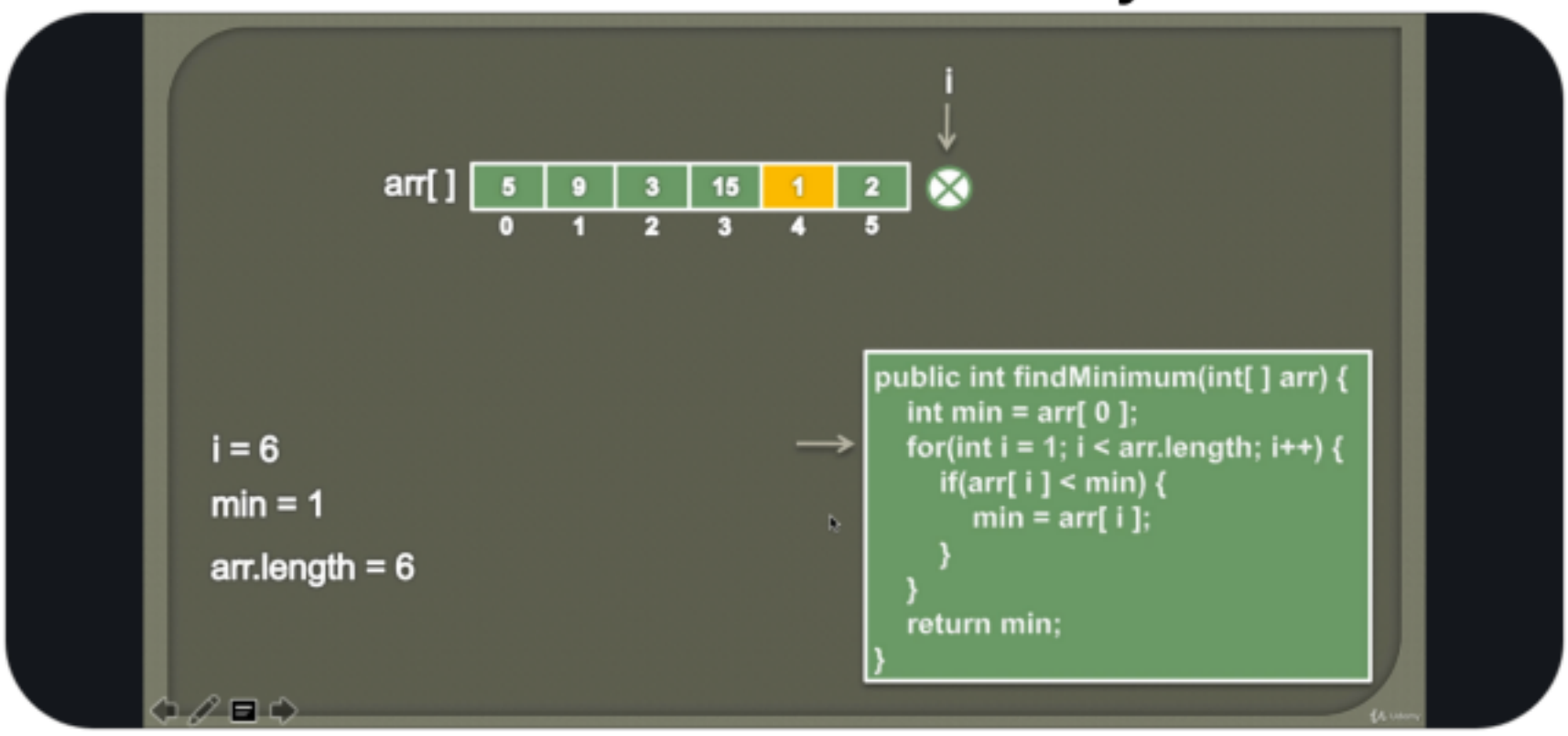


1. It's a collection (box) of data elements of specified type.
2. All data holding partitions have contiguous memory locations.
3. Each partition has two neighbors except first and last one.
4. Size of array is fixed and cannot be modified once it is created.
5. Being adjacent each partition is indexed and can be determined by its position.
6. Index starts at 0 and for (one dimensional array) ends at length - 1

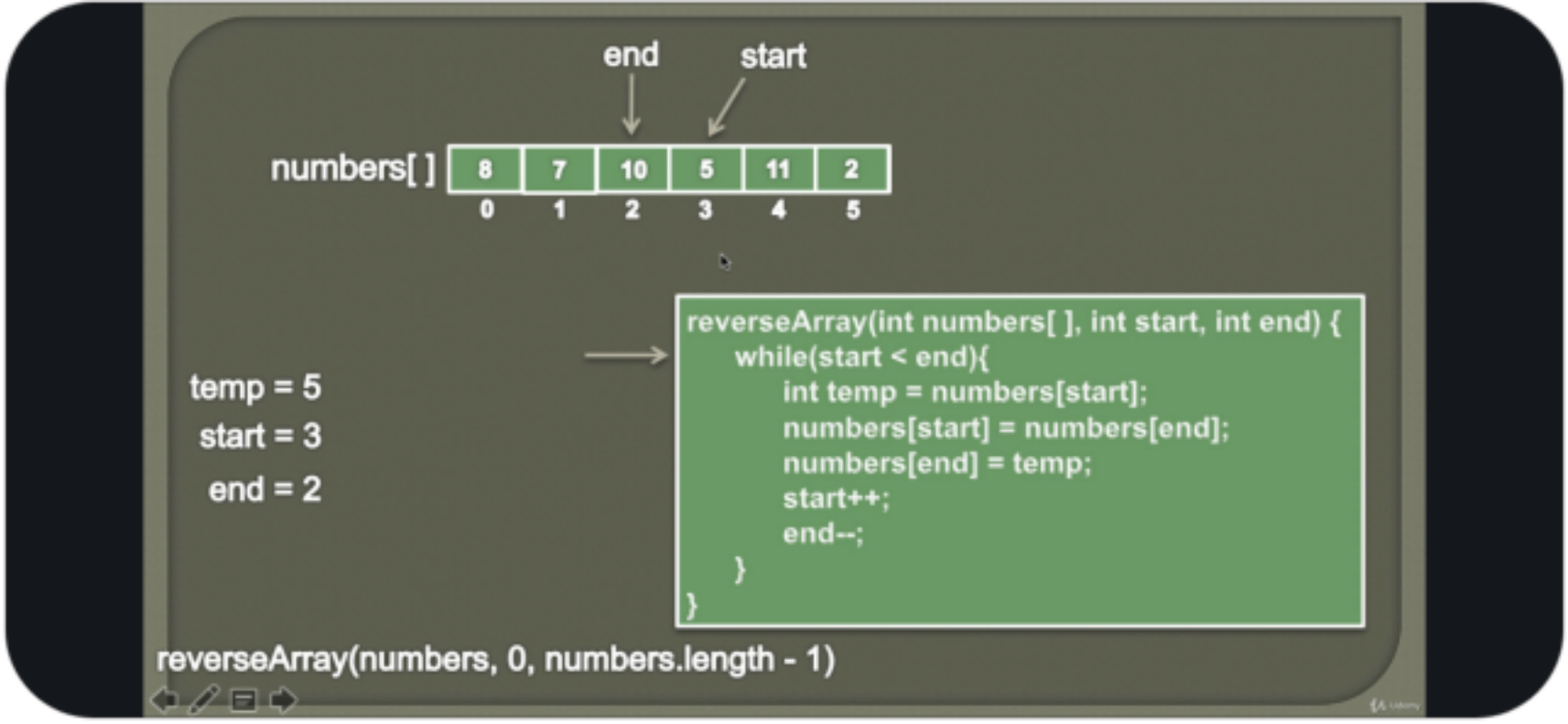
How to loop an array:



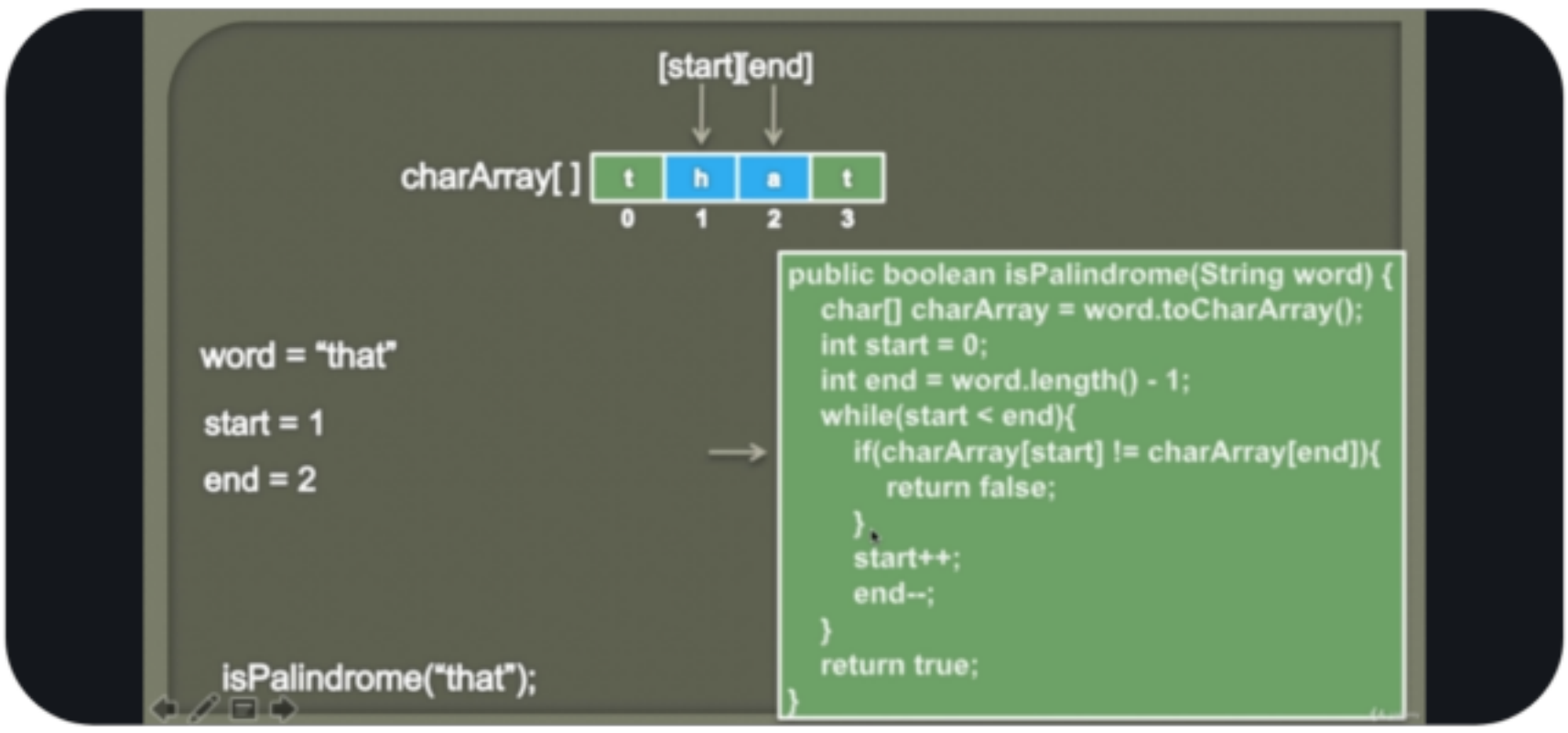
How to find min value in array:



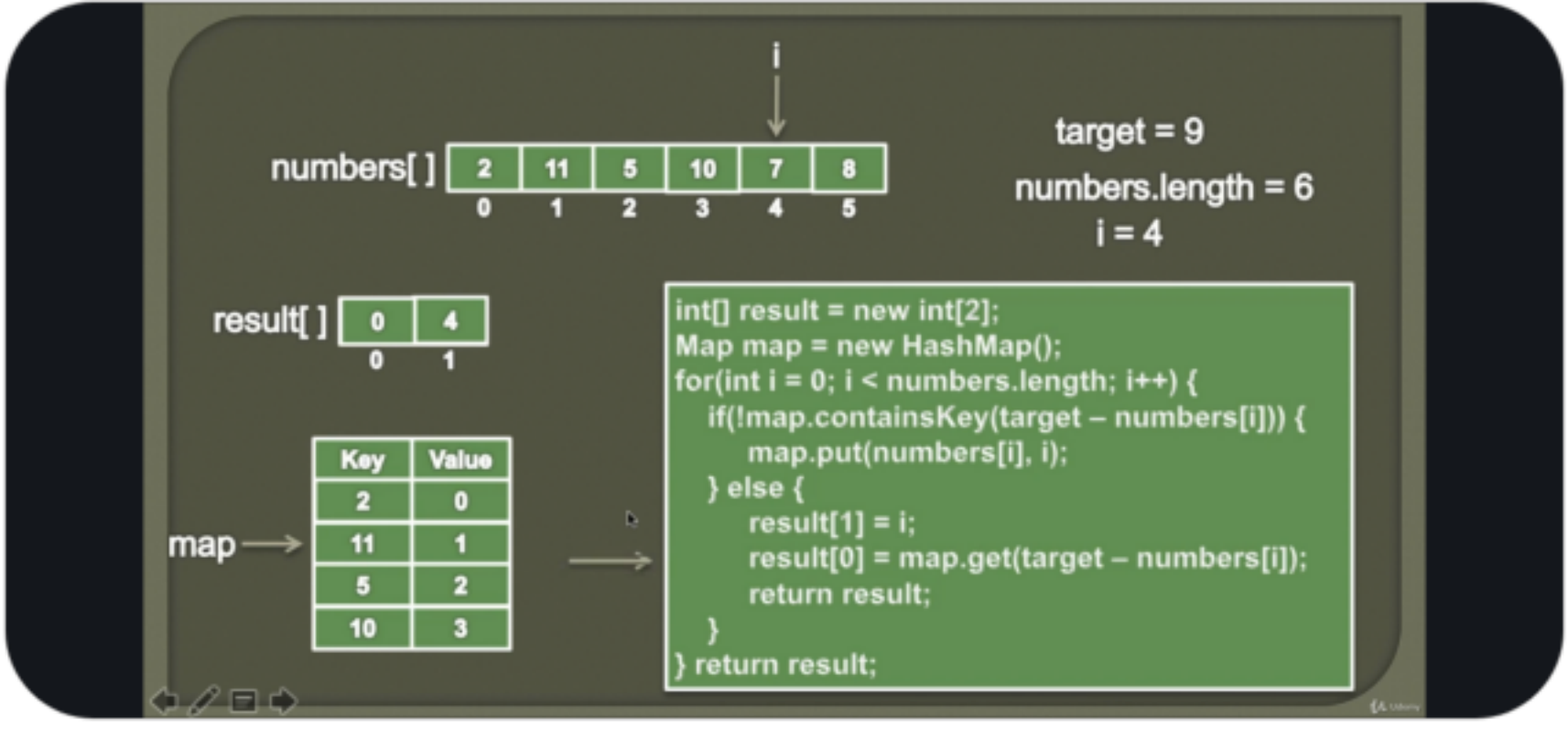
How to reverse an array



How to check if a string is Palindrome or not:



How to solve two sum problem



- **Declaration**: int[] marks;
- **Memory Allocation**: marks = new int[5]; - 5×4= 20 bytes continuous memory space
- **Initialization**: marks = {30, 40, 50, 60, 70}
- All three in one line: Int[] intValues = {1, 2, 3, 6,10}

2-D Array:

- int[][] flats = new int[2][3];
- A 2-D array is called as Jagged array if all columns are of

different size

- int[][] numbers = new int[2][]; - Here we dont specify the size column. User can create column of different sizes

Time Complexities

- Access / Update - O(1)
- Insert / Delete - O(n)
- Search - O(n)
- Binary Search - O(logn)

Methods

- length
- intValues[1]; To get a value
- intValues[2] = 5;To update a value
- new ArrayList<>(Arrays.asList(array))
- Collections.reverse(Arrays.asList(array))
- int[] a1 = {1,2,3}; int[] a2 = a1.clone();

- To merge 2 arrays:
  System.arraycopy(arr1, 0, resultArray, arr1.length);
  System.arraycopy(arr2, 0, resultArray, arr2.length);
- Arrays.sort(arr);
- Arrays.binarySearch(arr, key); -> returns index of key if it is found else returns (-(insertion_point) - 1).

Problems: Find minimum / maximum number in a array, find duplicate element, reverse of an array, find 2nd largest element

Array List

(Ordered, Fast access)

- ArrayList internally uses a dynamic array to store the elements
- It is like an array, but there is no

size limit. We can add or remove elements anytime.

- ≡ ○ The ArrayList in Java can have the duplicate elements also.
- ≡ ○ The ArrayList maintains the insertion order internally.
- ≡ ○ Java ArrayList class is non synchronized.
- ≡ ○ Java ArrayList allows random access because array works at the index basis.
- ≡ ○ In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Time complexities

- ≡ ○ Access - O(1)
- ≡ ○ Insert at the end - O(1)
- ≡ ○ Insert at the index - O(n)
- ≡ ○ Remove - O(n)

Methods

- add(E e)
- add(int index, E e)
- addAll(Collection)
- clear()
- isEmpty()
- get(int index)
- set(index, value)
- iterator
- listIterator
- lastIndexOf(Object o) / indexOf()
- toArray(new Integer[5]);
- contains(Object o)
- remove(int index)
- remove(list.size() - 1);
- remove(Object o)
- removeAll(collection)
- size()
- subList(int start, int end)
- List<Integer> v1 = new ArrayList<>(Arrays.asList({1,2,3}))

- List<Integer> v2 = new ArrayList<>(v1);
- Collections.sort(v1);
- Collections.reverse(list);
- int[] a1 = v1.toArray();
- Collections.binarySearch(list, key); -> returns index of key if it is found else returns (-(insertion_point) - 1).

String

- Its a non primitive data
- Strings are immutable in java which cant be changes once initialized
- String s = "Prasanna"; - By literals which creates new string in string pool area
- String s = new String("Prasanna"); - By java object which creates a

new string in both string pool area and Heap area

Methods

- char charAt(index)
- int length() - counts empty space
- toCharArray() - string to char array
- new String(char[]) - char array to string
- contains("Prasanna");
- s1.equals(s2);
- isEmpty();
- s1.concat(s2);
- s1.replace(oldChar, newChar);
- String[] split(regex); - to split by space or coma etc
- s1.toLowerCase() or s1.toUpperCase
- s1.trim();
- s1.indexOf(Str2) -> -1 or index of first occurance

- substring(startIndex)
- substring(startIndex, endIndex)
- new StringBuilder(str.length())
- StringBuilder.append(str)
- StringBuilder.toString()
- Integer.parseInt("100"); - String to int
- String.valueOf(10); - int to String

Character methods

- Character.isDigit(char a);
- Character.isLowerCase(char a);

Linked List

(Fast Manipulation)

- LinkedList internally uses a doubly linked list to store the elements
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains

insertion order.

- ○ Java LinkedList class is non synchronized.
- ○ In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- ○ Java LinkedList class can be used as a list, stack or queue.

List related methods

- ○ add(value)
- ○ add(index, value)
- ○ addAll(Collection)
- ○ addFirst(value) / getFirst()
- ○ addLast(value) / getLast()
- ○ get(index)
- ○ remove(int index)
- ○ remove(Object o)
- ○ removeFirst() / removeLast()
- ○ indexOf(Object o)
- ○ clear()

- contains(Object o)
- size()
- isEmpty()
- iterator()
- iterator().next()
- iterator().hasNext()
- listIterator()
- listIterator.hasPrevious()
- lisIterator().previous()
- descendingIterator() - to traverse in reverse order
- descendingIterator().hasNext()
- descendingIterator().next()
- (list == null || list.isEmpty())
- list.toArray()
- new ArrayList<>(data)
- List<String> strings = new ArrayList<>()
- for(String s: strings){}

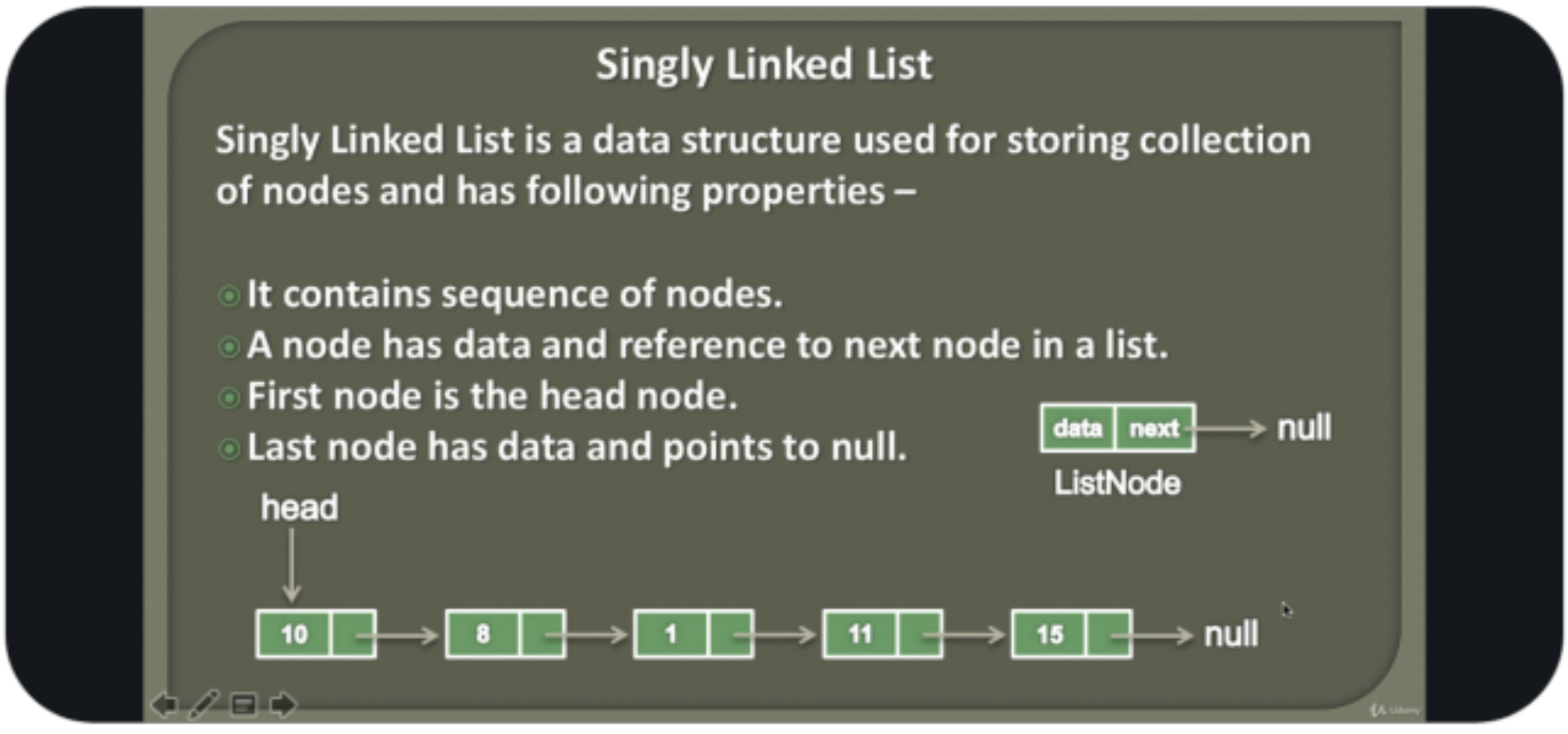
Queue related methods

- offer(E e) - adds at rear
- offerFirst(E e) - adds at front
- offerLast(E e) - adds at rear
- peek()
- peekFirst() - returns null if empty
- poll()
- pollFirst()
- pollLast()

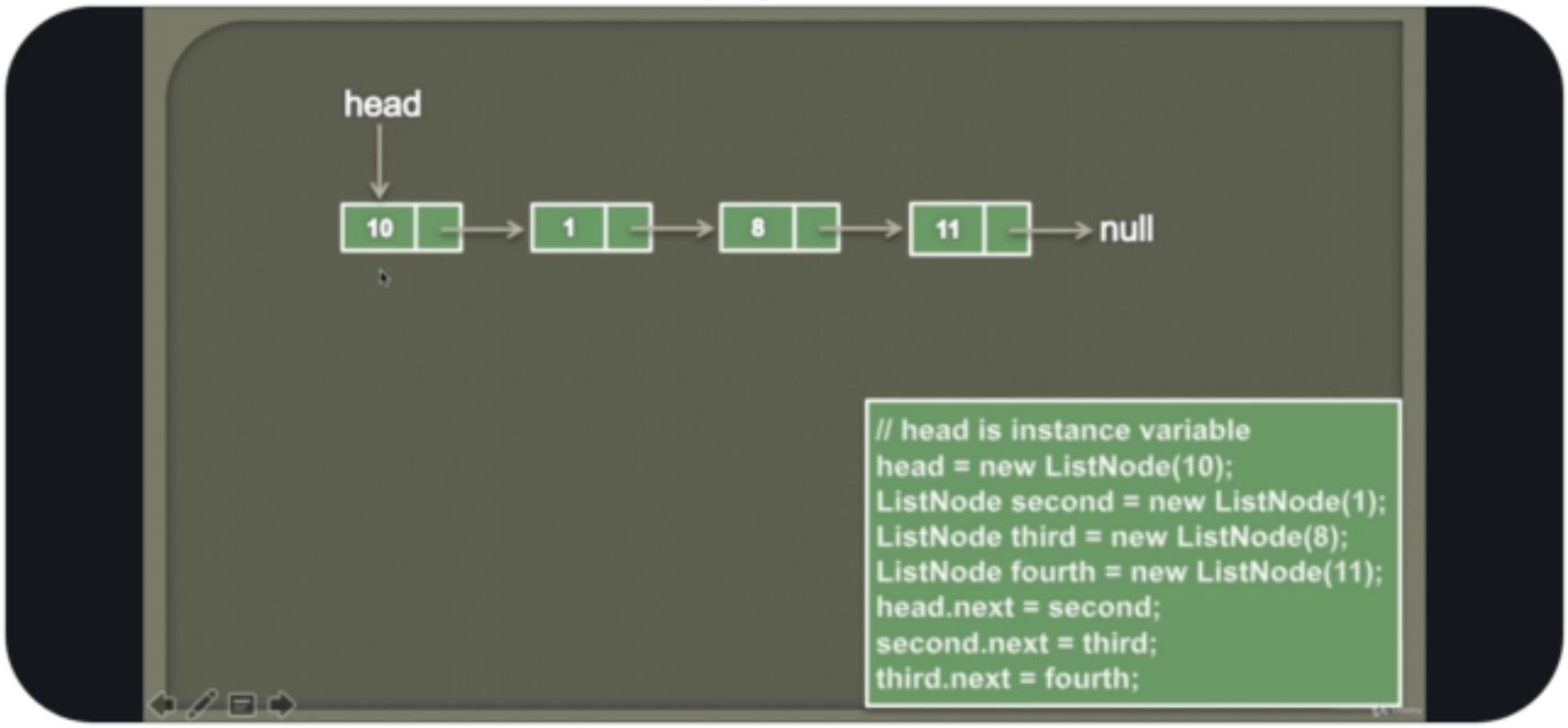
Stack related methods

- E pop()
- push(E e)

Singly Linked list



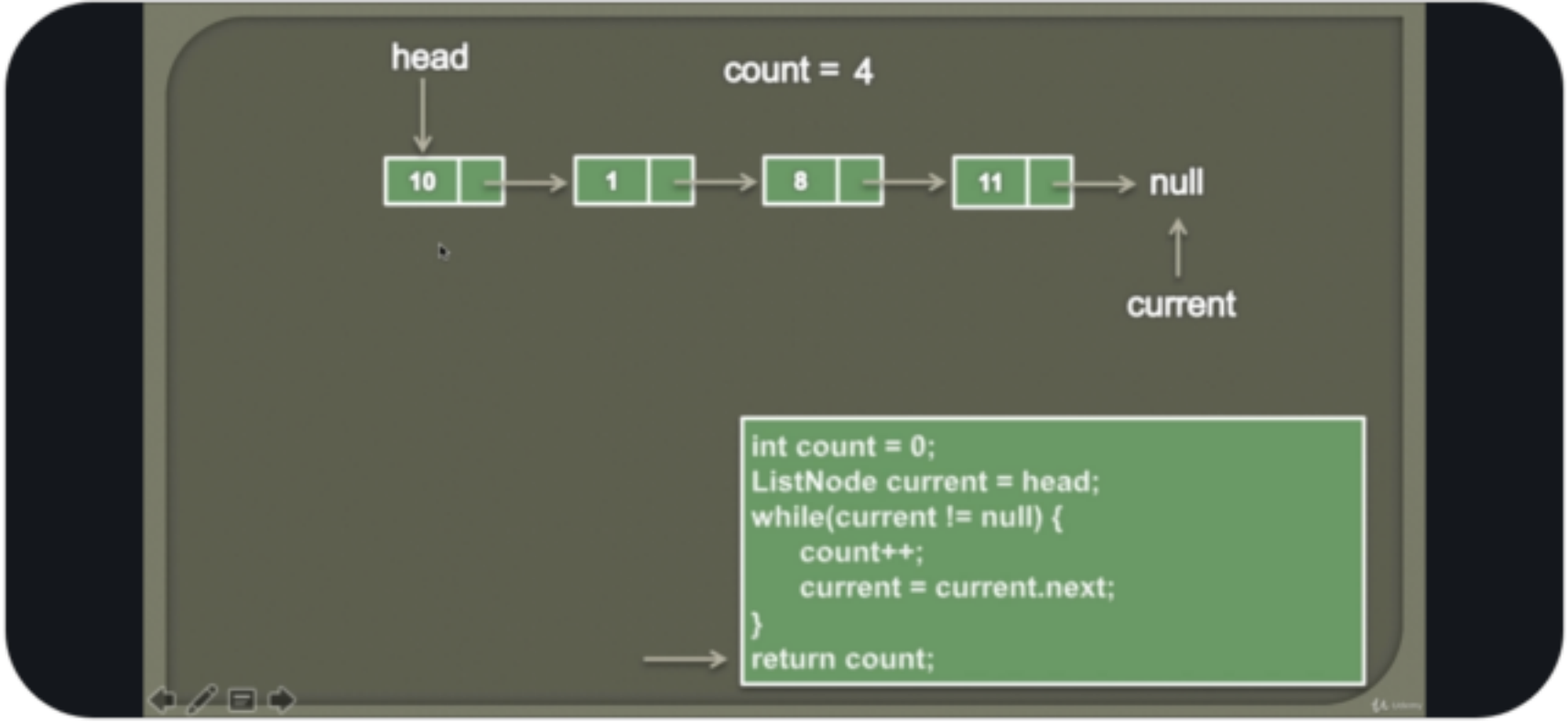
How to create singly linked list



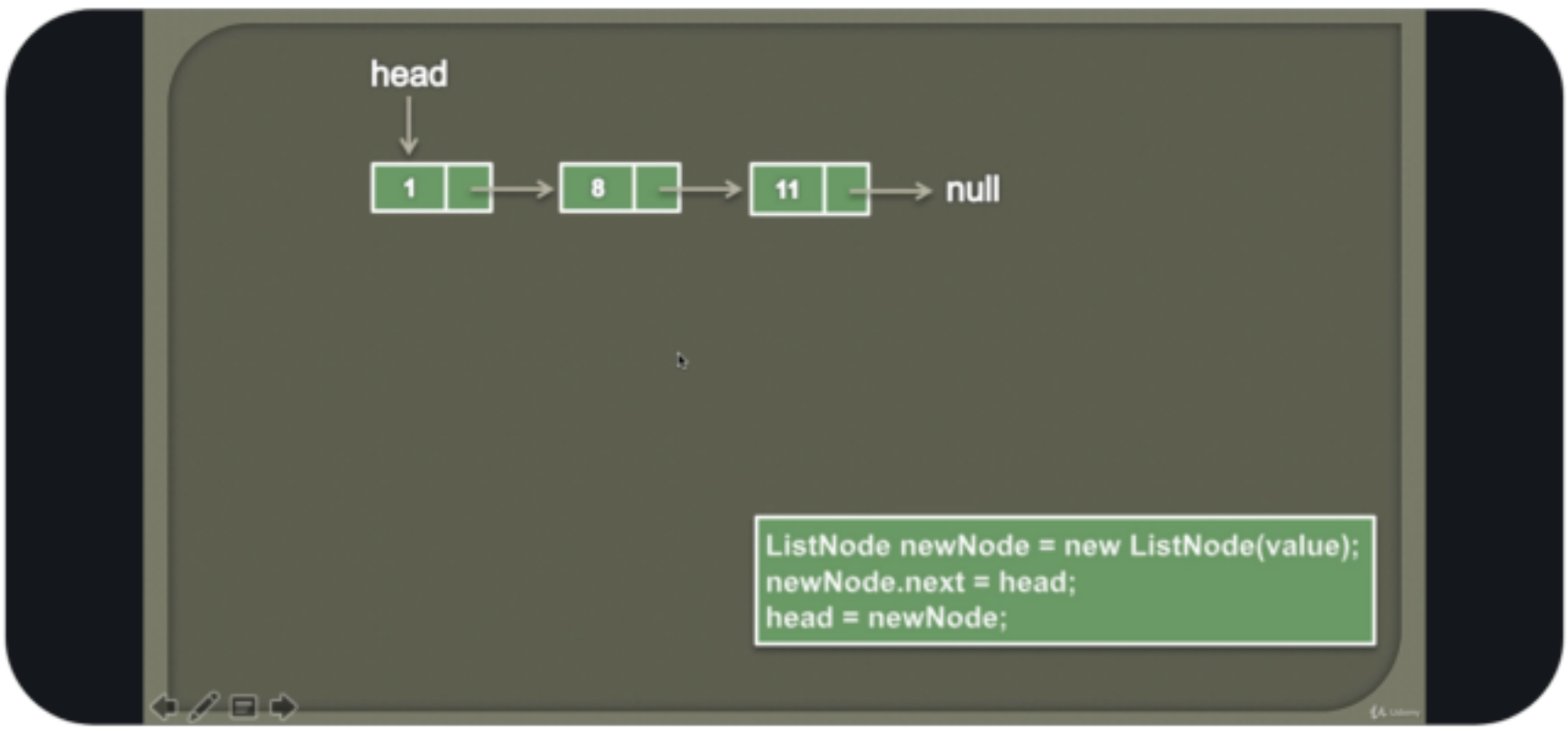
How to print elements of a singly linked list

```
public void display() {
    ListNode current = head;
    while(current != null) {
        System.out.print(current.data);
        current = current.next;
     }
     System.out.print("null");
}
```

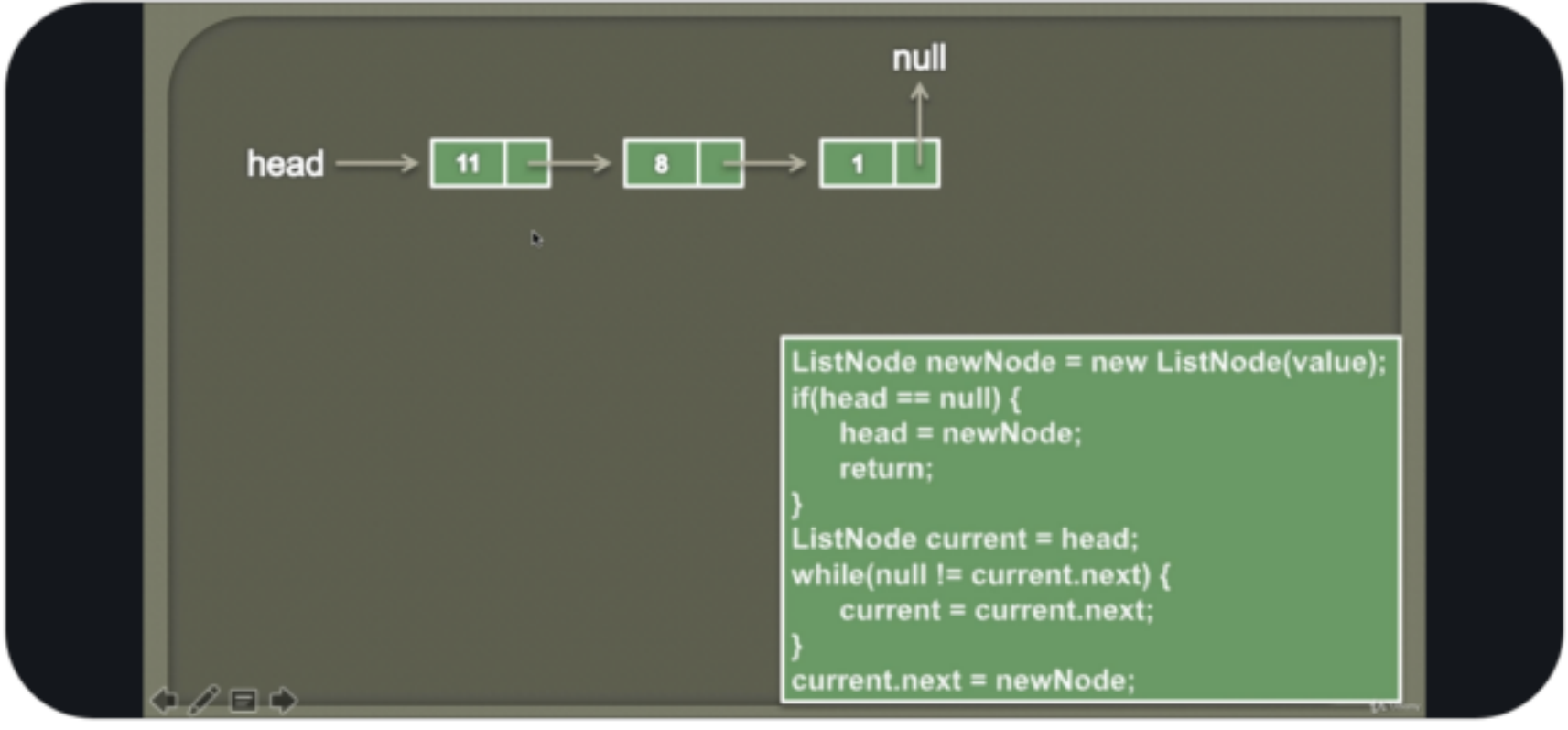
How to find length of a single linked list



How to insert node at the beginning of a linked list



How to insert a node at the end of a linked list

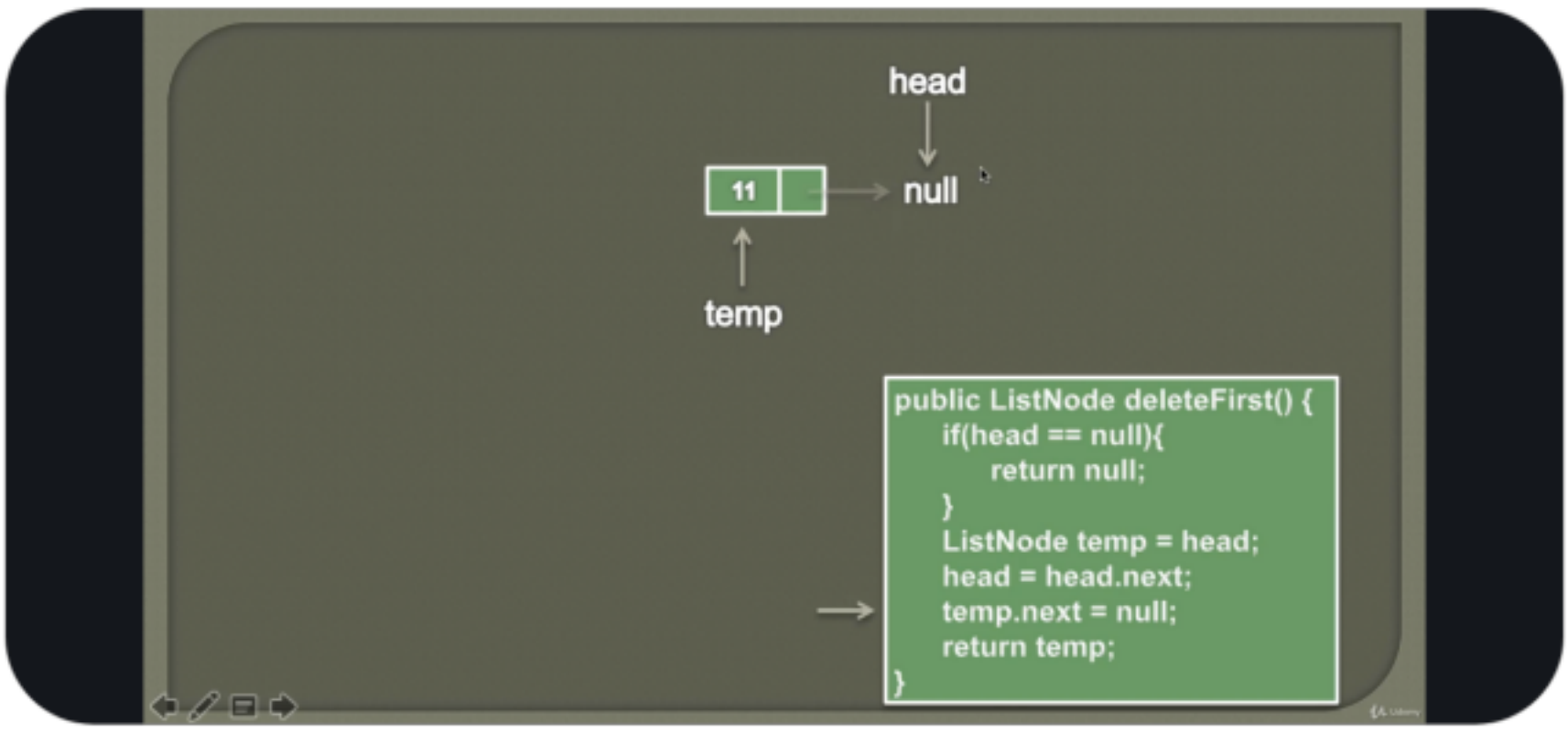


Insert a node at a given position

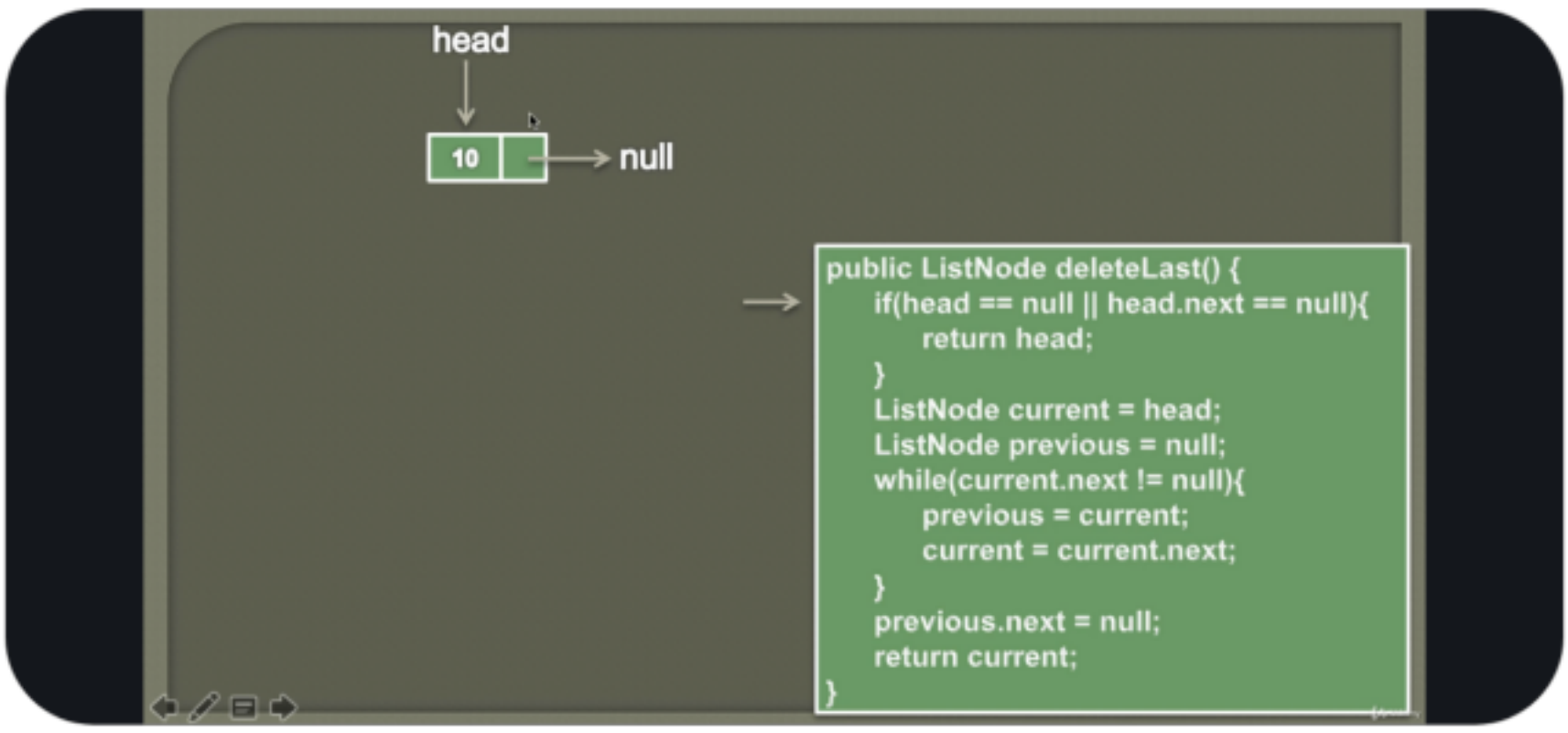
```
public void insertAtPosition(int data, int position) {
    ListNode newNode = new ListNode(data);
    if(position == 0){
        newNode.next = head;
        head = newNode;
    } else {
        ListNode previous = head;
        int count = 0;
        while(count < position - 1){
            previous = previous.next;
            count++;
        }
```

```
        ListNode current = previous.next;
        newNode.next = current;
        previous.next = newNode;
    }
}
```

How to delete first node



How to delete last node

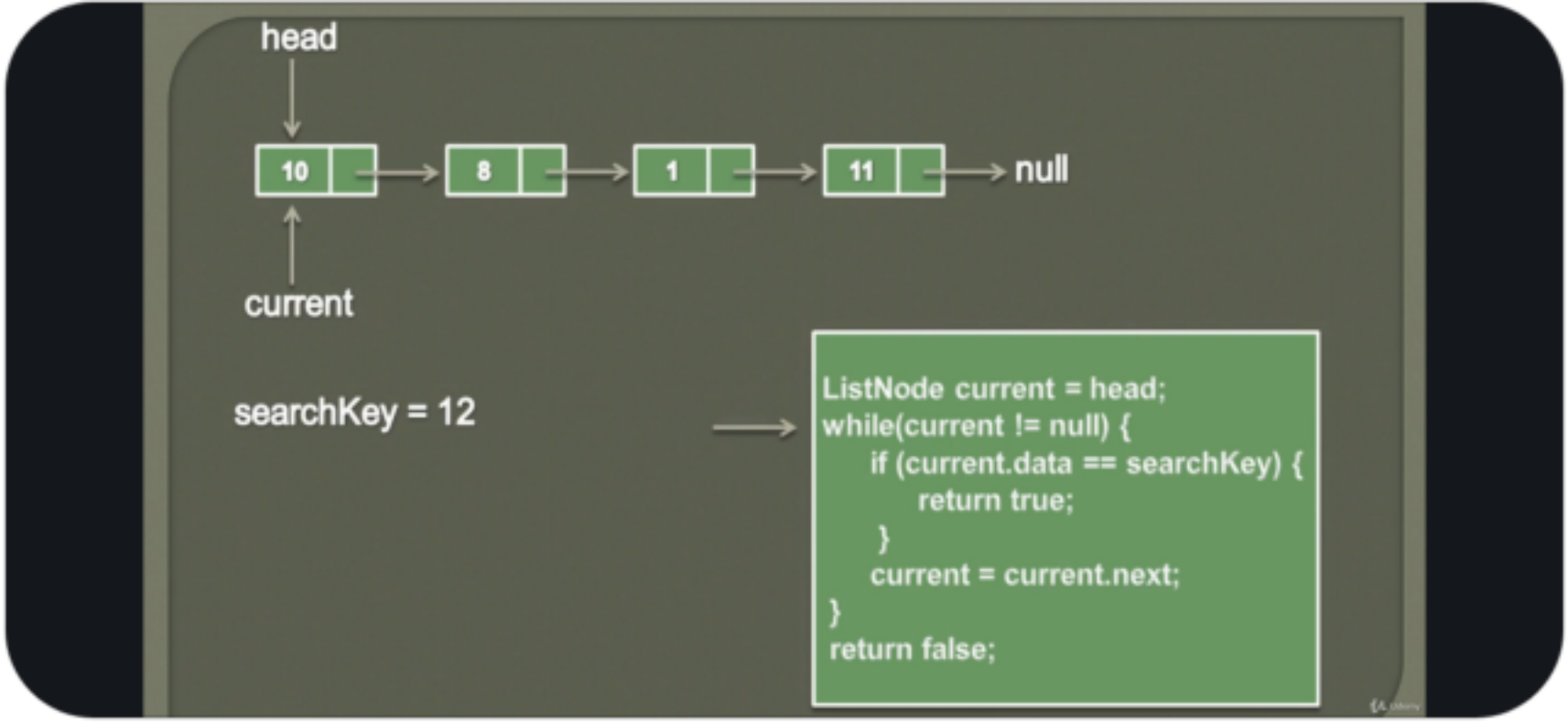


How to delete a node at given position

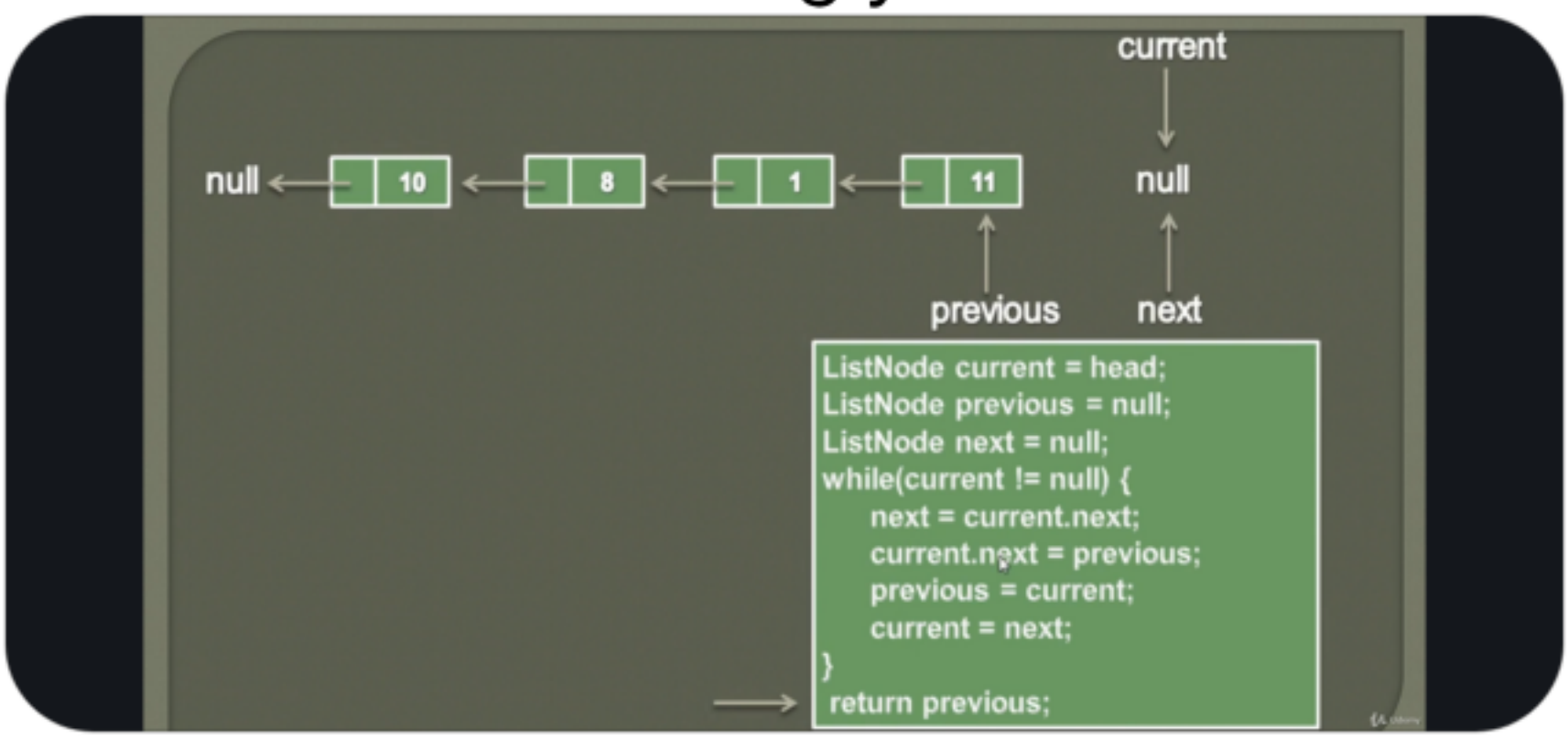
```
public void deleteAtPosition(int
position) {
    if(position == 0){
        ListNode temp = head;
        head = head.next;
        temp.next = null;
    } else {
        ListNode previous = head;
        int count = 0;
        while(count < position - 1){
            previous = previous.next;
            count++;
        }

        ListNode current = previous.next;
        previous.next = current.next;
        current.next = null;
    }
}
```

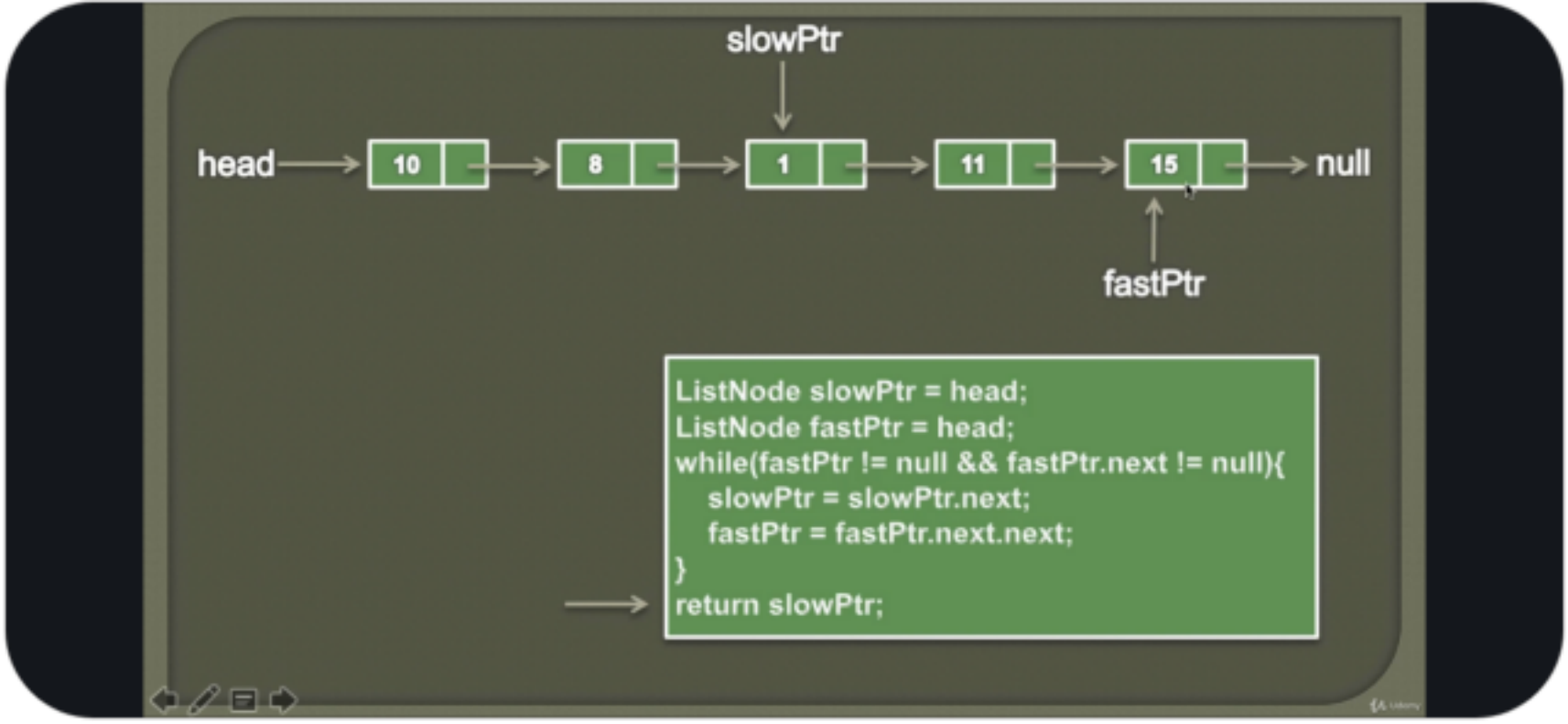
How to search an element



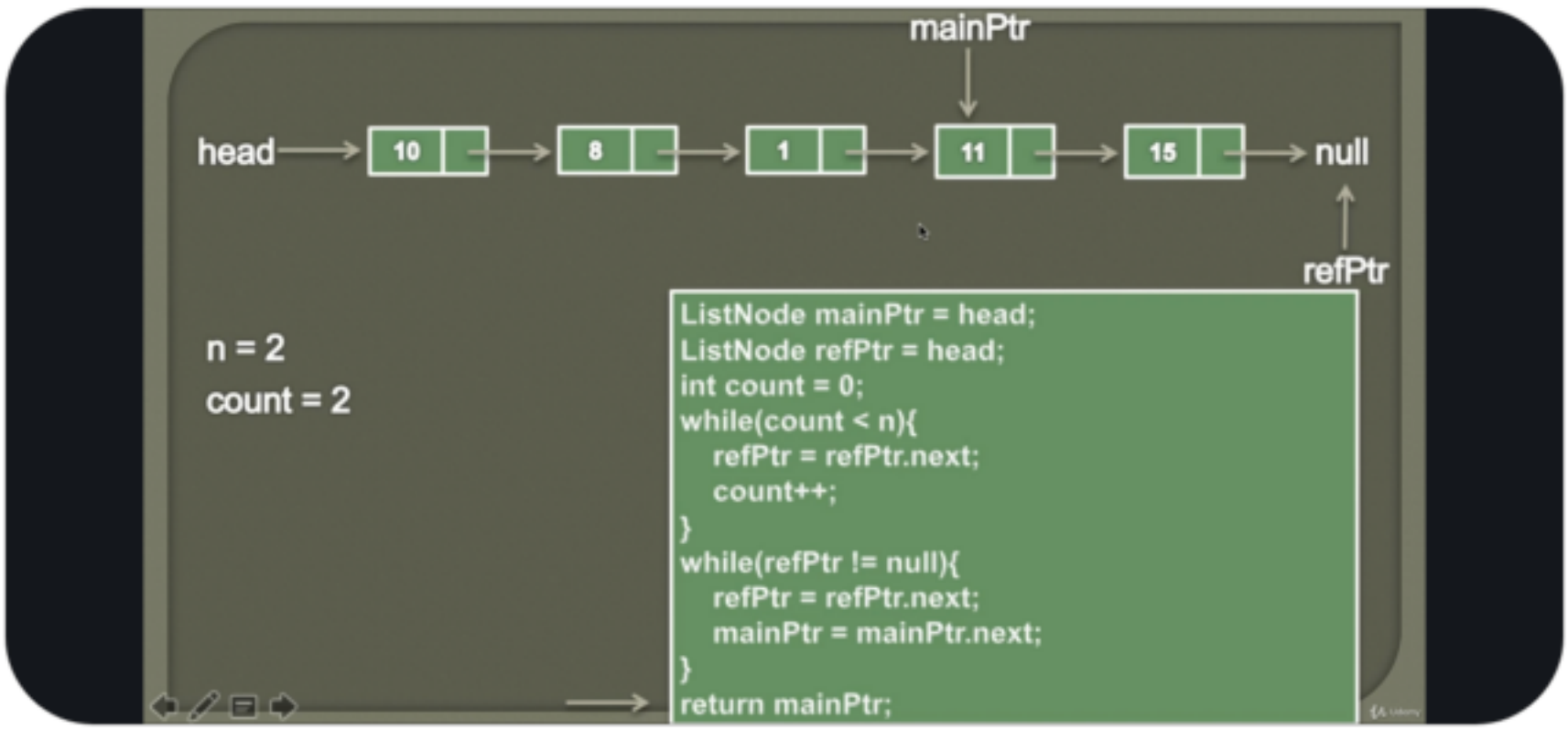
How to reverse a singly linked list



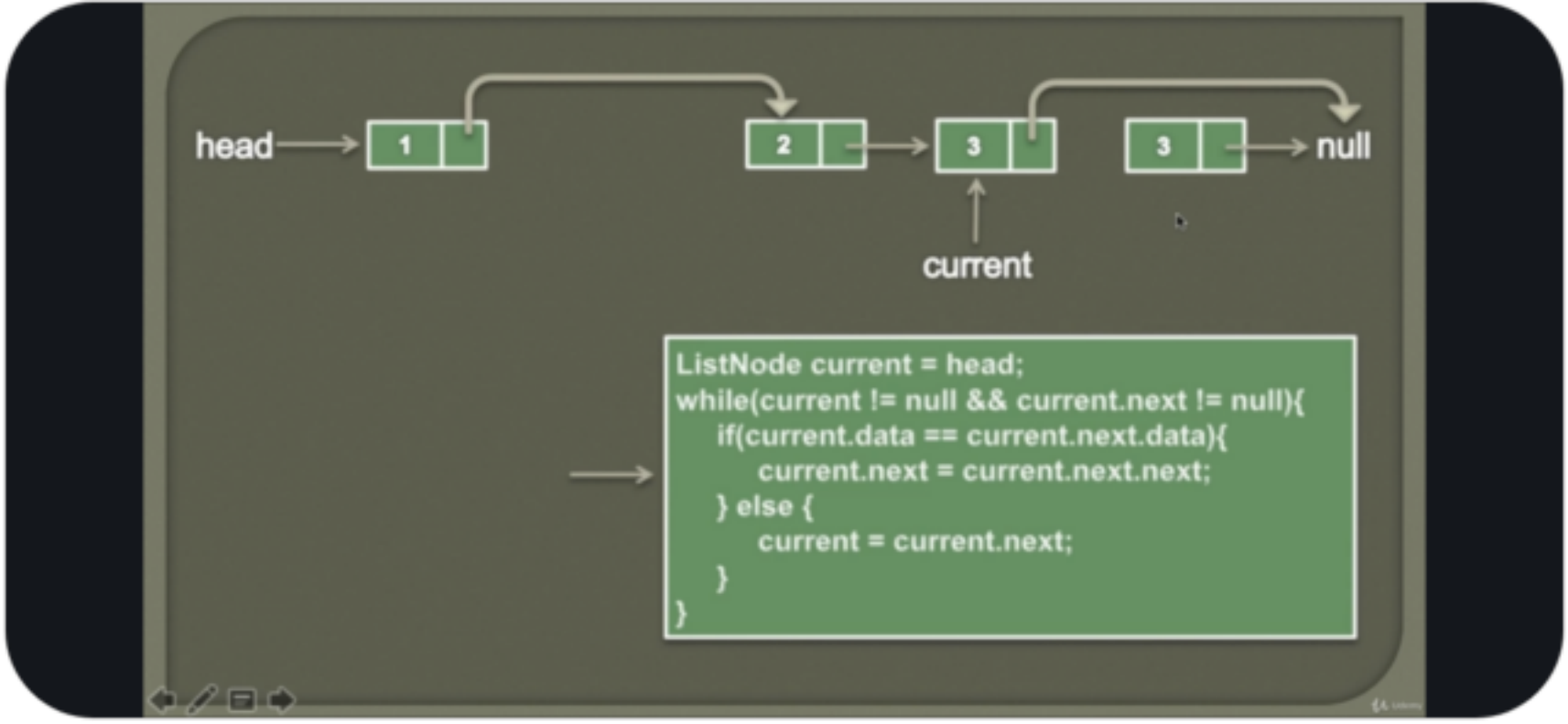
How to find middle node



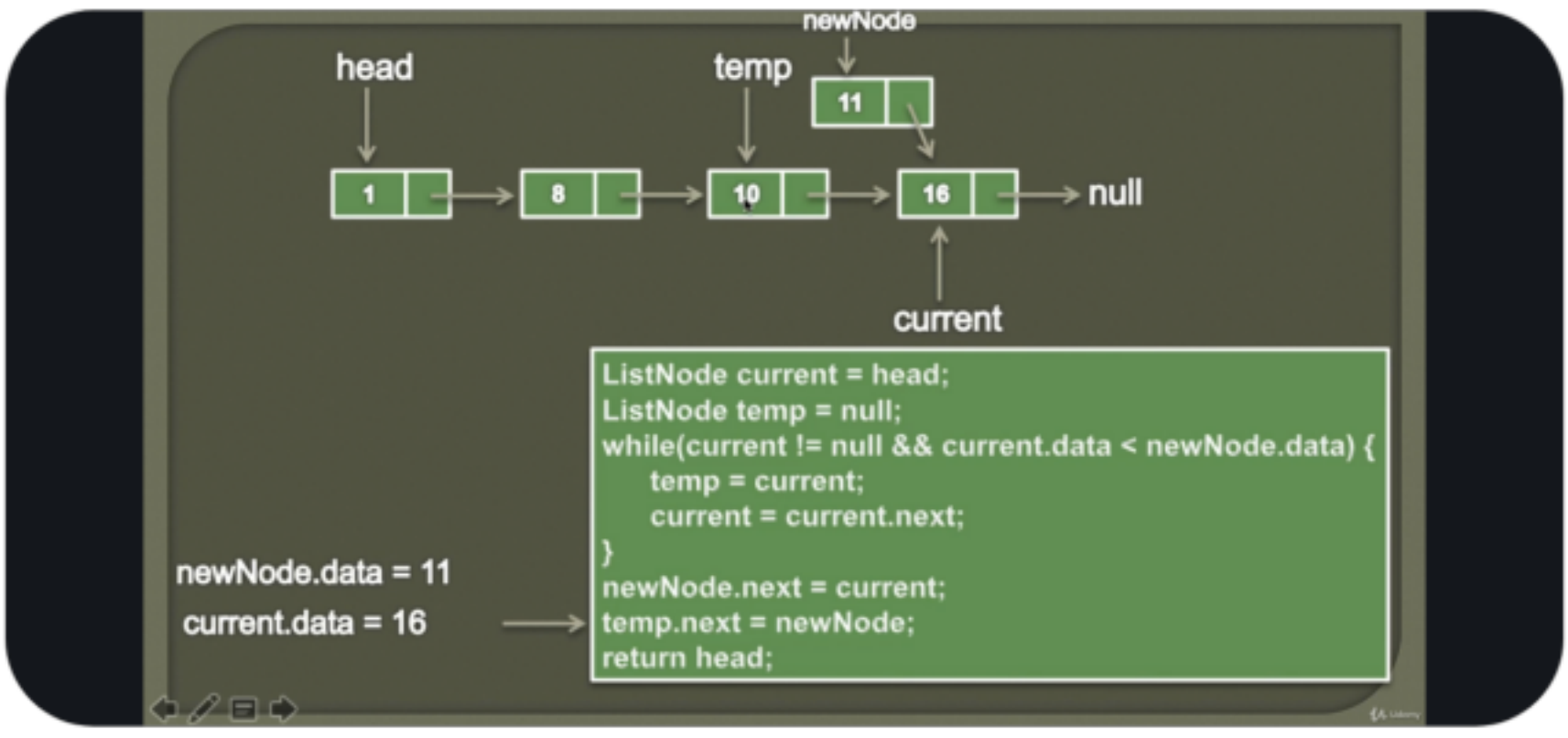
How to find nth node from end



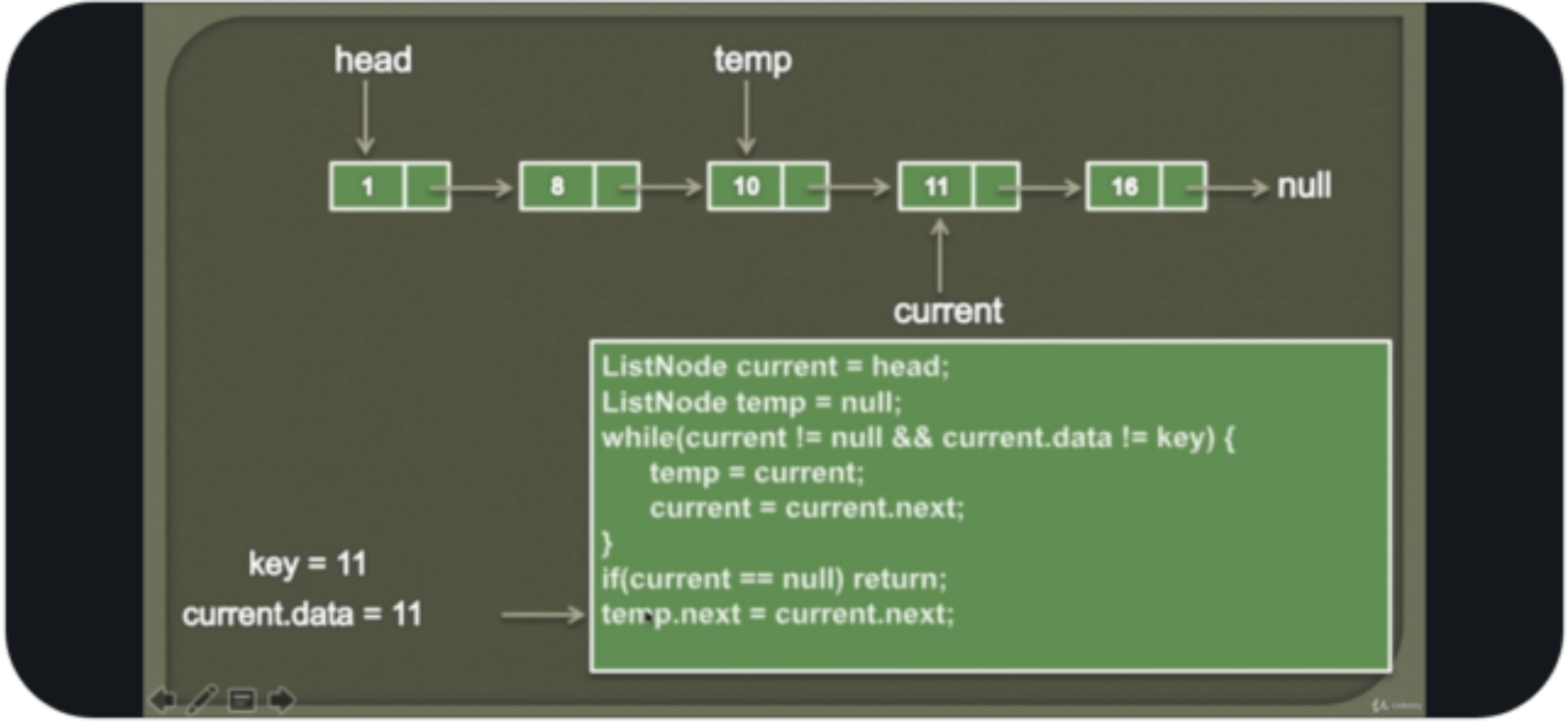
How to remove duplicate from sorted singly list



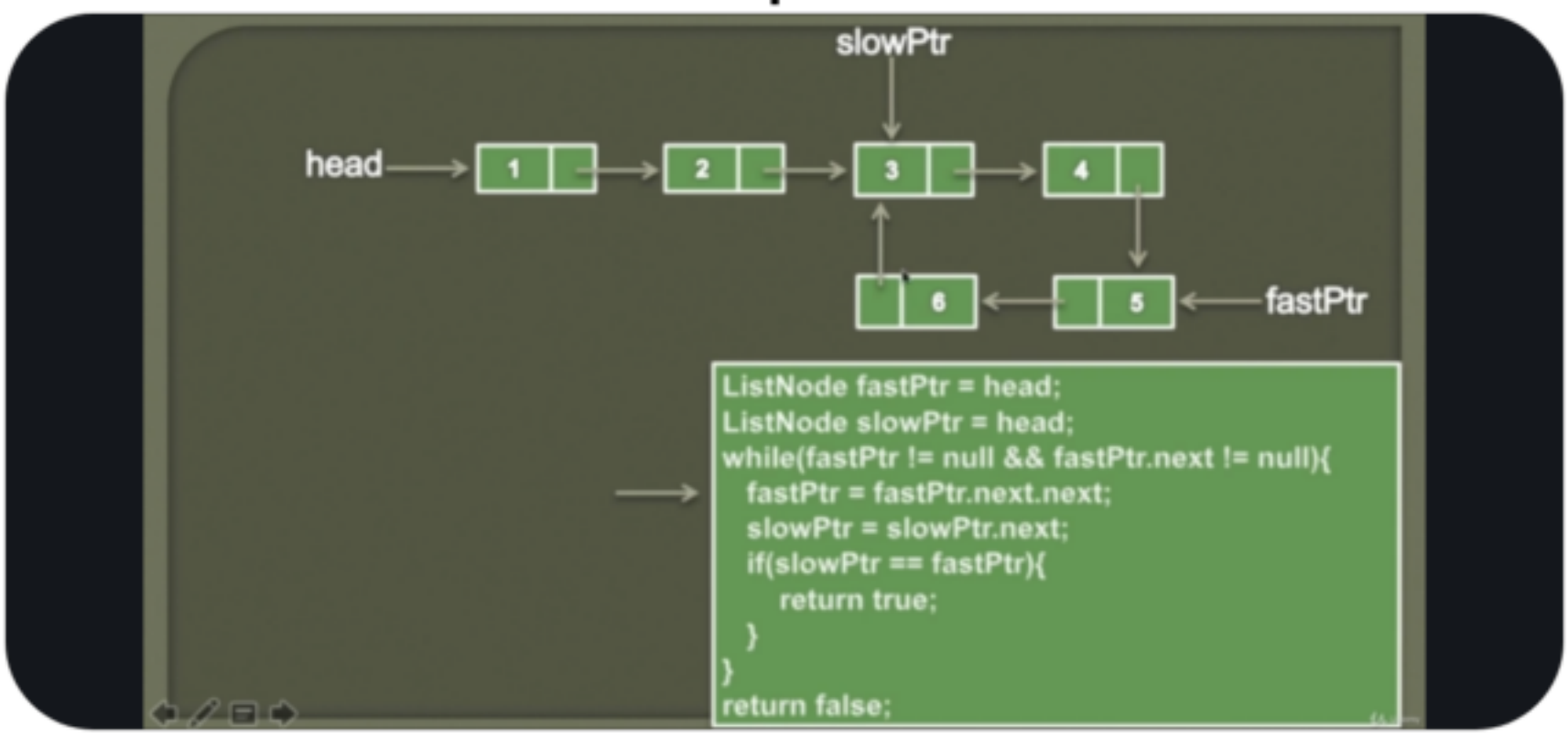
How to insert a node in a singly linked liat



How to remove a key



How to detect a loop

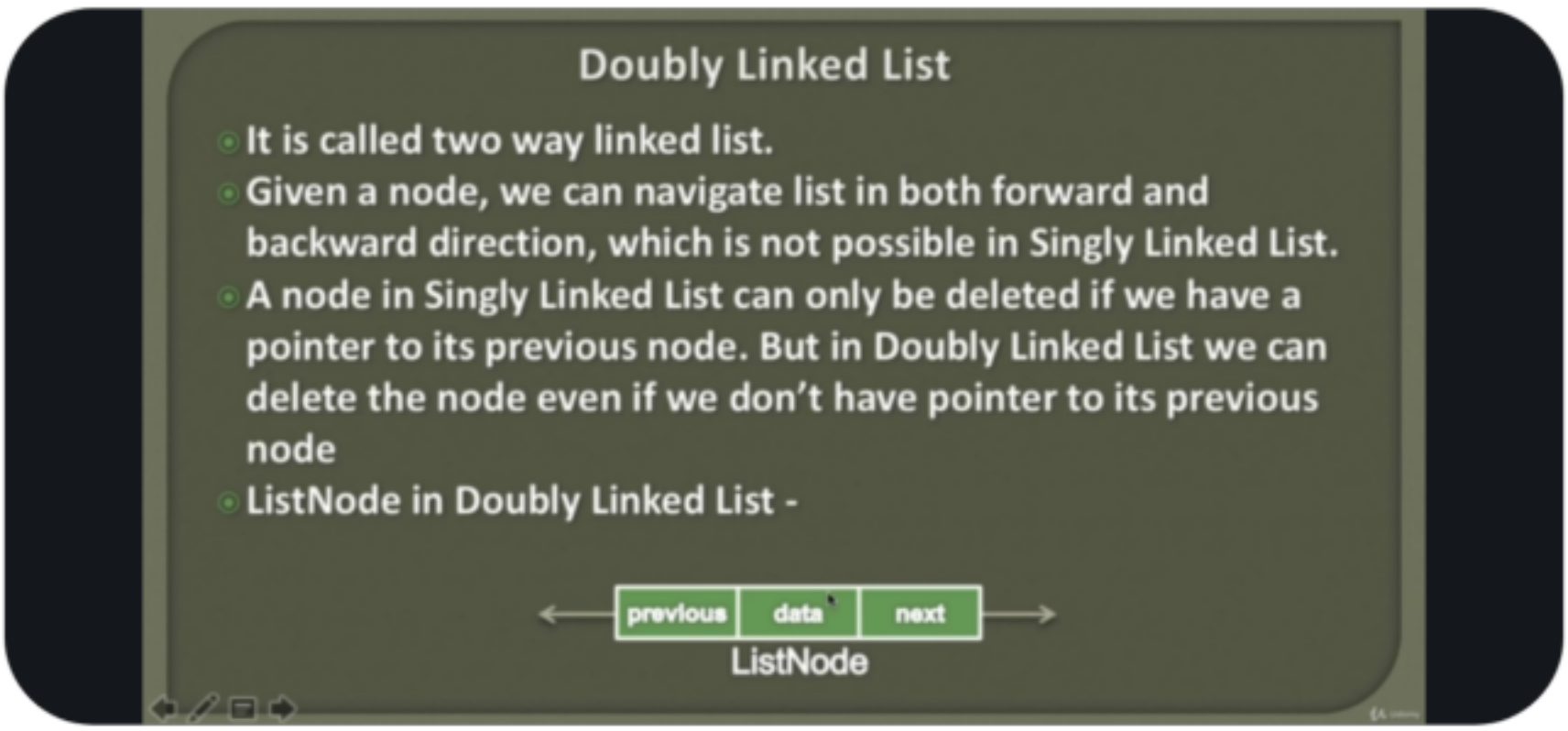


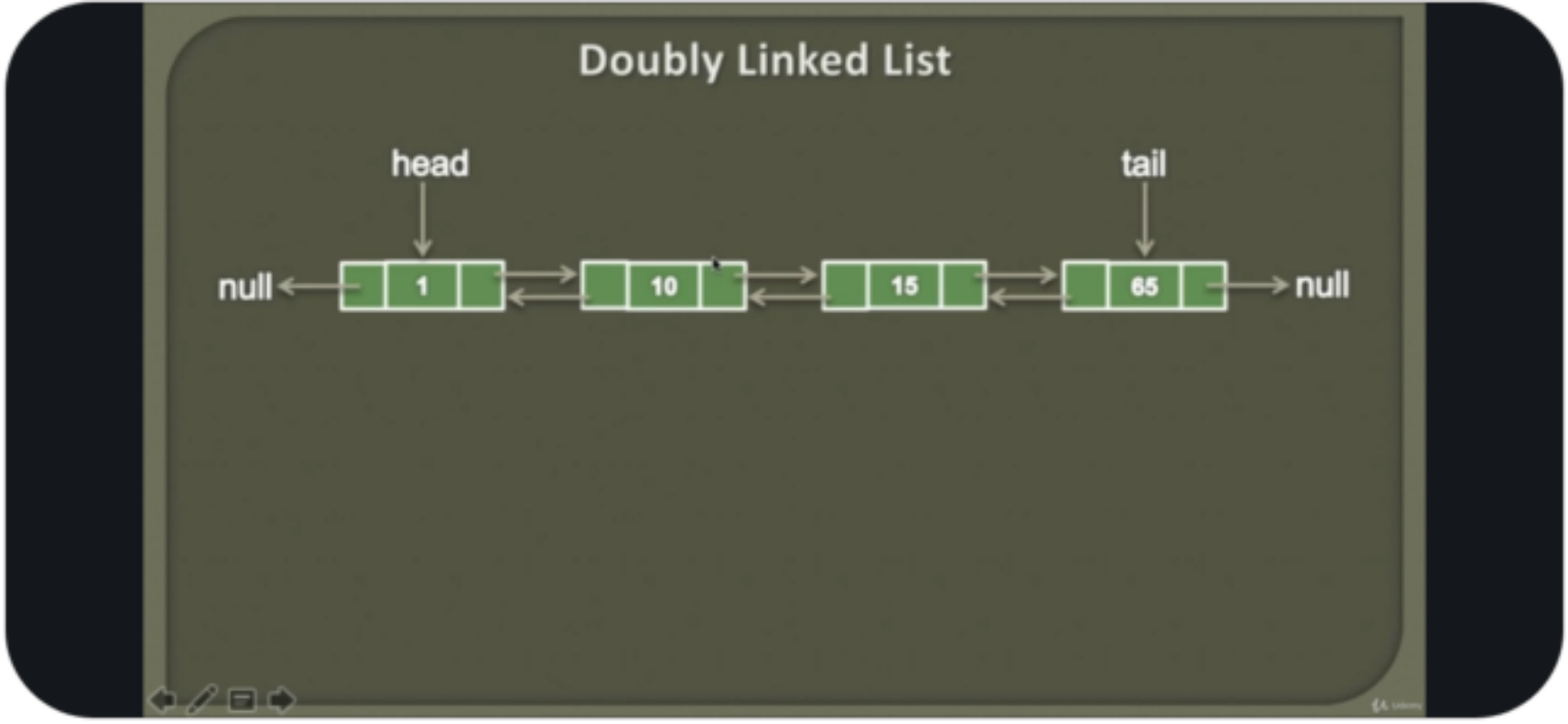
Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous

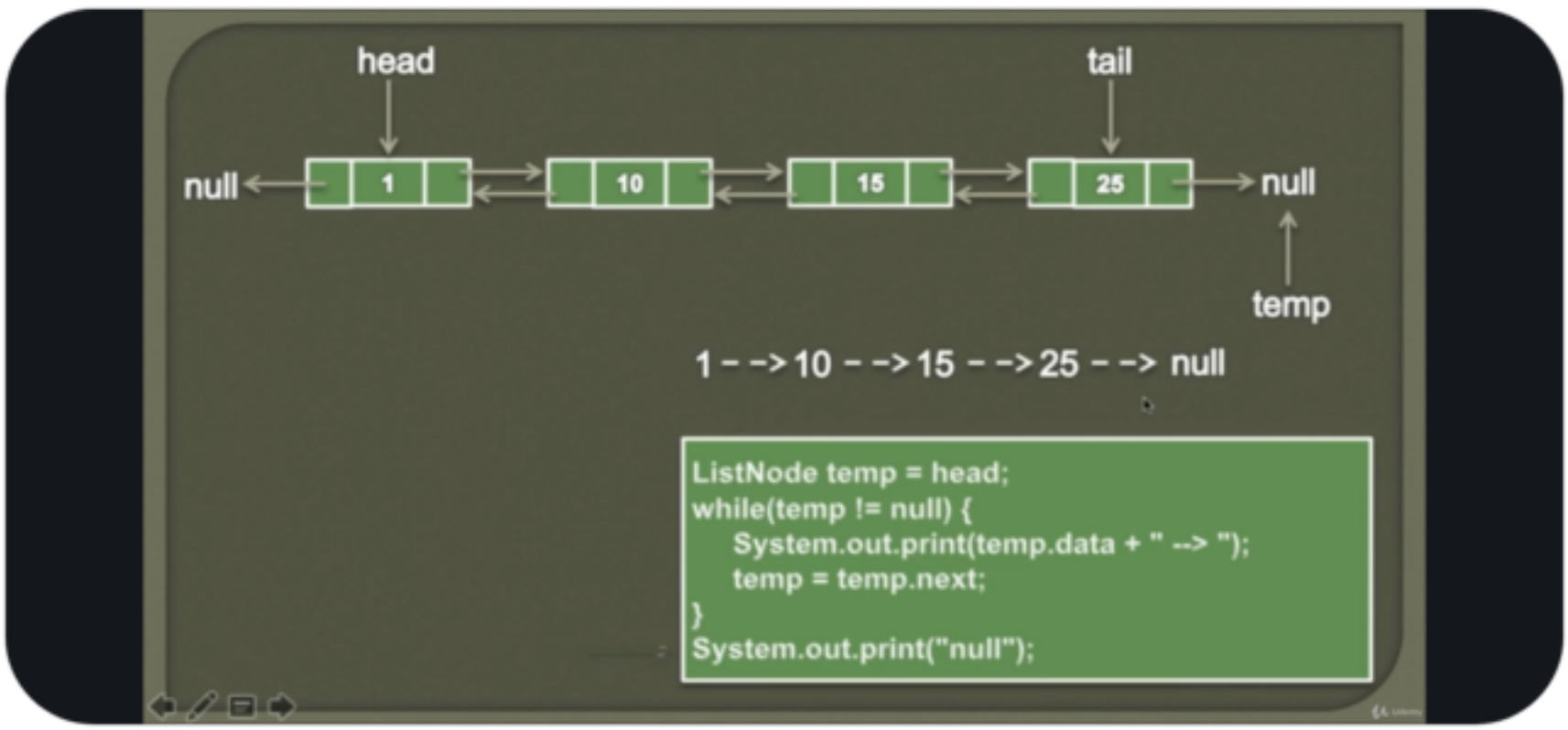
as well as the next node in the sequence.

- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).

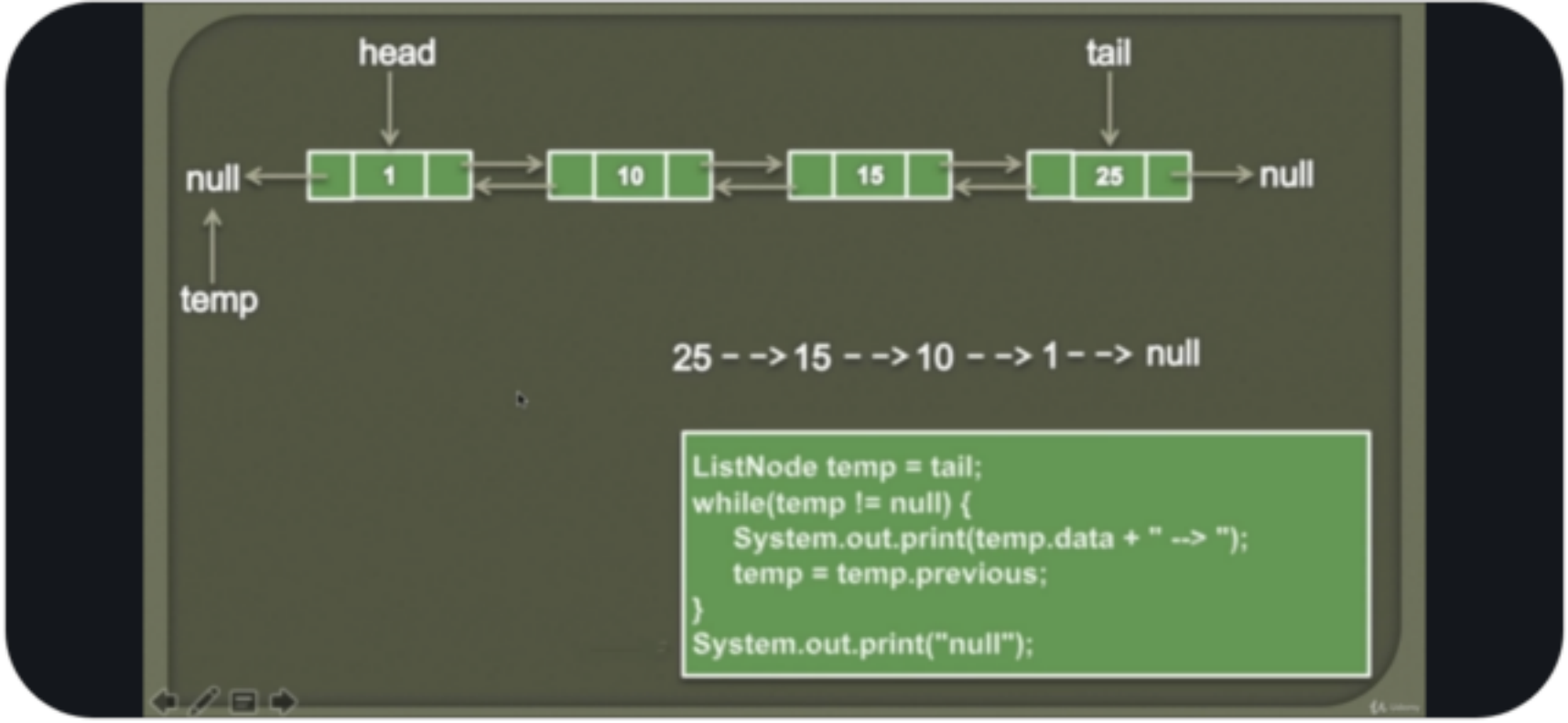




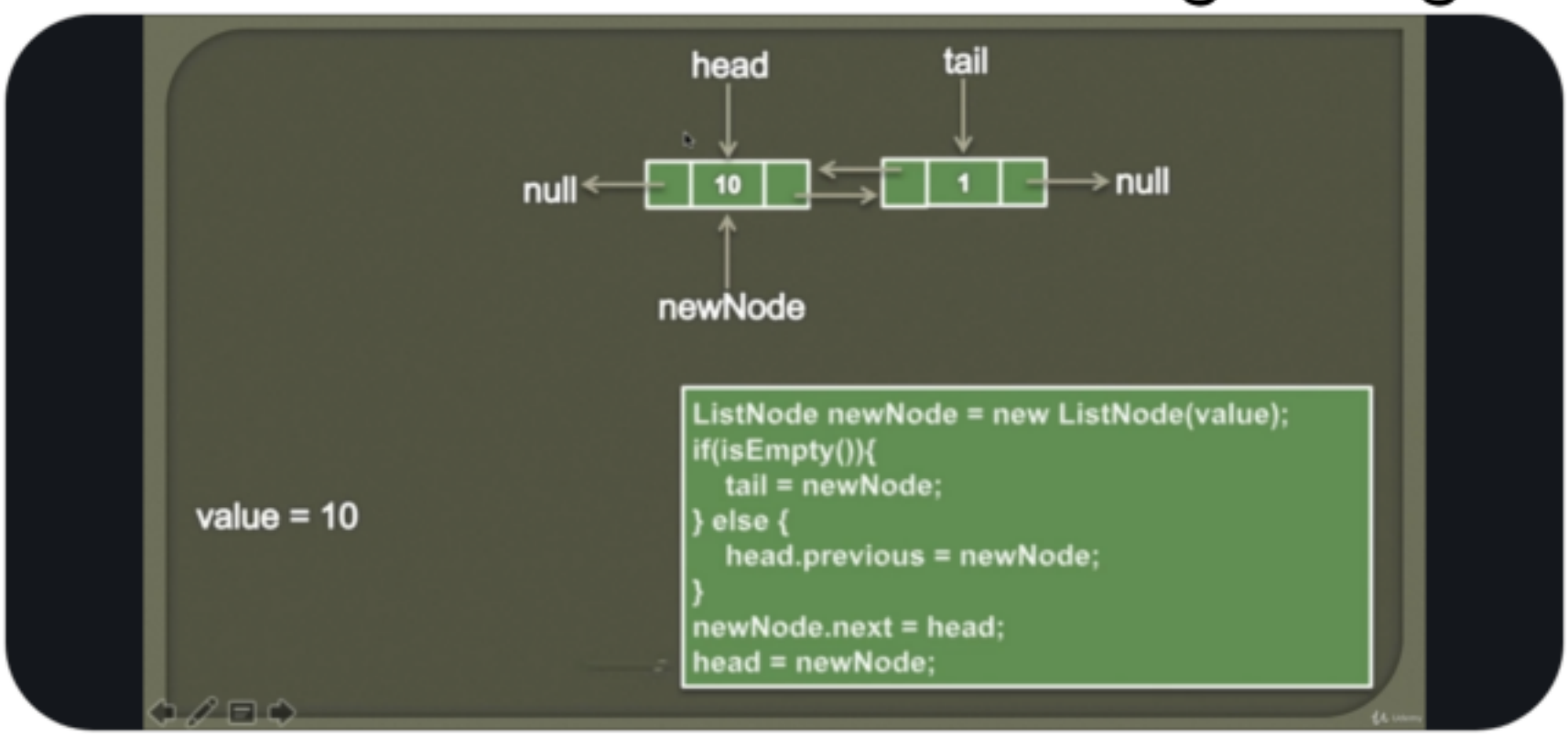
How to traverse in forward direction



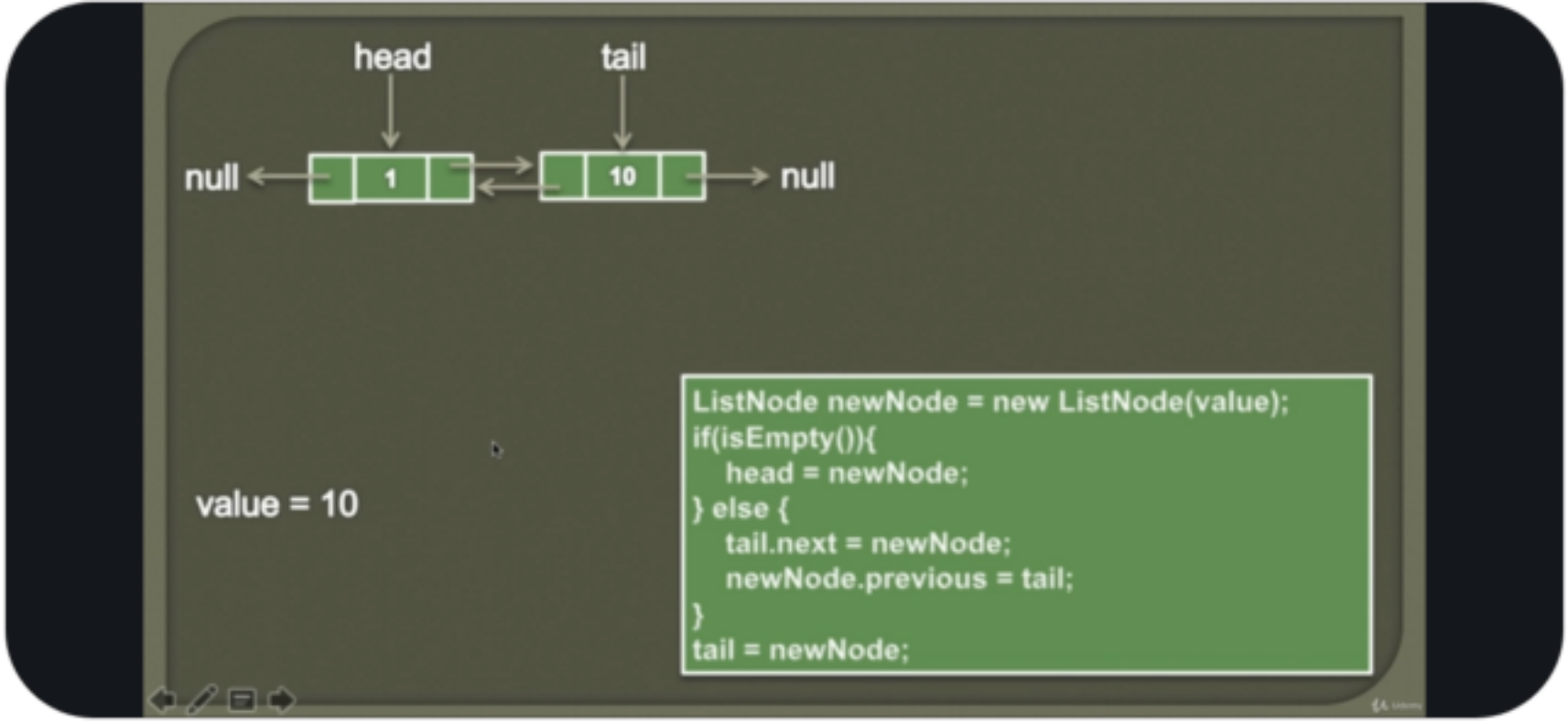
How to traverse in backward direction



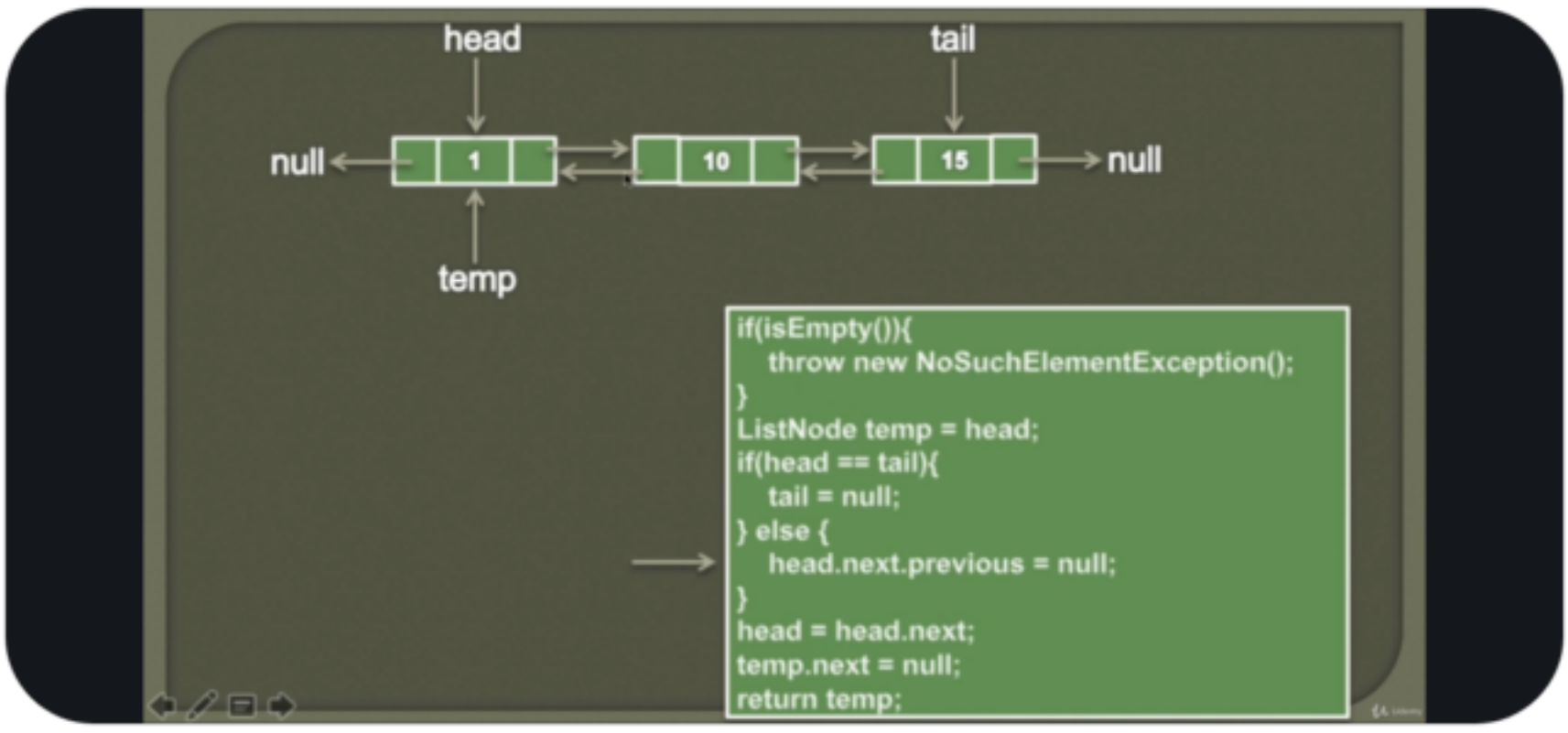
How to insert a node at the beginning



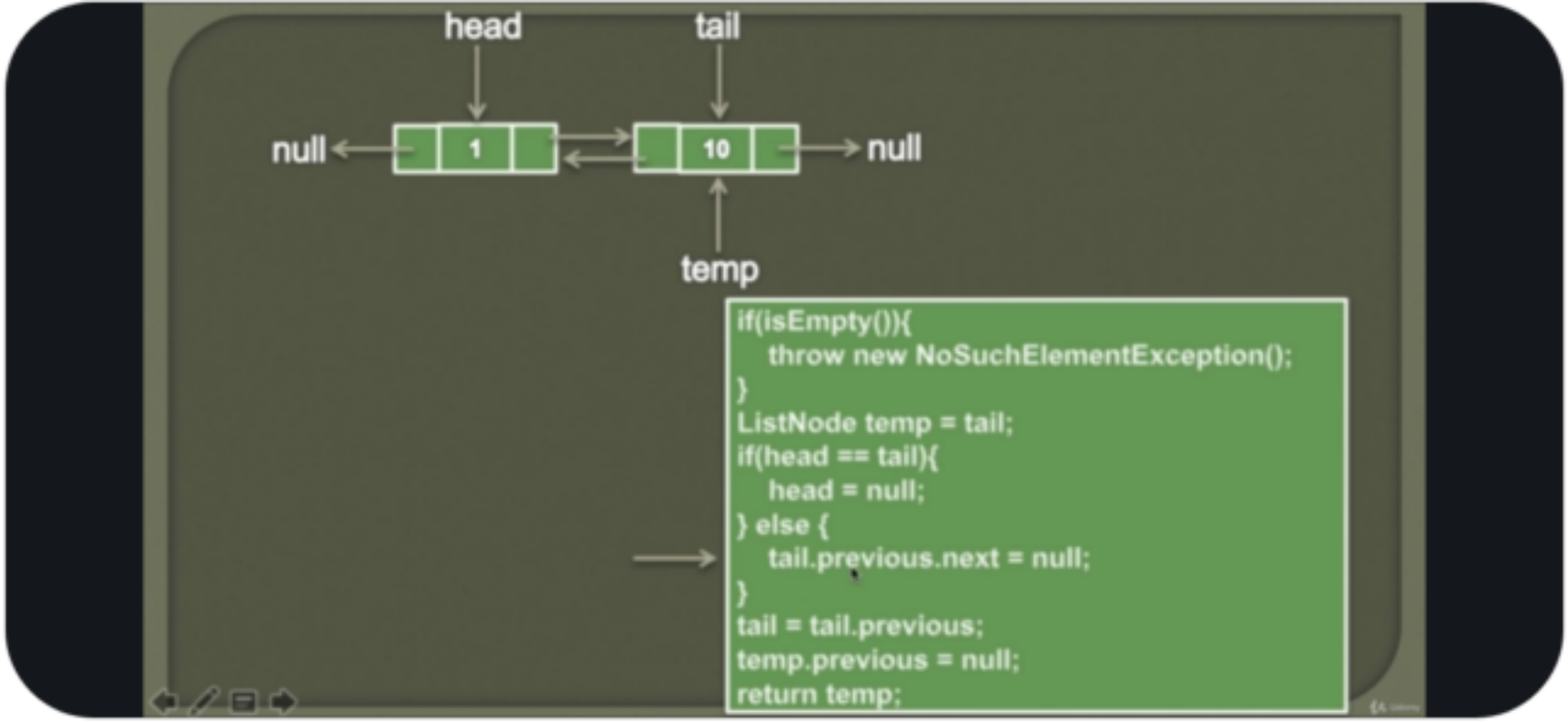
How to insert a node at the end



How to delete first bide



How to delete last node

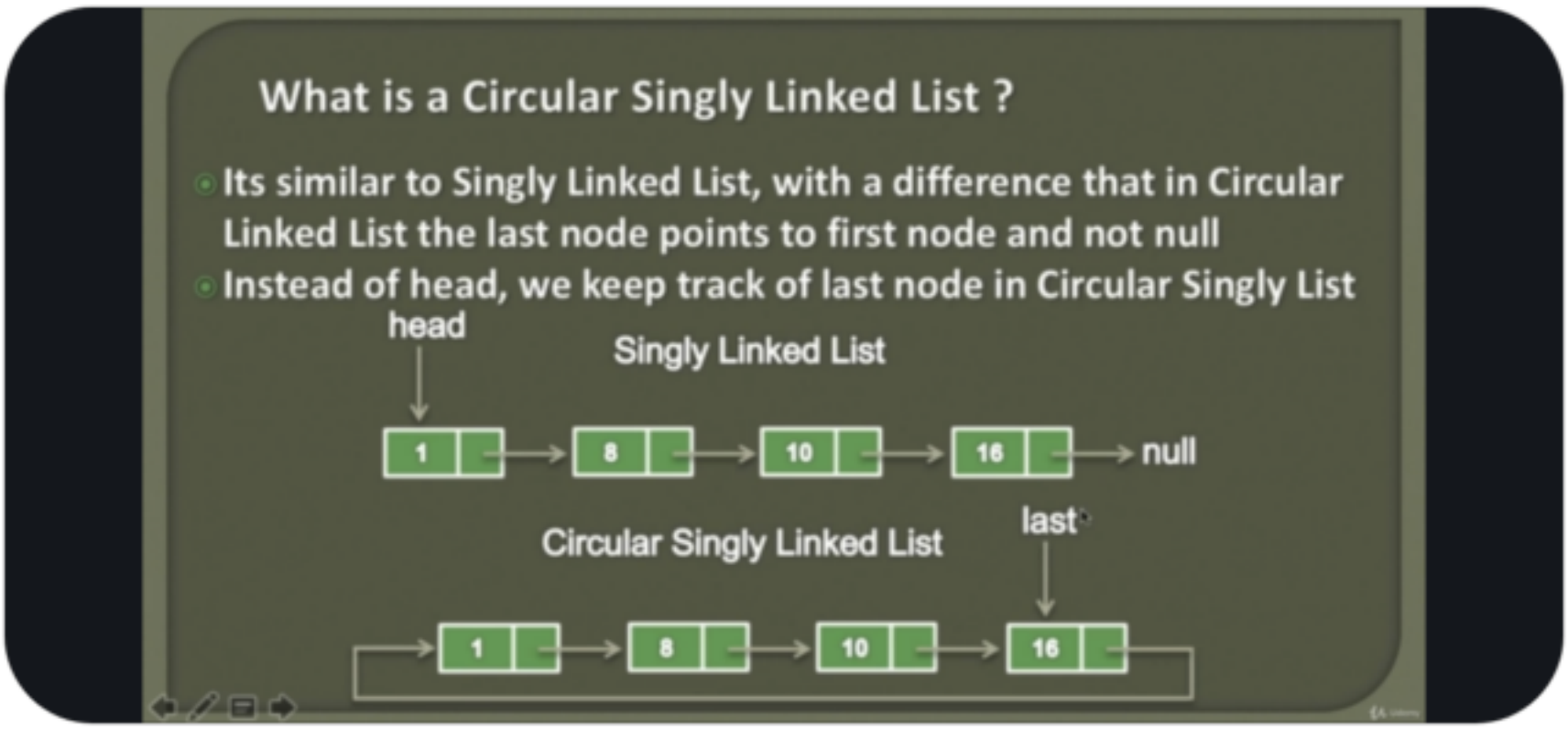


Circular singly linked list

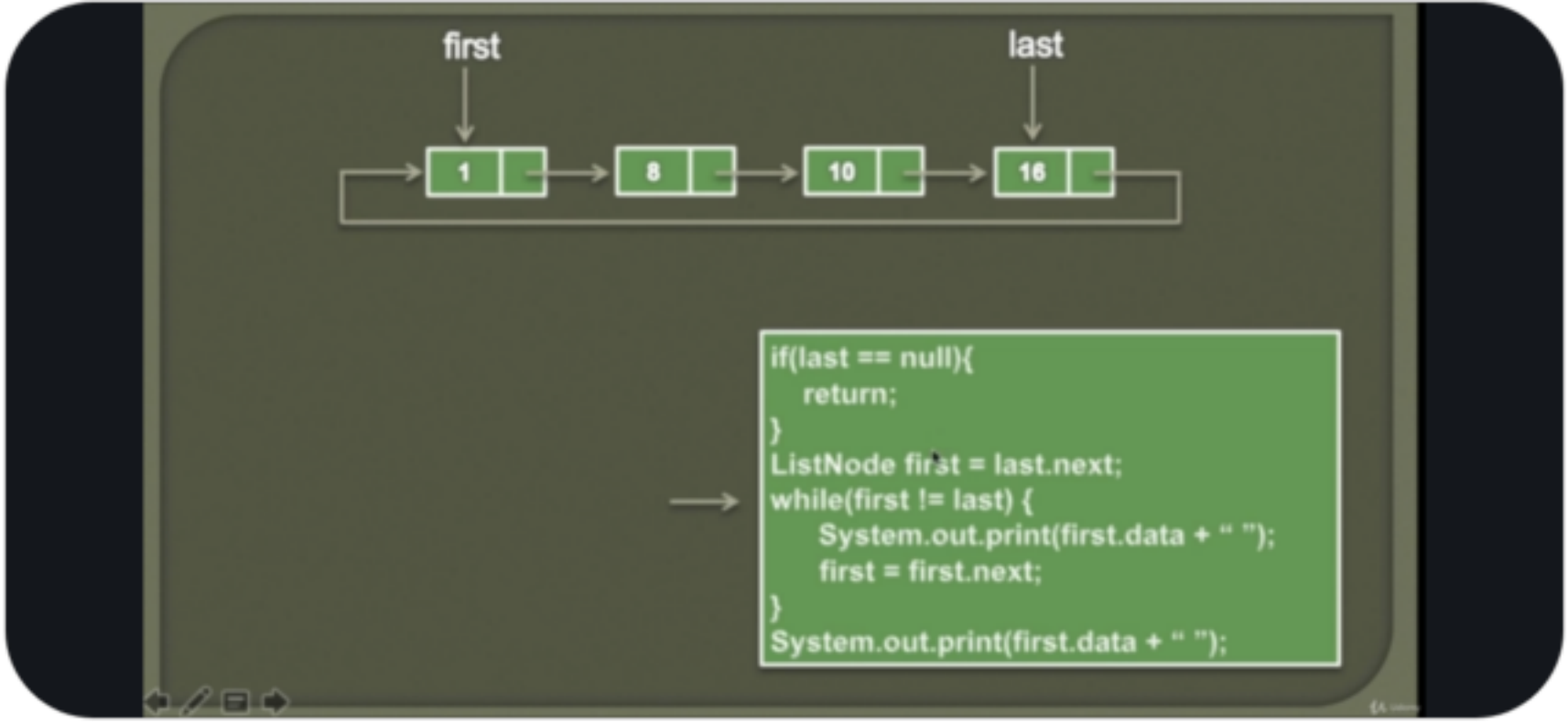
- A Circular linked list is yet another variation of a linked list in which the tail node points to the head node of the list and hence a circular linked list does not have an end unlike singly linked list in which the tail node points to NULL.
- The complete list can be traversed starting with any node
- The queue data structure implementation in Java uses a

circular linked list as its internal implementation.

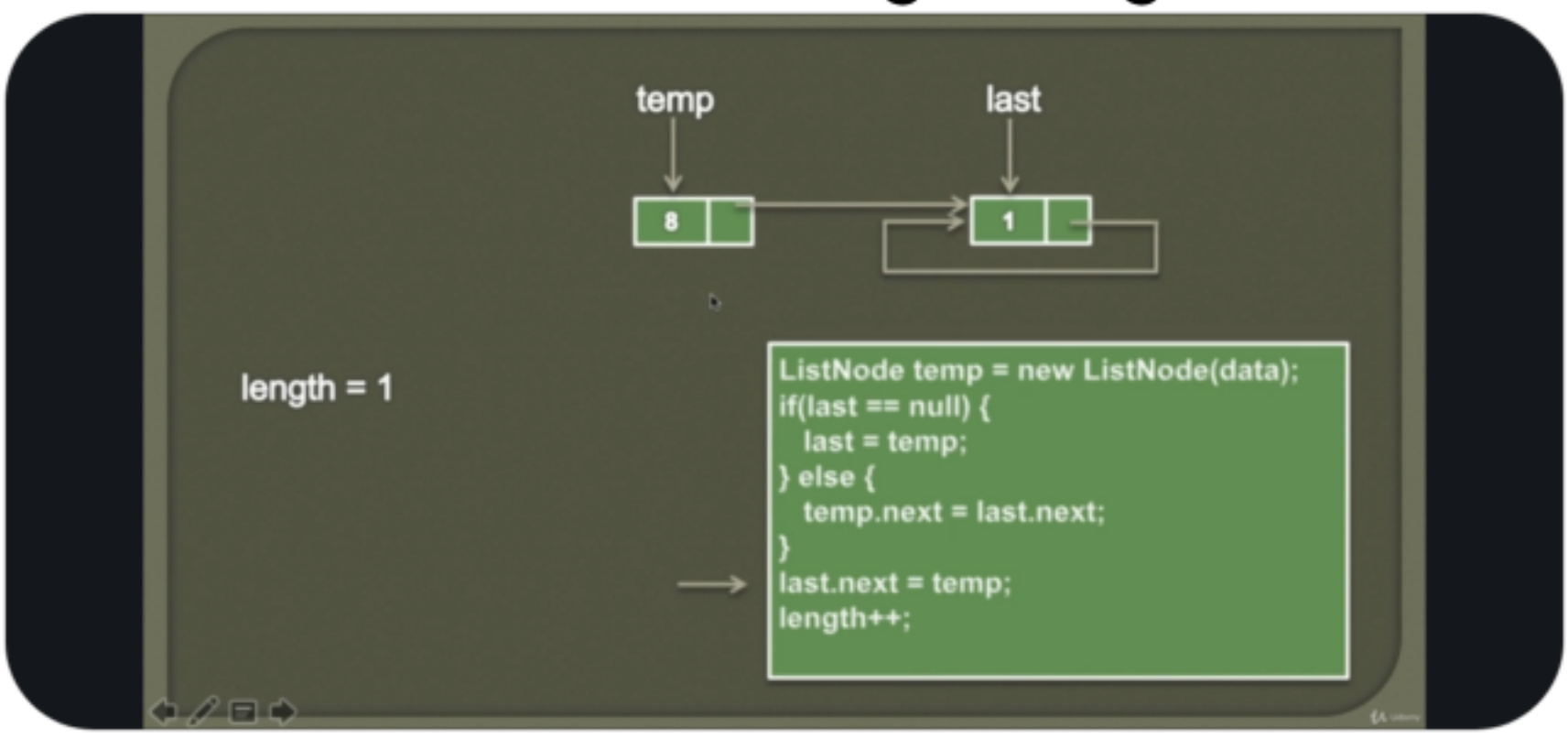
- Traversing from the last node to the head node can be done in constant time. With conventional linked lists, this is a linear operation.



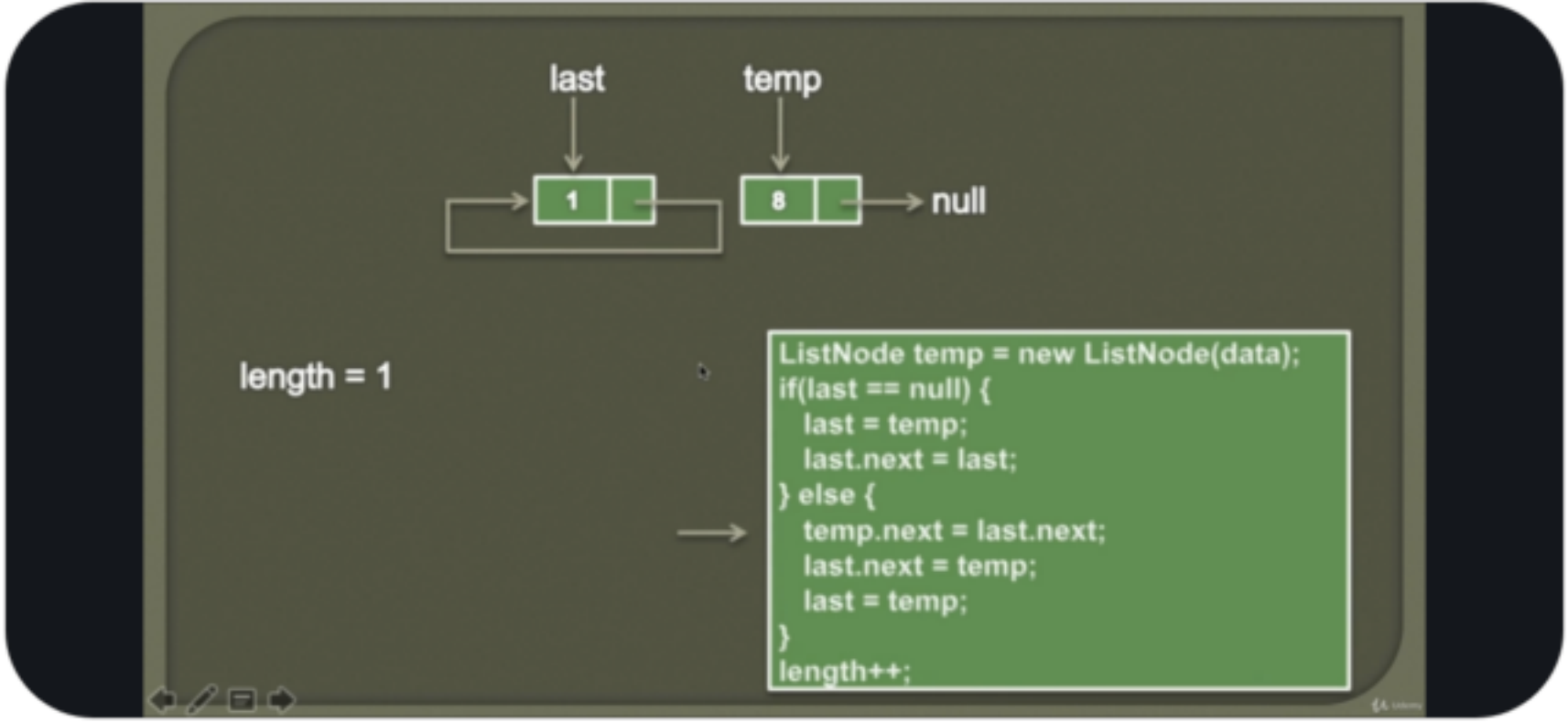
How to traverse



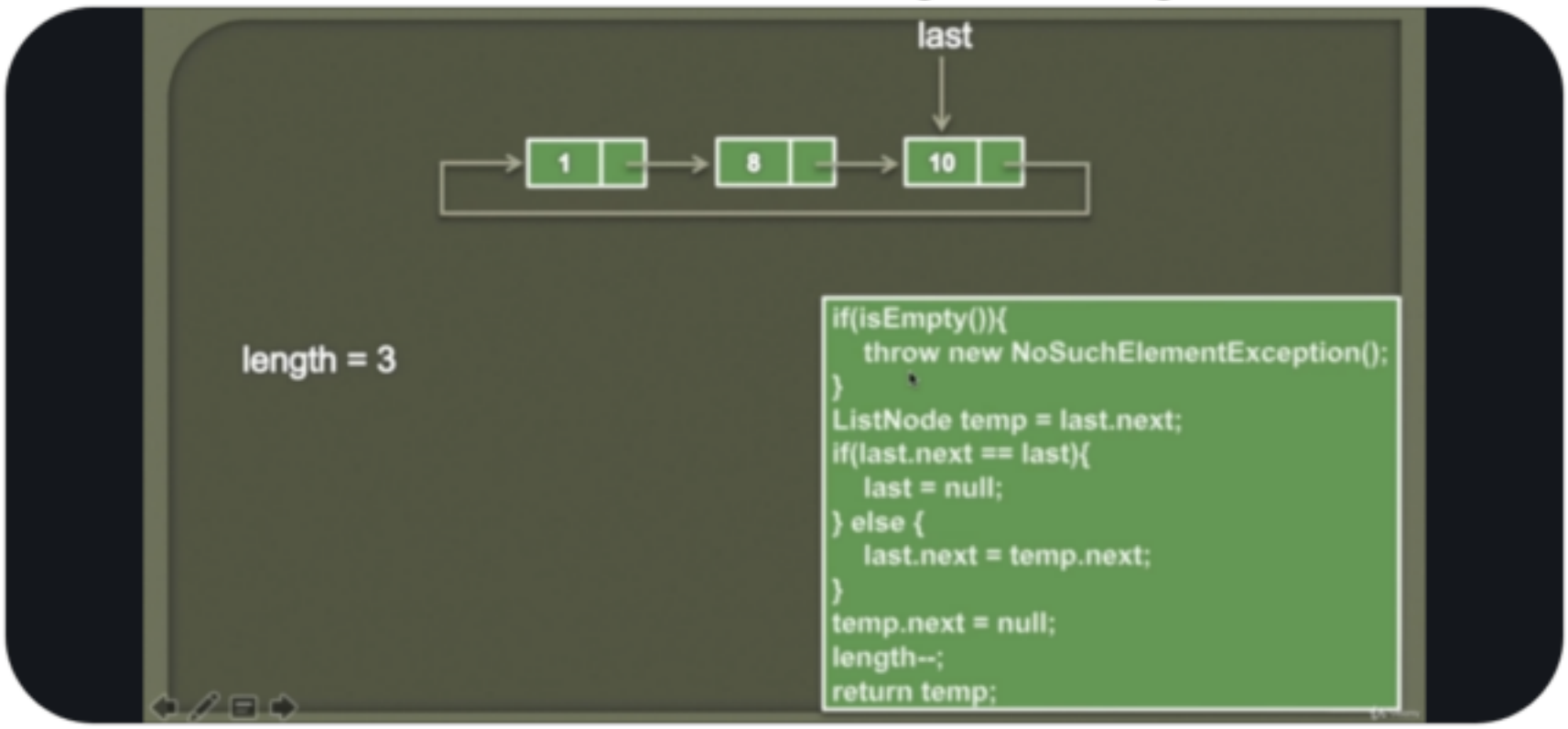
Insert a node at the beginning



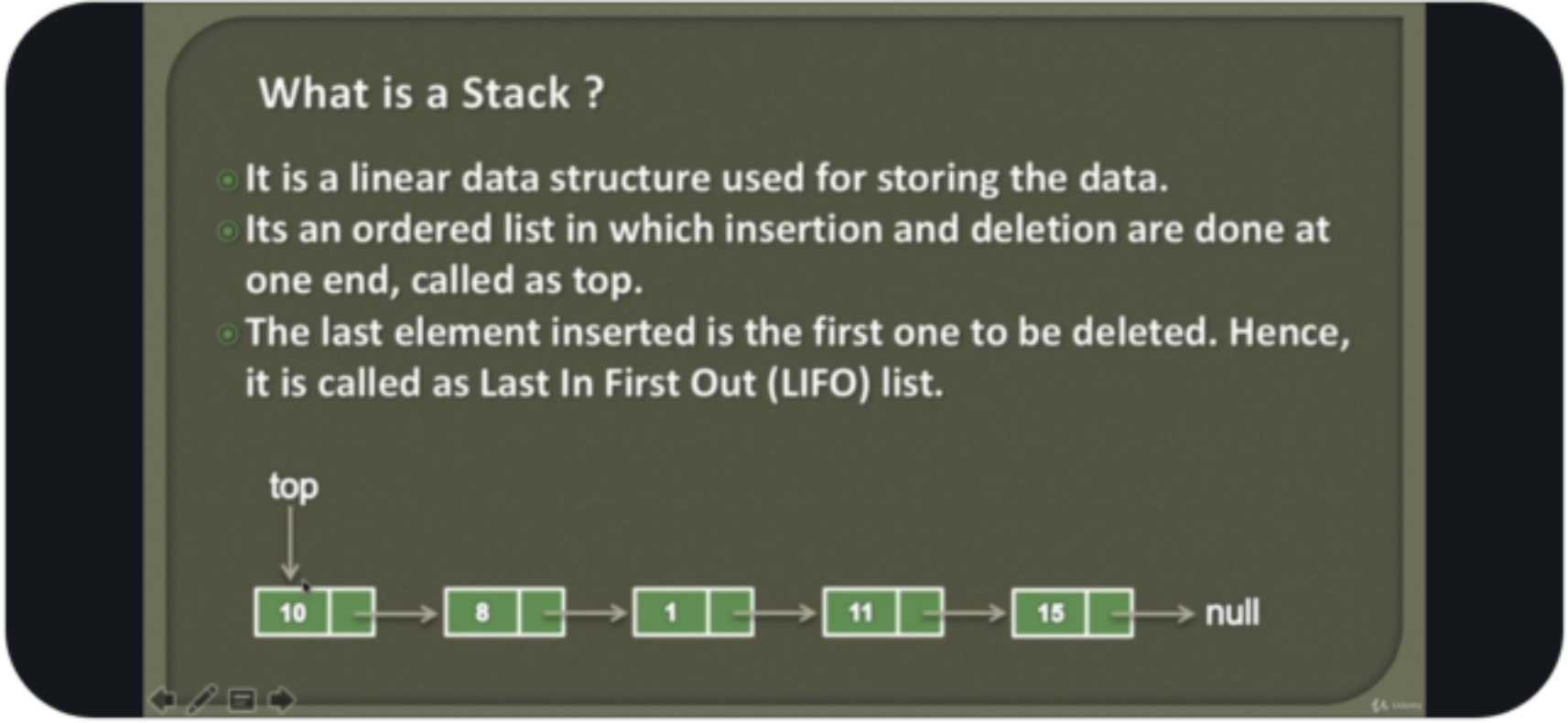
Insert a node at the end



Delete a node at the beginning



Stack



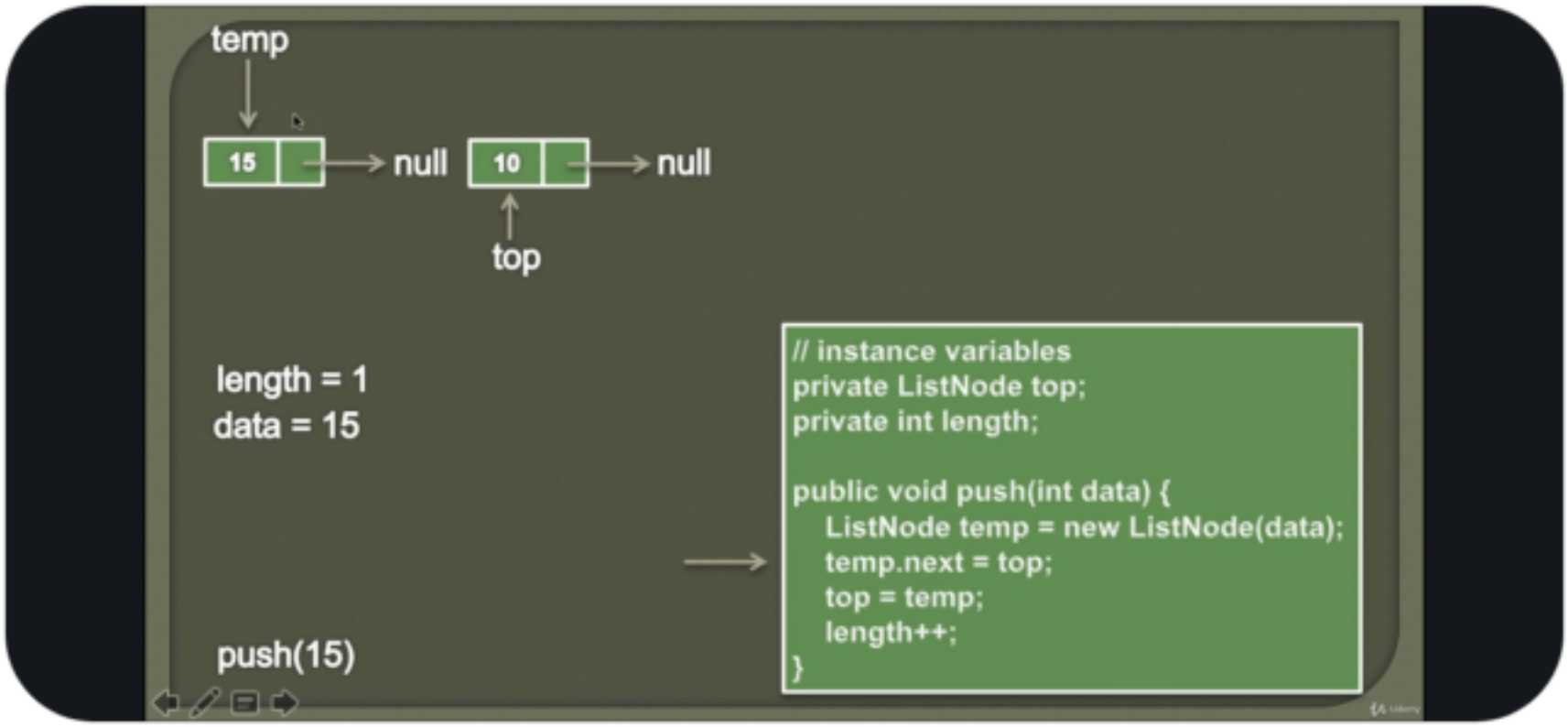
- The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO)
- push operation inserts an element into the stack and pop operation removes an element from the top of the stack.
- Empty Stack: If the stack has no element is known as an empty stack. When the stack is empty the value of the top variable is -1
- When we push an element into the stack the top is increased by 1 so it will be 0

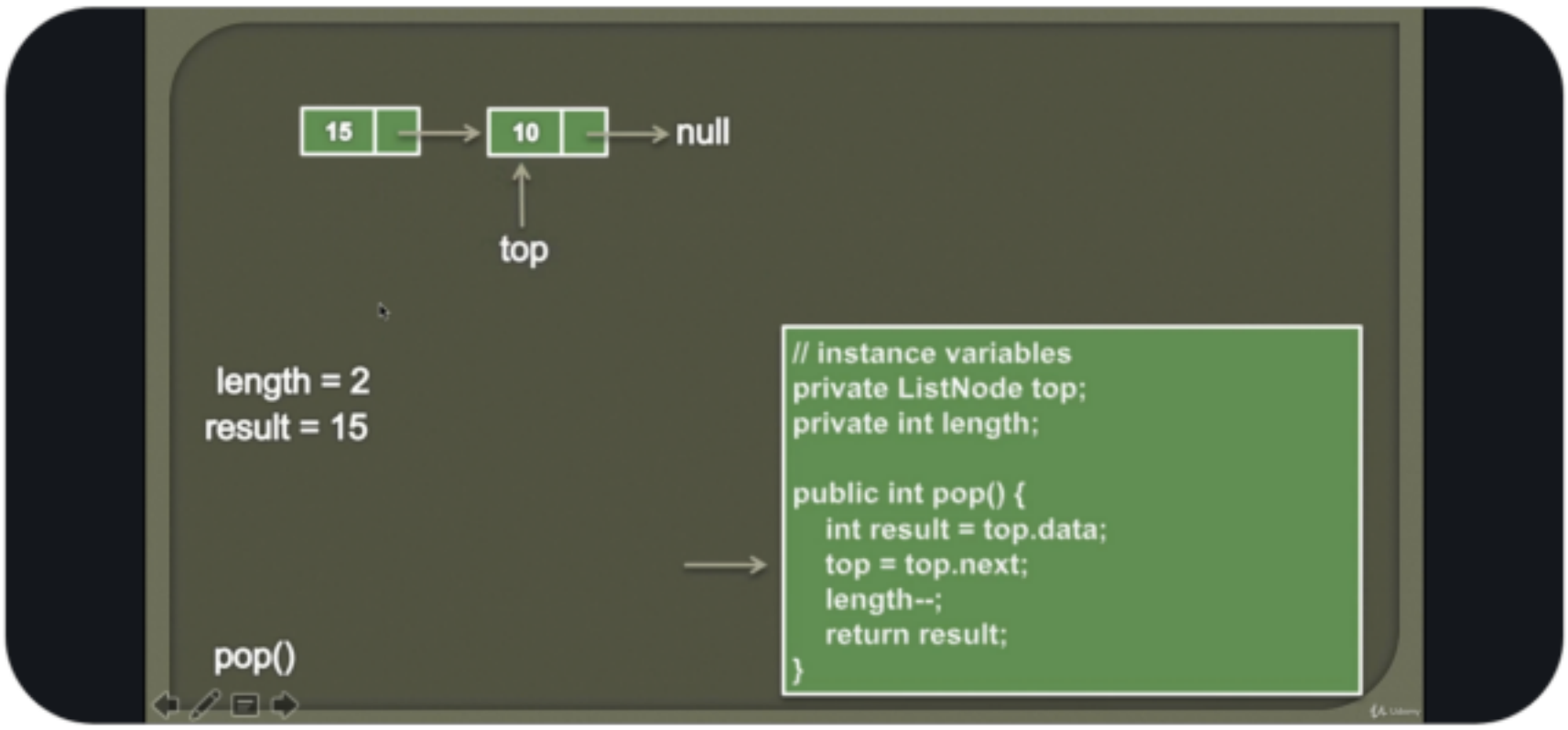
- When we pop an element from the stack the value of top is decreased by 1.
- Creating a Stack: Stack<type> stk = new Stack<>();
- Use ArrayDeque version of stack provided in Java against the Stack version. Since Stack version provides thread safety feature. ArrayDeque<Integer> ad = new ArrayDeque<>();

Methods

- empty() - check empty or not
- push(E e)
- pop()
- peek()
- search(Object o) - search and return object position
- size()
- iterator
- listIterator

How to implement stack using list





How to reverse a string using stack

String str = "ABCD"

str.length() = 4

chars[ ]

A	B	C	D
0	1	2	3

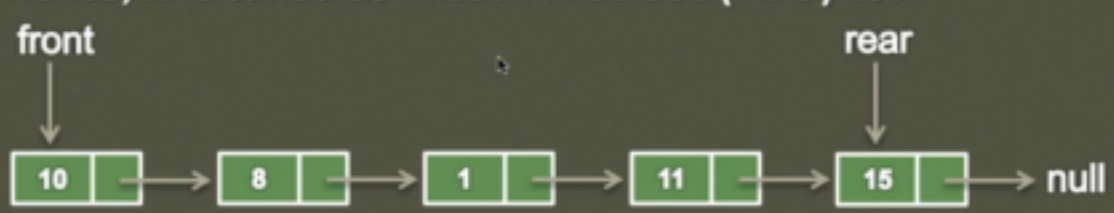
i = 0

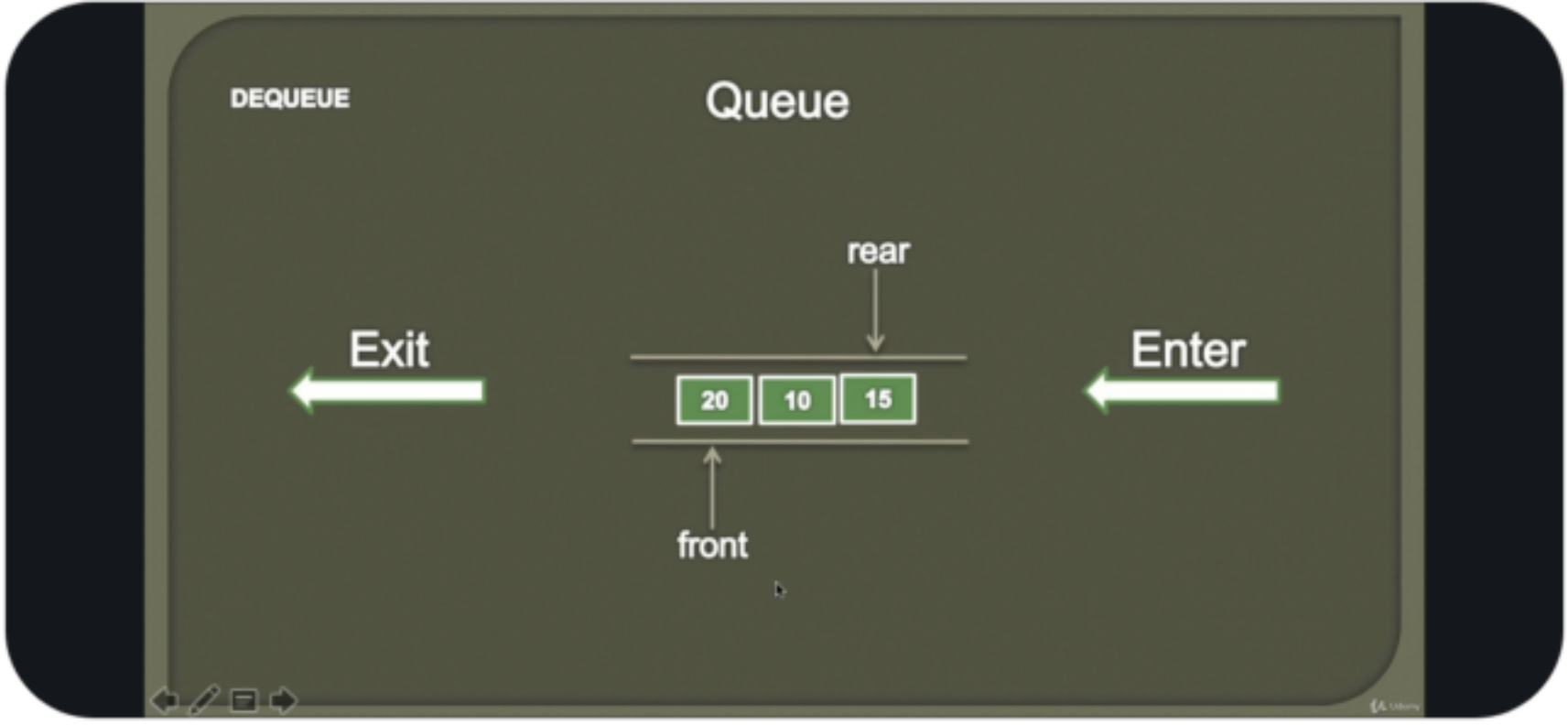

```
Stack<Character> stack = new Stack<>();
char[ ] chars = str.toCharArray();
for(char c : chars) {
    stack.push(c);
}
for(int i = 0; i < str.length(); i++) {
    chars[ i ] = stack.pop();
}
return new String(chars);
```

Queue

What is a Queue ?

- It is a linear data structure used for storing the data.
- Its an ordered list in which insertion are done at one end, called as rear and deletion are done at other end called as front.
- The first element inserted is the first one to be deleted. Hence, it is called as First In First Out (FIFO) list.





- In queues, elements are stored and accessed in First In, First Out manner
- That is, elements are added from the behind called rear end and removed from beginning called front end
- Adding an element is called Enqueue and deleting an element is called Dequeue
- Implementation:

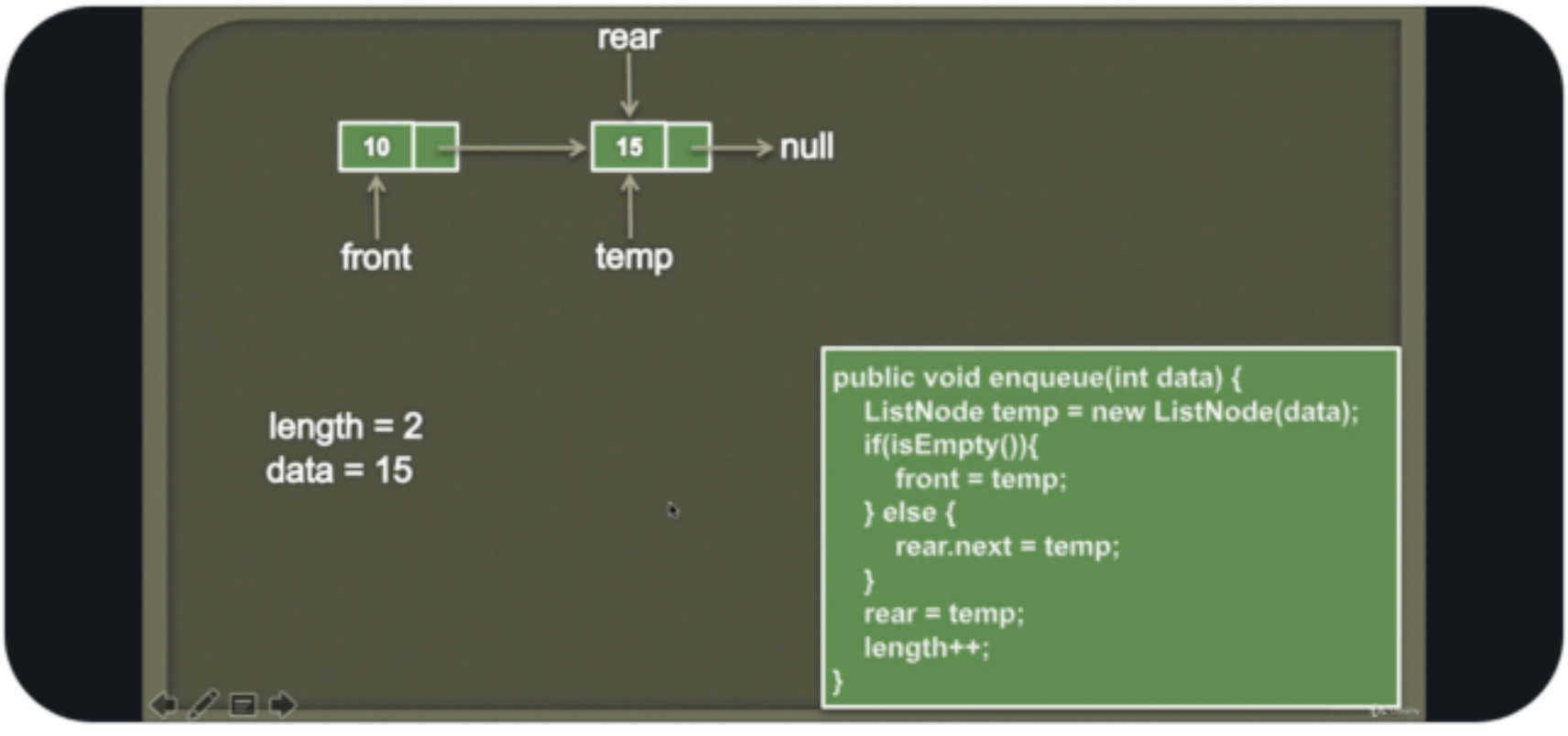
```
//List implementation of queue
Queue<String> animal1 = new
                LinkedList<>();
//Array implementation of Queue
```

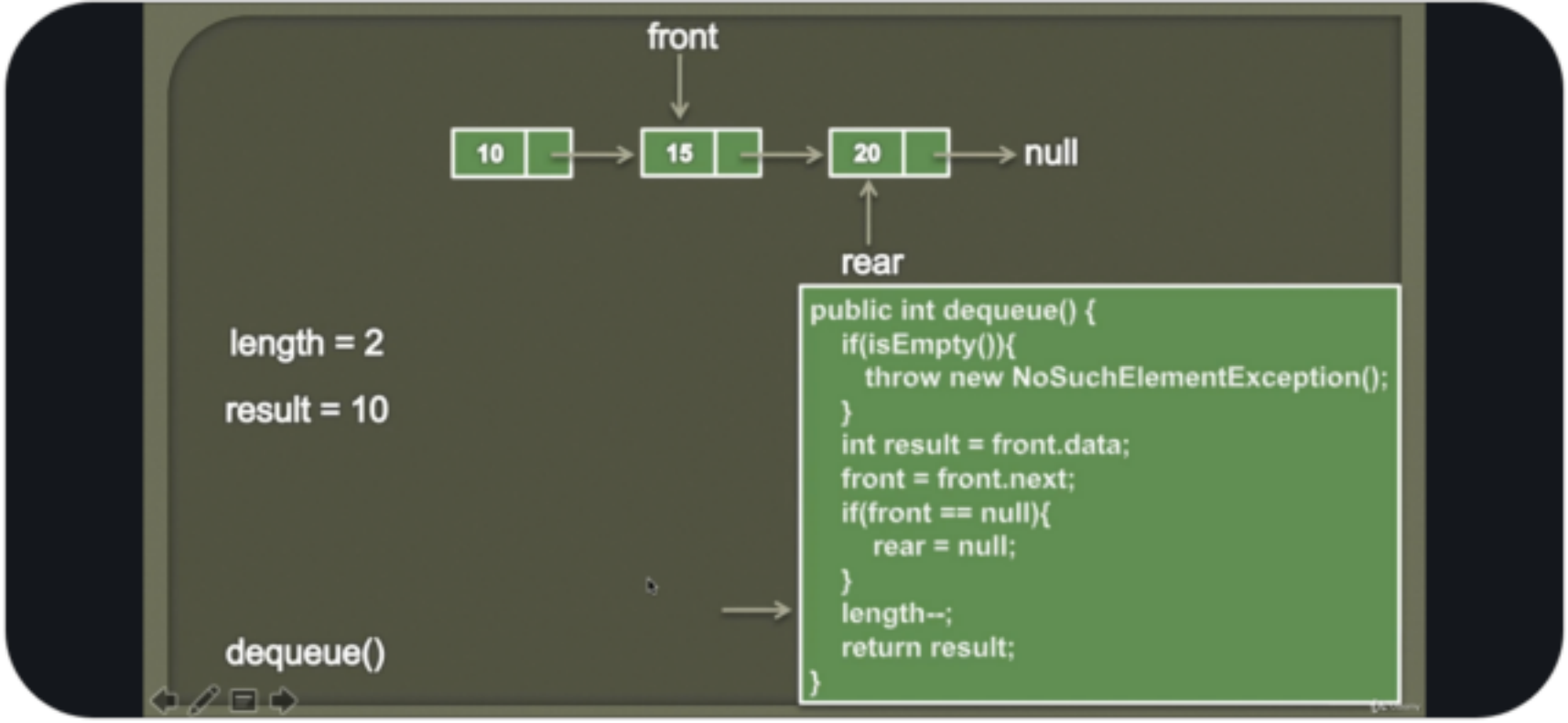
```
Queue<String> animal2 = new
ArrayDeque<>();
//Priority Queue implementation of
Queue
Queue<String> animal 3 = new
PriorityQueue<>();
```

Methods

- boolean add(object) / offer(object)
- poll() - returns null if empty
- peek() - does not removes

How to implement queue





Generate binary numbers from 1 to n:

```
public class GenerateBinary {
    public String[] generateBinary(int
number) {
        String[] result = new
String[number];
        Queue<String> queue = new
LinkedList<String>();
        queue.offer("1");
        for (int i = 0; i < number; i++) {
            result[i] = queue.poll();
            String s1 = result[i] + "0";
            String s2 = result[i] + "1";
            queue.offer(s1);
            queue.offer(s2);
        }
```

```
        return result;
    }
}
```

Priority Queues and Heaps

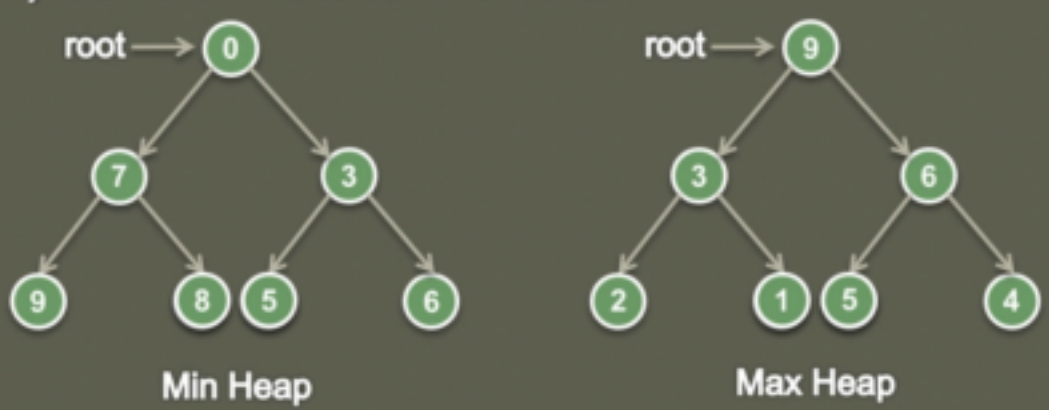
Priority Queue

Priority Queue is a data structure that allows us to find minimum/maximum element among a collection of elements in constant time. It supports following operations –

- insert(key) – Insert a key into the priority queue.
- deleteMax() / deleteMin() – return and remove largest / smallest key.
- getMax() / getMin() – return largest/smallest key.

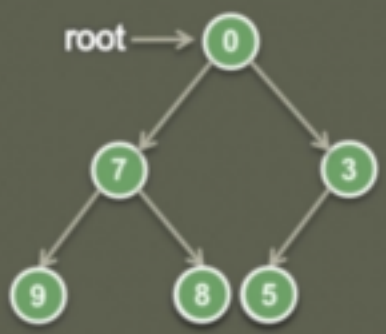
Binary Heap

The binary heap is a data structure that helps us in implementing Priority Queue operations efficiently. A binary heap is a complete binary tree in which each node value is >= (or <=) than the values of its children.

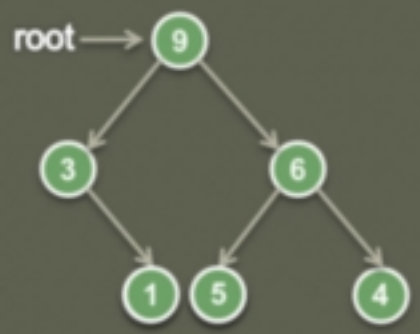


Complete Binary Tree

A complete binary tree is a binary tree where all levels are completely filled except last level and last level has nodes in such a way that left side is never empty.



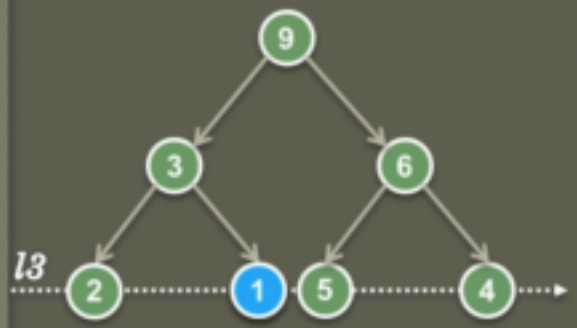
Complete Binary Tree



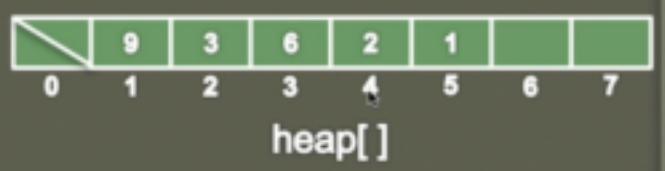
Incomplete Binary Tree

Representation of a Binary Heap

- Binary Heaps usually are implemented using arrays.
- The first entry of array is taken as empty.
- As Binary Heaps are complete binary tree, the values are stored in array by traversing tree level by level from left to right.



Max Heap



heap[ ]

Parent - Child calculations

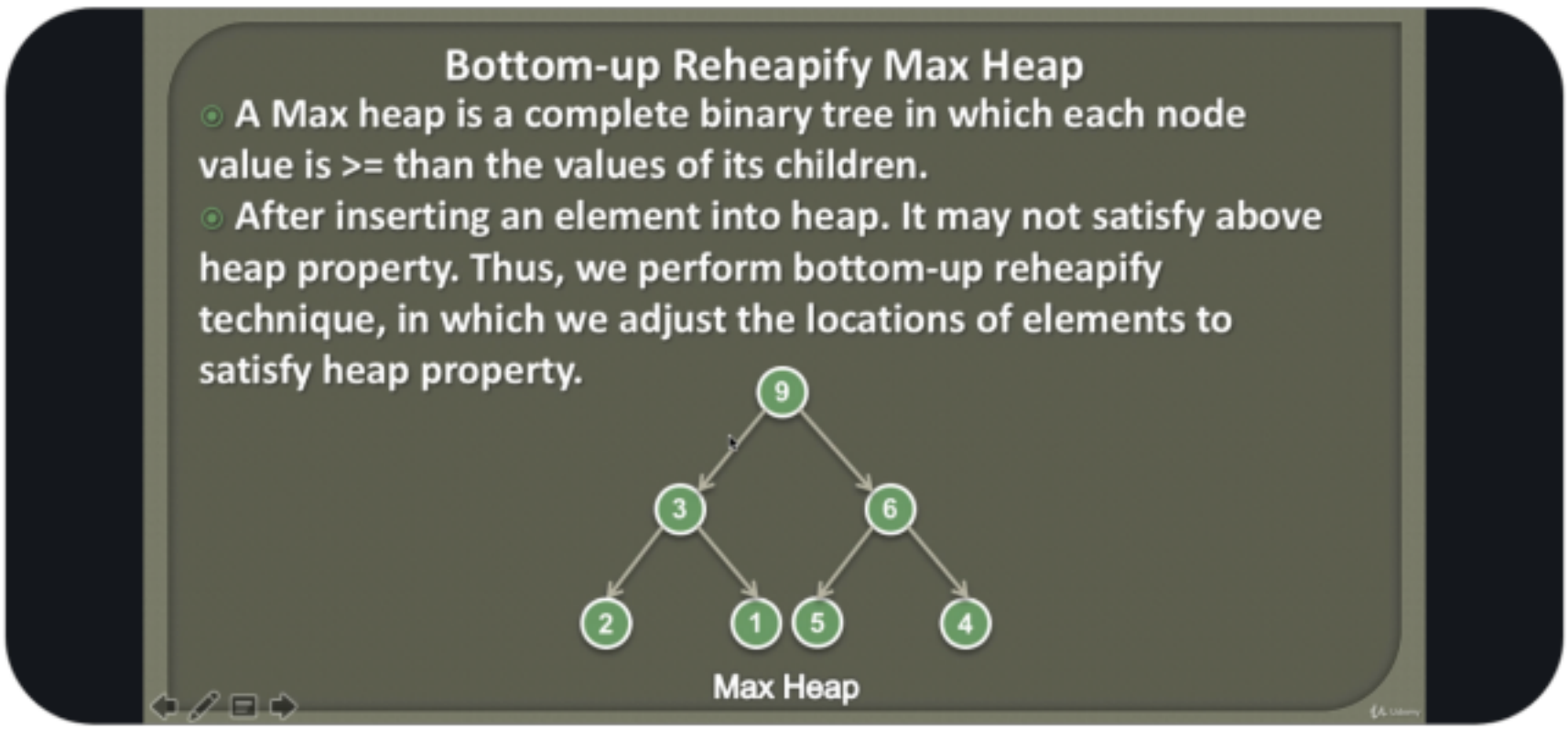
Max Heap

Children of 1st index = 2, 3
Children of 2nd index = 4, 5
Children of 3rd index = 6, 7

Children of k^{th} index = 2*k, 2*k + 1

Parent of 7th index = 3
Parent of 6th index = 3
Parent of 5th index = 2

Parent of k^{th} index = k/2



- Unlike normal queues, priority queue elements are retrieved in sorted order.
- Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.
- It is important to note that the elements of a priority queue may not be sorted. However, elements

are always retrieved in sorted order.

- ≡ ○ we can customize the ordering of elements with the help of the Comparator interface
- ≡ ○ A binary heap is a data structure, which looks similar to a complete binary tree
- ≡ ○ A binary heap can be classified further as either a max-heap or a min-heap based on the ordering property.

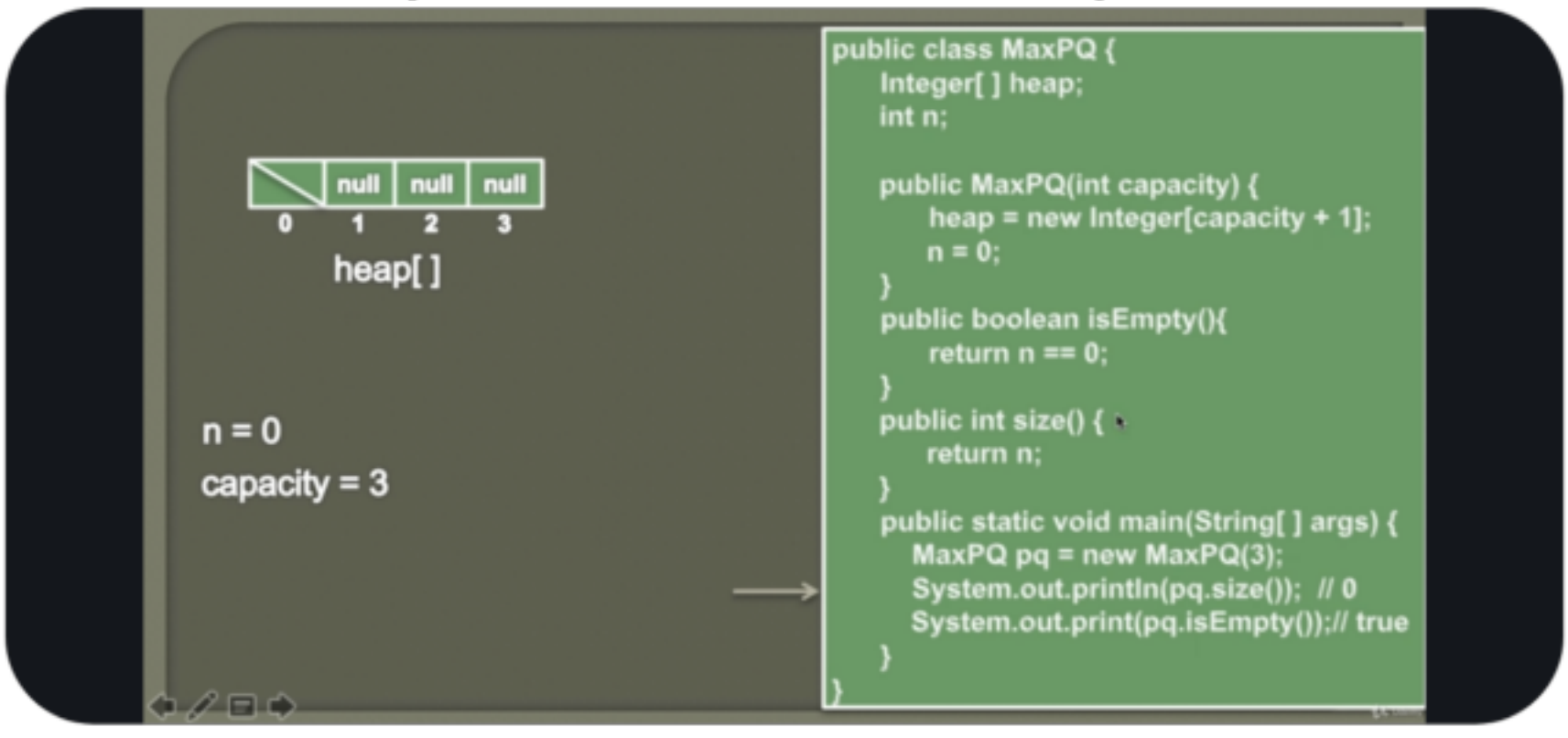
Methods

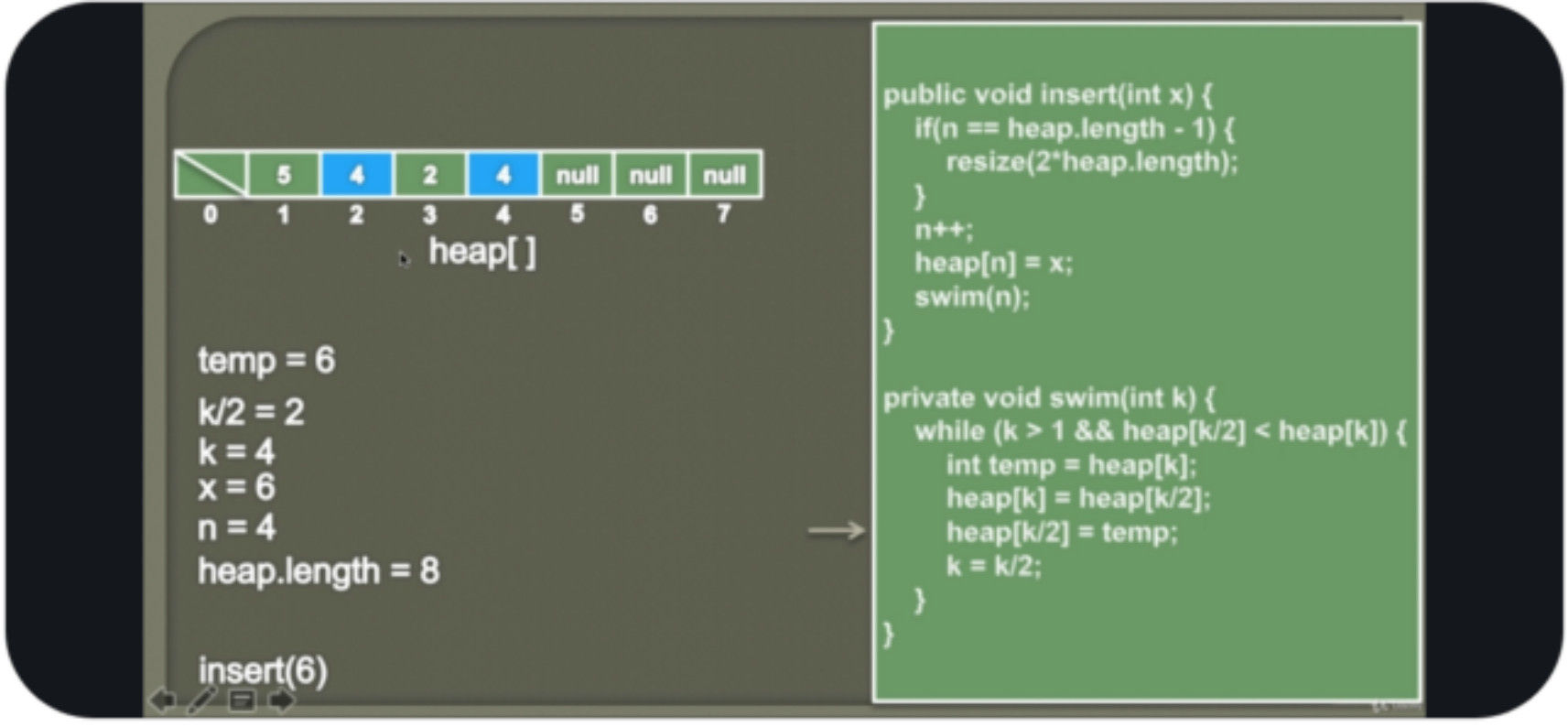
- ≡ ○ add() - if queue is full throws exception
- ≡ ○ offer() - return false if queue is full
- ≡ ○ peek() - returns head of the queue
- ≡ ○ remove() - removes the specified element from the queue
- ≡ ○ poll() - returns and removes the

head of the queue

- ≡ ○ iterator()
- ≡ ○ contains(element)
- ≡ ○ size()
- ≡ ○ toArray()
- ≡ ○ isEmpty()
- ≡ ○ new PriorityQueue(size, (a,b) -> Integer.compare(a.id, b.id))
- ≡ ○ addAll(map.entrySet())

How to implement Max Heap

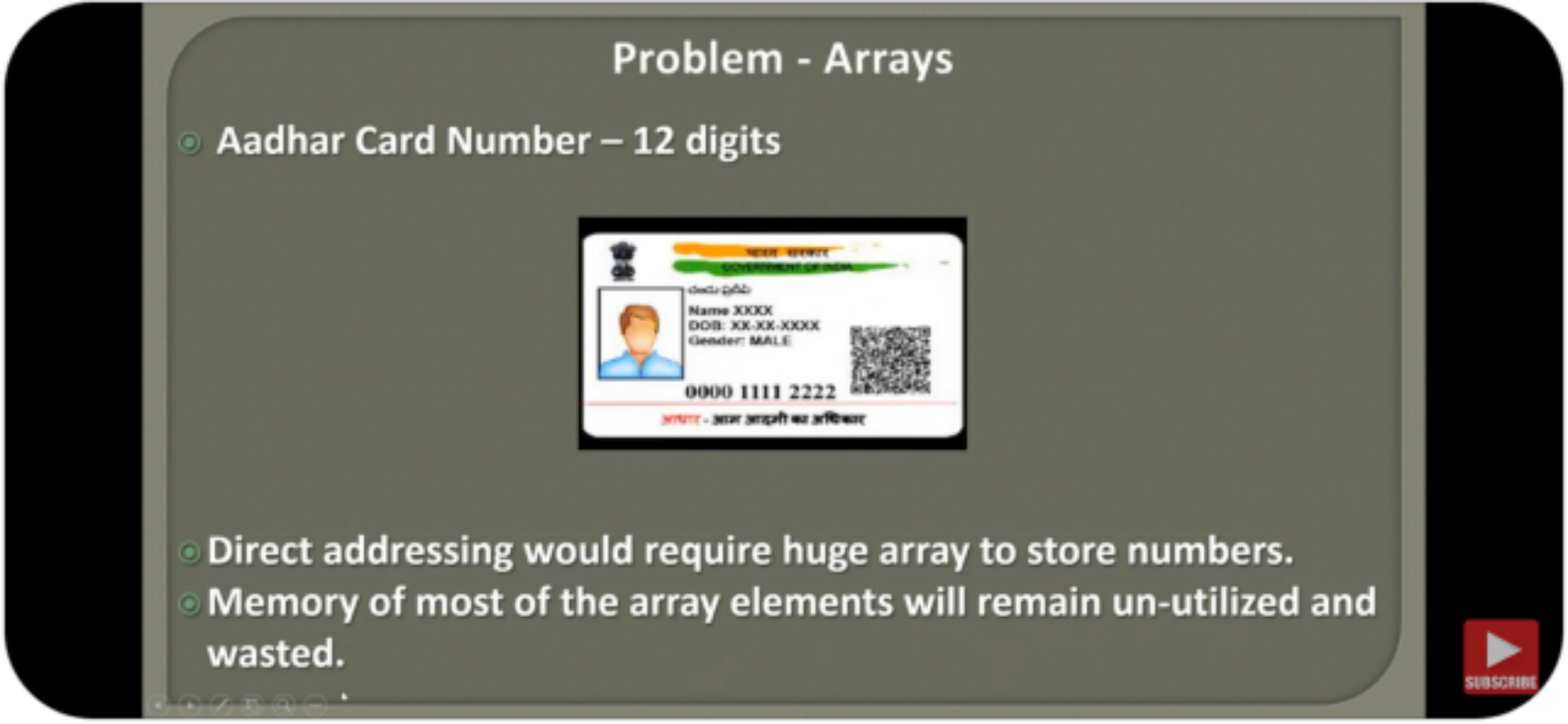




Hash Table

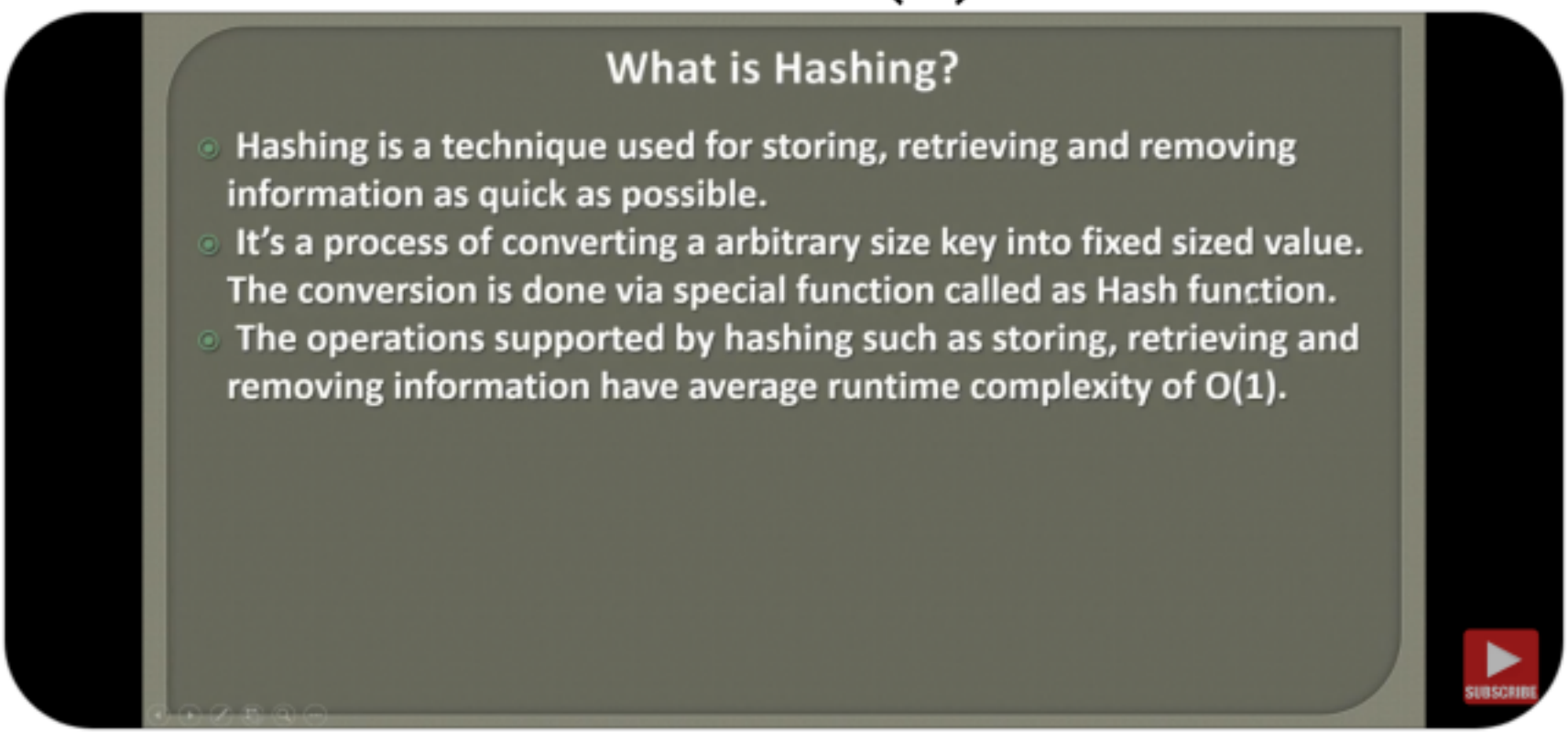
(Fast insertion, searching, deletion)

- Searching an element in an array through linear search takes o(n) time and through binary search it takes O(log n)
- Is there any data structure which gives O(1) for searching an element?
- Array is one such data structure which gives O(1) searches using indexes.



Hashing

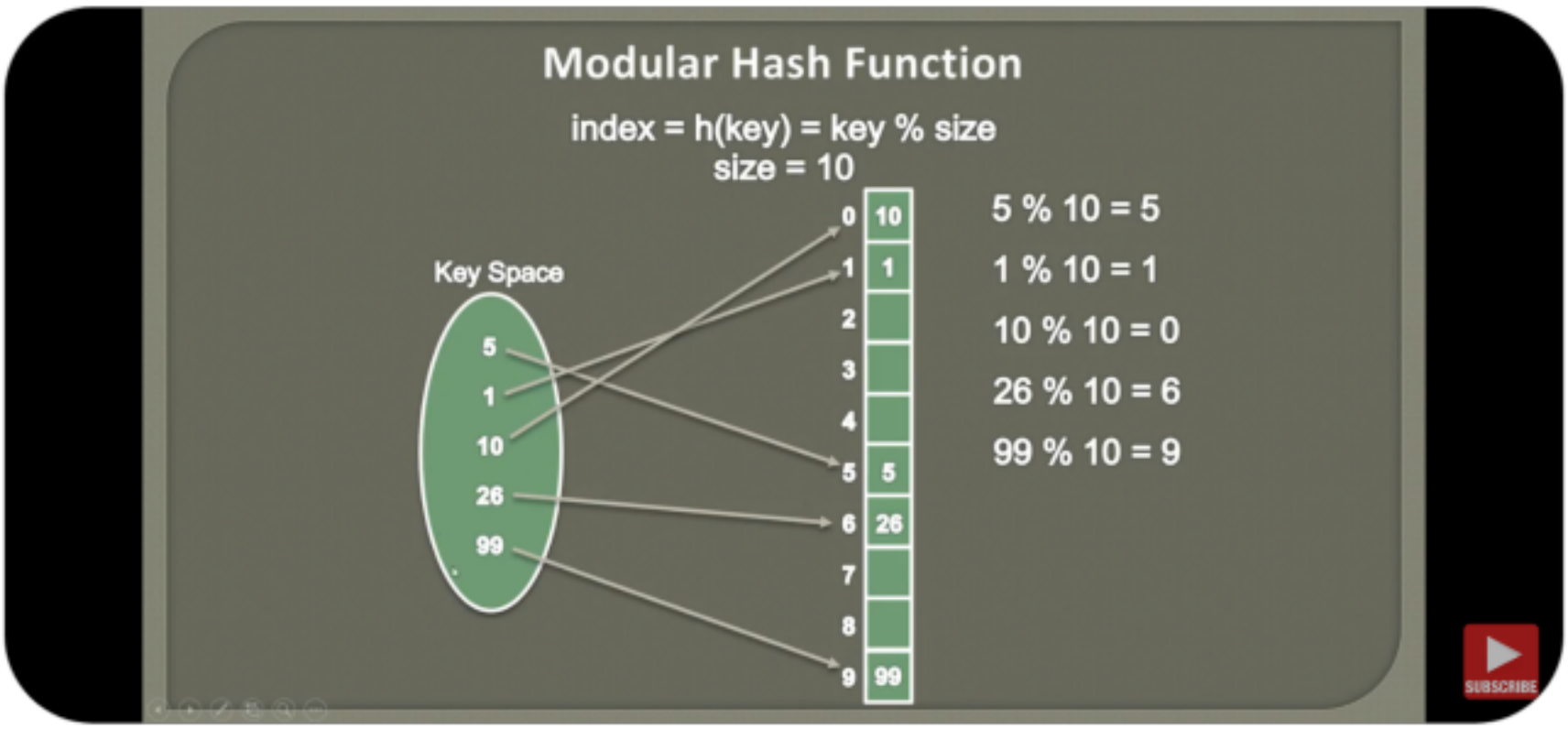
≡ ○ Because of above issues with array we use hashing approach to store, retrieve and remove elements with O(n) time



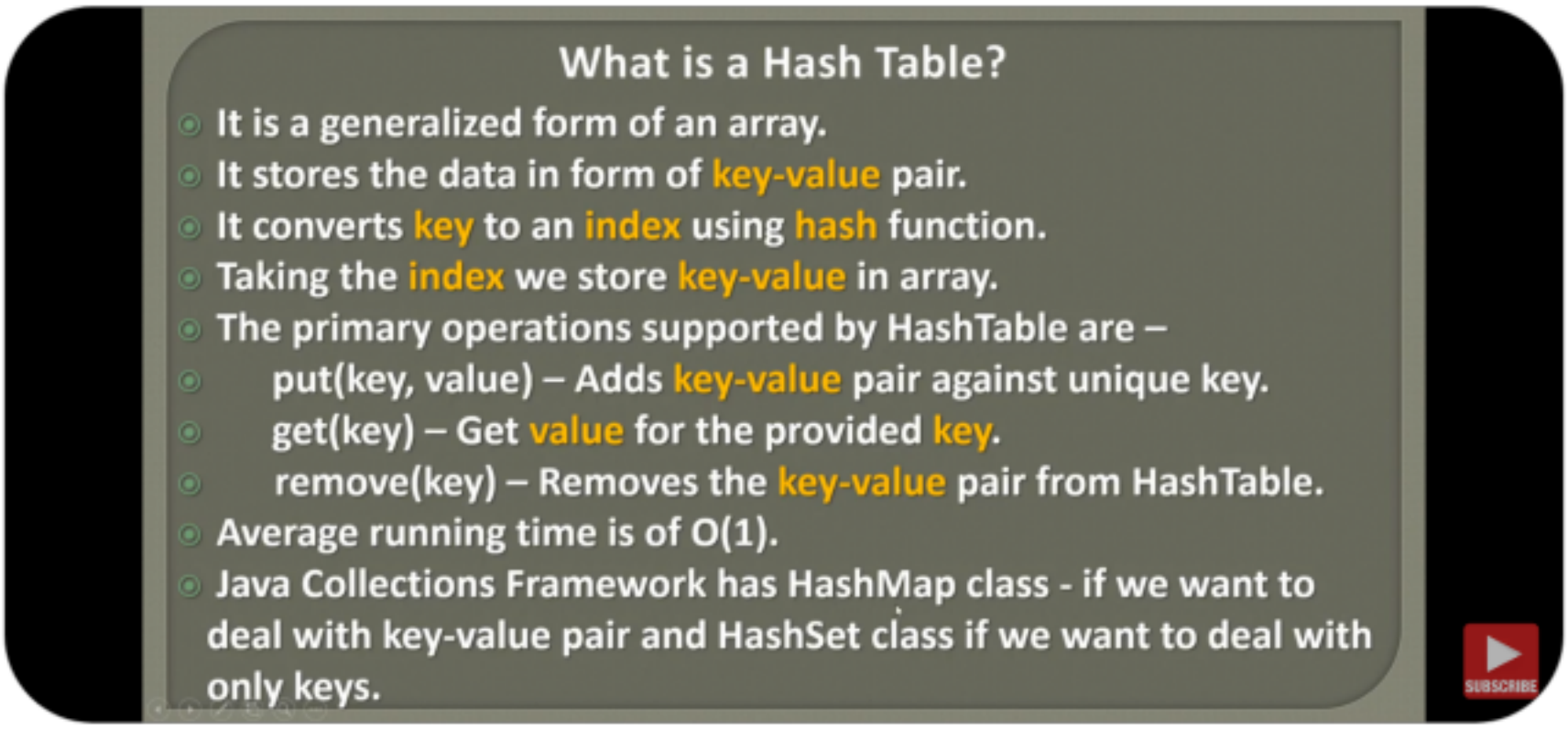
Hash Function

≡ ○ One such hash function is

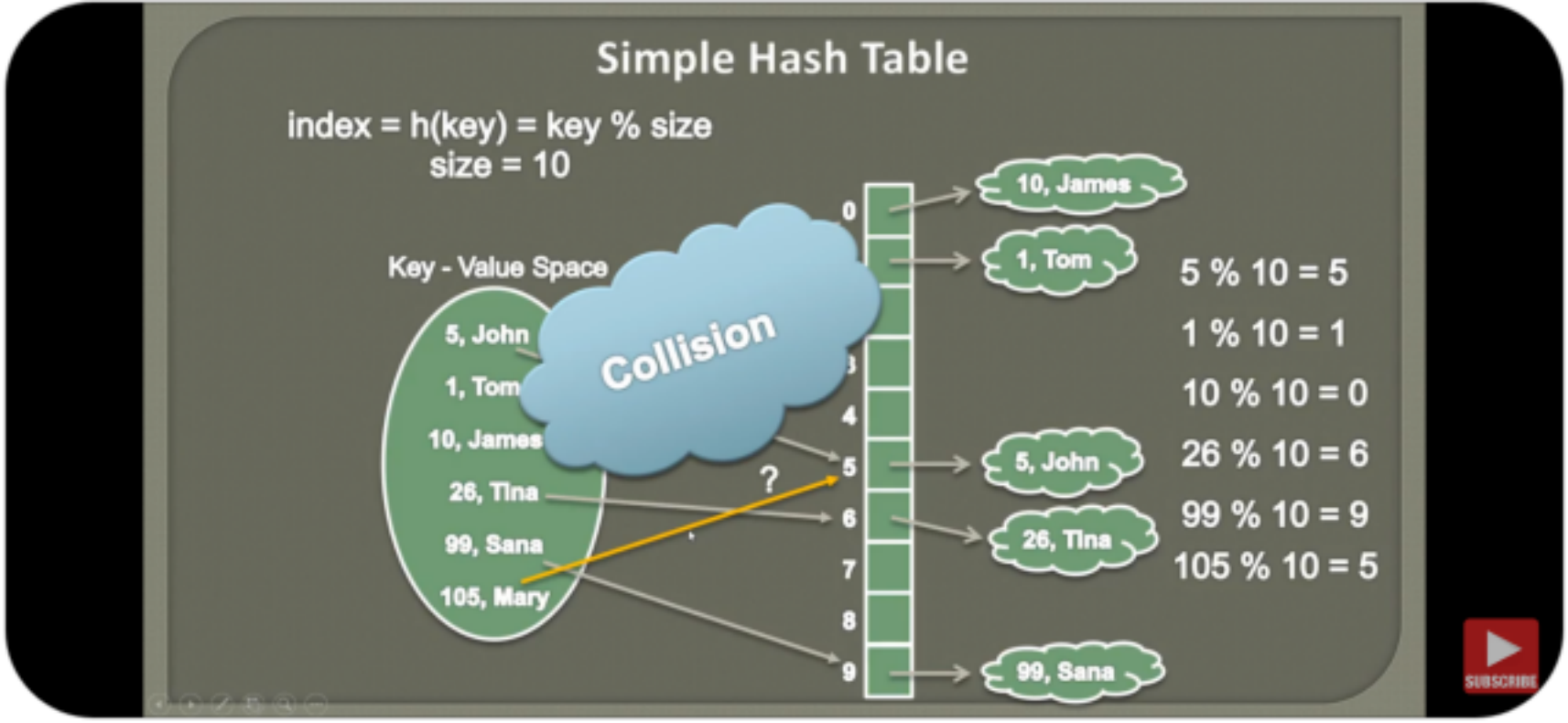
Modular hash function which can provide index value as index=key%array.length. This will helps to create smaller array instead of bigger one



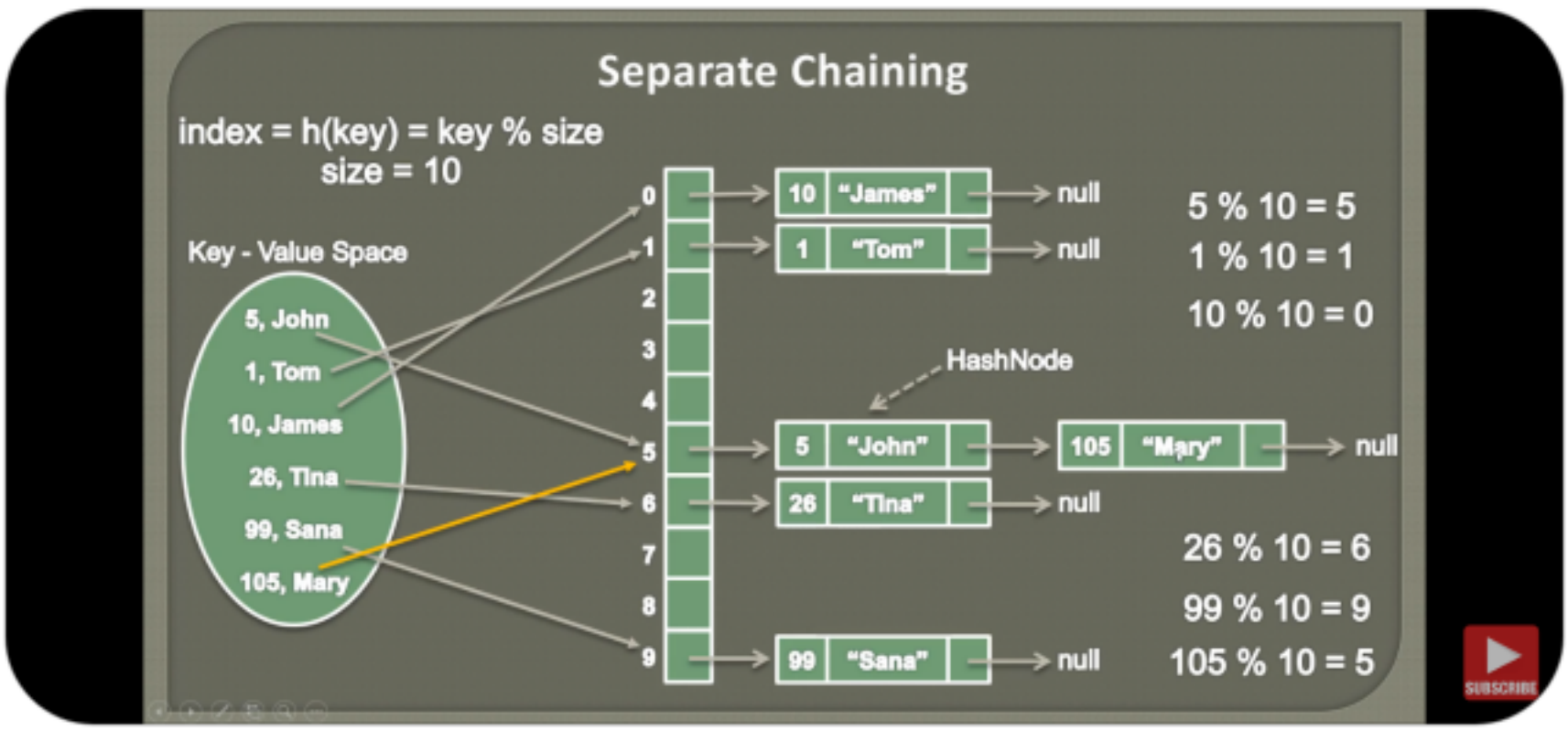
HashTable



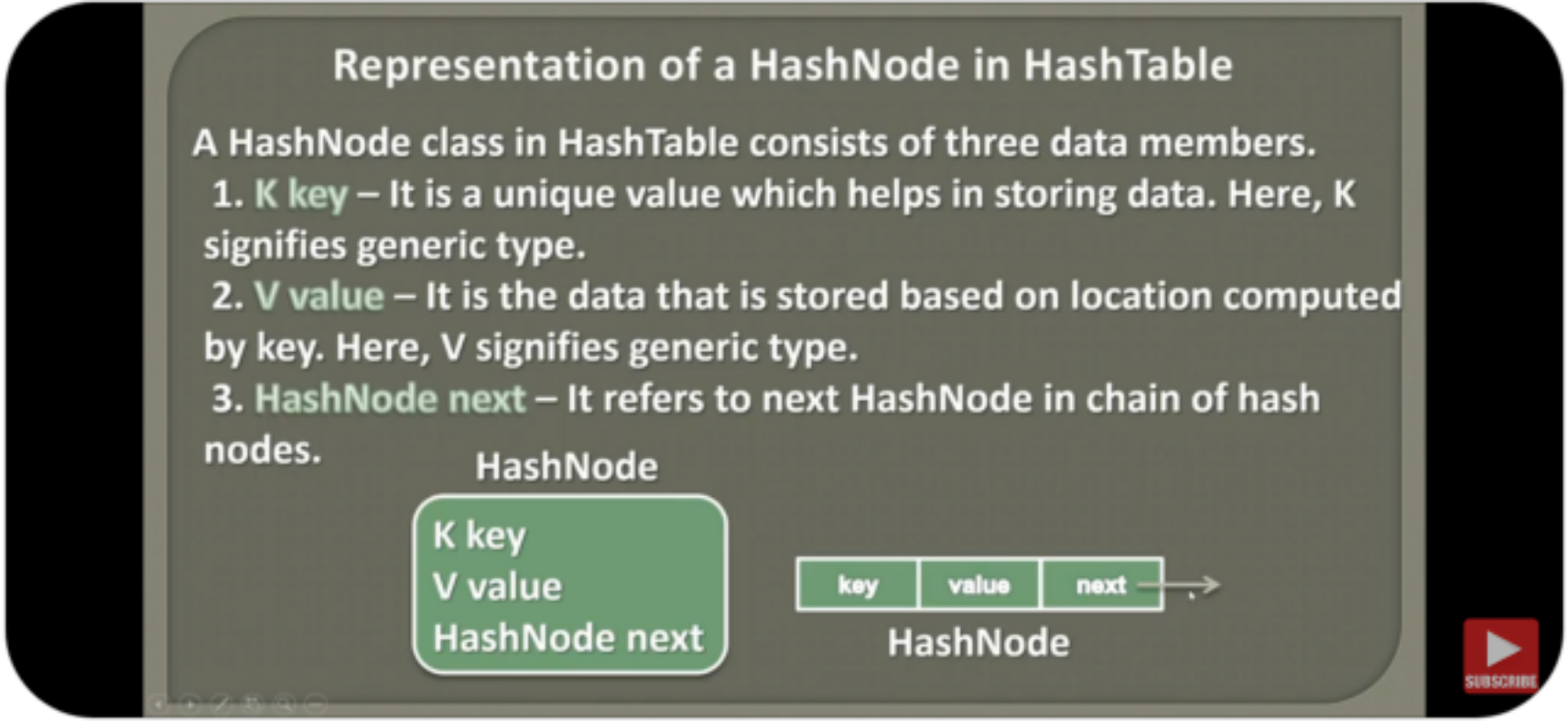
Collision



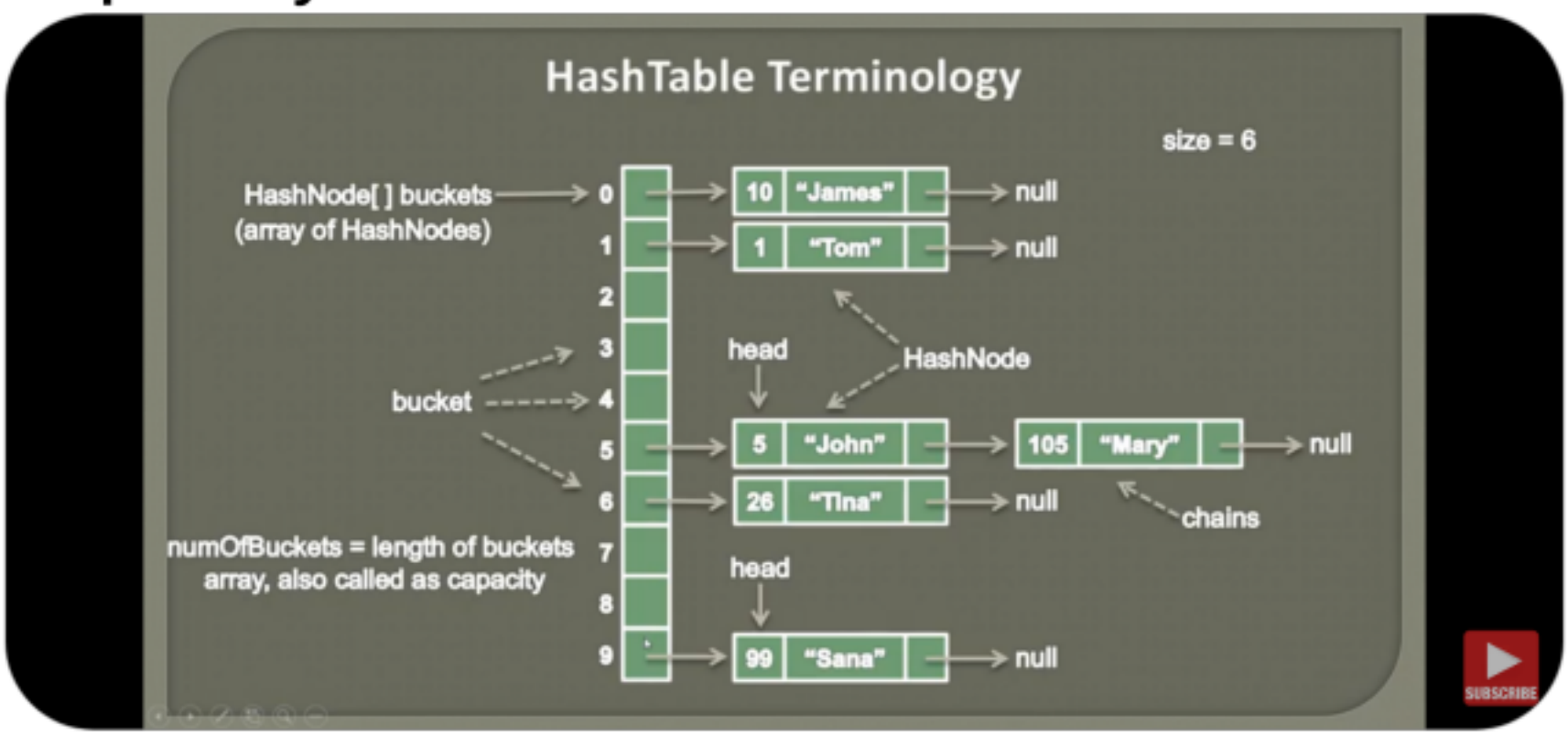
How to solve collision



HashNode



Hash Table terminology. Size=6, capacity=10



Hash Map

- It contains values based on the key.
- It contains only unique keys.
- It may have one null key and

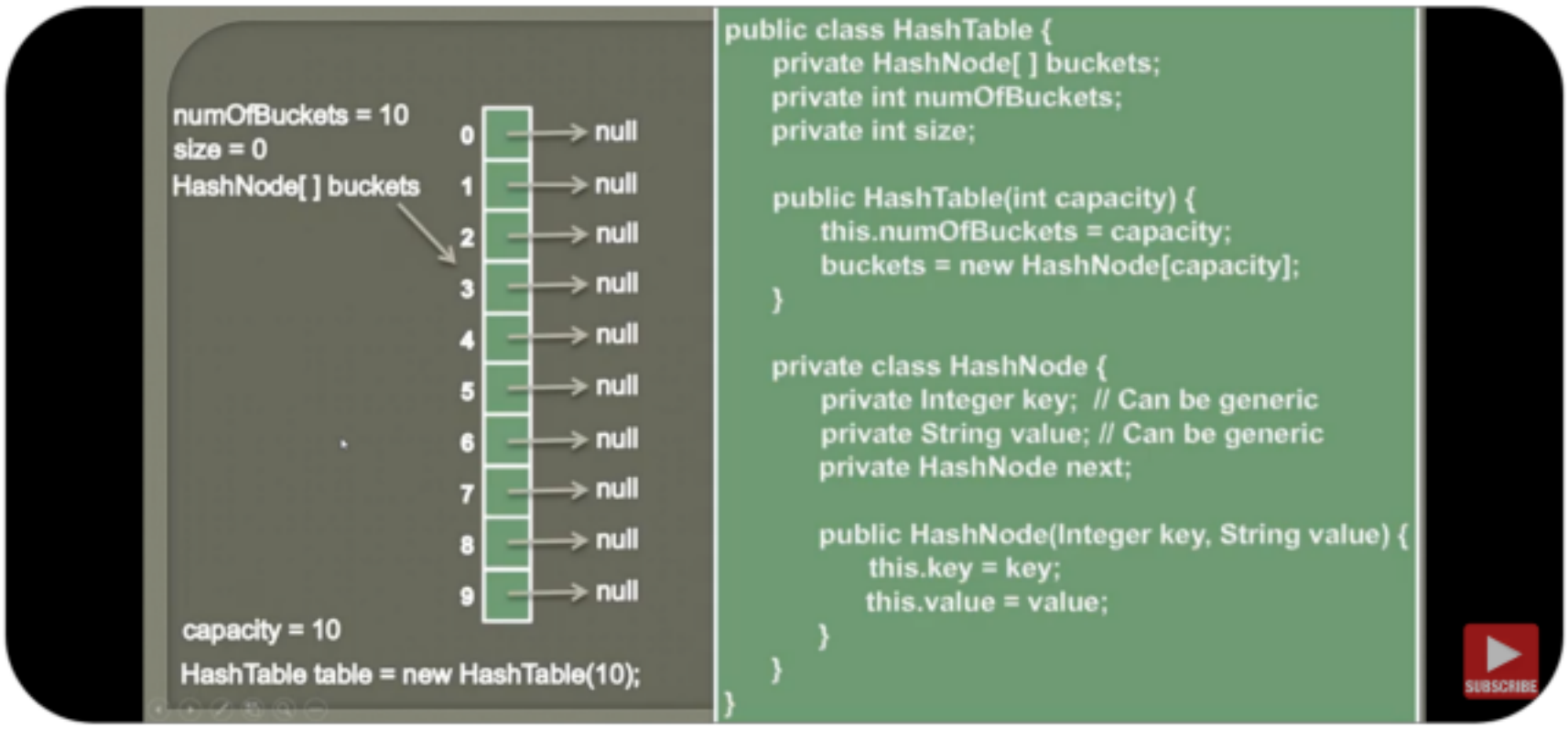
multiple null values.

- ○ It is non synchronized.
- ○ It maintains no order.
- ○ The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

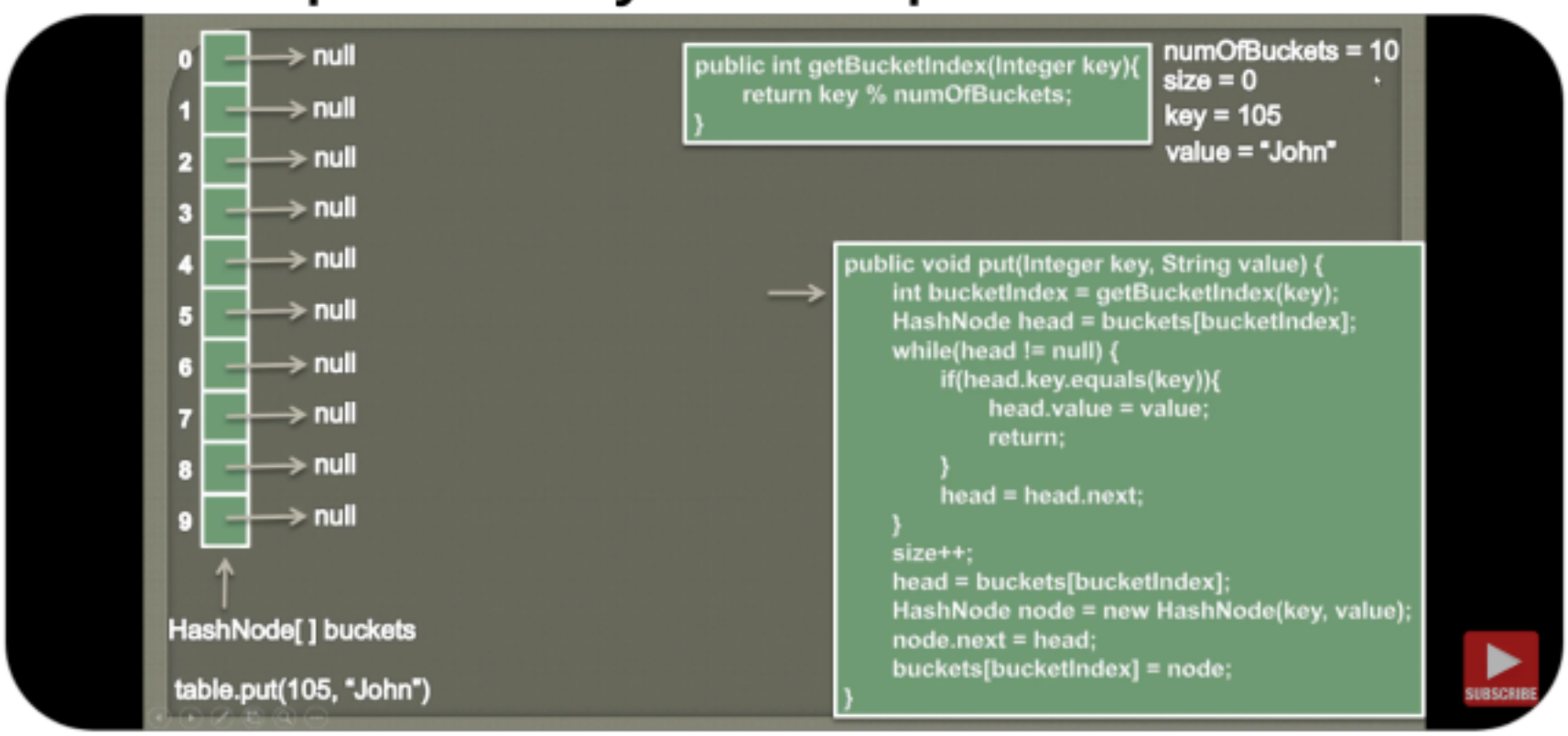
Hash Set

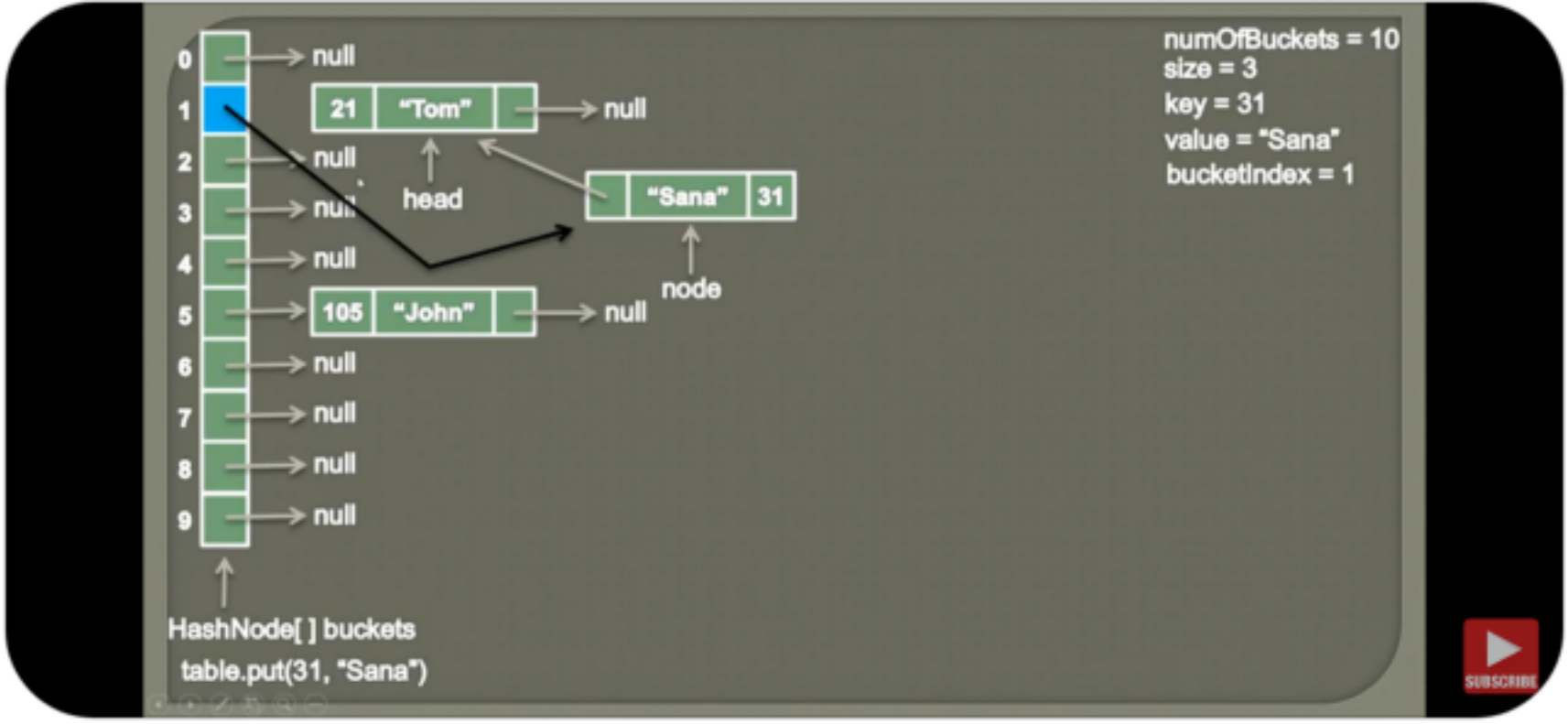
- ○ Stores the elements by using a mechanism called hashing.
- ○ Contains unique elements only.
- ○ Allows null value.
- ○ Is non synchronized.
- ○ Doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- ○ HashSet is the best approach for search operations.
- ○ The initial default capacity of HashSet is 16, and the load factor is 0.75.

Implementation of Hash Table

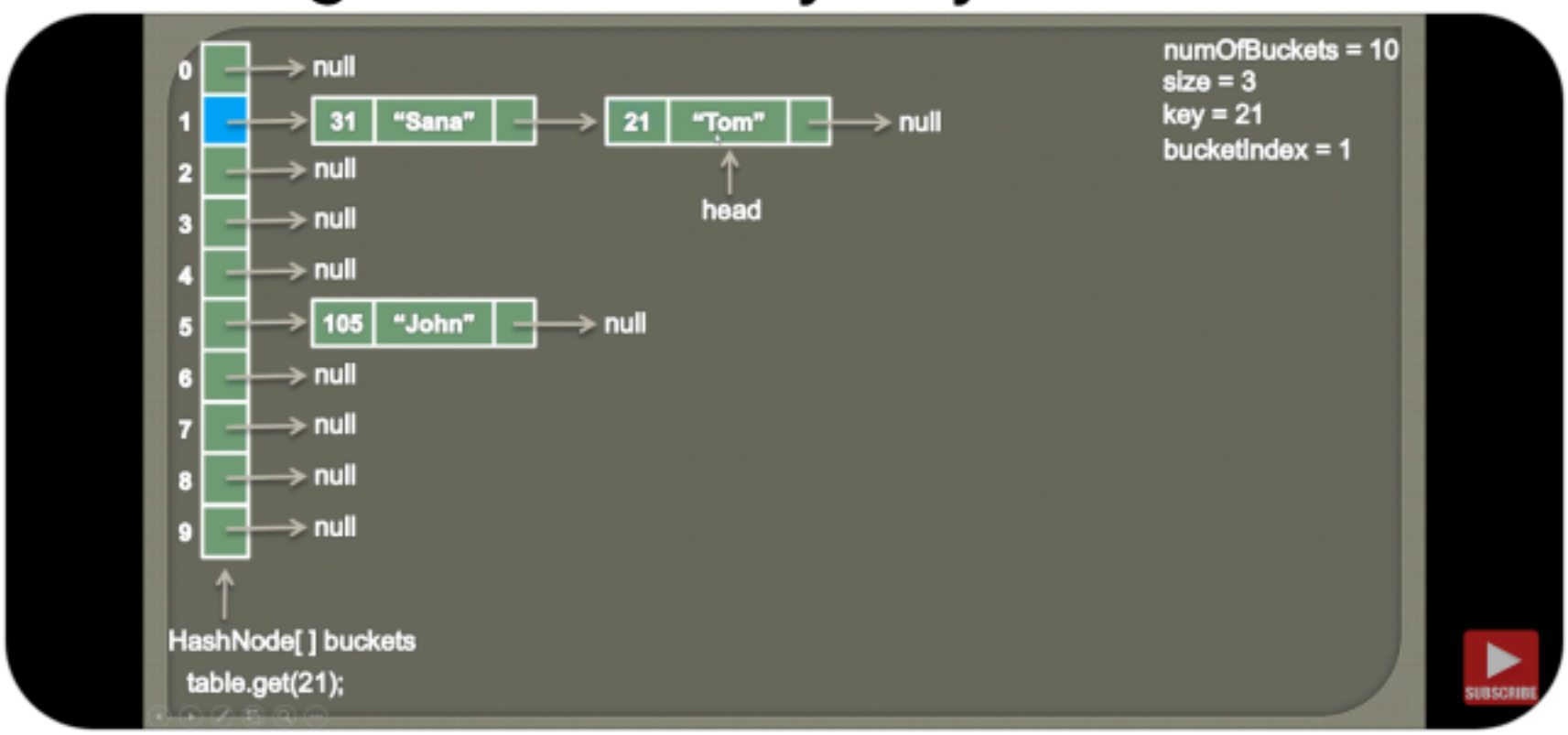


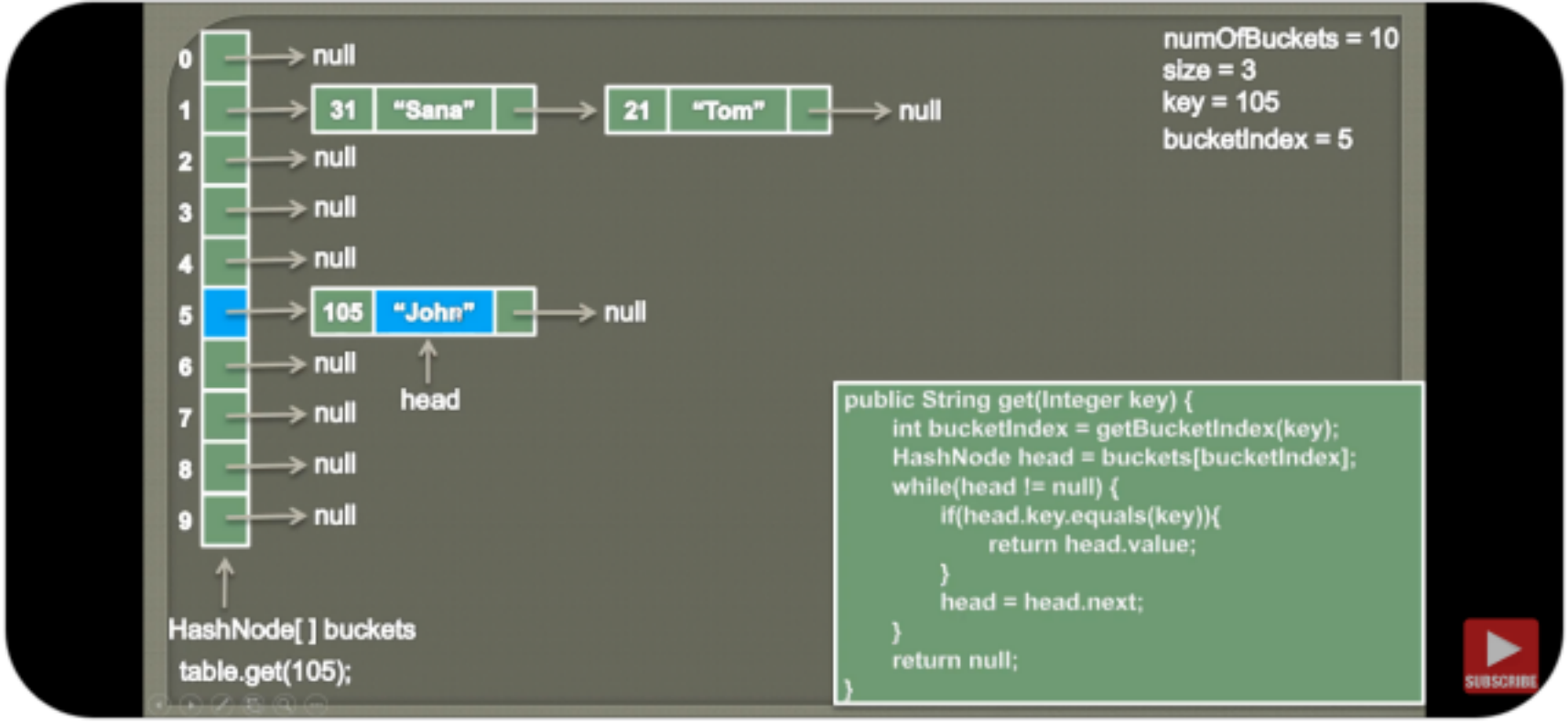
How to put a key value pair





How to get a value by key





How to remove a key from Hash Table

Use cases

- Used to store each character in a String and its corresponding counts of occurrences
- Also, it can be used to store each character in a String and its corresponding index position

Methods:

Hash Map

- put(key, value)
- putAll(Map map)
- get(key)

≡ ○ getOrDefault(key, defaultValue)

≡ ○ remove(key)

≡ ○ remove(key, value)

≡ ○ Set entrySet()

≡ ○ Set keySet()

≡ ○ Collection values()

≡ ○ isEmpty()

≡ ○ clear()

≡ ○ size()

≡ ○ containsKey(key)

≡ ○ containsValue(value)

≡ ○ replace(key, value)

≡ ○ for(Map.Entry m : map.entrySet())
{
System.out.println(m.getKey()
+" "+m.getValue());
}

≡ ○ for (String State : map.keySet()) {
System.out.println("State: " +
State); }

≡ ○ for (String Capital :

map.values()) {
System.out.println("Capiatl: " +
Capital);
}

≡ ○ Iterator <Integer> it =
hm.keySet().iterator;

≡ ○ map.forEach((k,v) ->
System.out.println("Company: "+
k + ", Net worth: " + v));
}

Hash Set

≡ ○ add(E e)

≡ ○ contains(key o)

≡ ○ clear()

≡ ○ isEmpty()

≡ ○ size()

≡ ○ remove(key o)

≡ ○ iterator()

≡ ○ For each loop

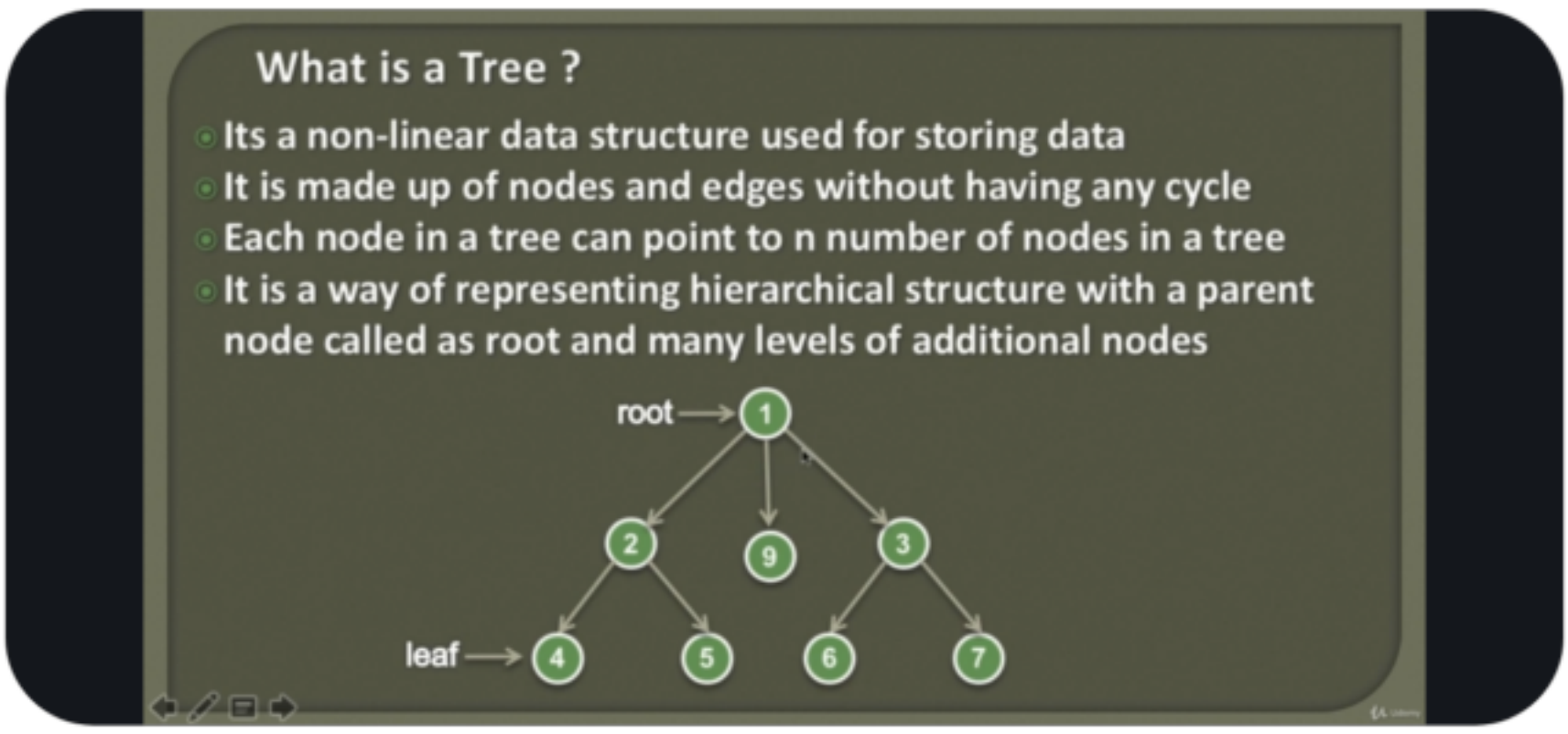
≡ ○ new HashSet(list)

Tree

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data

and the link or reference of other nodes that can be called children.

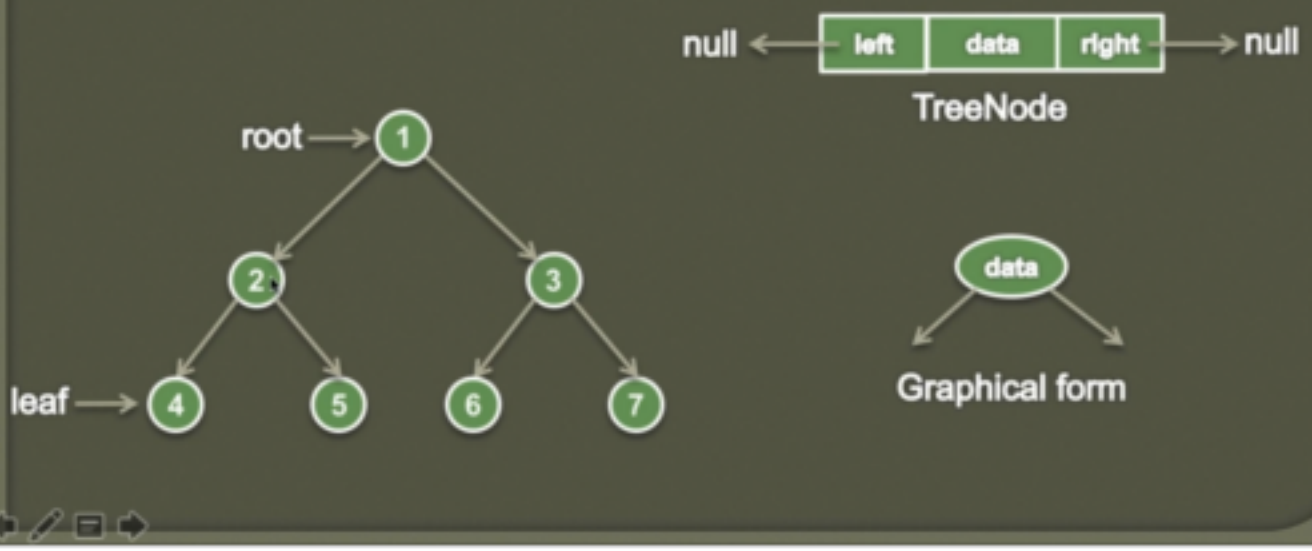
- The bottom most nodes are called as leaf nodes. So there are root node, parent node, child node, sibling node and leaf node



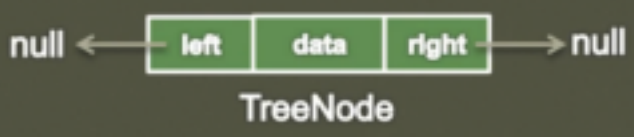
Binary Tree

What is a Binary Tree ?

- A tree is called as Binary Tree, if each node has zero, one or two children.



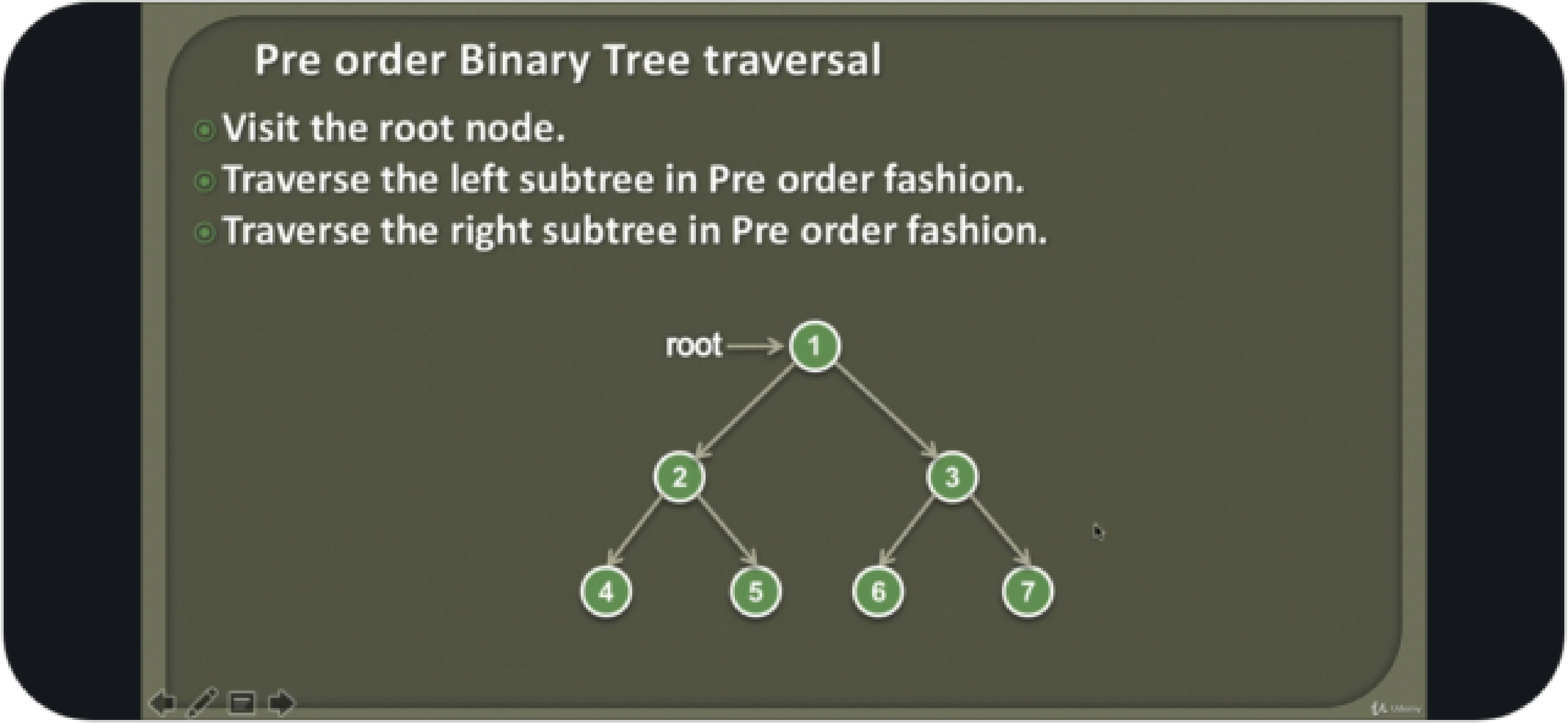
Structure of TreeNode in a Binary Tree

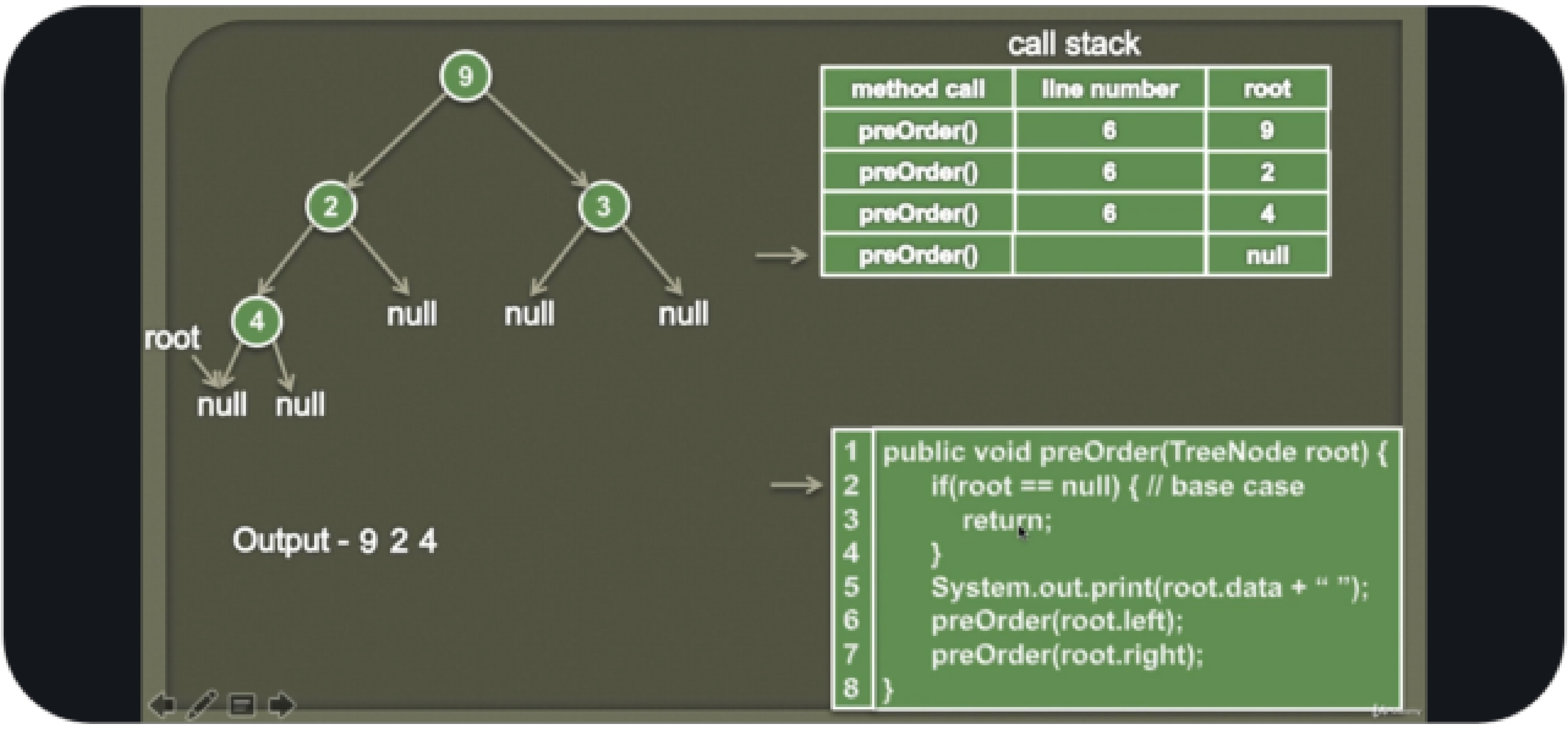


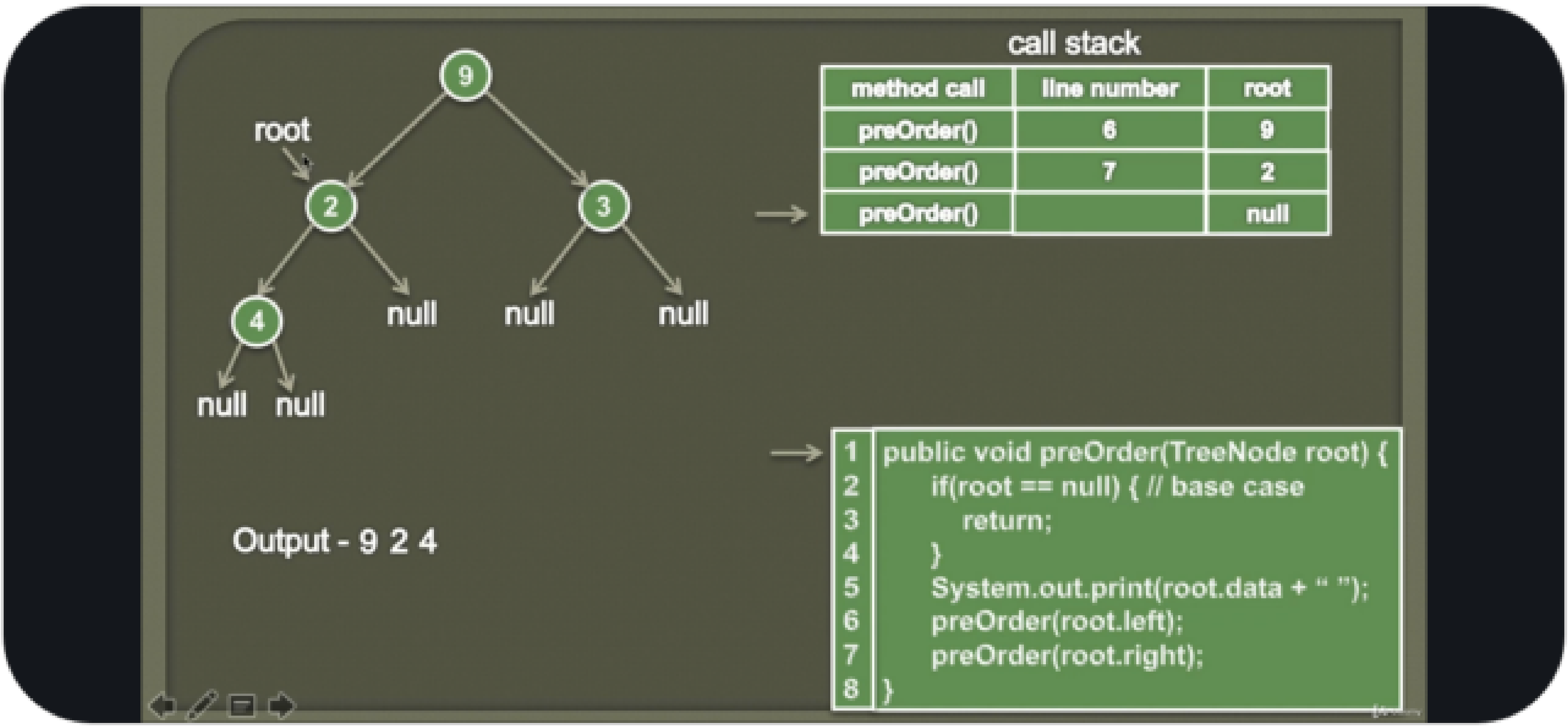

```
public class TreeNode {
    private int data;  // Generic type
    private TreeNode left;
    private TreeNode right;

    public TreeNode(int data) {
        this.data = data;
    }
}
```

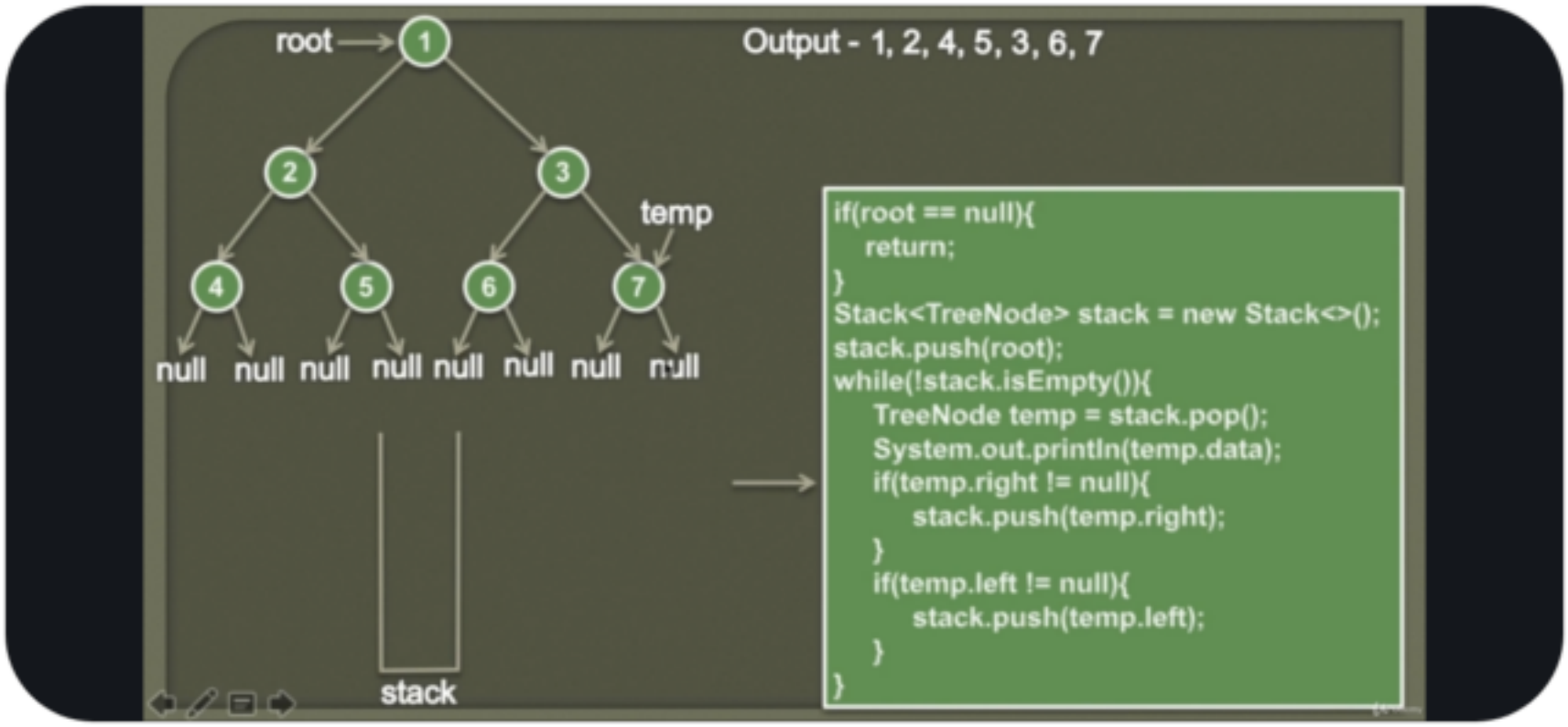
Recursive Pre-Order Traversal


Pre order Binary Tree traversal
Visit the root node.
Traverse the left subtree in Pre order fashion.
Traverse the right subtree in Pre order fashion.
root
1
2
3
4
5
6
7

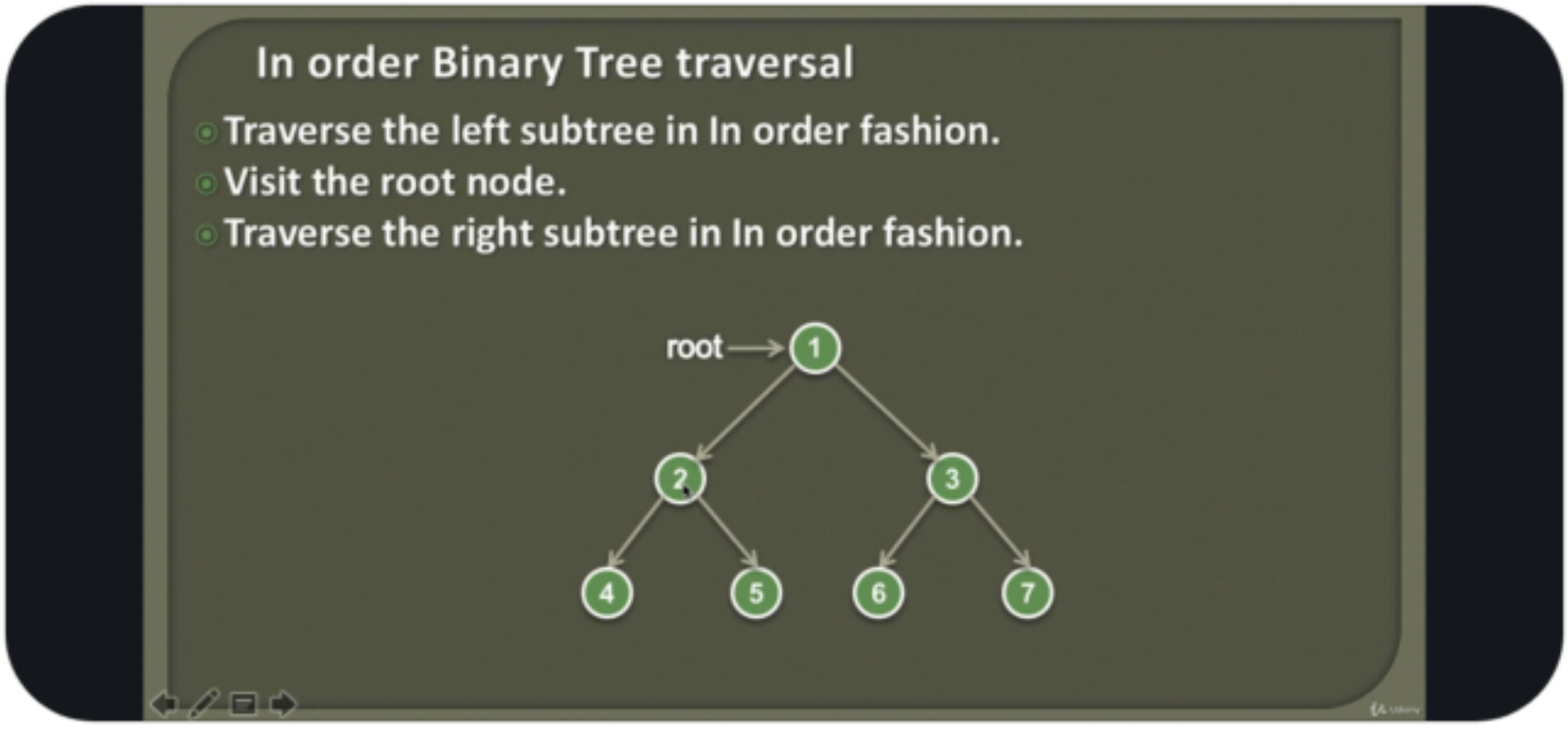

9
2
3
4
null
null
null
null
null
root
Output - 9 2 4
call stack
method call | line number | root
preOrder() | 6 | 9
preOrder() | 6 | 2
preOrder() | 6 | 4
preOrder() | | null
public void preOrder(TreeNode root) {
if(root == null) { // base case
return;
}
System.out.print(root.data + " ");
preOrder(root.left);
preOrder(root.right);
}

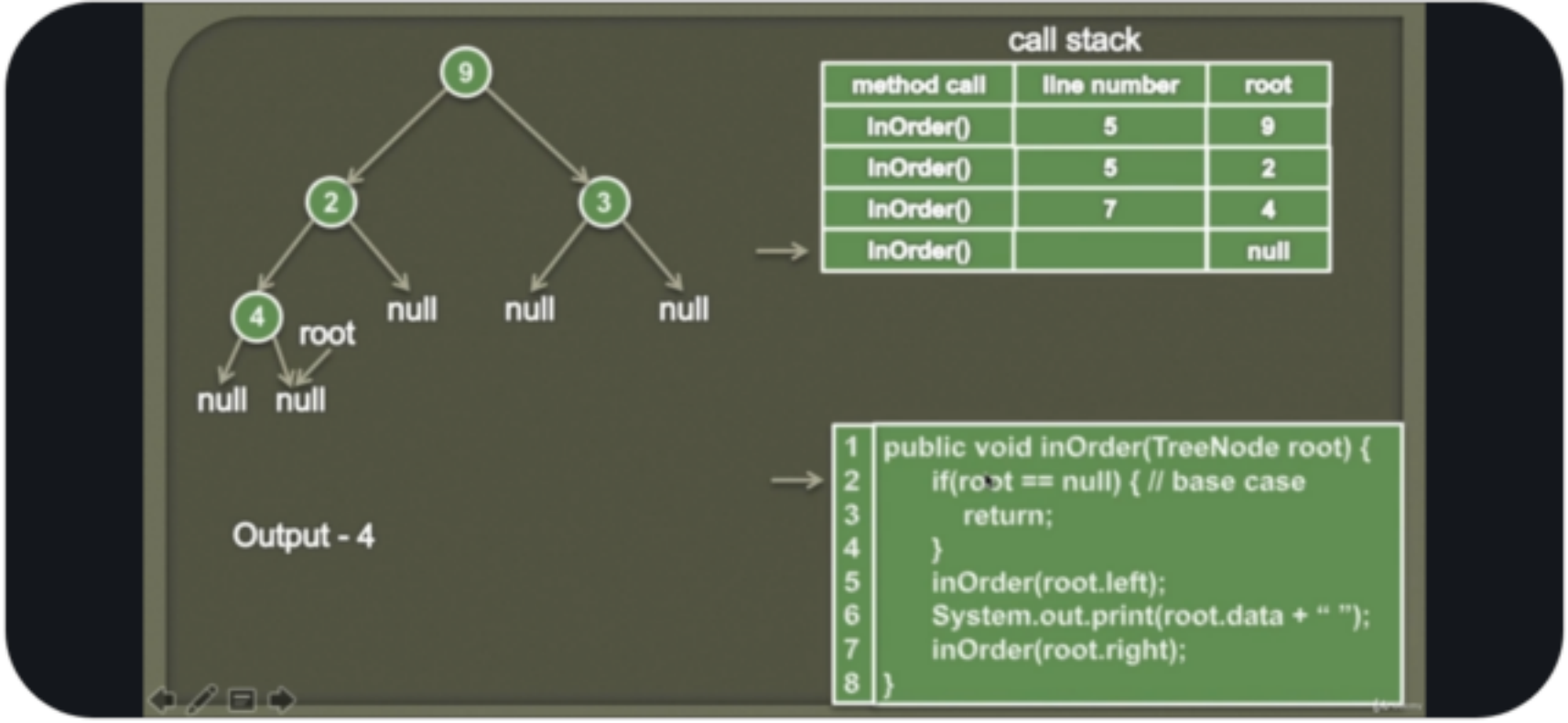

9
root
2
3
4
null
null
null
null
null
Output - 9 2 4
call stack
method call | line number | root
preOrder() | 6 | 9
preOrder() | 7 | 2
preOrder() | | null
public void preOrder(TreeNode root) {
if(root == null) { // base case
return;
}
System.out.print(root.data + " ");
preOrder(root.left);
preOrder(root.right);
}

Iterative Pre-Order traversal using stack

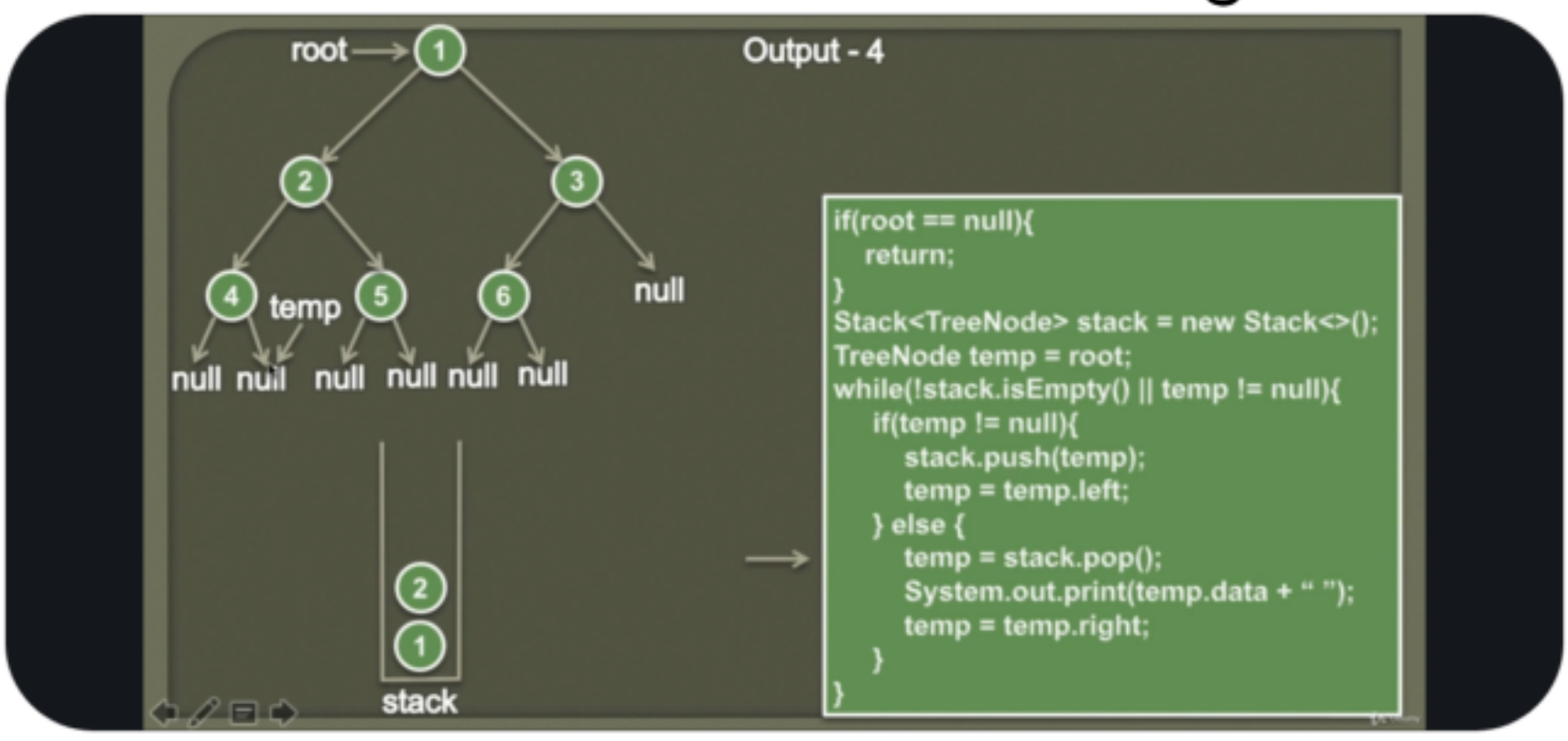


Recursive In-order traversal





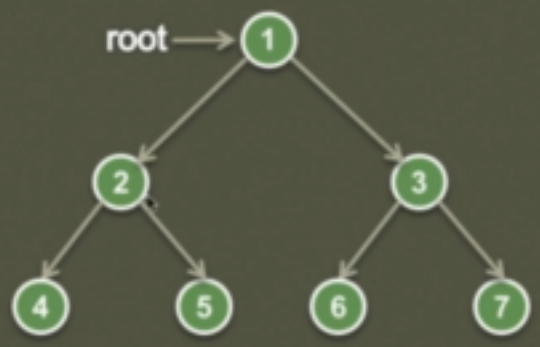
Iterative In-Order traversal using stack

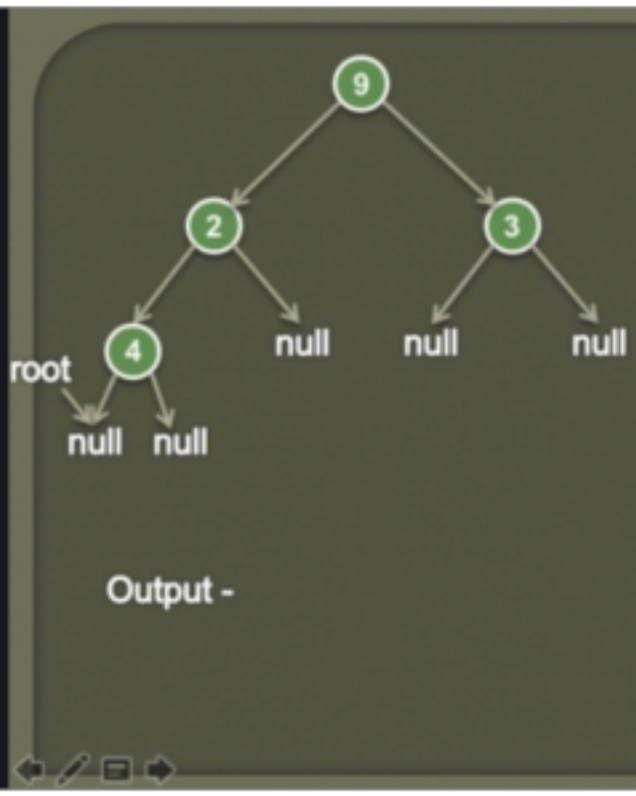


Recursive In-order traversal

Post order Binary Tree traversal

- Traverse the left subtree in Post order fashion.
- Traverse the right subtree in Post order fashion.
- Visit the node.



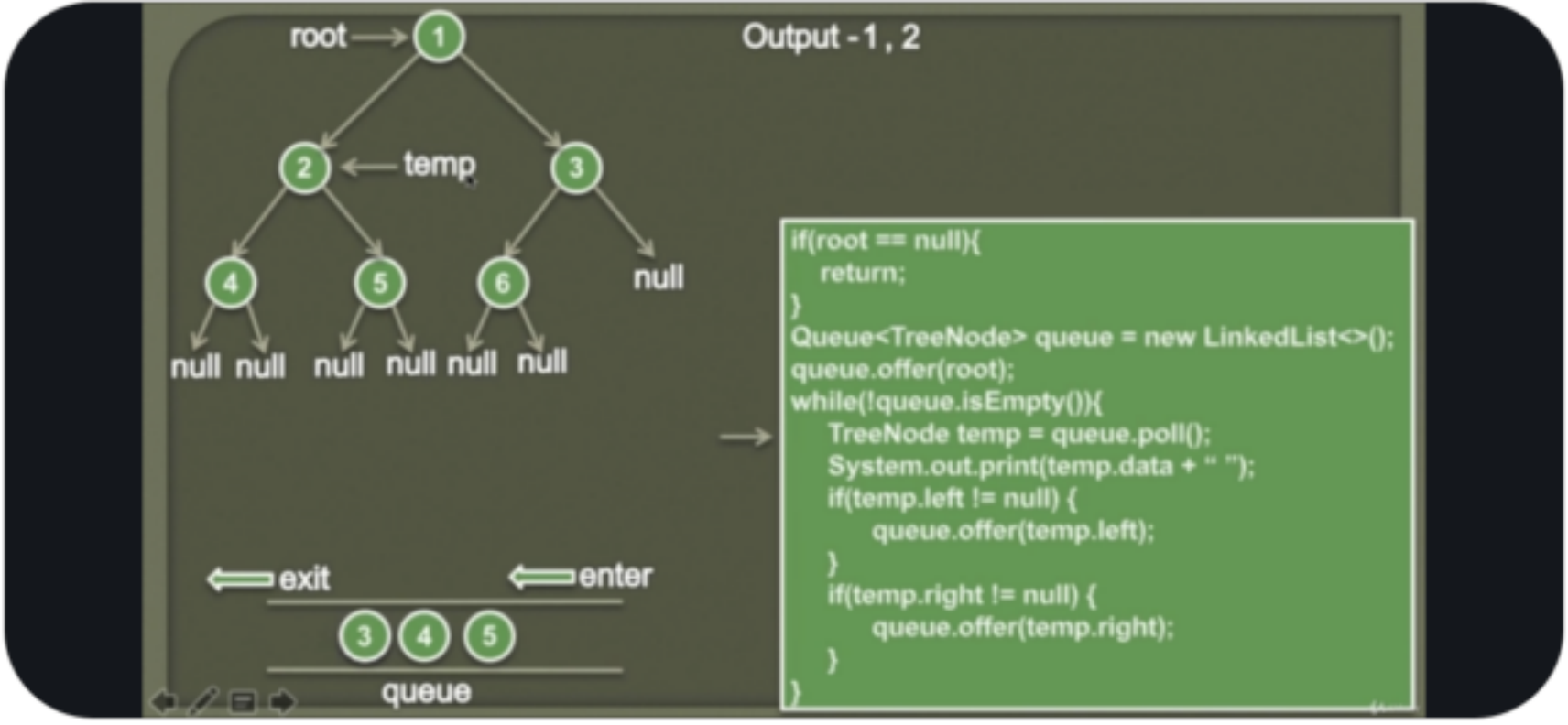


call stack

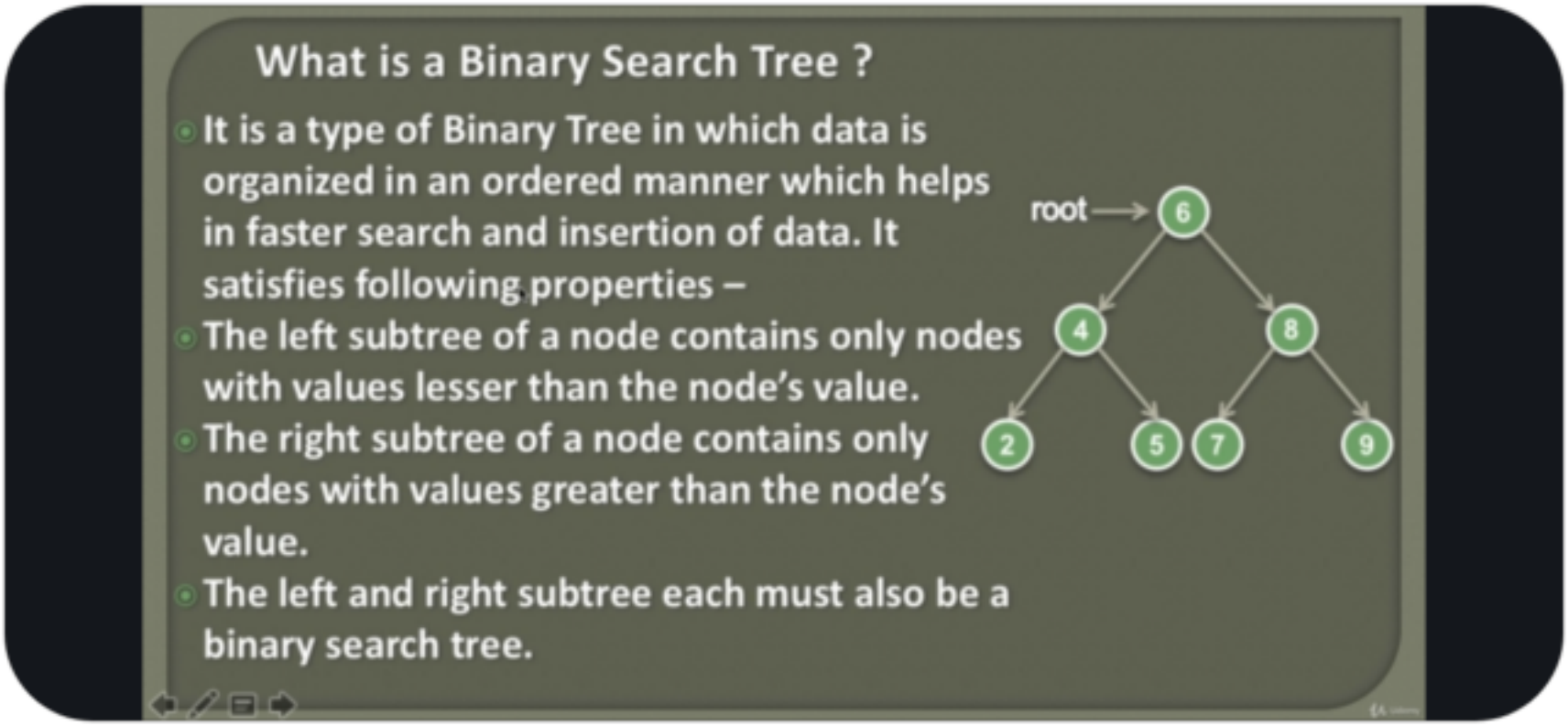
method call	line number	root
postOrder()	5	9
postOrder()	5	2
postOrder()	5	4
postOrder()		null

```
1  public void postOrder(TreeNode root) {
2      if(root == null) { // base case
3          return;
4      }
5      postOrder(root.left);
6      postOrder(root.right);
7      System.out.print(root.data + " ");
8  }
```

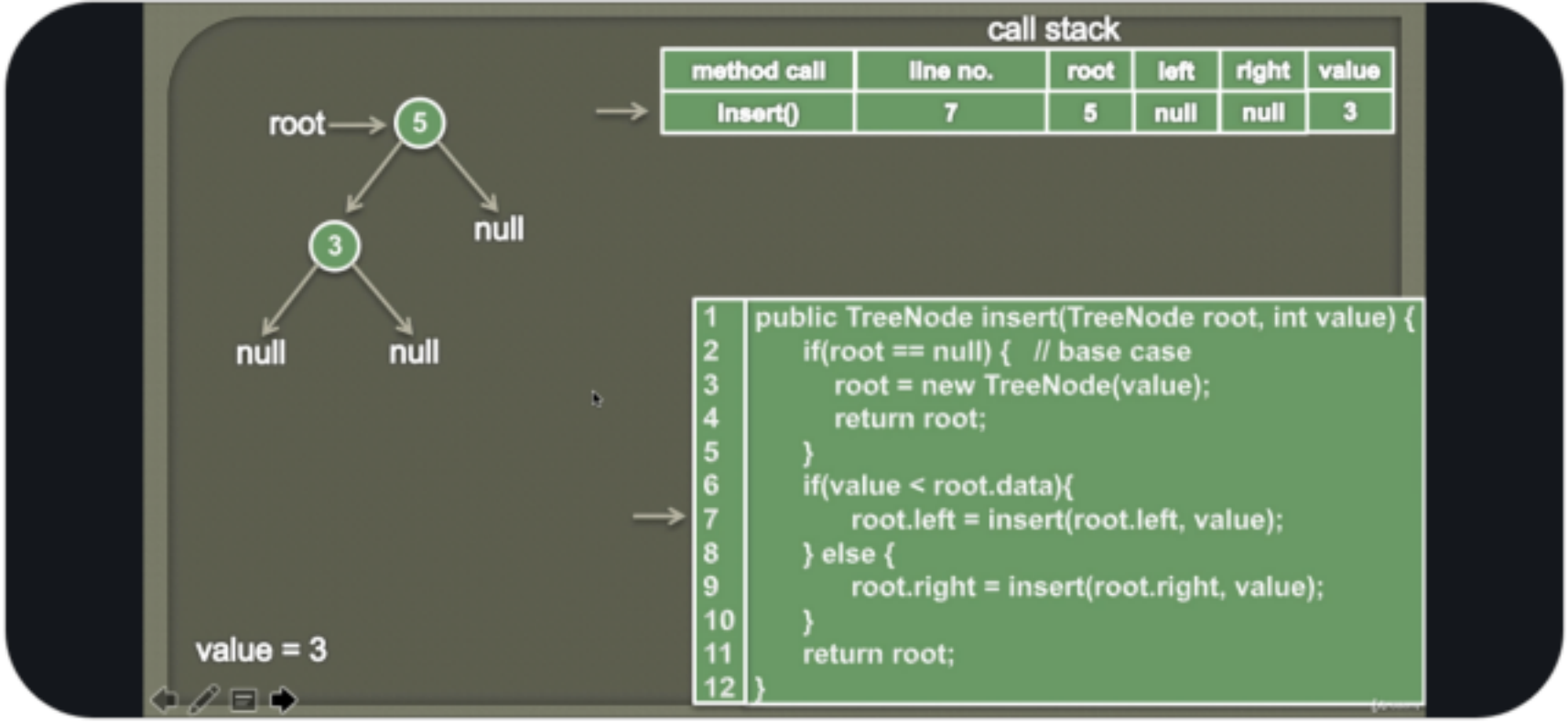
Level order traversal using queue



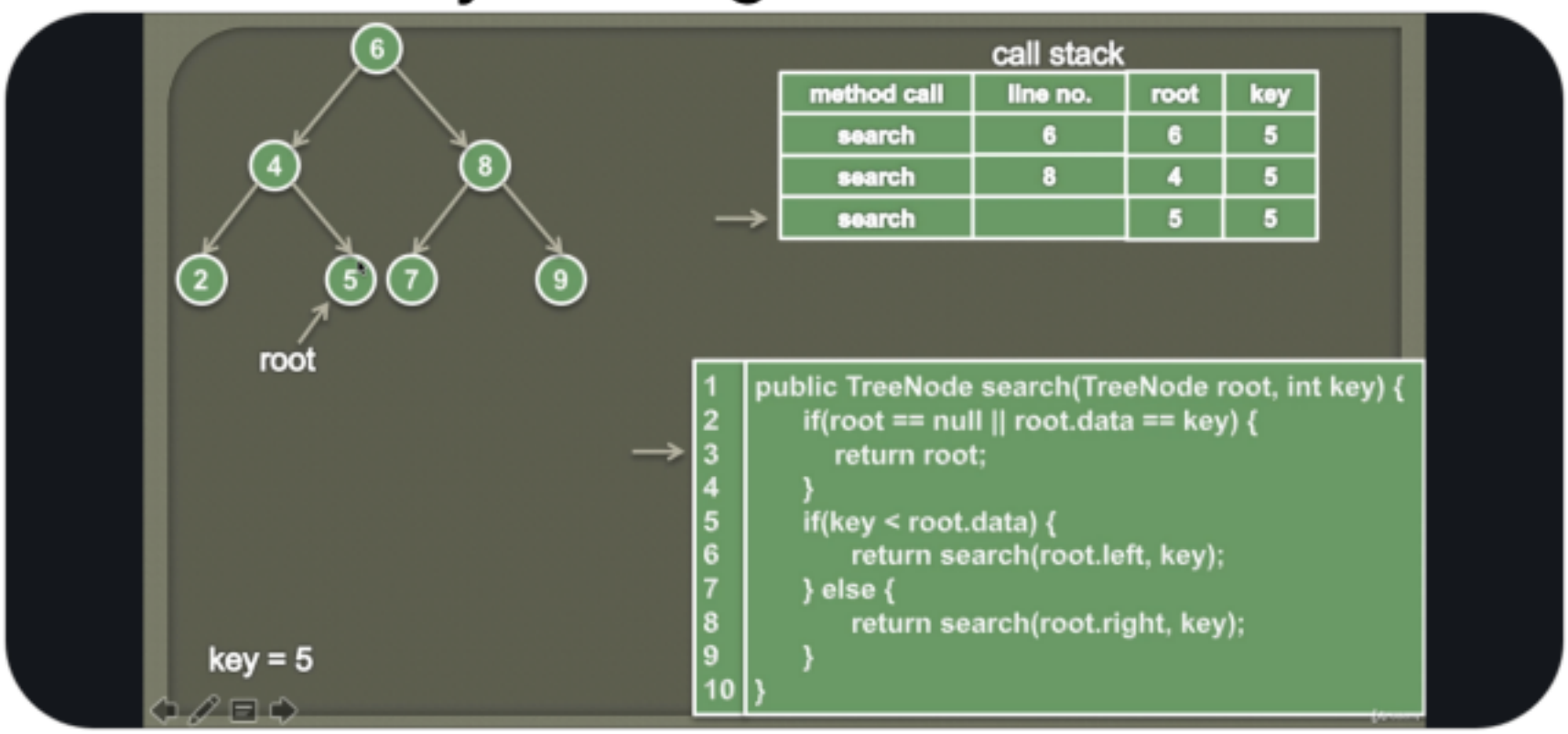
Binary Search Tree



Insert a value through recursion



Search a key through recursion



Graphs

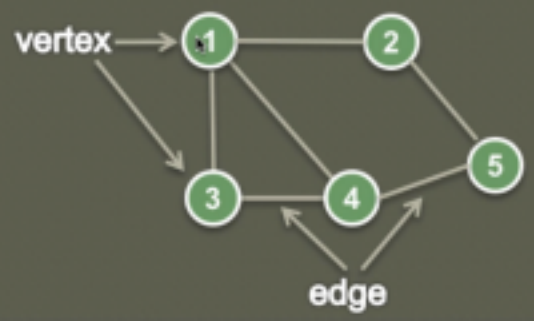
- A graph data structure mainly stores connected data, for example, a network of people or a

network of cities. A graph data structure typically consists of nodes or points called vertices. Each vertex is connected to another vertex using links called edges.

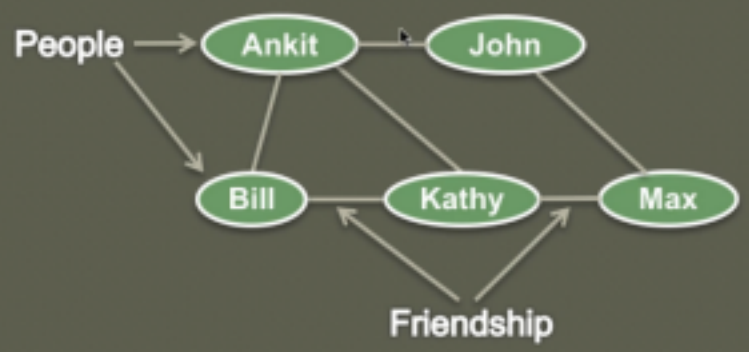
- ○ A vertex represents the entity (for example, people) and an edge represents the relationship between entities (for example, a person's friendships).
- ○ Directed graph: Graphs having edges with direction
- ○ Weighted graph: Graphs having edges with relative weight
- ○ Cyclic and Acyclic graph

What is a Graph?

- Its a non-linear data structure used for storing data
- It is a set of *vertices* and a collection of *edges* that connects a pair of vertices
- In below example 1, 2, 3, 4, 5 are the *Vertex* of Graph and each line connecting them is called as *Edge*

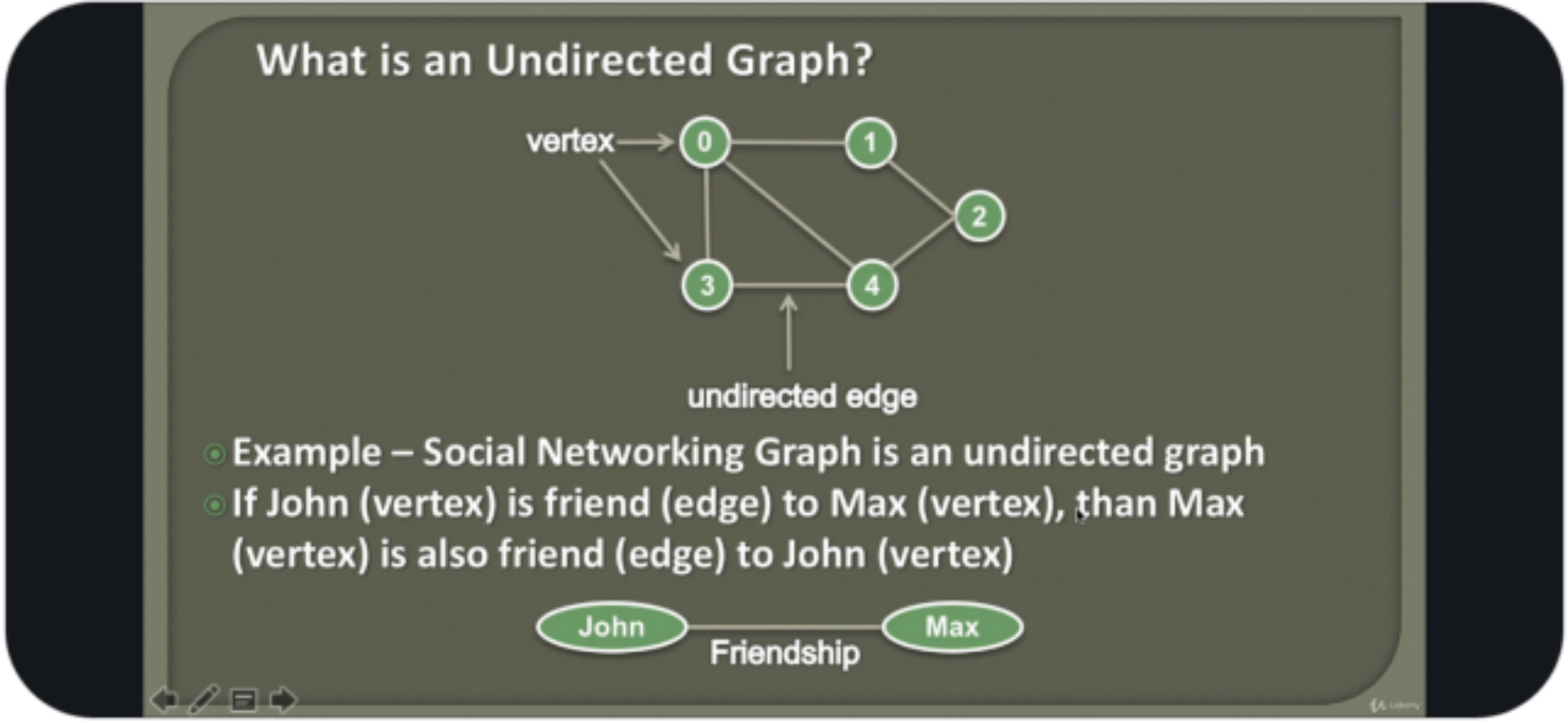


Applications – Social Network

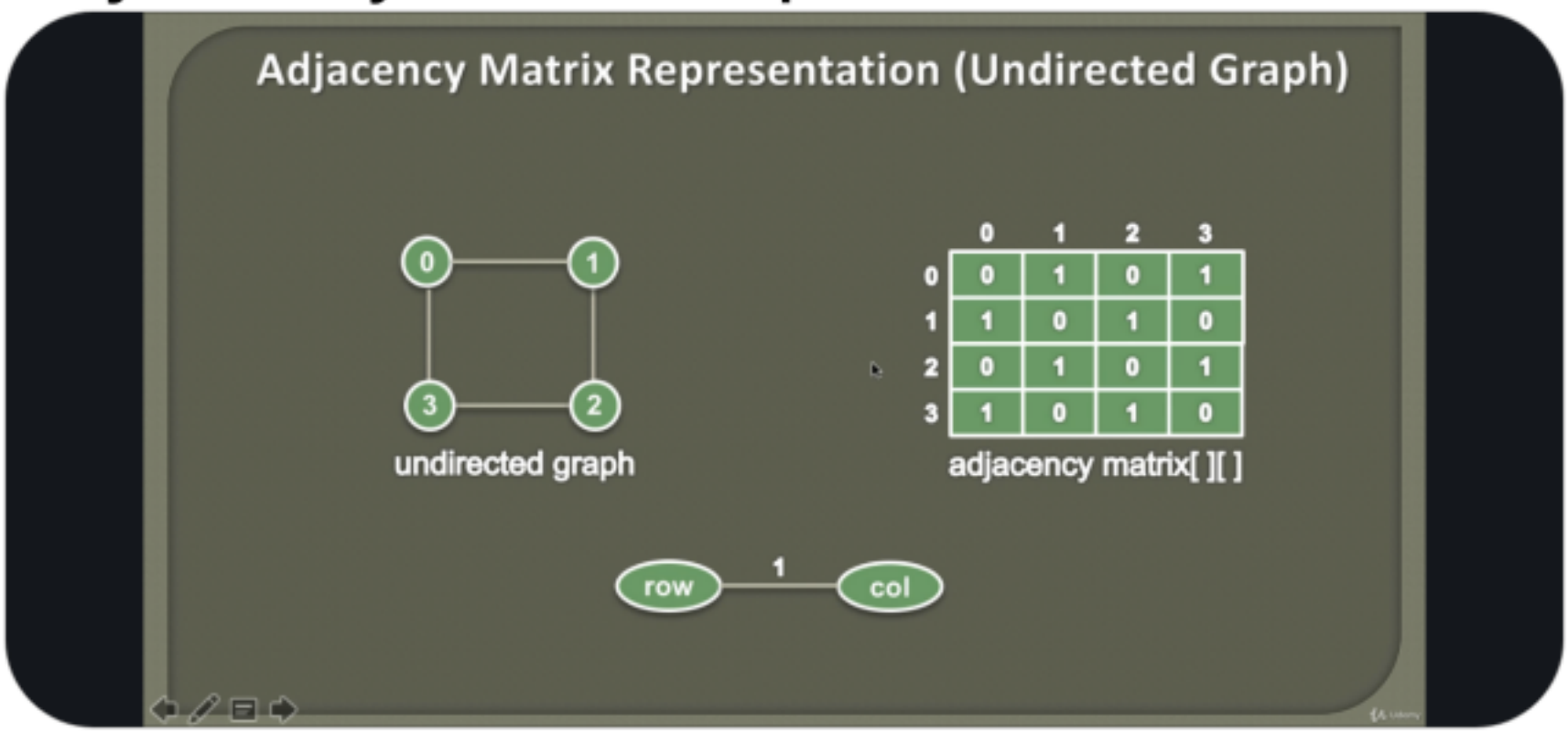


- Graphs help us implement Social Networking sites such as, Facebook, Twitter etc. It can be called as Social Networking graph
- Names of people represent vertices of Graph.
- Friendship between two people can be represented as an Edge of Graph

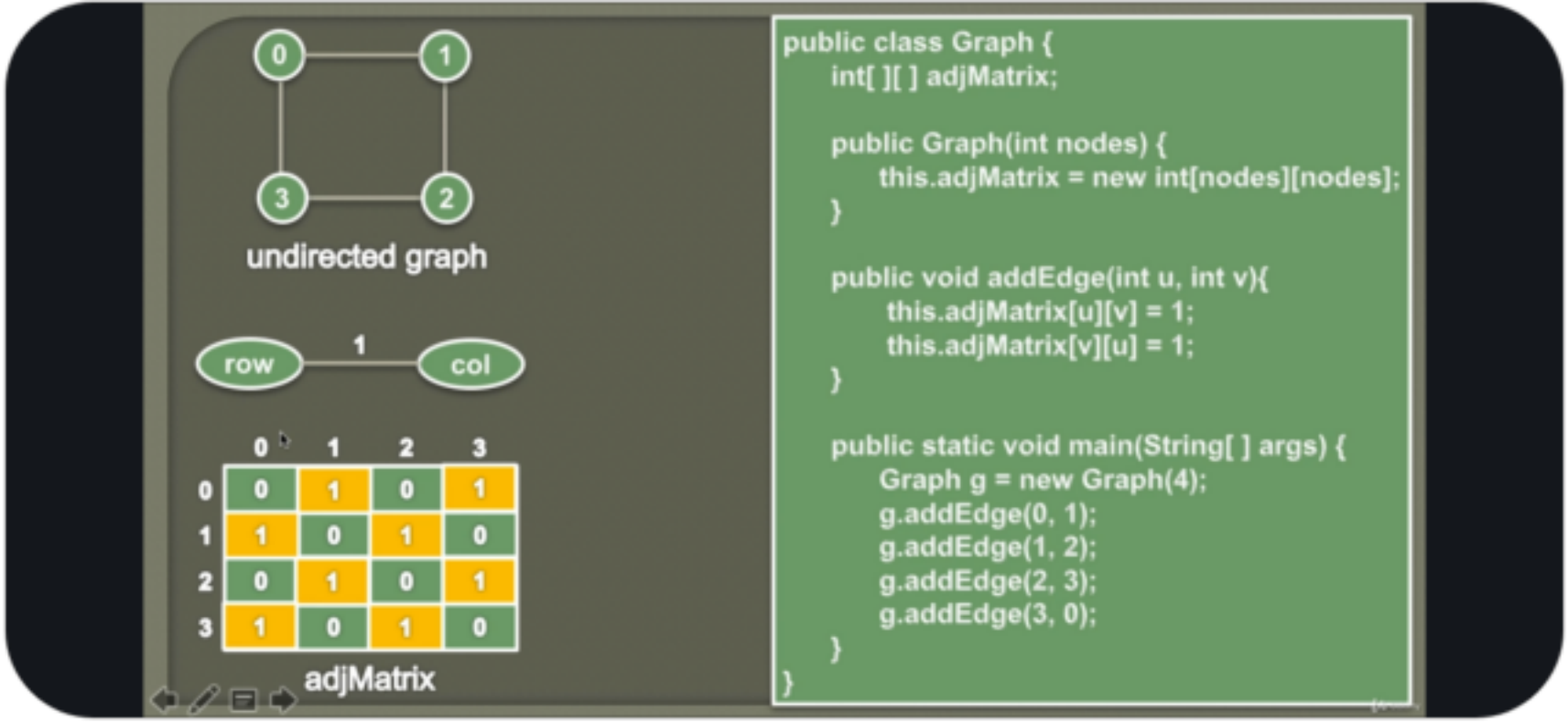
Undirected Graph



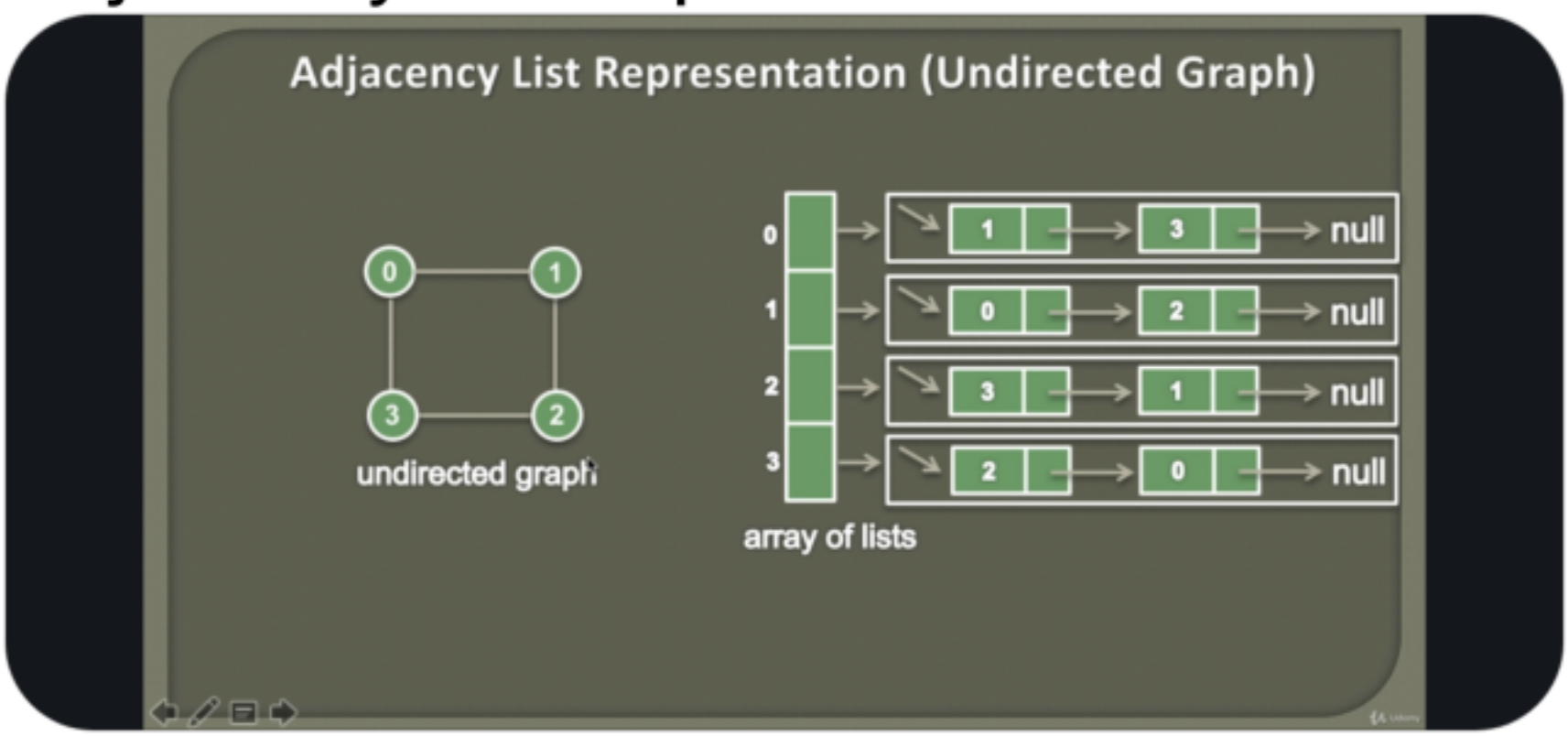
Adjacency Matrix representation



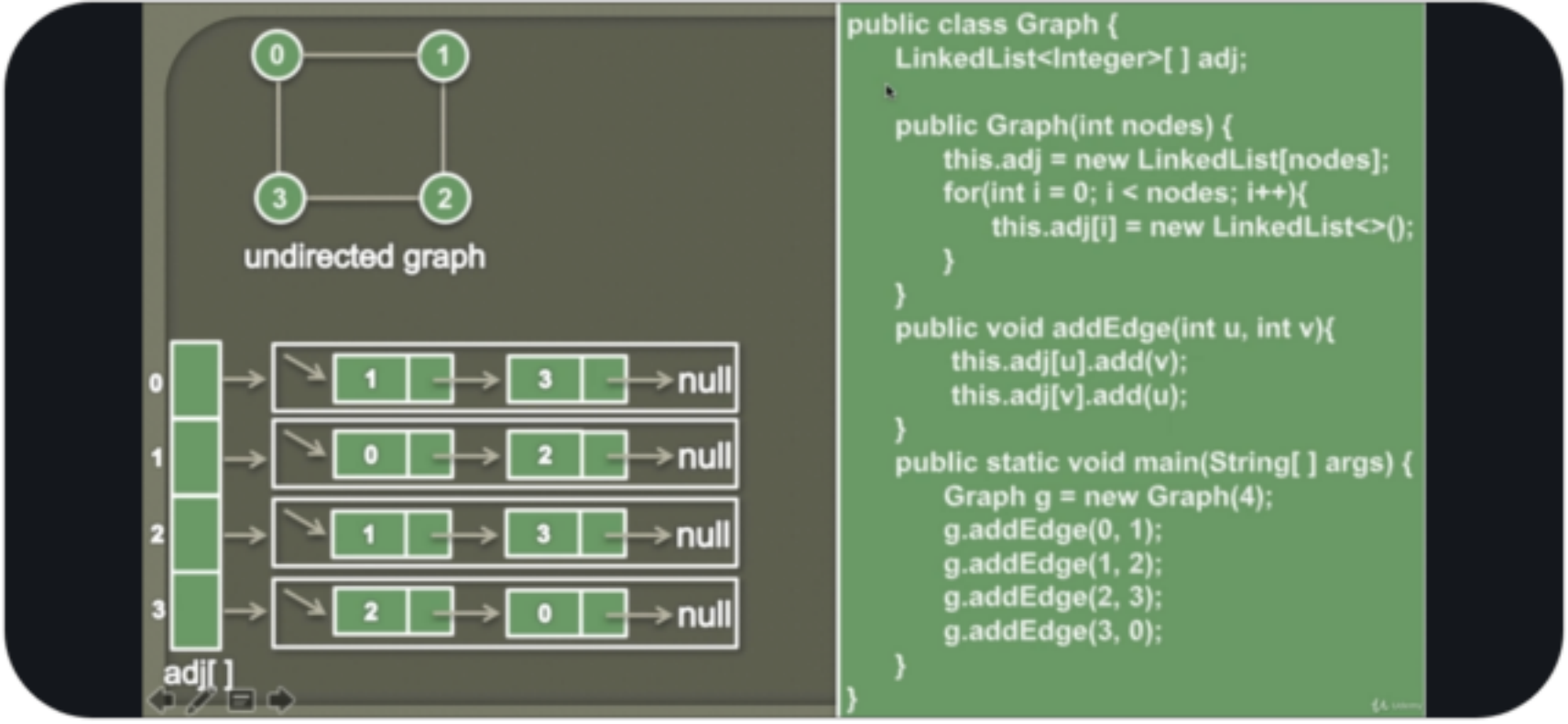
Adjacency Matrix implementation



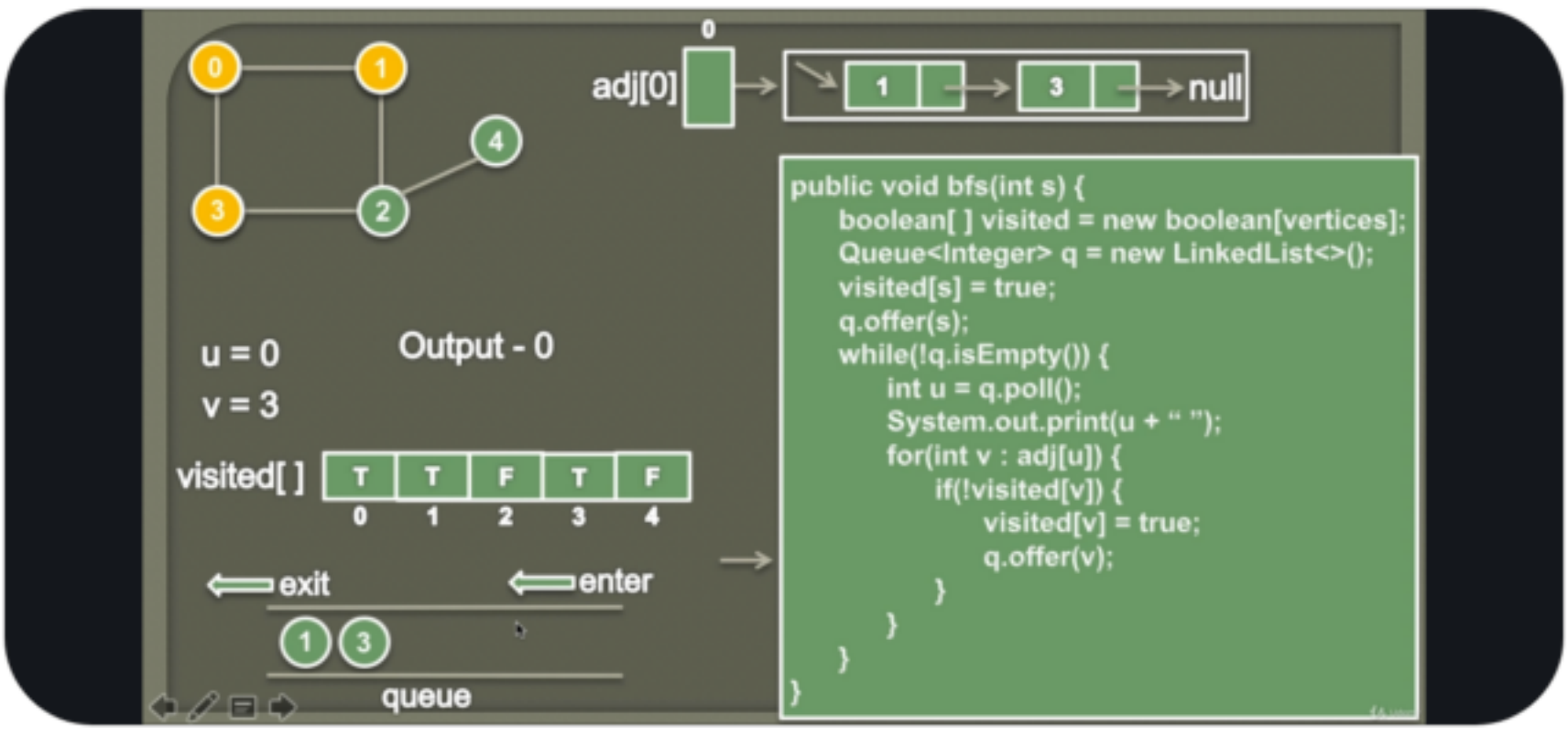
Adjacency List representation

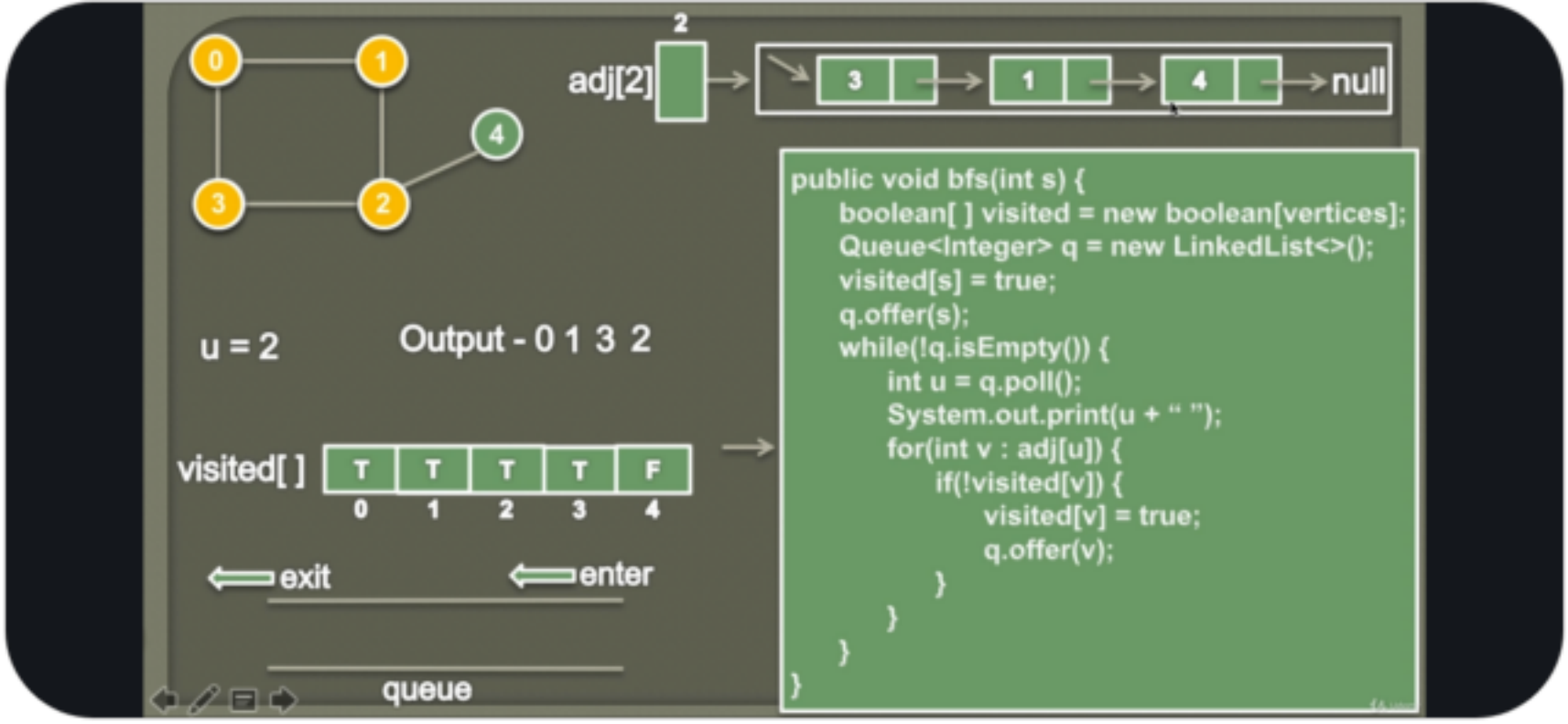


Adjacency List implementation

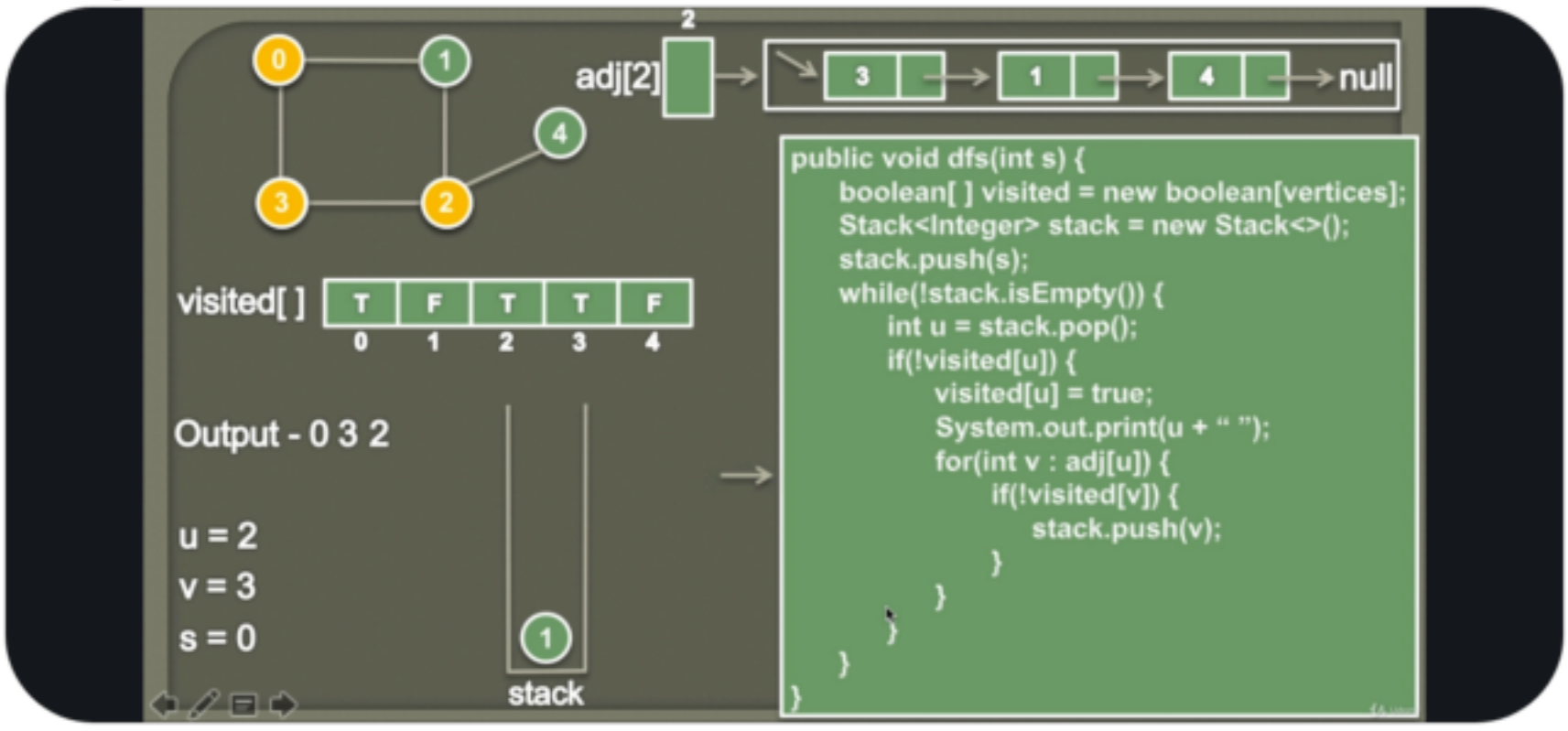


Breadth First Search





Depth First Search



Dynamic Programming

- It is mainly an optimization over recursion
- Dynamic Programming =

Recursion + Memorization

Jonathan Paulson amazingly explains What's Dynamic Programming on Quora as -

- Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
- "What's that equal to?"
- Counting - "Eight!"
- Writes down another "1+" on the left.
- "What about that?"
- "Nine!" - "How'd you know it was nine so fast?"
- "You just added one more!"
- "So you didn't need to recount because you remembered there were eight!
- Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic Programming

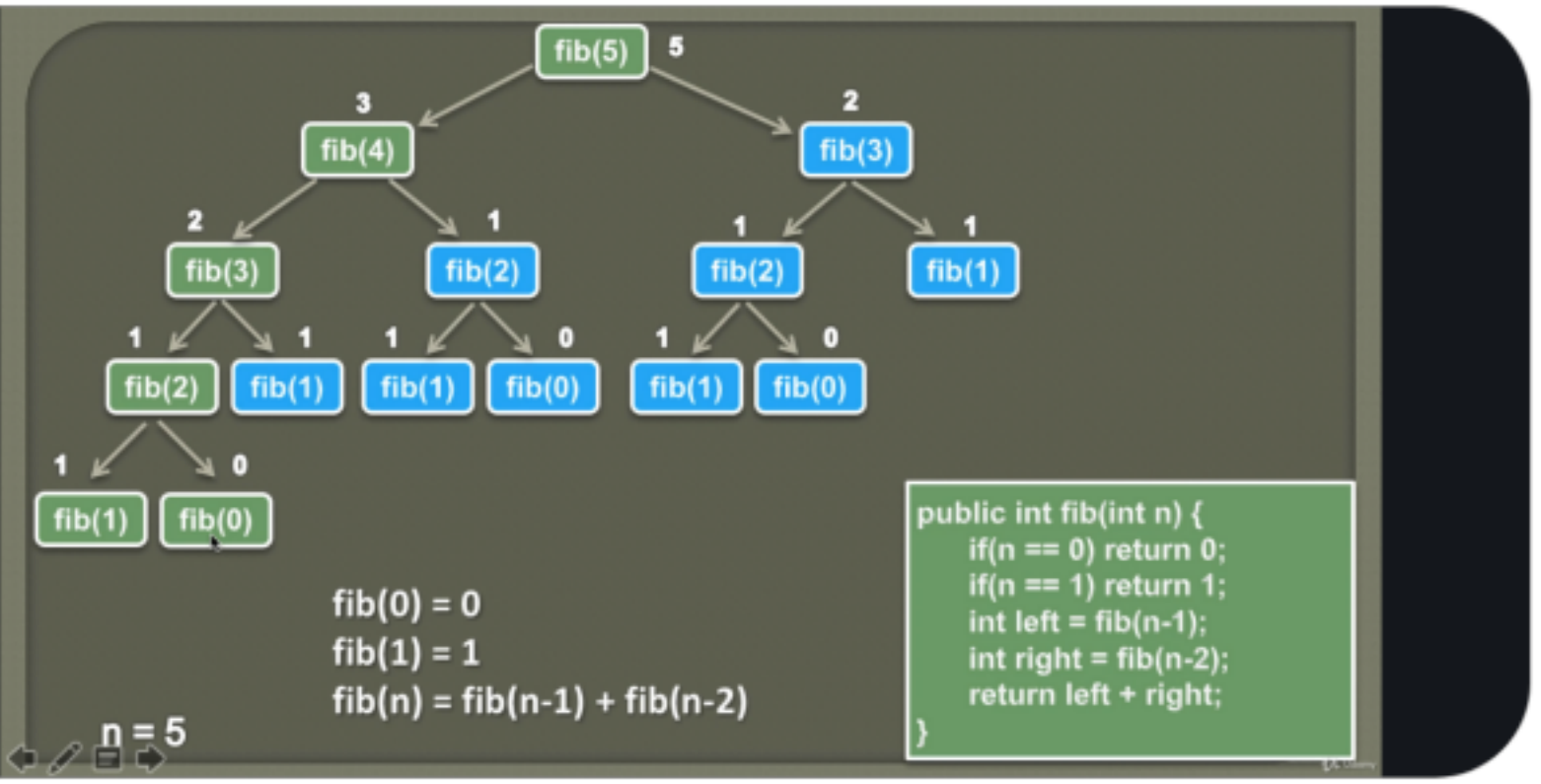
- It is a technique in which a complex problem is solved by –
  1. Breaking it into smaller sub-problems.
  2. Solving those sub-problems and simply storing their results.
  3. Re-use those stored results if sub-problems occurs/overlaps again. (Avoid solving sub-problem again)
  4. Finally using solutions to smaller problems build up solution to complex problem.

Fibonacci Series

- It is a series of numbers in which first two numbers are 0 and 1. After that each number is the sum of the two preceding numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21 ...

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

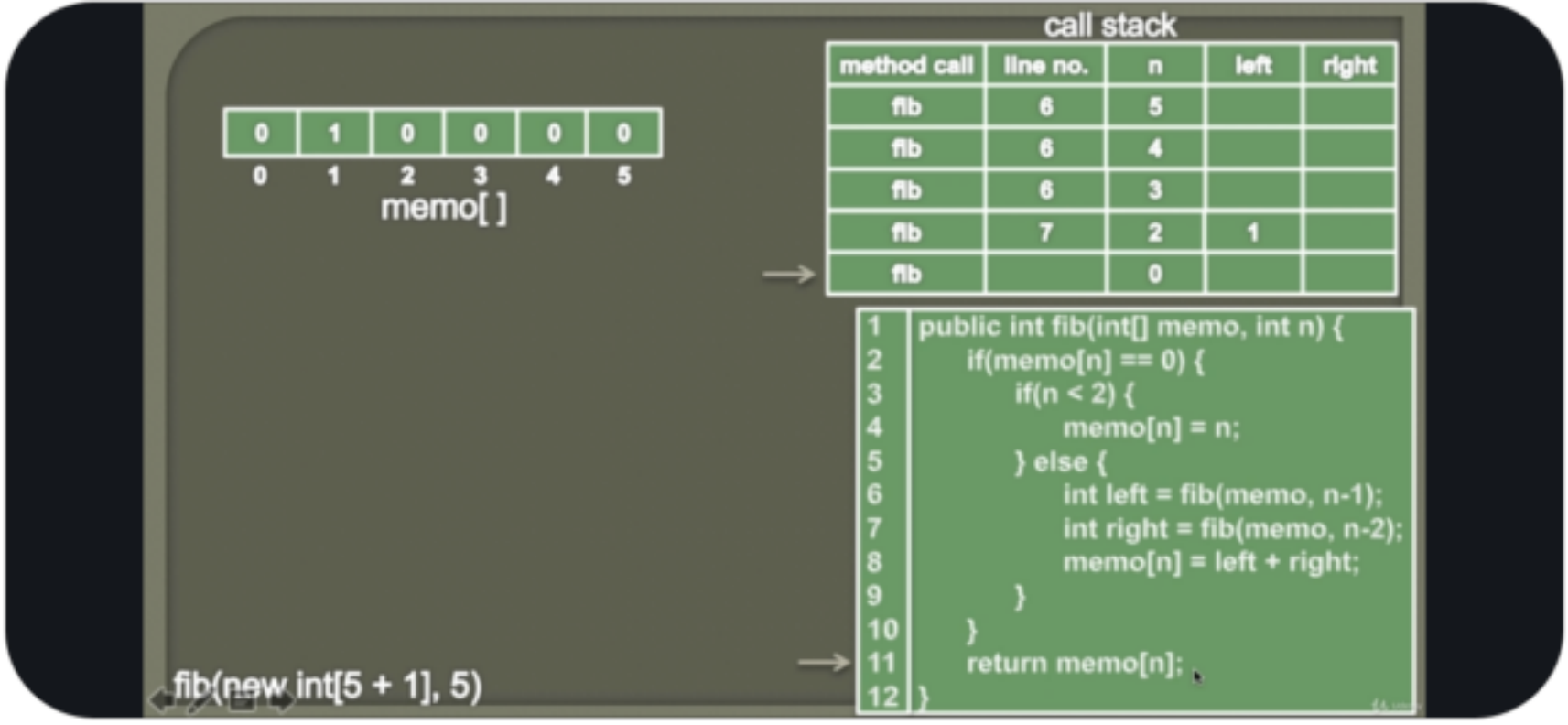


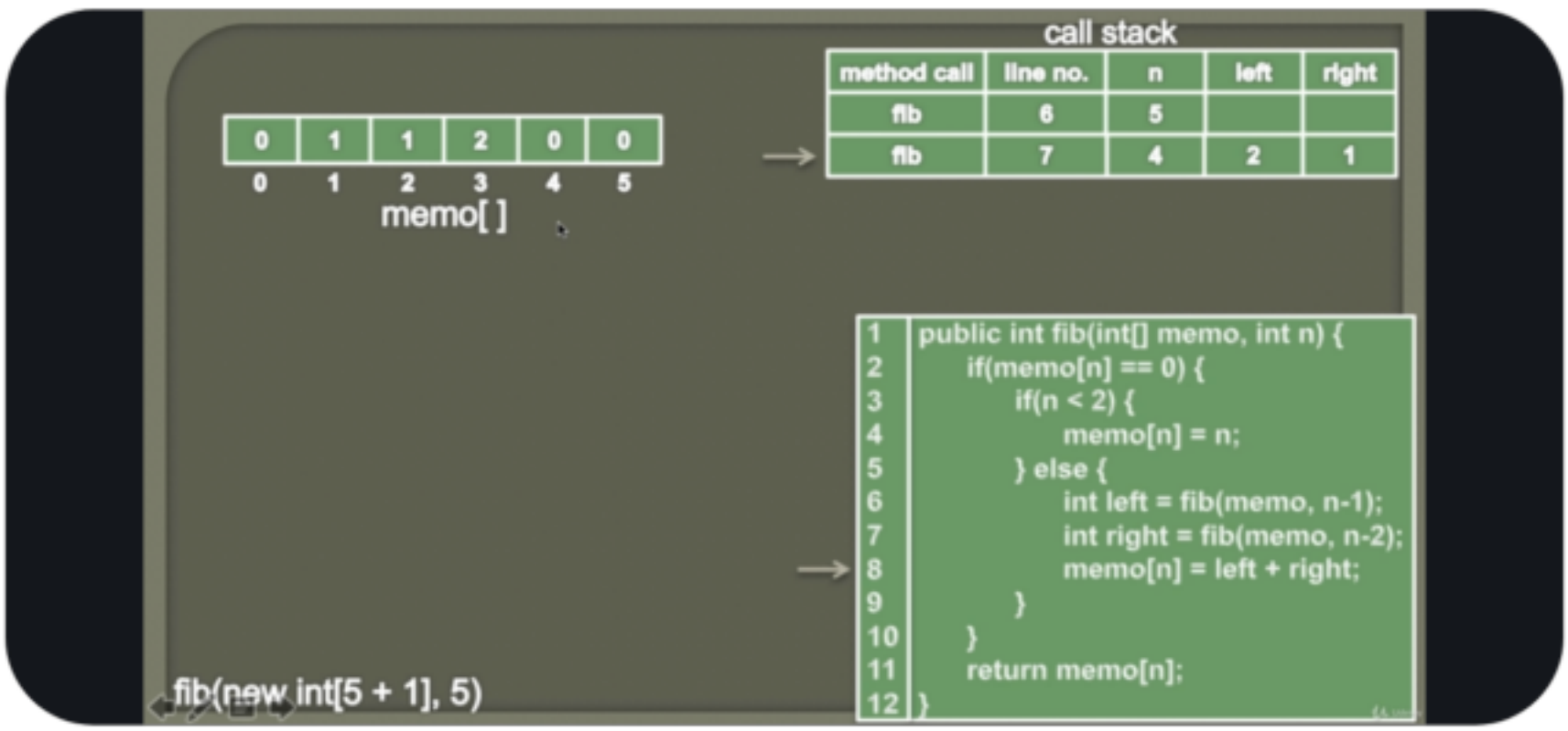
Bottom up approach

Bottom Up Approach

- We try to solve smaller sub-problems first, use their solution to build on and arrive at solutions to bigger sub-problems.
- It is also called as Tabulation method.
- The solution is build in a tabular form by using solutions of smaller sub-problems iteratively and generating solutions to bigger sub-problems.

Top down approach / Memoization





Java Utility

- Collections.sort(list) / Collections.sort(list, Collections.reverseOrder())

- ≡ ○ Collections.reverse(list)
- ≡ ○ Math.max(value1, value2)
- ≡ ○ Math.min(value1, value2)
- ≡ ○ Math.abs(intValue)
- ≡ ○ Arrays.sort(new int[] {4,2,1})
- ≡ ○ Arrays.asList(any object/value)
- ≡ ○ Integer.MAX_VALUE;
- ≡ ○ BigInteger factorial = BigInteger.ONE; factorial = factorial.multiply(BigInteger.valueOf(counter));
- ≡ ○ int[][] intervals = {{1,2}, {3,4}}
- ≡ ○ Arrays.sort(intervals, Comparator.comparingInt((int[] a) -> a[0]));

==========

Algorithm Patterns - reference educative.io - Grokking the coding interview

1. Sliding Window

- In many problems dealing with an array (or a LinkedList), we are asked to find or calculate something among all the contiguous sub arrays (or sublists) of a given size.
- For example, Given an array, find the average of all contiguous sub arrays of size ‘K’ in it
- Array: [1, 3, 2, 6, -1, 4, 1, 8, 2], K=5
- Output: [2.2, 2.8, 2.4, 3.6, 2.8]
- In some problems, the size of the sliding window is not fixed. We have to expand or shrink the window based on the problem constraints
- **Recheck:** Longest Substring with Same Letters after Replacement (hard), Problem Challenge 1, Problem Challenge 2, **Problem Challenge 3**, **Problem Challenge**

4

2. Two Pointers

- In problems where we deal with sorted arrays (or LinkedLists) and need to find a set of elements that fulfill certain constraints, the Two Pointers approach becomes quite useful. The set of elements could be a pair, a triplet or even a subarray
- For example, Given an array of sorted numbers and a target sum, find a pair in the array whose sum is equal to the given target.
- Given that the input array is sorted, an efficient way would be to start with one pointer in the beginning and another pointer at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do not, either we will

decrement the higher pointer if sum is greater than the target, else we will increment the lower pointer if sum is less than the target.

- **Recheck**: Squaring a Sorted Array (easy), Triplet Sum to Zero (medium),

3. Fast and Slow Pointers

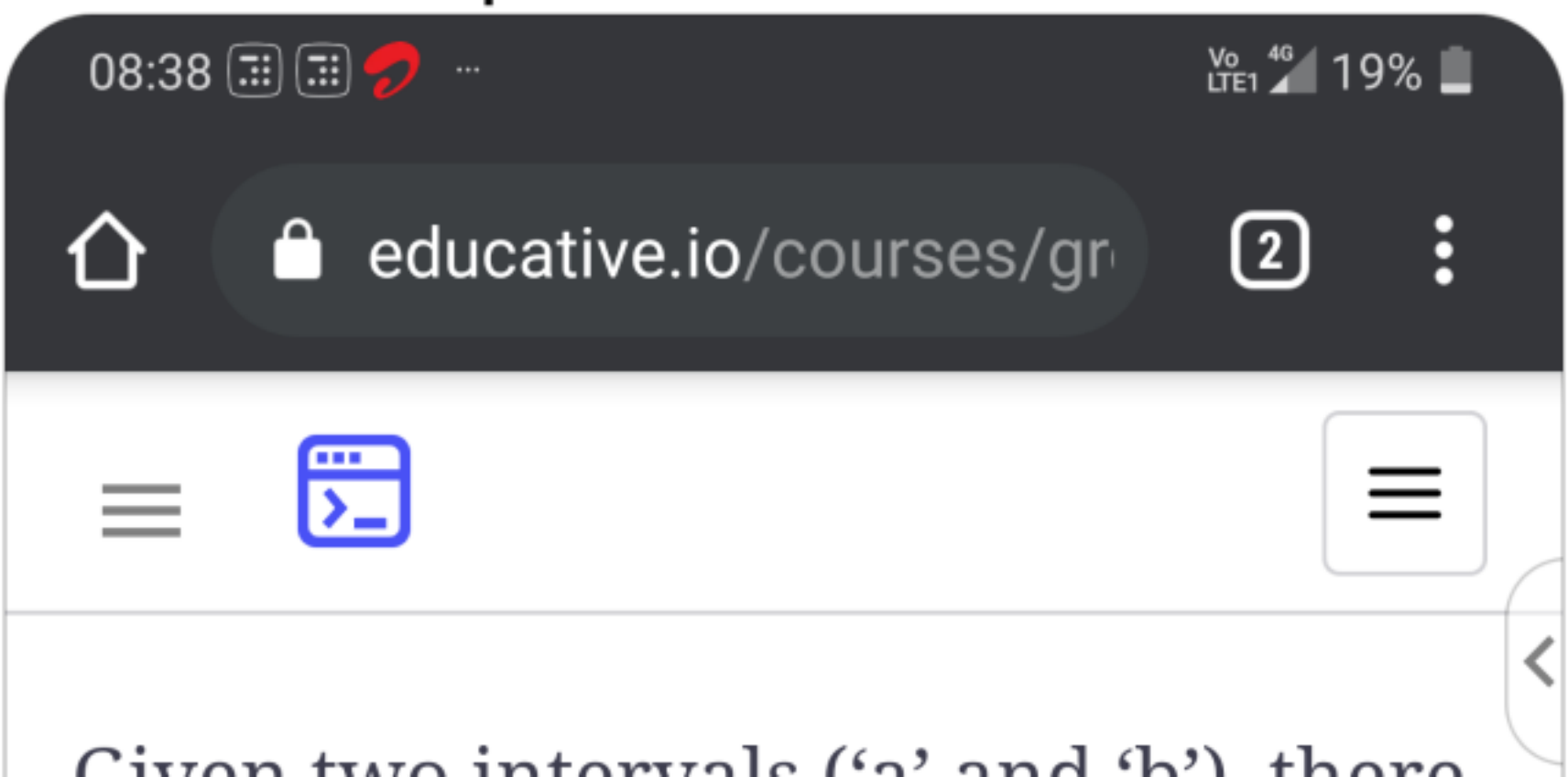
- It is a pointer algorithm that uses two pointers which move through the array (or sequence/ LinkedList) at different speeds. This approach is quite useful when dealing with cyclic LinkedLists or arrays
- By moving at different speeds (say, in a cyclic LinkedList), the algorithm proves that the two pointers are bound to meet. The fast pointer should catch the

slow pointer once both the pointers are in a cyclic loop

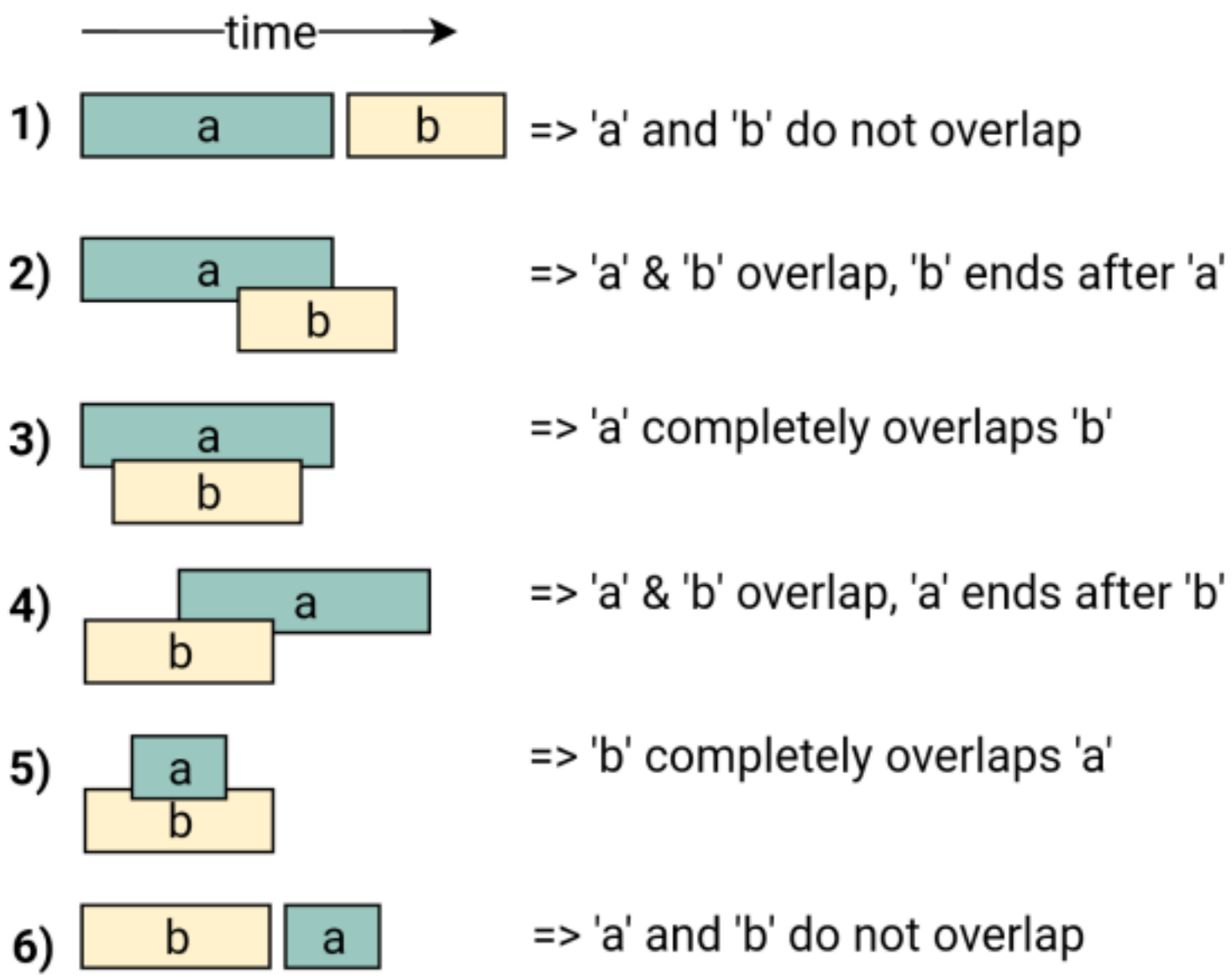
- One of the famous problems solved using this technique was Finding a cycle in a LinkedList.
- **Recheck:** Problem Challenge 1, Problem Challenge 2,

4. Merge Intervals

- This pattern describes an efficient technique to deal with overlapping intervals. In a lot of problems involving intervals, we either need to find overlapping intervals or merge intervals if they overlap.



Given two intervals ('a' and 'b'), there will be six different ways the two intervals can relate to each other:



Understanding the above six cases will help us in solving all intervals related problems. Let's jump onto our first problem to understand the **Merge Interval** pattern.

- **Recheck**: Merge Intervals (medium), Insert Interval (medium), Intervals Intersection (medium), **Problem Challenge 1**, **Problem Challenge 2**, **Problem Challenge 3**

5. Top 'K' Elements

- Any problem that asks us to find the top/smallest/frequent ‘K’ elements among a given set falls under this pattern.
- The best data structure that comes to mind to keep track of ‘K’ elements is Heap (Max or Min Heap)
- One of the implementation of Heap data structure in java is Priority Queue.
- **Recheck:** Top 'k' numbers,

connect ropes, Top 'K' Frequent Numbers (medium), Kth Largest Number in a Stream, **'K' Closest Numbers,** Sum of Elements, **Problem Challenge 2**, **Problem Challenge 3**

6. Cyclic Sort

- For sorting 0 to n or 1 to n numbers
- To identify missing single or multiple numbers
- To identify duplicate single or multiple numbers

Introduction

This pattern describes an interesting approach to deal with problems involving arrays containing numbers in a given range. For example, take the following problem:

> You are given an unsorted array containing numbers taken from the range 1 to 'n'. The array can have duplicates, which means that some numbers will be missing. Find all the missing numbers.

To efficiently solve this problem, we can use the fact that the input array contains numbers in the range of 1 to 'n'. For example, to efficiently sort the array, we can try placing each number in its correct place, i.e., placing '1' at index '0', placing '2' at index '1', and so on. Once we are done with the sorting, we can iterate the array to find all indices that are missing the correct numbers. These will be our required numbers.

Let's jump on to our first problem to understand the **Cyclic Sort** pattern in detail.

- For 1 to n numbers the check for index position will be if(numbs[i] != numbs[numbs[i]- 1]) { swap numbers }
- For 0 to n numbers the check for index position will be if if(numbs[i] != numbs[numbs[i]) { swap numbers }
- **Recheck:** Find the duplicate number(easy), Problem Challenge 3

7. In-place Reversal of a Linked List

- In a lot of problems, we are asked to reverse the links between a set of nodes of a LinkedList. Often, the constraint is that we need to do this in-place, i.e., using the existing node objects and without using extra memory.
- For example reverse a linked list, reverse a sublist, reverse every k element sublist etc
- **Recheck:** Reverse a sublist(medium), Reverse every k element sublist, Problem Challenge 1, Problem Challenge 2

8. Two Heaps

- In many problems, where we are given a set of elements such that we can divide them into two parts. To solve the problem, we are interested in knowing the smallest element in one part and

the biggest element in the other part. This pattern is an efficient approach to solve such problems.

- This pattern uses two Heaps to solve these problems; A Min Heap to find the smallest element and a Max Heap to find the biggest element.
- **Recheck:** Sliding window median (hard),

9. Tree Breadth First Search

- This approach is based on the Breadth First Search (BFS) technique to traverse a tree
- Any problem involving the traversal of a tree in a level-by-level order can be efficiently solved using this approach.
- We will use a Queue to keep track of all the nodes of a level before we jump onto the next level. This

also means that the space complexity of the algorithm will be O(W)O(W), where ‘W’ is the maximum number of nodes on any level.

- **Recheck:** Binary tree level order traversal, Zigzag Traversal (medium), Connect **Level Order Siblings** (medium), Problem challenge 1, problem challenge 2

10. Tree Depth First Search

- This pattern is based on the Depth First Search (DFS) technique to traverse a tree.
- We will be using recursion (or we can also use a stack for the iterative approach) to keep track of all the previous (parent) nodes while traversing. This also means that the space complexity of the algorithm will be O(H)O(H), where

‘H’ is the maximum height of the tree

- **Recheck:** All Paths for a Sum (medium), **Path With Given Sequence (medium), Count Paths for a Sum (medium), Problem challenge 1, problem challenge 2**

11. Bitwise XoR

- XOR is a logical bitwise operator that returns 0 (false) if both bits are the same and returns 1 (true) otherwise. In other words, it only returns 1 if exactly one bit is set to 1 out of the two bits in comparison.
- For example, Given an array of n−1 integers in the range from 1 to n, find the one number that is missing from the array.
- Input: 1, 5, 2, 6, 4
  Answer: 3

- **Recheck:** Introduction, Single Number (easy), problem challenge 1

12. K-Way Merge

- This pattern helps us solve problems that involve a list of sorted arrays.
- Whenever we are given ‘K’ sorted arrays, we can use a Heap to efficiently perform a sorted traversal of all the elements of all arrays.
- We can push the smallest (first) element of each sorted array in a Min Heap to get the overall minimum. While inserting elements to the Min Heap we keep track of which array the element came from.
- We can, then, remove the top element from the heap to get the smallest element and push the

next element from the same array, to which this smallest element belonged, to the heap.

- ≡ ○ We can repeat this process to make a sorted traversal of all elements.
- ≡ ○ **Recheck:** Merge K Sorted Lists (medium), Kth Smallest Number in M Sorted Lists (Medium), **Kth Smallest Number in a Sorted Matrix (Hard)**